Contents

ASP.NET Core 文档

概述

- 关于 ASP.NET Core
- 比较 ASP.NET Core 和 ASP.NET
- 比较 .NET Core 和 .NET Framework

入门

新增功能

- 2.2 中的新变化
- 2.1 版中的新增功能
- 2.0 版中的新增功能
- 1.1 中的新增功能

教程

- Web API 应用
  - 创建 Web API
  - 使用 MongoDB 创建 Web API
- Web 应用
  - Razor 页面
  - MVC
- 实时 web 应用
  - JavaScript 和 SignalR
  - TypeScript 和 SignalR
- 为本机移动应用创建后端服务
- 数据访问
  - EF Core 和 Razor Pages
  - EF Core 和 MVC - 现有 DB
  - EF Core 与 MVC - 新 DB
  - EF Core 和 MVC - 长篇教程

基础知识

- 概述

应用启动
依赖关系注入(服务)
路由
环境(开发、分阶段、生产)
配置
选项
日志记录
处理错误
中间件
概述
基于工厂的中间件
Factory-based middleware with third-party container(第三方容器中基于工厂的中间件)
Host
概述
Web 主机
通用主机
服务器
启动 HTTP 请求
Web 应用
Razor 页面
概述
Razor Pages 教程
入门
添加模型
基架
使用 DB
更新页面
添加搜索
添加新字段
添加验证
MVC
MVC 概述

MVC 教程
入门
添加控制器
添加视图
添加模型
使用 DB
控制器操作和视图
添加搜索
添加新字段
添加验证
检查 Details 和 Delete 方法
筛选器
Razor 类库
路由和应用约定
上载文件
Razor SDK
视图
部分视图
控制器
路由
文件上传
依赖关系注入 - 控制器
依赖关系注入 - 视图
单元测试
会话和应用状态
标记帮助程序
概述
创建标记帮助程序
在窗体中使用标记帮助程序
标记帮助程序组件
内置标记帮助程序
定位点

缓存
分布式缓存
环境
窗体
Image
输入
标签
Partial
选择
Textarea
验证消息
验证摘要
布局
静态文件
模型绑定
模型验证
Razor 语法
高级
视图组件
视图编译
应用模型
筛选器
区域
应用部件
自定义模型绑定
兼容性版本
Web API 应用
概述
教程
创建 Web API
使用 MongoDB 创建 Web API
Swagger/OpenAPI

- 概述
- Swashbuckle 入门
- NSwag 入门
- 操作返回类型
- 格式化响应数据
- 自定义格式化程序
- 分析器
- 约定
- 实时应用
- SignalR 概述
- 支持的平台
- 教程
  - JavaScript 和 SignalR
  - TypeScript 和 SignalR
- 示例
- 服务器概念
  - 中心
  - HubContext
  - 用户和组
  - 发布到 Azure
  - API 设计注意事项
- 客户端
  - .NET 客户端
  - .NET API 参考
  - Java 客户端
  - Java API 参考
  - JavaScript 客户端
  - JavaScript API 参考
- 托管和缩放
  - 概述
  - Azure SignalR 服务
  - Redis 底板

配置
身份验证和授权
安全注意事项
MessagePack 中心协议
流式处理
比较 SignalR 和 SignalR Core
不使用 SignalR 的 WebSockets
测试、调试和疑难解答
单元测试
Razor 页面单元测试
测试控制器
远程调试
快照调试
Visual Studio 中的快照调试
集成测试
负载测试和压力测试
疑难解答
日志记录
数据访问
教程
EF Core 和 Razor Pages
概述
入门
创建、读取、更新和删除
排序、筛选器、页和组
迁移
创建复杂数据模型
读取相关数据
更新相关数据
处理并发冲突
EF Core 与 MVC - 新 DB
EF Core 和 MVC - 现有 DB

EF Core 和 MVC - 长篇教程
概述
入门
创建、读取、更新和删除
排序、筛选器、页和组
迁移
创建复杂数据模型
读取相关数据
更新相关数据
处理并发冲突
继承
高级主题
EF 6 和 ASP.NET Core
使用 Visual Studio 的 Azure 存储
连接的服务
Blob 存储
队列存储
表存储
客户端开发
概述
Gulp
Grunt
LibMan
概述
CLI
Visual Studio
Bower
LESS、Sass 和 Font Awesome
捆绑和缩小
浏览器链接
单页应用
概述

- Angular
- React
- React 和 Redux
- JavaScriptServices

托管和部署

- 概述
- 在 Azure 应用服务上托管
  - 概述
  - 使用 Visual Studio 进行发布
  - 使用 CLI 工具进行发布
  - 使用 Visual Studio 和 Git 进行发布
  - 使用 Azure Pipelines 连续部署
  - 解决启动错误
- DevOps
  - 概述
  - 工具和下载
  - 部署到应用服务
  - 持续集成和部署
  - 监视和故障排除
  - 后续步骤
- 使用 IIS 在 Windows 上进行托管
  - 概述
  - IIS 疑难解答
  - ASP.NET Core 模块
  - Visual Studio 中的 IIS 支持
  - IIS 模块
- Kestrel
- HTTP.sys
- 在 Windows 服务中进行托管
- 在 Linux 上使用 Nginx 进行托管
- 在 Linux 上使用 Apache 进行托管
- 在 Docker 中托管

概述
生成 Docker 映像
Visual Studio Tools
发布到 Docker 映像
代理和负载均衡器配置
在 Web 场中托管
Visual Studio 发布配置文件
目录结构
Azure 应用服务和 IIS 的错误参考
运行状况检查
安全和标识
概述
身份验证
标识简介
基架标识
将自定义用户数据添加到标识
自定义标识
社区 OSS 身份验证选项
配置标识
配置 Windows 身份验证
自动以标识的存储提供程序
外部提供程序
概述
Facebook 身份验证
Twitter 身份验证
Google 身份验证
Microsoft 身份验证
外部身份验证提供程序
其他声明
WS 联合身份验证
帐户确认和密码恢复
在标识中启用 QR 代码生成

使用 SMS 设置双因素身份验证
在没有标识的情况下使用 cookie 身份验证
Azure Active Directory
概述
将 Azure AD 集成到 Web 应用中
将 Azure AD B2C 集成到 Web 应用中
将 Azure AD B2C 集成到 Web API 中
从 WPF 调用 Web API
使用 Azure AD 在 Web 应用中调用 Web API
使用 IdentityServer4 保护 ASP.NET Core 应用
使用 Azure App Service 身份验证保护 ASP.NET Core 应用(简易身份验证)
各个用户帐户
授权
概述
通过身份验证创建 Web 应用
Razor 页面授权约定
简单授权
基于角色的授权
基于声明的授权
基于策略的授权
授权策略提供程序
要求处理程序中的依赖关系注入
基于资源的授权
基于视图的授权
使用方案限制标识
数据保护
概述
数据保护 API
使用者 API
概述
目标字符串
目标层次结构和多租户

哈希密码
限制受保护负载的生存期
取消保护已撤消其密钥的负载
配置
概述
配置数据保护
默认设置
计算机范围的策略
非 DI 感知方案
扩展性 API
概述
核心加密扩展性
密钥管理扩展性
其他 API
实现
概述
已验证的加密详细信息
子项派生和已验证的加密
上下文标头
密钥管理
密钥存储提供程序
静态密钥加密
密钥永久性和设置
密钥存储格式
短数据保护提供程序
兼容性
概述
在 ASP.NET 中替换 <machineKey>
在开发过程中保护机密
Enforce HTTPS
欧盟一般数据保护条例 (GDPR) 支持
Azure Key Vault 配置提供程序

反请求伪造
阻止打开重定向攻击
阻止跨站点脚本编写
启用跨域请求 (CORS)
在应用之间共享 Cookie
IP 安全列表
性能
概述
响应缓存
概述
内存中缓存
分布式缓存
响应缓存中间件
响应压缩
诊断工具
负载测试和压力测试
其他主题
全球化和本地化
使用 Orchard Core 的可移植对象本地化
URL 重写
文件提供程序
请求功能
访问 HttpContext
更改令牌
.NET 的开放 Web 接口 (OWIN)
使用托管服务的后台任务
承载启动程序集
Microsoft.AspNetCore.App 元包
Microsoft.AspNetCore.All metapackage
日志记录与 LoggerMessage
使用文件观察程序
迁移
2.2 至 3.0

2.1 至 2.2
2.0 至 2.1
1.x 至 2.0
概述
身份验证和标识
ASP.NET 到 ASP.NET Core
概述
MVC
Web API
配置
身份验证和标识
ClaimsPrincipal.Current
共成员身份到标识
HTTP 模块到中间件
日志记录（不是 ASP.NET Core）
API 参考
参与

ASP.NET Core 简介

2018/12/14 • • Edit Online

作者：Daniel RothRick Anderson 和 Shaun Luttin

ASP.NET Core 是一个跨平台的高性能开源框架，用于生成基于云且连接 Internet 的新式应用程序。使用 ASP.NET Core，您可以：

- 建置 Web 应用程序和服务、IoT 应用和移动后端。
- 在 Windows、macOS 和 Linux 上使用喜爱的开发工具。
- 部署到云或本地。
- 在 .NET Core 或 .NET Framework 上运行。

为何使用 ASP.NET Core？

数百万开发人员使用过（并将继续使用）ASP.NET 4.x 创建 Web 应用。ASP.NET Core 是重新设计的 ASP.NET 4.x，更改了体系结构，形成了更精简的模块化框架。

ASP.NET Core 具有如下优点：

- 生成 Web UI 和 Web API 的统一场景。
- 针对可测试性进行构建。
- Razor Pages 可以使基于页面的编码方式更简单高效。
- 能够在 Windows、macOS 和 Linux 上进行开发和运行。
- 开放源代码和以社区为中心。
- 集成新式客户端框架和开发工作流。
- 基于环境的云就绪配置系统。
- 内置依赖项注入。
- 轻型的高性能模块化 HTTP 请求管道。
- 能够在 IIS、Nginx、Apache、Docker 上进行托管或在自己的进程中进行自托管。
- 定目标到 .NET Core 时，可以使用并行应用版本控制。
- 简化新式 Web 开发的工具。

使用 ASP.NET Core MVC 生成 Web API 和 Web UI

ASP.NET Core MVC 提供生成 Web API 和 Web 应用所需的功能：

- Model-View-Controller (MVC) 模式 使 Web API 和 Web 应用可测试。
- Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。
- Razor 标记提供了适用于 Razor 页面和 MVC 视图的高效语法。
- 标记帮助程序使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。
- 内置的多数据格式和内容协商支持使 Web API 可访问多种客户端，包括浏览器和移动设备。
- 模型绑定自动将 HTTP 请求中的数据映射到操作方法参数。
- 模型验证自动执行客户端和服务器端验证。

客户端开发

ASP.NET Core 与常用客户端框架和库（包括 Angular、React 和 Bootstrap）无缝集成。有关详细信息，请参阅客户端开发。

面向 .NET Framework 的 ASP.NET Core

ASP.NET Core 2.x 可以面向 .NET Core 或 .NET Framework。面向 .NET Framework 的 ASP.NET Core 应用无法跨平台，它们仅在 Windows 上运行。通常，ASP.NET Core 2.x 由 .NET Standard 库组成。使用 .NET Standard 2.0 编写的应用可在 NET Standard 2.0 支持的任何位置运行。

与 .NET Standard 2.0 兼容的 .NET Framework 版本支持 ASP.NET Core 2.x：

- 强烈建议使用 .NET Framework 4.7.1 及更高版本。
- .NET Framework 4.6.1 及更高版本。

ASP.NET Core 3.0 以及更高版本只能在 .NET Core 中运行。有关此更改的详细信息，请参阅 A first look at changes coming in ASP.NET Core 3.0（抢先了解 ASP.NET Core 3.0 即将推出的更改）。

面向 .NET Core 有以下几个优势，并且这些优势会随着每次发布增加。与 .NET Framework 相比，.NET Core 的部分优势包括：

- 跨平台。在 macOS、Linux 和 Windows 上运行。
- 提高的性能
- 并行版本控制
- 新 API
- 开源

我们正努力缩小 .NET Framework 与 .NET Core 的 API 差距。Windows 兼容性包使数千个仅可在Windows运行的API 可在 .NET Core 中使用。这些 API 在 .NET Core 1.x 中不可用。

如何下载示例

很多文章和教程中都包含有示例代码链接。

1. 下载 ASP.NET 存储库 zip 文件。
2. 解压缩 Docs-master.zip 文件。
3. 使用示例链接中的 URL 帮助你导航到示例目录。

示例代码中的预处理器指令

为了演示多个方案，示例应用将使用 `#define` 和 `#if-#else/#elif-#endif` C# 语句选择性地编译和运行不同的示例代码段。对于那些利用此方法的示例，请将 C# 文件顶部的 `#define` 语句设置为与你想要运行的方案相关联的符号。一些示例要求在多个文件的顶部设置符号才能运行方案。

例如，以下 `#define` 符号列表指示四个方案可用（每个符号一个方案）。当前示例配置运行 `TemplateCode` 方案：

```
#define TemplateCode // or LogFromMain or ExpandDefault or FilterInCode
```

若要更改示例以运行 `ExpandDefault` 方案，请定义 `ExpandDefault` 符号并保留剩余的符号处于被注释掉的状态：

```
#define ExpandDefault // TemplateCode or LogFromMain or FilterInCode
```

若要详细了解如何使用 C# 预处理器指令选择性地编译代码段，请参阅 #define（C# 参考）和 #if（C# 参考）。

示例代码中的区域

一些示例应用包含由 #region 和 #end-region C# 语句包围的代码段。文档生成系统会将这些区域注入到所呈现的文档主题中。

区域名称通常包含“代码段”一词。下面的示例显示了一个名为 `snippet_FilterInCode` 的区域：

```
#region snippet_FilterInCode
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging =>
        logging.AddFilter("System", LogLevel.Debug)
            .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace))
            .Build();
#endregion
```

主题的 markdown 文件在以下行中应用了前面的 C# 代码段：

```
[!code-csharp[](sample/SampleApp/Program.cs?name=snippet_FilterInCode)]
```

你可放心忽略（或删除）代码两侧的 `#region` 和 `#end-region` 语句。如果计划运行主题中所述的示例方案，请不要更改这些语句中的代码。试用其他方案时，可随时更改代码。

有关详细信息，请参阅参与 ASP.NET 文档：代码片段。

后续步骤

有关更多信息，请参见以下资源：

- Razor 页面入门
- 使用 Visual Studio 将 ASP.NET Core 应用发布到 Azure
- ASP.NET Core 基础知识
- 每周 ASP.NET Community Standup 介绍了团队的工作进度和计划。它以新博客和第三方软件为重点。

在 ASP.NET 4.x 和 ASP.NET Core 之间进行选择

2019/1/8 • • Edit Online

ASP.NET Core 是 ASP.NET 4.x 的重新设计。本文列出了两者之间的区别。

ASP.NET Core

ASP.NET Core 是一个跨平台的开源框架，用于在 Windows、macOS 或 Linux 上生成基于云的新式 Web 应用。

ASP.NET Core 具有如下优点：

- 生成 Web UI 和 Web API 的统一场景。
- 针对可测试性进行构建。
- Razor Pages 可以使基于页面的编码方式更简单高效。
- 能够在 Windows、macOS 和 Linux 上进行开发和运行。
- 开放源代码和以社区为中心。
- 集成新式客户端框架和开发工作流。
- 基于环境的云就绪配置系统。
- 内置依赖项注入。
- 轻型的高性能模块化 HTTP 请求管道。
- 能够在 IIS、Nginx、Apache、Docker 上进行托管或在自己的进程中进行自托管。
- 定目标到 .NET Core 时，可以使用并行应用版本控制。
- 简化新式 Web 开发的工具。

ASP.NET 4.x

ASP.NET 4.x 是一个成熟的框架，提供在 Windows 上生成基于服务器的企业级 Web 应用所需的服务。

框架选择

下表将 ASP.NET Core 与 ASP.NET 4.x 进行比较。

ASP.NET CORE	ASP.NET 4.X
针对 Windows、macOS 或 Linux 进行生成	针对 Windows 进行生成
Razor 页面 是在 ASP.NET Core 2.x 及更高版本中创建 Web UI 时建议使用的方法。另请参阅 MVCWeb API 和 SignalR。	使用 Web 窗体、SignalR、MVC、Web API、WebHooks 或网页
每个计算机多个版本	每个计算机一个版本
使用 C# 或 F# 通过 Visual Studio、Visual Studio for Mac 或 Visual Studio Code 进行开发	使用 C#、VB 或 F# 通过 Visual Studio 进行开发
比 ASP.NET 4.x 性能更高	良好的性能
选择 .NET Framework 或 .NET Core 运行时	使用 .NET Framework 运行时

有关 .NET Framework 上的 ASP.NET Core 2.x 支持的信息，请参阅面向 .NET Framework 的 ASP.NET Core。

ASP.NET Core 方案

- Razor 页面 是在 ASP.NET Core 2.x 及更高版本中创建 Web UI 时建议使用的方法。
- 网站
- API
- 实时
- 将 ASP.NET Core 应用部署到 Azure

ASP.NET 4.x 方案

- 网站
- API
- 实时
- 在 Azure 中创建 ASP.NET 4.x Web 应用

其他资源

- ASP.NET 简介
- ASP.NET Core 简介
- 将 ASP.NET Core 应用部署到 Azure 应用服务

教程：ASP.NET Core 入门

2018/12/13 • • Edit Online

本教程演示如何使用 .NET Core 命令行接口创建 ASP.NET Core Web 应用。

你将了解如何：

- 创建 Web 应用项目。
- 启用本地 HTTPS。
- 运行应用。
- 编辑 Razor 页面。

最后，在本地计算机上运行工作 Web 应用。

系统必备

- .NET Core 2.2 SDK

创建 Web 应用项目

打开命令行界面，然后输入以下命令：

```
dotnet new webapp -o aspnetcoreapp
```

启用本地 HTTPS

信任 HTTPS 开发证书：

- Windows
- macOS
- Linux

```
dotnet dev-certs https --trust
```

以上命令会显示以下对话：

如果你同意信任开发证书，请选择“是”。

运行应用

运行以下命令：

```
cd aspnetcoreapp
dotnet run
```

浏览到 https://localhost:5001。单击“接受”，接受隐私和 cookie 政策。此应用不保留个人信息。

编辑 Razor 页面

打开 Pages/About.cshtml，并使用以下突出显示标记修改页面：

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Hello, world! The time on the server is @DateTime.Now</p>
</div>
```

转到 https://localhost:5001，并验证更改是否显示。

后续步骤

在本教程中，你将了解：

- 创建 Web 应用项目。
- 启用本地 HTTPS。
- 运行该项目。
- 进行更改。

若要了解有关 ASP.NET Core 的详细信息，请参阅简介：

ASP.NET Core 简介

ASP.NET Core 2.2 的新增功能

2019/1/10 • • Edit Online

本文重点介绍 ASP.NET Core 2.2 中最重要的更改，并提供相关文档的链接。

开放 API 分析器和约定

开放 API（也称为 Swagger）是一个与语言无关的规范，用于描述 REST API。开放 API 生态系统具有一些工具，可用于发现、测试和生成使用该规范的客户端代码。对在 ASP.NET Core MVC 中生成和可视化开放 API 文档的支持通过社区驱动型项目（如 NSwag 和 Swashbuckle.AspNetCore）来提供。对于创建开放 API 文档，ASP.NET Core 2.2 提供了改进的工具和运行时体验。

有关更多信息，请参见以下资源：

- 使用 Web API 分析器
- 使用 Web API 约定
- ASP.NET Core 2.2.0-preview1：开放 API 分析器和约定

问题详细信息支持

ASP.NET Core 2.1 引入了 `ProblemDetails`（基于用于通过 HTTP 响应传送错误详细信息的 RFC 7807 规范）。在 2.2 中，`ProblemDetails` 是针对具有 `ApiControllerAttribute` 特性的控制器中客户端错误代码的标准响应。返回客户端错误状态代码 (4xx) 的 `IActionResult` 现在返回 `ProblemDetails` 正文。结果还包括可以用于将使用请求日志的错误相关联的关联 ID。对于客户端错误，`ProducesResponseType` 默认为使用 `ProblemDetails` 作为响应类型。这会记录在使用 NSwag 或 Swashbuckle.AspNetCore 生成的开放 API/Swagger 输出中。

终结点路由

ASP.NET Core 2.2 使用新的终结点路由系统来改进请求的调度。更改包括新链接生成 API 成员和路由参数转换器。

有关更多信息，请参见以下资源：

- 2.2 中的终结点路由
- 路由参数转换器（请参阅"路由"部分）
- 基于 IRouter 与基于终结点的路由之间的差异

运行状况检查

通过新的运行状况检查服务可以更轻松地在需要运行状况检查的环境（如 Kubernetes）中使用 ASP.NET Core。运行状况检查包括中间件和一组定义 `IHealthCheck` 抽象和服务的库。

运行状况检查由容器业务流程协调程序或负载均衡器用于快速确定系统是否正常响应请求。容器业务流程协调程序可以通过停止滚动部署或重新启动容器来响应失败的运行状况检查。负载均衡器可以通过路由流量以避开失败的服务实例，来响应运行状况检查。

运行状况检查由应用程序公开为监视系统所使用的 HTTP 终结点。可以为各种实时监视方案和监视系统配置运行状况检查。运行状况检查服务与 BeatPulse 项目集成。从而可以更轻松地添加为众多常用系统和依赖项添加检查。

有关详细信息，请参阅 ASP.NET Core 中的运行状况检查。

Kestrel 中的 HTTP/2

ASP.NET Core 2.2 添加了对 HTTP/2 的支持。

HTTP/2 是 HTTP 协议的主要修订版本。HTTP/2 的一些值得注意的功能包括支持标头压缩，以及单个连接上的完整多路复用流。尽管 HTTP/2 保留了 HTTP 的语义（HTTP 标头、方法等），不过与 HTTP/1.x 相比，在线路上对此数据进行组帧和发送的方式发生了重大更改。

由于组帧中的这一更改，服务器和客户端需要协商所使用的协议版本。应用层协议协商 (ALPN) 是一种 TLS 扩展，允许服务器和客户端协商用作其 TLS 握手一部分的协议版本。虽然服务器与客户端之间可以事先了解协议，不过所有主要浏览器都支持 ALPN 作为建立 HTTP/2 连接的唯一方法。

有关详细信息，请参阅 HTTP/2 支持。

Kestrel 配置

在早期版本的 ASP.NET Core 中，Kestrel 选项通过调用 `UseKestrel` 来配置。在 2.2 中，Kestrel 选项通过在主机生成器上调用 `ConfigureKestrel` 来配置。此更改为进程内承载解决了 `IServer` 注册顺序方面的问题。有关更多信息，请参见以下资源：

- 缓解 UseIIS 冲突
- 使用 ConfigureKestrel 配置 Kestrel 服务器选项

IIS 进程内承载

在早期版本的 ASP.NET Core 中，IIS 用作反向代理。在 2.2 中，ASP.NET Core 模块可以启动 CoreCLR 并在 IIS 工作进程 (w3wp.exe) 内承载应用。在使用 IIS 运行时，进程内承载可提供性能和诊断提升。

有关详细信息，请参阅 IIS 进程内承载。

SignalR Java 客户端

ASP.NET Core 2.2 引入了适用于 SignalR 的 Java 客户端。此客户端支持通过 Java 代码连接到 ASP.NET Core SignalR 服务器（包括 Android 应用）。

有关详细信息，请参阅 ASP.NET Core SignalR Java 客户端。

CORS 改进

在早期版本的 ASP.NET Core 中，CORS 中间件允许发送 `Accept`、`Accept-Language`、`Content-Language` 和 `Origin` 标头（不考虑在 `CorsPolicy.Headers` 中配置的值）。在 2.2 中，仅当在 `Access-Control-Request-Headers` 中发送的标头与 `WithHeaders` 中声明的标头完全匹配时，才能进行 CORS 中间件策略匹配。

有关详细信息，请参阅 CORS 中间件。

响应压缩

ASP.NET Core 2.2 可以使用 Brotli 压缩格式来压缩响应。

有关详细信息，请参阅响应压缩中间件支持 Brotli 压缩。

项目模板

ASP.NET Core Web 项目模板已更新为 Bootstrap 4 和 Angular 6。新的外观在视觉上更简单，可以更轻松地查看应用的重要结构。

验证性能

MVC 的验证系统设计为具有可扩展性和灵活性，从而使你可以基于每个请求确定应用于给定模型的验证程序。这非常适用于创作复杂的验证提供程序。但是在最常见的情况下，应用程序仅使用内置验证程序，无需这一额外的灵活性。内置验证程序包括 DataAnnotations（如 [Required] 和 [StringLength]）和 `IValidatableObject`。

在 ASP.NET Core 2.2 中，如果 MVC 确定给定模型关系图不需要验证，则可以绕过验证。在验证无法具有或不具有任何验证程序的模型时，跳过验证可获得显著改进。这包括诸如基元集合（如 `byte[]`、`string[]`、`Dictionary<string, string>`）之类的对象，或没有许多验证程序的复杂对象关系图。

HTTP 客户端性能

在 ASP.NET Core 2.2 中，通过减少连接池锁定争用提高了 `SocketsHttpHandler` 的性能。对于进行大量传出 HTTP 请求的应用（如某些微服务体系结构），吞吐量得到了提高。在负载下，`HttpClient` 吞吐量在 Linux 上可以提高多达 60%，在 Windows 上提高多达 20%。

有关详细信息，请参阅实现此改进的拉取请求。

其他信息

要获取完整的更改列表，请参阅 ASP.NET Core 2.2 发行说明。

ASP.NET Core 2.1 的新增功能

2018/10/30 • • Edit Online

本文重点介绍 ASP.NET Core 2.1 中最重要的更改，并提供相关文档的链接。

SignalR

已针对 ASP.NET Core 2.1 重新编写 SignalR。ASP.NET Core SignalR 包含大量改进：

- 简化横向扩展模型。
- 新 JavaScript 客户端不具有 jQuery 依赖项。
- 新紧凑型二进制协议基于 MessagePack。
- 支持自定义协议。
- 新的流式处理响应模型。
- 支持基于裸机 WebSocket 的客户端。

有关详细信息，请参阅 ASP.NET Core SignalR。

Razor 类库

通过 ASP.NET Core 2.1 可以更容易地在库中生成和包括基于 Razor 的 UI，并跨多个项目共享 UI。新 Razor SDK 支持将 Razor 文件生成到可打包为 NuGet 包的类库项目中。应用可以自动发现和覆盖库中的视图和页面。通过将 Razor 编译集成到生成中：

- 应用启动时间可显著加快。
- 在迭代开发工作流过程中，仍可在运行时快速更新 Razor 视图和页面。

有关详细信息，请参阅使用 Razor 类库项目创建可重用 UI。

标识 UI 库和基架

ASP.NET Core 2.1 提供 ASP.NET Core 标识作为 Razor 类库。包含标识的应用可以应用新的标识基架，以便有选择地添加标识 Razor 类库 (RCL) 中包含的源代码。建议生成源代码，以便修改代码和更改行为。例如，可以指示基架生成在注册过程中使用的代码。生成的代码优先于标识 RCL 中的相同代码。

不包含身份验证的应用可以应用标识基架以添加 RCL 标识包。可以选择要生成的标识代码。

有关详细信息，请参阅 ASP.NET Core 项目中的基架标识。

HTTPS

随着对安全和隐私的关注度日益增加，为 Web 应用启用 HTTPS 变得非常重要。Web 上正在越来越严格要求强制使用 HTTPS。不使用 HTTPS 的站点会被视为不安全的站点。浏览器(Chromium、Mozilla)开始强制要求必须在安全的上下文中使用 Web 功能。GDPR 要求使用 HTTPS 保护用户隐私。不仅在生产环境中使用 HTTPS 至关重要，而且在开发环境中使用 HTTPS 还可以帮助防止部署中的各种问题(例如，不安全的链接)。ASP.NET Core 2.1 包含大量改进，更方便在开发环境使用 HTTPS 和在生产环境配置 HTTPS。有关详细信息，请参阅强制使用 HTTPS。

默认开启

为了提高网站开发的安全性，现在默认启用 HTTPS。从 2.1 开始，当本地具有开发证书时，Kestrel 将侦听 `https://localhost:5001`。关于开发证书的创建：

- 首次使用 SDK 时，在首次运行 .NET Core SDK 时会创建开发证书。

- 手动使用新 `dev-certs` 工具。

运行 `dotnet dev-certs https --trust` 以信任证书。

HTTPS 重定向和强制使用

Web 应用通常需要侦听 HTTP 和 HTTPS，但随后会将所有 HTTP 流量重定向到 HTTPS。在 2.1 中，引入了专用的 HTTPS 重定向中间件，可根据配置或绑定服务器端口的存在智能执行重定向。

使用 HTTP 严格传输安全协议 (HSTS) 可进一步强制使用 HTTPS。HSTS 指示浏览器始终通过 HTTPS 访问站点。ASP.NET Core 2.1 添加 HSTS 中间件，该中间件支持选择最大年限、子域和 HSTS 预加载列表。

适用于生产环境的配置

在生产环境中，必须显式配置 HTTPS。在 2.1 中，添加了针对 Kestrel 配置 HTTPS 的默认配置架构。可以将应用配置为使用：

- 多个终结点（包括 URL）。有关详细信息，请参阅 Kestrel Web 服务器实现：终结点配置。
- 用于 HTTPS 的证书可能来自于磁盘上的文件或证书存储。

GDPR

ASP.NET Core 提供 API 和模板，帮助满足欧盟一般数据保护条例 (GDPR) 的部分要求。有关详细信息，请参阅 ASP.NET Core 中的 GDPR 支持。示例应用演示如何使用并允许测试已添加到 ASP.NET Core 2.1 模板中的大多数 GDPR 扩展点和 API。

集成测试

引入了可简化创建和执行测试的新包。Microsoft.AspNetCore.Mvc.Testing 包可处理以下任务：

- 将依赖项文件 (*.deps) 从已测试的应用复制到测试项目的 bin 文件夹中。
- 将内容根目录设置为已测试应用的项目根目录，以便可在执行测试时找到静态文件和页面/视图。
- 提供 WebApplicationFactory 类，以简化已测试应用在 TestServer 中的启动过程。

以下测试使用 xUnit 检查索引页是否加载了成功状态代码和正确的 Content-Type 标头：

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly HttpClient _client;

    public BasicTests(WebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task GetHomePage()
    {
        // Act
        var response = await _client.GetAsync("/");

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
}
```

有关详细信息，请参阅集成测试主题。

[ApiController], ActionResult<T>

ASP.NET Core 2.1 添加了新编程约定，方便构建简洁的说明性 Web API。`ActionResult<T>` 是一个新增类型，可允许应用返回响应类型，或返回任何其他操作结果（与 IActionResult 相似），同时仍然指示响应类型。此外还添加了 `[ApiController]` 特性，作为选择加入特定于 Web API 的约定和行为的方式。

有关详细信息，请参阅使用 ASP.NET Core 构建 Web API。

IHttpClientFactory

ASP.NET Core 2.1 引入了新的 `IHttpClientFactory` 服务，方便在应用中配置和使用 `HttpClient` 的实例。`HttpClient` 已经具有委托处理程序的概念，这些委托处理程序可以链接在一起，处理出站 HTTP 请求。工厂可以：

- 使根据已命名客户端注册 `HttpClient` 的实例变得更直观。
- 实现一个 Polly 处理程序，允许将 Polly 策略用于 Retry、CircuitBreakers 等。

有关详细信息，请参阅启动 HTTP 请求。

Kestrel 传输配置

对于 ASP.NET Core 2.1 版，Kestrel 默认传输不再基于 Libuv，而是基于托管的套接字。有关详细信息，请参阅 Kestrel Web 服务器实现：传输配置。

通用主机生成器

引入了通用主机生成器（`HostBuilder`）。此生成器可用于不处理 HTTP 请求（消息传送、后台任务等等）的应用。

有关详细信息，请参阅 .NET 通用主机。

更新的 SPA 模板

更新了适用于 Angular、React 和 React 结合 Redux 的单页应用程序模板，以使用标准项目结构和为每个框架构建系统。

Angular 模板基于 Angular CLI，而 React 模板基于 create-react-app。有关详细信息，请参阅通过 ASP.NET Core 使用单页应用程序模板。

Razor Pages 搜索 Razor 资产

在 2.1 中，Razor Pages 按所列顺序搜索以下目录中的 Razor 资产（例如布局和分区）：

1. 当前 Pages 文件夹。
2. /Pages/Shared/
3. /Views/Shared/

某个区域内的 Razor Pages

Razor Pages 现在支持区域。要查看区域示例，请使用个人用户帐户创建新 Razor Pages Web 应用。使用个人用户帐户的 Razor Pages Web 应用包括 /Areas/Identity/Pages。

MVC 兼容性版本

SetCompatibilityVersion 方法允许应用选择加入或退出 ASP.NET Core MVC 2.1 或更高版本中引入的潜在中断行为变更。

有关更多信息，请参见ASP.NET Core MVC 的兼容性版本。

从 2.0 迁移到 2.1

请参阅从 ASP.NET Core 2.0 迁移到 2.1。

其他信息

要获取完整的更改列表，请参阅 ASP.NET Core 2.1 发行说明。

ASP.NET Core 2.0 中的新增功能

2018/10/30 • • Edit Online

本文重点介绍 ASP.NET Core 2.0 中最重要的更改，并提供相关文档的链接。

Razor 页面

Razor 页面是 ASP.NET Core MVC 的一个新功能，它可以使基于页面的编码方式更简单高效。

有关详细信息，请参阅相关介绍和教程：

- Razor 页面介绍
- Razor 页面入门

ASP.NET Core 元包

新的 ASP.NET Core 元包包含 ASP.NET Core 和 Entity Framework 团队生成和提供支持的所有包及其内部和第三方依赖项。无需再通过包选择单个 ASP.NET Core 功能。Microsoft.AspNetCore.All 包中包含所有的功能。默认模板使用此包。

有关详细信息，请参阅 ASP.NET Core 2.0 的 Microsoft.AspNetCore.All 元包。

运行时存储

使用 `Microsoft.AspNetCore.All` 元包的应用程序会自动使用新的 .NET Core 运行时存储。此存储包含运行 ASP.NET Core 2.0 应用程序所需的所有运行时资产。使用 `Microsoft.AspNetCore.All` 元包时，应用程序不会部署引用的 ASP.NET Core NuGet 包中的任何资产，因为目标系统中已存在这些资产。运行时存储中的资产也已经过预编译，以便缩短应用程序启动时间。

有关详细信息，请参阅运行时存储

.NET Standard 2.0

ASP.NET Core 2.0 包面向 NET Standard 2.0。这些包可以由其他 .NET Standard 2.0 库引用，也可以在兼容 .NET Standard 2.0 的 .NET 实现上运行，其中包括 .NET Core 2.0 和 .NET Framework 4.6.1。

`Microsoft.AspNetCore.All` 元包仅面向 .NET Core 2.0，因为它旨在与 .NET Core 2.0 运行时存储一起使用。

配置更新

在 ASP.NET Core 2.0 中，已默认将 `IConfiguration` 实例添加到服务容器。服务容器中的 `IConfiguration` 可以使应用程序更轻松地从容器中检索配置值。

有关已规划文档的状态的信息，请参阅 GitHub 问题。

日志记录更新

在 ASP.NET Core 2.0 中，已默认将日志记录并入依存关系注入 (DI) 系统。在 Program.cs 文件（而非 Startup.cs 文件）中添加提供程序并配置筛选。此外，默认的 `ILoggerFactory` 支持进行筛选，并且你可以使用灵活的方式来进行跨提供程序筛选和特定于提供程序的筛选。

有关详细信息，请参阅日志记录介绍。

身份验证更新

新的身份验证模型简化了使用 DI 为应用程序配置身份验证的过程。

使用 [Azure AD B2C] (https://azure.microsoft.com/services/active-directory-b2c/)) 为 Web 应用和 Web API 配置身份验证时可使用新模板。

有关已规划文档的状态的信息，请参阅 GitHub 问题。

标识更新

在 ASP.NET Core 2.0 中，我们简化了使用标识生成安全的 Web API 的过程。可以使用 Microsoft 身份验证库 (MSAL)获取用于访问 Web API 的访问令牌。

有关 2.0 中的身份验证更改的详细信息，请参阅以下资源：

- ASP.NET Core 中的帐户确认和密码恢复
- 为 ASP.NET Core 中的验证器应用启用 QR 码生成
- 将身份验证和标识迁移到 ASP.NET Core 2.0

SPA 模板

已提供适用于 Angular、Aurelia、Knockout.js、React.js 及 React.js 和 Redux 的单页应用程序 (SPA) 项目模板。Angular 模板已更新至 Angular 4。默认情况下，Angular 和 React 模板已可用；有关如何获取其他模板的信息，请参阅新建 SPA 项目。有关如何在 ASP.NET Core 中生成 SPA 的信息，请参阅使用 JavaScriptServices 创建单页应用程序。

Kestrel 改进

Kestrel Web 服务器包含一项新功能，使其更适合作为面向 Internet 的服务器。在 `KestrelServerOptions` 类的新 `Limits` 属性中添加大量服务器约束配置选项。为以下内容添加限制：

- 客户端最大连接数
- 请求正文最大大小
- 请求正文最小数据速率

有关详细信息，请参阅 ASP.NET Core 中的 Kestrel Web 服务器实现。

WebListener 已重命名为 HTTP.sys

`Microsoft.AspNetCore.Server.WebListener` 和 `Microsoft.Net.Http.Server` 包已合并为一个新包 `Microsoft.AspNetCore.Server.HttpSys`。命名空间已进行更新以保持一致。

有关详细信息，请参阅 ASP.NET Core 中的 HTTP.sys Web 服务器实现。

增强了 HTTP 标头支持

使用 MVC 传输 `FileStreamResult` 或 `FileContentResult` 时，现在可以选择对传输的内容设置 `ETag` 或 `LastModified` 日期。可以使用如下所示的代码在返回的内容上设置这些值：

```
var data = Encoding.UTF8.GetBytes("This is a sample text from a binary array");
var entityTag = new EntityTagHeaderValue("\"MyCalculatedEtagValue\"");
return File(data, "text/plain", "downloadName.txt", lastModified: DateTime.UtcNow.AddSeconds(-5), entityTag:
entityTag);
```

返回给访问者的文件将附带 `ETag` 和 `LastModified` 值的适当 HTTP 标头。

如果应用程序访问者使用范围请求标头请求内容，ASP.NET Core 将识别出该请求，并会处理该标头。如果可以对请求的内容执行部分传输操作，ASP.NET Core 将适当地跳过一些内容，只返回请求的字节集。不必为了采用或处理此功能而将任何特殊的处理程序写入方法；系统会自动处理。

托管启动和 Application Insights

托管环境现在可以在应用程序启动时插入额外的包依赖项并执行代码，而应用程序无需显式使用依赖项或调用任何方法。可以使用此功能来允许某些环境"启用"该环境特有的功能，而应用程序无需提前获知。

在 ASP.NET Core 2.0 中，如果在 Visual Studio 中调试并且（选择加入后）在 Azure App Services 中运行，将使用此功能自动启用 Application Insights 诊断。因此，默认情况下，项目模板不再添加 Application Insights 包和代码。

有关已规划文档的状态的信息，请参阅 GitHub 问题。

自动使用防伪标记

默认情况下，ASP.NET Core 始终在帮助对内容进行 HTML 编码，但是在新版本中，还采用了额外的措施来帮助预防跨网站请求伪造 (XSRF) 攻击。现在在默认情况下，ASP.NET Core 会发出防伪标记，并在窗体 POST 操作和页面上验证它们，且无需其他配置。

有关详细信息，请参阅预防跨网站请求伪造 (XSRF/CSRF) 攻击。

自动预编译

默认情况下，会在发布时启用 Razor 视图预编译，以缩减发布输出大小和应用程序启动时间。

有关详细信息，请参阅 ASP.NET Core 中的 Razor 视图编译和预编译。

Razor 支持 C# 7.1

Razor 视图引擎已更新为可使用新的 Roslyn 编译器。其中包含对 C# 7.1 功能的支持，例如默认表达式、推断元组名称和泛型模式匹配。若要在项目中使用 C# 7.1，请在项目文件中添加以下属性，然后重新加载解决方案：

```
<LangVersion>latest</LangVersion>
```

有关 C# 7.1 功能的状态的信息，请参阅 Roslyn GitHub 存储库。

2.0 的其他文档更新

- 用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件
- 密钥管理
- 配置 Facebook 身份验证
- 配置 Twitter 身份验证
- 配置 Google 身份验证
- 配置 Microsoft 帐户身份验证

迁移指南

有关如何将 ASP.NET Core 1.x 应用程序迁移到 ASP.NET Core 2.0 的指南，请参阅以下资源：

- 从 ASP.NET Core 1.x 迁移到 ASP.NET Core 2.0
- 将身份验证和标识迁移到 ASP.NET Core 2.0

其他信息

有关更改的完整列表，请参阅 ASP.NET Core 2.0 发行说明。

若要实时了解 ASP.NET Core 开发团队的进度和计划，请收看 ASP.NET Community Standup。

ASP.NET Core 1.1 的新增功能

2019/1/10 • • Edit Online

ASP.NET Core 1.1 新增了以下功能：

- URL 重写中间件
- 响应缓存中间件
- 查看组件即标记帮助程序
- MVC 型中间件筛选器
- 基于 Cookie 的 TempData 提供程序
- Azure App Service 日志记录提供程序
- Azure Key Vault 配置提供程序
- Azure 和 Redis 存储数据保护密钥存储库
- 适用于 Windows 的 WebListener 服务器
- WebSockets 支持

在 ASP.NET Core 的 1.0 和 1.1 版本之间进行选择

ASP.NET Core 1.1 比 1.0 功能更多。通常情况下，建议使用最新版本。

其他信息

- ASP.NET Core 1.1.0 发行说明
- 若要实时了解 ASP.NET Core 开发团队的进度和计划，请收看 ASP.NET Community Standup。

教程：使用 ASP.NET Core MVC 创建 Web API

2019/1/10 • • Edit Online

作者：Rick Anderson 和 Mike Wasson

本教程介绍使用 ASP.NET Core 构建 Web API 的基础知识。

在本教程中，你将了解：

- 创建 Web API 项目。
- 添加模型类。
- 创建数据库上下文。
- 注册数据库上下文。
- 添加控制器。
- 添加 CRUD 方法。
- 配置路由和 URL 路径。
- 指定返回值。
- 使用 Postman 调用 Web API。
- 使用 jQuery 调用 Web API。

在结束时，你会获得可以管理存储在关系数据库中的“待办事项”的 Web API。

概述

本教程将创建以下 API：

API	说明	请求正文	响应正文
GET /api/todo	获取所有待办事项	无	待办事项的数组
GET /api/todo/{id}	按 ID 获取项	无	待办事项
POST /api/todo	添加新项	待办事项	待办事项
PUT /api/todo/{id}	更新现有项	待办事项	无
DELETE /api/todo/{id}	删除项	无	无

下图显示了应用的设计。

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

创建 Web 项目

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 从“文件”菜单中选择“新建” > “项目”。
- 选择“ASP.NET Core Web 应用程序”模板。将项目命名为 TodoApi，然后单击“确定”。
- 在“新建 ASP.NET Core Web 应用程序 - TodoApi”对话框中，选择 ASP.NET Core 版本。选择“API”模板，然后单击“确定”。请不要选择“启用 Docker 支持”。

测试 API

项目模板会创建 `values` API。从浏览器调用 `Get` 方法以测试应用。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

按 Ctrl+F5 运行应用。Visual Studio 启动浏览器并导航到 `https://localhost:<port>/api/values`，其中 `<port>` 是随机选择的端口号。

如果出现询问是否应信任 IIS Express 证书的对话框，则选择"是"。在接下来出现的"安全警告"对话框中，选择"是"。

会返回以下 JSON：

```
["value1","value2"]
```

添加模型类

模型是一组表示应用管理的数据的类。此应用的模型是单个 `TodoItem` 类。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 在"解决方案资源管理器"中，右键单击项目。选择"添加" > "新建文件夹"。将文件夹命名为"Models"。
- 右键单击"Models"文件夹，然后选择"添加" > "类"。将类命名为 TodoItem，然后选择"添加"。
- 将模板代码替换为以下代码：

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

`Id` 属性用作关系数据库中的唯一键。

模型类可位于项目的任意位置，但按照惯例会使用 Models 文件夹。

添加数据库上下文

数据库上下文是为数据模型协调 Entity Framework 功能的主类。此类由 `Microsoft.EntityFrameworkCore.DbContext` 类派生而来。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 右键单击“Models”文件夹，然后选择“添加” > “类”。将类命名为 TodoContext，然后单击“添加”。
- 将模板代码替换为以下代码：

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

注册数据库上下文

在 ASP.NET Core 中，服务（如数据库上下文）必须向依赖关系注入 (DI) 容器进行注册。该容器向控制器提供服务。

使用以下突出显示的代码更新 Startup.cs：

```
// Unused usings removed
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the
        //container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt =>
                opt.UseInMemoryDatabase("TodoList"));
            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
        }

        // This method gets called by the runtime. Use this method to configure the HTTP
        //request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                // The default HSTS value is 30 days. You may want to change this for
                // production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseMvc();
        }
    }
}
```

前面的代码:

- 删除未使用的 `using` 声明。
- 将数据库上下文添加到 DI 容器。
- 指定数据库上下文将使用内存中数据库。

添加控制器

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 右键单击 Controllers 文件夹。

- 选择 **添加** > **新建**项。
- 在"添加新项"对话框中，选择"API 控制器类"模板。
- 将类命名为 TodoController，然后选择"添加"。

- 将模板代码替换为以下代码：

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using TodoApi.Models;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoController : ControllerBase
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                // Create a new TodoItem if collection is empty,
                // which means you can't delete all TodoItems.
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

前面的代码：

- 定义了没有方法的 API 控制器类。
- 使用 `[ApiController]` 属性修饰类。此属性指示控制器响应 Web API 请求。有关该属性启用的特定行为的信息，请参阅使用 ApiController 属性进行注释。

- 使用 DI 将数据库上下文（`TodoContext`）注入到控制器中。数据库上下文将在控制器中的每个 CRUD 方法中使用。
- 如果数据库为空，则将名为 `Item1` 的项添加到数据库。此代码位于构造函数中，因此在每次出现新 HTTP 请求时运行。如果删除所有项，则构造函数会在下次调用 API 方法时再次创建 `Item1`。因此删除可能看上去不起作用，不过实际上确实有效。

添加 Get 方法

若要提供检索待办事项的 API，请将以下方法添加到 `TodoController` 类中：

```
// GET: api/Todo
[HttpGet]
public async Task<ActionResult<IEnumerable<TodoItem>>> GetTodoItems()
{
    return await _context.TodoItems.ToListAsync();
}

// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

这些方法实现两个 GET 终结点：

- `GET /api/todo`
- `GET /api/todo/{id}`

通过从浏览器调用两个终结点来测试应用。例如：

- `https://localhost:<port>/api/todo`
- `https://localhost:<port>/api/todo/1`

以下 HTTP 响应通过调用 `GetTodoItems` 来生成：

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

路由和 URL 路径

`[HttpGet]` 属性表示响应 HTTP GET 请求的方法。每个方法的 URL 路径构造如下所示：

- 在控制器的 `Route` 属性中以模板字符串开头：

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoController : ControllerBase
    {
        private readonly TodoContext _context;
```

- 将 `[controller]` 替换为控制器的名称，按照惯例，在控制器类名称中去掉"Controller"后缀。对于此示例，控制器类名称为"Todo"控制器，因此控制器名称为"todo"。ASP.NET Core 路由不区分大小写。
- 如果 `[HttpGet]` 属性具有路由模板（例如 `[HttpGet("/products")]`），则将它追加到路径。此示例不使用模板。有关详细信息，请参阅使用 Http [Verb] 特性的特性路由。

在下面的 `GetTodoItem` 方法中，`"{id}"` 是待办事项的唯一标识符的占位符变量。调用 `GetTodoItem` 时，URL 中 `"{id}"` 的值会在 `id` 参数中提供给方法。

```
// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

`Name = "GetTodo"` 参数会创建命名路由。你会在后面看到应用如何使用名称创建使用路由名称的 HTTP 链接。

返回值

`GetTodoItems` 和 `GetTodoItem` 方法的返回类型是 ActionResult<T> 类型。ASP.NET Core 自动将对象序列化为 JSON，并将 JSON 写入响应消息的正文中。在假设没有未经处理的异常的情况下，此返回类型的响应代码为 200。未经处理的异常将转换为 5xx 错误。

`ActionResult` 返回类型可以表示大范围的 HTTP 状态代码。例如，`GetTodoItem` 可以返回两个不同的状态值：

- 如果没有任何项与请求的 ID 匹配，则该方法将返回 404 NotFound 错误代码。
- 否则，此方法将返回具有 JSON 响应正文的 200。返回 `item` 则产生 HTTP 200 响应。

测试 GetTodoItems 方法

本教程使用 Postman 测试 Web API。

- 安装 Postman
- 启动 Web 应用。
- 启动 Postman。
- 禁用 **SSL** 证书验证
  - 在"文件">"设置"（"常规"*选项卡）中，禁用"SSL 证书验证"。

> **WARNING**
>
> 在测试控制器之后重新启用 SSL 证书验证。

- 创建新请求。
  - 将 HTTP 方法设置为“GET”。
  - 将请求 URL 设置为 `https://localhost:<port>/api/todo`。例如 `https://localhost:5001/api/todo`。
- 在 Postman 中设置“两窗格视图”。
- 选择“发送”。

添加创建方法

添加以下 `PostTodoItem` 方法：

```
// POST: api/Todo
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem);
}
```

正如 [HttpPost] 属性所指示，前面的代码是 HTTP POST 方法。该方法从 HTTP 请求正文获取待办事项的值。

`CreatedAtRoute` 方法：

- 返回 201 响应。HTTP 201 是在服务器上创建新资源的 HTTP POST 方法的标准响应。
- 向响应添加位置标头。位置标头指定新建的待办事项的 URI。有关详细信息，请参阅创建的 10.2.2 201。
- 使用名为 route 的“GetTodo”来创建 URL。已在 `GetTodoItem` 中定义名为 route 的“GetTodo”：

```
// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

测试 PostTodoItem 方法

- 生成项目。
- 在 Postman 中，将 HTTP 方法设置为 `POST`。
- 选择“正文”选项卡。
- 选择“原始”单选按钮。
- 将类型设置为 JSON (application/json)
- 在请求正文中，输入待办事项的 JSON：

```
{
  "name":"walk dog",
  "isComplete":true
}
```

- 选择“发送”。

如果收到 405 不允许的方法错误，则可能是由于未在添加 `PostTodoItem` 方法之后编译项目。

测试位置标头 URI

- 在"响应"窗格中选择"标头"选项卡。
- 复制"位置"标头值：

- 将方法设置为“GET”。
- 粘贴 URI（例如，`https://localhost:5001/api/Todo/2`）
- 选择“发送”。

添加 PutTodoItem 方法

添加以下 `PutTodoItem` 方法：

```
// PUT: api/Todo/5
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;
    await _context.SaveChangesAsync();

    return NoContent();
}
```

`PutTodoItem` 与 `PostTodoItem` 类似，但是使用的是 HTTP PUT。响应是 204（无内容）。根据 HTTP 规范，PUT 请求需要客户端发送整个更新的实体，而不仅仅是更改。若要支持部分更新，请使用 HTTP PATCH。

测试 PutTodoItem 方法

更新 id = 1 的待办事项并将其名称设置为“feed fish”：

```
{
  "ID":1,
  "name":"feed fish",
  "isComplete":true
}
```

下图显示 Postman 更新：

添加 DeleteTodoItem 方法

添加以下 `DeleteTodoItem` 方法：

```
// DELETE: api/Todo/5
[HttpDelete("{id}")]
public async Task<ActionResult<TodoItem>> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return todoItem;
}
```

`DeleteTodoItem` 响应是 204（无内容）。

测试 DeleteTodoItem 方法

使用 Postman 删除待办事项：

- 将方法设置为 `DELETE`。
- 设置要删除的对象的 URI，例如 `https://localhost:5001/api/todo/1`
- 选择“Send”

示例应用允许删除所有项，但是在删除最后一项后，模型类构造函数会在下次调用 API 时创建一个新项。

使用 jQuery 调用 API

在本部分中，添加了使用 jQuery 调用 Web API 的 HTML 页面。jQuery 启动请求，并用 API 响应中的详细信息更新页面。

配置应用提供静态文件并启用默认文件映射：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // The default HSTS value is 30 days. You may want to change this for
        // production scenarios, see https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseHttpsRedirection();
    app.UseMvc();
}
```

在项目目录中创建 wwwroot 文件夹。

将一个名为 index.html 的 HTML 文件添加到 wwwroot 目录。用以下标记替代其内容：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <style>
        input[type='submit'], button, [aria-label] {
            cursor: pointer;
        }

        #spoiler {
            display: none;
        }

        table {
            font-family: Arial, sans-serif;
            border: 1px solid;
            border-collapse: collapse;
        }

        th {
            background-color: #0066CC;
            color: white;
        }

        td {
            border: 1px solid;
            padding: 5px;
        }
    </style>
</head>
<body>
    <h1>To-do CRUD</h1>
    <h3>Add</h3>
    <form action="javascript:void(0);" method="POST" onsubmit="addItem()">
        <input type="text" id="add-name" placeholder="New to-do">
        <input type="submit" value="Add">
    </form>
```

```
    </form>

    <div id="spoiler">
        <h3>Edit</h3>
        <form class="my-form">
            <input type="hidden" id="edit-id">
            <input type="checkbox" id="edit-isComplete">
            <input type="text" id="edit-name">
            <input type="submit" value="Save">
            <a onclick="closeInput()" aria-label="Close">&#10006;</a>
        </form>
    </div>

    <p id="counter"></p>

    <table>
        <tr>
            <th>Is Complete</th>
            <th>Name</th>
            <th></th>
            <th></th>
        </tr>
        <tbody id="todos"></tbody>
    </table>

    <script src="https://code.jquery.com/jquery-3.3.1.min.js"
            integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
            crossorigin="anonymous"></script>
    <script src="site.js"></script>
</body>
</html>
```

将名为 site.js 的 JavaScript 文件添加到 wwwroot 目录。用以下代码替代其内容：

```
const uri = "api/todo";
let todos = null;
function getCount(data) {
  const el = $("#counter");
  let name = "to-do";
  if (data) {
    if (data > 1) {
      name = "to-dos";
    }
    el.text(data + " " + name);
  } else {
    el.text("No " + name);
  }
}

$(document).ready(function() {
  getData();
});

function getData() {
  $.ajax({
    type: "GET",
    url: uri,
    cache: false,
    success: function(data) {
      const tBody = $("#todos");

      $(tBody).empty();

      getCount(data.length);

      $.each(data, function(key, item) {
        const tr = $("<tr></tr>")
          append(
```

```
          .append(
            $("<td></td>").append(
              $("<input/>", {
                type: "checkbox",
                disabled: true,
                checked: item.isComplete
              })
            )
          )
          .append($("<td></td>").text(item.name))
          .append(
            $("<td></td>").append(
              $("<button>Edit</button>").on("click", function() {
                editItem(item.id);
              })
            )
          )
          .append(
            $("<td></td>").append(
              $("<button>Delete</button>").on("click", function() {
                deleteItem(item.id);
              })
            )
          );

        tr.appendTo(tBody);
      });

      todos = data;
    }
  });
}

function addItem() {
  const item = {
    name: $("#add-name").val(),
    isComplete: false
  };

  $.ajax({
    type: "POST",
    accepts: "application/json",
    url: uri,
    contentType: "application/json",
    data: JSON.stringify(item),
    error: function(jqXHR, textStatus, errorThrown) {
      alert("Something went wrong!");
    },
    success: function(result) {
      getData();
      $("#add-name").val("");
    }
  });
}

function deleteItem(id) {
  $.ajax({
    url: uri + "/" + id,
    type: "DELETE",
    success: function(result) {
      getData();
    }
  });
}

function editItem(id) {
  $.each(todos, function(key, item) {
    if (item.id === id) {
      $("#edit-name").val(item.name);
      $("#edit-id").val(item.id);
```

```
      $("#edit-id").val(item.id);
      $("#edit-isComplete")[0].checked = item.isComplete;
    }
  });
  $("#spoiler").css({ display: "block" });
}

$(".my-form").on("submit", function() {
  const item = {
    name: $("#edit-name").val(),
    isComplete: $("#edit-isComplete").is(":checked"),
    id: $("#edit-id").val()
  };

  $.ajax({
    url: uri + "/" + $("#edit-id").val(),
    type: "PUT",
    accepts: "application/json",
    contentType: "application/json",
    data: JSON.stringify(item),
    success: function(result) {
      getData();
    }
  });

  closeInput();
  return false;
});

function closeInput() {
  $("#spoiler").css({ display: "none" });
}
```

可能需要更改 ASP.NET Core 项目的启动设置，以便对 HTML 页面进行本地测试：

- 打开 Properties\launchSettings.json。
- 删除 `launchUrl` 以便在项目的默认文件 index.html 强制打开应用。

有多种方式可以获取 jQuery。在前面的代码片段中，库是从 CDN 中加载的。

此示例调用 API 的所有 CRUD 方法。以下是 API 调用的说明。

获取待办事项的列表

jQuery ajax 函数将 `GET` 请求发送至 API，这将返回表示待办事项数组的 JSON。如果请求成功，则调用 `success` 回调函数。在该回调中使用待办事项信息更新 DOM。

```
$(document).ready(function() {
  getData();
});

function getData() {
  $.ajax({
    type: "GET",
    url: uri,
    cache: false,
    success: function(data) {
      const tBody = $("#todos");

      $(tBody).empty();

      getCount(data.length);

      $.each(data, function(key, item) {
        const tr = $("<tr></tr>")
          .append(
            $("<td></td>").append(
              $("<input/>", {
                type: "checkbox",
                disabled: true,
                checked: item.isComplete
              })
            )
          )
          .append($("<td></td>").text(item.name))
          .append(
            $("<td></td>").append(
              $("<button>Edit</button>").on("click", function() {
                editItem(item.id);
              })
            )
          )
          .append(
            $("<td></td>").append(
              $("<button>Delete</button>").on("click", function() {
                deleteItem(item.id);
              })
            )
          );

        tr.appendTo(tBody);
      });

      todos = data;
    }
  });
}
```

添加待办事项

Ajax 函数发送 `POST`，请求正文中包含待办事项。将 `accepts` 和 `contentType` 选项设置为 `application/json`，以便指定接收和发送的媒体类型。待办事项使用 JSON.stringify 转换为 JSON。当 API 返回成功状态的代码时，将调用 `getData` 函数来更新 HTML 表。

```
function addItem() {
  const item = {
    name: $("#add-name").val(),
    isComplete: false
  };

  $.ajax({
    type: "POST",
    accepts: "application/json",
    url: uri,
    contentType: "application/json",
    data: JSON.stringify(item),
    error: function(jqXHR, textStatus, errorThrown) {
      alert("Something went wrong!");
    },
    success: function(result) {
      getData();
      $("#add-name").val("");
    }
  });
}
```

更新待办事项

更新待办事项与添加类似。`url` 更改为添加项的唯一标识符，并且 `type` 为 `PUT`。

```
$.ajax({
  url: uri + "/" + $("#edit-id").val(),
  type: "PUT",
  accepts: "application/json",
  contentType: "application/json",
  data: JSON.stringify(item),
  success: function(result) {
    getData();
  }
});
```

删除待办事项

若要删除待办事项，请将 AJAX 调用上的 `type` 设为 `DELETE` 并指定该项在 URL 中的唯一标识符。

```
$.ajax({
  url: uri + "/" + id,
  type: "DELETE",
  success: function(result) {
    getData();
  }
});
```

其他资源

查看或下载本教程的示例代码。请参阅如何下载。

有关更多信息，请参见以下资源：

- 使用 ASP.NET Core 构建 Web API
- 带有 Swagger/OpenAPI 的 ASP.NET Core Web API 帮助页
- ASP.NET Core Razor 页面和 EF Core - 教程系列
- 在 ASP.NET Core 中路由到控制器操作
- ASP.NET Core Web API 中控制器操作的返回类型

- 将 ASP.NET Core 应用部署到 Azure 应用服务
- 托管和部署 ASP.NET Core

后续步骤

在本教程中，你将了解：

- 创建 Web API 项目。
- 添加模型类。
- 创建数据库上下文。
- 注册数据库上下文。
- 添加控制器。
- 添加 CRUD 方法。
- 配置路由和 URL 路径。
- 指定返回值。
- 使用 Postman 调用 Web API。
- 使用 jQuery 调用 Web API。

前进到下一个教程，了解如何生成 API 帮助页：

Swashbuckle 和 ASP.NET Core 入门

使用 ASP.NET Core 和 MongoDB 创建 Web API

2019/1/8 • • Edit Online

作者 Pratik Khandelwal 和 Scott Addie

本教程创建对 MongoDB NoSQL 数据库执行创建、读取、更新和删除 (CRUD) 操作的 Web API。

在本教程中，你将了解：

- 配置 MongoDB
- 创建 MongoDB 数据库
- 定义 MongoDB 集合和架构
- 从 Web API 执行 MongoDB CRUD 操作

查看或下载示例代码（如何下载）

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- .NET Core SDK 2.2 或更高版本
- 已安装"ASP.NET 和 Web 开发"工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- MongoDB

配置 MongoDB

如果使用的是 Windows，MongoDB 将默认安装在 C:\Program Files\MongoDB 中。将 C:\Program Files\MongoDB\Server<version_number>\bin 添加到 `Path` 环境变量。通过此更改可以从开发计算机上的任意位置访问 MongoDB。

使用以下步骤中的 mongo Shell 可以创建数据库、创建集合和存储文档。有关 mongo Shell 命令的详细信息，请参阅使用 mongo Shell。

1. 选择开发计算机上用于存储数据的目录。例如，在 Windows 上为 *C:\BooksData*。创建目录（如果不存在）。mongo Shell 不会创建新目录。

2. 打开命令行界面。运行以下命令以连接到默认端口 27017 上的 MongoDB。请记得将 `<data_directory_path>` 替换为上一步中选择的目录。

```
mongod --dbpath <data_directory_path>
```

3. 打开另一个命令行界面实例。通过运行以下命令来连接到默认测试数据库：

```
mongo
```

4. 在命令行界面中运行下面的命令：

```
use BookstoreDb
```

如果该命令尚不存在，则将创建名为 BookstoreDb 的数据库。如果该数据库存在，则将为事务打开其连接。

5. 使用以下命令创建 `Books` 集合：

```
db.createCollection('Books')
```

显示以下结果：

```
{ "ok" : 1 }
```

6. 使用以下命令定义 `Books` 集合的架构并插入两个文档：

```
db.Books.insertMany([{'Name':'Design Patterns','Price':54.93,'Category':'Computers','Author':'Ralph
Johnson'}, {'Name':'Clean Code','Price':43.15,'Category':'Computers','Author':'Robert C. Martin'}])
```

显示以下结果：

```
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("5bfd996f7b8e48dc15ff215d"),
    ObjectId("5bfd996f7b8e48dc15ff215e")
  ]
}
```

7. 使用以下命令查看数据库中的文档：

```
db.Books.find({}).pretty()
```

显示以下结果：

```
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215d"),
  "Name" : "Design Patterns",
  "Price" : 54.93,
  "Category" : "Computers",
  "Author" : "Ralph Johnson"
}
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215e"),
  "Name" : "Clean Code",
  "Price" : 43.15,
  "Category" : "Computers",
  "Author" : "Robert C. Martin"
}
```

该架构将为每个文档添加类型 `ObjectId` 的自动生成的 `_id` 属性。

数据库可供使用了。你可以开始创建 ASP.NET Core Web API。

创建 ASP.NET Core Web API 项目

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

1. 转到“文件” > “新建” > “项目”。
2. 选择“ASP.NET Core Web 应用程序”，将项目命名为“BooksApi”，然后单击“确定”。
3. 选择“.NET Core”目标框架和“ASP.NET Core 2.1”。选择“API”项目模板，然后单击“确定”：
4. 在“包管理器控制台”窗口中，导航到项目根。运行以下命令以安装适用于 MongoDB 的 .NET 驱动程序：

```
Install-Package MongoDB.Driver -Version 2.7.2
```

添加模型

1. 将 Models 目录添加到项目根。
2. 使用以下代码将 `Book` 类添加到 Models 目录：

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BooksApi.Models
{
    public class Book
    {
        public ObjectId Id { get; set; }

        [BsonElement("Name")]
        public string BookName { get; set; }

        [BsonElement("Price")]
        public decimal Price { get; set; }

        [BsonElement("Category")]
        public string Category { get; set; }

        [BsonElement("Author")]
        public string Author { get; set; }
    }
}
```

在前面的类中，将公共语言运行时 (CLR) 对象映射到 MongoDB 集合时需要 `Id` 属性。类中的其他属性使用 `[BsonElement]` 属性进行修饰。该属性的值表示 MongoDB 集合中的属性名称。

添加 CRUD 操作类

1. 将 Services 目录添加到项目根。
2. 使用以下代码将 `BookService` 类添加到 Services 目录：

```
using System.Collections.Generic;
using System.Linq;
using BooksApi.Models;
using Microsoft.Extensions.Configuration;
using MongoDB.Bson;
using MongoDB.Driver;

namespace BooksApi.Services
{
    public class BookService
    {
        private readonly IMongoCollection<Book> _books;

        public BookService(IConfiguration config)
        {
            var client = new MongoClient(config.GetConnectionString("BookstoreDb"));
            var database = client.GetDatabase("BookstoreDb");
            _books = database.GetCollection<Book>("Books");
        }

        public List<Book> Get()
        {
            return _books.Find(book => true).ToList();
        }

        public Book Get(string id)
        {
            var docId = new ObjectId(id);

            return _books.Find<Book>(book => book.Id == docId).FirstOrDefault();
        }

        public Book Create(Book book)
        {
            _books.InsertOne(book);
            return book;
        }

        public void Update(string id, Book bookIn)
        {
            var docId = new ObjectId(id);

            _books.ReplaceOne(book => book.Id == docId, bookIn);
        }

        public void Remove(Book bookIn)
        {
            _books.DeleteOne(book => book.Id == bookIn.Id);
        }

        public void Remove(ObjectId id)
        {
            _books.DeleteOne(book => book.Id == id);
        }
    }
}
```

3. 将 MongoDB 连接字符串添加到 appsettings.json：

```
{
  "ConnectionStrings": {
    "BookstoreDb": "mongodb://localhost:27017"
  },
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  }
}
```

将在 `BookService` 类构造函数中访问前面的 `BookstoreDb` 属性。

4. 在 `Startup.ConfigureServices` 中，向依赖关系注入系统注册 `BookService` 类：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<BookService>();
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

必须执行前面的服务注册才能支持消费类中的构造函数注入。

`BookService` 类使用以下 `MongoDB.Driver` 成员对数据库执行 CRUD 操作：

- `MongoClient` – 读取执行数据库操作的服务器实例。此类的构造函数提供了 MongoDB 连接字符串：

```
public BookService(IConfiguration config)
{
    var client = new MongoClient(config.GetConnectionString("BookstoreDb"));
    var database = client.GetDatabase("BookstoreDb");
    _books = database.GetCollection<Book>("Books");
}
```

- `IMongoDatabase` – 表示用于执行操作的 Mongo 数据库。本教程在界面上使用泛型 `GetCollection<T>(collection)` 方法来获取对特定集合中的数据的访问。调用此方法后，可以对集合执行 CRUD 操作。在 `GetCollection<T>(collection)` 方法调用中：
  - `collection` 表示集合名称。
  - `T` 表示存储在集合中的 CLR 对象类型。

`GetCollection<T>(collection)` 返回 `MongoCollection` 对象，该对象表示集合。在本教程中，对集合调用以下方法：

- `Find<T>` – 返回集合中与提供的搜索条件匹配的所有文档。
- `InsertOne` – 插入提供的对象作为集合中的新文档。
- `ReplaceOne` – 将与提供的搜索条件匹配的单个文档替换为提供的对象。
- `DeleteOne` – 删除与提供的搜索条件匹配的单个文档。

添加控制器

1. 使用以下代码将 `BooksController` 类添加到 Controllers 目录：

```
using System.Collections.Generic;
using BooksApi.Models;
using BooksApi.Services;
using Microsoft.AspNetCore.Mvc;

namespace BooksApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BooksController : ControllerBase
    {
        private readonly BookService _bookService;

        public BooksController(BookService bookService)
        {
            _bookService = bookService;
        }

        [HttpGet]
        public ActionResult<List<Book>> Get()
        {
            return _bookService.Get();
        }

        [HttpGet("{id:length(24)}", Name = "GetBook")]
        public ActionResult<Book> Get(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            return book;
        }

        [HttpPost]
        public ActionResult<Book> Create(Book book)
        {
            _bookService.Create(book);

            return CreatedAtRoute("GetBook", new { id = book.Id.ToString() }, book);
        }

        [HttpPut("{id:length(24)}")]
        public IActionResult Update(string id, Book bookIn)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            _bookService.Update(id, bookIn);

            return NoContent();
        }

        [HttpDelete("{id:length(24)}")]
        public IActionResult Delete(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
```

```
            if (book == null)
            {
                return NotFound();
            }

            _bookService.Remove(book.Id);

            return NoContent();
        }
    }
}
```

前面的 Web API 控制器：

- 使用 `BookService` 类执行 CRUD 操作。
- 包含操作方法以支持 GET、POST、PUT 和 DELETE HTTP 请求。

2. 生成并运行应用。

3. 在浏览器中导航到 `http://localhost:<port>/api/books`。将显示下面的 JSON 响应：

```
[
  {
    "id":"5bfd996f7b8e48dc15ff215d",
    "bookName":"Design Patterns",
    "price":54.93,
    "category":"Computers",
    "author":"Ralph Johnson"
  },
  {
    "id":"5bfd996f7b8e48dc15ff215e",
    "bookName":"Clean Code",
    "price":43.15,
    "category":"Computers",
    "author":"Robert C. Martin"
  }
]
```

后续步骤

有关构建 ASP.NET Core Web API 的详细信息，请参阅以下资源：

- 使用 ASP.NET Core 构建 Web API
- ASP.NET Core Web API 中控制器操作的返回类型

教程：使用 ASP.NET Core 创建 Razor 页面 Web 应用

2019/1/3 • • Edit Online

此系列教程介绍了生成 Razor Pages Web 应用的基础知识。本系列包括以下教程：

1. 创建 Razor 页面 Web 应用
2. 向 Razor 页面应用添加模型
3. 基架（生成）Razor 页面
4. 使用数据库
5. 更新 Razor 页面
6. 添加搜索
7. 添加新字段
8. 添加验证

结束时，你将有一个应用来显示和管理电影数据库。

在带有 Visual Studio 的 Windows 上使用 ASP.NET Core MVC 创建 Web 应用

2019/1/2 • • Edit Online

本教程介绍具有控制器和视图的 ASP.NET Core MVC Web 开发。如果你是初次接触 ASP.NET Core Web 开发，请考虑本教程的 Razor Pages 版本，此版本提供了更简单的起点。

本系列教程包括以下部分：

1. 入门
2. 添加控制器
3. 添加视图
4. 添加模型
5. 使用 SQL Server LocalDB
6. 控制器方法和视图
7. 添加搜索
8. 添加新字段
9. 添加验证
10. 检查 Details 和 Delete 方法

教程：ASP.NET Core SignalR 入门

2019/1/10 • • Edit Online

本教程介绍生成使用 SignalR 的实时应用的基础知识。您将学习如何：

- 创建 Web 项目。
- 添加 SignalR 客户端库。
- 创建 SignalR 中心。
- 配置项目以使用 SignalR。
- 添加可将消息从任何客户端发送到所有连接客户端的代码。

最终将创建一个正常运行的聊天应用：

查看或下载示例代码（如何下载）。

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

创建 Web 项目

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 从菜单中选择“文件”>“新建项目”。
- 在“新建项目”对话框中，选择“已安装”>“Visual C#”>“Web”>“ASP.NET Core Web 应用”。将项目命名

为“SignalRChat”。

- 选择“Web 应用”，以创建使用 Razor Pages 的项目。
- 选择“.NET Core”目标框架，选择“ASP.NET Core 2.2”，然后单击“确定”。

添加 SignalR 客户端库

`Microsoft.AspNetCore.App` 元包中包括 SignalR 服务器库。JavaScript 客户端库不会自动包含在项目中。对于此教

程，使用库管理器 (LibMan) 从 unpkg 获取客户端库。unpkg 是一个内容分发网络 (CDN)，可以分发在 npm（即 Node.js 包管理器）中找到的任何内容。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 在“解决方案资源管理器”中，右键单击项目，然后选择“添加” > “客户端库”。
- 在“添加客户端库”对话框中，对于“提供程序”，选择“unpkg”。
- 对于“库”，输入 `@aspnet/signalr@1`，然后选择不是预览版的最新版本。

- 选择“选择特定文件”，展开“dist/browser”文件夹，然后选择“signalr.js”和“signalr.min.js”。
- 将“目标位置”设置为 wwwroot/lib/signalr/，然后选择“安装”。

LibMan 创建 wwwroot/lib/signalr 文件夹并将所选文件复制到该文件夹。

创建 SignalR 中心

*中心*是一个类，用作处理客户端 - 服务器通信的高级管道。

- 在 SignalRChat 项目文件夹中，创建 Hubs 文件夹。
- 在 Hubs 文件夹中，使用以下代码创建 ChatHub.cs 文件：

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

`ChatHub` 类继承自 SignalR `Hub` 类。`Hub` 类管理连接、组和消息。

可通过已连接客户端调用 `SendMessage`，以向所有客户端发送消息。本教程后面部分将显示调用该方法的 JavaScript 客户端代码。SignalR 代码是异步模式，可提供最大的可伸缩性。

配置 SignalR

必须配置 SignalR 服务器，以将 SignalR 请求传递到 SignalR。

- 将以下突出显示的代码添加到 Startup.cs 文件。

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using SignalRChat.Hubs;

namespace SignalRChat
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>(options =>
            {
                // This lambda determines whether user consent for non-essential cookies is needed for a
given request.
                options.CheckConsentNeeded = context => true;
                options.MinimumSameSitePolicy = SameSiteMode.None;
            });


            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

            services.AddSignalR();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            app.UseSignalR(routes =>
            {
                routes.MapHub<ChatHub>("/chatHub");
            });
            app.UseMvc();
        }
    }
}
```

这些更改将 SignalR 添加到 ASP.NET Core 依赖关系注入系统和中间件管道。

添加 SignalR 客户端代码

- 使用以下代码替换 Pages\Index.cshtml 中的内容：

```
@page
<div class="container">
    <div class="row"> </div>
    <div class="row">
        <div class="col-6"> </div>
        <div class="col-6">
            User..........<input type="text" id="userInput" />
            <br />
            Message...<input type="text" id="messageInput" />
            <input type="button" id="sendButton" value="Send Message" />
        </div>
    </div>
    <div class="row">
        <div class="col-12">
            <hr />
        </div>
    </div>
    <div class="row">
        <div class="col-6"> </div>
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
</div>
<script src="~/lib/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

  前面的代码：

  - 创建名称以及消息文本的文本框和"提交"按钮。
  - 使用 `id="messagesList"` 创建一个列表，用于显示从 SignalR 中心接收的消息。
  - 包含对 SignalR 的脚本引用以及在下一步中创建的 chat.js 应用程序代码。
- 在 wwwroot/js 文件夹中，使用以下代码创建 chat.js 文件：

```
"use strict";

var connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();

connection.on("ReceiveMessage", function (user, message) {
    var msg = message.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;");
    var encodedMsg = user + " says " + msg;
    var li = document.createElement("li");
    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});

connection.start().catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function (event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

  前面的代码：

- 创建并启动连接。
- 向"提交"按钮添加一个用于向中心发送消息的处理程序。
- 向连接对象添加一个用于从中心接收消息并将其添加到列表的处理程序。

运行应用

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 按 Ctrl+F5 可运行应用而不进行调试。
- 从地址栏复制 URL，打开另一个浏览器实例或选项卡，并在地址栏中粘贴该 URL。
- 选择任一浏览器，输入名称和消息，然后选择"发送消息"按钮。

  两个页面上立即显示名称和消息。

TIP

如果应用不起作用，请打开浏览器开发人员工具 (F12) 并转到控制台。可能会看到与 HTML 和 JavaScript 代码相关的错误。例如，假设将 signalr.js 放在不同于系统指示的文件夹中。在这种情况下，对该文件的引用将不起作用，并且你将在控制台中看到 404 错误。

后续步骤

在本教程中，你将了解：

- 创建 Web 应用项目。
- 添加 SignalR 客户端库。
- 创建 SignalR 中心。
- 配置项目以使用 SignalR。
- 添加代码，以便使用此中心将消息从任何客户端发送到所有连接的客户端。

若要详细了解 SignalR，请参阅简介：

ASP.NET Core SignalR 简介

配合使用 ASP.NET Core SignalR 和 TypeScript 以及 Webpack

2019/1/10 • • Edit Online

作者：Sébastien Sougnez 和 Scott Addie

开发人员可以通过 Webpack 捆绑和生成 Web 应用的客户端资源。本教程介绍在 ASP.NET Core SignalR Web 应用中使用 Webpack，该应用的客户端是使用 TypeScript 编写的。

在本教程中，你将了解：

- 为入门 ASP.NET Core SignalR 应用搭建基架
- 配置 SignalR TypeScript 客户端
- 使用 Webpack 配置生成管道
- 配置 SignalR 服务器
- 启用客户端和服务器之间的通信

查看或下载示例代码（如何下载）

Prerequisites

- Visual Studio
- Visual Studio Code

- 已安装"ASP.NET 和 Web 开发"工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

创建 ASP.NET Core Web 应用

- Visual Studio
- Visual Studio Code

配置 Visual Studio，在 PATH 环境变量中查找 npm。默认情况下，Visual Studio 使用在安装目录中找到的 npm 版本。在 Visual Studio 中按照以下说明执行操作：

1. 导航到"工具" > "选项" > "项目和解决方案" > "Web 包管理" > "外部 Web 工具"。

2. 在列表中选择 $(PATH) 项。单击向上键将项移动列表第二个位置。

已完成 Visual Studio 配置。可以开始创建项目了。

1. 使用"文件" > "新建" > "项目"菜单选项，然后选择"ASP.NET Core Web 应用程序"模板。
2. 将项目命名为 SignalRWebPack 并选择"确定"。
3. 从目标框架下拉列表选择 .NET Core 并从框架选择器下拉列表选择 ASP.NET Core 2.2。选择"空白"模板并选择"确定"。

配置 Webpack 和 TypeScript

以下步骤配置 TypeScript 到 JavaScript 的转换和客户端资源的捆绑。

1. 在项目根目录中执行以下命令，创建 package.json 文件：

```
npm init -y
```

2. 将突出显示的属性添加到 package.json 文件：

```
{
  "name": "SignalRWebPack",
  "version": "1.0.0",
  "private": true,
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

   将 `private` 属性设置为 `true`，防止下一步出现包安装警告。

3. 安装所需的 npm 包。从项目根执行以下命令：

```
npm install -D -E clean-webpack-plugin@0.1.19 css-loader@0.28.11 html-webpack-plugin@3.2.0 mini-css-extract-plugin@0.4.0 ts-loader@4.4.1 typescript@2.9.2 webpack@4.12.0 webpack-cli@3.0.6
```

需要注意的一些命令细节：

- 每个包名称中 `@` 符号后是版本号。npm 安装这些特定的包版本。
- `-E` 选项禁用 npm 将语义化版本控制范围运算符写到 package.json 的默认行为。例如，使用 `"webpack": "4.12.0"` 而不是 `"webpack": "^4.12.0"`。此选项防止意外升级到新的包版本。

有关详细信息，请参阅官方 npm-install 文档。

4. 将 package.json 文件的 `scripts` 属性替换为以下代码片段：

```
"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

脚本的一些解释：

- `build`：在开发模式下捆绑客户端资源并观察文件更改。文件观察程序使捆绑在每次项目文件发生更改时重新生成。`mode` 选项可禁用生产优化，例如摇树优化和缩小优化。仅在开发中使用 `build`。
- `release`：在生产模式下捆绑客户端资源。
- `publish`：运行 `release` 脚本，在生产模式下捆绑客户端资源。它调用 .NET Core CLI 的 publish 命令发布应用。

5. 在项目根中创建名为 webpack.config.js 的文件，包含以下内容：

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const CleanWebpackPlugin = require("clean-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
    entry: "./src/index.ts",
    output: {
        path: path.resolve(__dirname, "wwwroot"),
        filename: "[name].[chunkhash].js",
        publicPath: "/"
    },
    resolve: {
        extensions: [".js", ".ts"]
    },
    module: {
        rules: [
            {
                test: /\.ts$/,
                use: "ts-loader"
            },
            {
                test: /\.css$/,
                use: [MiniCssExtractPlugin.loader, "css-loader"]
            }
        ]
    },
    plugins: [
        new CleanWebpackPlugin(["wwwroot/*"]),
        new HtmlWebpackPlugin({
            template: "./src/index.html"
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css"
        })
    ]
};
```

前面的文件配置 Webpack 编译。需要注意的一些配置细节：

- `output` 属性替代 dist 的默认值。捆绑反而在 wwwroot 目录中发出。
- `resolve.extensions` 数组包含 .js，以便导入 SignalR 客户端 JavaScript。

6. 在项目根中创建新的 src 目录。目的是存储项目的客户端资产。

7. 创建包含以下内容的 src/index.html。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>ASP.NET Core SignalR</title>
</head>
<body>
    <div id="divMessages" class="messages">
    </div>
    <div class="input-zone">
        <label id="lblMessage" for="tbMessage">Message:</label>
        <input id="tbMessage" class="input-zone-input" type="text" />
        <button id="btnSend">Send</button>
    </div>
</body>
</html>
```

前面的 HTML 定义主页的样板标记。

8. 创建新的 src/css 目录。目的是存储项目的 .css 文件。

9. 创建包含以下内容的 src/css/main.css：

```
*, *::before, *::after {
    box-sizing: border-box;
}

html, body {
    margin: 0;
    padding: 0;
}

.input-zone {
    align-items: center;
    display: flex;
    flex-direction: row;
    margin: 10px;
}

.input-zone-input {
    flex: 1;
    margin-right: 10px;
}

.message-author {
    font-weight: bold;
}

.messages {
    border: 1px solid #000;
    margin: 10px;
    max-height: 300px;
    min-height: 300px;
    overflow-y: auto;
    padding: 5px;
}
```

前面的 main.css 文件设计应用样式。

10. 创建包含以下内容的 src/tsconfig.json：

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

前面的代码配置 TypeScript 编译器，生成与 ECMAScript 5 兼容的 JavaScript。

11. 创建包含以下内容的 src/index.ts：

```
import "./css/main.css";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

前面的 TypeScript 检索对 DOM 元素的引用并附加两个事件处理程序：

- `keyup`：用户在文本框中键入标识为 `tbMessage` 的内容时触发此事件。用户按 Enter 时调用 `send` 函数。
- `click`：用户单击“发送”按钮时触发此事件。调用 `send` 函数。

配置 ASP.NET Core 应用

1. `Startup.Configure` 方法中提供的代码显示 Hello World!。将 `app.Run` 方法调用替换为对 UseDefaultFiles 和 UseStaticFiles 的调用。

```
app.UseDefaultFiles();
app.UseStaticFiles();
```

前面的代码允许服务器定位并提供 index.html 文件，无论用户输入完整 URL 还是 Web 应用的根 URL。

2. 在 `Startup.ConfigureServices` 方法中调用 AddSignalR。此操作会将 SignalR 服务添加到项目。

```
services.AddSignalR();
```

3. 将 /hub 路由映射到 `ChatHub` 中心。在 `Startup.Configure` 方法的末尾添加以下行：

```
app.UseSignalR(options =>
{
    options.MapHub<ChatHub>("/hub");
});
```

4. 在项目根中创建名为 Hubs 的新目录。目的是存储 SignalR 中心（在下一步中创建）。

5. 创建包含以下代码的中心 Hubs/ChatHub.cs：

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
    }
}
```

6. 在 Startup.cs 文件顶部添加以下代码，解析 `ChatHub` 引用：

```
using SignalRWebPack.Hubs;
```

启用客户端和服务器通信

应用当前显示一个发送消息的简单窗体。尝试执行此操作时没有任何反应。服务器正在侦听特定的路由，但是不涉及发送消息。

1. 在项目根执行以下命令：

```
npm install @aspnet/signalr
```

前面的命令安装 SignalR TypeScript 客户端，它允许客户端向服务器发送消息。

2. 将突出显示的代码添加到 src/index.ts 文件：

```
import "./css/main.css";
import * as signalR from "@aspnet/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.start().catch(err => document.write(err));

connection.on("messageReceived", (username: string, message: string) => {
    let m = document.createElement("div");

    m.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(m);
    divMessages.scrollTop = divMessages.scrollHeight;
});

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

前面的代码支持从服务器接收消息。`HubConnectionBuilder` 类创建新的生成器，用于配置服务器连接。`withUrl` 函数配置中心 URL。

SignalR 启用客户端和服务器之间的消息交换。每个消息都有特定的名称。例如，名为 `messageReceived` 的消息可以执行负责在消息区域显示新消息的逻辑。可以通过 `on` 函数完成对特定消息的侦听。可以侦听任意数量的消息名称。还可以将参数传递到消息，例如所接收消息的作者姓名和内容。客户端收到一条消息后，会创建一个新的 `div` 元素并在其 `innerHTML` 属性中显示作者姓名和消息内容。它添加到显示消息的主要

`div` 元素。

3. 客户端可以接收消息后，将它配置为发送消息。将突出显示的代码添加到 src/index.ts 文件：

```
import "./css/main.css";
import * as signalR from "@aspnet/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.start().catch(err => document.write(err));

connection.on("messageReceived", (username: string, message: string) => {
    let messageContainer = document.createElement("div");

    messageContainer.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(messageContainer);
    divMessages.scrollTop = divMessages.scrollHeight;
});

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
    connection.send("newMessage", username, tbMessage.value)
              .then(() => tbMessage.value = "");
}
```

通过 WebSockets 连接发送消息需要调用 `send` 方法。该方法的第一个参数是消息名称。消息数据包含其他参数。在此示例中，一条标识为 `newMessage` 的消息已发送到服务器。该消息包含用户名和文本框中的用户输入。如果发送成功，会清空文本框。

4. 将突出显示的方法添加到 `ChatHub` 类：

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
        public async Task NewMessage(string username, string message)
        {
            await Clients.All.SendAsync("messageReceived", username, message);
        }
    }
}
```

服务器收到消息后，前面的代码会将这些消息播发到所有连接的用户。没有必要使用泛型 `on` 方法接收所有消息。使用以消息名称命名的方法就可以了。

在此示例中，TypeScript 客户端发送一条标识为 `newMessage` 的消息。C# `NewMessage` 方法需要客户端发送的数据。在 Clients.All 调用 SendAsync 方法。接收的消息会发送到所有连接到中心的客户端。

测试应用

确认应用遵循以下步骤。

- Visual Studio
- Visual Studio Code

1. 在 release 模式下运行 Webpack。使用"包管理器控制台"窗口，在项目根中运行以下命令。如果不在项目根中，请在输入该命令之前输入 `cd SignalRWebPack`。

   ```
   npm run release
   ```

   此命令在运行应用时生成要提供的客户端资产。资产位于 wwwroot 文件夹。

   Webpack 完成了以下任务：

   - 清除了 wwwroot 目录的内容。
   - 将 TypeScript 转换为 JavaScript，该过程称为"转译"。
   - 破坏生成的 JavaScript 以降低文件大小，该过程称为"缩小"。
   - 将已处理的 JavaScript、CSS 和 HTML 文件从 src 复制到 wwwroot 目录。
   - 将以下元素注入 wwwroot/index.html 文件：
     - 一个引用 wwwroot/main.<hash>.css 文件的 `<link>` 标记。此标记紧挨着 `</head>` 结束标记之前。
     - 一个引用缩小后的 wwwroot/main.<hash>.js 文件的 `<script>` 标记。此标记紧挨着 `</body>` 结束标记之前。
2. 选择"调试" > "开始执行(不调试)"，在不附加调试器的情况下在浏览器中启动应用。在 `http://localhost:<port_number>` 上提供 *wwwroot/index.html* 文件。

3. 打开另一个浏览器实例(任意浏览器)。在地址栏中粘贴 URL。

4. 选择一个浏览器，在"消息"文本框中键入任意内容，然后单击"发送"按钮。两个页面上立即显示唯一的用户名和消息。

其他资源

- ASP.NET Core SignalR JavaScript 客户端
- 在 ASP.NET Core SignalR 使用中心

使用 ASP.NET Core 为本机移动应用创建后端服务

2018/10/1 • • Edit Online

作者：Steve Smith

移动应用可与 ASP.NET Core 后端服务轻松通信。

查看或下载后端服务代码示例

本机移动应用示例

本教程演示如何创建使用 ASP.NET Core MVC 支持本机移动应用的后端服务。它使用 Xamarin Forms ToDoRest 应用 作为其本机客户端，其中包括 Android、iOS、Windows Universal 和 Window Phone 设备的单独本机客户端。你可以遵循链接中的教程来创建本机应用程序（并安装需要的免费 Xamarin 工具），以及下载 Xamarin 示例解决方案。Xamarin 示例包含一个 ASP.NET Web API 2 服务项目，使用本文中的 ASP.NET Core 应用替换（客户端无需进行任何更改）。

功能

ToDoRest 应用支持列出、添加、删除和更新待办事项。每个项都有一个 ID、Name（名称）、Notes（说明）以及一个指示该项是否已完成的属性 Done。

待办事项的主视图如上所示，列出每个项的名称，并使用复选标记指示它是否已完成。

点击 + 图标打开"添加项"对话框：

点击主列表屏幕上的项将打开一个编辑对话框，在其中可以修改项的名称、说明以及是否完成，或删除项目：

此示例默认配置为使用托管在 developer.xamarin.com上的后端服务，允许只读操作。若要使用在你计算机上运行的下一节创建的 ASP.NET Core 应用对其进行测试，你需要更新应用程序的 `RestUrl` 常量。导航到 `ToDoREST` 项目，然后打开 *Constants.cs* 文件。使用包含计算机 IP 的 URL 地址替换 `RestUrl`（不是 localhost 或 127.0.0.1，因为此地址用于从设备模拟器中，而不是从你的计算机中访问）。请包括端口号 (5000)。为了测试你的服务能否在设备上正常运行，请确保没有活动的防火墙阻止访问此端口。

```
// URL of REST service (Xamarin ReadOnly Service)
//public static string RestUrl = "http://developer.xamarin.com:8081/api/todoitems{0}";

// use your machine's IP address
public static string RestUrl = "http://192.168.1.207:5000/api/todoitems/{0}";
```

创建 ASP.NET Core 项目

在 Visual Studio 中创建一个新的 ASP.NET Core Web 应用程序。选择 Web API 模板和 No Authentication（无身份验证）。将项目命名为 *ToDoApi*。

对于向端口 5000 进行的请求，应用程序均需作出响应。更新 Program.cs，使其包含 `.UseUrls("http://*:5000")`，以便实现以下操作：

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

NOTE

请确保直接运行应用程序，而不是在 IIS Express 后运行，因为在默认情况下，后者会忽略非本地请求。从命令提示符处运行 dotnet run，或从 Visual Studio 工具栏中的"调试目标"下拉列表中选择应用程序名称配置文件。

添加一个模型类来表示待办事项。使用 `[Required]` 属性标记必需字段：

```
using System.ComponentModel.DataAnnotations;

namespace ToDoApi.Models
{
    public class ToDoItem
    {
        [Required]
        public string ID { get; set; }

        [Required]
        public string Name { get; set; }

        [Required]
        public string Notes { get; set; }

        public bool Done { get; set; }
    }
}
```

API 方法需要通过某种方式处理数据。使用原始 Xamarin 示例所用的 `IToDoRepository` 接口：

```
using System.Collections.Generic;
using ToDoApi.Models;

namespace ToDoApi.Interfaces
{
    public interface IToDoRepository
    {
        bool DoesItemExist(string id);
        IEnumerable<ToDoItem> All { get; }
        ToDoItem Find(string id);
        void Insert(ToDoItem item);
        void Update(ToDoItem item);
        void Delete(string id);
    }
}
```

在此示例中，该实现仅使用一个专用项集合：

```
using System.Collections.Generic;
using System.Linq;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Services
{
    public class ToDoRepository : IToDoRepository
    {
        private List<ToDoItem> _toDoList;

        public ToDoRepository()
        {
            InitializeData();
        }

        public IEnumerable<ToDoItem> All
        {
            get { return _toDoList; }
        }

        public bool DoesItemExist(string id)
        {
            return _toDoList.Any(item => item.ID == id);
        }
```

```
        public ToDoItem Find(string id)
        {
            return _toDoList.FirstOrDefault(item => item.ID == id);
        }

        public void Insert(ToDoItem item)
        {
            _toDoList.Add(item);
        }

        public void Update(ToDoItem item)
        {
            var todoItem = this.Find(item.ID);
            var index = _toDoList.IndexOf(todoItem);
            _toDoList.RemoveAt(index);
            _toDoList.Insert(index, item);
        }

        public void Delete(string id)
        {
            _toDoList.Remove(this.Find(id));
        }

        private void InitializeData()
        {
            _toDoList = new List<ToDoItem>();

            var todoItem1 = new ToDoItem
            {
                ID = "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
                Name = "Learn app development",
                Notes = "Attend Xamarin University",
                Done = true
            };

            var todoItem2 = new ToDoItem
            {
                ID = "b94afb54-a1cb-4313-8af3-b7511551b33b",
                Name = "Develop apps",
                Notes = "Use Xamarin Studio/Visual Studio",
                Done = false
            };

            var todoItem3 = new ToDoItem
            {
                ID = "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
                Name = "Publish apps",
                Notes = "All app stores",
                Done = false,
            };

            _toDoList.Add(todoItem1);
            _toDoList.Add(todoItem2);
            _toDoList.Add(todoItem3);
        }
    }
}
```

在 *Startup.cs* 中配置该实现:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<IToDoRepository,ToDoRepository>();
}
```

现可创建 ToDoItemsController。

> **TIP**
>
> 有关创建 Web API 的详细信息，请参阅使用 ASP.NET Core MVC 和 Visual Studio 生成首个 Web API。

创建控制器

在项目中添加新控制器 *ToDoItemsController*。它应继承 Microsoft.AspNetCore.Mvc.Controller。添加 `Route` 属性以指示控制器将处理路径以 `api/todoitems` 开始的请求。路由中的 `[controller]` 标记会被控制器的名称代替（省略 `Controller` 后缀），这对全局路由特别有用。详细了解 路由。

控制器需要 `IToDoRepository` 才能正常运行；通过控制器的构造函数请求该类型的实例。在运行时，此实例将使用框架对 依赖关系注入 的支持来提供。

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Controllers
{
    [Route("api/[controller]")]
    public class ToDoItemsController : Controller
    {
        private readonly IToDoRepository _toDoRepository;

        public ToDoItemsController(IToDoRepository toDoRepository)
        {
            _toDoRepository = toDoRepository;
        }
```

此 API 支持四个不同的 HTTP 谓词来执行对数据源的 CRUD（创建、读取、更新、删除）操作。最简单的是读取操作，它对应于 HTTP GET 请求。

读取项目

要请求项列表，可对 `List` 方法使用 GET 请求。`[HttpGet]` 方法的 `List` 属性指示此操作应仅处理 GET 请求。此操作的路由是在控制器上指定的路由。你不一定必须将操作名称用作路由的一部分。你只需确保每个操作都有唯一的和明确的路由。路由属性可以分别应用在控制器和方法级别，以此生成特定的路由。

```
[HttpGet]
public IActionResult List()
{
    return Ok(_toDoRepository.All);
}
```

`List` 方法返回 200 OK 响应代码和所有 ToDo 项，并序列化为 JSON 。

你可以使用多种工具测试新的 API 方法，如 Postman，如此处所示：

创建项目

按照约定，创建新数据项映射到 HTTP POST 谓词。`Create` 方法具有应用于该对象的 `[HttpPost]` 属性，并接受 `ToDoItem` 实例。由于 `item` 参数将在 POST 的正文中传递，因此该参数用 `[FromBody]` 属性修饰。

在该方法中，会检查项的有效性和之前是否存在于数据存储，并且如果没有任何问题，则使用存储库添加。检查 `ModelState.IsValid` 将执行 模型验证，应该在每个接受用户输入的 API 方法中执行此步骤。

```
[HttpPost]
public IActionResult Create([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        bool itemExists = _toDoRepository.DoesItemExist(item.ID);
        if (itemExists)
        {
            return StatusCode(StatusCodes.Status409Conflict, ErrorCode.TodoItemIDInUse.ToString());
        }
        _toDoRepository.Insert(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotCreateItem.ToString());
    }
    return Ok(item);
}
```

示例中使用一个枚举，后者包含传递到移动客户端的错误代码：

```
public enum ErrorCode
{
    TodoItemNameAndNotesRequired,
    TodoItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}
```

使用 Postman 测试添加新项，选择 POST 谓词并在请求正文中以 JSON 格式提供新对象。你还应添加一个请求标头指定 `Content-Type` 为 `application/json`。

该方法返回在响应中新建的项。

更新项目

通过使用 HTTP PUT 请求来修改记录。除了此更改之外，`Edit` 方法几乎与 `Create` 完全相同。请注意，如果未找到记录，则 `Edit` 操作将返回 `NotFound` (404) 响应。

```
[HttpPut]
public IActionResult Edit([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        var existingItem = _toDoRepository.Find(item.ID);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _toDoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}
```

若要使用 Postman 进行测试，将谓词更改为 PUT。在请求正文中指定要更新的对象数据。

为了与预先存在的 API 保持一致，此方法在成功时返回 `NoContent` (204) 响应。

删除项目

删除记录可以通过向服务发出 DELETE 请求并传递要删除项的 ID 来完成。与更新一样，请求的项不存在时会收到 `NotFound` 响应。请求成功会得到 `NoContent` (204) 响应。

```
[HttpDelete("{id}")]
public IActionResult Delete(string id)
{
    try
    {
        var item = _toDoRepository.Find(id);
        if (item == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _toDoRepository.Delete(id);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotDeleteItem.ToString());
    }
    return NoContent();
}
```

请注意，在测试删除功能时，请求正文中不需要任何内容。

常见的 Web API 约定

开发应用程序的后端服务时，你将想要使用一组一致的约定或策略来处理横切关注点。例如，在上面所示服务中，针对不存在的特定记录的请求会收到 `NotFound` 响应，而不是 `BadRequest` 响应。同样，对于此服务，传递模型绑定类型的命令始终检查 `ModelState.IsValid` 并为无效的模型类型返回 `BadRequest`。

一旦为 Api 指定通用策略，一般可以将其封装在 Filter（筛选器）。详细了解 如何封装 ASP.NET Core MVC 应用程序中的通用 API 策略。

其他资源

- 身份验证和授权

ASP.NET Core Razor 页面和 EF Core - 教程系列

2018/7/11 • • Edit Online

下面的一系列教程介绍如何创建使用 Entity Framework (EF) Core 进行数据访问的 ASP.NET Core Razor Pages Web 应用。

1. 入门
2. 创建、读取、更新和删除操作
3. 排序、筛选、分页和分组
4. 迁移
5. 创建复杂数据模型
6. 读取相关数据
7. 更新相关数据
8. 处理并发冲突

ASP.NET Core MVC 和 EF Core - 教程系列

2018/6/21 • • Edit Online

本教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor 页面是 ASP.NET Core 2.0 中的一个新选择，它是基于页面的编程模型，可以实现更简单、更高效地生成 Web UI。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果通过 Razor 页面版本选择本教程，请在此 GitHub 问题中说明原因。

1. 入门
2. 创建、读取、更新和删除操作
3. 排序、筛选、分页和分组
4. 迁移
5. 创建复杂数据模型
6. 读取相关数据
7. 更新相关数据
8. 处理并发冲突
9. 继承
10. 高级主题

ASP.NET Core 基础知识

2019/1/9 • • Edit Online

ASP.NET Core 应用是一个控制台应用，它在其 `Program.Main` 方法中创建 Web 服务器。`Main` 方法是应用的托管入口点：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

.NET Core 主机：

- 加载 .NET Core 运行时。
- 使用第一个命令行参数作为包含入口点（`Main`）的托管二进制文件的路径，并开始执行代码。

`Main` 方法调用 WebHost.CreateDefaultBuilder，后者按照生成器模式来创建 Web 主机。生成器包含定义 Web 服务器（例如，UseKestrel）和启动类（UseStartup）的方法。在前面的例子中，自动分配了 Kestrel Web 服务器。ASP.NET Core 的 Web 主机尝试在 Internet Information Services (IIS)（若有）上运行。对于其他 Web 服务器（如 HTTP.sys），可通过调用相应的扩展方法来使用。启动部分进一步介绍了 `UseStartup`。

IWebHostBuilder 是 `WebHost.CreateDefaultBuilder` 调用的返回类型，它提供了许多可选方法。其中的一些方法包括用于在 HTTP.sys 中托管应用的 `UseHttpSys`，以及用于指定根内容目录的 UseContentRoot。Build 和 Run 方法生成 IWebHost 对象，该对象托管应用并开始侦听 HTTP 请求。

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseStartup<Startup>()
            .Build();

        host.Run();
    }
}
```

.NET Core 主机：

- 加载 .NET Core 运行时。
- 使用第一个命令行参数作为包含入口点（`Main`）的托管二进制文件的路径，并开始执行代码。

`Main` 方法使用 WebHostBuilder，后者按照生成器模式来创建 Web 主机。生成器提供定义 Web 服务器（例如，UseKestrel）和启动类（UseStartup）的方法。在前面的示例中，使用了 Kestrel Web 服务器。对于其他 Web 服务器（如 HTTP.sys），可通过调用相应的扩展方法来使用。启动部分进一步介绍了 `UseStartup`。

`WebHostBuilder` 提供了许多可选方法，其中包括用于在 IIS 和 IIS Express 中进行托管的 UseIISIntegration，以及用于指定根内容目录的 UseContentRoot。Build 和 Run 方法生成 IWebHost 对象，该对象托管应用并开始侦听 HTTP 请求。

启动

`WebHostBuilder` 上的 `UseStartup` 方法为你的应用指定 `Startup` 类：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseStartup<Startup>()
            .Build();

        host.Run();
    }
}
```

`Startup` 类用于在其中配置应用所需的任何服务和定义请求处理管道。`Startup` 类必须是公共类，通常包含以下方法。`Startup.ConfigureServices` 是可选的。

```
public class Startup
{
    // This method gets called by the runtime. Use this method
    // to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
    }

    // This method gets called by the runtime. Use this method
    // to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app)
    {
    }
}
```

```
public class Startup
{
    // This method gets called by the runtime. Use this method
    // to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
    }

    // This method gets called by the runtime. Use this method
    // to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app)
    {
    }
}
```

ConfigureServices 定义应用所使用的服务(如 ASP.NET Core MVC、Entity Framework Core 和标识)。Configure 定义在请求管道中调用的中间件。

有关更多信息，请参见ASP.NET Core 中的应用启动。

内容根

内容根是应用所使用的任何内容的基路径，如 Razor Pages、MVC 视图和静态资产。默认情况下，内容根位置与用于托管应用的可执行文件的应用基路径相同。

Web 根 (webroot)

应用的 webroot 是项目中的目录，其中包含公共资源、CSS 等静态资源、JavaScript 和图形文件。wwwroot 默认为 webroot。

对于 Razor (.cshtml) 文件，波浪号斜杠 `~/` 指向 webroot。以 `~/` 开头的路径称为虚拟路径。

依赖关系注入(服务)

服务是应用中常用的组件。可以通过依存关系注入 (DI) 来获取服务。ASP.NET Core 包括默认支持构造函数注入的本机控制反转 (IoC) 容器。可根据需要替换默认容器。DI 除了具备松散耦合优势以外，还可以使服务(例如日志记录)在整个应用中可用。

有关更多信息，请参见在 ASP.NET Core 依赖注入。

中间件

在 ASP.NET Core 中，使用中间件来撰写请求管道。ASP.NET Core 中间件在 `HttpContext` 上执行异步操作，然后调用管道中的下一个中间件或终止请求。

按照惯例，通过在 `Configure` 方法中调用 `UseXYZ` 扩展方法，向管道添加名为“XYZ”的中间件组件。

ASP.NET Core 包含一组丰富的内置中间件，你也可以编写自己的自定义中间件。ASP.NET Core 应用中支持 .NET 的开放 Web 接口 (OWIN)，它将 Web 应用与 Web 服务器分离。

有关详细信息，请参阅 ASP.NET Core 中间件 和 ASP.NET Core 中 .NET 的开放 Web 接口 (OWIN)。

启动 HTTP 请求

IHttpClientFactory 可访问 HttpClient 实例以发出 HTTP 请求。

有关更多信息，请参见启动 HTTP 请求。

环境

环境(如"开发"环境和"生产"环境)是 ASP.NET Core 的高级概念，可使用环境变量、设置文件和命令行参数进行设置。

有关更多信息，请参见在 ASP.NET Core 中使用多个环境。

宿主

ASP.NET Core 应用可配置和启动一个*主机*，负责应用启动和生存期管理。

有关更多信息，请参见ASP.NET Core 中的 Web 主机和通用主机。

服务器

ASP.NET Core 托管模型不直接侦听请求。托管模型依赖 HTTP 服务器实现将请求转发到应用。

- Windows
- macOS
- Linux

ASP.NET Core 提供以下服务器实现：

- Kestrel 服务器是跨平台托管 Web 服务器。Kestrel 通常使用 IIS 在反向代理配置中运行。在 ASP.NET Core 2.0 或更高版本中，Kestrel 也可作为面向公众的边缘服务器运行，直接向 Internet 公开。
- IIS HTTP 服务器（`IISHttpServer`）是 IIS 的进程内服务器。
- HTTP.sys 服务器是 Windows 上适用于 ASP.NET Core 的 Web 服务器。

- Windows
- macOS
- Linux

ASP.NET Core 提供以下服务器实现：

- Kestrel 服务器是跨平台托管 Web 服务器。Kestrel 通常使用 IIS 在反向代理配置中运行。在 ASP.NET Core 2.0 或更高版本中，Kestrel 也可作为面向公众的边缘服务器运行，直接向 Internet 公开。
- HTTP.sys 服务器是 Windows 上适用于 ASP.NET Core 的 Web 服务器。

有关更多信息，请参见ASP.NET Core 中的 Web 服务器实现。

配置

ASP.NET Core 基于名称/值对使用配置模型。配置模型不基于 System.Configuration 或 *web.config*。配置从一组有序的配置提供程序获取设置。内置配置提供程序支持各种文件格式(XML、JSON、INI)、环境变量和命令行参数。也可以编写你自己的自定义配置提供程序。

有关更多信息，请参见ASP.NET Core 中的配置。

日志记录

ASP.NET Core 支持适用于各种日志记录提供程序的日志记录 API。内置提供程序支持向一个或多个目标发送日志。可使用第三方记录框架。

有关更多信息，请参见ASP.NET Core 中的日志记录。

错误处理

ASP.NET Core 的内置方案可处理应用中的错误，包括开发人员异常页、自定义错误页、静态状态代码页和启动异常处理。

有关更多信息，请参见处理 ASP.NET Core 中的错误。

路由

ASP.NET Core 提供将应用请求路由到路由处理程序的方案。

有关更多信息，请参见ASP.NET Core 中的路由。

后台任务

后台任务作为*托管服*务实现。托管服务是一个类，具有实现 IHostedService 接口的后台任务逻辑。

有关更多信息，请参见在 ASP.NET Core 中使用托管服务实现后台任务。

访问 HttpContext

用 Razor Pages 和 MVC 处理请求时，`HttpContext` 自动可用。当 `HttpContext` 不可立即使用时，可通过 IHttpContextAccessor 接口及其默认实现 HttpContextAccessor 访问 `HttpContext`。

有关更多信息，请参见在 ASP.NET Core 中访问 HttpContext。

ASP.NET Core 中的应用启动

2018/11/5 • • Edit Online

作者：Steve Smith、Tom Dykstra 和 Luke Latham

`Startup` 类配置服务和应用的请求管道。

查看或下载示例代码（如何下载）。

Startup 类

ASP.NET Core 应用使用 `Startup` 类，按照约定命名为 `Startup`。`Startup` 类：

- 可选择性地包括 ConfigureServices 方法以配置应用的服务。
- 必须包括 Configure 方法以创建应用的请求处理管道。

当应用启动时，运行时调用 `ConfigureServices` 和 `Configure`：

```
public class Startup
{
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        ...
    }

    // Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app)
    {
        ...
    }
}
```

通过 WebHostBuilderExtensions、UseStartup<TStartup> 方法指定 `Startup` 类：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Web 主机提供 `Startup` 类构造函数可用的某些服务。应用通过 `ConfigureServices` 添加其他服务。然后，主机和应用服务都可以在 `Configure` 和整个应用中使用。

在 `Startup` 类中注入依赖关系的常见用途为注入：

- IHostingEnvironment 以按环境配置服务。
- IConfiguration 以读取配置。
- ILoggerFactory 以在 `Startup.ConfigureServices` 中创建记录器。

```
public class Startup
{
    private readonly IHostingEnvironment _env;
    private readonly IConfiguration _config;
    private readonly ILoggerFactory _loggerFactory;

    public Startup(IHostingEnvironment env, IConfiguration config,
        ILoggerFactory loggerFactory)
    {
        _env = env;
        _config = config;
        _loggerFactory = loggerFactory;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        var logger = _loggerFactory.CreateLogger<Startup>();

        if (_env.IsDevelopment())
        {
            // Development service configuration

            logger.LogInformation("Development environment");
        }
        else
        {
            // Non-development service configuration

            logger.LogInformation($"Environment: {_env.EnvironmentName}");
        }

        // Configuration is available during startup.
        // Examples:
        //   _config["key"]
        //   _config["subsection:suboption1"]
    }
}
```

注入 `IHostingEnvironment` 的替代方法是使用基于约定的方法。应用为不同的环境(例如，`StartupDevelopment`)单独定义 `Startup` 类时，相应的 `Startup` 类会在运行时被选中。优先考虑名称后缀与当前环境相匹配的类。如果应用在开发环境中运行并包含 `Startup` 类和 `StartupDevelopment` 类，则使用 `StartupDevelopment` 类。有关详细信息，请参阅使用多个环境。

若要详细了解 `WebHostBuilder`，请参阅承载主题。有关在启动过程中处理错误的信息，请参阅启动异常处理。

ConfigureServices 方法

ConfigureServices 方法是：

- Optional
- 在 `Configure` 方法配置应用服务之前，由 Web 主机调用。
- 其中按常规设置配置选项。

典型模式是调用所有 `Add{Service}` 方法，然后调用所有 `services.Configure{Service}` 方法。有关示例，请参阅配置标识服务。

将服务添加到服务容器，使其在应用和 `Configure` 方法中可用。这些服务通过依赖关系注入或 IApplicationBuilder.ApplicationServices 解析。

Web 主机可能会在调用 `Startup` 方法之前配置某些服务。有关详细信息，请参阅在 ASP.NET Core 中托管主题。

对于需要大量设置的功能，IServiceCollection 上有 `Add[Service]` 扩展方法。典型 ASP.NET Core 应用将为实体框架、标识和 MVC 注册服务：

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

Configure 方法

Configure 方法用于指定应用响应 HTTP 请求的方式。可通过将中间件组件添加到 IApplicationBuilder 实例来配置请求管道。`Configure` 方法可使用 `IApplicationBuilder`，但未在服务容器中注册。托管创建 `IApplicationBuilder` 并将其直接传递到 `Configure`。

ASP.NET Core 模板配置支持开发人员异常页、BrowserLink、错误页、静态文件和 ASP.NET Core MVC 的管道：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller}/{action=Index}/{id?}");
    });
}
```

每个 `Use` 扩展方法将中间件组件添加到请求管道。例如，`UseMvc` 扩展方法将路由中间件添加到请求管道，并将 MVC 配置为默认处理程序。

请求管道中的每个中间件组件负责调用管道中的下一个组件，或在适当情况下使链发生短路。如果中间件链中未发生短路，则每个中间件都有第二次机会在将请求发送到客户端前处理该请求。

其他服务（如 `IHostingEnvironment` 和 `ILoggerFactory`），也可以在方法签名中指定。如果指定，其他服务如果可用，将被注入。

有关如何使用 `IApplicationBuilder` 和中间件处理顺序的详细信息，请参阅中间件。

便利方法

可使用 ConfigureServices 和 Configure 便利方法，而不是指定 `Startup` 类。多次调用 `ConfigureServices` 将追加到另一个。多次调用 `Configure` 将使用上一个方法调用。

```
public class Program
{
    public static IHostingEnvironment HostingEnvironment { get; set; }
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                HostingEnvironment = hostingContext.HostingEnvironment;
                Configuration = config.Build();
            })
            .ConfigureServices(services =>
            {
                services.AddMvc();
            })
            .Configure(app =>
            {
                var loggerFactory = app.ApplicationServices
                    .GetRequiredService<ILoggerFactory>();
                var logger = loggerFactory.CreateLogger<Program>();
                logger.LogInformation("Logged in Configure");

                if (HostingEnvironment.IsDevelopment())
                {
                    app.UseDeveloperExceptionPage();
                }
                else
                {
                    app.UseExceptionHandler("/Error");
                }

                // Configuration is available during startup. Examples:
                // Configuration["key"]
                // Configuration["subsection:suboption1"]

                app.UseMvcWithDefaultRoute();
                app.UseStaticFiles();
            });
}
```

使用 Startup 筛选器扩展 Startup

在应用的 Configure 中间件管道的开头或末尾使用 IStartupFilter 来配置中间件。`IStartupFilter` 有助于确保中间件在应用请求处理管道的开始或结束时由库添加的中间件之前或之后运行。

`IStartupFilter` 实现单个方法(即 Configure)，该方法接收并返回 `Action<IApplicationBuilder>`。IApplicationBuilder 定义用于配置应用请求管道的类。有关详细信息，请参阅使用 IApplicationBuilder 创建中间件管道。

在请求管道中，每个 `IStartupFilter` 实现一个或多个中间件。筛选器按照添加到服务容器的顺序调用。筛选器可在将控件传递给下一个筛选器之前或之后添加中间件，从而附加到应用管道的开头或末尾。

示例应用演示如何使用 `IStartupFilter` 注册中间件。示例应用包含一个中间件，该中间件从查询字符串参数中设置选项值：

```
public class RequestSetOptionsMiddleware
{
    private readonly RequestDelegate _next;
    private IOptions<AppOptions> _injectedOptions;

    public RequestSetOptionsMiddleware(
        RequestDelegate next, IOptions<AppOptions> injectedOptions)
    {
        _next = next;
        _injectedOptions = injectedOptions;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        Console.WriteLine("RequestSetOptionsMiddleware.Invoke");

        var option = httpContext.Request.Query["option"];

        if (!string.IsNullOrWhiteSpace(option))
        {
            _injectedOptions.Value.Option = WebUtility.HtmlEncode(option);
        }

        await _next(httpContext);
    }
}
```

在 `RequestSetOptionsStartupFilter` 类中配置 `RequestSetOptionsMiddleware`：

```
public class RequestSetOptionsStartupFilter : IStartupFilter
{
    public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next)
    {
        return builder =>
        {
            builder.UseMiddleware<RequestSetOptionsMiddleware>();
            next(builder);
        };
    }
}
```

`IStartupFilter` 在 IWebHostBuilder.ConfigureServices 中的服务容器中注册，以演示 Startup 是如何从 `Startup` 类外部筛选参数 `Startup` 的：

```
WebHost.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddTransient<IStartupFilter,
            RequestSetOptionsStartupFilter>();
    })
    .UseStartup<Startup>()
    .Build();
```

当提供 `option` 的查询字符串参数时，中间件在 MVC 中间件呈现响应之前处理分配值：

中间件执行顺序由 `IStartupFilter` 注册顺序设置：

- 多个 `IStartupFilter` 实现可能与相同的对象进行交互。如果顺序很重要，请将它们的 `IStartupFilter` 服务注册进行排序，以匹配其中间件应有的运行顺序。
- 库可能添加包含一个或多个 `IStartupFilter` 实现的中间件，这些实现在向 `IStartupFilter` 注册的其他应用中间件之前或之后运行。若要在库的 `IStartupFilter` 添加中间件之前调用 `IStartupFilter` 中间件，请在将库添加到服务容器之前定位服务注册。若要在此后调用，请在添加库之后定位服务注册。

在启动时从外部程序集添加配置

通过 IHostingStartup 实现，可在启动时从应用 `Startup` 类之外的外部程序集向应用添加增强功能。有关详细信息，请参阅从外部程序集增强应用。

其他资源

- ASP.NET Core 中的 Web 主机和通用主机
- 在 ASP.NET Core 中使用多个环境
- ASP.NET Core 中间件
- ASP.NET Core 中的日志记录
- ASP.NET Core 中的配置

在 ASP.NET Core 依赖注入

2018/11/10 • • Edit Online

作者：Steve Smith、Scott Addie 和 Luke Latham

ASP.NET Core 支持依赖关系注入 (DI) 软件设计模式，这是一种在类及其依赖关系之间实现控制反转 (IoC) 的技术。

有关特定于 MVC 控制器中依赖关系注入的详细信息，请参阅 在 ASP.NET Core 中将依赖项注入到控制器。

查看或下载示例代码（如何下载）

依赖关系注入概述

依赖项是另一个对象所需的任何对象。使用应用中其他类所依赖的 `WriteMessage` 方法检查以下 `MyDependency` 类：

```
public class MyDependency
{
    public MyDependency()
    {
    }

    public Task WriteMessage(string message)
    {
        Console.WriteLine(
            $"MyDependency.WriteMessage called. Message: {message}");

        return Task.FromResult(0);
    }
}
```

可以创建 `MyDependency` 类的实例以使 `WriteMessage` 方法可用于类。`MyDependency` 类是 `IndexModel` 类的依赖项：

```
public class IndexModel : PageModel
{
    MyDependency _dependency = new MyDependency();

    public async Task OnGetAsync()
    {
        await _dependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

可以创建 `MyDependency` 类的实例以使 `WriteMessage` 方法可用于类。`MyDependency` 类是 `HomeController` 类的依赖项：

```
public class HomeController : Controller
{
    MyDependency _dependency = new MyDependency();

    public async Task<IActionResult> Index()
    {
        await _dependency.WriteMessage(
            "HomeController.Index created this message.");

        return View();
    }
}
```

该类创建并直接依赖于 `MyDependency` 实例。代码依赖关系(如前面的示例)存在问题，应该避免使用，原因如下：

- 要用不同的实现替换 `MyDependency`，必须修改类。
- 如果 `MyDependency` 具有依赖关系，则必须由类对其进行配置。在具有多个依赖于 `MyDependency` 的类的大型项目中，配置代码在整个应用中会变得分散。
- 这种实现很难进行单元测试。应用应使用模拟或存根 `MyDependency` 类，该类不能使用此方法。

依赖关系注入通过以下方式解决了这些问题：

- 使用接口抽象化依赖关系实现。
- 注册服务容器中的依赖关系。ASP.NET Core 提供了一个内置的服务容器 IServiceProvider。服务已在应用的 `Startup.ConfigureServices` 方法中注册。
- 将服务注入到使用它的类的构造函数中。框架负责创建依赖关系的实例，并在不再需要时对其进行处理。

在示例应用中，`IMyDependency` 接口定义了服务为应用提供的方法：

```
public interface IMyDependency
{
    Task WriteMessage(string message);
}
```


```
public interface IMyDependency
{
    Task WriteMessage(string message);
}
```


此接口由具体类型 `MyDependency` 实现：

```
public class MyDependency : IMyDependency
{
    private readonly ILogger<MyDependency> _logger;

    public MyDependency(ILogger<MyDependency> logger)
    {
        _logger = logger;
    }

    public Task WriteMessage(string message)
    {
        _logger.LogInformation(
            "MyDependency.WriteMessage called. Message: {MESSAGE}",
            message);

        return Task.FromResult(0);
    }
}
```


```
public class MyDependency : IMyDependency
{
    private readonly ILogger<MyDependency> _logger;

    public MyDependency(ILogger<MyDependency> logger)
    {
        _logger = logger;
    }

    public Task WriteMessage(string message)
    {
        _logger.LogInformation(
            "MyDependency.WriteMessage called. Message: {MESSAGE}",
            message);

        return Task.FromResult(0);
    }
}
```


`MyDependency` 在其构造函数中请求 ILogger<TCategoryName>。以链式方式使用依赖关系注入并不罕见。每个请求的依赖关系相应地请求其自己的依赖关系。容器解析图中的依赖关系并返回完全解析的服务。必须被解析的依赖关系的集合通常被称为"依赖关系树"、"依赖关系图"或"对象图"。

必须在服务容器中注册 `IMyDependency` 和 `ILogger<TCategoryName>`。`IMyDependency` 已在 `Startup.ConfigureServices` 中注册。`ILogger<TCategoryName>` 由日志记录抽象基础结构注册，因此它是框架默认注册的框架提供的服务。

在示例应用中，使用具体类型 `MyDependency` 注册 `IMyDependency` 服务。注册将服务生存期的范围限定为单个请求的生存期。本主题后面将介绍服务生存期。

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

> **NOTE**
>
> 每个 `services.Add{SERVICE_NAME}` 扩展方法添加(并可能配置)服务。例如，`services.AddMvc()` 添加 Razor Pages 和 MVC 需要的服务。我们建议应用遵循此约定。将扩展方法置于 Microsoft.Extensions.DependencyInjection 命名空间中以封装服务注册的组。

如果服务的构造函数需要基元(如 `string`)，则可以使用配置或选项模式注入基元：

```
public class MyDependency : IMyDependency
{
    public MyDependency(IConfiguration config)
    {
        var myStringValue = config["MyStringKey"];

        // Use myStringValue
    }

    ...
}
```

通过使用服务并分配给私有字段的类的构造函数请求服务的实例。该字段用于在整个类中根据需要访问服务。

在示例应用中，请求 `IMyDependency` 实例并用于调用服务的 `WriteMessage` 方法：

```
public class IndexModel : PageModel
{
    private readonly IMyDependency _myDependency;

    public IndexModel(
        IMyDependency myDependency,
        OperationService operationService,
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance singletonInstanceOperation)
    {
        _myDependency = myDependency;
        OperationService = operationService;
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = singletonInstanceOperation;
    }

    public OperationService OperationService { get; }
    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }

    public async Task OnGetAsync()
    {
        await _myDependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

```
public class MyDependencyController : Controller
{
    private readonly IMyDependency _myDependency;

    public MyDependencyController(IMyDependency myDependency)
    {
        _myDependency = myDependency;
    }

    // GET: /mydependency/
    public async Task<IActionResult> Index()
    {
        await _myDependency.WriteMessage(
            "MyDependencyController.Index created this message.");

        return View();
    }
}
```

框架提供的服务

`Startup.ConfigureServices` 方法负责定义应用使用的服务，包括 Entity Framework Core 和 ASP.NET Core MVC 等平台功能。最初，提供给 `ConfigureServices` 的 `IServiceCollection` 定义了以下服务（具体取决于配置主机的方式）：

服务类型	生存期
Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory	暂时
Microsoft.AspNetCore.Hosting.IApplicationLifetime	单例
Microsoft.AspNetCore.Hosting.IHostingEnvironment	单例
Microsoft.AspNetCore.Hosting.IStartup	单例
Microsoft.AspNetCore.Hosting.IStartupFilter	暂时
Microsoft.AspNetCore.Hosting.Server.IServer	单例
Microsoft.AspNetCore.Http.IHttpContextFactory	暂时
Microsoft.Extensions.Logging.ILogger<T>	单例
Microsoft.Extensions.Logging.ILoggerFactory	单例
Microsoft.Extensions.ObjectPool.ObjectPoolProvider	单一实例
Microsoft.Extensions.Options.IConfigureOptions<T>	暂时
Microsoft.Extensions.Options.IOptions<T>	单一实例
System.Diagnostics.DiagnosticSource	单一实例
System.Diagnostics.DiagnosticListener	单例

当服务集合扩展方法可用于注册服务(及其依赖服务，如果需要)时，约定使用单个 `Add{SERVICE_NAME}` 扩展方法来注册该服务所需的所有服务。以下代码是如何使用扩展方法 AddDbContext、AddIdentity 和 AddMvc 向容器添加其他服务的示例：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();
}
```

有关详细信息，请参阅 API 文档中的 ServiceCollection 类。

服务生存期

为每个注册的服务选择适当的生存期。可以使用以下生存期配置 ASP.NET Core 服务：

暂时

暂时生存期服务是每次请求时创建的。这种生存期适合轻量级、无状态的服务。

作用域（Scoped）

作用域生存期服务以每个请求一次的方式创建。

> **WARNING**
> 在中间件内使用有作用域的服务时，请将该服务注入至 `Invoke` 或 `InvokeAsync` 方法。请不要通过构造函数注入进行注入，因为它会强制服务的行为与单一实例类似。有关更多信息，请参见ASP.NET Core 中间件。

单例

单一实例生存期服务是在第一次请求时（或者在运行 `ConfigureServices` 并且使用服务注册指定实例时）创建的。每个后续请求都使用相同的实例。如果应用需要单一实例行为，建议允许服务容器管理服务的生存期。不要实现单一实例设计模式并提供用户代码来管理对象在类中的生存期。

> **WARNING**
> 从单一实例解析有作用域的服务很危险。当处理后续请求时，它可能会导致服务处于不正确的状态。

构造函数注入行为

服务可以通过两种机制来解析：

- `IServiceProvider`
- ActivatorUtilities – 允许在依赖关系注入容器中创建没有服务注册的对象。`ActivatorUtilities` 用于面向用户的抽象，例如标记帮助器、MVC 控制器和模型绑定器。

构造函数可以接受依赖关系注入不提供的参数，但参数必须分配默认值。

当服务由 `IServiceProvider` 或 `ActivatorUtilities` 解析时，构造函数注入需要 *public* 构造函数。

当服务由 `ActivatorUtilities` 解析时，构造函数注入要求只存在一个适用的构造函数。支持构造函数重载，但其参数可以全部通过依赖注入来实现的重载只能存在一个。

实体框架上下文

应使用有作用域的生存期将实体框架上下文添加到服务容器中。在注册数据库上下文时，通过调用 AddDbContext 方法自动处理。使用数据库上下文的服务也应使用有作用域的生存期。

生存期和注册选项

为了演示生存期和注册选项之间的差异，请考虑以下接口，将任务表示为具有唯一标识符 `OperationId` 的操作。根据为以下接口配置操作服务的生存期的方式，容器在类请求时提供相同或不同的服务实例：

```
public interface IOperation
{
    Guid OperationId { get; }
}

public interface IOperationTransient : IOperation
{
}

public interface IOperationScoped : IOperation
{
}

public interface IOperationSingleton : IOperation
{
}

public interface IOperationSingletonInstance : IOperation
{
}
```


```
public interface IOperation
{
    Guid OperationId { get; }
}

public interface IOperationTransient : IOperation
{
}

public interface IOperationScoped : IOperation
{
}

public interface IOperationSingleton : IOperation
{
}

public interface IOperationSingletonInstance : IOperation
{
}
```


接口在 `Operation` 类中实现。`Operation` 构造函数将生成一个 GUID（如果未提供）：

```
public class Operation : IOperationTransient,
    IOperationScoped,
    IOperationSingleton,
    IOperationSingletonInstance
{
    public Operation() : this(Guid.NewGuid())
    {
    }

    public Operation(Guid id)
    {
        OperationId = id;
    }

    public Guid OperationId { get; private set; }
}
```

```
public class Operation : IOperationTransient,
    IOperationScoped,
    IOperationSingleton,
    IOperationSingletonInstance
{
    public Operation() : this(Guid.NewGuid())
    {
    }

    public Operation(Guid id)
    {
        OperationId = id;
    }

    public Guid OperationId { get; private set; }
}
```

注册 `OperationService` 取决于，每个其他 `Operation` 类型。当通过依赖关系注入请求 `OperationService` 时，它将接收每个服务的新实例或基于从属服务的生存期的现有实例。

- 如果在请求时创建了临时服务，则 `IOperationTransient` 服务的 `OperationId` 与 `OperationService` 的 `OperationId` 不同。`OperationService` 将接收 `IOperationTransient` 类的新实例。新实例将生成一个不同的 `OperationId`。
- 如果按请求创建有作用域的服务，则 `IOperationScoped` 服务的 `OperationId` 与请求中 `OperationService` 的该 ID 相同。在请求中，两个服务共享不同的 `OperationId` 值。
- 如果单一数据库和单一实例服务只创建一次并在所有请求和所有服务中使用，则 `OperationId` 在所有服务请求中保持不变。

```
public class OperationService
{
    public OperationService(
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance instanceOperation)
    {
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = instanceOperation;
    }

    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }
}
```

```
public class OperationService
{
    public OperationService(IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance instanceOperation)
    {
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = instanceOperation;
    }

    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }
}
```

在 `Startup.ConfigureServices` 中，根据其指定的生存期，将每个类型添加到容器中：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

`IOperationSingletonInstance` 服务正在使用已知 ID 为 `Guid.Empty` 的特定实例。此类型在使用时很明显（其 GUID 全部为零）。

示例应用演示了各个请求中和之间的对象生存期。示例应用的 `IndexModel` 请求每种 `IOperation` 类型和 `OperationService`。然后，页面通过属性分配显示所有页面模型类和服务的 `OperationId` 值：

```
public class IndexModel : PageModel
{
    private readonly IMyDependency _myDependency;

    public IndexModel(
        IMyDependency myDependency,
        OperationService operationService,
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance singletonInstanceOperation)
    {
        _myDependency = myDependency;
        OperationService = operationService;
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = singletonInstanceOperation;
    }

    public OperationService OperationService { get; }
    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }

    public async Task OnGetAsync()
    {
        await _myDependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

示例应用演示了各个请求中和之间的对象生存期。示例应用包含一个 `OperationsController`，它会请求各种 `IOperation` 类型和 `OperationService`。`Index` 操作将服务设置为 `ViewBag` 以显示服务的 `OperationId` 值：

```
public class OperationsController : Controller
{
    private readonly OperationService _operationService;
    private readonly IOperationTransient _transientOperation;
    private readonly IOperationScoped _scopedOperation;
    private readonly IOperationSingleton _singletonOperation;
    private readonly IOperationSingletonInstance _singletonInstanceOperation;

    public OperationsController(
        OperationService operationService,
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance singletonInstanceOperation)
    {
        _operationService = operationService;
        _transientOperation = transientOperation;
        _scopedOperation = scopedOperation;
        _singletonOperation = singletonOperation;
        _singletonInstanceOperation = singletonInstanceOperation;
    }

    public IActionResult Index()
    {
        // Viewbag contains controller-requested services.
        ViewBag.Transient = _transientOperation;
        ViewBag.Scoped = _scopedOperation;
        ViewBag.Singleton = _singletonOperation;
        ViewBag.SingletonInstance = _singletonInstanceOperation;

        // Operation service has its own requested services.
        ViewBag.Service = _operationService;

        return View();
    }
}
```

以下两个输出显示了两个请求的结果：

第一个请求：

控制器操作：

暂时性：d233e165-f417-469b-a866-1cf1935d2518
有作用域：5d997e2d-55f5-4a64-8388-51c4e3a1ad19
单一实例：01271bc1-9e31-48e7-8f7c-7261b040ded9
实例：00000000-0000-0000-0000-000000000000

`OperationService` 操作：

暂时性：c6b049eb-1318-4e31-90f1-eb2dd849ff64
有作用域：5d997e2d-55f5-4a64-8388-51c4e3a1ad19
单一实例：01271bc1-9e31-48e7-8f7c-7261b040ded9
实例：00000000-0000-0000-0000-000000000000

第二个请求：

控制器操作：

暂时性：b63bd538-0a37-4ff1-90ba-081c5138dda0
作用域：31e820c5-4834-4d22-83fc-a60118acb9f4
单一实例：01271bc1-9e31-48e7-8f7c-7261b040ded9
实例：00000000-0000-0000-0000-000000000000

`OperationService` 操作：

暂时性：c4cbacb8-36a2-436d-81c8-8c1b78808aaf
作用域：31e820c5-4834-4d22-83fc-a60118acb9f4
单一实例：01271bc1-9e31-48e7-8f7c-7261b040ded9
实例：00000000-0000-0000-0000-000000000000

观察哪个 `OperationId` 值会在一个请求之内和不同请求之间变化：

- 暂时性对象始终不同。请注意，第一个和第二个请求的暂时性 `OperationId` 值对于 `OperationService` 操作和在请求内都是不同的。为每个服务和请求提供了一个新实例。
- 有作用域的对象在一个请求内是相同的，但在请求之间是不同的。
- 单一实例对象对每个对象和每个请求都是相同的（不管 `ConfigureServices` 中是否提供 `Operation` 实例）。

从 main 调用服务

采用 IServiceScopeFactory.CreateScope 创建 IServiceScope，以解析应用作用域中有作用域的服务。此方法可以用于在启动时访问有作用域的服务以便运行初始化任务。以下示例演示如何在 `Program.Main` 中获取 `MyScopedService` 的上下文：

```
public static void Main(string[] args)
{
    var host = CreateWebHostBuilder(args).Build();

    using (var serviceScope = host.Services.CreateScope())
    {
        var services = serviceScope.ServiceProvider;

        try
        {
            var serviceContext = services.GetRequiredService<MyScopedService>();
            // Use the context here
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred.");
        }
    }

    host.Run();
}
```

作用域验证

如果在开发环境中运行应用，默认的服务提供程序会执行检查，从而确认以下内容：

- 没有从根服务提供程序直接或间接解析到有作用域的服务。
- 未将有作用域的服务直接或间接注入到单一实例。

调用 BuildServiceProvider 时，会创建根服务提供程序。在启动提供程序和应用时，根服务提供程序的生存期对应于应用/服务的生存期，并在关闭应用时释放。

有作用域的服务由创建它们的容器释放。如果作用域创建于根容器，则该服务的生存会有效地提升至单一实例，因为根容器只会在应用/服务关闭时将其释放。验证服务作用域，将在调用 `BuildServiceProvider` 时收集这类情况。

有关更多信息，请参见ASP.NET Core Web 主机。

请求服务

来自 `HttpContext` 的 ASP.NET Core 请求中可用的服务通过 HttpContext.RequestServices 集合公开。

请求服务表示作为应用的一部分配置和请求的服务。当对象指定依赖关系时，`RequestServices`（而不是 `ApplicationServices`）中的类型将满足这些要求。

通常，应用不应直接使用这些属性。相反，通过类构造函数请求类所需的类型，并允许框架注入依赖关系。这样生成的类更易于测试。

> **NOTE**
>
> 与访问 `RequestServices` 集合相比，以构造函数参数的形式请求依赖项是更优先的选择。

设计能够进行依赖关系注入的服务

最佳做法是：

- 设计服务以使用依赖关系注入来获取其依赖关系。
- 避免有状态的静态方法调用（称为静态粘附的实践）。
- 避免在服务中直接实例化依赖类。直接实例化将代码耦合到特定实现。

遵循面向对象设计的 SOLID 原则，往往会获得构造良好且易于测试的小型类。

如果一个类似乎有过多的注入依赖关系，这通常表明该类拥有过多的责任并且违反了单一责任原则 (SRP)。尝试通过将某些职责移动到一个新类来重构类。请记住，Razor Pages 页模型类和 MVC 控制器类应关注用户界面问题。业务规则和数据访问实现细节应保留在适用于这些分离的关注点的类中。

服务处理

容器为其创建的 `IDisposable` 类型调用 `Dispose`。如果通过用户代码将实例添加到容器中，则不会自动处理该实例。

```
// Services that implement IDisposable:
public class Service1 : IDisposable {}
public class Service2 : IDisposable {}
public class Service3 : IDisposable {}

public interface ISomeService {}
public class SomeServiceImplementation : ISomeService, IDisposable {}

public void ConfigureServices(IServiceCollection services)
{
    // The container creates the following instances and disposes them automatically:
    services.AddScoped<Service1>();
    services.AddSingleton<Service2>();
    services.AddSingleton<ISomeService>(sp => new SomeServiceImplementation());

    // The container doesn't create the following instances, so it doesn't dispose of
    // the instances automatically:
    services.AddSingleton<Service3>(new Service3());
    services.AddSingleton(new Service3());
}
```

> **NOTE**
> 在 ASP.NET Core 1.0 中，容器将对所有 `IDisposable` 对象调用 dispose，包括那些并非由它创建的对象。

默认服务容器替换

内置的服务容器旨在满足框架和大多数消费者应用的需求。我们建议使用内置容器，除非你需要的特定功能不受它支持。内置容器中找不到第三方容器支持的某些功能：

- 属性注入
- 基于名称的注入
- 子容器
- 自定义生存期管理
- 对迟缓初始化的 `Func<T>` 支持

有关支持适配器的部分容器列表，请参阅依赖关系注入 readme.md 文件。

以下示例将内置容器替换为 Autofac：

- 安装适当的容器包：
  - Autofac
  - Autofac.Extensions.DependencyInjection
- 在 `Startup.ConfigureServices` 中配置容器并返回 `IServiceProvider`：

  ```
  public IServiceProvider ConfigureServices(IServiceCollection services)
  {
      services.AddMvc();
      // Add other framework services

      // Add Autofac
      var containerBuilder = new ContainerBuilder();
      containerBuilder.RegisterModule<DefaultModule>();
      containerBuilder.Populate(services);
      var container = containerBuilder.Build();
      return new AutofacServiceProvider(container);
  }
  ```

  要使用第三方容器，`Startup.ConfigureServices` 必须返回 `IServiceProvider`。

- 在 `DefaultModule` 中配置 Autofac：

  ```
  public class DefaultModule : Module
  {
      protected override void Load(ContainerBuilder builder)
      {
          builder.RegisterType<CharacterRepository>().As<ICharacterRepository>();
      }
  }
  ```

在运行时，使用 Autofac 来解析类型，并注入依赖关系。要了解有关结合使用 Autofac 和 ASP.NET Core 的详细信息，请参阅 Autofac 文档。

线程安全

单例服务需要是线程安全的。如果单例服务依赖于一个瞬时服务，那么瞬时服务可能也需要

是线程安全的，具体取决于单例使用它的方式。

单个服务的工厂方法，例如 AddSingleton<TService>(IServiceCollection, Func<IServiceProvider,TService>) 的第二个参数，不需要是线程安全的。像类型（`static`）构造函数一样，它保证由单个线程调用一次。

建议

- 不支持基于 `async/await` 和 `Task` 的服务解析。C# 不支持异步构造函数，因此推荐的模式是在同步解析服务后使用异步方法。
- 避免在服务容器中直接存储数据和配置。例如，用户的购物车通常不应添加到服务容器中。配置应使用 选项模型。同样，避免"数据持有者"对象，也就是仅仅为实现对某些其他对象的访问而存在的对象。最好通过 DI 请求实际项目。
- 避免静态访问服务（例如，静态键入 IApplicationBuilder.ApplicationServices 以便在其他地方使用）。
- 避免使用服务定位器模式。例如，可以改为使用 DI 时，不要调用 GetService 来获取服务实例。要避免的另一个服务定位器变体是注入可在运行时解析依赖项的工厂。这两种做法混合了控制反转策略。
- 避免静态访问 `HttpContext`（例如，IHttpContextAccessor.HttpContext）。

像任何一组建议一样，你可能会遇到需要忽略某建议的情况。例外情况很少见 — 主要是框架本身内部的特殊情况。

DI 是静态/全局对象访问模式的替代方法。如果将其与静态对象访问混合使用，则可能无法实现 DI 的优点。

其他资源

- 在 ASP.NET Core 中将依赖项注入到视图
- 在 ASP.NET Core 中将依赖项注入到控制器
- 在 ASP.NET Core要求处理程序中的依赖关系注入
- ASP.NET Core 中的应用启动
- ASP.NET Core 中基于工厂的中间件激活
- 在 ASP.NET Core 中使用依赖关系注入编写干净代码 (MSDN)
- Container-Managed Application Design, Prelude: Where does the Container Belong?（容器托管的应用程序设计，序言：容器属于何处？）
- Explicit Dependencies Principle（显式依赖关系原则）
- 控制反转容器和依赖关系注入模式 (Martin Fowler)
- New 具有粘附性（将代码"粘附"到特定的实现）
- 如何在 ASP.NET Core DI 中注册具有多个接口的服务

ASP.NET Core 中的路由

2019/1/10 • • Edit Online

撰写者：Ryan Nowak、Steve Smith 和 Rick Anderson

对于本主题的 1.1 版本，请下载 ASP.NET Core（版本 1.1，PDF）中的路由。

路由负责将请求 URI 映射到终结点选择器并向终结点调度传入的请求。路由在应用中定义，并在应用启动时进行配置。路由可以选择从请求包含的 URL 中提取值，然后这些值便可用于处理请求。通过使用应用中的路由信息，路由还能生成映射到终结点选择器的 URL。

要在 ASP.NET Core 2.2 中使用最新路由方案，请在 `Startup.ConfigureServices` 中为 MVC 服务注册指定兼容性版本：

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

`EnableEndpointRouting` 选项确定路由是应在内部使用 ASP.NET Core 2.1 或更早版本的基于终结点的逻辑还是使用其基于 IRouter 的逻辑。兼容性版本设置为 2.2 或更高版本时，默认值为 `true`。将值设置为 `false` 以使用先前的路由逻辑：

```
// Use the routing logic of ASP.NET Core 2.1 or earlier:
services.AddMvc(options => options.EnableEndpointRouting = false)
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

有关基于 IRouter 的路由的详细信息，请参阅本主题的 ASP.NET Core 2.1 版本。

路由负责将请求 URI 映射到路由处理程序和调度传入的请求。路由在应用中定义，并在应用启动时进行配置。路由可以选择从请求包含的 URL 中提取值，然后这些值便可用于处理请求。如果使用应用中配置的路由，路由能生成映射到路由处理程序的 URL。

要在 ASP.NET Core 2.1 中使用最新路由方案，请在 `Startup.ConfigureServices` 中为 MVC 服务注册指定兼容性版本：

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

> **IMPORTANT**
>
> 本文档介绍较低级别的 ASP.NET Core 路由。有关 ASP.NET Core MVC 路由的信息，请参阅 在 ASP.NET Core 中路由到控制器操作。有关 Razor Pages 中路由约定的信息，请参阅 ASP.NET Core 中 Razor 页面的路由和应用约定。

查看或下载示例代码（如何下载）

路由基础知识

大多数应用应选择基本的描述性路由方案，让 URL 有可读性和意义。默认传统路由 `{controller=Home}/{action=Index}/{id?}`：

- 支持基本的描述性路由方案。

- 是基于 UI 的应用的有用起点。

开发人员通常在专业情况下(例如，博客和电子商务终结点)使用属性路由或专用传统路由向应用的高流量区域添加其他简洁路由。

Web API 应使用属性路由，将应用功能建模为一组资源，其中操作是由 HTTP 谓词表示。也就是说，对同一逻辑资源执行的许多操作(例如，GET、POST)都将使用相同 URL。属性路由提供了精心设计 API 的公共终结点布局所需的控制级别。

Razor Pages 应用使用默认的传统路由从应用的"页面"文件夹中提供命名资源。还可以使用其他约定来自定义 Razor Pages 路由行为。有关详细信息，请参阅 ASP.NET Core 中的 Razor 页面介绍 和 ASP.NET Core 中 Razor 页面的路由和应用约定。

借助 URL 生成支持，无需通过硬编码 URL 将应用关联到一起，即可开发应用。此支持允许从基本路由配置入手，并在确定应用的资源布局后修改路由。

路由使用"终结点"(`Endpoint`)来表示应用中的逻辑终结点。

终结点定义用于处理请求的委托和任意元数据的集合。元数据用于实现横切关注点，该实现基于附加到每个终结点的策略和配置。

路由系统具有以下特征：

- 路由模板语法用于通过标记化路由参数来定义路由。
- 可以使用常规样式和属性样式终结点配置。
- `IRouteConstraint` 用于确定 URL 参数是否包含给定的终结点约束的有效值。
- 应用模型(如 MVC/Razor Pages)注册其所有终结点，这些终结点具有可预测的路由方案实现。
- 路由实现会在中间件管道中任何所需位置制定路由决策。
- 路由中间件之后出现的中间件可以检查路由中间件针对给定请求 URI 的终结点决策结果。
- 可以在中间件管道中的任何位置枚举应用中的所有终结点。
- 应用可根据终结点信息使用路由生成 URL(例如，用于重定向或链接)，从而避免硬编码 URL，这有助于可维护性。
- URL 生成是基于支持任意可扩展性的地址：
  - 可以使用依赖关系注入 (DI) 在任意位置解析链接生成器 API (`LinkGenerator`) 以生成 URL。
  - 如果无法通过 DI 获得链接生成器 API，则 `IUrlHelper` 会提供生成 URL 的方法。

> **NOTE**
> 在 ASP.NET Core 2.2 中发布终结点路由后，终结点链接的适用范围限制为 MVC/Razor Pages 操作和页面。将计划在未来发布的版本中扩展终结点链接的功能。

路由使用路由(IRouter 的实现)：

- 将传入请求映射到路由处理程序。
- 生成响应中使用的 URL。

默认情况下，应用只有一个路由集合。当请求到达时，将按照路由在集合中的存在顺序来处理路由。框架试图通过在集合中的每个路由上调用 RouteAsync 方法，将传入请求的 URL 与集合中的路由进行匹配。响应可根据路由信息使用路由生成 URL(例如，用于重定向或链接)，从而避免硬编码 URL，这有助于可维护性。

路由系统具有以下特征：

- 路由模板语法用于通过标记化路由参数来定义路由。
- 可以使用常规样式和属性样式终结点配置。
- `IRouteConstraint` 用于确定 URL 参数是否包含给定的终结点约束的有效值。
- 应用模型（如 MVC/Razor Pages）注册了其所有具有可预测的路由方案实现的路由。
- 响应可根据路由信息使用路由生成 URL（例如，用于重定向或链接），从而避免硬编码 URL，这有助于可维护性。
- URL 生成基于支持任意可扩展性的路由。`IUrlHelper` 提供生成 URL 的方法。

路由通过 RouterMiddleware 类连接到中间件管道。ASP.NET Core MVC 向中间件管道添加路由，作为其配置的一部分，并处理 MVC 和 Razor Pages 应用中的路由。要了解如何将路由用作独立组件，请参阅使用路由中间件部分。

URL 匹配

URL 匹配是路由向终结点调度传入请求的过程。此过程基于 URL 路径中的数据，但可以进行扩展以考虑请求中的任何数据。向单独的处理程序调度请求的功能是缩放应用的大小和复杂性的关键。

终结点路由中的路由系统负责所有的调度决策。由于中间件是基于所选终结点来应用策略，因此任何可能影响调度或安全策略应用的决策都应在路由系统内部制定，这一点很重要。

执行终结点委托时，将根据迄今执行的请求处理将 `RouteContext.RouteData` 的属性设置为适当的值。

URL 匹配是一个过程，通过该过程，路由可向处理程序调度传入请求。此过程基于 URL 路径中的数据，但可以进行扩展以考虑请求中的任何数据。向单独的处理程序调度请求的功能是缩放应用的大小和复杂性的关键。

传入请求将进入 `RouterMiddleware`，后者将对序列中的每个路由调用 RouteAsync 方法。IRouter 实例将选择是否通过将 RouteContext.Handler 设置为非 NULL RequestDelegate 来处理请求。如果路由为请求设置处理程序，将停止路由处理，并调用处理程序来处理该请求。如果未找到用于处理请求的路由处理程序，中间件会将请求传递给请求管道中的下一个中间件。

`RouteAsync` 的主要输入是与当前请求关联的 RouteContext.HttpContext。`RouteContext.Handler` 和 RouteContext.RouteData 是路由匹配后设置的输出。

调用 `RouteAsync` 的匹配还可根据迄今执行的请求处理将 `RouteContext.RouteData` 的属性设为适当的值。

RouteData.Values 是从路由中生成的路由值的字典。这些值通常通过标记 URL 来确定，可用来接受用户输入，或者在应用内作出进一步的调度决策。

RouteData.DataTokens 是一个与匹配的路由相关的其他数据的属性包。提供 `DataTokens` 以支持将状态数据与每个路由相关联，以便应用可根据所匹配的路由作出决策。这些值是开发者定义的，不会影响通过任何方式路由的行为。此外，存储于 `RouteData.DataTokens` 中的值可以属于任何类型，与 `RouteData.Values` 相反，后者必须能够转换为字符串，或从字符串进行转换。

RouteData.Routers 是参与成功匹配请求的路由的列表。路由可以相互嵌套。`Routers` 属性可以通过导致匹配的逻辑路由树反映该路径。通常情况下，`Routers` 中的第一项是路由集合，应该用于生成 URL。`Routers` 中的最后一项是匹配的路由处理程序。

URL 生成

URL 生成是通过其可根据一组路由值创建 URL 路径的过程。这需要考虑终结点与访问它们的 URL 之间的逻辑分隔。

终结点路由包含链接生成器 API（`LinkGenerator`）。`LinkGenerator` 是可从 DI 检索的单一实例服务。该 API 可在执行请求的上下文之外使用。MVC 的 `IUrlHelper` 和依赖 `IUrlHelper` 的方案（如 Tag Helpers、HTML Helpers 和 Action Results）使用链接生成器提供链接生成功能。

链接生成器基于“地址”和“地址方案”概念。地址方案是确定哪些终结点用于链接生成的方式。例如，许多用户熟悉的来自 MVC/Razor Pages 的路由名称和路由值方案都是作为地址方案实现的。

链接生成器可以通过以下扩展方法链接到 MVC/Razor Pages 操作和页面：

- `GetPathByAction`
- `GetUriByAction`
- `GetPathByPage`
- `GetUriByPage`

这些方法的重载接受包含 `HttpContext` 的参数。这些方法在功能上等同于 `Url.Action` 和 `Url.Page`，但提供了更大的灵活性和更多的选项。

`GetPath*` 方法与 `Url.Action` 和 `Url.Page` 最相似，因为它们生成包含绝对路径的 URI。`GetUri*` 方法始终生成包含方案和主机的绝对 URI。接受 `HttpContext` 的方法在执行请求的上下文中生成 URI。除非重写，否则将使用来自执行请求的环境路由值、URL 基本路径、方案和主机。

使用地址调用 `LinkGenerator`。生成 URI 的过程分两步进行：

1. 将地址绑定到与地址匹配的终结点列表。
2. 计算每个终结点的 `RoutePattern`，直到找到与提供的值匹配的路由模式。输出结果会与提供给链接生成器的其他 URI 部分进行组合并返回。

对任何类型的地址，`LinkGenerator` 提供的方法均支持标准链接生成功能。使用链接生成器的最简便方法是通过扩展方法对特定地址类型执行操作。

扩展方法	说明
`GetPathByAddress`	根据提供的值生成具有绝对路径的 URI。
`GetUriByAddress`	根据提供的值生成绝对 URI。

WARNING

请注意有关调用 `LinkGenerator` 方法的下列含义：

- 对于不验证传入请求的 `Host` 标头的应用配置，请谨慎使用 `GetUri*` 扩展方法。如果未验证传入请求的 `Host` 标头，则可能以视图/页面中 URI 的形式将不受信任的请求输入发送回客户端。建议所有生产应用都将其服务器配置为针对已知有效值验证 `Host` 标头。
- 在中间件中将 `LinkGenerator` 与 `Map` 或 `MapWhen` 结合使用时，请小心谨慎。`Map*` 会更改执行请求的基路径，这会影响链接生成的输出。所有 `LinkGenerator` API 都允许指定基路径。始终指定一个空的基路径来撤消 `Map*` 对链接生成的影响。

与早期版本路由的差异

ASP.NET Core 2.2 或更高版本中的终结点路由与 ASP.NET Core 中早期版本的路由之间存在一些差异：

- 终结点路由系统不支持基于 `IRouter` 的可扩展性，包括从 `Route` 继承。
- 终结点路由不支持 WebApiCompatShim。使用 2.1 兼容性版本（`.SetCompatibilityVersion(CompatibilityVersion.Version_2_1)`）以继续使用兼容性填充码。
- 在使用传统路由时，终结点路由对生成的 URI 的大小写具有不同的行为。

  请考虑以下默认路由模板：

```
app.UseMvc(routes =>
{
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

假设使用以下路由生成某个操作的链接：

```
var link = Url.Action("ReadPost", "blog", new { id = 17, });
```

如果使用基于 `IRouter` 的路由，此代码生成 URI `/blog/ReadPost/17`，该 URI 遵循所提供路由值的大小写。ASP.NET Core 2.2 或更高版本中的终结点路由生成 `/Blog/ReadPost/17`（"Blog"大写）。终结点路由提供 `IOutboundParameterTransformer` 接口，可用于在全局范围自定义此行为或应用不同的约定来映射 URL。

有关详细信息，请参阅参数转换器参考部分。

- 在试图链接到不存在的控制器/操作或页面时，MVC/Razor Pages 通过传统路由执行的链接生成，其操作具有不同的行为。

  请考虑以下默认路由模板：

  ```
  app.UseMvc(routes =>
  {
      routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
  });
  ```

  假设使用以下路由通过默认模板生成某个操作的链接：

  ```
  var link = Url.Action("ReadPost", "Blog", new { id = 17, });
  ```

  如果使用基于 `IRouter` 的路由，即使 `BlogController` 不存在或没有 `ReadPost` 操作方法，结果也始终为 `/Blog/ReadPost/17`。正如所料，如果操作方法存在，ASP.NET Core 2.2 或更高版本中的终结点路由会生成 `/Blog/ReadPost/17`。但是，如果操作不存在，终结点路由会生成空字符串。从概念上讲，如果操作不存在，终结点路由不会假定终结点存在。

- 与终结点路由一起使用时，链接生成环境值失效算法的行为会有所不同。

  *环境值失效*是一种算法，用于决定当前正在执行的请求（环境值）中的哪些路由值可用于链接生成操作。链接到不同操作时，传统路由会使额外的路由值失效。ASP.NET Core 2.2 之前的版本中，属性路由不具有此行为。在 ASP.NET Core 的早期版本中，如果有另一个操作使用同一路由参数名称，则该操作的链接会导致发生链接生成错误。在 ASP.NET Core 2.2 或更高版本中，链接到另一个操作时，这两种路由形式都会使值失效。

  请考虑 ASP.NET Core2.1 或更高版本中的以下示例。链接到另一个操作（或另一页面）时，路由值可能会按非预期的方式被重用。

  在 /Pages/Store/Product.cshtml 中：

  ```
  @page "{id}"
  @Url.Page("/Login")
  ```

  在 /Pages/Login.cshtml 中：

```
@page "{id?}"
```

如果 ASP.NET Core 2.1 或更早版本中的 URI 为 `/Store/Product/18`，则由 `@Url.Page("/Login")` 在 Store/Info 页面上生成的链接为 `/Login/18`。即使链接目标完全是应用的其他部分，仍会重用 `id` 值 18。`/Login` 页面上下文中的 `id` 路由值可能是用户 ID 值，而非存储产品 ID 值。

在 ASP.NET Core 2.2 或更高版本的终结点路由中，结果为 `/Login`。当链接的目标是另一个操作或页面时，不会重复使用环境值。

- 往返路由参数语法：使用双星号（`**`）catch-all 参数语法时，不对正斜杠进行编码。

  在链接生成期间，路由系统对双星号（`**`）catch-all 参数（例如，`{**myparametername}`）中捕获的除正斜杠外的值进行编码。在 ASP.NET Core 2.2 或更高版本中，基于 `IRouter` 的路由支持双星号 catch-all 参数。

  ASP.NET Core（`{*myparametername}`）早期版本中的单星号 catch-all 参数语法仍然受支持，并对正斜杠进行编码。

路由	链接生成方式为 `URL.ACTION(NEW { CATEGORY = "ADMIN/PRODUCTS" })`...
`/search/{*page}`	`/search/admin%2Fproducts`（对正斜杠进行编码）
`/search/{**page}`	`/search/admin/products`

中间件示例

在以下示例中，中间件使用 `LinkGenerator` API 创建列出存储产品的操作方法的链接。应用中的任何类都可通过将链接生成器注入类并调用 `GenerateLink` 来使用链接生成器。

```
using Microsoft.AspNetCore.Routing;

public class ProductsLinkMiddleware
{
    private readonly LinkGenerator _linkGenerator;

    public ProductsLinkMiddleware(RequestDelegate next, LinkGenerator linkGenerator)
    {
        _linkGenerator = linkGenerator;
    }

    public async Task InvokeAsync(HttpContext httpContext)
    {
        var url = _linkGenerator.GetPathByAction("ListProducts", "Store");

        httpContext.Response.ContentType = "text/plain";

        await httpContext.Response.WriteAsync($"Go to {url} to see our products.");
    }
}
```

URL 生成是通过其可根据一组路由值创建 URL 路径的过程。这需要考虑处理程序与访问它们的 URL 之间的逻辑分隔。

URL 遵循类似的迭代过程，但开头是将用户或框架代码调用到路由集合的 GetVirtualPath 方法中。每个路由按顺序调用其 `GetVirtualPath` 方法，直到返回非 NULL 的 VirtualPathData。

`GetVirtualPath` 的主输入有：

- VirtualPathContext.HttpContext
- VirtualPathContext.Values
- VirtualPathContext.AmbientValues

路由主要使用 `Values` 和 `AmbientValues` 提供的路由值来确定是否可能生成 URL 以及要包括哪些值。`AmbientValues` 是通过匹配当前请求而生成的路由值集。与此相反，`Values` 是指定如何为当前操作生成所需 URL 的路由值。如果路由应获取与当前上下文关联的服务或其他数据时，则提供 `HttpContext`。

> **TIP**
>
> 将 VirtualPathContext.Values 视为 VirtualPathContext.AmbientValues 的一组替代。URL 生成尝试重复使用当前请求中的路由值，以便使用相同路由或路由值生成链接的 URL。

`GetVirtualPath` 的输出是 `VirtualPathData`。`VirtualPathData` 是 `RouteData` 的并行值。`VirtualPathData` 包含输出 URL 的 `VirtualPath` 以及路由应该设置的某些其他属性。

VirtualPathData.VirtualPath 属性包含路由生成的虚拟路径。你可能需要进一步处理路径，具体取决于你的需求。如果要在 HTML 中呈现生成的 URL，请预置应用的基路径。

VirtualPathData.Router 是对已成功生成 URL 的路由的引用。

VirtualPathData.DataTokens 属性是生成 URL 的路由的其他相关数据的字典。这是 RouteData.DataTokens 的并性值。

创建路由

路由提供 Route 类，作为 IRouter 的标准实现。`Route` 使用 route template 语法来定义模式，以便在调用 RouteAsync 时匹配 URL 路径。调用 `GetVirtualPath` 时，`Route` 使用同一路由模板生成 URL。

大多数应用通过调用 MapRoute 或 IRouteBuilder 上定义的一种类似的扩展方法来创建路由。任何 `IRouteBuilder` 扩展方法都会创建 Route 的实例并将其添加到路由集合中。

`MapRoute` 不接受路由处理程序参数。`MapRoute` 仅添加由 DefaultHandler 处理的路由。要了解 MVC 中的路由的详细信息，请参阅 在 ASP.NET Core 中路由到控制器操作。

`MapRoute` 不接受路由处理程序参数。`MapRoute` 仅添加由 DefaultHandler 处理的路由。默认处理程序是 `IRouter`，该处理程序可能无法处理该请求。例如，ASP.NET Core MVC 通常被配置为默认处理程序，仅处理与可用控制器和操作匹配的请求。要了解 MVC 中的路由的详细信息，请参阅 在 ASP.NET Core 中路由到控制器操作。

以下代码示例是典型 ASP.NET Core MVC 路由定义使用的一个 `MapRoute` 调用示例：

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

此模板与 URL 路径相匹配，并且提取路由值。例如，路径 `/Products/Details/17` 生成以下路由值：`{ controller = Products, action = Details, id = 17 }`。

路由值是通过将 URL 路径拆分成段，并且将每段与路由模板中的路由参数名称相匹配来确定的。路由参数已命名。参数通过将参数名称括在大括号 `{ ... }` 中来定义。

上述模板还可匹配 URL 路径 `/`，并且生成值 `{ controller = Home, action = Index }`。这是因为 `{controller}` 和 `{action}` 路由参数具有默认值，且 `id` 路由参数是可选的。路由参数名称为参数定义默认值后，等号 (`=`) 后将有一个值。路由参数名称后面的问号 (`?`) 定义了可选参数。

路由匹配时，具有默认值的路由参数始终会生成路由值。如果没有相应的 URL 路径段，则可选参数不会生成

路由值。有关路由模板方案和语法的详细说明，请参阅路由模板参考部分。

在以下示例中，路由参数定义 `{id:int}` 为 `id` 路由参数定义路由约束：

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id:int}");
```

此模板与类似于 `/Products/Details/17` 而不是 `/Products/Details/Apples` 的 URL 路径相匹配。路由约束实现 IRouteConstraint 并检查路由值，以验证它们。在此示例中，路由值 `id` 必须可转换为整数。有关框架提供的路由约束的说明，请参阅路由约束参考。

其他 `MapRoute` 重载接受 `constraints`、`dataTokens` 和 `defaults` 的值。这些参数的典型用法是传递匿名类型化对象，其中匿名类型的属性名匹配路由参数名称。

以下 `MapRoute` 示例可创建等效路由：

```
routes.MapRoute(
    name: "default_route",
    template: "{controller}/{action}/{id?}",
    defaults: new { controller = "Home", action = "Index" });

routes.MapRoute(
    name: "default_route",
    template: "{controller=Home}/{action=Index}/{id?}");
```

TIP

定义约束和默认值的内联语法对于简单路由很方便。但是，存在内联语法不支持的方案，例如数据令牌。

以下示例演示了一些其他方案：

```
routes.MapRoute(
    name: "blog",
    template: "Blog/{**article}",
    defaults: new { controller = "Blog", action = "ReadArticle" });
```

上述模板与 `/Blog/All-About-Routing/Introduction` 等的 URL 路径相匹配，并提取值 `{ controller = Blog, action = ReadArticle, article = All-About-Routing/Introduction }`。`controller` 和 `action` 的默认路由值由路由生成，即便模板中没有对应的路由参数。可在路由模板中指定默认值。根据路由参数名称前的双星号 (`**`) 外观，`article` 路由参数被定义为 catch-all。全方位路由参数可捕获 URL 路径的其余部分，也能匹配空白字符串。

```
routes.MapRoute(
    name: "blog",
    template: "Blog/{*article}",
    defaults: new { controller = "Blog", action = "ReadArticle" });
```

上述模板与 `/Blog/All-About-Routing/Introduction` 等的 URL 路径相匹配，并提取值 `{ controller = Blog, action = ReadArticle, article = All-About-Routing/Introduction }`。`controller` 和 `action` 的默认路由值由路由生成，即便模板中没有对应的路由参数。可在路由模板中指定默认值。根据路由参数名称前的星号 (`*`) 外观，`article` 路由参数被定义为 catch-all。全方位路由参数可捕获 URL 路径的其余部分，也能匹配空白字符串。

以下示例添加了路由约束和数据令牌：

```
routes.MapRoute(
    name: "us_english_products",
    template: "en-US/Products/{id}",
    defaults: new { controller = "Products", action = "Details" },
    constraints: new { id = new IntRouteConstraint() },
    dataTokens: new { locale = "en-US" });
```

上述模板与 `/en-US/Products/5` 等 URL 路径相匹配，并且提取值 `{ controller = Products, action = Details, id = 5 }` 和数据令牌 `{ locale = en-US }`。

路由类 URL 生成

`Route` 类还可通过将一组路由值与其路由模板组合来生成 URL。从逻辑上来说，这是匹配 URL 路径的反向过程。

> **TIP**
> 为更好了解 URL 生成，请想象你要生成的 URL，然后考虑路由模板将如何匹配该 URL。将生成什么值？ 这大致相当于 URL 在 `Route` 类中的生成方式。

以下示例使用常规 ASP.NET Core MVC 默认路由：

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

使用路由值 `{ controller = Products, action = List }`，将生成 URL `/Products/List`。路由值将替换为相应的路由参数，以形成 URL 路径。由于 `id` 是可选路由参数，因此成功生成的 URL 不具有 `id` 的值。

使用路由值 `{ controller = Home, action = Index }`，将生成 URL `/`。提供的路由值与默认值匹配，并且会安全地省略与默认值对应的段。

已生成的两个 URL 将往返以下路由定义（`/Home/Index` 和 `/`），并生成用于生成该 URL 的相同路由值。

> **NOTE**
> 使用 ASP.NET Core MVC 应用应该使用 UrlHelper 生成 URL，而不是直接调用到路由。

有关 URL 生成的详细信息，请参阅 Url 生成参考部分。

使用路由中间件

引用应用项目文件中的 Microsoft.AspNetCore.App 元包。

将路由添加到 `Startup.ConfigureServices` 中的服务容器：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRouting();
}
```

必须在 `Startup.Configure` 方法中配置路由。该示例应用使用以下 API：

- RouteBuilder
- MapGet – 仅匹配 HTTP GET 请求。
- UseRouter

```
var trackPackageRouteHandler = new RouteHandler(context =>
{
    var routeValues = context.GetRouteData().Values;
    return context.Response.WriteAsync(
        $"Hello! Route values: {string.Join(", ", routeValues)}");
});

var routeBuilder = new RouteBuilder(app, trackPackageRouteHandler);

routeBuilder.MapRoute(
    "Track Package Route",
    "package/{operation:regex(^track|create|detonate$)}/{id:int}");

routeBuilder.MapGet("hello/{name}", context =>
{
    var name = context.GetRouteValue("name");
    // The route handler when HTTP GET "hello/<anything>" matches
    // To match HTTP GET "hello/<anything>/<anything>,
    // use routeBuilder.MapGet("hello/{*name}"
    return context.Response.WriteAsync($"Hi, {name}!");
});

var routes = routeBuilder.Build();
app.UseRouter(routes);
```

下表显示了具有给定 URI 的响应。

URI	响应
`/package/create/3`	Hello! Route values: [operation, create], [id, 3]
`/package/track/-3`	Hello! Route values: [operation, track], [id, -3]
`/package/track/-3/`	Hello! Route values: [operation, track], [id, -3]
`/package/track/`	请求失败，不匹配。
`GET /hello/Joe`	Hi, Joe!
`POST /hello/Joe`	请求失败，仅匹配 HTTP GET。

URI	响应
`GET /hello/Joe/Smith`	请求失败，不匹配。

如果要配置单个路由，请在 `IRouter` 实例中调用 UseRouter 传入。无需使用 RouteBuilder。

框架可提供一组用于创建路由 (RequestDelegateRouteBuilderExtensions) 的扩展方法：

- `MapDelete`
- `MapGet`
- `MapMiddlewareDelete`
- `MapMiddlewareGet`
- `MapMiddlewarePost`
- `MapMiddlewarePut`
- `MapMiddlewareRoute`
- `MapMiddlewareVerb`
- `MapPost`
- `MapPut`
- `MapRoute`
- `MapVerb`

一些列出的方法（如 `MapGet`）需要 `RequestDelegate`。路由匹配时，`RequestDelegate` 会用作路由处理程序。此系列中的其他方法允许配置中间件管道，将其用作路由处理程序。如果 `Map*` 方法不接受处理程序（例如 `MapRoute`），则它将使用 DefaultHandler。

`Map[Verb]` 方法将使用约束来将路由限制为方法名称中的 HTTP 谓词。有关示例，请参阅 MapGet 和 MapVerb。

路由模板参考

如果路由找到匹配项，大括号（`{ ... }`）内的令牌定义绑定的路由参数。可在路由段中定义多个路由参数，但必须由文本值隔开。例如，`{controller=Home}{action=Index}` 不是有效的路由，因为 `{controller}` 和 `{action}` 之间没有文本值。这些路由参数必须具有名称，且可能指定了其他属性。

路由参数以外的文本（例如 `{id}`）和路径分隔符 `/` 必须匹配 URL 中的文本。文本匹配区分大小写，并且基于 URL 路径已解码的表示形式。要匹配文字路由参数分隔符（`{` 或 `}`），请通过重复该字符（`{{` 或 `}}`）来转义分隔符。

尝试捕获具有可选文件扩展名的文件名的 URL 模式还有其他注意事项。例如，考虑模板 `files/{filename}.{ext?}`。当 `filename` 和 `ext` 的值都存在时，将填充这两个值。如果 URL 中仅存在 `filename` 的值，则路由匹配，因为尾随句点（`.`）是可选的。以下 URL 与此路由相匹配：

- `/files/myFile.txt`
- `/files/myFile`

可以使用星号（`*`）或双星号（`**`）作为路由参数的前缀，以绑定到 URI 的其余部分。这些称为 catch-all 参数。例如，`blog/{**slug}` 将匹配以 `/blog` 开头且其后带有任何值（将分配给 `slug` 路由值）的 URI。全方位参数还可以匹配空字符串。

使用路由生成 URL（包括路径分隔符（`/`））时，catch-all 参数会转义相应的字符。例如，路由值为 `{ path = "my/path" }` 的路由 `foo/{*path}` 生成 `foo/my%2Fpath`。请注意转义的正斜杠。要往返路径分隔符，请使用 `**` 路由参数前缀。`{ path = "my/path" }` 的路由 `foo/{**path}` 生成 `foo/my/path`。

可以使用星号（`*`）作为路由参数的前缀，以绑定到 URI 的其余部分。这称为 catch-all 参数。例如，`blog/{*slug}` 将匹配以 `/blog` 开头且其后带有任何值（将分配给 `slug` 路由值）的 URI。全方位参数还可以匹配空字符串。

使用路由生成 URL（包括路径分隔符（`/`））时，catch-all 参数会转义相应的字符。例如，路由值为 `{ path = "my/path" }` 的路由 `foo/{*path}` 生成 `foo/my%2Fpath`。请注意转义的正斜杠。

路由参数可能具有指定的默认值，方法是在参数名称后使用等号（`=`）隔开以指定默认值。例如，`{controller=Home}` 将 `Home` 定义为 `controller` 的默认值。如果参数的 URL 中不存在任何值，则使用默认值。通过在参数名称的末尾附加问号（`?`）可使路由参数成为可选项，如 `id?` 中所示。可选值和默认路径参数的区别在于具有默认值的路由参数始终会生成值 —，而可选参数仅当请求 URL 提供值时才会具有值。

路由参数可能具有必须与从 URL 中绑定的路由值匹配的约束。在路由参数后面添加一个冒号（`:`）和约束名称可指定路由参数上的内联约束。如果约束需要参数，将以在约束名称后括在括号（`(...)`）中的形式提供。通过追加另一个冒号（`:`）和约束名称，可指定多个内联约束。

约束名称和参数将传递给 IInlineConstraintResolver 服务，以创建 IRouteConstraint 的实例，用于处理 URL。例如，路由模板 `blog/{article:minlength(10)}` 使用参数 `10` 指定 `minlength` 约束。有关路由约束详情以及框架提供的约束列表，请参阅路由约束引用部分。

路由参数还可以具有参数转换器，用于在生成链接以及将操作和页面匹配到 URI 时转换参数的值。与约束类似，可以通过在路由参数名称后面添加冒号（`:`）和转换器名称，将参数变换器内联添加到路径参数。例如，路由模板 `blog/{article:slugify}` 指定 `slugify` 转换器。有关参数转换的详细信息，请参阅参数转换器参考部分。

下表演示了示例路由模板及其行为。

路由模板	示例匹配 URI	请求 URI...
`hello`	`/hello`	仅匹配单个路径 `/hello`。
`{Page=Home}`	`/`	匹配并将 `Page` 设置为 `Home`。
`{Page=Home}`	`/Contact`	匹配并将 `Page` 设置为 `Contact`。
`{controller}/{action}/{id?}`	`/Products/List`	映射到 `Products` 控制器和 `List` 操作。
`{controller}/{action}/{id?}`	`/Products/Details/123`	映射到 `Products` 控制器和 `Details` 操作（`id` 设置为 123）。
`{controller=Home}/{action=Index}/{id?}`	`/`	映射到 `Home` 控制器和 `Index` 方法（忽略 `id`）。

使用模板通常是进行路由最简单的方法。还可在路由模板外指定约束和默认值。

> **TIP**
>
> 启用日志记录以查看内置路由实现（如 `Route`）如何匹配请求。

保留的路由名称

以下关键字是保留的名称，它们不能用作路由名称或参数：

- `action`
- `area`
- `controller`
- `handler`
- `page`

路由约束参考

路由约束在传入 URL 发生匹配时执行，URL 路径标记为路由值。路径约束通常检查通过路径模板关联的路径值，并对该值是否可接受做出是/否决定。某些路由约束使用路由值以外的数据来考虑是否可以路由请求。例如，HttpMethodRouteConstraint 可以根据其 HTTP 谓词接受或拒绝请求。约束用于路由请求和链接生成。

> **WARNING**
> 请勿将约束用于“输入验证”。如果将约束用于“输入约束”，那么无效输入将导致“404 - 未找到”响应，而不是含相应错误消息的“400 - 错误请求”。路由约束用于消除类似路由的歧义，而不是验证特定路由的输入。

下表演示示例路由约束及其预期行为。

约束	示例	匹配项示例	说明
`int`	`{id:int}`	`123456789`，`-123456789`	匹配任何整数
`bool`	`{active:bool}`	`true`，`FALSE`	匹配 `true` 或 `false`（区分大小写）
`datetime`	`{dob:datetime}`	`2016-12-31`，`2016-12-31 7:32pm`	匹配有效的 `DateTime` 值（位于固定区域性中 - 查看警告）
`decimal`	`{price:decimal}`	`49.99`，`-1,000.01`	匹配有效的 `decimal` 值（位于固定区域性中 - 查看警告）
`double`	`{weight:double}`	`1.234`，`-1,001.01e8`	匹配有效的 `double` 值（位于固定区域性中 - 查看警告）
`float`	`{weight:float}`	`1.234`，`-1,001.01e8`	匹配有效的 `float` 值（位于固定区域性中 - 查看警告）
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`，`{CD2C1638-1638-72D5-1638-DEADBEEF1638}`	匹配有效的 `Guid` 值
`long`	`{ticks:long}`	`123456789`，`-123456789`	匹配有效的 `long` 值

约束	示例	匹配项示例	说明
`minlength(value)`	`{username:minlength(4)}`	`Rick`	字符串必须至少为 4 个字符
`maxlength(value)`	`{filename:maxlength(8)}`	`Richard`	字符串不得超过 8 个字符
`length(length)`	`{filename:length(12)}`	`somefile.txt`	字符串必须正好为 12 个字符
`length(min,max)`	`{filename:length(8,16)}`	`somefile.txt`	字符串必须至少为 8 个字符，且不得超过 16 个字符
`min(value)`	`{age:min(18)}`	`19`	整数值必须至少为 18
`max(value)`	`{age:max(120)}`	`91`	整数值不得超过 120
`range(min,max)`	`{age:range(18,120)}`	`91`	整数值必须至少为 18，且不得超过 120
`alpha`	`{name:alpha}`	`Rick`	字符串必须由一个或多个字母字符（`a`-`z`，区分大小写）组成
`regex(expression)`	`{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}`	`123-45-6789`	字符串必须匹配正则表达式（参见有关定义正则表达式的提示）
`required`	`{name:required}`	`Rick`	用于强制在 URL 生成过程中存在非参数值

可向单个参数应用多个由冒号分隔的约束。例如，以下约束将参数限制为大于或等于 1 的整数值：

```
[Route("users/{id:int:min(1)}")]
public User GetUserById(int id) { }
```

> **WARNING**
> 验证 URL 的路由约束并将转换为始终使用固定区域性的 CLR 类型（例如 `int` 或 `DateTime`）。这些约束假定 URL 不可本地化。框架提供的路由约束不会修改存储于路由值中的值。从 URL 中分析的所有路由值都将存储为字符串。例如，`float` 约束会尝试将路由值转换为浮点数，但转换后的值仅用来验证其是否可转换为浮点数。

正则表达式

ASP.NET Core 框架将向正则表达式构造函数添加 `RegexOptions.IgnoreCase | RegexOptions.Compiled | RegexOptions.CultureInvariant`。有关这些成员的说明，请参阅 RegexOptions。

正则表达式与路由和 C# 计算机语言使用的分隔符和令牌相似。必须对正则表达式令牌进行转义。要在路由中使用正则表达式 `^\d{3}-\d{2}-\d{4}$`，表达式必须在字符串中提供 `\`（单反斜杠）字符，正如 C# 源文件中的 `\\`（双反斜杠）字符一样，以便对 `\` 字符串转义字符进行转义（除非使用字符串文本）。要对路由参数分隔符进行转义（`{`、`}`、`[`、`]`），请将表达式（`{{`、`}`、`[[`、`]]`）中的字符数加倍。下表展示了正则表达式和转义版本。

正则表达式	转义后的正则表达式
`^\d{3}-\d{2}-\d{4}$`	`^\\d{{3}}-\\d{{2}}-\\d{{4}}$`
`^[a-z]{2}$`	`^[[a-z]]{{2}}$`

路由中使用的正则表达式通常以脱字号 (`^`) 开头，并匹配字符串的起始位置。表达式通常以美元符号 (`$`) 字符结尾，并匹配字符串的结尾。`^` 和 `$` 字符可确保正则表达式匹配整个路由参数值。如果没有 `^` 和 `$` 字符，正则表达式将匹配字符串内的所有子字符串，而这通常是不需要的。下表提供了示例并说明了它们匹配或匹配失败的原因。

表达式	STRING	匹配	注释
`[a-z]{2}`	hello	是	子字符串匹配
`[a-z]{2}`	123abc456	是	子字符串匹配
`[a-z]{2}`	mz	是	匹配表达式
`[a-z]{2}`	MZ	是	不区分大小写
`^[a-z]{2}$`	hello	No	参阅上述 `^` 和 `$`
`^[a-z]{2}$`	123abc456	No	参阅上述 `^` 和 `$`

有关正则表达式语法的详细信息，请参阅 .NET Framework 正则表达式。

若要将参数限制为一组已知的可能值，可使用正则表达式。例如，`{action:regex(^(list|get|create)$)}` 仅将 `action` 路由值匹配到 `list`、`get` 或 `create`。如果传递到约束字典中，字符串 `^(list|get|create)$` 将等效。已传递到约束字典(不与模板内联)且不匹配任何已知约束的约束还将被视为正则表达式。

参数转换器参考

参数转换器：

- 在为 `Route` 生成链接时执行。
- 实现 `Microsoft.AspNetCore.Routing.IOutboundParameterTransformer`。
- 使用 ConstraintMap 进行配置。
- 获取参数的路由值并将其转换为新的字符串值。
- 在生成的链接中使用转换后的值的结果。

例如，路由模式 `blog\{article:slugify}`(具有 `Url.Action(new { article = "MyTestArticle" })`)中的自定义 `slugify` 参数转换器生成 `blog\my-test-article`。

若要在路由模式中使用参数转换器，请先在 `Startup.ConfigureServices` 中使用 ConstraintMap 对其进行配置：

```
services.AddRouting(options =>
{
    // Replace the type and the name used to refer to it with your own
    // IOutboundParameterTransformer implementation
    options.ConstraintMap["slugify"] = typeof(SlugifyParameterTransformer);
});
```

框架使用参数转化器来转换进行终结点解析的 URI。例如，ASP.NET Core MVC 使用参数转换器来转换用于匹配 `area``controller``action` 和 `page` 的路由值。

```
routes.MapRoute(
    name: "default",
    template: "{controller:slugify=Home}/{action:slugify=Index}/{id?}");
```

使用上述路由，操作 `SubscriptionManagementController.GetAll()` 与 URI `/subscription-management/get-all` 匹配。参数转换器不会更改用于生成链接的路由值。例如，`Url.Action("GetAll", "SubscriptionManagement")` 输出 `/subscription-management/get-all`。

对于结合使用参数转换器和所生成的路由，ASP.NET Core 提供了 API 约定：

- ASP.NET Core MVC 还具有 `Microsoft.AspNetCore.Mvc.ApplicationModels.RouteTokenTransformerConvention` API 约定。该约定将指定的参数转换器应用于应用中的所有属性路由。在替换属性路径令牌时，参数转换器将转换这些令牌。有关详细信息，请参阅使用参数转换器自定义标记替换。
- Razor Pages 具有 `Microsoft.AspNetCore.Mvc.ApplicationModels.PageRouteTransformerConvention` API 约定。此约定将指定的参数转换器应用于所有自动发现的 Razor Pages。参数转换器转换 Razor Pages.路由的文件夹和文件名段。有关详细信息，请参阅使用参数转换器自定义页面路由。

URL 生成参考

以下示例演示如何在给定路由值字典和 RouteCollection 的情况下生成路由链接。

```
app.Run(async (context) =>
{
    var dictionary = new RouteValueDictionary
    {
        { "operation", "create" },
        { "id", 123}
    };

    var vpc = new VirtualPathContext(context, null, dictionary,
        "Track Package Route");
    var path = routes.GetVirtualPath(vpc).VirtualPath;

    context.Response.ContentType = "text/html";
    await context.Response.WriteAsync("Menu<hr/>");
    await context.Response.WriteAsync(
        $"<a href='{path}'>Create Package 123</a><br/>");
});
```

上述示例末尾生成的 `VirtualPath` 为 `/package/create/123`。字典提供"跟踪包路由"模板 `package/{operation}/{id}` 的 `operation` 和 `id` 路由值。有关详细信息，请参阅使用路由中间件部分或示例应用中的示例代码。

`VirtualPathContext` 构造函数的第二个参数是环境值的集合。由于环境值限制了开发人员在请求上下文中必须指定的值数，因此环境值使用起来很方便。当前请求的当前路由值被视为链接生成的环境值。在 ASP.NET Core MVC 应用 `HomeController` 的 `About` 操作中，无需指定控制器路由值即可链接到使用 `Home` 环境值的 `Index` 操作—。

忽略与参数不匹配的环境值。在显式提供的值会覆盖环境值的情况下，也会忽略环境值。在 URL 中将从左到右进行匹配。

显式提供但与路由片段不匹配的值将添加到查询字符串中。下表显示使用路由模板 `{controller}/{action}/{id?}` 时的结果。

环境值	显式值	结果
控制器 ="Home"	操作 ="About"	`/Home/About`
控制器 ="Home"	控制器 ="Order", 操作 ="About"	`/Order/About`
控制器 = "Home", 颜色 = "Red"	操作 ="About"	`/Home/About`
控制器 ="Home"	操作 ="About", 颜色 ="Red"	`/Home/About?color=Red`

如果路由具有不对应于参数的默认值，且该值以显式方式提供，则它必须与默认值相匹配：

```
routes.MapRoute("blog_route", "blog/{*slug}",
    defaults: new { controller = "Blog", action = "ReadPost" });
```

当提供 `controller` 和 `action` 的匹配值时，链接生成仅为此路由生成链接。

在 ASP.NET Core 中使用多个环境

2018/10/30 • • Edit Online

作者：Rick Anderson

ASP.NET Core 基于使用环境变量的运行时环境配置应用行为。

查看或下载示例代码（如何下载）

环境

ASP.NET Core 在应用启动时读取环境变量 `ASPNETCORE_ENVIRONMENT`，并将该值存储在 IHostingEnvironment.EnvironmentName 中。`ASPNETCORE_ENVIRONMENT` 可设置为任意值，但框架支持三个值：Development、Staging 和 Production。如果未设置 `ASPNETCORE_ENVIRONMENT`，则默认为 `Production`。

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    if (env.IsProduction() || env.IsStaging() || env.IsEnvironment("Staging_2"))
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseMvc();
}
```

前面的代码：

- 当 `ASPNETCORE_ENVIRONMENT` 设置为 `Development` 时，调用 UseDeveloperExceptionPage。
- 当 `ASPNETCORE_ENVIRONMENT` 的值设置为下列之一时，调用 UseExceptionHandler：
  - `Staging`
  - `Production`
  - `Staging_2`

环境标记帮助程序使用 `IHostingEnvironment.EnvironmentName` 的值来包含或排除元素中的标记：

```
<environment include="Development">
    <div>&lt;environment include="Development"&gt;</div>
</environment>
<environment exclude="Development">
    <div>&lt;environment exclude="Development"&gt;</div>
</environment>
<environment include="Staging,Development,Staging_2">
    <div>
        &lt;environment include="Staging,Development,Staging_2"&gt;
    </div>
</environment>
```

在 Windows 和 macOS 上，环境变量和值不区分大小写。默认情况下，Linux 环境变量和值要区分大小写。

开发

开发环境可以启用不应该在生产中公开的功能。例如，ASP.NET Core 模板在开发环境中启用了开发人员异常页。

本地计算机开发环境可以在项目的 Properties\launchSettings.json 文件中设置。在 launchSettings.json 中设置的环境值替代在系统环境中设置的值。

以下 JSON 显示 launchSettings.json 文件中的三个配置文件：

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:54339/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_My_Environment": "1",
        "ASPNETCORE_DETAILEDERRORS": "1",
        "ASPNETCORE_ENVIRONMENT": "Staging"
      }
    },
    "EnvironmentsSample": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging"
      },
      "applicationUrl": "http://localhost:54340/"
    },
    "Kestrel Staging": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_My_Environment": "1",
        "ASPNETCORE_DETAILEDERRORS": "1",
        "ASPNETCORE_ENVIRONMENT": "Staging"
      },
      "applicationUrl": "http://localhost:51997/"
    }
  }
}
```

NOTE

launchSettings.json 中的 `applicationUrl` 属性可指定服务器 URL 的列表。在列表中的 URL 之间使用分号：

```
"EnvironmentsSample": {
   "commandName": "Project",
   "launchBrowser": true,
   "applicationUrl": "https://localhost:5001;http://localhost:5000",
   "environmentVariables": {
     "ASPNETCORE_ENVIRONMENT": "Development"
   }
}
```

使用 dotnet run 启动应用时，使用具有 `"commandName": "Project"` 的第一个配置文件。`commandName` 的值指定要启动的 Web 服务器。`commandName` 可为以下任一项：

- IIS Express
- IIS
- Project（启动 Kestrel 的项目）

使用 dotnet run 启动应用时：

- 如果可用，读取 launchSettings.json。launchSettings.json 中的 `environmentVariables` 设置会替代环境变量。
- 此时显示承载环境。

以下输出显示了使用 dotnet run 启动的应用：

```
PS C:\Websites\EnvironmentsSample> dotnet run
Using launch settings from C:\Websites\EnvironmentsSample\Properties\launchSettings.json...
Hosting environment: Staging
Content root path: C:\Websites\EnvironmentsSample
Now listening on: http://localhost:54340
Application started. Press Ctrl+C to shut down.
```

Visual Studio 项目属性“调试”选项卡提供 GUI 来编辑 launchSettings.json 文件：

在 Web 服务器重新启动之前，对项目配置文件所做的更改可能不会生效。必须重新启动 Kestrel 才能检测到对其环境所做的更改。

> **WARNING**
>
> launchSettings.json 不应存储机密。机密管理器工具可用于存储本地开发的机密。

使用 Visual Studio Code 时，可以在 .vscode/launch.json 文件中设置环境变量。以下示例将环境设置为 `Development`：

```
{
   "version": "0.2.0",
   "configurations": [
        {
            "name": ".NET Core Launch (web)",

            ... additional VS Code configuration settings ...

            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        }
    ]
}
```

使用与 Properties/launchSettings.json 相同的方法通过 `dotnet run` 启动应用时，不读取项目中的 .vscode/launch.json 文件。在没有 launchSettings.json 文件的 Development 环境中启动应用时，需要使用环境变量设置环境或者将命令行参数设为 `dotnet run` 命令。

生产

Production 环境应配置为最大限度地提高安全性、性能和应用可靠性。不同于开发的一些通用设置包括：

- 缓存。
- 客户端资源被捆绑和缩小，并可能从 CDN 提供。
- 已禁用诊断错误页。
- 已启用友好错误页。
- 已启用生产记录和监视。例如，Application Insights。

设置环境

为测试设置特定环境通常很有用。如果未设置环境，默认值为 `Production`，这会禁用大多数调试功能。设置环境的方法取决于操作系统。

Azure 应用服务

若要在 Azure 应用服务中设置环境，请执行以下步骤：

1. 从“应用服务”边栏选项卡中选择应用。
2. 在“设置”组中，选择“应用程序设置”边栏选项卡。
3. 在“应用程序设置”区域中，选择“添加新设置”。
4. 在“输入名称”中提供 `ASPNETCORE_ENVIRONMENT`。在“输入值”中提供环境（例如 `Staging`）。
5. 交换部署槽位时，如果希望环境设置保持当前槽位，请选中“槽位设置”复选框。有关详细信息，请参阅 Azure 文档：交换了哪些设置？。
6. 选择边栏选项卡顶部的“保存”。

在 Azure 门户中添加、更改或删除应用设置（环境变量）后，Azure 应用服务自动重启应用。

Windows

若要在使用 dotnet run 启动该应用时为当前会话设置 `ASPNETCORE_ENVIRONMENT`，则使用以下命令：

命令提示符

```
set ASPNETCORE_ENVIRONMENT=Development
```

PowerShell

```
$Env:ASPNETCORE_ENVIRONMENT = "Development"
```

这些命令仅对当前窗口有效。窗口关闭时，`ASPNETCORE_ENVIRONMENT` 设置将恢复为默认设置或计算机值。

若要在 Windows 中全局设置值，请采用下列两种方法之一：

- 依次打开“控制面板” > “系统” > “高级系统设置”，再添加或编辑“`ASPNETCORE_ENVIRONMENT`”值：

- 打开管理命令提示符并运行 `setx` 命令，或打开管理 PowerShell 命令提示符并运行 `[Environment]::SetEnvironmentVariable`：

  命令提示符

  ```
  setx ASPNETCORE_ENVIRONMENT Development /M
  ```

  `/M` 开关指明，在系统一级设置环境变量。如果未使用 `/M` 开关，就会为用户帐户设置环境变量。

  PowerShell

  ```
  [Environment]::SetEnvironmentVariable("ASPNETCORE_ENVIRONMENT", "Development", "Machine")
  ```

  `Machine` 选项值指明，在系统一级设置环境变量。如果将选项值更改为 `User`，就会为用户帐户设置环境变量。

如果全局设置 `ASPNETCORE_ENVIRONMENT` 环境变量，它就会对在值设置后打开的任何命令窗口中对 `dotnet run` 起作用。

web.config

若要使用 web.config 设置 `ASPNETCORE_ENVIRONMENT` 环境变量，请参阅 ASP.NET Core 模块的“设置环境变量”部分。使用 web.config 设置 `ASPNETCORE_ENVIRONMENT` 环境变量后，它的值会替代系统级设置。

每个 IIS 应用程序池

若要为在独立应用池中运行的应用设置 `ASPNETCORE_ENVIRONMENT` 环境变量（IIS 10.0 或更高版本支持此操作），请参阅环境变量 <environmentVariables> 主题中的“AppCmd.exe 命令”部分。为应用池设置 `ASPNETCORE_ENVIRONMENT` 环境变量后，它的值会替代系统级设置。

> **IMPORTANT**
>
> 在 IIS 中托管应用并添加或更改 `ASPNETCORE_ENVIRONMENT` 环境变量时，请采用下列方法之一，让新值可供应用拾取：
>
> - 在命令提示符处依次执行 `net stop was /y` 和 `net start w3svc`。
> - 重启服务器。

macOS

设置 macOS 的当前环境可在运行应用时完成：

```
ASPNETCORE_ENVIRONMENT=Development dotnet run
```

或者，在运行应用前使用 `export` 设置环境：

```
export ASPNETCORE_ENVIRONMENT=Development
```

在 .bashrc 或 .bash_profile 文件中设置计算机级环境变量。使用任意文本编辑器编辑文件。添加以下语句：

```
export ASPNETCORE_ENVIRONMENT=Development
```

Linux

对于 Linux 发行版，请在命令提示符中使用 `export` 命令进行基于会话的变量设置，并使用 bash_profile 文件进行计算机级环境设置。

按环境配置

若要按环境加载配置，我们建议：

- appsettings 文件 (*appsettings.<>.json)。请参阅配置：文件配置提供程序。
- 环境变量（在托管应用的每个系统上进行设置）。请参阅配置：文件配置提供程序和开发环境中应用密码的安全存储：环境变量。
- 密码管理器（仅限开发环境中）。请参阅 安全存储中 ASP.NET Core 中开发的应用程序机密。

基于环境的 Startup 类和方法

Startup 类约定

当 ASP.NET Core 应用启动时，Startup 类启动应用。应用可以为不同的环境单独定义 `Startup` 类（例如，`StartupDevelopment`），相应 `Startup` 类会在运行时得到选择。优先考虑名称后缀与当前环境相匹配的类。如果找不到匹配的 `Startup{EnvironmentName}`，就会使用 `Startup` 类。

若要实现基于环境的 `Startup` 类，请为使用中的每个环境创建 `Startup{EnvironmentName}` 类，并创建回退 `Startup` 类：

```
// Startup class to use in the Development environment
public class StartupDevelopment
{
    public void ConfigureServices(IServiceCollection services)
    {
        ...
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        ...
    }
}

// Startup class to use in the Production environment
public class StartupProduction
{
    public void ConfigureServices(IServiceCollection services)
    {
        ...
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        ...
    }
}

// Fallback Startup class
// Selected if the environment doesn't match a Startup{EnvironmentName} class
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        ...
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        ...
    }
}
```

使用接受程序集名称的 UseStartup(IWebHostBuilder, String) 重载：

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args)
{
    var assemblyName = typeof(Startup).GetTypeInfo().Assembly.FullName;

    return WebHost.CreateDefaultBuilder(args)
        .UseStartup(assemblyName);
}
```

```
public static void Main(string[] args)
{
    CreateWebHost(args).Run();
}

public static IWebHost CreateWebHost(string[] args)
{
    var assemblyName = typeof(Startup).GetTypeInfo().Assembly.FullName;

    return WebHost.CreateDefaultBuilder(args)
        .UseStartup(assemblyName)
        .Build();
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        var assemblyName = typeof(Startup).GetTypeInfo().Assembly.FullName;

        var host = new WebHostBuilder()
            .UseStartup(assemblyName)
            .Build();

        host.Run();
    }
}
```

Startup 方法约定

Configure 和 ConfigureServices 支持窗体 `Configure<EnvironmentName>` 和 `Configure<EnvironmentName>Services` 的环境特定版本：

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        StartupConfigureServices(services);
    }

    public void ConfigureStagingServices(IServiceCollection services)
    {
        StartupConfigureServices(services);
    }

    private void StartupConfigureServices(IServiceCollection services)
    {
        services.Configure<CookiePolicyOptions>(options =>
        {
            options.CheckConsentNeeded = context => true;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });

        services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        if (env.IsProduction() || env.IsStaging() || env.IsEnvironment("Staging_2"))
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();
        app.UseMvc();
    }

    public void ConfigureStaging(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (!env.IsStaging())
        {
            throw new Exception("Not staging.");
        }

        app.UseExceptionHandler("/Error");
        app.UseHsts();
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();
        app.UseMvc();
    }
}
```

其他资源

- ASP.NET Core 中的应用启动
- ASP.NET Core 中的配置
- IHostingEnvironment.EnvironmentName

ASP.NET Core 中的配置

2018/12/12 • • Edit Online

作者：Luke Latham

ASP.NET Core 中的应用配置基于配置提供程序建立的键值对。配置提供程序将配置数据从各种配置源读取到键值对：

- Azure Key Vault
- 命令行参数
- （已安装或已创建的）自定义提供程序
- 目录文件
- 环境变量
- 内存中的 .NET 对象
- 设置文件

- Azure Key Vault
- 命令行参数
- （已安装或已创建的）自定义提供程序
- 环境变量
- 内存中的 .NET 对象
- 设置文件

- 命令行参数
- （已安装或已创建的）自定义提供程序
- 环境变量
- 内存中的 .NET 对象
- 设置文件

选项模式是本主题中描述的配置概念的扩展。选项使用类来表示相关设置的组。有关使用选项模式的详细信息，请参阅 ASP.NET Core 中的选项模式。

查看或下载示例代码（如何下载）

本主题中提供的示例依赖于以下内容：

- 使用 SetBasePath 设置应用的基本路径。通过引用 Microsoft.Extensions.Configuration.FileExtensions 包，可以向应用提供 `SetBasePath`。
- 使用 GetSection 解析配置文件的各个部分。通过引用 Microsoft.Extensions.Configuration 包向应用提供 `GetSection`。
- 使用 Bind 和 Get<T> 将配置绑定到 .NET 类。通过引用 Microsoft.Extensions.Configuration.Binder 包向应用提供 `Bind` 和 `Get<T>`。ASP.NET Core 1.1 或更高版本中提供了 `Get<T>`。

这三个包均包括在 Microsoft.AspNetCore.App 元包中。

这三个包均包括在 Microsoft.AspNetCore.All 元包中。

主机与应用配置

在配置并启动应用之前，配置并启动主机。主机负责应用程序启动和生存期管理。应用和主机均使用本主题中所述的配置提供程序进行配置。主机配置键值对成为应用的全局配置的一部分。有关在构建

主机时如何使用配置提供程序以及配置源如何影响主机配置的详细信息，请参阅 ASP.NET Core 中的 Web 主机和通用主机。

安全性

采用以下最佳实践：

- 请勿在配置提供程序代码或纯文本配置文件中存储密码或其他敏感数据。
- 不要在开发或测试环境中使用生产机密。
- 请在项目外部指定机密，避免将其意外提交到源代码存储库。

详细了解如何使用多个环境和管理使用 Secret Manager 的开发中的应用机密的安全存储（包括使用环境变量存储敏感数据的建议）。Secret Manager 使用文件配置提供程序将用户机密存储在本地系统上的 JSON 文件中。本主题后面将介绍文件配置提供程序。

Azure Key Vault 是安全存储应用机密的一种选择。有关更多信息，请参见在 ASP.NET Core 的 azure 密钥保管库配置提供程序。

分层配置数据

配置 API 能够通过在配置键中使用分隔符来展平分层数据以保持分层配置数据。

在以下 JSON 文件中，两个节的结构化层次结构中存在四个键：

```
{
  "section0": {
    "key0": "value",
    "key1": "value"
  },
  "section1": {
    "key0": "value",
    "key1": "value"
  }
}
```

将文件读入配置时，将创建唯一键以保持配置源的原始分层数据结构。使用冒号 (`:`) 展平节和键以保持原始结构：

- section0:key0
- section0:key1
- section1:key0
- section1:key1

GetSection 和 GetChildren 方法可用于隔离各个节和配置数据中某节的子节。稍后将在 GetSection、GetChildren 和 Exists 中介绍这些方法。

约定

在应用启动时，将按照指定的配置提供程序的顺序读取配置源。

应用启动后，在更改基础设置文件时，文件配置提供程序可以重载配置。本主题后面将介绍文件配置提供程序。

应用的依赖关系注入 (DI) 容器中提供了 IConfiguration。配置提供程序不能使用 DI，因为主机在设置这些提供程序时 DI 不可用。

配置键采用以下约定：

- 键不区分大小写。例如，`ConnectionString` 和 `connectionstring` 被视为等效键。
- 如果由相同或不同的配置提供程序设置相同键的值，则键上设置的最后一个值就是所使用的值。
- 分层键
  - 在配置 API 中，冒号分隔符（`:`）适用于所有平台。
  - 在环境变量中，冒号分隔符可能无法适用于所有平台。而所有平台均支持采用双下划线（`__`），并可以将其转换为冒号。
  - 在 Azure Key Vault 中，分层键使用 `--`（两个破折号）作为分隔符。将机密加载到应用的配置中时，必须提供代码以用冒号替换破折号。
- ConfigurationBinder 支持使用配置键中的数组索引将数组绑定到对象。数组绑定将在将数组绑定到类部分中进行介绍。

配置值采用以下约定：

- 值是字符串。
- NULL 值不能存储在配置中或绑定到对象。

提供程序

下表显示了 ASP.NET Core 应用可用的配置提供程序。

提供程序	通过以下对象提供配置...
Azure Key Vault 配置提供程序（安全主题）	Azure Key Vault
命令行配置提供程序	命令行参数
自定义配置提供程序	自定义源
环境变量配置提供程序	环境变量
文件配置提供程序	文件（INI、JSON、XML）
Key-per-file 配置提供程序	目录文件
内存配置提供程序	内存中集合
用户机密 (Secret Manager)（安全主题）	用户配置文件目录中的文件

提供程序	通过以下对象提供配置...
Azure Key Vault 配置提供程序（安全主题）	Azure Key Vault
命令行配置提供程序	命令行参数
自定义配置提供程序	自定义源
环境变量配置提供程序	环境变量
文件配置提供程序	文件（INI、JSON、XML）
内存配置提供程序	内存中集合

提供程序	通过以下对象提供配置...
用户机密 (Secret Manager)(安全主题)	用户配置文件目录中的文件

提供程序	通过以下对象提供配置...
命令行配置提供程序	命令行参数
自定义配置提供程序	自定义源
环境变量配置提供程序	环境变量
文件配置提供程序	文件(INI、JSON、XML)
内存配置提供程序	内存中集合
用户机密 (Secret Manager)(安全主题)	用户配置文件目录中的文件

按照启动时指定的配置提供程序的顺序读取配置源。本主题中所述的配置提供程序按字母顺序进行介绍，而不是按代码排列顺序进行介绍。代码中的配置提供程序应以特定顺序排列以符合基础配置源的优先级。

配置提供程序的典型顺序为：

1. 文件(appsettings.json、appsettings.{Environment}.json，其中 `{Environment}` 是应用的当前托管环境)
2. Azure 密钥保管库
3. 用户机密 (Secret Manager)(仅限开发环境中)
4. 环境变量
5. 命令行参数

通常的做法是将命令行配置提供程序置于一系列提供程序的末尾，以允许命令行参数替代由其他提供程序设置的配置。

在使用 CreateDefaultBuilder 初始化新的 WebHostBuilder 时，将使用此提供程序序列。有关详细信息，请参阅 Web 主机：设置主机。

可以使用 ConfigurationBuilder 和在 `Startup` 中调用其 Build 方法来为应用(而不是主机)创建此提供程序序列：

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true,
            reloadOnChange: true);

    var appAssembly = Assembly.Load(new AssemblyName(env.ApplicationName));

    if (appAssembly != null)
    {
        builder.AddUserSecrets(appAssembly, optional: true);
    }

    builder.AddEnvironmentVariables();

    Configuration = builder.Build();
}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IConfiguration>(Configuration);
}
```

在前面的示例中，IHostingEnvironment 提供了环境名称（`env.EnvironmentName`）和应用程序集名称（`env.ApplicationName`）。有关更多信息，请参见在 ASP.NET Core 中使用多个环境。

ConfigureAppConfiguration

构建 Web 主机时调用 ConfigureAppConfiguration 以指定应用的配置提供程序以及 CreateDefaultBuilder 自动添加的配置提供程序：

```
public class Program
{
    public static Dictionary<string, string> arrayDict = new Dictionary<string, string>
        {
            {"array:entries:0", "value0"},
            {"array:entries:1", "value1"},
            {"array:entries:2", "value2"},
            {"array:entries:4", "value4"},
            {"array:entries:5", "value5"}
        };

    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.SetBasePath(Directory.GetCurrentDirectory());
                config.AddInMemoryCollection(arrayDict);
                config.AddJsonFile("json_array.json", optional: false, reloadOnChange: false);
                config.AddJsonFile("starship.json", optional: false, reloadOnChange: false);
                config.AddXmlFile("tvshow.xml", optional: false, reloadOnChange: false);
                config.AddEFConfiguration(options => options.UseInMemoryDatabase("InMemoryDb"));
                config.AddCommandLine(args);
            })
            .UseStartup<Startup>();
}
```

命令行配置提供程序

CommandLineConfigurationProvider 在运行时从命令行参数键值对加载配置。

要激活命令行配置，请在 ConfigurationBuilder 的实例上调用 AddCommandLine 扩展方法。

使用 CreateDefaultBuilder 初始化新的 WebHostBuilder 时会自动调用 `AddCommandLine`。有关详细信息，请参阅 Web 主机：设置主机。

此外，`CreateDefaultBuilder` 也会加载：

- appsettings.json 和 appsettings.{Environment}.json 的可选配置。
- 用户机密 (Secret Manager)（在开发环境中）。
- 环境变量。

`CreateDefaultBuilder` 最后添加命令行配置提供程序。在运行时传递的命令行参数会替代由其他提供程序设置的配置。

`CreateDefaultBuilder` 在构造主机时起作用。因此，`CreateDefaultBuilder` 激活的命令行配置可能会影响主机的配置方式。

构建主机时调用 ConfigureAppConfiguration 以指定应用的配置。

`CreateDefaultBuilder` 已经调用了 `AddCommandLine`。如果需要提供应用配置并仍然能够使用命令行参数覆盖该配置，请在 ConfigureAppConfiguration 中调用应用的其他提供程序并最后调用 `AddCommandLine`。

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                // Call other providers here and call AddCommandLine last.
                config.AddCommandLine(args);
            })
            .UseStartup<Startup>();
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

使用 UseConfiguration 方法将配置应用到 WebHostBuilder。

调用 `UseConfiguration` 时，`CreateDefaultBuilder` 已调用 `AddCommandLine`。如果需要提供应用配置并仍然能够使用命令行参数覆盖该配置，请在 `ConfigurationBuilder` 中调用应用的其他提供程序并最后调用 `AddCommandLine`。

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var config = new ConfigurationBuilder()
            // Call other providers here and call AddCommandLine last.
            .AddCommandLine(args)
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseConfiguration(config)
            .UseStartup<Startup>();
    }
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

要激活命令行配置，请在 ConfigurationBuilder 的实例上调用 AddCommandLine 扩展方法。

最后调用提供程序，以允许在运行时传递的命令行参数替代由其他配置提供程序设置的配置。

使用 UseConfiguration 方法将配置应用到 WebHostBuilder：

```
var config = new ConfigurationBuilder()
    // Call additional providers here as needed.
    // Call AddCommandLine last to allow arguments to override other configuration.
    .AddCommandLine(args)
    .Build();

var host = new WebHostBuilder()
    .UseConfiguration(config)
    .UseKestrel()
    .UseStartup<Startup>();
```

示例

2.x 示例应用利用静态便捷方法 `CreateDefaultBuilder` 来构建主机，其中包括对 AddCommandLine 的调用。

1.x 示例应用在 ConfigurationBuilder 上调用 AddCommandLine。

1. 在项目的目录中打开命令提示符。
2. 为 `dotnet run` 命令提供命令行参数 `dotnet run CommandLineKey=CommandLineValue`。
3. 应用运行后，在 `http://localhost:5000` 打开应用的浏览器。
4. 观察输出是否包含提供给 `dotnet run` 的配置命令行参数的键值对。

自变量

该值必须后跟一个等号（`=`），否则当值后跟一个空格时，键必须具有前缀（`--` 或 `/`）。如果使用等号（例如，`CommandLineKey=`），则该值可以为 null。

键前缀	示例
无前缀	`CommandLineKey1=value1`
双划线（`--`）	`--CommandLineKey2=value2`， `--CommandLineKey2 value2`
正斜杠（`/`）	`/CommandLineKey3=value3`， `/CommandLineKey3 value3`

在同一命令中，不要将使用等号的命令行参数键值对与使用空格的键值对混合使用。

示例命令：

```
dotnet run CommandLineKey1=value1 --CommandLineKey2=value2 /CommandLineKey3=value3
dotnet run --CommandLineKey1 value1 /CommandLineKey2 value2
dotnet run CommandLineKey1= CommandLineKey2=value2
```

交换映射

交换映射支持键名替换逻辑。使用 ConfigurationBuilder 手动构建配置时，可以为 AddCommandLine 方法提供交换替换字典。

当使用交换映射字典时，会检查字典中是否有与命令行参数提供的键匹配的键。如果在字典中找到命令行键，则传回字典值（键替换）以将键值对设置为应用的配置。对任何具有单划线（`-`）前缀的命令行键而言，交换映射都是必需的。

交换映射字典键规则：

- 交换必须以单划线（`-`）或双划线（`--`）开头。

- 交换映射字典不得包含重复键。

构建主机时调用 ConfigureAppConfiguration 以指定应用的配置：

```
public class Program
{
    public static readonly Dictionary<string, string> _switchMappings =
        new Dictionary<string, string>
        {
            { "-CLKey1", "CommandLineKey1" },
            { "-CLKey2", "CommandLineKey2" }
        };

    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    // Do not pass the args to CreateDefaultBuilder
    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder()
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddCommandLine(args, _switchMappings);
            })
            .UseStartup<Startup>();
}
```

如前面的示例所示，当使用交换映射时，对 `CreateDefaultBuilder` 的调用不应传递参数。`CreateDefaultBuilder` 方法的 `AddCommandLine` 调用不包括映射的交换，并且无法将交换映射字典传递给 `CreateDefaultBuilder`。如果参数包含映射的交换并传递给 `CreateDefaultBuilder`，则其 `AddCommandLine` 提供程序无法使用 FormatException 进行初始化。解决方案不是将参数传递给 `CreateDefaultBuilder`，而是允许 `ConfigurationBuilder` 方法的 `AddCommandLine` 方法处理参数和交换映射字典。

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var switchMappings = new Dictionary<string, string>
            {
                { "-CLKey1", "CommandLineKey1" },
                { "-CLKey2", "CommandLineKey2" }
            };

        var config = new ConfigurationBuilder()
            .AddCommandLine(args, switchMappings)
            .Build();

        // Do not pass the args to CreateDefaultBuilder
        return WebHost.CreateDefaultBuilder()
            .UseConfiguration(config)
            .UseStartup<Startup>();
    }
}
```

如前面的示例所示，当使用交换映射时，对 `CreateDefaultBuilder` 的调用不应传递参数。

`CreateDefaultBuilder` 方法的 `AddCommandLine` 调用不包括映射的交换，并且无法将交换映射字典传递给 `CreateDefaultBuilder`。如果参数包含映射的交换并传递给 `CreateDefaultBuilder`，则其 `AddCommandLine` 提供程序无法使用 FormatException 进行初始化。解决方案不是将参数传递给 `CreateDefaultBuilder`，而是允许 `ConfigurationBuilder` 方法的 `AddCommandLine` 方法处理参数和交换映射字典。

```
public static void Main(string[] args)
{
    var switchMappings = new Dictionary<string, string>
        {
            { "-CLKey1", "CommandLineKey1" },
            { "-CLKey2", "CommandLineKey2" }
        };

    var config = new ConfigurationBuilder()
        .AddCommandLine(args, switchMappings)
        .Build();

    var host = new WebHostBuilder()
        .UseConfiguration(config)
        .UseKestrel()
        .UseStartup<Startup>()
        .Start();

    using (host)
    {
        Console.ReadLine();
    }
}
```

创建交换映射字典后，它将包含下表所示的数据。

键	值
`-CLKey1`	`CommandLineKey1`
`-CLKey2`	`CommandLineKey2`

如果在启动应用时使用了交换映射的键，则配置将接收字典提供的密钥上的配置值：

```
dotnet run -CLKey1=value1 -CLKey2=value2
```

运行上述命令后，配置包含下表中显示的值。

键	值
`CommandLineKey1`	`value1`
`CommandLineKey2`	`value2`

环境变量配置提供程序

EnvironmentVariablesConfigurationProvider 在运行时从环境变量键值对加载配置。

要激活环境变量配置，请在 ConfigurationBuilder 的实例上调用 AddEnvironmentVariables 扩展方法。

在环境变量中使用分层键时，冒号分隔符（`:`）可能无法适用于所有平台。所有平台均支持采用双下划

线 (`__`)，并可以用冒号替换。

借助 Azure 应用服务，用户可以在 Azure 门户中设置使用环境变量配置提供程序替代应用配置的环境变量。有关详细信息，请参阅 Azure 应用：使用 Azure 门户替代应用配置。

初始化一个新的 WebHostBuilder 时，对于前缀为 `ASPNETCORE_` 的环境变量，会自动调用 `AddEnvironmentVariables`。有关详细信息，请参阅 Web 主机：设置主机。

此外，`CreateDefaultBuilder` 也会加载：

- 来自没有前缀的环境变量的应用配置，方法是通过调用不带前缀的 `AddEnvironmentVariables`。
- appsettings.json 和 appsettings.{Environment}.json 的可选配置。
- 用户机密 (Secret Manager)（在开发环境中）。
- 命令行参数。

环境变量配置提供程序是在配置已根据用户机密和 appsettings 文件建立后调用。在此位置调用提供程序允许在运行时读取的环境变量替代由用户机密和 appsettings 文件设置的配置。

构建主机时调用 ConfigureAppConfiguration 以指定应用的配置。

如果需要从其他环境变量提供应用配置，请在 ConfigureAppConfiguration 中调用应用的其他提供程序，并使用前缀调用 `AddEnvironmentVariables`。

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                // Call additional providers here as needed.
                // Call AddEnvironmentVariables last if you need to allow environment
                // variables to override values from other providers.
                config.AddEnvironmentVariables(prefix: "PREFIX_");
            })
            .UseStartup<Startup>();
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

在 ConfigurationBuilder 的实例上调用 `AddEnvironmentVariables` 扩展方法。使用 UseConfiguration 方法将配置应用到 WebHostBuilder。

如果需要从其他环境变量提供应用配置，请在 ConfigureAppConfiguration 中调用应用的其他提供程序，并使用前缀调用 `AddEnvironmentVariables`。

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var config = new ConfigurationBuilder()
            // Call additional providers here as needed.
            // Call AddEnvironmentVariables last if you need to allow environment
            // variables to override values from other providers.
            .AddEnvironmentVariables(prefix: "PREFIX_")
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseConfiguration(config)
            .UseStartup<Startup>();
    }
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

使用 UseConfiguration 方法将配置应用到 WebHostBuilder：

```
var config = new ConfigurationBuilder()
    .AddEnvironmentVariables()
    .Build();

var host = new WebHostBuilder()
    .UseConfiguration(config)
    .UseKestrel()
    .UseStartup<Startup>();
```

示例

2.x 示例应用利用静态便捷方法 `CreateDefaultBuilder` 来构建主机，其中包括对 `AddEnvironmentVariables` 的调用。

1.x 示例应用在 `ConfigurationBuilder` 上调用 `AddEnvironmentVariables`。

1. 运行示例应用。在 `http://localhost:5000` 打开应用的浏览器。
2. 观察输出是否包含环境变量 `ENVIRONMENT` 的键值对。该值反映了应用运行的环境，在本地运行时通常为 `Development`。

为了使应用呈现的环境变量列表简短，应用将环境变量筛选为以下列内容开头的变量：

- ASPNETCORE_
- urls
- 日志记录
- ENVIRONMENT
- contentRoot
- AllowedHosts
- applicationName
- CommandLine

如果希望公开应用可用的所有环境变量，请将 Pages/Index.cshtml.cs 中的 `FilteredConfiguration` 更改为以下内容：

如果希望公开应用可用的所有环境变量，请将 Controllers/HomeController.cs 中的 `FilteredConfiguration` 更改为以下内容：

```
FilteredConfiguration = _config.AsEnumerable();
```

前缀

为 `AddEnvironmentVariables` 方法提供前缀时，将筛选加载到应用的配置中的环境变量。例如，要筛选前缀 `CUSTOM_` 上的环境变量，请将前缀提供给配置提供程序：

```
var config = new ConfigurationBuilder()
    .AddEnvironmentVariables("CUSTOM_")
    .Build();
```

创建配置键值对时，将去除前缀。

静态便捷方法 `CreateDefaultBuilder` 创建一个 WebHostBuilder 以建立应用的主机。创建 `WebHostBuilder` 时，它会在前缀为 `ASPNETCORE_` 的环境变量中找到其主机配置。

连接字符串前缀

针对为应用环境配置 Azure 连接字符串所涉及的四个连接字符串环境变量，配置 API 具有特殊的处理规则。如果没有向 `AddEnvironmentVariables` 提供前缀，则具有表中所示前缀的环境变量将加载到应用中。

连接字符串前缀	提供程序
`CUSTOMCONNSTR_`	自定义提供程序
`MYSQLCONNSTR_`	MySQL
`SQLAZURECONNSTR_`	Azure SQL 数据库
`SQLCONNSTR_`	SQL Server

当发现环境变量并使用表中所示的四个前缀中的任何一个加载到配置中时：

- 通过删除环境变量前缀并添加配置键节（`ConnectionStrings`）来创建配置键。
- 创建一个新的配置键值对，表示数据库连接提供程序（`CUSTOMCONNSTR_` 除外，它没有声明的提供程序）。

环境变量键	转换的配置键	提供程序配置条目
`CUSTOMCONNSTR_<KEY>`	`ConnectionStrings:<KEY>`	配置条目未创建。
`MYSQLCONNSTR_<KEY>`	`ConnectionStrings:<KEY>`	键：`ConnectionStrings:<KEY>_ProviderName` ： 值：`MySql.Data.MySqlClient`

环境变量键	转换的配置键	提供程序配置条目
`SQLAZURECONNSTR_<KEY>`	`ConnectionStrings:<KEY>`	键：`ConnectionStrings:<KEY>_ProviderName` ： 值：`System.Data.SqlClient`
`SQLCONNSTR_<KEY>`	`ConnectionStrings:<KEY>`	键：`ConnectionStrings:<KEY>_ProviderName` ： 值：`System.Data.SqlClient`

文件配置提供程序

FileConfigurationProvider 是从文件系统加载配置的基类。以下配置提供程序专用于特定文件类型：

- INI 配置提供程序
- JSON 配置提供程序
- XML 配置提供程序

INI 配置提供程序

IniConfigurationProvider 在运行时从 INI 文件键值对加载配置。

若要激活 INI 文件配置，请在 ConfigurationBuilder 的实例上调用 AddIniFile 扩展方法。

冒号可用作 INI 文件配置中的节分隔符。

重载允许指定：

- 文件是否可选。
- 如果文件更改，是否重载配置。
- IFileProvider 用于访问该文件。

构建主机时调用 ConfigureAppConfiguration 以指定应用的配置：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.SetBasePath(Directory.GetCurrentDirectory());
                config.AddIniFile("config.ini", optional: true, reloadOnChange: true);
            })
            .UseStartup<Startup>();
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

调用 `CreateDefaultBuilder` 时，请使用以下配置调用 UseConfiguration：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddIniFile("config.ini", optional: true, reloadOnChange: true)
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseConfiguration(config)
            .UseStartup<Startup>();
    }
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

使用 UseConfiguration 方法将配置应用到 WebHostBuilder：

```
var config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddIniFile("config.ini", optional: true, reloadOnChange: true)
    .Build();

var host = new WebHostBuilder()
    .UseConfiguration(config)
    .UseKestrel()
    .UseStartup<Startup>();
```

INI 配置文件的通用示例：

```
[section0]
key0=value
key1=value

[section1]
subsection:key=value

[section2:subsection0]
key=value

[section2:subsection1]
key=value
```

以前的配置文件使用 `value` 加载以下键：

- section0:key0
- section0:key1
- section1:subsection:key
- section2:subsection0:key
- section2:subsection1:key

JSON 配置提供程序

JsonConfigurationProvider 在运行时期间从 JSON 文件键值对加载配置。

若要激活 JSON 文件配置，请在 ConfigurationBuilder 的实例上调用 AddJsonFile 扩展方法。

重载允许指定：

- 文件是否可选。
- 如果文件更改，是否重载配置。
- IFileProvider 用于访问该文件。

使用 CreateDefaultBuilder 初始化新的 WebHostBuilder 时，会自动调用 `AddJsonFile` 两次。调用该方法来从以下文件加载配置：

- appsettings.json – 首先读取此文件。该文件的环境版本可以替代 appsettings.json 文件提供的值。
- *appsettings.{Environment}.json*– 根据 IHostingEnvironment.EnvironmentName 加载文件的环境版本。

有关详细信息，请参阅 Web 主机：设置主机。

此外，`CreateDefaultBuilder` 也会加载：

- 环境变量。
- 用户机密 (Secret Manager)（在开发环境中）。
- 命令行参数。

首先建立 JSON 配置提供程序。因此，用户机密、环境变量和命令行参数会替代由 appsettings 文件设置的配置。

构建主机时调用 ConfigureAppConfiguration 以指定除 appsettings.json 和 appsettings.{Environment}.json 以外的文件的应用配置：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.SetBasePath(Directory.GetCurrentDirectory());
                config.AddJsonFile("config.json", optional: true, reloadOnChange: true);
            })
            .UseStartup<Startup>();
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

还可以在 ConfigurationBuilder 的实例上直接调用 `AddJsonFile` 扩展方法。

调用 `CreateDefaultBuilder` 时，请使用以下配置调用 UseConfiguration：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("config.json", optional: true, reloadOnChange: true)
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseConfiguration(config)
            .UseStartup<Startup>();
    }
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

使用 UseConfiguration 方法将配置应用到 WebHostBuilder：

```
var config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("config.json", optional: true, reloadOnChange: true)
    .Build();

var host = new WebHostBuilder()
    .UseConfiguration(config)
    .UseKestrel()
    .UseStartup<Startup>();
```

示例

2.x 示例应用利用静态便捷方法 `CreateDefaultBuilder` 来构建主机，其中包括对 `AddJsonFile` 的两次调用。配置从 appsettings.json 和 appsettings.{Environment}.json 进行加载。

1.x 示例应用在 `ConfigurationBuilder` 上调用 `AddJsonFile` 两次。配置从 appsettings.json 和 appsettings.{Environment}.json 进行加载。

1. 运行示例应用。在 `http://localhost:5000` 打开应用的浏览器。
2. 观察输出是否包含表中所示的配置的键值对，具体取决于环境。记录配置键使用冒号 (`:`) 作为分层分隔符。

键	开发值	生产值
Logging:LogLevel:System	信息	信息
Logging:LogLevel:Microsoft	信息	信息
Logging:LogLevel:Default	调试	Error
AllowedHosts	*	*

XML 配置提供程序

XmlConfigurationProvider 在运行时从 XML 文件键值对加载配置。

若要激活 XML 文件配置，请在 ConfigurationBuilder 的实例上调用 AddXmlFile 扩展方法。

重载允许指定：

- 文件是否可选。
- 如果文件更改，是否重载配置。
- IFileProvider 用于访问该文件。

创建配置键值对时，将忽略配置文件的根节点。不要在文件中指定文档类型定义 (DTD) 或命名空间。

构建主机时调用 ConfigureAppConfiguration 以指定应用的配置：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.SetBasePath(Directory.GetCurrentDirectory());
                config.AddXmlFile("config.xml", optional: true, reloadOnChange: true);
            })
            .UseStartup<Startup>();
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

调用 `CreateDefaultBuilder` 时，请使用以下配置调用 UseConfiguration：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddXmlFile("config.xml", optional: true, reloadOnChange: true)
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseConfiguration(config)
            .UseStartup<Startup>();
    }
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

使用 UseConfiguration 方法将配置应用到 WebHostBuilder：

```
var config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddXmlFile("config.xml", optional: true, reloadOnChange: true)
    .Build();

var host = new WebHostBuilder()
    .UseConfiguration(config)
    .UseKestrel()
    .UseStartup<Startup>();
```

XML 配置文件可以为重复节使用不同的元素名称：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <section0>
    <key0>value</key0>
    <key1>value</key1>
  </section0>
  <section1>
    <key0>value</key0>
    <key1>value</key1>
  </section1>
</configuration>
```

以前的配置文件使用 `value` 加载以下键：

- section0:key0
- section0:key1
- section1:key0
- section1:key1

如果使用 `name` 属性来区分元素，则使用相同元素名称的重复元素可以正常工作：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <section name="section0">
    <key name="key0">value</key>
    <key name="key1">value</key>
  </section>
  <section name="section1">
    <key name="key0">value</key>
    <key name="key1">value</key>
  </section>
</configuration>
```

以前的配置文件使用 `value` 加载以下键：

- section:section0:key:key0
- section:section0:key:key1
- section:section1:key:key0
- section:section1:key:key1

属性可用于提供值：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <key attribute="value" />
  <section>
    <key attribute="value" />
  </section>
</configuration>
```

以前的配置文件使用 `value` 加载以下键：

- key:attribute
- section:key:attribute

Key-per-file 配置提供程序

KeyPerFileConfigurationProvider 使用目录的文件作为配置键值对。该键是文件名。该值包含文件的内容。Key-per-file 配置提供程序用于 Docker 托管方案。

若要激活 Key-per-file 配置，请在 ConfigurationBuilder 的实例上调用 AddKeyPerFile 扩展方法。文件的 `directoryPath` 必须是绝对路径。

重载允许指定：

- 配置源的 `Action<KeyPerFileConfigurationSource>` 委托。
- 目录是否可选以及目录的路径。

双下划线字符（`__`）用作文件名中的配置键分隔符。例如，文件名 `Logging__LogLevel__System` 生成配置键 `Logging:LogLevel:System`。

构建主机时调用 ConfigureAppConfiguration 以指定应用的配置：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.SetBasePath(Directory.GetCurrentDirectory());
                var path = Path.Combine(Directory.GetCurrentDirectory(), "path/to/files");
                config.AddKeyPerFile(directoryPath: path, optional: true);
            })
            .UseStartup<Startup>();
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

```
var path = Path.Combine(Directory.GetCurrentDirectory(), "path/to/files");
var config = new ConfigurationBuilder()
    .AddKeyPerFile(directoryPath: path, optional: true)
    .Build();

var host = new WebHostBuilder()
    .UseConfiguration(config)
    .UseKestrel()
    .UseStartup<Startup>();
```

内存配置提供程序

MemoryConfigurationProvider 使用内存中集合作为配置键值对。

若要激活内存中集合配置，请在 ConfigurationBuilder 的实例上调用 AddInMemoryCollection 扩展方法。

可以使用 `IEnumerable<KeyValuePair<String,String>>` 初始化配置提供程序。

构建主机时调用 ConfigureAppConfiguration 以指定应用的配置：

```
public class Program
{
    public static readonly Dictionary<string, string> _dict =
        new Dictionary<string, string>
        {
            {"MemoryCollectionKey1", "value1"},
            {"MemoryCollectionKey2", "value2"}
        };

    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddInMemoryCollection(_dict);
            })
            .UseStartup<Startup>();
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

调用 `CreateDefaultBuilder` 时，请使用以下配置调用 UseConfiguration：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var dict = new Dictionary<string, string>
            {
                {"MemoryCollectionKey1", "value1"},
                {"MemoryCollectionKey2", "value2"}
            };

        var config = new ConfigurationBuilder()
            .AddInMemoryCollection(dict)
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseConfiguration(config)
            .UseStartup<Startup>();
    }
}
```

直接创建 WebHostBuilder 时，请使用以下配置调用 UseConfiguration：

使用 UseConfiguration 方法将配置应用到 WebHostBuilder：

```
var dict = new Dictionary<string, string>
    {
        {"MemoryCollectionKey1", "value1"},
        {"MemoryCollectionKey2", "value2"}
    };

var config = new ConfigurationBuilder()
    .AddInMemoryCollection(dict)
    .Build();

var host = new WebHostBuilder()
    .UseConfiguration(config)
    .UseKestrel()
    .UseStartup<Startup>();
```

GetValue

ConfigurationBinder.GetValue<T> 从具有指定键的配置中提取一个值，并将其转换为指定类型。如果未找到该键，则过载允许你提供默认值。

以下示例使用键 `NumberKey` 从配置中提取字符串值，键入该值作为 `int`，并将值存储在变量 `intValue` 中。如果在配置键中找不到 `NumberKey`，则 `intValue` 会接收 `99` 的默认值：

```
var intValue = config.GetValue<int>("NumberKey", 99);
```

GetSection、GetChildren 和 Exists

对于下面的示例，请考虑以下 JSON 文件。在两个节中找到四个键，其中一个包含一对子节：

```
{
  "section0": {
    "key0": "value",
    "key1": "value"
  },
  "section1": {
    "key0": "value",
    "key1": "value"
  },
  "section2": {
    "subsection0" : {
      "key0": "value",
      "key1": "value"
    },
    "subsection1" : {
      "key0": "value",
      "key1": "value"
    }
  }
}
```

将文件读入配置时，会创建以下唯一的分层键来保存配置值：

- section0:key0
- section0:key1
- section1:key0
- section1:key1
- section2:subsection0:key0
- section2:subsection0:key1
- section2:subsection1:key0
- section2:subsection1:key1

GetSection

IConfiguration.GetSection 使用指定的子节键提取配置子节。

若要返回仅包含 `section1` 中键值对的 IConfigurationSection，请调用 `GetSection` 并提供节名称：

```
var configSection = _config.GetSection("section1");
```

同样，若要获取 `section2:subsection0` 中键的值，请调用 `GetSection` 并提供节路径：

```
var configSection = _config.GetSection("section2:subsection0");
```

`GetSection` 永远不会返回 `null`。如果找不到匹配的节，则返回空 `IConfigurationSection`。

GetChildren

在 `section2` 上调用 IConfiguration.GetChildren 会获得 `IEnumerable<IConfigurationSection>`，其中包括：

- `subsection0`
- `subsection1`

```
var configSection = _config.GetSection("section2");

var children = configSection.GetChildren();
```

存在

使用 ConfigurationExtensions.Exists 确定配置节是否存在：

```
var sectionExists = _config.GetSection("section2:subsection2").Exists();
```

给定示例数据，`sectionExists` 为 `false`，因为配置数据中没有 `section2:subsection2` 节。

绑定至类

可以使用选项模式将配置绑定到表示相关设置组的类。有关更多信息，请参见ASP.NET Core 中的选项模式。

配置值作为字符串返回，但调用 Bind 可以构造 POCO 对象。

示例应用包含 `Starship` 模型 (Models/Starship.cs)：

```
public class Starship
{
    public string Name { get; set; }
    public string Registry { get; set; }
    public string Class { get; set; }
    public decimal Length { get; set; }
    public bool Commissioned { get; set; }
}
```


```
public class Starship
{
    public string Name { get; set; }
    public string Registry { get; set; }
    public string Class { get; set; }
    public decimal Length { get; set; }
    public bool Commissioned { get; set; }
}
```


当示例应用使用 JSON 配置提供程序加载配置时，starship.json 文件的 `starship` 节会创建配置：

```
{
  "starship": {
    "name": "USS Enterprise",
    "registry": "NCC-1701",
    "class": "Constitution",
    "length": 304.8,
    "commissioned": false
  },
  "trademark": "Paramount Pictures Corp. http://www.paramount.com"
}
```


```
{
  "starship": {
    "name": "USS Enterprise",
    "registry": "NCC-1701",
    "class": "Constitution",
    "length": 304.8,
    "commissioned": false
  },
  "trademark": "Paramount Pictures Corp. http://www.paramount.com"
}
```

创建以下配置键值对：

键	值
starship:name	USS Enterprise
starship:registry	NCC-1701
starship:class	Constitution
starship:length	304.8
starship:commissioned	False
trademark	Paramount Pictures Corp. http://www.paramount.com

示例应用使用 `starship` 键调用 `GetSection`。`starship` 键值对是独立的。在子节传入 `Starship` 类的实例时调用 `Bind` 方法。绑定实例值后，将实例分配给用于呈现的属性：

```
var starship = new Starship();
_config.GetSection("starship").Bind(starship);
Starship = starship;
```

```
var starship = new Starship();
_config.GetSection("starship").Bind(starship);
viewModel.Starship = starship;
```

绑定至对象图

Bind 能够绑定整个 POCO 对象图。

该示例包含 `TvShow` 模型，其对象图包含 `Metadata` 和 `Actors` 类 (Models/TvShow.cs)：

```
public class TvShow
{
    public Metadata Metadata { get; set; }
    public Actors Actors { get; set; }
    public string Legal { get; set; }
}

public class Metadata
{
    public string Series { get; set; }
    public string Title { get; set; }
    public DateTime AirDate { get; set; }
    public int Episodes { get; set; }
}

public class Actors
{
    public string Names { get; set; }
}
```

```
public class TvShow
{
    public Metadata Metadata { get; set; }
    public Actors Actors { get; set; }
    public string Legal { get; set; }
}

public class Metadata
{
    public string Series { get; set; }
    public string Title { get; set; }
    public DateTime AirDate { get; set; }
    public int Episodes { get; set; }
}

public class Actors
{
    public string Names { get; set; }
}
```

示例应用有一个包含配置数据的 tvshow.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <tvshow>
    <metadata>
      <series>Dr. Who</series>
      <title>The Sun Makers</title>
      <airdate>11/26/1977</airdate>
      <episodes>4</episodes>
    </metadata>
    <actors>
      <names>Tom Baker, Louise Jameson, John Leeson</names>
    </actors>
    <legal>(c)1977 BBC https://www.bbc.co.uk/programmes/b006q2x0</legal>
  </tvshow>
</configuration>
```


```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <tvshow>
    <metadata>
      <series>Dr. Who</series>
      <title>The Sun Makers</title>
      <airdate>11/26/1977</airdate>
      <episodes>4</episodes>
    </metadata>
    <actors>
      <names>Tom Baker, Louise Jameson, John Leeson</names>
    </actors>
    <legal>(c)1977 BBC https://www.bbc.co.uk/programmes/b006q2x0</legal>
  </tvshow>
</configuration>
```


使用 `Bind` 方法将配置绑定到整个 `TvShow` 对象图。将绑定实例分配给用于呈现的属性：

```
var tvShow = new TvShow();
_config.GetSection("tvshow").Bind(tvShow);
TvShow = tvShow;
```

```
var tvShow = new TvShow();
_config.GetSection("tvshow").Bind(tvShow);
viewModel.TvShow = tvShow;
```

ConfigurationBinder.Get<T> 绑定并返回指定类型。`Get<T>` 比使用 `Bind` 更方便。以下代码显示如何将 `Get<T>` 与前面的示例一起使用，该示例允许将绑定实例直接分配给用于呈现的属性：

```
TvShow = _config.GetSection("tvshow").Get<TvShow>();
```

```
viewModel.TvShow = _config.GetSection("tvshow").Get<TvShow>();
```

将数组绑定至类

示例应用演示了本部分中介绍的概念。

Bind 支持使用配置键中的数组索引将数组绑定到对象。公开数字键段（`:0:`、`:1:`、… `:{n}:`）的任何数组格式都能够与 POCO 类数组进行数组绑定。

> **NOTE**
> 绑定是按约定提供的。不需要自定义配置提供程序实现数组绑定。

内存中数组处理

请考虑下表中所示的配置键和值。

键	值
array:entries:0	value0
array:entries:1	value1
array:entries:2	value2
array:entries:4	value4
array:entries:5	value5

使用内存配置提供程序在示例应用中加载这些键和值：

```
public class Program
{
    public static Dictionary<string, string> arrayDict = new Dictionary<string, string>
        {
            {"array:entries:0", "value0"},
            {"array:entries:1", "value1"},
            {"array:entries:2", "value2"},
            {"array:entries:4", "value4"},
            {"array:entries:5", "value5"}
        };

    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.SetBasePath(Directory.GetCurrentDirectory());
                config.AddInMemoryCollection(arrayDict);
                config.AddJsonFile("json_array.json", optional: false, reloadOnChange: false);
                config.AddJsonFile("starship.json", optional: false, reloadOnChange: false);
                config.AddXmlFile("tvshow.xml", optional: false, reloadOnChange: false);
                config.AddEFConfiguration(options => options.UseInMemoryDatabase("InMemoryDb"));
                config.AddCommandLine(args);
            })
            .UseStartup<Startup>();
}
```

```
public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        var arrayDict = new Dictionary<string, string>
        {
            {"array:entries:0", "value0"},
            {"array:entries:1", "value1"},
            {"array:entries:2", "value2"},
            {"array:entries:4", "value4"},
            {"array:entries:5", "value5"}
        };

        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddInMemoryCollection(arrayDict)
            .AddJsonFile("json_array.json", optional: false, reloadOnChange: false)
            .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true,
                reloadOnChange: true)
            .AddEnvironmentVariables()
            .AddJsonFile("starship.json", optional: false, reloadOnChange: false)
            .AddXmlFile("tvshow.xml", optional: false, reloadOnChange: false)
            .AddEFConfiguration(options => options.UseInMemoryDatabase("InMemoryDb"))
            .AddCommandLine(Program.Args);

        Configuration = builder.Build();
    }
```

该数组跳过索引 #3 的值。配置绑定程序无法绑定 null 值，也无法在绑定对象中创建 null 条目，这在演示将此数组绑定到对象的结果时变得清晰。

在示例应用中，POCO 类可用于保存绑定的配置数据：

```
public class ArrayExample
{
    public string[] Entries { get; set; }
}
```


```
public class ArrayExample
{
    public string[] Entries { get; set; }
}
```


将配置数据绑定至对象：

```
var arrayExample = new ArrayExample();
_config.GetSection("array").Bind(arrayExample);
```

还可以使用 ConfigurationBinder.Get<T> 语法，从而产生更精简的代码：

```
ArrayExample = _config.GetSection("array").Get<ArrayExample>();
```

```
viewModel.ArrayExample = _config.GetSection("array").Get<ArrayExample>();
```

绑定对象（`ArrayExample` 的实例）从配置接收数组数据。

`ARRAYEXAMPLE.ENTRIES` 索引	`ARRAYEXAMPLE.ENTRIES` 值
0	value0
1	value1
2	value2
3	value4
4	value5

绑定对象中的索引 #3 保留 `array:4` 配置键的配置数据及其值 `value4`。当绑定包含数组的配置数据时，配置键中的数组索引仅用于在创建对象时迭代配置数据。无法在配置数据中保留 null 值，并且当配置键中的数组跳过一个或多个索引时，不会在绑定对象中创建 null 值条目。

可以在由任何在配置中生成正确键值对的配置提供程序绑定到 `ArrayExample` 实例之前提供索引 #3 的缺失配置项。如果示例包含具有缺失键值对的其他 JSON 配置提供程序，则 `ArrayExample.Entries` 与完整配置数组相匹配：

missing_value.json:

```
{
  "array:entries:3": "value3"
}
```

在 ConfigureAppConfiguration中：

```
config.AddJsonFile("missing_value.json", optional: false, reloadOnChange: false);
```

在 `Startup` 构造函数中：

```
.AddJsonFile("missing_value.json", optional: false, reloadOnChange: false);
```

将表中所示的键值对加载到配置中。

键	值
array:entries:3	value3

如果在 JSON 配置提供程序包含索引 #3 的条目之后绑定 `ArrayExample` 类实例，则 `ArrayExample.Entries` 数组包含该值。

`ARRAYEXAMPLE.ENTRIES` 索引	`ARRAYEXAMPLE.ENTRIES` 值
0	value0
1	value1
2	value2
3	value3
4	value4
5	value5

JSON 数组处理

如果 JSON 文件包含数组，则会为具有从零开始的节索引的数组元素创建配置键。在以下配置文件中，`subsection` 是一个数组：

```
{
  "json_array": {
    "key": "valueA",
    "subsection": [
      "valueB",
      "valueC",
      "valueD"
    ]
  }
}
```

```
{
    "json_array": {
      "key": "valueA",
      "subsection": [
        "valueB",
        "valueC",
        "valueD"
      ]
    }
  }
```

JSON 配置提供程序将配置数据读入以下键值对：

键	值
json_array:key	valueA
json_array:subsection:0	valueB
json_array:subsection:1	valueC
json_array:subsection:2	valueD

在示例应用中，以下 POCO 类可用于绑定配置键值对：

```
public class JsonArrayExample
{
    public string Key { get; set; }
    public string[] Subsection { get; set; }
}
```


```
public class JsonArrayExample
{
    public string Key { get; set; }
    public string[] Subsection { get; set; }
}
```


绑定后，`JsonArrayExample.Key` 保存值 `valueA`。子节值存储在 POCO 数组属性 `Subsection` 中。

`JSONARRAYEXAMPLE.SUBSECTION` 索引	`JSONARRAYEXAMPLE.SUBSECTION` 值
0	valueB
1	valueC
2	valueD

自定义配置提供程序

该示例应用演示了如何使用实体框架 (EF) 创建从数据库读取配置键值对的基本配置提供程序。

提供程序具有以下特征：

- EF 内存中数据库用于演示目的。若要使用需要连接字符串的数据库，请实现辅助

`ConfigurationBuilder` 以从另一个配置提供程序提供连接字符串。

- 提供程序在启动时将数据库表读入配置。提供程序不会基于每个键查询数据库。
- 未实现更改时重载，因此在应用启动后更新数据库对应用的配置没有任何影响。

定义用于在数据库中存储配置值的 `EFConfigurationValue` 实体。

Models/EFConfigurationValue.cs：

```
public class EFConfigurationValue
{
    public string Id { get; set; }
    public string Value { get; set; }
}
```


```
public class EFConfigurationValue
{
    public string Id { get; set; }
    public string Value { get; set; }
}
```


添加 `EFConfigurationContext` 以存储和访问配置的值。

EFConfigurationProvider/EFConfigurationContext.cs：

```
public class EFConfigurationContext : DbContext
{
    public EFConfigurationContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<EFConfigurationValue> Values { get; set; }
}
```


```
public class EFConfigurationContext : DbContext
{
    public EFConfigurationContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<EFConfigurationValue> Values { get; set; }
}
```


创建用于实现 IConfigurationSource 的类。

EFConfigurationProvider/EFConfigurationSource.cs：

```
public class EFConfigurationSource : IConfigurationSource
{
    private readonly Action<DbContextOptionsBuilder> _optionsAction;

    public EFConfigurationSource(Action<DbContextOptionsBuilder> optionsAction)
    {
        _optionsAction = optionsAction;
    }

    public IConfigurationProvider Build(IConfigurationBuilder builder)
    {
        return new EFConfigurationProvider(_optionsAction);
    }
}
```


```
public class EFConfigurationSource : IConfigurationSource
{
    private readonly Action<DbContextOptionsBuilder> _optionsAction;

    public EFConfigurationSource(Action<DbContextOptionsBuilder> optionsAction)
    {
        _optionsAction = optionsAction;
    }

    public IConfigurationProvider Build(IConfigurationBuilder builder)
    {
        return new EFConfigurationProvider(_optionsAction);
    }
}
```


通过从 ConfigurationProvider 继承来创建自定义配置提供程序。当数据库为空时，配置提供程序将对其进行初始化。

EFConfigurationProvider/EFConfigurationProvider.cs：

```
public class EFConfigurationProvider : ConfigurationProvider
{
    public EFConfigurationProvider(Action<DbContextOptionsBuilder> optionsAction)
    {
        OptionsAction = optionsAction;
    }

    Action<DbContextOptionsBuilder> OptionsAction { get; }

    // Load config data from EF DB.
    public override void Load()
    {
        var builder = new DbContextOptionsBuilder<EFConfigurationContext>();

        OptionsAction(builder);

        using (var dbContext = new EFConfigurationContext(builder.Options))
        {
            dbContext.Database.EnsureCreated();

            Data = !dbContext.Values.Any()
                ? CreateAndSaveDefaultValues(dbContext)
                : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
        }
    }

    private static IDictionary<string, string> CreateAndSaveDefaultValues(
        EFConfigurationContext dbContext)
    {
        // Quotes (c)2005 Universal Pictures: Serenity
        // https://www.uphe.com/movies/serenity
        var configValues = new Dictionary<string, string>
            {
                { "quote1", "I aim to misbehave." },
                { "quote2", "I swallowed a bug." },
                { "quote3", "You can't stop the signal, Mal." }
            };

        dbContext.Values.AddRange(configValues
            .Select(kvp => new EFConfigurationValue
                {
                    Id = kvp.Key,
                    Value = kvp.Value
                })
            .ToArray());

        dbContext.SaveChanges();

        return configValues;
    }
}
```

```
public class EFConfigurationProvider : ConfigurationProvider
{
    public EFConfigurationProvider(Action<DbContextOptionsBuilder> optionsAction)
    {
        OptionsAction = optionsAction;
    }

    Action<DbContextOptionsBuilder> OptionsAction { get; }

    // Load config data from EF DB.
    public override void Load()
    {
        var builder = new DbContextOptionsBuilder<EFConfigurationContext>();

        OptionsAction(builder);

        using (var dbContext = new EFConfigurationContext(builder.Options))
        {
            dbContext.Database.EnsureCreated();

            Data = !dbContext.Values.Any()
                ? CreateAndSaveDefaultValues(dbContext)
                : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
        }
    }

    private static IDictionary<string, string> CreateAndSaveDefaultValues(
        EFConfigurationContext dbContext)
    {
        // Quotes (c)2005 Universal Pictures: Serenity
        // https://www.uphe.com/movies/serenity
        var configValues = new Dictionary<string, string>
            {
                { "quote1", "I aim to misbehave." },
                { "quote2", "I swallowed a bug." },
                { "quote3", "You can't stop the signal, Mal." }
            };

        dbContext.Values.AddRange(configValues
            .Select(kvp => new EFConfigurationValue
                {
                    Id = kvp.Key,
                    Value = kvp.Value
                })
            .ToArray());

        dbContext.SaveChanges();

        return configValues;
    }
}
```

可以使用 `AddEFConfiguration` 扩展方法将配置源添加到 `ConfigurationBuilder`。

Extensions/EntityFrameworkExtensions.cs：

```
public static class EntityFrameworkExtensions
{
    public static IConfigurationBuilder AddEFConfiguration(
        this IConfigurationBuilder builder,
        Action<DbContextOptionsBuilder> optionsAction)
    {
        return builder.Add(new EFConfigurationSource(optionsAction));
    }
}
```


```
public static class EntityFrameworkExtensions
{
    public static IConfigurationBuilder AddEFConfiguration(
        this IConfigurationBuilder builder,
        Action<DbContextOptionsBuilder> optionsAction)
    {
        return builder.Add(new EFConfigurationSource(optionsAction));
    }
}
```


下面的代码演示如何在 Program.cs 中使用自定义的 `EFConfigurationProvider`:

```
public class Program
{
    public static Dictionary<string, string> arrayDict = new Dictionary<string, string>
        {
            {"array:entries:0", "value0"},
            {"array:entries:1", "value1"},
            {"array:entries:2", "value2"},
            {"array:entries:4", "value4"},
            {"array:entries:5", "value5"}
        };

    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.SetBasePath(Directory.GetCurrentDirectory());
                config.AddInMemoryCollection(arrayDict);
                config.AddJsonFile("json_array.json", optional: false, reloadOnChange: false);
                config.AddJsonFile("starship.json", optional: false, reloadOnChange: false);
                config.AddXmlFile("tvshow.xml", optional: false, reloadOnChange: false);
                config.AddEFConfiguration(options => options.UseInMemoryDatabase("InMemoryDb"));
                config.AddCommandLine(args);
            })
            .UseStartup<Startup>();
}
```

```
public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        var arrayDict = new Dictionary<string, string>
        {
            {"array:entries:0", "value0"},
            {"array:entries:1", "value1"},
            {"array:entries:2", "value2"},
            {"array:entries:4", "value4"},
            {"array:entries:5", "value5"}
        };

        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddInMemoryCollection(arrayDict)
            .AddJsonFile("json_array.json", optional: false, reloadOnChange: false)
            .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true,
                reloadOnChange: true)
            .AddEnvironmentVariables()
            .AddJsonFile("starship.json", optional: false, reloadOnChange: false)
            .AddXmlFile("tvshow.xml", optional: false, reloadOnChange: false)
            .AddEFConfiguration(options => options.UseInMemoryDatabase("InMemoryDb"))
            .AddCommandLine(Program.Args);

        Configuration = builder.Build();
    }
```

在启动期间访问配置

将 `IConfiguration` 注入 `Startup` 构造函数以访问 `Startup.ConfigureServices` 中的配置值。若要访问 `Startup.Configure` 中的配置，请将 `IConfiguration` 直接注入方法或使用构造函数中的实例：

```
public class Startup
{
    private readonly IConfiguration _config;

    public Startup(IConfiguration config)
    {
        _config = config;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        var value = _config["key"];
    }

    public void Configure(IApplicationBuilder app, IConfiguration config)
    {
        var value = config["key"];
    }
}
```

有关使用启动便捷方法访问配置的示例，请参阅应用启动：便捷方法。

在 Razor Pages 页或 MVC 视图中访问配置

若要访问 Razor Pages 页或 MVC 视图中的配置设置，请为 Microsoft.Extensions.Configuration 命名空间添加 using 指令（C# 参考：using 指令）并将 IConfiguration 注入页面或视图。

在 Razor 页面页中：

```
@page
@model IndexModel
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Index Page</title>
</head>
<body>
    <h1>Access configuration in a Razor Pages page</h1>
    <p>Configuration value for 'key': @Configuration["key"]</p>
</body>
</html>
```

在 MVC 视图中：

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Index View</title>
</head>
<body>
    <h1>Access configuration in an MVC view</h1>
    <p>Configuration value for 'key': @Configuration["key"]</p>
</body>
</html>
```

从外部程序集添加配置

通过 IHostingStartup 实现，可在启动时从应用 `Startup` 类之外的外部程序集向应用添加增强功能。有关更多信息，请参见在 ASP.NET Core 中使用承载启动程序集。

其他资源

- ASP.NET Core 中的选项模式
- 深入了解 Microsoft 配置

ASP.NET Core 中的选项模式

2019/1/10 • • Edit Online

作者：Luke Latham

对于本主题的 1.1 版本，请下载 ASP.NET Core（版本 1.1，PDF）中的选项模式。

选项模式使用类来表示相关设置的组。当配置设置由方案隔离到单独的类时，应用遵循两个重要软件工程原则：

- 接口分离原则 (ISP) 或封装 – 依赖于配置设置的方案（类）仅依赖于其使用的配置设置。
- 关注点分离 – 应用的不同部件的设置不彼此依赖或相互耦合。

选项还提供验证配置数据的机制。有关详细信息，请参阅选项验证部分。

查看或下载示例代码（如何下载）

系统必备

引用 Microsoft.AspNetCore.App 元包或将包引用添加到 Microsoft.Extensions.Options.ConfigurationExtensions 包。

选项接口

IOptionsMonitor<TOptions> 用于检索选项并管理 `TOptions` 实例的选项通知。IOptionsMonitor<TOptions> 支持以下方案：

- 更改通知
- 命名选项
- 可重载配置
- 选择性选项失效 (IOptionsMonitorCache<TOptions>)

后期配置方案允许你在进行所有 IConfigureOptions<TOptions> 配置后设置或更改选项。

IOptionsFactory<TOptions> 负责新建选项实例。它具有单个 Create 方法。默认实现采用所有已注册 IConfigureOptions<TOptions> 和 IPostConfigureOptions<TOptions> 并首先运行所有配置，然后才进行后期配置。它区分 IConfigureNamedOptions<TOptions> 和 IConfigureOptions<TOptions> 且仅调用适当的接口。

IOptionsMonitorCache<TOptions> 由 IOptionsMonitor<TOptions> 用于缓存 `TOptions` 实例。IOptionsMonitorCache<TOptions> 可使监视器中的选项实例无效，以便重新计算值 (TryRemove)。可以通过 TryAdd 手动引入值。在应按需重新创建所有命名实例时使用 Clear 方法。

IOptionsSnapshot<TOptions> 在每次请求时应重新计算选项的方案中有用。有关详细信息，请参阅通过 IOptionsSnapshot 重新加载配置数据部分。

IOptions<TOptions> 可用于支持选项。但是，IOptions<TOptions> 不支持前面的 IOptionsMonitor<TOptions> 方案。你可以在已使用 IOptions<TOptions> 接口的现有框架和库中继续使用 IOptions<TOptions>，并且不需要 IOptionsMonitor<TOptions> 提供的方案。

常规选项配置

常规选项配置已作为示例 #1 在示例应用中进行了演示。

选项类必须为包含公共无参数构造函数的非抽象类。以下类 `MyOptions` 具有两种属性：`Option1` 和 `Option2`。

设置默认值为可选，但以下示例中的类构造函数设置了 `Option1` 的默认值。`Option2` 具有通过直接初始化属性设置的默认值 (Models/MyOptions.cs)：

```
public class MyOptions
{
    public MyOptions()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }

    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}
```

`MyOptions` 类已通过 Configure 添加到服务容器并绑定到配置：

```
// Example #1: General configuration
// Register the Configuration instance which MyOptions binds against.
services.Configure<MyOptions>(Configuration);
```

以下页面模型通过 IOptionsMonitor<TOptions> 使用构造函数依赖关系注入来访问设置 (Pages/Index.cshtml.cs)：

```
private readonly MyOptions _options;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #1: Simple options
var option1 = _options.Option1;
var option2 = _options.Option2;
SimpleOptions = $"option1 = {option1}, option2 = {option2}";
```

示例的 appsettings.json 文件指定 `option1` 和 `option2` 的值：

```
{
  "option1": "value1_from_json",
  "option2": -1,
  "subsection": {
    "suboption1": "subvalue1_from_json",
    "suboption2": 200
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

运行应用时，页面模型的 `OnGet` 方法返回显示选项类值的字符串：

```
option1 = value1_from_json, option2 = -1
```

NOTE

使用自定义 ConfigurationBuilder 从设置文件加载选项配置时，请确认基路径设置正确：

```
var configBuilder = new ConfigurationBuilder()
   .SetBasePath(Directory.GetCurrentDirectory())
   .AddJsonFile("appsettings.json", optional: true);
var config = configBuilder.Build();

services.Configure<MyOptions>(config);
```

通过 CreateDefaultBuilder 从设置文件加载选项配置时，不需要显式设置基路径。

通过委托配置简单选项

通过委托配置简单选项已作为示例 #2 在示例应用中进行了演示。

使用委托设置选项值。此示例应用使用 `MyOptionsWithDelegateConfig` 类 (Models/MyOptionsWithDelegateConfig.cs)：

```
public class MyOptionsWithDelegateConfig
{
    public MyOptionsWithDelegateConfig()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }

    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}
```

在以下代码中，已向服务容器添加第二个 IConfigureOptions<TOptions> 服务。它通过 `MyOptionsWithDelegateConfig` 使用委托来配置绑定：

```
// Example #2: Options bound and configured by a delegate
services.Configure<MyOptionsWithDelegateConfig>(myOptions =>
{
    myOptions.Option1 = "value1_configured_by_delegate";
    myOptions.Option2 = 500;
});
```

Index.cshtml.cs:

```
private readonly MyOptionsWithDelegateConfig _optionsWithDelegateConfig;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #2: Options configured by delegate
var delegate_config_option1 = _optionsWithDelegateConfig.Option1;
var delegate_config_option2 = _optionsWithDelegateConfig.Option2;
SimpleOptionsWithDelegateConfig =
    $"delegate_option1 = {delegate_config_option1}, " +
    $"delegate_option2 = {delegate_config_option2}";
```

可添加多个配置提供程序。配置提供程序可从 NuGet 包中获取，并按照注册的顺序应用。有关更多信息，请参见ASP.NET Core 中的配置。

每次调用 Configure 都会将 IConfigureOptions<TOptions> 服务添加到服务容器。在前面的示例中，`Option1` 和 `Option2` 的值同时在 appsettings.json 中指定，但 `Option1` 和 `Option2` 的值被配置的委托替代。

当启用多个配置服务时，指定的最后一个配置源优于其他源，由其设置配置值。运行应用时，页面模型的 `OnGet` 方法返回显示选项类值的字符串：

```
delegate_option1 = value1_configured_by_delgate, delegate_option2 = 500
```

子选项配置

子选项配置已作为示例 #3 在示例应用中进行了演示。

应用应创建适用于应用中特定方案组(类)的选项类。需要配置值的部分应用应仅有权访问其使用的配置值。

将选项绑定到配置时，选项类型中的每个属性都将绑定到窗体 `property[:sub-property:]` 的配置键。例如，`MyOptions.Option1` 属性将绑定到从 appsettings.json 中的 `option1` 属性读取的键 `Option1`。

在以下代码中，已向服务容器添加第三个 IConfigureOptions<TOptions> 服务。它将 `MySubOptions` 绑定到 appsettings.json 文件的 `subsection` 部分：

```
// Example #3: Suboptions
// Bind options using a sub-section of the appsettings.json file.
services.Configure<MySubOptions>(Configuration.GetSection("subsection"));
```

`GetSection` 扩展方法需要 Microsoft.Extensions.Options.ConfigurationExtensions NuGet 包。如果应用使用 Microsoft.AspNetCore.App metapackage 元包（ASP.NET Core 2.1 或更高版本），将自动包含此包。

示例的 appsettings.json 文件定义具有 `suboption1` 和 `suboption2` 的键的 `subsection` 成员：

```
{
  "option1": "value1_from_json",
  "option2": -1,
  "subsection": {
    "suboption1": "subvalue1_from_json",
    "suboption2": 200
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

`MySubOptions` 类将属性 `SubOption1` 和 `SubOption2` 定义为保留选项值 (Models/MySubOptions.cs)：

```
public class MySubOptions
{
    public MySubOptions()
    {
        // Set default values.
        SubOption1 = "value1_from_ctor";
        SubOption2 = 5;
    }

    public string SubOption1 { get; set; }
    public int SubOption2 { get; set; }
}
```

页面模型的 `OnGet` 方法返回包含选项值的字符串 (Pages/Index.cshtml.cs)：

```
private readonly MySubOptions _subOptions;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #3: Suboptions
var subOption1 = _subOptions.SubOption1;
var subOption2 = _subOptions.SubOption2;
SubOptions = $"subOption1 = {subOption1}, subOption2 = {subOption2}";
```

运行应用时，`OnGet` 方法返回显示子选项类值的字符串：

```
subOption1 = subvalue1_from_json, subOption2 = 200
```

视图模型或通过直接视图注入提供的选项

视图模型或通过直接视图注入提供的选项已作为示例 #4 在示例应用中进行了演示。

可在视图模型中或通过将 IOptionsMonitor<TOptions> 直接注入到视图 (Pages/Index.cshtml.cs) 来提供选项：

```
private readonly MyOptions _options;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #4: Bind options directly to the page
MyOptions = _options;
```

示例应用演示如何使用 `@inject` 指令注入 `IOptionsMonitor<MyOptions>`：

```
@page
@model IndexModel
@using Microsoft.Extensions.Options
@inject IOptionsMonitor<MyOptions> OptionsAccessor
@{
    ViewData["Title"] = "Options Sample";
}

<h1>@ViewData["Title"]</h1>
```

当应用运行时，选项值显示在呈现的页面中：

Example #4: Model and injected options

Options provided by the model

Options provided by the model: `@Model.MyOptions.Option1` and `@Model.MyOptions.Option2`

Option1: value1_from_json

Option2: -1

Options injected into the page

Options injected into the page: `@inject IOptions<MyOptions> OptionsAccessor` with `@OptionsAccessor.Value.Option1` and `@OptionsAccessor.Value.Option2`

Option1: value1_from_json

Option2: -1

通过 IOptionsSnapshot 重新加载配置数据

通过 IOptionsSnapshot<TOptions> 重新加载配置数据已作为示例 #5 在示例应用中进行了演示。

IOptionsSnapshot<TOptions> 支持包含最小处理开销的重新加载选项。

针对请求生存期访问和缓存选项时，每个请求只能计算一次选项。

以下示例演示如何在更改 appsettings.json (Pages/Index.cshtml.cs) 后创建新的 IOptionsSnapshot<TOptions>。在更改文件和重新加载配置之前，针对服务器的多个请求返回 appsettings.json 文件提供的常数值。

```
private readonly MyOptions _snapshotOptions;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #5: Snapshot options
var snapshotOption1 = _snapshotOptions.Option1;
var snapshotOption2 = _snapshotOptions.Option2;
SnapshotOptions =
    $"snapshot option1 = {snapshotOption1}, " +
    $"snapshot option2 = {snapshotOption2}";
```

下图显示从 appsettings.json 文件加载的初始 `option1` 和 `option2` 值：

```
snapshot option1 = value1_from_json, snapshot option2 = -1
```

将 appsettings.json 文件中的值更改为 `value1_from_json UPDATED` 和 `200`。保存 appsettings.json 文件。刷新浏览器，查看更新的选项值：

```
snapshot option1 = value1_from_json UPDATED, snapshot option2 = 200
```

包含 IConfigureNamedOptions 的命名选项支持

包含 IConfigureNamedOptions<TOptions> 的命名选项支持已作为示例 #6 在示例应用中进行了演示。

命名选项支持允许应用在命名选项配置之间进行区分。在示例应用中，命名选项是使用 Configure 声明的。`Configure` 调用了扩展方法 Configure 方法：

```
// Example #6: Named options (named_options_1)
// Register the ConfigurationBuilder instance which MyOptions binds against.
// Specify that the options loaded from configuration are named
// "named_options_1".
services.Configure<MyOptions>("named_options_1", Configuration);

// Example #6: Named options (named_options_2)
// Specify that the options loaded from the MyOptions class are named
// "named_options_2".
// Use a delegate to configure option values.
services.Configure<MyOptions>("named_options_2", myOptions =>
{
    myOptions.Option1 = "named_options_2_value1_from_action";
});
```

示例应用通过 Get (Pages/Index.cshtml.cs) 访问命名选项：

```
private readonly MyOptions _named_options_1;
private readonly MyOptions _named_options_2;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #6: Named options
var named_options_1 =
    $"named_options_1: option1 = {_named_options_1.Option1}, " +
    $"option2 = {_named_options_1.Option2}";
var named_options_2 =
    $"named_options_2: option1 = {_named_options_2.Option1}, " +
    $"option2 = {_named_options_2.Option2}";
NamedOptions = $"{named_options_1} {named_options_2}";
```

运行示例应用，将返回命名选项：

```
named_options_1: option1 = value1_from_json, option2 = -1
named_options_2: option1 = named_options_2_value1_from_action, option2 = 5
```

从配置中提供从 appsettings.json 文件中加载的 `named_options_1` 值。通过以下内容提供 `named_options_2` 值：

- 针对 `Option1` 的 `ConfigureServices` 中的 `named_options_2` 委托。
- `MyOptions` 类提供的 `Option2` 的默认值。

使用 ConfigureAll 方法配置所有选项

使用 ConfigureAll 方法配置所有选项实例。以下代码将针对包含公共值的所有配置实例配置 `Option1`。将以下代码手动添加到 `Startup.ConfigureServices` 方法：

```
services.ConfigureAll<MyOptions>(myOptions =>
{
    myOptions.Option1 = "ConfigureAll replacement value";
});
```

添加代码后运行示例应用将产生以下结果：

```
named_options_1: option1 = ConfigureAll replacement value, option2 = -1
named_options_2: option1 = ConfigureAll replacement value, option2 = 5
```

> **NOTE**
>
> 所有选项都是命名实例。现有 IConfigureOptions<TOptions> 实例将被视为面向为 `string.Empty` 的 `Options.DefaultName` 实例。IConfigureNamedOptions<TOptions> 还可实现 IConfigureOptions<TOptions>。IOptionsFactory<TOptions> 的默认实现具有适当地使用每个实例的逻辑。`null` 命名选项用于面向所有命名实例而不是某一特定命名实例（ConfigureAll 和 PostConfigureAll 使用此约定）。

OptionsBuilder API

OptionsBuilder<TOptions> 用于配置 `TOptions` 实例。`OptionsBuilder` 简化了创建命名选项的过程，因为它只是初始 `AddOptions<TOptions>(string optionsName)` 调用的单个参数，而不会出现在所有后续调用中。选项验证和接受服务依赖关系的 `ConfigureOptions` 重载仅可通过 `OptionsBuilder` 获得。

```
// Options.DefaultName = "" is used.
services.AddOptions<MyOptions>().Configure(o => o.Property = "default");

services.AddOptions<MyOptions>("optionalName")
    .Configure(o => o.Property = "named");
```

配置 <TOptions、TDep1、...TDep4> 方法

使用 DI 中的服务，通过以样本方式实现 `IConfigure[Named]Options` 来配置选项的方法非常详细。`OptionsBuilder<TOptions>` 上 `ConfigureOptions` 的重载允许使用最多五个服务来配置选项：

```
services.AddOptions<MyOptions>("optionalName")
    .Configure<Service1, Service2, Service3, Service4, Service5>(
        (o, s, s2, s3, s4, s5) =>
            o.Property = DoSomethingWith(s, s2, s3, s4, s5));
```

重载会注册临时通用 IConfigureNamedOptions<TOptions>，它具有接受指定的通用服务类型的构造函数。

选项验证

借助选项验证，可以验证已配置的选项。使用验证方法调用 `Validate`。如果选项有效，方法返回 `true`；如果无效，方法返回 `false`：

```
// Registration
services.AddOptions<MyOptions>("optionalOptionsName")
    .Configure(o => { }) // Configure the options
    .Validate(o => YourValidationShouldReturnTrueIfValid(o),
        "custom error");

// Consumption
var monitor = services.BuildServiceProvider()
    .GetService<IOptionsMonitor<MyOptions>>();

try
{
    var options = monitor.Get("optionalOptionsName");
}
catch (OptionsValidationException e)
{
   // e.OptionsName returns "optionalOptionsName"
   // e.OptionsType returns typeof(MyOptions)
   // e.Failures returns a list of errors, which would contain
   //     "custom error"
}
```

上面的示例将命名选项实例设置为 `optionalOptionsName`。默认选项实例为 `Options.DefaultName`。

选项验证在选项实例创建后运行。系统保证在选项实例首次获得访问时通过验证。

> **IMPORTANT**
>
> 选项验证无法防止在最初配置和验证选项后发生选项修改。

`Validate` 方法接受 `Func<TOptions, bool>`。若要完全自定义验证，请实现 `IValidateOptions<TOptions>`，它支持：

- 验证多种类型的选项：`class ValidateTwo : IValidateOptions<Option1>, IValidationOptions<Option2>`
- 验证取决于其他选项类型：`public DependsOnAnotherOptionValidator(IOptionsMonitor<AnotherOption> options)`

`IValidateOptions` 验证：

- 特定的命名选项实例。
- 所有选项（如果 `name` 为 `null` 的话）。

通过接口实现返回 `ValidateOptionsResult`：

```
public interface IValidateOptions<TOptions> where TOptions : class
{
    ValidateOptionsResult Validate(string name, TOptions options);
}
```

通过调用 `OptionsBuilder<TOptions>` 上的 `ValidateDataAnnotations` 方法，可以从 Microsoft.Extensions.Options.DataAnnotations 包中获得基于数据注释的验证。Microsoft.AspNetCore.App 元包（ASP.NET Core 2.2 或更高版本）中包括 `Microsoft.Extensions.Options.DataAnnotations`。

```
private class AnnotatedOptions
{
    [Required]
    public string Required { get; set; }

    [StringLength(5, ErrorMessage = "Too long.")]
    public string StringLength { get; set; }

    [Range(-5, 5, ErrorMessage = "Out of range.")]
    public int IntRange { get; set; }
}

[Fact]
public void CanValidateDataAnnotations()
{
    var services = new ServiceCollection();
    services.AddOptions<AnnotatedOptions>()
        .Configure(o =>
        {
            o.StringLength = "111111";
            o.IntRange = 10;
            o.Custom = "nowhere";
        })
        .ValidateDataAnnotations();

    var sp = services.BuildServiceProvider();

    var error = Assert.Throws<OptionsValidationException>(() =>
        sp.GetRequiredService<IOptionsMonitor<AnnotatedOptions>>().Value);
    ValidateFailure<AnnotatedOptions>(error, Options.DefaultName, 1,
        "DataAnnotation validation failed for members Required " +
            "with the error 'The Required field is required.'.",
        "DataAnnotation validation failed for members StringLength " +
            "with the error 'Too long.'.",
        "DataAnnotation validation failed for members IntRange " +
            "with the error 'Out of range.'.");
}
```

设计团队正在考虑为未来版本提供预先验证(在启动时快速失败)。

选项后期配置

使用 IPostConfigureOptions<TOptions> 设置后期配置。进行所有 IConfigureOptions<TOptions> 配置后运行后期配置:

```
services.PostConfigure<MyOptions>(myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

PostConfigure 可用于对命名选项进行后期配置:

```
services.PostConfigure<MyOptions>("named_options_1", myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

使用 PostConfigureAll 对所有配置实例进行后期配置:

```
services.PostConfigureAll<MyOptions>(myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

在启动期间访问选项

IOptions<TOptions> 和 IOptionsMonitor<TOptions> 可用于 `Startup.Configure` 中，因为在 `Configure` 方法执行之前已生成服务。

```
public void Configure(IApplicationBuilder app, IOptionsMonitor<MyOptions> optionsAccessor)
{
    var option1 = optionsAccessor.CurrentValue.Option1;
}
```

不使用 `Startup.ConfigureServices` 中的 IOptions<TOptions> 或 IOptionsMonitor<TOptions>。由于服务注册的顺序，可能存在不一致的选项状态。

其他资源

- ASP.NET Core 中的配置

ASP.NET Core 中的日志记录

2018/10/30 • • Edit Online

作者：Steve Smith 和 Tom Dykstra

ASP.NET Core 支持适用于各种内置和第三方日志记录提供程序的日志记录 API。本文介绍了如何将日志记录 API 与内置提供程序一起使用。

查看或下载示例代码（如何下载）

添加提供程序

日志记录提供程序会显示或存储日志。例如，控制台提供程序在控制台上显示日志，Azure Application Insights 提供程序会将这些日志存储在 Azure Application Insights 中。可通过添加多个提供程序将日志发送到多个目标。

要添加提供程序，请在 Program.cs 中调用提供程序的 `Add{provider name}` 扩展方法：

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var env = hostingContext.HostingEnvironment;
            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                  .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                      optional: true, reloadOnChange: true);
            config.AddEnvironmentVariables();
        })
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
            logging.AddEventSourceLogger();
        })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}
```

默认项目模板调用 CreateDefaultBuilder 扩展方法，该操作将添加以下日志记录提供程序：

- 控制台
- 调试
- EventSource（启动位置：ASP.NET Core 2.2）

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
```

如果使用 `CreateDefaultBuilder`，则可自行选择提供程序来替换默认提供程序。调用 ClearProviders，然后添加所需的提供程序。

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    var todoRepository = host.Services.GetRequiredService<ITodoRepository>();
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Feed the dog" });
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Walk the dog" });

    var logger = host.Services.GetRequiredService<ILogger<Program>>();
    logger.LogInformation("Seeded the database.");

    host.Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        })
        .Build();
```

要使用提供程序，请安装其 NuGet 包，并在 ILoggerFactory 的实例上调用提供程序的扩展方法：

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddConsole()
        .AddDebug();
```

ASP.NET Core 依赖关系注入 (DI) 将提供 `ILoggerFactory` 实例。在 Microsoft.Extensions.Logging.Console 和 Microsoft.Extensions.Logging.Debug 包中定义了 `AddConsole` 和 `AddDebug` 扩展方法。每个扩展方法都调用 `ILoggerFactory.AddProvider` 方法，传入提供程序的一个实例。

> **NOTE**
>
> 示例应用在 `Startup.Configure` 方法中添加了日志提供程序。要从先前执行的代码获取日志输出，请在 `Startup` 类构造函数中添加日志提供程序。

详细了解内置日志记录提供程序，以及本文稍后部分介绍的第三方日志记录提供程序。

创建日志

从 DI 中获取 ILogger<TCategoryName> 对象。

以下控制器示例会创建 `Information` 和 `Warning` 日志。类别为 `TodoApiSample.Controllers.TodoController`（示例应用中 `TodoController` 的完全限定类名）：

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

以下 Razor 页面示例会创建“级别”为 `Information` 且“类别”为 `TodoApiSample.Pages.AboutModel` 的日志：

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }
```

```
public void OnGet()
{
    Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";
    _logger.LogInformation("Message displayed: {Message}", Message);
}
```

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

上述示例创建了“级别”为 `Information` 和 `Warning`，且“类别”为 `TodoController` 的日志。

日志“级别”代表所记录事件的严重程度。日志“类别”是与每个日志关联的字符串。`ILogger<T>` 实例会创建“类别”为类型 `T` 的完全限定名称的日志。本文稍后部分将更详细地介绍级别和类别。

启动时创建日志

要将日志写入 `Startup` 类，构造函数签名需包含 `ILogger` 参数：

```
public class Startup
{
    private readonly ILogger _logger;

    public Startup(IConfiguration configuration, ILogger<Startup> logger)
    {
        Configuration = configuration;
        _logger = logger;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();

        // Add our repository type
        services.AddSingleton<ITodoRepository, TodoRepository>();
        _logger.LogInformation("Added TodoRepository to services");
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            _logger.LogInformation("In Development environment");
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();

        app.UseMvc();
    }
}
```

在程序中创建日志

要将日志写入 `Program` 类，请从 DI 获取 `ILogger` 实例：

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    var todoRepository = host.Services.GetRequiredService<ITodoRepository>();
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Feed the dog" });
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Walk the dog" });

    var logger = host.Services.GetRequiredService<ILogger<Program>>();
    logger.LogInformation("Seeded the database.");

    host.Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        })
        .Build();
```

没有异步记录器方法

日志记录应该会很快，不值得牺牲性能来使用异步代码。如果你的日志数据存储很慢，请不要直接写入它。首先考虑将日志消息写入快速存储，售后再将其变为慢速存储。例如，记录到由另一进程读取和暂留以减缓存储的消息队列。

配置

日志记录提供程序配置由一个或多个配置提供程序提供：

- 文件格式（INI、JSON 和 XML）。
- 命令行参数。
- 环境变量。
- 内存中的 .NET 对象。
- 未加密的机密管理器存储。
- 加密的用户存储，如 Azure Key Vault。
- （已安装或已创建的）自定义提供程序。

例如，日志记录配置通常由应用设置文件的 `Logging` 部分提供。以下示例显示了典型 *appsettings.Development.json* 文件的内容：

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    },
    "Console":
    {
      "IncludeScopes": true
    }
  }
}
```

`Logging` 属性可具有 `LogLevel` 和日志提供程序属性（显示控制台）。

`Logging` 下的 `LogLevel` 属性指定了用于记录所选类别的最低级别。在本例中，`System` 和 `Microsoft` 类别在 `Information` 级别记录，其他均在 `Debug` 级别记录。

`Logging` 下的其他属性均指定了日志记录提供程序。本示例针对控制台提供程序。如果提供程序支持日志作用域，则 `IncludeScopes` 将指示是否启用这些域。提供程序属性（例如本例的 `Console`）也可指定 `LogLevel` 属性。提供程序下的 `LogLevel` 指定了该提供程序记录的级别。

如果在 `Logging.{providername}.LogLevel` 中指定了级别，则这些级别将重写 `Logging.LogLevel` 中设置的所有内容。

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

`LogLevel` 项表示日志名称。`Default` 项适用于未显式列出的日志。其值表示应用于给定日志的日志级别。

若要了解如何实现配置提供程序，请参阅 ASP.NET Core 中的配置。

日志记录输出示例

使用上一部分中显示的示例代码从命令行运行应用时，将在控制台中看到日志。以下是控制台输出示例：

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/api/todo/0
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) -
ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      GetById(0) NOT FOUND
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 42.9286ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 148.889ms 404
```

通过向 `http://localhost:5000/api/todo/0` 处的示例应用发出 HTTP Get 请求来生成前述日志。

在 Visual Studio 中运行示例应用时，“调试”窗口中将显示如下日志：

```
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request starting HTTP/1.1 GET
http://localhost:53104/api/todo/0
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executing action method
TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) - ModelState is Valid
TodoApi.Controllers.TodoController:Information: Getting item 0
TodoApi.Controllers.TodoController:Warning: GetById(0) NOT FOUND
Microsoft.AspNetCore.Mvc.StatusCodeResult:Information: Executing HttpStatusCodeResult, setting HTTP status code
404
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executed action
TodoApi.Controllers.TodoController.GetById (TodoApi) in 152.5657ms
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request finished in 316.3195ms 404
```

由上一部分所示的 `ILogger` 调用创建的日志是以“TodoApi.Controllers.TodoController”开头的。以“Microsoft”类别

开头的日志来自 ASP.NET Core 框架代码。ASP.NET Core 和应用程序代码使用相同的日志记录 API 和提供程序。

本文余下部分将介绍有关日志记录的某些详细信息及选项。

NuGet 包

`ILogger` 和 `ILoggerFactory` 接口位于 Microsoft.Extensions.Logging.Abstractions 中，其默认实现位于 Microsoft.Extensions.Logging 中。

日志类别

创建 `ILogger` 对象后，将为其指定“类别”。该类别包含在由此 `Ilogger` 实例创建的每条日志消息中。类别可以是任何字符串，但约定需使用类名，例如“TodoApi.Controllers.TodoController”。

使用 `ILogger<T>` 获取一个 `ILogger` 实例，该实例使用 `T` 的完全限定类型名称作为类别：

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```


```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```


要显式指定类别，请调用 `ILoggerFactory.CreateLogger`：

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILoggerFactory logger)
    {
        _todoRepository = todoRepository;
        _logger = logger.CreateLogger("TodoApiSample.Controllers.TodoController");
    }
```

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILoggerFactory logger)
    {
        _todoRepository = todoRepository;
        _logger = logger.CreateLogger("TodoApiSample.Controllers.TodoController");
    }
```

`ILogger<T>` 相当于使用 `T` 的完全限定类型名称来调用 `CreateLogger`。

日志级别

每个日志都指定了一个 LogLevel 值。日志级别指示严重性或重要程度。例如，可在方法正常结束时写入 `Information` 日志，在方法返回"404 找不到"状态代码时写入 `Warning` 日志。

下面的代码会创建 `Information` 和 `Warning` 日志：

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```


```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```


在上述代码中，第一个参数是日志事件 ID。第二个参数是消息模板，其中的占位符用于填写剩余方法形参提供的实参值。稍后将在本文的消息模板部分介绍方法参数。

在方法名称中包含级别的日志方法（例如 `LogInformation` 和 `LogWarning`）是 ILogger 的扩展方法。这些方法会调用可接受 `LogLevel` 参数的 `Log` 方法。可直接调用 `Log` 方法而不调用其中某个扩展方法，但语法相对复杂。有关详细信息，请参阅 ILogger 和记录器扩展源代码。

ASP.NET Core 定义了以下日志级别（按严重性从低到高排列）。

- 跟踪 = 0

  有关通常仅用于调试的信息。这些消息可能包含敏感应用程序数据，因此不得在生产环境中启用它们。默认情况下禁用。

- 调试 = 1

有关在开发和调试中可能有用的信息。示例：由于 日志数量过多，因此 `Entering method Configure with flag set to true.` 仅当故障排除时才在生产中启用 `Debug` 级别日志。

- 信息 = 2

  用于跟踪应用的常规流。这些日志通常有长期价值。示例：`Request received for path /api/todo`

- 警告 = 3

  表示应用流中的异常或意外事件。可能包括不会中断应用运行但仍需调查的错误或其他条件。`Warning` 日志级别常用于已处理的异常。示例：`FileNotFoundException for file quotes.txt.`

- 错误 = 4

  表示无法处理的错误和异常。这些消息指示的是当前活动或操作（例如当前 HTTP 请求）中的失败，而不是整个应用中的失败。日志消息示例：`Cannot insert record due to duplicate key violation.`

- 严重 = 5

  需要立即关注的失败。例如数据丢失、磁盘空间不足。

使用日志级别控制写入到特定存储介质或显示窗口的日志输出量。例如:

- 在生产中，通过 `Information` 级别将 `Trace` 发送到卷数据存储。通过 `Critical` 级别将 `Warning` 发送到值数据存储。
- 在开发过程中，通过 `Critical` 级别将 `Warning` 发送到控制台，并在进行故障排除时通过 `Information` 级别添加 `Trace`。

本文稍后的日志筛选部分介绍如何控制提供程序处理的日志级别。

ASP.NET Core 为框架事件写入日志。本文前面部分提供的日志示例排除了低于 `Information` 级别的日志，因此未创建 `Debug` 或 `Trace` 级别日志。以下示例介绍了通过运行配置为显示 `Debug` 日志的示例应用而生成的控制台日志：

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:62555/api/todo/0
dbug: Microsoft.AspNetCore.Routing.Tree.TreeRouter[1]
      Request successfully matched the route with name 'GetTodo' and template 'api/Todo/{id}'.
dbug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'TodoApi.Controllers.TodoController.Update (TodoApi)' with id '089d59b6-92ec-472d-b552-
cc613dfd625d' did not match the constraint 'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
dbug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'TodoApi.Controllers.TodoController.Delete (TodoApi)' with id 'f3476abe-4bd9-4ad3-9261-
3ead09607366' did not match the constraint 'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
dbug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action TodoApi.Controllers.TodoController.GetById (TodoApi)
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) -
ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      GetById(0) NOT FOUND
dbug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action method TodoApi.Controllers.TodoController.GetById (TodoApi), returned result
Microsoft.AspNetCore.Mvc.NotFoundResult.
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 0.8788ms
dbug: Microsoft.AspNetCore.Server.Kestrel[9]
      Connection id "0HL6L7NEFF2QD" completed keep alive response.
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 2.7286ms 404
```

日志事件 ID

每个日志都可指定一个事件 ID。该示例应用通过使用本地定义的 `LoggingEvents` 类来执行此操作：

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

```
public class LoggingEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems = 1001;
    public const int GetItem = 1002;
    public const int InsertItem = 1003;
    public const int UpdateItem = 1004;
    public const int DeleteItem = 1005;

    public const int GetItemNotFound = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

```
public class LoggingEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems = 1001;
    public const int GetItem = 1002;
    public const int InsertItem = 1003;
    public const int UpdateItem = 1004;
    public const int DeleteItem = 1005;

    public const int GetItemNotFound = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

事件 ID 与一组事件相关联。例如，与在页面上显示项列表相关的所有日志可能是 1001。

日志记录提供程序可将事件 ID 存储在 ID 字段中，存储在日志记录消息中，或者不进行存储。调试提供程序不显示事件 ID。控制台提供程序在类别后的括号中显示事件 ID：

```
info: TodoApi.Controllers.TodoController[1002]
      Getting item invalidid
warn: TodoApi.Controllers.TodoController[4000]
      GetById(invalidid) NOT FOUND
```

日志消息模板

每个日志都会指定一个消息模板。消息模板可包含要填写参数的占位符。请在占位符中使用名称而不是数字。

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

占位符的顺序(而非其名称)决定了为其提供值的参数。在以下代码中，请注意消息模板中的参数名称未按顺序排列：

```
string p1 = "parm1";
string p2 = "parm2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

此代码创建了一个参数值按顺序排列的日志消息：

```
Parameter values: parm1, parm2
```

日志记录框架按此方式工作，这样日志记录提供程序即可实现语义日志记录，也称为结构化日志记录。参数本身会传递给日志记录系统，而不仅仅是格式化的消息模板。通过此信息，日志记录提供程序能够将参数值存储为字段。例如，假设记录器方法调用如下所示：

```
_logger.LogInformation("Getting item {ID} at {RequestTime}", id, DateTime.Now);
```

如果要将日志发送到 Azure 表存储，则每个 Azure 表实体都可具有 `ID` 和 `RequestTime` 属性，这简化了对日志数据的查询。无需分析文本消息的超时，查询即可找到特定 `RequestTime` 范围内的全部日志。

日志记录异常

记录器方法有可传入异常的重载，如下方示例所示：

```
catch (Exception ex)
{
    _logger.LogWarning(LoggingEvents.GetItemNotFound, ex, "GetById({ID}) NOT FOUND", id);
    return NotFound();
}
return new ObjectResult(item);
```

```
catch (Exception ex)
{
    _logger.LogWarning(LoggingEvents.GetItemNotFound, ex, "GetById({ID}) NOT FOUND", id);
    return NotFound();
}
return new ObjectResult(item);
```

不同的提供程序处理异常信息的方式不同。以下是上示代码的调试提供程序输出示例。

```
TodoApi.Controllers.TodoController:Warning: GetById(036dd898-fb01-47e8-9a65-f92eb73cf924) NOT FOUND

System.Exception: Item not found exception.
 at TodoApi.Controllers.TodoController.GetById(String id) in
C:\logging\sample\src\TodoApi\Controllers\TodoController.cs:line 226
```

日志筛选

可为特定或所有提供程序和类别指定最低日志级别。最低级别以下的日志不会传递给该提供程序，因此不会显示或存储它们。

要禁止显示所有日志，可将 `LogLevel.None` 指定为最低日志级别。`LogLevel.None` 的整数值为 6，它大于 `LogLevel.Critical` (5)。

在配置中创建筛选规则

项目模板代码调用 `CreateDefaultBuilder` 来为控制台和调试提供程序设置日志记录。`CreateDefaultBuilder` 方法还使用如下所示的代码，设置日志记录以查找 `Logging` 部分的配置：

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var env = hostingContext.HostingEnvironment;
            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                  .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                      optional: true, reloadOnChange: true);
            config.AddEnvironmentVariables();
        })
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
            logging.AddEventSourceLogger();
        })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}
```

配置数据按提供程序和类别指定最低日志级别，如下方示例所示：

```
{
  "Logging": {
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "IncludeScopes": false,
      "LogLevel": {
        "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
        "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
        "Microsoft.AspNetCore.Mvc.Razor": "Error",
        "Default": "Information"
      }
    },
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

此 JSON 将创建 6 条筛选规则：1 条用于调试提供程序，4 条用于控制台提供程序，1 条用于所有提供程序。创建 `ILogger` 对象时，为每个提供程序选择一个规则。

代码中的筛选规则

下面的示例演示了如何在代码中注册筛选规则：

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging =>
        logging.AddFilter("System", LogLevel.Debug)
               .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace))
    .Build();
```

第二个 `AddFilter` 使用类型名称来指定调试提供程序。第一个 `AddFilter` 应用于全部提供程序，因为它未指定提供程序类型。

如何应用筛选规则

先前示例中显示的配置数据和 `AddFilter` 代码会创建下表所示的规则。前六条由配置示例创建，后两条由代码示例创建。

数字	提供程序	类别的开头为...	最低日志级别
1	调试	全部类别	信息
2	控制台	Microsoft.AspNetCore.Mvc.Razor.Internal	警告
3	控制台	Microsoft.AspNetCore.Mvc.Razor.Razor	调试
4	控制台	Microsoft.AspNetCore.Mvc.Razor	Error
5	控制台	全部类别	信息

数字	提供程序	类别的开头为...	最低日志级别
6	全部提供程序	全部类别	调试
7	全部提供程序	系统	调试
8	调试	Microsoft	跟踪

创建 `ILogger` 对象时，`ILoggerFactory` 对象将根据提供程序选择一条规则，将其应用于该记录器。将按所选规则筛选 `ILogger` 实例写入的所有消息。从可用规则中为每个提供程序和类别对选择最具体的规则。

在为给定的类别创建 `ILogger` 时，以下算法将用于每个提供程序：

- 选择匹配提供程序或其别名的所有规则。如果找不到任何匹配项，则选择提供程序为空的所有规则。
- 根据上一步的结果，选择具有最长匹配类别前缀的规则。如果找不到任何匹配项，则选择未指定类别的所有规则。
- 如果选择了多条规则，则采用最后一条。
- 如果未选择任何规则，则使用 `MinimumLevel`。

假设你使用上述规则列表为类别“Microsoft.AspNetCore.Mvc.Razor.RazorViewEngine”创建了 `ILogger` 对象：

- 对于调试提供程序，规则 1、6 和 8 适用。规则 8 最为具体，因此选择了它。
- 对于控制台提供程序，符合的有规则 3、规则 4、规则 5 和规则 6。规则 3 最为具体。

生成的 `ILogger` 实例将 `Trace` 级别及更高级别的日志发送到调试提供程序。`Debug` 级别及更高级别的日志会发送到控制台提供程序。

提供程序别名

每个提供程序都定义了一个别名；可在配置中使用该别名来代替完全限定的类型名称。对于内置提供程序，请使用以下别名：

- 控制台
- 调试
- EventLog
- AzureAppServices
- TraceSource
- EventSource

默认最低级别

仅当配置或代码中的规则对给定提供程序和类别都不适用时，最低级别设置才会生效。下面的示例演示如何设置最低级别：

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging => logging.SetMinimumLevel(LogLevel.Warning))
    .Build();
```

如果没有明确设置最低级别，则默认值为 `Information`，它表示 `Trace` 和 `Debug` 日志将被忽略。

筛选器函数

对配置或代码没有向其分配规则的所有提供程序和类别调用筛选器函数。函数中的代码可访问提供程序类型、类别和日志级别。例如：

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logBuilder =>
    {
        logBuilder.AddFilter((provider, category, logLevel) =>
        {
            if (provider == "Microsoft.Extensions.Logging.Console.ConsoleLoggerProvider" &&
                category == "TodoApiSample.Controllers.TodoController")
            {
                return false;
            }
            return true;
        });
    })
    .Build();
```

通过某些日志记录提供程序，可根据日志级别和类别指定何时向存储介质写入日志、何时忽略日志。

`AddConsole` 和 `AddDebug` 扩展方法提供了接受筛选条件的重载。下列示例代码将使控制台提供程序忽略低于 `Warning` 级别的日志，而使调试提供程序忽略由框架创建的日志。

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddConsole(LogLevel.Warning)
        .AddDebug((category, logLevel) => (category.Contains("TodoApi") && logLevel >= LogLevel.Trace));
```

`AddEventLog` 方法拥有接受 `EventLogSettings` 实例的重载，该实例的 `Filter` 属性中可能包含筛选函数。TraceSource 提供程序不提供以上任何重载，因为其日志记录级别和其他参数都以它使用的 `SourceSwitch` 和 `TraceListener` 为依据。

可使用 `WithFilter` 扩展方法为所有通过 `ILoggerFactory` 实例注册的提供程序设置筛选规则。下面的示例将（类别以“Microsoft”或“System”开头的）框架日志限制为警告，并在调试级别记录应用程序代码创建的日志。

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .WithFilter(new FilterLoggerSettings
        {
            { "Microsoft", LogLevel.Warning },
            { "System", LogLevel.Warning },
            { "ToDoApi", LogLevel.Debug }
        })
        .AddConsole()
        .AddDebug();
```

要防止写入任何日志，请将 `LogLevel.None` 指定为最低日志级别。`LogLevel.None` 的整数值为 6，它大于 `LogLevel.Critical` (5)。

`WithFilter` 扩展方法由 Microsoft.Extensions.Logging.Filter NuGet 包提供。该方法返回一个新的 `ILoggerFactory` 实例，该实例将筛选传递给注册的所有记录器提供程序的日志消息。它不会影响其他任何 `ILoggerFactory` 实例，包括原始 `ILoggerFactory` 实例。

系统类别和级别

下面是 ASP.NET Core 和 Entity Framework Core 使用的一些类别，备注中说明了可从这些类别获取的具体日志：

类别	说明
Microsoft.AspNetCore	常规 ASP.NET Core 诊断。
Microsoft.AspNetCore.DataProtection	考虑、找到并使用了哪些密钥。
Microsoft.AspNetCore.HostFiltering	所允许的主机。
Microsoft.AspNetCore.Hosting	HTTP 请求完成的时间和启动时间。加载了哪些承载启动程序集。
Microsoft.AspNetCore.Mvc	MVC 和 Razor 诊断。模型绑定、筛选器执行、视图编译和操作选择。
Microsoft.AspNetCore.Routing	路由匹配信息。
Microsoft.AspNetCore.Server	连接启动、停止和保持活动响应。HTTP 证书信息。
Microsoft.AspNetCore.StaticFiles	提供的文件。
Microsoft.EntityFrameworkCore	常规 Entity Framework Core 诊断。数据库活动和配置、更改检测、迁移。

日志作用域

“作用域”可对一组逻辑操作分组。此分组可用于将相同的数据附加到作为集合的一部分而创建的每个日志。例如，在处理事务期间创建的每个日志都可包括事务 ID。

范围是由 BeginScope 方法返回的 `IDisposable` 类型，持续至释放为止。要使用作用域，请在 `using` 块中包装记录器调用：

```
public IActionResult GetById(string id)
{
    TodoItem item;
    using (_logger.BeginScope("Message attached to logs created in the using block"))
    {
        _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
        item = _todoRepository.Find(id);
        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
            return NotFound();
        }
    }
    return new ObjectResult(item);
}
```

下列代码为控制台提供程序启用作用域：

Program.cs:

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
    logging.AddConsole(options => options.IncludeScopes = true);
    logging.AddDebug();
})
```

> **NOTE**
>
> 要启用基于作用域的日志记录，必须先配置 `IncludeScopes` 控制台记录器选项。
>
> 若要了解关配置，请参阅配置部分。

Program.cs:

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
    logging.AddConsole(options => options.IncludeScopes = true);
    logging.AddDebug();
})
```

> **NOTE**
>
> 要启用基于作用域的日志记录，必须先配置 `IncludeScopes` 控制台记录器选项。

Startup.cs:

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddConsole(includeScopes: true)
        .AddDebug();
```

每条日志消息都包含作用域内的信息：

```
info: TodoApi.Controllers.TodoController[1002]
      => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 => TodoApi.Controllers.TodoController.GetById
(TodoApi) => Message attached to logs created in the using block
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 => TodoApi.Controllers.TodoController.GetById
(TodoApi) => Message attached to logs created in the using block
      GetById(0) NOT FOUND
```

内置日志记录提供程序

ASP.NET Core 提供以下提供程序：

- 控制台
- 调试
- EventSource

- EventLog
- TraceSource

本文稍后将介绍 Azure 的日志记录选项。

有关 stdout 日志记录的信息，请参阅 对 IIS 上的 ASP.NET Core 进行故障排除 和 对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除。

控制台提供程序

Microsoft.Extensions.Logging.Console 提供程序包向控制台发送日志输出。

```
logging.AddConsole();
```

```
loggerFactory.AddConsole();
```

通过 AddConsole 重载，可传入一个最低日志级别、一个筛选器函数，以及一个用于指示作用域是否受支持的布尔值。另一个选项是传递 `IConfiguration` 对象，可通过它来指定支持的作用域及日志记录级别。

控制台提供程序对性能有重大影响，通常不适合在生产中使用。

在 Visual Studio 中创建新项目时，`AddConsole` 方法如下所示：

```
loggerFactory.AddConsole(Configuration.GetSection("Logging"));
```

此代码引用 appSettings.json 文件的 `Logging` 部分：

```
{
  "Logging": {
    "Console": {
      "IncludeScopes": false
    },
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

所显示的设置将框架日志限制为警告，并允许在调试级别记录应用日志，如日志筛选部分所述。有关详细信息，请参阅配置。

要查看控制台日志记录输出，请在项目文件夹中打开命令提示符并运行以下命令：

```
dotnet run
```

调试提供程序

Microsoft.Extensions.Logging.Debug 提供程序包使用 System.Diagnostics.Debug 类（`Debug.WriteLine` 方法调用）来写入日志输出。

在 Linux 中，此提供程序将日志写入 /var/log/message。

```
logging.AddDebug();
```

```
loggerFactory.AddDebug();
```

AddDebug 重载允许传入最低日志级别或筛选器函数。

EventSource 提供程序

对于面向 ASP.NET Core 1.1.0 或更高版本的应用，Microsoft.Extensions.Logging.EventSource 提供程序包可实现事件跟踪。在 Windows 中，它使用 ETW。提供程序可跨平台使用，但尚无支持 Linux 或 macOS 的事件集合和显示工具。

```
logging.AddEventSourceLogger();
```

```
loggerFactory.AddEventSourceLogger();
```

可使用 PerfView 实用工具收集和查看日志。虽然其他工具也可以查看 ETW 日志，但在处理由 ASP.NET 发出的 ETW 事件时，使用 PerfView 能获得最佳体验。

要将 PerfView 配置为收集此提供程序记录的事件，请向 Additional Providers 列表添加字符串 `*Microsoft-Extensions-Logging`。（请勿遗漏字符串起始处的星号。）

Windows EventLog 提供程序

Microsoft.Extensions.Logging.EventLog 提供程序包向 Windows 事件日志发送日志输出。

```
logging.AddEventLog();
```

```
loggerFactory.AddEventLog();
```

AddEventLog 重载允许传入 `EventLogSettings` 或最低日志级别。

TraceSource 提供程序

Microsoft.Extensions.Logging.TraceSource 提供程序包使用 TraceSource 库和提供程序。

```
logging.AddTraceSource(sourceSwitchName);
```

```
loggerFactory.AddTraceSource(sourceSwitchName);
```

AddTraceSource 重载允许传入资源开关和跟踪侦听器。

要使用此提供程序，应用必须在 .NET Framework（而非 .NET Core）上运行。提供程序可将消息路由到多个侦听器，例如示例应用中使用的 TextWriterTraceListener。

以下示例配置了 `TraceSource` 提供程序，用于向控制台窗口记录 `Warning` 和更高级别的消息。

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddDebug();

    // add Trace Source logging
    var testSwitch = new SourceSwitch("sourceSwitch", "Logging Sample");
    testSwitch.Level = SourceLevels.Warning;
    loggerFactory.AddTraceSource(testSwitch,
        new TextWriterTraceListener(writer: Console.Out));
```

Azure 中的日志记录

要了解 Azure 中的日志记录，请参阅下列部分：

- Azure 应用服务提供程序
- Azure 日志流式处理
- Azure Application Insights 跟踪日志记录

Azure 应用服务提供程序

Microsoft.Extensions.Logging.AzureAppServices 提供程序包将日志写入 Azure App Service 应用的文件系统，以及 Azure 存储帐户中的 blob 存储。面向 .NET Core 1.1 或更高版本的应用可使用该提供程序包。

如果面向 .NET Core，请注意以下几点：

- 提供程序包随附在 ASP.NET Core Microsoft.AspNetCore.All 元包中。
- Microsoft.AspNetCore.App 元包中不包括此提供程序包。要使用提供程序，请安装该程序包。
- 不要显式调用 AddAzureWebAppDiagnostics。将应用部署到 Azure 应用服务时，会自动使提供程序对应用可用。

如果面向 .NET Framework 或引用 `Microsoft.AspNetCore.App` 元包，请向项目添加提供程序包。在 ILoggerFactory 实例上调用 `AddAzureWebAppDiagnostics`：

```
logging.AddAzureWebAppDiagnostics();
```

```
loggerFactory.AddAzureWebAppDiagnostics();
```

AddAzureWebAppDiagnostics 重载允许传入 AzureAppServicesDiagnosticsSettings。设置对象可以覆盖默认设

置，例如日志记录输出模板、blob 名称和文件大小限制。（“输出模板”是一个消息模板，除了通过 `ILogger` 方法调用提供的内容之外，还可将其应用于所有日志。）

在部署应用服务应用时，应用程序将遵循 Azure 门户中“应用服务”页面下的诊断日志部分的设置。更新这些设置后，更改会立即生效，无需重新启动或重新部署应用。

日志文件的默认位置是 D:\home\LogFiles\Application 文件夹，默认文件名为 diagnostics-yyyymmdd.txt。默认文件大小上限为 10 MB，默认最大保留文件数为 2。默认 blob 名为 {app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt。要详细了解默认行为，请参阅 AzureAppServicesDiagnosticsSettings。

该提供程序仅当项目在 Azure 环境中运行时有效。项目在本地运行时，该提供程序无效 — 它不会写入本地文件或 blob 的本地开发存储。

Azure 日志流式处理

通过 Azure 日志流式处理，可从以下位置实时查看日志活动：

- 应用服务器
- Web 服务器
- 请求跟踪失败

要配置 Azure 日志流式处理，请执行以下操作：

- 从应用的门户页导航到“诊断日志”页。
- 将“应用程序日志记录(Filesystem)”设置为“开”。

导航到"日志流式处理"页，查看应用消息。它们由应用通过 `ILogger` 接口记录。

Azure Application Insights 跟踪日志记录

Application Insights SDK 可收集和报告 ASP.NET Core 日志记录基础结构生成的日志。有关更多信息，请参见以下资源：

- Application Insights 概述
- 用于 ASP.NET Core 的 Application Insights
- Microsoft/ApplicationInsights-aspnetcore Wiki：日志记录。

第三方日志记录提供程序

适用于 ASP.NET Core 的第三方日志记录框架：

- elmah.io（GitHub 存储库）
- Gelf（GitHub 存储库）
- JSNLog（GitHub 存储库）
- KissLog.net（GitHub 存储库）
- Loggr（GitHub 存储库）
- NLog（GitHub 存储库）
- Sentry（GitHub 存储库）
- Serilog（GitHub 存储库）
- Stackdriver（Github 存储库）

某些第三方框架可以执行语义日志记录（又称结构化日志记录）。

使用第三方框架类似于使用以下内置提供程序之一：

1. 将 NuGet 包添加到你的项目。
2. 调用 `ILoggerFactory`。

有关详细信息，请参阅各提供程序的相关文档。Microsoft 不支持第三方日志记录提供程序。

其他资源

- 在 ASP.NET Core 中使用 LoggerMessage 的高性能日志记录

处理 ASP.NET Core 中的错误

2019/1/8 • • Edit Online

作者：Steve Smith 和 Tom Dykstra

本文介绍了处理 ASP.NET Core 应用中常见错误的一些方法。

查看或下载示例代码（如何下载）

开发人员异常页

要将应用配置为显示有关异常的详细信息的页面，请使用开发人员异常页。该页面通过 Microsoft.AspNetCore.App 元包中的 Microsoft.AspNetCore.Diagnostics 包提供。向 `Startup.Configure` 方法添加代码行：

要将应用配置为显示有关异常的详细信息的页面，请使用开发人员异常页。该页面通过 Microsoft.AspNetCore.All 元包中的 Microsoft.AspNetCore.Diagnostics 包提供。向 `Startup.Configure` 方法添加代码行：

要将应用配置为显示有关异常的详细信息的页面，请使用开发人员异常页。该页面通过在项目文件中添加 Microsoft.AspNetCore.Diagnostics 包的包引用提供。向 `Startup.Configure` 方法添加代码行：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    env.EnvironmentName = EnvironmentName.Production;

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
```

将对 UseDeveloperExceptionPage 的调用放在要对其捕获异常的任何中间件（例如，`app.UseMvc`）前面。

> **WARNING**
>
> 仅当应用程序在开发环境中运行时才启用开发人员异常页。否则当应用程序在生产环境中运行时，详细的异常信息会向公众泄露 了解环境配置。

要查看开发人员异常页，请将环境设置为 `Development`，运行示例应用，并向应用的基 URL 添加 `?throw=true`。该页面包括几个选项卡，这些选项卡中包含关于异常和请求的信息。第一个选项卡包括堆栈跟踪：

下一个选项卡显示查询字符串参数(如有):

如果请求具有 cookie，它们在"Cookie"选项卡上显示。标头出现在最后一个选项卡中:

配置自定义异常处理页

配置当应用未在 `Development` 环境中运行时要使用的异常处理程序页：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    env.EnvironmentName = EnvironmentName.Production;

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
```

在 Razor Pages 应用中，dotnet new Razor Pages 模板在 Pages 文件夹中提供"错误"页面和错误 `PageModel` 类。

在 MVC 应用中，不要使用 HTTP 方法特性（如 `HttpGet`）修饰错误处理程序操作方法。显式谓词可阻止某些请求访问方法。允许匿名访问方法，以便未经身份验证的用户能够接收错误视图。

例如，以下错误处理程序方法由 dotnet new MVC 模板提供并在主控制器中显示：

```
[AllowAnonymous]
public IActionResult Error()
{
    return View(new ErrorViewModel
        { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
}
```

配置状态代码页

默认情况下，应用不会为 HTTP 状态代码提供丰富状态代码页，例如 404 未找到。要提供状态代码页，请使用状态代码页中间件。

该中间件通过 Microsoft.AspNetCore.App 元包中的 Microsoft.AspNetCore.Diagnostics 包提供。

该中间件通过 Microsoft.AspNetCore.All 元包中的 Microsoft.AspNetCore.Diagnostics 包提供。

该中间件通过在项目文件中添加 Microsoft.AspNetCore.Diagnostics 包的包引用提供。

向 `Startup.Configure` 方法添加代码行：

```
app.UseStatusCodePages();
```

应在管道中的请求处理中间件之前调用 UseStatusCodePages（例如，静态文件中间件和 MVC 中间件）。

默认情况下，状态代码页中间件为常见状态代码（如 404）添加纯文本处理程序：

该中间件支持多种扩展方法。一种方法采用 Lambda 表达式：

```
// Expose the members of the 'Microsoft.AspNetCore.Http' namespace
// at the top of the file:
// using Microsoft.AspNetCore.Http;
app.UseStatusCodePages(async context =>
{
    context.HttpContext.Response.ContentType = "text/plain";

    await context.HttpContext.Response.WriteAsync(
        "Status code page, status code: " +
        context.HttpContext.Response.StatusCode);
});
```

`UseStatusCodePages` 重载需要使用内容类型和格式字符串：

```
app.UseStatusCodePages("text/plain", "Status code page, status code: {0}");
```

重定向重新执行扩展方法

UseStatusCodePagesWithRedirects：

- 向客户端发送"302 - 已找到"状态代码。
- 将客户端重定向到 URL 模板中的位置。

该模板可能包括状态代码的 `{0}` 占位符。模板必须以正斜杠（`/`）开头。

```
app.UseStatusCodePagesWithRedirects("/error/{0}");
```

UseStatusCodePagesWithReExecute：

- 向客户端返回原始状态代码。
- 指定应使用备用路径重新执行请求管道，从而生成响应正文。

该模板可能包括状态代码的 `{0}` 占位符。模板必须以正斜杠（`/`）开头。

```
app.UseStatusCodePagesWithReExecute("/error/{0}");
```

可禁用 Razor 页处理程序方法或 MVC 控制器中的特定请求的状态代码页。要禁用状态代码页，请尝试从请求的 HttpContext.Features 集合中检索 IStatusCodePagesFeature，并在功能可用时禁用该功能：

```
var statusCodePagesFeature = HttpContext.Features.Get<IStatusCodePagesFeature>();

if (statusCodePagesFeature != null)
{
    statusCodePagesFeature.Enabled = false;
}
```

若要在应用中使用指向终结点的 `UseStatusCodePages*` 重载，请为终结点创建 MVC 视图或 Razor Page。例如，Razor Pages 应用的 dotnet new 模板将生成以下页面和页面模型类：

Error.cshtml：

```
@page
@model ErrorModel
@{
    ViewData["Title"] = "Error";
}

<h1 class="text-danger">Error.</h1>
<h2 class="text-danger">An error occurred while processing your request.</h2>

@if (Model.ShowRequestId)
{
    <p>
        <strong>Request ID:</strong> <code>@Model.RequestId</code>
    </p>
}

<h3>Development Mode</h3>
<p>
    Swapping to <strong>Development</strong> environment will display more detailed
    information about the error that occurred.
</p>
<p>
    <strong>Development environment should not be enabled in deployed applications
    </strong>, as it can result in sensitive information from exceptions being
    displayed to end users. For local debugging, development environment can be
    enabled by setting the <strong>ASPNETCORE_ENVIRONMENT</strong> environment
    variable to <strong>Development</strong>, and restarting the application.
</p>
```

Error.cshtml.cs：

```
public class ErrorModel : PageModel
{
    public string RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
        NoStore = true)]
    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
    }
}
```

异常处理代码

异常处理页中的代码可能会引发异常。建议在生产错误页面中包含纯静态内容。

此外还要注意，一旦发送响应的标头，则无法更改响应的状态代码，也不能运行任何异常页或处理程序。必须完成响应或中止连接。

服务器异常处理

除应用程序中的异常处理逻辑外，托管应用程序的服务器服务器也会执行一些异常处理。如果服务器在发送标头之前捕获异常，服务器将发送没有正文的 500 内部服务器错误响应。如果服务器在已发送标头后捕获异常，服务器会关闭连接。应用程序无法处理的请求将由服务器进行处理。发生的任何异常都将由服务器进行处理。任何配置的自定义错误页面或异常处理中间件或筛选器都不会影响此行为。

启动异常处理

应用程序启动期间发生的异常仅可在承载层进行处理。利用 Web 主机，你可以使用 `captureStartupErrors` 和 `detailedErrors` 键配置主机在响应启动期间出现错误时的行为方式。

仅当捕获的启动错误发生在主机地址/端口绑定之后，承载层才会为该错误显示错误页。如果绑定因任何原因而失败，则承载层会记录关键异常，dotnet 进程崩溃，且在 Kestrel 服务器上运行应用时，不会显示任何错误页。

在 IIS 或 IIS Express 上运行应用时，如果无法启动进程，ASP.NET Core 模块将返回 502.5 进程失败。要了解在通过 IIS 托管时如何排查启动问题，请参阅 对 IIS 上的 ASP.NET Core 进行故障排除。要了解如何排查 Azure 应用服务的启动问题，请参阅 对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除。

ASP.NET Core MVC 错误处理

MVC 应用还有一些其他的错误处理选项，例如配置异常筛选器和执行模型验证。

异常筛选器

在 MVC 应用中，异常筛选器可以进行全局配置，也可以为每个控制器或每个操作单独配置。这些筛选器处理在执行控制器操作或其他筛选器时出现的任何未处理的异常。不会以其他方式调用这些筛选器。要了解详细信息，请参阅筛选器。

> **TIP**
>
> 异常筛选器非常适用于捕获 MVC 操作内出现的异常，但灵活性不如错误处理中间件。一般情况下最好使用中间件；仅在需要根据所选 MVC 操作以不同方式执行错误处理时，才使用筛选器。

处理模型状态错误

模型验证在每个控制器操作被调用之前发生，操作方法负责检查 `ModelState.IsValid` 并相应地作出反应。

某些应用使用标准约定处理模型验证错误，在这种情况下，使用筛选器可以更好地实施这种策略。你需要使用无效模型状态测试操作的行为。查看测试控制器逻辑了解详情。

其他资源

- Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考
- 对 IIS 上的 ASP.NET Core 进行故障排除
- 对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除

ASP.NET Core 中间件

2018/12/5 • • Edit Online

作者：Rick Anderson 和 Steve Smith

中间件是一种装配到应用管道以处理请求和响应的软件。每个组件：

- 选择是否将请求传递到管道中的下一个组件。
- 可在调用管道中的下一个组件前后执行工作。

请求委托用于生成请求管道。请求委托处理每个 HTTP 请求。

使用 RunMap 和 Use 扩展方法来配置请求委托。可将一个单独的请求委托并行指定为匿名方法（称为并行中间件），或在可重用的类中对其进行定义。这些可重用的类和并行匿名方法即为中间件，也叫中间件组件。请求管道中的每个中间件组件负责调用管道中的下一个组件，或使管道短路。

将 HTTP 处理程序和模块迁移到 ASP.NET Core 中间件 介绍了 ASP.NET Core 和 ASP.NET 4.x 中请求管道之间的差异，并提供了更多的中间件示例。

使用 IApplicationBuilder 创建中间件管道

ASP.NET Core 请求管道包含一系列请求委托，依次调用。下图演示了这一概念。沿黑色箭头执行。

每个委托均可在下一个委托前后执行操作。此外，委托还可以决定不将请求传递给下一个委托，这就是对请求管道进行短路。通常需要短路，因为这样可以避免不必要的工作。例如，静态文件中间件可以返回静态文件请求并使管道的其余部分短路。先在管道中调用异常处理委托，以便它们可以捕获在管道的后期阶段所发生的异常。

尽可能简单的 ASP.NET Core 应用设置了处理所有请求的单个请求委托。这种情况不包括实际请求管道。调用单个匿名函数以响应每个 HTTP 请求。

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello, World!");
        });
    }
}
```

第一个 Run 委托终止了管道。

用 Use 将多个请求委托链接在一起。`next` 参数表示管道中的下一个委托。可通过不调用 next 参数使管道短路。通常可在下一个委托前后执行操作，如以下示例所示：

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}
```

WARNING

在向客户端发送响应后，请勿调用 `next.Invoke`。响应启动后，针对 HttpResponse 的更改将引发异常。例如，设置标头和状态代码更改将引发异常。调用 `next` 后写入响应正文：

- 可能导致违反协议。例如，写入的长度超过规定的 `Content-Length`。
- 可能损坏正文格式。例如，向 CSS 文件中写入 HTML 页脚。

HasStarted 是一个有用的提示，指示是否已发送标头或已写入正文。

顺序

向 `Startup.Configure` 方法添加中间件组件的顺序定义了针对请求调用这些组件的顺序，以及响应的相反顺序。此排序对于安全性、性能和功能至关重要。

以下 `Startup.Configure` 方法将为常见应用方案添加中间件组件：

1. 异常/错误处理
2. HTTP 严格传输安全协议
3. HTTPS 重定向
4. 静态文件服务器
5. Cookie 策略实施
6. 身份验证
7. 会话

8. MVC

```
public void Configure(IApplicationBuilder app)
{
    if (env.IsDevelopment())
    {
        // When the app runs in the Development environment:
        //   Use the Developer Exception Page to report app runtime errors.
        //   Use the Database Error Page to report database runtime errors.
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        // When the app doesn't run in the Development environment:
        //   Enable the Exception Handler Middleware to catch exceptions
        //     thrown in the following middlewares.
        //   Use the HTTP Strict Transport Security Protocol (HSTS)
        //     Middleware.
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    // Use HTTPS Redirection Middleware to redirect HTTP requests to HTTPS.
    app.UseHttpsRedirection();

    // Return static files and end the pipeline.
    app.UseStaticFiles();

    // Use Cookie Policy Middleware to conform to EU General Data
    // Protection Regulation (GDPR) regulations.
    app.UseCookiePolicy();

    // Authenticate before the user accesses secure resources.
    app.UseAuthentication();

    // If the app uses session state, call Session Middleware after Cookie
    // Policy Middleware and before MVC Middleware.
    app.UseSession();

    // Add MVC to the request pipeline.
    app.UseMvc();
}
```

1. 异常/错误处理
2. 静态文件
3. 身份验证
4. 会话
5. MVC

```
public void Configure(IApplicationBuilder app)
{
    // Enable the Exception Handler Middleware to catch exceptions
    //   thrown in the following middlewares.
    app.UseExceptionHandler("/Home/Error");

    // Return static files and end the pipeline.
    app.UseStaticFiles();

    // Authenticate before you access secure resources.
    app.UseIdentity();

    // If the app uses session state, call UseSession before
    // MVC Middleware.
    app.UseSession();

    // Add MVC to the request pipeline.
    app.UseMvcWithDefaultRoute();
}
```

在前面的示例代码中，每个中间件扩展方法都通过 Microsoft.AspNetCore.Builder 命名空间在 IApplicationBuilder 上公开。

UseExceptionHandler 是添加到管道的第一个中间件组件。因此，异常处理程序中间件可捕获稍后调用中发生的任何异常。

尽早在管道中调用静态文件中间件，以便它可以处理请求并使其短路，而无需通过剩余组件。静态文件中间件不提供授权检查。可公开访问由静态文件中间件服务的任何文件，包括 wwwroot 下的文件。若要了解如何保护静态文件，请参阅 ASP.NET Core 中的静态文件。

如果静态文件中间件未处理请求，则请求将被传递给执行身份验证的身份验证中间件 (UseAuthentication)。身份验证不使未经身份验证的请求短路。虽然身份验证中间件对请求进行身份验证，但仅在 MVC 选择特定 Razor 页或 MVC 控制器和操作后，才发生授权（和拒绝）。

如果静态文件中间件未处理请求，则请求将被传递给执行身份验证的标识中间件 (UseIdentity)。标识不使未经身份验证的请求短路。虽然标识对请求进行身份验证，但仅在 MVC 选择特定控制器和操作后，才发生授权（和拒绝）。

以下示例演示中间件排序，其中静态文件的请求在响应压缩中间件前由静态文件中间件进行处理。使用此中间件顺序不压缩静态文件。可以压缩来自 UseMvcWithDefaultRoute 的 MVC 响应。

```
public void Configure(IApplicationBuilder app)
{
    // Static files not compressed by Static File Middleware.
    app.UseStaticFiles();
    app.UseResponseCompression();
    app.UseMvcWithDefaultRoute();
}
```

Use、Run 和 Map

使用 `Use`、`Run` 和 `Map` 配置 HTTP 管道。`Use` 方法可使管道短路（即不调用 `next` 请求委托）。`Run` 是一种约定，并且某些中间件组件可公开在管道末尾运行的 `Run[Middleware]` 方法。

Map 扩展用作约定来创建管道分支。`Map*` 基于给定请求路径的匹配项来创建请求管道分支。如果请求路径以给定路径开头，则执行分支。

```
public class Startup
{
    private static void HandleMapTest1(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 1");
        });
    }

    private static void HandleMapTest2(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 2");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1", HandleMapTest1);

        app.Map("/map2", HandleMapTest2);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

下表使用前面的代码显示来自 `http://localhost:1234` 的请求和响应。

请求	响应
localhost:1234	Hello from non-Map delegate.
localhost:1234/map1	Map Test 1
localhost:1234/map2	Map Test 2
localhost:1234/map3	Hello from non-Map delegate.

使用 `Map` 时，将从 `HttpRequest.Path` 中删除匹配的线段，并针对每个请求将该线段追加到 `HttpRequest.PathBase`。

MapWhen 基于给定谓词的结果创建请求管道分支。`Func<HttpContext, bool>` 类型的任何谓词均可用于将请求映射到管道的新分支。在以下示例中，谓词用于检测查询字符串变量 `branch` 是否存在：

```
public class Startup
{
    private static void HandleBranch(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            var branchVer = context.Request.Query["branch"];
            await context.Response.WriteAsync($"Branch used = {branchVer}");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.MapWhen(context => context.Request.Query.ContainsKey("branch"),
                                HandleBranch);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

下表使用前面的代码显示来自 `http://localhost:1234` 的请求和响应。

请求	响应
localhost:1234	Hello from non-Map delegate.
localhost:1234/?branch=master	Branch used = master

`Map` 支持嵌套，例如：

```
app.Map("/level1", level1App => {
    level1App.Map("/level2a", level2AApp => {
        // "/level1/level2a" processing
    });
    level1App.Map("/level2b", level2BApp => {
        // "/level1/level2b" processing
    });
});
```

此外，`Map` 还可同时匹配多个段：

```
public class Startup
{
    private static void HandleMultiSeg(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map multiple segments.");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1/seg1", HandleMultiSeg);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate.");
        });
    }
}
```

内置中间件

ASP.NET Core 附带以下中间件组件。顺序列提供备注，说明中间件在请求管道中的放置，以及中间件可能终止请求并阻止其他中间件处理请求的条件。

中间件	描述	顺序
身份验证	提供身份验证支持。	在需要 `HttpContext.User` 之前。OAuth 回叫的终端。
Cookie 策略	跟踪用户是否同意存储个人信息，并强制实施 cookie 字段（如 `secure` 和 `SameSite`）的最低标准。	在发出 cookie 的中间件之前。示例：身份验证、会话、MVC (TempData)。
CORS	配置跨域资源共享。	在使用 CORS 的组件之前。
诊断	配置诊断。	在生成错误的组件之前。
转接头	将代理标头转发到当前请求。	在使用已更新字段的组件之前。示例：方案、主机、客户端 IP、方法。
运行状况检查	检查 ASP.NET Core 应用及其依赖项的运行状况，如检查数据库可用性。	如果请求与运行状况检查终结点匹配，则为终端。
HTTP 方法重写	允许传入 POST 请求重写方法。	在使用已更新方法的组件之前。
HTTPS 重定向	将所有 HTTP 请求重定向到 HTTPS（ASP.NET Core 2.1 或更高版本）。	在使用 URL 的组件之前。
HTTP 严格传输安全性 (HSTS)	添加特殊响应标头的安全增强中间件（ASP.NET Core 2.1 或更高版本）。	在发送响应之前，修改请求的组件之后。示例：转接头、URL 重写。
MVC	用 MVC/Razor Pages 处理请求（ASP.NET Core 2.0 或更高版本）。	如果请求与路由匹配，则为终端。

中间件	描述	顺序
OWIN	与基于 OWIN 的应用、服务器和中间件进行互操作。	如果 OWIN 中间件处理完请求，则为终端。
响应缓存	提供对缓存响应的支持。	在需要缓存的组件之前。
响应压缩	提供对压缩响应的支持。	在需要压缩的组件之前。
请求本地化	提供本地化支持。	在对本地化敏感的组件之前。
路由	定义和约束请求路由。	用于匹配路由的终端。
会话	提供对管理用户会话的支持。	在需要会话的组件之前。
静态文件	为提供静态文件和目录浏览提供支持。	如果请求与文件匹配，则为终端。
URL 重写	提供对重写 URL 和重定向请求的支持。	在使用 URL 的组件之前。
WebSockets	启用 WebSockets 协议。	在接受 WebSocket 请求所需的组件之前。

编写中间件

通常，中间件封装在类中，并且通过扩展方法公开。请考虑以下中间件，该中间件通过查询字符串设置当前请求的区域性：

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use((context, next) =>
        {
            var cultureQuery = context.Request.Query["culture"];
            if (!string.IsNullOrWhiteSpace(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;
            }

            // Call the next delegate/middleware in the pipeline
            return next();
        });

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });

    }
}
```

以上示例代码用于演示创建中间件组件。有关 ASP.NET Core 的内置本地化支持，请参阅 ASP.NET

Core 全球化和本地化。

可通过传入区域性(如 `http://localhost:7997/?culture=no`)测试中间件。

以下代码将中间件委托移动到类:

```
using Microsoft.AspNetCore.Http;
using System.Globalization;
using System.Threading.Tasks;

namespace Culture
{
    public class RequestCultureMiddleware
    {
        private readonly RequestDelegate _next;

        public RequestCultureMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task InvokeAsync(HttpContext context)
        {
            var cultureQuery = context.Request.Query["culture"];
            if (!string.IsNullOrWhiteSpace(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;

            }

            // Call the next delegate/middleware in the pipeline
            await _next(context);
        }
    }
}
```

中间件 `Task` 方法的名称必须是 `Invoke`。在 ASP.NET Core 2.0 或更高版本中,该名称可以为 `Invoke` 或 `InvokeAsync`。

以下扩展方法通过 IApplicationBuilder 公开中间件:

```
using Microsoft.AspNetCore.Builder;

namespace Culture
{
    public static class RequestCultureMiddlewareExtensions
    {
        public static IApplicationBuilder UseRequestCulture(
            this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<RequestCultureMiddleware>();
        }
    }
}
```

以下代码通过 `Startup.Configure` 调用中间件:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseRequestCulture();

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });

    }
}
```

中间件应通过在其构造函数中公开其依赖项来遵循显式依赖项原则。在每个应用程序生存期构造一次中间件。如果需要与请求中的中间件共享服务，请参阅按请求依赖项部分。

中间件组件可通过构造函数参数从依赖关系注入 (DI) 解析其依赖项。UseMiddleware<T> 也可直接接受其他参数。

按请求依赖项

由于中间件是在应用启动时构造的，而不是按请求构造的，因此在每个请求过程中，中间件构造函数使用的范围内生存期服务不与其他依赖关系注入类型共享。如果必须在中间件和其他类型之间共享范围内服务，请将这些服务添加到 `Invoke` 方法的签名。`Invoke` 方法可接受由 DI 填充的其他参数：

```
public class CustomMiddleware
{
    private readonly RequestDelegate _next;

    public CustomMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    // IMyScopedService is injected into Invoke
    public async Task Invoke(HttpContext httpContext, IMyScopedService svc)
    {
        svc.MyProperty = 1000;
        await _next(httpContext);
    }
}
```

其他资源

- 将 HTTP 处理程序和模块迁移到 ASP.NET Core 中间件
- ASP.NET Core 中的应用启动
- ASP.NET Core 中的请求功能
- ASP.NET Core 中基于工厂的中间件激活
- 使用 ASP.NET Core 中的第三方容器激活中间件

ASP.NET Core 中基于工厂的中间件激活

2018/10/30 • • Edit Online

作者：Luke Latham

IMiddlewareFactory/IMiddleware 是中间件激活的扩展点。

`UseMiddleware` 扩展方法检查中间件的已注册类型是否实现 `IMiddleware`。如果是，则使用在容器中注册的 `IMiddlewareFactory` 实例来解析 `IMiddleware` 实现，而不使用基于约定的中间件激活逻辑。中间件在应用的服务容器中注册为作用域或瞬态服务。

优点：

- 按请求（作用域服务的注入）激活
- 让中间件强类型化

`IMiddleware` 按请求激活，因此作用域服务可以注入到中间件的构造函数中。

查看或下载示例代码（如何下载）

示例应用演示了使用以下两种方式激活的中间件：

- 约定。有关使用约定激活中间件的详细信息，请参阅中间件主题。
- IMiddleware 实现。默认的 MiddlewareFactory 类可激活中间件。

这两种中间件实现的功能相同，并能记录由查询字符串参数（`key`）提供的值。中间件使用插入的数据库上下文（作用域服务）将查询字符串值记录在内存中数据库。

IMiddleware

IMiddleware 定义应用的请求管道的中间件。InvokeAsync(HttpContext, RequestDelegate) 方法处理请求，并返回代表中间件执行的 `Task`。

使用约定激活的中间件：

```
public class ConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public ConventionalMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context, AppDbContext db)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation = "ConventionalMiddleware",
                    Value = keyValue
                });

            await db.SaveChangesAsync();
        }

        await _next(context);
    }
}
```

使用 `MiddlewareFactory` 激活的中间件：

```
public class FactoryActivatedMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public FactoryActivatedMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation = "FactoryActivatedMiddleware",
                    Value = keyValue
                });

            await _db.SaveChangesAsync();
        }

        await next(context);
    }
}
```

程序会为中间件创建扩展：

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseConventionalMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<ConventionalMiddleware>();
    }

    public static IApplicationBuilder UseFactoryActivatedMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<FactoryActivatedMiddleware>();
    }
}
```

无法通过 `UseMiddleware` 将对象传递给工厂激活的中间件：

```
public static IApplicationBuilder UseFactoryActivatedMiddleware(
    this IApplicationBuilder builder, bool option)
{
    // Passing 'option' as an argument throws a NotSupportedException at runtime.
    return builder.UseMiddleware<FactoryActivatedMiddleware>(option);
}
```

将工厂激活的中间件添加到 *Startup.cs* 的内置容器中：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<AppDbContext>(options =>
        options.UseInMemoryDatabase("InMemoryDb"));

    services.AddTransient<FactoryActivatedMiddleware>();

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

两个中间件均在 `Configure` 的请求处理管道中注册：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseConventionalMiddleware();
    app.UseFactoryActivatedMiddleware();

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseMvc();
}
```

IMiddlewareFactory

IMiddlewareFactory 提供中间件的创建方法。中间件工厂实现在容器中注册为作用域服务。

可在 Microsoft.AspNetCore.Http 包中找到默认的 `IMiddlewareFactory` 实现(即 MiddlewareFactory)。

其他资源

- ASP.NET Core 中间件
- 使用 ASP.NET Core 中的第三方容器激活中间件

使用 ASP.NET Core 中的第三方容器激活中间件

2018/10/30 • • Edit Online

作者：Luke Latham

本文演示如何使用 IMiddlewareFactory 和 IMiddleware 作为使用第三方容器激活中间件的可扩展点。有关 `IMiddlewareFactory` 和 `IMiddleware` 的介绍性信息，请参阅基于工厂的中间件激活主题。

查看或下载示例代码（如何下载）

示例应用演示了使用 `IMiddlewareFactory`、`SimpleInjectorMiddlewareFactory` 实现激活的中间件。此示例使用 Simple Injector 依赖项注入 (DI) 容器。

此示例的中间件实现记录了查询字符串参数（`key`）提供的值。中间件使用插入的数据库上下文（有作用域的服务）将查询字符串值记录在内存中数据库。

> **NOTE**
> 此示例应用仅出于演示目的使用 Simple Injector。不认可使用 Simple Injector。Simple Injector 文档中描述的中间件激活方法和 Simple Injector 维护人员推荐的 GitHub 问题。有关详细信息，请参阅 Simple Injector 文档和 Simple Injector GitHub 存储库。

IMiddlewareFactory

IMiddlewareFactory 提供中间件的创建方法。

在示例应用中，实现了中间件工厂以创建 `SimpleInjectorActivatedMiddleware` 实例。中间件工厂使用 Simple Injector 容器来解析中间件：

```
public class SimpleInjectorMiddlewareFactory : IMiddlewareFactory
{
    private readonly Container _container;

    public SimpleInjectorMiddlewareFactory(Container container)
    {
        _container = container;
    }

    public IMiddleware Create(Type middlewareType)
    {
        return _container.GetInstance(middlewareType) as IMiddleware;
    }

    public void Release(IMiddleware middleware)
    {
        // The container is responsible for releasing resources.
    }
}
```

IMiddleware

IMiddleware 定义应用的请求管道的中间件。

由 `IMiddlewareFactory` 实现 (Middleware/SimpleInjectorActivatedMiddleware.cs) 激活的中间件：

```
public class SimpleInjectorActivatedMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public SimpleInjectorActivatedMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation = "SimpleInjectorActivatedMiddleware",
                    Value = keyValue
                });

            await _db.SaveChangesAsync();
        }

        await next(context);
    }
}
```

为中间件创建扩展 (Middleware/MiddlewareExtensions.cs)：

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSimpleInjectorActivatedMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<SimpleInjectorActivatedMiddleware>();
    }
}
```

`Startup.ConfigureServices` 必须执行多项任务：

- 设置 Simple Injector 容器。
- 注册工厂和中间件。
- 为 Razor 页面的 Simple Injector 容器创建应用的数据库上下文。

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // Replace the default middleware factory with the
    // SimpleInjectorMiddlewareFactory.
    services.AddTransient<IMiddlewareFactory>(_ =>
    {
        return new SimpleInjectorMiddlewareFactory(_container);
    });

    // Wrap ASP.NET Core requests in a Simple Injector execution
    // context.
    services.UseSimpleInjectorAspNetRequestScoping(_container);

    // Provide the database context from the Simple
    // Injector container whenever it's requested from
    // the default service container.
    services.AddScoped<AppDbContext>(provider =>
        _container.GetInstance<AppDbContext>());

    _container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

    _container.Register<AppDbContext>(() =>
    {
        var optionsBuilder = new DbContextOptionsBuilder<DbContext>();
        optionsBuilder.UseInMemoryDatabase("InMemoryDb");
        return new AppDbContext(optionsBuilder.Options);
    }, Lifestyle.Scoped);

    _container.Register<SimpleInjectorActivatedMiddleware>();

    _container.Verify();
}
```

中间件在 `Startup.Configure` 的请求处理管道中注册:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseSimpleInjectorActivatedMiddleware();

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseMvc();
}
```

其他资源

- 中间件
- 基于工厂的中间件激活
- Simple Injector GitHub 存储库
- Simple Injector 文档

ASP.NET Core 中的 Web 主机和通用主机

2018/12/12 • • Edit Online

.NET 应用配置和启动主机。主机负责应用程序启动和生存期管理。两个主机 API 可供使用：

- Web 主机 – 适用于托管 Web 应用。
- 通用主机（ASP.NET Core 2.1 或更高版本）– 适用于托管非 Web 应用（例如，运行后台任务的应用）。在未来的版本中，通用主机将适用于托管任何类型的应用，包括 Web 应用。通用主机最终将取代 Web 主机。

为托管 ASP.NET Core Web 应用，开发人员应使用基于 IWebHostBuilder 的 Web 主机。为托管非 Web 应用，开发人员应使用基于 HostBuilder 的通用主机。

在 ASP.NET Core 中使用托管服务实现后台任务

了解如何在 ASP.NET Core 中使用托管服务实现后台任务。

在 ASP.NET Core 中使用承载启动程序集

了解如何使用 IHostingStartup 实现从引用或未引用程序集增强 ASP.NET Core 应用。

ASP.NET Core Web 主机

2019/1/10 • • Edit Online

作者：Luke Latham

对于本主题的 1.1 版本，请下载 ASP.NET Core Web 主机（版本 1.1，PDF）。

ASP.NET Core 应用配置和启动"主机"。主机负责应用程序启动和生存期管理。至少，主机配置服务器和请求处理管道。本主题介绍 ASP.NET Core Web 主机 (IWebHostBuilder)，它可用于托管 Web 应用。有关 .NET 通用主机 (IHostBuilder) 的介绍，请参阅 .NET 通用主机。

设置主机

创建使用 IWebHostBuilder 实例的主机。通常在应用的入口点来执行 `Main` 方法。在项目模板中，`Main` 位于 Program.cs。典型的 Program.cs 调用 CreateDefaultBuilder 以开始设置主机：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

`CreateDefaultBuilder` 执行下列任务：

- 使用应用的托管配置提供程序将 Kestrel 服务器配置为 Web 服务器。有关 Kestrel 服务器默认选项，请参阅 ASP.NET Core 中的 Kestrel Web 服务器实现。
- 将内容根设置为由 Directory.GetCurrentDirectory 返回的路径。
- 通过以下对象加载主机配置：
  - 前缀为 `ASPNETCORE_` 的环境变量（例如，`ASPNETCORE_ENVIRONMENT`）。
  - 命令行参数。
- 按以下顺序加载应用配置：
  - appsettings.json。
  - appsettings.{Environment}.json。
  - 应用在使用入口程序集的 `Development` 环境中运行时的机密管理器。
  - 环境变量。
  - 命令行参数。
- 配置控制台和调试输出的日志记录。日志记录包含 appsettings.json 或 appsettings.{Environment}.json 文件的日志记录配置部分中指定的日志筛选规则。
- 使用 ASP.NET Core 模块在 IIS 后面运行时，`CreateDefaultBuilder` 会启用 IIS 集成，这会配置应用的基址和端口。IIS 集成还配置应用以捕获启动错误。有关 IIS 默认选项，请参阅 使用 IIS 在 Windows 上托管 ASP.NET Core。
- 如果应用环境为"开发"，请将 ServiceProviderOptions.ValidateScopes 设为 `true`。有关详细信息，请参阅作用域验证。

ConfigureAppConfiguration、ConfigureLogging 以及 IWebHostBuilder 的其他方法和扩展方法可重写和增强 `CreateDefaultBuilder` 定义的配置。下面是一些示例：

- ConfigureAppConfiguration 用于指定应用的其他 `IConfiguration`。下面的 `ConfigureAppConfiguration` 调用添加委托，以在 appsettings.xml 文件中添加应用配置。可多次调用 `ConfigureAppConfiguration`。请注意，此配置不适用于主机（例如，服务器 URL 或环境）。请参阅主机配置值部分。

```
WebHost.CreateDefaultBuilder(args)
    .ConfigureAppConfiguration((hostingContext, config) =>
    {
        config.AddXmlFile("appsettings.xml", optional: true, reloadOnChange: true);
    })
    ...
```

- 下面的 `ConfigureLogging` 调用添加委托，以将最小日志记录级别 (SetMinimumLevel) 配置为 LogLevel.Warning。此设置重写 `CreateDefaultBuilder` 在 appsettings.Development.json 和 appsettings.Production.json 中配置的设置，分别为 `LogLevel.Debug` 和 `LogLevel.Error`。可多次调用 `ConfigureLogging`。

```
WebHost.CreateDefaultBuilder(args)
    .ConfigureLogging(logging =>
    {
        logging.SetMinimumLevel(LogLevel.Warning);
    })
    ...
```

- 下面调用 `ConfigureKestrel` 来重写 `CreateDefaultBuilder` 在配置 Kestrel 时建立的 30,000,000 字节默认 Limits.MaxRequestBodySize：

```
WebHost.CreateDefaultBuilder(args)
    .ConfigureKestrel((context, options) =>
    {
        options.Limits.MaxRequestBodySize = 20000000;
    });
```

- 下面调用 UseKestrel 来重写 `CreateDefaultBuilder` 在配置 Kestrel 时建立的 30,000,000 字节默认 Limits.MaxRequestBodySize：

```
WebHost.CreateDefaultBuilder(args)
    .UseKestrel(options =>
    {
        options.Limits.MaxRequestBodySize = 20000000;
    });
```

内容根确定主机搜索内容文件（如 MVC 视图文件）的位置。应用从项目的根文件夹启动时，会将项目的根文件夹用作内容根。这是 Visual Studio 和 dotnet new 模板中使用的默认值。

有关应用配置的详细信息，请参阅 ASP.NET Core 中的配置。

> **NOTE**
>
> 作为使用静态 `CreateDefaultBuilder` 方法的替代方法，从 WebHostBuilder 创建主机是一种受 ASP.NET Core 2.x 支持的方法。有关详细信息，请参阅 ASP.NET Core 1.x 选项卡。

设置主机时，可以提供配置和 ConfigureServices 方法。如果指定 `Startup` 类，必须定义 `Configure` 方法。有关更多信息，请参见ASP.NET Core 中的应用启动。多次调用 `ConfigureServices` 将追加到另一个。多次调用 `WebHostBuilder` 上的 `Configure` 或 `UseStartup` 将替换以前的设置。

主机配置值

WebHostBuilder 依赖于以下的方法设置主机配置值：

- 主机生成器配置，其中包括格式 `ASPNETCORE_{configurationKey}` 的环境变量。例如 `ASPNETCORE_ENVIRONMENT`。
- UseContentRoot 和 UseConfiguration 等扩展（请参阅重写配置部分）。
- UseSetting 和关联键。使用 `UseSetting` 设置值时，该值设置为无论何种类型的字符串。

主机使用任何一个选项设置上一个值。有关详细信息，请参阅下一部分中的重写配置。

应用程序键（名称）

在主机构造期间调用 UseStartup 或 Configure 时，会自动设置 IHostingEnvironment.ApplicationName 属性。该值设置为包含应用入口点的程序集的名称。要显式设置值，请使用 WebHostDefaults.ApplicationKey：

密钥：applicationName
类型：string
默认值：包含应用入口点的程序集的名称。
设置使用：`UseSetting`
环境变量：`ASPNETCORE_APPLICATIONNAME`

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.ApplicationKey, "CustomApplicationName")
```

捕获启动错误

此设置控制启动错误的捕获。

键：captureStartupErrors
类型：布尔型（`true` 或 `1`）
默认值：默认为 `false`，除非应用使用 Kestrel 在 IIS 后方运行，其中默认值是 `true`。
设置使用：`CaptureStartupErrors`
环境变量：`ASPNETCORE_CAPTURESTARTUPERRORS`

当 `false` 时，启动期间出错导致主机退出。当 `true` 时，主机在启动期间捕获异常并尝试启动服务器。

```
WebHost.CreateDefaultBuilder(args)
    .CaptureStartupErrors(true)
```

内容根

此设置确定 ASP.NET Core 开始搜索内容文件，如 MVC 视图等。

键：contentRoot
类型：string
默认值：默认为应用程序集所在的文件夹。
设置使用：`UseContentRoot`
环境变量：`ASPNETCORE_CONTENTROOT`

内容根也用作 Web 根设置的基路径。如果路径不存在，主机将无法启动。

```
WebHost.CreateDefaultBuilder(args)
    .UseContentRoot("c:\\<content-root>")
```

详细错误

确定是否应捕获详细错误。

键：detailedErrors
类型：布尔型（`true` 或 `1`）
默认值：false
设置使用：`UseSetting`
环境变量：`ASPNETCORE_DETAILEDERRORS`

启用（或当环境设置为 `Development` ）时，应用捕获详细的异常。

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.DetailedErrorsKey, "true")
```

环境

设置应用的环境。

键：环境
类型：string
默认值：生产
设置使用：`UseEnvironment`
环境变量：`ASPNETCORE_ENVIRONMENT`

环境可以设置为任何值。框架定义的值包括 `Development``Staging` 和 `Production` 。值不区分大小写。默认情况下，从 `ASPNETCORE_ENVIRONMENT` 环境变量读取环境。使用 Visual Studio 时，可能会在 launchSettings.json 文件中设置环境变量。有关更多信息，请参见在 ASP.NET Core 中使用多个环境。

```
WebHost.CreateDefaultBuilder(args)
    .UseEnvironment(EnvironmentName.Development)
```

承载启动程序集

设置应用的承载启动程序集。

键：hostingStartupAssemblies
类型：string
默认值：空字符串
设置使用：`UseSetting`
环境变量：`ASPNETCORE_HOSTINGSTARTUPASSEMBLIES`

承载启动程序集的以分号分隔的字符串在启动时加载。

虽然配置值默认为空字符串，但是承载启动程序集会始终包含应用的程序集。提供承载启动程序集时，当应用在启动过程中生成其公用服务时将它们添加到应用的程序集加载。

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.HostingStartupAssembliesKey, "assembly1;assembly2")
```

HTTPS 端口

设置 HTTPS 重定向端口。用于强制实施 HTTPS。

键：https_port；类型：字符串；默认值：未设置默认值。设置使用：`UseSetting` 环境变量：`ASPNETCORE_HTTPS_PORT`

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting("https_port", "8080")
```

承载启动排除程序集

承载启动程序集的以分号分隔的字符串在启动时排除。

键：hostingStartupExcludeAssemblies
类型：string
默认值：空字符串
设置使用：`UseSetting`
环境变量：`ASPNETCORE_HOSTINGSTARTUPEXCLUDEASSEMBLIES`

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.HostingStartupExcludeAssembliesKey, "assembly1;assembly2")
```

首选承载 URL

指示主机是否应该侦听使用 `WebHostBuilder` 配置的 URL，而不是使用 `IServer` 实现配置的 URL。

键：preferHostingUrls
类型：布尔型（`true` 或 `1`）
默认值：true
设置使用：`PreferHostingUrls`
环境变量：`ASPNETCORE_PREFERHOSTINGURLS`

```
WebHost.CreateDefaultBuilder(args)
    .PreferHostingUrls(false)
```

阻止承载启动

阻止承载启动程序集自动加载，包括应用的程序集所配置的承载启动程序集。有关更多信息，请参见在 ASP.NET Core 中使用承载启动程序集。

键：preventHostingStartup
类型：布尔型（`true` 或 `1`）
默认值：false
设置使用：`UseSetting`
环境变量：`ASPNETCORE_PREVENTHOSTINGSTARTUP`

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.PreventHostingStartupKey, "true")
```

服务器 URL

指示 IP 地址或主机地址，其中包含服务器应针对请求侦听的端口和协议。

键：urls
类型：string
默认：http://localhost:5000
设置使用：`UseUrls`
环境变量：`ASPNETCORE_URLS`

设置为服务器应响应的以分号分隔 (;) 的 URL 前缀列表。例如 `http://localhost:123`。使用“*”指示服务器应针

对请求侦听的使用特定端口和协议(例如 `http://*:5000`)的 IP 地址或主机名。协议(`http://` 或 `https://`)必须包含每个 URL。不同的服务器支持的格式有所不同。

```
WebHost.CreateDefaultBuilder(args)
    .UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002")
```

Kestrel 具有自己的终结点配置 API。有关更多信息，请参见ASP.NET Core 中的 Kestrel Web 服务器实现。

关闭超时

指定等待 Web 主机关闭的时长。

键：shutdownTimeoutSeconds
类型：int
默认值：5
设置使用：`UseShutdownTimeout`
环境变量：`ASPNETCORE_SHUTDOWNTIMEOUTSECONDS`

虽然键使用 `UseSetting` 接受 int(例如 `.UseSetting(WebHostDefaults.ShutdownTimeoutKey, "10")`)，但是 UseShutdownTimeout 扩展方法采用 TimeSpan。

在超时时间段中，托管：

- 触发器 IApplicationLifetime.ApplicationStopping。
- 尝试停止托管服务，对服务停止失败的任何错误进行日志记录。

如果在所有托管服务停止之前就达到了超时时间，则会在应用关闭时会终止剩余的所有活动的服务。即使没有完成处理工作，服务也会停止。如果停止服务需要额外的时间，请增加超时时间。

```
WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
```

启动程序集

确定要在其中搜索 `Startup` 类的程序集。

键：startupAssembly
类型：string
默认值：应用的程序集
设置使用：`UseStartup`
环境变量：`ASPNETCORE_STARTUPASSEMBLY`

按名称(`string`)或类型(`TStartup`)的程序集可以引用。如果调用多个 `UseStartup` 方法，优先选择最后一个方法。

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup("StartupAssemblyName")
```

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<TStartup>()
```

Web 根路径

设置应用的静态资产的相对路径。

键：webroot
类型：string

默认值：如果未指定，默认值是“(Content Root)/wwwroot”（如果该路径存在）。如果该路径不存在，则使用无操作文件提供程序。
设置使用：`UseWebRoot`
环境变量：`ASPNETCORE_WEBROOT`

```
WebHost.CreateDefaultBuilder(args)
    .UseWebRoot("public")
```

重写配置

配置可用于配置 Web 主机。在下面的示例中，主机配置是根据需要在 hostsettings.json 文件中指定。命令行参数可能会重写从 hostsettings.json 文件加载的任何配置。生成的配置（在 `config` 中）用于通过 UseConfiguration 配置主机。`IWebHostBuilder` 配置会添加到应用配置中，但反之不亦然—`ConfigureAppConfiguration` 不影响 `IWebHostBuilder` 配置。

先用 hostsettings.json config 重写 `UseUrls` 提供的配置，再用命令行参数 config：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("hostsettings.json", optional: true)
            .AddCommandLine(args)
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseUrls("http://*:5000")
            .UseConfiguration(config)
            .Configure(app =>
            {
                app.Run(context =>
                    context.Response.WriteAsync("Hello, World!"));
            });
    }
}
```

hostsettings.json：

```
{
    urls: "http://*:5005"
}
```

> **NOTE**
>
> UseConfiguration 扩展方法当前不能分析由 `GetSection` 返回的配置部分(例如 `.UseConfiguration(Configuration.GetSection("section"))`)。`GetSection` 方法将配置键筛选到所请求的部分，但将节名称保留在键上(例如 `section:urls`、`section:environment`)。`UseConfiguration` 方法需要键来匹配 `WebHostBuilder` 键(例如 `urls`、`environment`)。键上存在的节名称阻止节的值配置主机。将在即将发布的版本中解决此问题。有关详细信息和解决方法，请参阅将配置节传入到 WebHostBuilder.UseConfiguration 使用完整的键。
>
> `UseConfiguration` 只将所提供的 `IConfiguration` 中的密钥复制到主机生成器配置中。因此，JSON、INI 和 XML 设置文件的设置 `reloadOnChange: true` 没有任何影响。

若要指定在特定的 URL 上运行的主机，所需的值可以在执行 dotnet 运行时从命令提示符传入。命令行参数重写 hostsettings.json 文件中的 `urls` 值，且服务器侦听端口 8080：

```
dotnet run --urls "http://*:8080"
```

管理主机

运行

`Run` 方法启动 Web 应用并阻止调用线程，直到关闭主机：

```
host.Run();
```

Start

通过调用 `Start` 方法以非阻止方式运行主机：

```
using (host)
{
    host.Start();
    Console.ReadLine();
}
```

如果 URL 列表传递给 `Start` 方法，该列表侦听指定的 URL：

```
var urls = new List<string>()
{
    "http://*:5000",
    "http://localhost:5001"
};

var host = new WebHostBuilder()
    .UseKestrel()
    .UseStartup<Startup>()
    .Start(urls.ToArray());

using (host)
{
    Console.ReadLine();
}
```

应用可以使用通过静态便捷方法预配置的 `CreateDefaultBuilder` 默认值初始化并启动新的主机。这些方法在没有控制台输出的情况下启动服务器，并使用 WaitForShutdown 等待中断(Ctrl-C/SIGINT 或 SIGTERM)：

Start(RequestDelegate app)

从 `RequestDelegate` 开始：

```
using (var host = WebHost.Start(app => app.Response.WriteAsync("Hello, World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

在浏览器中向 `http://localhost:5000` 发出请求，接收响应“Hello World!” `WaitForShutdown` 受到阻止，直到发出中断（Ctrl-C/SIGINT 或 SIGTERM）。应用显示 `Console.WriteLine` 消息并等待 keypress 退出。

Start(string url, RequestDelegate app)

从 URL 和 `RequestDelegate` 开始：

```
using (var host = WebHost.Start("http://localhost:8080", app => app.Response.WriteAsync("Hello, World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

生成与 Start(RequestDelegate app) 相同的结果，除非应用在 `http://localhost:8080` 上响应。

Start(Action<IRouteBuilder> routeBuilder)

使用 `IRouteBuilder` 的实例 (Microsoft.AspNetCore.Routing) 用于路由中间件：

```
using (var host = WebHost.Start(router => router
    .MapGet("hello/{name}", (req, res, data) =>
        res.WriteAsync($"Hello, {data.Values["name"]}!"))
    .MapGet("buenosdias/{name}", (req, res, data) =>
        res.WriteAsync($"Buenos dias, {data.Values["name"]}!"))
    .MapGet("throw/{message?}", (req, res, data) =>
        throw new Exception((string)data.Values["message"] ?? "Uh oh!"))
    .MapGet("{greeting}/{name}", (req, res, data) =>
        res.WriteAsync($"{data.Values["greeting"]}, {data.Values["name"]}!"))
    .MapGet("", (req, res, data) => res.WriteAsync("Hello, World!"))))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

该示例中使用以下浏览器请求：

请求	响应
`http://localhost:5000/hello/Martin`	Hello, Martin!
`http://localhost:5000/buenosdias/Catrina`	Buenos dias, Catrina!
`http://localhost:5000/throw/ooops!`	使用“ooops!”字符串引发异常
`http://localhost:5000/throw`	使用“Uh oh!”字符串引发异常
`http://localhost:5000/Sante/Kevin`	Sante, Kevin!
`http://localhost:5000`	Hello World!

`WaitForShutdown` 受到阻止，直到发出中断（Ctrl-C/SIGINT 或 SIGTERM）。应用显示 `Console.WriteLine` 消息并等待 keypress 退出。

Start(string url, Action<IRouteBuilder> routeBuilder)

使用 URL 和 `IRouteBuilder` 实例：

```
using (var host = WebHost.Start("http://localhost:8080", router => router
    .MapGet("hello/{name}", (req, res, data) =>
        res.WriteAsync($"Hello, {data.Values["name"]}!"))
    .MapGet("buenosdias/{name}", (req, res, data) =>
        res.WriteAsync($"Buenos dias, {data.Values["name"]}!"))
    .MapGet("throw/{message?}", (req, res, data) =>
        throw new Exception((string)data.Values["message"] ?? "Uh oh!"))
    .MapGet("{greeting}/{name}", (req, res, data) =>
        res.WriteAsync($"{data.Values["greeting"]}, {data.Values["name"]}!"))
    .MapGet("", (req, res, data) => res.WriteAsync("Hello, World!"))))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

生成与 Start(Action<IRouteBuilder> routeBuilder) 相同的结果，除非应用在 `http://localhost:8080` 上响应。

StartWith(Action<IApplicationBuilder> app)

提供委托以配置 `IApplicationBuilder`：

```
using (var host = WebHost.StartWith(app =>
    app.Use(next =>
    {
        return async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        };
    })))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

在浏览器中向 `http://localhost:5000` 发出请求，接收响应“Hello World!” `WaitForShutdown` 受到阻止，直到发出中断（Ctrl-C/SIGINT 或 SIGTERM）。应用显示 `Console.WriteLine` 消息并等待 keypress 退出。

StartWith(string url, Action<IApplicationBuilder> app)

提供 URL 和委托以配置 `IApplicationBuilder`：

```
using (var host = WebHost.StartWith("http://localhost:8080", app =>
    app.Use(next =>
    {
        return async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        };
    })))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

生成与 StartWith(Action<IApplicationBuilder> app) 相同的结果，除非应用在 `http://localhost:8080` 上响应。

IHostingEnvironment 接口

IHostingEnvironment 接口提供有关应用的 Web 承载环境的信息。使用构造函数注入获取 `IHostingEnvironment` 以使用其属性和扩展方法：

```
public class CustomFileReader
{
    private readonly IHostingEnvironment _env;

    public CustomFileReader(IHostingEnvironment env)
    {
        _env = env;
    }

    public string ReadFile(string filePath)
    {
        var fileProvider = _env.WebRootFileProvider;
        // Process the file here
    }
}
```

基于约定的方法可以用于在启动时基于环境配置应用。或者，将 `IHostingEnvironment` 注入到 `Startup` 构造函数用于 `ConfigureServices`：

```
public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        HostingEnvironment = env;
    }

    public IHostingEnvironment HostingEnvironment { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        if (HostingEnvironment.IsDevelopment())
        {
            // Development configuration
        }
        else
        {
            // Staging/Production configuration
        }

        var contentRootPath = HostingEnvironment.ContentRootPath;
    }
}
```

> **NOTE**
>
> 除了 `IsDevelopment` 扩展方法，`IHostingEnvironment` 提供 `IsStaging`、`IsProduction` 和 `IsEnvironment(string environmentName)` 方法。有关更多信息，请参见在 ASP.NET Core 中使用多个环境。

`IHostingEnvironment` 服务还可以直接注入到 `Configure` 方法以设置处理管道：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        // In Development, use the developer exception page
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // In Staging/Production, route exceptions to /error
        app.UseExceptionHandler("/error");
    }

    var contentRootPath = env.ContentRootPath;
}
```

创建自定义中间件时可以将 `IHostingEnvironment` 注入 `Invoke` 方法：

```
public async Task Invoke(HttpContext context, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        // Configure middleware for Development
    }
    else
    {
        // Configure middleware for Staging/Production
    }

    var contentRootPath = env.ContentRootPath;
}
```

IApplicationLifetime 接口

IApplicationLifetime 允许后启动和关闭活动。接口上的三个属性是用于注册 `Action` 方法（用于定义启动和关闭事件）的取消标记。

取消标记	触发条件
ApplicationStarted	主机已完全启动。
ApplicationStopped	主机正在完成正常关闭。应处理所有请求。关闭受到阻止，直到完成此事件。
ApplicationStopping	主机正在执行正常关闭。仍在处理请求。关闭受到阻止，直到完成此事件。

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IApplicationLifetime appLifetime)
    {
        appLifetime.ApplicationStarted.Register(OnStarted);
        appLifetime.ApplicationStopping.Register(OnStopping);
        appLifetime.ApplicationStopped.Register(OnStopped);

        Console.CancelKeyPress += (sender, eventArgs) =>
        {
            appLifetime.StopApplication();
            // Don't terminate the process immediately, wait for the Main thread to exit gracefully.
            eventArgs.Cancel = true;
        };
    }

    private void OnStarted()
    {
        // Perform post-startup activities here
    }

    private void OnStopping()
    {
        // Perform on-stopping activities here
    }

    private void OnStopped()
    {
        // Perform post-stopped activities here
    }
}
```

StopApplication 请求应用终止。以下类在调用类的 `Shutdown` 方法时使用 `StopApplication` 正常关闭应用：

```
public class MyClass
{
    private readonly IApplicationLifetime _appLifetime;

    public MyClass(IApplicationLifetime appLifetime)
    {
        _appLifetime = appLifetime;
    }

    public void Shutdown()
    {
        _appLifetime.StopApplication();
    }
}
```

作用域验证

如果应用环境为"开发"，则 CreateDefaultBuilder 将 ServiceProviderOptions.ValidateScopes 设为 `true`。

若将 `ValidateScopes` 设为 `true`，默认服务提供程序会执行检查来验证以下内容：

- 没有从根服务提供程序直接或间接解析到有作用域的服务。
- 未将有作用域的服务直接或间接注入到单一实例。

调用 BuildServiceProvider 时，会创建根服务提供程序。在启动提供程序和应用时，根服务提供程序的生存期对应于应用/服务的生存期，并在关闭应用时释放。

有作用域的服务由创建它们的容器释放。如果作用域创建于根容器，则该服务的生存会有效地提升至单一实例，

因为根容器只会在应用/服务关闭时将其释放。验证服务作用域，将在调用 `BuildServiceProvider` 时收集这类情况。

若要始终验证作用域（包括在生存环境中验证），请使用主机生成器上的 UseDefaultServiceProvider 配置 ServiceProviderOptions：

```
WebHost.CreateDefaultBuilder(args)
    .UseDefaultServiceProvider((context, options) => {
        options.ValidateScopes = true;
    })
```

其他资源

- 使用 IIS 在 Windows 上托管 ASP.NET Core
- 使用 Nginx 在 Linux 上托管 ASP.NET Core
- 使用 Apache 在 Linux 上托管 ASP.NET Core
- 在 Windows 服务中托管 ASP.NET Core

.NET 通用主机

2018/12/12 • • Edit Online

作者：Luke Latham

.NET Core 应用配置和启动"主机"。主机负责应用程序启动和生存期管理。本主题介绍 ASP.NET Core 通用主机 (HostBuilder)，该主机对于托管不处理 HTTP 请求的应用非常有用。有关 Web 主机 (WebHostBuilder) 的介绍，请参阅 ASP.NET Core Web 主机。

通用主机的目标是将 HTTP 管道从 Web 主机 API 中分离出来，从而启用更多的主机方案。基于通用主机的消息、后台任务和其他非 HTTP 工作负载可从横切功能（如配置、依赖关系注入 [DI] 和日志记录）中受益。

通用主机是 ASP.NET Core 2.1 中的新增功能，不适用于 Web 承载方案。对于 Web 承载方案，请使用 Web 主机。通用主机正处于开发阶段，用于在未来版本中替换 Web 主机，并在 HTTP 和非 HTTP 方案中充当主要的主机 API。

查看或下载示例代码（如何下载）

在 Visual Studio Code 中运行示例应用时，请使用外部或集成终端。请勿在 `internalConsole` 中运行示例。

在 Visual Studio Code 中设置控制台：

1. 打开 .vscode/launch.json 文件。
2. 在 .NET Core 启动（控制台）配置中，找到控制台条目。将值设置为 `externalTerminal` 或 `integratedTerminal` 。

介绍

通用主机库位于 Microsoft.Extensions.Hosting 命名空间中，由 Microsoft.Extensions.Hosting 包提供。Microsoft.AspNetCore.App 元包（ASP.NET Core 2.1 或更高版本）中包括 `Microsoft.Extensions.Hosting` 包。

IHostedService 是执行代码的入口点。每个 `IHostedService` 实现都按照 ConfigureServices 中服务注册的顺序执行。主机启动时，每个 `IHostedService` 上都会调用 StartAsync，主机正常关闭时，以反向注册顺序调用 StopAsync。

设置主机

IHostBuilder 是供库和应用初始化、生成和运行主机的主要组件：

```
public static async Task Main(string[] args)
{
    var host = new HostBuilder()
        .Build();

    await host.RunAsync();
}
```

选项

HostOptions 配置 IHost 的选项。

关闭超时值

ShutdownTimeout 设置 StopAsync 的超时值。默认值为 5 秒。

`Program.Main` 中的以下选项配置将默认值为 5 秒的关闭超时值增加至 20 秒：

```
var host = new HostBuilder()
    .ConfigureServices((hostContext, services) =>
    {
        services.Configure<HostOptions>(option =>
        {
            option.ShutdownTimeout = System.TimeSpan.FromSeconds(20);
        });
    })
    .Build();
```

默认服务

在主机初始化期间注册以下服务：

- 环境 (IHostingEnvironment)
- HostBuilderContext
- 配置 (IConfiguration)
- IApplicationLifetime (ApplicationLifetime)
- IHostLifetime (ConsoleLifetime)
- IHost
- 选项 (AddOptions)
- 日志记录 (AddLogging)

主机配置

主机配置的创建方式如下：

- 调用 IHostBuilder 上的扩展方法以设置"内容根"和"环境"。
- 从 ConfigureHostConfiguration 中的配置提供程序读取配置。

扩展方法

应用程序键（名称）

IHostingEnvironment.ApplicationName 属性是在主机构造期间通过主机配置设定的。要显式设置值，请使用 HostDefaults.ApplicationKey：

密钥：applicationName
类型：string
默认：包含应用入口点的程序集的名称。
设置使用：`HostBuilderContext.HostingEnvironment.ApplicationName`
环境变量：`<PREFIX_>APPLICATIONNAME`（`<PREFIX_>` 是用户定义的可选前缀）

内容根

此设置确定主机从哪里开始搜索内容文件。

键：contentRoot
类型：string
默认值：默认为应用程序集所在的文件夹。
设置使用：`UseContentRoot`
环境变量：`<PREFIX_>CONTENTROOT`（`<PREFIX_>` 是用户定义的可选前缀）

如果路径不存在，主机将无法启动。

```
var host = new HostBuilder()
    .UseContentRoot("c:\\<content-root>")
```

环境

设置应用的环境。

键：环境
类型：string
默认值：生产
设置使用：`UseEnvironment`
环境变量：`<PREFIX_>ENVIRONMENT`（`<PREFIX_>` 是用户定义的可选前缀）

环境可以设置为任何值。框架定义的值包括 `Development``Staging` 和 `Production`。值不区分大小写。

```
var host = new HostBuilder()
    .UseEnvironment(EnvironmentName.Development)
```

ConfigureHostConfiguration

`ConfigureHostConfiguration` 使用 IConfigurationBuilder 来为主机创建 IConfiguration。主机配置用于初始化 IHostingEnvironment，以供在应用的构建过程中使用。

可多次调用 `ConfigureHostConfiguration`，并得到累计结果。主机使用上一次在一个给定键上设置值的选项。

主机配置自动流向应用配置（ConfigureAppConfiguration 和应用的其余部分）。

默认情况下不包括提供程序。必须在 `ConfigureHostConfiguration` 中显式指定应用所需的任何配置提供程序，包括：

- 文件配置（例如，来自 hostsettings.json 文件）。
- 环境变量配置。
- 命令行参数配置。
- 任何其他所需的配置提供程序。

通过使用 `SetBasePath` 指定应用的基本路径，然后调用其中一个文件配置提供程序，可以启用主机的文件配置。示例应用使用 JSON 文件 hostsettings.json，并调用 AddJsonFile 来使用文件的主机配置设置。

要添加主机的环境变量配置，请在主机生成器上调用 AddEnvironmentVariables。`AddEnvironmentVariables` 接受用户定义的前缀（可选）。示例应用使用前缀 `PREFIX_`。当系统读取环境变量时，便会删除前缀。配置示例应用的主机后，`PREFIX_ENVIRONMENT` 的环境变量值就变成 `environment` 密钥的主机配置值。

在开发过程中，如果使用 Visual Studio 或通过 `dotnet run` 运行应用，可能会在 Properties/launchSettings.json 文件中设置环境变量。若在开发过程中使用 Visual Studio Code，可能会在 .vscode/launch.json 文件中设置环境变量。有关更多信息，请参见在 ASP.NET Core 中使用多个环境。

通过调用 AddCommandLine 可添加命令行配置。最后添加命令行配置以允许命令行参数替代之前配置提供程序提供的配置。

hostsettings.json：

```
{
  "environment": "Development"
}
```

可以通过 applicationName 和 contentRoot 键提供其他配置。

示例 `HostBuilder` 配置使用 `ConfigureHostConfiguration`：

```
var host = new HostBuilder()
    .ConfigureHostConfiguration(configHost =>
    {
        configHost.SetBasePath(Directory.GetCurrentDirectory());
        configHost.AddJsonFile("hostsettings.json", optional: true);
        configHost.AddEnvironmentVariables(prefix: "PREFIX_");
        configHost.AddCommandLine(args);
    })
```

ConfigureAppConfiguration

通过在 IHostBuilder 实现上调用 ConfigureAppConfiguration 创建应用配置。`ConfigureAppConfiguration` 使用 IConfigurationBuilder 来为应用创建 IConfiguration。可多次调用 `ConfigureAppConfiguration`，并得到累计结果。应用使用上一次在一个给定键上设置值的选项。HostBuilderContext.Configuration 中提供 `ConfigureAppConfiguration` 创建的配置，以供进行后续操作和在 Services 中使用。

应用配置会自动接收 ConfigureHostConfiguration 提供的主机配置。

示例应用配置使用 `ConfigureAppConfiguration`：

```
var host = new HostBuilder()
    .ConfigureAppConfiguration((hostContext, configApp) =>
    {
        configApp.SetBasePath(Directory.GetCurrentDirectory());
        configApp.AddJsonFile("appsettings.json", optional: true);
        configApp.AddJsonFile(
            $"appsettings.{hostContext.HostingEnvironment.EnvironmentName}.json",
            optional: true);
        configApp.AddEnvironmentVariables(prefix: "PREFIX_");
        configApp.AddCommandLine(args);
    })
```

appsettings.json：

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

appsettings.Development.json：

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

appsettings.Production.json：

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Error",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

要将设置文件移动到输出目录，请在项目文件中将设置文件指定为 MSBuild 项目项。示例应用移动具有以下 `<Content>` 项的 JSON 应用设置文件和 hostsettings.json：

```
<ItemGroup>
  <Content Include="**\*.json" Exclude="bin\**\*;obj\**\*" CopyToOutputDirectory="PreserveNewest" />
</ItemGroup>
```

ConfigureServices

ConfigureServices 将服务添加到应用的依赖关系注入容器。可多次调用 `ConfigureServices`，并得到累计结果。

托管服务是一个类，具有实现 IHostedService 接口的后台任务逻辑。有关更多信息，请参见在 ASP.NET Core 中使用托管服务实现后台任务。

示例应用使用 `AddHostedService` 扩展方法向应用添加生存期事件 `LifetimeEventsHostedService` 和定时后台任务 `TimedHostedService` 服务：

```
var host = new HostBuilder()
    .ConfigureServices((hostContext, services) =>
    {
        if (hostContext.HostingEnvironment.IsDevelopment())
        {
            // Development service configuration
        }
        else
        {
            // Non-development service configuration
        }

        services.AddHostedService<LifetimeEventsHostedService>();
        services.AddHostedService<TimedHostedService>();
    })
```

ConfigureLogging

ConfigureLogging 添加了一个委托来配置提供的 ILoggingBuilder。可以利用相加结果多次调用 `ConfigureLogging`。

```
var host = new HostBuilder()
    .ConfigureLogging((hostContext, configLogging) =>
    {
        configLogging.AddConsole();
        configLogging.AddDebug();
    })
```

UseConsoleLifetime

UseConsoleLifetime 侦听 `Ctrl+C` /SIGINT 或 SIGTERM 并调用 StopApplication 来启动关闭进程。`UseConsoleLifetime` 解除阻止 RunAsync 和 WaitForShutdownAsync 等扩展。ConsoleLifetime 预注册为默认生存期实现。使用注册的最后一个生存期。

```
var host = new HostBuilder()
    .UseConsoleLifetime()
```

容器配置

为支持插入其他容器中，主机可以接受 IServiceProviderFactory<TContainerBuilder>。提供工厂不属于 DI 容器注册，而是用于创建具体 DI 容器的主机内部函数。UseServiceProviderFactory(IServiceProviderFactory<TContainerBuilder>) 重写用于创建应用的服务提供程序的默认工厂。

ConfigureContainer 方法托管自定义容器配置。`ConfigureContainer` 提供在基础主机 API 的基础之上配置容器的强类型体验。可以利用相加结果多次调用 `ConfigureContainer`。

为应用创建服务容器：

```
namespace GenericHostSample
{
    internal class ServiceContainer
    {
    }
}
```

提供服务容器工厂：

```
using System;
using Microsoft.Extensions.DependencyInjection;

namespace GenericHostSample
{
    internal class ServiceContainerFactory : IServiceProviderFactory<ServiceContainer>
    {
        public ServiceContainer CreateBuilder(IServiceCollection services)
        {
            return new ServiceContainer();
        }

        public IServiceProvider CreateServiceProvider(ServiceContainer containerBuilder)
        {
            throw new NotImplementedException();
        }
    }
}
```

使用该工厂并为应用配置自定义服务容器：

```
var host = new HostBuilder()
    .UseServiceProviderFactory<ServiceContainer>(new ServiceContainerFactory())
    .ConfigureContainer<ServiceContainer>((hostContext, container) =>
    {
    })
```

扩展性

在 `IHostBuilder` 上使用扩展方法实现主机扩展性。以下示例介绍扩展方法如何使用 在 ASP.NET Core 中使用托管服务实现后台任务 中所示的 TimedHostedService 示例来扩展 `IHostBuilder` 实现。

```
var host = new HostBuilder()
    .UseHostedService<TimedHostedService>()
    .Build();

await host.StartAsync();
```

应用建立 `UseHostedService` 扩展方法，以注册在 `T` 中传递的托管服务：

```
using System;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

public static class Extensions
{
    public static IHostBuilder UseHostedService<T>(this IHostBuilder hostBuilder)
        where T : class, IHostedService, IDisposable
    {
        return hostBuilder.ConfigureServices(services =>
            services.AddHostedService<T>());
    }
}
```

管理主机

IHost 实现负责启动和停止服务容器中注册的 `IHostedService` 实现。

运行

Run 运行应用并阻止调用线程，直到关闭主机：

```
public class Program
{
    public void Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        host.Run();
    }
}
```

RunAsync

RunAsync 运行应用并返回在触发取消令牌或关闭时完成的 `Task`：

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        await host.RunAsync();
    }
}
```

RunConsoleAsync

RunConsoleAsync 启用控制台支持、生成和启动主机，以及等待 `Ctrl+C` /SIGINT 或 SIGTERM 关闭。

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var hostBuilder = new HostBuilder();

        await hostBuilder.RunConsoleAsync();
    }
}
```

Start 和 StopAsync

Start 同步启动主机。

StopAsync 尝试在提供的超时时间内停止主机。

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        using (host)
        {
            host.Start();

            await host.StopAsync(TimeSpan.FromSeconds(5));
        }
    }
}
```

StartAsync 和 StopAsync

StartAsync 启动应用。

StopAsync 停止应用。

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        using (host)
        {
            await host.StartAsync();

            await host.StopAsync();
        }
    }
}
```

WaitForShutdown

WaitForShutdown 通过 IHostLifetime 触发，例如 ConsoleLifetime（侦听 `Ctrl+C` /SIGINT 或 SIGTERM）。`WaitForShutdown` 调用 StopAsync。

```
public class Program
{
    public void Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        using (host)
        {
            host.Start();

            host.WaitForShutdown();
        }
    }
}
```

WaitForShutdownAsync

WaitForShutdownAsync 返回在通过给定的令牌和调用 StopAsync 来触发关闭时完成的 `Task`。

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        using (host)
        {
            await host.StartAsync();

            await host.WaitForShutdownAsync();
        }

    }
}
```

外部控件

使用可从外部调用的方法，能够实现主机的外部控件：

```
public class Program
{
    private IHost _host;

    public Program()
    {
        _host = new HostBuilder()
            .Build();
    }

    public async Task StartAsync()
    {
        _host.StartAsync();
    }

    public async Task StopAsync()
    {
        using (_host)
        {
            await _host.StopAsync(TimeSpan.FromSeconds(5));
        }
    }
}
```

在 StartAsync 开始时调用 WaitForStartAsync，在继续之前，会一直等待该操作完成。它可用于延迟启动，直到外部事件发出信号。

IHostingEnvironment 接口

IHostingEnvironment 提供有关应用托管环境的信息。使用构造函数注入获取 `IHostingEnvironment` 以使用其属性和扩展方法：

```
public class MyClass
{
    private readonly IHostingEnvironment _env;

    public MyClass(IHostingEnvironment env)
    {
        _env = env;
    }

    public void DoSomething()
    {
        var environmentName = _env.EnvironmentName;
    }
}
```

有关更多信息，请参见在 ASP.NET Core 中使用多个环境。

IApplicationLifetime 接口

IApplicationLifetime 允许启动后和关闭活动，包括正常关闭请求。接口上的三个属性是用于注册 `Action` 方法（用于定义启动和关闭事件）的取消标记。

取消标记	触发条件
ApplicationStarted	主机已完全启动。

取消标记	触发条件
ApplicationStopped	主机正在完成正常关闭。应处理所有请求。关闭受到阻止，直到完成此事件。
ApplicationStopping	主机正在执行正常关闭。仍在处理请求。关闭受到阻止，直到完成此事件。

构造函数将 `IApplicationLifetime` 服务注入到任何类中。示例应用将构造函数注入到 `LifetimeEventsHostedService` 类（一个 `IHostedService` 实现）中，用于注册事件。

LifetimeEventsHostedService.cs：

```
internal class LifetimeEventsHostedService : IHostedService
{
    private readonly ILogger _logger;
    private readonly IApplicationLifetime _appLifetime;

    public LifetimeEventsHostedService(
        ILogger<LifetimeEventsHostedService> logger, IApplicationLifetime appLifetime)
    {
        _logger = logger;
        _appLifetime = appLifetime;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _appLifetime.ApplicationStarted.Register(OnStarted);
        _appLifetime.ApplicationStopping.Register(OnStopping);
        _appLifetime.ApplicationStopped.Register(OnStopped);

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        return Task.CompletedTask;
    }

    private void OnStarted()
    {
        _logger.LogInformation("OnStarted has been called.");

        // Perform post-startup activities here
    }

    private void OnStopping()
    {
        _logger.LogInformation("OnStopping has been called.");

        // Perform on-stopping activities here
    }

    private void OnStopped()
    {
        _logger.LogInformation("OnStopped has been called.");

        // Perform post-stopped activities here
    }
}
```

StopApplication 请求终止应用。以下类在调用类的 `Shutdown` 方法时使用 `StopApplication` 正常关闭应用：

```
public class MyClass
{
    private readonly IApplicationLifetime _appLifetime;

    public MyClass(IApplicationLifetime appLifetime)
    {
        _appLifetime = appLifetime;
    }

    public void Shutdown()
    {
        _appLifetime.StopApplication();
    }
}
```

其他资源

- 在 ASP.NET Core 中使用托管服务实现后台任务
- GitHub 上托管的存储库示例

ASP.NET Core 中的 Web 服务器实现

2019/1/14 • • Edit Online

作者：Tom Dykstra、Steve Smith、Stephen Halter 和 Chris Ross

ASP.NET Core 应用与进程内 HTTP 服务器实现一起运行。该服务器实现侦听 HTTP 请求，并以组成 HttpContext 的请求功能集形式，将它们呈现给应用。

- Windows
- macOS
- Linux

ASP.NET Core 随附以下组件：

- Kestrel 服务器是默认跨平台 HTTP 服务器实现。
- IIS HTTP 服务器是 IIS 的进程内服务器。
- HTTP.sys 服务器是仅用于 Windows 的 HTTP 服务器，它基于 HTTP.sys 核心驱动程序和 HTTP 服务器 API。

使用 IIS 或 IIS Express 时，应用会在以下其中一个进程中运行：

- 在与 IIS 工作进程（进程内托管模型）和 IIS HTTP 服务器相同的进程中。“进程内”建议的配置。
- 在独立于 IIS 工作进程（进程外托管模型）和 Kestrel 服务器的进程中。

ASP.NET Core 模块是本机 IIS 模块，用于处理 IIS 和进程内 IIS HTTP 服务器或 Kestrel 之间的本机 IIS 请求。有关更多信息，请参见ASP.NET Core 模块。

托管模型

进程内托管模型

使用进程内托管，ASP.NET Core 在与其 IIS 工作进程相同的进程中运行。这样可消除通过环回适配器代理请求时的进程外性能损失，环回适配器是一个网络接口，用于将传出的网络流量返回给同一计算机。IIS 使用 Windows 进程激活服务 (WAS) 处理进程管理。

ASP.NET Core 模块：

- 执行应用初始化。
  - 加载 CoreCLR。
  - 调用 `Program.Main`。
- 处理 IIS 本机请求的生存期。

定目标到 .NET Framework 的 ASP.NET Core 应用不支持进程内托管模型。

下图说明了 IIS、ASP.NET Core 模块和进程内托管的应用之间的关系：

请求从 Web 到达内核模式 HTTP.sys 驱动程序。驱动程序将本机请求路由到网站的配置端口上的 IIS，通常为 80 (HTTP) 或 443 (HTTPS)。该模块接收本机请求，并将它传递给 IIS HTTP 服务器（`IISHttpServer`）。IIS HTTP 服务器是将请求从本机转换为托管的 IIS 进程内服务器实现。

IIS HTTP 服务器处理请求之后，请求会被推送到 ASP.NET Core 中间件管道中。中间件管道处理该请求并将其作为 `HttpContext` 实例传递给应用的逻辑。应用的响应传递回 IIS，IIS 将响应推送回发起请求的客户端。

进程内托管选择使用现有应用，但 dotnet new 模板默认使用所有 IIS 和 IIS Express 方案的进程内托管模型。

进程外托管模型

由于 ASP.NET Core 应用在独立于 IIS 工作进程的进程中运行，因此该模块会处理进程管理。该模块在第一个请求到达时启动 ASP.NET Core 应用的进程，并在应用关闭或崩溃时重新启动该应用。这基本上与在 Windows 进程激活服务 (WAS) 托管的进程内运行的应用中出现的行为相同。

下图说明了 IIS、ASP.NET Core 模块和进程外托管的应用之间的关系：

请求从 Web 到达内核模式 HTTP.sys 驱动程序。驱动程序将请求路由到网站的配置端口上的 IIS，通常为 80 (HTTP) 或 443 (HTTPS)。该模块将该请求转发到应用的随机端口（非端口 80/443）上的 Kestrel。

该模块在启动时通过环境变量指定端口，IIS 集成中间件将服务器配置为侦听 `http://localhost:{PORT}`。执行其他检查，拒绝不是来自该模块的请求。该模块不支持 HTTPS 转发，因此即使请求由 IIS 通过 HTTPS 接收，它们还是通过 HTTP 转发。

Kestrel 从模块获取请求后，请求会被推送到 ASP.NET Core 中间件管道中。中间件管道处理该请求并将其作为 `HttpContext` 实例传递给应用的逻辑。IIS 集成添加的中间件会将方案、远程 IP 和 pathbase 更新到帐户以将请求转发到 Kestrel。应用的响应传递回 IIS，IIS 将响应推送回发起请求的 HTTP 客户端。

有关 IIS 和 ASP.NET Core 模块的配置指南，请参阅以下主题：

- 使用 IIS 在 Windows 上托管 ASP.NET Core
- ASP.NET Core 模块

- Windows
- macOS
- Linux

ASP.NET Core 随附以下组件：

- Kestrel 服务器是默认跨平台 HTTP 服务器。
- HTTP.sys 服务器是仅用于 Windows 的 HTTP 服务器，它基于 HTTP.sys 核心驱动程序和 HTTP 服务器 API。

使用 IIS 或 IIS Express 时，应用在独立于 IIS 工作进程（进程外）和 Kestrel 服务器的进程中运行。

由于 ASP.NET Core 应用在独立于 IIS 工作进程的进程中运行，因此该模块会处理进程管理。该模块在第一个请求到达时启动 ASP.NET Core 应用的进程，并在应用关闭或崩溃时重新启动该应用。这基本上与在 Windows 进程激活服务 (WAS) 托管的进程内运行的应用中出现的行为相同。

下图说明了 IIS、ASP.NET Core 模块和进程外托管的应用之间的关系：

请求从 Web 到达内核模式 HTTP.sys 驱动程序。驱动程序将请求路由到网站的配置端口上的 IIS，通常为 80 (HTTP) 或 443 (HTTPS)。该模块将该请求转发到应用的随机端口（非端口 80/443）上的 Kestrel。

该模块在启动时通过环境变量指定端口，IIS 集成中间件将服务器配置为侦听 `http://localhost:{port}`。执行其他检查，拒绝不是来自该模块的请求。该模块不支持 HTTPS 转发，因此即使请求由 IIS 通过 HTTPS 接收，它们还是通过 HTTP 转发。

Kestrel 从模块获取请求后，请求会被推送到 ASP.NET Core 中间件管道中。中间件管道处理该请求并将其作为 `HttpContext` 实例传递给应用的逻辑。IIS 集成添加的中间件会将方案、远程 IP 和 pathbase 更新到帐户以将请求转发到 Kestrel。应用的响应传递回 IIS，IIS 将响应推送回发起请求的 HTTP 客户端。

有关 IIS 和 ASP.NET Core 模块的配置指南，请参阅以下主题：

- 使用 IIS 在 Windows 上托管 ASP.NET Core
- ASP.NET Core 模块

Kestrel

Kestrel 是 ASP.NET Core 项目模板中包括的默认 Web 服务器。

Kestrel 的使用方式如下：

- 本身作为边缘服务器，处理直接来自网络（包括 Internet）的请求。

- 与反向代理服务器（如 Internet Information Services (IIS)、Nginx 或 Apache）结合使用。反向代理服务器接收来自 Internet 的 HTTP 请求，并将这些请求转发到 Kestrel。

使用或不使用反向代理服务器对 ASP.NET Core 2.1 或更高版本的应用来说都是受支持的托管配置。

如果应用仅接受来自内部网络的请求，则可单独使用 Kestrel。

如果应用已公开到 Internet，Kestrel 必须使用反向代理服务器，如 Internet Information Services (IIS)、Nginx 或 Apache。反向代理服务器接收来自 Internet 的 HTTP 请求，并将这些请求转发到 Kestrel。

为公共边缘服务器部署（直接公开到 Internet）使用反向代理的最重要的原因是出于安全考虑。Kestrel 的 1.x 版本不包含防御 Internet 攻击的重要安全功能。这包括但不限于相应的超时、请求大小限制和并发连接限制。

有关 Kestrel 配置指南和何时在反向代理配置中使用 Kestrel 的信息，请参阅 ASP.NET Core 中的 Kestrel Web 服务器实现。

Nginx 与 Kestrel

若要了解如何在 Linux 上使用 Nginx 作为 Kestrel 的反向代理服务器，请参阅 使用 Nginx 在 Linux 上托管 ASP.NET Core。

Apache 与 Kestrel

若要了解如何在 Linux 上使用 Apache 作为 Kestrel 的反向代理服务器，请参阅 使用 Apache 在 Linux 上托管 ASP.NET Core。

IIS HTTP 服务器

IIS HTTP 服务器是 IIS 的进程内服务器且为进程内部署所必需。ASP.NET Core 模块用于处理 IIS 和 IIS HTTP 服务器之间的本机 IIS 请求。有关更多信息，请参见ASP.NET Core 模块。

HTTP.sys

如果 ASP.NET Core 应用在 Windows 上运行，则 HTTP.sys 是 Kestrel 的替代选项。为了获得最佳性能，通常建议使用 Kestrel。在应用向 Internet 公开且所需功能受 HTTP.sys（而不是 Kestrel）支持的方案中，可以使用 HTTP.sys。有关更多信息，请参见ASP.NET Core 中的 HTTP.sys Web 服务器实现。

对于仅向内部网络公开的应用，HTTP.sys 同样适用。

有关 HTTP.sys 的配置指南，请参阅 ASP.NET Core 中的 HTTP.sys Web 服务器实现。

ASP.NET Core 服务器基础结构

`Startup.Configure` 方法中提供的 IApplicationBuilder 公开了类型 IFeatureCollection 的 ServerFeatures 属性。Kestrel 和 HTTP.sys 各自仅公开单个功能，即 IServerAddressesFeature，但是不同的服务器实现可能公开其他功能。

`IServerAddressesFeature` 可用于查找服务器实现在运行时绑定的端口。

自定义服务器

如果内置服务器无法满足应用需求，可以创建一个自定义服务器实现。.NET 的开放 Web 接口 (OWIN) 指南 演示了如何编写基于 Nowin 的 IServer 实现。只有应用使用的功能接口需要实现，但至少必须支持 IHttpRequestFeature 和 IHttpResponseFeature。

服务器启动

集成开发环境 (IDE) 或编辑器启动以下应用时，会启动服务器：

- Visual Studio – 可使用启动配置文件通过 IIS Express/ASP.NET Core 模块或控制台来启动应用和服务器。
- Visual Studio Code – 由Omnisharp 通过激活 CoreCLR 调试程序来启动应用和服务器。
- Visual Studio for Mac – 由 Mono Soft-Mode Debugger 启动应用和服务器。

从项目文件夹中的命令提示符启动应用时，dotnet run 会启动该应用和服务器（仅 Kestrel 和 HTTP.sys）。可通过 `-c|--configuration` 选项指定此配置，该选项设置为 `Debug`（默认值）或 `Release`。如果启动配置文件位于 launchSettings.json 文件中，请使用 `--launch-profile <NAME>` 选项设置启动配置文件（例如 `Development` 或 `Production`）。有关详细信息，请参阅 dotnet run 和 .NET Core 分发打包。

HTTP/2 支持

以下部署方案中的 ASP.NET Core 支持 HTTP/2：

- Kestrel
  - 操作系统
    - Windows Server 2016/Windows 10 或更高版本†
    - 具有 OpenSSL 1.0.2 或更高版本的 Linux（例如，Ubuntu 16.04 或更高版本）
    - macOS 的未来版本将支持 HTTP/2。
  - 目标框架：.NET Core 2.2 或更高版本
- HTTP.sys
  - Windows Server 2016/Windows 10 或更高版本
  - 目标框架：不适用于 HTTP.sys 部署。
- IIS（进程内）
  - Windows Server 2016/Windows 10 或更高版本；IIS 10 或更高版本
  - 目标框架：.NET Core 2.2 或更高版本
- IIS（进程外）
  - Windows Server 2016/Windows 10 或更高版本；IIS 10 或更高版本
  - 面向公众的边缘服务器连接使用 HTTP/2，但与 Kestrel 的反向代理连接使用 HTTP/1.1。
  - 目标框架：不适用于 IIS 进程外部署。

†Kestrel 在 Windows Server 2012 R2 和 Windows 8.1 上对 HTTP/2 的支持有限。支持受限是因为可在这些操作系统上使用的受支持 TLS 密码套件列表有限。可能需要使用椭圆曲线数字签名算法 (ECDSA) 生成的证书来保护 TLS 连接。

- HTTP.sys
  - Windows Server 2016/Windows 10 或更高版本
  - 目标框架：不适用于 HTTP.sys 部署。
- IIS（进程外）
  - Windows Server 2016/Windows 10 或更高版本；IIS 10 或更高版本
  - 面向公众的边缘服务器连接使用 HTTP/2，但与 Kestrel 的反向代理连接使用 HTTP/1.1。
  - 目标框架：不适用于 IIS 进程外部署。

HTTP/2 连接必须使用应用程序层协议协商 (ALPN) 和 TLS 1.2 或更高版本。有关详细信息，请参阅与服务器部署方案相关的主题。

其他资源

- ASP.NET Core 中的 Kestrel Web 服务器实现
- ASP.NET Core 模块
- 使用 IIS 在 Windows 上托管 ASP.NET Core
- 将 ASP.NET Core 应用部署到 Azure 应用服务
- 使用 Nginx 在 Linux 上托管 ASP.NET Core
- 使用 Apache 在 Linux 上托管 ASP.NET Core
- ASP.NET Core 中的 HTTP.sys Web 服务器实现

启动 HTTP 请求

2018/10/30 • • Edit Online

作者：Glenn CondronRyan Nowak 和 Steve Gordon

可以注册 IHttpClientFactory 并将其用于配置和创建应用中的 HttpClient 实例。这能带来以下好处：

- 提供一个中心位置，用于命名和配置逻辑 `HttpClient` 实例。例如，可以注册 github 客户端，并将它配置为访问 GitHub。可以注册一个默认客户端用于其他用途。
- 通过委托 `HttpClient` 中的处理程序整理出站中间件的概念，并提供适用于基于 Polly 的中间件的扩展来利用概念。
- 管理基础 `HttpClientMessageHandler` 实例的池和生存期，避免在手动管理 `HttpClient` 生存期时出现常见的 DNS 问题。
- （通过 `ILogger`）添加可配置的记录体验，以处理工厂创建的客户端发送的所有请求。

查看或下载示例代码（如何下载）

系统必备

面向.NET Framework 的项目要求安装 Microsoft.Extensions.Http NuGet 包。面向 .NET Core 且引用 Microsoft.AspNetCore.App 元包的项目已经包括 `Microsoft.Extensions.Http` 包。

消耗模式

在应用中可以通过以下多种方式使用 `IHttpClientFactory`：

- 基本用法
- 命名客户端
- 类型化客户端
- 生成的客户端

它们之间不存在严格的优先级。最佳方法取决于应用的约束条件。

基本用法

在 `Startup.ConfigureServices` 方法中，通过在 `IServiceCollection` 上调用 `AddHttpClient` 扩展方法可以注册 `IHttpClientFactory`。

```
services.AddHttpClient();
```

注册后，在可以使用依赖关系注入 (DI) 注入服务的任何位置，代码都能接受 `IHttpClientFactory`。`IHttpClientFactory` 可以用于创建 `HttpClient` 实例：

```
public class BasicUsageModel : PageModel
{
    private readonly IHttpClientFactory _clientFactory;

    public IEnumerable<GitHubBranch> Branches { get; private set; }

    public bool GetBranchesError { get; private set; }

    public BasicUsageModel(IHttpClientFactory clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task OnGet()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
            "https://api.github.com/repos/aspnet/docs/branches");
        request.Headers.Add("Accept", "application/vnd.github.v3+json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        var client = _clientFactory.CreateClient();

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            Branches = await response.Content
                .ReadAsAsync<IEnumerable<GitHubBranch>>();
        }
        else
        {
            GetBranchesError = true;
            Branches = Array.Empty<GitHubBranch>();
        }
    }
}
```

以这种方式使用 `IHttpClientFactory` 非常适合重构现有应用。这不会影响 `HttpClient` 的使用方式。在当前创建 `HttpClient` 实例的位置上，通过调用 CreateClient 替换出现的这些实例。

命名客户端

如果应用需要有许多不同的 `HttpClient` 用法(每种用法的配置都不同)，可以视情况使用命名客户端。可以在 `HttpClient` 中注册时指定命名 `Startup.ConfigureServices` 的配置。

```
services.AddHttpClient("github", c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    // Github API versioning
    c.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    // Github requires a user-agent
    c.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample");
});
```

上面的代码调用 `AddHttpClient`，同时提供名称“github”。此客户端应用了一些默认配置，也就是需要基址和两个标头来使用 GitHub API。

每次调用 `CreateClient` 时，都会创建 `HttpClient` 的新实例，并调用配置操作。

要使用命名客户端，可将字符串参数传递到 `CreateClient`。指定要创建的客户端的名称：

```
public class NamedClientModel : PageModel
{
    private readonly IHttpClientFactory _clientFactory;

    public IEnumerable<GitHubPullRequest> PullRequests { get; private set; }

    public bool GetPullRequestsError { get; private set; }

    public bool HasPullRequests => PullRequests.Any();

    public NamedClientModel(IHttpClientFactory clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task OnGet()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
            "repos/aspnet/docs/pulls");

        var client = _clientFactory.CreateClient("github");

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            PullRequests = await response.Content
                .ReadAsAsync<IEnumerable<GitHubPullRequest>>();
        }
        else
        {
            GetPullRequestsError = true;
            PullRequests = Array.Empty<GitHubPullRequest>();
        }
    }
}
```

在上述代码中，请求不需要指定主机名。可以仅传递路径，因为采用了为客户端配置的基址。

类型化客户端

类型化客户端提供与命名客户端一样的功能，不需要将字符串用作密钥。类型化客户端方法在使用客户端时提供 IntelliSense 和编译器帮助。它们提供单个地址来配置特定 `HttpClient` 并与其进行交互。例如，单个类型化客户端可能用于单个后端终结点，并封装此终结点的所有处理逻辑。另一个优势是它们使用 DI 且可以被注入到应用中需要的位置。

类型化客户端在构造函数中接收 `HttpClient` 参数：

```
public class GitHubService
{
    public HttpClient Client { get; }

    public GitHubService(HttpClient client)
    {
        client.BaseAddress = new Uri("https://api.github.com/");
        // GitHub API versioning
        client.DefaultRequestHeaders.Add("Accept",
            "application/vnd.github.v3+json");
        // GitHub requires a user-agent
        client.DefaultRequestHeaders.Add("User-Agent",
            "HttpClientFactory-Sample");

        Client = client;
    }

    public async Task<IEnumerable<GitHubIssue>> GetAspNetDocsIssues()
    {
        var response = await Client.GetAsync(
            "/repos/aspnet/docs/issues?state=open&sort=created&direction=desc");

        response.EnsureSuccessStatusCode();

        var result = await response.Content
            .ReadAsAsync<IEnumerable<GitHubIssue>>();

        return result;
    }
}
```

在上述代码中，配置转移到了类型化客户端中。`HttpClient` 对象公开为公共属性。可以定义公开 `HttpClient` 功能的特定于 API 的方法。`GetAspNetDocsIssues` 方法从 GitHub 存储库封装查询和分析最新待解决问题所需的代码。

要注册类型化客户端，可在 `Startup.ConfigureServices` 中使用通用的 `AddHttpClient` 扩展方法，指定类型化客户端类：

```
services.AddHttpClient<GitHubService>();
```

使用 DI 将类型客户端注册为暂时客户端。可以直接插入或使用类型化客户端：

```
public class TypedClientModel : PageModel
{
    private readonly GitHubService _gitHubService;

    public IEnumerable<GitHubIssue> LatestIssues { get; private set; }

    public bool HasIssue => LatestIssues.Any();

    public bool GetIssuesError { get; private set; }

    public TypedClientModel(GitHubService gitHubService)
    {
        _gitHubService = gitHubService;
    }

    public async Task OnGet()
    {
        try
        {
            LatestIssues = await _gitHubService.GetAspNetDocsIssues();
        }
        catch(HttpRequestException)
        {
            GetIssuesError = true;
            LatestIssues = Array.Empty<GitHubIssue>();
        }
    }
}
```

根据你的喜好，可以在 `Startup.ConfigureServices` 中注册时指定类型化客户端的配置，而不是在类型化客户端的构造函数中指定：

```
services.AddHttpClient<RepoService>(c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    c.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    c.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample");
});
```

可以将 `HttpClient` 完全封装在类型化客户端中。不是将它公开为属性，而是可以提供公共方法，用于在内部调用 `HttpClient`。

```
public class RepoService
{
    // _httpClient isn't exposed publicly
    private readonly HttpClient _httpClient;

    public RepoService(HttpClient client)
    {
        _httpClient = client;
    }

    public async Task<IEnumerable<string>> GetRepos()
    {
        var response = await _httpClient.GetAsync("aspnet/repos");

        response.EnsureSuccessStatusCode();

        var result = await response.Content
            .ReadAsAsync<IEnumerable<string>>();

        return result;
    }
}
```

在上述代码中，`HttpClient` 存储未私有字段。进行外部调用的所有访问都经由 `GetRepos` 方法。

生成的客户端

`IHttpClientFactory` 可结合其他第三方库(例如 Refit)使用。Refit 是.NET 的 REST 库。它将 REST API 转换为实时接口。`RestService` 动态生成该接口的实现，使用 `HttpClient` 进行外部 HTTP 调用。

定义了接口和答复来代表外部 API 及其响应：

```
public interface IHelloClient
{
    [Get("/helloworld")]
    Task<Reply> GetMessageAsync();
}

public class Reply
{
    public string Message { get; set; }
}
```

可以添加类型化客户端，使用 Refit 生成实现：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient("hello", c =>
    {
        c.BaseAddress = new Uri("http://localhost:5000");
    })
    .AddTypedClient(c => Refit.RestService.For<IHelloClient>(c));

    services.AddMvc();
}
```

可以在必要时使用定义的接口，以及由 DI 和 Refit 提供的实现：

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHelloClient _client;

    public ValuesController(IHelloClient client)
    {
        _client = client;
    }

    [HttpGet("/")]
    public async Task<ActionResult<Reply>> Index()
    {
        return await _client.GetMessageAsync();
    }
}
```

出站请求中间件

`HttpClient` 已经具有委托处理程序的概念，这些委托处理程序可以链接在一起，处理出站 HTTP 请求。`IHttpClientFactory` 可以轻松定义处理程序并应用于每个命名客户端。它支持注册和链接多个处理程序，以生成出站请求中间件管道。每个处理程序都可以在出站请求前后执行工作。此模式类似于 ASP.NET Core 中的入站中间件管道。此模式提供了一种用于管理围绕 HTTP 请求的横切关注点的机制，包括缓存、错误处理、序列化以及日志记录。

要创建处理程序，请定义一个派生自 `DelegatingHandler` 的类。重写 `SendAsync` 方法，在将请求传递至管道中的下一个处理程序之前执行代码：

```
public class ValidateHeaderHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        if (!request.Headers.Contains("X-API-KEY"))
        {
            return new HttpResponseMessage(HttpStatusCode.BadRequest)
            {
                Content = new StringContent(
                    "You must supply an API key header called X-API-KEY")
            };
        }

        return await base.SendAsync(request, cancellationToken);
    }
}
```

上述代码定义了基本处理程序。它检查请求中是否包含 `X-API-KEY` 头。如果标头缺失，它可以避免 HTTP 调用，并返回合适的响应。

在注册期间可将一个或多个标头添加到 `HttpClient` 的配置。此任务通过 IHttpClientBuilder 的扩展方法完成。

```
services.AddTransient<ValidateHeaderHandler>();

services.AddHttpClient("externalservice", c =>
{
    // Assume this is an "external" service which requires an API KEY
    c.BaseAddress = new Uri("https://localhost:5000/");
})
.AddHttpMessageHandler<ValidateHeaderHandler>();
```

在上述代码中通过 DI 注册了 `ValidateHeaderHandler`。处理程序必须在 DI 中注册为临时处理程序。注册后，即可调用 AddHttpMessageHandler，同时传入头类型。

可以按处理程序应该执行的顺序注册多个处理程序。每个处理程序都会覆盖下一个处理程序，直到最终 `HttpClientHandler` 执行请求：

```
services.AddTransient<SecureRequestHandler>();
services.AddTransient<RequestDataHandler>();

services.AddHttpClient("clientwithhandlers")
    // This handler is on the outside and called first during the
    // request, last during the response.
    .AddHttpMessageHandler<SecureRequestHandler>()
    // This handler is on the inside, closest to the request being
    // sent.
    .AddHttpMessageHandler<RequestDataHandler>();
```

使用基于 Polly 的处理程序

`IHttpClientFactory` 与一个名为 Polly 的热门第三方库集成。Polly 是适用于 .NET 的全面恢复和临时故障处理库。开发人员通过它可以表达策略，例如以流畅且线程安全的方式处理重试、断路器、超时、Bulkhead 隔离和回退。

提供了扩展方法，以实现将 Polly 策略用于配置的 `HttpClient` 实例。Microsoft.Extensions.Http.Polly NuGet 包中提供 Polly 扩展。Microsoft.AspNetCore.App 元包中不包括此包。若要使用扩展，项目中应该包括显式 `<PackageReference />`。

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.Extensions.Http.Polly" Version="2.1.1" />
  </ItemGroup>

</Project>
```

还原此包后，可以使用扩展方法来支持将基于 Polly 的处理程序添加至客户端。

处理临时故障

大多数常见错误在暂时执行外部 HTTP 调用时发生。包含了一种简便的扩展方法，该方法名为 `AddTransientHttpErrorPolicy`，允许定义策略来处理临时故障。使用这种扩展方法配置的策略可以处理 `HttpRequestException`、HTTP 5xx 响应以及 HTTP 408 响应。

`AddTransientHttpErrorPolicy` 扩展可在 `Startup.ConfigureServices` 内使用。该扩展可以提供 `PolicyBuilder` 对象的访问权限，该对象配置为处理表示可能的临时故障的错误：

```
services.AddHttpClient<UnreliableEndpointCallerService>()
    .AddTransientHttpErrorPolicy(p =>
        p.WaitAndRetryAsync(3, _ => TimeSpan.FromMilliseconds(600)));
```

上述代码中定义了 `WaitAndRetryAsync` 策略。请求失败后最多可以重试三次，每次尝试间隔 600 ms。

动态选择策略

存在其他扩展方法，可以用于添加基于 Polly 的处理程序。这类扩展的其中一个是 `AddPolicyHandler`，它具备多个重载。一个重载允许在定义要应用的策略时检查该请求：

```
var timeout = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(10));
var longTimeout = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(30));

services.AddHttpClient("conditionalpolicy")
// Run some code to select a policy based on the request
    .AddPolicyHandler(request =>
        request.Method == HttpMethod.Get ? timeout : longTimeout);
```

在上述代码中，如果出站请求为 GET，则应用 10 秒超时。其他所有 HTTP 方法应用 30 秒超时。

添加多个 Polly 处理程序

嵌套 Polly 策略以增强功能是很常见的：

```
services.AddHttpClient("multiplepolicies")
    .AddTransientHttpErrorPolicy(p => p.RetryAsync(3))
    .AddTransientHttpErrorPolicy(
        p => p.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)));
```

在上述示例中，添加两个处理程序。第一个使用 `AddTransientHttpErrorPolicy` 扩展添加重试策略。若请求失败，最多可重试三次。第一个调用 `AddTransientHttpErrorPolicy` 添加断路器策略。如果尝试连续失败了五次，则会阻止后续外部请求 30 秒。断路器策略处于监控状态。通过此客户端进行的所有调用都共享同样的线路状态。

从 Polly 注册表添加策略

管理常用策略的一种方法是一次性定义它们并使用 `PolicyRegistry` 注册它们。提供了一种扩展方法，可以使用注册表中的策略添加处理程序：

```
var registry = services.AddPolicyRegistry();

registry.Add("regular", timeout);
registry.Add("long", longTimeout);

services.AddHttpClient("regulartimeouthandler")
    .AddPolicyHandlerFromRegistry("regular");
```

在上面的代码中，两个策略在 `PolicyRegistry` 添加到 `ServiceCollection` 中时进行注册。若要使用注册表中的策略，请使用 `AddPolicyHandlerFromRegistry` 方法，同时传递要应用的策略的名称。

要进一步了解 `IHttpClientFactory` 和 Polly 集成，请参考 Polly Wiki。

HttpClient 和生存期管理

每次对 `IHttpClientFactory` 调用 `CreateClient` 都会返回一个新 `HttpClient` 实例。每个命名客户端都有一个 HttpMessageHandler。`IHttpClientFactory` 将工厂创建的 `HttpMessageHandler` 实例汇集到池中，以减少资源消

耗。新建 `HttpClient` 实例时，可能会重用池中的 `HttpMessageHandler` 实例（如果生存期尚未到期的话）。

由于每个处理程序通常管理自己的基础 HTTP 连接，因此需要池化处理程序。创建超出必要数量的处理程序可能会导致连接延迟。部分处理程序还保持连接无期限地打开，这样可以防止处理程序对 DNS 更改作出反应。

处理程序的默认生存期为两分钟。可在每个命名客户端上重写默认值。若要替代值，请对创建客户端时返回的 `IHttpClientBuilder` 调用 SetHandlerLifetime：

```
services.AddHttpClient("extendedhandlerlifetime")
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

无需处置客户端。处置既取消传出请求，又保证在调用 Dispose 后无法使用给定的 `HttpClient` 实例。`IHttpClientFactory` 跟踪和处置 `HttpClient` 实例使用的资源。`HttpClient` 实例通常可视为无需处置的 .NET 对象。

保持各个 `HttpClient` 实例长时间处于活动状态是在 `IHttpClientFactory` 推出前使用的常见模式。迁移到 `IHttpClientFactory` 后，就无需再使用此模式。

日志记录

通过 `IHttpClientFactory` 创建的客户端记录所有请求的日志消息。在日志记录配置中启用合适的信息级别可以查看默认日志消息。仅在跟踪级别包含附加日志记录（例如请求标头的日志记录）。

用于每个客户端的日志类别包含客户端名称。例如，名为“MyNamedClient”的客户端使用 `System.Net.Http.HttpClient.MyNamedClient.LogicalHandler` 类别来记录消息。后缀为 LogicalHandler 的消息在请求处理程序管道外部发生。在请求时，在管道中的任何其他处理程序处理请求之前记录消息。在响应时，在任何其他管道处理程序接收响应之后记录消息。

日志记录还在请求处理程序管道内部发生。在“MyNamedClient”示例中，这些消息是针对日志类别 `System.Net.Http.HttpClient.MyNamedClient.ClientHandler` 进行记录。在请求时，在所有其他处理程序运行后，以及刚好在通过网络发出请求之前记录消息。在响应时，此日志记录包含响应在通过处理程序管道被传递回去之前的状态。

在管道内外启用日志记录，可以检查其他管道处理程序做出的更改。例如，其中可能包含对请求标头的更改，或者对响应状态代码的更改。

通过在日志类别中包含客户端名称，可以在必要时对特定的命名客户端筛选日志。

配置 HttpMessageHandler

控制客户端使用的内部 `HttpMessageHandler` 的配置是有必要的。

在添加命名客户端或类型化客户端时，会返回 `IHttpClientBuilder`。ConfigurePrimaryHttpMessageHandler 扩展方法可用于定义委托。委托用于创建和配置客户端使用的主要 `HttpMessageHandler`：

```
services.AddHttpClient("configured-inner-handler")
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        return new HttpClientHandler()
        {
            AllowAutoRedirect = false,
            UseDefaultCredentials = true
        };
    });
```

ASP.NET Core 中的 Razor 页面介绍

2018/12/26 • • Edit Online

作者：Rick Anderson 和 Ryan Nowak

Razor 页面是 ASP.NET Core MVC 的一个新特性，它可以使基于页面的编码方式更简单高效。

若要查找使用模型视图控制器方法的教程，请参阅 ASP.NET Core MVC 入门。

本文档介绍 Razor 页面。它并不是分步教程。如果认为某些部分过于复杂，请参阅 Razor 页面入门。有关 ASP.NET Core 的概述，请参阅 ASP.NET Core 简介。

系统必备

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

创建 Razor Pages 项目

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code

请参阅 Razor Pages 入门，获取关于如何创建 Razor Pages 项目的详细说明。

Razor 页面

Startup.cs 中已启用 Razor 页面：

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Includes support for Razor Pages and controllers.
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvc();
    }
}
```

请考虑一个基本页面：

```
@page

<h1>Hello, world!</h1>
<h2>The time on the server is @DateTime.Now</h2>
```

上述代码看上去类似于一个 Razor 视图文件。不同之处在于 `@page` 指令。`@page` 使文件转换为一个 MVC 操作，这意味着它将直接处理请求，而无需通过控制器处理。`@page` 必须是页面上的第一个 Razor 指令。`@page` 将影响其他 Razor 构造的行为。

将在以下两个文件中显示使用 `PageModel` 类的类似页面。Pages/Index2.cshtml 文件：

```
@page
@using RazorPagesIntro.Pages
@model IndexModel2

<h2>Separate page model</h2>
<p>
    @Model.Message
</p>
```

Pages/Index2.cshtml.cs 页面模型：

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using System;

namespace RazorPagesIntro.Pages
{
    public class IndexModel2 : PageModel
    {
        public string Message { get; private set; } = "PageModel in C#";

        public void OnGet()
        {
            Message += $" Server time is { DateTime.Now }";
        }
    }
}
```

按照惯例，`PageModel` 类文件的名称与追加 .cs 的 Razor 页面文件名称相同。例如，前面的 Razor 页面的名称为 Pages/Index2.cshtml。包含 `PageModel` 类的文件的名称为 Pages/Index2.cshtml.cs。

页面的 URL 路径的关联由页面在文件系统中的位置决定。下表显示了 Razor 页面路径及匹配的 URL：

文件名和路径	匹配的 URL
/Pages/Index.cshtml	`/` 或 `/Index`
/Pages/Contact.cshtml	`/Contact`
/Pages/Store/Contact.cshtml	`/Store/Contact`
/Pages/Store/Index.cshtml	`/Store` 或 `/Store/Index`

注意：

- 默认情况下，运行时在“Pages”文件夹中查找 Razor 页面文件。
- URL 未包含页面时，`Index` 为默认页面。

编写基本窗体

由于 Razor 页面的设计，在构建应用时可轻松实施用于 Web 浏览器的常用模式。模型绑定、标记帮助程序和 HTML 帮助程序均只可用于 Razor 页面类中定义的属性。请参考为 `Contact` 模型实现基本的“联系我们”窗体的页面：

在本文档中的示例中，`DbContext` 在 Startup.cs 文件中进行初始化。

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesContacts.Data;

namespace RazorPagesContacts
{
    public class Startup
    {
        public IHostingEnvironment HostingEnvironment { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(options =>
                                options.UseInMemoryDatabase("name"));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseMvc();
        }
    }
}
```

数据模型：

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Data
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(100)]
        public string Name { get; set; }
    }
}
```

数据库上下文：

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions options)
            : base(options)
        {
        }

        public DbSet<Customer> Customers { get; set; }
    }
}
```

Pages/Create.cshtml 视图文件：

```
@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

Pages/Create.cshtml.cs 页面模型：

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class CreateModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }
    }
}
```

按照惯例，`PageModel` 类命名为 `<PageName>Model` 并且它与页面位于同一个命名空间中。

使用 `PageModel` 类，可以将页面的逻辑与其展示分离开来。它定义了页面处理程序，用于处理发送到页面的请求和用于呈现页面的数据。借助这种分离，可以通过依赖关系注入管理页面依赖关系，并对页面执行单元测试。

页面包含 `OnPostAsync` 处理程序方法，它在 `POST` 请求上运行（当用户发布窗体时）。可以为任何 HTTP 谓词添加处理程序方法。最常见的处理程序是：

- `OnGet`，用于初始化页面所需的状态。OnGet 示例。
- `OnPost`，用于处理窗体提交。

`Async` 命名后缀为可选，但是按照惯例通常会将它用于异步函数。前面示例中的 `OnPostAsync` 代码看上去与通常在控制器中编写的内容相似。前面的代码通常用于 Razor 页面。多数 MVC 基元（例如模型绑定、验证和操作结果）都是共享的。

之前的 `OnPostAsync` 方法：

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    return RedirectToPage("/Index");
}
```

`OnPostAsync` 的基本流：

检查验证错误。

- 如果没有错误，则保存数据并重定向。
- 如果有错误，则再次显示页面并附带验证消息。客户端验证与传统的 ASP.NET Core MVC 应用程序相同。很多情况下，都会在客户端上检测到验证错误，并且从不将它们提交到服务器。

成功输入数据后，`OnPostAsync` 处理程序方法调用 `RedirectToPage` 帮助程序方法来返回 `RedirectToPageResult` 的实例。`RedirectToPage` 是新的操作结果，类似于 `RedirectToAction` 或 `RedirectToRoute`，但是已针对页面进行自定义。在前面的示例中，它将重定向到根索引页（`/Index`）。页面 URL 生成部分中详细介绍了 `RedirectToPage`。

提交的窗体存在（已传递到服务器的）验证错误时，`OnPostAsync` 处理程序方法调用 `Page` 帮助程序方法。`Page` 返回 `PageResult` 的实例。返回 `Page` 的过程与控制器中的操作返回 `View` 的过程相似。`PageResult` 是处理程序方法的默认 返回类型。返回 `void` 的处理程序方法将显示页面。

`Customer` 属性使用 `[BindProperty]` 特性来选择加入模型绑定。

```
public class CreateModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateModel(AppDbContext db)
    {
        _db = db;
    }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        return RedirectToPage("/Index");
    }
}
```

默认情况下，Razor 页面只绑定带有非 GET 谓词的属性。绑定属性可以减少需要编写的代码量。绑定通过使用相同的属性显示窗体字段（`<input asp-for="Customer.Name" />`）来减少代码，并接受输入。

> **WARNING**
>
> 出于安全原因，必须选择绑定 `GET` 请求数据以对模型属性进行分页。请在将用户输入映射到属性前对其进行验证。当处理依赖查询字符串或路由值的方案时，选择加入 `GET` 绑定非常有用。
>
> 要将属性绑定在 `GET` 请求上，请将 [BindProperty] 特性的 `SupportsGet` 属性设置为 `true`：
> `[BindProperty(SupportsGet = true)]`

主页 (Index.cshtml)：

```
@page
@model RazorPagesContacts.Pages.IndexModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h1>Contacts</h1>
<form method="post">
    <table class="table">
        <thead>
            <tr>
                <th>ID</th>
                <th>Name</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var contact in Model.Customers)
            {
                <tr>
                    <td>@contact.Id</td>
                    <td>@contact.Name</td>
                    <td>
                        <a asp-page="./Edit" asp-route-id="@contact.Id">edit</a>
                        <button type="submit" asp-page-handler="delete"
                                asp-route-id="@contact.Id">delete</button>
                    </td>
                </tr>
            }
        </tbody>
    </table>

    <a asp-page="./Create">Create</a>
</form>
```

关联的 `PageModel` 类 (Index.cshtml.cs)：

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Pages
{
    public class IndexModel : PageModel
    {
        private readonly AppDbContext _db;

        public IndexModel(AppDbContext db)
        {
            _db = db;
        }

        public IList<Customer> Customers { get; private set; }

        public async Task OnGetAsync()
        {
            Customers = await _db.Customers.AsNoTracking().ToListAsync();
        }

        public async Task<IActionResult> OnPostDeleteAsync(int id)
        {
            var contact = await _db.Customers.FindAsync(id);

            if (contact != null)
            {
                _db.Customers.Remove(contact);
                await _db.SaveChangesAsync();
            }

            return RedirectToPage();
        }
    }
}
```

Index.cshtml 文件包含以下标记来创建每个联系人项的编辑链接：

```
<a asp-page="./Edit" asp-route-id="@contact.Id">edit</a>
```

定位点标记帮助程序 使用 `asp-route-{value}` 属性生成"编辑"页面的链接。此链接包含路由数据及联系人 ID。例如 `http://localhost:5000/Edit/1`。

Pages/Edit.cshtml 文件：

```
@page "{id:int}"
@model RazorPagesContacts.Pages.EditModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

@{
    ViewData["Title"] = "Edit Customer";
}

<h1>Edit Customer - @Model.Customer.Id</h1>
<form method="post">
    <div asp-validation-summary="All"></div>
    <input asp-for="Customer.Id" type="hidden" />
    <div>
        <label asp-for="Customer.Name"></label>
        <div>
            <input asp-for="Customer.Name" />
            <span asp-validation-for="Customer.Name" ></span>
        </div>
    </div>

    <div>
        <button type="submit">Save</button>
    </div>
</form>
```

第一行包含 `@page "{id:int}"` 指令。路由约束 `"{id:int}"` 告诉页面接受包含 `int` 路由数据的页面请求。如果页面请求未包含可转换为 `int` 的路由数据，则运行时返回 HTTP 404（未找到）错误。若要使 ID 可选，请将 `?` 追加到路由约束：

```
@page "{id:int?}"
```

Pages/Edit.cshtml.cs 文件：

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly AppDbContext _db;

        public EditModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Customer = await _db.Customers.FindAsync(id);

            if (Customer == null)
            {
                return RedirectToPage("/Index");
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Attach(Customer).State = EntityState.Modified;

            try
            {
                await _db.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                throw new Exception($"Customer {Customer.Id} not found!");
            }

            return RedirectToPage("/Index");
        }
    }
}
```

Index.cshtml 文件还包含用于为每个客户联系人创建删除按钮的标记：

```
<button type="submit" asp-page-handler="delete"
        asp-route-id="@contact.Id">delete</button>
```

删除按钮采用 HTML 呈现，其 `formaction` 包括参数：

- `asp-route-id` 属性指定的客户联系人 ID。

- `asp-page-handler` 属性指定的 `handler`。

下面是呈现的删除按钮的示例，其中客户联系人 ID 为 `1`：

```
<button type="submit" formaction="/?id=1&amp;handler=delete">delete</button>
```

选中按钮时，向服务器发送窗体 `POST` 请求。按照惯例，根据方案 `OnPost[handler]Async` 基于 `handler` 参数的值来选择处理程序方法的名称。

因为本示例中 `handler` 是 `delete`，因此 `OnPostDeleteAsync` 处理程序方法用于处理 `POST` 请求。如果 `asp-page-handler` 设置为不同值（如 `remove`），则选择名称为 `OnPostRemoveAsync` 的页面处理程序方法。

```
public async Task<IActionResult> OnPostDeleteAsync(int id)
{
    var contact = await _db.Customers.FindAsync(id);

    if (contact != null)
    {
        _db.Customers.Remove(contact);
        await _db.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

`OnPostDeleteAsync` 方法：

- 接受来自查询字符串的 `id`。
- 使用 `FindAsync` 查询客户联系人的数据库。
- 如果找到客户联系人，则从客户联系人列表将其删除。数据库将更新。
- 调用 `RedirectToPage`，重定向到根索引页（`/Index`）。

按需标记页面属性

`PageModel` 上的属性可通过 Required 特性进行修饰：

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        [Required(ErrorMessage = "Color is required")]
        public string Color { get; set; }

        public IActionResult OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            // Process color.

            return RedirectToPage("./Index");
        }
    }
}
```

有关详细信息，请参阅模型验证。

使用 OnGet 处理程序管理 HEAD 请求

HEAD 请求可以检索特定资源的标头。与 GET 请求不同，HEAD 请求不返回响应正文。

通常，针对 HEAD 请求创建和调用 HEAD 处理程序：

```
public void OnHead()
{
    HttpContext.Response.Headers.Add("HandledBy", "Handled by OnHead!");
}
```

如果未定义 HEAD 处理程序（`OnHead`），Razor 页面会回退以调用 ASP.NET Core 2.1 或更高版本中的 GET 页处理程序（`OnGet`）。在 ASP.NET Core 2.1 和 2.2 中，`Startup.Configure` 中的 SetCompatibilityVersion 会发生此行为：

```
services.AddMvc()
    .SetCompatibilityVersion(Microsoft.AspNetCore.Mvc.CompatibilityVersion.Version_2_1);
```

默认模板在 ASP.NET Core 2.1 和 2.2 中生成 `SetCompatibilityVersion` 调用。

`SetCompatibilityVersion` 有效地将 Razor 页面选项 `AllowMappingHeadRequestsToGetHandler` 设置为 `true`。

可以显式地选择使用特定行为，而不是通过 `SetCompatibilityVersion` 选择使用所有 2.1 行为。以下代码选择使用将 HEAD 映射到 GET 处理程序这一行为。

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.AllowMappingHeadRequestsToGetHandler = true;
    });
```

XSRF/CSRF 和 Razor 页面

无需为防伪验证编写任何代码。Razor 页面自动将防伪标记生成过程和验证过程包含在内。

将布局、分区、模板和标记帮助程序用于 Razor 页面

页面可使用 Razor 视图引擎的所有功能。布局、分区、模板、标记帮助程序、_ViewStart.cshtml 和 _ViewImports.cshtml 的工作方式与它们在传统的 Razor 视图中的工作方式相同。

让我们使用其中的一些功能来整理此页面。

向 Pages/Shared/_Layout.cshtml 添加布局页面：

向 Pages/_Layout.cshtml 添加布局页面：

```
<!DOCTYPE html>
<html>
<head>
    <title>Razor Pages Sample</title>
</head>
<body>
   <a asp-page="/Index">Home</a>
    @RenderBody()
    <a asp-page="/Customers/Create">Create</a> <br />
</body>
</html>
```

布局：

- 控制每个页面的布局（页面选择退出布局时除外）。
- 导入 HTML 结构，例如 JavaScript 和样式表。

请参阅布局页面了解详细信息。

在 Pages/_ViewStart.cshtml 中设置 Layout 属性：

```
@{
    Layout = "_Layout";
}
```

布局位于“页面/共享”文件夹中。页面按层次结构从当前页面的文件夹开始查找其他视图（布局、模板、分区）。可以从“页面/共享”文件夹下的任意 Razor 页面使用“页面”文件夹中的布局。

布局文件应位于 Pages/Shared 文件夹中。

布局位于“页面”文件夹中。页面按层次结构从当前页面的文件夹开始查找其他视图（布局、模板、分区）。可以从“页面”文件夹下的任意 Razor 页面使用“页面”文件夹中的布局。

建议不要将布局文件放在“视图/共享”文件夹中。视图/共享 是一种 MVC 视图模式。Razor 页面旨在依赖文件夹层次结构，而非路径约定。

Razor 页面中的视图搜索包含“页面”文件夹。用于 MVC 控制器和传统 Razor 视图的布局、模板和分区可直接

工作。

添加 Pages/_ViewImports.cshtml 文件：

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

本教程的后续部分中将介绍 `@namespace`。`@addTagHelper` 指令将内置标记帮助程序引入"页面"文件夹中的所有页面。

页面上显式使用 `@namespace` 指令后：

```
@page
@namespace RazorPagesIntro.Pages.Customers

@model NameSpaceModel

<h2>Name space</h2>
<p>
    @Model.Message
</p>
```

此指令将为页面设置命名空间。`@model` 指令无需包含命名空间。

_ViewImports.cshtml 中包含 `@namespace` 指令后，指定的命名空间将为在导入 `@namespace` 指令的页面中生成的命名空间提供前缀。生成的命名空间的剩余部分（后缀部分）是包含 _ViewImports.cshtml 的文件夹与包含页面的文件夹之间以点分隔的相对路径。

例如，`PageModel` 类 Pages/Customers/Edit.cshtml.cs 显式设置命名空间：

```
namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly AppDbContext _db;

        public EditModel(AppDbContext db)
        {
            _db = db;
        }

        // Code removed for brevity.
```

Pages/_ViewImports.cshtml 文件设置以下命名空间：

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

为 Pages/Customers/Edit.cshtml Razor 页面生成的命名空间与 `PageModel` 类相同。

`@namespace` 也可用于传统的 Razor 视图。

原始的 Pages/Create.cshtml 视图文件：

```
@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

更新后的 Pages/Create.cshtml 视图文件：

```
@page
@model CreateModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

Razor 页面初学者项目包含 Pages/_ValidationScriptsPartial.cshtml，它与客户端验证联合。

有关分部视图的详细信息，请参阅 ASP.NET Core 中的分部视图。

页面的 URL 生成

之前显示的 `Create` 页面使用 `RedirectToPage`：

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    return RedirectToPage("/Index");
}
```

应用具有以下文件/文件夹结构：

- /Pages
  - Index.cshtml
  - /Customers

- Create.cshtml
- Edit.cshtml
- Index.cshtml

成功后，Pages/Customers/Create.cshtml 和 Pages/Customers/Edit.cshtml 页面将重定向到 Pages/Index.cshtml。字符串 `/Index` 是用于访问上一页的 URI 的组成部分。可以使用字符串 `/Index` 生成 Pages/Index.cshtml 页面的 URI。例如:

- `Url.Page("/Index", ...)`
- `<a asp-page="/Index">My Index Page</a>`
- `RedirectToPage("/Index")`

页面名称是从根"/Pages"文件夹到页面的路径(包含前导 `/`，例如 `/Index`)。与硬编码 URL 相比，前面的 URL 生成示例提供了改进的选项和功能。URL 生成使用路由，并且可以根据目标路径定义路由的方式生成参数并对参数编码。

页面的 URL 生成支持相对名称。下表显示了 Pages/Customers/Create.cshtml 中不同的 `RedirectToPage` 参数选择的索引页:

REDIRECTTOPAGE(X)	页
RedirectToPage("/Index")	*Pages/Index*
RedirectToPage("./Index");	*Pages/Customers/Index*
RedirectToPage("../Index")	*Pages/Index*
RedirectToPage("Index")	*Pages/Customers/Index*

`RedirectToPage("Index")`、`RedirectToPage("./Index")` 和 `RedirectToPage("../Index")` 是相对名称。结合 `RedirectToPage` 参数与当前页的路径来计算目标页面的名称。

构建结构复杂的站点时，相对名称链接很有用。如果使用相对名称链接文件夹中的页面，则可以重命名该文件夹。所有链接仍然有效(因为这些链接未包含此文件夹名称)。

ViewData 特性

可以通过 ViewDataAttribute 将数据传递到页面。控制器或 Razor 页面模型上使用 `[ViewData]` 修饰的属性将其值存储在 ViewDataDictionary 中并从此处进行加载。

在下面的示例中，`AboutModel` 包含使用 `[ViewData]` 修饰的 `Title` 属性。`Title` 属性设置为"关于"页面的标题:

```
public class AboutModel : PageModel
{
    [ViewData]
    public string Title { get; } = "About";

    public void OnGet()
    {
    }
}
```

在"关于"页面中，以模型属性的形式访问 `Title` 属性:

```
<h1>@Model.Title</h1>
```

在布局中，从 ViewData 字典读取标题：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
```

TempData

ASP.NET 在控制器上公开了 TempData 属性。此属性存储未读取的数据。`Keep` 和 `Peek` 方法可用于检查数据，而不执行删除。多个请求需要数据时，`TempData` 有助于进行重定向。

`[TempData]` 是 ASP.NET Core 2.0 中的新属性，在控制器和页面上受支持。

下面的代码使用 `TempData` 设置 `Message` 的值：

```
public class CreateDotModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateDotModel(AppDbContext db)
    {
        _db = db;
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";
        return RedirectToPage("./Index");
    }
}
```

Pages/Customers/Index.cshtml 文件中的以下标记使用 `TempData` 显示 `Message` 的值。

```
<h3>Msg: @Model.Message</h3>
```

Pages/Customers/Index.cshtml.cs 页面模型将 `[TempData]` 属性应用到 `Message` 属性。

```
[TempData]
public string Message { get; set; }
```

有关详细信息，请参阅 TempData。

针对一个页面的多个处理程序

以下页面使用 `asp-page-handler` 标记帮助程序为两个页面处理程序生成标记：

```
@page
@model CreateFATHModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
    </form>
</body>
</html>
```

前面示例中的窗体包含两个提交按钮，每个提交按钮均使用 `FormActionTagHelper` 提交到不同的 URL。`asp-page-handler` 是 `asp-page` 的配套属性。`asp-page-handler` 生成提交到页面定义的各个处理程序方法的 URL。未指定 `asp-page`，因为示例已链接到当前页面。

页面模型：

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateFATHModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateFATHModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostJoinListAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }

        public async Task<IActionResult> OnPostJoinListUCAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }
            Customer.Name = Customer.Name?.ToUpper();
            return await OnPostJoinListAsync();
        }
    }
}
```

前面的代码使用已命名处理程序方法。已命名处理程序方法通过采用名称中 `On<HTTP Verb>` 之后及 `Async` 之前的文本（如果有）创建。在前面的示例中，页面方法是 OnPost**JoinList**Async 和 OnPost**JoinListUC**Async。删除 OnPost 和 Async 后，处理程序名称为 `JoinList` 和 `JoinListUC`。

```
<input type="submit" asp-page-handler="JoinList" value="Join" />
<input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
```

使用前面的代码时，提交到 `OnPostJoinListAsync` 的 URL 路径为 `http://localhost:5000/Customers/CreateFATH?handler=JoinList`。提交到 `OnPostJoinListUCAsync` 的 URL 路径为 `http://localhost:5000/Customers/CreateFATH?handler=JoinListUC`。

自定义路由

使用 `@page` 指令，执行以下操作：

- 指定页面的自定义路由。例如，使用 `@page "/Some/Other/Path"` 将“关于”页面的路由设置为 `/Some/Other/Path`。
- 将段追加到页面的默认路由。例如，可使用 `@page "item"` 将“item”段添加到页面的默认路由。

- 将参数追加到页面的默认路由。例如，`@page "{id}"` 页面需要 ID 参数 `id`。

支持开头处以波形符 (`~`) 指定的相对于根目录的路径。例如，`@page "~/Some/Other/Path"` 和 `@page "/Some/Other/Path"` 相同。

可以通过指定路由模板 `@page "{handler?}"`，将 URL 中的查询字符串 `?handler=JoinList` 更改为路由段 `/JoinList`。

如果你不喜欢 URL 中的查询字符串 `?handler=JoinList`，可以更改路由，将处理程序名称放在 URL 的路径部分。可以通过在 `@page` 指令后面添加使用双引号括起来的路由模板来自定义路由。

```
@page "{handler?}"
@model CreateRouteModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
    </form>
</body>
</html>
```

使用前面的代码时，提交到 `OnPostJoinListAsync` 的 URL 路径为 `http://localhost:5000/Customers/CreateFATH/JoinList`。提交到 `OnPostJoinListUCAsync` 的 URL 路径为 `http://localhost:5000/Customers/CreateFATH/JoinListUC`。

`handler` 前面的 `?` 表示路由参数为可选。

配置和设置

若要配置高级选项，请在 MVC 生成器上使用 `AddRazorPagesOptions` 扩展方法：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
        {
            options.RootDirectory = "/MyPages";
            options.Conventions.AuthorizeFolder("/MyPages/Admin");
        });
}
```

目前，可以使用 `RazorPagesOptions` 设置页面的根目录，或者为页面添加应用程序模型约定。通过这种方式，我们在将来会实现更多扩展功能。

若要预编译视图，请参阅 Razor 视图编译。

下载或查看示例代码.

请参阅 Razor 页面入门，这篇文章以本文为基础编写。

指定 Razor 页面位于内容根目录中

默认情况下，Razor 页面位于 /Pages 目录的根位置。向 AddMvc 添加 WithRazorPagesAtContentRoot，以指定 Razor 页面位于应用的内容根目录 (ContentRootPath) 中：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        ...
    })
    .WithRazorPagesAtContentRoot();
```

指定 Razor 页面位于自定义根目录中

向 AddMvc 添加 WithRazorPagesRoot，以指定 Razor 页面位于应用中自定义根目录位置（提供相对路径）：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        ...
    })
    .WithRazorPagesRoot("/path/to/razor/pages");
```

其他资源

- ASP.NET Core 简介
- ASP.NET Core 的 Razor 语法参考
- 教程：在 ASP.NET Core 中开始使用 Razor Pages
- 在 ASP.NET Core razor 页授权约定
- ASP.NET Core 中 Razor 页面的路由和应用约定
- 在 ASP.NET Core razor 页单元测试
- ASP.NET Core 中的分部视图

教程：使用 ASP.NET Core 创建 Razor 页面 Web 应用

2019/1/3 • • Edit Online

此系列教程介绍了生成 Razor Pages Web 应用的基础知识。本系列包括以下教程：

1. 创建 Razor 页面 Web 应用
2. 向 Razor 页面应用添加模型
3. 基架（生成）Razor 页面
4. 使用数据库
5. 更新 Razor 页面
6. 添加搜索
7. 添加新字段
8. 添加验证

结束时，你将有一个应用来显示和管理电影数据库。

教程：在 ASP.NET Core 中开始使用 Razor Pages

2019/1/10 • • Edit Online

作者：Rick Anderson

这是系列中的第一个教程。本系列介绍构建 ASP.NET Core Razor Pages Web 应用的基础知识。系列结束时，你将有一个应用来管理电影数据库。

View or download sample code (how to download).

在本教程中，你将了解：

- 创建 Razor 页面 Web 应用。
- 运行应用。
- 检查项目文件。

本教程结束时，你将有一个工作 Razor Pages Web 应用，可于后续教程在其基础上进行构建。

主页或索引页

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

创建 Razor 页面 Web 应用

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 从 Visual Studio“文件”菜单中选择“新建” > “项目”。

- 创建新的 ASP.NET Core Web 应用程序。将项目命名为“RazorPagesMovie”。将项目命名为“RazorPagesMovie”，以便命名空间在复制和粘贴代码时相互匹配。

- 在下拉列表中选择“ASP.NET Core 2.2”，然后选择“Web 应用程序”。

创建以下初学者项目：

运行 Web 应用程序

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 按 Ctrl+F5 以在不使用调试程序的情况下运行。

  Visual Studio 启动 IIS Express 并运行应用。地址栏显示 `localhost:port#`，而不是显示 `example.com`。这是因为 `localhost` 是本地计算机的标准主机名。Localhost 仅为来自本地计算机的 Web 请求提供服务。Visual Studio 创建 Web 项目时，Web 服务器使用的是随机端口。在上图中，端口号为 5001。运行应用时，将看到不同的端口号。

- 在应用的主页上，选择"接受"以同意跟踪。

  此应用不会跟踪个人信息，但项目模板包括许可功能，以防需要它来符合欧盟的一般数据保护条例 (GDPR)。

下图展示了同意跟踪后的应用：

检查项目文件

下面是主项目文件夹和文件的概述，将在后续教程中使用。

“页面”文件夹

包含 Razor 页面和支持文件。每个 Razor 页面都是一对文件：

- 一个 .cshtml 文件，其中使用 Razor 语法包含 HTML 标记与 C# 代码。
- 一个 .cshtml.cs 文件，其中包含处理页事件的 C# 代码。

支持文件的名称以下划线开头。例如，_Layout.cshtml 文件可配置所有页面通用的 UI 元素。此文件设置页面顶部的导航菜单和页面底部的版权声明。有关更多信息，请参见ASP.NET Core 中的布局。

wwwroot 文件夹

包含静态文件，如 HTML 文件、JavaScript 文件和 CSS 文件。有关更多信息，请参见ASP.NET Core 中的静态文件。

appSettings.json

包含配置数据，如连接字符串。有关更多信息，请参见ASP.NET Core 中的配置。

Program.cs

包含程序的入口点。有关更多信息，请参见ASP.NET Core Web 主机。

Startup.cs

包含配置应用行为的代码，例如，是否需要同意 cookie。有关更多信息，请参见ASP.NET Core 中的应用启动。

后续步骤

在本教程中，你将了解：

- 创建 Razor Pages Web 应用。
- 运行应用。
- 检查项目文件。

进入系列的下一教程：

添加模型

在 ASP.NET Core 中向 Razor 页面应用添加模型

2019/1/10 • • Edit Online

作者：Rick Anderson

View or download sample code (how to download).

在此部分中，添加了用于管理数据库中的电影的类。可以结合 Entity Framework Core (EF Core) 使用这些类来处理数据库。EF Core 是对象关系映射 (ORM) 框架，可以简化数据访问代码。

模型类称为 POCO 类（源自“简单传统 CLR 对象”），因为它们与 EF Core 没有任何依赖关系。它们定义数据库中存储的数据属性。

查看或下载示例。

添加数据模型

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

右键单击“RazorPagesMovie”项目 >“添加” > “新建文件夹”。将文件夹命名为“Models”。

右键单击“Models”文件夹。选择“添加” > “类”。将类命名“Movie”。

向 `Movie` 类添加以下属性：

```
using System;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

`Movie` 类包含：

- 数据库需要 `ID` 字段以获取主键。
- `[DataType(DataType.Date)]`：DataType 属性指定数据的类型（日期）。通过此特性：
  - 用户无需在数据字段中输入时间信息。
  - 仅显示日期，而非时间信息。

DataAnnotations 会在后续教程中介绍。

生成项目以验证没有任何编译错误。

搭建“电影”模型的基架

在此部分，将搭建“电影”模型的基架。确切地说，基架工具将生成页面，用于对“电影”模型执行创建、读取、更新和删除 (CRUD) 操作。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

创建“Pages/Movies”文件夹：

- 右键单击 Pages 文件夹 >“添加” > “新建文件夹”。
- 将文件夹命名为“Movies”

右键单击 Pages/Movies 文件夹 >“添加” > “新搭建基架的项目”。

在“添加基架”对话框中，选择“使用实体框架生成 Razor Pages (CRUD)” > “添加”。

完成"使用实体框架(CRUD)添加 Razor Pages"对话框：

- 在"模型类"下拉列表中，选择"Movie (RazorPagesMovie.Models)。
- 在"数据上下文类"行中，选择 +(加号)并接受生成的名称"RazorPagesMovie.Models.RazorPagesMovieContext"。
- 选择"添加"。

appsettings.json 文件通过用于连接到本地数据的连接字符串进行更新。

在搭建基架时，会创建并更新以下文件：

创建的文件

- *Pages/Movies*："创建"、"删除"、"详细信息"、"编辑"和"索引"。
- Data/RazorPagesMovieContext.cs

文件已更新

- *Startup.cs*

下一部分中说明了创建和更新的文件。

初始迁移

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

在此部分中，程序包管理器控制台 (PMC) 用于：

- 添加初始迁移。
- 使用初始迁移来更新数据库。

从“工具”菜单中，选择“NuGet 包管理器” > “包管理器控制台”。

在 PMC 中，输入以下命令：

```
Add-Migration Initial
Update-Database
```

`ef migrations add InitialCreate` 命令生成用于创建初始数据库架构的代码。此架构的依据为 `DbContext` 中指定的模型（在 RazorPagesMovieContext.cs 文件中）。`InitialCreate` 参数用于为迁移命名。可以使用任何名称，但是按照惯例，会选择可说明迁移的名称。

`ef database update` 命令在 Migrations/<time-stamp>_InitialCreate.cs 文件中运行 `Up` 方法。`Up` 方法会创建数据库。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

检查通过依赖关系注入注册的上下文

ASP.NET Core 通过依赖关系注入进行生成。服务（例如 EF Core 数据库上下文）在应用程序启动期间通过依赖关系注入进行注册。需要这些服务（如 Razor 页面）的组件通过构造函数提供相应服务。本教程的后续部分介绍了用于获取 DB 上下文实例的构造函数代码。

基架工具自动创建 DB 上下文并将其注册到依赖关系注入容器。

检查 `Startup.ConfigureServices` 方法。基架添加了突出显示的行：

```
// This method gets called by the runtime.
// Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is
        // needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(
          Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

`RazorPagesMovieContext` 为 `Movie` 模型协调 EF Core 功能(创建、读取、更新、删除等)。数据上下文(`RazorPagesMovieContext`)派生自 Microsoft.EntityFrameworkCore.DbContext。数据上下文指定数据模型中包含哪些实体。

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Models
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext (DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; }
    }
}
```

前面的代码为实体集创建 DbSet/<Movie> 属性。在实体框架术语中，实体集通常与数据表相对应。实体对应表中的行。

通过调用 DbContextOptions 对象中的一个方法将连接字符串名称传递到上下文。进行本地开发时，ASP.NET Core 配置系统 在 *appsettings.json* 文件中读取数据库连接字符串。

`Add-Migration` 命令生成用于创建初始数据库架构的代码。此架构的依据为 `RazorPagesMovieContext` 中指定的模型(在 Data/RazorPagesMovieContext.cs 文件中)。`Initial` 参数用于为迁移命名。可以使用任何名称，但是按照惯例，会使用可说明迁移的名称。有关详细信息，请参阅迁移简介。

`Update-Database` 命令在用于创建数据库的 Migrations/{time-stamp}_InitialCreate.cs 文件中运行 `Up` 方法。

测试应用

- 运行应用并将 `/Movies` 追加到浏览器中的 URL (`http://localhost:port/movies`)。

如果收到错误:

```
SqlException: Cannot open database "RazorPagesMovieContext-GUID" requested by the login. The login failed.
Login failed for user 'User-name'.
```

缺少迁移步骤。

- 测试"创建"链接。

> **NOTE**
>
> 可能无法在 `Price` 字段中输入十进制逗号。若要使 jQuery 验证支持使用逗号(",")表示小数点的非英语区域设置，以及支持非美国英语日期格式，应用必须进行全球化。有关全球化的说明，请参阅此 GitHub 问题。

- 测试"编辑"、"详细信息"和"删除"链接。

下一个教程介绍由基架创建的文件。

上一篇：入门

下一篇：已搭建基架的 RAZOR PAGES

ASP.NET Core 中已搭建基架的 Razor 页面

2018/12/14 • • Edit Online

作者：Rick Anderson

本教程介绍上一教程中通过搭建基架创建的 Razor 页面。

查看或下载示例。

“创建”、“删除”、“详细信息”和“编辑”页面

检查 Pages/Movies/Index.cshtml.cs 页面模型：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Movies
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

        public IndexModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IList<Movie> Movie { get;set; }

        public async Task OnGetAsync()
        {
            Movie = await _context.Movie.ToListAsync();
        }
    }
}
```

Razor 页面派生自 `PageModel`。按照约定，`PageModel` 派生的类称为 `<PageName>Model`。此构造函数使用依赖关系注入将 `RazorPagesMovieContext` 添加到页。所有已搭建基架的页面都遵循此模式。请参阅异步代码，了解有关使用实体框架的异步编程的详细信息。

对页面发出请求时，`OnGetAsync` 方法向 Razor 页面返回影片列表。在 Razor 页面上调用 `OnGetAsync` 或 `OnGet` 以初始化页面状态。在这种情况下，`OnGetAsync` 将获得影片列表并显示出来。

当 `OnGet` 返回 `void` 或 `OnGetAsync` 返回 `Task` 时，不使用任何返回方法。当返回类型是 `IActionResult` 或 `Task<IActionResult>` 时，必须提供返回语句。例如，Pages/Movies/Create.cshtml.cs `OnPostAsync` 方法：

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

检查 Pages/Movies/Index.cshtml Razor 页面：

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Razor 可以从 HTML 转换为 C# 或 Razor 特定标记。当 `@` 符号后跟 Razor 保留关键字时，它会转换为 Razor 特定标记，否则会转换为 C#。

`@page` Razor 指令将文件转换为一个 MVC 操作，这意味着它可以处理请求。`@page` 必须是页面上的第一个 Razor 指令。`@page` 是转换到 Razor 特定标记的一个示例。有关详细信息，请参阅 Razor 语法。

检查以下 HTML 帮助程序中使用的 Lambda 表达式：

```
@Html.DisplayNameFor(model => model.Movie[0].Title))
```

`DisplayNameFor` HTML 帮助程序检查 Lambda 表达式中引用的 `Title` 属性来确定显示名称。检查 Lambda 表达式（而非求值）。这意味着当 `model`、`model.Movie` 或 `model.Movie[0]` 为 `null` 或为空时，不会存在任何访问冲突。对 Lambda 表达式求值时（例如，使用 `@Html.DisplayFor(modelItem => item.Title)`），将求得该模型的属性值。

@model 指令

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel
```

`@model` 指令指定传递给 Razor 页面的模型类型。在前面的示例中，`@model` 行使 `PageModel` 派生的类可用于 Razor 页面。在页面上的 `@Html.DisplayNameFor` 和 `@Html.DisplayFor` HTML 帮助程序中使用该模型。

ViewData 和布局

考虑来自 Pages/Movies/Index.cshtml 文件中的以下代码：

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}
```

前面突出显示的代码是 Razor 转换为 C# 的一个示例。`{` 和 `}` 字符括住 C# 代码块。

`PageModel` 基类具有 `ViewData` 字典属性，可用于添加要传递到某个视图的数据。可以使用键/值模式将对象添加到 `ViewData` 字典。在前面的示例中，“Title”属性被添加到 `ViewData` 字典。

“Title”属性用于 Pages/Shared/_Layout.cshtml 文件。以下标记显示 _Layout.cshtml 文件的前几行。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorPagesMovie</title>

    @*Markup removed for brevity.*@
```

行 `@*Markup removed for brevity.*@` 是不会出现在布局文件中的 Razor 注释。与 HTML 注释不同（`<!-- -->`），Razor 注释不会发送到客户端。

更新布局

更改 Pages/Shared/_Layout.cshtml 文件中的 `<title>` 元素以显示 Movie 而不是 RazorPagesMovie。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie</title>
```

在 Pages/Shared/_Layout.cshtml 文件中，查找以下定位点元素。

```
<a class="navbar-brand" asp-area="" asp-page="/Index">RazorPagesMovie</a>
```

将前面的元素替换为以下标记。

```
<a class="navbar-brand" asp-page="/Movies/Index">RpMovie</a>
```

前面的定位点元素是一个标记帮助程序。此处它是定位点标记帮助程序。`asp-page="/Movies/Index"` 标记帮助程序属性和值可以创建指向 `/Movies/Index` Razor 页面的链接。`asp-area` 属性值为空，因此在链接中未使用区域。有关详细信息，请参阅区域。

保存所做的更改，并通过单击“RpMovie”链接测试应用。如果遇到任何问题，请参阅 GitHub 中的 _Layout.cshtml 文件。

测试其他链接（“主页”、“RpMovie”、“创建”、“编辑”和“删除”）。每个页面都设置有标题，可以在浏览器选项卡中看到标题。将某个页面加入书签时，标题用于该书签。Pages/Index.cshtml 和 Pages/Movies/Index.cshtml 当前具有相同的标题，但可以修改它们以具有不同的值。

> **NOTE**
>
> 可能无法在 `Price` 字段中输入十进制逗号。若要使 jQuery 验证支持使用逗号（“,”）表示小数点的的非英语区域设置，以及支持非美国英语日期格式，必须执行使应用全球化的步骤。有关添加十进制逗号的说明，请参阅 GitHub 问题 4076。

在 Pages/_ViewStart.cshtml 文件中设置 `Layout` 属性：

```
@{
    Layout = "_Layout";
}
```

前面的标记针对所有 Razor 文件将布局文件设置为 Pages 文件夹下的 Pages/Shared/_Layout.cshtml。请参阅布局了解详细信息。

“创建”页面模型

检查 *Pages/Movies/Create.cshtml.cs* 页面模型：

```
// Unused usings removed.
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
using System;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

        public CreateModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

`OnGet` 方法初始化页面所需的任何状态。“创建”页没有任何要初始化的状态，因此返回 `Page`。教程的后面部分将介绍 `OnGet` 方法初始化状态。`Page` 方法创建用于呈现 Create.cshtml 页的 `PageResult` 对象。

`Movie` 属性使用 `[BindProperty]` 特性来选择加入模型绑定。当“创建”表单发布表单值时，ASP.NET Core 运行时将发布的值绑定到 `Movie` 模型。

当页面发布表单数据时，运行 `OnPostAsync` 方法：

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

如果不存在任何模型错误，将重新显示表单，以及发布的任何表单数据。在发布表单前，可以在客户端捕获到大部

分模型错误。模型错误的一个示例是，发布的日期字段值无法转换为日期。本教程后面讨论了客户端验证和模型验证。

如果不存在模型错误，将保存数据，并且浏览器会重定向到索引页。

创建 Razor 页面

检查 Pages/Movies/Create.cshtml Razor 页面文件：

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Visual Studio 以用于标记帮助程序的特殊加粗字体显示 `<form method="post">` 标记：

`<form method="post">` 元素是一个表单标记帮助程序。表单标记帮助程序会自动包含防伪令牌。

基架引擎在模型中为每个字段(ID 除外)创建 Razor 标记，如下所示：

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

验证标记帮助程序(`<div asp-validation-summary` 和 `<span asp-validation-for`)显示验证错误。本系列后面的部分将更详细地讨论有关验证的信息。

标签标记帮助程序(`<label asp-for="Movie.Title" class="control-label"></label>`)生成标签描述和 `Title` 属性的 `for` 特性。

输入标记帮助程序(`<input asp-for="Movie.Title" class="form-control" />`)使用 DataAnnotations 属性并在客户端生成 jQuery 验证所需的 HTML 属性。

上一篇：添加模型 下一篇：数据库

使用数据库和 ASP.NET Core

2018/12/14 • • Edit Online

作者：Rick Anderson 和 Joe Audette

View or download sample code (how to download).

`RazorPagesMovieContext` 对象处理连接到数据库并将 `Movie` 对象映射到数据库记录的任务。在 Startup.cs 的 `ConfigureServices` 方法中向依赖关系注入容器注册数据库上下文：

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

```
// This method gets called by the runtime.
// Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is
        // needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(
          Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

有关 `ConfigureServices` 中使用的方法的详细信息，请参阅：

- 面向 `CookiePolicyOptions` 的 ASP.NET Core 中的欧盟一般数据保护条例 (GDPR) 支持。
- SetCompatibilityVersion

ASP.NET Core 配置系统会读取 `ConnectionString`。为了进行本地开发，它会从 appsettings.json 文件获取连接字符串。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

生成代码的数据库名称值 (`Database={Database name}`) 将并不不同。名称值是任意的。

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "RazorPagesMovieContext": "Server=(localdb)\\mssqllocaldb;Database=RazorPagesMovieContext-
1234;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

将应用部署到测试或生产服务器时，可以使用环境变量将连接字符串设置为实际的数据库服务器。有关详细信息，请参阅配置。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

SQL Server Express LocalDB

LocalDB 是轻型版的 SQL Server Express 数据库引擎，以程序开发为目标。LocalDB 作为按需启动并在用户模式下运行的轻量级数据库没有复杂的配置。默认情况下，LocalDB 数据库在 `C:/Users/<user/>` 目录下创建 `*.mdf` 文件。

- 从“视图”菜单中，打开“SQL Server 对象资源管理器”(SSOX)。

- 右键单击 `Movie` 表，然后选择“视图设计器”：

请注意 `ID` 旁边的密钥图标。默认情况下，EF 为该主键创建一个名为 `ID` 的属性。

- 右键单击 `Movie` 表，然后选择“查看数据”：

ID	Genre	Price	ReleaseDate	Title
3	Action	1.99	1/1/0001 12:00:...	Conan
2003	Comedy	1.99	8/4/2017 6:27:3...	Back to the Future
2004	Western	1.19	8/4/2017 7:08:3...	The Good, the bad, and t...
NULL	NULL	NULL	NULL	NULL

设定数据库种子

使用以下代码在 Models 文件夹中创建一个名为 `SeedData` 的新类：

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace RazorPagesMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new RazorPagesMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<RazorPagesMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

如果 DB 中有任何电影，则会返回种子初始值设定项，并且不会添加任何电影。

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

添加种子初始值设定项

在 Program.cs 中，修改 `Main` 方法以执行以下操作：

- 从依赖关系注入容器获取数据库上下文实例。
- 调用 seed 方法，并将上下文传递给它。
- Seed 方法完成时释放上下文。

下面的代码显示更新后的 Program.cs 文件。

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using RazorPagesMovie.Models;
using System;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateWebHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context=services.
                        GetRequiredService<RazorPagesMovieContext>();
                    context.Database.Migrate();
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

生产应用不会调用 `Database.Migrate`。它会添加到上面的代码中，以防止在未运行 `Update-Database` 时出现以下异常：

SqlException：无法打开登录请求的数据库“RazorPagesMovieContext-21”。登录失败。用户“用户名”登录失败。

测试应用

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 删除 DB 中的所有记录。可以使用浏览器中的删除链接，也可以从 SSOX 执行此操作
- 强制应用初始化（调用 `Startup` 类中的方法），使种子方法能够正常运行。若要强制进行初始化，必须先停止 IIS Express，然后再重新启动它。可以使用以下任一方法来执行此操作：
  - 右键单击通知区域中的 IIS Express 系统任务栏图标，然后点击"退出"或"停止站点"：

  - 如果是在非调试模式下运行 VS 的，请按 F5 以在调试模式下运行。
  - 如果是在调试模式下运行 VS 的，请停止调试程序并按 F5。

应用将显示设定为种子的数据：

在下一教程中将对数据的展示进行整理。

上一篇：已搭建基架的 RAZOR 页面

下一篇：更新页面

在 ASP.NET Core 应用中更新生成的页面

2019/1/10 • • Edit Online

作者：Rick Anderson

构架的电影应用有个不错的开始，但是展示效果还不够理想。ReleaseDate 应是 Release Date（两个词）。

更新生成的代码

打开 Models/Movie.cs 文件，并添加以下代码中突出显示的行：

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

`[Column(TypeName = "decimal(18, 2)")]` 数据注释使 Entity Framework Core 可以将 `Price` 正确映射到数据库中的货币。有关详细信息，请参阅数据类型。

DataAnnotations 未包括在下一个教程中。Display 特性指定要显示的字段名称的内容（本例中应为“Release Date”，而不是“ReleaseDate”）。DataType 属性指定数据的类型（日期），使字段中存储的时间信息不会显示。

浏览到 Pages/Movies，并将鼠标悬停在“编辑”链接上以查看目标 URL。

“编辑”、“详细信息”和“删除”链接是在 Pages/Movies/Index.cshtml 文件中由定位标记帮助程序生成的。

```
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

标记帮助程序使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。在前面的代码中，

`AnchorTagHelper` 从 Razor 页面(路由是相对的)、`asp-page` 和路由 ID (`asp-route-id`) 动态生成 HTML `href` 特性值。有关详细信息，请参阅页面的 URL 生成。

在最喜欢的浏览器中使用"查看源"来检查生成的标记。生成的 HTML 的一部分如下所示：

```
<td>
  <a href="/Movies/Edit?id=1">Edit</a> |
  <a href="/Movies/Details?id=1">Details</a> |
  <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

动态生成的链接通过查询字符串传递电影 ID(例如，`https://localhost:5001/Movies/Details?id=1` 中的 `?id=1`)。

更新"编辑"、"详细信息"和"删除"Razor 页面以使用"{id:int?}"路由模板。将上述每个页面的页面指令从 `@page` 更改为 `@page "{id:int}"`。运行应用，然后查看源。生成的 HTML 会将 ID 添加到 URL 的路径部分：

```
<td>
  <a href="/Movies/Edit/1">Edit</a> |
  <a href="/Movies/Details/1">Details</a> |
  <a href="/Movies/Delete/1">Delete</a>
</td>
```

如果对具有"{id: int}" 路由模板的页面进行的请求中不包含整数，则将返回 HTTP 404(未找到)错误。例如，`http://localhost:5000/Movies/Details` 将返回 404 错误。若要使 ID 可选，请将 `?` 追加到路由约束：

```
@page "{id:int?}"
```

若要测试 `@page "{id:int?}"` 的行为：

- 在 Pages/Movies/Details.cshtml 中将 page 指令设置为 `@page "{id:int?}"`。
- 在 `public async Task<IActionResult> OnGetAsync(int? id)` 中(位于 Pages/Movies/Details.cshtml.cs 中)设置断点。
- 导航到 `https://localhost:5001/Movies/Details/`。

使用 `@page "{id:int}"` 指令时，永远不会命中断点。路由引擎返回 HTTP 404。使用 `@page "{id:int?}"` 时，`OnGetAsync` 方法返回 `NotFound` (HTTP 404)。

尽管不建议这样做，但仍可以将 `OnGetAsync` 方法(在 Pages/Movies/Delete.cshtml.cs 中)编写为：

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        Movie = await _context.Movie.FirstOrDefaultAsync();
    }
    else
    {
        Movie = await _context.Movie.FirstOrDefaultAsync(m => m.ID == id);
    }

    if (Movie == null)
    {
        return NotFound();
    }
    return Page();
}
```

测试上述代码：

- 选择删除链接。
- 从 URL 中删除 ID。例如，将 `https://localhost:5001/Movies/Delete/8` 更改为 `https://localhost:5001/Movies/Delete`。
- 在调试程序中逐步执行代码。

查看并发异常处理

查看 Pages/Movies/Edit.cshtml.cs 文件中的 `OnPostAsync` 方法：

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie.ID))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}

private bool MovieExists(int id)
{
    return _context.Movie.Any(e => e.ID == id);
}
```

当一个客户端删除电影并且另一个客户端对电影发布更改时，前面的代码会检测并发异常。

测试 `catch` 块：

- 在 `catch (DbUpdateConcurrencyException)` 上设置断点
- 对电影选择“编辑”，进行更改，但不要输入“保存”。
- 在其他浏览器窗口中，选择同一电影的“删除”链接，然后删除此电影。
- 在之前的浏览器窗口中，将更改发布到电影。

生产代码可能要检测并发冲突。有关详细信息，请参阅处理并发冲突。

发布和绑定审阅

检查 Pages/Movies/Edit.cshtml.cs 文件：

```
public class EditModel : PageModel
{
    private readonly RazorPagesMovieContext _context;

    public EditModel(RazorPagesMovieContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Movie Movie { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);

        if (Movie == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Attach(Movie).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!_context.Movie.Any(e => e.ID == Movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }

        return RedirectToPage("./Index");
    }
}
```

当对 Movies/Edit 页面进行 HTTP GET 请求时（例如 `http://localhost:5000/Movies/Edit/2`）：

- `OnGetAsync` 方法从数据库提取电影并返回 `Page` 方法。
- `Page` 方法呈现“Pages/Movies/Edit.cshtml”Razor 页面。Pages/Movies/Edit.cshtml 文件包含模型指令（`@model RazorPagesMovie.Pages.Movies.EditModel`），这使电影模型在页面上可用。
- “编辑”表单中会显示电影的值。

当发布 Movies/Edit 页面时：

- 此页面上的表单值将绑定到 `Movie` 属性。`[BindProperty]` 特性会启用模型绑定。

```
[BindProperty]
public Movie Movie { get; set; }
```

- 如果模型状态中存在错误(例如，`ReleaseDate` 无法被转换为日期)，则会使用已提交的值显示表单。
- 如果没有模型错误，则电影已保存。

“索引”、“创建”和“删除”Razor 页面中的 HTTP GET 方法遵循一个类似的模式。“创建”Razor 页面中的 HTTP POST `OnPostAsync` 方法遵循的模式类似于“编辑”Razor 页面中的 `OnPostAsync` 方法所遵循的模式。

在下一教程中将添加搜索。

上一篇：使用数据库 下一篇：添加搜索

将搜索添加到 ASP.NET Core Razor 页面

2019/1/10 • • Edit Online

作者：Rick Anderson

View or download sample code (how to download).

在以下部分中，添加了按流派或名称搜索电影。

将以下突出显示的属性添加到 Pages/Movies/Index.cshtml.cs：

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get; set; }
    [BindProperty(SupportsGet = true)]
    public string SearchString { get; set; }
    // Requires using Microsoft.AspNetCore.Mvc.Rendering;
    public SelectList Genres { get; set; }
    [BindProperty(SupportsGet = true)]
    public string MovieGenre { get; set; }
```

- `SearchString`：包含用户在搜索文本框中输入的文本。`SearchString` 使用 `[BindProperty]` 特性进行修饰。`[BindProperty]` 会绑定名称与属性相同的表单值和查询字符串。在 GET 请求中进行绑定需要 `(SupportsGet = true)`。
- `Genres`：包含流派列表。`Genres` 使用户能够从列表中选择一种流派。`SelectList` 需要 `using Microsoft.AspNetCore.Mvc.Rendering;`
- `MovieGenre`：包含用户选择的特定流派（例如“西部”）。
- `Genres` 和 `MovieGenre` 会在本教程的后续部分中使用。

> **WARNING**
>
> 出于安全原因，必须选择绑定 `GET` 请求数据以对模型属性进行分页。请在将用户输入映射到属性前对其进行验证。当处理依赖查询字符串或路由值的方案时，选择加入 `GET` 绑定非常有用。
>
> 要将属性绑定在 `GET` 请求上，请将 [BindProperty] 特性的 `SupportsGet` 属性设置为 `true`：`[BindProperty(SupportsGet = true)]`

使用以下代码更新索引页面的 `OnGetAsync` 方法：

```
public async Task OnGetAsync()
{
    var movies = from m in _context.Movie
                 select m;
    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    Movie = await movies.ToListAsync();
}
```

`OnGetAsync` 方法的第一行创建了 LINQ 查询用于选择电影：

```
// using System.Linq;
var movies = from m in _context.Movie
             select m;
```

此时仅对查询进行了定义，它还不会针对数据库运行。

如果 `SearchString` 属性不为 null 或空，则电影查询会修改为根据搜索字符串进行筛选：

```
if (!string.IsNullOrEmpty(SearchString))
{
    movies = movies.Where(s => s.Title.Contains(SearchString));
}
```

`s => s.Title.Contains()` 代码是 Lambda 表达式。Lambda 在基于方法的 LINQ 查询中用作标准查询运算符方法的参数，如 Where 方法或 `Contains`（前面的代码中所使用的）。在对 LINQ 查询进行定义或通过调用方法（如 `Where`、`Contains` 或 `OrderBy`）进行修改后，此查询不会被执行。相反，会延迟执行查询。这意味着表达式的计算会延迟，直到循环访问其实现的值或者调用 `ToListAsync` 方法为止。有关详细信息，请参阅 Query Execution（查询执行）。

注意：Contains 方法在数据库中运行，而不是在 C# 代码中运行。查询是否区分大小写取决于数据库和排序规则。在 SQL Server 上，`Contains` 映射到 SQL LIKE，这是不区分大小写的。在 SQLite 中，由于使用了默认排序规则，因此需要区分大小写。

导航到电影页面，并向 URL追加一个如 `?searchString=Ghost` 的查询字符串（例如 `https://localhost:5001/Movies?searchString=Ghost`）。筛选的电影将显示出来。

如果向索引页面添加了以下路由模板，搜索字符串则可作为 URL 段传递（例如 `https://localhost:5001/Movies/Ghost`）。

```
@page "{searchString?}"
```

前面的路由约束允许按路由数据（URL 段）搜索标题，而不是按查询字符串值进行搜索。`"{searchString?}"` 中的 `?` 表示这是可选路由参数。

ASP.NET Core 运行时使用模型绑定，通过查询字符串（`?searchString=Ghost`）或路由数据（

`https://localhost:5001/Movies/Ghost`）设置 `SearchString` 属性的值。模型绑定搜索不区分大小写。

但是，不能指望用户修改 URL 来搜索电影。在此步骤中，会添加 UI 来筛选电影。如果已添加路由约束 `"{searchString?}"`，请将它删除。

打开 Pages/Movies/Index.cshtml 文件，并添加以下代码中突出显示的 `<form>` 标记：

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

HTML `<form>` 标记使用以下标记帮助程序：

- 表单标记帮助程序。提交表单时，筛选器字符串将通过查询字符串发送到 Pages/Movies/Index 页面。
- 输入标记帮助程序

保存更改并测试筛选器。

按流派搜索

使用以下代码更新 `OnGetAsync` 方法：

```
public async Task OnGetAsync()
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    if (!string.IsNullOrEmpty(MovieGenre))
    {
        movies = movies.Where(x => x.Genre == MovieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

下面的代码是一种 LINQ 查询，可从数据库中检索所有流派。

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

流派的 `SelectList` 是通过投影截然不同的流派创建的。

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

将按流派搜索添加到 Razor 页面

更新 Index.cshtml，如下所示：

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

通过按流派或/和电影标题搜索来测试应用。

上一篇：更新页面 下一篇：添加新字段

将新字段添加到 ASP.NET Core 中的 Razor 页面

2019/1/10 • • Edit Online

作者：Rick Anderson

View or download sample code (how to download).

在此部分中，Entity Framework Code First 迁移用于：

- 将新字段添加到模型。
- 将新字段架构更改迁移到数据库。

使用 EF Code First 自动创建数据库时，Code First 将：

- 向数据库添加表格，以跟踪数据库的架构是否与从生成它的模型类同步。
- 如果该模型类未与数据库同步，EF 将引发异常。

通过自动验证同步的架构/模型可以更容易地发现不一致的数据库/代码问题。

向电影模型添加分级属性

打开 Models/Movie.cs 文件，并添加 `Rating` 属性：

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

构建应用程序。

编辑 Pages/Movies/Index.cshtml，并添加 `Rating` 字段：

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
```

```
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">

    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movie)
        {
        <tr><td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Rating)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
        }
    </tbody>
</table>
```

更新以下页面：

- 将 `Rating` 字段添加到“删除”和“详细信息”页面。
- 使用 `Rating` 字段更新 Create.cshtml。
- 将 `Rating` 字段添加到“编辑”页面。

在 DB 更新为包括新字段之前，应用将不会正常工作。如果立即运行，应用会引发 `SqlException`：

```
SqlException: Invalid column name 'Rating'.
```

此错误是由于更新的 Movie 模型类与数据库的 Movie 表架构不同导致的。（数据库表中没有 `Rating` 列。）

可通过几种方法解决此错误：

1. 让 Entity Framework 自动丢弃并使用新的模型类架构重新创建数据库。此方法在开发周期早期很方便；通过它可以一起快速改进模型和数据库架构。此方法的缺点是会导致数据库中的现有数据丢失。请勿对生产数据库使用此方法！ 当架构更改时丢弃数据库并使用初始值设定项以使用测试数据自动设定数据库种子，这通常是开发应用的有效方式。

2. 对现有数据库架构进行显式修改，使它与模型类相匹配。此方法的优点是可以保留数据。可以手动或通过创建数据库更改脚本进行此更改。

3. 使用 Code First 迁移更新数据库架构。

对于本教程，请使用 Code First 迁移。

更新 `SeedData` 类，使它提供新列的值。示例更改如下所示，但可能需要对每个 `new Movie` 块做出此更改。

```
context.Movie.AddRange(
    new Movie
    {
        Title = "When Harry Met Sally",
        ReleaseDate = DateTime.Parse("1989-2-12"),
        Genre = "Romantic Comedy",
        Price = 7.99M,
        Rating = "R"
    },
```

请参阅已完成的 SeedData.cs 文件。

生成解决方案。

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

添加用于评级字段的迁移

从“工具”菜单中，选择“NuGet 包管理器”>“包管理器控制台”。在 PMC 中，输入以下命令：

```
Add-Migration Rating
Update-Database
```

`Add-Migration` 命令会通知框架执行以下操作：

- 将 `Movie` 模型与 `Movie` DB 架构进行比较。
- 创建代码以将 DB 架构迁移到新模型。

名称“Rating”是任意的，用于对迁移文件进行命名。为迁移文件使用有意义的名称是有帮助的。

`Update-Database` 命令指示框架将架构更改应用到数据库。

如果删除 DB 中的所有记录，种子初始值设定项会设定 DB 种子，并将包括 `Rating` 字段。可以使用浏览器中的删除链接，也可以从 Sql Server 对象资源管理器 (SSOX) 执行此操作。

另一个方案是删除数据库，并使用迁移来重新创建该数据库。删除 SSOX 中的数据库：

- 在 SSOX 中选择数据库。
- 右键单击数据库，并选择“删除”。
- 检查“关闭现有连接”。

- 选择“确定”。
- 在 PMC 中更新数据库：

```
Update-Database
```

运行应用，并验证是否可以创建/编辑/显示具有 `Rating` 字段的电影。如果数据库未设定种子，则在 `SeedData.Initialize` 方法中设置断点。

上一篇：添加搜索 下一篇：添加验证

将验证添加到 ASP.NET Core Razor 页面

2018/12/14 • • Edit Online

作者：Rick Anderson

本部分中向 `Movie` 模型添加了验证逻辑。每当用户创建或编辑电影时，都会强制执行验证规则。

验证

软件开发的一个关键原则被称为 DRY（即“不要自我重复”）。Razor 页面鼓励进行仅指定一次功能的开发，且功能在整个应用中反映。DRY 可以帮助：

- 减少应用中的代码量。
- 使代码更加不易出错，且更易于测试和维护。

Razor 页面和 Entity Framework 提供的验证支持是 DRY 原则的极佳示例。验证规则在模型类中的某处以声明方式指定，且在应用的所有位置强制执行。

将验证规则添加到电影模型

打开 Models/Movie.cs 文件。DataAnnotations 提供一组内置验证特性，可通过声明方式应用于类或属性。DataAnnotations 还包含 `DataType` 等格式特性，有助于格式设置但不提供验证。

更新 `Movie` 类以使用 `Required`、`StringLength`、`RegularExpression` 和 `Range` 验证特性。

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

验证特性用于指定模型属性上强制执行的行为：

- `Required` 和 `MinimumLength` 特性指示属性必须具有一个值。但是，用户可以随时输入空格以对可以为 null 的类型进行验证约束。从本质上来说，需要不可以为 null 的值类型（如 `decimal`、`int`、`float` 和 `DateTime`），但

不需要 `Required` 特性。

- `RegularExpression` 特性限制用户可以输入的字符。在前面的代码中，`Genre` 必须以一个或多个大写字母开始，并且后接零个或多个字母、单引号或双引号、空白字符或短划线。`Rating` 必须以一个或多个大写字母开始，并且后接零个或多个字母、数字、单引号或双引号、空白字符或短划线。
- `Range` 特性将值限制在指定范围内。
- `StringLength` 特性设置字符串的最大长度，且可视情况设置最小长度。

让 ASP.NET Core 强制自动执行验证规则有助于提升应用的可靠性。自动验证模型有助于保护应用，因为添加新代码时无需手动应用它们。

Razor 页面中的“验证错误”UI

运行应用并导航到“页面/电影”。

选择“新建”链接。使用无效值填写表单。当 jQuery 客户端验证检测到错误时，会显示一条错误消息。

> **NOTE**
>
> 可能无法在小数字段中输入十进制逗号。若要使 jQuery 验证支持使用逗号(“,”)表示小数点的的非英语区域设置，以及支持非美国英语日期格式，必须执行使应用全球化的步骤。有关添加十进制逗号的说明，请参阅 GitHub 问题 4076。

请注意表单如何自动呈现每个包含无效值的字段中的验证错误消息。客户端(使用 JavaScript 和 jQuery)和服务器

端(若用户禁用 JavaScript)都必定会遇到这些错误。

一项重要优势是，无需在“创建”或“编辑”页面中更改代码。在模型应用 DataAnnotations 后，即已启用验证 UI。本教程中自动创建的 Razor 页面自动选取了验证规则(使用 `Movie` 模型类的属性上的验证特性)。使用“编辑”页面测试验证后，即已应用相同验证。

存在客户端验证错误时，不会将表单数据发布到服务器。请通过以下一种或多种方法验证是否未发布表单数据：

- 在 `OnPostAsync` 方法中放置一个断点。提交表单(选择“创建”或“保存”)。从未命中断点。
- 使用 Fiddler 工具。
- 使用浏览器开发人员工具监视网络流量。

服务器端验证

在浏览器中禁用 JavaScript 后，提交出错表单将发布到服务器。

(可选)测试服务器端验证：

- 在浏览器中禁用 JavaScript。可以使用浏览器的开发人员工具执行此操作。如果无法在浏览器中禁用 JavaScript，请尝试其他浏览器。
- 在“创建”或“编辑”页面的 `OnPostAsync` 方法中设置断点。
- 提交带有验证错误的表单。
- 验证模型状态是否无效：

```
if (!ModelState.IsValid)
{
    return Page();
}
```

以下代码显示了之前在本教程中设定其基架的“Create.cshtml”的一部分。它用于在“创建”和“编辑”页面中显示初始表单并在发生错误后重新显示表单。

```
<form method="post">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Movie.Title" class="control-label"></label>
        <input asp-for="Movie.Title" class="form-control" />
        <span asp-validation-for="Movie.Title" class="text-danger"></span>
    </div>
```

输入标记帮助程序使用 DataAnnotations 特性并在客户端生成 jQuery 验证所需的 HTML 特性。验证标记帮助程序用于显示验证错误。有关详细信息，请参阅验证。

“创建”和“编辑”页面中没有验证规则。仅可在 `Movie` 类中指定验证规则和错误字符串。这些验证规则将自动应用于编辑 `Movie` 模型的 Razor 页面。

需要更改验证逻辑时，也只能在该模型中更改。将始终在整个应用程序中应用验证(在一处定义验证逻辑)。单处验证有助于保持代码干净，且更易于维护和更新。

使用 DataType 特性

检查 `Movie` 类。除了一组内置的验证特性，`System.ComponentModel.DataAnnotations` 命名空间还提供格式特性。`DataType` 特性应用于 `ReleaseDate` 和 `Price` 属性。

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

`DataType` 特性仅提供相关提示来帮助视图引擎设置数据格式(并提供特性，例如向 URL 提供 `<a>` 和向电子邮件提供 `<a href="mailto:EmailAddress.com">`)。使用 `RegularExpression` 特性验证数据的格式。`DataType` 属性用于指定比数据库内部类型更具体的数据类型。`DataType` 特性不是验证特性。示例应用程序中仅显示日期，不显示时间。

`DataType` 枚举提供了多种数据类型，例如日期、时间、电话号码、货币、电子邮件地址等。应用程序还可通过 `DataType` 特性自动提供类型特定的功能。例如，可为 `DataType.EmailAddress` 创建 `mailto:` 链接。可在支持 HTML5 的浏览器中为 `DataType.Date` 提供日期选择器。`DataType` 特性发出 HTML 5 `data-`(读作 data dash)特性供 HTML 5 浏览器使用。`DataType` 特性不提供任何验证。

`DataType.Date` 不指定显示日期的格式。默认情况下，数据字段根据基于服务器的 `CultureInfo` 的默认格式进行显示。

要使 Entity Framework Core 能将 `Price` 正确地映射到数据库中的货币，则必须使用 `[Column(TypeName = "decimal(18, 2)")]` 数据注释。有关详细信息，请参阅数据类型。

`DisplayFormat` 特性用于显式指定日期格式：

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

`ApplyFormatInEditMode` 设置用于指定在显示值进行编辑时需应用格式。可能不希望某些字段具有此行为。例如，在货币值中，可能不希望编辑 UI 中存在货币符号。

可单独使用 `DisplayFormat` 特性，但通常建议使用 `DataType` 特性。`DataType` 特性传达数据的语义而不是传达如何在屏幕上呈现数据，并提供 DisplayFormat 不具备的以下优势：

- 浏览器可启用 HTML5 功能(例如显示日历控件、区域设置适用的货币符号、电子邮件链接等)
- 默认情况下，浏览器将根据区域设置采用正确的格式呈现数据。
- 借助 `DataType` 特性，ASP.NET Core 框架可选择适当的字段模板来呈现数据。单独使用时，`DisplayFormat` 特性将使用字符串模板。

注意：jQuery 验证不适用于 `Range` 属性和 `DateTime`。例如，以下代码将始终显示客户端验证错误，即便日期在指定的范围内：

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

通常，在模型中编译固定日期是不恰当的，因此不推荐使用 `Range` 特性和 `DateTime`。

以下代码显示组合在一行上的特性：

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

Razor Pages 和 EF Core 入门显示了 Razor Pages 的高级 EF Core 操作。

发布到 Azure

有关部署到 Azure 的信息，请参阅教程：使用 SQL 数据库在 Azure 中生成 ASP.NET 应用。此说明适用于 ASP.NET 应用，而不适用于 ASP.NET Core 应用，但步骤均相同。

感谢读完这篇 Razor 页面简介。我们非常感谢你的反馈。Razor Pages 和 EF Core 入门是本教程的优选后续教程。

其他资源

- ASP.NET Core 表单中的标记帮助程序
- ASP.NET Core 全球化和本地化
- ASP.NET Core 中的标记帮助程序
- 在 ASP.NET Core 中创作标记帮助程序

上一篇：添加新字段

ASP.NET Core MVC 概述

2018/8/31 • • Edit Online

作者：Steve Smith

ASP.NET Core MVC 是使用"模型-视图-控制器"设计模式构建 Web 应用和 API 的丰富框架。

什么是 MVC 模式？

模型-视图-控制器 (MVC) 体系结构模式将应用程序分成 3 个主要组件组：模型、视图和控制器。此模式有助于实现关注点分离。使用此模式，用户请求被路由到控制器，后者负责使用模型来执行用户操作和/或检索查询结果。控制器选择要显示给用户的视图，并为其提供所需的任何模型数据。

下图显示 3 个主要组件及其相互引用关系：

这种责任划分有助于根据复杂性缩放应用程序，因为这更易于编码、调试和测试有单一作业（并遵循 Single Responsibility Principle（单一责任原则））的某些内容（模型、视图或控制器）。但这会加大更新、测试和调试代码的难度，该代码在这 3 个领域的两个或多个领域间存在依赖关系。例如，用户界面逻辑的变更频率往往高于业务逻辑。如果将表示代码和业务逻辑组合在单个对象中，则每次更改用户界面时都必须修改包含业务逻辑的对象。这常常会引发错误，并且需要在每次进行细微的用户界面更改后重新测试业务逻辑。

> **NOTE**
>
> 视图和控制器均依赖于模型。但是，模型既不依赖于视图，也不依赖于控制器。这是分离的一个关键优势。这种分离允许模型独立于可视化展示进行构建和测试。

模型责任

MVC 应用程序的模型 (M) 表示应用程序和任何应由其执行的业务逻辑或操作的状态。业务逻辑应与保持应用程序状态的任何实现逻辑一起封装在模型中。强类型视图通常使用 ViewModel 类型，旨在包含要在该视图上显示的数据。控制器从模型创建并填充 ViewModel 实例。

> **NOTE**
>
> 可通过多种方法在使用 MVC 体系结构模式的应用中组织模型。详细了解某些不同种类的模型类型。

视图责任

视图 (V) 负责通过用户界面展示内容。它们使用 Razor 视图引擎在 HTML 标记中嵌入 .NET 代码。视图中应该有最小逻辑，并且其中的任何逻辑都必须与展示内容相关。如果发现需要在视图文件中执行大量逻辑以显示复杂模型中的数据，请考虑使用 View Component、ViewModel 或视图模板来简化视图。

控制器职责

控制器 (C) 是处理用户交互、使用模型并最终选择要呈现的视图的组件。在 MVC 应用程序中，视图仅显示信息；控制器处理并响应用户输入和交互。在 MVC 模式中，控制器是初始入口点，负责选择要使用的模型类型和要呈现的视图（因此得名 - 它控制应用如何响应给定请求）。

> **NOTE**
>
> 控制器不应由于责任过多而变得过于复杂。要阻止控制器逻辑变得过于复杂，请使用 Single Responsibility Principle（单一责任原则）将业务逻辑推出控制器并推入域模型。

> **TIP**
>
> 如果发现控制器操作经常执行相同类型的操作，则可将这些常见操作移入筛选器，并遵守"不要自我重复"原则。

什么是 ASP.NET Core MVC

ASP.NET Core MVC 框架是轻量级、开源、高度可测试的演示框架，并针对 ASP.NET Core 进行了优化。

ASP.NET Core MVC 提供一种基于模式的方式，用于生成可彻底分开管理事务的动态网站。它提供对标记的完全控制，支持 TDD 友好开发并使用最新的 Web 标准。

功能

ASP.NET Core MVC 包括以下功能：

- 路由
- 模型绑定
- 模型验证
- 依赖关系注入
- 筛选器
- 区域
- Web API
- 可测试性
- Razor 视图引擎
- 强类型视图
- 标记帮助程序
- 视图组件

路由

ASP.NET Core MVC 建立在 ASP.NET Core 的路由之上，是一个功能强大的 URL 映射组件，可用于生成具有易于理解和可搜索 URL 的应用程序。它可让你定义适用于搜索引擎优化 (SEO) 和链接生成的应用程序 URL 命名模式，而不考虑如何组织 Web 服务器上的文件。可以使用支持路由值约束、默认值和可选值的方便路由模板语法来定义路由。

通过基于约定的路由，可以全局定义应用程序接受的 URL 格式以及每个格式映射到给定控制器上特定操作方法的方式。接收传入请求时，路由引擎分析 URL 并将其匹配到定义的 URL 格式之一，然后调用关联的控制器操作方法。

```
routes.MapRoute(name: "Default", template: "{controller=Home}/{action=Index}/{id?}");
```

借助属性路由，可以通过用定义应用程序路由的属性修饰控制器和操作来指定路由信息。这意味着路由定义位于与之相关联的控制器和操作旁。

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
  [HttpGet("{id}")]
  public IActionResult GetProduct(int id)
  {
    ...
  }
}
```

模型绑定

ASP.NET Core MVC 模型绑定将客户端请求数据（窗体值、路由数据、查询字符串参数、HTTP 头）转换到控制器可以处理的对象中。因此，控制器逻辑不必找出传入的请求数据；它只需具备作为其操作方法的参数的数据。

```
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null) { ... }
```

模型验证

ASP.NET Core MVC 通过使用数据注释验证属性修饰模型对象来支持验证。验证属性在值发布到服务器前在客户端上进行检查，并在调用控制器操作前在服务器上进行检查。

```
using System.ComponentModel.DataAnnotations;
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Display(Name = "Remember me?")]
    public bool RememberMe { get; set; }
}
```

控制器操作：

```
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    if (ModelState.IsValid)
    {
      // work with the model
    }
    // At this point, something failed, redisplay form
    return View(model);
}
```

框架处理客户端和服务器上的验证请求数据。在模型类型上指定的验证逻辑作为非介入式注释添加到呈现的视图，并使用 jQuery 验证在浏览器中强制执行。

依赖关系注入

ASP.NET Core 内置有对依赖关系注入 (DI) 的支持。在 ASP.NET Core MVC 中，控制器可通过其构造函数请求所需服务，使其能够遵循 Explicit Dependencies Principle（显式依赖关系原则）。

应用还可通过 `@inject` 指令使用视图文件中的依赖关系注入：

```
@inject SomeService ServiceName
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ServiceName.GetTitle</title>
</head>
<body>
    <h1>@ServiceName.GetTitle</h1>
</body>
</html>
```

筛选器

筛选器帮助开发者封装横切关注点，例如异常处理或授权。筛选器允许操作方法运行自定义预处理和后处理逻辑，并且可以配置为在给定请求的执行管道内的特定点上运行。筛选器可以作为属性应用于控制器或操作（也可以全局运行）。此框架中包括多个筛选器（例如 `Authorize`）。`[Authorize]` 是用于创建 MVC 授权筛选器的属性。

```
[Authorize]
   public class AccountController : Controller
   {
```

区域

区域提供将大型 ASP.NET Core MVC Web 应用分区为较小功能分组的方法。区域是应用程序内的一个 MVC 结构。在 MVC 项目中，模型、控制器和视图等逻辑组件保存在不同的文件夹中，MVC 使用命名约定来创建这些组件之间的关系。对于大型应用，将应用分区为独立的高级功能区域可能更有利。例如，具有多个业务单位（如结账、计费、搜索等）的电子商务应用。每个单位都有自己的逻辑组件视图、控制器和模型。

Web API

除了作为生成网站的强大平台，ASP.NET Core MVC 还对生成 Web API 提供强大的支持。可以生成可连接大量客户端（包括浏览器和移动设备）的服务。

该框架包括对 HTTP 内容协商的支持，后者有允许设置数据格式为 JSON 或 XML 的内置支持。编写自定义格式化程序以添加对自己格式的支持。

使用链接生成启用对超媒体的支持。轻松启用对跨域资源共享 (CORS) 的支持，以便 Web API 可以跨多个 Web 应用程序共享。

可测试性

框架对界面和依赖项注入的使用非常适用于单元测试，并且该框架还包括使得集成测试快速轻松的功能（例如 TestHost 和实体框架的 InMemory 提供程序）。详细了解如何测试控制器逻辑。

Razor 视图引擎

ASP.NET Core MVC 视图使用 Razor 视图引擎呈现视图。Razor 是一种紧凑、富有表现力且流畅的模板标记语言，用于使用嵌入式 C# 代码定义视图。Razor 用于在服务器上动态生成 Web 内容。可以完全混合服务器代码与客户端内容和代码。

```
<ul>
  @for (int i = 0; i < 5; i++) {
    <li>List item @i</li>
  }
</ul>
```

使用 Razor 视图引擎可以定义布局、分部视图和可替换部分。

强类型视图

可以基于模型强类型化 MVC 中的 Razor 视图。控制器可以将强类型化的模型传递给视图，使视图具备类型检查和 IntelliSense 支持。

例如，以下视图呈现类型为 `IEnumerable<Product>` 的模型：

```
@model IEnumerable<Product>
<ul>
    @foreach (Product p in Model)
    {
        <li>@p.Name</li>
    }
</ul>
```

标记帮助程序

标记帮助程序使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。可以使用标记帮助程序定义自定义标记（例如 `<environment>`），或者修改现有标记的行为（例如 `<label>`）。标记帮助程序基于元素名称及其属性绑定到特定的元素。它们提供了服务器端呈现的优势，同时仍然保留了 HTML 编辑体验。

有多种常见任务（例如创建窗体、链接，加载资产等）的内置标记帮助程序，公共 GitHub 存储库和 NuGet 包中甚至还有更多可用标记帮助程序。标记帮助程序使用 C# 创建，基于元素名称、属性名称或父标记以 HTML 元素为目标。例如，内置 LinkTagHelper 可以用来创建指向 `AccountsController` 的 `Login` 操作的链接：

```
<p>
    Thank you for confirming your email.
    Please <a asp-controller="Account" asp-action="Login">Click here to Log in</a>.
</p>
```

可以使用 `EnvironmentTagHelper` 在视图中包括基于运行时环境（例如开发、暂存或生产）的不同脚本（例如原始或缩减脚本）：

```
<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"
            asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
            asp-fallback-test="window.jQuery">
    </script>
</environment>
```

标记帮助程序提供 HTML 友好型开发体验和用于创建 HTML 和 Razor 标记的丰富 IntelliSense 环境。大多数内置标记帮助程序以现有 HTML 元素为目标，为该元素提供服务器端属性。

视图组件

通过视图组件可以包装呈现逻辑并在整个应用程序中重用它。这些组件类似于分部视图，但具有关联逻辑。

兼容性版本

SetCompatibilityVersion 方法允许应用选择加入或退出 ASP.NET Core MVC 2.1 或更高版本中引入的潜在中断行为变更。

有关更多信息，请参见ASP.NET Core MVC 的兼容性版本。

在带有 Visual Studio 的 Windows 上使用 ASP.NET Core MVC 创建 Web 应用

2019/1/2 • • Edit Online

本教程介绍具有控制器和视图的 ASP.NET Core MVC Web 开发。如果你是初次接触 ASP.NET Core Web 开发，请考虑本教程的 Razor Pages 版本，此版本提供了更简单的起点。

本系列教程包括以下部分：

1. 入门
2. 添加控制器
3. 添加视图
4. 添加模型
5. 使用 SQL Server LocalDB
6. 控制器方法和视图
7. 添加搜索
8. 添加新字段
9. 添加验证
10. 检查 Details 和 Delete 方法

ASP.NET Core MVC 入门

2019/1/10 • • Edit Online

作者：Rick Anderson

本教程介绍具有控制器和视图的 ASP.NET Core MVC Web 开发。如果你是初次接触 ASP.NET Core Web 开发，请考虑本教程的 Razor Pages 版本，此版本提供了更简单的起点。

https://docs.microsoft.com/en-us/visualstudio/ide/visual-studio-ide?view=vs-2017

本教程介绍构建 ASP.NET Core MVC Web 应用的基础知识。

该应用管理电影标题的数据库。您将学习如何：

- 创建 Web 应用。
- 添加和构架模型。
- 使用数据库。
- 添加搜索和验证。

在结束时，你会获得可以管理和显示电影数据的应用。

查看或下载示例代码（如何下载）。

> **NOTE**
>
> 我们要测试所建议的新的 ASP.NET Core 目录结构是否可用。如果你有几分钟时间进行练习，来了解当前目录或所建议目录中的 7 个不同主题，请单击此处来参与调查。

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

创建 Web 应用

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

在 Visual Studio 中，选择“文件”>“新建”>“项目”。

填写"新建项目"对话框：

- 在左侧窗格中，选择".NET Core"
- 在中间窗格中，选择"ASP.NET Core Web 应用程序(.NET Core)"
- 将项目命名为"MvcMovie"（请务必将项目命名为"MvcMovie"，以便在复制代码时可以与命名空间匹配。）
- 选择"确定"

完成"新建 ASP.NET Core Web 应用程序(.NET Core) - MvcMovie"对话框：

- 在版本选择器下拉框中选择"ASP.NET Core 2.2"
- 选择"Web 应用(模型-视图-控制器)"
- 选择"确定"。

Visual Studio 为刚刚创建的 MVC 项目使用默认模板。输入项目名称并选择几个选项后，就拥有了一个可正常运行的应用。这是一个基本的初学者项目，适合入门使用。

选择 Ctrl+F5 以在非调试模式下运行应用。

- Visual Studio 启动 IIS Express 并运行应用。请注意，地址栏显示 `localhost:port#`，而不显示 `example.com` 之类的内容。这是因为 `localhost` 是本地计算机的标准主机名。Visual Studio 创建 Web 项目时，Web 服务器使用的是随机端口。在上图中，端口号为 5000。浏览器中的 URL 显示 `localhost:5000`。运行应用时，将看到不同的端口号。
- 使用"Ctrl+F5"启动应用(非调试模式)后，可执行代码更改、保存文件、刷新浏览器和查看代码更改等操作。许多开发人员更喜欢使用非调试模式快速启动应用并查看更改。
- 可以从"调试"菜单项中以调试或非调试模式启动应用：

- 可以通过选择"IIS Express"按钮来调试应用

- 选择"接受"以同意跟踪。此应用不会跟踪个人信息。模板生成的代码包含有助于符合一般数据保护条例 (GDPR) 的资产。

下图展示了接受跟踪后的应用：

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Visual Studio 帮助

- 了解如何使用 Visual Studio 调试 C# 代码
- Visual Studio IDE 简介

在本教程的下一部分中，你将了解 MVC 并开始撰写一些代码。

下一页

将控制器添加到 ASP.NET Core MVC 应用

2019/1/10 • • Edit Online

作者：Rick Anderson

模型-视图-控制器 (MVC) 体系结构模式将应用分成 3 个主要组件：模型 (M)、视图 (V) 和控制器 (C)。MVC 模式有助于创建比传统单片应用更易于测试和更新的应用。基于 MVC 的应用包含：

- 模型 (M)：表示应用数据的类。模型类使用验证逻辑来对该数据强制实施业务规则。通常，模型对象检索模型状态并将其存储在数据库中。本教程中，`Movie` 模型将从数据库中检索电影数据，并将其提供给视图或对其进行更新。更新后的数据将写入到数据库。
- 视图 (V)：视图是显示应用用户界面 (UI) 的组件。此 UI 通常会显示模型数据。
- 控制器 (C)：处理浏览器请求的类。它们检索模型数据并调用返回响应的视图模板。在 MVC 应用中，视图仅显示信息；控制器处理并响应用户输入和交互。例如，控制器处理路由数据和查询字符串值，并将这些值传递给模型。该模型可使用这些值查询数据库。例如，`https://localhost:1234/Home/About` 具有 `Home`（控制器）的路由数据和 `About`（在 Home 控制器上调用的操作方法）。`https://localhost:1234/Movies/Edit/5` 是一个请求，用于通过电影控制器编辑 ID 为 5 的电影。本教程的后续部分中将介绍路由数据。

MVC 模式可帮助创建分隔不同应用特性（输入逻辑、业务逻辑和 UI 逻辑）的应用，同时让这些元素之间实现松散耦合。该模式可指定应用中每种逻辑的位置。UI 逻辑位于视图中。输入逻辑位于控制器中。业务逻辑位于模型中。这种隔离有助于控制构建应用时的复杂程度，因为它可用于一次处理一个实现特性，而不影响其他特性的代码。例如，处理视图代码时不必依赖业务逻辑代码。

本教程系列中介绍了这些概念，并展示了如何使用它们构建电影应用。MVC 项目包含“控制器”和“视图”文件夹。

添加控制器

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 在“解决方案资源管理器”中，右键单击“控制器”，然后单击“添加”>“控制器”

- 在“添加基架”对话框中，选择“MVC 控制器 - 空”

- 在“添加空 MVC 控制器”对话框中，输入 HelloWorldController 并选择“ADD”。

将“Controllers/HelloWorldController.cs”的内容替换为以下内容：

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        //
        // GET: /HelloWorld/Welcome/

        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

控制器中的每个 `public` 方法均可作为 HTTP 终结点调用。上述示例中，两种方法均返回一个字符串。请注意每个方法前面的注释。

HTTP 终结点是 Web 应用程序中可定向的 URL（例如 `https://localhost:5001/HelloWorld`），其中结合了所用的协议 `HTTPS`、TCP 端口等 Web 服务器的网络位置 `localhost:5001`，以及目标 URI `HelloWorld`。

第一条注释指出这是一个 HTTP GET 方法，它通过向基 URL 追加 `/HelloWorld/` 进行调用。第二条注释指定一个 HTTP GET 方法，它通过向 URL 追加 `/HelloWorld/Welcome/` 进行调用。本教程稍后将使用基架引擎生成 `HTTP POST` 方法，用于更新数据。

在非调试模式下运行应用，并将"HelloWorld"追加到地址栏中的路径。`Index` 方法返回一个字符串。

MVC 根据入站 URL 调用控制器类（及其中的操作方法）。MVC 所用的默认 URL 路由逻辑使用如下格式来确定调

用的代码：

`/[Controller]/[ActionName]/[Parameters]`

在 Startup.cs 文件的 `Configure` 方法中设置路由格式。

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

如果浏览到应用且不提供任何 URL 段，它将默认为上面突出显示的模板行中指定的“Home”控制器和“Index”方法。

第一个 URL 段决定要运行的控制器类。因此 `localhost:xxxx/HelloWorld` 映射到 `HelloWorldController` 类。该 URL 段的第二部分决定类上的操作方法。因此 `localhost:xxxx/HelloWorld/Index` 将触发 `HelloWorldController` 类的 `Index` 运行。请注意，只需浏览到 `localhost:xxxx/HelloWorld`，而 `Index` 方法默认调用。原因是 `Index` 是默认方法，如果未显式指定方法名称，则将在控制器上调用它。URL 段的第三部分（`id`）针对的是路由数据。本教程的后续部分中将介绍路由数据。

浏览到 `https://localhost:xxxx/HelloWorld/Welcome`。`Welcome` 方法将运行并返回字符串 `This is the Welcome action method...`。对于此 URL，采用 `HelloWorld` 控制器和 `Welcome` 操作方法。目前尚未使用 URL 的 `[Parameters]` 部分。

修改代码，将一些参数信息从 URL 传递到控制器。例如 `/HelloWorld/Welcome?name=Rick&numtimes=4`。更改 `Welcome` 方法以包括以下代码中显示的两个参数：

```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

前面的代码：

- 使用 C# 可选参数功能指示，未为 `numTimes` 参数传递值时该参数默认为 1。
- 使用 `HtmlEncoder.Default.Encode` 防止恶意输入（即 JavaScript）损害应用。
- 在 `$"Hello {name}, NumTimes is: {numTimes}"` 中使用内插字符串。

运行应用并浏览到：

`https://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4`

（将 xxxx 替换为端口号。）可在 URL 中对 `name` 和 `numtimes` 使用其他值。MVC 模型绑定系统可将命名参数从地址栏中的查询字符串自动映射到方法中的参数。有关详细信息，请参阅模型绑定。

在上图中，未使用 URL 段（`Parameters`），且 `name` 和 `numTimes` 参数作为查询字符串进行传递。上述 URL 中的 `?`（问号）为分隔符，后接查询字符串。`&` 字符用于分隔查询字符串。

将 `Welcome` 方法替换为以下代码：

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

运行应用并输入以下 URL：`https://localhost:xxx/HelloWorld/Welcome/3?name=Rick`

此时，第三个 URL 段与路由参数 `id` 相匹配。`Welcome` 方法包含 `MapRoute` 方法中匹配 URL 模板的参数 `id`。后面的 `?`（`id?` 中）表示 `id` 参数可选。

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

上述示例中，控制器始终执行 MVC 的“VC”部分，即视图和控制器工作。控制器将直接返回 HTML。通常不希望控制器直接返回 HTML，因为编码和维护非常繁琐。通常，需使用单独的 Razor 视图模板文件来帮助生成 HTML 响应。可在下一教程中执行该操作。

将视图添加到 ASP.NET Core MVC 应用

2019/1/10 • • Edit Online

作者：Rick Anderson

在本部分中，将修改 `HelloWorldController` 类，进而使用 Razor 视图文件来顺利封装为客户端生成 HTML 响应的过程。

使用 Razor 创建视图模板文件。基于 Razor 的模板具有“.cshtml”文件扩展名。它们提供了一种巧妙的方法来使用 C# 创建 HTML 输出。

当前，`Index` 方法返回带有在控制器类中硬编码的消息的字符串。在 `HelloWorldController` 类中，将 `Index` 方法替换为以下代码：

```
public IActionResult Index()
{
    return View();
}
```

上述代码返回 `View` 对象。它使用视图模板对浏览器生成 HTML 响应。类似上述 `Index` 方法的控制器方法（也称为操作方法）通常返回 IActionResult（或派生自 `ActionResult` 的类），而非类似字符串的类型。

添加视图

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 右键单击“视图”文件夹，然后单击“添加”>“新文件夹”，并将文件夹命名为“HelloWorld”。
- 右键单击“Views/HelloWorld”文件夹，然后单击“添加”>“新项”。
- 在“添加新项 - MvcMovie”对话框中
  - 在右上角的搜索框中，输入“视图”
  - 选择“Razor 视图”
  - 保持“名称”框的值：Index.cshtml。
  - 选择“添加”

使用以下内容替换 Razor 视图文件 Views/HelloWorld/Index.cshtml 的内容：

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

导航到 `https://localhost:xxxx/HelloWorld`。`HelloWorldController` 中的 `Index` 方法作用不大；它运行 `return View();` 语句，指定此方法应使用视图模板文件来呈现对浏览器的响应。因为没有显式指定视图模板文件的名称，所以 MVC 默认使用 /Views/HelloWorld 文件夹中的 Index.cshtml 视图文件。下面图片显示了视图中硬编码的 字符串“Hello from our View Template!”

更改视图和布局页面

选择菜单链接（“MvcMovie”、“首页”和“隐私”）。每页显示相同的菜单布局。菜单布局是在 Views/Shared/_Layout.cshtml 文件中实现的。打开 Views/Shared/_Layout.cshtml 文件。

布局模板使你能够在一个位置指定网站的 HTML 容器布局，然后将它应用到网站中的多个页面。查找 `@RenderBody()` 行。`RenderBody` 是显示创建的所有特定于视图的页面的占位符，已包装在布局页面中。例如，如果选择“隐私”链接，Views/Home/Privacy.cshtml 视图将在 `RenderBody` 方法中呈现。

更改布局文件中的标题和菜单链接

- 在标题元素中，将 `MvcMovie` 更改为 `Movie App`。
- 将定位点元素 `<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>` 更改为 `<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>`。

下列标记显示了突出显示的更改：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute"
              crossorigin="anonymous"
              integrity="sha256-eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE="/>
    </environment>
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-
shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <partial name="_CookieConsentPartial" />
        <main role="main" class="pb-3">
```

```
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - Movie App - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=">
        </script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/js/bootstrap.bundle.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha256-E/V4cWE4qvAeO5MOhjtGtqDzPndRO1LBk8lJ/PR7CA4=">
        </script>
    </environment>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

在前面的标记中，省略了 `asp-area` 定位点标记帮助程序属性，因为此应用未使用区域。

说明：`Movies` 控制器尚未实现。此时，`Movie App` 链接不起作用。

保存更改并选择“隐私”链接。请注意浏览器选项卡上的标题现在显示的是“隐私 - 电影应用”，而不是“隐私 - Mvc 电影”：

选择“主页”链接，请注意，标题和定位文本还会显示“电影应用”。我们能够在布局模板中进行一次更改，让网站上的所有页面都反映新的链接文本和新标题。

检查 Views/_ViewStart.cshtml 文件：

```
@{
    Layout = "_Layout";
}
```

Views/_ViewStart.cshtml 文件将 Views/Shared/_Layout.cshtml 文件引入到每个视图中。可以使用 `Layout` 属性设置不同的布局视图，或将它设置为 `null`，这样将不会使用任何布局文件。

更改 Views/HelloWorld/Index.cshtml 视图文件的 `<h2>` 元素：

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

稍稍对标题和 `<h2>` 元素进行一些更改，这样可以看出哪一段代码更改了显示。

上述代码中的 `ViewData["Title"] = "Movie List";` 将 `ViewData` 字典的 `Title` 属性设置为“Movie List”。`Title` 属性在布局页面中的 `<title>` HTML 元素中使用：

```
<title>@ViewData["Title"] - Movie App</title>
```

保存更改并导航到 `https://localhost:xxxx/HelloWorld`。请注意，浏览器标题、主标题和辅助标题已更改。（如果没有在浏览器中看到更改，则可能正在查看缓存的内容。在浏览器中按 Ctrl + F5 强制加载来自服务器的响应。）浏览器标题是使用我们在 Index.cshtml 视图模板中设置的 `ViewData["Title"]` 以及在布局文件中添加的额外“ - Movie App”创建的。

还要注意，Index.cshtml 视图模板中的内容是如何与 Views/Shared/_Layout.cshtml 视图模板合并的，并且注意单个 HTML 响应被发送到了浏览器。凭借布局模板可以很容易地对应用程序中所有页面进行更改。若要了解更多信息，请参阅布局。

但我们这一点点“数据”(在此示例中为“Hello from our View Template!” 消息)是硬编码的。MVC 应用程序有一个“V”(视图)，而你已有一个“C”(控制器)，但还没有“M”(模型)。

将数据从控制器传递给视图

控制器操作会被调用以响应传入的 URL 请求。控制器类是编写处理传入浏览器请求的代码的地方。控制器从数据源检索数据，并决定将哪些类型的响应发送回浏览器。可以从控制器使用视图模板来生成并格式化对浏览器的 HTML 响应。

控制器负责提供所需的数据，使视图模板能够呈现响应。最佳做法：视图模板不应该直接执行业务逻辑或与数据库进行交互。相反，视图模板应仅使用由控制器提供给它的数据。保持此“分离关注点”有助于保持代码干净以及可测试性和可维护性。

目前，`HelloWorldController` 类中的 `Welcome` 方法采用 `name` 和 `ID` 参数，然后将值直接输出到浏览器。应将控制器更改为使用视图模板，而不是使控制器将此响应呈现为字符串。视图模板会生成动态响应，这意味着必须将适当的数据位从控制器传递给视图以生成响应。为此，可以让控制器将视图模板所需的动态数据(参数)放置在视图模板稍后可以访问的 `ViewData` 字典中。

在 HelloWorldController.cs 中，更改 `Welcome` 方法以将 `Message` 和 `NumTimes` 值添加到 `ViewData` 字典。`ViewData` 字典是一个动态对象，这意味着可以使用任何类型；只有将内容放在其中后 `ViewData` 对象才具有定义的属性。MVC 模型绑定系统自动将命名参数(`name` 和 `numTimes`)从地址栏中的查询字符串映射到方法中的参数。完整的 HelloWorldController.cs 文件如下所示：

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

`ViewData` 字典对象包含将传递给视图的数据。

创建一个名为 Views/HelloWorld/Welcome.cshtml 的 Welcome 视图模板。

在 Welcome.cshtml 视图模板中创建一个循环，显示“Hello” `NumTimes` 。使用以下内容替换 Views/HelloWorld/Welcome.cshtml 的内容：

```
@{
    ViewData["Title"] = "Welcome";
}

<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

保存更改并浏览到以下 URL：

`https://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4`

数据取自 URL，并传递给使用 MVC 模型绑定器的控制器。控制器将数据打包到 `ViewData` 字典中，并将该对象传递给视图。然后，视图将数据作为 HTML 呈现给浏览器。

在上面的示例中，我们使用 `ViewData` 字典将数据从控制器传递给视图。稍后在本教程中，我们将使用视图模型将数据从控制器传递给视图。传递数据的视图模型方法通常比 `ViewData` 字典方法更为优先。有关详细信息，请参阅何时使用 ViewBag、ViewData 或 TempData。

在下一个教程中，将创建电影数据库。

上一页 下一页

将模型添加到 ASP.NET Core MVC 应用

2019/1/10 • • Edit Online

作者：Rick Anderson 和 Tom Dykstra

在本部分中将添加用于管理数据库中的电影的类。这些类将是 MVC 应用的“Model”部分。

可以结合 Entity Framework Core (EF Core) 使用这些类来处理数据库。EF Core 是对象关系映射 (ORM) 框架，可以简化需要编写的数据访问代码。

要创建的模型类称为 POCO 类（源自“简单传统 CLR 对象”），因为它们与 EF Core 没有任何依赖关系。它们只定义将存储在数据库中的数据的属性。

在本教程中，首先要编写模型类，然后 EF Core 将创建数据库。有一种备选方法（此处未介绍）：从现有数据库生成模型类。有关此方法的信息，请参阅 ASP.NET Core - Existing Database（ASP.NET Core - 现有数据库）。

添加数据模型类

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

右键单击 Models 文件夹，然后单击“添加” > “类”。将类命名“Movie”。

Add the following properties to the `Movie` class:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `Movie` class contains:

- The `Id` field which is required by the database for the primary key.
- `[DataType(DataType.Date)]`: The DataType attribute specifies the type of the data (`Date`). With this attribute:
  - The user is not required to enter time information in the date field.
  - Only the date is displayed, not time information.

DataAnnotations are covered in a later tutorial.

搭建“电影”模型的基架

在此部分，将搭建“电影”模型的基架。确切地说，基架工具将生成页面，用于对“电影”模型执行创建、读取、更新和删除 (CRUD) 操作。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

在解决方案资源管理器中，右键单击“Controllers”文件夹 >“添加”>“新搭建基架的项目”。

在“添加基架”对话框中，选择“包含视图的 MVC 控制器(使用 Entity Framework)”>“添加”。

填写“添加控制器”对话框：

- 模型类：Movie (MvcMovie.Models)

- 数据上下文类：选择 + 图标并添加默认的 MvcMovie.Models.MvcMovieContext

- 视图：将每个选项保持为默认选中状态
- 控制器名称：保留默认的 MoviesController
- 选择“添加”

Visual Studio 将创建：

- Entity Framework Core 数据库上下文类 (Data/MvcMovieContext.cs)
- 电影控制器 (Controllers/MoviesController.cs)
- “创建”、“删除”、“详细信息”、“编辑”和“索引”页面的 Razor 视图文件 (Views/Movies/*.cshtml)

自动创建数据库上下文和 CRUD（创建、读取、更新和删除）操作方法和视图的过程称为“搭建基架”。

如果运行应用并单击“Mvc 电影”链接，则会出现以下类似的错误：

```
An unhandled exception occurred while processing the request.

SqlException: Cannot open database "MvcMovieContext-<GUID removed>" requested by the login. The login failed.
Login failed for user 'Rick'.

System.Data.SqlClient.SqlInternalConnectionTds..ctor(DbConnectionPoolIdentity identity, SqlConnectionString
```

你需要创建数据库，并且使用 EF Core 迁移功能来执行此操作。通过迁移可创建与数据模型匹配的数据库，并在

数据模型更改时更新数据库架构。

初始迁移

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

在此部分中，程序包管理器控制台 (PMC) 用于：

- 添加初始迁移。
- 使用初始迁移来更新数据库。

从“工具”菜单中，选择“NuGet 包管理器” > “包管理器控制台”。

在 PMC 中，输入以下命令：

```
Add-Migration Initial
Update-Database
```

`Add-Migration` 命令生成用于创建初始数据库架构的代码。

此架构以（Models/MvcMovieContext.cs 文件中的）`DbContext` 中指定的模型为基础。`InitialCreate` 参数用于为迁移命名。可以使用任何名称，但是按照惯例，会选择可说明迁移的名称。

`ef database update` 命令在 Migrations/<time-stamp>_InitialCreate.cs 文件中运行 `Up` 方法。`Up` 方法会创建数据库。

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

检查通过依赖关系注入注册的上下文

ASP.NET Core 通过依赖关系注入进行生成。服务（例如 EF Core 数据库上下文）在应用程序启动期间通过依赖关系注入进行注册。需要这些服务（如 Razor 页面）的组件通过构造函数提供相应服务。本教程的后续部分介绍了用于获取 DB 上下文实例的构造函数代码。

基架工具自动创建 DB 上下文并将其注册到依赖关系注入容器。

检查 `Startup.ConfigureServices` 方法。基架添加了突出显示的行：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies
        // is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<MvcMovieContext>(options =>
         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

`MvcMovieContext` 为 `Movie` 模型协调 EF Core 功能(创建、读取、更新、删除等)。数据上下文 (`MvcMovieContext`) 派生自 Microsoft.EntityFrameworkCore.DbContext。数据上下文指定数据模型中包含哪些实体:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace MvcMovie.Models
{
    public class MvcMovieContext : DbContext
    {
        public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
            : base(options)
        {
        }

        public DbSet<MvcMovie.Models.Movie> Movie { get; set; }
    }
}
```

前面的代码为实体集创建 `DbSet<Movie>` 属性。在实体框架术语中，实体集通常与数据表相对应。实体对应表中的行。

通过调用 DbContextOptions 对象中的一个方法将连接字符串名称传递到上下文。进行本地开发时，ASP.NET Core 配置系统 在 *appsettings.json* 文件中读取数据库连接字符串。

此架构基于在 Data/MvcMovieContext.cs 文件中的 `MvcMovieContext` 中指定的模型。`Initial` 参数用于为迁移命名。可以使用任何名称，但是按照惯例，会使用可说明迁移的名称。有关详细信息，请参阅迁移简介。

`Update-Database` 命令在用于创建数据库的 Migrations/{time-stamp}_InitialCreate.cs 文件中运行 `Up` 方法。

测试应用

- 运行应用并将 `/Movies` 追加到浏览器中的 URL (`http://localhost:port/movies`)。

如果收到如下所示数据库异常:

```
SqlException: Cannot open database "MvcMovieContext-GUID" requested by the login. The login failed.
Login failed for user 'User-name'.
```

缺少迁移步骤。

- 测试“创建”链接。

> **NOTE**
>
> 可能无法在 `Price` 字段中输入十进制逗号。若要使 jQuery 验证支持使用逗号(“,”)表示小数点的非英语区域设置，以及支持非美国英语日期格式，应用必须进行全球化。有关全球化的说明，请参阅此 GitHub 问题。

- 测试“编辑”、“详细信息”和“删除”链接。

检查 `Startup` 类：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies
        // is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<MvcMovieContext>(options =>
         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

上面突出显示的代码显示了要添加到依赖关系注入容器的电影数据库上下文：

- `services.AddDbContext<MvcMovieContext>(options =>` 指定要使用的数据库和连接字符串。
- `=>` 是 lambda 运算符

打开 Controllers/MoviesController.cs 文件并检查构造函数：

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
```

构造函数使用依赖关系注入将数据库上下文（`MvcMovieContext`）注入到控制器中。数据库上下文将在控制器中的每个 CRUD 方法中使用。

强类型模型和 @model 关键词

在本教程之前的内容中，已经介绍了控制器如何使用 `ViewData` 字典将数据或对象传递给视图。`ViewData` 字典是一个动态对象，提供了将信息传递给视图的方便的后期绑定方法。

MVC 还提供将强类型模型对象传递给视图的功能。凭借此强类型方法可更好地对代码进行编译时检查。基架机制在创建方法和视图时，通过 `MoviesController` 类和视图使用了此方法(即传递强类型模型)。

检查 Controllers/MoviesController.cs 文件中生成的 `Details` 方法：

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

`id` 参数通常作为路由数据传递。例如 `https://localhost:5001/movies/details/1` 的设置如下：

- 控制器被设置为 `movies` 控制器（第一个 URL 段）。
- 操作被设置为 `details`（第二个 URL 段）。
- ID 被设置为 1（最后一个 URL 段）。

还可以使用查询字符串传入 `id`，如下所示：

`https://localhost:5001/movies/details?id=1`

在未提供 ID 值的情况下，`id` 参数可定义为可以为 null 的类型（`int?`）。

Lambda 表达式会被传入 `FirstOrDefaultAsync` 以选择与路由数据或查询字符串值相匹配的电影实体。

```
var movie = await _context.Movie
    .FirstOrDefaultAsync(m => m.Id == id);
```

如果找到了电影，`Movie` 模型的实例则会被传递到 `Details` 视图：

```
return View(movie);
```

检查 Views/Movies/Details.cshtml 文件的内容：

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

通过将 `@model` 语句包括在视图文件的顶端，可以指定视图期望的对象类型。创建电影控制器时，会自动在 Details.cshtml 文件的顶端包括以下 `@model` 语句：

```
@model MvcMovie.Models.Movie
```

此 `@model` 指令使你能够使用强类型的 `Model` 对象访问控制器传递给视图的电影。例如，在 Details.cshtml 视图中，代码通过强类型的 `Model` 对象将每个电影字段传递给 `DisplayNameFor` 和 `DisplayFor` HTML 帮助程序。`Create` 和 `Edit` 方法以及视图也传递一个 `Movie` 模型对象。

检查电影控制器中的 Index.cshtml 视图和 `Index` 方法。请注意代码在调用 `View` 方法时是如何创建 `List` 对象的。代码将此 `Movies` 列表从 `Index` 操作方法传递给视图：

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

创建电影控制器时，基架会自动在 Index.cshtml 文件的顶端包含以下 `@model` 语句：

```
@model IEnumerable<MvcMovie.Models.Movie>
```

`@model` 指令使你能够使用强类型的 `Model` 对象访问控制器传递给视图的电影列表。例如，在 Index.cshtml 视图中，代码使用 `foreach` 语句通过强类型 `Model` 对象对电影进行循环遍历：

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

因为 `Model` 对象为强类型（作为 `IEnumerable<Movie>` 对象），因此循环中的每个项都被类型化为 `Movie`。除其他优点之外，这意味着可对代码进行编译时检查：

其他资源

- 标记帮助程序
- 全球化和本地化

上一篇：添加视图

下一篇：使用 SQL

在 ASP.NET Core 中使用 SQL

2019/1/10 • • Edit Online

作者：Rick Anderson

`MvcMovieContext` 对象处理连接到数据库并将 `Movie` 对象映射到数据库记录的任务。在 Startup.cs 文件的 `ConfigureServices` 方法中向依赖关系注入容器注册数据库上下文：

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies
        // is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<MvcMovieContext>(options =>
         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

ASP.NET Core 配置系统会读取 `ConnectionString` 。为了进行本地开发，它会从 appsettings.json 文件获取连接字符串：

```
"ConnectionStrings": {
  "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-
2;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

将应用部署到测试或生产服务器时，可使用环境变量或另一种方法将连接字符串设置为实际的 SQL Server。有关详细信息，请参阅配置。

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

SQL Server Express LocalDB

LocalDB 是轻型版的 SQL Server Express 数据库引擎，以程序开发为目标。LocalDB 作为按需启动并在用户模式下运行的轻量级数据库没有复杂的配置。默认情况下，LocalDB 数据库在 C:/Users/{user} 目录中创建 .mdf 文件。

- 从“视图”菜单中，打开“SQL Server 对象资源管理器”(SSOX)。

- 右键单击 `Movie` 表，然后单击“视图设计器”

请注意 `ID` 旁边的密钥图标。默认情况下，EF 将名为 `ID` 的属性设置为主键。

- 右键单击 `Movie` 表，然后单击“查看数据”

设定数据库种子

在 Models 文件夹中创建一个名为 `SeedData` 的新类。将生成的代码替换为以下代码：

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

如果 DB 中没有任何电影，则会返回种子初始值设定项，并且不会添加任何电影。

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

添加种子初始值设定项

将 Program.cs 的内容替换为以下代码：

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using System;
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;
using MvcMovie;

namespace MvcMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateWebHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<MvcMovieContext>();
                    context.Database.Migrate();
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

测试应用

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- 删除 DB 中的所有记录。可以使用浏览器中的删除链接，也可从 SSOX 执行此操作。
- 强制应用初始化（调用 `Startup` 类中的方法），使种子方法能够正常运行。若要强制进行初始化，必须先停止 IIS Express，然后再重新启动它。可以使用以下任一方法来执行此操作：
  - 右键单击通知区域中的 IIS Express 系统任务栏图标，然后点击“退出”或“停止站点”

- 如果是在非调试模式下运行 VS 的，请按 F5 以在调试模式下运行
- 如果是在调试模式下运行 VS 的，请停止调试程序并按 F5

应用将显示设定为种子的数据。

ASP.NET Core 中的控制器方法和视图

2019/1/10 • • Edit Online

作者：Rick Anderson

我们的电影应用有个不错的开始，但是展示效果还不够理想。我们不希望看到时间（如下图所示的 12:00:00 AM），并且“ReleaseDate”应为两个词。

打开 Models/Movie.cs 文件，并添加以下代码中突出显示的行：

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

下一教程将介绍 DataAnnotations。Display 特性指定要显示的字段名称的内容(本例中应为"Release Date"，而不是"ReleaseDate")。DataType 属性指定数据的类型(日期)，使字段中存储的时间信息不会显示。

要使 Entity Framework Core 能将 `Price` 正确地映射到数据库中的货币，则必须使用 `[Column(TypeName = "decimal(18, 2)")]` 数据注释。有关详细信息，请参阅数据类型。

浏览到 `Movies` 控制器，并将鼠标指针悬停在"编辑"链接上以查看目标 URL。

"编辑"、"详细信息"和"删除"链接是在 Views/Movies/Index.cshtml 文件中由 Core MVC 定位标记帮助程序生成的。

```
        <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
        <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
        <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
    </td>
</tr>
```

标记帮助程序使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。在上面的代码中，`AnchorTagHelper` 从控制器操作方法和路由 ID 动态生成 HTML `href` 特性值。在最喜欢的浏览器中使用“查看源”，或使用开发人员工具来检查生成的标记。生成的 HTML 的一部分如下所示：

```
 <td>
    <a href="/Movies/Edit/4"> Edit </a> |
    <a href="/Movies/Details/4"> Details </a> |
    <a href="/Movies/Delete/4"> Delete </a>
</td>
```

重新调用在 Startup.cs 文件中设置的路由的格式：

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

ASP.NET Core 将 `https://localhost:5001/Movies/Edit/4` 转换为对 `Movies` 控制器的 `Edit` 操作方法的请求，参数 `Id` 为 4。（控制器方法也称为操作方法。）

标记帮助程序是 ASP.NET Core 中最受欢迎的新功能之一。有关详细信息，请参阅其他资源。

打开 `Movies` 控制器并检查两个 `Edit` 操作方法。以下代码显示了 `HTTP GET Edit` 方法，此方法将提取电影并填充由 Edit.cshtml Razor 文件生成的编辑表单。

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

以下代码显示 `HTTP POST Edit` 方法，它会处理已发布的电影值：

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

以下代码显示 `HTTP POST Edit` 方法，它会处理已发布的电影值：

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

`[Bind]` 特性是防止过度发布的一种方法。只应在 `[Bind]` 特性中包含想要更改的属性。有关详细信息，请参阅防止控制器过度发布。ViewModels 提供了一种替代方法以防止过度发布。

请注意第二个 `Edit` 操作方法的前面是 `[HttpPost]` 特性。

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

`HttpPost` 特性指定只能为 `POST` 请求调用此 `Edit` 方法。可将 `[HttpGet]` 属性应用于第一个编辑方法，但不是必需，因为 `[HttpGet]` 是默认设置。

`ValidateAntiForgeryToken` 特性用于防止请求伪造，并与编辑视图文件 (Views/Movies/Edit.cshtml) 中生成的防伪标记相配对。编辑视图文件使用表单标记帮助程序生成防伪标记。

```
<form asp-action="Edit">
```

表单标记帮助程序会生成隐藏的防伪标记，此标记必须与电影控制器的 `Edit` 方法中 `[ValidateAntiForgeryToken]` 生成的防伪标记相匹配。有关详细信息，请参阅反请求伪造。

`HttpGet Edit` 方法采用电影 `ID` 参数，使用Entity Framework `SingleOrDefaultAsync` 方法查找电影，并将所选电影返回到“编辑”视图。如果无法找到电影，则返回 `NotFound` (HTTP 404)。

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

当基架系统创建“编辑”视图时，它会检查 `Movie` 类并创建代码为类的每个属性呈现 `<label>` 和 `<input>` 元素。以下示例显示由 Visual Studio 基架系统生成的“编辑”视图：

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Id" />
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ReleaseDate" class="control-label"></label>
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Genre" class="control-label"></label>
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

请注意视图模板在文件顶端有一个 `@model MvcMovie.Models.Movie` 语句。`@model MvcMovie.Models.Movie` 指定视图期望的视图模板的模型为 `Movie` 类型。

基架的代码使用几个标记帮助程序方法来简化 HTML 标记。标签标记帮助程序显示字段的名称（“Title”、“ReleaseDate”、“Genre”或“Price”）。输入标记帮助程序呈现 HTML `<input>` 元素。验证标记帮助程序显示与该属性相关联的任何验证消息。

运行应用程序并导航到 `/Movies` URL。点击“编辑”链接。在浏览器中查看页面的源。为 `<form>` 元素生成的 HTML 如下所示。

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field is required." id="ID" name="ID"
value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
                <input class="form-control" type="text" id="Genre" name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg-
replace="true"></span>
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <input class="form-control" type="text" data-val="true" data-val-number="The field Price must
be a number." data-val-required="The Price field is required." id="Price" name="Price" value="3.99" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Price" data-valmsg-
replace="true"></span>
            </div>
        </div>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmqUyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>
```

`<input>` 元素位于 `HTML <form>` 元素中，后者的 `action` 特性设置为发布到 `/Movies/Edit/id` URL。当单击 `Save` 按钮时，表单数据将发布到服务器。关闭 `</form>` 元素之前的最后一行显示表单标记帮助程序生成的隐藏的 XSRF 标记。

处理 POST 请求

以下列表显示了 `Edit` 操作方法的 `[HttpPost]` 版本。

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

`[ValidateAntiForgeryToken]` 特性验证表单标记帮助程序中的防伪标记生成器生成的隐藏的 XSRF 标记

模型绑定系统采用发布的表单值，并创建一个作为 `movie` 参数传递的 `Movie` 对象。`ModelState.IsValid` 方法验证表单中提交的数据是否可以用于修改(编辑或更新) `Movie` 对象。如果数据有效，将保存此数据。通过调用数据库上下文的 `SaveChangesAsync` 方法，将更新(编辑)的电影数据保存到数据库。保存数据后，代码将用户重定向到 `MoviesController` 类的 `Index` 操作方法，此方法显示电影集合，包括刚才所做的更改。

在表单发布到服务器之前，客户端验证会检查字段上的任何验证规则。如果有任何验证错误，则将显示错误消息，并且不会发布表单。如果禁用 JavaScript，则不会进行客户端验证，但服务器将检测无效的发布值，并且表单值将与错误消息一起重新显示。稍后在本教程中，我们将更详细地研究模型验证。Views/Movies/Edit.cshtml 视图模板中的验证标记帮助程序负责显示相应的错误消息。

电影控制器中的所有 `HttpGet` 方法都遵循类似的模式。它们获取电影对象(对于 `Index` 获取的是对象列表)并将对象(模型)传递给视图。`Create` 方法将空的电影对象传递给 `Create` 视图。在方法的 `[HttpPost]` 重载中，创建、编辑、删除或以其他方式修改数据的所有方法都执行此操作。以 `HTTP GET` 方式修改数据是一种安全隐患。以 `HTTP GET` 方法修改数据也违反了 HTTP 最佳做法和架构 REST 模式，后者指定 GET 请求不应更改应用程序的状态。换句话说，执行 GET 操作应是没有任何隐患的安全操作，也不会修改持久数据。

其他资源

- 全球化和本地化
- 标记帮助程序简介
- 创作标记帮助程序
- 防请求伪造
- 防止控制器过度发布
- ViewModels
- 表单标记帮助程序

- 输入标记帮助程序
- 标签标记帮助程序
- 选择标记帮助程序
- 验证标记帮助程序

上一页 下一页

将搜索添加到 ASP.NET Core MVC 应用

2019/1/10 • • Edit Online

作者：Rick Anderson

在本部分中，将向 `Index` 操作方法添加搜索功能，以实现按"类型"或"名称"搜索电影。

使用以下代码更新 `Index` 方法：

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

`Index` 操作方法的第一行创建了 LINQ 查询用于选择电影：

```
var movies = from m in _context.Movie
             select m;
```

此时仅对查询进行了定义，它还不会针对数据库运行。

如果 `searchString` 参数包含一个字符串，电影查询则会被修改为根据搜索字符串的值进行筛选：

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

上面的 `s => s.Title.Contains()` 代码是 Lambda 表达式。Lambda 在基于方法的 LINQ 查询中用作标准查询运算符方法的参数，如 Where 方法或 `Contains`（上述的代码中所使用的）。在对 LINQ 查询进行定义或通过调用方法（如 `Where`、`Contains` 或 `OrderBy`）进行修改后，此查询不会被执行。相反，会延迟执行查询。这意味着表达式的计算会延迟，直到真正循环访问其实现的值或者调用 `ToListAsync` 方法为止。有关延迟执行查询的详细信息，请参阅Query Execution（查询执行）。

注意:Contains 方法在数据库上运行，而不是在上面显示的 C# 代码中运行。查询是否区分大小写取决于数据库和排序规则。在 SQL Server 上，Contains 映射到 SQL LIKE，这是不区分大小写的。在 SQLlite 中，由于使用了默认排序规则，因此需要区分大小写。

导航到 `/Movies/Index`。将查询字符串（如 `?searchString=Ghost`）追加到 URL。筛选的电影将显示出来。

如果将 `Index` 方法的签名更改为具有名称为 `id` 的参数，则 `id` 参数将匹配 Startup.cs 中设置的默认路由的可选 `{id}` 占位符。

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

将参数更改为 `id`，并将出现的所有 `searchString` 更改为 `id`。

之前的 `Index` 方法：

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

更新后带 `id` 参数的 `Index` 方法：

```
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(await movies.ToListAsync());
}
```

现可将搜索标题作为路由数据（URL 段）而非查询字符串值进行传递。

但是，不能指望用户在每次要搜索电影时都修改 URL。因此需要添加 UI 元素来帮助他们筛选电影。若已更改 `Index` 方法的签名，以测试如何传递绑定路由的 `ID` 参数，请改回原样，使其采用名为 `searchString` 的参数：

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

打开“Views/Movies/Index.cshtml”文件，并添加以下突出显示的 `<form>` 标记：

```
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
```

此 HTML `<form>` 标记使用表单标记帮助程序，因此提交表单时，筛选器字符串会发布到电影控制器的 `Index` 操作。保存更改，然后测试筛选器。

如你所料，不存在 `Index` 方法的 `[HttpPost]` 重载。无需重载，因为该方法不更改应用的状态，仅筛选数据。

可添加以下 `[HttpPost] Index` 方法。

```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

`notUsed` 参数用于创建 `Index` 方法的重载。本教程稍后将对此进行探讨。

如果添加此方法，则操作调用程序将与 `[HttpPost] Index` 方法匹配，且将运行 `[HttpPost] Index` 方法，如下图所示。

但是，即使添加 `Index` 方法的 `[HttpPost]` 版本，其实现方式也受到限制。假设你想要将特定搜索加入书签，或向朋友发送一个链接，让他们单击链接即可查看筛选出的相同电影列表。请注意，HTTP POST 请求的 URL 与 GET 请求的 URL 相同 (localhost:xxxxx/Movies/Index)，其中不包含搜索信息。搜索字符串信息作为表单域值发送给服务器。可使用浏览器开发人员工具或出色的 Fiddler 工具对其进行验证。下图展示了 Chrome 浏览器开发人员工具：

在请求正文中，可看到搜索参数和 XSRF 标记。请注意，正如之前教程所述，表单标记帮助程序 会生成一个 XSRF 防伪标记。不会修改数据，因此无需验证控制器方法中的标记。

搜索参数位于请求正文而非 URL 中，因此无法捕获该搜索信息进行书签设定或与他人共享。将通过指定请求为 `HTTP GET` 进行修复：

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
```

现在提交搜索后，URL 将包含搜索查询字符串。即使具备 `HttpPost Index` 方法，搜索也将转到 `HttpGet Index` 操作方法。

以下标记显示对 `form` 标记的更改：

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

添加按流派搜索

将以下 `MovieGenreViewModel` 类添加到“模型”文件夹：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie> Movies;
        public SelectList Genres;
        public string MovieGenre { get; set; }
        public string SearchString { get; set; }
    }
}
```

“电影流派”视图模型将包含：

- 电影列表。
- 包含流派列表的 `SelectList`。这使用户能够从列表中选择一种流派。
- 包含所选流派的 `MovieGenre`。
- `SearchString` 包含用户在搜索文本框中输入的文本。

将 `MoviesController.cs` 中的 `Index` 方法替换为以下代码：

```
// GET: Movies
public async Task<IActionResult> Index(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel
    {
        Genres = new SelectList(await genreQuery.Distinct().ToListAsync()),
        Movies = await movies.ToListAsync()
    };

    return View(movieGenreVM);
}
```

以下代码是一种 `LINQ` 查询，可从数据库中检索所有流派。

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

通过投影不同的流派创建 `SelectList`（我们不希望选择列表中的流派重复）。

当用户搜索某个项目时，搜索值会保留在搜索框中。

向“索引”视图添加“按流派搜索”

按如下更新 `Index.cshtml`：

```
@model MvcMovie.Models.MovieGenreViewModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>

        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

检查以下 HTML 帮助程序中使用的 Lambda 表达式：

```
@Html.DisplayNameFor(model => model.Movies[0].Title)
```

在上述代码中，`DisplayNameFor` HTML 帮助程序检查 Lambda 表达式中引用的 `Title` 属性来确定显示名称。由于只检查但未计算 Lambda 表达式，因此当 `model`、`model.Movies[0]` 或 `model.Movies` 为 `null` 或空时，你不会收到访问冲突。对 Lambda 表达式求值时（例如，`@Html.DisplayFor(modelItem => item.Title)`），将求得该模型的属性值。

通过按流派搜索、按电影标题搜索以及按流派和电影标题搜索来测试应用：

将新字段添加到 ASP.NET Core MVC 应用

2019/1/10 • • Edit Online

作者：Rick Anderson

在此部分中，Entity Framework Code First 迁移用于：

- 将新字段添加到模型。
- 将新字段迁移到数据库。

使用 EF Code First 自动创建数据库时，Code First 将：

- 将表添加到数据库，以跟踪数据库的架构。
- 验证数据库与生成它的模型类是否同步。如果它们不同步，EF 则会引发异常。这使查找不一致的数据库/代码问题变得更加轻松。

向电影模型添加分级属性

将 `Rating` 属性添加到 Models/Movie.cs：

```
public class Movie
{
    public int Id { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

生成应用 (Ctrl+Shift+B)。

因为已经添加新字段到 `Movie` 类，所以需要更新绑定允许名单，将此新属性纳入其中。在 MoviesController.cs 中，更新 `Create` 和 `Edit` 操作方法的 `[Bind]` 属性，以包括 `Rating` 属性：

```
[Bind("ID,Title,ReleaseDate,Genre,Price,Rating")]
```

更新视图模板以在浏览器视图中显示、创建和编辑新的 `Rating` 属性。

编辑 /Views/Movies/Index.cshtml 文件并添加 `Rating` 字段：

```
<thead>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Genre)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Rating)
        </th>
        <th></th>
    </tr>
</thead>
<tbody>
    @foreach (var item in Model.Movies)
    {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Rating)
        </td>
        <td>
            <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
```

使用 `Rating` 字段更新 /Views/Movies/Create.cshtml。

- Visual Studio / Visual Studio for Mac
- Visual Studio Code

可以复制/粘贴之前的“窗体组”，并让 intelliSense 帮助更新字段。IntelliSense 适用于标记帮助程序。

```
        </div>
        <div class="form-group">
            <label asp-for="Title" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger" />
            </div>
        </div>
        <div class="form-group">
            <label asp-for="R" class="col-md-2 control-label"></label>
            <div class="col-
                <input asp-f                                    n-control" />
                <span asp-va                                    class="text-danger" />
            </div>
        </div>
        <div class="form-gro
            <div class="col-                                    ">
                <input type=                                    e" class="btn btn-default" />
            </div>
        </div>
    </div>
</form>
```

```
GetHashCode
GetType
ID
Price
Rating
ReleaseDate
Title
ToString
```

更新 `SeedData` 类，使它提供新列的值。示例更改如下所示，但可能需要对每个 `new Movie` 做出此更改。

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

在 DB 更新为包括新字段之前，应用将不会正常工作。如果它现在运行，将引发以下 `SqlException`：

```
SqlException: Invalid column name 'Rating'.
```

发生此错误是因为更新的 Movie 模型类与现有数据库的 Movie 表架构不同。（数据库表中没有 `Rating` 列。）

可通过几种方法解决此错误：

1. 让 Entity Framework 自动丢弃，并基于新的模型类架构重新创建数据库。在测试数据库上进行开发时，此方法在开发周期早期很方便；通过它可以一起快速改进模型和数据库架构。但其缺点是会丢失数据库中的现有数据 - 因此请勿对生产数据库使用此方法！ 使用初始值设定项，以使用测试数据自动设定数据库种子，这通常是开发应用程序的有效方式。对于早期开发和使用 SQLite 的情况，这是一个不错的方法。

2. 对现有数据库架构进行显式修改，使它与模型类相匹配。此方法的优点是可以保留数据。可以手动或通过创建数据库更改脚本进行此更改。

3. 使用 Code First 迁移更新数据库架构。

对于本教程，请使用 Code First 迁移。

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

从“工具”菜单中，选择“NuGet 包管理器”>“包管理器控制台”。

在 PMC 中，输入以下命令：

```
Add-Migration Rating
Update-Database
```

`Add-Migration` 命令会通知迁移框架使用当前 `Movie` DB 架构检查当前 `Movie` 模型，并创建必要的代码，将 DB 迁移到新模型。

名称“Rating”是任意的，用于对迁移文件进行命名。为迁移文件使用有意义的名称是有帮助的。

如果删除 DB 中的所有记录，初始化方法会设定 DB 种子，并将包括 `Rating` 字段。

运行应用，并验证是否可以创建/编辑/显示具有 `Rating` 字段的电影。应向 `Edit`、`Details` 和 `Delete` 视图模板添加 `Rating` 字段。

上一页 下一页

将验证添加到 ASP.NET Core MVC 应用

2019/1/10 • • Edit Online

作者：Rick Anderson

本节内容：

- 向 `Movie` 模型添加了验证逻辑。
- 确保每当用户创建或编辑电影时，都会强制执行验证规则。

坚持 DRY 原则

MVC 的设计原则之一是 DRY（“不要自我重复”）。ASP.NET Core MVC 支持你仅指定一次功能或行为，然后使它应用到整个应用中。这可以减少所需编写的代码量，并使编写的代码更少出错，更易于测试和维护。

MVC 和 Entity Framework Core Code First 提供的验证支持是 DRY 原则在实际操作中的极佳示例。可以在一个位置（模型类中）以声明方式指定验证规则，并且在应用中的所有位置强制执行。

将验证规则添加到电影模型

打开 Movie.cs 文件。DataAnnotations 提供一组内置验证特性，可通过声明方式应用于任何类或属性。（它还包含 `DataType` 等格式特性，这些特性可帮助进行格式设置，但不提供任何验证。）

更新 `Movie` 类以使用内置的 `Required`、`StringLength`、`RegularExpression` 和 `Range` 验证特性。

```
public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

验证特性指定要对应用这些特性的模型属性强制执行的行为：

- `Required` 和 `MinimumLength` 特性表示属性必须有值；但用户可输入空格来满足此验证。

- `RegularExpression` 特性用于限制可输入的字符。在上述代码中，`Genre` 和 `Rating` 必须使用纯字母（禁用首字母大写、空格、数字和特殊字符）。
- `Range` 特性将值限制在指定范围内。
- `StringLength` 特性使你能够设置字符串属性的最大长度，以及可选的最小长度。
- 从本质上来说，需要值类型（如 `decimal`、`int`、`float`、`DateTime`），但不需要 `[Required]` 特性。

让 ASP.NET Core 强制自动执行验证规则有助于提升你的应用的可靠性。同时它能确保你无法忘记验证某些内容，并防止你无意中将错误数据导入数据库。

MVC 中的验证错误 UI

运行应用并导航到电影控制器。

点击“新建”连接添加新电影的链接。使用无效值填写表单。当 jQuery 客户端验证检测到错误时，会显示一条错误消息。

NOTE

可能无法在小数字段中输入十进制逗号。若要使 jQuery 验证支持使用逗号(“,”)表示小数点的的非英语区域设置，以及支持非美国英语日期格式，必须执行使应用全球化的步骤。有关添加十进制逗号的说明，请参阅 GitHub 问题 4076。

请注意表单如何自动呈现每个包含无效值的字段中相应的验证错误消息。客户端(使用 JavaScript 和 jQuery)和服

务器端(若用户禁用 JavaScript)都必定会遇到这些错误。

明显的好处在于不需要在 `MoviesController` 类或 Create.cshtml 视图中更改单个代码行来启用此验证 UI。在本教程前面创建的控制器和视图会自动选取验证规则，这些规则是通过在 `Movie` 模型类的属性上使用验证特性所指定的。使用 `Edit` 操作方法测试验证后，即已应用相同的验证。

存在客户端验证错误时，不会将表单数据发送到服务器。可通过使用 Fiddler 工具或 F12 开发人员工具在 `HTTP Post` 方法中设置断点来对此进行验证。

验证工作原理

你可能想知道在不对控制器或视图中的代码进行任何更新的情况下，验证 UI 是如何生成的。下列代码显示两种 `Create` 方法。

```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

第一个 (HTTP GET) `Create` 操作方法显示初始的“创建”表单。第二个 (`[HttpPost]`) 版本处理表单发布。第二个 `Create` 方法(`[HttpPost]` 版本)调用 `ModelState.IsValid` 以检查电影是否有任何验证错误。调用此方法将评估已应用于对象的任何验证特性。如果对象有验证错误，则 `Create` 方法会重新显示此表单。如果没有错误，此方法则将新电影保存在数据库中。在我们的电影示例中，在检测到客户端上存在验证错误时，表单不会发布到服务器。当存在客户端验证错误时，第二个 `Create` 方法永远不会被调用。如果在浏览器中禁用 JavaScript，客户端验证将被禁用，而你可以测试 HTTP POST `Create` 方法 `ModelState.IsValid` 检测任何验证错误。

可以在 `[HttpPost] Create` 方法中设置断点，并验证方法从未被调用，客户端验证在检测到存在验证错误时不会提交表单数据。如果在浏览器中禁用 JavaScript，然后提交错误的表单，将触发断点。在没有 JavaScript 的情况下仍然可以进行完整的验证。

以下图片显示如何在 FireFox 浏览器中禁用 JavaScript。

以下图片显示如何在 Chrome 浏览器中禁用 JavaScript。

禁用 JavaScript 后，发布无效数据并单步执行调试程序。

```
74          // POST: Movies/Create
75          // To protect from overposting attacks, please enable t
76          // more details see http://go.microsoft.com/fwlink/?Lin
77          [HttpPost]
78          [ValidateAntiForgeryToken]
            0 references
79          public async Task<IAc[illegible]lt> Create([Bind("ID,Title
80          {                          false
81              if (ModelState.IsValid)
82              {
83                  _context.Add(movie);
84                  await _context.SaveChangesAsync();
85                  return RedirectToAction("Index");
86              }
87              return View(movie);
88          }
89
```

Create.cshtml 视图模板的一部分在以下标记中显示：

```
<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>

        @*Markup removed for brevity.*@
```

操作方法使用上述标记来显示初始表单，并在发生错误时重新显示此表单。

输入标记帮助程序使用 DataAnnotations 特性，并在客户端上生成 jQuery 验证所需的 HTML 特性。验证标记帮助程序用于显示验证错误。有关详细信息，请参阅验证。

此方法真正好的一点是：无论是控制器还是 `Create` 视图模板都不知道强制实施的实际验证规则或显示的特定错误消息。仅可在 `Movie` 类中指定验证规则和错误字符串。这些相同的验证规则自动应用于 `Edit` 视图和可能创建用于编辑模型的任何其他视图模板。

需要更改验证逻辑时，可以通过将验证特性添加到模型在同一个位置实现此操作。（在此示例中为 `Movie` 类）。无需担心对应用程序的不同部分所强制执行规则的方式不一致 - 所有验证逻辑都将定义在一个位置并用于整个应用程序。这使代码非常简洁，并且更易于维护和改进。这意味着对 DRY 原则的完全遵守。

使用 DataType 特性

打开 Movie.cs 文件并检查 `Movie` 类。除了一组内置的验证特性，`System.ComponentModel.DataAnnotations` 命名空间还提供格式特性。我们已经在发布日期和价格字段中应用了 `DataType` 枚举值。以下代码显示具有适当 `DataType` 特性的 `ReleaseDate` 和 `Price` 属性。

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

`DataType` 属性仅提供相关提示来帮助视图引擎设置数据格式(并提供元素/属性，例如向 URL 提供 `<a>` 和向电子邮件提供 `<a href="mailto:EmailAddress.com">`)。可以使用 `RegularExpression` 特性验证数据的格式。`DataType` 属性用于指定比数据库内部类型更具体的数据类型，它们不是验证属性。在此示例中，我们只想跟踪日期，而不是时间。`DataType` 枚举提供了多种数据类型，例如日期、时间、电话号码、货币、电子邮件地址等。应用程序还可通过 `DataType` 特性自动提供类型特定的功能。例如，可以为 `DataType.EmailAddress` 创建 `mailto:` 链接，并且可以在支持 HTML5 的浏览器中为 `DataType.Date` 提供日期选择器。`DataType` 特性发出 HTML 5 `data-`(读作 data dash)特性供 HTML 5 浏览器理解。`DataType` 特性不提供任何验证。

`DataType.Date` 不指定显示日期的格式。默认情况下，数据字段根据基于服务器的 `CultureInfo` 的默认格式进行显示。

`DisplayFormat` 特性用于显式指定日期格式：

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

`ApplyFormatInEditMode` 设置指定在文本框中显示值以进行编辑时也应用格式。(你可能不想为某些字段执行此操作 — 例如对于货币值，你可能不希望文本框中的货币符号可编辑。)

可以单独使用 `DisplayFormat` 特性，但通常建议使用 `DataType` 特性。`DataType` 特性传达数据的语义而不是传达如何在屏幕上呈现数据，并提供 DisplayFormat 不具备的以下优势：

- 浏览器可启用 HTML5 功能(例如显示日历控件、区域设置适用的货币符号、电子邮件链接等)
- 默认情况下，浏览器将根据区域设置采用正确的格式呈现数据。
- `DataType` 特性使 MVC 能够选择正确的字段模板来呈现数据(如果 `DisplayFormat` 由自身使用，则使用的是字符串模板)。

> **NOTE**
> jQuery 验证不适用于 `Range` 属性和 `DateTime`。例如，以下代码将始终显示客户端验证错误，即便日期在指定的范围内：
>
> `[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]`

需要禁用 jQuery 日期验证才能使用具有 `DateTime` 的 `Range` 特性。通常，在模型中编译固定日期是不恰当的，因此不推荐使用 `Range` 特性和 `DateTime`。

以下代码显示组合在一行上的特性：

```
public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

在本系列的下一部分中，我们将回顾应用，并对自动生成的 `Details` 和 `Delete` 方法进行一些改进。

其他资源

- 使用表单
- 全球化和本地化
- 标记帮助程序简介
- 创作标记帮助程序

上一页 下一页

检查 ASP.NET Core 应用的 Details 和 Delete 方法

2019/1/10 • • Edit Online

作者：Rick Anderson

打开电影控制器，并检查 `Details` 方法：

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

创建此操作方法的 MVC 基架引擎添加显示调用方法的 HTTP 请求的注释。在此情况下，它是包含三个 URL 段的 GET 请求，这三个段为 `Movies` 控制器、`Details` 方法和 `id` 值。回顾这些在 Startup.cs 中定义的段。

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

EF 可以使用 `SingleOrDefaultAsync` 方法轻松搜索数据。该方法中内置的一个重要安全功能是，代码会先验证搜索方法已经找到电影，然后再执行操作。例如，一个黑客可能通过将链接创建的 URL 从 `http://localhost:xxxx/Movies/Details/1` 更改为类似 `http://localhost:xxxx/Movies/Details/12345` 的值（或者不代表任何实际电影的其他值）将错误引入站点。如果未检查是否有空电影，则应用可能引发异常。

检查 `Delete` 和 `DeleteConfirmed` 方法。

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.FindAsync(id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

请注意，`HTTP GET Delete` 方法不删除指定的电影，而是返回可在其中提交 (HttpPost) 删除的电影视图。执行删除操作以响应 GET 请求（或者说，执行编辑操作、创建操作或更改数据的任何其他操作）会打开安全漏洞。

删除数据的 `[HttpPost]` 方法命名为 `DeleteConfirmed`，以便为 HTTP POST 方法提供一个唯一的签名或名称。下面显示了两个方法签名：

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
```

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

公共语言运行时 (CLR) 需要重载方法拥有唯一的参数签名（相同的方法名称但不同的参数列表）。但是，这里需要两个 `Delete` 方法 -- 一个用于 GET，另一个用于 POST -- 这两个方法拥有相同的参数签名。（它们都需要接受单个整数作为参数。）

可通过两种方法解决此问题，一种是为方法提供不同的名称。这正是前面的示例中的基架机制进行的操作。但是，这会造成一个小问题：ASP.NET 按名称将 URL 段映射到操作方法，如果重命名方法，则路由通常无法找到该方法。该示例中也提供了解决方案，即向 `DeleteConfirmed` 方法添加 `ActionName("Delete")` 属性。该属性对路由系统执行映射，以便包括 POST 请求的 /Delete/ 的 URL可找到 `DeleteConfirmed` 方法。

对于名称和签名相同的方法，另一个常用解决方法是手动更改 POST 方法的签名以包括额外（未使用）的参数。这正是前面的文章中添加 `notUsed` 参数时进行的操作。这里为了 `[HttpPost] Delete` 方法可以执行同样的操作：

```
// POST: Movies/Delete/6
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

发布到 Azure

若要了解如何部署到 Azure，请参阅教程：在 Azure 应用服务中生成 .NET Core 和 SQL 数据库 Web 应用。

上一篇

ASP.NET Core 中的 Razor 页面的筛选方法

2018/10/30 • • Edit Online

作者：Rick Anderson

Razor 页面筛选器 IPageFilter 和 IAsyncPageFilter 允许 Razor 页面在运行 Razor 页面处理程序的前后运行代码。Razor 页面筛选器与 ASP.NET Core MVC 操作筛选器类似，但它们不能应用于单个页面处理程序方法。

Razor 页面筛选器：

- 在选择处理程序方法后但在模型绑定发生前运行代码。
- 在模型绑定完成后，执行处理程序方法前运行代码。
- 在执行处理程序方法后运行代码。
- 可在页面或全局范围内实现。
- 无法应用于特定的页面处理程序方法。

代码可在使用页面构造函数或中间件执行处理程序方法前运行，但仅 Razor 页面筛选器可访问 HttpContext。筛选器具有 FilterContext 派生参数，可用于访问 `HttpContext`。例如，实现筛选器属性示例将标头添加到响应中，而使用构造函数或中间件则无法做到这点。

查看或下载示例代码（如何下载）

Razor 页面筛选器提供的以下方法可在全局或页面级应用：

- 同步方法：
  - OnPageHandlerSelected：在选择处理程序方法后但在模型绑定发生前调用。
  - OnPageHandlerExecuting：在执行处理器方法前，模型绑定完成后调用。
  - OnPageHandlerExecuted：在执行处理器方法后，生成操作结果前调用。
- 异步方法：
  - OnPageHandlerSelectionAsync：在选择处理程序方法后，但在模型绑定发生前，进行异步调用。
  - OnPageHandlerExecutionAsync：在调用处理程序方法前，但在模型绑定结束后，进行异步调用。

> **NOTE**
>
> 筛选器接口的同步和异步版本任意实现一个，而不是同时实现。该框架会先查看筛选器是否实现了异步接口，如果是，则调用该接口。如果不是，则调用同步接口的方法。如果两个接口都已实现，则只会调用异步方法。对页面中的替代应用相同的规则，同步替代或异步替代只能任选其一实现，不可二者皆选。

全局实现 Razor 页面筛选器

以下代码实现了 `IAsyncPageFilter`：

```
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace PageFilter.Filters
{
    public class SampleAsyncPageFilter : IAsyncPageFilter
    {
        private readonly ILogger _logger;

        public SampleAsyncPageFilter(ILogger logger)
        {
            _logger = logger;
        }

        public async Task OnPageHandlerSelectionAsync(
                                        PageHandlerSelectedContext context)
        {
            _logger.LogDebug("Global OnPageHandlerSelectionAsync called.");
            await Task.CompletedTask;
        }

        public async Task OnPageHandlerExecutionAsync(
                                        PageHandlerExecutingContext context,
                                        PageHandlerExecutionDelegate next)
        {
            _logger.LogDebug("Global OnPageHandlerExecutionAsync called.");
            await next.Invoke();
        }
    }
}
```

在前面的代码中，ILogger 不是必需的。它在示例中用于提供应用程序的跟踪信息。

以下代码启用 `Startup` 类中的 `SampleAsyncPageFilter`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new SampleAsyncPageFilter(_logger));
    });
}
```

以下代码显示完整的 `Startup` 类：

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using PageFilter.Filters;

namespace PageFilter
{
    public class Startup
    {
        ILogger _logger;
        public Startup(ILoggerFactory loggerFactory, IConfiguration configuration)
        {
            _logger = loggerFactory.CreateLogger<GlobalFiltersLogger>();
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc(options =>
            {
                options.Filters.Add(new SampleAsyncPageFilter(_logger));
            });
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();

            app.UseMvc();
        }
    }
}
```

以下代码调用 `AddFolderApplicationModelConvention` 将 `SampleAsyncPageFilter` 仅应用于 /subFolder 中的页面：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
       .AddRazorPagesOptions(options =>
       {
           options.Conventions.AddFolderApplicationModelConvention(
               "/subFolder",
               model => model.Filters.Add(new SampleAsyncPageFilter(_logger)));
       });
}
```

以下代码实现同步的 `IPageFilter`：

```
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Logging;

namespace PageFilter.Filters
{
    public class SamplePageFilter : IPageFilter
    {
        private readonly ILogger _logger;

        public SamplePageFilter(ILogger logger)
        {
            _logger = logger;
        }

        public void OnPageHandlerSelected(PageHandlerSelectedContext context)
        {
            _logger.LogDebug("Global sync OnPageHandlerSelected called.");
        }

        public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
        {
            _logger.LogDebug("Global sync PageHandlerExecutingContext called.");
        }

        public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
        {
            _logger.LogDebug("Global sync OnPageHandlerExecuted called.");
        }
    }
}
```

以下代码启用 `SamplePageFilter`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new SamplePageFilter(_logger));
    });
}
```

通过重写筛选器方法实现 Razor 页面筛选器

以下代码替代同步的 Razor 页面筛选器：

```
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;

namespace PageFilter.Pages
{
    public class IndexModel : PageModel
    {
        private readonly ILogger _logger;

        public IndexModel(ILogger<IndexModel> logger)
        {
            _logger = logger;
        }
        public string Message { get; set; }

        public void OnGet()
        {
            _logger.LogDebug("IndexModel/OnGet");
        }

        public override void OnPageHandlerSelected(
                                PageHandlerSelectedContext context)
        {
            _logger.LogDebug("IndexModel/OnPageHandlerSelected");
        }

        public override void OnPageHandlerExecuting(
                                PageHandlerExecutingContext context)
        {
            Message = "Message set in handler executing";
            _logger.LogDebug("IndexModel/OnPageHandlerExecuting");
        }


        public override void OnPageHandlerExecuted(
                                PageHandlerExecutedContext context)
        {
            _logger.LogDebug("IndexModel/OnPageHandlerExecuted");
        }
    }
}
```

实现筛选器属性

基于属性的内置筛选器 OnResultExecutionAsync 可以进行子类化。以下筛选器向响应添加标头：

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace PageFilter.Filters
{
    public class AddHeaderAttribute  : ResultFilterAttribute
    {
        private readonly string _name;
        private readonly string _value;

        public AddHeaderAttribute (string name, string value)
        {
            _name = name;
            _value = value;
        }

        public override void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(_name, new string[] { _value });
        }
    }
}
```

以下代码应用 `AddHeader` 属性：

```
[AddHeader("Author", "Rick")]
public class ContactModel : PageModel
{
    private readonly ILogger _logger;

    public ContactModel(ILogger<ContactModel> logger)
    {
        _logger = logger;
    }
    public string Message { get; set; }

    public async Task OnGetAsync()
    {
        Message = "Your contact page.";
        _logger.LogDebug("Contact/OnGet");
        await Task.CompletedTask;
    }
}
```

有关重写顺序的说明，请参阅重写默认顺序。

有关将筛选器管道与筛选器短路的说明，请参阅取消和设置短路。

授权筛选器属性

Authorize 属性可应用于 `PageModel`：

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace PageFilter.Pages
{
    [Authorize]
    public class ModelWithAuthFilterModel : PageModel
    {
        public IActionResult OnGet() => Page();
    }
}
```

创建使用 ASP.NET Core 中 Razor 类库项目的可重用 UI

2018/12/10 • • Edit Online

作者：Rick Anderson

Razor 视图、页面、控制器、页面模型、视图组件和数据模型可以构建到 Razor 类库 (RCL) 中。RCL 可以打包并重复使用。应用程序可以包括 RCL，并重写其中包含的视图和页面。如果在 Web 应用和 RCL 中都能找到视图、分部视图或 Razor 页面，则 Web 应用中的 Razor 标记（.cshtml 文件）优先。

此功能要求 .NET Core 2.1 SDK 或更高版本

查看或下载示例代码（如何下载）

创建一个包含 Razor UI 的类库

- Visual Studio
- .NET Core CLI

- 从 Visual Studio“文件”菜单中选择“新建” > “项目”。
- 选择“ASP.NET Core Web 应用程序”。
- 命名库（例如，“RazorClassLib”）>“确定”。为避免与已生成的视图库发生文件名冲突，请确保库名称不以 `.Views` 结尾。
- 验证是否已选择 ASP.NET Core 2.1 或更高版本。
- 选择“Razor 类库” > “确定”。

一个 Razor 类库具有以下项目文件：

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.1.2" />
  </ItemGroup>

</Project>
```

将 Razor 文件添加到 RCL。

ASP.NET Core 模板假定 RCL 内容位于*领域*文件夹。请参阅RCL 页面布局若要创建中的内容公开 RCL `~/Pages` 而非 `~/Areas/Pages`。

引用 Razor 类库内容

可以通过以下方式引用 RCL：

- NuGet 包。请参阅创建 NuGet 包、dotnet 添加包和创建和发布 NuGet 包。
- {ProjectName}.csproj。请查看 dotnet-add 引用。

演练：创建 Razor 类库项目并从 Razor 页面项目使用它

可以下载并测试完整项目，无需创建项目。示例下载包含附加代码和链接，以方便测试项目。可以在此 GitHub 问题中留下反馈，评论下载示例和分步说明的对比。

测试下载应用

如果尚未下载已完成的应用，并更愿意创建演练项目，请跳转至下一节。

- Visual Studio
- .NET Core CLI

在 Visual Studio 中打开 .sln 文件。运行应用。

按"测试 Test WebApp1"中的说明进行操作

创建 Razor 类库

本部分创建 Razor 类库 (RCL)。将 Razor 文件添加到 RCL。

- Visual Studio
- .NET Core CLI

创建 RCL 项目：

- 从 Visual Studio"文件"菜单中选择"新建" > "项目"。
- 选择"ASP.NET Core Web 应用程序"。
- 命名应用程序**RazorUIClassLib** > 确定。
- 验证是否已选择 ASP.NET Core 2.1 或更高版本。
- 选择"Razor 类库" > "确定"。
- 添加一个名为 RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml 的 Razor 分部视图文件。

将 Razor 文件和文件夹添加到项目

- 使用以下代码替换 RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml 中的标记：

```
<h3>_Message.cshtml partial view.</h3>

<p>RazorUIClassLib\Areas\MyFeature\Pages\Shared\_Message.cshtml</p>
```

- 使用以下代码替换 RazorUIClassLib/Areas/MyFeature/Pages/Page1.cshtml 中的标记：

```
@page
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h2>Hello from a Razor UI class library!</h2>
<p> From  RazorUIClassLib\Areas\MyFeature\Pages\Page1.cshtml</p>

<partial name="_Message" />
```

使用分步视图（`<partial name="_Message" />`）需要 `@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers`。可以添加一个 _ViewImports.cshtml 文件，无需包含 `@addTagHelper` 指令。例如：

```
dotnet new viewimports -o RazorUIClassLib/Areas/MyFeature/Pages
```

有关详细信息 *_ViewImports.cshtml*，请参阅导入共享指令

- 生成类库以验证是否不存在编译器错误：

```
dotnet build RazorUIClassLib
```

生成输出内容包含 RazorUIClassLib.dll 和 RazorUIClassLib.Views.dll。RazorUIClassLib.Views.dll 包含已编译的 Razor 内容。

从 Razor 页面项目使用 Razor UI 库

- Visual Studio
- .NET Core CLI

创建 Razor 页面 Web 应用：

- 在解决方案资源管理器中，右键单击解决方案 >“添加” > “新建项目”。
- 选择“ASP.NET Core Web 应用程序”。
- 将应用命名为 WebApp1。
- 验证是否已选择 ASP.NET Core 2.1 或更高版本。
- 选择“Web 应用程序” > “确定”。
- 在解决方案资源管理器中，右键单击“WebApp1”，然后选择“设为启动项目”。
- 在解决方案资源管理器中，右键单击“WebApp1”，然后选择“生成依赖项” > “项目依赖项”。
- 将 RazorUIClassLib 勾选为 WebApp1 的依赖项。
- 在解决方案资源管理器中，右键单击“WebApp1”，选择“添加” > “引用”。
- 在“引用管理器”对话框中勾选“RazorUIClassLib” > “确定”。

运行应用。

测试 WebApp1

验证正在使用的 Razor UI 类库。

- 浏览到 `/MyFeature/Page1`。

重写视图、分部视图和页面

如果在 Web 应用和 Razor 类库中都能找到视图、分部视图或 Razor 页面，则 Web 应用中的 Razor 标记（.cshtml 文件）优先。例如，添加*WebApp1/Areas/MyFeature/Pages/Page1.cshtml*到 WebApp1，并在 WebApp1 Page1 将优先于 Razor 类库中的 Page1。

在示例下载中，将 WebApp1/Areas/MyFeature2 重命名为 WebApp1/Areas/MyFeature 来测试优先级。

将 RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml 分部视图复制到 WebApp1/Areas/MyFeature/Pages/Shared/_Message.cshtml。更新标记以指示新的位置。生成并运行应用，验证使用部分的应用版本。

RCL 页面布局

为引用 RCL 内容就好像它是 web 应用的一部分*页面*文件夹中，创建具有以下文件结构 RCL 项目：

- *RazorUIClassLib/页*
- *RazorUIClassLib/页/Shared*

假设*RazorUIClassLib/页/共享*包含两个部分的文件：*_Header.cshtml*并 *_Footer.cshtml*。`<partial>`标记无法添加

到 *_Layout.cshtml*文件：

```
<body>
  <partial name="_Header">
  @RenderBody()
  <partial name="_Footer">
</body>
```

ASP.NET Core 中 Razor 页面的路由和应用约定

2018/10/30 • • Edit Online

作者：Luke Latham

了解如何使用页面路由和应用模型提供程序约定来控制 Razor 页面应用中的页面路由、发现和处理。

需要为各个页面配置自定义页面路由时，可使用本主题稍后所述的 AddPageRoute 约定配置页面路由。

若要指定页面路由、添加路由段，或将参数添加到路由，请使用页面的 `@page` 指令。有关详细信息，请参阅自定义路由。

有不能用作路由段或参数名称的保留的字。有关详细信息，请参阅路由：保留路由名称。

查看或下载示例代码（如何下载）

方案	示例演示...
模型约定 Conventions.Add • IPageRouteModelConvention • IPageApplicationModelConvention	将路由模板和标头添加到应用的页面。
页面路由操作约定 • AddFolderRouteModelConvention • AddPageRouteModelConvention • AddPageRoute	将路由模板添加到某个文件夹中的页面以及单个页面。
页面模型操作约定 • AddFolderApplicationModelConvention • AddPageApplicationModelConvention • ConfigureFilter（筛选器类、Lambda 表达式或筛选器工厂）	将标头添加到某个文件夹中的多个页面，将标头添加到单个页面，以及配置筛选器工厂以将标头添加到应用的页面。
默认页面应用模型提供程序	取代默认页面模型提供程序，用于更改处理程序命名约定。

方案	示例演示...
模型约定 Conventions.Add • IPageRouteModelConvention • IPageApplicationModelConvention • IPageHandlerModelConvention	将路由模板和标头添加到应用的页面。
页面路由操作约定 • AddFolderRouteModelConvention • AddPageRouteModelConvention • AddPageRoute	将路由模板添加到某个文件夹中的页面以及单个页面。

方案	示例演示...
页面模型操作约定 • AddFolderApplicationModelConvention • AddPageApplicationModelConvention • ConfigureFilter(筛选器类、Lambda 表达式或筛选器工厂)	将标头添加到某个文件夹中的多个页面，将标头添加到单个页面，以及配置筛选器工厂以将标头添加到应用的页面。
默认页面应用模型提供程序	取代默认页面模型提供程序，用于更改处理程序命名约定。

使用 AddRazorPagesOptions 扩展方法向 `Startup` 类中服务集合的 AddMvc 添加和配置 Razor 页面约定。本主题稍后会介绍以下约定示例：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
            {
                options.Conventions.Add( ... );
                options.Conventions.AddFolderRouteModelConvention("/OtherPages", model => { ... });
                options.Conventions.AddPageRouteModelConvention("/About", model => { ... });
                options.Conventions.AddPageRoute("/Contact", "TheContactPage/{text?}");
                options.Conventions.AddFolderApplicationModelConvention("/OtherPages", model => { ... });
                options.Conventions.AddPageApplicationModelConvention("/About", model => { ... });
                options.Conventions.ConfigureFilter(model => { ... });
                options.Conventions.ConfigureFilter( ... );
            });
}
```

路由顺序

路由指定Order用于处理（路由匹配）。

顺序	行为
-1	处理其他路由之前处理该路由。
0	未指定顺序（默认值）。不分配 `Order`（`Order = null`）将默认路由 `Order` 为 0（零）进行处理。
1、2、... n	指定路由处理顺序。

按照约定来建立路由处理：

- 按顺序处理的路由 (-1、0、1、2、... n)。
- 当路由具有相同 `Order`、最具体的路由匹配跟有更具体的路由。
- 当具有相同的路由 `Order` 和相同数量的参数匹配请求 URL，将它们添加到顺序处理路由PageConventionCollection。

如果可能，避免具体取决于已建立的路由处理顺序。通常情况下，路由选择与 URL 匹配的正确路由。如果必须设置路由 `Order` 属性路由请求是否正确，应用程序的路由方案，可能令人困惑，向客户端和脆弱来维护。查找以简化应用程序的路由方案。示例应用程序需要处理订单，若要演示几个路由方案使用的单个应用程序的显式路由。但是，您应尝试避免设置路由的做法 `Order` 生产应用中。

Razor Pages 路由和 MVC 控制器路由共享一个实现。路由顺序 MVC 主题中的信息位于路由到控制器操作：对属性路由排序。

模型约定

为 IPageConvention 添加委托，以添加应用于 Razor 页面的模型约定。

将路由模型约定添加到所有页面

使用约定创建 IPageRouteModelConvention 并将其添加到 IPageConvention 实例集合中，这些实例将在页面路由模型构造过程中应用。

示例应用将 `{globalTemplate?}` 路由模板添加到应用中的所有页面：

```
public class GlobalTemplatePageRouteModelConvention
    : IPageRouteModelConvention
{
    public void Apply(PageRouteModel model)
    {
        var selectorCount = model.Selectors.Count;
        for (var i = 0; i < selectorCount; i++)
        {
            var selector = model.Selectors[i];
            model.Selectors.Add(new SelectorModel
            {
                AttributeRouteModel = new AttributeRouteModel
                {
                    Order = 1,
                    Template = AttributeRouteModel.CombineTemplates(
                        selector.AttributeRouteModel.Template,
                        "{globalTemplate?}"),
                }
            });
        }
    }
}
```

AttributeRouteModel 的 Order 属性设置为 `1`。这可确保匹配行为示例应用程序中的以下路由：

- 路由模板 `TheContactPage/{text?}` 添加本主题后面。联系人页面路由的默认顺序 `null`（`Order = 0`），因此它匹配之前 `{globalTemplate?}` 路由模板。
- `{aboutTemplate?}` 路由模板添加本主题后面。为 `{aboutTemplate?}` 模板指定的 `Order` 为 `2`。当在 `/About/RouteDataValue` 中请求"关于"页面时，由于设置了 `Order` 属性，"RouteDataValue"会加载到 `RouteData.Values["globalTemplate"]`（`Order = 1`）而不是 `RouteData.Values["aboutTemplate"]`（`Order = 2`）中。
- `{otherPagesTemplate?}` 路由模板添加本主题后面。为 `{otherPagesTemplate?}` 模板指定的 `Order` 为 `2`。中的任何页时*Pages/OtherPages*文件夹请求与路由参数（例如，`/OtherPages/Page1/RouteDataValue`），"RouteDataValue"加载到 `RouteData.Values["globalTemplate"]`（`Order = 1`），而不 `RouteData.Values["otherPagesTemplate"]`（`Order = 2`）由于设置 `Order` 属性。

如有可能，未设置 `Order`，这会导致 `Order = 0`。依赖于路由，以选择正确的路由。

将 MVC 添加到 `Startup.ConfigureServices` 中的服务集合时，会添加 Razor 页面选项，例如添加约定。有关示例，请参阅示例应用。

```
options.Conventions.Add(new GlobalTemplatePageRouteModelConvention());
```

在 `localhost:5000/About/GlobalRouteValue` 中请求示例的"关于"页面并检查结果：

将应用模型约定添加到所有页面

使用约定创建 IPageApplicationModelConvention 并将其添加到 IPageConvention 实例集合中，这些实例将在页面应用模型构造过程中应用。

为了演示此约定以及本主题后面的其他约定，示例应用包含了一个 `AddHeaderAttribute` 类。类构造函数采用 `name` 字符串和 `values` 字符串数组。将在其 `OnResultExecuting` 方法中使用这些值来设置响应标头。本主题后面的页面模型操作约定部分展示了完整的类。

示例应用使用 `AddHeaderAttribute` 类将标头 `GlobalHeader` 添加到应用中的所有页面：

```
public class GlobalHeaderPageApplicationModelConvention
    : IPageApplicationModelConvention
{
    public void Apply(PageApplicationModel model)
    {
        model.Filters.Add(new AddHeaderAttribute(
            "GlobalHeader", new string[] { "Global Header Value" }));
    }
}
```

Startup.cs：

```
options.Conventions.Add(new GlobalHeaderPageApplicationModelConvention());
```

在 `localhost:5000/About` 中请求示例的“关于”页面，并检查标头以查看结果：

将处理程序模型约定添加到的所有页面

使用约定创建 IPageHandlerModelConvention 并将其添加到 IPageConvention 实例集合中，这些实例将在页面处理程序模型构造过程中应用。

```
public class GlobalPageHandlerModelConvention
    : IPageHandlerModelConvention
{
    public void Apply(PageHandlerModel model)
    {
        ...
    }
}
```

`Startup.ConfigureServices`：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
        {
            options.Conventions.Add(new GlobalPageHandlerModelConvention());
        });
```

页面路由操作约定

默认路由模型提供程序派生自 IPageRouteModelProvider，可调用旨在为页面路由配置提供扩展点的约定。

文件夹路由模型约定

使用 AddFolderRouteModelConvention 创建并添加 IPageRouteModelConvention，后者可以为指定文件夹下的所有页面调用 PageRouteModel 上的操作。

示例应用使用 `AddFolderRouteModelConvention` 将 `{otherPagesTemplate?}` 路由模板添加到 *OtherPages* 文件夹中的页面：

```
options.Conventions.AddFolderRouteModelConvention("/OtherPages", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel.Template,
                    "{otherPagesTemplate?}"),
            }
        });
    }
});
```

AttributeRouteModel 的 Order 属性设置为 `2`。这可以保证的模板 `{globalTemplate?}`（设置到主题中前面 `1`）提供单个路由值时，第一个路由数据值位置分配优先级。如果在页*Pages/OtherPages*文件夹请求与路由参数值（例如，`/OtherPages/Page1/RouteDataValue`），"RouteDataValue"加载到 `RouteData.Values["globalTemplate"]`（`Order = 1`），而不 `RouteData.Values["otherPagesTemplate"]`（`Order = 2`）由于设置 `Order` 属性。

如有可能，未设置 `Order`，这会导致 `Order = 0`。依赖于路由，以选择正确的路由。

在 `localhost:5000/OtherPages/Page1/GlobalRouteValue/OtherPagesRouteValue` 中请求示例的 Page1 页面并检查结果：

页面路由模型约定

使用 AddPageRouteModelConvention 创建并添加 IPageRouteModelConvention，后者可以为具有指定名称的页面调用 PageRouteModel 上的操作。

示例应用使用 `AddPageRouteModelConvention` 将 `{aboutTemplate?}` 路由模板添加到"关于"页面：

```
options.Conventions.AddPageRouteModelConvention("/About", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel.Template,
                    "{aboutTemplate?}"),
            }
        });
    }
});
```

AttributeRouteModel 的 Order 属性设置为 `2`。这可以保证的模板 `{globalTemplate?}`（设置到主题中前面 `1`）提供单个路由值时，第一个路由数据值位置分配优先级。如果使用路由参数值在请求关于页面 `/About/RouteDataValue`，"RouteDataValue"加载到 `RouteData.Values["globalTemplate"]`（`Order = 1`），而不 `RouteData.Values["aboutTemplate"]`（`Order = 2`）因设置而 `Order` 属性。

如有可能，未设置 `Order`，这会导致 `Order = 0`。依赖于路由，以选择正确的路由。

在 `localhost:5000/About/GlobalRouteValue/AboutRouteValue` 中请求示例的"关于"页面并检查结果：

参数转换器用于自定义页面路由

使用参数转换器可以自定义页面路由生成的 ASP.NET Core。参数转换程序实现 `IOutboundParameterTransformer` 并转换参数值。例如，一个自定义 `SlugifyParameterTransformer` 参数转换程序可将 `SubscriptionManagement` 路由值更改为 `subscription-management`。

`PageRouteTransformerConvention` 页面路由模型约定适用于在应用中自动生成的页路由文件夹和文件名称段参数转换器。例如，Razor 页面文件位于 */Pages/SubscriptionManagement/ViewAll.cshtml*必须重写中其路由 `/SubscriptionManagement/ViewAll` 到 `/subscription-management/view-all`。

`PageRouteTransformerConvention` 仅将转换的页面路由来自 Razor 页面文件夹和文件名称自动生成的段。它不会转换与添加的路由段 `@page` 指令。约定也不会转换添加的路由AddPageRoute。

`PageRouteTransformerConvention` 注册中的一个选项为 `Startup.ConfigureServices`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
            {
                options.Conventions.Add(
                    new PageRouteTransformerConvention(
                        new SlugifyParameterTransformer()));
            });
}

public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string TransformOutbound(object value)
    {
        if (value == null) { return null; }

        // Slugify value
        return Regex.Replace(value.ToString(), "([a-z])([A-Z])", "$1-$2").ToLower();
    }
}
```

配置页面路由

使用 AddPageRoute 配置路由，该路由指向指定页面路径中的页面。生成的页面链接使用指定的路由。

`AddPageRoute` 使用 `AddPageRouteModelConvention` 建立路由。

示例应用为 *Contact.cshtml* 创建指向 `/TheContactPage` 的路由：

```
options.Conventions.AddPageRoute("/Contact", "TheContactPage/{text?}");
```

还可在 `/Contact` 中通过默认路由访问“联系人”页面。

示例应用的“联系人”页面自定义路由允许使用可选的 `text` 路由段（`{text?}`）。该页面还在其 `@page` 指令中包含此可选段，以便访问者在 `/Contact` 路由中访问该页面：

```
@page "{text?}"
@model ContactModel
@{
    ViewData["Title"] = "Contact";
}

<h1>@ViewData["Title"]</h1>
<h2>@Model.Message</h2>

<address>
    One Microsoft Way<br>
    Redmond, WA 98052-6399<br>
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong> <a href="mailto:Support@example.com">Support@example.com</a><br>
    <strong>Marketing:</strong> <a href="mailto:Marketing@example.com">Marketing@example.com</a>
</address>

<p>@Model.RouteDataTextTemplateValue</p>
```

请注意，在呈现的页面中，为联系人链接生成的 URL 反映了已更新的路由：

在常规路由 `/Contact` 或自定义路由 `/TheContactPage` 中访问“联系人”页面。如果提供附加的 `text` 路由段，该页面会显示所提供的 HTML 编码段：

页面模型操作约定

实现 IPageApplicationModelProvider 的默认页面模型提供程序可调用约定，这些约定旨在为页面模型配置提供扩展点。在生成和修改页面发现及处理方案时，可使用这些约定。

对于此部分中的示例，示例应用使用 `AddHeaderAttribute` 类（一个 ResultFilterAttribute）来应用响应标头：

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string _name;
    private readonly string[] _values;

    public AddHeaderAttribute(string name, string[] values)
    {
        _name = name;
        _values = values;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_name, _values);
        base.OnResultExecuting(context);
    }
}
```

示例演示了如何使用约定将该属性应用于某个文件夹中的所有页面以及单个页面。

文件夹应用模型约定

使用 AddFolderApplicationModelConvention 创建并添加 IPageApplicationModelConvention，后者可以为指定文件夹下的所有页面调用 PageApplicationModel 实例上的操作。

示例演示了如何使用 `AddFolderApplicationModelConvention` 将标头 `OtherPagesHeader` 添加到应用的 *OtherPages* 文件夹内的页面：

```
options.Conventions.AddFolderApplicationModelConvention("/OtherPages", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "OtherPagesHeader", new string[] { "OtherPages Header Value" }));
});
```

在 `localhost:5000/OtherPages/Page1` 中请求示例的 Page1 页面，并检查标头以查看结果：

```
▼ Response Headers    view source
  Content-Type: text/html; charset=utf-8
  Date: Sat, 21 Oct 2017 06:13:16 GMT
  FilterFactoryHeader: Filter Factory Header Value 1
  FilterFactoryHeader: Filter Factory Header Value 2
  GlobalHeader: Global Header Value
  OtherPagesHeader: OtherPages Header Value
  Server: Kestrel
  Transfer-Encoding: chunked
```

页面应用模型约定

使用AddPageApplicationModelConvention创建并添加IPageApplicationModelConvention，它在调用操作PageApplicationModel页使用指定的名称。

示例演示了如何使用 `AddPageApplicationModelConvention` 将标头 `AboutHeader` 添加到"关于"页面：

```
options.Conventions.AddPageApplicationModelConvention("/About", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "AboutHeader", new string[] { "About Header Value" }));
});
```

在 `localhost:5000/About` 中请求示例的"关于"页面，并检查标头以查看结果：

```
▼ Response Headers    view source
  AboutHeader: About Header Value
  Content-Type: text/html; charset=utf-8
  Date: Thu, 19 Oct 2017 21:09:07 GMT
  FilterFactoryHeader: Filter Factory Header Value 1
  FilterFactoryHeader: Filter Factory Header Value 2
  GlobalHeader: Global Header Value
  Server: Kestrel
  Transfer-Encoding: chunked
```

配置筛选器

ConfigureFilter 可配置要应用的指定筛选器。用户可以实现筛选器类，但示例应用演示了如何在 Lambda 表达式中实现筛选器，该筛选器在后台作为可返回筛选器的工厂实现：

```
options.Conventions.ConfigureFilter(model =>
{
    if (model.RelativePath.Contains("OtherPages/Page2"))
    {
        return new AddHeaderAttribute(
            "OtherPagesPage2Header",
            new string[] { "OtherPages/Page2 Header Value" });
    }
    return new EmptyFilter();
});
```

页面应用模型用于检查指向 *OtherPages* 文件夹中 Page2 页面的段的相对路径。如果条件通过，则添加标头。如果不通过，则应用 `EmptyFilter`。

`EmptyFilter` 是一种操作筛选器。由于 Razor 页面会忽略操作筛选器，因此，如果路径不包含 `OtherPages/Page2`，`EmptyFilter` 会按预期发出空操作指令。

在 `localhost:5000/OtherPages/Page2` 中请求示例的 Page2 页面，并检查标头以查看结果：

```
▼ Response Headers      view source
    Content-Type: text/html; charset=utf-8
    Date: Mon, 23 Oct 2017 06:06:52 GMT
    FilterFactoryHeader: Filter Factory Header Value 1
    FilterFactoryHeader: Filter Factory Header Value 2
    GlobalHeader: Global Header Value
    OtherPagesHeader: OtherPages Header Value
    OtherPagesPage2Header: OtherPages/Page2 Header Value
    Server: Kestrel
    Transfer-Encoding: chunked
```

配置筛选器工厂

ConfigureFilter 可配置指定的工厂，以将筛选器应用于所有 Razor 页面。

示例应用提供了一个示例，说明如何使用筛选器工厂将具有两个值的标头 `FilterFactoryHeader` 添加到应用的页面：

```
options.Conventions.ConfigureFilter(new AddHeaderWithFactory());
```

AddHeaderWithFactory.cs：

```
public class AddHeaderWithFactory : IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new AddHeaderFilter();
    }

    private class AddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "FilterFactoryHeader",
                new string[]
                {
                    "Filter Factory Header Value 1",
                    "Filter Factory Header Value 2"
                });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

在 `localhost:5000/About` 中请求示例的"关于"页面，并检查标头以查看结果：

替换默认页面应用模型提供程序

Razor 页面使用 `IPageApplicationModelProvider` 接口创建 DefaultPageApplicationModelProvider。用户可以从默认模型提供程序继承，以便为处理程序发现和处理提供自己的实现逻辑。默认实现（引用源）为*未命名*和*已命名*处理程序的命名建立了约定，如下所述。

默认的未命名处理程序方法

Http 谓词的处理程序方法（"未命名"的处理程序方法）遵循以下约定：`On<HTTP verb>[Async]`（追加 `Async` 是可选操作，但建议为异步方法执行此操作）。

未命名处理程序方法	操作
`OnGet` / `OnGetAsync`	初始化页面状态。

未命名处理程序方法	操作
`OnPost` / `OnPostAsync`	处理 POST 请求。
`OnDelete` / `OnDeleteAsync`	处理 DELETE 请求†。
`OnPut` / `OnPutAsync`	处理 PUT 请求†。
`OnPatch` / `OnPatchAsync`	处理 PATCH 请求†。

†用于向页面发出 API 调用。

默认的已命名处理程序方法

由开发人员提供的处理程序方法（“已命名”的处理程序方法）遵循类似的约定。处理程序名称出现在 Http 谓词之后或者 Http 谓词与 `Async` 之间：`On<HTTP verb><handler name>[Async]`（追加 `Async` 是可选操作，但建议为异步方法执行此操作）。例如，用于处理消息的方法可能采用下表所示的命名。

已命名处理程序方法示例	操作示例
`OnGetMessage` / `OnGetMessageAsync`	获取消息。
`OnPostMessage` / `OnPostMessageAsync`	POST 消息。
`OnDeleteMessage` / `OnDeleteMessageAsync`	DELETE 消息†。
`OnPutMessage` / `OnPutMessageAsync`	PUT 消息†。
`OnPatchMessage` / `OnPatchMessageAsync`	PATCH 消息†。

†用于向页面发出 API 调用。

自定义处理程序方法名称

假设用户想更改未命名和已命名处理程序方法的命名方式。备选命名方案是避免让方法名称以“On”开头，并使用第一个分词来确定 Http 谓词。也可以进行其他更改，比如将 DELETE、PUT 和 PATCH 的谓词转换为 POST。这种方案提供下表所示的方法名称。

处理程序方法	操作
`Get`	初始化页面状态。
`Post` / `PostAsync`	处理 POST 请求。
`Delete` / `DeleteAsync`	处理 DELETE 请求†。
`Put` / `PutAsync`	处理 PUT 请求†。
`Patch` / `PatchAsync`	处理 PATCH 请求†。
`GetMessage`	获取消息。
`PostMessage` / `PostMessageAsync`	POST 消息。

处理程序方法	操作
`DeleteMessage` / `DeleteMessageAsync`	POST 消息以进行删除。
`PutMessage` / `PutMessageAsync`	POST 消息以进行放置。
`PatchMessage` / `PatchMessageAsync`	POST 消息以进行修补。

†用于向页面发出 API 调用。

若要建立此方案，请从 `DefaultPageApplicationModelProvider` 类继承并重写 CreateHandlerModel 方法，以提供自定义逻辑来解析 PageModel 处理程序名称。示例应用展示了如何在其 `CustomPageApplicationModelProvider` 类中执行此操作：

```
public class CustomPageApplicationModelProvider :
    DefaultPageApplicationModelProvider
{
    public CustomPageApplicationModelProvider(IModelMetadataProvider modelMetadataProvider,
IOptions<MvcOptions> options)
        : base (modelMetadataProvider, options)
    {
    }

    protected override PageHandlerModel CreateHandlerModel(MethodInfo method)
    {
        if (method == null)
        {
            throw new ArgumentNullException(nameof(method));
        }

        if (!IsHandler(method))
        {
            return null;
        }

        if (!TryParseHandlerMethod(
            method.Name, out var httpMethod, out var handlerName))
        {
            return null;
        }

        var handlerModel = new PageHandlerModel(
            method,
            method.GetCustomAttributes(inherit: true))
        {
            Name = method.Name,
            HandlerName = handlerName,
            HttpMethod = httpMethod,
        };

        var methodParameters = handlerModel.MethodInfo.GetParameters();

        for (var i = 0; i < methodParameters.Length; i++)
        {
            var parameter = methodParameters[i];
            var parameterModel = CreateParameterModel(parameter);
            parameterModel.Handler = handlerModel;

            handlerModel.Parameters.Add(parameterModel);
        }

        return handlerModel;
    }
```

```
    private static bool TryParseHandlerMethod(
        string methodName, out string httpMethod, out string handler)
    {
        httpMethod = null;
        handler = null;

        // Parse the method name according to our conventions to
        // determine the required HTTP verb and optional
        // handler name.
        //
        // Valid names look like:
        //  - Get
        //  - Post
        //  - PostAsync
        //  - GetMessage
        //  - PostMessage
        //  - DeleteMessage
        //  - DeleteMessageAsync

        var length = methodName.Length;
        if (methodName.EndsWith("Async", StringComparison.Ordinal))
        {
            length -= "Async".Length;
        }

        if (length == 0)
        {
            // The method is named "Async". Exit processing.
            return false;
        }

        // The HTTP verb is at the start of the method name. Use
        // casing to determine where it ends.
        var handlerNameStart = 1;
        for (; handlerNameStart < length; handlerNameStart++)
        {
            if (char.IsUpper(methodName[handlerNameStart]))
            {
                break;
            }
        }

        httpMethod = methodName.Substring(0, handlerNameStart);

        // The handler name follows the HTTP verb and is optional.
        // It includes everything up to the end excluding the
        // "Async" suffix, if present.
        handler = handlerNameStart == length ? null : methodName.Substring(0, length);

        if (string.Equals(httpMethod, "GET", StringComparison.OrdinalIgnoreCase) ||
            string.Equals(httpMethod, "POST", StringComparison.OrdinalIgnoreCase))
        {
            // Do nothing. The httpMethod is correct for GET and POST.
            return true;
        }
        if (string.Equals(httpMethod, "DELETE", StringComparison.OrdinalIgnoreCase) ||
            string.Equals(httpMethod, "PUT", StringComparison.OrdinalIgnoreCase) ||
            string.Equals(httpMethod, "PATCH", StringComparison.OrdinalIgnoreCase))
        {
            // Convert HTTP verbs for DELETE, PUT, and PATCH to POST
            // For example: DeleteMessage, PutMessage, PatchMessage -> POST
            httpMethod = "POST";
            return true;
        }
        else
        {
            return false;
        }
    }
```

```
}
```

该类的要点包括：

- 该类继承自 `DefaultPageApplicationModelProvider`。
- `TryParseHandlerMethod` 对处理程序进行处理，以便在创建 `PageHandlerModel` 时确定 Http 谓词（`httpMethod`）和已命名处理程序名称（`handlerName`）。
  - 忽略 `Async` 后缀（如果存在）。
  - 使用大小写分析方法名称中的 Http 谓词。
  - 当方法名称（不带 `Async`）与 Http 谓词名称相同时，就不存在已命名处理程序。`handlerName` 设置为 `null`，且方法名称为 `Get`、`Post`、`Delete`、`Put` 或 `Patch`。
  - 当方法名称（不带 `Async`）的长度超过 Http 谓词名称时，就存在已命名处理程序。`handlerName` 设置为 `<method name (less 'Async', if present)>`。例如，“GetMessage”和“GetMessageAsync”都生成处理程序名称“GetMessage”。
  - DELETE、PUT 和 PATCH Http 谓词都转换为 POST。

在 `Startup` 类中注册 `CustomPageApplicationModelProvider`：

```
services.AddSingleton<IPageApplicationModelProvider,
    CustomPageApplicationModelProvider>();
```

Index.cshtml.cs 中的页面模型演示了如何针对应用中的页面更改常规的处理程序方法命名约定。用于 Razor 页面的常规“On”前缀命名已被删除。用于初始化页面状态的方法现在命名为 `Get`。为任何页面打开任何页面模型时，都可以看到此约定贯穿于整个应用。

其他各个方法均以描述其处理的 Http 谓词开头。以 `Delete` 开头的两个方法通常被视为 DELETE Http 谓词，但 `TryParseHandlerMethod` 中的逻辑会针对这两种处理程序将该谓词显式设置为 POST。

请注意，`Async` 在 `DeleteAllMessages` 和 `DeleteMessageAsync` 之间是可选的。它们都是异步方法，但用户可以选择是否使用 `Async` 后缀；我们建议使用。此处使用 `DeleteAllMessages` 只是为了进行演示，但建议将此类方法命名为 `DeleteAllMessagesAsync`。它对示例的实现处理没有影响，但使用 `Async` 后缀强调了它是异步方法的事实。

```
public async Task Get()
{
    Messages = await _db.Messages.AsNoTracking().ToListAsync();
}

public async Task<IActionResult> PostMessageAsync()
{
    _db.Messages.Add(Message);
    await _db.SaveChangesAsync();

    Result = $"{nameof(PostMessageAsync)} handler: Message '{Message.Text}' added.";

    return RedirectToPage();
}

public async Task<IActionResult> DeleteAllMessages()
{
    foreach (Message message in _db.Messages)
    {
        _db.Messages.Remove(message);
    }
    await _db.SaveChangesAsync();

    Result = $"{nameof(DeleteAllMessages)} handler: All messages deleted.";

    return RedirectToPage();
}

public async Task<IActionResult> DeleteMessageAsync(int id)
{
    var message = await _db.Messages.FindAsync(id);

    if (message != null)
    {
        _db.Messages.Remove(message);
        await _db.SaveChangesAsync();
    }

    Result = $"{nameof(DeleteMessageAsync)} handler: Message with Id: {id} deleted.";

    return RedirectToPage();
}
```

请注意，*Index.cshtml* 中提供的处理程序名称与 `DeleteAllMessages` 和 `DeleteMessageAsync` 处理程序方法相匹配：

```
<div class="row">
    <div class="col-md-3">
        <form method="post">
            <h2>Clear all messages</h2>
            <hr>
            <div class="form-group">
                <button type="submit" asp-page-handler="DeleteAllMessages"
                        class="btn btn-danger">Clear All</button>
            </div>
        </form>
    </div>
</div>

<div class="row">
    <div class="col-md-12">
        <form method="post">
            <h2>Messages</h2>
            <hr>
            <ol>
                @foreach (var message in Model.Messages)
                {
                    <li>
                        @message.Text
                        <button type="submit" asp-page-handler="DeleteMessage"
                            class="btn btn-danger"
                            asp-route-id="@message.Id">Delete</button>
                    </li>
                }
            </ol>
        </form>
    </div>
```

为了方便处理程序将 POST 请求与方法进行匹配，`TryParseHandlerMethod` 剔除了处理程序方法名称 `DeleteMessageAsync` 中的 `Async`。`DeleteMessage` 的 `asp-page-handler` 名称与处理程序方法 `DeleteMessageAsync` 匹配。

MVC 筛选器和页面筛选器 (IPageFilter)

Razor 页面会忽略 MVC 操作筛选器，因为 Razor 页面使用处理程序方法。可使用其他类型的 MVC 筛选器：授权、异常、资源和结果。有关详细信息，请参阅筛选器主题。

页面筛选器 (IPageFilter) 是应用于 Razor 页面的一种筛选器。有关详细信息，请参阅 Razor 页面的筛选方法。

其他资源

- Razor 页面授权约定

将文件上传到 ASP.NET Core 中的 Razor 页面

2018/12/10 • • Edit Online

作者：Luke Latham

本主题是基于示例应用中教程：在 ASP.NET Core 中开始使用 Razor Pages。

本主题演示如何使用简单的模型绑定来上传文件，这非常适合上传小文件。有关流式传输大文件的信息，请参阅通过流式传输上传大文件。

在下列步骤中，向示例应用添加电影计划文件上传功能。每个电影计划由一个 `Schedule` 类表示。该类包括两个版本的计划。其中一个版本（`PublicSchedule`）提供给客户。另一个版本（`PrivateSchedule`）用于公司员工。每个版本作为单独的文件进行上传。本教程演示如何通过单个 POST 将两个文件上传至服务器。

查看或下载示例代码（如何下载）

安全注意事项

向用户提供向服务器上传文件的功能时，必须格外小心。攻击者可能对系统执行拒绝服务和其他攻击。一些降低成功攻击可能性的安全措施如下：

- 将文件上传到系统上的专用文件上传区域，这样可以更轻松地对上传内容实施安全措施。如果允许文件上传，请确保在上传位置禁用执行权限。
- 使用由应用确定的安全文件名，而不是采用用户输入或已上传文件的文件名。
- 仅允许使用一组特定的已批准文件扩展名。
- 验证是否在服务器上执行了客户端检查。客户端检查很容易规避。
- 检查上传文件大小，防止上传比预期大的文件。
- 对上传的内容运行病毒/恶意软件扫描程序。

> **WARNING**
>
> 将恶意代码上传到系统通常是执行代码的第一步，这些代码可以：
>
> - 完全接管系统。
> - 重载系统，导致系统完全崩溃。
> - 泄露用户或系统数据。
> - 将涂鸦应用于公共接口。

添加 FileUpload 类

创建 Razor 页以处理一对文件上传。添加 `FileUpload` 类（此类与页面绑定以获取计划数据）。右键单击“Models”文件夹。选择“添加” > “类”。将类命名为“FileUpload”，并添加以下属性：

```
using Microsoft.AspNetCore.Http;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models
{
    public class FileUpload
    {
        [Required]
        [Display(Name="Title")]
        [StringLength(60, MinimumLength = 3)]
        public string Title { get; set; }

        [Required]
        [Display(Name="Public Schedule")]
        public IFormFile UploadPublicSchedule { get; set; }

        [Required]
        [Display(Name="Private Schedule")]
        public IFormFile UploadPrivateSchedule { get; set; }
    }
}
```


```
using Microsoft.AspNetCore.Http;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models
{
    public class FileUpload
    {
        [Required]
        [Display(Name="Title")]
        [StringLength(60, MinimumLength = 3)]
        public string Title { get; set; }

        [Required]
        [Display(Name="Public Schedule")]
        public IFormFile UploadPublicSchedule { get; set; }

        [Required]
        [Display(Name="Private Schedule")]
        public IFormFile UploadPrivateSchedule { get; set; }
    }
}
```


此类有一个属性对应计划标题，另各有一个属性对应计划的两个版本。3 个属性皆为必需属性，标题长度必须为 3-60 个字符。

添加用于上传文件的 helper 方法

为避免处理未上传计划文件时出现代码重复，请首先上传一个静态 helper 方法。在此应用中创建一个“Utilities”文件夹，然后在“FileHelpers.cs”文件中添加以下内容。helper 方法 `ProcessFormFile` 接受 IFormFile 和 ModelStateDictionary，并返回包含文件大小和内容的字符串。检查内容类型和长度。如果文件未通过验证检查，将向 `ModelState` 添加一个错误。

```
using System;
using System.ComponentModel.DataAnnotations;
using System.IO;
using System.Net;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
```

```
using Microsoft.AspNetCore.Mvc.ModelBinding;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Utilities
{
    public class FileHelpers
    {
        public static async Task<string> ProcessFormFile(IFormFile formFile,
            ModelStateDictionary modelState)
        {
            var fieldDisplayName = string.Empty;

            // Use reflection to obtain the display name for the model
            // property associated with this IFormFile. If a display
            // name isn't found, error messages simply won't show
            // a display name.
            MemberInfo property =
                typeof(FileUpload).GetProperty(
                    formFile.Name.Substring(formFile.Name.IndexOf(".") + 1));

            if (property != null)
            {
                var displayAttribute =
                    property.GetCustomAttribute(typeof(DisplayAttribute))
                        as DisplayAttribute;

                if (displayAttribute != null)
                {
                    fieldDisplayName = $"{displayAttribute.Name} ";
                }
            }

            // Use Path.GetFileName to obtain the file name, which will
            // strip any path information passed as part of the
            // FileName property. HtmlEncode the result in case it must
            // be returned in an error message.
            var fileName = WebUtility.HtmlEncode(
                Path.GetFileName(formFile.FileName));

            if (formFile.ContentType.ToLower() != "text/plain")
            {
                modelState.AddModelError(formFile.Name,
                    $"The {fieldDisplayName}file ({fileName}) must be a text file.");
            }

            // Check the file length and don't bother attempting to
            // read it if the file contains no content. This check
            // doesn't catch files that only have a BOM as their
            // content, so a content length check is made later after
            // reading the file's content to catch a file that only
            // contains a BOM.
            if (formFile.Length == 0)
            {
                modelState.AddModelError(formFile.Name,
                    $"The {fieldDisplayName}file ({fileName}) is empty.");
            }
            else if (formFile.Length > 1048576)
            {
                modelState.AddModelError(formFile.Name,
                    $"The {fieldDisplayName}file ({fileName}) exceeds 1 MB.");
            }
            else
            {
                try
                {
                    string fileContents;

                    // The StreamReader is created to read files that are UTF-8 encoded.
                    // If uploads require some other encoding, provide the encoding in the
```

```
                    // using statement. To change to 32-bit encoding, change
                    // new UTF8Encoding(...) to new UTF32Encoding().
                    using (
                        var reader =
                            new StreamReader(
                                formFile.OpenReadStream(),
                                new UTF8Encoding(encoderShouldEmitUTF8Identifier: false,
                                    throwOnInvalidBytes: true),
                                detectEncodingFromByteOrderMarks: true))
                    {
                        fileContents = await reader.ReadToEndAsync();

                        // Check the content length in case the file's only
                        // content was a BOM and the content is actually
                        // empty after removing the BOM.
                        if (fileContents.Length > 0)
                        {
                            return fileContents;
                        }
                        else
                        {
                            modelState.AddModelError(formFile.Name,
                                $"The {fieldDisplayName}file ({fileName}) is empty.");
                        }
                    }
                }
                catch (Exception ex)
                {
                    modelState.AddModelError(formFile.Name,
                        $"The {fieldDisplayName}file ({fileName}) upload failed. " +
                        $"Please contact the Help Desk for support. Error: {ex.Message}");
                    // Log the exception
                }
            }

            return string.Empty;
        }
    }
}
```

```
using System;
using System.ComponentModel.DataAnnotations;
using System.IO;
using System.Net;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Utilities
{
    public class FileHelpers
    {
        public static async Task<string> ProcessFormFile(
            IFormFile formFile, ModelStateDictionary modelState)
        {
            var fieldDisplayName = string.Empty;

            // Use reflection to obtain the display name for the model
            // property associated with this IFormFile. If a display
            // name isn't found, error messages simply won't show
            // a display name.
            MemberInfo property =
                typeof(FileUpload).GetProperty(formFile.Name.Substring(
                    formFile.Name.IndexOf(".") + 1));
```

```
            if (property != null)
            {
                var displayAttribute =
                    property.GetCustomAttribute(typeof(DisplayAttribute))
                        as DisplayAttribute;

                if (displayAttribute != null)
                {
                    fieldDisplayName = $"{displayAttribute.Name} ";
                }
            }

            // Use Path.GetFileName to obtain the file name, which will
            // strip any path information passed as part of the
            // FileName property. HtmlEncode the result in case it must
            // be returned in an error message.
            var fileName = WebUtility.HtmlEncode(
                Path.GetFileName(formFile.FileName));

            if (formFile.ContentType.ToLower() != "text/plain")
            {
                modelState.AddModelError(formFile.Name,
                    $"The {fieldDisplayName}file ({fileName}) must be a text file.");
            }

            // Check the file length and don't bother attempting to
            // read it if the file contains no content. This check
            // doesn't catch files that only have a BOM as their
            // content, so a content length check is made later after
            // reading the file's content to catch a file that only
            // contains a BOM.
            if (formFile.Length == 0)
            {
                modelState.AddModelError(formFile.Name,
                    $"The {fieldDisplayName}file ({fileName}) is empty.");
            }
            else if (formFile.Length > 1048576)
            {
                modelState.AddModelError(formFile.Name,
                    $"The {fieldDisplayName}file ({fileName}) exceeds 1 MB.");
            }
            else
            {
                try
                {
                    string fileContents;

                    // The StreamReader is created to read files that are UTF-8 encoded.
                    // If uploads require some other encoding, provide the encoding in the
                    // using statement. To change to 32-bit encoding, change
                    // new UTF8Encoding(...) to new UTF32Encoding().
                    using (
                        var reader =
                            new StreamReader(
                                formFile.OpenReadStream(),
                                new UTF8Encoding(encoderShouldEmitUTF8Identifier: false,
                                    throwOnInvalidBytes: true),
                                detectEncodingFromByteOrderMarks: true))
                    {
                        fileContents = await reader.ReadToEndAsync();

                        // Check the content length in case the file's only
                        // content was a BOM and the content is actually
                        // empty after removing the BOM.
                        if (fileContents.Length > 0)
                        {
                            return fileContents;
                        }
```

```
                        }
                        else
                        {
                            modelState.AddModelError(formFile.Name,
                                $"The {fieldDisplayName}file ({fileName}) is empty.");
                        }
                    }
                }
                catch (Exception ex)
                {
                    modelState.AddModelError(formFile.Name,
                        $"The {fieldDisplayName}file ({fileName}) upload failed. " +
                        $"Please contact the Help Desk for support. Error: {ex.Message}");
                    // Log the exception
                }
            }

            return string.Empty;
        }
    }
}
```

将文件保存到磁盘

示例应用将已上传的文件保存到数据库字段中。要将文件保存到磁盘，请使用 FileStream。以下示例将 `FileUpload.UploadPublicSchedule` 所保存的文件复制到 `OnPostAsync` 方法中的 `FileStream`。`FileStream` 将文件写入 `<PATH-AND-FILE-NAME>` 提供的磁盘：

```
public async Task<IActionResult> OnPostAsync()
{
    // Perform an initial check to catch FileUpload class attribute violations.
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var filePath = "<PATH-AND-FILE-NAME>";

    using (var fileStream = new FileStream(filePath, FileMode.Create))
    {
        await FileUpload.UploadPublicSchedule.CopyToAsync(fileStream);
    }

    return RedirectToPage("./Index");
}
```

工作进程必须对 `filePath` 指定的位置具有写入权限。

> **NOTE**
>
> `filePath` 必须包含文件名。如果未提供文件名，则会在运行时引发 UnauthorizedAccessException。

> **WARNING**
>
> 切勿将上传的文件保存在与应用相同的目录树中。
>
> 代码示例不提供针对恶意文件上传的服务器端保护。有关在接受用户文件时减少攻击外围应用的信息，请参阅以下资源：
>
> - Unrestricted File Upload（不受限制的文件上传）
> - Azure 安全性：确保在接受用户文件时采取适当的控制措施

将文件保存到 Azure Blob 存储

若要将文件内容上传到 Azure Blob 存储，请参阅使用 .NET 的 Azure Blob 存储入门。本主题演示如何使用 UploadFromStream 将文件流保存到 blob 存储。

添加 Schedule 类

右键单击“Models”文件夹。选择“添加” > “类”。将类命名为“Schedule”，并添加以下属性：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Schedule
    {
        public int ID { get; set; }
        public string Title { get; set; }

        public string PublicSchedule { get; set; }

        [Display(Name = "Public Schedule Size (bytes)")]
        [DisplayFormat(DataFormatString = "{0:N1}")]
        public long PublicScheduleSize { get; set; }

        public string PrivateSchedule { get; set; }

        [Display(Name = "Private Schedule Size (bytes)")]
        [DisplayFormat(DataFormatString = "{0:N1}")]
        public long PrivateScheduleSize { get; set; }

        [Display(Name = "Uploaded (UTC)")]
        [DisplayFormat(DataFormatString = "{0:F}")]
        public DateTime UploadDT { get; set; }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Schedule
    {
        public int ID { get; set; }
        public string Title { get; set; }

        public string PublicSchedule { get; set; }

        [Display(Name = "Public Schedule Size (bytes)")]
        [DisplayFormat(DataFormatString = "{0:N1}")]
        public long PublicScheduleSize { get; set; }

        public string PrivateSchedule { get; set; }

        [Display(Name = "Private Schedule Size (bytes)")]
        [DisplayFormat(DataFormatString = "{0:N1}")]
        public long PrivateScheduleSize { get; set; }

        [Display(Name = "Uploaded (UTC)")]
        [DisplayFormat(DataFormatString = "{0:F}")]
        public DateTime UploadDT { get; set; }
    }
}
```

此类使用 `Display` 和 `DisplayFormat` 特性，呈现计划数据时，这些特性会生成友好型的标题和格式。

更新 RazorPagesMovieContext

在 `RazorPagesMovieContext` (Data/RazorPagesMovieContext.cs) 中为计划指定 `DbSet`：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Models
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext (DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; }
        public DbSet<RazorPagesMovie.Models.Schedule> Schedule { get; set; }
    }
}
```

更新 MovieContext

在 `MovieContext` (*Models/MovieContext.cs*) 中为计划指定 `DbSet`：

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Models
{
    public class MovieContext : DbContext
    {
        public MovieContext(DbContextOptions<MovieContext> options)
            : base(options)
        {
        }

        public DbSet<Movie> Movie { get; set; }
        public DbSet<Schedule> Schedule { get; set; }
    }
}
```

将 Schedule 表添加到数据库

打开包管理器控制台 (PMC)："工具" > "NuGet 包管理器" > "包管理器控制台"。

在 PMC 中执行以下命令。这些命令将向数据库添加 `Schedule` 表：

```
Add-Migration AddScheduleTable
Update-Database
```

添加文件上传 Razor 页面

在"Pages"文件夹中创建"Schedules"文件夹。在"Schedules"文件夹中，创建名为"Index.cshtml"的页面，用于上传具有如下内容的计划：

```
@page
@model RazorPagesMovie.Pages.Schedules.IndexModel

@{
    ViewData["Title"] = "Schedules";
}

<h2>Schedules</h2>
<hr />
```

```
<h3>Upload Schedules</h3>
<div class="row">
    <div class="col-md-4">
        <form method="post" enctype="multipart/form-data">
            <div class="form-group">
                <label asp-for="FileUpload.Title" class="control-label"></label>
                <input asp-for="FileUpload.Title" type="text" class="form-control" />
                <span asp-validation-for="FileUpload.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FileUpload.UploadPublicSchedule" class="control-label"></label>
                <input asp-for="FileUpload.UploadPublicSchedule" type="file" class="form-control"
style="height:auto" />
                <span asp-validation-for="FileUpload.UploadPublicSchedule" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FileUpload.UploadPrivateSchedule" class="control-label"></label>
                <input asp-for="FileUpload.UploadPrivateSchedule" type="file" class="form-control"
style="height:auto" />
                <span asp-validation-for="FileUpload.UploadPrivateSchedule" class="text-danger"></span>
            </div>
            <input type="submit" value="Upload" class="btn btn-default" />
        </form>
    </div>
</div>

<h3>Loaded Schedules</h3>
<table class="table">
    <thead>
        <tr>
            <th></th>
            <th>
                @Html.DisplayNameFor(model => model.Schedule[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Schedule[0].UploadDT)
            </th>
            <th class="text-center">
                @Html.DisplayNameFor(model => model.Schedule[0].PublicScheduleSize)
            </th>
            <th class="text-center">
                @Html.DisplayNameFor(model => model.Schedule[0].PrivateScheduleSize)
            </th>
        </tr>
    </thead>
    <tbody>
    @foreach (var item in Model.Schedule) {
        <tr>
            <td>
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.UploadDT)
            </td>
            <td class="text-center">
                @Html.DisplayFor(modelItem => item.PublicScheduleSize)
            </td>
            <td class="text-center">
                @Html.DisplayFor(modelItem => item.PrivateScheduleSize)
            </td>
        </tr>
    }
    </tbody>
</table>

@section Scripts {
```

```
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

```
@page
@model RazorPagesMovie.Pages.Schedules.IndexModel

@{
    ViewData["Title"] = "Schedules";
}

<h2>Schedules</h2>
<hr />

<h3>Upload Schedules</h3>
<div class="row">
    <div class="col-md-4">
        <form method="post" enctype="multipart/form-data">
            <div class="form-group">
                <label asp-for="FileUpload.Title" class="control-label"></label>
                <input asp-for="FileUpload.Title" type="text" class="form-control" />
                <span asp-validation-for="FileUpload.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FileUpload.UploadPublicSchedule" class="control-label"></label>
                <input asp-for="FileUpload.UploadPublicSchedule" type="file" class="form-control"
style="height:auto" />
                <span asp-validation-for="FileUpload.UploadPublicSchedule" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FileUpload.UploadPrivateSchedule" class="control-label"></label>
                <input asp-for="FileUpload.UploadPrivateSchedule" type="file" class="form-control"
style="height:auto" />
                <span asp-validation-for="FileUpload.UploadPrivateSchedule" class="text-danger"></span>
            </div>
            <input type="submit" value="Upload" class="btn btn-default" />
        </form>
    </div>
</div>

<h3>Loaded Schedules</h3>
<table class="table">
    <thead>
        <tr>
            <th></th>
            <th>
                @Html.DisplayNameFor(model => model.Schedule[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Schedule[0].UploadDT)
            </th>
            <th class="text-center">
                @Html.DisplayNameFor(model => model.Schedule[0].PublicScheduleSize)
            </th>
            <th class="text-center">
                @Html.DisplayNameFor(model => model.Schedule[0].PrivateScheduleSize)
            </th>
        </tr>
    </thead>
    <tbody>
    @foreach (var item in Model.Schedule) {
        <tr>
            <td>
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
```

```
            <td>
                @Html.DisplayFor(modelItem => item.UploadDT)
            </td>
            <td class="text-center">
                @Html.DisplayFor(modelItem => item.PublicScheduleSize)
            </td>
            <td class="text-center">
                @Html.DisplayFor(modelItem => item.PrivateScheduleSize)
            </td>
        </tr>
    }
    </tbody>
</table>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

每个窗体组包含一个 <label>，它显示每个类属性的名称。`FileUpload` 模型中的 `Display` 特性提供这些标签的显示值。例如，`UploadPublicSchedule` 特性的显示名称通过 `[Display(Name="Public Schedule")]` 进行设置，因此呈现窗体时会在此标签中显示“Public Schedule”。

每个窗体组包含一个验证 。如果用户输入未能满足 `FileUpload` 类中设置的属性特性，或者任何 `ProcessFormFile` 方法文件检查失败，则模型验证会失败。模型验证失败时，会向用户呈现有用的验证消息。例如，`Title` 属性带有 `[Required]` 和 `[StringLength(60, MinimumLength = 3)]` 注释。用户若未提供标题，会接收到一条指示需要提供值的消息。如果用户输入的值少于 3 个字符或多于 60 个字符，则会接收到一条指示值长度不正确的消息。如果提供不含内容的文件，则会显示一条指示文件为空的消息。

添加页面模型

将页面模型 (Index.cshtml.cs) 添加到“Schedules”文件夹中：

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;
using RazorPagesMovie.Utilities;

namespace RazorPagesMovie.Pages.Schedules
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

        public IndexModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
        {
            _context = context;
        }

        [BindProperty]
        public FileUpload FileUpload { get; set; }

        public IList<Schedule> Schedule { get; private set; }

        public async Task OnGetAsync()
        {
            Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        }

        public async Task<IActionResult> OnPostAsync()
        {
```

```
        {
            // Perform an initial check to catch FileUpload class
            // attribute violations.
            if (!ModelState.IsValid)
            {
                Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
                return Page();
            }

            var publicScheduleData =
                await FileHelpers.ProcessFormFile(FileUpload.UploadPublicSchedule, ModelState);

            var privateScheduleData =
                await FileHelpers.ProcessFormFile(FileUpload.UploadPrivateSchedule, ModelState);

            // Perform a second check to catch ProcessFormFile method
            // violations.
            if (!ModelState.IsValid)
            {
                Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
                return Page();
            }

            var schedule = new Schedule()
                {
                    PublicSchedule = publicScheduleData,
                    PublicScheduleSize = FileUpload.UploadPublicSchedule.Length,
                    PrivateSchedule = privateScheduleData,
                    PrivateScheduleSize = FileUpload.UploadPrivateSchedule.Length,
                    Title = FileUpload.Title,
                    UploadDT = DateTime.UtcNow
                };

            _context.Schedule.Add(schedule);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;
using RazorPagesMovie.Utilities;

namespace RazorPagesMovie.Pages.Schedules
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public IndexModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        [BindProperty]
        public FileUpload FileUpload { get; set; }

        public IList<Schedule> Schedule { get; private set; }

        public async Task OnGetAsync()
        {
```

```
            Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            // Perform an initial check to catch FileUpload class
            // attribute violations.
            if (!ModelState.IsValid)
            {
                Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
                return Page();
            }

            var publicScheduleData =
                await FileHelpers.ProcessFormFile(FileUpload.UploadPublicSchedule, ModelState);

            var privateScheduleData =
                await FileHelpers.ProcessFormFile(FileUpload.UploadPrivateSchedule, ModelState);

            // Perform a second check to catch ProcessFormFile method
            // violations.
            if (!ModelState.IsValid)
            {
                Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
                return Page();
            }

            var schedule = new Schedule()
                {
                    PublicSchedule = publicScheduleData,
                    PublicScheduleSize = FileUpload.UploadPublicSchedule.Length,
                    PrivateSchedule = privateScheduleData,
                    PrivateScheduleSize = FileUpload.UploadPrivateSchedule.Length,
                    Title = FileUpload.Title,
                    UploadDT = DateTime.UtcNow
                };

            _context.Schedule.Add(schedule);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

页面模型(Index.cshtml.cs 中的 `IndexModel`)绑定 `FileUpload` 类:

```
[BindProperty]
public FileUpload FileUpload { get; set; }
```


```
[BindProperty]
public FileUpload FileUpload { get; set; }
```


此模型还使用计划列表(`IList<Schedule>`)在页面上显示数据库中存储的计划:

```
public IList<Schedule> Schedule { get; private set; }
```


```
public IList<Schedule> Schedule { get; private set; }
```


页面加载 `OnGetAsync` 时,会从数据库填充 `Schedules`,用于生成已加载计划的 HTML 表:

```
public async Task OnGetAsync()
{
    Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
}
```


```
public async Task OnGetAsync()
{
    Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
}
```


将窗体发布到服务器时，会检查 `ModelState`。如果无效，会重新生成 `Schedule`，且页面会呈现一个或多个验证消息，陈述页面验证失败的原因。如果有效，`FileUpload` 属性将用于“OnPostAsync”中，以完成两个计划版本的文件上传，并创建一个用于存储数据的新 `Schedule` 对象。然后会将此计划保存到数据库：

```
public async Task<IActionResult> OnPostAsync()
{
    // Perform an initial check to catch FileUpload class
    // attribute violations.
    if (!ModelState.IsValid)
    {
        Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        return Page();
    }

    var publicScheduleData =
        await FileHelpers.ProcessSchedule(FileUpload.UploadPublicSchedule, ModelState);

    var privateScheduleData =
        await FileHelpers.ProcessSchedule(FileUpload.UploadPrivateSchedule, ModelState);

    // Perform a second check to catch ProcessSchedule method
    // violations.
    if (!ModelState.IsValid)
    {
        Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        return Page();
    }

    var schedule = new Schedule()
        {
            PublicSchedule = publicScheduleData,
            PublicScheduleSize = FileUpload.UploadPublicSchedule.Length,
            PrivateSchedule = privateScheduleData,
            PrivateScheduleSize = FileUpload.UploadPrivateSchedule.Length,
            Title = FileUpload.Title,
            UploadDT = DateTime.UtcNow
        };

    _context.Schedule.Add(schedule);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

```
public async Task<IActionResult> OnPostAsync()
{
    // Perform an initial check to catch FileUpload class
    // attribute violations.
    if (!ModelState.IsValid)
    {
        Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        return Page();
    }

    var publicScheduleData =
        await FileHelpers.ProcessSchedule(FileUpload.UploadPublicSchedule, ModelState);

    var privateScheduleData =
        await FileHelpers.ProcessSchedule(FileUpload.UploadPrivateSchedule, ModelState);

    // Perform a second check to catch ProcessSchedule method
    // violations.
    if (!ModelState.IsValid)
    {
        Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        return Page();
    }

    var schedule = new Schedule()
        {
            PublicSchedule = publicScheduleData,
            PublicScheduleSize = FileUpload.UploadPublicSchedule.Length,
            PrivateSchedule = privateScheduleData,
            PrivateScheduleSize = FileUpload.UploadPrivateSchedule.Length,
            Title = FileUpload.Title,
            UploadDT = DateTime.UtcNow
        };

    _context.Schedule.Add(schedule);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

链接文件上传 Razor 页面

打开“Pages/Shared/_Layout.cshtml”，然后向导航栏添加一个链接以访问“计划”页面：

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a asp-page="/Index">Home</a></li>
        <li><a asp-page="/Schedules/Index">Schedules</a></li>
        <li><a asp-page="/About">About</a></li>
        <li><a asp-page="/Contact">Contact</a></li>
    </ul>
</div>
```

添加计划删除确认页面

用户单击删除计划时，为其提供取消此操作的机会。向“Schedules”文件夹添加删除确认页面 (Delete.cshtml)：

```
@page "{id:int}"
@model RazorPagesMovie.Pages.Schedules.DeleteModel

@{
    ViewData["Title"] = "Delete Schedule";
}

<h2>Delete Schedule</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Schedule</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.PublicScheduleSize)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.PublicScheduleSize)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.PrivateScheduleSize)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.PrivateScheduleSize)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.UploadDT)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.UploadDT)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Schedule.ID" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

```
@page "{id:int}"
@model RazorPagesMovie.Pages.Schedules.DeleteModel

@{
    ViewData["Title"] = "Delete Schedule";
}

<h2>Delete Schedule</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Schedule</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.PublicScheduleSize)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.PublicScheduleSize)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.PrivateScheduleSize)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.PrivateScheduleSize)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.UploadDT)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.UploadDT)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Schedule.ID" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

页面模型 (Delete.cshtml.cs) 在请求的路由数据中加载由 `id` 标识的单个计划。将“Delete.cshtml.cs”文件添加到“Schedules”文件夹：

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Schedules
{
    public class DeleteModel : PageModel
    {
        private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

        public DeleteModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Schedule Schedule { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Schedule = await _context.Schedule.SingleOrDefaultAsync(m => m.ID == id);

            if (Schedule == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Schedule = await _context.Schedule.FindAsync(id);

            if (Schedule != null)
            {
                _context.Schedule.Remove(Schedule);
                await _context.SaveChangesAsync();
            }

            return RedirectToPage("./Index");
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Schedules
{
    public class DeleteModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public DeleteModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Schedule Schedule { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Schedule = await _context.Schedule.SingleOrDefaultAsync(m => m.ID == id);

            if (Schedule == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Schedule = await _context.Schedule.FindAsync(id);

            if (Schedule != null)
            {
                _context.Schedule.Remove(Schedule);
                await _context.SaveChangesAsync();
            }

            return RedirectToPage("./Index");
        }
    }
}
```

`OnPostAsync` 方法按 `id` 处理计划删除：

```
public async Task<IActionResult> OnPostAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Schedule = await _context.Schedule.FindAsync(id);

    if (Schedule != null)
    {
        _context.Schedule.Remove(Schedule);
        await _context.SaveChangesAsync();
    }

    return RedirectToPage("./Index");
}
```


```
public async Task<IActionResult> OnPostAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Schedule = await _context.Schedule.FindAsync(id);

    if (Schedule != null)
    {
        _context.Schedule.Remove(Schedule);
        await _context.SaveChangesAsync();
    }

    return RedirectToPage("./Index");
}
```


成功删除计划后，`RedirectToPage` 将返回到计划的"Index.cshtml"页面。

有效的 Schedules Razor 页面

页面加载时，计划标题、公用计划和专用计划的标签和输入将呈现提交按钮：

在不填充任何字段的情况下选择"上传"按钮会违反此模型上的 `[Required]` 特性。`ModelState` 无效。会向用户显示验证错误消息：

在"标题"字段中键入两个字母。验证消息改为指示标题长度必须介于 3-60 个字符之间：

上传一个或多个计划时，"已加载计划"部分会显示已加载计划：

用户可单击该表中的"删除"链接以访问删除确认视图，并在其中选择确认或取消删除操作。

疑难解答

有关疑难解答的信息与 `IFormFile` 上传，请参阅ASP.NET Core 中的文件上传：疑难解答。

ASP.NET Core Razor SDK

2018/12/12 • • Edit Online

作者：Rick Anderson

.NET Core 2.1 SDK 或更高版本包括 `Microsoft.NET.Sdk.Razor` MSBuild SDK (Razor SDK)。Razor SDK：

- 针对基于 ASP.NET Core MVC 的项目，围绕包含 Razor 文件的项目的生成、打包和发布设定了体验标准。
- 包含一组预定义的目标、属性和项目，它们允许自定义 Razor 文件的编译。

系统必备

.NET Core 2.1 SDK 或更高版本

使用 Razor SDK

大多数 web 应用程序无需显式引用 Razor SDK。

要使用 Razor SDK 来生成包含 Razor 视图或 Razor 页面的类库，请执行以下操作：

- 使用 `Microsoft.NET.Sdk.Razor` 而非 `Microsoft.NET.Sdk`：

```
<Project SDK="Microsoft.NET.Sdk.Razor">
  ...
</Project>
```

- 通常情况下，对的包引用 `Microsoft.AspNetCore.Mvc`，才能接收生成和编译 Razor 页面和 Razor 视图所需的附加依赖项。至少，你的项目应添加到包引用：
  - `Microsoft.AspNetCore.Razor.Design`
  - `Microsoft.AspNetCore.Mvc.Razor.Extensions`

  `Microsoft.AspNetCore.Razor.Design` 包提供的 Razor 编译任务和目标项目。

  前面的包包含在 `Microsoft.AspNetCore.Mvc` 中。以下标记显示了使用 Razor SDK 用于生成 ASP.NET Core Razor 页面应用程序的 Razor 文件的项目文件：

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.1.3" />
  </ItemGroup>

</Project>
```

> **WARNING**
> `Microsoft.AspNetCore.Razor.Design` 并 `Microsoft.AspNetCore.Mvc.Razor.Extensions` 包中包含Microsoft.AspNetCore.App 元包。但是，与版本无关 `Microsoft.AspNetCore.App` 包引用提供一个元包不包括的最新版本的应用程序到 `Microsoft.AspNetCore.Razor.Design` 。项目必须引用的一致版本 `Microsoft.AspNetCore.Razor.Design` (或 `Microsoft.AspNetCore.Mvc`)，以便包含 razor 的最新生成时修补程序。有关详细信息，请参阅此 GitHub 问题。

Properties

以下属性控制项目生成过程中 Razor 的 SDK 行为：

- `RazorCompileOnBuild` – 当 `true` 、编译并发出作为生成项目的一部分 Razor 程序集。默认为 `true` 。
- `RazorCompileOnPublish` – 当 `true` 、编译并发出作为发布项目的一部分 Razor 程序集。默认为 `true` 。

配置输入和输出到 Razor SDK 用于属性和下表中的项。

项	描述
`RazorGenerate`	输入到代码生成目标的项元素（.cshtml 文件）。
`RazorCompile`	项元素 (*.cs*文件)，是 Razor 编译目标的输入。使用此 ItemGroup 指定要编译到 Razor 程序集中的其他文件。

`RazorTargetAssemblyAttribute`	用于编码生成 Razor 程序集属性的项元素。例如：
	`RazorAssemblyAttribute`
	`Include="System.Reflection.AssemblyMetadataAttribute"`
	`_Parameter1="BuildSource"`
	`_Parameter2="https://docs.microsoft.com/">`

`RazorEmbeddedResource`	作为嵌入资源添加到生成的 Razor 程序集的项元素。

属性	描述
`RazorTargetName`	Razor 生成的程序集的文件名（不含扩展名）。
`RazorOutputPath`	Razor 输出目录。
`RazorCompileToolset`	用于确定用于生成 Razor 程序集的工具集。有效值为 `Implicit` 、`RazorSDK` 和 `PrecompilationTool` 。
`EnableDefaultContentItems`	为 `true` 时，包括某些文件类型（例如 .cshtml 文件）作为项目中内容。当通过引用 `Microsoft.NET.Sdk.Web` ，文件下*wwwroot*和，还提供了配置文件。
`EnableDefaultRazorGenerateItems`	为 `true` 时，包括 `RazorGenerate` 项中 `Content` 项的 .cshtml 文件。
`GenerateRazorTargetAssemblyInfo`	当 `true` ，生成 *.cs*包含指定的属性文件 `RazorAssemblyAttribute` 和编译输出中包括的文件。
`EnableDefaultRazorTargetAssemblyInfoAttributes`	为 `true` 时，将一组默认的程序集属性添加到 `RazorAssemblyAttribute` 。

属性	描述
`CopyRazorGenerateFilesToPublishDirectory`	当 `true`，复制 `RazorGenerate` 项 (*.cshtml*) 文件复制到发布目录。通常情况下，Razor 文件不需要的已发布的应用，如果他们参与在生成时或发布时编译。默认为 `false`。
`CopyRefAssembliesToPublishDirectory`	为 `true` 时，将引用程序集项复制到发布目录。通常情况下，引用程序集不需要的已发布的应用，如果 Razor 编译发生在生成时或发布时间。设置为 `true` 如果你已发布的应用需要运行时编译。例如，将值设置为 `true` 如果应用程序修改 *.cshtml*文件在运行时或使用嵌入的视图。默认为 `false`。
`IncludeRazorContentInPack`	当 `true`，所有 Razor 内容项 (*.cshtml*文件) 标记为要包含在生成的 NuGet 包中。默认为 `false`。
`EmbedRazorGenerateSources`	为 `true` 时，将 RazorGenerate (.cshtml) 项作为嵌入的文件添加到生成的 Razor 程序集中。默认为 `false`。
`UseRazorBuildServer`	为 `true` 时，使用永久生成服务器进程来卸载代码生成工作。默认值为 `UseSharedCompilation`。

目标

Razor SDK 定义两个主要目标：

- `RazorGenerate` – 代码将生成 *.cs*文件从 `RazorGenerate` 项元素。使用 `RazorGenerateDependsOn` 属性指定可在此目标之前或之后运行的其他目标。
- `RazorCompile` – 生成的编译 *.cs*到 Razor 程序集文件中。使用 `RazorCompileDependsOn` 指定可在此目标之前或之后运行的其他目标。

Razor 视图的运行时编译

- 默认情况下，Razor SDK 不发布执行运行时编译所需的引用程序集。当应用程序模型依赖于运行时编译时，这会导致编译失败—例如，应用在发布后使用嵌入视图或更改视图。将 `CopyRefAssembliesToPublishDirectory` 设置为 `true`，以继续发布引用程序集。
- 对于 web 应用，请确保您的应用程序所面向 `Microsoft.NET.Sdk.Web` SDK。

ASP.NET Core MVC 中的视图

2018/8/2 • • Edit Online

作者：Steve Smith 和 Luke Latham

本文档介绍在 ASP.NET Core MVC 应用程序中使用的视图。有关 Razor 页的信息，请参阅 Razor 页简介。

在“模型-视图-控制器(MVC)”模式中，视图处理应用的数据表示和用户交互。视图是嵌入了 Razor 标记的 HTML 模板。Razor 标记一个代码，用于与 HTML 标记交互以生成发送给客户端的网页。

在 ASP.NET Core MVC 中，视图是在 Razor 标记中使用 C# 编程语言的 .cshtml 文件。通常，视图文件会分组到以每个应用的控制器命名的文件夹中。此文件夹存储在应用根目录的“Views”文件夹中：

主页控制器由“Views”文件夹内的“Home”文件夹表示。“Home”文件夹包含“关于”、“联系人”和“索引”（主页）网页的视图。用户请求这三个网页中的一个时，主页控制器中的控制器操作决定使用三个视图中的哪一个来生成网页并将其返回给用户。

使用布局提供一致的网页部分并减少代码重复。布局通常包含页眉、导航和菜单元素以及页脚。页眉和页脚通常包含许多元数据元素的样板标记以及脚本和样式资产的链接。布局有助于在视图中避免这种样板标记。

分部视图通过管理视图的可重用部分来减少代码重复。例如，分部视图可用于在多个视图中出现的博客网站上的作者简介。作者简介是普通的视图内容，不需要执行代码就能生成网页的内容。可以仅通过模型绑定查看作者简介内容，因此使用这种内容类型的分部视图是理想的选择。

视图组件与分部视图的相似之处在于它们可以减少重复性代码，但视图组件还适用于需要在服务器上运行代码才能呈现网页的视图内容。呈现的内容需要数据库交互时（例如网站购物车），视图组件非常有用。为了生成网页输出，视图组件不局限于模型绑定。

使用视图的好处

视图可帮助在 MVC 应用内建立关注点分离 (SoC) 设计，方法是分隔用户界面标记与应用的其他部分。采用 SoC 设计可使应用模块化，从而提供以下几个好处：

- 应用组织地更好，因此更易于维护。视图一般按应用功能进行分组。这使得在处理功能时更容易找到相关的视图。
- 应用的若干部分是松散耦合的。可以生成和更新独立于业务逻辑和数据访问组件的应用视图。可以修改应用的视图，而不必更新应用的其他部分。
- 因为视图是独立的单元，所以更容易测试应用的用户界面部分。
- 由于应用组织得更好，因此你不太可能会意外重复用户界面的各个部分。

创建视图

在 Views / [ControllerName] 文件夹中创建特定于控制器的视图。控制器之间共享的视图都将置于 Views/Shared 文件夹。要创建一个视图，请添加新文件，并将其命名为与 .cshtml 文件扩展名相关联的控制器操作的相同名称。要创建与主页控制器中 About 操作相对应的视图，请在 Views/Home 文件夹中创建一个 About.cshtml 文件：

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>Use this area to provide additional information.</p>
```

Razor 标记以 `@` 符号开头。通过将 C# 代码放置在用大括号（`{ ... }`）括住的 Razor 代码块内，运行 C# 语句。有关示例，请参阅上面显示的"About"到 `ViewData["Title"]` 的分配。只需用 `@` 符号来引用值，即可在 HTML 中显示这些值。请参阅上面的 `<h2>` 和 `<h3>` 元素的内容。

以上所示的视图内容只是呈现给用户的整个网页中的一部分。其他视图文件中指定了页面布局的其余部分和视图的其他常见方面。要了解详细信息，请参阅布局主题。

控制器如何指定视图

视图通常以 ViewResult 的形式从操作返回，这是一种 ActionResult 。操作方法可以直接创建并返回 `ViewResult` ，但通常不会这样做。由于大多数控制器均继承自控制器，因此只需使用 `View` 帮助程序方法即可返回 `ViewResult` ：

HomeController.cs

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

此操作返回时，最后一节显示的 About.cshtml 视图呈现为以下网页：

`View` 帮助程序方法有多个重载。可选择指定：

- 要返回的显式视图：

```
return View("Orders");
```

- 要传递给视图的模型：

```
return View(Orders);
```

- 视图和模型：

```
return View("Orders", Orders);
```

视图发现

操作返回一个视图时，会发生称为"视图发现"的过程。此过程基于视图名称确定使用哪个视图文件。

`View` 方法 的默认行为（`return View();`）旨在返回与其从中调用的操作方法同名的视图。例如，控制器的 About `ActionResult` 方法名称用于搜索名为 About.cshtml 的视图文件。运行时首先在 Views/[ControllerName] 文件夹中搜索该视图。如果在此处找不到匹配的视图，则会在"Shared"文件夹中搜索该视图。

用 `return View();` 隐式返回 `ViewResult` 还是用 `return View("<ViewName>");` 将视图名称显式传递给 `View` 方法并不重要。在这两种情况下，视图发现都会按以下顺序搜索匹配的视图文件：

1. Views/[ControllerName]/[ViewName].cshtml
2. Views/Shared/[ViewName].cshtml

可以提供视图文件路径而不提供视图名称。如果使用从应用根目录开始的绝对路径（可选择以"/"或"~/"开头），则须指定 .cshtml 扩展名：

```
return View("Views/Home/About.cshtml");
```

也可使用相对路径在不同目录中指定视图，而无需指定 .cshtml 扩展名。在 `HomeController` 内，可以使用相对路径返回 Manage 视图的 Index 视图：

```
return View("../Manage/Index");
```

同样，可以用"./"前缀来指示当前的控制器特定目录：

```
return View("./About");
```

分部视图和视图组件使用类似（但不完全相同）的发现机制。

可以使用自定义 IViewLocationExpander 自定义如何在应用中定位视图的默认约定。

视图发现依赖于按文件名称查找视图文件。如果基础文件系统区分大小写，则视图名称也可能区分大小写。为了各操作系统的兼容性，请在控制器与操作名称之间，关联视图文件夹与文件名称之间匹配大小写。如果在处理区分大小写的文件系统时遇到无法找到视图文件的错误，请确认请求的视图文件与实际视图文件名称之间的大小写是否匹配。

按照组织视图文件结构的最佳做法，反映控制器、操作和视图之间的关系，实现可维护性和清晰度。

向视图传递数据

使用多种方法将数据传递给视图：

- 强类型数据：viewmodel
- 弱类型数据
  - `ViewData`（`ViewDataAttribute`）
  - `ViewBag`

强类型数据 (viewmodel)

最可靠的方法是在视图中指定模型类型。此模型通常称为 viewmodel。将 viewmodel 类型的实例传递给此操作的视图。

使用 viewmodel 将数据传递给视图可让视图充分利用强类型检查。强类型化（或强类型）意味着每个变量和常量都有明确定义的类型（例如 `string`、`int` 或 `DateTime`）。在编译时检查视图中使用的类型是否有效。

Visual Studio 和 Visual Studio Code 列出了使用 IntelliSense 功能的强类型类成员。如果要查看 viewmodel 的属性，请键入 viewmodel 的变量名称，后跟句点（`.`）。这有助于提高编写代码的速度并降低错误率。

使用 `@model` 指令指定模型。使用带有 `@Model` 的模型：

```
@model WebApplication1.ViewModels.Address

<h2>Contact</h2>
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

为了将模型提供给视图，控制器将其作为参数进行传递：

```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    var viewModel = new Address()
    {
        Name = "Microsoft",
        Street = "One Microsoft Way",
        City = "Redmond",
        State = "WA",
        PostalCode = "98052-6399"
    };

    return View(viewModel);
}
```

没有针对可以提供给视图的模型类型的限制。建议使用普通旧 CLR 对象 (POCO) viewmodel，它几乎没有已定义的行为（方法）。通常，viewmodel 类要么存储在“Models”文件夹中，要么存储在应用根目录处的单独“ViewModels”文件夹中。上例中使用的 Address viewmodel 是存储在 Address.cs 文件中的 POCO viewmodel：

```
namespace WebApplication1.ViewModels
{
    public class Address
    {
        public string Name { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
    }
}
```

可随意对 viewmodel 类型和业务模型类型使用相同的类。但是，使用单独的模型可使视图独立于应用的业务逻辑和数据访问部分。模型为用户发送给应用的数据使用模型绑定和验证时，模型和 viewmodel 的分离也会提供安全优势。

弱类型数据（ViewData、ViewData 属性和 ViewBag）

`ViewBag` 在 Razor 页中不可用。

除了强类型视图，视图还可以访问弱类型（也称为松散类型）的数据集合。与强类型不同，弱类型（或松散类型）意味着不显式声明要使用的数据类型。可以使用弱类型数据集合将少量数据传入及传出控制器和视图。

传递数据于...	示例
控制器和视图	用数据填充下拉列表。
视图和布局视图	从视图文件设置布局视图中的 < title>元素内容。
分部视图和视图	基于用户请求的网页显示数据的小组件。

可以通过控制器和视图上的 `ViewData` 或 `ViewBag` 属性来引用此集合。`ViewData` 属性是弱类型对象的字典。`ViewBag` 属性是 `ViewData` 的包装器，为基础 `ViewData` 集合提供动态属性。

`ViewData` 和 `ViewBag` 在运行时进行动态解析。由于它们不提供编译时类型检查，因此使用这两者通常比使用 viewmodel 更容易出错。出于上述原因，一些开发者希望尽量减少或根本不使用 `ViewData` 和 `ViewBag`。

ViewData

`ViewData` 是通过 `string` 键访问的 ViewDataDictionary 对象。字符串数据可以直接存储和使用，而不需要强制转换，但是在提取其他 `ViewData` 对象值时必须将其强制转换为特定类型。可以使用 `ViewData` 将数据从控制器传递到视图，以及在视图（包括分部视图和布局）内传递数据。

以下是在操作中使用 `ViewData` 设置问候语和地址值的示例：

```
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello";
    ViewData["Address"]  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

在视图中处理数据：

```
@{
    // Since Address isn't a string, it requires a cast.
    var address = ViewData["Address"] as Address;
}

@ViewData["Greeting"] World!

<address>
    @address.Name<br>
    @address.Street<br>
    @address.City, @address.State @address.PostalCode
</address>
```

ViewData 特性

另一种会使用 ViewDataDictionary 的方法是 ViewDataAttribute。控制器或 Razor 页面模型上使用 `[ViewData]` 修饰的属性将其值存储在字典中并从此处进行加载。

在下面的示例中，“主页”控制器包含使用 `[ViewData]` 修饰的 `Title` 属性。`About` 方法设置“关于”视图的标题：

```
public class HomeController : Controller
{
    [ViewData]
    public string Title { get; set; }

    public IActionResult About()
    {
        Title = "About Us";
        ViewData["Message"] = "Your application description page.";

        return View();
    }
}
```

在“关于”视图中，以模型属性的形式访问 `Title` 属性：

```
<h1>@Model.Title</h1>
```

在布局中，从 ViewData 字典读取标题：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
```

ViewBag

`ViewBag` 在 Razor 页中不可用。

`ViewBag` 是 DynamicViewData 对象，可提供对存储在 `ViewData` 中的对象的动态访问。`ViewBag` 不需要强制转换，因此使用起来更加方便。下例演示如何使用与上述 `ViewData` 有相同结果的 `ViewBag`：

```
public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";
    ViewBag.Address  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

```
@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br>
    @ViewBag.Address.Street<br>
    @ViewBag.Address.City, @ViewBag.Address.State @ViewBag.Address.PostalCode
</address>
```

同时使用 ViewData 和 ViewBag

`ViewBag` 在 Razor 页中不可用。

由于 `ViewData` 和 `ViewBag` 引用相同的基础 `ViewData` 集合，因此在读取和写入值时，可以同时使用 `ViewData` 和 `ViewBag`，并在两者之间进行混合和匹配。

在 About.cshtml 视图顶部，使用 `ViewBag` 设置标题并使用 `ViewData` 设置说明：

```
@{
    Layout = "/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "About Contoso";
    ViewData["Description"] = "Let us tell you about Contoso's philosophy and mission.";
}
```

读取属性，但反向使用 `ViewData` 和 `ViewBag`。在 _Layout.cshtml 文件中，使用 `ViewData` 获取标题并使用 `ViewBag` 获取说明：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"]</title>
    <meta name="description" content="@ViewBag.Description">
    ...
```

请记住，字符串不需要为 `ViewData` 进行强制转换。可以使用 `@ViewData["Title"]` 而不需要强制转换。

可同时使用 `ViewData` 和 `ViewBag` 也可混合和匹配读取及写入属性。呈现以下标记：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>About Contoso</title>
    <meta name="description" content="Let us tell you about Contoso's philosophy and mission.">
    ...
```

ViewData 和 ViewBag 之间差异的摘要

`ViewBag` 在 Razor 页中不可用。

- `ViewData`
  - 派生自 ViewDataDictionary，因此它有可用的字典属性，如 `ContainsKey`、`Add`、`Remove` 和 `Clear`。
  - 字典中的键是字符串，因此允许有空格。示例：`ViewData["Some Key With Whitespace"]`
  - 任何非 `string` 类型均须在视图中进行强制转换才能使用 `ViewData`。
- `ViewBag`
  - 派生自 DynamicViewData，因此它可使用点表示法（`@ViewBag.SomeKey = <value or object>`）创建动态属性，且无需强制转换。`ViewBag` 的语法使添加到控制器和视图的速度更快。
  - 更易于检查 NULL 值。示例：`@ViewBag.Person?.Name`

何时使用 ViewData 或 ViewBag

`ViewData` 和 `ViewBag` 都是在控制器和视图之间传递少量数据的有效方法。根据偏好选择使用哪种方法。可以混合和匹配 `ViewData` 和 `ViewBag` 对象，但是，使用一致的方法可以更轻松地读取和维护代码。这两种方法都是在运行时进行动态解析的，因此容易造成运行时错误。因而，一些开发团队会避免使用它们。

动态视图

不使用 `@model` 声明模型类型，但有模型实例传递给它们的视图（如 `return View(Address);`）可动态引用实例的属性：

```
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

此功能提供了灵活性，但不提供编译保护或 IntelliSense。如果属性不存在，则网页生成在运行时会失败。

更多视图功能

标记帮助程序可以轻松地将服务器端行为添加到现有的 HTML 标记。使用标记帮助程序可避免在视图内编写自定义代码或帮助程序。标记帮助程序作为属性应用于 HTML 元素，并被无法处理它们的编辑器忽略。这可让你在各种工具中编辑和呈现视图标记。

通过许多内置 HTML 帮助程序可生成自定义 HTML 标记。通过视图组件可以处理更复杂的用户界面逻辑。视图组件提供的 SoC 与控制器和视图提供的相同。它们无需使用处理数据（由常见用户界面元素使用）的操作和视图。

与 ASP.NET Core 的许多其他方面一样，视图支持依赖关系注入，可将服务注入视图。

ASP.NET Core 中的分部视图

2018/10/30 • • Edit Online

作者：Steve Smith、Luke Latham、Maher JENDOUBI、Rick Anderson 和 Scott Sauber

分部视图是 Razor 标记文件 (.cshtml)，它在另一个标记文件呈现的输出中呈现 HTML 输出。

在开发 MVC 应用程序（其中标记文件称为“视图”）或 Razor Pages 应用程序（其中标记文件称为“页”）时，均会使用术语“分部视图”。本主题通常将 MVC 视图和 Razor Pages 页面称为“标记文件”。

查看或下载示例代码（如何下载）

何时使用分部视图

分部视图是执行下列操作的有效方式：

- 将大型标记文件分解为更小的组件。

  在由多个逻辑部分组成的大型复杂标记文件中，在分部视图中处理隔开的每个部分是有利的。标记文件中的代码是可管理的，因为标记仅包含整体页面结构和对分部视图的引用。

- 减少跨标记文件中常见标记内容的重复。

  当在标记文件中使用相同的标记元素时，分部视图会将重复的标记内容移到一个分部视图文件中。在分部视图中更改标记后，它会更新使用该分部视图的标记文件呈现的输出。

不应使用分部视图来维护常见布局元素。常见布局元素应在 _Layout.cshtml 文件中指定。

请勿使用需要复杂呈现逻辑或代码执行来呈现标记的分部视图。使用视图组件而不是分部视图。

声明分部视图

分部视图是在 Views 文件夹 (MVC) 或 Pages 文件夹 (Razor Pages) 中维护的 .cshtml 标记文件。

在 ASP.NET Core MVC 中，控制器的 ViewResult 能够返回视图或分部视图。为 ASP.NET Core 2.2 中的 Razor Pages 规划了类似功能。在 Razor Pages 中，PageModel 可以返回 PartialViewResult。引用分部视图部分介绍了引用和呈现分部视图。

与 MVC 视图或页面呈现不同，分部视图不会运行 _ViewStart.cshtml。有关 _ViewStart.cshtml 的详细信息，请参阅 ASP.NET Core 中的布局。

分部视图的文件名通常以下划线 (`_`) 开头。虽然未强制要求遵从此命名约定，但它有助于直观地将分部视图与视图和页面区分开来。

分部视图是在 Views 文件夹中维护的 .cshtml 标记文件。

控制器的 ViewResult 能够返回视图或分部视图。

与 MVC 视图呈现不同，分部视图不会运行 _ViewStart.cshtml。有关 _ViewStart.cshtml 的详细信息，请参阅 ASP.NET Core 中的布局。

分部视图的文件名通常以下划线 (`_`) 开头。虽然未强制要求遵从此命名约定，但它有助于直观地将分部视图与视图区分开来。

引用分部视图

在标记文件中，有多种方法可引用分部视图。我们建议应用程序使用以下异步呈现方法之一：

- 部分标记帮助程序
- 异步 HTML 帮助程序

在标记文件中，有两种方法可引用分部视图：

- 异步 HTML 帮助程序
- 同步 HTML 帮助程序

我们建议应用程序使用异步 HTML 帮助程序。

分部标记帮助程序

分部标记帮助程序要求 ASP.NET Core 2.1 或更高版本。

分部标记帮助程序会异步呈现内容并使用类似 HTML 的语法：

```
<partial name="_PartialName" />
```

当存在文件扩展名时，标记帮助程序会引用分部视图，该视图必须与调用分部视图的标记文件位于同一文件夹中：

```
<partial name="_PartialName.cshtml" />
```

以下示例从应用程序根目录引用分部视图。以波形符斜杠（`~/`）或斜杠（`/`）开头的路径指代应用程序根目录：

Razor 页面

```
<partial name="~/Pages/Folder/_PartialName.cshtml" />
<partial name="/Pages/Folder/_PartialName.cshtml" />
```

MVC

```
<partial name="~/Views/Folder/_PartialName.cshtml" />
<partial name="/Views/Folder/_PartialName.cshtml" />
```

以下示例引用使用相对路径的分部视图：

```
<partial name="../Account/_PartialName.cshtml" />
```

有关更多信息，请参见ASP.NET Core 中的部分标记帮助程序。

异步 HTML 帮助程序

使用 HTML 帮助程序时，最佳做法是使用 PartialAsync。`PartialAsync` 返回包含在 Task<TResult> 中的 IHtmlContent 类型。通过在等待的调用前添加 `@` 字符前缀来引用该方法：

```
@await Html.PartialAsync("_PartialName")
```

当存在文件扩展名时，HTML 帮助程序会引用分部视图，该视图必须与调用分部视图的标记文件位于同一文件夹中：

```
@await Html.PartialAsync("_PartialName.cshtml")
```

以下示例从应用程序根目录引用分部视图。以波形符斜杠（`~/`）或斜杠（`/`）开头的路径指代应用程序根目录：

Razor 页面

```
@await Html.PartialAsync("~/Pages/Folder/_PartialName.cshtml")
@await Html.PartialAsync("/Pages/Folder/_PartialName.cshtml")
```

MVC

```
@await Html.PartialAsync("~/Views/Folder/_PartialName.cshtml")
@await Html.PartialAsync("/Views/Folder/_PartialName.cshtml")
```

以下示例引用使用相对路径的分部视图：

```
@await Html.PartialAsync("../Account/_LoginPartial.cshtml")
```

或者，也可以使用 RenderPartialAsync 呈现分部视图。此方法不返回 IHtmlContent。它将呈现的输出直接流式传输到响应。因为该方法不返回结果，所以必须在 Razor 代码块内调用它：

```
@{
    await Html.RenderPartialAsync("_AuthorPartial");
}
```

由于 `RenderPartialAsync` 流式传输呈现的内容，因此在某些情况下它可提供更好的性能。在性能起关键作用的情况下，使用两种方法对页面进行基准测试，并使用生成更快响应的方法。

同步 HTML 帮助程序

Partial 和 RenderPartial 分别是 `PartialAsync` 和 `RenderPartialAsync` 的同步等效项。但不建议使用同步等效项，因为可能会出现死锁的情况。同步方法针对以后版本中的删除功能。

> **IMPORTANT**
>
> 如果需要执行代码，请使用视图组件，而不是使用分部视图。

调用 `Partial` 或 `RenderPartial` 会导致 Visual Studio 分析器警告。例如，使用 `Partial` 会产生以下警告消息：

> 使用 IHtmlHelper.Partial 可能会导致应用程序死锁。考虑使用 <分部> 标记帮助程序或 IHtmlHelper.PartialAsync。

将对 `@Html.Partial` 的调用替换为 `@await Html.PartialAsync` 或分部标记帮助程序。有关分部标记帮助程序迁移的详细信息，请参阅从 HTML 帮助程序迁移。

分部视图发现

如果按名称（无文件扩展名）引用分部视图，则按所述顺序搜索以下位置：

Razor 页面

1. 当前正在执行页面的文件夹
2. 该页面文件夹上方的目录图
3. `/Shared`

4. `/Pages/Shared`
5. `/Views/Shared`

MVC

1. `/Areas/<Area-Name>/Views/<Controller-Name>`
2. `/Areas/<Area-Name>/Views/Shared`
3. `/Views/Shared`
4. `/Pages/Shared`

1. `/Areas/<Area-Name>/Views/<Controller-Name>`
2. `/Areas/<Area-Name>/Views/Shared`
3. `/Views/Shared`

以下约定适用于分部视图发现：

- 当分部视图位于不同的文件夹中时，允许使用具有相同文件名的不同分部视图。
- 当按名称（无文件扩展名）引用分部视图且分部视图出现在调用方的文件夹和 文件夹中时，调用方文件夹中的分部视图会提供分部视图。如果调用方文件夹中不存在分部视图，则会从 文件夹中提供分部视图。文件夹中的分部视图称为“共享分部视图”或“默认分部视图”。
- 可以链接分部视图—如果调用没有形成循环引用，则分部视图可以调用另一个分部视图。相对路径始终相对于当前文件，而不是相对于文件的根视图或父视图。

> **NOTE**
> 分部视图中定义的 Razor `section` 对父标记文件不可见。`section` 仅对定义它时所在的分部视图可见。

通过分部视图访问数据

实例化分部视图时，它会获得父视图的 `ViewData` 字典的副本。在分部视图内对数据所做的更新不会保存到父视图中。在分部视图中的 `ViewData` 更改会在分部视图返回时丢失。

以下示例演示如何将 ViewDataDictionary的实例传递给分部视图：

```
@await Html.PartialAsync("_PartialName", customViewData)
```

还可将模型传入分部视图。模型可以是自定义对象。你可以使用 `PartialAsync`（向调用方呈现内容块）或 `RenderPartialAsync`（将内容流式传输到输出）传递模型：

```
@await Html.PartialAsync("_PartialName", model)
```

Razor 页面

示例应用程序中的以下标记来自 *Pages/ArticlesRP/ReadRP.cshtml* 页面。此页包含两个分部视图。第二个分部视图将模型和 `ViewData` 传入分部视图。`ViewDataDictionary` 构造函数重载可用于传递新 `ViewData` 字典，同时保留现有的 `ViewData` 字典。

```
@model ReadRPModel

<h2>@Model.Article.Title</h2>
@* Pass the author's name to Pages\Shared\_AuthorPartialRP.cshtml *@
@await Html.PartialAsync("../Shared/_AuthorPartialRP", Model.Article.AuthorName)
@Model.Article.PublicationDate

@* Loop over the Sections and pass in a section and additional ViewData to
   the strongly typed Pages\ArticlesRP\_ArticleSectionRP.cshtml partial view. *@
@{
    var index = 0;

    @foreach (var section in Model.Article.Sections)
    {
        @await Html.PartialAsync("_ArticleSectionRP", section,
                                 new ViewDataDictionary(ViewData)
                                 {
                                     { "index", index }
                                 })

        index++;
    }
}
```

Pages/Shared/_AuthorPartialRP.cshtml 是 ReadRP.cshtml 标记文件引用的第一个分部视图：

```
@model string
<div>
    <h3>@Model</h3>
    This partial view from /Pages/Shared/_AuthorPartialRP.cshtml.
</div>
```

Pages/ArticlesRP/_ArticleSectionRP.cshtml 是 ReadRP.cshtml 标记文件引用的第二个分部视图：

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title Index: @ViewData["index"]</h3>
<div>
    @Model.Content
</div>
```

MVC

示例应用中的以下标记显示 Views/Articles/Read.cshtml 视图。此视图包含两个分部视图。第二个分部视图将模型和 `ViewData` 传入分部视图。`ViewDataDictionary` 构造函数重载可用于传递新 `ViewData` 字典，同时保留现有的 `ViewData` 字典。

```
@model PartialViewsSample.ViewModels.Article

<h2>@Model.Title</h2>
@* Pass the author's name to Views\Shared\_AuthorPartial.cshtml *@
@await Html.PartialAsync("_AuthorPartial", Model.AuthorName)
@Model.PublicationDate

@* Loop over the Sections and pass in a section and additional ViewData to
   the strongly typed Views\Articles\_ArticleSection.cshtml partial view. *@
@{
    var index = 0;

    @foreach (var section in Model.Sections)
    {
        @await Html.PartialAsync("_ArticleSection", section,
                                 new ViewDataDictionary(ViewData)
                                 {
                                     { "index", index }
                                 })

        index++;
    }
}
```

Views/Shared/_AuthorPartial.cshtml 是 ReadRP.cshtml 标记文件引用的第一个分部视图：

```
@model string
<div>
    <h3>@Model</h3>
    This partial view from /Views/Shared/_AuthorPartial.cshtml.
</div>
```

Views/Articles/_ArticleSection.cshtml 是 Read.cshtml 标记文件引用的第二个分部视图：

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title Index: @ViewData["index"]</h3>
<div>
    @Model.Content
</div>
```

在运行时，分部视图在父标记文件呈现的输出中呈现，而父标记文件本身在共享的 _Layout.cshtml 内呈现。第一个分部视图呈现文章作者的姓名和发布日期：

> Abraham Lincoln
>
> 来自 <共享分部视图文件路径> 的分部视图。1863 年 11 月 19 日中午 12:00:00

第二个分部视图呈现文章的各部分：

> 第一节索引：0
>
> 八十七年前…
>
> 第二节索引：1
>
> 如今，我们正在进行一场伟大的内战，考验着……
>
> 第三节索引：2

> 然而，从更广泛的意义上说，我们无法奉献...

其他资源

- ASP.NET Core 的 Razor 语法参考
- ASP.NET Core 中的标记帮助程序
- ASP.NET Core 中的部分标记帮助程序
- ASP.NET Core 中的视图组件
- ASP.NET Core 中的区域

- ASP.NET Core 的 Razor 语法参考
- ASP.NET Core 中的视图组件
- ASP.NET Core 中的区域

在 ASP.NET Core MVC 中使用控制器处理请求

2018/6/25 • • Edit Online

作者：Steve Smith 和 Scott Addie

控制器、操作和操作结果是开发人员如何使用 ASP.NET Core MVC 生成应用的一个基本组成部分。

什么是控制器？

控制器用于对一组操作进行定义和分组。操作（或操作方法）是控制器上一种用来处理请求的方法。控制器按逻辑将类似的操作集合到一起。通过这种操作的聚合，可以共同应用路由、缓存和授权等通用规则集。请求通过路由映射到操作。

根据惯例，控制器类：

- 驻留在项目的根级别“Controllers”文件夹中
- 继承自 `Microsoft.AspNetCore.Mvc.Controller`

控制器是一个可实例化的类，其中下列条件至少某一个为 true：

- 类名称带有“Controller”后缀
- 该类继承自带有“Controller”后缀的类
- 使用 `[Controller]` 属性修饰该类

控制器类不可含有关联的 `[NonController]` 属性。

控制器应遵循 Explicit Dependencies Principle（显式依赖关系原则）。以下几种方法可以实现此原则。如果多个控制器操作需要相同的服务，请考虑使用构造函数注入来请求这些依赖关系。如果该服务仅需要一个操作方法，请考虑使用操作注入来请求依赖关系。

在“模型-视图-控制器”模式中，控制器负责请求的初始处理和模型的实例化操作。通常情况下，应在模型中执行业务决策。

控制器获取模型处理的结果（如果有），并返回正确的视图及其关联的视图数据或 API 调用的结果。请参阅 ASP.NET Core MVC 概述以及 ASP.NET Core MVC 和 Visual Studio 入门了解详细信息。

控制器是一个 UI 级别的抽象。它的职责是确保请求的数据有效，并选择应当返回的视图（或 API 的结果）。在构造良好的应用程序中，它不会直接包括数据访问或业务逻辑。相反，控制器会委托给处理这些责任的服务。

定义操作

控制器上的公共方法（除了那些使用 `[NonAction]` 属性装饰的方法）均为操作。操作上的参数会绑定到请求数据，并使用模型绑定进行验证。所有模型绑定的内容都会执行模型验证。`ModelState.IsValid` 属性值指示模型绑定和验证是否成功。

操作方法应包含用于将请求映射到某个业务关注点的逻辑。业务关注点通常应当表示为控制器通过依赖关系注入来访问的服务。然后，操作将业务操作的结果映射到应用程序状态。

操作可以返回任何内容，但是经常返回生成响应的 `IActionResult`（或异步方法的 `Task<IActionResult>`）的实例。操作方法负责选择响应的类型。操作结果会做出响应。

控制器帮助程序方法

控制器通常继承自控制器（尽管没有要求）。派生自 `Controller` 会提供对三个帮助程序方法类别的访问：

1.导致空响应正文的方法

没有包含 `Content-Type` HTTP 响应标头，因为响应正文缺少要描述的内容。

该类别中有两种结果类型：重定向和 HTTP 状态代码。

- **HTTP 状态代码**

  此类型返回 HTTP 状态代码。此类型的几种帮助程序方法是 `BadRequest`、`NotFound` 和 `Ok`。例如，`return BadRequest();` 执行时生成 400 状态代码。重载 `BadRequest``NotFound` 和 `Ok` 等方法时，它们不再符合 HTTP 状态代码响应方的资格，因为正在进行内容协商。

- **重定向**

  此类型（使用 `Redirect``LocalRedirect``RedirectToAction` 或 `RedirectToRoute`）返回一个到操作或目标的重定向。例如，`return RedirectToAction("Complete", new {id = 123});` 重定向到 `Complete`，传递一个匿名对象。

  重定向结果类型与 HTTP 状态代码类型的不同之处主要在于 `Location` HTTP 响应标头的添加。

2.导致含有预定义内容类型的非空响应正文的方法

此类别中的大多数帮助程序方法都包含一个 `ContentType` 属性，通过它可以设置 `Content-Type` 响应标头来描述响应正文。

此类别中有两种结果类型：视图和已格式化的响应。

- 视图

  此类型返回一个使用模型呈现 HTML 的视图。例如，`return View(customer);` 将模型传递给视图以进行数据绑定。

- **已格式化的响应**

  此类型返回 JSON 或类似的数据交换格式，从而以特定方式表示某个对象。例如，`return Json(customer);` 将提供的对象串行化为 JSON 格式。

  此类型的其他常见方法包括 `File`、`PhysicalFile` 和 `VirtualFile`。例如，`return PhysicalFile(customerFilePath, "text/xml");` 返回由 `Content-Type` 响应标头值“text/xml”所描述的 XML 文件。

3.导致在与客户端协商的内容类型中格式化为非空响应正文的方法

此类别更为熟知的说法是“内容协商”。每当操作返回 ObjectResult 类型或除 IActionResult 实现之外的任何实现时，会应用内容协商。返回非 `IActionResult` 实现（例如 `object`）的操作也会返回已格式化的响应。

此类型的一些帮助程序方法包括 `BadRequest`、`CreatedAtRoute` 和 `Ok`。这些方法的示例分别为 `return BadRequest(modelState);`、`return CreatedAtRoute("routename", values, newobject);` 和 `return Ok(value);`。请注意，`BadRequest` 和 `Ok` 仅在传递了值的时候才执行内容协商；在没有传递值的情况下，它们充当 HTTP 状态码结果类型。另一方面，`CreatedAtRoute` 方法始终执行内容协商，因为它的重载均要求传递一个值。

横切关注点

应用程序通常会共享其部分工作流程。示例包括需要身份验证才能访问购物车的应用，或者在某些页面上缓存数据的应用。要在某个操作方法之前或之后执行逻辑，请使用筛选器。在横切关注点上使用筛选器可以减少重复，使它们遵循不要自我重复 (DRY) 原则。

可在控制器或操作级别上应用大多数筛选器属性（例如 `[Authorize]`），具体取决于所需的粒度级别。

错误处理和响应缓存通常是横切关注点：

- 处理错误
- 响应缓存

使用筛选器或自定义中间件可处理许多横切关注点。

在 ASP.NET Core 中路由到控制器操作

2018/10/29 • • Edit Online

作者：Ryan Nowak 和 Rick Anderson

ASP.NET Core MVC 使用路由中间件来匹配传入请求的 URL 并将它们映射到操作。路由在启动代码或属性中定义。路由描述应如何将 URL 路径与操作相匹配。它还用于在响应中生成送出的 URL（用于链接）。

操作既支持传统路由，也支持属性路由。通过在控制器或操作上放置路由可实现属性路由。有关详细信息，请参阅混合路由。

本文档将介绍 MVC 与路由之间的交互，以及典型的 MVC 应用如何使用各种路由功能。有关高级路由的详细信息，请参阅路由。

设置路由中间件

在 *Configure* 方法中，可能会看到与下面类似的代码：

```
app.UseMvc(routes =>
{
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

在对 `UseMvc` 的调用中，`MapRoute` 用于创建单个路由，亦称 `default` 路由。大多数 MVC 应用使用带有模板的路由，与 `default` 路由类似。

路由模板 `"{controller=Home}/{action=Index}/{id?}"` 可以匹配诸如 `/Products/Details/5` 之类的 URL 路径，并通过对路径进行标记来提取路由值 `{ controller = Products, action = Details, id = 5 }`。MVC 将尝试查找名为 `ProductsController` 的控制器并运行 `Details` 操作：

```
public class ProductsController : Controller
{
   public IActionResult Details(int id) { ... }
}
```

请注意，在此示例中，当调用此操作时，模型绑定会使用值 `id = 5` 将 `id` 参数设置为 `5`。有关更多详细信息，请参阅模型绑定。

使用 `default` 路由：

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

路由模板：

- `{controller=Home}` 将 `Home` 定义为默认 `controller`
- `{action=Index}` 将 `Index` 定义为默认 `action`
- `{id?}` 将 `id` 定义为可选参数

默认路由参数和可选路由参数不必包含在 URL 路径中进行匹配。有关路由模板语法的详细说明，请参阅路由模板参考。

`"{controller=Home}/{action=Index}/{id?}"` 可以匹配 URL 路径 `/` 并生成路由值 `{ controller = Home, action = Index }`。`controller` 和 `action` 的值使用默认值，`id` 不生成值，因为 URL 路径中没有相应的段。MVC 使用这些路由值选择 `HomeController` 和 `Index` 操作：

```
public class HomeController : Controller
{
  public IActionResult Index() { ... }
}
```

通过使用此控制器定义和路由模板，将对以下任意 URL 路径执行 `HomeController.Index` 操作：

- `/Home/Index/17`
- `/Home/Index`
- `/Home`
- `/`

简便方法 `UseMvcWithDefaultRoute`：

```
app.UseMvcWithDefaultRoute();
```

可用于替换：

```
app.UseMvc(routes =>
{
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

`UseMvc` 和 `UseMvcWithDefaultRoute` 可向中间件管道添加 `RouterMiddleware` 的实例。MVC 不直接与中间件交互，而是使用路由来处理请求。MVC 通过 `MvcRouteHandler` 实例连接到路由。`UseMvc` 内的代码与下面类似：

```
var routes = new RouteBuilder(app);

// Add connection to MVC, will be hooked up by calls to MapRoute.
routes.DefaultHandler = new MvcRouteHandler(...);

// Execute callback to register routes.
// routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");

// Create route collection and add the middleware.
app.UseRouter(routes.Build());
```

`UseMvc` 不直接定义任何路由，它向 `attribute` 路由的路由集合添加占位符。重载 `UseMvc(Action<IRouteBuilder>)` 则允许用户添加自己的路由，并且还支持属性路由。`UseMvc` 及其所有变体都会为属性路由添加占位符：无论如何配置 `UseMvc`，属性路由始终可用。`UseMvcWithDefaultRoute` 定义默认路由并支持属性路由。属性路由部分提供了有关属性路由的更多详细信息。

传统路由

`default` 路由：

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

是一种*传统路由*。将这种样式称为*传统路由*的原因在于，它为 URL 路径设立了一个*约定*:

- 第一个路径段映射到控制器名称
- 第二段映射到操作名称。
- 第三段用于可选 `id`（用于映射到模型实体）

使用此 `default` 路由时，URL 路径 `/Products/List` 映射到 `ProductsController.List` 操作，`/Blog/Article/17` 映射到 `BlogController.Article`。此映射仅基于控制器和操作名称，而不基于命名空间、源文件位置或方法参数。

> **TIP**
> 使用默认路由进行传统路由时，可快速生成应用程序，无需为所定义的每项操作提供一个新的 URL 模式。对于包含 CRUD 样式操作的应用程序，通过保持各控制器间 URL 的一致性，可帮助简化代码，使 UI 更易预测。

> **WARNING**
> 路由模板将 `id` 定义为可选参数，意味着无需在 URL 中提供 ID 也可执行操作。从 URL 中省略 `id` 通常会导致模型绑定将它设置为 `0`，进而导致在数据库中找不到与 `id == 0` 匹配的实体。属性路由可以提供细化控制，使某些操作需要 ID，某些操作不需要 ID。按照惯例，当可选参数（比如 `id`）有可能在正确的用法中出现时，本文档将涵盖这些参数。

多个路由

通过添加对 `MapRoute` 的多次调用，可以在 `UseMvc` 内添加多个路由。这样做可以定义多个约定，或添加专用于特定操作的传统路由，比如:

```
app.UseMvc(routes =>
{
   routes.MapRoute("blog", "blog/{*article}",
            defaults: new { controller = "Blog", action = "Article" });
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

此处的 `blog` 路由是一个*专用的传统路由*，这表示它使用传统路由系统，但专用于特定的操作。由于 `controller` 和 `action` 不会在路由模板中作为参数显示，它们只能有默认值，因此，此路由将始终映射到 `BlogController.Article` 操作。

路由集合中的路由会进行排序，并按添加顺序进行处理。因此，在此示例中，将先尝试 `blog` 路由，再尝试 `default` 路由。

> **NOTE**
> *专用传统路由*通常使用 catch-all 路由参数（比如 `{*article}`）来捕获 URL 路径的剩余部分。这会使某个路由变得"太贪婪"，也就是说，它会匹配用户想要使用其他路由来匹配的 URL。将"贪婪的"路由放在路由表中靠后的位置可解决此问题。

回退

在处理请求时，MVC 将验证路由值能否用于在应用程序中查找控制器和操作。如果路由值与任何操作都不匹配，则将该路由视为不匹配，并尝试下一个路由。这称为*回退*，其目的是简化传统路由重叠的情况。

区分操作

当通过路由匹配到两项操作时，MVC 必须进行区分，以选择"最佳"候选项，否则会引发异常。例如:

```
public class ProductsController : Controller
{
   public IActionResult Edit(int id) { ... }

   [HttpPost]
   public IActionResult Edit(int id, Product product) { ... }
}
```

此控制器定义了两项操作，这两项操作均与 URL 路径 `/Products/Edit/17` 和路由数据 `{ controller = Products, action = Edit, id = 17 }` 匹配。这是 MVC 控制器的典型模式，其中 `Edit(int)` 显示用于编辑产品的表单，`Edit(int, Product)` 处理已发布的表单。为此，MVC 需要在请求为 HTTP `POST` 时选择 `Edit(int, Product)`，在 Http 谓词为任何其他内容时选择 `Edit(int)`。

`HttpPostAttribute`（`[HttpPost]`）是 `IActionConstraint` 的实现，它仅允许执行当 Http 谓词为 `POST` 时选择的操作。`IActionConstraint` 的存在使 `Edit(int, Product)` 成为比 `Edit(int)` "更好"的匹配项，因此会先尝试 `Edit(int, Product)`。

只需在特殊化方案中编写自定义 `IActionConstraint` 实现，但务必了解 `HttpPostAttribute` 等属性的角色 — 为其他 Http 谓词定义了类似的属性。在传统路由中，当操作属于 `show form -> submit form` 工作流时通常使用相同的操作名称。在阅读了解 IActionConstraint 部分后，此模式的便利性将变得更加明显。

如果匹配多个路由，但 MVC 找不到"最佳"路由，则会引发 `AmbiguousActionException`。

路由名称

以下示例中的字符串 `"blog"` 和 `"default"` 都是路由名称：

```
app.UseMvc(routes =>
{
   routes.MapRoute("blog", "blog/{*article}",
               defaults: new { controller = "Blog", action = "Article" });
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

路由名称为路由提供一个逻辑名称，以便使用命名路由来生成 URL。路由排序会使 URL 生成复杂化，而这极大地简化了 URL 创建。路由名称必须在应用程序范围内唯一。

路由名称不影响请求的 URL 匹配或处理；它们仅用于 URL 生成。路由提供了有关 URL 生成（包括 MVC 特定帮助程序中的 URL 生成）的更多详细信息。

属性路由

属性路由使用一组属性将操作直接映射到路由模板。在下面的示例中，`Configure` 方法使用 `app.UseMvc();`，不传递任何路由。`HomeController` 将匹配一组 URL，这组 URL 与默认路由 `{controller=Home}/{action=Index}/{id?}` 匹配的 URL 类似：

```
public class HomeController : Controller
{
   [Route("")]
   [Route("Home")]
   [Route("Home/Index")]
   public IActionResult Index()
   {
      return View();
   }
   [Route("Home/About")]
   public IActionResult About()
   {
      return View();
   }
   [Route("Home/Contact")]
   public IActionResult Contact()
   {
      return View();
   }
}
```

将针对任意 URL 路径 `/`、`/Home` 或 `/Home/Index` 执行 `HomeController.Index()` 操作。

> **NOTE**
>
> 此示例重点介绍属性路由与传统路由之间的主要编程差异。属性路由需要更多输入来指定路由；传统的默认路由处理路由的方式则更简洁。但是，属性路由允许（并需要）精确控制应用于每项操作的路由模板。

使用属性路由时，控制器名称和操作名称对于操作的选择**没有**影响。此示例匹配的 URL 与上一示例相同。

```
public class MyDemoController : Controller
{
   [Route("")]
   [Route("Home")]
   [Route("Home/Index")]
   public IActionResult MyIndex()
   {
      return View("Index");
   }
   [Route("Home/About")]
   public IActionResult MyAbout()
   {
      return View("About");
   }
   [Route("Home/Contact")]
   public IActionResult MyContact()
   {
      return View("Contact");
   }
}
```

> **NOTE**
>
> 上述路由模板未定义 `action`、`area` 和 `controller` 的路由参数。事实上，属性路由中不允许使用这些路由参数。由于路由模板已与某项操作关联，因此，分析 URL 中的操作名称毫无意义。

使用 Http[Verb] 属性的属性路由

属性路由还可以使用 `Http[Verb]` 属性，比如 `HttpPostAttribute`。所有这些属性都可采用路由模板。此示例

展示与同一路由模板匹配的两项操作：

```
[HttpGet("/products")]
public IActionResult ListProducts()
{
   // ...
}

[HttpPost("/products")]
public IActionResult CreateProduct(...)
{
   // ...
}
```

对于诸如 `/products` 之类的 URL 路径，当 Http 谓词为 `GET` 时将执行 `ProductsApi.ListProducts` 操作，当 Http 谓词为 `POST` 时将执行 `ProductsApi.CreateProduct`。属性路由首先将 URL 与路由属性定义的路由模板集进行匹配。一旦某个路由模板匹配，就会应用 `IActionConstraint` 约束来确定可以执行的操作。

> **TIP**
> 生成 REST API 时，很少会在操作方法上使用 `[Route(...)]`。建议使用更特定的 `Http*Verb*Attributes` 来明确 API 所支持的操作。REST API 的客户端需要知道映射到特定逻辑操作的路径和 Http 谓词。

由于属性路由适用于特定操作，因此，使参数变成路由模板定义中的必需参数很简单。在此示例中，`id` 是 URL 路径中的必需参数。

```
public class ProductsApiController : Controller
{
   [HttpGet("/products/{id}", Name = "Products_List")]
   public IActionResult GetProduct(int id) { ... }
}
```

将针对诸如 `/products/3`（而非 `/products`）之类的 URL 路径执行 `ProductsApi.GetProduct(int)` 操作。请参阅路由了解路由模板和相关选项的完整说明。

路由名称

以下代码定义 `Products_List` 的*路由名称*：

```
public class ProductsApiController : Controller
{
   [HttpGet("/products/{id}", Name = "Products_List")]
   public IActionResult GetProduct(int id) { ... }
}
```

可以使用路由名称基于特定路由生成 URL。路由名称不影响路由的 URL 匹配行为，仅用于生成 URL。路由名称必须在应用程序范围内唯一。

> **NOTE**
> 这一点与传统的*默认路由*相反，后者将 `id` 参数定义为可选参数（`{id?}`）。这种精确指定 API 的功能可带来一些好处，比如允许将 `/products` 和 `/products/5` 分派到不同的操作。

合并路由

若要使属性路由减少重复，可将控制器上的路由属性与各个操作上的路由属性合并。控制器上定义的所有路

由模板均作为操作上路由模板的前缀。在控制器上放置路由属性会使控制器中的**所有**操作都使用属性路由。

```
[Route("products")]
public class ProductsApiController : Controller
{
   [HttpGet]
   public IActionResult ListProducts() { ... }

   [HttpGet("{id}")]
   public ActionResult GetProduct(int id) { ... }
}
```

在此示例中，URL 路径 `/products` 可以匹配 `ProductsApi.ListProducts`，URL 路径 `/products/5` 可以匹配 `ProductsApi.GetProduct(int)`。这两项操作仅匹配 HTTP `GET`，因为它们用 `HttpGetAttribute` 修饰。

应用于操作的以 `/` 开头的路由模板不与应用于控制器的路由模板合并。此示例匹配一组与*默认路由*类似的 URL 路径。

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("")]      // Combines to define the route template "Home"
    [Route("Index")] // Combines to define the route template "Home/Index"
    [Route("/")]     // Doesn't combine, defines the route template ""
    public IActionResult Index()
    {
        ViewData["Message"] = "Home index";
        var url = Url.Action("Index", "Home");
        ViewData["Message"] = "Home index" + "var url = Url.Action; =  " + url;
        return View();
    }

    [Route("About")] // Combines to define the route template "Home/About"
    public IActionResult About()
    {
        return View();
    }
}
```

对属性路由排序

与按照已定义顺序执行的传统路由相反，属性路由会生成树，并同时匹配所有路由。其行为就像路由条目是以理想排序方式放置的一样；最特定的路由有机会比较一般的路由先执行。

例如，像 `blog/search/{topic}` 这样的路由比像 `blog/{*article}` 这样的路由更特定。从逻辑上讲，`blog/search/{topic}` 路由默认情况下先"运行"，因为这是唯一合理的排序。使用传统路由时，开发人员负责按所需顺序放置路由。

属性路由可以使用框架提供的所有路由属性的 `Order` 属性来配置顺序。路由按 `Order` 属性的升序进行处理。默认顺序为 `0`。使用 `Order = -1` 设置的路由比未设置顺序的路由先运行。使用 `Order = 1` 设置的路由在默认路由排序后运行。

> **TIP**
>
> 避免依赖 `Order`。如果 URL 空间需要有显式顺序值才能正确进行路由，则同样可能使客户端混淆不清。属性路由通常选择与 URL 匹配的正确路由。如果用于 URL 生成的默认顺序不起作用，使用路由名称作为替代项通常比应用 `Order` 属性更简单。

Razor Pages 路由和 MVC 控制器路由共享一个实现。可在 路由和应用约定：路由顺序中找到有关 Razor

Pages 主题中路由顺序的信息。

路由模板中的标记替换（[controller]、[action]、[area]）

为方便起见，属性路由支持*标记替换*，方法是将标记用大括号（`[`、`]`）括起来。标记 `[action]`、`[area]` 和 `[controller]` 替换为定义了路由的操作中的操作名称值、区域名称值和控制器名称值。在接下来的示例中，操作与注释中所述的 URL 路径匹配：

```
[Route("[controller]/[action]")]
public class ProductsController : Controller
{
    [HttpGet] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("{id}")] // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}
```

标记替换发生在属性路由生成的最后一步。上述示例的行为方式将与以下代码相同：

```
public class ProductsController : Controller
{
    [HttpGet("[controller]/[action]")] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("[controller]/[action]/{id}")] // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}
```

属性路由还可以与继承结合使用。与标记替换结合使用时尤为强大。

```
[Route("api/[controller]")]
public abstract class MyBaseController : Controller { ... }

public class ProductsController : MyBaseController
{
   [HttpGet] // Matches '/api/Products'
   public IActionResult List() { ... }

   [HttpPut("{id}")] // Matches '/api/Products/{id}'
   public IActionResult Edit(int id) { ... }
}
```

标记替换也适用于属性路由定义的路由名称。
`[Route("[controller]/[action]", Name="[controller]_[action]")]` 为每项操作生成一个唯一的路由名称。

若要匹配文本标记替换分隔符 `[` 或 `]`，可通过重复该字符（`[[` 或 `]]`）对其进行转义。

使用参数转换程序自定义标记替换

使用参数转换程序可以自定义标记替换。参数转换程序实现 `IOutboundParameterTransformer` 并转换参数值。例如，一个自定义 `SlugifyParameterTransformer` 参数转换程序可将 `SubscriptionManagement` 路由值更改为

`subscription-management`。

`RouteTokenTransformerConvention` 是应用程序模型约定，可以：

- 将参数转换程序应用到应用程序中的所有属性路由。
- 在替换属性路由标记值时对其进行自定义。

```
public class SubscriptionManagementController : Controller
{
    [HttpGet("[controller]/[action]")] // Matches '/subscription-management/list-all'
    public IActionResult ListAll() { ... }
}
```

`RouteTokenTransformerConvention` 在 `ConfigureServices` 中注册为选项。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new RouteTokenTransformerConvention(
                                    new SlugifyParameterTransformer()));
    });
}

public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string TransformOutbound(object value)
    {
        if (value == null) { return null; }

        // Slugify value
        return Regex.Replace(value.ToString(), "([a-z])([A-Z])", "$1-$2").ToLower();
    }
}
```

多个路由

属性路由支持定义多个访问同一操作的路由。此操作最常用于模拟*默认传统路由*的行为，如以下示例所示：

```
[Route("[controller]")]
public class ProductsController : Controller
{
   [Route("")]     // Matches 'Products'
   [Route("Index")] // Matches 'Products/Index'
   public IActionResult Index()
}
```

在控制器上放置多个路由属性意味着，每个路由属性将与操作方法上的每个路由属性合并。

```
[Route("Store")]
[Route("[controller]")]
public class ProductsController : Controller
{
   [HttpPost("Buy")]     // Matches 'Products/Buy' and 'Store/Buy'
   [HttpPost("Checkout")] // Matches 'Products/Checkout' and 'Store/Checkout'
   public IActionResult Buy()
}
```

当在某个操作上放置多个路由属性（可实现 `IActionConstraint`）时，每个操作约束将与定义它的属性中的路由模板合并。

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
   [HttpPut("Buy")]      // Matches PUT 'api/Products/Buy'
   [HttpPost("Checkout")] // Matches POST 'api/Products/Checkout'
   public IActionResult Buy()
}
```

> **TIP**
> 在操作上使用多个路由可能看起来很强大，但更建议使应用程序的 URL 空间保持简洁且定义完善。仅在需要时，例如为了支持现有客户端，才在操作上使用多个路由。

指定属性路由的可选参数、默认值和约束

属性路由支持使用与传统路由相同的内联语法，来指定可选参数、默认值和约束。

```
[HttpPost("product/{id:int}")]
public IActionResult ShowProduct(int id)
{
   // ...
}
```

有关路由模板语法的详细说明，请参阅路由模板参考。

使用 `IRouteTemplateProvider` 的自定义路由属性

该框架中提供的所有路由属性（`[Route(...)]`、`[HttpGet(...)]` 等）都可实现 `IRouteTemplateProvider` 接口。当应用启动时，MVC 会查找控制器类和操作方法上的属性，并使用可实现 `IRouteTemplateProvider` 的属性生成一组初始路由。

用户可以实现 `IRouteTemplateProvider` 来定义自己的路由属性。每个 `IRouteTemplateProvider` 都允许定义一个包含自定义路由模板、顺序和名称的路由：

```
public class MyApiControllerAttribute : Attribute, IRouteTemplateProvider
{
   public string Template => "api/[controller]";

   public int? Order { get; set; }

   public string Name { get; set; }
}
```

应用 `[MyApiController]` 时，上述示例中的属性会自动将 `Template` 设置为 `"api/[controller]"`。

使用应用程序模型自定义属性路由

*应用程序模型*是一个在启动时创建的对象模型，MVC 可使用其中的所有元数据来路由和执行操作。*应用程序模型*包含从路由属性收集（通过 `IRouteTemplateProvider`）的所有数据。可通过编写*约定*在启动时修改应用程序模型，以便自定义路由的行为方式。此部分通过一个简单的示例说明了如何使用应用程序模型自定义路由。

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;
using System.Text;
public class NamespaceRoutingConvention : IControllerModelConvention
{
    private readonly string _baseNamespace;

    public NamespaceRoutingConvention(string baseNamespace)
    {
        _baseNamespace = baseNamespace;
    }

    public void Apply(ControllerModel controller)
    {
        var hasRouteAttributes = controller.Selectors.Any(selector =>
                                                selector.AttributeRouteModel != null);
        if (hasRouteAttributes)
        {
            // This controller manually defined some routes, so treat this
            // as an override and not apply the convention here.
            return;
        }

        // Use the namespace and controller name to infer a route for the controller.
        //
        // Example:
        //
        //  controller.ControllerTypeInfo ->    "My.Application.Admin.UsersController"
        //  baseNamespace ->                    "My.Application"
        //
        //  template =>                         "Admin/[controller]"
        //
        // This makes your routes roughly line up with the folder structure of your project.
        //
        var namespc = controller.ControllerType.Namespace;
        if (namespc == null)
            return;
        var template = new StringBuilder();
        template.Append(namespc, _baseNamespace.Length + 1,
                        namespc.Length - _baseNamespace.Length - 1);
        template.Replace('.', '/');
        template.Append("/[controller]");

        foreach (var selector in controller.Selectors)
        {
            selector.AttributeRouteModel = new AttributeRouteModel()
            {
                Template = template.ToString()
            };
        }
    }
}
```

混合路由:属性路由与传统路由

MVC 应用程序可以混合使用传统路由与属性路由。通常将传统路由用于为浏览器处理 HTML 页面的控制器，将属性路由用于处理 REST API 的控制器。

操作既支持传统路由，也支持属性路由。通过在控制器或操作上放置路由可实现属性路由。不能通过传统路由访问定义属性路由的操作，反之亦然。控制器上的**任何**路由属性都会使控制器中的所有操作使用属性路由。

> **NOTE**
> 这两种路由系统的区别在于 URL 与路由模板匹配后所应用的过程。在传统路由中，将使用匹配项中的路由值，从包含所有传统路由操作的查找表中选择操作和控制器。在属性路由中，每个模板都与某项操作关联，无需进行进一步的查找。

复杂段

复杂段(例如，`[Route("/dog{token}cat")]`)通过非贪婪的方式从右到左匹配文字进行处理。有关说明，请参阅源代码。有关详细信息，请参阅此问题。

URL 生成

MVC 应用程序可以使用路由的 URL 生成功能，生成指向操作的 URL 链接。生成 URL 可消除硬编码 URL，使代码更稳定、更易维护。此部分重点介绍 MVC 提供的 URL 生成功能，并且仅涵盖 URL 生成工作原理的基础知识。有关 URL 生成的详细说明，请参阅路由。

`IUrlHelper` 接口用于生成 URL，是 MVC 与路由之间的基础结构的基础部分。在控制器、视图和视图组件中，可通过 `Url` 属性找到 `IUrlHelper` 的实例。

在此示例中，将通过 `Controller.Url` 属性使用 `IUrlHelper` 接口来生成指向另一项操作的 URL。

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    public IActionResult Source()
    {
        // Generates /UrlGeneration/Destination
        var url = Url.Action("Destination");
        return Content($"Go check out {url}, it's really great.");
    }

    public IActionResult Destination()
    {
        return View();
    }
}
```

如果应用程序使用的是传统默认路由，则 `url` 变量的值将为 URL 路径字符串 `/UrlGeneration/Destination`。此 URL 路径由路由创建，方法是将当前请求中的路由值(环境值)与传递到 `Url.Action` 的值合并，并将这些值替换到路由模板中：

```
ambient values: { controller = "UrlGeneration", action = "Source" }
values passed to Url.Action: { controller = "UrlGeneration", action = "Destination" }
route template: {controller}/{action}/{id?}

result: /UrlGeneration/Destination
```

路由模板中的每个路由参数都会通过将名称与这些值和环境值匹配，来替换掉原来的值。没有值的路由参数如果有默认值，则可使用默认值；如果本身是可选参数(比如此示例中的 `id`)，则可直接跳过。如果任何所需路由参数没有对应的值，URL 生成将失败。如果某个路由的 URL 生成失败，则尝试下一个路由，直到尝试所有路由或找到匹配项为止。

上面的 `Url.Action` 示例假定使用传统路由，但 URL 生成功能的工作方式与属性路由相似，只不过概念不同。在传统路由中，路由值用于扩展模板，`controller` 和 `action` 的路由值通常出现在该模板中 — 这种做法可行是因为通过路由匹配的 URL 遵守某项约定。在属性路由中，`controller` 和 `action` 的路由值不能出现在模板

中，它们用于查找要使用的模板。

此示例使用属性路由：

```
// In Startup class
public void Configure(IApplicationBuilder app)
{
    app.UseMvc();
}
```

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.Action("Destination"); // Generates /custom/url/to/destination
        return Content($"Go check out {url}, it's really great.");
    }

    [HttpGet("custom/url/to/destination")]
    public IActionResult Destination() {
        return View();
    }
}
```

MVC 生成一个包含所有属性路由操作的查找表，并匹配 `controller` 和 `action` 的值，以选择要用于生成 URL 的路由模板。在上述示例中，生成了 `custom/url/to/destination`。

根据操作名称生成 URL

`Url.Action`（`IUrlHelper`.`Action`）以及所有相关重载都基于这样一种想法：用户想通过指定控制器名称和操作名称来指定要链接的内容。

> **NOTE**
> 使用 `Url.Action` 时，将为用户指定 `controller` 和 `action` 的当前路由值，`controller` 和 `action` 的值是*环境值*和*值*的一部分。`Url.Action` 方法始终使用 `action` 和 `controller` 的当前值，并将生成将路由到当前操作的 URL 路径。

路由尝试使用环境值中的值来填充生成 URL 时未提供的信息。通过使用路由（比如 `{a}/{b}/{c}/{d}`）和环境值 `{ a = Alice, b = Bob, c = Carol, d = David }`，路由就具有足够的信息来生成 URL，而无需任何附加值，因为所有路由参数都有值。如果添加了值 `{ d = Donovan }`，则会忽略值 `{ d = David }`，生成的 URL 路径将为 `Alice/Bob/Carol/Donovan`。

> **WARNING**
> URL 路径是分层的。在上述示例中，如果添加了值 `{ c = Cheryl }`，则会忽略 `{ c = Carol, d = David }` 这两个值。在这种情况下，`d` 不再具有任何值，URL 生成将失败。用户需要指定 `c` 和 `d` 所需的值。使用默认路由（`{controller}/{action}/{id?}`）时可能会遇到此问题，但在实际操作中很少遇到此行为，因为 `Url.Action` 始终显式指定 `controller` 和 `action` 值。

较长的 `Url.Action` 重载还采用附加*路由值*对象，为 `controller` 和 `action` 以外的路由参数提供值。此重载最常与 `id` 结合使用，比如 `Url.Action("Buy", "Products", new { id = 17 })`。按照惯例，*路由值*对象通常是匿名类型的对象，但它也可以是 `IDictionary<>` 或*普通旧.NET 对象*。任何与路由参数不匹配的附加路由值都

放在查询字符串中。

```
using Microsoft.AspNetCore.Mvc;

public class TestController : Controller
{
    public IActionResult Index()
    {
        // Generates /Products/Buy/17?color=red
        var url = Url.Action("Buy", "Products", new { id = 17, color = "red" });
        return Content(url);
    }
}
```

TIP

若要创建绝对 URL，请使用采用 `protocol` 的重载：

`Url.Action("Buy", "Products", new { id = 17 }, protocol: Request.Scheme)`

根据路由生成 URL

上面的代码演示了如何通过传入控制器和操作名称来生成 URL。`IUrlHelper` 还提供 `Url.RouteUrl` 系列的方法。这些方法类似于 `Url.Action`，但它们不会将 `action` 和 `controller` 的当前值复制到路由值。最常见的用法是指定一个路由名称，以使用特定路由来生成 URL，通常不指定控制器或操作名称。

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.RouteUrl("Destination_Route"); // Generates /custom/url/to/destination
        return Content($"See {url}, it's really great.");
    }

    [HttpGet("custom/url/to/destination", Name = "Destination_Route")]
    public IActionResult Destination() {
        return View();
    }
}
```

在 HTML 中生成 URL

`IHtmlHelper` 提供 `HtmlHelper` 方法 `Html.BeginForm` 和 `Html.ActionLink`，可分别生成 `<form>` 和 `<a>` 元素。这些方法使用 `Url.Action` 方法来生成 URL，并且采用相似的参数。`HtmlHelper` 的配套 `Url.RouteUrl` 为 `Html.BeginRouteForm` 和 `Html.RouteLink`，两者具有相似的功能。

TagHelper 通过 `form` TagHelper 和 `<a>` TagHelper 生成 URL。两者均通过 `IUrlHelper` 来实现。有关详细信息，请参阅使用表单。

在视图内，可通过 `Url` 属性将 `IUrlHelper` 用于前文未涵盖的任何临时 URL 生成。

在操作结果中生成 URL

以上示例展示了如何在控制器中使用 `IUrlHelper`，不过，控制器中最常见的用法是将 URL 生成为操作结果的一部分。

`ControllerBase` 和 `Controller` 基类为操作结果提供简便的方法来引用另一项操作。一种典型用法是在接受用户输入后进行重定向。

```
public IActionResult Edit(int id, Customer customer)
{
    if (ModelState.IsValid)
    {
        // Update DB with new details.
        return RedirectToAction("Index");
    }
    return View(customer);
}
```

操作结果工厂方法遵循与 `IUrlHelper` 上的方法类似的模式。

专用传统路由的特殊情况

传统路由可以使用一种特殊的路由定义，称为*专用传统路由*。在下面的示例中，名为 `blog` 的路由是一种专用传统路由。

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

使用这些路由定义，`Url.Action("Index", "Home")` 将通过 `default` 路由生成 URL 路径 `/`，但是，为什么会这样？用户可能认为使用 `blog`，路由值 `{ controller = Home, action = Index }` 就足以生成 URL，且结果为 `/blog?action=Index&controller=Home`。

专用传统路由依赖于不具有相应路由参数的默认值的特殊行为，以防止路由在 URL 生成过程中"太贪婪"。在此例中，默认值是为 `{ controller = Blog, action = Article }`，`controller` 和 `action` 均未显示为路由参数。当路由执行 URL 生成时，提供的值必须与默认值匹配。使用 `blog` 的 URL 生成将失败，因为值 `{ controller = Home, action = Index }` 与 `{ controller = Blog, action = Article }` 不匹配。然后，路由回退，尝试使用 `default`，并最终成功。

区域

区域是一种 MVC 功能，用于将相关功能整理到一个组中，作为单独的路由命名空间（用于控制器操作）和文件夹结构（用于视图）。通过使用区域，应用程序可以有多个名称相同的控制器，只要它们具有不同的*区域*。通过向 `controller` 和 `action` 添加另一个路由参数 `area`，可使用区域为路由创建层次结构。此部分将讨论路由如何与区域交互；有关如何将区域与视图结合使用的详细信息，请参阅区域。

下面的示例将 MVC 配置为使用默认传统路由和*区域路由*（用于名为 `Blog` 的区域）：

```
app.UseMvc(routes =>
{
    routes.MapAreaRoute("blog_route", "Blog",
        "Manage/{controller}/{action}/{id?}");
    routes.MapRoute("default_route", "{controller}/{action}/{id?}");
});
```

与 URL 路径（比如 `/Manage/Users/AddUser`）匹配时，第一个路由将生成路由值 `{ area = Blog, controller = Users, action = AddUser }`。`area` 路由值由 `area` 的默认值生成，事实上，通过 `MapAreaRoute` 创建的路由等效于以下路由：

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog_route", "Manage/{controller}/{action}/{id?}",
        defaults: new { area = "Blog" }, constraints: new { area = "Blog" });
    routes.MapRoute("default_route", "{controller}/{action}/{id?}");
});
```

`MapAreaRoute` 通过为使用所提供的区域名称(本例中为 `Blog`)的 `area` 提供默认值和约束，来创建路由。默认值确保路由始终生成 `{ area = Blog, ... }`，约束要求在生成 URL 时使用值 `{ area = Blog, ... }`。

> **TIP**
> 传统路由依赖于顺序。一般情况下，具有区域的路由应放在路由表中靠前的位置，因为它们比没有区域的路由更特定。

在上面的示例中，路由值将与以下操作匹配：

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}
```

`AreaAttribute` 用于将控制器表示为某个区域的一部分，比方说，此控制器位于 `Blog` 区域中。没有 `[Area]` 属性的控制器不是任何区域的成员，在路由提供 `area` 路由值时**不**匹配。在下面的示例中，只有所列出的第一个控制器才能与路由值 `{ area = Blog, controller = Users, action = AddUser }` 匹配。

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace2
{
    // Matches { area = Zebra, controller = Users, action = AddUser }
    [Area("Zebra")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace3
{
    // Matches { area = string.Empty, controller = Users, action = AddUser }
    // Matches { area = null, controller = Users, action = AddUser }
    // Matches { controller = Users, action = AddUser }
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();

        }
    }
}
```

NOTE

出于完整性考虑，此处显示了每个控制器的命名空间，否则，控制器会发生命名冲突并生成编译器错误。类命名空间对 MVC 的路由没有影响。

前两个控制器是区域成员，仅在 `area` 路由值提供其各自的区域名称时匹配。第三个控制器不是任何区域的成员，只能在路由没有为 `area` 提供任何值时匹配。

NOTE

就*不匹配任何值*而言，缺少 `area` 值相当于 `area` 的值为 NULL 或空字符串。

在某个区域内执行某项操作时，`area` 的路由值将以*环境值*的形式提供，以便路由用于生成 URL。这意味着默认情况下，区域在 URL 生成中具有*粘性*，如以下示例所示。

```
app.UseMvc(routes =>
{
    routes.MapAreaRoute("duck_route", "Duck",
        "Manage/{controller}/{action}/{id?}");
    routes.MapRoute("default", "Manage/{controller=Home}/{action=Index}/{id?}");
});
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace4
{
    [Area("Duck")]
    public class UsersController : Controller
    {
        public IActionResult GenerateURLInArea()
        {
            // Uses the 'ambient' value of area
            var url = Url.Action("Index", "Home");
            // returns /Manage
            return Content(url);
        }

        public IActionResult GenerateURLOutsideOfArea()
        {
            // Uses the empty value for area
            var url = Url.Action("Index", "Home", new { area = "" });
            // returns /Manage/Home/Index
            return Content(url);
        }
    }
}
```

了解 IActionConstraint

> **NOTE**
>
> 此部分深入介绍框架内部结构以及 MVC 如何选择要执行的操作。典型的应用程序不需要自定义 `IActionConstraint`

即使不熟悉 `IActionConstraint`，也可能已经用过该接口。`[HttpGet]` 属性和类似的 `[Http-VERB]` 属性可实现 `IActionConstraint` 来限制操作方法的执行。

```
public class ProductsController : Controller
{
    [HttpGet]
    public IActionResult Edit() { }

    public IActionResult Edit(...) { }
}
```

假定使用默认传统路由，URL 路径 `/Products/Edit` 将生成值 `{ controller = Products, action = Edit }`，这将匹配此处所示的两项操作。在 `IActionConstraint` 术语中，我们会说，这两项操作都被视为候选项，因为它们都与该路由数据匹配。

当 `HttpGetAttribute` 执行时，它认为 *Edit()* 是 *GET* 的匹配项，而不是任何其他 Http 谓词的匹配项。`Edit(...)` 操作未定义任何约束，因此将匹配任何 Http 谓词。因此，假定 Http 谓词为 `POST`，则仅 `Edit(...)` 匹配。不过，对于 `GET`，这两项操作仍然都能匹配，只是具有 `IActionConstraint` 的操作始终被认为比没有该接口的操作*更匹配*。因此，由于 `Edit()` 具有 `[HttpGet]`，则认为它更特定，在两项操作都能匹配的情况将选择它。

从概念上讲，`IActionConstraint` 是一种*重载*形式，但它并不重载具有相同名称的方法，而在匹配相同 URL 的操作之间重载。属性路由也使用 `IActionConstraint`，这可能会导致将不同控制器中的操作都视为候选项。

实现 IActionConstraint

实现 `IActionConstraint` 最简单的方法是创建派生自 `System.Attribute` 的类，并将其置于操作和控制器上。

MVC 将自动发现任何应用为属性的 `IActionConstraint`。可使用应用程序模型应用约束，这可能是最灵活的一种方法，因为它允许对其应用方式进行元编程。

在下面的示例中，约束基于路由数据中的*国家/地区代码*选择操作。GitHub 上的完整示例。

```
public class CountrySpecificAttribute : Attribute, IActionConstraint
{
    private readonly string _countryCode;

    public CountrySpecificAttribute(string countryCode)
    {
        _countryCode = countryCode;
    }

    public int Order
    {
        get
        {
            return 0;
        }
    }

    public bool Accept(ActionConstraintContext context)
    {
        return string.Equals(
            context.RouteContext.RouteData.Values["country"].ToString(),
            _countryCode,
            StringComparison.OrdinalIgnoreCase);
    }
}
```

用户负责实现 `Accept` 方法，并为要执行的约束选择"顺序"。在此例中，当 `country` 路由值匹配时，`Accept` 方法返回 `true` 以表示该操作是匹配项。它与 `RouteValueAttribute` 的不同之处在于，它允许回退到非属性化操作。通过该示例可以了解到，如果定义 `en-US` 操作，则像 `fr-FR` 这样的国家/地区代码将回退到一个未应用 `[CountrySpecific(...)]` 的较通用的控制器。

`Order` 属性决定约束所属的*阶段*。操作约束基于 `Order` 分组运行。例如，该框架提供的所有 HTTP 方法属性均使用相同的 `Order` 值，以便在相同的阶段运行。用户可以按需设置阶段数来实现所需的策略。

> **TIP**
>
> 若要确定 `Order` 的值，请考虑是否应在 HTTP 方法前应用约束。数值较低的先运行。

ASP.NET Core 中的文件上传

2018/10/30 • • Edit Online

作者：Steve Smith

ASP.NET MVC 操作支持使用简单的模型绑定（针对较小文件）或流式处理（针对较大文件）上传一个或多个文件。

查看或下载 GitHub 中的示例

使用模型绑定上传小文件

要上传小文件，可使用多部分 HTML 窗体或使用 JavaScript 构造 POST 请求。下方显示使用 Razor（支持上传多个文件）的示例窗体：

```
<form method="post" enctype="multipart/form-data" asp-controller="UploadFiles" asp-action="Index">
    <div class="form-group">
        <div class="col-md-10">
            <p>Upload one or more files using this form:</p>
            <input type="file" name="files" multiple />
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-10">
            <input type="submit" value="Upload" />
        </div>
    </div>
</form>
```

为支持文件上传，HTML 窗体必须指定 `multipart/form-data` 的 `enctype`。上面显示的 `files` 输入元素支持上传多个文件。忽略此输入元素上的 `multiple` 属性，只允许上传单个文件。上述标记在浏览器中呈现为：

上传到服务器的单个文件可使用 IFormFile 接口通过模型绑定进行访问。`IFormFile` 具有以下结构：

```
public interface IFormFile
{
    string ContentType { get; }
    string ContentDisposition { get; }
    IHeaderDictionary Headers { get; }
    long Length { get; }
    string Name { get; }
    string FileName { get; }
    Stream OpenReadStream();
    void CopyTo(Stream target);
    Task CopyToAsync(Stream target, CancellationToken cancellationToken = null);
}
```

WARNING

切勿依赖或信任未经验证的 `FileName` 属性。`FileName` 属性应仅用于显示目的。

使用模型绑定和 `IFormFile` 接口上传文件时，操作方法可接受单个 `IFormFile` 或代表多个文件的 `IEnumerable<IFormFile>`（或 `List<IFormFile>`）。以下示例循环访问一个或多个上传的文件、将其保存到本地文件系统，并返回上传的文件总数和大小。

警告：以下代码使用 `GetTempFileName`。如果在未删除先前临时文件的情况下创建了 65535 个以上的文件，则该代码将抛出一个 `IOException`。实际应用须删除临时文件或使用 `GetTempPath` 和 `GetRandomFileName` 创建临时文件名称。每个服务器限制为 65535 个文件，因此服务器上的其他应用可能已占用全部 65535 个文件。

```
[HttpPost("UploadFiles")]
public async Task<IActionResult> Post(List<IFormFile> files)
{
    long size = files.Sum(f => f.Length);

    // full path to file in temp location
    var filePath = Path.GetTempFileName();

    foreach (var formFile in files)
    {
        if (formFile.Length > 0)
        {
            using (var stream = new FileStream(filePath, FileMode.Create))
            {
                await formFile.CopyToAsync(stream);
            }
        }
    }

    // process uploaded files
    // Don't rely on or trust the FileName property without validation.

    return Ok(new { count = files.Count, size, filePath});
}
```

使用 `IFormFile` 技术上传的文件在处理之前会缓存在内存中或 Web 服务器的磁盘中。在操作方法中，`IFormFile` 内容可作为流访问。除了本地文件系统之外，还可将文件流式传输到 Azure Blob 存储或实体框架。

要使用实体框架将二进制文件数据存储在数据库中，请在实体上定义类型为 `byte[]` 属性：

```
public class ApplicationUser : IdentityUser
{
    public byte[] AvatarImage { get; set; }
}
```

指定类型为 `IFormFile` 的 viewmodel 属性：

```
public class RegisterViewModel
{
    // other properties omitted

    public IFormFile AvatarImage { get; set; }
}
```

> **NOTE**
>
> `IFormFile` 可直接用作操作方法参数或 viewmodel 属性，如上所示。

将 `IFormFile` 复制到流并将其保存到字节数组中：

```
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    ViewData["ReturnUrl"] = returnUrl;
    if  (ModelState.IsValid)
    {
        var user = new ApplicationUser {
          UserName = model.Email,
          Email = model.Email
        };
        using (var memoryStream = new MemoryStream())
        {
            await model.AvatarImage.CopyToAsync(memoryStream);
            user.AvatarImage = memoryStream.ToArray();
        }
    // additional logic omitted

    // Don't rely on or trust the model.AvatarImage.FileName property
    // without validation.
}
```

> **NOTE**
>
> 在关系数据库中存储二进制数据时要格外小心，因为它可能对性能产生不利影响。

使用流式处理上传大文件

如果文件上传的大小或频率会导致应用出现资源问题，请考虑使用流式处理上传文件，而不是像如上所示的模型绑定方法那样全部缓冲文件。尽管使用 `IFormFile` 和模型绑定是更为简单的一种解决方案，但流式处理需要大量步骤才能正确实现。

> **NOTE**
>
> 任何超过 64KB 的单个缓冲文件会从 RAM 移动到服务器磁盘上的临时文件中。文件上传所用的资源（磁盘、RAM）取决于并发文件上传的数量和大小。流式处理与性能没有太大的关系，而是与规模有关。如果尝试缓冲过多上传，站点就会在内存或磁盘空间不足时崩溃。

以下示例演示如何通过 JavaScript/Angular 来流式传输到控制器操作。使用自定义筛选器属性生成文件的防伪令

牌，并在 HTTP 头中（而不是在请求正文中）传递该令牌。由于操作方法直接处理上传的数据，所以其他筛选器会禁用模型绑定。在该操作中，使用 `MultipartReader` 读取窗体的内容，它会读取每个单独的 `MultipartSection`，从而根据需要处理文件或存储内容。读取所有节之后，该操作会执行自己的模型绑定。

初始操作加载窗体并将防伪令牌保存在 Cookie 中（通过 `GenerateAntiforgeryTokenCookieForAjax` 属性）：

```
[HttpGet]
[GenerateAntiforgeryTokenCookieForAjax]
public IActionResult Index()
{
    return View();
}
```

该属性使用 ASP.NET Core 的内置防伪支持来设置包含请求令牌的 Cookie：

```
public class GenerateAntiforgeryTokenCookieForAjaxAttribute : ActionFilterAttribute
{
    public override void OnActionExecuted(ActionExecutedContext context)
    {
        var antiforgery = context.HttpContext.RequestServices.GetService<IAntiforgery>();

        // We can send the request token as a JavaScript-readable cookie,
        // and Angular will use it by default.
        var tokens = antiforgery.GetAndStoreTokens(context.HttpContext);
        context.HttpContext.Response.Cookies.Append(
            "XSRF-TOKEN",
            tokens.RequestToken,
            new CookieOptions() { HttpOnly = false });
    }
}
```

Angular 会在名为 `X-XSRF-TOKEN` 的请求标头中自动传递防伪令牌。ASP.NET Core MVC 应用配置为在 Startup.cs 的配置中引用此标头：

```
public void ConfigureServices(IServiceCollection services)
{
    // Angular's default header name for sending the XSRF token.
    services.AddAntiforgery(options => options.HeaderName = "X-XSRF-TOKEN");

    services.AddMvc();
}
```

如下所示的 `DisableFormValueModelBinding` 属性用于禁用针对 `Upload` 操作方法的模型绑定。

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class DisableFormValueModelBindingAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        var factories = context.ValueProviderFactories;
        factories.RemoveType<FormValueProviderFactory>();
        factories.RemoveType<JQueryFormValueProviderFactory>();
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

由于已禁用模型绑定，因此 `Upload` 操作方法不接受参数。它直接使用 `ControllerBase` 的 `Request` 属性。

`MultipartReader` 用于读取每个节。该文件以 GUID 文件名保存，并且键/值数据存储在 `KeyValueAccumulator` 中。读取所有节之后，系统会使用 `KeyValueAccumulator` 的内容将窗体数据绑定到模型类型。

完整的 `Upload` 方法如下所示：

警告：以下代码使用 `GetTempFileName`。如果在未删除先前临时文件的情况下创建了 65535 个以上的文件，则该代码将抛出一个 `IOException`。实际应用须删除临时文件或使用 `GetTempPath` 和 `GetRandomFileName` 创建临时文件名称。每个服务器限制为 65535 个文件，因此服务器上的其他应用可能已占用全部 65535 个文件。

```
// 1. Disable the form value model binding here to take control of handling
//    potentially large files.
// 2. Typically antiforgery tokens are sent in request body, but since we
//    do not want to read the request body early, the tokens are made to be
//    sent via headers. The antiforgery token filter first looks for tokens
//    in the request header and then falls back to reading the body.
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Upload()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        return BadRequest($"Expected a multipart request, but got {Request.ContentType}");
    }

    // Used to accumulate all the form url encoded key value pairs in the
    // request.
    var formAccumulator = new KeyValueAccumulator();
    string targetFilePath = null;

    var boundary = MultipartRequestHelper.GetBoundary(
        MediaTypeHeaderValue.Parse(Request.ContentType),
        _defaultFormOptions.MultipartBoundaryLengthLimit);
    var reader = new MultipartReader(boundary, HttpContext.Request.Body);

    var section = await reader.ReadNextSectionAsync();
    while (section != null)
    {
        ContentDispositionHeaderValue contentDisposition;
        var hasContentDispositionHeader = ContentDispositionHeaderValue.TryParse(section.ContentDisposition,
out contentDisposition);

        if (hasContentDispositionHeader)
        {
            if (MultipartRequestHelper.HasFileContentDisposition(contentDisposition))
            {
                targetFilePath = Path.GetTempFileName();
                using (var targetStream = System.IO.File.Create(targetFilePath))
                {
                    await section.Body.CopyToAsync(targetStream);

                    _logger.LogInformation($"Copied the uploaded file '{targetFilePath}'");
                }
            }
            else if (MultipartRequestHelper.HasFormDataContentDisposition(contentDisposition))
            {
                // Content-Disposition: form-data; name="key"
                //
                // value

                // Do not limit the key name length here because the
                // multipart headers length limit is already in effect.
                var key = HeaderUtilities.RemoveQuotes(contentDisposition.Name);
                var encoding = GetEncoding(section);
                using (var streamReader = new StreamReader(
                    section.Body,
                    encoding
```

```
                    encoding,
                    detectEncodingFromByteOrderMarks: true,
                    bufferSize: 1024,
                    leaveOpen: true))
                {
                    // The value length limit is enforced by MultipartBodyLengthLimit
                    var value = await streamReader.ReadToEndAsync();
                    if (String.Equals(value, "undefined", StringComparison.OrdinalIgnoreCase))
                    {
                        value = String.Empty;
                    }
                    formAccumulator.Append(key, value);

                    if (formAccumulator.ValueCount > _defaultFormOptions.ValueCountLimit)
                    {
                        throw new InvalidDataException($"Form key count limit
{_defaultFormOptions.ValueCountLimit} exceeded.");
                    }
                }
            }
        }

        // Drains any remaining section body that has not been consumed and
        // reads the headers for the next section.
        section = await reader.ReadNextSectionAsync();
    }

    // Bind form data to a model
    var user = new User();
    var formValueProvider = new FormValueProvider(
        BindingSource.Form,
        new FormCollection(formAccumulator.GetResults()),
        CultureInfo.CurrentCulture);

    var bindingSuccessful = await TryUpdateModelAsync(user, prefix: "",
        valueProvider: formValueProvider);
    if (!bindingSuccessful)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
    }

    var uploadedData = new UploadedData()
    {
        Name = user.Name,
        Age = user.Age,
        Zipcode = user.Zipcode,
        FilePath = targetFilePath
    };
    return Json(uploadedData);
}
```

疑难解答

以下是上传文件时遇到的一些常见问题及其可能的解决方案。

IIS 发生意外错误“找不到”

以下错误表示文件上传超过服务器配置的 `maxAllowedContentLength`：

```
HTTP 404.13 - Not Found
The request filtering module is configured to deny a request that exceeds the request content length.
```

默认设置为 `30000000`（大约 28.6MB）。可通过编辑 web.config 自定义该值：

```
<system.webServer>
  <security>
    <requestFiltering>
      <!-- This will handle requests up to 50MB -->
      <requestLimits maxAllowedContentLength="52428800" />
    </requestFiltering>
  </security>
</system.webServer>
```

此设置仅适用于 IIS。在 Kestrel 上托管时，默认情况下不会出现此行为。有关详细信息，请参阅 Request Limits <requestLimits>（请求限制）。

IFormFile 的空引用异常

如果控制器正在接受使用 `IFormFile` 上传的文件，但你发现该值始终为 NULL，请确认 HTML 窗体指定的 `enctype` 值是否为 `multipart/form-data`。如果未在 `<form>` 元素上设置此属性，则不会发生文件上传，并且任何绑定的 `IFormFile` 参数都将为 NULL。

在 ASP.NET Core 中将依赖项注入到控制器

2018/10/30 • • Edit Online

作者：Steve Smith

ASP.NET Core MVC 控制器应该通过构造函数显式请求其依赖关系。在某些情况下，单独的控制器操作可能需要服务，但在控制器级别上请求可能没有意义。在此情况下，也可在操作方法上选择将服务作为参数注入。

查看或下载示例代码（如何下载）

依赖关系注入

依赖关系注入是遵循 Dependency Inversion Principle（依赖关系反向原则）的方法，允许应用程序由松散耦合的模块组成。ASP.NET Core 具有对依赖关系注入的内置支持，能更易于测试和维护应用程序。

构造函数注入

ASP.NET Core 对基于构造函数的依赖关系注入的内置支持扩展到 MVC 控制器。通过将服务类型作为构造函数参数添加到控制器中，ASP.NET Core 将尝试使用其内置的服务容器来解析该类型。通常（但不总是）使用接口来定义服务。例如，如果应用程序具有依赖于当前时间的业务逻辑，则可以注入一个检索时间（而不是对其进行硬编码）的服务，这将允许测试进入使用设置时间的实现。

```
using System;

namespace ControllerDI.Interfaces
{
    public interface IDateTime
    {
        DateTime Now { get; }
    }
}
```

实现这样一个接口，以便在运行时使用系统时钟，此操作很简单：

```
using System;
using ControllerDI.Interfaces;

namespace ControllerDI.Services
{
    public class SystemDateTime : IDateTime
    {
        public DateTime Now
        {
            get { return DateTime.Now; }
        }
    }
}
```

准备就绪后，我们可在控制器中使用服务。在此情况下，我们在 `HomeController` `Index` 方法中添加了一些逻辑，根据每天的时间向用户显示问候语。

```
using ControllerDI.Interfaces;
using Microsoft.AspNetCore.Mvc;

namespace ControllerDI.Controllers
{
    public class HomeController : Controller
    {
        private readonly IDateTime _dateTime;

        public HomeController(IDateTime dateTime)
        {
            _dateTime = dateTime;
        }

        public IActionResult Index()
        {
            var serverTime = _dateTime.Now;
            if (serverTime.Hour < 12)
            {
                ViewData["Message"] = "It's morning here - Good Morning!";
            }
            else if (serverTime.Hour < 17)
            {
                ViewData["Message"] = "It's afternoon here - Good Afternoon!";
            }
            else
            {
                ViewData["Message"] = "It's evening here - Good Evening!";
            }
            return View();
        }
    }
}
```

如果现在运行应用程序，很可能会遇到错误：

```
An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'ControllerDI.Interfaces.IDateTime' while
attempting to activate 'ControllerDI.Controllers.HomeController'.
Microsoft.Extensions.DependencyInjection.ActivatorUtilities.GetService(IServiceProvider sp, Type type, Type
requiredBy, Boolean isDefaultParameterRequired)
```

当我们尚未在 `Startup` 类的 `ConfigureServices` 方法中配置服务时，会发生此错误。若要指定应使用 `SystemDateTime` 的实例解析 `IDateTime` 的请求，请将下面列表中突出显示的行添加到 `ConfigureServices` 方法中：

```
public void ConfigureServices(IServiceCollection services)
{
    // Add application services.
    services.AddTransient<IDateTime, SystemDateTime>();
}
```

> **NOTE**
>
> 可以使用几种不同的生存期选项（`Transient``Scoped` 或 `Singleton`）中的任意一个来实现此特定服务。请参阅依赖关系注入，了解每个作用域选项将如何影响服务的行为。

一旦服务配置完毕，运行应用程序并导航到主页应按预期显示基于时间的消息：

> **TIP**
> 请参阅测试控制器逻辑，了解如何显式请求控制器中的依赖项 http://deviq.com/explicit-dependencies-principle/ 可以更轻松地测试代码。

ASP.NET Core 的内置依赖关系注入支持请求服务的类只拥有一个构造函数。如果有多个构造函数，可能会收到如下异常消息：

```
An unhandled exception occurred while processing the request.

InvalidOperationException: Multiple constructors accepting all given argument types have been found in type
'ControllerDI.Controllers.HomeController'. There should only be one applicable constructor.
Microsoft.Extensions.DependencyInjection.ActivatorUtilities.FindApplicableConstructor(Type instanceType,
Type[] argumentTypes, ConstructorInfo& matchingConstructor, Nullable`1[]& parameterMap)
```

如错误消息所述，使用一个构造函数可更正此问题。还可将默认依赖关系注入容器替换为第三方实现，其中许多可支持多个构造函数。

FromServices 的操作注入

有时，控制器中不需要多个操作的服务。在此情况下，将服务作为参数注入操作方法可能是有意义的。通过标记具有属性 `[FromServices]` 的参数来完成此操作，如下所示：

```
public IActionResult About([FromServices] IDateTime dateTime)
{
    ViewData["Message"] = "Currently on the server the time is " + dateTime.Now;

    return View();
}
```

从控制器访问设置

从控制器中访问应用程序或配置设置是一种常见模式。此访问应使用配置中所述的选项模式。通常不应使用依赖关系注入直接从控制器请求设置。最好请求 `IOptions<T>` 实例，其中 `T` 是所需的配置类。

若要使用选项模式，需要创建一个表示选项的类，例如：

```
namespace ControllerDI.Model
{
    public class SampleWebSettings
    {
        public string Title { get; set; }
        public int Updates { get; set; }
    }
}
```

然后，需要将应用程序配置为使用选项模型，并将配置类添加到 `ConfigureServices` 中的服务集合：

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("samplewebsettings.json");
    Configuration = builder.Build();
}

public IConfigurationRoot Configuration { get; set; }

// This method gets called by the runtime. Use this method to add services to the container.
// For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?
LinkID=398940
public void ConfigureServices(IServiceCollection services)
{
    // Required to use the Options<T> pattern
    services.AddOptions();

    // Add settings from configuration
    services.Configure<SampleWebSettings>(Configuration);

    // Uncomment to add settings from code
    //services.Configure<SampleWebSettings>(settings =>
    //{
    //    settings.Updates = 17;
    //});

    services.AddMvc();

    // Add application services.
    services.AddTransient<IDateTime, SystemDateTime>();
}
```

NOTE

在上面的列表中，我们要将应用程序配置为从 JSON 格式的文件中读取设置。也可以完全使用代码配置设置，如上面注释的代码所示。有关进一步的配置选项，请参阅配置。

一旦指定了强类型的配置对象（在本例中为 `SampleWebSettings`）并将其添加到服务集合中，就可通过请求 `IOptions<T>` 的实例，从任何 Controller 或 Action 方法请求它（在本例中为 `IOptions<SampleWebSettings>`）。以下代码显示了如何从控制器请求设置：

```
public class SettingsController : Controller
{
    private readonly SampleWebSettings _settings;

    public SettingsController(IOptions<SampleWebSettings> settingsOptions)
    {
        _settings = settingsOptions.Value;
    }

    public IActionResult Index()
    {
        ViewData["Title"] = _settings.Title;
        ViewData["Updates"] = _settings.Updates;
        return View();
    }
}
```

遵循选项模式，可将设置和配置相互分离，并确保控制器遵循 separation of concerns（问题分离），因为它不需要知道如何或在哪里找到设置信息。由于控制器类中没有 static cling（静态粘附）或设置类的直接实例化，因此控制

器更易于对测试控制器逻辑进行单元测试。

在 ASP.NET Core 中将依赖项注入到视图

2018/10/30 • • Edit Online

作者：Steve Smith

ASP.NET Core 支持将依赖关系注入到视图。这对于视图特定服务很有用，例如仅为填充视图元素所需的本地化或数据。应尽量在控制器和视图之间保持问题分离。视图显示的大部分数据应该从控制器传入。

查看或下载示例代码（如何下载）

简单示例

可使用 `@inject` 指令将服务注入到视图中。可将 `@inject` 看作向视图添加属性，并用 DI 填充该属性。

`@inject` 的语法：`@inject <type> <name>`

操作中的 `@inject` 示例：

```
@using System.Threading.Tasks
@using ViewInjectSample.Model
@using ViewInjectSample.Model.Services
@model IEnumerable<ToDoItem>
@inject StatisticsService StatsService
<!DOCTYPE html>
<html>
<head>
    <title>To Do Items</title>
</head>
<body>
    <div>
        <h1>To Do Items</h1>
        <ul>
            <li>Total Items: @StatsService.GetCount()</li>
            <li>Completed: @StatsService.GetCompletedCount()</li>
            <li>Avg. Priority: @StatsService.GetAveragePriority()</li>
        </ul>
        <table>
            <tr>
                <th>Name</th>
                <th>Priority</th>
                <th>Is Done?</th>
            </tr>
            @foreach (var item in Model)
            {
                <tr>
                    <td>@item.Name</td>
                    <td>@item.Priority</td>
                    <td>@item.IsDone</td>
                </tr>
            }
        </table>
    </div>
</body>
</html>
```

此视图显示 `ToDoItem` 实例的列表，以及显示总体统计信息的摘要。摘要从已注入的 `StatisticsService` 中填充。在 Startup.cs 的 `ConfigureServices` 中为依赖关系注入注册此服务：

```
// For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?
LinkID=398940
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddTransient<IToDoItemRepository, ToDoItemRepository>();
    services.AddTransient<StatisticsService>();
    services.AddTransient<ProfileOptionsService>();
```

`StatisticsService` 通过存储库访问 `ToDoItem` 实例集并执行某些计算：

```
using System.Linq;
using ViewInjectSample.Interfaces;

namespace ViewInjectSample.Model.Services
{
    public class StatisticsService
    {
        private readonly IToDoItemRepository _toDoItemRepository;

        public StatisticsService(IToDoItemRepository toDoItemRepository)
        {
            _toDoItemRepository = toDoItemRepository;
        }

        public int GetCount()
        {
            return _toDoItemRepository.List().Count();
        }

        public int GetCompletedCount()
        {
            return _toDoItemRepository.List().Count(x => x.IsDone);
        }

        public double GetAveragePriority()
        {
            if (_toDoItemRepository.List().Count() == 0)
            {
                return 0.0;
            }

            return _toDoItemRepository.List().Average(x => x.Priority);
        }
    }
}
```

示例存储库使用内存中集合。建议不将上示实现（对内存中的所有数据进行操作）用于远程访问的大型数据集。

该示例显示绑定到视图的模型数据以及注入到视图中的服务：

localhost:10653/todo

To Do Items

- Total Items: 50
- Completed: 17
- Avg. Priority: 3

Name	Priority	Is Done?
Task 1	1	True
Task 2	2	False
Task 3	3	False
Task 4	4	True
Task 5	5	False

填充查找数据

视图注入可用于填充 UI 元素(如下拉列表)中的选项。请考虑这样的用户个人资料窗体，其中包含用于指定性别、状态和其他首选项的选项。使用标准 MVC 方法呈现这样的窗体，需让控制器为每组选项请求数据访问服务，然后用要绑定的每组选项填充模型或 `ViewBag`。

另一种方法是将服务直接注入视图以获取选项。这最大限度地减少了控制器所需的代码量，将此视图元素构造逻辑移入视图本身。显示个人资料编辑窗体的控制器操作只需要传递个人资料实例的窗体：

```
using Microsoft.AspNetCore.Mvc;
using ViewInjectSample.Model;

namespace ViewInjectSample.Controllers
{
    public class ProfileController : Controller
    {
        [Route("Profile")]
        public IActionResult Index()
        {
            // TODO: look up profile based on logged-in user
            var profile = new Profile()
            {
                Name = "Steve",
                FavColor = "Blue",
                Gender = "Male",
                State = new State("Ohio","OH")
            };
            return View(profile);
        }
    }
}
```

用于更新这些首选项的 HTML 窗体包括三个属性的下拉列表：

localhost:10653/profile

Update Profile

Name: Steve
Gender: Male
State: Ohio
Fav. Color: Blue

这些列表由已注入视图的服务填充：

```
@using System.Threading.Tasks
@using ViewInjectSample.Model.Services
@model ViewInjectSample.Model.Profile
@inject ProfileOptionsService Options
<!DOCTYPE html>
<html>
<head>
    <title>Update Profile</title>
</head>
<body>
<div>
    <h1>Update Profile</h1>
    Name: @Html.TextBoxFor(m => m.Name)
    <br/>
    Gender: @Html.DropDownList("Gender",
           Options.ListGenders().Select(g =>
                new SelectListItem() { Text = g, Value = g }))
    <br/>

    State: @Html.DropDownListFor(m => m.State.Code,
           Options.ListStates().Select(s =>
                new SelectListItem() { Text = s.Name, Value = s.Code}))
    <br />

    Fav. Color: @Html.DropDownList("FavColor",
           Options.ListColors().Select(c =>
                new SelectListItem() { Text = c, Value = c }))
    </div>
</body>
</html>
```

`ProfileOptionsService` 是 UI 级别的服务，旨在准确提供此窗体所需的数据：

```
using System.Collections.Generic;

namespace ViewInjectSample.Model.Services
{
    public class ProfileOptionsService
    {
        public List<string> ListGenders()
        {
            // keeping this simple
            return new List<string>() {"Female", "Male"};
        }

        public List<State> ListStates()
        {
            // a few states from USA
            return new List<State>()
            {
                new State("Alabama", "AL"),
                new State("Alaska", "AK"),
                new State("Ohio", "OH")
            };
        }

        public List<string> ListColors()
        {
            return new List<string>() { "Blue","Green","Red","Yellow" };
        }
    }
}
```

> **IMPORTANT**
>
> 请记得在 `Startup.ConfigureServices` 中注册通过依赖项注入请求的类型。注销的类型将在运行时引发异常，因为服务提供程序通过 GetRequiredService 接受内部查询。

替代服务

除了注入新的服务之外，此方法也可用于替代以前在页面上注入的服务。下图显示了第一个示例中使用的页面上的所有可用字段：

如你所见，包括默认字段 `Html`、`Component` 和 `Url`（以及我们注入的 `StatsService`）。如果想用自己的 HTML 帮助程序替换默认的 HTML 帮助程序，可使用 `@inject` 轻松完成：

```
@using System.Threading.Tasks
@using ViewInjectSample.Helpers
@inject MyHtmlHelper Html
<!DOCTYPE html>
<html>
<head>
    <title>My Helper</title>
</head>
<body>
    <div>
        Test: @Html.Value
    </div>
</body>
</html>
```

如果想扩展现有的服务，用自己的方法继承或包装现有的实现时，只需使用此方法。

请参阅

- Simon Timms 的博客：在视图中查找数据

ASP.NET Core 中的测试控制器逻辑

2018/11/10 • • Edit Online

作者：Steve Smith

控制器在任何 ASP.NET Core MVC 应用中起着核心作用。因此，应该对控制器表现达到预期怀有信心。在将应用部署到生产环境之前，自动测试可以检测到错误。

查看或下载示例代码（如何下载）

控制器逻辑的单元测试

单元测试涉及通过基础结构和依赖项单独测试应用的一部分。单元测试控制器逻辑时，仅测试单个操作的内容，不测试其依赖项或框架自身的行为。

将控制器操作的单元测试设置为专注于控制器的行为。控制器单元测试将避开筛选器、路由或模型绑定等方案。涵盖共同响应请求的组件之间的交互的测试由*集成测试*处理。有关集成测试的详细信息，请参阅在 ASP.NET Core 中的集成测试。

如果编写自定义筛选器和路由，应对其单独进行单元测试，而不是在测试特定控制器操作时进行。

要演示控制器单元测试，请查看以下示例应用中的控制器。主控制器显示集体讨论会话列表并允许使用 POST 请求创建新的集体讨论会话：

```
public class HomeController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public HomeController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index()
    {
        var sessionList = await _sessionRepository.ListAsync();

        var model = sessionList.Select(session => new StormSessionViewModel()
        {
            Id = session.Id,
            DateCreated = session.DateCreated,
            Name = session.Name,
            IdeaCount = session.Ideas.Count
        });

        return View(model);
    }

    public class NewSessionModel
    {
        [Required]
        public string SessionName { get; set; }
    }

    [HttpPost]
    public async Task<IActionResult> Index(NewSessionModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        else
        {
            await _sessionRepository.AddAsync(new BrainstormSession()
            {
                DateCreated = DateTimeOffset.Now,
                Name = model.SessionName
            });
        }

        return RedirectToAction(actionName: nameof(Index));
    }
}
```

前面的控制器：

- 遵循显式依赖关系原则。
- 期望依赖关系注入 (DI) 提供 `IBrainstormSessionRepository` 的实例。
- 可以通过使用 mock 对象框架（如 Moq）的模拟 `IBrainstormSessionRepository` 服务进行测试。*模拟对象*是由一组预先确定的用于测试的属性和方法行为的对象。有关详细信息，请参阅集成测试简介。

`HTTP GET Index` 方法没有循环或分支，且仅调用一个方法。此操作的单元测试：

- 使用 `GetTestSessions` 方法模拟 `IBrainstormSessionRepository` 服务。`GetTestSessions` 使用日期和会话名称创建两个 mock 集体讨论会话。
- 执行 `Index` 方法。
- 根据该方法返回的结果进行断言：

- 将返回 ViewResult。
- ViewDataDictionary.Model 是 `StormSessionViewModel`。
- 有两个集体讨论会话存储在 `ViewDataDictionary.Model` 中。

```
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
        viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

```
private List<BrainstormSession> GetTestSessions()
{
    var sessions = new List<BrainstormSession>();
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    });
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 1),
        Id = 2,
        Name = "Test Two"
    });
    return sessions;
}
```

主控制器的 `HTTP POST Index` 方法测试验证：

- 当 ModelState.IsValid 为 `false` 时，操作方法将返回有相应数据的 *400 错误请求* ViewResult。
- 当 `ModelState.IsValid` 为 `true` 时：
  - 将调用存储库上的 `Add` 方法。
  - 将返回有正确参数的 RedirectToActionResult。

通过使用 AddModelError 添加错误来测试无效模型状态，如下第一个测试所示：

```
[Fact]
public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}
```

当 ModelState 无效时，将返回与 GET 请求相同的 `ViewResult`。测试不会尝试传入无效模型。传入无效模型不是有效的方法，因为不会运行模型绑定（尽管集成测试使用模型绑定）。在本例中，不测试模型绑定。这些单元测试仅测试操作方法中的代码。

第二个测试验证 `ModelState` 有效时的情况：

- 已通过存储库添加新的 `BrainstormSession`。
- 该方法将返回带有所需属性的 `RedirectToActionResult`。

通常会忽略未调用的模拟调用，但在设置调用末尾调用 `Verifiable` 就可以在测试中进行 mock 验证。这通过对 `mockRepo.Verify` 的调用来完成，进行这种调用时，如果未调用所需方法，则测试将失败。

> **NOTE**
>
> 通过此示例中使用的 Moq 库，可以混合可验证（或称“严格”）mock 和非可验证 mock（也称为“宽松”mock 或存根）。详细了解使用 Moq 自定义 Mock 行为。

示例应用中的 SessionController 显示与特定集体讨论会话相关的信息。该控制器包含用于处理无效 `id` 值的逻辑（以下示例中有两个 `return` 方案可用来应对这些情况）。最后的 `return` 语句向视图

(Controllers/SessionController.cs) 返回一个新的 `StormSessionViewModel`：

```
public class SessionController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public SessionController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index(int? id)
    {
        if (!id.HasValue)
        {
            return RedirectToAction(actionName: nameof(Index),
                controllerName: "Home");
        }

        var session = await _sessionRepository.GetByIdAsync(id.Value);
        if (session == null)
        {
            return Content("Session not found.");
        }

        var viewModel = new StormSessionViewModel()
        {
            DateCreated = session.DateCreated,
            Name = session.Name,
            Id = session.Id
        };

        return View(viewModel);
    }
}
```

单元测试包括对会话控制器 `Index` 操作中的每个 `return` 方案执行一个测试：

```
[Fact]
public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
{
    // Arrange
    var controller = new SessionController(sessionRepository: null);

    // Act
    var result = await controller.Index(id: null);

    // Assert
    var redirectToActionResult =
        Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
}

[Fact]
public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var contentResult = Assert.IsType<ContentResult>(result);
    Assert.Equal("Session not found.", contentResult.Content);
}

[Fact]
public async Task IndexReturnsViewResultWithStormSessionViewModel()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSessions().FirstOrDefault(
            s => s.Id == testSessionId));
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsType<StormSessionViewModel>(
        viewResult.ViewData.Model);
    Assert.Equal("Test One", model.Name);
    Assert.Equal(2, model.DateCreated.Day);
    Assert.Equal(testSessionId, model.Id);
}
```

移动到想法控制器，应用会将功能公开为 `api/ideas` 路由上的 Web API：

- `ForSession` 方法将返回与集体讨论会话关联的想法列表（`IdeaDTO`）。
- `Create` 方法会向会话中添加新想法。

```
[HttpGet("forsession/{sessionId}")]
public async Task<IActionResult> ForSession(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);
    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return Ok(result);
}

[HttpPost("create")]
public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);
    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return Ok(session);
}
```

避免直接通过 API 调用返回企业域实体。域实体：

- 包含的数据通常比客户端所需的数据更多。
- 无需将应用的内部域模型与公开的 API 结合。

可以执行域实体与返回到客户端的类型之间的映射：

- 手动执行 LINQ `Select`，如同示例应用所用的那样。有关详细信息，请参阅 LINQ（语言集成查询）。
- 自动生成库，如 AutoMapper。

接着，示例应用会演示想法控制器的 `Create` 和 `ForSession` API 方法的单元测试。

示例应用包含两个 `ForSession` 测试。第一个测试可确定 `ForSession` 是否返回无效会话的 NotFoundObjectResult（找不到 HTTP）：

```
[Fact]
public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
    Assert.Equal(testSessionId, notFoundObjectResult.Value);
}
```

第二个 `ForSession` 测试可确定 `ForSession` 是否返回有效会话的会话想法列表（`<List<IdeaDTO>>`）。这些测试还会检查第一个想法，以确认其 `Name` 属性正确：

```
[Fact]
public async Task ForSession_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

若要测试 `Create` 方法在 `ModelState` 无效时的行为，示例应用会在测试中将模型错误添加到控制器。请勿在单元测试中尝试测试模型有效性或模型绑定—仅测试操作方法在遇到无效 `ModelState` 时的行为：

```
[Fact]
public async Task Create_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.Create(model: null);

    // Assert
    Assert.IsType<BadRequestObjectResult>(result);
}
```

`Create` 的第二个测试依赖存储库返回 `null`，所以 mock 存储库配置为返回 `null`。无需创建测试数据库（在内存中或其他位置）并构建将返回此结果的查询。该测试可以在单个语句中完成，如示例代码所示：

```
[Fact]
public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.Create(new NewIdeaModel());

    // Assert
    Assert.IsType<NotFoundObjectResult>(result);
}
```

第三个 `Create` 测试 `Create_ReturnsNewlyCreatedIdeaForSession` 验证调用了存储库的 `UpdateAsync` 方法。使用 `Verifiable` 调用 mock，然后调用模拟存储库的 `Verify` 方法，以确认执行了可验证的方法。确保 `UpdateAsync` 方法保存了数据不是单元测试的职责—这可以通过集成测试完成。

```
[Fact]
public async Task Create_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.Create(newIdea);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnSession.Ideas.Count());
    Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
}
```

测试 ActionResult<T>

在 ASP.NET Core 2.1 或更高版本中，ActionResult<T> (ActionResult<TValue>) 支持返回从 `ActionResult` 派生的类型或返回特定类型。

示例应用包含将返回给定会话 `id` 的 `List<IdeaDTO>` 的方法。如果会话 `id` 不存在，控制器将返回 NotFound：

```
[HttpGet("forsessionactionresult/{sessionId}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<ActionResult<List<IdeaDTO>>> ForSessionActionResult(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);

    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return result;
}
```

`ApiIdeasControllerTests` 中包含 `ForSessionActionResult` 控制器的两个测试。

第一个测试可确认控制器将返回 `ActionResult`，而不是不存在会话 `id` 的不存在想法列表：

- `ActionResult` 类型为 `ActionResult<List<IdeaDTO>>`。
- Result 为 NotFoundObjectResult。

```
[Fact]
public async Task ForSessionActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    var nonExistentSessionId = 999;

    // Act
    var result = await controller.ForSessionActionResult(nonExistentSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

对于有效会话 `id`，第二个测试可确认该方法将返回：

- 类型为 `List<IdeaDTO>` 的 `ActionResult`。
- ActionResult<T>.Value 的类型为 `List<IdeaDTO>`。
- 列表中的第一项是与 mock 会话中存储的想法匹配的有效想法（通过调用 `GetTestSession` 获取）。

```
[Fact]
public async Task ForSessionActionResult_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSessionActionResult(testSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(actionResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

示例应用还包含用于为给定会话创建新的 `Idea` 的方法。控制器将返回：

- BadRequest（对于无效模型）。
- NotFound（如果会话不存在）。
- CreatedAtAction（当使用新想法更新会话时）。

```
[HttpPost("createactionresult")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
public async Task<ActionResult<BrainstormSession>> CreateActionResult([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);

    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return CreatedAtAction(nameof(CreateActionResult), new { id = session.Id }, session);
}
```

`ApiIdeasControllerTests` 中包含 `CreateActionResult` 的三个测试。

第一个测试可确认将返回 BadRequest（对于无效模型）。

```
[Fact]
public async Task CreateActionResult_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.CreateActionResult(model: null);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<BadRequestObjectResult>(actionResult.Result);
}
```

第二个测试可确认将返回 NotFound（如果会话不存在）。

```
[Fact]
public async Task CreateActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var nonExistentSessionId = 999;
    string testName = "test name";
    string testDescription = "test description";
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = nonExistentSessionId
    };

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

对于有效会话 `id`，最后一个测试可确认：

- 该方法将返回类型为 `BrainstormSession` 的 `ActionResult`。
- ActionResult<T>.Result 为 CreatedAtActionResult。`CreatedAtActionResult` 类似于包含 `Location` 标头的 *201 Created* 响应。
- ActionResult<T>.Value 的类型为 `BrainstormSession`。
- 调用了用于更新会话 `UpdateAsync(testSession)` 的 mock 调用。通过执行断言中的 `mockRepo.Verify()` 来检查 `Verifiable` 方法调用。
- 将返回该会话的两个 `Idea` 对象。
- 最后一项（通过对 `UpdateAsync` 的 mock 调用而添加的 `Idea`）与添加到测试中的会话的 `newIdea` 匹配。

```
[Fact]
public async Task CreateActionResult_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    var createdAtActionResult = Assert.IsType<CreatedAtActionResult>(actionResult.Result);
    var returnValue = Assert.IsType<BrainstormSession>(createdAtActionResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnValue.Ideas.Count());
    Assert.Equal(testName, returnValue.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnValue.Ideas.LastOrDefault().Description);
}
```

其他资源

- 在 ASP.NET Core 中的集成测试
- 使用 Visual Studio 创建和运行单位测试。
- Explicit Dependencies Principle（显式依赖关系原则）

ASP.NET Core 中的会话和应用状态

2019/1/10 • • Edit Online

作者：Rick Anderson、Steve Smith、Diana LaRose 和 Luke Latham

HTTP 是无状态的协议。不采取其他步骤的情况下，HTTP 请求是不保留用户值或应用状态的独立消息。本文介绍了几种保留请求间用户数据和应用状态的方法。

查看或下载示例代码（如何下载）

状态管理

可以使用几种方法存储状态。本主题稍后将对每个方法进行介绍。

存储方法	存储机制
Cookie	HTTP Cookie（可能包括使用服务器端应用代码存储的数据）
会话状态	HTTP Cookie 和服务器端应用代码
TempData	HTTP Cookie 或会话状态
查询字符串	HTTP 查询字符串
隐藏字段	HTTP 窗体字段
HttpContext.Items	服务器端应用代码
缓存	服务器端应用代码
依赖关系注入	服务器端应用代码

Cookie

Cookie 存储所有请求的数据。因为 Cookie 是随每个请求发送的，所以它们的大小应该保持在最低限度。理想情况下，仅标识符应存储在 Cookie 中，而数据则由应用存储。大多数浏览器 Cookie 大小限制为 4096 个字节。每个域仅有有限数量的 Cookie 可用。

由于 Cookie 易被篡改，因此它们必须由服务器进行验证。客户端上的 Cookie 可能被用户删除或者过期。但是，Cookie 通常是客户端上最持久的数据暂留形式。

Cookie 通常用于个性化设置，其中的内容是为已知用户定制的。大多数情况下，仅标识用户，但不对其进行身份验证。Cookie 可以存储用户名、帐户名或唯一的用户 ID（例如 GUID）。然后，可以使用 Cookie 访问用户的个性化设置，例如首选的网站背景色。

发布 Cookie 和处理隐私问题时，请留意欧盟一般数据保护条例 (GDPR)。有关详细信息，请参阅 ASP.NET Core 中的一般数据保护条例 (GDPR) 支持。

会话状态

会话状态是在用户浏览 Web 应用时用来存储用户数据的 ASP.NET Core 方案。会话状态使用应用维护的存储来

保存客户端所有请求的数据。会话数据由缓存支持并被视为临时数据 - 站点应在没有会话数据的情况下继续运行。

> **NOTE**
> SignalR 应用不支持会话，因为 SignalR 中心可能独立于 HTTP 上下文执行。例如，当中心打开的长轮询请求超出请求的 HTTP 上下文的生存期时，可能发生这种情况。

ASP.NET Core 通过向客户端提供包含会话 ID 的 Cookie 来维护会话状态，该会话 ID 与每个请求一起发送到应用。应用使用会话 ID 来获取会话数据。

会话状态具有以下行为：

- 由于会话 Cookie 是特定于浏览器的，因此不能跨浏览器共享会话。
- 浏览器会话结束时删除会话 Cookie。
- 如果收到过期的会话 Cookie，则创建使用相同会话 Cookie 的新会话。
- 不会保留空会话 - 会话中必须设置了至少一个值以保存所有请求的会话。会话未保留时，为每个新的请求生成新会话 ID。
- 应用在上次请求后保留会话的时间有限。应用设置会话超时，或者使用 20 分钟的默认值。会话状态适用于存储特定于特定会话的用户数据，但该数据无需永久的会话存储。
- 调用 ISession.Clear 实现或者会话过期时，会删除会话数据。
- 没有默认机制告知客户端浏览器已关闭或者客户端上的会话 Cookie 被删除或过期的应用代码。ASP.NET Core MVC 和 Razor 页面模板包括对一般数据保护条例 (GDPR) 的支持。会话状态 cookie 不是必需的，如果禁用跟踪，则会话状态不起作用。

> **WARNING**
> 请勿将敏感数据存储在会话状态中。用户可能不会关闭浏览器并清除会话 Cookie。某些浏览器会保留所有浏览器窗口中的有效会话 Cookie。会话可能不限于单个用户 - 下一个用户可能继续使用同一会话 Cookie 浏览应用。

内存中缓存提供程序在应用驻留的服务器内存中存储会话数据。在服务器场方案中：

- 使用粘性会话将每个会话加入到单独服务器上的特定应用实例。默认情况下，Azure 应用服务使用应用程序请求路由 (ARR) 强制实施粘性会话。然而，粘性会话可能会影响可伸缩性，并使 Web 应用更新变得复杂。更好的方法是使用 Redis 或 SQL Server 分布式缓存，它们不需要粘性会话。有关更多信息，请参见分布式缓存在 ASP.NET Core 中。
- 通过 IDataProtector 加密会话 Cookie。必须正确配置数据保护，以在每台计算机上读取会话 Cookie。有关详细信息，请参阅 ASP.NET Core 数据保护 和密钥存储提供程序。

配置会话状态

Microsoft.AspNetCore.App metapackage 中包含的 Microsoft.AspNetCore.Session 包提供中间件来管理会话状态。若要启用会话中间件，`Startup` 必须包含：

Microsoft.AspNetCore.Session 包提供中间件来管理会话状态。若要启用会话中间件，`Startup` 必须包含：

- 任一 IDistributedCache 内存缓存。`IDistributedCache` 实现用作会话后备存储。有关更多信息，请参见分布式缓存在 ASP.NET Core 中。
- 对 `ConfigureServices` 中 AddSession 的调用。
- 对 `Configure` 中 UseSession 的调用。

以下代码演示如何使用 `IDistributedCache` 的默认内存中实现设置内存中会话提供程序：

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<CookiePolicyOptions>(options =>
        {
            options.CheckConsentNeeded = context => true;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });

        services.AddDistributedMemoryCache();

        services.AddSession(options =>
        {
            // Set a short timeout for easy testing.
            options.IdleTimeout = TimeSpan.FromSeconds(10);
            options.Cookie.HttpOnly = true;
        });

        services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();
        app.UseSession();
        app.UseHttpContextItemsMiddleware();
        app.UseMvc();
    }
}
```

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDistributedMemoryCache();

        services.AddSession(options =>
        {
            // Set a short timeout for easy testing.
            options.IdleTimeout = TimeSpan.FromSeconds(10);
            options.CookieHttpOnly = true;
        });

        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseSession();
        app.UseHttpContextItemsMiddleware();
        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

中间件的顺序很重要。在前面的示例中，在 `UseMvc` 之后调用 `UseSession` 时会发生 `InvalidOperationException` 异常。有关详细信息，请参阅中间件排序。

配置会话状态后，HttpContext.Session 可用。

调用 `UseSession` 以前无法访问 `HttpContext.Session`。

在应用已经开始写入到响应流之后，不能创建有新会话 Cookie 的新会话。此异常记录在 Web 服务器日志中但不显示在浏览器中。

以异步方式加载会话状态

只有在 TryGetValue、Set 或 Remove 方法之前显式调用 ISession.LoadAsync 方法，ASP.NET Core 中的默认会话提供程序才会从基础 IDistributedCache 后备存储以异步方式加载会话记录。如果未先调用 `LoadAsync`，则会同步加载基础会话记录，这可能对性能产生大规模影响。

若要让应用强制实施此模式，如果未在 `TryGetValue`、`Set` 或 `Remove` 之前调用 `LoadAsync` 方法，那么使用引起异常的版本包装 DistributedSessionStore 和 DistributedSession 实现。在服务容器中注册的已包装的版本。

会话选项

若要替代会话默认值，请使用 SessionOptions。

选项	说明
Cookie	确定用于创建 Cookie 的设置。名称默认为 SessionDefaults.CookieName（`.AspNetCore.Session`）。路径默认为 SessionDefaults.CookiePath（`/`）。SameSite 默认为 SameSiteMode.Lax（`1`）。HttpOnly 默认为 `true`。IsEssential 默认为 `false`。

选项	说明
IdleTimeout	`IdleTimeout` 显示放弃其内容前，内容可以空闲多长时间。每个会话访问都会重置超时。此设置仅适用于会话内容，不适用于 Cookie。默认为 20 分钟。
IOTimeout	允许从存储加载会话或者将其提交回存储的最大时长。此设置可能仅适用于异步操作。可以使用 InfiniteTimeSpan 禁用超时。默认值为 1 分钟。

会话使用 Cookie 跟踪和标识来自单个浏览器的请求。默认情况下，此 Cookie 名为 `.AspNetCore.Session` ，并使用路径 `/` 。由于 Cookie 默认值不指定域，因此它不提供页上的客户端脚本（因为 HttpOnly 默认为 `true` ）。

选项	说明
CookieDomain	确定用于创建 Cookie 的域。默认情况下不设置 `CookieDomain` 。
CookieHttpOnly	确定浏览器是否应该允许客户端 JavaScript 访问 Cookie。默认为 `true` ，这意味着 Cookie 仅传递到 HTTP 请求，且不适用于页面上的脚本。
CookieName	确定用来保留会话 ID 的 Cookie 名称。默认为 SessionDefaults.CookieName (`.AspNetCore.Session`)。
CookiePath	确定用于创建 Cookie 的路径。默认为 SessionDefaults.CookiePath (`/`)。
CookieSecure	确定 Cookie 是否应仅在 HTTPS 请求上传输。默认为 CookieSecurePolicy.None (`2`)。
IdleTimeout	`IdleTimeout` 显示放弃其内容前，内容可以空闲多长时间。每个会话访问都会重置超时。请注意，这仅适用于会话内容，不适用于 Cookie。默认为 20 分钟。

会话使用 Cookie 跟踪和标识来自单个浏览器的请求。默认情况下，此 Cookie 名为 `.AspNet.Session` ，并使用路径 `/` 。

若要替换 Cookie 会话默认值，请使用 `SessionOptions` :

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDistributedMemoryCache();

    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddSession(options =>
    {
        options.Cookie.Name = ".AdventureWorks.Session";
        options.IdleTimeout = TimeSpan.FromSeconds(10);
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDistributedMemoryCache();

    services.AddSession(options =>
    {
        options.CookieName = ".AdventureWorks.Session";
        options.IdleTimeout = TimeSpan.FromSeconds(10);
    });

    services.AddMvc();
}
```

应用使用 IdleTimeout 属性确定放弃服务器缓存中的内容前，内容可以空闲多长时间。此属性独立于 Cookie 到期时间。通过会话中间件传递的每个请求都会重置超时。

会话状态为"非锁定"。如果两个请求同时尝试修改同一会话的内容，则后一个请求替代前一个请求。`Session` 是作为一个连贯会话实现的，这意味着所有内容都存储在一起。两个请求试图修改不同的会话值时，后一个请求可能替代前一个做出的会话更改。

设置和获取会话值

使用 HttpContext.Session 从 Razor Pages PageModel 类或 MVC 控制器类访问会话状态。此属性是 ISession 实现。

`ISession` 实现提供设置和检索整数和字符串值的若干扩展方法。项目引用 Microsoft.AspNetCore.Http.Extensions 包时，扩展方法位于 Microsoft.AspNetCore.Http 命名空间中（添加 `using Microsoft.AspNetCore.Http;` 语句获取对扩展方法的访问权限）。这两个包均包括在 Microsoft.AspNetCore.App 元包中。

`ISession` 实现提供设置和检索整数和字符串值的若干扩展方法。项目引用 Microsoft.AspNetCore.Http.Extensions 包时，扩展方法位于 Microsoft.AspNetCore.Http 命名空间中（添加 `using Microsoft.AspNetCore.Http;` 语句获取对扩展方法的访问权限）。

`ISession` 扩展方法：

- Get(ISession, String)
- GetInt32(ISession, String)
- GetString(ISession, String)
- SetInt32(ISession, String, Int32)

- SetString(ISession, String, String)

以下示例在 Razor Pages 页中检索 `IndexModel.SessionKeyName` 键(示例应用中的 `_Name`)的会话值:

```
@page
@using Microsoft.AspNetCore.Http
@model IndexModel

...

Name: @HttpContext.Session.GetString(IndexModel.SessionKeyName)
```

以下示例显示如何设置和获取整数和字符串:

```
public class IndexModel : PageModel
{
    public const string SessionKeyName = "_Name";
    public const string SessionKeyAge = "_Age";
    const string SessionKeyTime = "_Time";

    public string SessionInfo_Name { get; private set; }
    public string SessionInfo_Age { get; private set; }
    public string SessionInfo_CurrentTime { get; private set; }
    public string SessionInfo_SessionTime { get; private set; }
    public string SessionInfo_MiddlewareValue { get; private set; }

    public void OnGet()
    {
        // Requires: using Microsoft.AspNetCore.Http;
        if (string.IsNullOrEmpty(HttpContext.Session.GetString(SessionKeyName)))
        {
            HttpContext.Session.SetString(SessionKeyName, "The Doctor");
            HttpContext.Session.SetInt32(SessionKeyAge, 773);
        }

        var name = HttpContext.Session.GetString(SessionKeyName);
        var age = HttpContext.Session.GetInt32(SessionKeyAge);
```

```
public class HomeController : Controller
{
    const string SessionKeyName = "_Name";
    const string SessionKeyYearsMember = "_YearsMember";
    const string SessionKeyDate = "_Date";

    public IActionResult Index()
    {
        // Requires using Microsoft.AspNetCore.Http;
        HttpContext.Session.SetString(SessionKeyName, "Rick");
        HttpContext.Session.SetInt32(SessionKeyYearsMember, 3);

        return RedirectToAction("SessionNameYears");
    }

    public IActionResult SessionNameYears()
    {
        var name = HttpContext.Session.GetString(SessionKeyName);
        var yearsMember = HttpContext.Session.GetInt32(SessionKeyYearsMember);

        return Content($"Name: \"{name}\",  Membership years: \"{yearsMember}\"");
    }
```

必须对所有会话数据进行序列化以启用分布式缓存方案，即使是在使用内存中缓存的时候。提供最小的字符串和

数字序列化程序(请参阅 ISession 的方法和扩展方法)。用户必须使用另一种机制(例如 JSON)序列化复杂类型。

添加以下扩展方法以设置和获取可序列化的对象:

```
public static class SessionExtensions
{
    public static void Set<T>(this ISession session, string key, T value)
    {
        session.SetString(key, JsonConvert.SerializeObject(value));
    }

    public static T Get<T>(this ISession session, string key)
    {
        var value = session.GetString(key);

        return value == null ? default(T) :
            JsonConvert.DeserializeObject<T>(value);
    }
}
```

```
public static class SessionExtensions
{
    public static void Set<T>(this ISession session, string key, T value)
    {
        session.SetString(key, JsonConvert.SerializeObject(value));
    }

    public static T Get<T>(this ISession session,string key)
    {
        var value = session.GetString(key);
        return value == null ? default(T) :
            JsonConvert.DeserializeObject<T>(value);
    }
}
```

以下示例演示如何使用扩展方法设置和获取可序列化的对象:

```
// Requires you add the Set and Get extension method mentioned in the topic.
if (HttpContext.Session.Get<DateTime>(SessionKeyTime) == default(DateTime))
{
    HttpContext.Session.Set<DateTime>(SessionKeyTime, currentTime);
}
```

```
public IActionResult SetDate()
{
    // Requires you add the Set extension method mentioned in the topic.
    HttpContext.Session.Set<DateTime>(SessionKeyDate, DateTime.Now);

    return RedirectToAction("GetDate");
}

public IActionResult GetDate()
{
    // Requires you add the Get extension method mentioned in the topic.
    var date = HttpContext.Session.Get<DateTime>(SessionKeyDate);
    var sessionTime = date.TimeOfDay.ToString();
    var currentTime = DateTime.Now.TimeOfDay.ToString();

    return Content($"Current time: {currentTime} - "
                 + $"session time: {sessionTime}");
}
```

TempData

ASP.NET Core 公开 Razor Pages 页模型的 TempData 属性或 MVC 控制器的 TempData。此属性存储未读取的数据。Keep 和 Peek 方法可用于检查数据，而不执行删除。多个请求需要数据时，TempData 非常有助于进行重定向。使用 Cookie 或会话状态通过 TempData 提供程序实现 TempData。

TempData 提供程序

ASP.NET Core 2.0 或更高版本中，默认使用基于 Cookie 的 TempData 提供程序，以将 TempData 存储在 Cookie 中。

使用由 Base64UrlTextEncoder 编码的 IDataProtector 对 Cookie 数据进行加密，然后进行分块。因为 Cookie 进行了分块，所以 ASP.NET Core 1.x 中的单个 Cookie 大小限制不适用。未压缩 Cookie 数据，因为压缩加密的数据会导致安全问题，如 CRIME 和 BREACH 攻击。有关基于 Cookie 的 TempData 提供程序的详细信息，请参阅 CookieTempDataProvider。

在 ASP.NET Core 1.0 和 1.1 中，会话状态 TempData 提供程序是默认提供程序。

选择 TempData 提供程序

选择 TempData 提供程序涉及几个注意事项，例如：

1. 应用是否已使用会话状态？ 如果是，使用会话状态 TempData 提供程序对应用没有额外的成本（除了数据的大小）。
2. 应用是否只对相对较小的数据量（最多 500 个字节）使用 TempData？ 如果是，Cookie TempData 提供程序将为每个携带 TempData 的请求增加较小的成本。如果不是，会话状态 TempData 提供程序有助于在使用 TempData 前，避免在每个请求中来回切换大量数据。
3. 应用是否在多个服务器上的服务器场中运行？ 如果是，无需其他任何配置，即可在数据保护外使用 Cookie TempData 提供程序（请参阅 ASP.NET Core 数据保护 和密钥存储提供程序）。

> **NOTE**
>
> 大多数 Web 客户端（如 Web 浏览器）针对每个 Cookie 的最大大小和/或 Cookie 总数强制实施限制。使用 Cookie TempData 提供程序时，请验证应用未超过这些限制。考虑数据的总大小。解释加密和分块导致的 Cookie 大小增加。

配置 TempData 提供程序

默认情况下启用基于 Cookie 的 TempData 提供程序。

若要启用基于会话的 TempData 提供程序，请使用 AddSessionStateTempDataProvider 扩展方法：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1)
        .AddSessionStateTempDataProvider();

    services.AddSession();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseSession();
    app.UseMvc();
}
```

以下 `Startup` 类代码配置基于会话的 TempData 提供程序：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddSession();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    app.UseSession();
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

中间件的顺序很重要。在前面的示例中，在 `UseMvc` 之后调用 `UseSession` 时会发生 `InvalidOperationException` 异常。有关详细信息，请参阅中间件排序。

> **IMPORTANT**
>
> 如果面向 .NET Framework 并使用基于会话的 TempData 提供程序，请将 Microsoft.AspNetCore.Session 包添加到项目。

查询字符串

可以将有限的数据从一个请求传递到另一个请求，方法是将其添加到新请求的查询字符串中。这有利于以一种持久的方式捕获状态，这种方式允许通过电子邮件或社交网络共享嵌入式状态的链接。由于 URL 查询字符串是公共的，因此请勿对敏感数据使用查询字符串。

除了意外的共享，在查询字符串中包含数据还会为跨站点请求伪造 (CSRF) 攻击创造机会，从而欺骗用户在通过身份验证时访问恶意网站。然后，攻击者可以从应用中窃取用户数据，或者代表用户采取恶意操作。任何保留的应用或会话状态必须防止 CSRF 攻击。有关详细信息，请参阅预防跨网站请求伪造 (XSRF/CSRF) 攻击。

隐藏字段

数据可以保存在隐藏的表单域中，并在下一个请求上回发。这在多页窗体中很常见。由于客户端可能篡改数据，因此应用必须始终重新验证存储在隐藏字段中的数据。

HttpContext.Items

处理单个请求时，使用 HttpContext.Items 集合存储数据。处理请求后，放弃集合的内容。通常使用 `Items` 集合允许组件或中间件在请求期间在不同时间点操作且没有直接传递参数的方法时进行通信。

在下面示例中，中间件将 `isVerified` 添加到 `Items` 集合。

```
app.Use(async (context, next) =>
{
    // perform some verification
    context.Items["isVerified"] = true;
    await next.Invoke();
});
```

然后，在管道中，另一个中间件可以访问 `isVerified` 的值：

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync($"Verified: {context.Items["isVerified"]}");
});
```

对于只供单个应用使用的中间件，`string` 键是可以接受的。应用实例间共享的中间件应使用唯一的对象键以避免键冲突。以下示例演示如何使用中间件类中定义的唯一对象键：

```
public class HttpContextItemsMiddleware
{
    private readonly RequestDelegate _next;
    public static readonly object HttpContextItemsMiddlewareKey = new Object();

    public HttpContextItemsMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        httpContext.Items[HttpContextItemsMiddlewareKey] = "K-9";

        await _next(httpContext);
    }
}

public static class HttpContextItemsMiddlewareExtensions
{
    public static IApplicationBuilder
        UseHttpContextItemsMiddleware(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<HttpContextItemsMiddleware>();
    }
}
```

```
// You may need to install the Microsoft.AspNetCore.Http.Abstractions package
public class HttpContextItemsMiddleware
{
    private readonly RequestDelegate _next;
    public static readonly object HttpContextItemsMiddlewareKey = new Object();

    public HttpContextItemsMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        httpContext.Items[HttpContextItemsMiddlewareKey] = "K-9";

        await _next(httpContext);
    }
}

public static class HttpContextItemsMiddlewareExtensions
{
    public static IApplicationBuilder
        UseHttpContextItemsMiddleware(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<HttpContextItemsMiddleware>();
    }
}
```

其他代码可以使用通过中间件类公开的键访问存储在 `HttpContext.Items` 中的值：

```
HttpContext.Items
    .TryGetValue(HttpContextItemsMiddleware.HttpContextItemsMiddlewareKey,
        out var middlewareSetValue);
SessionInfo_MiddlewareValue =
    middlewareSetValue?.ToString() ?? "Middleware value not set!";
```

```
public IActionResult GetMiddlewareValue()
{
    var middlewareSetValue =
        HttpContext.Items[HttpContextItemsMiddleware.HttpContextItemsMiddlewareKey];

    return Content($"Value: {middlewareSetValue}");
}
```

此方法还有避免在代码中使用关键字符串的优势。

缓存

缓存是存储和检索数据的有效方法。应用可以控制缓存项的生存期。

缓存数据未与特定请求、用户或会话相关联。请**注意不要缓存可能由其他用户请求检索的特定于用户的数据。**

有关更多信息，请参见响应缓存在 ASP.NET Core。

依赖关系注入

使用依赖关系注入可向所有用户提供数据：

1. 定义一项包含数据的服务。例如，定义一个名为 `MyAppData` 的类：

```
public class MyAppData
{
    // Declare properties and methods
}
```

2. 将服务类添加到 `Startup.ConfigureServices`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<MyAppData>();
}
```

3. 使用数据服务类：

```
public class IndexModel : PageModel
{
    public IndexModel(MyAppData myService)
    {
        // Do something with the service
        //    Examples: Read data, store in a field or property
    }
}
```

```
public class HomeController : Controller
{
    public HomeController(MyAppData myService)
    {
        // Do something with the service
        //    Examples: Read data, store in a field or property
    }
}
```

常见错误

- “在尝试激活‘Microsoft.AspNetCore.Session.DistributedSessionStore’时无法为类型‘Microsoft.Extensions.Caching.Distributed.IDistributedCache’解析服务。”

  这通常是由于不能配置至少一个 `IDistributedCache` 实现而造成的。有关详细信息，请参阅 分布式缓存在 ASP.NET Core 中 和 缓存在内存中 ASP.NET Core。

- 在会话中间件保存会话失败的事件中（例如，如果后备存储不可用），中间件记录异常而请求继续正常进行。这会导致不可预知的行为。

  例如，用户将购物车存储在会话中。用户将商品添加到购物车，但提交失败。应用不知道有此失败，因此它向用户报告商品已添加到购物车，但事实并非如此。

  检查此类错误的建议方法是完成将应用写入到该会话后，从应用代码调用 `await feature.Session.CommitAsync();`。如果后备存储不可用，则 `CommitAsync` 引发异常。如果 `CommitAsync` 失败，应用可以处理异常。在与数据存储不可用的相同的条件下，`LoadAsync` 引发异常。

其他资源

在 Web 场中托管 ASP.NET Core

ASP.NET Core 中的标记帮助程序

2018/10/29 • • Edit Online

作者：Rick Anderson

什么是标记帮助程序？

标记帮助程序使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。例如，内置 `ImageTagHelper` 可以将版本号追加到映像名称。每当映像发生变化时，服务器都会为映像生成一个新的唯一版本，因此客户端总能获得当前映像（而不是过时的缓存映像）。有多种常见任务（例如创建窗体、链接，加载资产等）的内置标记帮助程序，公共 GitHub 存储库和 NuGet 包中甚至还有更多可用标记帮助程序。标记帮助程序使用 C# 创建，基于元素名称、属性名称或父标记以 HTML 元素为目标。例如，应用 `LabelTagHelper` 属性时，内置 `LabelTagHelper` 可以 HTML `<label>` 元素为目标。如果熟悉 HTML 帮助程序，则标记帮助程序将减少 Razor 视图中 HTML 和 C# 之间的显式转换。在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。标记帮助程序与 HTML 帮助程序的比较更详细地介绍了两者之间的差异。

标记帮助程序的功能

HTML 友好开发体验 在多数情况下，使用标记帮助程序的 Razor 标记看起来像是标准 HTML。熟悉 HTML/CSS/JavaScript 的前端设计师，无需学习 C# Razor 语法即可编辑 Razor。

用于创建 HTML 和 Razor 标记的丰富 IntelliSense 环境 这与 HTML 帮助程序形成鲜明对比，HTML 帮助程序是 Razor 视图中标记的曾用服务器端创建方法。标记帮助程序与 HTML 帮助程序的比较更详细地介绍了两者之间的差异。标记帮助程序的 IntelliSense 支持解释了 IntelliSense 环境。即使是熟悉 Razor C# 语法的开发人员，使用标记帮助程序也比编写 C# Razor 标记更高效。

使用仅在服务器上可用的信息，可提高生产力，并能生成更稳定、可靠和可维护的代码 例如，过去更新映像时，必须在更改映像时更改映像名称。出于性能原因，要主动缓存映像，而若不更改映像的名称，客户端就可能获得过时的副本。以前，编辑完映像后，必须更改名称，而且需要更新 Web 应用中对该映像的每个引用。这不仅大费周章，还容易出错（可能会漏掉某个引用、意外输入错误的字符串等等）内置 `ImageTagHelper` 可自动执行此操作。`ImageTagHelper` 可将版本号追加到映像名称，这样每当映像出现更改时，服务器都会自动为该映像生成新的唯一版本。客户端总是能获得最新映像。使用 `ImageTagHelper` 实质上是免费获得稳健性而节省劳动力。

大多数内置标记帮助程序以标准 HTML 元素为目标，为该元素提供服务器端属性。例如，`<input>` 用于包含 `asp-for` 特性的"视图/帐户"文件夹中的很多视图。此特性将指定模型属性的名称提取至所呈现的 HTML。以一个具备以下模型的 Razor 视图为例：

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
}
```

以下 Razor 标记：

```
<label asp-for="Movie.Title"></label>
```

则会生成以下 HTML：

```
<label for="Movie_Title">Title</label>
```

通过 LabelTagHelper 中的 `For` 属性，可使用 `asp-for` 特性。请参阅创作标记帮助程序，获取详细信息。

管理标记帮助程序作用域

标记帮助程序作用域由 `@addTagHelper`、`@removeTagHelper` 和"!"选择退出字符联合控制。

使用 `@addTagHelper` 添加标记帮助程序

如果创建名为 AuthoringTagHelpers 的新 ASP.NET Core Web 应用，将向项目添加以下 Views/_ViewImports.cshtml 文件：

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

`@addTagHelper` 指令让视图可以使用标记帮助程序。在此示例中，视图文件是 Pages/_ViewImports.cshtml，"Pages"文件夹及其子目录中的所有视图文件都会默认继承它，使得标记帮助程序可用。上面的代码使用通配符语法("*")，指定程序集 (Microsoft.AspNetCore.Mvc.TagHelpers) 中的所有标记帮助程序对于 Views 目录或子目录中的所有视图文件均可用。`@addTagHelper` 后第一个参数指定要加载的标记帮助程序(我们使用"*"指定加载所有标记帮助程序)，第二个参数"Microsoft.AspNetCore.Mvc.TagHelpers"指定包含标记帮助程序的程序集。Microsoft.AspNetCore.Mvc.TagHelpers 是内置 ASP.NET Core 标记帮助程序的程序集。

要公开此项目中的所有标记帮助程序(将创建名为 AuthoringTagHelpers 的程序集)，可使用以下内容：

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

如果项目包含具有默认命名空间 (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`) 的 `EmailTagHelper`，则可提供标记帮助程序的完全限定名称 (FQN)：

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers
```

若要使用 FQN 将标记帮助程序添加到视图，请首先添加 FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`)，然后添加程序集名称 (AuthoringTagHelpers)。大多数开发人员更喜欢使用"*"通配符语法。使用通配符语法，可在 FQN 中插入通配符"*"作为后缀。例如，以下任何指令都将引入 `EmailTagHelper`：

```
@addTagHelper AuthoringTagHelpers.TagHelpers.E*, AuthoringTagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.Email*, AuthoringTagHelpers
```

如前所述，将 `@addTagHelper` 指令添加到 Views/_ViewImports.cshtml 文件，将使标记帮助程序对于 Views 目录及子目录中的所有视图文件可用。如果想选择仅对特定视图公开标记帮助程序，可在这些视图文件中使用 `@addTagHelper` 指令。

`@removeTagHelper` 删除标记帮助程序

`@removeTagHelper` 与 `@addTagHelper` 具有相同的两个参数，它会删除之前添加的标记帮助程序。例如，应用于特定视图的 `@removeTagHelper` 会删除该视图中的指定标记帮助程序。在 Views/Folder/_ViewImports.cshtml 文件中使用 `@removeTagHelper`，将从 Folder 中的所有视图删除指定的标记帮助程序。

使用 _ViewImports.cshtml 文件控制标记帮助程序作用域

可将 _ViewImports.cshtml 添加到任何视图文件夹，视图引擎将同时应用该文件和 Views/_ViewImports.cshtml 文件中的指令。如果为 Home 视图添加空的 Views/Home/_ViewImports.cshtml 文件，则不会发生任何更改，因为 _ViewImports.cshtml 文件是附加的。添加到 Views/Home/_ViewImports.cshtml 文件（不在默认 Views/_ViewImports.cshtml 文件中）的任何 `@addTagHelper` 指令，都只会将这些标记帮助程序公开给 Home 文件夹中的视图。

选择退出各个元素

使用标记帮助程序选择退出字符（"!"），可在元素级别禁用标记帮助程序。例如，使用标记帮助程序选择退出字符在 `<span>` 中禁用 `Email` 验证：

```
<!span asp-validation-for="Email" class="text-danger"></!span>
```

须将标记帮助程序选择退出字符应用于开始和结束标记。（将选择退出字符添加到开始标记时，Visual Studio 编辑器会自动为结束标记添加相应字符）。添加选择退出字符后，元素和标记帮助程序属性不再以独特字体显示。

使用 `@tagHelperPrefix` 阐明标记帮助程序用途

`@tagHelperPrefix` 指令可指定一个标记前缀字符串，以启用标记帮助程序支持并阐明标记帮助程序用途。例如，可以将以下标记添加到 Views/_ViewImports.cshtml 文件：

```
@tagHelperPrefix th:
```

在以下代码图像中，标记帮助程序前缀设置为 `th:`，所以只有使用前缀 `th:` 的元素才支持标记帮助程序（可使用标记帮助程序的元素以独特字体显示）。`<label>` 和 `<input>` 元素具有标记帮助程序前缀，可使用标记帮助程序，而 `<span>` 元素则相反。

```
<div class="form-group">
    <th:label asp-for="Password" class="col-md-2"></th:label>
    <div class="col-md-10">
        <th:input asp-for="Password" class="form-control" />
        <span asp-validation-for="Password" class="text-danger"></span>
    </div>
</div>
```

适用于 `@addTagHelper` 的层次结构规则也适用于 `@tagHelperPrefix`。

自结束标记帮助程序

许多标记帮助程序都不能用作自结束标记。某些标记帮助程序被设计为自结束标记。使用未被

设计为自结束的标记帮助程序会抑制呈现的输出。自结束标记帮助程序会在呈现的输出中生成自结束标记。有关详细信息，请在创作标记帮助程序中查看此问题。

标记帮助程序的 Intellisense 支持

在 Visual Studio 中创建新的 ASP.NET Core Web 应用时，会添加 NuGet 包“Microsoft.AspNetCore.Razor.Tools”。这是添加标记帮助程序工具的包。

请考虑编写 HTML `<label>` 元素。只要在 Visual Studio 编辑器中输入 `<l`，IntelliSense 就会显示匹配的元素：

不仅会获得 HTML 帮助，还会有图标（下方带有“<>”的“@" symbol with "）

将该元素标识为标记帮助程序的目标。纯 HTML 元素（如 `fieldset`）显示“<>”图标。

纯 HTML `<label>` 标记以棕色字体显示 HTML 标记（使用默认 Visual Studio 颜色主题时），以红色字体显示属性，并以蓝色字体显示属性值。

```
<label class="col-md-2">Email</label>
```

输入 `<label` 后，IntelliSense 会列出可用的 HTML/CSS 属性和以标记帮助程序为目标的属性：

通过 IntelliSense 语句完成功能，按 Tab 键即可用选择的值完成语句：

只要输入标记帮助程序属性，标记和属性字体就会更改。如果使用默认的 Visual Studio“蓝色”或“浅色”颜色主题，则字体是粗体紫色。如果使用“深色”主题，则字体为粗体青色。本文档中的图像在使用默认主题时截取的。

```
<label asp-for
```

可在双引号 ("") 内输入 Visual Studio CompleteWord 快捷方式(默认值为 Ctrl+空格键)，即可使用 C#，就像在 C# 类中一样。IntelliSense 会显示页面模型上的所有方法和属性。由于属性类型是 `ModelExpression`，所以这些方法和属性可用。在下图中，我正在编辑 `Register` 视图，所以 `RegisterViewModel` 是可用的。

IntelliSense 会列出页面上模型可用的属性和方法。丰富 IntelliSense 环境可帮助选择 CSS 类：

标记帮助程序与 HTML 帮助程序的比较

标记帮助程序附加到 Razor 视图中的 HTML 元素，HTML 帮助程序是作为与 Razor 视图中 HTML 交织的方法被调用的。请考虑下列 Razor 标记，它创建具有 CSS 类"caption"的 HTML 标签：

```
@Html.Label("FirstName", "First Name:", new {@class="caption"})
```

艾特 (`@`) 符号告诉 Razor 这是代码的开始。接下来的两个参数("FirstName"和"First Name:")是字符串，所以 IntelliSense 无法提供帮助。最后一个参数：

```
new {@class="caption"}
```

是用于表示属性的匿名对象。由于 class 是 C# 中的保留关键字，因此要使用 `@` 符号强制 C# 将"@class="解释为符号(属性名称)。对于前端设计师(熟悉 HTML/CSS/JavaScript 及其他客户端技术，但不熟悉 C# 和 Razor 的人)，这行代码中大部分内容是陌生是。必须在没有 IntelliSense 帮助的情况下编写整行代码。

使用 `LabelTagHelper`，相同标记可以编写为：

```
<label class="caption" asp-for="FirstName"></label>
```

使用标记帮助程序版本，只要在 Visual Studio 编辑器中输入 `<l`，IntelliSense 就会显示匹配的

元素：

IntelliSense 可帮助编写整行。`LabelTagHelper` 也默认将 `asp-for` 属性值（"FirstName"）的内容设置为"First Name"：即在属性值中每个大写字母前添加一个空格，将驼峰式大小写的属性转换为由属性名称组成的语句。在下列标记中：

```
<label class="caption" asp-for="FirstName"></label>
```

生成：

```
<label class="caption" for="FirstName">First Name</label>
```

如果将内容添加到 `<label>`，则不会使用由驼峰式大小写转换为语句式书写的内容。例如：

```
<label class="caption" asp-for="FirstName">Name First</label>
```

生成：

```
<label class="caption" for="FirstName">Name First</label>
```

以下代码图像显示了由 Visual Studio 2015 包含的旧版 ASP.NET 4.5.x MVC 模板生成的 Views/Account/Register.cshtml Razor 视图的 Form 部分。

```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizo
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}
```

Visual Studio 编辑器以灰色背景显示 C# 代码。例如，`AntiForgeryToken` HTML 帮助程序：

```
@Html.AntiForgeryToken()
```

以灰色背景显示。Register 视图中的标记大部分是 C#。将其与使用标记帮助程序的等效方法进行比较：

```
<form asp-controller="Account" asp-action="Register" method="post" class="form-hori
    <h4>Create a new account.</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Password" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Password" class="form-control" />
            <span asp-validation-for="Password" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="ConfirmPassword" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="ConfirmPassword" class="form-control" />
            <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Register</button>
        </div>
    </div>
</form>
```

与 HTML 帮助程序方法相比，此标记更清晰，更容易阅读、编辑和维护。C# 代码会被减少至服务器需要知道的最小值。Visual Studio 编辑器以独特的字体显示标记帮助程序的目标标记。

请考虑 Email 组：

```
<div class="form-group">
    <label asp-for="Email" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="Email" class="form-control" />
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>
</div>
```

每个"asp-"属性都有一个"Email"值，但是"Email"不是字符串。在此上下文中，"Email"是 `RegisterViewModel` 的 C# 模型表达式属性。

Visual Studio 编辑器可帮助编写注册窗体的标记帮助程序方法中的所有标记，而 Visual Studio 不会为 HTML 帮助程序方法中的大多数代码提供帮助。标记帮助程序的 IntelliSense 支持详细介绍了如何在 Visual Studio 编辑器中使用标记帮助程序。

标记帮助程序与 Web 服务器控件的比较

- 标记帮助程序不拥有与其相关的元素：它们只是参与元素和内容的呈现。ASP.NET Web 服务器控件在页面上进行声明和调用。
- Web 服务器控件具有可观的生命周期，因而难以进行开发和调试。
- 通过 Web 服务器控件，可使用客户端控件向客户端文档对象模型 (DOM) 元素添加功能。标记帮助程序没有 DOM。
- Web 服务器控件包括自动浏览器检测。标记帮助程序不了解浏览器。
- 通常不能撰写 Web 服务器控件时，多个标记帮助程序可作用于同一元素（请参阅避免标记帮助程序冲突）。
- 标记帮助程序可以修改其作用域内 HTML 元素的标记和内容，但不会直接修改页面上的其他内容。Web 服务器控件的作用域较广，并且可以执行影响页面其他部分的操作，从而可能造成意想不到的副作用。
- Web 服务器控件使用类型转换器将字符串转换为对象。使用标记帮助程序时，本身就用 C# 语言工作，因此无需进行类型转换。
- Web 服务器控件使用 System.ComponentModel 实现组件和控件的运行时和设计时行为。`System.ComponentModel` 包括用于属性和类型转换器的实现、数据源绑定和组件授权的基类和接口。与通常派生自 `TagHelper` 的标记帮助程序相比，`TagHelper` 基类仅公开两个方法，即 `Process` 和 `ProcessAsync`。

自定义标记帮助程序元素字体

可以在"工具" > "选项" > "环境" > "字体和颜色"中自定义字体和着色：

其他资源

- 创作标记帮助程序
- 使用窗体
- GitHub 上的 TagHelperSamples 包含用于处理 Bootstrap 的标记帮助程序示例。

在 ASP.NET Core 中创作标记帮助程序

2018/12/14 • • Edit Online

作者：Rick Anderson

查看或下载示例代码（如何下载）

标记帮助程序入门

本教程介绍标记帮助程序编程。标记帮助程序简介描述了标记帮助程序提供的优势。

标记帮助程序是实现 `ITagHelper` 接口的任何类。但是，在创作标记帮助程序时，通常从 `TagHelper` 派生，这样可以访问 `Process` 方法。

1. 创建一个名为 AuthoringTagHelpers 的新 ASP.NET Core 项目。此项目不需要身份验证。
2. 创建一个名为“TagHelpers”的文件夹来保存标记帮助程序。“TagHelpers”文件夹不是必需的，但它是合理的约定。现在让我们开始编写一些简单的标记帮助程序。

最小的标记帮助程序

在本部分中，你将编写一个更新电子邮件标记的标记帮助程序。例如：

```
<email>Support</email>
```

服务器将使用电子邮件标记帮助程序将该标记转换为以下内容：

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

即，使其成为电子邮件链接的定位标记。如果你正在编写博客引擎，并且需要它将营销、支持和其他联系人的电子邮件全部发送到同一个域，则可能需要执行此操作。

1. 将以下 `EmailTagHelper` 类添加到“TagHelpers”文件夹。

```
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Threading.Tasks;

namespace AuthoringTagHelpers.TagHelpers
{
    public class EmailTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.TagName = "a";    // Replaces <email> with <a> tag
        }
    }
}
```

- 标记帮助程序使用面向根类名称的元素的命名约定（减去类名称的 TagHelper 部分）。在此示例中，EmailTagHelper 的根名称是 email，因此 `<email>` 标记将作为目标名称。此命名约定应适用于大多数标记帮助程序，稍后将介绍如何重写它。

- `EmailTagHelper` 类派生自 `TagHelper`。`TagHelper` 类提供编写标记帮助程序的方法和属性。
- 重写的 `Process` 方法控制标记帮助程序在执行时的操作。`TagHelper` 类还提供具有相同参数的异步版本（`ProcessAsync`）。
- `Process`（和 `ProcessAsync`）的上下文参数包含与执行当前 HTML 标记相关的信息。
- `Process`（和 `ProcessAsync`）的输出参数包含监控状态的 HTML 元素，它代表用于生成 HTML 标记和内容的原始源。
- 类名称的后缀是 TagHelper，这不是必需的，但被认为是最佳做法约定。可将类声明为：

```
public class Email : TagHelper
```

2. 要使 `EmailTagHelper` 类可用于所有 Razor 视图，请将 `addTagHelper` 指令添加到 Views/_ViewImports.cshtml 文件：[!code-html]

   上面的代码使用通配符语法来指定程序集中的所有标记帮助程序都将可用。`@addTagHelper` 之后的第一个字符串指定要加载的标记帮助程序（对所有标记帮助程序使用"*"），第二个字符串"AuthoringTagHelpers"指定标记帮助程序所在的程序集。另请注意，第二行使用通配符语法引入了 ASP.NET Core MVC 标记帮助程序（标记帮助程序简介中讨论了这些帮助程序。）要使标记帮助程序可用于 Razor 视图，请使用 `@addTagHelper` 指令。或者，也可以提供标记帮助程序的完全限定的名称 (FQN)，如下所示：

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers
```

若要使用 FQN 将标记帮助程序添加到视图中，请依次添加 FQN（`AuthoringTagHelpers.TagHelpers.EmailTagHelper`）和程序集名称（AuthoringTagHelpers，不一定是 `namespace`）。大多数开发者更喜欢使用通配符语法。标记帮助程序简介详细介绍了标记帮助程序的添加和删除方法，以及层次结构和通配符语法。

1. 使用以下更改更新 Views/Home/Contact.cshtml 文件中的标记：

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>
```

2. 运行应用并使用你喜爱的浏览器来查看 HTML 源，以便验证电子邮件标记是否替换为定位标记（例如，`<a>Support</a>`）。Support 和 Marketing 呈现为链接，但它们不具备使其正常工作的 `href` 属性。此问题将在下一部分得以解决。

SetAttribute 和 SetContent

在本部分中，我们将更新 `EmailTagHelper`，使其能够为电子邮件创建有效的定位标记。我们将对其进行更新以获取 Razor 视图中的信息（采用 `mail-to` 属性的形式）并使用该信息来生成定位点。

使用以下内容更新 `EmailTagHelper` 类：

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";

    // Can be passed via <email mail-to="..." />.
    // Pascal case gets translated into lower-kebab-case.
    public string MailTo { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a";    // Replaces <email> with <a> tag

        var address = MailTo + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + address);
        output.Content.SetContent(address);
    }
}
```

- 标记帮助程序采用 Pascal 大小写格式的类和属性名将转换为各自相应的小写短横线格式。因此，要使用 `MailTo` 属性，请使用 `<email mail-to="value"/>` 等效项。
- 最后一行为最小功能标记帮助程序设置已完成的内容。
- 突出显示的行显示了添加属性的语法：

```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    output.TagName = "a";    // Replaces <email> with <a> tag

    var address = MailTo + "@" + EmailDomain;
    output.Attributes.SetAttribute("href", "mailto:" + address);
    output.Content.SetContent(address);
}
```

只要属性集合中当前不存在“href”属性，该方法就适用于此属性。也可使用 `output.Attributes.Add` 方法将标记帮助程序属性添加到标记属性集合的末尾。

1. 使用以下更改更新 Views/Home/Contact.cshtml 文件中的标记：[!code-html]
2. 运行应用并验证它是否生成正确的链接。

> **NOTE**
>
> 如果打算编写电子邮件标记自结束（`<email mail-to="Rick" />`），最终输出也将为自结束。要启用只使用开始标记（`<email mail-to="Rick">`）来编写标记的功能，必须用以下内容修饰类：
>
> ```
> [HtmlTargetElement("email", TagStructure = TagStructure.WithoutEndTag)]
> public class EmailVoidTagHelper : TagHelper
> {
>  private const string EmailDomain = "contoso.com";
>  // Code removed for brevity
> ```

鉴于自结束电子邮件标记帮助程序，输出将为 `<a href="mailto:Rick@contoso.com" />`。自结束定位标记不是有效的 HTML，因此你不想创建这样的标记，但你可能想要创建一个自结束的标记帮助程序。标记帮助程序在读取标

记后设置 `TagMode` 属性的类型。

ProcessAsync

在本部分中，我们将编写异步电子邮件帮助程序。

1. 将 `EmailTagHelper` 类替换为以下代码：

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a";                                // Replaces <email> with <a> tag
        var content = await output.GetChildContentAsync();
        var target = content.GetContent() + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + target);
        output.Content.SetContent(target);
    }
}
```

   注意：

   - 此版本使用异步 `ProcessAsync` 方法。异步 `GetChildContentAsync` 返回包含 `TagHelperContent` 的 `Task`。
   - 使用 `output` 参数获取 HTML 元素的内容。

2. 对 Views/Home/Contact.cshtml 文件进行以下更改，以便标记帮助程序能够获取目标电子邮件。

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>
```

3. 运行应用并验证它是否生成有效的电子邮件链接。

RemoveAll、PreContent.SetHtmlContent 和 PostContent.SetHtmlContent

1. 将以下 `BoldTagHelper` 类添加到“TagHelpers”文件夹。

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = "bold")]
    public class BoldTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.Attributes.RemoveAll("bold");
            output.PreContent.SetHtmlContent("<strong>");
            output.PostContent.SetHtmlContent("</strong>");
        }
    }
}
```

- `[HtmlTargetElement]` 属性传递一个属性参数，该参数指定包含名为“bold”的 HTML 属性的任何 HTML 元素都将匹配，并且该类中的 `Process` 重写方法将会运行。在此示例中，`Process` 方法删除了“bold”属性，并用 `<strong></strong>` 围住了包含的标记。
- 因为你不想替换现有的标记内容，所以必须用 `PreContent.SetHtmlContent` 方法编写开头的 `<strong>` 标记，并用 `PostContent.SetHtmlContent` 方法编写结尾的 `</strong>` 标记。

2. 修改 About.cshtml 视图，以包含 `bold` 属性值。完成的代码如下所示。

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>
```

3. 运行应用。可以使用你喜爱的浏览器来检查源并验证标记。

   上面的 `[HtmlTargetElement]` 属性仅针对提供属性名称“bold”的 HTML 标记。标记帮助程序未修改 `<bold>` 元素。

4. 标注出 `[HtmlTargetElement]` 属性行，它将默认为目标 `<bold>` 标记，也就是 `<bold>` 格式的 HTML 标记。请记住，默认的命名约定会将类名称 BoldTagHelper 与 `<bold>` 标记相匹配。

5. 运行应用并验证 `<bold>` 标记是否由标记帮助程序进行处理。

用多个 `[HtmlTargetElement]` 属性修饰类会导致目标出现逻辑 OR。例如，使用下面的代码时，系统将匹配出 bold 标记或 bold 属性。

```
[HtmlTargetElement("bold")]
[HtmlTargetElement(Attributes = "bold")]
public class BoldTagHelper : TagHelper
{
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.Attributes.RemoveAll("bold");
        output.PreContent.SetHtmlContent("<strong>");
        output.PostContent.SetHtmlContent("</strong>");
    }
}
```

将多个属性添加到同一语句时，运行时会将其视为逻辑 AND。例如，在下面的代码中，HTML 元素必须命名为“bold”并具有名为“bold”的属性（`<bold bold />`）才能匹配。

```
[HtmlTargetElement("bold", Attributes = "bold")]
```

也可使用 `[HtmlTargetElement]` 更改目标元素的名称。例如，如果你希望 `BoldTagHelper` 以 `<MyBold>` 标记为目标，则可使用以下属性：

```
[HtmlTargetElement("MyBold")]
```

将模型传递到标记帮助程序

1. 添加“Models”文件夹。
2. 将以下 `WebsiteContext` 类添加到“模型”文件夹：

```
using System;

namespace AuthoringTagHelpers.Models
{
    public class WebsiteContext
    {
        public Version Version { get; set; }
        public int CopyrightYear { get; set; }
        public bool Approved { get; set; }
        public int TagsToShow { get; set; }
    }
}
```

3. 将以下 `WebsiteInformationTagHelper` 类添加到“TagHelpers”文件夹。

```
using System;
using AuthoringTagHelpers.Models;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    public class WebsiteInformationTagHelper : TagHelper
    {
        public WebsiteContext Info { get; set; }

      public override void Process(TagHelperContext context, TagHelperOutput output)
      {
         output.TagName = "section";
         output.Content.SetHtmlContent(
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
<li><strong>Copyright Year:</strong> {Info.CopyrightYear}</li>
<li><strong>Approved:</strong> {Info.Approved}</li>
<li><strong>Number of tags to show:</strong> {Info.TagsToShow}</li></ul>");
         output.TagMode = TagMode.StartTagAndEndTag;
      }
   }
}
```

- 如前所述，标记帮助程序会将标记帮助程序采用 Pascal 大小写格式的 C# 类名和属性转换为小写短横线格式。因此，要在 Razor 中使用 `WebsiteInformationTagHelper`，请编写 `<website-information />`。

- 未显式标识具有 `[HtmlTargetElement]` 属性的目标元素，因此 `website-information` 的默认值将成为目标元素。如果应用了以下属性（请注意，它虽不是短横线格式，但却与类名相匹配）：

```
[HtmlTargetElement("WebsiteInformation")]
```

小写短横线格式标记 `<website-information />` 不匹配。若要使用 `[HtmlTargetElement]` 属性，请使用短横线格式，如下所示：

```
[HtmlTargetElement("Website-Information")]
```

- 自结束的元素没有任何内容。在此示例中，Razor 标记将使用自结束标记，但标记帮助程序将创建 section 元素（这不是自结束元素，并且你将在 `section` 元素中编写内容）。因此，需要将 `TagMode` 设置为 `StartTagAndEndTag` 以写入输出。或者，可以标注出行设置 `TagMode` 并用结束标记编写标记。（本教程后面将提供示例标记。）

- 下一行中的 `$`（美元符号）使用内插字符串：

```
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
```

4. 将以下标记添加到 About.cshtml 视图。突出显示的标记显示 Web 站点信息。

```
@using AuthoringTagHelpers.Models
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>

<h3> web site info </h3>
<website-information info="new WebsiteContext {
                                    Version = new Version(1, 3),
                                    CopyrightYear = 1638,
                                    Approved = true,
                                    TagsToShow = 131 }" />
```

> **NOTE**
>
> 在 Razor 中，标记如下所示：
>
> ```
> <website-information info="new WebsiteContext {
> Version = new Version(1, 3),
> CopyrightYear = 1638,
> Approved = true,
> TagsToShow = 131 }" />
> ```
>
> Razor 知道 `info` 属性是一个类，而不是字符串，并且你想要编写 C# 代码。编写任何非字符串标记帮助程序属性时，都不应使用 `@` 字符。

5. 运行应用，并导航到“关于”视图查看 Web 站点信息。

NOTE

可使用带有结束标记的以下标记，并在标记帮助程序中删除带有 `TagMode.StartTagAndEndTag` 的行：

```
<website-information info="new WebsiteContext {
                                    Version = new Version(1, 3),
                                    CopyrightYear = 1638,
                                    Approved = true,
                                    TagsToShow = 131 }" >
</website-information>
```

条件标记帮助程序

条件标记帮助程序在传递 true 值时呈现输出。

1. 将以下 `ConditionTagHelper` 类添加到“TagHelpers”文件夹。

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = nameof(Condition))]
    public class ConditionTagHelper : TagHelper
    {
        public bool Condition { get; set; }

        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            if (!Condition)
            {
                output.SuppressOutput();
            }
        }
    }
}
```

2. 将 Views/Home/Index.cshtml 文件的内容替换为以下标记：

```
@using AuthoringTagHelpers.Models
@model WebsiteContext

@{
    ViewData["Title"] = "Home Page";
}

<div>
    <h3>Information about our website (outdated):</h3>
    <Website-InforMation info="Model" />
    <div condition="Model.Approved">
        <p>
            This website has <strong surround="em">@Model.Approved</strong> been approved yet.
            Visit www.contoso.com for more information.
        </p>
    </div>
</div>
```

3. 将 `Home` 控制器中的 `Index` 方法替换为以下代码：

```
public IActionResult Index(bool approved = false)
{
    return View(new WebsiteContext
    {
        Approved = approved,
        CopyrightYear = 2015,
        Version = new Version(1, 3, 3, 7),
        TagsToShow = 20
    });
}
```

4. 运行应用并浏览到主页。条件 `div` 中的标记不会呈现。将查询字符串 `?approved=true` 追加到 URL（例如，`http://localhost:1235/Home/Index?approved=true`）。`approved` 设置为 true，并将显示条件标记。

> **NOTE**
>
> 使用 nameof 运算符将属性指定为目标，而不是像使用 bold 标记帮助程序那样指定字符串：
>
> ```
> [HtmlTargetElement(Attributes = nameof(Condition))]
> //   [HtmlTargetElement(Attributes = "condition")]
> public class ConditionTagHelper : TagHelper
> {
>  public bool Condition { get; set; }
>
>  public override void Process(TagHelperContext context, TagHelperOutput output)
>  {
>  if (!Condition)
>  {
>  output.SuppressOutput();
>  }
>  }
> }
> ```
>
> 如果代码被重构，nameof 运算符将保护它（可能需要将名称更改为 `RedCondition`）。

避免标记帮助程序冲突

在本部分中，你将编写一对自动链接标记帮助程序。第一个标记帮助程序会将包含以 HTTP 开头的 URL 的标记替换为包含相同 URL 的 HTML 定位标记（从而产生指向 URL 的链接）。第二个标记帮助程序也会对以 WWW 开头的 URL 执行相同的操作。

由于这两个帮助程序密切相关，并且你将来可能会重构它们，因此可将其保存在同一文件中。

1. 将以下 `AutoLinkerHttpTagHelper` 类添加到"TagHelpers"文件夹。

```
[HtmlTargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent.GetContent(),
             @"\b(?:https?://)(\S+)\b",
              "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http link version}
    }
}
```

> **NOTE**
>
> `AutoLinkerHttpTagHelper` 类以 `p` 元素为目标，并使用正则表达式来创建定位点。

2. 将以下标记添加到 Views/Home/Contact.cshtml 文件的末尾：

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>

<p>Visit us at http://docs.asp.net or at www.microsoft.com</p>
```

3. 运行应用并验证标记帮助程序是否正确呈现定位点。

4. 更新 `AutoLinker` 类以包含 `AutoLinkerWwwTagHelper`，这会将 www 文本转换为还包含原始 www 文本的定位标记。更新后的代码在下方突出显示：

```
    [HtmlTargetElement("p")]
    public class AutoLinkerHttpTagHelper : TagHelper
    {
        public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
        {
            var childContent = await output.GetChildContentAsync();
            // Find Urls in the content and replace them with their anchor tag equivalent.
            output.Content.SetHtmlContent(Regex.Replace(
                 childContent.GetContent(),
                 @"\b(?:https?://)(\S+)\b",
                  "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http link version}
        }
    }

    [HtmlTargetElement("p")]
    public class AutoLinkerWwwTagHelper : TagHelper
    {
        public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
        {
            var childContent = await output.GetChildContentAsync();
            // Find Urls in the content and replace them with their anchor tag equivalent.
            output.Content.SetHtmlContent(Regex.Replace(
                childContent.GetContent(),
                 @"\b(www\.)(\S+)\b",
                 "<a target=\"_blank\" href=\"http://$0\">$0</a>"));  // www version
        }
    }
}
```

5. 运行应用。请注意 www 文本呈现为链接，但 HTTP 文本不是。如果将中断点放在这两个类中，可以看到 HTTP 标记帮助程序类首先运行。问题是，标记帮助程序输出已缓存，当运行 WWW 标记帮助程序时，它

会覆盖 HTTP 标记帮助程序的缓存输出。本教程稍后将介绍如何控制标记帮助程序中的运行顺序。我们将用以下方法修复代码：

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent,
             @"\b(?:https?://)(\S+)\b",
              "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http link version}
    }
}

[HtmlTargetElement("p")]
public class AutoLinkerWwwTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent,
             @"\b(www\.)(\S+)\b",
             "<a target=\"_blank\" href=\"http://$0\">$0</a>"));  // www version
    }
}
```

NOTE

在自动链接标记帮助程序的第一版中，使用以下代码获取了目标的内容：

```
var childContent = await output.GetChildContentAsync();
```

也就是说，使用传递到 `ProcessAsync` 方法的 `TagHelperOutput` 调用 `GetChildContentAsync`。如前所述，由于输出已缓存，因此将调用最后运行的标记帮助程序。使用以下代码解决了这个问题：

```
var childContent = output.Content.IsModified ? output.Content.GetContent() :
    (await output.GetChildContentAsync()).GetContent();
```

上面的代码检查内容是否已修改，如果已修改，则从输出缓冲区获取内容。

6. 运行应用并验证这两个链接是否按预期工作。虽然它可能显示自动链接器标记帮助程序是正确且完整的，但它有一个细微的问题。如果首先运行 WWW 标记帮助程序，则 www 链接不正确。通过添加 `Order` 重载更新代码来控制标记运行的顺序。`Order` 属性确定相对于面向同一元素的其他标记帮助程序的执行顺序。默认顺序值为零，并首先执行具有较低值的实例。

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    // This filter must run before the AutoLinkerWwwTagHelper as it searches and replaces http and
    // the AutoLinkerWwwTagHelper adds http to the markup.
    public override int Order
    {
        get  {  return int.MinValue;   }
    }
```

上面的代码可以保证 HTTP 标记帮助程序在 WWW 标记帮助程序之前运行。将 `Order` 更改为 `MaxValue` 并验证为 WWW 标记生成的标记是否不正确。

检查和检索子内容

标记帮助程序提供多个属性来检索内容。

- 可将 `GetChildContentAsync` 的结果追加到 `output.Content`。
- 可使用 `GetContent` 检查 `GetChildContentAsync` 的结果。
- 如果修改 `output.Content`，则不会执行或呈现 TagHelper 主体，除非像自动链接器示例中那样调用 `GetChildContentAsync`：

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent,
             @"\b(?:https?://)(\S+)\b",
              "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http link version}
    }
}
```

- 对 `GetChildContentAsync` 的多次调用返回相同的值，且不重新执行 `TagHelper` 主体，除非传入一个指示不使用缓存结果的 false 参数。

ASP.NET Core 表单中的标记帮助程序

2018/11/10 • • Edit Online

作者：Rick Anderson、Dave Paquette 和 Jerrie Pelser

本文档演示如何使用表单和表单中常用的 HTML 元素。HTML Form 元素提供 Web 应用用于向服务器回发数据的主要机制。本文档的大部分内容介绍标记帮助程序及其如何帮助高效创建可靠的 HTML 表单。建议在阅读本文档前先阅读标记帮助程序简介。

在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。如果存在 HTML 帮助程序替代项，文中会提到。

表单标记帮助程序

表单标记帮助程序：

- 为 MVC 控制器操作或命名路由生成 HTML <FORM> `action` 属性值
- 生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）
- 提供 `asp-route-<Parameter Name>` 属性，其中 `<Parameter Name>` 添加到路由值。`Html.BeginForm` 和 `Html.BeginRouteForm` 的 `routeValues` 参数提供类似的功能。
- 具有 HTML 帮助程序替代项 `Html.BeginForm` 和 `Html.BeginRouteForm`

示例:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

上述表单标记帮助程序生成以下 HTML：

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

MVC 运行时通过表单标记帮助程序属性 `asp-controller` 和 `asp-action` 生成 `action` 属性值。表单标记帮助程序还会生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）。保护纯 HTML 表单免受跨站点请求伪造的影响很难，但表单标记帮助程序可提供此服务。

使用命名路由

`asp-route` 标记帮助程序属性还可为 HTML `action` 属性生成标记。具有名为 `register` 的路由的应用可将以下标记用于注册页：

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Views/Account 文件夹中的许多视图(在新建使用*个人用户帐户*身份验证的 Web 应用时生成)包含 asp-route-returnurl 属性:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **NOTE**
> 使用内置模板时, `returnUrl` 仅会在用户尝试访问授权资源, 但未验证身份或未获得授权的情况下自动填充。如果尝试执行未经授权的访问, 安全中间件会使用 `returnUrl` 集将用户重定向至登录页。

输入标记帮助程序

输入标记帮助程序将 HTML <input> 元素绑定到 Razor 视图中的模型表达式。

语法:

```
<input asp-for="<Expression Name>" />
```

输入标记帮助程序:

- 为 `asp-for` 属性中指定的表达式名称生成 `id` 和 `name` HTML 属性。`asp-for="Property1.Property2"` 与 `m => m.Property1.Property2` 相等。表达式的名称用于 `asp-for` 属性值。有关其他信息, 请参阅表达式名称部分。
- 根据模型类型和应用于模型属性的数据注释特性设置 HTML `type` 特性值
- 如果已经指定, 不会覆盖 HTML `type` 属性值
- 通过应用于模型属性的数据注释特性生成 HTML5 验证特性
- 具有与 `Html.TextBoxFor` 和 `Html.EditorFor` 重叠的 HTML 帮助程序功能。有关详细信息, 请参阅**输入标记帮助程序的 HTML 帮助程序替代**项部分。
- 提供强类型化。如果属性的名称更改, 但未更新标记帮助程序, 则会收到类似如下内容的错误:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
 'RegisterViewModel' does not contain a definition for 'Email' and no
 extension method 'Email' accepting a first argument of type 'RegisterViewModel'
 could be found (are you missing a using directive or an assembly reference?)
```

`Input` 标记帮助程序根据 .NET 类型设置 HTML `type` 属性。下表列出一些常见的 .NET 类型和生成的 HTML 类型(并未列出每个 .NET 类型)。

.NET 类型	输入类型
Bool	type="checkbox"
String	type="text"

.NET 类型	输入类型
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single、Double	type="number"

下表显示输入标记帮助程序会映射到特定输入类型的一些常见数据注释属性（并未列出每个验证属性）：

特性	输入类型
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

上述代码生成以下 HTML：

```
<form method="post" action="/Demo/RegisterInput">
     Email:
     <input type="email" data-val="true"
            data-val-email="The Email Address field is not a valid email address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value="" /> <br>
     Password:
     <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password" /><br>
     <button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
 </form>
```

应用于 `Email` 和 `Password` 属性的数据注释在模型中生成元数据。输入标记帮助程序使用模型元数据并生成 HTML5 `data-val-*` 属性(请参阅模型验证)。这些属性描述要附加到输入字段的验证程序。这样可以提供非介入式 HTML5 和 jQuery 验证。非介入式属性具有格式 `data-val-rule="Error Message"`，其中规则是验证规则的名称(例如 `data-val-required`、`data-val-email`、`data-val-maxlength` 等)。如果在属性中提供错误消息，则该错误消息会作为 `data-val-rule` 属性的值显示。还有表单 `data-val-ruleName-argumentName="argumentValue"` 的属性，这些属性提供有关规则的其他详细信息，例如，`data-val-maxlength-max="1024"`。

输入标记帮助程序的 HTML 帮助程序替代项

`Html.TextBox`、`Html.TextBoxFor`、`Html.Editor` 和 `Html.EditorFor` 与输入标记帮助程序的功能存在重叠。输入标记帮助程序会自动设置 `type` 属性；而 `Html.TextBox` 和 `Html.TextBoxFor` 不会。`Html.Editor` 和 `Html.EditorFor` 处理集合、复杂对象和模板；而输入标记帮助程序不会。输入标记帮助程序、`Html.EditorFor` 和 `Html.TextBoxFor` 是强类型(使用 lambda 表达式)；而 `Html.TextBox` 和 `Html.Editor` 不是(使用表达式名称)。

HtmlAttributes

`@Html.Editor()` 和 `@Html.EditorFor()` 在执行其默认模板时使用名为 `htmlAttributes` 的特殊 `ViewDataDictionary` 条目。此行为可选择使用 `additionalViewData` 参数增强。键"htmlAttributes"区分大小写。键"htmlAttributes"的处理方式与传递到输入帮助程序的 `htmlAttributes` 对象(例如 `@Html.TextBox()`)的处理方式类似。

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

表达式名称

`asp-for` 属性值是 `ModelExpression`，并且是 lambda 表达式的右侧。因此，`asp-for="Property1"` 在生成的代码中变成 `m => m.Property1`，这也是无需使用 `Model` 前缀的原因。"@"字符可用作内联表达式的开头，并移到 `m.` 前面：

```
@{
       var joe = "Joe";
   }
   <input asp-for="@joe" />
```

生成以下 HTML：

```
<input type="text" id="joe" name="joe" value="Joe" />
```

使用集合属性时，`asp-for="CollectionProperty[23].Member"` 在 `i` 具有值 `23` 时生成与 `asp-for="CollectionProperty[i].Member"` 相同的名称。

在 ASP.NET Core MVC 计算 `ModelExpression` 的值时，它会检查多个源，包括 `ModelState`。以 `<input type="text" asp-for="@Name" />` 为例。计算出的 `value` 属性是第一个非 null 值，属于：

- 带有“Name”键的 `ModelState` 条目。
- `Model.Name` 表达式的结果。

导航子属性

还可使用视图模型的属性路径导航到子属性。设想一个包含子 `Address` 属性的更复杂的模型类。

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

在视图中，绑定到 `Address.AddressLine1`：

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

为 `Address.AddressLine1` 生成以下 HTML：

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

表达式名称和集合

包含 `Colors` 数组的模型示例：

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

操作方法：

```
public IActionResult Edit(int id, int colorIndex)
   {
       ViewData["Index"] = colorIndex;
       return View(GetPerson(id));
   }
```

以下 Razor 显示如何访问特定 `Color` 元素：

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

Views/Shared/EditorTemplates/String.cshtml 模板：

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

使用 `List<T>` 的示例：

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

以下 Razor 演示如何循环访问集合：

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

Views/Shared/EditorTemplates/ToDoItem.cshtml 模板：

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

应尽量在 `asp-for` 或 `Html.DisplayFor` 等效上下文中使用值 `foreach`。一般情况下，`for` 优于 `foreach`（如果情况允许使用的话），因为它不需要分配枚举器，但在 LINQ 表达式中评估索引器的成本高昂，应最大限度地减少使用它。

> **NOTE**
> 上述带有注释的示例代码演示如何将 lambda 表达式替换为 `@` 运算符来访问列表中的每个 `ToDoItem`。

文本区标记帮助程序

`Textarea Tag Helper` 标记帮助程序类似于输入标记帮助程序。

- 通过模型为 <textarea> 元素生成 `id` 和 `name` 属性以及数据验证属性。
- 提供强类型化。
- HTML 帮助程序替代项：`Html.TextAreaFor`

示例：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

生成以下 HTML：

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

标签标记帮助程序

- 在 元素中为表达式名称生成标签描述和 `for` 属性
- HTML 帮助程序替代项：`Html.LabelFor`。

`Label Tag Helper` 通过纯 HTML 标签元素提供如下优势：

- 可自动从 `Display` 属性中获取描述性标签值。预期的显示名称可能会随时间变化，`Display` 属性和标签标记帮助程序的组合会在其被使用的所有位置应用 `Display`。
- 源代码中的标记更少
- 模型属性的强类型化。

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

为 `<label>` 元素生成以下 HTML：

```
<label for="Email">Email Address</label>
```

标签标记帮助程序生成“Email”的 `for` 属性值，即与 `<input>` 元素关联的 ID。标记帮助程序生成一致的 `id` 和 `for` 元素，方便将其正确关联。本示例中的描述来自 `Display` 属性。如果模型不包含 `Display` 特性，描述将为表达式的属性名称。

验证标记帮助程序

有两个验证标记帮助程序。`Validation Message Tag Helper`（为模型中的单个属性显示验证消息）和 `Validation Summary Tag Helper`（显示验证错误的摘要）。`Input Tag Helper` 根据模型类的数据注释属性将 HTML5 客户端验证属性添加到输入元素中。同时在服务器上执行验证。验证标记帮助程序会在发生验证错误时显示这些错误消息。

验证消息标记帮助程序

- 将 HTML5 `data-valmsg-for="property"` 属性添加到 span 元素中，该元素会附加指定模型属性的输入字段中的验证错误消息。jQuery 会在发生客户端验证错误时在 `<span>` 元素中显示错误消息。
- 还会在服务器上执行验证。客户端可能已禁用 JavaScript，一些验证仅可在服务器端执行。
- HTML 帮助程序替代项：`Html.ValidationMessageFor`

`Validation Message Tag Helper` 与 HTML span 元素中的 `asp-validation-for` 属性配合使用。

```
<span asp-validation-for="Email"></span>
```

验证消息标记帮助程序会生成以下 HTML：

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

对于同一属性，通常在 `Input` 标记帮助程序后使用 `Validation Message Tag Helper`。这样做可在导致错误的输入附近显示所有验证错误消息。

> **NOTE**
>
> 必须拥有包含正确的 JavaScript 和 jQuery 脚本引用的视图才能执行客户端验证。有关详细信息，请参阅模型验证。

发生服务器端验证错误时（例如，禁用自定义服务器端验证或客户端验证时），MVC 会将该错误消息作为 `<span>` 元素的主体。

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

验证摘要标记帮助程序

- 针对具有 `asp-validation-summary` 属性的 `<div>` 元素
- HTML 帮助程序替代项：`@Html.ValidationSummary`

`Validation Summary Tag Helper` 用于显示验证消息的摘要。`asp-validation-summary` 属性值可以是以下任意值：

ASP-VALIDATION-SUMMARY	显示的验证消息
ValidationSummary.All	属性和模型级别
ValidationSummary.ModelOnly	模型
ValidationSummary.None	无

示例

在以下示例中，数据模型使用 `DataAnnotation` 属性修饰，在 `<input>` 元素中生成验证错误消息。验证标记帮助程序会在发生验证错误时显示错误消息：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

生成的 HTML（如果模型有效）：

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"> <br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

选择标记帮助程序

- 为模型属性生成 select 元素和关联的 option 元素。
- 具有 HTML 帮助程序替代项 `Html.DropDownListFor` 和 `Html.ListBoxFor`

`Select Tag Helper` `asp-for` 为 select 元素指定模型属性名称，`asp-items` 指定 option 元素。例如:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

示例:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

`Index` 方法初始化 `CountryViewModel`，设置选定的国家/地区并将其传递到 `Index` 视图。

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTTP POST `Index` 方法显示选定内容:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country +  " selected";
        return RedirectToAction("IndexSuccess", new { message = msg});
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

`Index` 视图:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

生成以下 HTML(选择"CA"时):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
   </form>
```

> **NOTE**
> 不建议将 `ViewBag` 或 `ViewData` 与选择标记帮助程序配合使用。视图模型在提供 MVC 元数据方面更可靠且通常更不容易出现问题。

`asp-for` 属性值是特殊情况，它不要求提供 `Model` 前缀，但其他标记帮助程序属性需要该前缀(例如 `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

枚举绑定

通常可方便地将 `<select>` 与 `enum` 属性配合使用并通过 `enum` 值生成 `SelectListItem` 元素。

示例:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

`GetEnumSelectList` 方法为枚举生成 `SelectList` 对象。

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

可使用 `Display` 属性修饰枚举器列表，以获取更丰富的 UI：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

生成以下 HTML：

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
  </form>
```

选项组

如果视图模型包含一个或多个 `SelectListGroup` 对象，则会生成 HTML `<optgroup>` 元素。

`CountryViewModelGroup` 将 `SelectListItem` 元素分组为“North America”组和“Europe”组：

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

两个组如下所示:

生成的 HTML：

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

多重选择

如果 `asp-for` 属性中指定的属性为 `IEnumerable`，选择标记帮助程序会自动生成 multiple = "multiple" 属性。例如，如果给定以下模型：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

及以下视图：

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

则会生成以下 HTML：

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

无选定内容

如果发现自己在多个页面中使用“未指定”选项，可创建模板用于消除重复的 HTML：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

Views/Shared/EditorTemplates/CountryViewModel.cshtml 模板：

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

添加 HTML <option> 元素并不局限于*无选定内容*用例。例如，以下视图和操作方法会生成与上述代码类似的 HTML：

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

根据当前的 `Country` 值选择正确的 `<option>` 元素（包含 `selected="selected"` 属性）。

```
<form method="post" action="/Home/IndexEmpty">
     <select id="Country" name="Country">
         <option value="">&lt;none&gt;</option>
         <option value="MX">Mexico</option>
         <option value="CA" selected="selected">Canada</option>
         <option value="US">USA</option>
     </select>
     <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

其他资源

- ASP.NET Core 中的标记帮助程序
- HTML Form 元素
- 请求验证令牌
- ASP.NET Core 中的模型绑定
- ASP.NET Core MVC 中的模型验证
- IAttributeAdapter 接口
- 本文档的代码片段

ASP.NET Core 中的标记帮助程序组件

2018/10/30 • • Edit Online

作者：Scott Addie 和 Fiyaz Bin Hasan

标记帮助程序组件是可用于有条件地修改或添加服务器端代码中的 HTML 元素的标记帮助程序。ASP.NET Core 2.0 或更高版本中提供了此功能。

ASP.NET Core 包括两个内置标记帮助程序组件：`head` 和 `body`。它们位于 Microsoft.AspNetCore.Mvc.Razor.TagHelpers 命名空间中，可用于 MVC 和 Razor Pages。标记帮助程序组件不需要在 *_ViewImports.cshtml* 中注册应用。

查看或下载示例代码（如何下载）

用例

标记帮助程序组件的两个常见用例包括：

1. 将 `<link>` 注入到 `<head>` 中。
2. 将 `<script>` 注入到 `<body>` 中。

以下各节介绍了这些用例。

注入到 HTML head 元素中

在 HTML `<head>` 元素中，通常使用 HTML `<link>` 元素导入 CSS 文件。以下代码使用 `head` 标记帮助程序组件将 `<link>` 元素注入到 `<head>` 元素中：

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace RazorPagesSample.TagHelpers
{
    public class AddressStyleTagHelperComponent : TagHelperComponent
    {
        private readonly string _style =
            @"<link rel=""stylesheet"" href=""/css/address.css"" />";

        public override int Order => 1;

        public override Task ProcessAsync(TagHelperContext context,
                                          TagHelperOutput output)
        {
            if (string.Equals(context.TagName, "head",
                              StringComparison.OrdinalIgnoreCase))
            {
                output.PostContent.AppendHtml(_style);
            }

            return Task.CompletedTask;
        }
    }
}
```

在上述代码中：

- `AddressStyleTagHelperComponent` 可实现 TagHelperComponent。抽象：

- 允许初始化带有 TagHelperContext 的类。
- 启用标记帮助程序组件以添加或修改 HTML 元素。

- Order 属性可定义呈现组件的顺序。应用中的标记帮助程序组件有多种用法时，`Order` 是必需的。
- ProcessAsync 会将执行上下文的 TagName 属性值与 `head` 进行比较。如果比较的计算结果为 true，则 `_style` 字段的内容将注入到 HTML `<head>` 元素中。

注入到 HTML body 元素中

`body` 标记帮助程序组件可以将 `<script>` 元素注入到 `<body>` 元素中。以下代码演示了此项技术：

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace RazorPagesSample.TagHelpers
{
    public class AddressScriptTagHelperComponent : TagHelperComponent
    {
        public override int Order => 2;

        public override async Task ProcessAsync(TagHelperContext context,
                                                TagHelperOutput output)
        {
            if (string.Equals(context.TagName, "body",
                              StringComparison.OrdinalIgnoreCase))
            {
                var script = await File.ReadAllTextAsync(
                    "TagHelpers/Templates/AddressToolTipScript.html");
                output.PostContent.AppendHtml(script);
            }
        }
    }
}
```

单独的 HTML 文件用于存储 `<script>` 元素。HTML 文件使代码更清洁且更易于维护。上述代码可读取 *TagHelpers/Templates/AddressToolTipScript.html* 的内容并将其追加到标记帮助程序输出中。*AddressToolTipScript.html* 文件包括以下标记：

```
<script>
$("address[printable]").hover(function() {
    $(this).attr({
        "data-toggle": "tooltip",
        "data-placement": "right",
        "title": "Home of Microsoft!"
    });
});
</script>
```

上述代码可将启动工具提示小组件绑定到包含 `printable` 属性的任何 `<address>` 元素。当鼠标指针悬停在元素上时，可以显示效果。

注册组件

标记帮助程序组件必须添加到应用的标记帮助程序组件集合。有以下三种方法可添加到集合：

1. 通过服务容器注册
2. 通过 Razor 文件注册
3. 通过页面模型或控制器注册

通过服务容器注册

如果未使用 ITagHelperComponentManager 管理标记帮助程序组件类，则必须向依赖关系注入 (DI) 系统注册。以下 `Startup.ConfigureServices` 代码可注册带有瞬态生存期的 `AddressStyleTagHelperComponent` 和 `AddressScriptTagHelperComponent` 类：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddTransient<ITagHelperComponent,
        AddressScriptTagHelperComponent>();
    services.AddTransient<ITagHelperComponent,
        AddressStyleTagHelperComponent>();
}
```

通过 Razor 文件注册

如果未向 DI 注册标记帮助程序组件，则可以从 Razor Pages 页或 MVC 视图注册。此项技术用于控制注入的标记和 Razor 文件中的组件执行顺序。

`ITagHelperComponentManager` 用于添加标记帮助程序组件或从应用中删除这些组件。以下代码演示了使用 `AddressTagHelperComponent` 的此项技术：

```
@using RazorPagesSample.TagHelpers;
@using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
@inject ITagHelperComponentManager manager;

@{
    string markup;

    if (Model.IsWeekend)
    {
        markup = "<em class='text-warning'>Office closed today!</em>";
    }
    else
    {
        markup = "<em class='text-info'>Office open today!</em>";
    }

    manager.Components.Add(new AddressTagHelperComponent(markup, 1));
}
```

在上述代码中：

- `@inject` 指令提供了 `ITagHelperComponentManager` 的实例。该实例将分配到名为 `manager` 的变量以访问 Razor 文件中的下游。
- `AddressTagHelperComponent` 的实例将添加到应用的标记帮助程序组件集合。

修改 `AddressTagHelperComponent` 以适应接受 `markup` 和 `order` 参数的构造函数：

```
private readonly string _markup;

public override int Order { get; }

public AddressTagHelperComponent(string markup = "", int order = 1)
{
    _markup = markup;
    Order = order;
}
```

提供的 `markup` 参数用于 `ProcessAsync`，如下所示：

```
public override async Task ProcessAsync(TagHelperContext context,
                                        TagHelperOutput output)
{
    if (string.Equals(context.TagName, "address",
            StringComparison.OrdinalIgnoreCase) &&
        output.Attributes.ContainsName("printable"))
    {
        TagHelperContent childContent = await output.GetChildContentAsync();
        string content = childContent.GetContent();
        output.Content.SetHtmlContent(
            $"<div>{content}<br>{_markup}</div>{_printableButton}");
    }
}
```

通过页面模型或控制器注册

如果未向 DI 注册标记帮助程序组件，则可以从 Razor Pages 页面模型或 MVC 控制器注册。此项技术可用于将 C# 逻辑与 Razor 文件隔离。

构造函数注入用于访问 `ITagHelperComponentManager` 的实例。标记帮助程序组件将添加到该实例的标记帮助程序组件集合。以下 Razor Pages 页面模型演示了使用 `AddressTagHelperComponent` 的此项技术：

```
using System;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesSample.TagHelpers;

public class IndexModel : PageModel
{
    private readonly ITagHelperComponentManager _tagHelperComponentManager;

    public bool IsWeekend
    {
        get
        {
            var dayOfWeek = DateTime.Now.DayOfWeek;

            return dayOfWeek == DayOfWeek.Saturday ||
                   dayOfWeek == DayOfWeek.Sunday;
        }
    }

    public IndexModel(ITagHelperComponentManager tagHelperComponentManager)
    {
        _tagHelperComponentManager = tagHelperComponentManager;
    }

    public void OnGet()
    {
        string markup;

        if (IsWeekend)
        {
            markup = "<em class='text-warning'>Office closed today!</em>";
        }
        else
        {
            markup = "<em class='text-info'>Office open today!</em>";
        }

        _tagHelperComponentManager.Components.Add(
            new AddressTagHelperComponent(markup, 1));
    }
}
```

在上述代码中：

- 构造函数注入用于访问 `ITagHelperComponentManager` 的实例。
- `AddressTagHelperComponent` 的实例将添加到应用的标记帮助程序组件集合。

创建组件

创建自定义标记帮助程序组件：

- 创建派生自 TagHelperComponentTagHelper 的公共类。
- 将 [HtmlTargetElement] 属性应用于该类。指定目标 HTML 元素的名称。
- *可选*：将 [EditorBrowsable(EditorBrowsableState.Never)] 属性应用于该类，以禁止在 IntelliSense 中显示类型。

以下代码可创建面向 `<address>` HTML 元素的自定义标记帮助程序组件：

```
using System.ComponentModel;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Microsoft.Extensions.Logging;

namespace RazorPagesSample.TagHelpers
{
    [HtmlTargetElement("address")]
    [EditorBrowsable(EditorBrowsableState.Never)]
    public class AddressTagHelperComponentTagHelper : TagHelperComponentTagHelper
    {
        public AddressTagHelperComponentTagHelper(
            ITagHelperComponentManager componentManager,
            ILoggerFactory loggerFactory) : base(componentManager, loggerFactory)
        {
        }
    }
}
```

按如下所示使用自定义 `address` 标记帮助程序组件注入 HTML 标记：

```
public class AddressTagHelperComponent : TagHelperComponent
{
    private readonly string _printableButton =
        "<button type='button' class='btn btn-info' onclick=\"window.open("
        "'https://binged.it/2AXRRYw')\">" +
        "<span class='glyphicon glyphicon-road' aria-hidden='true'></span>" +
        "</button>";

    public override int Order => 3;

    public override async Task ProcessAsync(TagHelperContext context,
                                            TagHelperOutput output)
    {
        if (string.Equals(context.TagName, "address",
                StringComparison.OrdinalIgnoreCase) &&
            output.Attributes.ContainsName("printable"))
        {
            var content = await output.GetChildContentAsync();
            output.Content.SetHtmlContent(
                $"<div>{content.GetContent()}</div>{_printableButton}");
        }
    }
}
```

上述 `ProcessAsync` 方法可将向 SetHtmlContent 提供的 HTML 注入到匹配的 `<address>` 元素中。在以下情况下进行注入：

- 执行上下文的 `TagName` 属性值等于 `address`。
- 相应的 `<address>` 元素具有 `printable` 属性。

例如，在处理以下 `<address>` 元素时，`if` 语句的计算结果为 true：

```
<address printable>
    One Microsoft Way<br />
    Redmond, WA 98052-6399<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>
```

其他资源

- 在 ASP.NET Core 依赖注入
- 在 ASP.NET Core 中将依赖项注入到视图
- ASP.NET Core 内置标记帮助程序

ASP.NET Core 中的定位点标记帮助程序

2019/1/10 • • Edit Online

作者：Peter Kellner 和 Scott Addie

定位点标记帮助程序可通过添加新属性来增强标准的 HTML 定位点（`<a ... ></a>`）标记。按照约定，属性名称将使用前缀 `asp-`。`asp-` 属性的值决定呈现的定位点元素的 `href` 属性值。

有关标记帮助程序的概述，请参阅 ASP.NET Core 中的标记帮助程序。

查看或下载示例代码（如何下载）

本文档中的示例均使用 SpeakerController：

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;

public class SpeakerController : Controller
{
    private List<Speaker> Speakers =
        new List<Speaker>
        {
            new Speaker {SpeakerId = 10},
            new Speaker {SpeakerId = 11},
            new Speaker {SpeakerId = 12}
        };

    [Route("Speaker/{id:int}")]
    public IActionResult Detail(int id) =>
        View(Speakers.FirstOrDefault(a => a.SpeakerId == id));

    [Route("/Speaker/Evaluations",
           Name = "speakerevals")]
    public IActionResult Evaluations() => View();

    [Route("/Speaker/EvaluationsCurrent",
           Name = "speakerevalscurrent")]
    public IActionResult Evaluations(
        int speakerId,
        bool currentYear) => View();

    public IActionResult Index() => View(Speakers);
}

public class Speaker
{
    public int SpeakerId { get; set; }
}
```

`asp-` 属性的清单如下所示。

asp-controller

asp-controller 属性可分配用于生成 URL 的控制器。下面的标记列出了所有发言人：

```
<a asp-controller="Speaker"
   asp-action="Index">All Speakers</a>
```

生成的 HTML：

```
<a href="/Speaker">All Speakers</a>
```

如果指定了 `asp-controller` 属性，而未指定 `asp-action` 属性，则默认的 `asp-action` 值为与当前正在执行的视图关联的控制器操作。如果前面的标记中省略了 `asp-action`，并在 HomeController 的索引视图 (/Home) 中使用了定位点标记帮助程序，则生成的 HTML 为：

```
<a href="/Home">All Speakers</a>
```

asp-action

asp-action 属性值表示生成的 `href` 属性中包含的控制器操作名称。下面的标记可将生成的 `href` 属性值设置为发言人评估页：

```
<a asp-controller="Speaker"
   asp-action="Evaluations">Speaker Evaluations</a>
```

生成的 HTML：

```
<a href="/Speaker/Evaluations">Speaker Evaluations</a>
```

如果未指定 `asp-controller` 属性，则使用默认控制器，该控制器调用执行当前视图的视图。

如果 `asp-action` 属性值为 `Index`，则不向 URL 追加任何操作，从而导致调用默认的 `Index` 操作。`asp-controller` 引用的控制器中必须存在指定的(或默认的)操作。

asp-route-{value}

asp-route-{value} 属性可实现通配符路由前缀。占用 `{value}` 占位符的所有值都解释为潜在的路由参数。如果找不到默认路由，则将此路由前缀作为请求参数和值追加到生成的 `href` 属性。否则，将在路由模板中替换它。

考虑以下控制器操作：

```
public IActionResult AnchorTagHelper(int id)
{
    var speaker = new Speaker
    {
        SpeakerId = id
    };

    return View(speaker);
}
```

在 Startup.Configure 中定义默认路由模板：

```
app.UseMvc(routes =>
{
    // need route and attribute on controller: [Area("Blogs")]
    routes.MapRoute(name: "mvcAreaRoute",
                    template: "{area:exists}/{controller=Home}/{action=Index}");

    // default route for non-areas
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

MVC 视图使用操作提供的模型，如下所示：

```
@model Speaker
<!DOCTYPE html>
<html>
<body>
    <a asp-controller="Speaker"
       asp-action="Detail"
       asp-route-id="@Model.SpeakerId">SpeakerId: @Model.SpeakerId</a>
</body>
</html>
```

默认路由的 `{id?}` 占位符得以匹配。生成的 HTML：

```
<a href="/Speaker/Detail/12">SpeakerId: 12</a>
```

假设路由前缀不属于匹配路由模板的一部分，如下面的 MVC 视图所示：

```
@model Speaker
<!DOCTYPE html>
<html>
<body>
    <a asp-controller="Speaker"
       asp-action="Detail"
       asp-route-speakerid="@Model.SpeakerId">SpeakerId: @Model.SpeakerId</a>
<body>
</html>
```

生成以下 HTML，因为匹配的路由中未找到 `speakerid`：

```
<a href="/Speaker/Detail?speakerid=12">SpeakerId: 12</a>
```

如果 `asp-controller` 或 `asp-action` 均未指定，则会执行与 `asp-route` 属性中相同的默认处理。

asp-route

asp-route 属性用于创建直接链接到命名路由的 URL。使用路由属性，路由可以按 `SpeakerController` 中所示进行命名并用于其 `Evaluations` 操作：

```
[Route("/Speaker/Evaluations",
       Name = "speakerevals")]
public IActionResult Evaluations() => View();
```

在下列标记中，`asp-route` 属性引用命名路由：

```
<a asp-route="speakerevals">Speaker Evaluations</a>
```

定位点标记帮助程序使用 URL /Speaker/Evaluations 生成直接指向该控制器操作的路由。生成的 HTML：

```
<a href="/Speaker/Evaluations">Speaker Evaluations</a>
```

如果除了 `asp-route`，还指定了 `asp-controller` 或 `asp-action`，则可能不会生成预期的路由。为了避免发生路由冲突，不应将 `asp-route` 与 `asp-controller` 和 `asp-action` 属性结合使用。

asp-all-route-data

asp-all-route-data 属性支持创建键值对字典。键是参数名称，值是参数值。

在下面的示例中，将对字典进行初始化并将其传递给 Razor 视图。或者，也可以使用模型传入数据。

```
@{
var parms = new Dictionary<string, string>
            {
                { "speakerId", "11" },
                { "currentYear", "true" }
            };
}

<a asp-route="speakerevalscurrent"
   asp-all-route-data="parms">Speaker Evaluations</a>
```

前面的代码生成以下 HTML：

```
<a href="/Speaker/EvaluationsCurrent?speakerId=11&currentYear=true">Speaker Evaluations</a>
```

平展 `asp-all-route-data` 字典，以生成满足重载 `Evaluations` 操作要求的查询字符串：

```
[Route("/Speaker/EvaluationsCurrent",
       Name = "speakerevalscurrent")]
public IActionResult Evaluations(
    int speakerId,
    bool currentYear) => View();
```

如果字典中的任何键匹配路由参数，则将根据需要在路由中替换这些值。其他不匹配的值作为请求参数生成。

asp-fragment

asp-fragment 属性可定义要追加到 URL 的 URL 片段。定位点标记帮助程序添加哈希字符 (#)。请考虑以下标记：

```
<a asp-controller="Speaker"
   asp-action="Evaluations"
   asp-fragment="SpeakerEvaluations">Speaker Evaluations</a>
```

生成的 HTML：

```
<a href="/Speaker/Evaluations#SpeakerEvaluations">Speaker Evaluations</a>
```

生成客户端应用时，哈希标记很有用。它们可用于在 JavaScript 中轻松地执行标记和搜索等操作。

asp-area

asp-area 属性可设置用来设置相应路由的区域名称。以下示例展示了 `asp-area` 属性如何导致重新映射路由。

Razor Pages 中的使用情况

ASP.NET Core 2.1 或更高版本中支持 Razor Pages 区域。

考虑以下目录层次结构：

- **{项目名称}**
  - **wwwroot**
  - **区域**
    - **会**话
      - 页
        - ViewStart.cshtml*_
        - Index.cshtml
        - Index.cshtml.cs
  - 页

用于引用"会话"区域"索引"Razor 页的标记是：

```
<a asp-area="Sessions"
   asp-page="/Index">View Sessions</a>
```

生成的 HTML：

```
<a href="/Sessions">View Sessions</a>
```

TIP

若要支持 Razor Pages 应用中的区域，请执行以下 `Startup.ConfigureServices` 之一：

- 将兼容性版本设置为 2.1 或更高版本。
- 将 RazorPagesOptions.AllowAreas 属性设置为 `true`：

```
services.AddMvc()
        .AddRazorPagesOptions(options => options.AllowAreas = true);
```

MVC 中的使用情况

考虑以下目录层次结构：

- **{项目名称}**
  - **wwwroot**
  - **区域**
    - **博客**
      - **控制器**
        - HomeController.cs
      - 视图
        - **主文件**夹

        - *AboutBlog.cshtml*
        - Index.cshtml
    - ViewStart.cshtml*_
- **控制器**

如果将 `asp-area` 设置为 "Blogs"，则会为此定位点标记的关联控制器和视图的路由添加目录 Areas/Blogs 作为前缀。用于引用 AboutBlog 视图的标记是：

```
<a asp-area="Blogs"
   asp-controller="Home"
   asp-action="AboutBlog">About Blog</a>
```

生成的 HTML：

```
<a href="/Blogs/Home/AboutBlog">About Blog</a>
```

TIP

若要在 MVC 应用中支持区域，路由模板必须包含对该区域（如果存在）的引用。该模板由 Startup.Configure 中的 `routes.MapRoute` 方法调用的第二个参数表示：

```
app.UseMvc(routes =>
{
    // need route and attribute on controller: [Area("Blogs")]
    routes.MapRoute(name: "mvcAreaRoute",
                    template: "{area:exists}/{controller=Home}/{action=Index}");

    // default route for non-areas
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

asp-protocol

asp-protocol 属性用于在 URL 中指定协议（比如 `https`）。例如:

```
<a asp-protocol="https"
   asp-controller="Home"
   asp-action="About">About</a>
```

生成的 HTML：

```
<a href="https://localhost/Home/About">About</a>
```

示例中的主机名为 localhost。生成 URL 时，定位点标记帮助程序会使用网站的公共域。

asp-host

asp-host 属性用于在 URL 中指定主机名。例如:

```
<a asp-protocol="https"
   asp-host="microsoft.com"
   asp-controller="Home"
   asp-action="About">About</a>
```

生成的 HTML：

```
<a href="https://microsoft.com/Home/About">About</a>
```

asp-page

asp-page 属性适用于 Razor 页面。使用它向特定页设置定位点标记的 `href` 属性值。通过在页面名称前面使用正斜杠 ("/") 作为前缀，可创建 URL。

下列示例指向与会者 Razor 页面：

```
<a asp-page="/Attendee">All Attendees</a>
```

生成的 HTML：

```
<a href="/Attendee">All Attendees</a>
```

`asp-page` 属性与 `asp-route`、`asp-controller` 和 `asp-action` 属性互斥。但是，`asp-page` 可与 `asp-route-{value}` 结合使用以控制路由，如以下标记所示：

```
<a asp-page="/Attendee"
   asp-route-attendeeid="10">View Attendee</a>
```

生成的 HTML：

```
<a href="/Attendee?attendeeid=10">View Attendee</a>
```

asp-page-handler

asp-page-handler 属性适用于 Razor 页面。它用于链接到特定的页处理程序。

请考虑以下页处理程序：

```
public void OnGetProfile(int attendeeId)
{
    ViewData["AttendeeId"] = attendeeId;

    // code omitted for brevity
}
```

页模型的关联标记链接到 `OnGetProfile` 页处理程序。注意，`asp-page-handler` 属性值中省略了页处理程序方法名称的 `On<Verb>` 前缀。如果方法是异步的，也省略 `Async` 后缀。

```
<a asp-page="/Attendee"
   asp-page-handler="Profile"
   asp-route-attendeeid="12">Attendee Profile</a>
```

生成的 HTML：

```
<a href="/Attendee?attendeeid=12&handler=Profile">Attendee Profile</a>
```

其他资源

- ASP.NET Core 中的区域
- ASP.NET Core 中的 Razor 页面介绍
- ASP.NET Core MVC 的兼容性版本

ASP.NET Core MVC 中的缓存标记帮助程序

2018/11/10 • • Edit Online

作者：Peter Kellner 和 Luke Latham

缓存标记帮助程序通过将其内容缓存到内部 ASP.NET Core 缓存提供程序中，极大地提高了 ASP.NET Core 应用的性能。

有关标记帮助程序的概述，请参阅 ASP.NET Core 中的标记帮助程序。

以下 Razor 标记缓存当前日期：

```
<cache>@DateTime.Now</cache>
```

对包含标记帮助程序的页面的第一个请求显示当前日期。其他请求将显示已缓存的值，直到缓存过期（默认 20 分钟）或因内存压力而逐出。

缓存标记帮助程序属性

enabled

属性类型	示例	默认
Boolean	`true`，`false`	`true`

`enabled` 确定是否缓存了缓存标记帮助程序所包含的内容。默认值为 `true`。如果设置为 `false`，则不会缓存呈现的输出。

示例：

```
<cache enabled="true">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-on

属性类型	示例
`DateTimeOffset`	`@new DateTime(2025,1,29,17,02,0)`

`expires-on` 为缓存项设置一个绝对到期日期。

以下示例将在 2025 年 1 月 29 日下午 5:02 之前缓存缓存标记帮助程序的内容：

```
<cache expires-on="@new DateTime(2025,1,29,17,02,0)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-after

属性类型	示例	默认
`TimeSpan`	`@TimeSpan.FromSeconds(120)`	20 分钟

`expires-after` 设置从第一个请求时间到缓存内容的时间长度。

示例:

```
<cache expires-after="@TimeSpan.FromSeconds(120)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

Razor 视图引擎将默认的 `expires-after` 值设置为 20 分钟。

expires-sliding

属性类型	示例
`TimeSpan`	`@TimeSpan.FromSeconds(60)`

设置某个缓存项的值未被访问时，该缓存项应被逐出的时间。

示例:

```
<cache expires-sliding="@TimeSpan.FromSeconds(60)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-header

属性类型	示例
String	`User-Agent`, `User-Agent,content-encoding`

`vary-by-header` 接受逗号分隔的标头值列表，在标头值发生更改时触发缓存刷新。

以下示例监视标头值 `User-Agent`。该示例将缓存提供给 Web 服务器的每个不同 `User-Agent` 的内容:

```
<cache vary-by-header="User-Agent">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-query

属性类型	示例
String	`Make`, `Make,Model`

`vary-by-query` 接受查询字符串(Query) 中逗号分隔的 Keys 列表，它们在任何列出的键值发生更改时触发缓存刷新。

以下示例监视 `Make` 和 `Model` 值。该示例将缓存提供给 Web 服务器的每个不同 `Make` 和 `Model` 的内容:

```
<cache vary-by-query="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-route

属性类型	示例
String	`Make`, `Make,Model`

`vary-by-route` 接受路由参数名称的逗号分隔列表，用于在路由数据参数值发生更改时触发缓存刷新。

示例:

Startup.cs:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{Make?}/{Model?}");
```

Index.cshtml:

```
<cache vary-by-route="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-cookie

属性类型	示例
String	`.AspNetCore.Identity.Application`, `.AspNetCore.Identity.Application,HairColor`

`vary-by-cookie` 接受 Cookie 名称的逗号分隔列表，用于在 Cookie 值发生更改时触发缓存刷新。

下例监视与 ASP.NET Core 标识相关联的 cookie。当用户进行身份验证时，标识 cookie 中的更改就会触发缓存刷新:

```
<cache vary-by-cookie=".AspNetCore.Identity.Application">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-user

属性类型	示例	默认
Boolean	`true`, `false`	`true`

`vary-by-user` 指定当已登录用户(或上下文主体)发生更改时是否应重置缓存。当前用户也称为请求上下文主体，可通过引用 `@User.Identity.Name` 在 Razor 视图中查看。

下面的示例监视当前登录的用户触发缓存刷新:

```
<cache vary-by-user="true">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

通过登录和注销周期，使用此属性将内容维护在缓存中。当值设置为 `true` 时，身份验证周期会使已经过身份验证的用户的缓存失效。缓存无效是因为用户进行身份验证时生成了一个新的唯一 cookie 值。如果 cookie 不存在或已过期，则会维持缓存以呈现匿名状态。如果用户未经过身份验证，则会维持缓存。

vary-by

属性类型	示例
String	`@Model`

`vary-by` 允许自定义缓存的数据。当属性的字符串值引用的对象发生更改时，会更新缓存标记帮助程序的内容。通常将模型值的字符串串联分配给此属性。从效果上看，这导致了更新任何已连接的值都会使缓存无效。

以下示例假定视图的控制器方法将两个路由参数 `myParam1` 和 `myParam2` 的整数值相加，并将其作为单个模型属性返回。当此总和更改时，会再次呈现并缓存缓存标记帮助程序的内容。

操作：

```
public IActionResult Index(string myParam1, string myParam2, string myParam3)
{
    int num1;
    int num2;
    int.TryParse(myParam1, out num1);
    int.TryParse(myParam2, out num2);
    return View(viewName, num1 + num2);
}
```

Index.cshtml：

```
<cache vary-by="@Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

priority

属性类型	示例	默认
`CacheItemPriority`	`High`，`Low`，`NeverRemove`，`Normal`	`Normal`

`priority` 为内置缓存提供程序提供缓存逐出指导。在内存压力下，Web 服务器将首先逐出 `Low` 缓存项。

示例:

```
<cache priority="High">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

`priority` 属性并不能保证特定级别的缓存保留。`CacheItemPriority` 仅供参考。将此属性设置为 `NeverRemove` 并不能保证缓存项将始终保留。请参阅其他资源部分中的相关主题，以获取详细信息。

缓存标记帮助程序依赖于内存缓存服务。如果尚未添加该服务，缓存标记帮助程序将为你添加。

其他资源

- 缓存在内存中 ASP.NET Core
- ASP.NET Core 上的标识简介

ASP.NET Core 中的分布式缓存标记帮助程序

2018/10/29 • • Edit Online

作者：Peter Kellner 和 Luke Latham

分布式缓存标记帮助程序将其内容缓存到分布式缓存源，从而大幅提高 ASP.NET Core 应用的性能。

有关标记帮助程序的概述，请参阅 ASP.NET Core 中的标记帮助程序。

分布式缓存标记帮助程序与缓存标记帮助程序继承自相同的基类。分布式标记帮助程序可以使用所有缓存标记帮助程序属性。

分布式缓存标记帮助程序使用构造函数注入。IDistributedCache 接口将传递到分布式缓存标记帮助程序的构造函数中。如果在 `Startup.ConfigureServices`（Startup.cs）中未创建 `IDistributedCache` 的具体实现，则分布式缓存标记帮助程序会使用与缓存标记帮助程序用于存储缓存数据相同的内存中提供程序。

分布式缓存标记帮助程序属性

与缓存标记帮助程序共享的属性

- `enabled`
- `expires-on`
- `expires-after`
- `expires-sliding`
- `vary-by-header`
- `vary-by-query`
- `vary-by-route`
- `vary-by-cookie`
- `vary-by-user`
- `vary-by priority`

分布式缓存标记帮助程序继承自与缓存标记帮助程序相同的类。有关这些属性的说明，请参阅缓存标记帮助程序。

name

属性类型	示例
String	`my-distributed-cache-unique-key-101`

`name` 是必需的。`name` 属性用作每个存储的缓存实例的键。分布式缓存标记帮助程序分配缓存键时只以属性 `name` 上的键为基础，这点与缓存标记帮助程序不同，后者基于 Razor 页面中的 Razor 页面名称和位置为每个实例分配缓存键。

示例：

```
<distributed-cache name="my-distributed-cache-unique-key-101">
    Time Inside Cache Tag Helper: @DateTime.Now
</distributed-cache>
```

分布式缓存标记帮助程序 IDistributedCache 实现

ASP.NET Core 中内置了 IDistributedCache 的两个实现。一个是基于 SQL Server，另一个是基于 Redis。有关这些实现的详细信息，请参阅 分布式缓存在 ASP.NET Core 中。这两种实现都需要在 `Startup` 中设置 `IDistributedCache` 的实例。

没有专门与使用 `IDistributedCache` 的任何特定实现相关的标记属性。

其他资源

- ASP.NET Core MVC 中的缓存标记帮助程序
- 在 ASP.NET Core 依赖注入
- 分布式缓存在 ASP.NET Core 中
- 缓存在内存中 ASP.NET Core
- ASP.NET Core 上的标识简介

ASP.NET Core 中的环境标记帮助程序

2018/10/29 • • Edit Online

作者：Peter Kellner、Hisham Bin Ateya 和 Luke Latham

环境标记帮助程序根据当前宿主环境，有条件地呈现其包含的内容。环境标记帮助程序的单个属性 `names` 是以逗号分隔的环境名称列表。任何提供的环境名称与当前环境匹配时，都会呈现包含的内容。

有关标记帮助程序的概述，请参阅 ASP.NET Core 中的标记帮助程序。

环境标记帮助程序属性

名称

`names` 采用单个宿主环境名称或以逗号分隔的宿主环境名称列表，用于触发已包含内容的呈现。

将环境值与 IHostingEnvironment.EnvironmentName 返回的当前值进行比较。比较不区分大小写。

下面的示例使用图像标记帮助程序。如果宿主环境是暂存或生产，则呈现内容：

```
<environment names="Staging,Production">
    <strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>
</environment>
```

include 和 exclude 属性

`include` & `exclude` 属性基于已包括或已排除的宿主环境名称，控制已包含内容的呈现。

include

`include` 属性表现出与 `names` 属性相似的行为。`include` 属性值中列出的环境必须与应用程序的托管环境 (IHostingEnvironment.EnvironmentName) 匹配才能呈现 `<environment>` 标记的内容。

```
<environment include="Staging,Production">
    <strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>
</environment>
```

exclude

与 `include` 属性相反，当托管环境与 `exclude` 属性值中列出的环境不匹配时，将呈现 `<environment>` 标记的内容。

```
<environment exclude="Development">
    <strong>HostingEnvironment.EnvironmentName is not Development</strong>
</environment>
```

其他资源

- 在 ASP.NET Core 中使用多个环境

ASP.NET Core 表单中的标记帮助程序

2018/11/10 • • Edit Online

作者：Rick Anderson、Dave Paquette 和 Jerrie Pelser

本文档演示如何使用表单和表单中常用的 HTML 元素。HTML Form 元素提供 Web 应用用于向服务器回发数据的主要机制。本文档的大部分内容介绍标记帮助程序及其如何帮助高效创建可靠的 HTML 表单。建议在阅读本文档前先阅读标记帮助程序简介。

在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。如果存在 HTML 帮助程序替代项，文中会提到。

表单标记帮助程序

表单标记帮助程序：

- 为 MVC 控制器操作或命名路由生成 HTML <FORM> `action` 属性值
- 生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）
- 提供 `asp-route-<Parameter Name>` 属性，其中 `<Parameter Name>` 添加到路由值。`Html.BeginForm` 和 `Html.BeginRouteForm` 的 `routeValues` 参数提供类似的功能。
- 具有 HTML 帮助程序替代项 `Html.BeginForm` 和 `Html.BeginRouteForm`

示例:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

上述表单标记帮助程序生成以下 HTML：

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

MVC 运行时通过表单标记帮助程序属性 `asp-controller` 和 `asp-action` 生成 `action` 属性值。表单标记帮助程序还会生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）。保护纯 HTML 表单免受跨站点请求伪造的影响很难，但表单标记帮助程序可提供此服务。

使用命名路由

`asp-route` 标记帮助程序属性还可为 HTML `action` 属性生成标记。具有名为 `register` 的路由的应用可将以下标记用于注册页：

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Views/Account 文件夹中的许多视图(在新建使用*个人用户帐户*身份验证的 Web 应用时生成)包含 asp-route-returnurl 属性:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **NOTE**
> 使用内置模板时, `returnUrl` 仅会在用户尝试访问授权资源, 但未验证身份或未获得授权的情况下自动填充。如果尝试执行未经授权的访问, 安全中间件会使用 `returnUrl` 集将用户重定向至登录页。

输入标记帮助程序

输入标记帮助程序将 HTML <input> 元素绑定到 Razor 视图中的模型表达式。

语法:

```
<input asp-for="<Expression Name>" />
```

输入标记帮助程序:

- 为 `asp-for` 属性中指定的表达式名称生成 `id` 和 `name` HTML 属性。`asp-for="Property1.Property2"` 与 `m => m.Property1.Property2` 相等。表达式的名称用于 `asp-for` 属性值。有关其他信息, 请参阅表达式名称部分。
- 根据模型类型和应用于模型属性的数据注释特性设置 HTML `type` 特性值
- 如果已经指定, 不会覆盖 HTML `type` 属性值
- 通过应用于模型属性的数据注释特性生成 HTML5 验证特性
- 具有与 `Html.TextBoxFor` 和 `Html.EditorFor` 重叠的 HTML 帮助程序功能。有关详细信息, 请参阅**输入标记帮助程序的 HTML 帮助程序替代**项部分。
- 提供强类型化。如果属性的名称更改, 但未更新标记帮助程序, 则会收到类似如下内容的错误:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
 'RegisterViewModel' does not contain a definition for 'Email' and no
 extension method 'Email' accepting a first argument of type 'RegisterViewModel'
 could be found (are you missing a using directive or an assembly reference?)
```

`Input` 标记帮助程序根据 .NET 类型设置 HTML `type` 属性。下表列出一些常见的 .NET 类型和生成的 HTML 类型(并未列出每个 .NET 类型)。

.NET 类型	输入类型
Bool	type="checkbox"
String	type="text"

.NET 类型	输入类型
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single、Double	type="number"

下表显示输入标记帮助程序会映射到特定输入类型的一些常见数据注释属性（并未列出每个验证属性）：

特性	输入类型
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

上述代码生成以下 HTML：

```
<form method="post" action="/Demo/RegisterInput">
     Email:
     <input type="email" data-val="true"
            data-val-email="The Email Address field is not a valid email address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value="" /> <br>
     Password:
     <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password" /><br>
     <button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
 </form>
```

应用于 `Email` 和 `Password` 属性的数据注释在模型中生成元数据。输入标记帮助程序使用模型元数据并生成 HTML5 `data-val-*` 属性(请参阅模型验证)。这些属性描述要附加到输入字段的验证程序。这样可以提供非介入式 HTML5 和 jQuery 验证。非介入式属性具有格式 `data-val-rule="Error Message"`，其中规则是验证规则的名称(例如 `data-val-required`、`data-val-email`、`data-val-maxlength` 等)。如果在属性中提供错误消息，则该错误消息会作为 `data-val-rule` 属性的值显示。还有表单 `data-val-ruleName-argumentName="argumentValue"` 的属性，这些属性提供有关规则的其他详细信息，例如，`data-val-maxlength-max="1024"`。

输入标记帮助程序的 HTML 帮助程序替代项

`Html.TextBox`、`Html.TextBoxFor`、`Html.Editor` 和 `Html.EditorFor` 与输入标记帮助程序的功能存在重叠。输入标记帮助程序会自动设置 `type` 属性；而 `Html.TextBox` 和 `Html.TextBoxFor` 不会。`Html.Editor` 和 `Html.EditorFor` 处理集合、复杂对象和模板；而输入标记帮助程序不会。输入标记帮助程序、`Html.EditorFor` 和 `Html.TextBoxFor` 是强类型(使用 lambda 表达式)；而 `Html.TextBox` 和 `Html.Editor` 不是(使用表达式名称)。

HtmlAttributes

`@Html.Editor()` 和 `@Html.EditorFor()` 在执行其默认模板时使用名为 `htmlAttributes` 的特殊 `ViewDataDictionary` 条目。此行为可选择使用 `additionalViewData` 参数增强。键"htmlAttributes"区分大小写。键"htmlAttributes"的处理方式与传递到输入帮助程序的 `htmlAttributes` 对象(例如 `@Html.TextBox()`)的处理方式类似。

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

表达式名称

`asp-for` 属性值是 `ModelExpression`，并且是 lambda 表达式的右侧。因此，`asp-for="Property1"` 在生成的代码中变成 `m => m.Property1`，这也是无需使用 `Model` 前缀的原因。"@"字符可用作内联表达式的开头，并移到 `m.` 前面：

```
@{
       var joe = "Joe";
   }
   <input asp-for="@joe" />
```

生成以下 HTML：

```
<input type="text" id="joe" name="joe" value="Joe" />
```

使用集合属性时，`asp-for="CollectionProperty[23].Member"` 在 `i` 具有值 `23` 时生成与 `asp-for="CollectionProperty[i].Member"` 相同的名称。

在 ASP.NET Core MVC 计算 `ModelExpression` 的值时，它会检查多个源，包括 `ModelState`。以 `<input type="text" asp-for="@Name" />` 为例。计算出的 `value` 属性是第一个非 null 值，属于：

- 带有“Name”键的 `ModelState` 条目。
- `Model.Name` 表达式的结果。

导航子属性

还可使用视图模型的属性路径导航到子属性。设想一个包含子 `Address` 属性的更复杂的模型类。

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

在视图中，绑定到 `Address.AddressLine1`：

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

为 `Address.AddressLine1` 生成以下 HTML：

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

表达式名称和集合

包含 `Colors` 数组的模型示例：

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

操作方法：

```
public IActionResult Edit(int id, int colorIndex)
   {
       ViewData["Index"] = colorIndex;
       return View(GetPerson(id));
   }
```

以下 Razor 显示如何访问特定 `Color` 元素：

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

Views/Shared/EditorTemplates/String.cshtml 模板：

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

使用 `List<T>` 的示例：

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

以下 Razor 演示如何循环访问集合：

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

Views/Shared/EditorTemplates/ToDoItem.cshtml 模板：

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

应尽量在 `asp-for` 或 `Html.DisplayFor` 等效上下文中使用值 `foreach`。一般情况下，`for` 优于 `foreach`（如果情况允许使用的话），因为它不需要分配枚举器，但在 LINQ 表达式中评估索引器的成本高昂，应最大限度地减少使用它。

> **NOTE**
> 上述带有注释的示例代码演示如何将 lambda 表达式替换为 `@` 运算符来访问列表中的每个 `ToDoItem`。

文本区标记帮助程序

`Textarea Tag Helper` 标记帮助程序类似于输入标记帮助程序。

- 通过模型为 <textarea> 元素生成 `id` 和 `name` 属性以及数据验证属性。
- 提供强类型化。
- HTML 帮助程序替代项：`Html.TextAreaFor`

示例：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

生成以下 HTML：

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

标签标记帮助程序

- 在 元素中为表达式名称生成标签描述和 `for` 属性
- HTML 帮助程序替代项：`Html.LabelFor`。

`Label Tag Helper` 通过纯 HTML 标签元素提供如下优势：

- 可自动从 `Display` 属性中获取描述性标签值。预期的显示名称可能会随时间变化，`Display` 属性和标签标记帮助程序的组合会在其被使用的所有位置应用 `Display`。
- 源代码中的标记更少
- 模型属性的强类型化。

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

为 `<label>` 元素生成以下 HTML：

```
<label for="Email">Email Address</label>
```

标签标记帮助程序生成“Email”的 `for` 属性值，即与 `<input>` 元素关联的 ID。标记帮助程序生成一致的 `id` 和 `for` 元素，方便将其正确关联。本示例中的描述来自 `Display` 属性。如果模型不包含 `Display` 特性，描述将为表达式的属性名称。

验证标记帮助程序

有两个验证标记帮助程序。`Validation Message Tag Helper`（为模型中的单个属性显示验证消息）和 `Validation Summary Tag Helper`（显示验证错误的摘要）。`Input Tag Helper` 根据模型类的数据注释属性将 HTML5 客户端验证属性添加到输入元素中。同时在服务器上执行验证。验证标记帮助程序会在发生验证错误时显示这些错误消息。

验证消息标记帮助程序

- 将 HTML5 `data-valmsg-for="property"` 属性添加到 span 元素中，该元素会附加指定模型属性的输入字段中的验证错误消息。jQuery 会在发生客户端验证错误时在 `<span>` 元素中显示错误消息。
- 还会在服务器上执行验证。客户端可能已禁用 JavaScript，一些验证仅可在服务器端执行。
- HTML 帮助程序替代项：`Html.ValidationMessageFor`

`Validation Message Tag Helper` 与 HTML span 元素中的 `asp-validation-for` 属性配合使用。

```
<span asp-validation-for="Email"></span>
```

验证消息标记帮助程序会生成以下 HTML：

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

对于同一属性，通常在 `Input` 标记帮助程序后使用 `Validation Message Tag Helper`。这样做可在导致错误的输入附近显示所有验证错误消息。

> **NOTE**
>
> 必须拥有包含正确的 JavaScript 和 jQuery 脚本引用的视图才能执行客户端验证。有关详细信息，请参阅模型验证。

发生服务器端验证错误时（例如，禁用自定义服务器端验证或客户端验证时），MVC 会将该错误消息作为 `<span>` 元素的主体。

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

验证摘要标记帮助程序

- 针对具有 `asp-validation-summary` 属性的 `<div>` 元素
- HTML 帮助程序替代项：`@Html.ValidationSummary`

`Validation Summary Tag Helper` 用于显示验证消息的摘要。`asp-validation-summary` 属性值可以是以下任意值：

ASP-VALIDATION-SUMMARY	显示的验证消息
ValidationSummary.All	属性和模型级别
ValidationSummary.ModelOnly	模型
ValidationSummary.None	无

示例

在以下示例中，数据模型使用 `DataAnnotation` 属性修饰，在 `<input>` 元素中生成验证错误消息。验证标记帮助程序会在发生验证错误时显示错误消息：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

生成的 HTML（如果模型有效）：

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"> <br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

选择标记帮助程序

- 为模型属性生成 select 元素和关联的 option 元素。
- 具有 HTML 帮助程序替代项 `Html.DropDownListFor` 和 `Html.ListBoxFor`

`Select Tag Helper` `asp-for` 为 select 元素指定模型属性名称，`asp-items` 指定 option 元素。例如:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

示例:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

`Index` 方法初始化 `CountryViewModel`，设置选定的国家/地区并将其传递到 `Index` 视图。

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTTP POST `Index` 方法显示选定内容:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country +  " selected";
        return RedirectToAction("IndexSuccess", new { message = msg});
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

`Index` 视图:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

生成以下 HTML(选择"CA"时):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
   </form>
```

> **NOTE**
> 不建议将 `ViewBag` 或 `ViewData` 与选择标记帮助程序配合使用。视图模型在提供 MVC 元数据方面更可靠且通常更不容易出现问题。

`asp-for` 属性值是特殊情况，它不要求提供 `Model` 前缀，但其他标记帮助程序属性需要该前缀(例如 `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

枚举绑定

通常可方便地将 `<select>` 与 `enum` 属性配合使用并通过 `enum` 值生成 `SelectListItem` 元素。

示例:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

`GetEnumSelectList` 方法为枚举生成 `SelectList` 对象。

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

可使用 `Display` 属性修饰枚举器列表，以获取更丰富的 UI：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

生成以下 HTML：

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
  </form>
```

选项组

如果视图模型包含一个或多个 `SelectListGroup` 对象，则会生成 HTML `<optgroup>` 元素。

`CountryViewModelGroup` 将 `SelectListItem` 元素分组为“North America”组和“Europe”组：

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

两个组如下所示:

生成的 HTML：

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

多重选择

如果 `asp-for` 属性中指定的属性为 `IEnumerable`，选择标记帮助程序会自动生成 multiple = "multiple" 属性。例如，如果给定以下模型：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

及以下视图：

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

则会生成以下 HTML：

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

无选定内容

如果发现自己在多个页面中使用“未指定”选项，可创建模板用于消除重复的 HTML：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

Views/Shared/EditorTemplates/CountryViewModel.cshtml 模板：

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

添加 HTML <option> 元素并不局限于*无选定内容*用例。例如，以下视图和操作方法会生成与上述代码类似的 HTML：

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

根据当前的 `Country` 值选择正确的 `<option>` 元素（包含 `selected="selected"` 属性）。

```
<form method="post" action="/Home/IndexEmpty">
     <select id="Country" name="Country">
         <option value="">&lt;none&gt;</option>
         <option value="MX">Mexico</option>
         <option value="CA" selected="selected">Canada</option>
         <option value="US">USA</option>
     </select>
     <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

其他资源

- ASP.NET Core 中的标记帮助程序
- HTML Form 元素
- 请求验证令牌
- ASP.NET Core 中的模型绑定
- ASP.NET Core MVC 中的模型验证
- IAttributeAdapter 接口
- 本文档的代码片段

ASP.NET Core 中的图像标记帮助程序

2018/10/29 • • Edit Online

作者：Peter Kellner

图像标记帮助程序增强了 `<img>` 标记，为静态图像文件提供缓存破坏行为。

缓存破坏字符串是一个唯一值，表示附加到资产 URL 的静态图像文件的哈希值。唯一字符串会提示客户端（和某些代理）从主机 Web 服务器重新加载图像，而不是从客户端的缓存重新加载。

如果图像源（`src`）是主机 web 服务器上的静态文件：

- 将唯一缓存破坏字符串作为图像源的查询参数进行追加。
- 如果主机 Web 服务器上的文件发生更改，将生成包含已更新请求参数的唯一请求 URL。

有关标记帮助程序的概述，请参阅 ASP.NET Core 中的标记帮助程序。

图像标记帮助程序属性

src

若要激活图像标记帮助程序，`<img>` 元素需要有 `src` 属性。

图像源（`src`）必须指向服务器上的物理静态文件。如果 `src` 是一个远程 URI，则不会生成缓存破坏查询字符串参数。

asp-append-version

如果使用 `true` 值和 `src` 属性指定 `asp-append-version`，则会调用图像标记帮助程序。

下面的示例使用图像标记帮助程序：

```
<img src="~/images/asplogo.png" asp-append-version="true" />
```

如果目录 /wwwroot/images/ 中存在静态文件，则生成的 html 与下面类似（哈希有所不同）：

```
<img src="/images/asplogo.png?v=Kl_dqr9NVtnMdsM2MUg4qthUnWZm5T1fCEimBPWDNgM" />
```

分配给参数 `v` 的值是磁盘上的 asplogo.png 文件的哈希值。如果 Web 服务器无法获取对静态文件的读取访问权限，则不会向呈现在标记中的 `src` 属性添加 `v` 参数。

哈希缓存行为

图像标记帮助程序使用本地 Web 服务器上的缓存提供程序来存储给定文件的已计算 `Sha512` 哈希。如果多次请求文件，则不重新计算哈希值。当计算该文件的 `Sha512` 哈希时，附加到该文件的文件观察程序会让 Cache 失效。当磁盘上的的文件发生更改时，将会计算和缓存新的哈希。

其他资源

- 缓存在内存中 ASP.NET Core

ASP.NET Core 表单中的标记帮助程序

2018/11/10 • • Edit Online

作者：Rick Anderson、Dave Paquette 和 Jerrie Pelser

本文档演示如何使用表单和表单中常用的 HTML 元素。HTML Form 元素提供 Web 应用用于向服务器回发数据的主要机制。本文档的大部分内容介绍标记帮助程序及其如何帮助高效创建可靠的 HTML 表单。建议在阅读本文档前先阅读标记帮助程序简介。

在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。如果存在 HTML 帮助程序替代项，文中会提到。

表单标记帮助程序

表单标记帮助程序：

- 为 MVC 控制器操作或命名路由生成 HTML <FORM> `action` 属性值
- 生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）
- 提供 `asp-route-<Parameter Name>` 属性，其中 `<Parameter Name>` 添加到路由值。`Html.BeginForm` 和 `Html.BeginRouteForm` 的 `routeValues` 参数提供类似的功能。
- 具有 HTML 帮助程序替代项 `Html.BeginForm` 和 `Html.BeginRouteForm`

示例:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

上述表单标记帮助程序生成以下 HTML：

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

MVC 运行时通过表单标记帮助程序属性 `asp-controller` 和 `asp-action` 生成 `action` 属性值。表单标记帮助程序还会生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）。保护纯 HTML 表单免受跨站点请求伪造的影响很难，但表单标记帮助程序可提供此服务。

使用命名路由

`asp-route` 标记帮助程序属性还可为 HTML `action` 属性生成标记。具有名为 `register` 的路由的应用可将以下标记用于注册页：

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Views/Account 文件夹中的许多视图(在新建使用*个人用户帐户*身份验证的 Web 应用时生成)包含 asp-route-returnurl 属性:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **NOTE**
> 使用内置模板时, `returnUrl` 仅会在用户尝试访问授权资源, 但未验证身份或未获得授权的情况下自动填充。如果尝试执行未经授权的访问, 安全中间件会使用 `returnUrl` 集将用户重定向至登录页。

输入标记帮助程序

输入标记帮助程序将 HTML <input> 元素绑定到 Razor 视图中的模型表达式。

语法:

```
<input asp-for="<Expression Name>" />
```

输入标记帮助程序:

- 为 `asp-for` 属性中指定的表达式名称生成 `id` 和 `name` HTML 属性。`asp-for="Property1.Property2"` 与 `m => m.Property1.Property2` 相等。表达式的名称用于 `asp-for` 属性值。有关其他信息, 请参阅表达式名称部分。
- 根据模型类型和应用于模型属性的数据注释特性设置 HTML `type` 特性值
- 如果已经指定, 不会覆盖 HTML `type` 属性值
- 通过应用于模型属性的数据注释特性生成 HTML5 验证特性
- 具有与 `Html.TextBoxFor` 和 `Html.EditorFor` 重叠的 HTML 帮助程序功能。有关详细信息, 请参阅**输入标记帮助程序的 HTML 帮助程序替代**项部分。
- 提供强类型化。如果属性的名称更改, 但未更新标记帮助程序, 则会收到类似如下内容的错误:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
 'RegisterViewModel' does not contain a definition for 'Email' and no
 extension method 'Email' accepting a first argument of type 'RegisterViewModel'
 could be found (are you missing a using directive or an assembly reference?)
```

`Input` 标记帮助程序根据 .NET 类型设置 HTML `type` 属性。下表列出一些常见的 .NET 类型和生成的 HTML 类型(并未列出每个 .NET 类型)。

.NET 类型	输入类型
Bool	type="checkbox"
String	type="text"

.NET 类型	输入类型
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single、Double	type="number"

下表显示输入标记帮助程序会映射到特定输入类型的一些常见数据注释属性（并未列出每个验证属性）：

特性	输入类型
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

上述代码生成以下 HTML：

```
<form method="post" action="/Demo/RegisterInput">
     Email:
     <input type="email" data-val="true"
            data-val-email="The Email Address field is not a valid email address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value="" /> <br>
     Password:
     <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password" /><br>
     <button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
 </form>
```

应用于 `Email` 和 `Password` 属性的数据注释在模型中生成元数据。输入标记帮助程序使用模型元数据并生成 HTML5 `data-val-*` 属性(请参阅模型验证)。这些属性描述要附加到输入字段的验证程序。这样可以提供非介入式 HTML5 和 jQuery 验证。非介入式属性具有格式 `data-val-rule="Error Message"`，其中规则是验证规则的名称(例如 `data-val-required`、`data-val-email`、`data-val-maxlength` 等)。如果在属性中提供错误消息，则该错误消息会作为 `data-val-rule` 属性的值显示。还有表单 `data-val-ruleName-argumentName="argumentValue"` 的属性，这些属性提供有关规则的其他详细信息，例如，`data-val-maxlength-max="1024"`。

输入标记帮助程序的 HTML 帮助程序替代项

`Html.TextBox`、`Html.TextBoxFor`、`Html.Editor` 和 `Html.EditorFor` 与输入标记帮助程序的功能存在重叠。输入标记帮助程序会自动设置 `type` 属性；而 `Html.TextBox` 和 `Html.TextBoxFor` 不会。`Html.Editor` 和 `Html.EditorFor` 处理集合、复杂对象和模板；而输入标记帮助程序不会。输入标记帮助程序、`Html.EditorFor` 和 `Html.TextBoxFor` 是强类型(使用 lambda 表达式)；而 `Html.TextBox` 和 `Html.Editor` 不是(使用表达式名称)。

HtmlAttributes

`@Html.Editor()` 和 `@Html.EditorFor()` 在执行其默认模板时使用名为 `htmlAttributes` 的特殊 `ViewDataDictionary` 条目。此行为可选择使用 `additionalViewData` 参数增强。键"htmlAttributes"区分大小写。键"htmlAttributes"的处理方式与传递到输入帮助程序的 `htmlAttributes` 对象(例如 `@Html.TextBox()`)的处理方式类似。

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

表达式名称

`asp-for` 属性值是 `ModelExpression`，并且是 lambda 表达式的右侧。因此，`asp-for="Property1"` 在生成的代码中变成 `m => m.Property1`，这也是无需使用 `Model` 前缀的原因。"@"字符可用作内联表达式的开头，并移到 `m.` 前面：

```
@{
       var joe = "Joe";
   }
   <input asp-for="@joe" />
```

生成以下 HTML：

```
<input type="text" id="joe" name="joe" value="Joe" />
```

使用集合属性时，`asp-for="CollectionProperty[23].Member"` 在 `i` 具有值 `23` 时生成与 `asp-for="CollectionProperty[i].Member"` 相同的名称。

在 ASP.NET Core MVC 计算 `ModelExpression` 的值时，它会检查多个源，包括 `ModelState`。以 `<input type="text" asp-for="@Name" />` 为例。计算出的 `value` 属性是第一个非 null 值，属于：

- 带有“Name”键的 `ModelState` 条目。
- `Model.Name` 表达式的结果。

导航子属性

还可使用视图模型的属性路径导航到子属性。设想一个包含子 `Address` 属性的更复杂的模型类。

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

在视图中，绑定到 `Address.AddressLine1`：

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

为 `Address.AddressLine1` 生成以下 HTML：

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

表达式名称和集合

包含 `Colors` 数组的模型示例：

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

操作方法：

```
public IActionResult Edit(int id, int colorIndex)
   {
       ViewData["Index"] = colorIndex;
       return View(GetPerson(id));
   }
```

以下 Razor 显示如何访问特定 `Color` 元素：

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

Views/Shared/EditorTemplates/String.cshtml 模板：

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

使用 `List<T>` 的示例：

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

以下 Razor 演示如何循环访问集合：

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

Views/Shared/EditorTemplates/ToDoItem.cshtml 模板：

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

应尽量在 `asp-for` 或 `Html.DisplayFor` 等效上下文中使用值 `foreach`。一般情况下，`for` 优于 `foreach`（如果情况允许使用的话），因为它不需要分配枚举器，但在 LINQ 表达式中评估索引器的成本高昂，应最大限度地减少使用它。

> **NOTE**
> 上述带有注释的示例代码演示如何将 lambda 表达式替换为 `@` 运算符来访问列表中的每个 `ToDoItem`。

文本区标记帮助程序

`Textarea Tag Helper` 标记帮助程序类似于输入标记帮助程序。

- 通过模型为 <textarea> 元素生成 `id` 和 `name` 属性以及数据验证属性。
- 提供强类型化。
- HTML 帮助程序替代项：`Html.TextAreaFor`

示例：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

生成以下 HTML：

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

标签标记帮助程序

- 在 元素中为表达式名称生成标签描述和 `for` 属性
- HTML 帮助程序替代项：`Html.LabelFor`。

`Label Tag Helper` 通过纯 HTML 标签元素提供如下优势：

- 可自动从 `Display` 属性中获取描述性标签值。预期的显示名称可能会随时间变化，`Display` 属性和标签标记帮助程序的组合会在其被使用的所有位置应用 `Display`。
- 源代码中的标记更少
- 模型属性的强类型化。

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

为 `<label>` 元素生成以下 HTML：

```
<label for="Email">Email Address</label>
```

标签标记帮助程序生成“Email”的 `for` 属性值，即与 `<input>` 元素关联的 ID。标记帮助程序生成一致的 `id` 和 `for` 元素，方便将其正确关联。本示例中的描述来自 `Display` 属性。如果模型不包含 `Display` 特性，描述将为表达式的属性名称。

验证标记帮助程序

有两个验证标记帮助程序。`Validation Message Tag Helper`（为模型中的单个属性显示验证消息）和 `Validation Summary Tag Helper`（显示验证错误的摘要）。`Input Tag Helper` 根据模型类的数据注释属性将 HTML5 客户端验证属性添加到输入元素中。同时在服务器上执行验证。验证标记帮助程序会在发生验证错误时显示这些错误消息。

验证消息标记帮助程序

- 将 HTML5 `data-valmsg-for="property"` 属性添加到 span 元素中，该元素会附加指定模型属性的输入字段中的验证错误消息。jQuery 会在发生客户端验证错误时在 `<span>` 元素中显示错误消息。
- 还会在服务器上执行验证。客户端可能已禁用 JavaScript，一些验证仅可在服务器端执行。
- HTML 帮助程序替代项：`Html.ValidationMessageFor`

`Validation Message Tag Helper` 与 HTML span 元素中的 `asp-validation-for` 属性配合使用。

```
<span asp-validation-for="Email"></span>
```

验证消息标记帮助程序会生成以下 HTML：

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

对于同一属性，通常在 `Input` 标记帮助程序后使用 `Validation Message Tag Helper`。这样做可在导致错误的输入附近显示所有验证错误消息。

> **NOTE**
>
> 必须拥有包含正确的 JavaScript 和 jQuery 脚本引用的视图才能执行客户端验证。有关详细信息，请参阅模型验证。

发生服务器端验证错误时（例如，禁用自定义服务器端验证或客户端验证时），MVC 会将该错误消息作为 `<span>` 元素的主体。

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

验证摘要标记帮助程序

- 针对具有 `asp-validation-summary` 属性的 `<div>` 元素
- HTML 帮助程序替代项：`@Html.ValidationSummary`

`Validation Summary Tag Helper` 用于显示验证消息的摘要。`asp-validation-summary` 属性值可以是以下任意值：

ASP-VALIDATION-SUMMARY	显示的验证消息
ValidationSummary.All	属性和模型级别
ValidationSummary.ModelOnly	模型
ValidationSummary.None	无

示例

在以下示例中，数据模型使用 `DataAnnotation` 属性修饰，在 `<input>` 元素中生成验证错误消息。验证标记帮助程序会在发生验证错误时显示错误消息：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

生成的 HTML（如果模型有效）：

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"> <br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

选择标记帮助程序

- 为模型属性生成 select 元素和关联的 option 元素。
- 具有 HTML 帮助程序替代项 `Html.DropDownListFor` 和 `Html.ListBoxFor`

`Select Tag Helper` `asp-for` 为 select 元素指定模型属性名称，`asp-items` 指定 option 元素。例如:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

示例:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

`Index` 方法初始化 `CountryViewModel`，设置选定的国家/地区并将其传递到 `Index` 视图。

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTTP POST `Index` 方法显示选定内容:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country +  " selected";
        return RedirectToAction("IndexSuccess", new { message = msg});
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

`Index` 视图:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

生成以下 HTML(选择"CA"时):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
   </form>
```

> **NOTE**
> 不建议将 `ViewBag` 或 `ViewData` 与选择标记帮助程序配合使用。视图模型在提供 MVC 元数据方面更可靠且通常更不容易出现问题。

`asp-for` 属性值是特殊情况，它不要求提供 `Model` 前缀，但其他标记帮助程序属性需要该前缀(例如 `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

枚举绑定

通常可方便地将 `<select>` 与 `enum` 属性配合使用并通过 `enum` 值生成 `SelectListItem` 元素。

示例:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

`GetEnumSelectList` 方法为枚举生成 `SelectList` 对象。

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

可使用 `Display` 属性修饰枚举器列表，以获取更丰富的 UI：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

生成以下 HTML：

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
  </form>
```

选项组

如果视图模型包含一个或多个 `SelectListGroup` 对象，则会生成 HTML `<optgroup>` 元素。

`CountryViewModelGroup` 将 `SelectListItem` 元素分组为“North America”组和“Europe”组：

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

两个组如下所示:

生成的 HTML：

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

多重选择

如果 `asp-for` 属性中指定的属性为 `IEnumerable`，选择标记帮助程序会自动生成 multiple = "multiple" 属性。例如，如果给定以下模型：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

及以下视图：

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

则会生成以下 HTML：

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

无选定内容

如果发现自己在多个页面中使用“未指定”选项，可创建模板用于消除重复的 HTML：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

Views/Shared/EditorTemplates/CountryViewModel.cshtml 模板：

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

添加 HTML <option> 元素并不局限于*无选定内容*用例。例如，以下视图和操作方法会生成与上述代码类似的 HTML：

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

根据当前的 `Country` 值选择正确的 `<option>` 元素（包含 `selected="selected"` 属性）。

```
<form method="post" action="/Home/IndexEmpty">
     <select id="Country" name="Country">
         <option value="">&lt;none&gt;</option>
         <option value="MX">Mexico</option>
         <option value="CA" selected="selected">Canada</option>
         <option value="US">USA</option>
     </select>
     <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

其他资源

- ASP.NET Core 中的标记帮助程序
- HTML Form 元素
- 请求验证令牌
- ASP.NET Core 中的模型绑定
- ASP.NET Core MVC 中的模型验证
- IAttributeAdapter 接口
- 本文档的代码片段

ASP.NET Core 表单中的标记帮助程序

2018/11/10 • • Edit Online

作者：Rick Anderson、Dave Paquette 和 Jerrie Pelser

本文档演示如何使用表单和表单中常用的 HTML 元素。HTML Form 元素提供 Web 应用用于向服务器回发数据的主要机制。本文档的大部分内容介绍标记帮助程序及其如何帮助高效创建可靠的 HTML 表单。建议在阅读本文档前先阅读标记帮助程序简介。

在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。如果存在 HTML 帮助程序替代项，文中会提到。

表单标记帮助程序

表单标记帮助程序：

- 为 MVC 控制器操作或命名路由生成 HTML <FORM> `action` 属性值
- 生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）
- 提供 `asp-route-<Parameter Name>` 属性，其中 `<Parameter Name>` 添加到路由值。`Html.BeginForm` 和 `Html.BeginRouteForm` 的 `routeValues` 参数提供类似的功能。
- 具有 HTML 帮助程序替代项 `Html.BeginForm` 和 `Html.BeginRouteForm`

示例:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

上述表单标记帮助程序生成以下 HTML：

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

MVC 运行时通过表单标记帮助程序属性 `asp-controller` 和 `asp-action` 生成 `action` 属性值。表单标记帮助程序还会生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）。保护纯 HTML 表单免受跨站点请求伪造的影响很难，但表单标记帮助程序可提供此服务。

使用命名路由

`asp-route` 标记帮助程序属性还可为 HTML `action` 属性生成标记。具有名为 `register` 的路由的应用可将以下标记用于注册页：

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Views/Account 文件夹中的许多视图(在新建使用*个人用户帐户*身份验证的 Web 应用时生成)包含 asp-route-returnurl 属性:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **NOTE**
> 使用内置模板时, `returnUrl` 仅会在用户尝试访问授权资源, 但未验证身份或未获得授权的情况下自动填充。如果尝试执行未经授权的访问, 安全中间件会使用 `returnUrl` 集将用户重定向至登录页。

输入标记帮助程序

输入标记帮助程序将 HTML <input> 元素绑定到 Razor 视图中的模型表达式。

语法:

```
<input asp-for="<Expression Name>" />
```

输入标记帮助程序:

- 为 `asp-for` 属性中指定的表达式名称生成 `id` 和 `name` HTML 属性。`asp-for="Property1.Property2"` 与 `m => m.Property1.Property2` 相等。表达式的名称用于 `asp-for` 属性值。有关其他信息, 请参阅表达式名称部分。
- 根据模型类型和应用于模型属性的数据注释特性设置 HTML `type` 特性值
- 如果已经指定, 不会覆盖 HTML `type` 属性值
- 通过应用于模型属性的数据注释特性生成 HTML5 验证特性
- 具有与 `Html.TextBoxFor` 和 `Html.EditorFor` 重叠的 HTML 帮助程序功能。有关详细信息, 请参阅**输入标记帮助程序的 HTML 帮助程序替代**项部分。
- 提供强类型化。如果属性的名称更改, 但未更新标记帮助程序, 则会收到类似如下内容的错误:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
 'RegisterViewModel' does not contain a definition for 'Email' and no
 extension method 'Email' accepting a first argument of type 'RegisterViewModel'
 could be found (are you missing a using directive or an assembly reference?)
```

`Input` 标记帮助程序根据 .NET 类型设置 HTML `type` 属性。下表列出一些常见的 .NET 类型和生成的 HTML 类型(并未列出每个 .NET 类型)。

.NET 类型	输入类型
Bool	type="checkbox"
String	type="text"

.NET 类型	输入类型
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single、Double	type="number"

下表显示输入标记帮助程序会映射到特定输入类型的一些常见数据注释属性（并未列出每个验证属性）：

特性	输入类型
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

上述代码生成以下 HTML：

```
<form method="post" action="/Demo/RegisterInput">
     Email:
     <input type="email" data-val="true"
            data-val-email="The Email Address field is not a valid email address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value="" /> <br>
     Password:
     <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password" /><br>
     <button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
 </form>
```

应用于 `Email` 和 `Password` 属性的数据注释在模型中生成元数据。输入标记帮助程序使用模型元数据并生成 HTML5 `data-val-*` 属性(请参阅模型验证)。这些属性描述要附加到输入字段的验证程序。这样可以提供非介入式 HTML5 和 jQuery 验证。非介入式属性具有格式 `data-val-rule="Error Message"`，其中规则是验证规则的名称(例如 `data-val-required`、`data-val-email`、`data-val-maxlength` 等)。如果在属性中提供错误消息，则该错误消息会作为 `data-val-rule` 属性的值显示。还有表单 `data-val-ruleName-argumentName="argumentValue"` 的属性，这些属性提供有关规则的其他详细信息，例如，`data-val-maxlength-max="1024"`。

输入标记帮助程序的 HTML 帮助程序替代项

`Html.TextBox`、`Html.TextBoxFor`、`Html.Editor` 和 `Html.EditorFor` 与输入标记帮助程序的功能存在重叠。输入标记帮助程序会自动设置 `type` 属性；而 `Html.TextBox` 和 `Html.TextBoxFor` 不会。`Html.Editor` 和 `Html.EditorFor` 处理集合、复杂对象和模板；而输入标记帮助程序不会。输入标记帮助程序、`Html.EditorFor` 和 `Html.TextBoxFor` 是强类型(使用 lambda 表达式)；而 `Html.TextBox` 和 `Html.Editor` 不是(使用表达式名称)。

HtmlAttributes

`@Html.Editor()` 和 `@Html.EditorFor()` 在执行其默认模板时使用名为 `htmlAttributes` 的特殊 `ViewDataDictionary` 条目。此行为可选择使用 `additionalViewData` 参数增强。键"htmlAttributes"区分大小写。键"htmlAttributes"的处理方式与传递到输入帮助程序的 `htmlAttributes` 对象(例如 `@Html.TextBox()`)的处理方式类似。

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

表达式名称

`asp-for` 属性值是 `ModelExpression`，并且是 lambda 表达式的右侧。因此，`asp-for="Property1"` 在生成的代码中变成 `m => m.Property1`，这也是无需使用 `Model` 前缀的原因。"@"字符可用作内联表达式的开头，并移到 `m.` 前面：

```
@{
       var joe = "Joe";
   }
   <input asp-for="@joe" />
```

生成以下 HTML：

```
<input type="text" id="joe" name="joe" value="Joe" />
```

使用集合属性时，`asp-for="CollectionProperty[23].Member"` 在 `i` 具有值 `23` 时生成与 `asp-for="CollectionProperty[i].Member"` 相同的名称。

在 ASP.NET Core MVC 计算 `ModelExpression` 的值时，它会检查多个源，包括 `ModelState`。以 `<input type="text" asp-for="@Name" />` 为例。计算出的 `value` 属性是第一个非 null 值，属于：

- 带有“Name”键的 `ModelState` 条目。
- `Model.Name` 表达式的结果。

导航子属性

还可使用视图模型的属性路径导航到子属性。设想一个包含子 `Address` 属性的更复杂的模型类。

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

在视图中，绑定到 `Address.AddressLine1`：

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

为 `Address.AddressLine1` 生成以下 HTML：

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

表达式名称和集合

包含 `Colors` 数组的模型示例：

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

操作方法：

```
public IActionResult Edit(int id, int colorIndex)
   {
       ViewData["Index"] = colorIndex;
       return View(GetPerson(id));
   }
```

以下 Razor 显示如何访问特定 `Color` 元素：

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

Views/Shared/EditorTemplates/String.cshtml 模板：

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

使用 `List<T>` 的示例：

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

以下 Razor 演示如何循环访问集合：

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

Views/Shared/EditorTemplates/ToDoItem.cshtml 模板：

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

应尽量在 `asp-for` 或 `Html.DisplayFor` 等效上下文中使用值 `foreach`。一般情况下，`for` 优于 `foreach`（如果情况允许使用的话），因为它不需要分配枚举器，但在 LINQ 表达式中评估索引器的成本高昂，应最大限度地减少使用它。

> **NOTE**
> 上述带有注释的示例代码演示如何将 lambda 表达式替换为 `@` 运算符来访问列表中的每个 `ToDoItem`。

文本区标记帮助程序

`Textarea Tag Helper` 标记帮助程序类似于输入标记帮助程序。

- 通过模型为 <textarea> 元素生成 `id` 和 `name` 属性以及数据验证属性。
- 提供强类型化。
- HTML 帮助程序替代项：`Html.TextAreaFor`

示例：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

生成以下 HTML：

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

标签标记帮助程序

- 在 元素中为表达式名称生成标签描述和 `for` 属性
- HTML 帮助程序替代项：`Html.LabelFor`。

`Label Tag Helper` 通过纯 HTML 标签元素提供如下优势：

- 可自动从 `Display` 属性中获取描述性标签值。预期的显示名称可能会随时间变化，`Display` 属性和标签标记帮助程序的组合会在其被使用的所有位置应用 `Display`。
- 源代码中的标记更少
- 模型属性的强类型化。

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

为 `<label>` 元素生成以下 HTML：

```
<label for="Email">Email Address</label>
```

标签标记帮助程序生成“Email”的 `for` 属性值，即与 `<input>` 元素关联的 ID。标记帮助程序生成一致的 `id` 和 `for` 元素，方便将其正确关联。本示例中的描述来自 `Display` 属性。如果模型不包含 `Display` 特性，描述将为表达式的属性名称。

验证标记帮助程序

有两个验证标记帮助程序。`Validation Message Tag Helper`（为模型中的单个属性显示验证消息）和 `Validation Summary Tag Helper`（显示验证错误的摘要）。`Input Tag Helper` 根据模型类的数据注释属性将 HTML5 客户端验证属性添加到输入元素中。同时在服务器上执行验证。验证标记帮助程序会在发生验证错误时显示这些错误消息。

验证消息标记帮助程序

- 将 HTML5 `data-valmsg-for="property"` 属性添加到 span 元素中，该元素会附加指定模型属性的输入字段中的验证错误消息。jQuery 会在发生客户端验证错误时在 `<span>` 元素中显示错误消息。
- 还会在服务器上执行验证。客户端可能已禁用 JavaScript，一些验证仅可在服务器端执行。
- HTML 帮助程序替代项：`Html.ValidationMessageFor`

`Validation Message Tag Helper` 与 HTML span 元素中的 `asp-validation-for` 属性配合使用。

```
<span asp-validation-for="Email"></span>
```

验证消息标记帮助程序会生成以下 HTML：

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

对于同一属性，通常在 `Input` 标记帮助程序后使用 `Validation Message Tag Helper`。这样做可在导致错误的输入附近显示所有验证错误消息。

> **NOTE**
>
> 必须拥有包含正确的 JavaScript 和 jQuery 脚本引用的视图才能执行客户端验证。有关详细信息，请参阅模型验证。

发生服务器端验证错误时（例如，禁用自定义服务器端验证或客户端验证时），MVC 会将该错误消息作为 `<span>` 元素的主体。

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

验证摘要标记帮助程序

- 针对具有 `asp-validation-summary` 属性的 `<div>` 元素
- HTML 帮助程序替代项：`@Html.ValidationSummary`

`Validation Summary Tag Helper` 用于显示验证消息的摘要。`asp-validation-summary` 属性值可以是以下任意值：

ASP-VALIDATION-SUMMARY	显示的验证消息
ValidationSummary.All	属性和模型级别
ValidationSummary.ModelOnly	模型
ValidationSummary.None	无

示例

在以下示例中，数据模型使用 `DataAnnotation` 属性修饰，在 `<input>` 元素中生成验证错误消息。验证标记帮助程序会在发生验证错误时显示错误消息：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

生成的 HTML（如果模型有效）：

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"> <br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

选择标记帮助程序

- 为模型属性生成 select 元素和关联的 option 元素。
- 具有 HTML 帮助程序替代项 `Html.DropDownListFor` 和 `Html.ListBoxFor`

`Select Tag Helper` `asp-for` 为 select 元素指定模型属性名称，`asp-items` 指定 option 元素。例如:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

示例:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

`Index` 方法初始化 `CountryViewModel`，设置选定的国家/地区并将其传递到 `Index` 视图。

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTTP POST `Index` 方法显示选定内容:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country +  " selected";
        return RedirectToAction("IndexSuccess", new { message = msg});
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

`Index` 视图:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

生成以下 HTML(选择"CA"时):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
   </form>
```

> **NOTE**
> 不建议将 `ViewBag` 或 `ViewData` 与选择标记帮助程序配合使用。视图模型在提供 MVC 元数据方面更可靠且通常更不容易出现问题。

`asp-for` 属性值是特殊情况，它不要求提供 `Model` 前缀，但其他标记帮助程序属性需要该前缀(例如 `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

枚举绑定

通常可方便地将 `<select>` 与 `enum` 属性配合使用并通过 `enum` 值生成 `SelectListItem` 元素。

示例:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

`GetEnumSelectList` 方法为枚举生成 `SelectList` 对象。

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

可使用 `Display` 属性修饰枚举器列表，以获取更丰富的 UI：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

生成以下 HTML：

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
  </form>
```

选项组

如果视图模型包含一个或多个 `SelectListGroup` 对象，则会生成 HTML `<optgroup>` 元素。

`CountryViewModelGroup` 将 `SelectListItem` 元素分组为“North America”组和“Europe”组：

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

两个组如下所示:

生成的 HTML：

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

多重选择

如果 `asp-for` 属性中指定的属性为 `IEnumerable`，选择标记帮助程序会自动生成 multiple = "multiple" 属性。例如，如果给定以下模型：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

及以下视图：

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

则会生成以下 HTML：

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

无选定内容

如果发现自己在多个页面中使用“未指定”选项，可创建模板用于消除重复的 HTML：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

Views/Shared/EditorTemplates/CountryViewModel.cshtml 模板：

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

添加 HTML <option> 元素并不局限于*无选定内容*用例。例如，以下视图和操作方法会生成与上述代码类似的 HTML：

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

根据当前的 `Country` 值选择正确的 `<option>` 元素（包含 `selected="selected"` 属性）。

```
<form method="post" action="/Home/IndexEmpty">
     <select id="Country" name="Country">
         <option value="">&lt;none&gt;</option>
         <option value="MX">Mexico</option>
         <option value="CA" selected="selected">Canada</option>
         <option value="US">USA</option>
     </select>
     <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

其他资源

- ASP.NET Core 中的标记帮助程序
- HTML Form 元素
- 请求验证令牌
- ASP.NET Core 中的模型绑定
- ASP.NET Core MVC 中的模型验证
- IAttributeAdapter 接口
- 本文档的代码片段

ASP.NET Core 中的部分标记帮助程序

2018/10/30 • • Edit Online

作者：Scott Addie

有关标记帮助程序的概述，请参阅 ASP.NET Core 中的标记帮助程序。

查看或下载示例代码（如何下载）

概述

Partial 标记帮助程序用于在 Razor 页面和 MVC 应用中呈现分部视图。请考虑：

- 需要 ASP.NET Core 2.1 或更高版本。
- 是 HTML 帮助程序语法的替代方法。
- 以异步方式呈现分部视图。

用于呈现分部视图的 HTML 帮助程序选项包括：

- @await Html.PartialAsync
- @await Html.RenderPartialAsync
- @Html.Partial
- @Html.RenderPartial

本文档中的示例均使用产品模型：

```
namespace TagHelpersBuiltIn.Models
{
    public class Product
    {
        public int Number { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }
    }
}
```

以下是分部标记帮助程序属性的清单。

name

需要 `name` 属性。它指示要呈现的分部视图的名称或路径。提供分部视图名称时，会启动视图发现进程。提供显式路径时，将绕过该进程。有关所有可接受的 `name` 值，请参阅分部视图发现。

以下标记使用显式路径，指示要从共享文件夹加载 _ProductPartial.cshtml。使用 for 属性，将模型传递给分部视图进行绑定。

```
<partial name="Shared/_ProductPartial.cshtml"
         for="Product" />
```

for

`for` 属性分配要根据当前模型评估的 ModelExpression。`ModelExpression` 推断 `@Model.` 语法。例如，可使用 `for="Product"` 而非 `for="@Model.Product"`。通过使用 `@` 符号定义内联表达式来替代此默认推理行为。`for` 属性不能与 model 属性一起使用。

以下标记加载 _ProductPartial.cshtml：

```
<partial name="_ProductPartial"
         for="Product" />
```

分部视图绑定到关联页模型的 `Product` 属性：

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using TagHelpersBuiltIn.Models;

namespace TagHelpersBuiltIn.Pages
{
    public class ProductModel : PageModel
    {
        public Product Product { get; set; }

        public void OnGet()
        {
            Product = new Product
            {
                Number = 1,
                Name = "Test product",
                Description = "This is a test product"
            };
        }
    }
}
```

model

`model` 属性分配模型实例，以传递到分部视图。`model` 属性不能与 for 属性一起使用。

在以下标记中，实例化新的 `Product` 对象并将其传递给 `model` 属性进行绑定：

```
<partial name="_ProductPartial"
         model='new Product { Number = 1, Name = "Test product", Description = "This is a test" }' />
```

view-data

`view-data` 属性分配 ViewDataDictionary，以传递到分部视图。以下标记使整个 ViewData 集合可访问分部视图：

```
@{
    ViewData["IsNumberReadOnly"] = true;
}

<partial name="_ProductViewDataPartial"
         for="Product"
         view-data="ViewData" />
```

在前面的代码中，`IsNumberReadOnly` 键值设置为 `true` 并添加到 ViewData 集合中。因此，在以下分部视图中可访问 `ViewData["IsNumberReadOnly"]`：

```
@model TagHelpersBuiltIn.Models.Product

<div class="form-group">
    <label asp-for="Number"></label>
    @if ((bool)ViewData["IsNumberReadOnly"])
    {
        <input asp-for="Number" type="number" class="form-control" readonly />
    }
    else
    {
        <input asp-for="Number" type="number" class="form-control" />
    }
</div>
<div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" type="text" class="form-control" />
</div>
<div class="form-group">
    <label asp-for="Description"></label>
    <textarea asp-for="Description" rows="4" cols="50" class="form-control"></textarea>
</div>
```

在此示例中，`ViewData["IsNumberReadOnly"]` 的值确定 Number 字段是否显示为只读。

从 HTML 帮助程序迁移

请考虑以下异步 HTML 帮助程序示例。循环访问和显示产品集合。依据 `PartialAsync` 方法的第一个参数，加载 _ProductPartial.cshtml 分部视图。`Product` 模型的实例传递给分部视图进行绑定。

```
@foreach (var product in Model.Products)
{
    @await Html.PartialAsync("_ProductPartial", product)
}
```

以下分部标记帮助程序可实现与 `PartialAsync` HTML 帮助程序相同的异步呈现行为。`model` 属性分配有 `Product` 模型实例以绑定到分部视图。

```
@foreach (var product in Model.Products)
{
    <partial name="_ProductPartial" model="product" />
}
```

其他资源

- ASP.NET Core 中的分部视图
- ASP.NET Core MVC 中的视图

ASP.NET Core 表单中的标记帮助程序

2018/11/10 • • Edit Online

作者：Rick Anderson、Dave Paquette 和 Jerrie Pelser

本文档演示如何使用表单和表单中常用的 HTML 元素。HTML Form 元素提供 Web 应用用于向服务器回发数据的主要机制。本文档的大部分内容介绍标记帮助程序及其如何帮助高效创建可靠的 HTML 表单。建议在阅读本文档前先阅读标记帮助程序简介。

在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。如果存在 HTML 帮助程序替代项，文中会提到。

表单标记帮助程序

表单标记帮助程序：

- 为 MVC 控制器操作或命名路由生成 HTML <FORM> `action` 属性值
- 生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）
- 提供 `asp-route-<Parameter Name>` 属性，其中 `<Parameter Name>` 添加到路由值。`Html.BeginForm` 和 `Html.BeginRouteForm` 的 `routeValues` 参数提供类似的功能。
- 具有 HTML 帮助程序替代项 `Html.BeginForm` 和 `Html.BeginRouteForm`

示例:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

上述表单标记帮助程序生成以下 HTML：

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

MVC 运行时通过表单标记帮助程序属性 `asp-controller` 和 `asp-action` 生成 `action` 属性值。表单标记帮助程序还会生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）。保护纯 HTML 表单免受跨站点请求伪造的影响很难，但表单标记帮助程序可提供此服务。

使用命名路由

`asp-route` 标记帮助程序属性还可为 HTML `action` 属性生成标记。具有名为 `register` 的路由的应用可将以下标记用于注册页：

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Views/Account 文件夹中的许多视图(在新建使用*个人用户帐户*身份验证的 Web 应用时生成)包含 asp-route-returnurl 属性:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **NOTE**
> 使用内置模板时, `returnUrl` 仅会在用户尝试访问授权资源, 但未验证身份或未获得授权的情况下自动填充。如果尝试执行未经授权的访问, 安全中间件会使用 `returnUrl` 集将用户重定向至登录页。

输入标记帮助程序

输入标记帮助程序将 HTML <input> 元素绑定到 Razor 视图中的模型表达式。

语法:

```
<input asp-for="<Expression Name>" />
```

输入标记帮助程序:

- 为 `asp-for` 属性中指定的表达式名称生成 `id` 和 `name` HTML 属性。`asp-for="Property1.Property2"` 与 `m => m.Property1.Property2` 相等。表达式的名称用于 `asp-for` 属性值。有关其他信息, 请参阅表达式名称部分。
- 根据模型类型和应用于模型属性的数据注释特性设置 HTML `type` 特性值
- 如果已经指定, 不会覆盖 HTML `type` 属性值
- 通过应用于模型属性的数据注释特性生成 HTML5 验证特性
- 具有与 `Html.TextBoxFor` 和 `Html.EditorFor` 重叠的 HTML 帮助程序功能。有关详细信息, 请参阅**输入标记帮助程序的 HTML 帮助程序替代**项部分。
- 提供强类型化。如果属性的名称更改, 但未更新标记帮助程序, 则会收到类似如下内容的错误:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
 'RegisterViewModel' does not contain a definition for 'Email' and no
 extension method 'Email' accepting a first argument of type 'RegisterViewModel'
 could be found (are you missing a using directive or an assembly reference?)
```

`Input` 标记帮助程序根据 .NET 类型设置 HTML `type` 属性。下表列出一些常见的 .NET 类型和生成的 HTML 类型(并未列出每个 .NET 类型)。

.NET 类型	输入类型
Bool	type="checkbox"
String	type="text"

.NET 类型	输入类型
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single、Double	type="number"

下表显示输入标记帮助程序会映射到特定输入类型的一些常见数据注释属性（并未列出每个验证属性）：

特性	输入类型
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

上述代码生成以下 HTML：

```
<form method="post" action="/Demo/RegisterInput">
     Email:
     <input type="email" data-val="true"
            data-val-email="The Email Address field is not a valid email address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value="" /> <br>
     Password:
     <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password" /><br>
     <button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
 </form>
```

应用于 `Email` 和 `Password` 属性的数据注释在模型中生成元数据。输入标记帮助程序使用模型元数据并生成 HTML5 `data-val-*` 属性(请参阅模型验证)。这些属性描述要附加到输入字段的验证程序。这样可以提供非介入式 HTML5 和 jQuery 验证。非介入式属性具有格式 `data-val-rule="Error Message"`，其中规则是验证规则的名称(例如 `data-val-required`、`data-val-email`、`data-val-maxlength` 等)。如果在属性中提供错误消息，则该错误消息会作为 `data-val-rule` 属性的值显示。还有表单 `data-val-ruleName-argumentName="argumentValue"` 的属性，这些属性提供有关规则的其他详细信息，例如，`data-val-maxlength-max="1024"`。

输入标记帮助程序的 HTML 帮助程序替代项

`Html.TextBox`、`Html.TextBoxFor`、`Html.Editor` 和 `Html.EditorFor` 与输入标记帮助程序的功能存在重叠。输入标记帮助程序会自动设置 `type` 属性；而 `Html.TextBox` 和 `Html.TextBoxFor` 不会。`Html.Editor` 和 `Html.EditorFor` 处理集合、复杂对象和模板；而输入标记帮助程序不会。输入标记帮助程序、`Html.EditorFor` 和 `Html.TextBoxFor` 是强类型(使用 lambda 表达式)；而 `Html.TextBox` 和 `Html.Editor` 不是(使用表达式名称)。

HtmlAttributes

`@Html.Editor()` 和 `@Html.EditorFor()` 在执行其默认模板时使用名为 `htmlAttributes` 的特殊 `ViewDataDictionary` 条目。此行为可选择使用 `additionalViewData` 参数增强。键"htmlAttributes"区分大小写。键"htmlAttributes"的处理方式与传递到输入帮助程序的 `htmlAttributes` 对象(例如 `@Html.TextBox()`)的处理方式类似。

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

表达式名称

`asp-for` 属性值是 `ModelExpression`，并且是 lambda 表达式的右侧。因此，`asp-for="Property1"` 在生成的代码中变成 `m => m.Property1`，这也是无需使用 `Model` 前缀的原因。"@"字符可用作内联表达式的开头，并移到 `m.` 前面：

```
@{
       var joe = "Joe";
   }
   <input asp-for="@joe" />
```

生成以下 HTML：

```
<input type="text" id="joe" name="joe" value="Joe" />
```

使用集合属性时，`asp-for="CollectionProperty[23].Member"` 在 `i` 具有值 `23` 时生成与 `asp-for="CollectionProperty[i].Member"` 相同的名称。

在 ASP.NET Core MVC 计算 `ModelExpression` 的值时，它会检查多个源，包括 `ModelState`。以 `<input type="text" asp-for="@Name" />` 为例。计算出的 `value` 属性是第一个非 null 值，属于：

- 带有“Name”键的 `ModelState` 条目。
- `Model.Name` 表达式的结果。

导航子属性

还可使用视图模型的属性路径导航到子属性。设想一个包含子 `Address` 属性的更复杂的模型类。

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

在视图中，绑定到 `Address.AddressLine1`：

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

为 `Address.AddressLine1` 生成以下 HTML：

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

表达式名称和集合

包含 `Colors` 数组的模型示例：

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

操作方法：

```
public IActionResult Edit(int id, int colorIndex)
   {
       ViewData["Index"] = colorIndex;
       return View(GetPerson(id));
   }
```

以下 Razor 显示如何访问特定 `Color` 元素：

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

Views/Shared/EditorTemplates/String.cshtml 模板：

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

使用 `List<T>` 的示例：

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

以下 Razor 演示如何循环访问集合：

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

Views/Shared/EditorTemplates/ToDoItem.cshtml 模板：

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

应尽量在 `asp-for` 或 `Html.DisplayFor` 等效上下文中使用值 `foreach`。一般情况下，`for` 优于 `foreach`（如果情况允许使用的话），因为它不需要分配枚举器，但在 LINQ 表达式中评估索引器的成本高昂，应最大限度地减少使用它。

> **NOTE**
> 上述带有注释的示例代码演示如何将 lambda 表达式替换为 `@` 运算符来访问列表中的每个 `ToDoItem`。

文本区标记帮助程序

`Textarea Tag Helper` 标记帮助程序类似于输入标记帮助程序。

- 通过模型为 <textarea> 元素生成 `id` 和 `name` 属性以及数据验证属性。
- 提供强类型化。
- HTML 帮助程序替代项：`Html.TextAreaFor`

示例：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

生成以下 HTML：

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

标签标记帮助程序

- 在 元素中为表达式名称生成标签描述和 `for` 属性
- HTML 帮助程序替代项：`Html.LabelFor`。

`Label Tag Helper` 通过纯 HTML 标签元素提供如下优势：

- 可自动从 `Display` 属性中获取描述性标签值。预期的显示名称可能会随时间变化，`Display` 属性和标签标记帮助程序的组合会在其被使用的所有位置应用 `Display`。
- 源代码中的标记更少
- 模型属性的强类型化。

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

为 `<label>` 元素生成以下 HTML：

```
<label for="Email">Email Address</label>
```

标签标记帮助程序生成“Email”的 `for` 属性值，即与 `<input>` 元素关联的 ID。标记帮助程序生成一致的 `id` 和 `for` 元素，方便将其正确关联。本示例中的描述来自 `Display` 属性。如果模型不包含 `Display` 特性，描述将为表达式的属性名称。

验证标记帮助程序

有两个验证标记帮助程序。`Validation Message Tag Helper`（为模型中的单个属性显示验证消息）和 `Validation Summary Tag Helper`（显示验证错误的摘要）。`Input Tag Helper` 根据模型类的数据注释属性将 HTML5 客户端验证属性添加到输入元素中。同时在服务器上执行验证。验证标记帮助程序会在发生验证错误时显示这些错误消息。

验证消息标记帮助程序

- 将 HTML5 `data-valmsg-for="property"` 属性添加到 span 元素中，该元素会附加指定模型属性的输入字段中的验证错误消息。jQuery 会在发生客户端验证错误时在 `<span>` 元素中显示错误消息。
- 还会在服务器上执行验证。客户端可能已禁用 JavaScript，一些验证仅可在服务器端执行。
- HTML 帮助程序替代项：`Html.ValidationMessageFor`

`Validation Message Tag Helper` 与 HTML span 元素中的 `asp-validation-for` 属性配合使用。

```
<span asp-validation-for="Email"></span>
```

验证消息标记帮助程序会生成以下 HTML：

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

对于同一属性，通常在 `Input` 标记帮助程序后使用 `Validation Message Tag Helper`。这样做可在导致错误的输入附近显示所有验证错误消息。

> **NOTE**
>
> 必须拥有包含正确的 JavaScript 和 jQuery 脚本引用的视图才能执行客户端验证。有关详细信息，请参阅模型验证。

发生服务器端验证错误时（例如，禁用自定义服务器端验证或客户端验证时），MVC 会将该错误消息作为 `<span>` 元素的主体。

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

验证摘要标记帮助程序

- 针对具有 `asp-validation-summary` 属性的 `<div>` 元素
- HTML 帮助程序替代项：`@Html.ValidationSummary`

`Validation Summary Tag Helper` 用于显示验证消息的摘要。`asp-validation-summary` 属性值可以是以下任意值：

ASP-VALIDATION-SUMMARY	显示的验证消息
ValidationSummary.All	属性和模型级别
ValidationSummary.ModelOnly	模型
ValidationSummary.None	无

示例

在以下示例中，数据模型使用 `DataAnnotation` 属性修饰，在 `<input>` 元素中生成验证错误消息。验证标记帮助程序会在发生验证错误时显示错误消息：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

生成的 HTML（如果模型有效）：

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"> <br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

选择标记帮助程序

- 为模型属性生成 select 元素和关联的 option 元素。
- 具有 HTML 帮助程序替代项 `Html.DropDownListFor` 和 `Html.ListBoxFor`

`Select Tag Helper` `asp-for` 为 select 元素指定模型属性名称，`asp-items` 指定 option 元素。例如:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

示例:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

`Index` 方法初始化 `CountryViewModel`，设置选定的国家/地区并将其传递到 `Index` 视图。

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTTP POST `Index` 方法显示选定内容:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country +  " selected";
        return RedirectToAction("IndexSuccess", new { message = msg});
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

`Index` 视图:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

生成以下 HTML(选择"CA"时):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
   </form>
```

> **NOTE**
> 不建议将 `ViewBag` 或 `ViewData` 与选择标记帮助程序配合使用。视图模型在提供 MVC 元数据方面更可靠且通常更不容易出现问题。

`asp-for` 属性值是特殊情况，它不要求提供 `Model` 前缀，但其他标记帮助程序属性需要该前缀(例如 `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

枚举绑定

通常可方便地将 `<select>` 与 `enum` 属性配合使用并通过 `enum` 值生成 `SelectListItem` 元素。

示例:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

`GetEnumSelectList` 方法为枚举生成 `SelectList` 对象。

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

可使用 `Display` 属性修饰枚举器列表，以获取更丰富的 UI：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

生成以下 HTML：

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
  </form>
```

选项组

如果视图模型包含一个或多个 `SelectListGroup` 对象，则会生成 HTML `<optgroup>` 元素。

`CountryViewModelGroup` 将 `SelectListItem` 元素分组为“North America”组和“Europe”组：

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

两个组如下所示:

生成的 HTML：

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

多重选择

如果 `asp-for` 属性中指定的属性为 `IEnumerable`，选择标记帮助程序会自动生成 multiple = "multiple" 属性。例如，如果给定以下模型：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

及以下视图：

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

则会生成以下 HTML：

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

无选定内容

如果发现自己在多个页面中使用“未指定”选项，可创建模板用于消除重复的 HTML：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

Views/Shared/EditorTemplates/CountryViewModel.cshtml 模板：

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

添加 HTML <option> 元素并不局限于*无选定内容*用例。例如，以下视图和操作方法会生成与上述代码类似的 HTML：

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

根据当前的 `Country` 值选择正确的 `<option>` 元素（包含 `selected="selected"` 属性）。

```
<form method="post" action="/Home/IndexEmpty">
     <select id="Country" name="Country">
         <option value="">&lt;none&gt;</option>
         <option value="MX">Mexico</option>
         <option value="CA" selected="selected">Canada</option>
         <option value="US">USA</option>
     </select>
     <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

其他资源

- ASP.NET Core 中的标记帮助程序
- HTML Form 元素
- 请求验证令牌
- ASP.NET Core 中的模型绑定
- ASP.NET Core MVC 中的模型验证
- IAttributeAdapter 接口
- 本文档的代码片段

ASP.NET Core 表单中的标记帮助程序

2018/11/10 • • Edit Online

作者：Rick Anderson、Dave Paquette 和 Jerrie Pelser

本文档演示如何使用表单和表单中常用的 HTML 元素。HTML Form 元素提供 Web 应用用于向服务器回发数据的主要机制。本文档的大部分内容介绍标记帮助程序及其如何帮助高效创建可靠的 HTML 表单。建议在阅读本文档前先阅读标记帮助程序简介。

在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。如果存在 HTML 帮助程序替代项，文中会提到。

表单标记帮助程序

表单标记帮助程序：

- 为 MVC 控制器操作或命名路由生成 HTML <FORM> `action` 属性值
- 生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）
- 提供 `asp-route-<Parameter Name>` 属性，其中 `<Parameter Name>` 添加到路由值。`Html.BeginForm` 和 `Html.BeginRouteForm` 的 `routeValues` 参数提供类似的功能。
- 具有 HTML 帮助程序替代项 `Html.BeginForm` 和 `Html.BeginRouteForm`

示例:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

上述表单标记帮助程序生成以下 HTML：

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

MVC 运行时通过表单标记帮助程序属性 `asp-controller` 和 `asp-action` 生成 `action` 属性值。表单标记帮助程序还会生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）。保护纯 HTML 表单免受跨站点请求伪造的影响很难，但表单标记帮助程序可提供此服务。

使用命名路由

`asp-route` 标记帮助程序属性还可为 HTML `action` 属性生成标记。具有名为 `register` 的路由的应用可将以下标记用于注册页：

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Views/Account 文件夹中的许多视图(在新建使用*个人用户帐户*身份验证的 Web 应用时生成)包含 asp-route-returnurl 属性:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **NOTE**
> 使用内置模板时, `returnUrl` 仅会在用户尝试访问授权资源, 但未验证身份或未获得授权的情况下自动填充。如果尝试执行未经授权的访问, 安全中间件会使用 `returnUrl` 集将用户重定向至登录页。

输入标记帮助程序

输入标记帮助程序将 HTML <input> 元素绑定到 Razor 视图中的模型表达式。

语法:

```
<input asp-for="<Expression Name>" />
```

输入标记帮助程序:

- 为 `asp-for` 属性中指定的表达式名称生成 `id` 和 `name` HTML 属性。`asp-for="Property1.Property2"` 与 `m => m.Property1.Property2` 相等。表达式的名称用于 `asp-for` 属性值。有关其他信息, 请参阅表达式名称部分。
- 根据模型类型和应用于模型属性的数据注释特性设置 HTML `type` 特性值
- 如果已经指定, 不会覆盖 HTML `type` 属性值
- 通过应用于模型属性的数据注释特性生成 HTML5 验证特性
- 具有与 `Html.TextBoxFor` 和 `Html.EditorFor` 重叠的 HTML 帮助程序功能。有关详细信息, 请参阅**输入标记帮助程序的 HTML 帮助程序替代**项部分。
- 提供强类型化。如果属性的名称更改, 但未更新标记帮助程序, 则会收到类似如下内容的错误:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
 'RegisterViewModel' does not contain a definition for 'Email' and no
 extension method 'Email' accepting a first argument of type 'RegisterViewModel'
 could be found (are you missing a using directive or an assembly reference?)
```

`Input` 标记帮助程序根据 .NET 类型设置 HTML `type` 属性。下表列出一些常见的 .NET 类型和生成的 HTML 类型(并未列出每个 .NET 类型)。

.NET 类型	输入类型
Bool	type="checkbox"
String	type="text"

.NET 类型	输入类型
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single、Double	type="number"

下表显示输入标记帮助程序会映射到特定输入类型的一些常见数据注释属性（并未列出每个验证属性）：

特性	输入类型
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

上述代码生成以下 HTML：

```
<form method="post" action="/Demo/RegisterInput">
     Email:
     <input type="email" data-val="true"
            data-val-email="The Email Address field is not a valid email address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value="" /> <br>
     Password:
     <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password" /><br>
     <button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
 </form>
```

应用于 `Email` 和 `Password` 属性的数据注释在模型中生成元数据。输入标记帮助程序使用模型元数据并生成 HTML5 `data-val-*` 属性(请参阅模型验证)。这些属性描述要附加到输入字段的验证程序。这样可以提供非介入式 HTML5 和 jQuery 验证。非介入式属性具有格式 `data-val-rule="Error Message"`，其中规则是验证规则的名称(例如 `data-val-required`、`data-val-email`、`data-val-maxlength` 等)。如果在属性中提供错误消息，则该错误消息会作为 `data-val-rule` 属性的值显示。还有表单 `data-val-ruleName-argumentName="argumentValue"` 的属性，这些属性提供有关规则的其他详细信息，例如，`data-val-maxlength-max="1024"`。

输入标记帮助程序的 HTML 帮助程序替代项

`Html.TextBox`、`Html.TextBoxFor`、`Html.Editor` 和 `Html.EditorFor` 与输入标记帮助程序的功能存在重叠。输入标记帮助程序会自动设置 `type` 属性；而 `Html.TextBox` 和 `Html.TextBoxFor` 不会。`Html.Editor` 和 `Html.EditorFor` 处理集合、复杂对象和模板；而输入标记帮助程序不会。输入标记帮助程序、`Html.EditorFor` 和 `Html.TextBoxFor` 是强类型(使用 lambda 表达式)；而 `Html.TextBox` 和 `Html.Editor` 不是(使用表达式名称)。

HtmlAttributes

`@Html.Editor()` 和 `@Html.EditorFor()` 在执行其默认模板时使用名为 `htmlAttributes` 的特殊 `ViewDataDictionary` 条目。此行为可选择使用 `additionalViewData` 参数增强。键"htmlAttributes"区分大小写。键"htmlAttributes"的处理方式与传递到输入帮助程序的 `htmlAttributes` 对象(例如 `@Html.TextBox()`)的处理方式类似。

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

表达式名称

`asp-for` 属性值是 `ModelExpression`，并且是 lambda 表达式的右侧。因此，`asp-for="Property1"` 在生成的代码中变成 `m => m.Property1`，这也是无需使用 `Model` 前缀的原因。"@"字符可用作内联表达式的开头，并移到 `m.` 前面：

```
@{
       var joe = "Joe";
   }
   <input asp-for="@joe" />
```

生成以下 HTML：

```
<input type="text" id="joe" name="joe" value="Joe" />
```

使用集合属性时，`asp-for="CollectionProperty[23].Member"` 在 `i` 具有值 `23` 时生成与 `asp-for="CollectionProperty[i].Member"` 相同的名称。

在 ASP.NET Core MVC 计算 `ModelExpression` 的值时，它会检查多个源，包括 `ModelState`。以 `<input type="text" asp-for="@Name" />` 为例。计算出的 `value` 属性是第一个非 null 值，属于：

- 带有“Name”键的 `ModelState` 条目。
- `Model.Name` 表达式的结果。

导航子属性

还可使用视图模型的属性路径导航到子属性。设想一个包含子 `Address` 属性的更复杂的模型类。

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

在视图中，绑定到 `Address.AddressLine1`：

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

为 `Address.AddressLine1` 生成以下 HTML：

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

表达式名称和集合

包含 `Colors` 数组的模型示例：

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

操作方法：

```
public IActionResult Edit(int id, int colorIndex)
   {
       ViewData["Index"] = colorIndex;
       return View(GetPerson(id));
   }
```

以下 Razor 显示如何访问特定 `Color` 元素：

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

Views/Shared/EditorTemplates/String.cshtml 模板：

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

使用 `List<T>` 的示例：

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

以下 Razor 演示如何循环访问集合：

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

Views/Shared/EditorTemplates/ToDoItem.cshtml 模板：

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

应尽量在 `asp-for` 或 `Html.DisplayFor` 等效上下文中使用值 `foreach`。一般情况下，`for` 优于 `foreach`（如果情况允许使用的话），因为它不需要分配枚举器，但在 LINQ 表达式中评估索引器的成本高昂，应最大限度地减少使用它。

> **NOTE**
> 上述带有注释的示例代码演示如何将 lambda 表达式替换为 `@` 运算符来访问列表中的每个 `ToDoItem`。

文本区标记帮助程序

`Textarea Tag Helper` 标记帮助程序类似于输入标记帮助程序。

- 通过模型为 <textarea> 元素生成 `id` 和 `name` 属性以及数据验证属性。
- 提供强类型化。
- HTML 帮助程序替代项：`Html.TextAreaFor`

示例：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

生成以下 HTML：

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

标签标记帮助程序

- 在 元素中为表达式名称生成标签描述和 `for` 属性
- HTML 帮助程序替代项：`Html.LabelFor`。

`Label Tag Helper` 通过纯 HTML 标签元素提供如下优势：

- 可自动从 `Display` 属性中获取描述性标签值。预期的显示名称可能会随时间变化，`Display` 属性和标签标记帮助程序的组合会在其被使用的所有位置应用 `Display`。
- 源代码中的标记更少
- 模型属性的强类型化。

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

为 `<label>` 元素生成以下 HTML：

```
<label for="Email">Email Address</label>
```

标签标记帮助程序生成“Email”的 `for` 属性值，即与 `<input>` 元素关联的 ID。标记帮助程序生成一致的 `id` 和 `for` 元素，方便将其正确关联。本示例中的描述来自 `Display` 属性。如果模型不包含 `Display` 特性，描述将为表达式的属性名称。

验证标记帮助程序

有两个验证标记帮助程序。`Validation Message Tag Helper`（为模型中的单个属性显示验证消息）和 `Validation Summary Tag Helper`（显示验证错误的摘要）。`Input Tag Helper` 根据模型类的数据注释属性将 HTML5 客户端验证属性添加到输入元素中。同时在服务器上执行验证。验证标记帮助程序会在发生验证错误时显示这些错误消息。

验证消息标记帮助程序

- 将 HTML5 `data-valmsg-for="property"` 属性添加到 span 元素中，该元素会附加指定模型属性的输入字段中的验证错误消息。jQuery 会在发生客户端验证错误时在 `<span>` 元素中显示错误消息。
- 还会在服务器上执行验证。客户端可能已禁用 JavaScript，一些验证仅可在服务器端执行。
- HTML 帮助程序替代项：`Html.ValidationMessageFor`

`Validation Message Tag Helper` 与 HTML span 元素中的 `asp-validation-for` 属性配合使用。

```
<span asp-validation-for="Email"></span>
```

验证消息标记帮助程序会生成以下 HTML：

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

对于同一属性，通常在 `Input` 标记帮助程序后使用 `Validation Message Tag Helper`。这样做可在导致错误的输入附近显示所有验证错误消息。

> **NOTE**
>
> 必须拥有包含正确的 JavaScript 和 jQuery 脚本引用的视图才能执行客户端验证。有关详细信息，请参阅模型验证。

发生服务器端验证错误时（例如，禁用自定义服务器端验证或客户端验证时），MVC 会将该错误消息作为 `<span>` 元素的主体。

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

验证摘要标记帮助程序

- 针对具有 `asp-validation-summary` 属性的 `<div>` 元素
- HTML 帮助程序替代项：`@Html.ValidationSummary`

`Validation Summary Tag Helper` 用于显示验证消息的摘要。`asp-validation-summary` 属性值可以是以下任意值：

ASP-VALIDATION-SUMMARY	显示的验证消息
ValidationSummary.All	属性和模型级别
ValidationSummary.ModelOnly	模型
ValidationSummary.None	无

示例

在以下示例中，数据模型使用 `DataAnnotation` 属性修饰，在 `<input>` 元素中生成验证错误消息。验证标记帮助程序会在发生验证错误时显示错误消息：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

生成的 HTML（如果模型有效）：

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"> <br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

选择标记帮助程序

- 为模型属性生成 select 元素和关联的 option 元素。
- 具有 HTML 帮助程序替代项 `Html.DropDownListFor` 和 `Html.ListBoxFor`

`Select Tag Helper` `asp-for` 为 select 元素指定模型属性名称，`asp-items` 指定 option 元素。例如:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

示例:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

`Index` 方法初始化 `CountryViewModel`，设置选定的国家/地区并将其传递到 `Index` 视图。

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTTP POST `Index` 方法显示选定内容:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country +  " selected";
        return RedirectToAction("IndexSuccess", new { message = msg});
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

`Index` 视图:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

生成以下 HTML(选择"CA"时):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
   </form>
```

> **NOTE**
> 不建议将 `ViewBag` 或 `ViewData` 与选择标记帮助程序配合使用。视图模型在提供 MVC 元数据方面更可靠且通常更不容易出现问题。

`asp-for` 属性值是特殊情况，它不要求提供 `Model` 前缀，但其他标记帮助程序属性需要该前缀(例如 `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

枚举绑定

通常可方便地将 `<select>` 与 `enum` 属性配合使用并通过 `enum` 值生成 `SelectListItem` 元素。

示例:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

`GetEnumSelectList` 方法为枚举生成 `SelectList` 对象。

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

可使用 `Display` 属性修饰枚举器列表，以获取更丰富的 UI：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

生成以下 HTML：

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
  </form>
```

选项组

如果视图模型包含一个或多个 `SelectListGroup` 对象，则会生成 HTML `<optgroup>` 元素。

`CountryViewModelGroup` 将 `SelectListItem` 元素分组为“North America”组和“Europe”组：

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

两个组如下所示:

生成的 HTML：

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

多重选择

如果 `asp-for` 属性中指定的属性为 `IEnumerable`，选择标记帮助程序会自动生成 multiple = "multiple" 属性。例如，如果给定以下模型：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

及以下视图：

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

则会生成以下 HTML：

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

无选定内容

如果发现自己在多个页面中使用“未指定”选项，可创建模板用于消除重复的 HTML：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

Views/Shared/EditorTemplates/CountryViewModel.cshtml 模板：

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

添加 HTML <option> 元素并不局限于*无选定内容*用例。例如，以下视图和操作方法会生成与上述代码类似的 HTML：

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

根据当前的 `Country` 值选择正确的 `<option>` 元素（包含 `selected="selected"` 属性）。

```
<form method="post" action="/Home/IndexEmpty">
     <select id="Country" name="Country">
         <option value="">&lt;none&gt;</option>
         <option value="MX">Mexico</option>
         <option value="CA" selected="selected">Canada</option>
         <option value="US">USA</option>
     </select>
     <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

其他资源

- ASP.NET Core 中的标记帮助程序
- HTML Form 元素
- 请求验证令牌
- ASP.NET Core 中的模型绑定
- ASP.NET Core MVC 中的模型验证
- IAttributeAdapter 接口
- 本文档的代码片段

ASP.NET Core 表单中的标记帮助程序

2018/11/10 • • Edit Online

作者：Rick Anderson、Dave Paquette 和 Jerrie Pelser

本文档演示如何使用表单和表单中常用的 HTML 元素。HTML Form 元素提供 Web 应用用于向服务器回发数据的主要机制。本文档的大部分内容介绍标记帮助程序及其如何帮助高效创建可靠的 HTML 表单。建议在阅读本文档前先阅读标记帮助程序简介。

在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。如果存在 HTML 帮助程序替代项，文中会提到。

表单标记帮助程序

表单标记帮助程序：

- 为 MVC 控制器操作或命名路由生成 HTML <FORM> `action` 属性值
- 生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）
- 提供 `asp-route-<Parameter Name>` 属性，其中 `<Parameter Name>` 添加到路由值。`Html.BeginForm` 和 `Html.BeginRouteForm` 的 `routeValues` 参数提供类似的功能。
- 具有 HTML 帮助程序替代项 `Html.BeginForm` 和 `Html.BeginRouteForm`

示例:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

上述表单标记帮助程序生成以下 HTML：

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

MVC 运行时通过表单标记帮助程序属性 `asp-controller` 和 `asp-action` 生成 `action` 属性值。表单标记帮助程序还会生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）。保护纯 HTML 表单免受跨站点请求伪造的影响很难，但表单标记帮助程序可提供此服务。

使用命名路由

`asp-route` 标记帮助程序属性还可为 HTML `action` 属性生成标记。具有名为 `register` 的路由的应用可将以下标记用于注册页：

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Views/Account 文件夹中的许多视图(在新建使用*个人用户帐户*身份验证的 Web 应用时生成)包含 asp-route-returnurl 属性:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **NOTE**
> 使用内置模板时, `returnUrl` 仅会在用户尝试访问授权资源, 但未验证身份或未获得授权的情况下自动填充。如果尝试执行未经授权的访问, 安全中间件会使用 `returnUrl` 集将用户重定向至登录页。

输入标记帮助程序

输入标记帮助程序将 HTML <input> 元素绑定到 Razor 视图中的模型表达式。

语法:

```
<input asp-for="<Expression Name>" />
```

输入标记帮助程序:

- 为 `asp-for` 属性中指定的表达式名称生成 `id` 和 `name` HTML 属性。`asp-for="Property1.Property2"` 与 `m => m.Property1.Property2` 相等。表达式的名称用于 `asp-for` 属性值。有关其他信息, 请参阅表达式名称部分。
- 根据模型类型和应用于模型属性的数据注释特性设置 HTML `type` 特性值
- 如果已经指定, 不会覆盖 HTML `type` 属性值
- 通过应用于模型属性的数据注释特性生成 HTML5 验证特性
- 具有与 `Html.TextBoxFor` 和 `Html.EditorFor` 重叠的 HTML 帮助程序功能。有关详细信息, 请参阅**输入标记帮助程序的 HTML 帮助程序替代**项部分。
- 提供强类型化。如果属性的名称更改, 但未更新标记帮助程序, 则会收到类似如下内容的错误:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
 'RegisterViewModel' does not contain a definition for 'Email' and no
 extension method 'Email' accepting a first argument of type 'RegisterViewModel'
 could be found (are you missing a using directive or an assembly reference?)
```

`Input` 标记帮助程序根据 .NET 类型设置 HTML `type` 属性。下表列出一些常见的 .NET 类型和生成的 HTML 类型(并未列出每个 .NET 类型)。

.NET 类型	输入类型
Bool	type="checkbox"
String	type="text"

.NET 类型	输入类型
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single、Double	type="number"

下表显示输入标记帮助程序会映射到特定输入类型的一些常见数据注释属性（并未列出每个验证属性）：

特性	输入类型
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

上述代码生成以下 HTML：

```
<form method="post" action="/Demo/RegisterInput">
     Email:
     <input type="email" data-val="true"
            data-val-email="The Email Address field is not a valid email address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value="" /> <br>
     Password:
     <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password" /><br>
     <button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
 </form>
```

应用于 `Email` 和 `Password` 属性的数据注释在模型中生成元数据。输入标记帮助程序使用模型元数据并生成 HTML5 `data-val-*` 属性(请参阅模型验证)。这些属性描述要附加到输入字段的验证程序。这样可以提供非介入式 HTML5 和 jQuery 验证。非介入式属性具有格式 `data-val-rule="Error Message"`，其中规则是验证规则的名称(例如 `data-val-required`、`data-val-email`、`data-val-maxlength` 等)。如果在属性中提供错误消息，则该错误消息会作为 `data-val-rule` 属性的值显示。还有表单 `data-val-ruleName-argumentName="argumentValue"` 的属性，这些属性提供有关规则的其他详细信息，例如，`data-val-maxlength-max="1024"`。

输入标记帮助程序的 HTML 帮助程序替代项

`Html.TextBox`、`Html.TextBoxFor`、`Html.Editor` 和 `Html.EditorFor` 与输入标记帮助程序的功能存在重叠。输入标记帮助程序会自动设置 `type` 属性；而 `Html.TextBox` 和 `Html.TextBoxFor` 不会。`Html.Editor` 和 `Html.EditorFor` 处理集合、复杂对象和模板；而输入标记帮助程序不会。输入标记帮助程序、`Html.EditorFor` 和 `Html.TextBoxFor` 是强类型(使用 lambda 表达式)；而 `Html.TextBox` 和 `Html.Editor` 不是(使用表达式名称)。

HtmlAttributes

`@Html.Editor()` 和 `@Html.EditorFor()` 在执行其默认模板时使用名为 `htmlAttributes` 的特殊 `ViewDataDictionary` 条目。此行为可选择使用 `additionalViewData` 参数增强。键"htmlAttributes"区分大小写。键"htmlAttributes"的处理方式与传递到输入帮助程序的 `htmlAttributes` 对象(例如 `@Html.TextBox()`)的处理方式类似。

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

表达式名称

`asp-for` 属性值是 `ModelExpression`，并且是 lambda 表达式的右侧。因此，`asp-for="Property1"` 在生成的代码中变成 `m => m.Property1`，这也是无需使用 `Model` 前缀的原因。"@"字符可用作内联表达式的开头，并移到 `m.` 前面：

```
@{
       var joe = "Joe";
   }
   <input asp-for="@joe" />
```

生成以下 HTML：

```
<input type="text" id="joe" name="joe" value="Joe" />
```

使用集合属性时，`asp-for="CollectionProperty[23].Member"` 在 `i` 具有值 `23` 时生成与 `asp-for="CollectionProperty[i].Member"` 相同的名称。

在 ASP.NET Core MVC 计算 `ModelExpression` 的值时，它会检查多个源，包括 `ModelState`。以 `<input type="text" asp-for="@Name" />` 为例。计算出的 `value` 属性是第一个非 null 值，属于：

- 带有“Name”键的 `ModelState` 条目。
- `Model.Name` 表达式的结果。

导航子属性

还可使用视图模型的属性路径导航到子属性。设想一个包含子 `Address` 属性的更复杂的模型类。

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

在视图中，绑定到 `Address.AddressLine1`：

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

为 `Address.AddressLine1` 生成以下 HTML：

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

表达式名称和集合

包含 `Colors` 数组的模型示例：

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

操作方法：

```
public IActionResult Edit(int id, int colorIndex)
   {
       ViewData["Index"] = colorIndex;
       return View(GetPerson(id));
   }
```

以下 Razor 显示如何访问特定 `Color` 元素：

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

Views/Shared/EditorTemplates/String.cshtml 模板：

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

使用 `List<T>` 的示例：

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

以下 Razor 演示如何循环访问集合：

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

Views/Shared/EditorTemplates/ToDoItem.cshtml 模板：

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

应尽量在 `asp-for` 或 `Html.DisplayFor` 等效上下文中使用值 `foreach`。一般情况下，`for` 优于 `foreach`（如果情况允许使用的话），因为它不需要分配枚举器，但在 LINQ 表达式中评估索引器的成本高昂，应最大限度地减少使用它。

> **NOTE**
> 上述带有注释的示例代码演示如何将 lambda 表达式替换为 `@` 运算符来访问列表中的每个 `ToDoItem`。

文本区标记帮助程序

`Textarea Tag Helper` 标记帮助程序类似于输入标记帮助程序。

- 通过模型为 <textarea> 元素生成 `id` 和 `name` 属性以及数据验证属性。
- 提供强类型化。
- HTML 帮助程序替代项：`Html.TextAreaFor`

示例：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

生成以下 HTML：

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

标签标记帮助程序

- 在 元素中为表达式名称生成标签描述和 `for` 属性
- HTML 帮助程序替代项：`Html.LabelFor`。

`Label Tag Helper` 通过纯 HTML 标签元素提供如下优势：

- 可自动从 `Display` 属性中获取描述性标签值。预期的显示名称可能会随时间变化，`Display` 属性和标签标记帮助程序的组合会在其被使用的所有位置应用 `Display`。
- 源代码中的标记更少
- 模型属性的强类型化。

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

为 `<label>` 元素生成以下 HTML：

```
<label for="Email">Email Address</label>
```

标签标记帮助程序生成“Email”的 `for` 属性值，即与 `<input>` 元素关联的 ID。标记帮助程序生成一致的 `id` 和 `for` 元素，方便将其正确关联。本示例中的描述来自 `Display` 属性。如果模型不包含 `Display` 特性，描述将为表达式的属性名称。

验证标记帮助程序

有两个验证标记帮助程序。`Validation Message Tag Helper`（为模型中的单个属性显示验证消息）和 `Validation Summary Tag Helper`（显示验证错误的摘要）。`Input Tag Helper` 根据模型类的数据注释属性将 HTML5 客户端验证属性添加到输入元素中。同时在服务器上执行验证。验证标记帮助程序会在发生验证错误时显示这些错误消息。

验证消息标记帮助程序

- 将 HTML5 `data-valmsg-for="property"` 属性添加到 span 元素中，该元素会附加指定模型属性的输入字段中的验证错误消息。jQuery 会在发生客户端验证错误时在 `<span>` 元素中显示错误消息。
- 还会在服务器上执行验证。客户端可能已禁用 JavaScript，一些验证仅可在服务器端执行。
- HTML 帮助程序替代项：`Html.ValidationMessageFor`

`Validation Message Tag Helper` 与 HTML span 元素中的 `asp-validation-for` 属性配合使用。

```
<span asp-validation-for="Email"></span>
```

验证消息标记帮助程序会生成以下 HTML：

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

对于同一属性，通常在 `Input` 标记帮助程序后使用 `Validation Message Tag Helper`。这样做可在导致错误的输入附近显示所有验证错误消息。

> **NOTE**
>
> 必须拥有包含正确的 JavaScript 和 jQuery 脚本引用的视图才能执行客户端验证。有关详细信息，请参阅模型验证。

发生服务器端验证错误时（例如，禁用自定义服务器端验证或客户端验证时），MVC 会将该错误消息作为 `<span>` 元素的主体。

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

验证摘要标记帮助程序

- 针对具有 `asp-validation-summary` 属性的 `<div>` 元素
- HTML 帮助程序替代项：`@Html.ValidationSummary`

`Validation Summary Tag Helper` 用于显示验证消息的摘要。`asp-validation-summary` 属性值可以是以下任意值：

ASP-VALIDATION-SUMMARY	显示的验证消息
ValidationSummary.All	属性和模型级别
ValidationSummary.ModelOnly	模型
ValidationSummary.None	无

示例

在以下示例中，数据模型使用 `DataAnnotation` 属性修饰，在 `<input>` 元素中生成验证错误消息。验证标记帮助程序会在发生验证错误时显示错误消息：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

生成的 HTML（如果模型有效）：

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"> <br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

选择标记帮助程序

- 为模型属性生成 select 元素和关联的 option 元素。
- 具有 HTML 帮助程序替代项 `Html.DropDownListFor` 和 `Html.ListBoxFor`

`Select Tag Helper` `asp-for` 为 select 元素指定模型属性名称，`asp-items` 指定 option 元素。例如:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

示例:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

`Index` 方法初始化 `CountryViewModel`，设置选定的国家/地区并将其传递到 `Index` 视图。

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTTP POST `Index` 方法显示选定内容:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country +  " selected";
        return RedirectToAction("IndexSuccess", new { message = msg});
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

`Index` 视图:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

生成以下 HTML(选择"CA"时):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
   </form>
```

> **NOTE**
> 不建议将 `ViewBag` 或 `ViewData` 与选择标记帮助程序配合使用。视图模型在提供 MVC 元数据方面更可靠且通常更不容易出现问题。

`asp-for` 属性值是特殊情况，它不要求提供 `Model` 前缀，但其他标记帮助程序属性需要该前缀(例如 `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

枚举绑定

通常可方便地将 `<select>` 与 `enum` 属性配合使用并通过 `enum` 值生成 `SelectListItem` 元素。

示例:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

`GetEnumSelectList` 方法为枚举生成 `SelectList` 对象。

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

可使用 `Display` 属性修饰枚举器列表，以获取更丰富的 UI：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

生成以下 HTML：

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
  </form>
```

选项组

如果视图模型包含一个或多个 `SelectListGroup` 对象，则会生成 HTML `<optgroup>` 元素。

`CountryViewModelGroup` 将 `SelectListItem` 元素分组为“North America”组和“Europe”组：

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

两个组如下所示:

生成的 HTML：

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

多重选择

如果 `asp-for` 属性中指定的属性为 `IEnumerable`，选择标记帮助程序会自动生成 multiple = "multiple" 属性。例如，如果给定以下模型：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

及以下视图：

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

则会生成以下 HTML：

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

无选定内容

如果发现自己在多个页面中使用“未指定”选项，可创建模板用于消除重复的 HTML：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

Views/Shared/EditorTemplates/CountryViewModel.cshtml 模板：

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

添加 HTML <option> 元素并不局限于*无选定内容*用例。例如，以下视图和操作方法会生成与上述代码类似的 HTML：

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

根据当前的 `Country` 值选择正确的 `<option>` 元素（包含 `selected="selected"` 属性）。

```
<form method="post" action="/Home/IndexEmpty">
     <select id="Country" name="Country">
         <option value="">&lt;none&gt;</option>
         <option value="MX">Mexico</option>
         <option value="CA" selected="selected">Canada</option>
         <option value="US">USA</option>
     </select>
     <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

其他资源

- ASP.NET Core 中的标记帮助程序
- HTML Form 元素
- 请求验证令牌
- ASP.NET Core 中的模型绑定
- ASP.NET Core MVC 中的模型验证
- IAttributeAdapter 接口
- 本文档的代码片段

ASP.NET Core 表单中的标记帮助程序

2018/11/10 • • Edit Online

作者：Rick Anderson、Dave Paquette 和 Jerrie Pelser

本文档演示如何使用表单和表单中常用的 HTML 元素。HTML Form 元素提供 Web 应用用于向服务器回发数据的主要机制。本文档的大部分内容介绍标记帮助程序及其如何帮助高效创建可靠的 HTML 表单。建议在阅读本文档前先阅读标记帮助程序简介。

在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。如果存在 HTML 帮助程序替代项，文中会提到。

表单标记帮助程序

表单标记帮助程序：

- 为 MVC 控制器操作或命名路由生成 HTML <FORM> `action` 属性值
- 生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTML Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）
- 提供 `asp-route-<Parameter Name>` 属性，其中 `<Parameter Name>` 添加到路由值。`Html.BeginForm` 和 `Html.BeginRouteForm` 的 `routeValues` 参数提供类似的功能。
- 具有 HTML 帮助程序替代项 `Html.BeginForm` 和 `Html.BeginRouteForm`

示例：

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

上述表单标记帮助程序生成以下 HTML：

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

MVC 运行时通过表单标记帮助程序属性 `asp-controller` 和 `asp-action` 生成 `action` 属性值。表单标记帮助程序还会生成隐藏的请求验证令牌，防止跨站点请求伪造（在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时）。保护纯 HTML 表单免受跨站点请求伪造的影响很难，但表单标记帮助程序可提供此服务。

使用命名路由

`asp-route` 标记帮助程序属性还可为 HTML `action` 属性生成标记。具有名为 `register` 的路由的应用可将以下标记用于注册页：

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Views/Account 文件夹中的许多视图(在新建使用个人用户帐户身份验证的 Web 应用时生成)包含 asp-route-returnurl 属性:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **NOTE**
>
> 使用内置模板时, `returnUrl` 仅会在用户尝试访问授权资源, 但未验证身份或未获得授权的情况下自动填充。如果尝试执行未经授权的访问, 安全中间件会使用 `returnUrl` 集将用户重定向至登录页。

输入标记帮助程序

输入标记帮助程序将 HTML <input> 元素绑定到 Razor 视图中的模型表达式。

语法:

```
<input asp-for="<Expression Name>" />
```

输入标记帮助程序:

- 为 `asp-for` 属性中指定的表达式名称生成 `id` 和 `name` HTML 属性。`asp-for="Property1.Property2"` 与 `m => m.Property1.Property2` 相等。表达式的名称用于 `asp-for` 属性值。有关其他信息, 请参阅表达式名称部分。
- 根据模型类型和应用于模型属性的数据注释特性设置 HTML `type` 特性值
- 如果已经指定, 不会覆盖 HTML `type` 属性值
- 通过应用于模型属性的数据注释特性生成 HTML5 验证特性
- 具有与 `Html.TextBoxFor` 和 `Html.EditorFor` 重叠的 HTML 帮助程序功能。有关详细信息, 请参阅输入标记帮助程序的 **HTML 帮助程序替代项**部分。
- 提供强类型化。如果属性的名称更改, 但未更新标记帮助程序, 则会收到类似如下内容的错误:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
 'RegisterViewModel' does not contain a definition for 'Email' and no
 extension method 'Email' accepting a first argument of type 'RegisterViewModel'
 could be found (are you missing a using directive or an assembly reference?)
```

`Input` 标记帮助程序根据 .NET 类型设置 HTML `type` 属性。下表列出一些常见的 .NET 类型和生成的 HTML 类型(并未列出每个 .NET 类型)。

.NET 类型	输入类型
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single、Double	type="number"

下表显示输入标记帮助程序会映射到特定输入类型的一些常见数据注释属性（并未列出每个验证属性）：

特性	输入类型
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

上述代码生成以下 HTML：

```
  <form method="post" action="/Demo/RegisterInput">
      Email:
      <input type="email" data-val="true"
             data-val-email="The Email Address field is not a valid email address."
             data-val-required="The Email Address field is required."
             id="Email" name="Email" value="" /> <br>
      Password:
      <input type="password" data-val="true"
             data-val-required="The Password field is required."
             id="Password" name="Password" /><br>
      <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
  </form>
```

应用于 `Email` 和 `Password` 属性的数据注释在模型中生成元数据。输入标记帮助程序使用模型元数据并生成 HTML5 `data-val-*` 属性（请参阅模型验证）。这些属性描述要附加到输入字段的验证程序。这样可以提供非介入式 HTML5 和 jQuery 验证。非介入式属性具有格式 `data-val-rule="Error Message"`，其中规则是验证规则的名称（例如 `data-val-required`、`data-val-email`、`data-val-maxlength` 等）。如果在属性中提供错误消息，则该错误消息会作为 `data-val-rule` 属性的值显示。还有表单 `data-val-ruleName-argumentName="argumentValue"` 的属性，这些属性提供有关规则的其他详细信息，例如，`data-val-maxlength-max="1024"`。

输入标记帮助程序的 HTML 帮助程序替代项

`Html.TextBox`、`Html.TextBoxFor`、`Html.Editor` 和 `Html.EditorFor` 与输入标记帮助程序的功能存在重叠。输入标记帮助程序会自动设置 `type` 属性；而 `Html.TextBox` 和 `Html.TextBoxFor` 不会。`Html.Editor` 和 `Html.EditorFor` 处理集合、复杂对象和模板；而输入标记帮助程序不会。输入标记帮助程序、`Html.EditorFor` 和 `Html.TextBoxFor` 是强类型（使用 lambda 表达式）；而 `Html.TextBox` 和 `Html.Editor` 不是（使用表达式名称）。

HtmlAttributes

`@Html.Editor()` 和 `@Html.EditorFor()` 在执行其默认模板时使用名为 `htmlAttributes` 的特殊 `ViewDataDictionary` 条目。此行为可选择使用 `additionalViewData` 参数增强。键"htmlAttributes"区分大小写。键"htmlAttributes"的处理方式与传递到输入帮助程序的 `htmlAttributes` 对象（例如 `@Html.TextBox()`）的处理方式类似。

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

表达式名称

`asp-for` 属性值是 `ModelExpression`，并且是 lambda 表达式的右侧。因此，`asp-for="Property1"` 在生成的代码中变成 `m => m.Property1`，这也是无需使用 `Model` 前缀的原因。"@"字符可用作内联表达式的开头，并移到 `m.` 前面：

```
@{
        var joe = "Joe";
    }
    <input asp-for="@joe" />
```

生成以下 HTML：

```
<input type="text" id="joe" name="joe" value="Joe" />
```

使用集合属性时，`asp-for="CollectionProperty[23].Member"` 在 `i` 具有值 `23` 时生成与 `asp-for="CollectionProperty[i].Member"` 相同的名称。

在 ASP.NET Core MVC 计算 `ModelExpression` 的值时，它会检查多个源，包括 `ModelState`。以 `<input type="text" asp-for="@Name" />` 为例。计算出的 `value` 属性是第一个非 null 值，属于：

- 带有“Name”键的 `ModelState` 条目。
- `Model.Name` 表达式的结果。

导航子属性

还可使用视图模型的属性路径导航到子属性。设想一个包含子 `Address` 属性的更复杂的模型类。

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

在视图中，绑定到 `Address.AddressLine1`：

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

为 `Address.AddressLine1` 生成以下 HTML：

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

表达式名称和集合

包含 `Colors` 数组的模型示例：

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

操作方法:

```
public IActionResult Edit(int id, int colorIndex)
   {
       ViewData["Index"] = colorIndex;
       return View(GetPerson(id));
   }
```

以下 Razor 显示如何访问特定 `Color` 元素:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

Views/Shared/EditorTemplates/String.cshtml 模板:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

使用 `List<T>` 的示例:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

以下 Razor 演示如何循环访问集合:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

Views/Shared/EditorTemplates/ToDoItem.cshtml 模板：

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

应尽量在 `asp-for` 或 `Html.DisplayFor` 等效上下文中使用值 `foreach`。一般情况下，`for` 优于 `foreach`（如果情况允许使用的话），因为它不需要分配枚举器，但在 LINQ 表达式中评估索引器的成本高昂，应最大限度地减少使用它。

> **NOTE**
> 上述带有注释的示例代码演示如何将 lambda 表达式替换为 `@` 运算符来访问列表中的每个 `ToDoItem`。

文本区标记帮助程序

`Textarea Tag Helper` 标记帮助程序类似于输入标记帮助程序。

- 通过模型为 <textarea> 元素生成 `id` 和 `name` 属性以及数据验证属性。
- 提供强类型化。
- HTML 帮助程序替代项：`Html.TextAreaFor`

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

生成以下 HTML:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length
of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length
of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

标签标记帮助程序

- 在 元素中为表达式名称生成标签描述和 `for` 属性
- HTML 帮助程序替代项: `Html.LabelFor` 。

`Label Tag Helper` 通过纯 HTML 标签元素提供如下优势:

- 可自动从 `Display` 属性中获取描述性标签值。预期的显示名称可能会随时间变化, `Display` 属性和标签标记帮助程序的组合会在其被使用的所有位置应用 `Display` 。
- 源代码中的标记更少
- 模型属性的强类型化。

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

为 `<label>` 元素生成以下 HTML：

```
<label for="Email">Email Address</label>
```

标签标记帮助程序生成"Email"的 `for` 属性值，即与 `<input>` 元素关联的 ID。标记帮助程序生成一致的 `id` 和 `for` 元素，方便将其正确关联。本示例中的描述来自 `Display` 属性。如果模型不包含 `Display` 特性，描述将为表达式的属性名称。

验证标记帮助程序

有两个验证标记帮助程序。`Validation Message Tag Helper`（为模型中的单个属性显示验证消息）和 `Validation Summary Tag Helper`（显示验证错误的摘要）。`Input Tag Helper` 根据模型类的数据注释属性将 HTML5 客户端验证属性添加到输入元素中。同时在服务器上执行验证。验证标记帮助程序会在发生验证错误时显示这些错误消息。

验证消息标记帮助程序

- 将 HTML5 `data-valmsg-for="property"` 属性添加到 span 元素中，该元素会附加指定模型属性的输入字段中的验证错误消息。jQuery 会在发生客户端验证错误时在 `<span>` 元素中显示错误消息。
- 还会在服务器上执行验证。客户端可能已禁用 JavaScript，一些验证仅可在服务器端执行。
- HTML 帮助程序替代项：`Html.ValidationMessageFor`

`Validation Message Tag Helper` 与 HTML span 元素中的 `asp-validation-for` 属性配合使用。

```
<span asp-validation-for="Email"></span>
```

验证消息标记帮助程序会生成以下 HTML：

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

对于同一属性，通常在 `Input` 标记帮助程序后使用 `Validation Message Tag Helper`。这样做可在导致错误的输入附近显示所有验证错误消息。

> **NOTE**
> 必须拥有包含正确的 JavaScript 和 jQuery 脚本引用的视图才能执行客户端验证。有关详细信息，请参阅模型验证。

发生服务器端验证错误时（例如，禁用自定义服务器端验证或客户端验证时），MVC 会将该错误消息作为 `<span>` 元素的主体。

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

验证摘要标记帮助程序

- 针对具有 `asp-validation-summary` 属性的 `<div>` 元素
- HTML 帮助程序替代项：`@Html.ValidationSummary`

`Validation Summary Tag Helper` 用于显示验证消息的摘要。`asp-validation-summary` 属性值可以是以下任意值：

ASP-VALIDATION-SUMMARY	显示的验证消息
ValidationSummary.All	属性和模型级别
ValidationSummary.ModelOnly	模型
ValidationSummary.None	无

示例

在以下示例中，数据模型使用 `DataAnnotation` 属性修饰，在 `<input>` 元素中生成验证错误消息。验证标记帮助程序会在发生验证错误时显示错误消息：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

生成的 HTML（如果模型有效）：

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"> <br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

选择标记帮助程序

- 为模型属性生成 select 元素和关联的 option 元素。
- 具有 HTML 帮助程序替代项 `Html.DropDownListFor` 和 `Html.ListBoxFor`

`Select Tag Helper` `asp-for` 为 select 元素指定模型属性名称，`asp-items` 指定 option 元素。例如：

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

示例：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

`Index` 方法初始化 `CountryViewModel`，设置选定的国家/地区并将其传递到 `Index` 视图。

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTTP POST `Index` 方法显示选定内容：

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country +  " selected";
        return RedirectToAction("IndexSuccess", new { message = msg});
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

`Index` 视图：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

生成以下 HTML（选择"CA"时）：

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
   </form>
```

> **NOTE**
>
> 不建议将 `ViewBag` 或 `ViewData` 与选择标记帮助程序配合使用。视图模型在提供 MVC 元数据方面更可靠且通常更不容易出现问题。

`asp-for` 属性值是特殊情况，它不要求提供 `Model` 前缀，但其他标记帮助程序属性需要该前缀（例如 `asp-items`）

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

枚举绑定

通常可方便地将 `<select>` 与 `enum` 属性配合使用并通过 `enum` 值生成 `SelectListItem` 元素。

示例:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

`GetEnumSelectList` 方法为枚举生成 `SelectList` 对象。

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

可使用 `Display` 属性修饰枚举器列表，以获取更丰富的 UI：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

生成以下 HTML：

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
  </form>
```

选项组

如果视图模型包含一个或多个 `SelectListGroup` 对象，则会生成 HTML `<optgroup>` 元素。

`CountryViewModelGroup` 将 `SelectListItem` 元素分组为“North America”组和“Europe”组：

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

两个组如下所示:

生成的 HTML：

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

多重选择

如果 `asp-for` 属性中指定的属性为 `IEnumerable`，选择标记帮助程序会自动生成 multiple = "multiple" 属性。例如，如果给定以下模型：

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

及以下视图：

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

则会生成以下 HTML：

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

无选定内容

如果发现自己在多个页面中使用“未指定”选项，可创建模板用于消除重复的 HTML：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

Views/Shared/EditorTemplates/CountryViewModel.cshtml 模板：

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

添加 HTML <option> 元素并不局限于*无选定内容*用例。例如，以下视图和操作方法会生成与上述代码类似的 HTML：

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

根据当前的 `Country` 值选择正确的 `<option>` 元素（包含 `selected="selected"` 属性）。

```
<form method="post" action="/Home/IndexEmpty">
     <select id="Country" name="Country">
         <option value="">&lt;none&gt;</option>
         <option value="MX">Mexico</option>
         <option value="CA" selected="selected">Canada</option>
         <option value="US">USA</option>
     </select>
     <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

其他资源

- ASP.NET Core 中的标记帮助程序
- HTML Form 元素
- 请求验证令牌
- ASP.NET Core 中的模型绑定
- ASP.NET Core MVC 中的模型验证
- IAttributeAdapter 接口
- 本文档的代码片段

ASP.NET Core 中的布局

2018/10/29 • • Edit Online

作者：Steve Smith 和 Dave Brock

页面和视图经常共享视觉对象和程序元素。本文演示了以下内容的操作方法：

- 使用通用布局。
- 共享指令。
- 在呈现页面或视图之前运行通用代码。

本文档讨论了 ASP.NET Core MVC 的两种不同方法的布局：Razor Pages 和带有视图的控制器。在本主题中，差异很小：

- Razor Pages 位于“页面”文件夹。
- 具有视图的控制器使用视图的“视图”文件夹。

什么是布局

大多数 Web 应用都有一个通用布局，可在页面间切换时为用户提供一致体验。该布局通常包括应用标头、导航或菜单元素以及页脚等常见的用户界面元素。

应用中的许多页面也经常使用脚本和样式表等常用的 HTML 结构。所有这些共享元素均可在布局文件中进行定义，应用内使用的任何视图随后均可引用此文件。布局可减少视图中的重复代码，帮助它们遵循不要自我重复 (DRY) 原则。

按照约定，ASP.NET Core 应用的默认布局名为 _Layout.cshtml。使用模板创建的新 ASP.NET Core 项目的布局文件：

- Razor Pages：Pages/Shared/_Layout.cshtml

- 包含视图的控制器：Views/Shared/_Layout.cshtml

布局定义应用中的视图的最高级别模板。应用不需要布局。应用可以定义多个布局，并且不同的视图指定不同的布局。

下面的代码演示了使用控制器和视图创建的项目模板的布局文件：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebApplication1</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-
test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-brand">WebApplication1</a>
            </div>
            <div class="navbar-collapse collapse">
```

```
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Contact">Contact</a></li>
                </ul>
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 - WebApplication1</p>
        </footer>
    </div>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha384-tsQFqpEReu7ZLhBV2VZlAu7zcOV+rXbYlF2cqB8txI/8aZajjp4Bqd+V6D5IgvKT">
        </script>
        <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

指定布局

Razor 视图具有 `Layout` 属性。单个视图通过设置此属性来指定布局：

```
@{
    Layout = "_Layout";
}
```

指定的布局可以使用完整路径（例如 /Pages/Shared/_Layout.cshtml 或 /Views/Shared/_Layout.cshtml）或部分名称（示例：`_Layout`）。如果提供部分名称，Razor 视图引擎将使用其标准发现过程来搜索布局文件。首先搜索处理程序方法（或控制器）所在的文件夹，然后搜索 Shared 文件夹。此发现过程与用于发现分部视图的过程相同。

默认情况下，每个布局必须调用 `RenderBody`。无论在何处调用 `RenderBody`，都会呈现视图的内容。

部分

布局可以通过调用 `RenderSection` 来选择引用一个或多个节。节提供一种方法来组织某些页面元素应当放置的位置。每次调用 `RenderSection` 时都可指定该部分是必需还是可选：

```
@section Scripts {
    @RenderSection("Scripts", required: false)
}
```

如果找不到所需的节，将引发异常。单个视图使用 `@section` Razor 语法指定要在节中呈现的内容。如果某个页面或视图定义了一个部分，则必须呈现该部分（否则将发生错误）。

Razor Pages 视图中的示例 `@section` 定义：

```
@section Scripts {
     <script type="text/javascript" src="/scripts/main.js"></script>
}
```

在上面的代码中，scripts / main.js 添加到了页面或视图的 `scripts` 部分。相同应用中的其他页面或视图可能不需要此脚本且不会定义脚本部分。

以下标记使用部分标记帮助程序来呈现 _ValidationScriptsPartial.cshtml：

```
@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

前面的标记由基架标识生成。

在页面或视图中定义的部分仅在其即时布局页面中可用。不能从部分、视图组件或视图系统的其他部分引用它们。

忽略节

默认情况下，必须由布局页面呈现内容页中的正文和所有节。Razor 视图引擎通过跟踪是否已呈现正文和每个节来强制执行此操作。

要让视图引擎忽略正文或节，请调用 `IgnoreBody` 和 `IgnoreSection` 方法。

必须呈现或忽略 Razor 页面中的正文和每个节。

导入共享指令

视图和页面可以使用 Razor 指令来导入命名空间并使用依赖项注入。由多个视图共享的指令可以在通用 _ViewImports.cshtml 文件中进行指定。`_ViewImports` 文件支持以下指令：

- `@addTagHelper`
- `@removeTagHelper`
- `@tagHelperPrefix`
- `@using`
- `@model`
- `@inherits`
- `@inject`

该文件不支持函数和节定义等其他 Razor 功能。

示例 `_ViewImports.cshtml` 文件：

```
@using WebApplication1
@using WebApplication1.Models
@using WebApplication1.Models.AccountViewModels
@using WebApplication1.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

ASP.NET Core MVC 应用的 _ViewImports.cshtml 文件通常放在"页面"(或"视图")文件夹中。_ViewImports.cshtml 文件可以放在任何文件夹中，在这种情况下，它只会应用于该文件夹及其子文件夹中的页面或视图。从根级别开始处理 `_ViewImports` 文件，然后处理在页面或视图本身的位置之前的每个文件夹。可以在文件夹级别覆盖根级别指定的 `_ViewImports` 设置。

例如，假设：

- 根级别 _ViewImports.cshtml 文件包含 `@model MyModel1` 和 `@addTagHelper *, MyTagHelper1`。
- 子文件夹 _ViewImports.cshtml 文件包含 `@model MyModel2` 和 `@addTagHelper *, MyTagHelper2`。

子文件夹中的页面和视图将有权访问标记帮助程序和 `MyModel2` 模型。

如果在文件层次结构中找到多个 _ViewImports.cshtml 文件，指令的合并行为是：

- `@addTagHelper``@removeTagHelper`：按顺序全部运行
- `@tagHelperPrefix`：最接近视图的文件会替代任何其他文件
- `@model`：最接近视图的文件会替代任何其他文件
- `@inherits`：最接近视图的文件会替代任何其他文件
- `@using`：全部包括在内；忽略重复项
- `@inject`：针对每个属性，最接近视图的属性会替代具有相同属性名的任何其他属性

在呈现每个视图之前运行代码

需要在每个视图或页面之前运行的代码应置于 _ViewStart.cshtml 文件中。按照约定，_ViewStart.cshtml 文件位于"页面"(或"视图")文件夹。在呈现每个完整视图(不是布局，也不是分部视图)之前运行 _ViewStart.cshtml 中列出的语句。与 ViewImports.cshtml 一样，_ViewStart.cshtml 也是分层结构。如果在视图或页面文件夹中定义了 _ViewStart.cshtml 文件，则它将在页面(或视图)的根目录中定义的文件之后运行)文件夹(如果有)。

一个示例 _ViewStart.cshtml 文件：

```
@{
    Layout = "_Layout";
}
```

上述文件指定所有视图都将使用 _Layout.cshtml 布局。

_ViewStart.cshtml 和 _ViewImports.cshtml 通常不置于 /Pages/Shared(或 /Views/Shared)文件夹中。这些应用级别版本的文件应直接放置在 /Pages(或 /Views)文件夹中。

ASP.NET Core 中的静态文件

2019/1/8 • • Edit Online

作者：Rick Anderson 和 Scott Addie

静态文件（如 HTML、CSS、图像和 JavaScript）是 ASP.NET Core 应用直接提供给客户端的资产。需要进行一些配置才能提供这些文件。

查看或下载示例代码（如何下载）

提供静态文件

静态文件存储在项目的 Web 根目录中。默认目录是 <content_root>/wwwroot，但可通过 UseWebRoot 方法更改目录。有关详细信息，请参阅内容根目录和 Web 根目录。

应用的 Web 主机必须识别内容根目录。

采用 `WebHost.CreateDefaultBuilder` 方法可将内容根目录设置为当前目录：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

调用 `Program.Main` 中的 UseContentRoot 将内容根目录设为当前目录：

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .UseIISIntegration()
            .UseStartup<Startup>()
            .UseApplicationInsights()
            .Build();

        host.Run();
    }
}
```

可通过 Web 根目录的相关路径访问静态文件。例如，Web 应用程序项目模板包含 wwwroot 文件夹中的多个文件夹：

- **wwwroot**
  - **css**
  - **images**

- **js**

用于访问 images 子文件夹中的文件的 URI 格式为 http://<server_address>/images/<image_file_name>。例如，*http://localhost:9189/images/banner3.svg*。

如果以 .NET Framework 为目标，请将 Microsoft.AspNetCore.StaticFiles 包添加到项目。如果以 .NET Core 为目标，Microsoft.AspNetCore.App 元包将包括此包。

如果以 .NET Framework 为目标，请将 Microsoft.AspNetCore.StaticFiles 包添加到项目。如果以 .NET Core 为目标，请将 Microsoft.AspNetCore.All 元包加入此包。

将 Microsoft.AspNetCore.StaticFiles 包添加到项目。

配置提供静态文件的中间件。

提供 Web 根目录内的文件

调用 `Startup.Configure` 中的 UseStaticFiles 方法：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

无参数 `UseStaticFiles` 方法重载将 Web 根目录中的文件标记为可用。以下标记引用 wwwroot/images/banner1.svg：

```
<img src="~/images/banner1.svg" alt="ASP.NET" class="img-responsive" />
```

在上面的代码中，波形符 `~/` 指向 Web 根目录。有关详细信息，请参阅 Web 根目录。

提供 Web 根目录外的文件

考虑一个目录层次结构，其中要提供的静态文件位于 Web 根目录之外：

- **wwwroot**
  - **css**
  - **images**
  - **js**
- **MyStaticFiles**
  - **images**
    - banner1.svg

按如下方式配置静态文件中间件后，请求可访问 banner1.svg 文件：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "MyStaticFiles")),
        RequestPath = "/StaticFiles"
    });
}
```

在前面的代码中，MyStaticFiles 目录层次结构通过 StaticFiles URI 段公开。请求

http://<server_address>/StaticFiles/images/banner1.svg 提供 banner1.svg 文件。

以下标记引用 MyStaticFiles/images/banner1.svg：

```
<img src="~/StaticFiles/images/banner1.svg" alt="ASP.NET" class="img-responsive" />
```

设置 HTTP 响应标头

StaticFileOptions 对象可用于设置 HTTP 响应标头。除配置从 Web 根目录提供静态文件外，以下代码还设置 `Cache-Control` 标头：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    var cachePeriod = env.IsDevelopment() ? "600" : "604800";
    app.UseStaticFiles(new StaticFileOptions
    {
        OnPrepareResponse = ctx =>
        {
            // Requires the following import:
            // using Microsoft.AspNetCore.Http;
            ctx.Context.Response.Headers.Append("Cache-Control", $"public, max-age={cachePeriod}");
        }
    });
}
```

HeaderDictionaryExtensions.Append 方法存在于 Microsoft.AspNetCore.Http 包中。

在开发环境中可公开缓存这些文件 10 分钟（600 秒）：

静态文件授权

静态文件中间件不提供授权检查。可公开访问由静态文件中间件提供的任何文件，包括 wwwroot 下的文件。根据授权提供文件：

- 将文件存储在 wwwroot 和静态文件中间件可访问的任何目录之外。
- 通过有授权的操作方法提供文件。返回 FileResult 对象：

```
[Authorize]
public IActionResult BannerImage()
{
    var file = Path.Combine(Directory.GetCurrentDirectory(),
                            "MyStaticFiles", "images", "banner1.svg");

    return PhysicalFile(file, "image/svg+xml");
}
```

启用目录浏览

通过目录浏览，Web 应用的用户可查看目录列表和指定目录中的文件。出于安全考虑，目录浏览默认处于禁用状态（请参阅注意事项）。调用 `Startup.Configure` 中的 UseDirectoryBrowser 方法来启用目录浏览：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });
}
```

调用 `Startup.ConfigureServices` 中的AddDirectoryBrowser 方法来添加所需服务：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDirectoryBrowser();
}
```

上述代码允许使用 URL http://<server_address>/MyImages 浏览 wwwroot/images 文件夹的目录，并链接到每个文件和文件夹：

有关启用浏览时的安全风险，请参阅注意事项。

请注意以下示例中的两个 `UseStaticFiles` 调用。第一个调用提供 wwwroot 文件夹中的静态文件。第二个调用使用 URL http://<server_address>/MyImages 浏览 wwwroot/images 文件夹的目录：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });
}
```

提供默认文档

设置默认主页为访问者访问网站时提供了逻辑起点。若要在用户不完全限定 URI 的情况下提供默认页面，请调用 `Startup.Configure` 中的 UseDefaultFiles 方法：

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

> **IMPORTANT**
> 要提供默认文件，必须在 `UseStaticFiles` 前调用 `UseDefaultFiles`。`UseDefaultFiles` 实际上用于重写 URL，不提供文件。通过 `UseStaticFiles` 启用静态文件中间件来提供文件。

使用 `UseDefaultFiles` 请求文件夹搜索：

- default.htm
- default.html
- index.htm
- index.html

将请求视为完全限定 URI，提供在列表中找到的第一个文件。浏览器 URL 继续反映请求的 URI。

以下代码将默认文件名更改为 mydefault.html：

```
public void Configure(IApplicationBuilder app)
{
    // Serve my app-specific default file, if present.
    DefaultFilesOptions options = new DefaultFilesOptions();
    options.DefaultFileNames.Clear();
    options.DefaultFileNames.Add("mydefault.html");
    app.UseDefaultFiles(options);
    app.UseStaticFiles();
}
```

UseFileServer

UseFileServer 结合了 `UseStaticFiles`、`UseDefaultFiles` 和 `UseDirectoryBrowser` 的功能。

以下代码提供静态文件和默认文件。未启用目录浏览。

```
app.UseFileServer();
```

以下代码通过启用目录浏览基于无参数重载进行构建：

```
app.UseFileServer(enableDirectoryBrowsing: true);
```

考虑以下目录层次结构：

- **wwwroot**
  - **css**
  - **images**
  - **js**
- **MyStaticFiles**
  - **images**
    - banner1.svg
  - default.html

以下代码启用静态文件、默认文件和及 `MyStaticFiles` 的目录浏览：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseFileServer(new FileServerOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "MyStaticFiles")),
        RequestPath = "/StaticFiles",
        EnableDirectoryBrowsing = true
    });
}
```

`EnableDirectoryBrowsing` 属性值为 `true` 时必须调用 `AddDirectoryBrowser`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDirectoryBrowser();
}
```

使用文件层次结构和前面的代码，URL 解析如下：

URI	响应
http://<server_address>/StaticFiles/images/banner1.svg	MyStaticFiles/images/banner1.svg
http://<server_address>/StaticFiles	MyStaticFiles/default.html

如果 MyStaticFiles 目录中不存在默认命名文件，则 http://<server_address>/StaticFiles 返回包含可单击链接的

目录列表：

NOTE

`UseDefaultFiles` 和 `UseDirectoryBrowser` 使用不带尾部反斜杠的 URL http://<server_address>/StaticFiles 触发客户端并重定向到 http://<server_address>/StaticFiles/。注意添加尾部反斜杠。无尾部反斜杠，文档中的相对 URL 被视为无效。

FileExtensionContentTypeProvider

FileExtensionContentTypeProvider 类包含 `Mappings` 属性，用作文件扩展名到 MIME 内容类型的映射。在以下示例中，多个文件扩展名注册到了已知的 MIME 类型。替换了 .rtf 扩展名，删除了 .mp4。

```
public void Configure(IApplicationBuilder app)
{
    // Set up custom content types - associating file extension to MIME type
    var provider = new FileExtensionContentTypeProvider();
    // Add new mappings
    provider.Mappings[".myapp"] = "application/x-msdownload";
    provider.Mappings[".htm3"] = "text/html";
    provider.Mappings[".image"] = "image/png";
    // Replace an existing mapping
    provider.Mappings[".rtf"] = "application/x-msdownload";
    // Remove MP4 videos.
    provider.Mappings.Remove(".mp4");

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages",
        ContentTypeProvider = provider
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });
}
```

请参阅 MIME 内容类型。

非标准内容类型

静态文件中间件可理解近 400 种已知文件内容类型。如果用户请求文件类型未知的文件，则静态文件中间件将请求传递给管道中的下一个中间件。如果没有中间件处理请求，则返回“404 未找到”响应。如果启用了目录浏览，则在目录列表中会显示该文件的链接。

以下代码提供未知类型，并以图像形式呈现未知文件：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(new StaticFileOptions
    {
        ServeUnknownFileTypes = true,
        DefaultContentType = "image/png"
    });
}
```

使用前面的代码，请求的文件含未知内容类型时，以图像形式返回请求。

> **WARNING**
>
> 启用 ServeUnknownFileTypes 存在安全风险。它默认处于禁用状态，不建议使用。FileExtensionContentTypeProvider 提供了更安全的替代方法来提供含非标准扩展名的文件。

注意事项

> **WARNING**
>
> `UseDirectoryBrowser` 和 `UseStaticFiles` 可能会泄漏机密。强烈建议在生产中禁用目录浏览。请仔细查看 `UseStaticFiles` 或 `UseDirectoryBrowser` 启用了哪些目录。整个目录及其子目录均可公开访问。将适合公开的文件存储在专用目录中，如 <content_root>/wwwroot。将这些文件与 MVC 视图、Razor 页面（仅限 2.x）和配置文件等分开

- 使用 `UseDirectoryBrowser` 和 `UseStaticFiles` 公开的内容的 URL 受大小写和基础文件系统字符限制的影响。例如，Windows 不区分大小写 — macOS 和 Linux 却要区分。
- 托管于 IIS 中的 ASP.NET Core 应用使用 ASP.NET Core 模块将所有请求转发到应用，包括静态文件请求。未使用 IIS 静态文件处理程序。在模块处理请求前，处理程序没有机会处理请求。
- 在 IIS Manager 中完成以下步骤，删除服务器或网站级别的 IIS 静态文件处理程序：
  1. 转到“模块”功能。
  2. 在列表中选择 StaticFileModule。
  3. 单击“操作”侧栏中的“删除”。

> **WARNING**
>
> 如果启用了 IIS 静态文件处理程序且 ASP.NET Core 模块配置不正确，则会提供静态文件。例如，如果未部署 web.config 文件，则会发生这种情况。

- 将代码文件（包括 .cs 和 .cshtml）放在应用项目的 Web 根目录之外。这样就在应用的客户端内容和基于服务器的代码间创建了逻辑分隔。可以防止服务器端代码泄漏。

其他资源

- 中间件

- ASP.NET Core 简介

ASP.NET Core 中的模型绑定

2018/11/21 • • Edit Online

作者：Rachel Appel

模型绑定简介

ASP.NET Core MVC 中的模型绑定将 HTTP 请求中的数据映射到操作方法参数。这些参数可能是简单类型的参数，如字符串、整数或浮点数，也可能是复杂类型的参数。这是 MVC 的一项强大功能，因为不管数据的大小和复杂性，将传入数据映射到对应位置都是经常重复的方案。MVC 通过将绑定抽象出来解决了这一问题，使开发者不必在每个应用中重写同一代码的稍微不同版本。向类型转换器代码写入自己的文本不仅繁琐乏味，而且容易出错。

模型绑定的工作原理

当 MVC 收到 HTTP 请求时，它会将此请求路由到控制器的特定操作方法。它基于路由数据中的内容决定要运行的操作方法，然后将 HTTP 请求中的值绑定到该操作方法的参数。以下列 URL 为例：

`http://contoso.com/movies/edit/2`

由于路由模板为 `{controller=Home}/{action=Index}/{id?}`，因此，`movies/edit/2` 将路由到 `Movies` 控制器及其 `Edit` 操作方法。此外，它还接受名为 `id` 的可选参数。操作方法的代码应如下所示：

```
public IActionResult Edit(int? id)
```

注意：URL 路由中的字符串不区分大小写。

MVC 将尝试按名称将请求数据绑定到操作参数。MVC 将使用参数名称和其公共可设置属性的名称查找每个参数的值。在以上示例中，唯一的操作参数名为 `id`，MVC 会将此参数绑定到路由值中具有相同名称的值。除路由值外，MVC 还会绑定来自请求各个部分的数据，并按一定顺序执行此操作。下面是一个数据源列表（按模型绑定查看的顺序排列）：

1. `Form values`：这些是使用 POST 方法进入 HTTP 请求的窗体值。（包括 jQuery POST 请求）。
2. `Route values`：路由提供的路由值集
3. `Query strings`：URI 的查询字符串部分。

注意：窗体值、路由数据和查询字符串均存储为名称/值对。

由于模型绑定请求名为 `id` 的键，而窗体值中没有任何名为 `id` 的键，因此，它将转到路由值以查找该键。在本示例中，该键是一个匹配项。发生绑定，并且值被转换为整数 2。使用 Edit(string id) 的同一请求将转换为字符串“2”。

到目前为止，示例使用的都是简单类型。在 MVC 中，简单类型是任何 .NET 基元类型或包含字符串类型转换器的类型。如果操作方法的参数是一个属性为简单类型和复杂类型的类，如 `Movie` 类型，则 MVC 的模型绑定仍可对其进行妥善处理。它使用反射和递归来遍历查找匹配项的复杂类型的属性。模型绑定查找模式 parameter_name.property_name 以将值绑定到属性 。如果未找到此窗体的匹配值，它将尝试仅使用属性名称进行绑定。对于诸如 `Collection` 类型的类型，模型绑定将查找 parameter_name[index] 或仅 [index] 的匹配项。对于 `Dictionary` 类型，模型绑定的处理方法与之类似，即请求 parameter_name[key] 或仅 [key]（只要键是简单类型）。受支持的键匹配字段名称 HTML 和针对相同模型类型生成的标记帮助程序。这样就可以启用往返值，以便用户仍能方便地输入内容来

填充窗体字段，例如当创建或编辑的绑定数据未通过验证时。

若要实现模型绑定，该类必须具有要绑定的公共默认构造函数和公共可写属性。发生模型绑定时，在使用公共默认构造函数对类进行实例化后才可设置属性。

绑定参数时，模型绑定将停止查找具有该名称的值并向前移动以绑定下一个参数。否则，默认模型绑定行为将根据参数的类型将参数设置为其默认值：

- `T[]`：除 `byte[]` 类型的数组外，绑定将 `T[]` 类型的参数设置为 `Array.Empty<T>()`。`byte[]` 类型的数组设置为 `null`。
- 引用类型：绑定通过默认构造函数创建类的实例而不设置属性。但是，模型绑定将 `string` 参数设置为 `null`。
- 可以为 null 的类型：可以为 null 的类型设置为 `null`。在上面的示例中，模型绑定将 `id` 设置为 `null`，因为其类型为 `int?`。
- 值类型：`T` 类型的不可以为 null 的值类型设置为 `default(T)`。例如，模型绑定将参数 `int id` 设置为 0。请考虑使用模型验证或可以为 null 的类型，而不是依赖于默认值。

如果绑定失败，MVC 不会引发错误。接受用户输入的每个操作均应检查 `ModelState.IsValid` 属性。

注意：控制器的 `ModelState` 属性中的每个输入均为包含 `Errors` 属性的 `ModelStateEntry`。很少需要自行查询此集合。请改用 `ModelState.IsValid`。

此外，MVC 在执行模型绑定时必须考虑一些特殊数据类型：

- `IFormFile`、`IEnumerable<IFormFile>`：作为 HTTP 请求一部分的一个或多个已上传文件。
- `CancellationToken`：用于取消异步控制器中的活动。

这些类型可绑定到操作参数或类类型的属性。

模型绑定完成后，将发生验证。对于绝大多数开发方案，默认模型绑定效果极佳。它还可以扩展，因此如果你有特殊需求，则可自定义内置行为。

通过特性自定义模型绑定行为

MVC 包含多种特性，可用于将其默认模型绑定行为定向到不同的源。例如，可以使用 `[BindRequired]` 或 `[BindNever]` 特性指定属性是否需要绑定，或绑定是否根本不应发生。或者，可以替代默认数据源，并指定模型绑定器的数据源。下面是模型绑定特性的列表：

- `[BindRequired]`：如果无法发生绑定，此特性将添加模型状态错误。
- `[BindNever]`：指示模型绑定器从不绑定到此参数。
- `[FromHeader]`、`[FromQuery]`、`[FromRoute]`、`[FromForm]`：使用这些特性指定要应用的确切绑定源。
- `[FromServices]`：此特性使用依赖关系注入绑定服务中的参数。
- `[FromBody]`：使用配置的格式化程序绑定请求正文中的数据。基于请求的内容类型，选择格式化程序。
- `[ModelBinder]`：用于替代默认模型绑定器、绑定源和名称。

需要替代模型绑定的默认行为时，特性是非常有用的工具。

全局自定义模型绑定和验证

由 ModelMetadata 驱动模型绑定和验证系统的行为，该行为描述：

- 如何绑定模型。
- 如何验证类型及其属性。

通过向 MvcOptions.ModelMetadataDetailsProviders 添加详细信息提供程序，可以全局配置系统行为的各个方面。MVC 有一些内置的详细信息提供程序，可通过它们配置禁用模型绑定或验证某些类型等行为。

若要禁用对特定类型的所有模型的模型绑定，请在 `Startup.ConfigureServices` 中添加 ExcludeBindingMetadataProvider。例如，禁用对 `System.Version` 类型的所有模型的模型绑定：

```
services.AddMvc().AddMvcOptions(options =>
    options.ModelMetadataDetailsProviders.Add(
        new ExcludeBindingMetadataProvider(typeof(System.Version))));
```

要禁用对特定类型的属性的验证，请在 `Startup.ConfigureServices` 中添加 SuppressChildValidationMetadataProvider。例如，禁用对 `System.Guid` 类型的属性的验证：

```
services.AddMvc().AddMvcOptions(options =>
    options.ModelMetadataDetailsProviders.Add(
        new SuppressChildValidationMetadataProvider(typeof(System.Guid))));
```

绑定请求正文中的带格式数据

请求数据可以有各种格式，包括 JSON、XML 和许多其他格式。使用 [FromBody] 特性指示要将参数绑定到请求正文中的数据时，MVC 会使用一组已配置的格式化程序基于请求数据的内容类型对请求数据进行处理。默认情况下，MVC 包括用于处理 JSON 数据的 `JsonInputFormatter` 类，但你可以添加用于处理 XML 和其他自定义格式的其他格式化程序。

> **NOTE**
>
> 对于用 `[FromBody]` 修饰的每个操作，最多可以有一个参数。ASP.NET Core MVC 运行时向格式化程序委托读取请求流的责任。读取参数的请求流后，通常不能为绑定其他 `[FromBody]` 参数而再次读取请求流。

> **NOTE**
>
> `JsonInputFormatter` 为默认格式化程序且基于 Json.NET。

除非有特性应用于 ASP.NET Core，否则它将基于 Content-Type 标头和参数类型来选择输入格式化程序。如果想要使用 XML 或其他格式，则必须在 Startup.cs 文件中配置该格式，但可能必须先使用 NuGet 获取对 `Microsoft.AspNetCore.Mvc.Formatters.Xml` 的引用。启动代码应如下所示：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddXmlSerializerFormatters();
   }
```

Startup.cs 文件中的代码包含具有 `services` 参数的 `ConfigureServices` 方法，此方法可用于为 ASP.NET Core 应用生成服务。在此示例中，我们将添加 XML 格式化程序作为 MVC 将为此应用提供的服务。通过传递给 `AddMvc` 方法的 `options` 参数，可在应用启动后添加和管理筛选器、格式化程序

和其他 MVC 系统选项。然后，将 `Consumes` 特性应用于控制器类或操作方法，以使用所需格式。

自定义模型绑定

可通过编写自己的自定义模型绑定器来扩展模型绑定。详细了解自定义模型绑定。

ASP.NET Core MVC 中的模型验证

2018/11/14 • • Edit Online

作者：Rachel Appel

模型验证简介

在将数据存储到数据库之前，应用必须先验证数据。必须检查数据是否存在潜在的安全威胁，确保数据已设置适当的类型和大小格式，并且必须符合相关规则。实施验证的过程可能有些单调乏味，但却必不可少。在 MVC 中，验证发生在客户端和服务器上。

幸运的是，.NET 已将验证抽象化为验证属性。这些属性包含验证代码，从而减少了所需编写的代码量。

在 ASP.NET Core 2.2 及更高版本中，如果能够确定给定模型关系图不需要进行验证，ASP.NET Core 运行时便会简化（跳过）验证。验证无法或没有关联任何验证程序的模型时，跳过验证可能会显著提升性能。已跳过的验证包括诸如基元集合（`byte[]`、`string[]`、`Dictionary<string, string>` 等）之类的对象，或没有任何验证程序的复杂对象关系图。

查看或下载 GitHub 中的示例。

验证属性

验证属性用于配置模型验证，因此，在概念上类似于数据库表中字段上的验证。它包括诸如分配数据类型或必填字段之类的约束。其他类型的验证包括将向数据应用模式以强制实施业务规则，比如信用卡、电话号码或电子邮件地址。验证属性更易使用，并使这些要求的实施变得更简单。

验证特性在属性级别指定：

```
[Required]
public string MyProperty { get; set; }
```

下面是一个应用的已批注 `Movie` 模型，该应用用于存储电影和电视节目的相关信息。大多数属性都是必需属性，多个字符串属性具有长度要求。此外，还有一个针对 `Price` 属性设置的从 0 到 $999.99 的数值范围限制，以及一个自定义验证特性。

```
public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; }

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    [Required]
    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

通过读取整个模型即可显示有关此应用的数据的规则，从而使代码维护变得更轻松。下面是几个常用的内置验证属性：

- `[CreditCard]`：验证属性是否具有信用卡格式。
- `[Compare]`：验证某个模型中的两个属性是否匹配。
- `[EmailAddress]`：验证属性是否具有电子邮件格式。
- `[Phone]`：验证属性是否具有电话格式。
- `[Range]`：验证属性值是否落在给定范围内。
- `[RegularExpression]`：验证数据是否与指定的正则表达式匹配。
- `[Required]`：将属性设置为必需属性。
- `[StringLength]`：验证字符串属性是否最多具有给定的最大长度。
- `[Url]`：验证属性是否具有 URL 格式。

MVC 支持从 `ValidationAttribute` 派生的所有用于验证的属性。在 System.ComponentModel.DataAnnotations 命名空间中可找到许多有用的验证属性。

在某些情况下，内置属性可能无法提供所需的功能。这时，就可以通过从 `ValidationAttribute` 派生或将模型更改为实现 `IValidatableObject`，来创建自定义验证属性。

必需属性的使用说明

从本质上来说，需要不可以为 null 的值类型（如 `decimal`、`int`、`float` 和 `DateTime`），但不需要 `Required` 特性。应用不会对标记为 `Required` 的不可为 null 的类型执行任何服务器端验证检查。

对于不可为 null 的类型，MVC 模型绑定（与验证和验证属性无关）会拒绝包含缺失值或空白的表单域提交。如果目标属性上缺少 `BindRequired` 特性，模型绑定会忽略不可为 null 的类型的缺失数据，导致传入表单数据中缺少表单域。

BindRequired 特性（另请参阅 ASP.NET Core 中的模型绑定）可用于确保表单数据完整。当应用于

某个属性时，模型绑定系统要求该属性具有值。当应用于某个类型时，模型绑定系统要求该类型的所有属性都具有值。

使用 Nullable<T> 类型（例如，`decimal?` 或 `System.Nullable<decimal>`）并将其标记为 `Required` 时，将执行服务器端验证检查，就像该属性是标准的可以为 null 的类型（例如，`string`）一样。

客户端验证要求与标记为 `Required` 的模型属性对应的表单域以及未标记为 `Required` 的不可为 null 的类型属性具有值。`Required` 可用于控制客户端验证错误消息。

模型状态

模型状态表示已提交的 HTML 表单值中的验证错误。

MVC 将继续验证字段，直至达到错误数上限（默认为 200 个）。可以使用 `Startup.ConfigureServices` 中的以下代码配置该数字：

```
}
```

处理模型状态错误

在执行控制器操作之前进行模型验证。该操作负责检查 `ModelState.IsValid` 并做出相应响应。在许多情况下，正确的反应是返回错误响应，理想状况下会详细说明模型验证失败的原因。

如果在使用 `[ApiController]` 属性的 web API 控制器中，`ModelState.IsValid` 的计算结果为 `false`，将返回包含问题详细信息的自动 HTTP 400 响应。有关详细信息，请参阅自动 HTTP 400 响应。

某些应用会选择遵循标准约定来处理模型验证错误，在这种情况下，可以在筛选器中实现此类策略。应测试操作在有效模型状态和无效模型状态下的行为方式。

手动验证

完成模型绑定和验证后，可能需要重复其中的某些步骤。例如，用户可能在应输入整数的字段中输入了文本，或者你可能需要计算模型的某个属性的值。

你可能需要手动运行验证。为此，请调用 `TryValidateModel` 方法，如下所示：

```
TryValidateModel(movie);
```

自定义验证

验证属性适用于大多数验证需求。但是，某些验证规则特定于你的业务。你的规则可能不是常见的数据验证技术，比如确保字段是必填字段或符合一系列值。在这些情况下，自定义验证属性是一种不错的解决方案。在 MVC 中创建你自己的自定义验证属性很简单。只需从 `ValidationAttribute` 继承并重写 `IsValid` 方法。`IsValid` 方法采用两个参数，第一个是名为 *value* 的对象，第二个是名为 *validationContext* 的 `ValidationContext` 对象。*Value* 引用自定义验证程序要验证的字段中的实际值。

在下面的示例中，一项业务规则规定，用户不能将 1960 年以后发行的电影的流派设置为 *Classic*。`[ClassicMovie]` 属性会先检查流派，如果是经典流派，则查看发行日期是否晚于 1960 年。如果晚于 1960 年，则验证失败。此属性采用一个表示年份的整数参数，可用于验证数据。可以在该属性的构造函数中捕获该参数的值，如下所示：

```
public class ClassicMovieAttribute : ValidationAttribute, IClientModelValidator
{
    private int _year;

    public ClassicMovieAttribute(int year)
    {
        _year = year;
    }

    protected override ValidationResult IsValid(object value, ValidationContext
validationContext)
    {
        Movie movie = (Movie)validationContext.ObjectInstance;

        if (movie.Genre == Genre.Classic && movie.ReleaseDate.Year > _year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }
```

上面的 `movie` 变量表示一个 `Movie` 对象，其中包含要验证的表单提交中的数据。在此例中，验证代码会根据规则检查 `ClassicMovieAttribute` 类的 `IsValid` 方法中的日期和流派。验证成功时，`IsValid` 返回 `ValidationResult.Success` 代码。验证失败时，返回 `ValidationResult` 和错误消息：

```
private string GetErrorMessage()
{
    return $"Classic movies must have a release year earlier than {_year}.";
}
```

当用户修改 `Genre` 字段并提交表单时，`ClassicMovieAttribute` 的 `IsValid` 方法将验证该电影是否为经典电影。将 `ClassicMovieAttribute` 像所有内置特性一样应用于属性（如 `ReleaseDate`）以确保执行验证，如前面的代码示例所示。由于此示例仅适用于 `Movie` 类型，因此建议使用 `IValidatableObject`，如下一段中所示。

也可以通过实现 `IValidatableObject` 接口上的 `Validate` 方法，将这段代码直接放入模型中。如果自定义验证特性可用于验证各个属性，则可使用 `IValidatableObject` 来实现类级别的验证，如下所示。

```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
    if (Genre == Genre.Classic && ReleaseDate.Year > _classicYear)
    {
        yield return new ValidationResult(
            $"Classic movies must have a release year earlier than {_classicYear}.",
            new[] { "ReleaseDate" });
    }
}
```

客户端验证

客户端验证极大地方便了用户。它节省了时间，让用户不必浪费时间等待服务器往返。从商业角度而言，即使每次只有几分之一秒，但如果每天有几百次，也会耗费大量的时间和成本，带来很多不必要的烦恼。简单直接的验证能够提高用户的工作效率和投入产出比。

你必须有一个包含适当的 JavaScript 脚本引用的视图，才能让客户端验证正常工作，如下所示。

```
<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.0.min.js"></script>
```

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/jquery.validate.min.js">
</script>
<script
src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.6/jquery.validate.unobt
rusive.min.js"></script>
```

jQuery 非介入式验证脚本是一个基于热门 jQuery Validate 插件的自定义 Microsoft 前端库。如果没有 jQuery 非介入式验证，则必须在两个位置编码相同的验证逻辑：一次是在模型属性上的服务器端验证特性中，一次是在客户端脚本中（jQuery Validate 的 `validate()` 方法示例展示了这种情况可能的复杂程度）。MVC 的标记帮助程序和 HTML 帮助程序则能够使用模型属性中的验证特性和类型元数据，呈现需要验证的表单元素中的 HTML 5 data- 特性。MVC 为内置属性和自定义属性生成 `data-` 属性。然后，jQuery 非介入式验证分析 `data-` 属性并将逻辑传递给 jQuery Validate，从而将服务器端验证逻辑有效地"复制"到客户端。可以使用相关标记帮助程序在客户端上显示验证错误，如下所示：

```
<div class="form-group">
    <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="ReleaseDate" class="form-control" />
        <span asp-validation-for="ReleaseDate" class="text-danger"></span>
    </div>
</div>
```

上面的标记帮助程序将呈现以下 HTML。请注意，HTML 输出中的 `data-` 特性与 `ReleaseDate` 属性的验证特性相对应。下面的 `data-val-required` 属性包含在用户未填写发行日期字段时将显示的错误消息。jQuery 非介入式验证将此值传递给 jQuery Validate `required()` 方法，该方法随后在随附的 **** 元素中显示该消息。

```
<form action="/Movies/Create" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <div class="text-danger"></div>
        <div class="form-group">
            <label class="col-md-2 control-label" for="ReleaseDate">ReleaseDate</label>
            <div class="col-md-10">
                <input class="form-control" type="datetime"
                data-val="true" data-val-required="The ReleaseDate field is required."
                id="ReleaseDate" name="ReleaseDate" value="" />
                <span class="text-danger field-validation-valid"
                data-valmsg-for="ReleaseDate" data-valmsg-replace="true"></span>
            </div>
        </div>
    </div>
</form>
```

客户端验证将阻止提交，直到表单变为有效为止。"提交"按钮运行 JavaScript：要么提交表单要么显示错误消息。

MVC 基于属性的 .NET 数据类型确定类型特性值（有可能使用 `[DataType]` 特性进行重写）。`[DataType]` 基本特性不执行真正的服务器端验证。浏览器选择自己的错误消息，并根据需要显示这些错误，但 jQuery 非介入式验证包可以重写消息，并使它们与其他消息的显示保持一致。当用户应用 `[DataType]` 子类（比如 `[EmailAddress]`）时，最常发生这种情况。

向动态**表**单**添加**验证

由于 jQuery 非介入式验证会在第一次加载页面时将验证逻辑和参数传递到 jQuery Validate，因此，动态生成的表单不会自动展示验证。你必须指示 jQuery 非介入式验证在创建动态表单后立即对其进行分析。例如，下面的代码展示如何对通过 AJAX 添加的表单设置客户端验证。

```
$.get({
    url: "https://url/that/returns/a/form",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add form. " + errorThrown);
    },
    success: function(newFormHTML) {
        var container = document.getElementById("form-container");
        container.insertAdjacentHTML("beforeend", newFormHTML);
        var forms = container.getElementsByTagName("form");
        var newForm = forms[forms.length - 1];
        $.validator.unobtrusive.parse(newForm);
    }
})
```

`$.validator.unobtrusive.parse()` 方法采用 jQuery 选择器作为它的一个参数。此方法指示 jQuery 非介入式验证分析该选择器内表单的 `data-` 属性。这些属性的值随后传递到 jQuery Validate 插件中，以便表单展示所需的客户端验证规则。

向动态控件添加验证

也可以在动态生成各个控件（比如 `<input/>` 和 `<select/>`）时，更新表单上的验证规则。不能将用于这些元素的选择器直接传递到 `parse()` 方法，因为周围表单已进行分析并且不会更新。应当先删除现有的验证数据，然后重新分析整个表单，如下所示：

```
$.get({
    url: "https://url/that/returns/a/control",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add control. " + errorThrown);
    },
    success: function(newInputHTML) {
        var form = document.getElementById("my-form");
        form.insertAdjacentHTML("beforeend", newInputHTML);
        $(form).removeData("validator")    // Added by jQuery Validate
               .removeData("unobtrusiveValidation");   // Added by jQuery Unobtrusive
Validation
        $.validator.unobtrusive.parse(form);
    }
})
```

IClientModelValidator

可为自定义属性创建客户端逻辑，创建 jQuery 验证的适配器的非介入式验证将在验证过程中，在客户端上自动为你执行此逻辑。第一步是通过实现 `IClientModelValidator` 接口来控制要添加哪些 data- 属性，如下所示：

```
public void AddValidation(ClientModelValidationContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    MergeAttribute(context.Attributes, "data-val", "true");
    MergeAttribute(context.Attributes, "data-val-classicmovie", GetErrorMessage());

    var year = _year.ToString(CultureInfo.InvariantCulture);
    MergeAttribute(context.Attributes, "data-val-classicmovie-year", year);
}
```

实现此接口的属性可以将 HTML 属性添加到生成的字段。检查 `ReleaseDate` 元素的输出时，将显示与上一示例类似的 HTML，唯一不同的是，此示例包含一个已在 `IClientModelValidator` 的 `AddValidation` 方法中定义的 `data-val-classicmovie` 属性。

```
<input class="form-control" type="datetime"
    data-val="true"
    data-val-classicmovie="Classic movies must have a release year earlier than 1960."
    data-val-classicmovie-year="1960"
    data-val-required="The ReleaseDate field is required."
    id="ReleaseDate" name="ReleaseDate" value="" />
```

非介入式验证使用 `data-` 属性中的数据来显示错误消息。不过，除非将规则或消息添加到 jQuery 的 `validator` 对象，否则 jQuery 并不知道它们的存在。如以下示例所示，将一个自定义 `classicmovie` 客户端验证方法添加到 `validator` 对象。有关 `unobtrusive.adapters.add` 方法的说明，请参阅 ASP.NET MVC 中的非介入式客户端验证。

```
$.validator.addMethod('classicmovie',
    function (value, element, params) {
        // Get element value. Classic genre has value '0'.
        var genre = $(params[0]).val(),
            year = params[1],
            date = new Date(value);
        if (genre && genre.length > 0 && genre[0] === '0') {
            // Since this is a classic movie, invalid if release date is after given year.
            return date.getFullYear() <= year;
        }

        return true;
    });

$.validator.unobtrusive.adapters.add('classicmovie',
    ['year'],
    function (options) {
        var element = $(options.form).find('select#Genre')[0];
        options.rules['classicmovie'] = [element, parseInt(options.params['year'])];
        options.messages['classicmovie'] = options.message;
    });
```

`classicmovie` 方法使用前面的代码对电影发行日期执行客户端验证。如果该方法返回 `false`，则显示错误消息。

远程验证

远程验证是一项非常不错的功能，可在需要根据服务器上的数据验证客户端上的数据时使用。例如，应用可能需要验证某个电子邮件或用户名是否已被使用，并且它必须为此查询大量数据。为验

证一个或几个字段而下载大量数据会占用过多资源。它还有可能暴露敏感信息。一种替代方法是发出往返请求来验证字段。

可以分两步实现远程验证。首先，必须使用 `[Remote]` 属性为模型添加批注。`[Remote]` 属性采用多个重载，可用于将客户端 JavaScript 定向到要调用的相应代码。下面的示例指向 `Users` 控制器的 `VerifyEmail` 操作方法。

```
[Remote(action: "VerifyEmail", controller: "Users")]
public string Email { get; set; }
```

第二步是按照 `[Remote]` 属性中的定义，将验证代码放入相应的操作方法。根据 jQuery Validate remote 方法文档，服务器响应必须是符合以下条件的 JSON 字符串：

- 对于有效元素，为 `"true"`。
- 对于无效元素，为 `"false"`、`undefined` 或 `null`，使用默认错误消息。

如果服务器响应是一个字符串（例如，`"That name is already taken, try peter123 instead"`），则该字符串显示为一条自定义错误消息来替代默认字符串。

`VerifyEmail` 方法的定义遵循这些规则，如下所示。如果电子邮件已被占用，它会返回验证错误消息；如果电子邮件可用，则返回 `true`，并将结果包装在 `JsonResult` 对象中。然后，客户端可以使用返回的值，继续进行下一步操作或根据需要显示错误。

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyEmail(string email)
{
    if (!_userRepository.VerifyEmail(email))
    {
        return Json($"Email {email} is already in use.");
    }

    return Json(true);
}
```

现在，当用户输入电子邮件时，视图中的 JavaScript 会发出远程调用，以了解该电子邮件是否已被占用，如果是，则显示错误消息。如果不是，用户就可以像往常一样提交表单。

`[Remote]` 特性的 `AdditionalFields` 属性可用于根据服务器上的数据验证字段组合。例如，如果上面的 `User` 模型具有两个附加属性，名为 `FirstName` 和 `LastName`，你可能想要验证该名称对尚未被现有用户占用。按以下代码所示定义新属性：

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(LastName))]
public string FirstName { get; set; }
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstName))]
public string LastName { get; set; }
```

`AdditionalFields` 可能已显式设置为字符串 `"FirstName"` 和 `"LastName"`，但使用 `nameof` 这样的操作符可简化稍后的重构过程。然后，用于执行验证的操作方法必须采用两个参数，一个用于 `FirstName` 的值，一个用于 `LastName` 的值。

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyName(string firstName, string lastName)
{
    if (!_userRepository.VerifyName(firstName, lastName))
    {
        return Json(data: $"A user named {firstName} {lastName} already exists.");
    }

    return Json(data: true);
}
```

现在，当用户输入名和姓时，JavaScript 会：

- 发出远程调用，以了解该名称对是否已被占用。
- 如果被占用，则显示一条错误消息。
- 如果未被占用，则用户可以提交表单。

如果需要使用 `[Remote]` 特性验证两个或更多附加字段，可将其以逗号分隔的列表形式列出。例如，若要向模型中添加 `MiddleName` 属性，可按以下代码所示设置 `[Remote]` 特性：

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstName) + ","
+ nameof(LastName))]
public string MiddleName { get; set; }
```

`AdditionalFields` 与所有属性参数一样，必须是常量表达式。因此，不能使用内插字符串或调用 `string.Join()` 来初始化 `AdditionalFields`。对于添加到 `[Remote]` 特性的每个附加字段，都必须向相应的控制器操作方法另外添加一个参数。

ASP.NET Core 的 Razor 语法参考

2018/12/10 • • Edit Online

作者：Rick Anderson、Luke Latham、Taylor Mullen 和 Dan Vicarel

Razor 是一种标记语法，用于将基于服务器的代码嵌入网页中。Razor 语法由 Razor 标记、C# 和 HTML 组成。包含 Razor 的文件通常具有 *.cshtml* 文件扩展名。

呈现 HTML

默认 Razor 语言为 HTML。从 Razor 标记呈现 HTML 与从 HTML 文件呈现 HTML 并没有什么不同。服务器会按原样呈现 *.cshtml* Razor 文件中的 HTML 标记。

Razor 语法

Razor 支持 C#，并使用 `@` 符号从 HTML 转换为 C#。Razor 计算 C# 表达式，并将它们呈现在 HTML 输出中。

当 `@` 符号后跟 Razor 保留关键字时，它会转换为 Razor 特定标记。否则会转换为纯 C#。

若要对 Razor 标记中的 `@` 符号进行转义，请使用另一个 `@` 符号：

```
<p>@@Username</p>
```

该代码在 HTML 中使用单个 `@` 符号呈现：

```
<p>@Username</p>
```

包含电子邮件地址的 HTML 属性和内容不将 `@` 符号视为转换字符。Razor 分析不会处理以下示例中的电子邮件地址：

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

隐式 Razor 表达式

隐式 Razor 表达式以 `@` 开头，后跟 C# 代码：

```
<p>@DateTime.Now</p>
<p>@DateTime.IsLeapYear(2016)</p>
```

隐式表达式不能包含空格，但 C# `await` 关键字除外。如果该 C# 语句具有明确的结束标记，则可以混用空格：

```
<p>@await DoSomething("hello", "world")</p>
```

隐式表达式**不能**包含 C# 泛型，因为括号（`<>`）内的字符会被解释为 HTML 标记。以下代码**无效**：

```
<p>@GenericMethod<int>()</p>
```

上述代码生成与以下错误之一类似的编译器错误：

- "int" 元素未结束。所有元素都必须自结束或具有匹配的结束标记。
- 无法将方法组 "GenericMethod" 转换为非委托类型 "object"。是否希望调用此方法?`

泛型方法调用必须包装在显式 Razor 表达式或 Razor 代码块中。

显式 Razor 表达式

显式 Razor 表达式由 `@` 符号和平衡圆括号组成。若要呈现上一周的时间，可使用以下 Razor 标记：

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

将计算 `@()` 括号中的所有内容，并将其呈现到输出中。

前面部分中所述的隐式表达式通常不能包含空格。在下面的代码中，不会从当前时间减去一周：

```
<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>
```

该代码呈现以下 HTML：

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```

可以使用显式表达式将文本与表达式结果串联起来：

```
@{
    var joe = new Person("Joe", 33);
}

<p>Age@(joe.Age)</p>
```

如果不使用显式表达式，`<p>Age@joe.Age</p>` 会被视为电子邮件地址，因此会呈现 `<p>Age@joe.Age</p>`。如果编写为显式表达式，则呈现 `<p>Age33</p>`。

显式表达式可用于从 *.cshtml* 文件中的泛型方法呈现输出。以下标记显示了如何更正之前出现的由 C# 泛型的括号引起的错误。此代码以显式表达式的形式编写：

```
<p>@(GenericMethod<int>())</p>
```

表达式编码

计算结果为字符串的 C# 表达式采用 HTML 编码。计算结果为 `IHtmlContent` 的 C# 表达式直接通过 `IHtmlContent.WriteTo` 呈现。计算结果不为 `IHtmlContent` 的 C# 表达式通过 `ToString` 转换为字符串，并在呈现前进行编码。

```
@("<span>Hello World</span>")
```

该代码呈现以下 HTML：

```
&lt;span&gt;Hello World&lt;/span&gt;
```

该 HTML 在浏览器中显示为：

```
<span>Hello World</span>
```

`HtmlHelper.Raw` 输出不进行编码，但呈现为 HTML 标记。

> **WARNING**
> 对未经审查的用户输入使用 `HtmlHelper.Raw` 会带来安全风险。用户输入可能包含恶意的 JavaScript 或其他攻击。审查用户输入比较困难。应避免对用户输入使用 `HtmlHelper.Raw` 。

```
@Html.Raw("<span>Hello World</span>")
```

该代码呈现以下 HTML：

```
<span>Hello World</span>
```

Razor 代码块

Razor 代码块以 `@` 开头，并括在 `{}` 中。代码块内的 C# 代码不会呈现，这点与表达式不同。一个视图中的代码块和表达式共享相同的作用域并按顺序进行定义：

```
@{
    var quote = "The future depends on what you do today. - Mahatma Gandhi";
}

<p>@quote</p>

@{
    quote = "Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.";
}

<p>@quote</p>
```

该代码呈现以下 HTML：

```
<p>The future depends on what you do today. - Mahatma Gandhi</p>
<p>Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.</p>
```

隐式转换

代码块中的默认语言为 C#，不过，Razor 页面可以转换回 HTML：

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

带分隔符的显式转换

若要定义应呈现 HTML 的代码块子节，请使用 Razor **<text>** 标记将要呈现的字符括起来：

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

使用此方法可呈现未被 HTML 标记括起来的 HTML。如果没有 HTML 或 Razor 标记，会发生 Razor 运行时错误。

<text> 标记可用于在呈现内容时控制空格：

- 仅呈现 **<text>** 标记之间的内容。
- **<text>** 标记之前或之后的空格不会显示在 HTML 输出中。

使用 @ 的显式行转换：

若要在代码块内以 HTML 的形式呈现整个行的其余内容，请使用 `@:` 语法：

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

如果代码中没有 `@:`，会生成 Razor 运行时错误。

警告：Razor 文件中多余的 `@` 字符可能会导致代码块中后面的语句发生编译器错误。这些编译器错误可能难以理解，因为实际错误发生在报告的错误之前。将多个隐式/显式表达式合并到单个代码块以后，经常会发生此错误。

控制结构

控制结构是对代码块的扩展。代码块的各个方面（转换为标记、内联 C#）同样适用于以下结构：

条件语句 @if、else if、else 和 @switch

`@if` 控制何时运行代码：

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
```

`else` 和 `else if` 不需要 `@` 符号：

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value is odd and small.</p>
}
```

以下标记展示如何使用 switch 语句：

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number wasn't 1 or 1337.</p>
        break;
}
```

循环语句 @for、@foreach、@while 和 @do while

可以使用循环控制语句呈现模板化 HTML。若要呈现一组人员：

```
@{
    var people = new Person[]
    {
          new Person("Weston", 33),
          new Person("Johnathon", 41),
          ...
    };
}
```

支持以下循环语句：

`@for`

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@foreach`

```
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@while`

```
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
}
```

`@do while`

```
@{ var i = 0; }
@do
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
} while (i < people.Length);
```

复合语句 @using

在 C# 中，`using` 语句用于确保释放对象。在 Razor 中，可使用相同的机制来创建包含附加内容的 HTML 帮助程序。在下面的代码中，HTML 帮助程序使用 `@using` 语句呈现表单标记：

```
@using (Html.BeginForm())
{
    <div>
        email:
        <input type="email" id="Email" value="">
        <button>Register</button>
    </div>
}
```

可以使用标记帮助程序执行作用域级别的操作。

@try、catch、finally

异常处理与 C# 类似：

```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```

@lock

Razor 可以使用 lock 语句来保护关键节：

```
@lock (SomeLock)
{
    // Do critical section work
}
```

注释

Razor 支持 C# 和 HTML 注释：

```
@{
    /* C# comment */
    // Another C# comment
}
<!-- HTML comment -->
```

该代码呈现以下 HTML：

```
<!-- HTML comment -->
```

在呈现网页之前，服务器会删除 Razor 注释。Razor 使用 `@* *@` 来分隔注释。以下代码已被注释禁止，因此服务器不呈现任何标记：

```
@*
    @{
        /* C# comment */
        // Another C# comment
    }
    <!-- HTML comment -->
*@
```

指令

Razor 指令由隐式表达式表示：`@` 符号后跟保留关键字。指令通常用于更改视图分析方式或启用不同的功能。

通过了解 Razor 如何为视图生成代码，更易理解指令的工作原理。

```
@{
    var quote = "Getting old ain't for wimps! - Anonymous";
}

<div>Quote of the Day: @quote</div>
```

该代码生成与下面类似的类：

```
public class _Views_Something_cshtml : RazorPage<dynamic>
{
    public override async Task ExecuteAsync()
    {
        var output = "Getting old ain't for wimps! - Anonymous";

        WriteLiteral("/r/n<div>Quote of the Day: ");
        Write(output);
        WriteLiteral("</div>");
    }
}
```

本文后面的检查为视图生成的 Razor C# 类部分说明了如何查看此生成的类。

@using

`@using` 指令用于向生成的视图添加 C# `using` 指令：

```
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

@model

`@model` 指令指定传递到视图的模型类型：

```
@model TypeNameOfModel
```

在使用个人用户帐户创建的 ASP.NET Core MVC 应用中，*Views/Account/Login.cshtml* 视图包含以下模型声明：

```
@model LoginViewModel
```

生成的类继承自 `RazorPage<dynamic>`：

```
public class _Views_Account_Login_cshtml : RazorPage<LoginViewModel>
```

Razor 公开了 `Model` 属性，用于访问传递到视图的模型：

```
<div>The Login Email: @Model.Email</div>
```

`@model` 指令指定此属性的类型。该指令将 `RazorPage<T>` 中的 `T` 指定为生成的类，视图便派生自该类。如果未指定 `@model` 指令，则 `Model` 属性的类型为 `dynamic`。模型的值会从控制器传递到视图。有关详细信息，请参阅强类型模型和@模型关键字。

@inherits

`@inherits` 指令对视图继承的类提供完全控制：

```
@inherits TypeNameOfClassToInheritFrom
```

下面的代码是一种自定义 Razor 页面类型：

```
using Microsoft.AspNetCore.Mvc.Razor;

public abstract class CustomRazorPage<TModel> : RazorPage<TModel>
{
    public string CustomText { get; } = "Gardyloo! - A Scottish warning yelled from a window before
dumping a slop bucket on the street below.";
}
```

`CustomText` 显示在视图中：

```
@inherits CustomRazorPage<TModel>

<div>Custom text: @CustomText</div>
```

该代码呈现以下 HTML：

```
<div>Custom text: Gardyloo! - A Scottish warning yelled from a window before dumping a slop bucket on the
street below.</div>
```

`@model` 和 `@inherits` 可在同一视图中使用。`@inherits` 可位于视图导入的 *_ViewImports.cshtml* 文件中：

```
@inherits CustomRazorPage<TModel>
```

下面的代码是一种强类型视图：

```
@inherits CustomRazorPage<TModel>

<div>The Login Email: @Model.Email</div>
<div>Custom text: @CustomText</div>
```

如果在模型中传递“rick@contoso.com”，视图将生成以下 HTML 标记：

```
<div>The Login Email: rick@contoso.com</div>
<div>Custom text: Gardyloo! - A Scottish warning yelled from a window before dumping a slop bucket on the
street below.</div>
```

@inject

`@inject` 指令允许 Razor 页面将服务从服务容器注入到视图。有关详细信息，请参阅视图中的依赖关系注入。

@functions

`@functions` 指令允许 Razor 页面将 C# 代码块添加到视图中：

```
@functions { // C# Code }
```

例如：

```
@functions {
    public string GetHello()
    {
        return "Hello";
    }
}

<div>From method: @GetHello()</div>
```

该代码生成以下 HTML 标记：

```
<div>From method: Hello</div>
```

以下代码是生成的 Razor C# 类：

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Razor;

public class _Views_Home_Test_cshtml : RazorPage<dynamic>
{
    // Functions placed between here
    public string GetHello()
    {
        return "Hello";
    }
    // And here.
#pragma warning disable 1998
    public override async Task ExecuteAsync()
    {
        WriteLiteral("\r\n<div>From method: ");
        Write(GetHello());
        WriteLiteral("</div>\r\n");
    }
#pragma warning restore 1998
```

@section

`@section` 指令与布局结合使用，允许视图将内容呈现在 HTML 页面的不同部分。有关详细信息，请参阅部分。

模板化 Razor 委托

通过 Razor 模板，可使用以下格式定义 UI 代码片段：

```
@<tag>...</tag>
```

下面的示例演示如何指定模板化 Razor 委托作为 Func<T,TResult>。为委托封装的方法的参数指定动态类型。将对象类型指定为委托的返回值。该模板与 `Pet`（具有 `Name` 属性）的 List<T> 一起使用。

```
public class Pet
{
    public string Name { get; set; }
}
```

```
@{
    Func<dynamic, object> petTemplate = @<p>You have a pet named @item.Name.</p>;

    var pets = new List<Pet>
    {
        new Pet { Name = "Rin Tin Tin" },
        new Pet { Name = "Mr. Bigglesworth" },
        new Pet { Name = "K-9" }
    };
}
```

使用 `foreach` 语句提供的 `pets` 呈现该模板：

```
@foreach (var pet in pets)
{
    @petTemplate2(pet)
}
```

呈现的输出：

```
<p>You have a pet named <strong>Rin Tin Tin</strong>.</p>
<p>You have a pet named <strong>Mr. Bigglesworth</strong>.</p>
<p>You have a pet named <strong>K-9</strong>.</p>
```

还可以提供内联 Razor 模板作为方法的参数。如下示例中，`Repeat` 方法收到一个 Razor 模板。该方法使用模板生成 HTML 内容，其中包含列表中提供的重复项：

```
@using Microsoft.AspNetCore.Html

@functions {
    public static IHtmlContent Repeat(IEnumerable<dynamic> items, int times,
        Func<dynamic, IHtmlContent> template)
    {
        var html = new HtmlContentBuilder();

        foreach (var item in items)
        {
            for (var i = 0; i < times; i++)
            {
                html.AppendHtml(template(item));
            }
        }

        return html;
    }
}
```

使用前面示例中的 pets 列表，调用 `Repeat` 方法以及：

- `Pet` 的 List<T>。
- 每只宠物的重复次数。
- 用于无序列表的列表项的内联模板。

```
<ul>
    @Repeat(pets, 3, @<li>@item.Name</li>)
</ul>
```

呈现的输出：

```
<ul>
    <li>Rin Tin Tin</li>
    <li>Rin Tin Tin</li>
    <li>Rin Tin Tin</li>
    <li>Mr. Bigglesworth</li>
    <li>Mr. Bigglesworth</li>
    <li>Mr. Bigglesworth</li>
    <li>K-9</li>
    <li>K-9</li>
    <li>K-9</li>
</ul>
```

标记帮助程序

标记帮助程序有三个相关指令。

指令	函数
@addTagHelper	向视图提供标记帮助程序。
@removeTagHelper	从视图中删除以前添加的标记帮助程序。
@tagHelperPrefix	指定标记前缀，以启用标记帮助程序支持并阐明标记帮助程序的用法。

Razor 保留关键字

Razor 关键字

- page（需要 ASP.NET Core 2.0 及更高版本）
- namespace
- functions
- inherits
- model
- section
- helper（ASP.NET Core 当前不支持）

Razor 关键字使用 `@(Razor Keyword)` 进行转义（例如，`@(functions)`）。

C# Razor 关键字

- case
- do
- default
- for
- foreach
- if
- else
- lock
- switch
- try
- catch
- finally
- using
- while

C# Razor 关键字必须使用 `@(@C# Razor Keyword)` 进行双转义（例如，`@(@case)`）。第一个 `@` 对 Razor 分析器转义。第二个 `@` 对 C# 分析器转义。

Razor 不使用的保留关键字

- class

检查为视图生成的 Razor C# 类

在 .NET Core SDK 2.1 或更高版本中，Razor SDK 负责编译 Razor 文件。生成项目时，Razor SDK 在项目根目录中生成 obj/<build_configuration>/<target_framework_moniker>/Razor 目录。Razor 目录中的目录结构反映项目的目录结构。

在面向 .NET Core 2.1 的 ASP.NET Core 2.1 Razor Pages 项目中，请考虑以下目录结构：

- Areas/
  - Admin/
    - Pages/
      - Index.cshtml
      - Index.cshtml.cs
- Pages/
  - Shared/
    - _Layout.cshtml
  - _ViewImports.cshtml
  - *_ViewStart.cshtml*
  - Index.cshtml
  - Index.cshtml.cs

在 Debug 配置下生成项目将生成以下 obj 目录：

- obj/
  - Debug/
    - netcoreapp2.1/
      - Razor/
        - Areas/
          - Admin/
            - Pages/
              - Index.g.cshtml.cs
        - Pages/
          - Shared/
            - _Layout.g.cshtml.cs
          - _ViewImports.g.cshtml.cs
          - _ViewStart.g.cshtml.cs
          - Index.g.cshtml.cs

若要查看 Pages/Index.cshtml 的生成类，请打开 obj/Debug/netcoreapp2.1/Razor/Pages/Index.g.cshtml.cs。

将下面的类添加到 ASP.NET Core MVC 项目：

```
using Microsoft.AspNetCore.Mvc.Razor.Extensions;
using Microsoft.AspNetCore.Razor.Language;

public class CustomTemplateEngine : MvcRazorTemplateEngine
{
    public CustomTemplateEngine(RazorEngine engine, RazorProject project)
        : base(engine, project)
    {
    }

    public override RazorCSharpDocument GenerateCode(RazorCodeDocument codeDocument)
    {
        var csharpDocument = base.GenerateCode(codeDocument);
        var generatedCode = csharpDocument.GeneratedCode;

        // Look at generatedCode

        return csharpDocument;
    }
}
```

在 `Startup.ConfigureServices` 中，使用 `CustomTemplateEngine` 类替代 MVC 添加的 `RazorTemplateEngine`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddSingleton<RazorTemplateEngine, CustomTemplateEngine>();
}
```

在 `CustomTemplateEngine` 的 `return csharpDocument;` 语句上设置断点。当程序执行在断点处停止时，查看 `generatedCode` 的值。

视图查找和区分大小写

Razor 视图引擎为视图执行区分大小写的查找。但是，实际查找取决于基础文件系统：

- 基于文件的源：
  - 在使用不区分大小写的文件系统的操作系统（例如，Windows）上，物理文件提供程序查找不区分大小写。例如，`return View("Test")` 可匹配 */Views/Home/Test.cshtml*、*/Views/home/test.cshtml* 以及任何其他大小写变体。
  - 在区分大小写的文件系统（例如，Linux、OSX 以及使用 `EmbeddedFileProvider` 构建的文件系统）上，查找区分大小写。例如，`return View("Test")` 专门匹配 */Views/Home/Test.cshtml*。
- 预编译视图：在 ASP.NET Core 2.0 及更高版本中，预编译视图查找在所有操作系统上均不区分大小写。该行为与 Windows 上物理文件提供程序的行为相同。如果两个预编译视图仅大小写不同，则查找的结果具有不确定性。

建议开发人员将文件和目录名称的大小写与以下项的大小写匹配：

- 区域、控制器和操作名称。
- Razor 页面。

匹配大小写可确保无论使用哪种基础文件系统，部署都能找到其视图。

ASP.NET Core 中的视图组件

2018/12/14 • • Edit Online

作者：Rick Anderson

查看或下载示例代码（如何下载）

视图组件

视图组件与分部视图类似，但它们的功能更加强大。视图组件不使用模型绑定，并且仅依赖调用时提供的数据。本文通过使用 ASP.NET Core MVC 撰写，但视图组件也适用于 Razor 页面。

视图组件：

- 呈现一个区块而不是整个响应。
- 包括控制器和视图间发现的相同关注点分离和可测试性优势。
- 可以有参数和业务逻辑。
- 通常从布局页调用。

视图组件可用于具有可重用呈现逻辑（对分部视图来说过于复杂）的任何位置，例如：

- 动态导航菜单
- 标记云（查询数据库的位置）
- 登录面板
- 购物车
- 最近发布的文章
- 典型博客上的边栏内容
- 一个登录面板，呈现在每页上并显示注销或登录链接，具体取决于用户的登录状态

视图组件由两部分组成：类（通常派生自 ViewComponent）及其返回的结果（通常为视图）。与控制器一样，视图组件也可以是 POCO，但大多数开发人员都希望利用派生自 `ViewComponent` 的可用方法和属性。

创建视图组件

本部分包含创建视图组件的高级别要求。本文后续部分将详细检查每个步骤并创建视图组件。

视图组件类

可通过以下任一方法创建视图组件类：

- 从 ViewComponent 派生
- 使用 `[ViewComponent]` 属性修饰类，或者从具有 `[ViewComponent]` 属性的类派生
- 创建名称以 ViewComponent 后缀结尾的类

与控制器一样，视图组件必须是公共、非嵌套和非抽象的类。视图组件名称是删除了“ViewComponent”后缀的类名。也可以使用 `ViewComponentAttribute.Name` 属性显式指定它。

视图组件类：

- 完全支持构造函数依赖关系注入
- 不参与控制器生命周期，这意味着不能在视图组件中使用筛选器

视图组件方法

视图组件以返回 `Task<IViewComponentResult>` 的 `InvokeAsync` 方法，或是以返回 `IViewComponentResult` 的同步 `Invoke` 方法定义其逻辑。参数直接来自视图组件的调用，而不是来自模型绑定。视图组件从不直接处理请求。通常，视图组件通过调用 `View` 方法来初始化模型并将其传递到视图。总之，视图组件方法：

- 定义返回 `Task<IViewComponentResult>` 的 `InvokeAsync` 方法，或是返回 `IViewComponentResult` 的同步 `Invoke` 方法。
- 一般通过调用 `ViewComponent` `View` 方法来初始化模型并将其传递到视图。
- 参数来自调用方法，而不是 HTTP。没有模型绑定。
- 不可直接作为 HTTP 终结点进行访问。通过代码调用它们（通常在视图中）。视图组件从不处理请求。
- 在签名上重载，而不是当前 HTTP 请求的任何详细信息。

视图搜索路径

运行时在以下路径中搜索视图：

- /Pages/Components/{View Component Name}/{View Name}
- /Views/{Controller Name}/Components/{View Component Name}/{View Name}
- /Views/Shared/Components/{View Component Name}/{View Name}

视图组件的默认视图名称为“默认”，这意味着视图文件通常命名为“Default.cshtml”。可以在创建视图组件结果或调用 `View` 方法时指定不同的视图名称。

建议将视图文件命名为 Default.cshtml 并使用 Views/Shared/Components/{View Component Name}/{View Name} 路径。此示例中使用的 `PriorityList` 视图组件对视图组件视图使用 Views/Shared/Components/PriorityList/Default.cshtml。

调用视图组件

要使用视图组件，请在视图中调用以下内容：

```
@await Component.InvokeAsync("Name of view component", {Anonymous Type Containing Parameters})
```

参数将传递给 `InvokeAsync` 方法。本文中开发的 `PriorityList` 视图组件是从 Views/Todo/Index.cshtml 视图文件中调用的。在下例中，使用两个参数调用 `InvokeAsync` 方法：

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

调用视图组件作为标记帮助程序

对于 ASP.NET Core 1.1 及更高版本，可以调用视图组件作为标记帮助程序：

```
<vc:priority-list max-priority="2" is-done="false">
</vc:priority-list>
```

标记帮助程序采用 Pascal 大小写格式的类和方法参数将转换为各自相应的小写短横线格式。要调用视图组件的标记帮助程序使用 `<vc></vc>` 元素。按如下方式指定视图组件：

```
<vc:[view-component-name]
  parameter1="parameter1 value"
  parameter2="parameter2 value">
</vc:[view-component-name]>
```

若要将视图组件用作标记帮助程序，请使用 `@addTagHelper` 指令注册包含视图组件的程序集。如果视图组件位

于名为“`MyWebApp`”的程序集中，请将以下指令添加到 _ViewImports.cshtml 文件：

```
@addTagHelper *, MyWebApp
```

可将视图组件作为标记帮助程序注册到任何引用视图组件的文件。要详细了解如何注册标记帮助程序，请参阅管理标记帮助程序作用域。

本教程中使用的 `InvokeAsync` 方法：

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

在标记帮助程序标记中：

```
<vc:priority-list max-priority="2" is-done="false">
</vc:priority-list>
```

在以上示例中，`PriorityList` 视图组件变为 `priority-list`。视图组件的参数作为小写短横线格式的属性进行传递。

从控制器直接调用视图组件

视图组件通常从视图调用，但你可以直接从控制器方法调用它们。尽管视图组件不定义控制器等终结点，但你可以轻松实现返回 `ViewComponentResult` 内容的控制器操作。

在此示例中，视图组件直接从控制器调用：

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```

演练：创建简单的视图组件

下载、生成和测试起始代码。它是一个带有 `Todo` 控制器的简单项目，该控制器显示 Todo 项的列表。

添加 ViewComponent 类

创建一个 ViewComponents 文件夹并添加以下 `PriorityListViewComponent` 类：

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityListViewComponent : ViewComponent
    {
        private readonly ToDoContext db;

        public PriorityListViewComponent(ToDoContext context)
        {
            db = context;
        }

        public async Task<IViewComponentResult> InvokeAsync(
        int maxPriority, bool isDone)
        {
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(items);
        }
        private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
            return db.ToDo.Where(x => x.IsDone == isDone &&
                                 x.Priority <= maxPriority).ToListAsync();
        }
    }
}
```

代码说明：

- 视图组件类可以包含在项目的任意文件夹中。
- 因为类名 PriorityListViewComponent 以后缀 ViewComponent 结尾，所以运行时将在从视图引用类组件时使用字符串“PriorityList”。我稍后将进行详细解释。
- `[ViewComponent]` 属性可以更改用于引用视图组件的名称。例如，我们可以将类命名为 `XYZ` 并应用 `ViewComponent` 属性：

  ```
  [ViewComponent(Name = "PriorityList")]
     public class XYZ : ViewComponent
  ```

- 上面的 `[ViewComponent]` 属性通知视图组件选择器在查找与组件相关联的视图时使用名称 `PriorityList`，以及在从视图引用类组件时使用字符串“PriorityList”。我稍后将进行详细解释。
- 组件使用依赖关系注入以使数据上下文可用。
- `InvokeAsync` 公开可以从视图调用的方法，且可以采用任意数量的参数。
- `InvokeAsync` 方法返回满足 `isDone` 和 `maxPriority` 参数的 `ToDo` 项集。

创建视图组件 Razor 视图

- 创建 Views/Shared/Components 文件夹。此文件夹 **必** 须 命名为 *Components*。
- 创建 Views/Shared/Components/PriorityList 文件夹。此文件夹名称必须与视图组件类的名称或类名去掉后缀（如果遵照约定并在类名中使用了“ViewComponent”后缀）的名称相匹配。如果使用了

`ViewComponent` 属性，则类名称需要匹配指定的属性。

- 创建 Views/Shared/Components/PriorityList/Default.cshtml Razor 视图：[!code-cshtml]

  Razor 视图获取并显示 `TodoItem` 列表。如果视图组件 `InvokeAsync` 方法不传递视图名称（如示例中所示），则按照约定使用"默认"作为视图名称。在本教程后面部分，我将演示如何传递视图名称。要替代特定控制器的默认样式，请将视图添加到控制器特定的视图文件夹（例如 Views/Todo/Components/PriorityList/Default.cshtml）。

  如果视图组件是控制器特定的，则可将其添加到控制器特定的文件夹 (Views/Todo/Components/PriorityList/Default.cshtml)。

- 将包含优先级列表组件调用的 `div` 添加到 Views/Todo/index.cshtml 文件底部：

```
</table>
<div>
    @await Component.InvokeAsync("PriorityList", new { maxPriority = 2, isDone = false })
</div>
```

标记 `@await Component.InvokeAsync` 显示调用视图组件的语法。第一个参数是要调用的组件的名称。后续参数将传递给该组件。`InvokeAsync` 可以采用任意数量的参数。

测试应用。下图显示 ToDo 列表和优先级项：

也可直接从控制器调用视图组件：

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```

指定视图名称

在某些情况下，复杂的视图组件可能需要指定非默认视图。以下代码显示如何从 `InvokeAsync` 方法指定"PVC"视图。更新 `PriorityListViewComponent` 类中的 `InvokeAsync` 方法。

```
public async Task<IViewComponentResult> InvokeAsync(
    int maxPriority, bool isDone)
{
    string MyView = "Default";
    // If asking for all completed tasks, render with the "PVC" view.
    if (maxPriority > 3 && isDone == true)
    {
        MyView = "PVC";
    }
    var items = await GetItemsAsync(maxPriority, isDone);
    return View(MyView, items);
}
```

将 Views/Shared/Components/PriorityList/Default.cshtml 文件复制到名为 Views/Shared/Components/PriorityList/PVC.cshtml 的视图。添加标题以指示正在使用 PVC 视图。

```
@model IEnumerable<ViewComponentSample.Models.TodoItem>

<h2> PVC Named Priority Component View</h2>
<h4>@ViewBag.PriorityMessage</h4>
<ul>
    @foreach (var todo in Model)
    {
        <li>@todo.Name</li>
    }
</ul>
```

更新 Views/TodoList/Index.cshtml：

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

运行应用并验证 PVC 视图。

如果不呈现 PVC 视图，请验证是否调用优先级为 4 或更高的视图组件。

检查视图路径

- 将优先级参数更改为 3 或更低，从而不返回优先级视图。
- 将 Views/Todo/Components/PriorityList/Default.cshtml 暂时重命名为 1Default.cshtml。
- 测试应用，你将收到以下错误：

```
An unhandled exception occurred while processing the request.
InvalidOperationException: The view 'Components/PriorityList/Default' wasn't found. The following
locations were searched:
/Views/ToDo/Components/PriorityList/Default.cshtml
/Views/Shared/Components/PriorityList/Default.cshtml
EnsureSuccessful
```

- 将 Views/Todo/Components/PriorityList/1Default.cshtml 复制到 Views/Shared/Components/PriorityList/Default.cshtml。
- 将一些标记添加到共享 Todo 视图组件视图以指示视图来自“Shared”文件夹。
- 测试“共享”组件视图。

避免魔幻字符串

若要确保编译时的安全性，可以用类名替换硬编码的视图组件名称。创建没有“ViewComponent”后缀的视图组件：

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityList : ViewComponent
    {
        private readonly ToDoContext db;

        public PriorityList(ToDoContext context)
        {
            db = context;
        }

        public async Task<IViewComponentResult> InvokeAsync(
        int maxPriority, bool isDone)
        {
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(items);
        }
        private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
            return db.ToDo.Where(x => x.IsDone == isDone &&
                                 x.Priority <= maxPriority).ToListAsync();
        }
    }
}
```

将 `using` 语句添加到 Razor 视图文件，并使用 `nameof` 运算符：

```
@using ViewComponentSample.Models
@using ViewComponentSample.ViewComponents
@model IEnumerable<TodoItem>

<h2>ToDo nameof</h2>
<!-- Markup removed for brevity.  -->

<div>

    @await Component.InvokeAsync(nameof(PriorityList), new { maxPriority = 4, isDone = true })
</div>
```

执行同步工作

如果不需要执行异步工作，框架将处理调用同步 `Invoke` 方法。以下方法将创建同步 `Invoke` 视图组件：

```
public class PriorityList : ViewComponent
{
    public IViewComponentResult Invoke(int maxPriority, bool isDone)
    {
        var items = new List<string> { $"maxPriority: {maxPriority}", $"isDone: {isDone}" };
        return View(items);
    }
}
```

视图组件的 Razor 文件列出了传递给 `Invoke` 方法的字符串（Views/Home/Components/PriorityList/Default.cshtml）：

```
@model List<string>

<h3>Priority Items</h3>
<ul>
    @foreach (var item in Model)
    {
        <li>@item</li>
    }
</ul>
```

使用以下方法之一在 Razor 文件（例如，Views/Home/Index.cshtml）中调用视图组件：

- IViewComponentHelper
- 标记帮助程序

若要使用 IViewComponentHelper 方法，请调用 `Component.InvokeAsync`：

使用 IViewComponentHelper 在 Razor 文件（例如，Views/Home/Index.cshtml）中调用视图组件。

调用 `Component.InvokeAsync`：

```
@await Component.InvokeAsync(nameof(PriorityList), new { maxPriority = 4, isDone = true })
```

若要使用标记帮助程序，请使用 `@addTagHelper` 指令注册包含视图组件的程序集（视图组件位于名为 `MyWebApp` 的程序集中）：

```
@addTagHelper *, MyWebApp
```

在 Razor 标记文件中使用视图组件标记帮助程序：

```
<vc:priority-list max-priority="999" is-done="false">
</vc:priority-list>
```

`PriorityList.Invoke` 的方法签名是同步的，但 Razor 在标记文件中使用 `Component.InvokeAsync` 找到并调用该方法。

其他资源

- 视图中的依赖关系注入

ASP.NET Core 中的 Razor 文件编译

2018/9/24 • • Edit Online

作者：Rick Anderson

调用相关的 MVC 视图时，Razor 文件在运行时进行编译。不支持在生成时发布 Razor 文件。可以使用预编译工具，在发布时选择编译 Razor 文件并将其与应用一起部署。

调用相关的 Razor 页和 MVC 视图时，Razor 文件在运行时进行编译。不支持在生成时发布 Razor 文件。可以使用预编译工具，在发布时选择编译 Razor 文件并将其与应用一起部署。

调用相关的 Razor 页和 MVC 视图时，Razor 文件在运行时进行编译。在生成时和发布时使用 Razor SDK 编译 Razor 文件。

预编译注意事项

以下是预编译 Razor 文件的意外后果：

- 发布捆绑包更小
- 启动速度更快
- 无法编辑 Razor 文件—关联内容不会出现在发布捆绑包中。

部署预编译文件

Razor SDK 默认启用 Razor 文件的生成时和发布时编译。Razor 文件更新后，支持在生成时编辑这些文件。默认情况下，仅通过应用部署编译的 Views.dll 而不部署 cshtml 文件。

> **IMPORTANT**
>
> ASP.NET Core 3.0 中将删除预编译工具。建议迁移到 Razor Sdk。
>
> 仅当项目文件中未设置特定于预编译的属性时，Razor SDK 才有效。例如，通过将 .csproj 文件的 `MvcRazorCompileOnPublish` 属性设置为 `true` 来禁用 Razor SDK。

如果项目面向 .NET Framework，请安装 Microsoft.AspNetCore.Mvc.Razor.ViewCompilation NuGet 包：

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation"
                  Version="2.0.4"
                  PrivateAssets="All" />
```

如果项目面向 .NET Core，则无需进行任何更改。

默认情况下，ASP.NET Core 2.x 项目模板将 `MvcRazorCompileOnPublish` 属性隐式设置为 `true`。因此，可以从 .csproj 文件中安全地删除此元素。

> **IMPORTANT**
>
> ASP.NET Core 3.0 中将删除预编译工具。建议迁移到 Razor Sdk。
>
> 在 ASP.NET Core 2.0 中执行独立部署 (SCD) 时，无法使用 Razor 文件预编译。

将 `MvcRazorCompileOnPublish` 属性设置为 `true`，然后安装 Microsoft.AspNetCore.Mvc.Razor.ViewCompilation

NuGet 包。以下 .csproj 示例突出显示了这些设置：

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
    <MvcRazorCompileOnPublish>true</MvcRazorCompileOnPublish>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="1.1.0-*" />
  </ItemGroup>
</Project>
```

使用 .NET Core CLI 发布命令，让应用做好框架依赖型部署的准备。例如，在项目根目录中执行以下命令：

```
dotnet publish -c Release
```

预编译成功后，将生成包含已编译 Razor 文件的 <project_name>.PrecompiledViews.dll 文件。例如，以下屏幕截图描述了 WebApplication1.PrecompiledViews.dll 中 Index.cshtml 的内容：

其他资源

- ASP.NET Core MVC 中的视图
- ASP.NET Core 中的 Razor 页面介绍
- ASP.NET Core MVC 中的视图
- ASP.NET Core 中的 Razor 页面介绍
- ASP.NET Core MVC 中的视图
- ASP.NET Core Razor SDK

使用 ASP.NET Core 中的应用程序模型

2018/11/13 • • Edit Online

作者：Steve Smith

ASP.NET Core MVC 会定义一个*应用程序模型*，用于表示 MVC 应用的各个组件。通过读取和处理此模型可修改 MVC 元素的行为方式。默认情况下，MVC 遵循特定的约定，以确定将哪些类视作控制器，这些类上的哪些方法是操作，以及参数和路由的行为方式。你可以自定义此行为以满足应用的需要，方法如下：创建自己的约定，并将它们应用于全局或作为属性应用。

模型和提供程序

ASP.NET Core MVC 应用程序模型包括用于描述 MVC 应用程序的抽象接口和具体实现类。此模型是 MVC 根据默认约定发现应用的控制器、操作、操作参数、路由和筛选器的结果。通过使用应用程序模型，可以修改应用以遵循与默认 MVC 行为不同的约定。参数、名称、路由和筛选器都用作操作和控制器的配置数据。

ASP.NET Core MVC 应用程序模型具有以下结构：

- ApplicationModel
  - 控制器 (ControllerModel)
    - 操作 (ActionModel)
      - 参数 (ParameterModel)

该模型的每个级别都有权访问公用 `Properties` 集合，层次结构中的较低级别可以访问和覆盖由较高级别设置的属性值。创建操作时，属性保存到 `ActionDescriptor.Properties` 中。之后，当处理请求时，可通过 `ActionContext.ActionDescriptor.Properties` 访问某个约定添加或修改的任何属性。若要基于每项操作对筛选器、模型绑定器等进行配置，使用属性不失为一个好办法。

> **NOTE**
>
> 一旦完成应用启动，`ActionDescriptor.Properties` 集合就不再是线程安全的（针对写入）。约定是将数据安全添加到此集合的最佳方式。

IApplicationModelProvider

ASP.NET Core MVC 使用提供程序模式（由 IApplicationModelProvider 接口定义）加载应用程序模型。此部分介绍此提供程序的工作原理的一些内部实现细节。这是一项高级主题 — 利用应用程序模型的大多数应用应使用约定来执行此操作。

`IApplicationModelProvider` 接口的实现相互"包装"，每个实现都基于其 `Order` 属性以升序调用 `OnProvidersExecuting`。然后，按相反的顺序调用 `OnProvidersExecuted` 方法。该框架定义了多个提供程序：

首先（`Order=-1000`）：

- `DefaultApplicationModelProvider`

然后（`Order=-990`）：

- `AuthorizationApplicationModelProvider`
- `CorsApplicationModelProvider`

> **NOTE**
> 未定义具有相同 `Order` 值的两个提供程序的调用顺序，因此不应依赖此顺序。

> **NOTE**
> `IApplicationModelProvider` 是一种高级概念，框架创建者可对其进行扩展。一般情况下，应用应使用约定，而框架应使用提供程序。主要不同之处在于提供程序始终先于约定运行。

`DefaultApplicationModelProvider` 建立了由 ASP.NET Core MVC 使用的许多默认行为。其职责包括：

- 将全局筛选器添加到上下文
- 将控制器添加到上下文
- 将公共控制器方法作为操作添加
- 将操作方法参数添加到上下文
- 应用路由和其他属性

某些内置行为由 `DefaultApplicationModelProvider` 实现。此提供程序负责构造 `ControllerModel`，后者又引用 `ActionModel`、`PropertyModel` 和 `ParameterModel` 实例。`DefaultApplicationModelProvider` 类是一个内部框架实现细节，未来将对其进行更改。

`AuthorizationApplicationModelProvider` 负责应用与 `AuthorizeFilter` 和 `AllowAnonymousFilter` 属性关联的行为。详细了解这些属性。

`CorsApplicationModelProvider` 可实现与 `IEnableCorsAttribute`、`IDisableCorsAttribute` 和 `DisableCorsAuthorizationFilter` 关联的行为。详细了解 CORS。

约定

应用程序模型定义了约定抽象，通过约定抽象来自定义模型行为比重写整个模型或提供程序更简单。建议使用这些抽象来修改应用的行为。通过使用约定，可以编写能动态应用自定义项的代码。使用筛选器可以修改框架的行为，而利用自定义项可以控制整个应用连接在一起的方式。

可用约定如下：

- `IApplicationModelConvention`
- `IControllerModelConvention`
- `IActionModelConvention`
- `IParameterModelConvention`

可通过以下方式应用约定：将它们添加到 MVC 选项，或实现 `Attribute` 并将它们应用于控制器、操作或操作参数（类似于 `Filters`）。与筛选器不同的是，约定仅在应用启动时执行，而不作为每个请求的一部分执行。

示例：修改 ApplicationModel

以下约定用于向应用程序模型添加属性。

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ApplicationDescription : IApplicationModelConvention
    {
        private readonly string _description;

        public ApplicationDescription(string description)
        {
            _description = description;
        }

        public void Apply(ApplicationModel application)
        {
            application.Properties["description"] = _description;
        }
    }
}
```

当在 `Startup` 的 `ConfigureServices` 中添加 MVC 时，应用程序模型约定作为选项应用。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
        //options.Conventions.Add(new IdsMustBeInRouteParameterModelConvention());
    });
}
```

可从控制器操作内的 `ActionDescriptor` 属性集合中访问属性：

```
public class AppModelController : Controller
{
    public string Description()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

示例：修改 ControllerModel 说明

与上一个示例一样，也可以修改控制器模型，以包含自定义属性。这些属性将覆盖应用程序模型中指定的具有相同名称的现有属性。以下约定属性可在控制器级别添加说明：

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ControllerDescriptionAttribute : Attribute, IControllerModelConvention
    {
        private readonly string _description;

        public ControllerDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ControllerModel controllerModel)
        {
            controllerModel.Properties["description"] = _description;
        }
    }
}
```

此约定在控制器上作为属性应用。

```
[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
```

访问“description”属性的方式与前面示例中一样。

示例：修改 ActionModel 说明

可向各项操作应用不同的属性约定，并覆盖已在应用程序或控制器级别应用的行为。

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ActionDescriptionAttribute : Attribute, IActionModelConvention
    {
        private readonly string _description;

        public ActionDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ActionModel actionModel)
        {
            actionModel.Properties["description"] = _description;
        }
    }
}
```

通过将此约定应用于上一示例的控制器中的某项操作，演示了它如何覆盖控制器级别的约定：

```
[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }

    [ActionDescription("Action Description")]
    public string UseActionDescriptionAttribute()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

示例：修改 ParameterModel

可将以下约定应用于操作参数，以修改其 `BindingInfo` 。以下约定要求参数为路由参数；忽略其他可能的绑定源（比如查询字符串值）。

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace AppModelSample.Conventions
{
    public class MustBeInRouteParameterModelConvention : Attribute, IParameterModelConvention
    {
        public void Apply(ParameterModel model)
        {
            if (model.BindingInfo == null)
            {
                model.BindingInfo = new BindingInfo();
            }
            model.BindingInfo.BindingSource = BindingSource.Path;
        }
    }
}
```

该属性可应用于任何操作参数：

```
public class ParameterModelController : Controller
{
    // Will bind:  /ParameterModel/GetById/123
    // WON'T bind: /ParameterModel/GetById?id=123
    public string GetById([MustBeInRouteParameterModelConvention]int id)
    {
        return $"Bound to id: {id}";
    }
}
```

示例：修改 ActionModel 名称

以下约定可修改 `ActionModel` ，以更新其应用到的操作的*名称*。新名称以参数形式提供给该属性。此新名称供路由使用，因此它将影响用于访问此操作方法的路由。

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class CustomActionNameAttribute : Attribute, IActionModelConvention
    {
        private readonly string _actionName;

        public CustomActionNameAttribute(string actionName)
        {
            _actionName = actionName;
        }

        public void Apply(ActionModel actionModel)
        {
            // this name will be used by routing
            actionModel.ActionName = _actionName;
        }
    }
}
```

此属性应用于 `HomeController` 中的操作方法：

```
// Route: /Home/MyCoolAction
[CustomActionName("MyCoolAction")]
public string SomeName()
{
    return ControllerContext.ActionDescriptor.ActionName;
}
```

即使方法名称为 `SomeName`，该属性也会覆盖 MVC 使用该方法名称的约定，并将操作名称替换为 `MyCoolAction`。因此，用于访问此操作的路由为 `/Home/MyCoolAction`。

> **NOTE**
>
> 此示例本质上与使用内置 ActionName 属性相同。

示例：自定义路由约定

可以使用 `IApplicationModelConvention` 来自定义路由的工作方式。例如，以下约定会将控制器的命名空间合并到其路由中，并将命名空间中的 `.` 替换为路由中的 `/`：

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;

namespace AppModelSample.Conventions
{
    public class NamespaceRoutingConvention : IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            foreach (var controller in application.Controllers)
            {
                var hasAttributeRouteModels = controller.Selectors
                    .Any(selector => selector.AttributeRouteModel != null);

                if (!hasAttributeRouteModels
                    && controller.ControllerName.Contains("Namespace")) // affect one controller in this sample
                {
                    // Replace the . in the namespace with a / to create the attribute route
                    // Ex: MySite.Admin namespace will correspond to MySite/Admin attribute route
                    // Then attach [controller], [action] and optional {id?} token.
                    // [Controller] and [action] is replaced with the controller and action
                    // name to generate the final template
                    controller.Selectors[0].AttributeRouteModel = new AttributeRouteModel()
                    {
                        Template = controller.ControllerType.Namespace.Replace('.', '/') +
"/[controller]/[action]/{id?}"
                    };
                }
            }

            // You can continue to put attribute route templates for the controller actions depending on the
way you want them to behave
        }
    }
}
```

该约定作为一个选项添加到 Startup 中。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
        //options.Conventions.Add(new IdsMustBeInRouteParameterModelConvention());
    });
}
```

TIP

可以使用 `services.Configure<MvcOptions>(c => c.Conventions.Add(YOURCONVENTION));` 来访问 `MvcOptions`，以将约定添加到中间件

此示例将此约定应用于未使用属性路由的路由，其中，控制器名称包含“Namespace”。以下控制器演示了此约定：

```
using Microsoft.AspNetCore.Mvc;

namespace AppModelSample.Controllers
{
    public class NamespaceRoutingController : Controller
    {
        // using NamespaceRoutingConvention
        // route: /AppModelSample/Controllers/NamespaceRouting/Index
        public string Index()
        {
            return "This demonstrates namespace routing.";
        }
    }
}
```

应用程序模型在 WebApiCompatShim 中的使用

ASP.NET Core MVC 使用一组不同于 ASP.NET Web API 2 的约定。使用自定义约定，可以修改 ASP.NET Core MVC 应用的行为，使其与 Web API 应用保持一致。Microsoft 附带了专用于此的 WebApiCompatShim。

> **NOTE**
>
> 详细了解从 ASP.NET Web API 迁移。

若要使用 Web API Compatibility Shim，需将该包添加到项目中，然后通过调用 `Startup` 中的 `AddWebApiConventions`，将约定添加到 MVC：

```
services.AddMvc().AddWebApiConventions();
```

该填充程序提供的约定仅适用于应用中已应用特定属性的部分。以下四个属性用于控制哪些控制器应使用该填充程序的约定来修改自己的约定：

- UseWebApiActionConventionsAttribute
- UseWebApiOverloadingAttribute
- UseWebApiParameterConventionsAttribute
- UseWebApiRoutesAttribute

操作约定

`UseWebApiActionConventionsAttribute` 用于根据名称将 HTTP 方法映射到操作（例如，`Get` 将映射到 `HttpGet`）。它仅适用于不使用属性路由的操作。

重载

`UseWebApiOverloadingAttribute` 用于应用 `WebApiOverloadingApplicationModelConvention` 约定。此约定可向操作选择过程添加 `OverloadActionConstraint`，以将候选操作限制为其请求满足所有非可选参数的操作。

参数约定

`UseWebApiParameterConventionsAttribute` 用于应用 `WebApiParameterConventionsApplicationModelConvention` 操作约定。此约定指定用作操作参数的简单类型默认来自 URI，而复杂类型来自请求正文。

路由

`UseWebApiRoutesAttribute` 控制是否应用 `WebApiApplicationModelConvention` 控制器约定。启用后，此约定用于向路由添加对区域的支持。

除了一组约定外，该兼容性包还包含一个 `System.Web.Http.ApiController` 基类，用于替换 Web API 提供的等效项。

这允许针对 Web API 编写并且继承自 `ApiController` 的控制器在 ASP.NET Core MVC 上运行时，能够按照设计的方式运行。此控制器基类使用上面列出的所有 `UseWebApi*` 属性进行修饰。`ApiController` 公开了与在 Web API 中找到的属性、方法和结果类型兼容的属性、方法和结果类型。

使用 ApiExplorer 记录应用

应用程序模型在每个级别公开了 `ApiExplorer` 属性，该属性可用于遍历应用的结构。这可用于使用 Swagger 等工具为 Web API 生成帮助页。`ApiExplorer` 属性公开了 `IsVisible` 属性，后者可设置为指定应公开的应用模型部分。可以使用约定配置此设置：

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class EnableApiExplorerApplicationConvention : IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            application.ApiExplorer.IsVisible = true;
        }
    }
}
```

使用此方法（和附加约定，如有需要），可以在应用中的任何级别启用或禁用 API 可见性。

ASP.NET Core 中的筛选器

2018/12/14 • • Edit Online

作者：Rick Anderson、Tom Dykstra 和 Steve Smith

ASP.NET Core MVC 中的筛选器允许在请求处理管道中的特定阶段之前或之后运行代码。

> **IMPORTANT**
>
> 本主题不适用于 Razor 页面。ASP.NET Core 2.1 及更高版本支持适用于 Razor 页面的 IPageFilter 和 IAsyncPageFilter。有关详细信息，请参阅 Razor 页面的筛选方法。

内置筛选器处理任务，例如：

- 授权（防止用户访问未获授权的资源）。
- 确保所有请求都使用 HTTPS。
- 响应缓存（对请求管道进行短路出路，以便返回缓存的响应）。

可以创建自定义筛选器来处理横切关注点。筛选器可以避免跨操作复制代码。例如，错误处理异常筛选器可以合并错误处理。

查看或下载 GitHub 中的示例。

筛选器的工作原理

筛选器在 *MVC 操作调用管道*（有时称为*筛选器管道*）内运行。筛选器管道在 MVC 选择要执行的操作之后运行。

筛选器类型

每种筛选器类型都在筛选器管道中的不同阶段执行。

- 授权筛选器最先运行，用于确定是否已针对当前请求为当前用户授权。如果请求未获授权，它们可以让管道短路。
- 资源筛选器是授权后最先处理请求的筛选器。它们可以在筛选器管道的其余阶段运行之前以及管道的其余阶段完成之后运行代码。出于性能方面的考虑，可以使用它们来实现缓存或以其他方式让筛选器管道短路。它们在模型绑定之前运行，所以可以影响模型绑定。
- 操作筛选器可以在调用单个操作方法之前和之后立即运行代码。它们可用于处理传入某个操作的参数以及从该操作返回的结果。
- 异常筛选器用于在向响应正文写入任何内容之前，对未经处理的异常应用全局策略。
- 结果筛选器可以在执行单个操作结果之前和之后立即运行代码。仅当操作方法成功执行时，它们才会运行。对于必须围绕视图或格式化程序的执行的逻辑，它们很有用。

下图展示了这些筛选器类型在筛选器管道中的交互方式。

实现

筛选器通过不同的接口定义支持同步和异步实现。

可在其管道阶段之前和之后运行代码的同步筛选器定义 On*Stage*Executing 方法和 On*Stage*Executed 方法。例如，在调用操作方法之前调用 `OnActionExecuting`，在操作方法返回之后调用 `OnActionExecuted`。

```
using FiltersSample.Helper;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class SampleActionFilter : IActionFilter
    {
        public void OnActionExecuting(ActionExecutingContext context)
        {
            // do something before the action executes
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            // do something after the action executes
        }
    }
}
```

异步筛选器定义单一的 On*Stage*ExecutionAsync 方法。此方法采用 *FilterType*ExecutionDelegate 委托来执行筛选器的管道阶段。例如，`ActionExecutionDelegate` 调用该操作方法或下一个操作筛选器，用户可以在调用它之前和之后执行代码。

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class SampleAsyncActionFilter : IAsyncActionFilter
    {
        public async Task OnActionExecutionAsync(
            ActionExecutingContext context,
            ActionExecutionDelegate next)
        {
            // do something before the action executes
            var resultContext = await next();
            // do something after the action executes; resultContext.Result will be set
        }
    }
}
```

可以在单个类中为多个筛选器阶段实现接口。例如，ActionFilterAttribute 类实现 `IActionFilter` 和 `IResultFilter`，以及它们的异步等效接口。

> **NOTE**
>
> 筛选器接口的同步和异步版本任意实现一个，而不是同时实现。该框架会先查看筛选器是否实现了异步接口，如果是，则调用该接口。如果不是，则调用同步接口的方法。如果在一个类中同时实现了这两种接口，则仅调用异步方法。使用抽象类时，比如 ActionFilterAttribute，将为每种筛选器类型仅重写同步方法或仅重写异步方法。

IFilterFactory

IFilterFactory 实现 IFilterMetadata。因此，`IFilterFactory` 实例可在筛选器管道中的任意位置用作 `IFilterMetadata` 实例。当该框架准备调用筛选器时，它会尝试将其转换为 `IFilterFactory`。如果强制转换成功，则调用 CreateInstance 方法来创建将调用的 `IFilterMetadata` 实例。这提供了一种很灵活的设计，因为无需在应用启动时显式设置精确的筛选器管道。

用户可以在自己的属性实现上实现 `IFilterFactory` 作为另一种创建筛选器的方法：

```
public class AddHeaderWithFactoryAttribute : Attribute, IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new InternalAddHeaderFilter();
    }

    private class InternalAddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "Internal", new string[] { "Header Added" });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

内置筛选器属性

该框架包含许多可子类化和自定义的基于属性的内置筛选器。例如，以下结果筛选器会向响应添加标头。

```
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class AddHeaderAttribute : ResultFilterAttribute
    {
        private readonly string _name;
        private readonly string _value;

        public AddHeaderAttribute(string name, string value)
        {
            _name = name;
            _value = value;
        }

        public override void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                _name, new string[] { _value });
            base.OnResultExecuting(context);
        }
    }
}
```

属性允许筛选器采用参数，如上面的示例所示。可将此属性添加到控制器或操作方法，并指定 HTTP 标头的名称和值：

```
[AddHeader("Author", "Steve Smith @ardalis")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using developer tools.");
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header should be set.");
    }
```

`Index` 操作的结果如下所示：响应标头显示在右下角。

多种筛选器接口具有相应属性，这些属性可用作自定义实现的基类。

筛选器属性：

- `ActionFilterAttribute`
- `ExceptionFilterAttribute`
- `ResultFilterAttribute`
- `FormatFilterAttribute`
- `ServiceFilterAttribute`
- `TypeFilterAttribute`

本文稍后会对 `TypeFilterAttribute` 和 `ServiceFilterAttribute` 进行介绍。

筛选器作用域和执行顺序

可以将筛选器添加到管道中的三个*作用域*之一。可以使用属性将筛选器添加到特定的操作方法或控制器类。或者，也可以注册所有控制器和操作的全局筛选器。通过将筛选器添加到 `ConfigureServices` 中的 `MvcOptions.Filters` 集合，可以将其添加为全局筛选器：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",
            "Result filter added to MvcOptions.Filters")); // an instance
        options.Filters.Add(typeof(SampleActionFilter)); // by type
        options.Filters.Add(new SampleGlobalActionFilter()); // an instance
    });

    services.AddScoped<AddHeaderFilterWithDi>();
}
```

默认执行顺序

当管道的某个特定阶段有多个筛选器时，作用域可确定筛选器执行的默认顺序。全局筛选器涵盖类筛选器，类筛选器又涵盖方法筛选器。这种模式有时称为“俄罗斯套娃”嵌套，因为增加的每个作用域都包装在前一个作用域中，就像套娃一样。通常情况下，无需显式确定排序便可获得所需的重写行为。

在这种嵌套模式下，筛选器的 *after* 代码会按照与 *before* 代码相反的顺序运行。其序列如下所示：

- 筛选器的 *before* 代码应用于全局
  - 筛选器的 *before* 代码应用于控制器
    - 筛选器的 *before* 代码应用于操作方法
    - 筛选器的 *after* 代码应用于操作方法
  - 筛选器的 *after* 代码应用于控制器
- 筛选器的 *after* 代码应用于全局

下面的示例阐释了为同步操作筛选器调用筛选器方法的顺序。

序列	筛选器作用域	筛选器方法
1	Global	`OnActionExecuting`
2	控制器	`OnActionExecuting`
3	方法	`OnActionExecuting`
4	方法	`OnActionExecuted`
5	控制器	`OnActionExecuted`
6	Global	`OnActionExecuted`

此序列显示：

- 方法筛选器已嵌套在控制器筛选器中。
- 控制器筛选器已嵌套在全局筛选器中。

换句话说，如果处于异步筛选器的 On*Stage*ExecutionAsync 方法内，则当代码位于堆栈上时，所有筛选器都在更严格的作用域中运行。

> **NOTE**
> 继承自 `Controller` 基类的每个控制器都包括 `OnActionExecuting` 和 `OnActionExecuted` 方法。这些方法包装针对某项给定操作运行的筛选器：`OnActionExecuting` 在所有筛选器之前调用，`OnActionExecuted` 在所有筛选器之后调用。

重写默认顺序

可以通过实现 `IOrderedFilter` 来重写默认执行序列。此接口公开了一个 `Order` 属性来确定执行顺序，该属性优先于作用域。具有较低 `Order` 值的筛选器会在具有较高 `Order` 值的筛选器之前执行其 *before* 代码。具有较低 `Order` 值的筛选器会在具有较高 `Order` 值的筛选器之后执行其 *after* 代码。可使用构造函数参数来设置 `Order` 属性：

```
[MyFilter(Name = "Controller Level Attribute", Order=1)]
```

如果具有上述示例中所示的 3 个相同的操作筛选器，但将控制器和全局筛选器的 `Order` 属性分别设置为 1 和 2，则会反转执行顺序。

序列	筛选器作用域	`ORDER` 属性	筛选器方法
1	方法	0	`OnActionExecuting`
2	控制器	1	`OnActionExecuting`
3	Global	2	`OnActionExecuting`
4	Global	2	`OnActionExecuted`
5	控制器	1	`OnActionExecuted`
6	方法	0	`OnActionExecuted`

在确定筛选器的运行顺序时，`Order` 属性优先于作用域。先按顺序对筛选器排序，然后使用作用域消除并列问题。所有内置筛选器实现 `IOrderedFilter` 并将默认 `Order` 值设为 0。对于内置筛选器，作用域会确定顺序，除非将 `Order` 设为非零值。

取消和设置短路

通过设置提供给筛选器方法的 `context` 参数上的 `Result` 属性，可以在筛选器管道的任意位置设置短路。例如，以下资源筛选器将阻止执行管道的其余阶段。

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class ShortCircuitingResourceFilterAttribute : Attribute,
            IResourceFilter
    {
        public void OnResourceExecuting(ResourceExecutingContext context)
        {
            context.Result = new ContentResult()
            {
                Content = "Resource unavailable - header should not be set"
            };
        }

        public void OnResourceExecuted(ResourceExecutedContext context)
        {
        }
    }
}
```

在下面的代码中，`ShortCircuitingResourceFilter` 和 `AddHeader` 筛选器都以 `SomeResource` 操作方法为目标。`ShortCircuitingResourceFilter`：

- 先运行，因为它是资源筛选器且 `AddHeader` 是操作筛选器。
- 对管道的其余部分进行短路处理。

这样 `AddHeader` 筛选器就不会为 `SomeResource` 操作运行。如果这两个筛选器都应用于操作方法级别，只要 `ShortCircuitingResourceFilter` 先运行，此行为就不会变。先运行 `ShortCircuitingResourceFilter`（考虑到它的筛选器类型），或显式使用 `Order` 属性。

```
[AddHeader("Author", "Steve Smith @ardalis")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using developer tools.");
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header should be set.");
    }
```

依赖关系注入

可按类型或实例添加筛选器。如果添加实例，该实例将用于每个请求。如果添加类型，则将激活该类型，这意味着将为每个请求创建一个实例，并且依赖关系注入 (DI) 将填充所有构造函数依赖项。按类型添加筛选器等效于 `filters.Add(new TypeFilterAttribute(typeof(MyFilter)))`。

如果将筛选器作为属性实现并直接添加到控制器类或操作方法中，则该筛选器不能由依赖关系注入 (DI) 提供构造函数依赖项。这是因为属性在应用时必须提供自己的构造函数参数。这是属性工作原理上的限制。

如果筛选器具有一些需要从 DI 访问的依赖项，有几种受支持的方法可用。可以使用以下接口之一，将筛选器应用于类或操作方法：

- `ServiceFilterAttribute`

- `TypeFilterAttribute`
- 在属性上实现的 `IFilterFactory`

> **NOTE**
>
> 记录器就是一种可能需要从 DI 获取的依赖项。但是，应避免单纯为进行日志记录而创建和使用筛选器，因为内置的框架日志记录功能可能已经提供用户所需。如果要将日志记录功能添加到筛选器，它应重点关注业务领域问题或特定于筛选器的行为，而非 MVC 操作或其他框架事件。

ServiceFilterAttribute

在 DI 中注册服务筛选器实现类型。`ServiceFilterAttribute` 可从 DI 检索筛选器实例。将 `ServiceFilterAttribute` 添加到 `Startup.ConfigureServices` 中的容器中，并在 `[ServiceFilter]` 属性中引用它：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",
            "Result filter added to MvcOptions.Filters")); // an instance
        options.Filters.Add(typeof(SampleActionFilter)); // by type
        options.Filters.Add(new SampleGlobalActionFilter()); // an instance
    });

    services.AddScoped<AddHeaderFilterWithDi>();
}
```

```
[ServiceFilter(typeof(AddHeaderFilterWithDi))]
public IActionResult Index()
{
    return View();
}
```

使用 `ServiceFilterAttribute` 时，`IsReusable` 设置会提示：筛选器实例可能在其创建的请求范围之外被重用。该框架不保证在稍后的某个时刻将创建筛选器的单个实例，或不会从 DI 容器重新请求筛选器。如果使用的筛选器依赖于具有除单一实例以外的生命周期的服务，请避免使用 `IsReusable`。

使用 `ServiceFilterAttribute` 时不注册筛选器类型会引发异常：

```
System.InvalidOperationException: No service for type
'FiltersSample.Filters.AddHeaderFilterWithDI' has been registered.
```

`ServiceFilterAttribute` 可实现 `IFilterFactory`。`IFilterFactory` 公开用于创建 `IFilterMetadata` 实例的 `CreateInstance` 方法。`CreateInstance` 方法从服务容器 (DI) 中加载指定的类型。

TypeFilterAttribute

`TypeFilterAttribute` 与 `ServiceFilterAttribute` 类似，但不会直接从 DI 容器解析其类型。它使用 `Microsoft.Extensions.DependencyInjection.ObjectFactory` 对类型进行实例化。

由于存在这种差异，所以存在以下情况：

- 使用 `TypeFilterAttribute` 引用的类型不需要先注册在容器中。它们具备由容器实现的依赖项。
- `TypeFilterAttribute` 可以选择为类型接受构造函数参数。

使用 `TypeFilterAttribute` 时，`IsReusable` 设置会提示：筛选器实例可能在其创建的请求范围之外被重用。

该框架不保证将创建筛选器的单一实例。如果使用的筛选器依赖于具有除单一实例以外的生命周期的服务，请避免使用 `IsReusable`。

下面的示例演示如何使用 `TypeFilterAttribute` 将参数传递到类型：

```
[TypeFilter(typeof(AddHeaderAttribute),
    Arguments = new object[] { "Author", "Steve Smith (@ardalis)" })]
public IActionResult Hi(string name)
{
    return Content($"Hi {name}");
}
```

在属性上实现 IFilterFactory

如果你的筛选器符合以下描述：

- 不需要任何参数。
- 具备需要由 DI 填充的构造函数依赖项。

在类和方法上可以不使用 `[TypeFilter(typeof(FilterType))]` 改用自己命名的属性。下面的筛选器展示了如何实现此操作：

```
public class SampleActionFilterAttribute : TypeFilterAttribute
{
    public SampleActionFilterAttribute():base(typeof(SampleActionFilterImpl))
    {
    }

    private class SampleActionFilterImpl : IActionFilter
    {
        private readonly ILogger _logger;
        public SampleActionFilterImpl(ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<SampleActionFilterAttribute>();
        }

        public void OnActionExecuting(ActionExecutingContext context)
        {
            _logger.LogInformation("Business action starting...");
            // perform some business logic work

        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            // perform some business logic work
            _logger.LogInformation("Business action completed.");
        }
    }
}
```

可以使用 `[SampleActionFilter]` 语法将此筛选器应用于类或方法，而不必使用 `[TypeFilter]` 或 `[ServiceFilter]`。

授权筛选器

授权筛选器：

- 控制对操作方法的访问。
- 是筛选器管道中要执行的第一个筛选器。
- 具有在它之前的执行的方法，但没有之后执行的方法。

用户只有在编写自己的授权框架时，才应编写自定义授权筛选器。建议配置授权策略或编写自定义授权策略，而不是编写自定义筛选器。内置筛选器实现只负责调用授权系统。

切勿在授权筛选器内引发异常，因为没有任何能处理该异常的组件（异常筛选器不会进行处理）。在出现异常时请小心应对。

详细了解授权。

资源筛选器

- 实现 `IResourceFilter` 或 `IAsyncResourceFilter` 接口，
- 它们的执行会覆盖筛选器管道的绝大部分。
- 只有授权筛选器在资源筛选器之前运行。

如果需要使某个请求正在执行的大部分工作短路，资源筛选器会很有用。例如，如果响应在缓存中，则缓存筛选器可以绕开管道的其余阶段。

前面所示的短路资源筛选器便是一种资源筛选器。另一个示例是 DisableFormValueModelBindingAttribute：

- 可以防止模型绑定访问表单数据。
- 如果要上传大型文件，同时想防止表单被读入内存，那么此筛选器会很有用。

操作筛选器

操作筛选器：

- 实现 `IActionFilter` 或 `IAsyncActionFilter` 接口。
- 它们的执行围绕着操作方法的执行。

下面是一个操作筛选器示例：

```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // do something before the action executes
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // do something after the action executes
    }
}
```

ActionExecutingContext 提供以下属性：

- `ActionArguments`：用于处理对操作的输入。
- `Controller`：用于处理控制器实例。
- `Result`：设置此属性会使操作方法和后续操作筛选器的执行短路。引发异常也会阻止操作方法和后续筛选器的执行，但会被视为失败，而不是一个成功的结果。

ActionExecutedContext 提供 `Controller` 和 `Result` 以及下列属性：

- `Canceled`：如果操作执行已被另一个筛选器设置短路，则为 true。
- `Exception`：如果操作或后续操作筛选器引发了异常，则为非 NULL 值。将此属性设置为 NULL 可有效地“处理”异常，并且将执行 `Result`，就像它是从操作方法正常返回的一样。

对于 `IAsyncActionFilter`，一个向 `ActionExecutionDelegate` 的调用可以达到以下目的：

- 执行所有后续操作筛选器和操作方法。
- 返回 `ActionExecutedContext`。

若要设置短路，可将 `ActionExecutingContext.Result` 分配到某个结果实例，并且不调用 `ActionExecutionDelegate`。

该框架提供一个可子类化的抽象 `ActionFilterAttribute`。

操作筛选器可用于验证模型状态，并在状态为无效时返回任何错误：

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class ValidateModelAttribute : ActionFilterAttribute
    {
        public override void OnActionExecuting(ActionExecutingContext context)
        {
            if (!context.ModelState.IsValid)
            {
                context.Result = new BadRequestObjectResult(context.ModelState);
            }
        }
    }
}
```

`OnActionExecuted` 方法在操作方法之后运行，可通过 `ActionExecutedContext.Result` 属性查看和处理操作结果。如果操作执行已被另一个筛选器设置短路，则 `ActionExecutedContext.Canceled` 设置为 true。如果操作或后续操作筛选器引发了异常，则 `ActionExecutedContext.Exception` 设置为非 NULL 值。将 `ActionExecutedContext.Exception` 设置为 null：

- 有效地"处理"异常。
- 执行 `ActionExectedContext.Result`，从操作方法中将它正常返回。

异常筛选器

*异常筛选器*可实现 `IExceptionFilter` 或 `IAsyncExceptionFilter` 接口。它们可用于为应用实现常见的错误处理策略。

下面的异常筛选器示例使用自定义开发人员错误视图，显示在开发应用时发生的异常的相关详细信息：

```
public class CustomExceptionFilterAttribute : ExceptionFilterAttribute
{
    private readonly IHostingEnvironment _hostingEnvironment;
    private readonly IModelMetadataProvider _modelMetadataProvider;

    public CustomExceptionFilterAttribute(
        IHostingEnvironment hostingEnvironment,
        IModelMetadataProvider modelMetadataProvider)
    {
        _hostingEnvironment = hostingEnvironment;
        _modelMetadataProvider = modelMetadataProvider;
    }

    public override void OnException(ExceptionContext context)
    {
        if (!_hostingEnvironment.IsDevelopment())
        {
            // do nothing
            return;
        }
        var result = new ViewResult {ViewName = "CustomError"};
        result.ViewData = new ViewDataDictionary(_modelMetadataProvider,context.ModelState);
        result.ViewData.Add("Exception", context.Exception);
        // TODO: Pass additional detailed data via ViewData
        context.Result = result;
    }
}
```

异常筛选器：

- 没有之前和之后的事件。
- 实现 `OnException` 或 `OnExceptionAsync`。
- 处理控制器创建、模型绑定、操作筛选器或操作方法中发生的未经处理的异常。
- 请不要捕获资源筛选器、结果筛选器或 MVC 结果执行中发生的异常。

若要处理异常，请将 `ExceptionContext.ExceptionHandled` 属性设置为 true，或编写响应。这将停止传播异常。异常筛选器无法将异常转变为“成功”。只有操作筛选器才能执行该转变。

> **NOTE**
> 在 ASP.NET Core 1.1 中，如果将 `ExceptionHandled` 设置为 true 并编写响应，则不会发送响应。在这种情况下，ASP.NET Core 1.0 不发送响应，ASP.NET Core 1.1.2 则恢复为 1.0 的行为。有关详细信息，请参阅 GitHub 存储库中的问题编号 5594。

异常筛选器：

- 非常适合捕获发生在 MVC 操作中的异常。
- 并不像错误处理中间件那么灵活。

建议使用中间件处理异常。仅在需要根据所选 MVC 操作以不同方式执行错误处理时，才使用异常筛选器。例如，应用可能具有用于 API 终结点和视图/HTML 的操作方法。API 终结点可能返回 JSON 形式的错误信息，而基于视图的操作可能返回 HTML 形式的错误页。

`ExceptionFilterAttribute` 可以子类化。

结果筛选器

- 实现 `IResultFilter` 或 `IAsyncResultFilter` 接口。
- 它们的执行围绕着操作结果的执行。

下面是一个添加 HTTP 标头的结果筛选器示例。

```
public class AddHeaderFilterWithDi : IResultFilter
{
    private ILogger _logger;
    public AddHeaderFilterWithDi(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<AddHeaderFilterWithDi>();
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        var headerName = "OnResultExecuting";
        context.HttpContext.Response.Headers.Add(
            headerName, new string[] { "ResultExecutingSuccessfully" });
        _logger.LogInformation($"Header added: {headerName}");
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Can't add to headers here because response has already begun.
    }
}
```

要执行的结果类型取决于所执行的操作。返回视图的 MVC 操作会将所有 Razor 处理作为要执行的 `ViewResult` 的一部分。API 方法可能会将某些序列化操作作为结果执行的一部分。详细了解操作结果

当操作或操作筛选器生成操作结果时，仅针对成功的结果执行结果筛选器。当异常筛选器处理异常时，不执行结果筛选器。

`OnResultExecuting` 方法可以将 `ResultExecutingContext.Cancel` 设置为 true，使操作结果和后续结果筛选器的执行短路。设置短路时，通常应写入响应对象，以免生成空响应。如果引发异常，则会导致：

- 阻止操作结果和后续筛选器的执行。
- 结果被视为失败而不是成功。

当 `OnResultExecuted` 方法运行时，响应可能已发送到客户端，而且不能再更改(除非引发了异常)。如果操作结果执行已被另一个筛选器设置短路，则 `ResultExecutedContext.Canceled` 设置为 true。

如果操作结果或后续结果筛选器引发了异常，则 `ResultExecutedContext.Exception` 设置为非 NULL 值。将 `Exception` 设置为 NULL 可有效地"处理"异常，并防止 MVC 在管道的后续阶段重新引发该异常。在处理结果筛选器中的异常时，可能无法向响应写入任何数据。如果操作结果在其执行过程中引发异常，并且标头已刷新到客户端，则没有任何可靠的机制可用于发送失败代码。

对于 `IAsyncResultFilter`，通过调用 `ResultExecutionDelegate` 上的 `await next` 可执行所有后续结果筛选器和操作结果。若要设置短路，可将 `ResultExecutingContext.Cancel` 设置为 true，并且不调用 `ResultExectionDelegate`。

该框架提供一个可子类化的抽象 `ResultFilterAttribute`。前面所示的 AddHeaderAttribute 类是一种结果筛选器属性。

在筛选器管道中使用中间件

资源筛选器的工作方式与中间件类似，即涵盖管道中的所有后续执行。但筛选器又不同于中间件，它们是 MVC 的一部分，这意味着它们有权访问 MVC 上下文和构造。

在 ASP.NET Core 1.1 中，可以在筛选器管道中使用中间件。如果有一个中间件组件，该组件需要访问 MVC 路由数据，或者只能针对特定控制器或操作运行，则可能需要这样做。

若要将中间件用作筛选器，可创建一个具有 `Configure` 方法的类型，该方法可指定要注入到筛选器管道的中

间件。下面的示例使用本地化中间件为请求建立当前区域性：

```
public class LocalizationPipeline
{
    public void Configure(IApplicationBuilder applicationBuilder)
    {
        var supportedCultures = new[]
        {
            new CultureInfo("en-US"),
            new CultureInfo("fr")
        };

        var options = new RequestLocalizationOptions
        {

            DefaultRequestCulture = new RequestCulture(culture: "en-US", uiCulture: "en-US"),
            SupportedCultures = supportedCultures,
            SupportedUICultures = supportedCultures
        };
        options.RequestCultureProviders = new[]
            { new RouteDataRequestCultureProvider() { Options = options } };

        applicationBuilder.UseRequestLocalization(options);
    }
}
```

然后，可以使用 `MiddlewareFilterAttribute` 为所选控制器或操作或者在全局范围内运行中间件：

```
[Route("{culture}/[controller]/[action]")]
[MiddlewareFilter(typeof(LocalizationPipeline))]
public IActionResult CultureFromRouteData()
{
    return Content($"CurrentCulture:{CultureInfo.CurrentCulture.Name},"
        + $"CurrentUICulture:{CultureInfo.CurrentUICulture.Name}");
}
```

中间件筛选器与资源筛选器在筛选器管道的相同阶段运行，即，在模型绑定之前以及管道的其余阶段之后。

后续操作

若要尝试使用筛选器，请下载、测试并修改该示例。

ASP.NET Core 中的区域

2018/9/10 • • Edit Online

作者：Dhananjay Kumar 和 Rick Anderson

区域是 ASP.NET MVC 功能，用于将相关功能以单独的名称空间（用于路由）和文件夹结构（用于视图）的形式组织到一个组中。使用区域，将通过为 `controller` 和 `action` 添加另一个路由参数 `area`，创建用于路由目的的层次结构。

区域提供了将大型 ASP.NET Core MVC Web 应用分区为较小功能分组的方法。区域实际上是应用程序内的一个 MVC 结构。在 MVC 项目中，模型、控制器和视图等逻辑组件保存在不同的文件夹中，MVC 使用命名约定来创建这些组件之间的关系。对于大型应用，将应用分区为独立的高级功能区域可能更有利。例如，具有多个业务单位（如结账、计费、搜索等）的电子商务应用。每个单位都有自己的逻辑组件视图、控制器和模型。在此方案中，可使用区域对同一项目中的业务组件进行物理分区。

区域可以说是 ASP.NET Core MVC 项目中的较小功能单位，它具有自己的一组控制器、视图和模型。

以下情况，请考虑在 MVC 项目中使用区域：

- 应用程序由应进行逻辑区分的多个高级功能组件组成
- 想对 MVC 项目进行分区，使每个功能区域可以独立工作

区域特性：

- 一个 ASP.NET Core MVC 应用可以有任意数量的区域。
- 每个区域都有自己的控制器、模型和视图。
- 区域可用于将大型 MVC 项目组织为可以独立工作的多个高级组件。
- 区域支持具有相同名称的多个控制器 - 只要它们具有不同的区域。

我们来看看演示如何创建和使用区域的示例。假设你有一个商店应用，它有两组不同的控制器和视图：产品和服务。使用 MVC 区域的典型文件夹结构如下所示：

- 项目名称
  - 区域
    - 产品
      - Controllers
        - HomeController.cs
        - ManageController.cs
      - 视图
        - 主页
          - Index.cshtml
        - 管理
          - Index.cshtml
    - 服务

- Controllers
  - HomeController.cs
- 视图
  - 主页
    - Index.cshtml

当 MVC 尝试呈现 Area 中的视图时，它将默认尝试检查以下位置：

```
/Areas/<Area-Name>/Views/<Controller-Name>/<Action-Name>.cshtml
   /Areas/<Area-Name>/Views/Shared/<Action-Name>.cshtml
   /Views/Shared/<Action-Name>.cshtml
```

以上是默认位置，可以通过 `Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions` 上的 `AreaViewLocationFormats` 进行更改。

例如，在下面的代码中，文件夹名称不是“Area”，而是改为了“Categories”。

```
services.Configure<RazorViewEngineOptions>(options =>
   {
       options.AreaViewLocationFormats.Clear();
       options.AreaViewLocationFormats.Add("/Categories/{2}/Views/{1}/{0}.cshtml");
       options.AreaViewLocationFormats.Add("/Categories/{2}/Views/Shared/{0}.cshtml");
       options.AreaViewLocationFormats.Add("/Views/Shared/{0}.cshtml");
   });
```

值得注意的一点是，这里唯独看重的是 Views 文件夹的结构，而 Controllers 和 Models 等其他文件夹则无关紧要。例如，完全可以不设 Controllers 和 Models 文件夹。这是可行的，因为 Controllers 和 Models 的内容只是请求相应视图时才会编译到 .dll 中的代码，而 Views 的内容则不是。

定义文件夹层次结构后，需告知 MVC 每个控制器与一个区域相关联。可使用 `[Area]` 属性修饰控制器名称，来完成此操作。

```
...
   namespace MyStore.Areas.Products.Controllers
   {
       [Area("Products")]
       public class HomeController : Controller
       {
           // GET: /Products/Home/Index
           public IActionResult Index()
           {
               return View();
           }

           // GET: /Products/Home/Create
           public IActionResult Create()
           {
               return View();
           }
       }
   }
```

设置适用于新创建区域的路由定义。路由到控制器操作文章详细介绍了如何创建路由定义，包括使用常规路由和属性路由。在此示例中，我们将使用常规路由。为此，请打开 Startup.cs 文件，并通过添加以下名为 `areaRoute` 的路由定义修改文件。

```
...
   app.UseMvc(routes =>
   {
     routes.MapRoute(
         name: "areaRoute",
         template: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

     routes.MapRoute(
         name: "default",
         template: "{controller=Home}/{action=Index}/{id?}");
   });
```

浏览到 `http://<yourApp>/products`，将调用 `Products` 区域中 `HomeController` 的 `Index` 操作方法。

链接生成

- 生成从基于区域的控制器内的动作到同一控制器内另一个动作的链接。

  假设当前的请求路径类似于 `/Products/Home/Create`

  HtmlHelper 语法：`@Html.ActionLink("Go to Product's Home Page", "Index")`

  TagHelper 语法：`<a asp-action="Index">Go to Product's Home Page</a>`

  请注意，我们不需要提供“area”和“controller”值，因为它们在当前请求的上下文中已经可用。此类值称为 `ambient` 值。

- 生成从基于区域的控制器内的动作到不同控制器上另一个动作的链接

  假设当前的请求路径类似于 `/Products/Home/Create`

  HtmlHelper 语法：`@Html.ActionLink("Go to Manage Products Home Page", "Index", "Manage")`

  TagHelper 语法：`<a asp-controller="Manage" asp-action="Index">Go to Manage Products Home Page</a>`

  请注意，此处使用“area”的环境值，但上面显式指定了“controller”值。

- 生成从基于区域的控制器内的动作到不同区域中不同控制器上另一个动作的链接。

  假设当前的请求路径类似于 `/Products/Home/Create`

  HtmlHelper 语法：
  `@Html.ActionLink("Go to Services Home Page", "Index", "Home", new { area = "Services" })`

  TagHelper 语法：
  `<a asp-area="Services" asp-controller="Home" asp-action="Index">Go to Services Home Page</a>`

  请注意，此处不使用环境值。

- 生成从基于区域的控制器内的动作到不在区域中的不同控制器上另一个动作的链接。

  HtmlHelper 语法：
  `@Html.ActionLink("Go to Manage Products Home Page", "Index", "Home", new { area = "" })`

  TagHelper 语法：
  `<a asp-area="" asp-controller="Manage" asp-action="Index">Go to Manage Products Home Page</a>`

  由于我们要生成到基于非区域的控制器操作的链接，因此在此清空“area”的环境值。

发布区域

当 .csproj 文件中包含 `<Project Sdk="Microsoft.NET.Sdk.Web">` 时，所有 `*.cshtml` 和 `wwwroot/**` 文件将发布到输出。

ASP.NET Core 中的应用程序部件

2018/10/30 • • Edit Online

查看或下载示例代码(如何下载)

*应用程序部件*是应用程序资源的一种抽象，可通过它发现控制器、视图组件或标记帮助程序等 MVC 功能。AssemblyPart 就是一种应用程序部件，用于封装程序集引用以及公开类型和编译引用。*功能提供程序*使用应用程序部件填充 ASP.NET Core MVC 应用的功能。应用程序部件的主要用例是允许将应用配置为从程序集中发现(或避免加载)MVC 功能。

应用程序部件简介

MVC 应用从应用程序部件中加载其功能。具体而言，AssemblyPart 类表示受程序集支持的应用程序部件。可以使用这些类发现和加载 MVC 功能，比如控制器、视图组件、标记帮助程序和 Razor 编译源。ApplicationPartManager 负责跟踪可用于 MVC 应用的应用程序部件和功能提供程序。配置 MVC 时，可以与 `Startup` 中的 `ApplicationPartManager` 交互：

```
// create an assembly part from a class's assembly
var assembly = typeof(Startup).GetTypeInfo().Assembly;
services.AddMvc()
    .AddApplicationPart(assembly);

// OR
var assembly = typeof(Startup).GetTypeInfo().Assembly;
var part = new AssemblyPart(assembly);
services.AddMvc()
    .ConfigureApplicationPartManager(apm => apm.ApplicationParts.Add(part));
```

默认情况下，MVC 将搜索依赖项树并查找控制器(甚至在其他程序集中)。若要加载任意程序集(例如，从在编译时未引用的插件)，可以使用应用程序部件。

可以使用应用程序部件*避免*查找特定程序集或位置中的控制器。可以通过修改 `ApplicationPartManager` 的 `ApplicationParts` 集合，控制应用可用的部件(或程序集)。`ApplicationParts` 集合中条目的顺序并不重要。重要的是在使用 `ApplicationPartManager` 配置容器中的服务之前，对该类进行完全配置。例如，应在调用 `AddControllersAsServices` 之前完全配置 `ApplicationPartManager`。如果未进行完全配置，就意味着，在该方法调用之后添加的应用程序部件中的控制器不受影响(不注册为服务)，这可能导致不正确的应用程序行为。

如果某个程序集包含你不想使用的控制器，则将该程序集从 `ApplicationPartManager` 中删除：

```
services.AddMvc()
    .ConfigureApplicationPartManager(apm =>
    {
        var dependentLibrary = apm.ApplicationParts
            .FirstOrDefault(part => part.Name == "DependentLibrary");

        if (dependentLibrary != null)
        {
           apm.ApplicationParts.Remove(dependentLibrary);
        }
    })
```

除了项目的程序集及其从属程序集，`ApplicationPartManager` 还默认包含 `Microsoft.AspNetCore.Mvc.TagHelpers` 和 `Microsoft.AspNetCore.Mvc.Razor` 的部件。

应用程序功能提供程序

应用程序功能提供程序用于检查应用程序部件，并为这些部件提供功能。以下 MVC 功能有内置功能提供程序：

- 控制器
- 元数据引用
- 标记帮助程序
- 视图组件

功能提供程序从 `IApplicationFeatureProvider<T>` 继承，其中 `T` 是功能的类型。你可以为上面列出的任意 MVC 功能类型实现自己的功能提供程序。`ApplicationPartManager.FeatureProviders` 集合中功能提供程序的顺序可能很重要，因为靠后的提供程序可以对前面的提供程序所执行的操作作出反应。

示例：泛型控制器功能

默认情况下，ASP.NET Core MVC 会忽略泛型控制器（例如，`SomeController<T>`）。此示例使用的控制器功能提供程序在默认提供程序后面运行并为指定的类型列表（在 `EntityTypes.Types` 中定义）添加泛型控制器实例：

```
public class GenericControllerFeatureProvider : IApplicationFeatureProvider<ControllerFeature>
{
    public void PopulateFeature(IEnumerable<ApplicationPart> parts, ControllerFeature feature)
    {
        // This is designed to run after the default ControllerTypeProvider,
        // so the list of 'real' controllers has already been populated.
        foreach (var entityType in EntityTypes.Types)
        {
            var typeName = entityType.Name + "Controller";
            if (!feature.Controllers.Any(t => t.Name == typeName))
            {
                // There's no 'real' controller for this entity, so add the generic version.
                var controllerType = typeof(GenericController<>)
                    .MakeGenericType(entityType.AsType()).GetTypeInfo();
                feature.Controllers.Add(controllerType);
            }
        }
    }
}
```

实体类型：

```
public static class EntityTypes
{
    public static IReadOnlyList<TypeInfo> Types => new List<TypeInfo>()
    {
        typeof(Sprocket).GetTypeInfo(),
        typeof(Widget).GetTypeInfo(),
    };

    public class Sprocket { }
    public class Widget { }
}
```

将该功能提供程序添加到 `Startup` 中：

```
services.AddMvc()
    .ConfigureApplicationPartManager(apm =>
        apm.FeatureProviders.Add(new GenericControllerFeatureProvider()));
```

默认情况下，用于路由的泛型控制器名称的格式为 *GenericController`1[Widget]*，而不是 *Widget*。以下属性用于修改该名称，以便与控制器使用的泛型类型对应：

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System;

namespace AppPartsSample
{
    // Used to set the controller name for routing purposes. Without this convention the
    // names would be like 'GenericController`1[Widget]' instead of 'Widget'.
    //
    // Conventions can be applied as attributes or added to MvcOptions.Conventions.
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = true)]
    public class GenericControllerNameConvention : Attribute, IControllerModelConvention
    {
        public void Apply(ControllerModel controller)
        {
            if (controller.ControllerType.GetGenericTypeDefinition() !=
                typeof(GenericController<>))
            {
                // Not a GenericController, ignore.
                return;
            }

            var entityType = controller.ControllerType.GenericTypeArguments[0];
            controller.ControllerName = entityType.Name;
        }
    }
}
```

`GenericController` 类：

```
using Microsoft.AspNetCore.Mvc;

namespace AppPartsSample
{
    [GenericControllerNameConvention] // Sets the controller name based on typeof(T).Name
    public class GenericController<T> : Controller
    {
        public IActionResult Index()
        {
            return Content($"Hello from a generic {typeof(T).Name} controller.");
        }
    }
}
```

当请求匹配的路由时，结果如下：

示例：显示可用功能

可循环访问可用于应用的已填充功能，方法为通过依赖关系注入请求 `ApplicationPartManager`，并用它来填充相应功能的实例：

```
using AppPartsSample.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using System.Linq;
using Microsoft.AspNetCore.Mvc.Razor.Compilation;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.ViewComponents;

namespace AppPartsSample.Controllers
{
    public class FeaturesController : Controller
    {
        private readonly ApplicationPartManager _partManager;

        public FeaturesController(ApplicationPartManager partManager)
        {
            _partManager = partManager;
        }

        public IActionResult Index()
        {
            var viewModel = new FeaturesViewModel();

            var controllerFeature = new ControllerFeature();
            _partManager.PopulateFeature(controllerFeature);
            viewModel.Controllers = controllerFeature.Controllers.ToList();

            var metaDataReferenceFeature = new MetadataReferenceFeature();
            _partManager.PopulateFeature(metaDataReferenceFeature);
            viewModel.MetadataReferences = metaDataReferenceFeature.MetadataReferences
                                            .ToList();

            var tagHelperFeature = new TagHelperFeature();
            _partManager.PopulateFeature(tagHelperFeature);
            viewModel.TagHelpers = tagHelperFeature.TagHelpers.ToList();

            var viewComponentFeature = new ViewComponentFeature();
            _partManager.PopulateFeature(viewComponentFeature);
            viewModel.ViewComponents = viewComponentFeature.ViewComponents.ToList();

            return View(viewModel);
        }
    }
}
```

示例输出:

Home Page - AppPartSa

localhost:28431/Features

Features

Controllers:

- FeaturesController
- HomeController
- HelloController
- GenericController`1
- GenericController`1

Metadata References:

- C:\dev\ssmith\github.com\aspnet-docs\aspnetcore\mvc\extensibility\app-parts\sample\src\AppPartSample\bin\Debug\netcoreapp1.0\AppPartSample.dll
- C:\Users\steve_000\.nuget\packages\Microsoft.AspNetCore.Antiforgery\1.0.1\lib/netstandard1.3/Microsoft.AspNetCore.Antiforgery.dll
- C:\Users\steve_000\.nuget\packages\Microsoft.AspNetCore.Authorization\1.0.0

Tag Helpers:

- AnchorTagHelper
- CacheTagHelper
- DistributedCacheTagHelper
- EnvironmentTagHelper
- FormTagHelper

View Components:

- AnchorTagHelper
- CacheTagHelper
- DistributedCacheTagHelper
- EnvironmentTagHelper
- FormTagHelper

Home

ASP.NET Core 中的自定义模型绑定

2018/11/21 • • Edit Online

作者：Steve Smith

通过模型绑定，控制器操作可直接使用模型类型（作为方法参数传入）而不是 HTTP 请求。由模型绑定器处理传入的请求数据和应用程序模型之间的映射。开发人员可以通过实现自定义模型绑定器来扩展内置的模型绑定功能（尽管通常不需要编写自己的提供程序）。

查看或下载 GitHub 中的示例

默认模型绑定器限制

默认模型绑定器支持大多数常见的 .NET Core 数据类型，能够满足大部分开发人员的需求。他们希望将基于文本的输入从请求直接绑定到模型类型。绑定输入之前，可能需要对其进行转换。例如，当拥有某个可以用来查找模型数据的键时。基于该键，用户可以使用自定义模型绑定器来获取数据。

模型绑定查看

模型绑定为其操作对象的类型使用特定定义。简单类型转换自输入中的单个字符串。复杂类型转换自多个输入值。框架基于是否存在 `TypeConverter` 来确定差异。如果简单 `string` -> `SomeType` 映射不需要外部资源，建议创建类型转换器。

创建自己的自定义模型绑定器之前，有必要查看现有模型绑定器的实现方式。考虑使用 ByteArrayModelBinder，它可将 base64 编码的字符串转换为字节数组。字节数组通常存储为文件或数据库 BLOB 字段。

使用 ByteArrayModelBinder

Base64 编码的字符串可用来表示二进制数据。例如，下图可编码为一个字符串。

如下显示了编码字符串的一小部分：

Base64 Encoded Dotnet Bot.txt - Notepad

File Edit Format View Help


```
iVBORw0KGgoAAAANSUhEUgAAAQUAAAEFCAYAAADqlvKRAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQU
AAAAJcEhZcwAADsMAAA7DAcdvqGQAAF4vSURBVHhe7d0L0G5ldSf4zLXnVjM105WSsiiLoqAwioUUlmU5Z2
acDCbTls1IVzLV0t3VlZZKTabsYaqTRukkTroNalqaCJmIoLYhyIgJSKLQKkcNGhAvNCThEjGo5BQXD9KcN
B4oQcyZ/dvu9bnO/p537/1+7+W7Patq1fnOd9nv3s+z1v/5r8vz7B
+rUqVKlSpVqlSpVqVKlSpVKlSpVKlSp
```

只要绑定器可以将请求数据绑定到相应命名的属性或参数，模型绑定就会成功。以下示例演示如何将 `ByteArrayModelBinder` 与 视图模型结合使用：

```
[HttpPost("Profile")]
public void SaveProfile(ProfileViewModel model)
{
    string filePath = Path.Combine(_env.ContentRootPath, "wwwroot/images/upload", model.FileName);
    if (System.IO.File.Exists(model.FileName)) return;
    System.IO.File.WriteAllBytes(filePath, model.File);
}

public class ProfileViewModel
{
    public byte[] File { get; set; }
    public string FileName { get; set; }
}
```

自定义模型绑定器示例

在本部分中，我们将实现具有以下功能的自定义模型绑定器：

- 将传入的请求数据转换为强类型键参数。
- 使用 Entity Framework Core 来提取关联的实体。
- 将关联的实体作为自变量传递给操作方法。

以下示例在 `Author` 模型上使用 `ModelBinder` 属性：

```
using CustomModelBindingSample.Binders;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace CustomModelBindingSample.Data
{
    [ModelBinder(BinderType = typeof(AuthorEntityBinder))]
    public class Author
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string GitHub { get; set; }
        public string Twitter { get; set; }
        public string BlogUrl { get; set; }
    }
}
```

在前面的代码中，`ModelBinder` 属性指定应当用于绑定 `Author` 操作参数的 `IModelBinder` 的类型。

以下 `AuthorEntityBinder` 类通过 Entity Framework Core 和 `authorId` 提取数据源中的实体来绑定 `Author` 参数：

```
public class AuthorEntityBinder : IModelBinder
{
    private readonly AppDbContext _db;
    public AuthorEntityBinder(AppDbContext db)
    {
        _db = db;
    }

    public Task BindModelAsync(ModelBindingContext bindingContext)
    {
        if (bindingContext == null)
        {
            throw new ArgumentNullException(nameof(bindingContext));
        }

        var modelName = bindingContext.ModelName;

        // Try to fetch the value of the argument by name
        var valueProviderResult =
            bindingContext.ValueProvider.GetValue(modelName);

        if (valueProviderResult == ValueProviderResult.None)
        {
            return Task.CompletedTask;
        }

        bindingContext.ModelState.SetModelValue(modelName,
            valueProviderResult);

        var value = valueProviderResult.FirstValue;

        // Check if the argument value is null or empty
        if (string.IsNullOrEmpty(value))
        {
            return Task.CompletedTask;
        }

        int id = 0;
        if (!int.TryParse(value, out id))
        {
            // Non-integer arguments result in model state errors
            bindingContext.ModelState.TryAddModelError(
                                    modelName,
                                    "Author Id must be an integer.");
            return Task.CompletedTask;
        }

        // Model will be null if not found, including for
        // out of range id values (0, -3, etc.)
        var model = _db.Authors.Find(id);
        bindingContext.Result = ModelBindingResult.Success(model);
        return Task.CompletedTask;
    }
}
```

NOTE

前面的 `AuthorEntityBinder` 类旨在说明自定义模型绑定器。该类不是意在说明查找方案的最佳做法。对于查找，请绑定 `authorId` 并在操作方法中查询数据库。此方法将模型绑定故障与 `NotFound` 案例分开。

以下代码显示如何在操作方法中使用 `AuthorEntityBinder`：

```
[HttpGet("get/{authorId}")]
public IActionResult Get(Author author)
{
    return Ok(author);
}
```

可使用 `ModelBinder` 属性将 `AuthorEntityBinder` 应用于不使用默认约定的参数：

```
[HttpGet("{id}")]
public IActionResult GetById([ModelBinder(Name = "id")]Author author)
{
    if (author == null)
    {
        return NotFound();
    }
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    return Ok(author);
}
```

在此示例中，由于参数的名称不是默认的 `authorId`，因此，使用 `ModelBinder` 属性在参数上指定该名称。请注意，比起在操作方法中查找实体，控制器和操作方法都得到了简化。使用 Entity Framework Core 获取创建者的逻辑会移动到模型绑定器。如果有多种方法绑定到 `Author` 模型，就能得到很大程度的简化，并且有助于遵循 DRY 原则。

可以将 `ModelBinder` 属性应用到各个模型属性（例如视图模型上）或操作方法参数，以便为该类型或操作指定某一模型绑定器或模型名称。

实现 ModelBinderProvider

可以实现 `IModelBinderProvider`，而不是应用属性。这就是内置框架绑定器的实现方式。指定绑定器所操作的类型时，指定它生成的参数的类型，而不是绑定器接受的输入。以下绑定器提供程序适用于 `AuthorEntityBinder`。将其添加到 MVC 提供程序的集合中时，无需在 `Author` 或 `Author` 类型参数上使用 `ModelBinder` 属性。

```
using CustomModelBindingSample.Data;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ModelBinding.Binders;
using System;

namespace CustomModelBindingSample.Binders
{
    public class AuthorEntityBinderProvider : IModelBinderProvider
    {
        public IModelBinder GetBinder(ModelBinderProviderContext context)
        {
            if (context == null)
            {
                throw new ArgumentNullException(nameof(context));
            }

            if (context.Metadata.ModelType == typeof(Author))
            {
                return new BinderTypeModelBinder(typeof(AuthorEntityBinder));
            }

            return null;
        }
    }
}
```

> 注意：上述代码返回 `BinderTypeModelBinder`。`BinderTypeModelBinder` 充当模型绑定器中心，并提供依赖关系注入 (DI)。`AuthorEntityBinder` 需要 DI 来访问 EF Core。如果模型绑定器需要 DI 中的服务，请使用 `BinderTypeModelBinder`。

若要使用自定义模型绑定器提供程序，请将其添加到 `ConfigureServices` 中：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options => options.UseInMemoryDatabase());

    services.AddMvc(options =>
    {
        // add custom binder to beginning of collection
        options.ModelBinderProviders.Insert(0, new AuthorEntityBinderProvider());
    });
}
```

评估模型绑定器时，按顺序检查提供程序的集合。使用返回绑定器的第一个提供程序。

下图显示调试程序中的默认模型绑定器。

向集合的末尾添加提供程序，可能会导致在调用自定义绑定器之前调用内置模型绑定器。在此示例中，向集合的开头添加自定义提供程序，确保它用于 `Author` 操作参数。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options => options.UseInMemoryDatabase());

    services.AddMvc(options =>
    {
        // add custom binder to beginning of collection
        options.ModelBinderProviders.Insert(0, new AuthorEntityBinderProvider());
    });
}
```

建议和最佳做法

自定义模型绑定：

- 不应尝试设置状态代码或返回结果（例如 404 Not Found）。如果模型绑定失败，那么该操作方法本身的操作筛选器或逻辑会处理失败。
- 对于消除操作方法中的重复代码和跨领域问题最为有用。
- 通常不应用其将字符串转换为自定义类型，而应选择用 `TypeConverter` 来完成此操作。

ASP.NET Core MVC 的兼容性版本

2019/1/10 • • Edit Online

作者：Rick Anderson

SetCompatibilityVersion 方法允许应用选择加入或退出 ASP.NET Core MVC 2.1 或更高版本中引入的潜在中断行为变更。这些潜在的中断行为变更通常取决于 MVC 子系统的行为方式以及运行时调用“代码”的方式。通过选择加入，你将获取最新的行为以及 ASP.NET Core 的长期行为。

以下代码将兼容模式设置为 ASP.NET Core 2.2：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

建议使用最新版本（`CompatibilityVersion.Version_2_2`）来测试应用。我们预计大多数应用不会使用最新版本进行中断行为变更。

调用 `SetCompatibilityVersion(CompatibilityVersion.Version_2_0)` 的应用会被阻止进行 ASP.NET Core 2.1 MVC 和更高的 2.x 版本中引入的潜在中断行为变更。该阻止操作：

- 不适用于所有 2.1 和更高版本的更改，它的目标是潜在地中断 MVC 子系统中的 ASP.NET Core 运行时行为变更。
- 不会扩展到下一个主要版本。

未调用 `SetCompatibilityVersion` 的 ASP.NET Core 2.1 和更高 2.x 版本的应用的默认兼容性是 2.0 兼容性。即，未调用 `SetCompatibilityVersion` 与调用 `SetCompatibilityVersion(CompatibilityVersion.Version_2_0)` 相同。

以下代码将兼容模式设置为 ASP.NET Core 2.2（以下行为除外）：

- AllowCombiningAuthorizeFilters
- InputFormatterExceptionPolicy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        // Include the 2.2 behaviors
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
        // Except for the following.
        .AddMvcOptions(options =>
        {
            // Don't combine authorize filters (keep 2.0 behavior).
            options.AllowCombiningAuthorizeFilters = false;
            // All exceptions thrown by an IInputFormatter are treated
            // as model state errors (keep 2.0 behavior).
            options.InputFormatterExceptionPolicy =
                InputFormatterExceptionPolicy.AllExceptions;
        });
}
```

对于遇到中断行为变更的应用，请使用适当的兼容性开关：

- 允许使用最新版本并选择退出特定的中断行为变更。
- 请用些时间更新应用，以便其适用于最新更改。

MvcOptions 类源注释充分地阐述了更改的内容以及为什么更改对大多数用户来说是一种改进。

将来会推出 ASP.NET Core 3.0 版本。在 3.0 版本中，将删除兼容性开关支持的旧行为。我们认为这些积极的变化几乎使所有用户受益。现在通过引入这些更改，大多数应用可以立即受益，其他人员将有时间更新其应用。

使用 ASP.NET Core 构建 Web API

2019/1/14 • • Edit Online

作者：Scott Addie

查看或下载示例代码（如何下载）

本文档说明如何在 ASP.NET Core 中构建 Web API 以及每项功能的最佳适用场景。

从 ControllerBase 派生类

从控制器（旨在用作 Web API）中的 ControllerBase 类继承。例如：

```
[Produces("application/json")]
[Route("api/[controller]")]
public class PetsController : ControllerBase
{
    private readonly PetsRepository _repository;

    public PetsController(PetsRepository repository)
    {
        _repository = repository;
    }

    [HttpGet]
    public async Task<ActionResult<List<Pet>>> GetAllAsync()
    {
        return await _repository.GetPetsAsync();
    }

    [HttpGet("{id}")]
    [ProducesResponseType(404)]
    public async Task<ActionResult<Pet>> GetByIdAsync(int id)
    {
        var pet = await _repository.GetPetAsync(id);

        if (pet == null)
        {
            return NotFound();
        }

        return pet;
    }

    [HttpPost]
    [ProducesResponseType(400)]
    public async Task<ActionResult<Pet>> CreateAsync(Pet pet)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        await _repository.AddPetAsync(pet);

        return CreatedAtAction(nameof(GetByIdAsync),
            new { id = pet.Id }, pet);
    }
}
```

```
[Produces("application/json")]
[Route("api/[controller]")]
public class PetsController : ControllerBase
{
    private readonly PetsRepository _repository;

    public PetsController(PetsRepository repository)
    {
        _repository = repository;
    }

    [HttpGet]
    [ProducesResponseType(typeof(IEnumerable<Pet>), 200)]
    public async Task<IActionResult> GetAllAsync()
    {
        var pets = await _repository.GetPetsAsync();

        return Ok(pets);
    }

    [HttpGet("{id}")]
    [ProducesResponseType(typeof(Pet), 200)]
    [ProducesResponseType(404)]
    public async Task<IActionResult> GetByIdAsync(int id)
    {
        var pet = await _repository.GetPetAsync(id);

        if (pet == null)
        {
            return NotFound();
        }

        return Ok(pet);
    }

    [HttpPost]
    [ProducesResponseType(typeof(Pet), 201)]
    [ProducesResponseType(400)]
    public async Task<IActionResult> CreateAsync([FromBody] Pet pet)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        await _repository.AddPetAsync(pet);

        return CreatedAtAction(nameof(GetByIdAsync),
            new { id = pet.Id }, pet);
    }
}
```

通过 `ControllerBase` 类可使用一些属性和方法。在前面的代码中，示例包括 BadRequest(ModelStateDictionary) 和 CreatedAtAction(String, Object, Object)。将在操作方法中调用这些方法以分别返回 HTTP 400 和 201 状态代码。将使用 ModelState属性（还可由 `ControllerBase` 提供）处理请求模型验证。

使用 ApiController 属性进行注释

ASP.NET Core 2.1 引入了 [ApiController] 特性，用于批注 Web API 控制器类。例如:

```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
```

为在控制器级别使用此特性，需要兼容性版本 2.1 或更高版本（通过 SetCompatibilityVersion 设置）。例如，`Startup.ConfigureServices` 中突出显示的代码设置了 2.1 兼容性标志：

```
services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

有关更多信息，请参见ASP.NET Core MVC 的兼容性版本。

在 ASP.NET Core 2.2 或更高版本中，可将 `[ApiController]` 特性应用于程序集。以这种方式进行注释，会将 web API 行为应用到程序集中的所有控制器。请注意，无法针对单个控制器执行选择退出操作。建议将程序集级别的特性应用于 `Startup` 类：

```
[assembly: ApiController]
namespace WebApiSample.Api._22
{
    public class Startup
    {
```

为在程序集级别使用此特性，需要兼容性版本 2.2 或更高版本（通过 SetCompatibilityVersion 设置）。

`[ApiController]` 特性通常结合 `ControllerBase` 来为控制器启用特定于 REST 行为。通过 `ControllerBase` 可使用NotFound 和 File 等方法。

另一种方法是创建使用 `[ApiController]` 特性进行批注的自定义基本控制器类：

```
[ApiController]
public class MyBaseController
{
}
```

以下各部分说明该特性添加的便利功能。

自动 HTTP 400 响应

模型验证错误会自动触发 HTTP 400 响应。因此，操作中不需要以下代码：

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

使用 InvalidModelStateResponseFactory 自定义所生成的响应的输出。

如果操作可从模型验证错误中恢复，禁用默认行为是有用的。当 SuppressModelStateInvalidFilter 属性设置为 `true` 时，会禁用默认行为。在 `Startup.ConfigureServices` 中的 `services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_<version_number>);` 后添加下列代码：

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;

        options.ClientErrorMapping[404].Link =
            "https://httpstatuses.com/404";
    });
```

```
services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressConsumesConstraintForFormFileParameters = true;
    options.SuppressInferBindingSourcesForParameters = true;
    options.SuppressModelStateInvalidFilter = true;
});
```

使用 2.2 或更高版本的兼容性标志时，HTTP 400 响应的默认响应类型为 ValidationProblemDetails。`ValidationProblemDetails` 类型符合 RFC 7807 规范。将 `SuppressUseValidationProblemDetailsForInvalidModelStateResponses` 属性设置为 `true` 以返回 ASP.NET Core 2.1 错误格式 SerializableError。在 `Startup.ConfigureServices` 中添加下列代码：

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options
          .SuppressUseValidationProblemDetailsForInvalidModelStateResponses = true;
    });
```

绑定源参数推理

绑定源特性定义可找到操作参数值的位置。存在以下绑定源特性：

特性	绑定源
[FromBody]	请求正文
[FromForm]	请求正文中的表单数据
[FromHeader]	请求标头
[FromQuery]	请求查询字符串参数
[FromRoute]	当前请求中的路由数据
[FromServices]	作为操作参数插入的请求服务

WARNING

当值可能包含 `%2f`（即 `/`）时，请勿使用 `[FromRoute]`。`%2f` 不会转换为 `/`（非转移形式）。如果值可能包含 `%2f`，则使用 `[FromQuery]`。

没有 `[ApiController]` 特性时，将显式定义绑定源特性。在下面的示例中，`[FromQuery]` 特性指示 `discontinuedOnly` 参数值在请求 URL 的查询字符串中提供：

```
[HttpGet]
public async Task<ActionResult<List<Product>>> GetAsync(
    [FromQuery] bool discontinuedOnly = false)
{
    List<Product> products = null;

    if (discontinuedOnly)
    {
        products = await _repository.GetDiscontinuedProductsAsync();
    }
    else
    {
        products = await _repository.GetProductsAsync();
    }

    return products;
}
```

推理规则应用于操作参数的默认数据源。这些规则将配置绑定资源，否则你可以手动应用操作参数。绑定源特性的行为如下：

- **[FromBody]**，针对复杂类型参数进行推断。此规则不适用于具有特殊含义的任何复杂的内置类型，如 IFormCollection 和 CancellationToken。绑定源推理代码将忽略这些特殊类型。对于简单类型（例如 `string` 或 `int`），推断不出 `[FromBody]`。因此，如果需要该功能，对于简单类型，应使用 `[FromBody]` 属性。当操作中的多个参数为显式指定（通过 `[FromBody]`）或在请求正文中作为绑定进行推断时，将会引发异常。例如，下面的操作签名会导致异常：

  ```
  // Don't do this. All of the following actions result in an exception.
  [HttpPost]
  public IActionResult Action1(Product product,
                               Order order) => null;

  [HttpPost]
  public IActionResult Action2(Product product,
                               [FromBody] Order order) => null;

  [HttpPost]
  public IActionResult Action3([FromBody] Product product,
                               [FromBody] Order order) => null;
  ```

  > **NOTE**
  >
  > 在 ASP.NET Core 2.1 中，集合类型参数（如列表和数组）被不正确地推断为 [FromQuery]。若要从请求正文中绑定参数，应对这些参数使用 [FromBody]。此行为在 ASP.NET Core 2.2 或更高版本中得到了修复，其中集合类型参数默认被推断为从正文中绑定。

- [FromForm] 是针对 IFormFile 和 IFormFileCollection 类型的操作参数推断出来的。该特性不针对任何简单类型或用户定义类型进行推断。

- **[FromRoute]**，针对与路由模板中的参数相匹配的任何操作参数名称进行推断。当多个路由与一个操作参数匹配时，任何路由值都视为 `[FromRoute]`。

- **[FromQuery]**，针对任何其他操作参数进行推断。

当 SuppressInferBindingSourcesForParameters 属性设置为 `true` 时，会禁用默认推理规则。在

`Startup.ConfigureServices` 中的 `services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_<version_number>);` 后添加下列代码：

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;

        options.ClientErrorMapping[404].Link =
            "https://httpstatuses.com/404";
    });
```

```
services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressConsumesConstraintForFormFileParameters = true;
    options.SuppressInferBindingSourcesForParameters = true;
    options.SuppressModelStateInvalidFilter = true;
});
```

Multipart/form-data 请求推理

使用 [FromForm] 特性批注操作参数时，将推断 `multipart/form-data` 请求内容类型。

当 SuppressConsumesConstraintForFormFileParameters 属性设置为 `true` 时，会禁用默认行为。

在 `Startup.ConfigureServices` 中添加下列代码：

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;

        options.ClientErrorMapping[404].Link =
            "https://httpstatuses.com/404";
    });
```

在 `Startup.ConfigureServices` 中的 `services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);` 后添加下列代码：

```
services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressConsumesConstraintForFormFileParameters = true;
    options.SuppressInferBindingSourcesForParameters = true;
    options.SuppressModelStateInvalidFilter = true;
});
```

特性路由要求

特性路由是必要条件。例如:

```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
```

不能通过 UseMvc 中定义的传统路由或通过 `Startup.Configure` 中的 UseMvcWithDefaultRoute 访问操作。

错误状态代码的问题详细信息响应

在 ASP.NET Core 2.2 或更高版本中，MVC 会将错误结果（状态代码为 400 或更高的结果）转换为状态代码为 ProblemDetails 的结果。`ProblemDetails` 为：

- 基于 RFC 7807 规范的类型。
- 用于指定 HTTP 响应中计算机可读的错误详细信息的标准化格式。

考虑在控制器操作中使用以下代码：

```
if (product == null)
{
    return NotFound();
}
```

`NotFound` 的 HTTP 响应具有 404 状态代码和 `ProblemDetails` 正文。例如：

```
{
    type: "https://tools.ietf.org/html/rfc7231#section-6.5.4",
    title: "Not Found",
    status: 404,
    traceId: "0HLHLV31KRN83:00000001"
}
```

需具备 2.2 版本或更高版本的兼容性标志，才可使用问题详细信息功能。当 `SuppressMapClientErrors` 属性设置为 `true` 时，会禁用默认行为。在 `Startup.ConfigureServices` 中添加下列代码：

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;

        options.ClientErrorMapping[404].Link =
            "https://httpstatuses.com/404";
    });
```

使用 `ClientErrorMapping` 属性配置 `ProblemDetails` 响应的内容。例如，以下代码会更新 404 响应的 `type` 属性：

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;

        options.ClientErrorMapping[404].Link =
            "https://httpstatuses.com/404";
    });
```

其他资源

- ASP.NET Core Web API 中控制器操作的返回类型
- ASP.NET Core Web API 中的自定义格式化程序
- 设置 ASP.NET Core Web API 中响应数据的格式
- 带有 Swagger/OpenAPI 的 ASP.NET Core Web API 帮助页
- 在 ASP.NET Core 中路由到控制器操作

教程：使用 ASP.NET Core MVC 创建 Web API

2019/1/10 • • Edit Online

作者：Rick Anderson 和 Mike Wasson

本教程介绍使用 ASP.NET Core 构建 Web API 的基础知识。

在本教程中，你将了解：

- 创建 Web API 项目。
- 添加模型类。
- 创建数据库上下文。
- 注册数据库上下文。
- 添加控制器。
- 添加 CRUD 方法。
- 配置路由和 URL 路径。
- 指定返回值。
- 使用 Postman 调用 Web API。
- 使用 jQuery 调用 Web API。

在结束时，你会获得可以管理存储在关系数据库中的“待办事项”的 Web API。

概述

本教程将创建以下 API：

API	说明	请求正文	响应正文
GET /api/todo	获取所有待办事项	无	待办事项的数组
GET /api/todo/{id}	按 ID 获取项	无	待办事项
POST /api/todo	添加新项	待办事项	待办事项
PUT /api/todo/{id}	更新现有项	待办事项	无
DELETE /api/todo/{id}	删除项	无	无

下图显示了应用的设计。

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

创建 Web 项目

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 从“文件”菜单中选择“新建” > “项目”。
- 选择“ASP.NET Core Web 应用程序”模板。将项目命名为 TodoApi，然后单击“确定”。
- 在“新建 ASP.NET Core Web 应用程序 - TodoApi”对话框中，选择 ASP.NET Core 版本。选择“API”模板，然后单击“确定”。请不要选择“启用 Docker 支持”。

测试 API

项目模板会创建 `values` API。从浏览器调用 `Get` 方法以测试应用。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

按 Ctrl+F5 运行应用。Visual Studio 启动浏览器并导航到 `https://localhost:<port>/api/values`，其中 `<port>` 是随机选择的端口号。

如果出现询问是否应信任 IIS Express 证书的对话框，则选择"是"。在接下来出现的"安全警告"对话框中，选择"是"。

会返回以下 JSON：

```
["value1","value2"]
```

添加模型类

模型是一组表示应用管理的数据的类。此应用的模型是单个 `TodoItem` 类。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 在"解决方案资源管理器"中，右键单击项目。选择"添加" > "新建文件夹"。将文件夹命名为"Models"。
- 右键单击"Models"文件夹，然后选择"添加" > "类"。将类命名为 TodoItem，然后选择"添加"。
- 将模板代码替换为以下代码：

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

`Id` 属性用作关系数据库中的唯一键。

模型类可位于项目的任意位置，但按照惯例会使用 Models 文件夹。

添加数据库上下文

数据库上下文是为数据模型协调 Entity Framework 功能的主类。此类由 `Microsoft.EntityFrameworkCore.DbContext` 类派生而来。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 右键单击“Models”文件夹，然后选择“添加” > “类”。将类命名为 TodoContext，然后单击“添加”。
- 将模板代码替换为以下代码：

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

注册数据库上下文

在 ASP.NET Core 中，服务（如数据库上下文）必须向依赖关系注入 (DI) 容器进行注册。该容器向控制器提供服务。

使用以下突出显示的代码更新 Startup.cs：

```
// Unused usings removed
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the
        //container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt =>
                opt.UseInMemoryDatabase("TodoList"));
            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
        }

        // This method gets called by the runtime. Use this method to configure the HTTP
        //request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                // The default HSTS value is 30 days. You may want to change this for
                // production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseMvc();
        }
    }
}
```

前面的代码:

- 删除未使用的 `using` 声明。
- 将数据库上下文添加到 DI 容器。
- 指定数据库上下文将使用内存中数据库。

添加控制器

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 右键单击 Controllers 文件夹。

- 选择 **添加** > **新建项**。
- 在“添加新项”对话框中，选择“API 控制器类”模板。
- 将类命名为 TodoController，然后选择“添加”。

- 将模板代码替换为以下代码：

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using TodoApi.Models;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoController : ControllerBase
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                // Create a new TodoItem if collection is empty,
                // which means you can't delete all TodoItems.
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

前面的代码：

- 定义了没有方法的 API 控制器类。
- 使用 `[ApiController]` 属性修饰类。此属性指示控制器响应 Web API 请求。有关该属性启用的特定行为的信息，请参阅使用 ApiController 属性进行注释。

- 使用 DI 将数据库上下文（`TodoContext`）注入到控制器中。数据库上下文将在控制器中的每个 CRUD 方法中使用。
- 如果数据库为空，则将名为 `Item1` 的项添加到数据库。此代码位于构造函数中，因此在每次出现新 HTTP 请求时运行。如果删除所有项，则构造函数会在下次调用 API 方法时再次创建 `Item1`。因此删除可能看上去不起作用，不过实际上确实有效。

添加 Get 方法

若要提供检索待办事项的 API，请将以下方法添加到 `TodoController` 类中：

```
// GET: api/Todo
[HttpGet]
public async Task<ActionResult<IEnumerable<TodoItem>>> GetTodoItems()
{
    return await _context.TodoItems.ToListAsync();
}

// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

这些方法实现两个 GET 终结点：

- `GET /api/todo`
- `GET /api/todo/{id}`

通过从浏览器调用两个终结点来测试应用。例如：

- `https://localhost:<port>/api/todo`
- `https://localhost:<port>/api/todo/1`

以下 HTTP 响应通过调用 `GetTodoItems` 来生成：

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

路由和 URL 路径

`[HttpGet]` 属性表示响应 HTTP GET 请求的方法。每个方法的 URL 路径构造如下所示：

- 在控制器的 `Route` 属性中以模板字符串开头：

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoController : ControllerBase
    {
        private readonly TodoContext _context;
```

- 将 `[controller]` 替换为控制器的名称，按照惯例，在控制器类名称中去掉“Controller”后缀。对于此示例，控制器类名称为“Todo”控制器，因此控制器名称为“todo”。ASP.NET Core 路由不区分大小写。
- 如果 `[HttpGet]` 属性具有路由模板（例如 `[HttpGet("/products")]`），则将它追加到路径。此示例不使用模板。有关详细信息，请参阅使用 Http [Verb] 特性的特性路由。

在下面的 `GetTodoItem` 方法中，`"{id}"` 是待办事项的唯一标识符的占位符变量。调用 `GetTodoItem` 时，URL 中 `"{id}"` 的值会在 `id` 参数中提供给方法。

```
// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

`Name = "GetTodo"` 参数会创建命名路由。你会在后面看到应用如何使用名称创建使用路由名称的 HTTP 链接。

返回值

`GetTodoItems` 和 `GetTodoItem` 方法的返回类型是 ActionResult<T> 类型。ASP.NET Core 自动将对象序列化为 JSON，并将 JSON 写入响应消息的正文中。在假设没有未经处理的异常的情况下，此返回类型的响应代码为 200。未经处理的异常将转换为 5xx 错误。

`ActionResult` 返回类型可以表示大范围的 HTTP 状态代码。例如，`GetTodoItem` 可以返回两个不同的状态值：

- 如果没有任何项与请求的 ID 匹配，则该方法将返回 404 NotFound 错误代码。
- 否则，此方法将返回具有 JSON 响应正文的 200。返回 `item` 则产生 HTTP 200 响应。

测试 GetTodoItems 方法

本教程使用 Postman 测试 Web API。

- 安装 Postman
- 启动 Web 应用。
- 启动 Postman。
- 禁用 **SSL** 证书验证
  - 在“文件”>“设置”（“常规”*选项卡）中，禁用“SSL 证书验证”。

> **WARNING**
>
> 在测试控制器之后重新启用 SSL 证书验证。

- 创建新请求。
  - 将 HTTP 方法设置为“GET”。
  - 将请求 URL 设置为 `https://localhost:<port>/api/todo`。例如 `https://localhost:5001/api/todo`。
- 在 Postman 中设置“两窗格视图”。
- 选择“发送”。

添加创建方法

添加以下 `PostTodoItem` 方法：

```
// POST: api/Todo
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem);
}
```

正如 [HttpPost] 属性所指示，前面的代码是 HTTP POST 方法。该方法从 HTTP 请求正文获取待办事项的值。

`CreatedAtRoute` 方法：

- 返回 201 响应。HTTP 201 是在服务器上创建新资源的 HTTP POST 方法的标准响应。
- 向响应添加位置标头。位置标头指定新建的待办事项的 URI。有关详细信息，请参阅创建的 10.2.2 201。
- 使用名为 route 的“GetTodo”来创建 URL。已在 `GetTodoItem` 中定义名为 route 的“GetTodo”：

```
// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

测试 PostTodoItem 方法

- 生成项目。
- 在 Postman 中，将 HTTP 方法设置为 `POST`。
- 选择“正文”选项卡。
- 选择“原始”单选按钮。
- 将类型设置为 JSON (application/json)
- 在请求正文中，输入待办事项的 JSON：

```
{
  "name":"walk dog",
  "isComplete":true
}
```

- 选择“发送”。

如果收到 405 不允许的方法错误，则可能是由于未在添加 `PostTodoItem` 方法之后编译项目。

测试位置标头 URI

- 在“响应”窗格中选择“标头”选项卡。
- 复制“位置”标头值：

- 将方法设置为“GET”。

- 粘贴 URI（例如，`https://localhost:5001/api/Todo/2`）
- 选择“发送”。

添加 PutTodoItem 方法

添加以下 `PutTodoItem` 方法：

```
// PUT: api/Todo/5
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;
    await _context.SaveChangesAsync();

    return NoContent();
}
```

`PutTodoItem` 与 `PostTodoItem` 类似，但是使用的是 HTTP PUT。响应是 204（无内容）。根据 HTTP 规范，PUT 请求需要客户端发送整个更新的实体，而不仅仅是更改。若要支持部分更新，请使用 HTTP PATCH。

测试 PutTodoItem 方法

更新 id = 1 的待办事项并将其名称设置为“feed fish”：

```
  {
    "ID":1,
    "name":"feed fish",
    "isComplete":true
  }
```

下图显示 Postman 更新：

添加 DeleteTodoItem 方法

添加以下 `DeleteTodoItem` 方法：

```
// DELETE: api/Todo/5
[HttpDelete("{id}")]
public async Task<ActionResult<TodoItem>> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return todoItem;
}
```

`DeleteTodoItem` 响应是 204（无内容）。

测试 DeleteTodoItem 方法

使用 Postman 删除待办事项：

- 将方法设置为 `DELETE`。
- 设置要删除的对象的 URI，例如 `https://localhost:5001/api/todo/1`
- 选择"Send"

示例应用允许删除所有项，但是在删除最后一项后，模型类构造函数会在下次调用 API 时创建一个新项。

使用 jQuery 调用 API

在本部分中，添加了使用 jQuery 调用 Web API 的 HTML 页面。jQuery 启动请求，并用 API 响应中的详细信息更新页面。

配置应用提供静态文件并启用默认文件映射：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // The default HSTS value is 30 days. You may want to change this for
        // production scenarios, see https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseHttpsRedirection();
    app.UseMvc();
}
```

在项目目录中创建 wwwroot 文件夹。

将一个名为 index.html 的 HTML 文件添加到 wwwroot 目录。用以下标记替代其内容：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <style>
        input[type='submit'], button, [aria-label] {
            cursor: pointer;
        }

        #spoiler {
            display: none;
        }

        table {
            font-family: Arial, sans-serif;
            border: 1px solid;
            border-collapse: collapse;
        }

        th {
            background-color: #0066CC;
            color: white;
        }

        td {
            border: 1px solid;
            padding: 5px;
        }
    </style>
</head>
<body>
    <h1>To-do CRUD</h1>
    <h3>Add</h3>
    <form action="javascript:void(0);" method="POST" onsubmit="addItem()">
        <input type="text" id="add-name" placeholder="New to-do">
        <input type="submit" value="Add">
    </form>
```

```
    </form>

    <div id="spoiler">
        <h3>Edit</h3>
        <form class="my-form">
            <input type="hidden" id="edit-id">
            <input type="checkbox" id="edit-isComplete">
            <input type="text" id="edit-name">
            <input type="submit" value="Save">
            <a onclick="closeInput()" aria-label="Close">&#10006;</a>
        </form>
    </div>

    <p id="counter"></p>

    <table>
        <tr>
            <th>Is Complete</th>
            <th>Name</th>
            <th></th>
            <th></th>
        </tr>
        <tbody id="todos"></tbody>
    </table>

    <script src="https://code.jquery.com/jquery-3.3.1.min.js"
            integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
            crossorigin="anonymous"></script>
    <script src="site.js"></script>
</body>
</html>
```

将名为 site.js 的 JavaScript 文件添加到 wwwroot 目录。用以下代码替代其内容：

```
const uri = "api/todo";
let todos = null;
function getCount(data) {
  const el = $("#counter");
  let name = "to-do";
  if (data) {
    if (data > 1) {
      name = "to-dos";
    }
    el.text(data + " " + name);
  } else {
    el.text("No " + name);
  }
}

$(document).ready(function() {
  getData();
});

function getData() {
  $.ajax({
    type: "GET",
    url: uri,
    cache: false,
    success: function(data) {
      const tBody = $("#todos");

      $(tBody).empty();

      getCount(data.length);

      $.each(data, function(key, item) {
        const tr = $("<tr></tr>")
          append(
```

```
          .append(
            $("<td></td>").append(
              $("<input/>", {
                type: "checkbox",
                disabled: true,
                checked: item.isComplete
              })
            )
          )
          .append($("<td></td>").text(item.name))
          .append(
            $("<td></td>").append(
              $("<button>Edit</button>").on("click", function() {
                editItem(item.id);
              })
            )
          )
          .append(
            $("<td></td>").append(
              $("<button>Delete</button>").on("click", function() {
                deleteItem(item.id);
              })
            )
          );

        tr.appendTo(tBody);
      });

      todos = data;
    }
  });
}

function addItem() {
  const item = {
    name: $("#add-name").val(),
    isComplete: false
  };

  $.ajax({
    type: "POST",
    accepts: "application/json",
    url: uri,
    contentType: "application/json",
    data: JSON.stringify(item),
    error: function(jqXHR, textStatus, errorThrown) {
      alert("Something went wrong!");
    },
    success: function(result) {
      getData();
      $("#add-name").val("");
    }
  });
}

function deleteItem(id) {
  $.ajax({
    url: uri + "/" + id,
    type: "DELETE",
    success: function(result) {
      getData();
    }
  });
}

function editItem(id) {
  $.each(todos, function(key, item) {
    if (item.id === id) {
      $("#edit-name").val(item.name);
      $("#edit-id").val(item.id);
```

```
      $("#edit-id").val(item.id);
      $("#edit-isComplete")[0].checked = item.isComplete;
    }
  });
  $("#spoiler").css({ display: "block" });
}

$(".my-form").on("submit", function() {
  const item = {
    name: $("#edit-name").val(),
    isComplete: $("#edit-isComplete").is(":checked"),
    id: $("#edit-id").val()
  };

  $.ajax({
    url: uri + "/" + $("#edit-id").val(),
    type: "PUT",
    accepts: "application/json",
    contentType: "application/json",
    data: JSON.stringify(item),
    success: function(result) {
      getData();
    }
  });

  closeInput();
  return false;
});

function closeInput() {
  $("#spoiler").css({ display: "none" });
}
```

可能需要更改 ASP.NET Core 项目的启动设置，以便对 HTML 页面进行本地测试：

- 打开 Properties\launchSettings.json。
- 删除 `launchUrl` 以便在项目的默认文件 index.html 强制打开应用。

有多种方式可以获取 jQuery。在前面的代码片段中，库是从 CDN 中加载的。

此示例调用 API 的所有 CRUD 方法。以下是 API 调用的说明。

获取待办事项的列表

jQuery ajax 函数将 `GET` 请求发送至 API，这将返回表示待办事项数组的 JSON。如果请求成功，则调用 `success` 回调函数。在该回调中使用待办事项信息更新 DOM。

```
$(document).ready(function() {
  getData();
});

function getData() {
  $.ajax({
    type: "GET",
    url: uri,
    cache: false,
    success: function(data) {
      const tBody = $("#todos");

      $(tBody).empty();

      getCount(data.length);

      $.each(data, function(key, item) {
        const tr = $("<tr></tr>")
          .append(
            $("<td></td>").append(
              $("<input/>", {
                type: "checkbox",
                disabled: true,
                checked: item.isComplete
              })
            )
          )
          .append($("<td></td>").text(item.name))
          .append(
            $("<td></td>").append(
              $("<button>Edit</button>").on("click", function() {
                editItem(item.id);
              })
            )
          )
          .append(
            $("<td></td>").append(
              $("<button>Delete</button>").on("click", function() {
                deleteItem(item.id);
              })
            )
          );

        tr.appendTo(tBody);
      });

      todos = data;
    }
  });
}
```

添加待办事项

Ajax 函数发送 `POST`，请求正文中包含待办事项。将 `accepts` 和 `contentType` 选项设置为 `application/json`，以便指定接收和发送的媒体类型。待办事项使用 JSON.stringify 转换为 JSON。当 API 返回成功状态的代码时，将调用 `getData` 函数来更新 HTML 表。

```
function addItem() {
  const item = {
    name: $("#add-name").val(),
    isComplete: false
  };

  $.ajax({
    type: "POST",
    accepts: "application/json",
    url: uri,
    contentType: "application/json",
    data: JSON.stringify(item),
    error: function(jqXHR, textStatus, errorThrown) {
      alert("Something went wrong!");
    },
    success: function(result) {
      getData();
      $("#add-name").val("");
    }
  });
}
```

更新待办事项

更新待办事项与添加类似。`url` 更改为添加项的唯一标识符，并且 `type` 为 `PUT`。

```
$.ajax({
  url: uri + "/" + $("#edit-id").val(),
  type: "PUT",
  accepts: "application/json",
  contentType: "application/json",
  data: JSON.stringify(item),
  success: function(result) {
    getData();
  }
});
```

删除待办事项

若要删除待办事项，请将 AJAX 调用上的 `type` 设为 `DELETE` 并指定该项在 URL 中的唯一标识符。

```
$.ajax({
  url: uri + "/" + id,
  type: "DELETE",
  success: function(result) {
    getData();
  }
});
```

其他资源

查看或下载本教程的示例代码。请参阅如何下载。

有关更多信息，请参见以下资源：

- 使用 ASP.NET Core 构建 Web API
- 带有 Swagger/OpenAPI 的 ASP.NET Core Web API 帮助页
- ASP.NET Core Razor 页面和 EF Core - 教程系列
- 在 ASP.NET Core 中路由到控制器操作
- ASP.NET Core Web API 中控制器操作的返回类型

- 将 ASP.NET Core 应用部署到 Azure 应用服务
- 托管和部署 ASP.NET Core

后续步骤

在本教程中，你将了解：

- 创建 Web API 项目。
- 添加模型类。
- 创建数据库上下文。
- 注册数据库上下文。
- 添加控制器。
- 添加 CRUD 方法。
- 配置路由和 URL 路径。
- 指定返回值。
- 使用 Postman 调用 Web API。
- 使用 jQuery 调用 Web API。

前进到下一个教程，了解如何生成 API 帮助页：

Swashbuckle 和 ASP.NET Core 入门

使用 ASP.NET Core 和 MongoDB 创建 Web API

2019/1/8 • • Edit Online

作者 Pratik Khandelwal 和 Scott Addie

本教程创建对 MongoDB NoSQL 数据库执行创建、读取、更新和删除 (CRUD) 操作的 Web API。

在本教程中，你将了解：

- 配置 MongoDB
- 创建 MongoDB 数据库
- 定义 MongoDB 集合和架构
- 从 Web API 执行 MongoDB CRUD 操作

查看或下载示例代码（如何下载）

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- .NET Core SDK 2.2 或更高版本
- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- MongoDB

配置 MongoDB

如果使用的是 Windows，MongoDB 将默认安装在 C:\Program Files\MongoDB 中。将 C:\Program Files\MongoDB\Server<version_number>\bin 添加到 `Path` 环境变量。通过此更改可以从开发计算机上的任意位置访问 MongoDB。

使用以下步骤中的 mongo Shell 可以创建数据库、创建集合和存储文档。有关 mongo Shell 命令的详细信息，请参阅使用 mongo Shell。

1. 选择开发计算机上用于存储数据的目录。例如，在 Windows 上为 *C:\BooksData*。创建目录（如果不存在）。mongo Shell 不会创建新目录。

2. 打开命令行界面。运行以下命令以连接到默认端口 27017 上的 MongoDB。请记得将 `<data_directory_path>` 替换为上一步中选择的目录。

   ```
   mongod --dbpath <data_directory_path>
   ```

3. 打开另一个命令行界面实例。通过运行以下命令来连接到默认测试数据库：

   ```
   mongo
   ```

4. 在命令行界面中运行下面的命令：

```
use BookstoreDb
```

如果该命令尚不存在，则将创建名为 BookstoreDb 的数据库。如果该数据库存在，则将为事务打开其连接。

5. 使用以下命令创建 `Books` 集合：

```
db.createCollection('Books')
```

显示以下结果：

```
{ "ok" : 1 }
```

6. 使用以下命令定义 `Books` 集合的架构并插入两个文档：

```
db.Books.insertMany([{'Name':'Design Patterns','Price':54.93,'Category':'Computers','Author':'Ralph
Johnson'}, {'Name':'Clean Code','Price':43.15,'Category':'Computers','Author':'Robert C. Martin'}])
```

显示以下结果：

```
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("5bfd996f7b8e48dc15ff215d"),
    ObjectId("5bfd996f7b8e48dc15ff215e")
  ]
}
```

7. 使用以下命令查看数据库中的文档：

```
db.Books.find({}).pretty()
```

显示以下结果：

```
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215d"),
  "Name" : "Design Patterns",
  "Price" : 54.93,
  "Category" : "Computers",
  "Author" : "Ralph Johnson"
}
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215e"),
  "Name" : "Clean Code",
  "Price" : 43.15,
  "Category" : "Computers",
  "Author" : "Robert C. Martin"
}
```

该架构将为每个文档添加类型 `ObjectId` 的自动生成的 `_id` 属性。

数据库可供使用了。你可以开始创建 ASP.NET Core Web API。

创建 ASP.NET Core Web API 项目

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

1. 转到“文件” > “新建” > “项目”。
2. 选择“ASP.NET Core Web 应用程序”，将项目命名为“BooksApi”，然后单击“确定”。
3. 选择“.NET Core”目标框架和“ASP.NET Core 2.1”。选择“API”项目模板，然后单击“确定”：
4. 在“包管理器控制台”窗口中，导航到项目根。运行以下命令以安装适用于 MongoDB 的 .NET 驱动程序：

```
Install-Package MongoDB.Driver -Version 2.7.2
```

添加模型

1. 将 Models 目录添加到项目根。
2. 使用以下代码将 `Book` 类添加到 Models 目录：

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BooksApi.Models
{
    public class Book
    {
        public ObjectId Id { get; set; }

        [BsonElement("Name")]
        public string BookName { get; set; }

        [BsonElement("Price")]
        public decimal Price { get; set; }

        [BsonElement("Category")]
        public string Category { get; set; }

        [BsonElement("Author")]
        public string Author { get; set; }
    }
}
```

在前面的类中，将公共语言运行时 (CLR) 对象映射到 MongoDB 集合时需要 `Id` 属性。类中的其他属性使用 `[BsonElement]` 属性进行修饰。该属性的值表示 MongoDB 集合中的属性名称。

添加 CRUD 操作类

1. 将 Services 目录添加到项目根。
2. 使用以下代码将 `BookService` 类添加到 Services 目录：

```
using System.Collections.Generic;
using System.Linq;
using BooksApi.Models;
using Microsoft.Extensions.Configuration;
using MongoDB.Bson;
using MongoDB.Driver;

namespace BooksApi.Services
{
    public class BookService
    {
        private readonly IMongoCollection<Book> _books;

        public BookService(IConfiguration config)
        {
            var client = new MongoClient(config.GetConnectionString("BookstoreDb"));
            var database = client.GetDatabase("BookstoreDb");
            _books = database.GetCollection<Book>("Books");
        }

        public List<Book> Get()
        {
            return _books.Find(book => true).ToList();
        }

        public Book Get(string id)
        {
            var docId = new ObjectId(id);

            return _books.Find<Book>(book => book.Id == docId).FirstOrDefault();
        }

        public Book Create(Book book)
        {
            _books.InsertOne(book);
            return book;
        }

        public void Update(string id, Book bookIn)
        {
            var docId = new ObjectId(id);

            _books.ReplaceOne(book => book.Id == docId, bookIn);
        }

        public void Remove(Book bookIn)
        {
            _books.DeleteOne(book => book.Id == bookIn.Id);
        }

        public void Remove(ObjectId id)
        {
            _books.DeleteOne(book => book.Id == id);
        }
    }
}
```

3. 将 MongoDB 连接字符串添加到 appsettings.json：

```
{
  "ConnectionStrings": {
    "BookstoreDb": "mongodb://localhost:27017"
  },
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  }
}
```

将在 `BookService` 类构造函数中访问前面的 `BookstoreDb` 属性。

4. 在 `Startup.ConfigureServices` 中，向依赖关系注入系统注册 `BookService` 类：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<BookService>();
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

必须执行前面的服务注册才能支持消费类中的构造函数注入。

`BookService` 类使用以下 `MongoDB.Driver` 成员对数据库执行 CRUD 操作：

- `MongoClient` – 读取执行数据库操作的服务器实例。此类的构造函数提供了 MongoDB 连接字符串：

```
public BookService(IConfiguration config)
{
    var client = new MongoClient(config.GetConnectionString("BookstoreDb"));
    var database = client.GetDatabase("BookstoreDb");
    _books = database.GetCollection<Book>("Books");
}
```

- `IMongoDatabase` – 表示用于执行操作的 Mongo 数据库。本教程在界面上使用泛型 `GetCollection<T>(collection)` 方法来获取对特定集合中的数据的访问。调用此方法后，可以对集合执行 CRUD 操作。在 `GetCollection<T>(collection)` 方法调用中：
  - `collection` 表示集合名称。
  - `T` 表示存储在集合中的 CLR 对象类型。

`GetCollection<T>(collection)` 返回 `MongoCollection` 对象，该对象表示集合。在本教程中，对集合调用以下方法：

- `Find<T>` – 返回集合中与提供的搜索条件匹配的所有文档。
- `InsertOne` – 插入提供的对象作为集合中的新文档。
- `ReplaceOne` – 将与提供的搜索条件匹配的单个文档替换为提供的对象。
- `DeleteOne` – 删除与提供的搜索条件匹配的单个文档。

添加控制器

1. 使用以下代码将 `BooksController` 类添加到 Controllers 目录：

```
using System.Collections.Generic;
using BooksApi.Models;
using BooksApi.Services;
using Microsoft.AspNetCore.Mvc;

namespace BooksApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BooksController : ControllerBase
    {
        private readonly BookService _bookService;

        public BooksController(BookService bookService)
        {
            _bookService = bookService;
        }

        [HttpGet]
        public ActionResult<List<Book>> Get()
        {
            return _bookService.Get();
        }

        [HttpGet("{id:length(24)}", Name = "GetBook")]
        public ActionResult<Book> Get(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            return book;
        }

        [HttpPost]
        public ActionResult<Book> Create(Book book)
        {
            _bookService.Create(book);

            return CreatedAtRoute("GetBook", new { id = book.Id.ToString() }, book);
        }

        [HttpPut("{id:length(24)}")]
        public IActionResult Update(string id, Book bookIn)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            _bookService.Update(id, bookIn);

            return NoContent();
        }

        [HttpDelete("{id:length(24)}")]
        public IActionResult Delete(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
```

```
            if (book == null)
            {
                return NotFound();
            }

            _bookService.Remove(book.Id);

            return NoContent();
        }
    }
}
```

前面的 Web API 控制器：

- 使用 `BookService` 类执行 CRUD 操作。
- 包含操作方法以支持 GET、POST、PUT 和 DELETE HTTP 请求。

2. 生成并运行应用。

3. 在浏览器中导航到 `http://localhost:<port>/api/books`。将显示下面的 JSON 响应：

```
[
  {
    "id":"5bfd996f7b8e48dc15ff215d",
    "bookName":"Design Patterns",
    "price":54.93,
    "category":"Computers",
    "author":"Ralph Johnson"
  },
  {
    "id":"5bfd996f7b8e48dc15ff215e",
    "bookName":"Clean Code",
    "price":43.15,
    "category":"Computers",
    "author":"Robert C. Martin"
  }
]
```

后续步骤

有关构建 ASP.NET Core Web API 的详细信息，请参阅以下资源：

- 使用 ASP.NET Core 构建 Web API
- ASP.NET Core Web API 中控制器操作的返回类型

带有 Swagger/OpenAPI 的 ASP.NET Core Web API 帮助页

2018/12/14 • • Edit Online

作者：Christoph Nienaber 和 Rico Suter

使用 Web API 时，了解其各种方法对开发人员来说可能是一项挑战。Swagger 也称为OpenAPI，解决了为 Web API 生成有用文档和帮助页的问题。它具有诸如交互式文档、客户端 SDK 生成和 API 可发现性等优点。

本文展示了 Swashbuckle.AspNetCore 和 NSwag .NET Swagger 实现：

- **Swashbuckle.AspNetCore** 是一个开源项目，用于生成 ASP.NET Core Web API 的 Swagger 文档。
- **NSwag** 是另一个用于生成 Swagger 文档并将 Swagger UI 或 ReDoc 集成到 ASP.NET Core Web API 中的开源项目。此外，NSwag 还提供了为 API 生成 C# 和 TypeScript 客户端代码的方法。

什么是 Swagger/OpenAPI？

Swagger 是一个与语言无关的规范，用于描述 REST API。Swagger 项目已捐赠给 OpenAPI 计划，现在它被称为开放 API。这两个名称可互换使用，但 OpenAPI 是首选。它允许计算机和人员了解服务的功能，而无需直接访问实现（源代码、网络访问、文档）。其中一个目标是尽量减少连接取消关联的服务所需的工作量。另一个目标是减少准确记录服务所需的时间。

Swagger 规范 (swagger.json)

Swagger 流的核心是 Swagger 规范，默认情况下是名为 swagger.json 的文档。它由 Swagger 工具链（或其第三方实现）根据你的服务生成。它描述了 API 的功能以及使用 HTTP 对其进行访问的方式。它驱动 Swagger UI，并由工具链用来启用发现和客户端代码生成。下面是为简洁起见而缩减的 Swagger 规范的示例：

```
{
   "swagger": "2.0",
   "info": {
       "version": "v1",
       "title": "API V1"
   },
   "basePath": "/",
   "paths": {
       "/api/Todo": {
           "get": {
               "tags": [
                   "Todo"
               ],
               "operationId": "ApiTodoGet",
               "consumes": [],
               "produces": [
                   "text/plain",
                   "application/json",
                   "text/json"
               ],
               "responses": {
                   "200": {
                       "description": "Success",
                       "schema": {
                           "type": "array",
                           "items": {
                               "$ref": "#/definitions/TodoItem"
                           }
                       }
                   }
                }
           },
           "post": {
               ...
           }
       },
       "/api/Todo/{id}": {
           "get": {
               ...
           },
           "put": {
               ...
           },
           "delete": {
               ...
   },
   "definitions": {
       "TodoItem": {
           "type": "object",
            "properties": {
                "id": {
                    "format": "int64",
                    "type": "integer"
                },
                "name": {
                    "type": "string"
                },
                "isComplete": {
                    "default": false,
                    "type": "boolean"
                }
            }
       }
   },
   "securityDefinitions": {}
}
```

Swagger UI

Swagger UI 提供了基于 Web 的 UI，它使用生成的 Swagger 规范提供有关服务的信息。Swashbuckle 和 NSwag 均包含 Swagger UI 的嵌入式版本，因此可使用中间件注册调用将该嵌入式版本托管在 ASP.NET Core 应用中。Web UI 如下所示：

控制器中的每个公共操作方法都可以从 UI 中进行测试。单击方法名称可以展开该部分。添加所有必要的参数，然后单击“试试看!”。

Todo

Show/Hide | List Operations | Expand Operations

GET /api/Todo

Response Class (Status 200)

Success

Model Example Value

```
[
  {
    "id": 0,
    "name": "string",
    "isComplete": false
  }
]
```

Response Content Type text/plain

Try it out! Hide Response

Curl

```
curl -X GET --header 'Accept: application/json' 'http://localhost:60201/api/Todo'
```

Request URL

```
http://localhost:60201/api/Todo
```

Response Body

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

Response Code

```
200
```

Response Headers

```
{
  "date": "Thu, 31 Aug 2017 17:29:04 GMT",
  "server": "Kestrel",
  "transfer-encoding": "chunked",
  "content-type": "application/json; charset=utf-8"
}
```

NOTE

用于屏幕截图的 Swagger UI 版本是版本 2。有关版本 3 的示例，请参阅 Petstore 示例。

后续步骤

- Swashbuckle 入门
- NSwag 入门

Swashbuckle 和 ASP.NET Core 入门

2019/1/10 • • Edit Online

作者：Shayne Boyer 和 Scott Addie

查看或下载示例代码（如何下载）

Swashbuckle 有三个主要组成部分：

- Swashbuckle.AspNetCore.Swagger：将 `SwaggerDocument` 对象公开为 JSON 终结点的 Swagger 对象模型和中间件。
- Swashbuckle.AspNetCore.SwaggerGen：从路由、控制器和模型直接生成 `SwaggerDocument` 对象的 Swagger 生成器。它通常与 Swagger 终结点中间件结合，以自动公开 Swagger JSON。
- Swashbuckle.AspNetCore.SwaggerUI：Swagger UI 工具的嵌入式版本。它解释 Swagger JSON 以构建描述 Web API 功能的可自定义的丰富体验。它包括针对公共方法的内置测试工具。

包安装

可以使用以下方法来添加 Swashbuckle：

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code
- .NET Core CLI

- 从“程序包管理器控制台”窗口：
  - 转到“视图” > “其他窗口” > “程序包管理器控制台”
  - 导航到包含 TodoApi.csproj 文件的目录
  - 请执行以下命令：

    ```
    Install-Package Swashbuckle.AspNetCore
    ```

- 从“管理 NuGet 程序包”对话框中：
  - 右键单击“解决方案资源管理器” > “管理 NuGet 包”中的项目
  - 将“包源”设置为“nuget.org”
  - 在搜索框中输入“Swashbuckle.AspNetCore”
  - 从“浏览”选项卡中选择“Swashbuckle.AspNetCore”包，然后单击“安装”

添加并配置 Swagger 中间件

将 Swagger 生成器添加到 `Startup.ConfigureServices` 方法中的服务集合中：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });
    });
}
```

导入以下命名空间以使用 `Info` 类：

```
using Swashbuckle.AspNetCore.Swagger;
```

在 `Startup.Configure` 方法中，启用中间件为生成的 JSON 文档和 Swagger UI 提供服务：

```
public void Configure(IApplicationBuilder app)
{
    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
    // specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseMvc();
}
```

前面的 `UseSwaggerUI` 方法调用启用了静态文件中间件。如果以 .NET Framework 或 .NET Core 1.x 为目标，请将 Microsoft.AspNetCore.StaticFiles NuGet 包添加到项目。

启动应用，并导航到 `http://localhost:<port>/swagger/v1/swagger.json`。生成的描述终结点的文档显示在 Swagger 规范 (swagger.json) 中。

可在 `http://localhost:<port>/swagger` 找到 Swagger UI。通过 Swagger UI 浏览 API，并将其合并其他计划中。

TIP

要在应用的根（`http://localhost:<port>/`）处提供 Swagger UI，请将 `RoutePrefix` 属性设置为空字符串：

```
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    c.RoutePrefix = string.Empty;
});
```

如果使用目录及 IIS 或反向代理，请使用 `./` 前缀将 Swagger 终结点设置为相对路径。例如 `./swagger/v1/swagger.json`。使用 `/swagger/v1/swagger.json` 指示应用在 URL 的真实根目录中查找 JSON 文件（如果使用，加上路由前缀）。例如，请使用 `http://localhost:<port>/<route_prefix>/swagger/v1/swagger.json` 而不是 `http://localhost:<port>/<virtual_directory>/<route_prefix>/swagger/v1/swagger.json`。

自定义和扩展

Swagger 提供了为对象模型进行归档和自定义 UI 以匹配你的主题的选项。

API 信息和说明

传递给 `AddSwaggerGen` 方法的配置操作会添加诸如作者、许可证和说明的信息：

```
// Register the Swagger generator, defining 1 or more Swagger documents
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new Info
    {
        Version = "v1",
        Title = "ToDo API",
        Description = "A simple example ASP.NET Core Web API",
        TermsOfService = "None",
        Contact = new Contact
        {
            Name = "Shayne Boyer",
            Email = string.Empty,
            Url = "https://twitter.com/spboyer"
        },
        License = new License
        {
            Name = "Use under LICX",
            Url = "https://example.com/license"
        }
    });
});
```

Swagger UI 显示版本的信息：

XML 注释

可使用以下方法启用 XML 注释：

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code
- .NET Core CLI

- 在“解决方案资源管理器”中右键单击该项目，然后选择“编辑 <project_name>.csproj”。
- 手动将突出显示的行添加到 .csproj 文件：

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

- 右键单击“解决方案资源管理器”中的项目，再选择“属性”。
- 选中“生成”选项卡的“输出”部分下的“XML 文档文件”框。

启用 XML 注释，为未记录的公共类型和成员提供调试信息。警告消息指示未记录的类型和成员。例如，以下消息指示违反警告代码 1591：

```
warning CS1591: Missing XML comment for publicly visible type or member 'TodoController.GetAll()'
```

要在项目范围内取消警告，请定义要在项目文件中忽略的以分号分隔的警告代码列表。将警告代码追加到 `$(NoWarn);` 也会应用 C# 默认值。

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

```
<PropertyGroup>
  <DocumentationFile>bin\$(Configuration)\$(TargetFramework)\$(AssemblyName).xml</DocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

要仅针对特定成员取消警告，请将代码附入 #pragma warning 预处理程序指令中。此方法对于不应通过 API 文档公开的代码非常有用。在以下示例中，将忽略整个 `Program` 类的警告代码 CS1591。在类定义结束时还原警告代码的强制执行。使用逗号分隔的列表指定多个警告代码。

```
namespace TodoApi
{
#pragma warning disable CS1591
    public class Program
    {
        public static void Main(string[] args) =>
            BuildWebHost(args).Run();

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
#pragma warning restore CS1591
}
```

配置 Swagger 以使用生成的 XML 文件。对于 Linux 或非 Windows 操作系统，文件名和路径区分大小写。例如，“TodoApi.XML”文件在 Windows 上有效，但在 CentOS 上无效。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info
        {
            Version = "v1",
            Title = "ToDo API",
            Description = "A simple example ASP.NET Core Web API",
            TermsOfService = "None",
            Contact = new Contact
            {
                Name = "Shayne Boyer",
                Email = string.Empty,
                Url = "https://twitter.com/spboyer"
            },
            License = new License
            {
                Name = "Use under LICX",
                Url = "https://example.com/license"
            }
        });

        // Set the comments path for the Swagger JSON and UI.
        var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
        var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
        c.IncludeXmlComments(xmlPath);
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info
        {
            Version = "v1",
            Title = "ToDo API",
            Description = "A simple example ASP.NET Core Web API",
            TermsOfService = "None",
            Contact = new Contact
            {
                Name = "Shayne Boyer",
                Email = string.Empty,
                Url = "https://twitter.com/spboyer"
            },
            License = new License
            {
                Name = "Use under LICX",
                Url = "https://example.com/license"
            }
        });

        // Set the comments path for the Swagger JSON and UI.
        var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
        var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
        c.IncludeXmlComments(xmlPath);
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info
        {
            Version = "v1",
            Title = "ToDo API",
            Description = "A simple example ASP.NET Core Web API",
            TermsOfService = "None",
            Contact = new Contact
            {
                Name = "Shayne Boyer",
                Email = string.Empty,
                Url = "https://twitter.com/spboyer"
            },
            License = new License
            {
                Name = "Use under LICX",
                Url = "https://example.com/license"
            }
        });

        // Set the comments path for the Swagger JSON and UI.
        var xmlFile = $"{Assembly.GetEntryAssembly().GetName().Name}.xml";
        var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
        c.IncludeXmlComments(xmlPath);
    });
}
```

在上述代码中，反射用于生成与 Web API 项目相匹配的 XML 文件名。AppContext.BaseDirectory属性用于构造 XML 文件的路径。

通过向节标题添加说明，将三斜杠注释添加到操作增强了 Swagger UI。执行 `Delete` 操作前添加 <summary> 元素：

```
/// <summary>
/// Deletes a specific TodoItem.
/// </summary>
/// <param name="id"></param>
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.Find(id);

    if (todo == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todo);
    _context.SaveChanges();

    return NoContent();
}
```

Swagger UI 显示上述代码的 `<summary>` 元素的内部文本：

生成的 JSON 架构驱动 UI：

```
"delete": {
    "tags": [
        "Todo"
    ],
    "summary": "Deletes a specific TodoItem.",
    "operationId": "ApiTodoByIdDelete",
    "consumes": [],
    "produces": [],
    "parameters": [
        {
            "name": "id",
            "in": "path",
            "description": "",
            "required": true,
            "type": "integer",
            "format": "int64"
        }
    ],
    "responses": {
        "200": {
            "description": "Success"
        }
    }
}
```

将 <remarks> 元素添加到 `Create` 操作方法文档。它可以补充 `<summary>` 元素中指定的信息，并提供更可靠的 Swagger UI。`<remarks>` 元素内容可包含文本、JSON 或 XML。

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(typeof(TodoItem), 201)]
[ProducesResponseType(400)]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
public ActionResult<TodoItem> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

请注意带这些附加注释的 UI 增强功能：

数据注释

使用 System.ComponentModel.DataAnnotations 命名空间中的属性来修饰模型，以帮助驱动 Swagger UI 组件。

将 `[Required]` 属性添加到 `TodoItem` 类的 `Name` 属性：

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }

        [Required]
        public string Name { get; set; }

        [DefaultValue(false)]
        public bool IsComplete { get; set; }
    }
}
```

此属性的状态更改 UI 行为并更改基础 JSON 架构：

```
"definitions": {
    "TodoItem": {
        "required": [
            "name"
        ],
        "type": "object",
        "properties": {
            "id": {
                "format": "int64",
                "type": "integer"
            },
            "name": {
                "type": "string"
            },
            "isComplete": {
                "default": false,
                "type": "boolean"
            }
        }
    }
},
```

将 `[Produces("application/json")]` 属性添加到 API 控制器。这样做的目的是声明控制器的操作支持 application/json 的响应内容类型：

```
[Produces("application/json")]
[Route("api/[controller]")]
public class TodoController : ControllerBase
{
    private readonly TodoContext _context;
```

```
[Produces("application/json")]
[Route("api/[controller]")]
[ApiController]
public class TodoController : ControllerBase
{
    private readonly TodoContext _context;
```

“响应内容类型”下拉列表选此内容类型作为控制器的默认 GET 操作：

随着 Web API 中的数据注释的使用越来越多，UI 和 API 帮助页变得更具说明性和更为有用。

描述响应类型

使用 Web API 的开发人员最关心的问题是返回的内容，特别是响应类型和错误代码（如果不标准）。在 XML 注释和数据注释中表示响应类型和错误代码。

`Create` 操作成功后返回 HTTP 201 状态代码。发布的请求正文为 NULL 时，将返回 HTTP 400 状态代码。如果 Swagger UI 中没有提供合适的文档，那么使用者会缺少对这些预期结果的了解。在以下示例中，通过添加突出显示的行解决此问题：

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(typeof(TodoItem), 201)]
[ProducesResponseType(400)]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
public ActionResult<TodoItem> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

Swagger UI 现在清楚地记录预期的 HTTP 响应代码：

在 ASP.NET Core 2.2 或更高版本中，约定可以用作使用 `[ProducesResponseType]` 显式修饰各操作的替代方法。有关更多信息，请参见使用 Web API 约定。

自定义 UI

股票 UI 既实用又可呈现。但是，API 文档页应代表品牌或主题。将 Swashbuckle 组件标记为需要添加资源以提供静态文件，并构建文件夹结构以托管这些文件。

如果以 .NET Framework 或 .NET Core 1.x 为目标，请将 Microsoft.AspNetCore.StaticFiles NuGet 包添加到项目：

```
<PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.0.0" />
```

如果以 .NET Core 2.x 为目标并使用元包，则已安装上述 NuGet 包。

启用静态文件中间件：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
    // specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseMvc();
}
```

从 Swagger UI GitHub 存储库中获取 dist 文件夹的内容。此文件夹包含 Swagger UI 页必需的资产。

创建 wwwroot/swagger/ui 文件夹，然后将 dist 文件夹的内容复制到其中。

使用以下 CSS 在 wwwroot/swagger/ui 中创建 custom.css 文件，以自定义页面标题：

```
.swagger-ui .topbar {
    background-color: #000;
    border-bottom: 3px solid #547f00;
}
```

引用其他 CSS 文件后，引用 index.html 文件中的“custom.css”：

```
<link href="https://fonts.googleapis.com/css?
family=Open+Sans:400,700|Source+Code+Pro:300,600|Titillium+Web:400,600,700" rel="stylesheet">
<link rel="stylesheet" type="text/css" href="./swagger-ui.css">
<link rel="stylesheet" type="text/css" href="custom.css">
```

浏览到 `http://localhost:<port>/swagger/ui/index.html` 中的 index.html 页。在标题文本框中输入 `http://localhost:<port>/swagger/v1/swagger.json`，然后单击“浏览”按钮。生成的页面如下所示：

还可以对页面执行更多操作。在 Swagger UI GitHub 存储库中查看 UI 资源的完整功能。

NSwag 和 ASP.NET Core 入门

2019/1/10 • • Edit Online

作者：Christoph Nienaber 和 Rico Suter

查看或下载示例代码（如何下载）

查看或下载示例代码（如何下载）

注册 NSwag 中间件，以便：

- 生成已实现的 Web API 的 Swagger 规范。
- 为 Swagger UI 提供服务以浏览和测试 Web API。

若要使用 NSwag ASP.NET Core 中间件，请安装 NSwag.AspNetCore NuGet 包。此包包含用于生成 Swagger 规范、Swagger UI（v2 和 v3）和 ReDoc UI 并为其提供服务的中间件。

此外，强烈建议使用 NSwag 的代码生成功能。请选择下列选项之一以使用代码生成功能：

- 使用 NSwagStudio，这是一款 Windows 桌面应用，用于在 C# 和 TypeScript 中为 API 生成客户端代码。
- 使用 NSwag.CodeGeneration.CSharp 或 NSwag.CodeGeneration.TypeScript NuGet 包在项目中执行代码生成。
- 使用命令行中的 NSwag。
- 使用 NSwag.MSBuild NuGet 包。

功能

使用 NSwag 的主要原因是它不仅能引入 Swagger UI 和 Swagger 生成器，还能利用灵活的代码生成功能。无需现有API — 可使用包含 Swagger 的第三方 API 并让 NSwag 生成客户端实现。无论哪种方式，开发周期都会加快，可以更轻松地适应 API 更改。

包安装

可使用以下方法添加 NuGet 包：

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code
- .NET Core CLI

- 从"程序包管理器控制台"窗口：
  - 转到"视图" > "其他窗口" > "程序包管理器控制台"
  - 导航到包含 TodoApi.csproj 文件的目录
  - 请执行以下命令：

    ```
    Install-Package NSwag.AspNetCore
    ```

- 从"管理 NuGet 程序包"对话框中：
  - 右键单击"解决方案资源管理器" > "管理 NuGet 包"中的项目
  - 将"包源"设置为"nuget.org"

- 在搜索框中输入“NSwag.AspNetCore”
- 从“浏览”选项卡中选择“NSwag.AspNetCore”包，然后单击“安装”

添加并配置 Swagger 中间件

在 `Startup` 类中导入以下命名空间：

```
using NJsonSchema;
using NSwag.AspNetCore;
```

在 `Startup.ConfigureServices` 方法中，注册所需的 Swagger 服务：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger services
    services.AddSwaggerDocument();
}
```

在 `Startup.Configure` 方法中，启用中间件为生成的 Swagger 规范和 Swagger UI v3 提供服务：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    // Register the Swagger generator and the Swagger UI middlewares
    app.UseSwagger();
    app.UseSwaggerUi3();

    app.UseMvc();
}
```

启动应用。导航到 `http://localhost:<port>/swagger` 查看 Swagger UI。导航到 `http://localhost:<port>/swagger/v1/swagger.json` 查看 Swagger 规范。

代码生成

通过 NSwagStudio

- 从官方 GitHub 存储库安装 NSwagStudio。
- 启动 NSwagStudio。在“Swagger 规范 URL”文本框中输入 swagger.json 文件 URL，然后单击“创建本地副本”按钮。
- 选择“CSharp 客户端”作为客户端输出类型。其他选项包括“TypeScript 客户端”和“CSharp Web API 控制器”。使用 Web API 控制器基本上是一种反向生成。它使用服务的规范来重新生成服务。
- 单击“生成输出”按钮。随即生成完整的 TodoApi.NSwag 项目 C# 客户端实现。单击“输出”部分的“CSharp 客户端”选项卡，查看生成的客户端代码：

```
//----------------------
// <auto-generated>
//     Generated using the NSwag toolchain v11.17.3.0 (NJsonSchema v9.10.46.0 (Newtonsoft.Json v9.0.0.0))
(http://NSwag.org)
// </auto-generated>
//----------------------

namespace MyNamespace
{
    #pragma warning disable // Disable all warnings

    [System.CodeDom.Compiler.GeneratedCode("NSwag",
        "11.17.3.0 (NJsonSchema v9.10.46.0 (Newtonsoft.Json v9.0.0.0))")]
    public partial class TodoClient
    {
        private string _baseUrl = "http://localhost:50499";
        private System.Lazy<Newtonsoft.Json.JsonSerializerSettings> _settings;

        public TodoClient()
        {
            _settings = new System.Lazy
                <Newtonsoft.Json.JsonSerializerSettings>(() =>
            {
                var settings = new Newtonsoft.Json.JsonSerializerSettings();
                UpdateJsonSerializerSettings(settings);
                return settings;
            });
        }

        public string BaseUrl
        {
            get { return _baseUrl; }
            set { _baseUrl = value; }
        }

        // code omitted for brevity
```

TIP

基于“CSharp 客户端”选项卡的“设置”选项卡中定义的设置，生成 C# 客户端代码。修改设置以执行任务，例如默认命名空间重命名和同步方法生成。

- 将生成的 C# 代码复制到客户端项目的某个文件中（例如，Xamarin.Forms 应用）。
- 开始使用 Web API：

```
var todoClient = new TodoClient();

// Gets all to-dos from the API
var allTodos = await todoClient.GetAllAsync();

// Create a new TodoItem, and save it in the API
var createdTodo = await todoClient.CreateAsync(new TodoItem());

// Get a single to-do by ID
var foundTodo = await todoClient.GetByIdAsync(1);
```

NOTE

可将基 URL 和/或 HTTP 客户端注入 API 客户端。最佳做法是始终重复使用 HttpClient。

生成客户端代码的其他方法

可通过其他更适合你的工作流的方式生成客户端代码：

- MSBuild
- 在代码中生成
- 通过 T4 模板生成

自定义

Swagger 提供用于记录对象模型以便于使用 Web API 的选项。

API 信息和说明

在 `Startup.Configure` 方法中，传递给 `UseSwagger` 方法的配置操作会添加诸如作者、许可证和说明的信息：

```

```

Swagger UI 显示版本的信息：

XML 注释

可使用以下方法启用 XML 注释：

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code

- 在“解决方案资源管理器”中右键单击该项目，然后选择“编辑 <project_name>.csproj”。
- 手动将突出显示的行添加到 .csproj 文件：

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

- 右键单击“解决方案资源管理器”中的项目，然后选择“属性”

- 查看“生成”选项卡的“输出”部分下的“XML 文档文件”框

数据注释

NSwag 使用反射，建议使用 IActionResult 作为 Web API 操作的返回类型。因此，NSwag 无法推断正在执行的操作和返回的结果。请看下面的示例：

```
[HttpPost]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

上述操作返回 `IActionResult`，但在操作内部返回 CreatedAtRoute 或 BadRequest。使用数据注释告知客户端此操作会返回哪些 HTTP 状态代码。使用以下属性修饰该操作：

```
[ProducesResponseType(typeof(TodoItem), 201)]   // Created
[ProducesResponseType(400)]                     // BadRequest
```

NSwag 使用反射，建议的 Web API 操作返回类型为 ActionResult<T>。因此，NSwag 仅可以推断 `T` 定义的返回类型。无法推断操作中其他可能的返回类型。请看下面的示例：

```
[HttpPost]
public ActionResult<TodoItem> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

上述操作将返回 `ActionResult<T>`。在操作中，它将返回 CreatedAtRoute。由于使用 [ApiController] 属性修饰控制器，所以也可能出现 BadRequest 响应。有关详细信息，请参阅自动 HTTP 400 响应。使用数据注释告知客户端此操作会返回哪些 HTTP 状态代码。使用以下属性修饰该操作：

```
[ProducesResponseType(201)]     // Created
[ProducesResponseType(400)]     // BadRequest
```

在 ASP.NET Core 2.2 或更高版本中，约定可以用作使用 `[ProducesResponseType]` 显式修饰各操作的替代方法。有关更多信息，请参见使用 Web API 约定。

Swagger 生成器现在可准确地描述此操作，且生成的客户端知道调用终结点时收到的内容。强烈建议使用这些属性来修饰所有操作。有关 API 操作应返回的 HTTP 响应的指导原则，请参阅 RFC 7231 规范。

ASP.NET Core Web API 中控制器操作的返回类型

2018/11/28 • • Edit Online

作者：Scott Addie

查看或下载示例代码（如何下载）

ASP.NET Core 提供以下 Web API 控制器操作返回类型选项：

- 特定类型
- IActionResult
- ActionResult<T>

- 特定类型
- IActionResult

本文档说明每种返回类型的最佳适用情况。

特定类型

最简单的操作返回基元或复杂数据类型（如 `string` 或自定义对象类型）。请参考以下操作，该操作返回自定义 `Product` 对象的集合：

```
[HttpGet]
public IEnumerable<Product> Get()
{
    return _repository.GetProducts();
}
```

在执行操作期间无需防范已知条件，返回特定类型即可满足要求。上述操作不接受任何参数，因此不需要参数约束验证。

当在操作中需要考虑已知条件时，将引入多个返回路径。在此类情况下，通常会将 ActionResult 返回类型和基元或复杂返回类型混合。要支持此类操作，必须使用 IActionResult 或 ActionResult<T>。

IActionResult 类型

当操作中可能有多个 ActionResult 返回类型时，适合使用 IActionResult 返回类型。`ActionResult` 类型表示多种 HTTP 状态代码。属于此类别的一些常见返回类型包括：BadRequestResult (400)、NotFoundResult (404) 和 OkObjectResult (200)。

由于操作中有多个返回类型和路径，因此必须自由使用 [ProducesResponseType] 特性。此特性可针对 Swagger 等工具生成的 API 帮助页生成更多描述性响应详细信息。`[ProducesResponseType]` 指示操作将返回的已知类型和 HTTP 状态代码。

同步操作

请参考以下同步操作，其中有两种可能的返回类型：

```
[HttpGet("{id}")]
[ProducesResponseType(200, Type = typeof(Product))]
[ProducesResponseType(404)]
public IActionResult GetById(int id)
{
    if (!_repository.TryGetProduct(id, out var product))
    {
        return NotFound();
    }

    return Ok(product);
}
```

在上述操作中，当 `id` 代表的产品不在基础数据存储中时，则返回 404 状态代码。调用 NotFound 帮助方法作为 `return new NotFoundResult();` 的快捷方式。如果产品确实存在，则返回代表有效负载的 `Product` 对象和状态代码 200。调用 Ok 帮助方法作为 `return new OkObjectResult(product);` 的快捷方式。

异步操作

请参考以下异步操作，其中有两种可能的返回类型：

```
[HttpPost]
[ProducesResponseType(201, Type = typeof(Product))]
[ProducesResponseType(400)]
public async Task<IActionResult> CreateAsync([FromBody] Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    await _repository.AddProductAsync(product);

    return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);
}
```

在上述操作中，模型验证失败时返回 400 状态代码，并且调用了 BadRequest 帮助方法。例如，以下模型指示请求必须提供 `Name` 属性和值。因此，若请求中未能提供合适的 `Name`，则会导致模型验证失败。

```
public class Product
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }

    [Required]
    public string Description { get; set; }
}
```

上述操作的其他已知返回代码是 201，由 CreatedAtAction 帮助方法生成。在此路径中，返回了 `Product` 对象。

ActionResult<T> 类型

ASP.NET Core 2.1 引入了面向 Web API 控制器操作的 ActionResult<T> 返回类型。它支持返回从 ActionResult 派生的类型或返回特定类型。`ActionResult<T>` 通过 IActionResult 类型可提供以下优势：

- 可排除 [ProducesResponseType] 特性的 `Type` 属性。例如，`[ProducesResponseType(200, Type = typeof(Product))]` 可简化为 `[ProducesResponseType(200)]`。此操作的预期返回类型改为根据 `ActionResult<T>` 中的 `T` 进行推断。

- 隐式强制转换运算符支持将 `T` 和 `ActionResult` 均转换为 `ActionResult<T>`。将 `T` 转换为 ObjectResult，也就是将 `return new ObjectResult(T);` 简化为 `return T;`。

C# 不支持对接口使用隐式强制转换运算符。因此，必须使用 `ActionResult<T>`，才能将接口转换为具体类型。例如，在下面的示例中，使用 `IEnumerable` 不起作用：

```
[HttpGet]
public ActionResult<IEnumerable<Product>> Get()
{
    return _repository.GetProducts();
}
```

上面代码的一种修复方法是返回 `_repository.GetProducts().ToList();`。

大多数操作具有特定返回类型。执行操作期间可能出现意外情况，不返回特定类型就是其中之一。例如，操作的输入参数可能无法通过模型验证。在此情况下，通常会返回相应的 `ActionResult` 类型，而不是特定类型。

同步操作

请参考以下同步操作，其中有两种可能的返回类型：

```
[HttpGet("{id}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public ActionResult<Product> GetById(int id)
{
    if (!_repository.TryGetProduct(id, out var product))
    {
        return NotFound();
    }

    return product;
}
```

在上述代码中，当产品不在数据库中时则返回状态代码 404。如果产品确实存在，则返回相应的 `Product` 对象。ASP.NET Core 2.1 之前，`return product;` 行是 `return Ok(product);`。

> **TIP**
> 从 ASP.NET Core 2.1 开始，使用 `[ApiController]` 特性修饰控制器类时，将启用操作参数绑定源推理。与路由模板中的名称相匹配的参数名称将通过请求路由数据自动绑定。因此，不会使用 [FromRoute] 特性对上述操作中的 `id` 参数进行显示批注。

异步操作

请参考以下异步操作，其中有两种可能的返回类型：

```
[HttpPost]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
public async Task<ActionResult<Product>> CreateAsync(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    await _repository.AddProductAsync(product);

    return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);
}
```

如果模型验证失败，则调用 BadRequest 方法以返回 400 状态代码。向它传递包含特定验证错误的 ModelState 属性。如果模型验证成功，则在数据库中创建产品。返回 201 状态代码。

> **TIP**
> 从 ASP.NET Core 2.1 开始，使用 `[ApiController]` 特性修饰控制器类时，将启用操作参数绑定源推理。复杂类型参数通过请求正文自动绑定。因此，不会使用 [FromBody] 特性对前面操作中的 `product` 参数进行显示批注。

其他资源

- 在 ASP.NET Core MVC 中使用控制器处理请求
- ASP.NET Core MVC 中的模型验证
- 带有 Swagger/OpenAPI 的 ASP.NET Core Web API 帮助页

设置 ASP.NET Core Web API 中响应数据的格式

2018/10/30 • • Edit Online

作者：Steve Smith

ASP.NET Core MVC 包含对通过固定格式或根据客户端规范来设置响应数据格式的内置支持。

查看或下载示例代码（如何下载）

特定于格式的操作结果

一些操作结果类型特定于特殊格式，例如 `JsonResult` 和 `ContentResult`。操作可以返回始终以特定方式进行格式设置的特定结果。例如，返回 `JsonResult` 将返回 JSON 格式的数据，而不考虑客户端首选项。同样，返回 `ContentResult` 将返回纯文本格式的字符串数据（仅返回字符串也是如此）。

> **NOTE**
>
> 不需要操作返回任何特定类型；MVC 支持任意对象返回值。如果操作返回 `IActionResult` 实现且控制器继承自 `Controller`，则开发人员具有对应于多种选择的帮助程序方法。返回非 `IActionResult` 类型对象的操作结果将使用相应的 `IOutputFormatter` 实现来进行序列化。

若要从继承自 `Controller` 基类的控制器返回特定格式的数据，请使用内置的帮助程序方法 `Json` 来返回 JSON，并使用 `Content` 来返回纯文本。操作方法应返回特定的结果类型（例如 `JsonResult`）或 `IActionResult`）。

返回 JSON 格式的数据：

```
// GET: api/authors
[HttpGet]
public JsonResult Get()
{
    return Json(_authorRepository.List());
}
```

此操作的示例响应：

请注意，响应的内容类型为 `application/json`，网络请求的列表和"响应标头"部分均有显示。另请注意"响应标头"部分 Accept 标头的浏览器（此示例中为 Microsoft Edge）所呈现的选项列表。当前技术忽略此标头；下面将讨论遵守它的情况。

若要返回纯文本格式数据，请使用 `ContentResult` 和 `Content` 帮助程序：

```
// GET api/authors/about
[HttpGet("About")]
public ContentResult About()
{
    return Content("An API listing authors of docs.asp.net.");
}
```

此操作的响应：

请注意此示例中返回的 `Content-Type` 为 `text/plain`。还可以仅使用字符串响应类型来达到此相同行为：

```
// GET api/authors/version
[HttpGet("version")]
public string Version()
{
    return "Version 1.0.0";
}
```

> **TIP**
> 对于有多个返回类型或选项的重要操作（例如基于所执行操作的结果的不同 HTTP 状态代码），请首选 `IActionResult` 作为返回类型。

内容协商

当客户端指定 Accept 标头时，会发生内容协商（简称 conneg）。ASP.NET Core MVC 使用的默认格式是 JSON。内容协商由 `ObjectResult` 实现。它还内置于从帮助程序方法（全部基于 `ObjectResult`）返回的特定于状态代码的操作结果中。还可以返回一个模型类型（已定义为数据传输类型的类），框架将自动将其打包在 `ObjectResult` 中。

以下操作方法使用 `Ok` 和 `NotFound` 帮助程序方法：

```
// GET: api/authors/search?namelike=th
[HttpGet("Search")]
public IActionResult Search(string namelike)
{
    var result = _authorRepository.GetByNameSubstring(namelike);
    if (!result.Any())
    {
        return NotFound(namelike);
    }
    return Ok(result);
}
```

将返回 JSON 格式的响应，除非请求了另一个格式且服务器可以返回所请求格式。可以使用 Fiddler 等工具创建包括 Accept 标头的请求并指定另一种格式。在此情况下，如果服务器有可以生成所请求格式的响应的格式化程序，则结果会以服务器首选的格式返回。

在上面的屏幕快照中，使用 Fiddler Composer 生成请求，指定 `Accept: application/xml`。默认情况下，ASP.NET Core MVC 仅支持 JSON。所以，即使指定另一种格式，返回的结果仍然是 JSON 格式。将在下一节中看到如何添加其他格式化程序。

控制器操作可以返回 POCO（普通旧 CLR 对象），在这种情况下，ASP.NET Core MVC 将自动创建打包对象的 `ObjectResult`。客户端将获取设有格式的序列化对象（默认为 JSON 格式，可以配置 XML 或其他格式）。如果返回的对象为 `null`，那么框架将返回 `204 No Content` 响应。

返回对象类型：

```
// GET api/authors/ardalis
[HttpGet("{alias}")]
public Author Get(string alias)
{
    return _authorRepository.GetByAlias(alias);
}
```

在示例中，请求有效作者别名将收到具有作者数据的“200 正常”响应。请求无效别名将收到“204 无内容”响应。下面列出了以 XML 和 JSON 格式显示响应的屏幕快照。

内容协商过程

内容协商仅在 `Accept` 标头出现在请求中时发生。请求包含 accept 标头时，框架会以最佳顺序枚举 accept 标头中的媒体类型，并且尝试查找可以生成一种由 accept 标头指定格式的响应的格式化程序。如果未找到可以满足客户端请求的格式化程序，框架将尝试找到第一个可以生成响应的格式化程序（除非开发人员配置 `MvcOptions` 上的选项以返回“406 不可接受”）。如果请求指定 XML，但是未配置 XML 格式化程序，那么将使用 JSON 格式化程序。一般来说，如果没有配置可以提供所请求格式的格式化程序，那么使用第一个可以设置对象格式的格式化程序。如果不提供任何标头，则将使用第一个可以处理要返回的对象的格式化程序来序列化响应。在此情况下，没有任何协商发生 - 服务器确定将使用的格式。

> **NOTE**
>
> 如果 Accept 标头包含 `*/*`，则将忽略该标头，除非 `RespectBrowserAcceptHeader` 在 `MvcOptions` 上设置为 true。

浏览器和内容协商

不同于传统 API 客户端，Web 浏览器倾向于提供包括各种格式（含通配符）的 `Accept` 标头。默认情况下，当框架检测到请求来自浏览器时，它将忽略 `Accept` 标头转而以应用程序的配置默认格式（JSON，除非有其他配置）返回内容。这在使用不同浏览器使用 API 时提供更一致的体验。

如果首选应用程序服从浏览器 accept 标头，则可以将此配置为 MVC 配置的一部分，方法是在 Startup.cs 中以 `ConfigureServices` 方法将 `RespectBrowserAcceptHeader` 设置为 `true`。

```
services.AddMvc(options =>
{
    options.RespectBrowserAcceptHeader = true; // false by default
});
```

配置格式化程序

如果应用程序需要支持默认 JSON 格式以外的其他格式，那么可以添加 NuGet 包并配置 MVC 来支持它们。输入和输出的格式化程序不同。输入格式化程序由模型绑定使用；输出格式化程序用来设置响应格式。还可以配置自定义格式化程序。

添加 XML 格式支持

若要添加对 XML 格式的支持，请安装 `Microsoft.AspNetCore.Mvc.Formatters.Xml` NuGet 包。

将 XmlSerializerFormatters 添加到 Startup.cs 中 MVC 的配置：

```
services.AddMvc()
    .AddXmlSerializerFormatters();
```

或者，可以仅添加输出格式化程序：

```
services.AddMvc(options =>
{
    options.OutputFormatters.Add(new XmlSerializerOutputFormatter());
});
```

这两个方法将使用 `System.Xml.Serialization.XmlSerializer` 来序列化结果。如果愿意，可以通过添加相关联的格式化程序使用 `System.Runtime.Serialization.DataContractSerializer`：

```
services.AddMvc(options =>
{
    options.OutputFormatters.Add(new XmlDataContractSerializerOutputFormatter());
});
```

一旦添加了对 XML 格式的支持，控制器方法应基于请求的 `Accept` 标头返回相应的格式，如以下 Fiddler 示例所示：

可以在"检查器"选项卡中看到使用 `Accept: application/xml` 标头集做出 Raw GET 请求。响应窗格显示 `Content-Type: application/xml` 标头，且 `Author` 对象已序列化为 XML。

使用“编辑器”选项卡修改请求以指定 `Accept` 标头中的 `application/json`。执行请求，响应格式将设为 JSON：

在此屏幕快照中，可以看到请求了设置 `Accept: application/json` 的标头，且响应也将它指定为其 `Content-Type`。`Author` 对象以 JSON 格式显示在响应正文中。

强制执行特定格式

如果需要限制特定操作的响应格式，那么可以应用 `[Produces]` 筛选器。`[Produces]` 筛选器指定特定操作（或控制器）的响应格式。如同大多筛选器，这可以在操作层面、控制器层面或全局范围内应用。

```
[Produces("application/json")]
public class AuthorsController
```

`[Produces]` 筛选器将强制 `AuthorsController` 中的所有操作返回 JSON 格式的响应，即使已经为应用程序配置其他格式化程序且客户端提供了请求其他可用格式的 `Accept` 标头也是如此。请参阅筛选器了解详细信息，包括如何全局应用筛选器。

特例格式化程序

一些特例是使用内置格式化程序实现的。默认情况下，`string` 返回类型的格式将设为 text/plain（如果通过 `Accept` 标头请求则为 text/html）。可以通过删除 `TextOutputFormatter` 删除此行为。在 Startup.cs 中通过 `Configure` 方法删除格式化程序（如下所示）。有模型对象返回类型的操作将在返回 `null` 时返回“204 无内容”响应。可以通过删除 `HttpNoContentOutputFormatter` 删除此行为。以下代码删除 `TextOutputFormatter` 和 `HttpNoContentOutputFormatter`。

```
services.AddMvc(options =>
{
    options.OutputFormatters.RemoveType<TextOutputFormatter>();
    options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();
});
```

没有 `TextOutputFormatter` 时，`string` 返回类型将返回“406 不可接受”等内容。请注意，如果 XML 格式化程序存在，当删除 `TextOutputFormatter` 时，它将设置 `string` 返回类型的格式。

没有 `HttpNoContentOutputFormatter`，null 对象将使用配置的格式化程序来进行格式设置。例如，JSON 格式化程序仅返回正文为 `null` 的响应，而 XML 格式化程序将返回属性设为 `xsi:nil="true"` 的空 XML 元素。

响应格式 URL 映射

客户端可要求特定格式作为 URL 一部分，例如在查询字符串中或在路径的一部分中，或者通过使用特定格式的文

件扩展名（例如 .xml 或 .json）。请求路径的映射必须在 API 使用的路由中指定。例如:

```
[FormatFilter]
public class ProductsController
{
    [Route("[controller]/[action]/{id}.{format?}")]
    public Product GetById(int id)
```

此路由将允许所所请求格式指定为可选文件扩展名。`[FormatFilter]` 属性检查 `RouteData` 中格式值是否存在，并在响应创建时将响应格式映射到相应格式化程序。

路由	格式化程序
`/products/GetById/5`	默认输出格式化程序
`/products/GetById/5.json`	JSON 格式化程序（如配置）
`/products/GetById/5.xml`	XML 格式化程序（如配置）

ASP.NET Core Web API 中的自定义格式化程序

2019/1/10 • • Edit Online

作者：Tom Dykstra

ASP.NET Core MVC 使用 JSON 或 XML，为 Web API 中的数据交换提供内置支持。本文展示如何通过创建自定义格式化程序，添加对其他格式的支持。

查看或下载示例代码（如何下载）

何时使用自定义格式化程序

如果希望内容协商过程支持内置格式化程序（JSON 和 XML）所不支持的内容类型，可使用自定义格式化程序。

例如，如果 Web API 的某些客户端可以处理 Protobuf 格式，你可能想在这些客户端上使用 Protobuf，因为它更高效。或者，你可能希望 Web API 使用 vCard 格式发送联系人姓名和地址，这种格式经常用于交换联系人数据。本文提供的示例应用可实现简单的 vCard 格式化程序。

有关如何使用自定义格式化程序的概述

创建和使用自定义格式化程序的步骤如下：

- 如果想对要发送到客户端的数据进行序列化，则创建输出格式化程序类。
- 如果想对从客户端接收的数据进行反序列化，则创建输入格式化程序类。
- 将格式化程序的实例添加到 MvcOptions 中的 `InputFormatters` 和 `OutputFormatters` 集合。

以下部分针对其中每个步骤提供了指南和代码示例。

如何创建自定义格式化程序类

若要创建格式化程序，请执行以下操作：

- 从相应的基类中派生类。
- 在构造函数中指定有效的媒体类型和编码。
- 重写 `CanReadType` / `CanWriteType` 方法
- 重写 `ReadRequestBodyAsync` / `WriteResponseBodyAsync` 方法

从相应的基类中派生

对于文本媒体类型（例如，vCard），从 TextInputFormatter 或 TextOutputFormatter 基类派生。

```
public class VcardOutputFormatter : TextOutputFormatter
```

有关输入格式化程序示例，请参阅示例应用。

对于二进制类型，从 InputFormatter 或 OutputFormatter 基类派生。

指定有效的媒体类型和编码

在构造函数中，通过添加到 `SupportedMediaTypes` 和 `SupportedEncodings` 集合来指定有效的媒体类型和编码。

```
public VcardOutputFormatter()
{
    SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

    SupportedEncodings.Add(Encoding.UTF8);
    SupportedEncodings.Add(Encoding.Unicode);
}
```

有关输入格式化程序示例，请参阅示例应用。

> **NOTE**
>
> 不能在格式化程序类中执行构造函数依赖关系注入。例如，不能通过向构造函数添加记录器参数来获取记录器。若要访问服务，必须使用传递到方法的上下文对象。下面的代码示例展示了如何执行此操作。

重写 CanReadType/CanWriteType

通过重写 `CanReadType` 或 `CanWriteType` 方法，指定可反序列化为或从其序列化的类型。例如，可能只能从 `Contact` 类型创建 vCard 文本，反之亦然。

```
protected override bool CanWriteType(Type type)
{
    if (typeof(Contact).IsAssignableFrom(type)
        || typeof(IEnumerable<Contact>).IsAssignableFrom(type))
    {
        return base.CanWriteType(type);
    }
    return false;
}
```

有关输入格式化程序示例，请参阅示例应用。

CanWriteResult 方法

在某些情况下，必须重写 `CanWriteResult`，而不是 `CanWriteType`。如果满足以下条件，则使用 `CanWriteResult`：

- 操作方法返回模型类。
- 具有可能在运行时返回的派生类。
- 需要知道操作在运行时返回了哪个派生类。

例如，假设操作方法签名返回 `Person` 类型，但它可能返回从 `Person` 派生的 `Student` 或 `Instructor` 类型。如果希望格式化程序仅处理 `Student` 对象，请检查提供给 `CanWriteResult` 方法的上下文对象中的对象类型。请注意，当操作方法返回 `IActionResult` 时，不必使用 `CanWriteResult`；在这种情况下，`CanWriteType` 方法可接收运行时类型。

重写 ReadRequestBodyAsync/WriteResponseBodyAsync

实际的反序列化或序列化工作在 `ReadRequestBodyAsync` 或 `WriteResponseBodyAsync` 中执行。以下示例中突出显示的行展示了如何从依赖关系注入容器中获取服务（不能从构造函数参数中获取它们）。

```
public override Task WriteResponseBodyAsync(OutputFormatterWriteContext context, Encoding selectedEncoding)
{
    IServiceProvider serviceProvider = context.HttpContext.RequestServices;
    var logger = serviceProvider.GetService(typeof(ILogger<VcardOutputFormatter>)) as ILogger;

    var response = context.HttpContext.Response;

    var buffer = new StringBuilder();
    if (context.Object is IEnumerable<Contact>)
    {
        foreach (Contact contact in context.Object as IEnumerable<Contact>)
        {
            FormatVcard(buffer, contact, logger);
        }
    }
    else
    {
        var contact = context.Object as Contact;
        FormatVcard(buffer, contact, logger);
    }
    return response.WriteAsync(buffer.ToString());
}

private static void FormatVcard(StringBuilder buffer, Contact contact, ILogger logger)
{
    buffer.AppendLine("BEGIN:VCARD");
    buffer.AppendLine("VERSION:2.1");
    buffer.AppendFormat($"N:{contact.LastName};{contact.FirstName}\r\n");
    buffer.AppendFormat($"FN:{contact.FirstName} {contact.LastName}\r\n");
    buffer.AppendFormat($"UID:{contact.ID}\r\n");
    buffer.AppendLine("END:VCARD");
    logger.LogInformation($"Writing {contact.FirstName} {contact.LastName}");
}
```

有关输入格式化程序示例，请参阅示例应用。

如何将 MVC 配置为使用自定义格式化程序

若要使用自定义格式化程序，请将格式化程序类的实例添加到 `InputFormatters` 或 `OutputFormatters` 集合。

```
services.AddMvc(options =>
{
    options.InputFormatters.Insert(0, new VcardInputFormatter());
    options.OutputFormatters.Insert(0, new VcardOutputFormatter());
});
```

按格式化程序的插入顺序对其进行计算。第一个优先。

后续步骤

- GitHub 上的纯文本格式化程序示例代码。
- 此文档的示例应用，它可实现简单的 vCard 输入和输出格式化程序。该应用可读取和写入与以下示例类似的 vCard：

```
BEGIN:VCARD
VERSION:2.1
N:Davolio;Nancy
FN:Nancy Davolio
UID:20293482-9240-4d68-b475-325df4a83728
END:VCARD
```

若要查看 vCard 输出，请运行该应用程序，并向 `http://localhost:63313/api/contacts/`（从 Visual Studio 运行时）或 `http://localhost:5000/api/contacts/`（从命令行运行时）发送具有 Accept 标头“text/vcard”的 Get 请求。

若要将 vCard 添加到内存中联系人集合，请向相同的 URL 发送具有 Content-Type 标头“text/vcard”且正文中包含 vCard 文本的 Post 请求，格式化方式与上面的示例类似。

使用 Web API 分析器

2019/1/10 • • Edit Online

ASP.NET Core 2.2 和更高版本包含 Microsoft.AspNetCore.Mvc.Api.Analyzers NuGet 包，其中包含适用于 Web API 的分析器。分析器使用带有 ApiControllerAttribute 批注的控制器，同时构建 API 约定。

包安装

可以使用以下方法之一添加 `Microsoft.AspNetCore.Mvc.Api.Analyzers`：

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code
- .NET Core CLI

- 从"程序包管理器控制台"窗口：
  - 转到"视图" > "其他窗口" > "包管理器控制台"。
  - 导航到 ApiConventions.csproj 文件所在的目录。
  - 请执行以下命令：

    ```
    Install-Package Microsoft.AspNetCore.Mvc.Api.Analyzers
    ```

- 从"管理 NuGet 程序包"对话框中：
  - 右键单击"解决方案资源管理器" > "管理 NuGet 包"中的项目。
  - 将"包源"设置为"nuget.org"。
  - 在搜索框中输入"Microsoft.AspNetCore.Mvc.Api.Analyzers"。
  - 从"浏览"选项卡中选择"Microsoft.AspNetCore.Mvc.Api.Analyzers"包，然后单击"安装"。

API 约定的分析器

OpenAPI 文档包含操作可能返回的状态代码和响应类型。在 ASP.NET Core MVC 中，ProducesResponseTypeAttribute 和 ProducesAttribute 等属性用于记录操作。带有 Swagger/OpenAPI 的 ASP.NET Core Web API 帮助页 进一步介绍有关记录 API 的详细信息。

包中的其中一个分析器检查使用 ApiControllerAttribute 进行批注的控制器，并标识不完全记录其响应的操作。请看下面的示例：

```
// GET api/contacts/{guid}
[HttpGet("{id}", Name = "GetById")]
[ProducesResponseType(typeof(Contact), StatusCodes.Status200OK)]
public IActionResult Get(string id)
{
    var contact = _contacts.Get(id);

    if (contact == null)
    {
        return NotFound();
    }

    return Ok(contact);
}
```

上述操作记录了 HTTP 200 成功返回类型，但未记录 HTTP 404 失败状态代码。分析器将 HTTP 404 状态代码的缺失文档报告为警告。提供了修复此问题的选项。

其他资源

- 使用 Web API 约定
- 带有 Swagger/OpenAPI 的 ASP.NET Core Web API 帮助页
- 使用 ApiController 属性进行注释

使用 Web API 约定

2019/1/10 • • Edit Online

作者：Pranav Krishnamoorthy 和 Scott Addie

ASP.NET Core 2.2 及更高版本附带一种方法，可提取常见的 API 文档并将其应用于多个操作、控制器或某程序集内的所有控制器。Web API 约定可替代使用 [ProducesResponseType] 来装饰单个操作。

使用此约定，可以：

- 定义通过特定操作类型返回的、最常见的返回类型和状态代码。
- 识别偏离所定义的标准的操作。

ASP.NET Core MVC 2.2 及更高版本在 `Microsoft.AspNetCore.Mvc.DefaultApiConventions` 中包含一组默认的约定。约定基于 ASP.NET Core API 项目模板中提供的控件 (ValuesController.cs)。若操作遵循模板中的模式，则应成功使用默认约定。如果默认约定不能满足需要，请参阅创建 Web API 约定。

在运行时，Microsoft.AspNetCore.Mvc.ApiExplorer 会了解约定。`ApiExplorer` 是 MVC 与 OpenAPI（也称为 Swagger）文档生成器进行通信的抽象内容。已应用的约定中的属性与某个操作相关联，并包含在操作的 OpenAPI 文档中。API 分析器也会了解约定。若操作为非常规操作（例如，它返回已应用的约定未记录的状态代码），则会生成警告，建议记录该状态代码。

查看或下载示例代码（如何下载）

应用 Web API 约定

约定不是组合而成的，每个操作可能只与一个约定相关联。更明确的约定优先于不太明确的约定。当具有相同优先级的两个或更多约定应用于某个操作时，选择是不确定的。存在以下可将约定应用于操作的选项，明确性依次降低：

1. `Microsoft.AspNetCore.Mvc.ApiConventionMethodAttribute` — 适用于单独的操作，并指定适用的约定类型和约定方法。

   在以下示例中，默认约定类型的 `Microsoft.AspNetCore.Mvc.DefaultApiConventions.Put` 约定方法将应用于 `Update` 操作：

   ```
   // PUT api/contactsconvention/{guid}
   [HttpPut("{id}")]
   [ApiConventionMethod(typeof(DefaultApiConventions),
                        nameof(DefaultApiConventions.Put))]
   public IActionResult Update(string id, Contact contact)
   {
       var contactToUpdate = _contacts.Get(id);

       if (contactToUpdate == null)
       {
           return NotFound();
       }

       _contacts.Update(contact);

       return NoContent();
   }
   ```

   `Microsoft.AspNetCore.Mvc.DefaultApiConventions.Put` 约定方法可将以下属性应用于操作：

```
[ProducesDefaultResponseType]
[ProducesResponseType(204)]
[ProducesResponseType(404)]
[ProducesResponseType(400)]
```

2. `Microsoft.AspNetCore.Mvc.ApiConventionTypeAttribute` 应用于控制器 — 将指定约定类型应用于控制器上的所有操作。约定方法都带有提示，可确定要向其应用约定方法的操作。有关提示的详细信息，请参阅创建 Web API 约定）。

   在以下示例中，默认的约定集将应用于 ContactsConventionController 中的所有操作：

```
[ApiController]
[ApiConventionType(typeof(DefaultApiConventions))]
[Route("api/[controller]")]
public class ContactsConventionController : ControllerBase
{
```

3. `Microsoft.AspNetCore.Mvc.ApiConventionTypeAttribute` 应用于程序集 — 将指定约定类型应用于当前程序集中的所有控制器。建议将程序集级别的属性应用于 `Startup` 类。

   在以下示例中，默认的约定集将应用于程序集中的所有操作：

```
[assembly: ApiConventionType(typeof(DefaultApiConventions))]
namespace ApiConventions
{
    public class Startup
    {
```

创建 Web API 约定

如果默认 API 约定不能满足需要，请创建自己的约定。约定是：

- 带有方法的静态类型。
- 能够对操作定义响应类型和命名要求。

响应类型

这些方法使用 `[ProducesResponseType]` 或 `[ProducesDefaultResponseType]` 属性进行批注。例如：

```
public static class MyAppConventions
{
    [ProducesResponseType(200)]
    [ProducesResponseType(404)]
    public static void Find(int id)
    {
    }
}
```

如果没有更具体的元数据属性，则将此约定应用于程序集可强制实现以下内容：

- 该约定方法应用于所有名为 `Find` 的操作。
- `Find` 操作上存在名为 `id` 的参数。

命名要求

`[ApiConventionNameMatch]` 和 `[ApiConventionTypeMatch]` 属性可应用于约定方法，确定它们所要应用的操作。例如：

```
[ProducesResponseType(200)]
[ProducesResponseType(404)]
[ApiConventionNameMatch(ApiConventionNameMatchBehavior.Prefix)]
public static void Find(
    [ApiConventionNameMatch(ApiConventionNameMatchBehavior.Suffix)]
    int id)
{ }
```

在上面的示例中：

- 应用于该方法的 `Microsoft.AspNetCore.Mvc.ApiExplorer.ApiConventionNameMatchBehavior.Prefix` 选项表示该约定可匹配前缀是“Find”的任何操作。匹配的操作可以是 `Find`、`FindPet` 和 `FindById`。
- 应用于该参数的 `Microsoft.AspNetCore.Mvc.ApiExplorer.ApiConventionNameMatchBehavior.Suffix` 表示该约定可匹配带有唯一以标识符作为后缀结尾的参数的方法。示例包括 `id` 或 `petId` 等参数。与此类似，可将 `ApiConventionTypeMatch` 应用于类型，以约束参数类型。`params[]` 参数指示无需显式匹配的剩余参数。

其他资源

- 使用 Web API 分析器
- 带有 Swagger/OpenAPI 的 ASP.NET Core Web API 帮助页

ASP.NET Core SignalR 简介

2018/12/27 • • Edit Online

SignalR 是什么？

ASP.NET Core SignalR 是一个开源代码库，它简化了向应用添加实时 Web 功能的过程。实时 Web 功能使服务器端代码能够即时将内容推送到客户端。

SignalR 的适用对象：

- 需要来自服务器的高频率更新的应用。例如：游戏、社交网络、投票、拍卖、地图和 GPS 应用。
- 仪表板和监视应用。示例包括公司仪表板、销售状态即时更新或行程警示。
- 协作应用。协作应用的示例包括白板应用和团队会议软件。
- 需要通知的应用。社交网络、电子邮件、聊天、游戏、行程警示以及许多其他应用都使用通知。

SignalR 提供了一个用于创建服务器到客户端远程过程调用（RPC）的 API。RPC 通过服务器端 .NET Core 代码调用客户端上的 JavaScript 函数。

以下是 ASP.NET Core SignalR 的一些功能：

- 自动管理连接。
- 同时向所有连接的客户端发送消息。例如，聊天室。
- 将消息发送到特定的客户端或客户端组。
- 扩展以处理增加的流量。

源代码托管在 GitHub 上的 SignalR 存储库中。

传输

SignalR 支持几种方法用于处理实时通信：

- WebSockets
- 服务器发送事件
- 长轮询

SignalR 会从服务器和客户端支持的功能中自动选择最佳传输方法

中心

SignalR 使用*中心*在客户端和服务器之间进行通信。

“中心”是一种高级管道，允许客户端和服务器相互调用方法。SignalR 自动处理跨计算机边界的调度，允许客户端和服务器相互调用方法。可以将强类型参数传递给方法，从而启用模型绑定。SignalR 提供两个内置中心协议：基于 JSON 的文本协议和基于 MessagePack 的二进制协议。与 JSON 相比，MessagePack 创建的消息通常比较小。旧版浏览器必须支持 XHR 2 才能提供 MessagePack 协议支持。

中心通过发送包含客户端方法的名称和参数的消息来调用客户端代码。使用配置的协议对作为方法参数发送的对象进行反序列化。客户端会尝试将方法名称与客户端代码中的方法匹配。当客户端找到匹配项时，它会调用该方法并将反序列化的参数数据传递给它。

其他资源

- 开始使用 ASP.NET Core SignalR
- 支持的平台
- 中心
- JavaScript 客户端

ASP.NET Core SignalR 支持的平台

2018/12/6 • • Edit Online

服务器系统要求

ASP.NET Core SignalR 适用于 ASP.NET Core 支持的任何服务器平台。

JavaScript 客户端

JavaScript 客户端需要NodeJS 8或更高版本和以下浏览器上运行：

浏览者	版本
Microsoft Edge	当前
Mozilla Firefox	当前
Google Chrome;包括 Android	当前
Safari;包含 iOS	当前
Microsoft Internet Explorer	11

.NET 客户端

.NET 客户端可以在 ASP.NET Core 支持的任何平台上运行。例如，Xamarin 开发人员可以使用 SignalR用于构建 Android 应用程序使用 Xamarin.Android 8.4.0.1 或更高版本和 iOS 应用程序使用 Xamarin.iOS 11.14.0.4 或更高版本。

如果服务器运行 IIS，Websocket 传输要求安装 IIS 8.0 或更高版本 Windows Server 2012 或更高版本。其他传输在所有平台上都受支持。

Java 客户端

Java 客户端支持 Java 8 或更高版本。

不受支持的客户端

以下客户端可用，但是是实验性的或非官方。它们目前不支持，可能永远不会。

- C + + 客户端
- Swift 的客户端

教程：ASP.NET Core SignalR 入门

2019/1/10 • • Edit Online

本教程介绍生成使用 SignalR 的实时应用的基础知识。您将学习如何：

- 创建 Web 项目。
- 添加 SignalR 客户端库。
- 创建 SignalR 中心。
- 配置项目以使用 SignalR。
- 添加可将消息从任何客户端发送到所有连接客户端的代码。

最终将创建一个正常运行的聊天应用：

查看或下载示例代码（如何下载）。

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

创建 Web 项目

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 从菜单中选择“文件”>“新建项目”。
- 在“新建项目”对话框中，选择“已安装”>“Visual C#”>“Web”>“ASP.NET Core Web 应用”。将项目命名

为“SignalRChat”。

- 选择“Web 应用”，以创建使用 Razor Pages 的项目。
- 选择“.NET Core”目标框架，选择“ASP.NET Core 2.2”，然后单击“确定”。

添加 SignalR 客户端库

`Microsoft.AspNetCore.App` 元包中包括 SignalR 服务器库。JavaScript 客户端库不会自动包含在项目中。对于此教程，使用库管理器 (LibMan) 从 unpkg 获取客户端库。unpkg 是一个内容分发网络 (CDN)，可以分发在 npm（即

Node.js 包管理器）中找到的任何内容。

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 在“解决方案资源管理器”中，右键单击项目，然后选择“添加” > “客户端库”。
- 在“添加客户端库”对话框中，对于“提供程序”，选择“unpkg”。
- 对于“库”，输入 `@aspnet/signalr@1`，然后选择不是预览版的最新版本。

- 选择“选择特定文件”，展开“dist/browser”文件夹，然后选择“signalr.js”和“signalr.min.js”。
- 将“目标位置”设置为 wwwroot/lib/signalr/，然后选择“安装”。

LibMan 创建 wwwroot/lib/signalr 文件夹并将所选文件复制到该文件夹。

创建 SignalR 中心

*中心*是一个类，用作处理客户端 - 服务器通信的高级管道。

- 在 SignalRChat 项目文件夹中，创建 Hubs 文件夹。
- 在 Hubs 文件夹中，使用以下代码创建 ChatHub.cs 文件：

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

`ChatHub` 类继承自 SignalR `Hub` 类。`Hub` 类管理连接、组和消息。

可通过已连接客户端调用 `SendMessage`，以向所有客户端发送消息。本教程后面部分将显示调用该方法的 JavaScript 客户端代码。SignalR 代码是异步模式，可提供最大的可伸缩性。

配置 SignalR

必须配置 SignalR 服务器，以将 SignalR 请求传递到 SignalR。

- 将以下突出显示的代码添加到 Startup.cs 文件。

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using SignalRChat.Hubs;

namespace SignalRChat
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>(options =>
            {
                // This lambda determines whether user consent for non-essential cookies is needed for
a given request.
                options.CheckConsentNeeded = context => true;
                options.MinimumSameSitePolicy = SameSiteMode.None;
            });


            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

            services.AddSignalR();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            app.UseSignalR(routes =>
            {
                routes.MapHub<ChatHub>("/chatHub");
            });
            app.UseMvc();
        }
    }
}
```

这些更改将 SignalR 添加到 ASP.NET Core 依赖关系注入系统和中间件管道。

添加 SignalR 客户端代码

- 使用以下代码替换 Pages\Index.cshtml 中的内容：

```
@page
<div class="container">
    <div class="row"> </div>
    <div class="row">
        <div class="col-6"> </div>
        <div class="col-6">
            User..........<input type="text" id="userInput" />
            <br />
            Message...<input type="text" id="messageInput" />
            <input type="button" id="sendButton" value="Send Message" />
        </div>
    </div>
    <div class="row">
        <div class="col-12">
            <hr />
        </div>
    </div>
    <div class="row">
        <div class="col-6"> </div>
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
</div>
<script src="~/lib/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

前面的代码：

  - 创建名称以及消息文本的文本框和"提交"按钮。
  - 使用 `id="messagesList"` 创建一个列表，用于显示从 SignalR 中心接收的消息。
  - 包含对 SignalR 的脚本引用以及在下一步中创建的 chat.js 应用程序代码。
- 在 wwwroot/js 文件夹中，使用以下代码创建 chat.js 文件：

```
"use strict";

var connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();

connection.on("ReceiveMessage", function (user, message) {
    var msg = message.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;");
    var encodedMsg = user + " says " + msg;
    var li = document.createElement("li");
    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});

connection.start().catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function (event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

前面的代码：

- 创建并启动连接。
- 向“提交”按钮添加一个用于向中心发送消息的处理程序。
- 向连接对象添加一个用于从中心接收消息并将其添加到列表的处理程序。

运行应用

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 按 Ctrl+F5 可运行应用而不进行调试。
- 从地址栏复制 URL，打开另一个浏览器实例或选项卡，并在地址栏中粘贴该 URL。
- 选择任一浏览器，输入名称和消息，然后选择“发送消息”按钮。

  两个页面上立即显示名称和消息。

TIP

如果应用不起作用，请打开浏览器开发人员工具 (F12) 并转到控制台。可能会看到与 HTML 和 JavaScript 代码相关的错误。例如，假设将 signalr.js 放在不同于系统指示的文件夹中。在这种情况下，对该文件的引用将不起作用，并且你将在控制台中看到 404 错误。

后续步骤

在本教程中，你将了解：

- 创建 Web 应用项目。
- 添加 SignalR 客户端库。
- 创建 SignalR 中心。
- 配置项目以使用 SignalR。
- 添加代码，以便使用此中心将消息从任何客户端发送到所有连接的客户端。

若要详细了解 SignalR，请参阅简介：

ASP.NET Core SignalR 简介

配合使用 ASP.NET Core SignalR 和 TypeScript 以及 Webpack

2019/1/10 • • Edit Online

作者：Sébastien Sougnez 和 Scott Addie

开发人员可以通过 Webpack 捆绑和生成 Web 应用的客户端资源。本教程介绍在 ASP.NET Core SignalR Web 应用中使用 Webpack，该应用的客户端是使用 TypeScript 编写的。

在本教程中，你将了解：

- 为入门 ASP.NET Core SignalR 应用搭建基架
- 配置 SignalR TypeScript 客户端
- 使用 Webpack 配置生成管道
- 配置 SignalR 服务器
- 启用客户端和服务器之间的通信

查看或下载示例代码（如何下载）

Prerequisites

- Visual Studio
- Visual Studio Code

- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

创建 ASP.NET Core Web 应用

- Visual Studio
- Visual Studio Code

配置 Visual Studio，在 PATH 环境变量中查找 npm。默认情况下，Visual Studio 使用在安装目录中找到的 npm 版本。在 Visual Studio 中按照以下说明执行操作：

1. 导航到“工具” > “选项” > “项目和解决方案” > “Web 包管理” > “外部 Web 工具”。

2. 在列表中选择 $(PATH) 项。单击向上键将项移动列表第二个位置。

已完成 Visual Studio 配置。可以开始创建项目了。

1. 使用"文件" > "新建" > "项目"菜单选项，然后选择"ASP.NET Core Web 应用程序"模板。
2. 将项目命名为 SignalRWebPack 并选择"确定"。
3. 从目标框架下拉列表选择 .NET Core 并从框架选择器下拉列表选择 ASP.NET Core 2.2。选择"空白"模板并选择"确定"。

配置 Webpack 和 TypeScript

以下步骤配置 TypeScript 到 JavaScript 的转换和客户端资源的捆绑。

1. 在项目根目录中执行以下命令，创建 package.json 文件：

```
npm init -y
```

2. 将突出显示的属性添加到 package.json 文件：

```
{
  "name": "SignalRWebPack",
  "version": "1.0.0",
  "private": true,
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

   将 `private` 属性设置为 `true`，防止下一步出现包安装警告。

3. 安装所需的 npm 包。从项目根执行以下命令：

```
npm install -D -E clean-webpack-plugin@0.1.19 css-loader@0.28.11 html-webpack-plugin@3.2.0 mini-css-extract-plugin@0.4.0 ts-loader@4.4.1 typescript@2.9.2 webpack@4.12.0 webpack-cli@3.0.6
```

需要注意的一些命令细节：

- 每个包名称中 `@` 符号后是版本号。npm 安装这些特定的包版本。
- `-E` 选项禁用 npm 将语义化版本控制范围运算符写到 package.json 的默认行为。例如，使用 `"webpack": "4.12.0"` 而不是 `"webpack": "^4.12.0"`。此选项防止意外升级到新的包版本。

有关详细信息，请参阅官方 npm-install 文档。

4. 将 package.json 文件的 `scripts` 属性替换为以下代码片段：

```
"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

脚本的一些解释：

- `build`：在开发模式下捆绑客户端资源并观察文件更改。文件观察程序使捆绑在每次项目文件发生更改时重新生成。`mode` 选项可禁用生产优化，例如摇树优化和缩小优化。仅在开发中使用 `build`。
- `release`：在生产模式下捆绑客户端资源。
- `publish`：运行 `release` 脚本，在生产模式下捆绑客户端资源。它调用 .NET Core CLI 的 publish 命令发布应用。

5. 在项目根中创建名为 webpack.config.js 的文件，包含以下内容：

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const CleanWebpackPlugin = require("clean-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
    entry: "./src/index.ts",
    output: {
        path: path.resolve(__dirname, "wwwroot"),
        filename: "[name].[chunkhash].js",
        publicPath: "/"
    },
    resolve: {
        extensions: [".js", ".ts"]
    },
    module: {
        rules: [
            {
                test: /\.ts$/,
                use: "ts-loader"
            },
            {
                test: /\.css$/,
                use: [MiniCssExtractPlugin.loader, "css-loader"]
            }
        ]
    },
    plugins: [
        new CleanWebpackPlugin(["wwwroot/*"]),
        new HtmlWebpackPlugin({
            template: "./src/index.html"
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css"
        })
    ]
};
```

前面的文件配置 Webpack 编译。需要注意的一些配置细节：

- `output` 属性替代 dist 的默认值。捆绑反而在 wwwroot 目录中发出。
- `resolve.extensions` 数组包含 .js，以便导入 SignalR 客户端 JavaScript。

6. 在项目根中创建新的 src 目录。目的是存储项目的客户端资产。

7. 创建包含以下内容的 src/index.html。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>ASP.NET Core SignalR</title>
</head>
<body>
    <div id="divMessages" class="messages">
    </div>
    <div class="input-zone">
        <label id="lblMessage" for="tbMessage">Message:</label>
        <input id="tbMessage" class="input-zone-input" type="text" />
        <button id="btnSend">Send</button>
    </div>
</body>
</html>
```

前面的 HTML 定义主页的样板标记。

8. 创建新的 src/css 目录。目的是存储项目的 .css 文件。

9. 创建包含以下内容的 src/css/main.css：

```
*, *::before, *::after {
    box-sizing: border-box;
}

html, body {
    margin: 0;
    padding: 0;
}

.input-zone {
    align-items: center;
    display: flex;
    flex-direction: row;
    margin: 10px;
}

.input-zone-input {
    flex: 1;
    margin-right: 10px;
}

.message-author {
    font-weight: bold;
}

.messages {
    border: 1px solid #000;
    margin: 10px;
    max-height: 300px;
    min-height: 300px;
    overflow-y: auto;
    padding: 5px;
}
```

前面的 main.css 文件设计应用样式。

10. 创建包含以下内容的 src/tsconfig.json：

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

前面的代码配置 TypeScript 编译器，生成与 ECMAScript 5 兼容的 JavaScript。

11. 创建包含以下内容的 src/index.ts：

```
import "./css/main.css";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

前面的 TypeScript 检索对 DOM 元素的引用并附加两个事件处理程序：

- `keyup`：用户在文本框中键入标识为 `tbMessage` 的内容时触发此事件。用户按 Enter 时调用 `send` 函数。
- `click`：用户单击“发送”按钮时触发此事件。调用 `send` 函数。

配置 ASP.NET Core 应用

1. `Startup.Configure` 方法中提供的代码显示 Hello World!。将 `app.Run` 方法调用替换为对 UseDefaultFiles 和 UseStaticFiles 的调用。

```
app.UseDefaultFiles();
app.UseStaticFiles();
```

前面的代码允许服务器定位并提供 index.html 文件，无论用户输入完整 URL 还是 Web 应用的根 URL。

2. 在 `Startup.ConfigureServices` 方法中调用 AddSignalR。此操作会将 SignalR 服务添加到项目。

```
services.AddSignalR();
```

3. 将 /hub 路由映射到 `ChatHub` 中心。在 `Startup.Configure` 方法的末尾添加以下行：

```
app.UseSignalR(options =>
{
    options.MapHub<ChatHub>("/hub");
});
```

4. 在项目根中创建名为 Hubs 的新目录。目的是存储 SignalR 中心（在下一步中创建）。

5. 创建包含以下代码的中心 Hubs/ChatHub.cs：

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
    }
}
```

6. 在 Startup.cs 文件顶部添加以下代码，解析 `ChatHub` 引用：

```
using SignalRWebPack.Hubs;
```

启用客户端和服务器通信

应用当前显示一个发送消息的简单窗体。尝试执行此操作时没有任何反应。服务器正在侦听特定的路由，但是不涉及发送消息。

1. 在项目根执行以下命令：

```
npm install @aspnet/signalr
```

前面的命令安装 SignalR TypeScript 客户端，它允许客户端向服务器发送消息。

2. 将突出显示的代码添加到 src/index.ts 文件：

```
import "./css/main.css";
import * as signalR from "@aspnet/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.start().catch(err => document.write(err));

connection.on("messageReceived", (username: string, message: string) => {
    let m = document.createElement("div");

    m.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(m);
    divMessages.scrollTop = divMessages.scrollHeight;
});

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

前面的代码支持从服务器接收消息。`HubConnectionBuilder` 类创建新的生成器，用于配置服务器连接。`withUrl` 函数配置中心 URL。

SignalR 启用客户端和服务器之间的消息交换。每个消息都有特定的名称。例如，名为 `messageReceived` 的消息可以执行负责在消息区域显示新消息的逻辑。可以通过 `on` 函数完成对特定消息的侦听。可以侦听任意数量的消息名称。还可以将参数传递到消息，例如所接收消息的作者姓名和内容。客户端收到一条消息后，会创建一个新的 `div` 元素并在其 `innerHTML` 属性中显示作者姓名和消息内容。它添加到显示消息的主

要 `div` 元素。

3. 客户端可以接收消息后，将它配置为发送消息。将突出显示的代码添加到 src/index.ts 文件：

```
import "./css/main.css";
import * as signalR from "@aspnet/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.start().catch(err => document.write(err));

connection.on("messageReceived", (username: string, message: string) => {
    let messageContainer = document.createElement("div");

    messageContainer.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(messageContainer);
    divMessages.scrollTop = divMessages.scrollHeight;
});

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
    connection.send("newMessage", username, tbMessage.value)
              .then(() => tbMessage.value = "");
}
```

通过 WebSockets 连接发送消息需要调用 `send` 方法。该方法的第一个参数是消息名称。消息数据包含其他参数。在此示例中，一条标识为 `newMessage` 的消息已发送到服务器。该消息包含用户名和文本框中的用户输入。如果发送成功，会清空文本框。

4. 将突出显示的方法添加到 `ChatHub` 类：

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
        public async Task NewMessage(string username, string message)
        {
            await Clients.All.SendAsync("messageReceived", username, message);
        }
    }
}
```

服务器收到消息后，前面的代码会将这些消息播发到所有连接的用户。没有必要使用泛型 `on` 方法接收所有消息。使用以消息名称命名的方法就可以了。

在此示例中，TypeScript 客户端发送一条标识为 `newMessage` 的消息。C# `NewMessage` 方法需要客户端发送的数据。在 Clients.All 调用 SendAsync 方法。接收的消息会发送到所有连接到中心的客户端。

测试应用

确认应用遵循以下步骤。

- Visual Studio
- Visual Studio Code

1. 在 release 模式下运行 Webpack。使用"包管理器控制台"窗口，在项目根中运行以下命令。如果不在项目根中，请在输入该命令之前输入 `cd SignalRWebPack`。

   ```
   npm run release
   ```

   此命令在运行应用时生成要提供的客户端资产。资产位于 wwwroot 文件夹。

   Webpack 完成了以下任务：

   - 清除了 wwwroot 目录的内容。
   - 将 TypeScript 转换为 JavaScript，该过程称为"转译"。
   - 破坏生成的 JavaScript 以降低文件大小，该过程称为"缩小"。
   - 将已处理的 JavaScript、CSS 和 HTML 文件从 src 复制到 wwwroot 目录。
   - 将以下元素注入 wwwroot/index.html 文件：
     - 一个引用 wwwroot/main.<hash>.css 文件的 `<link>` 标记。此标记紧挨着 `</head>` 结束标记之前。
     - 一个引用缩小后的 wwwroot/main.<hash>.js 文件的 `<script>` 标记。此标记紧挨着 `</body>` 结束标记之前。
2. 选择"调试" > "开始执行(不调试)"，在不附加调试器的情况下在浏览器中启动应用。在 `http://localhost:<port_number>` 上提供 *wwwroot/index.html* 文件。

3. 打开另一个浏览器实例(任意浏览器)。在地址栏中粘贴 URL。

4. 选择一个浏览器，在"消息"文本框中键入任意内容，然后单击"发送"按钮。两个页面上立即显示唯一的用户名和消息。

其他资源

- ASP.NET Core SignalR JavaScript 客户端
- 在 ASP.NET Core SignalR 使用中心

ASP.NET Core 使用 SignalR 中的中心

2018/12/26 • • Edit Online

通过Rachel Appel和Kevin Griffin

查看或下载示例代码（如何下载）

SignalR 中心是什么

SignalR 中心 API，可从服务器在连接的客户端上调用方法。在服务器代码中，定义客户端调用的方法。在客户端代码中，定义从服务器调用的方法。SignalR 负责在后台实现实时的客户端到服务器和服务器到客户端通信的所有内容。

配置 SignalR 集线器

SignalR 中间件需要一些服务，这些服务通过调用 `services.AddSignalR` 来配置。

```
services.AddSignalR();
```

将 SignalR 功能添加到 ASP.NET Core 应用程序时，通过在 `Startup.Configure` 方法中调用 `app.UseSignalR` 来设置 SignalR 路由。

```
app.UseSignalR(route =>
{
    route.MapHub<ChatHub>("/chathub");
});
```

创建和使用中心

通过声明从 `Hub` 继承的类来创建中心，并向其添加公共方法。客户端可以调用定义为 `public` 的方法。

```
public class ChatHub : Hub
{
    public Task SendMessage(string user, string message)
    {
        return Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

您可以指定返回类型和参数，包括复杂类型和数组，就像在任何 C# 方法中。SignalR 处理序列化和反序列化复杂对象和数组中参数和返回值。

> **NOTE**
>
> 中心是暂时的：
>
> - 不要将状态存储在 hub 类的属性中。每个 hub 方法调用都在新的 hub 实例上执行。
> - 使用 `await` 调用取决于中心保持活动状态的异步方法时。例如，如方法 `Clients.All.SendAsync(...)` 如果调用，但不可能会失败 `await` 和集线器方法完成之前 `SendAsync` 完成。

上下文对象

`Hub` 类具有 `Context` 属性，其中包含以下属性与连接有关的信息：

属性	描述
`ConnectionId`	获取用于此连接，由 SignalR 分配的唯一 ID。没有为每个连接的一个连接 ID。
`UserIdentifier`	获取用户标识符。默认情况下，使用 SignalR `ClaimTypes.NameIdentifier` 从 `ClaimsPrincipal` 与作为用户标识符连接相关联。
`User`	获取 `ClaimsPrincipal` 与当前用户相关联。
`Items`	获取可用于共享此连接的作用域内的数据的键/值集合。数据可以存储在此集合中，它将连接在不同的集线器方法调用中保持原样。
`Features`	获取在连接上的可用功能的集合。现在，此集合不需要在大多数情况下，因此它不尚未记录在详细信息。
`ConnectionAborted`	获取 `CancellationToken`，时连接中止通知。

`Hub.Context` 此外包含以下方法：

方法	描述
`GetHttpContext`	返回 `HttpContext` 用于此连接，或 `null` 如果连接不是与 HTTP 请求相关联。对于 HTTP 连接，可以使用此方法以获取 HTTP 标头和查询字符串等信息。
`Abort`	中止的连接。

客户端对象

`Hub` 类具有 `Clients` 属性，其中包含服务器和客户端之间通信的以下属性：

属性	描述
`All`	在所有连接的客户端上调用的方法
`Caller`	在客户端调用集线器方法调用的方法
`Others`	除调用该方法的客户端的所有已连接客户端上调用的方法

`Hub.Clients` 此外包含以下方法：

方法	描述
`AllExcept`	除了指定连接的所有已连接客户端上调用的方法
`Client`	调用特定连接的客户端上的方法

方法	描述
`Clients`	调用特定连接的客户端上的方法
`Group`	指定的组中的所有连接上调用的方法
`GroupExcept`	在指定组中，指定的连接除外的所有连接上调用的方法
`Groups`	在多个组的连接上调用的方法
`OthersInGroup`	上一组连接，不包括客户端调用集线器方法调用的方法
`User`	与特定用户关联的所有连接上调用的方法
`Users`	使用指定的用户关联的所有连接上调用的方法

每个属性或方法上表中的返回一个包含对象 `SendAsync` 方法。 `SendAsync` 方法允许您提供的名称和要调用的客户端方法的参数。

将消息发送到客户端

若要使对特定的客户端的调用，使用的属性 `Clients` 对象。在以下示例中，有三个中心的方法：

- `SendMessage` 将消息发送到所有已连接客户端，使用 `Clients.All` 。
- `SendMessageToCaller` 将消息发送回调用方，使用 `Clients.Caller` 。
- `SendMessageToGroups` 将消息发送到中的所有客户端 `SignalR Users` 组。

```
public Task SendMessage(string user, string message)
{
    return Clients.All.SendAsync("ReceiveMessage", user, message);
}

public Task SendMessageToCaller(string message)
{
    return Clients.Caller.SendAsync("ReceiveMessage", message);
}

public Task SendMessageToGroup(string message)
{
    return Clients.Group("SignalR Users").SendAsync("ReceiveMessage", message);
}
```

强类型化的中心

使用的一个缺点 `SendAsync` 是依赖于使用魔幻字符串来指定客户端方法调用。这将使运行时错误，如果方法名称的拼写错误的代码打开或缺少从客户端。

使用的替代方法 `SendAsync` 是强类型化 `Hub` 与Hub<T>。在以下示例中， `ChatHub` 客户端方法具有出提取到一个接口，称为 `IChatClient` 。

```
public interface IChatClient
{
    Task ReceiveMessage(string user, string message);
    Task ReceiveMessage(string message);
}
```

此接口可用于重构前面 `ChatHub` 示例。

```
public class StronglyTypedChatHub : Hub<IChatClient>
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.ReceiveMessage(user, message);
    }

    public Task SendMessageToCaller(string message)
    {
        return Clients.Caller.ReceiveMessage(message);
    }
}
```

使用 `Hub<IChatClient>` 启用编译时检查的客户端方法。这可以防止由于使用魔幻字符串导致的问题 `Hub<T>` 可以仅提供访问权限在接口中定义的方法。

使用强类型化 `Hub<T>` 禁用的功能使用 `SendAsync` 。该接口上定义的任何方法仍可以定义为异步。实际上，每种方法应返回 `Task` 。由于它是一个接口，不要使用 `async` 关键字。例如：

```
public interface IClient
{
    Task ClientMethod();
}
```

> **NOTE**
>
> `Async` 后缀不从方法名称中去除。除非客户端方法使用定义 `.on('MyMethodAsync')` ，则不应使用 `MyMethodAsync` 作为名称。

将集线器方法的名称更改

默认情况下，服务器中心方法名称为.NET 方法的名称。但是，可以使用HubMethodName属性来更改此默认设置，并手动指定方法的名称。调用该方法时，客户端应而不是.NET 方法名称，使用此名称。

```
[HubMethodName("SendMessageToUser")]
public Task DirectMessage(string user, string message)
{
    return Clients.User(user).SendAsync("ReceiveMessage", message);
}
```

处理连接事件

SignalR 中心 API 提供了 `OnConnectedAsync` 和 `OnDisconnectedAsync` 管理和跟踪连接的虚拟方法。重写 `OnConnectedAsync` 虚拟方法，客户端连接到中心，如将其添加到组时，执行操作。

```
public override async Task OnConnectedAsync()
{
    await Groups.AddToGroupAsync(Context.ConnectionId, "SignalR Users");
    await base.OnConnectedAsync();
}
```

重写 `OnDisconnectedAsync` 虚拟方法，当客户端断开连接时执行的操作。如果客户端有意断开连接（通过调用 `connection.stop()`，例如），则 `exception` 参数将为 `null`。但是，如果客户端已断开连接错误（例如网络故障），由于 `exception` 参数将包含描述失败的异常。

```
public override async Task OnDisconnectedAsync(Exception exception)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, "SignalR Users");
    await base.OnDisconnectedAsync(exception);
}
```

处理错误

集线器方法中引发的异常将发送到的客户端的调用的方法。在 JavaScript 客户端 `invoke` 方法将返回JavaScript Promise。当客户端收到的错误处理程序附加到承诺使用 `catch`，它具有调用，并且作为 JavaScript 传递 `Error` 对象。

```
connection.invoke("SendMessage", user, message).catch(err => console.error(err));
```

如果你的中心引发了异常，不会关闭连接。默认情况下，SignalR 返回到客户端的一般性错误消息。例如：

```
Microsoft.AspNetCore.SignalR.HubException: An unexpected error occurred invoking 'MethodName' on the server.
```

意外的异常通常包含敏感信息，例如触发数据库连接失败时引发异常的数据库服务器的名称。SignalR 并不作为一种安全措施，默认情况下公开这些详细的错误消息。请参阅安全注意事项文章异常详细信息所抑制的原因的详细信息。

如果有异常条件您*做*想要传播到客户端，则可以使用 `HubException` 类。如果引发 `HubException` 从中心方法，SignalR将将整个消息发送到客户端，以未经修改的。

```
public Task ThrowException()
{
    throw new HubException("This error will be sent to the client!");
}
```

> **NOTE**
>
> SignalR 只发送 `Message` 到客户端异常的属性。堆栈跟踪和异常上的其他属性不是可用于客户端。

相关资源

- ASP.NET Core SignalR 简介
- JavaScript 客户端
- 发布到 Azure

从向外发送邮件中心

2018/11/16 • • Edit Online

通过Mikael Mengistu

SignalR 中心是用于将消息发送到客户端连接到 SignalR 服务器的核心抽象。还有可能将消息从你的应用使用中的其他位置发送 `IHubContext` 服务。此文章介绍了如何访问 SignalR `IHubContext` 将通知发送到外部的客户端从一个中心。

查看或下载示例代码（如何下载）

获取 IHubContext 的实例

在 ASP.NET Core SignalR，您可以访问的实例 `IHubContext` 通过依赖关系注入。您可以注入的实例 `IHubContext` 到控制器、中间件或其他 DI 服务。使用的实例将消息发送到客户端。

> **NOTE**
>
> 这不同于 ASP.NET 4.x SignalR GlobalHost 用于提供对访问 `IHubContext` 。ASP.NET Core具有的依赖关系注入框架，无需此全局单一实例。

注入控制器中的 IHubContext 实例

您可以注入的实例 `IHubContext` 插入控制器将其添加到您的构造函数：

```
public class HomeController : Controller
{
    private readonly IHubContext<NotificationHub> _hubContext;

    public HomeController(IHubContext<NotificationHub> hubContext)
    {
        _hubContext = hubContext;
    }
}
```

现在，有权访问的实例 `IHubContext` ，好像您在中心本身中，可以调用集线器方法。

```
public async Task<IActionResult> Index()
{
    await _hubContext.Clients.All.SendAsync("Notify", $"Home page loaded at: {DateTime.Now}");
    return View();
}
```

获取中间件中的 IHubContext 实例

访问 `IHubContext` 中间件管道中如下所示：

```
app.Use(async (context, next) =>
{
    var hubContext = context.RequestServices
                            .GetRequiredService<IHubContext<MyHub>>();
    //...
});
```

> **NOTE**
>
> 外部的从调用集线器方法时 `Hub` 类中，没有与调用相关联的任何调用方。因此，没有不能访问 `ConnectionId`， `Caller`，和 `Others` 属性。

注入强类型化 HubContext

若要注入强类型化 HubContext，请确保你的中心继承 `Hub<T>` 。将使用其注入 `IHubContext<THub, T>` 接口而非 `IHubContext<THub>` 。

```
public class ChatController : Controller
{
    public IHubContext<ChatHub, IChatClient> _strongChatHubContext { get; }

    public ChatController(IHubContext<ChatHub, IChatClient> chatHubContext)
    {
        _strongChatHubContext = chatHubContext;
    }

    public async Task SendMessage(string message)
    {
        await _strongChatHubContext.Clients.All.ReceiveMessage(message);
    }
}
```

相关资源

- 入门
- 中心
- 发布到 Azure

SignalR 中管理用户和组

2018/11/2 • • Edit Online

通过brennan，第 Conroy

SignalR 允许消息发送到与特定用户关联的所有连接，以及要命名的连接组。

查看或下载示例代码（如何下载）

SignalR 中的用户

SignalR，可将消息发送到与特定用户关联的所有连接。默认情况下，使用 SignalR `ClaimTypes.NameIdentifier` 从 `ClaimsPrincipal` 与作为用户标识符连接相关联。单个用户可以向 SignalR 应用程序的多个连接。例如，用户无法连接其桌面，以及他们的手机上。每个设备都有单独的 SignalR 连接，但它们具有相同的用户所有关联。如果向用户发送一条消息，所有与该用户关联的连接收到的消息。可以通过访问连接的用户标识符 `Context.UserIdentifier` 中心中的属性。

向特定用户发送消息，通过将传递到的用户标识符 `User` 函数集线器方法中，在下面的示例所示：

> **NOTE**
> 用户标识符是区分大小写。

```
public Task SendPrivateMessage(string user, string message)
{
    return Clients.User(user).SendAsync("ReceiveMessage", message);
}
```

可以通过创建自定义用户标识符 `IUserIdProvider`，并在注册 `ConfigureServices`。

```
public class CustomUserIdProvider : IUserIdProvider
{
    public virtual string GetUserId(HubConnectionContext connection)
    {
        return connection.User?.FindFirst(ClaimTypes.Email)?.Value;
    }
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR();

    services.AddSingleton<IUserIdProvider, CustomUserIdProvider>();
}
```

> **NOTE**
> 注册自定义 SignalR 服务之前，必须调用 AddSignalR。

SignalR 中的组

组是与名称相关联的连接的集合。可以将消息发送到组中的所有连接。组是发送到的连接或多个连接，因为组管理的应用程序的建议的方法。连接可以是多个组的成员。这使得组理想之选类似的聊天应用程序，其中每个房间可以表示为一个组。可以添加到连接或通过组中删除 `AddToGroupAsync` 和 `RemoveFromGroupAsync` 方法。

```
public async Task AddToGroup(string groupName)
{
    await Groups.AddToGroupAsync(Context.ConnectionId, groupName);

    await Clients.Group(groupName).SendAsync("Send", $"{Context.ConnectionId} has joined the group
{groupName}.");
}

public async Task RemoveFromGroup(string groupName)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, groupName);

    await Clients.Group(groupName).SendAsync("Send", $"{Context.ConnectionId} has left the group
{groupName}.");
}
```

当连接重新连接时，不会保留组成员身份。需要重新建立时重新加入组的连接。不能进行计数的一组成员，因为此信息不可用，如果应用程序扩展到多个服务器。

若要使用组的同时保护资源的访问权限，请使用身份验证和授权ASP.NET Core 中的功能。如果您仅将用户添加到组的凭据有效该组时，发送到该组的消息将仅转到经过授权的用户。但是，组不是一项安全功能。身份验证声明具有组却没有，如到期和吊销的功能。如果撤消用户的权限访问组，您必须手动检测，并从组中删除它们。

> **NOTE**
> 组名称是区分大小写。

相关资源

- 入门
- 中心
- 发布到 Azure

发布 ASP.NET Core SignalR 应用程序到 Azure Web 应用程序

2018/9/29 • • Edit Online

Azure Web 应用程序是Microsoft 云计算用于承载 web 应用，包括 ASP.NET Core 平台服务。

> **NOTE**
>
> 本文是指从 Visual Studio 发布 ASP.NET Core SignalR 应用程序。请访问Azure SignalR 服务有关如何在 Azure 上使用 SignalR 的详细信息。

发布应用

Visual Studio 提供的内置工具发布到 Azure Web 应用。可以使用 visual Studio Code 用户Azure CLI命令以将应用发布到 Azure。本文介绍如何发布使用 Visual Studio 中的工具。若要发布应用程序使用 Azure CLI，请参阅将 ASP.NET Core 应用发布到 Azure 中，使用命令行工具。

右键单击该项目中**解决方案资源管理器**，然后选择**发布**。确认**新建**签入选**取**发布目标对话框中，然后选择**发布**。

输入中的以下信息创建应用服务对话框并选择创建。

项	描述
应用名称	应用的唯一名称。
订阅	该应用使用 Azure 订阅。
资源组	应用程序所属的相关资源的组。
托管计划	用于 web 应用的定价计划。

Visual Studio 将完成以下任务：

- 创建发布配置文件包含发布设置。
- 创建或使用现有*的 Azure Web 应用*与提供的详细信息。
- 会将应用发布。
- 将浏览器中，启动已发布的 web 应用程序加载。

请注意 URL 的格式，应用程序 *{应用名称}.azurewebsites.net.net*。例如，名为的应用 `SignalRChattR` 具有如下所示的 URL `https://signalrchattr.azurewebsites.net`。

如果发生 HTTP 502.2 错误，请参阅到 Azure App Service 部署 ASP.NET Core 预览版来解决。

SignalR web 应用配置

ASP.NET Core SignalR 应用程序作为 Azure Web 应用必须发布ARR 地缘启用。Websocket应启用，以允许 Websocket 传输到该函数。

在 Azure 门户中，导航到应用设置为 web 应用。设置**Websocket**到上，并验证**ARR 相关性**是上。

Websocket 和其他传输都是有限的应用服务计划上基于。

相关资源

- 使用命令行工具将 ASP.NET Core 应用发布到 Azure
- 向具有 Visual Studio 的 Azure 发布 ASP.NET Core 应用
- 承载并将其部署在 Azure 上的 ASP.NET Core 预览应用

SignalR API 设计时的注意事项

2018/11/14 • • Edit Online

通过Andrew Stanton-nurse

本文提供了用于构建基于 SignalR 的 Api 的指南。

使用自定义对象参数来确保向后兼容性

将参数添加到 SignalR 集线器方法（在客户端或服务器）是*重大更改*。这意味着较旧的客户端/服务器会在尝试调用不带适当数量的参数的方法时出现错误。但是，将属性添加到自定义对象参数是**不**一项重大更改。这可以用于设计兼容的 Api 时可复原的客户端或服务器上的更改。

例如，考虑如下所示的服务器端 API:

```
public async Task<string> GetTotalLength(string param1)
{
    return param1.Length;
}
```

JavaScript 客户端调用此方法使用 `invoke`，如下所示：

```
connection.invoke("GetTotalLength", "value1");
```

如果以后服务器方法中添加第二个参数，较旧的客户端不会提供此参数值。例如：

```
public async Task<string> GetTotalLength(string param1, string param2)
{
    return param1.Length + param2.Length;
}
```

当旧客户端尝试调用此方法时，它会收到如下错误：

```
Microsoft.AspNetCore.SignalR.HubException: Failed to invoke 'GetTotalLength' due to an error on the server.
```

在服务器上，将看到一条日志消息如下：

```
System.IO.InvalidDataException: Invocation provides 1 argument(s) but target expects 2.
```

旧客户端仅发送一个参数，但较新版本的服务器 API 所需的两个参数。使用自定义对象作为参数提供了更大的灵活性。让我们重新设计要使用自定义对象的原始 API:

```
public class TotalLengthRequest
{
    public string Param1 { get; set; }
}

public async Task GetTotalLength(TotalLengthRequest req)
{
    return req.Param1.Length;
}
```

现在，客户端使用对象调用方法：

```
connection.invoke("GetTotalLength", { param1: "value1" });
```

无需添加参数，将属性添加到 `TotalLengthRequest` 对象：

```
public class TotalLengthRequest
{
    public string Param1 { get; set; }
    public string Param2 { get; set; }
}

public async Task GetTotalLength(TotalLengthRequest req)
{
    var length = req.Param1.Length;
    if (req.Param2 != null)
    {
        length += req.Param2.Length;
    }
    return length;
}
```

当旧客户端发送一个参数，额外 `Param2` 属性将保留 `null` 。你可以检测到通过检查由较旧的客户端发送的消息 `Param2` 为 `null` 并应用默认值。新的客户端可以发送两个参数。

```
connection.invoke("GetTotalLength", { param1: "value1", param2: "value2" });
```

相同的方法适用于在客户端上定义的方法。可以从服务器端发送自定义对象：

```
public async Task Broadcast(string message)
{
    await Clients.All.SendAsync("ReceiveMessage", new
    {
        Message = message
    });
}
```

在客户端上，访问 `Message` 属性，而无需使用参数：

```
connection.on("ReceiveMessage", (req) => {
    appendMessageToChatWindow(req.message);
});
```

如果你稍后决定要将消息的发件人添加到负载，请向对象添加属性：

```
public async Task Broadcast(string message)
{
    await Clients.All.SendAsync("ReceiveMessage", new
    {
        Sender = Context.User.Identity.Name,
        Message = message
    });
}
```

较旧的客户端不应为 `Sender` 值，因此将其忽略。新的客户端可以通过接受更新，以读取新属性：

```
connection.on("ReceiveMessage", (req) => {
    let message = req.message;
    if (req.sender) {
        message = req.sender + ": " + message;
    }
    appendMessageToChatWindow(message);
});
```

在这种情况下，新的客户端也是不提供的旧服务器的容错 `Sender` 值。由于旧的服务器不会提供 `Sender` 值时，客户端检查以查看是否存在之前对其进行访问。

ASP.NET Core SignalR.NET 客户端

2018/10/30 • • Edit Online

ASP.NET Core SignalR.NET 客户端库可让你与 SignalR 集线器从.NET 应用程序进行通信。

> **NOTE**
>
> Xamarin 有特殊要求 Visual Studio 版本。有关详细信息，请参阅在 Xamarin 中的 SignalR 客户端 2.1.1。

查看或下载示例代码（如何下载）

这篇文章中的代码示例是使用 ASP.NET Core SignalR.NET 客户端的 WPF 应用。

安装 SignalR.NET 客户端包

`Microsoft.AspNetCore.SignalR.Client` 包所需的.NET 客户端连接到 SignalR 集线器。若要安装客户端库，请运行以下命令**程序包管理器控制台**窗口：

```
Install-Package Microsoft.AspNetCore.SignalR.Client
```

连接到中心

若要建立连接，创建 `HubConnectionBuilder`，并调用 `Build`。生成一个连接时，可以配置中心 URL、协议、传输类型、日志级别、标头，和其他选项。配置任何必需的选项的方法是插入的任何 `HubConnectionBuilder` 方法转换 `Build`。启动与连接 `StartAsync`。

```
using System;
using System.Threading.Tasks;
using System.Windows;
using Microsoft.AspNetCore.SignalR.Client;

namespace SignalRChatClient
{
    public partial class MainWindow : Window
    {
        HubConnection connection;
        public MainWindow()
        {
            InitializeComponent();

            connection = new HubConnectionBuilder()
                .WithUrl("http://localhost:53353/ChatHub")
                .Build();

            connection.Closed += async (error) =>
            {
                await Task.Delay(new Random().Next(0,5) * 1000);
                await connection.StartAsync();
            };
        }

        private async void connectButton_Click(object sender, RoutedEventArgs e)
        {
            connection.On<string, string>("ReceiveMessage", (user, message) =>
            {
                this.Dispatcher.Invoke(() =>
                {
                   var newMessage = $"{user}: {message}";
                   messagesList.Items.Add(newMessage);
                });
            });

            try
            {
                await connection.StartAsync();
                messagesList.Items.Add("Connection started");
                connectButton.IsEnabled = false;
                sendButton.IsEnabled = true;
            }
            catch (Exception ex)
            {
                messagesList.Items.Add(ex.Message);
            }
        }

        private async void sendButton_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                await connection.InvokeAsync("SendMessage",
                    userTextBox.Text, messageTextBox.Text);
            }
            catch (Exception ex)
            {
                messagesList.Items.Add(ex.Message);
            }
        }
    }
}
```

已断开连接的句柄

使用Closed事件响应丢失连接。例如，你可能想要自动执行重新连接。

`Closed` 事件需要返回的委托 `Task`，它允许异步代码运行，而不使用 `async void` 。若要满足中的委托签名 `Closed` 运行以同步方式，返回的事件处理程序 `Task.CompletedTask`：

```
connection.Closed += (error) => {
    // Do your close logic.
    return Task.CompletedTask;
};
```

异步支持的主要原因是使您可以重新启动连接。正在启动的连接是一个异步操作。

在 `Closed` 处理程序重新启动该连接，请考虑等待某些随机延迟，以防止服务器，重载，如下面的示例中所示：

```
connection.Closed += async (error) =>
{
    await Task.Delay(new Random().Next(0,5) * 1000);
    await connection.StartAsync();
};
```

从客户端调用集线器方法

`InvokeAsync` 在该中心将调用方法。将中心方法名称和到中心方法中定义的任何参数传递 `InvokeAsync` 。SignalR 是异步的因此请使用 `async` 和 `await` 时进行调用。

```
await connection.InvokeAsync("SendMessage",
    userTextBox.Text, messageTextBox.Text);
```

从集线器调用客户端方法

定义使用集线器调用的方法 `connection.On` 之后生成，但之前启动连接。

```
connection.On<string, string>("ReceiveMessage", (user, message) =>
{
    this.Dispatcher.Invoke(() =>
    {
       var newMessage = $"{user}: {message}";
       messagesList.Items.Add(newMessage);
    });
});
```

在前面的代码 `connection.On` 运行时服务器端代码将调用它使用 `SendAsync` 方法。

```
public async Task SendMessage(string user, string message)
{
    await Clients.All.SendAsync("ReceiveMessage", user,message);
}
```

错误处理和日志记录

处理 try catch 语句的错误。检查 `Exception` 对象，以确定要执行后发生错误的适当操作。

```
try
{
    await connection.InvokeAsync("SendMessage",
        userTextBox.Text, messageTextBox.Text);
}
catch (Exception ex)
{
    messagesList.Items.Add(ex.Message);
}
```

其他资源

- 中心
- JavaScript 客户端
- 发布到 Azure

ASP.NET Core SignalR Java 客户端

2018/12/5 • • Edit Online

通过Mikael Mengistu

Java 客户端，包括 Android 应用程序的 Java 代码中连接到 ASP.NET Core SignalR 服务器。像JavaScript 客户端并.NET 客户端，Java 客户端，可接收并将消息发送到实时的中心。Java 客户端是可在 ASP.NET Core 2.2 及更高版本。

在这篇文章中引用示例 Java 控制台应用程序使用 SignalR Java 客户端。

查看或下载示例代码（如何下载）

SignalR Java 客户端程序包安装

Signalr 1.0.0 JAR 文件允许客户端连接到 SignalR 集线器。若要查找最新的 JAR 文件版本号，请参阅Maven 搜索结果。

如果使用 Gradle，添加下面的代码行 `dependencies` 一部分您*build.gradle*文件：

```
implementation 'com.microsoft.signalr:signalr:1.0.0'
```

如果使用 Maven，添加以下行 `<dependencies>` 的元素在*pom.xml*文件：

```
<dependency>
    <groupId>com.microsoft.signalr</groupId>
    <artifactId>signalr</artifactId>
    <version>1.0.0</version>
</dependency>
```

连接到中心

若要建立 `HubConnection`，则 `HubConnectionBuilder` 应使用。生成连接时，可以配置中心 URL 和日志级别。配置任何必需的选项，方法是调用的任何 `HubConnectionBuilder` 方法之前 `build`。启动与连接 `start`。

```
HubConnection hubConnection = HubConnectionBuilder.create(input)
        .build();
```

从客户端调用集线器方法

调用 `send` 调用集线器方法。将中心方法名称和到中心方法中定义的任何参数传递 `send`。

```
hubConnection.send("Send", input);
```

从集线器调用客户端方法

使用 `hubConnection.on` 可以调用该集线器的客户端上定义的方法。构建之后启动连接之前定义的方法。

```
hubConnection.on("Send", (message) -> {
    System.out.println("New Message: " + message);
}, String.class);
```

添加日志记录

SignalR Java 客户端使用SLF4J用于日志记录库。它是库的一个高级日志记录 API，允许通过将特定的日志记录依赖项中选择其自己特定的日志记录实现的用户。下面的代码段演示如何使用 `java.util.logging` 的 SignalR Java 客户端。

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

如果没有配置依赖项中的日志记录，SLF4J 加载具有以下警告消息的默认无操作记录器：

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

这可以安全地忽略。

Android 开发说明

与 SignalR 客户端功能的 Android SDK 兼容性时，请考虑以下各项指定你的目标 Android SDK 版本：

- SignalR Java 客户端将运行 Android API 级别 16 和更高版本。
- 通过 Azure SignalR 服务连接将需要 20 及更高版本的 Android API 级别因为Azure SignalR 服务需要 TLS 1.2，并且不支持基于 SHA-1 的密码套件。Android添加支持 SHA-256（和更高版本）的密码套件API 20 级中。

配置持有者令牌身份验证

在 SignalR Java 客户端，可配置的持有者令牌，用于通过提供"访问令牌工厂"进行身份验证，向HttpHubConnectionBuilder。使用withAccessTokenFactory提供RxJava 单一。通过调用Single.defer，可以编写逻辑来为您的客户端生成访问令牌。

```
HubConnection hubConnection = HubConnectionBuilder.create("YOUR HUB URL HERE")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
        return Single.just("An Access Token");
    })).build();
```

已知限制

- 支持仅 JSON 协议。
- 支持仅 Websocket 传输。
- 流式处理尚不支持。

其他资源

- Java API 参考
- 在 ASP.NET Core SignalR 使用中心
- ASP.NET Core SignalR JavaScript 客户端

- 发布 ASP.NET Core SignalR 应用程序到 Azure Web 应用

ASP.NET Core SignalR JavaScript 客户端

2018/12/7 • • Edit Online

作者：Rachel Appel

ASP.NET Core SignalR JavaScript 客户端库，开发人员可以调用服务器端中心代码。

查看或下载示例代码（如何下载）

安装 SignalR 客户端包

作为提供 SignalR JavaScript 客户端库npm包。如果使用 Visual Studio，运行 `npm install` 从程序包管理器控制台时的根文件夹中。对于 Visual Studio Code 中，运行中的命令集成终端。

```
npm init -y
npm install @aspnet/signalr
```

npm 安装中的包内容 *node_modules\@aspnet\signalr\dist\browser* 文件夹。创建一个名为的新文件夹*signalr*下*wwwroot\lib*文件夹。复制 *signalr.js* 的文件 *wwwroot\lib\signalr* 文件夹。

使用 SignalR JavaScript 客户端

引用中的 SignalR JavaScript 客户端 `<script>` 元素。

```
<script src="~/lib/signalr/signalr.js"></script>
```

连接到中心

下面的代码创建并启动连接。在中心的名称是不区分大小写。

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

跨域的连接

通常情况下，浏览器请求的页面所在的域从加载的连接。但是，有一些情况下需要与另一个域的连接时。

若要防止恶意站点读取另一个站点中的敏感数据跨域连接默认处于禁用状态。若要允许跨域请求，请启用它在 `Startup` 类。

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using SignalRChat.Hubs;

namespace SignalRChat
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>(options =>
            {
                options.CheckConsentNeeded = context => true;
                options.MinimumSameSitePolicy = SameSiteMode.None;
            });

            services.AddMvc();

            services.AddCors(options => options.AddPolicy("CorsPolicy",
            builder =>
            {
                builder.AllowAnyMethod().AllowAnyHeader()
                       .WithOrigins("http://localhost:55830")
                       .AllowCredentials();
            }));

            services.AddSignalR();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseBrowserLink();
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            app.UseCors("CorsPolicy");
            app.UseSignalR(routes =>
            {
                routes.MapHub<ChatHub>("/chathub");
            });
            app.UseMvc();
        }
    }
}
```

从客户端调用集线器方法

JavaScript 客户端上中心通过调用公共方法调用方法HubConnection。`invoke`方法接受两个参数：

- 集线器方法的名称。在以下示例中，中心上的方法名称是`SendMessage`。
- 在集线器方法中定义的任何参数。在以下示例中，参数名称是`message`。

```
connection.invoke("SendMessage", user, message).catch(err => console.error(err.toString()));
```

从集线器调用客户端方法

若要从中心接收消息，定义方法使用上方法的`HubConnection`。

- JavaScript 客户端方法的名称。在以下示例中，方法名称是`ReceiveMessage`。
- 该中心将传递给方法的参数。在以下示例中，参数值是`message`。

```
connection.on("ReceiveMessage", (user, message) => {
    const encodedMsg = user + " says " + message;
    const li = document.createElement("li");
    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});
```

在前面的代码`connection.on`运行时服务器端代码将调用它使用SendAsync方法。

```
public async Task SendMessage(string user, string message)
{
    await Clients.All.SendAsync("ReceiveMessage", user, message);
}
```

SignalR 确定要进行匹配的方法名称来调用的客户端方法和参数中定义`SendAsync`和`connection.on`。

> **NOTE**
> 最佳做法是，调用启动方法`HubConnection`后`on`。这样做可确保您的处理程序注册之前接收任何消息。

错误处理和日志记录

链`catch`方法的末尾`start`方法以处理客户端错误。使用`console.error`向浏览器的控制台输出错误。

```
connection.start().catch(function (err) {
    return console.error(err.toString());
});
```

通过传递要进行连接时记录的记录器和事件类型的安装程序客户端的日志跟踪。使用指定的日志级别和更高版本记录的消息。可用日志级别为按如下所示：

- `signalR.LogLevel.Error` – 错误消息。日志`Error`仅消息。
- `signalR.LogLevel.Warning` – 有关潜在错误的警告消息。日志`Warning`，和`Error`消息。
- `signalR.LogLevel.Information` – 不包含错误的状态消息。日志`Information`，`Warning`，和`Error`消息。
- `signalR.LogLevel.Trace` – 跟踪消息。记录所有内容，包括数据中心和客户端之间传输。

使用configureLogging方法HubConnectionBuilder若要配置日志级别。消息会记录到浏览器控制台。

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

重新连接客户端

SignalR JavaScript 客户端不会自动重新连接。必须编写代码将手动重新连接你的客户端。下面的代码演示了典型的重新连接方法：

1. 一个函数（在这种情况下，`start`函数）创建以启动连接。
2. 调用`start`中的连接函数`onclose`事件处理程序。

```
async function start() {
    try {
        await connection.start();
        console.log('connected');
    } catch (err) {
        console.log(err);
        setTimeout(() => start(), 5000);
    }
};

connection.onclose(async () => {
    await start();
});
```

实际的实现会使用指数退让或重试的次数后放弃了指定的次数。

其他资源

- JavaScript API 参考
- JavaScript 教程
- WebPack 和 TypeScript 教程
- 中心
- .NET 客户端
- 发布到 Azure
- 跨域请求 (CORS)

ASP.NET Core SignalR 承载和扩展

2018/11/28 • • Edit Online

通过Andrew Stanton-nurse，Brady Gaster，并Tom Dykstra，

本文介绍承载和扩展使用 ASP.NET Core SignalR 的高流量应用的注意事项。

TCP 连接资源

Web 服务器可以支持的并发 TCP 连接数会受到限制。使用标准 HTTP 客户端*临时*连接。当客户端进入空闲状态，并在以后再次打开时，可以关闭这些连接。但是，SignalR 连接是*持久*。甚至当客户端进入空闲状态时，SignalR 连接保持打开状态。在高流量应用中提供多个客户端，这些持久连接可能会导致服务器达到其最大连接数。

持久连接还使用一些额外的内存，则还跟踪每个连接。

大量使用 SignalR 通过与连接相关的资源可能会影响托管在同一台服务器的其他 web 应用。当 SignalR 打开和保存最后一个可用的 TCP 连接时，在同一服务器上的其他 web 应用还可提供给他们没有更多连接。

如果服务器的连接，会看到随机套接字错误和连接重置错误。例如：

```
An attempt was made to access a socket in a way forbidden by its access permissions...
```

若要防止在其他 web 应用中导致错误 SignalR 资源使用情况，比在其他 web 应用程序的不同服务器上运行 SignalR。

若要防止 SignalR 资源使用情况导致出错的 SignalR 应用程序，横向扩展以限制服务器必须处理的连接数。

横向扩展

使用 SignalR 的应用程序需要跟踪的所有连接，这将创建服务器场的问题。添加服务器，并获取其他服务器不了解的内容的新连接。例如，以下关系图中每个服务器上的 SignalR 是不知道的其他服务器上的连接数。当在一台服务器上的 SignalR 想要将消息发送到所有客户端时，消息将只发送到连接到该服务器的客户端。

解决此问题的选项都Azure SignalR 服务并Redis 底板。

Azure SignalR 服务

Azure SignalR 服务是一个代理而不是基架。客户端启动连接到服务器，每次客户端将重定向连接到服务。下图说明了该过程：

结果是，该服务管理的所有客户端连接，而每台服务器需要仅小的一定数量的连接到服务，如以下关系图中所示：

若要横向扩展此方法具有几大优势，Redis 底板替代方法：

- 粘性会话，也称为客户端相关性，不是必需的因为客户端将立即重定向到 Azure SignalR 服务在连接时。
- 应用程序可以向外扩展 SignalR 基于 Azure SignalR 服务可以自动缩放以处理任意数量的连接时发送的消息数。例如，可能有数千个客户端，但如果只发送少量消息数 / 秒，SignalR 应用程序不需要向外扩展到多个服务器只是为了处理连接本身。
- SignalR 应用程序不会使用更多的连接资源，而无需 SignalR 的 web 应用比。

出于这些原因，我们建议 Azure SignalR 服务托管在 Azure 中，包括应用服务、Vm 和容器上的所有 ASP.NET Core SignalR 应用程序。

有关详细信息请参阅Azure SignalR 服务文档。

Redis 底板

Redis是支持发布/订阅模型的消息传送系统的内存中键 / 值存储。Redis 的 SignalR 基架使用发布/订阅功能以将消息转发到其他服务器。当客户端进行连接时，连接信息传递给基架。当服务器想要将消息发送到所有客户端时，它将发送到底板。底板知道所有连接的客户端和服务器它们是在。它将消息发送给所有客户端通过其各自的服务器。此过程是在下图中所示：

Redis 底板是用于在自己的基础结构上托管的应用建议的向外缩放方法。Azure SignalR 服务并不可行的本地应用，原因是你的数据中心和 Azure 数据中心之间的连接延迟用于生产。

前面记下 Azure SignalR 服务的优点是 Redis 底板的缺点：

- 粘性会话，也称为客户端相关性，是必需的。在服务器上启动的连接，连接后继续使用该服务器。
- SignalR 应用程序必须横向扩展根据的客户端数，即使正发送的几个消息。
- SignalR 应用程序使用的更多 web 应用而无需 SignalR 连接资源。

后续步骤

有关更多信息，请参见以下资源：

- Azure SignalR 服务文档
- 设置 Redis 底板

设置 ASP.NET Core SignalR 横向扩展 Redis 底板

2019/1/8 • • Edit Online

通过Andrew Stanton-nurse，Brady Gaster，并Tom Dykstra，

此文章介绍了设置的特定于 SignalR 的方面Redis要用于横向扩展 ASP.NET Core SignalR 应用程序服务器。

设置 Redis 底板

- 部署 Redis 服务器。

  > **IMPORTANT**
  >
  > 对于生产用途，Redis 底板时，建议仅在与 SignalR 应用位于同一数据中心中运行。否则，网络延迟会降低性能。如果您的 SignalR 应用程序运行在 Azure 云中，我们将建议 Azure SignalR 服务而不是 Redis 底板。可以使用 Azure Redis 缓存服务进行开发和测试环境。

  有关更多信息，请参见以下资源：

  - ASP.NET Core SignalR 生产承载和扩展
  - Redis 文档
  - Azure Redis 缓存文档

- 在 SignalR 应用程序中，将安装 `Microsoft.AspNetCore.SignalR.Redis` NuGet 包。（此外，还有 `Microsoft.AspNetCore.SignalR.StackExchangeRedis` 包，但该之一是 ASP.NET Core 2.2 及更高版本。）

- 在中 `Startup.ConfigureServices` 方法中，调用 `AddRedis` 后 `AddSignalR`：

  ```
  services.AddSignalR().AddRedis("<your_Redis_connection_string>");
  ```

- 根据需要配置选项：

  可以设置大多数选项，在连接字符串中或在ConfigurationOptions对象。中指定的选项 `ConfigurationOptions` 在连接字符串中设置的替代。

  下面的示例演示如何在中设置选项 `ConfigurationOptions` 对象。此示例添加通道前缀，以便多个应用程序可以共享同一 Redis 实例下，一步中所述。

  ```
  services.AddSignalR()
    .AddRedis(connectionString, options => {
        options.Configuration.ChannelPrefix = "MyApp";
    });
  ```

  在前面的代码，`options.Configuration` 使用连接字符串中指定了任何初始化。

- 在 SignalR 应用程序中，将安装以下 NuGet 包之一：

  - `Microsoft.AspNetCore.SignalR.StackExchangeRedis` -依赖于 StackExchange.Redis 2.X.X。这是建议的包的 ASP.NET Core 2.2 及更高版本。
  - `Microsoft.AspNetCore.SignalR.Redis` -依赖于 StackExchange.Redis 1.X.X。此包不会在 ASP.NET Core 3.0 发布。

- 在中 `Startup.ConfigureServices` 方法中，调用 `AddStackExchangeRedis` 后 `AddSignalR`：

  ```
  services.AddSignalR().AddStackExchangeRedis("<your_Redis_connection_string>");
  ```

- 根据需要配置选项：

  可以设置大多数选项，在连接字符串中或在ConfigurationOptions对象。中指定的选项 `ConfigurationOptions` 在连接字符串中设置的替代。

  下面的示例演示如何在中设置选项 `ConfigurationOptions` 对象。此示例添加通道前缀，以便多个应用程序可以共享同一 Redis 实例下，一步中所述。

  ```
  services.AddSignalR()
    .AddStackExchangeRedis(connectionString, options => {
        options.Configuration.ChannelPrefix = "MyApp";
    });
  ```

  在前面的代码，`options.Configuration` 使用连接字符串中指定了任何初始化。

  有关 Redis 选项的信息，请参阅StackExchange Redis 文档。

- 如果您正在使用多个 SignalR 应用一个 Redis 服务器，为每个 SignalR 应用使用不同的通道前缀。

  设置通道前缀将从其他人使用不同的通道前缀的一个 SignalR 应用程序隔离开来。如果没有分配不同的前缀，从一个应用到所有其自己的客户端发送的消息将转到用作基架使用 Redis 服务器的所有应用的所有客户端。

- 配置服务器场负载平衡软件为粘性会话。下面是文档的有关如何执行该操作的一些示例：

  - IIS
  - HAProxy
  - Nginx
  - pfSense

Redis 服务器错误

当 Redis 服务器出现故障时，SignalR 会引发异常，指示不会将消息传递。一些典型异常消息：

- *失败的编写消息*
- *无法调用集线器方法 MethodName*
- *未能连接到 Redis*

SignalR 不缓冲消息，当服务器重新启动后发送它们。Redis 服务器已关闭时发送任何消息都将丢失。

当 Redis 服务器再次可用时，自动重新连接 SignalR。

连接失败的自定义行为

下面是一个示例，演示如何处理 Redis 连接失败事件。

```
services.AddSignalR()
        .AddRedis(o =>
        {
            o.ConnectionFactory = async writer =>
            {
                var config = new ConfigurationOptions
                {
                    AbortOnConnectFail = false
                };
                config.EndPoints.Add(IPAddress.Loopback, 0);
                config.SetDefaultPorts();
                var connection = await ConnectionMultiplexer.ConnectAsync(config, writer);
                connection.ConnectionFailed += (_, e) =>
                {
                    Console.WriteLine("Connection to Redis failed.");
                };

                if (!connection.IsConnected)
                {
                    Console.WriteLine("Did not connect to Redis.");
                }

                return connection;
            };
        });
```

```
services.AddSignalR()
        .AddMessagePackProtocol()
        .AddStackExchangeRedis(o =>
        {
            o.ConnectionFactory = async writer =>
            {
                var config = new ConfigurationOptions
                {
                    AbortOnConnectFail = false
                };
                config.EndPoints.Add(IPAddress.Loopback, 0);
                config.SetDefaultPorts();
                var connection = await ConnectionMultiplexer.ConnectAsync(config, writer);
                connection.ConnectionFailed += (_, e) =>
                {
                    Console.WriteLine("Connection to Redis failed.");
                };

                if (!connection.IsConnected)
                {
                    Console.WriteLine("Did not connect to Redis.");
                }

                return connection;
            };
        });
```

聚类分析

聚类分析是一种方法使用多个 Redis 服务器实现高可用性。聚类分析不正式支持，但它可能起作用。

后续步骤

有关更多信息，请参见以下资源：

- ASP.NET Core SignalR 生产承载和扩展

- Redis 文档
- StackExchange Redis 文档
- Azure Redis 缓存文档

ASP.NET Core SignalR 配置

2018/10/18 • • Edit Online

JSON/MessagePack 序列化选项

ASP.NET Core SignalR 支持两个协议为消息编码：JSON和MessagePack。每个协议具有序列化配置选项。

JSON 序列化可以配置服务器使用AddJsonProtocol扩展方法，可以添加后AddSignalR在你 `Startup.ConfigureServices` 方法。 `AddJsonProtocol` 方法采用一个委托，接收 `options` 对象。PayloadSerializerSettings对该对象的属性是 JSON.NET `JsonSerializerSettings` 可用于配置序列化的参数和返回值的对象。请参阅JSON.NET 文档的更多详细信息。

例如，若要配置的序列化程序，而不是默认的"驼峰式大小写"名称，使用"pascal 命名法"属性名称，使用以下代码：

```
services.AddSignalR()
    .AddJsonProtocol(options => {
        options.PayloadSerializerSettings.ContractResolver =
        new DefaultContractResolver();
    });
```

在.NET 客户端，相同 `AddJsonProtocol` 上存在扩展方法HubConnectionBuilder。 `Microsoft.Extensions.DependencyInjection` 必须导入命名空间，若要解决的扩展方法：

```
// At the top of the file:
using Microsoft.Extensions.DependencyInjection;

// When constructing your connection:
var connection = new HubConnectionBuilder()
    .AddJsonProtocol(options => {
        options.PayloadSerializerSettings.ContractResolver =
            new DefaultContractResolver();
    })
    .Build();
```

> **NOTE**
>
> 不能在这一次配置 JavaScript 客户端中的 JSON 序列化。

MessagePack 序列化选项

可以通过提供委托，配置 MessagePack 序列化AddMessagePackProtocol调用。请参阅SignalR 中的MessagePack的更多详细信息。

> **NOTE**
>
> 不能在此时间在 JavaScript 客户端中配置 MessagePack 序列化。

配置服务器选项

下表描述了用于配置 SignalR 集线器的选项：

选项	默认值	描述
`HandshakeTimeout`	15 秒	如果客户端不会在此时间间隔内发送初始握手消息，该连接已关闭。这是一种高级的设置，如果由于出现严重的网络延迟发生握手超时错误应仅修改。握手过程的更多详细信息，请参阅SignalR 集线器协议规范。
`KeepAliveInterval`	15 秒	如果服务器尚未在此时间间隔内发送一条消息，是自动发送一条 ping 消息使连接保持打开状态。更改时 `KeepAliveInterval`，更改 `ServerTimeout` / `serverTimeoutInMilliseconds` 设置在客户端上。推荐 `ServerTimeout` / `serverTimeoutInMilliseconds` 值是双精度 `KeepAliveInterval` 值。
`SupportedProtocols`	所有已安装的协议	此中心支持的协议。默认情况下，允许在服务器上注册的所有协议，但可以从禁用特定协议的单个中心此列表中删除协议。
`EnableDetailedErrors`	`false`	如果 `true`、详细异常消息返回到客户端集线器方法中引发异常。默认值是 `false`，因为这些异常消息可能包含敏感信息。

可以通过提供一个选项委托到的所有中心配置选项 `AddSignalR` 调用中 `Startup.ConfigureServices`。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(hubOptions =>
    {
        hubOptions.EnableDetailedErrors = true;
        hubOptions.KeepAliveInterval = TimeSpan.FromMinutes(1);
    });
}
```

有关单个集线器的选项重写中提供的全局选项 `AddSignalR`，可以使用配置AddHubOptions<T >:

```
services.AddSignalR().AddHubOptions<MyHub>(options =>
{
    options.EnableDetailedErrors = true;
});
```

使用 `HttpConnectionDispatcherOptions` 配置与传输和内存缓冲区管理相关的高级的设置。通过将传递委托，配置这些选项MapHub<T >。

选项	默认值	描述
`ApplicationMaxBufferSize`	32 KB	从客户端接收的字节数最大数量的服务器缓冲区。增加此值允许服务器以接收更大的消息，但会降低内存占用情况。

选项	默认值	描述
`AuthorizationData`	从自动收集数据 `Authorize` 应用于集线器类的特性。	一系列IAuthorizeData对象用于确定客户端有权连接到中心。
`TransportMaxBufferSize`	32 KB	最大的应用发送的字节数的服务器缓冲区。增加此值允许服务器以发送较大的消息，但会降低内存占用情况。
`Transports`	启用所有传输。	一个位掩码的 `HttpTransportType` 值可用于限制传输客户端可用来连接。
`LongPolling`	请参阅下文。	特定于长轮询传输的其他选项。
`WebSockets`	请参阅下文。	其他选项特定于 Websocket 传输。

长轮询传输中包含其他选项，可以使用配置 `LongPolling` 属性：

选项	默认值	描述
`PollTimeout`	90 秒	服务器等待要终止单个轮询请求之前发送到客户端的消息的最大时间量。减小此值会导致客户端更频繁地发出新轮询请求。

WebSocket 传输中包含其他选项，可以使用配置 `WebSockets` 属性：

选项	默认值	描述
`CloseTimeout`	5 秒	在服务器关闭，如果客户端无法在此时间间隔内关闭后，连接将终止。
`SubProtocolSelector`	`null`	一个委托，它可以用来设置 `Sec-WebSocket-Protocol` 为自定义值的标头。该委托接收作为输入客户端请求的值，并应返回所需的值。

配置客户端选项

可以在配置客户端选项 `HubConnectionBuilder` 类型（适用于.NET 和 JavaScript 客户端），以及 `HubConnection` 本身。

配置日志记录

使用.NET 客户端中配置日志记录 `ConfigureLogging` 方法。可以在相同的方式注册日志记录提供程序和筛选器，因为它们是在服务器上。请参阅登录 ASP.NET Core 文档以了解更多信息。

> **NOTE**
>
> 若要注册日志记录提供程序，必须安装所需的包。请参阅内置日志记录提供程序部分有关的完整列表。

例如，若要启用控制台日志记录，安装 `Microsoft.Extensions.Logging.Console` NuGet 包。调用 `AddConsole` 扩展方法：

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/myhub")
    .ConfigureLogging(logging => {
        logging.SetMinimumLevel(LogLevel.Information);
        logging.AddConsole();
    })
    .Build();
```

在 JavaScript 客户端，类似 `configureLogging` 方法存在。提供 `LogLevel` 值，该值指示要生成的日志消息的最小级别。日志将写入浏览器控制台窗口。

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/myhub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

> **NOTE**
> 若要禁用完全日志记录，请指定 `signalR.LogLevel.None` 在 `configureLogging` 方法。

下面列出了可用于 JavaScript 客户端的日志级别。将日志级别设置为下列值之一，则记录上的消息或更高版本该级别。

级别	描述
`None`	未不记录任何消息。
`Critical`	表示在整个应用程序时失败的消息。
`Error`	表示在当前操作失败的消息。
`Warning`	表示非致命性问题的消息。
`Information`	信息性消息。
`Debug`	用于调试的诊断消息。
`Trace`	用于诊断特定问题非常详细的诊断消息。

配置允许的传输

可以在中配置的 SignalR 使用的传输 `WithUrl` 调用（`withUrl` 在 JavaScript 中）。位 OR 运算的值 `HttpTransportType` 可以用于限制客户端仅使用指定的传输。默认情况下启用所有传输。

例如，若要禁用服务器发送事件传输，但允许 Websocket 和长轮询连接：

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/myhub", HttpTransportType.WebSockets | HttpTransportType.LongPolling)
    .Build();
```

在 JavaScript 客户端，通过设置来配置传输 `transport` 字段上的选项对象提供给 `withUrl`：

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/myhub", { transport: signalR.HttpTransportType.WebSockets |
signalR.HttpTransportType.LongPolling })
    .build();
```

配置持有者身份验证

若要提供身份验证数据以及 SignalR 请求，请使用 `AccessTokenProvider` 选项（`accessTokenFactory` 在 JavaScript 中）指定函数返回的所需的访问令牌。在.NET 客户端，此访问令牌作为传入 HTTP"持有者身份验证"令牌（使用 `Authorization` 具有一种类型的标头 `Bearer`）。在 JavaScript 客户端，使用访问令牌作为持有者令牌，除在少数情况下，在其中浏览器 Api 限制应用（具体而言，在服务器发送事件和 Websocket 请求）的标头的能力。在这些情况下，查询字符串值中提供的访问令牌 `access_token`。

在.NET 客户端，`AccessTokenProvider` 可以使用中的选项委托指定选项 `WithUrl`：

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/myhub", options => {
        options.AccessTokenProvider = async () => {
            // Get and return the access token.
        };
    })
    .Build();
```

在 JavaScript 客户端，通过设置配置的访问令牌 `accessTokenFactory` 字段中的选项对象 `withUrl`：

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/myhub", {
        accessTokenFactory: () => {
            // Get and return the access token.
            // This function can return a JavaScript Promise if asynchronous
            // logic is required to retrieve the access token.
        }
    })
    .build();
```

配置超时和保持活动状态的选项

还提供用于配置超时和保持活动状态的行为的其他选项 `HubConnection` 对象本身：

.NET 选项	JAVASCRIPT 选项	默认值	描述
`ServerTimeout`	`serverTimeoutInMilliseconds`	30 秒（30000 毫秒）	服务器活动的超时时间。如果服务器尚未在此时间间隔内发送一条消息，客户端会考虑服务器断开连接和触发器 `Closed` 事件（`onclose` 在 JavaScript 中）。此值必须足够大，以便从服务器发送的 ping 消息和超时间隔内收到的客户端。建议的值是一个数字至少两倍的服务器的 `KeepAliveInterval` 值，以允许 ping 到达的时间。

.NET 选项	JAVASCRIPT 选项	默认值	描述
`HandshakeTimeout`	不可配置	15 秒	初始服务器握手的超时时间。如果服务器不在此时间间隔内发送握手响应，客户端取消握手和触发器 `Closed` 事件（`onclose` 在 JavaScript 中）。这是一种高级的设置，如果由于出现严重的网络延迟发生握手超时错误应仅修改。握手过程的更多详细信息，请参阅SignalR 集线器协议规范。

在.NET 客户端，超时值指定为 `TimeSpan` 值。在 JavaScript 客户端，超时值指定为一个数字，指示以毫秒为单位的持续时间。

配置其他选项

可以在配置其他选项 `WithUrl`（`withUrl` 在 JavaScript 中）上的方法 `HubConnectionBuilder`：

.NET 选项	JAVASCRIPT 选项	默认值	描述
`AccessTokenProvider`	`accessTokenFactory`	`null`	返回一个字符串，作为持有者身份验证令牌的 HTTP 请求中提供的函数。
`SkipNegotiation`	`skipNegotiation`	`false`	将此设置为 `true` 跳过协商步骤。**WebSockets 传输是唯一的已启用的传输时，才支持**。使用 Azure SignalR 服务时，不能启用此设置。
`ClientCertificates`	不可配置 *	空	若要发送请求进行身份验证的 TLS 证书的集合。
`Cookies`	不可配置 *	空	要与每个 HTTP 请求一起发送的 HTTP cookie 的集合。
`Credentials`	不可配置 *	空	要与每个 HTTP 请求一起发送的凭据。
`CloseTimeout`	不可配置 *	5 秒	仅 Websocket。最长时间之后关闭服务器以确认关闭请求等待客户端。如果服务器不在此时间内收到结束时，客户端断开连接。
`Headers`	不可配置 *	空	附加 HTTP 标头要与每个 HTTP 请求一起发送的字典。

.NET 选项	JAVASCRIPT 选项	默认值	描述
`HttpMessageHandlerFactory`	不可配置 *	`null`	一个委托，它可用于配置或替换 `HttpMessageHandler` 用于发送 HTTP 请求。不用于 WebSocket 连接。此委托必须返回一个非 null 值，并接收作为参数的默认值。修改该默认值上的设置，返回它，或者返回一个新 `HttpMessageHandler` 实例。**时替换处理程序请务必复制你想要保留从提供的处理程序的设置，否则，配置的选项（例如 Cookie 和标头）不会应用于新的处理程序。**
`Proxy`	不可配置 *	`null`	发送 HTTP 请求时要使用 HTTP 代理。
`UseDefaultCredentials`	不可配置 *	`false`	设置此布尔值，若要发送的 HTTP 和 Websocket 请求的默认凭据。这使使用 Windows 身份验证。
`WebSocketConfiguration`	不可配置 *	`null`	一个委托，可用于配置更多的 WebSocket 选项。接收的实例ClientWebSocketOptions 可用于配置选项。

选项标有星号（*）不是可在 JavaScript 客户端，由于浏览器 Api 中的限制中配置的。

在.NET 客户端，这些选项可以修改选项委托提供给 `WithUrl`：

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/myhub", options => {
        options.Headers["Foo"] = "Bar";
        options.Cookies.Add(new Cookie(/* ... */);
        options.ClientCertificates.Add(/* ... */);
    })
    .Build();
```

在 JavaScript 客户端，可以提供给 JavaScript 对象中提供这些选项 `withUrl`：

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/myhub", {
        skipNegotiation: true,
        transport: signalR.HttpTransportType.WebSockets
    })
    .build();
```

其他资源

- ASP.NET Core SignalR 入门
- 在 ASP.NET Core SignalR 使用中心
- ASP.NET Core SignalR JavaScript 客户端

- ASP.NET Core SignalR.NET 客户端
- 使用 ASP.NET Core MessagePack 中心协议中 SignalR
- ASP.NET Core SignalR 支持的平台

身份验证和授权在 ASP.NET Core SignalR

2018/12/12 • • Edit Online

通过Andrew Stanton-nurse

查看或下载示例代码（如何下载）

连接到的 SignalR hub 的用户进行身份验证

可与 SignalR ASP.NET Core 身份验证若要将用户与每个连接相关联。在中心，可以从访问身份验证数据 `HubConnectionContext.User` 属性。身份验证允许中心与用户关联的所有连接上调用方法（请参阅SignalR 中管理用户和组有关详细信息）。多个连接可能包含单个用户相关联。

Cookie 身份验证

在基于浏览器的应用中，cookie 身份验证允许你现有的用户凭据以自动传递到 SignalR 连接。如果使用浏览器客户端，则需要无额外配置。如果用户登录到你的应用，SignalR 连接会自动继承此身份验证。

Cookie 是特定于浏览器的方法，将发送访问令牌，但非浏览器客户端可以向他们发送。使用时.NET 客户端，则 `Cookies` 属性可以配置在 `.WithUrl` 调用，以提供一个 cookie。但是，使用 cookie 身份验证从.NET 客户端要求提供要交换的 cookie 身份验证数据的 API 应用。

持有者令牌身份验证

客户端可以提供访问令牌而不是使用 cookie。服务器验证该令牌，并使用它来标识用户。仅在建立连接时，才执行此验证。连接的生命周期，服务器不会自动重新验证令牌吊销检查。

在服务器上，使用配置持有者令牌身份验证JWT 持有者中间件。

在 JavaScript 客户端，该令牌可以使用提供accessTokenFactory选项。

```
this.connection = new signalR.HubConnectionBuilder()
    .withUrl("/hubs/chat", { accessTokenFactory: () => this.loginToken })
    .build();
```

在.NET 客户端，没有类似AccessTokenProvider属性，可用于配置的令牌：

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/myhub", options =>
    {
        options.AccessTokenProvider = () => Task.FromResult(_myAccessToken);
    })
    .Build();
```

> **NOTE**
>
> 你提供的访问令牌函数之前调用每个所做的 SignalR 的 HTTP 请求。如果需要续订令牌才能保持连接处于活动状态（因为它在连接期间可能会过期），此函数中执行此操作从并返回更新的令牌。

在标准 web Api，持有者令牌将发送 HTTP 标头中。但是，SignalR 是无法使用某些传输通道时在浏览器中设置这些标头。使用 Websocket 和服务器发送事件时，会将令牌传输作为查询字符串参数。若要在服务器上支持此功能，则需要其他配置：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddAuthentication(options =>
        {
            // Identity made Cookie authentication the default.
            // However, we want JWT Bearer Auth to be the default.
            options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
            options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
        })
        .AddJwtBearer(options =>
        {
            // Configure JWT Bearer Auth to expect our security key
            options.TokenValidationParameters =
                new TokenValidationParameters
                {
                    LifetimeValidator = (before, expires, token, param) =>
                    {
                        return expires > DateTime.UtcNow;
                    },
                    ValidateAudience = false,
                    ValidateIssuer = false,
                    ValidateActor = false,
                    ValidateLifetime = true,
                    IssuerSigningKey = SecurityKey
                };

            // We have to hook the OnMessageReceived event in order to
            // allow the JWT authentication handler to read the access
            // token from the query string when a WebSocket or
            // Server-Sent Events request comes in.
            options.Events = new JwtBearerEvents
            {
                OnMessageReceived = context =>
                {
                    var accessToken = context.Request.Query["access_token"];

                    // If the request is for our hub...
                    var path = context.HttpContext.Request.Path;
                    if (!string.IsNullOrEmpty(accessToken) &&
                        (path.StartsWithSegments("/hubs/chat")))
                    {
                        // Read the token out of the query string
                        context.Token = accessToken;
                    }
                    return Task.CompletedTask;
                }
            };
        });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddSignalR();

    // Change to use Name as the user identifier for SignalR
    // WARNING: This requires that the source of your JWT token
    // ensures that the Name claim is unique!
    // If the Name claim isn't unique, users could receive messages
    // intended for a different user!
    services.AddSingleton<IUserIdProvider, NameUserIdProvider>();
}
```

与持有者令牌的 cookie

因为 cookie 是特定于浏览器，将它们发送从其他类型的客户端会增加复杂性相比发送持有者令牌。出于此原因，不被建议的 cookie 身份验证，除非应用只需要从浏览器客户端的用户进行身份验证。使用非浏览器客户端的客户端时，持有者令牌身份验证是建议的方法。

Windows 身份验证

如果Windows 身份验证配置为在应用中，SignalR 可以使用该标识安全中心。但是，若要将消息发送给单个用户，您需要添加自定义的用户 ID 提供程序。这是因为 Windows 身份验证系统不提供 SignalR 使用来确定用户名称的"名称标识符"声明。

添加新的类实现 `IUserIdProvider` 和检索的声明之一中要用作标识符的用户。例如，若要使用"Name"声明 (即窗体中的 Windows 用户名 `[Domain]\[Username]`)，创建以下类：

```
public class NameUserIdProvider : IUserIdProvider
{
    public string GetUserId(HubConnectionContext connection)
    {
        return connection.User?.FindFirst(ClaimTypes.Name)?.Value;
    }
}
```

而非 `ClaimTypes.Name`，可以使用的任何值 `User`（例如 Windows SID 标识符等）。

> **NOTE**
> 您选择的值必须是在系统中所有用户之间唯一的。否则，适用于一个用户的消息可能最终转到不同的用户。

注册此组件中的您 `Startup.ConfigureServices` 方法后对的调用 `.AddSignalR`

```
public void ConfigureServices(IServiceCollection services)
{
    // ... other services ...

    services.AddSignalR();
    services.AddSingleton<IUserIdProvider, NameUserIdProvider>();
}
```

在.NET 客户端，必须通过设置启用 Windows 身份验证UseDefaultCredentials属性：

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/myhub", options =>
    {
        options.UseDefaultCredentials = true;
    })
    .Build();
```

使用 Microsoft Internet Explorer 或 Microsoft Edge 时，浏览器客户端仅支持 Windows 身份验证。

授权用户访问集线器和集线器方法

默认情况下，可以通过未经身份验证的用户调用一个中心中的所有方法。为了要求身份验证，应用Authorize属性为中心：

```
[Authorize]
public class ChatHub: Hub
{
}
```

可以使用的构造函数参数和属性 `[Authorize]` 属性以限制只有匹配特定的用户访问授权策略。例如，如果具有名为的自定义授权策略 `MyAuthorizationPolicy` 可以确保只有匹配该策略的用户可以访问的中心使用以下代码：

```
[Authorize("MyAuthorizationPolicy")]
public class ChatHub: Hub
{
}
```

单个集线器方法可以具有 `[Authorize]` 也应用属性。如果当前用户不匹配的策略应用于方法，是向调用方返回错误：

```
[Authorize]
public class ChatHub: Hub
{
    public async Task Send(string message)
    {
        // ... send a message to all users ...
    }

    [Authorize("Administrators")]
    public void BanUser(string userName)
    {
        // ... ban a user from the chat room (something only Administrators can do) ...
    }
}
```

其他资源

- 在 ASP.NET Core 中的持有者令牌身份验证

ASP.NET Core SignalR 中的安全注意事项

2018/11/13 • • Edit Online

通过Andrew Stanton-nurse

本文提供有关保护 SignalR 的信息。

跨域资源共享

跨域资源共享 (CORS)可用于在浏览器中允许跨域 SignalR 连接。如果在不同的域从 SignalR 应用程序中托管的 JavaScript 代码CORS 中间件必须启用才能允许 JavaScript 才能连接到 SignalR 应用程序。允许仅从您信任的域或控件的跨域请求。例如：

- 上托管你的站点 `http://www.example.com`
- 上托管您的 SignalR 应用程序 `http://signalr.example.com`

应为只允许原点的 SignalR 应用中配置 CORS `www.example.com`。

配置 CORS 的详细信息，请参阅启用跨域请求 (CORS)。SignalR需要以下 CORS 策略：

- 允许特定的预期的来源。允许任何源，但不安全或建议。
- HTTP 方法 `GET` 和 `POST` 必须允许。
- 必须启用凭据，即使不使用身份验证。

例如，以下的 CORS 策略允许 SignalR 浏览器客户端上托管 `https://example.com` 访问上托管的 SignalR 应用 `https://signalr.example.com`：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    // ... other middleware ...

    // Make sure the CORS middleware is ahead of SignalR.
    app.UseCors(builder =>
    {
        builder.WithOrigins("https://example.com")
            .AllowAnyHeader()
            .WithMethods("GET", "POST")
            .AllowCredentials();
    });

    // ... other middleware ...

    app.UseSignalR(routes =>
    {
        routes.MapHub<ChatHub>("/chatHub");
    });

    // ... other middleware ...
}
```

> **NOTE**
>
> SignalR 是不与 Azure 应用服务中内置的 CORS 功能兼容。

WebSocket 源限制

CORS 提供的保护功能不会应用到 Websocket。有关源限制对 Websocket，请阅读Websocket 原点限制。

CORS 提供的保护功能不会应用到 Websocket。浏览器执行操作不：

- 执行 CORS 预检请求。
- 遵循中指定的限制 `Access-Control` 标头时发出 WebSocket 请求。

但是，浏览器是否将发送 `Origin` 标头时发出 WebSocket 请求。应用程序应配置为验证这些标头，以确保允许仅来自预期的来源的 Websocket。

ASP.NET Core 2.1 及更高版本，可以使用放置自定义中间件实现标头验证之前 `UseSignalR`，和身份验证中间件中 `Configure`：

```
// In Startup, add a static field listing the allowed Origin values:
private static readonly HashSet<string> _allowedOrigins = new HashSet<string>()
{
    // Add allowed origins here. For example:
    "https://www.mysite.com",
    "https://mysite.com",
};

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    // ... other middleware ...

    // Validate Origin header on WebSocket requests to prevent unexpected cross-site
    // WebSocket requests.
    app.Use((context, next) =>
    {
        // Check for a WebSocket request.
        if (string.Equals(context.Request.Headers["Upgrade"], "websocket"))
        {
            var origin = context.Request.Headers["Origin"];

            // If there is an origin header, and the origin header doesn't match
            // an allowed value:
            if (!string.IsNullOrEmpty(origin) && !_allowedOrigins.Contains(origin))
            {
                // The origin is not allowed, reject the request
                context.Response.StatusCode = StatusCodes.Status403Forbidden;
                return Task.CompletedTask;
            }
        }

        // The request is a valid Origin or not a WebSocket request, so continue.
        return next();
    });

    // ... other middleware ...

    app.UseSignalR(routes =>
    {
        routes.MapHub<ChatHub>("/chatHub");
    });

    // ... other middleware ...
}
```

> **NOTE**
> `Origin` 标头控制客户端和其他如 `Referer` 标头，可能伪造的。这些标头应不用作身份验证机制。

访问令牌的日志记录

使用 Websocket 或服务器发送事件时，浏览器客户端将查询字符串中发送的访问令牌。接收访问令牌通过查询字符串就象通常那样使用标准安全 `Authorization` 标头。应始终使用 HTTPS 以确保客户端和服务器之间的端到端安全连接。许多 web 服务器日志每个请求，包括查询字符串的 URL。日志记录 Url 可能记录的访问令牌。ASP.NET Core 日志默认情况下，其中会包括查询字符串的每个请求的 URL。例如：

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/myhub?access_token=1234
```

如果您有关于此数据与服务器日志的日志记录的相关问题，可以通过配置来完全禁用此日志记录 `Microsoft.AspNetCore.Hosting` 记录器 `Warning` 级别或更高版本（这些消息将写入在 `Info` 级别）。在查看文档日志筛选有关详细信息。如果你仍想要记录特定请求的信息，可以编写中间件来记录的数据需要并筛选出 `access_token` 查询字符串值（如果存在）。

异常

异常消息通常被视为不应泄露给客户端的敏感数据。默认情况下，SignalR 不会发送到客户端集线器方法引发的异常的详细信息。相反，客户端收到一条指示发生了错误的常规消息。异常消息传递到客户端可以重写（例如在开发或测试）使用 `EnableDetailedErrors` 。不应在生产应用中的客户端公开异常消息。

缓冲区管理

SignalR 使用每个连接缓冲区来管理传入和传出消息。默认情况下，SignalR 限制为 32 KB 这些缓冲区。客户端或服务器可以发送的最大消息为 32 KB。消息的连接所占用的最大内存为 32 KB。如果你的消息始终小于 32 KB，可以减少了限制，其中：

- 防止客户端发送大消息。
- 服务器永远不需要分配大型缓冲区接受的消息。

如果你的消息大小大于 32 KB，则可以增加限制。增加此限制意味着：

- 客户端可能导致服务器分配较大内存缓冲区。
- 大型缓冲区的服务器分配可能会降低并发连接数。

限制传入和传出消息，同时可以根据 `HttpConnectionDispatcherOptions` 对象中配置 `MapHub`：

- `ApplicationMaxBufferSize` 表示最大字节数从客户端的服务器缓冲区。如果客户端尝试发送一条消息大小超过此限制，可能会关闭连接。
- `TransportMaxBufferSize` 表示最大服务器都可以发送的字节数。如果服务器尝试发送一条消息（包括从中心的方法返回的值）大于此限制，将引发异常。

将限制设置为 `0` 禁用限制。删除限制允许客户端发送任意大小的消息。恶意客户端发送大消息可能会导致过多内存，并将其分配。过多内存使用情况可以显著减少并发连接数。

使用 ASP.NET Core MessagePack 中心协议中 SignalR

2018/10/15 • • Edit Online

通过brennan，第 Conroy

本文假定读者熟悉中所涉及的主题开始。

MessagePack 是什么？

MessagePack是快速和精简的二进制序列化格式。性能和带宽是一个问题，因为它将创建较小的消息相比时很有用JSON。因为它是以二进制格式，则消息是不可读，查看网络跟踪和日志，除非通过 MessagePack 分析器传递字节时。SignalR 为 MessagePack 格式中，提供内置支持，并提供 Api，可用于要使用的客户端和服务器。

在服务器上配置 MessagePack

若要启用 MessagePack 中心协议的服务器上，安装 `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` 应用程序中的包。在 Startup.cs 文件中添加 `AddMessagePackProtocol` 到 `AddSignalR` 调用，以启用 MessagePack 支持在服务器上的。

> **NOTE**
>
> 默认情况下启用 JSON。添加 MessagePack，对 JSON 和 MessagePack 客户端的支持。

```
services.AddSignalR()
    .AddMessagePackProtocol();
```

若要自定义 MessagePack 设置你的数据的格式 `AddMessagePackProtocol` 接受委托，用于配置选项。在该委托 `FormatterResolvers` 属性可以用于配置 MessagePack 序列化选项。冲突解决程序的工作原理的详细信息，请访问位于的 MessagePack library MessagePack CSharp。要序列化定义应如何处理它们的对象上，可以使用属性。

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        options.FormatterResolvers = new List<MessagePack.IFormatterResolver>()
        {
            MessagePack.Resolvers.StandardResolver.Instance
        };
    });
```

在客户端上配置 MessagePack

> **NOTE**
>
> 支持的客户端的情况下，默认情况下启用 JSON。客户端可以只支持单个协议。添加 MessagePack 支持会将为任何以前配置的协议。

.NET 客户端

若要启用 MessagePack.NET 客户端中的，安装 `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` 包并调用 `AddMessagePackProtocol` 上 `HubConnectionBuilder` 。

```
var hubConnection = new HubConnectionBuilder()
                        .WithUrl("/chatHub")
                        .AddMessagePackProtocol()
                        .Build();
```

> **NOTE**
>
> 这 `AddMessagePackProtocol` 调用接受委托用于配置服务器一样的选项。

JavaScript 客户端

MessagePack 支持 Javascript 客户端提供的 `@aspnet/signalr-protocol-msgpack` NPM 包。

```
npm install @aspnet/signalr-protocol-msgpack
```

安装 npm 包之后，该模块可以使用 JavaScript 模块加载程序通过直接或通过引用导入到浏览器 *node_modules\@aspnet\signalr-protocol-msgpack\dist\browser\signalr-protocol-msgpack.js* 文件。在浏览器 `msgpack5` 还必须引用库。使用 `<script>` 标记创建的引用。可以在找到的库*node_modules\msgpack5\dist\msgpack5.js*。

> **NOTE**
>
> 当使用 `<script>` 元素的顺序非常重要。如果*signalr 协议 msgpack.js*引用之前*msgpack5.js*，尝试使用 MessagePack 连接时出错。*signalr.js*也需要前*signalr 协议 msgpack.js*。

```
<script src="~/lib/signalr/signalr.js"></script>
<script src="~/lib/msgpack5/msgpack5.js"></script>
<script src="~/lib/signalr/signalr-protocol-msgpack.js"></script>
```

添加 `.withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())` 到 `HubConnectionBuilder` 将配置客户端连接到服务器时使用 MessagePack 协议。

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub")
    .withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())
    .build();
```

> **NOTE**
>
> 在此期间，有 JavaScript 客户端上的 MessagePack 协议没有配置选项。

相关资源

- 入门
- .NET 客户端
- JavaScript 客户端

使用 ASP.NET Core SignalR 中流式处理

2019/1/8 • • Edit Online

通过brennan，第 Conroy

ASP.NET Core SignalR 支持流式处理服务器方法的返回值。这是适用于其中的数据片段会随时间的方案。当返回值流式传输到客户端时，每个片段是发送到客户端变得可用，而非等待所有的数据变得可用。

查看或下载示例代码（如何下载）

设置中心

集线器方法自动成为流式处理的集线器方法时它将返回 `ChannelReader<T>` 或 `Task<ChannelReader<T>>` 。下面是一个示例，显示的数据流式传输到客户端的基础知识。每当将对象写入到 `ChannelReader` 该对象将立即发送给客户端。在结束时，`ChannelReader` 完成告诉客户端流已关闭。

> **NOTE**
>
> - 写入 `ChannelReader` 在后台线程并返回 `ChannelReader` 越早越好。将阻止其他集线器调用，直到 `ChannelReader` 返回。
> - 包装中的逻辑 `try ... catch` 并完成 `Channel` catch 和外部的关键点，以确保在中心正确完成方法调用。

```
public class StreamHub : Hub
{
    public ChannelReader<int> Counter(int count, int delay)
    {
        var channel = Channel.CreateUnbounded<int>();

        // We don't want to await WriteItemsAsync, otherwise we'd end up waiting
        // for all the items to be written before returning the channel back to
        // the client.
        _ = WriteItemsAsync(channel.Writer, count, delay);

        return channel.Reader;
    }

    private async Task WriteItemsAsync(
        ChannelWriter<int> writer,
        int count,
        int delay)
    {
        try
        {
            for (var i = 0; i < count; i++)
            {
                await writer.WriteAsync(i);
                await Task.Delay(delay);
            }
        }
        catch (Exception ex)
        {
            writer.TryComplete(ex);
        }

        writer.TryComplete();
    }
}
```

```
public class StreamHub : Hub
{
    public ChannelReader<int> Counter(
        int count,
        int delay,
        CancellationToken cancellationToken)
    {
        var channel = Channel.CreateUnbounded<int>();

        // We don't want to await WriteItemsAsync, otherwise we'd end up waiting
        // for all the items to be written before returning the channel back to
        // the client.
        _ = WriteItemsAsync(channel.Writer, count, delay, cancellationToken);

        return channel.Reader;
    }

    private async Task WriteItemsAsync(
        ChannelWriter<int> writer,
        int count,
        int delay,
        CancellationToken cancellationToken)
    {
        try
        {
            for (var i = 0; i < count; i++)
            {
                // Check the cancellation token regularly so that the server will stop
                // producing items if the client disconnects.
                cancellationToken.ThrowIfCancellationRequested();
                await writer.WriteAsync(i);

                // Use the cancellationToken in other APIs that accept cancellation
                // tokens so the cancellation can flow down to them.
                await Task.Delay(delay, cancellationToken);
            }
        }
        catch (Exception ex)
        {
            writer.TryComplete(ex);
        }

        writer.TryComplete();
    }
}
```

在 ASP.NET Core 2.2 或更高版本，流式处理集线器方法可以接受 `CancellationToken` 从流的客户端取消订阅时，将触发的参数。使用此令牌来停止服务器操作和释放任何资源，如果客户端断开连接之前流的末尾。

.NET 客户端

`StreamAsChannelAsync` 方法 `HubConnection` 用于调用流式处理方法。将中心方法名称和到中心方法中定义的自变量传递 `StreamAsChannelAsync` 。上的泛型参数 `StreamAsChannelAsync<T>` 指定流式处理方法返回的对象的类型。一个 `ChannelReader<T>` 返回从流调用，并且表示在客户端上的流。若要读取数据，常见模式是循环访问 `WaitToReadAsync` ，并调用 `TryRead` 数据时可用。循环将结束时该流已关闭服务器，或取消标记传递给 `StreamAsChannelAsync` 被取消。

```
// Call "Cancel" on this CancellationTokenSource to send a cancellation message to
// the server, which will trigger the corresponding token in the Hub method.
var cancellationTokenSource = new CancellationTokenSource();
var channel = await hubConnection.StreamAsChannelAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

// Wait asynchronously for data to become available
while (await channel.WaitToReadAsync())
{
    // Read all currently available data synchronously, before waiting for more data
    while (channel.TryRead(out var count))
    {
        Console.WriteLine($"{count}");
    }
}

Console.WriteLine("Streaming completed");
```

```
var channel = await hubConnection
    .StreamAsChannelAsync<int>("Counter", 10, 500, CancellationToken.None);

// Wait asynchronously for data to become available
while (await channel.WaitToReadAsync())
{
    // Read all currently available data synchronously, before waiting for more data
    while (channel.TryRead(out var count))
    {
        Console.WriteLine($"{count}");
    }
}

Console.WriteLine("Streaming completed");
```

JavaScript 客户端

JavaScript 客户端调用集线器上流式处理方法使用 `connection.stream` 。 `stream` 方法接受两个参数：

- 集线器方法的名称。在以下示例中，中心方法名称是 `Counter` 。
- 在集线器方法中定义的参数。在以下示例中，参数是：若要接收的流项和流项之间的延迟的数量的计数。

`connection.stream` 返回 `IStreamResult` 其中包含 `subscribe` 方法。传递 `IStreamSubscriber` 到 `subscribe` 并设置 `next` ， `error` ，并 `complete` 回调以获取来自通知 `stream` 调用。

```
connection.stream("Counter", 10, 500)
    .subscribe({
        next: (item) => {
            var li = document.createElement("li");
            li.textContent = item;
            document.getElementById("messagesList").appendChild(li);
        },
        complete: () => {
            var li = document.createElement("li");
            li.textContent = "Stream completed";
            document.getElementById("messagesList").appendChild(li);
        },
        error: (err) => {
            var li = document.createElement("li");
            li.textContent = err;
            document.getElementById("messagesList").appendChild(li);
        },
});
```

若要结束从客户端的流，请调用 `dispose` 方法 `ISubscription` 从返回 `subscribe` 方法。

若要结束从客户端的流，请调用 `dispose` 方法 `ISubscription` 从返回 `subscribe` 方法。调用此方法将导致 `CancellationToken`（如果提供）的集线器方法的参数以取消。

相关资源

- 中心
- .NET 客户端
- JavaScript 客户端
- 发布到 Azure

ASP.NET SignalR 和 ASP.NET Core SignalR 之间的差异

2018/12/27 • • Edit Online

ASP.NET Core SignalR 与 ASP.NET SignalR 的客户端或服务器不兼容。本文详细介绍 ASP.NET Core SignalR 中已删除或已更改的功能。

如何识别 SignalR 版本

	ASP.NET SIGNALR	ASP.NET CORE SIGNALR
服务器 NuGet 包	Microsoft.AspNet.SignalR	Microsoft.AspNetCore.App (.NET Core) Microsoft.AspNetCore.SignalR (.NET Framework)
客户端 NuGet 包	Microsoft.AspNet.SignalR.Client Microsoft.AspNet.SignalR.JS	Microsoft.AspNetCore.SignalR.Client
客户端 npm 包	signalr	@aspnet/signalr
Java 客户端	GitHub 存储库（已弃用）	Maven 包com.microsoft.signalr
服务器应用类型	ASP.NET (System.Web) 或 OWIN 自承载	ASP.NET Core
受支持的服务器平台	.NET framework 4.5 或更高版本	.NET Framework 4.6.1 或更高版本 .NET core 2.1 或更高版本

功能差异

自动重新连接

ASP.NET Core SignalR 不支持自动重新连接。如果客户端已断开连接，则用户必须显式启动新连接才能重新连接。在 ASP.NET SignalR 中，如果连接断开，SignalR 会尝试重新连接到服务器。

协议支持

ASP.NET Core SignalR 支持 JSON 以及一种基于 MessagePack 的新二进制协议。此外，还可创建自定义协议。

传输

ASP.NET Core SignalR 不支持永久帧传输。

服务器上的差异

ASP.NET Core SignalR 服务器端库包含在 Microsoft.AspNetCore.App 元包中，该包属于适用于 Razor 和 MVC 项目的 **ASP.NET Core Web** 应用程序模板。

ASP.NET Core SignalR 是一个 ASP.NET Core 中间件，因此必须通过在 `Startup.ConfigureServices` 中调用 AddSignalR 来配置它。

```
services.AddSignalR()
```

若要配置路由，请在 `Startup.Configure` 方法中将路由映射到 UseSignalR 方法调用内的中心。

```
app.UseSignalR(routes =>
{
    routes.MapHub<ChatHub>("/hub");
});
```

粘滞会话

ASP.NET SignalR 的横向扩展模型允许客户端重新连接场中的任何服务器并将消息发送到这些服务器。在 ASP.NET Core SignalR 中，客户端必须在连接期间与同一服务器进行交互。对于使用 Redis 的横向扩展，这意味着需要粘滞会话。对于使用 Azure SignalR 服务的横向扩展，则不需要粘滞会话，因为该服务会处理与客户端的连接。

一个连接一个中心

在ASP.NET Core SignalR中，连接模型已经简化。可直接连接到单个中心，不必通过单个连接来共享对多个中心的访问。

流式处理

ASP.NET Core SignalR 现在支持从中心流式传输数据到客户端。

状态

已删除在客户端和中心之间传递任意状态（通常称为 HubState）的功能以及对进度消息的支持。目前没有中心代理的对应项。

删除 PersistentConnection

在 ASP.NET Core SignalR 中，PersistentConnection 类已删除。

GlobalHost

ASP.NET Core 在框架中内置了依赖关系注入 (DI)。服务可以使用 DI 来访问 HubContext。ASP.NET CoreR SignalR 中不存在用于在 ASP.NET SignalR 中获取 `HubContext` 的 `GlobalHost` 对象。

HubPipeline

ASP.NET Core SignalR 没有对 `HubPipeline` 模块的支持。

客户端上的差异

TypeScript

ASP.NET Core SignalR 客户端以 TypeScript 编写。使用 JavaScript 客户端时，可以以 JavaScript 或 TypeScript 编写。

JavaScript 客户端托管在 npm 上

在以前的版本中，JavaScript 客户端是通过 Visual Studio 中的 NuGet 包获取的。Core 版本的 @aspnet/signalr npm 包包含 JavaScript 库。此包不包括在**ASP.NET Core Web 应用程序**模板。使用 npm 获取并安装 `@aspnet/signalr` npm 包。

```
npm init -y
npm install @aspnet/signalr
```

jQuery

已删除对 jQuery 的依赖关系，但项目仍然可以使用 jQuery。

Internet Explorer 支持

ASP.NET Core SignalR 需要 Microsoft Internet Explorer 11 或更高版本（ASP.NET SignalR 支持 Microsoft Internet Explorer 8 及更高版本）。

JavaScript 客户端方法语法

JavaScript 语法已与 Signalr 早期版本中的相应语法不同。请使用 HubConnectionBuilder API 而非 `$connection` 对象创建连接。

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();
```

使用 on 方法指定可供中心调用的客户端方法。

```
connection.on("ReceiveMessage", (user, message) => {
    const msg = message.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;");
    const encodedMsg = user + " says " + msg;
    log(encodedMsg);
});
```

创建客户端方法后，启动中心连接。链接一种 catch 方法以记录或处理错误。

```
connection.start().catch(err => console.error(err.toString()));
```

中心代理

中心代理不再自动生成。与之相反，方法名称将以字符串形式传递到 invoke API 中。

.NET 和其他客户端

`Microsoft.AspNetCore.SignalR.Client` NuGet 包中包含 ASP.NET Core SignalR 的 .NET 客户端库。

使用 HubConnectionBuilder 创建和生成到中心的连接的实例。

```
connection = new HubConnectionBuilder()
    .WithUrl("url")
    .Build();
```

横向扩展差异

ASP.NET SignalR 支持 SQL Server 和 Redis。ASP.NET Core SignalR 支持 Azure SignalR 服务和 Redis。

ASP.NET

- 使用 Azure 服务总线的 SignalR 横向扩展
- 使用 Redis 的 SignalR 横向扩展
- 使用 SQL Server 的 SignalR 横向扩展

ASP.NET Core

- Azure SignalR 服务

其他资源

- 中心
- JavaScript 客户端
- .NET 客户端
- 支持的平台

ASP.NET Core 中的 WebSocket 支持

2018/11/14 • • Edit Online

作者：Tom Dykstra 和 Andrew Stanton-Nurse

本文介绍 ASP.NET Core 中 WebSocket 的入门方法。WebSocket (RFC 6455) 是一个协议，支持通过 TCP 连接建立持久的双向信道。它用于从快速实时通信中获益的应用，如聊天、仪表板和游戏应用。

查看或下载示例代码（如何下载）。有关详细信息，请参阅后续步骤部分。

系统必备

- ASP.NET Core 1.1 或更高版本
- 支持 ASP.NET Core 的任何操作系统：
  - Windows 7/Windows Server 2008 或更高版本
  - Linux
  - macOS
- 如果应用在安装了 IIS 的 Windows 上运行：
  - Windows 8 / Windows Server 2012 及更高版本
  - IIS 8 / IIS 8 Express
  - 必须启用 WebSocket（请参阅 IIS/IIS Express 支持部分。）。
- 如果应用在 HTTP.sys 上运行：
  - Windows 8 / Windows Server 2012 及更高版本
- 有关支持的浏览器，请参阅 https://caniuse.com/#feat=websockets。

何时使用 WebSocket

通过 WebSocket 可直接使用套接字连接。例如，使用 WebSocket 可以让实时游戏达到最佳性能。

ASP.NET Core SignalR 是一个库，可用于简化向应用添加实时 Web 功能。它会尽可能地使用 WebSocket。

如何使用 Websocket

- 安装 Microsoft.AspNetCore.WebSockets 包。
- 配置中间件。
- 接受 WebSocket 请求。
- 发送和接收消息。

配置中间件

在 `Startup` 类的 `Configure` 方法中添加 WebSocket 中间件：

```
app.UseWebSockets();
```


```
app.UseWebSockets();
```


可配置以下设置：

- `KeepAliveInterval` - 向客户端发送“ping”帧的频率，以确保代理保持连接处于打开状态。默认值为 2 分钟。
- `ReceiveBufferSize` - 用于接收数据的缓冲区的大小。高级用户可能需要对其进行更改，以便根据数据大小调整性能。默认值为 4 KB。

可配置以下设置：

- `KeepAliveInterval` - 向客户端发送“ping”帧的频率，以确保代理保持连接处于打开状态。默认值为 2 分钟。
- `ReceiveBufferSize` - 用于接收数据的缓冲区的大小。高级用户可能需要对其进行更改，以便根据数据大小调整性能。默认值为 4 KB。
- `AllowedOrigins` - 用于 WebSocket 请求的允许的 Origin 标头值列表。默认情况下，允许使用所有源。有关详细信息，请参阅以下“WebSocket 源限制”。

```
var webSocketOptions = new WebSocketOptions()
{
    KeepAliveInterval = TimeSpan.FromSeconds(120),
    ReceiveBufferSize = 4 * 1024
};
app.UseWebSockets(webSocketOptions);
```


```
var webSocketOptions = new WebSocketOptions()
{
    KeepAliveInterval = TimeSpan.FromSeconds(120),
    ReceiveBufferSize = 4 * 1024
};
app.UseWebSockets(webSocketOptions);
```


接受 WebSocket 请求

在请求生命周期后期（例如在 `Configure` 方法或 MVC 操作的后期），检查它是否是 WebSocket 请求并接受 WebSocket 请求。

以下示例来自 `Configure` 方法的后期：

```
app.Use(async (context, next) =>
{
    if (context.Request.Path == "/ws")
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync();
            await Echo(context, webSocket);
        }
        else
        {
            context.Response.StatusCode = 400;
        }
    }
    else
    {
        await next();
    }

});
```

```
app.Use(async (context, next) =>
{
    if (context.Request.Path == "/ws")
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync();
            await Echo(context, webSocket);
        }
        else
        {
            context.Response.StatusCode = 400;
        }
    }
    else
    {
        await next();
    }

});
```

WebSocket 请求可以来自任何 URL，但此示例代码只接受 `/ws` 的请求。

发送和接收消息

`AcceptWebSocketAsync` 方法将 TCP 连接升级到 WebSocket 连接，并提供 WebSocket 对象。使用 `WebSocket` 对象发送和接收消息。

之前显示的接受 WebSocket 请求的代码将 `WebSocket` 对象传递给 `Echo` 方法。代码接收消息并立即发回相同的消息。循环发送和接收消息，直到客户端关闭连接：

```
private async Task Echo(HttpContext context, WebSocket webSocket)
{
    var buffer = new byte[1024 * 4];
    WebSocketReceiveResult result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
CancellationToken.None);
    while (!result.CloseStatus.HasValue)
    {
        await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, result.Count), result.MessageType,
result.EndOfMessage, CancellationToken.None);

        result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);
    }
    await webSocket.CloseAsync(result.CloseStatus.Value, result.CloseStatusDescription,
CancellationToken.None);
}
```


```
private async Task Echo(HttpContext context, WebSocket webSocket)
{
    var buffer = new byte[1024 * 4];
    WebSocketReceiveResult result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
CancellationToken.None);
    while (!result.CloseStatus.HasValue)
    {
        await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, result.Count), result.MessageType,
result.EndOfMessage, CancellationToken.None);

        result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);
    }
    await webSocket.CloseAsync(result.CloseStatus.Value, result.CloseStatusDescription,
CancellationToken.None);
}
```

如果在开始循环之前接受 WebSocket 连接，中间件管道会结束。关闭套接字后，管道展开。即接受 WebSocket 时，请求停止在管道中推进。循环结束且套接字关闭时，请求继续回到管道。

WebSocket 源限制

CORS 提供的保护不适用于 WebSocket。浏览器不会：

- 执行 CORS 预检请求。
- 在发出 WebSocket 请求时，遵守 `Access-Control` 标头中指定的限制。

但是，浏览器在发出 WebSocket 请求时会发送 `Origin` 标头。应将应用程序配置为验证这些标头，以确保只允许来自预期来源的 WebSocket。

如果在“https://server.com”上托管服务器并在“https://client.com”上托管客户端，请将“https://client.com”添加到 `AllowedOrigins` 列表以验证 WebSocket。

```
app.UseWebSockets(new WebSocketOptions()
{
    AllowedOrigins.Add("https://client.com");
    AllowedOrigins.Add("https://www.client.com");
});
```

> **NOTE**
> 与 `Referer` 标头一样，`Origin` 标头由客户端控制，并可以伪造。请勿将这些标头用作身份验证机制。

IIS/IIS Express 支持

安装了 IIS/IIS Express 8 或更高版本的 Windows Server 2012 或更高版本以及 Windows 8 或更高版本支持 WebSocket 协议。

> **NOTE**
> 使用 IIS Express 时始终启用 WebSocket。

在 IIS 上启用 Websocket

在 Windows Server 2012 或更高版本上启用对 WebSocket 协议的支持：

> **NOTE**
> 使用 IIS Express 时无需执行这些步骤

1. 通过“管理”菜单或“服务器管理器”中的链接使用“添加角色和功能”向导。
2. 选择“基于角色或基于功能的安装”。选择“下一步”。
3. 选择适当的服务器（默认情况下选择本地服务器）。选择“下一步”。
4. 在“角色”树中展开“Web 服务器 (IIS)”、然后依次展开“Web 服务器”和“应用程序开发”。
5. 选择“WebSocket 协议”。选择“下一步”。
6. 如果无需其他功能，请选择“下一步”。
7. 选择“安装” 。
8. 安装完成后，选择“关闭”以退出向导。

在 Windows 8 或更高版本上启用对 WebSocket 协议的支持：

> **NOTE**
> 使用 IIS Express 时无需执行这些步骤

1. 导航到“控制面板” > “程序” > “程序和功能” > “打开或关闭 Windows 功能”(位于屏幕左侧)。
2. 打开以下节点:“Internet Information Services” > “万维网服务” > “应用程序开发功能”。
3. 选择“WebSocket 协议”功能。选择“确定”。

在 Node.js 上使用 socket.io 时禁用 WebSocket

如果在 Node.js 的 socket.io 中使用 WebSocket 支持，请使用 web.config 或 applicationHost.config 中的 `webSocket` 元素禁用默认的 IIS WebSocket 模块。如果不执行此步骤，IIS WebSocket 模块将尝试处理 WebSocket 通信而不是 Node.js 和应用。

```
<system.webServer>
  <webSocket enabled="false" />
</system.webServer>
```

后续步骤

本文附带的示例应用是一个 echo 应用。它有一个可建立 WebSocket 连接的网页，且服务器将其收到的消息重新发回到客户端。从命令提示符运行该应用(它未设置为在安装了 IIS Express 的 Visual Studio 中运行)并导航到 http://localhost:5000。该网页的左上方显示连接状态:

选择“连接”，向显示的 URL 发送 WebSocket 请求。输入测试消息并选择“发送”。完成后，请选择“关闭套接字”。“通信日志”部分会报告每一个发生的“打开”、“发送”和“关闭”操作。

localhost:5000
localhost:5000
Apps
ASP.NET Repo
ASP.NET Core Docs
WebSocket Sample Application
Closed
WebSocket Server URL:
ws://localhost:5000/ws
Connect
Message to send:
second test message
Send
Close Socket
Communication Log
From
To
Data
Connection opened
Client
Server
first test message
Server
Client
first test message
Client
Server
second test message
Server
Client
second test message
Connection closed. Code: 1000. Reason: Closing from client

在 ASP.NET Core razor 页单元测试

2018/10/30 • • Edit Online

作者：Luke Latham

ASP.NET Core 支持 Razor 页应用的单元测试。测试数据的访问层 (DAL) 和页面模型帮助确保：

- Razor 页面应用的部分独立工作和一起作为一个单元应用构建过程。
- 类和方法具有有限的作用域的责任。
- 应用程序的行为方式上存在的其他文档。
- 自动的生成和部署期间发现错误由更新的代码，这些错误的回归。

本主题假定你基本了解 Razor 页面应用程序和单元测试。如果您是熟悉 Razor 页面应用程序或测试概念，请参阅以下主题：

- Razor 页面介绍
- Razor 页面入门
- 单元测试 C# 中使用 dotnet 测试和 xUnit 的.NET Core

查看或下载示例代码（如何下载）

示例项目包含两个应用：

应用	项目文件夹	描述
消息应用程序	*src/RazorPagesTestSample*	允许用户添加、删除其中一个，删除所有，并分析消息。
测试应用	*tests/RazorPagesTestSample.Tests*	用于对消息应用进行单元测试：数据访问层 (DAL) 和索引页面模型。

可以使用内置测试功能的 IDE，如运行测试Visual Studio。如果使用Visual Studio Code或命令行中，执行以下命令在命令提示符处*tests/RazorPagesTestSample.Tests*文件夹：

```
dotnet test
```

消息应用的组织

消息应用程序是一个简单的 Razor 页面消息系统，具有以下特征：

- 应用程序的索引页 (*pages/Index.cshtml*并*Pages/Index.cshtml.cs*) 提供 UI 和页面模型方法，用于控制添加、删除和分析的消息（每个消息的平均词）.
- 一条消息由描述 `Message` 类 (*Data/Message.cs*) 具有两个属性：`Id`（密钥）和 `Text`（消息）。`Text` 属性是必需的限制为 200 个字符。
- 使用存储的消息Entity Framework 的内存中数据库†。
- 应用程序包含在其数据库上下文类中，数据访问层 (DAL) `AppDbContext` (*Data/AppDbContext.cs*)。DAL 方法被标记为 `virtual`，它允许模拟在测试中使用的方法。
- 如果数据库为空应用程序启动时，使用三个消息初始化消息存储区。这些*设定种子的消息*还在测试中使用。

†EF 主题测试与 InMemory，说明如何使用内存中数据库的使用 MSTest 的测试。本主题使用xUnit测试框架。测试

概念和跨不同测试框架的测试实现有类似，但不是完全相同。

虽然应用不使用存储库模式，并且不是有效的示例工作单元 (UoW) 模式，Razor 页面支持开发这些模式。有关详细信息，请参阅设计基础结构持久性层并测试控制器逻辑（此示例实现存储库模式）。

测试应用程序的组织

测试应用程序是一个控制台应用程序内的*tests/RazorPagesTestSample.Tests*文件夹。

测试应用程序文件夹	描述
UnitTests	• *DataAccessLayerTest.cs* DAL 包含单元测试。 • *IndexPageTests.cs*包含针对索引页面模型的单元测试。
实用程序	包含 `TestingDbContextOptions` 用来创建新的数据库上下文选项为每个 DAL 单元测试，以便将数据库重置为其基线条件的每个测试方法。

测试框架这xUnit。模拟框架的对象是Moq。

单元测试的数据访问层 (DAL)

消息应用程序中包含的四个方法都具有 DAL `AppDbContext` 类 (*src/RazorPagesTestSample/Data/AppDbContext.cs*)。每个方法中测试应用程序具有一个或两个单元测试。

DAL 方法	函数
`GetMessagesAsync`	获取 `List<Message>` 从数据库按 `Text` 属性。
`AddMessageAsync`	添加 `Message` 到数据库。
`DeleteAllMessagesAsync`	将删除所有 `Message` 数据库中的条目。
`DeleteMessageAsync`	将删除单个 `Message` 从数据库 `Id` 。

需要的 DAL 单元测试DbContextOptions时创建一个新 `AppDbContext` 为每个测试。一种方法创建 `DbContextOptions` 每个测试是使用DbContextOptionsBuilder:

```
var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
    .UseInMemoryDatabase("InMemoryDb");

using (var db = new AppDbContext(optionsBuilder.Options))
{
    // Use the db here in the unit test.
}
```

此方法的问题是每个测试中的任何状态前次测试保留它接收的数据库。在尝试写入不会相互干扰的原子单元测试时，则可能存在问题。若要强制 `AppDbContext` 若要为每个测试中使用新的数据库上下文，提供 `DbContextOptions` 基于新的服务提供程序的实例。测试应用演示了如何执行此操作使用其 `Utilities` 类方法 `TestingDbContextOptions` (*tests/RazorPagesTestSample.Tests/Utilities/Utilities.cs*):

```
public static DbContextOptions<AppDbContext> TestDbContextOptions()
{
    // Create a new service provider to create a new in-memory database.
    var serviceProvider = new ServiceCollection()
        .AddEntityFrameworkInMemoryDatabase()
        .BuildServiceProvider();

    // Create a new options instance using an in-memory database and
    // IServiceProvider that the context should resolve all of its
    // services from.
    var builder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb")
        .UseInternalServiceProvider(serviceProvider);

    return builder.Options;
}
```

使用 `DbContextOptions` 接 DAL 单元测试，以使用新数据库实例以原子方式运行每个测试：

```
using (var db = new AppDbContext(Utilities.TestingDbContextOptions()))
{
    // Use the db here in the unit test.
}
```

在每个测试方法 `DataAccessLayerTest` 类 (*UnitTests/DataAccessLayerTest.cs*) 遵循类似的排列 Act 断言模式：

1. 排列：为测试配置了数据库和/或定义预期的结果。
2. Act：执行测试。
3. 断言：断言进行以确定测试结果是否成功完成。

例如，`DeleteMessageAsync` 方法负责删除一条消息由标识其 `Id` (*src/RazorPagesTestSample/Data/AppDbContext.cs*)：

```
public async virtual Task DeleteMessageAsync(int id)
{
    var message = await Messages.FindAsync(id);

    if (message != null)
    {
        Messages.Remove(message);
        await SaveChangesAsync();
    }
}
```

有两个测试此方法。一个测试检查数据库中存在消息时，将方法删除一条消息。如果不会更改数据库的其他方法测试消息 `Id` 为删除不存在。`DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound` 方法如下所示：

```
[Fact]
public async Task DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound()
{
    using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
    {
        // Arrange
        var seedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(seedMessages);
        await db.SaveChangesAsync();
        var recId = 1;
        var expectedMessages =
            seedMessages.Where(message => message.Id != recId).ToList();

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(x => x.Id),
            actualMessages.OrderBy(x => x.Id),
            new Utilities.MessageComparer());
    }
}
```

首先，该方法执行准备步骤中，执行步骤准备发生。获取和保存在种子设定消息 `seedMessages` 。种子设定消息保存到数据库中。与消息 `Id` 的 `1` 设置为删除。当 `DeleteMessageAsync` 执行方法时，预期的消息应具有的所有消息使用除 `Id` 的 `1` 。 `expectedMessages` 变量表示此预期的结果。

```
// Arrange
var seedMessages = AppDbContext.GetSeedingMessages();
await db.AddRangeAsync(seedMessages);
await db.SaveChangesAsync();
var recId = 1;
var expectedMessages =
    seedMessages.Where(message => message.Id != recId).ToList();
```

该方法的作用： `DeleteMessageAsync` 执行方法并传入 `recId` 的 `1` ：

```
// Act
await db.DeleteMessageAsync(recId);
```

最后，此方法获取 `Messages` 上下文中并将其到比较 `expectedMessages` 这两个相等的断言：

```
// Assert
var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

要比较的两个 `List<Message>` 相同：

- 消息按排序 `Id` 。
- 消息对比较上 `Text` 属性。

类似的测试方法， `DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound` 检查尝试删除不存在一条消息的结果。在这种情况下，在数据库中预期的消息应等于后的实际消息 `DeleteMessageAsync` 执行方法。应未更改数据库的内容：

```
[Fact]
public async Task DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound()
{
    using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
    {
        // Arrange
        var expectedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(expectedMessages);
        await db.SaveChangesAsync();
        var recId = 4;

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

页面模型方法的单元测试

另一组单元测试的测试的页面模型方法负责。在邮件应用中，索引页面模型中找到 `IndexModel` 类中*src/RazorPagesTestSample/Pages/Index.cshtml.cs*。

页面模型方法	函数
`OnGetAsync`	从 UI 使用 DAL 获取消息 `GetMessagesAsync` 方法。
`OnPostAddMessageAsync`	如果 `ModelState` 有效，调用 `AddMessageAsync` 向数据库添加一条消息。
`OnPostDeleteAllMessagesAsync`	调用 `DeleteAllMessagesAsync` 若要删除所有数据库中的消息。
`OnPostDeleteMessageAsync`	执行 `DeleteMessageAsync` 若要删除的消息 `Id` 指定。
`OnPostAnalyzeMessagesAsync`	如果一个或多个消息是在数据库中，将计算的平均每个消息的单词数。

使用七个测试中的测试页面模型方法 `IndexPageTests` 类(*tests/RazorPagesTestSample.Tests/UnitTests/IndexPageTests.cs*)。测试使用熟悉的排列断言 Act 模式。这些测试的重点：

- 确定是否方法遵循正确的行为时 `ModelState` 无效。
- 确认方法产生正确 `IActionResult` 。
- 正在检查正确进行属性值分配。

测试此组通常模拟 DAL 以生成预期的页面模型方法会执行步骤的数据的方法。例如， `GetMessagesAsync` 方法的 `AppDbContext` 模拟以生成输出。当页面模型方法执行此方法时，模拟将返回的结果。数据并不是来自该数据库。这将创建页面模型测试中使用 DAL 可预测、可靠的测试的条件。

`OnGetAsync_PopulatesThePageModel_WithAListOfMessages` 测试显示了如何将 `GetMessagesAsync` 方法模拟的页面模型：

```
var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
var expectedMessages = AppDbContext.GetSeedingMessages();
mockAppDbContext.Setup(
    db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
var pageModel = new IndexModel(mockAppDbContext.Object);
```

当 `OnGetAsync` Act 步骤中执行方法时，它调用的页面模型 `GetMessagesAsync` 方法。

单元测试执行步骤 (*tests/RazorPagesTestSample.Tests/UnitTests/IndexPageTests.cs*):

```
// Act
await pageModel.OnGetAsync();
```

`IndexPage` 页面模型 `OnGetAsync` 方法 (*src/RazorPagesTestSample/Pages/Index.cshtml.cs*):

```
public async Task OnGetAsync()
{
    Messages = await _db.GetMessagesAsync();
}
```

`GetMessagesAsync` DAL 中的方法不会返回此方法调用的结果。模拟的版本的方法返回的结果。

在中 `Assert` 步骤，实际消息 (`actualMessages`) 从分配的 `Messages` 的页面模型的属性。分配消息时，也执行类型检查。通过进行比较的预期和实际消息及其 `Text` 属性。测试断言，这两个 `List<Message>` 实例包含相同的消息。

```
// Assert
var actualMessages = Assert.IsAssignableFrom<List<Message>>(pageModel.Messages);
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

此组中的其他测试创建页包含的模型对象 `DefaultHttpContext` ，则 `ModelStateDictionary` 、 `ActionContext` 建立 `PageContext` 即 `ViewDataDictionary` ，和一个 `PageContext` 。这些可执行测试。例如，消息应用程序建立 `ModelState` 错误 `AddModelError` 检查是否是有效 `PageResult` 时，将返回 `OnPostAddMessageAsync` 执行：

```
[Fact]
public async Task OnPostAddMessageAsync_ReturnsAPageResult_WhenModelStateIsInvalid()
{
    // Arrange
    var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb");
    var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
    var expectedMessages = AppDbContext.GetSeedingMessages();
    mockAppDbContext.Setup(db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
    var httpContext = new DefaultHttpContext();
    var modelState = new ModelStateDictionary();
    var actionContext = new ActionContext(httpContext, new RouteData(), new PageActionDescriptor(),
modelState);
    var modelMetadataProvider = new EmptyModelMetadataProvider();
    var viewData = new ViewDataDictionary(modelMetadataProvider, modelState);
    var tempData = new TempDataDictionary(httpContext, Mock.Of<ITempDataProvider>());
    var pageContext = new PageContext(actionContext)
    {
        ViewData = viewData
    };
    var pageModel = new IndexModel(mockAppDbContext.Object)
    {
        PageContext = pageContext,
        TempData = tempData,
        Url = new UrlHelper(actionContext)
    };
    pageModel.ModelState.AddModelError("Message.Text", "The Text field is required.");

    // Act
    var result = await pageModel.OnPostAddMessageAsync();

    // Assert
    Assert.IsType<PageResult>(result);
}
```

其他资源

- 单元测试 C# 中使用 dotnet 测试和 xUnit 的.NET Core
- 测试控制器
- 单元测试代码(Visual Studio)
- 集成测试
- xUnit.net
- 入门 xUnit.net（.NET Core/ASP.NET 核）
- Moq
- Moq 快速入门

ASP.NET Core 中的测试控制器逻辑

2018/11/10 • • Edit Online

作者：Steve Smith

控制器在任何 ASP.NET Core MVC 应用中起着核心作用。因此，应该对控制器表现达到预期怀有信心。在将应用部署到生产环境之前，自动测试可以检测到错误。

查看或下载示例代码（如何下载）

控制器逻辑的单元测试

单元测试涉及通过基础结构和依赖项单独测试应用的一部分。单元测试控制器逻辑时，仅测试单个操作的内容，不测试其依赖项或框架自身的行为。

将控制器操作的单元测试设置为专注于控制器的行为。控制器单元测试将避开筛选器、路由或模型绑定等方案。涵盖共同响应请求的组件之间的交互的测试由*集成测试*处理。有关集成测试的详细信息，请参阅在 ASP.NET Core 中的集成测试。

如果编写自定义筛选器和路由，应对其单独进行单元测试，而不是在测试特定控制器操作时进行。

要演示控制器单元测试，请查看以下示例应用中的控制器。主控制器显示集体讨论会话列表并允许使用 POST 请求创建新的集体讨论会话：

```
public class HomeController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public HomeController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index()
    {
        var sessionList = await _sessionRepository.ListAsync();

        var model = sessionList.Select(session => new StormSessionViewModel()
        {
            Id = session.Id,
            DateCreated = session.DateCreated,
            Name = session.Name,
            IdeaCount = session.Ideas.Count
        });

        return View(model);
    }

    public class NewSessionModel
    {
        [Required]
        public string SessionName { get; set; }
    }

    [HttpPost]
    public async Task<IActionResult> Index(NewSessionModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        else
        {
            await _sessionRepository.AddAsync(new BrainstormSession()
            {
                DateCreated = DateTimeOffset.Now,
                Name = model.SessionName
            });
        }

        return RedirectToAction(actionName: nameof(Index));
    }
}
```

前面的控制器：

- 遵循显式依赖关系原则。
- 期望依赖关系注入 (DI) 提供 `IBrainstormSessionRepository` 的实例。
- 可以通过使用 mock 对象框架（如 Moq）的模拟 `IBrainstormSessionRepository` 服务进行测试。*模拟对象*是由一组预先确定的用于测试的属性和方法行为的对象。有关详细信息，请参阅集成测试简介。

`HTTP GET Index` 方法没有循环或分支，且仅调用一个方法。此操作的单元测试：

- 使用 `GetTestSessions` 方法模拟 `IBrainstormSessionRepository` 服务。`GetTestSessions` 使用日期和会话名称创建两个 mock 集体讨论会话。
- 执行 `Index` 方法。
- 根据该方法返回的结果进行断言：

- 将返回 ViewResult。
- ViewDataDictionary.Model 是 `StormSessionViewModel`。
- 有两个集体讨论会话存储在 `ViewDataDictionary.Model` 中。

```
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
        viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

```
private List<BrainstormSession> GetTestSessions()
{
    var sessions = new List<BrainstormSession>();
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    });
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 1),
        Id = 2,
        Name = "Test Two"
    });
    return sessions;
}
```

主控制器的 `HTTP POST Index` 方法测试验证：

- 当 ModelState.IsValid 为 `false` 时，操作方法将返回有相应数据的 *400 错误请求* ViewResult。
- 当 `ModelState.IsValid` 为 `true` 时：
  - 将调用存储库上的 `Add` 方法。
  - 将返回有正确参数的 RedirectToActionResult。

通过使用 AddModelError 添加错误来测试无效模型状态，如下第一个测试所示：

```
[Fact]
public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}
```

当 ModelState 无效时，将返回与 GET 请求相同的 `ViewResult`。测试不会尝试传入无效模型。传入无效模型不是有效的方法，因为不会运行模型绑定（尽管集成测试使用模型绑定）。在本例中，不测试模型绑定。这些单元测试仅测试操作方法中的代码。

第二个测试验证 `ModelState` 有效时的情况：

- 已通过存储库添加新的 `BrainstormSession`。
- 该方法将返回带有所需属性的 `RedirectToActionResult`。

通常会忽略未调用的模拟调用，但在设置调用末尾调用 `Verifiable` 就可以在测试中进行 mock 验证。这通过对 `mockRepo.Verify` 的调用来完成，进行这种调用时，如果未调用所需方法，则测试将失败。

> **NOTE**
>
> 通过此示例中使用的 Moq 库，可以混合可验证（或称“严格”）mock 和非可验证 mock（也称为“宽松”mock 或存根）。详细了解使用 Moq 自定义 Mock 行为。

示例应用中的 SessionController 显示与特定集体讨论会话相关的信息。该控制器包含用于处理无效 `id` 值的逻辑（以下示例中有两个 `return` 方案可用来应对这些情况）。最后的 `return` 语句向视图

(Controllers/SessionController.cs) 返回一个新的 `StormSessionViewModel`：

```
public class SessionController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public SessionController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index(int? id)
    {
        if (!id.HasValue)
        {
            return RedirectToAction(actionName: nameof(Index),
                controllerName: "Home");
        }

        var session = await _sessionRepository.GetByIdAsync(id.Value);
        if (session == null)
        {
            return Content("Session not found.");
        }

        var viewModel = new StormSessionViewModel()
        {
            DateCreated = session.DateCreated,
            Name = session.Name,
            Id = session.Id
        };

        return View(viewModel);
    }
}
```

单元测试包括对会话控制器 `Index` 操作中的每个 `return` 方案执行一个测试：

```
[Fact]
public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
{
    // Arrange
    var controller = new SessionController(sessionRepository: null);

    // Act
    var result = await controller.Index(id: null);

    // Assert
    var redirectToActionResult =
        Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
}

[Fact]
public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var contentResult = Assert.IsType<ContentResult>(result);
    Assert.Equal("Session not found.", contentResult.Content);
}

[Fact]
public async Task IndexReturnsViewResultWithStormSessionViewModel()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSessions().FirstOrDefault(
            s => s.Id == testSessionId));
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsType<StormSessionViewModel>(
        viewResult.ViewData.Model);
    Assert.Equal("Test One", model.Name);
    Assert.Equal(2, model.DateCreated.Day);
    Assert.Equal(testSessionId, model.Id);
}
```

移动到想法控制器，应用会将功能公开为 `api/ideas` 路由上的 Web API：

- `ForSession` 方法将返回与集体讨论会话关联的想法列表（`IdeaDTO`）。
- `Create` 方法会向会话中添加新想法。

```
[HttpGet("forsession/{sessionId}")]
public async Task<IActionResult> ForSession(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);
    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return Ok(result);
}

[HttpPost("create")]
public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);
    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return Ok(session);
}
```

避免直接通过 API 调用返回企业域实体。域实体：

- 包含的数据通常比客户端所需的数据更多。
- 无需将应用的内部域模型与公开的 API 结合。

可以执行域实体与返回到客户端的类型之间的映射：

- 手动执行 LINQ `Select`，如同示例应用所用的那样。有关详细信息，请参阅 LINQ（语言集成查询）。
- 自动生成库，如 AutoMapper。

接着，示例应用会演示想法控制器的 `Create` 和 `ForSession` API 方法的单元测试。

示例应用包含两个 `ForSession` 测试。第一个测试可确定 `ForSession` 是否返回无效会话的 NotFoundObjectResult（找不到 HTTP）：

```
[Fact]
public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
    Assert.Equal(testSessionId, notFoundObjectResult.Value);
}
```

第二个 `ForSession` 测试可确定 `ForSession` 是否返回有效会话的会话想法列表（`<List<IdeaDTO>>`）。这些测试还会检查第一个想法，以确认其 `Name` 属性正确：

```
[Fact]
public async Task ForSession_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

若要测试 `Create` 方法在 `ModelState` 无效时的行为，示例应用会在测试中将模型错误添加到控制器。请勿在单元测试中尝试测试模型有效性或模型绑定—仅测试操作方法在遇到无效 `ModelState` 时的行为：

```
[Fact]
public async Task Create_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.Create(model: null);

    // Assert
    Assert.IsType<BadRequestObjectResult>(result);
}
```

`Create` 的第二个测试依赖存储库返回 `null`，所以 mock 存储库配置为返回 `null`。无需创建测试数据库（在内存中或其他位置）并构建将返回此结果的查询。该测试可以在单个语句中完成，如示例代码所示：

```
[Fact]
public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.Create(new NewIdeaModel());

    // Assert
    Assert.IsType<NotFoundObjectResult>(result);
}
```

第三个 `Create` 测试 `Create_ReturnsNewlyCreatedIdeaForSession` 验证调用了存储库的 `UpdateAsync` 方法。使用 `Verifiable` 调用 mock，然后调用模拟存储库的 `Verify` 方法，以确认执行了可验证的方法。确保 `UpdateAsync` 方法保存了数据不是单元测试的职责—这可以通过集成测试完成。

```
[Fact]
public async Task Create_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.Create(newIdea);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnSession.Ideas.Count());
    Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
}
```

测试 ActionResult<T>

在 ASP.NET Core 2.1 或更高版本中，ActionResult<T> (ActionResult<TValue>) 支持返回从 `ActionResult` 派生的类型或返回特定类型。

示例应用包含将返回给定会话 `id` 的 `List<IdeaDTO>` 的方法。如果会话 `id` 不存在，控制器将返回 NotFound：

```
[HttpGet("forsessionactionresult/{sessionId}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<ActionResult<List<IdeaDTO>>> ForSessionActionResult(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);

    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return result;
}
```

`ApiIdeasControllerTests` 中包含 `ForSessionActionResult` 控制器的两个测试。

第一个测试可确认控制器将返回 `ActionResult`，而不是不存在会话 `id` 的不存在想法列表：

- `ActionResult` 类型为 `ActionResult<List<IdeaDTO>>`。
- Result 为 NotFoundObjectResult。

```
[Fact]
public async Task ForSessionActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    var nonExistentSessionId = 999;

    // Act
    var result = await controller.ForSessionActionResult(nonExistentSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

对于有效会话 `id`，第二个测试可确认该方法将返回：

- 类型为 `List<IdeaDTO>` 的 `ActionResult`。
- ActionResult<T>.Value 的类型为 `List<IdeaDTO>`。
- 列表中的第一项是与 mock 会话中存储的想法匹配的有效想法（通过调用 `GetTestSession` 获取）。

```
[Fact]
public async Task ForSessionActionResult_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSessionActionResult(testSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(actionResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

示例应用还包含用于为给定会话创建新的 `Idea` 的方法。控制器将返回：

- BadRequest（对于无效模型）。
- NotFound（如果会话不存在）。
- CreatedAtAction（当使用新想法更新会话时）。

```
[HttpPost("createactionresult")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
public async Task<ActionResult<BrainstormSession>> CreateActionResult([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);

    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return CreatedAtAction(nameof(CreateActionResult), new { id = session.Id }, session);
}
```

`ApiIdeasControllerTests` 中包含 `CreateActionResult` 的三个测试。

第一个测试可确认将返回 BadRequest（对于无效模型）。

```
[Fact]
public async Task CreateActionResult_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.CreateActionResult(model: null);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<BadRequestObjectResult>(actionResult.Result);
}
```

第二个测试可确认将返回 NotFound（如果会话不存在）。

```
[Fact]
public async Task CreateActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var nonExistentSessionId = 999;
    string testName = "test name";
    string testDescription = "test description";
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = nonExistentSessionId
    };

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

对于有效会话 `id`，最后一个测试可确认：

- 该方法将返回类型为 `BrainstormSession` 的 `ActionResult`。
- ActionResult<T>.Result 为 CreatedAtActionResult。`CreatedAtActionResult` 类似于包含 `Location` 标头的 *201 Created* 响应。
- ActionResult<T>.Value 的类型为 `BrainstormSession`。
- 调用了用于更新会话 `UpdateAsync(testSession)` 的 mock 调用。通过执行断言中的 `mockRepo.Verify()` 来检查 `Verifiable` 方法调用。
- 将返回该会话的两个 `Idea` 对象。
- 最后一项（通过对 `UpdateAsync` 的 mock 调用而添加的 `Idea`）与添加到测试中的会话的 `newIdea` 匹配。

```
[Fact]
public async Task CreateActionResult_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    var createdAtActionResult = Assert.IsType<CreatedAtActionResult>(actionResult.Result);
    var returnValue = Assert.IsType<BrainstormSession>(createdAtActionResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnValue.Ideas.Count());
    Assert.Equal(testName, returnValue.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnValue.Ideas.LastOrDefault().Description);
}
```

其他资源

- 在 ASP.NET Core 中的集成测试
- 使用 Visual Studio 创建和运行单位测试。
- Explicit Dependencies Principle（显式依赖关系原则）

在 ASP.NET Core 中的集成测试

2019/1/12 • • Edit Online

通过Luke Latham和Steve Smith

集成测试可确保应用程序的组件正常工作，包括应用程序的支持的基础结构，如数据库、文件系统和网络级别。ASP.NET Core 支持集成测试的测试 web 主机和内存中测试服务器中使用单元测试框架。

本主题假定你基本了解单元测试。如果测试概念不太熟悉，请参阅.NET Core 和.NET Standard 中的单元测试主题和其链接的内容。

查看或下载示例代码（如何下载）

示例应用是 Razor 页面应用，并假定你基本了解 Razor 页面。如果不熟悉使用 Razor 页面，请参阅以下主题：

- Razor 页面介绍
- Razor 页面入门
- Razor 页面单元测试

> **NOTE**
>
> 用于测试 Spa，我们建议使用一种工具如Selenium，这可以自动执行浏览器。

集成测试简介

集成测试评估上水平比更为广泛的应用程序的组件单元测试。单元测试用于测试独立的软件组件，例如单独的类的方法。集成测试确认，两个或多个应用程序组件的协作产生预期的结果，可能包括将完全处理一个请求所需的每个组件。

这些更广泛的测试用于测试应用程序的基础结构和整个 framework 中，通常包括以下组件：

- 数据库
- 文件系统
- 网络设备
- 请求-响应管道

单元测试使用制造的组件，称为*fakes*或*mock 对象*，代替基础结构组件。

与单元测试，集成测试：

- 使用该应用在生产环境中使用的实际组件。
- 需要更多代码和数据处理。
- 需要更长时间运行。

因此，限制为最重要的基础结构方案的集成测试使用。如果使用单元测试或集成测试，可以测试行为，请选择单元测试。

> **TIP**
> 不编写与数据库和文件系统的数据和文件访问的每个可能的排列的集成测试。无论多少位跨应用程序与数据库和文件系统、已设定焦点的一组读取、写入、更新和删除集成测试通常能够充分地测试数据库和文件系统组件进行交互。使用单元测试的与这些组件进行交互的例程测试的方法逻辑。在单元测试基础结构使用虚设/模拟测试执行速度更快的结果。

> **NOTE**
> 在讨论的集成测试时，通常称为测试的项目*待测试系统*，或简称"SUT"。

ASP.NET Core 集成测试

在 ASP.NET Core 中的集成测试需要以下项：

- 测试项目用于包含和执行测试。测试项目具有对名为经过测试的 ASP.NET Core 项目的引用*待测试系统*(SUT)。*在本主题中使用"SUT"来指代经过测试的应用。*
- 测试项目创建 SUT 的测试 web 主机，并使用测试服务器客户端处理请求和对 SUT 的响应。
- 测试运行程序用于执行测试和报告测试结果。

集成测试遵循一系列事件，其中包括常用*排列*，*Act*，并*Assert*测试步骤：

1. SUT 的 web 主机配置。
2. 创建测试服务器客户端以将请求提交到应用程序。
3. *排列*执行测试步骤：测试应用程序准备请求。
4. *Act*执行测试步骤：客户端提交该请求并接收响应。
5. *Assert*执行测试步骤：*实际*响应中进行验证*传递*或*失败*基于*预期*响应。
6. 过程持续进行的所有测试执行。
7. 报告测试结果。

通常情况下，测试 web 主机配置以不同的方式不是测试应用程序的普通的 web 主机在运行。例如，测试可能使用不同的数据库或不同的应用设置。

基础结构组件，如测试 web 主机和内存中测试服务器 (TestServer)、提供或由托管Microsoft.AspNetCore.Mvc.Testing包。使用此包的简化了测试创建和执行。

`Microsoft.AspNetCore.Mvc.Testing` 程序包处理以下任务：

- 将复制依赖项文件 (*.*deps*) 到测试项目的 sut *bin*文件夹。
- 内容根设置为 SUT 的项目根，使静态文件和网页/视图时执行的测试发现。
- 提供了WebApplicationFactory类，以简化启动与 SUT `TestServer`。

单元测试文档介绍了如何设置测试项目和测试运行程序以及如何运行测试和的建议的名称测试和测试类的详细说明。

> **NOTE**
> 在创建应用程序的测试项目时，分离到不同的项目集成测试中的单元测试。这有助于确保基础结构的测试组件不意外包含在单元测试。单元测试和集成测试的分离还允许控制对哪些组的测试运行。

几乎的 Razor 页面应用程序测试的配置和 MVC 应用程序之间没有差异。唯一区别是在测试的命名方式。在 Razor 页面应用中，测试页终结点的名称通常以在页面模型类后 (例如，`IndexPageTests` 测试索引页的组件集成)。在 MVC 应用中，测试是通常按控制器类和命名这些测试的控制器 (例如，`HomeControllerTests` 测试 Home 控制器的组件集成)。

测试应用程序必备组件

测试项目必须：

- 引用以下包：
  - Microsoft.AspNetCore.App
  - Microsoft.AspNetCore.Mvc.Testing
- 在项目文件中指定 Web SDK（`<Project Sdk="Microsoft.NET.Sdk.Web">`）。Web SDK 时是必需的引用Microsoft.AspNetCore.App 元包。

这些系统必备组件中所示示例应用。检查*tests/RazorPagesProject.Tests/RazorPagesProject.Tests.csproj*文件。示例应用使用xUnit测试框架和AngleSharp分析器库，因此示例应用还引用：

- xunit
- xunit.runner.visualstudio
- AngleSharp

默认值 WebApplicationFactory 基本测试

WebApplicationFactory<TEntryPoint> 用来创建TestServer集成测试。`TEntryPoint` 通常是 SUT 的入口点类 `Startup` 类。

测试类将实现*类装置*接口 (IClassFixture) 表明类包含测试以及在测试类中提供共享的对象实例。

基本应用程序终结点的测试

以下测试类，`BasicTests`，使用 `WebApplicationFactory` 若要启动 SUT 和提供HttpClient到测试方法、`Get_EndpointsReturnSuccessAndCorrectContentType`。该方法检查是否成功的响应状态代码（状态代码在 200-299 范围）和 `Content-Type` 标头是 `text/html; charset=utf-8` 若干应用程序页。

CreateClient创建的实例 `HttpClient` 的自动遵循重定向，并处理 cookie。

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly WebApplicationFactory<RazorPagesProject.Startup> _factory;

    public BasicTests(WebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _factory = factory;
    }

    [Theory]
    [InlineData("/")]
    [InlineData("/Index")]
    [InlineData("/About")]
    [InlineData("/Privacy")]
    [InlineData("/Contact")]
    public async Task Get_EndpointsReturnSuccessAndCorrectContentType(string url)
    {
        // Arrange
        var client = _factory.CreateClient();

        // Act
        var response = await client.GetAsync(url);

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
```

测试安全终结点

中的另一个测试 `BasicTests` 类将检查安全终结点到应用程序的登录页将未经身份验证的用户重定向。

在 SUT 中 `/SecurePage` 页上使用AuthorizePage约定应用AuthorizeFilter到页。有关详细信息，请参阅Razor 页面授权约定。

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/SecurePage");
    });
```

在中 `Get_SecurePageRequiresAnAuthenticatedUser` 测试，请WebApplicationFactoryClientOptions设置为不重定向允许通过设置AllowAutoRedirect到 `false`：

```
[Fact]
public async Task Get_SecurePageRequiresAnAuthenticatedUser()
{
    // Arrange
    var client = _factory.CreateClient(
        new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });

    // Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.StartsWith("http://localhost/Identity/Account/Login",
        response.Headers.Location.OriginalString);
}
```

通过禁止客户端遵循重定向，可以进行以下检查：

- 可以检查返回的 SUT 的状态代码针对预期HttpStatusCode.Redirect结果中，不是最终状态代码后将重定向到登录页面，将是HttpStatusCode.OK.
- `Location` 响应标头中的标头值进行检查以确认它开头 `http://localhost/Identity/Account/Login`，不将最终的登录页响应，其中 `Location` 标头不会为存在。

有关详细信息 `WebApplicationFactoryClientOptions`，请参阅客户端选项部分。

自定义 WebApplicationFactory

Web 主机配置可以通过继承创建独立的测试类于 `WebApplicationFactory` 来创建一个或多个自定义工厂：

1. 继承自 `WebApplicationFactory` 并重写ConfigureWebHost。IWebHostBuilder允许使用的服务集合的配置ConfigureServices:

```
public class CustomWebApplicationFactory<TStartup>
    : WebApplicationFactory<TStartup> where TStartup: class
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            // Create a new service provider.
            var serviceProvider = new ServiceCollection()
                .AddEntityFrameworkInMemoryDatabase()
                .BuildServiceProvider();

            // Add a database context (ApplicationDbContext) using an in-memory
            // database for testing.
            services.AddDbContext<ApplicationDbContext>(options =>
            {
                options.UseInMemoryDatabase("InMemoryDbForTesting");
                options.UseInternalServiceProvider(serviceProvider);
            });

            // Build the service provider.
            var sp = services.BuildServiceProvider();

            // Create a scope to obtain a reference to the database
            // context (ApplicationDbContext).
            using (var scope = sp.CreateScope())
            {
                var scopedServices = scope.ServiceProvider;
                var db = scopedServices.GetRequiredService<ApplicationDbContext>();
                var logger = scopedServices
                    .GetRequiredService<ILogger<CustomWebApplicationFactory<TStartup>>>();

                // Ensure the database is created.
                db.Database.EnsureCreated();

                try
                {
                    // Seed the database with test data.
                    Utilities.InitializeDbForTests(db);
                }
                catch (Exception ex)
                {
                    logger.LogError(ex, $"An error occurred seeding the " +
                        "database with test messages. Error: {ex.Message}");
                }
            }
        });
    }
}
```

数据库在种子设定示例应用由执行 `InitializeDbForTests` 方法。中介绍了方法集成测试示例:测试应用程序的组织部分。

2. 使用自定义 `CustomWebApplicationFactory` 测试类中。下面的示例使用中的工厂 `IndexPageTests` 类:

```
public class IndexPageTests :
    IClassFixture<CustomWebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly HttpClient _client;
    private readonly CustomWebApplicationFactory<RazorPagesProject.Startup>
        _factory;

    public IndexPageTests(
        CustomWebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _factory = factory;
        _client = factory.CreateClient(new WebApplicationFactoryClientOptions
            {
                AllowAutoRedirect = false
            });
    }
```

示例应用程序的客户端配置为阻止 `HttpClient` 从以下重定向。中所述测试安全终结点部分中，这允许测试以检查应用程序的第一个响应的结果。第一个响应是许多与这些测试中的重定向 `Location` 标头。

3. 典型测试中使用 `HttpClient` 和帮助器方法来处理请求和响应：

```
[Fact]
public async Task Post_DeleteAllMessagesHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var defaultPage = await _client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await _client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),
        (IHtmlButtonElement)content.QuerySelector("button[id='deleteAllBtn']"));

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

任何对 SUT 的 POST 请求必须满足应用的自动所做的防伪复选的数据保护防伪系统。若要安排某个测试的 POST 请求，测试应用程序必须：

1. 发出页请求。
2. 分析防伪 cookie 和请求验证令牌响应中。
3. 在位置进行 POST 请求，其中防伪 cookie 和请求验证令牌。

`SendAsync` Helper 扩展方法 (*Helpers/HttpClientExtensions.cs*) 和 `GetDocumentAsync` 帮助器方法 (*Helpers/HtmlHelpers.cs*) 中的示例应用使用AngleSharp分析器以使用以下方法处理防伪检查：

- `GetDocumentAsync` – 接收HttpResponseMessage，并返回 `IHtmlDocument`。`GetDocumentAsync` 使用工厂，用于准备*虚拟响应*根据原始 `HttpResponseMessage`。有关详细信息，请参阅AngleSharp 文档。
- `SendAsync` 扩展方法 `HttpClient` compose HttpRequestMessage，并调用SendAsync(HttpRequestMessage)以将请求提交到 SUT。有关重载 `SendAsync` 接受 HTML 窗体 (`IHtmlFormElement`) 和以下：
  - 提交窗体的按钮 (`IHtmlElement`)
  - 窗体值集合 (`IEnumerable<KeyValuePair<string, string>>`)
  - 提交按钮 (`IHtmlElement`) 和窗体值 (`IEnumerable<KeyValuePair<string, string>>`)

> **NOTE**
>
> AngleSharp第三方分析用于演示目的，在本主题和示例应用程序中的库。AngleSharp 不受支持或所需的集成测试的 ASP.NET Core 应用。其他分析器可以使用，如Html 灵活性包 (HAP)。另一种方法是编写代码来直接处理防伪系统的请求验证令牌和防伪 cookie。

自定义客户端与 WithWebHostBuilder

在测试方法中，需要额外的配置时WithWebHostBuilder创建一个新 `WebApplicationFactory` 与IWebHostBuilder的配置通过进一步自定义。

`Post_DeleteMessageHandler_ReturnsRedirectToRoot` 测试的方法示例应用程序演示如何使用 `WithWebHostBuilder` 。此测试可执行记录删除数据库中通过触发在 SUT 中的窗体提交。

因为另一个测试中 `IndexPageTests` 类执行的操作将删除所有数据库中的记录，并且可能运行之前 `Post_DeleteMessageHandler_ReturnsRedirectToRoot` 方法中，数据库将植入以确保记录存在的 SUT，若要删除此测试方法中。选择 `deleteBtn1` 按钮的 `messages` 窗体在 SUT 中的模拟对 SUT 的请求中：

```
[Fact]
public async Task Post_DeleteMessageHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureServices(services =>
            {
                var serviceProvider = services.BuildServiceProvider();

                using (var scope = serviceProvider.CreateScope())
                {
                    var scopedServices = scope.ServiceProvider;
                    var db = scopedServices
                        .GetRequiredService<ApplicationDbContext>();
                    var logger = scopedServices
                        .GetRequiredService<ILogger<IndexPageTests>>();

                    try
                    {
                        Utilities.InitializeDbForTests(db);
                    }
                    catch (Exception ex)
                    {
                        logger.LogError(ex, "An error occurred seeding " +
                            "the database with test messages. Error: " +
                            ex.Message);
                    }
                }
            });
        })
        .CreateClient(new WebApplicationFactoryClientOptions
            {
                AllowAutoRedirect = false
            });
    var defaultPage = await client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),
        (IHtmlButtonElement)content.QuerySelector("button[id='deleteBtn1']"));

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

客户端选项

下表显示的默认WebApplicationFactoryClientOptions时创建可用 `HttpClient` 实例。

选项	描述	默认
AllowAutoRedirect	获取或设置是否 `HttpClient` 实例应自动跟随重定向响应。	`true`
BaseAddress	获取或设置的基址 `HttpClient` 实例。	`http://localhost`
HandleCookies	获取或设置是否 `HttpClient` 实例应处理 cookie。	`true`

选项	描述	默认
MaxAutomaticRedirections	获取或设置重定向响应的最大数目 `HttpClient` 实例应遵循。	7

创建 `WebApplicationFactoryClientOptions` 类，并将其传递给CreateClient方法（默认值代码示例所示）：

```
// Default client option values are shown
var clientOptions = new WebApplicationFactoryClientOptions();
clientOptions.AllowAutoRedirect = true;
clientOptions.BaseAddress = new Uri("http://localhost");
clientOptions.HandleCookies = true;
clientOptions.MaxAutomaticRedirections = 7;

_client = _factory.CreateClient(clientOptions);
```

插入模拟服务

服务可以通过调用测试中重写ConfigureTestServices主机生成器上。若要插入的模拟服务，SUT 必须具有 `Startup` 类的 `Startup.ConfigureServices` 方法。

示例 SUT 包含作用域内返回的服务的引号。请求索引页时，在索引页的隐藏字段中嵌入引号。

Services/IQuoteService.cs:

```
public interface IQuoteService
{
    Task<string> GenerateQuote();
}
```

Services/QuoteService.cs:

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Dr. Who: Planet of Evil
// https://www.bbc.co.uk/programmes/p00pyrx6
public class QuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult<string>(
            "Come on, Sarah. We've an appointment in London, " +
            "and we're already 30,000 years late.");
    }
}
```

Startup.cs：

```
services.AddScoped<IQuoteService, QuoteService>();
```

Pages/Index.cshtml.cs：

```
public class IndexModel : PageModel
{
    private readonly ApplicationDbContext _db;
    private readonly IQuoteService _quoteService;

    public IndexModel(ApplicationDbContext db, IQuoteService quoteService)
    {
        _db = db;
        _quoteService = quoteService;
    }

    [BindProperty]
    public Message Message { get; set; }

    public IList<Message> Messages { get; private set; }

    [TempData]
    public string MessageAnalysisResult { get; set; }

    public string Quote { get; private set; }

    public async Task OnGetAsync()
    {
        Messages = await _db.GetMessagesAsync();

        Quote = await _quoteService.GenerateQuote();
    }
```

Pages/Index.cs:

```
<input id="quote" type="hidden" value="@Model.Quote">
```

当 SUT 应用运行时，会生成以下标记：

```
<input id="quote" type="hidden" value="Come on, Sarah. We&#x27;ve an appointment in
    London, and we&#x27;re already 30,000 years late.">
```

若要在集成测试中测试服务和引号注入，模拟服务注入到 SUT 的测试。模拟服务替代了应用程序的 `QuoteService` 测试应用程序提供的服务，名为 `TestQuoteService`：

IntegrationTests.IndexPageTests.cs:

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Pyramids of Mars
// https://www.bbc.co.uk/programmes/p00pys55
public class TestQuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult<string>(
            "Something's interfering with time, Mr. Scarman, " +
            "and time is my business.");
    }
}
```

`ConfigureTestServices` 调用时，并指定了作用域的服务将注册：

```
[Fact]
public async Task Get_QuoteService_ProvidesQuoteInPage()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddScoped<IQuoteService, TestQuoteService>();
            });
        })
        .CreateClient();

    //Act
    var defaultPage = await client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);
    var quoteElement = content.QuerySelector("#quote");

    // Assert
    Assert.Equal("Something's interfering with time, Mr. Scarman, " +
        "and time is my business.", quoteElement.Attributes["value"].Value);
}
```

在执行测试的执行期间生成的标记反映了由提供的报价文本 `TestQuoteService`，因此在断言阶段：

```
<input id="quote" type="hidden" value="Something&#x27;s interfering with time,
    Mr. Scarman, and time is my business.">
```

测试基础结构如何推断应用内容根路径

`WebApplicationFactory` 构造函数通过搜索来推断应用内容根路径WebApplicationFactoryContentRootAttribute包含其键等于集成测试的程序集上 `TEntryPoint` 的程序集 `System.Reflection.Assembly.FullName`. 如果找不到具有正确的密钥的属性，`WebApplicationFactory` 回退到搜索解决方案文件 (*.sln)，并追加 `TEntryPoint` 到解决方案目录的程序集名称。应用程序根目录下（内容根路径）用于发现视图和内容文件。

在大多数情况下，无需显式设置应用程序内容根，在搜索逻辑通常在运行时找到正确的内容根。内容根位置找不到的特殊方案中使用内置的搜索算法，内容可以指定根目录，显式或使用自定义逻辑应用。若要将应用内容根目录设置在这些情况下，调用 `UseSolutionRelativeContentRoot` 扩展方法来源Microsoft.AspNetCore.TestHost包。提供解决方案的相对路径和可选的解决方案文件的名称或 glob 模式 (默认值 = `*.sln`)。

调用UseSolutionRelativeContentRoot扩展方法使用一个以下方法之一：

- 配置与测试类时 `WebApplicationFactory`，提供的自定义配置IWebHostBuilder:

  ```
  public IndexPageTests(
      WebApplicationFactory<RazorPagesProject.Startup> factory)
  {
      var _factory = factory.WithWebHostBuilder(builder =>
      {
          builder.UseSolutionRelativeContentRoot("<SOLUTION-RELATIVE-PATH>");

          ...
      });
  }
  ```

- 使用自定义配置测试类时 `WebApplicationFactory`，继承自 `WebApplicationFactory` 并重

写ConfigureWebHost:

```
public class CustomWebApplicationFactory<TStartup>
     : WebApplicationFactory<RazorPagesProject.Startup>
{
     protected override void ConfigureWebHost(IWebHostBuilder builder)
     {
         builder.ConfigureServices(services =>
         {
             builder.UseSolutionRelativeContentRoot("<SOLUTION-RELATIVE-PATH>");

             ...
         });
     }
}
```

禁用卷影复制

卷影复制会导致要在不同的文件夹的输出文件夹中执行的测试。对于测试才能正常工作，卷影复制必须禁用。示例应用使用 xUnit 和禁用卷影复制对于 xUnit 通过包括*xunit.runner.json*文件，并正确的配置设置。有关详细信息，请参阅使用 JSON 配置 xUnit.net。

添加*xunit.runner.json*到测试项目包含以下内容的根目录的文件：

```
{
  "shadowCopy": false
}
```

可供使用的对象

之后的测试`IClassFixture`实现会执行，TestServer并HttpClient xUnit 释放时，将释放WebApplicationFactory. 如果对象实例化由开发人员需要可供使用，释放它们在`IClassFixture`实现。有关详细信息，请参阅实现 Dispose 方法。

集成测试示例

示例应用由两个应用的组成：

应用	项目文件夹	描述
消息应用程序 (SUT)	*src/RazorPagesProject*	允许用户添加、删除其中一个，删除所有，并分析消息。
测试应用	*tests/RazorPagesProject.Tests*	用于集成测试 SUT。

可以使用内置测试功能的 IDE，如运行测试Visual Studio。如果使用Visual Studio Code或命令行中，执行以下命令在命令提示符处*tests/RazorPagesProject.Tests*文件夹：

```
dotnet test
```

消息应用 (SUT) 组织

SUT 是 Razor 页面消息系统具有以下特征：

- 应用程序的索引页 (*pages/Index.cshtml*并*Pages/Index.cshtml.cs*) 提供 UI 和页面模型方法，用于控制添加、删除和分析的消息（每个消息的平均词）.

- 一条消息由描述 `Message` 类 (*Data/Message.cs*) 具有两个属性：`Id`（密钥）和 `Text`（消息）。`Text` 属性是必需的限制为 200 个字符。
- 使用存储的消息Entity Framework 的内存中数据库†。
- 应用程序包含在其数据库上下文类中，数据访问层 (DAL) `AppDbContext` (*Data/AppDbContext.cs*)。
- 如果数据库为空应用程序启动时，使用三个消息初始化消息存储区。
- 该应用包含 `/SecurePage` 仅可通过身份验证的用户访问。

†EF 主题测试与 InMemory，说明如何使用内存中数据库的使用 MSTest 的测试。本主题使用xUnit测试框架。测试概念和跨不同测试框架的测试实现有类似，但不是完全相同。

虽然应用不使用存储库模式，并且不是有效的示例工作单元 (UoW) 模式，Razor 页面支持开发这些模式。有关详细信息，请参阅设计基础结构持久性层并测试控制器逻辑（此示例实现存储库模式）。

测试应用程序的组织

测试应用程序是一个控制台应用程序内的*tests/RazorPagesProject.Tests*文件夹。

测试应用程序文件夹	描述
BasicTests	*BasicTests.cs*包含测试方法的路由、访问的安全页由未经身份验证的用户，并获取 GitHub 用户配置文件和检查配置文件的用户登录名。
IntegrationTests	*IndexPageTests.cs*包含使用自定义的索引页的集成测试 `WebApplicationFactory` 类。
帮助程序/实用程序	• *Utilities.cs*包含 `InitializeDbForTests` 方法用于设置测试数据与数据库的种子。 • *HtmlHelpers.cs*提供了一个方法来返回 AngleSharp `IHtmlDocument` 以供测试方法。 • *HttpClientExtensions.cs*提供的重载 `SendAsync` 以将请求提交到 SUT。

测试框架这xUnit。使用进行集成测试Microsoft.AspNetCore.TestHost，其中包括TestServer。因为Microsoft.AspNetCore.Mvc.Testing包用于配置测试主机和测试服务器 `TestHost` 和 `TestServer` 的包不需要测试应用的项目文件中的直接包引用或在测试应用程序的开发人员配置。

用于测试数据库进行种子设定

集成测试通常需要在测试执行之前在数据库中的一个小型数据集。例如，删除测试的数据库记录删除的调用，因此数据库必须具有至少一个记录删除请求才能成功。

示例应用程序中的三个消息数据库种子*Utilities.cs*测试可以使用它们执行时：

```
public static void InitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.AddRange(GetSeedingMessages());
    db.SaveChanges();
}

public static List<Message> GetSeedingMessages()
{
    return new List<Message>()
    {
        new Message(){ Text = "TEST RECORD: You're standing on my scarf." },
        new Message(){ Text = "TEST RECORD: Would you like a jelly baby?" },
        new Message(){ Text = "TEST RECORD: To the rational mind, " +
            "nothing is inexplicable; only unexplained." }
    };
}
```

其他资源

- 单元测试
- Razor 页面单元测试
- 中间件
- 测试控制器

负载测试和压力测试 ASP.NET Core

2019/1/11 • • Edit Online

负载测试和压力测试是非常重要的 web 应用的性能和可缩放。其目标是不同甚至它们通常共享类似测试。

负载测试:测试是否应用可以处理表示特定的场景中的用户指定的负载，同时仍能满足响应目标。在正常情况下运行该应用。

压力测试:测试应用程序稳定性下极端条件和通常长时间运行时：

- 较高用户负载 – 峰值或逐渐增加。
- 有限的计算资源。

在负载下，可以应用从故障中恢复并适当地返回到预期的行为？ 应用程序是在压力下不在正常情况下运行。

Visual Studio Tools

Visual Studio 允许用户创建、开发和调试 web 性能和负载测试。一个选项，可通过 web 浏览器中录制操作创建测试。

快速入门：创建负载测试项目演示如何创建、配置和运行负载测试项目使用 Visual Studio 2017。

有关详细信息，请参阅其他资源。

可以配置负载测试运行中的本地或在云中使用 Azure DevOps 运行。

Azure DevOps

可以使用启动负载测试运行Azure DevOps 测试计划服务。

该服务支持以下类型的测试格式：

- Visual Studio 测试 – Visual Studio 中创建的 web 测试。
- 在测试期间重播基于 HTTP 存档的测试 – 存档内捕获的 HTTP 流量。
- 基于 URL 的测试– 允许指定的 Url 来加载测试、请求类型、标头和查询字符串。运行设置参数，如持续时间，可以配置负载模式，用户等。
- Apache JMeter测试。

Azure 门户

Azure 门户允许设置和运行负载测试的 Web 应用，直接从 Azure 门户中的应用服务的性能选项卡。

测试可以是包含指定的 URL 或可以测试多个 Url 的 Visual Studio Web 测试文件的手动测试。

在测试结束时，会生成报表来显示该应用程序的性能特征。示例统计信息包括：

- 平均响应时间
- 最大吞吐量：每秒请求数
- 失败百分比

第三方工具

以下列表包含各种功能集与第三方 web 性能工具：

- Apache JMeter :负载测试工具的完整特色的套件。线程绑定：需要每个用户的一个线程。
- ab 的基准测试工具的 Apache HTTP 服务器

- Gatling :使用 GUI 和测试的刻录机的桌面工具。JMeter 性能更强。
- Locust.io :不受线程。

其他资源

加载测试博客系列作者：Charles Sterling。日期，但大多数主题都仍适用。

解决 ASP.NET Core 项目

2018/11/28 • • Edit Online

作者：Rick Anderson

以下链接提供的故障排除指南：

- 对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除
- 对 IIS 上的 ASP.NET Core 进行故障排除
- Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考
- ASP.NET Core 应用程序中（ndc）会议（伦敦，2018年）：诊断问题
- ASP.NET 博客：解决 ASP.NET Core 性能问题

.NET Core SDK 警告

安装的 32 位和 64 位版本的.NET Core SDK

在**新项目**对话框为 ASP.NET Core，你可能会看到以下警告：

> 安装了.NET Core SDK 32 和 64 位版本。仅从安装在 64 位版本的模板 c:\Program Files\dotnet\sdk\将显示。

时，此警告会出现 (x86) 32 位和 64 位 (x64) 版本的.NET Core SDK安装。可能安装这两个版本的常见原因包括：

- 你最初下载.NET Core SDK 安装程序使用 32 位计算机，但然后复制它跨并安装在 64 位计算机上。
- 由另一个应用程序安装了 32 位.NET Core SDK。
- 下载并安装了错误的版本。

卸载 32 位.NET Core SDK，以防止出现此警告。从卸载**Control Panel** > **程序和功能** > **卸载或更改程序**。如果

您了解为何会出现的警告和其影响，则可以忽略该警告。

.NET Core SDK 的安装在多个位置

在**新项目**对话框为 ASP.NET Core，你可能会看到以下警告：

> .NET Core SDK 安装在多个位置中。仅安装在上 SDK 的模板 c:\Program Files\dotnet\sdk\将显示。

外部的一个目录中有至少一个安装的.NET Core SDK 时，将显示此消息*c:\Program Files\dotnet\sdk*。这通常发生在使用复制/粘贴，而不 MSI 安装程序的计算机上部署了.NET Core SDK 时。

卸载 32 位.NET Core SDK，以防止出现此警告。从卸载**Control Panel** > **程序和功能** > **卸载或更改程序**。如果您了解为何会出现的警告和其影响，则可以忽略该警告。

检测到没有.NET Core Sdk

在**新项目**对话框为 ASP.NET Core，你可能会看到以下警告：

> 检测到没有.NET Core Sdk，请确保将它们包括在环境变量 PATH。

时，此警告会出现的环境变量 `PATH` 没有指向计算机上任何.NET Core Sdk。若要解决此问题：

- 安装或验证.NET Core SDK 的安装。
- 验证 `PATH` 环境变量指向在其中安装了 SDK 的位置。安装程序通常设置 `PATH` 。

从应用程序获取数据

如果应用程序能够对请求作出响应，你可以从应用使用中间件获取以下数据：

- 请求–方法、方案、主机、pathbase、路径、查询字符串，标头
- 连接–远程 IP 地址、远程端口、本地 IP 地址、本地端口、客户端证书
- 标识–名称、显示名称
- 配置设置
- 环境变量

将以下项放中间件代码的开头 `Startup.Configure` 方法的请求处理管道。中间件运行以确保仅在开发环境中执行的代码之前，将检查该环境。

若要获取该环境，请使用以下方法之一：

- 注入 `IHostingEnvironment` 到 `Startup.Configure` 方法，并检查本地变量的环境。下面的示例代码演示了这种方法。
- 将在环境中的属性分配 `Startup` 类。检查在环境中使用属性 (例如， `if (Environment.IsDevelopment())`)。

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    IConfiguration config)
{
    if (env.IsDevelopment())
    {
        app.Run(async (context) =>
        {
            var sb = new StringBuilder();
```

```
            var sb = new StringBuilder();
            var nl = System.Environment.NewLine;
            var rule = string.Concat(nl, new string('-', 40), nl);
            var authSchemeProvider = app.ApplicationServices
                .GetRequiredService<IAuthenticationSchemeProvider>();

            sb.Append($"Request{rule}");
            sb.Append($"{DateTimeOffset.Now}{nl}");
            sb.Append($"{context.Request.Method} {context.Request.Path}{nl}");
            sb.Append($"Scheme: {context.Request.Scheme}{nl}");
            sb.Append($"Host: {context.Request.Headers["Host"]}{nl}");
            sb.Append($"PathBase: {context.Request.PathBase.Value}{nl}");
            sb.Append($"Path: {context.Request.Path.Value}{nl}");
            sb.Append($"Query: {context.Request.QueryString.Value}{nl}{nl}");

            sb.Append($"Connection{rule}");
            sb.Append($"RemoteIp: {context.Connection.RemoteIpAddress}{nl}");
            sb.Append($"RemotePort: {context.Connection.RemotePort}{nl}");
            sb.Append($"LocalIp: {context.Connection.LocalIpAddress}{nl}");
            sb.Append($"LocalPort: {context.Connection.LocalPort}{nl}");
            sb.Append($"ClientCert: {context.Connection.ClientCertificate}{nl}{nl}");

            sb.Append($"Identity{rule}");
            sb.Append($"User: {context.User.Identity.Name}{nl}");
            var scheme = await authSchemeProvider
                .GetSchemeAsync(IISDefaults.AuthenticationScheme);
            sb.Append($"DisplayName: {scheme?.DisplayName}{nl}{nl}");

            sb.Append($"Headers{rule}");
            foreach (var header in context.Request.Headers)
            {
                sb.Append($"{header.Key}: {header.Value}{nl}");
            }
            sb.Append(nl);

            sb.Append($"Websockets{rule}");
            if (context.Features.Get<IHttpUpgradeFeature>() != null)
            {
                sb.Append($"Status: Enabled{nl}{nl}");
            }
            else
            {
                sb.Append($"Status: Disabled{nl}{nl}");
            }

            sb.Append($"Configuration{rule}");
            foreach (var pair in config.AsEnumerable())
            {
                sb.Append($"{pair.Path}: {pair.Value}{nl}");
            }
            sb.Append(nl);

            sb.Append($"Environment Variables{rule}");
            var vars = System.Environment.GetEnvironmentVariables();
            foreach (var key in vars.Keys.Cast<string>().OrderBy(key => key,
                StringComparer.OrdinalIgnoreCase))
            {
                var value = vars[key];
                sb.Append($"{key}: {value}{nl}");
            }

            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync(sb.ToString());
        });
    }
}
```

ASP.NET Core 中的日志记录

2018/10/30 • • Edit Online

作者：Steve Smith 和 Tom Dykstra

ASP.NET Core 支持适用于各种内置和第三方日志记录提供程序的日志记录 API。本文介绍了如何将日志记录 API 与内置提供程序一起使用。

查看或下载示例代码（如何下载）

添加提供程序

日志记录提供程序会显示或存储日志。例如，控制台提供程序在控制台上显示日志，Azure Application Insights 提供程序会将这些日志存储在 Azure Application Insights 中。可通过添加多个提供程序将日志发送到多个目标。

要添加提供程序，请在 Program.cs 中调用提供程序的 `Add{provider name}` 扩展方法：

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var env = hostingContext.HostingEnvironment;
            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                  .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                      optional: true, reloadOnChange: true);
            config.AddEnvironmentVariables();
        })
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
            logging.AddEventSourceLogger();
        })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}
```

默认项目模板调用 CreateDefaultBuilder 扩展方法，该操作将添加以下日志记录提供程序：

- 控制台
- 调试
- EventSource（启动位置：ASP.NET Core 2.2）

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
```

如果使用 `CreateDefaultBuilder`，则可自行选择提供程序来替换默认提供程序。调用 ClearProviders，然后添加所需的提供程序。

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    var todoRepository = host.Services.GetRequiredService<ITodoRepository>();
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Feed the dog" });
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Walk the dog" });

    var logger = host.Services.GetRequiredService<ILogger<Program>>();
    logger.LogInformation("Seeded the database.");

    host.Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        })
        .Build();
```

要使用提供程序，请安装其 NuGet 包，并在 ILoggerFactory 的实例上调用提供程序的扩展方法：

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddConsole()
        .AddDebug();
```

ASP.NET Core 依赖关系注入 (DI) 将提供 `ILoggerFactory` 实例。在 Microsoft.Extensions.Logging.Console 和 Microsoft.Extensions.Logging.Debug 包中定义了 `AddConsole` 和 `AddDebug` 扩展方法。每个扩展方法都调用 `ILoggerFactory.AddProvider` 方法，传入提供程序的一个实例。

> **NOTE**
>
> 示例应用在 `Startup.Configure` 方法中添加了日志提供程序。要从先前执行的代码获取日志输出，请在 `Startup` 类构造函数中添加日志提供程序。

详细了解内置日志记录提供程序，以及本文稍后部分介绍的第三方日志记录提供程序。

创建日志

从 DI 中获取 ILogger<TCategoryName> 对象。

以下控制器示例会创建 `Information` 和 `Warning` 日志。类别为 `TodoApiSample.Controllers.TodoController`（示例应用中 `TodoController` 的完全限定类名）：

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

以下 Razor 页面示例会创建"级别"为 `Information` 且"类别"为 `TodoApiSample.Pages.AboutModel` 的日志：

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }
```

```
public void OnGet()
{
    Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";
    _logger.LogInformation("Message displayed: {Message}", Message);
}
```

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

上述示例创建了“级别”为 `Information` 和 `Warning`，且“类别”为 `TodoController` 的日志。

日志“级别”代表所记录事件的严重程度。日志“类别”是与每个日志关联的字符串。`ILogger<T>` 实例会创建“类别”为类型 `T` 的完全限定名称的日志。本文稍后部分将更详细地介绍级别和类别。

启动时创建日志

要将日志写入 `Startup` 类，构造函数签名需包含 `ILogger` 参数：

```
public class Startup
{
    private readonly ILogger _logger;

    public Startup(IConfiguration configuration, ILogger<Startup> logger)
    {
        Configuration = configuration;
        _logger = logger;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();

        // Add our repository type
        services.AddSingleton<ITodoRepository, TodoRepository>();
        _logger.LogInformation("Added TodoRepository to services");
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            _logger.LogInformation("In Development environment");
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();

        app.UseMvc();
    }
}
```

在程序中创建日志

要将日志写入 `Program` 类，请从 DI 获取 `ILogger` 实例：

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    var todoRepository = host.Services.GetRequiredService<ITodoRepository>();
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Feed the dog" });
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Walk the dog" });

    var logger = host.Services.GetRequiredService<ILogger<Program>>();
    logger.LogInformation("Seeded the database.");

    host.Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        })
        .Build();
```

没有异步记录器方法

日志记录应该会很快，不值得牺牲性能来使用异步代码。如果你的日志数据存储很慢，请不要直接写入它。首先考虑将日志消息写入快速存储，售后再将其变为慢速存储。例如，记录到由另一进程读取和暂留以减缓存储的消息队列。

配置

日志记录提供程序配置由一个或多个配置提供程序提供：

- 文件格式（INI、JSON 和 XML）。
- 命令行参数。
- 环境变量。
- 内存中的 .NET 对象。
- 未加密的机密管理器存储。
- 加密的用户存储，如 Azure Key Vault。
- （已安装或已创建的）自定义提供程序。

例如，日志记录配置通常由应用设置文件的 `Logging` 部分提供。以下示例显示了典型 *appsettings.Development.json* 文件的内容：

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    },
    "Console":
    {
      "IncludeScopes": true
    }
  }
}
```

`Logging` 属性可具有 `LogLevel` 和日志提供程序属性（显示控制台）。

`Logging` 下的 `LogLevel` 属性指定了用于记录所选类别的最低级别。在本例中，`System` 和 `Microsoft` 类别在 `Information` 级别记录，其他均在 `Debug` 级别记录。

`Logging` 下的其他属性均指定了日志记录提供程序。本示例针对控制台提供程序。如果提供程序支持日志作用域，则 `IncludeScopes` 将指示是否启用这些域。提供程序属性（例如本例的 `Console`）也可指定 `LogLevel` 属性。提供程序下的 `LogLevel` 指定了该提供程序记录的级别。

如果在 `Logging.{providername}.LogLevel` 中指定了级别，则这些级别将重写 `Logging.LogLevel` 中设置的所有内容。

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

`LogLevel` 项表示日志名称。`Default` 项适用于未显式列出的日志。其值表示应用于给定日志的日志级别。

若要了解如何实现配置提供程序，请参阅 ASP.NET Core 中的配置。

日志记录输出示例

使用上一部分中显示的示例代码从命令行运行应用时，将在控制台中看到日志。以下是控制台输出示例：

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/api/todo/0
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) -
ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      GetById(0) NOT FOUND
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 42.9286ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 148.889ms 404
```

通过向 `http://localhost:5000/api/todo/0` 处的示例应用发出 HTTP Get 请求来生成前述日志。

在 Visual Studio 中运行示例应用时，“调试”窗口中将显示如下日志：

```
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request starting HTTP/1.1 GET
http://localhost:53104/api/todo/0
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executing action method
TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) - ModelState is Valid
TodoApi.Controllers.TodoController:Information: Getting item 0
TodoApi.Controllers.TodoController:Warning: GetById(0) NOT FOUND
Microsoft.AspNetCore.Mvc.StatusCodeResult:Information: Executing HttpStatusCodeResult, setting HTTP
status code 404
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executed action
TodoApi.Controllers.TodoController.GetById (TodoApi) in 152.5657ms
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request finished in 316.3195ms 404
```

由上一部分所示的 `ILogger` 调用创建的日志是以“TodoApi.Controllers.TodoController”开头的。以“Microsoft”类别开头的日志来自 ASP.NET Core 框架代码。ASP.NET Core 和应用程序代码使用相同的日志记录 API 和提供程序。

本文余下部分将介绍有关日志记录的某些详细信息及选项。

NuGet 包

`ILogger` 和 `ILoggerFactory` 接口位于 Microsoft.Extensions.Logging.Abstractions 中，其默认实现位于 Microsoft.Extensions.Logging 中。

日志类别

创建 `ILogger` 对象后，将为其指定“类别”。该类别包含在由此 `Ilogger` 实例创建的每条日志消息中。类别可以是任何字符串，但约定需使用类名，例如“TodoApi.Controllers.TodoController”。

使用 `ILogger<T>` 获取一个 `ILogger` 实例，该实例使用 `T` 的完全限定类型名称作为类别：

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```


```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```


要显式指定类别，请调用 `ILoggerFactory.CreateLogger`：

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILoggerFactory logger)
    {
        _todoRepository = todoRepository;
        _logger = logger.CreateLogger("TodoApiSample.Controllers.TodoController");
    }
```

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILoggerFactory logger)
    {
        _todoRepository = todoRepository;
        _logger = logger.CreateLogger("TodoApiSample.Controllers.TodoController");
    }
```

`ILogger<T>` 相当于使用 `T` 的完全限定类型名称来调用 `CreateLogger`。

日志级别

每个日志都指定了一个 LogLevel 值。日志级别指示严重性或重要程度。例如，可在方法正常结束时写入 `Information` 日志，在方法返回"404 找不到"状态代码时写入 `Warning` 日志。

下面的代码会创建 `Information` 和 `Warning` 日志：

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```


```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```


在上述代码中，第一个参数是日志事件 ID。第二个参数是消息模板，其中的占位符用于填写剩余方法形参提供的实参值。稍后将在本文的消息模板部分介绍方法参数。

在方法名称中包含级别的日志方法（例如 `LogInformation` 和 `LogWarning`）是 ILogger 的扩展方法。这些方法会调用可接受 `LogLevel` 参数的 `Log` 方法。可直接调用 `Log` 方法而不调用其中某个扩展方法，但语法相对复杂。有关详细信息，请参阅 ILogger 和记录器扩展源代码。

ASP.NET Core 定义了以下日志级别（按严重性从低到高排列）。

- 跟踪 = 0

  有关通常仅用于调试的信息。这些消息可能包含敏感应用程序数据，因此不得在生产环境中启用它们。默认情况下禁用。

- 调试 = 1

有关在开发和调试中可能有用的信息。示例：由于 日志数量过多，因此 `Entering method Configure with flag set to true.` 仅当故障排除时才在生产中启用 `Debug` 级别日志。

- 信息 = 2

  用于跟踪应用的常规流。这些日志通常有长期价值。示例：`Request received for path /api/todo`

- 警告 = 3

  表示应用流中的异常或意外事件。可能包括不会中断应用运行但仍需调查的错误或其他条件。`Warning` 日志级别常用于已处理的异常。示例：`FileNotFoundException for file quotes.txt.`

- 错误 = 4

  表示无法处理的错误和异常。这些消息指示的是当前活动或操作（例如当前 HTTP 请求）中的失败，而不是整个应用中的失败。日志消息示例：`Cannot insert record due to duplicate key violation.`

- 严重 = 5

  需要立即关注的失败。例如数据丢失、磁盘空间不足。

使用日志级别控制写入到特定存储介质或显示窗口的日志输出量。例如：

- 在生产中，通过 `Information` 级别将 `Trace` 发送到卷数据存储。通过 `Critical` 级别将 `Warning` 发送到值数据存储。
- 在开发过程中，通过 `Critical` 级别将 `Warning` 发送到控制台，并在进行故障排除时通过 `Information` 级别添加 `Trace`。

本文稍后的日志筛选部分介绍如何控制提供程序处理的日志级别。

ASP.NET Core 为框架事件写入日志。本文前面部分提供的日志示例排除了低于 `Information` 级别的日志，因此未创建 `Debug` 或 `Trace` 级别日志。以下示例介绍了通过运行配置为显示 `Debug` 日志的示例应用而生成的控制台日志：

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:62555/api/todo/0
dbug: Microsoft.AspNetCore.Routing.Tree.TreeRouter[1]
      Request successfully matched the route with name 'GetTodo' and template 'api/Todo/{id}'.
dbug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'TodoApi.Controllers.TodoController.Update (TodoApi)' with id '089d59b6-92ec-472d-b552-
cc613dfd625d' did not match the constraint
'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
dbug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'TodoApi.Controllers.TodoController.Delete (TodoApi)' with id 'f3476abe-4bd9-4ad3-9261-
3ead09607366' did not match the constraint
'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
dbug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action TodoApi.Controllers.TodoController.GetById (TodoApi)
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) -
ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      GetById(0) NOT FOUND
dbug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action method TodoApi.Controllers.TodoController.GetById (TodoApi), returned result
Microsoft.AspNetCore.Mvc.NotFoundResult.
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 0.8788ms
dbug: Microsoft.AspNetCore.Server.Kestrel[9]
      Connection id "0HL6L7NEFF2QD" completed keep alive response.
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 2.7286ms 404
```

日志事件 ID

每个日志都可指定一个事件 ID。该示例应用通过使用本地定义的 `LoggingEvents` 类来执行此操作：

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

```
public class LoggingEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems = 1001;
    public const int GetItem = 1002;
    public const int InsertItem = 1003;
    public const int UpdateItem = 1004;
    public const int DeleteItem = 1005;

    public const int GetItemNotFound = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

```
public class LoggingEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems = 1001;
    public const int GetItem = 1002;
    public const int InsertItem = 1003;
    public const int UpdateItem = 1004;
    public const int DeleteItem = 1005;

    public const int GetItemNotFound = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

事件 ID 与一组事件相关联。例如，与在页面上显示项列表相关的所有日志可能是 1001。

日志记录提供程序可将事件 ID 存储在 ID 字段中，存储在日志记录消息中，或者不进行存储。调试提供程序不显示事件 ID。控制台提供程序在类别后的括号中显示事件 ID：

```
info: TodoApi.Controllers.TodoController[1002]
      Getting item invalidid
warn: TodoApi.Controllers.TodoController[4000]
      GetById(invalidid) NOT FOUND
```

日志消息模板

每个日志都会指定一个消息模板。消息模板可包含要填写参数的占位符。请在占位符中使用名称而不是数字。

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

占位符的顺序(而非其名称)决定了为其提供值的参数。在以下代码中，请注意消息模板中的参数名称未按顺序排列：

```
string p1 = "parm1";
string p2 = "parm2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

此代码创建了一个参数值按顺序排列的日志消息：

```
Parameter values: parm1, parm2
```

日志记录框架按此方式工作，这样日志记录提供程序即可实现语义日志记录，也称为结构化日志记录。参数本身会传递给日志记录系统，而不仅仅是格式化的消息模板。通过此信息，日志记录提供程序能够将参数值存储为字段。例如，假设记录器方法调用如下所示：

```
_logger.LogInformation("Getting item {ID} at {RequestTime}", id, DateTime.Now);
```

如果要将日志发送到 Azure 表存储，则每个 Azure 表实体都可具有 `ID` 和 `RequestTime` 属性，这简化了对日志数据的查询。无需分析文本消息的超时，查询即可找到特定 `RequestTime` 范围内的全部日志。

日志记录异常

记录器方法有可传入异常的重载，如下方示例所示：

```
catch (Exception ex)
{
    _logger.LogWarning(LoggingEvents.GetItemNotFound, ex, "GetById({ID}) NOT FOUND", id);
    return NotFound();
}
return new ObjectResult(item);
```


```
catch (Exception ex)
{
    _logger.LogWarning(LoggingEvents.GetItemNotFound, ex, "GetById({ID}) NOT FOUND", id);
    return NotFound();
}
return new ObjectResult(item);
```


不同的提供程序处理异常信息的方式不同。以下是上示代码的调试提供程序输出示例。

```
TodoApi.Controllers.TodoController:Warning: GetById(036dd898-fb01-47e8-9a65-f92eb73cf924) NOT FOUND

System.Exception: Item not found exception.
 at TodoApi.Controllers.TodoController.GetById(String id) in
C:\logging\sample\src\TodoApi\Controllers\TodoController.cs:line 226
```

日志筛选

可为特定或所有提供程序和类别指定最低日志级别。最低级别以下的日志不会传递给该提供程序，因此不会显示或存储它们。

要禁止显示所有日志，可将 `LogLevel.None` 指定为最低日志级别。`LogLevel.None` 的整数值为 6，它大于 `LogLevel.Critical` (5)。

在配置中创建筛选规则

项目模板代码调用 `CreateDefaultBuilder` 来为控制台和调试提供程序设置日志记录。`CreateDefaultBuilder` 方法还使用如下所示的代码，设置日志记录以查找 `Logging` 部分的配置：

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var env = hostingContext.HostingEnvironment;
            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                  .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                      optional: true, reloadOnChange: true);
            config.AddEnvironmentVariables();
        })
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
            logging.AddEventSourceLogger();
        })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}
```

配置数据按提供程序和类别指定最低日志级别，如下方示例所示：

```
{
  "Logging": {
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "IncludeScopes": false,
      "LogLevel": {
        "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
        "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
        "Microsoft.AspNetCore.Mvc.Razor": "Error",
        "Default": "Information"
      }
    },
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

此 JSON 将创建 6 条筛选规则：1 条用于调试提供程序，4 条用于控制台提供程序，1 条用于所有提供程序。创建 `ILogger` 对象时，为每个提供程序选择一个规则。

代码中的筛选规则

下面的示例演示了如何在代码中注册筛选规则：

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging =>
        logging.AddFilter("System", LogLevel.Debug)
               .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace))
    .Build();
```

第二个 `AddFilter` 使用类型名称来指定调试提供程序。第一个 `AddFilter` 应用于全部提供程序，因为它未指定提供程序类型。

如何应用筛选规则

先前示例中显示的配置数据和 `AddFilter` 代码会创建下表所示的规则。前六条由配置示例创建，后两条由代码示例创建。

数字	提供程序	类别的开头为...	最低日志级别
1	调试	全部类别	信息
2	控制台	Microsoft.AspNetCore.Mvc.Razor.Internal	警告
3	控制台	Microsoft.AspNetCore.Mvc.Razor.Razor	调试
4	控制台	Microsoft.AspNetCore.Mvc.Razor	Error
5	控制台	全部类别	信息

数字	提供程序	类别的开头为...	最低日志级别
6	全部提供程序	全部类别	调试
7	全部提供程序	系统	调试
8	调试	Microsoft	跟踪

创建 `ILogger` 对象时，`ILoggerFactory` 对象将根据提供程序选择一条规则，将其应用于该记录器。将按所选规则筛选 `ILogger` 实例写入的所有消息。从可用规则中为每个提供程序和类别对选择最具体的规则。

在为给定的类别创建 `ILogger` 时，以下算法将用于每个提供程序：

- 选择匹配提供程序或其别名的所有规则。如果找不到任何匹配项，则选择提供程序为空的所有规则。
- 根据上一步的结果，选择具有最长匹配类别前缀的规则。如果找不到任何匹配项，则选择未指定类别的所有规则。
- 如果选择了多条规则，则采用最后一条。
- 如果未选择任何规则，则使用 `MinimumLevel`。

假设你使用上述规则列表为类别“Microsoft.AspNetCore.Mvc.Razor.RazorViewEngine”创建了 `ILogger` 对象：

- 对于调试提供程序，规则 1、6 和 8 适用。规则 8 最为具体，因此选择了它。
- 对于控制台提供程序，符合的有规则 3、规则 4、规则 5 和规则 6。规则 3 最为具体。

生成的 `ILogger` 实例将 `Trace` 级别及更高级别的日志发送到调试提供程序。`Debug` 级别及更高级别的日志会发送到控制台提供程序。

提供程序别名

每个提供程序都定义了一个别名；可在配置中使用该别名来代替完全限定的类型名称。对于内置提供程序，请使用以下别名：

- 控制台
- 调试
- EventLog
- AzureAppServices
- TraceSource
- EventSource

默认最低级别

仅当配置或代码中的规则对给定提供程序和类别都不适用时，最低级别设置才会生效。下面的示例演示如何设置最低级别：

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging => logging.SetMinimumLevel(LogLevel.Warning))
    .Build();
```

如果没有明确设置最低级别，则默认值为 `Information`，它表示 `Trace` 和 `Debug` 日志将被忽略。

筛选器函数

对配置或代码没有向其分配规则的所有提供程序和类别调用筛选器函数。函数中的代码可访问提供程序类型、类别和日志级别。例如：

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logBuilder =>
    {
        logBuilder.AddFilter((provider, category, logLevel) =>
        {
            if (provider == "Microsoft.Extensions.Logging.Console.ConsoleLoggerProvider" &&
                category == "TodoApiSample.Controllers.TodoController")
            {
                return false;
            }
            return true;
        });
    })
    .Build();
```

通过某些日志记录提供程序，可根据日志级别和类别指定何时向存储介质写入日志、何时忽略日志。

`AddConsole` 和 `AddDebug` 扩展方法提供了接受筛选条件的重载。下列示例代码将使控制台提供程序忽略低于 `Warning` 级别的日志，而使调试提供程序忽略由框架创建的日志。

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddConsole(LogLevel.Warning)
        .AddDebug((category, logLevel) => (category.Contains("TodoApi") && logLevel >= LogLevel.Trace));
```

`AddEventLog` 方法拥有接受 `EventLogSettings` 实例的重载，该实例的 `Filter` 属性中可能包含筛选函数。TraceSource 提供程序不提供以上任何重载，因为其日志记录级别和其他参数都以它使用的 `SourceSwitch` 和 `TraceListener` 为依据。

可使用 `WithFilter` 扩展方法为所有通过 `ILoggerFactory` 实例注册的提供程序设置筛选规则。下面的示例将（类别以"Microsoft"或"System"开头的）框架日志限制为警告，并在调试级别记录应用程序代码创建的日志。

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .WithFilter(new FilterLoggerSettings
        {
            { "Microsoft", LogLevel.Warning },
            { "System", LogLevel.Warning },
            { "ToDoApi", LogLevel.Debug }
        })
        .AddConsole()
        .AddDebug();
```

要防止写入任何日志，请将 `LogLevel.None` 指定为最低日志级别。`LogLevel.None` 的整数值为 6，它大于 `LogLevel.Critical` (5)。

`WithFilter` 扩展方法由 Microsoft.Extensions.Logging.Filter NuGet 包提供。该方法返回一个新的 `ILoggerFactory` 实例，该实例将筛选传递给注册的所有记录器提供程序的日志消息。它不会影响其他任何 `ILoggerFactory` 实例，包括原始 `ILoggerFactory` 实例。

系统类别和级别

下面是 ASP.NET Core 和 Entity Framework Core 使用的一些类别，备注中说明了可从这些类别获取的具体日志：

类别	说明
Microsoft.AspNetCore	常规 ASP.NET Core 诊断。
Microsoft.AspNetCore.DataProtection	考虑、找到并使用了哪些密钥。
Microsoft.AspNetCore.HostFiltering	所允许的主机。
Microsoft.AspNetCore.Hosting	HTTP 请求完成的时间和启动时间。加载了哪些承载启动程序集。
Microsoft.AspNetCore.Mvc	MVC 和 Razor 诊断。模型绑定、筛选器执行、视图编译和操作选择。
Microsoft.AspNetCore.Routing	路由匹配信息。
Microsoft.AspNetCore.Server	连接启动、停止和保持活动响应。HTTP 证书信息。
Microsoft.AspNetCore.StaticFiles	提供的文件。
Microsoft.EntityFrameworkCore	常规 Entity Framework Core 诊断。数据库活动和配置、更改检测、迁移。

日志作用域

“作用域”可对一组逻辑操作分组。此分组可用于将相同的数据附加到作为集合的一部分而创建的每个日志。例如，在处理事务期间创建的每个日志都可包括事务 ID。

范围是由 BeginScope 方法返回的 `IDisposable` 类型，持续至释放为止。要使用作用域，请在 `using` 块中包装记录器调用：

```
public IActionResult GetById(string id)
{
    TodoItem item;
    using (_logger.BeginScope("Message attached to logs created in the using block"))
    {
        _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
        item = _todoRepository.Find(id);
        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
            return NotFound();
        }
    }
    return new ObjectResult(item);
}
```

下列代码为控制台提供程序启用作用域：

Program.cs:

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
    logging.AddConsole(options => options.IncludeScopes = true);
    logging.AddDebug();
})
```

> **NOTE**
>
> 要启用基于作用域的日志记录，必须先配置 `IncludeScopes` 控制台记录器选项。
>
> 若要了解关配置，请参阅配置部分。

Program.cs:

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
    logging.AddConsole(options => options.IncludeScopes = true);
    logging.AddDebug();
})
```

> **NOTE**
>
> 要启用基于作用域的日志记录，必须先配置 `IncludeScopes` 控制台记录器选项。

Startup.cs:

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddConsole(includeScopes: true)
        .AddDebug();
```

每条日志消息都包含作用域内的信息：

```
info: TodoApi.Controllers.TodoController[1002]
      => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 => TodoApi.Controllers.TodoController.GetById
(TodoApi) => Message attached to logs created in the using block
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 => TodoApi.Controllers.TodoController.GetById
(TodoApi) => Message attached to logs created in the using block
      GetById(0) NOT FOUND
```

内置日志记录提供程序

ASP.NET Core 提供以下提供程序：

- 控制台
- 调试
- EventSource

- EventLog
- TraceSource

本文稍后将介绍 Azure 的日志记录选项。

有关 stdout 日志记录的信息，请参阅 对 IIS 上的 ASP.NET Core 进行故障排除 和 对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除。

控制台提供程序

Microsoft.Extensions.Logging.Console 提供程序包向控制台发送日志输出。

```
logging.AddConsole();
```

```
loggerFactory.AddConsole();
```

通过 AddConsole 重载，可传入一个最低日志级别、一个筛选器函数，以及一个用于指示作用域是否受支持的布尔值。另一个选项是传递 `IConfiguration` 对象，可通过它来指定支持的作用域及日志记录级别。

控制台提供程序对性能有重大影响，通常不适合在生产中使用。

在 Visual Studio 中创建新项目时，`AddConsole` 方法如下所示：

```
loggerFactory.AddConsole(Configuration.GetSection("Logging"));
```

此代码引用 appSettings.json 文件的 `Logging` 部分：

```
{
  "Logging": {
    "Console": {
      "IncludeScopes": false
    },
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

所显示的设置将框架日志限制为警告，并允许在调试级别记录应用日志，如日志筛选部分所述。有关详细信息，请参阅配置。

要查看控制台日志记录输出，请在项目文件夹中打开命令提示符并运行以下命令：

```
dotnet run
```

调试提供程序

Microsoft.Extensions.Logging.Debug 提供程序包使用 System.Diagnostics.Debug 类（`Debug.WriteLine` 方法调用）来写入日志输出。

在 Linux 中，此提供程序将日志写入 /var/log/message。

```
logging.AddDebug();
```

```
loggerFactory.AddDebug();
```

AddDebug 重载允许传入最低日志级别或筛选器函数。

EventSource 提供程序

对于面向 ASP.NET Core 1.1.0 或更高版本的应用，Microsoft.Extensions.Logging.EventSource 提供程序包可实现事件跟踪。在 Windows 中，它使用 ETW。提供程序可跨平台使用，但尚无支持 Linux 或 macOS 的事件集合和显示工具。

```
logging.AddEventSourceLogger();
```

```
loggerFactory.AddEventSourceLogger();
```

可使用 PerfView 实用工具收集和查看日志。虽然其他工具也可以查看 ETW 日志，但在处理由 ASP.NET 发出的 ETW 事件时，使用 PerfView 能获得最佳体验。

要将 PerfView 配置为收集此提供程序记录的事件，请向 Additional Providers 列表添加字符串 `*Microsoft-Extensions-Logging`。（请勿遗漏字符串起始处的星号。）

Windows EventLog 提供程序

Microsoft.Extensions.Logging.EventLog 提供程序包向 Windows 事件日志发送日志输出。

```
logging.AddEventLog();
```

```
loggerFactory.AddEventLog();
```

AddEventLog 重载允许传入 `EventLogSettings` 或最低日志级别。

TraceSource 提供程序

Microsoft.Extensions.Logging.TraceSource 提供程序包使用 TraceSource 库和提供程序。

```
logging.AddTraceSource(sourceSwitchName);
```

```
loggerFactory.AddTraceSource(sourceSwitchName);
```

AddTraceSource 重载 允许传入资源开关和跟踪侦听器。

要使用此提供程序，应用必须在 .NET Framework（而非 .NET Core）上运行。提供程序可将消息路由到多个侦听器，例如示例应用中使用的 TextWriterTraceListener。

以下示例配置了 `TraceSource` 提供程序，用于向控制台窗口记录 `Warning` 和更高级别的消息。

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddDebug();

    // add Trace Source logging
    var testSwitch = new SourceSwitch("sourceSwitch", "Logging Sample");
    testSwitch.Level = SourceLevels.Warning;
    loggerFactory.AddTraceSource(testSwitch,
        new TextWriterTraceListener(writer: Console.Out));
```

Azure 中的日志记录

要了解 Azure 中的日志记录，请参阅下列部分：

- Azure 应用服务提供程序
- Azure 日志流式处理
- Azure Application Insights 跟踪日志记录

Azure 应用服务提供程序

Microsoft.Extensions.Logging.AzureAppServices 提供程序包将日志写入 Azure App Service 应用的文件系统，以及 Azure 存储帐户中的 blob 存储。面向 .NET Core 1.1 或更高版本的应用可使用该提供程序包。

如果面向 .NET Core，请注意以下几点：

- 提供程序包随附在 ASP.NET Core Microsoft.AspNetCore.All 元包中。
- Microsoft.AspNetCore.App 元包中不包括此提供程序包。要使用提供程序，请安装该程序包。
- 不要显式调用 AddAzureWebAppDiagnostics。将应用部署到 Azure 应用服务时，会自动使提供程序对应用可用。

如果面向 .NET Framework 或引用 `Microsoft.AspNetCore.App` 元包，请向项目添加提供程序包。在 ILoggerFactory 实例上调用 `AddAzureWebAppDiagnostics`：

```
logging.AddAzureWebAppDiagnostics();
```

```
loggerFactory.AddAzureWebAppDiagnostics();
```

AddAzureWebAppDiagnostics 重载允许传入 AzureAppServicesDiagnosticsSettings。设置对象可以覆盖默认设置，例如日志记录输出模板、blob 名称和文件大小限制。（“输出模板”是一个消息模板，除了通过

`ILogger` 方法调用提供的内容之外，还可将其应用于所有日志。）

在部署应用服务应用时，应用程序将遵循 Azure 门户中“应用服务”页面下的诊断日志部分的设置。更新这些设置后，更改会立即生效，无需重新启动或重新部署应用。

日志文件的默认位置是 D:\home\LogFiles\Application 文件夹，默认文件名为 diagnostics-yyyymmdd.txt。默认文件大小上限为 10 MB，默认最大保留文件数为 2。默认 blob 名为 {app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt。要详细了解默认行为，请参阅 AzureAppServicesDiagnosticsSettings。

该提供程序仅当项目在 Azure 环境中运行时有效。项目在本地运行时，该提供程序无效 — 它不会写入本地文件或 blob 的本地开发存储。

Azure 日志流式处理

通过 Azure 日志流式处理，可从以下位置实时查看日志活动：

- 应用服务器
- Web 服务器
- 请求跟踪失败

要配置 Azure 日志流式处理，请执行以下操作：

- 从应用的门户页导航到“诊断日志”页。
- 将“应用程序日志记录(Filesystem)”设置为“开”。

导航到"日志流式处理"页，查看应用消息。它们由应用通过 `ILogger` 接口记录。

Azure Application Insights 跟踪日志记录

Application Insights SDK 可收集和报告 ASP.NET Core 日志记录基础结构生成的日志。有关更多信息，请参见以下资源：

- Application Insights 概述
- 用于 ASP.NET Core 的 Application Insights
- Microsoft/ApplicationInsights-aspnetcore Wiki：日志记录。

第三方日志记录提供程序

适用于 ASP.NET Core 的第三方日志记录框架：

- elmah.io（GitHub 存储库）
- Gelf（GitHub 存储库）
- JSNLog（GitHub 存储库）
- KissLog.net（GitHub 存储库）
- Loggr（GitHub 存储库）
- NLog（GitHub 存储库）
- Sentry（GitHub 存储库）
- Serilog（GitHub 存储库）
- Stackdriver（Github 存储库）

某些第三方框架可以执行语义日志记录（又称结构化日志记录）。

使用第三方框架类似于使用以下内置提供程序之一：

1. 将 NuGet 包添加到你的项目。
2. 调用 `ILoggerFactory` 。

有关详细信息，请参阅各提供程序的相关文档。Microsoft 不支持第三方日志记录提供程序。

其他资源

- 在 ASP.NET Core 中使用 LoggerMessage 的高性能日志记录

ASP.NET Core Razor 页面和 EF Core - 教程系列

2018/7/11 • • Edit Online

下面的一系列教程介绍如何创建使用 Entity Framework (EF) Core 进行数据访问的 ASP.NET Core Razor Pages Web 应用。

1. 入门
2. 创建、读取、更新和删除操作
3. 排序、筛选、分页和分组
4. 迁移
5. 创建复杂数据模型
6. 读取相关数据
7. 更新相关数据
8. 处理并发冲突

ASP.NET Core 中的 Razor 页面和 Entity Framework Core - 第 1 个教程(共 8 个)

2019/1/10 • • Edit Online

可在此 PDF 文件中找到本教程的 ASP.NET Core 2.0 版。

相对于 2.0 版，本教程的 ASP.NET Core 2.1 版有很多改进。

作者：Tom Dykstra 和 Rick Anderson

Contoso University 示例 Web 应用演示了如何使用 Entity Framework (EF) Core 创建 ASP.NET Core Razor Pages 应用。

该示例应用是一个虚构的 Contoso University 的网站。其中包括学生录取、课程创建和讲师分配等功能。本页是介绍如何构建 Contoso University 示例应用系列教程中的第一部分。

下载或查看已完成的应用。下载说明。

系统必备

- Visual Studio
- .NET Core CLI

具有以下工作负载的 Visual Studio 2017 15.7.3 版或更高版本：

- **ASP.NET 和 Web 开发**
- **.NET Core 跨平台开发**

.NET Core 2.1 SDK 或更高版本

熟悉 Razor 页面。新程序员在开始学习本系列之前，应先完成 Razor 页面入门。

疑难解答

如果遇到无法解决的问题，可以通过与 已完成的项目对比代码来查找解决方案。获取帮助的一个好方法是将问题发布到适用于 ASP.NET Core 或 EF Core 的 StackOverflow.com。

Contoso University Web 应用

这些教程中所构建的应用是一个基本的大学网站。

用户可以查看和更新学生、课程和讲师信息。以下是在本教程中创建的几个屏幕。

此网站的 UI 样式与内置模板生成的 UI 样式类似。教程的重点是 EF Core 和 Razor 页面，而非 UI。

创建 ContosoUniversity Razor Pages Web 应用

- Visual Studio
- .NET Core CLI

- 从 Visual Studio“文件”菜单中，选择“新建”>“项目”。
- 创建新的 ASP.NET Core Web 应用程序。将该项目命名为 ContosoUniversity 。务必将该项目命名为 ContosoUniversity，以便复制/粘贴代码时命名空间相匹配。
- 在下拉列表中选择“ASP.NET Core 2.1”，然后选择“Web 应用程序”。

有关上述步骤的图像，请参阅创建 Razor Web 应用。运行应用。

设置网站样式

设置网站菜单、布局和主页时需作少量更改。进行以下更改以更新 Pages/Shared/_Layout.cshtml：

- 将文件中的"ContosoUniversity"更改为"Contoso University"。需要更改三个地方。

- 添加菜单项 **Students**，**Courses**，**Instructors**，和 **Department**，并删除 **Contact**菜单项。

突出显示所作更改。（所有标记均不显示。）

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] : Contoso University</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-
test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-brand">Contoso University</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Students/Index">Students</a></li>
                    <li><a asp-page="/Courses/Index">Courses</a></li>
                    <li><a asp-page="/Instructors/Index">Instructors</a></li>
                    <li><a asp-page="/Departments/Index">Departments</a></li>
                </ul>
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 : Contoso University</p>
        </footer>
    </div>

    @*Remaining markup not shown for brevity.*@
```

在 Pages/Index.cshtml 中，将文件内容替换为以下代码，以将有关 ASP.NET 和 MVC 的文本替换为有关本应用的文本：

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core Razor Pages web app.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p>
            <a class="btn btn-default"
               href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro">
                See the tutorial &raquo;
            </a>
        </p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p>
            <a class="btn btn-default"
               href="https://github.com/aspnet/Docs/tree/master/aspnetcore/data/ef-
rp/intro/samples/cu-final">
                See project source code &raquo;
            </a>
        </p>
    </div>
</div>
```

创建数据模型

创建 Contoso University 应用的实体类。从以下三个实体开始：

`Student` 和 `Enrollment` 实体之间存在一对多关系。`Course` 和 `Enrollment` 实体之间存在一对多关系。一名学生可以报名参加任意数量的课程。一门课程中可以包含任意数量的学生。

以下部分将为这几个实体中的每一个实体创建一个类。

Student 实体

创建 Models 文件夹。在 Models 文件夹中，使用以下代码创建一个名为 Student.cs 的类文件：

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`ID` 属性成为此类对应的数据库 (DB) 表的主键列。默认情况下，EF Core 将名为 `ID` 或 `classnameID` 的属性视为主键。在 `classnameID` 中，`classname` 为类名称。另一种自动识别的主键是上例中的 `StudentID`。

`Enrollments` 属性是导航属性。导航属性链接到与此实体相关的其他实体。在这种情况下，`Student entity` 的 `Enrollments` 属性包含与该 `Student` 相关的所有 `Enrollment` 实体。例如，如果数据库中的 Student 行有两个相关的 Enrollment 行，则 `Enrollments` 导航属性包含这两个 `Enrollment` 实体。相关的 `Enrollment` 行是 `StudentID` 列中包含该学生的主键值的行。例如，假设 ID=1 的学生在 `Enrollment` 表中有两行。`Enrollment` 表中有两行的 `StudentID` = 1。`StudentID` 是 `Enrollment` 表中的外键，用于指定 `Student` 表中的学生。

如果导航属性包含多个实体，则导航属性必须是列表类型，例如 `ICollection<T>`。可以指定 `ICollection<T>` 或诸如 `List<T>` 或 `HashSet<T>` 的类型。使用 `ICollection<T>` 时，EF Core 会默认创建 `HashSet<T>` 集合。包含多个实体的导航属性来自于多对多和一对多关系。

Enrollment 实体

在 Models 文件夹中，使用以下代码创建 Enrollment.cs：

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

`EnrollmentID` 属性为主键。`Student` 实体使用的是 `ID` 模式，而本实体使用的是 `classnameID` 模式。通常情况下，开发者会选择一种模式并在整个数据模型中都使用该模式。下一个教程将介绍如何使用不带类名的 ID，以便更轻松地在数据模型中实现集成。

`Grade` 属性为 `enum`。`Grade` 声明类型后的 `?` 表示 `Grade` 属性可以为 null。评级为 null 和评级为零是有区别的 --null 意味着评级未知或者尚未分配。

`StudentID` 属性是外键，其对应的导航属性为 `Student`。`Enrollment` 实体与一个 `Student` 实体相关联，因此该属性只包含一个 `Student` 实体。`Student` 实体与 `Student.Enrollments` 导航属性不同，后者包含多个 `Enrollment` 实体。

`CourseID` 属性是外键，其对应的导航属性为 `Course`。`Enrollment` 实体与一个 `Course` 实体相关联。

如果属性命名为 `<navigation property name><primary key property name>`，EF Core 会将其视为外键。例如 `Student` 导航属性的 `StudentID`，因为 `Student` 实体的主键为 `ID`。还可以将外键属性命名为 `<primary key property name>`。例如 `CourseID`，因为 `Course` 实体的主键为 `CourseID`。

Course 实体

在 Models 文件夹中，使用以下代码创建 Course.cs：

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`Enrollments` 属性是导航属性。`Course` 实体可与任意数量的 `Enrollment` 实体相关。

应用可以通过 `DatabaseGenerated` 特性指定主键，而无需靠数据库生成。

为“学生”模型搭建基架

本部分将为“学生”模型搭建基架。确切地说，基架工具将生成页面，用于对“学生”模型执行创建、读取、更新和删除 (CRUD) 操作。

- 生成项目。
- 创建 Pages/Students 文件夹。

- Visual Studio
- .NET Core CLI

- 在“解决方案资源管理器”中，右键单击“Pages/Students”文件夹>“添加”>“新搭建基架的项目”。
- 在“添加基架”对话框中，选择“使用实体框架生成 Razor Pages (CRUD)”>“添加”。

完成“使用实体框架(CRUD)添加 Razor Pages”对话框：

- 在“模型类”下拉列表中，选择“学生(ContosoUniversity.Models)”。
- 在“数据上下文类”行中，选择加号 (+) 并将生成的名称更改为 ContosoUniversity.Models.SchoolContext。
- 在“数据上下文类”下拉列表中，选择“ContosoUniversity.Models.SchoolContext”
- 选择“添加”。

如果对前面的步骤有疑问，请参阅搭建"电影"模型的基架。

搭建基架的过程会创建并更改以下文件：

创建的文件

- Pages/Students："创建"、"删除"、"详细信息"、"编辑"、"索引"。
- Data/SchoolContext.cs

文件更新

- *Startup.cs*：下一部分详细介绍对此文件所作的更改。
- *appsettings.json*：添加用于连接到本地数据库的连接字符串。

检查通过依赖关系注入注册的上下文

ASP.NET Core 通过依赖关系注入进行生成。服务（例如 EF Core 数据库上下文）在应用程序启动期间通过依赖关系注入进行注册。需要这些服务（如 Razor 页面）的组件通过构造函数提供相应服务。本教程的后续部分介绍了用于获取数据库上下文实例的构造函数代码。

基架工具自动创建 DB 上下文并将其注册到依赖关系注入容器。

在 Startup.cs 中检查 `ConfigureServices` 方法。基架添加了突出显示的行：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for
        //non -essential cookies is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddDbContext<SchoolContext>(options =>
       options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));
}
```

通过调用 DbContextOptions 对象中的一个方法将连接字符串名称传递到上下文。进行本地开发时，

ASP.NET Core 配置系统 在 *appsettings.json* 文件中读取数据库连接字符串。

更新 main

在 Program.cs 中，修改 `Main` 方法以执行以下操作：

- 从依赖关系注入容器获取数据库上下文实例。
- 调用 EnsureCreated。
- `EnsureCreated` 方法完成时释放上下文。

下面的代码显示更新后的 Program.cs 文件。

```
using ContosoUniversity.Models;                   // SchoolContext
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;   // CreateScope
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateWebHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    context.Database.EnsureCreated();
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the DB.");
                }
            }

            host.Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

`EnsureCreated` 确保存在上下文数据库。如果存在，则不需要任何操作。如果不存在，则会创建数据库及其所有架构。`EnsureCreated` 不使用迁移创建数据库。使用 `EnsureCreated` 创建的数据库稍后无法使用迁移更新。

启动应用时会调用 `EnsureCreated`，以进行以下工作流：

- 删除数据库。
- 更改数据库架构（例如添加一个 `EmailAddress` 字段）。
- 运行应用。
- `EnsureCreated` 创建一个带有 `EmailAddress` 列的数据库。

架构快速演变时，在开发初期使用 `EnsureCreated` 很方便。本教程后面将删除 DB 并使用迁移。

测试应用

运行应用并接受 cookie 策略。此应用不保留个人信息。有关 cookie 策略的信息，请参阅欧盟一般数据保护条例 (GDPR) 支持。

- 依次选择"学生"链接、"新建"。
- 测试"编辑"、"详细信息"和"删除"链接。

检查 SchoolContext DB 上下文

数据库上下文类是为给定数据模型协调 EF Core 功能的主类。数据上下文派生自 Microsoft.EntityFrameworkCore.DbContext。数据上下文指定数据模型中包含哪些实体。在此项目中将数据库上下文类命名为 `SchoolContext`。

使用以下代码更新 SchoolContext.cs：

```
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Models
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options)
            : base(options)
        {
        }

        public DbSet<Student> Student { get; set; }
        public DbSet<Enrollment> Enrollment { get; set; }
        public DbSet<Course> Course { get; set; }
    }
}
```

突出显示的代码为每个实体集创建 DbSet<TEntity> 属性。在 EF Core 术语中：

- 实体集通常对应一个数据库表。
- 实体对应表中的行。

可以省略 `DbSet<Enrollment>` 和 `DbSet<Course>`。EF Core 隐式包含了它们，因为 `Student` 实体引用 `Enrollment` 实体，而 `Enrollment` 实体引用 `Course` 实体。在本教程中，将 `DbSet<Enrollment>` 和 `DbSet<Course>` 保留在 `SchoolContext` 中。

SQL Server Express LocalDB

连接字符串指定 SQL Server LocalDB。LocalDB 是轻型版本 SQL Server Express 数据库引擎，专门针对应用开发，而非生产使用。LocalDB 作为按需启动并在用户模式下运行的轻量级数据库没有复杂的配置。默认情况下，LocalDB 会在 `C:/Users/<user>` 目录中创建 .mdf 数据库文件。

添加代码，以使用测试数据初始化该数据库

EF Core 会创建一个空的数据库。本部分中编写了 `Initialize` 方法来使用测试数据填充该数据库。

在 Data 文件夹中，新建一个名为 DbInitializer.cs 的类文件，并添加以下代码：

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Models
```

```
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            // context.Database.EnsureCreated();

            // Look for any students.
            if (context.Student.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
            new
Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-01")},
            new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-
09-01")},
            new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-
01")},
            new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-
09-01")},
            new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-
09-01")},
            new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-
09-01")}
            };
            foreach (Student s in students)
            {
                context.Student.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
            new Course{CourseID=1050,Title="Chemistry",Credits=3},
            new Course{CourseID=4022,Title="Microeconomics",Credits=3},
            new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
            new Course{CourseID=1045,Title="Calculus",Credits=4},
            new Course{CourseID=3141,Title="Trigonometry",Credits=4},
            new Course{CourseID=2021,Title="Composition",Credits=3},
            new Course{CourseID=2042,Title="Literature",Credits=4}
            };
            foreach (Course c in courses)
            {
                context.Course.Add(c);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
            new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
            new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
            new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
            new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
            new Enrollment{StudentID=3,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
            new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
            new Enrollment{StudentID=6,CourseID=1045},
            new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
```

```
            };
            foreach (Enrollment e in enrollments)
            {
                context.Enrollment.Add(e);
            }
            context.SaveChanges();
        }
    }
}
```

该代码会检查数据库中是否存在任何学生。如果 DB 中没有任何学生，则会使用测试数据初始化该 DB。代码中使用数组存放测试数据而不是使用 `List<T>` 集合是为了优化性能。

`EnsureCreated` 方法自动为数据库上下文创建数据库。如果数据库已存在，则返回 `EnsureCreated`，并且不修改数据库。

在 Program.cs 中，将 `Main` 方法修改为调用 `Initialize`：

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateWebHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;

            try
            {
                var context = services.GetRequiredService<SchoolContext>();
                // using ContosoUniversity.Data;
                DbInitializer.Initialize(context);
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred creating the DB.");
            }
        }

        host.Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

删除任何学生记录并重启应用。如果未初始化 DB，则在 `Initialize` 中设置断点以诊断问题。

查看数据库

从 Visual Studio 中的“视图”菜单打开 SQL Server 对象资源管理器 (SSOX)。在 SSOX 中，单击“(localdb)\MSSQLLocalDB”>“数据库”>“ContosoUniversity1”。

展开“表”节点。

右键单击 Student 表，然后单击“查看数据”，以查看创建的列和插入到表中的行。

异步代码

异步编程是 ASP.NET Core 和 EF Core 的默认模式。

Web 服务器的可用线程是有限的，而在高负载情况下的可能所有线程都被占用。当发生这种情况的时候，服务器就无法处理新请求，直到线程被释放。使用同步代码时，可能会出现多个线程被占用但不能执行任何操作的情况，因为它们正在等待 I/O 完成。使用异步代码时，当进程正在等待 I/O 完成，服务器可以将其线程释放用于处理其他请求。因此，使用异步代码可以更有效地利用服务器资源，并且可以让服务器在没有延迟的情况下处理更多流量。

异步代码会在运行时引入少量开销。流量较低时，对性能的影响可以忽略不计，但流量较高时，潜在的性能改善非常显著。

在以下代码中，async 关键字和 `Task<T>` 返回值，`await` 关键字和 `ToListAsync` 方法让代码异步执行。

```
public async Task OnGetAsync()
{
    Student = await _context.Student.ToListAsync();
}
```

- `async` 关键字让编译器执行以下操作：
  - 为方法主体的各部分生成回调。
  - 自动创建返回的 Task 对象。有关详细信息，请参阅任务返回类型。
- 隐式返回类型 `Task` 表示正在进行的工作。
- `await` 关键字让编译器将该方法拆分为两个部分。第一部分是以异步方式结束已启动的操作。第二部分是当操作完成时注入调用回调方法的地方。
- `ToListAsync` 是 `ToList` 扩展方法的异步版本。

编写使用 EF Core 的异步代码时需要注意的一些事项：

- 只会异步执行导致查询或命令被发送到数据库的语句。这包括 `ToListAsync`、`SingleOrDefaultAsync`、`FirstOrDefaultAsync` 和 `SaveChangesAsync`。不包括只会更改 `IQueryable` 的语句，例如 `var students = context.Students.Where(s => s.LastName == "Davolio")`。
- EF Core 上下文并非线程安全：请勿尝试并行执行多个操作。
- 若要利用异步代码的性能优势，请验证在调用向数据库发送查询的 EF Core 方法时，库程序包（如用于分页）是否使用异步。

有关 .NET 中异步编程的详细信息，请参阅异步概述和使用 Async 和 Await 的异步编程。

下一个教程将介绍基本的 CRUD（创建、读取、更新、删除）操作。

下一页

ASP.NET Core 中的 Razor 页面和 EF Core - CRUD - 第 2 个教程(共 8 个)

2018/10/29 • • Edit Online

可在此 PDF 文件中找到本教程的 ASP.NET Core 2.0 版。

相对于 2.0 版，本教程的 ASP.NET Core 2.1 版有很多改进。

作者：Tom Dykstra、Jon P Smith 和 Rick Anderson

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程，请参阅第一个教程。

本教程将介绍和自定义已搭建基架的 CRUD（创建、读取、更新、删除）代码。

为最大程度降低复杂性并让这些教程集中介绍 EF Core，将在页面模型中使用 EF Core 代码。某些开发人员使用服务层或存储库模式在 UI（Razor 页面）和数据访问层之间创建抽象层。

本教程将检查"学生"文件夹中的"创建"、"编辑"、"删除"和"详细信息"Razor Pages。

基架代码将以下模式用于"创建"、"编辑"和"删除"页面：

- 使用 HTTP GET 方法 `OnGetAsync` 获取和显示请求数据。
- 使用 HTTP POST 方法 `OnPostAsync` 将更改保存到数据。

"索引"和"详细信息"页面使用 HTTP GET 方法 `OnGetAsync` 获取和显示请求数据

SingleOrDefaultAsync 与FirstOrDefaultAsync

生成的代码使用 FirstOrDefaultAsync其推荐度通常高于 SingleOrDefaultAsync。

提取一个实体时，使用 `FirstOrDefaultAsync` 比使用 `SingleOrDefaultAsync` 更高效：

- 代码需要验证查询仅返回一个实体时除外。
- `SingleOrDefaultAsync` 会提取更多数据并执行不必要的工作。
- 如果有多个实体符合筛选部分，`SingleOrDefaultAsync` 将引发异常。
- 如果有多个实体符合筛选部分，`FirstOrDefaultAsync` 不引发异常。

FindAsync

在大部分基架代码中，FindAsync 可用于替代 `FirstOrDefaultAsync`。

`FindAsync`：

- 查找具有主键 (PK) 的实体。如果具有 PK 的实体正在由上下文跟踪，会返回该实体且不向 DB 发出请求。
- 既简单又简洁。
- 经过优化后可查找单个实体。
- 在某些情况下可以提供性能优势，但很少发生在典型的 Web 应用中。
- 以隐式方式使用 FirstAsync 而不是 SingleAsync。

如果想要 `Include` 其他实体，则 `FindAsync` 将不再适用。这意味着可能需要放弃 `FindAsync` 并随着应用运行移动到查询。

自定义“详细信息”页

浏览到 `Pages/Students` 页面。“编辑”、“详细信息”和“删除”链接是在 Pages/Students/Index.cshtml 文件中由定位点标记帮助器生成的。

```
<td>
    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
</td>
```

运行应用并选择“详细信息”链接。URL 的格式为 `http://localhost:5000/Students/Details?id=2`。“学生 ID”通过查询字符串（`?id=2`）进行传递。

更新“编辑”、“详细信息”和“删除”Razor 页面以使用 `"{id:int}"` 路由模板。将上述每个页面的页面指令从 `@page` 更改为 `@page "{id:int}"`。

如果对具有不包含整数路由值的“{id:int}”路由模板的页面发起请求，则该请求将返回 HTTP 404（找不到）错误。例如，`http://localhost:5000/Students/Details` 返回 404 错误。若要使 ID 可选，请将 `?` 追加到路由约束：

```
@page "{id:int?}"
```

运行应用，单击“详细信息”链接，并验证确认 URL 正在将 ID 作为路由数据（`http://localhost:5000/Students/Details/2`）进行传递。

不要将 `@page` 全局更改为 `@page "{id:int}"`，执行此操作会将链接拆分为“主页”和“创建”页。

添加相关数据

“学生索引”页的基架代码不包括 `Enrollments` 属性。在本部分，`Enrollments` 集合的内容显示在“详细信息”页中。

Pages/Students/Details.cshtml.cs 的 `OnGetAsync` 方法使用 `FirstOrDefaultAsync` 方法检索单个 `Student` 实体。添加以下突出显示的代码：

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Student
                        .Include(s => s.Enrollments)
                            .ThenInclude(e => e.Course)
                        .AsNoTracking()
                        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

Include 和 ThenInclude 方法使上下文加载 `Student.Enrollments` 导航属性，并在每个注册中加载 `Enrollment.Course` 导航属性。这些方法将在与数据读取相关的教程中进行详细介绍。

对于返回的实体未在当前上下文中更新的情况，AsNoTracking 方法将会提升性能。`AsNoTracking` 将在本教程的后续部分中讨论。

在“详细信息”页中显示相关注册

打开 Pages/Students/Details.cshtml。添加以下突出显示的代码以显示注册列表：

```
@page "{id:int}"
@model ContosoUniversity.Pages.Students.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd>
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

如果代码缩进在粘贴代码后出现错误，请按 CTRL-K-D 进行更正。

上面的代码循环通过 `Enrollments` 导航属性中的实体。它将针对每个注册显示课程标题和成绩。课程标题从

Course 实体中检索，该实体存储在 Enrollments 实体的 `Course` 导航属性中。

运行应用，选择"学生"选项卡，然后单击学生的"详细信息"链接。随即显示出所选学生的课程和成绩列表。

更新"创建"页

将 Pages/Students/Create.cshtml.cs 中的 `OnPostAsync` 方法更新为以下代码：

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Student.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return null;
}
```

TryUpdateModelAsync

检查 TryUpdateModelAsync 代码：

```
var emptyStudent = new Student();

if (await TryUpdateModelAsync<Student>(
    emptyStudent,
    "student",   // Prefix for form value.
    s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
{
```

在前面的代码中，`TryUpdateModelAsync<Student>` 尝试使用 PageModel 的 PageContext 属性中已发布的表单值更新 `emptyStudent` 对象。`TryUpdateModelAsync` 仅更新列出的属性（`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`）。

在上述示例中：

- 第二个自变量（`"student", // Prefix`）是用于查找值的前缀。该自变量不区分大小写。
- 已发布的表单值通过模型绑定转换为 `Student` 模型中的类型。

过多发布

使用 `TryUpdateModel` 更新具有已发布值的字段是一种最佳的安全做法，因为这能阻止过多发布。例如，假设 Student 实体包含此网页不应更新或添加的 `Secret` 属性：

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

即使应用的创建/更新 Razor 页面上没有 `Secret` 字段，黑客仍可利用过多发布设置 `Secret` 值。黑客也可使用 Fiddler 等工具或通过编写某个 JavaScript 来发布 `Secret` 表单值。原始代码不会限制模型绑定器在创建“学生”实例时使用的字段。

黑客为 `Secret` 表单字段指定的任何值都会在 DB 中更新。下图显示 Fiddler 工具正在将 `Secret` 字段（值为“OverPost”）添加到已发布的表单值。

值“OverPost”已成功添加到所插入行的 `Secret` 属性中。应用程序设计器绝不会在“创建”页设置 `Secret` 属性。

视图模型

视图模型通常包含应用程序所用的模型中包括的属性的子集。应用程序模型通常称为域模型。域模型通常包含 DB 中对应实体所需的全部属性。视图模型仅包含 UI 层（例如“创建”页）所需的属性。除视图模型外，某些应用使用绑定模型或输入模型在“Razor 页面”页面模型类和浏览器之间传递数据。请考虑以下 `Student` 视图模型：

```
using System;

namespace ContosoUniversity.Models
{
    public class StudentVM
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }
    }
}
```

视图模型还提供了一种防止过度发布的方法。视图模型仅包含要查看（显示）或更新的属性。

以下代码使用 `StudentVM` 视图模型创建新的学生：

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

SetValues 方法通过从另一个 PropertyValues 对象读取值来设置此对象的值。`SetValues` 使用属性名称匹配。视图模型类型不需要与模型类型相关，它只需要具有匹配的属性。

使用 `StudentVM` 时需要更新 CreateVM.cshtml 才能使用 `StudentVM` 而非 `Student`。

在 Razor 页面，`PageModel` 派生类就是视图模型。

更新“编辑”页

更新“编辑”页的页面模型。突出显示所作的主要更改：

```
public class EditModel : PageModel
{
    private readonly SchoolContext _context;

    public EditModel(SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Student Student { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Student = await _context.Student.FindAsync(id);

        if (Student == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var studentToUpdate = await _context.Student.FindAsync(id);

        if (await TryUpdateModelAsync<Student>(
            studentToUpdate,
            "student",
            s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
        {
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }

        return Page();
    }
}
```

代码更改与“创建”页类似，但有少数例外：

- `OnPostAsync` 具有可选的 `id` 参数。
- 当前学生是从 DB 提取的，而非通过创建空学生获得。
- 已将 `FirstOrDefaultAsync` 替换为 FindAsync。从主键中选择实体时，使用 `FindAsync` 是一个不错的选择。请参阅 FindAsync 了解详细信息。

测试“编辑”和“创建”页

创建和编辑几个学生实体。

实体状态

DB 上下文会随时跟踪内存中的实体是否已与其在 DB 中的对应行进行同步。DB 上下文同步信息可决定调用

SaveChangesAsync 后的行为。例如，将新实体传递到 AddAsync 方法时，该实体的状态设置为 Added。调用 `SaveChangesAsync` 时，DB 上下文会发出 SQL INSERT 命令。

实体可能处于以下状态之一：

- `Added`：DB 中尚不存在实体。`SaveChanges` 方法发出 INSERT 语句。
- `Unchanged`：无需保存对该实体所作的任何更改。从 DB 中读取实体时，该实体将具有此状态。
- `Modified`：已修改实体的部分或全部属性值。`SaveChanges` 方法发出 UPDATE 语句。
- `Deleted`：已标记该实体进行删除。`SaveChanges` 方法发出 DELETE 语句。
- `Detached`：DB 上下文未跟踪该实体。

在桌面应用中，通常会自动设置状态更改。读取实体并执行更改后，实体状态自动更改为 `Modified`。调用 `SaveChanges` 会生成仅更新已更改属性的 SQL UPDATE 语句。

在 Web 应用中，读取实体并显示数据的 `DbContext` 将在页面呈现后进行处理。调用页面 `OnPostAsync` 方法时，将发出具有 `DbContext` 的新实例的 Web 请求。如果在这个新的上下文中重新读取实体，则会模拟桌面处理。

更新“删除”页

在此部分中，当对 `SaveChanges` 的调用失败时，将添加用于实现自定义错误消息的代码。添加字符串，使其包含可能的错误消息：

```
public class DeleteModel : PageModel
{
    private readonly SchoolContext _context;

    public DeleteModel(SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Student Student { get; set; }
    public string ErrorMessage { get; set; }
```

将 `OnGetAsync` 方法替换为以下代码：

```
public async Task<IActionResult> OnGetAsync(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Student
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ErrorMessage = "Delete failed. Try again";
    }

    return Page();
}
```

上述代码包含可选参数 `saveChangesError`。`saveChangesError` 指示学生对象删除失败后是否调用该方法。删除操作可能由于暂时性网络问题而失败。云端更可能出现暂时性网络错误。通过 UI 调用"删除"页 `OnGetAsync` 时，`saveChangesError` 为 false。当 `OnPostAsync` 调用 `OnGetAsync`（由于删除操作失败）时，`saveChangesError` 参数为 true。

"删除"页 OnPostAsync 方法

将 `OnPostAsync` 替换为以下代码：

```
public async Task<IActionResult> OnPostAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Student
                    .AsNoTracking()
                    .FirstOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    try
    {
        _context.Student.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction("./Delete",
                              new { id, saveChangesError = true });
    }
}
```

上述代码检索所选的实体，然后调用 Remove 方法，将实体的状态设置为 `Deleted`。调用 `SaveChanges` 时生成

SQL DELETE 命令。如果 `Remove` 失败：

- 会捕获 DB 异常。
- 通过 `saveChangesError=true` 调用“删除”页 `OnGetAsync` 方法。

更新“删除”Razor 页面

将以下突出显示的错误消息添加到“删除”Razor 页面。

```
@page "{id:int}"
@model ContosoUniversity.Pages.Students.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
```

测试“删除”。

常见错误

“学生/索引”或其他链接不起作用：

验证确认 Razor 页面包含正确的 `@page` 指令。例如，“学生/索引”Razor Pages 不得包含路由模板：

```
@page "{id:int}"
```

每个 Razor 页面均必须包含 `@page` 指令。

上一页 下一页

ASP.NET Core 中的 Razor 页面和 EF Core - 排序、筛选、分页 - 第 3 个教程(共 8 个)

2018/10/30 • • Edit Online

可在此 PDF 文件中找到本教程的 ASP.NET Core 2.0 版。

相对于 2.0 版，本教程的 ASP.NET Core 2.1 版有很多改进。

作者：Tom Dykstra、Rick Anderson 和 Jon P Smith

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程，请参阅第一个教程。

本教程将添加排序、筛选、分组和分页功能。

下图显示完整的页面。列标题是可单击的链接，可用于对列进行排序。重复单击列标题可在升降和降序排序顺序之间切换。

如果遇到无法解决的问题，请下载已完成应用。

向索引页添加排序

向 Students/Index.cshtml.cs `PageModel` 添加字符串，使其包含排序参数：

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;

    public IndexModel(SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }
```

用以下代码更新 Students/Index.cshtml.cs `OnGetAsync`：

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

上述代码接收来自 URL 中的查询字符串的 `sortOrder` 参数。该 URL（包括查询字符串）由定位点标记帮助器生成

`sortOrder` 参数为"名称"或"日期"。`sortOrder` 参数后面可跟"_desc"以指定降序（可选）。默认排序顺序为升序。

如果通过"学生"链接对"索引"页发起请求，则不会有任何查询字符串。学生按姓氏升序显示。按姓氏升序是 `switch` 语句中的默认顺序 (fall-through case)。用户单击列标题链接时，查询字符串值中会提供相应的 `sortOrder` 值。

Razor 页面使用 `NameSort` 和 `DateSort` 为列标题超链接配置相应的查询字符串值：

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

以下代码包含 C# 条件 ?: 运算符：

```
NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";
```

第一行指定当 `sortOrder` 为 NULL 或为空时，`NameSort` 设置为"name_desc"。如果 `sortOrder` 不为 NULL 或不为空，则 `NameSort` 设置为空字符串。

`?: operator` 也称为三元运算符。

通过这两个语句，页面可如下设置列标题超链接：

当前排序顺序	姓氏超链接	日期超链接
姓氏升序	descending	ascending
姓氏降序	ascending	ascending
日期升序	ascending	descending
日期降序	ascending	ascending

该方法使用 LINQ to Entities 指定要作为排序依据的列。此代码会初始化 switch 语句前面的 `IQueryable<Student>`，并在 switch 语句中对其进行修改：

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

创建或修改 `IQueryable` 时，不会向数据库发送任何查询。将 `IQueryable` 对象转换成集合后才能执行查询。通过调用 `IQueryable` 等方法可将 `ToListAsync` 转换成集合。因此，`IQueryable` 代码会生成单个查询，此查询直到出现以下语句才执行：

```
Student = await studentIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` 可能获得包含大量可排序列的详细信息。

向“学生索引”页添加列标题超链接

将 Students/Index.cshtml 中的代码替换为以下突出显示的代码：

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Student[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Student)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

前面的代码:

- 向 `LastName` 和 `EnrollmentDate` 列标题添加超链接。
- 使用 `NameSort` 和 `DateSort` 中的信息为超链接设置当前的排序顺序值。

若要验证排序是否生效:

- 运行应用并选择“学生”选项卡。
- 单击“姓氏”。
- 单击“注册日期”。

若要更好地了解此代码：

- 请在 Student/Index.cshtml.cs 中的 `switch (sortOrder)` 上设置断点。
- 添加对 `NameSort` 和 `DateSort` 的监视。
- 请在 Student/Index.cshtml.cs 中的 `@Html.DisplayNameFor(model => model.Student[0].LastName)` 上设置断点。

单步执行调试程序。

向“学生索引”页添加搜索框

若要向“学生索引”页添加筛选：

- 需要向 Razor 页面添加一个文本框和一个提交按钮。文本框会针对名字或姓氏提供一个搜索字符串。
- 页面模型随即更新以使用文本框值。

向 Index 方法添加筛选功能

用以下代码更新 Students/Index.cshtml.cs `OnGetAsync`：

```
public async Task OnGetAsync(string sortOrder, string searchString)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

前面的代码：

- 向 `OnGetAsync` 方法添加 `searchString` 参数。从下一部分中添加的文本框中所接收搜索字符串值。
- 已向 LINQ 语句添加 `Where` 子句。`Where` 子句仅选择其名字或姓氏中包含搜索字符串的学生。只有存在要搜索的值时才执行 LINQ 语句。

请注意：上述代码调用 `IQueryable` 对象上的 `Where` 方法，且在服务器上处理该筛选器。在某些情况下，应用可能会对内存中的集合调用 `Where` 方法作为扩展方法。例如，假设 `_context.Students` 从 EF Core `DbSet` 更改为可返回 `IEnumerable` 集合的存储库方法。结果通常是相同的，但在某些情况下可能不同。

例如，`Contains` 的 .NET Framework 实现会默认执行区分大小写的比较。在 SQL Server 中，`Contains` 区分大小写由 SQL Server 实例的排序规则设置决定。SQL Server 默认为不区分大小写。可调用 `ToUpper`，进行不区分大小写的显式测试：

`Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())`

如果上述代码改为使用 `IEnumerable`，则该代码会确保结果区分大小写。如果在 `IEnumerable` 集合上调用 `Contains`，则使用 .NET Core 实现。如果在 `IQueryable` 对象上调用 `Contains`，则使用数据库实现。从存储库返回 `IEnumerable` 可能会大幅降低性能：

1. 所有行均从 DB 服务器返回。
2. 筛选应用于应用程序中所有返回的行。

调用 `ToUpper` 不会对性能产生负面影响。`ToUpper` 代码会在 TSQL SELECT 语句的 WHERE 子句中添加一个函数。添加的函数会防止优化器使用索引。如果安装的 SQL 区分大小写，则最好避免在不必要时调用 `ToUpper`。

向“学生索引”页添加搜索框

在 Pages/Students/Index.cshtml中，添加以下突出显示的代码以创建“搜索”按钮和各种 chrome。

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
```

上述代码使用 `<form>` 标记帮助器来添加搜索文本框和按钮。默认情况下，`<form>` 标记帮助器利用 POST 提交表单数据。借助 POST，会在 HTTP 消息正文中而不是在 URL 中传递参数。使用 HTTP GET 时，表单数据作为查询字符串在 URL 中进行传递。通过查询字符串传递数据时，用户可对 URL 添加书签。W3C 指南建议应在操作不引起更新的情况下使用 GET。

测试应用：

- 选择“学生”选项卡并输入搜索字符串。
- 选择“搜索”。

请注意，该 URL 包含搜索字符串。

```
http://localhost:5000/Students?SearchString=an
```

如果页面具有书签，该书签将包含该页面的 URL 和 `SearchString` 查询字符串。`form` 标记中的 `method="get"` 会导致生成查询字符串。

目前，选中列标题排序链接时，“搜索”框中的筛选值会丢失。丢失的筛选值在下一部分进行修复。

向“学生索引”页添加分页功能

本部分将创建一个 `PaginatedList` 类来支持分页。`PaginatedList` 类使用 `Skip` 和 `Take` 语句在服务器上筛选数据，而不是检索所有表格行。下图显示了分页按钮。

在项目文件夹中，使用以下代码创建 `PaginatedList.cs`：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(
            IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip(
                (pageIndex - 1) * pageSize)
                .Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

上述代码中的 `CreateAsync` 方法会提取页面大小和页码，并将相应的 `Skip` 和 `Take` 语句应用于 `IQueryable`。当在 `IQueryable` 上调用 `ToListAsync` 时，它将返回仅包含所请求页的列表。属性 `HasPreviousPage` 和 `HasNextPage` 用于启用或禁用"上一页"和"下一页"分页按钮。

`CreateAsync` 方法用于创建 `PaginatedList<T>`。构造函数不能创建 `PaginatedList<T>` 对象；构造函数不能运行异步代码。

向 Index 方法添加分页功能

在 Students/Index.cshtml.cs 中，将 `Student` 的类型从 `IList<Student>` 更新到 `PaginatedList<Student>`：

```
public PaginatedList<Student> Student { get; set; }
```

用以下代码更新 Students/Index.cshtml.cs `OnGetAsync`：

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
{
    CurrentSort = sortOrder;
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    if (searchString != null)
    {
        pageIndex = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                                || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    Student = await PaginatedList<Student>.CreateAsync(
        studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
}
```

上述代码会向方法签名添加页面索引、当前的 `sortOrder` 和 `currentFilter`。

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
```

出现以下情况时，所有参数均为 NULL：

- 从"学生"链接调用页面。
- 用户尚未单击分页或排序链接。

单击分页链接后，页面索引变量将包含要显示的页码。

`CurrentSort` 为 Razor 页面提供当前排序顺序。必须在分页链接中包含当前排序顺序才能在分页时保留排序顺序。

`CurrentFilter` 为 Razor 页面提供当前的筛选字符串。`CurrentFilter` 值：

- 必须包含在分页链接中才能在分页过程中保留筛选设置。
- 必须在重新显示页面时还原到文本框。

如果在分页时更改搜索字符串，页码会重置为 1。页面必须重置为 1，因为新的筛选器会导致显示不同的数据。输入搜索值并选择“提交”时：

- 搜索字符串将会更改。
- `searchString` 参数不为 NULL。

```
if (searchString != null)
{
    pageIndex = 1;
}
else
{
    searchString = currentFilter;
}
```

`PaginatedList.CreateAsync` 方法会将学生查询转换为支持分页的集合类型中的单个学生页面。单个学生页面会传递到 Razor 页面。

```
Student = await PaginatedList<Student>.CreateAsync(
    studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
```

`PaginatedList.CreateAsync` 中的两个问号表示 NULL 合并运算符。NULL 合并运算符定义可为 NULL 的类型的默认值。`(pageIndex ?? 1)` 表达式表示返回 `pageIndex` 的值（若带有值）。如果 `pageIndex` 没有值，则返回 1。

向“学生”Razor 页面添加分页链接

更新 Students/Index.cshtml 中的标记。突出显示所作更改：

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Student[0].LastName)
```

```
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Student)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@{
    var prevDisabled = !Model.Student.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.Student.HasNextPage ? "disabled" : "";
}

<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

列标题链接使用查询字符串将当前搜索字符串传递到 `OnGetAsync` 方法，让用户可对筛选结果进行排序：

```
<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Student[0].LastName)
</a>
```

分页按钮由标记帮助器显示：

```
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

运行应用并导航到学生页面。

- 为确保分页生效，请单击不同排序顺序的分页链接。
- 要验证确保分页后可正确地排序和筛选，请输入搜索字符串并尝试分页。

若要更好地了解此代码：

- 请在 Student/Index.cshtml.cs 中的 `switch (sortOrder)` 上设置断点。
- 添加对 `NameSort`、`DateSort`、`CurrentSort` 和 `Model.Student.PageIndex` 的监视。
- 请在 Student/Index.cshtml.cs 中的 `@Html.DisplayNameFor(model => model.Student[0].LastName)` 上设置断点。

单步执行调试程序。

更新“关于”页以显示学生统计信息

此步骤将更新 Pages/About.cshtml，显示每个注册日期的已注册学生的数量。更新需使用分组并包括以下步骤：

- 为“关于”页使用的数据创建视图模型。
- 更新“关于”页以使用视图模型。

创建视图模型

在 Models 文件夹中创建一个 SchoolViewModels 文件夹。

在 SchoolViewModels 文件夹中，使用以下代码添加 EnrollmentDateGroup.cs：

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

更新“关于”页面模型

用以下代码更新 Pages/About.cshtml.cs 文件：

```
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ContosoUniversity.Models;
using ContosoUniversity.Data;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

        public IList<EnrollmentDateGroup> Student { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Student
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Student = await data.AsNoTracking().ToListAsync();
        }
    }
}
```

LINQ 语句按注册日期对学生实体进行分组，计算每组中实体的数量，并将结果存储在 `EnrollmentDateGroup` 视图模型对象的集合中。

修改“关于”Razor 页面

将 Pages/About.cshtml 文件中的代码替换为以下代码：

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model.Student)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

运行应用并导航到“关于”页面。表格中会显示每个注册日期的学生计数。

如果遇到无法解决的问题，请下载本阶段的已完成应用。

其他资源

- 调试 ASP.NET Core 2.x 源

在下一教程中，应用将利用迁移更新数据模型。

ASP.NET Core 中的 Razor 页面和 EF Core - 迁移 - 第 4 个教程(共 8 个)

2018/7/16 • • Edit Online

可在此 PDF 文件中找到本教程的 ASP.NET Core 2.0 版。

相对于 2.0 版，本教程的 ASP.NET Core 2.1 版有很多改进。

作者：Tom Dykstra、Jon P Smith 和 Rick Anderson

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程，请参阅第一个教程。

本教程使用 EF Core 迁移功能管理数据模型更改。

如果遇到无法解决的问题，请下载已完成应用。

开发新应用时，数据模型会频繁更改。每当模型发生更改时，都无法与数据库进行同步。本教程首先配置 Entity Framework 以创建数据库(如果不存在)。每当数据模型发生更改时：

- DB 都会被删除。
- EF 都会创建一个新数据库来匹配该模型。
- 应用使用测试数据为 DB 设定种子。

这种使 DB 与数据模型保持同步的方法适用于多种情况，但将应用部署到生产环境的情况除外。当应用在生产环境中运行时，应用通常会存储需要保留的数据。每当发生更改(例如添加新列)时，应用都无法在具有测试 DB 的环境下启动。EF Core 迁移功能可通过使 EF Core 更新 DB 架构而不是创建新 DB 来解决此问题。

数据模型发生更改时，迁移将更新架构并保留现有数据，而无需删除或重新创建 DB。

删除数据库

使用 SQL Server 对象资源管理器 (SSOX) 或 `database drop` 命令：

- Visual Studio
- .NET Core CLI

在"包管理器控制台"(PMC) 中运行以下命令：

```
Drop-Database
```

从 PMC 运行 `Get-Help about_EntityFrameworkCore`，获取帮助信息。

创建初始迁移并更新 DB

生成项目并创建第一个迁移。

- Visual Studio
- .NET Core CLI

```
Add-Migration InitialCreate
Update-Database
```

了解 Up 和 Down 方法

EF Core `migrations add` 命令已生成用于创建 DB 的代码。此迁移代码位于 Migrations<timestamp>_InitialCreate.cs 文件中。`InitialCreate` 类的 `Up` 的方法创建与数据模型实体集相对应的 DB 表。`Down` 方法删除这些表，如下例所示：

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true),
                Credits = table.Column<int>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        migrationBuilder.CreateTable(
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");

        migrationBuilder.DropTable(
            name: "Course");

        migrationBuilder.DropTable(
            name: "Student");
    }
}
```

迁移调用 `Up` 方法为迁移实现数据模型更改。输入用于回退更新的命令时，迁移调用 `Down` 方法。

前面的代码适用于初始迁移。该代码是运行 `migrations add InitialCreate` 命令时创建的。迁移名称参数（本示例中为“InitialCreate”）用于指定文件名。迁移名称可以是任何有效的文件名。最好选择能概括迁移中所执行操作的字词或短语。例如，添加了系表的迁移可称为“AddDepartmentTable”。

如果创建了初始迁移并且存在 DB：

- 会生成 DB 创建代码。
- DB 创建代码不需要运行，因为 DB 已与数据模型相匹配。即使 DB 创建代码运行也不会做出任何更改，因为 DB 已与数据模型相匹配。

如果将应用部署到新环境，则必须运行 DB 创建代码才能创建 DB。

先前删除了 DB，因此已不存在，所以迁移会创建新的 DB。

数据模型快照

迁移在 Migrations/SchoolContextModelSnapshot.cs 中创建当前数据库架构的快照。添加迁移时，EF 会通过将数据模型与快照文件进行对比来确定已更改的内容。

若要删除迁移，请使用以下命令：

- Visual Studio
- .NET Core CLI

Remove-Migration

删除迁移命令会删除迁移并确保正确重置快照。

删除 EnsureCreated 并测试应用

早期开发使用了 `EnsureCreated`。本教程将使用迁移。`EnsureCreated` 具有以下限制：

- 绕过迁移并创建 DB 和架构。
- 不会创建迁移表。
- 不能与迁移一起使用。
- 专门用于在频繁删除并重新创建 DB 的情况下进行测试或快速制作原型。

删除 `DbInitializer` 中的以下行：

```
context.Database.EnsureCreated();
```

运行应用并验证 DB 设定为种子。

检查数据库

使用 SQL Server 对象资源管理器检查 DB。请注意，增加了 `__EFMigrationsHistory` 表。`__EFMigrationsHistory` 表跟踪已应用到 DB 的迁移。查看 `__EFMigrationsHistory` 表中的数据，其中显示对应初始迁移的一行数据。上面的 CLI 输出示例中最后部分的日志显示了创建此行的 INSERT 语句。

运行应用并验证一切正常运行。

在生产环境中应用迁移

不建议生产应用在应用程序启动时调用 Database.Migrate。不应从服务器场中的应用调用 `Migrate`。例如，已将应用在云中部署为横向扩展（运行应用的多个示例）的情况。

应在部署过程中以受控的方式执行数据库迁移。生产数据库迁移方法包括：

- 使用迁移创建 SQL 脚本，并在部署过程中使用 SQL 脚本。
- 在受控的环境中运行 `dotnet ef database update`。

EF Core 使用 `__MigrationsHistory` 表查看是否需要运行任何迁移。如果 DB 已是最新，则无需运行迁移。

疑难解答

下载已完成应用。

应用会生成以下异常：

```
SqlException: Cannot open database "ContosoUniversity" requested by the login.
The login failed.
Login failed for user 'user name'.
```

解决方案：运行 `dotnet ef database update`

其他资源

- .NET Core CLI.
- 包管理器控制台 (Visual Studio)

ASP.NET Core 中的 Razor 页面和 EF Core - 数据模型 - 第 5 个教程(共 8 个)

2019/1/10 • • Edit Online

可在此 PDF 文件中找到本教程的 ASP.NET Core 2.0 版。

相对于 2.0 版，本教程的 ASP.NET Core 2.1 版有很多改进。

作者：Tom Dykstra 和 Rick Anderson

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程，请参阅第一个教程。

前面的教程介绍了由三个实体组成的基本数据模型。本教程将演示如何：

- 添加更多实体和关系。
- 通过指定格式设置、验证和数据库映射规则来自定义数据模型。

已完成数据模型的实体类如下图所示：

如果遇到无法解决的问题，请下载已完成应用。

使用特性自定义数据模型

此部分将使用特性自定义数据模型。

DataType 特性

学生页面当前显示注册日期。通常情况下，日期字段仅显示日期，不显示时间。

用以下突出显示的代码更新 *Models/Student.cs*：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

DataType 特性指定比数据库内部类型更具体的数据类型。在此情况下，应仅显示日期，而不是日期加时间。DataType 枚举提供多种数据类型，例如日期、时间、电话号码、货币、电子邮件地址等。应用还可通过 `DataType` 特性自动提供类型特定的功能。例如：

- `mailto:` 链接将依据 `DataType.EmailAddress` 自动创建。
- 大多数浏览器中都提供面向 `DataType.Date` 的日期选择器。

`DataType` 特性发出 HTML 5 `data-`（读作 data dash）特性供 HTML 5 浏览器使用。`DataType` 特性不提供验证。

`DataType.Date` 不指定显示日期的格式。默认情况下，日期字段根据基于服务器的 CultureInfo 的默认格式进行显示。

`DisplayFormat` 特性用于显式指定日期格式：

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

`ApplyFormatInEditMode` 设置指定还应对编辑 UI 应用该格式设置。某些字段不应使用 `ApplyFormatInEditMode`。例如，编辑文本框中通常不应显示货币符号。

`DisplayFormat` 特性可由其本身使用。搭配使用 `DataType` 特性和 `DisplayFormat` 特性通常是很好的做法。`DataType` 特性按照数据在屏幕上的呈现方式传达数据的语义。`DataType` 特性可提供 `DisplayFormat` 中所不具有的以下优点：

- 浏览器可启用 HTML5 功能。例如，显示日历控件、区域设置适用的货币符号、电子邮件链接、客户端输入验证等。
- 默认情况下，浏览器将根据区域设置采用正确的格式呈现数据。

有关详细信息，请参阅 <input> 标记帮助器文档。

运行应用。导航到学生索引页。将不再显示时间。使用 `Student` 模型的每个视图将显示日期，不显示时间。

StringLength 特性

可使用特性指定数据验证规则和验证错误消息。StringLength 特性指定数据字段中允许的字符的最小长度和最大长度。`StringLength` 特性还提供客户端和服务器端验证。最小值对数据库架构没有任何影响。

使用以下代码更新 `Student` 模型：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

上面的代码将名称限制为不超过 50 个字符。`StringLength` 特性不会阻止用户在名称中输入空格。
RegularExpression 特性用于向输入应用限制。例如，以下代码要求第一个字符为大写，其余字符按字母顺序排列：

```
[RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$")]
```

运行应用：

- 导航到学生页。

- 选择“新建”并输入不超过 50 个字符的名称。
- 选择“创建”时，客户端验证会显示一条错误消息。

在“SQL Server 对象资源管理器”(SSOX) 中，双击 Student 表，打开 Student 表设计器。

上图显示 `Student` 表的架构。名称字段的类型为 `nvarchar(MAX)`，因为数据库上尚未运行迁移。稍后在本教程中运行迁移时，名称字段将变成 `nvarchar(50)`。

Column 特性

特性可以控制类和属性映射到数据库的方式。在本部分，`Column` 特性用于将 `FirstMidName` 属性的名称映射到数据库中的“FirstName”。

创建数据库后，模型上的属性名将用作列名（使用 `Column` 特性时除外）。

`Student` 模型使用 `FirstMidName` 作为名字字段，因为该字段也可能包含中间名。

用以下突出显示的代码更新 *Student.cs* 文件：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

进行上述更改后，应用中的 `Student.FirstMidName` 将映射到 `Student` 表的 `FirstName` 列。

添加 `Column` 特性后，`SchoolContext` 的支持模型会发生改变。`SchoolContext` 的支持模型将不再与数据库匹配。如果在执行迁移前运行应用，则会生成如下异常：

```
SqlException: Invalid column name 'FirstName'.
```

若要更新数据库：

- 生成项目。
- 在项目文件夹中打开命令窗口。输入以下命令以创建新迁移并更新数据库：

- Visual Studio
- .NET Core CLI

```
Add-Migration ColumnFirstName
Update-Database
```

`migrations add ColumnFirstName` 命令将生成如下警告消息：

```
An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
```

生成警告的原因是名称字段现已限制为 50 个字符。如果数据库中的名称超过 50 个字符，则第 51 个字符及后面的所有字符都将丢失。

- 测试应用。

在 SSOX 中打开 Student 表：

执行迁移前，名称列的类型为 nvarchar (MAX)。名称列现在的类型为 `nvarchar(50)` 。列名已从 `FirstMidName` 更改为 `FirstName` 。

> **NOTE**
>
> 在下一部分中，在某些阶段生成应用会生成编译器错误。说明用于指定生成应用的时间。

Student 实体更新

用以下代码更新 *Models/Student.cs*：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Required 特性

`Required` 特性使名称属性成为必填字段。值类型（`DateTime`、`int`、`double`）等不可为 NULL 的类型不需要 `Required` 特性。系统会将不可为 NULL 的类型自动视为必填字段。

不能用 `StringLength` 特性中的最短长度参数替换 `Required` 特性：

```
[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }
```

Display 特性

`Display` 特性指定文本框的标题栏应为“FirstName”、“LastName”、“FullName”和“EnrollmentDate”。标题栏默认不使用空格分隔词语，如“Lastname”。

FullName 计算属性

`FullName` 是计算属性，可返回通过串联两个其他属性创建的值。`FullName` 不能设置并且仅具有一个 get 访问器。数据库中不会创建任何 `FullName` 列。

创建 Instructor 实体

用以下代码创建 Models/Instructor.cs：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

一行可包含多个特性。可按如下方式编写 `HireDate` 特性：

```
[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

CourseAssignments 和 OfficeAssignment 导航属性

`CourseAssignments` 和 `OfficeAssignment` 是导航属性。

一名讲师可以教授任意数量的课程，因此 `CourseAssignments` 定义为集合。

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

如果导航属性包含多个实体：

- 它必须是可在其中添加、删除和更新实体的列表类型。

导航属性类型包括：

- `ICollection<T>`
- `List<T>`
- `HashSet<T>`

如果指定了 `ICollection<T>`，EF Core 会默认创建 `HashSet<T>` 集合。

`CourseAssignment` 实体在"多对多关系"部分进行介绍。

Contoso University 业务规则规定一名讲师最多可获得一间办公室。`OfficeAssignment` 属性包含一个 `OfficeAssignment` 实体。如果未分配办公室，则 `OfficeAssignment` 为 NULL。

```
public OfficeAssignment OfficeAssignment { get; set; }
```

创建 OfficeAssignment 实体

用以下代码创建 Models/OfficeAssignment.cs：

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

Key 特性

`[Key]` 特性用于在属性名不是 classnameID 或 ID 时将属性标识为主键 (PK)。

`Instructor` 和 `OfficeAssignment` 实体之间存在一对零或一关系。仅当与分配到办公室的讲师之间建立关系时才存在办公室分配。`OfficeAssignment` PK 也是其到 `Instructor` 实体的外键 (FK)。EF Core 无法自动将 `InstructorID` 识别为 `OfficeAssignment` 的 PK，因为：

- `InstructorID` 不遵循 ID 或 classnameID 命名约定。

因此，`Key` 特性用于将 `InstructorID` 识别为 PK：

```
[Key]
public int InstructorID { get; set; }
```

默认情况下，EF Core 将键视为非数据库生成，因为该列面向的是识别关系。

Instructor 导航属性

`Instructor` 实体的 `OfficeAssignment` 导航属性可以为 NULL，因为：

- 引用类型（例如，类可以为 NULL）。
- 一名讲师可能没有办公室分配。

`OfficeAssignment` 实体具有不可为 NULL 的 `Instructor` 导航属性，因为：

- `InstructorID` 不可为 NULL。
- 没有讲师则不可能存在办公室分配。

当 `Instructor` 实体具有相关 `OfficeAssignment` 实体时，每个实体都具有对其导航属性中的另一个实体的引用。

`[Required]` 特性可以应用于 `Instructor` 导航属性：

```
[Required]
public Instructor Instructor { get; set; }
```

上面的代码指定必须存在相关的讲师。上面的代码没有必要，因为 `InstructorID` 外键（也是 PK）不可为 NULL。

修改 Course 实体

用以下代码更新 *Models/Course.cs*：

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

`Course` 实体具有外键 (FK) 属性 `DepartmentID`。`DepartmentID` 指向相关的 `Department` 实体。`Course` 实体具有 `Department` 导航属性。

当数据模型具有相关实体的导航属性时，EF Core 不要求此模型具有 FK 属性。

EF Core 可在数据库中的任何所需位置自动创建 FK。EF Core 为自动创建的 FK 创建阴影属性。数据模型中包含 FK 后可使更新更简单和更高效。例如，假设某个模型中不包含 FK 属性 `DepartmentID`。当提取 Course 实体进行编辑时：

- 如果未显式加载 `Department` 实体，则该实体将为 NULL。
- 若要更新 Course 实体，则必须先提取 `Department` 实体。

如果数据模型中包含 FK 属性 `DepartmentID`，则无需在更新前提取 `Department` 实体。

DatabaseGenerated 特性

`[DatabaseGenerated(DatabaseGeneratedOption.None)]` 特性指定 PK 由应用程序提供而不是由数据库生成。

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

默认情况下，EF Core 假定 PK 值由数据库生成。由数据库生成 PK 值通常是最佳方法。`Course` 实体的 PK 由用户指定。例如，对于课程编号，数学系可以使用 1000 系列的编号，英语系可以使用 2000 系列的编号。

`DatabaseGenerated` 特性还可用于生成默认值。例如，数据库可以自动生成日期字段以记录数据行的创建或更新日期。有关详细信息，请参阅生成的属性。

外键和导航属性

`Course` 实体中的外键 (FK) 属性和导航属性可反映以下关系：

课程将分配到一个系，因此将存在 `DepartmentID` FK 和 `Department` 导航属性。

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

参与一门课程的学生数量不定，因此 `Enrollments` 导航属性是一个集合：

```
public ICollection<Enrollment> Enrollments { get; set; }
```

一门课程可能由多位讲师讲授，因此 `CourseAssignments` 导航属性是一个集合：

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

`CourseAssignment` 在后文介绍。

创建 Department 实体

用以下代码创建 Models/Department.cs：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

Column 特性

`Column` 特性以前用于更改列名映射。在 `Department` 实体的代码中，`Column` 特性用于更改 SQL 数据类型映射。`Budget` 列通过数据库中的 SQL Server 货币类型进行定义：

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

通常不需要列映射。EF Core 通常基于属性的 CLR 类型选择相应的 SQL Server 数据类型。CLR `decimal` 类型会映射到 SQL Server `decimal` 类型。`Budget` 用于货币，但货币数据类型更适合货币。

外键和导航属性

FK 和导航属性可反映以下关系：

- 一个系可能有也可能没有管理员。
- 管理员始终由讲师担任。因此，`InstructorID` 属性作为到 `Instructor` 实体的 FK 包含在其中。

导航属性名为 `Administrator`，但其中包含 `Instructor` 实体：

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

上面代码中的问号 (?) 指定属性可以为 NULL。

一个系可以有多门课程，因此存在 Course 导航属性：

```
public ICollection<Course> Courses { get; set; }
```

注意:按照约定，EF Core 能针对不可为 NULL 的 FK 和多对多关系启用级联删除。级联删除可能导致形成循环级联删除规则。循环级联删除规则会在添加迁移时引发异常。

例如，如果未将 `Department.InstructorID` 属性定义为可以为 NULL：

- EF Core 会配置将在删除系时删除讲师的级联删除规则。
- 在删除系时删除讲师并不是预期行为。

如果业务规则要求 `InstructorID` 属性不可为 NULL，请使用以下 Fluent API 语句：

```
modelBuilder.Entity<Department>()
   .HasOne(d => d.Administrator)
   .WithMany()
   .OnDelete(DeleteBehavior.Restrict)
```

上面的代码会针对“系-讲师”关系禁用级联删除。

更新 Enrollment 实体

一份注册记录面向一名学生所注册的一门课程。

用以下代码更新 *Models/Enrollment.cs*：

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

外键和导航属性

FK 属性和导航属性可反映以下关系：

注册记录面向一门课程，因此存在 `CourseID` FK 属性和 `Course` 导航属性：

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

一份注册记录面向一门课程，因此存在 `StudentID` FK 属性和 `Student` 导航属性：

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

多对多关系

`Student` 和 `Course` 实体之间存在多对多关系。`Enrollment` 实体充当数据库中"具有有效负载"的多对多联接表。"具有有效负载"表示 `Enrollment` 表除了联接表的 FK 外还包含其他数据（本教程中为 PK 和 `Grade`）。

下图显示这些关系在实体关系图中的外观。（此关系图是使用适用于 EF 6.X 的 EF Power Tools 生成的。本教程不介绍如何创建此关系图。）

每条关系线的一端显示 1，另一端显示星号 (*)，这表示一对多关系。

如果 `Enrollment` 表不包含年级信息，则它只需包含两个 FK（`CourseID` 和 `StudentID`）。无有效负载的多对多联接表有时称为纯联接表 (PJT)。

`Instructor` 和 `Course` 实体具有使用纯联接表的多对多关系。

注意:EF 6.x 支持多对多关系的隐式联接表，但 EF Core 不支持。有关详细信息，请参阅 EF Core 2.0 中的多对多关系。

CourseAssignment 实体

用以下代码创建 Models/CourseAssignment.cs：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

讲师-课程

讲师-课程的多对多关系：

- 要求必须用实体集表示联接表。
- 为纯联接表（无有效负载的表）。

常规做法是将联接实体命名为 `EntityName1EntityName2`。例如，使用此模式的“讲师-课程”联接表是 `CourseInstructor`。但是，我们建议使用可描述关系的名称。

数据模型开始时很简单，其内容会逐渐增加。无有效负载联接 (PJT) 通常会发展为包含有效负载。该名称以描述性实体名称开始，因此不需要随联接表更改而更改。理想情况下，联接实体在业务域中可能具有自己的自带名称（可能是单个字）。例如，可以使用名为“比率”的联接实体链接“账目”和“客户”。对于“讲师-课程”多对多关系，建议使用 `CourseAssignment` 而不是 `CourseInstructor`。

组合键

FK 不能为 NULL。`CourseAssignment` 中的两个 FK（`InstructorID` 和 `CourseID`）共同唯一标识 `CourseAssignment`

表的每一行。`CourseAssignment` 不需要专用的 PK。`InstructorID` 和 `CourseID` 属性充当组合 PK。使用 Fluent API 是向 EF Core 指定组合 PK 的唯一方法。下一部分演示如何配置组合 PK。

组合键可确保：

- 允许一门课程对应多行。
- 允许一名讲师对应多行。
- 不允许相同的讲师和课程对应多行。

`Enrollment` 联接实体定义其自己的 PK，因此可能会出现此类重复。若要防止此类重复：

- 请在 FK 字段上添加唯一索引，或
- 配置具有主要组合键（与 `CourseAssignment` 类似）的 `Enrollment`。有关详细信息，请参阅索引。

更新数据库上下文

将以下突出显示的代码添加到 Data/SchoolContext.cs：

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Models
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollment { get; set; }
        public DbSet<Student> Student { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

上面的代码添加新实体并配置 `CourseAssignment` 实体的组合 PK。

用 Fluent API 替代特性

上面代码中的 `OnModelCreating` 方法使用 Fluent API 配置 EF Core 行为。API 称为“Fluent”，因为它通常在将一系列方法调用连接成单个语句后才能使用。下面的代码是 Fluent API 的示例：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

在本教程中，Fluent API 仅用于不能通过特性完成的数据库映射。但是，Fluent API 可以指定可通过特性完成的大多数格式设置、验证和映射规则。

`MinimumLength` 等特性不能通过 Fluent API 应用。`MinimumLength` 不会更改架构，它仅应用最小长度验证规则。

某些开发者倾向于仅使用 Fluent API 以保持实体类的"纯净"。特性和 Fluent API 可以相互混合。某些配置只能通过 Fluent API 完成（指定组合 PK）。有些配置只能通过特性完成（`MinimumLength`）。使用 Fluent API 或特性的建议做法：

- 选择以下两种方法之一。
- 尽可能以前后一致的方法使用所选的方法。

本教程中使用的某些特性可用于：

- 仅限验证（例如，`MinimumLength`）。
- 仅限 EF Core 配置（例如，`HasKey`）。
- 验证和 EF Core 配置（例如，`[StringLength(50)]`）。

有关特性和 Fluent API 的详细信息，请参阅配置方法。

显示关系的实体关系图

下图显示 EF Power Tools 针对已完成的学校模型创建的关系图。

上面的关系图显示：

- 几条一对多关系线（1 到 *）。
- `Instructor` 和 `OfficeAssignment` 实体之间的一对零或一关系线（1 到 0..1）。
- `Instructor` 和 `Department` 实体之间的零或一到多关系线（0..1 到 *）。

使用测试数据为数据库设定种子

更新 Data/DbInitializer.cs 中的代码：

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();
```

```
            // Look for any students.
            if (context.Student.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson",   LastName = "Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy",    LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura",    LastName = "Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-09-01") }
            };

            foreach (Student s in students)
            {
                context.Student.Add(s);
            }
            context.SaveChanges();

            var instructors = new Instructor[]
            {
                new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger",   LastName = "Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger",   LastName = "Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };

            foreach (Instructor i in instructors)
            {
                context.Instructors.Add(i);
            }
            context.SaveChanges();

            var departments = new Department[]
            {
                new Department { Name = "English",     Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Harui").ID },
                new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Kapoor").ID }
            };

            foreach (Department d in departments)
```

```
{
    context.Departments.Add(d);
}
context.SaveChanges();

var courses = new Course[]
{
    new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
    },
    new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
    new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
};

foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var officeAssignments = new OfficeAssignment[]
{
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
        Location = "Smith 17" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
        Location = "Gowan 27" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
        Location = "Thompson 304" },
};

foreach (OfficeAssignment o in officeAssignments)
{
    context.OfficeAssignments.Add(o);
}
context.SaveChanges();

var courseInstructors = new CourseAssignment[]
{
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
        },
```

```
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
            },
    };

    foreach (CourseAssignment ci in courseInstructors)
    {
        context.CourseAssignments.Add(ci);
    }
    context.SaveChanges();

    var enrollments = new Enrollment[]
    {
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            Grade = Grade.A
        },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
            Grade = Grade.C
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
            Grade = Grade.B
            }
```

```
                    },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Li").ID,
                    CourseID = courses.Single(c => c.Title == "Composition").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Justice").ID,
                    CourseID = courses.Single(c => c.Title == "Literature").CourseID,
                    Grade = Grade.B
                    }
            };

            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollment.Where(
                    s =>
                            s.Student.ID == e.StudentID &&
                            s.Course.CourseID == e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
                    context.Enrollment.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

前面的代码为新实体提供种子数据。大多数此类代码会创建新实体对象并加载示例数据。示例数据用于测试。有关如何对多对多联接表进行种子设定的示例，请参阅 `Enrollments` 和 `CourseAssignments`。

添加迁移

生成项目。

- Visual Studio
- .NET Core CLI

```
Add-Migration ComplexDataModel
```

前面的命令显示可能存在数据丢失的相关警告。

```
An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

如果运行 `database update` 命令，则会生成以下错误：

```
The ALTER TABLE statement conflicted with the FOREIGN KEY constraint
"FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in
database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.
```

应用迁移

现已有一个数据库，需要考虑如何将未来的更改应用到其中。本教程演示两种方法：

- 删除并重新创建数据库
- 将迁移应用到现有数据库。虽然此方法更复杂且耗时，但在实际应用和生产环境中为首选方法。说明：这是本教程的一个可选部分。你可以执行删除和重新创建的相关步骤并跳过此部分。如果希望执行本部分中的步骤，请勿执行删除和重新创建步骤。

删除并重新创建数据库

已更新 `DbInitializer` 中的代码将为新实体添加种子数据。若要强制 EF Core 创建新的 DB，请删除并更新 DB：

- Visual Studio
- .NET Core CLI

在“包管理器控制台”(PMC) 中运行以下命令：

```
Drop-Database
Update-Database
```

从 PMC 运行 `Get-Help about_EntityFrameworkCore`，获取帮助信息。

运行应用。运行应用后将运行 `DbInitializer.Initialize` 方法。`DbInitializer.Initialize` 将填充新数据库。

在 SSOX 中打开数据库：

- 如果之前已打开过 SSOX，请单击“刷新”按钮。
- 展开“表”节点。随后将显示出已创建的表。

查看 CourseAssignment 表：

- 右键单击 CourseAssignment 表，然后选择“查看数据”。
- 验证 CourseAssignment 表包含数据。

将迁移应用到现有数据库

本部分是可选的。只有当跳过之前的删除并重新创建数据库部分时才可以执行上述步骤。

当现有数据与迁移一起运行时，可能存在不满足现有数据的 FK 约束。使用生产数据时，必须采取步骤来迁移现有数据。本部分提供修复 FK 约束冲突的示例。务必在备份后执行这些代码更改。如果已完成上述部分并更新数据库，则不要执行这些代码更改。

{timestamp}_ComplexDataModel.cs 文件包含以下代码：

```
migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    type: "int",
    nullable: false,
    defaultValue: 0);
```

上面的代码将向 `Course` 表添加不可为 NULL 的 `DepartmentID` FK。前面教程中的数据库在 `Course` 中包含行，以便迁移时不会更新表。

若要使 `ComplexDataModel` 迁移可与现有数据搭配运行：

- 请更改代码以便为新列（`DepartmentID`）赋予默认值。
- 创建名为"临时"的虚拟系来充当默认的系。

修复外键约束

更新 `ComplexDataModel` 类 `Up` 方法：

- 打开 {timestamp}_ComplexDataModel.cs 文件。
- 对将 `DepartmentID` 列添加到 `Course` 表的代码行添加注释。

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

添加以下突出显示的代码。新代码在 `.CreateTable( name: "Department"` 块后：[!code-csharp]

经过上面的更改，`Course` 行将在 `ComplexDataModel` `Up` 方法运行后与“临时”系建立联系。

生产应用可能：

- 包含用于将 `Department` 行和相关 `Course` 行添加到新 `Department` 行的代码或脚本。
- 不会使用“临时”系或 `Course.DepartmentID` 的默认值。

下一教程将介绍相关数据。

上一页 下一页

ASP.NET Core 中的 Razor 页面和 EF Core - 读取相关数据 - 第 6 个教程(共 8 个)

2018/10/30 • • Edit Online

作者:Tom Dykstra、Jon P Smith 和 Rick Anderson

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程,请参阅第一个教程。

在本教程中,将读取和显示相关数据。相关数据为 EF Core 加载到导航属性中的数据。

如果遇到无法解决的问题,请下载或查看已完成的应用。下载说明。

下图显示了本教程中已完成的页面:

相关数据的预先加载、显式加载和延迟加载

EF Core 可采用多种方式将相关数据加载到实体的导航属性中：

- 预先加载。预先加载是指对查询某类型的实体时一并加载相关实体。读取实体时，会检索其相关数据。此时通常会出现单一联接查询，检索所有必需数据。EF Core 将针对预先加载的某些类型发出多个查询。与存在单一查询的 EF6 中的某些查询相比，发出多个查询可能更有效。预先加载通过 `Include` 和 `ThenInclude` 方法进行指定。

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

当包含集合导航时，预先加载会发送多个查询：

- 一个查询用于主查询
- 一个查询用于加载树中每个集合“边缘”。

- 使用 `Load` 的单独查询：可在单独的查询中检索数据，EF Core会“修复”导航属性。“修复”是指 EF Core 自动填充导航属性。与预先加载相比，使用 `Load` 的单独查询更像是显式加载。

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

请注意：EF Core 会将导航属性自动“修复”为之前加载到上下文实例中的任何其他实体。即使导航属性的数据非显式包含在内，但如果先前加载了部分或所有相关实体，则仍可能填充该属性。

- 显式加载。首次读取实体时，不检索相关数据。必须编写代码才能在需要时检索相关数据。使用单独查询进行显式加载时，会向数据库发送多个查询。该代码通过显式加载指定要加载的导航属性。使用 `Load` 方法进行显式加载。例如：

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- 延迟加载。延迟加载已添加到版本 2.1 中的 EF Core。首次读取实体时，不检索相关数据。首次访问导航属性时，会自动检索该导航属性所需的数据。首次访问导航属性时，都会向数据库发送一个查询。

- `Select` 运算符仅加载所需的相关数据。

创建显示院系名称的“课程”页

课程实体包括一个带 `Department` 实体的导航属性。`Department` 实体包含要分配课程的院系。

要在课程列表中显示已分配院系的名称：

- 从 `Department` 实体中获取 `Name` 属性。
- `Department` 实体来自于 `Course.Department` 导航属性。

为课程模型创建基架

- Visual Studio
- .NET Core CLI

按照为"学生"模型搭建基架中的说明操作，并对模型类使用 `Course`。

上述命令为 `Course` 模型创建基架。在 Visual Studio 中打开项目。

打开 Pages/Courses/Index.cshtml.cs 并检查 `OnGetAsync` 方法。基架引擎为 `Department` 导航属性指定了预先加载。`Include` 方法指定预先加载。

运行应用并选择"课程"链接。院系列显示 `DepartmentID`（该项无用）。

使用以下代码更新 `OnGetAsync` 方法：

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

上述代码添加了 `AsNoTracking`。由于未跟踪返回的实体，因此 `AsNoTracking` 提升了性能。未跟踪实体，因为未在当前上下文中更新这些实体。

使用以下突出显示的标记更新 Pages/Courses/Index.cshtml：

```
@page
@model ContosoUniversity.Pages.Courses.IndexModel
@{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Course)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

对基架代码进行了以下更改：

- 将标题从“索引”更改为“课程”。
- 添加了显示 `CourseID` 属性值的“数字”列。默认情况下，不针对主键进行架构，因为对最终用户而言，它们通常没有意义。但在此情况下主键是有意义的。
- 更改“院系”列，显示院系名称。该代码显示已加载到 `Department` 导航属性中的 `Department` 实体的 `Name` 属性：

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

运行应用并选择"课程"选项卡，查看包含系名称的列表。

使用 Select 加载相关数据

`OnGetAsync` 方法使用 `Include` 方法加载相关数据：

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

`Select` 运算符仅加载所需的相关数据。对于单个项（如 `Department.Name`），它使用 SQL INNER JOIN。对于集合，它使用另一个数据库访问，但集合上的 `Include` 运算符也是如此。

以下代码使用 `Select` 方法加载相关数据：

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
            .Select(p => new CourseViewModel
            {
                CourseID = p.CourseID,
                Title = p.Title,
                Credits = p.Credits,
                DepartmentName = p.Department.Name
            }).ToListAsync();
}
```

`CourseViewModel`：

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

有关完整示例的信息，请参阅 IndexSelect.cshtml 和 IndexSelect.cshtml.cs。

创建显示“课程”和“注册”的“讲师”页

在本部分中，将创建“讲师”页。

该页面通过以下方式读取和显示相关数据：

- 讲师列表显示 `OfficeAssignment` 实体(上图中的办公室)的相关数据。`Instructor` 和 `OfficeAssignment` 实体之间存在一对零或一的关系。预先加载适用于 `OfficeAssignment` 实体。需要显示相关数据时，预先加载通常更高效。在此情况下，会显示讲师的办公室分配。
- 当用户选择一名讲师(上图中的 Harui)时，显示相关的 `Course` 实体。`Instructor` 和 `Course` 实体之间存在多对多关系。对 `Course` 实体及其相关的 `Department` 实体使用预先加载。这种情况下，单独查询可能更有效，因为仅需显示所选讲师的课程。此示例演示如何在位于导航实体内的实体中预先加载这些导航实体。

- 当用户选择一门课程(上图中的化学)时，显示 `Enrollments` 实体的相关数据。上图中显示了学生姓名和成绩。`Course` 和 `Enrollment` 实体之间存在一对多的关系。

创建“讲师索引”视图的视图模型

“讲师”页显示来自三个不同表格的数据。创建一个视图模型，该模型中包含表示三个表格的三个实体。

在 SchoolViewModels 文件夹中，使用以下代码创建 InstructorIndexData.cs：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

为讲师模型创建基架

- Visual Studio
- .NET Core CLI

按照为“学生”模型搭建基架中的说明操作，并对模型类使用 `Instructor`。

上述命令为 `Instructor` 模型创建基架。运行应用并导航到“讲师”页。

将 Pages/Instructors/Index.cshtml.cs 替换为以下代码：

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData Instructor { get; set; }
        public int InstructorID { get; set; }

        public async Task OnGetAsync(int? id)
        {
            Instructor = new InstructorIndexData();
            Instructor.Instructors = await _context.Instructors
                  .Include(i => i.OfficeAssignment)
                  .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                  .AsNoTracking()
                  .OrderBy(i => i.LastName)
                  .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
            }
        }
    }
}
```

`OnGetAsync` 方法接受所选讲师 ID 的可选路由数据。

检查 Pages/Instructors/Index.cshtml.cs 文件中的查询：

```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

查询包括两项内容：

- `OfficeAssignment`：在讲师视图中显示。
- `CourseAssignments`：课程的教学内容。

更新“讲师索引”页

使用以下标记更新 Pages/Instructors/Index.cshtml：

```
@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructor.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @:  @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

上述标记进行以下更改：

- 将 `page` 指令从 `@page` 更新为 `@page "{id:int?}"`。`"{id:int?}"` 是一个路由模板。路由模板将 URL 中的整数查询字符串更改为路由数据。例如，单击仅具有 `@page` 指令的讲师的“选择”链接将生成如下 URL：

  `http://localhost:1234/Instructors?id=2`

  当页面指令是 `@page "{id:int?}"` 时，之前的 URL 为：

  `http://localhost:1234/Instructors/2`

- 页标题为“讲师”。

- 添加了仅在 `item.OfficeAssignment` 不为 null 时才显示 `item.OfficeAssignment.Location` 的“办公室”列。由于这是一对零或一的关系，因此可能没有相关的 OfficeAssignment 实体。

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

- 添加了显示每位讲师所授课程的“课程”列。有关此 Razor 语法的详细信息，请参阅使用 `@:` 进行显式行转换。

- 添加了向所选讲师的 `tr` 元素中动态添加 `class="success"` 的代码。此时会使用 Bootstrap 类为所选行设置背景色。

```
string selectedRow = "";
if (item.CourseID == Model.CourseID)
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- 添加了标记为“选择”的新的超链接。该链接将所选讲师的 ID 发送给 `Index` 方法并设置背景色。

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

运行应用并选择“讲师”选项卡。该页显示来自相关 `OfficeAssignment` 实体的 `Location`（办公室）。如果 OfficeAssignment` 为 NULL，则显示空白表格单元格。

单击“选择”链接。随即更改行样式。

添加由所选讲师教授的课程

将 Pages/Instructors/Index.cshtml.cs 中的 `OnGetAsync` 方法替换为以下代码：

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Where(
            i => i.ID == id.Value).Single();
        Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }
}
```

添加 `public int CourseID { get; set; }`

```
public class IndexModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public IndexModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    public InstructorIndexData Instructor { get; set; }
    public int InstructorID { get; set; }
    public int CourseID { get; set; }

    public async Task OnGetAsync(int? id, int? courseID)
    {
        Instructor = new InstructorIndexData();
        Instructor.Instructors = await _context.Instructors
              .Include(i => i.OfficeAssignment)
              .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Department)
              .AsNoTracking()
              .OrderBy(i => i.LastName)
              .ToListAsync();

        if (id != null)
        {
            InstructorID = id.Value;
            Instructor instructor = Instructor.Instructors.Where(
                i => i.ID == id.Value).Single();
            Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
        }

        if (courseID != null)
        {
            CourseID = courseID.Value;
            Instructor.Enrollments = Instructor.Courses.Where(
                x => x.CourseID == courseID).Single().Enrollments;
        }
    }
```

检查更新后的查询：

```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

先前查询添加了 `Department` 实体。

选择讲师时（`id != null`），将执行以下代码。从视图模型中的讲师列表检索所选讲师。向视图模型的 `Courses` 属性加载来自讲师 `CourseAssignments` 导航属性的 `Course` 实体。

```
if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = Instructor.Instructors.Where(
        i => i.ID == id.Value).Single();
    Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

`Where` 方法返回一个集合。在前面的 `Where` 方法中，仅返回单个 `Instructor` 实体。`Single` 方法将集合转换为单个 `Instructor` 实体。`Instructor` 实体提供对 `CourseAssignments` 属性的访问。`CourseAssignments` 提供对相关 `Course` 实体的访问。

当集合仅包含一个项时，集合使用 `Single` 方法。如果集合为空或包含多个项，`Single` 方法会引发异常。还可使用 `SingleOrDefault`，该方式在集合为空时返回默认值（本例中为 null）。在空集合上使用 `SingleOrDefault`：

- 引发异常（因为尝试在空引用上找到 `Courses` 属性）。
- 异常信息不太能清楚指出问题原因。

选中课程时，视图模型的 `Enrollments` 属性将填充以下代码：

```
if (courseID != null)
{
    CourseID = courseID.Value;
    Instructor.Enrollments = Instructor.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

在 Pages/Instructors/Index.cshtml Razor 页面末尾添加以下标记：

```
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@if (Model.Instructor.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Instructor.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    <a asp-page="./Index" asp-route-courseID="@item.CourseID">Select</a>
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

上述标记显示选中某讲师时与该讲师相关的课程列表。

测试应用。单击讲师页面上的“选择”链接。

显示学生数据

在本部分中，更新应用以显示所选课程的学生数据。

使用以下代码在 Pages/Instructors/Index.cshtml.cs 中更新 `OnGetAsync` 方法中的查询：

```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

更新 Pages/Instructors/Index.cshtml。在文件末尾添加以下标记：

```
@if (Model.Instructor.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Instructor.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

上述标记显示已注册所选课程的学生列表。

刷新页面并选择讲师。选择一门课程，查看已注册的学生及其成绩列表。

使用 Single 方法

`Single` 方法可在 `Where` 条件中进行传递，无需分别调用 `Where` 方法：

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();

    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Enrollments)
                        .ThenInclude(i => i.Student)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Single(
            i => i.ID == id.Value);
        Instructor.Courses = instructor.CourseAssignments.Select(
            s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Single(
            x => x.CourseID == courseID).Enrollments;
    }
}
```

使用 `Where` 时，前面的 `Single` 方法不适用。一些开发人员更喜欢 `Single` 方法样式。

显式加载

当前代码为 `Enrollments` 和 `Students` 指定预先加载：

```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

假设用户几乎不希望课程中显示注册情况。在此情况下，可仅在请求时加载注册数据进行优化。在本部分中，会更新 `OnGetAsync` 以使用 `Enrollments` 和 `Students` 的显式加载。

使用以下代码更新 `OnGetAsync`：

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            //.Include(i => i.CourseAssignments)
            //    .ThenInclude(i => i.Course)
            //        .ThenInclude(i => i.Enrollments)
            //            .ThenInclude(i => i.Student)
         // .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();


    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Where(
            i => i.ID == id.Value).Single();
        Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        var selectedCourse = Instructor.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        Instructor.Enrollments = selectedCourse.Enrollments;
    }
}
```

上述代码取消针对注册和学生数据的 ThenInclude 方法调用。如果已选中课程，则突出显示的代码会检索：

- 所选课程的 `Enrollment` 实体。
- 每个 `Enrollment` 的 `Student` 实体。

请注意，上述代码为 `.AsNoTracking()` 加上注释。对于跟踪的实体，仅可显式加载导航属性。

测试应用。对用户而言，该应用的行为与上一版本相同。

下一个教程将介绍如何更新相关数据。

上一页 下一页

ASP.NET Core 中的 Razor 页面和 EF Core - 更新相关数据 - 第 7 个教程(共 8 个)

2018/10/30 • • Edit Online

作者:Tom Dykstra 和 Rick Anderson

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程，请参阅第一个教程。

本教程演示如何更新相关数据。如果遇到无法解决的问题，请下载或查看已完成的应用。下载说明。

下图显示了部分已完成页面。

查看并测试"创建"和"编辑"课程页。创建新课程。系按其主键(一个整数)进行选择，而不是按名称。编辑新课程。完成测试后，请删除新课程。

创建基类以共享通用代码

"课程/创建"和"课程/编辑"页分别需要一个系名称列表。针对"创建"和"编辑"页创建 Pages/Courses/DepartmentNamePageModel.cshtml.cs 基类：

```
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                                   orderby d.Name // Sort by name.
                                   select d;

            DepartmentNameSL = new SelectList(departmentsQuery.AsNoTracking(),
                        "DepartmentID", "Name", selectedDepartment);
        }
    }
}
```

上面的代码创建 SelectList 以包含系名称列表。如果指定了 `selectedDepartment`，可在 `SelectList` 中选择该系。

“创建”和“编辑”页模型类将派生自 `DepartmentNamePageModel`。

自定义课程页

创建新的课程实体时，新实体必须与现有院系有关系。若要在创建课程的同时添加系，则“创建”和“编辑”的基类必须包含用于选择系的下拉列表。该下拉列表设置 `Course.DepartmentID` 外键 (FK) 属性。EF Core 使用 `Course.DepartmentID` FK 加载 `Department` 导航属性。

使用以下代码更新“创建”页模型：

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                 emptyCourse,
                 "course",   // Prefix for form value.
                 s => s.CourseID, s => s.DepartmentID, s => s.Title, s => s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, emptyCourse.DepartmentID);
            return Page();
        }
      }
}
```

前面的代码：

- 派生自 `DepartmentNamePageModel`。
- 使用 `TryUpdateModelAsync` 防止过多发布。
- 将 `ViewData["DepartmentID"]` 替换为 `DepartmentNameSL`（来自基类）。

`ViewData["DepartmentID"]` 将替换为强类型 `DepartmentNameSL`。建议使用强类型而非弱类型。有关详细信息，请参阅弱类型数据（ViewData 和 ViewBag）。

更新“课程创建”页

使用以下标记更新 Pages/Courses/Create.cshtml：

```
@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

上述标记进行以下更改：

- 将标题从“DepartmentID”更改为“Department”。
- 将 `"ViewBag.DepartmentID"` 替换为 `DepartmentNameSL`（来自基类）。
- 添加“选择系”选项。此更改将导致呈现“选择系”而不是第一个系。
- 在未选择系时添加验证消息。

Razor 页面使用选择标记帮助器：

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
            asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

测试“创建”页。“创建”页显示系名称，而不是系 ID。

更新“课程编辑”页。

使用以下代码更新“编辑”页模型：

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context,Course.DepartmentID);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (await TryUpdateModelAsync<Course>(
                 courseToUpdate,
                 "course",   // Prefix for form value.
                   c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, courseToUpdate.DepartmentID);
            return Page();
        }
    }
}
```

这些更改与在“创建”页模型中所做的更改相似。在上面的代码中，`PopulateDepartmentsDropDownList` 在系 ID 中传递并将选择下拉列表中指定的系。

使用以下标记更新 Pages/Courses/Edit.cshtml：

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

上述标记进行以下更改：

- 显示课程 ID。通常不显示实体的主键 (PK)。PK 对用户不具有任何意义。在这种情况下，PK 就是课程编号。
- 将标题从“DepartmentID”更改为“Department”。
- 将 `"ViewBag.DepartmentID"` 替换为 `DepartmentNameSL`（来自基类）。

该页面包含课程编号的隐藏域（`<input type="hidden">`）。添加具有 `asp-for="Course.CourseID"` 的 `<label>` 标记帮助器也同样需要隐藏域。用户单击“保存”时，需要 `<input type="hidden">`，以便在已发布的数据中包括课程编号。

测试更新的代码。创建、编辑和删除课程。

将 AsNoTracking 添加到“详细信息”和“删除”页模型

AsNoTracking 可以在不需要跟踪时提高性能。将 `AsNoTracking` 添加到“删除”和“详细信息”页模型。下面的代码显示更新的“删除”页模型：

```
public class DeleteModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public DeleteModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Course Course { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Course = await _context.Courses
            .AsNoTracking()
            .Include(c => c.Department)
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Course = await _context.Courses
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course != null)
        {
            _context.Courses.Remove(Course);
            await _context.SaveChangesAsync();
        }

        return RedirectToPage("./Index");
    }
}
```

在 Pages/Courses/Details.cshtml.cs 文件中更新 `OnGetAsync` 方法：

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Course = await _context.Courses
         .AsNoTracking()
         .Include(c => c.Department)
         .FirstOrDefaultAsync(m => m.CourseID == id);

    if (Course == null)
    {
        return NotFound();
    }
    return Page();
}
```

修改“删除”和“详细信息”页

使用以下标记更新“删除”Razor 页面：

```
@page
@model ContosoUniversity.Pages.Courses.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Department.DepartmentID)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

对“详细信息”页执行相同更改。

测试“课程”页

测试“创建”、“编辑”、“详细信息”和“删除”。

更新“讲师”页

以下各部分介绍更新“讲师”页。

添加办公室位置

编辑讲师记录时，可能希望更新讲师的办公室分配。`Instructor` 实体与 `OfficeAssignment` 实体是一对零或一关系。讲师代码必须处理：

- 如果用户清除办公室分配，则需删除 `OfficeAssignment` 实体。
- 如果用户输入办公室分配，且办公室分配原本为空，则需创建一个新 `OfficeAssignment` 实体。
- 如果用户更改办公室分配，则需更新 `OfficeAssignment` 实体。

使用以下代码更新讲师“编辑”页模型：

```
public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            await _context.SaveChangesAsync();
        }
        return RedirectToPage("./Index");

    }
}
```

前面的代码：

- 使用 `OfficeAssignment` 导航属性的预先加载从数据库获取当前的 `Instructor` 实体。
- 用模型绑定器中的值更新检索到的 `Instructor` 实体。`TryUpdateModel` 可防止过多发布。

- 如果办公室位置为空，则将 `Instructor.OfficeAssignment` 设置为 null。当 `Instructor.OfficeAssignment` 为 null 时，`OfficeAssignment` 表中的相关行将会删除。

更新讲师“编辑”页

使用办公室位置更新 Pages/Instructors/Edit.cshtml：

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

验证是否可以更改讲师办公室位置。

向“讲师编辑”页添加课程分配

讲师可能教授任意数量的课程。本部分将添加更改课程分配的功能。下图显示更新的讲师“编辑”页：

`Course` 和 `Instructor` 具有多对多关系。若要添加和删除关系，可以从 `CourseAssignments` 联接实体集中添加和删除实体。

通过复选框可对分配给讲师的课程进行更改。数据库中的每一门课程均有对应显示的复选框。已分配给讲师的课程将会被勾选。用户可以通过选择或清除复选框来更改课程分配。如果课程数量过多：

- 可以使用其他用户界面显示课程。
- 使用联接实体创建或删除关系的方法不会发生更改。

添加类以支持“创建”和“编辑”讲师页

使用以下代码创建 SchoolViewModels/AssignedCourseData.cs：

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

`AssignedCourseData` 类包含的数据可用于为讲师已分配的课程创建复选框。

创建 Pages/Instructors/InstructorCoursesPageModel.cshtml.cs 基类：

```
using ContosoUniversity.Data;
```

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {

        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                               Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.CourseAssignments.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }

        public void UpdateInstructorCourses(SchoolContext context,
            string[] selectedCourses, Instructor instructorToUpdate)
        {
            if (selectedCourses == null)
            {
                instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
                return;
            }

            var selectedCoursesHS = new HashSet<string>(selectedCourses);
            var instructorCourses = new HashSet<int>
                (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
            foreach (var course in context.Courses)
            {
                if (selectedCoursesHS.Contains(course.CourseID.ToString()))
                {
                    if (!instructorCourses.Contains(course.CourseID))
                    {
                        instructorToUpdate.CourseAssignments.Add(
                            new CourseAssignment
                            {
                                InstructorID = instructorToUpdate.ID,
                                CourseID = course.CourseID
                            });
                    }
                }
                else
                {
                    if (instructorCourses.Contains(course.CourseID))
                    {
                        CourseAssignment courseToRemove
                            = instructorToUpdate
                                .CourseAssignments
                                .SingleOrDefault(i => i.CourseID == course.CourseID);
                        context.Remove(courseToRemove);
                    }
                }
```

```
            }
        }
    }
}
```

`InstructorCoursesPageModel` 是将用于“编辑”和“创建”页模型的基类。`PopulateAssignedCourseData` 读取所有 `Course` 实体以填充 `AssignedCourseDataList`。该代码将设置每门课程的 `CourseID` 和标题，并决定是否为讲师分配该课程。HashSet 用于创建高效查找。

讲师“编辑”页模型

使用以下代码更新讲师“编辑”页模型：

```
public class EditModel : InstructorCoursesPageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        PopulateAssignedCourseData(_context, Instructor);
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id, string[] selectedCourses)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
        UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
        PopulateAssignedCourseData(_context, instructorToUpdate);
        return Page();
    }
}
```

上面的代码处理办公室分配更改。

更新“讲师”Razor 视图：

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                               @course.CourseID @:  @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
```

```
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

> **NOTE**
>
> 将代码粘贴到 Visual Studio 中时，换行符会发生更改并导致代码中断。按 Ctrl+Z 一次可撤消自动格式设置。按 Ctrl+Z 可以修复换行符，使其看起来如此处所示。缩进不一定要完美，但 `@</tr><tr>`、`@:<td>`、`@:</td>` 和 `@:</tr>` 行必须各成一行（如下所示）。选中新的代码块后，按 Tab 三次，使新代码与现有代码对齐。通过此链接投票或查看此 bug 的状态。

上面的代码将创建一个具有三列的 HTML 表。每列均具有一个复选框和包含课程编号及标题的标题。所有复选框均具有相同的名称（"selectedCourses"）。使用相同名称可指示模型绑定器将它们视为一个组。每个复选框的值特性都设置为 `CourseID`。发布页面时，模型绑定器会传递一个数组，该数组只包括所选复选框的 `CourseID` 值。

初次呈现复选框时，分配给讲师的课程具有已勾选的特性。

运行应用并测试更新的讲师"编辑"页。更改某些课程分配。这些更改将反映在"索引"页上。

注意：此处所使用的编辑讲师课程数据的方法适用于数量有限的课程。对于规模远大于此的集合，则使用不同的 UI 和不同的更新方法会更实用和更高效。

更新讲师"创建"页

使用以下代码更新讲师"创建"页模型：

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.CourseAssignments = new List<CourseAssignment>();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
            PopulateAssignedCourseData(_context, instructor);
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
        {
```

```
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var newInstructor = new Instructor();
            if (selectedCourses != null)
            {
                newInstructor.CourseAssignments = new List<CourseAssignment>();
                foreach (var course in selectedCourses)
                {
                    var courseToAdd = new CourseAssignment
                    {
                        CourseID = int.Parse(course)
                    };
                    newInstructor.CourseAssignments.Add(courseToAdd);
                }
            }

            if (await TryUpdateModelAsync<Instructor>(
                newInstructor,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                _context.Instructors.Add(newInstructor);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            PopulateAssignedCourseData(_context, newInstructor);
            return Page();
        }
    }
}
```

前面的代码与 Pages/Instructors/Edit.cshtml.cs 代码类似。

使用以下标记更新讲师“创建”Razor 页面：

```
@page
@model ContosoUniversity.Pages.Instructors.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
```

```
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>

            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                               @course.CourseID @:  @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

测试讲师“创建”页。

更新“删除”页

使用以下代码更新“删除”页模型：

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor = await _context.Instructors.SingleAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            Instructor instructor = await _context.Instructors
                .Include(i => i.CourseAssignments)
                .SingleAsync(i => i.ID == id);

            var departments = await _context.Departments
                .Where(d => d.InstructorID == id)
                .ToListAsync();
            departments.ForEach(d => d.InstructorID = null);

            _context.Instructors.Remove(instructor);

            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
    }
}
```

上面的代码执行以下更改：

- 对 `CourseAssignments` 导航属性使用预先加载。必须包含 `CourseAssignments`，否则删除讲师时将不会删除课程。为避免阅读它们，可以在数据库中配置级联删除。
- 如果要删除的讲师被指派为任何系的管理员，则需从这些系中删除该讲师分配。

ASP.NET Core 中的 Razor 页面和 EF Core - 并发 - 第 8 个教程(共 8 个)

2019/1/10 • • Edit Online

作者:Rick Anderson、Tom Dykstra 和 Jon P Smith

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程,请参阅第一个教程。

本教程介绍如何处理多个用户并发更新同一实体(同时)时出现的冲突。如果遇到无法解决的问题,请下载或查看已完成的应用。下载说明。

并发冲突

在以下情况下,会发生并发冲突:

- 用户导航到实体的编辑页面。
- 第一个用户的更改还未写入数据库之前,另一用户更新同一实体。

如果未启用并发检测,当发生并发更新时:

- 最后一个更新优先。也就是最后一个更新的值保存至数据库。
- 第一个并发更新将会丢失。

开放式并发

乐观并发允许发生并发冲突,并在并发冲突发生时作出正确反应。例如,Jane 访问院系编辑页面,将英语系的预算从 350,000.00 美元更改为 0.00 美元。

在 Jane 单击"保存"之前，John 访问了相同页面，并将开始日期字段从 2007/1/9 更改为 2013/1/9。

Jane 先单击“保存”，并在浏览器显示索引页时看到她的更改。

John 单击“编辑”页面上的“保存”，但页面的预算仍显示为 350,000.00 美元。接下来的情况取决于并发冲突的处理方式。

乐观并发包括以下选项：

- 可以跟踪用户已修改的属性，并仅更新数据库中相应的列。

  在这种情况下，数据不会丢失。两个用户更新了不同的属性。下次有人浏览英语系时，将看到 Jane 和 John 两个人的更改。这种更新方法可以减少导致数据丢失的冲突数。这种方法：

  - 无法避免数据丢失，如果对同一属性进行竞争性更改的话。
  - 通常不适用于 Web 应用。它需要维持重要状态，以便跟踪所有提取值和新值。维持大量状态可能影响应用性能。
  - 可能会增加应用复杂性（与实体上的并发检测相比）。
- 可让 John 的更改覆盖 Jane 的更改。

  下次有人浏览英语系时，将看到 2013/9/1 和提取的值 350,000.00 美元。这种方法称为“客户端优先”或“最后一个优先”方案。（客户端的所有值优先于数据存储的值。）如果不对并发处理进行任何编码，则自动执行“客户端优先”。

- 可以阻止在数据库中更新 John 的更改。应用通常会：

  - 显示错误消息。
  - 显示数据的当前状态。
  - 允许用户重新应用更改。

  这称为“存储优先”方案。（数据存储值优先于客户端提交的值。）本教程实施“存储优先”方案。此方法可确保用户在未收到警报时不会覆盖任何更改。

处理并发

当属性配置为并发令牌时：

- EF Core 验证提取属性后是否未更改属性。调用 SaveChanges 或 SaveChangesAsync 时会执行此检查。
- 如果提取属性后更改了属性，将引发 DbUpdateConcurrencyException。

数据库和数据模型必须配置为支持引发 `DbUpdateConcurrencyException`。

检测属性的并发冲突

可使用 ConcurrencyCheck 特性在属性级别检测并发冲突。该特性可应用于模型上的多个属性。有关详细信息，请参阅数据注释 - ConcurrencyCheck。

本教程中不使用 `[ConcurrencyCheck]` 特性。

检测行的并发冲突

要检测并发冲突，请将 rowversion 跟踪列添加到模型。`rowversion`：

- 是 SQL Server 特定的。其他数据库可能无法提供类似功能。
- 用于确定从数据库提取实体后未更改实体。

数据库生成 `rowversion` 序号，该数字随着每次行的更新递增。在 `Update` 或 `Delete` 命令中，`Where` 子句包括 `rowversion` 的提取值。如果要更新的行已更改：

- `rowversion` 不匹配提取值。
- `Update` 或 `Delete` 命令不能找到行，因为 `Where` 子句包含提取的 `rowversion`。
- 引发一个 `DbUpdateConcurrencyException`。

在 EF Core 中，如果未通过 `Update` 或 `Delete` 命令更新行，则引发并发异常。

向 Department 实体添加跟踪属性

在 Models/Department.cs 中，添加名为 RowVersion 的跟踪属性：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

Timestamp 特性指定此列包含在 `Update` 和 `Delete` 命令的 `Where` 子句中。该特性称为 `Timestamp`，因为之前版本的 SQL Server 在 SQL `rowversion` 类型将其替换之前使用 SQL `timestamp` 数据类型。

Fluent API 还可指定跟踪属性：

```
modelBuilder.Entity<Department>()
  .Property<byte[]>("RowVersion")
  .IsRowVersion();
```

以下代码显示更新 Department 名称时由 EF Core 生成的部分 T-SQL：

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

前面突出显示的代码显示包含 `RowVersion` 的 `WHERE` 子句。如果数据库 `RowVersion` 不等于 `RowVersion` 参数（`@p2`），则不更新行。

以下突出显示的代码显示验证更新哪一行的 T-SQL：

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

@@ROWCOUNT 返回受上一语句影响的行数。在没有行更新的情况下，EF Core 引发 `DbUpdateConcurrencyException`。

在 Visual Studio 的输出窗口中可看见 EF Core 生成的 T-SQL。

更新数据库

添加 `RowVersion` 属性可更改数据库模型，这需要迁移。

生成项目。在命令窗口中输入以下命令：

```
dotnet ef migrations add RowVersion
dotnet ef database update
```

前面的命令：

- 添加 Migrations/{time stamp}_RowVersion.cs 迁移文件。
- 更新 Migrations/SchoolContextModelSnapshot.cs 文件。此次更新将以下突出显示的代码添加到 `BuildModel` 方法：

```
modelBuilder.Entity("ContosoUniversity.Models.Department", b =>
    {
        b.Property<int>("DepartmentID")
            .ValueGeneratedOnAdd();

        b.Property<decimal>("Budget")
            .HasColumnType("money");

        b.Property<int?>("InstructorID");

        b.Property<string>("Name")
            .HasMaxLength(50);

        b.Property<byte[]>("RowVersion")
            .IsConcurrencyToken()
            .ValueGeneratedOnAddOrUpdate();

        b.Property<DateTime>("StartDate");

        b.HasKey("DepartmentID");

        b.HasIndex("InstructorID");

        b.ToTable("Department");
    });
```

- 运行迁移以更新数据库。

构架院系模型

- Visual Studio
- .NET Core CLI

按照为"学生"模型搭建基架中的说明操作，并对模型类使用 `Department`。

上述命令为 `Department` 模型创建基架。在 Visual Studio 中打开项目。

生成项目。

更新院系索引页

基架引擎为索引页创建 `RowVersion` 列，但不应显示该字段。本教程中显示 `RowVersion` 的最后一个字节，以帮助理解并发。不能保证最后一个字节是唯一的。实际应用不会显示 `RowVersion` 或 `RowVersion` 的最后一个字节。

更新索引页：

- 用院系替换索引。
- 将包含 `RowVersion` 的标记替换为 `RowVersion` 的最后一个字节。
- 将 FirstMidName 替换为 FullName。

以下标记显示更新后的页面：

```
@page
@model ContosoUniversity.Pages.Departments.IndexModel

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].Name)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].Budget)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].StartDate)
                </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Administrator)
            </th>
            <th>
                RowVersion
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Department) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Budget)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.StartDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Administrator.FullName)
            </td>
            <td>
                @item.RowVersion[7]
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.DepartmentID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.DepartmentID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

更新编辑页模型

使用以下代码更新 pages\departments\edit.cshtml.cs：

```
using ContosoUniversity.Data;
```

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class EditModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        // Replace ViewData["InstructorID"]
        public SelectList InstructorNameSL { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)  // eager loading
                .AsNoTracking()                 // tracking not required
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                return NotFound();
            }

            // Use strongly typed data rather than ViewData.
            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FirstMidName");

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var departmentToUpdate = await _context.Departments
                .Include(i => i.Administrator)
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            // null means Department was deleted by another user.
            if (departmentToUpdate == null)
            {
                return await HandleDeletedDepartment();
            }

            // Update the RowVersion to the value when this entity was
            // fetched. If the entity has been updated after it was
            // fetched, RowVersion won't match the DB RowVersion and
            // a DbUpdateConcurrencyException is thrown.
            // A second postback will make them match, unless a new
            // concurrency issue happens.
            _context.Entry(departmentToUpdate)
                .Property("RowVersion").OriginalValue = Department.RowVersion;
```

```
            if (await TryUpdateModelAsync<Department>(
                departmentToUpdate,
                "Department",
                s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
            {
                try
                {
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
                catch (DbUpdateConcurrencyException ex)
                {
                    var exceptionEntry = ex.Entries.Single();
                    var clientValues = (Department)exceptionEntry.Entity;
                    var databaseEntry = exceptionEntry.GetDatabaseValues();
                    if (databaseEntry == null)
                    {
                        ModelState.AddModelError(string.Empty, "Unable to save. " +
                            "The department was deleted by another user.");
                        return Page();
                    }

                    var dbValues = (Department)databaseEntry.ToObject();
                    await setDbErrorMessage(dbValues, clientValues, _context);

                    // Save the current RowVersion so next postback
                    // matches unless an new concurrency issue happens.
                    Department.RowVersion = (byte[])dbValues.RowVersion;
                    // Must clear the model error for the next postback.
                    ModelState.Remove("Department.RowVersion");
                }
            }

            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FullName", departmentToUpdate.InstructorID);

            return Page();
        }

       private async Task<IActionResult> HandleDeletedDepartment()
        {
            Department deletedDepartment = new Department();
            // ModelState contains the posted data because of the deletion error and will overide the
Department instance values when displaying Page().
            ModelState.AddModelError(string.Empty,
                "Unable to save. The department was deleted by another user.");
            InstructorNameSL = new SelectList(_context.Instructors, "ID", "FullName",
Department.InstructorID);
            return Page();
        }

        private async Task setDbErrorMessage(Department dbValues,
                Department clientValues, SchoolContext context)
        {

            if (dbValues.Name != clientValues.Name)
            {
                ModelState.AddModelError("Department.Name",
                    $"Current value: {dbValues.Name}");
            }
            if (dbValues.Budget != clientValues.Budget)
            {
                ModelState.AddModelError("Department.Budget",
                    $"Current value: {dbValues.Budget:c}");
            }
            if (dbValues.StartDate != clientValues.StartDate)
            {
                ModelState.AddModelError("Department.StartDate",
                    $"Current value: {dbValues.StartDate:d}");
```

```
            }
            if (dbValues.InstructorID != clientValues.InstructorID)
            {
                Instructor dbInstructor = await _context.Instructors
                   .FindAsync(dbValues.InstructorID);
                ModelState.AddModelError("Department.InstructorID",
                    $"Current value: {dbInstructor?.FullName}");
            }

            ModelState.AddModelError(string.Empty,
                "The record you attempted to edit "
              + "was modified by another user after you. The "
              + "edit operation was canceled and the current values in the database "
              + "have been displayed. If you still want to edit this record, click "
              + "the Save button again.");
        }
    }
}
```

要检测并发问题，请使用来自所提取实体的 `rowVersion` 值更新 OriginalValue。EF Core 使用包含原始 `RowVersion` 值的 WHERE 子句生成 SQL UPDATE 命令。如果没有行受到 UPDATE 命令影响（没有行具有原始 `RowVersion` 值），将引发 `DbUpdateConcurrencyException` 异常。

```
public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var departmentToUpdate = await _context.Departments
        .Include(i => i.Administrator)
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    // null means Department was deleted by another user.
    if (departmentToUpdate == null)
    {
        return await HandleDeletedDepartment();
    }

    // Update the RowVersion to the value when this entity was
    // fetched. If the entity has been updated after it was
    // fetched, RowVersion won't match the DB RowVersion and
    // a DbUpdateConcurrencyException is thrown.
    // A second postback will make them match, unless a new
    // concurrency issue happens.
    _context.Entry(departmentToUpdate)
        .Property("RowVersion").OriginalValue = Department.RowVersion;
```

在前面的代码中，`Department.RowVersion` 为实体提取后的值。使用此方法调用 `FirstOrDefaultAsync` 时，`OriginalValue` 为数据库中的值。

以下代码获取客户端值（向此方法发布的值）和数据库值：

```
try
{
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
catch (DbUpdateConcurrencyException ex)
{
    var exceptionEntry = ex.Entries.Single();
    var clientValues = (Department)exceptionEntry.Entity;
    var databaseEntry = exceptionEntry.GetDatabaseValues();
    if (databaseEntry == null)
    {
        ModelState.AddModelError(string.Empty, "Unable to save. " +
            "The department was deleted by another user.");
        return Page();
    }

    var dbValues = (Department)databaseEntry.ToObject();
    await setDbErrorMessage(dbValues, clientValues, _context);

    // Save the current RowVersion so next postback
    // matches unless an new concurrency issue happens.
    Department.RowVersion = (byte[])dbValues.RowVersion;
    // Must clear the model error for the next postback.
    ModelState.Remove("Department.RowVersion");
}
```

以下代码为每列添加自定义错误消息，这些列中的数据库值与发布到 `OnPostAsync` 的值不同：

```
private async Task setDbErrorMessage(Department dbValues,
        Department clientValues, SchoolContext context)
{

    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
           .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit "
      + "was modified by another user after you. The "
      + "edit operation was canceled and the current values in the database "
      + "have been displayed. If you still want to edit this record, click "
      + "the Save button again.");
}
```

以下突出显示的代码将 `RowVersion` 值设置为从数据库检索的新值。用户下次单击“保存”时，将仅捕获最后一次显示编辑页后发生的并发错误。

```
try
{
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
catch (DbUpdateConcurrencyException ex)
{
    var exceptionEntry = ex.Entries.Single();
    var clientValues = (Department)exceptionEntry.Entity;
    var databaseEntry = exceptionEntry.GetDatabaseValues();
    if (databaseEntry == null)
    {
        ModelState.AddModelError(string.Empty, "Unable to save. " +
            "The department was deleted by another user.");
        return Page();
    }

    var dbValues = (Department)databaseEntry.ToObject();
    await setDbErrorMessage(dbValues, clientValues, _context);

    // Save the current RowVersion so next postback
    // matches unless an new concurrency issue happens.
    Department.RowVersion = (byte[])dbValues.RowVersion;
    // Must clear the model error for the next postback.
    ModelState.Remove("Department.RowVersion");
}
```

`ModelState` 具有旧的 `RowVersion` 值，因此需使用 `ModelState.Remove` 语句。在 Razor 页面中，当两者都存在时，字段的 `ModelState` 值优于模型属性值。

更新“编辑”页

使用以下标记更新 Pages/Departments/Edit.cshtml：

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Department.DepartmentID" />
            <input type="hidden" asp-for="Department.RowVersion" />
            <div class="form-group">
                <label>RowVersion</label>
                @Model.Department.RowVersion[7]
            </div>
            <div class="form-group">
                <label asp-for="Department.Name" class="control-label"></label>
                <input asp-for="Department.Name" class="form-control" />
                <span asp-validation-for="Department.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.Budget" class="control-label"></label>
                <input asp-for="Department.Budget" class="form-control" />
                <span asp-validation-for="Department.Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.StartDate" class="control-label"></label>
                <input asp-for="Department.StartDate" class="form-control" />
                <span asp-validation-for="Department.StartDate" class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <label class="control-label">Instructor</label>
                <select asp-for="Department.InstructorID" class="form-control"
                        asp-items="@Model.InstructorNameSL"></select>
                <span asp-validation-for="Department.InstructorID" class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="./Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

前面的标记：

- 将 `page` 指令从 `@page` 更新为 `@page "{id:int}"`。
- 添加隐藏的行版本。必须添加 `RowVersion`，以便回发绑定值。
- 显示 `RowVersion` 的最后一个字节以进行调试。
- 将 `ViewData` 替换为强类型 `InstructorNameSL`。

使用编辑页测试并发冲突

在英语系打开编辑的两个浏览器实例：

- 运行应用，然后选择“院系”。
- 右键单击英语系的“编辑”超链接，然后选择“在新选项卡中打开”。
- 在第一个选项卡中，单击英语系的“编辑”超链接。

两个浏览器选项卡显示相同信息。

在第一个浏览器选项卡中更改名称，然后单击“保存”。

浏览器显示更改值并更新 rowVersion 标记后的索引页。请注意更新后的 rowVersion 标记，它在其他选项卡的第二回发中显示。

在第二个浏览器选项卡中更改不同字段。

单击“保存”。可看见所有不匹配数据库值的字段的错误消息：

此浏览器窗口将不会更改名称字段。将当前值(语言)复制并粘贴到名称字段。退出选项卡。客户端验证将删除错误消息。

再次单击"保存"。保存在第二个浏览器选项卡中输入的值。在索引页中可以看到保存的值。

更新"删除"页

使用以下代码更新"删除"页模型：

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
```

```
        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        public string ConcurrencyErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int id, bool? concurrencyError)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                 return NotFound();
            }

            if (concurrencyError.GetValueOrDefault())
            {
                ConcurrencyErrorMessage = "The record you attempted to delete "
                  + "was modified by another user after you selected delete. "
                  + "The delete operation was canceled and the current values in the "
                  + "database have been displayed. If you still want to delete this "
                  + "record, click the Delete button again.";
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            try
            {
                if (await _context.Departments.AnyAsync(
                    m => m.DepartmentID == id))
                {
                    // Department.rowVersion value is from when the entity
                    // was fetched. If it doesn't match the DB, a
                    // DbUpdateConcurrencyException exception is thrown.
                    _context.Departments.Remove(Department);
                    await _context.SaveChangesAsync();
                }
                return RedirectToPage("./Index");
            }
            catch (DbUpdateConcurrencyException)
            {
                return RedirectToPage("./Delete",
                    new { concurrencyError = true, id = id });
            }
        }
    }
}
```

删除页检测提取实体并更改时的并发冲突。提取实体后，`Department.RowVersion` 为行版本。EF Core 创建 SQL DELETE 命令时，它包括具有 `RowVersion` 的 WHERE 子句。如果 SQL DELETE 命令导致零行受影响：

- SQL DELETE 命令中的 `RowVersion` 与数据库中的 `RowVersion` 不匹配。
- 引发 DbUpdateConcurrencyException 异常。
- 使用 `concurrencyError` 调用 `OnGetAsync`。

更新“删除”页

使用以下代码更新 Pages/Departments/Delete.cshtml：

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Department.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.RowVersion)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.RowVersion[7])
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Administrator.FullName)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Department.DepartmentID" />
        <input type="hidden" asp-for="Department.RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-page="./Index">Back to List</a>
        </div>
</form>
</div>
```

上述标记进行以下更改：

- 将 `page` 指令从 `@page` 更新为 `@page "{id:int}"`。
- 添加错误消息。
- 将“管理员”字段中的 FirstMidName 替换为 FullName。
- 更改 `RowVersion` 以显示最后一个字节。

- 添加隐藏的行版本。必须添加 `RowVersion`，以便回发绑定值。

使用删除页测试并发冲突

创建测试系。

在测试系打开删除的两个浏览器实例：

- 运行应用，然后选择“院系”。
- 右键单击测试系的“删除”超链接，然后选择“在新选项卡中打开”。
- 单击测试系的“编辑”超链接。

两个浏览器选项卡显示相同信息。

在第一个浏览器选项卡中更改预算，然后单击“保存”。

浏览器显示更改值并更新 rowVersion 标记后的索引页。请注意更新后的 rowVersion 标记，它在其他选项卡的第二回发中显示。

从第二个选项卡中删除测试部门。并发错误显示来自数据库的当前值。单击“删除”将删除实体，除非 `RowVersion` 已更新，院系已删除。

请参阅继承了解如何继承数据模型。

其他资源

- EF Core 中的并发令牌
- EF Core 中的并发处理

上一篇

ASP.NET Core MVC 和 EF Core - 教程系列

2018/6/21 • • Edit Online

本教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor 页面是 ASP.NET Core 2.0 中的一个新选择，它是基于页面的编程模型，可以实现更简单、更高效地生成 Web UI。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果通过 Razor 页面版本选择本教程，请在此 GitHub 问题中说明原因。

1. 入门
2. 创建、读取、更新和删除操作
3. 排序、筛选、分页和分组
4. 迁移
5. 创建复杂数据模型
6. 读取相关数据
7. 更新相关数据
8. 处理并发冲突
9. 继承
10. 高级主题

ASP.NET Core MVC 和 Entity Framework Core - 第 1 个教程，共 10 个教程

2019/1/10 • • Edit Online

本教程没有升级至 ASP.NET Core 2.1。若要使用本教程的 ASP.NET Core 2.0 版，可以在目录表上方或者页面顶部选择“ASP.NET Core 2.0”：

相对于 2.0 版，本教程的 ASP.NET Core 2.1 Razor Pages 版有很多改进。

2.0 版教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。例如显著简化了基架代码。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果选择本教程的 Razor Pages 版，请在此 GitHub 讨论中说明原因。

作者：Tom Dykstra 和 Rick Anderson

本教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor 页面是 ASP.NET Core 2.0 中的一个新选择，它是基于页面的编程模型，可以实现更简单、更高效地生成 Web UI。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果通过 Razor 页面版本选择本教程，请在此 GitHub 问题中说明原因。

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework (EF) Core 2.0 和 Visual Studio 2017 创建 ASP.NET Core 2.0 MVC web 应用程序。

示例应用程序供一个虚构的 Contoso 大学网站使用。它包括诸如学生入学、课程创建和导师分配等功能。这是一系列教程中的第一个，这一系列教程主要展示了如何从零开始构建 Contoso 大学示例应用程序。

下载或查看已完成的应用程序。

EF Core 2.0 是 EF 的最新版本，但还没有包括 EF 6.x 的所有功能 。有关如何在 EF 6.x 和 EF Core 之间选择，请参阅 EF Core vs.EF6.x。如果你选择使用 EF 6.x，请参阅 本系列教程的上一个版本。

> **NOTE**
>
> 本教程的 ASP.NET Core 1.1 版本，请参阅 本教程中 VS 2017 Update 2 版本的 PDF 文档。

系统必备

安装以下工具之一：

- CLI 工具：Windows、Linux 或 macOS：.NET Core SDK 2.0 或更高版本
- IDE/编辑器工具
  - Windows：Visual Studio for Windows
    - ASP.NET 和 Web 开发工作负荷
    - .NET Core 跨平台开发工作负荷
  - Linux：Visual Studio Code
  - macOS：Visual Studio for Mac

疑难解答

如果遇到无法解决的问题，可以通过与 已完成的项目对比代码来查找解决方案。常见错误以及对应的解决方案，请参阅 最新教程中的故障排除。如果没有找到遇到的问题的解决方案，可以将问题发布到 StackOverflow.com 的 ASP.NET Core 或 EF Core 版块。

> **TIP**
>
> 这是一系列一共有十个教程，其中每个都是在前面教程已完成的基础上继续。请考虑在完成每一个教程后保存项目的副本。之后如果遇到问题，你可以从保存的副本中开始寻找问题，而不是从头开始。

Contoso University Web 应用程序

你将在这些教程中学习构建一个简单的大学网站的应用程序。

用户可以查看和更新学生、课程和教师信息。以下是一些你即将创建的页面。

本教程主要关注于如何使用 Entity Framework，所以此站点的UI样式都是直接套用内置的模板。

创建 ASP.NET Core MVC web 应用程序

打开 Visual Studio 并创建一个新 ASP.NET Core C# web 项目名为"ContosoUniversity"。

- 从"文件"菜单中选择"新建">"项目"。
- 从左窗格中依次选择"已安装">"Visual C#">"Web"。
- 选择"ASP.NET Core Web 应用程序"项目模板。
- 输入"ContosoUniversity"作为名称，然后单击"确定"。

- 等待 **新 ASP.NET Core Web 应用程序 (.NET Core)** 显示对话框

- 选择 **ASP.NET Core 2.0** 和 **Web 应用程序（模型-视图-控制器）**模板。

  注意：本教程需要安装 ASP.NET Core 2.0 和 EF Core 2.0 或更高版本 - 请确保未选中 ASP.NET Core 1.1。

- 请确保 **身份**验证 设置为 **不进行身份**验证。

- 单击"确定"

设置网站样式

通过几个简单的更改设置站点菜单、布局和主页。

打开 Views/Shared/_Layout.cshtml 并进行以下更改：

- 将文件中的"ContosoUniversity"更改为"Contoso University"。需要更改三个地方。

- 添加菜单项 **Students**，**Courses**，**Instructors**，和 **Department**，并删除 **Contact**菜单项。

突出显示所作更改。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
```

```
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">Contoso
University</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                    <li><a asp-area="" asp-controller="Students" asp-action="Index">Students</a></li>
                    <li><a asp-area="" asp-controller="Courses" asp-action="Index">Courses</a></li>
                    <li><a asp-area="" asp-controller="Instructors" asp-action="Index">Instructors</a></li>
                    <li><a asp-area="" asp-controller="Departments" asp-action="Index">Departments</a></li>
                </ul>
            </div>
        </div>
    </nav>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2017 - Contoso University</p>
        </footer>
    </div>

    <environment names="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment names="Staging,Production">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
        </script>
        <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

在 *Views/Home/Index.cshtml*，将文件的内容替换为以下代码以将有关 ASP.NET 和 MVC 的内容替换为有关此应用程序的内容：

```
@{
    ViewData["Title"] = "Home Page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p><a class="btn btn-default" href="https://docs.asp.net/en/latest/data/ef-mvc/intro.html">See the
tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default" href="https://github.com/aspnet/Docs/tree/master/aspnetcore/data/ef-
mvc/intro/samples/cu-final">See project source code &raquo;</a></p>
    </div>
</div>
```

按 CTRL + F5 来运行该项目或从菜单选择 **调试 > 开始执行(不调试)**。你会看到首页，以及通过这个教程创建的页对应的选项卡。

Entity Framework Core NuGet 包

若要为项目添加 EF Core 支持，需要安装相应的数据库驱动包。本教程使用 SQL Server，相关驱动包Microsoft.EntityFrameworkCore.SqlServer。此包包含在 Microsoft.AspNetCore.App 元包中，因此，如果应用具有对 `Microsoft.AspNetCore.App` 包的包引用，则无需引用该包。

此包和其依赖项（`Microsoft.EntityFrameworkCore` 和 `Microsoft.EntityFrameworkCore.Relational`）一起提供 EF 的运行时支持。你将在之后的 迁移 教程中学习添加工具包。

有关其他可用于 EF Core 的数据库驱动的信息，请参阅 数据库驱动。

创建数据模型

接下来你将创建 Contoso 大学应用程序的实体类。你将从以下三个实体类开始。

`Student` 和 `Enrollment` 实体之间是一对多的关系，`Course` 和 `Enrollment` 实体之间也是一个对多的关系。换而言之，一名学生可以修读任意数量的课程，并且某一课程可以被任意数量的学生修读。

接下来，你将创建与这些实体对应的类。

Student 实体

在 *Models* 文件夹中，创建一个名为 *Student.cs* 的类文件并且将模板代码替换为以下代码。

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`ID` 属性将成为对应于此类的数据库表中的主键。默认情况下，EF 将会将名为 `ID` 或 `classnameID` 的属性解析为主键。

`Enrollments` 属性是导航属性。导航属性中包含与此实体相关的其他实体。在这个案例下，`Student entity` 中的 `Enrollments` 属性会保留所有与 `Student` 实体相关的 `Enrollment`。换而言之，如果在数据库中有两行描述同一个学生的修读情况（两行的 StudentID 值相同，而且 StudentID 作为外键和某位学生的主键值相同），`Student` 实体的 `Enrollments` 导航属性将包含那两个 `Enrollment` 实体。

如果导航属性可以具有多个实体（如多对多或一对多关系），那么导航属性的类型必须是可以添加、删除和更新条目的容器，如 `ICollection<T>`。你可以指定 `ICollection<T>` 或实现该接口类型，如 `List<T>` 或 `HashSet<T>`。如果指定 `ICollection<T>`，EF在默认情况下创建 `HashSet<T>` 集合。

Enrollment 实体

在 *Models* 文件夹中，创建 *Enrollment.cs* 并且用以下代码替换现有代码：

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

`EnrollmentID` 属性将被设为主键; 此实体使用 `classnameID` 模式而不是如 `Student` 实体那样直接使用 `ID`。通常情况下，你选择一个主键模式，并在你的数据模型自始至终使用这种模式。在这里，使用了两种不同的模式只是为了说明你可以使用任一模式来指定主键。在 后面的教程，你将了解到使用 `ID` 这种模式可以更轻松地在数据模型之间实现继承。

`Grade` 属性是 `enum`。`Grade` 声明类型后的 `?` 表示 `Grade` 属性可以为 null。评级为 null 和评级为零是有区别的 --null 意味着评级未知或者尚未分配。

`StudentID` 属性是一个外键，`Student` 是与其且对应的导航属性。`Enrollment` 实体与一个 `Student` 实体相关联，因此该属性只包含单个 `Student` 实体 (与前面所看到的 `Student.Enrollments` 导航属性不同后，`Student` 中可以容纳多个 `Enrollment` 实体)。

`CourseID` 属性是一个外键， `Course` 是与其对应的导航属性。`Enrollment` 实体与一个 `Course` 实体相关联。

如果一个属性名为 `<navigation property name><primary key property name>`，Entity Framework 就会将这个属性解析为外键属性(例如， `Student` 实体的主键是 `ID`， `Student` 是 `Enrollment` 的导航属性所以 `Enrollment` 实体中 `StudentID` 会被解析为外键)。此外还可以将需要解析为外键的属性命名为 `<primary key property name>` (例如， `CourseID` 由于 `Course` 实体的主键所以 `CourseID` 也被解析为外键)。

Course 实体

在 *Models* 文件夹中，创建 *Course.cs* 并且用以下代码替换现有代码：

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`Enrollments` 属性是导航属性。一个 `Course` 体可以与任意数量的 `Enrollment` 实体相关。

我们在本系列 后面的教程 中会有更多有关 `DatabaseGenerated` 特性的例子。简单来说，此特性让你能自行指定主键，而不是让数据库自动指定主键。

创建数据库上下文

使得给定的数据模型与 Entity Framework 功能相协调的主类是数据库上下文类。可以通过继承 `Microsoft.EntityFrameworkCore.DbContext` 类的方式创建此类。在该类中你可以指定数据模型中包含哪些实体。你还可以定义某些 Entity Framework 行为。在此项目中将数据库上下文类命名为 `SchoolContext`。

在项目文件夹中，创建名为的文件夹 *Data*。

在 *Data* 文件夹创建名为 *SchoolContext.cs* 的类文件，并将模板代码替换为以下代码：

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

此代码将为每个实体集创建 `DbSet` 属性。在 Entity Framework 中，实体集通常与数据表相对应，具体实体与表中的行相对应。

在这里可以省略 `DbSet<Enrollment>` 和 `DbSet<Course>` 语句，实现的功能没有任何改变。Entity Framework 会隐式包含这两个实体因为 `Student` 实体引用了 `Enrollment` 实体、`Enrollment` 实体引用了 `Course` 实体。

当数据库创建完成后，EF 创建一系列数据表，表名默认和 `DbSet` 属性名相同。集合属性的名称一般使用复数形式，但不同的开发人员的命名习惯可能不一样，开发人员根据自己的情况确定是否使用复数形式。在定义 DbSet 属性的代码之后，添加下面高亮代码，对 DbContext 指定单数的表名来覆盖默认的表名。此教程在最后一个 DbSet 属性之后，添加以下高亮显示的代码。

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

使用依赖注入注册上下文

ASP.NET Core 默认实现 依赖注入。在应用程序启动过程通过依赖注入注册相关服务（例如 EF 数据库上下文）。需要这些服务的组件（如 MVC 控制器）可以通过向构造函数添加相关参数来获得对应服务。在本教程后面你将看到控制器构造函数的代码，就是通过上述方式获得上下文实例。

若要将 `SchoolContext` 注册为一种服务，打开 *Startup.cs*，并将高亮代码添加到 `ConfigureServices` 方法中。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

通过调用 `DbContextOptionsBuilder` 中的一个方法将数据库连接字符串在配置文件中的名称传递给上下文对象。进行本地开发时，ASP.NET Core 配置系统 在 *appsettings.json* 文件中读取数据库连接字符串。

添加 `using` 语句引用 `ContosoUniversity.Data` 和 `Microsoft.EntityFrameworkCore` 命名空间，然后生成项目。

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
```

打开 appsettings.json 文件，并如以下示例所示添加连接字符串。

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

SQL Server Express LocalDB

数据库连接字符串指定使用 SQL Server LocalDB 数据库。LocalDB 是 SQL Server Express 数据库引擎的轻量级版本，用于应用程序开发，不在生产环境中使用。LocalDB 作为按需启动并在用户模式下运行的轻量级数据库没有复杂的配置。默认情况下，LocalDB 在 `C:/Users/<user>` 目录下创建 *.mdf* 数据库文件。

添加代码以使用测试数据初始化数据库

Entity Framework 已经为你创建了一个空数据库。在本部分中，你将编写一个方法用于向数据库填充测试数据，该方法会在数据库创建完成之后执行。

此处将使用 `EnsureCreated` 方法来自动创建数据库。在 后面的教程 你将了解如何通过使用 Code First Migration 来更改而不是删除并重新创建数据库来处理模型更改。

在 *Data* 文件夹中，创建名为的新类文件 *DbInitializer.cs* 并且将模板代码替换为以下代码，使得在需要时能创建数据库并向其填充测试数据。

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
            new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-
01")},
            new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-
01")},
            new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-
01")},
            new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-
01")},
            new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-
01")},
            new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-
01")},
            new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-
```

```
            new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-
01")}
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
            new Course{CourseID=1050,Title="Chemistry",Credits=3},
            new Course{CourseID=4022,Title="Microeconomics",Credits=3},
            new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
            new Course{CourseID=1045,Title="Calculus",Credits=4},
            new Course{CourseID=3141,Title="Trigonometry",Credits=4},
            new Course{CourseID=2021,Title="Composition",Credits=3},
            new Course{CourseID=2042,Title="Literature",Credits=4}
            };
            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
            new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
            new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
            new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
            new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
            new Enrollment{StudentID=3,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
            new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
            new Enrollment{StudentID=6,CourseID=1045},
            new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
            foreach (Enrollment e in enrollments)
            {
                context.Enrollments.Add(e);
            }
            context.SaveChanges();
        }
    }
}
```

这段代码首先检查是否有学生数据在数据库中，如果没有的话，就可以假定数据库是新建的，然后使用测试数据进行填充。代码中使用数组存放测试数据而不是使用 `List<T>` 集合是为了优化性能。

在 *Program.cs*，修改 `Main` 方法，使得在应用程序启动时能执行以下操作：

- 从依赖注入容器中获取数据库上下文实例。
- 调用 seed 方法，将上下文传递给它。
- Seed 方法完成此操作时释放上下文。

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the database.");
        }
    }

    host.Run();
}
```

添加 `using` 语句：

```
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Data;
```

在旧版教程中，你可能会在 *Startup.cs* 中的 `Configure` 方法看到类似的代码。我们建议你只在为了设置请求管道时使用 `Configure` 方法。将应用程序启动代码放入 `Main` 方法。

现在首次运行该应用程序，创建数据库并使用测试数据作为种子数据。每当你更改数据模型时，可以删除数据库、更新你的 Initialize 方法，然后使用上述方式更新新数据库。在之后的教程中，你将了解如何在数据模型更改时，只需修改数据库而无需删除重建数据库。

创建控制器和视图

接下来，将使用 Visual Studio 中的基架引擎添加一个 MVC 控制器，以及使用 EF 来查询和保存数据的视图。

CRUD 操作方法和视图的自动创建被称为基架。基架与代码生成不同，基架的代码是一个起点，可以修改基架以满足自己需求，而你通常无需修改生成的代码。当你需要自定义生成代码时，可使用一部分类或需求发生变化时重新生成代码。

- 右键单击 **解决方案资源管理器** 中的 **Controllers** 文件夹选择 **添加 > 新搭建基架的项目**。

如果出现"添加 MVC 依赖项"对话框：

- 将 Visual Studio 更新到最新版本。15.5 之前的 Visual Studio 版本显示此对话框。
- 如果无法更新，请选择"添加"，然后再次按照添加控制器步骤操作。
- 在"添加基架"对话框中：
  - 选择 视图**使用 Entity Framework 的 MVC 控制器**。
  - 单击 **添加**。
- 在"添加控制器"对话框中：
  - 在 **模型**类 选择 **Student**。
  - 在"数据上下文类"中选择 SchoolContext。

- 使用 **StudentsController** 作为默认名称。
- 单击 **添加**。

当你单击 **添加** 后，Visual Studio 基架引擎创建 *StudentsController.cs* 文件和一组对应于控制器的视图（*.cshtml* 文件）。

（如果你之前手动创建数据库上下文，基架引擎还可以自动创建。你可以在 **添加控制器** 对话框中单击右侧的加号框 **数据上下文类** 来指定在一个新上下文类。然后，Visual Studio 将创建你的 `DbContext`，控制器和视图类。）

注意控制器将 `SchoolContext` 作为构造函数参数。

```
namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
```

ASP.NET Core 依赖关系注入负责将 `SchoolContext` 实例传递到控制器。在前面的教程中已经通过修改 *Startup.cs* 文件来配置注入规则。

控制器包含 `Index` 操作方法，用于显示数据库中的所有学生。该方法从学生实体集中获取学生列表，学生实体集则是通过读取数据库上下文实例中的 `Students` 属性获得：

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

本教程后面部分将介绍此代码中的异步编程元素。

Views/Students/Index.cshtml 视图使用table标签显示此列表：

```
@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    @Html.DisplayNameFor(model => model.LastName)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.EnrollmentDate)
                </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

按 CTRL + F5 来运行该项目，或从菜单选择 调试 > 开始执行(不调试)。

单击学生选项卡以查看 `DbInitializer.Initialize` 插入的测试的数据。你将看到 `Student` 选项卡链接在页的顶部或在单击右上角后的导航图标中，具体显示在哪里取决于浏览器窗口宽度。

查看数据库

当你启动了应用程序，`DbInitializer.Initialize` 方法调用 `EnsureCreated`。EF 没有检测到相关数据库，因此自己创建了一个，接着 `Initialize` 方法的其余代码向数据库中填充数据。你可以使用 Visual Studio 中的 **SQL Server 对象资源管理器** (SSOX) 查看数据库。

关闭浏览器。

如果 SSOX 窗口尚未打开，请从Visual Studio 中的视图 菜单中选择。

在 SSOX 中，单击 **(localdb) \MSSQLLocalDB > 数据**库，然后单击和 *appsettings.json* 文件中的连接字符串对应的数据库。

展开"表"节点，查看数据库中的表。

右键单击 **Student** 表，然后单击 查看数据，即可查看已创建的列和已插入到表的行。

.mdf 和 *.ldf* 数据库文件位于 *C:\Users* 文件夹中。

因为调用 `EnsureCreated` 的初始化方法在启动应用程序时才运行，所以在这之前你可以更改 `Student` 类、删除数据库、再运行一次应用程序，这时候数据库将自动重新创建，以匹配所做的更改。例如，如果向 `Student` 类添加 `EmailAddress` 属性，重新的创建表中会有 `EmailAddress` 列。

约定

由于 Entity Framwork 有一定的约束条件，你只需要按规则编写很少的代码就能够创建一个完整的数据库。

- `DbSet` 类型的属性用作表名。如果实体未被 `DbSet` 属性引用，实体类名称用作表名称。
- 使用实体属性名作为列名。
- 以 ID 或 classnameID 命名的实体属性被视为主键属性。
- 如果属性名为 将被解释为外键属性 (例如，`StudentID` 对应 `Student` 导航属性，`Student` 实体的主键是 `ID`，所以 `StudentID` 被解释为外键属性)。此外也可以将外键属性命名为 (例如，`EnrollmentID`，由于 `Enrollment` 实体的主键是 `EnrollmentID`，因此被解释为外键)。

约定行为可以重写。例如，本教程前半部分显式指定表名称。本系列 后面教程 则设置列名称并将任何属性设置为主键或外键。

异步代码

异步编程是 ASP.NET Core 和 EF Core 的默认模式。

Web 服务器的可用线程是有限的，而在高负载情况下的可能所有线程都被占用。当发生这种情况的时候，服务器就无法处理新请求，直到线程被释放。使用同步代码时，可能会出现多个线程被占用但不能执行任何操作的情况，因为它们正在等待 I/O 完成。使用异步代码时，当进程正在等待 I/O 完成，服务器可以将其线程释放用于处理其他请求。因此，异步代码使得服务器更有效地使用资源，并且该服务器可以无延迟地处理更多流量。

异步代码在运行时，会引入的少量开销，在低流量时对性能的影响可以忽略不计，但在针对高流量情况下潜在的性能提升是可观的。

在以下代码中，`async` 关键字、`Task<T>` 返回值、`await` 关键字和 `ToListAsync` 方法让代码异步执行。

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

- `async` 关键字用于告知编译器该方法主体将生成回调并自动创建 `Task<IActionResult>` 返回对象。
- 返回类型 `Task<IActionResult>` 表示正在进行的工作返回的结果为 `IActionResult` 类型。
- `await` 关键字会使得编译器将方法拆分为两个部分。第一部分是以异步方式结束已启动的操作。第二部分是当操作完成时注入调用回调方法的地方。
- `ToListAsync` 是 `ToList` 方法的的异步扩展版本。

使用 Entity Framework 编写异步代码时的一些注意事项：

- 只有导致查询或发送数据库命令的语句才能以异步方式执行。包括 `ToListAsync`，`SingleOrDefaultAsync`，和 `SaveChangesAsync`。不包括只需更改 `IQueryable` 的语句，如 `var students = context.Students.Where(s => s.LastName == "Davolio")`。
- EF 上下文是线程不安全的：请勿尝试并行执行多个操作。当调用异步 EF 方法时，始终使用 `await` 关键字。
- 如果你想要利用异步代码的性能优势，请确保你所使用的任何库和包在它们调用导致 Entity Framework 数据库查询方法时也使用异步。

有关在 .NET 异步编程的详细信息，请参阅 异步概述。

总结

你现在已创建了一个使用 Entity Framework Core 和 SQL Server Express LocalDB 来存储和显示数据的简单应用程序。在下一个教程中，你将学习如何执行基本的 CRUD（创建、读取、更新、删除）操作。

下一页

ASP.NET Core MVC 和 EF Core 教程 - 创建、读取、更新和删除 (2/10)

2018/11/5 • • Edit Online

本教程没有升级至 ASP.NET Core 2.1。若要使用本教程的 ASP.NET Core 2.0 版，可以在目录表上方或者页面顶部选择“ASP.NET Core 2.0”：

相对于 2.0 版，本教程的 ASP.NET Core 2.1 Razor Pages 版有很多改进。

2.0 版教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。例如显著简化了基架代码。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果选择本教程的 Razor Pages 版，请在此 GitHub 讨论中说明原因。

作者：Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列，请参阅本系列中的第一个教程。

在上一个教程中，创建了一个使用 Entity Framework 和 SQL Server LocalDB 来存储和显示数据的 MVC 应用程序。在本教程中，将评审和自定义 MVC 基架在控制器和视图中自动创建的 CRUD（创建、读取、更新、删除）代码。

> **NOTE**
> 为了在控制器和数据访问层之间创建一个抽象层，常见的做法是实现存储库模式。为了保持这些教程内容简单并重点介绍如何使用 Entity Framework 本身，它们不使用存储库。有关存储库和 EF 的信息，请参阅本系列中的最后一个教程。

在本教程中，将使用以下网页：

Details - Contoso Unive

localhost:5813/Students/Details/1

Contoso University Home About Students Courses Instructors Departments

Register Log in

Student

LastName Alexander
FirstMidName Carson
EnrollmentDate 9/1/2005 12:00:00 AM
Enrollments

Course Title	Grade
Chemistry	A
Microeconomics	C
Macroeconomics	B

Create

s/Create

Contoso University

Create

Student

LastName

FirstMidName

EnrollmentDate

Create

自定义“详细信息”页

学生索引页的基架代码省略了 `Enrollments` 属性，因为该属性包含一个集合。在“详细信息”页上，将以 HTML 表形式显示集合的内容。

在 Controllers/StudentsController.cs 中，“详细信息”视图的操作方法使用 `SingleOrDefaultAsync` 方法检索单个 `Student` 实体。添加调用 `Include` 的代码。`ThenInclude` 和 `AsNoTracking` 方法，如以下突出显示的代码所示。

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .Include(s => s.Enrollments)
            .ThenInclude(e => e.Course)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    return View(student);
}
```

`Include` 和 `ThenInclude` 方法使上下文加载 `Student.Enrollments` 导航属性，并在每个注册中加载 `Enrollment.Course` 导航属性。有关这些方法的详细信息，请参阅读取相关数据教程。

对于返回的实体未在当前上下文生存期中更新的情况，`AsNoTracking` 方法将会提升性能。本教程末尾将介绍有关 `AsNoTracking` 的详细信息。

路由数据

传递到 `Details` 方法的键值来自路由数据。路由数据是模型绑定器在 URL 的段中找到的数据。例如，默认路由指定控制器、操作和 ID 段：

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

在下面的 URL 中，默认路由将 Instructor 映射作为控制器、Index 作为操作，1 作为 ID；这些都是路由数据值。

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```

URL 的最后部分 ("?courseID=2021") 是一个查询字符串。如果将 `id` 作为查询字符串值传递，模型绑定器也会将 ID 值作为参数传递给 `Details` 方法：

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

在索引页中，超链接 URL 由 Razor 视图中的标记帮助器语句创建。在以下 Razor 代码中，`id` 参数与默认路由相匹配，因此 `id` 将添加到路由数据。

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

`item.ID` 为 6 时，会生成以下 HTML：

```
<a href="/Students/Edit/6">Edit</a>
```

在以下 Razor 代码中，`studentID` 与默认路由中的参数不匹配，所以将它作为查询字符串添加。

```
<a asp-action="Edit" asp-route-studentID="@item.ID">Edit</a>
```

`item.ID` 为 6 时，会生成以下 HTML：

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

有关标记帮助程序的详细信息，请参阅 ASP.NET Core 中的标记帮助程序。

将注册添加到“详细信息”视图

打开 *Views/Students/Details.cshtml*。每个字段都使用 `DisplayNameFor` 和 `DisplayFor` 帮助器来显示，如下面的示例中所示：

```
<dt>
    @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd>
    @Html.DisplayFor(model => model.LastName)
</dd>
```

在最后一个字段之后和 `</dl>` 闭合标记之前，添加以下代码以显示注册列表：

```
<dt>
    @Html.DisplayNameFor(model => model.Enrollments)
</dt>
<dd>
    <table class="table">
        <tr>
            <th>Course Title</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Course.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
</dd>
```

如果代码缩进在粘贴代码后出现错误，请按 CTRL+K+D 进行更正。

此代码循环通过 `Enrollments` 导航属性中的实体。它将针对每个注册显示课程标题和成绩。课程标题从 Course 实体中检索，该实体存储在 Enrollments 实体的 `Course` 导航属性中。

运行应用，选择“学生”选项卡，然后单击学生的“详细信息”链接。将看到所选学生的课程和年级列表：

更新“创建”页

在 *StudentsController.cs* 中修改 HttpPost `Create` 方法，在 `Bind` 特性中添加 try catch 块并删除 ID 值。

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

此代码将 ASP.NET Core MVC 模型绑定器创建的 Student 实体添加到 Students 实体集，然后将更改保存到数据库。（模型绑定器指的是 ASP.NET Core MVC 功能，用户可利用它来轻松处理使用表单提交的数据；模型绑定器将已发布的表单值转换为 CLR 类型，并将其传递给操作方法的参数。在本例中，模型绑定器将使用 Form 集合的属性值实例化 Student 实体。）

已从 `Bind` 特性删除 `ID`，因为 ID 是插入行时 SQL Server 将自动设置的主键值。来自用户的输入不会设置 ID 值。

除了 `Bind` 特性，try-catch 块是对基架代码所做的唯一更改。如果保存更改时捕获到来自 `DbUpdateException` 的异常，则会显示一般错误消息。有时 `DbUpdateException` 异常是由应用程序外部的某些内容而非编程错误引起的，因此建议用户再次尝试。尽管在本示例中未实现，但生产质量应用程序会记录异常。有关详细信息，请参阅监视和遥测（使用 Azure 构建真实世界云应用）中的"见解记录"部分。

`ValidateAntiForgeryToken` 特性帮助抵御跨网站请求伪造 (CSRF) 攻击。令牌通过 FormTagHelper 自动注入到视图中，并在用户提交表单时包含该令牌。令牌由 `ValidateAntiForgeryToken` 特性验证。有关 CSRF 的详细信息，请参阅反请求伪造。

有关过多发布的安全说明

基架代码包含在 `Create` 方法中的 `Bind` 特性是防止在创建方案中过多发布的一种方法。例如，假设 Student 实体包含不希望此网页设置的 `Secret` 属性。

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

即使网页上没有 `Secret` 字段，黑客也可以使用 Fiddler 之类的工具，或者编写一些 JavaScript 来发布 `Secret` 表单值。创建 Student 实例时，如果不利用 `Bind` 特性来限制模型绑定器使用的字段，模型绑定器会选取该 `Secret` 表单值并使用它来创建 Student 实体实例。然后将在数据库中更新黑客为 `Secret` 表单字段指定的任意值。下图显示 Fiddler 工具正在将 `Secret` 字段（值为"OverPost"）添加到已发布的表单值。

然后值"OverPost"将成功添加到插入行的 `Secret` 属性，尽管你从未打算网页可设置该属性。

可以防止在编辑方案中过多发布，方法是首先从数据库读取实体，然后调用 `TryUpdateModel` 并在显式允许的属性列表中传递。这些教程中使用的也是这种方法。

许多开发者首选的防止过多发布的另一种方法是使用视图模型，而不是包含模型绑定的实体类。仅包含想要在视图模型中更新的属性。完成 MVC 模型绑定器后，根据需要使用 AutoMapper 之类的工具将视图模型属性复制到实体实例。使用实体实例上的 `_context.Entry` 将其状态设置为 `Unchanged`，然后在视图模型中包含的每个实体属性上将 `Property("PropertyName").IsModified` 设置为 true。此方法同时适用于编辑和创建方案。

测试创建页

Views/Students/Create.cshtml 中的代码对每个字段使用 `label`、`input` 和 `span`（适用于验证消息）标记帮助器。

运行应用，选择“学生”选项卡，并单击“新建”。

输入姓名和日期。如果浏览器允许输入无效日期，请尝试输入。（某些浏览器强制要求使用日期选取器。）然后单击“创建”，查看错误消息。

这是默认获取的服务器端验证；在下一个教程中，还将介绍如何添加生成客户端验证代码的特性。以下突出显示的代码显示 `Create` 方法中的模型验证检查。

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

将日期更改为有效值，并单击“创建”，查看“索引”页中显示的新学生。

更新“编辑”页

在 StudentController.cs 中，HttpGet `Edit` 方法（不具有 `HttpPost` 特性）使用 `SingleOrDefaultAsync` 方法检索所选的 Student 实体，如 `Details` 方法中所示。不需要更改此方法。

建议的 HttpPost 编辑代码：读取和更新

使用以下代码替换 HttpPost Edit 操作方法。

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var studentToUpdate = await _context.Students.SingleOrDefaultAsync(s => s.ID == id);
    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(studentToUpdate);
}
```

这些更改实现安全最佳做法，防止过多发布。基架生成了 `Bind` 特性，并将模型绑定器创建的实体添加到具有 `Modified` 标记的实体集。不建议将该代码用于多个方案，因为 `Bind` 特性将清除未在 `Include` 参数中列出的字段中的任何以前存在的数据。

新代码读取现有实体并调用 `TryUpdateModel`，以基于已发布表单数据中的用户输入更新已检索实体中的字段。Entity Framework 的自动更改跟踪在由表单输入更改的字段上设置 `Modified` 标记。调用 `SaveChanges` 方法时，Entity Framework 会创建 SQL 语句，以更新数据库行。忽略并发冲突，并且仅在数据库中更新由用户更新的表列。（下一个教程将介绍如何处理并发冲突。）

作为防止过多发布的最佳做法，请将希望通过"编辑"页更新的字段列入 `TryUpdateModel` 参数。（参数列表中字段列表之前的空字符串用于与表单字段名称一起使用的前缀。）目前没有要保护的额外字段，但是列出希望模型绑定器绑定的字段可确保以后将字段添加到数据模型时，它们将自动受到保护，直到明确将其添加到此处为止。

这些更改会导致 HttpPost `Edit` 方法与 HttpGet `Edit` 方法的方法签名相同，因此已重命名 `EditPost` 方法。

可选 HttpPost 编辑代码：创建和附加

建议的 HttpPost 编辑代码确保只更新已更改的列，并保留不希望包含在模型绑定内的属性中的数据。但是，读取优先的方法需要额外的数据库读取，并可能产生处理并发冲突的更复杂代码。另一种方法是将模型绑定器创建的实体附加到 EF 上下文，并将其标记为已修改。（请勿使用此代码更新项目，它只是显示一种可选的方法。）

```
public async Task<IActionResult> Edit(int id, [Bind("ID,EnrollmentDate,FirstMidName,LastName")] Student
student)
{
    if (id != student.ID)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(student);
}
```

网页 UI 包含实体中的所有字段并能更新其中任意字段时，可以使用此方法。

基架代码使用创建和附加方法，但仅捕获 `DbUpdateConcurrencyException` 异常并返回 404 错误代码。显示的示例捕获任意数据库更新异常并显示错误消息。

实体状态

数据库上下文跟踪内存中的实体是否与数据库中相应的行同步，并且此信息确定调用 `SaveChanges` 方法时会发生的情况。例如，将新实体传递到 `Add` 方法时，该实体的状态会设置为 `Added`。然后调用 `SaveChanges` 方法时，数据库上下文发出 SQL INSERT 命令。

实体可能处于以下状态之一：

- `Added`。数据库中尚不存在实体。`SaveChanges` 方法发出 INSERT 语句。
- `Unchanged`。不需要通过 `SaveChanges` 方法对此实体执行操作。从数据库读取实体时，实体将从此状态开始。
- `Modified`。已修改实体的部分或全部属性值。`SaveChanges` 方法发出 UPDATE 语句。
- `Deleted`。已标记该实体进行删除。`SaveChanges` 方法发出 DELETE 语句。
- `Detached`。数据库上下文未跟踪该实体。

在桌面应用程序中，通常会自动设置状态更改。读取一个实体并对其某些属性值做出更改。这将使其实体状态自动更改为 `Modified`。然后调用 `SaveChanges` 时，Entity Framework 生成 SQL UPDATE 语句，该语句仅更新已更改的实际属性。

在 Web 应用中，最初读取实体并显示其要编辑的数据的 `DbContext` 将在页面呈现后进行处理。调用 HttpPost `Edit` 操作方法时，将发出新的 Web 请求并且具有 `DbContext` 的新实例。如果在新的上下文中重新读取实体，则将模拟桌面处理。

但如果不希望进行额外的读取操作，则必须使用模型绑定器创建的实体对象。执行此操作最简单的方法是将实体状态设置为“已修改”，就像在之前所示的替代 HttpPost 编辑代码中完成的一样。然后调用 `SaveChanges` 时，Entity Framework 会更新数据库行的所有列，因为上下文无法知道已更改的属性。

如果想避免读取优先的方法，但还希望 SQL UPDATE 语句只更新用户实际更改的字段，则代码将更复杂。调用 HttpPost `Edit` 方法时，必须以某种方式保存初始值（如使用隐藏字段），以便初始值可用。然后可以使用初始值创建 Student 实体、调用具有初始实体版本的 `Attach` 方法、将实体的值更新为新值，再调用 `SaveChanges`。

测试编辑页

运行应用，选择"学生"选项卡，然后单击"编辑"超链接。

更改某些数据并单击"保存"。将打开"索引"页，将看到已更改的数据。

更新"删除"页

在 StudentController.cs 中，HttpGet `Delete` 方法的模板代码使用 `SingleOrDefaultAsync` 方法来检索所选的 Student 实体，如 Details 和 Edit 方法中所示。但是，若要在调用 `SaveChanges` 失败时实现自定义错误消息，请将部分功能添加到此方法及其相应的视图中。

正如所看到的更新和创建操作，删除操作需要两个操作方法。为响应 GET 请求而调用的方法将显示一个视图，使用户有机会批准或取消操作。如果用户批准，则创建 POST 请求。发生此情况时，将调用 HttpPost `Delete` 方法，然后该方法实际执行删除操作。

将 try-catch 块添加到 HttpPost `Delete` 方法，以处理更新数据库时可能出现的任何错误。如果发生错误，HttpPost Delete 方法会调用 HttpGet Delete 方法，并向其传递一个指示发生错误的参数。然后 HttpGet Delete 方法重新显示确认页以及错误消息，向用户提供取消或重试的机会。

使用以下管理错误报告的代码替换 HttpGet `Delete` 操作方法。

```
public async Task<IActionResult> Delete(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }

    return View(student);
}
```

此代码接受可选参数，指示保存更改失败后是否调用此方法。没有失败的情况下调用 HttpGet `Delete` 方法时，此参数为 false。由 HttpPost `Delete` 方法调用以响应数据库更新错误时，此参数为 true，并且将错误消息传递到视图。

HttpPost Delete 的读取优先方法

使用以下执行实际删除操作并捕获任何数据库更新错误的代码替换 HttpPost `Delete` 操作方法（名为 `DeleteConfirmed`）。

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return RedirectToAction(nameof(Index));
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}
```

此代码检索所选的实体，然后调用 `Remove` 方法以将实体的状态设置为 `Deleted`。调用 `SaveChanges` 时生成 SQL DELETE 命令。

HttpPost Delete 的创建和附加方法

如果在大容量应用程序中提高性能是优先事项，则可以通过只使用主键值实例化 Student 实体，然后将实体状态设置为 `Deleted` 来避免不必要的 SQL 查询。这是 Entity Framework 删除实体需要执行的所有操作。（请勿将此代码放在项目中；这里只是为了说明替代方法。）

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {
        Student studentToDelete = new Student() { ID = id };
        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}
```

如果实体包含还应删除的相关数据，请确保在数据库中配置了级联删除。若使用此方法删除实体，EF 可能不知道有需要删除的相关实体。

更新“删除”视图

在 Views/Student/Delete.cshtml 中，在 H2 标题和 H3 标题之间添加错误消息，如以下示例所示：

```
<h2>Delete</h2>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>
```

运行应用，选择“学生”选项卡，并单击“删除”超链接：

单击“删除”。将显示不含已删除学生的索引页。（将看到并发教程中错误处理代码的效果示例。）

关闭数据库连接

若要释放数据库连接包含的资源，完成此操作时必须尽快处理上下文实例。ASP.NET Core 内置依赖关系注入会完成此任务。

在 Startup.cs 中，调用 AddDbContext 扩展方法来预配 ASP.NET Core DI 容器中的 `DbContext` 类。默认情况下，该方法将服务生存期设置为 `Scoped`。`Scoped` 表示上下文对象生存期与 Web 请求生存期一致，并在 Web 请求结束时将自动调用 `Dispose` 方法。

处理事务

默认情况下，Entity Framework 隐式实现事务。在对多个行或表进行更改并调用 `SaveChanges` 的情况下，Entity Framework 自动确保所有更改都成功或全部失败。如果完成某些更改后发生错误，这些更改会自动回退。如果需要更多控制操作（例如，如果想要在事务中包含在 Entity Framework 外部完成的操作），请参阅事务。

非跟踪查询

当数据库上下文检索表行并创建表示它们的实体对象时，默认情况下，它会跟踪内存中的实体是否与数据库中的内容同步。更新实体时，内存中的数据充当缓存并使用该数据。在 Web 应用程序中，此缓存通常是不必要的，因为上下文实例通常生存期较短（创建新的实例并用于处理每个请求），并且通常在再次使用该实体之前处理读取实体的上下文。

可以通过调用 `AsNoTracking` 方法禁用对内存中的实体对象的跟踪。可能想要执行的典型方案包括以下操作：

- 在上下文生存期内，不需要更新任何实体，并且不需要 EF 自动加载具有由单独的查询检索的实体的导航属性。在控制器的 HttpGet 操作方法中经常遇到这些情况。
- 正在运行检索大量数据的查询，将只更新一小部分返回的数据。关闭对大型查询的跟踪可能更有效，稍后为少数需要更新的实体运行查询。
- 想要附加一个实体来更新它，但之前为了其他目的，已检索了相同的实体。由于数据库上下文已跟踪了该实体，因此无法附加要更改的实体。处理这种情况的一种方法是在早前的查询上调用 `AsNoTracking`。

有关详细信息，请参阅跟踪与非跟踪。

总结

现在拥有一组完整的页面，可对 Student 实体执行简单的 CRUD 操作。在下一个教程中，将通过添加排序、筛选和分页来扩展“索引”页。

ASP.NET Core MVC 和 EF Core - 排序、筛选、分页 - 第 3 个教程，共 10 个教程

2018/7/16 • • Edit Online

本教程没有升级至 ASP.NET Core 2.1。若要使用本教程的 ASP.NET Core 2.0 版，可以在目录表上方或者页面顶部选择“ASP.NET Core 2.0”：

相对于 2.0 版，本教程的 ASP.NET Core 2.1 Razor Pages 版有很多改进。

2.0 版教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。例如显著简化了基架代码。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果选择本教程的 Razor Pages 版，请在此 GitHub 讨论中说明原因。

作者：Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列，请参阅本系列中的第一个教程。

在上一个教程中，已为 Student 实体实现了一组网页用于执行基本的 CRUD 操作。在本教程中，将向学生索引页添加排序、筛选和分页功能。同时，还将创建一个执行简单分组的页面。

下图展示了完成操作后的页面外观。列标题是用户可以单击以按该列排序的链接。重复单击列标题可在升降排序顺序之间切换。

向学生索引页添加列排序链接

要向学生索引页添加排序功能，需更改学生控制器的 `Index` 方法并将代码添加到学生索引视图。

向 Index 方法添加排序功能

在 StudentsController.cs 中，将 `Index` 方法替换为以下代码：

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

此代码接收来自 URL 中的查询字符串的 `sortOrder` 参数。查询字符串值由 ASP.NET Core MVC 提供，作为操作方法的参数。该参数将是一个字符串，可为“Name”或“Date”，可选择后跟下划线和字符串“desc”来指定降序。默认排序顺序为升序。

首次请求索引页时，没有任何查询字符串。学生按照姓氏升序显示，这是 `switch` 语句中的 fall-through 事例所建

立的默认值。当用户单击列标题超链接时，查询字符串中会提供相应的 `sortOrder` 值。

视图使用两个 `ViewData` 元素（NameSortParm 和 DateSortParm）来为列标题超链接配置相应的查询字符串值。

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

这些是三元语句。第一个指定如果 `sortOrder` 参数为 NULL 或空，则应将 NameSortParm 设置为"name_desc"；否则，应将其设置为空字符串。通过这两个语句，视图可如下设置列标题超链接：

当前排序顺序	姓氏超链接	日期超链接
姓氏升序	descending	ascending
姓氏降序	ascending	ascending
日期升序	ascending	descending
日期降序	ascending	ascending

该方法使用 LINQ to Entities 指定要作为排序依据的列。该代码在 switch 语句之前创建一个 `IQueryable` 变量，在 switch 语句中对其进行修改，并在 `switch` 语句后调用 `ToListAsync` 方法。当创建和修改 `IQueryable` 变量时，不会向数据库发送任何查询。只有通过调用 `ToListAsync` 之类的方法将 `IQueryable` 对象转换为集合，查询才会执行。因此，此代码导致直到 `return View` 语句才会执行单个查询。

此代码可能获得包含大量列的详细信息。本系列的最后一个教程演示了如何编写在字符串变量中传递 `OrderBy` 列名的代码。

向"学生索引"视图添加列标题超链接

将 Views / Students / Index.cshtml 中的代码替换为以下代码，以添加列标题超链接。已更改的行为突出显示状态。

```
@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model => model.LastName)</a>
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model => model.EnrollmentDate)</a>
                </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

此代码使用 `ViewData` 属性中的信息来设置具有相应查询字符串值的超链接。

运行应用，选择“学生”选项卡，然后单击“姓氏”和“注册日期”列标题以验证排序是否正常工作。

向“学生索引”页添加搜索框

要向学生索引页添加筛选功能，需将文本框和提交按钮添加到视图，并在 `Index` 方法中做出相应的更改。在文本框中输入一个字符串以在名字和姓氏字段中进行搜索。

向 Index 方法添加筛选功能

在 StudentsController.cs 中，将 `Index` 方法替换为以下代码（所做的更改为突出显示状态）。

```
public async Task<IActionResult> Index(string sortOrder, string searchString)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                                || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

已向 `Index` 方法添加 `searchString` 参数。从要添加到索引视图的文本框中接收搜索字符串值。并且，还向

LINQ 语句添加了 where 子句，该子句仅选择名字或姓氏中包含搜索字符串的学生。只有在有搜索值的情况下，才会执行添加了 where 子句的语句。

> **NOTE**
>
> 此处正在调用 `IQueryable` 对象上的 `Where` 方法，并且将在服务器上处理该筛选器。在某些情况下，可能会将 `Where` 方法作为内存中集合上的扩展方法进行调用。（例如，假设将引用更改为 `_context.Students`，这样它引用的不再是 EF `DbSet`，而是一个返回 `IEnumerable` 集合的存储库方法。）结果通常是相同的，但在某些情况下可能不同。
>
> 例如，`Contains` 方法的 .NET Framework 实现在默认情况下执行区分大小写比较，但在 SQL Server 中，这由 SQL Server 实例的排序规则设置决定。该设置默认为不区分大小写。可调用 `ToUpper` 方法使测试显式不区分大小写：Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())。如果稍后更改代码以使用返回 `IEnumerable` 集合而不是 `IQueryable` 对象的存储库，则可确保结果保持不变。（在 `IEnumerable` 集合上调用 `Contains` 方法时，将获得 .NET Framework 实现；当在 `IQueryable` 对象上调用它时，将获得数据库提供程序实现。）但是，此解决方案会降低性能。`ToUpper` 代码将在 TSQL SELECT 语句的 WHERE 子句中放置一个函数。这将阻止优化器使用索引。鉴于 SQL 主要安装为不区分大小写，在迁移到区分大小写的数据存储之前，最好避免使用 `ToUpper` 代码。

向“学生索引”视图添加搜索框

在 Views/Student/Index.cshtml 中，请在打开表格标签之前立即添加突出显示的代码，以创建标题栏、文本框和搜索按钮。

```
<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["currentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
```

此代码使用 `<form>` 标记帮助器添加搜索文本框和按钮。默认情况下，`<form>` 标记帮助器使用 POST 提交表单数据，这意味着参数在 HTTP 消息正文中传递，而不是作为查询字符串在 URL 中传递。当指定 HTTP GET 时，表单数据作为查询字符串在 URL 中传递，从而使用户能够将 URL 加入书签。W3C 指南建议在操作不会导致更新时使用 GET。

运行应用，选择“学生”选项卡，输入搜索字符串，然后单击“搜索”以验证筛选是否正常工作。

请注意，该 URL 包含搜索字符串。

```
http://localhost:5813/Students?SearchString=an
```

如果将此页加入书签，当使用书签时，将获得已筛选的列表。向 `form` 标记添加 `method="get"` 是导致生成查询字符串的原因。

在此阶段，如果单击列标题排序链接，则会丢失已在“搜索”框中输入的筛选器值。此问题将在下一部分得以解决。

向“学生索引”页添加分页功能

要向学生索引页添加分页功能，需创建一个使用 `Skip` 和 `Take` 语句的 `PaginatedList` 类来筛选服务器上的数据，而不总是对表中的所有行进行检索。然后在 `Index` 方法中进行其他更改，并将分页按钮添加到 `Index` 视图。下图显示了分页按钮。

在项目文件夹中，创建 `PaginatedList.cs`，然后用以下代码替换模板代码。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T> source, int pageIndex, int
pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip((pageIndex - 1) * pageSize).Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

此代码中的 `CreateAsync` 方法将提取页面大小和页码，并将相应的 `Skip` 和 `Take` 语句应用于 `IQueryable`。当在 `IQueryable` 上调用 `ToListAsync` 时，它将返回仅包含请求页的列表。属性 `HasPreviousPage` 和 `HasNextPage` 可用于启用或禁用“上一页”和“下一页”的分页按钮。

由于构造函数不能运行异步代码，因此使用 `CreateAsync` 方法来创建 `PaginatedList<T>` 对象，而非构造函数。

向 Index 方法添加分页功能

在 StudentsController.cs 中，将 `Index` 方法替换为以下代码。

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";

    if (searchString != null)
    {
        page = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), page ?? 1, pageSize));
}
```

该代码向方法签名中添加一个页码参数、一个当前排序顺序参数和一个当前筛选器参数。

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)
```

第一次显示页面时，或者如果用户没有单击分页或排序链接，所有参数都将为 NULL。如果单击了分页链接，页面变量将包含要显示的页码。

名为 CurrentSort 的 `ViewData` 元素为视图提供当前排序顺序，因为此值必须包含在分页链接中，以便在分页时保持排序顺序相同。

名为 CurrentFilter 的 `ViewData` 元素为视图提供当前筛选器字符串。此值必须包含在分页链接中，以便在分页过

程中保持筛选器设置，并且在页面重新显示时必须将其还原到文本框中。

如果在分页过程中搜索字符串发生变化，则页面必须重置为 1，因为新的筛选器会导致显示不同的数据。在文本框中输入值并按下“提交”按钮时，搜索字符串将被更改。在这种情况下，`searchString` 参数不为 NULL。

```
if (searchString != null)
{
    page = 1;
}
else
{
    searchString = currentFilter;
}
```

在 `Index` 方法最后，`PaginatedList.CreateAsync` 方法会将学生查询转换为支持分页的集合类型中的学生的单个页面。然后将学生的单个页面传递给视图。

```
return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), page ?? 1, pageSize));
```

`PaginatedList.CreateAsync` 方法需要一个页码。两个问号表示 NULL 合并运算符。NULL 合并运算符为可为 NULL 的类型定义默认值；表达式 `(page ?? 1)` 表示如果 `page` 有值，则返回该值，如果 `page` 为 NULL，则返回 1。

向学生索引视图添加分页链接

在 Views/Students/Index.cshtml 中，将现有代码替换为以下代码。突出显示所作更改。

```
@model PaginatedList<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["currentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
            </th>
            <th>
                First Name
            </th>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
```

```
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-page="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-page="@(Model.PageIndex + 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

页面顶部的 `@model` 语句指定视图现在获取的是 `PaginatedList<T>` 对象，而不是 `List<T>` 对象。

列标题链接使用查询字符串向控制器传递当前搜索字符串，以便用户可以在筛选结果中进行排序：

```
<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-currentFilter
="@ViewData["CurrentFilter"]">Enrollment Date</a>
```

分页按钮由标记帮助器显示：

```
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-page="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
   Previous
</a>
```

运行应用并转到"学生"页。

单击不同排序顺序的分页链接，以确保分页正常工作。然后输入一个搜索字符串并再次尝试分页，以验证分页也可以正确地进行排序和筛选。

创建显示学生统计信息的"关于"页

对于 Contoso 大学网站的"关于"页，将显示每个注册日期注册了多少名学生。这需要对组进行分组和简单计算。若要完成此操作，需要执行以下操作：

- 为需要传递给视图的数据创建一个视图模型类。
- 修改主控制器中的"关于"方法。
- 修改"关于"视图。

创建视图模型

在 Models 文件夹中创建一个 SchoolViewModels 文件夹。

在新文件夹中，添加一个类文件 EnrollmentDateGroup.cs，并用以下代码替换模板代码：

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

修改主控制器

在 HomeController.cs 中，使用文件顶部的语句添加以下内容：

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
```

在类的左大括号之后立即为数据库上下文添加一个类变量，并从 ASP.NET Core DI 获取上下文的实例：

```
public class HomeController : Controller
{
    private readonly SchoolContext _context;

    public HomeController(SchoolContext context)
    {
        _context = context;
    }
```

将 `About` 方法替换为以下代码：

```
public async Task<ActionResult> About()
{
    IQueryable<EnrollmentDateGroup> data =
        from student in _context.Students
        group student by student.EnrollmentDate into dateGroup
        select new EnrollmentDateGroup()
        {
            EnrollmentDate = dateGroup.Key,
            StudentCount = dateGroup.Count()
        };
    return View(await data.AsNoTracking().ToListAsync());
}
```

LINQ 语句按注册日期对学生实体进行分组，计算每组中实体的数量，并将结果存储在 `EnrollmentDateGroup` 视图模型对象的集合中。

> **NOTE**
>
> 在 1.0 版 Entity Framework Core 中，整个结果集被返回给客户端，并在客户端上完成分组。在某些情况下，这可能会造成性能问题。请务必使用生产数据量测试性能，必要时请使用原始 SQL 在服务器上进行分组。有关如何使用原始 SQL 的信息，请参阅本系列中的最后一个教程。

修改“关于”视图

将 Views/Home/About.cshtml 文件中的代码替换为以下代码：

```
@model IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

运行应用并转到"关于"页。表格中会显示每个注册日期的学生计数。

总结

在本教程中，你已了解了如何执行排序、筛选、分页和分组。在下一个教程中，你将了解如何使用迁移来处理数据模型更改。

上一页 下一页

ASP.NET Core MVC 和 EF Core - 迁移 - 第 4 个教程(共 10 个)

2018/11/5 • • Edit Online

本教程没有升级至 ASP.NET Core 2.1。若要使用本教程的 ASP.NET Core 2.0 版，可以在目录表上方或者页面顶部选择“ASP.NET Core 2.0”：

相对于 2.0 版，本教程的 ASP.NET Core 2.1 Razor Pages 版有很多改进。

2.0 版教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。例如显著简化了基架代码。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果选择本教程的 Razor Pages 版，请在此 GitHub 讨论中说明原因。

作者：Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列，请参阅本系列中的第一个教程。

本教程使用 EF Core 迁移功能管理数据模型更改。后续教程将在更改数据模型时添加更多迁移。

迁移简介

开发新应用程序时，数据模型会频繁更改。每当模型更改时，模型都无法与数据库保持同步。本系列教程首先配置 Entity Framework 以创建数据库(如果不存在)。之后，每当更改数据模型(添加、删除或更改实体类或更改 DbContext 类)时，你都可以删除数据库，EF 将创建匹配该模型的新数据库并用测试数据为其设定种子。

这种使数据库与数据模型保持同步的方法适用于多种情况，但将应用程序部署到生产环境的情况除外。当应用程序在生产环境中运行时，它通常会存储要保留的数据，以便不会在每次更改(如添加新列)时丢失所有数据。EF Core 迁移功能可通过使 EF 更新数据库 架构而不是创建新数据库来解决此问题。

用于进行迁移的 Entity Framework Core NuGet 包

要使用迁移，可使用“包管理器控制台”(PMC) 或命令行接口 (CLI)。以下教程演示如何使用 CLI 命令。有关 PMC 的信息，请转到本教程末尾。

Microsoft.EntityFrameworkCore.Tools.DotNet 中提供了适用于命令行接口 (CLI) 的 EF 工具。若要安装此程序包，请将它添加到 .csproj 文件中的 `DotNetCliToolReference` 集合，如下所示。注意：必须通过编辑 .csproj 文件来安装此包；不能使用 `install-package` 命令或程序包管理器 GUI。若要编辑 .csproj 文件，可右键单击解决方案资源管理器中的项目名称，然后选择“编辑 ContosoUniversity.csproj”。

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
</ItemGroup>
```

（编写本教程时，本示例中的版本号是最新的。）

更改连接字符串

在 appsettings.json 文件中，将连接字符串中的数据库的名称更改为 ContosoUniversity2 或正在使用的计算机上未使用过的其他名称。

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity2;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
```

此更改将设置项目，以便初始迁移创建新的数据库。这并不是开始使用迁移的必要操作，但稍后你便会了解这样做的好处。

NOTE

除更改数据库名称外，删除数据库同样可行。使用 SQL Server 对象资源管理器 (SSOX) 或 `database drop` CLI 命令：

```
dotnet ef database drop
```

下面的部分说明如何运行 CLI 命令。

创建初始迁移

保存更改并生成项目。然后打开命令窗口并导航到项目文件夹。下面是执行此操作的快速方法：

- 在解决方案资源管理器中，右键单击项目，然后从上下文菜单中选择“在文件资源管理器中打开”。

- 在地址栏中输入“cmd”，然后按 Enter。

在命令窗口中输入以下命令：

```
dotnet ef migrations add InitialCreate
```

命令窗口中出现如下输出：

```
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\username\AppData\Local\ASP.NET\DataProtection-Keys' as key
repository and Windows DPAPI to encrypt keys at rest.
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]
      Entity Framework Core 2.0.0-rtm-26452 initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

> **NOTE**
>
> 如果出现错误消息“找不到任何匹配 "dotnet-ef" 命令的可执行文件”，请参阅此博客文章获取故障排除帮助。

如果看到错误消息“无法访问文件...ContosoUniversity.dll，因为它正被另一个进程使用。”，请在 Windows 系统托盘中找到 IIS Express 图标并右键单击，然后单击“ContosoUniversity”>“停止站点”。

了解 Up 和 Down 方法

执行 `migrations add` 命令时，EF 已生成将用于从头创建数据库的代码。此代码位于“Migrations”文件夹中名为 <timestamp>_InitialCreate.cs 的文件中。`InitialCreate` 类的 `Up` 的方法将创建与数据模型实体集相对应的数据库表，`Down` 方法将删除这些表，如下面的示例所示。

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Credits = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        // Additional code not shown
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");
        // Additional code not shown
    }
}
```

迁移调用 `Up` 方法为迁移实现数据模型更改。输入用于回退更新的命令时，迁移调用 `Down` 方法。

此代码适用于输入 `migrations add InitialCreate` 命令时所创建的初始迁移。迁移名称参数（本示例中为“InitialCreate”）用于指定文件名，并且你可以按需使用任何名称。最好选择能概括迁移中所执行操作的字词或短语。例如，可将后面的迁移命名为“AddDepartmentTable”。

如果创建初始迁移时已存在数据库，则会生成数据库创建代码，但此代码不必运行，因为数据库已与数据库模型相匹配。将应用部署到其中尚不存在数据库的其他环境时，此代码将运行以创建数据库，因此最好提前进行测试。这也是提前更改连接字符串中数据库的名称的原因，这样迁移才能从头创建新数据库。

数据模型快照

迁移在 Migrations/SchoolContextModelSnapshot.cs 中创建当前数据库架构的快照。添加迁移时，EF 会通过将数据模型与快照文件进行对比来确定已更改的内容。

删除迁移时，请使用 dotnet ef migrations remove 命令。`dotnet ef migrations remove` 删除迁移，并确保正确重置快照。

有关如何使用快照文件的详细信息，请参阅团队环境中的 EF Core 迁移。

将迁移应用到数据库

在命令窗口中，输入以下命令以创建数据库并在其中创建表。

```
dotnet ef database update
```

该命令的输出与 `migrations add` 命令的输出相似，但其中还包含设置该数据库的 SQL 命令的日志。下面的示例输出中省略了大部分日志。如果希望日志消息中不使用此详细级别，则可更改 appsettings.Development.json 文件中的日志级别。有关更多信息，请参见ASP.NET Core 中的日志记录。

```
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\username\AppData\Local\ASP.NET\DataProtection-Keys' as key
repository and Windows DPAPI to encrypt keys at rest.
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]
      Entity Framework Core 2.0.0-rtm-26452 initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (467ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      CREATE DATABASE [ContosoUniversity2];
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (20ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
          [MigrationId] nvarchar(150) NOT NULL,
          [ProductVersion] nvarchar(32) NOT NULL,
          CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );

<logs omitted for brevity>

info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20170816151242_InitialCreate', N'2.0.0-rtm-26452');
Done.
```

使用 SQL Server 对象资源管理器检查数据库(与第一个教程中的做法相同)。你会发现添加了 __EFMigrationsHistory 表，该表可用于跟踪已应用到数据库的迁移。查看该表中的数据，其中显示对应初始迁移的一行数据。(上面的 CLI 输出示例中最后部分的日志显示了创建此行的 INSERT 语句。)

运行应用程序以验证所有内容照旧运行。

命令行接口 (CLI) 与包管理器控制台 (PMC)

可通过 .NET Core CLI 命令或 Visual Studio 包管理器控制台 (PMC) 窗口中的 PowerShell cmdlet 使用可管理迁移的 EF 工具。本教程演示如何使用 CLI，但也可以根据喜好使用 PMC。

适用于 PMC 命令的 EF 命令位于 Microsoft.EntityFrameworkCore.Tools 程序包中。此包包含在

Microsoft.AspNetCore.App 元包中，因此，如果应用具有对 `Microsoft.AspNetCore.App` 的包引用，则无需添加包引用。

重要说明：此程序包与通过编辑 .csproj 文件为 CLI 安装的程序包不同。此程序包的名称以 `Tools` 结尾，而 CLI 程序包的名称以 `Tools.DotNet` 结尾。

有关 CLI 命令的详细信息，请参阅 .NET Core CLI。

有关 PMC 命令的详细信息，请参阅包管理器控制台 (Visual Studio)。

总结

本教程已介绍如何创建并应用初始迁移。下一教程将介绍有关展开数据模型的更高级主题。同时还将介绍创建并应用其他迁移的方法。

上一页 下一页

ASP.NET Core MVC 和 EF Core - 数据模型 - 第 5 个教程(共 10 个)

2018/10/30 • • Edit Online

本教程没有升级至 ASP.NET Core 2.1。若要使用本教程的 ASP.NET Core 2.0 版，可以在目录表上方或者页面顶部选择“ASP.NET Core 2.0”：

相对于 2.0 版，本教程的 ASP.NET Core 2.1 Razor Pages 版有很多改进。

2.0 版教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。例如显著简化了基架代码。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果选择本教程的 Razor Pages 版，请在此 GitHub 讨论中说明原因。

作者：Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列，请参阅本系列中的第一个教程。

之前的教程介绍了由三个实体组成的简单数据模型。本教程将添加更多实体和关系，并通过指定格式化、验证和数据库映射规则来自定义数据模型。

完成本教程后，实体类将构成下图所示的完整数据模型：

使用特性自定义数据模型

本节介绍如何使用指定格式化、验证和数据库映射规则的特性来自定义数据模型。随后在接下来的几节中创建完整的学校数据模型，创建方法：向已创建的类添加特性，并为模型中剩余的实体类型创建新类。

DataType 特性

对于学生注册日期，目前所有网页都显示有时间和日期，尽管对此字段而言重要的只是日期。使用数据注释特性，可更改一次代码，修复每个视图中数据的显示格式。若要查看如何执行此操作，请向 `Student` 类的 `EnrollmentDate` 属性添加一个特性。

在 Models/Student.cs 中，为 `System.ComponentModel.DataAnnotations` 命名空间添加一个 `using` 语句，将 `DataType` 和 `DisplayFormat` 特性添加到 `EnrollmentDate` 属性，如以下示例所示：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`DataType` 属性用于指定比数据库内部类型更具体的数据类型。在此示例中，我们只想跟踪日期，而不是日期和时间。`DataType` 枚举提供了多种数据类型，例如日期、时间、电话号码、货币、电子邮件地址等。应用程序还可通过 `DataType` 特性自动提供类型特定的功能。例如，可以为 `DataType.EmailAddress` 创建 `mailto:` 链接，并且可以在支持 HTML5 的浏览器中为 `DataType.Date` 提供日期选择器。`DataType` 特性发出 HTML 5 `data-`（读作 data dash）特性供 HTML 5 浏览器理解。`DataType` 特性不提供任何验证。

`DataType.Date` 不指定显示日期的格式。默认情况下，数据字段根据默认格式显示，后者以服务器的 CultureInfo 为基础。

`DisplayFormat` 特性用于显式指定日期格式：

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

`ApplyFormatInEditMode` 设置指定在文本框中显示值以进行编辑时也应用格式。（你可能不想为某些字段执行此操作 — 例如，对于货币值，你可能不希望文本框中的货币符号可编辑。）

可单独使用 `DisplayFormat` 特性，但通常建议同时使用 `DataType` 特性。`DataType` 特性传达数据的语义而不是传达数据在屏幕上的呈现方式，并提供 `DisplayFormat` 不具备的以下优势：

- 浏览器可启用 HTML5 功能（例如，显示日历控件、区域设置适用的货币符号、电子邮件链接、某种客户端输入验证等）。
- 默认情况下，浏览器将根据区域设置采用正确的格式呈现数据。

有关详细信息，请参阅 <input> 标记帮助器文档。

运行应用，进入“学生”索引页，注意注册日期中不再显示时间。任何使用“学生”模型的视图都是如此。

StringLength 特性

还可使用特性指定数据验证规则和验证错误消息。`StringLength` 特性设置数据库中的最大长度，并为 ASP.NET Core MVC 提供客户端和服务器端验证。还可在此属性中指定最小字符串长度，但最小值对数据库架构没有影响。

假设要确保用户输入的名称不超过 50 个字符。若要添加此限制，请将 `StringLength` 特性添加到 `LastName` 和 `FirstMidName` 属性，如下例所示：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`StringLength` 特性不会阻止用户在名称中输入空格。可使用 `RegularExpression` 特性应用输入限制。例如，以下代码要求第一个字符为大写，其余字符按字母顺序排列：

```
[RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$")]
```

`MaxLength` 特性的作用与 `StringLength` 特性类似，但不提供客户端验证。

现在数据库模型发生了变化，需要更改数据库架构。请使用迁移来更新架构，迁移时不会丢失通过应用程序 UI 添加到数据库的任何数据。

保存更改并生成项目。随后打开项目文件夹中的命令窗口并输入以下命令：

```
dotnet ef migrations add MaxLengthOnNames
```

```
dotnet ef database update
```

`migrations add` 命令发出警告：可能出现数据丢失，因为更改后，有两列的最大长度缩短。迁移时会创建一个名为 <timeStamp>_MaxLengthOnNames.cs 的文件。此文件包含 `Up` 方法中的代码，该代码将更新数据库以匹配当前数据模型。`database update` 命令运行该代码。

Entity Framework 使用迁移文件名的前缀时间戳发出迁移命令。可在运行 update-database 命令前创建多个迁移，然后按照创建顺序应用所有迁移。

运行该应用，选择"学生"选项卡，单击"新建"，然后输入名称（超过 50 个字符）。单击"创建"时，客户端验证会显示一条错误消息。

Column 特性

还可使用特性来控制类和属性映射到数据库的方式。假设在名字字段使用了 `FirstMidName`，这是因为该字段也可能包含中间名。但却希望将数据库列命名为 `FirstName`，因为要针对数据库编写即席查询的用户习惯使用该姓名。若要进行此映射，可使用 `Column` 特性。

`Column` 特性指定，创建数据库时，映射到 `FirstMidName` 属性的 `Student` 表的列将被命名为 `FirstName`。换言之，在代码引用 `Student.FirstMidName` 时，数据来源将是 `Student` 表的 `FirstName` 列或在其中进行更新。如果不指定列名称，则其名称与属性名称相同。

在 Student.cs 文件中，为 `System.ComponentModel.DataAnnotations.Schema` 添加一个 `using` 语句，并将列名称特性

添加到 `FirstMidName` 属性，如以下突出显示的代码所示：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

添加 `Column` 特性后，`SchoolContext` 的支持模型会发生改变，与数据库不再匹配。

保存更改并生成项目。随后打开项目文件夹中的命令窗口，输入以下命令，创建另一个迁移：

```
dotnet ef migrations add ColumnFirstName
```

```
dotnet ef database update
```

在 SQL Server 对象资源管理器中，双击 Student 表，打开 Student 表设计器。

在进行前两次迁移前，名称列的类型为 nvarchar (MAX)。而现在则是 nvarchar (50)，列名从 FirstMidName 变为 FirstName。

> **NOTE**
> 如果尚未按以下各节所述创建所有实体类就尝试进行编译，则可能会出现编译器错误。

Student 实体的最终更改

在 Models/Student.cs 中，将之前添加的代码替换为以下代码。突出显示所作更改。

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Required 特性

`Required` 特性使名称属性成为必填字段。值类型（DateTime、int、double、float 等）等不可为 null 的类型不需要 `Required` 特性。系统会将不可为 null 的类型自动视为必填字段。

可删除 `Required` 特性，并用 `StringLength` 特性的最小长度参数来替换：

```
[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }
```

Display 特性

`Display` 特性指定文本框的标题应是“名”、“姓”、“全名”和“注册日期”，而不是每个实例中的属性名称（其中没有分隔单词的空格）。

FullName 计算属性

`FullName` 是计算属性，可返回通过串联两个其他属性创建的值。因此它只有一个 get 访问器，且数据库中不会生成 `FullName` 列。

创建 Instructor 实体

创建 Models/Instructor.cs，使用以下代码替换模板代码：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

请注意，在 Student 和 Instructor 实体中有几个属性是相同的。本系列后面的实现继承教程将重构此代码以消除冗余。

可在一行中放置多个特性，因此也可以按如下所示编写 `HireDate` 特性：

```
[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

CourseAssignments 和 OfficeAssignment 导航属性

`CourseAssignments` 和 `OfficeAssignment` 是导航属性。

一名讲师可以教授任意数量的课程，因此 `CourseAssignments` 定义为集合。

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

如果导航属性可包含多个实体，则其类型必须是可添加、可删除和可更新实体的列表。可指定 `ICollection<T>` 或诸如 `List<T>` 或 `HashSet<T>` 的类型。如果指定 `ICollection<T>`，EF 默认会创建一个 `HashSet<T>` 集合。

下面在关于多对多关系的部分中解释了这些作为 `CourseAssignment` 实体的原因。

Contoso University 业务规则规定，讲师最多只能有一个办公室，因此 `OfficeAssignment` 属性拥有一个 OfficeAssignment 实体（如果未分配办公室，则该实体可能为 null）。

```
public OfficeAssignment OfficeAssignment { get; set; }
```

创建 OfficeAssignment 实体

用以下代码创建 Models/OfficeAssignment.cs：

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

Key 特性

讲师与 OfficeAssignment 实体间存在一对零或一的关系。办公室分配仅与分配有办公室的讲师相关，因此其主键也是 Instructor 实体的外键。但 Entity Framework 无法自动识别 InstructorID 作为此实体的主键，因为其名称不符合 ID 或 classnameID 命名约定。因此，`Key` 特性用于将其识别为主键：

```
[Key]
public int InstructorID { get; set; }
```

如果实体具有自己的主键，但你希望使用 classnameID 或 ID 以外的其他属性名称，则也可使用 `Key` 特性。

默认情况下，EF 将键视为非数据库生成，因为该列面向的是识别关系。

Instructor 导航属性

Instructor 实体具有可为 null `OfficeAssignment` 导航属性（因为可能未向讲师分配办公室），而 OfficeAssignment 实体具有不可为 null `Instructor` 导航属性（因为在没有讲师的情况下不会分配办公室 - `InstructorID` 不可为 null）。Instructor 实体具有相关 OfficeAssignment 实体时，每个实体都将在其导航属性中引用另一实体。

可向 Instructor 导航属性添加 `[Required]` 特性，指定必须有相关讲师，但这不是必须的，因为 `InstructorID` 外键（也是此表的键）不可为 null。

修改 Course 实体

在 Models/Course.cs 中，将之前添加的代码替换为以下代码。突出显示所作更改。

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

Course 实体具有外键属性 `DepartmentID`，该属性指向相关 Department 实体，同时它还具有 `Department` 导航属性。

如果拥有相关实体的导航属性，则 Entity Framework 不会要求为数据模型添加外键属性。只要有需要，EF 就会自动在数据库中创建外键，并为其创建阴影属性。但如果数据模型包含外键，则更新会变得更简单、更高效。例如，提取 Course 实体进行编辑时，如果未加载该实体，那么 Department 实体为 null，因此，更新 Course 实体时，必须先提取 Department 实体。数据模型中包含外键属性 `DepartmentID` 时，更新前无需提取 Department 实体。

DatabaseGenerated 特性

`CourseID` 属性上具有 `None` 参数的 `DatabaseGenerated` 特性指定主键值由用户提供，而不是由数据库生成。

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

默认情况下，Entity Framework 假定主键值由数据库生成。大多数情况下，这是理想情况。但对 Course 实体而言，需使用用户指定的课程编号，例如一个系为 1000 系列，另一个系为 2000 系列等。

`DatabaseGenerated` 特性也可用于生成默认值，如在用于记录行创建或更新日期的数据库列中。有关详细信息，请参阅生成的属性。

外键和导航属性

Course 实体中的外键属性和导航属性可反映以下关系：

向一个系分配课程后，出于上述原因，会出现 `DepartmentID` 外键和 `Department` 导航属性。

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

参与一门课程的学生数量不定，因此 `Enrollments` 导航属性是一个集合：

```
public ICollection<Enrollment> Enrollments { get; set; }
```

一门课程可能有多位授课讲师，因此 `CourseAssignments` 导航属性是一个集合（稍后会解释 `CourseAssignment` 类型）：

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

创建 Department 实体

用以下代码创建 Models/Department.cs：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

Column 特性

`Column` 特性之前用于更改列名称映射。在 Department 实体的代码中，`Column` 特性用于更改 SQL 数据类型映射，以便使用数据库中的 SQL Server 货币类型来定义该列：

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

通常不需要列映射，因为 Entity Framework 会根据你为属性定义的 CLR 类型选择适当的 SQL Server 数据类型。CLR `decimal` 类型会映射到 SQL Server `decimal` 类型。但假如你知道该列要表示货币金额，那么货币数据类型会更加合适。

外键和导航属性

外键和导航属性可反映以下关系：

一个系可能有也可能没有管理员，而管理员始终是讲师。因此 `InstructorID` 属性作为 Instructor 实体内外键，且 `int` 类型指定后跟有一个问号，将该属性标记为可为 null。导航属性名为 `Administrator`，但其中包含 Instructor 实体：

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

一个系可以有多门课程，因此存在 Course 导航属性：

```
public ICollection<Course> Courses { get; set; }
```

NOTE

按照约定，Entity Framework 能针对不可为 null 的外键和多对多关系启用级联删除。这可能导致循环级联删除规则，尝试添加迁移时该规则会造成异常。例如，如果未将 Department.InstructorID 属性定义为可为 null，那么在删除系时，EF 会配置级联删除规则来删除讲师，这是预期外的情况。如果业务规则要求 `InstructorID` 属性不可为 null，则必须使用以下 Fluent API 语句禁用关系中的级联删除：

```
modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

修改 Enrollment 实体

在 Models/Enrollment.cs 中，将之前添加的代码替换为以下代码：

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

外键和导航属性

外键属性和导航属性可反映以下关系：

注册记录面向一门课程，因此存在 `CourseID` 外键属性和 `Course` 导航属性：

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

注册记录面向一名学生，因此存在 `StudentID` 外键属性和 `Student` 导航属性：

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

多对多关系

Student 和 Course 实体间存在多对多关系，Enrollment 实体在数据库中充当带有效负载的多对多联接表。“带有效负载”是指 Enrollment 表包含除联接表外键之外的其他数据（在此示例中为主键和 Grade 属性）。

下图显示这些关系在实体关系图中的外观。（该图是使用 EF 6.x 的 Entity Framework Power Tools 生成的，教程未介绍如何创建该图，此处仅作为示例使用。）

每条关系线的一端显示 1，另一端显示星号 (*)，这表示一对多关系。

如果 Enrollment 表不包含年级信息，则它只需包含两个外键（CourseID 和 StudentID）。在这种情况下，该表是数据库中不带有效负载的多对多联接表（或纯联接表）。Instructor 和 Course 实体具有这种多对多关系，下一步是创建实体类，将其作为不带有效负载的联接表。

（EF 6.x 支持多对多关系的隐式联接表，但 EF Core 不支持。有关详细信息，请参阅 EF Core GitHub 存储库中的讨论。）

CourseAssignment 实体

用以下代码创建 Models/CourseAssignment.cs：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

联接实体名称

数据库中的 Instructor 与 Course 多对多关系需要联接表，并且必须由实体集表示。将联接实体命名为 `EntityName1EntityName2` 很常见，在此示例中实体名为 `CourseInstructor`。但是，建议选择一个可描述关系的名称。数据模型开始很简单，且会不断增长，随后无有效负载联接会频繁获取有效负载。如果一开始实体名称为描述性名称，那么之后就不必更改名称。理想情况下，联接实体在业务域中可能具有自己的自带名称（可能是单个字）。例如，可通过 Rating 链接 Book 和 Customer。对于此关系，相比 `CourseInstructor`，`CourseAssignment` 是更好的选择。

组合键

由于外键不可为 null，且它们共同唯一标识表的每一行，因此不需要单独的主键。InstructorID 和 CourseID 属性应充当组合主键。标识 EF 组合主键的唯一方法是使用 Fluent API（无法借助特性来完成）。下一节将介绍如何配置组合主键。

在一个课程可以有多个行，一个讲师可以有多个行的情况下，组合键可确保同一讲师和课程不会有多个行。`Enrollment` 联接实体定义其自己的主键，因此可能会出现此类重复。若要防止出现此类重复，可在外键字段上添加唯一索引，或使用类似于 `CourseAssignment` 的主组合键配置 `Enrollment`。有关详细信息，请参阅索引。

更新数据库上下文

将以下突出显示的代码添加到 Data/SchoolContext.cs 文件：

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

此代码添加新实体并配置 CourseAssignment 实体的组合主键。

用 Fluent API 替代特性

`DbContext` 类的 `OnModelCreating` 方法中的代码使用 Fluent API 来配置 EF 行为。API 称为“Fluent”，因为它通常在将一系列方法调用连接成单个语句后才能使用，如 EF Core 文档 中的此示例所示：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

本教程仅将 Fluent API 用于数据库映射，这是无法使用特性实现的。但 Fluent API 还可用于指定大多数格式化、验证和映射规则，这可通过特性完成。`MinimumLength` 等特性不能通过 Fluent API 应用。如前所述，`MinimumLength` 不会更改架构，它只应用客户端和服务器端验证规则。

某些开发者倾向于仅使用 Fluent API 以保持实体类的“纯净”。如有需要，可混合使用特性和 Fluent API，且有些自定义只能通过 Fluent API 实现，但通常建议选择一种方法并尽可能坚持使用这一种。如果确实要使用两种，请注意，只要出现冲突，Fluent API 就会覆盖特性。

有关特性和 Fluent API 的详细信息，请参阅配置方法。

显示关系的实体关系图

下图显示 Entity Framework Power Tools 针对已完成的学校模型创建的关系图。

除一对多关系线(1到 *)外，此处还显示了 Instructor 和 OfficeAssignment 实体间的一对零或一关系线(1 到 0..1)，以及 Instructor 和 Department 实体间的零对多或一对多关系线(0..1 到 *)。

使用测试数据设定数据库种子

使用以下代码替换 Data/DbInitializer.cs 文件中的代码，从而为创建的新实体提供种子数据。

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
```

```
{
    return;   // DB has been seeded
}

var students = new Student[]
{
    new Student { FirstMidName = "Carson",   LastName = "Alexander",
        EnrollmentDate = DateTime.Parse("2010-09-01") },
    new Student { FirstMidName = "Meredith", LastName = "Alonso",
        EnrollmentDate = DateTime.Parse("2012-09-01") },
    new Student { FirstMidName = "Arturo",   LastName = "Anand",
        EnrollmentDate = DateTime.Parse("2013-09-01") },
    new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
        EnrollmentDate = DateTime.Parse("2012-09-01") },
    new Student { FirstMidName = "Yan",      LastName = "Li",
        EnrollmentDate = DateTime.Parse("2012-09-01") },
    new Student { FirstMidName = "Peggy",    LastName = "Justice",
        EnrollmentDate = DateTime.Parse("2011-09-01") },
    new Student { FirstMidName = "Laura",    LastName = "Norman",
        EnrollmentDate = DateTime.Parse("2013-09-01") },
    new Student { FirstMidName = "Nino",     LastName = "Olivetto",
        EnrollmentDate = DateTime.Parse("2005-09-01") }
};

foreach (Student s in students)
{
    context.Students.Add(s);
}
context.SaveChanges();

var instructors = new Instructor[]
{
    new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie",
        HireDate = DateTime.Parse("1995-03-11") },
    new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",
        HireDate = DateTime.Parse("2002-07-06") },
    new Instructor { FirstMidName = "Roger",   LastName = "Harui",
        HireDate = DateTime.Parse("1998-07-01") },
    new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
        HireDate = DateTime.Parse("2001-01-15") },
    new Instructor { FirstMidName = "Roger",   LastName = "Zheng",
        HireDate = DateTime.Parse("2004-02-12") }
};

foreach (Instructor i in instructors)
{
    context.Instructors.Add(i);
}
context.SaveChanges();

var departments = new Department[]
{
    new Department { Name = "English",     Budget = 350000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID  = instructors.Single( i => i.LastName == "Abercrombie").ID },
    new Department { Name = "Mathematics", Budget = 100000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID  = instructors.Single( i => i.LastName == "Fakhouri").ID },
    new Department { Name = "Engineering", Budget = 350000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID  = instructors.Single( i => i.LastName == "Harui").ID },
    new Department { Name = "Economics",   Budget = 100000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID  = instructors.Single( i => i.LastName == "Kapoor").ID }
};

foreach (Department d in departments)
{
    context.Departments.Add(d);
```

```
            context.Departments.Add(d);
        }
        context.SaveChanges();

        var courses = new Course[]
        {
            new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
                DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
            },
            new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
                DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
            },
            new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
                DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
            },
            new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
                DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
            },
            new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
                DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
            },
            new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
                DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
            },
            new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
                DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
            },
        };

        foreach (Course c in courses)
        {
            context.Courses.Add(c);
        }
        context.SaveChanges();

        var officeAssignments = new OfficeAssignment[]
        {
            new OfficeAssignment {
                InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
                Location = "Smith 17" },
            new OfficeAssignment {
                InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
                Location = "Gowan 27" },
            new OfficeAssignment {
                InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
                Location = "Thompson 304" },
        };

        foreach (OfficeAssignment o in officeAssignments)
        {
            context.OfficeAssignments.Add(o);
        }
        context.SaveChanges();

        var courseInstructors = new CourseAssignment[]
        {
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
                InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
                },
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
                InstructorID = instructors.Single(i => i.LastName == "Harui").ID
                },
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
                InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
                },
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
```

```
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
            },
    };

    foreach (CourseAssignment ci in courseInstructors)
    {
        context.CourseAssignments.Add(ci);
    }
    context.SaveChanges();

    var enrollments = new Enrollment[]
    {
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            Grade = Grade.A
        },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
            Grade = Grade.C
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
            Grade = Grade.B
            },
        new Enrollment {
```

```
                    StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Li").ID,
                    CourseID = courses.Single(c => c.Title == "Composition").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Justice").ID,
                    CourseID = courses.Single(c => c.Title == "Literature").CourseID,
                    Grade = Grade.B
                    }
            };

            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollments.Where(
                    s =>
                            s.Student.ID == e.StudentID &&
                            s.Course.CourseID == e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
                    context.Enrollments.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

如第一个教程所述，大部分此类代码仅创建新实体对象，并按测试要求将示例数据加载到属性中。注意多对多关系的处理方法：代码在 `Enrollments` 和 `CourseAssignment` 联接实体集中创建实体，以此来创建关系。

添加迁移

保存更改并生成项目。然后打开项目文件夹中的命令窗口，输入 `migrations add` 命令（先不要执行 update-database 命令）：

```
dotnet ef migrations add ComplexDataModel
```

会出现数据可能丢失的警告。

```
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

如果此时尝试运行 `database update` 命令（先不要执行此操作），则会出现以下错误：

> ALTER TABLE 语句与 FOREIGN KEY 约束“FK_dbo.Course_dbo.Department_DepartmentID”冲突。冲突发生位置：数据库“ContosoUniversity”、表“dbo.Department”和列“DepartmentID”。

有时使用现有数据执行迁移时，需将存根数据插入数据库，满足外键约束。`Up` 方法中生成的代码将不可为 null 的 DepartmentID 外键添加到 Course 表中。如果代码运行时，Course 表中已经有了行，则 `AddColumn` 操作失败，因为 SQL Server 不知道要向不可为 null 的列中放入什么值。本教程将在新数据库上运行迁移，但在生产应用程序中，必须使迁移处理现有数据，因此下方通过示例介绍如何执行此操作。

为使此迁移处理现有数据，必须更改代码，赋予新列默认值，并创建一个名为“Temp”的存根系，作为默认系。之后，Course 行将在 `Up` 方法运行后与“Temp”系建立联系。

- 打开 {timestamp}_ComplexDataModel.cs 文件。
- 对将 DepartmentID 列添加到 Course 表的代码行添加注释。

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

- 在创建 Department 表的代码后添加以下突出显示的代码：

```
migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(nullable: true),
        Name = table.Column<string>(maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00,
GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

在生产应用程序中，可编写代码或脚本来添加 Department 行并将 Course 行与新 Department 行相关联。随后将不在需要“Temp”系或 Course.DepartmentID 列中的默认值。

保存更改并生成项目。

更改连接字符串并更新数据库

现在 `DbInitializer` 类中就有了新代码，可将新实体的种子数据添加到空数据库。若要让 EF 创建新的空数据库，请将 appsettings.json 中连接字符串内的数据库名称更改为 ContosoUniversity3 或正在使用的计算机上未使

用过的其他名称。

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
```

将更改保存到 appsettings.json。

> **NOTE**
>
> 除更改数据库名称外，删除数据库同样可行。使用 SQL Server 对象资源管理器 (SSOX) 或 `database drop` CLI 命令：
>
> ```
> dotnet ef database drop
> ```

更改数据库名称或删除数据库后，在命令窗口运行 `database update` 命令，执行迁移。

```
dotnet ef database update
```

运行应用，使 `DbInitializer.Initialize` 方法运行并填充新数据库。

像之前一样在 SSOX 中打开数据库，然后展开 Tables 节点，查看是否已创建所有表。（如果之前打开的 SSOX 尚未关闭，请单击"刷新"按钮。）

运行应用，触发设定数据库种子的初始化代码。

右键单击"CourseAssignment"表，然后选择"查看数据"，验证其中是否存在数据。

CourseID	InstructorID
2021	1
2042	1
1045	2
1050	3
3141	3
1050	4
4022	5
4041	5
NULL	NULL

总结

现在你就得到了更复杂的数据模型和相应的数据库。后面教程将更多详细的介绍如何访问相关数据。

上一页 下一页

ASP.NET Core MVC 和 EF Core - 读取相关数据 - 第 6 个课程(共 10 个课程)

2018/7/16 • • Edit Online

本教程没有升级至 ASP.NET Core 2.1。若要使用本教程的 ASP.NET Core 2.0 版，可以在目录表上方或者页面顶部选择“ASP.NET Core 2.0”：

相对于 2.0 版，本教程的 ASP.NET Core 2.1 Razor Pages 版有很多改进。

2.0 版教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。例如显著简化了基架代码。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果选择本教程的 Razor Pages 版，请在此 GitHub 讨论中说明原因。

作者：Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列，请参阅本系列中的第一个教程。

前面教程创建了学校数据模型。本教程将读取并显示相关数据 - 即 Entity Framework 加载到导航属性中的数据。

下图是将会用到的页面。

相关数据的预先加载、显式加载和延迟加载

对象关系映射 (ORM) 软件(如 Entity Framework)可通过多种方式将相关数据加载到实体的导航属性中:

- 预先加载。读取该实体时，会同时检索相关数据。此时通常会出现单一联接查询，检索所有必需数据。可使用 `Include` 和 `ThenInclude` 方法指定 Entity Framework Core 中的预先加载。

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

可在单独查询中检索一些数据，EF 会"修正"导航属性。也就是说，EF 会自动添加单独检索的实体，将其添加到之前检索的实体的导航属性中所属的位置。对于检索相关数据的查询，可使用 `Load` 方法，而不采用返回列表或对象的方法，如 `ToList` 或 `Single`。

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- 显式加载。首次读取实体时，不检索相关数据。如有需要，可编写检索相关数据的代码。就像使用单独查询进行预先加载一样，显式加载时会向数据库发送多个查询。二者的区别在于，代码通过显式加载指定要加载的导航属性。在 Entity Framework Core 1.1 中，可使用 `Load` 方法执行显式加载。例如:

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- 延迟加载。首次读取实体时，不检索相关数据。然而，首次尝试访问导航属性时，会自动检索导航属性所需的数据。每次首次尝试从导航属性获取数据时，都向数据库发送查询。Entity Framework Core 1.0 不支持延迟加载。

性能注意事项

如果知道自己需要每个检索的实体的相关数据，选择预先加载可获得最佳性能，因为相比每个检索的实体的单独查询，发送到数据库的单个查询更加有效。例如，假设每个系有十个相关课程。预先加载所有相关数据时，只会进行单一(联接)查询，往返数据库一次。单独查询每个系的课程时，会往返数据库十一次。延迟较高时，额外往返数据库对性能尤为不利。

另一方面，在某些情况下，单独查询会更加高效。在一个查询中预先加载所有相关数据时，可能会生成一个非常复杂的联接，SQL Server 无法有效处理该联接。或者，如果你正在处理一组实体且只需访问其子集的导航属性，那么采用单独查询可获得更佳性能，因为预先加载所有数据后，会检索不需要的数据。如果看重性能，那么最好测试两种方式的性能，以便做出最佳选择。

创建显示院系名称的“课程”页

Course 实体包括导航属性，其中包含分配有课程的系的 Department 实体。若要在课程列表中显示接受分配的系的名称，需从位于 `Course.Department` 导航属性中的 Department 实体获取 Name 属性。

使用与带视图的 MVC 控制器相同的选项，及之前用于学生控制器的 Entity Framework 基架为 Course 实体类型创建名为 CoursesController 的控制器，如下图所示:

打开 CoursesController.cs 并检查 `Index` 方法。自动基架使用 `Include` 方法为 `Department` 导航属性指定了预先加载。

将 `Index` 方法替换为以下代码，该代码为返回 Course 实体（是 `courses` 而不是 `schoolContext`）的 `IQueryable` 赋予了更合适的名称：

```
public async Task<IActionResult> Index()
{
    var courses = _context.Courses
        .Include(c => c.Department)
        .AsNoTracking();
    return View(await courses.ToListAsync());
}
```

在 Views/Courses/Index.cshtml 中，将模板代码替换为以下代码。突出显示所作更改：

```
@model IEnumerable<ContosoUniversity.Models.Course>

@{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

已对基架代码进行了如下更改：

- 将标题从“索引”更改为“课程”。
- 添加了显示 `CourseID` 属性值的“数字”列。默认情况下，不针对主键进行架构，因为对最终用户而言，它们通常没有意义。但在这种情况下主键是有意义的，而你需要将其呈现出来。
- 更改“院系”列，显示院系名称。该代码显示已加载到 `Department` 导航属性中的 Department 实体的 `Name` 属性：

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

运行应用并选择“课程”选项卡，查看包含系名称的列表。

创建显示“课程”和“注册”的“讲师”页

本节将为 Instructor 实体创建一个控制器和视图，从而显示“讲师”页：

该页面通过以下方式读取和显示相关数据：

- 讲师列表显示 OfficeAssignment 实体的相关数据。Instructor 与 OfficeAssignment 实体间存在一对零或一的关系。将预先加载 OfficeAssignment 实体。如前所述，需要主表所有检索行的相关数据时，预先加载通常更有效。在这种情况下，你希望显示所有显示的讲师的办公室分配情况。
- 用户选择一名讲师时，显示相关 Course 实体。Instructor 和 Course 实体是多对多关系。预先加载 Course 实体及其相关 Department 实体。在这种情况下，单独查询可能更有效，因为仅需显示所选讲师的课程。但此示例显示的是如何在本身就位于导航属性内的实体中预先加载导航属性。
- 用户选择一门课程时，会显示 Enrollment 实体集的相关数据。Course 和 Enrollment 实体是一对多关系。单独查询 Enrollment 实体及其相关 Student 实体。

创建"讲师索引"视图的视图模型

"讲师"页显示来自三个不同表格的数据。因此将创建包含三个属性的视图模型，每个属性都包含一个表的数据。

在 SchoolViewModels 文件夹中，创建 InstructorIndexData.cs，并使用以下代码替换现有代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

创建讲师控制器和视图

使用 EF 读/写操作创建讲师控制器，如下图所示：

打开 InstructorsController.cs 并为 ViewModels 名称空间添加 using 语句：

```
using ContosoUniversity.Models.SchoolViewModels;
```

使用以下代码替换 Index 方法，预先加载相关数据并将其放入视图模型。

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}
```

该方法接受可选路由数据（`id`）和查询字符串参数（`courseID`），二者提供所选讲师和课程的 ID 值。参数由页面上的“选择”超链接提供。

代码先创建一个视图模型实例，并在其中放入讲师列表。代码指定预先加载 `Instructor.OfficeAssignment` 和 `Instructor.CourseAssignments` 导航属性。在 `CourseAssignments` 属性中，加载 `Course` 属性，在其中加载 `Enrollments` 和 `Department` 属性，同时在每个 `Enrollment` 实体中加载 `Student` 属性。

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

由于视图始终需要 OfficeAssignment 实体，因此更有效的做法是在同一查询中获取。在网页中选择讲师后，需要 Course 实体，因此只有在页面频繁显示选中课程时，单个查询才比多个查询更有效。

代码重复 `CourseAssignments` 和 `Course`，因为你需要 `Course` 中的两个属性。`ThenInclude` 调用的第一个字符串获取 `CourseAssignment.Course`、`Course.Enrollments` 和 `Enrollment.Student`。

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

此时，代码中的另一个 `ThenInclude` 将成为 `Student` 的导航属性，你不需要该属性。但调用 `Include` 是从 `Instructor` 属性重新开始，因此必须再次遍历该链，这次指定 `Course.Department` 而不是 `Course.Enrollments`。

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

选择讲师时，将执行以下代码。从视图模型中的讲师列表检索所选讲师。然后，该视图模型的 `Courses` 属性加载讲师 `CourseAssignments` 导航属性中的 Course 实体。

```
if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
    viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

`Where` 方法返回一个集合，但在这种情况下，向该方法传入条件后，只返回一个 Instructor 实体。`Single` 方法将集合转换为单个 Instructor 实体，让你可以访问该实体的 `CourseAssignments` 属性。`CourseAssignments` 属性包含多个 `CourseAssignment` 实体，而你只需要相关的 `Course` 实体。

如果知道集合只有一个项，则可在集合上使用 `Single` 方法。如果集合为空或包含多个项，Single 方法将引发异常。还可使用 `SingleOrDefault`，该方式在集合为空时返回默认值（本例中为 null）。但在这种情况下，仍会引发异常（尝试在 null 引用上查找 `Courses` 属性时），并且异常消息不会清楚指出异常原因。调用 `Single` 方法时，还可传入 Where 条件，而不是分别调用 `Where` 方法：

```
.Single(i => i.ID == id.Value)
```

而不是：

```
.Where(i => i.ID == id.Value).Single()
```

接着，如果选择了课程，则从视图模型中的课程列表中检索所选课程。然后，视图模型的 `Enrollments` 属性加载

该课程 `Enrollments` 导航属性中的 Enrollment 实体。

```
if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

修改讲师索引视图

在 Views/Instructors/Index.cshtml 中，将模板代码替换为以下代码。突出显示所作更改。

```
@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructors)
        {
            string selectedRow = "";
            if (item.ID == (int?)ViewData["InstructorID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @:  @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-action="Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
           }
    </tbody>
</table>
```

已对现有代码进行了如下更改:

- 将模型类更改为了 `InstructorIndexData`。
- 将页标题从“索引”更改为了“讲师”。
- 添加了仅在 `item.OfficeAssignment` 不为 null 时才显示 `item.OfficeAssignment.Location` 的“办公室”列。（由于这是一对零或一的关系，因此可能没有相关的 OfficeAssignment 实体。）

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

- 添加了显示每位讲师所授课程的“课程”列。有关此 Razor 语法的详细信息，请参阅使用 `@:` 进行显式行转换。
- 添加了向所选讲师的 `tr` 元素中动态添加 `class="success"` 的代码。此时会使用 Bootstrap 类为所选行设置背景色。

```
string selectedRow = "";
if (item.ID == (int?)ViewData["InstructorID"])
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- 紧贴每行其他链接的前端添加了标有 Select 的新超链接，从而使所选讲师 ID 发送到 `Index` 方法。

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

运行应用并选择“讲师”选项卡。没有相关 OfficeAssignment 实体时，该页面显示相关 OfficeAssignment 实体的 Location 属性和空表格单元格。

在 Views/Instructors/Index.cshtml 文件中，关闭表格元素（在文件末尾）后，添加以下代码。选择讲师时，此代码显示与讲师相关的课程列表。

```
@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == (int?)ViewData["CourseID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

此代码读取视图模型的 `Courses` 属性以显示课程列表。它还提供 Select 超链接，该链接可将所选课程的 ID 发送到 `Index` 操作方法。

刷新页面并选择讲师。此时会出现一个网格，其中显示有分配给所选讲师的课程，且还显示有每个课程的分配系的名称。

在刚刚添加的代码块后，添加以下代码。选择课程后，代码将显示参与课程的学生列表。

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

此代码读取视图模型的 Enrollment 属性，从而显示参与课程的学生列表。

再次刷新该页并选择讲师。然后选择一门课程，查看参与的学生列表及其成绩。

显式加载

在 InstructorsController.cs 中检索讲师列表时，指定了预先加载 `CourseAssignments` 导航属性。

假设你希望用户在选中讲师和课程时尽量少查看注册情况。此时建议只在有请求时加载注册数据。若要查看如何执行显式加载的示例，请使用以下代码替换 `Index` 方法，这将删除预先加载 Enrollment 并显式加载该属性。代码所作更改为突出显示状态。

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}
```

新代码将从检索 Instructor 实体的代码中删除注册数据的 ThenInclude 方法调用。如果选择了讲师和课程，突出显示的代码会检索所选课程的 Enrollment 实体，及每个 Enrollment 的 Student 实体。

运行应用，立即转到“讲师”索引页，尽管已经更改了数据的检索方式，但该页上显示的内容没有任何不同。

总结

现在你已经使用了预先加载和一个查询及多个查询来读取导航属性中的相关数据。下一个教程将介绍如何更新相关数据。

上一页 下一页

ASP.NET Core MVC 和 EF Core - 更新相关数据 - 第 7 个教程(共 10 个)

2018/10/30 • • Edit Online

本教程没有升级至 ASP.NET Core 2.1。若要使用本教程的 ASP.NET Core 2.0 版，可以在目录表上方或者页面顶部选择“ASP.NET Core 2.0”：

相对于 2.0 版，本教程的 ASP.NET Core 2.1 Razor Pages 版有很多改进。

2.0 版教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。例如显著简化了基架代码。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果选择本教程的 Razor Pages 版，请在此 GitHub 讨论中说明原因。

作者：Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列，请参阅本系列中的第一个教程。

上一个教程显示出了相关数据，本教程将通过更新外键字段和导航属性来更新相关数据。

下图是一些将会用到的页面。

Edit - Co
localhost:58
Contoso University
Edit
Course
Number
1000
Title
Algebra 2
Credits
5
Department
Mathematics
Save

自定义课程的创建和编辑页面

创建新的课程实体时，新实体必须与现有院系有关系。为此，基架代码需包括控制器方法、创建视图和编辑视图，且视图中应包括用于选择院系的下拉列表。下拉列表设置了 `Course.DepartmentID` 外键属性，而这正是 Entity Framework 使用适当的 Department 实体加载 `Department` 导航属性所需要的。将用到基架代码，但需对其稍作更改，以便添加错误处理和对下拉列表进行排序。

在 CoursesController.cs 中，删除四种 Create 和 Edit 方法，并将其替换为以下代码：

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("CourseID,Credits,DepartmentID,Title")] Course course)
{
    if (ModelState.IsValid)
    {
        _context.Add(course);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var courseToUpdate = await _context.Courses
        .SingleOrDefaultAsync(c => c.CourseID == id);

    if (await TryUpdateModelAsync<Course>(courseToUpdate,
        "",
        c => c.Credits, c => c.DepartmentID, c => c.Title))
    {
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}
```

在 `Edit` HttpPost 方法之后，新建一个方法来为下拉列表加载院系信息。

```
private void PopulateDepartmentsDropDownList(object selectedDepartment = null)
{
    var departmentsQuery = from d in _context.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(), "DepartmentID", "Name",
selectedDepartment);
}
```

`PopulateDepartmentsDropDownList` 方法获取按名称排序的所有院系的列表，为下拉列表创建 `SelectList` 集合，并将该集合传递给 `ViewBag` 中的视图。该方法可以使用可选的 `selectedDepartment` 参数，而调用的代码可以通过该参数来指定呈现下拉列表时被选择的项。视图将 DepartmentID 名称传递给 `<select>` 标记帮助器，该帮助器就知道在 `ViewBag` 对象中查找名为 DepartmentID 的 `SelectList`。

HttpGet `Create` 方法调用 `PopulateDepartmentsDropDownList` 方法，但不会设置选定项，因为对于新课程而言，其院系尚未建立：

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

HttpGet `Edit` 方法根据正在编辑的课程已分配到的院系 ID 设置选定项：

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

`Create` 和 `Edit` 这二者的 HttpPost 方法还包括一段代码，用于在错误后重新显示页面时设置选定项。这样可以确保当页面重新显示出现错误消息时，选择的任何院系都将保持选中状态。

将 .AsNoTracking 添加到 Details 和 Delete 方法

为优化"课程详细信息"和"删除"页面的性能，请在 `Details` 和 HttpGet `Delete` 方法中添加 `AsNoTracking` 调用。

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

修改课程视图

在 Views/Courses/Create.cshtml 中，向"院系"下拉列表添加一个"选择院系"选项，将标题从 DepartmentID 更改为 Department，并添加一条验证消息。

```
<div class="form-group">
    <label asp-for="Department" class="control-label"></label>
    <select asp-for="DepartmentID" class="form-control" asp-items="ViewBag.DepartmentID">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="DepartmentID" class="text-danger" />
```

在 Views/Courses/Edit.cshtml 中，对"院系"字段进行与 Create.cshtml 中相同的更改。

另外，在 Views/Courses/Edit.cshtml 中，在"标题"字段之前添加一个课程编号字段。课程编号是主键，因此只会显示，无法更改。

```
<div class="form-group">
    <label asp-for="CourseID" class="control-label"></label>
    <div>@Html.DisplayFor(model => model.CourseID)</div>
</div>
```

"编辑"视图中已有一个隐藏的课程编号字段（`<input type="hidden">`。添加 `<label>` 标记帮助器后仍然需要该隐藏字段，因为添加标记帮助器后，用户在"编辑"页面上单击"保存"时，已发布数据中并不会包含课程编号。

在 Views/Courses/Delete.cshtml 中，在顶部添加一个课程编号字段，并将院系 ID 更改为院系名称。

```
@model ContosoUniversity.Models.Course

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
    </dl>

    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

在 Views/Courses/Details.cshtml 中，进行对 Delete.cshtml 所作相同的更改。

测试“课程”页

运行应用，选择“课程”选项卡，单击“新建”，然后输入新课程的数据：

单击“创建”。课程索引页面随即显示，并且新课程已添加在列表中。索引页列表中的院系名称来自导航属性，表明已正确建立关系。

在课程索引页中的课程上，单击“编辑”。

更改页面上的数据，然后单击“保存”。含有更新后的课程数据的“课程索引”页面随即显示。

添加讲师的编辑页面

编辑讲师记录时，有时希望能更新讲师的办公室分配。Instructor 实体和 OfficeAssignment 实体之间存在一对零或一的关系，这意味着代码必须处理一下情况：

- 如果用户清除了办公室分配，并且办公室分配最初具有一个值，则删除 OfficeAssignment 实体。
- 如果用户输入了办公室分配值，并且该值最初为空，则创建一个新的 OfficeAssignment 实体。
- 如果用户更改了办公室分配的值，则更改现有 OfficeAssignment 实体中的值。

更新讲师控制器

在 InstructorsController.cs 中，更改 HttpGet `Edit` 方法中的代码，使其加载 Instructor 实体的 `OfficeAssignment` 导航属性并调用 `AsNoTracking`：

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}
```

将 HttpPost `Edit` 方法更新为以下代码，以便处理办公室分配更新：

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .SingleOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instructorToUpdate);
}
```

该代码执行以下操作：

- 将方法名称更改为 `EditPost`，因为现在的签名与 HttpGet `Edit` 方法相同（`ActionName` 特性指定仍然使用 `/Edit/` URL）。
- 使用 `OfficeAssignment` 导航属性的预先加载从数据库获取当前的 Instructor 实体。此操作与在 HttpGet

`Edit` 方法中进行的操作相同。

- 将检索出的 Instructor 实体更新为模型绑定器中的值。通过 `TryUpdateModel` 重载可以将想包括的属性列入到允许列表。这样可以防止第二个教程中所述的过度发布。

```
if (await TryUpdateModelAsync<Instructor>(
    instructorToUpdate,
    "",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
```

- 如果办公室位置为空，请将 Instructor.OfficeAssignment 属性设置为 NULL，以便删除 OfficeAssignment 表中的相关行。

```
if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

- 将更改保存到数据库。

更新讲师编辑视图

在 Views/Instructors/Edit.cshtml 中，在“保存”按钮之前的末尾处，添加一个用于编辑办公室位置的新字段：

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>
```

运行应用，选择“讲师”选项卡，然后单击讲师页面上的“编辑”。更改“办公室位置”，然后单击“保存”。

向“讲师编辑”页添加课程分配

讲师可能教授任意数量的课程。现在可以通过使用一组复选框来更改课程分配，从而增强讲师编辑页面的性能，如以下屏幕截图所示：

Course 和 Instructor 实体之间是多对多的关系。若要添加和删除关系，可以向 CourseAssignments 联接实体集添加实体和从中删除实体。

用于更改讲师所对应的课程的 UI 是一组复选框。该复选框中会显示数据库中的所有课程，选中讲师当前对应的课程即可。用户可以通过选择或清除复选框来更改课程分配。如果课程的数量过大，建议使用其他方法在视图中呈现数据，但创建或删除关系的方法与操作联接实体的方法相同。

更新讲师控制器

若要为复选框列表的视图提供数据，将使用视图模型类。

在 SchoolViewModels 文件夹中创建 AssignedCourseData.cs，并将现有代码替换为以下代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

在 InstructorsController.cs 中，将 HttpGet `Edit` 方法替换为以下代码。突出显示所作更改。

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>(instructor.CourseAssignments.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}
```

该代码为 `Courses` 导航属性添加了预先加载，并调用新的 `PopulateAssignedCourseData` 方法使用 `AssignedCourseData` 视图模型类为复选框数组提供信息。

`PopulateAssignedCourseData` 方法中的代码会读取所有 Course 实体，以便使用视图模型类加载课程列表。对每门课程而言，该代码都会检查讲师的 `Courses` 导航属性中是否存在该课程。为高效检查某门课程是否被分配给了讲师，可将分配给该讲师的课程放置于 `HashSet` 集合中。对于讲师分配到的课程，`Assigned` 属性则设置为 true。视图将使用此属性来确定应将哪些复选框显示为选中状态。最后，该列表会被传递给 `ViewData` 中的视图。

接下来，添加用户单击“保存”时执行的代码。将 `EditPost` 方法替换为以下代码，并添加一个新方法，用于更新 Instructor 实体的 `Courses` 导航属性。

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
        .SingleOrDefaultAsync(m => m.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    UpdateInstructorCourses(selectedCourses, instructorToUpdate);
    PopulateAssignedCourseData(instructorToUpdate);
    return View(instructorToUpdate);
}
```

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

现在的方法签名与 HttpGet `Edit` 方法不同，因此方法名称将从 `EditPost` 变回 `Edit`。

视图没有 Course 实体的集合，因此模型绑定器无法自动更新 `CourseAssignments` 导航属性。可在新的 `UpdateInstructorCourses` 方法中更新 `CourseAssignments` 导航属性，而不必使用模型绑定器。为此，需要从模型绑定中排除 `CourseAssignments` 属性。此操作无需对调用 `TryUpdateModel` 的代码进行任何更改，因为使用的是允许列表重载，并且 `CourseAssignments` 不包括在该列表中。

如果未选中任何复选框，则 `UpdateInstructorCourses` 中的代码将使用空集合初始化 `CourseAssignments` 导航属性，并返回以下内容：

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

之后，代码会循环访问数据库中的所有课程，并逐一检查当前分配给讲师的课程和视图中处于选中状态的课程。为便于高效查找，后两个集合存储在 `HashSet` 对象中。

如果某课程的复选框处于选中状态，但该课程不在 `Instructor.CourseAssignments` 导航属性中，则会将该课程添加到导航属性中的集合中。

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

如果某课程的复选框未处于选中状态，但该课程存在 `Instructor.CourseAssignments` 导航属性中，则会从导航属性中删除该课程。

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

更新讲师视图

在 Views/Instructors/Edit.cshtml 中，通过在"办公室"字段的 `div` 元素之后和"保存"按钮的 `div` 元素之前添加以下代码，以便添加带有一系列复选框的"课程"字段。

> **NOTE**
>
> 将代码粘贴到 Visual Studio 中时，换行符会发生更改，从而导致代码中断。按 Ctrl+Z 一次可撤消自动格式设置。这样可以修复换行符，使其看起来如此处所示。缩进不一定要完美，但 `@</tr><tr>`、`@:<td>`、`@:</td>` 和 `@:</tr>` 行必须各成一行（如下所示），否则会出现运行时错误。选中新的代码块后，按 Tab 三次，使新代码与现有代码对齐。可在此处查看此问题的状态。

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                        @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

此代码将创建一个具有三列的 HTML 表。每个列中都有一个复选框，随后是一段由课程编号和标题组成的描述文字。所有复选框都具有相同的名称，即 selectedCourses，以告知模型绑定器将它们视为一组。每个复选框的值特性被设置为 `CourseID` 的值。发布页面时，模型绑定器会向控制器传递一个数组，其中只包括所选复选框的 `CourseID` 值。

这些复选框最开始呈现时，对于分配给讲师的课程的复选框，其特性处于选中状态。

运行应用，选择“讲师”选项卡，然后单击讲师页面上的“编辑”以查看“编辑”页面。

更改某些课程分配并单击“保存”。所作更改将反映在索引页上。

> **NOTE**
> 此处所使用的编辑讲师课程数据的方法适用于数量有限的课程。若是远大于此的集合，则需要使用不同的 UI 和不同的更新方法。

更新“删除”页

在 InstructorsController.cs 中，删除 `DeleteConfirmed` 方法，并在其位置插入以下代码。

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
        .SingleAsync(i => i.ID == id);

    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
    departments.ForEach(d => d.InstructorID = null);

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

此代码会更改以下内容：

- 对 `CourseAssignments` 导航属性执行预先加载。必须包括此内容，否则 EF 不知道相关的 `CourseAssignment` 实体，也不会删除它们。为避免在此处阅读它们，可以在数据库中配置级联删除。
- 如果要删除的讲师被指派为任何系的管理员，则需从这些系中删除该讲师分配。

向创建页添加办公室位置和课程

在 InstructorsController.cs 中，删除 HttpGet 和 HttpPost `Create` 方法，然后在其位置添加以下代码：

```
public IActionResult Create()
{
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
    PopulateAssignedCourseData(instructor);
    return View();
}

// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor
instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = new CourseAssignment { InstructorID = instructor.ID, CourseID =
int.Parse(course) };
            instructor.CourseAssignments.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}
```

此代码与 `Edit` 方法中所示内容类似，只是最开始未选择任何课程。HttpGet `Create` 方法调用 `PopulateAssignedCourseData` 方法不是因为可能有课程处于选中状态，而是为了在视图中为 `foreach` 循环提供空集合（否则该视图代码将引发空引用异常）。

检查是否存在验证错误并向数据库添加新讲师之前，HttpPost `Create` 方法会将每个选定课程添加到 `CourseAssignments` 导航属性。即使存在模型错误也会添加课程，因此出现模型错误（例如用户键入了无效的日期）并且页面重新显示并出现错误消息时，所作的任何课程选择都会自动还原。

请注意，为了能够向 `CourseAssignments` 导航属性添加课程，必须将该属性初始化为空集合：

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

除了在控制器代码中进行此操作之外，还可以在“导师”模型中进行此操作，方法是将该属性更改为不存在集合时自动创建集合，如以下示例所示：

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

如果通过这种方式修改 `CourseAssignments` 属性，则可以删除控制器中的显式属性初始化代码。

在 Views/Instructor/Create.cshtml 中，添加一个办公室位置文本框和课程的复选框，然后按“提交”按钮。与“编辑”页面中一样，如果粘贴代码时 Visual Studio 重新设置了其格式，则修复该格式。

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                            @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

通过运行应用并创建讲师来进行测试。

处理事务

如 CRUD 教程中所述，Entity Framework 隐式实现事务。如果需要更多控制操作（例如，如果想要在事务中包含在 Entity Framework 外部完成的操作），请参阅事务。

总结

处理相关数据的介绍至此已告一段落。下一个教程将介绍如何处理并发冲突。

上一页 下一页

ASP.NET Core MVC 和 EF Core - 并发 - 第 8 个教程（共 10 个）

2018/10/30 • • Edit Online

本教程没有升级至 ASP.NET Core 2.1。若要使用本教程的 ASP.NET Core 2.0 版，可以在目录表上方或者页面顶部选择“ASP.NET Core 2.0”：

相对于 2.0 版，本教程的 ASP.NET Core 2.1 Razor Pages 版有很多改进。

2.0 版教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。例如显著简化了基架代码。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果选择本教程的 Razor Pages 版，请在此 GitHub 讨论中说明原因。

作者：Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列，请参阅本系列中的第一个教程。

在之前的教程中，你学习了如何更新数据。本教程介绍如何处理多个用户同时更新同一实体时出现的冲突。

你将创建可处理 Department 实体的 Web 页面并处理并发错误。下图显示了“编辑”和“删除”页面，包括发生并发冲突时显示的一些消息。

Departmen
Edit - Cont
localhost:5813/Departr
Contoso University
Edit
Department
The record you attempted to edit was modified by another user after you got the original value. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again. Otherwise click the Back to List hyperlink.
Name
English
Budget
200000.00
Current value: $50,000.00
Start Date
9/1/2007
InstructorID
Abercrombie, Kim
Save

并发冲突

当某用户显示实体数据以对其进行编辑，而另一用户在上一用户的更改写入数据库之前更新同一实体的数据时，会发生并发冲突。如果不启用此类冲突的检测，则最后更新数据库的人员将覆盖其他用户的更改。在许多应用程序中，此风险是可接受的：如果用户很少或更新很少，或者一些更改被覆盖并不重要，则并发编程可能弊大于利。在此情况下，不必配置应用程序来处理并发冲突。

悲观并发（锁定）

如果应用程序确实需要防止并发情况下出现意外数据丢失，一种方法是使用数据库锁定。这称为悲观并发。例如，在从数据库读取一行内容之前，请求锁定为只读或更新访问。如果将一行锁定为更新访问，则其他用户无法将该行锁定为只读或更新访问，因为他们得到的是正在更改的数据的副本。如果将一行锁定为只读访问，则其他人也可将其锁定为只读访问，但不能进行更新。

管理锁定有缺点。编程可能很复杂。它需要大量的数据库管理资源，且随着应用程序用户数量的增加，可能会导致性能问题。由于这些原因，并不是所有的数据库管理系统都支持悲观并发。Entity Framework Core 未提供对它的内置支持，并且本教程不展示其实现方式。

开放式并发

悲观并发的替代方法是乐观并发。悲观并发是指允许发生并发冲突，并在并发冲突发生时作出正确反应。例如，Jane 访问“院系编辑”页面，并将英语系的预算从 350,000.00 美元更改为 0.00 美元。

在 Jane 单击“保存”之前，John 访问了相同页面，并将开始日期字段从 2007/1/9 更改为 2013/1/9。

Jane 先单击“保存”，并在浏览器返回索引页时看到她的更改。

然后 John 单击“编辑”页面上的“保存”，但页面上预算仍显示为 350,000.00 美元。接下来的情况取决于并发冲突的处理方式。

其中一些选项包括：

- 可以跟踪用户已修改的属性，并仅更新数据库中相应的列。

  在示例方案中，不会有数据丢失，因为是由两个用户更新不同的属性。下次有人浏览英语系时，将看到 Jane 和 John 两个人的更改 - 开始日期为 9/1/2013，预算为零美元。这种更新方法可减少可能导致数据丢失的冲突次数，但是如果对实体的同一属性进行竞争性更改，则数据难免会丢失。Entity Framework 是否以这种方式工作取决于更新代码的实现方式。通常不适合在 Web 应用程序中使用，因为它要求保持大量的状态，以便跟踪实体的所有原始属性值以及新值。维护大量的状态可能会影响应用程序的性能，因为它需要服务器资源或必须包含在网页本身（例如隐藏字段）或 Cookie 中。

- 可让 John 的更改覆盖 Jane 的更改。

  下次有人浏览英语系时，将看到 2013/1/9 和还原的值 350,000.00 美元。这称为“客户端优先”或“最后一个优先”。（客户端的所有值优先于数据存储的值。）正如本部分的介绍所述，如果不为并发处理编写任何代码，则自动执行此操作。

- 可以阻止在数据库中更新 John 的更改。

  通常，将显示一条错误消息，向他显示数据的当前状态，并允许他重新应用其更改（若他仍想更改）。这称为“存储优先”方案。（数据存储值优先于客户端提交的值。）本教程将执行“存储优先”方案。此方法可确保用户在未收到具体发生内容的警报时，不会覆盖任何更改。

检测并发冲突

你可以通过处理 Entity Framework 引发的 `DbConcurrencyException` 异常来解决冲突。为了知道何时引发这些异常，Entity Framework 必须能够检测到冲突。因此，你必须正确配置数据库和数据模型。启用冲突检测的某些选项包括：

- 数据库表中包含一个可用于确定某行更改时间的跟踪列。然后可配置 Entity Framework，将该列包含在 SQL Update 或 Delete 命令的 Where 子句中。

  跟踪列的数据类型通常是 `rowversion`。`rowversion` 值是一个序列号，该编号随着每次行的更新递增。在 Update 或 Delete 命令中，Where 子句包含跟踪列的原始值（原始行版本）。如果正在更新的行已被其他用户更改，则 `rowversion` 列中的值与原始值不同，这导致 Update 或 Delete 语句由于 Where 子句而找不到要更新的行。当 Entity Framework 发现 Update 或 Delete 命令没有更新行（即受影响的行数为零）时，便将其解释为并发冲突。

- 配置 Entity Framework，在 Update 或 Delete 命令的 Where 子句中包含表中每个列的原始值。

与第一个选项一样，如果行自第一次读取后发生更改，Where 子句将不返回要更新的行，Entity Framework 会将其解释为并发冲突。对于包含许多列的数据库表，此方法可能导致非常多的 Where 子句，并且可能需要维持大量的状态。如前所述，维持大量的状态会影响应用程序的性能。因此通常不建议使用此方法，并且它也不是本教程中使用的方法。

如果确实想要实现此并发方法，必须通过向要跟踪并发的实体添加 `ConcurrencyCheck` 属性来标记它们的所有非主键属性。所作更改使 Entity Framework 能够将所有列包含在 Update 和 Delete 语句的 SQL Where 子句中。

在本教程的其余部分，将向 Department 实体添加 `rowversion` 跟踪属性，创建控制器和视图，并进行测试以验证是否一切正常工作。

向 Department 实体添加跟踪属性

在 Models/Department.cs 中，添加名为 RowVersion 的跟踪属性：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

`Timestamp` 属性指定此列将包含在发送到数据库的 Update 和 Delete 命令的 Where 子句中。该属性称为 `Timestamp`，因为 SQL Server 的之前版本在 SQL `rowversion` 类型将其替换之前使用 SQL `timestamp` 数据类型。用于 `rowversion` 的 .NET 类型为字节数组。

如果更愿意使用 Fluent API，可使用 `IsConcurrencyToken` 方法（路径为 Data/SchoolContext.cs）指定跟踪属性，如下例所示：

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsConcurrencyToken();
```

通过添加属性，更改了数据库模型，因此需要再执行一次迁移。

保存更改并生成项目，然后在命令窗口中输入以下命令：

```
dotnet ef migrations add RowVersion
```

```
dotnet ef database update
```

创建“院系”控制器和视图

像之前在学生、课程和讲师教程中操作的那样，为“院系”控制器和视图创建基架。

在 DepartmentsController.cs 文件中，将出现的 4 次“FirstMidName”更改为“FullName”，以便院系管理员下拉列表将包含讲师的全名，而不仅仅是姓氏。

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName", department.InstructorID);
```

更新“院系索引”视图

基架引擎在索引视图中创建 RowVersion 列，但不应显示该字段。

将 Views/Departments/Index.cshtml 中的代码替换为以下代码。

```
@model IEnumerable<ContosoUniversity.Models.Department>

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Administrator.FullName)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.DepartmentID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

这会将标题更改为“院系”，删除 RowVersion 列，并显示全名（而非管理员的名字）。

更新“院系”控制器中的编辑方法

在 HttpGet `Edit` 方法和 `Details` 方法中，添加 `AsNoTracking`。在 HttpGet `Edit` 方法中，为管理员添加预先加载。

```
var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .SingleOrDefaultAsync(m => m.DepartmentID == id);
```

将 HttpPost `Edit` 方法的现有代码替换为以下代码：

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, byte[] rowVersion)
{
    if (id == null)
    {
        return NotFound();
    }

    var departmentToUpdate = await _context.Departments.Include(i => i.Administrator).SingleOrDefaultAsync(m
=> m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        Department deletedDepartment = new Department();
        await TryUpdateModelAsync(deletedDepartment);
        ModelState.AddModelError(string.Empty,
            "Unable to save changes. The department was deleted by another user.");
        ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
deletedDepartment.InstructorID);
        return View(deletedDepartment);
    }

    _context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;

    if (await TryUpdateModelAsync<Department>(
        departmentToUpdate,
        "",
        s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Department)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty,
                    "Unable to save changes. The department was deleted by another user.");
            }
            else
            {
                var databaseValues = (Department)databaseEntry.ToObject();

                if (databaseValues.Name != clientValues.Name)
                {
                    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
                }
                if (databaseValues.Budget != clientValues.Budget)
                {
                    ModelState.AddModelError("Budget", $"Current value: {databaseValues.Budget:c}");
                }
                if (databaseValues.StartDate != clientValues.StartDate)
                {
```

```
                    ModelState.AddModelError("StartDate", $"Current value: {databaseValues.StartDate:d}");
                }
                if (databaseValues.InstructorID != clientValues.InstructorID)
                {
                    Instructor databaseInstructor = await _context.Instructors.SingleOrDefaultAsync(i => i.ID
== databaseValues.InstructorID);
                    ModelState.AddModelError("InstructorID", $"Current value:
{databaseInstructor?.FullName}");
                }

                ModelState.AddModelError(string.Empty, "The record you attempted to edit "
                        + "was modified by another user after you got the original value. The "
                        + "edit operation was canceled and the current values in the database "
                        + "have been displayed. If you still want to edit this record, click "
                        + "the Save button again. Otherwise click the Back to List hyperlink.");
                departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
                ModelState.Remove("RowVersion");
            }
        }
    }
    ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
departmentToUpdate.InstructorID);
    return View(departmentToUpdate);
}
```

代码先尝试读取要更新的院系。如果 `SingleOrDefaultAsync` 方法返回 NULL，则该院系已被另一用户删除。此情况下，代码将使用已发布的表单值创建院系实体，以便“编辑”页重新显示错误消息。或者，如果仅显示错误消息而未重新显示院系字段，则不必重新创建 Department 实体。

该视图将原始 `RowVersion` 值存储在隐藏字段中，且此方法在 `rowVersion` 参数中接收该值。在调用 `SaveChanges` 之前，必须将该原始 `RowVersion` 属性值置于实体的 `OriginalValues` 集合中。

```
_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;
```

然后，当 Entity Framework 创建 SQL UPDATE 命令时，该命令将包含一个 WHERE 子句，用于查找具有原始 `RowVersion` 值的行。如果没有行受到 UPDATE 命令影响（没有行具有原始 `RowVersion` 值），则 Entity Framework 会引发 `DbUpdateConcurrencyException` 异常。

该异常的 catch 块中的代码获取受影响的 Department 实体，该实体具有来自异常对象上的 `Entries` 属性的更新值。

```
var exceptionEntry = ex.Entries.Single();
```

`Entries` 集合将仅包含一个 `EntityEntry` 对象。该对象可用于获取用户输入的新值和当前的数据库值。

```
var clientValues = (Department)exceptionEntry.Entity;
var databaseEntry = exceptionEntry.GetDatabaseValues();
```

对于所含的数据库值与用户在“编辑”页上输入的值不同的每一列，该代码均为其添加一个自定义错误消息（为简洁起见，此处仅显示一个字段）。

```
var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)
{
    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
```

最后，该代码将 `departmentToUpdate` 的 `RowVersion` 值设置为从数据库中检索到的新值。重新显示“编辑”页时，这个新的 `RowVersion` 值将存储在隐藏字段中，当用户下次单击“保存”时，将只捕获自“编辑”页重新显示起发生的并发错误。

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
ModelState.Remove("RowVersion");
```

`ModelState` 具有旧的 `RowVersion` 值，因此需使用 `ModelState.Remove` 语句。在此视图中，当两者都存在时，字段的 `ModelState` 值优于模型属性值。

更新“院系编辑”视图

在 Views/Departments/Edit.cshtml 中，进行以下更改：

- 添加隐藏字段以保存 `RowVersion` 属性值，紧跟在 `DepartmentID` 属性的隐藏字段后面。
- 向下拉列表添加“选择管理员”选项。

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="DepartmentID" />
            <input type="hidden" asp-for="RowVersion" />
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
                <span asp-validation-for="InstructorID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

测试“编辑”页中的并发冲突

运行应用并转到“院系索引”页。右键单击英语系的“编辑”超链接，并选择“在新选项卡中打开”，然后单击英语系的“编辑”超链接。现在，两个浏览器选项卡显示相同的信息。

在第一个浏览器选项卡中更改一个字段，然后单击“保存”。

浏览器显示具有更改值的索引页。

在第二个浏览器选项卡中更改一个字段。

单击"保存"。看见一条错误消息:

再次单击"保存"。保存在第二个浏览器选项卡中输入的值。在索引页中出现时，可以看到已保存的值。

更新"删除"页

对于"删除"页，Entity Framework 以类似方式检测其他人编辑院系所引起的并发冲突。当 HttpGet `Delete` 方法显示确认视图时，该视图包含隐藏字段中的原始 `RowVersion` 值。然后，该值可用于在用户确认删除时进行调用的 HttpPost `Delete` 方法。当 Entity Framework 创建 SQL DELETE 命令时，它包含具有原始 `RowVersion` 值的 WHERE 子句。如果该命令不影响任何行(即在显示"删除"确认页面后更改了行)，则引发并发异常，调用 HttpGet `Delete` 方法，并将错误标志设置为 true，以重新显示具有一条错误消息的确认页面。任意行均未受影响的原因也可能是，该行已被其他用户删除，因此在此情况下不显示错误消息。

更新"院系"控制器中的删除方法

在 DepartmentsController.cs 中，将 HttpGet `Delete` 方法替换为以下代码：

```
public async Task<IActionResult> Delete(int? id, bool? concurrencyError)
{
    if (id == null)
    {
        return NotFound();
    }

    var department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.DepartmentID == id);
    if (department == null)
    {
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction(nameof(Index));
        }
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ViewData["ConcurrencyErrorMessage"] = "The record you attempted to delete "
            + "was modified by another user after you got the original values. "
            + "The delete operation was canceled and the current values in the "
            + "database have been displayed. If you still want to delete this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }

    return View(department);
}
```

该方法接受可选参数，该参数指示是否在并发错误之后重新显示页面。如果此标志为 true 且指定的院系不复存在，则它已被其他用户删除。此情况下，代码将重定向到索引页。如果此标志为 true 且该院系确实存在，则它已被其他用户更改。此情况下，代码将使用 `ViewData` 向视图发送一条错误消息。

将 HttpPost `Delete` 方法（名为 `DeleteConfirmed`）中的代码替换为以下代码：

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(Department department)
{
    try
    {
        if (await _context.Departments.AnyAsync(m => m.DepartmentID == department.DepartmentID))
        {
            _context.Departments.Remove(department);
            await _context.SaveChangesAsync();
        }
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateConcurrencyException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { concurrencyError = true, id = department.DepartmentID
});
    }
}
```

在刚替换的基架代码中，此方法仅接受记录 ID：

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

已将此参数更改为由模型绑定器创建的 Department 实体实例。这样，EF 就可访问除记录键之外的 RowVersion 属性值。

```
public async Task<IActionResult> Delete(Department department)
```

你还将操作方法名称从 `DeleteConfirmed` 更改为了 `Delete` 。基架代码使用 `DeleteConfirmed` 名称来为 HttpPost 方法提供唯一签名。（CLR 要求重载方法具有不同的方法参数。）既然签名是唯一的，就可继续使用 MVC 约定，并为 HttpPost 和 HttpGet 删除方法使用同一名称。

如果已删除院系，则 `AnyAsync` 方法返回 false，应用程序仅返回到 Index 方法。

如果捕获到并发错误，代码将重新显示“删除”确认页，并提供一个指示它应显示并发错误消息的标志。

更新“删除”视图

在 Views/Departments/Delete.cshtml 中，将基架代码替换为添加 DepartmentID 和 RowVersion 属性的错误消息字段和隐藏字段的以下代码。突出显示所作更改。

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>

    <form asp-action="Delete">
        <input type="hidden" asp-for="DepartmentID" />
        <input type="hidden" asp-for="RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

这将进行以下更改：

- 在 `h2` 和 `h3` 标题之间添加错误消息。
- 将“管理员”字段中的 FirstMidName 替换为 FullName。
- 删除 RowVersion 字段。
- 添加 `RowVersion` 属性的隐藏字段。

运行应用并转到“院系索引”页。右键单击英语系的“删除”超链接，并选择“在新选项卡中打开”，然后在第一个选项卡中单击英语系的“编辑”超链接。

在第一个窗口中，更改其中一个值，然后单击“保存”：

在第二个选项卡中，单击“删除”。你将看到并发错误消息，且已使用数据库中的当前内容刷新了“院系”值。

如果再次单击“删除”，会重定向到已删除显示院系的索引页。

更新“详细信息”和“创建”视图

可选择性地清理“详细信息”和“创建”视图中的基架代码。

替换 Views/Departments/Details.cshtml 中的代码，以删除 RowVersion 列并显示管理员的全名。

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

替换 Views/Departments/Create.cshtml 中的代码，向下拉列表添加“选择”选项。

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

总结

处理并发冲突已介绍完毕。要深入了解如何处理 EF Core 中的并发，请参阅并发冲突。下一个教程将介绍如何为 Instructor 和 Students 实体实现“每个层次结构一个表”继承。

上一页 下一页

ASP.NET Core MVC 和 EF Core - 继承 - 第 9 个教程(共 10 个)

2018/10/30 • • Edit Online

本教程没有升级至 ASP.NET Core 2.1。若要使用本教程的 ASP.NET Core 2.0 版，可以在目录表上方或者页面顶部选择“ASP.NET Core 2.0”：

相对于 2.0 版，本教程的 ASP.NET Core 2.1 Razor Pages 版有很多改进。

2.0 版教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。例如显著简化了基架代码。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果选择本教程的 Razor Pages 版，请在此 GitHub 讨论中说明原因。

作者：Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列，请参阅本系列中的第一个教程。

在上一个教程中，已经处理了并发异常。本教程将演示如何在数据模型中实现继承。

在面向对象的编程中，可以使用继承以便于重用代码。在本教程中，将更改 `Instructor` 和 `Student` 类，以便从 `Person` 基类中派生，该基类包含教师和学生所共有的属性（如 `LastName`）。不会添加或更改任何网页，但会更改部分代码，并将在数据库中自动反映这些更改。

将继承映射到数据库表的选项

学校数据模型中的 `Instructor` 和 `Student` 类具有多个相同的属性：

假设想要消除由 `Instructor` 和 `Student` 实体共享的属性的冗余代码。或者想要写入可以格式化姓名的服务，而无需关注该姓名来自教师还是学生。可以创建只包含这些共享属性的 `Person` 基类，然后使 `Instructor` 和 `Student` 类继承该基类，如下图所示：

有多种方法可以在数据库中表示此继承结构。可以创建一个 Person 表，将学生和教师的相关信息包含在一个表中。某些列可能仅适用于教师 (HireDate)，某些列仅适用于学生 (EnrollmentDate)，某些列同时适用于两者 (LastName、FirstName)。通常情况下，将有一个鉴别器列来指示每行所代表的类型。例如，鉴别器列可能包含"Instructor"来指示教师，包含"Student"来指示学生。

从单个数据库表生成实体继承结构的模式称为每个层次结构一张 (TPH) 继承。

另一种方法是使数据库看起来更像继承结构。例如，可以仅将姓名字段包含到 Person 表中，在单独的 Instructor 和 Student 表中包含日期字段。

为每个实体类创建数据库表的模式称为每个类型一张表 (TPT) 继承。

另一种方法是将所有非抽象类型映射到单独的表。类的所有属性(包括继承的属性)映射到相应表的列。此模式称为每个具体类一张表 (TPC) 继承。如果为前面所示的 Person、Student 和 Instructor 类实现了 TPC 继承，那么在实现继承之后，Student 和 Instructor 表看起来将与以前没什么不同。

TPC 和 TPH 继承模式的性能通常比 TPT 继承模式好，因为 TPT 模式会导致复杂的联接查询。

本教程将演示如何实现 TPH 继承。TPH 是 Entity Framework Core 唯一支持的继承模式。需要执行的操作是创建 `Person` 类、将 `Instructor` 和 `Student` 类更改为从 `Person` 派生、将新的类添加到 `DbContext`，以及创建迁移。

> **TIP**
>
> 在进行以下更改之前，请考虑保存项目的副本。如果遇到问题并需要重新开始，可以更轻松地从已保存的项目开始，而不用反向操作本教程中的步骤或者返回到整个系列的开始。

创建 Person 类

在 Models 文件夹中，创建 Person.cs 并使用以下代码替换模板代码：

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }
    }
}
```

使 Student 和 Instructor 类从 Person 继承

在 Instructor.cs 中，从 Person 类派生 Instructor 类并删除键和姓名字段。代码将如下所示：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

在 Student.cs 中做出相同更改。

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }


        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

将 Person 实体类型添加到数据模型

将 Person 实体类型添加到 SchoolContext.cs。新的行突出显示。

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }
        public DbSet<Person> People { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");
            modelBuilder.Entity<Person>().ToTable("Person");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

以上是 Entity Framework 配置每个层次结构一张表继承所需的全部操作。正如将看到的，更新数据库时，将有一个 Person 表来代替 Student 和 Instructor 表。

创建和自定义迁移代码

保存更改并生成项目。随后在项目文件夹中打开命令窗口并输入以下命令：

```
dotnet ef migrations add Inheritance
```

暂不运行 `database update` 命令。该命令将导致数据丢失，因为它将删除 Instructor 表并将 Student 表重命名为 Person。需要提供自定义代码来保留现有数据。

打开 Migrations/<timestamp>_Inheritance.cs 并使用以下代码替换 `Up` 方法：

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropForeignKey(
        name: "FK_Enrollment_Student_StudentID",
        table: "Enrollment");

    migrationBuilder.DropIndex(name: "IX_Enrollment_StudentID", table: "Enrollment");

    migrationBuilder.RenameTable(name: "Instructor", newName: "Person");
    migrationBuilder.AddColumn<DateTime>(name: "EnrollmentDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<string>(name: "Discriminator", table: "Person", nullable: false, maxLength:
128, defaultValue: "Instructor");
    migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<int>(name: "OldId", table: "Person", nullable: true);

    // Copy existing Student data into new Person table.
    migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate, EnrollmentDate,
Discriminator, OldId) SELECT LastName, FirstName, null AS HireDate, EnrollmentDate, 'Student' AS
Discriminator, ID AS OldId FROM dbo.Student");
    // Fix up existing relationships to match new PK's.
    migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID FROM dbo.Person WHERE OldId =
Enrollment.StudentId AND Discriminator = 'Student')");

    // Remove temporary key
    migrationBuilder.DropColumn(name: "OldID", table: "Person");

    migrationBuilder.DropTable(
        name: "Student");

    migrationBuilder.CreateIndex(
         name: "IX_Enrollment_StudentID",
         table: "Enrollment",
         column: "StudentID");

    migrationBuilder.AddForeignKey(
        name: "FK_Enrollment_Person_StudentID",
        table: "Enrollment",
        column: "StudentID",
        principalTable: "Person",
        principalColumn: "ID",
        onDelete: ReferentialAction.Cascade);
}
```

此代码负责以下数据库更新任务：

- 删除指向 Student 表的外键约束和索引。
- 将 Instructor 表重命名为 Person，根据需要做出更改以存储学生数据：
- 为学生添加可为 NULL 的 EnrollmentDate。
- 添加鉴别器列来指示行代表学生还是教师。
- HireDate 可为 NULL，因为学生行不会包含聘用日期。
- 添加临时字段，用于更新指向学生的外键。将学生复制到 Person 表时，将获取新的主键值。
- 将数据从 Student 表复制到 Person 表。这将使学生获取分配的新主键值。
- 修复指向学生的外键值。
- 重新创建外键约束和索引，现在将它们指向 Person 表。

（如果已使用 GUID 而不是整数作为主键类型，那么将不需要更改学生主键值，并且可能已省略其中多个步骤。）

运行 `database update` 命令：

```
dotnet ef database update
```

（在生产系统中，可以对 `Down` 方法进行相应更改，以防必须使用该方法返回到以前的数据库版本。本教程中将不使用 `Down` 方法。）

> **NOTE**
>
> 在包含现有数据的数据库中更改架构时，可能会发生其他错误。如果出现无法解决的迁移错误，可以在连接字符串中更改数据库名或者删除数据库。若是新数据库，则没有要迁移的数据，因此在完成更新数据库命令时很可能不会出错。若要删除数据库，请使用 SSOX 或运行 `database drop` CLI 命令。

使用已实现的继承进行测试

运行应用并尝试各种页面。一切都和以前一样。

在“SQL Server 对象资源管理器” 中，展开“数据连接/SchoolContext”和“表”，将看到 Student 和 Instructor 表已替换为 Person 表。打开 Person 表设计器，将看到它包含在 Student 和 Instructor 表中使用的所有列。

右键单击 Person 表，然后单击“显示表数据”以查看鉴别器列。

ID	FirstName	HireDate	LastName	Enrollme...	Discriminator
1	Kim	3/11/1995...	Abercrombie	NULL	Instructor
2	Fadi	7/6/2002 ...	Fakhouri	NULL	Instructor
3	Roger	7/1/1998 ...	Harui	NULL	Instructor
4	Candace	1/15/2001...	Kapoor	NULL	Instructor
5	Roger	2/12/2004...	Zheng	NULL	Instructor
7	Nancy	8/17/2016...	Davolio	NULL	Instructor
8	Carson	NULL	Alexander	9/1/2010 ...	Student
9	Meredith	NULL	Alonso	9/1/2012 ...	Student
10	Arturo	NULL	Anand	9/1/2013 ...	Student
11	Gytis	NULL	Barzdukas	9/1/2012 ...	Student
12	Yan	NULL	Li	9/1/2012 ...	Student

总结

你已经为 `Person`、`Student` 和 `Instructor` 类实现了每个层次结构一张表继承。若要详细了解 Entity Framework Core 中的继承，请参阅继承。下一个教程将介绍如何处理各种相对高级的 Entity Framework 方案。

上一页 下一页

使用 EF Core 创建 ASP.NET Core MVC - 高级 - 第 10 个教程(共 10 个)

2018/9/24 • • Edit Online

本教程没有升级至 ASP.NET Core 2.1。若要使用本教程的 ASP.NET Core 2.0 版，可以在目录表上方或者页面顶部选择“ASP.NET Core 2.0”：

相对于 2.0 版，本教程的 ASP.NET Core 2.1 Razor Pages 版有很多改进。

2.0 版教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor Pages 是基于页面的编程模型，它让 Web UI 的生成更加简单高效。建议使用 MVC 版本的 Razor 页面教程。Razor 页面教程：

- 易于关注。例如显著简化了基架代码。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

如果选择本教程的 Razor Pages 版，请在此 GitHub 讨论中说明原因。

作者：Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解系列教程，请参阅本系列中的第一个教程。

之前的教程中，已经以每个类一张表的方式实现了继承。本教程将会介绍在掌握开发基础 ASP.NET Core web 应用程序之后使用 Entity Framework Core 开发时需要注意的几个问题。

原生 SQL 查询

使用 Entity Framework 的优点之一是它可避免你编写跟数据库过于耦合的代码 它会自动生成 SQL 查询和命令，使得你无需自行编写。但有一些特殊情况，你需要执行手动创建的特定 SQL 查询。对于这些情况下，Entity Framework Code First API 包括直接传递 SQL 命令将到数据库的方法。在 EF Core 1.0 中具有以下选项：

- 使用 `DbSet.FromSql` 返回实体类型的查询方法。返回的对象必须是 `DbSet` 对象期望的类型，并且它们会自动跟踪数据库上下文中除非你手动关闭跟踪。
- 对于非查询命令使用 `Database.ExecuteSqlCommand` 。

如果需要运行该返回类型不是实体的查询，你可以使用由 EF 提供的 ADO.NET 中使用数据库连接。数据库上下文不会跟踪返回的数据，即使你使用该方法来检索实体类型也是如此。

在 Web 应用程序中执行 SQL 命令时，请务必采取预防措施来保护站点免受 SQL 注入攻击。一种方法是使用参数化查询，确保不会将网页提交的字符串视为 SQL 命令。在本教程中，将用户输入集成到查询中时会使用参数化查询。

调用返回实体的查询

`DbSet<TEntity>` 类提供了可用于执行查询并返回 `TEntity` 类型实体的方法。若要查看实现细节，你需要更改部门控制器中 `Details` 方法的代码。

在*DepartmentsController.cs*中的`Details`方法，通过代码调用`FromSql`方法检索一个部门，如以下高亮代码所示：

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    string query = "SELECT * FROM Department WHERE DepartmentID = {0}";
    var department = await _context.Departments
        .FromSql(query, id)
        .Include(d => d.Administrator)
        .AsNoTracking()
        .SingleOrDefaultAsync();

    if (department == null)
    {
        return NotFound();
    }

    return View(department);
}
```

为了验证新代码是否工作正常，请选择**Department**选项卡，然后点击某个部门的**Detail**。

调用返回其他类型的查询

之前你在"关于"页面创建了一个学生统计信息网格，显示每个注册日期的学生数量。可以从学生实体集中获取数据（`_context.Students`），使用 LINQ 将结果投影到`EnrollmentDateGroup`视图模型对象的列表。假设你想要 SQL 本身编写，而不使用 LINQ。需要运行 SQL 查询中返回实体对象之外的内容。在 EF Core 1.0 中，执行该操作的另一种方法是编写 ADO.NET 代码，并从 EF 获取数据库连接。

在*HomeController.cs*中，将`About`方法替换为以下代码：

```
public async Task<ActionResult> About()
{
    List<EnrollmentDateGroup> groups = new List<EnrollmentDateGroup>();
    var conn = _context.Database.GetDbConnection();
    try
    {
        await conn.OpenAsync();
        using (var command = conn.CreateCommand())
        {
            string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount "
                + "FROM Person "
                + "WHERE Discriminator = 'Student' "
                + "GROUP BY EnrollmentDate";
            command.CommandText = query;
            DbDataReader reader = await command.ExecuteReaderAsync();

            if (reader.HasRows)
            {
                while (await reader.ReadAsync())
                {
                    var row = new EnrollmentDateGroup { EnrollmentDate = reader.GetDateTime(0), StudentCount
= reader.GetInt32(1) };
                    groups.Add(row);
                }
            }
            reader.Dispose();
        }
    }
    finally
    {
        conn.Close();
    }
    return View(groups);
}
```

添加 using 语句：

```
using System.Data.Common;
```

运行应用并转到“关于”页面。显示的数据和之前一样。

调用更新查询

假设 Contoso 大学管理员想要在数据库中执行全局更改，例如如更改的每个课程的可修读人数。如果该大学有大量的课程，检索所有实体并单独更改会降低效率。在本节中，你将实现一个页面，使用户能够指定一个参数，通过这个参数可以更改所有课程的可修读人数，在这里你会通过执行 SQL UPDATE 语句来进行更改。页面外观类似于下图：

在 CoursesContoller.cs 中，为 HttpGet 和 HttpPost 添加 UpdateCourseCredits 方法：

```
public IActionResult UpdateCourseCredits()
{
    return View();
}
```

```
[HttpPost]
public async Task<IActionResult> UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewData["RowsAffected"] =
            await _context.Database.ExecuteSqlCommandAsync(
                "UPDATE Course SET Credits = Credits * {0}",
                parameters: multiplier);
    }
    return View();
}
```

当控制器处理 HttpGet 请求时，`ViewData["RowsAffected"]`中不会返回任何东西，并且在视图中显示一个空文本框和提交按钮，如上图所示。

当单击**Update**按钮时，将调用 HttpPost 方法，且从文本框中输入的值获取乘数。代码接着执行 SQL 语句更新课程，并向视图的`ViewData`返回受影响的行数。当视图获取`RowsAffected`值，它将显示更新的行数。

在"解决方案资源管理器"中，右键单击"Views/Courses"文件夹，然后依次单击"添加"和"新建项"。

在**添加新**项对话框中，在**已安装**下单击**ASP.NET**，在左窗格中，单击**MVC** 视图页，并将新视图命名为*UpdateCourseCredits.cshtml*。

在*Views/Courses/UpdateCourseCredits.cshtml*中，将模板代码替换为以下代码：

```
@{
    ViewBag.Title = "UpdateCourseCredits";
}

<h2>Update Course Credits</h2>

@if (ViewData["RowsAffected"] == null)
{
    <form asp-action="UpdateCourseCredits">
        <div class="form-actions no-color">
            <p>
                Enter a number to multiply every course's credits by: @Html.TextBox("multiplier")
            </p>
            <p>
                <input type="submit" value="Update" class="btn btn-default" />
            </p>
        </div>
    </form>
}
@if (ViewData["RowsAffected"] != null)
{
    <p>
        Number of rows updated: @ViewData["RowsAffected"]
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

通过选择**Courses**选项卡运行`UpdateCourseCredits`方法，然后在浏览器地址栏中 URL 的末尾添加"/UpdateCourseCredits"到（例如：`http://localhost:5813/Courses/UpdateCourseCredits`）。在文本框中输入数字：

单击**Update**。你会看到受影响的行数：

单击**Back To List**可以看到课程列表，其中可修读人数已经替换成修改后的数字。

请注意生产代码将确保更新最终得到有效的数据。此处所示的简化代码会使得相乘后可修读人数大于 5。(`Credits` 属性具有 `[Range(0, 5)]` 特性。)更新查询将起作用，但无效的数据会导致意外的结果，例如在系统的其他部分中加入可修读人数为 5 或更少可能会导致意外的结果。

有关原生 SQL 查询的详细信息，请参阅原生 SQL 查询。

检查发送到数据库的 SQL

有时能够以查看发送到数据库的实际 SQL 查询对于开发者来说是很有用的。EF Core 自动使用 ASP.NET Core 的内置日志记录功能来编写包含 SQL 查询和更新的日志。在本部分中，你将看到记录 SQL 日志的一些示例。

打开*StudentsController.cs*并在 `Details` 方法的 `if (student == null)` 语句上设置断点。

在调试模式下运行应用，并转到某位学生的"详细信息"页面。

转到**输出窗口**显示调试输出，就可以看到查询语句：

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (56ms) [Parameters=
[@__id_0='?'], CommandType='Text', CommandTimeout='30']
SELECT TOP(2) [s].[ID], [s].[Discriminator], [s].[FirstName], [s].[LastName], [s].[EnrollmentDate]
FROM [Person] AS [s]
WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @__id_0)
ORDER BY [s].[ID]
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (122ms) [Parameters=
[@__id_0='?'], CommandType='Text', CommandTimeout='30']
SELECT [s.Enrollments].[EnrollmentID], [s.Enrollments].[CourseID], [s.Enrollments].[Grade], [s.Enrollments].
[StudentID], [e.Course].[CourseID], [e.Course].[Credits], [e.Course].[DepartmentID], [e.Course].[Title]
FROM [Enrollment] AS [s.Enrollments]
INNER JOIN [Course] AS [e.Course] ON [s.Enrollments].[CourseID] = [e.Course].[CourseID]
INNER JOIN (
    SELECT TOP(1) [s0].[ID]
    FROM [Person] AS [s0]
    WHERE ([s0].[Discriminator] = N'Student') AND ([s0].[ID] = @__id_0)
    ORDER BY [s0].[ID]
) AS [t] ON [s.Enrollments].[StudentID] = [t].[ID]
ORDER BY [t].[ID]
```

你会注意到一些可能会让你觉得惊讶的操作：SQL 从 Person 表最多选择 2 行（`TOP(2)`）。`SingleOrDefaultAsync` 方法服务器上不会解析为 1 行。原因是：

- 如果查询返回多个行，该方法会返回 null。
- 如果想知道查询是否返回多个行，EF 必须检查是否至少返回 2。

请注意，你不必使用调试模式，并在断点处停止，然后在输出窗口获取日志记录。这种方法非常便捷，只需在想查看输出时停止日志记录即可。如果不进行此操作，程序将继续进行日志记录，要查看感兴趣的部分则必须向后滚动。

存储库和工作单元模式

许多开发人员编写代码实现存储库和工作模式单元以作为使用 Entity Framework 代码的包装器。这些模式用于在应用程序的数据访问层和业务逻辑层之间创建抽象层。实现这些模式可让你的应用程序对数据存储介质的更改不敏感，而且很容易进行自动化单元测试和进行测试驱动开发 (TDD)。但是，编写附加代码以实现这些模式对于使用 EF 的应用程序并不总是最好的选择，原因有以下几个：

- EF 上下文类可以为使用 EF 的数据库更新充当工作单位类。
- 对于使用 EF 进行的数据库更新，EF 上下文类可充当工作单元类。
- EF 包括用于无需编写存储库代码就实现 TDD 的功能。

有关如何实现存储库和工作单元模式的详细信息，请参阅本系列教程的 Entity Framework 5 版本。

Entity Framework Core 的实现可用于测试的内存数据库驱动程序。有关详细信息，请参阅测试以及 InMemory。

自动脏值检测

Entity Framework 通过比较的实体的当前值与原始值来判断更改实体的方式（因此需要发送更新到数据库）。查询或附加该实体时会存储的原始值。如下方法会导致自动脏值：

- DbContext.SaveChanges
- DbContext.Entry
- ChangeTracker.Entries

如果正在跟踪大量实体，并且这些方法之一在循环中多次调用，通过使用 `ChangeTracker.AutoDetectChangesEnabled` 属性暂时关闭自动脏值检测，能够显著改进性能。例如：

```
_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

Entity Framework Core 源代码与开发计划

Entity Framework Core 源位于 https://github.com/aspnet/EntityFrameworkCore。仓库中除了有源代码，还可包括每夜生成、问题跟踪、功能规范、设计会议备忘录和将来的开发路线图。你可以归档或查找 bug 并进行更改。

尽管源代码处于开源状态，Entity Framework Core 是由 Microsoft 完全支持的产品。Microsoft Entity Framework 团队将控制接受哪些贡献和测试所有的代码更改，以确保每个版本的质量。

现有数据库逆向工程

若想要通过对现有数据库中的实体类反向工程得出数据模型，可以使用scaffold-dbcontext。可以参阅入门教程。

使用动态 LINQ 来简化对所选内容排序的代码

本系列的第三个教程演示如何通过在 `switch` 语句中对列名称进行硬编码来编写 LINQ 。如果只有两列可供选择，这种方法可行，但是如果拥有许多列，代码可能会变得冗长。要解决该问题，可使用 `EF.Property` 方法将属性名称指定为一个字符串。要尝试此方法，请将 `StudentsController` 中的 `Index` 方法替换为以下代码。

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] =
        String.IsNullOrEmpty(sortOrder) ? "LastName_desc" : "";
    ViewData["DateSortParm"] =
        sortOrder == "EnrollmentDate" ? "EnrollmentDate_desc" : "EnrollmentDate";

    if (searchString != null)
    {
        page = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;

    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                                || s.FirstMidName.Contains(searchString));
    }

    if (string.IsNullOrEmpty(sortOrder))
    {
        sortOrder = "LastName";
    }

    bool descending = false;
    if (sortOrder.EndsWith("_desc"))
    {
        sortOrder = sortOrder.Substring(0, sortOrder.Length - 5);
        descending = true;
    }

    if (descending)
    {
        students = students.OrderByDescending(e => EF.Property<object>(e, sortOrder));
    }
    else
    {
        students = students.OrderBy(e => EF.Property<object>(e, sortOrder));
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(),
        page ?? 1, pageSize));
}
```

后续步骤

这将完成在 ASP.NET Core MVC 应用程序中使用 Entity Framework Core 这一系列教程。

有关 EF Core 的详细信息，请参阅 Entity Framework 的Core文档。另外也可参阅 Entity Framework Core in Action（实际运用 Entity Framework Core）一书。

有关如何部署 web 应用程序的信息，请参阅托管和部署。

有关 ASP.NET Core MVC 相关的其他主题（如身份验证与授权）的信息，请参阅ASP.NET Core文档。

鸣谢

Tom Dykstra 和 Rick Anderson (twitter @RickAndMSFT) 共同编写了本教程。Rowan Miller、Diego Vega 和 Entity Framework 团队的其他成员协助代码评审和帮助解决编写教程代码时出现的调试问题。

常见错误

ContosoUniversity.dll 被另一个进程使用

错误消息：

> 无法打开"...bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- "进程无法访问文件"...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll"，因为它正在由其他进程使用。

解决方案：

停止 IIS Express 中的站点。请转到 Windows 系统任务栏中，找到 IIS Express 并右键单击其图标、选择 Contoso 大学站点，然后单击**停止站点**。

迁移基架的 Up 和 Down 方法中没有代码

可能的原因：

EF CLI 命令不会自动关闭并保存代码文件。如果在运行 `migrations add` 命令时，你未保存更改，EF 将找不到所做的更改。

解决方案：

运行 `migrations remove` 命令，保存你更改的代码并重新运行 `migrations add` 命令。

运行数据库更新时出错

在有数据的数据库中进行架构更改时，很有可能发生其他错误。如果遇到无法解决的迁移错误，你可以更改连接字符串中的数据库名称，或删除数据库。若要迁移，创建新的数据库，在数据库尚没有数据时使用更新数据库命令更有望完成且不发生错误。

最简单方法是在*appsettings.json*中重命名数据库。下次运行 `database update` 时，会创建一个新数据库。

若要在 SSOX 中删除数据库，右键单击数据库，单击**删除**，然后在**删除数据**库对话框框中，选择**关闭现有连接**，单击**确定**。

若要使用 CLI 删除数据库，可以运行 `database drop` CLI 命令：

```
dotnet ef database drop
```

定位 SQL Server 实例出错

错误消息：

> 建立到 SQL Server 的连接时出现与网络相关或特定于实例的错误。未找到或无法访问服务器。请验证实例名称是否正确，SQL Server 是否已配置为允许远程连接。（提供程序：SQL 网络接口，错误：26 - 定位指定的服务器/实例时出错）

解决方案：

请检查连接字符串。如果你已手动删除数据库文件，更改数据库的构造字符串中数据库的名称，然后从头开始使

用新的数据库。

上一篇

ASP.NET Core 和 Entity Framework 6 入门

2018/10/30 • • Edit Online

作者：Paweł Grudzień、Damien Pontifex 和 Tom Dykstra

本文演示如何在 ASP.NET Core 应用程序中使用 Entity Framework 6。

概述

若要使用 Entity Framework 6，则项目必须面向 .NET Framework 进行编译，因为 Entity Framework 6 不支持 .NET Core。如果需要跨平台功能，需升级到 Entity Framework Core。

在 ASP.NET Core 应用程序中使用 Entity Framework 6 的推荐方法是：将 EF6 上下文和模型类放入面向完整框架的类库项目中。添加对 ASP.NET Core 项目中的类库的引用。请参阅示例针对 EF6 和 ASP.NET Core 项目的 Visual Studio 解决方案。

不能将 EF6 上下文放入 ASP.NET Core 项目，因为 .NET Core 项目不支持 EF6 命令（如 Enable-Migrations）所需的的各项功能。

无论 EF6 上下文属于哪种项目类型，只有 EF6 命令行工具才能使用 EF6 上下文。例如，`Scaffold-DbContext` 仅在 Entity Framework Core 中可用。如果需要对数据库执行反向工程以使其成为 EF6 模型，请参阅从 Code First 到现有数据库。

在 ASP.NET Core 项目中引用完整框架和 EF6

ASP.NET Core 项目需要引用 .NET Framework 和 EF6。例如，ASP.NET Core 项目的 .csproj 文件将与以下示例类似（仅显示该文件的相关部分）。

```
<PropertyGroup>
  <TargetFramework>net452</TargetFramework>
  <PreserveCompilationContext>true</PreserveCompilationContext>
  <AssemblyName>MVCCore</AssemblyName>
  <OutputType>Exe</OutputType>
  <PackageId>MVCCore</PackageId>
</PropertyGroup>
```

创建新项目时，请使用 ASP.NET Core Web 应用程序（.NET Framework）模板。

处理连接字符串

需通过默认构造函数在 EF6 类库项目中使用 EF6 命令行工具，以便它们能够实例化上下文。但是，如果想指定要在 ASP.NET Core 项目中使用的连接字符串，则上下文构造函数必须具有可允许你在连接字符串中进行传递的参数。示例如下。

```
public class SchoolContext : DbContext
{
    public SchoolContext(string connString) : base(connString)
    {
    }
```

由于 EF6 上下文不具有无参数构造函数，因此 EF6 项目必须提供 IDbContextFactory 的实现。EF6 命令行工具将查找和使用该实现，以便它们能够实例化上下文。示例如下。

```
public class SchoolContextFactory : IDbContextFactory<SchoolContext>
{
    public SchoolContext Create()
    {
        return new EF6.SchoolContext("Server=
(localdb)\\mssqllocaldb;Database=EF6MVCCore;Trusted_Connection=True;MultipleActiveResultSets=true");
    }
}
```

在此示例代码中，`IDbContextFactory` 实现将在硬编码的连接字符串中传递。这是命令行工具要使用的连接字符串。你需要实施策略以确保类库与调用应用程序使用相同的连接字符串。例如，可从这两个项目的环境变量中获取值。

在 ASP.NET Core 项目中设置依赖项注入

在 Core 项目的 Startup.cs 文件中，为 `ConfigureServices` 中的依赖项注入 (DI) 设置 EF6 上下文。应将 EF 上下文对象的范围设置为按请求生存期。

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
    services.AddScoped<SchoolContext>(_ => new
SchoolContext(Configuration.GetConnectionString("DefaultConnection")));
}
```

然后即可使用 DI 在控制器中获取上下文的实例。此代码与针对 EF Core 上下文编写的代码相似：

```
public class StudentsController : Controller
{
    private readonly SchoolContext _context;

    public StudentsController(SchoolContext context)
    {
        _context = context;
    }
```

示例应用程序

若要获取有效的示例应用程序，请参阅本文随附的示例 Visual Studio 解决方案。

可在 Visual Studio 中按照以下步骤从头创建此示例：

- 创建解决方案。
- 添加 > 新建项目 > Web > ASP.NET Core Web 应用程序
  - 在"项目模板选择"对话框的下拉列表中，选择 API 和 .NET Framework
- 添加 > 新建项目 > Windows 桌面 > 类库(.NET Framework)
- 在两个项目的"包管理器控制台"(PMC) 中运行 `Install-Package Entityframework` 命令。
- 在类库项目中，创建数据模型类和上下文类，并创建 `IDbContextFactory` 的实现。
- 在类库项目的 PMC 中，运行 `Enable-Migrations` 和 `Add-Migration Initial` 命令。如果已将 ASP.NET Core 项目设置为启动项目，请向这些命令添加 `-StartupProjectName EF6`。
- 在 Core 项目中，添加对类库项目的项目引用。
- 在 Core 项目的 Startup.cs 中，为 DI 注册上下文。

- 在 Core 项目的 appsettings.json 中，添加连接字符串。
- 在 Core 项目中，添加控制器和视图以验证可读取和写入数据。（请注意，ASP.NET Core MVC 基架不会使用从类库引用的 EF6 上下文。）

总结

本文提供了在 ASP.NET Core 应用程序中使用 Entity Framework 6 的基本指南。

其他资源

- Entity Framework - 基于代码的配置

ASP.NET Core 中的客户端开发

2018/10/18 • • Edit Online

- 使用 Gulp
- 使用 Grunt
- 使用 LibMan
  - LibMan CLI
  - Visual Studio 中的 LibMan
- 使用 Bower 管理客户端包
- 使用 LESS、Sass 和 Font Awesome 为应用设置样式
- 捆绑和缩小
- TypeScript
- 使用浏览器链接
- 对 SPA 使用 JavaScriptServices
- 使用 SPA 项目模板
  - Angular 项目模板
  - React 项目模板
  - 带 Redux 的 React 项目模板

在 ASP.NET Core 中使用 Gulp

2018/12/17 • • Edit Online

通过Scott Addie，Shayne Boyer，和David 松树

在典型的现代 Web 应用中，生成过程可能包括以下操作：

- 捆绑和缩小 JavaScript 和 CSS 文件。
- 运行工具以调用之前每个生成的绑定和缩减任务。
- 将 LESS 或 SASS 文件编译成 CSS。
- 将 CoffeeScript 或 TypeScript 文件编译成 JavaScript。

任务运行程序是一种自动执行这些常规开发任务和其他任务的工具。Visual Studio 为下述两种常用的基于 JavaScript 的任务运行程序提供内置支持：Gulp 和 Grunt。

Gulp

Gulp 是一个基于 JavaScript 的流式处理生成工具包，用于客户端代码。它通常用于通过一系列的进程的客户端文件流式传输特定的事件触发的生成环境中时。例如，使用 Gulp 来自动执行捆绑和缩小或清理之前新生成的开发环境。

中定义一组的 Gulp 任务*gulpfile.js*。以下 JavaScript 包括 Gulp 模块，并指定文件路径，以引用中即将推出的任务：

```
/// <binding Clean='clean' />
"use strict";

var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");

var paths = {
  webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.minJs = paths.webroot + "js/**/*.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.minCss = paths.webroot + "css/**/*.min.css";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.concatCssDest = paths.webroot + "css/site.min.css";
```

上面的代码中指定需要哪些节点模块。`require`函数导入的每个模块，以便依赖任务能够利用其功能。每个导入的模块被分配给一个变量。模块可位于通过名称或路径。在此示例中，模块名为`gulp`，`rimraf`，`gulp-concat`，`gulp-cssmin`，和`gulp-uglify`按名称检索。此外，创建一系列的路径，以便可以重复使用并在任务中引用的 CSS 和 JavaScript 文件的位置。下表提供了说明中包含的模块*gulpfile.js*。

模块名	描述
Gulp	Gulp 流式处理生成系统。有关详细信息，请参阅gulp。
rimraf	节点删除模块。有关详细信息，请参阅rimraf。

模块名	描述
gulp concat	连接基于操作系统的换行字符的文件的模块。有关详细信息，请参阅gulp concat。
gulp-cssmin	缩减 CSS 文件的模块。有关详细信息，请参阅gulp cssmin。
gulp 丑化	一个模块，缩减 *.js*文件。有关详细信息，请参阅gulp 丑化。

一旦导入必要模块，可以指定任务。此处有六个任务注册，通过以下代码表示：

```
gulp.task("clean:js", done => rimraf(paths.concatJsDest, done));
gulp.task("clean:css", done => rimraf(paths.concatCssDest, done));
gulp.task("clean", gulp.series(["clean:js", "clean:css"]));

gulp.task("min:js", () => {
  return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
    .pipe(concat(paths.concatJsDest))
    .pipe(uglify())
    .pipe(gulp.dest("."));
});

gulp.task("min:css", () => {
  return gulp.src([paths.css, "!" + paths.minCss])
    .pipe(concat(paths.concatCssDest))
    .pipe(cssmin())
    .pipe(gulp.dest("."));
});

gulp.task("min", gulp.series(["min:js", "min:css"]));

// A 'default' task is required by Gulp v4
gulp.task("default", gulp.series(["min"]));
```

下表提供了上面的代码中指定的任务的说明：

任务名称	描述
clean:js	一个使用 rimraf Node 删除模块删除缩减版 site.js 文件的任务。
clean:css	一个使用 rimraf Node 删除模块删除缩减版 site.css 文件的的任务。
清理	调用的任务 `clean:js` 任务中后，跟 `clean:css` 任务。
min:js	缩减，并将连接的 js 文件夹中的所有.js 文件的任务。。Min.js 文件将不会。
min:css	一个任务，缩减，并将连接的 css 文件夹中的所有.css 文件。。Min.css 文件将不会。
min	调用的任务 `min:js` 任务中后，跟 `min:css` 任务。

正在运行的默认任务

如果你尚未创建新的 Web 应用，Visual Studio 中创建一个新的 ASP.NET Web 应用程序项目。

1. 打开*package.json*文件 (添加如果不是存在) 并添加以下。

```
{
  "devDependencies": {
    "gulp": "^4.0.0",
    "gulp-concat": "2.6.1",
    "gulp-cssmin": "0.2.0",
    "gulp-uglify": "3.0.0",
    "rimraf": "2.6.1"
  }
}
```

2. 将新的 JavaScript 文件添加到你的项目并将其命名*gulpfile.js*，然后将以下代码复制。

```
/// <binding Clean='clean' />
"use strict";

const gulp = require("gulp"),
      rimraf = require("rimraf"),
      concat = require("gulp-concat"),
      cssmin = require("gulp-cssmin"),
      uglify = require("gulp-uglify");

const paths = {
  webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.minJs = paths.webroot + "js/**/*.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.minCss = paths.webroot + "css/**/*.min.css";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.concatCssDest = paths.webroot + "css/site.min.css";

gulp.task("clean:js", done => rimraf(paths.concatJsDest, done));
gulp.task("clean:css", done => rimraf(paths.concatCssDest, done));
gulp.task("clean", gulp.series(["clean:js", "clean:css"]));

gulp.task("min:js", () => {
  return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
    .pipe(concat(paths.concatJsDest))
    .pipe(uglify())
    .pipe(gulp.dest("."));
});

gulp.task("min:css", () => {
  return gulp.src([paths.css, "!" + paths.minCss])
    .pipe(concat(paths.concatCssDest))
    .pipe(cssmin())
    .pipe(gulp.dest("."));
});

gulp.task("min", gulp.series(["min:js", "min:css"]));

// A 'default' task is required by Gulp v4
gulp.task("default", gulp.series(["min"]));
```

3. 在中**解决方案资源管理器**，右键单击*gulpfile.js*，然后选择**Task Runner Explorer**。

Task Runner Explorer显示 Gulp 任务的列表。(您可能需要单击**刷新**显示项目名称左侧的按钮。)

> **IMPORTANT**
>
> **Task Runner Explorer**才会显示上下文菜单项*gulpfile.js*根项目目录中。

4. 在"任务运行程序资源管理器"中的"任务"下，右键单击"clean"，然后从弹出菜单中选择"运行"。

任务运行程序资源管理器将创建名为"clean"的新选项卡，并根据 *gulpfile.js* 中的定义执行 clean 任务。

5. 右键单击"clean"任务，然后选择"绑定">"生成之前" > 。

"生成之前"绑定会配置 clean 任务，使之在每次生成项目之前自动运行。

使用任务**运行程序**资**源管理器**设置的绑定以注释形式存储在 *gulpfile.js* 顶部，仅在 Visual Studio 中有效。一种不需要 Visual Studio 的替代方法是将 *csproj* 文件中的 gulp 任务配置为自动执行。例如，将以下代码置于 *.csproj* 文件中：

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="gulp clean" />
</Target>
```

现在，在 Visual Studio 中或从命令提示符处使用运行项目时执行清理任务dotnet 运行命令 (运行 `npm install` 第一个)。

定义和运行新任务

若要定义新的 Gulp 任务，请修改*gulpfile.js*。

1. 将以下 JavaScript 添加到末尾*gulpfile.js*:

```
gulp.task('first', done => {
  console.log('first task! <-----');
  done(); // signal completion
});
```

   此任务名为 `first`，和它就会显示一个字符串。

2. 保存*gulpfile.js*。

3. 在中**解决方案**资**源管理器**，右键单击*gulpfile.js*，然后选择*Task Runner Explorer*。

4. 在中**Task Runner Explorer**，右键单击**第一个**，然后选择**运行**。

此时会显示输出文本。若要查看基于常见方案的示例，请参阅 Gulp 脚本。

定义和运行一系列任务

运行多个任务时，默认情况下这些任务会并发运行。但是，如果需要以特定顺序运行任务，则必须指定每个任务的具体完成时间，以及哪些任务依赖于其他任务的完成。

1. 若要定义一系列任务按顺序运行，请替换 `first` 中前面添加的任务*gulpfile.js*以下：

```
gulp.task('series:first', done => {
  console.log('first task! <-----');
  done(); // signal completion
});
gulp.task('series:second', done => {
  console.log('second task! <-----');
  done(); // signal completion
});

gulp.task('series', gulp.series(['series:first', 'series:second']), () => { });

// A 'default' task is required by Gulp v4
gulp.task('default', gulp.series('series'));
```

   现在有三个任务：`series:first`，`series:second`，和 `series`。`series:second` 任务包括第二个参数，以便指定要运行和前完成的任务的数组 `series:second` 任务将运行。上面，唯一的代码中指定的那样 `series:first` 必须在任务完成之前 `series:second` 任务将运行。

2. 保存*gulpfile.js*。

3. 在中**解决方案资源管理器**，右键单击*gulpfile.js*，然后选择**Task Runner Explorer**如果尚未打开。

4. 在中**Task Runner Explorer**，右键单击**系列**，然后选择**运行**。

IntelliSense

IntelliSense 提供了代码完成、参数说明和其他功能以提高工作效率并减少错误。Gulp 任务是用 JavaScript 编写的;因此，IntelliSense 可以提供开发时获得帮助。使用 JavaScript 时，IntelliSense 会列出对象、函数、属性和可用的参数根据当前上下文。从 intellisense 以完成代码提供的弹出列表中选择编码选项。

有关 IntelliSense 的详细信息，请参阅JavaScript IntelliSense。

开发、过渡和生产环境

当 Gulp 用于优化过渡和生产客户端的文件时，已处理的文件保存到本地的过渡和生产位置。*_Layout.cshtml*文件使用环境的标记帮助程序提供两个不同版本的 CSS 文件。CSS 文件的一个版本用于开发，另一个版本适用于过渡和生产。在 Visual Studio 2017 中，更改时**ASPNETCORE_ENVIRONMENT**环境变量为 `Production`，Visual Studio 将生成 Web 应用并链接到最小化的 CSS 文件。下面的标记演示环境标记帮助程序包含将标记与 `Development` CSS 文件和缩小 `Staging, Production` CSS 文件。

```
<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
            asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
            asp-fallback-test="window.jQuery"
            crossorigin="anonymous"
            integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
            asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
            asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
            crossorigin="anonymous"
            integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
    </script>
    <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>
```

在环境之间切换

若要针对不同的环境编译之间切换，请修改**ASPNETCORE_ENVIRONMENT**环境变量的值。

1. 在**任务运行程序资源管理器**中，验证 **min** 任务是否已设置为在**生成之前**运行。

2. 在中**解决方案资源管理器**，右键单击项目名称并选择**属性**。

显示 Web 应用的属性表。

3. 单击“调试”选项卡。
4. 设置的值**宿主：环境**环境变量为 `Production` 。
5. 按**F5**在浏览器中运行应用程序。
6. 在浏览器窗口中，右键单击页并选择查**看源**若要查看该页面的 HTML。

   请注意，样式表链接指向的缩减的 CSS 文件。

7. 关闭浏览器来停止 Web 应用。
8. 在 Visual Studio 中，返回到 Web 应用的属性表，并更改**宿主：环境**环境变量回 `Development` 。
9. 按**F5**再次在浏览器中运行应用程序。
10. 在浏览器窗口中，右键单击页并选择查**看源**若要查看该页面的 HTML。

    请注意，样式表链接指向 CSS 文件的 unminified 版本。

有关 ASP.NET Core 中的环境的详细信息，请参阅使用多个环境。

任务和模块的详细信息

Gulp 任务已注册到函数名称。如果当前任务之前必须运行其他任务，你可以指定依赖关系。其他函数，您可以运行和监视的 Gulp 任务，以及将源设置 (*src*) 和目标 (*dest*) 正在修改的文件。下面是主要的 Gulp API 函数：

GULP 函数	语法	描述
任务	`gulp.task(name[, deps], fn) { }`	`task` 函数创建的任务。 `name` 参数定义任务的名称。 `deps` 参数包含要完成此任务运行之前任务的数组。 `fn` 参数表示执行任务的操作的回调函数。
监视	`gulp.watch(glob [, opts], tasks) { }`	`watch` Function 可监视文件和运行任务发生文件更改时。 `glob` 参数是 `string` 或 `array` ，它确定要监视哪些文件。 `opts` 参数提供了监视选项的其他文件。
src	`gulp.src(globs[, options]) { }`	`src` 函数提供了与 glob 值匹配的文件。 `glob` 参数是 `string` 或 `array` ，它确定哪些文件读取。 `options` 参数提供了附加文件选项。
dest	`gulp.dest(path[, options]) { }`	`dest` 函数用于定义要向其写入文件的位置。 `path` 参数是字符串或函数，用于确定目标文件夹。 `options` 参数是一个对象，指定输出文件夹选项。

有关其他 Gulp API 参考信息，请参阅Gulp Docs API。

Gulp 脚本

Gulp 社区提供 Gulp 脚本。这些脚本包含适用于常见方案的 Gulp 任务。

其他资源

- Gulp 文档
- 绑定和缩减中 ASP.NET Core
- 在 ASP.NET Core 中使用 Grunt

在 ASP.NET Core中使用 Grunt

2018/7/4 • • Edit Online

通过了米

Grunt 是自动化脚本缩减、TypeScript 编译、代码质量"链接形式"工具、CSS 预处理器和几乎任何重复繁琐的工作，需要采取措施来支持客户端开发的 JavaScript 任务运行程序。尽管 ASP.NET 项目模板默认情况下使用 Gulp Visual Studio 中，完全支持 grunt (请参阅使用 Gulp)。

此示例使用一个空的 ASP.NET Core 项目作为起始点，以显示如何从零开始的客户端生成过程中自动运行。

完成的示例清除目标部署目录、将合并 JavaScript 文件，检查代码质量，会将 JavaScript 文件内容和将部署到 web 应用程序的根目录。我们将使用下列包：

- **grunt**: Grunt 任务运行程序包。
- **grunt contrib 清理**：删除文件或目录的插件。
- **grunt contrib jshint**：查看 JavaScript 代码质量的插件。
- **grunt contrib concat**：将文件联接成单个文件的插件。
- **grunt contrib uglify**: minifies JavaScript 以减少大小的插件。
- **grunt contrib 监视**：监视文件活动的插件。

准备好应用程序

若要开始，设置新的空 web 应用程序，并添加 TypeScript 示例文件。TypeScript 文件将会自动编译到 JavaScript 使用默认 Visual Studio 设置，并且将是我们原材料处理使用 Grunt。

1. 在 Visual Studio 中，创建一个新 `ASP.NET Web Application` 。
2. 在**新建 ASP.NET 项目**对话框中，选择 ASP.NET Core**空模板**，然后单击**确定**按钮。
3. 在解决方案资源管理器，查看的项目结构。 `\src` 文件夹包含空 `wwwroot` 和 `Dependencies` 节点。

4. 添加一个名为的新文件夹 `TypeScript` 到你的项目目录。
5. 在添加之前的任何文件，请确保 Visual Studio 具有选项编译保存 TypeScript 文件检查。导航到**工具** > 选项 > **文本编辑器** > **Typescript** > 项目：

6. 右键单击 `TypeScript` 目录，然后选择**添加 > 新项**从上下文菜单。选择**JavaScript 文件**项并将该文件命名*Tastes.ts* (请注意*.ts 扩展)。将以下 TypeScript 代码的行复制到文件 (在您保存，新*Tastes.js*文件将出现与 JavaScript 源)。

```
enum Tastes { Sweet, Sour, Salty, Bitter }
```

7. 添加的第二个文件，以便**TypeScript**目录并将其命名 `Food.ts` 。将下面的代码复制到文件。

```
class Food {
  constructor(name: string, calories: number) {
    this._name = name;
    this._calories = calories;
  }

  private _name: string;
  get Name() {
    return this._name;
  }

  private _calories: number;
  get Calories() {
    return this._calories;
  }

  private _taste: Tastes;
  get Taste(): Tastes { return this._taste }
  set Taste(value: Tastes) {
    this._taste = value;
  }
}
```

配置 NPM

接下来，配置 NPM 来下载 grunt 和 grunt 任务。

1. 在解决方案资源管理器，右键单击该项目并选择**添加 > 新项**从上下文菜单。选择**NPM 配置文件**项，保留默认名称，*package.json*，然后单击**添加**按钮。

2. 在*package.json*文件内，`devDependencies` 对象大括号中，输入"grunt"。选择 `grunt` 从智能感知列表并按 Enter 键。Visual Studio 将 quote grunt 包名称，并添加一个冒号。从 Intellisense 列表的顶部中的冒号右侧，选择包的最新稳定版本 (按 `Ctrl-Space` 如果 Intellisense 不显示)。

> **NOTE**
>
> 使用 NPM语义版本控制来组织依赖关系。语义版本控制，也称为 SemVer，标识包的编号方案。。.Intellisense 显示仅几个常用的选项，从而简化了语义版本控制。Intellisense 列表（在上面的示例 0.4.5）中的顶级项被视为包的最新稳定版本。脱字号（^）符号匹配的最新的主版本和波形符（）与最新次要版本匹配。请参阅NPM semver 版本分析器参考作为 SemVer 提供其完整表达的指南。

3. 添加更多的依赖关系，以加载 grunt-contrib-*打包以*干净*，*jshint*，*concat*，*uglify*，和*监视*下面的示例中所示。版本不需要与示例匹配。

```
"devDependencies": {
  "grunt": "0.4.5",
  "grunt-contrib-clean": "0.6.0",
  "grunt-contrib-jshint": "0.11.0",
  "grunt-contrib-concat": "0.5.1",
  "grunt-contrib-uglify": "0.8.0",
  "grunt-contrib-watch": "0.6.1"
}
```

4. 保存*package.json*文件。

将下载的每个 devDependencies 项包，以及每个包需要的任何文件。你可以查找中的包文件 `node_modules` 通过启用目录显**示所有文件**在解决方案资源管理器的按钮。

> **NOTE**
>
> 如果需要可以通过右键单击来手动还原解决方案资源管理器中的依赖关系 `Dependencies\NPM` 并选择**还原包**菜单选项。

配置 Grunt

使用名为的清单配置 grunt *Gruntfile.js*的定义、加载和注册任务可以手动运行或配置以自动基于运行 Visual Studio 中的事件。

1. 右键单击项目并选择**添加 > 新**项。选择**Grunt 配置文件**选项，请保留默认名称，*Gruntfile.js*，然后单击**添加**按钮。

   初始代码包含了一个模块定义和 `grunt.initConfig()` 方法。 `initConfig()` 用于为每个包中，设置选项和模块的剩余部分将会加载和注册任务。

```
module.exports = function (grunt) {
  grunt.initConfig({
  });
};
```

2. 内部 `initConfig()` 方法，添加用于 `clean` 任务示例中所示*Gruntfile.js*下面。Clean 任务，接受目录字符串的数组。此任务从 wwwroot/lib 中删除文件，并将移除整个/临时目录。

```
module.exports = function (grunt) {
  grunt.initConfig({
    clean: ["wwwroot/lib/*", "temp/"],
  });
};
```

3. 下面 initConfig() 方法中，添加对的调用 `grunt.loadNpmTasks()` 。这将从 Visual Studio 来使该任务可运行。

```
grunt.loadNpmTasks("grunt-contrib-clean");
```

4. 保存*Gruntfile.js*。该文件应类似于下面的屏幕截图。

5. 右键单击*Gruntfile.js*和选择**任务运行程序资源管理器**从上下文菜单。任务运行程序资源管理器窗口将打开。

6. 验证 `clean` 下显示任务在任务运行程序资源管理器。

7. 右键单击 clean 任务，然后选择**运行**从上下文菜单。命令窗口将显示任务的进度。

> **NOTE**
>
> 没有任何文件或目录尚未清理。如果您愿意，你可以在解决方案资源管理器中手动创建它们，然后运行为测试的 clean 任务。

8. 在 initConfig() 方法中，添加一个条目 `concat` 使用下面的代码。

   `src` 属性数组列出文件合并，它们应合并的顺序。 `dest` 属性将路径分配给组合生成的文件。

```
concat: {
  all: {
    src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
    dest: 'temp/combined.js'
  }
},
```

> **NOTE**
>
> `all` 在上面的代码的属性为目标的名称。某些 Grunt 任务中使用的目标用于允许多个生成环境。你可以查看内置的目标使用 Intellisense，或指定你自己。

9. 添加 `jshint` 任务使用下面的代码。

   Jshint 代码质量实用工具运行的临时目录中找到的每个 JavaScript 文件。

```
jshint: {
  files: ['temp/*.js'],
  options: {
    '-W069': false,
  }
},
```

> **NOTE**
>
> 选项"-W069"一个错误时产生的 jshint JavaScript 使用方括号语法，即分配而不是点表示法，属性 `Tastes["Sweet"]` 而不是 `Tastes.Sweet` 。选项关闭了警告，以允许进程继续的其余部分。

10. 添加 `uglify` 任务使用下面的代码。

    任务 minifies *combined.js*文件的临时目录中找到并在 wwwroot/lib 以下标准命名约定中创建结果文件*<文件名>。min.js*.

```
uglify: {
 all: {
   src: ['temp/combined.js'],
   dest: 'wwwroot/lib/combined.min.js'
 }
},
```

11. 在加载 grunt contrib 清理调用 grunt.loadNpmTasks()，包括相同的调用，用于 jshint，concat 并 uglify 使用下面的代码使用。

```
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');
```

12. 保存*Gruntfile.js*。该文件应类似于下面的示例。

13. 请注意，该任务运行程序资源管理器任务列表包括 `clean`， `concat`， `jshint` 和 `uglify` 任务。按顺序运行每个任务，并观察解决方案资源管理器中的结果。每个任务应运行且未发生错误。

Concat 任务创建一个新*combined.js*文件并将其放到临时目录。Jshint 任务只需运行，但不生成输出。Uglify 任务创建一个新*combined.min.js*文件并将其放到 wwwroot/lib。完成时，解决方案应类似下面的屏幕截图：

> **NOTE**
>
> 每个包的选项的详细信息，请访问 https://www.npmjs.com/ 和查找在主页上的搜索框中的包名称。例如，你可以查找 grunt contrib 清理包以获取说明它的所有参数的文档链接。

总的来说

使用 Grunt `registerTask()` 方法以按特定顺序运行一系列任务。例如，若要运行示例上述顺序干净的步骤-> concat-> jshint-> uglify，将下面的代码添加到该模块。应将代码添加到与外部 initConfig loadNpmTasks() 调用相同的级别。

```
grunt.registerTask("all", ['clean', 'concat', 'jshint', 'uglify']);
```

新任务将显示在任务运行程序资源管理器的别名任务中。您可以右键单击，并运行它，就像其他任务。 `all` 任务将运行 `clean` ， `concat` ， `jshint` 和 `uglify` ，按顺序。

监视更改

A `watch` 任务保留文件和目录。如果它检测到更改，监视将自动触发任务。将下面的代码添加到 initConfig，若要监视的更改*TypeScript 目录中的.js 文件。如果更改 JavaScript 文件， `watch` 将运行 `all` 任务。

```
watch: {
  files: ["TypeScript/*.js"],
  tasks: ["all"]
}
```

添加对的调用 `loadNpmTasks()` 以显示 `watch` 任务运行程序资源管理器中的任务。

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

右键单击任务运行程序资源管理器中的监视任务，并从上下文菜单中选择运行。显示运行的监视任务的命令窗口将显示"等待..." 消息。打开一个 TypeScript 文件，添加一个空格，然后保存该文件。这将触发监视任务并触发其他任务以按顺序运行。下面的屏幕截图显示了示例运行。

绑定到 Visual Studio 事件

除非你想要手动启动你的任务，每次在 Visual Studio 中工作时，你可以将绑定到的任务**之前生成**，**后生成**，**清理**，和项**目打开**事件。

接下来将绑定 `watch` 以便使其运行每次 Visual Studio 将打开。在任务运行程序资源管理器，右键单击监视任务并选择**绑定** > 项**目打开**从上下文菜单。

卸载并重新加载项目。在项目加载后再次，监视任务将开始自动运行。

总结

Grunt 是可以用于自动执行大多数客户端生成任务的功能强大的任务运行程序。Grunt 利用 NPM 来提供其包和工具与 Visual Studio 的集成的功能。Visual Studio 的任务运行程序资源管理器检测到配置文件的更改，并提供方便的界面以运行任务，查看正在运行的任务，并将任务绑定到 Visual Studio 事件。

其他资源

- 使用 Gulp

通过 LibMan 在 ASP.NET Core 中获取客户端库

2018/11/30 • • Edit Online

作者：Scott Addie

库管理器 (LibMan) 是一个轻量型客户端库获取工具。LibMan 可从文件系统或从内容分发网络 (CDN) 下载库和框架。支持的 CDN 包括 CDNJS 和 unpkg。将提取所选库文件，并将其置于 ASP.NET Core 项目中的相应位置。

LibMan 用例

LibMan 提供以下优势：

- 只会下载所需的库文件。
- 无需使用其他工具（例如 Node.js、npm 和 WebPack），即可获取库中文件的子集。
- 可将文件放置在特定位置，无需执行生成任务，也不需手动进行文件复制。

有关 LibMan 优势的详细信息，请观看 Modern front-end web development in Visual Studio 2017: LibMan segment（Visual Studio 2017 中的新式前端 Web 开发：LibMan 段）。

LibMan 不是程序包管理系统。如果已在使用包管理器（例如 npm 或yarn），请继续使用它们。LibMan 不是为取代这些工具而开发的。

其他资源

- 在 Visual Studio 中的 ASP.NET Core 中使用 LibMan
- ASP.NET Core 中使用 LibMan 命令行接口 (CLI)
- LibMan GitHub 存储库

ASP.NET Core 中使用 LibMan 命令行接口 (CLI)

2018/9/27 • • Edit Online

作者：Scott Addie

LibMan CLI 是具有支持的任何位置均支持.NET Core 的跨平台工具。

系统必备

- .NET Core 2.1 SDK 或更高版本

安装

若要安装 LibMan CLI:

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli
```

一个.NET Core 全局工具从安装Microsoft.Web.LibraryManager.Cli NuGet 包。

若要从特定的 NuGet 包源安装 LibMan CLI:

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli --version 1.0.94-g606058a278 --add-source C:\Temp\
```

在前面的示例中，从本地 Windows 计算机的安装.NET Core 全局工具*C:\Temp\Microsoft.Web.LibraryManager.Cli.1.0.94-g606058a278.nupkg*文件。

用法

安装成功后的 cli，可以使用以下命令：

```
libman
```

若要查看已安装的 CLI 版本：

```
libman --version
```

若要查看可用的 CLI 命令：

```
libman --help
```

上述命令将显示类似于以下输出：

```
 1.0.163+g45474d37ed

Usage: libman [options] [command]

Options:
  --help|-h  Show help information
  --version  Show version information

Commands:
  cache      List or clean libman cache contents
  clean      Deletes all library files defined in libman.json from the project
  init       Create a new libman.json
  install    Add a library definition to the libman.json file, and download the
             library to the specified location
  restore    Downloads all files from provider and saves them to specified
             destination
  uninstall  Deletes all files for the specified library from their specified
             destination, then removes the specified library definition from
             libman.json
  update     Updates the specified library

Use "libman [command] --help" for more information about a command.
```

以下各节概述了可用的 CLI 命令。

在项目中初始化 LibMan

`libman init` 命令创建*libman.json*文件尚不存在。使用默认项目模板内容创建该文件。

摘要

```
libman init [-d|--default-destination] [-p|--default-provider] [--verbosity]
libman init [-h|--help]
```

选项

有以下选项 `libman init` 命令：

- `-d|--default-destination <PATH>`

  当前文件夹的相对路径。如果不是库文件在此位置安装 `destination` 属性定义为库*libman.json*。`<PATH>` 值写入 `defaultDestination` 的属性*libman.json*。

- `-p|--default-provider <PROVIDER>`

  如果没有提供程序定义为给定库使用的提供程序。`<PROVIDER>` 值写入 `defaultProvider` 的属性*libman.json*。替换为 `<PROVIDER>` 使用以下值之一：

  - `cdnjs`
  - `filesystem`
  - `unpkg`

- `-h|--help`

  显示帮助信息。

- `--verbosity <LEVEL>`

  设置输出的详细级别。替换为 `<LEVEL>` 使用以下值之一：

  - `quiet`

- `normal`
- `detailed`

示例

若要创建*libman.json* ASP.NET Core 项目文件中：

- 导航到项目根目录。
- 运行下面的命令：

```
libman init
```

- 键入的默认提供程序或按名称 `Enter` 以使用默认 CDNJS 提供程序。有效值包括：
  - `cdnjs`
  - `filesystem`
  - `unpkg`

一个*libman.json*文件添加到项目根目录包含以下内容：

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": []
}
```

添加库文件

`libman install` 命令下载和安装到项目的库文件。一个*libman.json*如果不存在添加文件。*Libman.json*修改文件来存储配置的库文件的详细信息。

摘要

```
libman install <LIBRARY> [-d|--destination] [--files] [-p|--provider] [--verbosity]
libman install [-h|--help]
```

自变量

`LIBRARY`

若要安装的库的名称。此名称可能包含版本的数字符号 (例如， `@1.2.0`)。

选项

有以下选项 `libman install` 命令：

- `-d|--destination <PATH>`

  要安装此库的位置。如果未指定，使用默认位置。如果没有`defaultDestination`属性中指定*libman.json*，则需要此选项。

- `--files <FILE>`

  指定要从库中安装的文件的名称。如果未指定，安装库中的所有文件。提供一个`--files`每个文件的选项来安装。也支持相对路径。例如：`--files dist/browser/signalr.js`。

- `-p|--provider <PROVIDER>`

  要用于库获取提供程序的名称。替换为`<PROVIDER>`使用以下值之一：

  - `cdnjs`
  - `filesystem`
  - `unpkg`

  如果未指定，否则`defaultProvider`中的属性*libman.json*使用。如果没有`defaultProvider`属性中指定*libman.json*，则需要此选项。

- `-h|--help`

  显示帮助信息。

- `--verbosity <LEVEL>`

  设置输出的详细级别。替换为`<LEVEL>`使用以下值之一：

  - `quiet`
  - `normal`
  - `detailed`

示例

请考虑以下*libman.json*文件：

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": []
}
```

若要安装 jQuery 版本 3.2.1 *jquery.min.js*的文件*wwwroot/脚本/jquery*使用 CDNJS 提供程序的文件夹：

```
libman install jquery@3.2.1 --provider cdnjs --destination wwwroot/scripts/jquery --files jquery.min.js
```

*Libman.json*文件如下所示：

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.2.1",
      "destination": "wwwroot/scripts/jquery",
      "files": [
        "jquery.min.js"
      ]
    }
  ]
}
```

若要安装*calendar.js*并*calendar.css*从文件*c:\temp\contosoCalendar* 使用文件系统提供程序：

```
libman install C:\temp\contosoCalendar\ --provider filesystem --files calendar.js --files calendar.css
```

以下提示将显示以下两个原因：

- *Libman.json*文件不包含 `defaultDestination` 属性。
- `libman install` 命令不包含 `-d|--destination` 选项。

接受默认目标之后，*libman.json*文件如下所示：

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.2.1",
      "destination": "wwwroot/scripts/jquery",
      "files": [
        "jquery.min.js"
      ]
    },
    {
      "library": "C:\\temp\\contosoCalendar\\",
      "provider": "filesystem",
      "destination": "wwwroot/lib/contosoCalendar",
      "files": [
        "calendar.js",
        "calendar.css"
      ]
    }
  ]
}
```

库将文件还原

`libman restore`命令安装库文件中定义*libman.json*。适用以下规则：

- 如果没有*libman.json*项目根目录中存在文件，则返回错误。
- 如果库指定的提供程序，`defaultProvider`中的属性*libman.json*将被忽略。
- 如果库指定目标`defaultDestination`中的属性*libman.json*将被忽略。

摘要

```
libman restore [--verbosity]
libman restore [-h|--help]
```

选项

有以下选项`libman restore`命令：

- `-h|--help`

  显示帮助信息。

- `--verbosity <LEVEL>`

  设置输出的详细级别。替换为`<LEVEL>`使用以下值之一：

  - `quiet`
  - `normal`
  - `detailed`

示例

若要还原中定义的库文件*libman.json*:

```
libman restore
```

删除库文件

`libman clean`命令可删除以前还原通过 LibMan 库文件。会删除此操作后，会为空的文件夹。库文件的关联中的配置`libraries`的属性*libman.json*不会删除。

摘要

```
libman clean [--verbosity]
libman clean [-h|--help]
```

选项

有以下选项`libman clean`命令：

- `-h|--help`

  显示帮助信息。

- `--verbosity <LEVEL>`

  设置输出的详细级别。替换为`<LEVEL>`使用以下值之一：

  - `quiet`

- `normal`
- `detailed`

示例

若要删除通过 LibMan 安装的库文件：

```
libman clean
```

卸载库文件

`libman uninstall` 命令：

- 删除与指定的库中的目标从关联的所有文件*libman.json*。
- 将删除从关联的库配置*libman.json*。

发生错误时：

- 否*libman.json*项目根目录中存在的文件。
- 不存在指定的库。

如果安装多个具有相同名称的库，则系统会提示选择一个。

摘要

```
libman uninstall <LIBRARY> [--verbosity]
libman uninstall [-h|--help]
```

自变量

`LIBRARY`

要卸载的库的名称。此名称可能包含版本的数字符号（例如，`@1.2.0`）。

选项

有以下选项 `libman uninstall` 命令：

- `-h|--help`

  显示帮助信息。

- `--verbosity <LEVEL>`

  设置输出的详细级别。替换为 `<LEVEL>` 使用以下值之一：

  - `quiet`
  - `normal`
  - `detailed`

示例

请考虑以下*libman.json*文件：

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.3.1",
      "files": [
        "jquery.min.js",
        "jquery.js",
        "jquery.min.map"
      ],
      "destination": "wwwroot/lib/jquery/"
    },
    {
      "provider": "unpkg",
      "library": "bootstrap@4.1.3",
      "destination": "wwwroot/lib/bootstrap/"
    },
    {
      "provider": "filesystem",
      "library": "C:\\temp\\lodash\\",
      "files": [
        "lodash.js",
        "lodash.min.js"
      ],
      "destination": "wwwroot/lib/lodash/"
    }
  ]
}
```

- 若要卸载 jQuery，以下命令之一成功：

```
libman uninstall jquery
```

```
libman uninstall jquery@3.3.1
```

- 若要卸载 lodash 等文件通过安装 `filesystem` 提供程序：

```
libman uninstall C:\temp\lodash\
```

更新库版本

`libman update` 命令更新通过 LibMan 安装到指定版本的库。

发生错误时：

- 否*libman.json*项目根目录中存在的文件。
- 不存在指定的库。

如果安装多个具有相同名称的库，则系统会提示选择一个。

摘要

```
libman update <LIBRARY> [-pre] [--to] [--verbosity]
libman update [-h|--help]
```

自变量

`LIBRARY`

要更新的库的名称。

选项

有以下选项 `libman update` 命令：

- `-pre`

  获取库的最新的预发布版本。

- `--to <VERSION>`

  获取特定版本的库。

- `-h|--help`

  显示帮助信息。

- `--verbosity <LEVEL>`

  设置输出的详细级别。替换为 `<LEVEL>` 使用以下值之一：

  - `quiet`
  - `normal`
  - `detailed`

示例

- 若要更新到最新版本的 jQuery:

```
libman update jquery
```

- 若要更新到版本 3.3.1 jQuery:

```
libman update jquery --to 3.3.1
```

- 若要更新到最新的预发布版本的 jQuery:

```
libman update jquery -pre
```

管理库缓存

`libman cache` 命令管理 LibMan 库缓存。`filesystem` 提供程序不使用库缓存。

摘要

```
libman cache clean [<PROVIDER>] [--verbosity]
libman cache list [--files] [--libraries] [--verbosity]
libman cache [-h|--help]
```

自变量

`PROVIDER`

仅用于 `clean` 命令。指定要清除缓存的提供程序。有效值包括：

- `cdnjs`

- `filesystem`
- `unpkg`

选项

有以下选项 `libman cache` 命令：

- `--files`

  列出缓存的文件的名称。

- `--libraries`

  列出的已缓存的库的名称。

- `-h|--help`

  显示帮助信息。

- `--verbosity <LEVEL>`

  设置输出的详细级别。替换为 `<LEVEL>` 使用以下值之一：

  - `quiet`
  - `normal`
  - `detailed`

示例

- 若要查看的每个提供程序的缓存库的名称，请使用以下命令之一：

  ```
  libman cache list
  ```

  ```
  libman cache list --libraries
  ```

  显示了类似下面的输出：

  ```
  Cache contents:
  ---------------
  unpkg:
      knockout
      react
      vue
  cdnjs:
      font-awesome
      jquery
      knockout
      lodash.js
      react
  ```

- 若要查看的每个提供程序的缓存的库文件的名称：

  ```
  libman cache list --files
  ```

  显示了类似下面的输出：

```
Cache contents:
---------------
unpkg:
    knockout:
        <list omitted for brevity>
    react:
        <list omitted for brevity>
    vue:
        <list omitted for brevity>
cdnjs:
    font-awesome
        metadata.json
    jquery
        metadata.json
        3.2.1\core.js
        3.2.1\jquery.js
        3.2.1\jquery.min.js
        3.2.1\jquery.min.map
        3.2.1\jquery.slim.js
        3.2.1\jquery.slim.min.js
        3.2.1\jquery.slim.min.map
        3.3.1\core.js
        3.3.1\jquery.js
        3.3.1\jquery.min.js
        3.3.1\jquery.min.map
        3.3.1\jquery.slim.js
        3.3.1\jquery.slim.min.js
        3.3.1\jquery.slim.min.map
    knockout
        metadata.json
        3.4.2\knockout-debug.js
        3.4.2\knockout-min.js
    lodash.js
        metadata.json
        4.17.10\lodash.js
        4.17.10\lodash.min.js
    react
        metadata.json
```

请注意，上面的输出显示了在 CDNJS 提供程序缓存版本 3.2.1 和 3.3.1，jQuery。

- 若要在清空 CDNJS 提供程序库缓存：

```
libman cache clean cdnjs
```

之后清空缓存的 CDNJS 提供程序 `libman cache list` 命令显示以下信息：

```
Cache contents:
---------------
unpkg:
    knockout
    react
    vue
cdnjs:
    (empty)
```

- 若要清空缓存的所有受支持的提供程序：

```
libman cache clean
```

之后清空所有提供程序缓存，`libman cache list` 命令显示以下信息：

```
Cache contents:
---------------
unpkg:
    (empty)
cdnjs:
    (empty)
```

其他资源

- 安装一个全局工具
- 在 Visual Studio 中的 ASP.NET Core 中使用 LibMan
- LibMan GitHub 存储库

在 Visual Studio 中的 ASP.NET Core 中使用 LibMan

2018/12/12 • • Edit Online

作者：Scott Addie

Visual Studio 在 ASP.NET Core 项目中内置了对 LibMan 的支持，包括：

- 支持在生成时配置和运行 LibMan 还原操作。
- 用于触发 LibMan 还原和清理操作的菜单项。
- 搜索对话框，查找库并将文件添加到项目中。
- 编辑对*libman.json*（LibMan 清单文件）的支持。—

查看或下载示例代码（如何下载）

系统必备

- Visual Studio 2017 版本 15.8 或更高版本，且具有**ASP.NET 和 Web 开发工作负载**

添加库文件

可通过两种不同的方式将库文件添加到 ASP.NET Core 项目中：

1. 使用"添加客户端库"对话框
2. 手动配置 LibMan 清单文件条目

使用"添加客户端库"对话框

请按照下列步骤以安装客户端库：

- 在中**解决方案资源管理器**，右键单击项目文件夹应在其中添加文件。选择**添加** > **客户端库**。**添加客户端库**此时将显示对话框：

- 选择从库提供程序**提供程序**下拉列表。CDNJS 是默认提供程序。
- 类型库名称中提取库文本框。IntelliSense 提供了一系列从提供的文本的库。
- 从智能感知列表中选择的库。请注意，库名称作为后缀 `@` 符号和已知的所选提供程序的最新稳定版本。
- 确定要包括的文件：

- 选择**包括所有库文件**单选按钮，以都包括所有库的文件。
- 选择**选择特定的文件**单选按钮，以包含库的文件的子集。选中的单选按钮后，启用文件选择器树。检查要下载的文件名称左侧的框。

- 指定用于存储中的文件的项目文件夹**目标位置**文本框。为建议，将每个库存储在单独的文件夹。

  建议**目标位置**文件夹根据从中启动对话框的位置：

  - 如果从项目根目录中启动：
    - *wwwroot/lib*如果，则使用*wwwroot*存在。
    - *lib*如果，则使用*wwwroot*不存在。
  - 如果从项目文件夹启动，则使用相应的文件夹名称。

  库名称作为后缀文件夹建议。Razor 页面项目中安装 jQuery 时下，表说明了文件夹的建议。

启动位置	建议的文件夹
项目根目录（如果*wwwroot*存在）	*wwwroot/lib/jquery /*
项目根目录（如果*wwwroot*不存在）	*lib/jquery /*
*页*项目文件夹中	*页/jquery /*

- 单击**安装**按钮以下载的文件，每个中配置*libman.json*。

- 审阅库**管理器的源输出安装**详细信息窗口。例如：

```
Restore operation started...
Restoring libraries for project LibManSample
Restoring library jquery@3.3.1... (LibManSample)
wwwroot/lib/jquery/jquery.min.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.min.map written to destination (LibManSample)
Restore operation completed
1 libraries restored in 2.32 seconds
```

手动配置 LibMan 清单文件条目

在 Visual Studio 中的所有 LibMan 操作都基于项目根 LibMan 清单的内容（*libman.json*）。您可以手动编辑*libman.json*若要配置的项目的库文件。Visual Studio 还原所有库文件一次*libman.json*保存。

若要打开*libman.json*进行编辑，存在以下选项：

- 双击*libman.json*中的文件**解决方案资源管理器**。
- 右键单击该项目中的**解决方案资源管理器**，然后选择**管理客户端库**。†
- 选择**管理客户端库**从 Visual Studio**项目**菜单。†

† 如果*libman.json*文件尚不存在项目根目录中，它将使用默认项目模板内容的创建。

Visual Studio 提供了丰富 JSON 编辑支持，例如颜色设置、格式设置、IntelliSense 和架构验证。在找到 LibMan 清单的 JSON 架构 http://json.schemastore.org/libman 。

使用以下清单文件，LibMan 检索每中定义的配置文件 `libraries` 属性。中定义的对象文字说明 `libraries` 后面：

- 一个子集jQuery版本 3.3.1 检索从 CDNJS 提供程序。在中定义的子集 `files` 属性—*jquery.min.js*，*只需要jquery.js*，以及*jquery.min.map*。将文件放置在项目的*wwwroot/lib/jquery*文件夹。
- 整个Bootstrap检索并放入版本 4.1.3 *wwwroot/lib/bootstrap*文件夹。对象文字 `provider` 属性重写 `defaultProvider` 属性值。LibMan 从 unpkg 提供程序中检索启动文件。
- 一个子集lodash 等监管主体在组织内已批准。*Lodash.js*并*lodash.min.js*从本地文件系统中检索文

件*c:\temp\lodash 等*。将文件复制到项目的*lodash 等wwwroot/lib/* 文件夹。

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.3.1",
      "files": [
        "jquery.min.js",
        "jquery.js",
        "jquery.min.map"
      ],
      "destination": "wwwroot/lib/jquery/"
    },
    {
      "provider": "unpkg",
      "library": "bootstrap@4.1.3",
      "destination": "wwwroot/lib/bootstrap/"
    },
    {
      "provider": "filesystem",
      "library": "C:\\temp\\lodash\\",
      "files": [
        "lodash.js",
        "lodash.min.js"
      ],
      "destination": "wwwroot/lib/lodash/"
    }
  ]
}
```

> **NOTE**
> LibMan 仅支持每个提供程序从每个库的一个版本。*Libman.json*文件未通过架构验证，如果它包含与给定的提供程序库同名的两个库。

库将文件还原

若要还原从 Visual Studio 中的库文件，必须有一个有效*libman.json*项目根目录中的文件。还原的文件放置在项目中指定每个库的位置。

可以通过两种方式的 ASP.NET Core 项目中还原库文件：

1. 在生成过程中还原文件
2. 手动还原文件

在生成过程中还原文件

LibMan 可以作为生成过程的一部分还原的已定义的库文件。默认情况下*还原上生成*禁用行为。

若要启用和测试还原上生成行为：

- 右键单击*libman.json*中**解决方案资源管理器**，然后选择**启用还原客户端上的库生成**从上下文菜单。
- 单击**是**按钮当系统提示安装 NuGet 包。Microsoft.Web.LibraryManager.Build NuGet 包添加到项目：

  ```
  <PackageReference Include="Microsoft.Web.LibraryManager.Build" Version="1.0.113" />
  ```

- 生成项目，以确认 LibMan 文件还原发生。`Microsoft.Web.LibraryManager.Build` 包注入项目的生成操作期间

运行 LibMan 的 MSBuild 目标。

- 审阅**构建**的源输出LibMan 活动日志窗口：

```
1>------ Build started: Project: LibManSample, Configuration: Debug Any CPU ------
1>
1>Restore operation started...
1>Restoring library jquery@3.3.1...
1>Restoring library bootstrap@4.1.3...
1>
1>2 libraries restored in 10.66 seconds
1>LibManSample -> C:\LibManSample\bin\Debug\netcoreapp2.1\LibManSample.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

启用还原上生成行为后，*libman.json*上下文菜单将显示**禁用还原客户端上的**库**生成**选项。选择此选项删除 `Microsoft.Web.LibraryManager.Build` 包从项目文件的引用。因此，客户端库不再存储在每个版本。

而不考虑还原上生成设置中，您可以手动还原在从任何时候*libman.json*上下文菜单。有关详细信息，请参阅手动还原文件。

手动还原文件

若要手动还原库文件：

- 对于解决方案中的所有项目：
  - 右键单击解决方案的名称**解决方案**资**源管理器**。
  - 选择还**原客**户**端**库选项。
- 对于特定的项目：
  - 右键单击*libman.json*中的文件**解决方案**资**源管理器**。
  - 选择还**原客**户**端**库选项。

尽管还原操作正在运行：

- Visual Studio 在状态栏上的任务状态中心 (TSC) 图标进行动画处理，并将读取*还原操作已启动*。单击该图标可打开列出已知的后台任务的工具提示。

- 消息将发送到状态栏并库**管理器**的源输出窗口。例如：

```
Restore operation started...
Restoring libraries for project LibManSample
Restoring library jquery@3.3.1... (LibManSample)
wwwroot/lib/jquery/jquery.min.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.min.map written to destination (LibManSample)
Restore operation completed
1 libraries restored in 2.32 seconds
```

删除库文件

若要执行*干净*操作，这将删除以前还原在 Visual Studio 中的库文件：

- 右键单击*libman.json*中的文件**解决方案**资**源管理器**。
- 选择**干净的客**户**端**库选项。

若要防止无意删除非库文件，清除操作不会删除整个目录。而只会删除包括在上一个还原的文件。

尽管运行清理操作：

- Visual Studio 在状态栏上的 TSC 图标进行动画处理，并将读取*启动客户端库操作*。单击该图标可打开列出已知

的后台任务的工具提示。

- 消息发送到状态栏并库**管理器**的源输出窗口。例如：

```
Clean libraries operation started...
Clean libraries operation completed
2 libraries were successfully deleted in 1.91 secs
```

清理操作只会从项目删除文件。在将来还原操作的更快地检索缓存中保留库文件。若要管理存储在本地计算机的缓存中的库文件，请使用LibMan CLI。

卸载库文件

若要卸载库文件：

- 打开*libman.json*。
- 定位插入符号置于相应 `libraries` 对象文字。
- 单击左边距中显示的灯泡图标，然后选择**卸载 <library_name > @<library_version >**：

或者，可以手动编辑和保存 LibMan 清单 (*libman.json*)。还原操作运行时保存该文件。不能再中定义的库文件*libman.json*从项目中删除。

更新库版本

若要检查有更新的库版本：

- 打开*libman.json*。
- 定位插入符号置于相应 `libraries` 对象文字。
- 单击灯泡图标显示在左边距中。将鼠标悬停检查**是否有更新**。

LibMan 检查安装的版本比新的库版本。会出现以下结果：

- 一个**找不到更新**如果已安装最新版本，则会显示消息。
- 最新稳定版本是显示如果还没有安装。

- 如果现已安装的版本比新预发布版本，会显示预发行版。

若要降级到较旧的库版本，手动编辑*libman.json*文件。保存文件时，LibMan还原操作：

- 从以前的版本中移除冗余的文件。
- 新版本中添加新的和更新文件。

其他资源

- ASP.NET Core 中使用 LibMan 命令行接口 (CLI)
- LibMan GitHub 存储库

管理 ASP.NET Core 中的 Bower 的客户端包

2018/11/14 • • Edit Online

通过Rick Anderson， Noel 大米，和Scott Addie

> **IMPORTANT**
>
> 而维护 Bower，其维护人员建议使用另一种解决方案。库管理器(简称 LibMan) 是 Visual Studio 的新客户端库获取工具 (Visual Studio 15.8 或更高版本)。有关详细信息，请参阅 通过 LibMan 在 ASP.NET Core 中获取客户端库 。Bower 在 Visual Studio 中受支持版本 15.5 通过。
>
> Yarn Webpack 使用是的其中一个受欢迎的替代方法迁移说明可用。

Bower调用其自身"web 包管理器"。在.NET 生态系统，它填补留下的 NuGet 的无法传送静态内容的文件。对于 ASP.NET Core 项目，这些静态文件，则所固有的客户端库，如jQuery和Bootstrap。对于.NET 库，您仍使用NuGet包管理器。

设置客户端的 ASP.NET Core 项目模板创建的新项目生成过程。jQuery并Bootstrap安装，并支持 Bower。

客户端的包中列出*bower.json*文件。ASP.NET Core 项目模板配置*bower.json* jQuery、jQuery 验证与 Bootstrap。

在本教程中，我们将添加对的支持Font Awesome。可以使用安装 bower 程序包**管理 Bower 程序包**UI 或手动*bower.json*文件。

通过管理 Bower 程序包 UI 安装

- 创建新的 ASP.NET Core Web 应用程序与**ASP.NET Core Web 应用程序 (.NET Core)** 模板。选择**Web 应用程序并无身份**验证。
- 右键单击解决方案资源管理器中的项目并选择**管理 Bower 程序包**(或者从主菜单项目 > **管理 Bower 程序包**).
- 在中**Bower:<项目名称>** 窗口中，单击"浏览"选项卡，并输入筛选在包列表 `font-awesome` 在搜索框中：

- 确认"保存更改为*bower.json*"复选框已选中。从下拉列表中选择一个版本，然后单击**安装**按钮。输出窗口将显示安装详细信息。

在 bower.json 的手动安装

打开*bower.json*文件，并将"字体 awesome"添加到依赖项。IntelliSense 显示可用的包。选择包后，显示可用的版本。下图是较旧和不匹配，请参阅。

使用 bower语义化版本控制来组织依赖项。语义版本控制，也称为 SemVer，标识包具有单独的编号方案<主要 >。<次要 >。<修补程序 >。IntelliSense 显示仅几个常用的选项，从而简化了语义化版本控制。智能感知列表 (在上面的示例 4.6.3) 中的顶级项被视为包的最新稳定版本。插入符号 (^) 符号匹配的最新的主版本和颚化符 (~) 与最新次要版本匹配。

保存*bower.json*文件。Visual Studio 会监视*bower.json*文件的更改。保存后，*bower 安装*执行命令。请参见输出窗口**Bower/npm**视图执行的确切命令。

打开 *.bowerrc*文件下*bower.json*。`directory`属性设置为*wwwroot/lib*指示位置 Bower 将安装的包资产。

```
{
 "directory": "wwwroot/lib"
}
```

可以使用解决方案资源管理器中的搜索框查找并显示字体令人惊叹的包。

打开*views/shared_Layout.cshtml*文件，并将字体出色的 CSS 文件添加到环境标记帮助程序为 `Development` 。从解决方案资源管理器拖放*字体 awesome.css*内 `<environment names="Development">` 元素。

```
<environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
    <link href="~/lib/font-awesome/css/font-awesome.css" rel="stylesheet" />
</environment>
```

在生产应用中添加*字体 awesome.min.css*环境标记帮助程序对于 `Staging,Production` 。

内容替换为*Views\Home\About.cshtml* Razor 文件使用以下标记：

```
@{
    ViewData["Title"] = "About";
}

<div class="list-group">
    <a class="list-group-item" href="#"><i class="fa fa-home fa-fw" aria-hidden="true"></i>  Home</a>
    <a class="list-group-item" href="#"><i class="fa fa-book fa-fw" aria-hidden="true"></i>  Library</a>
    <a class="list-group-item" href="#"><i class="fa fa-pencil fa-fw" aria-hidden="true"></i> 
Applications</a>
    <a class="list-group-item" href="#"><i class="fa fa-cog fa-fw" aria-hidden="true"></i>  Settings</a>
</div>
```

运行应用并导航到关于视图以确保出色字体包正常运行。

浏览客户端的生成过程

大多数 ASP.NET Core 项目模板已配置为使用 Bower。此下一个演练中创建空的 ASP.NET Core 项目启动，并手动添加每个部分，以便您可以如何在项目中使用 Bower 获取获得感觉。您所见到的项目结构和运行时输出会和每个配置更改会发生什么情况。

若要与 Bower 一起使用客户端的生成过程的常规步骤如下：

- 定义项目中使用的包。
- 从您的 web 页面的引用包。

定义包

列表中的包后*bower.json*文件，Visual Studio 将下载它们。下面的示例使用 Bower 加载 jQuery 和到 Bootstrap *wwwroot*文件夹。

- 创建新的 ASP.NET Core Web 应用程序与**ASP.NET Core Web 应用程序 (.NET Core)** 模板。选择**空**项目模板，然后单击**确定**。
- 在解决方案资源管理器中右键单击该项目 >**添加新**项，然后选择**Bower 配置文件**。注意：一个 *.bowerrc*文件也添加。
- 打开*bower.json*，并添加 jquery 和启动到 `dependencies` 部分。得到*bower.json*文件看起来如下例所示。版本会随时间变化，可能与下面的图像不匹配。

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "jquery": "3.1.1",
    "bootstrap": "3.3.7"
  }
}
```

- 保存*bower.json*文件。

  验证该项目包括*bootstrap*并*jQuery*中的目录*wwwroot/lib*。使用 bower *.bowerrc*文件以安装中的资产*wwwroot/lib*。

  注意:"管理 Bower 程序包"UI 提供手动文件编辑。

启用静态文件

- 添加 `Microsoft.AspNetCore.StaticFiles` 到项目的 NuGet 包。
- 启用静态文件提供与静态文件中间件。添加对的调用UseStaticFiles到 `Configure` 方法的 `Startup` 。

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseStaticFiles();

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    }
}
```

引用包

在本部分中，将创建一个 HTML 页，以验证它可以访问已部署的包。

- 添加一个名为的新 HTML 页*Index.html*到*wwwroot*文件夹。注意：您必须将添加到 HTML 文件*wwwroot*文件夹。默认情况下，静态内容不能提供外部*wwwroot*。请参阅静态文件有关详细信息。

  内容替换为*Index.html*使用以下标记：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Bower Example</title>
    <link href="lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body>
    <div class="jumbotron">
        <h1>Using the jumbotron style</h1>
        <p>
            <a class="btn btn-primary btn-lg" role="button">Stateful button</a>
        </p>
    </div>
    <script src="lib/jquery/dist/jquery.js"></script>
    <script src="lib/bootstrap/dist/js/bootstrap.js"></script>
    <script>
        $(".btn").click(function () {
            $(this).text('loading')
                .delay(1000)
                .queue(function () {
                    $(this).text('reset');
                    $(this).dequeue();
                });
        });
    </script>
</body>

</html>
```

- 运行应用并导航到 `http://localhost:<port>/Index.html` 。或者，使用*Index.html*打开，按 `Ctrl+Shift+W` 。验证应用 jumbotron 样式、jQuery 代码进行响应时单击该按钮，并且启动按钮更改状态。

小于，Sass 和 ASP.NET Core 中出色的字体

2018/7/17 • • Edit Online

作者：Steve Smith

Web 应用程序的用户有越来越高的期望时设置的样式和总体体验。现代 web 应用程序频繁地利用丰富的工具和框架用于定义和管理其外观和感觉以一致的方式。框架喜欢Bootstrap可以地长定义一组通用的样式和网站的布局选项。但是，大多数重要站点还受益于能够有效地定义和维护样式和级联样式表 (CSS) 文件，以及能够轻松访问帮助使站点的接口更直观的非图像图标。就是在此处的支持语言和工具较少和Sass，库等，以及将字体出色，进入。

CSS 预处理器语言

为了改进的体验的使用基础的语言中，编译成其它语言的语言被称为预处理器。有两个常用的预处理器针对 CSS：不太和 Sass。这些预处理器将功能添加到 CSS，如对变量和嵌套的规则，以改进可维护性的大型、复杂的样式表的支持。作为一种语言的 CSS 是非常基本，缺少那样简单变量，即使对于支持，这往往会重复和臃肿使 CSS 文件。添加通过预处理器的实际编程语言功能可以帮助减少重复，提供更好地组织样式规则。Visual Studio 提供这两个较低的内置支持和 Sass，以及可以进一步提高开发体验，使用这些语言时的扩展。

作为预处理器可以如何改进可读性和可维护性的样式信息的快速示例，请考虑以下 CSS:

```
.header {
    color: black;
    font-weight: bold;
    font-size: 18px;
    font-family: Helvetica, Arial, sans-serif;
}

.small-header {
    color: black;
    font-weight: bold;
    font-size: 14px;
    font-family: Helvetica, Arial, sans-serif;
}
```

使用更少，这可以重写以消除所有重复，使用*mixin* (这样命名是因为它允许你"中混合使用"从一个类或到另一个规则集的属性):

```
.header {
    color: black;
    font-weight: bold;
    font-size: 18px;
    font-family: Helvetica, Arial, sans-serif;
}

.small-header {
    .header;
    font-size: 14px;
}
```

Less

使用 Node.js 不太 CSS 预处理器运行。若要安装小于，使用 Node 程序包管理器 (npm) 从命令提示符 (-g 意味着"全局"):

```
npm install -g less
```

如果你使用 Visual Studio，你可以开始使用小于通过将一个或多个更少文件添加到你的项目，然后配置 Gulp（或 Grunt）在编译时处理它们。添加*样式*到你的项目的文件夹，然后添加新的名为的文件小于*main.less*到此文件夹。

添加后，您的文件夹结构应如下所示：

现在你可以添加到文件中，这将编译到 CSS 并部署到的 wwwroot 文件夹 Gulp 的一些基本的样式。

修改*main.less*为包括以下内容，这将从一种基颜色创建一个简单的调色板。

```
@base: #663333;
@background: spin(@base, 180);
@lighter: lighten(spin(@base, 5), 10%);
@lighter2: lighten(spin(@base, 10), 20%);
@darker: darken(spin(@base, -5), 10%);
@darker2: darken(spin(@base, -10), 20%);

body {
    background-color:@background;
}
.baseColor  {color:@base}
.bgLight    {color:@lighter}
.bgLight2   {color:@lighter2}
.bgDark     {color:@darker}
.bgDark2    {color:@darker2}
```

`@base` 和其他@-prefixed项是变量。每个表示一种颜色。除 `@base`，设置它们并使用颜色函数：加亮、变暗，和旋转。淡化和加深执行几乎你将预期;数值调节钮调整颜色色调的大量度（围绕颜色盘中）。较少处理器是足够智能，可忽略不使用的变量，因此若要展示了这些变量的工作原理，我们需要某个位置使用它们。类 `.baseColor`，等将演示每个生成的 CSS 文件中的变量的计算的值。

入门

创建**npm 配置文件**(*package.json*) 在你的项目文件夹并编辑它以引用 `gulp` 和 `gulp-less`：

```
{
  "version": "1.0.0",
  "name": "asp.net",
  "private": true,
  "devDependencies": {
    "gulp": "3.9.1",
    "gulp-less": "3.3.0"
  }
}
```

在你的项目文件夹，或在 Visual Studio 中安装的依赖关系，请在命令提示符下**解决方案资源管理器**(**依赖项 > npm** > 还**原程序包**)。

```
npm install
```

在项目文件夹中，创建**Gulp 配置文件**(*gulpfile.js*) 以定义自动的过程。在更少，表示的文件及要运行更少的任务的顶部添加一个变量：

```
var gulp = require("gulp"),
  fs = require("fs"),
  less = require("gulp-less");

gulp.task("less", function () {
  return gulp.src('Styles/main.less')
    .pipe(less())
    .pipe(gulp.dest('wwwroot/css'));
});
```

打开任务运行程序资源管理器(视图 > **其他 Windows** > 任务运行程序资源管理器)。在任务之间，你应看到一个名为的新任务 `less` 。你可能必须刷新窗口。

运行 `less` 任务，并看到类似于此处所示的输出：

*Wwwroot/css*文件夹现在包含一个新的文件，*main.css*:

打开*main.css*，你将看到与下面类似：

```
body {
    background-color: #336666;
}
.baseColor {
    color: #663333;
}
.bgLight {
    color: #884a44;
}
.bgLight2 {
    color: #aa6355;
}
.bgDark {
    color: #442225;
}
.bgDark2 {
    color: #221114;
}
```

添加到一个简单的 HTML 页面*wwwroot*文件夹，然后引用*main.css*若要查看操作中的调色板。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <link href="css/main.css" rel="stylesheet" />
    <title></title>
</head>
<body>
    <div>
        <div class="baseColor">BaseColor</div>
        <div class="bgLight">Light</div>
        <div class="bgLight2">Light2</div>
        <div class="bgDark">Dark</div>
        <div class="bgDark2">Dark2</div>
    </div>
</body>
</html>
```

你可以看到上旋转 180 度 `@base` 用于生成 `@background` 导致相反颜色的颜色盘 `@base`：

小于还提供对嵌套的规则以及嵌套的媒体查询的支持。例如，像菜单可能会导致详细 CSS 规则的定义嵌套层次结构像这样：

```
nav {
    height: 40px;
    width: 100%;
}
nav li {
    height: 38px;
    width: 100px;
}
nav li a:link {
    color: #000;
    text-decoration: none;
}
nav li a:visited {
    text-decoration: none;
    color: #CC3333;
}
nav li a:hover {
    text-decoration: underline;
    font-weight: bold;
}
nav li a:active {
    text-decoration: underline;
}
```

理想情况下的所有相关的样式规则将被放在一起在 CSS 文件中，但在实践中没有任何强制实施此规则约定和可能块注释除外。

定义使用小于这些相同的规则类似如下所示：

```
nav {
    height: 40px;
    width: 100%;
    li {
        height: 38px;
        width: 100px;
        a {
            color: #000;
            &:link { text-decoration:none}
            &:visited { color: #CC3333; text-decoration:none}
            &:hover { text-decoration:underline; font-weight:bold}
            &:active {text-decoration:underline}
        }
    }
}
```

请注意，在此情况下，所有的从属元素 `nav` 包含在其作用域内。不再父元素的任何重复（`nav`， `li`， `a`），并且总的行计数也已删除（尽管某些就是将值放在第二个示例的同一行上的结果）。它可以是非常有帮助，组织，若要查看的所有规则的给定的用户界面元素在显式限定范围内，在这种情况下设置从文件的其余部分由大括号。

`&` 语法是较少的选择器功能，与（&）表示当前的选择器父级。这样，在 {…} 块中，`&` 表示 `a` 标记，因此 `&:link` 等效于 `a:link`。

创建响应式设计非常有用的媒体查询还可能影响很大程度重复和 CSS 中的复杂性。小于允许媒体查询嵌套在类，以便整个类定义不需要重复内不同顶级 `@media` 元素。例如，下面是响应式菜单的 CSS：

```
.navigation {
    margin-top: 30%;
    width: 100%;
}
@media screen and (min-width: 40em) {
    .navigation {
        margin: 0;
    }
}
@media screen and (min-width: 62em) {
    .navigation {
        width: 960px;
        margin: 0;
    }
}
```

这可以更好地定义以秒为:

```
.navigation {
    margin-top: 30%;
    width: 100%;
    @media screen and (min-width: 40em) {
        margin: 0;
    }
    @media screen and (min-width: 62em) {
        width: 960px;
        margin: 0;
    }
}
```

小于我们已经看到的另一个功能是其支持的数学运算，允许从预定义的变量构造的样式特性。这使得更新相关的样式容易得多，因为可以修改基变量和所有依赖值自动更改。

CSS 文件，尤其是对于大型站点（和尤其是如果正在使用媒体查询），就会变得很大随着时间推移，从而使用它们难以操作。Less 文件可以单独定义，然后一起使用，取出 `@import` 指令。小于还可导入单个 CSS 文件，同样，如果需要。

*Mixins*可以接受参数，并且小于 mixin 防护，提供声明性方式定义某些 mixins 生效的形式支持条件逻辑。Mixin 临界条件的一个常见用途是调整颜色如何细基于或深色的源颜色是。Mixin 防护给定 mixin 接受颜色的参数，用于修改基于该颜色 mixin:

```
.box (@color) when (lightness(@color) >= 50%) {
    background-color: #000;
}
.box (@color) when (lightness(@color) < 50%) {
    background-color: #FFF;
}
.box (@color) {
    color: @color;
}

.feature {
    .box (@base);
}
```

给定我们当前 `@base` 值 `#663333`，小于此脚本将生成以下 CSS:

```
.feature {
    background-color: #FFF;
    color: #663333;
}
```

小于提供许多其他功能，但这应为你提供此强大的一些思路预处理语言。

Sass

Sass 类似于较少，是为许多相同功能，但使用略有不同的语法提供支持。它使用 Ruby，而不是 JavaScript 中，生成，因此不同的安装要求。原始 Sass 语言未使用大括号或分号，但改为定义使用空白和缩进的作用域。在 Sass 3 版本中，引入了新的语法，**SCSS** ("Sassy CSS")。SCSS 是类似于 CSS，可忽略缩进级别和空格，并改为使用分号和大括号。

若要安装 Sass，通常你将首先安装 Ruby（预安装在 macOS 上），，然后运行：

```
gem install sass
```

但是，如果要运行 Visual Studio，你可以开始使用 Sass 在很大程度的相同方式就像对待小于。打开*package.json*并添加到"gulp sass"包 `devDependencies`：

```
"devDependencies": {
  "gulp": "3.9.1",
  "gulp-less": "3.3.0",
  "gulp-sass": "3.1.0"
}
```

接下来，修改*gulpfile.js*添加 sass 变量和编译 Sass 文件并将结果放的 wwwroot 文件夹中的任务：

```
var gulp = require("gulp"),
  fs = require("fs"),
  less = require("gulp-less"),
  sass = require("gulp-sass");

// other content removed

gulp.task("sass", function () {
  return gulp.src('Styles/main2.scss')
    .pipe(sass())
    .pipe(gulp.dest('wwwroot/css'));
});
```

现在，您可以将 Sass 文件*main2.scss*到*样式*项目的根目录中的文件夹：

打开*main2.scss*，添加以下内容：

```
$base: #CC0000;
body {
    background-color: $base;
}
```

保存你所有文件。现在刷新任务**运行程序**资**源管理器**，你看到`sass`任务。运行它，并查找*/wwwroot/css*文件夹。现在有了*main2.css*文件，使用以下内容：

```
body {
    background-color: #CC0000;
}
```

Sass 支持嵌套大体相同时，小于存在，提供类似的好处。文件可以由函数拆分和包含使用`@import`指令：

```
@import 'anotherfile';
```

Sass 支持 mixins 同样，使用`@mixin`关键字定义它们和`@include`以便将它们，包含从如此示例所示sass lang.com:

```
@mixin border-radius($radius) {
    -webkit-border-radius: $radius;
    -moz-border-radius: $radius;
    -ms-border-radius: $radius;
    border-radius: $radius;
}

.box { @include border-radius(10px); }
```

Sass 除了 mixins，它还支持继承，的概念允许一个类，以扩展另一个。它是从概念上讲类似于 mixin，但在更少的 CSS 代码中的结果。它通过`@extend`关键字。若要尝试 mixins，将以下代码添加到你*main2.scss*文件：

```
@mixin alert {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
}

.success {
    @include alert;
    border-color: green;
}

.error {
    @include alert;
    color: red;
    border-color: red;
    font-weight:bold;
}
```

检查在输出*main2.css*运行之后 `sass` 任务中**任务运行程序资源管理器**:

```
.success {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
    border-color: green;
 }

.error {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
    color: red;
    border-color: red;
    font-weight: bold;
}
```

请注意所有警报 mixin 的常见属性在每个类中有重复。Mixin 好地帮助消除重复在开发时，但它仍包含中的重复，从而导致大于必要 CSS 文件-一个潜在的性能问题很多要创建 CSS。

现在将与警报 mixin `.alert` 类，并更改 `@include` 到 `@extend`（记住来扩展 `.alert`，而不 `alert`）:

```
.alert {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
}

.success {
    @extend .alert;
    border-color: green;
}

.error {
    @extend .alert;
    color: red;
    border-color: red;
    font-weight:bold;
}
```

运行 Sass 次，并检查生成的 CSS:

```
.alert, .success, .error {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
}

.success {
    border-color: green;
}

.error {
    color: red;
    border-color: red;
    font-weight: bold;
}
```

现在仅为根据需要多次定义属性，并更好地生成 CSS。

Sass 还包括函数和条件逻辑操作，类似于小于。事实上，这两种语言的功能是非常相似。

较低或 Sass？

仍没有关于是否通常更好的做法将占用更少或 Sass 没有共识（或甚至是否首选原始 Sass 中的较新 SCSS 语法）。可能的最重要的决定是**使用这些工具之一**，而不是只使用手工编码 CSS 文件。一旦你所做的决策，这两个不太和 Sass 是不错的选择。

Font Awesome

除了 CSS 预处理器样式现代 web 应用程序的另一个很好的办法是字体出色。字体 Awesome 是一个工具包，提供 500 多个可以自由地使用 web 应用程序中的可缩放的向量图标。它最初设计成可使用 Bootstrap，但它并不依赖于该框架或在任何 JavaScript 库。

若要开始使用字体出色的最简单方法是添加对它，使用其公用内容交付网络 (CDN) 位置的引用：

```
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/font-awesome/4.3.0/css/font-awesome.min.css">
```

你可以还将其添加到你的 Visual Studio 项目通过将它添加到"依赖关系"中*bower.json*:

```
{
  "name": "ASP.NET",
  "private": true,
  "dependencies": {
    "bootstrap": "3.0.0",
    "jquery": "1.10.2",
    "jquery-validation": "1.11.1",
    "jquery-validation-unobtrusive": "3.2.2",
    "hammer.js": "2.0.4",
    "bootstrap-touch-carousel": "0.8.0",
    "Font-Awesome": "4.3.0"
  }
}
```

指向字体出色页上的引用后，你可以将图标添加到你的应用程序通过应用字体出色的类，通常前缀为"fa-"，为嵌入式 HTML 元素 (如 `<span>` 或 `<i>`)。例如，你可以将图标添加到简单列表和菜单使用如下代码：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
    <link href="lib/font-awesome/css/font-awesome.css" rel="stylesheet" />
</head>
<body>
    <ul class="fa-ul">
        <li><i class="fa fa-li fa-home"></i> Home</li>
        <li><i class="fa fa-li fa-cog"></i> Settings</li>
    </ul>
</body>
</html>
```

这将生成浏览器中的以下-请注意每个项旁边的图标：

你可以查看可用的图标的完整列表：

http://fontawesome.io/icons/

总结

现代 web 应用程序越来越要求清晰、直观，编写和更易于使用的各种设备从的响应速度快、流体设计。管理实现这些目标所需的 CSS 样式表的复杂性最好的做法是不太使用预处理器类似或 Sass。此外，如字体出色的工具包快速提供的已知图标添加到文本导航菜单和按钮，提高整体用户体验的应用程序。

捆绑和缩小在 ASP.NET Core 中的静态资产

2018/11/22 • • Edit Online

通过Scott Addie和David 松树

本文介绍了应用绑定和缩减，包括如何使用 ASP.NET Core web apps 使用这些功能的好处。

捆绑和缩小是什么

捆绑和缩小可以应用的 web 应用中的两个明显的性能优化。一起使用时，捆绑和缩小通过提高性能降低的服务器请求数和减小请求的静态资产的大小。

捆绑和缩小主要提高第一个页面请求加载时间。一旦请求 web 页，在浏览器缓存静态资产（JavaScript、CSS 和图像）。因此，绑定和缩减不提高性能时请求同一页上或请求同一资产相同站点上的页。如果过期标头未正确设置对资产并没有使用绑定和缩减，如果浏览器的新鲜度试探法将标记资产过时几天后。此外，在浏览器为每个资产需要验证请求。在这种情况下，绑定和缩减提供性能改进甚至后第一个页面请求。

捆绑

捆绑将多个文件合并到单个文件。绑定可减少所需呈现 web 资产，例如网页的服务器请求的数量。专门为 CSS、JavaScript 等，可以创建任意数量的单独的捆绑包。更少的文件意味着较少的 HTTP 请求从浏览器到服务器或提供你的应用程序的服务。这会改进了第一个页面加载性能。

缩小

缩减而无需更改功能从代码中删除不必要的字符。结果是很大大小减少请求的资产（如 CSS、图像和 JavaScript 文件）。缩减的常见副作用包括缩短为一个字符的变量名称和删除注释和不必要的空格。

请考虑以下的 JavaScript 函数：

```
AddAltToImg = function (imageTagAndImageID, imageContext) {
    ///<signature>
    ///<summary> Adds an alt tab to the image
    // </summary>
    //<param name="imgElement" type="String">The image selector.</param>
    //<param name="ContextForImage" type="String">The image context.</param>
    ///</signature>
    var imageElement = $(imageTagAndImageID, imageContext);
    imageElement.attr('alt', imageElement.attr('id').replace(/ID/, ''));
}
```

缩小简化为以下函数：

```
AddAltToImg=function(t,a){var r=$(t,a);r.attr("alt",r.attr("id").replace(/ID/,""))};
```

除了删除注释和不必要的空格，以下参数和变量名称已重命名，如下所示：

原始	重命名
`imageTagAndImageID`	`t`
`imageContext`	`a`

原始	重命名
`imageElement`	`r`

捆绑和缩减的影响

下表概述了单独加载资产与使用绑定和缩减之间的差异：

操作	B/M	而无需 B/M	更改
文件请求	7	18	157%
传输的 KB	156	264.68	70%
加载时间（毫秒）	885	2360	167%

浏览器是相当冗长关于 HTTP 请求标头。总字节数发送指标时将看到显著降低。加载时间显示了重大改进，但此示例在本地运行。与资产使用捆绑和缩小通过网络传输时实现更高的性能提升。

选择捆绑和缩减策略

MVC 和 Razor 页面项目模板提供开箱解决方案捆绑和缩小包含 JSON 配置文件。第三方工具，如Gulp并Grunt任务运行程序中，完成相同的任务具有更多的复杂性。第三方工具是理想的选择时开发工作流需要处理超出捆绑和缩小—linting 和映像优化等。通过使用设计时绑定和缩减，在应用程序的部署前创建缩小化的文件。捆绑和缩小在部署前提供减少的服务器负载的优点。但是，务必要识别该设计时捆绑和缩小会增大复杂性，生成和仅适用于静态文件。

配置捆绑和缩减

MVC 和 Razor 页面项目模板提供了在 ASP.NET Core 2.0 或更早*bundleconfig.json*定义每个捆绑包的选项的配置文件：

在 ASP.NET Core 2.1 或更高版本，添加一个新的 JSON 文件，名为*bundleconfig.json*，与 MVC 或 Razor 页面项目根目录。作为起始点，在该文件中包含以下 JSON:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

*Bundleconfig.json*文件定义每个捆绑包的选项。在上述示例中，一个捆绑包配置为定义了自定义 JavaScript

(*wwwroot/js/site.js*) 和样式表 (*wwwroot/css/site.css*) 文件。

配置选项包括：

- `outputFileName`：要输出的捆绑包文件的名称。可以包含中的相对路径*bundleconfig.json*文件。**必填**
- `inputFiles`：要捆绑在一起的文件的数组。这些是配置文件的相对路径。**可选**，* 空值会导致空的输出文件。通配支持模式。
- `minify`：输出类型缩减选项。**可选**，*默认值-* `minify: { enabled: true }`
  - 每个输出文件类型提供了配置选项。
    - CSS 缩小器
    - JavaScript 缩减程序
    - HTML 缩小器
- `includeInProject`：指示是否将生成的文件添加到项目文件的标记。**可选**，*默认值-false*
- `sourceMap`：指示是否生成将捆绑的文件的源映射的标记。**可选**，*默认值-false*
- `sourceMapRootPath`：用于存储生成的源映射文件的根路径。

生成时执行的捆绑和缩小

BuildBundlerMinifier NuGet 程序包可执行的捆绑和缩小在生成时启用。包将注入MSBuild 目标其在生成和清理的时间运行。*Bundleconfig.json*文件分析的生成过程以生成基于定义的配置的输出文件。

> **NOTE**
> BuildBundlerMinifier 属于 Microsoft 为其提供任何支持的 GitHub 上的社区驱动项目。问题应将归档此处。

- Visual Studio
- .NET Core CLI

添加*BuildBundlerMinifier*包到你的项目。

生成项目。将显示以下输出窗口中：

```
1>------ Build started: Project: BuildBundlerMinifierApp, Configuration: Debug Any CPU ------
1>
1>Bundler: Begin processing bundleconfig.json
1>  Minified wwwroot/css/site.min.css
1>  Minified wwwroot/js/site.min.js
1>Bundler: Done processing bundleconfig.json
1>BuildBundlerMinifierApp -> C:\BuildBundlerMinifierApp\bin\Debug\netcoreapp2.0\BuildBundlerMinifierApp.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

清除该项目。将显示以下输出窗口中：

```
1>------ Clean started: Project: BuildBundlerMinifierApp, Configuration: Debug Any CPU ------
1>
1>Bundler: Cleaning output from bundleconfig.json
1>Bundler: Done cleaning output file from bundleconfig.json
========== Clean: 1 succeeded, 0 failed, 0 skipped ==========
```

捆绑和缩小的即席执行

很可能在某些特殊情况，运行绑定和缩减任务，而无需生成项目。添加BundlerMinifier.Core到你的项目的 NuGet 包：

```
<DotNetCliToolReference Include="BundlerMinifier.Core" Version="2.6.362" />
```

> **NOTE**
>
> BundlerMinifier.Core 属于 Microsoft 为其提供任何支持的 GitHub 上的社区驱动项目。问题应将归档此处。

此包扩展以包括.NET Core CLI *dotnet 捆绑*工具。在包管理器控制台 (PMC) 窗口或命令行界面中，可以执行以下命令：

```
dotnet bundle
```

> **IMPORTANT**
>
> NuGet 包管理器将依赖项添加到 *.csproj 文件作为 `<PackageReference />` 节点。`dotnet bundle` 命令注册.NET Core CLI 时，才 `<DotNetCliToolReference />` 使用节点。相应地修改 *.csproj 文件。

将文件添加到工作流

在其中一个示例，请考虑额外*custom.css*与下面类似的添加文件：

```
.about, [role=main], [role=complementary] {
    margin-top: 60px;
}

footer {
    margin-top: 10px;
}
```

若要缩小*custom.css*并将其与捆绑*site.css*到*site.min.css*文件中，添加的相对路径*bundleconfig.json*:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css",
      "wwwroot/css/custom.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

NOTE

或者，可使用以下通配模式：

```
"inputFiles": ["wwwroot/**/*(*.css|!(*.min.css))"]
```

此通配模式匹配所有 CSS 文件，并不包括缩小的文件模式。

生成应用程序。打开*site.min.css*，并注意的内容*custom.css*追加到文件末尾。

基于环境的捆绑和缩减

作为最佳做法，应在生产环境中使用你的应用的捆绑和缩小化文件。在开发期间，为方便调试的应用程序将使原始文件。

指定要使用包括在页面中的文件环境标记帮助程序在视图中。环境标记帮助程序中特定于运行时只呈现其内容环境。

以下 `environment` 标记将转换的未处理的 CSS 文件在运行时 `Development` 环境：

```
<environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

```
<environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

以下 `environment` 标记呈现的捆绑和缩小 CSS 文件，而不在环境中运行时 `Development`。例如，在运行 `Production` 或 `Staging` 触发这些样式表的呈现：

```
<environment exclude="Development">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
          asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

```
<environment names="Staging,Production">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
          asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

使用 Gulp 从 bundleconfig.json

有些情况下在其中应用的捆绑和缩小工作流需要进行其他处理。示例包括图像优化、清除缓存，和 CDN 资产处理。若要满足这些要求，可以将转换使用 Gulp 的捆绑和缩小工作流。

使用捆绑程序和缩小器扩展

Visual Studio捆绑程序和缩小器扩展处理到 Gulp 的转换。

> **NOTE**
>
> 捆绑程序和缩小器扩展属于 Microsoft 为其提供任何支持的 GitHub 上的社区驱动项目。问题应将归档此处。

右键单击*bundleconfig.json*文件在解决方案资源管理器中，然后选择**捆绑程序和缩小器** > 转换 **Gulp...**:

*Gulpfile.js*并*package.json*文件添加到项目。支持npm中列出的包*package.json*文件的 `devDependencies` 部分安装。

若要安装 Gulp CLI 作为全局依赖项在 PMC 窗口中运行以下命令：

```
npm i -g gulp-cli
```

*Gulpfile.js*文件读取*bundleconfig.json*输入、输出和设置的文件。

```
'use strict';

var gulp = require('gulp'),
    concat = require('gulp-concat'),
    cssmin = require('gulp-cssmin'),
    htmlmin = require('gulp-htmlmin'),
    uglify = require('gulp-uglify'),
    merge = require('merge-stream'),
    del = require('del'),
    bundleconfig = require('./bundleconfig.json');

// Code omitted for brevity
```

手动转换

如果 Visual Studio 和/或捆绑程序和缩小器扩展插件不可用，手动转换。

添加*package.json*文件中的，使用以下 `devDependencies`，到项目根目录：

```
"devDependencies": {
  "del": "^3.0.0",
  "gulp": "^4.0.0",
  "gulp-concat": "^2.6.1",
  "gulp-cssmin": "^0.2.0",
  "gulp-htmlmin": "^3.0.0",
  "gulp-uglify": "^3.0.0",
  "merge-stream": "^1.0.1"
}
```

通过与同一级别中运行以下命令安装依赖项*package.json*:

```
npm i
```

为全局的依赖项安装 Gulp CLI:

```
npm i -g gulp-cli
```

复制*gulpfile.js*到项目根文件如下:

```
'use strict';

var gulp = require('gulp'),
    concat = require('gulp-concat'),
    cssmin = require('gulp-cssmin'),
    htmlmin = require('gulp-htmlmin'),
    uglify = require('gulp-uglify'),
    merge = require('merge-stream'),
    del = require('del'),
    bundleconfig = require('./bundleconfig.json');

const regex = {
    css: /\.css$/,
    html: /\.(html|htm)$/,
    js: /\.js$/
};

gulp.task('min:js',
    () => merge(getBundles(regex.js).map(bundle => {
        return gulp.src(bundle.inputFiles, { base: '.' })
            .pipe(concat(bundle.outputFileName))
            .pipe(uglify())
            .pipe(gulp.dest('.'));
    })));

gulp.task('min:css',
    () => merge(getBundles(regex.css).map(bundle => {
        return gulp.src(bundle.inputFiles, { base: '.' })
            .pipe(concat(bundle.outputFileName))
            .pipe(cssmin())
            .pipe(gulp.dest('.'));
    })));

gulp.task('min:html',
    () => merge(getBundles(regex.html).map(bundle => {
        return gulp.src(bundle.inputFiles, { base: '.' })
            .pipe(concat(bundle.outputFileName))
            .pipe(htmlmin({ collapseWhitespace: true, minifyCSS: true, minifyJS: true }))
            .pipe(gulp.dest('.'));
    })));

gulp.task('min', gulp.series(['min:js', 'min:css', 'min:html']));

gulp.task('clean', () => {
    return del(bundleconfig.map(bundle => bundle.outputFileName));
});

gulp.task('watch', () => {
    getBundles(regex.js).forEach(
        bundle => gulp.watch(bundle.inputFiles, gulp.series(["min:js"])));

    getBundles(regex.css).forEach(
        bundle => gulp.watch(bundle.inputFiles, gulp.series(["min:css"])));

    getBundles(regex.html).forEach(
        bundle => gulp.watch(bundle.inputFiles, gulp.series(['min:html'])));
});

const getBundles = (regexPattern) => {
    return bundleconfig.filter(bundle => {
        return regexPattern.test(bundle.outputFileName);
    });
};

gulp.task('default', gulp.series(['min']));
```

运行 Gulp 任务

若要触发 Gulp 缩小任务之前在 Visual Studio 中生成项目，添加以下MSBuild 目标到 *.csproj 文件：

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="gulp min" />
</Target>
```

在此示例中定义的任何任务 `MyPreCompileTarget` 目标运行之前在预定义 `Build` 目标。在 Visual Studio 输出窗口中显示类似于以下输出：

```
1>------ Build started: Project: BuildBundlerMinifierApp, Configuration: Debug Any CPU ------
1>BuildBundlerMinifierApp -> C:\BuildBundlerMinifierApp\bin\Debug\netcoreapp2.0\BuildBundlerMinifierApp.dll
1>[14:17:49] Using gulpfile C:\BuildBundlerMinifierApp\gulpfile.js
1>[14:17:49] Starting 'min:js'...
1>[14:17:49] Starting 'min:css'...
1>[14:17:49] Starting 'min:html'...
1>[14:17:49] Finished 'min:js' after 83 ms
1>[14:17:49] Finished 'min:css' after 88 ms
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

或者，可能会使用 Visual Studio 的任务运行程序资源管理器来将 Gulp 任务绑定到特定的 Visual Studio 事件。请参阅正在运行的默认任务有关执行此操作的说明。

其他资源

- 使用 Gulp
- 使用 Grunt
- 使用多个环境
- 标记帮助程序

在 ASP.NET Core 中的浏览器链接

2018/8/22 • • Edit Online

通过Nicolò Carandini， Mike Wasson，和Tom Dykstra

浏览器链接是一项功能在 Visual Studio 中创建开发环境和一个或多个 web 浏览器之间的通信通道。可以使用浏览器链接以刷新 web 应用程序在多个浏览器中的，这对于跨浏览器测试很有用。

浏览器链接安装程序

转换为 ASP.NET Core 2.1，并转换为 ASP.NET Core 2.0 项目时Microsoft.AspNetCore.App 元包，安装Microsoft.VisualStudio.Web.BrowserLink包浏览器链接功能。ASP.NET Core 2.1 项目模板使用 `Microsoft.AspNetCore.App` 默认情况下的元包。

ASP.NET Core 2.0 **Web 应用程序**，**空**，并**Web API**项目模板使用Microsoft.AspNetCore.All 元包其中包含的包引用Microsoft.VisualStudio.Web.BrowserLink。因此，使用 `Microsoft.AspNetCore.All` 元包不需要任何进一步的操作，以使浏览器链接可供使用。

ASP.NET Core 1.x **Web 应用程序**项目模板具有的包引用Microsoft.VisualStudio.Web.BrowserLink包。**空**或**Web API**模板项目需要您添加到包引用 `Microsoft.VisualStudio.Web.BrowserLink` 。

由于这是一项 Visual Studio 功能，最简单的方法添加到包**空**或**Web API**模板项目是打开**程序包管理器控制台**(视图 > **其他 Windows** > **程序包管理器控制台**) 并运行以下命令：

```
install-package Microsoft.VisualStudio.Web.BrowserLink
```

或者，可以使用**NuGet 包管理器**。右键单击项目名称在**解决方案资源管理器**，然后选择**管理 NuGet 包**：

查找和安装包：

配置

在 `Startup.Configure` 方法中：

```
app.UseBrowserLink();
```

该代码通常位于 `if` 块仅在开发环境中，启用浏览器链接，如下所示：

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseBrowserLink();
}
```

有关详细信息，请参阅使用多个环境。

如何使用浏览器链接

如果你拥有打开 ASP.NET Core 项目，Visual Studio 将旁边显示浏览器链接工具栏控件调试目标工具栏控件：

从浏览器链接工具栏控件，您可以：

- 刷新 web 应用程序在多个浏览器中的，一次。
- 打开浏览器链接仪表板。
- 启用或禁用浏览器链接。注意：默认情况下，Visual Studio 2017 (15.3) 中禁用浏览器链接。
- 启用或禁用CSS 自动同步。

> **NOTE**
>
> 某些 Visual Studio 插件，最值得注意的是*Web 扩展包 2015 年*并*Web 扩展包 2017 年*，对于浏览器链接，提供扩展的功能，但某些附加功能不与 ASP 配合使用。NET Core 项目。

同时刷新多个浏览器中的 web 应用

若要选择的单个 web 浏览器启动启动项目时，使用中的下拉列表菜单调试目标工具栏控件：

若要同时打开多个浏览器，选择**浏览使用...** 从相同的下拉列表。按住 CTRL 键以选择所需的浏览器，然后单击**浏览**：

下面是屏幕截图显示带有索引视图的 Visual Studio 打开和两个打开的浏览器：

若要查看的浏览器连接到该项目的浏览器链接工具栏控件上悬停：

更改索引视图，并单击浏览器链接刷新按钮时，会更新所有连接的浏览器：

浏览器链接也适用于浏览器从 Visual Studio 外部启动并导航到应用程序 URL。

在浏览器链接仪表板

从浏览器链接下拉列表菜单来管理与打开的浏览器的连接打开浏览器链接仪表板：

如果浏览器不连接，您可以通过选择启动非调试会话*在浏览器中的查看*链接：

否则，连接的浏览器显示与每个浏览器显示的页的路径：

如果你愿意，您可以单击列出的浏览器名称，若要刷新该单个浏览器。

启用或禁用浏览器链接

重新启用时浏览器链接后禁用它，您必须刷新浏览器，将它们重新连接。

启用或禁用 CSS 自动同步

启用 CSS 自动同步后，对 CSS 文件进行任何更改时自动刷新连接的浏览器。

其工作原理

浏览器链接使用 SignalR 创建 Visual Studio 和浏览器之间的通信通道。启用浏览器链接后，Visual Studio 将充当多个客户端（浏览器）可以连接到的 SignalR 服务器。浏览器链接还会在 ASP.NET Core 请求管道中注册中间件组件。此组件将注入特殊 `<script>` 到每个页面请求来自服务器的引用。您可以通过选择查看脚本引用查看源浏览器并滚动到末尾 `<body>` 标记的内容：

```
    <!-- Visual Studio Browser Link -->
    <script type="application/json" id="__browserLink_initializationData">
        {"requestId":"a717d5a07c1741949a7cefd6fa2bad08","requestMappingFromServer":false}
    </script>
    <script type="text/javascript" src="http://localhost:54139/b6e36e429d034f578ebccd6a79bf19bf/browserLink"
async="async"></script>
    <!-- End Browser Link -->
</body>
```

源文件不会被修改。中间件组件动态插入脚本引用。

由于浏览器端代码是所有 JavaScript，它适用于 SignalR 支持，无需浏览器插件的所有浏览器。

通过 ASP.NET Core 使用单页应用程序模板

2018/6/21 • • Edit Online

> **NOTE**
> 已发布的 .NET Core 2.0.x SDK 包括 Angular、React 以及带 Redux 的 React 早期项目模板。本文档并不涉及这些早期项目模板。本文档适用于最新的 Angular、React 以及带 Redux 的 React 模板，这些模板可手动安装到 ASP.NET Core 2.0 中。ASP.NET Core 2.1 中默认包含这些模板。

系统必备

- .NET Core SDK 2.0 或更高版本
- Node.js 版本 6 或更高版本

安装

模板已随 ASP.NET Core 2.1 一起安装。

如果拥有 ASP.NET Core 2.0，请运行以下命令，安装 Angular、React 以及带 Redux 的 React 更新 ASP.NET Core 模板：

```
dotnet new --install Microsoft.DotNet.Web.Spa.ProjectTemplates::2.0.0
```

使用模板

- 使用 Angular 项目模板
- 使用 React 项目模板
- 使用带 Redux 的 React 项目模板

通过 ASP.NET Core 使用 Angular 项目模板

2018/10/15 • • Edit Online

> **NOTE**
> 本文档不涉及 ASP.NET Core 2.0 中包含的 Angular 项目模板。本文介绍的是你可以手动更新的新版 Angular 模板。该模板默认包含在 ASP.NET Core 2.1 中。

更新的 Angular 项目模板为使用 Angular 和 Angular CLI 实现丰富的客户端用户界面 (UI) 的 ASP.NET Core 应用提供了便捷起点。

该模板等同于创建用作 API 后端的 ASP.NET Core 项目，而 Angular CLI 项目用作 UI。该模板提供在单个应用项目中托管两种项目类型的便利。因此，应用项目可以作为单个单元来生成和发布。

创建新应用

如果使用 ASP.NET Core 2.0，请确保安装了更新的 Angular 项目模板。

如果您有安装 ASP.NET Core 2.1，则无需安装角度项目模板。

在空目录中使用命令 `dotnet new angular` 从命令提示符创建一个新项目。例如，以下命令在 my-new-app 目录中创建应用并切换到该目录：

```
dotnet new angular -o my-new-app
cd my-new-app
```

从 Visual Studio 或 .NET Core CLI 运行应用：

- Visual Studio
- .NET Core CLI

打开生成的 .csproj 文件，并从此文件正常运行应用。

生成过程会在首次运行时还原 npm 依赖关系，这可能需要几分钟的时间。后续版本要快得多。

该项目模板创建一个 ASP.NET Core 应用和一个 Angular 应用。ASP.NET Core 应用旨在用于数据访问、授权和其他服务器端问题。位于 ClientApp 子目录中的 Angular 应用旨在用于所有 UI 问题。

添加页面、映像、样式、模块等。

ClientApp 目录包含标准的 Angular CLI 应用。有关详细信息，请参阅官方 Angular 文档。

此模板创建的 Angular 应用与 Angular CLI 本身创建的应用（通过 `ng new`）之间存在细微差异；但是，该应用的功能未变。该模板创建的应用包含基于 Bootstrap 的布局和基本路由示例。

运行 ng 命令

在命令提示符中，切换到 ClientApp 子目录：

```
cd ClientApp
```

如果全局安装了 `ng` 工具，则可以运行其任何命令。例如，你可以运行 `ng lint`、`ng test` 或任何其他 Angular CLI 命令。不过无需运行 `ng serve`，因为 ASP.NET Core 应用为应用的服务器端和客户端部分提供服务。在内部，它在开发中使用 `ng serve`。

如果未安装 `ng` 工具，请改为运行 `npm run ng`。例如，你可以运行 `npm run ng lint` 或 `npm run ng test`。

安装 npm 包

要安装第三方 npm 程序包，请使用 ClientApp 子目录中的命令提示符。例如：

```
cd ClientApp
npm install --save <package_name>
```

发布和部署

在开发过程中，应用在为开发人员之便而优化的模式下运行。例如，JavaScript 捆绑包包含源映射（这样在调试时可以查看原始 TypeScript 代码）。该应用监视磁盘上的 TypeScript、HTML 和 CSS 文件更改，并在看到这些文件更改时自动重新编译和重新加载。

在生产中，提供针对性能进行了优化的应用版本。该操作被配置为自动发生。发布时，生成配置会发出预先 (AoT) 编译的压缩客户端代码版本。与开发版本不同，生产版本不需要在服务器上安装 Node.js（除非已启用服务器端预呈现）。

你可以使用标准 ASP.NET Core 托管和部署方法。

独立运行“ng serve”

在 ASP.NET Core 应用以开发模式启动时，该项目配置为在后台启动自己的 Angular CLI 服务器实例。这很方便，因为你无需手动运行单独的服务器。

这种默认设置有一个缺点。每次修改 C# 代码并且需要重启 ASP.NET Core 应用时，Angular CLI 服务器都会重启。大约需要 10 秒才能开始备份。如果你经常进行 C# 代码编辑并且不希望等待 Angular CLI 重启，请在外部运行独立于 ASP.NET Core 进程的 Angular CLI 服务器。为此，请执行以下操作：

1. 在命令提示符中，切换到 ClientApp 子目录，并启动 Angular CLI 开发服务器：

   ```
   cd ClientApp
   npm start
   ```

   > **IMPORTANT**
   > 使用 `npm start` 而不是 `ng serve` 启动 Angular CLI 开发服务器，以便遵守 package.json 中的配置。要将其他参数传递给 Angular CLI 服务器，请将它们添加到 package.json 文件中的相关 `scripts` 行。

2. 修改 ASP.NET Core 应用以使用外部 Angular CLI 实例，而不是启动它自己的实例。在 Startup 类中，将 `spa.UseAngularCliServer` 调用替换为以下内容：

   ```
   spa.UseProxyToSpaDevelopmentServer("http://localhost:4200");
   ```

当启动 ASP.NET Core 应用时，它不会启动 Angular CLI 服务器，而是使用你手动启动的实例。这使它能够更快地启动和重新启动。不再需要每次等待 Angular CLI 重新生成客户端应用。

服务器端呈现

作为一项性能特性，你可以选择在服务器上预呈现 Angular 应用并在客户端运行它。这意味着浏览器会收到表示应用初始 UI 的 HTML 标记，所以即使在下载并执行 JavaScript 捆绑包之前，浏览器也会显示它。该操作的大部分实现过程由名为 Angular Universal 的 Angular 功能完成。

> **TIP**
> 启用服务器端呈现 (SSR) 会在开发和部署期间引入一些额外的复杂问题。阅读 SSR 缺点，以确定 SSR 是否符合你的要求。

要启用 SSR，你需要将大量内容添加到项目。

在 Startup 类中，在配置 `spa.Options.SourcePath` 的行之后并且在调用 `UseAngularCliServer` 或 `UseProxyToSpaDevelopmentServer` 之前，添加以下内容：

```
app.UseSpa(spa =>
{
    spa.Options.SourcePath = "ClientApp";

    spa.UseSpaPrerendering(options =>
    {
        options.BootModulePath = $"{spa.Options.SourcePath}/dist-server/main.bundle.js";
        options.BootModuleBuilder = env.IsDevelopment()
            ? new AngularCliBuilder(npmScript: "build:ssr")
            : null;
        options.ExcludeUrls = new[] { "/sockjs-node" };
    });

    if (env.IsDevelopment())
    {
        spa.UseAngularCliServer(npmScript: "start");
    }
});
```

在开发模式下，此代码尝试通过运行在 ClientApp\package.json 中定义的脚本 `build:ssr` 来构建 SSR 捆绑包。这将生成一个名为 `ssr` 的 Angular 应用，该应用尚未定义。

在 ClientApp/.angular-cli.json 中 `apps` 数组的末尾，定义一个名为 `ssr` 的额外应用。使用以下选项：

```
{
  "name": "ssr",
  "root": "src",
  "outDir": "dist-server",
  "assets": [
    "assets"
  ],
  "main": "main.server.ts",
  "tsconfig": "tsconfig.server.json",
  "prefix": "app",
  "scripts": [],
  "environmentSource": "environments/environment.ts",
  "environments": {
    "dev": "environments/environment.ts",
    "prod": "environments/environment.prod.ts"
  },
  "platform": "server"
}
```

启用了 SSR 的新应用配置需要另外两个文件：tsconfig.server.json 和 main.server.ts。tsconfig.server.json 文件指定 TypeScript 编译选项。main.server.ts 文件在 SSR 期间用作代码入口点。

在 ClientApp/src（与现有 tsconfig.app.json 同时存在）中添加名为 tsconfig.server.json 的新文件，其中包含以下内容：

```
{
    "extends": "../tsconfig.json",
    "compilerOptions": {
      "baseUrl": "./",
      "module": "commonjs"
    },
    "angularCompilerOptions": {
      "entryModule": "app/app.server.module#AppServerModule"
    }
}
```

该文件将 Angular 的 AoT 编译器配置为查找名为 `app.server.module` 的模块。通过在 ClientApp/src/app/app.server.module.ts（与现有 app.module.ts 同时存在）处创建新文件来添加该模块，其中包含以下内容：

```
import { NgModule } from '@angular/core';
import { ServerModule } from '@angular/platform-server';
import { ModuleMapLoaderModule } from '@nguniversal/module-map-ngfactory-loader';
import { AppComponent } from './app.component';
import { AppModule } from './app.module';

@NgModule({
    imports: [AppModule, ServerModule, ModuleMapLoaderModule],
    bootstrap: [AppComponent]
})
export class AppServerModule { }
```

该模块继承自客户端 `app.module` 并定义在 SSR 期间哪些额外的 Angular 模块可用。

回想一下，.angular-cli.json 中的新 `ssr` 条目引用了名为 main.server.ts 的入口点文件。如果你尚未添加该文件，那么现在是时候添加了。在 ClientApp/src/main.server.ts 处创建一个新文件（与现有 main.ts 同时存在），其中包含以下内容：

```
import 'zone.js/dist/zone-node';
import 'reflect-metadata';
import { renderModule, renderModuleFactory } from '@angular/platform-server';
import { APP_BASE_HREF } from '@angular/common';
import { enableProdMode } from '@angular/core';
import { provideModuleMap } from '@nguniversal/module-map-ngfactory-loader';
import { createServerRenderer } from 'aspnet-prerendering';
export { AppServerModule } from './app/app.server.module';

enableProdMode();

export default createServerRenderer(params => {
  const { AppServerModule, AppServerModuleNgFactory, LAZY_MODULE_MAP } = (module as any).exports;

  const options = {
    document: params.data.originalHtml,
    url: params.url,
    extraProviders: [
      provideModuleMap(LAZY_MODULE_MAP),
      { provide: APP_BASE_HREF, useValue: params.baseUrl },
      { provide: 'BASE_URL', useValue: params.origin + params.baseUrl }
    ]
  };

  const renderPromise = AppServerModuleNgFactory
    ? /* AoT */ renderModuleFactory(AppServerModuleNgFactory, options)
    : /* dev */ renderModule(AppServerModule, options);

  return renderPromise.then(html => ({ html }));
});
```

此文件的代码是 ASP.NET Core 在运行添加到 Startup 类的 `UseSpaPrerendering` 中间件时为每个请求执行的内容。它处理从 .NET 代码（例如所请求的 URL）收到的 `params`，并调用 Angular SSR API 以获取生成的 HTML。

严格来说，这足以在开发模式下启用 SSR。重要的是做出最后的更改，以便应用在发布时能够正常工作。在应用的主 .csproj 文件中，将 `BuildServerSideRenderer` 属性值设置为 `true`：

```
<!-- Set this to true if you enable server-side prerendering -->
<BuildServerSideRenderer>true</BuildServerSideRenderer>
```

这会将生成过程配置为在发布期间运行 `build:ssr`，并将 SSR 文件部署到服务器。如果未启用此功能，则 SSR 将在生产中失败。

当应用在开发模式或生产模式下运行时，Angular 代码会在服务器上预呈现为 HTML。客户端代码执行正常。

将 .NET 代码中的数据传递到 TypeScript 代码中

在 SSR 期间，你可能希望将从 ASP.NET Core 应用预请求的数据传递到 Angular 应用。例如，你可以传递 cookie 信息或从数据库读取的某些内容。若要执行此操作，请编辑 Startup 类。在 `UseSpaPrerendering` 的回叫中，为 `options.SupplyData` 设置一个值，如下所示：

```
options.SupplyData = (context, data) =>
{
    // Creates a new value called isHttpsRequest that's passed to TypeScript code
    data["isHttpsRequest"] = context.Request.IsHttps;
};
```

`SupplyData` 回叫允许传递按请求的任意 JSON 序列化数据（例如字符串、布尔值或数字）。main.server.ts 代码将此数据接收为 `params.data`。例如，前面的代码示例将一个布尔值作为 `params.data.isHttpsRequest` 传递给 `createServerRenderer` 回叫。你可以通过 Angular 支持的任何方式将此传递给应用的其他部分。例如，请参阅

main.server.ts 如何将 `BASE_URL` 值传递给构造函数被声明为接收它的任何组件。

SSR 的缺点

并非所有应用都受益于 SSR。主要优点是可感知的性能。通过缓慢的网络连接或缓慢的移动设备连接你的应用的访问者可以快速查看初始 UI，即使需要一段时间才能提取或分析 JavaScript 捆绑包也是如此。但是，许多 SPA 主要在具有快速内部公司网络的快速计算机（其中应用几乎立即显示）上使用。

同时，启用 SSR 也存在重大缺点。它增加了开发过程的复杂性。代码必须在两个不同的环境中运行：客户端和服务器端（在从 ASP.NET Core 中调用的 Node.js 环境中）。以下是一些需要谨记的事项：

- SSR 需要在生产服务器上安装 Node.js。对于某些部署方案（例如 Azure 应用服务）而言，情况自然如此，但对于其他方案（例如 Azure Service Fabric）则不适用。
- 启用 `BuildServerSideRenderer` 生成标志会导致发布 node_modules 目录。该文件夹包含 20,000 多个文件，这会增加部署时间。
- 要在 Node.js 环境中运行代码，则不能依赖于特定浏览器的 JavaScript API（如 `window` 或 `localStorage`）。如果代码（或引用的某个第三方库）尝试使用这些 API，则在 SSR 期间会出现错误。例如，不要使用 jQuery，因为它在许多位置引用特定浏览器的 API。要避免错误，你必须避免使用 SSR 或避免使用特定于浏览器的 API 或库。你可以将对这些 API 的任何调用包装在检查中，以确保它们在 SSR 期间不被调用。例如，在 JavaScript 或 TypeScript 代码中使用如下所示的检查：

  ```
  if (typeof window !== 'undefined') {
      // Call browser-specific APIs here
  }
  ```

通过 ASP.NET Core 使用 React 项目模板

2018/9/18 • • Edit Online

> **NOTE**
> 本文档不涉及 ASP.NET Core 2.0 中包含的 React 项目模板。本文介绍你可以手动更新的新版 React 模板。该模板默认包含在 ASP.NET Core 2.1 中。

更新的 React 项目模板为使用 React 和 create-react-app (CRA) 约定实现丰富的客户端用户界面 (UI) 的 ASP.NET Core 应用程序提供了便捷起点。

该模板等同于创建两个项目，即用作 API 后端的 ASP.NET Core 项目和用作 UI 的标准 CRA React 项目，但均可在可以生成并发布为单个单元的单个应用程序项目中进行托管。

创建新应用

如果使用 ASP.NET Core 2.0，请确保安装了更新的 React 项目模板。

如果您有安装 ASP.NET Core 2.1，则无需安装响应项目模板。

在空目录中使用命令 `dotnet new react` 从命令提示符创建一个新项目。例如，以下命令在 my-new-app 目录中创建应用并切换到该目录：

```
dotnet new react -o my-new-app
cd my-new-app
```

从 Visual Studio 或 .NET Core CLI 运行应用：

- Visual Studio
- .NET Core CLI

打开生成的 .csproj 文件，并从此文件正常运行应用。

生成过程会在首次运行时还原 npm 依赖关系，这可能需要几分钟的时间。后续版本要快得多。

该项目模板创建一个 ASP.NET Core 应用和一个 React 应用。ASP.NET Core 应用旨在用于数据访问、授权和其他服务器端问题。位于 ClientApp 子目录中的 React 应用旨在用于所有 UI 问题。

添加页面、映像、样式、模块等。

ClientApp 目录是标准的 CRA React 应用。有关详细信息，请参阅官方 CRA 文档。

此模板创建的 React 应用与 CRA 自身创建的应用之间存在细微差异；但是，该应用的功能未变。该模板创建的应用包含基于 Bootstrap 的布局和基本路由示例。

安装 npm 包

要安装第三方 npm 程序包，请使用 ClientApp 子目录中的命令提示符。例如：

```
cd ClientApp
npm install --save <package_name>
```

发布和部署

在开发过程中，应用在为开发人员之便而优化的模式下运行。例如，JavaScript 捆绑包包含源映射（这样在调试时可以查看原始源代码）。该应用程序监视磁盘上的 JavaScript、HTML 和 CSS 文件更改，并在看到这些文件更改时自动重新编译和重新加载。

在生产中，提供针对性能进行了优化的应用版本。该操作被配置为自动发生。发布时，生成配置会发出转换的压缩客户端代码版本。与开发版本不同，生产版本不需要在服务器上安装 Node.js。

你可以使用标准 ASP.NET Core 托管和部署方法。

独立运行 CRA 服务器

在 ASP.NET Core 应用以开发模式启动时，该项目配置为在后台启动自己的 CRA 开发服务器实例。这很方便，因为这表示你无需手动运行单独的服务器。

这种默认设置有一个缺点。每次修改 C# 代码并且需要重启 ASP.NET Core 应用时，CRA 服务器都会重启。大约需要几秒才能开始备份。如果你经常进行 C# 代码编辑并且不希望等待 CRA 服务器重启，请在外部运行独立于 ASP.NET Core 进程的 CRA 服务器。为此，请执行以下操作：

1. 在命令提示符中，切换到 ClientApp 子目录，并启动 CRA 开发服务器：

   ```
   cd ClientApp
   npm start
   ```

2. 修改 ASP.NET Core 应用以使用外部 CRA 服务器实例，而不是启动它自己的实例。在 Startup 类中，将 `spa.UseReactDevelopmentServer` 调用替换为以下内容：

   ```
   spa.UseProxyToSpaDevelopmentServer("http://localhost:3000");
   ```

当启动 ASP.NET Core 应用时，它不会启动 CRA 服务器，而是使用你手动启动的实例。这使它能够更快地启动和重新启动。不再需要每次等待 React 应用重新生成。

通过 ASP.NET Core 使用带 Redux 的 React 项目模板

2018/6/22 • • Edit Online

> **NOTE**
>
> 本文档不涉及 ASP.NET Core 2.0 中包含的带 Redux 的 React项目模板。本文介绍你可以手动更新的新版带 Redux 的 React 模板。该模板默认包含在 ASP.NET Core 2.1 中。

更新的带 Redux 的 React 项目模板为使用 React、Redux 和 create-react-app (CRA) 约定实现丰富的客户端用户界面 (UI) 的 ASP.NET Core 应用程序提供了便捷起点。

除了项目创建命令外，关于带 Redux 的 React 模板的所有信息都与 React 模板相同。要创建此项目类型，请运行 `dotnet new reactredux` 而不是 `dotnet new react`。有关这两个基于 React 的模板的通用功能的详细信息，请参阅 React 模板文档。

使用 JavaScriptServices 在 ASP.NET Core 中创建单页应用程序

2018/12/17 • • Edit Online

通过Scott Addie和Fiyaz Hasan

单页应用程序 (SPA) 因其固有的丰富用户体验而成为一种常用的 Web 应用程序。将客户端 SPA 框架或库(例如 Angular 或 React)与服务器端框架(例如 ASP.NET Core)集成可能很困难。开发 JavaScriptServices 的目的是减少集成过程中的问题。使用它可以在不同的客户端和服务器技术堆栈之间进行无缝操作。

什么是 JavaScriptServices

JavaScriptServices 是 ASP.NET Core 的客户端技术的集合。其目标是将 ASP.NET Core 定位为开发人员用于构建 SPA 的首选服务器端平台。

JavaScriptServices 包含三个不同的 NuGet 包:

- Microsoft.AspNetCore.NodeServices (NodeServices)
- Microsoft.AspNetCore.SpaServices (SpaServices)
- Microsoft.AspNetCore.SpaTemplates (SpaTemplates)

这些包适用于以下情况:

- 在服务器上运行 JavaScript
- 使用 SPA 框架或库
- 生成客户端的资产的 Webpack

在本文的重点放在使用 SpaServices 包。

什么是 SpaServices

创建 SpaServices 是为了将 ASP.NET Core 定位为开发人员构建 SPA 的首选服务器端平台。SpaServices 不是使用 ASP.NET Core 开发 SPA 所必需的，它也不会将你锁定到特定的客户端框架。

SpaServices 提供有用的基础结构，例如:

- 服务器端预呈现
- Webpack 开发中间件
- 热模块更换
- 路由帮助程序

总体来说，这些基础结构组件增强了开发工作流和运行时体验。组件可单独采用。

使用 SpaServices 的先决条件

若要使用 SpaServices，安装以下组件:

- Node.js（6 或更高版本）与 npm
  - 若要验证这些组件安装，并可找到，运行以下命令从命令行:

```
node -v && npm -v
```

注意：如果要部署到 Azure 网站，您不需要此处执行任何操作—Node.js 已安装并且可用的服务器环境中。

- .NET Core SDK 2.0 或更高版本
  - 如果你使用 Visual Studio 2017 在 Windows 上，通过选择安装 SDK **.NET Core 跨平台开**发工作负荷。
- Microsoft.AspNetCore.SpaServices NuGet 包

服务器端预呈现

通用（也称"同构"）应用程序是在服务器和客户端上都能运行的 JavaScript 应用程序。Angular、React 和其他常用框架提供了一个适合此应用程序开发风格的通用平台。这其中的理念是，先通过 Node.js 在服务器上呈现框架组件，然后将下一步的执行操作委托到客户端。

ASP.NET Core标记帮助程序由 SpaServices 简化通过调用服务器上的 JavaScript 函数的服务器端预呈现的实现。

系统必备

安装以下组件：

- aspnet 预呈现npm 包：

```
npm i -S aspnet-prerendering
```

配置

标记帮助程序在项目的供发现通过命名空间注册 *_ViewImports.cshtml*文件：

```
@using SpaServicesSampleApp
@addTagHelper "*, Microsoft.AspNetCore.Mvc.TagHelpers"
@addTagHelper "*, Microsoft.AspNetCore.SpaServices"
```

这些标记帮助程序抽象出与低级 Api 直接通信通过利用 Razor 视图中的类似于 HTML 的语法的复杂性：

```
<app asp-prerender-module="ClientApp/dist/main-server">Loading...</app>
```

`asp-prerender-module`**标记帮助程序**

`asp-prerender-module`标记帮助程序，使用在前面的代码示例中，执行*ClientApp/dist/main-server.js*通过 Node.js 服务器上。为清晰起见*main server.js*文件是一个项目中的 TypeScript JavaScript 的转译任务Webpack生成过程。Webpack 定义入口点的别名`main-server`；并遍历此别名的依赖项关系图的开始处*ClientApp/启动 server.ts*文件：

```
entry: { 'main-server': './ClientApp/boot-server.ts' },
```

在下述 Angular 示例中，*ClientApp/boot-server.ts*文件利用`aspnet-prerendering` npm 包的`createServerRenderer`函数和`RenderResult`类型通过 Node.js 来配置服务器呈现。需要在服务器端呈现的 HTML 标记会传递给一个解析函数调用，该调用包装在强类型化的 JavaScript `Promise` 对象中。`Promise`对象的意义在于，它以异步方式将 HTML 标记提供给页面，以便该标记能够注入到 DOM 的占位符元素中。

```
import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
    const providers = [
        { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
        { provide: 'ORIGIN_URL', useValue: params.origin }
    ];

    return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
        const appRef = moduleRef.injector.get(ApplicationRef);
        const state = moduleRef.injector.get(PlatformState);
        const zone = moduleRef.injector.get(NgZone);

        return new Promise<RenderResult>((resolve, reject) => {
            zone.onError.subscribe(errorInfo => reject(errorInfo));
            appRef.isStable.first(isStable => isStable).subscribe(() => {
                // Because 'onStable' fires before 'onError', we have to delay slightly before
                // completing the request in case there's an error to report
                setImmediate(() => {
                    resolve({
                        html: state.renderToString()
                    });
                    moduleRef.destroy();
                });
            });
        });
    });
});
```

`asp-prerender-data` 标记帮助程序

当结合 `asp-prerender-module` 标记帮助程序 `asp-prerender-data` 标记帮助程序可用于将从 Razor 视图的上下文信息传递到服务器端 JavaScript。例如，以下标记将传递到的用户数据 `main-server` 模块：

```
<app asp-prerender-module="ClientApp/dist/main-server"
        asp-prerender-data='new {
            UserName = "John Doe"
        }'>Loading...</app>
```

收到 `UserName` 参数使用内置的 JSON 序列化程序序列化并存储在 `params.data` 对象。在以下 Angular 示例中，数据用于在 `h1` 元素中构造个性化的问候语：

```
import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
    const providers = [
        { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
        { provide: 'ORIGIN_URL', useValue: params.origin }
    ];

    return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
        const appRef = moduleRef.injector.get(ApplicationRef);
        const state = moduleRef.injector.get(PlatformState);
        const zone = moduleRef.injector.get(NgZone);

        return new Promise<RenderResult>((resolve, reject) => {
            const result = `<h1>Hello, ${params.data.userName}</h1>`;

            zone.onError.subscribe(errorInfo => reject(errorInfo));
            appRef.isStable.first(isStable => isStable).subscribe(() => {
                // Because 'onStable' fires before 'onError', we have to delay slightly before
                // completing the request in case there's an error to report
                setImmediate(() => {
                    resolve({
                        html: result
                    });
                    moduleRef.destroy();
                });
            });
        });
    });
});
```

注意：在标记帮助程序中传递的属性名称表示与**pascal 命名法**表示法。为 JavaScript，其中，相同的属性名称表示与对比驼**峰式大小写**。默认 JSON 序列化配置负责这种差异。

若要展开在前面的代码示例时，数据可从服务器到视图通过传递 hydrating `globals` 属性提供给 `resolve` 函数：

```
import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
    const providers = [
        { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
        { provide: 'ORIGIN_URL', useValue: params.origin }
    ];

    return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
        const appRef = moduleRef.injector.get(ApplicationRef);
        const state = moduleRef.injector.get(PlatformState);
        const zone = moduleRef.injector.get(NgZone);

        return new Promise<RenderResult>((resolve, reject) => {
            const result = `<h1>Hello, ${params.data.userName}</h1>`;

            zone.onError.subscribe(errorInfo => reject(errorInfo));
            appRef.isStable.first(isStable => isStable).subscribe(() => {
                // Because 'onStable' fires before 'onError', we have to delay slightly before
                // completing the request in case there's an error to report
                setImmediate(() => {
                    resolve({
                        html: result,
                        globals: {
                            postList: [
                                'Introduction to ASP.NET Core',
                                'Making apps with Angular and ASP.NET Core'
                            ]
                        }
                    });
                    moduleRef.destroy();
                });
            });
        });
    });
});
```

`postList` 内部定义的数组 `globals` 对象附加到浏览器的全局 `window` 对象。为全局作用域此变量提升可消除重复工作，特别是因为它与加载一次在服务器上，再次在客户端上的相同数据。

Webpack 开发中间件

Webpack 开发中间件引入了 Webpack 按需生成资源的由此简化了的开发工作流。中间件会自动编译并在浏览器中重新加载页面时提供客户端的资源。另一种方法是手动 Webpack 调用通过项目的 npm 生成脚本的第三方依赖项或自定义代码发生更改时。Npm 生成脚本*package.json*文件显示在下面的示例：

```
"build": "npm run build:vendor && npm run build:custom",
```

系统必备

安装以下组件：

- aspnet webpack npm 包：

```
npm i -D aspnet-webpack
```

配置

到 HTTP 请求管道中的以下代码通过注册 Webpack 开发中间件*Startup.cs*文件的 `Configure` 方法：

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseWebpackDevMiddleware();
}
else
{
    app.UseExceptionHandler("/Home/Error");
}

// Call UseWebpackDevMiddleware before UseStaticFiles
app.UseStaticFiles();
```

`UseWebpackDevMiddleware` 前必须调用扩展方法注册静态文件托管通过 `UseStaticFiles` 扩展方法。出于安全原因，仅当应用程序在开发模式下运行时注册该中间件。

*Webpack.config.js*文件的 `output.publicPath` 属性会指示要观看的中间件 `dist` 文件夹的更改：

```
module.exports = (env) => {
        output: {
            filename: '[name].js',
            publicPath: '/dist/' // Webpack dev middleware, if enabled, handles requests for this URL prefix
        },
```

热模块更换

Webpack 的思考动态模块更换(HMR) 功能作为一种演变Webpack 开发中间件。HMR 引入了完全相同的好处，但它进一步简化开发工作流通过自动编译所做的更改后更新页面内容。不要混淆这与刷新浏览器中，这会干扰的当前内存中状态和 SPA 的调试会话。没有 Webpack 开发中间件服务和浏览器中，这意味着更改推送到浏览器之间的活动链接。

系统必备

安装以下组件：

- webpack 的热的中间件npm 包：

```
npm i -D webpack-hot-middleware
```

配置

HMR 组件必须注册到 MVC 的 HTTP 请求管道中 `Configure` 方法：

```
app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions {
    HotModuleReplacement = true
});
```

作为是如此Webpack 开发中间件，则 `UseWebpackDevMiddleware` 前必须调用扩展方法 `UseStaticFiles` 扩展方法。出于安全原因，仅当应用程序在开发模式下运行时注册该中间件。

*Webpack.config.js*文件必须定义 `plugins`，即使它保留为空数组：

```
module.exports = (env) => {
        plugins: [new CheckerPlugin()]
```

加载后在浏览器中的应用，开发人员工具的控制台选项卡提供了 HMR 激活的确认：

路由帮助程序

在大多数基于 ASP.NET Core 的 Spa，需要将客户端的路由除了服务器端的路由。SPA 和 MVC 路由系统可以不受干扰地独立工作。没有，但是，一个边缘事例造成面临的难题：标识 404 HTTP 响应。

请考虑在该方案中的无扩展名路由 `/some/page` 使用。假定该请求不模式匹配的服务器端的路由，但其模式匹配的客户端的路由。现在，考虑的传入请求 `/images/user-512.png`，这通常需要查找服务器上的图像文件。如果该请求的资源路径不符合的任何服务器端的路由或静态文件，它不大可能的客户端应用程序对其进行处理，你通常想要返回 404 HTTP 状态代码。

系统必备

安装以下组件：

- 客户端的路由 npm 包。使用 Angular 作为示例：

  ```
  npm i -S @angular/router
  ```

配置

名为的扩展方法 `MapSpaFallbackRoute` 中使用 `Configure` 方法：

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");

    routes.MapSpaFallbackRoute(
        name: "spa-fallback",
        defaults: new { controller = "Home", action = "Index" });
});
```

提示：路由配置它们的顺序计算。因此，`default`进行模式匹配第一次使用在前面的代码示例中的路由。

创建新的项目

JavaScriptServices 提供了预配置的应用程序模板。SpaServices 在这些模板中与不同的框架和库（例如 Angular、React 和 Redux）配合使用。

可以通过.NET Core CLI 安装这些模板，通过运行以下命令：

```
dotnet new --install Microsoft.AspNetCore.SpaTemplates::*
```

显示可用的 SPA 模板的列表：

模板	短名称	语言	TAGS
带 Angular 的 MVC ASP.NET Core	angular	[C#]	Web/MVC/SPA
带有 React.js 的 MVC ASP.NET Core	react	[C#]	Web/MVC/SPA
含 React.js 和 Redux 的 MVC ASP.NET Core	reactredux	[C#]	Web/MVC/SPA

若要使用 SPA 模板之一创建新项目，请在 dotnet new 命令中包括模板的**短名称**。以下命令创建 Angular 应用程序，并为服务器端配置 ASP.NET Core MVC：

```
dotnet new angular
```

设置运行时配置模式

存在两种主要的运行时配置模式：

- **开**发：
  - 包括源映射，以便进行调试。
  - 不会优化性能的客户端代码。
- **生**产：
  - 不包括源映射。
  - 可优化通过捆绑和缩小客户端代码。

ASP.NET Core 使用名为的环境变量`ASPNETCORE_ENVIRONMENT`来存储配置模式。请参阅 **将环境设置** 有关详细信息。

运行使用.NET Core CLI

通过在项目根目录运行以下命令还原所需的 NuGet 和 npm 包：

```
dotnet restore && npm i
```

生成并运行应用程序：

```
dotnet run
```

在应用程序根据本地主机上启动运行时配置模式。导航到 `http://localhost:5000` 在浏览器中显示的登录页。

运行使用 Visual Studio 2017

打开 *.csproj*生成的文件dotnet 新命令。在项目中打开时自动还原所需的 NuGet 和 npm 包。此还原过程可能需要几分钟时间，并在应用程序已准备好在它完成后运行。单击绿色的运行的按钮或按 `Ctrl + F5`，并在浏览器打开到应用程序的登录页。应用程序运行于 localhost 根据运行时配置模式。

测试应用程序

SpaServices 模板是预配置为运行客户端的测试使用Karma并Jasmine。Jasmine 是常用的单元测试框架，适用于 JavaScript，而 Karma 是这些测试的测试运行程序。Karma 配置为使用Webpack 开发中间件这样开发人员不需要停止并运行测试，每次进行更改。无论是针对测试用例或测试用例本身运行的代码，则将自动运行测试。

以 Angular 应用程序为例，我们在 `CounterComponent` 文件中为 *counter.component.spec.ts*提供了两个 Jasmine 测试用例：

```
it('should display a title', async(() => {
    const titleText = fixture.nativeElement.querySelector('h1').textContent;
    expect(titleText).toEqual('Counter');
}));

it('should start with count 0, then increments by 1 when clicked', async(() => {
    const countElement = fixture.nativeElement.querySelector('strong');
    expect(countElement.textContent).toEqual('0');

    const incrementButton = fixture.nativeElement.querySelector('button');
    incrementButton.click();
    fixture.detectChanges();
    expect(countElement.textContent).toEqual('1');
}));
```

打开命令提示符中*ClientApp*目录。运行下面的命令：

```
npm test
```

该脚本将启动 Karma 测试运行程序，其内容中定义的设置*karma.conf.js*文件。在其他设置*karma.conf.js*识别的测试文件，通过执行其 `files` 数组：

```
module.exports = function (config) {
    config.set({
        files: [
            '../../wwwroot/dist/vendor.js',
            './boot-tests.ts'
        ],
```

发布应用程序

将生成的客户端的资产和已发布的 ASP.NET Core 项目合并为随时可部署的包可能会很麻烦。幸运的是，SpaServices 协调与名为的自定义 MSBuild 目标的整个发布过程 `RunWebpack`：

```
<Target Name="RunWebpack" AfterTargets="ComputeFilesToPublish">
  <!-- As part of publishing, ensure the JS resources are freshly built in production mode -->
  <Exec Command="npm install" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --config webpack.config.vendor.js --env.prod" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --env.prod" />

  <!-- Include the newly-built files in the publish output -->
  <ItemGroup>
    <DistFiles Include="wwwroot\dist\**; ClientApp\dist\**" />
    <ResolvedFileToPublish Include="@(DistFiles->'%(FullPath)')" Exclude="@(ResolvedFileToPublish)">
      <RelativePath>%(DistFiles.Identity)</RelativePath>
      <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
    </ResolvedFileToPublish>
  </ItemGroup>
</Target>
```

MSBuild 目标具有下列职责：

1. 还原 npm 包
2. 创建第三方、客户端的资产的生产级版本
3. 创建自定义客户端的资产的生产级版本
4. Webpack 生成资产复制到 publish 文件夹

运行时，会调用 MSBuild 目标：

```
dotnet publish -c Release
```

其他资源

- Angular 文档

托管和部署 ASP.NET Core

2018/12/13 • • Edit Online

一般而言，向托管环境部署 ASP.NET Core 应用需执行以下操作：

- 将已发布应用部署到托管服务器上的文件夹。
- 设置进程管理器，该管理器在收到请求时启动应用，并在应用发生故障或服务器重启后重新启动应用。
- 对于反向代理配置，将反向代理设置为将请求转发到应用。

发布到文件夹

dotnet publish 命令编译应用代码，并复制在“发布”文件夹中运行应用所需的文件。使用 Visual Studio 进行部署时，自动先执行 `dotnet publish` 步骤，再将文件复制到部署目标。

文件夹内容

“发布”文件夹包含一个或多个应用程序集文件、依赖项以及（可选）.NET 运行时。

.NET Core 应用可以发布为“独立式部署”，也可以发布为“依赖框架的部署”。如果应用是独立式，“发布”文件夹中有包含 .NET 运行时的程序集文件。如果应用依赖于框架，.NET 运行时文件就不包含在内，因为应用包含对服务器上安装的 .NET 版本的引用。默认部署模型是依赖于框架的模型。有关详细信息，请参阅 .NET Core 应用程序部署。

除了“.exe”和“.dll”文件以外，ASP.NET Core 应用的“发布”文件夹通常包含配置文件、静态资产和 MVC 视图。有关更多信息，请参见ASP.NET Core 目录结构。

设置进程管理器

ASP.NET Core 应用是一个控制台应用，在服务器启动时必须启动该应用，并且在出现故障后必须重新启动它。若要自动启动和重新启动，需要使用进程管理器。用于 ASP.NET Core 的最常见进程管理器是：

- Linux
  - Nginx
  - Apache
- Windows
  - IIS
  - Windows 服务

设置反向代理

如果应用使用 Kestrel 服务器，Nginx、Apache 或 IIS 可用作反向代理服务器。反向代理服务器接收来自 Internet 的 HTTP 请求，并将这些请求转发到 Kestrel。

无论配置是否使用反向代理服务器——，都是 ASP.NET Core 2.0 或更高版本应用的支持托管配置。有关详细信息，请参阅何时结合使用 Kestrel 和反向代理。

如果应用使用 Kestrel 服务器，并被公开到 Internet，Nginx、Apache 或 IIS 可用作反向代理服务器。反向代理服务器接收来自 Internet 的 HTTP 请求，并将这些请求转发到 Kestrel。使用反向代理的主要原因是出于安全考虑。有关详细信息，请参阅何时结合使用 Kestrel 和反向代理。

代理服务器和负载均衡器方案

对于托管在代理服务器和负载均衡器后方的应用，可能需要附加配置。如不提供附加配置，应用可能无法访问方案 (HTTP/HTTPS) 和发出请求的远程 IP 地址。有关详细信息，请参阅配置 ASP.NET Core 以使用代理服务器和负载均衡器。

使用 Visual Studio 和 MSBuild 自动执行部署

除将输出从 dotnet publish 复制到服务器外，部署通常还需要其他任务。例如，可能需要从"发布"文件夹获取或排除额外文件。Visual Studio 使用 MSBuild 进行 Web 部署，用户可以自定义 MSBuild 以在部署期间执行其他任务。有关详细信息，请参阅 用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件 和使用 MSBuild 和 Team Foundation Build 一书。

通过发布 Web 功能或内置 Git 支持，可以将应用从 Visual Studio 直接部署到 Azure 应用服务。Azure DevOps Services 支持对 Azure 应用服务进行持续部署。有关详细信息，请参阅通过 ASP.NET Core 和 Azure 实现 DevOps。

发布到 Azure

有关如何使用 Visual Studio 将应用发布到 Azure 的说明，请参阅 使用 Visual Studio 将 ASP.NET Core 应用发布到 Azure。在 Azure 中创建 ASP.NET Core Web 应用提供了其他示例。

在 Windows 中使用 MSDeploy 发布

请参阅 用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件，了解如何使用 Visual Studio 发布配置文件发布应用，其中包括在 Windows 命令提示符处使用 dotnet msbuild 命令。

在 Web 场中托管

有关如何配置在 Web 场环境中托管 ASP.NET Core 应用（例如，部署应用的多个实例以实现可伸缩性）的信息，请参阅 在 Web 场中托管 ASP.NET Core。

执行运行状况检查

使用运行状况检查中间件，对应用及其依赖项执行运行状况检查。有关更多信息，请参见ASP.NET Core 中的运行状况检查。

其他资源

- 在 Docker 容器中托管 ASP.NET Core
- 解决 ASP.NET Core 项目

将 ASP.NET Core 应用部署到 Azure 应用服务

2019/1/2 • • Edit Online

Azure 应用服务是一个用于托管 Web 应用(包括 ASP.NET Core)的 Microsoft 云计算平台服务。

有用资源

Azure 应用文档、教程、示例、操作指南和其他资源由 Azure Web 应用文档提供。两个有关托管 ASP.NET Core 应用的重要教程为:

快速入门:在 Azure 中创建 ASP.NET Core Web 应用
使用 Visual Studio 创建 ASP.NET Core Web 应用，并将其部署到 Windows 上的 Azure 应用服务。

快速入门:在 Linux 应用服务中创建 .NET Core Web 应用
使用命令行创建 ASP.NET Core Web 应用，并将其部署到 Linux 上的 Azure 应用服务。

ASP.NET Core 文档中提供以下文章:

使用 Visual Studio 将 ASP.NET Core 应用发布到 Azure
了解如何使用 Visual Studio 将 ASP.NET Core 应用发布到 Azure 应用服务。

使用 Visual Studio 和 Git 将 ASP.NET Core 持续部署到 Azure
了解如何使用 Visual Studio 创建 ASP.NET Core Web 应用并使用 Git 将它部署到 Azure 应用服务以实现持续部署。

使用 Azure Pipelines 创建你的第一个管道
为 ASP.NET Core 应用设置 CI 生成，然后创建针对 Azure 应用服务的持续部署发布。

Azure Web 应用沙盒
探索 Azure Apps 平台强制实施的 Azure 应用服务运行时执行限制。

应用程序配置

平台

64 位 (x64) 和 32 位 (x86) 应用的运行时存在于 Azure 应用服务上。应用服务上提供的 .NET Core SDK 为 32 位，但可以使用 Kudu 控制台或通过使用 Visual Studio 发布配置文件或 CLI 命令的 MSDeploy 来部署 64 位应用。

对于具有本机依赖项的应用，32 位 (x86) 应用的运行时存在于 Azure 应用服务上。应用服务上提供的 .NET Core SDK 为 32 位。

包

包含以下 NuGet 包，以便为部署到 Azure 应用服务的应用提供自动日志记录功能:

- Microsoft.AspNetCore.AzureAppServices.HostingStartup 使用 IHostingStartup 提供 ASP.NET Core 与 Azure 应用服务的启动集成。添加的日志记录功能由 `Microsoft.AspNetCore.AzureAppServicesIntegration` 包提供。
- Microsoft.AspNetCore.AzureAppServicesIntegration 执行 AddAzureWebAppDiagnostics，在 `Microsoft.Extensions.Logging.AzureAppServices` 包中添加 Azure 应用服务诊断日志记录提供程序。
- Microsoft.Extensions.Logging.AzureAppServices 提供记录器实现，支持 Azure 应用服务诊断日志和日志流式处理功能。

Microsoft.AspNetCore.App 元包中不提供前述包。面向 .NET Framework 或引用 `Microsoft.AspNetCore.App` 元包的应用必须显式引用应用的项目文件中的各个包。

使用 Azure 门户重写应用配置

Azure 门户中的应用设置允许为应用设置环境变量。可以通过环境变量配置提供程序来使用环境变量。

在 Azure 门户中创建或修改应用设置并选择“保存”按钮时，Azure 应用将重启。服务重启后，应用即可使用环境变量。

当应用使用 Web 主机并使用 WebHost.CreateDefaultBuilder 构建主机时，配置主机的环境变量使用 `ASPNETCORE_` 前缀。有关详细信息，请参阅 ASP.NET Core Web 主机 和环境变量配置提供程序。

当应用使用通用主机时，环境变量在默认情况下不会加载到应用的配置，且配置提供程序必须由开发人员添加。在添加配置提供程序时，开发人员将确定环境变量前缀。有关详细信息，请参阅 .NET 通用主机 和环境变量配置提供程序。

代理服务器和负载均衡器方案

配置转发头中间件的 IIS 集成中间件和 ASP.NET Core 模块将配置为转发方案 (HTTP/HTTPS) 和发出请求的远程 IP 地址。对于托管在其他代理服务器和负载均衡器后方的应用，可能需要附加配置。有关详细信息，请参阅配置 ASP.NET Core 以使用代理服务器和负载均衡器。

监视和日志记录

部署到应用服务的 ASP.NET Core 应用自动接收应用服务扩展“ASP.NET Core 日志记录扩展”。此扩展启用 Azure 日志记录。

有关监视、日志记录和故障排除的信息，请参阅以下文章：

如何：在 Azure 应用服务中监视应用
了解如何查看应用和应用服务计划的配额和指标。

在 Azure 应用服务中为应用启用诊断日志记录
了解如何启用和访问 HTTP 状态代码、失败请求和 Web 服务器活动的诊断日志记录。

处理 ASP.NET Core 中的错误
了解在 ASP.NET Core 应用中处理错误的常见方法。

对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除
了解如何使用 ASP.NET Core 应用诊断 Azure 应用服务部署问题。

Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考
使用故障排除建议查看 Azure 应用服务/IIS 托管的应用的常见部署配置错误。

数据保护密钥环和部署槽位

数据保护密钥保存在 %HOME%\ASP.NET\DataProtection-Keys 文件夹中。此文件夹由网络存储提供支持，并跨托管应用的所有计算机同步。密钥不是静态保护的。此文件夹向单个部署槽位中应用的所有实例提供密钥环。各部署槽位（例如过渡槽和生成槽）不共享密钥环。

在部署槽位之间交换时，使用数据保护的任意系统都无法使用之前槽位中的密钥环来解密存储的数据。ASP.NET Cookie 中间件使用数据保护来保护其 cookie。这导致用户注销使用标准 ASP.NET Cookie 中间件的应用。对于独立于槽位的密钥环解决方案，请使用外部密钥环提供程序，例如：

- Azure Blob 存储
- Azure Key Vault
- SQL 存储
- Redis 缓存

有关更多信息，请参见在 ASP.NET Core 中的密钥存储提供程序。

将 ASP.NET Core 预览版部署到 Azure 应用服务

请使用以下方法之一：

- 安装预览站点扩展。
- 部署自包含应用。
- 对用于容器的 Web 应用使用 Docker。

安装预览站点扩展

如果使用预览站点扩展时遇到问题，请在 GitHub 上打开相应的问题。

1. 从 Azure 门户导航到“应用服务”。
2. 选择 Web 应用。
3. 在搜索框中键入“ex”以筛选“扩展”，或向下滚动管理工具列表。
4. 选择“扩展”。
5. 选择“添加”。
6. 从列表选择“ASP.NET Core {X.Y} ({x64|x86}) 运行时”扩展，其中 `{X.Y}` 是 ASP.NET Core 预览版本，并且 `{x64|x86}` 指定平台。
7. 选择“确定”以接受法律条款。
8. 选择“确定”安装扩展。

操作完成时，即表示已安装最新的 .NET Core 预览版。验证安装：

1. 选择“高级工具”。
2. 选择“高级工具”中的“Go”。
3. 选择“调试控制台” > “PowerShell”菜单项。
4. 从 PowerShell 命令提示符处执行以下命令。在以下命令中，将 ASP.NET Core 运行时版本替换为 `{X.Y}`，并将平台替换为 `{PLATFORM}`：

   ```
   Test-Path D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.{PLATFORM}\
   ```

   如果安装 x64 预览版运行时，该命令将返回 `True`。

> **NOTE**
>
> 对于在 A 系列计算机上或更高级托管层上托管的应用，可在 Azure 门户中的应用设置中设置应用服务应用的平台体系结构 (x86/x64)。如果应用在进程内模式下运行并且平台体系结构配置为 64 位 (x64)，则 ASP.NET Core 模块会使用 64 位预览版运行时（如存在）。安装“ASP.NET Core {X.Y} (x64) 运行时”扩展。
>
> 安装 x64 预览版运行时后，在 Kudu PowerShell 命令窗口中运行以下命令以验证该安装。在以下命令中，将 ASP.NET Core 运行时版本替换为 `{X.Y}`：
>
> ```
> Test-Path D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64\
> ```
>
> 如果安装 x64 预览版运行时，该命令将返回 `True`。

> **NOTE**
> ASP.NET Core 扩展可为 Azure 应用服务上的 ASP.NET Core 启用附加功能，例如启用 Azure 日志记录。从 Visual Studio 进行部署时，将自动安装该扩展。如果未安装该扩展，请为应用安装它。

通过 ARM 模板使用预览站点扩展

如果使用 ARM 模板创建和部署应用，则可使用 `siteextensions` 资源类型将站点扩展添加到 Web 应用。例如：

```
{
    "type": "siteextensions",
    "name": "AspNetCoreRuntime",
    "apiVersion": "2015-04-01",
    "location": "[resourceGroup().location]",
    "properties": {
        "version": "[parameters('aspnetcoreVersion')]"
    },
    "dependsOn": [
        "[resourceId('Microsoft.Web/Sites', parameters('siteName'))]"
    ]
}
```

部署自包含应用

针对预览运行时的自包含部署 (SCD)在部署中承载预览运行时。

部署自包含应用时：

- Azure 应用服务中的站点不需要预览站点扩展名。
- 必须使用不同于发布依赖框架部署 (FDD) 的方法发布应用。

使用 Visual Studio 发布

1. 从 Visual Studio 工具栏中选择“构建” > “发布{应用程序名称}”。
2. 在“选择发布目标”对话框中，确认已选中“应用服务”。
3. 选择“高级”。随即会打开“发布”对话框。
4. 在“发布”对话框中：
   - 确认已选中“发布”配置。
   - 打开“部署模式”下拉列表，然后选择“自包含”。
   - 从“目标运行时”下拉列表中选择目标运行时。默认值为 `win-x86`。
   - 如果需要在部署时删除其他文件，请打开“文件发布选项”，然后选中复选框以删除目标位置的其他文件。
   - 选择“保存”。
5. 按照发布向导的其余提示创建新站点或更新现有站点。

使用命令行接口 (CLI) 工具发布

1. 在项目文件中，指定一个或多个运行时标识符 (RID)。对单个 RID 使用 `<RuntimeIdentifier>`（单数），或使用 `<RuntimeIdentifiers>`（复数）提供以分号分隔的 RID 列表。在以下示例中，指定 `win-x86` RID：

   ```
   <PropertyGroup>
     <TargetFramework>netcoreapp2.1</TargetFramework>
     <RuntimeIdentifier>win-x86</RuntimeIdentifier>
   </PropertyGroup>
   ```

2. 在命令 shell 中，使用 dotnet publish 命令在主机运行时的发布配置中发布应用。在以下示例中，为 `win-x86` RID 发布应用。提供给 `--runtime` 选项的 RID 必须在项目文件的 `<RuntimeIdentifier>`（或

`<RuntimeIdentifiers>`）属性中提供。

```
dotnet publish --configuration Release --runtime win-x86
```

3. 将 bin/Release/{TARGET FRAMEWORK}/{RUNTIME IDENTIFIER}/publish 目录的内容移动到应用服务中的站点。

对用于容器的 Web 应用使用 Docker

Docker 中心包含最新的预览 Docker 映像。这些映像可以用作基础映像。按常规方法使用映像并部署到用于容器的 Web 应用。

协议设置 (HTTPS)

借助安全的协议绑定，可在通过 HTTPS 响应请求时指定要使用的证书。若要绑定，需要一个为特定主机名颁发的有效专用证书 (*.pfx*)。有关详细信息，请参阅教程：将现有自定义 SSL 证书绑定到 Azure Web 应用。

其他资源

- Web 应用概述（5 分钟概述视频）
- Azure 应用服务：托管 .NET 应用的最佳位置（55 分钟概览视频）
- Azure Friday：Azure 应用服务诊断和疑难解答体验（12 分钟视频）
- Azure 应用服务诊断概述
- 在 Web 场中托管 ASP.NET Core

Windows Server 上的 Azure 应用服务使用 Internet Information Services (IIS)。以下是基础 IIS 技术的相关主题：

- 使用 IIS 在 Windows 上托管 ASP.NET Core
- ASP.NET Core 模块
- ASP.NET Core 模块
- IIS 模块与 ASP.NET Core
- Microsoft TechNet 库：Windows Server

使用 Visual Studio 将 ASP.NET Core 应用发布到 Azure

2019/1/10 • • Edit Online

作者：Rick Anderson、Cesar Blum Silveira

> **IMPORTANT**
>
> **Azure 应用服务中的 ASP.NET Core 预览版**
>
> 默认情况下不会将 ASP.NET Core 预览版部署到 Azure 应用服务。要托管使用 ASP.NET Core 预览版的应用，请参阅将 ASP.NET Core 预览版部署到 Azure 应用服务。

如果你在 macOS 上工作，请参阅从 Visual Studio for Mac 发布到 Azure。

若要对应用服务部署问题进行故障排除，请参阅 对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除。

设置

- 如果没有免费 Azure 帐户，请开设一个。

创建 Web 应用

在 Visual Studio 起始页中，选择“文件”>“新建”>“项目...”

填写“新建项目”对话框：

- 在左侧窗格中，选择“.NET Core”。
- 在中间窗格中，选择“ASP.NET Core Web 应用程序”。

- 选择“确定”。

在“新建 ASP.NET Core Web 应用程序”对话框中：

- 选择“Web 应用程序”。
- 选择“更改身份验证”。

“更改身份验证”对话框随即出现。

- 选择“个人用户帐户”。

- 选择"确定"返回到"新建 ASP.NET Core Web 应用程序"，然后再次选择"确定"。

Visual Studio 随即创建解决方案。

运行应用

- 按 Ctrl+F5 运行项目。
- 测试"关于"和"联系"链接。

注册用户

- 选择"注册"并注册新用户。可使用虚构电子邮件地址。提交时，页面上会显示以下错误：

  "内部服务器错误:处理请求时，数据库操作失败。SQL 异常:无法打开数据库。可通过向应用程序数据库上下文应用现有迁移解决此问题。"

- 选择"应用迁移"，并在页面更新后刷新页面。

应用将显示用于注册新用户的电子邮件和一个"注销"链接。

将应用部署到 Azure

在解决方案资源管理器中右键单击该项目，然后选择“发布...”。

在“发布”对话框中：

- 选择“Microsoft Azure 应用服务”。
- 选择齿轮图标，然后选择“创建配置文件”。
- 选择“创建配置文件”。

创建 Azure 资源

“创建应用服务”对话框随即显示：

- 输入订阅。
- “应用名称”、“资源组”和“应用服务计划”输入字段已填充。可以保留这些名称，也可以进行更改。

- 选择"服务"选项卡以创建新的数据库。
- 选择绿色的 + 图标以创建新的 SQL 数据库

- 在“配置 SQL 数据库”对话框中选择“新建...”以创建新的数据库。

“配置 SQL Server”对话框随即出现。

- 输入管理员用户名和密码，然后选择“确定”。可保留默认的“服务器名称”。

> **NOTE**
>
> 不可使用“admin”作为管理员用户名。

- 选择"确定"。

Visual Studio 将返回到"创建应用服务"对话框。

- 选择"创建应用服务"对话框上的"创建"。

Visual Studio 在 Azure 上创建 Web 应用和 SQL Server。此步骤可能需要几分钟。有关创建的资源的信息，请参阅其他资源。

部署完成时，选择“设置”：

在“发布”对话框的“设置”页面上：

- 展开“数据库”并选中“在运行时使用此连接字符串”。
- 展开“Entity Framework 迁移”并选中“在发布时应用此迁移”。
- 选择“保存”。Visual Studio 将返回到“发布”对话框。

单击“发布” 。Visual Studio 将应用发布到 Azure。部署完成时，应用在浏览器中打开。

在 Azure 中测试应用

- 测试“关于”和“联系”链接
- 注册新用户

更新应用

- 编辑"Pages/About.cshtml"Razor 页面并更改其内容。例如，可以将段落修改为显示"Hello ASP.NET Core!"：

```
@page
@model AboutModel
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"]</h2>
<h3>@Model.Message</h3>

<p>Hello ASP.NET Core!</p>
```

- 右键单击项目，然后再次选择"发布…"。

- 应用发布后，验证所做的更改在 Azure 上是否可用。

清理

完成应用测试后，转到 Azure 门户并删除该应用。

- 选择"资源组"，然后选择所创建的资源组。

- 在"资源组"页面中，选择"删除"。

- 输入资源组的名称并选择“删除”。现已从 Azure 中删除了本教程中创建的应用和其他所有资源。

后续步骤

- 使用 Visual Studio 和 Git 将 ASP.NET Core 持续部署到 Azure

其他资源

- Azure 应用服务
- Azure 资源组
- Azure SQL 数据库
- 用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件
- 对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除

使用 Visual Studio 和 Git 将 ASP.NET Core 持续部署到 Azure

2019/1/10 • • Edit Online

作者：Erik Reitan

> **IMPORTANT**
>
> **Azure 应用服务中的 ASP.NET Core 预览版**
>
> 默认情况下不会将 ASP.NET Core 预览版部署到 Azure 应用服务。要托管使用 ASP.NET Core 预览版的应用，请参阅将 ASP.NET Core 预览版部署到 Azure 应用服务。

本教程演示如何使用 Visual Studio 创建 ASP.NET Core Web 应用并使用持续部署将其从 Visual Studio 部署到 Azure 应用服务。

另请参阅使用 Azure Pipelines 创建你的第一个管道，该文档介绍如何使用 Azure DevOps Services 为 Azure 应用服务配置持续交付 (CD) 工作流。Azure Pipelines（一种 Azure DevOps Services 服务）简化了可靠部署管道的设置，便于为 Azure 应用服务内托管的应用发布更新。可以从 Azure 门户配置管道以生成、运行测试、部署到过渡槽，然后部署到生产。

> **NOTE**
>
> 若要完成本教程，需要一个 Microsoft Azure 帐户。要获取帐户，可激活 MSDN 订阅者权益或注册免费试用版。

系统必备

本教程假定已安装以下软件：

- Visual Studio
- .NET Core SDK 2.0 或更高版本
- 用于 Windows 的 Git

创建 ASP.NET Core Web 应用

1. 启动 Visual Studio。
2. 从“文件”菜单中选择“新建” > “项目”。
3. 选择“ASP.NET Core Web 应用程序”项目模板。它出现在“已安装” > “模板” > “Visual C#” > “.NET Core”下。将项目命名为 `SampleWebAppDemo`。选择“新建 Git 存储库”选项，然后单击“确定”。

4. 在“新建 ASP.NET Core 项目”对话框中，选择 ASP.NET Core 的“空”模板，然后单击“确定”。

> **NOTE**
>
> .NET Core 的最新版本为 2.0。

本地运行 Web 应用

1. Visual Studio 完成创建应用后，请选择“调试” > “启动调试”以运行该应用。作为替代方法，也可以按 F5。

   可能需要一些时间对 Visual Studio 和新应用进行初始化。完成后，浏览器将显示正在运行的应用。

2. 查看完正在运行的 Web 应用后，关闭浏览器，然后选择 Visual Studio 工具栏中的“停止调试”图标以停止应用。

在 Azure 门户中创建 Web 应用

按照以下步骤在 Azure 门户中创建 Web 应用：

1. 登录到 Azure 门户。
2. 选择门户界面左上方的“新建”。
3. 选择“Web + 移动” > “Web 应用”。

4. 在“Web 应用”边栏选项卡中，输入“应用服务名称”的唯一值。

> **NOTE**
> “应用服务名称”的名称必须是唯一的。门户在提供名称时强制执行此规则。在提供一个不同的值后，需要为在本教程中每次出现的“SampleWebAppDemo”替换该值。

另外，在“Web 应用”边栏选项卡中，选择现有“应用服务计划/位置”或新建一个。如果创建新的计划，请选择“定价层”、“位置”和其他选项。有关应用服务计划的详细信息，请参阅 Azure App Service 计划深入概述。

5. 选择“创建”。Azure 将预配并启动 Web 应用。

为新 Web 应用启用 Git 发布

Git 是一个分布式版本控制系统，可用来部署 Azure App Service Web 应用。Web 应用代码存储在本地 Git 存储库中，并通过推送到远程存储库将代码部署到 Azure。

1. 登录到 Azure 门户。

2. 选择“应用服务”查看与 Azure 订阅关联的应用服务列表。

3. 选择在本教程的前一部分中创建的 Web 应用。

4. 在“部署”边栏选项卡中，选择“部署选项” > “选择源” > “本地 Git 存储库”。

5. 选择“确定”。

6. 如果事先未设置用于发布 Web 应用的部署凭据或其他应用服务应用，请立即设置：

   - 选择“设置” > “部署凭据”。“设置部署凭据”边栏选项卡将显示。
   - 创建用户名和密码。在设置 Git 时保存密码供以供将来使用。
   - 选择“保存”。

7. 在“Web 应用”边栏选项卡中，选择“设置” > “属性”。要部署到的远程 Git 存储库的 URL 会显示在“GIT URL”下。

8. 复制“GIT URL”的值，稍后将在本教程中用到。

将 Web 应用发布到 Azure App Service

在本部分中，使用 Visual Studio 创建本地 Git 存储库并从该存储库推送到 Azure 以部署 Web 应用。涉及到的步骤如下：

- 使用 GIT URL 值添加远程存储库设置，这样本地存储库就可部署到 Azure。
- 提交项目更改。
- 将项目更改从本地存储库推送到 Azure 上的远程存储库。

1. 在“解决方案资源管理器”中右键单击“解决方案 ‘SampleWebAppDemo’”并选择“提交”。此时将显示“团队资源管理器”。

2. 在“团队资源管理器”中，选择“主页”(主页图标)>“设置” > “存储库设置”。

3. 在“存储库设置”的“远程”部分中选择“添加”。将显示“添加远程”对话框。

4. 将远程的“名称”设置为“Azure-SampleApp”。

5. 将“提取”的值设置为在本教程前面部分从 Azure 复制的“Git URL”。请注意，此 URL 是以 .git 结尾的。

> **NOTE**
>
> 作为替代方法，可以通过打开“命令窗口”、更改为项目目录并输入命令从“命令窗口”指定远程存储库。示例：
>
> ```
> git remote add Azure-SampleApp https://me@sampleapp.scm.azurewebsites.net:443/SampleApp.git
> ```

6. 选择“主页”(主页图标)>“设置” > “全局设置”。确认名称和电子邮件地址已设置。如有必要，则选择“更新”。

7. 选择“主页” > “更改”以返回到“更改”视图。

8. 输入提交消息，如“初始推送 #1”，然后选择“提交”。此操作将本地创建“提交”。

> **NOTE**
>
> 作为替代方法，可以通过打开“命令窗口”、更改为项目目录并输入 git 命令从“命令窗口”提交更改。示例：
>
> `git add .`
>
> `git commit -am "Initial Push #1"`

9. 选择“主页” > “同步” > “操作” > “打开命令提示符”。命令提示符打开到项目目录。

10. 在命令窗口中输入以下命令：

    ```
    git push -u Azure-SampleApp master
    ```

11. 输入之前在 Azure 中创建的 Azure“部署凭据”密码。

    此命令将启动将本地项目文件推送到 Azure 的进程。上述命令的输出以部署成功的消息结尾。

    ```
    remote: Finished successfully.
    remote: Running post deployment command(s)...
    remote: Deployment successful.
    To https://username@samplewebappdemo01.scm.azurewebsites.net:443/SampleWebAppDemo01.git
    * [new branch]      master -> master
    Branch master set up to track remote branch master from Azure-SampleApp.
    ```

> **NOTE**
>
> 如果项目需要协作，请考虑推送到 GitHub，然后再推送到 Azure。

验证活动的部署

验证 Web 应用从本地环境传输到 Azure 是否成功。

在 Azure 门户中，选择“Web 应用”。然后，选择“部署” > “部署选项”。

在 Azure 中运行此应用

现在该 Web 应用已部署到 Azure，请运行该应用。

这可以通过以下两种方式实现：

- 在 Azure 门户中，找到 Web 应用的“Web 应用”边栏选项卡。选择“浏览”以在默认浏览器中查看该应用程序。
- 打开浏览器并输入 Web 应用的 URL。示例：`http://SampleWebAppDemo.azurewebsites.net`

更新 Web 应用并重新发布

更改本地代码后重新发布：

1. 在 Visual Studio 的“解决方案资源管理器”中，打开 Startup.cs 文件。
2. 在 `Configure` 方法中，修改 `Response.WriteAsync` 方法，使它显示以下内容：

```
await context.Response.WriteAsync("Hello World! Deploy to Azure.");
```

3. 将更改保存到 Startup.cs。
4. 在“解决方案资源管理器”中右键单击“解决方案 ‘SampleWebAppDemo’”并选择“提交”。此时将显示“团队资源管理器”。
5. 输入提交消息，如 `Update #2`。
6. 按“提交”按钮以提交项目更改。
7. 选择“主页” > “同步” > “操作” > “推送”。

> **NOTE**
>
> 作为替代方法，可以通过打开“命令窗口”、更改为项目目录并输入 git 命令从“命令窗口”推送更改。示例：
>
> `git push -u Azure-SampleApp master`

在 Azure 中查看更新的 Web 应用

通过从 Azure 门户中的“Web 应用”边栏选项卡选择“浏览”，或通过打开浏览器并输入 Web 应用的 URL 以查看更

新的 Web 应用。示例：`http://SampleWebAppDemo.azurewebsites.net`

其他资源

- 使用 Azure Pipelines 创建你的第一个管道
- 项目 Kudu
- 用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件

对 Azure 应用服务上的 ASP.NET Core 进行故障排除

2019/1/10 • • Edit Online

作者：Luke Latham

> **IMPORTANT**
>
> **Azure 应用服务中的 ASP.NET Core 预览版**
>
> 默认情况下不会将 ASP.NET Core 预览版部署到 Azure 应用服务。要托管使用 ASP.NET Core 预览版的应用，请参阅将 ASP.NET Core 预览版部署到 Azure 应用服务。

本文说明了如何使用 Azure 应用服务的诊断工具诊断 ASP.NET Core 应用启动问题。有关其他故障排除建议，请参阅 Azure 文档中的 Azure 应用服务诊断概述和如何：在 Azure 应用服务中监视应用。

应用启动错误

502.5 进程故障
工作进程失败。应用不启动。

ASP.NET Core 模块尝试启动工作进程，但启动失败。检查应用程序事件日志通常可帮助解决此类型的问题。应用程序事件日志部分中介绍了访问日志。

配置错误的应用导致工作进程失败时，将返回“502.5 进程故障”错误页面：

500 内部服务器错误
应用启动，但某个错误阻止了服务器完成请求。

在启动期间或在创建响应时，应用的代码内出现此错误。响应可能不包含任何内容，或响应可能会在浏览器中显示为“500 内部服务器错误”。应用程序事件日志通常表明应用正常启动。从服务器的角度来看，这是正确的。应用已启动，但无法生成有效的响应。在 Kudu 控制台中运行应用或启用 ASP.NET Core 模块 stdout 日志以解决该问题。

连接重置

如果在发送标头后出现错误，则服务器在出现错误时发送“500 内部服务器错误”已经太晚了。通常在序列化响应

的复杂对象期间出现错误时发生这种情况。此类型的错误在客户端上显示为“连接重置”错误。应用程序日志记录可以帮助解决这些类型的错误。

默认启动限制

ASP.NET Core 模块的默认“startupTimeLimit”配置为 120 秒。保留默认值时，在模块记录进程故障之前，可能最多需要两分钟来启动应用。有关配置模块的信息，请参阅 aspNetCore 元素的属性。

解决应用启动错误

应用程序事件日志

若要访问应用程序事件日志，请在 Azure 门户中使用“诊断并解决问题”边栏选项卡：

1. 在 Azure 门户中，打开“应用服务”边栏选项卡中的应用边栏选项卡。
2. 选择“诊断并解决问题”边栏选项卡。
3. 在“选择问题类别”下，选择“Web 应用关闭”按钮。
4. 在“建议的解决方案”下，打开“打开应用程序事件日志”对应的窗格。选择“打开应用程序事件日志”按钮。
5. 检查“源”列中由 IIS AspNetCoreModule 提供的最新错误。

使用“诊断并解决问题”边栏选项卡的替代方法是直接使用 Kudu 检查应用程序事件日志文件：

1. 选择“开发工具”区域中的“高级工具”边栏选项卡。选择“转到→”按钮。此时将在新的浏览器选项卡或窗口中打开 Kudu 控制台。
2. 使用页面顶部的导航栏，打开“调试控制台”并选择“CMD”。
3. 打开 LogFiles 文件夹。
4. 选择 eventlog.xml 文件旁边的铅笔图标。
5. 检查日志。滚动到日志底部以查看最新事件。

在 Kudu 控制台中运行应用

许多启动错误未在应用程序事件日志中生成有用信息。可以在 Kudu 远程执行控制台中运行应用以发现错误：

1. 选择“开发工具”区域中的“高级工具”边栏选项卡。选择“转到→”按钮。此时将在新的浏览器选项卡或窗口中打开 Kudu 控制台。
2. 使用页面顶部的导航栏，打开“调试控制台”并选择“CMD”。
3. 打开路径“site > wwwroot”下的文件夹。
4. 在控制台中，通过执行应用的程序集来运行该应用。
   - 如果应用是依赖框架的部署，则使用 dotnet.exe 运行应用的程序集。在以下命令中，替换 `<assembly_name>` 的应用程序集的名称：`dotnet .\<assembly_name>.dll`
   - 如果应用是独立部署，则运行应用的可执行文件。在以下命令中，替换 `<assembly_name>` 的应用程序集的名称：`<assembly_name>.exe`
5. 来自应用且显示任何错误的控制台输出将传送到 Kudu 控制台。

ASP.NET Core 模块 stdout 日志

ASP.NET Core 模块 stdout 日志通常记录应用程序事件日志中找不到的有用错误消息。若要启用和查看 stdout 日志，请执行以下操作：

1. 在 Azure 门户中导航到“诊断并解决问题”边栏选项卡。
2. 在“选择问题类别”下，选择“Web 应用关闭”按钮。
3. 在“建议的解决方案” > “启用 Stdout 日志重定向”下，选择“打开 Kudu 控制台以编辑 Web.Config”对应的按钮。
4. 在 Kudu 诊断控制台中，打开路径“站点 > wwwroot”下的文件夹。向下滚动以在列表底部显示“web.config”文件。
5. 单击“web.config”文件旁边的铅笔图标。

6. 将“stdoutLogEnabled”设置为 `true`，并将“stdoutLogFile”路径更改为 `\\?\%home%\LogFiles\stdout`。
7. 选择“保存”以保存已更新的 web.config 文件。
8. 向应用发出请求。
9. 返回到 Azure 门户。选择“开发工具”区域中的“高级工具”边栏选项卡。选择“转到→”按钮。此时将在新的浏览器选项卡或窗口中打开 Kudu 控制台。
10. 使用页面顶部的导航栏，打开“调试控制台”并选择“CMD”。
11. 选择“LogFiles”文件夹。
12. 检查“已修改”列并选择铅笔图标以编辑具有最新修改日期的 stdout 日志。
13. 打开日志文件后，将显示错误。

重要提示！ 故障排除完成后，禁用 stdout 日志记录。

1. 在 Kudu 诊断控制台中，返回到路径“site > wwwroot”以显示 web.config 文件。通过选择铅笔图标再次打开 web.config 文件。
2. 将“stdoutLogEnabled”设置为 `false`。
3. 选择“保存”以保存文件。

> **WARNING**
>
> 无法禁用 stdout 日志可能会导致应用或服务器出现故障。日志文件大小或创建的日志文件数没有限制。仅使用 stdout 日志记录来解决应用启动问题。
>
> 对于在 ASP.NET Core 应用启动后生成的常规日志记录，使用限制日志文件大小和旋转日志的日志记录库。有关详细信息，请参阅第三方日志记录提供程序。

常见启动错误

请参阅 Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考。参考主题介绍了阻止应用启动的大部分常见问题。

应用缓慢或挂起

当应用响应缓慢或挂起请求时，请参阅解决 Azure 应用服务中 Web 应用性能缓慢的问题以获取调试指导。

远程调试

请参见下面的主题：

- 使用 Visual Studio 对 Azure 应用服务中的 Web 应用进行故障排除的远程调试 Web 应用部分（Azure 文档）
- 在 Visual Studio 2017 的 Azure 中的 IIS 上远程调试 ASP.NET Core（Visual Studio 文档）

Application Insights

Application Insights 提供来自 Azure 应用服务中托管的应用的遥测，包括错误日志记录和报告功能。当应用的日志记录功能变得可用时，Application Insights 只能报告应用启动后出现的错误。有关详细信息，请参阅用于 ASP.NET Core 的 Application Insights。

监视边栏选项卡

监视边栏选项卡提供了本主题前面所述的方法的替代故障排除体验。这些边栏选项卡可用于诊断 500 系列错误。

确认是否已安装 ASP.NET Core 扩展。如果未安装扩展，请手动进行安装：

1. 在“开发工具”边栏选项卡部分中，选择“扩展”边栏选项卡。
2. “ASP.NET Core 扩展”应显示在列表中。
3. 如果未安装扩展，请选择“添加”按钮。
4. 从列表中选择“ASP.NET Core 扩展”。
5. 选择“确定”以接受法律条款。
6. 选择“添加扩展”边栏选项卡上的“确定”。
7. 信息性弹出消息指示成功安装扩展的时间。

如果未启用 stdout 日志记录，请执行以下步骤：

1. 在 Azure 门户中，选择“开发工具”区域中的“高级工具”边栏选项卡。选择“转到→”按钮。此时将在新的浏览器选项卡或窗口中打开 Kudu 控制台。
2. 使用页面顶部的导航栏，打开“调试控制台”并选择“CMD”。
3. 打开路径“site > wwwroot”下的文件夹，然后向下滚动以显示列表底部的 web.config 文件。
4. 单击“web.config”文件旁边的铅笔图标。
5. 将“stdoutLogEnabled”设置为 `true`，并将“stdoutLogFile”路径更改为 `\\?\%home%\LogFiles\stdout`。
6. 选择“保存”以保存已更新的 web.config 文件。

继续激活诊断日志记录：

1. 在 Azure 门户中，选择“诊断日志”边栏选项卡。
2. 选择“应用程序日志记录(文件系统)”和“详细错误消息”的“开”开关。选择边栏选项卡顶部的“保存”按钮。
3. 若要包含失败请求跟踪（也称为失败请求事件缓冲 (FREB) 日志记录），请选择“失败请求跟踪”的“开”开关。
4. 选择“日志流”边栏选项卡，将在门户中的“诊断日志”边栏选项卡下立即列出。
5. 向应用发出请求。
6. 在日志流数据中，指示了错误的原因。

重要提示！ 故障排除完成后，请务必禁用 stdout 日志记录。请参阅 ASP.NET Core 模块 stdout 日志部分中的说明。

若要查看失败请求跟踪日志（FREB 日志），请执行以下操作：

1. 在 Azure 门户中导航到“诊断并解决问题”边栏选项卡。
2. 从侧栏的“支持工具”区域中选择“失败请求跟踪日志”。

有关详细信息，请参阅“在 Azure 应用服务中启用 Web 应用的诊断日志记录”主题的“失败请求跟踪”部分和 Azure 中的 Web 应用的应用程序性能常见问题：如何打开失败请求跟踪？。

有关详细信息，请参阅在 Azure 应用服务中启用 Web 应用的诊断日志记录。

> **WARNING**
>
> 无法禁用 stdout 日志可能会导致应用或服务器出现故障。日志文件大小或创建的日志文件数没有限制。
>
> 对于 ASP.NET Core 应用中的例程日志记录，使用限制日志文件大小和旋转日志的日志记录库。有关详细信息，请参阅第三方日志记录提供程序。

其他资源

- 处理 ASP.NET Core 中的错误
- Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考
- 使用 Visual Studio 对 Azure 应用服务中的 Web 应用进行故障排除
- 解决 Azure Web 应用中的“502 错误的网关”和“503 服务不可用”HTTP 错误

- 解决 Azure 应用服务中 Web 应用性能缓慢的问题
- Azure 中的 Web 应用的应用程序性能常见问题
- Azure Web 应用沙盒（应用服务运行时执行限制）
- Azure Friday：Azure 应用服务诊断和疑难解答体验（12 分钟视频）

通过 ASP.NET Core 和 Azure 实现 DevOps

2018/12/10 • • Edit Online

作者：Cam SoperScott Addie

本指南以可下载 PDF 电子书的形式提供。

欢迎使用

欢迎使用适用于 .NET 的 Azure 开发生命周期指南！ 本指南介绍使用 .NET 工具和进程围绕 Azure 构建开发生命周期的基本概念。读完本指南后，可以获得成熟 DevOps 工具链的优势。

本指南适用对象

读者应是有经验的 ASP.NET Core 开发人员（200-300 级别）。无须了解有关 Azure 的任何内容，因为本指南中将进行介绍。本指南对于更侧重于操作（而不是开发）的 DevOps 工程师可能也很有用。

本指南面向 Windows 开发人员。但是，.NET Core 完全支持 Linux 和 macOS。若要针对 Linux/macOS 调整本指

南，请注意有关 Linux/macOS 差异的标注。

本指南未涵盖的内容

本指南专注于 .NET 开发人员的端到端持续部署体验。这不是介绍 Azure 中所有内容的全面指南，并不特别着重于适用于 Azure 服务的 .NET API。介绍的重点在于持续集成、部署、监视和调试。本指南结尾附近提供了后续步骤建议。建议包括对 ASP.NET Core 开发人员有用的 Azure 平台服务。

本指南内容

工具和下载

了解在何处获取本指南中使用的工具。

部署到应用服务

了解将 ASP.NET Core 应用部署到 Azure 应用服务的各种方法。

持续集成和部署

通过 GitHub、Azure DevOps Services 和 Azure，为 ASP.NET Core 应用生成端到端持续集成和部署解决方案。

监视和调试

使用 Azure 工具监视、故障排除和优化应用程序。

后续步骤

ASP.NET Core 开发人员学习 Azure 的其他学习路径。

其他介绍性读物

如果这是你首次接触云计算，这些文章介绍了基础知识。

- 什么是云计算？
- 云计算示例
- 什么是 IaaS？
- 什么是 PaaS？

工具和下载

2019/1/2 • • Edit Online

Azure 具有多个用于配置和管理资源的界面，例如 Azure 门户、Azure CLI、Azure PowerShell、Azure Cloud Shell和 Visual Studio。本指南采用极简主义方法，并尽可能使用 Azure Cloud Shell 来减少所需的步骤。但是，对于某些部分必须使用 Azure 门户。

系统必备

以下订阅是必需的：

- Azure—如果还没有帐户，获取免费试用版。
- Azure DevOps 服务 — Azure DevOps 订阅和组织将在第 4 章中创建。
- GitHub—如果还没有帐户，免费注册。

以下工具是必需的：

- Git — 本指南建议应对 Git 有基本的了解。请查看 Git 文档，特别是git remote 和 git push。
- .NET core SDK — 2.1.300 或更高版本是构建和运行示例应用的必需条件。如果 Visual Studio 安装了".NET Core 跨平台开发"工作负荷，则表示已安装了 .NET Core SDK。

  验证 .NET Core SDK 安装。打开命令外壳，并运行以下命令：

  ```
  dotnet --version
  ```

建议的工具 (仅 Windows)

- Visual Studio 可靠的 Azure 工具为本指南中描述的大多数功能提供了GUI。任何版本的 Visual Studio 都可以使用，包括免费的 Visual Studio Community Edition。编写本教程是为了演示在使用和不使用 Visual Studio 的情况下进行开发，部署和 DevOps。

  确认 Visual Studio 具有以下工作负荷安装：

  - ASP.NET 和 Web 开发
  - Azure 开发
  - .NET Core 跨平台开发

将应用部署到应用服务

2018/12/12 • • Edit Online

Azure 应用服务是 Azure 的 web 托管平台。手动或通过自动化过程，可以完成 web 应用部署到 Azure 应用服务。指南的本部分讨论可以手动或使用命令行中，脚本触发或触发手动使用 Visual Studio 的部署方法。

在本部分中，将完成以下任务：

- 下载并生成示例应用程序。
- 创建使用 Azure Cloud Shell 的 Azure 应用服务 Web 应用。
- 将示例应用程序部署到 Azure 中使用 Git。
- 将更改部署到使用 Visual Studio 的应用。
- 将过渡槽添加到 web 应用。
- 将更新部署到过渡槽。
- 交换过渡和生产槽。

下载并测试应用程序

本指南中使用的应用是一个预构建的 ASP.NET Core 应用，简单馈送读取器。它是使用 Razor 页面应用 `Microsoft.SyndicationFeed.ReaderWriter` API 以检索 RSS/Atom 馈送，并在列表中显示的新闻项。

随意查看代码，但请务必了解，没有什么特别此应用之处。正是出于演示目的只是简单的 ASP.NET Core 应用。

从命令行界面，下载的代码、生成项目时，和运行它，如下所示。

> *注意：Linux/macOS 用户应更改相应的路径，例如，使用正斜杠 (`/`) 而不是反斜杠 (`\`)。*

1. 代码克隆到本地计算机上的文件夹。

```
git clone https://github.com/Azure-Samples/simple-feed-reader/
```

2. 更改到您的工作文件夹*简单源阅读器*已创建的文件夹。

```
cd .\simple-feed-reader\SimpleFeedReader
```

3. 还原包，并生成解决方案。

```
dotnet build
```

4. 运行应用。

```
dotnet run
```

5. 打开浏览器并导航到 `http://localhost:5000`。应用程序，可键入或粘贴联合源 URL，并查看新闻项的列表。

6. 感到满意后应用程序是否正常工作，通过按关闭它**Ctrl**+**C**命令行界面中。

创建 Azure 应用服务 Web 应用

若要部署应用程序，你将需要创建应用服务Web 应用。创建后的 Web 应用，你将部署到它从使用 Git 在本地计算机。

1. 登录到Azure Cloud Shell。注意:在登录时第一次，Cloud Shell 会提示创建配置文件的存储帐户。接受默认值或提供唯一的名称。

2. 对于以下步骤使用 Cloud Shell。

   a. 声明一个变量来存储 web 应用程序的名称。名称必须是唯一的以在默认 URL 中使用。使用 `$RANDOM` Bash 函数来构造名称可保证唯一性和结果的格式 `webappname99999`。

   ```
   webappname=mywebapp$RANDOM
   ```

   b. 创建资源组。资源组提供一种方法可聚合 Azure 资源作为一个组进行管理。

```
az group create --location centralus --name AzureTutorial
```

`az` 命令可调用Azure CLI。可以本地运行 CLI，但 Cloud Shell 中使用它可以节省时间和配置。

c. 在 S1 层中创建应用服务计划。应用服务计划是共享相同的定价层的 web 应用的分组。S1 层不是免费的但它具有所需的过渡槽功能。

```
az appservice plan create --name $webappname --resource-group AzureTutorial --sku S1
```

d. 创建应用服务计划使用同一资源组中的 web 应用资源。

```
az webapp create --name $webappname --resource-group AzureTutorial --plan $webappname
```

e. 设置部署凭据。这些部署凭据适用于你的订阅中的所有 web 应用。不在用户名称中使用特殊字符。

```
az webapp deployment user set --user-name REPLACE_WITH_USER_NAME --password REPLACE_WITH_PASSWORD
```

f. 将 web 应用配置为接受来自本地 Git 和显示部署*Git 部署 URL*。记下此 **URL 以供日后参考**。

```
echo Git deployment URL: $(az webapp deployment source config-local-git --name $webappname --resource-group AzureTutorial --query url --output tsv)
```

g. 显示*web 应用 URL*。浏览到此 URL 即可查看空白 web 应用。记下此 **URL 以供日后参考**。

```
echo Web app URL: http://$webappname.azurewebsites.net
```

3. 在本地计算机上使用命令行界面导航到 web 应用的项目文件夹 (例如，`.\simple-feed-reader\SimpleFeedReader`)。执行以下命令以设置 Git 推送到部署 URL:

   a. 将远程 URL 添加到本地存储库。

   ```
   git remote add azure-prod GIT_DEPLOYMENT_URL
   ```

   b. 推送本地*主*到分支*azure prod*远程数据库的*主*分支。

   ```
   git push azure-prod master
   ```

   系统会提示你之前创建的部署凭据。观察到命令行界面中的输出。Azure 远程生成 ASP.NET Core 应用。

4. 在浏览器中，导航到*Web 应用 URL*并记下已生成并部署应用程序。可以将其他的更改提交到本地 Git 存储库具有 `git commit`。这些更改推送到 Azure 使用前面 `git push` 命令。

使用 Visual Studio 部署

> *注意：本部分仅适用于 Windows。Linux 和 macOS 用户应进行下面的步骤 2 中所述的更改。保存该文件，并将更改提交到本地存储库与 `git commit`。最后，推送将更改与 `git push`，如下所示的第一个部分。*

已从命令行界面部署应用程序。让我们使用 Visual Studio 的集成的工具将更新部署到应用。在后台，Visual Studio 的功能完全相同命令行工具，但在 Visual Studio 的熟悉用户界面内。

1. 打开*SimpleFeedReader.sln* Visual Studio 中。

2. 在解决方案资源管理器中打开*Pages\Index.cshtml*。更改 `<h2>Simple Feed Reader</h2>` 到 `<h2>Simple Feed Reader - V2</h2>` 。

3. 按**Ctrl**+**Shift**+**B**生成的应用。

4. 在解决方案资源管理器，右键单击项目并单击发布。

5. Visual Studio 可以创建新的应用服务资源，但此更新将发布对现有部署。在选取发布目标对话框中，选择应用服务从列表中的左侧，然后选择选择现有。单击“发布”。

6. 在中应用服务对话框中，确认已显示在右上方的 Microsoft 或组织帐户用于创建你的 Azure 订阅。如果不是这样，单击下拉列表，并将其添加。

7. 确认正确的 Azure订阅处于选中状态。有关视图，选择资源组。展开**AzureTutorial**资源组，然后选择现有的 web 应用。单击 **“确定”**。

Visual Studio 生成，并将该应用部署到 Azure。浏览到 web 应用 URL。验证 `<h2>` 元素修改处于活动状态。

部署槽

部署槽而不会影响生产环境中运行的应用程序支持暂存的更改。通过质量保证团队验证应用的过渡的版本后，可以交换生产和过渡槽。在过渡环境中的应用提升到生产环境中这种方式。以下步骤创建过渡槽，某些更改部署到它，并交换过渡槽与生产环境完成验证之后。

1. 登录到Azure Cloud Shell，如果尚未登录。

2. 创建过渡槽。

a. 使用名称创建部署槽*暂存*。

```
az webapp deployment slot create --name $webappname --resource-group AzureTutorial --slot staging
```

b. 配置要使用从本地 Git 和获取部署的过渡槽暂存部署 URL。记**下此 URL 以供日后参考**。

```
echo Git deployment URL for staging: $(az webapp deployment source config-local-git --name $webappname -
-resource-group AzureTutorial --slot staging --query url --output tsv)
```

c. 显示过渡槽的 URL。浏览到 URL 即可查看空的过渡槽。记**下此 URL 以供日后参考**。

```
echo Staging web app URL: http://$webappname-staging.azurewebsites.net
```

3. 在文本编辑器或 Visual Studio 中，修改*pages/Index.cshtml*再次以便 `<h2>` 元素中读取 `<h2>Simple Feed Reader - V3</h2>` 并保存该文件。

4. 将文件提交到本地 Git 存储库，使用任一**更改**Visual Studio 中的页*团队资源管理器*选项卡上，或通过输入以下使用本地计算机的命令行界面：

```
git commit -a -m "upgraded to V3"
```

5. 使用本地计算机的命令行界面，作为 Git 远程添加过渡的部署 URL，并推送已提交的更改：

   a. 将过渡环境的远程 URL 添加到本地 Git 存储库。

```
git remote add azure-staging <Git_staging_deployment_URL>
```

   b. 推送本地*主*到分支*azure 过渡环境*远程数据库的*主*分支。

```
git push azure-staging master
```

   等待 Azure 生成并部署应用程序。

6. 若要验证 V3 已部署到过渡槽，请打开两个浏览器窗口。在一个窗口中，导航到原始的 web 应用 URL。在其他窗口中，导航到过渡 web 应用 URL。生产 URL 提供服务的应用程序的 V2。过渡 URL 提供服务的应用程序的 V3。

7. 在 Cloud Shell 中，验证/上做好准备的增加将过渡槽交换到生产环境。

```
az webapp deployment slot swap --name $webappname --resource-group AzureTutorial --slot staging
```

8. 验证交换发生通过刷新两个浏览器窗口。

总结

在本部分中，已完成以下任务：

- 下载并构建示例应用程序。
- 创建使用 Azure Cloud Shell 的 Azure 应用服务 Web 应用。
- 示例应用部署到 Azure 中使用 Git。
- 部署到使用 Visual Studio 应用更改。
- 添加到 web 应用的过渡槽。
- 更新部署到过渡槽中。
- 交换过渡和生产槽。

在下一步部分中，将了解如何构建 DevOps 管道，其中包含 Azure 管道。

其他阅读材料

- Web 应用概述
- 生成 Azure 应用服务中的.NET Core 和 SQL 数据库 web 应用
- 为 Azure 应用服务配置部署凭据
- 设置过渡环境，在 Azure 应用服务

持续集成和部署

2018/12/9 • • Edit Online

在上一章，您创建简单的源读取器应用的本地 Git 存储库。在本章中，会将该代码发布到 GitHub 存储库并构建使用 Azure 管道的 Azure DevOps 服务管道。管道可使持续生成和部署的应用程序。GitHub 存储库的任何提交触发生成和部署到 Azure Web 应用的过渡槽。

在本部分中，将完成以下任务：

- 将应用程序的代码发布到 GitHub
- 断开连接本地 Git 部署
- 创建 Azure DevOps 组织
- 在 Azure DevOps 服务中创建团队项目
- 创建生成定义
- 创建发布管道
- 将更改提交到 GitHub 并自动部署到 Azure
- 检查 Azure 管道管道

将应用程序的代码发布到 GitHub

1. 打开浏览器窗口，并导航到 `https://github.com`。
2. 单击 + 标头中的下拉列表中，选择**新的存储库**：

3. 选择你的帐户**所有者**下拉列表中，并输入*简单源阅读器*中**存储库名称**文本框中。
4. 单击**创建存储库**按钮。
5. 打开本地计算机的命令行界面。导航到在其中的目录*简单源阅读器*存储 Git 存储库。
6. 重命名现有*原点*到远程*上游*。请执行以下命令：

```
git remote rename origin upstream
```

7. 添加一个新*原点*远程指向您的 GitHub 上的存储库副本。请执行以下命令：

```
git remote add origin https://github.com/<GitHub_username>/simple-feed-reader/
```

8. 将本地 Git 存储库发布到新创建的 GitHub 存储库。请执行以下命令：

```
git push -u origin master
```

9. 打开浏览器窗口，并导航到 `https://github.com/<GitHub_username>/simple-feed-reader/` 。验证你的代码会在 GitHub 存储库中。

断开连接本地 Git 部署

删除本地 Git 部署通过执行以下步骤。Azure 管道（Azure DevOps 服务）都将替换和增强该功能。

1. 打开Azure 门户，并导航到 *暂存 (mywebapp<unique_number>/暂存)* Web 应用。可以通过输入快速位于 Web 应用 *暂存*门户的搜索框中：

2. 单击**部署**选项。将显示新面板。单击**断开连接**来删除已添加在上一章中的本地 Git 源控件配置。通过单击**确认删除操作是**按钮。

3. 导航到*mywebapp* < *unique_number* > 应用服务。请注意，可以使用门户的搜索框中快速找到应用服务。

4. 单击**部署**选项。将显示新面板。单击**断开连接**来删除已添加在上一章中的本地 Git 源控件配置。通过单击**确认删除操作是**按钮。

创建 Azure DevOps 组织

1. 打开浏览器并导航到Azure DevOps 的组织创建页。

2. 键入唯一名称选取一个易记名称文本框中以形成用于访问你的 Azure DevOps 组织的 URL。

3. 选择**Git**单选按钮，因为代码托管在 GitHub 存储库。

4. 单击"继续"按钮。稍等片刻、帐户和团队项目之后，名为*MyFirstProject*，创建。

5. 打开指示 Azure DevOps 的组织和项目可供使用的确认电子邮件。单击**开始你的项目**按钮：

6. 浏览器将打开 <*account_name*>。*visualstudio.com*。单击*MyFirstProject*链接以开始配置项目的 DevOps 管道。

配置 Azure 管道管道

有三个不同的步骤才能完成。完成操作的 DevOps 管道中的以下三个部分结果中的步骤。

授予 Azure DevOps 到 GitHub 存储库的访问权限

1. **展开或从外部存**储库**的代**码**生成**accordion。单击**安装程序生成**按钮：

2. 选择**GitHub**选项从选择**一个源**部分：

3. 授权是必需的 Azure DevOps 可以访问 GitHub 存储库之前。输入 < *GitHub_username* > *GitHub 连接*中连接名称文本框中。例如：

4. 如果你的 GitHub 帐户启用双因素身份验证，则需要个人访问令牌。在这种情况下，单击**使用 GitHub 个人访问令牌的授**权链接。请参阅正式 GitHub 个人访问令牌创建说明有关的帮助。仅*存储库*所需权限的作用域。否则，请单击**使用 OAuth 授**权按钮。

5. 出现提示时，登录到你的 GitHub 帐户。然后，选择授权授予对你的 Azure DevOps 组织访问权限。如果成功，会创建新的服务终结点。

6. 单击省略号按钮旁边**存储库**按钮。选择 < *GitHub_username* > / *简单源阅读器*从列表中的存储库。单击选择按钮。

7. 选择*主*从分支手动和计划生成的默认分支下拉列表。单击"继续"按钮。模板选择页会显示。

创建生成定义

1. 从模板选择页中，输入*ASP.NET Core*在搜索框中：

2. 显示模板搜索结果。将鼠标悬停**ASP.NET Core**模板，然后单击应用按钮。

3. 任务显示的生成定义选项卡。单击"触发器" 选项卡。

4. 检查**启用持续集成**框。下**的分支**筛选器部分中，确认类型下拉列表设置为*Include*。设置**分支规范**下拉列表*master*。

这些设置会导致要进行任何更改推送到时触发的生成主GitHub 存储库分支。在中测试持续集成更改提交到 GitHub 并自动部署到 Azure部分。

5. 单击**保存并排队**按钮，然后选择**保存**选项：

6. 下面的模式对话框会显示：

使用的默认文件夹 \，然后单击**保存**按钮。

创建发布管道

1. 单击**版本**你的团队项目的选项卡。单击**新的管道**按钮。

模板选择窗格会显示。

2. 从模板选择页中，输入*应用服务*在搜索框中：

3. 显示模板搜索结果。将鼠标悬停**Azure 应用服务部署槽**模板，然后单击**应用**按钮。**管道**发布管道的选项卡将出现。

4. 单击**外**按钮项**目**框。**添加**项**目**面板将显示：

5. 选择**构建**磁贴**源类型**部分。此类型，用于对生成定义发布管道的链接。
6. 选择*MyFirstProject*从项**目**下拉列表。
7. 选择生成定义名称，*MyFirstProject ASP.NET Core CI*，**从源（生成定义）**下拉列表。
8. 选择*最新*从**默认版本**下拉列表。此选项可用于构建生成的最后一次运行的生成定义的项目。
9. 替换中的文本**源别名**具有 textbox *Drop*。
10. 单击"添加"按钮。项**目**部分更新以显示所做的更改。
11. 单击闪电形图标以启用持续部署：

启用此选项，每次新的生成时可执行部署。

12. 一个**持续部署触**发器面板右侧显示。单击切换按钮以启用该功能。但并不需要启用**拉取请求触**发器。
13. 单击**外**下拉列表中**版本分支**筛选器部分。选择**生成定义的默认分支**选项。此筛选器将导致仅对从 GitHub 存储库的生成触发发布*主*分支。
14. 单击"保存"按钮。单击**确定**在随后出现的按钮**保存**模式对话框。
15. 单击环**境 1**框。环境面板右侧显示。更改*环境 1*中的文本环**境名称**的 textbox*生产*。

16. 单击**1 阶段，2 个任**务中的链接生产框：

任务环境的选项卡将出现。

17. 单击**将 Azure** 应**用服**务**部署槽**任务。在右侧面板中显示其设置。
18. 选择与中的应用服务相关联的 Azure 订阅**Azure 订**阅下拉列表。选择后，单击**Authorize**按钮。
19. 选择*Web 应用*从应**用类型**下拉列表。
20. 选择*mywebapp / < unique_number / >* 从应**用服**务**名称**下拉列表。

21. 选择*AzureTutorial*从资源组下拉列表。

22. 选择*暂存*从槽下拉列表。

23. 单击"保存"按钮。

24. 将鼠标悬停默认发布管道名称。单击铅笔图标以对其进行编辑。使用*MyFirstProject ASP.NET Core CD*作为名称。

25. 单击"保存"按钮。

将更改提交到 GitHub 并自动部署到 Azure

1. 打开*SimpleFeedReader.sln* Visual Studio 中。

2. 在解决方案资源管理器中打开*Pages\Index.cshtml*。更改 `<h2>Simple Feed Reader - V3</h2>` 到 `<h2>Simple Feed Reader - V4</h2>`。

3. 按**Ctrl**+**Shift**+**B**生成的应用。

4. 将文件提交到 GitHub 存储库。可以使用两种更改Visual Studio 中的页*团队资源管理器*选项卡上，或执行以下命令使用本地计算机的命令行界面：

```
git commit -a -m "upgraded to V4"
```

5. 将更改推入*主*到分支*原点*远程的 GitHub 存储库：

```
git push origin master
```

GitHub 存储库中将显示提交*主*分支：

触发生成，因为在生成定义中启用持续集成触发器选项卡：

6. 导航到排队选项卡**Azure 管道** > **生成**Azure DevOps 服务中的页。已排队的生成显示分支和提交触发生成：

7. 生成成功后，会发生到 Azure 的部署。导航到在浏览器中的应用。请注意，显示在标题中的"V4"文本：

检查 Azure 管道管道

生成定义

生成定义已创建了同名*MyFirstProject ASP.NET Core CI*。完成后，生成过程将产生 *.zip*文件包括要发布的资产。发布管道将这些资产部署到 Azure。

生成定义**任务**选项卡列出了正在使用的各个步骤。有五个生成任务。

1. **还原**—执行 `dotnet restore` 命令还原应用程序的 NuGet 包。默认包使用的源是 nuget.org。

2. **构建**—执行 `dotnet build --configuration release` 命令以编译应用程序的代码。这 `--configuration` 选项用于生成代码，这是适用于部署到生产环境中的优化的版本。修改*BuildConfiguration*在生成定义的变量变量选项卡上，如果需要调试配置，例如。

3. **测试**—执行 `dotnet test --configuration release --logger trx --results-directory <local_path_on_build_agent>` 命令以运行应用的单元测试。在 C# 项目匹配的任何执行单元测试的 `**/*Tests/*.csproj` glob 模式。在保存测试结果 *.trx*文件在指定的位置 `--results-directory` 选项。如果任何测试失败，生成将失败，并且未部署。

> **NOTE**
>
> 若要验证的单元测试工作，请修改*SimpleFeedReader.Tests\Services\NewsServiceTests.cs*特意中断的测试之一。例如，更改 `Assert.True(result.Count > 0);` 到 `Assert.False(result.Count > 0);` 中 `Returns_News_Stories_Given_Valid_Uri` 方法。提交并推送到 GitHub 的更改。生成被触发，但失败。生成管道状态将变为失败。再次还原更改、提交并推送。生成成功。

4. **将发布**—执行 `dotnet publish --configuration release --output <local_path_on_build_agent>` 命令，生成 *.zip*要部署的项目文件。 `--output` 选项指定的发布位置 *.zip*文件。表示通过指定位置预定义的变量名为 `$(build.artifactstagingdirectory)` 。该变量将扩展到本地路径，如*c:\agent_work\1\a*，生成代理上。

5. **发布项目** — Publishes *.zip*生成文件发布任务。该任务接受 *.zip*文件作为参数，这是预定义的变量的位置 `$(build.artifactstagingdirectory)` 。*.Zip*文件发布为一个名为文件夹*drop*。

单击生成定义**摘要**链接以查看与定义的生成历史记录：

在结果页上，单击与唯一的内部版本号对应的链接：

显示此特定生成的摘要。单击**项目**选项卡，并请注意*drop*列出由生成内容生成的文件夹：

使用**下载**并**浏览**检查已发布的项目的链接。

发布管道

发布管道已创建了同名*MyFirstProject ASP.NET Core CD*:

发布管道的两个主要组件是项目并环境。单击框中的项目部分会显示以下窗格：

源（生成定义）值表示此发布管道链接到生成定义。*.Zip*成功运行的生成定义所生成文件提供给*生产*环境以部署到Azure。单击*1 阶段, 2 个任务*中的链接*生产*环境，若要查看发布管道任务：

发布管道的两个任务组成：*将 Azure 应用服务部署槽并管理 Azure 应用服务-槽交换*。单击第一个任务将显示下面的任务配置：

Azure 订阅、服务类型、web 应用名称、资源组和部署槽部署任务中定义。**包或文件**夹文本框中保留 *.zip*要提取并部署到的文件路径 *暂存槽mywebapp<唯一数目（_n)*> web 应用。

单击该槽交换任务将显示下面的任务配置：

提供订阅、资源组、服务类型、web 应用名称和部署槽详细信息。**与生成交**换选中复选框。因此，将位部署到*暂存*槽交换到生产环境。

其他阅读材料

- 使用 Azure Pipelines 创建你的第一个管道
- 生成和.NET Core 项目
- 部署 web 应用与 Azure 管道

监视和调试

2018/12/12 • • Edit Online

部署应用程序和构建 DevOps 管道，务必要了解如何监视和故障排除应用程序。

在本部分中，将完成以下任务：

- 查找基本的监视和故障排除 Azure 门户中的数据
- 了解 Azure Monitor 如何跨所有 Azure 服务提供更深层次查看的度量值
- 使用 Application Insights 的 web 应用连接有关的应用程序分析
- 启用日志记录并了解在何处下载日志
- Stream 实时日志
- 了解在何处设置警报
- 了解远程调试 Azure 应用服务 web 应用。

基本监视和故障排除

在真实时间中轻松地监视应用服务 web 应用。在 Azure 门户将呈现易于理解图表和图形中的指标。

1. 打开Azure 门户，然后导航到*mywebapp<unique_number>* 应用服务。

2. **概述**选项卡显示有用"眼"的信息，包括关系图显示最近的指标。

- **Http 5xx**:服务器端错误，通常在 ASP.NET Core 代码中的异常的计数。
- **中的数据**:数据入口进入你的 web 应用。
- **输出数据量**:数据流出量从 web 应用程序向客户端。

- 请求:计数的 HTTP 请求。
- 平均响应时间:Web 应用响应 HTTP 请求的平均时间。

此外在此页上找到多个自助服务工具进行故障排除和优化。

- 诊断并解决问题是自助服务的故障排除程序。
- **Application Insights**用于分析性能和应用程序行为，稍后在本部分中讨论。
- 应用服务顾问会提出建议来优化您的应用体验。

高级监视

Azure 监视器是用于监视所有度量值以及跨 Azure 服务中设置警报的集中式的服务。在 Azure Monitor，管理员可以精细地跟踪性能和确定趋势。每个 Azure 服务提供其自己度量值组到 Azure Monitor。

使用 Application Insights 配置文件

Application Insights是用于分析的性能和稳定性的 web 应用和用户如何使用这些 Azure 服务。从 Application Insights 的数据是更广泛的和更深入地与 Azure Monitor。开发人员和管理员使用的信息用于改进应用，可以提供数据。可以将 application Insights 添加到 Azure 应用服务资源无需更改代码。

1. 打开Azure 门户，然后导航到*mywebapp<unique_number>* 应用服务。
2. 从**概述**选项卡上，单击**Application Insights**磁贴。

3. 选择创**建新的资源**单选按钮。使用默认资源名称，并选择 Application Insights 资源的位置。位置不需要你的 web 应用的相匹配。

4. 有关**运行时/框架**，选择**ASP.NET Core**。接受默认设置。

5. 选择“**确定**”。如果系统提示你确认，请选择继续。

6. 创建资源后，单击要直接导航到 Application Insights 页的 Application Insights 资源的名称。

使用应用程序时，数据累积。选择**刷新**重新加载新数据的边栏选项卡。

Application Insights 提供有用的服务器端的信息，而无需其他配置。若要从 Application Insights，获取最大的价值使用 Application Insights SDK 检测应用。如果配置正确，该服务提供端对端监控跨 web 服务器和浏览器中，包括客户端的性能。有关详细信息，请参阅Application Insights 文档。

日志记录

在 Azure 应用服务中的默认情况下禁用 web 服务器和应用程序日志。启用日志通过以下步骤：

1. 打开Azure 门户，并导航到*mywebapp<unique_number>* 应用服务。

2. 在左侧菜单中，向下滚动到监视部分。选择诊**断日志**。

3. 开启应**用程序日志**记录（**文件系**统）。如果系统提示，请单击此框来安装扩展，使应用程序中 web 应用的日志记录。

4. 设置**Web 服务器日志**记录到**文件系**统。

5. 输入**保留期**以天为单位。例如，30。

6. 单击"保存" 。

为 web 应用生成 ASP.NET Core 和 web 服务器（应用服务）日志。可以使用显示的 FTP/FTPS 信息下载它们。密码是之前在本指南中创建的部署凭据相同。日志可能很流式传输到使用 PowerShell 或 Azure CLI 在本地计算机直接。也可以是日志Application Insights 中查看。

日志流式处理

可以通过门户实时传输应用程序和 web 服务器日志。

1. 打开Azure 门户，并导航到*mywebapp<unique_number>* 应用服务。

2. 在左侧菜单中，向下滚动到监视部分，并选择**日志流**。

也可以是日志流式处理通过 Azure CLI 或 Azure PowerShell，其中包括通过 Cloud Shell。

警报

Azure 监视器还提供实时警报基于度量值、管理事件和其他条件。

> *注意：当前在 web 应用度量值均可收到警报仅在警报（经典）服务中。*

警报（经典）服务在 Azure Monitor 或下可以找到监视部分中的应用服务设置。

实时调试

Azure 应用服务可以是使用 Visual Studio 远程调试日志时不提供足够的信息。但是，远程调试要求应用程序以使用调试符号编译。调试不应执行在生产中，除了作为最后的手段。

结束语

在本部分中，您将完成以下任务：

- 查找基本的监视和故障排除 Azure 门户中的数据
- 了解 Azure Monitor 如何跨所有 Azure 服务提供更深层次查看的度量值
- 使用 Application Insights 的 web 应用连接有关的应用程序分析
- 启用日志记录并了解在何处下载日志
- Stream 实时日志
- 了解在何处设置警报
- 了解远程调试 Azure 应用服务 web 应用。

其他阅读材料

- 对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除
- Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考
- 监视使用 Application Insights 的 Azure web 应用性能
- 在 Azure 应用服务中为应用启用诊断日志记录
- 使用 Visual Studio 对 Azure 应用服务中的 Web 应用进行故障排除
- Azure Monitor 中创建经典指标警报的 Azure 服务-Azure 门户

后续步骤

2018/12/12 • • Edit Online

在本指南中，您将创建 ASP.NET Core 示例应用的 DevOps 管道。祝贺你！ 我们希望您喜欢学习来将 ASP.NET Core web 应用发布到 Azure 应用服务，并自动执行更改的持续集成。

Web 宿主和 DevOps，超出 Azure 有广泛的平台-作为-服务 (PaaS) 服务适用于 ASP.NET Core 开发人员。本部分提供的一些最常使用的服务的简要概述。

存储和数据库

Redis 缓存是缓存可作为服务的高吞吐量、低延迟数据。它可以用于缓存页面输出、减少数据库请求和 ASP.NET Core 会话状态提供应用的多个实例。

Azure 存储是 Azure 的大规模可缩放的云存储。开发人员可以利用队列存储可靠的消息队列，并表存储是设计用于快速开发使用大规模、半结构化数据集的 NoSQL 键-值存储。

Azure SQL 数据库提供的熟悉的关系数据库功能与使用 Microsoft SQL Server 引擎的服务。

Cosmos DB全球分布的多模型 NoSQL 数据库服务。多个 Api 都可用，包括 SQL API（以前称为 DocumentDB）、Cassandra 和 MongoDB。

标识

Azure Active Directory并Azure Active Directory B2C是这两个标识服务。Azure Active Directory 专为企业方案和时在 Azure Active Directory B2C 是预期的业务客户方案，包括社交网络单一登录启用 Azure AD B2B（企业到企业）协作。

移动电话

通知中心是一种多平台且可扩展的推送通知引擎快速发送数百万条消息到各种类型的设备上运行的应用。

Web 基础结构

Azure 容器服务管理托管的 Kubernetes 环境，使它快速、轻松地部署和管理容器化的应用而无需容器业务流程专业知识。

Azure 搜索用于基于专用异类内容创建一种企业搜索解决方案。

Service Fabric是一个分布式的系统平台，轻松地打包、部署和管理可缩放和可靠的微服务和容器。

使用 IIS 在 Windows 上托管 ASP.NET Core

2019/1/14 • • Edit Online

作者：Luke Latham

安装 .NET Core 托管捆绑包

支持的操作系统

支持下列操作系统：

- Windows 7 或更高版本
- Windows Server 2008 R2 或更高版本

HTTP.sys 服务器（以前称为 WebListener）无法以反向代理配置形式与 IIS 结合使用。请使用 Kestrel 服务器。

有关在 Azure 上托管的信息，请参阅将 ASP.NET Core 应用部署到 Azure 应用服务。

受支持的平台

支持针对 32 位 (x86) 和 64 位 (x64) 部署发布应用。除非应用符合以下条件，否则部署 32 位应用：

- 需要适用于 64 位应用的更大虚拟内存地址空间。
- 需要更大 IIS 堆栈大小。
- 具有 64 位本机依赖项。

应用程序配置

启用 IISIntegration 组件

典型的 Program.cs 调用 CreateDefaultBuilder 以开始设置主机：

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        ...
```

典型的 Program.cs 调用 CreateDefaultBuilder 以开始设置主机：

```
public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        ...
```

进程内承载模型

`CreateDefaultBuilder` 调用 `UseIIS` 方法以启动 CoreCLR 并在 IIS 工作进程（w3wp.exe 或 iisexpress.exe）内托管应用。性能测试表明，与在进程外托管应用并将请求代理到 Kestrel 服务器相比，在进程中托管 .NET Core 应用可以大大提升请求吞吐量。

定目标到 .NET Framework 的 ASP.NET Core 应用不支持进程内托管模型。

进程外承载模型

对于使用 IIS 的进程外托管，`CreateDefaultBuilder` 将 Kestrel 服务器配置为 Web 服务器，并通过配置 ASP.NET Core 模块的基础路径和端口来启用 IIS 集成。

ASP.NET Core 模块生成分配给后端进程的动态端口。`CreateDefaultBuilder` 调用 UseIISIntegration 方法。`UseIISIntegration` 配置在 localhost IP 地址（`127.0.0.1`）中的动态端口上侦听的 Kestrel。如果动态端口为 1234，则 Kestrel 在 `127.0.0.1:1234` 中侦听。此配置将替换以下 API 提供的其他 URL 配置：

- `UseUrls`
- Kestrel 的侦听 API
- 配置（或 命令行 -- URL 选项）

使用模块时，不需要调用 `UseUrls` 或 Kestrel 的 `Listen` API。如果调用 `UseUrls` 或 `Listen`，则 Kestrel 仅会侦听在没有 IIS 的情况下运行应用时指定的端口。

有关进程内和进程外托管模型的详细信息，请参阅 ASP.NET Core 模块和 ASP.NET Core 模块配置参考。

`CreateDefaultBuilder` 将 Kestrel 服务器配置为 Web 服务器，并通过配置 ASP.NET Core 模块的基础路径和端口来启用 IIS 集成。

ASP.NET Core 模块生成分配给后端进程的动态端口。`CreateDefaultBuilder` 调用 UseIISIntegration 方法。`UseIISIntegration` 配置在 localhost IP 地址（`127.0.0.1`）中的动态端口上侦听的 Kestrel。如果动态端口为 1234，则 Kestrel 在 `127.0.0.1:1234` 中侦听。此配置将替换以下 API 提供的其他 URL 配置：

- `UseUrls`
- Kestrel 的侦听 API
- 配置（或 命令行 -- URL 选项）

使用模块时，不需要调用 `UseUrls` 或 Kestrel 的 `Listen` API。如果调用 `UseUrls` 或 `Listen`，则 Kestrel 仅会侦听在没有 IIS 的情况下运行应用时指定的端口。

`CreateDefaultBuilder` 将 Kestrel 服务器配置为 Web 服务器，并通过配置 ASP.NET Core 模块的基础路径和端口来启用 IIS 集成。

ASP.NET Core 模块生成分配给后端进程的动态端口。`CreateDefaultBuilder` 调用 UseIISIntegration 方法。`UseIISIntegration` 配置在 localhost IP 地址（`localhost`）中的动态端口上侦听的 Kestrel。如果动态端口为 1234，则 Kestrel 在 `localhost:1234` 中侦听。此配置将替换以下 API 提供的其他 URL 配置：

- `UseUrls`
- Kestrel 的侦听 API
- 配置（或 命令行 -- URL 选项）

使用模块时，不需要调用 `UseUrls` 或 Kestrel 的 `Listen` API。如果调用 `UseUrls` 或 `Listen`，则 Kestrel 仅会侦听在没有 IIS 的情况下运行应用时指定的端口。

在应用依赖项中加入对 Microsoft.AspNetCore.Server.IISIntegration 包的依赖项。通过向 WebHostBuilder 添加 UseIISIntegration 扩展方法来使用 IIS 集成中间件：

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseIISIntegration()
    ...
```

同时需要 UseKestrel 和 UseIISIntegration。调用 `UseIISIntegration` 的代码不会影响代码可移植性。如果应用不在 IIS 后面运行（例如，应用直接在 Kestrel 上运行），则 `UseIISIntegration` 不会运行。

ASP.NET Core 模块生成分配给后端进程的动态端口。`UseIISIntegration` 配置在 localhost IP 地址（`localhost`）中的动态端口上侦听的 Kestrel。如果动态端口为 1234，则 Kestrel 在 `localhost:1234` 中侦听。此配置将替换以下 API 提供的其他 URL 配置：

- `UseUrls`

- 配置(或 命令行 -- URL 选项)

使用模块时无需调用 `UseUrls`。如果调用 `UseUrls`，则 Kestrel 仅会侦听在没有 IIS 的情况下运行应用时指定的端口。

如果在 ASP.NET Core 1.0 应用中调用 `UseUrls`，请在调用 `UseIISIntegration` 前调用它，使模块配置的端口不会被覆盖。ASP.NET Core 1.1 不需要此调用顺序，因为模块设置会重写 `UseUrls`。

有关托管的详细信息，请参阅在 ASP.NET Core 中托管。

IIS 选项

进程内承载模型

若要配置 IIS 服务器选项，请在 ConfigureServices 中加入适用于 IISServerOptions 的服务配置。下面的示例禁用 AutomaticAuthentication：

```
services.Configure<IISServerOptions>(options =>
{
    options.AutomaticAuthentication = false;
});
```

选项	默认	设置
`AutomaticAuthentication`	`true`	若为 `true`，IIS 服务器将设置经过 Windows 身份验证进行身份验证的 `HttpContext.User`。若为 `false`，服务器仅提供 `HttpContext.User` 的标识并在 `AuthenticationScheme` 显式请求时响应质询。必须在 IIS 中启用 Windows 身份验证使 `AutomaticAuthentication` 得以运行。有关详细信息，请参阅 Windows 身份验证。
`AuthenticationDisplayName`	`null`	设置在登录页上向用户显示的显示名。

进程外承载模型

若要配置 IIS 选项，请在 ConfigureServices 中加入适用于 IISOptions 的服务配置。下面的示例阻止应用填充 `HttpContext.Connection.ClientCertificate`：

```
services.Configure<IISOptions>(options =>
{
    options.ForwardClientCertificate = false;
});
```

选项	默认	设置

选项	默认	设置
`AutomaticAuthentication`	`true`	若为 `true`，IIS 集成中间件将设置经过 Windows 身份验证进行身份验证的 `HttpContext.User`。若为 `false`，中间件仅提供 `HttpContext.User` 的标识并在 `AuthenticationScheme` 显式请求时响应质询。必须在 IIS 中启用 Windows 身份验证使 `AutomaticAuthentication` 得以运行。有关详细信息，请参阅 Windows 身份验证主题。
`AuthenticationDisplayName`	`null`	设置在登录页上向用户显示的显示名。
`ForwardClientCertificate`	`true`	若为 `true`，且存在 `MS-ASPNETCORE-CLIENTCERT` 请求头，则填充 `HttpContext.Connection.ClientCertificate`。

代理服务器和负载均衡器方案

配置转发头中间件的 IIS 集成中间件和 ASP.NET Core 模块将配置为转发方案 (HTTP/HTTPS) 和发出请求的远程 IP 地址。对于托管在其他代理服务器和负载均衡器后方的应用，可能需要附加配置。有关详细信息，请参阅配置 ASP.NET Core 以使用代理服务器和负载均衡器。

web.config 文件

web.config 文件配置 ASP.NET Core 模块。发布项目时，MSBuild 目标（`_TransformWebConfig`）负责创建、转换和发布 web.config 文件。此目标位于 Web SDK 目标（`Microsoft.NET.Sdk.Web`）中。SDK 设置在项目文件的顶部：

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

如果项目中不存在 web.config 文件，则会使用正确的 processPath 和参数创建该文件，以便配置 ASP.NET Core 模块，并将该文件移动到已发布的输出。

如果项目中存在 web.config 文件，则会使用正确的 processPath 和参数转换该文件，以便配置 ASP.NET Core 模块，并将该文件移动到已发布的输出。转换不会修改文件中的 IIS 配置设置。

web.config 文件可能会提供其他 IIS 配置设置，以控制活动的 IIS 模块。有关能够处理 ASP.NET Core 应用请求的 IIS 模块的信息，请参阅 IIS 模块主题。

要防止 Web SDK 转换 web.config 文件，请使用项目文件中的 <IsTransformWebConfigDisabled> 属性：

```
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

在禁用 Web SDK 转换文件时，开发人员应手动设置 processPath 和参数。有关详细信息，请参阅 ASP.NET Core 模块配置参考。

web.config 文件位置

为了正确设置 ASP.NET Core 模块web.config 文件必须存在于已部署应用的内容根路径（通常为应用基路径）中。该位置与向 IIS 提供的网站物理路径相同。若要使用 Web 部署发布多个应用，应用的根路径中需要包含 web.config 文件。

敏感文件存在于应用的物理路径中，如 <assembly>.runtimeconfig.json、<assembly>.xml（XML 文档注释）和 <assembly>.deps.json。当存在 web.config 文件且站点正常启动的情况下，若要请求获取这些敏感文件，IIS 不会提供。如果缺少 web.config 文件、命名不正确，或无法配置站点以正常启动，IIS 可能会公开提供敏感文件。

部署中必须始终存在 web.config 文件且正确命名，并可以配置站点以正常启动。请**勿从生产部署中删除 web.config 文件**。

IIS 配置

Windows Server 操作系统

启用 Web 服务器 (IIS) 服务器角色并建立角色服务。

1. 通过"管理"菜单或"服务器管理器"中的链接使用"添加角色和功能"向导。在"服务器角色"步骤中，选中"Web 服务器(IIS)"框。

2. 在"功能"步骤后，为 Web 服务器 (IIS) 加载"角色服务"步骤。选择所需 IIS 角色服务，或接受提供的默认角色服务。

Windows 身份验证(可选)
若要启用 Windows 身份验证，请依次展开以下节点:“Web 服务器” > “安全”。选择“Windows 身份验证”功能。有关详细信息，请参阅 Windows 身份验证 <windowsAuthentication> 和配置 Windows 身份验证。

Websocket(可选)
Websocket 支持 ASP.NET Core 1.1 或更高版本。若要启用 WebSocket，请依次展开以下节点:“Web 服务器” > “应用开发”。选择“WebSocket 协议”功能。有关详细信息，请参阅 WebSockets。

3. 继续执行“确认”步骤，安装 Web 服务器角色和服务。安装 Web 服务器 (IIS) 角色后无需重启服务器/IIS。

Windows 桌面操作系统

启用“IIS 管理控制台”和“万维网服务”。

1. 导航到“控制面板” > “程序” > “程序和功能” > “打开或关闭 Windows 功能”(位于屏幕左侧)。
2. 打开“Internet Information Services”节点。打开“Web 管理工具”节点。
3. 选中“IIS 管理控制台”框。
4. 选中“万维网服务”框。
5. 接受“万维网服务”的默认功能，或自定义 IIS 功能。

   Windows 身份验证(可选)
   若要启用 Windows 身份验证，请依次展开以下节点:“万维网服务” > “安全”。选择“Windows 身份验证”功能。有关详细信息，请参阅 Windows 身份验证 <windowsAuthentication> 和配置 Windows 身份验证。

   Websocket(可选)
   Websocket 支持 ASP.NET Core 1.1 或更高版本。若要启用 WebSocket，请依次展开以下节点:“万维网服务” > “应用开发功能”。选择“WebSocket 协议”功能。有关详细信息，请参阅 WebSockets。

6. 如果 IIS 安装需要重新启动，则重新启动系统。

安装 .NET Core 托管捆绑包

在托管系统上安装 .NET Core 托管捆绑包。捆绑包可安装 .NET Core 运行时、.NET Core 库和 ASP.NET Core 模块。该模块允许 ASP.NET Core 应用在 IIS 后面运行。如果系统没有 Internet 连接，请先获取并安装 Microsoft Visual C++ 2015 Redistributable，然后再安装 .NET Core 托管捆绑包。

> **IMPORTANT**
>
> 如果在 IIS 之前安装了托管捆绑包，则必须修复捆绑包安装。在安装 IIS 后再次运行托管捆绑包安装程序。

直接下载（当前版本）

使用以下链接下载安装程序：

当前 .NET Core 托管捆绑包安装程序（直接下载）

先前版本的安装程序

若要获取先前版本的安装程序：

1. 导航到 .NET 下载存档。
2. 在“.NET Core”下，选择 .NET Core 版本。
3. 在“运行应用 - 运行时”列中，查找所需的 .NET Core 运行时版本的那一行。
4. 使用“运行时和托管捆绑包”链接下载安装程序。

> **WARNING**
>
> 某些安装程序包含已到达其生命周期结束 (EOL) 且不再受 Microsoft 支持的发行版本。有关详细信息，请参阅支持策略。

安装托管捆绑包

1. 在服务器上运行安装程序。从管理员命令提示符运行安装程序时，以下开关将可用：

   - `OPT_NO_ANCM=1` – 跳过安装 ASP.NET Core 模块。
   - `OPT_NO_RUNTIME=1` – 跳过安装 .NET Core 运行时。
   - `OPT_NO_SHAREDFX=1` – 跳过安装 ASP.NET 共享框架（ASP.NET 运行时）。

- `OPT_NO_X86=1` – 跳过安装 x86 运行时。确定不会托管 32 位应用时，请使用此开关。如果有同时托管 32 位和 64 位应用的可能，请不要使用此开关并安装两个运行时。

2. 重启系统，或从命令提示符处依次执行 net stop was /y 和 net start w3svc。重启 IIS 会选取安装程序对系统 PATH（环境变量）所作的更改。

如果 Windows 托管捆绑包安装程序检测到 IIS 需要重置才能完成安装，则安装程序会重置 IIS。如果安装程序触发 IIS 重置，则会重新启动所有 IIS 应用池和网站。

> **NOTE**
>
> 有关 IIS 共享配置的信息，请参阅使用 IIS 共享配置的 ASP.NET Core 模块。

使用 Visual Studio 进行发布时安装 Web 部署

使用 Web 部署将应用部署到服务器时，请在服务器上安装最新版本的 Web 部署。要安装 Web 部署，请使用 Web 平台安装程序 (WebPI) 或直接从 Microsoft 下载中心获取安装程序。建议使用 WebPI。WebPI 为托管提供程序提供独立的安装程序和配置。

创建 IIS 站点

1. 在托管系统上，创建一个文件夹以包含应用已发布的文件夹和文件。目录结构主题中介绍了应用的部署布局。

2. 在新文件夹中创建一个“日志”文件夹，用于在启用 stdout 日志记录时保存 ASP.NET Core 模块 stdout 日志。如果部署应用时有效负载中包含了“日志”文件夹，请跳过此步骤。有关如何启用 MSBuild 以在本地生成项目时自动创建“日志”文件夹的说明，请参阅目录结构主题。

   > **IMPORTANT**
   >
   > 仅使用 stdout 日志来解决应用启动失败的问题。请勿使用 stdout 日志记录进行常规应用日志记录。日志文件大小或创建的日志文件数没有限制。应用池必须对写入日志的位置具有写入权限。日志位置路径上的所有文件夹都必须存在。有关 stdout 日志的详细信息，请参阅日志创建和重定向。有关 ASP.NET Core 应用中的日志记录信息，请参阅日志记录主题。

3. 在“IIS 管理器”中，打开“连接”面板中的服务器节点。右键单击“站点”文件夹。选择上下文菜单中的“添加网站”。

4. 提供网站名称，并将物理路径设置为应用的部署文件夹。提供“绑定”配置，并通过选择“确定”创建网站：

> **WARNING**
>
> 不应使用顶级通配符绑定（`http://*:80/` 和 `http://+:80`）。顶级通配符绑定可能会为应用带来安全漏洞。此行为同时适用于强通配符和弱通配符。使用显式主机名而不是通配符。如果可控制整个父域（区别于易受攻击的 `*.com`），则子域通配符绑定（例如，`*.mysub.com`）不具有此安全风险。有关详细信息，请参阅 rfc7230 第 5.4 条。

5. 在服务器节点下，选择“应用程序池”。

6. 右键单击站点的应用池，然后从上下文菜单中选择“基本设置”。

7. 在“编辑应用程序池”窗口中，将“.NET CLR 版本”设置为“无托管代码”：

ASP.NET Core 在单独的进程中运行，并管理运行时。ASP.NET Core 不依赖加载桌面 CLR。将“.NET CLR 版本”设置为“无托管代码”为可选步骤。

8. *ASP.NET Core 2.2 或更高版本*：对于使用进程内托管模型的 64 位 (x64) 独立部署，为 32 位 (x86) 进程禁

用应用池。

在 IIS 管理员的“应用程序池”的“操作”侧栏中，选择“设置应用程序池默认设置”或“高级设置”。找到“启用 32 位应用程序”并将值设置为 `False`。此设置不会影响针对进程外托管部署的应用。

9. 确认进程模型标识拥有适当的权限。

   如果将应用池的默认标识（“进程模型” > “标识”）从 ApplicationPoolIdentity 更改为另一标识，请验证新标识拥有所需的权限，可访问应用的文件夹、数据库和其他所需资源。例如，应用池需要对文件夹的读取和写入权限，以便应用在其中读取和写入文件。

Windows 身份验证配置（可选）

有关详细信息，请参阅配置 Windows 身份验证。

部署应用

将应用部署到在托管系统上创建的文件夹。建议使用的部署机制是 Web 部署。

在 Visual Studio 内使用 Web 部署

要了解如何创建用于 Web 部署的发布配置文件，请参阅用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件。如果托管提供程序提供了发布配置文件或支持创建发布配置文件，请下载配置文件并使用 Visual Studio 的“发布”对话框将其导入。

在 Visual Studio 之外使用 Web 部署

也可以在 Visual Studio 之外从命令行使用 Web 部署。有关详细信息，请参阅 Web Deployment Tool（Web 部署工具）。

Web 部署的替代方法

有多种方法可将应用移动到托管系统，例如手动复制、Xcopy、Robocopy 或 PowerShell，可使用其中任何一种方法。

有关将 ASP.NET Core 部署到 IIS 的详细信息，请参阅面向 IIS 管理员的部署资源部分。

浏览网站

锁定的部署文件

如果应用正在运行，部署文件夹中的文件会被锁定。在部署期间，无法覆盖已锁定的文件。若要在部署中解除已锁定的文件，请使用以下方法之一停止应用池：

- 使用 Web 部署并在项目文件中引用 `Microsoft.NET.Sdk.Web`。系统会在 Web 应用目录的根目录中放置一个 app_offline.htm 文件。该文件存在时，ASP.NET Core 模块会在部署过程中正常关闭该应用并提供 app_offline.htm 文件。有关详细信息，请参阅 ASP.NET Core 模块配置参考。
- 在服务器上的 IIS 管理器中手动停止应用池。
- 使用 PowerShell 删除 app_offline.html（需要使用 PowerShell 5 或更高版本）：

```
$pathToApp = 'PATH_TO_APP'

# Stop the AppPool
New-Item -Path $pathToApp app_offline.htm

# Provide script commands here to deploy the app

# Restart the AppPool
Remove-Item -Path $pathToApp app_offline.htm
```

数据保护

ASP.NET Core 数据保护堆栈由多个 ASP.NET Core 中间件使用，包括用于身份验证的中间件。即使用户代码不调用数据保护 API，也应该使用部署脚本或在用户代码中配置数据保护，以创建持久的加密密钥存储。如果不配置数据保护，则密钥存储在内存中。重启应用时，密钥会被丢弃。

如果密钥环存储于内存中，则在应用重启时：

- 所有基于 cookie 的身份验证令牌都无效。
- 用户需要在下一次请求时再次登录。
- 无法再解密使用密钥环保护的任何数据。这可能包括 CSRF 令牌和 ASP.NET Core MVC TempData cookie。

若要在 IIS 下配置数据保护以持久化密钥环，请使用以下方法之一：

- **创建数据保护注册表项**

  ASP.NET Core 应用使用的数据保护密钥存储在应用外部的注册表中。要持久保存给定应用的密钥，需为应用池创建注册表项。

  对于独立的非 Web 场 IIS 安装，可以对用于 ASP.NET Core 应用的每个应用池使用数据保护 Provision-AutoGenKeys.ps1 PowerShell 脚本。此脚本在 HKLM 注册表中创建注册表项，仅应用程序的应用池工作进程帐户可对其进行访问。通过计算机范围的密钥使用 DPAPI 对密钥静态加密。

  在 web 场方案中，可以将应用配置为使用 UNC 路径存储其数据保护密钥环。默认情况下，数据保护密钥未加密。确保网络共享的文件权限仅限于应用在其下运行的 Windows 帐户。可使用 X509 证书来保护静态密钥。考虑允许用户上传证书的机制：将证书置于用户信任的证书存储中，并确保这些证书对所有运行用户应用的计算机都可用。有关详细信息，请参阅配置 ASP.NET Core 数据保护。

- **配置 IIS 应用程序池以加载用户配置文件**

  此设置位于应用池“高级设置”下的“进程模型”部分。将“加载用户配置文件”设置为 `True`。这会将密钥存储在用户配置文件目录下，并使用 DPAPI 和特定于用户帐户（用于应用池）的密钥进行保护。

- **将文件系统用作密钥环存储**

  调整应用代码，将文件系统用作密钥环存储。使用 X509 证书保护密钥环，并确保该证书是受信任的证书。如果它是自签名证书，则将该证书放置于受信任的根存储中。

  当在 Web 场中使用 IIS 时：

  - 使用所有计算机都可以访问的文件共享。
  - 将 X509 证书部署到每台计算机。通过代码配置数据保护。
- **设置用于数据保护的计算机范围的策略**

  数据保护系统对以下操作提供有限支持：为使用数据保护 API 的所有应用设置默认计算机范围的策略。有关更多信息，请参见ASP.NET Core 数据保护。

虚拟目录

ASP.NET Core 应用不支持 IIS 虚拟目录。可将应用托管为子应用程序。

子应用程序

可将 ASP.NET Core 应用托管为 IIS 子应用程序（子应用）。子应用的路径成为根应用 URL 的一部分。

子应用不应将 ASP.NET Core 模块作为处理程序包含在其中。如果在子应用的 web.config 文件中将该模块添加为处理程序，则在尝试浏览子应用时会收到“500.19 内部服务器错误”，即引用错误的配置文件。

以下示例显示 ASP.NET Core 子应用的已发布 web.config 文件：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

在 ASP.NET Core 应用下托管非 ASP.NET Core 子应用时，需在子应用的 web.config 文件中显式删除继承的处理程序：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <remove name="aspNetCore" />
    </handlers>
    <aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

子应用内的静态资产链接应使用波形符-斜杠（`~/`）符号。波形符-斜杠符号触发标记帮助器，来将子应用的基路径追加到呈现的相关链接前面。对于 `/subapp_path` 处的子应用，使用 `src="~/image.png"` 链接的图像将呈现为 `src="/subapp_path/image.png"`。根应用的静态文件中间件不处理静态文件请求。此请求由子应用的静态文件中间件处理。

若将静态资产的 `src` 属性设置为绝对路径（如 `src="/image.png"`），则呈现的链接不包含子应用的基路径。根应用的静态文件中间件试图从根应用的 webroot 提供资产，这会导致“404 - 找不到”响应，除非可从根应用获得此静态资产。

若要将 ASP.NET Core 应用作为子应用托管在其他 ASP.NET Core 应用下：

1. 为此子应用创建应用池。将“.NET CLR 版本”设置为“无托管代码”。
2. 在 IIS 管理器中添加根网站，并且此子应用在根网站的某个文件夹中。
3. 在 IIS 管理器中右击此子应用文件夹，并选择“转换为应用程序”。
4. 在“添加应用程序”对话框中，使用“应用程序池”的“选择”按钮来分配为子应用创建的应用池。选择“确定”。

使用进程内托管模型时，需要向子应用分配单独的应用池。

有关进程内托管模型及 ASP.NET Core 模块配置的详细信息，请参阅 ASP.NET Core 模块 和 ASP.NET Core 模块。

使用 web.config 配置 IIS

IIS 配置受用于 IIS 方案（适用于包含 ASP.NET Core 模块的 ASP.NET Core 应用）的 web.config 的 `<system.webServer>` 部分影响。例如，IIS 配置适用于动态压缩。如果在服务器一级将 IIS 配置为使用动态压缩，可通过应用的 web.config 文件中的 `<urlCompression>` 元素，对 ASP.NET Core 应用禁用它。

有关详细信息，请参阅 <system.webServer> 的配置参考、ASP.NET Core 模块配置参考和使用 ASP.NET Core 的 IIS 模块。若要为在独立应用池中运行的各应用设置环境变量（IIS 10.0 或更高版本中支持此操作），请参阅 IIS 参考文档的环境变量 <environmentVariables> 主题中的“AppCmd.exe 命令”部分。

web.config 的配置节

ASP.NET Core 应用不会使用 web.config 中的 ASP.NET 4.x 应用的配置部分进行配置：

- `<system.web>`
- `<appSettings>`
- `<connectionStrings>`
- `<location>`

会使用其他的配置提供程序配置 ASP.NET Core 应用。有关详细信息，请参阅配置。

应用程序池

由托管模型决定应用池隔离：

- 进程内托管 – 应用都需要在单独的应用池中运行。
- 进程外托管 – 建议在每个应用自己的应用池中运行各应用，以彼此隔离。

IIS“添加网站”对话框默认为每应用一个应用池。提供了站点名称时，该文本会自动传输到“应用程序池”文本框。添加站点时，会使用该站点名称创建新的应用池。

在服务器上托管多个网站时，建议在每个应用自己的应用池中运行各应用，以彼此隔离。IIS“添加网站”对话框默认执行此配置。提供了站点名称时，该文本会自动传输到“应用程序池”文本框。添加站点时，会使用该站点名称创建新的应用池。

应用程序池标识

通过应用池标识帐户，可以在唯一帐户下运行应用，而无需创建和管理域或本地帐户。在 IIS 8.0 或更高版本上，IIS 管理员工作进程 (WAS) 将使用新应用池的名称创建一个虚拟帐户，并默认在此帐户下运行应用池的工作进程。在 IIS 管理控制台中，确保应用池“高级设置”下的“标识”设置为使用 ApplicationPoolIdentity：

IIS 管理进程使用 Windows 安全系统中应用池的名称创建安全标识符。可使用此标识保护资源。但是，此标识

不是真实的用户帐户，不会在 Windows 用户管理控制台中显示。

如果 IIS 工作进程需要对应用的高级访问权限，请为包含该应用的目录修改访问控制列表 (ACL)：

1. 打开 Windows 资源管理器并导航到目录。
2. 右键单击该目录，然后选择“属性”。
3. 在“安全”选项卡下，选择“编辑”按钮，然后单击“添加”按钮。
4. 选择“位置”按钮，并确保该系统处于选中状态。
5. 在“输入要选择的对象名称”区域中输入“IIS AppPool\<app_pool_name>”。选择“检查名称”按钮。有关 DefaultAppPool，请检查使用 IIS AppPool\DefaultAppPool 的名称。当选择“检查名称”按钮时，对象名称区域中会显示 DefaultAppPool 的值。无法直接在对象名称区域中输入应用池名称。检查对象名称时，请使用 IIS AppPool\<app_pool_name> 格式。

6. 选择“确定”。

7. 默认情况下应授予读取 & 执行权限。根据需要请提供其他权限。

也可使用 ICACLS 工具在命令提示符处授予访问权限。以 DefaultAppPool 为例，使用以下命令：

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool":F
```

有关详细信息，请参阅 icacls 主题。

HTTP/2 支持

以下 IIS 部署方案中的 ASP.NET Core 支持 HTTP/2：

- 进程内
  - Windows Server 2016/Windows 10 或更高版本；IIS 10 或更高版本
  - TLS 1.2 或更高版本的连接
- 进程外

- Windows Server 2016/Windows 10 或更高版本；IIS 10 或更高版本
- 面向公众的边缘服务器连接使用 HTTP/2，但与 Kestrel 服务器的反向代理连接使用 HTTP/1.1。
- TLS 1.2 或更高版本的连接

对于已建立 HTTP/2 连接时的进程内部署，HttpRequest.Protocol 会报告 `HTTP/2`。对于已建立 HTTP/2 连接时的进程外部署，HttpRequest.Protocol 会报告 `HTTP/1.1`。

有关进程内和进程外托管模型的详细信息，请参阅 ASP.NET Core 模块 主题和 ASP.NET Core 模块。

满足以下基本要求的进程外部署支持 HTTP/2：

- Windows Server 2016/Windows 10 或更高版本；IIS 10 或更高版本
- 面向公众的边缘服务器连接使用 HTTP/2，但与 Kestrel 服务器的反向代理连接使用 HTTP/1.1。
- 目标框架：不适用于进程外部署，因为 HTTP/2 连接完全由 IIS 处理。
- TLS 1.2 或更高版本的连接

如果已建立 HTTP/2 连接，HttpRequest.Protocol 会报告 `HTTP/1.1`。

默认情况下将启用 HTTP/2。如果未建立 HTTP/2 连接，连接会回退到 HTTP/1.1。有关使用 IIS 部署的 HTTP/2 配置的详细信息，请参阅 IIS 上的 HTTP/2。

面向 IIS 管理员的部署资源

在 IIS 文档中深入了解 IIS。
IIS 文档

了解 .NET Core 应用部署模型。
.NET Core 应用程序部署

了解 ASP.NET Core 模块如何使 Kestrel Web 服务器将 IIS 或 IIS Express 用作反向代理服务器。
ASP.NET Core 模块

了解如何配置 ASP.NET Core 模块以托管 ASP.NET Core 应用。
ASP.NET Core 模块配置参考

了解已发布的 ASP.NET Core 应用的目录结构。
目录结构

了解适用于 ASP.NET Core 应用的活动和非活动 IIS 模块以及如何管理 IIS 模块。
IIS 模块

了解如何诊断 ASP.NET Core 应用的 IIS 部署的问题。
疑难解答

了解在 IIS 上托管 ASP.NET Core 应用的常见错误。
Azure 应用服务和 IIS 的常见错误参考

其他资源

- 解决 ASP.NET Core 项目
- ASP.NET Core 简介
- Microsoft IIS 官方网站
- Windows Server 技术内容库
- IIS 上的 HTTP/2

对 IIS 上的 ASP.NET Core 进行故障排除

2019/1/10 • • Edit Online

作者：Luke Latham

本文说明了如何在使用 Internet Information Services (IIS) 托管时诊断 ASP.NET Core 应用启动问题。本文提供的信息适用于在 Windows Server 和 Windows 桌面上的 IIS 中托管。

在 Visual Studio 中，ASP.NET Core 项目默认为在调试期间进行 IIS Express 托管。本地调试时出现的“502.5 - 进程失败”或“500.30 - 启动失败”可以使用本主题中的建议进行故障排除。

在 Visual Studio 中，ASP.NET Core 项目默认为在调试期间进行 IIS Express 托管。可以使用本主题中的建议对本地调试进行故障排除时，出现“502.5 进程失败”。

其他故障排除主题：

对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除
虽然应用服务使用 ASP.NET Core 模块和 IIS 托管应用，但若要获取特定于应用服务的说明，请参阅专用主题。

处理 ASP.NET Core 中的错误
了解如何在本地系统上处理 ASP.NET Core 应用在开发期间的错误。

了解如何使用 Visual Studio 进行调试
本主题介绍了 Visual Studio 调试器的功能。

使用 Visual Studio Code 进行调试
了解 Visual Studio Code 中内置的调试支持。

应用启动错误

502.5 进程失败

工作进程失败。应用不启动。

ASP.NET Core 模块尝试启动后端 dotnet 进程，但启动失败。通常可以从“应用程序事件日志”和“ASP.NET Core 模块 stdout 日志”的条目中确定进程启动失败的原因。

常见的失败情况是，由于目标 ASP.NET Core 共享框架版本不存在，因此应用配置错误。检查目标计算机上安装的 ASP.NET Core 共享框架版本。

托管或应用配置错误导致工作进程失败时，将返回“502.5 进程失败”错误页面：

500.30 进程内启动失败

工作进程失败。应用不启动。

ASP.NET Core 模块尝试进程内启动 .NET Core CLR，但启动失败。通常可以从"应用程序事件日志"和"ASP.NET Core 模块 stdout 日志"的条目中确定进程启动失败的原因。

常见的失败情况是，由于目标 ASP.NET Core 共享框架版本不存在，因此应用配置错误。检查目标计算机上安装的 ASP.NET Core 共享框架版本。

500.0 进程内处理程序加载失败

工作进程失败。应用不启动。

ASP.NET Core 模块无法找到 .NET Core CLR 和进程内请求处理程序 (aspnetcorev2_inprocess.dll)。检查：

- 该应用针对 Microsoft.AspNetCore.Server.IIS NuGet 包或 Microsoft.AspNetCore.App 元包。
- 目标计算机上安装了该应用所针对的 ASP.NET Core 共享框架版本。

500.0 进程外处理程序加载失败

工作进程失败。应用不启动。

ASP.NET Core 模块无法找到进程外托管请求处理程序。请确保 aspnetcorev2.dll 旁边的子文件夹中存在 aspnetcorev2_outofprocess.dll。

500 内部服务器错误

应用启动，但某个错误阻止了服务器完成请求。

在启动期间或在创建响应时，应用的代码内出现此错误。响应可能不包含任何内容，或响应可能会在浏览器中显示为"500 内部服务器错误"。应用程序事件日志通常表明应用正常启动。从服务器的角度来看，这是正确的。应用已启动，但无法生成有效的响应。在服务器上在命令提示符处运行应用或启用 ASP.NET Core 模块 stdout 日志以解决该问题。

未能启动应用程序（错误代码"0x800700c1"）

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

应用未能启动，因为应用的程序集 (.dll) 无法加载。

当已发布的应用与 w3wp/iisexpress 进程之间的位数不匹配时，会出现此错误。

确认应用池的 32 位设置正确：

1. 在 IIS 管理器的“应用程序池”中选择应用池。
2. 在“操作”面板中的“编辑应用程序池”下选择“高级设置”。
3. 设置“启用 32 位应用程序”：
   - 如果部署 32 位 (x86) 应用，则将值设置为 `True`。
   - 如果部署 64 位 (x64) 应用，则将值设置为 `False`。

连接重置

如果在发送标头后出现错误，则服务器在出现错误时发送“500 内部服务器错误”已经太晚了。通常在序列化响应的复杂对象期间出现错误时发生这种情况。此类型的错误在客户端上显示为“连接重置”错误。应用程序日志记录可以帮助解决这些类型的错误。

默认启动限制

ASP.NET Core 模块的默认“startupTimeLimit”配置为 120 秒。保留默认值时，在模块记录进程故障之前，可能最多需要两分钟来启动应用。有关配置模块的信息，请参阅 aspNetCore 元素的属性。

解决应用启动错误

启用 ASP.NET Core 模块调试日志

将以下处理程序设置添加到应用的 web.config 文件以启用 ASP.NET Core 模块调试日志：

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

确认为日志指定的路径存在，并且应用池的标识具有该位置的写入权限。

应用程序事件日志

访问应用程序事件日志：

1. 打开“开始”菜单，搜索“事件查看器”，然后选择“事件查看器”应用。
2. 在“事件查看器”中，打开“Windows 日志”节点。
3. 选择“应用程序”以打开应用程序事件日志。
4. 搜索与失败应用相关联的错误。错误具有“源”列中“IIS AspNetCore 模块”或“IIS Express AspNetCore 模块”的值。

在命令提示符处运行应用

许多启动错误未在应用程序事件日志中生成有用信息。可以通过在托管系统上在命令提示符处运行应用来找到某些错误的原因。

依赖框架的部署

如果应用是依赖框架的部署：

1. 在命令提示符处，导航到部署文件夹并通过使用 dotnet.exe 执行应用的程序集来运行应用。在以下命令中，替换 <assembly_name> 的应用程序集的名称：`dotnet .\<assembly_name>.dll`。
2. 来自应用且显示任何错误的控制台输出将写入控制台窗口。
3. 如果向应用发出请求时出现错误，请向 Kestrel 侦听所在的主机和端口发出请求。如果使用默认主机和端口，请向 `http://localhost:5000/` 发出请求。如果应用在 Kestrel 终结点地址处正常响应，则问题更可能与承载配置相关，而不太可能在于应用。

独立部署

如果应用是独立部署：

1. 在命令提示符处，导航到部署文件夹并运行应用的可执行文件。在以下命令中，替换 <assembly_name> 的应用程序集的名称：`<assembly_name>.exe`。
2. 来自应用且显示任何错误的控制台输出将写入控制台窗口。
3. 如果向应用发出请求时出现错误，请向 Kestrel 侦听所在的主机和端口发出请求。如果使用默认主机和端口，请向 `http://localhost:5000/` 发出请求。如果应用在 Kestrel 终结点地址处正常响应，则问题更可能与承载配置相关，而不太可能在于应用。

ASP.NET Core 模块 stdout 日志

若要启用和查看 stdout 日志，请执行以下操作：

1. 在托管系统上导航到站点的部署文件夹。
2. 如果 logs 文件夹不存在，请创建该文件夹。有关如何启用 MSBuild 以在部署中自动创建 logs 文件夹的说明，请参阅目录结构主题。
3. 编辑 web.config 文件。将"stdoutLogEnabled"设置为 `true` 并更改"stdoutLogFile"路径以指向 logs 文件夹（例如，`.\logs\stdout`）。路径中的 `stdout` 是日志文件名的前缀。创建日志时，将自动添加时间戳、进程 ID 和文件扩展名。如果将 `stdout` 用作文件名的前缀，典型的日志文件将命名为"stdout_20180205184032_5412.log"。
4. 请确保应用程序池的标识具有对日志文件夹的写入权限。
5. 保存已更新的 web.config 文件。
6. 向应用发出请求。
7. 导航到 logs 文件夹。查找并打开最新的 stdout 日志。
8. 研究日志以查找错误。

> **IMPORTANT**
>
> 故障排除完成后，禁用 stdout 日志记录。

1. 编辑 web.config 文件。
2. 将"stdoutLogEnabled"设置为 `false`。
3. 保存该文件。

> **WARNING**
>
> 无法禁用 stdout 日志可能会导致应用或服务器出现故障。日志文件大小或创建的日志文件数没有限制。
>
> 对于 ASP.NET Core 应用中的例程日志记录，使用限制日志文件大小和旋转日志的日志记录库。有关详细信息，请参阅第三方日志记录提供程序。

启用开发人员异常页面

`ASPNETCORE_ENVIRONMENT` 环境变量可以添加到 web.config 以在开发环境中运行应用。只要在应用启动时环境不被主机生成器上的 `UseEnvironment` 重写，设置环境变量就会在运行应用时显示"开发人员异常页面"。

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

仅建议在未向 Internet 公开的暂存服务器和测试服务器上设置 `ASPNETCORE_ENVIRONMENT` 的环境变量。在故障排除后从 web.config 文件中删除环境变量。有关设置 web.config 中的环境变量的信息，请参阅 aspNetCore 的 environmentVariables 子元素。

常见启动错误

请参阅 Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考。参考主题介绍了阻止应用启动的大部分常见问题。

从应用中获取数据

如果应用能够响应请求，则使用终端内联中间件从应用中获取请求、连接和其他数据。有关详细信息和示例代码，请参阅解决 ASP.NET Core 项目。

应用缓慢或挂起

当应用响应缓慢或挂起请求时，获取并分析转储文件。可以使用以下任何工具获取转储文件：

- ProcDump
- DebugDiag
- WinDbg：下载 Windows 调试工具，使用 WinDbg 进行调试

远程调试

请参阅 Visual Studio 文档中的在 Visual Studio 2017 中远程调试远程 IIS 计算机上的 ASP.NET Core。

Application Insights

Application Insights 提供来自 IIS 托管的应用的遥测，包括错误日志记录和报告功能。当应用的日志记录功能变得可用时，Application Insights 只能报告应用启动后出现的错误。有关详细信息，请参阅用于 ASP.NET Core 的 Application Insights。

其他建议

有时，正常运行的应用在开发计算机上升级 .NET Core SDK 或在应用内升级包版本后立即出现故障。在某些情况下，不同的包可能在执行主要升级时中断应用。可以按照以下说明来修复其中大部分问题：

1. 删除 bin 和 obj 文件夹。
2. 清除 %UserProfile%\.nuget\packages 和 %LocalAppData%\Nuget\v3-cache 中的包缓存。
3. 还原并重新生成项目。
4. 确认在重新部署应用之前已完全删除服务器上的先前部署。

> **TIP**
>
> 清除包缓存的简便方法是从命令提示符执行 `dotnet nuget locals all --clear` 。
>
> 清除包缓存还可通过使用 nuget.exe 工具并执行命令 `nuget locals all -clear` 来完成。*nuget.exe* 不是与 Windows 桌面操作系统的捆绑安装，必须从 NuGet 网站中单独获取。

其他资源

- 解决 ASP.NET Core 项目
- 处理 ASP.NET Core 中的错误
- Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考
- ASP.NET Core 模块
- 对 Azure 应用服务上的 ASP.NET Core 启动错误进行故障排除

ASP.NET Core 模块

2019/1/10 • • Edit Online

作者：Tom Dykstra、Rick Strahl、Chris Ross、Rick Anderson、Sourabh Shirhatti、Justin Kotalik 和 Luke Latham

ASP.NET Core 模块是插入 IIS 管道的本机 IIS 模块，用于：

- 在 IIS 工作进程（`w3wp.exe`）内托管 ASP.NET Core 应用，称为进程内托管模型。
- 将 Web 请求转发到运行 Kestrel 服务器的后端 ASP.NET Core 应用，称为进程外托管模型。

受支持的 Windows 版本：

- Windows 7 或更高版本
- Windows Server 2008 R2 或更高版本

在进程内托管时，该模块会使用 IIS 进程内服务器实现，即 IIS HTTP 服务器（`IISHttpServer`）。

在进程外托管时，该模块仅适用于 Kestrel。该模块与 HTTP.sys 不兼容。

托管模型

进程内托管模型

若要配置用于进程内托管的应用，请将 `<AspNetCoreHostingModel>` 属性添加到值为 `InProcess`（进程外托管使用 `OutOfProcess` 进行设置）的应用项目文件：

```
<PropertyGroup>
  <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

如果文件中不存在 `<AspNetCoreHostingModel>` 属性，则默认值为 `OutOfProcess`。

在进程内托管时，将应用以下特征：

- 使用 IIS HTTP 服务器（`IISHttpServer`）而不是 Kestrel 服务器。
- requestTimeout 属性不适用于进程内托管。
- 不支持在应用之间共享应用池。每个应用使用一个应用池。
- 使用 Web 部署或手动将 app_offline.htm 文件置于部署中时，如果有已打开的连接，则应用可能不会立即关闭。例如，WebSocket 连接可能会延迟应用关闭。
- 应用和已安装的运行时（x64 或 x86）的体系结构（位数）必须与应用池的体系结构匹配。
- 如果使用 `WebHostBuilder`（而不是使用 CreateDefaultBuilder）手动设置应用的主机，并且应用曾经直接在 Kestrel 服务器上运行（自托管），则先调用 `UseKestrel`，再调用 `UseIISIntegration`。如果顺序颠倒，主机将无法启动。
- 检测到客户端连接断开。客户端连接断开时，HttpContext.RequestAborted 取消标记将会取消。
- GetCurrentDirectory 会返回 IIS 启动的进程的工作目录而非应用目录（例如，对于 w3wp.exe，是 C:\Windows\System32\inetsrv）。

对于设置应用的当前目录的示例代码，请参阅 CurrentDirectoryHelpers 类。调用 `SetCurrentDirectory` 方法。后续 GetCurrentDirectory 调用提供应用的目录。

进程外托管模型

若要配置进程外托管应用，请在项目文件中使用以下方法之一：

- 不指定 `<AspNetCoreHostingModel>` 属性。如果文件中不存在 `<AspNetCoreHostingModel>` 属性，则默认值为 `OutOfProcess`。
- 将 `<AspNetCoreHostingModel>` 属性的值设为 `OutOfProcess`（进程内托管设为 `InProcess`）：

```
<PropertyGroup>
  <AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

使用 Kestrel 服务器，而不是 IIS HTTP 服务器（`IISHttpServer`）。

托管模型更改

如果 `hostingModel` 设置在 web.config 文件中被更改（如 web.config 的配置部分中所述），则该模块会再循环 IIS 工作进程。

对于 IIS Express，该模块不会再循环工作进程，但改为触发当前 IIS Express 进程的正常关闭。应用的下一个请求会生成新的 IIS Express 进程。

进程名

`Process.GetCurrentProcess().ProcessName` 报告 `w3wp` / `iisexpress`（进程内）或 `dotnet`（进程外）。

ASP.NET Core 模块是插入 IIS 管道的本机 IIS 模块，用于将 Web 请求转发到后端 ASP.NET Core 应用。

受支持的 Windows 版本：

- Windows 7 或更高版本
- Windows Server 2008 R2 或更高版本

该模块仅适用于 Kestrel。该模块与 HTTP.sys 不兼容。

由于 ASP.NET Core 应用在独立于 IIS 辅助进程的进程中运行，因此该模块还处理进程管理。该模块在第一个请求到达时启动 ASP.NET Core 应用的进程，并在应用崩溃时重新启动该应用。这基本上与在 Windows 进程激活服务 (WAS) 托管的 IIS 中在进程内运行的 ASP.NET 4.x 应用中出现的行为相同。

下图说明了 IIS、ASP.NET Core 模块和应用之间的关系：

请求从 Web 到达内核模式 HTTP.sys 驱动程序。驱动程序将请求路由到网站的配置端口上的 IIS，通常为 80 (HTTP) 或 443 (HTTPS)。该模块将该请求转发到应用的随机端口（非端口 80/443）上的 Kestrel。

该模块在启动时通过环境变量指定端口，IIS 集成中间件将服务器配置为侦听 `http://localhost:{port}`。执行其他检查，拒绝不是来自该模块的请求。该模块不支持 HTTPS 转发，因此即使请求由 IIS 通过 HTTPS 接收，它们还是通过 HTTP 转发。

Kestrel 从模块获取请求后，请求会被推送到 ASP.NET Core 中间件管道中。中间件管道处理该请

求并将其作为 `HttpContext` 实例传递给应用的逻辑。IIS 集成添加的中间件会将方案、远程 IP 和 pathbase 更新到帐户以将请求转发到 Kestrel。应用的响应传递回 IIS，IIS 将响应推送回发起请求的 HTTP 客户端。

许多本机模块（如 Windows 身份验证）仍处于活动状态。要详细了解随 ASP.NET Core 模块活动的 IIS 模块，请参阅 IIS 模块与 ASP.NET Core。

ASP.NET Core 模块还可以：

- 为工作进程设置环境变量。
- 将 stdout 输出记录到文件存储器，以解决启动问题。
- 转发 Windows 身份验证令牌。

如何安装和使用 ASP.NET Core 模块

有关如何安装和使用 ASP.NET Core 模块的说明，请参阅 使用 IIS 在 Windows 上托管 ASP.NET Core。

web.config 的配置

在站点的 *web.config* 文件中使用 `system.webServer` 节点的 `aspNetCore` 部分配置 ASP.NET Core 模块。

以下 *web.config* 文件发布用于依赖框架的部署，并配置 ASP.NET Core 模块以处理站点请求：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet"
                  arguments=".\MyApp.dll"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="InProcess" />
    </system.webServer>
  </location>
</configuration>
```

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet"
                arguments=".\MyApp.dll"
                stdoutLogEnabled="false"
                stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

以下 web.config 发布用于独立部署：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath=".\MyApp.exe"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="InProcess" />
    </system.webServer>
  </location>
</configuration>
```

将 InheritInChildApplications 属性设置为 `false`，表示 <> 元素中指定的设置不会由驻留在应用子目录中的应用继承。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath=".\MyApp.exe"
                stdoutLogEnabled="false"
                stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

将应用部署为 Azure 应用服务时，`stdoutLogFile` 路径将设置为 `\\?\%home%\LogFiles\stdout`。该路径会将 stdout 日志保存到 *LogFiles* 文件夹，该文件夹是由服务自动创建的位置。

有关 IIS 子应用程序配置的信息，请参阅使用 IIS 在 Windows 上托管 ASP.NET Core。

aspNetCore 元素的属性

特性	说明	默认
`arguments`	可选的字符串属性。 processPath 中指定的可执行文件的参数。	
`disableStartUpErrorPage`	可选布尔属性。 如果为 true，将禁止显示“502.5 - 进程失败”页面，而会优先显示 web.config 中配置的 502 状态代码页面。	`false`

特性	说明	默认
`forwardWindowsAuthToken`	可选布尔属性。 如果为 true，会将令牌作为每个请求的标头“MS-ASPNETCORE-WINAUTHTOKEN”，转发到在 %ASPNETCORE_PORT% 上侦听的子进程。该进程负责在每个请求的此令牌上调用 CloseHandle。	`true`
`hostingModel`	可选的字符串属性。 将托管模型指定为进程内（`InProcess`）或进程外（`OutOfProcess`）。	`OutOfProcess`
`processesPerApplication`	可选的整数属性。 指定每个应用均可启动的 **processPath** 设置中指定的进程的实例数。 †对于进程内托管，值限制为 `1`。	默认值：`1` 最小值：`1` 最大值：`100` †
`processPath`	必需的字符串属性。 为 HTTP 请求启动进程侦听的可执行文件的路径。支持相对路径。如果路径以 `.` 开头，则该路径被视为与站点根目录相对。	
`rapidFailsPerMinute`	可选的整数属性。 指定允许 processPath 中指定的进程每分钟崩溃的次数。如果超出了此限制，模块将在剩余分钟数内停止启动该进程。 不支持进程内托管。	默认值：`10` 最小值：`0` 最大值：`100`

特性	说明	默认
`requestTimeout`	可选的 timespan 属性。 指定 ASP.NET Core 模块等待来自 %ASPNETCORE_PORT% 上侦听的进程的响应的持续时间。 在 ASP.NET Core 2.1 或更高版本附带的 ASP.NET Core 模块版本中，使用小时数、分钟数和秒数指定 `requestTimeout`。 不适用于进程内托管。对于进程内托管，该模块等待应用处理该请求。	默认值：`00:02:00` 最小值：`00:00:00` 最大值：`360:00:00`
`shutdownTimeLimit`	可选的整数属性。 检测到 app_offline.htm 文件时，模块等待可执行文件正常关闭的持续时间（以秒为单位）。	默认值：`10` 最小值：`0` 最大值：`600`
`startupTimeLimit`	可选的整数属性。 模块等待可执行文件启动端口上侦听的进程的持续时间（以秒为单位）。如果超出了此时间限制，模块将终止该进程。模块在收到新请求时尝试重新启动该进程，并在收到后续传入请求时继续尝试重新启动该进程，除非应用在上一回滚分钟内无法启动 rapidFailsPerMinute 次。 值 0（零）不被视为无限超时。	默认值：`120` 最小值：`0` 最大值：`3600`
`stdoutLogEnabled`	可选布尔属性。 如果为 true，processPath 中指定的 进程的 stdout 和 stderr 将重定向到 stdoutLogFile 中指定的文件。	`false`

特性	说明	默认
`stdoutLogFile`	可选的字符串属性。 指定在其中记录 processPath 中指定进程的 stdout 和 stderr 的相对路径或绝对路径。相对路径与站点根目录相对。以 `.` 开头的任何路径均与站点根目录相对，所有其他路径被视为绝对路径。路径中提供的任何文件夹都必须存在，以便模块创建日志文件。使用下划线分隔符，将时间戳、进程 ID 和文件扩展名 (.log) 添加到 stdoutLogFile 路径的最后一段。如果 `.\logs\stdout` 作为值提供，则在示例 stdout 日志使用进程 ID 1934 于 2018 年 2 月 5 日 19:41:32 保存时，将在 logs 文件夹中保存为 stdout_20180205194132_1934.log。	`aspnetcore-stdout`

特性	说明	默认
`arguments`	可选的字符串属性。 processPath 中指定的可执行文件的参数。	
`disableStartUpErrorPage`	可选布尔属性。 如果为 true，将禁止显示“502.5 - 进程失败”页面，而会优先显示 web.config 中配置的 502 状态代码页面。	`false`
`forwardWindowsAuthToken`	可选布尔属性。 如果为 true，会将令牌作为每个请求的标头“MS-ASPNETCORE-WINAUTHTOKEN”，转发到在 %ASPNETCORE_PORT% 上侦听的子进程。该进程负责在每个请求的此令牌上调用 CloseHandle。	`true`
`processesPerApplication`	可选的整数属性。 指定每个应用均可启动的 **processPath** 设置中指定的进程的实例数。	默认值：`1` 最小值：`1` 最大值：`100`

特性	说明	默认
`processPath`	必需的字符串属性。 为 HTTP 请求启动进程侦听的可执行文件的路径。支持相对路径。如果路径以 `.` 开头，则该路径被视为与站点根目录相对。	
`rapidFailsPerMinute`	可选的整数属性。 指定允许 processPath 中指定的进程每分钟崩溃的次数。如果超出了此限制，模块将在剩余分钟数内停止启动该进程。	默认值：`10` 最小值：`0` 最大值：`100`
`requestTimeout`	可选的 timespan 属性。 指定 ASP.NET Core 模块等待来自 %ASPNETCORE_PORT% 上侦听的进程的响应的持续时间。 在 ASP.NET Core 2.1 或更高版本附带的 ASP.NET Core 模块版本中，使用小时数、分钟数和秒数指定 `requestTimeout`。	默认值：`00:02:00` 最小值：`00:00:00` 最大值：`360:00:00`
`shutdownTimeLimit`	可选的整数属性。 检测到 app_offline.htm 文件时，模块等待可执行文件正常关闭的持续时间（以秒为单位）。	默认值：`10` 最小值：`0` 最大值：`600`
`startupTimeLimit`	可选的整数属性。 模块等待可执行文件启动端口上侦听的进程的持续时间（以秒为单位）。如果超出了此时间限制，模块将终止该进程。模块在收到新请求时尝试重新启动该进程，并在收到后续传入请求时继续尝试重新启动该进程，除非应用在上一回滚分钟内无法启动 rapidFailsPerMinute 次。 值 0（零）不被视为无限超时。	默认值：`120` 最小值：`0` 最大值：`3600`
`stdoutLogEnabled`	可选布尔属性。 如果为 true，processPath 中指定的 进程的 stdout 和 stderr 将重定向到 stdoutLogFile 中指定的文件。	`false`

特性	说明	默认
`stdoutLogFile`	可选的字符串属性。 指定在其中记录 processPath 中指定进程的 stdout 和 stderr 的相对路径或绝对路径。相对路径与站点根目录相对。以 `.` 开头的任何路径均与站点根目录相对，所有其他路径被视为绝对路径。路径中提供的任何文件夹都必须存在，以便模块创建日志文件。使用下划线分隔符，将时间戳、进程 ID 和文件扩展名 (.log) 添加到 stdoutLogFile 路径的最后一段。如果 `.\logs\stdout` 作为值提供，则在示例 stdout 日志使用进程 ID 1934 于 2018 年 2 月 5 日 19:41:32 保存时，将在 logs 文件夹中保存为 stdout_20180205194132_1934.log。	`aspnetcore-stdout`

特性	说明	默认
`arguments`	可选的字符串属性。 processPath 中指定的可执行文件的参数。	
`disableStartUpErrorPage`	可选布尔属性。 如果为 true，将禁止显示“502.5 - 进程失败”页面，而会优先显示 web.config 中配置的 502 状态代码页面。	`false`
`forwardWindowsAuthToken`	可选布尔属性。 如果为 true，会将令牌作为每个请求的标头“MS-ASPNETCORE-WINAUTHTOKEN”，转发到在 %ASPNETCORE_PORT% 上侦听的子进程。该进程负责在每个请求的此令牌上调用 CloseHandle。	`true`
`processesPerApplication`	可选的整数属性。 指定每个应用均可启动的 **processPath** 设置中指定的进程的实例数。	默认值：`1` 最小值：`1` 最大值：`100`

特性	说明	默认
`processPath`	必需的字符串属性。 为 HTTP 请求启动进程侦听的可执行文件的路径。支持相对路径。如果路径以 `.` 开头，则该路径被视为与站点根目录相对。	
`rapidFailsPerMinute`	可选的整数属性。 指定允许 processPath 中指定的进程每分钟崩溃的次数。如果超出了此限制，模块将在剩余分钟数内停止启动该进程。	默认值：`10` 最小值：`0` 最大值：`100`
`requestTimeout`	可选的 timespan 属性。 指定 ASP.NET Core 模块等待来自 %ASPNETCORE_PORT% 上侦听的进程的响应的持续时间。 在 ASP.NET Core 2.0 或更早版本附带的 ASP.NET Core 模块版本中，必须仅使用整分钟数指定 `requestTimeout`，否则默认为 2 分钟。	默认值：`00:02:00` 最小值：`00:00:00` 最大值：`360:00:00`
`shutdownTimeLimit`	可选的整数属性。 检测到 app_offline.htm 文件时，模块等待可执行文件正常关闭的持续时间（以秒为单位）。	默认值：`10` 最小值：`0` 最大值：`600`
`startupTimeLimit`	可选的整数属性。 模块等待可执行文件启动端口上侦听的进程的持续时间（以秒为单位）。如果超出了此时间限制，模块将终止该进程。模块在收到新请求时尝试重新启动该进程，并在收到后续传入请求时继续尝试重新启动该进程，除非应用在上一回滚分钟内无法启动 rapidFailsPerMinute 次。	默认值：`120` 最小值：`0` 最大值：`3600`
`stdoutLogEnabled`	可选布尔属性。 如果为 true，processPath 中指定的 进程的 stdout 和 stderr 将重定向到 stdoutLogFile 中指定的文件。	`false`

特性	说明	默认
`stdoutLogFile`	可选的字符串属性。 指定在其中记录 processPath 中指定进程的 stdout 和 stderr 的相对路径或绝对路径。相对路径与站点根目录相对。以 `.` 开头的任何路径均与站点根目录相对，所有其他路径被视为绝对路径。路径中提供的任何文件夹都必须存在，以便模块创建日志文件。使用下划线分隔符，将时间戳、进程 ID 和文件扩展名 (.log) 添加到 stdoutLogFile 路径的最后一段。如果 `.\logs\stdout` 作为值提供，则在示例 stdout 日志使用进程 ID 1934 于 2018 年 2 月 5 日 19:41:32 保存时，将在 logs 文件夹中保存为 stdout_20180205194132_1934.log。	`aspnetcore-stdout`

设置环境变量

可以为 `processPath` 属性中的进程指定环境变量。使用 `environmentVariables` 集合元素的 `environmentVariable` 子元素指定环境变量。本部分中设置的环境变量优先于系统环境变量。

以下示例设置了两个环境变量。`ASPNETCORE_ENVIRONMENT` 将应用的环境配置为 `Development`。开发人员可能会暂时在 web.config 文件中设置此值，以便在调试应用异常时强制加载开发人员异常页面。`CONFIG_DIR` 是用户定义的环境变量的一个示例，其中开发人员已写入可在启动时读取值的代码以便形成用于加载应用配置文件的路径。

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile="\\?\%home%\LogFiles\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
    <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
  </environmentVariables>
</aspNetCore>
```

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile="\\?\%home%\LogFiles\stdout">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
    <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
  </environmentVariables>
</aspNetCore>
```

> **WARNING**
> 在不可访问不受信任的网络(如 Internet)的暂存服务器和测试服务器上，仅将 `ASPNETCORE_ENVIRONMENT` 环境变量设置为 `Development` 。

app_offline.htm

如果在应用的根目录中检测到名为 “app_offline.htm”的文件，ASP.NET Core 模块将尝试正常关闭应用并停止处理传入请求。如果应用在 `shutdownTimeLimit` 中定义的秒数之后仍在运行，ASP.NET Core 模块将终止正在运行的进程。

存在 app_offline.htm 文件时，ASP.NET Core 模块会通过发送回 app_offline.htm 文件的内容来响应请求。删除 app_offline.htm 文件后，下一个请求将启动应用。

使用进程外托管模型时，如果有已打开的连接，则应用可能不会立即关闭。例如，WebSocket 连接可能会延迟应用关闭。

启动错误页面

进程内和进程外托管在无法启动应用时均会生成自定义错误页面。

如果 ASP.NET Core 模块无法找到进程内或进程外请求处理程序，则会显示“500.0 - 进程内/进程外处理程序加载失败”状态代码页。

对于进程内托管，如果 ASP.NET Core 模块无法启动应用，则会显示“500.30 - 启动失败”状态代码页。

对于进程外托管，如果 ASP.NET Core 模块无法启动后端进程或后端进程启动但无法在配置的端口上侦听，则将显示“502.5 - 进程失败”状态代码页。

若要禁止显示此页面并还原为默认 IIS 5xx 状态代码页，请使用 `disableStartUpErrorPage` 属性。有关配置自定义错误消息的详细信息，请参阅 HTTP 错误 <httpErrors>。

如果 ASP.NET Core 模块无法启动后端进程或后端进程启动但无法在配置的端口上侦听，则将显示“502.5 - 进程失败”状态代码页。若要禁止显示此页面并还原为默认 IIS 502 状态代码页面，请使用 `disableStartUpErrorPage` 属性。有关配置自定义错误消息的详细信息，请参阅 HTTP 错误 <httpErrors>。

日志创建和重定向

如果已设置 `aspNetCore` 元素的 `stdoutLogEnabled` 和 `stdoutLogFile` 属性，ASP.NET Core 模块会将 stdout 和 stderr 控制台输出重定向到磁盘。`stdoutLogFile` 路径中的任何文件夹都必须存在，以便模块创建日志文件。应用池必须对写入日志的位置具有写入权限（使用 `IIS AppPool\<app_pool_name>` 提供写入权限）。

日志不会旋转，除非回收/重新启动进程。宿主负责限制日志占用的磁盘空间。

仅建议使用 stdout 日志来解决应用启动问题。不要将 stdout 日志用于常规应用日志记录。对于 ASP.NET Core 应用中的例程日志记录，使用限制日志文件大小和旋转日志的日志记录库。有关详细信息，请参阅第三方日志记录提供程序。

创建日志文件时，将自动添加时间戳和文件扩展名。日志文件名是通过将时间戳、进程 ID 和文件扩展名 (.log) 附加到由下划线分隔的 `stdoutLogFile` 路径的最后一段（通常为 stdout）组合而成的。如果 `stdoutLogFile` 路径以 stdout 结尾，则 PID 为 1934 且于 2018 年 2 月 5 日 19:42:32 创建的应用的日志的文件名为 stdout_20180205194132_1934.log。

如果 `stdoutLogEnabled` 为 false，则会捕获应用启动时发生的错误，并将其发送到事件日志，最大为 30 KB。启动后，将丢弃所有其他日志。

以下示例 `aspNetCore` 元素为 Azure 应用服务中托管的应用配置 stdout 日志记录。本地日志记录可以接受本地路径或网络共享路径。确认应用池用户标识是否已提供写入路径的权限。

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="InProcess">
</aspNetCore>
```

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile="\\?\%home%\LogFiles\stdout">
</aspNetCore>
```

增强的诊断日志

可以配置 ASP.NET Core 模块提供内容以提供增强的诊断日志。向 web.config 中的 `<aspNetCore>` 元素添加 `<handlerSettings>` 元素。将 `debugLevel` 设置为 `TRACE` 将提供更准确的诊断信息：

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="InProcess">
  <handlerSettings>
    <handlerSetting name="debugFile" value="aspnetcore-debug.log" />
    <handlerSetting name="debugLevel" value="FILE,TRACE" />
  </handlerSettings>
</aspNetCore>
```

调试级别（`debugLevel`）值可以同时包含级别和位置。

级别（详细程度由低到高）：

- 错误
- 警告
- 信息
- TRACE

位置(允许多个位置):

- CONSOLE
- 事件日志
- 文件

还可以通过环境变量提供处理程序设置:

- `ASPNETCORE_MODULE_DEBUG_FILE` – 调试日志文件的路径。(默认值:aspnetcore debug.log)
- `ASPNETCORE_MODULE_DEBUG` – 调试级别设置。

> **WARNING**
> 部署中启用的调试日志记录的时间不要超出排除故障所需的时间。日志大小不限。持续启用调试日志可能耗尽可用磁盘空间并导致服务器或应用服务崩溃。

有关 web.config 文件中的 `aspNetCore` 元素的示例,请参阅 web.config 的配置。

代理配置使用 HTTP 协议和配对令牌

仅适用于进程外托管。

在 ASP.NET Core 模块和 Kestrel 之间创建的代理使用 HTTP 协议。使用 HTTP 是一种性能优化,其中模块和 Kestrel 之间的流量发生于脱离网络接口的环回地址。因此,不存在从脱离服务器的位置窃取模块和 Kestrel 之间的流量的风险。

配对令牌用于保证 Kestrel 收到的请求已由 IIS 代理且不来自某些其他源。模块已创建配对令牌并将其设置到环境变量(`ASPNETCORE_TOKEN`)。此外,配对令牌还设置到每个代理请求的标头(`MS-ASPNETCORE-TOKEN`)。IIS 中间件检查它所接收的每个请求,以确认配对令牌标头值与环境变量值相匹配。如果令牌值不匹配,则将记录请求并拒绝该请求。无法从脱离服务器的位置访问配对令牌环境变量及模块和 Kestrel 之间的流量。如果不知道配对令牌值,攻击者就无法提交绕过 IIS 中间件中的检查的请求。

具有 IIS 共享配置的 ASP.NET Core 模块

ASP.NET Core 模块安装程序使用系统帐户的权限运行。由于本地系统帐户没有对 IIS 共享配置所用的共享路径的修改权限,因此在尝试配置共享上的 applicationHost.config 中的模块设置时,安装程序将遇到拒绝访问错误。使用 IIS 共享配置时,请执行以下步骤:

1. 禁用 IIS 共享配置。
2. 运行安装程序。
3. 将已更新的 applicationHost.config 文件导出到共享。
4. 重新启用 IIS 共享配置。

模块版本和托管捆绑安装程序日志

若要确定已安装 ASP.NET Core 模块的版本,请执行以下操作:

1. 在托管系统上,导航到 %windir%\System32\inetsrv。

2. 找到 aspnetcore.dll 文件。
3. 右键单击该文件，然后从上下文菜单中选择“属性”。
4. 选择“详细信息”选项卡。“文件版本”和“产品版本”表示模块的已安装版本。

在 C:\Users\%UserName%\AppData\Local\Temp 中找到了模块的托管捆绑安装程序日志。该文件名为 dd_DotNetCoreWinSvrHosting__<timestamp>*000_AspNetCoreModule_x64.log*。

模块、架构和配置文件位置

模块

IIS (x86/amd64)：

- %windir%\System32\inetsrv\aspnetcore.dll
- %windir%\SysWOW64\inetsrv\aspnetcore.dll
- %ProgramFiles%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll
- %ProgramFiles(x86)%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll

IIS Express (x86/amd64)：

- %ProgramFiles%\IIS Express\aspnetcore.dll
- %ProgramFiles(x86)%\IIS Express\aspnetcore.dll
- %ProgramFiles%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll
- %ProgramFiles(x86)%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll

架构

IIS

- %windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml
- %windir%\System32\inetsrv\config\schema\aspnetcore_schema_v2.xml

IIS Express

- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml
- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema_v2.xml

配置

IIS

- %windir%\System32\inetsrv\config\applicationHost.config

IIS Express

- %ProgramFiles%\IIS Express\config\templates\PersonalWebServer\applicationHost.config

可以通过在 applicationHost.config 文件中搜索 aspnetcore 找到这些文件。

其他资源

- 使用 IIS 在 Windows 上托管 ASP.NET Core
- ASP.NET Core 模块 GitHub 存储库（引用源）
- IIS 模块与 ASP.NET Core

Visual Studio 中针对 ASP.NET Core 的开发时 IIS 支持

2019/1/10 • • Edit Online

作者：Sourabh Shirhatti、Luke Latham

本文介绍了对调试（在 Windows Server 上的 IIS 后方运行的）ASP.NET Core 应用的 Visual Studio 支持。本主题逐步介绍如何启用此功能并设置项目。

系统必备

- Visual Studio（适用于 Windows）
- ASP.NET 和 Web 开发工作负荷
- .NET Core 跨平台开发工作负荷
- X.509 安全证书

启用 IIS

1. 导航到“控制面板” > “程序” > “程序和功能” > “打开或关闭 Windows 功能”（位于屏幕左侧）。
2. 选中“Internet Information Services”复选框。

IIS 安装可能需要重启系统。

配置 IIS

IIS 必须具有具备以下配置的网站：

- 与应用启动配置文件 URL 主机名匹配的主机名。
- 使用分配的证书绑定端口 443。

例如，所添加网站的主机名设置为“localhost”（在本主题之后部分启动配置文件也将使用“localhost”）。端口设置为“443”(HTTPS)。“IIS Express 开发证书”已分配给该网站，但可使用任何有效的证书：

如果 IIS 安装已经有一个默认网站，且其主机名与应用的启动配置文件 URL 主机名相匹配：

- 为端口 443 (HTTPS) 添加端口绑定。
- 向网站分配一个有效证书。

在 Visual Studio 中启用开发时 IIS 支持

1. 启动 Visual Studio 安装程序。
2. 选择“开发时 IIS 支持”组件。该组件在“ASP.NET 和 Web 开发”工作负荷的“摘要”面板中列为可选。该组件将安装 ASP.NET Core 模块，该模块是运行具有 IIS 的 ASP.NET Core 应用所需的本机 IIS 模块。

配置项目

HTTPS 重定向

对于新项目，请选中"新建 ASP.NET Core Web 应用程序"窗口中的"配置 HTTPS"复选框：

在现有项目中，通过调用 UseHttpsRedirection 扩展方法在 `Startup.Configure` 中使用 HTTPS 重定向中间件：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseMvc();
}
```

IIS 启动配置文件

创建新的启动配置文件以添加开发时 IIS 支持：

1. 对于配置文件，请选择“新建”按钮。在弹出窗口中将该配置文件命名为“IIS”。选择“确定”创建配置文件。
2. 对于“启动”设置，请从列表中选择“IIS”。
3. 选中“启动浏览器”复选框并提供终结点 URL。使用 HTTPS 协议。本示例使用 `https://localhost/WebApplication1`。
4. 在“环境变量”部分中，选择“添加”按钮。提供一个密钥为 `ASPNETCORE_ENVIRONMENT` 和值为 `Development` 的环境变量。
5. 在“Web 服务器设置”区域中，设置“应用 URL”。本示例使用 `https://localhost/WebApplication1`。
6. 保存配置文件。

或者，可以将启动配置文件手动添加到应用中的 launchSettings.json 文件：

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iis": {
      "applicationUrl": "https://localhost/WebApplication1",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS": {
      "commandName": "IIS",
      "launchBrowser": true,
      "launchUrl": "https://localhost/WebApplication1",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

运行该项目

在 Visual Studio 中：

- 确认已将生成配置下拉列表设置为“调试”。
- 将“运行”按钮设置为 IIS 配置文件，然后选择此按钮以启动该应用。

如果不以管理员身份运行，Visual Studio 可能会提示重启。如果出现提示，请重启 Visual Studio。

如果使用不受信任的开发证书，则浏览器可能会要求你为不受信任的证书创建异常。

> **NOTE**
> 通过仅我的代码调试“发布”生成配置，从而导致降低编译器优化体验。例如，不会遇到断点。

其他资源

- 使用 IIS 在 Windows 上托管 ASP.NET Core
- ASP.NET Core 模块简介
- ASP.NET Core 模块配置参考
- Enforce HTTPS

IIS 模块与 ASP.NET Core

2018/12/14 • • Edit Online

作者：Luke Latham

某些本机 IIS 模块和所有 IIS 托管模块无法处理 ASP.NET Core 应用的请求。在许多情况下，ASP.NET Core 提供了 IIS 本机和托管模块解决的方案的替代方案。

本机模块

该表指示在使用 ASP.NET Core 应用和 ASP.NET Core 模块时正常工作的本机 IIS 模块。

模块	在 ASP.NET CORE 应用内可用	ASP.NET CORE 选项
匿名身份验证 `AnonymousAuthenticationModule`	是	
基本身份验证 `BasicAuthenticationModule`	是	
客户端证书**映射身份**验证 `CertificateMappingAuthenticationModule`	是	
CGI `CgiModule`	否	
配置验证 `ConfigurationValidationModule`	是	
HTTP 错误 `CustomErrorModule`	No	状态代码页中间件
自定义日志记录 `CustomLoggingModule`	是	
默认文档 `DefaultDocumentModule`	No	默认文件中间件
摘要式身份验证 `DigestAuthenticationModule`	是	
目录浏览 `DirectoryListingModule`	No	目录浏览中间件
动态压缩 `DynamicCompressionModule`	是	响应压缩中间件
跟踪 `FailedRequestsTracingModule`	是	ASP.NET Core 日志记录

模块	在 ASP.NET CORE 应用内可用	ASP.NET CORE 选项
文件缓存 `FileCacheModule`	No	响应缓存中间件
HTTP 缓存 `HttpCacheModule`	否	响应缓存中间件
HTTP 日志记录 `HttpLoggingModule`	是	ASP.NET Core 日志记录
HTTP 重定向 `HttpRedirectionModule`	是	URL 重写中间件
IIS 客户端证书映射身份验证 `IISCertificateMappingAuthenticationModule`	是	
IP 和域限制 `IpRestrictionModule`	是	
ISAPI 筛选器 `IsapiFilterModule`	是	中间件
ISAPI `IsapiModule`	是	中间件
协议支持 `ProtocolSupportModule`	是	
请求筛选 `RequestFilteringModule`	是	URL 重写中间件 `IRule`
请求监视器 `RequestMonitorModule`	是	
URL 重写† `RewriteModule`	是	URL 重写中间件
服务器端包括 `ServerSideIncludeModule`	No	
静态压缩 `StaticCompressionModule`	No	响应压缩中间件
静态内容 `StaticFileModule`	No	静态文件中间件
令牌缓存 `TokenCacheModule`	是	
URI 缓存 `UriCacheModule`	是	

模块	在 ASP.NET CORE 应用内可用	ASP.NET CORE 选项
URL 授权 `UrlAuthorizationModule`	是	ASP.NET Core 标识
Windows 身份验证 `WindowsAuthenticationModule`	是	

†由于目录结构中的更改，URL 重写模块的 `isFile` 和 `isDirectory` 匹配类型不适用于 ASP.NET Core 应用。

托管模块

当应用池的 .NET CLR 版本设置为"无托管代码"时，托管 ASP.NET Core 应用的托管模块无法使用。ASP.NET Core 在几种情况下提供了中间件替代方案。

模块	ASP.NET CORE 选项
AnonymousIdentification	
DefaultAuthentication	
FileAuthorization	
FormsAuthentication	Cookie 身份验证中间件
OutputCache	响应缓存中间件
配置文件	
RoleManager	
ScriptModule-4.0	
会话	会话中间件
UrlAuthorization	
UrlMappingsModule	URL 重写中间件
UrlRoutingModule-4.0	ASP.NET Core 标识
WindowsAuthentication	

IIS 管理器应用程序更改

使用 IIS 管理器配置设置时，应用的 web.config 文件已更改。如果部署应用并包括 web.config，则使用 IIS 管理器所做的任何更改都会被部署的 web.config 文件覆盖。如果更改服务器的 web.config 文件，请立即将服务器上已更新的 web.config 文件复制到本地项目。

禁用 IIS 模块

如果在服务器级别配置了必须为应用禁用的 IIS 模块，则应用 web.config 文件还可以禁用该模块。将模块留在原位并使用配置设置（如果可用）将其停用或从应用中移除模块。

模块停用

许多模块提供一个配置设置，允许在不从应用中删除模块的情况下禁用它们。这是停用模块最简单、快捷的方式。例如，可使用 web.config 中的 `<httpRedirect>` 元素禁用 HTTP 重定向模块：

```
<configuration>
  <system.webServer>
    <httpRedirect enabled="false" />
  </system.webServer>
</configuration>
```

有关通过配置设置禁用模块的详细信息，请按照 IIS <system.webServer> 的子元素部分中的链接进行操作。

模块删除

如果选择使用 web.config 中的设置删除模块，请先解锁此模块和 web.config 的 `<modules>` 部分：

1. 解锁服务器级别的模块。在 IIS 管理器"连接"边栏中选择 IIS 服务器。打开 IIS 区域中的模块。在列表中选择该模块。在右侧的"操作"边栏中，选择"解锁"。解锁尽可能多稍后计划从 web.config 中删除的模块。

2. 在不使用 web.config 的 `<modules>` 部分的情况下部署应用。如果使用包含 `<modules>` 部分的 web.config 部署应用，但未先在 IIS 管理器中解锁该部分，则配置管理器会在尝试解锁该部分时引发异常。因此，请在不使用 `<modules>` 部分的情况下部署该应用。

3. 解锁 web.config 的 `<modules>` 部分。在"连接"边栏中，选择"站点"中的网站。在"管理"区域中，打开"配置编辑器"。使用导航控件选择 `system.webServer/modules` 部分。在右侧的"操作"边栏中，选择"解锁"该部分。

4. 此时，可使用 `<remove>` 元素将 `<modules>` 部分添加到 web.config 文件，从应用中删除模块。可添加多个 `<remove>` 元素来删除多个模块。如果在服务器上更改 web.config，请立即在本地对项目的 web.config 文件执行相同的更改。以这种方式移除模块不会影响模块的使用与服务器上的其他应用程序。

   ```
   <configuration>
    <system.webServer>
      <modules>
        <remove name="MODULE_NAME" />
      </modules>
    </system.webServer>
   </configuration>
   ```

IIS 模块还可以使用 Appcmd.exe 删除。该命令中提供 `MODULE_NAME` 和 `APPLICATION_NAME`：

```
Appcmd.exe delete module MODULE_NAME /app.name:APPLICATION_NAME
```

例如，从默认网站中删除 `DynamicCompressionModule`：

```
%windir%\system32\inetsrv\appcmd.exe delete module DynamicCompressionModule /app.name:"Default Web Site"
```

最小模块配置

要求运行 ASP.NET Core 应用的模块只有匿名身份验证模块和 ASP.NET Core 模块。

URI 缓存模块（`UriCacheModule`）允许 IIS 在 URL 级别缓存网站配置。不使用此模块的话，即使重复请求相同的 URL，IIS 也必须读取并分析每个请求的配置。分析每个请求的配置会导致严重的性能损失。虽然托管的 ASP.NET Core 应用并非严格要求运行 URI 缓存模块，但我们建议为所有 ASP.NET Core 部署启用 URI 缓存模块。

HTTP 缓存模块（`HttpCacheModule`）实现 IIS 输出缓存以及用于在 HTTP.sys 缓存中缓存项目的逻辑。如果不使用此模块，内容不再以内核模式缓存，并且缓存配置文件将被忽略。删除 HTTP 缓存模块通常会对性能和资源使用情况产生不利影响。虽然托管的 ASP.NET Core 应用程序并非严格要求运行 HTTP 缓存模块，但我们建议为所有 ASP.NET Core 部署启用 HTTP 缓存模块。

其他资源

- 使用 IIS 在 Windows 上托管 ASP.NET Core
- IIS 体系结构简介：IIS 中的模块
- IIS 模块概述
- 自定义 IIS 7.0 角色和模块
- IIS `<system.webServer>`

ASP.NET Core 中的 Kestrel Web 服务器实现

2019/1/10 • • Edit Online

作者：Tom Dykstra、Chris Ross 和 Stephen Halter

对于本主题的 1.1 版本，请下载 ASP.NET Core（版本 1.1，PDF）中的 Kestrel Web 服务器实现。

Kestrel 是一个跨平台的适用于 ASP.NET Core 的 Web 服务器。Kestrel 是 Web 服务器，默认包括在 ASP.NET Core 项目模板中。

Kestrel 支持以下方案：

- HTTPS
- 用于启用 WebSocket 的不透明升级
- 用于获得 Nginx 高性能的 Unix 套接字
- HTTP/2（除非在 macOS† 上）

macOS 的未来版本将支持 †HTTP/2。

- HTTPS
- 用于启用 WebSocket 的不透明升级
- 用于获得 Nginx 高性能的 Unix 套接字

.NET Core 支持的所有平台和版本均支持 Kestrel。

查看或下载示例代码（如何下载）

HTTP/2 支持

如果满足以下基本要求，将为 ASP.NET Core 应用提供 HTTP/2：

- 操作系统†
  - Windows Server 2016/Windows 10 或更高版本‡
  - 具有 OpenSSL 1.0.2 或更高版本的 Linux（例如，Ubuntu 16.04 或更高版本）
- 目标框架：.NET Core 2.2 或更高版本
- 应用程序层协议协商 (ALPN) 连接
- TLS 1.2 或更高版本的连接

macOS 的未来版本将支持 †HTTP/2。‡Kestrel 在 Windows Server 2012 R2 和 Windows 8.1 上对 HTTP/2 的支持有限。支持受限是因为可在这些操作系统上使用的受支持 TLS 密码套件列表有限。可能需要使用椭圆曲线数字签名算法 (ECDSA) 生成的证书来保护 TLS 连接。

如果已建立 HTTP/2 连接，HttpRequest.Protocol 会报告 `HTTP/2`。

默认情况下，禁用 HTTP/2。有关配置的详细信息，请参阅 Kestrel 选项和 ListenOptions.Protocols 部分。

何时结合使用 Kestrel 和反向代理

可以单独使用 Kestrel，也可以将其与反向代理服务器（如 Internet Information Services (IIS)、Nginx 或 Apache）结合使用。反向代理服务器接收来自网络的 HTTP 请求，并将这些请求

转发到 Kestrel。

Kestrel 用作边缘(面向 Internet)Web 服务器:

Kestrel 用于反向代理配置:

无论配置是否使用反向代理服务器——，都是从 Internet 接收请求的 ASP.NET Core 2.1 或更高版本应用的支持托管配置。

在没有反向代理服务器的情况下用作边缘服务器的 Kestrel 不支持在多个进程间共享相同的 IP 和端口。如果将 Kestrel 配置为侦听某个端口，Kestrel 会处理该端口的所有流量(无视请求的 `Host` 标头)。可以共享端口的反向代理能在唯一的 IP 和端口上将请求转发至 Kestrel。

即使不需要反向代理服务器，使用反向代理服务器可能也是个不错的选择。

反向代理:

- 可以限制所承载的应用中的公开的公共外围应用。
- 提供额外的配置和防护层。
- 可以更好地与现有基础结构集成。
- 简化了负载均和和安全通信 (HTTPS) 配置。仅反向代理服务器需要 X.509 证书，并且该服务器可使用普通 HTTP 在内部网络上与应用服务器通信。

> **WARNING**
>
> 采用反向代理配置进行托管需要主机筛选。

如何在 ASP.NET Core 应用中使用 Kestrel

Microsoft.AspNetCore.App 元包中包括 Microsoft.AspNetCore.Server.Kestrel 包(ASP.NET Core 2.1 或更高版本)。

默认情况下，ASP.NET Core 项目模板使用 Kestrel。在 Program.cs 中，模板代码调用 CreateDefaultBuilder，后者在后台调用 UseKestrel。

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
```

若要在调用 `CreateDefaultBuilder` 后提供其他配置，请使用 `ConfigureKestrel`:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            // Set properties and call methods on options
        });
```

若要在调用 `CreateDefaultBuilder` 后提供其他配置，请调用 UseKestrel：

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
        {
            // Set properties and call methods on options
        });
```

Kestrel 选项

Kestrel Web 服务器具有约束配置选项，这些选项在面向 Internet 的部署中尤其有用。

可在 KestrelServerOptions 类的 Limits 属性上设置约束。`Limits` 属性包含 KestrelServerLimits 类的实例。

客户端最大连接数

MaxConcurrentConnections
MaxConcurrentUpgradedConnections

可使用以下代码为整个应用设置并发打开的最大 TCP 连接数：

```
.ConfigureKestrel((context, options) =>
{
    options.Limits.MaxConcurrentConnections = 100;
    options.Limits.MaxConcurrentUpgradedConnections = 100;
    options.Limits.MaxRequestBodySize = 10 * 1024;
    options.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Listen(IPAddress.Loopback, 5000);
    options.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
```

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
        {
            options.Limits.MaxConcurrentConnections = 100;
        });
```

对于已从 HTTP 或 HTTPS 升级到另一个协议（例如，Websocket 请求）的连接，有一个单独的限制。连接升级后，不会计入 `MaxConcurrentConnections` 限制。

```
.ConfigureKestrel((context, options) =>
{
    options.Limits.MaxConcurrentConnections = 100;
    options.Limits.MaxConcurrentUpgradedConnections = 100;
    options.Limits.MaxRequestBodySize = 10 * 1024;
    options.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Listen(IPAddress.Loopback, 5000);
    options.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
```

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
        {
            options.Limits.MaxConcurrentUpgradedConnections = 100;
        });
```

默认情况下，最大连接数不受限制 (NULL)。

请求正文最大大小

MaxRequestBodySize

默认的请求正文最大大小为 30,000,000 字节，大约 28.6 MB。

在 ASP.NET Core MVC 应用中替代限制的推荐方法是在操作方法上使用 RequestSizeLimit 属性：

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

以下示例演示如何为每个请求上的应用配置约束：

```
.ConfigureKestrel((context, options) =>
{
    options.Limits.MaxConcurrentConnections = 100;
    options.Limits.MaxConcurrentUpgradedConnections = 100;
    options.Limits.MaxRequestBodySize = 10 * 1024;
    options.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Listen(IPAddress.Loopback, 5000);
    options.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
```

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
        {
            options.Limits.MaxRequestBodySize = 10 * 1024;
        });
```

可在中间件中替代特定请求的设置：

```
app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;

    var minRequestRateFeature =
        context.Features.Get<IHttpMinRequestBodyDataRateFeature>();
    var minResponseRateFeature =
        context.Features.Get<IHttpMinResponseDataRateFeature>();

    if (minRequestRateFeature != null)
    {
        minRequestRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    if (minResponseRateFeature != null)
    {
        minResponseRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }
```

如果在应用开始读取请求后尝试配置请求限制，则会引发异常。`IsReadOnly` 属性指示 `MaxRequestBodySize` 属性处于只读状态，意味已经无法再配置限制。

请求正文最小数据速率

MinRequestBodyDataRate
MinResponseDataRate

Kestrel 每秒检查一次数据是否以指定的速率（字节/秒）传入。如果速率低于最小值，则连接超时。宽限期是 Kestrel 提供给客户端用于将其发送速率提升到最小值的时间量；在此期间不会检查速率。宽限期有助于避免最初由于 TCP 慢启动而以较慢速率发送数据的连接中断。

默认的最小速率为 240 字节/秒，包含 5 秒的宽限期。

最小速率也适用于响应。除了属性和接口名称中具有 `RequestBody` 或 `Response` 以外，用于设置请求限制和响应限制的代码相同。

以下示例演示如何在 Program.cs 中配置最小数据速率：

```
.ConfigureKestrel((context, options) =>
{
    options.Limits.MaxConcurrentConnections = 100;
    options.Limits.MaxConcurrentUpgradedConnections = 100;
    options.Limits.MaxRequestBodySize = 10 * 1024;
    options.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Listen(IPAddress.Loopback, 5000);
    options.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
```

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
        {
            options.Limits.MinRequestBodyDataRate =
                new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
            options.Limits.MinResponseDataRate =
                new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
        });
```

可以在中间件中替代每个请求的最低速率限制：

```
app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;

    var minRequestRateFeature =
        context.Features.Get<IHttpMinRequestBodyDataRateFeature>();
    var minResponseRateFeature =
        context.Features.Get<IHttpMinResponseDataRateFeature>();

    if (minRequestRateFeature != null)
    {
        minRequestRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    if (minResponseRateFeature != null)
    {
        minResponseRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }
```

由于协议支持请求多路复用，HTTP/2 不支持基于每个请求修改速率限制，因此 HTTP/2 请求的 `HttpContext.Features` 中不存在前面示例中引用的速率特性。通过 `KestrelServerOptions.Limits` 配置的服务器范围的速率限制仍适用于 HTTP/1.x 和 HTTP/2 连接。

每个连接的最大流

`Http2.MaxStreamsPerConnection` 限制每个 HTTP/2 连接的并发请求流的数量。拒绝过多的流。

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            options.Limits.Http2.MaxStreamsPerConnection = 100;
        });
```

默认值为 100。

标题表大小

HPACK 解码器解压缩 HTTP/2 连接的 HTTP 标头。`Http2.HeaderTableSize` 限制 HPACK 解码器使用的标头压缩表的大小。该值以八位字节提供，且必须大于零 (0)。

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            options.Limits.Http2.HeaderTableSize = 4096;
        });
```

默认值为 4096。

最大帧大小

`Http2.MaxFrameSize` 指示要接收的 HTTP/2 连接帧有效负载的最大大小。该值以八位字节提供，必须介于 2^14 (16,384) 和 2^24-1 (16,777,215) 之间。

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            options.Limits.Http2.MaxFrameSize = 16384;
        });
```

默认值为 2^14 (16,384)。

最大请求标头大小

`Http2.MaxRequestHeaderFieldSize` 表示请求标头值的允许的最大大小（用八进制表示）。此限制同时适用于压缩和未压缩表示形式中的名称和值。该值必须大于零 (0)。

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            options.Limits.Http2.MaxRequestHeaderFieldSize = 8192;
        });
```

默认值为 8,192。

初始连接窗口大小

`Http2.InitialConnectionWindowSize` 表示服务器一次性缓存的最大请求主体数据大小（每次连接时在所有请求（流）中汇总，以字节为单位）。请求也受 `Http2.InitialStreamWindowSize` 限制。该值必须大于或等于 65,535，并小于 2^31 (2,147,483,648)。

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            options.Limits.Http2.InitialConnectionWindowSize = 131072;
        });
```

默认值为 128 KB (131,072)。

初始流窗口大小

`Http2.InitialStreamWindowSize` 表示服务器针对每个请求(流)的一次性缓存的最大请求主体数据大小(以字节为单位)。请求也受 `Http2.InitialStreamWindowSize` 限制。该值必须大于或等于 65,535，并小于 2^31 (2,147,483,648)。

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            options.Limits.Http2.InitialStreamWindowSize = 98304;
        });
```

默认值为 96 KB (98,304)。

有关其他 Kestrel 选项和限制的信息，请参阅：

- KestrelServerOptions
- KestrelServerLimits
- ListenOptions

终结点配置

默认情况下，ASP.NET Core 绑定到：

- `http://localhost:5000`
- `https://localhost:5001`(存在本地开发证书时)

开发证书会创建于以下情况：

- 安装了 .NET Core SDK 时。
- dev-certs tool 用于创建证书。

部分浏览器需要获取信任本地开发证书的显示权限。

ASP.NET Core 2.1 及更高版本的项目模板将应用配置为默认情况下在 HTTPS 上运行，并包括 HTTPS 重定向和 HSTS 支持。

在 KestrelServerOptions 上调用 Listen 或 ListenUnixSocket 方法，从而配置 Kestrel 的 URL 前缀和端口。

`UseUrls`、`--urls` 命令行参数、`urls` 主机配置键以及 `ASPNETCORE_URLS` 环境变量也有用，但具有本节后面注明的限制(必须要有可用于 HTTPS 终结点配置的默认证书)。

ASP.NET Core 2.1 `KestrelServerOptions` 配置：

ConfigureEndpointDefaults(Action<ListenOptions>)

指定一个为每个指定的终结点运行的配置 `Action`。多次调用 `ConfigureEndpointDefaults`，用最

新指定的 `Action` 替换之前的 `Action`。

ConfigureHttpsDefaults(Action<HttpsConnectionAdapterOptions>)

指定一个为每个 HTTPS 终结点运行的配置 `Action`。多次调用 `ConfigureHttpsDefaults`，用最新指定的 `Action` 替换之前的 `Action`。

Configure(IConfiguration)

创建配置加载程序，用于设置将 IConfiguration 作为输入的 Kestrel。配置必须针对 Kestrel 的配置节。

ListenOptions.UseHttps

将 Kestrel 配置为使用 HTTPS。

`ListenOptions.UseHttps` 扩展：

- `UseHttps` – 将 Kestrel 配置为使用 HTTPS，采用默认证书。如果没有配置默认证书，则会引发异常。
- `UseHttps(string fileName)`
- `UseHttps(string fileName, string password)`
- `UseHttps(string fileName, string password, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(StoreName storeName, string subject)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid, StoreLocation location)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid, StoreLocation location, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(X509Certificate2 serverCertificate)`
- `UseHttps(X509Certificate2 serverCertificate, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(Action<HttpsConnectionAdapterOptions> configureOptions)`

`ListenOptions.UseHttps` 参数：

- `filename` 是证书文件的路径和文件名，关联包含应用内容文件的目录。
- `password` 是访问 X.509 证书数据所需的密码。
- `configureOptions` 是配置 `HttpsConnectionAdapterOptions` 的 `Action`。返回 `ListenOptions`。
- `storeName` 是从中加载证书的证书存储。
- `subject` 是证书的主题名称。
- `allowInvalid` 指示是否存在需要留意的无效证书，例如自签名证书。
- `location` 是从中加载证书的存储位置。
- `serverCertificate` 是 X.509 证书。

在生产中，必须显式配置 HTTPS。至少必须提供默认证书。

下面要描述的支持的配置：

- 无配置
- 从配置中替换默认证书
- 更改代码中的默认值

无配置

Kestrel 在 `http://localhost:5000` 和 `https://localhost:5001` 上进行侦听（如果默认证书可用）。

使用以下内容指定 URL：

- `ASPNETCORE_URLS` 环境变量。
- `--urls` 命令行参数。
- `urls` 主机配置键。
- `UseUrls` 扩展方法。

有关详细信息，请参阅服务器 URL 和重写配置。

采用这些方法提供的值可以是一个或多个 HTTP 和 HTTPS 终结点（如果默认证书可用，则为 HTTPS）。将值配置为以分号分隔的列表（例如 `"Urls": "http://localhost:8000;http://localhost:8001"`）。

从配置中替换默认证书

WebHost.CreateDefaultBuilder 在默认情况下调用 `serverOptions.Configure(context.Configuration.GetSection("Kestrel"))` 来加载 Kestrel 配置。Kestrel 可以使用默认 HTTPS 应用设置配置架构。从磁盘上的文件或从证书存储中配置多个终结点，包括要使用的 URL 和证书。

在以下 appsettings.json 示例中：

- 将 AllowInvalid 设置为 `true`，从而允许使用无效证书（例如自签名证书）。
- 任何未指定证书的 HTTPS 终结点（下例中的 HttpsDefaultCert）会回退至在 Certificates > Default 下定义的证书或开发证书。

```
{
"Kestrel": {
  "EndPoints": {
    "Http": {
      "Url": "http://localhost:5000"
    },

    "HttpsInlineCertFile": {
      "Url": "https://localhost:5001",
      "Certificate": {
        "Path": "<path to .pfx file>",
        "Password": "<certificate password>"
      }
    },

    "HttpsInlineCertStore": {
      "Url": "https://localhost:5002",
      "Certificate": {
        "Subject": "<subject; required>",
        "Store": "<certificate store; defaults to My>",
        "Location": "<location; defaults to CurrentUser>",
        "AllowInvalid": "<true or false; defaults to false>"
      }
    },

    "HttpsDefaultCert": {
      "Url": "https://localhost:5003"
    },

    "Https": {
      "Url": "https://*:5004",
      "Certificate": {
      "Path": "<path to .pfx file>",
      "Password": "<certificate password>"
      }
    }
    },
    "Certificates": {
      "Default": {
        "Path": "<path to .pfx file>",
        "Password": "<certificate password>"
      }
    }
  }
}
```

此外还可以使用任何证书节点的 Path 和 Password，采用证书存储字段指定证书。例如，可将 Certificates > Default 证书指定为：

```
"Default": {
  "Subject": "<subject; required>",
  "Store": "<cert store; defaults to My>",
  "Location": "<location; defaults to CurrentUser>",
  "AllowInvalid": "<true or false; defaults to false>"
}
```

架构的注意事项：

- 终结点的名称不区分大小写。例如，`HTTPS` 和 `Https` 都是有效的。
- 每个终结点都要具备 `Url` 参数。此参数的格式和顶层 `Urls` 配置参数一样，只不过它只能有单个值。
- 这些终结点不会添加进顶层 `Urls` 配置中定义的终结点，而是替换它们。通过 `Listen` 在

代码中定义的终结点与在配置节中定义的终结点相累积。

- `Certificate` 部分是可选的。如果为指定 `Certificate` 部分，则使用在之前的方案中定义的默认值。如果没有可用的默认值，服务器会引发异常且无法启动。
- `Certificate` 支持 Path-Password 和 Subject-Store 证书。
- 只要不会导致端口冲突，就能以这种方式定义任何数量的终结点。
- `options.Configure(context.Configuration.GetSection("Kestrel"))` 通过 `.Endpoint(string name, options => { })` 方法返回 `KestrelConfigurationLoader`，可以用于补充已配置的终结点设置：

  ```
  options.Configure(context.Configuration.GetSection("Kestrel"))
      .Endpoint("HTTPS", opt =>
      {
          opt.HttpsOptions.SslProtocols = SslProtocols.Tls12;
      });
  ```

  也可以直接访问 `KestrelServerOptions.ConfigurationLoader` 在现有加载程序上保持迭代，例如由 WebHost.CreateDefaultBuilder 提供的加载程序。
- 每个终结点的配置节都可用于 `Endpoint` 方法中的选项，以便读取自定义设置。
- 通过另一节再次调用 `options.Configure(context.Configuration.GetSection("Kestrel"))` 可能加载多个配置。只使用最新配置，除非之前的实例上显式调用了 `Load`。元包不会调用 `Load`，所以可能会替换它的默认配置节。
- `KestrelConfigurationLoader` 从 `KestrelServerOptions` 将 API 的 `Listen` 簇反射为 `Endpoint` 重载，因此可在同样的位置配置代码和配置终结点。这些重载不使用名称，且只使用配置中的默认设置。

更改代码中的默认值

可以使用 `ConfigureEndpointDefaults` 和 `ConfigureHttpsDefaults` 更改 `ListenOptions` 和 `HttpsConnectionAdapterOptions` 的默认设置，包括重写之前的方案指定的默认证书。需要在配置任何终结点之前调用 `ConfigureEndpointDefaults` 和 `ConfigureHttpsDefaults`。

```
options.ConfigureEndpointDefaults(opt =>
{
    opt.NoDelay = true;
});

options.ConfigureHttpsDefaults(httpsOptions =>
{
    httpsOptions.SslProtocols = SslProtocols.Tls12;
});
```

SNI 的 Kestrel 支持

服务器名称指示 (SNI) 可用于承载相同 IP 地址和端口上的多个域。为了运行 SNI，客户端在 TLS 握手过程中将进行安全会话的主机名发送至服务器，从而让服务器可以提供正确的证书。在 TLS 握手后的安全会话期间，客户端将服务器提供的证书用于与服务器进行加密通信。

Kestrel 通过 `ServerCertificateSelector` 回调支持 SNI。每次连接调用一次回调，从而允许应用检查主机名并选择合适的证书。

SNI 支持要求：

- 在目标框架 `netcoreapp2.1` 上运行。在 `netcoreapp2.0` 和 `net461` 上，回调也会调用，但是 `name` 始终为 `null`。如果客户端未在 TLS 握手过程中提供主机名参数，则 `name` 也为 `null`。
- 所有网站在相同的 Kestrel 实例上运行。Kestrel 在无反向代理时不支持跨多个实例共享一个 IP 地址和端口。

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            options.ListenAnyIP(5005, listenOptions =>
            {
                listenOptions.UseHttps(httpsOptions =>
                {
                    var localhostCert = CertificateLoader.LoadFromStoreCert(
                        "localhost", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var exampleCert = CertificateLoader.LoadFromStoreCert(
                        "example.com", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var subExampleCert = CertificateLoader.LoadFromStoreCert(
                        "sub.example.com", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var certs = new Dictionary<string, X509Certificate2>(
                        StringComparer.OrdinalIgnoreCase);
                    certs["localhost"] = localhostCert;
                    certs["example.com"] = exampleCert;
                    certs["sub.example.com"] = subExampleCert;

                    httpsOptions.ServerCertificateSelector = (connectionContext, name) =>
                    {
                        if (name != null && certs.TryGetValue(name, out var cert))
                        {
                            return cert;
                        }

                        return exampleCert;
                    };
                });
            });
        });
```

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel((context, options) =>
        {
            options.ListenAnyIP(5005, listenOptions =>
            {
                listenOptions.UseHttps(httpsOptions =>
                {
                    var localhostCert = CertificateLoader.LoadFromStoreCert(
                        "localhost", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var exampleCert = CertificateLoader.LoadFromStoreCert(
                        "example.com", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var subExampleCert = CertificateLoader.LoadFromStoreCert(
                        "sub.example.com", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var certs = new Dictionary<string, X509Certificate2>(
                        StringComparer.OrdinalIgnoreCase);
                    certs["localhost"] = localhostCert;
                    certs["example.com"] = exampleCert;
                    certs["sub.example.com"] = subExampleCert;

                    httpsOptions.ServerCertificateSelector = (connectionContext, name) =>
                    {
                        if (name != null && certs.TryGetValue(name, out var cert))
                        {
                            return cert;
                        }

                        return exampleCert;
                    };
                });
            });
        })
        .Build();
```

绑定到 TCP 套接字

Listen 方法绑定至 TCP 套接字，并可通过选项 lambda 配置 SSL 证书：

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            options.Listen(IPAddress.Loopback, 5000);
            options.Listen(IPAddress.Loopback, 5001, listenOptions =>
            {
                listenOptions.UseHttps("testCert.pfx", "testPassword");
            });
        });
```

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
        {
            options.Listen(IPAddress.Loopback, 5000);
            options.Listen(IPAddress.Loopback, 5001, listenOptions =>
            {
                listenOptions.UseHttps("testCert.pfx", "testPassword");
            });
        });
```

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
        {
            options.Listen(IPAddress.Loopback, 5000);
            options.Listen(IPAddress.Loopback, 5001, listenOptions =>
            {
                listenOptions.UseHttps("testCert.pfx", "testPassword");
            });
        });
```

该示例为带有 ListenOptions.的终结点配置 SSL。可使用相同 API 为特定终结点配置其他 Kestrel 设置。

在 Windows 上，可以使用 New-SelfSignedCertificate PowerShell cmdlet 创建自签名证书。有关不支持的示例，请参阅 UpdateIISExpressSSLForChrome.ps1。

在 macOS、Linux 和 Windows 上，可以使用 OpenSSL 创建证书。

绑定到 Unix 套接字

可通过 ListenUnixSocket 侦听 Unix 套接字以提高 Nginx 的性能，如以下示例所示：

```
.ConfigureKestrel((context, options) =>
{
    options.ListenUnixSocket("/tmp/kestrel-test.sock");
    options.ListenUnixSocket("/tmp/kestrel-test.sock", listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testpassword");
    });
});
```

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(options =>
        {
            options.ListenUnixSocket("/tmp/kestrel-test.sock");
            options.ListenUnixSocket("/tmp/kestrel-test.sock", listenOptions =>
            {
                listenOptions.UseHttps("testCert.pfx", "testpassword");
            });
        });
```

端口 0

如果指定端口号 `0`，Kestrel 将动态绑定到可用端口。以下示例演示如何确定 Kestrel 在运行时实际绑定到的端口：

```
public void Configure(IApplicationBuilder app)
{
    var serverAddressesFeature =
        app.ServerFeatures.Get<IServerAddressesFeature>();

    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        context.Response.ContentType = "text/html";
        await context.Response
            .WriteAsync("<!DOCTYPE html><html lang=\"en\"><head>" +
                "<title></title></head><body><p>Hosted by Kestrel</p>");

        if (serverAddressesFeature != null)
        {
            await context.Response
                .WriteAsync("<p>Listening on the following addresses: " +
                    string.Join(", ", serverAddressesFeature.Addresses) +
                    "</p>");
        }

        await context.Response.WriteAsync("<p>Request URL: " +
            $"{context.Request.GetDisplayUrl()}<p>");
    });
}
```

在应用运行时，控制台窗口输出指示可用于访问应用的动态端口：

```
Listening on the following addresses: http://127.0.0.1:48508
```

限制

使用以下方法配置终结点：

- UseUrls
- `--urls` 命令行参数
- `urls` 主机配置键
- `ASPNETCORE_URLS` 环境变量

若要将代码用于 Kestrel 以外的服务器，这些方法非常有用。不过，请注意以下限制：

- SSL 不能使用这些方法，除非 HTTPS 终结点配置中提供了默认证书（如本主题前面的部分所示，使用 `KestrelServerOptions` 配置或配置文件）。

- 如果同时使用 `Listen` 和 `UseUrls` 方法，`Listen` 终结点将覆盖 `UseUrls` 终结点。

IIS 终结点配置

使用 IIS 时，由 `Listen` 或 `UseUrls` 设置用于 IIS 覆盖绑定的 URL 绑定。有关详细信息，请参阅 ASP.NET Core 模块主题。

ListenOptions.Protocols

`Protocols` 属性建立在连接终结点上或为服务器启用的 HTTP 协议（`HttpProtocols`）。从 `HttpProtocols` 枚举向 `Protocols` 属性赋值。

`HTTPPROTOCOLS` 枚举值	允许的连接协议
`Http1`	仅 HTTP/1.1。可以在具有 TLS 或没有 TLS 的情况下使用。
`Http2`	仅 HTTP/2。主要在具有 TLS 的情况下使用。仅当客户端支持先验知识模式时，才可以在没有 TLS 的情况下使用。
`Http1AndHttp2`	HTTP/1.1 和 HTTP/2。需要 TLS 和应用程序层协议协商 (ALPN) 连接来协商 HTTP/2；否则，连接默认为 HTTP/1.1。

默认协议是 HTTP/1.1。

HTTP/2 的 TLS 限制：

- TLS 版本 1.2 或更高版本
- 重新协商已禁用
- 压缩已禁用
- 最小的临时密钥交换大小：
  - 椭圆曲线 Diffie-Hellman (ECDHE) [RFC4492] –最小 224 位
  - 有限字段 Diffie-Hellman (DHE) [`TLS12`] – 最小 2048 位
- 密码套件未列入阻止列表

默认情况下，支持具有 P-256 椭圆曲线 [`FIPS186`] 的 `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` [`TLS-ECDHE`]。

以下示例允许端口 8000 上的 HTTP/1.1 和 HTTP/2 连接。TLS 使用提供的证书来保护连接：

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            options.Listen(IPAddress.Any, 8000, listenOptions =>
            {
                listenOptions.Protocols = HttpProtocols.Http1AndHttp2;
                listenOptions.UseHttps("testCert.pfx", "testPassword");
            });
        });
```

（可选）创建 `IConnectionAdapter` 实现，以针对特定密码的每个连接筛选 TLS 握手：

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            options.Listen(IPAddress.Any, 8000, listenOptions =>
            {
                listenOptions.Protocols = HttpProtocols.Http1AndHttp2;
                listenOptions.UseHttps("testCert.pfx", "testPassword");
                listenOptions.ConnectionAdapters.Add(new TlsFilterAdapter());
            });
        });
```

```
private class TlsFilterAdapter : IConnectionAdapter
{
    public bool IsHttps => false;

    public Task<IAdaptedConnection> OnConnectionAsync(ConnectionAdapterContext context)
    {
        var tlsFeature = context.Features.Get<ITlsHandshakeFeature>();

        // Throw NotSupportedException for any cipher algorithm that you don't
        // wish to support. Alternatively, define and compare
        // ITlsHandshakeFeature.CipherAlgorithm to a list of acceptable cipher
        // suites.
        //
        // A ITlsHandshakeFeature.CipherAlgorithm of CipherAlgorithmType.Null
        // indicates that no cipher algorithm supported by Kestrel matches the
        // requested algorithm(s).
        if (tlsFeature.CipherAlgorithm == CipherAlgorithmType.Null)
        {
            throw new NotSupportedException("Prohibited cipher: " +
tlsFeature.CipherAlgorithm);
        }

        return Task.FromResult<IAdaptedConnection>(new
AdaptedConnection(context.ConnectionStream));
    }

    private class AdaptedConnection : IAdaptedConnection
    {
        public AdaptedConnection(Stream adaptedStream)
        {
            ConnectionStream = adaptedStream;
        }

        public Stream ConnectionStream { get; }

        public void Dispose()
        {
        }
    }
}
```

从配置中设置协议

WebHost.CreateDefaultBuilder 在默认情况下调用 `serverOptions.Configure(context.Configuration.GetSection("Kestrel"))` 来加载 Kestrel 配置。

在以下 appsettings.json 示例中，为 Kestrel 的所有终结点建立默认的连接协议（HTTP/1.1 和 HTTP/2）：

```
{
  "Kestrel": {
    "EndPointDefaults": {
      "Protocols": "Http1AndHttp2"
    }
  }
}
```

以下配置文件示例为特定终结点建立了连接协议：

```
{
  "Kestrel": {
    "EndPoints": {
      "HttpsDefaultCert": {
        "Url": "https://localhost:5001",
        "Protocols": "Http1AndHttp2"
      }
    }
  }
}
```

代码中指定的协议覆盖了由配置设置的值。

传输配置

对于 ASP.NET Core 2.1 版，Kestrel 默认传输不再基于 Libuv，而是基于托管的套接字。这是 ASP.NET Core 2.0 应用升级到 2.1 时的一个重大更改，它调用 WebHostBuilderLibuvExtensions.UseLibuv 并依赖于以下包中的一个：

- Microsoft.AspNetCore.Server.Kestrel（直接包引用）
- Microsoft.AspNetCore.App

对于使用 Microsoft.AspNetCore.App 元包且需要使用 Libuv 的 ASP.NET Core 2.1 或更高版本的项目：

- 将用于 Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv 包的依赖项添加到应用的项目文件：

  ```
  <PackageReference Include="Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv"
                    Version="<LATEST_VERSION>" />
  ```

- 调用 WebHostBuilderLibuvExtensions.UseLibuv：

  ```
  public class Program
  {
      public static void Main(string[] args)
      {
          CreateWebHostBuilder(args).Build().Run();
      }

      public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
          WebHost.CreateDefaultBuilder(args)
              .UseLibuv()
              .UseStartup<Startup>();
  }
  ```

URL 前缀

如果使用 `UseUrls`、`--urls` 命令行参数、`urls` 主机配置键或 `ASPNETCORE_URLS` 环境变量，URL 前缀可采用以下任意格式。

仅 HTTP URL 前缀是有效的。使用 `UseUrls` 配置 URL 绑定时，Kestrel 不支持 SSL。

- 包含端口号的 IPv4 地址

  ```
  http://65.55.39.10:80/
  ```

  `0.0.0.0` 是一种绑定到所有 IPv4 地址的特殊情况。

- 包含端口号的 IPv6 地址

  ```
  http://[0:0:0:0:0:ffff:4137:270a]:80/
  ```

  `[::]` 是 IPv4 `0.0.0.0` 的 IPv6 等效项。

- 包含端口号的主机名

  ```
  http://contoso.com:80/
  http://*:80/
  ```

  主机名、`*` 和 `+` 并不特殊。没有识别为有效 IP 地址或 `localhost` 的任何内容都将绑定到所有 IPv4 和 IPv6 IP。若要将不同主机名绑定到相同端口上的不同 ASP.NET Core 应用，请使用 HTTP.sys 或 IIS、Nginx 或 Apache 等反向代理服务器。

  > **WARNING**
  > 采用反向代理配置进行托管需要主机筛选。

- 包含端口号的主机 `localhost` 名称或包含端口号的环回 IP

  ```
  http://localhost:5000/
  http://127.0.0.1:5000/
  http://[::1]:5000/
  ```

  指定 `localhost` 后，Kestrel 将尝试绑定到 IPv4 和 IPv6 环回接口。如果其他服务正在任一环回接口上使用请求的端口，则 Kestrel 将无法启动。如果任一环回接口出于任何其他原因（通常是因为 IPv6 不受支持）而不可用，则 Kestrel 将记录一个警告。

主机筛选

尽管 Kestrel 支持基于前缀的配置（例如 `http://example.com:5000`），但 Kestrel 在很大程度上会忽略主机名。主机 `localhost` 是一个特殊情况，用于绑定至环回地址。除了显式 IP 地址以外的所有主机都绑定至所有公共 IP 地址。不验证 `Host` 标头。

解决方法是，使用主机筛选中间件。主机筛选中间件由 Microsoft.AspNetCore.HostFiltering 包提供，此包包含在 Microsoft.AspNetCore.App 元包中（ASP.NET Core 2.1 或更高版本）。该中间件由 CreateDefaultBuilder 添加，可调用 AddHostFiltering：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

默认情况下，主机筛选中间件处于禁用状态。要启用该中间件，请在 appsettings.json/appsettings.<EnvironmentName>.json 中定义一个 `AllowedHosts` 键。此值是以分号分隔的不带端口号的主机名列表：

appsettings.json：

```
{
  "AllowedHosts": "example.com;localhost"
}
```

> **NOTE**
>
> 转接头中间件同样提供 ForwardedHeadersOptions.AllowedHosts 选项。转接头中间件和主机筛选中间件具有适合不同方案的相似功能。如果未保留 `Host` 标头，并且使用反向代理服务器或负载均衡器转接请求，则使用转接头中间件设置 `AllowedHosts` 比较合适。将 Kestrel 用作面向公众的边缘服务器或直接转接 `Host` 标头时，使用主机筛选中间件设置 `AllowedHosts` 比较合适。
>
> 有关转接头中间件的详细信息，请参阅 配置 ASP.NET Core 以使用代理服务器和负载均衡器。

其他资源

- 解决 ASP.NET Core 项目
- 强制实施 HTTPS 在 ASP.NET Core
- 配置 ASP.NET Core 以使用代理服务器和负载均衡器
- Kestrel 源代码
- RFC 7230：消息语法和路由（5.4 节：主机）

ASP.NET Core 中的 HTTP.sys Web 服务器实现

2019/1/10 • • Edit Online

作者：Tom Dykstra、Chris Ross 和 Luke Latham

HTTP.sys 是仅在 Windows 上运行的适用于 ASP.NET Core 的 Web 服务器。HTTP.sys 是 Kestrel 服务器的替代选择，提供了一些 Kestrel 不提供的功能。

> **IMPORTANT**
>
> HTTP.sys 与 ASP.NET Core 模块不兼容，不能与 IIS 或 IIS Express 结合使用。

HTTP.sys 支持以下功能：

- Windows 身份验证
- 端口共享
- 具有 SNI 的 HTTPS
- 基于 TLS 的 HTTP/2（Windows 10 或更高版本）
- 直接文件传输
- 响应缓存
- WebSocket（Windows 8 或更高版本）

受支持的 Windows 版本：

- Windows 7 或更高版本
- Windows Server 2008 R2 或更高版本

查看或下载示例代码（如何下载）

何时使用 HTTP.sys

HTTP.sys 对于以下情形的部署来说很有用：

- 需要将服务器直接公开到 Internet 而不使用 IIS 的部署。

- 内部部署需要 Kestrel 中没有的功能，如 Windows 身份验证。

HTTP.sys 是一项成熟的技术，可以抵御多种攻击，并提供可靠、安全、可伸缩的全功能 Web 服务器。IIS 本身作为 HTTP.sys 之上的 HTTP 侦听器运行。

HTTP/2 支持

如果满足以下基本要求，将为 ASP.NET Core 应用启用 HTTP/2：

- Windows Server 2016/Windows 10 或更高版本
- 应用程序层协议协商 (ALPN) 连接
- TLS 1.2 或更高版本的连接

如果已建立 HTTP/2 连接，HttpRequest.Protocol 会报告 `HTTP/2`。

如果已建立 HTTP/2 连接，HttpRequest.Protocol 会报告 `HTTP/1.1`。

默认情况下将启用 HTTP/2。如果未建立 HTTP/2 连接，连接会回退到 HTTP/1.1。在 Windows 的未来版本中，将提供 HTTP/2 配置标志，包括使用 HTTP.sys 禁用 HTTP/2 的功能。

对 Kerberos 进行内核模式身份验证

HTTP.sys 通过 Kerberos 身份验证协议委托给内核模式身份验证。Kerberos 和 HTTP.sys 不支持用户模式身份验证。必须使用计算机帐户来解密从 Active Directory 获取的并由客户端转发到服务器的 Kerberos 令牌/票证，以便对用户进行身份验证。注册主机的服务主体名称 (SPN)，而不是应用的用户。

如何使用 HTTP.sys

配置 ASP.NET Core 应用以使用 HTTP.sys

1. 使用 Microsoft.AspNetCore.App metapackage (nuget.org)（ASP.NET Core 2.1 或更高版本）时，不需要项目文件中的包引用。未使用 `Microsoft.AspNetCore.App` 元包时，向 Microsoft.AspNetCore.Server.HttpSys 添加包引用。

2. 构建 Web 主机时调用 UseHttpSys 扩展方法，同时指定所需的 HTTP.sys 选项：

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseHttpSys(options =>
        {
            // The following options are set to default values.
            options.Authentication.Schemes = AuthenticationSchemes.None;
            options.Authentication.AllowAnonymous = true;
            options.MaxConnections = null;
            options.MaxRequestBodySize = 30000000;
            options.UrlPrefixes.Add("http://localhost:5000");
        });
```

通过注册表设置处理其他 HTTP.sys 配置。

HTTP.sys 选项

PROPERTY	说明	默认
AllowSynchronousIO	控制是否允许 `HttpContext.Request.Body` 和 `HttpContext.Response.Body` 的同步输入/输出。	`true`
Authentication.AllowAnonymous	允许匿名请求。	`true`
Authentication.Schemes	指定允许的身份验证方案。可能在处理侦听器之前随时修改。通过 AuthenticationSchemes 枚举 `Basic`、`Kerberos`、`Negotiate`、`None` 和 `NTLM` 提供值。	`None`

PROPERTY	说明	默认
EnableResponseCaching	尝试内核模式缓存，响应合格的标头。该响应可能不包括 `Set-Cookie`、`Vary` 或 `Pragma` 标头。它必须包括属性为 `public` 的 `Cache-Control` 标头和 `shared-max-age` 或 `max-age` 值，或 `Expires` 标头。	`true`
MaxAccepts	最大并发接受数量。	5 × 环境。ProcessorCount
MaxConnections	要接受的最大并发连接数。使用 `-1` 实现无限。通过 `null` 使用注册表的计算机范围内的设置。	`null` （无限制）
MaxRequestBodySize	请参阅 MaxRequestBodySize 部分。	30000000 个字节 (~28.6 MB)
RequestQueueLimit	队列中允许的最大请求数。	1000
ThrowWriteExceptions	指示由于客户端断开连接而失败的响应主体写入应引发异常还是正常完成。	`false` （正常完成）
超时	公开 HTTP.sys TimeoutManager 配置，也可以在注册表中进行配置。请访问 API 链接详细了解每个设置，包括默认值： • TimeoutManager.DrainEntityBody – HTTP 服务器 API 对保持的连接消耗实体正文的时间上限。 • TimeoutManager.EntityBody – 请求实体正文到达的时间上限。 • TimeoutManager.HeaderWait – HTTP 服务器 API 分析请求头的时间上限。 • TimeoutManager.IdleConnection – 空闲连接存在的时间上限。 • TimeoutManager.MinSendBytesPerSecond – 响应的最小发送速率。 • TimeoutManager.RequestQueue – 请求在被应用选择前在请求队列中保留的时间上限。	
UrlPrefixes	指定 UrlPrefixCollection 以注册 HTTP.sys。最有用的是 UrlPrefixCollection.Add，它用于将前缀添加到集合中。可能在处理侦听器之前随时对这些设置进行修改。	

MaxRequestBodySize

允许的请求正文的最大大小(以字节计)。当设置为 `null` 时，最大请求正文大小不受限制。此限制不会影响升级后的连接，这始终不受限制。

在 ASP.NET Core MVC 应用中为单个 `IActionResult` 替代限制的推荐方法是在操作方法上使用 RequestSizeLimitAttribute 属性：

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

如果在应用开始读取请求后尝试配置请求限制，则会引发异常。`IsReadOnly` 属性可用于指示 `MaxRequestBodySize` 属性是否处于只读状态。只读状态意味着已经太迟了，无法配置限制。

如果应用应替代每个请求的 MaxRequestBodySize，则使用 IHttpMaxRequestBodySizeFeature：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    ILogger<Startup> logger, IServer server)
{
    app.Use(async (context, next) =>
    {
        context.Features.Get<IHttpMaxRequestBodySizeFeature>()
            .MaxRequestBodySize = 10 * 1024;

        var serverAddressesFeature =
            app.ServerFeatures.Get<IServerAddressesFeature>();
        var addresses = string.Join(", ", serverAddressesFeature?.Addresses);

        logger.LogInformation($"Addresses: {addresses}");

        await next.Invoke();
    });

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    // Enable HTTPS Redirection Middleware when hosting the app securely.
    //app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseMvc();
}
```

3. 如果使用的是 Visual Studio，请确保应用未经配置以运行 IIS 或 IIS Express。

   在 Visual Studio 中，默认启动配置文件是针对 IIS Express 的。若要作为控制台应用运行该项目，请手动更改所选配置文件，如以下屏幕截图中所示：

配置 Windows Server

1. 如果应用为框架相关部署，则安装 .NET Core、.NET Framework 或两者（如果应用是面向 .NET Framework 的 .NET Core 应用）。

   - **.NET Core**– 如果应用需要 .NET Core，请从 .NET 所有下载获取并运行 .NET Core 安装程序。
   - **.NET framework** – 如果应用要求 .NET Framework，请参阅 .NET Framework：安装指南，以查找安装说明。安装所需的 .NET Framework。最新 .NET Framework 的安装程序可从 .NET 所有下载中找到。

2. 配置应用的 URL 和端口。

   默认情况下，ASP.NET Core 绑定到 `http://localhost:5000`。若要配置 URL 前缀和端口，选项包括使用：

   - UseUrls
   - `urls` 命令行参数
   - `ASPNETCORE_URLS` 环境变量
   - UrlPrefixes

   下方的代码示例演示了如何使用 UrlPrefixes：

   ```
   public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
       WebHost.CreateDefaultBuilder(args)
           .UseStartup<Startup>()
           .UseHttpSys(options =>
           {
               // The following options are set to default values.
               options.Authentication.Schemes = AuthenticationSchemes.None;
               options.Authentication.AllowAnonymous = true;
               options.MaxConnections = null;
               options.MaxRequestBodySize = 30000000;
               options.UrlPrefixes.Add("http://localhost:5000");
           });
   ```

   `UrlPrefixes` 的一个优点是会为格式不正确的前缀立即生成一条错误消息。

   `UrlPrefixes` 中的设置替代 `UseUrls` / `urls` / `ASPNETCORE_URLS` 设置。因此，`UseUrls`、`urls` 和 `ASPNETCORE_URLS` 环境变量的一个优点是在 Kestrel 和 HTTP.sys 之间切换变得更加容易。有关 `UseUrls`、`urls` 和 `ASPNETCORE_URLS` 的详细信息，请参阅在 ASP.NET Core 中托管主题。

   HTTP.sys 使用 HTTP 服务器 API UrlPrefix 字符串格式。

   > **WARNING**
   > 不应使用顶级通配符绑定（`http://*:80/` 和 `http://+:80`）。顶级通配符绑定可能会为应用带来安全漏洞。此行为同时适用于强通配符和弱通配符。使用显式主机名而不是通配符。如果可控制整个父域（区别于易受攻击的 `*.com`），则子域通配符绑定（例如，`*.mysub.com`）不具有此安全风险。有关详细信息，请参阅 rfc7230 第 5.4 条。

3. 预先注册 URL 前缀以绑定到 HTTP.sys，并设置 x.509 证书。

   如果未在 Windows 中预先注册 URL 前缀，请使用管理员特权运行应用。唯一的例外是当使用端口号大于 1024 的 HTTP（而非 HTTPS）绑定到 localhost 时。在这种情况下，无需使用管理员特权。

   a. 用于配置 HTTP.sys 的内置工具为 *netsh.exe*。*netsh.exe* 用于保留 URL 前缀并分配 X.509 证书。此工具需要管理员特权。

      以下示例显示了保留端口 80 和 443 的 URL 前缀的命令：

```
netsh http add urlacl url=http://+:80/ user=Users
netsh http add urlacl url=https://+:443/ user=Users
```

以下示例显示了如何分配 X.509 证书：

```
netsh http add sslcert ipport=0.0.0.0:443 certhash=MyCertHash_Here appid="{00000000-
0000-0000-0000-000000000000}"
```

netsh.exe 的参考文档：

- Netsh Commands for Hypertext Transfer Protocol (HTTP)（超文本传输协议 (HTTP) 的 Netsh 命令）
- UrlPrefix Strings（UrlPrefix 字符串）

b. 如果需要，请创建自签名的 X.509 证书。

在 Windows 上，可以使用 New-SelfSignedCertificate PowerShell cmdlet 创建自签名证书。有关不支持的示例，请参阅 UpdateIISExpressSSLForChrome.ps1。

在 macOS、Linux 和 Windows 上，可以使用 OpenSSL 创建证书。

4. 打开防火墙端口以允许流量到达 HTTP.sys。使用 *netsh.exe* 或 PowerShell cmdlet。

代理服务器和负载均衡器方案

如果应用由 HTTP.sys 托管并且与来自 Internet 或公司网络的请求进行交互，当在代理服务器和负载均衡器后托管时，可能需要其他配置。有关详细信息，请参阅配置 ASP.NET Core 以使用代理服务器和负载均衡器。

其他资源

- HTTP 服务器 API
- aspnet/HttpSysServer GitHub 存储库（源代码）
- ASP.NET Core 中的 Web 主机和通用主机
- 解决 ASP.NET Core 项目

在 Windows 服务中托管 ASP.NET Core

2019/1/10 • • Edit Online

作者：Luke Latham、Tom Dykstra

不使用 IIS 时，可以在 Windows 上将 ASP.NET Core 应用作为 Windows 服务进行托管。作为 Windows 服务进行托管时，应用将在重新启动后自动启动。

查看或下载示例代码（如何下载）

部署类型

可以创建依赖框架的 Windows 服务部署或独立的 Windows 服务部署。有关部署方案的信息和建议，请参阅 .NET Core 应用程序部署。

依赖框架的部署

依赖框架的部署 (FDD) 依赖目标系统上存在共享系统级版本的 .NET Core。当 FDD 方案用于 ASP.NET Core Windows 服务应用时，SDK 会生成一个称为"依赖框架的可执行文件"的可执行文件 (*.exe)。

独立部署

独立部署 (SCD) 不依赖目标系统上存在的共享组件。运行时和应用的依赖项将与应用一起部署到托管系统。

将项目转换为 Windows 服务

对现有 ASP.NET Core 项目进行以下更改，以将应用作为服务运行：

项目文件更新

根据所选的部署类型，更新项目文件：

依赖框架的部署 (FDD)

将 Windows 运行时标识符 (RID) 添加到包含目标框架的 `<PropertyGroup>` 中。将 `<SelfContained>` 属性集添加到 `false`。通过将 `<IsTransformWebConfigDisabled>` 属性集添加到 `true` 来禁止创建 web.config文件。

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.2</TargetFramework>
  <RuntimeIdentifier>win7-x64</RuntimeIdentifier>
  <SelfContained>false</SelfContained>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
  <RuntimeIdentifier>win7-x64</RuntimeIdentifier>
  <UseAppHost>true</UseAppHost>
  <SelfContained>false</SelfContained>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

独立部署 (SCD)

确认是否存在 Windows 运行时标识符 (RID)，或将 RID 添加到包含目标框架的 `<PropertyGroup>` 中。通过将 `<IsTransformWebConfigDisabled>` 属性集添加到 `true` 来禁止创建 web.config文件。

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.2</TargetFramework>
  <RuntimeIdentifier>win7-x64</RuntimeIdentifier>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

要发布多个 RID：

- 通过以分号分隔的列表提供 RID。
- 使用属性名称 `<RuntimeIdentifiers>`（复数）。

  有关详细信息，请参阅 .NET Core RID 目录。

为 Microsoft.AspNetCore.Hosting.WindowsServices 添加包引用。

若要启用 Windows 事件日志记录，请添加 Microsoft.Extensions.Logging.EventLog 的包引用。

有关详细信息，请参阅处理启动和停止事件部分。

Program.Main 更新

在 `Program.Main` 中，进行下列更改：

- 在服务之外运行时，若要进行测试和调试，请添加代码以确定应用是否作为服务或控制台应用运行。检查是否已连接调试器或是否存在 `--console` 命令行参数。

  如果其中一个条件为 true（应用不作为服务运行），请在 Web 主机上调用 Run。

  如果条件为 false（应用作为服务运行）：

  - 调用 SetCurrentDirectory 并使用应用的发布位置路径。不要调用 GetCurrentDirectory 来获取路径，因为在调用 `GetCurrentDirectory` 时，Windows 服务应用将返回 C:\WINDOWS\system32 文件夹。有关详细信息，请参阅当前目录和内容根部分。
  - 调用 RunAsService 以将应用作为服务运行。

  由于命令行配置提供程序需要命令行参数的名称/值对，因此将先从参数中删除 `--console` 开关，然后 CreateDefaultBuilder 会接收这些参数。

- 若要写入 Windows 事件日志，请将事件日志提供程序添加到 ConfigureLogging。使用 appsettings.Production.json文件中的 `Logging:LogLevel:Default` 键设置日志记录级别。出于演示和测试目的，示例应用的生产设置文件会将日志记录级别设置为 `Information`。在生产中，通常会将值设置为 `Error`。有关更多信息，请参见ASP.NET Core 中的日志记录。

```
public class Program
{
    public static void Main(string[] args)
    {
        var isService = !(Debugger.IsAttached || args.Contains("--console"));

        if (isService)
        {
            var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
            var pathToContentRoot = Path.GetDirectoryName(pathToExe);
            Directory.SetCurrentDirectory(pathToContentRoot);
        }

        var builder = CreateWebHostBuilder(
            args.Where(arg => arg != "--console").ToArray());

        var host = builder.Build();

        if (isService)
        {
            // To run the app without the CustomWebHostService change the
            // next line to host.RunAsService();
            host.RunAsCustomService();
        }
        else
        {
            host.Run();
        }
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureLogging((hostingContext, logging) =>
            {
                logging.AddEventLog();
            })
            .ConfigureAppConfiguration((context, config) =>
            {
                // Configure the app here.
            })
            .UseStartup<Startup>();
}
```

发布应用

使用 dotnet publish、Visual Studio 发布配置文件 或 Visual Studio Code 发布应用。使用 Visual Studio 时，先选择“FolderProfile”并配置“目标位置”，再选择“发布”按钮。

若要使用命令行接口 (CLI) 工具发布示例应用，请在项目文件夹的命令提示符处运行 dotnet publish 命令，并将发布配置传递到 -c|--configuration 选项。使用带有路径的 -o|--output 选项发布到应用之外的文件夹。

发布依赖框架的部署 (FDD)

在以下示例中，将应用发布到 *c:\svc* 文件夹：

```
dotnet publish --configuration Release --output c:\svc
```

发布独立部署 (SCD)

必须在项目文件的 `<RuntimeIdenfifier>`（或 `<RuntimeIdentifiers>`）属性中指定 RID。将运行时提供到 `dotnet publish` 命令的 -r|--runtime 选项。

在以下示例中，为 `win7-x64` 运行时发布应用并发布到 c:\svc 文件夹：

```
dotnet publish --configuration Release --runtime win7-x64 --output c:\svc
```

创建用户帐户

运行 `net user` 命令，创建服务用户帐户：

```
net user {USER ACCOUNT} {PASSWORD} /add
```

对于示例应用，使用用户名 `ServiceUser` 和密码创建用户帐户。在下面的命令中，将 `{PASSWORD}` 替换为强密码。

```
net user ServiceUser {PASSWORD} /add
```

如果需要将用户添加到组中，请运行 `net localgroup` 命令（其中 `{GROUP}` 是组名称）：

```
net localgroup {GROUP} {USER ACCOUNT} /add
```

有关详细信息，请参阅服务用户帐户。

设置权限

运行 icacls 命令，授予对应用文件夹的写入/读取/执行权限：

```
icacls "{PATH}" /grant {USER ACCOUNT}:(OI)(CI){PERMISSION FLAGS} /t
```

- `{PATH}` – 应用文件夹路径。
- `{USER ACCOUNT}` – 用户帐户 (SID)。
- `(OI)` – 对象继承标志将权限传播给从属文件。
- `(CI)` – 容器继承标志将权限传播给从属文件夹。
- `{PERMISSION FLAGS}` – 设置应用的访问权限。
  - 写入（`W`）
  - 读取（`R`）
  - 执行（`X`）
  - 完全（`F`）
  - 修改（`M`）
- `/t` – 以递归方式应用于现有从属文件夹和文件。

对于发布到 c:\sv 文件夹的示例应用，以及拥有写入/读取/执行权限的 `ServiceUser` 帐户，请运行以下命令：

```
icacls "c:\svc" /grant ServiceUser:(OI)(CI)WRX /t
```

有关详细信息，请参阅 icacls。

管理服务

创建服务

使用 sc.exe 命令行工具创建服务。`binPath` 值是应用的可执行文件的路径，其中包括可执行文件的文件名。每个参数和值的等于号和引号字符之间必须有空格。

```
sc create {SERVICE NAME} binPath= "{PATH}" obj= "{DOMAIN}\{USER ACCOUNT}" password= "{PASSWORD}"
```

- `{SERVICE NAME}` – 要在服务控制管理器中分配给服务的名称。
- `{PATH}` – 可执行服务的路径。
- `{DOMAIN}` – 已加入域的计算机的域。如果计算机未加入域，则为本地计算机名称。
- `{USER ACCOUNT}` – 运行服务所使用的用户帐户。
- `{PASSWORD}` – 用户帐户密码。

> **WARNING**
> 请勿省略 `obj` 参数。`obj` 的默认值为 LocalSystem 帐户。使用 `LocalSystem` 帐户运行服务会带来重大安全风险。请始终使用拥有有限权限的用户帐户运行服务。

在下面的示例应用示例中：

- 服务名为 MyService。
- 已发布的服务驻留在 c:\svc 文件夹中。应用可执行文件名为 SampleApp.exe。用双引号 (") 将 `binPath` 值引起来。
- 服务是使用 `ServiceUser` 帐户运行。将 `{DOMAIN}` 替换为用户帐户的域或本地计算机名称。用双引号 (") 将 `obj` 值引起来。示例:如果托管系统是名为 `MairaPC` 的本地计算机，请将 `obj` 设置为 `"MairaPC\ServiceUser"`。
- 将 `{PASSWORD}` 替换为用户帐户密码。用双引号 (") 将 `password` 值引起来。

```
sc create MyService binPath= "c:\svc\sampleapp.exe" obj= "{DOMAIN}\ServiceUser" password= "{PASSWORD}"
```

> **IMPORTANT**
> 请确保参数的等于号和参数值之间有空格。

启动服务

使用 `sc start {SERVICE NAME}` 命令启动服务。

要启动示例应用服务，请使用以下命令：

```
sc start MyService
```

此命令需要几秒钟才能启动服务。

确定服务状态

要检查服务的状态，请使用 `sc query {SERVICE NAME}` 命令。状态报告为以下值之一：

- `START_PENDING`
- `RUNNING`
- `STOP_PENDING`
- `STOPPED`

使用以下命令检查示例应用服务的状态：

```
sc query MyService
```

浏览 Web 应用服务

当服务处于 `RUNNING` 状态并且服务是 Web 应用时，在应用所在路径中浏览应用（默认路径为 `http://localhost:5000`；在使用 HTTPS 重定向中间件时，它将重定向到 `https://localhost:5001`）。

对于示例应用服务，请在 `http://localhost:5000` 浏览应用。

停止服务

使用 `sc stop {SERVICE NAME}` 命令停止服务。

以下命令可停止示例应用服务：

```
sc stop MyService
```

删除服务

停止服务并经过短暂延迟后，使用 `sc delete {SERVICE NAME}` 命令卸载服务。

检查示例应用服务的状态：

```
sc query MyService
```

当示例应用服务处于 `STOPPED` 状态时，使用以下命令卸载示例应用服务：

```
sc delete MyService
```

处理启动和停止事件

若要处理 OnStarting、OnStarted 和 OnStopping 事件，请执行以下其他更改：

1. 使用 `OnStarting`、`OnStarted` 和 `OnStopping` 方法创建从 WebHostService 派生的类：

```
internal class CustomWebHostService : WebHostService
{
    private ILogger _logger;

    public CustomWebHostService(IWebHost host) : base(host)
    {
        _logger = host.Services
            .GetRequiredService<ILogger<CustomWebHostService>>();
    }

    protected override void OnStarting(string[] args)
    {
        _logger.LogInformation("OnStarting method called.");
        base.OnStarting(args);
    }

    protected override void OnStarted()
    {
        _logger.LogInformation("OnStarted method called.");
        base.OnStarted();
    }

    protected override void OnStopping()
    {
        _logger.LogInformation("OnStopping method called.");
        base.OnStopping();
    }
}
```

2. 创建可将 `CustomWebHostService` 传递给 Run 的 IWebHost 的扩展方法：

```
public static class WebHostServiceExtensions
{
    public static void RunAsCustomService(this IWebHost host)
    {
        var webHostService = new CustomWebHostService(host);
        ServiceBase.Run(webHostService);
    }
}
```

3. 在 `Program.Main` 中，调用 `RunAsCustomService` 扩展方法，而不是 RunAsService：

```
host.RunAsCustomService();
```

若要在 `Program.Main` 中查看 RunAsService 的位置，请参阅将项目转换为 Windows 服务部分中所示的代码示例。

代理服务器和负载均衡器方案

与来自 Internet 或公司网络的请求进行交互且在代理或负载均衡器后方的服务可能需要其他配置。有关更多信息，请参见配置 ASP.NET Core 以使用代理服务器和负载均衡器。

配置 HTTPS

若要为服务配置安全终结点，请执行以下操作：

1. 使用平台的证书获取和部署机制，为托管系统创建 X.509 证书。

2. 将 Kestrel 服务器 HTTPS 终结点配置指定为使用此证书。

不支持使用 ASP.NET Core HTTPS 开发证书保护服务终结点。

当前目录和内容根

通过为 Windows 服务调用 GetCurrentDirectory 返回的当前工作目录是 C:\WINDOWS\system32 文件夹。system32 文件夹不是存储服务文件（如设置文件）的合适位置。使用以下方法之一来维护和访问服务的资产和设置文件。

设置应用文件夹的内容根路径

ContentRootPath 是创建服务时提供给 `binPath` 参数的同一路径。请调用包含应用内容根的路径的 SetCurrentDirectory，而不是调用 `GetCurrentDirectory` 来创建设置文件的路径。

在 `Program.Main` 中，确定服务可执行文件的文件夹路径，并使用该路径来建立应用的内容根：

```
var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
var pathToContentRoot = Path.GetDirectoryName(pathToExe);
Directory.SetCurrentDirectory(pathToContentRoot);

CreateWebHostBuilder(args)
    .Build()
    .RunAsService();
```

将服务的文件存储在磁盘中的合适位置

使用 IConfigurationBuilder 时，使用 SetBasePath 指定到包含文件的文件夹的绝对路径。

其他资源

- Kestrel 终结点配置（包括 HTTPS 配置和 SNI 支持）
- ASP.NET Core Web 主机
- 解决 ASP.NET Core 项目

使用 Nginx 在 Linux 上托管 ASP.NET Core

2019/1/10 • • Edit Online

作者：Sourabh Shirhatti

本指南介绍如何在 Ubuntu 16.04 服务器上设置生产就绪 ASP.NET Core 环境。这些说明可能适用于较新版本的 Ubuntu，但尚未使用较新版本进行测试。

有关 ASP.NET Core 支持的其他 Linux 分配的信息，请参阅 Linux 上 .NET Core 的先决条件。

> **NOTE**
> 对于 Ubuntu 14.04，建议进行监控，以此作为监视 Kestrel 进程的解决方案。在 Ubuntu 14.04 上不提供 systemd。有关 Ubuntu 14.04 的说明，请参阅本主题的以前版本。

本指南：

- 将现有 ASP.NET Core 应用置于反向代理服务器后方。
- 设置反向代理服务器，以便将请求转发到 Kestrel Web 服务器。
- 确保 Web 应用在启动时作为守护程序运行。
- 配置进程管理工具以帮助重新启动 Web 应用。

系统必备

1. 使用具有 sudo 特权的标准用户帐户访问 Ubuntu 16.04 服务器。
2. 在服务器上安装 .NET Core 运行时。
   a. 访问 .NET Core"所有下载"页。
   b. 从"运行时"下的列表中选择最新的非预览运行时。
   c. 选择并执行适用于 Ubuntu 且匹配服务器的 Ubuntu 版本的说明。
3. 一个现有 ASP.NET Core 应用。

通过应用发布和复制

配置应用以进行依赖框架的部署。

在开发环境中运行 dotnet publish，将应用打包到可在服务器上运行的目录中（例如 bin/Release/<target_framework_moniker>/publish）：

```
dotnet publish --configuration Release
```

如果不希望维护服务器上的 .NET Core 运行时，还可将应用发布为独立部署。

使用集成到组织工作流的工具（例如 SCP、SFTP）将 ASP.NET Core 应用复制到服务器。通常可在 var 目录（例如 var/www/helloapp）下找到 Web 应用。

> **NOTE**
> 在生产部署方案中，持续集成工作流会执行发布应用并将资产复制到服务器的工作。

测试应用：

1. 在命令行中运行应用：`dotnet <app_assembly>.dll`。
2. 在浏览器中，导航到 `http://<serveraddress>:<port>` 以确认应用在 Linux 本地正常运行。

配置反向代理服务器

反向代理是为动态 Web 应用提供服务的常见设置。反向代理终止 HTTP 请求，并将其转发到 ASP.NET Core 应用。

使用反向代理服务器

Kestrel 非常适合从 ASP.NET Core 提供动态内容。但是，Web 服务功能不像服务器（如 IIS、Apache 或 Nginx）那样功能丰富。反向代理服务器可以从 HTTP 服务器卸载服务静态内容、缓存请求、压缩请求和 SSL 终端等工作。反向代理服务器可能驻留在专用计算机上，也可能与 HTTP 服务器一起部署。

鉴于此指南的目的，使用 Nginx 的单个实例。它与 HTTP 服务器一起运行在同一服务器上。根据要求，可以选择不同的设置。

由于请求是通过反向代理转接的，因此使用 Microsoft.AspNetCore.HttpOverrides 包中的转接头中间件。此中间件使用 `X-Forwarded-Proto` 标头来更新 `Request.Scheme`，使重定向 URI 和其他安全策略能够正常工作。

调用转接头中间件后，必须放置依赖于该架构的组件，例如身份验证、链接生成、重定向和地理位置。作为一般规则，转接头中间件应在诊断和错误处理中间件以外的其他中间件之前运行。此顺序可确保依赖于转接头信息的中间件可以使用标头值进行处理。

在调用 UseAuthentication 或类似的身份验证方案中间件之前，调用 `Startup.Configure` 中的 UseForwardedHeaders 方法。配置中间件以转接 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头：

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();
```

在调用 UseIdentity 和 UseFacebookAuthentication 或类似的身份验证方案中间件之前，调用 `Startup.Configure` 中的 UseForwardedHeaders 方法。配置中间件以转接 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头：

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});

app.UseIdentity();
app.UseFacebookAuthentication(new FacebookOptions()
{
    AppId = Configuration["Authentication:Facebook:AppId"],
    AppSecret = Configuration["Authentication:Facebook:AppSecret"]
});
```

如果没有为中间件指定 ForwardedHeadersOptions，则要转接的默认标头为 `None`。

默认情况下仅信任 localhost (127.0.0.1, [::1]) 上运行的代理。如果组织内的其他受信任代理或网络处理 Internet 与 Web 服务器之间的请求，请使用 ForwardedHeadersOptions 将其添加到 KnownProxies 或 KnownNetworks 的列表。以下示例会将 IP 地址为 10.0.0.100 的受信任代理服务器添加到 `Startup.ConfigureServices` 中的转接头中间件 `KnownProxies`：

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});
```

有关更多信息，请参见配置 ASP.NET Core 以使用代理服务器和负载均衡器。

安装 Nginx

使用 `apt-get` 安装 Nginx。安装程序将创建一个 systemd init 脚本，该脚本运行 Nginx，作为系统启动时的守护程序。按照以下网站上的 Ubuntu 安装说明操作：Nginx：官方 Debian/Ubuntu 包。

> **NOTE**
>
> 如果需要可选 Nginx 模块，则可能需要从源代码生成 Nginx。

因为是首次安装 Nginx，通过运行以下命令显式启动：

```
sudo service nginx start
```

确认浏览器显示 Nginx 的默认登陆页。可在 `http://<server_IP_address>/index.nginx-debian.html` 访问登陆页面。

配置 Nginx

若要将 Nginx 配置为反向代理以将请求转接到 ASP.NET Core 应用，请修改 /etc/nginx/sites-available/default。在文本编辑器中打开它，并将内容替换为以下内容：

```
server {
    listen        80;
    server_name   example.com *.example.com;
    location / {
        proxy_pass         http://localhost:5000;
        proxy_http_version 1.1;
        proxy_set_header   Upgrade $http_upgrade;
        proxy_set_header   Connection keep-alive;
        proxy_set_header   Host $host;
        proxy_cache_bypass $http_upgrade;
        proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto $scheme;
    }
}
```

当没有匹配的 `server_name` 时，Nginx 使用默认服务器。如果没有定义默认服务器，则配置文件中的第一台服务器是默认服务器。作为最佳做法，添加指定默认服务器，它会在配置文件中返回状态代码 444。默认的服务器配置示例是：

```
server {
    listen   80 default_server;
    # listen [::]:80 default_server deferred;
    return   444;
}
```

使用上述配置文件和默认服务器，Nginx 接受主机标头 `example.com` 或 `*.example.com` 端口 80 上的公共流量。与这些主机不匹配的请求不会转接到 Kestrel。Nginx 将匹配的请求转接到 `http://localhost:5000` 中的 Kestrel。有关详细信息，请参阅 nginx 如何处理请求。若要更改 Kestrel 的 IP/端口，请参阅 Kestrel：终结点配置。

> **WARNING**
> 未能指定正确的 server_name 指令会公开应用的安全漏洞。如果可控制整个父域（区别于易受攻击的 `*.com`），则子域通配符绑定（例如，`*.example.com`）不具有此安全风险。有关详细信息，请参阅 rfc7230 第 5.4 条。

完成配置 Nginx 后，运行 `sudo nginx -t` 来验证配置文件的语法。如果配置文件测试成功，可以通过运行 `sudo nginx -s reload` 强制 Nginx 选取更改。

要直接在服务器上运行应用：

1. 请导航到应用目录。
2. 运行应用：`dotnet <app_assembly.dll>`，其中 `app_assembly.dll` 是应用的程序集文件名。

如果应用在服务器上运行，但无法通过 Internet 响应，请检查服务器的防火墙，并确认端口 80 已打开。如果使用 Azure Ubuntu VM，请添加启用入站端口 80 流量的网络安全组 (NSG) 规则。不需要启用出站端口 80 规则，因为启用入站规则后会自动许可出站流量。

测试应用完成后，请在命令提示符处按 `Ctrl+C` 关闭应用。

监视应用

服务器设置为将对 `http://<serveraddress>:80` 发起的请求转接到在 `http://127.0.0.1:5000` 中的 Kestrel 上运行的 ASP.NET Core 应用。但是，未将 Nginx 设置为管理 Kestrel 进程。systemd 可用于创建服务文件以启动和监视基础 Web 应用。systemd 是一个 init 系统，可以提供用于启动、停止和管理进程的许多强大的功能。

创建服务文件

创建服务定义文件：

```
sudo nano /etc/systemd/system/kestrel-helloapp.service
```

以下是应用的一个示例服务文件：

```
[Unit]
Description=Example .NET Web API App running on Ubuntu

[Service]
WorkingDirectory=/var/www/helloapp
ExecStart=/usr/bin/dotnet /var/www/helloapp/helloapp.dll
Restart=always
# Restart service after 10 seconds if the dotnet service crashes:
RestartSec=10
KillSignal=SIGINT
SyslogIdentifier=dotnet-example
User=www-data
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=DOTNET_PRINT_TELEMETRY_MESSAGE=false

[Install]
WantedBy=multi-user.target
```

如果配置未使用用户 www-data，则必须先创建此处定义的用户，并为该用户提供适当的文件所有权。

使用 `TimeoutStopSec` 配置在收到初始中断信号后等待应用程序关闭的持续时间。如果应用程序在此时间段内未关闭，则将发出 SIGKILL 以终止该应用程序。提供作为无单位秒数的值（例如，`150`）、时间跨度值（例如，`2min 30s`）或 `infinity` 以禁用超时。`TimeoutStopSec` 默认为管理器配置文件（*systemd-system.conf*、*system.conf.d*、*systemd-user.conf*、*user.conf.d*）中 `DefaultTimeoutStopSec` 的值。大多数分发版的默

认超时时间为 90 秒。

```
# The default value is 90 seconds for most distributions.
TimeoutStopSec=90
```

Linux 具有区分大小写的文件系统。将 ASPNETCORE_ENVIRONMENT 设置为“生产”会导致搜索配置文件 appsettings.Production.json，而不是 appsettings.production.json。

必须转义某些值（例如，SQL 连接字符串）以供配置提供程序读取环境变量。使用以下命令生成适当的转义值以供在配置文件中使用：

```
systemd-escape "<value-to-escape>"
```

保存该文件并启用该服务。

```
sudo systemctl enable kestrel-helloapp.service
```

启用该服务，并确认它正在运行。

```
sudo systemctl start kestrel-helloapp.service
sudo systemctl status kestrel-helloapp.service

● kestrel-helloapp.service - Example .NET Web API App running on Ubuntu
    Loaded: loaded (/etc/systemd/system/kestrel-helloapp.service; enabled)
    Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
Main PID: 9021 (dotnet)
    CGroup: /system.slice/kestrel-helloapp.service
            └─9021 /usr/local/bin/dotnet /var/www/helloapp/helloapp.dll
```

在配置了反向代理并通过 systemd 管理 Kestrel 后，Web 应用现已完全配置，并能在本地计算机上的浏览器中从 `http://localhost` 进行访问。也可以从远程计算机进行访问，同时限制可能进行阻止的任何防火墙。检查响应标头，`Server` 标头显示由 Kestrel 所提供的 ASP.NET Core 应用。

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

查看日志

使用 Kestrel 的 Web 应用是通过 `systemd` 进行管理的，因此所有事件和进程都被记录到集中日志。但是，此日志包含由 `systemd` 管理的所有服务和进程的全部条目。若要查看特定于 `kestrel-helloapp.service` 的项，请使用以下命令：

```
sudo journalctl -fu kestrel-helloapp.service
```

有关进一步筛选，使用时间选项（如 `--since today`、`--until 1 hour ago`）或这些选项的组合可以减少返回的条目数。

```
sudo journalctl -fu kestrel-helloapp.service --since "2016-10-18" --until "2016-10-18 04:00"
```

数据保护

ASP.NET Core 数据保护堆栈由多个 ASP.NET Core 中间件（包括 cookie 中间件等身份验证中间件）和跨站点请求伪造 (CSRF) 保护使用。即使用户代码不调用数据保护 API，也应该配置数据保护，以创建持久的加密密钥存储。如果不配置数据保护，则密钥存储在内存中。重启应用时，密钥会被丢弃。

如果密钥环存储于内存中，则在应用重启时：

- 所有基于 cookie 的身份验证令牌都无效。
- 用户需要在下一次请求时再次登录。
- 无法再解密使用密钥环保护的任何数据。这可能包括 CSRF 令牌和 ASP.NET Core MVC TempData cookie。

若要配置数据保护以持久保存并加密密钥环，请参阅：

- 在 ASP.NET Core 中的密钥存储提供程序
- 在 ASP.NET Core 中存放的密钥加密

较长的请求标头字段

如果应用需要的请求标头字段超过代理服务器的默认设置允许的长度（具体取决于平台，通常为 4K 或 8K），则需调整以下指令。要应用的值依赖应用场景。有关详细信息，请参见服务器文档。

- proxy_buffer_size
- proxy_buffers
- proxy_busy_buffers_size
- large_client_header_buffers

> **WARNING**
> 除非必要，否则不要提高代理缓冲区的默认值。提高这些值将增加缓冲区溢出的风险和恶意用户的拒绝服务 (DoS) 攻击风险。

保护应用

启用 AppArmor

Linux 安全模块 (LSM) 是一个框架，它是自 Linux 2.6 后的 Linux kernel 的一部分。LSM 支持安全模块的不同实现。AppArmor 是实现强制访问控制系统的 LSM，它允许将程序限制在一组有限的资源内。确保已启用并成功配置 AppArmor。

配置防火墙

关闭所有未使用的外部端口。通过为配置防火墙提供命令行接口，不复杂的防火墙 (ufw) 为 `iptables` 提供了前端。

> **WARNING**
> 如果未正确配置，防火墙将阻止对整个系统的访问。在使用 SSH 进行连接时，未能指定正确的 SSH 端口最终会将你关在系统之外。默认端口为 22。有关详细信息，请参阅 ufw 简介和手册。

安装 `ufw`，并将其配置为允许所需任何端口上的流量。

```
sudo apt-get install ufw

sudo ufw allow 22/tcp
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp

sudo ufw enable
```

保护 Nginx

更改 Nginx 响应名称

编辑 src/http/ngx_http_header_filter_module.c：

```
static char ngx_http_server_string[] = "Server: Web Server" CRLF;
static char ngx_http_server_full_string[] = "Server: Web Server" CRLF;
```

配置选项

用其他必需模块配置服务器。请考虑使用 ModSecurity 等 Web 应用防火墙来加强对应用的保护。

配置 SSL

- 通过指定由受信任的证书颁发机构 (CA) 颁发的有效证书来配置服务器，以侦听端口 `443` 上的 HTTPS 流量。
- 通过采用以下"/etc/nginx/nginx.conf"文件中所示的某些做法来增强安全保护。示例包括选择更强的密码并将通过 HTTP 的所有流量重定向到 HTTPS。
- 添加 `HTTP Strict-Transport-Security` (HSTS) 标头可确保由客户端发起的所有后续请求都通过 HTTPS。
- 如果以后会禁用 SSL，则不要添加 HSTS 标头或选择相应的 `max-age`。

添加 /etc/nginx/proxy.conf 配置文件：

```
proxy_redirect    off;
proxy_set_header   Host    $host;
proxy_set_header  X-Real-IP   $remote_addr;
proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header    X-Forwarded-Proto $scheme;
client_max_body_size  10m;
client_body_buffer_size 128k;
proxy_connect_timeout  90;
proxy_send_timeout   90;
proxy_read_timeout   90;
proxy_buffers   32 4k;
```

编辑 /etc/nginx/nginx.conf 配置文件。示例包含一个配置文件中的 `http` 和 `server` 部分。

```
http {
    include    /etc/nginx/proxy.conf;
    limit_req_zone $binary_remote_addr zone=one:10m rate=5r/s;
    server_tokens off;

    sendfile on;
    keepalive_timeout 29; # Adjust to the lowest possible value that makes sense for your use case.
    client_body_timeout 10; client_header_timeout 10; send_timeout 10;

    upstream hellomvc{
        server localhost:5000;
    }

    server {
        listen *:80;
        add_header Strict-Transport-Security max-age=15768000;
        return 301 https://$host$request_uri;
    }

    server {
        listen *:443    ssl;
        server_name     example.com;
        ssl_certificate /etc/ssl/certs/testCert.crt;
        ssl_certificate_key /etc/ssl/certs/testCert.key;
        ssl_protocols TLSv1.1 TLSv1.2;
        ssl_prefer_server_ciphers on;
        ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
        ssl_ecdh_curve secp384r1;
        ssl_session_cache shared:SSL:10m;
        ssl_session_tickets off;
        ssl_stapling on; #ensure your cert is capable
        ssl_stapling_verify on; #ensure your cert is capable

        add_header Strict-Transport-Security "max-age=63072000; includeSubdomains; preload";
        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;

        #Redirects all traffic
        location / {
            proxy_pass  http://hellomvc;
            limit_req   zone=one burst=10 nodelay;
        }
    }
}
```

保护 Nginx 免受点击劫持的侵害

点击劫持(也称为 *UI 伪装攻击*)是一种恶意攻击，其中网站访问者会上当受骗，从而导致在与当前要访问的页面不同的页面上单击链接或按钮。使用 `X-FRAME-OPTIONS` 可保护网站。

缓解点击劫持攻击：

1. 编辑 nginx.conf 文件：

   ```
   sudo nano /etc/nginx/nginx.conf
   ```

   添加行 `add_header X-Frame-Options "SAMEORIGIN";`。

2. 保存该文件。

3. 重启 Nginx。

MIME 类型探查

此标头可阻止大部分浏览器通过 MIME 方式探查来自已声明内容类型的响应，因为标头会指示浏览器不要替代

响应内容类型。使用 `nosniff` 选项后，如果服务器认为内容是“文本/html”，则浏览器将其显示为“文本/html”。

编辑 nginx.conf 文件：

```
sudo nano /etc/nginx/nginx.conf
```

添加行 `add_header X-Content-Type-Options "nosniff";` 并保存文件，然后重新启动 Nginx。

其他资源

- Linux 上 .NET Core 的先决条件
- Nginx：二进制版本：官方 Debian/Ubuntu 包
- 解决 ASP.NET Core 项目
- 配置 ASP.NET Core 以使用代理服务器和负载均衡器
- NGINX：使用转接头

使用 Apache 在 Linux 上托管 ASP.NET Core

2019/1/10 • • Edit Online

作者：Shayne Boyer

使用本指南，了解如何在 CentOS 7 上将 Apache 设置为反向代理服务器，以将 HTTP 流量重定向到在 Kestrel 服务器上运行的 ASP.NET Core Web 应用。mod_proxy extension 和相关模块可创建服务器的反向代理。

系统必备

- 运行 CentOS 7 的服务器，使用具有 sudo 特权的标准用户帐户。
- 在服务器上安装 .NET Core 运行时。
  1. 访问 .NET Core"所有下载"页。
  2. 从"运行时"下的列表中选择最新的非预览运行时。
  3. 选择并执行适用于 CentOS/Oracle 的说明。
- 一个现有 ASP.NET Core 应用。

通过应用发布和复制

配置应用以进行依赖框架的部署。

在开发环境中运行 dotnet publish，将应用打包到可在服务器上运行的目录中（例如 bin/Release/<target_framework_moniker>/publish）：

```
dotnet publish --configuration Release
```

如果不希望维护服务器上的 .NET Core 运行时，还可将应用发布为独立部署。

使用集成到组织工作流的工具（例如 SCP、SFTP）将 ASP.NET Core 应用复制到服务器。通常可在 var 目录（例如 var/www/helloapp）下找到 Web 应用。

> **NOTE**
> 在生产部署方案中，持续集成工作流会执行发布应用并将资产复制到服务器的工作。

配置代理服务器

反向代理是为动态 Web 应用提供服务的常见设置。反向代理终止 HTTP 请求，并将其转发到 ASP.NET 应用。

代理服务器将客户端请求转发到另一个服务器，而不是自身实现这些请求。反向代理转发到固定的目标，通常代表任意客户端。在本指南中，Apache 被配置为反向代理，在 Kestrel 为 ASP.NET Core 应用提供服务的同一服务器上运行。

由于请求是通过反向代理转接的，因此使用 Microsoft.AspNetCore.HttpOverrides 包中的转接头中间件。此中间件使用 `X-Forwarded-Proto` 标头来更新 `Request.Scheme`，使重定向 URI 和其他安全策略能够正常工作。

调用转接头中间件后，必须放置依赖于该架构的组件，例如身份验证、链接生成、重定向和地理位置。作为一般规则，转接头中间件应在诊断和错误处理中间件以外的其他中间件之前运行。此顺序可确保依赖于转接头信息的中间件可以使用标头值进行处理。

在调用 UseAuthentication 或类似的身份验证方案中间件之前，调用 `Startup.Configure` 中的 UseForwardedHeaders 方法。配置中间件以转接 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头：

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();
```

在调用 UseIdentity 和 UseFacebookAuthentication 或类似的身份验证方案中间件之前，调用 `Startup.Configure` 中的 UseForwardedHeaders 方法。配置中间件以转接 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头：

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});

app.UseIdentity();
app.UseFacebookAuthentication(new FacebookOptions()
{
    AppId = Configuration["Authentication:Facebook:AppId"],
    AppSecret = Configuration["Authentication:Facebook:AppSecret"]
});
```

如果没有为中间件指定 ForwardedHeadersOptions，则要转接的默认标头为 `None`。

默认情况下仅信任 localhost (127.0.0.1, [::1]) 上运行的代理。如果组织内的其他受信任代理或网络处理 Internet 与 Web 服务器之间的请求，请使用 ForwardedHeadersOptions 将其添加到 KnownProxies 或 KnownNetworks 的列表。以下示例会将 IP 地址为 10.0.0.100 的受信任代理服务器添加到 `Startup.ConfigureServices` 中的转接头中间件 `KnownProxies`：

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});
```

有关更多信息，请参见配置 ASP.NET Core 以使用代理服务器和负载均衡器。

安装 Apache

将 CentOS 包更新为其最新稳定版本：

```
sudo yum update -y
```

使用单个 `yum` 命令在 CentOS 上安装 Apache Web 服务器：

```
sudo yum -y install httpd mod_ssl
```

运行该命令后的示例输出：

```
Downloading packages:
httpd-2.4.6-40.el7.centos.4.x86_64.rpm                | 2.7 MB  00:00:01
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
Installing : httpd-2.4.6-40.el7.centos.4.x86_64      1/1
Verifying  : httpd-2.4.6-40.el7.centos.4.x86_64      1/1

Installed:
httpd.x86_64 0:2.4.6-40.el7.centos.4

Complete!
```

> **NOTE**
>
> 在此示例中，输出反映了 httpd.86_64，因为 CentOS 7 版本是 64 位的。若要验证 Apache 的安装位置，请从命令提示符运行 `whereis httpd`。

配置 Apache

Apache 的配置文件位于 `/etc/httpd/conf.d/` 目录内。除了 `/etc/httpd/conf.modules.d/` 中的模块配置文件外（其中包含加载模块所需的任何配置文件），将对任何带 .conf 扩展名的文件按字母顺序进行处理。

为应用创建名为 helloapp.conf 的配置文件：

```
<VirtualHost *:*>
    RequestHeader set "X-Forwarded-Proto" expr=%{REQUEST_SCHEME}
</VirtualHost>

<VirtualHost *:80>
    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5000/
    ServerName www.example.com
    ServerAlias *.example.com
    ErrorLog ${APACHE_LOG_DIR}helloapp-error.log
    CustomLog ${APACHE_LOG_DIR}helloapp-access.log common
</VirtualHost>
```

`VirtualHost` 块可以在服务器上的一个或多个文件中多次出现。在前面的配置文件中，Apache 接受端口 80 上的公共流量。正在向域 `www.example.com` 提供服务，`*.example.com` 别名解析为同一网站。有关详细信息，请参阅基于名称的虚拟主机支持。请求会通过代理从根位置转到 127.0.0.1 处的服务器的端口 5000。对于双向通信，需要 `ProxyPass` 和 `ProxyPassReverse`。若要更改 Kestrel 的 IP/端口，请参阅 Kestrel：终结点配置。

> **WARNING**
>
> 未能指定 VirtualHost 块中的正确 ServerName 指令会公开应用的安全漏洞。如果可控制整个父域（区别于易受攻击的 `*.com`），则子域通配符绑定（例如，`*.example.com`）不具有此安全风险。有关详细信息，请参阅 rfc7230 第 5.4 条。

可以使用 `ErrorLog` 和 `CustomLog` 指令配置每个 `VirtualHost` 的日志记录。`ErrorLog` 是服务器记录错误的位置，`CustomLog` 则设置文件名和日志文件的格式。在这种情况下，这是记录请求信息的位置。每个请求将各占一行。

保存文件，并测试配置。如果一切正常，响应应为 `Syntax [OK]`。

```
sudo service httpd configtest
```

重新启动 Apache：

```
sudo systemctl restart httpd
sudo systemctl enable httpd
```

监视应用

Apache 现在已设置为将对 `http://localhost:80` 发起的请求转发到运行在 `http://127.0.0.1:5000` 处的 Kestrel 上的 ASP.NET Core 应用。但是，未将 Apache 设置为管理 Kestrel 进程。使用 systemd，并创建服务文件以启动和监视基础 Web 应用。systemd 是一个 init 系统，可以提供用于启动、停止和管理进程的许多强大的功能。

创建服务文件

创建服务定义文件：

```
sudo nano /etc/systemd/system/kestrel-helloapp.service
```

应用的一个示例服务文件：

```
[Unit]
Description=Example .NET Web API App running on CentOS 7

[Service]
WorkingDirectory=/var/www/helloapp
ExecStart=/usr/local/bin/dotnet /var/www/helloapp/helloapp.dll
Restart=always
# Restart service after 10 seconds if the dotnet service crashes:
RestartSec=10
KillSignal=SIGINT
SyslogIdentifier=dotnet-example
User=apache
Environment=ASPNETCORE_ENVIRONMENT=Production

[Install]
WantedBy=multi-user.target
```

如果配置未使用用户 *apache*，则必须先创建用户，并为该用户提供适当的文件所有权。

使用 `TimeoutStopSec` 配置在收到初始中断信号后等待应用程序关闭的持续时间。如果应用程序在此时间段内未关闭，则将发出 SIGKILL 以终止该应用程序。提供作为无单位秒数的值（例如，`150`）、时间跨度值（例如，`2min 30s`）或 `infinity` 以禁用超时。`TimeoutStopSec` 默认为管理器配置文件（*systemd-system.conf*、*system.conf.d*、*systemd-user.conf*、*user.conf.d*）中 `DefaultTimeoutStopSec` 的值。大多数分发版的默认超时时间为 90 秒。

```
# The default value is 90 seconds for most distributions.
TimeoutStopSec=90
```

必须转义某些值（例如，SQL 连接字符串）以供配置提供程序读取环境变量。使用以下命令生成适当的转义值以供在配置文件中使用：

```
systemd-escape "<value-to-escape>"
```

保存该文件并启用该服务：

```
sudo systemctl enable kestrel-helloapp.service
```

启动该服务，并确认它正在运行：

```
sudo systemctl start kestrel-helloapp.service
sudo systemctl status kestrel-helloapp.service

● kestrel-helloapp.service - Example .NET Web API App running on CentOS 7
    Loaded: loaded (/etc/systemd/system/kestrel-helloapp.service; enabled)
    Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
Main PID: 9021 (dotnet)
    CGroup: /system.slice/kestrel-helloapp.service
            └─9021 /usr/local/bin/dotnet /var/www/helloapp/helloapp.dll
```

在配置了反向代理并通过 systemd 管理 Kestrel 后，Web 应用现已完全配置，并能在本地计算机上的浏览器中从 `http://localhost` 进行访问。检查响应标头，服务器标头表示 ASP.NET Core 应用由 Kestrel 提供服务：

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

查看日志

由于使用 Kestrel 的 Web 应用是通过 systemd 进行管理的，因此事件和进程将记录到集中日志。但是，此日志包含由 systemd 管理的所有服务和进程的条目。若要查看特定于 `kestrel-helloapp.service` 的项，请使用以下命令：

```
sudo journalctl -fu kestrel-helloapp.service
```

若要进行时间筛选，请使用命令指定时间选项。例如，使用 `--since today` 筛选出当天或 `--until 1 hour ago` 来查看前一小时的条目。有关详细信息，请参阅 journalctl 手册页。

```
sudo journalctl -fu kestrel-helloapp.service --since "2016-10-18" --until "2016-10-18 04:00"
```

数据保护

ASP.NET Core 数据保护堆栈由多个 ASP.NET Core 中间件（包括 cookie 中间件等身份验证中间件）和跨站点请求伪造 (CSRF) 保护使用。即使用户代码不调用数据保护 API，也应该配置数据保护，以创建持久的加密密钥存储。如果不配置数据保护，则密钥存储在内存中。重启应用时，密钥会被丢弃。

如果密钥环存储于内存中，则在应用重启时：

- 所有基于 cookie 的身份验证令牌都无效。
- 用户需要在下一次请求时再次登录。
- 无法再解密使用密钥环保护的任何数据。这可能包括 CSRF 令牌和 ASP.NET Core MVC TempData cookie。

若要配置数据保护以持久保存并加密密钥环，请参阅：

- 在 ASP.NET Core 中的密钥存储提供程序
- 在 ASP.NET Core 中存放的密钥加密

保护应用

配置防火墙

Firewalld 是管理防火墙的动态守护程序，支持网络区域。仍可以使用 iptable 管理端口和数据包筛选。默认情况下应安装 *Firewalld*。`yum` 可用于安装包或验证是否已安装。

```
sudo yum install firewalld -y
```

使用 `firewalld` 仅打开应用所需的端口。在此示例中，使用的是端口 80 和 443。以下命令将端口 80 和 443 永久设置为打开：

```
sudo firewall-cmd --add-port=80/tcp --permanent
sudo firewall-cmd --add-port=443/tcp --permanent
```

重新加载防火墙设置。检查默认区域中可用的服务和端口。通过检查 `firewall-cmd -h` 获取可用选项。

```
sudo firewall-cmd --reload
sudo firewall-cmd --list-all
```

```
public (default, active)
interfaces: eth0
sources:
services: dhcpv6-client
ports: 443/tcp 80/tcp
masquerade: no
forward-ports:
icmp-blocks:
rich rules:
```

SSL 配置

若要配置 Apache 用于 SSL，需使用 mod_ssl 模块。安装了 httpd 模块时，也会安装了 mod_ssl 模块。如果未安装，请使用 `yum` 将其添加到配置。

```
sudo yum install mod_ssl
```

若要强制使用 SSL，请安装 `mod_rewrite` 模块以启用 URL 重写：

```
sudo yum install mod_rewrite
```

修改 helloapp.conf 文件以启用 URL 重写和端口 443 上的安全通信：

```
<VirtualHost *:*>
    RequestHeader set "X-Forwarded-Proto" expr=%{REQUEST_SCHEME}
</VirtualHost>

<VirtualHost *:80>
    RewriteEngine On
    RewriteCond %{HTTPS} !=on
    RewriteRule ^/?(.*) https://%{SERVER_NAME}/$1 [R,L]
</VirtualHost>

<VirtualHost *:443>
    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5000/
    ErrorLog /var/log/httpd/helloapp-error.log
    CustomLog /var/log/httpd/helloapp-access.log common
    SSLEngine on
    SSLProtocol all -SSLv2
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:!RC4+RSA:+HIGH:+MEDIUM:!LOW:!RC4
    SSLCertificateFile /etc/pki/tls/certs/localhost.crt
    SSLCertificateKeyFile /etc/pki/tls/private/localhost.key
</VirtualHost>
```

NOTE

此示例中使用了本地生成的证书。SSLCertificateFile 应为域名的主证书文件。SSLCertificateKeyFile 应为创建 CSR 时生成的密钥文件。SSLCertificateChainFile 应为证书颁发机构提供的中间证书文件（如有）。

保存文件，并测试配置：

```
sudo service httpd configtest
```

重新启动 Apache：

```
sudo systemctl restart httpd
```

其他 Apache 建议

其他标头

为了防止恶意攻击，应对一些标头进行修改或添加一些标头。确保已安装 `mod_headers` 模块：

```
sudo yum install mod_headers
```

保护 Apache 免受点击劫持攻击

点击劫持（也称为 *UI 伪装攻击*）是一种恶意攻击，其中网站访问者会上当受骗，从而导致在与当前要访问的页面不同的页面上单击链接或按钮。使用 `X-FRAME-OPTIONS` 可保护网站。

缓解点击劫持攻击：

1. 编辑 *httpd.conf* 文件：

   ```
   sudo nano /etc/httpd/conf/httpd.conf
   ```

   添加行 `Header append X-FRAME-OPTIONS "SAMEORIGIN"`。

2. 保存该文件。

3. 重启 Apache。

MIME 类型探查

`X-Content-Type-Options` 标头阻止 Internet Explorer 进行 *MIME 探查*(从文件内容中确定文件的 `Content-Type`)。如果服务器通过设置 `nosniff` 选项将 `Content-Type` 标头设置为 `text/html`，则不管文件内容为何，Internet Explorer 都会将内容呈现为 `text/html`。

编辑 *httpd.conf* 文件：

```
sudo nano /etc/httpd/conf/httpd.conf
```

添加行 `Header set X-Content-Type-Options "nosniff"`。保存该文件。重启 Apache。

负载平衡

此示例演示如何在同一实例计算机上的 CentOS 7 和 Kestrel 上设置和配置 Apache。为了不出现单一故障点；使用 mod_proxy_balancer 并修改 VirtualHost 可实现在 Apache 代理服务器后方管理 Web 应用的多个实例。

```
sudo yum install mod_proxy_balancer
```

在下面所示的配置文件中，`helloapp` 的其他实例设置为在端口 5001 上运行。“代理”部分设置了具有两个成员的均衡器配置，以便对 byrequests 进行负载均衡。

```
<VirtualHost *:*>
    RequestHeader set "X-Forwarded-Proto" expr=%{REQUEST_SCHEME}
</VirtualHost>

<VirtualHost *:80>
    RewriteEngine On
    RewriteCond %{HTTPS} !=on
    RewriteRule ^/?(.*) https://%{SERVER_NAME}/$1 [R,L]
</VirtualHost>

<VirtualHost *:443>
    ProxyPass / balancer://mycluster/

    ProxyPassReverse / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5001/

    <Proxy balancer://mycluster>
        BalancerMember http://127.0.0.1:5000
        BalancerMember http://127.0.0.1:5001
        ProxySet lbmethod=byrequests
    </Proxy>

    <Location />
        SetHandler balancer
    </Location>
    ErrorLog /var/log/httpd/helloapp-error.log
    CustomLog /var/log/httpd/helloapp-access.log common
    SSLEngine on
    SSLProtocol all -SSLv2
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:!RC4+RSA:+HIGH:+MEDIUM:!LOW:!RC4
    SSLCertificateFile /etc/pki/tls/certs/localhost.crt
    SSLCertificateKeyFile /etc/pki/tls/private/localhost.key
</VirtualHost>
```

速率限制

使用 httpd 模块中包含的 mod_ratelimit，客户端的带宽可以限制为：

```
sudo nano /etc/httpd/conf.d/ratelimit.conf
```

示例文件将根位置下的带宽限制为 600 KB/秒：

```
<IfModule mod_ratelimit.c>
    <Location />
        SetOutputFilter RATE_LIMIT
        SetEnv rate-limit 600
    </Location>
</IfModule>
```

较长的请求标头字段

如果应用需要的请求标头字段超过代理服务器的默认设置允许的长度（通常为 8,190 字节），请调整 LimitRequestFieldSize 指令的值。要应用的值依赖应用场景。有关详细信息，请参见服务器文档。

> **WARNING**
> 除非必要，否则不要提高 `LimitRequestFieldSize` 的默认值。提高该值将增加缓冲区溢出的风险和恶意用户的拒绝服务 (DoS) 攻击风险。

其他资源

- Linux 上 .NET Core 的先决条件
- 解决 ASP.NET Core 项目
- 配置 ASP.NET Core 以使用代理服务器和负载均衡器

在 Docker 容器中托管 ASP.NET Core

2018/6/21 • • Edit Online

下面的文章可用于了解如何在 Docker 中托管 ASP.NET Core 应用：

容器和 Docker 简介

容器化是软件开发的一种方法，通过该方法可将应用程序或服务、其依赖项及其配置一起打包为容器映像。了解相关内容。可对该映像进行测试，然后将其部署到主机。

什么是 Docker

了解如何将 Docker 作为一种开源项目，用于将应用自动部署为可在云或本地运行的便携式独立容器。

Docker 术语

了解 Docker 技术的术语和定义。

Docker 容器、映像和注册表

了解如何将 Docker 容器映像存储在映像注册表中，以实现跨环境的一致部署。

为 .NET Core 应用程序生成 Docker 映像

了解如何生成和 Docker 化 ASP.NET Core 应用。了解由 Microsoft 维护的 Docker 映像并检查用例。

Visual Studio Tools for Docker

Visual Studio 2017 支持在用于 Windows 的 Docker 上生成、调试和运行面向 .NET Framework 或 .NET Core 的 ASP.NET Core 应用。了解相关内容。Windows 和 Linux 容器均受支持。

发布到 Docker 映像

了解如何通过 Visual Studio Tools for Docker 扩展使用 PowerShell 将 ASP.NET Core 应用部署到 Azure 上的 Docker 主机。

配置 ASP.NET Core 以使用代理服务器和负载均衡器

对于托管在代理服务器和负载均衡器后方的应用，可能需要附加配置。通过代理传递的请求通常会遮盖初始请求相关信息，例如方案和客户端 IP。可能必须将请求相关的一些信息手动转发给应用。

使用 ASP.NET Core 的 Visual Studio Tools for Docker

2018/10/30 • • Edit Online

Visual Studio 2017 支持生成、调试和运行面向 .NET Core 的容器化 ASP.NET Core 应用。Windows 和 Linux 容器均受支持。

查看或下载示例代码(如何下载)

系统必备

- Docker for Windows
- 具有".NET Core 跨平台开发"工作负载的 Visual Studio 2017

安装和设置

要安装 Docker，请先查看用于 Windows 的 Docker：安装须知了解相关信息。然后安装用于 Windows 的 Docker。

Docker for Windows 中的共享驱动器必须配置为支持卷映射和调试。右键单击系统托盘中的 Docker 图标，单击"设置"，然后选择"共享驱动器"。选择 Docker 存储文件的驱动器。单击"应用"。

> **TIP**
>
> 未配置共享驱动器时，Visual Studio 2017 15.6 及更高版本会发出提示。

向 Docker 容器添加项目

若要容器化 ASP.NET Core 项目，项目必须面向 .NET Core。同时支持 Linux 和 Windows 容器。

向项目添加 Docker 支持后，可选择 Windows 或 Linux 容器。Docker 主机必须运行类型相同的容器。要更改正在运行的 Docker 实例中的容器类型，请右键单击系统托盘中的 Docker 图标，再选择"切换到 Windows 容器..."或"切

换到 Linux 容器...”。

新应用

使用“ASP.NET Core Web 应用”项目模板新建应用时，请选中“启用 Docker 支持”复选框：

如果目标框架是 .NET Core，可通过 OS 下拉列表选择容器类型。

现有应用

对于面向 .NET Core 的 ASP.NET Core 项目，可通过两个选项使用工具添加 Docker 支持。在 Visual Studio 中打开项目，然后选择以下选项之一：

- 从“项目”菜单中选择“Docker 支持”。
- 右键单击“解决方案资源管理器”中的项目，然后选择“添加” > “Docker 支持”。

Visual Studio Tools for Docker 不支持向面向.NET Framework 的现有 ASP.NET Core 项目添加 Docker。

Dockerfile 概述

Dockerfile，用作创建最终 Docker 映像的方案，添加到项目根目录。请参阅 Dockerfile 引用，了解其中的命令。此特定 Dockerfile 使用多阶段生成，该生成包含四个不同的命名生成阶段：

```
FROM microsoft/dotnet:2.1-aspnetcore-runtime AS base
WORKDIR /app
EXPOSE 59518
EXPOSE 44364

FROM microsoft/dotnet:2.1-sdk AS build
WORKDIR /src
COPY HelloDockerTools/HelloDockerTools.csproj HelloDockerTools/
RUN dotnet restore HelloDockerTools/HelloDockerTools.csproj
COPY . .
WORKDIR /src/HelloDockerTools
RUN dotnet build HelloDockerTools.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish HelloDockerTools.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
```

前面的 Dockerfile 基于 microsoft/dotnet 映像。此基础映像包括 ASP.NET Core 运行时和 NuGet 包。对该包进行了实时 (JIT) 编译，以提高启动性能。

如果选中了新建项目对话框的“为 HTTPS 配置”复选框，则 Dockerfile 公开两个端口。一个端口用于 HTTP 流量；另一个端口用于 HTTPS。如果未选中该复选框，则为 HTTP 流量公开单个端口 (80)。

```
FROM microsoft/aspnetcore:2.0 AS base
WORKDIR /app
EXPOSE 80

FROM microsoft/aspnetcore-build:2.0 AS build
WORKDIR /src
COPY HelloDockerTools/HelloDockerTools.csproj HelloDockerTools/
RUN dotnet restore HelloDockerTools/HelloDockerTools.csproj
COPY . .
WORKDIR /src/HelloDockerTools
RUN dotnet build HelloDockerTools.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish HelloDockerTools.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
```

前面的 Dockerfile 基于 microsoft/aspnetcore 映像。此基础映像包括 ASP.NET Core NuGet 包，对该包进行了实时编译 (JIT)，以提高启动性能。

向应用添加容器业务流程协调程序支持

Visual Studio 2017 版本 15.7 或更早版本支持 Docker Compose 作为唯一的容器业务流程解决方案。可通过"添加" > "Docker 支持"添加 Docker Compose。

Visual Studio 2017 版本 15.8 或更高版本仅在获得指示时添加业务流程解决方案。右键单击"解决方案资源管理器"中的项目，然后选择"添加" > "容器业务流程协调程序支持"。提供了两个不同的选择：Docker Compose 和 Service Fabric。

Docker Compose

Visual Studio Tools for Docker 通过以下文件向解决方案添加 docker-compose 项目：

- docker-compose.dcproj – 表示项目的文件。包括 `<DockerTargetOS>` 元素，用于指定要使用的操作系统。
- .dockerignore – 列出在生成生成上下文时要排除的文件和目录模式。
- docker-compose.yml – 基本 Docker Compose 文件，用于定义要分别通过 `docker-compose build` 和 `docker-compose run` 生成和运行的映像集合。
- docker compose.override.yml – 一个可选文件，通过 Docker Compose 读取，包含服务的配置替代。Visual Studio 执行 `docker-compose -f "docker-compose.yml" -f "docker-compose.override.yml"` 以合并这些文件。

docker-compose.yml 文件引用在项目运行时创建的映像的名称：

```
version: '3.4'

services:
  hellodockertools:
    image: ${DOCKER_REGISTRY}hellodockertools
    build:
      context: .
      dockerfile: HelloDockerTools/Dockerfile
```

在前面的示例中，应用在"调试"模式下运行时，`image: hellodockertools` 生成映像 `hellodockertools:dev`。应用在"发布"模式下运行时，生成 `hellodockertools:latest` 映像。

如果将映像推送到注册表，则需要添加 Docker Hub 用户名（如 `dockerhubusername/hellodockertools`）作为映像名称的前缀名。或者，更改映像名称，使其包含专用注册表 URL（如 `privateregistry.domain.com/hellodockertools`），具

体取决于所用配置。

如果根据生成配置需要不同行为(例如，调试或发布)，请添加特定于配置的 docker-compose 文件。这些文件应根据生成配置进行命名(例如，docker-compose.vs.debug.yml 和 docker compose.vs.release.yml)并放置在与 docker-compose-override.yml 文件相同的位置。

使用特定于配置的替代文件，可以为调试和发布生成配置指定不同的配置设置(如环境变量或入口点)。

Service Fabric

除了基础必备组件外，Service Fabric 业务流程解决方案还需要以下必备组件：

- Microsoft Azure Service Fabric SDK 版本 2.6 或更高版本
- Visual Studio 2017 的“Azure 开发”工作负荷

Service Fabric 不支持在 Windows 上的本地开发群集中运行 Linux 容器。如果项目已在使用 Linux 容器，Visual Studio 会提示切换到 Windows 容器。

Visual Studio Tools for Docker 执行以下任务：

- 向解决方案添加 <project_name>Application Service Fabric 应用程序项目。
- 向 ASP.NET Core 项目添加 Dockerfile 和 .dockerignore 文件。如果 ASP.NET Core 项目中已存在 Dockerfile，则其重命名为 Dockerfile.original。创建类似于以下形式的新 Dockerfile：

  ```
  # See https://aka.ms/containerimagehelp for information on how to use Windows Server 1709 containers
  with Service Fabric.
  # FROM microsoft/aspnetcore:2.0-nanoserver-1709
  FROM microsoft/aspnetcore:2.0-nanoserver-sac2016
  ARG source
  WORKDIR /app
  COPY ${source:-obj/Docker/publish} .
  ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
  ```

- 将 `<IsServiceFabricServiceProject>` 元素添加到 ASP.NET Core 项目的 .csproj 文件：

  ```
  <IsServiceFabricServiceProject>True</IsServiceFabricServiceProject>
  ```

- 向 ASP.NET Core 项目添加 PackageRoot 文件夹。该文件夹包含服务清单和新服务的设置。

有关详细信息，请参阅将 Windows 容器中的 .NET 应用部署到 Azure Service Fabric。

调试

在工具栏的调试下拉列表中选择“Docker”，然后开始调试应用。“输出”窗口的“Docker”视图显示发生的以下操作：

- 已获取 microsoft/dotnet 运行时映像的 2.1-aspnetcore-runtime 标记(如果缓存中尚不存在)。该映像可安装 ASP.NET Core 和.NET Core 运行时及其关联的库。在生产环境中针对运行 ASP.NET Core 应用对其进行了优化。
- `ASPNETCORE_ENVIRONMENT` 环境变量设置为容器内的 `Development`。
- 公开了两个动态分配的端口：分别用于 HTTP 和 HTTPS。可以使用 `docker ps` 命令查询分配给本地主机的端口。
- 应用已复制到容器。
- 默认浏览器使用动态分配端口启动，带有附加到容器的调试程序。

最终得到的应用的 Docker 映像标记为“开发”。该映像基于 microsoft/dotnet 基础映像的 2.1-aspnetcore-runtime 标记。在“包管理器控制台”(PMC) 窗口中运行 `docker images` 命令。显示了计算机上的映像：

```
REPOSITORY        TAG                     IMAGE ID      CREATED         SIZE
hellodockertools  dev                     d72ce0f1dfe7  30 seconds ago  255MB
microsoft/dotnet  2.1-aspnetcore-runtime  fcc3887985bb  6 days ago      255MB
```

- 已获取 microsoft/aspnetcore 运行时映像(如果缓存中尚不存在)。
- `ASPNETCORE_ENVIRONMENT` 环境变量设置为容器内的 `Development`。
- 端口 80 公开，并映射到 localhost 的动态分配端口。该端口由 Docker 主机确定，并且可以使用 `docker ps` 进行查询。
- 应用已复制到容器。
- 默认浏览器使用动态分配端口启动，带有附加到容器的调试程序。

最终得到的应用的 Docker 映像标记为"开发"。该映像基于 microsoft/aspnetcore 基础映像。在"包管理器控制台"(PMC) 窗口中运行 `docker images` 命令。显示了计算机上的映像：

```
REPOSITORY            TAG  IMAGE ID      CREATED        SIZE
hellodockertools      dev  5fafe5d1ad5b  4 minutes ago  347MB
microsoft/aspnetcore  2.0  c69d39472da9  13 days ago    347MB
```

> **NOTE**
> 因为"调试"配置使用卷装载提供迭代体验，因此，开发映像中缺少应用内容。要推送映像，请使用"发布"配置。

在 PMC 中运行 `docker ps` 命令。请注意，应用使用容器运行：

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
PORTS                   NAMES
baf9a678c88d        hellodockertools:dev   "C:\\remote_debugge..."   21 seconds ago      Up 19 seconds
0.0.0.0:37630->80/tcp   dockercompose4642749010770307127_hellodockertools_1
```

编辑并继续

对静态文件和 Razor 视图的更改会自动更新，无需执行编译步骤。进行更改，保存并在浏览器中刷新，以查看更新。

必须在容器内编译和重启 Kestrel，才能修改代码文件。更改后，按 `CTRL+F5` 执行过程，并在容器内启动应用。未重新生成或停止 Docker 容器。在 PMC 中运行 `docker ps` 命令。请注意，截至 10 分钟前，原始容器仍在运行：

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
PORTS                   NAMES
baf9a678c88d        hellodockertools:dev   "C:\\remote_debugge..."   10 minutes ago      Up 10 minutes
0.0.0.0:37630->80/tcp   dockercompose4642749010770307127_hellodockertools_1
```

发布 Docker 映像

完成应用的开发和调试循环后，Visual Studio Tools for Docker 可帮助创建应用的生产映像。将配置下拉列表更改为"发布"，然后生成应用。该工具从 Docker Hub 中获取 compile/publish 映像(如果缓存中尚不存在)。生成了具有"latest"标签的映像，可以将其推送到专用注册表或 Docker Hub。

在 PMC 中运行 `docker images` 命令，查看映像列表。显示了类似下面的输出：

```
REPOSITORY        TAG                     IMAGE ID      CREATED             SIZE
hellodockertools  latest                  e3984a64230c  About a minute ago  258MB
hellodockertools  dev                     d72ce0f1dfe7  4 minutes ago       255MB
microsoft/dotnet  2.1-sdk                 9e243db15f91  6 days ago          1.7GB
microsoft/dotnet  2.1-aspnetcore-runtime  fcc3887985bb  6 days ago          255MB
```

```
REPOSITORY                  TAG     IMAGE ID      CREATED         SIZE
hellodockertools            latest  cd28f0d4abbd  12 seconds ago  349MB
hellodockertools            dev     5fafe5d1ad5b  23 minutes ago  347MB
microsoft/aspnetcore-build  2.0     7fed40fbb647  13 days ago     2.02GB
microsoft/aspnetcore        2.0     c69d39472da9  13 days ago     347MB
```

自 .NET Core 2.1 起，前面的输出中列出的 `microsoft/aspnetcore-build` 和 `microsoft/aspnetcore` 映像替换为 `microsoft/dotnet` 映像。有关详细信息，请参阅 Docker 存储库迁移公告。

> **NOTE**
>
> `docker images` 命令返回存储库名称和标记标识为 <none>（上面未列出）的中间映像。这些未命名映像由多阶段生成 *Dockerfile* 生成。它们可提高生成最终映像的效率 — 发生更改时，仅重新生成必要的层。不再需要中间映像时，请使用 docker rmi 命令将其删除。

可能希望生产或发布映像的大小比开发映像小。由于卷映射，调试程序和应用从本地计算机运行，而不在容器内运行。最新映像已打包必要的应用代码，以在主机上运行应用。因此，增量是应用代码的大小。

其他资源

- 利用 Visual Studio 进行容器开发
- : 准备开发环境
- 将 Windows 容器中的 .NET 应用部署到 Azure Service Fabric
- 排查使用 Docker 的 Visual Studio 2017 开发
- Visual Studio Tools for Docker GitHub 存储库

配置 ASP.NET Core 以使用代理服务器和负载均衡器

2018/10/12 • • Edit Online

作者：Luke Latham、Chris Ross

在 ASP.NET Core 推荐配置中，应用使用 IIS/ASP.NET Core 模块、Nginx 或 Apache 进行托管。代理服务器、负载均衡器和其他网络设备通常会在请求到达应用之前隐藏有关请求的信息：

- 当通过 HTTP 代理 HTTPS 请求时，原方案 (HTTPS) 将丢失，并且必须在标头中转接。
- 由于应用收到来自代理的请求，而不是 Internet 或公司网络上请求的真实源，因此原始客户端 IP 地址也必须在标头中转接。

此信息在请求处理中可能很重要，例如在重定向、身份验证、链接生成、策略评估和客户端地理位置中。

转接头

按照约定，代理转接 HTTP 标头中的信息。

HEADER	描述
X-Forwarded-For	保存代理链中关于发起请求的客户端和后续代理的信息。该参数可能包含 IP 地址（以及可选端口号）。在代理服务器链中，第一个参数表示最初发出请求的客户端。后续代理标识符以此类推。链中的最后一个代理不在参数列表中。最后一个代理的 IP 地址以及可选的端口号可用作传输层的远程 IP 地址。
X-Forwarded-Proto	原方案的值 (HTTP/HTTPS)。如果请求已遍历多个代理，则该值也可以是方案列表。
X-Forwarded-Host	主机标头字段的原始值。代理通常不会修改主机标头。有关特权提升漏洞的信息，请参阅 Microsoft 安全公告 CVE-2018-0787，该漏洞影响代理未验证或将主机标头限制为已知正确值的系统。

来自 Microsoft.AspNetCore.HttpOverrides 包的转接标头中间件读取这些标头，并填充 HttpContext 上的关联字段。

中间件更新：

- HttpContext.Connection.RemoteIpAddress – 使用 `X-Forwarded-For` 标头值进行设置。影响中间件如何设置 `RemoteIpAddress` 的其他设置。有关详细信息，请参阅转接头中间件选项。
- HttpContext.Request.Scheme – 使用 `X-Forwarded-Proto` 标头值进行设置。
- HttpContext.Request.Host – 使用 `X-Forwarded-Host` 标头值进行设置。

可以配置转接头中间件默认设置。默认设置为：

- 应用和请求源之间只有一个代理。
- 仅将环回地址配置为已知代理和已知网络。
- 转接头被命名为 `X-Forwarded-For` 和 `X-Forwarded-Proto`。

并非所有网络设备均可在没有其他配置的情况下添加 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头。如果代理请求在到达应用时未包含这些标头，请查阅设备制造商提供的指导。如果设备使用 `X-Forwarded-For` 和 `X-Forwarded-Proto` 以外的其他标头，请设置 ForwardedForHeaderName 和 ForwardedProtoHeaderName 选项，使其与设备所用的标头名称相匹配。有关详细信息，请参阅转接头中间件选项和配置使用不同标头名称的代理。

IIS/IIS Express 和 ASP.NET Core 模块

当应用在 IIS 和 ASP.NET Core 模块后方运行时，转接头中间件由 IIS 集成中间件默认启用。由于转接头的信任问题(例如，IP 欺骗)，转接头中间件被激活为首先在中间件管道中运行，并具有特定于 ASP.NET Core 模块的受限配置。中间件配置为转接 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头，并且被限制到单个本地 localhost 代理。如果需要其他配置，请参阅转接头中间件选项。

其他代理服务器和负载均衡器方案

除使用 IIS 集成中间件之外，不会默认启用转接头中间件。必须启用应用的转接头中间件才能处理带有 UseForwardedHeaders 的转接头。启用中间件后，如果没有为中间件指定 ForwardedHeadersOptions，那么默认的 ForwardedHeadersOptions.ForwardedHeaders 是 ForwardedHeaders.None。

使用 ForwardedHeadersOptions 配置中间件，以转接 `Startup.ConfigureServices` 中的 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头。调用其他中间件之前，调用 `Startup.Configure` 中的 UseForwardedHeaders 方法：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.Configure<ForwardedHeadersOptions>(options =>
    {
        options.ForwardedHeaders =
            ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
    });
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseForwardedHeaders();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();
    // In ASP.NET Core 1.x, replace the following line with: app.UseIdentity();
    app.UseAuthentication();
    app.UseMvc();
}
```

NOTE

如果 ForwardedHeadersOptions 没有在 `Startup.ConfigureServices` 中指定或没有通过 UseForwardedHeaders(IApplicationBuilder, ForwardedHeadersOptions) 直接指向扩展方法，则要转接的默认标头为 ForwardedHeaders.None。ForwardedHeadersOptions.ForwardedHeaders 属性必须配置为标头才能转接。

Nginx 配置

要转发 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头，请参阅 使用 Nginx 在 Linux 上托管 ASP.NET Core。有关详细信息，请参阅 NGINX：使用转接头。

Apache 配置

`X-Forwarded-For` 将会自动添加（请参阅 Apache 模块 mod_proxy：反向代理请求标头）。要了解如何转发 `X-Forwarded-Proto` 标头，请参阅 使用 Apache 在 Linux 上托管 ASP.NET Core。

转接头中间件选项

ForwardedHeadersOptions 控制转接头中间件的行为。以下示例更改了默认值：

- 将转接头中的条目数量限制为 `2`。
- 添加已知的代理地址 `127.0.10.1`。
- 将转接头名称从默认的 `X-Forwarded-For` 更改为 `X-Forwarded-For-My-Custom-Header-Name`。

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardLimit = 2;
    options.KnownProxies.Add(IPAddress.Parse("127.0.10.1"));
    options.ForwardedForHeaderName = "X-Forwarded-For-My-Custom-Header-Name";
});
```

选项	描述
AllowedHosts	通过 `X-Forwarded-Host` 标头将主机限制为提供的值。 • 使用 ordinal-ignore-case 比较值。 • 必须排除端口号。 • 如果列表为空，则允许使用所有主机。 • 顶级通配符 `*` 允许所有非空主机。 • 子域通配符允许使用，但不匹配根域。例如，`*.contoso.com` 匹配子域 `foo.contoso.com`，但不匹配根域 `contoso.com`。 • 允许使用 Unicode 主机名，但应转换为 Punycode 进行匹配。 • IPv6 地址必须包括边界方括号，而且必须位于常规窗体（例如，`[ABCD:EF01:2345:6789:ABCD:EF01:2345:6789]`）中。IPv6 地址并非专门用于检查不同格式之间的逻辑相等性，也不执行标准化。 • 未能限制允许的主机可能会允许攻击者访问该服务生成的欺骗链接。 默认值为一个空 IList<string>。
ForwardedForHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XForwardedForHeaderName 指定的标头。如果代理/转发器不使用 `X-Forwarded-For` 标头，但使用一些其他标头来转发信息，则使用此选项。 默认值为 `X-Forwarded-For`。

选项	描述
ForwardedHeaders	标识应处理的转发器。对于应用的字段的列表，请参阅 ForwardedHeaders 枚举。分配给此属性的典型值为 `ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto` 。 默认值是 ForwardedHeaders.None。
ForwardedHostHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XForwardedHostHeaderName 指定的标头。如果代理/转发器不使用 `X-Forwarded-Host` 标头，但使用一些其他标头来转发信息，则使用此选项。 默认值为 `X-Forwarded-Host` 。
ForwardedProtoHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XForwardedProtoHeaderName 指定的标头。如果代理/转发器不使用 `X-Forwarded-Proto` 标头，但使用一些其他标头来转发信息，则使用此选项。 默认值为 `X-Forwarded-Proto` 。
ForwardLimit	限制所处理的标头中的条目数。设置为 `null` 以禁用此限制，但仅应在已配置 `KnownProxies` 或 `KnownNetworks` 的情况下执行此操作。 默认值为 1。
KnownNetworks	从中接受转接头的已知网络的地址范围。使用无类别域际路由选择 (CIDR) 表示法提供 IP 范围。 如果服务器使用双模式套接字，则采用 IPv6 格式提供 IPv4 地址（例如，IPv4 格式的 `10.0.0.1` 以 IPv6 格式表示为 `::ffff:10.0.0.1` ）。请参阅 IPAddress.MapToIPv6。通过查看 HttpContext.Connection.RemoteIpAddress 来确定是否需要采用此格式。有关详细信息，请参阅对表示为 IPv6 地址的 IPv4 地址进行配置部分。 默认值是 IList<IPNetwork>，包含 `IPAddress.Loopback` 的单个条目。
KnownProxies	从中接受转接头的已知代理的地址。使用 `KnownProxies` 指定精确的 IP 地址匹配。 如果服务器使用双模式套接字，则采用 IPv6 格式提供 IPv4 地址（例如，IPv4 格式的 `10.0.0.1` 以 IPv6 格式表示为 `::ffff:10.0.0.1` ）。请参阅 IPAddress.MapToIPv6。通过查看 HttpContext.Connection.RemoteIpAddress 来确定是否需要采用此格式。有关详细信息，请参阅对表示为 IPv6 地址的 IPv4 地址进行配置部分。 默认值是 IList<IPAddress>，包含 `IPAddress.IPv6Loopback` 的单个条目。

选项	描述
OriginalForHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XOriginalForHeaderName 指定的标头。 默认值为 `X-Original-For` 。
OriginalHostHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XOriginalHostHeaderName 指定的标头。 默认值为 `X-Original-Host` 。
OriginalProtoHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XOriginalProtoHeaderName 指定的标头。 默认值为 `X-Original-Proto` 。
RequireHeaderSymmetry	要求正在处理的 ForwardedHeadersOptions.ForwardedHeaders 之间的标头值数量保持同步。 ASP.NET Core 1.x 默认值是 `true` 。ASP.NET Core 2.0 默认值是 `false` 。

选项	描述
ForwardedForHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XForwardedForHeaderName 指定的标头。如果代理/转发器不使用 `X-Forwarded-For` 标头，但使用一些其他标头来转发信息，则使用此选项。 默认值为 `X-Forwarded-For` 。
ForwardedHeaders	标识应处理的转发器。对于应用的字段的列表，请参阅 ForwardedHeaders 枚举。分配给此属性的典型值为 `ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto` 。 默认值是 ForwardedHeaders.None。
ForwardedHostHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XForwardedHostHeaderName 指定的标头。如果代理/转发器不使用 `X-Forwarded-Host` 标头，但使用一些其他标头来转发信息，则使用此选项。 默认值为 `X-Forwarded-Host` 。

选项	描述
ForwardedProtoHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XForwardedProtoHeaderName 指定的标头。如果代理/转发器不使用 `X-Forwarded-Proto` 标头，但使用一些其他标头来转发信息，则使用此选项。 默认值为 `X-Forwarded-Proto`。
ForwardLimit	限制所处理的标头中的条目数。设置为 `null` 以禁用此限制，但仅应在已配置 `KnownProxies` 或 `KnownNetworks` 的情况下执行此操作。 默认值为 1。
KnownNetworks	从中接受转接头的已知网络的地址范围。使用无类别域际路由选择 (CIDR) 表示法提供 IP 范围。 如果服务器使用双模式套接字，则采用 IPv6 格式提供 IPv4 地址（例如，IPv4 格式的 `10.0.0.1` 以 IPv6 格式表示为 `::ffff:10.0.0.1`）。请参阅 IPAddress.MapToIPv6。通过查看 HttpContext.Connection.RemoteIpAddress 来确定是否需要采用此格式。有关详细信息，请参阅对表示为 IPv6 地址的 IPv4 地址进行配置部分。 默认值是 IList<IPNetwork>，包含 `IPAddress.Loopback` 的单个条目。
KnownProxies	从中接受转接头的已知代理的地址。使用 `KnownProxies` 指定精确的 IP 地址匹配。 如果服务器使用双模式套接字，则采用 IPv6 格式提供 IPv4 地址（例如，IPv4 格式的 `10.0.0.1` 以 IPv6 格式表示为 `::ffff:10.0.0.1`）。请参阅 IPAddress.MapToIPv6。通过查看 HttpContext.Connection.RemoteIpAddress 来确定是否需要采用此格式。有关详细信息，请参阅对表示为 IPv6 地址的 IPv4 地址进行配置部分。 默认值是 IList<IPAddress>，包含 `IPAddress.IPv6Loopback` 的单个条目。
OriginalForHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XOriginalForHeaderName 指定的标头。 默认值为 `X-Original-For`。
OriginalHostHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XOriginalHostHeaderName 指定的标头。 默认值为 `X-Original-Host`。

选项	描述
OriginalProtoHeaderName	使用由此属性指定的标头，而不是由 ForwardedHeadersDefaults.XOriginalProtoHeaderName 指定的标头。 默认值为 `X-Original-Proto`。
RequireHeaderSymmetry	要求正在处理的 ForwardedHeadersOptions.ForwardedHeaders 之间的标头值数量保持同步。 ASP.NET Core 1.x 默认值是 `true`。ASP.NET Core 2.0 默认值是 `false`。

方案与使用案例

无法添加转接头并且所有请求都安全的情况

在某些情况下，可能无法将转接头添加到代理到应用的请求中。如果代理强制将所有公共外部请求执行为 HTTPS，则在使用任何类型的中间件之前，可以在 `Startup.Configure` 中手动设置该方案：

```
app.Use((context, next) =>
{
    context.Request.Scheme = "https";
    return next();
});
```

此代码可以通过环境变量或者开发环境或过渡环境中的其他配置设置来禁用。

处理改变请求路径的基路径和代理

一些代理可完整地传递路径，但是会删除应用基路径以便路由可正常工作。UsePathBaseExtensions.UsePathBase 中间件将路径拆分为 HttpRequest.Path，将应用基路径拆分为 HttpRequest.PathBase。

如果 `/foo` 是作为 `/foo/api/1` 传递的代理路径的应用基路径，则中间件使用以下命令将 `Request.PathBase` 设置为 `/foo`，将 `Request.Path` 设置为 `/api/1`：

```
app.UsePathBase("/foo");
```

当再次反向调用中间件时，将重新应用原始路径和基路径。要详细了解中间件处理顺序，请参阅 ASP.NET Core 中间件。

如果代理剪裁路径（例如，将 `/foo/api/1` 转接到 `/api/1`），请通过设置请求的 PathBase 属性来修复重定向和链接：

```
app.Use((context, next) =>
{
    context.Request.PathBase = new PathString("/foo");
    return next();
});
```

如果代理正在添加路径数据，请使用 StartsWithSegments(PathString, PathString) 并分配给 Path 属性，从而放弃部分路径以修复重定向和链接：

```
app.Use((context, next) =>
{
    if (context.Request.Path.StartsWithSegments("/foo", out var remainder))
    {
        context.Request.Path = remainder;
    }

    return next();
});
```

配置使用不同标头名称的代理

如果代理不使用名为 `X-Forwarded-For` 和 `X-Forwarded-Proto` 的标头来转发代理地址/端口和原始架构信息，则设置 ForwardedForHeaderName 和 ForwardedProtoHeaderName 选项，使其与代理所用的标头名称相匹配：

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedForHeaderName = "Header_Name_Used_By_Proxy_For_X-Forwarded-For_Header";
    options.ForwardedProtoHeaderName = "Header_Name_Used_By_Proxy_For_X-Forwarded-Proto_Header";
});
```

对表示为 IPv6 地址的 IPv4 地址进行配置

如果服务器使用双模式套接字，则采用 IPv6 格式提供 IPv4 地址（例如，IPv4 格式的 `10.0.0.1` 以 IPv6 格式表示为 `::ffff:10.0.0.1` 或 `::ffff:a00:1`）。请参阅 IPAddress.MapToIPv6。通过查看 HttpContext.Connection.RemoteIpAddress 来确定是否需要采用此格式。

在以下示例中，提供转接头的网络地址以 IPv6 格式添加到 `KnownNetworks` 列表中。

IPv4 地址：`10.11.12.1/8`

转换后的 IPv6 地址：`::ffff:10.11.12.1`
转换后的前缀长度：104

也可以提供十六进制格式的地址（`10.11.12.1` 以 IPv6 格式表示为 `::ffff:0a0b:0c01`）。将 IPv4 地址转换为 IPv6 时，将 96 添加到 CIDR 前缀长度（示例中的 `8`）以说明其他 `::ffff:` IPv6 前缀 (8 + 96 = 104)。

```
// To access IPNetwork and IPAddress, add the following namespaces:
// using using System.Net;
// using Microsoft.AspNetCore.HttpOverrides;
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
    options.KnownNetworks.Add(new IPNetwork(
        IPAddress.Parse("::ffff:10.11.12.1"), 104));
});
```

疑难解答

如果未按预期转接标头，请启用日志记录。如果日志没有提供足够的信息来解决问题，请枚举服务器收到的请求标头。使用内联中间件将请求标头写入应用程序响应或记录标头。调用 `Startup.Configure` 中的 UseForwardedHeaders 后，立即放置以下任意代码示例。

若要将标头写入应用程序的响应，请使用以下终端内联中间件：

```
app.Run(async (context) =>
{
    context.Response.ContentType = "text/plain";

    // Request method, scheme, and path
    await context.Response.WriteAsync(
        $"Request Method: {context.Request.Method}{Environment.NewLine}");
    await context.Response.WriteAsync(
        $"Request Scheme: {context.Request.Scheme}{Environment.NewLine}");
    await context.Response.WriteAsync(
        $"Request Path: {context.Request.Path}{Environment.NewLine}");

    // Headers
    await context.Response.WriteAsync($"Request Headers:{Environment.NewLine}");

    foreach (var header in context.Request.Headers)
    {
        await context.Response.WriteAsync($"{header.Key}: " +
            $"{header.Value}{Environment.NewLine}");
    }

    await context.Response.WriteAsync(Environment.NewLine);

    // Connection: RemoteIp
    await context.Response.WriteAsync(
        $"Request RemoteIp: {context.Connection.RemoteIpAddress}");
});
```

还可以通过使用以下内联中间件写入日志而不是响应正文。这样可以使站点在调试时正常运行。

```
var logger = _loggerFactory.CreateLogger<Startup>();

app.Use(async (context, next) =>
{
    // Request method, scheme, and path
    logger.LogDebug("Request Method: {METHOD}", context.Request.Method);
    logger.LogDebug("Request Scheme: {SCHEME}", context.Request.Scheme);
    logger.LogDebug("Request Path: {PATH}", context.Request.Path);

    // Headers
    foreach (var header in context.Request.Headers)
    {
        logger.LogDebug("Header: {KEY}: {VALUE}", header.Key, header.Value);
    }

    // Connection: RemoteIp
    logger.LogDebug("Request RemoteIp: {REMOTE_IP_ADDRESS}",
        context.Connection.RemoteIpAddress);

    await next();
});
```

处理时，`X-Forwarded-{For|Proto|Host}` 值将移至 `X-Original-{For|Proto|Host}`。如果给定标头中有多个值，请注意转接头中间件按照从右向左的相反顺序处理标头。默认 `ForwardLimit` 为 1(一)，因此只会处理标头最右侧的值，除非增加 `ForwardLimit` 的值。

在处理转接头之前，请求的原始远程 IP 必须与 `KnownProxies` 或 `KnownNetworks` 列表中的条目匹配。这通过不接受来自不受信任的代理的转发器来限制标头欺骗。检测到未知代理时，日志记录会指出代理的地址：

```
September 20th 2018, 15:49:44.168 Unknown proxy: 10.0.0.100:54321
```

在上述示例中，10.0.0.100 是代理服务器。如果该服务器是受信任的代理，请将服务器的 IP 地址添加到 `Startup.ConfigureServices` 中的 `KnownProxies`（或将受信任的网络添加到 `KnownNetworks`）。有关详细信息，请参阅转接头中间件选项部分。

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});
```

> **IMPORTANT**
>
> 仅允许受信任的代理和网络转接头。否则，可能会受到 IP 欺骗攻击。

其他资源

- 在 Web 场中托管 ASP.NET Core
- Microsoft 安全公告 CVE-2018-0787：ASP.NET Core 特权提升漏洞

在 Web 场中托管 ASP.NET Core

2018/12/12 • • Edit Online

作者：Luke Latham、Chris Ross

Web 场包含两个或多个 Web 服务器（亦称为"节点"），用于托管应用的多个实例。若有用户请求到达 Web 场，负载均衡器会将请求分发到 Web 场中的各个节点。Web 场提高了：

- **可靠性/可用性** – 如果一个或多个节点失败，负载均衡器可以将请求路由到其他正常运行的节点，以继续处理请求。
- **容量/性能** – 多个节点可以处理的请求数多于一个服务器。负载均衡器均衡工作负载的方式是，将请求分发到各个节点。
- **可伸缩性** – 如果需要更多或更少容量，可以增加或减少活动节点数，与工作负载保持一致。Azure 应用服务等 Web 场平台技术可以应系统管理员的请求自动添加或删除节点，也可以自动开始，而无需人为干预。
- **可维护性** – Web 场节点可以依赖一组共享服务，这就简化了系统管理。例如，Web 场中的节点可以依赖单一数据库服务器，以及静态资源（如图像和可下载文件）的公用网络位置。

本主题介绍了在 Web 场中托管且依赖共享资源的 ASP.NET Core 应用的配置和依赖项。

常规配置

托管和部署 ASP.NET Core
了解如何设置托管环境和部署 ASP.NET Core 应用。对 Web 场中的每个节点配置进程管理器，以自动启动和重启应用。每个节点都需要 ASP.NET Core 运行时。有关详细信息，请参阅文档的托管和部署区域中的主题。

配置 ASP.NET Core 以使用代理服务器和负载均衡器
了解在代理服务器和负载均衡器后方托管的应用程序的配置，这通常会隐藏重要的请求信息。

将 ASP.NET Core 应用部署到 Azure 应用服务
Azure 应用服务是一个用于托管 Web 应用（包括 ASP.NET Core）的 Microsoft 云计算平台服务。应用服务是提供自动缩放、负载均衡、修补和持续部署的完全托管平台。

应用数据

如果应用已缩放为多个实例，可能会有需要跨节点共享的应用状态。若为暂时状态，建议共享 IDistributedCache。如果需要暂留共享状态，建议在数据库中存储共享状态。

必需配置

必需为部署到 Web 场的应用配置数据保护和缓存。

数据保护

应用使用 ASP.NET Core 数据保护系统来保护数据。数据保护系统依赖一组在密钥环中存储的加密密钥。初始化后，数据保护系统会应用在本地存储密钥环的默认设置。根据默认配置，唯一密钥环存储在 Web 场的各个节点上。因此，Web 场中的每个节点都无法解密应用在其他任何节点上加密的数据。默认配置通常不适合在 Web 场中托管应用。若要实现共享密钥环，可以改为始终将用户请求路由到相同的节点。若要详细了解与 Web 场部署有关的数据保护系统配置，请参阅配置 ASP.NET Core 数据保护。

缓存

在 Web 场环境中，缓存机制必须跨 Web 场中的节点共享缓存项。缓存必须依赖公用 Redis 缓存、共享 SQL Server 数据库，或跨 Web 场共享缓存项的自定义缓存实现。有关更多信息，请参见分布式缓存在 ASP.NET Core

中。

依赖组件

下面的方案无需其他配置，但依赖需要配置 Web 场的技术。

方案	依赖...
身份验证	数据保护（请参阅配置 ASP.NET Core 数据保护）。 有关详细信息，请参阅 使用 cookie 而无需 ASP.NET Core 标识的身份验证 和 与 ASP.NET 和 ASP.NET Core 共享在应用之间的 cookie。
标识	身份验证和数据库配置。 有关更多信息，请参见ASP.NET Core 上的标识简介。
会话	数据保护（加密 Cookie）（请参阅配置 ASP.NET Core 数据保护）和缓存（请参阅分布式缓存在 ASP.NET Core 中）。 有关详细信息，请参阅会话和应用状态：会话状态。
TempData	数据保护（加密 Cookie）（请参阅配置 ASP.NET Core 数据保护）或会话（请参阅会话和应用状态：会话状态）。 有关详细信息，请参阅会话和应用状态：TempData。
防伪造	数据保护（请参阅配置 ASP.NET Core 数据保护）。 有关更多信息，请参见在 ASP.NET Core 防止跨站点请求伪造 (XSRF/CSRF) 攻击。

疑难解答

数据保护和缓存

如果未为 Web 场环境配置数据保护或缓存，就会在处理请求时发生间歇性错误。之所以会发生这种情况是因为，节点不共享相同的资源，并且用户请求并不总是路由回同一节点。

假设用户通过 Cookie 身份验证来登录应用。用户在 Web 场中的一个节点上登录应用。如果用户的下一个请求到达登录应用时所用的同一节点，应用便能解密身份验证 Cookie，并允许用户访问应用资源。如果用户的下一个请求到达其他节点，应用便无法从用户登录时所用的节点解密身份验证 Cookie，并且无法授权用户请求获取的资源。

如果以下任一症状间歇性出现，问题原因通常是为 Web 场环境配置的数据保护或缓存不正确：

- 身份验证中断 – 身份验证 Cookie 配置不正确或无法解密。OAuth（Facebook、Microsoft、Twitter）或 OpenIdConnect 登录失败，出现错误“关联失败”。
- 授权中断 – 标识丢失。
- 会话状态丢失数据。
- 缓存项消失。
- TempData 失败。
- POST 失败 – 防伪造检查失败。

若要详细了解与 Web 场部署有关的数据保护配置，请参阅配置 ASP.NET Core 数据保护。若要详细了解与 Web 场部署有关的缓存配置，请参阅分布式缓存在 ASP.NET Core 中。

从应用中获取数据

如果 Web 场应用能够响应请求，则使用终端内联中间件从应用中获取请求、连接和其他数据。有关详细信息和示例代码，请参阅解决 ASP.NET Core 项目。

用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件

2019/1/10 • • Edit Online

作者：Sayed Ibrahim Hashimi 和 Rick Anderson

对于本主题的 1.1 版本，请下载用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件（版本 1.1，PDF）。

本文档重点介绍如何使用 Visual Studio 2017 或更高版本创建并使用发布配置文件。可以从 MSBuild 和 Visual Studio 运行使用 Visual Studio 创建的发布配置文件。有关发布到 Azure 的说明，请参阅使用 Visual Studio 将 ASP.NET Core Web 应用发布到 Azure App Service。

使用命令 `dotnet new mvc` 创建以下项目文件：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

`<Project>` 元素的 `Sdk` 属性完成以下任务：

- 在开始时从 $(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.Props 导入属性文件。
- 在结束时从 $(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.targets 导入目标文件。

`MSBuildSDKsPath`（装有 Visual Studio 2017 Enterprise）的默认位置是 %programfiles(x86)%\Microsoft Visual Studio\2017\Enterprise\MSBuild\Sdks 文件夹。

`Microsoft.NET.Sdk.Web` SDK 依赖于：

- *Microsoft.NET.Sdk.Web.ProjectSystem*
- *Microsoft.NET.Sdk.Publish*

它们会导致导入以下属性和目标：

- $(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web.ProjectSystem\Sdk\Sdk.Props

- $(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web.ProjectSystem\Sdk\Sdk.targets
- $(MSBuildSDKsPath)\Microsoft.NET.Sdk.Publish\Sdk\Sdk.Props
- $(MSBuildSDKsPath)\Microsoft.NET.Sdk.Publish\Sdk\Sdk.targets

发布目标将根据使用的发布方法，导入正确的目标集。

MSBuild 或 Visual Studio 加载项目时，执行下列高级别操作：

- 生成项目
- 计算要发布的文件
- 将文件发布到目标

计算项目项

加载项目时，将计算项目项（文件）。`item type` 属性确定如何处理该文件。默认情况下，.cs 文件包含在 `Compile` 项列表内。会对 `Compile` 项列表中的文件进行编译。

除了生成输出，`Content` 项列表还包含已发布的文件。默认情况下，匹配模式 `wwwroot/**` 的文件将包含在 `Content` 项内。`wwwroot/\*\/\*` glob 模式 匹配 wwwroot 文件夹及其子文件夹中的所有文件。若要将文件显式添加到发布列表，请直接在 .csproj 文件中添加文件，如包含文件中所示。

在 Visual Studio 中选择"发布"按钮时或从命令行发布时：

- 计算属性/项目（需要生成的文件）。
- 仅限 **Visual Studio**：NuGet 包已还原。（用户需要在 CLI 上执行显式还原。）
- 生成项目。
- 计算发布项（需要发布的文件）。
- 文件已发布（计算的文件将被复制到发布目标）。

当 ASP.NET Core 项目在项目文件中引用 `Microsoft.NET.Sdk.Web` 时，会将 app_offline.htm 文件放在 Web 应用目录的根目录下。该文件存在时，ASP.NET Core 模块会在部署过程中正常关闭该应用并提供 app_offline.htm 文件。有关详细信息，请参阅 ASP.NET Core 模块配置参考。

基本命令行发布

命令行发布适用于所有支持 .NET Core 的平台，而且不需要 Visual Studio。在下面的示例中，从项目目录（其中包含 .csproj 文件）运行 dotnet publish 命令。如果不在项目文件夹中，则可以在项目文件路径中显式传递。例如：

```
dotnet publish C:\Webs\Web1
```

运行以下命令以创建并发布 Web 应用：

```
dotnet new mvc
dotnet publish
```

dotnet publish 命令生成类似下面的输出：

```
C:\Webs\Web1>dotnet publish
Microsoft (R) Build Engine version 15.3.409.57025 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Web1 -> C:\Webs\Web1\bin\Debug\netcoreapp2.0\Web1.dll
  Web1 -> C:\Webs\Web1\bin\Debug\netcoreapp2.0\publish\
```

默认发布文件夹为 `bin\$(Configuration)\netcoreapp<version>\publish`。`$(Configuration)` 的默认值为 Debug。在上述示例中，`<TargetFramework>` 是 `netcoreapp2.0`。

`dotnet publish -h` 显示用于发布的帮助信息。

以下命令指定 `Release` 生成和发布目录：

```
dotnet publish -c Release -o C:\MyWebs\test
```

dotnet publish 命令调用 MSBuild，它调用 `Publish` 目标。任何传递给 `dotnet publish` 的参数都将传递给 MSBuild。`-c` 参数映射到 `Configuration` MSBuild 属性。`-o` 参数映射到 `OutputPath`。

可以使用以下任一格式来传递 MSBuild 属性：

- `p:<NAME>=<VALUE>`
- `/p:<NAME>=<VALUE>`

以下命令将 `Release` 版本发布到网络共享：

`dotnet publish -c Release /p:PublishDir=//r8/release/AdminWeb`

网络共享通过正斜杠指定 (//r8/) 并适用于所有支持 .NET Core 的平台。

确认用于部署的发布应用未在运行。如果应用正在运行，publish 文件夹中的文件会被锁定。部署不会发生，因为无法复制锁定的文件。

发布配置文件

本部分使用 Visual Studio 2017 或更高版本创建发布配置文件。创建配置文件后，可以从 Visual Studio 或命令行进行发布。

发布配置文件可以简化发布过程，并且可以存在任意数量的配置文件。通过选择以下路径之一在 Visual Studio 中创建发布配置文件：

- 在解决方案资源管理器中右键单击该项目，然后选择“发布”。
- 可以从“生成”菜单中选择“发布 {项目名称}”。

随即显示应用程序容量页的“发布”选项卡。如果项目缺少发布配置文件，将显示以下页面：

当选择"文件夹"时，指定一个文件夹路径来存储发布的资产。默认文件夹是 bin\Release\PublishOutput。单击"创建配置文件"按钮即可完成。

创建发布配置文件后，"发布"选项卡将更改。新创建的配置文件显示在下拉列表中。单击"创建新配置文件"以创建其他新配置文件。

发布向导支持以下发布目标：

- Azure 应用服务
- Azure 虚拟机
- IIS、FTP 等（适用于任何 Web 服务器）
- 文件夹
- 导入配置文件

有关详细信息，请参阅哪些发布选项适合我。

使用 Visual Studio 创建发布配置文件时，将创建 Properties/PublishProfiles/{PROFILE NAME}.pubxml MSBuild 文件。此 .pubxml 文件为 MSBuild 文件，包含发布配置设置。可以更改此文件以自定义生成和发布过程。通过发布过程读取此文件。`<LastUsedBuildConfiguration>` 比较特殊，因为它是一个全局属性，不应出现在导入生成的任何文件中。有关详细信息，请参阅 MSBuild：如何设置配置属性。

发布到 Azure 目标时，.pubxml 文件包含 Azure 订阅标识符。不建议使用该目标类型将此文件添加到源代码管理。发布到非 Azure 目标时，签入 .pubxml 文件是安全的。

敏感信息（如发布密码）在每个用户/机器级别均进行加密。它存储在 Properties/PublishProfiles/{PROFILE NAME}.pubxml.user 文件中。由于此文件可以存储敏感信息，因此不应将其签入源代码管理。

有关如何在 ASP.NET Core 上发布 Web 应用的概述，请参阅托管和部署。发布 ASP.NET Core 应用程序所需的 MSBuild 任务和目标是开放源代码，位于：https://github.com/aspnet/websdk。

`dotnet publish` 可以使用文件夹、MSDeploy 和 Kudu 发布配置文件：

文件夹（跨平台工作）：

```
dotnet publish WebApplication.csproj /p:PublishProfile=<FolderProfileName>
```

MSDeploy（当前仅适用于 Windows，因为 MSDeploy 不是跨平台的）：

```
dotnet publish WebApplication.csproj /p:PublishProfile=<MsDeployProfileName> /p:Password=
<DeploymentPassword>
```

MSDeploy 包（当前仅适用于 Windows，因为 MSDeploy 不是跨平台的）：

```
dotnet publish WebApplication.csproj /p:PublishProfile=<MsDeployPackageProfileName>
```

在前面示例中，不会将 `deployonbuild` 传递到 `dotnet publish`。

有关详细信息，请参阅 Microsoft.NET.Sdk.Publish。

`dotnet publish` 支持 Kudu API 从任何平台发布到 Azure。Visual Studio 发布支持 Kudu API，但 WebSDK 支持其跨平台发布到 Azure。

向"属性/发布配置文件"文件夹添加包含以下内容的发布配置文件：

```
<Project>
  <PropertyGroup>
    <PublishProtocol>Kudu</PublishProtocol>
    <PublishSiteName>nodewebapp</PublishSiteName>
    <UserName>username</UserName>
    <Password>password</Password>
  </PropertyGroup>
</Project>
```

运行以下命令将会压缩发布内容并将其发布到使用 Kudu API 的 Azure：

```
dotnet publish /p:PublishProfile=Azure /p:Configuration=Release
```

使用发布配置文件时，请设置以下 MSBuild 属性：

- `DeployOnBuild=true`
- `PublishProfile={PUBLISH PROFILE}`

发布名为 FolderProfile 的配置文件时，可以执行以下命令之一：

- `dotnet build /p:DeployOnBuild=true /p:PublishProfile=FolderProfile`
- `msbuild /p:DeployOnBuild=true /p:PublishProfile=FolderProfile`

调用 dotnet build 时，它调用 `msbuild` 来运行生成和发布过程。当传递到文件夹配置文件中时，调用 `dotnet build` 或 `msbuild` 的作用是相同的。在 Windows 上直接调用 MSBuild 时，将使用 MSBuild 的 .NET Framework 版本。MSDeploy 目前仅限于在 Windows 计算机上进行发布。在非文件夹配置文件上调用 `dotnet build` 时，会调用 MSBuild，并且 MSBuild 在非文件夹配置文件上使用 MSDeploy。在非文件夹配置文件上调用 `dotnet build` 时，会调用 MSBuild（使用 MSDeploy）并导致失败（即使在 Windows 平台上运行也是如此）。若要使用非文件夹配置文件进行发布，请直接调用 MSBuild。

以下文件夹发布配置文件通过 Visual Studio 创建，并被发布到网络共享：

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This file is used by the publish/package process of your Web project.
You can customize the behavior of this process by editing this
MSBuild file.
-->
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <PublishProvider>FileSystem</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish />
    <LaunchSiteAfterPublish>True</LaunchSiteAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <PublishFramework>netcoreapp1.1</PublishFramework>
    <ProjectGuid>c30c453c-312e-40c4-aec9-394a145dee0b</ProjectGuid>
    <publishUrl>\\r8\Release\AdminWeb</publishUrl>
    <DeleteExistingFiles>False</DeleteExistingFiles>
  </PropertyGroup>
</Project>
```

请注意，`<LastUsedBuildConfiguration>` 已设置为 `Release`。从 Visual Studio 发布时，在启动发布过程后将使用该值设置 `<LastUsedBuildConfiguration>` 配置属性值。`<LastUsedBuildConfiguration>` 配置属性比较特殊，不应在导入的 MSBuild 文件中覆盖该属性。可以从命令行覆盖此属性。

使用 .NET Core CLI：

```
dotnet build -c Release /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

使用 MSBuild：

```
msbuild /p:Configuration=Release /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

从命令行发布到 MSDeploy 终结点

下面的示例使用由 Visual Studio 创建的 ASP.NET Core Web 应用，名为 AzureWebApp。通过 Visual Studio 添加 Azure 应用发布配置文件。有关如何创建配置文件的详细信息，请参阅发布配置文件部分。

若要使用发布配置文件部署应用，请在 Visual Studio 开发人员命令提示中执行 `msbuild` 命令。Windows 任务栏上的"开始"菜单的"Visual Studio"文件夹中提供命令提示符。为了便于访问，可将命令提示符添加到 Visual Studio 中的"工具"菜单中。有关详细信息，请参阅 Visual Studio 开发人员命令提示符。

MSBuild 使用以下命令语法：

```
msbuild {PATH}
    /p:DeployOnBuild=true
    /p:PublishProfile={PROFILE}
    /p:Username={USERNAME}
    /p:Password={PASSWORD}
```

- {PATH} – 应用的项目文件路径。
- {PROFILE} – 发布配置文件的名称。
- {USERNAME} – MSDeploy 用户名。可在发布配置文件中找到 {USERNAME}。
- {PASSWORD} – MSDeploy 密码。从 {PROFILE}.PublishSettings 文件中获取 {PASSWORD}。可以从以下位置下载 .PublishSettings 文件：

- 解决方案资源管理器：选择“视图” > “Cloud Explorer”。连接你的 Azure 订阅。打开“应用服务”。右键单击应用。选择“下载发布配置文件”。
- Azure 门户：选择 Web 应用“概述”面板上的“获取发布配置文件”。

下面的示例使用名为“AzureWebApp - Web 部署”的发布配置文件：

```
msbuild "AzureWebApp.csproj"
    /p:DeployOnBuild=true
    /p:PublishProfile="AzureWebApp - Web Deploy"
    /p:Username="$AzureWebApp"
    /p:Password=".........."
```

此外，还可通过 Windows 命令提示符将发布配置文件与 .NET Core CLI dotnet msbuild 命令配合使用：

```
dotnet msbuild "AzureWebApp.csproj"
    /p:DeployOnBuild=true
    /p:PublishProfile="AzureWebApp - Web Deploy"
    /p:Username="$AzureWebApp"
    /p:Password=".........."
```

> **NOTE**
> 可跨平台使用 dotnet msbuild 命令，并且可在 macOS 和 Linux 上编译 ASP.NET Core 应用。但是，macOS 或 Linux 版 MSBuild 并不能将应用部署到 Azure 或其他 MSDeploy 终结点。MSDeploy 仅在 Windows 上可用。

排除文件

在发布 ASP.NET Core Web 应用时，生成项目和 wwwroot 文件夹的内容包括在内。`msbuild` 支持通配模式。例如，以下 `<Content>` 元素将从 wwwroot/content 文件夹及其所有子文件夹中排除所有文本 (.txt) 文件。

```
<ItemGroup>
  <Content Update="wwwroot/content/**/*.txt" CopyToPublishDirectory="Never" />
</ItemGroup>
```

可以将上面的标记添加到发布配置文件或 .csproj 文件。添加到 .csproj 文件时，会将该规则添加到项目中的所有发布配置文件中。

以下 `<MsDeploySkipRules>` 元素排除 wwwroot/content 文件夹中的所有文件：

```
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFolder">
    <ObjectName>dirPath</ObjectName>
    <AbsolutePath>wwwroot\\content</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

`<MsDeploySkipRules>` 不会从部署站点删除跳过目标。从部署站点中删除 `<Content>` 目标文件和文件夹。例如，假设部署的 Web 应用具有以下文件：

- *Views/Home/About1.cshtml*
- *Views/Home/About2.cshtml*
- *Views/Home/About3.cshtml*

如果添加了以下 `<MsDeploySkipRules>` 元素，则不会在部署站点上删除这些文件。

```
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About1.cshtml</AbsolutePath>
  </MsDeploySkipRules>

  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About2.cshtml</AbsolutePath>
  </MsDeploySkipRules>

  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About3.cshtml</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

之前的 `<MsDeploySkipRules>` 元素阻止部署跳过的文件。一旦部署这些文件就不会删除它们。

以下 `<Content>` 元素将在部署站点中删除目标文件：

```
<ItemGroup>
  <Content Update="Views/Home/About?.cshtml" CopyToPublishDirectory="Never" />
</ItemGroup>
```

使用具有上述 `<Content>` 元素的命令行部署可生成以下输出：

```
MSDeployPublish:
  Starting Web deployment task from source:
manifest(C:\Webs\Web1\obj\Release\netcoreapp1.1\PubTmp\Web1.SourceManifest.
  xml) to Destination: auto().
  Deleting file (Web11112\Views\Home\About1.cshtml).
  Deleting file (Web11112\Views\Home\About2.cshtml).
  Deleting file (Web11112\Views\Home\About3.cshtml).
  Updating file (Web11112\web.config).
  Updating file (Web11112\Web1.deps.json).
  Updating file (Web11112\Web1.dll).
  Updating file (Web11112\Web1.pdb).
  Updating file (Web11112\Web1.runtimeconfig.json).
  Successfully executed Web deployment task.
  Publish Succeeded.
Done Building Project "C:\Webs\Web1\Web1.csproj" (default targets).
```

包含文件

以下标记将项目目录外的 images 文件夹包含到发布站点的 wwwroot/images 文件夹中：

```
<ItemGroup>
  <_CustomFiles Include="$(MSBuildProjectDirectory)/../images/**/*" />
  <DotnetPublishFiles Include="@(_CustomFiles)">
    <DestinationRelativePath>wwwroot/images/%(RecursiveDir)%(Filename)%(Extension)</DestinationRelativePath>
  </DotnetPublishFiles>
</ItemGroup>
```

可以将标记添加到 .csproj 文件或发布配置文件。如果将其添加到 .csproj 文件，它会包含在项目的每个发布配置文件中。

以下突出显示的标记显示如何：

- 将文件从项目外部复制到 wwwroot 文件夹。
- 排除 wwwroot\Content 文件夹。
- 排除 Views\Home\About2.cshtml。

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This file is used by the publish/package process of your Web project.
You can customize the behavior of this process by editing this
MSBuild file.
-->
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <PublishProvider>FileSystem</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish />
    <LaunchSiteAfterPublish>True</LaunchSiteAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <PublishFramework />
    <ProjectGuid>afa9f185-7ce0-4935-9da1-ab676229d68a</ProjectGuid>
    <publishUrl>bin\Release\PublishOutput</publishUrl>
    <DeleteExistingFiles>False</DeleteExistingFiles>
  </PropertyGroup>
  <ItemGroup>
    <ResolvedFileToPublish Include="..\ReadMe2.MD">
      <RelativePath>wwwroot\ReadMe2.MD</RelativePath>
    </ResolvedFileToPublish>

    <Content Update="wwwroot\Content\**\*" CopyToPublishDirectory="Never" />
    <Content Update="Views\Home\About2.cshtml" CopyToPublishDirectory="Never" />

  </ItemGroup>
</Project>
```

请参阅 WebSDK 自述文件，了解更多部署示例。

在发布前或发布后运行目标

内置 `BeforePublish` 和 `AfterPublish` 目标在发布目标前/后执行目标。将以下元素添加到发布配置文件，以在发布前/后均记录控制台消息：

```
<Target Name="CustomActionsBeforePublish" BeforeTargets="BeforePublish">
    <Message Text="Inside BeforePublish" Importance="high" />
  </Target>
  <Target Name="CustomActionsAfterPublish" AfterTargets="AfterPublish">
    <Message Text="Inside AfterPublish" Importance="high" />
</Target>
```

使用不受信任的证书发布到服务器

将值为 `True` 的 `<AllowUntrustedCertificate>` 属性添加到发布配置文件：

```
<PropertyGroup>
  <AllowUntrustedCertificate>True</AllowUntrustedCertificate>
</PropertyGroup>
```

Kudu 服务

要查看 Azure 应用服务 Web 应用部署中的文件，请使用 Kudu 服务。将 `scm` 令牌追加到 Web 应用名称。例如:

URL	结果
`http://mysite.azurewebsites.net/`	Web 应用
`http://mysite.scm.azurewebsites.net/`	Kudu 服务

选择调试控制台菜单项来查看、编辑、删除或添加文件。

其他资源

- Web 部署 (MSDeploy) 简化了 Web 应用和网站到 IIS 服务器的部署。
- https://github.com/aspnet/websdk：文件问题和部署的请求功能。
- 从 Visual Studio 将 ASP.NET Web 应用发布到 Azure VM

ASP.NET Core 目录结构

2019/1/10 • • Edit Online

作者：Luke Latham

发布目录包含应用的可部署资产，由 dotnet publish 命令生成。该目录包含：

- 应用程序文件
- 配置文件
- 静态资产
- 包
- 运行时（仅限独立部署）

应用类型	目录结构

依赖框架的部署	• publish†
	  ◦ Logs†（除非需要接收 stdout 日志，否则为可选）
	  ◦ Views†（MVC 应用；如果未预编译视图）
	  ◦ Pages†（MVC 或 Razor 页应用；如果未预编译页）
	  ◦ wwwroot†
	  ◦ *.dll 文件
	  ◦ {ASSEMBLY NAME}.deps.json
	  ◦ {ASSEMBLY NAME}.dll
	  ◦ {ASSEMBLY NAME}.pdb
	  ◦ {ASSEMBLY NAME}.Views.dll
	  ◦ {ASSEMBLY NAME}.Views.pdb
	  ◦ {ASSEMBLY NAME}.runtimeconfig.json
	  ◦ web.config（IIS 部署）

独立部署	• publish†
	  ◦ Logs†（除非需要接收 stdout 日志，否则为可选）
	  ◦ Views†（MVC 应用；如果未预编译视图）
	  ◦ Pages†（MVC 或 Razor 页应用；如果未预编译页）
	  ◦ wwwroot†
	  ◦ *.dll 文件
	  ◦ {ASSEMBLY NAME}.deps.json
	  ◦ {ASSEMBLY NAME}.dll
	  ◦ {ASSEMBLY NAME}.exe
	  ◦ {ASSEMBLY NAME}.pdb
	  ◦ {ASSEMBLY NAME}.Views.dll
	  ◦ {ASSEMBLY NAME}.Views.pdb
	  ◦ {ASSEMBLY NAME}.runtimeconfig.json
	  ◦ web.config（IIS 部署）

†指示目录

publish 目录代表部署的内容根路径，也称为应用程序基路径。无论对服务器上已部署应用的 publish 目录如何命名，其位置都可作为托管应用的服务器物理路径。

wwwroot 目录(如果存在)仅包含静态资产。

可以使用以下两种方法之一为部署创建 stdout Logs 目录:

- 向项目添加以下 `<Target>` 元素:

  ```
  <Target Name="CreateLogsFolder" AfterTargets="Publish">
     <MakeDir Directories="$(PublishDir)Logs"
              Condition="!Exists('$(PublishDir)Logs')" />
     <WriteLinesToFile File="$(PublishDir)Logs\.log"
                       Lines="Generated file"
                       Overwrite="True"
                       Condition="!Exists('$(PublishDir)Logs\.log')" />
  </Target>
  ```

  `<MakeDir>` 元素在发布的输出中创建一个空的 Logs 文件夹。该元素使用 `PublishDir` 属性来确定创建文件夹的目标位置。几种部署方法(如 Web 部署)均在部署期间跳过空文件夹。`<WriteLinesToFile>` 元素在 Logs 文件夹中生成一个文件,该文件可确保将文件夹部署到服务器。如果工作进程不具有对目标文件夹的写入权限,则使用此方法的文件夹创建操作将失败。

- 在部署中的服务器上物理创建 Logs 目录。

部署目录需要读取/执行权限。Logs 目录需要读/写权限。将文件写入其他目录需要读/写权限。

其他资源

- dotnet publish
- .NET Core 应用程序部署
- 目标框架
- .NET Core RID 目录

Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考

2018/6/25 • • Edit Online

作者：Luke Latham

以下不是错误的完整列表。如果遇到此处未列出的错误，请打开一个新问题，并随附详细说明以重现错误。

收集以下信息：

- 浏览器行为
- 应用程序事件日志条目
- ASP.NET Core 模块 stdout 日志条目

将该信息与以下常见错误进行比较。如果找到匹配项，请按照故障排除建议进行操作。

> **IMPORTANT**
>
> **Azure 应用服务中的 ASP.NET Core 预览版**
>
> 默认情况下不会将 ASP.NET Core 预览版部署到 Azure 应用服务。要托管使用 ASP.NET Core 预览版的应用，请参阅将 ASP.NET Core 预览版部署到 Azure 应用服务。

安装程序无法获取 VC++ Redistributable

- **安装程序异常**：0x80072efd 或 0x80072f76 - 未指定的错误
- **安装程序日志异常†**：0x80072efd 或 0x80072f76 错误：未能执行 EXE 包

  †此日志位于 C:\Users\{USER}\AppData\Local\Temp\dd_DotNetCoreWinSvrHosting__{timestamp}.log。

疑难解答：

- 如果在安装托管捆绑包时系统无法访问 Internet，则在系统阻止安装程序获取 Microsoft Visual C++ 2015 Redistributable 时会出现此异常。可从 Microsoft 下载中心获取安装程序。如果安装程序失败，服务器可能不会收到托管依赖框架的部署 (FDD) 所需的 .NET Core 运行时。如果托管 FDD，请在“程序和功能”中确认安装了运行时。如有必要，可从 .NET All 下载获取运行时安装程序。安装运行时后，重启系统，或通过从命令提示符依次执行 net stop was /y 和 net start w3svc 来重启 IIS。

OS 升级删除了 32 位 ASP.NET Core 模块

- **应用程序日志**：未能加载模块 DLL C:\WINDOWS\system32\inetsrv\aspnetcore.dll。该数据是个错误。

疑难解答：

- OS 升级期间不会保留 C:\Windows\SysWOW64\inetsrv 目录中的非 OS 文件。如果在 OS 升级前已安装 ASP.NET Core 模块，且随后在 OS 升级后在 32 位模式下运行任何应用池，则会遇到此问题。在 OS 升级后修复 ASP.NET Core 模块。请参阅安装 .NET Core 托管捆绑包。运行安装程序时，选择“修复”。

平台与 RID 冲突

- **浏览器**：HTTP 错误 502.5 - 进程失败

- 应用程序日志：物理根路径为 'C:{PATH}' 的应用程序 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' 未能通过 '"C:\{PATH}{assembly}.{exe|dll}" ' 命令行启动进程，ErrorCode = '0x80004005 : ff。
- ASP.NET Core 模块日志：未经处理的异常: System.BadImageFormatException: 无法加载文件或程序集 '{assembly}.dll'。试图加载的程序的格式不正确。

疑难解答：

- 确认应用在 Kestrel 上本地运行。进程失败可能是由应用的内部问题导致的。有关详细信息，请参阅疑难解答。
- 确认 .csproj 中的 `<PlatformTarget>` 不与 RID 冲突。例如，请勿通过使用 dotnet publish -c Release -r win10-x64 或通过在 .csproj 中将 `<RuntimeIdentifiers>` 设置为 `win10-x64` 来指定 `x86` 的 `<PlatformTarget>` 和以 `win10-x64` 的 RID 进行发布。项目在不显示任何警告或错误的情况下发布，但会失败，且系统上出现上述记录的异常。
- 如果在升级应用和部署更新的程序集时，Azure 应用部署出现此异常，请从先前的部署中手动删除所有文件。部署升级的应用时，延迟的不兼容程序集可能造成 `System.BadImageFormatException` 异常。

URI 终结点错误或网站已停止

- **浏览器**：ERR_CONNECTION_REFUSED
- **应用程序日志**：没有任何条目
- **ASP.NET Core 模块日志**：未创建日志文件

疑难解答：

- 确认正在使用的应用的 URI 端点是否正确。检查绑定。
- 确认 IIS 网站未处于"已停止"状态。

已禁用 CoreWebEngine 或 W3SVC 服务器功能

- **OS 异常**：必须安装 IIS 7.0 CoreWebEngine 和 W3SVC 功能，以便使用 ASP.NET Core 模块。

疑难解答：

- 确认启用了适当的角色和功能。请参阅 IIS 配置。

网站物理路径不正确或缺少应用

- **浏览器**：403 禁止访问 - 访问被拒绝 --或者-- 403.14 禁止访问 - Web 服务器被配置为不列出此目录的内容。
- **应用程序日志**：没有任何条目
- **ASP.NET Core 模块日志**：未创建日志文件

疑难解答：

- 检查 IIS 网站的"基本设置"和物理应用文件夹。确认应用所在的文件夹位于 IIS 网站的物理路径中。

角色不正确、未安装模块或权限不正确

- **浏览器**：500.19 内部服务器错误 - 无法访问请求的页面，因为该页面的相关配置数据无效。
- **应用程序日志**：没有任何条目

- **ASP.NET Core 模块日志**：未创建日志文件

疑难解答：

- 确认启用了适当的角色。请参阅 IIS 配置。
- 检查“程序和功能”，确认已安装Microsoft ASP.NET Core 模块。如果已安装程序列表中没有 Microsoft ASP.NET Core 模块，请安装该模块。请参阅安装 .NET Core 托管捆绑包。
- 请确保将“应用程序池” > “进程模型” > “标识”设置为 ApplicationPoolIdentity，或确保自定义标识具有访问应用部署文件夹的相应权限。

processPath 不正确、缺少 PATH 变量、未安装托管捆绑包、未重启系统/IIS、未安装 VC++ Redistributable 或 dotnet.exe 访问冲突

- **浏览器**：HTTP 错误 502.5 - 进程失败
- 应用程序日志：物理根路径为 'C:\{PATH}' 的应用程序 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' 未能通过 '".{assembly}.exe" ' 命令行启动进程，ErrorCode = '0x80070002 : 0。
- **ASP.NET Core 模块日志**：已创建日志文件，但该日志文件为空

疑难解答：

- 确认应用在 Kestrel 上本地运行。进程失败可能是由应用的内部问题导致的。有关详细信息，请参阅疑难解答。
- 检查 web.config 中 `<aspNetCore>` 元素的 processPath 属性，对于依赖框架的部署 (FDD)，确保它为 dotnet，对于独立部署 (SCD)，确保它为 {assembly}.exe。
- 对于 FDD，可能无法通过路径设置访问 dotnet.exe。确认系统路径设置中存在 *C:\Program Files\dotnet*。
- 对于 FDD，应用程序池的用户标识可能无法访问 dotnet.exe。确认应用程序池用户标识具有访问 C:\Program Files\dotnet 目录的权限。确认没有为应用池用户标识配置针对 C:\Program Files\dotnet 和应用目录的拒绝规则。
- 可能已部署 FDD 并安装了 .NET Core，但未重启 IIS。重启服务器，或通过从命令提示符依次执行 net stop was /y 和 net start w3svc 来重启 IIS。
- 可能已部署 FDD，但未在托管系统上安装 .NET Core 运行时。如果尚未安装 .NET Core 运行时，则运行系统上的 **.NET Core 托管捆绑包安装程序**。请参阅安装 .NET Core 托管捆绑包。如果在没有 Internet 连接的情况下尝试在系统上安装 .NET Core 运行时，请从 .NET All 下载获取运行时并运行托管捆绑包安装程序，以安装 ASP.NET Core 模块。重启系统，或通过从命令提示符依次执行 net stop was /y 和 net start w3svc 来重启 IIS，完成安装。
- 可能已部署 FDD，但系统上未安装 Microsoft Visual C++ 2015 Redistributable (x64)。可从 Microsoft 下载中心获取安装程序。

<aspNetCore> 元素的参数不正确

- **浏览器**：HTTP 错误 502.5 - 进程失败
- 应用程序日志：物理根路径为 'C:\{PATH}' 的应用程序 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' 未能通过 '"dotnet" .{assembly}.dll' 命令行启动进程，ErrorCode = '0x80004005 : 80008081。
- ASP.NET Core 模块日志：要执行的应用程序不存在: 'PATH{assembly}.dll'

疑难解答：

- 确认应用在 Kestrel 上本地运行。进程失败可能是由应用的内部问题导致的。有关详细信息，请参阅疑难解答。
- 检查 web.config 中 `<aspNetCore>` 元素的 arguments 属性，确认该属性：(a) 对于依赖框架的部署 (FDD) 为 {assembly}.dll；或 (b) 对于独立部署 (SCD)，则为不存在、为空字符串 (arguments=""), 或为应用参数列表 (arguments="arg1, arg2, ...")。

缺少 .NET Framework 版本

- **浏览器**：502.3 错误网关 - 尝试路由请求时出现连接错误。
- 应用程序日志：ErrorCode = 其物理根路径为 'C:\{PATH}' 的应用程序 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' 未能通过 '"dotnet" .{assembly}.dll' 命令行启动进程，ErrorCode = '0x80004005 : 80008081。
- **ASP.NET Core 模块日志**：缺少方法、文件或程序集异常。在异常中指定的方法、文件或程序集是 .NET Framework 方法、文件或程序集。

疑难解答：

- 安装系统缺少的 .NET Framework 版本。
- 对于依赖框架的部署 (FDD)，确认已在系统上安装正确的运行时。如果项目从 1.1 升级到 2.0、部署到托管系统，并且导致此异常，请确保 2.0 框架位于托管系统上。

应用程序池已停止

- **浏览器**：503 服务不可用
- **应用程序日志**：没有任何条目
- **ASP.NET Core 模块日志**：未创建日志文件

疑难解答

- 确认应用程序池未处于“已停止”状态。

未实现 IIS 集成中间件

- **浏览器**：HTTP 错误 502.5 - 进程失败
- 应用程序日志：物理根路径为 'C:\{PATH}' 的应用程序 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' 通过 '"C:\{PATH}{assembly}.{exe|dll}" ' 命令行创建了进程，但该进程出现故障、未响应或未在给定的端口 '{PORT}' 上侦听，ErrorCode = '0x800705b4'
- **ASP.NET Core 模块日志**：已创建日志文件，且该日志文件显示正常操作。

疑难解答

- 确认应用在 Kestrel 上本地运行。进程失败可能是由应用的内部问题导致的。有关详细信息，请参阅疑难解答。
- 确认以下项之一：
  - 通过调用应用的 `WebHostBuilder` 上的 `UseIISIntegration` 方法来引用 IIS 集成中间件 (ASP.NET Core 1.x)
  - 应用使用 `CreateDefaultBuilder` 方法 (ASP.NET Core 2.x)。

  有关详细信息，请参阅在 ASP.NET Core 中托管。

子应用程序包括 <handlers> 部分

- **浏览器**：HTTP 错误 500.19 - 内部服务器错误
- **应用程序日志**：没有任何条目
- **ASP.NET Core 模块日志**：已创建日志文件，且该日志文件显示根应用程序的正常操作。没有为子应用程序创建日志文件。

疑难解答

- 确认子应用的 web.config 文件不包括 `<handlers>` 部分。

stdout 日志路径不正确

- 浏览器：应用程序正常响应。
- 应用程序日志：警告: 无法创建 stdoutLogFile \?\C:_apps\app_folder\bin\Release\netcoreapp2.0\win10-x64\publish\logs\path_doesnt_exist\stdout_8748_201831835937.log, ErrorCode = -2147024893。
- **ASP.NET Core 模块日志**：未创建日志文件

疑难解答

- web.config 的 `<aspNetCore>` 元素中指定的 `stdoutLogFile` 路径不存在。有关详细信息，请参阅 ASP.NET Core 模块配置参考主题的日志创建和重定向部分。

应用程序配置常见问题

- **浏览器**：HTTP 错误 502.5 - 进程失败
- 应用程序日志：物理根路径为 'C:\{PATH}' 的应用程序 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' 通过 '"C:\{PATH}{assembly}.{exe|dll}" ' 命令行创建了进程，但该进程出现故障、未响应或未在给定的端口 '{PORT}' 上侦听，ErrorCode = '0x800705b4'
- **ASP.NET Core 模块日志**：已创建日志文件，但该日志文件为空

疑难解答

- 此一般异常表明进程未能启动，这很可能是由应用配置问题引起的。请参阅目录结构，确认应用的已部署文件和文件夹适当，应用的配置文件存在且包含适用于应用和环境的正确设置。有关详细信息，请参阅疑难解答。

ASP.NET Core 中的运行状况检查

2019/1/10 • • Edit Online

通过 Luke Latham 和 Glenn Condron

ASP.NET Core 提供运行状况检查中间件和库，以用于报告应用基础结构组件的运行状况。

运行状况检查由应用程序作为 HTTP 终结点公开。可以为各种实时监视方案配置运行状况检查终结点：

- 运行状况探测可以由容器业务流程协调程和负载均衡器用于检查应用的状态。例如，容器业务流程协调程序可以通过停止滚动部署或重新启动容器来响应失败的运行状况检查。负载均衡器可以通过将流量从失败的实例路由到正常实例，来应对不正常的应用。
- 可以监视内存、磁盘和其他物理服务器资源的使用情况来了解是否处于正常状态。
- 运行状况检查可以测试应用的依赖项（如数据库和外部服务终结点）以确认是否可用和正常工作。

查看或下载示例代码（如何下载）

示例应用包含本主题中所述的方案示例。若要运行给定方案的示例应用，请在命令行界面中从项目文件夹中使用 dotnet run 命令。请参阅示例应用的 README.md 文件和本主题中的方案说明，以了解有关如何使用示例应用的详细信息。

系统必备

运行状况检查通常与外部监视服务或容器业务流程协调程序一起用于检查应用的状态。向应用添加运行状况检查之前，需确定要使用的监视系统。监视系统决定了要创建的运行状况检查类型以及配置其终结点的方式。

引用 Microsoft.AspNetCore.App 元包或将包引用添加到 Microsoft.AspNetCore.Diagnostics.HealthChecks 包。

示例应用提供了启动代码来演示几个方案的运行状况检查。数据库探测方案使用 BeatPulse 检查数据库连接的运行状况。DbContext 探测方案使用 EF Core `DbContext` 检查数据库。若要探索数据库方案，示例应用将：

- 创建一个数据库，并在 appsettings.json 文件中提供其连接字符串。
- 其项目文件中具有以下包引用：
  - AspNetCore.HealthChecks.SqlServer
  - Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore

> **NOTE**
>
> BeatPulse 不由 Microsoft 进行支持或维护。

另一个运行状况检查方案演示如何将运行状况检查筛选到某个管理端口。示例应用要求创建包含管理 URL 和管理端口的 Properties/launchSettings.json 文件。有关详细信息，请参阅按端口筛选部分。

基本运行状况探测

对于许多应用，报告应用在处理请求方面的可用性（运行情况）的基本运行状况探测配置足以发现应用的状态。

基本配置会注册运行状况检查服务，并调用运行状况检查中间件以使用运行状况响应在 URL 终结点处进行响应。默认情况下，不会注册任何特定运行状况检查来测试任何特定依赖项或子系统。如果能够在运行状况终结点 URL 处进行响应，则应用被视为正常。默认响应编写器会以纯文本响应形式将状态（`HealthStatus`）写回到客户端，以便指示 `HealthStatus.Healthy`、`HealthStatus.Degraded` 或 `HealthStatus.Unhealthy` 状态。

在 `Startup.ConfigureServices` 中使用 `AddHealthChecks` 注册运行状况检查服务。在 `Startup.Configure` 的请求处理管道中使用 `UseHealthChecks` 注册运行状况检查中间件。

在示例应用中，在 `/health` 处创建运行状况检查终结点 (BasicStartup.cs)：

```
public class BasicStartup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddHealthChecks();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseHealthChecks("/health");

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                "Navigate to /health to see the health status.");
        });
    }
}
```

若要使用示例应用运行基本配置方案，请在命令行界面中从项目文件夹执行以下命令：

```
dotnet run --scenario basic
```

Docker 示例

Docker 提供内置 `HEALTHCHECK` 指令，该指令可以用于检查使用基本运行状况检查配置的应用的状态：

```
HEALTHCHECK CMD curl --fail http://localhost:5000/health || exit
```

创建运行状况检查

运行状况检查通过实现 `IHealthCheck` 接口进行创建。`IHealthCheck.CheckHealthAsync` 方法会返回 `Task<HealthCheckResult>`，它以 `Healthy`、`Degraded` 或 `Unhealthy` 的形式指示运行状况。结果会使用可配置状态代码（配置在运行状况检查选项部分中进行介绍）编写为纯文本响应。`HealthCheckResult` 还可以返回可选的键值对。

示例运行状况检查

下面的 `ExampleHealthCheck` 类演示运行状况检查的布局：

```
public class ExampleHealthCheck : IHealthCheck
{
    public ExampleHealthCheck()
    {
        // Use dependency injection (DI) to supply any required services to the
        // health check.
    }

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        // Execute health check logic here. This example sets a dummy
        // variable to true.
        var healthCheckResultHealthy = true;

        if (healthCheckResultHealthy)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("The check indicates a healthy result."));
        }

        return Task.FromResult(
            HealthCheckResult.Unhealthy("The check indicates an unhealthy result."));
    }
}
```

注册运行状况检查服务

`ExampleHealthCheck` 类型使用 `AddCheck` 添加到运行状况检查服务：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddCheck<ExampleHealthCheck>("example_health_check");
}
```

以下示例中显示的 `AddCheck` 重载会设置要在运行状况检查报告失败时报告的失败状态（`HealthStatus`）。如果失败状态设置为 `null`（默认值），则会报告 `HealthStatus.Unhealthy`。此重载对于库创建者是一种十分有用的方案，在这种情况下，如果运行状况检查实现遵循该设置，则在发生运行状况检查失败时，应用会强制实施库所指示的失败状态。

标记用于筛选运行状况检查（在筛选运行状况检查部分中进行了进一步介绍）。

```
services.AddHealthChecks()
    .AddCheck<ExampleHealthCheck>(
        "example_health_check",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "example" });
```

`AddCheck` 还可以执行 lambda 函数。在以下示例中，运行状况检查名称指定为 `Example`，并且检查始终返回正常状态：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddCheck("Example", () =>
            HealthCheckResult.Healthy("Example is OK!"), tags: new[] { "example" })
}
```

使用运行状况检查中间件

在 `Startup.Configure` 内，在处理管道中使用终结点 URL 或相对路径调用 `UseHealthChecks`：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHealthChecks("/health");
}
```

如果运行状况检查应侦听特定端口，则使用 `UseHealthChecks` 的重载设置端口（在按端口筛选部分中进行了进一步介绍）：

```
app.UseHealthChecks("/health", port: 8000);
```

运行状况检查中间件是应用请求处理管道中的*终端中间件*。遇到的与请求 URL 完全匹配的第一个运行状况检查终结点会进行执行，并且使中间件管道的其余部分短路。当短路发生时，不会执行匹配运行状况检查后面的任何中间件。

运行状况检查选项

`HealthCheckOptions` 使你可以自定义运行状况检查行为：

- 筛选运行状况检查
- 自定义 HTTP 状态代码
- 取消缓存标头
- 自定义输出

筛选运行状况检查

默认情况下，运行状况检查中间件会运行所有已注册的运行状况检查。若要运行运行状况检查的子集，请提供向 `Predicate` 选项返回布尔值的函数。在以下示例中，`Bar` 运行状况检查在函数条件语句中由于其标记（`bar_tag`）而被筛选掉，在条件语句中，仅当运行状况检查的 `Tag` 属性与 `foo_tag` 或 `baz_tag` 匹配时才返回 `true`：

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Diagnostics.HealthChecks;
using Microsoft.Extensions.Diagnostics.HealthChecks;

public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddCheck("Foo", () =>
            HealthCheckResult.Healthy("Foo is OK!"), tags: new[] { "foo_tag" })
        .AddCheck("Bar", () =>
            HealthCheckResult.Unhealthy("Bar is unhealthy!"), tags: new[] { "bar_tag" })
        .AddCheck("Baz", () =>
            HealthCheckResult.Healthy("Baz is OK!"), tags: new[] { "baz_tag" });
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHealthChecks("/health", new HealthCheckOptions()
    {
        // Filter out the 'Bar' health check. Only Foo and Baz execute.
        Predicate = (check) => check.Tags.Contains("foo_tag") ||
            check.Tags.Contains("baz_tag")
    });
}
```

自定义 HTTP 状态代码

使用 `ResultStatusCodes` 可自定义运行状况状态到 HTTP 状态代码的映射。以下 `StatusCode` 分配是中间件所使用的默认值。更改状态代码值以满足要求。

```
using Microsoft.AspNetCore.Diagnostics.HealthChecks;
using Microsoft.Extensions.Diagnostics.HealthChecks;

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHealthChecks("/health", new HealthCheckOptions()
    {
        // The following StatusCodes are the default assignments for
        // the HealthStatus properties.
        ResultStatusCodes =
        {
            [HealthStatus.Healthy] = StatusCodes.Status200OK,
            [HealthStatus.Degraded] = StatusCodes.Status200OK,
            [HealthStatus.Unhealthy] = StatusCodes.Status503ServiceUnavailable
        }
    });
}
```

取消缓存标头

`AllowCachingResponses` 控制运行状况检查中间件是否将 HTTP 标头添加到探测响应以防止响应缓存。如果值为 `false`（默认值），则中间件会设置或替代 `Cache-Control`、`Expires` 和 `Pragma` 标头以防止响应缓存。如果值为 `true`，则中间件不会修改响应的缓存标头。

```
using Microsoft.AspNetCore.Diagnostics.HealthChecks;
using Microsoft.Extensions.Diagnostics.HealthChecks;

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHealthChecks("/health", new HealthCheckOptions()
    {
        // The default value is false.
        AllowCachingResponses = false
    });
}
```

自定义输出

`ResponseWriter` 选项可获取或设置用于编写响应的委托。默认委托会使用 `HealthReport.Status` 字符串值编写最小的纯文本响应。

```
using Microsoft.AspNetCore.Diagnostics.HealthChecks;
using Microsoft.Extensions.Diagnostics.HealthChecks;

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHealthChecks("/health", new HealthCheckOptions()
    {
        // WriteResponse is a delegate used to write the response.
        ResponseWriter = WriteResponse
    });
}

private static Task WriteResponse(HttpContext httpContext,
    HealthReport result)
{
    httpContext.Response.ContentType = "application/json";

    var json = new JObject(
        new JProperty("status", result.Status.ToString()),
        new JProperty("results", new JObject(result.Entries.Select(pair =>
            new JProperty(pair.Key, new JObject(
                new JProperty("status", pair.Value.Status.ToString()),
                new JProperty("description", pair.Value.Description),
                new JProperty("data", new JObject(pair.Value.Data.Select(
                    p => new JProperty(p.Key, p.Value))))))))));
    return httpContext.Response.WriteAsync(
        json.ToString(Formatting.Indented));
}
```

数据库探测

运行状况检查可以指定数据库查询作为布尔测试来运行，以指示数据库是否在正常响应。

示例应用使用 BeatPulse（适用于 ASP.NET Core 应用的运行状况检查库）对 SQL Server 数据库执行运行状况检查。BeatPulse 会对数据库执行 `SELECT 1` 查询以确认数据库连接是否处于正常状态。

> **WARNING**
>
> 使用查询检查数据库连接时，请选择快速返回的查询。查询方法会面临使数据库过载和降低其性能的风险。在大多数情况下，无需运行测试查询。只需建立成功的数据库连接便足矣。如果发现需要运行查询，请选择简单的 SELECT 查询，如 `SELECT 1`。

若要使用 BeatPulse 库，请包含对 AspNetCore.HealthChecks.SqlServer 的包引用。

在应用的 appsettings.json 文件中提供有效数据库连接字符串。应用使用名为 `HealthCheckSample` 的 SQL Server 数据库：

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\MSSQLLocalDB;Database=HealthCheckSample;Trusted_Connection=True;MultipleActiveResultSets=true;Conn
ectRetryCount=0"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Debug"
    },
    "Console": {
      "IncludeScopes": "true"
    }
  }
}
```

在 `Startup.ConfigureServices` 中使用 `AddHealthChecks` 注册运行状况检查服务。示例应用使用数据库连接字符串 (DbHealthStartup.cs) 调用 BeatPulse 的 `AddSqlServer` 方法：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddSqlServer(Configuration["ConnectionStrings:DefaultConnection"]);
}
```

在 `Startup.Configure` 内，在应用处理管道中调用运行状况检查中间件：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHealthChecks("/health");
```

若要使用示例应用运行数据库探测方案，请在命令行界面中从项目文件夹执行以下命令：

```
dotnet run --scenario db
```

> **NOTE**
>
> BeatPulse 不由 Microsoft 进行支持或维护。

Entity Framework Core DbContext 探测

`DbContext` 检查确认应用可以与为 EF Core `DbContext` 配置的数据库通信。满足以下条件的应用支持 `DbContext` 检查：

- 使用 Entity Framework (EF) Core。
- 包括对 Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore 的包引用。

`AddDbContextCheck<TContext>` 为 `DbContext` 注册运行状况检查。`DbContext` 作为方法的 `TContext` 提供。重载可用于配置失败状态、标记和自定义测试查询。

默认情况下：

- `DbContextHealthCheck` 调用 EF Core 的 `CanConnectAsync` 方法。可以自定义在使用 `AddDbContextCheck` 方法重载检查运行状况时运行的操作。
- 运行状况检查的名称是 `TContext` 类型的名称。

在示例应用中，`AppDbContext` 会提供给 `AddDbContextCheck`，并在 `Startup.ConfigureServices` 中注册为服务。

DbContextHealthStartup.cs：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddDbContextCheck<AppDbContext>();

    services.AddDbContext<AppDbContext>(options =>
    {
        options.UseSqlServer(
            Configuration["ConnectionStrings:DefaultConnection"]);
    });
}
```

在示例应用中，`UseHealthChecks` 在 `Startup.Configure` 中添加运行状况检查中间件。

DbContextHealthStartup.cs：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHealthChecks("/health");
```

若要使用示例应用运行 `DbContext` 探测方案，请确认连接字符串指定的数据库在 SQL Server 实例中不存在。如果该数据库存在，请删除它。

在命令行界面中从项目文件夹执行以下命令：

```
dotnet run --scenario dbcontext
```

在应用运行之后，在浏览器中对 `/health` 终结点发出请求，从而检查运行状况。数据库和 `AppDbContext` 不存在，因此应用提供以下响应：

```
Unhealthy
```

触发示例应用以创建数据库。向 `/createdatabase` 发出请求。应用会进行以下响应：

```
Creating the database...
Done!
Navigate to /health to see the health status.
```

向 `/health` 终结点发出请求。数据库和上下文存在，因此应用会进行以下响应：

```
Healthy
```

触发示例应用以删除数据库。向 `/deletedatabase` 发出请求。应用会进行以下响应：

```
Deleting the database...
Done!
Navigate to /health to see the health status.
```

向 `/health` 终结点发出请求。应用提供不正常的响应：

```
Unhealthy
```

单独的就绪情况和运行情况探测

在某些托管方案中，会使用一对区分两种应用状态的运行状况检查：

- 应用正常运行，但尚未准备好接收请求。此状态是应用的就绪情况。
- 应用正常运行并响应请求。此状态是应用的运行情况。

就绪情况检查通常执行更广泛和耗时的检查集，以确定应用的所有子系统和资源是否都可用。运行情况检查只是执行一个快速检查，以确定应用是否可用于处理请求。应用通过其就绪情况检查之后，无需使用成本高昂的就绪情况检查集来进一步增加应用负荷 — 后续检查只需检查运行情况。

示例应用包含运行状况检查，以报告托管服务中长时间运行的启动任务的完成。`StartupHostedServiceHealthCheck` 公开了属性 `StartupTaskCompleted`，托管服务在其长时间运行的任务完成时可以将该属性设置为 `true` (StartupHostedServiceHealthCheck.cs)：

```
public class StartupHostedServiceHealthCheck : IHealthCheck
{
    public string Name => "slow_dependency_check";

    public bool StartupTaskCompleted { get; set; } = false;

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        if (StartupTaskCompleted)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("The startup task is finished."));
        }

        return Task.FromResult(
            HealthCheckResult.Unhealthy("The startup task is still running."));
    }
}
```

长时间运行的后台任务由托管服务 (Services/StartupHostedService) 启动。在该任务结束时，`StartupHostedServiceHealthCheck.StartupTaskCompleted` 设置为 `true`：

```
public class StartupHostedService : IHostedService, IDisposable
{
    private readonly int _delaySeconds = 15;
    private readonly ILogger _logger;
    private readonly StartupHostedServiceHealthCheck _startupHostedServiceHealthCheck;

    public StartupHostedService(ILogger<StartupHostedService> logger,
        StartupHostedServiceHealthCheck startupHostedServiceHealthCheck)
    {
        _logger = logger;
        _startupHostedServiceHealthCheck = startupHostedServiceHealthCheck;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation($"Startup Background Service is starting.");

        // Simulate the effect of a long-running startup task.
        Task.Run(async () =>
        {
            await Task.Delay(_delaySeconds * 1000);

            _startupHostedServiceHealthCheck.StartupTaskCompleted = true;

            _logger.LogInformation($"Startup Background Service has started.");
        });

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("Startup Background Service is stopping.");

        return Task.CompletedTask;
    }

    public void Dispose()
    {
    }
}
```

运行状况检查在 `Startup.ConfigureServices` 中使用 `AddCheck` 与托管服务一起注册。因为托管服务必须对运行状况检查设置该属性，所以运行状况检查也会在服务容器 (LivenessProbeStartup.cs) 中进行注册：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHostedService<StartupHostedService>();
    services.AddSingleton<StartupHostedServiceHealthCheck>();

    services.AddHealthChecks()
        .AddCheck<StartupHostedServiceHealthCheck>(
            "hosted_service_startup",
            failureStatus: HealthStatus.Degraded,
            tags: new[] { "ready" });
}
```

在 `Startup.Configure` 内，在应用处理管道中调用运行状况检查中间件。在示例应用中，在 `/health/ready` 处为就绪情况检查，并且在 `/health/live` 处为运行情况检查创建运行状况检查终结点。就绪情况检查会将运行状况检查筛选到具有 `ready` 标记的运行状况检查。运行情况检查通过在 `HealthCheckOptions.Predicate` 中返回 `false` 来筛选出 `StartupHostedServiceHealthCheck`（有关详细信息，请参阅筛选运行状况检查）：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    // The readiness check uses all registered checks with the 'ready' tag.
    app.UseHealthChecks("/health/ready", new HealthCheckOptions()
    {
        Predicate = (check) => check.Tags.Contains("ready"),
    });

    app.UseHealthChecks("/health/live", new HealthCheckOptions()
    {
        // Exclude all checks and return a 200-Ok.
        Predicate = (_) => false
    });
```

若要使用示例应用运行就绪情况/运行情况配置方案，请在命令行界面中从项目文件夹执行以下命令：

```
dotnet run --scenario liveness
```

在浏览器中，访问 `/health/ready` 几次，直到过了 15 秒。运行状况检查会在前 15 秒内报告 `Unhealthy`。15 秒之后，终结点会报告 `Healthy`，这反映托管服务完成了长时间运行的任务。

Kubernetes 示例

在诸如 Kubernetes 这类环境中，使用单独的就绪情况和运行情况检查会十分有用。在 Kubernetes 中，应用可能需要在接受请求之前执行耗时的启动工作，如基础数据库可用性测试。使用单独检查使业务流程协调程序可以区分应用是否正常运行但尚未准备就绪，或是应用程序是否未能启动。有关 Kubernetes 中的就绪情况和运行情况探测的详细信息，请参阅 Kubernetes 文档中的配置运行情况和就绪情况探测。

以下示例演示如何使用 Kubernetes 就绪情况探测配置：

```
spec:
  template:
  spec:
    readinessProbe:
      # an http probe
      httpGet:
        path: /health/ready
        port: 80
      # length of time to wait for a pod to initialize
      # after pod startup, before applying health checking
      initialDelaySeconds: 30
      timeoutSeconds: 1
    ports:
      - containerPort: 80
```

具有自定义响应编写器的基于指标的探测

示例应用演示具有自定义响应编写器的内存运行状况检查。

如果应用使用的内存多于给定内存阈值（在示例应用中为 1 GB），则 `MemoryHealthCheck` 报告降级状态。`HealthCheckResult` 包括应用的垃圾回收器 (GC) 信息 (MemoryHealthCheck.cs)：

```
public class MemoryHealthCheck : IHealthCheck
{
    private readonly IOptionsMonitor<MemoryCheckOptions> _options;

    public MemoryHealthCheck(IOptionsMonitor<MemoryCheckOptions> options)
    {
        _options = options;
    }

    public string Name => "memory_check";

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        var options = _options.Get(context.Registration.Name);

        // Include GC information in the reported diagnostics.
        var allocated = GC.GetTotalMemory(forceFullCollection: false);
        var data = new Dictionary<string, object>()
        {
            { "AllocatedBytes", allocated },
            { "Gen0Collections", GC.CollectionCount(0) },
            { "Gen1Collections", GC.CollectionCount(1) },
            { "Gen2Collections", GC.CollectionCount(2) },
        };

        var status = (allocated < options.Threshold) ?
            HealthStatus.Healthy : HealthStatus.Unhealthy;

        return Task.FromResult(new HealthCheckResult(
            status,
            description: "Reports degraded status if allocated bytes " +
                $">= {options.Threshold} bytes.",
            exception: null,
            data: data));
    }
}
```

在 `Startup.ConfigureServices` 中使用 `AddHealthChecks` 注册运行状况检查服务。`MemoryHealthCheck` 注册为服务，而不是通过将运行状况检查传递到 `AddCheck` 来启用它。所有 `IHealthCheck` 注册服务都可供运行状况检查服务和中间件使用。建议将运行状况检查服务注册为单一实例服务。

CustomWriterStartup.cs：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddMemoryHealthCheck("memory");
}
```

在 `Startup.Configure` 内，在应用处理管道中调用运行状况检查中间件。一个 `WriteResponse` 委托提供给 `ResponseWriter` 属性，以在执行运行状况检查时输出自定义 JSON 响应：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHealthChecks("/health", new HealthCheckOptions()
    {
        // This custom writer formats the detailed status as JSON.
        ResponseWriter = WriteResponse
    });
```

`WriteResponse` 方法将 `CompositeHealthCheckResult` 格式化为 JSON 对象，并生成运行状况检查响应的 JSON 输出：

```
private static Task WriteResponse(HttpContext httpContext,
    HealthReport result)
{
    httpContext.Response.ContentType = "application/json";

    var json = new JObject(
        new JProperty("status", result.Status.ToString()),
        new JProperty("results", new JObject(result.Entries.Select(pair =>
            new JProperty(pair.Key, new JObject(
                new JProperty("status", pair.Value.Status.ToString()),
                new JProperty("description", pair.Value.Description),
                new JProperty("data", new JObject(pair.Value.Data.Select(
                    p => new JProperty(p.Key, p.Value))))))))));
    return httpContext.Response.WriteAsync(
        json.ToString(Formatting.Indented));
}
```

若要使用示例应用运行具有自定义响应编写器输出的基于指标的探测，请在命令行界面中从项目文件夹执行以下命令：

```
dotnet run --scenario writer
```

> **NOTE**
>
> BeatPulse 包括基于指标的运行状况检查方案（包括磁盘存储和最大值运行情况检查）。
>
> BeatPulse 不由 Microsoft 进行支持或维护。

按端口筛选

使用端口调用 `UseHealthChecks` 会将运行状况检查请求限制到指定端口。这通常用于在容器环境中公开用于监视服务的端口。

示例应用使用环境变量配置提供程序配置端口。端口在 launchSettings.json 文件设置，并通过环境变量传递到配置提供程序。还必须配置服务器以在管理端口上侦听请求。

若要使用示例应用演示管理端口配置，请在 Properties 文件夹中创建 launchSettings.json 文件。

以下 launchSettings.json 文件未包含在示例应用的项目文件中，必须手动创建。

Properties/launchSettings.json：

```
{
  "profiles": {
    "SampleApp": {
      "commandName": "Project",
      "commandLineArgs": "",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_URLS": "http://localhost:5000/;http://localhost:5001/",
        "ASPNETCORE_MANAGEMENTPORT": "5001"
      },
      "applicationUrl": "http://localhost:5000/"
    }
  }
}
```

在 `Startup.ConfigureServices` 中使用 `AddHealthChecks` 注册运行状况检查服务。`UseHealthChecks` 调用指定管理端口 (ManagementPortStartup.cs)：

```
public class ManagementPortStartup
{
    public ManagementPortStartup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddHealthChecks();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {

        app.UseHealthChecks("/health", port: Configuration["ManagementPort"]);

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                "Navigate to " +
                $"http://localhost:{Configuration["ManagementPort"]}/health " +
                "to see the health status.");
        });
    }
}
```

NOTE

可以通过在代码中显式设置 URL 和管理端口，来避免在示例应用中创建 launchSettings.json 文件。在创建 `WebHostBuilder` 的 Program.cs 中，添加 `UseUrls` 调用并提供应用的正常响应终结点和管理端口终结点。在调用 `UseHealthChecks` 的 ManagementPortStartup.cs 中，显式指定管理端口。

Program.cs:

```
return new WebHostBuilder()
    .UseConfiguration(config)
    .UseUrls("http://localhost:5000/;http://localhost:5001/")
    .ConfigureLogging(builder =>
    {
        builder.SetMinimumLevel(LogLevel.Trace);
        builder.AddConfiguration(config);
        builder.AddConsole();
    })
    .UseKestrel()
    .UseStartup(startupType)
    .Build();
```

ManagementPortStartup.cs：

```
app.UseHealthChecks("/health", port: 5001);
```

若要使用示例应用运行管理端口配置方案，请在命令行界面中从项目文件夹执行以下命令：

```
dotnet run --scenario port
```

分发运行状况检查库

将运行状况检查作为库进行分发：

1. 编写将 `IHealthCheck` 接口作为独立类来实现的运行状况检查。该类可以依赖于依赖关系注入 (DI)类型激活和命名选项来访问配置数据。

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Diagnostics.HealthChecks;

namespace SampleApp
{
    public class ExampleHealthCheck : IHealthCheck
    {
        private readonly string _data1;
        private readonly int? _data2;

        public ExampleHealthCheck(string data1, int? data2)
        {
            _data1 = data1 ?? throw new ArgumentNullException(nameof(data1));
            _data2 = data2 ?? throw new ArgumentNullException(nameof(data2));
        }

        public async Task<HealthCheckResult> CheckHealthAsync(
            HealthCheckContext context, CancellationToken cancellationToken)
        {
            try
            {
                // Health check logic
                //
                // data1 and data2 are used in the method to
                // run the probe's health check logic.

                // Assume that it's possible for this health check
                // to throw an AccessViolationException.

                return HealthCheckResult.Healthy();
            }
            catch (AccessViolationException ex)
            {
                return new HealthCheckResult(
                    context.Registration.FailureStatus,
                    description: "An access violation occurred during the check.",
                    exception: ex,
                    data: null);
            }
        }
    }
}
```

2. 使用参数编写一个扩展方法，所使用的应用会在其 `Startup.Configure` 方法中调用它。在以下示例中，假设以下运行状况检查方法签名：

```
ExampleHealthCheck(string, string, int )
```

前面的签名指示 `ExampleHealthCheck` 需要其他数据来处理运行状况检查探测逻辑。当运行状况检查向扩展方法注册时，数据会提供给用于创建运行状况检查实例的委托。在以下示例中，调用方会指定可选的：

- 运行状况检查名称（`name`）。如果为 `null`，则使用 `example_health_check`。
- 运行状况检查的字符串数据点（`data1`）。
- 运行状况检查的整数数据点（`data2`）。如果为 `null`，则使用 `1`。
- 失败状态（`HealthStatus`）。默认值为 `null`。如果为 `null`，则对失败状态报告 `HealthStatus.Unhealthy`。
- 标记（`IEnumerable<string>`）。

```
using System.Collections.Generic;
using Microsoft.Extensions.Diagnostics.HealthChecks;

public static class ExampleHealthCheckBuilderExtensions
{
    const string NAME = "example_health_check";

    public static IHealthChecksBuilder AddExampleHealthCheck(
        this IHealthChecksBuilder builder,
        string name = default,
        string data1,
        int data2 = 1,
        HealthStatus? failureStatus = default,
        IEnumerable<string> tags = default)
    {
        return builder.Add(new HealthCheckRegistration(
            name ?? NAME,
            sp => new ExampleHealthCheck(data1, data2),
            failureStatus,
            tags));
    }
}
```

运行状况检查发布服务器

当 `IHealthCheckPublisher` 添加到服务容器时，运行状况检查系统，会定期执行运行状况检查并使用结果调用 `PublishAsync`。在期望每个进程定期调用监视系统以便确定运行状况的基于推送的运行状况监视系统方案中，这十分有用。

`IHealthCheckPublisher` 接口具有单个方法：

```
Task PublishAsync(HealthReport report, CancellationToken cancellationToken);
```

> **NOTE**
>
> BeatPulse 包括多个系统的发布服务器（包括 Application Insights）。
>
> BeatPulse 不由 Microsoft 进行支持或维护。

ASP.NET Core 安全性概述

2019/1/9 • • Edit Online

通过 ASP.NET Core，开发者可轻松配置和管理其应用的安全性。ASP.NET Core 的功能包括管理身份验证、授权、数据保护、HTTPS 强制、应用机密、请求防伪保护及 CORS 管理。通过这些安全功能，可以生成安全可靠的 ASP.NET Core 应用。

ASP.NET Core 安全性功能

ASP.NET Core 提供许多用于保护应用安全的工具和库（包括内置标识提供程序），但也可使用第三方标志服务（如 Facebook、Twitter 或 LinkedIn）。利用 ASP.NET Core 可以轻松管理应用机密，无需将机密信息暴露在代码中就可存储和使用它们。

身份验证 vs授权

身份验证是这样一个过程：由用户提供凭据，然后将其与存储在操作系统、数据库、应用或资源中的凭据进行比较。在授权过程中，如果凭据匹配，则用户身份验证成功，可执行已向其授权的操作。授权指判断允许用户执行的操作的过程。

对身份验证的另一种理解是将其看作进入某一空间（如服务器、数据库、应用或资源）的方式，而将授权看作用户可对该空间（服务器、数据库或应用）内的对象执行的操作。

软件中的常见漏洞

ASP.NET Core 和 EF 提供维护应用安全、预防安全漏洞的功能。下表中链接的文档详细介绍了在 Web 应用中避免最常见安全漏洞的技术：

- 跨站点脚本攻击
- SQL 注入式攻击
- 跨站点请求伪造 (CSRF)
- 打开重定向攻击

还应注意其他漏洞。有关详细信息，请参阅目录的“安全性和标识”部分中的其他文章。

ASP.NET Core 上的标识简介

2019/1/8 • • Edit Online

作者：Rick Anderson

ASP.NET Core 标识是一个成员身份系统，将登录功能添加到 ASP.NET Core 应用。用户可以标识中存储的登录信息创建一个帐户，或它们可以使用外部登录提供程序。受支持的外部登录提供程序包括Facebook、Google、Microsoft 帐户和 Twitter。

可以使用 SQL Server 数据库来存储用户名、密码和配置文件数据配置标识。或者，另一个的持久存储区可用，例如，Azure 表存储。

查看或下载示例代码(如何下载))。

在本主题中，将了解如何使用标识来注册、登录和注销用户。有关创建使用标识的应用程序的更多详细说明，请参阅本文末尾的后续步骤部分。

AddDefaultIdentity 和 AddIdentity

AddDefaultIdentity ASP.NET Core 2.1 中引入了。调用 `AddDefaultIdentity` 类似于调用以下：

- AddIdentity
- AddDefaultUI
- AddDefaultTokenProviders

请参阅AddDefaultIdentity 源有关详细信息。

使用身份验证创建的 Web 应用

使用单个用户帐户创建一个 ASP.NET Core Web 应用程序项目。

- Visual Studio
- .NET Core CLI

- 选择“文件” > “新建” > “项目”。
- 选择“ASP.NET Core Web 应用程序”。将项目命名**WebApp1**具有项目下载相同的命名空间。单击 **“确定”**。
- 选择 ASP.NET Core **Web 应用程序**ASP.NET Core 2.1，然后选择**更改身份验证**。
- 选择**单个用户帐户**然后单击**确定**。

生成的项目提供了ASP.NET Core 标识作为Razor 类库。

测试注册和登录名

运行应用并注册用户。具体取决于你的屏幕大小，可能需要选择导航切换按钮以查看**注册并登录名**链接。

查看标识数据库

- Visual Studio
- .NET Core CLI

- 从视图菜单中，选择**SQL Server 对象资源管理器**(SSOX)。
- 导航到 **(localdb) MSSQLLocalDB (SQL Server 13)**。右键单击**dbo。AspNetUsers** > 查看数据:

配置标识服务

在添加服务 `ConfigureServices` 。典型模式是调用所有 `Add{Service}` 方法，然后调用所有

`services.Configure{Service}` 方法。下面的代码不包括生成的模板 `CookiePolicyOptions`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.Configure<IdentityOptions>(options =>
    {
        // Password settings.
        options.Password.RequireDigit = true;
        options.Password.RequireLowercase = true;
        options.Password.RequireNonAlphanumeric = true;
        options.Password.RequireUppercase = true;
        options.Password.RequiredLength = 6;
        options.Password.RequiredUniqueChars = 1;

        // Lockout settings.
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
        options.Lockout.MaxFailedAccessAttempts = 5;
        options.Lockout.AllowedForNewUsers = true;

        // User settings.
        options.User.AllowedUserNameCharacters =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
        options.User.RequireUniqueEmail = false;
    });

    services.ConfigureApplicationCookie(options =>
    {
        // Cookie settings
        options.Cookie.HttpOnly = true;
        options.ExpireTimeSpan = TimeSpan.FromMinutes(5);

        options.LoginPath = "/Identity/Account/Login";
        options.AccessDeniedPath = "/Identity/Account/AccessDenied";
        options.SlidingExpiration = true;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

上述代码使用默认的选项值配置标识。服务可用于通过应用依赖关系注入。

标识启用通过调用UseAuthentication。`UseAuthentication` 添加了身份验证中间件到请求管道。

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseAuthentication();

    app.UseMvc();
}
```

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.Configure<IdentityOptions>(options =>
    {
        // Password settings
        options.Password.RequireDigit = true;
        options.Password.RequiredLength = 8;
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireUppercase = true;
        options.Password.RequireLowercase = false;
        options.Password.RequiredUniqueChars = 6;

        // Lockout settings
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
        options.Lockout.MaxFailedAccessAttempts = 10;
        options.Lockout.AllowedForNewUsers = true;

        // User settings
        options.User.RequireUniqueEmail = true;
    });

    services.ConfigureApplicationCookie(options =>
    {
        // Cookie settings
        options.Cookie.HttpOnly = true;
        options.ExpireTimeSpan = TimeSpan.FromMinutes(30);
        // If the LoginPath isn't set, ASP.NET Core defaults
        // the path to /Account/Login.
        options.LoginPath = "/Account/Login";
        // If the AccessDeniedPath isn't set, ASP.NET Core defaults
        // the path to /Account/AccessDenied.
        options.AccessDeniedPath = "/Account/AccessDenied";
        options.SlidingExpiration = true;
    });

    // Add application services.
    services.AddTransient<IEmailSender, EmailSender>();

    services.AddMvc();
}
```

服务都提供给应用程序通过依赖关系注入。

通过调用情况下，启用应用程序的身份标识 `UseAuthentication` 在 `Configure` 方法。 `UseAuthentication` 添加了身份验证中间件到请求管道。

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseAuthentication();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Configure Identity
    services.Configure<IdentityOptions>(options =>
    {
        // Password settings
        options.Password.RequireDigit = true;
        options.Password.RequiredLength = 8;
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireUppercase = true;
        options.Password.RequireLowercase = false;

        // Lockout settings
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
        options.Lockout.MaxFailedAccessAttempts = 10;

        // Cookie settings
        options.Cookies.ApplicationCookie.ExpireTimeSpan = TimeSpan.FromDays(150);
        options.Cookies.ApplicationCookie.LoginPath = "/Account/Login";

        // User settings
        options.User.RequireUniqueEmail = true;
    });

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

这些服务都提供给应用程序通过依赖关系注入。

通过调用情况下，启用应用程序的身份标识 `UseIdentity` 在 `Configure` 方法。 `UseIdentity` 添加了基于 cookie 的身份验证中间件到请求管道。

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseIdentity();

    // Add external authentication middleware below. To configure them please see
https://go.microsoft.com/fwlink/?LinkID=532715

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

有关详细信息，请参阅IdentityOptions 类并应用程序启动。

基架注册、登录和注销

请按照创建标识为具有授权的 Razor 项目基架如何生成在本部分中所示的代码。

- Visual Studio
- .NET Core CLI

添加注册、登录和注销文件。

检查注册

当用户单击**注册**链接， `RegisterModel.OnPostAsync` 调用操作。通过创建用户CreateAsync上 `_userManager` 对象。 `_userManager` 提供依赖关系注入）：

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = Input.Email, Email = Input.Email };
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with password.");

            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { userId = user.Id, code = code },
                protocol: Request.Scheme);

            await _emailSender.SendEmailAsync(Input.Email, "Confirm your email",
                $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

            await _signInManager.SignInAsync(user, isPersistent: false);
            return LocalRedirect(returnUrl);
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

当用户单击**注册**链接，链接 `Register` 上调用操作 `AccountController`。 `Register` 操作将创建用户，通过调用 `CreateAsync` 上 `_userManager` 对象（提供给 `AccountController` 通过依赖关系注入）：

```
//
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // For more information on how to enable account confirmation and password reset please
visit http://go.microsoft.com/fwlink/?LinkID=532713
            // Send an email with this link
            //var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            //var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code =
code }, protocol: HttpContext.Request.Scheme);
            //await _emailSender.SendEmailAsync(model.Email, "Confirm your account",
            //    "Please confirm your account by clicking this link: <a href=\"" + callbackUrl +
"\">link</a>");
            await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation(3, "User created a new account with password.");
            return RedirectToAction(nameof(HomeController.Index), "Home");
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

如果已成功创建用户，用户记录通过调用 `_signInManager.SignInAsync` 。

注意：请参阅帐户确认有关步骤，以防止在登记时立即登录。

登录

显示登录窗体时：

- 登录选择链接。
- 用户尝试访问受限的页面，他们不向被授权访问或时在还没有已完成身份验证系统。

登录页上的表单提交时， `OnPostAsync` 调用操作。 `PasswordSignInAsync` 对调用 `_signInManager` 对象（由依赖关系注入提供）。

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout,
        // set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.Email,
            Input.Password, Input.RememberMe, lockoutOnFailure: true);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new { ReturnUrl = returnUrl,
                RememberMe = Input.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

基 `Controller` 类公开 `User`，可以从控制器方法访问的属性。例如，可以枚举 `User.Claims` 和做出授权决定。有关详细信息，请参阅 在 ASP.NET Core 中授权简介。

当用户选择时，将显示登录窗体登录访问要求进行身份验证的页面时将被重定向或链接。当用户提交窗体上的登录页中， `AccountController` `Login` 调用操作。

`Login` 操作调用 `PasswordSignInAsync` 上 `_signInManager` 对象 (提供给 `AccountController` 通过依赖关系注入)。

```
//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(model.Email,
            model.Password, model.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation(1, "User logged in.");
            return RedirectToLocal(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToAction(nameof(SendCode), new { ReturnUrl = returnUrl, RememberMe =
model.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning(2, "User account locked out.");
            return View("Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return View(model);
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

基本（`Controller`或`PageModel`）类公开`User`属性。例如，`User.Claims`可以枚举才能做出授权决定。

注销

注销链接调用`LogoutModel.OnPost`操作。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;

namespace WebApp1.Areas.Identity.Pages.Account
{
    [AllowAnonymous]
    public class LogoutModel : PageModel
    {
        private readonly SignInManager<IdentityUser> _signInManager;
        private readonly ILogger<LogoutModel> _logger;

        public LogoutModel(SignInManager<IdentityUser> signInManager, ILogger<LogoutModel> logger)
        {
            _signInManager = signInManager;
            _logger = logger;
        }

        public void OnGet()
        {
        }

        public async Task<IActionResult> OnPost(string returnUrl = null)
        {
            await _signInManager.SignOutAsync();
            _logger.LogInformation("User logged out.");
            if (returnUrl != null)
            {
                return LocalRedirect(returnUrl);
            }
            else
            {
                return Page();
            }
        }
    }
}
```

SignOutAsync清除存储在 cookie 中的用户的声明。在调用后不重定向`SignOutAsync`或者在用户会不已注销。

中指定 post *Pages/Shared/_LoginPartial.cshtml*:

```
@using Microsoft.AspNetCore.Identity

@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="Identity" asp-page="/Account/Logout"
          asp-route-returnUrl="@Url.Page("/Index", new { area = "" })"
          method="post"
          id="logoutForm" class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="Identity" asp-page="/Account/Manage/Index" title="Manage">
                          Hello @UserManager.GetUserName(User)!</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn navbar-link">Logout</button>
            </li>
        </ul>
    </form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="Identity" asp-page="/Account/Register">Register</a></li>
        <li><a asp-area="Identity" asp-page="/Account/Login">Login</a></li>
    </ul>
}
```

单击**注销**链接调用 `LogOut` 操作。

```
//
// POST: /Account/LogOut
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> LogOut()
{
    await _signInManager.SignOutAsync();
    _logger.LogInformation(4, "User logged out.");
    return RedirectToAction(nameof(HomeController.Index), "Home");
}
```

上述代码调用 `_signInManager.SignOutAsync` 方法。 `SignOutAsync` 方法将清除用户的声明存储在 cookie 中。

测试标识

默认 web 项目模板允许匿名访问主页。若要测试标识，将添加 `[Authorize]` 到关于页面。

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace WebApp1.Pages
{
    [Authorize]
    public class AboutModel : PageModel
    {
        public string Message { get; set; }

        public void OnGet()
        {
            Message = "Your application description page.";
        }
    }
}
```

如果你登录，请注销。运行应用并选择**有关**链接。你将重定向到登录页。

了解标识

若要更详细地了解标识：

- 创建完整的标识 UI 源
- 检查每个页面和单步执行调试器的源。

标识组件

中包含所有标识依赖 NuGet 包Microsoft.AspNetCore.App 元包。

标识将主包是Microsoft.AspNetCore.Identity。此包包含 ASP.NET Core 标识的核心接口集，是 `Microsoft.AspNetCore.Identity.EntityFrameworkCore` 提供的。

迁移到 ASP.NET Core 标识

有关详细信息和指南迁移现有的标识存储，请参阅迁移身份验证和标识。

设置密码强度

请参阅配置有关设置的最小密码要求的示例。

后续步骤

- 配置标识
- 使用受授权的用户数据创建 ASP.NET Core 应用
- 添加、下载和删除标识到 ASP.NET Core 项目中的用户数据
- 启用在 ASP.NET Core TOTP 身份验证器应用的 QR 代码生成
- 将身份验证和标识迁移到 ASP.NET Core
- 帐户确认和 ASP.NET Core 中的密码恢复
- 在 ASP.NET Core SMS 的双因素身份验证
- 在 Web 场中托管 ASP.NET Core

ASP.NET Core 项目中的基架标识

2018/10/26 • • Edit Online

作者：Rick Anderson

ASP.NET Core 2.1 及更高版本提供了ASP.NET Core 标识作为Razor 类库。包含标识的应用程序可以应用基架来有选择地添加包含在标识 Razor 类库 (RCL) 的源代码。建议生成源代码，以便修改代码和更改行为。例如，可以指示基架生成在注册过程中使用的代码。生成的代码优先于标识 RCL 中的相同代码。若要获取的用户界面的完全控制，并且使用默认 RCL，请参阅部分创建完整的标识 UI 源。

执行操作的应用程序不包括身份验证可以应用基架添加 RCL 标识包。可以选择要生成的标识代码。

虽然基架生成大部分所需的代码，但您必须更新项目以完成该过程。本文档介绍完成标识基架更新所需的步骤。

当运行时标识基架时，*ScaffoldingReadme.txt*在项目目录中创建文件。*ScaffoldingReadme.txt*文件包含所需完成标识基架更新的一般说明。本文档包含更完整说明比*ScaffoldingReadme.txt*文件。

我们建议使用一个源控制系统，显示文件的差异，并允许您后退的更改。运行标识基架后检查所做的更改。

> **NOTE**
>
> 使用时，服务所需双因素身份验证，帐户确认和密码恢复，和使用标识其他安全功能。基架标识时，不会生成服务或服务存根（stub）。必须手动添加服务，以启用这些功能。有关示例，请参阅需要电子邮件确认。

基架标识为一个空项目

运行标识基架：

- Visual Studio
- .NET Core CLI

- 从**解决方案资源管理器**，右键单击该项目 >**添加** > **新基架项**。
- 从左窗格**添加基架**对话框中，选择**标识** > **添加**。
- 在中**ADD 标识添加**对话框中，选择所需的选项。
  - 选择现有的布局页上，或不正确的标记将被覆盖布局文件。例如`~/Pages/Shared/_Layout.cshtml`Razor 页面`~/Views/Shared/_Layout.cshtml`对于 MVC 项目
  - 选择**+** 按钮以创建一个新数据上下文类。
- 选择**添加**。

添加以下突出显示的调用到`Startup`类：

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseAuthentication();
        app.UseMvc();
    }
}
```

`UseHsts` 建议但不是必需的。请参阅HTTP 严格传输安全性协议有关详细信息。

生成的标识数据库代码需要Entity Framework Core 迁移。创建迁移并更新数据库。例如，运行以下命令：

- Visual Studio
- .NET Core CLI

在 Visual Studio**程序包管理器控制台**：

```
Add-Migration CreateIdentitySchema
Update-Database
```

"CreateIdentitySchema"name 参数为 `Add-Migration` 是任意的命令。 `"CreateIdentitySchema"` 介绍迁移。

基架的 Razor 项目没有现有的授权到标识

运行标识基架：

- Visual Studio
- .NET Core CLI

- **从解决方案资源管理器**，右键单击该项目 >**添加** > **新基架项**。
- **从左窗格添加基架**对话框中，选择**标识** > **添加**。
- 在中**ADD 标识添加**对话框中，选择所需的选项。
  - 选择现有的布局页上，或不正确的标记将被覆盖布局文件。例如 `~/Pages/Shared/_Layout.cshtml` Razor 页面 `~/Views/Shared/_Layout.cshtml` 对于 MVC 项目
  - 选择**+** 按钮以创建一个新数据上下文类。
- 选择**添加**。

在中配置标识*Areas/Identity/IdentityHostingStartup.cs*。有关详细信息，请参阅IHostingStartup。

迁移、UseAuthentication 和布局

生成的标识数据库代码需要Entity Framework Core 迁移。创建迁移并更新数据库。例如，运行以下命令：

- Visual Studio
- .NET Core CLI

在 Visual Studio**程序包管理器控制台**：

```
Add-Migration CreateIdentitySchema
Update-Database
```

"CreateIdentitySchema"name 参数为 `Add-Migration` 是任意的命令。 `"CreateIdentitySchema"` 介绍迁移。

启用身份验证

在中 `Configure` 方法 `Startup` 类中，调用UseAuthentication后 `UseStaticFiles`：

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseAuthentication();

        app.UseMvc();
    }
}
```

`UseHsts` 建议但不是必需的。请参阅HTTP 严格传输安全性协议有关详细信息。

布局更改

可选：添加部分的登录名（`_LoginPartial`）的布局文件：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorNoAuth8</title>

    <environment include="Development">
```

```
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-brand">RazorNoAuth8</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Contact">Contact</a></li>
                </ul>
                <partial name="_LoginPartial" />
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 - RazorNoAuth8</p>
        </footer>
    </div>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
        </script>
        <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

基架标识为具有授权的 Razor 项目

运行标识基架：

- Visual Studio
- .NET Core CLI

- 从**解决方案资源管理器**，右键单击该项目 >**添加** > **新基架项**。
- 从左窗格**添加基架**对话框中，选择**标识** > **添加**。
- 在中**ADD 标识添加**对话框中，选择所需的选项。
  - 选择现有的布局页上，或不正确的标记将被覆盖布局文件。当现有 *_Layout.cshtml*选择文件，它是不覆盖。

例如 `~/Pages/Shared/_Layout.cshtml` Razor 页面 `~/Views/Shared/_Layout.cshtml` 对于 MVC 项目

- 若要使用现有的数据上下文，请选择要重写的至少一个文件。必须选择至少一个文件以添加数据上下文。
  - 选择您的数据上下文类。
  - 选择**添加**。
- 若要创建新的用户上下文，并可能对标识创建一个自定义用户类：
  - 选择**+** 按钮以创建一个新数据上下文类。
  - 选择**添加**。

注意：如果要创建新的用户上下文，您无需选择要重写的文件。

在中配置一些标识选项*Areas/Identity/IdentityHostingStartup.cs*。有关详细信息，请参阅IHostingStartup。

基架标识到 MVC 项目中没有现有的授权

运行标识基架：

- Visual Studio
- .NET Core CLI

- 从**解决方案资源管理器**，右键单击该项目 >**添加** > **新基架项**。
- 从左窗格**添加基架**对话框中，选择**标识** > **添加**。
- 在中**ADD 标识添加**对话框中，选择所需的选项。
  - 选择现有的布局页上，或不正确的标记将被覆盖布局文件。例如 `~/Pages/Shared/_Layout.cshtml` Razor 页面 `~/Views/Shared/_Layout.cshtml` 对于 MVC 项目
  - 选择**+** 按钮以创建一个新数据上下文类。
- 选择**添加**。

可选：添加部分的登录名（`_LoginPartial`）到*Views/Shared/_Layout.cshtml*文件：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - MvcNoAuth3</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
```

```
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">MvcNoAuth3</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
                </ul>
                <partial name="_LoginPartial" />
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 - MvcNoAuth3</p>
        </footer>
    </div>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
        </script>
        <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

- 移动*Pages/Shared/_LoginPartial.cshtml*文件为*Views/Shared/_LoginPartial.cshtml*

在中配置标识*Areas/Identity/IdentityHostingStartup.cs*。有关详细信息，请参阅 IHostingStartup。

生成的标识数据库代码需要Entity Framework Core 迁移。创建迁移并更新数据库。例如，运行以下命令：

- Visual Studio
- .NET Core CLI

在 Visual Studio**程序包管理器控制台**：

```
Add-Migration CreateIdentitySchema
Update-Database
```

"CreateIdentitySchema"name 参数为`Add-Migration`是任意的命令。`"CreateIdentitySchema"`介绍迁移。

调用UseAuthentication后`UseStaticFiles`：

```
public class Startup
{

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseAuthentication();
        app.UseMvcWithDefaultRoute();
    }
}
```

`UseHsts`建议但不是必需的。请参阅HTTP 严格传输安全性协议有关详细信息。

基架标识为具有授权的 MVC 项目

运行标识基架：

- Visual Studio
- .NET Core CLI

- **从解决方案资源管理器**，右键单击该项目 >**添加** > **新基架项**。
- **从左窗格添加基架**对话框中，选择**标识** > **添加**。
- 在中**ADD 标识添加**对话框中，选择所需的选项。
  - 选择现有的布局页上，或不正确的标记将被覆盖布局文件。当现有 *_Layout.cshtml*选择文件，它是不覆盖。

例如`~/Pages/Shared/_Layout.cshtml`Razor 页面`~/Views/Shared/_Layout.cshtml`对于 MVC 项目

- 若要使用现有的数据上下文，请选择要重写的至少一个文件。必须选择至少一个文件以添加数据上下文。
  - 选择您的数据上下文类。
  - 选择**添加**。
- 若要创建新的用户上下文，并可能对标识创建一个自定义用户类：
  - 选择**+** 按钮以创建一个新**数据上下文**类。
  - 选择**添加**。

注意：如果要创建新的用户上下文，您无需选择要重写的文件。

删除*Pages/共享*文件夹和该文件夹中的文件。

创建完整的标识 UI 源

若要维护标识用户界面的完全控制，运行标识基架，并选择**重写的所有文件**。

以下突出显示的代码显示默认标识 UI 替换标识在 ASP.NET Core 2.1 web 应用的更改。您可能想要执行此操作以具有完全控制权限的标识 UI。

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<IdentityUser, IdentityRole>()
        // services.AddDefaultIdentity<IdentityUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1)
        .AddRazorPagesOptions(options =>
        {
            options.AllowAreas = true;
            options.Conventions.AuthorizeAreaFolder("Identity", "/Account/Manage");
            options.Conventions.AuthorizeAreaPage("Identity", "/Account/Logout");
        });

    services.ConfigureApplicationCookie(options =>
    {
        options.LoginPath = $"/Identity/Account/Login";
        options.LogoutPath = $"/Identity/Account/Logout";
        options.AccessDeniedPath = $"/Identity/Account/AccessDenied";
    });

    // using Microsoft.AspNetCore.Identity.UI.Services;
    services.AddSingleton<IEmailSender, EmailSender>();
}
```

在下面的代码替换默认标识：

```
services.AddIdentity<IdentityUser, IdentityRole>()
    // services.AddDefaultIdentity<IdentityUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

下面的代码集LoginPath，LogoutPath，并AccessDeniedPath:

```
services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = $"/Identity/Account/Login";
    options.LogoutPath = $"/Identity/Account/Logout";
    options.AccessDeniedPath = $"/Identity/Account/AccessDenied";
});
```

注册 `IEmailSender` 实现，例如：

```
// using Microsoft.AspNetCore.Identity.UI.Services;
services.AddSingleton<IEmailSender, EmailSender>();
```

```
public class EmailSender : IEmailSender
{
    public Task SendEmailAsync(string email, string subject, string message)
    {
        return Task.CompletedTask;
    }
}
```

其他资源

- 更改身份验证代码到 ASP.NET Core 2.1 及更高版本

添加、下载和删除标识到 ASP.NET Core 项目中的自定义用户数据

2018/12/17 • • Edit Online

作者：Rick Anderson

本文介绍如何：

- 将自定义用户数据添加到 ASP.NET Core web 应用程序。
- 修饰具有的自定义用户数据模型PersonalData属性是自动可供下载和删除。使能够下载和删除数据可帮助满足GDPR要求。

项目示例将创建从 Razor 页 web 应用，但了 ASP.NET Core MVC web 应用的类似的说明。

查看或下载示例代码（如何下载）

系统必备

.NET Core 2.1 SDK 或更高版本

创建 Razor Web 应用

- Visual Studio
- .NET Core CLI

- 从 Visual Studio“文件”菜单中选择“新建” > “项目”。将项目命名**WebApp1**如果你想与其匹配的命名空间下载示例代码。
- 选择**ASP.NET Core Web 应用程序** > **确定**
- 选择**ASP.NET Core 2.1**下拉列表中
- 选择**Web 应用程序** > **确定**
- 生成并运行该项目。

运行标识基架

- Visual Studio
- .NET Core CLI

- 从**解决方案资源管理器**，右键单击该项目 >**添加** > **新基架项**。
- 从左窗格**添加基架**对话框中，选择**标识** > **添加**。
- 在中**ADD 标识添加**对话框中，以下选项：
  - 选择现有的布局文件 *~/Pages/Shared/_Layout.cshtml*
  - 选择要重写的以下文件：
    - **帐户/注册**
    - **帐户/管理/索引**
  - 选择 **+** 按钮以创建一个新**数据上下文**类。接受的类型 (**WebApp1.Models.WebApp1Context**如果项目命名为**WebApp1**)。
  - 选择 **+** 按钮以创建一个新**User** 类。接受的类型 (**WebApp1User**如果项目命名为**WebApp1**) >**添加**。
- 选择**添加**。

按照中的说明迁移、UseAuthentication 和布局来执行以下步骤：

- 创建迁移并更新数据库。
- 将 `UseAuthentication` 添加到 `Startup.Configure` 。
- 添加 `<partial name="_LoginPartial" />` 布局文件。
- 测试应用：
  - 注册用户
  - 选择新的用户名称（旁边注销链接）。您可能需要展开窗口或选择要显示的用户名称和其他链接的导航栏图标。
  - 选择**个人数据**选项卡。
  - 选择**下载**按钮，然后检查*PersonalData.json*文件。
  - 测试**删除**按钮，删除已登录用户。

向标识数据库中添加自定义用户数据

更新 `IdentityUser` 派生类使用自定义属性。如果项目 WebApp1 命名为，将该文件命名*Areas/Identity/Data/WebApp1User.cs*。使用以下代码更新文件：

```
using Microsoft.AspNetCore.Identity;
using System;

namespace WebApp1.Areas.Identity.Data
{
    public class WebApp1User : IdentityUser
    {
        [PersonalData]
        public string Name { get; set; }
        [PersonalData]
        public DateTime DOB { get; set; }
    }
}
```

使用属性修饰PersonalData特性是：

- 时删除*Areas/Identity/Pages/Account/Manage/DeletePersonalData.cshtml* Razor 页面调用 `UserManager.Delete` 。
- 通过下载的数据中包含*Areas/Identity/Pages/Account/Manage/DownloadPersonalData.cshtml* Razor 页面。

更新 Account/Manage/Index.cshtml 页

更新 `InputModel` 中*Areas/Identity/Pages/Account/Manage/Index.cshtml.cs*用以下突出显示的代码：

```
public partial class IndexModel : PageModel
{
    private readonly UserManager<WebApp1User> _userManager;
    private readonly SignInManager<WebApp1User> _signInManager;
    private readonly IEmailSender _emailSender;

    public IndexModel(
        UserManager<WebApp1User> userManager,
        SignInManager<WebApp1User> signInManager,
        IEmailSender emailSender)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _emailSender = emailSender;
    }

    public string Username { get; set; }
```

```
    public bool IsEmailConfirmed { get; set; }

    [TempData]
    public string StatusMessage { get; set; }
    [BindProperty]
    public InputModel Input { get; set; }

    public class InputModel
    {
        [Required]
        [DataType(DataType.Text)]
        [Display(Name = "Full name")]
        public string Name { get; set; }

        [Required]
        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime DOB { get; set; }

        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Phone]
        [Display(Name = "Phone number")]
        public string PhoneNumber { get; set; }
    }

    public async Task<IActionResult> OnGetAsync()
    {
        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound($"Unable to load user with ID '{_userManager.GetUserId(User)}'.");
        }

        var userName = await _userManager.GetUserNameAsync(user);
        var email = await _userManager.GetEmailAsync(user);
        var phoneNumber = await _userManager.GetPhoneNumberAsync(user);

        Username = userName;

        Input = new InputModel
        {
            Name = user.Name,
            DOB = user.DOB,
            Email = email,
            PhoneNumber = phoneNumber
        };

        IsEmailConfirmed = await _userManager.IsEmailConfirmedAsync(user);

        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound($"Unable to load user with ID '{_userManager.GetUserId(User)}'.");
        }

        if (Input.Name != user.Name)
```

```
        if (Input.Name != user.Name)
        {
            user.Name = Input.Name;
        }

        if (Input.DOB != user.DOB)
        {
            user.DOB = Input.DOB;
        }


        var email = await _userManager.GetEmailAsync(user);
        if (Input.Email != email)
        {
            var setEmailResult = await _userManager.SetEmailAsync(user, Input.Email);
            if (!setEmailResult.Succeeded)
            {
                var userId = await _userManager.GetUserIdAsync(user);
                throw new InvalidOperationException($"Unexpected error occurred setting email for user with ID
'{userId}'.");
            }
        }

        var phoneNumber = await _userManager.GetPhoneNumberAsync(user);
        if (Input.PhoneNumber != phoneNumber)
        {
            var setPhoneResult = await _userManager.SetPhoneNumberAsync(user, Input.PhoneNumber);
            if (!setPhoneResult.Succeeded)
            {
                var userId = await _userManager.GetUserIdAsync(user);
                throw new InvalidOperationException($"Unexpected error occurred setting phone number for user
with ID '{userId}'.");
            }
        }

        await _userManager.UpdateAsync(user);

        await _signInManager.RefreshSignInAsync(user);
        StatusMessage = "Your profile has been updated";
        return RedirectToPage();
    }

    public async Task<IActionResult> OnPostSendVerificationEmailAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound($"Unable to load user with ID '{_userManager.GetUserId(User)}'.");
        }


        var userId = await _userManager.GetUserIdAsync(user);
        var email = await _userManager.GetEmailAsync(user);
        var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
        var callbackUrl = Url.Page(
            "/Account/ConfirmEmail",
            pageHandler: null,
            values: new { userId = userId, code = code },
            protocol: Request.Scheme);
        await _emailSender.SendEmailAsync(
            email,
            "Confirm your email",
            $"Please confirm your account by <a href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking
here</a>.");
```

```
        StatusMessage = "Verification email sent. Please check your email.";
        return RedirectToPage();
    }
}
```

更新*Areas/Identity/Pages/Account/Manage/Index.cshtml*与以下突出显示的标记：

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Profile";
}

<h4>@ViewData["Title"]</h4>
@Html.Partial("_StatusMessage", Model.StatusMessage)
<div class="row">
    <div class="col-md-6">
        <form id="profile-form" method="post">
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Username"></label>
                <input asp-for="Username" class="form-control" disabled />
            </div>
            <div class="form-group">
                <label asp-for="Input.Email"></label>
                @if (Model.IsEmailConfirmed)
                {
                    <div class="input-group">
                        <input asp-for="Input.Email" class="form-control" />
                        <span class="input-group-addon" aria-hidden="true"><span class="glyphicon glyphicon-ok
text-success"></span></span>
                    </div>
                }
                else
                {
                    <input asp-for="Input.Email" class="form-control" />
                    <button id="email-verification" type="submit" asp-page-handler="SendVerificationEmail"
class="btn btn-link">Send verification email</button>
                }
                <span asp-validation-for="Input.Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <div class="form-group">
                    <label asp-for="Input.Name"></label>
                    <input asp-for="Input.Name" class="form-control" />
                </div>
                <div class="form-group">
                    <label asp-for="Input.DOB"></label>
                    <input asp-for="Input.DOB" class="form-control" />
                </div>
                <label asp-for="Input.PhoneNumber"></label>
                <input asp-for="Input.PhoneNumber" class="form-control" />
                <span asp-validation-for="Input.PhoneNumber" class="text-danger"></span>
            </div>
            <button type="submit" class="btn btn-default">Save</button>
        </form>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

更新 account/Register.cshtml 页面

更新 `InputModel` 中*Areas/Identity/Pages/Account/Register.cshtml.cs*用以下突出显示的代码：

```
[BindProperty]
public InputModel Input { get; set; }

public string ReturnUrl { get; set; }

public class InputModel
{
    [Required]
    [DataType(DataType.Text)]
    [Display(Name = "Full name")]
    public string Name { get; set; }

    [Required]
    [Display(Name = "Birth Date")]
    [DataType(DataType.Date)]
    public DateTime DOB { get; set; }

    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.",
MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}

public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    if (ModelState.IsValid)
    {
        var user = new WebApp1User {
            UserName = Input.Email,
            Email = Input.Email,
            Name = Input.Name,
            DOB = Input.DOB
        };
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with password.");

            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { userId = user.Id, code = code },
                protocol: Request.Scheme);

            await _emailSender.SendEmailAsync(Input.Email, "Confirm your email",
                $"Please confirm your account by <a href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking
here</a>.");

            await _signInManager.SignInAsync(user, isPersistent: false);
            return LocalRedirect(returnUrl);
        }
        foreach (var error in result.Errors)
```

```
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

更新*Areas/Identity/Pages/Account/Register.cshtml*与以下突出显示的标记：

```
@page
@model RegisterModel
@{
    ViewData["Title"] = "Register";
}

<h2>@ViewData["Title"]</h2>

<div class="row">
    <div class="col-md-4">
        <form asp-route-returnUrl="@Model.ReturnUrl" method="post">
            <h4>Create a new account.</h4>
            <hr />
            <div asp-validation-summary="All" class="text-danger"></div>

            <div class="form-group">
                <label asp-for="Input.Name"></label>
                <input asp-for="Input.Name" class="form-control" />
                <span asp-validation-for="Input.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.DOB"></label>
                <input asp-for="Input.DOB" class="form-control" />
                <span asp-validation-for="Input.DOB" class="text-danger"></span>
            </div>

            <div class="form-group">
                <label asp-for="Input.Email"></label>
                <input asp-for="Input.Email" class="form-control" />
                <span asp-validation-for="Input.Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.Password"></label>
                <input asp-for="Input.Password" class="form-control" />
                <span asp-validation-for="Input.Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.ConfirmPassword"></label>
                <input asp-for="Input.ConfirmPassword" class="form-control" />
                <span asp-validation-for="Input.ConfirmPassword" class="text-danger"></span>
            </div>
            <button type="submit" class="btn btn-default">Register</button>
        </form>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

生成项目。

添加自定义用户数据的迁移

- Visual Studio

- .NET Core CLI

在 Visual Studio程序包管理器控制台:

```
Add-Migration CustomUserData
Update-Database
```

测试创建、查看、下载和删除自定义用户数据

测试应用:

- 注册一个新用户。
- 查看自定义用户数据 `/Identity/Account/Manage` 页。
- 下载并查看用户个人数据从 `/Identity/Account/Manage/PersonalData` 页。

在 ASP.NET Core 中的标识模型自定义

2018/9/27 • • Edit Online

通过Arthur Vickers

ASP.NET Core 标识提供了一个框架，用于管理和存储在 ASP.NET Core 应用中的用户帐户。标识添加到项目时单个用户帐户选择作为身份验证机制。默认情况下，标识，可以使用的 Entity Framework (EF) Core 数据模型。本文介绍如何自定义的身份标识模型。

标识和 EF Core 迁移

之前检查模型，它是有助于了解如何标识配合EF Core 迁移创建和更新数据库。最高级别的过程是：

1. 定义或更新代码中的数据模型。
2. 添加迁移，以将此模型转换为可以应用到数据库的更改。
3. 检查迁移正确表示你的意图。
4. 将应用迁移来更新要与模型同步的数据库。
5. 重复步骤 1 至 4 以进一步优化模型并使数据库保持同步。

使用以下方法之一来添加和应用迁移：

- **程序包管理器控制台**(PMC) 窗口，如果使用 Visual Studio。有关详细信息，请参阅EF Core PMC 工具。
- 使用命令行在.NET Core CLI。有关详细信息，请参阅EF Core.NET 命令行工具。
- 单击**应用迁移**时运行该应用程序错误页上的按钮。

ASP.NET Core 具有开发时间错误页处理程序。当应用运行时，该处理程序可以将应用迁移。对于生产应用程序，它通常会更适合从迁移生成 SQL 脚本并将其作为受控的应用和数据库部署的一部分部署数据库更改。

创建新的应用使用标识时，已经完成上述步骤 1 和 2。也就是说，初始数据模型已存在，并已向项目添加初始迁移。初始迁移仍需要将应用到数据库。可以通过以下方法之一应用初始迁移：

- 运行 `Update-Database` 在 PMC 中。
- 运行 `dotnet ef database update` 命令行界面中。
- 单击**应用迁移**时运行该应用程序错误页上的按钮。

根据对模型进行更改，请重复前面的步骤。

标识模型

实体类型

标识模型包含以下实体类型。

实体类型	描述
`User`	表示的用户。
`Role`	表示一个角色。
`UserClaim`	表示一个用户拥有的声明。

实体类型	描述
`UserToken`	表示用户的身份验证令牌。
`UserLogin`	将用户与登录名相关联。
`RoleClaim`	表示一个角色内的所有用户授予的声明。
`UserRole`	联接实体相关联的用户和角色。

实体的类型关系

实体类型彼此相关的以下方面：

- 每个 `User` 都可以具有许多 `UserClaims` 。
- 每个 `User` 都可以具有许多 `UserLogins` 。
- 每个 `User` 都可以具有许多 `UserTokens` 。
- 每个 `Role` 都可以具有许多关联 `RoleClaims` 。
- 每个 `User` 都可以具有许多关联 `Roles` ，和每个 `Role` 可与许多关联 `Users` 。这是需要在数据库中的联接表的多对多关系。联接表由 `UserRole` 实体。

默认模型配置

标识定义许多*上下文*类继承自DbContext若要配置和使用模型。进行此配置使用EF Core Code First Fluent API中OnModelCreating上下文类的方法。默认配置是：

```
builder.Entity<TUser>(b =>
{
    // Primary key
    b.HasKey(u => u.Id);

    // Indexes for "normalized" username and email, to allow efficient lookups
    b.HasIndex(u => u.NormalizedUserName).HasName("UserNameIndex").IsUnique();
    b.HasIndex(u => u.NormalizedEmail).HasName("EmailIndex");

    // Maps to the AspNetUsers table
    b.ToTable("AspNetUsers");

    // A concurrency token for use with the optimistic concurrency checking
    b.Property(u => u.ConcurrencyStamp).IsConcurrencyToken();

    // Limit the size of columns to use efficient database types
    b.Property(u => u.UserName).HasMaxLength(256);
    b.Property(u => u.NormalizedUserName).HasMaxLength(256);
    b.Property(u => u.Email).HasMaxLength(256);
    b.Property(u => u.NormalizedEmail).HasMaxLength(256);

    // The relationships between User and other entity types
    // Note that these relationships are configured with no navigation properties

    // Each User can have many UserClaims
    b.HasMany<TUserClaim>().WithOne().HasForeignKey(uc => uc.UserId).IsRequired();

    // Each User can have many UserLogins
    b.HasMany<TUserLogin>().WithOne().HasForeignKey(ul => ul.UserId).IsRequired();

    // Each User can have many UserTokens
    b.HasMany<TUserToken>().WithOne().HasForeignKey(ut => ut.UserId).IsRequired();

    // Each User can have many entries in the UserRole join table
    b.HasMany<TUserRole>().WithOne().HasForeignKey(ur => ur.UserId).IsRequired();
});
```

```
builder.Entity<TUserClaim>(b =>
{
    // Primary key
    b.HasKey(uc => uc.Id);

    // Maps to the AspNetUserClaims table
    b.ToTable("AspNetUserClaims");
});

builder.Entity<TUserLogin>(b =>
{
    // Composite primary key consisting of the LoginProvider and the key to use
    // with that provider
    b.HasKey(l => new { l.LoginProvider, l.ProviderKey });

    // Limit the size of the composite key columns due to common DB restrictions
    b.Property(l => l.LoginProvider).HasMaxLength(128);
    b.Property(l => l.ProviderKey).HasMaxLength(128);

    // Maps to the AspNetUserLogins table
    b.ToTable("AspNetUserLogins");
});

builder.Entity<TUserToken>(b =>
{
    // Composite primary key consisting of the UserId, LoginProvider and Name
    b.HasKey(t => new { t.UserId, t.LoginProvider, t.Name });

    // Limit the size of the composite key columns due to common DB restrictions
    b.Property(t => t.LoginProvider).HasMaxLength(maxKeyLength);
    b.Property(t => t.Name).HasMaxLength(maxKeyLength);

    // Maps to the AspNetUserTokens table
    b.ToTable("AspNetUserTokens");
});

builder.Entity<TRole>(b =>
{
    // Primary key
    b.HasKey(r => r.Id);

    // Index for "normalized" role name to allow efficient lookups
    b.HasIndex(r => r.NormalizedName).HasName("RoleNameIndex").IsUnique();

    // Maps to the AspNetRoles table
    b.ToTable("AspNetRoles");

    // A concurrency token for use with the optimistic concurrency checking
    b.Property(r => r.ConcurrencyStamp).IsConcurrencyToken();

    // Limit the size of columns to use efficient database types
    b.Property(u => u.Name).HasMaxLength(256);
    b.Property(u => u.NormalizedName).HasMaxLength(256);

    // The relationships between Role and other entity types
    // Note that these relationships are configured with no navigation properties

    // Each Role can have many entries in the UserRole join table
    b.HasMany<TUserRole>().WithOne().HasForeignKey(ur => ur.RoleId).IsRequired();

    // Each Role can have many associated RoleClaims
    b.HasMany<TRoleClaim>().WithOne().HasForeignKey(rc => rc.RoleId).IsRequired();
});

builder.Entity<TRoleClaim>(b =>
{
    // Primary key
    b.HasKey(rc => rc.Id);
```

```
    b.HasKey(r => r.Id);

    // Maps to the AspNetRoleClaims table
    b.ToTable("AspNetRoleClaims");
});

builder.Entity<TUserRole>(b =>
{
    // Primary key
    b.HasKey(r => new { r.UserId, r.RoleId });

    // Maps to the AspNetUserRoles table
    b.ToTable("AspNetUserRoles");
});
```

模型的泛型类型

标识定义默认值公共语言运行时上面列出的每个实体类型 (CLR) 类型。这些类型带有前缀标识:

- `IdentityUser`
- `IdentityRole`
- `IdentityUserClaim`
- `IdentityUserToken`
- `IdentityUserLogin`
- `IdentityRoleClaim`
- `IdentityUserRole`

而不是直接使用这些类型，类型可以用作基类，这些类对于应用自己的类型。`DbContext`定义标识的类是通用的这样不同的 CLR 类型可以用于一个或多个模型中的实体类型。这些泛型类型还允许`User`主键 (PK) 数据类型更改。

对于角色，支持使用标识时IdentityDbContext应使用类。例如：

```
// Uses all the built-in Identity types
// Uses `string` as the key type
public class IdentityDbContext
    : IdentityDbContext<IdentityUser, IdentityRole, string>
{
}

// Uses the built-in Identity types except with a custom User type
// Uses `string` as the key type
public class IdentityDbContext<TUser>
    : IdentityDbContext<TUser, IdentityRole, string>
        where TUser : IdentityUser
{
}

// Uses the built-in Identity types except with custom User and Role types
// The key type is defined by TKey
public class IdentityDbContext<TUser, TRole, TKey> : IdentityDbContext<
    TUser, TRole, TKey, IdentityUserClaim<TKey>, IdentityUserRole<TKey>,
    IdentityUserLogin<TKey>, IdentityRoleClaim<TKey>, IdentityUserToken<TKey>>
        where TUser : IdentityUser<TKey>
        where TRole : IdentityRole<TKey>
        where TKey : IEquatable<TKey>
{
}

// No built-in Identity types are used; all are specified by generic arguments
// The key type is defined by TKey
public abstract class IdentityDbContext<
    TUser, TRole, TKey, TUserClaim, TUserRole, TUserLogin, TRoleClaim, TUserToken>
    : IdentityUserContext<TUser, TKey, TUserClaim, TUserLogin, TUserToken>
         where TUser : IdentityUser<TKey>
         where TRole : IdentityRole<TKey>
         where TKey : IEquatable<TKey>
         where TUserClaim : IdentityUserClaim<TKey>
         where TUserRole : IdentityUserRole<TKey>
         where TUserLogin : IdentityUserLogin<TKey>
         where TRoleClaim : IdentityRoleClaim<TKey>
         where TUserToken : IdentityUserToken<TKey>
```

还有可能在这种情况下使用而无需角色（仅声明），标识IdentityUserContext<TUser>应使用类：

```
// Uses the built-in non-role Identity types except with a custom User type
// Uses `string` as the key type
public class IdentityUserContext<TUser>
    : IdentityUserContext<TUser, string>
        where TUser : IdentityUser
{
}

// Uses the built-in non-role Identity types except with a custom User type
// The key type is defined by TKey
public class IdentityUserContext<TUser, TKey> : IdentityUserContext<
    TUser, TKey, IdentityUserClaim<TKey>, IdentityUserLogin<TKey>,
    IdentityUserToken<TKey>>
        where TUser : IdentityUser<TKey>
        where TKey : IEquatable<TKey>
{
}

// No built-in Identity types are used; all are specified by generic arguments, with no roles
// The key type is defined by TKey
public abstract class IdentityUserContext<
    TUser, TKey, TUserClaim, TUserLogin, TUserToken> : DbContext
        where TUser : IdentityUser<TKey>
        where TKey : IEquatable<TKey>
        where TUserClaim : IdentityUserClaim<TKey>
        where TUserLogin : IdentityUserLogin<TKey>
        where TUserToken : IdentityUserToken<TKey>
{
}
```

自定义模型

模型自定义的起始点是派生自相应的上下文类型。请参阅泛型类型的模型部分。此上下文类型通常称为 `ApplicationDbContext` 和创建的 ASP.NET Core 模板。

使用的上下文来将模型配置为通过两种方式：

- 提供实体和键类型作为泛型类型参数。
- 重写 `OnModelCreating` 若要修改这些类型的映射。

重写时 `OnModelCreating`，`base.OnModelCreating` 应首先调用; 重写配置，应调用下一步。EF Core 通常具有配置的最后一个 wins 策略。例如，如果 `ToTable` 的实体类型的方法首先调用具有一个表名称，并用再说一遍更高版本使用不同的表名称，第二次调用中的表名。

自定义用户数据

自定义用户数据支持通过继承 `IdentityUser`。此类型命名惯例做法 `ApplicationUser`：

```
public class ApplicationUser : IdentityUser
{
    public string CustomTag { get; set; }
}
```

使用 `ApplicationUser` 作为上下文的泛型参数的类型：

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

无需重写 `OnModelCreating` 在 `ApplicationDbContext` 类。EF Core 映射 `CustomTag` 按照约定的属性。但是，需要更新，以创建新数据库 `CustomTag` 列。若要创建列，请添加迁移时，并如中所述，然后更新数据库标识和 EF Core 迁移。

更新 `Startup.ConfigureServices` 以使用新 `ApplicationUser` 类：

```
services.AddDefaultIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultUI();
```

在 ASP.NET Core 2.1 或更高版本，作为一个 Razor 类库提供标识。有关详细信息，请参阅ASP.NET Core 项目中的基架标识。因此，前面的代码需要调用AddDefaultUI。如果标识基架用于标识文件添加到项目，删除对调用 `AddDefaultUI` 。有关详细信息，请参见：

- 基架标识
- 添加、下载和删除到标识的自定义用户数据

```
services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
```

```
services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext, Guid>()
        .AddDefaultTokenProviders();
```

更改主键类型

为 PK 列的数据类型创建数据库后更改是很多数据库系统有问题。更改 PK 通常涉及删除并重新创建表。因此，密钥类型时，应指定在初始迁移创建的数据库。

请按照下列步骤更改 PK 类型：

1. 如果数据库已创建的 PK 更改之前，运行 `Drop-Database` (PMC) 或 `dotnet ef database drop` (.NET Core CLI) 将其删除。

2. 在确认删除数据库，删除以进行初始迁移 `Remove-Migration` (PMC) 或 `dotnet ef migrations remove` (.NET Core CLI)。

3. 更新 `ApplicationDbContext` 类进行派生IdentityDbContext<TUser,TRole,TKey>。指定的新类型 `TKey` 。例如，若要使用 `Guid` 密钥类型：

   ```
   public class ApplicationDbContext
       : IdentityDbContext<IdentityUser<Guid>, IdentityRole<Guid>, Guid>
   {
       public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
           : base(options)
       {
       }
   }
   ```

在前面的代码中，泛型类IdentityUser<TKey>和IdentityRole<TKey>必须指定要使用新的密钥类型。

在前面的代码中，泛型类IdentityUser<TKey>和IdentityRole<TKey>必须指定要使用新的密钥类型。

`Startup.ConfigureServices` 必须更新为使用一般用户：

```
services.AddDefaultIdentity<IdentityUser<Guid>>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
```

```
services.AddIdentity<IdentityUser<Guid>, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
```

```
services.AddIdentity<IdentityUser<Guid>, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext, Guid>()
        .AddDefaultTokenProviders();
```

4. 如果自定义 `ApplicationUser` 正在使用类中，更新要继承的类 `IdentityUser` 。例如：

```
using System;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

public class ApplicationUser : IdentityUser<Guid>
{
    public string CustomTag { get; set; }
}
```

```
using System;
using Microsoft.AspNetCore.Identity;

public class ApplicationUser : IdentityUser<Guid>
{
    public string CustomTag { get; set; }
}
```

更新 `ApplicationDbContext` 来引用自定义 `ApplicationUser` 类：

```
public class ApplicationDbContext
    : IdentityDbContext<ApplicationUser, IdentityRole<Guid>, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

注册自定义数据库上下文类添加中的标识服务时 `Startup.ConfigureServices` ：

```
services.AddDefaultIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultUI()
        .AddDefaultTokenProviders();
```

主键的数据类型推断通过分析DbContext对象。

在 ASP.NET Core 2.1 或更高版本，作为一个 Razor 类库提供标识。有关详细信息，请参阅ASP.NET Core 项目中的基架标识。因此，前面的代码需要调用AddDefaultUI。如果标识基架用于标识文件添加到项目，删除对调用 `AddDefaultUI` 。

```
services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
```

主键的数据类型推断通过分析DbContext对象。

```
services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext, Guid>()
        .AddDefaultTokenProviders();
```

AddEntityFrameworkStores方法接受 `TKey` ，该值指示主键的数据类型的类型。

5. 如果自定义 `ApplicationRole` 正在使用类中，更新要继承的类 `IdentityRole<TKey>` 。例如：

```
using System;
using Microsoft.AspNetCore.Identity;

public class ApplicationRole : IdentityRole<Guid>
{
    public string Description { get; set; }
}
```

更新 `ApplicationDbContext` 来引用自定义 `ApplicationRole` 类。例如，下面的类引用自定义 `ApplicationUser` 和自定义 `ApplicationRole` ：

```
using System;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

注册自定义数据库上下文类添加中的标识服务时 `Startup.ConfigureServices` ：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, ApplicationRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultUI()
            .AddDefaultTokenProviders();

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

主键的数据类型推断通过分析DbContext对象。

在 ASP.NET Core 2.1 或更高版本，作为一个 Razor 类库提供标识。有关详细信息，请参阅ASP.NET Core 项目中的基架标识。因此，前面的代码需要调用AddDefaultUI。如果标识基架用于标识文件添加到项目，删除对调用 `AddDefaultUI` 。

```
using System;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

注册自定义数据库上下文类添加中的标识服务时 `Startup.ConfigureServices` :

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, ApplicationRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultTokenProviders();

    services.AddMvc()
        .AddRazorPagesOptions(options =>
        {
            options.Conventions.AuthorizeFolder("/Account/Manage");
            options.Conventions.AuthorizePage("/Account/Logout");
        });

    services.AddSingleton<IEmailSender, EmailSender>();
}
```

主键的数据类型推断通过分析DbContext对象。

```
using System;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

注册自定义数据库上下文类添加中的标识服务时 `Startup.ConfigureServices`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, ApplicationRole>()
            .AddEntityFrameworkStores<ApplicationDbContext, Guid>()
            .AddDefaultTokenProviders();

    services.AddMvc();

    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

AddEntityFrameworkStores方法接受 `TKey`，该值指示主键的数据类型的类型。

添加导航属性

更改关系的模型配置可能会更难于进行其他更改。若要替换现有关系，而不是创建新的其他关系，必须格外小心。具体而言，已更改的关系必须指定相同的外键 (FK) 属性作为现有的关系。例如，之间的关系 `Users` 和 `UserClaims` 是，默认情况下，按如下所示指定：

```
builder.Entity<TUser>(b =>
{
    // Each User can have many UserClaims
    b.HasMany<TUserClaim>()
     .WithOne()
     .HasForeignKey(uc => uc.UserId)
     .IsRequired();
});
```

此关系的 FK 指定为 `UserClaim.UserId` 属性。 `HasMany` 和 `WithOne` 调用不带参数，以创建不使用导航属性的关系。

导航属性添加到 `ApplicationUser`，允许关联 `UserClaims` 若要从用户中引用：

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set; }
}
```

`TKey` 为 `IdentityUserClaim<TKey>` 是用户的 PK 为指定的类型。在这种情况下， `TKey` 是 `string` 因为正在使用默认

值。它具有不的 PK 类型 `UserClaim` 实体类型。

现在，已存在的导航属性，它必须在配置 `OnModelCreating`：

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();
        });
    }
}
```

请注意，完全按照以前，只能使用导航属性调用中指定配置关系 `HasMany` 。

在 EF 模型中，不是数据库仅存在导航属性。因为关系 fk 后未发生更改，这种类型的模型更改不需要更新的数据库。可以通过添加迁移后进行更改选中此项。`Up` 和 `Down` 方法为空。

添加所有用户导航属性

下面的示例作为指南使用上的一节中，在用户配置单向导航属性的所有关系：

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set; }
    public virtual ICollection<IdentityUserLogin<string>> Logins { get; set; }
    public virtual ICollection<IdentityUserToken<string>> Tokens { get; set; }
    public virtual ICollection<IdentityUserRole<string>> UserRoles { get; set; }
}
```

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne()
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne()
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne()
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });
    }
}
```

添加用户和角色导航属性

下面的示例作为指南使用上的一节中，用户和角色上配置所有关系的导航的属性：

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set; }
    public virtual ICollection<IdentityUserLogin<string>> Logins { get; set; }
    public virtual ICollection<IdentityUserToken<string>> Tokens { get; set; }
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationRole : IdentityRole
{
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationUserRole : IdentityUserRole<string>
{
    public virtual ApplicationUser User { get; set; }
    public virtual ApplicationRole Role { get; set; }
}
```

```
public class ApplicationDbContext
    : IdentityDbContext<
        ApplicationUser, ApplicationRole, string,
        IdentityUserClaim<string>, ApplicationUserRole, IdentityUserLogin<string>,
        IdentityRoleClaim<string>, IdentityUserToken<string>>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne()
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne()
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.User)
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });

        modelBuilder.Entity<ApplicationRole>(b =>
        {
            // Each Role can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.Role)
                .HasForeignKey(ur => ur.RoleId)
                .IsRequired();
        });

    }
}
```

注意：

- 此示例还包括 `UserRole` 联接实体，这将需要导航从用户到角色的多对多关系。
- 请记得要更改类型的导航属性以反映该 `ApplicationXxx` 类型现在正在使用而不是 `IdentityXxx` 类型。
- 请记住使用 `ApplicationXxx` 中泛型 `ApplicationContext` 定义。

添加所有导航属性

下面的示例作为指南使用上的一节中，所有实体类型上配置所有关系的导航的属性：

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<ApplicationUserClaim> Claims { get; set; }
    public virtual ICollection<ApplicationUserLogin> Logins { get; set; }
    public virtual ICollection<ApplicationUserToken> Tokens { get; set; }
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationRole : IdentityRole
{
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
    public virtual ICollection<ApplicationRoleClaim> RoleClaims { get; set; }
}

public class ApplicationUserRole : IdentityUserRole<string>
{
    public virtual ApplicationUser User { get; set; }
    public virtual ApplicationRole Role { get; set; }
}

public class ApplicationUserClaim : IdentityUserClaim<string>
{
    public virtual ApplicationUser User { get; set; }
}

public class ApplicationUserLogin : IdentityUserLogin<string>
{
    public virtual ApplicationUser User { get; set; }
}

public class ApplicationRoleClaim : IdentityRoleClaim<string>
{
    public virtual ApplicationRole Role { get; set; }
}

public class ApplicationUserToken : IdentityUserToken<string>
{
    public virtual ApplicationUser User { get; set; }
}
```

```
public class ApplicationDbContext
    : IdentityDbContext<
        ApplicationUser, ApplicationRole, string,
        ApplicationUserClaim, ApplicationUserRole, ApplicationUserLogin,
        ApplicationRoleClaim, ApplicationUserToken>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne(e => e.User)
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne(e => e.User)
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne(e => e.User)
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.User)
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });

        modelBuilder.Entity<ApplicationRole>(b =>
        {
            // Each Role can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.Role)
                .HasForeignKey(ur => ur.RoleId)
                .IsRequired();

            // Each Role can have many associated RoleClaims
            b.HasMany(e => e.RoleClaims)
                .WithOne(e => e.Role)
                .HasForeignKey(rc => rc.RoleId)
                .IsRequired();
        });
    }
}
```

使用复合键

前面几节所示更改的标识模型中使用的密钥类型。更改要使用的复合键的标识密钥模型不支持或建议。使用复合密钥与标识涉及更改标识管理器代码与模型交互的方式。此自定义已超出本文的讨论范围。

更改表/列名称和分面

若要更改表和列的名称，请调用 `base.OnModelCreating` 。然后，添加配置以重写任何默认值。例如，若要更改所有标

识表的名称：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.ToTable("MyUsers");
    });

    modelBuilder.Entity<IdentityUserClaim<string>>(b =>
    {
        b.ToTable("MyUserClaims");
    });

    modelBuilder.Entity<IdentityUserLogin<string>>(b =>
    {
        b.ToTable("MyUserLogins");
    });

    modelBuilder.Entity<IdentityUserToken<string>>(b =>
    {
        b.ToTable("MyUserTokens");
    });

    modelBuilder.Entity<IdentityRole>(b =>
    {
        b.ToTable("MyRoles");
    });

    modelBuilder.Entity<IdentityRoleClaim<string>>(b =>
    {
        b.ToTable("MyRoleClaims");
    });

    modelBuilder.Entity<IdentityUserRole<string>>(b =>
    {
        b.ToTable("MyUserRoles");
    });
}
```

这些示例使用默认标识类型。如果使用的应用类型，如 `ApplicationUser`，配置该类型而不是默认类型。

下面的示例更改某些列名称：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.Property(e => e.Email).HasColumnName("EMail");
    });

    modelBuilder.Entity<IdentityUserClaim<string>>(b =>
    {
        b.Property(e => e.ClaimType).HasColumnName("CType");
        b.Property(e => e.ClaimValue).HasColumnName("CValue");
    });
}
```

某些类型的数据库列可以配置某些*方面*(例如，最大值 `string` 允许的长度)。下面的示例设置多个列的最大长度 `string` 模型中的属性：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.Property(u => u.UserName).HasMaxLength(128);
        b.Property(u => u.NormalizedUserName).HasMaxLength(128);
        b.Property(u => u.Email).HasMaxLength(128);
        b.Property(u => u.NormalizedEmail).HasMaxLength(128);
    });

    modelBuilder.Entity<IdentityUserToken<string>>(b =>
    {
        b.Property(t => t.LoginProvider).HasMaxLength(128);
        b.Property(t => t.Name).HasMaxLength(128);
    });
}
```

将映射到不同的架构

跨数据库提供程序，架构的行为可能有所不同。对于 SQL Server，默认值是创建中的所有表*dbo*架构。可以在不同的架构中创建的表。例如：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.HasDefaultSchema("notdbo");
}
```

延迟加载

在本部分中，添加了支持的身份标识模型中的延迟加载代理。延迟加载很有用，因为它允许使用而无需首先确保它们正在加载导航属性。

实体类型可适用于采用多种方式的延迟加载中所述EF Core 文档。为简单起见，使用该软件需要延迟加载代理：

- 安装-Microsoft.EntityFrameworkCore.Proxies包。
- 调用UseLazyLoadingProxies内AddDbContext<TContext>。
- 公共实体类型与 `public virtual` 导航属性。

下面的示例演示如何调用 `UseLazyLoadingProxies` 在 `Startup.ConfigureServices`：

```
services
    .AddDbContext<ApplicationDbContext>(
        b => b.UseSqlServer(connectionString)
              .UseLazyLoadingProxies())
    .AddDefaultIdentity<ApplicationUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

请参阅前面的示例将导航属性添加到实体类型有关的指南。

其他资源

- ASP.NET Core 项目中的基架标识

用于 ASP.NET Core 社区 OSS 身份验证选项

2018/7/4 • • Edit Online

此页包含用于 ASP.NET Core 提供社区，开放源代码身份验证选项。此页将定期更新为变得可用的新提供程序。

OSS 身份验证提供程序

下面的列表按字母顺序排列。

NAME	描述
AspNet.Security.OpenIdConnect.Server (ASOS)	ASOS 是一个低级别，协议首先 OpenID Connect 服务器框架为 ASP.NET Core 和 OWIN/Katana。
Cierge	Cierge 是处理用户注册、登录名、配置文件、管理和社交登录名的 OpenID Connect 的服务器。
Gluu 服务器	企业就绪后，打开源代码软件标识，访问管理 (IAM) 和单一登录 (SSO)。有关详细信息，请参阅Gluu 产品文档。
IdentityServer	IdentityServer 是 OpenID Connect 和 OAuth 2.0 的框架，用于 ASP.NET Core，正式认证 OpenID foundation 和.NET Foundation 监管下。有关详细信息，请参阅欢迎 IdentityServer4（文档）。
OpenIddict	OpenIddict 是能够轻松使用 OpenID Connect 服务器 ASP.NET Core。

若要添加提供程序，编辑此页。

配置 ASP.NET Core 标识

2018/10/10 • • Edit Online

ASP.NET Core 标识设置，例如密码策略、锁定和 cookie 配置使用默认值。在中，可以重写这些设置 `Startup` 类。

标识选项

IdentityOptions类表示可用于标识系统配置的选项。 `IdentityOptions` 必须设置后调用 `AddIdentity` 或 `AddDefaultIdentity` 。

声明标识

IdentityOptions.ClaimsIdentity指定ClaimsIdentityOptions与下表中所示的属性。

属性	描述	默认
RoleClaimType	获取或设置用于为角色声明的声明类型。	ClaimTypes.Role
SecurityStampClaimType	获取或设置用于安全戳声明的声明类型。	`AspNet.Identity.SecurityStamp`
UserIdClaimType	获取或设置用于的用户标识符声明的声明类型。	ClaimTypes.NameIdentifier
UserNameClaimType	获取或设置用于用户名称声明的声明类型。	ClaimTypes.Name

锁定

在中设置锁定PasswordSignInAsync方法：

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(Input.Email,
            Input.Password, Input.RememberMe,
            lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new { ReturnUrl = returnUrl,
                Input.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

前面的代码基于 `Login` 标识模板。

在中设置锁定选项 `StartUp.ConfigureServices`：

```
services.Configure<IdentityOptions>(options =>
{
    // Default Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;
});
```

上述代码会设置IdentityOptions LockoutOptions具有默认值。

成功的身份验证失败的访问尝试计数重置并重置时钟。

IdentityOptions.Lockout指定LockoutOptions与表中所示的属性。

属性	描述	默认
AllowedForNewUsers	确定是否新用户会被锁定。	`true`
DefaultLockoutTimeSpan	时间量用户已锁定时在锁定时发生。	5 分钟
MaxFailedAccessAttempts	用户已被锁定，如果启用了锁定前的失败的访问尝试数。	5

Password

默认情况下，标识要求密码包含大写字符、小写字符、数字、和非字母数字字符。密码必须至少为六个字符。PasswordOptions可以设置 `Startup.ConfigureServices` 。

```
services.Configure<IdentityOptions>(options =>
{
    // Default Password settings.
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;
});
```

```
services.AddIdentity<ApplicationUser, IdentityRole>(options =>
    {
        // Password settings
        options.Password.RequireDigit = true;
        options.Password.RequiredLength = 8;
        options.Password.RequiredUniqueChars = 2;
        options.Password.RequireLowercase = true;
        options.Password.RequireNonAlphanumeric = true;
        options.Password.RequireUppercase = true;
    })
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

```
services.Configure<IdentityOptions>(options =>
{
    // Password settings
    options.Password.RequireDigit = true;
    options.Password.RequiredLength = 8;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireUppercase = true;
    options.Password.RequireLowercase = false;
});
```

IdentityOptions.Password指定PasswordOptions与表中所示的属性。

属性	描述	默认
RequireDigit	需要介于 0-9 的密码。	`true`
RequiredLength	密码最小长度。	6
RequireLowercase	要求密码中的小写字符。	`true`
RequireNonAlphanumeric	需要在密码中的非字母数字字符。	`true`
RequiredUniqueChars	仅适用于 ASP.NET Core 2.0 或更高版本。 要求在密码中非重复字符数。	1
RequireUppercase	需要大写字符的密码。	`true`

属性	描述	默认
RequireDigit	需要介于 0-9 的密码。	`true`
RequiredLength	密码最小长度。	6
RequireLowercase	要求密码中的小写字符。	`true`
RequireNonAlphanumeric	需要在密码中的非字母数字字符。	`true`
RequireUppercase	需要大写字符的密码。	`true`

单一登录

下面的代码设置 `SignIn`（为默认值）的设置：

```
services.Configure<IdentityOptions>(options =>
{
    // Default SignIn settings.
    options.SignIn.RequireConfirmedEmail = true;
    options.SignIn.RequireConfirmedPhoneNumber = false;
});
```

```
services.AddIdentity<ApplicationUser, IdentityRole>(options =>
    {
        // Signin settings
        options.SignIn.RequireConfirmedEmail = true;
        options.SignIn.RequireConfirmedPhoneNumber = false;
    })
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

IdentityOptions.SignIn指定SignInOptions与表中所示的属性。

属性	描述	默认
RequireConfirmedEmail	需要已确认的电子邮件，登录。	`false`
RequireConfirmedPhoneNumber	需要确认的电话号码进行登录。	`false`

标记

IdentityOptions.Tokens指定TokenOptions与表中所示的属性。

属性	描述
AuthenticatorTokenProvider	获取或设置 `AuthenticatorTokenProvider` 用于验证身份验证器使用双因素登录名。
ChangeEmailTokenProvider	获取或设置 `ChangeEmailTokenProvider` 用于生成电子邮件更改确认电子邮件中使用的令牌。
ChangePhoneNumberTokenProvider	获取或设置 `ChangePhoneNumberTokenProvider` 用于生成令牌更改电话号码时使用。

属性	描述
EmailConfirmationTokenProvider	获取或设置用于生成帐户确认电子邮件中使用的令牌的令牌提供程序。
PasswordResetTokenProvider	获取或设置IUserTwoFactorTokenProvider 用于生成在密码重置电子邮件中使用的令牌。
ProviderMap	用于构造用户令牌提供程序与键用作提供程序的名称。

“用户”

```
services.Configure<IdentityOptions>(options =>
{
    // Default User settings.
    options.User.AllowedUserNameCharacters =
            "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = true;

});
```

IdentityOptions.User指定UserOptions与表中所示的属性。

属性	描述	默认
AllowedUserNameCharacters	在用户名中允许的字符。	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789 -._@+
RequireUniqueEmail	要求每个用户必须拥有唯一的电子邮件。	`false`

Cookie 设置

配置中的应用程序的 cookie `Startup.ConfigureServices`。ConfigureApplicationCookie必须在调用后调用 `AddIdentity` 或 `AddDefaultIdentity`。

```
services.ConfigureApplicationCookie(options =>
{
    options.AccessDeniedPath = "/Identity/Account/AccessDenied";
    options.Cookie.Name = "YourAppCookieName";
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(60);
    options.LoginPath = "/Identity/Account/Login";
    // ReturnUrlParameter requires
    //using Microsoft.AspNetCore.Authentication.Cookies;
    options.ReturnUrlParameter = CookieAuthenticationDefaults.ReturnUrlParameter;
    options.SlidingExpiration = true;
});
```

```
services.ConfigureApplicationCookie(options =>
{
    options.AccessDeniedPath = "/Account/AccessDenied";
    options.Cookie.Name = "YourAppCookieName";
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(60);
    options.LoginPath = "/Account/Login";
    // ReturnUrlParameter requires `using Microsoft.AspNetCore.Authentication.Cookies;`
    options.ReturnUrlParameter = CookieAuthenticationDefaults.ReturnUrlParameter;
    options.SlidingExpiration = true;
});
```

```
services.Configure<IdentityOptions>(options =>
{
    // Cookie settings
    options.Cookies.ApplicationCookie.CookieName = "YourAppCookieName";
    options.Cookies.ApplicationCookie.ExpireTimeSpan = TimeSpan.FromDays(150);
    options.Cookies.ApplicationCookie.LoginPath = "/Account/LogIn";
    options.Cookies.ApplicationCookie.AccessDeniedPath = "/Account/AccessDenied";
    options.Cookies.ApplicationCookie.AutomaticAuthenticate = true;
    // Requires `using Microsoft.AspNetCore.Authentication.Cookies;`
    options.Cookies.ApplicationCookie.AuthenticationScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.Cookies.ApplicationCookie.ReturnUrlParameter = CookieAuthenticationDefaults.ReturnUrlParameter;
});
```

有关详细信息，请参阅CookieAuthenticationOptions。

在 ASP.NET Core 中配置 Windows 身份验证

2019/1/8 • • Edit Online

通过Scott Addie和Luke Latham

Windows 身份验证可以配置为使用托管的 ASP.NET Core 应用IIS或HTTP.sys。

Windows 身份验证依赖于操作系统的 ASP.NET Core 应用的用户进行身份验证。使用 Active Directory 域标识或 Windows 帐户标识的用户在公司网络上运行你的服务器时，可以使用 Windows 身份验证。Windows 身份验证是最适合于用户、客户端应用程序和 web 服务器属于同一个 Windows 域的 intranet 环境。

启用 Windows 身份验证在 ASP.NET Core 应用

Web 应用程序模板可通过 Visual Studio 或.NET Core CLI 可以配置为支持 Windows 身份验证。

- Visual Studio
- .NET Core CLI

新的项目中使用 Windows 身份验证应用程序模板

在 Visual Studio 中：

1. 创建一个新**ASP.NET Core Web 应用程序**。
2. 选择**Web 应用程序**从模板列表。
3. 选择**更改身份验证**按钮，然后选择**Windows 身份验证**。

运行应用。用户名将显示在呈现的应用程序用户界面。

手动配置为现有项目的

项目的属性，您可以启用 Windows 身份验证并禁用匿名身份验证：

1. 右键单击 Visual Studio 中的项目**解决方案资源管理器**，然后选择**属性**。
2. 选择“调试”选项卡。
3. 清除的复选框**启用匿名身份**验证。
4. 选中的复选框**启用 Windows 身份**验证。

或者，可以在配置属性`iisSettings`的节点*launchSettings.json*文件：

```
"iisSettings": {
    "windowsAuthentication": true,
    "anonymousAuthentication": false,
    "iisExpress": {
        "applicationUrl": "http://localhost:52171/",
        "sslPort": 0
    }
}
```

启用与 IIS Windows 身份验证

IIS 使用ASP.NET Core 模块到承载 ASP.NET Core 应用程序。Windows 身份验证配置为通过 IIS *web.config*文件。以下部分介绍如何：

- 提供本地*web.config*在部署应用时在服务器激活 Windows 身份验证的文件。
- 使用 IIS 管理器配置*web.config*已部署到服务器的 ASP.NET Core 应用的文件。

IIS 配置

如果尚未执行此操作，使 IIS 能够承载 ASP.NET Core 应用程序。有关详细信息，请参阅 使用 IIS 在 Windows 上托管 ASP.NET Core。

启用 Windows 身份验证的 IIS 角色服务。有关详细信息，请参阅IIS 角色服务（请参阅步骤 2）中启用 Windows 身份验证。

默认情况下，IIS 集成中间件配置为自动进行身份验证请求。有关详细信息，请参阅与 IIS 的 Windows 上托管 ASP.NET Core:IIS 选项 (AutomaticAuthentication)。

默认情况下，ASP.NET Core 模块配置为转发到应用的 Windows 身份验证令牌。有关详细信息，请参阅ASP.NET Core 模块配置参考：AspNetCore 元素的特性。

创建新的 IIS 站点

指定的名称和文件夹，并允许它创建新的应用程序池。

启用 IIS 中应用的 Windows 身份验证

使用任一以下方法之一：

- 发布应用程序之前对开发端配置(*推荐*)
- 发布应用后端服务器的配置

开发端配置与本地 web.config 文件

执行以下步骤**之前**您发布和部署你的项目。

添加以下*web.config*到项目根目录的文件：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <security>
        <authentication>
          <anonymousAuthentication enabled="false" />
          <windowsAuthentication enabled="true" />
          </authentication>
        </security>
    </system.webServer>
  </location>
</configuration>
```

由 SDK 发布项目时 (而无需 `<IsTransformWebConfigDisabled>` 属性设置为 `true` 项目文件中)，已发布*web.config*文件包括 `<location><system.webServer><security><authentication>` 部分。有关详细信息 `<IsTransformWebConfigDisabled>` 属性，请参阅使用 IIS 在 Windows 上托管 ASP.NET Core。

服务器端配置使用 IIS 管理器

执行以下步骤**后**您发布和部署你的项目。

1. 在 IIS 管理器中，选择下的 IIS 站点**站点**的节点连**接**侧栏。
2. 双击**身份**验证中**IIS**区域。
3. 选择**匿名身份**验证。选择**禁用**中**操作**侧栏。
4. 选择**Windows 身份**验证。选择**启用**中**操作**侧栏。

IIS 管理器时执行这些操作，会应用的修改*web.config*文件。一个 `<system.webServer><security><authentication>` 使用更新的设置的添加节点 `anonymousAuthentication` 和 `windowsAuthentication`：

```
<system.webServer>
  <security>
    <authentication>
      <anonymousAuthentication enabled="false" />
      <windowsAuthentication enabled="true" />
    </authentication>
  </security>
</system.webServer>
```

`<system.webServer>`部分添加到*web.config*文件由 IIS 管理器是在应用外`<location>`发布应用时由.NET Core SDK 添加的部分。因为部分添加之外`<location>`节点中，设置由任何继承子应用到当前应用。若要阻止继承，将在添加`<security>`内的部分`<location><system.webServer>`SDK 提供的部分。

当使用 IIS 管理器添加 IIS 配置时，它只影响应用程序的*web.config*在服务器上的文件。应用程序的后续部署可能会覆盖服务器上的设置，如果服务器的副本*web.config*替换为项目的*web.config*文件。使用任一以下方法来管理的设置之一：

- 使用 IIS 管理器中的设置重置*web.config*文件后部署上覆盖该文件。
- 添加*web.config 文件*向本地使用的设置应用程序。有关详细信息，请参阅开发端配置部分。

发布并将项目部署到 IIS 的站点文件夹

使用 Visual Studio 或.NET Core CLI，发布，并将应用部署到目标文件夹。

有关使用 IIS 承载的详细信息，发布和部署，请参阅以下主题：

- dotnet publish
- 使用 IIS 在 Windows 上托管 ASP.NET Core
- ASP.NET Core 模块
- 用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件

启动应用程序以验证 Windows 身份验证正常工作。

启用 Windows 身份验证使用 HTTP.sys

虽然 Kestrel 不支持 Windows 身份验证，您可以使用HTTP.sys以在 Windows 上支持自承载的方案。下面的示例配置以使用 Windows 身份验证使用 HTTP.sys 的应用程序的 web 主机：

```
public class Program
{
    public static void Main(string[] args) =>
        BuildWebHost(args).Run();

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .UseHttpSys(options =>
            {
                options.Authentication.Schemes =
                    AuthenticationSchemes.NTLM | AuthenticationSchemes.Negotiate;
                options.Authentication.AllowAnonymous = false;
            })
            .Build();
}
```

> **NOTE**
> HTTP.sys 通过 Kerberos 身份验证协议委托给内核模式身份验证。Kerberos 和 HTTP.sys 不支持用户模式身份验证。必须使用计算机帐户来解密从 Active Directory 获取的并由客户端转发到服务器的 Kerberos 令牌/票证，以便对用户进行身份验证。注册主机的服务主体名称 (SPN)，而不是应用的用户。

> **NOTE**
> HTTP.sys 不支持 Nano Server 版本 1709年或更高版本上。若要使用 Windows 身份验证和 HTTP.sys 与 Nano Server，使用Server Core (microsoft/windowsservercore) 容器。在 Server Core 上的详细信息，请参阅什么是 Windows Server 中的服务器核心安装选项？。

使用 Windows 身份验证

匿名访问的配置状态确定的方式 `[Authorize]` 和 `[AllowAnonymous]` 应用中使用属性。以下两个部分介绍如何处理的匿名访问权限的禁止和允许配置状态。

不允许匿名访问

启用 Windows 身份验证，并且禁用了匿名访问，则 `[Authorize]` 和 `[AllowAnonymous]` 特性不起作用。如果 IIS 站点（或 HTTP.sys）配置为不允许匿名访问，请求将永远不会到达您的应用程序。出于此原因，`[AllowAnonymous]` 属性不适用。

允许匿名访问

启用 Windows 身份验证和匿名访问，使用 `[Authorize]` 和 `[AllowAnonymous]` 属性。`[Authorize]` 属性允许你保护部分，该应用程序的真正需要 Windows 身份验证。`[AllowAnonymous]` 特性重写 `[Authorize]` 特性允许匿名访问的应用中的使用情况。请参阅简单授权属性使用情况详细信息。

在 ASP.NET Core 2.x，请 `[Authorize]` 属性需要进行额外配置中的*Startup.cs*以进行 Windows 身份验证质询匿名请求。建议的配置略有根据正在使用的 web 服务器而异。

> **NOTE**
> 默认情况下，缺少授权可访问的页面的用户会显示 HTTP 403 响应为空。StatusCodePages 中间件可以配置为向用户提供更好的"拒绝访问"体验。

IIS

如果使用 IIS，添加以下代码到 `ConfigureServices` 方法：

```
// IISDefaults requires the following import:
// using Microsoft.AspNetCore.Server.IISIntegration;
services.AddAuthentication(IISDefaults.AuthenticationScheme);
```

HTTP.sys

如果使用 HTTP.sys，添加以下代码到 `ConfigureServices` 方法：

```
// HttpSysDefaults requires the following import:
// using Microsoft.AspNetCore.Server.HttpSys;
services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);
```

Impersonation

ASP.NET Core 未实现模拟。应用程序运行具有的所有请求，使用应用程序池或进程标识的应用的标识。如果您需要显式执行代表用户操作，使用WindowsIdentity.RunImpersonated中终端的内联中间件中

`Startup.Configure`。在此上下文中运行的单个操作，然后关闭上下文。

```
app.Run(async (context) =>
{
    try
    {
        var user = (WindowsIdentity)context.User.Identity;

        await context.Response
            .WriteAsync($"User: {user.Name}\tState: {user.ImpersonationLevel}\n");

        WindowsIdentity.RunImpersonated(user.AccessToken, () =>
        {
            var impersonatedUser = WindowsIdentity.GetCurrent();
            var message =
                $"User: {impersonatedUser.Name}\t" +
                $"State: {impersonatedUser.ImpersonationLevel}";

            var bytes = Encoding.UTF8.GetBytes(message);
            context.Response.Body.Write(bytes, 0, bytes.Length);
        });
    }
    catch (Exception e)
    {
        await context.Response.WriteAsync(e.ToString());
    }
});
```

`RunImpersonated` 不支持异步操作，不应该用于复杂的方案。例如，包装整个请求或中间件链不受支持或推荐的。

ASP.NET Core标识的自定义的存储提供程序

2018/10/26 • • Edit Online

作者：Steve Smith

ASP.NET Core标识是一种可扩展系统，可用于创建自定义存储提供程序并将其连接到你的应用。本主题介绍如何创建 ASP.NET Core标识的自定义的存储提供程序。它介绍如何创建自己的存储提供程序的重要概念，但不是分步演练。

查看或下载 GitHub 中的示例。

介绍

默认情况下，ASP.NET Core标识系统中使用实体框架核心的 SQL Server 数据库存储用户信息。对于许多应用程序，此方法也适用。但是，您可能更愿意使用不同的持久性机制或数据架构。例如：

- 您使用Azure 表存储或其他数据存储。
- 对数据库表具有不同的结构。
- 您可能希望使用不同的数据访问方法，如Dapper。

在每个这种情况下，可以为你的存储机制编写自定义提供程序和该提供程序插入您的应用程序。

ASP.NET Core标识包含在 Visual Studio 中使用"单个用户帐户"选项的项目模板中。

在使用.NET Core CLI 时，添加 `-au Individual`：

```
dotnet new mvc -au Individual
dotnet new webapi -au Individual
```

ASP.NET Core标识体系结构

ASP.NET Core标识包含类称为管理器和存储区。*管理器*是高级类应用程序开发人员使用来执行操作，如创建标识用户。*存储*是指定如何保持实体，例如用户和角色，较低级别类。存储遵循存储库模式和紧密结合的持久性机制。管理器被分开存储，这意味着您可以替换持久性机制，而无需更改应用程序代码中的（除配置）。

下图显示了 web 应用与交互的方式管理器，而存储与数据访问层进行交互。

若要创建自定义存储提供程序，请使用此数据访问层（绿色和灰色框上面的关系图中）创建数据源、数据访问层和交互的存储类。不需要自定义管理器或与它们（蓝色框上面）进行交互的应用程序代码。

创建的新实例时 `UserManager` 或 `RoleManager` 提供用户类的类型和作为参数传递存储类的实例。此方法使您可以插入到 ASP.NET Core的自定义的类。

重新配置应用程序以使用新的存储提供程序说明如何实例化 `UserManager` 和 `RoleManager` 使用自定义存储。

ASP.NET Core标识将存储的数据类型

ASP.NET Core标识数据类型进行详细介绍下列各节：

用户

你的网站的已注册的用户。IdentityUser可能扩展类型，或将其用作自定义类型的示例。您不必继承来实现自定义标识存储解决方案的特定类型。

用户声明

一组语句（或声明）有关的用户的表示用户的标识。可以启用的用户的标识不是可以通过角色实现更大的表达式。

用户登录名

有关外部身份验证提供程序（如 Facebook 或 Microsoft 帐户）的信息日志记录使用户登录时使用。示例

角色

你的站点的授权组。包含角色 Id 和角色名称（如"管理员"或"Employee"）。示例

数据访问层

本主题假定你熟悉要使用的持久性机制以及如何创建实体，可用于该机制。本主题不提供有关如何创建存储库或数据访问类; 详细信息使用 ASP.NET Core标识时，它提供了有关设计决策的一些建议。

为自定义的存储提供程序设计的数据访问层时具有的自由度。只需创建要在应用中使用的功能的持久性机制。例如，如果不应用程序中使用角色，你不必创建的角色或用户角色关联的存储。你的技术和现有基础结构可能需要的 ASP.NET Core标识的默认实现不同的结构。在数据访问层，提供用于存储实现的结构的逻辑。

数据访问层提供逻辑来将数据从 ASP.NET Core标识保存到数据源。自定义的存储提供程序的数据访问层可能包括以下类用于存储用户和角色信息。

Context 类

若要连接到持久性机制并执行查询的信息进行封装。多个数据类需要通常情况下通过依赖关系注入提供此类的实例。示例。

用户存储

存储和检索用户信息（如用户名称和密码哈希）。示例

角色存储

存储和检索角色信息（例如角色名称）。示例

UserClaims 存储

存储和检索用户声明信息（如的声明类型和值）。示例

UserLogins 存储

存储和检索用户登录信息（例如外部身份验证提供程序）。示例

UserRole 存储

存储和检索哪些角色分配给哪些用户。示例

提示：仅实现想要使用应用程序中的类。

在数据访问类中，提供代码要执行数据操作的持久性机制。例如，在自定义提供程序，可能会具有以下代码以创建新的用户在*存储*类：

```
public async Task<IdentityResult> CreateAsync(ApplicationUser user,
    CancellationToken cancellationToken = default(CancellationToken))
{
    cancellationToken.ThrowIfCancellationRequested();
    if (user == null) throw new ArgumentNullException(nameof(user));

    return await _usersTable.CreateAsync(user);
}
```

创建用户的实现逻辑位于 `_usersTable.CreateAsync` 方法，如下所示。

自定义用户类

在实现存储提供程序时，创建一个用户类等效于IdentityUser 类。

您的用户类必须包含最低限度 `Id` 和一个 `UserName` 属性。

`IdentityUser` 类定义的属性的 `UserManager` 调用时执行请求的操作。默认类型 `Id` 属性是一个字符串，但可以继承自 `IdentityUser<TKey, TUserClaim, TUserRole, TUserLogin, TUserToken>` 和指定不同的类型。框架需要要处理的数据类型转换的存储实现。

自定义用户存储

创建 `UserStore` 类，该类提供用户的所有数据操作的方法。此类是等效于UserStore<TUser> 类。在你 `UserStore` 类，请实现 `IUserStore<TUser>` 和所需的可选接口。您选择实现的可选接口基于应用程序中提供的功能。

可选接口

- IUserRoleStore
- IUserClaimStore
- IUserPasswordStore
- IUserSecurityStampStore
- IUserEmailStore
- IPhoneNumberStore
- IQueryableUserStore
- IUserLoginStore
- IUserTwoFactorStore
- IUserLockoutStore

可选接口继承自 `IUserStore<TUser>` 。可以看到存储中的部分实现的示例用户示例应用。

在 `UserStore` 类，您使用您创建的用于执行操作的数据访问类。这些通过依赖关系注入进行传递。例如，在使用 Dapper 实现，SQL Server `UserStore` 类具有 `CreateAsync` 使用的实例方法 `DapperUsersTable` 来插入新记录：

```
public async Task<IdentityResult> CreateAsync(ApplicationUser user)
{
    string sql = "INSERT INTO dbo.CustomUser " +
        "VALUES (@id, @Email, @EmailConfirmed, @PasswordHash, @UserName)";

    int rows = await _connection.ExecuteAsync(sql, new { user.Id, user.Email, user.EmailConfirmed,
user.PasswordHash, user.UserName });

    if(rows > 0)
    {
        return IdentityResult.Success;
    }
    return IdentityResult.Failed(new IdentityError { Description = $"Could not insert user {user.Email}." });
}
```

要实现自定义用户存储区时接口

- **IUserStore**
  IUserStore<TUser> 接口是在用户存储中必须实现的唯一接口。它定义用于创建、更新、删除和检索用户的方法。
- **IUserClaimStore**
  IUserClaimStore<TUser> 接口定义实现启用用户声明的方法。它包含用于添加、删除和检索用户声明的方法。
- **IUserLoginStore**
  IUserLoginStore<TUser> 定义实现以启用外部身份验证提供程序的方法。它包含用于添加、删除和检索用户登录名和用于检索用户的登录信息基于方法的方法。
- **IUserRoleStore**
  IUserRoleStore<TUser> 接口定义实现将用户映射到角色的方法。它包含方法以添加、删除和检索用户的角色和一个方法以检查是否将用户分配到角色。
- **IUserPasswordStore**
  IUserPasswordStore<TUser> 接口定义实现保留哈希的密码的方法。它包含用于获取和设置哈希的密码和用于指示用户是否已设置密码的方法的方法。
- **IUserSecurityStampStore**
  IUserSecurityStampStore<TUser> 接口定义实现用于安全戳，该值指示是否已更改用户的帐户信息的方法。用户更改密码，或添加或删除登录名时，将更新此 stamp。它包含方法用于获取和设置的安全戳。
- **IUserTwoFactorStore**
  IUserTwoFactorStore<TUser> 接口定义实现以支持双因素身份验证的方法。它包含用于获取和设置是否为用户启用双因素身份验证方法。

- **IUserPhoneNumberStore**
  IUserPhoneNumberStore<TUser> 接口定义实现来存储用户的电话号码的方法。它包含用于获取和设置的电话号码和电话号码是否已确认的方法。
- **IUserEmailStore**
  IUserEmailStore<TUser> 接口定义实现来存储用户的电子邮件地址的方法。它包含用于获取和设置的电子邮件地址和电子邮件是否已确认的方法。
- **IUserLockoutStore**
  IUserLockoutStore<TUser> 接口定义实现以存储有关锁定的帐户信息的方法。它包含用于跟踪失败的访问尝试和锁定的方法。
- **IQueryableUserStore**
  IQueryableUserStore<TUser> 接口定义实现以提供可查询的用户存储区的成员。

在应用中实现所需的接口。例如：

```
public class UserStore : IUserStore<IdentityUser>,
                         IUserClaimStore<IdentityUser>,
                         IUserLoginStore<IdentityUser>,
                         IUserRoleStore<IdentityUser>,
                         IUserPasswordStore<IdentityUser>,
                         IUserSecurityStampStore<IdentityUser>
{
    // interface implementations not shown
}
```

IdentityUserClaim、IdentityUserLogin 和 IdentityUserRole

`Microsoft.AspNet.Identity.EntityFramework` 命名空间包含的实现IdentityUserClaim，IdentityUserLogin，并且IdentityUserRole类。如果使用这些功能，您可能想要创建这些类的版本并定义您的应用程序的属性。但是，有时它是更高效，若要执行基本操作（如添加或删除用户的声明）时不将这些实体加载到内存。相反后，端存储类可以执行这些操作直接在数据源上。例如，`UserStore.GetClaimsAsync` 方法可以调用 `userClaimTable.FindByUserId(user.Id)` 方法来执行查询，在直接表并返回声明的列表。

自定义角色类

在实现角色存储提供程序时，您可以创建自定义角色类型。无需实现特定接口，但是它必须具有 `Id`，其中包含通常 `Name` 属性。

下面是示例角色类：

```
using System;

namespace CustomIdentityProviderSample.CustomProvider
{
    public class ApplicationRole
    {
        public Guid Id { get; set; } = Guid.NewGuid();
        public string Name { get; set; }
    }
}
```

自定义角色存储

您可以创建 `RoleStore` 类，该类提供对角色的所有数据操作的方法。此类是等效于RoleStore<TRole> 类。在中 `RoleStore` 类，实现 `IRoleStore<TRole>` 和（可选）`IQueryableRoleStore<TRole>` 接口。

- **IRoleStore<TRole>**

IRoleStore<TRole> 接口定义了要在角色存储类中实现的方法。它包含用于创建、更新、删除和检索角色的方法。

- **RoleStore<TRole>**
  若要自定义 `RoleStore`，创建一个实现类 `IRoleStore<TRole>` 接口。

重新配置应用程序以使用新的存储提供程序

已实现存储提供程序之后，你将应用配置为使用它。如果您的应用程序使用默认的提供程序，请将它都替换为自定义提供程序。

1. 删除 `Microsoft.AspNetCore.EntityFramework.Identity` NuGet 包。
2. 如果存储提供程序驻留在单独的项目或包中，添加对它的引用。
3. 替换为对所有引用 `Microsoft.AspNetCore.EntityFramework.Identity` 使用 using 语句的命名空间的存储提供程序。
4. 在中 `ConfigureServices` 方法中，更改 `AddIdentity` 方法以使用自定义类型。您可以创建你自己的扩展方法实现此目的。请参阅IdentityServiceCollectionExtensions有关的示例。
5. 如果使用的角色，更新 `RoleManager` 若要使用你 `RoleStore` 类。
6. 对应用程序的配置更新连接字符串和凭据。

示例:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add identity types
    services.AddIdentity<ApplicationUser, ApplicationRole>()
        .AddDefaultTokenProviders();

    // Identity Services
    services.AddTransient<IUserStore<ApplicationUser>, CustomUserStore>();
    services.AddTransient<IRoleStore<ApplicationRole>, CustomRoleStore>();
    string connectionString = Configuration.GetConnectionString("DefaultConnection");
    services.AddTransient<SqlConnection>(e => new SqlConnection(connectionString));
    services.AddTransient<DapperUsersTable>();

    // additional configuration
}
```

参考资料

- ASP.NET 4.x 标识的自定义存储提供程序
- ASP.NET Core 标识–此存储库包括社区维护存储提供程序的链接。

ASP.NET Core 中的 Facebook、Google 和外部提供程序身份验证

2019/1/9 • • Edit Online

作者：Valeriy Novytskyy 和 Rick Anderson

本教程演示如何生成 ASP.NET Core 2.x 应用，该应用可让用户使用外部身份验证提供程序提供的凭据通过 OAuth 2.0 登录。

以下几节中介绍了 Facebook、Twitter、Google 和 Microsoft 提供程序。第三方程序包中提供了其他提供程序，例如 AspNet.Security.OAuth.Providers 和 AspNet.Security.OpenId.Providers。

使用户能够使用其当前凭据登录对用户来说十分便利，并且这样做可以将管理登录进程许多复杂操作转移给第三方。有关社交登录如何驱动流量和客户转换的示例，请参阅 Facebook 和 Twitter 的案例分析。

创建新的 ASP.NET Core 项目

- 在 Visual Studio 2017 中，从“开始”页创建新项目，或通过“文件”>“新建”>“项目”进行创建 > > 。
- 选择“Visual C#” > “.NET Core”分类中提供的“ASP.NET Core Web 应用程序”模板：

- 点击“Web 应用程序”，验证“身份验证”是否设置为“单个用户帐户”：

注意:本教程适用于 ASP.NET Core 2.0 SDK 版本，可以在向导顶部选择此版本。

应用迁移

- 运行应用并选择"登录"链接。
- 选择"以新用户身份注册"链接。
- 输入新帐户的电子邮件地址和密码，再选择"注册"。
- 按照说明操作来应用迁移。

要求使用 HTTPS

OAuth 2.0 需要使用 SSL/TLS 通过 HTTPS 协议进行身份验证。

使用 Web 应用程序或 Web API 项目模板以及 ASP.NET Core 2.1 或更高版本创建的项目会自动配置为启用 HTTPS。如果在项目向导的"更改身份验证"对话框中选择"单个用户帐户"选项，应用会在安全默认终结点启动。

有关更多信息，请参见强制实施 HTTPS 在 ASP.NET Core。

转发请求使用代理的信息或负载均衡器

如果代理服务器或负载均衡器后面部署应用，原始请求信息的一些可能会转发到请求标头中的应用程序。此信息通常包括安全请求方案（`https`），主机和客户端 IP 地址。应用程序不会自动读取这些请求标头，以发现和使用原始请求信息。

在生成链接会影响与外部提供商的身份验证流中使用方案。丢失的安全方案（`https`）生成不正确的不安全重定向 Url，应用程序。

使用转发头中间件可用于处理请求的应用提供的原始请求信息。

有关详细信息，请参阅 配置 ASP.NET Core 以使用代理服务器和负载均衡器 。

使用 SecretManager 存储登录提供程序分配的令牌

社交登录提供程序在注册过程中分配"应用程序 ID"和"应用程序机密"。确切的令牌名称因提供程序而异。这些

令牌代表应用用来访问其 API 的凭据。令牌构成“机密”，可利用机密管理器将其链接到应用配置。机密管理器是在配置文件（例如 appsettings.json）中存储令牌更安全替代方法。

> **IMPORTANT**
>
> 机密管理器仅用于开发目的。可使用 Azure Key Vault 配置提供程序存储和保护 Azure 测试和生产机密。

按照在 ASP.NET Core 中进行开发期间安全存储应用机密主题中的步骤进行操作，以便存储以下每个登录提供程序分配的令牌。

应用程序所需的安装登录提供程序

使用以下主题配置应用程序，以使用相应的提供程序：

- Facebook 相关说明
- Twitter 相关说明
- Google 相关说明
- Microsoft 相关说明
- 其他提供程序相关说明

多个身份验证提供程序

如果应用需要多个提供程序，请在 AddAuthentication 后面链接提供程序扩展方法：

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

选择性地设置密码

使用外部登录提供程序注册，即表明还没有向应用注册密码。这可让用户无需创建和记住站点密码，但也会使用户依赖外部登录提供程序。如果外部登录提供程序不可用，则无法登录网站。

使用外部提供程序在登录过程中设置的电子邮箱创建密码和登录：

- 点击右上角的“Hello <电子邮件别名>”链接，导航到“管理”视图。

- 点击“创建”

- 设置一个有效密码，可以用此密码和邮箱登录。

后续步骤

- 本文介绍了外部身份验证，并说明了向 ASP.NET Core 应用添加外部登录所需的先决条件。
- 引用特定于提供程序的页，为应用所需的提供程序配置登录。
- 可能需要保留有关用户及其访问和刷新令牌的其他数据。有关更多信息，请参见保留其他声明和 ASP.NET Core 中的外部提供程序颁发令牌。

在 ASP.NET Core Facebook 外部登录安装程序

2018/12/27 • • Edit Online

作者：Valeriy Novytskyy 和 Rick Anderson

本教程演示如何使用户可以使用示例 ASP.NET Core 2.0 项目上创建其 Facebook 帐户登录上一页。我们先按照创建 Facebook 应用程序 ID官方步骤。

在 Facebook 中创建应用

- 导航到Facebook 开发人员应用页上，并登录。如果还没有 Facebook 帐户，使用**注册适用于 Facebook**上登录页后，可以创建一个链接。
- 点击**添加新的应用**中创建新的应用程序 id。在右上角的按钮

- 填写表单，然后点击**创建应用程序 ID**按钮。

- 上选择**一个产品**页上，单击**Set Up**上**Facebook 登录**卡。

- **快速入门**向导将启动与选择一个平台作为第一页。现在跳过该向导，通过单击设置在左侧菜单中的链接：

- 此时会显示客户端 **OAuth** 设置页：

- 输入你的开发 URI 与 */signin-facebook*追加到**有效的 OAuth 重定向 Uri**字段（例如：`https://localhost:44320/signin-facebook`）。本教程中稍后配置 Facebook 身份验证将自动处理在请求 */signin-facebook*路由实现 OAuth 流。

> **NOTE**
> URI */signin-facebook*设置为 Facebook 身份验证提供程序的默认回调。配置 Facebook 身份验证中间件通过继承时，可以更改默认的回调 URI RemoteAuthenticationOptions.CallbackPath的属性FacebookOptions类。

- 单击**保存更改**。
- 单击**设置** > **基本**左侧导航窗格中的链接。

  在此页上，记下你`App ID`和你`App Secret`。你将添加到 ASP.NET Core 应用程序下一节中：

- 部署站点时需要重新访问**Facebook 登**录安装页并注册一个新的公共 URI。

应用商店 Facebook 应用程序 ID 和应用程序密码

链接敏感设置，例如 Facebook`App ID`并`App Secret`到应用程序的配置使用机密管理器。对于本教程的目的，命名为令牌`Authentication:Facebook:AppId`和`Authentication:Facebook:AppSecret`。

执行以下命令来安全地存储`App ID`和`App Secret`使用机密管理器：

```
dotnet user-secrets set Authentication:Facebook:AppId <app-id>
dotnet user-secrets set Authentication:Facebook:AppSecret <app-secret>
```

配置 Facebook 身份验证

将 Facebook 服务中的添加 `ConfigureServices` 中的方法*Startup.cs*文件：

```
services.AddDefaultIdentity<IdentityUser>()
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

services.AddAuthentication().AddFacebook(facebookOptions =>
{
    facebookOptions.AppId = Configuration["Authentication:Facebook:AppId"];
    facebookOptions.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
});
```

在调用AddIdentity配置的默认方案设置。AddAuthentication(String)重载集DefaultScheme属性。AddAuthentication (操作<AuthenticationOptions>)重载允许配置身份验证选项，可用于设置针对不同目的的默认身份验证方案。对后续调用 `AddAuthentication` 以前配置的重写AuthenticationOptions属性。

AuthenticationBuilder注册身份验证处理程序的扩展方法只能调用一次每个身份验证方案。重载存在允许配置方案属性、方案名称和显示名称。

多个身份验证提供程序

如果应用需要多个提供程序，请在 AddAuthentication 后面链接提供程序扩展方法：

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

安装Microsoft.AspNetCore.Authentication.Facebook包。

- 若要使用 Visual Studio 2017 中安装此包，请右键单击项目并选择**管理 NuGet 包**。
- 若要使用.NET Core CLI 安装，请在项目目录中执行以下命令：

  `dotnet add package Microsoft.AspNetCore.Authentication.Facebook`

添加在到 Facebook 中间件 `Configure` 中的方法*Startup.cs*文件：

```
app.UseFacebookAuthentication(new FacebookOptions()
{
    AppId = Configuration["Authentication:Facebook:AppId"],
    AppSecret = Configuration["Authentication:Facebook:AppSecret"]
});
```

请参阅FacebookOptions Facebook 身份验证支持的配置选项的详细信息的 API 参考。配置选项可用于：

- 请求有关用户的不同信息。
- 添加查询字符串参数以自定义登录体验。

使用 Facebook 登录

运行你的应用程序，然后单击登录。请参阅使用 Facebook 登录的选项。

当您单击**Facebook**，你将重定向到 Facebook 进行身份验证：

Facebook 身份验证请求的默认公共配置文件和电子邮件地址：

输入你的 Facebook 凭据后你重定向回你的站点，你可以设置你的电子邮件。

现在已在使用 Facebook 凭据登录：

转发请求使用代理的信息或负载均衡器

如果代理服务器或负载均衡器后面部署应用，原始请求信息的一些可能会转发到请求标头中的应用程序。此信息通常包括安全请求方案（`https`），主机和客户端 IP 地址。应用程序不会自动读取这些请求标头，以发现和使用原始请求信息。

在生成链接会影响与外部提供商的身份验证流中使用方案。丢失的安全方案（`https`）生成不正确的不安全重定向 Url，应用程序。

使用转发头中间件可用于处理请求的应用提供的原始请求信息。

有关详细信息，请参阅 配置 ASP.NET Core 以使用代理服务器和负载均衡器 。

疑难解答

- **ASP.NET Core 仅限 2.x:** 如果不通过调用配置标识 `services.AddIdentity` 中 `ConfigureServices`，尝试进行身份验证将导致*ArgumentException:必须提供 SignInScheme 选项*。在本教程中使用的项目模板可确保，此操作。

- 如果尚未通过应用初始迁移创建站点数据库，则获取*处理请求时，数据库操作失败*错误。点击应**用迁移**创建数据库，并刷新以忽略错误继续。

后续步骤

- 添加Microsoft.AspNetCore.Authentication.Facebook到 Facebook 身份验证的高级方案的项目的 NuGet 包。此包不需要将 Facebook 外部登录功能与您的应用程序集成。
- 本文介绍了您如何可以使用 Facebook 进行验证。可以遵循类似的方法来使用上列出其他提供程序进行身份验证上一页。
- 一旦您将您的网站发布到 Azure web 应用时，应重置 `AppSecret` Facebook 开发人员门户中。
- 设置 `Authentication:Facebook:AppId` 和 `Authentication:Facebook:AppSecret` 作为在 Azure 门户中的应用程序设置。配置系统设置以从环境变量读取密钥。

Twitter 外部登录名与 ASP.NET Core 的安装程序

2018/12/27 • • Edit Online

作者：Valeriy Novytskyy 和 Rick Anderson

本教程演示如何启用到你的用户使用 Twitter 帐户登录上使用示例 ASP.NET Core 2.0 项目创建上一页。

在 Twitter 中创建应用程序

- 导航到 https://apps.twitter.com/ 并登录。如果还没有 Twitter 帐户，使用 **立即注册** 链接创建一个链接。登录后，应**用程序管理**页所示：

- 点击创**建新的**应**用程序**，应用程序中填写**名称**，说**明**和公共**网站**URI（可以是临时直到注册的域名）：

- 输入你的开发 URI 与 `/signin-twitter` 追加到**有效的 OAuth 重定向 Uri**字段（例如：`https://localhost:44320/signin-twitter`）。本教程中稍后配置 Twitter 身份验证方案将自动处理在请求 `/signin-twitter` 路由实现 OAuth 流。

> **NOTE**
>
> URI 段 `/signin-twitter` 设置为 Twitter 身份验证提供程序的默认回调。配置 Twitter 身份验证中间件通过继承时，可以更改默认的回调 URI RemoteAuthenticationOptions.CallbackPath的属性TwitterOptions类。

- 填写窗体的其余部分，然后点击**创建 Twitter 应用程序**。显示新的应用程序详细信息：

- 部署站点时将需要重新访问**应用程序管理**页上，并注册一个新的公共 URI。

存储 Twitter ConsumerKey 和 ConsumerSecret

链接敏感设置，例如 Twitter `Consumer Key` 并 `Consumer Secret` 到应用程序的配置使用机密管理器。对于本教程的目的，命名为令牌 `Authentication:Twitter:ConsumerKey` 和 `Authentication:Twitter:ConsumerSecret` 。

这些令牌可在**密钥和访问令牌**选项卡后创建新的 Twitter 应用程序：

配置 Twitter 身份验证

在本教程中使用的项目模板可确保Microsoft.AspNetCore.Authentication.Twitter包已安装。

- 若要使用 Visual Studio 2017 中安装此包，请右键单击项目并选择**管理 NuGet 包**。
- 若要使用.NET Core CLI 安装，请在项目目录中执行以下命令：

```
dotnet add package Microsoft.AspNetCore.Authentication.Twitter
```

将 Twitter 服务中的添加`ConfigureServices`中的方法*Startup.cs*文件：

```
services.AddDefaultIdentity<IdentityUser>()
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

services.AddAuthentication().AddTwitter(twitterOptions =>
{
    twitterOptions.ConsumerKey = Configuration["Authentication:Twitter:ConsumerKey"];
    twitterOptions.ConsumerSecret = Configuration["Authentication:Twitter:ConsumerSecret"];
});
```

在调用AddIdentity配置的默认方案设置。AddAuthentication(String)重载集DefaultScheme属性。AddAuthentication (操作<AuthenticationOptions>)重载允许配置身份验证选项，可用于设置针对不同目的的默认身份验证方案。对后续调用`AddAuthentication`以前配置的重写AuthenticationOptions属性。

AuthenticationBuilder注册身份验证处理程序的扩展方法只能调用一次每个身份验证方案。重载存在允许配置方案属性、方案名称和显示名称。

多个身份验证提供程序

如果应用需要多个提供程序，请在 AddAuthentication 后面链接提供程序扩展方法：

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

添加中的 Twitter 中间件 `Configure` 中的方法*Startup.cs*文件：

```
app.UseTwitterAuthentication(new TwitterOptions()
{
    ConsumerKey = Configuration["Authentication:Twitter:ConsumerKey"],
    ConsumerSecret = Configuration["Authentication:Twitter:ConsumerSecret"]
});
```

请参阅TwitterOptions Twitter 身份验证支持的配置选项的详细信息的 API 参考。这可以用于请求有关用户的不同信息。

使用 Twitter 登录

运行你的应用程序，然后单击登录。通过 Twitter 进行登录的选项将显示：

单击**Twitter**将重定向到 Twitter 进行身份验证：

输入你的 Twitter 凭据后，你将重定向回 web 站点，你可以设置你的电子邮件。

现在已在使用你的 Twitter 凭据进行登录：

转发请求使用代理的信息或负载均衡器

如果代理服务器或负载均衡器后面部署应用，原始请求信息的一些可能会转发到请求标头中的应用程序。此信息通常包括安全请求方案（`https`），主机和客户端 IP 地址。应用程序不会自动读取这些请求标头，以发现和使用原始请求信息。

在生成链接会影响与外部提供商的身份验证流中使用方案。丢失的安全方案（`https`）生成不正确的不安全重定向 Url，应用程序。

使用转发头中间件可用于处理请求的应用提供的原始请求信息。

有关详细信息，请参阅 配置 ASP.NET Core 以使用代理服务器和负载均衡器 。

疑难解答

- **ASP.NET Core 仅限 2.x:** 如果不通过调用配置标识 `services.AddIdentity` 中 `ConfigureServices`，尝试进行身份验证将导致*ArgumentException:必须提供 SignInScheme 选项*。在本教程中使用的项目模板可确保，此操作。
- 如果尚未通过应用初始迁移创建站点数据库，则会收到*处理请求时，数据库操作失败*错误。点击应**用迁移**创建数据库，并刷新以忽略错误继续。

后续步骤

- 本文介绍了您如何可以使用 Twitter 进行验证。可以遵循类似的方法来使用上列出其他提供程序进行身份验证上一页。
- 一旦您将您的网站发布到 Azure web 应用时，应重置 `ConsumerSecret` Twitter 开发人员门户中。
- 设置 `Authentication:Twitter:ConsumerKey` 和 `Authentication:Twitter:ConsumerSecret` 作为在 Azure 门户中的应用程序设置。配置系统设置以从环境变量读取密钥。

在 ASP.NET Core Google 外部登录安装程序

2018/12/27 • • Edit Online

作者：Valeriy Novytskyy 和 Rick Anderson

本教程演示如何使用户可以使用示例 ASP.NET Core 2.0 项目上创建其 Google + 帐户登录上一页。我们首先需要遵循官方步骤Google API 控制台中创建新的应用程序。

在 Google API 控制台中创建应用

- 导航到 https://console.developers.google.com/projectselector/apis/library 并登录。如果还没有 Google 帐户，使用**更多**选项 > **创建帐户** 链接创建一个链接：

- 将重定向到**API 管理器**库页：

- 点击**创建**并输入你项**目名称**:

- 接受对话框中，你将重定向回到库页允许您选择新应用的功能。查找**Google + API**在列表中，单击其链接以添加 API 功能:

- 显示新添加的 API 的页。点击**启用**将 Google + 登录功能中添加到您的应用程序：

- 启用 API 后，点击创**建凭据**若要配置的机密：

- 选择：
  - **Google + API**
  - **Web 服务器 (例如 node.js，Tomcat)**，和
  - **用户数据**：

- 点击**我需要什么凭据？** 转到应用配置，第二个步骤创**建 OAuth 2.0 客户端 ID**:

- 因为我们将 Google + 项目创建与只是一项功能（登录），我们可以输入相同**名称**与我们的项目使用 OAuth 2.0 客户端 id。
- 输入你的开发 URI 与 `/signin-google` 追加到**已授权重定向 Uri**字段（例如：`https://localhost:44320/signin-google`）。本教程中稍后配置 Google 身份验证将自动处理在请求 `/signin-google` 路由实现 OAuth 流。

> **NOTE**
> URI 段 `/signin-google` 设置为默认的 Google 身份验证提供程序的回调。配置 Google 身份验证中间件通过继承时，可以更改默认的回调 URI RemoteAuthenticationOptions.CallbackPath的属性GoogleOptions类。

- 按 tab 键以添加**已授权重定向 Uri**条目。
- 点击**创建客户端 ID**，转到第三个步骤中，其中设置 **OAuth 2.0 许可屏幕**：

- 输入你的公众电子邮件地址并产品名称Google + 提示用户登录时为你的应用所示。其他选项都位于更多自定义选项。
- 点击继续继续到最后一个步骤中，下载凭据：

- 点击**下载**若要使用应用程序机密保存 JSON 文件并**完成**以完成创建新的应用。
- 部署站点时将需要重新访问**Google Console**并注册新的公共 url。

应用商店 Google ClientID 和 ClientSecret

链接敏感设置，例如 Google `Client ID` 并 `Client Secret` 到应用程序的配置使用机密管理器。对于本教程的目的，命名为令牌 `Authentication:Google:ClientId` 和 `Authentication:Google:ClientSecret` 。

可以在下一步中下载的 JSON 文件中找到这些令牌的值 `web.client_id` 和 `web.client_secret` 。

配置 Google 身份验证

将 Google 服务中的添加 `ConfigureServices` 中的方法*Startup.cs*文件：

```
services.AddDefaultIdentity<IdentityUser>()
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId = Configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
});
```

在调用AddIdentity配置的默认方案设置。AddAuthentication(String)重载集DefaultScheme属性。AddAuthentication (操作<AuthenticationOptions>)重载允许配置身份验证选项，可用于设置针对不同目的的默认身份验证方案。对后续调用 `AddAuthentication` 以前配置的重写AuthenticationOptions属性。

AuthenticationBuilder注册身份验证处理程序的扩展方法只能调用一次每个身份验证方案。重载存在允许配置方案属性、方案名称和显示名称。

多个身份验证提供程序

如果应用需要多个提供程序，请在 AddAuthentication 后面链接提供程序扩展方法：

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

在本教程中使用的项目模板可确保Microsoft.AspNetCore.Authentication.Google安装包。

- 若要使用 Visual Studio 2017 中安装此包，请右键单击项目并选择**管理 NuGet 包**。
- 若要使用.NET Core CLI 安装，请在项目目录中执行以下命令：

```
dotnet add package Microsoft.AspNetCore.Authentication.Google
```

添加中的 Google 中间件`Configure`中的方法*Startup.cs*文件：

```
app.UseGoogleAuthentication(new GoogleOptions()
{
    ClientId = Configuration["Authentication:Google:ClientId"],
    ClientSecret = Configuration["Authentication:Google:ClientSecret"]
});
```

请参阅GoogleOptions Google 身份验证支持的配置选项的详细信息的 API 参考。这可以用于请求有关用户的不同信息。

使用 Google 登录

运行你的应用程序，然后单击登录。若要使用 Google 登录的选项将显示：

当您单击 Google 时，你将重定向到 Google 进行身份验证：

后输入你的 Google 凭据，然后你将重定向回 web 站点，你可以设置你的电子邮件。

现在已在使用你的 Google 凭据进行登录：

转发请求使用代理的信息或负载均衡器

如果代理服务器或负载均衡器后面部署应用，原始请求信息的一些可能会转发到请求标头中的应用程序。此信息通常包括安全请求方案（`https`），主机和客户端 IP 地址。应用程序不会自动读取这些请求标头，以发现和使用原始请求信息。

在生成链接会影响与外部提供商的身份验证流中使用方案。丢失的安全方案（`https`）生成不正确的不安全重定向 Url，应用程序。

使用转发头中间件可用于处理请求的应用提供的原始请求信息。

有关详细信息，请参阅 配置 ASP.NET Core 以使用代理服务器和负载均衡器 。

疑难解答

- 如果你收到 `403 (Forbidden)` 从你自己的应用时，请确保在开发模式（或中断到调试器中相同的错误）中正在运行错误页**Google + API**中已启用**API 管理器**库按照列出的步骤本页上早期。如果在登录不起作用，并且如果没有收到任何错误，切换到开发模式，以使问题更易于调试。
- **ASP.NET Core 仅限 2.x:** 如果不通过调用配置标识 `services.AddIdentity` 中 `ConfigureServices` ，尝试进行身份验证将导致*ArgumentException:必须提供 SignInScheme 选项*。在本教程中使用的项目模板可确保，此操作。
- 如果尚未通过应用初始迁移创建站点数据库，则会收到*处理请求时，数据库操作失败*错误。点击应**用迁移**创建数据库，并刷新以忽略错误继续。

后续步骤

- 本文介绍了您如何可以使用 Google 进行验证。可以遵循类似的方法来使用上列出其他提供程序进行身份验证上一页。
- 一旦您将您的网站发布到 Azure web 应用时，应重置 `ClientSecret` 在 Google API 控制台中。
- 设置 `Authentication:Google:ClientId` 和 `Authentication:Google:ClientSecret` 作为在 Azure 门户中的应用程序设置。配置系统设置以从环境变量读取密钥。

使用 ASP.NET Core 的 Microsoft 帐户外部登录设置

2018/12/27 • • Edit Online

作者：Valeriy Novytskyy 和 Rick Anderson

本教程演示如何使用户可以使用示例 ASP.NET Core 2.0 项目上创建其 Microsoft 帐户登录上一页。

在 Microsoft 开发人员门户中创建应用

- 导航到 https://apps.dev.microsoft.com 和创建或登录到 Microsoft 帐户：

如果还没有 Microsoft 帐户，请点击 **创建一个 ！** 在登录后你将重定向到 **我的应用程序** 页：

- 点击**将应用添加**右上角，然后输入你应**用程序名称**并联**系人电子邮件**：

- 对于本教程的目的，清除**指导式设置**复选框。
- 点击**创建**以继续进入**注册**页。提供**名称**并记下的值应**用程序 Id**，其中使用即 `ClientId` 在本教程的后面：

- 点击**添加平台**中**平台**部分，并选择**Web**平台：

- 在新**Web**平台部分中，输入与你开发的 URL `/signin-microsoft` 追加到**重定向 Url**字段（例如：`https://localhost:44320/signin-microsoft`）。稍后在本教程中配置的 Microsoft 身份验证方案将自动处理在请求 `/signin-microsoft` 路由实现 OAuth 流：

> **NOTE**
>
> URI 段 `/signin-microsoft` 设置为默认的 Microsoft 身份验证提供程序的回调。配置 Microsoft 身份验证中间件通过继承时，可以更改默认的回调 URI RemoteAuthenticationOptions.CallbackPath属性的MicrosoftAccountOptions类。

- 点击**添加 URL**以确保该 URL 已添加。
- 填写任何其他应用程序设置，如有必要，并点击**保存**底部的页后，可以将更改保存到应用配置。
- 部署站点时将需要重新访问**注册**页，并将新的公共 URL。

Microsoft 应用程序 Id 和密码存储

- 请注意 `Application Id` 上显示**注册**页。
- 点击**生成新密**码中应**用程序机密**部分。这将显示一个框，您可以在其中复制应用程序密码：

链接敏感设置，例如 Microsoft `Application ID` 并 `Password` 到应用程序的配置使用机密管理器。对于本教程的目的，命名为令牌 `Authentication:Microsoft:ApplicationId` 和 `Authentication:Microsoft:Password` 。

配置 Microsoft 帐户身份验证

在本教程中使用的项目模板可确保Microsoft.AspNetCore.Authentication.MicrosoftAccount包已安装。

- 若要使用 Visual Studio 2017 中安装此包，请右键单击项目并选择**管理 NuGet 包**。
- 若要使用.NET Core CLI 安装，请在项目目录中执行以下命令：

  `dotnet add package Microsoft.AspNetCore.Authentication.MicrosoftAccount`

添加 Microsoft 帐户服务中的 `ConfigureServices` 中的方法*Startup.cs*文件：

```
services.AddDefaultIdentity<IdentityUser>()
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

services.AddAuthentication().AddMicrosoftAccount(microsoftOptions =>
{
    microsoftOptions.ClientId = Configuration["Authentication:Microsoft:ApplicationId"];
    microsoftOptions.ClientSecret = Configuration["Authentication:Microsoft:Password"];
});
```

在调用AddIdentity配置的默认方案设置。AddAuthentication(String)重载集DefaultScheme属性。AddAuthentication (操作<AuthenticationOptions>)重载允许配置身份验证选项，可用于设置针对不同目的的默认身份验证方案。对后续调用 `AddAuthentication` 以前配置的重写AuthenticationOptions属性。

AuthenticationBuilder注册身份验证处理程序的扩展方法只能调用一次每个身份验证方案。重载存在允许配置方案属性、方案名称和显示名称。

多个身份验证提供程序

如果应用需要多个提供程序，请在 AddAuthentication 后面链接提供程序扩展方法：

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

添加中的 Microsoft 帐户中间件 `Configure` 中的方法*Startup.cs*文件：

```
app.UseMicrosoftAccountAuthentication(new MicrosoftAccountOptions()
{
    ClientId = Configuration["Authentication:Microsoft:ApplicationId"],
    ClientSecret = Configuration["Authentication:Microsoft:Password"]
});
```

尽管 Microsoft 开发人员门户上使用的术语命名这些令牌 `ApplicationId` 和 `Password` ，它们作为公开 `ClientId` 和 `ClientSecret` 配置 API。

请参阅MicrosoftAccountOptions支持的 Microsoft 帐户身份验证的配置选项的详细信息的 API 参考。这可以用于请求有关用户的不同信息。

使用 Microsoft 帐户登录

运行你的应用程序，然后单击登录。若要使用 Microsoft 登录的选项将显示：

当单击 Microsoft 时，将重定向的身份验证到 Microsoft。（如果尚未登录），使用你的 Microsoft 帐户登录后您将会提示您允许应用访问你的信息：

点击是，并且将重定向回 web 站点，你可以设置你的电子邮件。

现在已登录中使用 Microsoft 凭据：

转发请求使用代理的信息或负载均衡器

如果代理服务器或负载均衡器后面部署应用，原始请求信息的一些可能会转发到请求标头中的应用程序。此信息通常包括安全请求方案（`https`），主机和客户端 IP 地址。应用程序不会自动读取这些请求标头，以发现和使用原始请求信息。

在生成链接会影响与外部提供商的身份验证流中使用方案。丢失的安全方案（`https`）生成不正确的不安全重定向 Url，应用程序。

使用转发头中间件可用于处理请求的应用提供的原始请求信息。

有关详细信息，请参阅 配置 ASP.NET Core 以使用代理服务器和负载均衡器 。

疑难解答

- 如果 Microsoft 帐户提供程序将你重定向到登录错误页，请记下错误标题和说明查询字符串后面的参数直接 `#`（井号标签）的 Uri 中。

  错误消息似乎指出了使用 Microsoft 身份验证问题，尽管最常见的原因是你的应用程序 Uri 不匹配的任何重定向 Uri指定为Web平台.

- **ASP.NET Core 仅限 2.x:** 如果不通过调用配置标识 `services.AddIdentity` 中 `ConfigureServices`，尝试进行身份验证将导致*ArgumentException:必须提供 SignInScheme 选项*。在本教程中使用的项目模板可确保，此操作。

- 如果尚未通过应用初始迁移创建站点数据库，则会收到*处理请求时，数据库操作失败*错误。点击应用迁移创建数据库，并刷新以忽略错误继续。

后续步骤

- 本文介绍了如何向 Microsoft。可以遵循类似的方法来使用上列出其他提供程序进行身份验证上一页。
- 一旦您将您的网站发布到 Azure web 应用时，应创建一个新 `Password` Microsoft 开发人员门户中。
- 设置 `Authentication:Microsoft:ApplicationId` 和 `Authentication:Microsoft:Password` 作为在 Azure 门户中的应用程序设置。配置系统设置以从环境变量读取密钥。

外部 OAuth 身份验证提供程序

2018/11/16 • • Edit Online

通过Rick Anderson， Pranav rastogi 撰写，和Valeriy Novytskyy

以下列表包含常见外部 OAuth 身份验证提供程序可使用 ASP.NET Core 应用。第三方 NuGet 包，如所维护的那些aspnet contrib，可以用于补充由 ASP.NET Core 团队实现的身份验证提供程序。

- LinkedIn (说明)
- Instagram (说明)
- Reddit (说明)
- Github (说明)
- Yahoo (说明)
- Tumblr (说明)
- Pinterest (说明)
- Pocket (说明)
- Flickr (说明)
- Dribble (说明)
- Vimeo (说明)
- SoundCloud (说明)
- VK (说明)

多个身份验证提供程序

如果应用需要多个提供程序，请在 AddAuthentication 后面链接提供程序扩展方法：

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

转发请求使用代理的信息或负载均衡器

如果代理服务器或负载均衡器后面部署应用，原始请求信息的一些可能会转发到请求标头中的应用程序。此信息通常包括安全请求方案 (`https`)，主机和客户端 IP 地址。应用程序不会自动读取这些请求标头，以发现和使用原始请求信息。

在生成链接会影响与外部提供商的身份验证流中使用方案。丢失的安全方案 (`https`) 生成不正确的不安全重定向 Url，应用程序。

使用转发头中间件可用于处理请求的应用提供的原始请求信息。

有关详细信息，请参阅 配置 ASP.NET Core 以使用代理服务器和负载均衡器 。

保留其他声明和 ASP.NET Core 中的外部提供程序颁发令牌

2018/11/16 • • Edit Online

作者：Luke Latham

ASP.NET Core 应用可以建立其他声明和来自外部身份验证提供程序，如 Facebook、Google、Microsoft 和 Twitter 令牌。每个提供程序会显示不同用户的信息在其平台上，但接收并将用户数据转换成其他声明的模式是相同的。

查看或下载示例代码（如何下载）

系统必备

决定哪些外部身份验证提供程序支持应用程序中。对于每个提供程序，注册应用程序和获取客户端 ID 和客户端机密。有关详细信息，请参阅 ASP.NET Core 中的 Facebook、Google 和外部提供程序身份验证 。示例应用使用Google 身份验证提供程序。

设置客户端 ID 和客户端机密

OAuth 身份验证提供程序使用客户端 ID 和客户端机密的应用与建立信任关系。客户端 ID 和客户端密钥值的应用的外部身份验证提供程序时创建应用注册到提供程序。提供程序的客户端 ID 和客户端机密，必须单独配置每个应用程序使用的外部提供程序。有关详细信息，请参阅适用于方案的外部身份验证提供程序主题：

- Facebook 身份验证
- Google 身份验证
- Microsoft 身份验证
- Twitter 身份验证
- 其他身份验证提供程序
- OpenIdConnect

示例应用程序使用客户端 ID 和 Google 提供的客户端机密配置 Google 身份验证提供程序：

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXX.apps.googleusercontent.com";
    // Provide the Google Secret
    options.ClientSecret = "g4GZ2#...GD5Gg1x";
    options.Scope.Add("https://www.googleapis.com/auth/plus.login");
    options.ClaimActions.MapJsonKey(ClaimTypes.Gender, "gender");
    options.SaveTokens = true;
    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens()
            as List<AuthenticationToken>;
        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });
        ctx.Properties.StoreTokens(tokens);
        return Task.CompletedTask;
    };
});
```

建立身份验证范围

指定要从提供程序检索由指定的权限列表Scope。下表中显示常见的外部提供程序的身份验证作用域。

提供程序	范围
Facebook	`https://www.facebook.com/dialog/oauth`
Google	`https://www.googleapis.com/auth/plus.login`
Microsoft	`https://login.microsoftonline.com/common/oauth2/v2.0/authorize`
Twitter	`https://api.twitter.com/oauth/authenticate`

该示例应用将添加 Google `plus.login` 作用域以请求登录 Google + 中的权限：

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXX.apps.googleusercontent.com";
    // Provide the Google Secret
    options.ClientSecret = "g4GZ2#...GD5Gg1x";
    options.Scope.Add("https://www.googleapis.com/auth/plus.login");
    options.ClaimActions.MapJsonKey(ClaimTypes.Gender, "gender");
    options.SaveTokens = true;
    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens()
            as List<AuthenticationToken>;
        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });
        ctx.Properties.StoreTokens(tokens);
        return Task.CompletedTask;
    };
});
```

将用户数据键映射和创建声明

在提供程序的选项中，指定MapJsonKey要阅读在登录的应用程序标识的外部提供程序的 JSON 用户数据中的每个键。声明类型的详细信息，请参阅ClaimTypes。

示例应用创建Gender将来自声明 `gender` Google 用户数据中的键：

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXX.apps.googleusercontent.com";
    // Provide the Google Secret
    options.ClientSecret = "g4GZ2#...GD5Gg1x";
    options.Scope.Add("https://www.googleapis.com/auth/plus.login");
    options.ClaimActions.MapJsonKey(ClaimTypes.Gender, "gender");
    options.SaveTokens = true;
    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens()
            as List<AuthenticationToken>;
        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });
        ctx.Properties.StoreTokens(tokens);
        return Task.CompletedTask;
    };
});
```

在中OnPostConfirmationAsync、一个IdentityUser(`ApplicationUser`)登录到应用程序与SignInAsync。在登录过程中，UserManager<TUser>可以存储`ApplicationUser`声明的用户数据可从Principal。

在示例应用`OnPostConfirmationAsync`(*Account/ExternalLogin.cshtml.cs*)建立Gender声明的签名中`ApplicationUser`：

```
public async Task<IActionResult> OnPostConfirmationAsync(
    string returnUrl = null)
{
    if (ModelState.IsValid)
    {
        // Get the information about the user from the external login
        // provider
        var info = await _signInManager.GetExternalLoginInfoAsync();

        if (info == null)
        {
            throw new ApplicationException(
                "Error loading external login data during confirmation.");
        }

        var user = new ApplicationUser
            {
                UserName = Input.Email,
                Email = Input.Email
            };
        var result = await _userManager.CreateAsync(user);

        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);

            if (result.Succeeded)
            {
                // Copy over the gender claim
                await _userManager.AddClaimAsync(user,
                    info.Principal.FindFirst(ClaimTypes.Gender));

                // Include the access token in the properties
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);

                await _signInManager.SignInAsync(user, props,
                    authenticationMethod: info.LoginProvider);
                _logger.LogInformation(
                    "User created an account using {Name} provider.",
                    info.LoginProvider);

                return LocalRedirect(Url.GetLocalUrl(returnUrl));
            }
        }

        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    ReturnUrl = returnUrl;

    return Page();
}
```

保存访问令牌

SaveTokens 定义访问和刷新令牌是否应存储在AuthenticationProperties成功授权后。`SaveTokens` 设置为 `false` 默认情况下以减小最终的身份验证 cookie 的大小。

示例应用程序设置的值 `SaveTokens` 到 `true` 中GoogleOptions:

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXX.apps.googleusercontent.com";
    // Provide the Google Secret
    options.ClientSecret = "g4GZ2#...GD5Gg1x";
    options.Scope.Add("https://www.googleapis.com/auth/plus.login");
    options.ClaimActions.MapJsonKey(ClaimTypes.Gender, "gender");
    options.SaveTokens = true;
    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens()
            as List<AuthenticationToken>;
        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });
        ctx.Properties.StoreTokens(tokens);
        return Task.CompletedTask;
    };
});
```

当 `OnPostConfirmationAsync` 执行时，存储的访问令牌 (ExternalLoginInfo.AuthenticationTokens) 中的外部提供程序从 `ApplicationUser` 的 `AuthenticationProperties` 。

示例应用将保存中的访问令牌：

- `OnPostConfirmationAsync` – 执行新的用户注册。
- `OnGetCallbackAsync` – 在以前已注册的用户登录到应用时执行。

Account/ExternalLogin.cshtml.cs:

```
public async Task<IActionResult> OnPostConfirmationAsync(
    string returnUrl = null)
{
    if (ModelState.IsValid)
    {
        // Get the information about the user from the external login
        // provider
        var info = await _signInManager.GetExternalLoginInfoAsync();

        if (info == null)
        {
            throw new ApplicationException(
                "Error loading external login data during confirmation.");
        }

        var user = new ApplicationUser
            {
                UserName = Input.Email,
                Email = Input.Email
            };
        var result = await _userManager.CreateAsync(user);

        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);

            if (result.Succeeded)
            {
                // Copy over the gender claim
                await _userManager.AddClaimAsync(user,
                    info.Principal.FindFirst(ClaimTypes.Gender));

                // Include the access token in the properties
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);

                await _signInManager.SignInAsync(user, props,
                    authenticationMethod: info.LoginProvider);
                _logger.LogInformation(
                    "User created an account using {Name} provider.",
                    info.LoginProvider);

                return LocalRedirect(Url.GetLocalUrl(returnUrl));
            }
        }

        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    ReturnUrl = returnUrl;

    return Page();
}
```

```
public async Task<IActionResult> OnGetCallbackAsync(
    string returnUrl = null, string remoteError = null)
{
    if (remoteError != null)
    {
        ErrorMessage = $"Error from external provider: {remoteError}";

        return RedirectToPage("./Login");
    }

    var info = await _signInManager.GetExternalLoginInfoAsync();

    if (info == null)
    {
        return RedirectToPage("./Login");
    }

    // Sign in the user with this external login provider if the user
    // already has a login
    var result = await _signInManager.ExternalLoginSignInAsync(
        info.LoginProvider, info.ProviderKey, isPersistent: false,
        bypassTwoFactor : true);

    if (result.Succeeded)
    {
        // Store the access token and resign in so the token is included in
        // in the cookie
        var user = await _userManager.FindByLoginAsync(info.LoginProvider,
            info.ProviderKey);

        var props = new AuthenticationProperties();
        props.StoreTokens(info.AuthenticationTokens);

        await _signInManager.SignInAsync(user, props, info.LoginProvider);

        _logger.LogInformation(
            "{Name} logged in with {LoginProvider} provider.",
            info.Principal.Identity.Name, info.LoginProvider);

        return LocalRedirect(Url.GetLocalUrl(returnUrl));
    }

    if (result.IsLockedOut)
    {
        return RedirectToPage("./Lockout");
    }
    else
    {
        // If the user does not have an account, then ask the user to
        // create an account
        ReturnUrl = returnUrl;
        LoginProvider = info.LoginProvider;

        if (info.Principal.HasClaim(c => c.Type == ClaimTypes.Email))
        {
            Input = new InputModel
            {
                Email = info.Principal.FindFirstValue(ClaimTypes.Email)
            };
        }

        return Page();
    }
}
```

如何添加其他自定义令牌

若要演示如何添加作为的一部分存储的自定义令牌 `SaveTokens`，示例应用添加AuthenticationToken与当前DateTime有关AuthenticationToken.Name的 `TicketCreated`：

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXX.apps.googleusercontent.com";
    // Provide the Google Secret
    options.ClientSecret = "g4GZ2#...GD5Gg1x";
    options.Scope.Add("https://www.googleapis.com/auth/plus.login");
    options.ClaimActions.MapJsonKey(ClaimTypes.Gender, "gender");
    options.SaveTokens = true;
    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens()
            as List<AuthenticationToken>;
        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });
        ctx.Properties.StoreTokens(tokens);
        return Task.CompletedTask;
    };
});
```

示例应用程序说明

示例应用演示了如何：

- 从 Google 获取用户的性别和存储的值的性别声明。
- 将 Google 访问令牌存储在用户的 `AuthenticationProperties` 。

若要使用示例应用程序：

1. 注册应用程序和获取有效的客户端 ID 和 Google 身份验证的客户端机密。有关详细信息，请参阅 在 ASP.NET Core Google 外部登录安装程序 。
2. 提供的客户端 ID 和客户端密钥对中的应用程序GoogleOptions的 `Startup.ConfigureServices` 。
3. 运行应用并请求我声明页。如果用户未登录，该应用将重定向到 Google。使用 Google 登录。Google 将用户重定向回应用程序（ `/Home/MyClaims` ）。用户进行身份验证，并加载我的声明页。不存在下性别声明**用户声明**使用从 Google 获取的值。访问令牌显示在**身份**验证**属性**。

```
User Claims

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier
    b36a7b09-9135-4810-b7a5-78697ff23e99
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
    username@gmail.com
AspNet.Identity.SecurityStamp
    29G2TB881ATCUQFJSRFG1S0QJ0OOAWVT
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/gender
    female
http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationmethod
    Google

Authentication Properties

.Token.access_token
    bv42.Dgw...GQMv9ArLPs
.Token.token_type
    Bearer
.Token.expires_at
    2018-08-27T19:08:00.0000000+00:00
.Token.TicketCreated
    8/27/2018 6:08:00 PM
.TokenNames
    access_token;token_type;expires_at;TicketCreated
.issued
    Mon, 27 Aug 2018 18:08:05 GMT
.expires
    Mon, 10 Sep 2018 18:08:05 GMT
```

转发请求使用代理的信息或负载均衡器

如果代理服务器或负载均衡器后面部署应用，原始请求信息的一些可能会转发到请求标头中的应用程序。此信息通常包括安全请求方案（`https`），主机和客户端 IP 地址。应用程序不会自动读取这些请求标头，以发现和使用原始请求信息。

在生成链接会影响与外部提供商的身份验证流中使用方案。丢失的安全方案（`https`）生成不正确的不安全重定向 Url，应用程序。

使用转发头中间件可用于处理请求的应用提供的原始请求信息。

有关详细信息，请参阅 配置 ASP.NET Core 以使用代理服务器和负载均衡器 。

使用 WS 联合身份验证在 ASP.NET Core 中的用户进行身份验证

2018/7/6 • • Edit Online

本教程演示如何使用户能够使用 WS 联合身份验证提供程序 (如 Active Directory 联合身份验证服务 (ADFS) 登录或Azure Active Directory (AAD)。它使用 ASP.NET Core 2.0 示例应用程序中所述Facebook、Google、和外部提供程序身份验证。

对于 ASP.NET Core 2.0 应用程序中，WS 联合身份验证的支持由提供Microsoft.AspNetCore.Authentication.WsFederation。此组件移植从Microsoft.Owin.Security.WsFederation和共享许多该组件的机制。但是，组件的几个重要方面不同。

默认情况下，新的中间件：

- 不允许未经请求的登录名。这一功能的 WS 联合身份验证协议容易受到 XSRF 攻击。但是，可以使用启用此 `AllowUnsolicitedLogins` 选项。
- 不会检查消息登录的每个窗体发布请求。只请求在到 `CallbackPath` 登录程序。检查 `CallbackPath` 默认为 `/signin-wsfed` 但可以更改通过继承RemoteAuthenticationOptions.CallbackPath属性WsFederationOptions类。此路径可以与其他身份验证提供程序共享通过启用SkipUnrecognizedRequests选项。

向 Active Directory 注册应用程序

Active Directory 联合身份验证服务

- 打开服务器的**添加信赖方信任向**导从 ADFS 管理控制台：

- 选择了手动输入数据：

- 输入信赖方的显示名称。名称并不重要到 ASP.NET Core 应用程序。
- Microsoft.AspNetCore.Authentication.WsFederation缺少用于令牌加密，因此未配置令牌加密证书的支持：

- 启用支持 WS 联合身份验证被动协议，使用应用的 URL。验证端口正确应用：

> **NOTE**
>
> 这必须是 HTTPS URL。在开发期间托管应用程序时，IIS Express 可以提供自签名的证书。Kestrel 需要手动证书配置。请参阅Kestrel 文档有关详细信息。

- 单击**下一步**完成该向导的其余部分和**关闭**末尾。
- ASP.NET Core 标识需要**名称 ID**声明。添加一个从编辑**声明**规则对话框：

- 在**添加**转换**声明**规则**向**导，保留默认**以声明方式**发**送 LDAP 属性**选择的模板，然后单击**下一步**。添加规则映射**SAM** 帐户**名** LDAP 属性到**名称 ID**传出声明：

- 单击**完成** > **确定**中编辑**声明**规则窗口。

Azure Active Directory

- 导航到 AAD 租户的应用注册边栏选项卡。单击**新应用程序注册**:

- 输入应用程序注册的名称。这并不重要到 ASP.NET Core 应用程序。
- 输入作为侦听的应用程序的 URL登录 **URL**:

- 单击终结**点并**记下联**合元数据文档**URL。这是 WS 联合身份验证中间件的 `MetadataAddress`：

- 导航到新的应用程序注册。单击**设置** > **属性**并记下应**用程序 ID URI**。这是 WS 联合身份验证中间件的 `Wtrealm`：

为 ASP.NET Core 标识的外部登录提供程序中添加 WS 联合身份验证

- 添加的依赖项Microsoft.AspNetCore.Authentication.WsFederation到项目。
- 添加 WS 联合身份验证到 `Configure` 中的方法*Startup.cs*:

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

services.AddAuthentication()
    .AddWsFederation(options =>
    {
        // MetadataAddress represents the Active Directory instance used to authenticate users.
        options.MetadataAddress = "https://<ADFS FQDN or AAD tenant>/FederationMetadata/2007-
06/FederationMetadata.xml";

        // Wtrealm is the app's identifier in the Active Directory instance.
        // For ADFS, use the relying party's identifier, its WS-Federation Passive protocol URL:
        options.Wtrealm = "https://localhost:44307/";

        // For AAD, use the App ID URI from the app registration's Properties blade:
        options.Wtrealm = "https://wsfedsample.onmicrosoft.com/bf0e7e6d-056e-4e37-b9a6-2c36797b9f01";
    });

services.AddMvc()
 // ...
```

在调用AddIdentity配置的默认方案设置。AddAuthentication(String)重载集DefaultScheme属性。AddAuthentication (操作<AuthenticationOptions>)重载允许配置身份验证选项，可用于设置针对不同目的的默认身份验证方案。对后续调用 `AddAuthentication` 以前配置的重写AuthenticationOptions属性。

AuthenticationBuilder注册身份验证处理程序的扩展方法只能调用一次每个身份验证方案。重载存在允许配置方案属性、方案名称和显示名称。

使用 WS 联合身份验证登录

浏览到应用程序并单击登录nav 标头中的链接。没有用于登录 WsFederation 选项：

使用作为提供程序的 ADFS，按钮将重定向到 ADFS 登录页：

与 Azure Active Directory 作为提供程序，该按钮将重定向到 AAD 登录页：

成功登录的新用户将重定向到应用程序的用户注册页：

使用 WS 联合身份验证而无需 ASP.NET Core 标识

没有标识，可以使用 WS 联合身份验证中间件。例如：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(sharedOptions =>
    {
        sharedOptions.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        sharedOptions.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        sharedOptions.DefaultChallengeScheme = WsFederationDefaults.AuthenticationScheme;
    })
    .AddWsFederation(options =>
    {
        options.Wtrealm = Configuration["wsfed:realm"];
        options.MetadataAddress = Configuration["wsfed:metadata"];
    })
    .AddCookie();
}

public void Configure(IApplicationBuilder app)
{
    app.UseAuthentication();
        // …
}
```

帐户确认和 ASP.NET Core 中的密码恢复

2018/11/2 • • Edit Online

请参阅此 PDF 文件ASP.NET Core 1.1 和 2.1 版本。

作者：Rick Anderson 和 Joe Audette

本教程演示如何生成 ASP.NET Core 应用使用电子邮件确认及密码重置。本教程是不开头主题。您应熟悉：

- ASP.NET Core
- 身份验证
- Entity Framework Core

系统必备

.NET Core 2.1 SDK 或更高版本

创建 web 应用并创建标识的基架

- Visual Studio
- .NET Core CLI

- 在 Visual Studio 中，创建一个新**Web 应用程序**名为项目**WebPWrecover**。
- 选择**ASP.NET Core 2.1**。
- 保留默认值**身份**验证设置为**无身份**验证。下一步中添加身份验证。

在下一步：

- 设置布局页为 *~/Pages/Shared/_Layout.cshtml*
- 选择*帐户/注册*
- 创建一个新**数据上下文**类

按照中的说明启用身份验证:

- 添加 `app.UseAuthentication();` 到 `Startup.Configure`
- 添加 `<partial name="_LoginPartial" />` 布局文件。

测试新的用户注册

运行该应用程序中，选择**注册**链接，并注册用户。在此情况下，电子邮件的唯一验证是使用[EmailAddress]属性。提交后注册，登录到应用程序。更高版本在本教程中，以便验证其电子邮件之前，新用户无法登录。更新代码。

查看标识数据库

- Visual Studio
- .NET Core CLI

- 从视图菜单中，选择**SQL Server 对象资源管理器**(SSOX)。
- 导航到 **(localdb) MSSQLLocalDB (SQL Server 13)**。右键单击**dbo。AspNetUsers** > 查看数据:

请注意此表的 `EmailConfirmed` 字段是 `False` 。

您可能想要再次在下一步中使用此电子邮件，该应用将发送确认电子邮件时。右键单击行并选择**删除**。删除电子邮件别名便于您在以下步骤。

需要电子邮件确认

它是最佳做法以确认新的用户注册的电子邮件地址。电子邮件确认可帮助以验证它们没有模拟其他人（即，他们尚未注册与其他人的电子邮件）。假设有论坛，并且您想阻止"yli@example.com"中注册为"nolivetto@contoso.com"。而无需电子邮件确认"nolivetto@contoso.com"可能会从您的应用程序收到不需要的电子邮件。假设用户意外地注册为"ylo@example.com"和"yli"的拼写错误的重视。它们将无法使用密码恢复，因为此应用没有其正确的电子邮件。电子邮件确认从智能机器人提供有限的保护。电子邮件确认不提供保护，免受恶意用户的与许多电子邮件帐户。

你通常想要发布到网站的任何数据，有已确认的电子邮件之前阻止新用户。

更新*Areas/Identity/IdentityHostingStartup.cs*需要确认电子邮件：

```
public class IdentityHostingStartup : IHostingStartup
{
    public void Configure(IWebHostBuilder builder)
    {
        builder.ConfigureServices((context, services) => {
            services.AddDbContext<WebPWrecoverContext>(options =>
                options.UseSqlServer(
                    context.Configuration.GetConnectionString("WebPWrecoverContextConnection")));

            services.AddDefaultIdentity<IdentityUser>(config =>
                {
                    config.SignIn.RequireConfirmedEmail = true;
                })
                .AddEntityFrameworkStores<WebPWrecoverContext>();
        });
    }
}
```

`config.SignIn.RequireConfirmedEmail = true;` 可防止已注册的用户登录之前确认其电子邮件。

配置电子邮件提供商

在本教程中，SendGrid用于发送电子邮件。你需要一个 SendGrid 帐户和密钥用于发送电子邮件。可以使用其他电子邮件提供商。ASP.NET Core 2.x 包括 `System.Net.Mail`，这允许你从你的应用程序发送电子邮件。我们建议你使用 SendGrid 或另一个电子邮件服务发送电子邮件。SMTP 很难保护并正确设置。

创建一个类来提取的安全电子邮件键。对于此示例中，创建*Services/AuthMessageSenderOptions.cs*:

```
public class AuthMessageSenderOptions
{
        public string SendGridUser { get; set; }
        public string SendGridKey { get; set; }
}
```

配置 SendGrid 用户机密

添加一个唯一 `<UserSecretsId>` 值设为 `<PropertyGroup>` 项目文件的元素：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <UserSecretsId>aspnet-WebPWrecover-1234</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.1.1" />
    <PackageReference Include="SendGrid" Version="9.9.0" />
  </ItemGroup>

</Project>
```

设置 `SendGridUser` 并 `SendGridKey` 与机密管理器工具。例如：

```
C:/WebAppl>dotnet user-secrets set SendGridUser RickAndMSFT
info: Successfully saved SendGridUser = RickAndMSFT to the secret store.
```

在 Windows 中，机密管理器存储中的键/值对*secrets.json*文件中 `%APPDATA%/Microsoft/UserSecrets/<WebAppName-userSecretsId>` 目录。

内容*secrets.json*未加密文件。*Secrets.json*文件如下所示（`SendGridKey`删除值。）

```
{
  "SendGridUser": "RickAndMSFT",
  "SendGridKey": "<key removed>"
}
```

有关详细信息，请参阅选项模式并配置。

安装 SendGrid

本教程演示如何添加通过电子邮件通知SendGrid，但是你可以发送电子邮件使用 SMTP 和其他机制。

安装`SendGrid` NuGet 包：

- Visual Studio
- .NET Core CLI

从包管理器控制台中，输入以下命令：

```
Install-Package SendGrid
```

请参阅免费开始使用 SendGrid注册一个免费的 SendGrid 帐户。

实现 IEmailSender

为实现`IEmailSender`，创建*Services/EmailSender.cs*，类似于以下代码：

```
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.Extensions.Options;
using SendGrid;
using SendGrid.Helpers.Mail;
using System.Threading.Tasks;

namespace WebPWrecover.Services
{
    public class EmailSender : IEmailSender
    {
        public EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public AuthMessageSenderOptions Options { get; } //set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            return Execute(Options.SendGridKey, subject, message, email);
        }

        public Task Execute(string apiKey, string subject, string message, string email)
        {
            var client = new SendGridClient(apiKey);
            var msg = new SendGridMessage()
            {
                From = new EmailAddress("Joe@contoso.com", "Joe Smith"),
                Subject = subject,
                PlainTextContent = message,
                HtmlContent = message
            };
            msg.AddTo(new EmailAddress(email));

            // Disable click tracking.
            // See https://sendgrid.com/docs/User_Guide/Settings/tracking.html
            msg.SetClickTracking(false, false);

            return client.SendEmailAsync(msg);
        }
    }
}
```

将启动以支持电子邮件配置

将以下代码添加到 `ConfigureServices` 中的方法*Startup.cs*文件：

- 添加 `EmailSender` 作为单一实例服务。
- 注册 `AuthMessageSenderOptions` 配置实例。

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // requires
    // using Microsoft.AspNetCore.Identity.UI.Services;
    // using WebPWrecover.Services;
    services.AddSingleton<IEmailSender, EmailSender>();
    services.Configure<AuthMessageSenderOptions>(Configuration);
}
```

启用帐户确认和密码恢复

该模板具有帐户确认和密码恢复的代码。查找 `OnPostAsync` 中的方法*Areas/Identity/Pages/Account/Register.cshtml.cs*。

禁止新注册的用户将自动记录通过注释掉以下行：

```
await _signInManager.SignInAsync(user, isPersistent: false);
```

完整的方法与更改突出显示的行所示：

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = Input.Email, Email = Input.Email };
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with password.");

            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { userId = user.Id, code = code },
                protocol: Request.Scheme);

            await _emailSender.SendEmailAsync(Input.Email, "Confirm your email",
                $"Please confirm your account by <a href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking
here</a>.");

            // await _signInManager.SignInAsync(user, isPersistent: false);
            return LocalRedirect(returnUrl);
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

注册、确认电子邮件，和重置密码

运行 web 应用和测试的帐户确认和密码恢复流。

- 运行应用并注册一个新用户

- 检查你的帐户确认链接的电子邮件。请参阅调试电子邮件如果没有收到电子邮件。
- 单击链接以确认你的电子邮件。
- 电子邮件和密码登录。
- 注销。

查看管理页

在浏览器中选择你的用户名：

您可能需要展开导航栏以查看用户名称。

管理页显示与**配置文件**选定的选项卡。**电子邮件**显示已确认一个复选框，表明该电子邮件。

测试密码重置

- 如果在登录，请选择**注销**。
- 选择**登录**链接，然后选择**忘记了密码？** 链接。
- 输入用于注册该帐户的电子邮件。
- 发送一封电子邮件包含用于重置密码的链接。检查你的电子邮件，并单击链接以重置密码。已成功重置密码后，可以登录你的电子邮件和新密码。

调试电子邮件

如果无法获取电子邮件工作：

- 中设置断点 `EmailSender.Execute` 若要验证 `SendGridClient.SendEmailAsync` 调用。
- 创建控制台应用以发送电子邮件使用类似的代码，到 `EmailSender.Execute` 。
- 审阅电子邮件活动页。
- 请检查垃圾邮件文件夹。
- 请尝试其他电子邮件提供程序（Microsoft、Yahoo、Gmail 等）上的另一个电子邮件别名
- 尝试发送到不同的电子邮件帐户。

最佳安全方案设置为**不**中使用生产机密中测试和开发。如果将应用发布到 Azure，可以为 Azure Web 应用门户中的应用程序设置来设置 SendGrid 机密。配置系统设置以从环境变量读取密钥。

合并的社交和本地登录帐户

若要完成本部分中，必须先启用外部身份验证提供程序。请参阅Facebook、Google、和外部提供程序身份验证。

您可以通过单击电子邮件链接组合本地和社交帐户。按以下顺序"RickAndMSFT@gmail.com"首次创建为本地登录名；但是，您可以创建帐户为社交登录名，然后添加本地登录名。

单击**管理**链接。请注意与此帐户关联的 0 外部（社交登录名）。

单击链接到另一个登录名服务，接受应用程序请求。下图中，在 Facebook 是外部身份验证提供程序：

已合并两个帐户。你将能够使用任一帐户登录。您可能希望用户其社交登录身份验证服务发生故障，或已为其社

交帐户无法访问更有可能的情况下添加本地帐户。

后一个站点中有用户启用帐户确认

与用户启用帐户确认站点上的阻止所有现有用户。现有用户被锁定，因为不确认其帐户。若要解决现有用户锁定，请使用以下方法之一：

- 更新要将标记为正在确认的所有现有用户的数据库。
- 确认现有用户。例如，批的发送确认链接的电子邮件。

启用在 ASP.NET Core TOTP 身份验证器应用的 QR 代码生成

2018/11/10 • • Edit Online

QR 代码需要 ASP.NET Core 2.0 或更高版本。

ASP.NET Core 附带的单个身份验证的身份验证器应用程序的支持。两个身份验证 (2FA) 身份验证器应用程序，使用基于时间的一次性密码算法 (TOTP)，是推荐的方法用于 2FA 的行业。2FA 使用 TOTP 优于 SMS 2FA。身份验证器应用程序提供了哪些用户必须输入其用户名和密码确认后一个 6 到 8 位代码。通常智能手机上安装验证器应用。

ASP.NET Core web 应用程序模板支持身份验证器，但不提供对 QRCode 生成的支持。QRCode 生成器简化 2FA 的安装程序。本文档将指导你完成添加QR 码生成到 2FA 配置页。

将 QR 代码添加到 2FA 配置页

使用这些说明*qrcode.js*从 https://davidshimjs.github.io/qrcodejs/存储库。

- 下载qrcode.js javascript 库到 `wwwroot\lib` 项目文件夹中的。
- 按照中的说明基架标识生成 */Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml*。
- 在中 */Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml*，找到 `Scripts` 文件的末尾部分：
- 在中*Pages/Account/Manage/EnableAuthenticator.cshtml*（Razor 页面）或*Views/Manage/EnableAuthenticator.cshtml*（MVC）中，找到 `Scripts` 文件的末尾部分：

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")
}
```

- 更新 `Scripts` 部分，以添加对引用 `qrcodejs` 您添加的库和生成 QR 代码的调用。其外观应按如下所示：

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")

    <script type="text/javascript" src="~/lib/qrcode.js"></script>
    <script type="text/javascript">
        new QRCode(document.getElementById("qrCode"),
            {
                text: "@Html.Raw(Model.AuthenticatorUri)",
                width: 150,
                height: 150
            });
    </script>
}
```

- 删除的段落，提供了指向这些说明。

运行你的应用，并确保可以扫描 QR 代码并验证身份验证者证明的代码。

更改 QR 代码中的站点名称

QR 代码中的站点名称是来自在最初创建你的项目时选择的项目名称。您可以通过查找对其进行更改

`GenerateQrCodeUri(string email, string unformattedKey)` 中的方法 */Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml*。

QR 代码中的站点名称是来自在最初创建你的项目时选择的项目名称。您可以通过查找对其进行更改 `GenerateQrCodeUri(string email, string unformattedKey)` 中的方法*Pages/Account/Manage/EnableAuthenticator.cshtml.cs*（Razor 页面）文件或*Controllers/ManageController.cs* (MVC) 文件。

模板中的默认代码如下所示：

```
private string GenerateQrCodeUri(string email, string unformattedKey)
{
    return string.Format(
        AuthenicatorUriFormat,
        _urlEncoder.Encode("Razor Pages"),
        _urlEncoder.Encode(email),
        unformattedKey);
}
```

对的调用中的第二个参数 `string.Format` 是你的站点名称，从你的解决方案名称。可以更改为任何值，但它必须始终为 URL 编码。

使用不同的 QR 代码库

QR 代码库可以替换你首选的库。包含 HTML `qrCode` 元素在其中可以通过任何机制将 QR 代码库提供。

QR 码的格式正确 URL 现已推出:

- `AuthenticatorUri` 模型的属性。
- `data-url` 中的属性 `qrCodeData` 元素。

TOTP 客户端和服务器时间倾斜

TOTP（基于时间的一次性密码）身份验证依赖于服务器和身份验证器的设备具有准确的时间。令牌仅持续 30 秒。如果 TOTP 2FA 登录名失败，则检查服务器时间是准确的并且最好是同步到准确的 NTP 服务。

在 ASP.NET Core SMS 的双因素身份验证

2019/1/8 • • Edit Online

通过Rick Anderson和瑞士开发人员

> **WARNING**
>
> 两个身份验证 (2FA) 身份验证器应用程序，使用基于时间的一次性密码算法 (TOTP)，是推荐的方法用于 2FA 的行业。2FA 使用 TOTP 优于 SMS 2FA。有关详细信息，请参阅TOTP 中 ASP.NET Core 的身份验证器应用启用 QR 代码生成ASP.NET Core 2.0 及更高版本。

本教程演示如何设置双因素身份验证 (2FA) 使用短信。说明提供有关twilio并ASPSMS，但你可以使用任何其他 SMS 提供程序。我们建议你先完成帐户确认和密码恢复之前开始学习本教程。

查看或下载示例代码。如何下载。

创建新的 ASP.NET Core 项目

创建新的 ASP.NET Core web 应用名为`Web2FA`与单个用户帐户。按照中的说明强制实施 HTTPS 在 ASP.NET Core设置并要求使用 HTTPS。

创建 SMS 帐户

创建一个 SMS 帐户，例如，从twilio或ASPSMS。记录身份验证凭据 (twilio: accountSid 和 authToken 的 ASPSMS: 用户密钥和密码)。

找出 SMS 提供程序凭据

Twilio: 从你的 Twilio 帐户的仪表板选项卡上，复制帐户 **SID**并身份验证令牌。

ASPSMS: 在帐户设置中，导航到**Userkey**并将其连同复制你密码。

我们将更高版本存储中密钥的机密管理器工具使用这些值`SMSAccountIdentification`和`SMSAccountPassword`。

指定 SenderID / 原始发件人

Twilio: 从数字选项卡，将复制你的 Twilio电话号码。

ASPSMS: 在解锁始发者的菜单中，解锁一个或多个原始发件人或选择字母数字发信方（不支持的所有网络）。

我们稍后将存储此值与中密钥的机密管理器工具`SMSAccountFrom`。

SMS 服务提供的凭据

我们将使用选项模式访问的用户帐户和密钥设置。

- 创建一个类来提取安全 SMS 项。此示例中，对于`SMSoptions`中创建类*Services/SMSoptions.cs*文件。

```
namespace Web2FA.Services
{
    public class SMSoptions
    {
        public string SMSAccountIdentification { get; set; }
        public string SMSAccountPassword { get; set; }
        public string SMSAccountFrom { get; set; }
    }
}
```

设置`SMSAccountIdentification`，`SMSAccountPassword`并`SMSAccountFrom`与机密管理器工具。例如：

```
C:/Web2FA/src/WebApp1>dotnet user-secrets set SMSAccountIdentification 12345
info: Successfully saved SMSAccountIdentification = 12345 to the secret store.
```

- SMS 提供程序添加 NuGet 包。从包管理器控制台 (PMC) 运行：

Twilio: `Install-Package Twilio`

ASPSMS: `Install-Package ASPSMS`

- 将代码中的添加*Services/MessageServices.cs*文件以启用短信。使用 Twilio 或 ASPSMS 部分：

Twilio: [!code-csharp]

ASPSMS: [!code-csharp]

将启动要使用配置 `SMSoptions`

添加`SMSoptions`对中的服务容器`ConfigureServices`中的方法*Startup.cs*:

```
    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
    services.Configure<SMSoptions>(Configuration);
}
```

启用双因素身份验证

打开*Views/Manage/Index.cshtml* Razor 视图文件并删除注释字符（因此没有标记为 commnted out）。

使用双因素身份验证登录

- 运行应用并注册一个新用户

- 点击您的用户名称，这便激活 `Index` 管理控制器中的操作方法。然后点击的电话号码**添加**链接。

- 添加电话号码，它将接收验证代码，并点击发**送**验证码。

- 则会使用验证码的短信。输入它，然后点击**提交**

如果没有收到短信，请参阅 twilio 日志页。

- 管理视图显示已成功添加你的电话号码。

- 点击**启用**若要启用双因素身份验证。

测试双因素身份验证

- 注销。
- 登录。
- 用户帐户已启用双因素身份验证，因此你必须提供身份验证的第二个因素。在本教程中已启用电话验证。内置的模板还可以设置电子邮件作为第二个因素。您可以设置其他身份验证的 QR 代码如第二个因素。点击**提交**。

- 输入你将获得的 SMS 消息中的代码。
- 单击记**住此浏览器**复选框将免除你无需使用 2FA 登录时使用相同的设备和浏览器。启用 2FA，并单击记**住此浏览器**将为您提供强 2FA 保护免受恶意用户尝试访问你的帐户，只要它们不能访问你的设备。可以在您经常使用任何专用设备上执行此操作。通过设置记**住此浏览器**，请从设备不经常使用的情况下，获取 2FA 提高的安全性，您可以方便地在无需在自己的设备上经过 2FA。

用于防范暴力破解攻击的帐户锁定

帐户锁定，建议使用 2FA。用户登录后通过本地帐户或社交帐户，存储在 2FA 每次失败的尝试。如果达到最大的失

败的访问尝试，则用户锁定（默认值：5 分钟后锁定 5 失败访问尝试）。成功的身份验证失败的访问尝试计数重置并重置时钟。最大失败访问尝试，可使用设置锁定时间MaxFailedAccessAttempts并DefaultLockoutTimeSpan。以下 10 分钟后访问尝试失败 10 次配置帐户锁定：

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    services.Configure<IdentityOptions>(options =>
    {
        options.Lockout.MaxFailedAccessAttempts = 10;
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(10);
    });

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
    services.Configure<SMSoptions>(Configuration);
}
```

确认PasswordSignInAsync设置 `lockoutOnFailure` 到 `true`：

```
var result = await _signInManager.PasswordSignInAsync(
                Input.Email, Input.Password, Input.RememberMe, lockoutOnFailure: true);
```

使用 cookie 而无需 ASP.NET Core 标识的身份验证

2019/1/12 • • Edit Online

通过Rick Anderson和Luke Latham

如你所见在早期的身份验证主题中，ASP.NET Core 标识完成、功能完备的身份验证提供程序是用于创建和维护登录名。但是，你可能想要使用基于 cookie 的身份验证有时使用您自己的自定义身份验证逻辑。你可以使用基于 cookie 的身份验证作为独立身份验证提供程序，没有 ASP.NET Core 标识。

查看或下载示例代码（如何下载）

出于演示目的，示例应用程序中，假设用户、Maria Rodriguez 的用户帐户是硬编码到应用程序。使用电子邮件用户名"maria.rodriguez@contoso.com"和任何密码以登录用户。用户进行身份验证中 `AuthenticateUser` 中的方法*Pages/Account/Login.cshtml.cs*文件。在实际示例中，用户会根据数据库身份验证。

有关从 ASP.NET Core 迁移基于 cookie 的身份验证信息 1.x 到 2.0，请参阅迁移身份验证和标识到 ASP.NET Core 2.0 主题（基于 Cookie 的身份验证）。

若要使用 ASP.NET Core 标识，请参阅标识简介主题。

配置

如果应用不使用Microsoft.AspNetCore.App 元包，在项目文件中创建的包引用Microsoft.AspNetCore.Authentication.Cookies包（版本 2.1.0 或更高版本）。

在中 `ConfigureServices` 方法中，创建具有的身份验证中间件服务 `AddAuthentication` 和 `AddCookie` 方法：

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie();
```

`AuthenticationScheme` 传递给 `AddAuthentication` 设置应用程序的默认身份验证方案。`AuthenticationScheme` 有多个实例的 cookie 身份验证并要时很有用与特定方案授权。设置 `AuthenticationScheme` 到 `CookieAuthenticationDefaults.AuthenticationScheme` 方案提供的值为"Cookie"。你可以提供任何字符串值，用于区分方案。

应用程序的身份验证方案是不同的应用程序的 cookie 身份验证方案。当 cookie 身份验证方案不提供给AddCookie，它使用CookieAuthenticationDefaults.AuthenticationScheme ("Cookie")。

在 `Configure` 方法中，使用 `UseAuthentication` 方法调用用于设置 `HttpContext.User` 属性的身份验证中间件。在调用 `UseMvcWithDefaultRoute` 或 `UseMvc` 之前调用 `UseAuthentication` 方法：

```
app.UseAuthentication();
```

AddCookie 选项

CookieAuthenticationOptions类用于配置身份验证提供程序选项。

选项	描述
AccessDeniedPath	提供的路径以提供 302 已找到（URL 重定向）时触发的 `HttpContext.ForbidAsync` 。默认值为 `/Account/AccessDenied` 。

选项	描述
ClaimsIssuer	要用于的颁发者颁发者上创建由 cookie 身份验证服务的任何声明的属性。
Cookie.Domain	Cookie 提供服务位置的域名。默认情况下，这是请求的主机名。在浏览器仅将 cookie 在请求中发送到匹配的主机名。您可能希望调整此项是在你的域中有 cookie 提供给任何主机。例如，将 cookie 域设置为 `.contoso.com` 使其可供 `contoso.com` ， `www.contoso.com` ，和 `staging.www.contoso.com` 。
Cookie.Expiration	获取或设置 cookie 的生命期。目前，此选项没有 ops，并且将会过时中 ASP.NET Core 2.1 +。使用 `ExpireTimeSpan` 选项可以设置 cookie 到期时间。有关详细信息，请参阅阐明行为 CookieAuthenticationOptions.Cookie.Expiration。
Cookie.HttpOnly	指示是否 cookie 应只允许到服务器可访问的标志。此值更改为 `false` 允许客户端脚本访问 cookie，并可能会打开 cookie 被盗应用应您的应用程序具有跨站点脚本 (XSS)漏洞。默认值为 `true` 。
Cookie.Name	设置的 cookie 的名称。
Cookie.Path	用于隔离同一主机名上运行的应用。如果必须在运行的应用 `/app1` 并且想要将 cookie 限制为该应用，设置 `CookiePath` 属性设置为 `/app1` 。通过此操作，cookie 是仅可在请求上 `/app1` 和其下的任何应用。
Cookie.SameSite	指示浏览器是否应允许要附加到同一站点的请求的 cookie (`SameSiteMode.Strict`) 或使用安全的 HTTP 方法和同一站点的请求的跨站点请求 (`SameSiteMode.Lax`)。如果设置为 `SameSiteMode.None` ，未设置的 cookie 标头值。请注意，Cookie 策略中间件可能会覆盖你提供的值。若要支持 OAuth 身份验证，默认值是 `SameSiteMode.Lax` 。有关详细信息，请参阅OAuth 身份验证由于 SameSite cookie 策略中断。
Cookie.SecurePolicy	一个标志，指示是否创建的 cookie 应限制为 HTTPS (`CookieSecurePolicy.Always`)，HTTP 或 HTTPS (`CookieSecurePolicy.None`)，或与请求相同的协议 (`CookieSecurePolicy.SameAsRequest`)。默认值为 `CookieSecurePolicy.SameAsRequest` 。
DataProtectionProvider	集 `DataProtectionProvider` 用于创建默认 `TicketDataFormat` 。如果 `TicketDataFormat` 属性设置，`DataProtectionProvider` 选项不会继续使用。如果未提供，则使用应用程序的默认数据保护提供程序。
事件	该处理程序处理的特定点上提供的应用程序控制的提供程序上调用方法。如果 `Events` 不提供的默认实例提供时才调用这些方法不执行任何操作。
EventsType	用作服务类型，以获取 `Events` 实例而不是属性。

选项	描述
ExpireTimeSpan	`TimeSpan` 后将存储在 cookie 的身份验证票证到期。`ExpireTimeSpan` 添加到当前时间来创建票证的到期时间。`ExpiredTimeSpan` 值始终会进入加密 AuthTicket 验证服务器。此外可能进入Set-cookie标头，但仅当 `IsPersistent` 设置。若要设置 `IsPersistent` 到 `true`，配置AuthenticationProperties传递给 `SignInAsync` 。默认值 `ExpireTimeSpan` 为 14 天。
LoginPath	提供的路径以提供 302 已找到（URL 重定向）时触发的 `HttpContext.ChallengeAsync` 。生成 401 的当前 URL 添加到 `LoginPath` 作为查询字符串参数的名为 `ReturnUrlParameter` 。一次请求 `LoginPath` 授予新登录标识，`ReturnUrlParameter` 值用于将浏览器重定向回导致原始未授权的状态代码的 URL。默认值为 `/Account/Login` 。
LogoutPath	如果 `LogoutPath` 提供给处理程序，则将重定向到该路径的请求值的基础 `ReturnUrlParameter` 。默认值为 `/Account/Logout` 。
ReturnUrlParameter	确定由 302 已找到（URL 重定向）响应的处理程序追加查询字符串参数的名称。`ReturnUrlParameter` 请求到达时，将使用 `LoginPath` 或 `LogoutPath` 来执行登录或注销操作后返回到原始 URL 的浏览器。默认值为 `ReturnUrl` 。
SessionStore	一个可选容器，用于标识存储在请求之间。使用时，只将会话标识符发送到客户端。`SessionStore` 可以用于缓解大型标识的潜在问题。
slidingExpiration	指示是否应动态颁发包含更新的到期时间的新 cookie 的标志。这可以在任何请求，其中当前 cookie 过期时间已超过 50%过期。向前移动新的到期日期为当前日期加上 `ExpireTimespan` 。绝对 cookie 到期时间可以通过使用设置 `AuthenticationProperties` 类调用时 `SignInAsync` 。绝对过期时间可以通过限制身份验证 cookie 是有效的时间量来提高您的应用程序的安全性。默认值为 `true` 。
TicketDataFormat	`TicketDataFormat` 用于保护和取消保护标识和存储在 cookie 值中的其他属性。如果未提供，`TicketDataFormat` 使用创建DataProtectionProvider。
验证	检查的选项是有效的方法。

设置 `CookieAuthenticationOptions` 中的身份验证的服务配置中 `ConfigureServices` 方法：

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        ...
    });
```

ASP.NET Core 1.x 使用 cookie中间件，序列化为一个用户主体的加密 cookie。在后续请求中，对 cookie 进行验证，并重新创建和分配给主体 `HttpContext.User` 属性。

安装Microsoft.AspNetCore.Authentication.Cookies在项目中的 NuGet 包。此包包含 cookie 中间件。

使用 `UseCookieAuthentication` 中的方法 `Configure` 中的方法您*Startup.cs*文件，然后 `UseMvc` 或 `UseMvcWithDefaultRoute`：

```
app.UseCookieAuthentication(new CookieAuthenticationOptions()
{
    AccessDeniedPath = "/Account/Forbidden/",
    AuthenticationScheme = CookieAuthenticationDefaults.AuthenticationScheme,
    AutomaticAuthenticate = true,
    AutomaticChallenge = true,
    LoginPath = "/Account/Unauthorized/"
});
```

CookieAuthenticationOptions 选项

CookieAuthenticationOptions类用于配置身份验证提供程序选项。

选项	描述
AuthenticationScheme	设置身份验证方案。`AuthenticationScheme` 时，可以有多个实例的身份验证，并且你想要与特定方案授权。设置 `AuthenticationScheme` 到 `CookieAuthenticationDefaults.AuthenticationScheme` 方案提供的值为"Cookie"。你可以提供任何字符串值，用于区分方案。
AutomaticAuthenticate	设置一个值，以指示 cookie 身份验证应在每个请求上运行并尝试验证并重新构造它创建的任何序列化的主体。
AutomaticChallenge	如果为 true，则身份验证中间件处理自动挑战。如果为 false，身份验证中间件仅更改时进行显式指定响应 `AuthenticationScheme` 。
ClaimsIssuer	要用于的颁发者颁发者上创建的 cookie 身份验证中间件的任何声明的属性。
CookieDomain	Cookie 提供服务位置的域名。默认情况下，这是请求的主机名。在浏览器只起到匹配的主机名的 cookie。您可能希望调整此项是在你的域中有 cookie 提供给任何主机。例如，将 cookie 域设置为 `.contoso.com` 使其可供 `contoso.com` ，`www.contoso.com` ，和 `staging.www.contoso.com` 。
CookieHttpOnly	指示是否 cookie 应只允许到服务器可访问的标志。此值更改为 `false` 允许客户端脚本访问 cookie，并可能会打开 cookie 被盗应用应您的应用程序具有跨站点脚本 (XSS)漏洞。默认值为 `true` 。
CookiePath	用于隔离同一主机名上运行的应用。如果必须在运行的应用 `/app1` 并且想要将 cookie 限制为该应用，设置 `CookiePath` 属性设置为 `/app1` 。通过此操作，cookie 是仅可在请求上 `/app1` 和其下的任何应用。
CookieSecure	一个标志，指示是否创建的 cookie 应限制为 HTTPS (`CookieSecurePolicy.Always`)，HTTP 或 HTTPS (`CookieSecurePolicy.None`)，或与请求相同的协议 (`CookieSecurePolicy.SameAsRequest`)。默认值为 `CookieSecurePolicy.SameAsRequest` 。

选项	描述
说明	有关提供给应用程序的身份验证类型的其他信息。
ExpireTimeSpan	`TimeSpan` 后将身份验证票证到期。它被添加到当前时间来创建票证的到期时间。若要使用 `ExpireTimeSpan`，则必须设置 `IsPersistent` 到 `true` 中 `AuthenticationProperties` 传递给 `SignInAsync` 。默认值为 14 天。
slidingExpiration	一个标志，指示 cookie 的到期日期重置多个的下半部分 `ExpireTimeSpan` 经过一段时间。新的 exipiration 时间向前移动，为当前日期加上 `ExpireTimespan` 。绝对 cookie 到期时间可以通过使用设置 `AuthenticationProperties` 类调用时 `SignInAsync` 。绝对过期时间可以通过限制身份验证 cookie 是有效的时间量来提高您的应用程序的安全性。默认值为 `true` 。

设置 `CookieAuthenticationOptions` Cookie 身份验证中间件中 `Configure` 方法：

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    ...
});
```

Cookie 策略中间件

Cookie 策略中间件使应用程序中的 cookie 策略功能。到应用程序处理管道添加中间件是顺序敏感;它只影响在管道中注册后的组件。

```
app.UseCookiePolicy(cookiePolicyOptions);
```

CookiePolicyOptions提供给 Cookie 策略中间件可用于控制时追加或删除 cookie 的 cookie 处理和挂钩全局特性 cookie 处理程序。

属性	描述
HttpOnly	会影响是否 cookie 必须 HttpOnly，该值一个标志，指示是否 cookie 应只允许到服务器可访问。默认值为 `HttpOnlyPolicy.None` 。
MinimumSameSitePolicy	影响的 cookie 的同一站点属性（见下文）。默认值为 `SameSiteMode.Lax` 。此选项仅供 ASP.NET Core 2.0 +。
OnAppendCookie	当会追加一个 cookie 时调用。
OnDeleteCookie	当删除 cookie 时调用。
安全	会影响是否 cookie 必须是安全。默认值为 `CookieSecurePolicy.None` 。

MinimumSameSitePolicy (ASP.NET Core 2.0 + 仅)

默认值 `MinimumSameSitePolicy` 值是 `SameSiteMode.Lax` 允许 OAuth2 身份验证。若要严格强制实施的同一站点策略 `SameSiteMode.Strict` ，请将 `MinimumSameSitePolicy` 。尽管此设置会中断的 OAuth2 和其他跨域身份验证方案，它会

提升为其他类型的应用不依赖于跨域请求处理的 cookie 安全级别。

```
var cookiePolicyOptions = new CookiePolicyOptions
{
    MinimumSameSitePolicy = SameSiteMode.Strict,
};
```

Cookie 策略中间件设置 `MinimumSameSitePolicy` 可能会影响您的设置的 `Cookie.SameSite` 中 `CookieAuthenticationOptions` 根据下面的表格的设置。

MINIMUMSAMESITEPOLICY	COOKIE.SAMESITE	最终的 COOKIE.SAMESITE 设置
SameSiteMode.None	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Lax	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Lax SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Strict SameSiteMode.Strict SameSiteMode.Strict

创建身份验证 cookie

若要创建保存用户信息的 cookie，必须构造ClaimsPrincipal。用户信息序列化并存储在 cookie 中。

创建ClaimsIdentity包含任何必需声明s 和调用SignInAsync以登录用户：

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("FullName", user.FullName),
    new Claim(ClaimTypes.Role, "Administrator"),
};

var claimsIdentity = new ClaimsIdentity(
    claims, CookieAuthenticationDefaults.AuthenticationScheme);

var authProperties = new AuthenticationProperties
{
    //AllowRefresh = <bool>,
    // Refreshing the authentication session should be allowed.

    //ExpiresUtc = DateTimeOffset.UtcNow.AddMinutes(10),
    // The time at which the authentication ticket expires. A
    // value set here overrides the ExpireTimeSpan option of
    // CookieAuthenticationOptions set with AddCookie.

    //IsPersistent = true,
    // Whether the authentication session is persisted across
    // multiple requests. Required when setting the
    // ExpireTimeSpan option of CookieAuthenticationOptions
    // set with AddCookie. Also required when setting
    // ExpiresUtc.

    //IssuedUtc = <DateTimeOffset>,
    // The time at which the authentication ticket was issued.

    //RedirectUri = <string>
    // The full path or absolute URI to be used as an http
    // redirect response value.
};

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    authProperties);
```

调用SignInAsync以登录用户：

```
await HttpContext.Authentication.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity));
```

`SignInAsync` 创建一个加密的 cookie，并将其添加到当前响应。如果未指定 `AuthenticationScheme`，则使用默认方案。

事实上，使用的加密是 ASP.NET Core数据保护系统。如果托管应用在多台计算机、负载平衡跨应用程序，或使用 web 场，则必须配置数据保护使用相同密钥环和应用程序标识符。

注销

若要注销当前用户，然后删除其 cookie，调用SignOutAsync:

```
await HttpContext.SignOutAsync(
    CookieAuthenticationDefaults.AuthenticationScheme);
```

若要注销当前用户，然后删除其 cookie，调用SignOutAsync:

```
await HttpContext.Authentication.SignOutAsync(
    CookieAuthenticationDefaults.AuthenticationScheme);
```

如果不使用 `CookieAuthenticationDefaults.AuthenticationScheme`（或"Cookie"）作为方案（例如，"ContosoCookie"），提供配置身份验证提供程序时使用的方案。否则，使用默认方案。

对后端更改做出响应

一旦创建了 cookie，它将成为标识的单个来源。即使在后端系统中禁用用户，cookie 身份验证系统不了解，并且用户保持登录状态，只要其 cookie 有效。

ValidatePrincipal中 ASP.NET Core 事件 2.x 或ValidateAsync中 ASP.NET Core 1.x 可用来截获和重写的 cookie 身份验证方法。此方法可减轻已吊销的用户访问应用的风险。

Cookie 验证的一种方法取决于跟踪的更改用户数据库时。如果数据库未已发生更改，因为发布用户的 cookie，则无需重新验证用户，如果其 cookie 仍然有效。若要实现此方案中，数据库中实现 `IUserRepository` 对于此示例中，存储 `LastChanged` 值。在数据库中，更新的任何用户时 `LastChanged` 值设置为当前时间。

为了使 cookie 无效时的数据库更改基于 `LastChanged` 值时，请创建与 cookie `LastChanged` 包含当前声明 `LastChanged` 值位于数据库中：

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("LastChanged", {Database Value})
};

var claimsIdentity = new ClaimsIdentity(
    claims,
    CookieAuthenticationDefaults.AuthenticationScheme);

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity));
```

若要实现的替代 `ValidatePrincipal` 事件，编写一个方法具有以下签名在从派生类中CookieAuthenticationEvents:

```
ValidatePrincipal(CookieValidatePrincipalContext)
```

示例如下所示：

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;

public class CustomCookieAuthenticationEvents : CookieAuthenticationEvents
{
    private readonly IUserRepository _userRepository;

    public CustomCookieAuthenticationEvents(IUserRepository userRepository)
    {
        // Get the database from registered DI services.
        _userRepository = userRepository;
    }

    public override async Task ValidatePrincipal(CookieValidatePrincipalContext context)
    {
        var userPrincipal = context.Principal;

        // Look for the LastChanged claim.
        var lastChanged = (from c in userPrincipal.Claims
                           where c.Type == "LastChanged"
                           select c.Value).FirstOrDefault();

        if (string.IsNullOrEmpty(lastChanged) ||
            !_userRepository.ValidateLastChanged(lastChanged))
        {
            context.RejectPrincipal();

            await context.HttpContext.SignOutAsync(
                CookieAuthenticationDefaults.AuthenticationScheme);
        }
    }
}
```

在中的 cookie 服务注册期间注册的事件实例 `ConfigureServices` 方法。提供的作用域内的服务注册你 `CustomCookieAuthenticationEvents` 类：

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.EventsType = typeof(CustomCookieAuthenticationEvents);
    });

services.AddScoped<CustomCookieAuthenticationEvents>();
```

若要实现的替代 `ValidateAsync` 事件，编写一个方法具有以下签名：

```
ValidateAsync(CookieValidatePrincipalContext)
```

ASP.NET Core 标识作为的一部分实现这一检查其SecurityStampValidator。示例如下所示：

```
public static class LastChangedValidator
{
    public static async Task ValidateAsync(CookieValidatePrincipalContext context)
    {
        // Pull database from registered DI services.
        var userRepository =
            context.HttpContext.RequestServices
                .GetRequiredService<IUserRepository>();
        var userPrincipal = context.Principal;

        // Look for the last changed claim.
        var lastChanged = (from c in userPrincipal.Claims
                           where c.Type == "LastChanged"
                           select c.Value).FirstOrDefault();

        if (string.IsNullOrEmpty(lastChanged) ||
            !userRepository.ValidateLastChanged(lastChanged))
        {
            context.RejectPrincipal();

            await context.HttpContext.SignOutAsync(
                CookieAuthenticationDefaults.AuthenticationScheme);
        }
    }
}
```

在中的 cookie 身份验证配置期间注册事件 `Configure` 方法：

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    Events = new CookieAuthenticationEvents
    {
        OnValidatePrincipal = LastChangedValidator.ValidateAsync
    }
});
```

请考虑在其中更新用户的名称的情况下—不会影响任何方式的安全决策。如果您需要非破坏性地更新用户主体，请调用 `context.ReplacePrincipal` 并设置 `context.ShouldRenew` 属性设置为 `true` 。

> **WARNING**
>
> 此处所述的方法上的每个请求触发。这可能导致应用程序对较大性能产生负面影响。

永久 cookie

你可能想要在浏览器会话之间持久保存的 cookie。仅进行显式用户同意，有一个"记住我"复选框上登录名或类似机制，应启用此持久性。

下面的代码段创建的标识和相应可以幸存，但通过浏览器闭包的 cookie。遵循以前配置任何滑动过期设置。如果 cookie 已过期浏览器处于关闭状态时，浏览器中清除 cookie 后重新启动它。

```
await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true
    });
```

AuthenticationProperties类驻留在 `Microsoft.AspNetCore.Authentication` 命名空间。

```
await HttpContext.Authentication.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true
    });
```

AuthenticationProperties类驻留在 `Microsoft.AspNetCore.Http.Authentication` 命名空间。

绝对 cookie 到期时间

可以设置使用绝对到期时间 `ExpiresUtc` 。您还必须设置 `IsPersistent` ；否则为 `ExpiresUtc` 将被忽略，并创建一个单会话 cookie。当 `ExpiresUtc` 上设置 `SignInAsync` ，它将重写的值 `ExpireTimeSpan` 的选项 `CookieAuthenticationOptions` ，如果设置。

下面的代码段创建的标识和相应的 cookie 的持续时间为 20 分钟。这会忽略以前配置的任何滑动过期设置。

```
await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true,
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
    });
```

```
await HttpContext.Authentication.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true,
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
    });
```

其他资源

- 身份验证 2.0 更改 / 迁移公告
- 使用 ASP.NET Core 中的特定方案授权
- ASP.NET Core 中基于声明的授权
- 基于策略的角色检查
- 在 Web 场中托管 ASP.NET Core

Azure Active Directory 与 ASP.NET Core

2018/12/10 • • Edit Online

Azure AD V1 示例

下面的示例演示如何集成 Azure AD V1，使用户能够使用工作和学校帐户登录：

- Integrating Azure AD Into an ASP.NET Core Web App（将 Azure AD 集成到 ASP.NET Core Web 应用中）
- Calling a ASP.NET Core Web API From a WPF Application Using Azure AD（从使用 Azure AD 的 WPF 应用程序调用 ASP.NET Core Web API）
- Calling a Web API in an ASP.NET Core Web Application Using Azure AD（在使用 Azure AD 的 ASP.NET Core Web 应用程序中调用 Web API）

Azure AD V2 示例

下面的示例演示如何集成 Azure AD V2，使用户能够使用工作和学校帐户或者 Microsoft 个人帐户（以前称为 Live 帐户）登录：

- Integrating Azure AD V2 into an ASP.NET Core 2.0 web app（将 Azure AD V2 集成到 ASP.NET Core 2.0 Web 应用中）：
  - 请观看这段相关视频
- Calling a ASP.NET Core 2.0 Web API from a WPF application using Azure AD V2（使用 Azure AD V2 从 WPF 应用程序调用 ASP.NET Core 2.0 Web API）：
  - 请观看这段相关视频

Azure AD B2C 示例

此示例演示如何集成 Azure AD B2C，使用户能够使用社交网站身份（如 Facebook、Google...）登录

- 带有 Azure AD B2C 的 ASP.NET Core Web API

Azure Active Directory B2C ASP.NET Core 中使用云身份验证

2019/1/8 • • Edit Online

作者：Cam Soper

Azure Active Directory B2C (Azure AD B2C) 是云标识管理解决方案，适用于 web 和移动应用。该服务提供用于在云中和本地托管的应用的身份验证。身份验证类型包括个人帐户，社交网络帐户和联合企业帐户。此外，Azure AD B2C 可以提供最小配置多重身份验证。

> **TIP**
>
> Azure Active Directory (Azure AD) 和 Azure AD B2C 是单独的产品产品/服务。Azure AD 租户表示组织，而 Azure AD B2C 租户表示与信赖方应用程序将使用的集合。若要了解详细信息，请参阅Azure AD B2C:常见问题 (FAQ)。

在本教程中，学习如何：

- 创建 Azure Active Directory B2C 租户
- 在 Azure AD B2C 中注册应用
- 使用 Visual Studio 创建 ASP.NET Core web 应用配置为使用 Azure AD B2C 租户进行身份验证
- 配置控制的 Azure AD B2C 租户行为的策略

系统必备

本演练需要以下项：

- Microsoft Azure 订阅
- Visual Studio 2017（任何版本）

创建 Azure Active Directory B2C 租户

创建 Azure Active Directory B2C 租户文档中所述。出现提示时，将租户与 Azure 订阅相关联是可选的用于本教程。

在 Azure AD B2C 中注册应用

在新创建的 Azure AD B2C 租户中注册应用程序使用文档中的步骤下**注册 web 应用**部分。在停止创**建 web 应用客户端机密**部分。本教程不需要客户端机密。

使用以下值：

设置	值	说明
名称	*<应用名称>*	输入**名称**描述你的应用向使用者的应用。
包括 web 应用 /web API	是	
允许隐式流	是	

设置	值	说明
回复 URL	`https://localhost:44300/signin-oidc`	回复 Url 属于终结点，Azure AD B2C 在其中返回应用请求的任何令牌。Visual Studio 提供了要使用的回复 URL。现在，输入 `https://localhost:44300/signin-oidc` 填写表单。
应用程序 ID URI	将保留为空	对于本教程不被必需。
包含本机客户端	否	

> **WARNING**
> 如果设置了非 localhost 答复 URL，请注意回复 URL 列表中允许的内容的约束。

注册应用后，将显示租户中的应用列表。选择刚注册的应用。选择**副本**右侧的图标应**用程序 ID**字段以将其复制到剪贴板。

执行任何操作的详细信息在此期间，可以在 Azure AD B2C 租户中配置，但使浏览器窗口保持打开状态。创建 ASP.NET Core 应用后，没有更多的配置。

在 Visual Studio 2017 年 1 中创建 ASP.NET Core 应用

Visual Studio Web 应用程序模板可以配置为使用 Azure AD B2C 租户进行身份验证。

在 Visual Studio 中：

1. 创建新的 ASP.NET Core Web 应用程序。
2. 选择**Web 应用程序**从模板列表。
3. 选择**更改身份验证**按钮。

4. 在中**更改身份**验证对话框中，选择单**个用户帐户**，然后选择连**接到现有用户存储在云中**下拉列表中。

5. 完成窗体具有以下值：

设置	值
域名	*<在 B2C 租户的域名>*
应用程序 ID	*<粘贴剪贴板中的应用程序 ID>*

设置	值
回调路径	*<使用默认值>*
注册或登录策略	`B2C_1_SiUpIn`
重置密码策略	`B2C_1_SSPR`
编辑配置文件策略	*<将保留为空>*

选择**副本**链接旁边**回复 URI**将回复 URI 复制到剪贴板。选择**确定**以关闭**更改身份**验证对话框。选择**确定**创建 web 应用。

完成 B2C 应用程序注册

返回到 B2C 应用程序属性仍然打开的浏览器窗口。更改临时**回复 URL**从 Visual Studio 指定更早的值复制。选择**保存**在窗口的顶部。

> **TIP**
> 如果未复制回复 URL，在 web 项目属性中，使用调试选项卡中的 HTTPS 地址并追加**CallbackPath**值从*appsettings.json*。

配置策略

使用到的 Azure AD B2C 文档中的步骤创建注册或登录策略，然后创建密码重置策略。使用提供的文档中的示例值标识**提供者**，**注册属性**，并应**用程序声明**。使用**立即运行**是可选的按钮以测试这些策略，如文档中所述。

> **WARNING**
> 确保策略名称完全按照说明在文档中，并与这些策略中使用**更改身份**验证Visual Studio 中的对话框。可以在中验证的策略名称*appsettings.json*。

运行应用

在 Visual Studio 中，按**F5**生成并运行应用程序。Web 应用启动后，选择**Accept**以接受使用 cookie（如果提示），然后选择**登录**。

在浏览器将重定向到 Azure AD B2C 租户。（如果已经创建了一个测试策略）的现有帐户登录，或选择**立即注册**若要创建新帐户。**忘记了密码？** 链接用于重置忘记的密码。

已成功登录后，浏览器将重定向到 web 应用。

后续步骤

在本教程中，你将了解：

- 创建 Azure Active Directory B2C 租户
- 在 Azure AD B2C 中注册应用
- 使用 Visual Studio 创建 ASP.NET Core Web 应用程序配置为使用 Azure AD B2C 租户进行身份验证
- 配置控制的 Azure AD B2C 租户行为的策略

现在，ASP.NET Core 应用程序配置为使用 Azure AD B2C 进行身份验证，Authorize 属性可用来保护你的应用。在继续开发您的应用程序通过学习：

- 自定义 Azure AD B2C 用户界面。
- 配置密码复杂性要求。
- 启用多重身份验证。
- 配置其他标识提供者，例如Microsoft，Facebook，Google，Amazon，Twitter，等等。
- 使用 Azure AD Graph API从 Azure AD B2C 租户中检索其他用户信息，例如组成员身份。
- 保护 web API 使用 Azure AD B2C 的 ASP.NET Core。
- 从.NET web 应用中使用 Azure AD B2C 调用.NET web API。

中的 web Api 使用 Azure Active Directory B2C 在 ASP.NET Core 中的身份验证

2019/1/8 • • Edit Online

作者：Cam Soper

Azure Active Directory B2C (Azure AD B2C) 是云标识管理解决方案，适用于 web 和移动应用。该服务提供用于在云中和本地托管的应用的身份验证。身份验证类型包括个人帐户，社交网络帐户和联合企业帐户。Azure AD B2C 还提供了最小配置多重身份验证。

Azure Active Directory (Azure AD) 和 Azure AD B2C 是单独的产品产品/服务。Azure AD 租户表示组织，而 Azure AD B2C 租户表示与信赖方应用程序将使用的集合。若要了解详细信息，请参阅Azure AD B2C:常见问题 (FAQ)。

由于 web Api 有没有用户界面，因此它们无法将用户重定向到安全令牌服务等 Azure AD B2C。相反，API 是具有已验证了用户与 Azure AD B2C 将调用应用程序中传递的持有者令牌。API 会验证该令牌而无需直接用户交互。

在本教程中，学习如何：

- 创建 Azure Active Directory B2C 租户。
- 在 Azure AD B2C 中注册 Web API。
- 使用 Visual Studio 来创建 Web API 配置为使用 Azure AD B2C 租户进行身份验证。
- 配置控制的 Azure AD B2C 租户行为的策略。
- 使用 Postman 来模拟一个 web 应用，用于显示登录对话框中，检索令牌，并使用它来对 web API 发出请求。

系统必备

本演练需要以下项：

- Microsoft Azure 订阅
- Visual Studio 2017（任何版本）
- Postman

创建 Azure Active Directory B2C 租户

创建一个 Azure AD B2C 租户文档中所述。出现提示时，将租户与 Azure 订阅相关联是可选的用于本教程。

配置注册或登录策略

使用到的 Azure AD B2C 文档中的步骤创建注册或登录策略。将策略命名**SiUpIn**。使用提供的文档中的示例值标识**提供者**，**注册属性**，并应**用程序声明**。使用**立即运行**是可选的按钮以测试策略，如文档中所述。

在 Azure AD B2C 中注册 API

在新创建的 Azure AD B2C 租户中注册 API 通过文档中的步骤下**注册 web API**部分。

使用以下值：

设置	值	说明
名称	*{API 名称}*	输入**名称**描述你的应用向使用者的应用。

设置	值	说明
包括 web 应用 /web API	是	
允许隐式流	是	
回复 URL	`https://localhost`	回复 Url 属于终结点，Azure AD B2C 在其中返回应用请求的任何令牌。
应用程序 ID URI	*api*	该 URI 不需要解析为物理地址。它只需是唯一的。
包含本机客户端	否	

注册 API 后，将显示在租户中的应用和 Api 的列表。选择以前注册的 API。选择**副本**右侧的图标**应用程序 ID**字段以将其复制到剪贴板。选择**发布的作用域**，并验证默认*user_impersonation*作用域是存在。

在 Visual Studio 2017 年 1 中创建 ASP.NET Core 应用

Visual Studio Web 应用程序模板可以配置为使用 Azure AD B2C 租户进行身份验证。

在 Visual Studio 中：

1. 创建新的 ASP.NET Core Web 应用程序。
2. 选择**Web API**从模板列表。
3. 选择**更改身份**验证按钮。

4. 在中**更改身份**验证对话框中，选择单**个用户帐户** > **连接到现有用户存储在云中**。

5. 完成窗体具有以下值：

设置	值
域名	*{域在 B2C 租户名称}*
应用程序 ID	*{粘贴剪贴板中的应用程序 ID}*
注册或登录策略	B2C_1_SiUpIn

选择**确定**以关闭**更改身份**验证对话框。选择**确定**创建 web 应用。

Visual Studio 创建 web API 的具有控制器名为*ValuesController.cs*返回硬编码值的 GET 请求。类用修饰Authorize属性，因此所有请求都需要身份验证。

运行 web API

在 Visual Studio 中，运行 API。Visual Studio 启动浏览器指向 API 的根 URL。请注意地址栏中的 URL 并将在后台运行的 API。

> **NOTE**
> 由于没有为根 URL 定义没有控制器，浏览器可能显示 404（找不到页）错误。这是预期行为。

使用 Postman 来获取令牌，并测试 API

Postman是工具用于测试 web Api。对于本教程，Postman 模拟访问 web API 代表该用户的 web 应用程序。

注册为 web 应用的 Postman

Postman 模拟 web 应用从 Azure AD B2C 租户中获取令牌，因为它必须注册在租户中为 web 应用。注册使用 Postman文档中的步骤下**注册 web 应用**部分。在停止创**建 web 应用客户端机密**部分。本教程不需要客户端机密。

使用以下值：

设置	值	说明
名称	Postman	

设置	值	说明
包括 web 应用 /web API	是	
允许隐式流	是	
回复 URL	`https://getpostman.com/postman`	
应用程序 ID URI	*{留}*	对于本教程不被必需。
包含本机客户端	否	

新注册的 web 应用需要访问 web API 代表该用户的权限。

1. 选择**Postman**的应用程序，然后选择列表中**API 访问**在左侧菜单中。
2. 选择 **+ 添加**。
3. 在中选择 **API**下拉列表中，选择 web API 的名称。
4. 在中选择**作用域**下拉列表中，确保选中所有作用域。
5. 选择**确定**。

请注意 Postman 应用的应用程序 ID，因为它所需获取持有者令牌。

创建 Postman 请求

启动 Postman。默认情况下，显示 Postman创**建新**对话框后启动。如果未显示对话框中，选择 **+ 新建**在左上角的按钮。

从创**建新**对话框：

1. 选择**请求**。

2. 输入*获取值*中请**求名称**框。

3. 选择 **+** 创**建集合**若要创建用于存储请求新的集合。命名集合*ASP.NET Core 教程*，然后选择复选标记。

4. 选择**将保存到 ASP.NET Core 教程**按钮。

测试 web API 而无需身份验证

若要验证 web API 需要身份验证，首先请不带身份验证的请求。

1. 在中**输入请求 URL**框中，输入的 URL `ValuesController`。URL 是使用浏览器中显示的相同**api/values**追加。例如 `https://localhost:44375/api/values`。

2. 选择**发送**按钮。

3. 请注意响应的状态是*401 未授权*。

> **IMPORTANT**
>
> 如果收到"无法获取任何响应"的错误，可能需要禁用中的 SSL 证书验证Postman 设置。

获取持有者令牌

若要向 web API 发出的经过身份验证的请求，持有者令牌是必需的。Postman 可轻松地登录到 Azure AD B2C 租户并获取的令牌。

1. 上授权选项卡上，在类型下拉列表中，选择**OAuth 2.0**。在中授权将数据添加到下拉列表中，选择请求标头。选择获取新访问令牌。

2. 完成获取新访问令牌，如下所示的对话框：

设置	值	说明
令牌名称	*{令牌名称}*	输入令牌的描述性名称。
授权类型	隐式	
回调 URL	`https://getpostman.com/postman`	
身份验证 URL	`https://login.microsoftonline.com/{tenant domain name}/oauth2/v2.0/authorize?p=B2C_1_SiUpIn`	替换*{租户名称}*与租户的域名。重要:此 URL 必须具有相同域名中找到的内容作为 `AzureAdB2C.Instance` 中 web API 的*appsettings.json*文件。请参阅备注†。
客户端 ID	*{输入 Postman 应用应用程序 ID}*	
范围	`https://{tenant domain name}/{api}/user_impersonation openid offline_access`	替换*{租户名称}*与租户的域名。替换*{api}*替换应用程序 ID URI 在 web API 时提供首次注册 (在这种情况下，`api`)。Url 模式是：`https://{tenant}.onmicrosoft.com/{api-id-uri}/{scope name}`。
状态	*{留}*	
客户端身份验证	在正文中发送客户端凭据	

> **NOTE**
>
> † 在 Azure Active Directory B2C 门户中的策略设置对话框将显示两个可能的 Url:一个采用格式 `https://login.microsoftonline.com/` {租户名称} / {附加路径信息} 和其他格式 `https://{tenant name}.b2clogin.com/` {租户名称} / {附加路径信息}。它具有关键域中在中找到 `AzureAdB2C.Instance` 中 web API 的*appsettings.json*文件与在 web 应用中使用*appsettings.json*文件。这是用于表示在 Postman 中的身份验证 URL 字段的同一个域。请注意，Visual Studio 将使用比在门户中显示的内容稍有不同的 URL 格式。只要域匹配，适用于 URL。

3. 选择请**求令牌**按钮。

4. Postman 会打开一个包含 Azure AD B2C 租户的登录对话框的新窗口。（如果已经创建了一个测试策略）的现有帐户登录，或选择**立即注册**若要创建新帐户。**忘记了密码？** 链接用于重置忘记的密码。

5. 已成功登录后，在窗口关闭并管理访问令牌此时将显示对话框。向下的滚动到底部，选择**使用令牌**按钮。

测试 web API 身份验证

选择发**送**按钮再次发送该请求。这一次，响应状态是*200 确定*，并在响应上可见的 JSON 有效负载**正文**选项卡。

后续步骤

在本教程中，你将了解：

- 创建 Azure Active Directory B2C 租户。
- 在 Azure AD B2C 中注册 Web API。
- 使用 Visual Studio 来创建 Web API 配置为使用 Azure AD B2C 租户进行身份验证。
- 配置控制的 Azure AD B2C 租户行为的策略。
- 使用 Postman 来模拟一个 web 应用，用于显示登录对话框中，检索令牌，并使用它来对 web API 发出请求。

继续通过学习开发你的 API:

- 保护 web 应用使用 Azure AD B2C 的 ASP.NET Core。
- 从.NET web 应用中使用 Azure AD B2C 调用.NET web API。
- 自定义 Azure AD B2C 用户界面。
- 配置密码复杂性要求。
- 启用多重身份验证。
- 配置其他标识提供者，例如Microsoft，Facebook，Google，Amazon，Twitter，等等。
- 使用 Azure AD Graph API从 Azure AD B2C 租户中检索其他用户信息，例如组成员身份。

基于 ASP.NET Core 项目使用单个用户帐户创建项目

2018/9/21 • • Edit Online

ASP.NET Core 标识包含在 Visual Studio 中使用"单个用户帐户"选项的项目模板中。

中使用的.NET Core CLI 中可用的身份验证模板 `-au Individual`：

```
dotnet new mvc -au Individual
dotnet new webapi -au Individual
dotnet new webapp -au Individual
```

```
dotnet new mvc -au Individual
dotnet new webapi -au Individual
dotnet new razor -au Individual
```

无身份验证

在.NET Core CLI 与指定身份验证 `-au` 选项。在 Visual Studio 中，**更改身份验证**对话框是适用于新 web 应用程序。默认值为 Visual Studio 中的新 web 应用**无身份**验证。

使用无身份验证创建的项目：

- 不包含 web 页面和 UI 中登录和注销。
- 不包含身份验证代码。

Windows 身份验证

Windows 身份验证指定为新 web 应用程序中使用.NET Core CLI `-au Windows` 选项。在 Visual Studio 中，**更改身份验证**对话框提供了**Windows 身份验证**选项。

如果选择 Windows 身份验证，则该应用程序配置为使用Windows 身份验证 IIS 模块。Windows 身份验证适用于 Intranet web 站点。

其他资源

以下文章演示了如何使用 ASP.NET Core 使用单个用户帐户的模板中生成的代码：

- 使用 SMS 设置双因素身份验证
- ASP.NET Core 中的帐户确认和密码恢复
- 使用受授权的用户数据创建 ASP.NET Core 应用

在 ASP.NET Core 中授权简介

2018/7/6 • • Edit Online

授权是指确定进程用户是能够执行。例如，管理用户可以创建文档库、将文档添加、编辑文档，以及删除它们。使用库的非管理用户仅有权读取文档。

授权是正交和独立于身份验证。但是，授权要求的身份验证机制。身份验证是有助于确定用户是谁的过程。身份验证可能会创建一个或多个标识为当前用户。

授权类型

ASP.NET Core 授权提供一个简单、声明性角色以及丰富基于策略的模型。要求，以表示授权，并且处理程序评估针对需求的用户的声明。命令性检查可以基于简单的策略或策略的评估用户标识和该用户尝试访问资源的属性。

命名空间

授权组件，包括 `AuthorizeAttribute` 和 `AllowAnonymousAttribute` 中找不到属性，`Microsoft.AspNetCore.Authorization` 命名空间。

请查阅上的文档简单授权。

使用受授权的用户数据创建 ASP.NET Core 应用

2019/1/8 • • Edit Online

作者：Rick Anderson 和 Joe Audette

请参阅此 PDF ASP.NET Core MVC 版本。本教程的 ASP.NET Core 1.1 版本是这文件夹。ASP.NET Core 示例是在 1.1示例。

请参阅此 pdf

本教程演示如何使用受授权的用户数据创建 ASP.NET Core web 应用。它显示的身份验证（注册）的用户的联系人列表已创建。有三个安全组：

- **注册用户**可以查看所有已批准的数据还可以编辑/删除其自己的数据。
- **管理器**可以批准或拒绝的联系人数据。仅已批准的联系人是对用户可见。
- **管理**员可以批准/拒绝和编辑/删除的任何数据。

在下图中，用户 Rick（`rick@example.com`）登录。Rick 只能查看允许的联系人和编辑/删除/新建其联系人的链接。只有最后一个记录，创建由 Rick，显示编辑并删除链接。其他用户不会看到的最后一个记录，直到经理或管理员的状态更改为"已批准"。

在下图中，`manager@contoso.com`在和中的管理器角色进行签名：

下图显示在管理器的联系人的详细信息视图：

批准并拒绝按钮仅显示经理和管理员。

在下图中，`admin@contoso.com` 进行签名和管理员角色中：

管理员具有所有权限。她可以读取、编辑或删除任何联系，并更改联系人的状态。

通过创建该应用基架以下 `Contact` 模型：

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

此示例包含以下授权处理程序：

- `ContactIsOwnerAuthorizationHandler`：可确保用户只能编辑其数据。
- `ContactManagerAuthorizationHandler`：允许管理人员批准或拒绝的联系人。
- `ContactAdministratorsAuthorizationHandler`：允许管理员可以批准或拒绝联系人并将编辑/删除联系人。

系统必备

本教程被高级。您应熟悉：

- ASP.NET Core
- 身份验证
- 帐户确认和密码恢复

- 授权
- Entity Framework Core

在 ASP.NET Core 2.1 `User.IsInRole` 使用时，失败 `AddDefaultIdentity` 。本教程使用 `AddDefaultIdentity` ，因此需要 ASP.NET Core 2.2 或更高版本。请参阅此 GitHub 问题的解决办法。

Starter 和已完成应用程序

下载完成应用。测试已完成的应用，使你熟悉其安全功能。

入门级应用

下载初学者应用。

运行应用，点击**ContactManager**链接，并验证是否可以创建、编辑和删除联系人。

保护用户数据

以下部分介绍了所有主要的步骤以创建安全的用户数据应用程序。您可能会发现引用的已完成项目很有帮助。

将绑定到用户的联系人数据

使用 ASP.NET标识用户 ID，以确保用户可以编辑其数据，而不是其他用户数据。添加 `OwnerID` 并 `ContactStatus` 到 `Contact` 模型：

```
public class Contact
{
    public int ContactId { get; set; }

    // user ID from AspNetUser table.
    public string OwnerID { get; set; }

    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }

    public ContactStatus Status { get; set; }
}

public enum ContactStatus
{
    Submitted,
    Approved,
    Rejected
}
```

`OwnerID` 是从用户的 ID `AspNetUser` 表中标识数据库。 `Status` 字段确定是否可由普通用户查看联系人。

创建新的迁移并更新数据库：

```
dotnet ef migrations add userID_Status
dotnet ef database update
```

将角色服务添加到标识

追加AddRoles添加角色服务：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>().AddRoles<IdentityRole>()
         .AddEntityFrameworkStores<ApplicationDbContext>();
```

需要身份验证的用户

设置默认身份验证策略以要求用户进行身份验证：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>().AddRoles<IdentityRole>()
         .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddMvc(config =>
    {
        // using Microsoft.AspNetCore.Mvc.Authorization;
        // using Microsoft.AspNetCore.Authorization;
        var policy = new AuthorizationPolicyBuilder()
                         .RequireAuthenticatedUser()
                         .Build();
        config.Filters.Add(new AuthorizeFilter(policy));
    })
       .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

可以选择在 Razor 页面、控制器或操作方法级别使用的身份验证禁用 `[AllowAnonymous]` 属性。设置默认身份验证策略以要求用户进行身份验证来保护新添加的 Razor 页面和控制器。默认情况下所需的身份验证是比依赖于新的控制器和 Razor 页，以包含更安全 `[Authorize]` 属性。

添加AllowAnonymous到索引中，因此匿名用户可以获取有关站点的信息注册有关，和联系人页面。

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ContactManager.Pages
{
    [AllowAnonymous]
    public class IndexModel : PageModel
    {
        public void OnGet()
        {

        }
    }
}
```

配置测试帐户

`SeedData` 类创建两个帐户：管理员和管理员。使用机密管理器工具设置这些帐户的密码。从项目目录中设置密码(目录包含*Program.cs*):

```
dotnet user-secrets set SeedUserPW <PW>
```

如果未指定强密码，将引发异常时 `SeedData.Initialize` 调用。

更新 `Main` 使用测试密码：

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateWebHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;
            var context = services.GetRequiredService<ApplicationDbContext>();
            context.Database.Migrate();

            // requires using Microsoft.Extensions.Configuration;
            var config = host.Services.GetRequiredService<IConfiguration>();
            // Set password with the Secret Manager tool.
            // dotnet user-secrets set SeedUserPW <pw>

            var testUserPw = config["SeedUserPW"];
            try
            {
                SeedData.Initialize(services, testUserPw).Wait();
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex.Message, "An error occurred seeding the DB.");
            }
        }

        host.Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

创建测试帐户和更新联系人

更新 `Initialize` 中的方法 `SeedData` 类，以创建测试帐户：

```
public static async Task Initialize(IServiceProvider serviceProvider, string testUserPw)
{
    using (var context = new ApplicationDbContext(
        serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))
    {
        // For sample purposes seed both with the same password.
        // Password is set with the following:
        // dotnet user-secrets set SeedUserPW <pw>
        // The admin user can do anything

        var adminID = await EnsureUser(serviceProvider, testUserPw, "admin@contoso.com");
        await EnsureRole(serviceProvider, adminID, Constants.ContactAdministratorsRole);

        // allowed user can create and edit contacts that they create
        var managerID = await EnsureUser(serviceProvider, testUserPw, "manager@contoso.com");
        await EnsureRole(serviceProvider, managerID, Constants.ContactManagersRole);

        SeedDB(context, adminID);
    }
}

private static async Task<string> EnsureUser(IServiceProvider serviceProvider,
                                             string testUserPw, string UserName)
{
    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>();

    var user = await userManager.FindByNameAsync(UserName);
    if (user == null)
    {
        user = new IdentityUser { UserName = UserName };
        await userManager.CreateAsync(user, testUserPw);
    }

    return user.Id;
}

private static async Task<IdentityResult> EnsureRole(IServiceProvider serviceProvider,
                                                     string uid, string role)
{
    IdentityResult IR = null;
    var roleManager = serviceProvider.GetService<RoleManager<IdentityRole>>();

    if (roleManager == null)
    {
        throw new Exception("roleManager null");
    }

    if (!await roleManager.RoleExistsAsync(role))
    {
        IR = await roleManager.CreateAsync(new IdentityRole(role));
    }

    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>();

    var user = await userManager.FindByIdAsync(uid);

    IR = await userManager.AddToRoleAsync(user, role);

    return IR;
}
```

添加管理员用户 ID 和 `ContactStatus` 到联系人。先创建一个"已提交"和一个"已拒绝"的联系人。将用户 ID 和状态

添加到所有联系人。只能有一个联系人所示：

```
public static void SeedDB(ApplicationDbContext context, string adminID)
{
    if (context.Contact.Any())
    {
        return;   // DB has been seeded
    }

    context.Contact.AddRange(
        new Contact
        {
            Name = "Debra Garcia",
            Address = "1234 Main St",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "debra@example.com",
            Status = ContactStatus.Approved,
            OwnerID = adminID
        },
```

创建所有者、经理和管理员授权处理程序

创建 `ContactIsOwnerAuthorizationHandler` 类中*授权*文件夹。`ContactIsOwnerAuthorizationHandler` 验证对资源进行操作的用户拥有的资源。

```
using System.Threading.Tasks;
using ContactManager.Data;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactIsOwnerAuthorizationHandler
                : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        UserManager<IdentityUser> _userManager;

        public ContactIsOwnerAuthorizationHandler(UserManager<IdentityUser>
            userManager)
        {
            _userManager = userManager;
        }

        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                                   OperationAuthorizationRequirement requirement,
                                   Contact resource)
        {
            if (context.User == null || resource == null)
            {
                // Return Task.FromResult(0) if targeting a version of
                // .NET Framework older than 4.6:
                return Task.CompletedTask;
            }

            // If we're not asking for CRUD permission, return.

            if (requirement.Name != Constants.CreateOperationName &&
                requirement.Name != Constants.ReadOperationName   &&
                requirement.Name != Constants.UpdateOperationName &&
                requirement.Name != Constants.DeleteOperationName )
            {
                return Task.CompletedTask;
            }

            if (resource.OwnerID == _userManager.GetUserId(context.User))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

`ContactIsOwnerAuthorizationHandler` 调用上下文。成功当前经过身份验证的用户是否联系所有者。授权处理程序通常：

- 返回 `context.Succeed` 满足的要求。
- 返回 `Task.CompletedTask` 时不符合要求。`Task.CompletedTask` 既不成功或失败—它允许运行其他授权处理程序。

如果你需要将显式失败，返回上下文。失败。

应用程序允许联系所有者到编辑/删除/创建他们自己的数据。`ContactIsOwnerAuthorizationHandler` 不需要检查要求参数中传递该操作。

创建管理器授权处理程序

创建 `ContactManagerAuthorizationHandler` 类中*授权*文件夹。`ContactManagerAuthorizationHandler` 验证对资源进行操作的用户是管理员。只有经理们才可以批准或拒绝内容的更改（新的或已更改）。

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactManagerAuthorizationHandler :
        AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                                   OperationAuthorizationRequirement requirement,
                                   Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.CompletedTask;
            }

            // If not asking for approval/reject, return.
            if (requirement.Name != Constants.ApproveOperationName &&
                requirement.Name != Constants.RejectOperationName)
            {
                return Task.CompletedTask;
            }

            // Managers can approve or reject.
            if (context.User.IsInRole(Constants.ContactManagersRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

创建管理员授权处理程序

创建 `ContactAdministratorsAuthorizationHandler` 类中*授权*文件夹。`ContactAdministratorsAuthorizationHandler` 验证对资源进行操作的用户是管理员。管理员可以执行所有操作。

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public class ContactAdministratorsAuthorizationHandler
                    : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task HandleRequirementAsync(
                                            AuthorizationHandlerContext context,
                                    OperationAuthorizationRequirement requirement,
                                     Contact resource)
        {
            if (context.User == null)
            {
                return Task.CompletedTask;
            }

            // Administrators can do anything.
            if (context.User.IsInRole(Constants.ContactAdministratorsRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

注册授权处理程序

必须为注册服务使用 Entity Framework Core依赖关系注入使用AddScoped。`ContactIsOwnerAuthorizationHandler` 使用 ASP.NET Core标识，这基于实体框架核心。注册服务集合的处理程序，以便它们可供 `ContactsController` 通过依赖关系注入。将以下代码添加到末尾 `ConfigureServices`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>().AddRoles<IdentityRole>()
         .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddMvc(config =>
    {
        // using Microsoft.AspNetCore.Mvc.Authorization;
        // using Microsoft.AspNetCore.Authorization;
        var policy = new AuthorizationPolicyBuilder()
                         .RequireAuthenticatedUser()
                         .Build();
        config.Filters.Add(new AuthorizeFilter(policy));
    })
       .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // Authorization handlers.
    services.AddScoped<IAuthorizationHandler,
                          ContactIsOwnerAuthorizationHandler>();

    services.AddSingleton<IAuthorizationHandler,
                          ContactAdministratorsAuthorizationHandler>();

    services.AddSingleton<IAuthorizationHandler,
                          ContactManagerAuthorizationHandler>();
}
```

`ContactAdministratorsAuthorizationHandler` 和 `ContactManagerAuthorizationHandler` 添加为单一实例。它们是单一实例，因为它们不使用 EF 和所需的所有信息都位于 `Context` 参数的 `HandleRequirementAsync` 方法。

支持授权

在本部分中，将更新 Razor 页面和添加操作要求类。

查看联系人操作要求类

查看 `ContactOperations` 类。此类包含要求应用支持：

```
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public static class ContactOperations
    {
        public static OperationAuthorizationRequirement Create =
          new OperationAuthorizationRequirement {Name=Constants.CreateOperationName};
        public static OperationAuthorizationRequirement Read =
          new OperationAuthorizationRequirement {Name=Constants.ReadOperationName};
        public static OperationAuthorizationRequirement Update =
          new OperationAuthorizationRequirement {Name=Constants.UpdateOperationName};
        public static OperationAuthorizationRequirement Delete =
          new OperationAuthorizationRequirement {Name=Constants.DeleteOperationName};
        public static OperationAuthorizationRequirement Approve =
          new OperationAuthorizationRequirement {Name=Constants.ApproveOperationName};
        public static OperationAuthorizationRequirement Reject =
          new OperationAuthorizationRequirement {Name=Constants.RejectOperationName};
    }

    public class Constants
    {
        public static readonly string CreateOperationName = "Create";
        public static readonly string ReadOperationName = "Read";
        public static readonly string UpdateOperationName = "Update";
        public static readonly string DeleteOperationName = "Delete";
        public static readonly string ApproveOperationName = "Approve";
        public static readonly string RejectOperationName = "Reject";

        public static readonly string ContactAdministratorsRole =
                                                              "ContactAdministrators";
        public static readonly string ContactManagersRole = "ContactManagers";
    }
}
```

创建联系人 Razor 页面的基类

创建一个包含在联系人 Razor 页面使用的服务的基类。基类将初始化代码放在一个位置：

```
using ContactManager.Data;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ContactManager.Pages.Contacts
{
    public class DI_BasePageModel : PageModel
    {
        protected ApplicationDbContext Context { get; }
        protected IAuthorizationService AuthorizationService { get; }
        protected UserManager<IdentityUser> UserManager { get; }

        public DI_BasePageModel(
            ApplicationDbContext context,
            IAuthorizationService authorizationService,
            UserManager<IdentityUser> userManager) : base()
        {
            Context = context;
            UserManager = userManager;
            AuthorizationService = authorizationService;
        }
    }
}
```

前面的代码：

- 添加 `IAuthorizationService` 服务对授权处理程序的访问权限。
- 将标识添加 `UserManager` 服务。
- 添加 `ApplicationDbContext` 。

更新 CreateModel

更新创建页面模型构造函数以使用 `DI_BasePageModel` 基类：

```
public class CreateModel : DI_BasePageModel
{
    public CreateModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }
```

更新 `CreateModel.OnPostAsync` 方法：

- 添加到的用户 ID `Contact` 模型。
- 调用授权处理程序，以验证用户有权创建联系人。

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    Contact.OwnerID = UserManager.GetUserId(User);

    // requires using ContactManager.Authorization;
    var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, Contact,
                                                ContactOperations.Create);
    if (!isAuthorized.Succeeded)
    {
        return new ChallengeResult();
    }

    Context.Contact.Add(Contact);
    await Context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

更新 IndexModel

更新 `OnGetAsync` 方法，以便仅被批准的联系人显示为普通用户：

```
public class IndexModel : DI_BasePageModel
{
    public IndexModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public IList<Contact> Contact { get; set; }

    public async Task OnGetAsync()
    {
        var contacts = from c in Context.Contact
                       select c;

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
                           User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        // Only approved contacts are shown UNLESS you're authorized to see them
        // or you are the owner.
        if (!isAuthorized)
        {
            contacts = contacts.Where(c => c.Status == ContactStatus.Approved
                                        || c.OwnerID == currentUserId);
        }

        Contact = await contacts.ToListAsync();
    }
}
```

更新 EditModel

添加授权处理程序以验证的用户拥有联系人。正在验证资源授权，因为 `[Authorize]` 属性不能满足。评估属性时，应用程序不具有对资源的访问。基于资源的授权必须是命令性。应用程序页面模型中加载或加载处理程序本身内获得资源的访问权限，则必须执行检查。您经常访问的资源，通过传入的资源键。

```
public class EditModel : DI_BasePageModel
{
    public EditModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(
                                            m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, Contact,
                                                ContactOperations.Update);
```

```
                                                    ContactOperations.Update);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        // Fetch Contact from DB to get OwnerID.
        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, contact,
                                                ContactOperations.Update);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }

        Contact.OwnerID = contact.OwnerID;

        Context.Attach(Contact).State = EntityState.Modified;

        if (contact.Status == ContactStatus.Approved)
        {
            // If the contact is updated after approval,
            // and the user cannot approve,
            // set the status back to submitted so the update can be
            // checked and approved.
            var canApprove = await AuthorizationService.AuthorizeAsync(User,
                                    contact,
                                    ContactOperations.Approve);

            if (!canApprove.Succeeded)
            {
                contact.Status = ContactStatus.Submitted;
            }
        }

        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }

    private bool ContactExists(int id)
    {
        return Context.Contact.Any(e => e.ContactId == id);
    }
}
```

更新 DeleteModel

更新要使用授权处理程序来验证用户具有 delete 权限 contact 上的删除页面模型。

```
public class DeleteModel : DI_BasePageModel
{
    public DeleteModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(
                                        m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                            User, Contact,
                                            ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        Contact = await Context.Contact.FindAsync(id);

        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                            User, contact,
                                            ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }

        Context.Contact.Remove(Contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

将授权服务注入到视图

目前，此 UI 显示编辑和删除的用户无法修改的联系人的链接。

注入中的授权服务*views/_viewimports.cshtml*文件，以便它可供所有视图：

```
@using Microsoft.AspNetCore.Identity
@using ContactManager
@using ContactManager.Data
@namespace ContactManager.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using ContactManager.Authorization;
@using Microsoft.AspNetCore.Authorization
@using ContactManager.Models
@inject IAuthorizationService AuthorizationService
```

上述标记添加了多种 `using` 语句。

更新编辑并删除中的链接*Pages/Contacts/Index.cshtml*以便仅在呈现具有适当权限的用户：

```
@page
@model ContactManager.Pages.Contacts.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Address)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].City)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].State)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Zip)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Email)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Status)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Contact)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Address)
                </td>
                <td>
```

```
                    @Html.DisplayFor(modelItem => item.City)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.State)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Zip)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Email)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Status)
                </td>
                <td>
                    @if ((await AuthorizationService.AuthorizeAsync(
                     User, item,
                     ContactOperations.Update)).Succeeded)
                    {
                        <a asp-page="./Edit" asp-route-id="@item.ContactId">Edit</a>
                        <text> | </text>
                    }

                    <a asp-page="./Details" asp-route-id="@item.ContactId">Details</a>

                    @if ((await AuthorizationService.AuthorizeAsync(
                     User, item,
                     ContactOperations.Delete)).Succeeded)
                    {
                        <text> | </text>
                        <a asp-page="./Delete" asp-route-id="@item.ContactId">Delete</a>
                    }
                </td>
            </tr>
        }
    </tbody>
</table>
```

WARNING

隐藏用户没有权限更改的数据中的链接不会保护应用。隐藏链接使应用更加友好的用户显示唯一有效的链接。用户可以 hack 生成的 Url 以调用编辑和删除操作上不拥有的数据。Razor 页面或控制器必须强制执行访问检查，以保护数据。

更新详细信息

更新的详细信息视图，使管理员可以批准或拒绝联系人：

```
        @*Precedng markup omitted for brevity.*@
        <dt>
            @Html.DisplayNameFor(model => model.Contact.Email)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Contact.Email)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Contact.Status)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Contact.Status)
        </dd>
    </dl>
</div>

@if (Model.Contact.Status != ContactStatus.Approved)
{
    @if ((await AuthorizationService.AuthorizeAsync(
     User, Model.Contact, ContactOperations.Approve)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Approved" />
            <button type="submit" class="btn btn-xs btn-success">Approve</button>
        </form>
    }
}

@if (Model.Contact.Status != ContactStatus.Rejected)
{
    @if ((await AuthorizationService.AuthorizeAsync(
     User, Model.Contact, ContactOperations.Reject)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Rejected" />
            <button type="submit" class="btn btn-xs btn-success">Reject</button>
        </form>
    }
}

<div>
    @if ((await AuthorizationService.AuthorizeAsync(
         User, Model.Contact,
         ContactOperations.Update)).Succeeded)
    {
        <a asp-page="./Edit" asp-route-id="@Model.Contact.ContactId">Edit</a>
        <text> | </text>
    }
    <a asp-page="./Index">Back to List</a>
</div>
```

更新详细信息页模型:

```
public class DetailsModel : DI_BasePageModel
{
    public DetailsModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
                           User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        if (!isAuthorized
            &&  currentUserId != Contact.OwnerID
            && Contact.Status != ContactStatus.Approved)
        {
            return new ChallengeResult();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id, ContactStatus status)
    {
        var contact = await Context.Contact.FirstOrDefaultAsync(
                                                m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var contactOperation = (status == ContactStatus.Approved)
                                                 ? ContactOperations.Approve
                                                 : ContactOperations.Reject;

        var isAuthorized = await AuthorizationService.AuthorizeAsync(User, contact,
                                   contactOperation);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }
        contact.Status = status;
        Context.Contact.Update(contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

添加或删除角色对用户的

请参阅本期有关的信息：

- 从用户删除的特权。例如静音聊天应用程序中的用户。
- 将权限添加到用户。

测试已完成的应用程序

如果你尚未设置设定为种子的用户帐户的密码，使用机密管理器工具设置密码：

- 选择一个强密码：使用八个或更多的字符和至少一个大写字符，其编号，和符号。例如，`Passw0rd!`符合强密码要求。
- 执行以下命令从项目的文件夹，其中`<PW>`的密码：

```
dotnet user-secrets set SeedUserPW <PW>
```

如果应用了联系人：

- 删除所有记录中`Contact`表。
- 重新启动应用以设定数据库种子。

测试已完成的应用程序的简单方法是启动三个不同的浏览器（或 incognito/InPrivate 会话）。在一个浏览器中注册一个新用户 (例如，`test@example.com`)。登录到每个浏览器使用不同的用户。验证以下操作：

- 已注册的用户可以查看所有已批准的联系人数据。
- 已注册的用户可以编辑/删除其自己的数据。
- 经理可以批准/拒绝的联系人数据。`Details`视图视图将显示**批准并拒**绝按钮。
- 管理员可以批准/拒绝和编辑/删除所有数据。

“用户”	由应用程序进行种子设定	选项
test@example.com	否	编辑/删除自己的数据。
manager@contoso.com	是	批准/拒绝和编辑/删除拥有的数据。
admin@contoso.com	是	批准/拒绝和编辑/删除所有数据。

在管理员的浏览器中创建联系人。删除 URL 复制并编辑从管理员的联系信息。将下面的链接粘贴到测试用户的浏览器以验证测试用户不能执行这些操作。

创建入门级应用

- 创建名为"ContactManager"Razor 页面应用
  - 创建包含应用单个用户帐户。
  - 它命名为"ContactManager"使命名空间匹配的示例中使用的命名空间。
  - `-uld` 指定 LocalDB，而不是 SQLite

```
dotnet new webapp -o ContactManager -au Individual -uld
```

- 添加*Models/Contact.cs*:

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

- 基架 `Contact` 模型。
- 创建初始迁移并更新数据库：

```
dotnet aspnet-codegenerator razorpage -m Contact -udl -dc ApplicationDbContext -outDir Pages\Contacts -
-referenceScriptLibraries
dotnet ef database drop -f
dotnet ef migrations add initial
dotnet ef database update
```

- 更新**ContactManager**中的定位点*pages/_layout.cshtml*文件：

```
<a asp-page="/Contacts/Index" class="navbar-brand">ContactManager</a>
```

- 测试应用程序的创建、编辑和删除联系人

设定数据库种子

添加SeedData类来*数据*文件夹。

调用 `SeedData.Initialize` 从 `Main`：

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateWebHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;

            try
            {
                var context = services.GetRequiredService<ApplicationDbContext>();
                context.Database.Migrate();
                SeedData.Initialize(services, "not used");
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred seeding the DB.");
            }
        }

        host.Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

测试应用程序设定数据库种子。如果联系人 DB 中有任何行，则不会运行 seed 方法。

其他资源

- 生成 Azure 应用服务中的.NET Core 和 SQL 数据库 web 应用
- ASP.NET Core 授权实验室。此实验室进入的安全功能，本教程中介绍的更多详细信息。
- 在 ASP.NET Core 中授权简介
- 基于自定义策略的授权

在 ASP.NET Core razor 页授权约定

2018/10/30 • • Edit Online

作者：Luke Latham

Razor 页面应用中控制访问权限的一种方法是在启动时使用授权约定。这些约定，可为用户授权，并允许匿名用户访问各个页面的文件夹。自动在本主题中所述的约定适用授权筛选器来控制访问。

查看或下载示例代码（如何下载）

此示例应用程序使用Cookie 身份验证，而无需 ASP.NET Core 标识。假设用户、Maria Rodriguez 的用户帐户是硬编码到应用程序。使用电子邮件用户名"maria.rodriguez@contoso.com"和任何密码以登录用户。用户进行身份验证中 `AuthenticateUser` 中的方法*Pages/Account/Login.cshtml.cs*文件。在实际示例中，用户会根据数据库身份验证。若要使用 ASP.NET Core 标识，请按照中的指导ASP.NET Core 上的标识简介主题。概念和本主题中所示的示例同样适用于使用 ASP.NET Core 标识的应用。

需要授权可访问的页面

使用AuthorizePage通过约定AddRazorPagesOptions若要添加AuthorizeFilter到指定路径的页面：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

指定的路径是视图引擎路径，即无需扩展和包含仅正斜杠的 Razor 页面根相对路径。

AuthorizePage 重载时才需要指定的授权策略可用。

> **NOTE**
>
> `AuthorizeFilter` 可以应用于具有的页面模型类 `[Authorize]` 筛选器特性。有关详细信息，请参阅Authorize 筛选器属性。

需要授权才能访问页面的文件夹

使用AuthorizeFolder通过约定AddRazorPagesOptions若要添加AuthorizeFilter到所有指定路径处的文件夹中的页：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

指定的路径是视图引擎路径，这是 Razor 页面根相对路径。

AuthorizeFolder 重载时才需要指定的授权策略可用。

需要授权才能访问区域页

使用AuthorizeAreaPage通过约定AddRazorPagesOptions若要添加AuthorizeFilter到指定路径的区域页面：

```
options.Conventions.AuthorizeAreaPage("Identity", "/Manage/Accounts");
```

页面名称是文件的指定的区域不相对于页面根目录具有扩展名的路径。例如，该文件的页名称*Areas/Identity/Pages/Manage/Accounts.cshtml*是 */管理/帐户*。

AuthorizeAreaPage 重载时才需要指定的授权策略可用。

需要授权才能访问区域的文件夹

使用AuthorizeAreaFolder通过约定AddRazorPagesOptions若要添加AuthorizeFilter到所有指定路径处的文件夹中的区域：

```
options.Conventions.AuthorizeAreaFolder("Identity", "/Manage");
```

文件夹路径是相对于指定区域的页面根目录文件夹的路径。例如下的文件的文件夹路径 *领域/Identity/页/管理/* 是 */管理*。

AuthorizeAreaFolder 重载时才需要指定的授权策略可用。

允许匿名访问的页

使用AllowAnonymousToPage通过约定AddRazorPagesOptions若要添加AllowAnonymousFilter到指定路径处的页面：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

指定的路径是视图引擎路径，即无需扩展和包含仅正斜杠的 Razor 页面根相对路径。

允许匿名访问的页面的文件夹

使用AllowAnonymousToFolder通过约定AddRazorPagesOptions若要添加AllowAnonymousFilter到所有指定路径处的文件夹中的页：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

指定的路径是视图引擎路径，这是 Razor 页面根相对路径。

请注意在组合授权和匿名访问

它是完全有效，若要指定页面的文件夹需要授权，并指定该文件夹中的页面，允许匿名访问：

```
// This works.
.AuthorizeFolder("/Private").AllowAnonymousToPage("/Private/Public")
```

但是，与之相反，不是这样。不能声明为匿名访问的页面的文件夹，并指定中进行授权的页面：

```
// This doesn't work!
.AllowAnonymousToFolder("/Public").AuthorizePage("/Public/Private")
```

需要专用的页上的授权不起作用，因为时同时 `AllowAnonymousFilter` 并 `AuthorizeFilter` 页上，应用了筛选器 `AllowAnonymousFilter` wins 和控制的访问。

其他资源

- Razor 页面自定义路由和页面模型提供程序
- PageConventionCollection类

在 ASP.NET Core 中的简单授权

2018/6/27 • • Edit Online

通过控制在 MVC 中的授权 `AuthorizeAttribute` 属性和其各种参数。简单地说，应用 `AuthorizeAttribute` 属性设为到控制器的控制器或操作限制访问或对任何经过身份验证的用户的操作。

例如，下面的代码可访问的限制 `AccountController` 到任何经过身份验证的用户。

```
[Authorize]
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

如果你想要应用于操作，而不是控制器的授权，应用 `AuthorizeAttribute` 属性设为操作本身：

```
public class AccountController : Controller
{
   public ActionResult Login()
   {
   }

   [Authorize]
   public ActionResult Logout()
   {
   }
}
```

现在，只有经过身份验证的用户可以访问 `Logout` 函数。

你还可以使用 `AllowAnonymous` 特性以允许使用的未经身份验证用户添加到各个操作进行访问。例如：

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

这将允许仅经过身份验证的用户到 `AccountController`，除 `Login` 操作，这是可访问的所有用户，而不考虑其经过身份验证或未经身份验证 / 匿名状态。

WARNING

`[AllowAnonymous]` 绕过所有授权语句。如果合并 `[AllowAnonymous]` 和任何 `[Authorize]` 属性，`[Authorize]` 属性将被忽略。例如，如果将应用 `[AllowAnonymous]` 控制器级别中，任何 `[Authorize]` 属性在同一个控制器上（或其中的任何操作）将被忽略。

ASP.NET Core 中基于角色的授权

2018/7/31 • • Edit Online

创建一个标识时它可能属于一个或多个角色。例如，爱妻 Tracy 可能属于管理员和用户角色，同时 Scott 可能仅属于用户角色。如何创建和管理这些角色取决于后备存储的授权过程。角色公开为通过开发人员IsInRole方法ClaimsPrincipal类。

> **IMPORTANT**
>
> 本主题不适用于 Razor 页面。Razor 页面支持IPageFilter并iasyncpagefilter 表示。有关详细信息，请参阅 Razor 页面的筛选方法。

添加角色检查

基于角色的授权检查是声明性—开发人员将其嵌入在其代码中，对控制器或在控制器内的动作指定当前用户必须是成员的访问请求的资源的角色。

例如，下面的代码上限制任何操作的访问权限 `AdministrationController` 谁是其成员的用户到 `Administrator` 角色：

```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{
}
```

以逗号分隔的列表，可以指定多个角色：

```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
}
```

将仅可由成员的用户访问此控制器的 `HRManager` 角色或 `Finance` 角色。

如果在应用多个属性，则访问用户必须是指定; 的所有角色的成员下面的示例要求用户必须是两个成员 `PowerUser` 和 `ControlPanelUser` 角色。

```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
}
```

通过应用其他角色授权属性在操作级别，可以进一步限制访问权限：

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

中的上一代码片段成员 `Administrator` 角色或 `PowerUser` 角色可访问该控制器并 `SetTime` 操作，但只有的成员 `Administrator` 角色可以访问 `ShutDown` 操作。

您还可以锁定一个控制器，但允许匿名、未经身份验证访问各项操作。

```
[Authorize]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [AllowAnonymous]
    public ActionResult Login()
    {
    }
}
```

基于策略角色检查

此外可以使用新的策略语法，其中一名开发人员将策略在启动时注册为授权服务配置的一部分表示角色的要求。这通常发生在 `ConfigureServices()` 在您*Startup.cs*文件。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdministratorRole", policy => policy.RequireRole("Administrator"));
    });
}
```

使用应用策略 `Policy` 属性上的 `AuthorizeAttribute` 属性：

```
[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown()
{
    return View();
}
```

如果你想要指定多个允许的角色中一项要求，则您可以将他们指定为参数 `RequireRole` 方法：

```
options.AddPolicy("ElevatedRights", policy =>
                  policy.RequireRole("Administrator", "PowerUser", "BackupAdministrator"));
```

此示例中授权用户属于 `Administrator`， `PowerUser` 或 `BackupAdministrator` 角色。

ASP.NET Core 中基于声明的授权

2018/7/6 • • Edit Online

创建一个标识时它可能会分配一个或多个由受信任方发出的声明。声明是表示哪些使用者名称值对，可以不哪些的主题。例如，你可能具有驱动程序的许可证，本地驱动的许可证颁发机构签发。驱动程序的许可证对其具有你的出生日期。在这种情况下将声明名称 `DateOfBirth`，声明值将是你为出生日期，例如 `8th June 1970` 和颁发者是驱动的许可证颁发机构。基于声明的授权，简单地说，将检查声明的值，并允许对基于该值资源的访问。例如，如果你想夜间俱乐部访问授权过程可能是：

门安全负责人将评估你的出生声明，并且它们是否在授予你访问之前信任颁发者（驱动许可证机构）的日期的值。

标识可以包含具有多个值的多个声明，并且可以包含多个相同类型的声明。

添加声明检查

声明基于的授权检查声明性-开发人员将它们嵌入在其代码中，针对控制器或者控制器中的某个操作指定当前用户必须拥有，并选择性地声明的值必须保留访问的声明请求的资源。要求是基于策略的声明，开发人员必须生成并注册表达声明要求的策略。

最简单类型的声明策略如下所示的声明存在，并且不会检查值。

首先，你需要生成和注册策略。这是作为授权服务配置的一部分，这通常需要一部分花在 `ConfigureServices()` 中你*Startup.cs*文件。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("EmployeeNumber"));
    });
}
```

在这种情况下 `EmployeeOnly` 策略将检查是否存在 `EmployeeNumber` 于当前的标识声明。

你随后可应用策略使用 `Policy` 属性 `AuthorizeAttribute` 特性以指定策略名称;

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    return View();
}
```

`AuthorizeAttribute` 特性可以应用于整个控制器，在这种情况，仅匹配策略的标识将在控制器上允许向任何操作的访问。

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public ActionResult VacationBalance()
    {
    }
}
```

如果您有一个控制器受 `AuthorizeAttribute` 属性，但想要允许匿名访问你应用的特定操作 `AllowAnonymousAttribute` 属性。

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public ActionResult VacationBalance()
    {
    }

    [AllowAnonymous]
    public ActionResult VacationPolicy()
    {
    }
}
```

大多数声明附带了一个值。创建策略时，你可以指定允许的值的列表。下面的示例将仅成功执行员工的员工数是 1、2、3、4 或 5。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Founders", policy =>
                          policy.RequireClaim("EmployeeNumber", "1", "2", "3", "4", "5"));
    });
}
```

添加泛型声明检查

如果声明值不是单个值或转换是必需的使用RequireAssertion。有关详细信息，请参阅func 用于满足策略。

多个策略评估

如果将多个策略应用到的控制器或操作，然后授予访问权限之前也必须传递所有策略。例如：

```
[Authorize(Policy = "EmployeeOnly")]
public class SalaryController : Controller
{
    public ActionResult Payslip()
    {
    }

    [Authorize(Policy = "HumanResources")]
    public ActionResult UpdateSalary()
    {
    }
}
```

在上例中任何标识了满足 `EmployeeOnly` 策略可以访问 `Payslip` 在控制器上强制执行该策略的操作。但是为了调用

`UpdateSalary` 标识必须满足的操作*同时* `EmployeeOnly` 策略和 `HumanResources` 策略。

如果你希望更复杂的策略，如将出生声明的日期，计算年龄字段从它，然后检查年龄为 21 或更低版本，则需要编写自定义策略处理程序。

ASP.NET Core中基于策略的授权

2019/1/11 • • Edit Online

实际上，基于角色的授权和基于声明的授权都使用要求、要求处理程序和预配置的策略。这些构建基块支持在代码中使用授权评估表达式。结果就是，授权结构更加丰富，可重复使用，并且可以测试。

授权策略包含一个或多个要求。授权策略包含一个或多个要求，并在 `Startup.ConfigureServices` 方法中作为授权服务配置的一部分注册：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });
}
```

在前面的示例中，创建了一个“AtLeast21”策略。它只有一个要求—，即最低年龄，以参数的形式传递给要求。

将 `[Authorize]` 属性和策略名称配合使用即可应用策略。例如：

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

[Authorize(Policy = "AtLeast21")]
public class AlcoholPurchaseController : Controller
{
    public IActionResult Login() => View();

    public IActionResult Logout() => View();
}
```

要求

授权要求是一个可供策略用来评估当前用户主体的数据参数的集合。在我们的“AtLeast21”策略中，此要求是单个参数—最低年龄。要求可以实现 IAuthorizationRequirement，这是一个空标记接口。参数化的最低年龄要求可以按如下方式实现：

```
using Microsoft.AspNetCore.Authorization;

public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public int MinimumAge { get; private set; }

    public MinimumAgeRequirement(int minimumAge)
    {
        MinimumAge = minimumAge;
    }
}
```

> **NOTE**
> 一项要求并不需要具有数据或属性。

授权处理程序

授权处理程序负责评估要求的属性。授权处理程序会针对提供的AuthorizationHandlerContext 来评估要求，确定是否允许访问。

一项要求可以有多个处理程序。处理程序可以继承 AuthorizationHandler<，其中的 `TRequirement` 是需处理的要求。另外，一个处理程序也可以通过实现 IAuthorizationHandler 来处理多个类型的要求。

一个要求使用一个处理程序

下面是一对一关系的示例，其中的单个最低年龄要求处理程序使用单个要求：

```
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;
using System;
using System.Security.Claims;
using System.Threading.Tasks;

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   MinimumAgeRequirement requirement)
    {
        if (!context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth &&
                                        c.Issuer == "http://contoso.com"))
        {
            //TODO: Use the following if targeting a version of
            //.NET Framework older than 4.6:
            //      return Task.FromResult(0);
            return Task.CompletedTask;
        }

        var dateOfBirth = Convert.ToDateTime(
            context.User.FindFirst(c => c.Type == ClaimTypes.DateOfBirth &&
                                        c.Issuer == "http://contoso.com").Value);

        int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;
        if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))
        {
            calculatedAge--;
        }

        if (calculatedAge >= requirement.MinimumAge)
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

前面的代码确定当前的用户主体是否有一个由已知的受信任颁发者颁发的出生日期声明。缺少声明时，无法进行授权，这种情况下会返回已完成的任务。存在声明时，会计算用户的年龄。如果用户满足此要求所定义的最低年龄，则可以认为授权成功。授权成功后，会调用 `context.Succeed`，使用满足的要求作为其唯一参数。

有关多个要求使用一个处理程序

以下是一个对多关系的权限的处理程序可以在其中处理三种不同类型的要求的示例：

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;

public class PermissionHandler : IAuthorizationHandler
{
    public Task HandleAsync(AuthorizationHandlerContext context)
    {
        var pendingRequirements = context.PendingRequirements.ToList();

        foreach (var requirement in pendingRequirements)
        {
            if (requirement is ReadPermission)
            {
                if (IsOwner(context.User, context.Resource) ||
                    IsSponsor(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
            else if (requirement is EditPermission ||
                     requirement is DeletePermission)
            {
                if (IsOwner(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }

    private bool IsOwner(ClaimsPrincipal user, object resource)
    {
        // Code omitted for brevity

        return true;
    }

    private bool IsSponsor(ClaimsPrincipal user, object resource)
    {
        // Code omitted for brevity

        return true;
    }
}
```

前面的代码会遍历PendingRequirements—不包含要求的属性标记为成功。有关 `ReadPermission` 要求，用户必须是所有者或主办方来访问所请求的资源。情况下 `EditPermission` 或 `DeletePermission` 要求、他或她必须是所有者才能访问请求的资源。

处理程序注册

处理程序是在配置期间在服务集合中注册的。例如：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });

    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
}
```

每个处理程序均通过调用 `services.AddSingleton<IAuthorizationHandler, YourHandlerClass>();` 添加到服务集合。

处理程序应返回什么？

请注意，`Handle` 处理程序示例中的 方法不返回值。如何表明状态是成功还是失败？

- 处理程序通过调用 `context.Succeed(IAuthorizationRequirement requirement)` 并传递已成功验证的要求来表示成功。
- 处理程序通常不需要处理失败，因为同一要求的其他处理程序可能会成功。
- 若要保证失败，即使其他要求处理程序会成功，请调用 `context.Fail` 。

设置为 `false` 时，InvokeHandlersAfterFailure 属性（在 ASP.NET Core 1.1 及更高版本中提供）会在已调用 `context.Fail` 的情况下不执行处理程序。`InvokeHandlersAfterFailure` 默认为 `true`，这种情况下会调用所有处理程序。这样要求以产生副作用，例如日志记录，这些始终发生即使 `context.Fail` 已在另一个处理程序调用。

为什么需要对一项要求使用多个处理程序？

在需要以 **OR** 逻辑为基础进行评估的情况下，可以针对单个要求实现多个处理程序。例如，Microsoft 的门只能使用门禁卡打开。如果你将门禁卡丢在家中，可以要求前台打印一张临时标签来开门。在这种情况下，只有一个要求，即 *BuildingEntry*，但有多个处理程序，每个处理程序针对单个要求进行检查。

BuildingEntryRequirement.cs

```
using Microsoft.AspNetCore.Authorization;

public class BuildingEntryRequirement : IAuthorizationRequirement
{
}
```

BadgeEntryHandler.cs

```
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;
using System.Security.Claims;
using System.Threading.Tasks;

public class BadgeEntryHandler : AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   BuildingEntryRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == ClaimTypes.BadgeId &&
                                       c.Issuer == "http://microsoftsecurity"))
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

TemporaryStickerHandler.cs

```
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;
using System.Security.Claims;
using System.Threading.Tasks;

public class TemporaryStickerHandler : AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   BuildingEntryRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == ClaimTypes.TemporaryBadgeId &&
                                       c.Issuer == "https://microsoftsecurity"))
        {
            // We'd also check the expiration date on the sticker.
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

请确保这两个处理程序已注册。当策略评估 `BuildingEntryRequirement` 时，如果有一个处理程序成功，则策略评估成功。

使用 func 满足策略

有些情况下，策略很容易用代码实现。可以在通过 `Func<AuthorizationHandlerContext, bool>` 策略生成器配置策略时提供 `RequireAssertion`。

例如，上一个 `BadgeEntryHandler` 可以重写，如下所示：

```
services.AddAuthorization(options =>
{
    options.AddPolicy("BadgeEntry", policy =>
        policy.RequireAssertion(context =>
            context.User.HasClaim(c =>
                (c.Type == ClaimTypes.BadgeId ||
                 c.Type == ClaimTypes.TemporaryBadgeId) &&
                 c.Issuer == "https://microsoftsecurity")));
});
```

访问处理程序中的 MVC 请求上下文

在授权处理程序中实现的 `HandleRequirementAsync` 方法有两个参数：`AuthorizationHandlerContext` 以及你正在处理的 `TRequirement`。MVC 或 Jabbr 之类的框架可以自由地将任何对象添加到 `Resource` 中的 `AuthorizationHandlerContext` 属性，以便传递额外信息。

例如，MVC 在 属性中传递 AuthorizationFilterContext `Resource` 实例。可以通过此属性访问 `HttpContext`、`RouteData` 以及 MVC 和 Razor 页面提供的所有其他内容。

对 `Resource` 属性的使用取决于框架。使用 `Resource` 属性中的信息时，授权策略就会局限于特定的框架。应使用 `Resource` 关键字来强制转换 `as` 属性，然后确认该强制转换是否成功，确保代码在其他框架上运行时不会崩溃并抛出 `InvalidCastException` 异常：

```
// Requires the following import:
//     using Microsoft.AspNetCore.Mvc.Filters;
if (context.Resource is AuthorizationFilterContext mvcContext)
{
    // Examine MVC-specific things like routing data.
}
```

在 ASP.NET Core 中使用 IAuthorizationPolicyProvider 的自定义授权策略提供程序

2019/1/12 • • Edit Online

通过Mike Rousos

通常使用时基于策略的授权，通过调用注册策略 `AuthorizationOptions.AddPolicy` 作为授权服务配置的一部分。在某些情况下，它可能不是可能（或最好）若要以这种方式注册所有授权策略。在这些情况下，你可以使用自定义 `IAuthorizationPolicyProvider` 来控制如何提供授权策略。

方案的示例在一个自定义IAuthorizationPolicyProvider可能会很有用包括：

- 使用外部服务提供的策略评估。
- （适用于不同的房间号或年龄，例如）使用了大量的策略，因此它不适合将与每个单个授权策略添加 `AuthorizationOptions.AddPolicy` 调用。
- 在运行时基于外部数据源（如数据库）中的信息创建策略或通过其他机制动态确定授权要求。

查看或下载示例代码从AspNetCore GitHub 存储库。Aspnet/AuthSamples 存储库 ZIP 文件下载。解压缩*AuthSamples master.zip*文件。导航到*示例/CustomPolicyProvider*项目文件夹。

自定义策略检索

ASP.NET Core 应用使用的实现 `IAuthorizationPolicyProvider` 接口以检索授权策略。默认情况下DefaultAuthorizationPolicyProvider注册和使用。 `DefaultAuthorizationPolicyProvider` 返回从策略 `AuthorizationOptions` 中提供 `IServiceCollection.AddAuthorization` 调用。

可以通过注册其他自定义此行为 `IAuthorizationPolicyProvider` 在应用中的实现依赖关系注入容器。

`IAuthorizationPolicyProvider` 接口包含两个 Api:

- GetPolicyAsync返回具有给定名称的授权策略。
- GetDefaultPolicyAsync返回的默认授权策略 (用于策略 `[Authorize]` 而无需指定的策略的属性)。

通过实现这两个 Api，可以自定义提供的授权策略的方式。

参数化授权特性的示例

一种情况其中 `IAuthorizationPolicyProvider` 可启用自定义 `[Authorize]` 其要求取决于参数的属性。例如，在基于策略的授权文档中，基于年龄的 ("AtLeast21") 作为一个示例所使用的策略。如果在应用中的不同控制器操作应该对可用的用户*不同*年龄，可能会很有用具有许多不同的基于年龄的策略。而不是注册所有不同年龄基于策略的应用程序中会用 `AuthorizationOptions` ，可以生成动态使用自定义策略 `IAuthorizationPolicyProvider` 。要使用的策略更轻松，你可以批注具有类似的自定义授权属性操作 `[MinimumAgeAuthorize(20)]` 。

自定义授权属性

通过名称标识的授权策略。自定义 `MinimumAgeAuthorizeAttribute` 所述之前需要将参数映射到一个字符串，可用于检索相应的授权策略。您可以执行此操作通过派生自 `AuthorizeAttribute` 并使 `Age` 属性包装 `AuthorizeAttribute.Policy` 属性。

```
internal class MinimumAgeAuthorizeAttribute : AuthorizeAttribute
{
    const string POLICY_PREFIX = "MinimumAge";

    public MinimumAgeAuthorizeAttribute(int age) => Age = age;

    // Get or set the Age property by manipulating the underlying Policy property
    public int Age
    {
        get
        {
            if (int.TryParse(Policy.Substring(POLICY_PREFIX.Length), out var age))
            {
                return age;
            }
            return default(int);
        }
        set
        {
            Policy = $"{POLICY_PREFIX}{value.ToString()}";
        }
    }
}
```

此属性类型具有 `Policy` 字符串基于的硬编码的前缀（`"MinimumAge"`）和一个整数，通过构造函数中传递。

可以将其应用于操作中方式不同于其他 `Authorize` 属性，但前者作为参数的整数。

```
[MinimumAgeAuthorize(10)]
public IActionResult RequiresMinimumAge10()
```

自定义 IAuthorizationPolicyProvider

自定义 `MinimumAgeAuthorizeAttribute` 可以方便地对请求授权策略的任何所需的最短期限。要解决的下一个问题确保为所有这些不同年龄段的授权策略可用。这就是 `IAuthorizationPolicyProvider` 非常有用。

使用时 `MinimumAgeAuthorizationAttribute`，授权策略名称将遵循的模式 `"MinimumAge" + Age`，因此自定义 `IAuthorizationPolicyProvider` 应生成的授权策略：

- 分析将年龄字段从策略名称。
- 使用 `AuthorizationPolicyBuilder` 若要创建一个新 `AuthorizationPolicy`
- 向策略添加要求根据与保留 `AuthorizationPolicyBuilder.AddRequirements`。在其他情况下，可能会使用 `RequireClaim`， `RequireRole`，或 `RequireUserName` 相反。

```
internal class MinimumAgePolicyProvider : IAuthorizationPolicyProvider
{
    const string POLICY_PREFIX = "MinimumAge";

    // Policies are looked up by string name, so expect 'parameters' (like age)
    // to be embedded in the policy names. This is abstracted away from developers
    // by the more strongly-typed attributes derived from AuthorizeAttribute
    // (like [MinimumAgeAuthorize()] in this sample)
    public Task<AuthorizationPolicy> GetPolicyAsync(string policyName)
    {
        if (policyName.StartsWith(POLICY_PREFIX, StringComparison.OrdinalIgnoreCase) &&
            int.TryParse(policyName.Substring(POLICY_PREFIX.Length), out var age))
        {
            var policy = new AuthorizationPolicyBuilder();
            policy.AddRequirements(new MinimumAgeRequirement(age));
            return Task.FromResult(policy.Build());
        }

        return Task.FromResult<AuthorizationPolicy>(null);
    }
}
```

多个授权策略提供程序

使用自定义时 `IAuthorizationPolicyProvider` 实现中，请记住，ASP.NET Core 仅使用的一个实例 `IAuthorizationPolicyProvider` 。如果自定义提供程序不能提供的所有策略名称的授权策略，它应回退到备份的提供程序。策略名称可能包含来自的默认策略 `[Authorize]` 没有名称的属性。

例如，考虑应用程序需要自定义保留时间策略和更传统的基于角色的策略检索。此类应用可以使用自定义授权策略提供程序的：

- 尝试分析策略名称。
- 调入不同的策略提供程序（如 `DefaultAuthorizationPolicyProvider` ）如果策略名称不包含年龄。

默认策略

除了提供命名的授权策略，自定义 `IAuthorizationPolicyProvider` 需要实现 `GetDefaultPolicyAsync` 提供的授权策略 `[Authorize]` 属性没有指定策略名称。

在许多情况下，此授权属性仅要求身份验证的用户，这样你就可以通过调用必要的策略 `RequireAuthenticatedUser` ：

```
public Task<AuthorizationPolicy> GetDefaultPolicyAsync() =>
    Task.FromResult(new AuthorizationPolicyBuilder().RequireAuthenticatedUser().Build());
```

自定义的所有方面与 `IAuthorizationPolicyProvider` ，你可以自定义此操作，请根据需要。在某些情况下：

- 可能无法使用默认的授权策略。
- 检索默认策略可以委派给回退 `IAuthorizationPolicyProvider` 。

使用自定义 IAuthorizationPolicyProvider

若要使用自定义策略从 `IAuthorizationPolicyProvider` ，您必须：

- 注册的相应 `AuthorizationHandler` 类型使用依赖关系注入（中所述基于策略的授权），与所有基于策略的授权方案。
- 注册自定义 `IAuthorizationPolicyProvider` 应用程序的依赖关系注入服务集合中的类型（在 `Startup.ConfigureServices` ）来替换默认策略提供程序。

```
services.AddSingleton<IAuthorizationPolicyProvider, MinimumAgePolicyProvider>();
```

完成自定义 `IAuthorizationPolicyProvider` 示例现已推出aspnet/AuthSamples GitHub 存储库。

在 ASP.NET Core要求处理程序中的依赖关系注入

2018/7/30 • • Edit Online

授权处理程序必须进行注册在配置期间服务集合中 (使用依赖关系注入)。

假设你有想要评估的授权处理程序内的规则的存储库和服务集合中已注册该存储库。授权将解决，然后将它注入到您的构造函数。

例如，如果你想要使用 ASP。NET 的日志记录基础结构要注入 `ILoggerFactory` 到您的处理程序。此类处理程序可能如下所示：

```
public class LoggingAuthorizationHandler : AuthorizationHandler<MyRequirement>
   {
       ILogger _logger;

       public LoggingAuthorizationHandler(ILoggerFactory loggerFactory)
       {
           _logger = loggerFactory.CreateLogger(this.GetType().FullName);
       }

       protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, MyRequirement
requirement)
       {
           _logger.LogInformation("Inside my handler");
           // Check if the requirement is fulfilled.
           return Task.CompletedTask;
       }
   }
```

将注册处理程序替换 `services.AddSingleton()`：

```
services.AddSingleton<IAuthorizationHandler, LoggingAuthorizationHandler>();
```

该处理程序将你的应用程序启动时创建的实例并注入的已注册的 DI 将 `ILoggerFactory` 到您的构造函数。

> **NOTE**
> 使用实体框架的处理程序不应注册为单一实例。

ASP.NET Core 中基于资源的授权

2018/11/16 • • Edit Online

授权策略取决于要访问的资源。请考虑具有 author 属性的文档。允许仅作者更新文档。因此，文档必须检索从数据存储区进行授权评估之前。

数据绑定之前和之后，页面处理程序或加载文档的操作的执行进行属性评估。出于这些原因，使用声明性授权 `[Authorize]` 属性不足以满足需求。相反，可以调用自定义授权方法—称为样式*命令性授权*。

查看或下载示例代码（如何下载）。

使用受授权的用户数据创建 ASP.NET Core 应用包含的示例应用程序使用的基于资源的授权。

使用命令性授权

授权作为实现IAuthorizationService服务并注册内的服务集合中 `Startup` 类。该服务可通过依赖关系注入页面处理程序或进行操作。

```
public class DocumentController : Controller
{
    private readonly IAuthorizationService _authorizationService;
    private readonly IDocumentRepository _documentRepository;

    public DocumentController(IAuthorizationService authorizationService,
                              IDocumentRepository documentRepository)
    {
        _authorizationService = authorizationService;
        _documentRepository = documentRepository;
    }
```

`IAuthorizationService` 具有两个 `AuthorizeAsync` 方法重载：一个接受资源和策略名称和其他接受资源和要求来评估的列表。

```
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          IEnumerable<IAuthorizationRequirement> requirements);
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          string policyName);
```

```
Task<bool> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          IEnumerable<IAuthorizationRequirement> requirements);
Task<bool> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          string policyName);
```

在以下示例中，要保护的资源加载到自定义 `Document` 对象。`AuthorizeAsync` 重载进行调用以确定当前用户是否允许编辑提供的文档。自定义"EditPolicy"授权策略可分解为决策。请参阅自定义基于策略的授权创建授权策略的详细信息。

> **NOTE**
>
> 下面的代码示例假定已经运行了身份验证和组 `User` 属性。

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
            .AuthorizeAsync(User, Document, "EditPolicy");

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
        return new ForbidResult();
    }
    else
    {
        return new ChallengeResult();
    }
}
```

```
[HttpGet]
public async Task<IActionResult> Edit(Guid documentId)
{
    Document document = _documentRepository.Find(documentId);

    if (document == null)
    {
        return new NotFoundResult();
    }

    if (await _authorizationService
        .AuthorizeAsync(User, document, "EditPolicy"))
    {
        return View(document);
    }
    else
    {
        return new ChallengeResult();
    }
}
```

编写基于资源的处理程序

编写处理程序的基于资源的授权没有太大区别比编写普通要求处理程序。创建自定义要求类，并实现要求处理程序类。创建要求类的详细信息，请参阅要求。

处理程序类指定的要求和资源类型。例如，处理程序利用 `SameAuthorRequirement` 和一个 `Document` 资源如下所示：

```
public class DocumentAuthorizationHandler :
    AuthorizationHandler<SameAuthorRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   SameAuthorRequirement requirement,
                                                   Document resource)
    {
        if (context.User.Identity?.Name == resource.Author)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

public class SameAuthorRequirement : IAuthorizationRequirement { }
```

```
public class DocumentAuthorizationHandler :
    AuthorizationHandler<SameAuthorRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   SameAuthorRequirement requirement,
                                                   Document resource)
    {
        if (context.User.Identity?.Name == resource.Author)
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}

public class SameAuthorRequirement : IAuthorizationRequirement { }
```

在上述示例中，假设 `SameAuthorRequirement` 是一种特殊情况的多个泛型 `SpecificAuthorRequirement` 类。`SpecificAuthorRequirement` 类（未显示）包含 `Name` 属性表示的作者的名称。`Name` 属性可以设置为当前用户。

注册的要求和处理程序中的 `Startup.ConfigureServices`：

```
services.AddMvc();

services.AddAuthorization(options =>
{
    options.AddPolicy("EditPolicy", policy =>
        policy.Requirements.Add(new SameAuthorRequirement()));
});

services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationHandler>();
services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationCrudHandler>();
services.AddScoped<IDocumentRepository, DocumentRepository>();
```

操作要求

若要进行决策基于 CRUD（创建、读取、更新、删除）操作的结果，请使用OperationAuthorizationRequirement帮助器类。此类，可为每个操作类型编写单个处理程序而不是单独的类。若要使用它，提供一些操作名称：

```
public static class Operations
{
    public static OperationAuthorizationRequirement Create =
        new OperationAuthorizationRequirement { Name = nameof(Create) };
    public static OperationAuthorizationRequirement Read =
        new OperationAuthorizationRequirement { Name = nameof(Read) };
    public static OperationAuthorizationRequirement Update =
        new OperationAuthorizationRequirement { Name = nameof(Update) };
    public static OperationAuthorizationRequirement Delete =
        new OperationAuthorizationRequirement { Name = nameof(Delete) };
}
```

处理程序的实现，如下所示，使用 `OperationAuthorizationRequirement` 要求和 `Document` 资源：

```
public class DocumentAuthorizationCrudHandler :
    AuthorizationHandler<OperationAuthorizationRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   OperationAuthorizationRequirement requirement,
                                                   Document resource)
    {
        if (context.User.Identity?.Name == resource.Author &&
            requirement.Name == Operations.Read.Name)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

```
public class DocumentAuthorizationCrudHandler :
    AuthorizationHandler<OperationAuthorizationRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   OperationAuthorizationRequirement requirement,
                                                   Document resource)
    {
        if (context.User.Identity?.Name == resource.Author &&
            requirement.Name == Operations.Read.Name)
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

前面的处理程序验证该操作使用的资源、用户的标识和需求的 `Name` 属性。

若要调用的操作的资源处理程序，指定该操作时调用 `AuthorizeAsync` 页面处理程序或操作中。下面的示例确定是否允许经过身份验证的用户若要查看提供的文档。

> **NOTE**
>
> 下面的代码示例假定已经运行了身份验证和组 `User` 属性。

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
            .AuthorizeAsync(User, Document, Operations.Read);

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
        return new ForbidResult();
    }
    else
    {
        return new ChallengeResult();
    }
}
```

如果授权成功，则返回查看文档的页。如果授权失败而用户进行身份验证，返回 `ForbidResult` 通知授权失败的任何身份验证中间件。一个 `ChallengeResult` 时必须执行身份验证，将返回。对于交互式浏览器客户端，可能需要将用户重定向到登录页。

```
[HttpGet]
public async Task<IActionResult> View(Guid documentId)
{
    Document document = _documentRepository.Find(documentId);

    if (document == null)
    {
        return new NotFoundResult();
    }

    if (await _authorizationService
        .AuthorizeAsync(User, document, Operations.Read))
    {
        return View(document);
    }
    else
    {
        return new ChallengeResult();
    }
}
```

如果授权成功，则返回文档的视图。如果授权失败，返回 `ChallengeResult` 授权失败，并将中间件可以采取适当的响应将通知任何身份验证中间件。相应的响应可能返回 401 或 403 状态代码。对于交互式浏览器客户端，这可能意味着将用户重定向到登录页。

ASP.NET Core mvc 视图基于授权

2018/7/30 • • Edit Online

开发人员通常想要显示、隐藏或以其他方式修改基于当前用户标识的用户界面。您可以访问通过 MVC 视图中的授权服务依赖关系注入。若要将授权服务注入到 Razor 视图中，使用 `@inject` 指令：

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService
```

如果希望每个视图中的授权服务，将放置 `@inject` 指令插入 *_ViewImports.cshtml*的文件 *视图*目录。有关详细信息，请参阅视图中的依赖关系注入。

使用注入的授权服务调用 `AuthorizeAsync` 中完全相同的方式会检查期间基于资源的授权：

- ASP.NET Core 2.x
- ASP.NET Core 1.x

```
@if ((await AuthorizationService.AuthorizeAsync(User, "PolicyName")).Succeeded)
{
    <p>This paragraph is displayed because you fulfilled PolicyName.</p>
}
```

在某些情况下，该资源将视图模型。调用 `AuthorizeAsync` 中完全相同的方式会检查期间基于资源的授权：

- ASP.NET Core 2.x
- ASP.NET Core 1.x

```
@if ((await AuthorizationService.AuthorizeAsync(User, Model, Operations.Edit)).Succeeded)
{
    <p><a class="btn btn-default" role="button"
        href="@Url.Action("Edit", "Document", new { id = Model.Id })">Edit</a></p>
}
```

在前面的代码中，模型应采取的策略评估的资源作为传递纳入考虑范围。

> **WARNING**
>
> 不要依赖于作为唯一的授权检查的应用程序的 UI 元素的切换可见性。隐藏 UI 元素可能无法完全阻止访问到其关联的控制器操作。例如，考虑前面的代码段中的按钮。用户可以调用 `Edit` 操作方法，如果他或她知道相对资源 URL 是 */Document/Edit/1*。出于此原因，`Edit` 操作方法应执行其自己的授权检查。

使用 ASP.NET Core 中的特定方案授权

2018/10/26 • • Edit Online

在某些情况下，如单页面应用程序 (Spa) 是通常使用多个身份验证方法。例如，应用可能会使用基于 cookie 的身份验证登录和 JWT 持有者身份验证的 JavaScript 请求。在某些情况下，应用程序可能具有的身份验证处理程序的多个实例。例如，其中一个包含基本身份标识的两个 cookie 处理程序，另一个时创建已触发多重身份验证 (MFA)。用户请求需要额外的安全性的操作，因此，可能会触发 MFA。

- ASP.NET Core 2.x
- ASP.NET Core 1.x

在身份验证过程中配置身份验证服务时，名为身份验证方案。例如：

```
public void ConfigureServices(IServiceCollection services)
{
    // Code omitted for brevity

    services.AddAuthentication()
        .AddCookie(options => {
            options.LoginPath = "/Account/Unauthorized/";
            options.AccessDeniedPath = "/Account/Forbidden/";
        })
        .AddJwtBearer(options => {
            options.Audience = "http://localhost:5001/";
            options.Authority = "http://localhost:5000/";
        });
```

已在前面的代码中，添加两个身份验证处理程序：一个用于 cookie，一个为持有者。

> **NOTE**
>
> 指定的默认方案会导致 `HttpContext.User` 属性设置为该标识。如果不需要该行为，请禁用调用的无参数形式 `AddAuthentication`。

选择具有 Authorize 属性的使用方案

在授权，终端应用指示要使用的处理程序。选择与应用程序将授权通过将传递到身份验证方案的以逗号分隔列表的处理程序 `[Authorize]`。`[Authorize]` 特性指定身份验证方案或方案使用而不考虑是否配置默认值。例如：

- ASP.NET Core 2.x
- ASP.NET Core 1.x

```
[Authorize(AuthenticationSchemes = AuthSchemes)]
public class MixedController : Controller
    // Requires the following imports:
    // using Microsoft.AspNetCore.Authentication.Cookies;
    // using Microsoft.AspNetCore.Authentication.JwtBearer;
    private const string AuthSchemes =
        CookieAuthenticationDefaults.AuthenticationScheme + "," +
        JwtBearerDefaults.AuthenticationScheme;
```

在前面的示例中，cookie 和持有者处理程序运行，并有机会创建并追加当前用户的标识。通过指定一种方案，在运行相应的处理程序。

- ASP.NET Core 2.x
- ASP.NET Core 1.x

```
[Authorize(AuthenticationSchemes =
    JwtBearerDefaults.AuthenticationScheme)]
public class MixedController : Controller
```

在上述代码中，仅具有"Bearer"方案处理程序运行。任何基于 cookie 的标识将被忽略。

选择使用策略的方案

如果想要指定在所需的方案策略，可以设置 `AuthenticationSchemes` 集合添加你的策略时：

```
services.AddAuthorization(options =>
{
    options.AddPolicy("Over18", policy =>
    {
        policy.AuthenticationSchemes.Add(JwtBearerDefaults.AuthenticationScheme);
        policy.RequireAuthenticatedUser();
        policy.Requirements.Add(new MinimumAgeRequirement());
    });
});
```

在上述示例中，"Over18"策略只在运行针对"Bearer"处理程序创建的标识。通过设置使用策略 `[Authorize]` 特性的 `Policy` 属性：

```
[Authorize(Policy = "Over18")]
public class RegistrationController : Controller
```

使用多个身份验证方案

某些应用可能需要支持多个类型的身份验证。例如，您的应用程序可能会对用户从 Azure Active Directory 和用户数据库进行身份验证。另一个示例是一个应用，用户从 Active Directory 联合身份验证服务和 Azure Active Directory B2C 进行身份验证。在这种情况下，应用程序应接受来自多个颁发者的 JWT 持有者令牌。

添加你想要接受的所有身份验证方案。例如，以下代码中 `Startup.ConfigureServices` 添加两个 JWT 持有者身份验证方案使用不同的颁发者：

```
public void ConfigureServices(IServiceCollection services)
{
    // Code omitted for brevity

    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://localhost:5000/identity/";
        })
        .AddJwtBearer("AzureAD", options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://login.microsoftonline.com/eb971100-6f99-4bdc-8611-1bc8edd7f436/";
        });
}
```

> **NOTE**
> 只有一个 JWT 持有者身份验证注册的默认身份验证方案使用 `JwtBearerDefaults.AuthenticationScheme` 。其他身份验证必须是唯一的身份验证方案使用其注册。

下一步是更新默认授权策略，以接受这两种身份验证方案。例如：

```
public void ConfigureServices(IServiceCollection services)
{
    // Code omitted for brevity

    services.AddAuthorization(options =>
    {
        var defaultAuthorizationPolicyBuilder = new AuthorizationPolicyBuilder(
            JwtBearerDefaults.AuthenticationScheme,
            "AzureAD");
        defaultAuthorizationPolicyBuilder =
            defaultAuthorizationPolicyBuilder.RequireAuthenticatedUser();
        options.DefaultPolicy = defaultAuthorizationPolicyBuilder.Build();
    });
}
```

如重写默认授权策略，则可以使用简单的 `[Authorize]` 控制器中的属性。然后，该控制器使用由第一个或第二个颁发者颁发的 JWT 接受请求。

ASP.NET Core 数据保护

2018/10/26 • • Edit Online

Web 应用程序通常需要将安全敏感数据存储。Windows 桌面应用程序提供 DPAPI，但这并不适合 web 应用程序。ASP.NET Core 数据保护堆栈提供一个简单、易于使用的加密 API，开发人员可以使用来保护数据，包括密钥管理和旋转。

ASP.NET Core 数据保护堆栈旨在用作的长期替代<machineKey>在 ASP.NET 中的元素 1.x-4.x。它旨在解决许多旧加密堆栈的不足之处同时为大多数现代应用程序可能会遇到的使用情况下提供开箱解决方案。

问题陈述

总体问题语句可以简洁地表示中的单个句子：我需要保留以供将来检索受信任的信息，但我不信任持久性机制。在 web 术语中，这可能会根据写入"我需要为往返通过不受信任的客户端的受信任状态。"

此规范的示例是身份验证 cookie 或持有者令牌。服务器会生成"我是 Groot，具有 xyz 的权限"令牌并将它传递到客户端。在将来某天的客户端将显示该令牌发送回服务器，但该服务器需要某种类型的客户端尚未伪造令牌的保障。因此第一个要求：真实性（也称为 完整性、防篡改攻击）。

持久化的状态由服务器信任的因为我们预计这种状态可能包含特定于操作环境的信息。这可能是在文件路径、权限、句柄或其他间接引用的窗体或其他一些特定于服务器的数据。此类信息通常不应公开到不受信任的客户端。因此第二个要求：保密性。

最后，现代应用程序的组件化，因为我们已了解是各个组件将想要充分利用此系统而不考虑其他组件的系统中。例如，如果持有者令牌组件使用的此堆栈，它应运行不受干扰地从一种反 CSRF 机制，也可能使用相同的堆栈。因此最后一项要求：隔离。

我们可以进一步约束以缩小我们的要求的范围。我们假定加密系统内的所有服务都是同等信任和数据无需生成或在我们的直接控制服务外部使用。此外，我们需要的操作是尽可能快，因为 web 服务的每个请求可能会经过加密系统的一个或多个时间。这使得对称加密非常适合用于我们的方案，和我们可以折扣此类具有所需的时间之前的非对称加密。

设计理念

我们首先使用现有的堆栈识别问题。一旦我们有的我们的调查了现有解决方案布局而得出的结论是，没有现有的解决方案没有我们查找的功能。然后，我们设计基于多个指导原则的解决方案。

- 系统应提供简单的配置。理想情况下系统将为零配置，并且开发人员可以快速开始使用。在开发人员需要配置一个特定方面（如密钥存储库）的情况下，应考虑到使这些特定的配置更简单。
- 提供一个简单的面向消费者的 API。Api 应易于正确使用且难以使用不正确。
- 开发人员不应了解密钥管理原则。系统应处理算法选择和开发人员的代表的密钥生存期。理想情况下，开发人员甚至不应具有访问原始密钥材料。
- 应尽可能的静态保护密钥。系统应找出相应的默认保护机制，并将其自动应用。

记住这些原则与我们开发了一个简单易于使用数据保护堆栈。

ASP.NET Core 数据保护 Api 主要不用于机密负载的无限期持久性。等其他技术Windows CNG DPAPI并Azure Rights Management更适合的无限期存储方案，并且必须相应地强密钥管理功能。也就是说，无需进行任何开发人员禁止使用 ASP.NET Core 数据保护 Api 进行长期保护的机密数据。

读者

数据保护系统分为五个主要的软件包。这些 Api 的各个方面定位三个主要受众;

1. 使用者 Api 概述目标应用程序和框架开发人员。

   "我不想要了解有关在堆栈的运行方式或配置方式。我只是想要执行某项操作中的最简单的方式尽可能高概率为成功使用 Api。"

2. 配置 Api目标应用程序开发人员和系统管理员。

   "我需要告诉我的环境需要非默认路径或设置数据保护系统。"

3. 扩展性 Api 目标，开发人员负责实现自定义策略。这些 Api 的使用情况将限于极少数情况下并遇到安全注意开发人员。

   "我需要替换的整个系统内组件，因为我有真正独特的行为要求。我愿意以了解不常见使用的 API 图面部分，若要生成可满足我的要求的插件。"

包布局

数据保护堆栈包含五个包。

- Microsoft.AspNetCore.DataProtection.Abstractions包含IDataProtectionProvider和IDataProtector接口，用于创建数据保护服务。它还包含用于处理这些类型的有用的扩展方法 (例如，IDataProtector.Protect)。如果数据保护系统进行实例化其他位置和要使用 API，引用 `Microsoft.AspNetCore.DataProtection.Abstractions` 。

- Microsoft.AspNetCore.DataProtection包含数据保护系统，包括核心加密操作、密钥管理、配置和可扩展性的核心实现。若要实例化的数据保护系统 (例如，将其添加到IServiceCollection) 或修改或扩展其行为，引用 `Microsoft.AspNetCore.DataProtection` 。

- Microsoft.AspNetCore.DataProtection.Extensions包含其他一些 Api，开发人员可能会发现很有用，但这不属于核心包中。例如，此包包含工厂方法来实例化数据保护系统，而无需依赖关系注入在文件系统上存储在一个位置的密钥 (请参阅DataProtectionProvider)。它还包含用于限制受保护负载的生存期的扩展方法 (请参阅ITimeLimitedDataProtector)。

- Microsoft.AspNetCore.DataProtection.SystemWeb可以安装到现有 ASP.NET 4.x 应用重定向其 `<machineKey>` 操作，使用新的 ASP.NET Core 数据保护堆栈。有关详细信息，请参阅 将在 ASP.NET Core ASP.NET machineKey 。

- Microsoft.AspNetCore.Cryptography.KeyDerivation提供 PBKDF2 密码哈希例程的实现，并可以由必须安全地处理用户密码的系统。有关详细信息，请参阅 在 ASP.NET Core中的哈希密码 。

其他资源

在 Web 场中托管 ASP.NET Core

开始使用 ASP.NET Core 中的数据保护 Api

2018/8/22 • • Edit Online

在其最简单、保护数据包括以下步骤：

1. 从数据保护提供程序创建数据保护程序。
2. 调用 `Protect` 你想要保护的数据的方法。
3. 调用 `Unprotect` 想要将返回转换为纯文本格式的数据的方法。

大多数框架和应用模型，如 ASP.NET Core 或 SignalR 中，已配置数据保护系统，并将其添加到服务容器还通过依赖关系注入来访问。下面的示例演示如何配置依赖关系注入的服务容器和注册数据保护堆栈、接收通过 DI 的数据保护提供程序、创建保护程序和数据保护，然后取消。

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload = _protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZZhlAlTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
 */
```

创建一个保护程序时必须提供一个或多个目标字符串。一个字符串，目的提供了使用者之间的隔离。例如，使用"green"的目的字符串创建的保护程序将无法取消保护数据的"紫色"目的提供的保护程序。

TIP

实例 `IDataProtectionProvider` 和 `IDataProtector` 是线程安全的多个调用方。它可用于的一个组件，获取对的引用后 `IDataProtector` 通过调用 `CreateProtector`，它将该引用用于对多个调用 `Protect` 和 `Unprotect`。

调用 `Unprotect` 将引发 CryptographicException，如果受保护的有效负载不能验证或中译解出来。某些组件可能想要忽略错误期间取消保护操作;一个组件，它读取身份验证 cookie 可能处理此错误，并将请求视为如同它在所有具有任何 cookie 而无法完全是请求。需要此行为的组件应专门捕获 CryptographicException，而不是抑制所有异常。

有关 ASP.NET Core 的使用者 Api 概述

2018/7/6 • • Edit Online

`IDataProtectionProvider` 和 `IDataProtector` 接口是通过该使用者使用数据保护系统的基本接口。它们位于Microsoft.AspNetCore.DataProtection.Abstractions包。

IDataProtectionProvider

提供程序接口表示数据保护系统的根目录。它不能直接用于保护或取消保护数据。相反，使用者必须获得对引用 `IDataProtector` 通过调用 `IDataProtectionProvider.CreateProtector(purpose)`，其中，目的是描述预期的使用者用例的字符串。请参阅目的字符串着眼于此参数以及如何选择适当的值的更多的信息。

IDataProtector

保护程序接口将返回通过调用 `CreateProtector`，和它的使用者可以用于执行此接口保护或取消保护操作。

若要保护的数据片段，将数据传递给 `Protect` 方法。基本接口定义的将 byte []-> byte [] 的方法，但没有还将字符串转换的重载（提供作为扩展方法）-> 字符串。提供两种方法的安全性是相同的;开发人员应选择任何重载是最方便对其用例。而不考虑的重载选择，则返回保护方法现在受保护（enciphered 和防考验）和应用程序可以将其发送到不受信任的客户端。

若要取消对以前受保护的数据片段的保护，将传递到受保护的数据 `Unprotect` 方法。(有 byte []-基于和基于字符串的重载，为开发人员方便起见。)如果受保护的负载由以前调用生成 `Protect` 此同一 `IDataProtector`、`Unprotect` 方法将返回原始的未受保护的负载。如果已被篡改或已由不同的受保护的负载 `IDataProtector`、`Unprotect` 方法会引发 CryptographicException。

与不同的相同的概念 `IDataProtector` ties 回目的的概念。如果两个 `IDataProtector` 实例生成从同一根 `IDataProtectionProvider` 但通过不同的用途的调用中的字符串 `IDataProtectionProvider.CreateProtector`，则它们正在被视为不同的保护程序，和一个将无法取消保护由其他生成的负载。

使用这些接口

对于 DI 感知的组件，预期的用法是，该组件需要 `IDataProtectionProvider` 在其构造函数的参数，并实例化组件时，DI 系统可以自动提供此服务。

> **NOTE**
>
> 某些应用程序（如控制台应用程序或 ASP.NET 4.x 应用程序）可能不是 DI 感知因此不能使用此处所述的机制。有关这些方案，请查阅非 DI 感知的情境有关获取的实例的详细信息的文档 `IDataProtection` 而无需通过 DI 的提供程序。

下面的示例演示三个概念：

1. 添加数据保护系统到服务容器
2. 使用 DI 接收的实例 `IDataProtectionProvider`，和
3. 创建 `IDataProtector` 从 `IDataProtectionProvider` 并使用它来保护和取消保护数据。

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload = _protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZZhlAlTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
 */
```

包 Microsoft.AspNetCore.DataProtection.Abstractions 包含的扩展方法 `IServiceProvider.GetDataProtector` 为开发人员方便起见。它作为单个操作封装这两个检索 `IDataProtectionProvider` 从服务提供商和调用 `IDataProtectionProvider.CreateProtector` 。下面的示例演示其用法。

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // get an IDataProtector from the IServiceProvider
        var protector = services.GetDataProtector("Contoso.Example.v2");
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
    }
}
```

TIP

实例 `IDataProtectionProvider` 和 `IDataProtector` 是线程安全的多个调用方。它具有一个组件获取的引用后应，`IDataProtector` 通过调用 `CreateProtector`，它将使用该引用，以便多个调用 `Protect` 和 `Unprotect`。调用 `Unprotect` 将引发 CryptographicException，如果无法验证或中译解出来的受保护的负载。某些组件可能想要忽略错误期间取消保护操作;组件它读取身份验证 cookie 可能处理此错误和请求则将视为根本具有任何 cookie，而不使迫切地请求失败。需要此行为的组件应专门捕获 CryptographicException，而不是忽略所有异常。

在 ASP.NET Core 目的字符串

2018/7/6 • • Edit Online

使用组件 `IDataProtectionProvider` 必须传递一个唯一*目的*参数 `CreateProtector` 方法。目的*参数*是固有的数据保护系统中，安全的因为它提供了加密的使用者之间的隔离，即使根加密密钥是相同的。

当使用者指定目的时，目的字符串用于以及根加密密钥派生加密子项唯一到该使用者。这将隔离应用程序中其他加密使用者的使用者：其他组件不能读取其有效负载，并且它无法读取任何其他组件的负载。这种隔离还会呈现不可行的整个类别的针对组件的攻击。

在上图中，`IDataProtector` 实例 A 和 B**无法**读取对方的负载，仅自己。

目的字符串不一定是机密。它只应是唯一其他任何功能良好组件不断将提供相同的目的字符串。

> **TIP**
>
> 使用该组件使用的数据保护 Api 的命名空间和类型名称是一个很好的经验法则，如下所示此信息将不会发生冲突的做法。
>
> Contoso 编写的组件，它负责 minting 持有者令牌可能 Contoso.Security.BearerToken 用作其用途字符串。或者-甚至更好地-它可能使用 Contoso.Security.BearerToken.v1 作为其用途字符串。追加版本号，为将来的版本，以 Contoso.Security.BearerToken.v2 用作其用途，并且负载到位会完全独立于另一个不同的版本。

由于目的参数 `CreateProtector` 是一个字符串数组，上述无法已改为指定为 `[ "Contoso.Security.BearerToken", "v1" ]`。这允许建立目的的层次结构，并打开与数据保护系统的多租户方案的可能性。

> **WARNING**
>
> 组件不应允许不受信任的用户输入要用于目的链的输入的唯一来源。
>
> 例如，考虑 Contoso.Messaging.SecureMessage 负责存储安全消息的组件。如果安全消息的组件是为了调用 `CreateProtector([ username ])`，则恶意用户可能创建的帐户用户名"Contoso.Security.BearerToken"在尝试获取要调用的组件 `CreateProtector([ "Contoso.Security.BearerToken" ])`，从而无意中造成安全消息传送可以预见到身份验证令牌的 mint 负载的系统。
>
> 将消息传递组件的更好目的链 `CreateProtector([ "Contoso.Messaging.SecureMessage", "User: username" ])`，它提供适当的隔离。

通过提供隔离和行为 `IDataProtectionProvider`， `IDataProtector`，和目的如下所示：

- 为给定 `IDataProtectionProvider` 对象，`CreateProtector` 方法将创建 `IDataProtector` 对象唯一绑定到同时 `IDataProtectionProvider` 对象创建它，并已传递到方法的目的参数。
- 目的参数不得为空。（如果目的指定为一个数组，这意味着数组不能为零长度和数组的所有元素必须都为非 null。）非空字符串目的从技术上讲允许但不建议这样做。
- 两个用途自变量是等效的当且仅当它们包含相同的字符串（使用序号比较器）中的顺序相同。单一用途参数等效于相应的单个元素目的数组。
- 两个 `IDataProtector` 对象相等，当且仅当它们在创建等效项从 `IDataProtectionProvider` 对象，并且等效目的形参。
- 为给定 `IDataProtector` 对象、调用 `Unprotect(protectedData)` 将返回原始 `unprotectedData` 当且仅当 `protectedData := Protect(unprotectedData)` 有关等效 `IDataProtector` 对象。

> **NOTE**
>
> 我们不会考虑的如果某些组件有意选择已知与另一个组件发生冲突的目的字符串。此类组件将考虑恶意内容，实质上是和此系统并不旨在中，恶意代码已在运行的工作进程内提供的安全保证。

目的层次结构和 ASP.NET Core 中的多租户

2018/8/23 • • Edit Online

由于 `IDataProtector` 也是隐式 `IDataProtectionProvider`，目的可以链接在一起。在这种情况下，`provider.CreateProtector([ "purpose1", "purpose2" ])` 等效于 `provider.CreateProtector("purpose1").CreateProtector("purpose2")`。

这允许通过数据保护系统的一些有趣的层次结构关系。在上例中的Contoso.Messaging.SecureMessage，可以调用 SecureMessage 组件 `provider.CreateProtector("Contoso.Messaging.SecureMessage")` 一次前期和到专用缓存的结果 `_myProvider` 字段。然后可以通过调用创建将来的保护程序 `_myProvider.CreateProtector("User: username")`，并且这些保护程序将用于保护单个消息。

这也可以被翻转。哪些主机可以使用其自己的身份验证和状态管理系统配置（一个 CMS 似乎是合理）的多个租户，以及每个租户，请考虑单个逻辑应用程序。总括性应用程序具有单一的主提供程序，并且它将调用 `provider.CreateProtector("Tenant 1")` 和 `provider.CreateProtector("Tenant 2")` 为每个租户提供其自己的数据保护系统的独立的切片。租户然后可以派生他们自己单独的保护程序，基于其自身的需求，但不管如何努力尝试不能创建其发生冲突的保护程序与其他任何租户系统中。以图形方式，这表示如下所示。

> **WARNING**
>
> 此操作假定涵盖性应用程序控制哪些 Api 可供各个租户和租户不能在服务器上执行任意代码。如果租户可以执行任意代码，它们可以执行专用的反射来中断隔离保证，或它们只是无法直接读取主密钥密钥材料和派生任何子项力度。

数据保护系统实际上使用其默认的箱配置中的多租户的排序。默认情况下工作进程帐户的用户配置文件文件夹（或用于 IIS 应用程序池标识注册表）中存储主密钥的密钥材料。但它实际上相当通常使用单个帐户来运行多个应用程序，并因此所有这些应用程序将最终共享主密钥材料。若要解决此问题，数据保护系统将自动作为整体的用途链中的第一个元素插入唯一对每个应用程序标识符。此隐式提供给将单个应用程序隔离从另一个方法是有效地将每个应用程序，如系统和保护程序的创建过程中唯一租户看起来与上面的图像相同。

在 ASP.NET Core中的哈希密码

2018/9/18 • • Edit Online

数据保护代码库包括包*Microsoft.AspNetCore.Cryptography.KeyDerivation*其中包含加密密钥派生函数。此包是一个独立组件，并不依赖于数据保护系统的其余部分。可以完全独立地使用它。源并存的数据保护代码库为方便起见。

包目前提供一种方法 `KeyDerivation.Pbkdf2` 允许使用密码哈希PBKDF2 算法。此 API 是非常类似于.NET Framework 的现有Rfc2898DeriveBytes 类型，但具有三个重要的区别：

1. `KeyDerivation.Pbkdf2` 方法支持使用多个 PRFs（目前 `HMACSHA1`， `HMACSHA256`，和 `HMACSHA512`），而 `Rfc2898DeriveBytes` 类型仅支持 `HMACSHA1`。
2. `KeyDerivation.Pbkdf2` 方法检测到当前操作系统，并尝试选择提供更好的性能，在某些情况下的例程的经过最完美优化的实现。（在 Windows 8 中，它提供了大约 10 倍的吞吐量 `Rfc2898DeriveBytes`。）
3. `KeyDerivation.Pbkdf2` 方法需要调用方指定所有参数（给加盐，PRF 和迭代次数）。`Rfc2898DeriveBytes` 类型提供的这些默认值。

```
using System;
using System.Security.Cryptography;
using Microsoft.AspNetCore.Cryptography.KeyDerivation;

public class Program
{
    public static void Main(string[] args)
    {
        Console.Write("Enter a password: ");
        string password = Console.ReadLine();

        // generate a 128-bit salt using a secure PRNG
        byte[] salt = new byte[128 / 8];
        using (var rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(salt);
        }
        Console.WriteLine($"Salt: {Convert.ToBase64String(salt)}");

        // derive a 256-bit subkey (use HMACSHA1 with 10,000 iterations)
        string hashed = Convert.ToBase64String(KeyDerivation.Pbkdf2(
            password: password,
            salt: salt,
            prf: KeyDerivationPrf.HMACSHA1,
            iterationCount: 10000,
            numBytesRequested: 256 / 8));
        Console.WriteLine($"Hashed: {hashed}");
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter a password: Xtw9NMgx
 * Salt: NZsP6NnmfBuYeJrrAKNuVQ==
 * Hashed: /OOoOer10+tGwTRDTrQSoeCxVTFr6dtYly7d0cPxIak=
 */
```

请参阅源代码有关 ASP.NET Core 标识 `PasswordHasher` 类型实际用例。

限制在 ASP.NET Core 的受保护负载的生存期

2018/7/6 • • Edit Online

有一些应用程序开发人员希望创建将在一段时间后过期的受保护的负载的情形。例如，受保护的负载可能表示只应为有效一小时的密码重置令牌。当然也可以为开发人员创建他们自己包含的嵌入的到期日期的负载格式和高级开发人员可能想要执行此操作，但对于开发人员的大多数管理这些过期时间可能变得需要很长时间。

若要简化此过程对于我们的开发人员受众，包Microsoft.AspNetCore.DataProtection.Extensions用于创建在一段时间后自动过期的负载包含实用工具 Api。这些 Api 的挂起注销 `ITimeLimitedDataProtector` 类型。

API 使用情况

`ITimeLimitedDataProtector` 接口是用于保护和取消保护的限时 / 自即将到期的负载的核心接口。若要创建的实例 `ITimeLimitedDataProtector`，你需要先正则表达式的实例IDataProtector构造具有特定的用途。一次 `IDataProtector` 实例可用时，调用 `IDataProtector.ToTimeLimitedDataProtector` 扩展方法，能获得重新保护程序内置到期功能。

`ITimeLimitedDataProtector` 公开以下 API 图面和扩展方法：

- CreateProtector（字符串目的）：ITimeLimitedDataProtector-此 API 已类似于现有 `IDataProtectionProvider.CreateProtector` 在于它可以用于创建目的链从根的限时保护程序。
- 保护（字节 [] 纯文本，DateTimeOffset 到期）：byte]
- 保护（字节 [] 纯文本、TimeSpan 生存期）：byte]
- 保护（字节 [] 纯文本）：byte]
- 保护（字符串纯文本，DateTimeOffset 到期）：字符串
- 保护（TimeSpan 生存期中的字符串纯文本）：字符串
- 保护（字符串纯文本）：字符串

除了核心 `Protect` 方法需要仅是纯文本形式，存在一些新重载允许指定的负载的到期日期。到期日期可以指定为绝对日期（通过 `DateTimeOffset`）或相对的时间（从当前系统时间，通过 `TimeSpan`）。如果调用不带过期的重载，则负载假定为永远不会过期。

- 取消保护 (字节 [] protectedData，出 DateTimeOffset 到期): byte]
- 取消保护 (字节 [] protectedData): byte]
- 取消保护 (出 DateTimeOffset 过期字符串 protectedData): 字符串
- 取消保护 (字符串 protectedData): 字符串

`Unprotect` 方法返回原始的未受保护的数据。如果尚未超过负载，绝对过期则返回为可选输出参数以及原始的未受保护数据。如果负载已过期，则取消保护方法的所有重载将都引发 CryptographicException。

> **WARNING**
>
> 它具有不建议使用这些 Api 来保护需要长期或无限期暂留的负载。"，我可以承受为要在一个月后永久性不可恢复的受保护负载？" 可用作一个很好的经验法则;如果问题的回答是没有然后开发人员应考虑备用 Api。

使用下面的示例非 DI 代码路径用于实例化数据保护系统的。若要运行此示例，请确保先添加对 Microsoft.AspNetCore.DataProtection.Extensions 包的引用。

```
using System;
using System.IO;
using System.Threading;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // create a protector for my application

        var provider = DataProtectionProvider.Create(new DirectoryInfo(@"c:\myapp-keys\"));
        var baseProtector = provider.CreateProtector("Contoso.TimeLimitedSample");

        // convert the normal protector into a time-limited protector
        var timeLimitedProtector = baseProtector.ToTimeLimitedDataProtector();

        // get some input and protect it for five seconds
        Console.Write("Enter input: ");
        string input = Console.ReadLine();
        string protectedData = timeLimitedProtector.Protect(input, lifetime: TimeSpan.FromSeconds(5));
        Console.WriteLine($"Protected data: {protectedData}");

        // unprotect it to demonstrate that round-tripping works properly
        string roundtripped = timeLimitedProtector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped data: {roundtripped}");

        // wait 6 seconds and perform another unprotect, demonstrating that the payload self-expires
        Console.WriteLine("Waiting 6 seconds...");
        Thread.Sleep(6000);
        timeLimitedProtector.Unprotect(protectedData);
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello!
 * Protected data: CfDJ8Hu5z0zwxn...nLk7Ok
 * Round-tripped data: Hello!
 * Waiting 6 seconds...
 * <<throws CryptographicException with message 'The payload expired at ...'>>

 */
```

取消保护已吊销在 ASP.NET Core 中键的有效负载

2018/10/26 • • Edit Online

ASP.NET Core 数据保护 Api 主要不用于机密负载的无限期持久性。等其他技术Windows CNG DPAPI并Azure Rights Management更适合的无限期存储方案，并且必须相应地强密钥管理功能。也就是说，无需进行任何开发人员禁止使用 ASP.NET Core 数据保护 Api 进行长期保护的机密数据。永远不会删除密钥从密钥环，因此 `IDataProtector.Unprotect` 始终可以恢复现有的负载，只要键是可用且有效。

但是，则会产生问题而开发人员尝试取消保护数据作为保护已撤消的密钥，`IDataProtector.Unprotect` 这种情况下将引发异常。这可能是相当不错的短期或临时负载（如身份验证令牌），如轻松可以通过在系统中，重新创建这些类型的有效负载和最坏的情形站点访问者可能需要重新登录。但对于持久化有效负载，让 `Unprotect` throw 可能导致不可接受的数据丢失。

IPersistedDataProtector

若要支持允许有效负载为即使在面临吊销密钥时未受保护的方案，数据保护系统包含 `IPersistedDataProtector` 类型。若要获取的实例 `IPersistedDataProtector`，只需获取的实例 `IDataProtector` 在正常情况下，请尝试强制转换 `IDataProtector` 到 `IPersistedDataProtector`。

> **NOTE**
>
> 不是所有 `IDataProtector` 实例可以强制转换为 `IPersistedDataProtector`。开发人员应使用C#运算符或类似引起无效强制转换，以避免运行时异常和它们应准备好适当地处理失败的情况。

`IPersistedDataProtector` 公开了以下 API 图面：

```
DangerousUnprotect(byte[] protectedData, bool ignoreRevocationErrors,
     out bool requiresMigration, out bool wasRevoked) : byte[]
```

此 API 采用受保护的负载（作为字节数组），并返回未受保护的有效负载。没有任何基于字符串的重载。这两个输出参数如下所示。

- `requiresMigration`：将设置为 true，如果用于保护此有效负载的密钥不再是活动的默认密钥，例如，用来保护此有效负载的关键是旧密钥滚动操作以来已执行的位置。调用方可能想要考虑重新保护的有效负载，具体取决于其业务需求。
- `wasRevoked`：将设置为 true，如果用于保护此有效负载的密钥已被吊销。

> **WARNING**
>
> 传递时应格外小心 `ignoreRevocationErrors: true` 到 `DangerousUnprotect` 方法。调用此方法后的，如果 `wasRevoked` 值为 true，然后用来保护此有效负载的密钥已被吊销，并且有效负载的真实性应将其视为可疑。在这种情况下，仅继续操作上不受保护的有效负载有一定程度上，它是可信的例如保证单独来自安全的数据库，而不是由不受信任的 web 客户端发送。

```
using System;
using System.IO;
using System.Text;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;
```

```
public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // get a protector and perform a protect operation
        var protector = services.GetDataProtector("Sample.DangerousUnprotect");
        Console.Write("Input: ");
        byte[] input = Encoding.UTF8.GetBytes(Console.ReadLine());
        var protectedData = protector.Protect(input);
        Console.WriteLine($"Protected payload: {Convert.ToBase64String(protectedData)}");

        // demonstrate that the payload round-trips properly
        var roundTripped = protector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped payload: {Encoding.UTF8.GetString(roundTripped)}");

        // get a reference to the key manager and revoke all keys in the key ring
        var keyManager = services.GetService<IKeyManager>();
        Console.WriteLine("Revoking all keys in the key ring...");
        keyManager.RevokeAllKeys(DateTimeOffset.Now, "Sample revocation.");

        // try calling Protect - this should throw
        Console.WriteLine("Calling Unprotect...");
        try
        {
            var unprotectedPayload = protector.Unprotect(protectedData);
            Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
        }

        // try calling DangerousUnprotect
        Console.WriteLine("Calling DangerousUnprotect...");
        try
        {
            IPersistedDataProtector persistedProtector = protector as IPersistedDataProtector;
            if (persistedProtector == null)
            {
                throw new Exception("Can't call DangerousUnprotect.");
            }

            bool requiresMigration, wasRevoked;
            var unprotectedPayload = persistedProtector.DangerousUnprotect(
                protectedData: protectedData,
                ignoreRevocationErrors: true,
                requiresMigration: out requiresMigration,
                wasRevoked: out wasRevoked);
            Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
            Console.WriteLine($"Requires migration = {requiresMigration}, was revoked = {wasRevoked}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
```

```
  * Input: Hello!
  * Protected payload: CfDJ8LHIzUCX1ZVBn2BZ...
  * Round-tripped payload: Hello!
  * Revoking all keys in the key ring...
  * Calling Unprotect...
  * CryptographicException: The key {...} has been revoked.
  * Calling DangerousUnprotect...
  * Unprotected payload: Hello!
  * Requires migration = True, was revoked = True
  */
```

ASP.NET Core 中的数据保护配置

2018/6/21 • • Edit Online

访问这些主题，了解 ASP.NET Core 中的数据保护配置：

- 配置 ASP.NET Core 数据保护
  配置 ASP.NET Core 数据保护概述。
- 数据保护密钥管理和生存期
  数据保护密钥管理和生存期的相关信息。
- 数据保护的计算机范围策略支持
  为所有使用数据保护的应用设置默认计算机范围策略的详细信息。
- ASP.NET Core 中数据保护的非 DI 感知情境
  如何通过 DataProtectionProvider 具体类型，在不经 DI 特定代码流程的情况下使用数据保护。

配置 ASP.NET Core 数据保护

2018/11/14 • • Edit Online

初始化数据保护系统时，它适用默认设置基于操作环境。这些设置通常适用于一台计算机上运行的应用。有一名开发人员可能想要更改默认设置的情况：

- 应用程序分布在多台计算机。
- 出于合规性原因。

对于这些情况下，数据保护系统提供了丰富的配置 API。

> **WARNING**
>
> 类似于配置文件，数据保护密钥环应受到保护使用适当的权限。可以选择加密密钥的静态，但这不会阻止攻击者创建新的密钥。因此，应用程序的安全将受到影响。使用数据保护配置的存储位置应具有其访问仅限于应用本身会保护配置文件的方式类似。例如，如果您选择将密钥环存储在磁盘上，使用文件系统权限。请确保仅下的标识该 web 应用将在运行具有读取、写入和创建权限访问此目录。如果使用 Azure 表存储，只有 web 应用程序应能够读取、写入或创建新的条目中的表存储，等等。
>
> 扩展方法AddDataProtection返回IDataProtectionBuilder。 `IDataProtectionBuilder` 显示扩展方法，您可以链接在一起以配置数据保护选项。

ProtectKeysWithAzureKeyVault

若要将密钥存储在Azure 密钥保管库，配置与系统ProtectKeysWithAzureKeyVault中 `Startup` 类：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blobUriWithSasToken>"))
        .ProtectKeysWithAzureKeyVault("<keyIdentifier>", "<clientId>", "<clientSecret>");
}
```

设置密钥环存储位置 (例如， PersistKeysToAzureBlobStorage)。必须设置位置，因为在调用 `ProtectKeysWithAzureKeyVault` 实现IXmlEncryptor禁用自动数据保护设置，包括密钥环存储位置。前面的示例中使用 Azure Blob 存储来持久保存密钥环。有关详细信息，请参阅密钥存储提供程序：Azure 和 Redis。您还可以保留使用本地密钥环PersistKeysToFileSystem。

`keyIdentifier` 是用于密钥加密的密钥保管库密钥标识符 (例如， `https://contosokeyvault.vault.azure.net/keys/dataprotection/`)。

`ProtectKeysWithAzureKeyVault` 重载：

- ProtectKeysWithAzureKeyVault (IDataProtectionBuilder，KeyVaultClient，String)能够KeyVaultClient以启用数据保护系统，以使用密钥保管库。
- （IDataProtectionBuilder、字符串、字符串、X509Certificate2）ProtectKeysWithAzureKeyVault能够 `ClientId` 和X509Certificate以启用数据保护系统，以使用密钥保管库。
- （IDataProtectionBuilder、字符串、字符串、字符串）的 ProtectKeysWithAzureKeyVault能够 `ClientId` 和 `ClientSecret` 以启用数据保护系统，以使用密钥保管库。

PersistKeysToFileSystem

若要将密钥存储在 UNC 共享而不是在 *%LOCALAPPDATA%* 的默认位置，配置与系统PersistKeysToFileSystem:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"));
}
```

> **WARNING**
>
> 如果更改密钥持久性位置时，系统将不再会自动加密静态情况下，密钥由于它不知道是否 DPAPI 是一种合适的加密机制。

ProtectKeysWith*

您可以将系统配置为通过调用的任何保护静态密钥ProtectKeysWith* 配置 Api。请考虑下面的示例，它将密钥存储的 UNC 共享上并对这些密钥在使用特定的 X.509 证书的其余部分进行加密：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"))
        .ProtectKeysWithCertificate("thumbprint");
}
```

在 ASP.NET Core 2.1 或更高版本，可以提供X509Certificate2到ProtectKeysWithCertificate，例如从文件加载证书：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"))
        .ProtectKeysWithCertificate(
            new X509Certificate2("certificate.pfx", "password"));
}
```

请参阅密钥静态加密的更多示例和讨论的内置密钥加密机制。

UnprotectKeysWithAnyCertificate

在 ASP.NET Core 2.1 或更高版本，可以轮换证书和解密密钥使用一个数组静止X509Certificate2证书与UnprotectKeysWithAnyCertificate:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"))
        .ProtectKeysWithCertificate(
            new X509Certificate2("certificate.pfx", "password"));
        .UnprotectKeysWithAnyCertificate(
            new X509Certificate2("certificate_old_1.pfx", "password_1"),
            new X509Certificate2("certificate_old_2.pfx", "password_2"));
}
```

SetDefaultKeyLifetime

若要配置系统而不是默认值 90 天使用的密钥生存期为 14 天，请使用SetDefaultKeyLifetime:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
}
```

SetApplicationName

默认情况下，数据保护系统隔离应用程序不同，即使它们共享同一个物理密钥存储库。这可以防止应用程序了解彼此的受保护的负载。

若要共享受保护的应用之间的有效负载：

- 配置SetApplicationName具有相同值的每个应用中。
- 应用程序之间使用相同版本的数据保护 API 堆栈。执行任一应用的项目文件中的以下：
  - 引用通过相同的共享的框架版本Microsoft.AspNetCore.App 元包。
  - 引用相同数据保护包版本。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .SetApplicationName("shared app name");
}
```

DisableAutomaticKeyGeneration

您可能不希望应用程序以自动滚动更新的密钥（创建新的密钥），因为它们接近过期的方案。这一个示例可能是应用设置中的主/辅助关系，其中只有主应用程序负责密钥管理问题，然后辅助应用程序只需将密钥环的只读视图。可以将辅助应用程序配置为将视为只读密钥环，通过配置系统DisableAutomaticKeyGeneration:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .DisableAutomaticKeyGeneration();
}
```

每个应用程序隔离

数据保护系统提供了 ASP.NET Core 主机，但它自动隔离了从另一个，应用程序，即使这些应用在相同的工作进程帐户下运行，并且使用相同的主密钥材料。这是某种程度上类似于 IsolateApps 修饰符将来自 System.Web 的 **<machineKey >** 元素。

隔离机制的工作原理的考虑在本地计算机上的每个应用使用作为唯一的租户，因此IDataProtector取得 root 权限的任何给定的应用会自动包括用作鉴别器的应用程序 ID。应用程序的唯一 ID 来自两个位置之一：

1. 如果应用托管在 IIS 中，唯一的标识符是应用程序的配置路径。如果在 web 场环境中部署应用，此值应为稳定，假定 web 场中的所有计算机上的 IIS 环境配置方式都类似。

2. 如果应用程序不承载于 IIS 中，唯一的标识符是应用程序的物理路径。

唯一标识符设计可以经受住重置—的单个应用程序和计算机本身。

此隔离机制假设应用程序不是恶意。恶意应用程序始终会影响在相同的工作进程帐户下运行的任何其他应用。在共享宿主环境中应用是相互不受信任，托管提供商应采取措施来确保应用程序，包括分离应用的基础密钥存储库之间的 OS 级别隔离。

如果由 ASP.NET Core 主机未提供数据保护系统 (例如，如果通过其实例化 `DataProtectionProvider` 具体类型) 应用程序隔离在默认情况下处于禁用状态。只要它们提供相应由相同的密钥材料提供支持的所有应用程序时禁用应用程序隔离，可以共享负载目的。若要提供应用隔离在此环境中的，调用SetApplicationName方法的配置对象，并提供每个应用的唯一名称。

更改算法与 UseCryptographicAlgorithms

数据保护堆栈可以更改使用新生成的键的默认算法。若要执行此操作的最简单方法是调用UseCryptographicAlgorithms从配置回调：

```
services.AddDataProtection()
    .UseCryptographicAlgorithms(
        new AuthenticatedEncryptorConfiguration()
    {
        EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
        ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
    });
```

```
services.AddDataProtection()
    .UseCryptographicAlgorithms(
        new AuthenticatedEncryptionSettings()
    {
        EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
        ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
    });
```

默认的 EncryptionAlgorithm AES-256-CBC，且默认 ValidationAlgorithm 是 HMACSHA256。可以由系统管理员通过设置默认策略计算机范围的策略，但显式调用 `UseCryptographicAlgorithms` 重写默认策略。

调用 `UseCryptographicAlgorithms`，可指定预定义的内置列表中的所需的算法。无需担心有关算法的实现。在上述方案中，数据保护系统尝试使用 AES 的 CNG 实现，如果在 Windows 上运行。否则，它将回退到托管System.Security.Cryptography.Aes类。

您可以手动指定通过调用实现UseCustomCryptographicAlgorithms。

> **TIP**
>
> 更改算法不会影响密钥环中的现有密钥。它只影响新生成的键。

指定托管的自定义算法

若要指定托管的自定义算法，创建ManagedAuthenticatedEncryptorConfiguration指向实现类型的实例：

```
serviceCollection.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new ManagedAuthenticatedEncryptorConfiguration()
    {
        // A type that subclasses SymmetricAlgorithm
        EncryptionAlgorithmType = typeof(Aes),

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // A type that subclasses KeyedHashAlgorithm
        ValidationAlgorithmType = typeof(HMACSHA256)
    });
```

若要指定托管的自定义算法，创建ManagedAuthenticatedEncryptionSettings指向实现类型的实例：

```
serviceCollection.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new ManagedAuthenticatedEncryptionSettings()
    {
        // A type that subclasses SymmetricAlgorithm
        EncryptionAlgorithmType = typeof(Aes),

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // A type that subclasses KeyedHashAlgorithm
        ValidationAlgorithmType = typeof(HMACSHA256)
    });
```

通常*类型属性必须指向具体，可实例化（通过公共的无参数构造函数）的实现SymmetricAlgorithm并KeyedHashAlgorithm，但系统特殊的情况下等某些值 `typeof(Aes)` 为方便起见。

> **NOTE**
>
> SymmetricAlgorithm 必须 ≥ 128 位的密钥长度和 ≥ 64 位为单位的块大小，并且它必须支持使用 PKCS #7 填充 CBC 模式加密。KeyedHashAlgorithm 必须具有的摘要的大小 > = 128 位，并且它必须支持密钥长度等于哈希算法的摘要的长度。KeyedHashAlgorithm 并非是严格要求，为 HMAC。

指定自定义 Windows CNG 算法

若要指定自定义 Windows CNG 算法的 HMAC 验证使用 CBC 模式加密，创建CngCbcAuthenticatedEncryptorConfiguration实例，它包含的算法的信息：

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngCbcAuthenticatedEncryptorConfiguration()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // Passed to BCryptOpenAlgorithmProvider
        HashAlgorithm = "SHA256",
        HashAlgorithmProvider = null
    });
```

若要指定自定义 Windows CNG 算法的 HMAC 验证使用 CBC 模式加密，创建CngCbcAuthenticatedEncryptionSettings实例，它包含的算法的信息：

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngCbcAuthenticatedEncryptionSettings()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // Passed to BCryptOpenAlgorithmProvider
        HashAlgorithm = "SHA256",
        HashAlgorithmProvider = null
    });
```

NOTE

对称块加密算法的密钥长度必须 > = 128 位的块大小 > = 64 位，并且它必须支持使用 PKCS #7 填充 CBC 模式加密。哈希算法必须具有的摘要的大小 > = 128 位并且必须支持正在打开与 BCRYPT_ALG_处理_HMAC_标志的标志。* 提供程序属性可以设置为 null 以使用指定的算法的默认提供程序。请参阅BCryptOpenAlgorithmProvider文档了解详细信息。

若要指定自定义 Windows CNG 算法使用 Galois/计数器模式加密进行验证，创建CngGcmAuthenticatedEncryptorConfiguration实例，它包含的算法的信息：

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngGcmAuthenticatedEncryptorConfiguration()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256
    });
```

若要指定自定义 Windows CNG 算法使用 Galois/计数器模式加密进行验证，创建CngGcmAuthenticatedEncryptionSettings实例，它包含的算法的信息：

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngGcmAuthenticatedEncryptionSettings()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256
    });
```

> **NOTE**
> 对称块加密算法的密钥长度必须 > = 128 位的块大小为完全 128 位，并且它必须支持 GCM 加密。可以设置EncryptionAlgorithmProvider属性设置为 null 以使用指定的算法的默认提供程序。请参阅BCryptOpenAlgorithmProvider文档了解详细信息。

指定其他自定义算法

尽管不公开为第一类 API，数据保护系统是算法的可扩展，足以允许指定几乎任何类型。例如，就可以使包含硬件安全模块 (HSM) 中的所有密钥并提供加密和解密例程的核心的自定义实现。请参阅IAuthenticatedEncryptor中核心加密扩展性有关详细信息。

保存密钥托管在 Docker 容器中时

在中承载时Docker容器，应在维护密钥：

- 是一个 Docker 卷仍然存在超出容器的生存期，如共享的卷或主机装入的卷，一个文件夹。
- 外部提供程序，如Azure 密钥保管库或Redis。

其他资源

- 非 DI 感知的情境中 ASP.NET Core 的数据保护
- 在 ASP.NET Core 中支持的数据保护计算机范围的策略
- 在 Web 场中托管 ASP.NET Core

数据保护密钥管理和 ASP.NET Core 中的生存期

2018/7/18 • • Edit Online

作者：Rick Anderson

密钥管理

是应用程序尝试检测其操作环境和处理自己的密钥配置。

1. 如果应用托管在Azure 应用，密钥保存到 *%HOME%\ASP.NET\DataProtection-Keys*文件夹。此文件夹由网络存储提供支持，并跨托管应用的所有计算机同步。
   - 密钥不是静态保护的。
   - *DataProtection 密钥*文件夹提供密钥环在单个部署槽位中应用的所有实例。
   - 各部署槽位（例如过渡槽和生成槽）不共享密钥环。在交换之间部署槽，如交换到生产环境的过渡环境或使用 A / B 测试，使用数据保护的任何应用将无法解密使用之前槽位中的密钥环存储的数据。这会导致用户正在注销的应用程序使用标准的 ASP.NET Core cookie 身份验证，因为它使用数据保护来保护其 cookie。如果你需要独立于槽位的密钥环，则使用外部密钥环提供程序，例如 Azure Blob 存储、Azure 密钥保管库，SQL 存储区中，或 Redis 缓存。
2. 如果用户配置文件不可用，将密钥保存到 *%LOCALAPPDATA%\ASP.NET\DataProtection-Keys*文件夹。如果操作系统是 Windows，静态使用 DPAPI 加密密钥。
3. 如果应用托管在 IIS 中，密钥保留到 HKLM 注册表中特殊的注册表项的列仅对工作进程帐户。使用 DPAPI 对密钥静态加密。
4. 如果没有这些条件匹配，不保留当前进程之外的密钥。进程关闭时，生成所有密钥都都将丢失。

开发人员始终完全控制，并且可重写方式和存储密钥的位置。上面的前三个选项应提供合理的默认值对于大多数应用程序类似于如何在 ASP.NET **<machineKey >** 过去的工作自动生成例程。最终的回退选项是要求开发人员指定的唯一情形配置前期如果他们想要密钥持久性，但在极少数情况下才会出现以上后备机制。

在托管在 Docker 容器中，应是（共享的卷或仍然存在超出容器的生存期的主机装入的卷）的 Docker 卷的文件夹中保存密钥或在外部提供程序，如Azure 密钥保管库或Redis。如果应用无法访问共享的网络卷，外部提供程序也是可在 web 场方案中（请参阅PersistKeysToFileSystem有关详细信息）。

> **WARNING**
>
> 如果开发人员重写上面所述的规则和点处为特定的密钥存储库的数据保护系统，则禁用自动加密静态密钥。静态保护，可以通过重新启用配置。

密钥生存期

默认情况下，密钥的 90 天生存期。密钥过期后，应用将自动生成新的密钥，并将新的密钥设置为活动密钥。只要已停用的密钥保留在系统中，您的应用程序可以解密保护与他们的任何数据。请参阅密钥管理有关详细信息。

默认算法

使用的默认负载保护算法是 AES-256-CBC 保密性和 HMACSHA256 的真实性。512 位主密钥，每隔 90 天更改一次用于派生两个用于在每个有效负载的基础上这些算法的子键。请参阅子项派生有关详细信息。

其他资源

- ASP.NET Core 的密钥管理可扩展性
- 在 Web 场中托管 ASP.NET Core

在 ASP.NET Core 中支持的数据保护计算机范围的策略

2018/7/6 • • Edit Online

作者：Rick Anderson

Windows 上运行时，数据保护系统具有有限的支持设置默认计算机范围的所有应用程序使用 ASP.NET Core 数据保护策略。常规的思路是管理员可能想要更改默认设置，如算法或密钥的生存期，而无需手动更新每个应用程序，在计算机上。

> **WARNING**
>
> 系统管理员可以设置默认策略，但它们无法强制执行它。应用程序开发人员始终可以重写其中一个自己的选择的任何值。默认策略仅影响应用程序开发人员未指定显式设置值的位置。

设置默认策略

若要设置默认策略，管理员可以设置以下注册表项下系统注册表中的已知的值：

HKLM\SOFTWARE\Microsoft\DotNetPackages\Microsoft.AspNetCore.DataProtection

如果你是在 64 位操作系统上，并想要影响的 32 位应用程序的行为，请记住将上面的项的 Wow6432Node 等效项。

支持的值如下所示。

“值”	类型	描述
EncryptionType	字符串	指定应使用哪些算法来进行数据保护。值必须为 CNG CBC、CNG 的 GCM 或托管和中更详细地介绍了。
DefaultKeyLifetime	DWORD	指定新生成的键的生存期。此值以天为单位指定，并且必须是 > = 7。
KeyEscrowSinks	字符串	指定用于密钥证书的类型。值是以分号分隔的密钥托管接收器，其中在列表中的每个元素都实现的类型的程序集限定名称列表IKeyEscrowSink。

加密类型

如果 EncryptionType CNG CBC，系统配置为使用 CBC 模式对称的块密码来进行保密性和 HMAC 真实性有通过 Windows CNG 提供服务（请参阅指定自定义 Windows CNG 算法为更多详细信息）。支持以下的其他值，其中每个对应于 CngCbcAuthenticatedEncryptionSettings 类型上的属性。

“值”	类型	描述
EncryptionAlgorithm	字符串	理解的 CNG 对称的块密码算法的名称。此算法在 CBC 模式下打开。

“值”	类型	描述
EncryptionAlgorithmProvider	字符串	可以生成 EncryptionAlgorithm 的算法的 CNG 提供程序实现的名称。
EncryptionAlgorithmKeySize	DWORD	要派生的对称块加密算法的密钥长度（以位为单位）。
HashAlgorithm	字符串	理解的 CNG 哈希算法的名称。此算法在 HMAC 模式打开。
HashAlgorithmProvider	字符串	可以生成算法的 HashAlgorithm CNG 提供程序实现的名称。

如果 EncryptionType CNG GCM，系统配置为使用保密性和身份验证与服务提供的 Windows CNG Galois/计数器模式对称的块密码（请参阅指定自定义 Windows CNG 算法有关详细信息）。支持以下的其他值，其中每个对应于 CngGcmAuthenticatedEncryptionSettings 类型上的属性。

“值”	类型	描述
EncryptionAlgorithm	字符串	理解的 CNG 对称的块密码算法的名称。此算法将在 Galois/计数器模式中打开。
EncryptionAlgorithmProvider	字符串	可以生成 EncryptionAlgorithm 的算法的 CNG 提供程序实现的名称。
EncryptionAlgorithmKeySize	DWORD	要派生的对称块加密算法的密钥长度（以位为单位）。

如果管理 EncryptionType，系统配置为用于托管的 SymmetricAlgorithm 保密性和 KeyedHashAlgorithm 真实性（请参阅指定自定义管理算法有关详细信息）。支持以下的其他值，其中每个对应于 ManagedAuthenticatedEncryptionSettings 类型上的属性。

“值”	类型	描述
EncryptionAlgorithmType	字符串	实现 SymmetricAlgorithm 的类型的程序集限定名称。
EncryptionAlgorithmKeySize	DWORD	要派生的对称加密算法的密钥长度（以位为单位）。
ValidationAlgorithmType	字符串	实现 KeyedHashAlgorithm 的类型的程序集限定名称。

如果 EncryptionType 具有任何其他值非 null 或空时，数据保护系统启动时引发的异常。

WARNING

在配置涉及类型名称（EncryptionAlgorithmType、ValidationAlgorithmType、KeyEscrowSinks）的默认策略设置时，类型必须是可用于应用。这意味着，对于在桌面 CLR 上运行的应用程序，包含这些类型的程序集应会显示在全局程序集缓存（GAC）中。对于在.NET Core 上运行的 ASP.NET Core 应用程序，应安装包含这些类型的包。

非 DI 感知的情境中 ASP.NET Core 的数据保护

2018/7/6 • • Edit Online

作者：Rick Anderson

ASP.NET Core 数据保护系统通常是添加到服务容器并且供通过依赖关系注入 (DI) 的从属组件。但是，一些情况下，这不可行或所需，尤其是在将系统导入到现有应用程序。

若要支持这些方案中，Microsoft.AspNetCore.DataProtection.Extensions包提供具体类型，DataProtectionProvider，它提供一种简单的方法为使用数据保护而不依赖于 DI。`DataProtectionProvider`类型实现IDataProtectionProvider。构造`DataProtectionProvider`只需提供DirectoryInfo实例，以指示应存储提供程序的加密密钥的位置，如下面的代码示例中所示：

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder));

        var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
        Console.Write("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8FWbAn6...ch3hAPm1NJA
 * Unprotect returned: Hello world!
 *
 * Press any key...
*/
```

默认情况下，`DataProtectionProvider`具体的类型不加密原始密钥材料之前将其保存到文件系统。这是为了支持开

发人员将指向网络共享和数据保护系统无法自动推导适当静态密钥加密机制的方案。

此外，`DataProtectionProvider`具体的类型不隔离应用程序默认情况下。使用相同密钥的目录的所有应用程序可以共享负载长达其目的参数匹配。

DataProtectionProvider构造函数接受一个可选配置回调，可以用于调整的系统行为。下面的示例演示与显式调用还原隔离SetApplicationName。此示例还演示将系统配置为自动加密持久化的密钥使用 Windows DPAPI。如果目录指向 UNC 共享，可能想要在所有相关计算机间分发共享的证书还可将系统配置为使用基于证书的加密和调用ProtectKeysWithCertificate。

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder),
            configuration =>
            {
                configuration.SetApplicationName("my app name");
                configuration.ProtectKeysWithDpapi();
            });

        var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
        Console.Write("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}
```

TIP

实例`DataProtectionProvider`具体类型是创建开销很大。如果应用程序维护此类型的多个实例，并且如果他们正在使用相同的密钥存储目录，应用程序性能可能会降低。如果你使用`DataProtectionProvider`类型，我们建议你一次创建此类型，并重复使用尽可能多地它。`DataProtectionProvider`类型及其所有IDataProtector从它创建的实例是线程安全的多个调用方。

ASP.NET Core 数据保护扩展性 API

2018/6/21 • • Edit Online

- 核心加密扩展性
- 密钥管理扩展性
- 其他 API

在 ASP.NET Core 中的核心加密可扩展性

2018/7/6 • • Edit Online

> **WARNING**
> 实现的任意以下接口的类型应该是线程安全的多个调用方。

IAuthenticatedEncryptor

IAuthenticatedEncryptor接口是加密子系统的基本构建基块。通常没有一个 IAuthenticatedEncryptor 每个键，并且 IAuthenticatedEncryptor 实例包装所有加密的密钥材料和算法执行加密操作所需的信息。

顾名思义，类型是负责提供经过身份验证的加密和解密服务。以下两个 Api，它公开。

- 解密 (ArraySegment已加密文本，ArraySegment additionalAuthenticatedData): byte]
- 加密 (ArraySegment纯文本，ArraySegment additionalAuthenticatedData): byte]

加密方法返回的 blob，包括 enciphered 纯文本和身份验证标记。身份验证标记必须包含附加经过身份验证的数据 (AAD) 时，尽管 AAD 本身不需要从最终的负载。解密方法验证身份验证标记，并返回被解码的负载。应为 CryptographicException homogenized（除 ArgumentNullException 和类似）的所有失败。

> **NOTE**
> 实际上，IAuthenticatedEncryptor 实例本身不需要包含密钥材料。例如，实现可以将委托到 HSM 中的所有操作。

如何创建 IAuthenticatedEncryptor

- ASP.NET Core 2.x
- ASP.NET Core 1.x

IAuthenticatedEncryptorFactory接口表示一种类型，知道如何创建IAuthenticatedEncryptor实例。其 API 是，如下所示。

- CreateEncryptorInstance（IKey 密钥）: IAuthenticatedEncryptor

对于任何给定的 IKey 实例，其 CreateEncryptorInstance 方法创建的任何经过身份验证的加密程序应被视为等效，如下面的代码示例。

```
// we have an IAuthenticatedEncryptorFactory instance and an IKey instance
IAuthenticatedEncryptorFactory factory = ...;
IKey key = ...;

// get an encryptor instance and perform an authenticated encryption operation
ArraySegment<byte> plaintext = new ArraySegment<byte>(Encoding.UTF8.GetBytes("plaintext"));
ArraySegment<byte> aad = new ArraySegment<byte>(Encoding.UTF8.GetBytes("AAD"));
var encryptor1 = factory.CreateEncryptorInstance(key);
byte[] ciphertext = encryptor1.Encrypt(plaintext, aad);

// get another encryptor instance and perform an authenticated decryption operation
var encryptor2 = factory.CreateEncryptorInstance(key);
byte[] roundTripped = encryptor2.Decrypt(new ArraySegment<byte>(ciphertext), aad);


// the 'roundTripped' and 'plaintext' buffers should be equivalent
```

IAuthenticatedEncryptorDescriptor（ASP.NET Core 2.x 仅）

- ASP.NET Core 2.x
- ASP.NET Core 1.x

IAuthenticatedEncryptorDescriptor接口表示知道如何将自身导出到 XML 的类型。其 API 是，如下所示。

- ExportToXml(): XmlSerializedDescriptorInfo

XML 序列化

IAuthenticatedEncryptor 和 IAuthenticatedEncryptorDescriptor 的主要区别是描述符知道如何创建加密程序和其提供有效的参数。请考虑其实现依赖于 SymmetricAlgorithm 和 KeyedHashAlgorithm IAuthenticatedEncryptor。加密器的作业是使用这些类型，但它不一定知道这些类型的来源，因此它不能真正写出如何重新创建本身应用程序重启时的正确说明。描述符充当基于此较高级别。由于描述符知道如何创建加密程序实例（例如，它知道如何创建所需的算法），以便可以重新创建加密器实例，应用程序重置后，它可以序列化该知识库中的 XML 格式。

通过其 ExportToXml 例程，描述符可序列化。此例程返回 XmlSerializedDescriptorInfo 其中包含两个属性：XElement 表示形式的说明符和表示的类型IAuthenticatedEncryptorDescriptorDeserializer可以是用于使重新起用给定相应 XElement 此描述符。

序列化的描述符可能包含敏感信息，例如加密的密钥材料。数据保护系统具有已持久化到存储空间之前加密信息的内置支持。若要利用此功能，该描述符应标记包含敏感信息的属性名称"requiresEncryption"的元素(xmlns"http://schemas.asp.net/2015/03/dataProtection")，值"true"。

> **TIP**
> 没有用于将此属性设置的帮助器 API。调用 XElement.MarkAsRequiresEncryption() 位于命名空间 Microsoft.AspNetCore.DataProtection.AuthenticatedEncryption.ConfigurationModel 扩展方法。

也可以是其中的序列化的描述符不包含敏感信息的情况。再次考虑存储在 HSM 中的加密密钥的大小写。在序列化本身由于 HSM 就不会暴露纯文本形式材料时可描述符无法写出密钥材料。相反，描述符可以编写出密钥包装的密钥（如果 HSM 允许以这种方式导出）或密钥的 HSM 自己唯一标识符版本。

IAuthenticatedEncryptorDescriptorDeserializer

IAuthenticatedEncryptorDescriptorDeserializer接口表示知道如何反序列化从 XElement IAuthenticatedEncryptorDescriptor 实例的类型。它公开一种方法：

- ImportFromXml（XElement 元素）：IAuthenticatedEncryptorDescriptor

ImportFromXml 方法采用返回 XElement IAuthenticatedEncryptorDescriptor.ExportToXml和创建原始 IAuthenticatedEncryptorDescriptor 等效项。

实现 IAuthenticatedEncryptorDescriptorDeserializer 该类型应具有以下两个公共构造函数之一：

- .ctor(IServiceProvider)
- .ctor()

> **NOTE**
>
> IServiceProvider 传递到构造函数可能为 null。

顶级工厂

- ASP.NET Core 2.x
- ASP.NET Core 1.x

AlgorithmConfiguration类表示知道如何创建该类型IAuthenticatedEncryptorDescriptor实例。它公开一个单一的 API。

- CreateNewDescriptor(): IAuthenticatedEncryptorDescriptor

将 AlgorithmConfiguration 视为顶级工厂。配置用作模板。它所包装算法信息（例如，此配置生成使用 AES 128 GCM 主密钥描述符），但它尚不与特定键关联。

当调用 CreateNewDescriptor、仅用于此调用中，创建新的密钥材料，并且生成新 IAuthenticatedEncryptorDescriptor 其包装此密钥材料以及所需使用材料算法信息。无法在软件中创建（和保存在内存中）的密钥材料，它无法创建并保存在 HSM 中，依次类推。关键点是 CreateNewDescriptor 任何两个调用应永远不会创建等效 IAuthenticatedEncryptorDescriptor 实例。

AlgorithmConfiguration 类型用作密钥创建例程的入口点如自动密钥滚动。若要更改的所有将来的键的实现，请在 KeyManagementOptions 中设置 AuthenticatedEncryptorConfiguration 属性。

ASP.NET Core 的密钥管理可扩展性

2018/12/12 • • Edit Online

> **TIP**
> 读取密钥管理然后再阅读本部分中，因为它介绍了这些 Api 的基本概念的某些部分。

> **WARNING**
> 实现以下接口的任何类型应该是线程安全的多个调用方。

键

`IKey` 接口是加密系统中的键的基本表示形式。在抽象意义上，"加密密钥材料"字面意义上不在此处使用术语该键。一个键具有以下属性：

- 激活、创建和过期日期
- 吊销状态
- 密钥标识符 (GUID)

此外，`IKey` 公开 `CreateEncryptor` 方法用于创建IAuthenticatedEncryptor实例绑定到此密钥。

此外，`IKey` 公开 `CreateEncryptorInstance` 方法用于创建IAuthenticatedEncryptor实例绑定到此密钥。

> **NOTE**
> 没有 API 来检索从原始的加密材料 `IKey` 实例。

IKeyManager

`IKeyManager` 接口表示负责常规密钥存储、检索和操作的对象。它公开三个高级操作：

- 创建新的密钥，并将其保存到存储。
- 从存储中获取所有键。
- 撤消一个或多个密钥并保存到存储吊销信息。

> **WARNING**
> 编写 `IKeyManager` 是一个非常高级的任务，和大多数开发人员不应尝试。相反，大多数开发人员应充分利用所提供的功能XmlKeyManager类。

XmlKeyManager

`XmlKeyManager` 类型是内置具体实现 `IKeyManager` 。它提供了几个有用的功能，包括密钥托管和静态密钥加密。在此系统中的密钥都表示为 XML 元素 (具体而言， XElement)。

`XmlKeyManager` 取决于在完成其任务的过程中的其他几个组件：

- `AlgorithmConfiguration` 这决定了使用新的密钥的算法。
- `IXmlRepository` 其中将密钥保存在存储中的控件。
- `IXmlEncryptor` [可选]，它允许加密静态密钥。
- `IKeyEscrowSink` [可选] 提供密钥托管服务。

- `IXmlRepository` 其中将密钥保存在存储中的控件。
- `IXmlEncryptor` [可选]，它允许加密静态密钥。
- `IKeyEscrowSink` [可选] 提供密钥托管服务。

下面是高级的关系图，指示如何这些组件连接在一起内 `XmlKeyManager`。

密钥创建 / CreateNewKey

在实现中的 `CreateNewKey`，则 `AlgorithmConfiguration` 组件用于创建唯一 `IAuthenticatedEncryptorDescriptor`，然后序列化为 XML。如果存在密钥托管接收器，则原始（未加密）的 XML 提供到接收器进行长期存储。未加密的 XML 然后通过运行 `IXmlEncryptor`（如果需要）来生成加密的 XML 文档。此加密的文档保存到长期存储通过 `IXmlRepository`。（如果没有 `IXmlEncryptor` 是配置，未加密的文档保存在 `IXmlRepository`。）

密钥创建 / CreateNewKey

在实现中的 `CreateNewKey`，则 `IAuthenticatedEncryptorConfiguration` 组件用于创建唯一 `IAuthenticatedEncryptorDescriptor`，然后序列化为 XML。如果存在密钥托管接收器，则原始（未加密）的 XML 提供到接收器进行长期存储。未加密的 XML 然后通过运行 `IXmlEncryptor`（如果需要）来生成加密的 XML 文档。此加密的文档保存到长期存储通过 `IXmlRepository`。（如果没有 `IXmlEncryptor` 是配置，未加密的文档保存在 `IXmlRepository`。）

密钥检索 / GetAllKeys

在实现中的 `GetAllKeys`、XML 文档表示密钥和吊销读取从基础 `IXmlRepository`。如果这些文档进行加密，系统将自动解密机密。`XmlKeyManager` 创建适当 `IAuthenticatedEncryptorDescriptorDeserializer` 实例进行反序列化文档回 `IAuthenticatedEncryptorDescriptor` 实例，然后将封装在单个 `IKey` 实例。此集合的 `IKey` 实例返回给调用方。

可在特定的 XML 元素的详细信息密钥存储格式的文档。

IXmlRepository

`IXmlRepository` 接口表示可以持久保存到 XML 并从后备存储中检索 XML 的类型。它公开两个 Api:

- `GetAllElements` : `IReadOnlyCollection<XElement>`
- `StoreElement(XElement element, string friendlyName)`

实现 `IXmlRepository` 无需分析通过其传递的 XML。它们应视为不透明的 XML 文档，让较高的层担心如何生成和分析文档。

有四种内置的具体类型实现 `IXmlRepository`:

- FileSystemXmlRepository
- RegistryXmlRepository
- AzureStorage.AzureBlobXmlRepository
- RedisXmlRepository

- FileSystemXmlRepository
- RegistryXmlRepository
- AzureStorage.AzureBlobXmlRepository
- RedisXmlRepository

请参阅密钥存储提供程序文档有关详细信息。

注册的自定义 `IXmlRepository` 适合使用不同的后备存储（例如，Azure 表存储）时。

若要更改默认存储库应用程序范围，请注册自定义 `IXmlRepository` 实例：

```
services.Configure<KeyManagementOptions>(options => options.XmlRepository = new MyCustomXmlRepository());
```

```
services.AddSingleton<IXmlRepository>(new MyCustomXmlRepository());
```

IXmlEncryptor

`IXmlEncryptor` 接口表示可以加密纯文本 XML 元素的类型。它公开一个 API:

- Encrypt(XElement plaintextElement):EncryptedXmlInfo

如果一个序列化 `IAuthenticatedEncryptorDescriptor` 包含任何元素标记为"需要加密"，然后 `XmlKeyManager` 将通过已配置运行这些元素 `IXmlEncryptor` 的 `Encrypt` 方法，并将保存到加密的元素而不是为纯文本元素 `IXmlRepository` 。输出 `Encrypt` 方法是 `EncryptedXmlInfo` 对象。此对象是包含两个结果到加密的包装 `XElement` 表示的类型和 `IXmlDecryptor` 这可用于解密的相应元素。

有四种内置的具体类型实现 `IXmlEncryptor`：

- CertificateXmlEncryptor
- DpapiNGXmlEncryptor
- DpapiXmlEncryptor
- NullXmlEncryptor

请参阅rest 文档在密钥加密有关详细信息。

若要更改默认在静态密钥加密机制整个应用程序范围，请注册自定义 `IXmlEncryptor` 实例：

```
services.Configure<KeyManagementOptions>(options => options.XmlEncryptor = new MyCustomXmlEncryptor());
```

```
services.AddSingleton<IXmlEncryptor>(new MyCustomXmlEncryptor());
```

IXmlDecryptor

`IXmlDecryptor` 接口表示一种类型，知道如何解密 `XElement` 的已加密通过 `IXmlEncryptor` 。它公开一个 API:

- 解密 (XElement encryptedElement):XElement

`Decrypt` 方法撤消由执行的加密 `IXmlEncryptor.Encrypt` 。通常情况下，每个具体 `IXmlEncryptor` 实现都将具有相应的具体 `IXmlDecryptor` 实现。

类型实现 `IXmlDecryptor` 应具有以下两个公共构造函数之一：

- .ctor(IServiceProvider)
- .ctor()

> **NOTE**
> `IServiceProvider` 传递给构造函数可能为 null。

IKeyEscrowSink

`IKeyEscrowSink` 接口表示可执行托管的敏感信息的类型。回想一下，序列化的描述符可能包含敏感信息（如加密材料），这是什么导致了引入IXmlEncryptor键入第一个位置中。但是，意外的发生，并且密钥环可以删除或损坏。

托管接口提供了允许访问原始序列化的 XML，通过配置的任何转换前一个紧急应急IXmlEncryptor。该接口公开单个 API:

- 应用商店（Guid keyId、XElement 元素）

这就需要通过 `IKeyEscrowSink` 实现来处理提供的元素以安全的方式与业务策略保持一致。一个可能的实现可能是托管接收器使用的是已知的公司 X.509 证书的 XML 元素进行加密的证书的私钥已托管; `CertificateXmlEncryptor` 可以帮助解决这个问题类型。 `IKeyEscrowSink` 实现程序还负责适当地保留提供的元素。

默认情况下没有托管启用了机制，但服务器管理员可以全局配置此。它还可以配置以编程方式通过 `IDataProtectionBuilder.AddKeyEscrowSink` 方法，如下面的示例中所示。 `AddKeyEscrowSink` 方法重载镜像 `IServiceCollection.AddSingleton` 并 `IServiceCollection.AddInstance` 重载，为 `IKeyEscrowSink` 实例都应是单一实例。如果多个 `IKeyEscrowSink` 注册实例，每个将调用在密钥生成过程，因此密钥可同时托管多个机制。

没有 API 中读取数据从 `IKeyEscrowSink` 实例。这与托管机制的设计从理论上讲是一致：它可用于使密钥材料的受信任的颁发机构，并且由于应用程序本身不是受信任的颁发机构，它不应该有权访问其自身托管的材料。

下面的示例代码演示如何创建并注册 `IKeyEscrowSink` 密钥托管，以便只有"CONTOSODomain 管理员"的成员可以恢复它们。

> **NOTE**
> 若要运行此示例，必须是到已加入域的 Windows 8 / Windows Server 2012 计算机和域控制器必须是 Windows Server 2012 或更高版本。

```
using System;
using System.IO;
using System.Xml.Linq;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.AspNetCore.DataProtection.XmlEncryption;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi()
```

```
            .ProtectKeysWithDpapi()
            .AddKeyEscrowSink(sp => new MyKeyEscrowSink(sp));
        var services = serviceCollection.BuildServiceProvider();

        // get a reference to the key manager and force a new key to be generated
        Console.WriteLine("Generating new key...");
        var keyManager = services.GetService<IKeyManager>();
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddDays(7));
    }

    // A key escrow sink where keys are escrowed such that they
    // can be read by members of the CONTOSO\Domain Admins group.
    private class MyKeyEscrowSink : IKeyEscrowSink
    {
        private readonly IXmlEncryptor _escrowEncryptor;

        public MyKeyEscrowSink(IServiceProvider services)
        {
            // Assuming I'm on a machine that's a member of the CONTOSO
            // domain, I can use the Domain Admins SID to generate an
            // encrypted payload that only they can read. Sample SID from
            // https://technet.microsoft.com/library/cc778824(v=ws.10).aspx.
            _escrowEncryptor = new DpapiNGXmlEncryptor(
                "SID=S-1-5-21-1004336348-1177238915-682003330-512",
                DpapiNGProtectionDescriptorFlags.None,
                services);
        }

        public void Store(Guid keyId, XElement element)
        {
            // Encrypt the key element to the escrow encryptor.
            var encryptedXmlInfo = _escrowEncryptor.Encrypt(element);

            // A real implementation would save the escrowed key to a
            // write-only file share or some other stable storage, but
            // in this sample we'll just write it out to the console.
            Console.WriteLine($"Escrowing key {keyId}");
            Console.WriteLine(encryptedXmlInfo.EncryptedElement);

            // Note: We cannot read the escrowed key material ourselves.
            // We need to get a member of CONTOSO\Domain Admins to read
            // it for us in the event we need to recover it.
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Generating new key...
 * Escrowing key 38e74534-c1b8-4b43-aea1-79e856a822e5
 * <encryptedKey>
 *   <!-- This key is encrypted with Windows DPAPI-NG. -->
 *   <!-- Rule: SID=S-1-5-21-1004336348-1177238915-682003330-512 -->
 *   <value>MIIIfAYJKoZIhvcNAQcDoIIIbTCCCGkCAQ...T5rA4g==</value>
 * </encryptedKey>
 */
```

杂项 ASP.NET Core 数据保护 Api

2018/7/6 • • Edit Online

> **WARNING**
>
> 实现的任意以下接口的类型应该是线程安全的多个调用方。

ISecret

`ISecret` 接口表示一个机密的值，如加密的密钥材料。它包含以下 API 图面：

- `Length` : `int`
- `Dispose()` : `void`
- `WriteSecretIntoBuffer(ArraySegment<byte> buffer)` : `void`

`WriteSecretIntoBuffer` 方法填充原始机密值与提供的缓冲区。此 API 将作为参数的缓冲区的原因而不返回 `byte[]` 直接是，这使调用方机会固定限制对托管的垃圾回收器的机密暴露该缓冲区对象。

`Secret` 类型是的具体实现 `ISecret` 机密的值在进程中内存中的存储位置。在 Windows 平台上的机密的值加密通过CryptProtectMemory。

ASP.NET Core 数据保护实现

2018/6/21 • • Edit Online

- 已验证的加密详细信息
- 子项派生和已验证的加密
- 上下文标头
- 密钥管理
- 密钥存储提供程序
- 静态密钥加密
- 密钥永久性和设置
- 密钥存储格式
- 短数据保护提供程序

经过身份验证的加密在 ASP.NET Core 的详细信息

2018/7/6 • • Edit Online

对 IDataProtector.Protect 调用都是经过身份验证的加密操作。保护方法提供保密性和真实性，并且它会绑定到用于此特定 IDataProtector 实例派生其根 IDataProtectionProvider 目的链。

IDataProtector.Protect 采用 byte [] 纯文本参数，并且生成的字节 [] 受保护的负载，其格式为下面所述。（此外还有一个扩展方法重载以采用字符串的纯文本参数和返回的字符串受保护的负载。如果使用此 API 的受保护的负载格式仍将下面结构，但它会base64url 编码。）

受保护的负载格式

受保护的负载格式包含三个主要组件：

- 32 位神奇标头，它标识数据保护系统的版本。
- 128 位密钥 id 标识该密钥用于保护此特定的负载。
- 受保护的负载的剩余部分特定于加密程序按此键封装。在下面的示例密钥表示 AES 256 CBC + HMACSHA256 加密器和负载进一步细分，如下所示: * 一个 128 位密钥修饰符。* 一个 128 位初始化向量。* AES 256 CBC 输出 48 个字节。* 一个 HMACSHA256 身份验证标记。

示例受保护的负载如下所示。

```
09 F0 C9 F0 80 9C 81 0C 19 66 19 40 95 36 53 F8
AA FF EE 57 57 2F 40 4C 3F 7F CC 9D CC D9 32 3E
84 17 99 16 EC BA 1F 4A A1 18 45 1F 2D 13 7A 28
79 6B 86 9C F8 B7 84 F9 26 31 FC B1 86 0A F1 56
61 CF 14 58 D3 51 6F CF 36 50 85 82 08 2D 3F 73
5F B0 AD 9E 1A B2 AE 13 57 90 C8 F5 7C 95 4E 6A
8A AA 06 EF 43 CA 19 62 84 7C 11 B2 C8 71 9D AA
52 19 2E 5B 4C 1E 54 F0 55 BE 88 92 12 C1 4B 5E
52 C9 74 A0
```

从上面的第一个 32 位或 4 个字节的负载格式是标识版本 (09 F0 C9 F0) 的幻标头

下一步的 128 位或 16 个字节是密钥标识符 (80 9 C 81 0 C 19 66 19 40 95 36 53 F8 AA FF EE 57)

其余部分包含负载，并特定于所使用的格式。

> **WARNING**
>
> 给定的键对受保护的所有负载将都开始与同一个 20 字节（神奇的值，密钥 id）标头。管理员可以使用这一事实以进行诊断来估计生成负载时间。例如，上面的负载对应于键 {0c819c80-6619-4019-9536-53f8aaffee57}。如果检查密钥的存储库后找到验证此特定密钥激活日期为 2015年-01-01 和其到期日期为 2015年-03-01，然后是合乎情理假定负载（如果未被篡改）在该窗口中，给予生成或采用一个较小在任意一侧奶油因子。

子项派生和 ASP.NET Core 中的经过身份验证的加密

2018/7/6 • • Edit Online

密钥链中的大多数键将包含某种形式的平均信息量，并且将有算法信息指出"CBC 模式下加密 + HMAC 验证"或"GCM 加密 + 验证"。在这些情况下，我们将嵌入的平均信息量称为此密钥的主密钥材料（或公里）和我们执行派生将使用为实际的加密操作的密钥的密钥派生函数。

> **NOTE**
>
> 键是抽象的并自定义实现可能不会按如下所示的那样。如果密钥提供其自己的实现 `IAuthenticatedEncryptor` 而不是使用我们的内置工厂之一，本节中所述的机制不再适用。

其他经过身份验证的数据和子项派生

`IAuthenticatedEncryptor` 接口用作所有经过身份验证的加密操作的核心接口。其 `Encrypt` 方法采用两个缓冲区：纯文本和 additionalAuthenticatedData (AAD)。纯文本内容流保持不变调用 `IDataProtector.Protect`，但 AAD 由系统生成和包括三个组件：

1. 32 位神奇标头 09 F0 C9 F0 标识此版本的数据保护系统。
2. 128 位密钥 id。
3. 从创建目的链形成的可变长度字符串 `IDataProtector`，正在执行此操作。

由于 AAD 是为所有三个组件的元组唯一的我们可以使用它从密钥主机派生新密钥，而不是使用密钥主机本身中所有的我们的加密操作。每次调用 `IAuthenticatedEncryptor.Encrypt`，发生以下密钥派生过程：

(K_E，K_H) = SP800_108_CTR_HMACSHA512 (K_M，AAD，contextHeader | | keyModifier)

在这里，我们正在呼叫 NIST SP800 108 KDF 中计数器模式 (请参阅NIST SP800-108，sec。5.1) 使用以下参数：

- 密钥派生密钥 (KDK) = K_M
- PRF = HMACSHA512
- 标签 = additionalAuthenticatedData
- 上下文 = contextHeader | |keyModifier

上下文标头的可变长度和本质就是我们要为其派生 K_E 和 K_H 的算法的指纹。密钥修饰符是每次调用随机生成一个 128 位字符串 `Encrypt` 并提供以确保与绝大多数概率 KE 和 KH 是唯一对于此特定身份验证加密操作，即使所有其他输入给 KDF 为常数。

CBC 模式下加密 + HMAC 验证操作，|K_E |的密钥长度是对称的块密码，和 |K_H |是的 HMAC 例程的摘要大小。对 GCM 加密 + 验证操作 |K_H |= 0。

CBC 模式下加密 + HMAC 验证

一旦 K_E 生成通过上述机制中，我们生成的随机初始化向量，并运行该对称的块密码算法来加密纯文本。然后通过使用密钥 K_H 以生成 MAC 初始化 HMAC 例程将的初始化向量和已加密文本运行 下面以图形方式表示此过程和返回值。

输出: = keyModifier | |iv | |E_cbc（K_E，iv，数据）| |HMAC（K_H、iv | |E_cbc（K_E，iv，数据）)

> **NOTE**
>
> `IDataProtector.Protect` 实现将前面预置的幻标头和密钥 id到之前将其返回到调用方的输出。因为神奇的标头和密钥 id 是隐式的一部分AAD，因为密钥修饰符作为输入提供给 KDF，这意味着 MAC 进行身份验证的最终返回负载的每个单字节

Galois/计数器模式加密 + 验证

一旦 K_E 生成通过上述机制中，我们将生成随机的 96 位 nonce 和运行对称块加密算法，来加密纯文本，并生成的 128 位身份验证标记。

输出: = keyModifier | |nonce | |E_gcm（K_E，nonce，数据）| |authTag

> **NOTE**
>
> 即使 GCM 本机支持 AAD 的概念，我们有一个要仍输入 AAD 仅对原始 KDF 中，选择要传递到 GCM 其 AAD 参数为空字符串。这样做的原因是两个折叠。首先，以支持灵活性我们永远不希望将 K_M 直接用作加密密钥。此外，GCM 有一定非常严格的唯一性要求其输入。GCM 加密例程是曾调用对两个或更清晰的概率具有相同（键，nonce）的输入数据集对不能超过 2 ^32。如果我们修复 K_E 我们无法执行多个 2 ^32 加密操作之前运行与我们的 2 ^-32 限制。这可能看起来非常大量的操作，但在几天内，这些密钥的正常生存期内良好的高流量 web 服务器可以通过 40 亿个请求。保持合规的 2 ^ 我们继续使用 128 位密钥修饰符和 96 位 nonce，因而可真正扩展任何给定 K_M 的可用操作计数-32 概率限制。为简单起见的设计中，我们将共享 CBC 和 GCM 操作之间的 KDF 代码路径，并且由于 AAD 已被视为 KDF 中没有无需将其转发给 GCM 例程。

在 ASP.NET Core 的上下文标头

2018/7/6 • • Edit Online

背景和理论上

在数据保护系统中，"密钥"意味着可以提供的对象进行身份验证加密服务。每个键由唯一的 id (GUID) 和它所携带算法信息和 entropic 材料。其用途是，每个密钥执行唯一平均信息量，但系统不能强制实施的而我们还需要考虑的开发人员可能会通过修改密钥链中的现有密钥的算法信息手动更改密钥链。若要实现提供这种情况下我们安全要求数据保护系统具有的概念的加密灵活性，这样，安全地跨多个加密算法使用单个 entropic 值。

大多数系统支持的加密灵活性做到这一点包括有关负载之内算法某些标识信息。算法的 OID 通常是适合于此。但是，我们遇到的一个问题是有多种方法来指定相同的算法:"AES"(CNG) 托管 Aes、AesManaged、AesCryptoServiceProvider、AesCng 和（如果有特定的参数）的 RijndaelManaged 类都确实是相同首先，，我们将需要维护所有这些到正确的 OID 的映射。如果开发人员想要提供自定义算法（或甚至另一个实现的 AES ！），他们将需要告诉我们其 OID。此额外的注册步骤使系统配置特别棘手。

回顾一下，我们决定我们已从错误方向接近问题。OID 提示您的算法是什么，但我们不实际关心这。如果我们需要在两个不同的算法中安全地使用单个 entropic 值，它不是我们要知道算法的实际的所必要的。我们实际上关心是它们的行为方式。任何有效的对称块加密算法也是强伪随机排列 (PRP): 修复（键，链接模式，IV，纯文本）的输入和已加密文本输出将与急剧的概率为不同于任何其他对称的块密码提供的相同的输入的算法。同样，任何相当的加密哈希函数也是强伪函数 (PRF)，并且给定的一组固定的输入其输出将极其能不同于任何其他加密哈希函数。

我们使用强 PRPs 和 PRFs 的概念构建上下文标头。此上下文标头实质上是充当稳定指纹通过对于任何给定的操作，所用的算法，并提供所需的数据保护系统的加密灵活性。此标头是可重现和更高版本用作的一部分子项派生过程。有两个不同的方法来生成具体的操作模式的基础的算法取决于上下文标头。

CBC 模式下加密 + HMAC 身份验证

上下文标头由以下组件组成:

- [16 位]值 00 00，是一个标记，这意味着"CBC 加密 + HMAC 身份验证"。
- [32 位]对称块加密算法密钥长度（以字节为单位，big endian）。
- [32 位]块大小（以字节为单位，big endian）的对称块加密算法。
- [32 位]HMAC 算法的密钥长度（以字节为单位，big endian）。（当前密钥的大小始终与匹配的摘要大小。）
- [32 位]HMAC 算法（以字节为单位，big endian）摘要大小。
- EncCBC (K_E、IV，""), 它是提供空字符串输入对称的块密码算法的输出，并且其中 IV 是全为零的向量。下面描述了 K_E 的构造。
- MAC (K_H，""), 这是提供空字符串输入 HMAC 算法的输出。下面描述了 K_H 的构造。

理想情况下，我们无法 K_E 和 K_H 传递全为零的向量。但是，我们想要避免这种情况，其中基础算法检查存在的共享密钥的安全性执行任何操作（值得注意的是 DES 和 3DES）之前，它可以阻止使用如全为零向量的简单或可重复模式。

相反，我们使用 NIST SP800 108 KDF 中计数器模式 (请参阅NIST SP800-108，sec。5.1) 具有零长度键、标签和上下文和作为基础 PRF HMACSHA512。我们派生 |K_E |+ |K_H |字节的输出，然后将分解结果为 K_E 和 K_H 本身。数学上，这表示，如下所示。

(K_E || K_H) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")

示例：AES 192 CBC + HMACSHA256

作为示例，请考虑对称块加密算法是 AES 192 CBC，其中的验证算法是 HMACSHA256 这种情况。系统将生成使用以下步骤的上下文标头。

首先，让 (K_E | |K_H) = SP800_108_CTR (prf = HMACSHA512，键 =""，标签 =""，上下文 ="")，其中 |K_E |= 192 位和 |K_H |= 每个指定的算法的 256 位。这将导致 K_E = 5BB6...21DD 和 K_H = A04A...在下面的示例 00A9:

```
5B B6 C9 83 13 78 22 1D 8E 10 73 CA CF 65 8E B0
61 62 42 71 CB 83 21 DD A0 4A 05 00 5B AB C0 A2
49 6F A5 61 E3 E2 49 87 AA 63 55 CD 74 0A DA C4
B7 92 3D BF 59 90 00 A9
```

接下来，计算 Enc_CBC (K_E、IV，"") 对于给定 IV 的 AES-192-CBC = 0 * 和 K_E 按上面所述。

结果: = F474B1872B3B53E4721DE19C0841DB6F

接下来，计算 MAC (K_H，"") 为 HMACSHA256 给定 K_H 按上面所述。

结果: = D4791184B996092EE1202F36E8608FA8FBD98ABDFF5402F264B1D7211536220C

这将产生下面的完整上下文标头：

```
00 00 00 00 00 18 00 00 00 10 00 00 00 20 00 00
00 20 F4 74 B1 87 2B 3B 53 E4 72 1D E1 9C 08 41
DB 6F D4 79 11 84 B9 96 09 2E E1 20 2F 36 E8 60
8F A8 FB D9 8A BD FF 54 02 F2 64 B1 D7 21 15 36
22 0C
```

此上下文标头是经过身份验证的加密算法对（AES 192 CBC 加密 + HMACSHA256 验证）的指纹。组件，如所述上面是：

- 标记 (00 00)
- 块密码密钥长度 (00 00 00 18)
- 块密码块大小 (00 00 00 10)
- HMAC 密钥长度 (00 00 00 20)
- HMAC 摘要的大小 (00 00 00 20)
- 块密码 PRP 输出 (F4 74-DB 6F) 和
- HMAC PRF 输出 (D4 79-结束)。

> **NOTE**
>
> CBC 模式下加密 + HMAC 身份验证上下文标头生成而不考虑算法实现是否提供通过 Windows CNG 或托管 SymmetricAlgorithm 和 KeyedHashAlgorithm 类型相同的方式。这允许在不同操作系统上运行的应用程序可靠地生成相同的上下文标头，即使之间的 Os 不同算法的实现。（在实践中，KeyedHashAlgorithm 不必是正确的 HMAC。它可以是任何加密哈希算法类型。）

示例：3DES 192 CBC + HMACSHA1

首先，让 (K_E | |K_H) = SP800_108_CTR (prf = HMACSHA512，键 =""，标签 =""，上下文 ="")，其中 |K_E |= 192 位和 |K_H |= 每个指定的算法的 160 位。这将导致 K_E = A219...E2BB 和 K_H = DC4A...在下面的示例 B464:

```
A2 19 60 2F 83 A9 13 EA B0 61 3A 39 B8 A6 7E 22
61 D9 F8 6C 10 51 E2 BB DC 4A 00 D7 03 A2 48 3E
D1 F7 5A 34 EB 28 3E D7 D4 67 B4 64
```

接下来，计算 Enc_CBC (K_E、IV，"") 对于给定 IV 的 3DES-192-CBC = 0 * 和 K_E 按上面所述。

结果: = ABB100F81E53E10E

接下来，计算 MAC (K_H，"") 为 HMACSHA1 给定 K_H 按上面所述。

result := 76EB189B35CF03461DDF877CD9F4B1B4D63A7555

此示例的完整上下文标头，这是经过身份验证的指纹将产生如下所示的加密算法对（3DES 192 CBC 加密 + HMACSHA1 验证）：

```
00 00 00 00 00 18 00 00 00 08 00 00 00 14 00 00
00 14 AB B1 00 F8 1E 53 E1 0E 76 EB 18 9B 35 CF
03 46 1D DF 87 7C D9 F4 B1 B4 D6 3A 75 55
```

组件，如下所示细分：

- 标记 (00 00)
- 块密码密钥长度 (00 00 00 18)
- 块密码块大小 (00 00 00 08)
- HMAC 密钥长度 (00 00 00 14)
- HMAC 摘要的大小 (00 00 00 14)
- 块密码 PRP 输出 (AB B1-E1 0E) 和
- HMAC PRF 输出 (76 EB-结束)。

Galois/计数器模式加密 + 身份验证

上下文标头由以下组件组成：

- [16 位]值 00 01，是一个标记，这意味着"GCM 加密 + 身份验证"。
- [32 位]对称块加密算法密钥长度（以字节为单位，big endian）。
- [32 位]在经过身份验证的加密操作期间使用的 nonce 大小（以字节为单位，big endian）。(对于我们的系统，这固定的 nonce 大小 = 96 位。)
- [32 位]块大小（以字节为单位，big endian）的对称块加密算法。(为 GCM，这固定的块大小 = 128 位。)
- [32 位]身份验证标记大小（以字节为单位，big endian）已经过身份验证的加密函数生成。(对于我们的系统，这固定为标记大小 = 128 位。)
- [128 位]Enc_GCM 的标记 (K_E，nonce，"")，它是给定的空字符串输入对称的块密码算法的输出，并且其中 nonce 是 96 位全为零向量。

K_E 派生使用相同的机制，如下所示 CBC 加密 + HMAC 身份验证方案。但是，由于在此处 play 中没有任何 K_H，我们实质上具有 |K_H |= 0，且该算法会折叠到下面的表单。

K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")

示例：AES 256 GCM

首先，让 K_E = SP800_108_CTR (prf = HMACSHA512，键 =""，标签 =""，上下文 ="")，其中 |K_E |= 256 位。

K_E := 22BC6F1B171C08C4AE2F27444AF8FC8B3087A90006CAEA91FDCFB47C1B8733B8

接下来，计算 Enc_GCM 的身份验证标记 (K_E，nonce，"") 适用于给定 nonce 的 AES-256-GCM = 096 和 K_E 按上面所述。

结果: = E7DCCE66DF855A323A6BB7BD7A59BE45

这将产生下面的完整上下文标头：

```
00 01 00 00 00 20 00 00 00 0C 00 00 00 10 00 00
00 10 E7 DC CE 66 DF 85 5A 32 3A 6B B7 BD 7A 59
BE 45
```

组件，如下所示细分：

- 标记 (00 01)
- 块密码密钥长度 (00 00 00 20)
- nonce 的大小 (00 00 00 0 C)
- 块密码块大小 (00 00 00 10)
- 身份验证标记大小 (00 00 00 10) 和
- 从运行块密码的身份验证标记 (E7 DC-结束)。

ASP.NET Core 中的密钥管理

2018/7/24 • • Edit Online

数据保护系统自动管理主密钥用于保护和取消保护负载的生存期。每个键可以存在于四个阶段之一：

- 此键创建的密钥环中存在但尚未激活。密钥不应用于新的保护操作，直到有足够的时间已过该密钥没有机会将传播到所有计算机正在使用此密钥环。
- 活动-密钥密钥环中存在，应使用对所有新的保护操作。
- 过期的密钥已经运行了其自然的生存期内，不再应该用于新的保护操作。
- 撤消-密钥遭到破坏，并且不能对新的保护操作。

创建、活动和已过期的密钥可能都可用来取消保护传入的负载。默认情况下已吊销的密钥不可能用于取消保护有效负载，但应用程序开发人员可以重写此行为如有必要。

> **WARNING**
>
> 开发人员可能想要删除密钥从密钥环（例如，通过从文件系统中删除相应的文件）。此时，密钥保护的所有数据都都将永久无法被破解，并且都没有不紧急的重写一样也使用已吊销的密钥。正在删除密钥是真正破坏性的行为，并因此数据保护系统公开没有第一类 API 执行此操作。

默认密钥选择

当数据保护系统从后备存储库读取密钥环时，它将尝试查找从密钥环中的"默认"密钥。默认密钥用于保护的新操作。

常规的启发式方法是，数据保护系统会选择最新的激活日期与默认密钥的密钥。（没有小种附加因素，以允许服务器到服务器时钟偏差。）如果密钥已过期或吊销，并且如果应用程序禁用自动密钥生成，则将立即激活每个与生成新密钥密钥过期和滚动下面的策略。

数据保护系统的原因会立即生成新密钥，而不是回退到不同的密钥是生成新密钥应视为隐式在新密钥之前激活的所有密钥的过期时间。一般理念是新的密钥可能已使用不同的算法或静态加密机制比旧密钥配置和系统应首选而回退的当前配置。

没有异常。如果应用程序开发人员具有禁用自动密钥生成，则数据保护系统必须选择用作默认键。在此回退方案中，系统将使用首选项提供给有时间传播到群集中的其他计算机的密钥选择的最新的激活日期，非撤消键。回退系统最终可能因此选择的已过期的默认密钥。回退系统永远不会选择用作默认键已撤消的键，如果密钥环为空或已被吊销的每个键，然后系统将发出初始化时出错。

密钥的过期和滚动

创建密钥时，它自动具有提供的激活日期为 {now + 2 天} 和 {now + 90 天} 的到期日期。激活前 2 天的延迟提供关键的时间才能传遍整个系统。也就是说，它允许在后备存储指向其他应用程序以观察该密钥在其下一步的自动刷新周期，从而最大化，密钥环会成为的活动已传播到所有应用程序，可能需要使用它的可能性。

如果默认密钥将在 2 天内过期，并且密钥环尚不具有一个键，它将处于活动状态的默认密钥到期后，数据保护系统将自动保留密钥环的新键。此新的密钥都有的激活日期为 {默认密钥到期日期} 和 {now + 90 天} 的到期日期。这允许系统以自动服务不会中断定期轮转密钥。

可能会有情况，在其中立即激活与创建密钥。一个示例是当应用程序未运行的时间和过期密钥环中的所有键。在此情况下，该密钥有 {现在} 的不正常 2 天的激活延迟激活日期。

默认密钥生存期为 90 天，但这是可配置，如以下示例所示。

```
services.AddDataProtection()
       // use 14-day lifetime instead of 90-day lifetime
       .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
```

管理员还可以更改默认系统范围内，但显式调用 `SetDefaultKeyLifetime` 将覆盖任何系统范围的策略。默认密钥生存期不能为短于 7 天。

自动密钥环刷新

数据保护系统初始化时，它从基础存储库中读取密钥环，并将其缓存在内存中。此缓存，保护和取消保护操作继续进行而不会达到后备存储。大约每隔 24 小时或当前的默认密钥过期，具体取决于第一个，系统将自动检查更改的后备存储。

> **WARNING**
>
> 开发人员应很少（如果有）需要直接使用密钥管理 Api。数据保护系统将执行自动密钥管理，如上文所述。

数据保护系统公开某接口 `IKeyManager`，可以用来检查和更改密钥环。提供的实例的 DI 系统 `IDataProtectionProvider` 还可以提供的一个实例 `IKeyManager` 使用费。或者，你可以拉取 `IKeyManager` 直接从 `IServiceProvider` 如下面的示例中所示。

这会修改密钥环（显式创建的新项或执行吊销）的任何操作将使内存中缓存失效。下次调用 `Protect` 或 `Unprotect` 将导致重新读取密钥环，并重新创建缓存的数据保护系统。

下面的示例演示了如何使用 `IKeyManager` 界面检查和操作密钥环，包括撤销的现有密钥和手动生成新的密钥。

```
using System;
using System.IO;
using System.Threading;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // perform a protect operation to force the system to put at least
        // one key in the key ring
        services.GetDataProtector("Sample.KeyManager.v1").Protect("payload");
        Console.WriteLine("Performed a protect operation.");
        Thread.Sleep(2000);

        // get a reference to the key manager
        var keyManager = services.GetService<IKeyManager>();

        // list all keys in the key ring
        var allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked =
```

```
            Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked =
{key.IsRevoked}");
        }

        // revoke all keys in the key ring
        keyManager.RevokeAllKeys(DateTimeOffset.Now, reason: "Revocation reason here.");
        Console.WriteLine("Revoked all existing keys.");

        // add a new key to the key ring with immediate activation and a 1-month expiration
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddMonths(1));
        Console.WriteLine("Added a new key.");

        // list all keys in the key ring
        allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked =
{key.IsRevoked}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Performed a protect operation.
 * The key ring contains 1 key(s).
 * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = False
 * Revoked all existing keys.
 * Added a new key.
 * The key ring contains 2 key(s).
 * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = True
 * Key {2266fc40-e2fb-48c6-8ce2-5fde6b1493f7}: Created = 2015-03-18 22:20:51Z, IsRevoked = False
 */
```

密钥存储

数据保护系统具有启发式方法，它会尝试自动推断相应的密钥存储位置和静态加密机制。密钥持久性机制也是由应用开发人员可配置的。以下文档讨论这些机制的内置实现：

- 在 ASP.NET Core 中的密钥存储提供程序
- 在 ASP.NET Core 中存放的密钥加密

在 ASP.NET Core 中的密钥存储提供程序

2018/12/27 • • Edit Online

数据保护系统默认情况下使用的发现机制来确定应该保留加密密钥的位置。开发人员可以重写默认的发现机制，并手动指定的位置。

> **WARNING**
>
> 如果指定了显式密钥持久性位置，数据保护系统注销 rest 机制，在默认密钥加密，因此不再静态加密密钥。我们建议你此外指定一个显式的密钥加密机制对于生产部署。

文件系统

若要配置的文件基于系统的密钥存储库，请调用PersistKeysToFileSystem配置例程，如下所示。提供DirectoryInfo指向存储密钥的存储库：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys\"));
}
```

`DirectoryInfo` 可以指向本地计算机上的某个目录，也可以指向网络共享上的文件夹。如果指向本地计算机上的目录（和的方案是只在本地计算机上的应用需要使用此存储库的访问权限），请考虑使用Windows DPAPI（在 Windows）来加密静态密钥。否则，请考虑使用X.509 证书来加密静态密钥。

Azure 和 Redis

Microsoft.AspNetCore.DataProtection.AzureStorage并Microsoft.AspNetCore.DataProtection.StackExchangeRedis包允许将数据保护密钥存储在 Azure 存储或 Redis缓存。可以在 web 应用的多个实例之间共享密钥。应用可以跨多个服务器共享身份验证 cookie 或 CSRF 保护。

Microsoft.AspNetCore.DataProtection.AzureStorage并Microsoft.AspNetCore.DataProtection.Redis包允许将数据保护密钥存储在 Azure 存储或 Redis 缓存。可以在 web 应用的多个实例之间共享密钥。应用可以跨多个服务器共享身份验证 cookie 或 CSRF 保护。

若要配置 Azure Blob 存储提供程序，请调用之一PersistKeysToAzureBlobStorage重载：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blob URI including SAS token>"));
}
```

若要配置 Redis，调用之一PersistKeysToStackExchangeRedis重载：

```
public void ConfigureServices(IServiceCollection services)
{
    var redis = ConnectionMultiplexer.Connect("<URI>");
    services.AddDataProtection()
        .PersistKeysToStackExchangeRedis(redis, "DataProtection-Keys");
}
```

若要配置 Redis，调用之一PersistKeysToRedis重载：

```
public void ConfigureServices(IServiceCollection services)
{
    var redis = ConnectionMultiplexer.Connect("<URI>");
    services.AddDataProtection()
        .PersistKeysToRedis(redis, "DataProtection-Keys");
}
```

有关详细信息，请参阅下列主题：

- StackExchange.Redis ConnectionMultiplexer
- Azure Redis 缓存
- aspnet/DataProtection 示例

注册表

仅适用于 Windows 的部署。

有时应用程序可能没有到文件系统的写访问权限。请考虑应用正在作为虚拟服务帐户的方案（如*w3wp.exe*的应用程序池标识）。在这些情况下，管理员可以预配的服务帐户标识都可访问的注册表项。调用PersistKeysToRegistry扩展方法，如下所示。提供RegistryKey指向存储加密密钥的位置：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToRegistry(Registry.CurrentUser.OpenSubKey(@"SOFTWARE\Sample\keys"));
}
```

> **IMPORTANT**
>
> 我们建议使用Windows DPAPI来加密静态密钥。

Entity Framework Core

Microsoft.AspNetCore.DataProtection.EntityFrameworkCore包提供了用于存储到使用 Entity Framework Core 的数据库的数据保护密钥的机制。`Microsoft.AspNetCore.DataProtection.EntityFrameworkCore` NuGet 包必须将添加到项目文件中，它不属于Microsoft.AspNetCore.App 元包。

通过此包，可以在 web 应用的多个实例之间共享密钥。

若要配置 EF Core 提供程序，请调用 `PersistKeysToDbContext<TContext>` 方法：

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    // using Microsoft.AspNetCore.DataProtection;
    services.AddDataProtection()
        .PersistKeysToDbContext<MyKeysContext>();

    services.AddDefaultIdentity<IdentityUser>()
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

泛型参数 `TContext`，必须继承自DbContext并IDataProtectionKeyContext:

```
using Microsoft.AspNetCore.DataProtection.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using WebApp1.Data;

namespace WebApp1
{
    class MyKeysContext : DbContext, IDataProtectionKeyContext
    {
        // A recommended constructor overload when using EF Core
        // with dependency injection.
        public MyKeysContext(DbContextOptions<ApplicationDbContext> options)
            : base(options) { }

        // This maps to the table that stores keys.
        public DbSet<DataProtectionKey> DataProtectionKeys { get; set; }
    }
}
```

创建 `DataProtectionKeys` 表。

- Visual Studio
- .NET Core CLI

执行以下命令中的**程序包管理器控制台**(PMC) 窗口：

```
Add-Migration AddDataProtectionKeys -Context MyKeysContext
Update-Database -Context MyKeysContext
```

`MyKeysContext` 是 `DbContext` 在上面的代码示例中定义。如果您使用的 `DbContext` 使用不同的名称，替换为你 `DbContext` 名称 `MyKeysContext` 。

`DataProtectionKeys` 类/实体采用下表中所示的结构。

属性/字段	CLR 类型	SQL 类型
`Id`	`int`	`int` PK，不为 null

属性/字段	CLR 类型	SQL 类型
`FriendlyName`	`string`	`nvarchar(MAX)` null
`Xml`	`string`	`nvarchar(MAX)` null

自定义密钥存储库

如果不适当的内置机制，开发人员可以通过提供自定义指定自己的密钥持久性机制IXmlRepository。

在 ASP.NET Core 中存放的密钥加密

2018/7/24 • • Edit Online

数据保护系统默认情况下使用的发现机制来确定如何加密密钥应静态加密。开发人员可以重写的发现机制，并手动指定如何静态加密密钥。

> **WARNING**
>
> 如果指定的显式密钥暂留位置，数据保护系统注销 rest 机制在默认密钥加密。因此，无法再进行静态加密密钥。我们建议您指定一个显式的密钥加密机制对于生产部署。本主题中介绍的静态加密机制选项。

Azure Key Vault

若要将密钥存储在Azure 密钥保管库，配置与系统ProtectKeysWithAzureKeyVault中 `Startup` 类：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blobUriWithSasToken>"))
        .ProtectKeysWithAzureKeyVault("<keyIdentifier>", "<clientId>", "<clientSecret>");
}
```

有关详细信息，请参阅配置 ASP.NET Core 数据保护：ProtectKeysWithAzureKeyVault。

Windows DPAPI

仅适用于 Windows 的部署。

使用 Windows DPAPI 时，使用加密密钥材料CryptProtectData之前保存到存储。DPAPI 是永远不会读取当前计算机之外的数据的适当的加密机制 (不过可以备份到 Active Directory 这些密钥; 请参阅DPAPI 和漫游配置文件)。若要配置 DPAPI 密钥静态加密，请调用之一ProtectKeysWithDpapi扩展方法：

```
public void ConfigureServices(IServiceCollection services)
{
    // Only the local user account can decrypt the keys
    services.AddDataProtection()
        .ProtectKeysWithDpapi();
}
```

如果 `ProtectKeysWithDpapi` 称为不带任何参数，仅在当前 Windows 用户帐户，才能解密持久化的密钥环。您可以选择指定的计算机（而不仅仅是当前用户帐户）上的任何用户帐户能够解密密钥环：

```
public void ConfigureServices(IServiceCollection services)
{
    // All user accounts on the machine can decrypt the keys
    services.AddDataProtection()
        .ProtectKeysWithDpapi(protectToLocalMachine: true);
}
```

X.509 证书

如果应用程序分布在多台计算机，它可能是可以方便地在计算机间分发共享的 X.509 证书和配置托管的应用，以对密钥静态加密使用的证书：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .ProtectKeysWithCertificate("3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0");
}
```

由于.NET Framework 的限制，支持仅使用 CAPI 私钥的证书。请参阅以下内容的可能的解决方法，这些限制。

Windows DPAPI NG

此机制是仅在 Windows 8/Windows Server 2012 或更高版本上可用。

从 Windows 8 开始，Windows OS 支持 DPAPI NG（也称为 CNG DPAPI）。有关详细信息，请参阅有关 CNG DPAPI。

主体被编码为保护描述符规则。在下面的示例调用ProtectKeysWithDpapiNG，只有指定的 SID 的已加入域的用户可以解密密钥环：

```
public void ConfigureServices(IServiceCollection services)
{
    // Uses the descriptor rule "SID=S-1-5-21-..."
    services.AddDataProtection()
        .ProtectKeysWithDpapiNG("SID=S-1-5-21-...",
        flags: DpapiNGProtectionDescriptorFlags.None);
}
```

此外，还有的无参数重载`ProtectKeysWithDpapiNG`。使用此便捷方法来指定规则"SID = {CURRENT_ACCOUNT_SID}"，其中*CURRENT_ACCOUNT_SID*是当前 Windows 用户帐户的 SID:

```
public void ConfigureServices(IServiceCollection services)
{
    // Use the descriptor rule "SID={current account SID}"
    services.AddDataProtection()
        .ProtectKeysWithDpapiNG();
}
```

在此方案中，AD 域控制器负责分发 DPAPI NG 操作所用的加密密钥。目标用户可以解密加密的负载从任何加入域的计算机（前提是该进程的标识来运行）。

基于证书的加密使用 Windows DPAPI NG

如果应用程序运行在 Windows 8.1 / Windows Server 2012 R2 或更高版本，可以使用 Windows DPAPI NG 进行基于证书的加密。使用规则描述符字符串"证书 = HashId:THUMBPRINT"，其中*指纹*是十六进制编码的 SHA1 指纹的证书：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .ProtectKeysWithDpapiNG("CERTIFICATE=HashId:3BCE558E2...B5AEA2A9BD2575A0",
            flags: DpapiNGProtectionDescriptorFlags.None);
}
```

指向此存储库的任何应用程序必须运行 Windows 8.1 / Windows Server 2012 R2 或更高版本才能解开密钥。

自定义密钥加密

如果不适当的内置机制，开发人员可以通过提供自定义指定自己的密钥加密机制IXmlEncryptor。

密钥不可变性和 ASP.NET Core 中的密钥设置

2018/7/24 • • Edit Online

后一个对象被保存到后备存储，它的表示形式是永久固定的。新的数据可以添加到后备存储，但可以永远不会改变现有数据。此行为的主要目的是为了防止数据损坏。

此行为的一个后果是，一旦密钥将写入后备存储，它是不可变。其创建、激活和到期日期可以永远不会更改，但它可以通过使用吊销 `IKeyManager` 。此外，其基础的算法信息、主密钥密钥材料和 rest 属性静态加密也是不可变。

如果开发人员更改会影响密钥持久性的任何设置，这些更改不会起开始下一次生成一个密钥，通过显式调用之前 `IKeyManager.CreateNewKey` 或通过数据保护系统的自己自动密钥生成行为。影响密钥持久性的设置如下所示：

- 默认密钥生存期
- 在 rest 机制密钥加密
- 算法的信息包含在项

如果需要这些设置，以开始执行下一步的自动密钥滚动时间之前，请考虑进行显式调用 `IKeyManager.CreateNewKey` 强制创建新密钥。请记住，若要提供显式激活日期（{现在 + 2 天} 是一个很好的经验以留出时间进行传播的更改）的调用中的有效期范围之内。

> **TIP**
> 触摸存储库中的所有应用程序应指定相同的设置 `IDataProtectionBuilder` 扩展方法。否则，持久化密钥的属性将取决于特定调用密钥生成例程的应用程序。

在 ASP.NET Core 中的密钥存储格式

2018/8/21 • • Edit Online

对象的 XML 表示形式中的静态存储。密钥存储的默认目录为 %localappdata%\asp.net\dataprotection-keys\。

<密钥 > 元素

注册表项存在作为密钥存储库中的顶级对象。按照约定键具有文件名**密钥-{guid}.xml**，其中 {guid} 是密钥的 id。每个此类文件包含单个密钥。文件的格式如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<key id="80732141-ec8f-4b80-af9c-c4d2d1ff8901" version="1">
  <creationDate>2015-03-19T23:32:02.3949887Z</creationDate>
  <activationDate>2015-03-19T23:32:02.3839429Z</activationDate>
  <expirationDate>2015-06-17T23:32:02.3839429Z</expirationDate>
  <descriptor deserializerType="{deserializerType}">
    <descriptor>
      <encryption algorithm="AES_256_CBC" />
      <validation algorithm="HMACSHA256" />
      <enc:encryptedSecret decryptorType="{decryptorType}" xmlns:enc="...">
        <encryptedKey>
          <!-- This key is encrypted with Windows DPAPI. -->
          <value>AQAAANCM...8/zeP8lcwAg==</value>
        </encryptedKey>
      </enc:encryptedSecret>
    </descriptor>
  </descriptor>
</key>
```

<密钥 > 元素包含以下属性和子元素：

- 密钥 id。此值将被处理为授权，;文件名是只需以方便人们阅读过去。
- 版本的<密钥 > 元素中，当前固定为 1。
- 密钥的创建、激活和到期日期。
- 一个<描述符 > 元素，它包含有关该注册表项中包含的已验证的加密实现的信息。

在上述示例中，密钥的 id 是 {80732141-ec8f-4b80-af9c-c4d2d1ff8901}，它已创建并激活 2015 年 3 月 19 日，并且有 90 天的生存期。（有时会激活日期可能略微之前如此示例所示的创建日期。这是因为中的 Api 如何工作和不会造成损害在实践中指出。）

<描述符 > 元素

外部<描述符 > 元素包含属性 deserializerType，这是实现 IAuthenticatedEncryptorDescriptorDeserializer 的类型的程序集限定名称。此类型负责读取内部<描述符 > 元素和用于分析中包含的信息。

特定的格式<描述符 > 元素取决于经过身份验证的加密程序实现封装的密钥，以及每个反序列化程序类型为此需要稍微不同的格式。一般情况下，不过，此元素将包含算法的信息（名称、类型、Oid，或类似）和机密密钥材料。在上述示例中，该描述符指定此密钥包装 AES-256-CBC 加密 + HMACSHA256 验证。

<EncryptedSecret > 元素

<EncryptedSecret> 元素，其中包含的密钥材料的加密的形式可能会显示如果启用加密的机密的静态。该属性

`decryptorType` 实现的类型的程序集限定名称IXmlDecryptor。此类型负责读取内部 **<encryptedKey>** 元素，并对其恢复原始的纯文本进行解密。

如同<描述符 >，特定的格式元素取决于正在使用的静态加密机制。在上述示例中，每个注释使用 Windows DPAPI 对主密钥进行加密。

<吊销 > 元素

吊销作为密钥存储库中的顶级对象存在。根据约定吊销使用文件名**吊销-{timestamp}.xml**（适用于特定日期前撤消所有密钥）或**吊销-{guid}.xml**（适用于都撤消特定键）。每个文件将包含一个<吊销 > 元素。

对于各个键的吊销情况，该文件的内容将如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T22:45:30.2616742Z</revocationDate>
  <key id="eb4fc299-8808-409d-8a34-23fc83d026c9" />
  <reason>human-readable reason</reason>
</revocation>
```

在这种情况下，撤消指定的键。如果密钥 id 为"*"，但是，如以下示例中，会撤销其创建日期是在指定的吊销日期之前的所有密钥。

```
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T15:45:45.7366491-07:00</revocationDate>
  <!-- All keys created before the revocation date are revoked. -->
  <key id="*" />
  <reason>human-readable reason</reason>
</revocation>
```

<原因 > 元素永远不会读取系统。它是只需很方便地存储吊销一个用户可读的理由。

ASP.NET Core 中的临时数据保护提供程序

2018/7/6 • • Edit Online

有的方案：应用程序需要的 throwaway `IDataProtectionProvider`。例如，开发人员可能只需在一次性的控制台应用程序中，进行试验或应用程序本身是暂时性（已编写脚本或单元测试项目）。若要支持这些方案Microsoft.AspNetCore.DataProtection包包括一种类型`EphemeralDataProtectionProvider`。此类型提供的基本实现`IDataProtectionProvider`其密钥的存储库保留仅在内存中和不写出到任何后备存储。

每个实例`EphemeralDataProtectionProvider`使用其自身唯一的主密钥。因此，如果`IDataProtector`处`EphemeralDataProtectionProvider`生成受保护的负载，该负载仅受等效`IDataProtector`(给定相同目的链) 位于同一个取得 root 权限`EphemeralDataProtectionProvider`实例。

下面的示例演示如何实例化`EphemeralDataProtectionProvider`并使用它来保护和取消保护数据。

```
using System;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        const string purpose = "Ephemeral.App.v1";

        // create an ephemeral provider and demonstrate that it can round-trip a payload
        var provider = new EphemeralDataProtectionProvider();
        var protector = provider.CreateProtector(purpose);
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        // if I create a new ephemeral provider, it won't be able to unprotect existing
        // payloads, even if I specify the same purpose
        provider = new EphemeralDataProtectionProvider();
        protector = provider.CreateProtector(purpose);
        unprotectedPayload = protector.Unprotect(protectedPayload); // THROWS
    }
}

/*
* SAMPLE OUTPUT
*
* Enter input: Hello!
* Protect returned: CfDJ8AAAAAAAAAAAAAAAAAAAAA...uGoxWLjGKtm1SkNACQ
* Unprotect returned: Hello!
* << throws CryptographicException >>
*/
```

ASP.NET Core 中的兼容性

2018/6/21 • • Edit Online

- 在 ASP.NET Core 中替换 ASP.NET <machineKey>

将在 ASP.NET Core ASP.NET machineKey

2018/6/22 • • Edit Online

实现 `<machineKey>` 在 ASP.NET 中的元素是可替换。这允许对 ASP.NET 加密例程的大多数调用，以通过替换数据保护机制，包括新的数据保护系统路由。

包安装

> **NOTE**
>
> 新的数据保护系统只能安装到现有的 ASP.NET 应用程序面向.NET 4.5.1 或更高版本。安装将失败，如果应用程序面向.NET 4.5 或降低。

若要安装新的数据保护系统到现有的 ASP.NET 4.5.1+ 项目之后，安装包 Microsoft.AspNetCore.DataProtection.SystemWeb。这将实例化数据保护系统使用默认配置设置。

当你安装此程序包时，它将插入行到*Web.config*这是告诉 ASP.NET，以将其用于最加密操作，包括窗体身份验证、视图状态和调用MachineKey.Protect。插入的行，如下所示读取。

```
<machineKey compatibilityMode="Framework45" dataProtectorType="..." />
```

> **TIP**
>
> 你可以判断是否处于活动状态通过检查类似的字段访问新的数据保护系统 `__VIEWSTATE`，这应该以开头"CfDJ8"以下示例所示。"CfDJ8"是标识受数据保护系统的负载的幻"09 F0 C9 F0"标头的 base64 表示。

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="CfDJ8AWPr2EQPTBGs3L2GCZOpk..." />
```

包配置

数据保护系统是具有默认零安装程序配置实例化。但是，由于默认情况下会将密钥保存到本地文件系统，这不起作用的应用程序的场中部署的状态。若要解决此问题，你可以通过创建一种类型的子类 DataProtectionStartup 提供配置，并重写其 ConfigureServices 方法。

下面是一个示例配置将在其中保留密钥，又如何要加密对静止的自定义的数据保护启动类型。它还通过提供其自己的应用程序名称来重写默认应用程序隔离策略。

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.SystemWeb;
using Microsoft.Extensions.DependencyInjection;

namespace DataProtectionDemo
{
    public class MyDataProtectionStartup : DataProtectionStartup
    {
        public override void ConfigureServices(IServiceCollection services)
        {
            services.AddDataProtection()
                .SetApplicationName("my-app")
                .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\myapp-keys\"))
                .ProtectKeysWithCertificate("thumbprint");
        }
    }
}
```

TIP

你还可以使用 `<machineKey applicationName="my-app" ... />` 代替 SetApplicationName 显式调用。这是不会强制执行开发人员可以创建 DataProtectionStartup 派生类型，如果他们想要配置所有已设置应用程序名称的方便机制。

若要启用此自定义配置，请返回到 Web.config，并查找 `<appSettings>` 包安装到的配置文件添加的元素。它将类似以下的标记：

```
<appSettings>
  <!--
  If you want to customize the behavior of the ASP.NET Core Data Protection stack, set the
  "aspnet:dataProtectionStartupType" switch below to be the fully-qualified name of a
  type which subclasses Microsoft.AspNetCore.DataProtection.SystemWeb.DataProtectionStartup.
  -->
  <add key="aspnet:dataProtectionStartupType" value="" />
</appSettings>
```

填写的空白值与你刚刚创建的 DataProtectionStartup 派生的类型的程序集限定名称。如果应用程序的名称是 DataProtectionDemo，这将如下所示下面。

```
<add key="aspnet:dataProtectionStartupType"
     value="DataProtectionDemo.MyDataProtectionStartup, DataProtectionDemo" />
```

最新配置数据保护系统现已在应用程序中使用。

安全存储中 ASP.NET Core 中开发的应用程序机密

2018/12/12 • • Edit Online

通过Rick Anderson，Daniel Roth，和Scott Addie

查看或下载示例代码（如何下载）

本文档介绍用于存储和检索敏感数据的 ASP.NET Core 应用程序开发过程的技术。永远不要将密码或其他敏感数据存储在源代码中。不应使用生产机密进行开发或测试。可使用 Azure Key Vault 配置提供程序存储和保护 Azure 测试和生产机密。

环境变量

使用环境变量以避免在代码中或在本地配置文件中的应用机密的存储。环境变量重写所有以前指定的配置源的配置的值。

通过调用配置的环境变量值的读取AddEnvironmentVariables中 `Startup` 构造函数：

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json",
                     optional: false,
                     reloadOnChange: true)
        .AddEnvironmentVariables();

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets<Startup>();
    }

    Configuration = builder.Build();
}
```

请考虑在其中一个 ASP.NET Core web 应用单个用户帐户已启用安全性。默认数据库连接字符串包含在项目的*appsettings.json*文件具有键 `DefaultConnection` 。默认连接字符串是 localdb，这在用户模式下运行，不需要密码。应用程序在部署期间，`DefaultConnection` 使用环境变量的值可以重写密钥值。环境变量可以存储敏感凭据与完整的连接字符串。

> **WARNING**
>
> 环境变量通常存储在普通的未加密的文本。如果计算机或进程受到攻击，可以由不受信任方访问环境变量。可能需要更多措施，防止用户机密泄露。

机密管理器

密码管理器工具在 ASP.NET Core 项目的开发过程中存储敏感数据。在此上下文中，一种敏感数据是应用程序密码。应用程序机密存储在项目树不同的位置。与特定项目关联或在多个项目之间共享的应用程序机密。应用机密不会签入源代码管理。

> **WARNING**
> 机密管理器工具不会加密存储的机密，不应被视为受信任存储区。它是仅限开发目的。在用户配置文件目录中的 JSON 配置文件中存储的键和值。

机密管理器工具的工作原理

机密管理器工具抽象化实现详细信息，例如位置和方式存储的值。如果不知道这些实现的详细信息，可以使用该工具。值存储在本地计算机上 JSON 配置文件中的系统保护的用户配置文件文件夹中：

- Windows
- macOS
- Linux

文件系统路径：

`%APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json`

在前面文件路径中，替换 `<user_secrets_id>` 与 `UserSecretsId` 中指定值 *.csproj*文件。

不编写代码，取决于使用机密管理器工具中保存的数据的格式的位置。这些实现细节可能会更改。例如，机密的值不会加密，但可能在将来。

安装机密管理器工具

机密管理器工具是可与.NET Core CLI，在.NET Core SDK 2.1.300 捆绑或更高版本。有关.NET Core SDK 2.1.300 之前的版本中，工具安装是必需的。

> **TIP**
> 运行 `dotnet --version` 从命令行界面中，若要查看已安装的.NET Core SDK 版本号。

如果正在使用的.NET Core SDK 包括的工具，将显示警告：

```
The tool 'Microsoft.Extensions.SecretManager.Tools' is now included in the .NET Core SDK.
Information on resolving this warning is available at (https://aka.ms/dotnetclitools-in-box).
```

安装Microsoft.Extensions.SecretManager.Tools ASP.NET Core 项目中的 NuGet 包。例如：

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
    <UserSecretsId>1242d6d6-9df3-4031-b031-d9b27d13c25a</UserSecretsId>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore"
                      Version="1.1.6" />
    <PackageReference Include="Microsoft.Extensions.Configuration.UserSecrets"
                      Version="1.1.2" />
    <PackageReference Include="System.Data.SqlClient"
                      Version="4.5.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools"
                            Version="1.0.1" />
  </ItemGroup>
</Project>
```

在验证工具的安装命令行界面中执行以下命令：

```
dotnet user-secrets -h
```

机密管理器工具会显示示例用法、选项和命令帮助：

```
Usage: dotnet user-secrets [options] [command]

Options:
  -?|-h|--help                       Show help information
  --version                          Show version information
  -v|--verbose                       Show verbose output
  -p|--project <PROJECT>             Path to project. Defaults to searching the current directory.
  -c|--configuration <CONFIGURATION> The project configuration to use. Defaults to 'Debug'.
  --id                               The user secret ID to use.

Commands:
  clear   Deletes all the application secrets
  list    Lists all the application secrets
  remove  Removes the specified user secret
  set     Sets the user secret to the specified value

Use "dotnet user-secrets [command] --help" for more information about a command.
```

> **NOTE**
>
> 您必须位于与相同的目录 *.csproj*文件中定义的工具在运行 *.csproj*文件的 `DotNetCliToolReference` 元素。

设置的机密

机密管理器工具对存储在用户配置文件中的特定于项目的配置设置进行操作。若要使用用户机密，定义 `UserSecretsId` 中的元素 `PropertyGroup` 的 *.csproj*文件。值 `UserSecretsId` 是任意的但仅适用于该项目。开发人员通常会生成的 GUID `UserSecretsId`。

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
  <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>
</PropertyGroup>
```

```
<PropertyGroup>
  <TargetFramework>netcoreapp1.1</TargetFramework>
  <UserSecretsId>1242d6d6-9df3-4031-b031-d9b27d13c25a</UserSecretsId>
</PropertyGroup>
```

TIP

在 Visual Studio 中，右键单击解决方案资源管理器中的项目并选择**管理用户机密**从上下文菜单。添加了此笔势 `UserSecretsId` 元素中，为填充 guid *.csproj*文件。Visual Studio 将打开*secrets.json*在文本编辑器中的文件。内容替换为*secrets.json*与要存储的键 / 值对。例如：

```
{
  "Movies": {
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
    "ServiceApiKey": "12345"
  }
}
```

JSON 结构平展后通过修改 `dotnet user-secrets remove` 或 `dotnet user-secrets set` 。例如，运行 `dotnet user-secrets remove "Movies:ConnectionString"` 折叠 `Movies` 对象文字。修改后的文件如下所示：

```
{
  "Movies:ServiceApiKey": "12345"
}
```

定义由一个键和其值组成的应用程序密码。密钥是与项目相关联 `UserSecretsId` 值。例如，从在其中的目录运行以下命令 *.csproj*文件是否存在：

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345"
```

在前面的示例中，冒号表示 `Movies` 是一个对象文字与 `ServiceApiKey` 属性。

机密管理器工具可以过使用从其他目录。使用 `--project` 选项可提供文件系统路径，在 *.csproj*文件存在。例如：

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345" --project "C:\apps\WebApp1\src\WebApp1"
```

设置多个机密

可以通过管道传递到 JSON 设置机密一批 `set` 命令。在以下示例中，*input.json*文件的内容通过管道传递给 `set` 命令。

- Windows
- macOS
- Linux

打开命令外壳中，并执行以下命令：

```
type .\input.json | dotnet user-secrets set
```

访问机密

ASP.NET Core 配置 API提供对机密 Manager 机密的访问。如果项目面向.NET Framework，安装Microsoft.Extensions.Configuration.UserSecrets NuGet 包。

在 ASP.NET Core 2.0 或更高版本，用户机密配置源时，自动添加在开发模式下项目调用CreateDefaultBuilder初始化带有预配置的默认值的主机的新实例。`CreateDefaultBuilder` 调用AddUserSecrets时EnvironmentName是开发:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
```

当 `CreateDefaultBuilder` 不是在主机构造过程中调用，添加用户机密配置源通过调用AddUserSecrets中 `Startup` 构造函数：

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json",
                     optional: false,
                     reloadOnChange: true)
        .AddEnvironmentVariables();

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets<Startup>();
    }

    Configuration = builder.Build();
}
```

ASP.NET Core 配置 API提供对机密 Manager 机密的访问。安装Microsoft.Extensions.Configuration.UserSecrets NuGet 包。

添加用户机密配置源通过调用AddUserSecrets中 `Startup` 构造函数：

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json",
                     optional: false,
                     reloadOnChange: true)
        .AddEnvironmentVariables();

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets<Startup>();
    }

    Configuration = builder.Build();
}
```

可以通过检索用户机密 `Configuration` API:

```
public class Startup
{
    private string _moviesApiKey = null;

    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        _moviesApiKey = Configuration["Movies:ServiceApiKey"];
    }

    public void Configure(IApplicationBuilder app)
    {
        var result = string.IsNullOrEmpty(_moviesApiKey) ? "Null" : "Not Null";
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync($"Secret is {result}");
        });
    }
}
```

```
public class Startup
{
    private string _moviesApiKey = null;

    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(env.ContentRootPath)
            .AddJsonFile("appsettings.json",
                         optional: false,
                         reloadOnChange: true)
            .AddEnvironmentVariables();

        if (env.IsDevelopment())
        {
            builder.AddUserSecrets<Startup>();
        }

        Configuration = builder.Build();
    }

    public IConfigurationRoot Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        _moviesApiKey = Configuration["Movies:ServiceApiKey"];
    }

    public void Configure(IApplicationBuilder app)
    {
        var result = string.IsNullOrEmpty(_moviesApiKey) ? "Null" : "Not Null";
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync($"Secret is {result}");
        });
    }
}
```

映射到 POCO 的机密

将整个对象文字映射到 POCO（具有属性的简单.NET 类）可用于聚合相关的属性。

假定应用程序的*secrets.json*文件包含以下两个密码：

```
{
  "Movies": {
    "ServiceApiKey": "12345",
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

若要映射到 POCO，前面的机密，使用 `Configuration` API 的对象 graph 绑定功能。以下代码将绑定到自定义 `MovieSettings` POCO 和访问 `ServiceApiKey` 属性值：

```
var moviesConfig = Configuration.GetSection("Movies")
                                .Get<MovieSettings>();
_moviesApiKey = moviesConfig.ServiceApiKey;
```

```
var moviesConfig = new MovieSettings();
Configuration.GetSection("Movies").Bind(moviesConfig);
_moviesApiKey = moviesConfig.ServiceApiKey;
```

`Movies:ConnectionString` 并 `Movies:ServiceApiKey` 机密映射到相应的属性中 `MovieSettings`：

```
public class MovieSettings
{
    public string ConnectionString { get; set; }

    public string ServiceApiKey { get; set; }
}
```

包含机密的字符串替换

以纯文本形式存储密码是不安全。例如，数据库连接字符串存储在*appsettings.json*可能包括指定用户的密码：

```
{
  "ConnectionStrings": {
    "Movies": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;User
Id=johndoe;Password=pass123;MultipleActiveResultSets=true"
  }
}
```

更安全的方法是将密码存储为机密。例如：

```
dotnet user-secrets set "DbPassword" "pass123"
```

删除 `Password` 中的连接字符串中的键 / 值对*appsettings.json*。例如：

```
{
  "ConnectionStrings": {
    "Movies": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;User
Id=johndoe;MultipleActiveResultSets=true"
  }
}
```

在上设置的机密的值SqlConnectionStringBuilder对象的密码属性来完成的连接字符串：

```
public class Startup
{
    private string _connection = null;

    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        var builder = new SqlConnectionStringBuilder(
            Configuration.GetConnectionString("Movies"));
        builder.Password = Configuration["DbPassword"];
        _connection = builder.ConnectionString;
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync($"DB Connection: {_connection}");
        });
    }
}
```

```
public class Startup
{
    private string _connection = null;

    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(env.ContentRootPath)
            .AddJsonFile("appsettings.json",
                         optional: false,
                         reloadOnChange: true)
            .AddEnvironmentVariables();

        if (env.IsDevelopment())
        {
            builder.AddUserSecrets<Startup>();
        }

        Configuration = builder.Build();
    }

    public IConfigurationRoot Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        var builder = new SqlConnectionStringBuilder(
            Configuration.GetConnectionString("Movies"));
        builder.Password = Configuration["DbPassword"];
        _connection = builder.ConnectionString;
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync($"DB Connection: {_connection}");
        });
    }
}
```

列出机密

假定应用程序的*secrets.json*文件包含以下两个密码：

```
{
  "Movies": {
    "ServiceApiKey": "12345",
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

从在其中的目录运行以下命令 *.csproj*文件是否存在：

```
dotnet user-secrets list
```

将显示以下输出：

```
Movies:ConnectionString = Server=(localdb)\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true
Movies:ServiceApiKey = 12345
```

在前面的示例中，在项名称中的冒号表示对象层次结构内的*secrets.json*。

删除单个机密

假定应用程序的*secrets.json*文件包含以下两个密码：

```
{
  "Movies": {
    "ServiceApiKey": "12345",
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

从在其中的目录运行以下命令 *.csproj*文件是否存在：

```
dotnet user-secrets remove "Movies:ConnectionString"
```

应用程序的*secrets.json*已修改文件，以删除与关联的键 / 值对 `MoviesConnectionString` 密钥：

```
{
  "Movies": {
    "ServiceApiKey": "12345"
  }
}
```

运行 `dotnet user-secrets list` 会显示以下消息：

```
Movies:ServiceApiKey = 12345
```

删除所有机密

假定应用程序的*secrets.json*文件包含以下两个密码：

```
{
  "Movies": {
    "ServiceApiKey": "12345",
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

从在其中的目录运行以下命令 *.csproj*文件是否存在：

```
dotnet user-secrets clear
```

从已删除应用程序的所有用户机密*secrets.json*文件：

```
{}
```

运行 `dotnet user-secrets list` 会显示以下消息：

```
No secrets configured for this application.
```

其他资源

- ASP.NET Core 中的配置
- 在 ASP.NET Core 的 azure 密钥保管库配置提供程序

强制实施 HTTPS 在 ASP.NET Core

2019/1/8 • • Edit Online

作者：Rick Anderson

本文档演示如何：

- 要求所有请求使用 HTTPS。
- 将所有 HTTP 请求重都定向到 HTTPS。

没有 API 可以防止客户端上的第一个请求发送敏感数据。

> **WARNING**
>
> 不要不使用RequireHttpsAttribute接收敏感信息的 Web api。`RequireHttpsAttribute` 使用 HTTP 状态代码将从 HTTP 到 HTTPS 的浏览器重定向。API 客户端可能无法理解或遵循从 HTTP 到 HTTPS 的重定向。此类客户端可能会通过 HTTP 发送的信息。Web Api 应具有下列任一：
>
> - 不侦听 HTTP。
> - 关闭与状态代码 400（错误请求）的连接并不为请求提供服务。

要求使用 HTTPS

我们建议该生产 ASP.NET Core web 应用调用：

- HTTPS 重定向中间件 (UseHttpsRedirection) 将 HTTP 请求重定向到 HTTPS。
- HSTS 中间件 (UseHsts) 发送给客户端的 HTTP 严格传输安全性协议 (HSTS) 标头。

> **NOTE**
>
> 反向代理配置中部署的应用程序允许代理在处理连接安全 (HTTPS)。如果代理还处理 HTTPS 的重定向，则无需使用 HTTPS 重定向中间件。如果代理服务器还处理编写 HSTS 标头 (例如，本机 HSTS 支持 IIS 10.0 (1709) 或更高版本)，HSTS 中间件不所需的应用程序。有关详细信息，请参阅退出的 HTTPS/HSTS 在项目创建。

UseHttpsRedirection

下面的代码调用 `UseHttpsRedirection` 在 `Startup` 类：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseMvc();
}
```

前面突出显示的代码：

- 使用默认HttpsRedirectionOptions.RedirectStatusCode (Status307TemporaryRedirect)。
- 使用默认HttpsRedirectionOptions.HttpsPort除非被重写 (null) `ASPNETCORE_HTTPS_PORT` 环境变量或IServerAddressesFeature。

我们建议使用临时重定向而不是永久重定向。链接缓存可以在开发环境中导致不稳定行为。如果想要发送的永久重定向状态代码时该应用位于非开发环境，请参阅配置在生产环境中的永久重定向部分。我们建议使用HSTS将信号发送到仅保护资源的客户端请求应发送到应用（仅在生产中）。

端口配置

端口必须是可用于中间件将不安全的请求重定向到 HTTPS。如果没有端口不可用：

- 不会发生重定向到 HTTPS。
- 中间件将记录警告"无法确定重定向的 https 端口。"

指定使用任何以下方法之一的 HTTPS 端口：

- 设置HttpsRedirectionOptions.HttpsPort。
- 设置 `ASPNETCORE_HTTPS_PORT` 环境变量或https_port Web 主机配置设置：

  密钥：`https_port`
  类型：string
  默认:未设置默认值。
  设置使用：`UseSetting`
  环境变量：`<PREFIX_>HTTPS_PORT` （该前缀 `ASPNETCORE_` 使用时Web 主机。）

  配置时IWebHostBuilder在 `Program`：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseSetting("https_port", "8080")
            .UseStartup<Startup>();
}
```

- 指示具有安全方案使用的端口 `ASPNETCORE_URLS` 环境变量。将服务器配置为环境变量。中间件间接发现通过 HTTPS 端口IServerAddressesFeature。反向代理部署这种方法不起作用。
- 在开发中，在中设置 HTTPS URL *launchsettings.json*。使用 IIS Express 时，请启用 HTTPS。
- 配置 HTTPS URL 终结点的面向公众 edge 部署Kestrel服务器或HTTP.sys服务器。仅一个 **HTTPS 端口**应用使用。中间件发现通过端口IServerAddressesFeature。

> **NOTE**
> 当应用在反向代理配置中，运行时IServerAddressesFeature不可用。设置使用一种其他方法在本部分中所述的端口。

当 Kestrel 或 HTTP.sys 用作面向公众的边缘服务器时，必须配置 Kestrel 或 HTTP.sys，若要在其上同时侦听：

- 其中重定向客户端的安全端口 (通常情况下，在生产和开发中的为 5001 的 443)。
- 不安全端口 (通常情况下，在生产环境中为 80) 和 5000 开发中。

不安全的端口必须可由客户端按顺序应用接收不安全的请求，并将客户端重定向到安全的端口。

有关详细信息，请参阅Kestrel 终结点配置或ASP.NET Core 中的 HTTP.sys Web 服务器实现。

部署方案

客户端和服务器之间的所有防火墙也都必须打开的流量的通信端口。

如果反向代理配置中将请求转发，则使用转发头中间件调用前 HTTPS 重定向中间件。转发头中间件更新 `Request.Scheme`，并使用 `X-Forwarded-Proto` 标头。中间件允许重定向 Uri 和其他安全策略才能正常工作。转发头中间件不使用时后，端应用程序可能不接收正确的方案和最终会在重定向循环。常见的最终用户错误消息是已发生过多的重定向。

部署到 Azure 应用服务时，请按照中的指导教程：将现有自定义 SSL 证书绑定到 Azure Web 应用。

选项

以下突出显示的代码调用AddHttpsRedirection配置中间件选项：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddHsts(options =>
    {
        options.Preload = true;
        options.IncludeSubDomains = true;
        options.MaxAge = TimeSpan.FromDays(60);
        options.ExcludedHosts.Add("example.com");
        options.ExcludedHosts.Add("www.example.com");
    });

    services.AddHttpsRedirection(options =>
    {
        options.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;
        options.HttpsPort = 5001;
    });
}
```

调用 `AddHttpsRedirection` 只是用来更改的值 `HttpsPort` 或 `RedirectStatusCode` 。

前面突出显示的代码：

- 集HttpsRedirectionOptions.RedirectStatusCode到Status307TemporaryRedirect，这是默认值。使用的字段StatusCodes分配到的类 `RedirectStatusCode` 。
- 设置 HTTPS 端口为 5001。默认值为 443。

在生产环境中配置永久重定向

中间件默认为发送Status307TemporaryRedirect与所有重定向。如果想要发送的永久重定向状态代码，非开发环境中应用程序时，包装在对非开发环境的条件检查中的中间件的选项配置。

配置时 `IWebHostBuilder` 中*Startup.cs*：

```
public void ConfigureServices(IServiceCollection services)
{
    // IHostingEnvironment (stored in _env) is injected into the Startup class.
    if (!_env.IsDevelopment())
    {
        services.AddHttpsRedirection(options =>
        {
            options.RedirectStatusCode = StatusCodes.Status308PermanentRedirect;
            options.HttpsPort = 443;
        });
    }
}
```

HTTPS 重定向中间件另一种方法

使用 HTTPS 重定向中间件的替代方法（`UseHttpsRedirection`）是使用 URL 重写中间件（`AddRedirectToHttps`）。`AddRedirectToHttps` 此外可以设置的状态代码和端口时执行重定向。有关详细信息，请参阅URL 重写中间件。

当将重定向到 HTTPS 而无需其他重定向规则，我们建议使用 HTTPS 重定向中间件（`UseHttpsRedirection`）本主题中所述。

RequireHttpsAttribute用于要求 HTTPS。`[RequireHttpsAttribute]` 可以修饰控制器或方法，也可以全局应用。若要全局应用该属性，将以下代码添加到 `ConfigureServices` 在 `Startup`：

```
// Requires using Microsoft.AspNetCore.Mvc;
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new RequireHttpsAttribute());
    });
```

前面突出显示的代码要求所有请求都使用 `HTTPS`；因此，HTTP 请求将被忽略。以下突出显示的代码将所有 HTTP 请求重都定向到 HTTPS：

```
// Requires using Microsoft.AspNetCore.Rewrite;
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    var options = new RewriteOptions()
       .AddRedirectToHttps();

    app.UseRewriter(options);
```

有关详细信息，请参阅URL 重写中间件。中间件还允许应用执行重定向时设置的状态代码或状态代码和端口。

全局需要 HTTPS（`options.Filters.Add(new RequireHttpsAttribute());`）是最佳安全方案。将应用 `[RequireHttps]` 属性设置为所有控制器/Razor 页面不会被视为与需要 HTTPS 全局一样安全。不能保证 `[RequireHttps]` 时添加新的控制器和 Razor 页面应用属性。

HTTP 严格传输安全协议 (HSTS)

每个OWASP，HTTP 严格传输安全性 (HSTS)是通过响应标头使用的 web 应用指定选择的安全增强功能。当浏览器支持 HSTS收到此标头：

- 在浏览器将会阻止通过 HTTP 发送的任何通信的域的配置存储。在浏览器强制通过 HTTPS 进行的所有通信。
- 在浏览器可防止用户使用不受信任或无效的证书。在浏览器禁用允许用户暂时信任此证书的提示。

因为由客户端强制执行 HSTS 它具有一些限制：

- 客户端必须支持 HSTS。
- HSTS 要求至少一个成功的 HTTPS 请求能够建立 HSTS 策略。
- 应用程序必须检查每个 HTTP 请求和重定向或拒绝的 HTTP 请求。

ASP.NET Core 2.1 或更高版本实现与 HSTS `UseHsts` 扩展方法。下面的代码调用 `UseHsts` 时应用不在开发模式：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseMvc();
}
```

`UseHsts` 不建议在开发过程中由于 HSTS 设置高度可缓存的浏览器。默认情况下，`UseHsts` 排除本地环回地址。

对于生产环境首次实现 HTTPS 设置初始HstsOptions.MaxAge为使用其中一个较小值TimeSpan方法。设置的值从小时数不超过一天的以防到时需要还原 HTTP 到 HTTPS 基础结构。确信 HTTPS 配置的可持续发展中后，增加 HSTS 最大期限值;常用的值为一年。

下面的代码：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddHsts(options =>
    {
        options.Preload = true;
        options.IncludeSubDomains = true;
        options.MaxAge = TimeSpan.FromDays(60);
        options.ExcludedHosts.Add("example.com");
        options.ExcludedHosts.Add("www.example.com");
    });

    services.AddHttpsRedirection(options =>
    {
        options.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;
        options.HttpsPort = 5001;
    });
}
```

- 设置预加载严格传输安全性标头的参数。预加载不属于RFC HSTS 规范，但要预加载 HSTS 站点上执行全新安装的 web 浏览器支持。有关详细信息，请参阅 https://hstspreload.org/。
- 使includeSubDomain，这将 HSTS 策略应用到主机的子域。
- 显式将严格传输安全性标头的最大期限参数设置为 60 天。如果未设置，默认值为 30 天。请参阅最大期限指令有关详细信息。
- 添加 `example.com` 到主机以排除列表。

`UseHsts` 不包括以下环回主机：

- `localhost`：IPv4 环回地址。
- `127.0.0.1`：IPv4 环回地址。
- `[::1]`：IPv6 环回地址。

选择退出的 HTTPS/HSTS 在项目创建

在其中连接安全处理面向公众边缘网络的某些后端服务情况下，无需在每个节点上配置连接安全。Web 应用从 Visual Studio 中或从模板生成dotnet 新命令启用HTTPS 的重定向并HSTS。对于不需要这些方案的部署，你可以选择退出的 HTTPS/HSTS 时从模板创建的应用。

若要选择退出的 HTTPS/HSTS:

- Visual Studio
- .NET Core CLI

取消选中**配置为支持 HTTPS**复选框。

信任在 Windows 和 macOS 上的 ASP.NET Core HTTPS 开发证书

.NET core SDK 包括 HTTPS 开发证书。首次运行体验的一部分安装证书。例如，`dotnet --info`生成类似于以下输出：

```
ASP.NET Core
------------
Successfully installed the ASP.NET Core HTTPS Development Certificate.
To trust the certificate run 'dotnet dev-certs https --trust' (Windows and macOS only).
For establishing trust on other platforms refer to the platform specific documentation.
For more information on configuring HTTPS see https://go.microsoft.com/fwlink/?linkid=848054.
```

安装.NET Core SDK 将 ASP.NET Core HTTPS 开发证书安装到本地用户证书存储区。已安装证书，但它具有不受信任。若要信任该证书执行一次性步骤，运行 dotnet`dev-certs`工具：

```
dotnet dev-certs https --trust
```

下面的命令上提供帮助`dev-certs`工具：

```
dotnet dev-certs https --help
```

如何设置适用于 Docker 的开发人员证书

请参阅此 GitHub 问题。

其他信息

- 配置 ASP.NET Core 以使用代理服务器和负载均衡器
- 使用 Apache 在 Linux 上托管 ASP.NET Core:HTTPS 配置
- 使用 Nginx 在 Linux 上托管 ASP.NET Core:HTTPS 配置
- 如何在 IIS 上设置 SSL
- OWASP HSTS 浏览器支持

在 ASP.NET Core 欧洲常规数据保护法规 (GDPR) 支持

2019/1/11 • • Edit Online

作者：Rick Anderson

ASP.NET Core 提供 Api 和模板，以帮助满足一些欧洲常规数据保护法规 (GDPR)要求：

- 项目模板包含扩展点以及您可以替换为您的隐私和 cookie 的使用策略的存根的标记。
- 用于存储的个人信息中，cookie 许可功能，可要求（和跟踪）同意的情况下从你的用户。如果某个用户尚未同意数据收集，并且应用了CheckConsentNeeded设置为 `true`，非必需 cookie 不会发送到浏览器。
- Cookie 可以标记为重要。用户尚未同意和跟踪被禁用时，甚至必不可少的 cookie 发送到浏览器。
- TempData 和会话 cookie跟踪处于禁用状态时无法正常工作。
- 标识管理页提供了用于下载和删除用户数据的链接。

示例应用允许测试大部分 GDPR 扩展点和 Api 添加到 ASP.NET Core 2.1 模板。请参阅自述文件测试说明的文件。

查看或下载示例代码（如何下载）

模板中的 ASP.NET Core GDPR 支持生成的代码

Razor 页面和 MVC 使用的项目模板创建的项目包括以下 GDPR 支持：

- CookiePolicyOptions并UseCookiePolicy中设置 `Startup`。
- *_CookieConsentPartial.cshtml* 分部视图。
- *Pages/Privacy.cshtml*页或*Views/Home/Privacy.cshtml*视图提供了详细介绍站点的隐私策略的页。*_CookieConsentPartial.cshtml*文件可生成隐私页的链接。
- 对于使用单个用户帐户创建的应用程序，管理页提供链接以下载和删除个人用户数据。

CookiePolicyOptions 和 UseCookiePolicy

CookiePolicyOptions中初始化 `Startup.ConfigureServices`：

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services
    // to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<CookiePolicyOptions>(options =>
        {
            // This lambda determines whether user consent for non-essential cookies
            // is needed for a given request.
            options.CheckConsentNeeded = context => true;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });

        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>()
            .AddEntityFrameworkStores<ApplicationDbContext>();

        // If the app uses session state, call AddSession.
        // services.AddSession();

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    }

    // This method gets called by the runtime. Use this method to configure the
    // HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();

        app.UseAuthentication();

        // If the app uses session state, call Session Middleware after Cookie
        // Policy Middleware and before MVC Middleware.
        // app.UseSession();

        app.UseMvc();
    }
}
```

UseCookiePolicy名为 `Startup.Configure`：

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services
    // to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<CookiePolicyOptions>(options =>
        {
            // This lambda determines whether user consent for non-essential cookies
            // is needed for a given request.
            options.CheckConsentNeeded = context => true;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });

        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>()
            .AddEntityFrameworkStores<ApplicationDbContext>();

        // If the app uses session state, call AddSession.
        // services.AddSession();

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    }

    // This method gets called by the runtime. Use this method to configure the
    // HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();

        app.UseAuthentication();

        // If the app uses session state, call Session Middleware after Cookie
        // Policy Middleware and before MVC Middleware.
        // app.UseSession();

        app.UseMvc();
    }
}
```

_CookieConsentPartial.cshtml 分部视图

_CookieConsentPartial.cshtml分部视图:

```
@using Microsoft.AspNetCore.Http.Features

@{
    var consentFeature = Context.Features.Get<ITrackingConsentFeature>();
    var showBanner = !consentFeature?.CanTrack ?? false;
    var cookieString = consentFeature?.CreateConsentCookie();
}

@if (showBanner)
{
    <nav id="cookieConsent" class="navbar navbar-default navbar-fixed-top" role="alert">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target="#cookieConsent .navbar-collapse">
                    <span class="sr-only">Toggle cookie consent banner</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <span class="navbar-brand"><span class="glyphicon glyphicon-info-sign" aria-
hidden="true"></span></span>
            </div>
            <div class="collapse navbar-collapse">
                <p class="navbar-text">
                    Use this space to summarize your privacy and cookie use policy.
                </p>
                <div class="navbar-right">
                    <a asp-page="/Privacy" class="btn btn-info navbar-btn">Learn More</a>
                    <button type="button" class="btn btn-default navbar-btn" data-cookie-
string="@cookieString">Accept</button>
                </div>
            </div>
        </div>
    </nav>
    <script>
        (function () {
            document.querySelector("#cookieConsent button[data-cookie-string]").addEventListener("click",
function (el) {
                document.cookie = el.target.dataset.cookieString;
                document.querySelector("#cookieConsent").classList.add("hidden");
            }, false);
        })();
    </script>
}
```

此部分中：

- 获取用户跟踪的状态。如果应用程序配置为需要同意的情况下，用户必须同意使用之前可以跟踪 cookie。如果需要同意的情况下，cookie 同意面板固定的创建的导航栏的顶部 *_Layout.cshtml*文件。
- 提供了一个 HTML `<p>` 元素以汇总您的隐私和 cookie 使用策略。
- 提供指向隐私页面或视图，其中详细介绍站点的隐私策略。

基本 cookie

如果尚未赋予同意的情况下，只有标记为重要的 cookie 发送到浏览器。下面的代码使 cookie 重要：

```
public IActionResult OnPostCreateEssentialAsync()
{
    HttpContext.Response.Cookies.Append(Constants.EssentialSec,
        DateTime.Now.Second.ToString(),
        new CookieOptions() { IsEssential = true });

    ResponseCookies = Response.Headers[HeaderNames.SetCookie].ToString();

    return RedirectToPage("./Index");
}
```

Tempdata 提供程序和会话状态的 cookie 不重要

Tempdata 提供程序cookie 不是必需的。如果禁用跟踪，Tempdata 提供程序不起作用。若要启用 Tempdata 提供程序跟踪处于禁用状态时，将 TempData cookie 标记为重要中 `Startup.ConfigureServices`：

```
// The Tempdata provider cookie is not essential. Make it essential
// so Tempdata is functional when tracking is disabled.
services.Configure<CookieTempDataProviderOptions>(options => {
    options.Cookie.IsEssential = true;
});
```

会话状态cookie 不重要。跟踪处于禁用状态时，会话状态不起作用。

个人数据

使用个人用户帐户创建的 ASP.NET Core 应用包括要下载和删除个人数据的代码。

选择的用户名称，然后选择**个人数据**：

注意：

- 若要生成 `Account/Manage` 代码，请参阅基架标识。
- 删除并下载链接只作用于的默认标识数据。创建自定义用户数据的应用，必须进行扩展，以删除/下载自定义用户数据。有关详细信息，请参阅添加、下载和删除自定义用户数据到标识。
- 保存用户的标识数据库表中存储的令牌 `AspNetUserTokens` 时通过级联删除方式，由于删除了用户会被删除外键。

静态加密

某些数据库和存储机制允许静态加密。静态加密：

- 自动对存储的数据进行加密。
- 无需配置、编程中或其他工作的软件的访问的数据进行加密。
- 是最简单且最安全的选项。
- 允许管理密钥和加密的数据库。

例如：

- Microsoft SQL 和 Azure SQL 提供了透明数据加密(TDE)。
- SQL Azure 默认情况下加密数据库
- 默认情况下加密 azure Blob、文件、表和队列存储。

对于未提供静态的内置加密的数据库，您可能能够使用磁盘加密来提供相同的保护。例如：

- 适用于 Windows Server 的 BitLocker
- Linux:
  - eCryptfs
  - EncFS。

其他资源

- Microsoft.com/GDPR

在 ASP.NET Core 的 azure 密钥保管库配置提供程序

2018/10/30 • • Edit Online

通过Luke Latham和Andrew Stanton-nurse

本文档介绍如何使用Microsoft Azure Key Vault配置提供程序将从 Azure Key Vault 机密中加载应用配置值。Azure Key Vault 是基于云的服务，可帮助你保护加密密钥和机密使用的应用和服务。常见的方案包括控制对敏感的配置数据的访问并符合 fips 140-2 的要求级别 2 硬件安全模块 (HSM) 时验证存储配置数据。此功能仅适用于面向 ASP.NET Core 1.1 的应用或更高版本。

查看或下载示例代码（如何下载）

Package

若要使用的提供程序，添加到引用Microsoft.Extensions.Configuration.AzureKeyVault包。

应用配置

你可以浏览的提供程序示例应用。建立密钥保管库并在保管库中创建机密后，示例应用安全地将机密值加载到其配置并在网页中显示它们。

提供程序添加到应用程序的配置中使用 `AddAzureKeyVault` 扩展。在示例应用中，则该扩展将从加载的三个配置值*appsettings.json*文件。

应用设置	描述	示例
`Vault`	Azure 密钥保管库名称	contosovault
`ClientId`	Azure Active Directory 应用 Id	627e911e-43cc-61d4-992e-12db9c81b413
`ClientSecret`	Azure Active Directory 应用密钥	g58K3dtg59o1Pa+e59v2Tx829w6VxTB2yv9sv/101di=

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
        {
            var builtConfig = config.Build();

            config.AddAzureKeyVault(
                $"https://{builtConfig["Vault"]}.vault.azure.net/",
                builtConfig["ClientId"],
                builtConfig["ClientSecret"]);
        })
        .UseStartup<Startup>();
```

创建密钥保管库机密并加载配置值（basic 示例）

1. 创建密钥保管库并将设置 Azure Active Directory (Azure AD) 应用程序中的指南开始使用 Azure 密钥保

管库。

- 将机密添加到密钥保管库使用AzureRM 密钥保管库 PowerShell 模块从可用PowerShell 库，则Azure 密钥保管库 REST API，或者Azure 门户。为创建的机密*手动*或*证书*机密。*证书*机密使用的应用和服务证书，但不是支持配置提供程序。应使用*手动*选项可创建带有配置提供程序使用的名称-值对机密。
  - 以名称-值对的形式创建简单的机密。Azure 密钥保管库机密名称被限制为字母数字字符和短划线。
  - 层次结构的值（配置节）使用 `--`（两个短划线）作为分隔符，在该示例。通常用于分隔的子项中的一部分的冒号ASP.NET Core 配置，机密名称中不允许出现。因此，两个短划线的使用和机密加载到应用的配置时交换的冒号。
  - 创建两个*手动*具有以下名称 / 值对的机密。第一个机密是一个简单的名称和值，并第二个密钥使用部分和子项中的机密名称创建机密值：
    - `SecretName`：`secret_value_1`
    - `Section--SecretName`：`secret_value_2`
- 与 Azure Active Directory 中注册示例应用程序。
- 授权应用访问密钥保管库。当你使用 `Set-AzureRmKeyVaultAccessPolicy` PowerShell cmdlet 来授权应用程序访问密钥保管库，提供 `List` 并 `Get` 对使用机密的访问 `-PermissionsToSecrets list,get`。

2. 更新应用程序的*appsettings.json*包含的值的文件 `Vault`， `ClientId`，和 `ClientSecret`。

3. 运行示例应用，获取从其配置值 `IConfigurationRoot` 具有作为机密名称相同的名称。

- 非层次结构值：的值 `SecretName` 一起被获取 `config["SecretName"]`。
- 层次结构的值（部分）：使用 `:`（冒号）表示法或 `GetSection` 扩展方法。使用任一方法获取的配置值：
  - `config["Section:SecretName"]`
  - `config.GetSection("Section")["SecretName"]`

运行应用时，网页显示加载的机密值：

Key Vault Configuration Provider Sample

Secret	Name in Key Vault	Obtained from Configuration	Value
SecretName	**SecretName**	`Configuration["SecretName"]`	secret_value_1
Section:SecretName	**Section--SecretName**	`Configuration["Section:SecretName"]`	secret_value_2
		`Configuration.GetSection("Section")["SecretName"]`	secret_value_2

将数组绑定至类

提供程序支持的配置值读入数组绑定到 POCO 数组。

从允许包含冒号的键的配置源读取数据时（`:`）使用分隔符，数字键段来区分对数组组成的密钥（`:0:`， `:1:`，… `:{n}:`）格式模式中出现的位置生成。有关详细信息，请参阅配置：将数组绑定到一个类。

Azure 密钥保管库密钥不能使用冒号作为分隔符。本主题中介绍的方法使用双短划线（`--`）作为（部分）的层次结构值的分隔符。使用双短划线和数值的关键段数组密钥存储在 Azure 密钥保管库（`--0--`， `--1--`，… `--{n}--`）格式模式中出现的位置生成。

请参阅以下Serilog日志记录提供程序配置提供的 JSON 文件。有两个对象中定义的文本 `WriteTo` 数组，它反映两个 Serilog*接收器*，其中介绍了日志记录输出的目标：

```
"Serilog": {
  "WriteTo": [
    {
      "Name": "AzureTableStorage",
      "Args": {
        "storageTableName": "logs",
        "connectionString": "DefaultEnd...ountKey=Eby8...GMGw=="
      }
    },
    {
      "Name": "AzureDocumentDB",
      "Args": {
        "endpointUrl": "https://contoso.documents.azure.com:443",
        "authorizationKey": "Eby8...GMGw=="
      }
    }
  ]
}
```

在上面的 JSON 文件中所示的配置存储在 Azure 密钥保管库中使用双短划线（`--`）表示法和数字段：

键	“值”
`Serilog--WriteTo--0--Name`	`AzureTableStorage`
`Serilog--WriteTo--0--Args--storageTableName`	`logs`
`Serilog--WriteTo--0--Args--connectionString`	`DefaultEnd...ountKey=Eby8...GMGw==`
`Serilog--WriteTo--1--Name`	`AzureDocumentDB`
`Serilog--WriteTo--1--Args--endpointUrl`	`https://contoso.documents.azure.com:443`
`Serilog--WriteTo--1--Args--authorizationKey`	`Eby8...GMGw==`

创建带前缀的密钥保管库机密并加载配置值（密钥的名称的前缀的示例）

`AddAzureKeyVault` 此外提供了接受的实现的重载 `IKeyVaultSecretManager`，可用于控制如何密钥保管库机密将转换为配置项。例如，可以实现的接口来加载基于你在应用启动时提供的前缀值的机密值。这可以使您例如，若要加载基于应用的版本的密钥。

> **WARNING**
> 若要将多个应用的机密放入同一个密钥保管库或将环境机密不使用在密钥保管库机密的前缀（例如，*开发*而不是*生产*机密）到相同保管库。我们建议，不同的应用程序和开发/生产环境使用单独的密钥保管库来隔离应用环境的最高级别的安全性。

使用第二个示例应用程序，在 key vault 中创建机密 `5000-AppSecret`（密钥保管库机密名称中不允许句点）表示您的应用程序的版本为 5.0.0.0 应用程序密码。有关另一个版本，5.1.0.0，创建的机密 `5100-AppSecret`。每个应用程序版本将其自己的机密值加载到作为其配置 `AppSecret`、在加载机密的版本剥离。该示例的实现如下所示：

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
        {
            // The appVersion obtains the app version (5.0.0.0), which
            // is set in the project file and obtained from the entry
            // assembly. The versionPrefix holds the version without
            // dot notation for the PrefixKeyVaultSecretManager.
            var appVersion = Assembly.GetEntryAssembly().GetName().Version.ToString();
            var versionPrefix = appVersion.Replace(".", string.Empty);

            var builtConfig = config.Build();

            config.AddAzureKeyVault(
                $"https://{builtConfig["Vault"]}.vault.azure.net/",
                builtConfig["ClientId"],
                builtConfig["ClientSecret"],
                new PrefixKeyVaultSecretManager(versionPrefix));
        })
        .UseStartup<Startup>();
```

```
public class PrefixKeyVaultSecretManager : IKeyVaultSecretManager
{
    private readonly string _prefix;

    public PrefixKeyVaultSecretManager(string prefix)
    {
        _prefix = $"{prefix}-";
    }

    public bool Load(SecretItem secret)
    {
        // Load a vault secret when its secret name starts with the
        // prefix. Other secrets won't be loaded.
        return secret.Identifier.Name.StartsWith(_prefix);
    }

    public string GetKey(SecretBundle secret)
    {
        // Remove the prefix from the secret name and replace two
        // dashes in any name with the KeyDelimiter, which is the
        // delimiter used in configuration (usually a colon). Azure
        // Key Vault doesn't allow a colon in secret names.
        return secret.SecretIdentifier.Name
            .Substring(_prefix.Length)
            .Replace("--", ConfigurationPath.KeyDelimiter);
    }
}
```

`Load` 方法调用循环访问保管库机密以找出具有版本前缀的提供程序算法。当版本前缀找到 `Load`，该算法使用 `GetKey` 方法返回的机密名称的配置名称。它剥离机密的名称从版本前缀，并返回到应用程序的配置名称-值对的加载的机密名称的其余部分。

当您实现这种方法：

1. 密钥保管库机密进行加载。
2. 字符串密码 `5000-AppSecret` 匹配。
3. 版本 `5000` 去除（替换为短划线），从离开的键名 `AppSecret` 机密值加载到应用的配置。

> **NOTE**
> 您还可以提供您自己 `KeyVaultClient` 实现 `AddAzureKeyVault` 。提供自定义客户端可以共享单个实例的配置提供程序和您的应用程序的其他部件之间的客户端。

1. 创建密钥保管库并将设置 Azure Active Directory (Azure AD) 应用程序中的指南开始使用 Azure 密钥保管库。
   - 将机密添加到密钥保管库使用AzureRM 密钥保管库 PowerShell 模块从可用PowerShell 库，则Azure 密钥保管库 REST API，或者Azure 门户。为创建的机密*手动*或*证书*机密。*证书*机密使用的应用和服务证书，但不是支持配置提供程序。应使用*手动*选项可创建带有配置提供程序使用的名称-值对机密。
     - 层次结构的值（配置节）使用 `--`（两个短划线）作为分隔符。
     - 创建两个*手动*具有以下名称 / 值对的机密：
       - `5000-AppSecret` : `5.0.0.0_secret_value`
       - `5100-AppSecret` : `5.1.0.0_secret_value`
   - 与 Azure Active Directory 中注册示例应用程序。
   - 授权应用访问密钥保管库。当你使用 `Set-AzureRmKeyVaultAccessPolicy` PowerShell cmdlet 来授权应用程序访问密钥保管库，提供 `List` 并 `Get` 对使用机密的访问 `-PermissionsToSecrets list,get` 。
2. 更新应用程序的*appsettings.json*包含的值的文件 `Vault` ， `ClientId` ，和 `ClientSecret` 。
3. 运行示例应用，获取从其配置值 `IConfigurationRoot` 具有作为带前缀的机密名称相同的名称。在此示例中，前缀是应用的版本，提供给 `PrefixKeyVaultSecretManager` 添加 Azure Key Vault 配置提供程序时。值 `AppSecret` 一起被获取 `config["AppSecret"]` 。由应用生成的网页显示加载的值：

Key Vault Configuration Provider Sample

Secret	Name in Key Vault	Obtained from Configuration	Value
AppSecret	**5000-AppSecret**	`Configuration["AppSecret"]`	5.0.0.0_secret_value

4. 更改中的项目文件中的应用程序集版本 `5.0.0.0` 到 `5.1.0.0` 并再次运行应用。这一次，返回的机密值是 `5.1.0.0_secret_value` 。由应用生成的网页显示加载的值：

Key Vault Configuration Provider Sample

Secret	Name in Key Vault	Obtained from Configuration	Value
AppSecret	**5100-AppSecret**	`Configuration["AppSecret"]`	5.1.0.0_secret_value

控制对客户端密码的访问

使用机密管理器工具维护 `ClientSecret` 外部项目源树。使用机密管理器中，你将应用程序机密与特定项目相关联并跨多个项目共享它们。

在开发环境支持证书中的.NET Framework 应用时，可以使用 X.509 证书验证到 Azure 密钥保管库。X.509 证书的私钥由操作系统管理。有关详细信息，请参阅使用证书而不是客户端密码进行身份验证。使用 `AddAzureKeyVault` 接受重载 `X509Certificate2` (`_env` 在下面的示例：

```
var builtConfig = config.Build();

var store = new X509Store(StoreLocation.CurrentUser);
store.Open(OpenFlags.ReadOnly);
var cert = store.Certificates
    .Find(X509FindType.FindByThumbprint,
        config["CertificateThumbprint"], false);

config.AddAzureKeyVault(
    builtConfig["Vault"],
    builtConfig["ClientId"],
    cert.OfType<X509Certificate2>().Single(),
    new EnvironmentSecretManager(context.HostingEnvironment.ApplicationName));

store.Close();
```

重新加载机密

机密缓存，直至 `IConfigurationRoot.Reload()` 调用。已过期，已禁用，并且已更新密钥保管库中的机密不遵循之前应用此 `Reload` 执行。

```
Configuration.Reload();
```

已禁用和过期的机密

已禁用并已过期机密引发 `KeyVaultClientException` 。若要防止应用引发，将为您的应用程序或更新已禁用或已过期机密。

疑难解答

如果应用程序无法加载使用提供程序的配置，一条错误消息写入到ASP.NET Core 日志记录基础结构。以下条件下将不加载配置：

- 应用未正确配置 Azure Active Directory 中。
- 密钥保管库不存在于 Azure 密钥保管库。
- 应用程序无权访问密钥保管库。
- 访问策略不包括 `Get` 和 `List` 权限。
- 在密钥保管库，配置数据（名称 / 值对）是错误命名为，缺少，禁用，或已过期。
- 应用了错误的密钥保管库名称（`Vault`），Azure AD 应用程序 Id（`ClientId`），或 Azure AD 密钥（`ClientSecret`）。
- Azure AD 密钥（`ClientSecret`）已过期。
- 配置密钥（名称）不正确的应用中的想要加载的值。

其他资源

- ASP.NET Core 中的配置
- Microsoft Azure：密钥保管库
- Microsoft Azure：密钥保管库文档
- 如何生成和传输受 HSM 保护密钥的 Azure 密钥保管库
- KeyVaultClient 类

在 ASP.NET Core 防止跨站点请求伪造 (XSRF/CSRF) 攻击

2019/1/8 • • Edit Online

通过Steve Smith，Fiyaz Hasan，和Rick Anderson

跨站点请求伪造 (也称为 XSRF 或 CSRF，读作，*请参阅冲浪*) 是针对恶意网站凭此可以影响客户端浏览器和信任的 web 应用之间的交互的 web 托管的应用程序的攻击浏览器。这些攻击是可能的因为 web 浏览器将自动随每个请求某些类型的身份验证令牌发送到网站。这种形式的攻击是也称为*一键式攻击*或*会话行进*因为此攻击利用用户的身份验证会话。

CSRF 攻击的示例：

1. 用户登录到 `www.good-banking-site.com` 使用窗体身份验证。服务器对用户进行身份验证，并会发出包含身份验证 cookie 的响应。站点是易受到攻击，因为它信任的任何请求都收到与有效的身份验证 cookie。

2. 在用户访问恶意站点， `www.bad-crook-site.com` 。

   恶意网站 `www.bad-crook-site.com` ，包含 HTML 窗体如下所示：

   ```
   <h1>Congratulations! You're a Winner!</h1>
   <form action="http://good-banking-site.com/api/account" method="post">
       <input type="hidden" name="Transaction" value="withdraw">
       <input type="hidden" name="Amount" value="1000000">
       <input type="submit" value="Click to collect your prize!">
   </form>
   ```

   请注意，窗体的 `action` 帖子到易受攻击的站点上，而不是恶意站点。这是 CSRF 的"跨站点"部分。

3. 用户选择提交按钮。浏览器发出请求，并会自动包括请求的域的身份验证 cookie `www.good-banking-site.com` 。

4. 运行该请求 `www.good-banking-site.com` 与用户的身份验证上下文的服务器，并且可以执行身份验证的用户可以执行任何操作。

除了其中用户选择按钮以提交窗体方案中，恶意网站可以：

- 运行自动提交窗体的脚本。
- AJAX 请求用以发送窗体提交。
- 隐藏窗体使用 CSS。

这些替代方案不需要任何操作或从用户而不是最初访问恶意网站的输入。

使用 HTTPS，则不会阻止 CSRF 攻击。可以发送恶意站点 `https://www.good-banking-site.com/` 请求一样方便它可以发送不安全的请求。

一些攻击目标终结点的响应 GET 请求，在这种情况下的图像标记可用于执行的操作。这种形式的攻击的允许的映像，但阻止 JavaScript 论坛站点上很常见。应用程序来更改的状态的 GET 请求，在其中更改变量或资源，是很容易受到恶意攻击。**更改状态的 GET 请求将不安全。最佳做法是永远不会更改上的 GET 请求的状态。**

CSRF 攻击是可能使用 cookie 进行身份验证，因为 web 应用：

- 浏览器存储 cookie 颁发的 web 应用。
- 存储的 cookie 包括用于身份验证的用户的会话 cookie。
- 浏览器发送的所有 cookie 与域关联到 web 应用而不考虑对应用程序的请求如何生成在浏览器中的每个请求。

但是，到限于 CSRF 攻击利用 cookie。例如，基本和摘要式身份验证也是易受攻击的。浏览器使用基本或摘要式身份验证的用户登录后，直到在会话自动发送凭据†结束。

†在此上下文中，*会话*指的是在此期间用户进行身份验证的客户端会话。它是与服务器端会话无关或ASP.NET Core 会话中间件。

用户可以通过采取预防措施来防止 CSRF 漏洞：

- 从 web 应用时将它们完成相应操作进行签名。
- 定期清除浏览器 cookie。

但是，CSRF 漏洞从根本上说是 web 应用，而不是最终用户的问题。

身份验证基础知识

基于 cookie 的身份验证是常用形式的身份验证。基于令牌的身份验证系统越来越流行，尤其是对于单页面应用程序 (Spa)。

基于 cookie 的身份验证

当用户通过使用其用户名和密码时，它们被颁发一个令牌，其中包含可用于身份验证和授权的身份验证票证。作为一个附带的每个请求客户端的 cookie 会使存储令牌。由 Cookie 身份验证中间件执行生成和验证此 cookie。中间件的加密 cookie 序列化为一个用户主体。在后续请求中，中间件将验证 cookie、重新创建主体，并将分配到主体用户的属性HttpContext。

基于令牌的身份验证

当用户进行身份验证时，它们被颁发的令牌（不防伪令牌）。该令牌包含的窗体中的用户信息声明或指向用户状态保留在应用程序的应用程序的引用标记。当用户尝试访问资源需要身份验证时，则会将令牌发送到其他 authorization 标头中的持有者令牌的窗体应用。这使得应用程序无状态。在每个后续请求的令牌请求中传递的服务器端验证。此令牌不是*加密*，它具有*编码*。在服务器上，将解码令牌来访问其信息。若要在后续请求中发送令牌，请在浏览器的本地存储中存储令牌。如果该令牌存储在浏览器的本地存储，则不会担心 CSRF 的漏洞。CSRF 令牌存储在 cookie 中时是一个问题。

在一个域上托管的多个应用

共享宿主环境为易受到会话劫持、登录名 CSRF 和其他攻击。

尽管 `example1.contoso.net` 并 `example2.contoso.net` 是不同的主机，在主机之间没有隐式信任关系 `*.contoso.net` 域。此隐式信任关系允许可能不受信任的主机会影响彼此的 cookie（控制 AJAX 请求的同域策略不一定适用于 HTTP cookie）。

通过不共享域，可以防止攻击，可利用同一个域上托管的应用程序之间的受信任的 cookie。在其自己的域中托管每个应用，则时利用任何隐式 cookie 信任关系。

ASP.NET Core antiforgery 配置

> **WARNING**
>
> ASP.NET Core 实现 antiforgery 使用ASP.NET Core 数据保护。数据保护堆栈必须配置为在服务器场中工作。请参阅配置数据保护有关详细信息。

在 ASP.NET Core 2.0 或更高版本，FormTagHelper antiforgery 令牌注入 HTML 窗体元素。Razor 文件中

的以下标记自动生成防伪令牌：

```
<form method="post">
    ...
</form>
```

类似地，IHtmlHelper.BeginForm默认情况下生成防伪令牌，如果窗体的方法不是 GET。

HTML 窗体元素自动生成的防伪令牌发生时 `<form>` 标记包含 `method="post"` 属性和以下任一条件成立：

- 操作属性为空（`action=""`）。
- 操作属性不提供（`<form method="post">`）。

可以禁用自动生成的防伪标记 HTML 窗体元素：

- 显式禁用防伪令牌，且 `asp-antiforgery` 属性：

  ```
  <form method="post" asp-antiforgery="false">
      ...
  </form>
  ```

- 窗体元素是选择向外的标记帮助程序使用标记帮助程序！ 选择退出符号：

  ```
  <!form method="post">
      ...
  </!form>
  ```

- 删除 `FormTagHelper` 从视图。 `FormTagHelper` 可以通过将以下指令添加到 Razor 视图从视图中删除：

  ```
  @removeTagHelper Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper,
  Microsoft.AspNetCore.Mvc.TagHelpers
  ```

> **NOTE**
>
> Razor 页面会自动防范 XSRF/CSRF。有关详细信息，请参阅XSRF/CSRF 和 Razor 页面。

为抵御 CSRF 攻击最常用的方法是使用*同步器标记模式*(STP)。当用户请求的页面包含窗体数据使用 STP：

1. 服务器发送到客户端的当前用户的标识相关联的令牌。
2. 客户端返回将令牌发送到服务器进行验证。
3. 如果服务器收到与经过身份验证的用户的标识不匹配的令牌，将拒绝请求。

该令牌唯一且不可预测。该令牌还可用于确保正确序列化的一系列的请求（例如，确保请求序列的：第 1 页–第 2 页–第 3 页）。ASP.NET Core MVC 和 Razor 页模板中的窗体的所有生成 antiforgery 令牌。以下两个视图的示例生成防伪令牌：

```
<form asp-controller="Manage" asp-action="ChangePassword" method="post">
    ...
</form>

@using (Html.BeginForm("ChangePassword", "Manage"))
{
    ...
}
```

显式添加到防伪令牌 `<form>` 而无需使用标记帮助程序与 HTML 帮助程序元素 @Html.AntiForgeryToken：

```
<form action="/" method="post">
    @Html.AntiForgeryToken()
</form>
```

在每个前面的情况下，ASP.NET Core 添加类似于以下一个隐藏的表单字段：

```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw">
```

ASP.NET Core 包括三个筛选器来处理 antiforgery 令牌：

- ValidateAntiForgeryToken
- AutoValidateAntiforgeryToken
- IgnoreAntiforgeryToken

防伪选项

自定义防伪选项中 `Startup.ConfigureServices`：

```
services.AddAntiforgery(options =>
{
    // Set Cookie properties using CookieBuilder properties†.
    options.FormFieldName = "AntiforgeryFieldname";
    options.HeaderName = "X-CSRF-TOKEN-HEADERNAME";
    options.SuppressXFrameOptionsHeader = false;
});
```

†设置防伪 `Cookie` 属性使用的属性CookieBuilder类。

选项	描述
Cookie	确定用于创建防伪 cookie 的设置。
FormFieldName	防伪系统用于呈现防伪令牌在视图中的隐藏的窗体字段的名称。
HeaderName	防伪系统使用的标头的名称。如果 `null`，系统会认为只有窗体数据。
SuppressXFrameOptionsHeader	指定是否禁止显示生成 `X-Frame-Options` 标头。默认情况下，值为"SAMEORIGIN"生成标头。默认为 `false`。

```
services.AddAntiforgery(options =>
{
    options.CookieDomain = "contoso.com";
    options.CookieName = "X-CSRF-TOKEN-COOKIENAME";
    options.CookiePath = "Path";
    options.FormFieldName = "AntiforgeryFieldname";
    options.HeaderName = "X-CSRF-TOKEN-HEADERNAME";
    options.RequireSsl = false;
    options.SuppressXFrameOptionsHeader = false;
});
```

选项	描述
Cookie	确定用于创建防伪 cookie 的设置。
CookieDomain	Cookie 的域。默认为 `null`。此属性已过时，将在未来版本中删除。建议的替代项是 Cookie.Domain。
CookieName	Cookie 的名称。如果未设置，则系统生成唯一的名称开头DefaultCookiePrefix (".AspNetCore.Antiforgery.") 下。此属性已过时，将在未来版本中删除。建议的替代项是 Cookie.Name。
CookiePath	设置 cookie 的路径。此属性已过时，将在未来版本中删除。建议的替代项是 Cookie.Path。
FormFieldName	防伪系统用于呈现防伪令牌在视图中的隐藏的窗体字段的名称。
HeaderName	防伪系统使用的标头的名称。如果 `null`，系统会认为只有窗体数据。
RequireSsl	指定是否由防伪系统需要 HTTPS。如果 `true`，非 HTTPS 发出的请求失败。默认为 `false`。此属性已过时，将在未来版本中删除。建议的替代项是设置 Cookie.SecurePolicy。
SuppressXFrameOptionsHeader	指定是否禁止显示生成 `X-Frame-Options` 标头。默认情况下，值为"SAMEORIGIN"生成标头。默认为 `false`。

有关详细信息，请参阅CookieAuthenticationOptions。

配置与 IAntiforgery 防伪功能

IAntiforgery提供 API 来配置防伪功能。 `IAntiforgery` 可以在请求 `Configure` 方法的 `Startup` 类。以下示例使用中间件应用程序的主页上生成防伪令牌并将其在响应中发送作为 cookie（使用本主题后面所述的默认 Angular 命名约定）：

```
public void Configure(IApplicationBuilder app, IAntiforgery antiforgery)
{
    app.Use(next => context =>
    {
        string path = context.Request.Path.Value;

        if (
            string.Equals(path, "/", StringComparison.OrdinalIgnoreCase) ||
            string.Equals(path, "/index.html", StringComparison.OrdinalIgnoreCase))
        {
            // The request token can be sent as a JavaScript-readable cookie,
            // and Angular uses it by default.
            var tokens = antiforgery.GetAndStoreTokens(context);
            context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken,
                new CookieOptions() { HttpOnly = false });
        }

        return next(context);
    });
}
```

需要防伪验证

ValidateAntiForgeryToken是操作筛选器可应用到单个操作，在控制器或全局范围内。除非请求包含有效的防伪令牌，将阻止对已应用此筛选器的操作发出的请求。

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> RemoveLogin(RemoveLoginViewModel account)
{
    ManageMessageId? message = ManageMessageId.Error;
    var user = await GetCurrentUserAsync();

    if (user != null)
    {
        var result =
            await _userManager.RemoveLoginAsync(
                user, account.LoginProvider, account.ProviderKey);

        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user, isPersistent: false);
            message = ManageMessageId.RemoveLoginSuccess;
        }
    }

    return RedirectToAction(nameof(ManageLogins), new { Message = message });
}
```

`ValidateAntiForgeryToken` 属性需要对操作方法请求修饰，包括 HTTP GET 请求令牌。如果 `ValidateAntiForgeryToken` 特性应用于应用程序的控制器，它可以替换 `IgnoreAntiforgeryToken` 属性。

> **NOTE**
>
> ASP.NET Core 不支持自动将 antiforgery 令牌添加到 GET 请求。

自动验证防伪令牌仅不安全的 HTTP 方法

ASP.NET Core 应用不生成 antiforgery 令牌进行安全的 HTTP 方法（GET、HEAD、选项和跟踪）。而不是广泛应用 `ValidateAntiForgeryToken` 属性，然后将其与重写 `IgnoreAntiforgeryToken` 属性，AutoValidateAntiforgeryToken属性可用。此属性工作原理相同于 `ValidateAntiForgeryToken` 属性，只不过它不需要使用以下 HTTP 方法发出的请求令牌：

- GET
- HEAD
- 选项
- TRACE

我们建议使用 `AutoValidateAntiforgeryToken` 广泛用于非 API 方案。这可确保默认被保护 POST 操作。替代方法是忽略防伪令牌，默认情况下，除非 `ValidateAntiForgeryToken` 应用于各个操作方法。它更有可能在此方案中保留的 POST 操作方法不受保护错误地使应用程序容易受到 CSRF 攻击。所有Post请求都应发送防伪令牌。

Api 没有发送令牌的非 cookie 一部分的自动机制。实现可能取决于客户端代码实现。一些示例如下所示：

类级别的示例：

```
[Authorize]
[AutoValidateAntiforgeryToken]
public class ManageController : Controller
{
```

全局示例：

```
services.AddMvc(options =>
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute()));
```

重写全局或控制器防伪特性

IgnoreAntiforgeryToken使用筛选器不再需要给定操作（或控制器）的防伪令牌。当应用时，此筛选器将重写 `ValidateAntiForgeryToken` 和 `AutoValidateAntiforgeryToken`（全局或控制器上）在较高级别指定的筛选器。

```
[Authorize]
[AutoValidateAntiforgeryToken]
public class ManageController : Controller
{
    [HttpPost]
    [IgnoreAntiforgeryToken]
    public async Task<IActionResult> DoSomethingSafe(SomeViewModel model)
    {
        // no antiforgery token required
    }
}
```

刷新令牌的身份验证后

用户进行身份验证通过将用户重定向到一个视图或 Razor 页面页后，应刷新令牌。

JavaScript、AJAX 和 Spa

在传统的基于 HTML 的应用中，防伪令牌将传递给使用隐藏的表单域的服务器。在基于 JavaScript 的新型应用程序和 Spa 中，以编程方式进行很多请求。这些 AJAX 请求可以使用其他技术（如请求标头或 cookie）来发送令牌。

如果使用 cookie 来存储身份验证令牌，并在服务器上的 API 请求进行身份验证，CSRF 是一个潜在的问题。如果本地存储用于存储令牌，因为从本地存储的值不会自动发送到每个请求的服务器可能会缓解 CSRF 的漏洞。因此，使用本地存储来存储客户端和发送令牌作为请求标头是建议的方法上的防伪令牌。

JavaScript

使用 JavaScript 与视图，该令牌可以创建使用从视图中的服务。注入Microsoft.AspNetCore.Antiforgery.IAntiforgery到视图并调用服务GetAndStoreTokens:

```
@{
    ViewData["Title"] = "AJAX Demo";
}
@inject Microsoft.AspNetCore.Antiforgery.IAntiforgery Xsrf
@functions{
    public string GetAntiXsrfRequestToken()
    {
        return Xsrf.GetAndStoreTokens(Context).RequestToken;
    }
}

<input type="hidden" id="RequestVerificationToken"
       name="RequestVerificationToken" value="@GetAntiXsrfRequestToken()">

<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<div class="row">
    <p><input type="button" id="antiforgery" value="Antiforgery"></p>
    <script>
        var xhttp = new XMLHttpRequest();
        xhttp.onreadystatechange = function() {
            if (xhttp.readyState == XMLHttpRequest.DONE) {
                if (xhttp.status == 200) {
                    alert(xhttp.responseText);
                } else {
                    alert('There was an error processing the AJAX request.');
                }
            }
        };

        document.addEventListener('DOMContentLoaded', function() {
            document.getElementById("antiforgery").onclick = function () {
                xhttp.open('POST', '@Url.Action("Antiforgery", "Home")', true);
                xhttp.setRequestHeader("RequestVerificationToken",
                    document.getElementById('RequestVerificationToken').value);
                xhttp.send();
            }
        });
    </script>
</div>
```

此方法无需直接处理从服务器设置 cookie 或从客户端读取它们。

前面的示例中使用 JavaScript 为 POST 的 AJAX 标头中读取的隐藏的字段值。

JavaScript 还可以访问 cookie 中的令牌和 cookie 的内容使用令牌的值创建的标头。

```
context.Response.Cookies.Append("CSRF-TOKEN", tokens.RequestToken,
    new Microsoft.AspNetCore.Http.CookieOptions { HttpOnly = false });
```

假设脚本请求调用的标头中发送令牌 `X-CSRF-TOKEN`，配置要寻找的防伪服务 `X-CSRF-TOKEN` 标头：

```
services.AddAntiforgery(options => options.HeaderName = "X-CSRF-TOKEN");
```

以下示例使用 JavaScript 来发出 AJAX 请求与相应的标头：

```
function getCookie(cname) {
    var name = cname + "=";
    var decodedCookie = decodeURIComponent(document.cookie);
    var ca = decodedCookie.split(';');
    for(var i = 0; i <ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0) == ' ') {
            c = c.substring(1);
        }
        if (c.indexOf(name) == 0) {
            return c.substring(name.length, c.length);
        }
    }
    return "";
}

var csrfToken = getCookie("CSRF-TOKEN");

var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (xhttp.readyState == XMLHttpRequest.DONE) {
        if (xhttp.status == 200) {
            alert(xhttp.responseText);
        } else {
            alert('There was an error processing the AJAX request.');
        }
    }
};
xhttp.open('POST', '/api/password/changepassword', true);
xhttp.setRequestHeader("Content-type", "application/json");
xhttp.setRequestHeader("X-CSRF-TOKEN", csrfToken);
xhttp.send(JSON.stringify({ "newPassword": "ReallySecurePassword999$$$" }));
```

AngularJS

AngularJS 使用到地址 CSRF 的约定。如果服务器发送包含名称的 cookie `XSRF-TOKEN`，AngularJS `$http` 服务将 cookie 值添加到标头时它将请求发送到服务器。此过程是自动的。标头不需要显式设置。标头名称是 `X-XSRF-TOKEN`。服务器应检测此标头，并验证其内容。

有关 ASP.NET Core API 使用此约定：

- 将应用配置为提供的令牌在 cookie 中名为 `XSRF-TOKEN`。
- 查找名为标头将防伪服务配置 `X-XSRF-TOKEN`。

```
services.AddAntiforgery(options => options.HeaderName = "X-XSRF-TOKEN");
```

查看或下载示例代码（如何下载）

扩展防伪

IAntiForgeryAdditionalDataProvider类型允许开发人员能够通过往返过程中的每个标记的其他数据来扩展反 CSRF 系统的行为。GetAdditionalData每次调用方法生成的字段标记，并返回值嵌入生成的令牌中。实现器可以返回一个时间戳、nonce 或任何其他值，然后调用ValidateAdditionalData验证令牌时验证此数据。客户端的用户名已嵌入在生成的令牌中，因此无需包含此信息。如果令牌包含补充数据但不是 `IAntiForgeryAdditionalDataProvider` 是配置，并不验证补充数据。

其他资源

- CSRF上打开 Web 应用程序安全项目(OWASP)。
- 在 Web 场中托管 ASP.NET Core

防止在 ASP.NET Core 中的打开重定向攻击

2018/8/23 • • Edit Online

将重定向到的 URL，例如查询字符串或窗体数据的请求通过指定的 web 应用可以有可能被篡改，同时将用户重定向到恶意的外部 URL。此篡改称为开放重定向攻击。

每当应用程序逻辑将重定向到指定的 URL，必须验证重定向 URL 尚未被篡改。ASP.NET Core 具有内置功能来防止应用打开重定向（也称为打开重定向）攻击。

打开重定向攻击是什么？

Web 应用程序频繁地将用户重定向到登录页时他们访问要求进行身份验证的资源。重定向通常包括 `returnUrl` 查询字符串参数，以便用户可以返回到最初请求的 URL 后它们已成功登录。对用户进行身份验证后，它们重定向到他们具有最初请求的 URL。

因为请求的查询字符串中指定的目标 URL，则恶意用户可能篡改在查询字符串。被篡改的查询字符串可能允许将用户重定向到外部、恶意站点的站点。这一技术称为开放重定向（或重定向）攻击。

示例攻击

恶意用户可以开发目的是允许对用户的凭据和敏感信息的恶意用户访问的攻击。若要开始攻击，恶意用户使用户能够单击指向你网站的登录页面的 `returnUrl` 添加到的 URL 查询字符串值。例如，假设应用程序在 `contoso.com` 包括在登录页 `http://contoso.com/Account/LogOn?returnUrl=/Home/About` 。攻击遵循以下步骤：

1. 用户单击的恶意链接 `http://contoso.com/Account/LogOn?returnUrl=http://contoso1.com/Account/LogOn`（第二个 URL 为"contoso**1**.com"，而非"contoso.com"）。
2. 在用户成功登录。
3. 用户（按站点）重定向到 `http://contoso1.com/Account/LogOn`（看上去确实像真正的站点的恶意网站）。
4. 在用户再次登录（使恶意网站其凭据）和重定向回真正的站点。

用户可能认为其第一次尝试登录失败，且其第二次尝试成功。用户很可能仍不知道其凭据被透露。

除了登录页的某些站点提供重定向页或终结点。假设您的应用程序的页面包含打开的重定向，`/Home/Redirect`。攻击者可以创建，例如，将转到一封电子邮件中的链接 `[yoursite]/Home/Redirect?url=http://phishingsite.com/Home/Login`。典型的用户将查看该 URL，请参阅它开始于你的站点名称。信任的他们会单击此链接。打开重定向然后会将用户发送到网页仿冒站点，这看起来与你相同，并且用户很可能会登录到他们认为是你的站点。

打开重定向攻击保护

当开发 web 应用程序，将所有用户提供的数据视为不受信任。如果你的应用程序具有基于 URL 的内容将用户重定向的功能，请确保，这种重定向仅这样的是本地应用程序中（或已知的 URL，没有任何可能会在查询字符串中提供的 URL）。

LocalRedirect

使用 `LocalRedirect` 帮助器方法基 `Controller` 类：

```
public IActionResult SomeAction(string redirectUrl)
{
    return LocalRedirect(redirectUrl);
}
```

`LocalRedirect` 如果指定了非本地 URL，将引发异常。否则，它的行为就像 `Redirect` 方法。

IsLocalUrl

使用IsLocalUrl方法来测试之前将重定向 Url:

下面的示例演示如何检查 URL 重定向之前是否是本地。

```
private IActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction(nameof(HomeController.Index), "Home");
    }
}
```

`IsLocalUrl` 方法可防止用户无意中重定向到恶意站点。你可以记录非本地 URL 提供在本地 URL 的预期的情况下时提供的 URL 的详细信息。日志记录重定向 Url 可用于诊断重定向攻击。

防止跨站点脚本 (XSS) 在 ASP.NET Core

2018/10/10 • • Edit Online

作者：Rick Anderson

跨站点脚本 (XSS) 是一个安全漏洞，这会使攻击者将客户端脚本 (通常是 JavaScript) 放入网页。当其他用户加载攻击者的脚本将运行受影响的页面时，攻击者可以窃取 cookie 和会话令牌，更改通过 DOM 操作网页的内容或将浏览器重定向到另一页。当应用程序接受用户输入，并将其输出到页面而无需验证、编码或进行转义，通常会发生 XSS 漏洞。

保护防御 XSS 应用程序

在基本级别 XSS 的工作方式是以欺骗手段使应用程序插入到 `<script>` 标记插入到呈现的页面，或通过插入 `On*` 事件的元素。开发人员应使用以下的预防步骤以避免 XSS 引入其应用程序。

1. 永远不会，将不受信任的数据放入 HTML 输入，除非按照下面的步骤的其余部分。不受信任的数据是可能由攻击者、HTML 窗体输入、查询字符串、HTTP 标头，即使数据源自数据库，因为攻击者可能能够破坏您的数据库，即使它们不能破坏您的应用程序控制的任何数据。
2. 将 HTML 元素中的不受信任的数据之前，先确保它是 HTML 编码。HTML 编码如采用字符<并将其更改到之类的安全形式&l t;
3. 将不受信任的数据放入 HTML 特性之前请确保它是 HTML 编码。HTML 特性编码为 HTML 编码的超集，并将其他字符编码如"和。
4. 将不受信任的数据放入 JavaScript 之前将在运行时检索其内容的 HTML 元素中的数据。如果不可行，请确保数据是 JavaScript 编码。JavaScript 编码采用适用于 JavaScript 的危险的字符并将其替换为其十六进制表示，例如<将编码为 `\u003C` 。
5. 将不受信任的数据放入 URL 查询字符串之前请确保它进行 URL 编码。

使用 Razor 的 HTML 编码

Razor 引擎会自动在 MVC 中使用编码所有输出源自变量，除非您真正努力工作以防止其执行此操作。它使用 HTML 特性编码规则，每当你使用*@* 指令。为 HTML 特性编码是 HTML 编码，这意味着无需关注是否应该使用 HTML 编码或 HTML 特性编码的超集。您必须确保您仅使用在 HTML 上下文中，不是在尝试将直接插入 JavaScript 不受信任的输入时。标记帮助程序还将对标记参数中使用的输入进行编码。

执行以下 Razor 视图：

```
@{
        var untrustedInput = "<\"123\">";
    }

    @untrustedInput
```

此视图输出的内容*untrustedInput*变量。此变量包含即 XSS 攻击中使用的一些字符<，"和>。检查源显示呈现的输出编码为：

```
&lt;&quot;123&quot;&gt;
```

> **WARNING**
> ASP.NET Core MVC 提供 `HtmlString` 在输出时不自动编码的类。这应永远不会用作与不受信任的输入结合使用这将公开 XSS 漏洞。

使用 Razor 的 JavaScript 编码

可能的有时你想要将值插入到 JavaScript 来处理在视图中。有两种方法可以实现此目的。插入值的最安全方法是将值放在标记的数据的属性中，并在 JavaScript 中检索它。例如：

```
@{
        var untrustedInput = "<\"123\">";
    }

    <div
        id="injectedData"
        data-untrustedinput="@untrustedInput" />

    <script>
      var injectedData = document.getElementById("injectedData");

      // All clients
      var clientSideUntrustedInputOldStyle =
          injectedData.getAttribute("data-untrustedinput");

      // HTML 5 clients only
      var clientSideUntrustedInputHtml5 =
          injectedData.dataset.untrustedinput;

      document.write(clientSideUntrustedInputOldStyle);
      document.write("<br />")
      document.write(clientSideUntrustedInputHtml5);
    </script>
```

这将生成以下 HTML

```
<div
     id="injectedData"
     data-untrustedinput="&lt;&quot;123&quot;&gt;" />

   <script>
     var injectedData = document.getElementById("injectedData");

     var clientSideUntrustedInputOldStyle =
         injectedData.getAttribute("data-untrustedinput");

     var clientSideUntrustedInputHtml5 =
         injectedData.dataset.untrustedinput;

     document.write(clientSideUntrustedInputOldStyle);
     document.write("<br />")
     document.write(clientSideUntrustedInputHtml5);
   </script>
```

其中，运行时，将呈现以下：

```
<"123">
   <"123">
```

您还可以直接调用 JavaScript 编码器：

```
@using System.Text.Encodings.Web;
   @inject JavaScriptEncoder encoder;

   @{
       var untrustedInput = "<\"123\">";
   }

   <script>
       document.write("@encoder.Encode(untrustedInput)");
   </script>
```

这将按如下所示呈现在浏览器中：

```
<script>
       document.write("\u003C\u0022123\u0022\u003E");
   </script>
```

WARNING

不串联不受信任的输入，在 JavaScript 中创建 DOM 元素。应使用 `createElement()` 并将属性值分配适当如 `node.TextContent=`，或使用 `element.SetAttribute()` / `element[attribute]=` 否则向基于 DOM 的 XSS 公开自己。

访问代码中的编码器

HTML、JavaScript 和 URL 编码器可供你的代码通过两种方式，您可以将它们通过注入依赖关系注入也可以使用中包含的默认编码器 `System.Text.Encodings.Web` 命名空间。如果你使用默认编码器，然后对应用的任何字符范围来为安全处理不会生效-默认编码器使用可能的最安全的编码规则。

若要使用通过 DI 在构造函数应采取的可配置编码器*HtmlEncoder*，*JavaScriptEncoder*并*UrlEncoder*作为适当的参数。例如，

```
public class HomeController : Controller
   {
       HtmlEncoder _htmlEncoder;
       JavaScriptEncoder _javaScriptEncoder;
       UrlEncoder _urlEncoder;

       public HomeController(HtmlEncoder htmlEncoder,
                             JavaScriptEncoder javascriptEncoder,
                             UrlEncoder urlEncoder)
       {
           _htmlEncoder = htmlEncoder;
           _javaScriptEncoder = javascriptEncoder;
           _urlEncoder = urlEncoder;
       }
   }
```

编码的 URL 参数

如果你想要生成的 URL 查询字符串以不受信任的输入作为值使用 `UrlEncoder` 对值进行编码。例如，应用于对象的

```
var example = "\"Quoted Value with spaces and &\"";
   var encodedValue = _urlEncoder.Encode(example);
```

编码 encodedValue 后变量将包含 `%22Quoted%20Value%20with%20spaces%20and%20%26%22` 。空格、引号、标点符号和其他不安全字符百分比编码为其十六进制值，例如空格字符将变成 %20。

> **WARNING**
> 不要使用不受信任的输入的 URL 路径的一部分。始终将不受信任的输入传递作为查询字符串值。

自定义编码器

默认情况下编码器使用限制为基本拉丁语 Unicode 范围的安全列表，并为其等效的字符代码在该范围以外的所有字符进行都编码。此行为还会影响 Razor TagHelper 和 HtmlHelper 呈现，因为它将使用编码器输出字符串。

这背后的原因是为了防止未知或将来的浏览器 bug（上一个浏览器 bug 具有掣肘：分析基于非英语字符的处理）。如果您的网站重度使用的非拉丁字符，例如中文，西里尔文或其他人这是可能不希望的行为。

你可以自定义编码器安全列表，包括 Unicode 范围适合于你的应用程序启动过程中，在 `ConfigureServices()` 。

例如，使用默认配置可能会使用 Razor HtmlHelper 如下所示;

```
<p>This link text is in Chinese: @Html.ActionLink("汉语/漢語", "Index")</p>
```

当你查看网页的源时将看到已呈现，如下所示，使用中文文本编码;

```
<p>This link text is in Chinese: <a href="/">&#x6C49;&#x8BED;/&#x6F22;&#x8A9E;</a></p>
```

若要扩大范围的字符视为安全编码器插入以下行 `ConfigureServices()` 中的方法 `startup.cs` ;

```
services.AddSingleton<HtmlEncoder>(
     HtmlEncoder.Create(allowedRanges: new[] { UnicodeRanges.BasicLatin,
                                               UnicodeRanges.CjkUnifiedIdeographs }));
```

此示例将加宽安全列表，以包括 Unicode 范围 CjkUnifiedIdeographs。现在将变为呈现的输出

```
<p>This link text is in Chinese: <a href="/">汉语/漢語</a></p>
```

安全列表范围被指定为 Unicode 编码图表，不语言。Unicode 标准具有一系列代码图表可用于查找包含你字符的图表。每个编码器，Html、JavaScript 和 Url，必须单独配置。

> **NOTE**
> 自定义的安全列表只会影响源自通过 DI 的编码器。如果你直接访问通过编码器 `System.Text.Encodings.Web.*Encoder.Default` 然后默认情况下，基本拉丁语将使用仅安全列表。

编码 take 应放置位置？

常规接受的做法是编码发生在输出时和编码的值应永远不会存储在数据库中。编码在输出时，可更改数据，例如，使用从为查询字符串值的 HTML。它还使您可以轻松地搜索你的数据而无需对值进行编码搜索之前，并允许您充分任何的利用更改或对编码器进行 bug 修复。

作为 XSS 防护技术验证

验证过程可能在限制的 XSS 攻击的有用工具。例如，包含字符 0-9 的数字字符串将不会触发 XSS 攻击。接受用

户输入中的 HTML 时，验证变得更加复杂。分析 HTML 输入是困难的即使不是不可能。Markdown，结合一个分析器，它去除嵌入式的 HTML 是用于接收丰富输入比较安全的选项。永远不会依赖于单独的验证。始终对不受信任的输入输出，无论执行何种验证或清理之前进行编码。

启用 ASP.NET Core 中的跨域请求 (CORS)

2018/11/28 • • Edit Online

作者：Mike Wasson、Shayne Boyer 和 Tom Dykstra

浏览器安全性以防止网页从发出请求到不同的域的 web 页面提供服务。此限制称为*同域策略*。同域策略可阻止恶意站点读取另一个站点中的敏感数据。有时，你可能想要允许其他站点对您的应用程序进行跨域请求。

跨域资源共享(CORS) 是一种 W3C 标准，允许服务器放宽同源策略。使用 CORS，服务器可以在显式允许某些跨域请求时拒绝其他跨域请求。CORS 是更安全、更灵活比早期技术，如JSONP。本主题演示如何在 ASP.NET Core 应用中启用 CORS。

相同的原点

如果它们具有相同方案、主机和端口，两个 Url 将具有相同的原点 (RFC 6454)。

以下两个 Url 具有相同的原点：

- `https://example.com/foo.html`
- `https://example.com/bar.html`

这些 Url 具有不同的源，比以前的两个 Url:

- `https://example.net` – 不同的域
- `https://www.example.com/foo.html` – 不同的子域
- `http://example.com/foo.html` – 不同的方案
- `https://example.com:9000/foo.html` – 不同的端口

> **NOTE**
> 比较来源时，Internet Explorer 不会考虑该端口。

注册的 CORS 服务

引用Microsoft.AspNetCore.App 元包或添加到的包引用Microsoft.AspNetCore.Cors包。

引用Microsoft.AspNetCore.All 元包或添加到的包引用Microsoft.AspNetCore.Cors包。

添加到包引用Microsoft.AspNetCore.Cors包。

调用AddCors在 `Startup.ConfigureServices` 将 CORS 服务添加到应用程序的服务容器：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors();
}
```

启用 CORS

注册之后的 CORS 服务，使用以下方法之一在 ASP.NET Core 应用中启用 CORS:

- CORS 中间件–全局到通过中间件应用程序的应用的 CORS 策略。

- 在 MVC 中的 CORS –每个操作或每个控制器应用 CORS 策略。不会使用 CORS 中间件。

启用 CORS 的 CORS 中间件

CORS 中间件处理向应用程序的跨域请求。若要在请求处理管道中启用 CORS 中间件，请调用UseCors中的扩展方法 `Startup.Configure` 。

CORS 中间件必须位于之前定义的任何终结点应用程序中你想要支持跨域请求 (例如，在对调用之前 `UseMvc` MVC/Razor 页中间件)。

一个*跨域策略*添加 CORS 中间件使用时可以指定CorsPolicyBuilder类。有两种方法用于定义 CORS 策略：

- 调用 `UseCors` lambda:

  ```
  public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
  {
      loggerFactory.AddConsole();

      if (env.IsDevelopment())
      {
          app.UseDeveloperExceptionPage();
      }

      // Shows UseCors with CorsPolicyBuilder.
      app.UseCors(builder =>
         builder.WithOrigins("http://example.com"));

      app.Run(async (context) =>
      {
          await context.Response.WriteAsync("Hello World!");
      });
  ```

  Lambda 采用CorsPolicyBuilder对象。配置选项，如 `WithOrigins` ，本主题后面所述。在前面的示例中，该策略允许跨域请求从 `https://example.com` 和任何其他来源。

  不含尾部斜杠，必须指定的 URL (`/`)。如果 URL 以终止 `/` ，则比较操作返回 `false` 并返回无标头。

  `CorsPolicyBuilder` 具有 fluent API，因此您可以将方法调用链接：

  ```
  app.UseCors(builder =>
      builder.WithOrigins("http://example.com")
             .AllowAnyHeader()
      );
  ```

- 定义一个或多个命名的 CORS 策略并按名称在运行时选择的策略。下面的示例添加名为的用户定义 CORS 策略*AllowSpecificOrigin*。若要选择的策略，将名称传递给 `UseCors` ：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy("AllowSpecificOrigin",
            builder => builder.WithOrigins("http://example.com"));
    });
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    // Shows UseCors with named policy.
    app.UseCors("AllowSpecificOrigin");

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

启用在 MVC 中的 CORS

或者，可以使用 MVC 应用特定 CORS 策略针对每个操作或控制器。在使用 MVC 启用 CORS，使用了已注册的 CORS 服务。不会使用 CORS 中间件。

每个操作

若要指定特定操作的 CORS 策略，添加 [EnableCors] 属性为该操作。指定策略名称。

```
[HttpGet]
[EnableCors("AllowSpecificOrigin")]
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}
```

每个控制器

若要指定特定控制器的 CORS 策略，添加 [EnableCors] 属性到控制器类。指定策略名称。

```
[Route("api/[controller]")]
[EnableCors("AllowSpecificOrigin")]
public class ValuesController : ControllerBase
```

优先顺序是：

1. action
2. 控制器

禁用 CORS

若要禁用 CORS 的控制器或操作，请使用 [DisableCors] 属性：

```
[HttpGet("{id}")]
[DisableCors]
public string Get(int id)
{
    return "value";
}
```

CORS 策略选项

本部分介绍您可以在 CORS 策略设置的各种选项。AddPolicy方法调用 `Startup.ConfigureServices` 。

- 设置允许的来源
- 设置允许的 HTTP 方法
- 设置允许的请求标头
- 设置公开的响应标头
- 跨域请求中的凭据
- 将预检过期时间设置

有关一些选项，可能会有帮助读取如何 CORS 工作部分第一次。

设置允许的来源

ASP.NET Core MVC 中的 CORS 中间件具有几种方法来指定允许的来源：

- WithOrigins – 允许指定一个或多个 Url。URL 可能包括方案、主机名和端口不包含任何路径信息。例如 `https://example.com` 。不含尾部斜杠，必须指定的 URL (`/`)。

  ```
  options.AddPolicy("AllowSpecificOrigins",
      builder =>
      {
          builder.WithOrigins("http://example.com",
              "http://www.contoso.com");
      });
  ```

- AllowAnyOrigin – 允许来自任何方案使用所有来源的 CORS 请求 (`http` 或 `https`)。

  ```
  options.AddPolicy("AllowAllOrigins",
      builder =>
      {
          builder.AllowAnyOrigin();
      });
  ```

  允许任何来源的请求之前，请仔细考虑。允许任何来源的请求意味着*的任何网站*可以对您的应用程序进行跨域请求。

  > **NOTE**
  > 指定 `AllowAnyOrigin` 和 `AllowCredentials` 是不安全的配置，可能会导致跨站点请求伪造。使用这两种方法配置应用程序时，CORS 服务返回了无效的 CORS 响应。

  > **NOTE**
  > 指定 `AllowAnyOrigin` 和 `AllowCredentials` 是不安全的配置，可能会导致跨站点请求伪造。请考虑指定确切的来源列表，如果客户端必须先授权本身访问服务器资源。

此设置会影响的预检请求和 `Access-Control-Allow-Origin` 标头。有关详细信息，请参阅预检请求部分。

- SetIsOriginAllowedToAllowWildcardSubdomains – 集IsOriginAllowed策略允许来源以匹配配置的通配符域，如果允许原点在评估时的函数的属性。

```
options.AddPolicy("AllowSubdomain",
    builder =>
    {
        builder.SetIsOriginAllowedToAllowWildcardSubdomains();
    });
```

设置允许的 HTTP 方法

若要允许所有 HTTP 方法，调用AllowAnyMethod:

```
options.AddPolicy("AllowAllMethods",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowAnyMethod();
    });
```

此设置会影响的预检请求和 `Access-Control-Allow-Methods` 标头。有关详细信息，请参阅预检请求部分。

设置允许的请求标头

若要允许特定标头发送在 CORS 请求中，调用*创作请求标头*，调用WithHeaders并指定允许的标头：

```
options.AddPolicy("AllowHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .WithHeaders(HeaderNames.ContentType, "x-custom-header");
    });
```

若要允许所有作者的请求标头调用AllowAnyHeader:

```
options.AddPolicy("AllowAllHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowAnyHeader();
    });
```

此设置会影响的预检请求和 `Access-Control-Request-Headers` 标头。有关详细信息，请参阅预检请求部分。

CORS 中间件策略匹配所指定的特定标头 `WithHeaders` 时，才可以发送标头 `Access-Control-Request-Headers` 与中所述的标头完全匹配 `WithHeaders` 。

例如，假设配置，如下所示的应用：

```
app.UseCors(policy => policy.WithHeaders(HeaderNames.CacheControl));
```

CORS 中间件拒绝包含以下请求标头的预检请求，因为 `Content-Language` (HeaderNames.ContentLanguage) 中未列出 `WithHeaders` :

```
Access-Control-Request-Headers: Cache-Control, Content-Language
```

应用程序返回*200 确定*响应但不会发送回的 CORS 标头。因此，在浏览器不会尝试执行跨域请求。

CORS 中间件始终允许在四个标头 `Access-Control-Request-Headers` 发送而不考虑 CorsPolicy.Headers 中配置的值。此列表的标头包括：

- `Accept`
- `Accept-Language`
- `Content-Language`
- `Origin`

例如，假设配置，如下所示的应用：

```
app.UseCors(policy => policy.WithHeaders(HeaderNames.CacheControl));
```

CORS 中间件成功响应以下请求标头的预检请求因为 `Content-Language` 始终是已列入允许列表：

```
Access-Control-Request-Headers: Cache-Control, Content-Language
```

设置公开的响应标头

默认情况下，在浏览器不会公开所有向应用程序的响应标头。有关详细信息，请参阅W3C 跨域资源共享（术语）：简单的响应标头。

默认情况下可用的响应标头是：

- `Cache-Control`
- `Content-Language`
- `Content-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

CORS 规范调用这些标头*简单响应标头*。若要使其他标头可用于应用程序，请调用WithExposedHeaders:

```
options.AddPolicy("ExposeResponseHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .WithExposedHeaders("x-custom-header");
    });
```

跨域请求中的凭据

凭据要求特殊处理 CORS 请求中。默认情况下，在浏览器不会发送与跨域请求的凭据。凭据包括 cookie 和 HTTP 身份验证方案。若要将发送具有跨源请求的凭据，客户端必须设置 `XMLHttpRequest.withCredentials` 到 `true`。

使用 `XMLHttpRequest` 直接：

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'https://www.example.com/api/test');
xhr.withCredentials = true;
```

在 jQuery 中：

```
$.ajax({
  type: 'get',
  url: 'https://www.example.com/home',
  xhrFields: {
    withCredentials: true
}
```

此外，服务器必须允许凭据。若要允许跨域凭据，调用AllowCredentials:

```
options.AddPolicy("AllowCredentials",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowCredentials();
    });
```

HTTP 响应包括 `Access-Control-Allow-Credentials` 标头，服务器允许跨域请求凭据将告诉浏览器。

如果浏览器发送凭据，但响应不包含有效 `Access-Control-Allow-Credentials` 标头，在浏览器不会公开的响应应用程序中，并且跨域请求失败。

允许跨域凭据时要小心。在另一个域的网站可以向应用而无需在用户不知情的用户的名义发送登录的用户的凭据。

CORS 规范还会说明该设置到来源 `"*"`（所有来源）是无效的如果 `Access-Control-Allow-Credentials` 标头。

预检请求

对于某些 CORS 请求，在浏览器发出实际请求之前发送其他请求。调用此请求*预检请求*。在浏览器可以跳过预检请求，如果满足以下条件：

- 请求方法是 GET、HEAD 或 POST。
- 应用程序不会设置以外的其他请求标头 `Accept`， `Accept-Language`， `Content-Language`， `Content-Type`，或 `Last-Event-ID`。
- `Content-Type` 标头，如果设置，具有以下值之一：
  - `application/x-www-form-urlencoded`
  - `multipart/form-data`
  - `text/plain`

在请求标头的规则集的客户端请求适用于通过调用来设置应用程序的标头 `setRequestHeader` 上 `XMLHttpRequest` 对象。CORS 规范调用这些标头*创作请求标头*。该规则不能应用于标头设置可以在浏览器，如 `User-Agent`， `Host`，或 `Content-Length`。

下面是预检请求的示例：

```
OPTIONS https://myservice.azurewebsites.net/api/test HTTP/1.1
Accept: */*
Origin: https://myclient.azurewebsites.net
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: accept, x-my-custom-header
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
Content-Length: 0
```

预检请求使用 HTTP OPTIONS 方法。它包括两个特殊的标头：

- `Access-Control-Request-Method`：将用作实际请求的 HTTP 方法。
- `Access-Control-Request-Headers`：设置实际请求中应用的请求标头的列表。如前面所述，这不包括标头的浏览器设置，如 `User-Agent`。

CORS 预检请求可能包括 `Access-Control-Request-Headers` 标头，指示服务器与实际请求一起发送的标头。

若要允许的特定标头，请调用WithHeaders:

```
options.AddPolicy("AllowHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .WithHeaders(HeaderNames.ContentType, "x-custom-header");
    });
```

若要允许所有作者的请求标头调用AllowAnyHeader:

```
options.AddPolicy("AllowAllHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowAnyHeader();
    });
```

浏览器不是在设置方式完全一致 `Access-Control-Request-Headers`。如果将标头设置到的任何内容以外 `"*"`(或使用AllowAnyHeader)，至少应包含 `Accept`， `Content-Type`，和 `Origin`，以及你想要支持任何自定义标头。

下面是预检请求（假设服务器允许请求）的示例响应：

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 0
Access-Control-Allow-Origin: https://myclient.azurewebsites.net
Access-Control-Allow-Headers: x-my-custom-header
Access-Control-Allow-Methods: PUT
Date: Wed, 20 May 2015 06:33:22 GMT
```

响应包括 `Access-Control-Allow-Methods` 标头，其中列出了允许的方法和（可选）`Access-Control-Allow-Headers` 标头，其中列出了允许的标头。如果预检请求成功，则浏览器发送实际请求。

如果预检请求被拒绝，该应用程序将返回*200 确定*响应但不会发送回的 CORS 标头。因此，在浏览器不会尝试执行跨域请求。

将预检过期时间设置

`Access-Control-Max-Age` 标头指定可以缓存预检请求的响应的时间。若要设置此标头，请调

用SetPreflightMaxAge:

```
options.AddPolicy("SetPreflightExpiration",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
    });
```

CORS 的工作原理

本部分介绍在 CORS 请求的 HTTP 消息级别中会发生什么情况。请务必了解，以便可以正确配置和调试时出现意外的行为的 CORS 策略 CORS 的工作原理。

CORS 规范引入了几个新的 HTTP 标头启用跨域请求。如果浏览器支持 CORS，则将设置自动跨域请求这些标头。自定义 JavaScript 代码不需要启用 CORS。

下面是跨域请求的示例。`Origin`标头提供发出请求的站点的域：

```
GET https://myservice.azurewebsites.net/api/test HTTP/1.1
Referer: https://myclient.azurewebsites.net/
Accept: */*
Accept-Language: en-US
Origin: https://myclient.azurewebsites.net
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
```

如果服务器允许该请求，它会设置`Access-Control-Allow-Origin`在响应中的标头。此标头的值或者与匹配`Origin`请求标头或为通配符值`"*"`，允许任何源的含义：

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: https://myclient.azurewebsites.net
Date: Wed, 20 May 2015 06:27:30 GMT
Content-Length: 12

Test message
```

如果响应不包括`Access-Control-Allow-Origin`标头，跨域请求会失败。具体而言，在浏览器不允许该请求。即使服务器返回成功的响应，请在浏览器不使响应可供客户端应用程序。

其他资源

- 跨域资源共享 (CORS)

与 ASP.NET 和 ASP.NET Core 共享在应用之间的 cookie

2018/10/30 • • Edit Online

通过Rick Anderson和Luke Latham

网站通常包含单独的 web 应用程序协同工作。若要提供单一登录 (SSO) 体验，站点内的 web 应用必须共享身份验证 cookie。若要支持这种情况下，数据保护堆栈允许在共享 Katana cookie 身份验证和 ASP.NET Core cookie 身份验证票证。

查看或下载示例代码(如何下载)

此示例演示了在使用 cookie 身份验证的三个应用之间共享 cookie:

- ASP.NET Core 2.0 Razor 页应用而无需使用ASP.NET Core 标识
- 使用 ASP.NET Core 标识的 ASP.NET Core 2.0 MVC 应用程序
- 使用 ASP.NET 标识的 ASP.NET Framework 4.6.1 MVC 应用程序

在下面的示例:

- 身份验证 cookie 名称设置为公用值的 `.AspNet.SharedCookie`。
- `AuthenticationType` 设置为 `Identity.Application` 显式或默认情况下。
- 常见的应用名称用于启用数据保护系统共享数据保护密钥 (`SharedCookieApp`)。
- `Identity.Application` 使用作为身份验证方案。使用任何方案，必须一致地使用它*内部和跨*共享的 cookie 应用作为默认方案或通过显式设置。加密和解密 cookie，因此，必须在应用之间使用一致的方案时，使用方案。
- 一种常见数据保护密钥使用存储位置。示例应用程序使用名为的文件夹*KeyRing*要保留数据保护密钥的解决方案的根目录。
- 在 ASP.NET Core 应用中，PersistKeysToFileSystem用于设置的密钥存储位置。SetApplicationName用于配置公用的共享的应用名。
- 在.NET Framework 应用中，cookie 身份验证中间件使用实现DataProtectionProvider。`DataProtectionProvider` 提供对身份验证 cookie 有效负载数据进行加密和解密的数据保护服务。`DataProtectionProvider` 实例都独立于应用的其他部分使用的数据保护系统。
  - DataProtectionProvider.Create (System.IO.DirectoryInfo、操作<IDataProtectionBuilder >)接受DirectoryInfo指定用于数据保护密钥存储的位置。示例应用程序提供的路径*KeyRing*向文件夹 `DirectoryInfo`。DataProtectionBuilderExtensions.SetApplicationName设置的常见应用程序。
  - DataProtectionProvider需要Microsoft.AspNetCore.DataProtection.Extensions NuGet 包。若要对 ASP.NET Core 2.1 和更高版本的应用程序中获取此包，引用Microsoft.AspNetCore.App metapackage。当面向.NET Framework，添加到包引用 `Microsoft.AspNetCore.DataProtection.Extensions`。

共享 ASP.NET Core 应用之间的身份验证 cookie

当使用 ASP.NET Core标识:

在中 `ConfigureServices` 方法，请使用ConfigureApplicationCookie扩展方法，以设置 cookie 的数据保护服务。

```
services.AddDataProtection()
    .PersistKeysToFileSystem(GetKeyRingDirInfo())
    .SetApplicationName("SharedCookieApp");

services.ConfigureApplicationCookie(options => {
    options.Cookie.Name = ".AspNet.SharedCookie";
});
```

数据保护密钥和应用名称必须在应用之间共享。在示例应用中，`GetKeyRingDirInfo`返回到的公共密钥存储位置PersistKeysToFileSystem方法。使用SetApplicationName来配置常见的共享的应用名称（`SharedCookieApp`示例中）。有关详细信息，请参阅配置数据保护。

托管在子域之间共享 cookie 的应用，指定在一个公共域Cookie.Domain属性。若要在应用之间共享 cookie `contoso.com`，如`first_subdomain.contoso.com`并`second_subdomain.contoso.com`，指定`Cookie.Domain`作为`.contoso.com`：

```
options.Cookie.Domain = ".contoso.com";
```

请参阅*CookieAuthWithIdentity.Core*项目中示例代码(如何下载)。

在中`Configure`方法，请使用CookieAuthenticationOptions设置：

- Cookie 数据保护服务。
- `AuthenticationScheme`以匹配 ASP.NET 4.x。

```
app.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    options.Cookies.ApplicationCookie.AuthenticationScheme =
        "ApplicationCookie";

    var protectionProvider =
        DataProtectionProvider.Create(
            new DirectoryInfo(@"PATH_TO_KEY_RING_FOLDER"));

    options.Cookies.ApplicationCookie.DataProtectionProvider =
        protectionProvider;

    options.Cookies.ApplicationCookie.TicketDataFormat =
        new TicketDataFormat(protectionProvider.CreateProtector(
            "Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationMiddleware",
            "Cookies",
            "v2"));
});
```

当直接使用 cookie:

```
services.AddDataProtection()
    .PersistKeysToFileSystem(GetKeyRingDirInfo())
    .SetApplicationName("SharedCookieApp");

services.AddAuthentication("Identity.Application")
    .AddCookie("Identity.Application", options =>
    {
        options.Cookie.Name = ".AspNet.SharedCookie";
    });
```

数据保护密钥和应用名称必须在应用之间共享。在示例应用中，`GetKeyRingDirInfo`返回到的公共密钥存储位置PersistKeysToFileSystem方法。使用SetApplicationName来配置常见的共享的应用名称（`SharedCookieApp`示例

中)。有关详细信息，请参阅配置数据保护。

托管在子域之间共享 cookie 的应用，指定在一个公共域Cookie.Domain属性。若要在应用之间共享 cookie `contoso.com`，如 `first_subdomain.contoso.com` 并 `second_subdomain.contoso.com`，指定 `Cookie.Domain` 作为 `.contoso.com`：

```
options.Cookie.Domain = ".contoso.com";
```

请参阅*CookieAuth.Core*项目中示例代码(如何下载)。

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    DataProtectionProvider =
        DataProtectionProvider.Create(
            new DirectoryInfo(@"PATH_TO_KEY_RING_FOLDER"))
});
```

加密静态数据保护密钥

对于生产部署，配置 `DataProtectionProvider` 来加密静态使用 DPAPI 或 x509 证书的密钥。请参阅密钥静态加密有关详细信息。

```
services.AddDataProtection()
    .ProtectKeysWithCertificate("thumbprint");
```

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    DataProtectionProvider = DataProtectionProvider.Create(
        new DirectoryInfo(@"PATH_TO_KEY_RING"),
        configure =>
        {
            configure.ProtectKeysWithCertificate("thumbprint");
        })
});
```

共享身份验证 cookie 之间 ASP.NET 4.x 和 ASP.NET Core 应用

ASP.NET 4.x 应用程序使用 Katana cookie 身份验证中间件该对话框可以配置为生成与 ASP.NET Core cookie 身份验证中间件兼容的身份验证 cookie。这样，段落将大型站点的单个应用升级同时在站点之间提供顺畅的 SSO 体验。

当应用使用 Katana cookie 身份验证中间件时，它将调用 `UseCookieAuthentication` 在项目的*Startup.Auth.cs*文件。ASP.NET 4.x web 应用程序项目创建与 Visual Studio 2013 和更高版本默认情况下使用 Katana cookie 身份验证中间件。尽管 `UseCookieAuthentication` 已过时，不支持对于 ASP.NET Core 应用程序，调用 `UseCookieAuthentication` 在 ASP.NET 4.x 应用程序使用 Katana cookie 身份验证中间件是有效的。

ASP.NET 4.x 应用程序必须面向.NET Framework 4.5.1 或更高版本。否则，无法安装所需的 NuGet 包。

若要共享 ASP.NET 4.x 应用程序和 ASP.NET Core 应用程序之间的身份验证 cookie，如前所述，将 ASP.NET Core 应用配置，然后通过执行以下步骤配置 ASP.NET 4.x 应用程序：

1. 安装包Microsoft.Owin.Security.Interop到每个 ASP.NET 4.x 应用程序。
2. 在中*Startup.Auth.cs*，找到调用 `UseCookieAuthentication` 和对其进行修改，如下所示。更改要匹配使用 ASP.NET Core cookie 身份验证中间件的名称的 cookie 名称。提供的一个实例 `DataProtectionProvider` 初始

化为常见的数据保护密钥存储位置。请确保应用程序名称设置为使用共享 cookie 的所有应用的常见应用程序名称 `SharedCookieApp` 示例应用程序中。

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationType = "Identity.Application",
    CookieName = ".AspNet.SharedCookie",
    LoginPath = new PathString("/Account/Login"),
    Provider = new CookieAuthenticationProvider
    {
        OnValidateIdentity =
            SecurityStampValidator
                .OnValidateIdentity<ApplicationUserManager, ApplicationUser>(
                    validateInterval: TimeSpan.FromMinutes(30),
                    regenerateIdentity: (manager, user) =>
                        user.GenerateUserIdentityAsync(manager))
    },
    TicketDataFormat = new AspNetTicketDataFormat(
        new DataProtectorShim(
            DataProtectionProvider.Create(GetKeyRingDirInfo(),
                (builder) => { builder.SetApplicationName("SharedCookieApp"); })
            .CreateProtector(
                "Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationMiddleware",
                "Identity.Application",
                "v2"))),
    CookieManager = new ChunkingCookieManager()
});

// If not setting http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier and
// http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider,
// then set UniqueClaimTypeIdentifier to a claim that distinguishes unique users.
System.Web.Helpers.AntiForgeryConfig.UniqueClaimTypeIdentifier =
    "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name";
```

请参阅*CookieAuthWithIdentity.NETFramework*项目中示例代码(如何下载)。

身份验证类型时生成的用户标识，必须与中定义的类型匹配 `AuthenticationType` 设置与 `UseCookieAuthentication` 。

Models/IdentityModels.cs:

```
public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser> manager)
{
    // Note the authenticationType must match the one defined in
CookieAuthenticationOptions.AuthenticationType
    var userIdentity = await manager.CreateIdentityAsync(this, "Identity.Application");
    // Add custom user claims here
    return userIdentity;
}
```

使用常见的用户数据库

确认每个应用的标识系统指向同一个用户数据库。否则，标识系统将生成在运行时失败时它将尝试匹配对其数据库中的信息的身份验证 cookie 中的信息。

其他资源

在 Web 场中托管 ASP.NET Core

适用于 ASP.NET Core 的客户端 IP 安全列表

2018/10/30 • • Edit Online

通过Damien Bowden和Tom Dykstra

本文介绍三种方法可以在 ASP.NET Core 应用中实施 IP 安全列表（也称为允许列表）。您可以使用：

- 若要检查的每个请求的远程 IP 地址的中间件。
- 若要检查为特定控制器或操作方法的请求的远程 IP 地址的操作筛选器。
- Razor 页面筛选器，以检查 Razor 页的请求的远程 IP 地址。

示例应用程序说明了这两种方法。在每种情况下，包含已批准的客户端 IP 地址的字符串存储在一个应用程序设置。中间件或筛选器将字符串分析为列表，并检查远程 IP 是否在列表中。如果没有，则返回 HTTP 403 禁止访问状态代码。

查看或下载示例代码（如何下载）

安全列表

在配置列表*appsettings.json*文件。它是以分号分隔的列表，并且可以包含 IPv4 和 IPv6 地址。

```
{
  "AdminSafeList": "127.0.0.1;192.168.1.5;::1",
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

中间件

`Configure` 方法添加中间件，并向其构造函数参数中传递的安全列表的字符串。

```
public void Configure(
    IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory.AddNLog();

    app.UseStaticFiles();

    app.UseMiddleware<AdminSafeListMiddleware>(
        Configuration["AdminSafeList"]);
    app.UseMvc();
}
```

中间件将字符串分析为一个数组，并查找数组中的远程 IP 地址。如果找不到的远程 IP 地址，中间件返回 HTTP 401 禁止访问。对于 HTTP Get 请求跳过此验证过程。

```
public class AdminSafeListMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<AdminSafeListMiddleware> _logger;
    private readonly string _adminSafeList;

    public AdminSafeListMiddleware(
        RequestDelegate next,
        ILogger<AdminSafeListMiddleware> logger,
        string adminSafeList)
    {
        _adminSafeList = adminSafeList;
        _next = next;
        _logger = logger;
    }

    public async Task Invoke(HttpContext context)
    {
        if (context.Request.Method != "GET")
        {
            var remoteIp = context.Connection.RemoteIpAddress;
            _logger.LogDebug($"Request from Remote IP address: {remoteIp}");

            string[] ip = _adminSafeList.Split(';');

            var bytes = remoteIp.GetAddressBytes();
            var badIp = true;
            foreach (var address in ip)
            {
                var testIp = IPAddress.Parse(address);
                if(testIp.GetAddressBytes().SequenceEqual(bytes))
                {
                    badIp = false;
                    break;
                }
            }

            if(badIp)
            {
                _logger.LogInformation(
                    $"Forbidden Request from Remote IP address: {remoteIp}");
                context.Response.StatusCode = (int)HttpStatusCode.Forbidden;
                return;
            }
        }

        await _next.Invoke(context);

    }
}
```

操作筛选器

如果您希望仅为特定控制器或操作方法的安全列表，请使用操作筛选器。以下是一个示例：

```
using System.Linq;
using System.Net;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Authorization;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace ClientIpAspNetCore.Filters
{
    public class ClientIdCheckFilter : ActionFilterAttribute
    {
        private readonly ILogger _logger;
        private readonly string _safelist;

        public ClientIdCheckFilter
            (ILoggerFactory loggerFactory, IConfiguration configuration)
        {
            _logger = loggerFactory.CreateLogger("ClientIdCheckFilter");
            _safelist = configuration["AdminSafeList"];
        }

        public override void OnActionExecuting(ActionExecutingContext context)
        {
            _logger.LogInformation(
                $"Remote IpAddress: {context.HttpContext.Connection.RemoteIpAddress}");

            var remoteIp = context.HttpContext.Connection.RemoteIpAddress;
            _logger.LogDebug($"Request from Remote IP address: {remoteIp}");

            string[] ip = _safelist.Split(';');

            var bytes = remoteIp.GetAddressBytes();
            var badIp = true;
            foreach (var address in ip)
            {
                var testIp = IPAddress.Parse(address);
                if (testIp.GetAddressBytes().SequenceEqual(bytes))
                {
                    badIp = false;
                    break;
                }
            }

            if (badIp)
            {
                _logger.LogInformation(
                    $"Forbidden Request from Remote IP address: {remoteIp}");
                context.Result = new StatusCodeResult(401);
                return;
            }

            base.OnActionExecuting(context);
        }
    }
}
```

操作筛选器添加到服务容器。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<ClientIdCheckFilter>();

    services.AddMvc(options =>
    {
        options.Filters.Add
            (new ClientIdCheckPageFilter
                (_loggerFactory, Configuration));
    }).SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

然后可以在控制器或操作方法上使用筛选器。

```
[ServiceFilter(typeof(ClientIdCheckFilter))]
[HttpGet]
public IEnumerable<string> Get()
```

在示例应用中，筛选器应用于 `Get` 方法。因此，当您测试应用程序通过发送 `Get` API 请求，该属性就验证客户端 IP 地址。当测试通过使用任何其他 HTTP 方法调用的 API 时，中间件就验证客户端 IP。

Razor 页面筛选器

如果你想为 Razor 页面应用的安全列表，使用 Razor 页面筛选器。以下是一个示例：

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System;
using System.Linq;
using System.Net;

namespace ClientIpAspNetCore
{
    public class ClientIdCheckPageFilter : IPageFilter
    {
        private readonly ILogger _logger;
        private readonly string _safelist;

        public ClientIdCheckPageFilter
            (ILoggerFactory loggerFactory, IConfiguration configuration)
        {
            _logger = loggerFactory.CreateLogger("ClientIdCheckPageFilter");
            _safelist = configuration["AdminSafeList"];
        }

        public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
        {
            _logger.LogInformation(
                $"Remote IpAddress: {context.HttpContext.Connection.RemoteIpAddress}");

            var remoteIp = context.HttpContext.Connection.RemoteIpAddress;
            _logger.LogDebug($"Request from Remote IP address: {remoteIp}");

            string[] ip = _safelist.Split(';');

            var bytes = remoteIp.GetAddressBytes();
            var badIp = true;
            foreach (var address in ip)
            {
                var testIp = IPAddress.Parse(address);
                if (testIp.GetAddressBytes().SequenceEqual(bytes))
                {
                    badIp = false;
                    break;
                }
            }

            if (badIp)
            {
                _logger.LogInformation(
                    $"Forbidden Request from Remote IP address: {remoteIp}");
                context.Result = new StatusCodeResult(401);
                return;
            }
        }

        public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
        {
        }

        public void OnPageHandlerSelected(PageHandlerSelectedContext context)
        {
        }
    }
}
```

通过将其添加到 MVC 筛选器集合启用该筛选器。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<ClientIdCheckFilter>();

    services.AddMvc(options =>
    {
        options.Filters.Add
            (new ClientIdCheckPageFilter
                (_loggerFactory, Configuration));
    }).SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

当您运行该应用程序，并请求 Razor 页面时，Razor 页面筛选器就验证客户端 IP。

后续步骤

了解有关 ASP.NET Core 中间件的详细信息。

ASP.NET Core 的性能最佳实践

2019/1/8 • • Edit Online

通过Mike Rousos

本主题提供了性能使用 ASP.NET Core 的最佳实践的指导原则。

本文档中的热代码路径被指频繁调用的函数和大部分执行时间发生的位置的代码路径。热代码路径通常限制应用向外缩放和性能。

主动缓存

在本文档的多个部分中讨论缓存。有关详细信息，请参阅 响应缓存在 ASP.NET Core。

避免阻止调用

ASP.NET Core 应用程序应设计为同时处理多个请求。异步 Api 允许线程通过不在阻塞调用在等待处理数千个并发请求的小型池。而不是等待长时间运行的同步任务，才能完成，线程可以处理其他请求。

ASP.NET Core 应用中的常见性能问题正在阻止可能是异步调用。很多同步阻塞调用会导致线程池资源不足和降低响应时间。

不这样做：

- 通过调用块异步执行Task.Wait或Task.Result。
- 获取在常见的代码路径中的锁。ASP.NET Core 应用是大多数的高性能设计为可并行运行代码时。

执行：

- 请热代码路径异步。
- 以异步方式调用数据访问和长时间运行的操作的 Api。
- 将控制器/Razor 页面操作异步。整个调用堆栈必须是以受益于异步异步/等待模式。

探查器喜欢PerfView可用于查找经常添加到线程线程池。
`Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Start`事件表示要添加到线程池线程。

最大程度减少大型对象分配

.NET Core 垃圾回收器管理的自动在 ASP.NET Core 应用中的内存分配和释放。自动垃圾回收通常意味着开发人员无需担心如何或何时释放内存。但是，清理未引用的对象会占用 CPU 时间，因此开发人员应尽量减少在分配对象热代码路径。垃圾回收，尤其是昂贵大型对象（> 85 K 字节为单位）。大型对象存储在大型对象堆和要求的完整（第 2 代）垃圾回收以进行清除。与不同的是第 0 代和第 1 代集合，第 2 代收集需要使用应用执行暂时挂起。频繁分配和解除分配的大型对象可能导致不一致的性能。

建议：

- 执行考虑缓存经常使用的大型对象。缓存大型对象防止昂贵的分配。
- 不要池使用的缓冲区 `ArrayPool<T>` 来存储大数组。
- 不这样做上分配了很多，生存期较短的大型对象热代码路径。

像前面可以通过查看垃圾回收 (GC) 中的统计信息来诊断内存问题PerfView并检查：

- 垃圾回收暂停时间。

- 垃圾回收中所用的处理器时间百分比。
- 多少垃圾回收是 0、1 和 2 代。

有关详细信息，请参阅垃圾回收和性能。

优化数据访问

与数据存储区或其他远程服务之间的交互通常是 ASP.NET Core 应用的最缓慢部分。读取和写入数据有效地实现良好性能的关键。

建议：

- 执行以异步方式调用 Api 的所有数据访问。
- 不这样做检索不必要的更多数据。编写查询以返回不仅仅是当前 HTTP 请求所需的数据。
- 执行考虑缓存经常访问的数据检索从数据库或远程服务，如果它是可以接受的要稍有过时的数据。根据方案，可能会使用MemoryCache或DistributedCache。有关详细信息，请参阅 响应缓存在 ASP.NET Core。
- 最大程度减少网络往返。目标是检索所需的单个调用中的所有数据，而不是多个调用。
- 不要使用无跟踪查询中 Entity Framework Core 用于只读目的访问数据时。EF Core 可以更有效地返回非跟踪查询的结果。
- 不要筛选器和聚合的 LINQ 查询 (使用 `.Where`， `.Select`，或 `.Sum` 语句，例如)，以便由数据库进行筛选。
- 执行考虑 EF Core 解析客户端可能会导致效率低下的查询执行一些查询运算符。有关详细信息，请参阅客户端评估性能问题
- 不这样做使用投影查询集合，这可能会导致在执行"N + 1"上的 SQL 查询。有关详细信息，请参阅的相关子查询的优化。

请参阅EF 高性能高规模的应用中可提高性能的方法：

- DbContext 池
- 显式编译的查询

我们建议度量值之前提交你的基本代码的前面高性能方法的影响。已编译的查询的其他复杂问题可能不产生的性能改进。

可以通过查看时间检测到问题的查询与访问数据所花Application Insights或使用分析工具。大多数数据库还提供统计信息与有关常执行的查询。

与 HttpClientFactory 池 HTTP 连接

尽管HttpClient实现 `IDisposable` 接口，其目的是重复使用。关闭 `HttpClient` 实例将在中打开套接字 `TIME_WAIT` 状态的短的时间。因此，如果创建和释放的代码路径 `HttpClient` 频繁使用对象，该应用程序可能会出现耗尽可用的套接字。HttpClientFactory中引入了 ASP.NET Core 2.1 作为一种解决方案对此问题。它处理共用的 HTTP 连接来优化性能和可靠性。

建议：

- 不这样做创建和释放 `HttpClient` 直接实例。
- 不要使用HttpClientFactory检索 `HttpClient` 实例。有关详细信息，请参阅来实现可复原的 HTTP 请求的使用 HttpClientFactory。

保持快速的常见代码路径

您希望将所有代码要快，但频繁调用的代码路径是最重要的优化：

- 应用程序的请求处理管道中的中间件组件，尤其是中间件运行尽早在管道中。这些组件对性能产生很大的影响。
- 为每个请求或多个时间每个请求执行的代码。例如，自定义日志记录、授权处理程序或暂时服务的初始化。

建议：

- 不这样做与长时间运行的任务中使用自定义中间件组件。
- 不要使用性能分析工具（如Visual Studio 诊断工具或PerfView）来标识热代码路径。

完成长时间运行的任务之外的 HTTP 请求

到 ASP.NET Core 应用的请求最多可以处理由控制器或调用必要的服务，并返回 HTTP 响应的页面模型。对于某些涉及长时间运行的任务的请求，它是更好的做法使整个请求-响应过程异步。

建议：

- 不这样做等待长时间运行的任务以完成作为普通 HTTP 请求处理的一部分。
- 不要处理与长时间运行请求，请考虑后台服务或与进程外Azure Function。完成的工作进程外是特别适用于 CPU 密集型任务。
- 不要使用实时通信选项，如SignalR以异步方式与客户端通信。

缩小客户端资产

许多 JavaScript、CSS 或图像文件，通常服务于复杂的前端使用 ASP.NET Core 应用。可以通过提高的初始加载请求的性能：

- 该绑定，将多个文件合并成一个。
- 缩小，这减少了通过文件的大小。

建议：

- 不要使用 ASP.NET Core内置支持进行捆绑和缩小客户端资产。
- 不要等其他第三方工具，请考虑Gulp或Webpack更复杂的客户端资产管理。

压缩响应

通常减少响应的大小通常显著增加的应用的响应能力。若要减少有效负载大小的一种方法是压缩应用程序的响应。有关详细信息，请参阅响应压缩。

使用最新的 ASP.NET Core 版本

每个新版本的 ASP.NET 包括性能改进。.NET Core 和 ASP.NET Core 中的优化意味着较新版本将性能优于较旧版本。例如，.NET Core 2.1 添加了支持的已编译的正则表达式和出来 `Span<T>` 。ASP.NET Core 2.2 添加支持 HTTP/2。如果性能是优先级，请考虑升级到 ASP.NET Core 的最新版本。

最大程度减少异常

异常应该很少见。引发和捕获异常速度较慢相对于其他代码流模式。正因为如此，异常不应使用正常控制程序流。

建议：

- 不这样做引发或捕捉异常作为正常程序流，尤其是在热代码路径中的一种方式使用。
- 执行逻辑包括在应用程序以检测和处理会导致异常的条件。
- 执行引发或捕获异常的异常或意外条件。

应用程序的诊断工具（如 Application Insights）可以帮助识别可能会影响性能的应用程序中常见的异常。

响应缓存在 ASP.NET Core

2019/1/8 • • Edit Online

通过John 卢奥语，Rick Anderson，Steve Smith，和Luke Latham

> **NOTE**
> 响应缓存在 Razor 页中是位于 ASP.NET Core 2.1 或更高版本。

查看或下载示例代码（如何下载）

响应缓存可减少客户端或代理到 web 服务器发出的请求数。响应缓存还减少了工作的 web 服务器执行以生成响应。响应缓存控制标头，指定要如何客户端、代理和响应缓存中间件。

ResponseCache 属性参与设置缓存标头，哪些客户端可能会接受缓存的响应时的响应。响应缓存中间件可用于在服务器上的缓存响应。可以使用中间件 `ResponseCache` 特性来影响服务器端的缓存行为的属性。

基于 HTTP 的响应缓存

HTTP 1.1 缓存规范介绍 Internet 缓存的行为方式。是用于缓存的主 HTTP 标头Cache-control，用于指定缓存*指令*。指令控制缓存行为，随着请求来自客户端对服务器进行自己的方式以及响应回客户端从服务器进行自己的方式。请求和响应将通过代理服务器和代理服务器还必须符合 HTTP 1.1 缓存规范。

常见 `Cache-Control` 指令下表中所示。

指令	操作
public	缓存可能会存储响应。
private	响应不是由共享缓存存储。专用缓存可能会存储和重用响应。
max-age	客户端将不会接受其年龄大于指定的秒数的响应。示例：`max-age=60`（60 秒），`max-age=2592000`（1 个月）
no-cache	**在请求**:缓存必须使用存储的响应来满足请求。注意:源服务器重新生成响应的客户端和中间件更新其缓存中存储的响应。 **响应**:响应不必须用于不带验证的源服务器上的后续请求。
no-store	**在请求**:缓存不得存储请求。 **响应**:缓存不得存储任何响应的一部分。

下表中显示其他播放的角色中缓存的缓存标头。

HEADER	函数
年龄	以秒为单位因为响应已生成或在源服务器已成功验证的时间量的估计值。
过期	日期/时间后，响应被视为过时的内容。

HEADER	函数
杂注	存在向后兼容性，使用 HTTP/1.0 缓存设置 `no-cache` 行为。如果 `Cache-Control` 标头存在，则 `Pragma` 忽略标头。
改变	指定缓存的响应必须不发送除非所有的 `Vary` 中缓存的响应的原始请求和新的请求标头字段匹配。

基于 HTTP 的缓存方面请求缓存控制指令

HTTP 1.1 缓存的缓存控制标头的规范需要接受是有效的缓存 `Cache-Control` 客户端发送的标头。客户端可以发出请求的 `no-cache` 标头值和强制服务器生成的每个请求的新响应。

始终遵循客户端 `Cache-Control` 请求标头是有意义，如果您考虑 HTTP 缓存的目标。在正式规范，缓存旨在减少的满足请求的客户端、代理和服务器的网络延迟和网络开销。它不一定是一种方法来控制源服务器上的负载。

没有任何开发人员可以控制此缓存的行为使用时响应缓存中间件因为中间件遵循正式缓存规范。计划的中间件的增强功能是配置要忽略的请求的中间件的机会 `Cache-Control` 标头决定用于缓存的响应时。计划内的增强功能提供更好地控制服务器负载的机会。

在 ASP.NET Core 中其他缓存技术

内存中缓存

内存中缓存使用的服务器内存来存储缓存的数据。此类型的缓存是适用于单个服务器或多个服务器使用*粘性会话*。粘性会话意味着，客户端的请求始终路由到同一个服务器进行处理。

有关详细信息，请参阅缓存在内存。

分布式的缓存

使用分布式的缓存在云或服务器场中托管应用时，将数据存储在内存中。处理请求的服务器之间共享缓存。客户端可以提交的请求，如果客户端的缓存的数据可由任何组中的服务器。ASP.NET Core 提供 SQL Server 和分布式的 Redis 缓存。

有关详细信息，请参阅 分布式缓存在 ASP.NET Core 中。

缓存标记帮助程序

可以使用缓存标记帮助程序缓存中的 MVC 视图或 Razor 页面的内容。缓存标记帮助程序使用内存中缓存来存储数据。

有关详细信息，请参阅ASP.NET Core mvc 缓存标记帮助器。

分布式缓存标记帮助程序

可以使用分布式缓存标记帮助程序缓存中的 MVC 视图或分布式的云或 web 场方案中的 Razor 页面的内容。分布式缓存标记帮助程序使用 SQL Server 或 Redis 存储数据。

有关详细信息，请参阅分布式缓存标记帮助程序。

ResponseCache 属性

ResponseCacheAttribute指定缓存响应中设置相应的标头所需的参数。

> **WARNING**
> 禁用缓存的内容，其中包含已经过身份验证的客户端的信息。仅应为不会更改基于用户的标识或是否在用户登录的内容启用缓存。

VaryByQueryKeys存储的响应因查询密钥的指定列表的值。时的单个值`*`是所有的响应请求查询字符串参数提供中间件会有所不同。`VaryByQueryKeys`需要 ASP.NET Core 1.1 或更高版本。

响应缓存中间件必须能够设置`VaryByQueryKeys`属性；否则，会引发运行时异常。没有为相应的 HTTP 标头`VaryByQueryKeys`属性。该属性是由响应缓存中间件处理的 HTTP 功能。中间件来提供缓存的响应，查询字符串和查询字符串值必须匹配上一个请求。例如，考虑请求和下表中所示的结果的序列。

请求	结果
`http://example.com?key1=value1`	从服务器返回
`http://example.com?key1=value1`	返回从中间件
`http://example.com?key1=value2`	从服务器返回

第一个请求将由服务器返回并缓存在中间件中。因为查询字符串匹配上一个请求，中间件会返回第二个请求。因为查询字符串值不匹配上一个请求，第三个请求不是中间件缓存中。

`ResponseCacheAttribute`用于配置和创建（通过`IFilterFactory`）ResponseCacheFilter。`ResponseCacheFilter`执行更新的适当的 HTTP 标头和响应的功能的工作。筛选器：

- 删除任何现有标头`Vary`，`Cache-Control`，和`Pragma`。
- 写出适当的标头中设置的属性上基于`ResponseCacheAttribute`。
- 更新缓存项 HTTP 功能中，如果响应`VaryByQueryKeys`设置。

改变

此标头时仅写入`VaryByHeader`属性设置。设置为`Vary`属性的值。下面的示例使用`VaryByHeader`属性：

```
[ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
public IActionResult About2()
{
```


```
[ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
public IActionResult About2()
{
```


您可以查看浏览器的网络工具的响应标头。下图显示了输出上边缘 F12网络选项卡时`About2`刷新操作方法：

Headers Body Parameters Cookies Timings

Request URL: http://localhost/Home/About2

Request Method: GET

Status Code: 200 / OK

Request Headers

Accept: text/html, application/xhtml+xml, image/jxr, */*

Accept-Encoding: gzip, deflate

Accept-Language: en-US, en; q=0.8, zh-Hans-CN; q=0.5, zh-Hans; q=0.3

Connection: Keep-Alive

Host: localhost

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36...

Response Headers

Cache-Control: public, max-age=30

Content-Type: text/html; charset=utf-8

Date: Fri, 18 Nov 2016 21:45:14 GMT

Server: Kestrel

Transfer-Encoding: chunked

Vary: User-Agent

NoStore 和 Location.None

`NoStore` 重写的大多数其他属性。当此属性设置为 `true`，则 `Cache-Control` 标头设置为 `no-store`。如果 `Location` 设置为 `None`：

- 将 `Cache-Control` 设置为 `no-store,no-cache`。
- 将 `Pragma` 设置为 `no-cache`。

如果 `NoStore` 是 `false` 并 `Location` 是 `None`，`Cache-Control` 并 `Pragma` 设置为 `no-cache`。

通常设置 `NoStore` 到 `true` 错误页上。例如：

```
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
    return View();
}
```

```
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
    return View();
}
```

这会导致以下标头：

```
Cache-Control: no-store,no-cache
Pragma: no-cache
```

位置和持续时间

若要启用缓存，`Duration` 必须设置为正值和 `Location` 必须是 `Any`（默认值）或 `Client`。在这种情况下，`Cache-Control` 标头设置为位置值后，接 `max-age` 的响应。

> **NOTE**
> `Location` 选项 `Any` 并 `Client` 转换为 `Cache-Control` 标头值的 `public` 和 `private` 分别。按前面所述，设置 `Location` 到 `None` 设置同时 `Cache-Control` 并 `Pragma` 标头 `no-cache` 。

下面生成通过设置显示标头的示例 `Duration` 并保留默认值 `Location` 值：

```
[ResponseCache(Duration = 60)]
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    return View();
}
```

```
[ResponseCache(Duration = 60)]
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    return View();
}
```

这将产生以下标头：

```
Cache-Control: public,max-age=60
```

缓存配置文件

而不是重复 `ResponseCache` 设置中的 MVC 时多个控制器操作属性，缓存配置文件上的设置可以配置为选项 `ConfigureServices` 中的方法 `Startup` 。引用的缓存配置文件中找到的值用作默认值由 `ResponseCache` 属性，并由属性指定任何属性重写。

设置缓存配置文件：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.CacheProfiles.Add("Default",
            new CacheProfile()
            {
                Duration = 60
            });
        options.CacheProfiles.Add("Never",
            new CacheProfile()
            {
                Location = ResponseCacheLocation.None,
                NoStore = true
            });
    }).SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.CacheProfiles.Add("Default",
            new CacheProfile()
            {
                Duration = 60
            });
        options.CacheProfiles.Add("Never",
            new CacheProfile()
            {
                Location = ResponseCacheLocation.None,
                NoStore = true
            });
    });
}
```

引用缓存配置文件：

```
[ResponseCache(Duration = 30)]
public class HomeController : Controller
{
    [ResponseCache(CacheProfileName = "Default")]
    public IActionResult Index()
    {
        return View();
    }
```


```
[ResponseCache(Duration = 30)]
public class HomeController : Controller
{
    [ResponseCache(CacheProfileName = "Default")]
    public IActionResult Index()
    {
        return View();
    }
```


`ResponseCache` 特性可应用于操作（方法）和控制器（类）。方法级属性重写在类级别特性中指定的设置。

在上述示例中，类级别属性指定持续时间为 30 秒，同时持续时间设置为 60 秒，方法级属性引用的缓存配置文件。

生成的标头：

```
Cache-Control: public,max-age=60
```

其他资源

- 在缓存中存储的响应
- Cache-Control
- 缓存在内存中 ASP.NET Core
- 分布式缓存在 ASP.NET Core 中
- 使用 ASP.NET Core 中的更改令牌检测更改
- 响应缓存在 ASP.NET Core 中的中间件
- ASP.NET Core MVC 中的缓存标记帮助程序

- ASP.NET Core 中的分布式缓存标记帮助程序

缓存在内存中 ASP.NET Core

2018/10/30 • • Edit Online

作者：Rick Anderson，John Luo，和Steve Smith

查看或下载示例代码（如何下载）

缓存的基础知识

通过减少生成内容所需的工作，缓存可以显著提高应用的性能和可伸缩性。缓存对不经常更改的数据效果最佳。缓存生成的数据副本的返回速度可以比从原始源返回更快。在编写并测试应用时，应避免依赖缓存的数据。

ASP.NET Core 支持多种不同的缓存。最简单的缓存基于 IMemoryCache，它表示存储在 Web 服务器内存中的缓存。在包含多个服务器的服务器场上运行的应用应确保在使用内存中缓存时，会话是粘性的。粘性会话可确保来自客户端的后续请求都转到同一台服务器。例如，Azure Web 应用使用应用程序请求路由(ARR) 将所有的后续请求路由到同一台服务器。

Web 场中的非粘性会话需要分布式缓存以避免缓存一致性问题。对于某些应用，分布式的缓存可以支持更高版本向外缩放比内存中缓存。使用分布式缓存可将缓存内存卸载到外部进程。

除非将CacheItemPriority设置为 `CacheItemPriority.NeverRemove`，否则 `IMemoryCache` 缓存会在内存压力下清除缓存条目。可以通过设置 `CacheItemPriority` 来调整缓存在内存压力下清除项目的优先级。

内存中缓存可以存储任何对象；分布式缓存接口仅限于 `byte[]`。

System.Runtime.Caching/MemoryCache

System.Runtime.Caching/MemoryCache (NuGet 包) 可用于：

- .NET standard 2.0 或更高版本。
- 任何.NET 实现面向.NET Standard 2.0 或更高版本。例如，ASP.NET Core 2.0 或更高版本。
- .NET framework 4.5 或更高版本。

Microsoft.Extensions.Caching.Memory / `IMemoryCache`（本主题中所述）建议 `System.Runtime.Caching` / `MemoryCache` 由于它更好地集成到 ASP.NET Core。例如，`IMemoryCache` 适用于 ASP.NET Core 的本机依赖关系注入。

使用 `System.Runtime.Caching` / `MemoryCache` 起到桥梁兼容性时将从 ASP.NET 代码移植到 ASP.NET Core 4.x。

缓存指南

- 代码应始终具有一个回退选项，以提取数据并不取决于可用的已缓存值。
- 缓存使用稀缺资源，内存。限制缓存增长：
  - 不要不外部输入用作缓存密钥。
  - 使用过期时间来限制缓存增长。
  - 使用 SetSize、大小和大小限制来限制缓存大小

使用 IMemoryCache

内存中缓存是使用依赖关系注入从应用中引用的服务。请在 `ConfigureServices` 中调用 `AddMemoryCache`：

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMemoryCache();

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvcWithDefaultRoute();
    }
}
```

在构造函数中请求 `IMemoryCache` 实例：

```
public class HomeController : Controller
{
    private IMemoryCache _cache;

    public HomeController(IMemoryCache memoryCache)
    {
        _cache = memoryCache;
    }
```

`IMemoryCache` 需要 NuGet 包Microsoft.Extensions.Caching.Memory。

`IMemoryCache` 需要 NuGet 包Microsoft.Extensions.Caching.Memory，其现已推出Microsoft.AspNetCore.All 元包。

`IMemoryCache` 需要 NuGet 包Microsoft.Extensions.Caching.Memory，其现已推出Microsoft.AspNetCore.App 元包。

下面的代码使用TryGetValue用于检查一次是否在缓存中。如果不缓存一次，创建并添加到缓存中，使用新的条目设置。

```
public static class CacheKeys
{
    public static string Entry { get { return "_Entry"; } }
    public static string CallbackEntry { get { return "_Callback"; } }
    public static string CallbackMessage { get { return "_CallbackMessage"; } }
    public static string Parent { get { return "_Parent"; } }
    public static string Child { get { return "_Child"; } }
    public static string DependentMessage { get { return "_DependentMessage"; } }
    public static string DependentCTS { get { return "_DependentCTS"; } }
    public static string Ticks { get { return "_Ticks"; } }
    public static string CancelMsg { get { return "_CancelMsg"; } }
    public static string CancelTokenSource { get { return "_CancelTokenSource"; } }
}
```

```
public IActionResult CacheTryGetValueSet()
{
    DateTime cacheEntry;

    // Look for cache key.
    if (!_cache.TryGetValue(CacheKeys.Entry, out cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now;

        // Set cache options.
        var cacheEntryOptions = new MemoryCacheEntryOptions()
            // Keep in cache for this time, reset time if accessed.
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        // Save data in cache.
        _cache.Set(CacheKeys.Entry, cacheEntry, cacheEntryOptions);
    }

    return View("Cache", cacheEntry);
}
```

将会显示当前时间和缓存的时间：

```
@model DateTime?

<div>
    <h2>Actions</h2>
    <ul>
        <li><a asp-controller="Home" asp-action="CacheTryGetValueSet">TryGetValue and Set</a></li>
        <li><a asp-controller="Home" asp-action="CacheGet">Get</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreate">GetOrCreate</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreateAsync">GetOrCreateAsync</a></li>
        <li><a asp-controller="Home" asp-action="CacheRemove">Remove</a></li>
    </ul>
</div>

<h3>Current Time: @DateTime.Now.TimeOfDay.ToString()</h3>
<h3>Cached Time: @(Model == null ? "No cached entry found" : Model.Value.TimeOfDay.ToString())</h3>
```

缓存 `DateTime` 值保留在缓存中时的超时期限（和由于内存压力而没有逐出）内的请求。下图显示当前时间以及从缓存中检索的较早时间：

下面的代码使用GetOrCreate并GetOrCreateAsync缓存数据。

```
public IActionResult CacheGetOrCreate()
{
    var cacheEntry = _cache.GetOrCreate(CacheKeys.Entry, entry =>
    {
        entry.SlidingExpiration = TimeSpan.FromSeconds(3);
        return DateTime.Now;
    });

    return View("Cache", cacheEntry);
}

public async Task<IActionResult> CacheGetOrCreateAsync()
{
    var cacheEntry = await
        _cache.GetOrCreateAsync(CacheKeys.Entry, entry =>
    {
        entry.SlidingExpiration = TimeSpan.FromSeconds(3);
        return Task.FromResult(DateTime.Now);
    });

    return View("Cache", cacheEntry);
}
```

以下代码调用Get来提取缓存的时间：

```
public IActionResult CacheGet()
{
    var cacheEntry = _cache.Get<DateTime?>(CacheKeys.Entry);
    return View("Cache", cacheEntry);
}
```

有关缓存方法的说明，请参阅IMemoryCache 方法和CacheExtensions 方法

MemoryCacheEntryOptions

下面的示例执行以下操作：

- 设置绝对到期时间。这是条目可以被缓存的最长时间，防止可调过期持续更新时该条目过时太多。
- 设置可调过期时间。访问此缓存项的请求将重置可调过期时钟。
- 缓存优先级设置为 `CacheItemPriority.NeverRemove` 。
- 设置一个PostEvictionDelegate它将在条目从缓存中清除后调用。在代码中运行该回调的线程不同于从缓存中移除条目的线程。

```
public IActionResult CreateCallbackEntry()
{
    var cacheEntryOptions = new MemoryCacheEntryOptions()
        // Pin to cache.
        .SetPriority(CacheItemPriority.NeverRemove)
        // Add eviction callback
        .RegisterPostEvictionCallback(callback: EvictionCallback, state: this);

    _cache.Set(CacheKeys.CallbackEntry, DateTime.Now, cacheEntryOptions);

    return RedirectToAction("GetCallbackEntry");
}

public IActionResult GetCallbackEntry()
{
    return View("Callback", new CallbackViewModel
    {
        CachedTime = _cache.Get<DateTime?>(CacheKeys.CallbackEntry),
        Message = _cache.Get<string>(CacheKeys.CallbackMessage)
    });
}

public IActionResult RemoveCallbackEntry()
{
    _cache.Remove(CacheKeys.CallbackEntry);
    return RedirectToAction("GetCallbackEntry");
}

private static void EvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.CallbackMessage, message);
}
```

使用 SetSize、大小和大小限制来限制缓存大小

一个 `MemoryCache` 实例可能会根据需要指定和强制实施大小限制。因为缓存中不有任何机制来度量值的条目的大小，没有一个定义的度量单位的内存大小限制。如果设置的缓存内存大小限制，则所有条目必须都指定大小。指定的大小为开发人员选择的单位。

例如：

- 如果 web 应用程序主要已缓存字符串，每个缓存项大小可能是字符串长度。
- 应用程序可以为 1，指定的所有条目的大小，大小限制为条目的计数。

下面的代码创建无单位固定大小MemoryCache可供源依赖关系注入：

```
// using Microsoft.Extensions.Caching.Memory;
public class MyMemoryCache
{
    public MemoryCache Cache { get; set; }
    public MyMemoryCache()
    {
        Cache = new MemoryCache(new MemoryCacheOptions
        {
            SizeLimit = 1024
        });
    }
}
```

SizeLimit没有单位。缓存的条目必须在他们认为最合适，如果缓存内存大小已设置任何单位指定大小。缓存实例的所有用户应都使用相同单位系统。如果已缓存的条目大小的总和超过指定的值，不会缓存条目 `SizeLimit` 。如果设置缓存大小限制，将忽略在条目上设置的缓存大小。

下面的代码寄存器 `MyMemoryCache` 与依赖关系注入容器。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddSingleton<MyMemoryCache>();
}
```

`MyMemoryCache` 创建作为独立的内存缓存的组件的此大小限制缓存的了解，并知道如何适当地设置缓存项大小。

下面的代码使用 `MyMemoryCache` :

```
public class AboutModel : PageModel
{
    private MemoryCache _cache;
    public static readonly string MyKey = "_MyKey";

    public AboutModel(MyMemoryCache memoryCache)
    {
        _cache = memoryCache.Cache;
    }

    [TempData]
    public string DateTime_Now { get; set; }

    public IActionResult OnGet()
    {
        if (!_cache.TryGetValue(MyKey, out string cacheEntry))
        {
            // Key not in cache, so get data.
            cacheEntry = DateTime.Now.TimeOfDay.ToString();

            var cacheEntryOptions = new MemoryCacheEntryOptions()
                // Set cache entry size by extension method.
                .SetSize(1)
                // Keep in cache for this time, reset time if accessed.
                .SetSlidingExpiration(TimeSpan.FromSeconds(3));

            // Set cache entry size via property.
            // cacheEntryOptions.Size = 1;

            // Save data in cache.
            _cache.Set(MyKey, cacheEntry, cacheEntryOptions);
        }

        DateTime_Now = cacheEntry;

        return RedirectToPage("./Index");
    }
}
```

可以通过设置缓存项的大小大小或SetSize扩展方法：

```
public IActionResult OnGet()
{
    if (!_cache.TryGetValue(MyKey, out string cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now.TimeOfDay.ToString();

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            // Set cache entry size by extension method.
            .SetSize(1)
            // Keep in cache for this time, reset time if accessed.
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        // Set cache entry size via property.
        // cacheEntryOptions.Size = 1;

        // Save data in cache.
        _cache.Set(MyKey, cacheEntry, cacheEntryOptions);
    }

    DateTime_Now = cacheEntry;

    return RedirectToPage("./Index");
}
```

缓存依赖项

以下示例演示在依赖项过期时如何使缓存项过期。会将 `CancellationChangeToken` 添加到缓存项。在 `Cancel` 上调用 `CancellationTokenSource`，时，这两个缓存条目都将被清除。

```
public IActionResult CreateDependentEntries()
{
    var cts = new CancellationTokenSource();
    _cache.Set(CacheKeys.DependentCTS, cts);

    using (var entry = _cache.CreateEntry(CacheKeys.Parent))
    {
        // expire this entry if the dependant entry expires.
        entry.Value = DateTime.Now;
        entry.RegisterPostEvictionCallback(DependentEvictionCallback, this);

        _cache.Set(CacheKeys.Child,
            DateTime.Now,
            new CancellationChangeToken(cts.Token));
    }

    return RedirectToAction("GetDependentEntries");
}

public IActionResult GetDependentEntries()
{
    return View("Dependent", new DependentViewModel
    {
        ParentCachedTime = _cache.Get<DateTime?>(CacheKeys.Parent),
        ChildCachedTime = _cache.Get<DateTime?>(CacheKeys.Child),
        Message = _cache.Get<string>(CacheKeys.DependentMessage)
    });
}

public IActionResult RemoveChildEntry()
{
    _cache.Get<CancellationTokenSource>(CacheKeys.DependentCTS).Cancel();
    return RedirectToAction("GetDependentEntries");
}

private static void DependentEvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Parent entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.DependentMessage, message);
}
```

使用 `CancellationTokenSource` 可以将多个缓存条目作为一个组来清除。使用上面代码中的 `using` 模式时 `using` 块中创建的缓存条目会继承触发器和过期时间设置。

附加说明

- 使用回调重新填充缓存项时：
  - 多个请求可能会发现缓存的键值为空，因为回调尚未完成。
  - 这可能导致重新填充缓存的项的多个线程。
- 使用一个缓存条目创建另一个缓存条目时，子条目会复制父条目的过期令牌以及基于时间的过期设置。子级不是通过手动删除过期的或更新的父项。
- 使用PostEvictionCallbacks设置从缓存中逐出缓存项后，将触发的回调。

其他资源

- 分布式缓存在 ASP.NET Core 中
- 使用 ASP.NET Core 中的更改令牌检测更改
- 响应缓存在 ASP.NET Core

- 响应缓存在 ASP.NET Core 中的中间件
- ASP.NET Core MVC 中的缓存标记帮助程序
- ASP.NET Core 中的分布式缓存标记帮助程序

分布式缓存在 ASP.NET Core 中

2018/11/2 • • Edit Online

作者：Steve Smith 和 Luke Latham

分布式的缓存是由多个应用程序服务器，通常作为对其进行访问的应用程序服务器的外部服务维护共享缓存。分布式的缓存可以提高性能和可伸缩性的 ASP.NET Core 应用，尤其是当应用程序托管的云服务或服务器场。

分布式的缓存具有几大优势，其中缓存的数据存储在单个应用程序服务器其他缓存方案。

当分布式缓存的数据，则数据：

- 是*连贯*（一致）跨多个服务器的请求。
- 服务器重新启动和应用部署仍然有效。
- 不使用本地内存。

分布式的缓存配置是特定于实现的。本文介绍如何配置 SQL Server 和 Redis 分布式的缓存。第三方实现也是可用，如NCache (GitHub 上的 NCache)。无论选择哪一种实现，该应用程序与使用缓存进行交互IDistributedCache接口。

查看或下载示例代码（如何下载）

系统必备

若要使用的 SQL Server 分布式缓存，引用Microsoft.AspNetCore.App 元包或添加到的包引用Microsoft.Extensions.Caching.SqlServer包。

若要使用 Redis 分布式缓存，引用Microsoft.AspNetCore.App 元包并添加到的包引用Microsoft.Extensions.Caching.Redis包。Redis 包不包括在`Microsoft.AspNetCore.App`包，因此您必须在项目文件中分别引用 Redis 包。

若要使用的 SQL Server 分布式缓存，引用Microsoft.AspNetCore.All 元包或添加到的包引用Microsoft.Extensions.Caching.SqlServer包。

若要使用 Redis 分布式缓存，引用Microsoft.AspNetCore.All 元包或添加到的包引用Microsoft.Extensions.Caching.Redis包。Redis 包包含在`Microsoft.AspNetCore.All`包，因此无需引用在项目文件中单独的 Redis 包。

若要使用的 SQL Server 分布式缓存中，添加到包引用Microsoft.Extensions.Caching.SqlServer包。

若要使用 Redis 分布式缓存中，添加到包引用Microsoft.Extensions.Caching.Redis包。

IDistributedCache 接口

IDistributedCache接口提供了以下方法操作的分布式的缓存实现中的项：

- GetGetAsync –接受字符串键和检索缓存的项作为`byte[]`数组如果在缓存中找到。
- SetSetAsync –中添加项（作为`byte[]`数组）到使用字符串键的缓存。
- RefreshRefreshAsync –刷新缓存基于其密钥，重置其滑动到期超时值（如果有）中的项。
- RemoveRemoveAsync –移除缓存项根据其字符串键值。

建立分布式缓存服务

注册的实现IDistributedCache在 `Startup.ConfigureServices` 。本主题中所述的框架提供实现包括：

- 分布式的内存缓存
- SQL Server 的分布式的缓存
- 分布式的 Redis 缓存

分布式的内存缓存

分布式内存缓存 (AddDistributedMemoryCache) 是框架提供的实现 `IDistributedCache` ，在内存中存储项。分布式内存缓存不是实际的分布式的缓存。缓存的项存储在运行该应用程序的服务器上的应用程序实例。

分布式内存缓存是一个有用的实现：

- 在开发和测试方案。
- 生产和内存消耗情况中使用一台服务器时不会产生问题。实现分布式内存缓存摘要缓存数据存储。它允许实现真正的分布式缓存解决方案在将来如果多个节点或容错能力变得非常必要。

示例应用将在开发环境中运行应用时使用的分布式内存缓存：

```
public void ConfigureServices(IServiceCollection services)
{
    if (_hostContext.IsDevelopment())
    {
        services.AddDistributedMemoryCache();
    }
    else
    {
        services.AddDistributedSqlServerCache(options =>
        {
            options.ConnectionString =
                _config["DistCache_ConnectionString"];
            options.SchemaName = "dbo";
            options.TableName = "TestCache";
        });
    }
```

分布式的 SQL 服务器缓存

SQL Server 的分布式缓存实现 (AddDistributedSqlServerCache) 允许使用 SQL Server 数据库作为其后备存储分布式的缓存。若要在 SQL Server 实例中创建的 SQL Server 缓存的项表中，可以使用 `sql-cache` 工具。该工具使用名称和指定的架构创建一个表。

添加 `SqlConfig.Tools` 到 `<ItemGroup>` 元素的项目文件并运行 `dotnet restore` 。

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.Extensions.Caching.SqlConfig.Tools"
                          Version="2.0.2" />
</ItemGroup>
```

通过运行 SQL Server 中创建一个表 `sql-cache create` 命令。提供 SQL Server 实例 (`Data Source`)，数据库 (`Initial Catalog`)，架构 (例如， `dbo`)，以及表名 (例如， `TestCache`)：

```
dotnet sql-cache create "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=DistCache;Integrated
Security=True;" dbo TestCache
```

记录一条消息以指示该工具已成功：

```
Table and index were created successfully.
```

创建的表 `sql-cache` 工具具有以下架构：

	Name	Data Type	Allow Nulls
🔑	Id	nvarchar(449)	☐
	Value	varbinary(MAX)	☐
	ExpiresAtTime	datetimeoffset(7)	☐
	SlidingExpirationInSeconds	bigint	☑
	AbsoluteExpiration	datetimeoffset(7)	☑

> **NOTE**
> 应用应操作使用的实例的缓存值IDistributedCache，而不SqlServerCache。

本示例应用实现SqlServerCache非开发环境中：

```
public void ConfigureServices(IServiceCollection services)
{
    if (_hostContext.IsDevelopment())
    {
        services.AddDistributedMemoryCache();
    }
    else
    {
        services.AddDistributedSqlServerCache(options =>
        {
            options.ConnectionString =
                _config["DistCache_ConnectionString"];
            options.SchemaName = "dbo";
            options.TableName = "TestCache";
        });
    }
```

> **NOTE**
> 一个ConnectionString(和（可选）SchemaName并TableName) 通常存储在源代码管理之外 (例如，通过存储机密管理器中或在*appsettings.json /appsettings。{Environment}.json*文件)。连接字符串可能包含应从源代码管理系统的凭据。

分布式的 Redis 缓存

Redis是一种开源的内存中数据存储，通常用作分布式缓存。您可以使用 Redis 本地，并且您可以配置Azure Redis 缓存Azure 托管 ASP.NET Core 应用。应用配置缓存实现使用RedisCache实例(AddDistributedRedisCache):

```
services.AddDistributedRedisCache(options =>
{
    options.Configuration = "localhost";
    options.InstanceName = "SampleInstance";
});
```

若要在本地计算机上安装 Redis:

- 安装Chocolatey Redis 包。
- 运行 `redis-server` 从命令提示符。

使用分布式的缓存

若要使用IDistributedCache接口，请求的实例 `IDistributedCache` 从任何应用程序中的构造函数。实例由提供依赖关系注入 (DI)。

当应用启动时 `IDistributedCache` 注入到 `Startup.Configure` 。使用缓存的当前时间IApplicationLifetime(有关详细信息，请参阅Web 主机：IApplicationLifetime 接口):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    IApplicationLifetime lifetime, IDistributedCache cache)
{
    lifetime.ApplicationStarted.Register(() =>
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        cache.Set("cachedTimeUTC", encodedCurrentTimeUTC, options);
    });
```

示例应用程序注入 `IDistributedCache` 到 `IndexModel` 以供索引页。

每次加载索引页时，缓存时间检查缓存 `OnGetAsync` 。如果尚未过期的缓存的时间，将显示时间。如果自上一次访问缓存的时间（已加载此页的最后一个时间）已过去 20 秒，该页将显示*缓存时间已过*。

通过选择，立即更新为当前时间的缓存的时间重置缓存时间按钮。按钮触发器 `OnPostResetCachedTime` 处理程序方法。

```
public class IndexModel : PageModel
{
    private readonly IDistributedCache _cache;

    public IndexModel(IDistributedCache cache)
    {
        _cache = cache;
    }

    public string CachedTimeUTC { get; set; }

    public async Task OnGetAsync()
    {
        CachedTimeUTC = "Cached Time Expired";
        var encodedCachedTimeUTC = await _cache.GetAsync("cachedTimeUTC");

        if (encodedCachedTimeUTC != null)
        {
            CachedTimeUTC = Encoding.UTF8.GetString(encodedCachedTimeUTC);
        }
    }

    public async Task<IActionResult> OnPostResetCachedTime()
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        await _cache.SetAsync("cachedTimeUTC", encodedCurrentTimeUTC, options);

        return RedirectToPage();
    }
}
```

> **NOTE**
>
> 无需为 `IDistributedCache` 实例使用 Singleton 或 Scoped 生命周期(至少对内置实现来说是这样的)。
>
> 此外可以创建 `IDistributedCache` 实例可能需要某一个而不是使用 DI，但在代码中创建实例会使代码难以测试和违反显式依赖关系原则。

建议

确定哪一种实现的时 `IDistributedCache` 最适合于您的应用程序，请考虑以下：

- 现有的基础结构
- 性能要求
- 成本
- 团队体验

缓存解决方案通常依赖于内存中存储提供快速检索的缓存数据，但内存是有限的资源，而且成本展开。仅存储通常用于缓存中的数据。

通常情况下，Redis 缓存提供更高的吞吐量和延迟低于 SQL Server 缓存。但是，进行基准测试时通常需要确定的缓存策略的性能特征。

当 SQL Server 用作分布式的缓存后备存储时，使用的同一个数据库缓存与应用程序的普通数据存储和检索可以对这两者的性能产生负面影响。我们建议使用专用的 SQL Server 实例为分布式缓存后备存储。

其他资源

- Redis 缓存在 Azure 上
- 在 Azure 上的 SQL 数据库
- ASP.NET Core IDistributedCache 提供程序的 Web 场中的 NCache (在 GitHub 上的 NCache)
- 缓存在内存中 ASP.NET Core
- 使用 ASP.NET Core 中的更改令牌检测更改
- 响应缓存在 ASP.NET Core
- 响应缓存在 ASP.NET Core 中的中间件
- ASP.NET Core MVC 中的缓存标记帮助程序
- ASP.NET Core 中的分布式缓存标记帮助程序
- 在 Web 场中托管 ASP.NET Core

响应缓存在 ASP.NET Core 中的中间件

2018/10/30 • • Edit Online

通过Luke Latham和John 卢奥语

查看或下载示例代码(如何下载)。

此文章介绍了如何在 ASP.NET Core 应用程序中配置缓存响应的中间件。中间件确定何时可缓存的响应、存储响应和从缓存提供服务响应。有关 HTTP 缓存的介绍和 `ResponseCache` 属性，请参阅响应缓存。

Package

引用Microsoft.AspNetCore.App 元包或添加到的包引用Microsoft.AspNetCore.ResponseCaching包。

引用Microsoft.AspNetCore.All 元包或添加到的包引用Microsoft.AspNetCore.ResponseCaching包。

添加到包引用Microsoft.AspNetCore.ResponseCaching包。

配置

在 `Startup.ConfigureServices`，将中间件添加到服务集合。

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddResponseCaching();
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

将应用配置为使用与中间件 `UseResponseCaching` 扩展方法，它将中间件添加到请求处理管道。该示例应用将添加 `Cache-Control` 缓存最多 10 秒的可缓存响应的响应标头。该示例将发送 `Vary` 标头用于配置中间件来提供缓存的响应才 `Accept-Encoding` 后续请求标头匹配的原始请求。中的代码示例中，CacheControlHeaderValue并HeaderNames需要 `using` 语句Microsoft.Net.Http.Headers命名空间。

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseResponseCaching();

    app.Use(async (context, next) =>
    {
        // For GetTypedHeaders, add: using Microsoft.AspNetCore.Http;
        context.Response.GetTypedHeaders().CacheControl =
            new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
            {
                Public = true,
                MaxAge = TimeSpan.FromSeconds(10)
            };
        context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
            new string[] { "Accept-Encoding" };

        await next();
    });

    app.UseMvc();
}
```

响应缓存中间件仅缓存导致 200（正常）状态代码的服务器响应。任何其他响应，其中包括错误页，将忽略由中间件。

> **WARNING**
> 响应包含经过身份验证的客户端的内容必须标记为不可缓存，以防止从存储和处理这些响应中间件。请参阅缓存的条件有关中间件如何确定响应是否可缓存的详细信息。

选项

中间件提供了三个选项用于控制响应缓存。

选项	描述
UseCaseSensitivePaths	确定是否在区分大小写的路径上会缓存响应。默认值为 `false`。
MaximumBodySize	以字节为单位的响应正文最大缓存大小。默认值是 `64 * 1024 * 1024`(64 MB)。
大小限制	以字节为单位的响应缓存中间件大小限制。默认值是 `100 * 1024 * 1024`(100 MB)。

下面的示例配置到中间件：

- 小于或等于 1024 字节的缓存响应。
- 将响应存储通过区分大小写的路径 (例如，`/page1` 和 `/Page1` 单独存储)。

```
services.AddResponseCaching(options =>
{
    options.UseCaseSensitivePaths = true;
    options.MaximumBodySize = 1024;
});
```

VaryByQueryKeys

在使用 MVC/Web API 控制器或 Razor 页面页模型 `ResponseCache` 属性指定设置适当的标头为响应缓存所需的参数。唯一参数 `ResponseCache` 严格需要中间件的属性是 `VaryByQueryKeys`，这并不对应于实际的 HTTP 标头。有关详细信息，请参阅ResponseCache 属性。

不使用时 `ResponseCache` 属性中，响应缓存可以与变化 `VaryByQueryKeys` 功能。使用 `ResponseCachingFeature` 直接从 `IFeatureCollection` 的 `HttpContext`：

```
var responseCachingFeature = context.HttpContext.Features.Get<IResponseCachingFeature>();
if (responseCachingFeature != null)
{
    responseCachingFeature.VaryByQueryKeys = new[] { "MyKey" };
}
```

使用单个值等于 `*` 在 `VaryByQueryKeys` 随缓存所有请求查询参数而都变化。

响应缓存中间件所使用的 HTTP 标头

使用 HTTP 标头配置中间件的响应缓存。

HEADER	详细信息
授权	如果标头存在，不缓存响应。
Cache-Control	中间件只考虑缓存响应标有 `public` 缓存指令。控制缓存使用以下参数： • 最大期限 • max-stale† • 最小值全新 • 必须重新 • 无缓存 • 无存储 • 仅当-缓存 • private • public • s maxage • 代理重新‡ †如果没有限制指定为 `max-stale`，中间件不执行任何操作。 ‡ `proxy-revalidate` 具有相同的效果 `must-revalidate`。 有关详细信息，请参阅RFC 7231：请求缓存控制指令。

HEADER	详细信息
杂注	一个 `Pragma: no-cache` 请求标头中的生成相同的效果 `Cache-Control: no-cache` 。此标头中的相关指令重写 `Cache-Control` 标头，如果存在。考虑使用 HTTP/1.0 的向后兼容性。
Set-Cookie	如果标头存在，不缓存响应。设置一个或多个 cookie 在请求处理管道中的任何中间件可防止响应缓存中间件缓存响应(例如，基于 cookie 的 TempData 提供程序)。
改变	`Vary` 标头可用于不同的缓存的响应的另一个标头。例如，缓存响应的编码，方法是包括 `Vary: Accept-Encoding` 标头，来缓存响应的请求的标头 `Accept-Encoding: gzip` 和 `Accept-Encoding: text/plain` 单独。响应标头值为 `*` 永远不会存储。
过期	此标头被视为陈旧的响应不是存储或检索除非重写由其他 `Cache-Control` 标头。
-None-If-match	如果值不是完整的响应从缓存提供 `*` 和 `ETag` 的响应中不匹配任何提供的值。否则，就会提供 304（未修改）响应。
如果-修改-自	如果 `If-None-Match` 标头不存在，如果缓存的响应日期比提供的值的完整响应从缓存提供。否则，就会提供 304（未修改）响应。
日期	从缓存中，提供服务时 `Date` 由中间件设置标头，如果它未提供原始响应。
内容长度	从缓存中，提供服务时 `Content-Length` 由中间件设置标头，如果它未提供原始响应。
年龄	`Age` 忽略原始响应中发送的标头。提供缓存的响应时，中间件会计算新值。

缓存遵循请求缓存控制指令

中间件遵从的规则HTTP 1.1 缓存规范。规则需要遵守是有效的缓存 `Cache-Control` 客户端发送的标头。在规范中，客户端可以发出请求的 `no-cache` 标头值和强制服务器生成的每个请求的新响应。目前，没有任何开发人员可以控制此缓存行为时使用中间件，因为中间件符合官方缓存规范。

为了更好地控制缓存行为，将介绍其他缓存功能的 ASP.NET Core。请参见下面的主题：

- 缓存在内存中 ASP.NET Core
- 分布式缓存在 ASP.NET Core 中
- ASP.NET Core MVC 中的缓存标记帮助程序
- ASP.NET Core 中的分布式缓存标记帮助程序

疑难解答

如果缓存行为不会按预期方式，，确认响应是可缓存，能够从缓存中提供。检查请求的传入标头和响应的传出标头。启用日志记录以帮助进行调试。

当测试和故障排除的缓存行为，浏览器可以设置影响缓存产生不利的请求标头。例如，设置浏览器可能

`Cache-Control` 标头 `no-cache` 或 `max-age=0` 刷新页面时。以下工具可以显式设置请求标头，并且是测试缓存的首选：

- Fiddler
- Postman

缓存的条件

- 请求必须导致服务器响应 200（正常）状态代码。
- 请求方法必须是 GET 或 HEAD。
- 终端中间件，如静态文件中间件，必须处理前响应缓存中间件的响应。
- `Authorization` 标头不能存在。
- `Cache-Control` 标头参数必须是有效的并且必须标记为响应 `public` 且未标记为 `private` 。
- `Pragma: no-cache` 标头不能存在如果 `Cache-Control` 标头不存在，作为 `Cache-Control` 标头重写 `Pragma` 标头时存在。
- `Set-Cookie` 标头不能存在。
- `Vary` 标头参数必须是有效且不等于 `*` 。
- `Content-Length` 标头值（如果设置）必须与匹配的响应正文的大小。
- IHttpSendFileFeature未使用。
- 响应不能由指定陈旧 `Expires` 标头和 `max-age` 和 `s-maxage` 缓存指令。
- 响应缓冲必须成功，并且响应的大小必须小于已配置或默认 `SizeLimit` 。
- 响应必须是可根据缓存RFC 7234规范。例如， `no-store` 指令不能存在请求或响应标头字段中。请参阅*第 3 部分：在缓存中存储响应*的RFC 7234有关详细信息。

> **NOTE**
> 防伪系统用于生成安全令牌，以防止跨站点请求伪造 (CSRF) 攻击集 `Cache-Control` 并 `Pragma` 标头 `no-cache` ，以便不会缓存响应。有关如何禁用 antiforgery 令牌 HTML 窗体元素的信息，请参阅ASP.NET Core antiforgery 配置。

其他资源

- ASP.NET Core 中的应用启动
- ASP.NET Core 中间件
- 缓存在内存中 ASP.NET Core
- 分布式缓存在 ASP.NET Core 中
- 使用 ASP.NET Core 中的更改令牌检测更改
- 响应缓存在 ASP.NET Core
- ASP.NET Core MVC 中的缓存标记帮助程序
- ASP.NET Core 中的分布式缓存标记帮助程序

在 ASP.NET Core 中的响应压缩

2019/1/8 • • Edit Online

作者：Luke Latham

查看或下载示例代码（如何下载）

网络带宽是一种有限的资源。通常减少响应的大小通常显著增加的应用的响应能力。若要减少有效负载大小的一种方法是压缩应用程序的响应。

何时使用响应压缩中间件

使用 IIS、Apache 或 Nginx 中的基于服务器的响应压缩技术。中间件的性能可能不匹配，服务器模块。HTTP.sys 服务器服务器和Kestrel server 当前不提供内置的压缩支持。

如果你已，使用响应压缩中间件：

- 无法使用以下基于服务器的压缩技术：
  - IIS 动态压缩模块
  - Apache mod_deflate 模块
  - Nginx 压缩和解压缩
- 直接在上托管：
  - HTTP.sys 服务器（以前称为 WebListener）
  - Kestrel 服务器

响应压缩

通常情况下，可以从响应压缩受益本身不压缩任何响应。响应本身不压缩通常包括：CSS、JavaScript、HTML、XML 和 JSON。不应压缩本机压缩的资产，如 PNG 文件。如果您尝试进一步将压缩本机压缩的响应，小型进一步降低大小和传输时间将有可能屏蔽处理压缩花费的时间。不压缩小于大约 150 1000年字节（具体取决于文件的内容和压缩的效率）的文件。压缩小文件的开销可能会产生大于未压缩的文件的压缩的文件。

客户端时客户端可以处理压缩的内容，必须通过发送通知的功能的服务器 `Accept-Encoding` 与请求的标头。当一台服务器发送压缩的内容时，它必须包括中的信息 `Content-Encoding` 标头压缩的响应的编码方式。下表中显示内容由中间件支持指定的编码内容。

`ACCEPT-ENCODING` 标头值	支持的中间件	描述
`br`	是（默认值）	Brotli 压缩的数据格式
`deflate`	否	DEFLATE 压缩的数据格式
`exi`	否	W3C 有效 XML 交换
`gzip`	是	Gzip 文件格式
`identity`	是	"没有 encoding"的标识符：响应必须不进行编码。
`pack200-gzip`	否	Java 存档文件的网络传输格式

`ACCEPT-ENCODING` 标头值	支持的中间件	描述
`*`	是	编码不显式请求任何可用内容

`ACCEPT-ENCODING` 标头值	支持的中间件	描述
`br`	否	Brotli 压缩的数据格式
`deflate`	否	DEFLATE 压缩的数据格式
`exi`	否	W3C 有效 XML 交换
`gzip`	是（默认值）	Gzip 文件格式
`identity`	是	"没有 encoding"的标识符：响应必须不进行编码。
`pack200-gzip`	否	Java 存档文件的网络传输格式
`*`	是	编码不显式请求任何可用内容

有关详细信息，请参阅IANA 官方内容编码列表。

中间件可以添加额外的压缩的自定义的提供程序 `Accept-Encoding` 标头值。有关详细信息，请参阅自定义提供程序下面。

中间件是能够对质量值作出反应 (qvalue， `q`) 权重时由客户端发送，以确定压缩方案的优先级。有关详细信息，请参阅RFC 7231:接受编码。

压缩算法会有所压缩速度和压缩的效率之间的权衡。*有效性*在此上下文中引用的输出大小在压缩之后的。最小大小通过最*最佳*压缩。

在请求中所涉及的标头，发送、缓存和接收压缩的内容是下表中所述。

HEADER	角色
`Accept-Encoding`	从客户端发送到服务器以指示编码方案可向客户端接受的内容。
`Content-Encoding`	从服务器发送到客户端以指示负载中的内容的编码。
`Content-Length`	压缩时，`Content-Length` 删除标头，因为压缩响应的正文内容更改。
`Content-MD5`	压缩时，`Content-MD5` 删除标头，因为正文内容已更改，哈希将不再有效。
`Content-Type`	指定内容的 MIME 的类型。每个响应应指定其 `Content-Type` 。中间件将检查此值以确定是否应压缩响应。中间件指定一组默认 MIME 类型，可以对进行编码，但您可以替换或添加 MIME 类型。

HEADER	角色
`Vary`	时，值为服务器发送 `Accept-Encoding` 到客户端和代理服务器，`Vary` 标头指示与客户端或应缓存的代理（有所不同）响应值的基础 `Accept-Encoding` 的请求标头。返回的内容的结果 `Vary: Accept-Encoding` 标头是同时压缩和未压缩的响应将单独进行缓存。

探索的功能与响应压缩中间件示例应用。此示例演示了：

- 使用 Gzip 和自定义压缩提供程序的应用程序响应的压缩。
- 如何将 MIME 类型添加到压缩的 MIME 类型的默认列表。

package

若要在项目中包含中间件，添加对的引用Microsoft.AspNetCore.App 元包，其中包括Microsoft.AspNetCore.ResponseCompression包。

若要在项目中包含中间件，添加对的引用Microsoft.AspNetCore.All 元包，其中包括Microsoft.AspNetCore.ResponseCompression包。

若要在项目中包含中间件，添加对的引用Microsoft.AspNetCore.ResponseCompression包。

配置

下面的代码演示如何启用响应压缩中间件默认 MIME 类型和压缩提供程序 (Brotli并Gzip):

下面的代码演示如何启用默认 MIME 类型响应压缩中间件和Gzip 压缩提供程序:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddResponseCompression();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseResponseCompression();
    }
}
```

注意：

- `app.UseResponseCompression` 前必须调用 `app.UseMvc`。
- 使用一种工具，如Fiddler， Firebug，或Postman设置 `Accept-Encoding` 请求标头并研究响应标头、大小和正文。

向示例应用程序而无需提交一个申请 `Accept-Encoding` 标头，并观察的响应是未压缩。 `Content-Encoding` 和 `Vary` 标头不存在的响应。

Headers | TextView | WebForms | HexView | Auth | Cookies | Raw | JSON | XML

Request Headers

GET / HTTP/1.1

Client

User-Agent: Fiddler

Transport

Host: localhost:5000

Get SyntaxView | Transformer | Headers | TextView | ImageView | HexView | WebView | Auth | Caching | Cookies | Raw

```
HTTP/1.1 200 OK
Date: Wed, 11 Jan 2017 18:30:11 GMT
Content-Type: text/plain
Server: Kestrel
Content-Length: 2032

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed non risus. Suspendisse lectus tortor, dign
elementum ultrices diam. Maecenas ligula massa, varius a, semper congue, euismod non, mi. Proin porttit
fermentum diam nisl sit amet erat. Duis semper. Duis arcu massa, scelerisque vitae, consequat in, preti
pharetra tempor. Cras vestibulum bibendum augue. Praesent egestas leo in pede. Praesent blandit odio eu
ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Aliquam nibh. Mauris ac ma
non diam sodales hendrerit.Ut velit mauris, egestas sed, gravida nec, ornare ut, mi. Aenean ut orci vel
non tempus aliquam, nunc turpis ullamcorper nibh, in tempus sapien eros vitae ligula.Pellentesque rhonc
diam. Integer quis metus vitae elit lobortis egestas.Lorem ipsum dolor sit amet, consectetuer adipiscin
sapien.Integer tortor tellus, aliquam faucibus, convallis id, congue eu, quam.Mauris ullamcorper felis
purus iaculis lectus, et tristique ligula justo vitae magna.Aliquam convallis sollicitudin purus. Praes
nisl, ac euismod nibh nisl eu lectus. Fusce vulputate sem at sapien.Vivamus leo. Aliquam euismod libero
suscipit nulla in justo.Suspendisse cursus rutrum augue. Nulla tincidunt tincidunt mi. Curabitur iaculi
ultricies lacus lorem varius purus. Curabitur eu amet.
```

向具有示例应用程序提交一个申请 `Accept-Encoding: br` 标头（Brotli 压缩），并观察压缩响应。`Content-Encoding` 和 `Vary` 均存在于响应标头。

Headers | TextView | SyntaxView | WebForms | HexView | Auth | Cookies | Raw | JSON | XML

Request Headers

GET / HTTP/1.1

Client

Accept-Encoding: br

User-Agent: Fiddler

Transport

Host: localhost:5000

Transformer | Headers | TextView | SyntaxView | ImageView | HexView | WebView | Auth | Caching | Cookies | Raw

```
HTTP/1.1 200 OK
Date: Wed, 03 Oct 2018 17:16:21 GMT
Content-Type: text/plain
Server: Kestrel
Transfer-Encoding: chunked
Content-Encoding: br
Vary: Accept-Encoding

3c1
□□□□□□□□□□□te□+□□R□□(H□□□H□□□□*□□□□□□□□□□e□□e□k?Y□□*□H□Kd%Ÿ□□D□|!'□□□□□F+□□□□□87□JgGhB6□a
□d□C□^□C□□.▽□cp□□□□□□mj□□8□□y<□7L}□F□F□□R2□□GC□t□@p64□v|□□□□iY{#□%□□□□□□□□□□>j     □DH□[\+
□}□1□=□
,□}&□X□⊛/?j□□gjo8>□□□;V{□□}□#□
4□`$□□w□□□□;□;}c□□*□□□x□□tY□K□□□&>
4□cp□5^E□5□□□]□□□Z□□L□
i□□□□□Z□d□□h□]ã□JB□x□UyIb□~p=ř.H□]y□□H□d□□□h□6□Ď/□□-□ #%□x□□□□;□□H&□□□□□□□□□□,E%□□s□ę□□□□□&□4$□_□[□j□
□U□□P□□Y□□%□□□□□□□n□T□Q.□ ↖□J6Lo⌀O)I
l)□@p□□$.2□□□(□□
kĒ□cz#□□□'m:]□□O□□□□□Dq□□□□□□)□□□q□'*□□v|□Q
□□□□□L□;□□C/M\□□j□□W□%□□□□j□{Š□I[*□⌐cH□□B□□□□l □/
□Z□□□□□ᵀA□□□r)
□□□D.\□□□□□?□□□□P□L;□□%t□ĝK□□□□M□
□□W□C□□y⊖J□□Y□□ᑊ□un□S□T.□□□z□□%<X.□□y□□1
]□vLOZ□8□□□4□□SG-□□
□□□□□'y<□#□VEk
E□_E[□□t6□□X
□□□?/ʳ□□Y]<K□/?j□□*□□D↺MŘ2□+□ę2□ę ji`qdE□□□□□□□□□;□W□G[2[&□È□□/□□□c□U□□i□□$□#□YK□□□□□L}D□j□□□,Uo□□N□
□B□□□□]□□□?□S□□;□R□L3□□□□□□□□□d□%□□□Oe□ □'v□j-□□i(□□K□□Q□□J□□a□□□□□□□□Y□□□□□□□□□□:y
□87□\q□s□iCL□ ñ□z□□□
V□X2*□qz□□fõ□□□□
1
>
0
```

向具有示例应用程序提交一个申请 `Accept-Encoding: gzip` 标头，并观察压缩响应。`Content-Encoding` 和 `Vary` 均存在于响应标头。

提供程序

Brotli 压缩提供程序

使用 `BrotliCompressionProvider` 若要压缩的响应Brotli 压缩的数据格式。

如果不压缩到提供程序显式添加到CompressionProviderCollection:

- Brotli 压缩提供程序默认情况下添加到与压缩提供程序的数组Gzip 压缩提供程序。
- 当客户端支持 Brotli 压缩的数据格式时，则将默认压缩为 Brotli 压缩。如果客户端不支持 Brotli 时客户端支持 Gzip 压缩，压缩默认设置为 Gzip。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();
}
```

显式添加任何压缩提供程序时，必须添加 Brotoli 压缩提供程序：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

设置压缩级别与 `BrotliCompressionProviderOptions` 。Brotli 压缩提供程序默认为最快的压缩级别 (CompressionLevel.Fastest)，这可能不会产生最高效的压缩。如果需要最高效的压缩，则配置理想的压缩的中间件。

压缩级别	描述
CompressionLevel.Fastest	即使不以最佳方式压缩生成的输出，应尽可能快地完成压缩。
CompressionLevel.NoCompression	应该在不进行压缩。
CompressionLevel.Optimal	响应应以最佳方式压缩，即使压缩操作将耗费更多时间来完成。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();

    services.Configure<BrotliCompressionProviderOptions>(options =>
    {
        options.Level = CompressionLevel.Fastest;
    });
}
```

Gzip 压缩提供程序

使用GzipCompressionProvider若要压缩的响应Gzip 文件格式。

如果不压缩到提供程序显式添加到CompressionProviderCollection:

- Gzip 压缩提供程序默认情况下添加到与压缩提供程序的数组Brotli 压缩提供程序。
- 当客户端支持 Brotli 压缩的数据格式时，则将默认压缩为 Brotli 压缩。如果客户端不支持 Brotli 时客户端支持 Gzip 压缩，压缩默认设置为 Gzip。
- 默认情况下，Gzip 压缩提供程序添加到压缩提供程序的数组。
- 压缩默认值为 Gzip 时客户端支持 Gzip 压缩。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();
}
```

显式添加任何压缩提供程序时，必须添加 Gzip 压缩提供程序：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```


```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```


设置压缩级别与GzipCompressionProviderOptions。Gzip 压缩提供程序默认为最快的压缩级别(CompressionLevel.Fastest)，这可能不会产生最高效的压缩。如果需要最高效的压缩，则配置理想的压缩的中间件。

压缩级别	描述
CompressionLevel.Fastest	即使不以最佳方式压缩生成的输出，应尽可能快地完成压缩。
CompressionLevel.NoCompression	应该在不进行压缩。
CompressionLevel.Optimal	响应应以最佳方式压缩，即使压缩操作将耗费更多时间来完成。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();

    services.Configure<GzipCompressionProviderOptions>(options =>
    {
        options.Level = CompressionLevel.Fastest;
    });
}
```

自定义提供程序

创建自定义压缩实现ICompressionProvider。EncodingName表示的内容编码此 `ICompressionProvider` 生成。中间件使用此信息来选择基于列表中指定的提供程序 `Accept-Encoding` 的请求标头。

使用示例应用程序，客户端提交的请求 `Accept-Encoding: mycustomcompression` 标头。中间件使用自定义压缩的实现，并返回与响应 `Content-Encoding: mycustomcompression` 标头。客户端必须能够将解压缩的自定义压缩实现以处理顺序中的自定义编码。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

```
public class CustomCompressionProvider : ICompressionProvider
{
    public string EncodingName => "mycustomcompression";
    public bool SupportsFlush => true;

    public Stream CreateStream(Stream outputStream)
    {
        // Create a custom compression stream wrapper here
        return outputStream;
    }
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

```
public class CustomCompressionProvider : ICompressionProvider
{
    public string EncodingName => "mycustomcompression";
    public bool SupportsFlush => true;

    public Stream CreateStream(Stream outputStream)
    {
        // Create a custom compression stream wrapper here
        return outputStream;
    }
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

```
public class CustomCompressionProvider : ICompressionProvider
{
    public string EncodingName => "mycustomcompression";
    public bool SupportsFlush => true;

    public Stream CreateStream(Stream outputStream)
    {
        // Create a custom compression stream wrapper here
        return outputStream;
    }
}
```

向具有示例应用程序提交一个申请 `Accept-Encoding: mycustomcompression` 标头，并观察响应标头。 `Vary` 和 `Content-Encoding` 均存在于响应标头。此示例将不会压缩响应正文（未显示）。不压缩的实现中存在 `CustomCompressionProvider` 类的示例。但是，此示例演示您可以用来实现此类的压缩算法。

MIME 类型

中间件指定一组默认的压缩的 MIME 类型：

- `application/javascript`
- `application/json`
- `application/xml`
- `text/css`
- `text/html`
- `text/json`

- `text/plain`
- `text/xml`

替换还是追加使用响应压缩中间件的选项的 MIME 类型。请注意该通配符 MIME 类型，如 `text/*` 不受支持。示例应用程序添加的 MIME 类型 `image/svg+xml` 和压缩，并提供横幅图像的 ASP.NET Core (*banner.svg*)。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

使用安全协议的压缩

可以使用控制通过安全连接的压缩的响应 `EnableForHttps` 选项，它默认处于禁用状态。使用动态生成的页面压缩可能会导致安全问题如犯罪并违反攻击。

添加将 Vary 标头

当压缩响应基于 `Accept-Encoding` 标头，有可能多个压缩的版本的响应的和未压缩的版本。若要指示客户端和代理服务器缓存，多个版本存在，并且应存储 `Vary` 标头添加与 `Accept-Encoding` 值。在 ASP.NET Core 2.0 或更高版本，该中间件将添加 `Vary` 压缩响应时自动标头。

当压缩响应基于 `Accept-Encoding` 标头，有可能多个压缩的版本的响应的和未压缩的版本。若要指示客户端和代理服务器缓存，多个版本存在，并且应存储 `Vary` 标头添加与 `Accept-Encoding` 值。在 ASP.NET Core 1.x 添加 `Vary` 手动完成到响应的标头：

```
// ONLY REQUIRED FOR ASP.NET CORE 1.x APPS
private void ManageVaryHeader(HttpContext context)
{
    // If the Accept-Encoding header is present, add the Vary header
    var accept = context.Request.Headers[HeaderNames.AcceptEncoding];
    if (!StringValues.IsNullOrEmpty(accept))
    {
        context.Response.Headers.Append(HeaderNames.Vary, HeaderNames.AcceptEncoding);
    }
}
```

当位于 Nginx 反向代理中间件问题

如果请求的是代理的 Nginx `Accept-Encoding` 删除标头。删除 `Accept-Encoding` 标头可防止中间件压缩响应。有关详细信息，请参阅NGINX:压缩和解压缩。此问题由跟踪找出传递压缩 Nginx (aspnet/BasicMiddleware #123)。

使用 IIS 动态压缩

如果您有一个活动 IIS 动态压缩模块你想要禁用的应用程序的服务器级别配置，禁用对模块进行补充*web.config*文件。有关详细信息，请参阅禁用 IIS 模块。

疑难解答

使用之类的工具Fiddler， Firebug，或Postman，这样便可以设置 `Accept-Encoding` 请求标头并研究响应标头、大小和正文。默认情况下，响应压缩中间件将压缩满足以下条件的响应：

- `Accept-Encoding` 值为标头，则 `br`， `gzip`， `*`，或与已建立的自定义压缩提供程序相匹配的自定义编码。值不能 `identity` 或具有一个质量值 (qvalue， `q`) 设置为 0（零）。
- MIME 类型（`Content-Type`）必须设置并且必须匹配在配置的 MIME 类型ResponseCompressionOptions。
- 请求必须包括 `Content-Range` 标头。
- 请求必须使用不安全的协议 (http)，除非在响应压缩中间件选项中配置安全协议 (https)。*请注意危险上面所述启用安全内容压缩时。*

- `Accept-Encoding` 值为标头，则 `gzip`， `*`，或与已建立的自定义压缩提供程序相匹配的自定义编码。值不能 `identity` 或具有一个质量值 (qvalue， `q`) 设置为 0（零）。
- MIME 类型（`Content-Type`）必须设置并且必须匹配在配置的 MIME 类型ResponseCompressionOptions。
- 请求必须包括 `Content-Range` 标头。
- 请求必须使用不安全的协议 (http)，除非在响应压缩中间件选项中配置安全协议 (https)。*请注意危险上面所述启用安全内容压缩时。*

其他资源

- ASP.NET Core 中的应用启动
- ASP.NET Core 中间件
- Mozilla 开发人员网络：接受编码
- RFC 7231 节 3.1.2.1:内容 Codings
- RFC 7230 4.2.3 节：Gzip 编码
- GZIP 文件格式规范版本 4.3

性能诊断工具

2019/1/8 • • Edit Online

通过Mike Rousos

本文列出了诊断 ASP.NET Core 中的性能问题的工具。

Visual Studio 诊断工具

分析和诊断工具内置到 Visual Studio 是开始调查性能问题的好时机。这些工具是功能强大且方便地从 Visual Studio 开发环境使用。此工具允许在 ASP.NET Core 应用中 CPU 使用情况、内存使用情况和性能事件的分析。正在内置，可以在开发时轻松分析。

中提供了详细信息Visual Studio 文档。

Application Insights

Application Insights为您的应用程序提供深入的性能数据。Application Insights 自动响应率、故障率、依赖项响应时间和的详细信息上收集数据。Application Insights 支持自定义事件和特定的指标记录到您的应用程序。

Azure Application Insights 提供了多种方式对受监视的应用程序提供见解：

- 应用程序映射– 有助于跨分布式应用程序的所有组件的发现的性能瓶颈或失败的热点。
- 在 Application Insights 门户中的指标边栏选项卡显示度量值和事件计数。
- 在 Application Insights 门户中的性能边栏选项卡:
  - 受监视的应用程序中显示不同的操作的性能详细信息。
  - 允许钻取到单个操作来检查所有部件/依赖项也会影响持续时间长。
  - 可以从此处可收集性能跟踪根据调用 Profiler。
- Azure Application Insights Profiler允许常规和按需分析的.NET 应用程序。Azure 门户会显示捕获的调用堆栈和热路径具有的性能跟踪。此外可以使用 PerfView 的更深入分析下载跟踪文件。

可以在各种环境中使用 application Insights:

- 优化，可在 Azure 中。
- 生产、开发和过渡环境中的工作原理。
- 可从在本地工作Visual Studio或其他托管环境中。

有关详细信息，请参阅用于 ASP.NET Core 的 Application Insights。

PerfView

PerfView是性能分析工具创建的.NET 团队专门为诊断.NET 性能问题。PerfView 允许分析 CPU 使用情况、内存和 GC 行为、性能事件和时钟时间。

您可以了解有关 PerfView 以及如何入门的详细信息PerfView 视频教程或通过读取用户的指南，可以在工具或GitHub 上。

Windows 性能工具包

Windows 性能工具包(WPT) 包含两个组件：Windows 性能记录器 (WPR) 和 Windows 性能分析器 (WPA)。这些工

具生成 Windows 操作系统和应用程序的深入的性能配置的文件。WPT 具有更多方式可视化数据，但收集其数据比 PerfView 的功能较弱。

PerfCollect

PerfView 时.NET 方案一有用的性能分析工具，它仅在上运行 Windows 因此不能使用它来从 ASP.NET Core 应用在 Linux 环境中运行收集跟踪。

PerfCollect是使用分析工具的本机 Linux 的 bash 脚本 (Perf并LTTng) 以 PerfView 可以分析在 Linux 上收集跟踪。在其中不能直接使用 PerfView 的 Linux 环境中出现的性能问题时，PerfCollect 非常有用。相反，PerfCollect 可以收集跟踪然后分析的.NET Core 应用中使用 PerfView 的 Windows 计算机上。

提供了有关如何安装和开始使用 PerfCollect 的详细信息GitHub 上。

其他第三方性能工具

下面列出了一些非常有用的.NET Core 应用程序的性能调查中的第三方性能工具。

- MiniProfiler
- dotTrace 和来自 JetBrains dotMemory
- 从 Intel VTune

负载测试和压力测试 ASP.NET Core

2019/1/11 • • Edit Online

负载测试和压力测试是非常重要的 web 应用的性能和可缩放。其目标是不同甚至它们通常共享类似测试。

负载测试:测试是否应用可以处理表示特定的场景中的用户指定的负载，同时仍能满足响应目标。在正常情况下运行该应用。

压力测试:测试应用程序稳定性下极端条件和通常长时间运行时：

- 较高用户负载 – 峰值或逐渐增加。
- 有限的计算资源。

在负载下，可以应用从故障中恢复并适当地返回到预期的行为？ 应用程序是在压力下*不*在正常情况下运行。

Visual Studio Tools

Visual Studio 允许用户创建、开发和调试 web 性能和负载测试。一个选项，可通过 web 浏览器中录制操作创建测试。

快速入门：创建负载测试项目演示如何创建、配置和运行负载测试项目使用 Visual Studio 2017。

有关详细信息，请参阅其他资源。

可以配置负载测试运行中的本地或在云中使用 Azure DevOps 运行。

Azure DevOps

可以使用启动负载测试运行Azure DevOps 测试计划服务。

该服务支持以下类型的测试格式：

- Visual Studio 测试 – Visual Studio 中创建的 web 测试。
- 在测试期间重播基于 HTTP 存档的测试 – 存档内捕获的 HTTP 流量。
- 基于 URL 的测试– 允许指定的 Url 来加载测试、请求类型、标头和查询字符串。运行设置参数，如持续时间，可以配置负载模式，用户等。
- Apache JMeter测试。

Azure 门户

Azure 门户允许设置和运行负载测试的 Web 应用，直接从 Azure 门户中的应用服务的性能选项卡。

测试可以是包含指定的 URL 或可以测试多个 Url 的 Visual Studio Web 测试文件的手动测试。

在测试结束时，会生成报表来显示该应用程序的性能特征。示例统计信息包括：

- 平均响应时间
- 最大吞吐量：每秒请求数
- 失败百分比

第三方工具

以下列表包含各种功能集与第三方 web 性能工具：

- Apache JMeter :负载测试工具的完整特色的套件。线程绑定：需要每个用户的一个线程。
- ab 的基准测试工具的 Apache HTTP 服务器

- Gatling :使用 GUI 和测试的刻录机的桌面工具。JMeter 性能更强。
- Locust.io :不受线程。

其他资源

加载测试博客系列作者：Charles Sterling。日期，但大多数主题都仍适用。

ASP.NET Core 全球化和本地化

2018/12/5 • • Edit Online

作者：Rick Anderson、Damien Bowden、Bart Calixto、Nadeem Afana 和 Hisham Bin Ateya

使用 ASP.NET Core 创建多语言网站，可让网站拥有更多受众。ASP.NET Core 提供的服务和中间件可将网站本地化为不同的语言和文化。

国际化涉及全球化和本地化。全球化是设计支持不同区域性的应用程序的过程。全球化添加了对一组有关特定地理区域的已定义语言脚本的输入、显示和输出支持。

本地化是将已经针对可本地化性进行处理的全球化应用调整为特定的区域性/区域设置的过程。有关详细信息，请参阅本文档邻近末尾的全球化和本地化术语。

应用本地化涉及以下内容：

1. 使应用内容可本地化
2. 为支持的语言和区域性提供本地化资源
3. 实施策略，为每个请求选择语言/区域性

查看或下载示例代码（如何下载）

使应用内容可本地化

ASP.NET Core 中引入并架构了 `IStringLocalizer` 和 `IStringLocalizer<T>`，以提高开发本地化应用的工作效率。`IStringLocalizer` 使用 ResourceManager 和 ResourceReader，在运行时提供区域性特定资源。简单接口具有一个索引器和一个用于返回本地化字符串的 `IEnumerable`。`IStringLocalizer` 不要求在资源文件中存储默认语言字符串。你可以开发针对本地化的应用，且无需在开发初期创建资源资源文件。下面的代码演示如何针对本地化包装字符串“About Title”。

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Localization;

namespace Localization.Controllers
{
    [Route("api/[controller]")]
    public class AboutController : Controller
    {
        private readonly IStringLocalizer<AboutController> _localizer;

        public AboutController(IStringLocalizer<AboutController> localizer)
        {
            _localizer = localizer;
        }

        [HttpGet]
        public string Get()
        {
            return _localizer["About Title"];
        }
    }
}
```

在上面的代码中，`IStringLocalizer<T>` 实现来源于依赖关系注入。如果找不到“About Title”的本地化值，则返回索引器键，即字符串“About Title”。可将默认语言文本字符串保留在应用中并将它们包装在本地化工具中，以便你

可集中精力开发应用。你使用默认语言开发应用，并针对本地化步骤进行准备，而无需首先创建默认资源文件。或者，你可以使用传统方法，并提供键以检索默认语言字符串。对于许多开发者而言，不具有默认语言 .resx 文件且简单包装字符串文本的新工作流可以减少本地化应用的开销。其他开发者将首选传统工作流，因为它可以更轻松地使用较长字符串文本，更轻松地更新本地化字符串。

对包含 HTML 的资源使用 `IHtmlLocalizer<T>` 实现。`IHtmlLocalizer` 对资源字符串中格式化的参数进行 HTML 编码，但不对资源字符串本身进行 HTML 编码。在下面突出显示的示例中，仅 `name` 参数的值被 HTML 编码。

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Localization;

namespace Localization.Controllers
{
    public class BookController : Controller
    {
        private readonly IHtmlLocalizer<BookController> _localizer;

        public BookController(IHtmlLocalizer<BookController> localizer)
        {
            _localizer = localizer;
        }

        public IActionResult Hello(string name)
        {
            ViewData["Message"] = _localizer["<b>Hello</b><i> {0}</i>", name];

            return View();
        }
```

注意：你通常只想要本地化文本，而不是 HTML。

最低程度，你可以从依赖关系注入获取 `IStringLocalizerFactory`：

```
{
    public class TestController : Controller
    {
        private readonly IStringLocalizer _localizer;
        private readonly IStringLocalizer _localizer2;

        public TestController(IStringLocalizerFactory factory)
        {
            var type = typeof(SharedResource);
            var assemblyName = new AssemblyName(type.GetTypeInfo().Assembly.FullName);
            _localizer = factory.Create(type);
            _localizer2 = factory.Create("SharedResource", assemblyName.Name);
        }

        public IActionResult About()
        {
            ViewData["Message"] = _localizer["Your application description page."]
                + " loc 2: " + _localizer2["Your application description page."];
```

上面的代码演示了这两种工厂创建方法。

可以按控制器、区域对本地化字符串分区，或只有一个容器。在示例应用中，名为 `SharedResource` 的虚拟类用于共享资源。

```
// Dummy class to group shared resources

namespace Localization
{
    public class SharedResource
    {
    }
}
```

某些开发者使用 `Startup` 类，以包含全局或共享字符串。在下面的示例中，使用 `InfoController` 和 `SharedResource` 本地化工具：

```
public class InfoController : Controller
{
    private readonly IStringLocalizer<InfoController> _localizer;
    private readonly IStringLocalizer<SharedResource> _sharedLocalizer;

    public InfoController(IStringLocalizer<InfoController> localizer,
                   IStringLocalizer<SharedResource> sharedLocalizer)
    {
        _localizer = localizer;
        _sharedLocalizer = sharedLocalizer;
    }

    public string TestLoc()
    {
        string msg = "Shared resx: " + _sharedLocalizer["Hello!"] +
                     " Info resx " + _localizer["Hello!"];
        return msg;
    }
```

视图本地化

`IViewLocalizer` 服务可为视图提供本地化字符串。`ViewLocalizer` 类可实现此接口，并从视图文件路径找到资源位置。下面的代码演示如何使用 `IViewLocalizer` 的默认实现：

```
@using Microsoft.AspNetCore.Mvc.Localization

@inject IViewLocalizer Localizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>@Localizer["Use this area to provide additional information."]</p>
```

`IViewLocalizer` 的默认实现可根据视图的文件名查找资源文件。没有使用全局共享资源文件的选项。`ViewLocalizer` 使用 `IHtmlLocalizer` 实现本地化工具，因此 Razor 不会对本地化字符串进行 HTML 编码。你可以参数化资源字符串，`IViewLocalizer` 将对参数进行 HTML 编码，但不会对资源字符串进行。请考虑以下 Razor 标记：

```
@Localizer["<i>Hello</i> <b>{0}!</b>", UserManager.GetUserName(User)]
```

法语资源文件可以包含以下信息：

键	“值”
`<i>Hello</i> <b>{0}!</b>`	`<i>Bonjour</i> <b>{0} !</b>`

呈现的视图可能包含资源文件中的 HTML 标记。

注意：你通常只想要本地化文本，而不是 HTML。

若要在视图中使用共享资源文件，请注入 `IHtmlLocalizer<T>`：

```
@using Microsoft.AspNetCore.Mvc.Localization
@using Localization.Services

@inject IViewLocalizer Localizer
@inject IHtmlLocalizer<SharedResource> SharedLocalizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>

<h1>@SharedLocalizer["Hello!"]</h1>
```

DataAnnotations 本地化

DataAnnotations 错误消息已使用 `IStringLocalizer<T>` 本地化。使用选项 `ResourcesPath = "Resources"`，`RegisterViewModel` 中的错误消息可以存储在以下路径之一：

- *Resources/ViewModels.Account.RegisterViewModel.fr.resx*
- *Resources/ViewModels/Account/RegisterViewModel.fr.resx*

```
public class RegisterViewModel
{
    [Required(ErrorMessage = "The Email field is required.")]
    [EmailAddress(ErrorMessage = "The Email field is not a valid email address.")]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "The Password field is required.")]
    [StringLength(8, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

在 ASP.NET Core MVC 1.1.0 和更高版本中，非验证属性已经进行了本地化。ASP.NET Core MVC 1.0 不会为非验证属性查找本地化字符串。

对多个类使用一个资源字符串

下面的代码演示如何针对具有多个类的验证属性使用一个资源字符串：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddDataAnnotationsLocalization(options => {
            options.DataAnnotationLocalizerProvider = (type, factory) =>
                factory.Create(typeof(SharedResource));
        });
}
```

在上面的代码中，`SharedResource` 是对应于存储验证消息的 resx 的类。使用此方法，DataAnnotations 将仅使用 `SharedResource`，而不是每个类的资源。

为支持的语言和区域性提供本地化资源

SupportedCultures 和 SupportedUICultures

ASP.NET Core 允许指定两个区域性值，`SupportedCultures` 和 `SupportedUICultures`。`SupportedCultures` 的 CultureInfo 对象可决定区域性相关函数的结果，如日期、时间、数字和货币格式等。`SupportedCultures` 确定文本的排序顺序、大小写约定和字符串比较。请参阅 CultureInfo.CurrentCulture 详细了解服务器如何获取区域性。`SupportedUICultures` 可确定哪些转换字符串（.resx 文件中）按 ResourceManager 查找。`ResourceManager` 只需查找 `CurrentUICulture` 决定的区域性特定字符串。.NET 中的每个线程都具有 `CurrentCulture` 和 `CurrentUICulture` 对象。呈现区域性相关函数时，ASP.NET Core 可检查这些值。例如，如果当前线程的区域性设置为“en-US”（英语，美国），`DateTime.Now.ToLongDateString()` 将显示“Thursday, February 18, 2016”，但如果 `CurrentCulture` 设置为“es-ES”（西班牙语，西班牙），则输出将为“jueves, 18 de febrero de 2016”。

资源文件

资源文件是将可本地化的字符串与代码分离的有用机制。非默认语言的转换字符串是独立的 .resx 资源文件。例如，你可能想要创建包含转换字符串、名为 Welcome.es.resx 的西班牙语资源文件。“es”是西班牙语的语言代码。要在 Visual Studio 中创建此资源文件，请支持以下操作：

1. 在“解决方案资源管理器”中，右键单击将包含资源文件的文件夹 >“添加” > “新项”。

2. 在"搜索已安装的模板"框中，输入"资源"并命名该文件。

3. 在"名称"列中输入键值(本机字符串)，在"值"列中输入转换字符串。

Visual Studio 将显示 Welcome.es.resx 文件。

资源文件命名

资源名称是类的完整类型名称减去程序集名称。例如，类 `LocalizationWebsite.Web.Startup` 的主要程序集为 `LocalizationWebsite.Web.dll` 的项目中的法语资源将命名为 Startup.fr.resx。类 `LocalizationWebsite.Web.Controllers.HomeController` 的资源将命名为 Controllers.HomeController.fr.resx。如果目标类的命名空间与将需要完整类型名称的程序集名称不同。例如，在示例项目中，类型 `ExtraNamespace.Tools` 的资源将命名为 ExtraNamespace.Tools.fr.resx。

在示例项目中，`ConfigureServices` 方法将 `ResourcesPath` 设置为"资源"，因此主控制器的法语资源文件的项目相对路径是 Resources/Controllers.HomeController.fr.resx。或者，你可以使用文件夹组织资源文件。对于主控制器，该路径将为 Resources/Controllers/HomeController.fr.resx。如果不使用 `ResourcesPath` 选项，.resx 文件将转到项目的基目录中。`HomeController` 的资源文件将命名为 Controllers.HomeController.fr.resx。选择使用圆点或路径命名约定具体取决于你想如何组织资源文件。

资源名称	圆点或路径命名
Resources/Controllers.HomeController.fr.resx	圆点

资源名称	圆点或路径命名
Resources/Controllers/HomeController.fr.resx	路径

Razor 视图中使用 `@inject IViewLocalizer` 的资源文件遵循类似的模式。可以使用圆点命名或路径命名约定对视图的资源文件进行命名。Razor 视图资源文件可模拟其关联视图文件的路径。假设我们将 `ResourcesPath` 设置为“资源”，与 Views/Home/About.cshtml 视图关联的法语资源文件可能是下面其中之一：

- Resources/Views/Home/About.fr.resx
- Resources/Views.Home.About.fr.resx

如果不使用 `ResourcesPath` 选项，视图的 .resx 文件将位于视图所在的文件夹。

RootNamespaceAttribute

RootNamespace 属性在程序集的根命名空间不同于程序集名称时，提供程序集的根命名空间。

如果程序集的根命名空间不同于程序集名称：

- 默认情况下无法进行本地化。
- 因程序集内搜索资源的方式导致本地化失败。`RootNamespace` 是生成时间值，不可用于正在执行的进程。

如果 `RootNamespace` 不同于 `AssemblyName`，请在 AssemblyInfo.cs 中包括以下内容（参数值替换为实际值）：

```
using System.Reflection;
using Microsoft.Extensions.Localization;

[assembly: ResourceLocation("Resource Folder Name")]
[assembly: RootNamespace("App Root Namespace")]
```

上述代码可成功解析 resx 文件。

区域性回退行为

在搜索资源时，本地化会进行“区域性回退”。从所请求的区域性开始，如果未能找到，则还原至该区域性的父区域性。另外，CultureInfo.Parent 属性代表父区域性。这通常（但并不是总是）意味着从 ISO 中移除区域签名。例如，墨西哥的西班牙语方言为“es-MX”。它具备一个父级“es”西班牙，没有特别指定国家/地区。

假设你的站点接收到了一个区域性为“fr-CA”的“Welcome”资源的请求。本地化系统按顺序查找以下资源，并选择第一个匹配项：

- *Welcome.fr-CA.resx*
- *Welcome.fr.resx*
- *Welcome.resx*（如果 `NeutralResourcesLanguage` 为“fr-CA”）

例如，如果删除了“.fr”区域性指示符，而且已将区域性设置为“法语”，将读取默认资源文件，并本地化字符串。对于不满足所请求区域性的情况，资源管理器可指定默认资源或回退资源。缺少适用于请求区域性的资源时，如果只想返回键，不得具有默认资源文件。

使用 Visual Studio 生成资源文件

如果在 Visual Studio 中创建文件名没有区域性的资源文件（例如 Welcome.resx），Visual Studio 将创建一个 C# 类，并且具有每个字符串的属性。这通常不是你想在 ASP.NET Core 中使用的。你通常没有默认的 .resx 资源文件（没有区域性名称的 .resx 文件）。建议创建具有区域性名称（例如 Welcome.fr.resx）的 .resx 文件。创建具有区域性名称的 .resx 文件时，Visual Studio 不会生成类文件。我们预计许多开发者不会创建默认语言资源文件。

添加其他区域性

每个语言和区域性组合(除默认语言外)都需要唯一资源文件。通过新建 ISO 语言代码属于名称一部分的资源文件，为不同的区域性和区域设置创建资源文件(例如，en-us、fr-ca 和 en-gb)。这些 ISO 编码位于文件名和 .resx 文件扩展之间，如 Welcome.es-MX.resx(西班牙语/墨西哥)。

实施策略，为每个请求选择语言/区域性

配置本地化

通过 `Startup.ConfigureServices` 方法配置本地化：

```
services.AddLocalization(options => options.ResourcesPath = "Resources");

services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
    .AddDataAnnotationsLocalization();
```

- `AddLocalization` 将本地化服务添加到服务容器。上面的代码还可将资源路径设置为“资源”。
- `AddViewLocalization` 添加对本地化视图文件的支持。在此示例视图中，本地化基于视图文件后缀。例如，Index.fr.cshtml 文件中的“fr”。
- `AddDataAnnotationsLocalization` 添加通过 `IStringLocalizer` 抽象对本地化 `DataAnnotations` 验证消息的支持。

本地化中间件

在本地化中间件中设置有关请求的当前区域性。在 `Startup.Configure` 方法中启用本地化中间件。必须在中间件前面配置本地化中间件，它可能检查请求区域性(例如，`app.UseMvcWithDefaultRoute()`)。

```
var supportedCultures = new[]
{
    new CultureInfo("en-US"),
    new CultureInfo("fr"),
};

app.UseRequestLocalization(new RequestLocalizationOptions
{
    DefaultRequestCulture = new RequestCulture("en-US"),
    // Formatting numbers, dates, etc.
    SupportedCultures = supportedCultures,
    // UI strings that we have localized.
    SupportedUICultures = supportedCultures
});

app.UseStaticFiles();
// To configure external authentication,
// see: http://go.microsoft.com/fwlink/?LinkID=532715
app.UseAuthentication();
app.UseMvcWithDefaultRoute();
```

`UseRequestLocalization` 初始化 `RequestLocalizationOptions` 对象。在每个请求上，枚举了 `RequestLocalizationOptions` 的 `RequestCultureProvider` 列表，使用了可成功决定请求区域性的第一个提供程序。默认提供程序来自 `RequestLocalizationOptions` 类：

1. `QueryStringRequestCultureProvider`
2. `CookieRequestCultureProvider`
3. `AcceptLanguageHeaderRequestCultureProvider`

默认列表从最具体到最不具体排序。在本文的后面部分，我们将了解如何更改顺序，甚至添加一个自定义区域性

提供程序。如果没有一个提供程序可以确定请求区域性，则使用 `DefaultRequestCulture`。

QueryStringRequestCultureProvider

某些应用将使用查询字符串来设置区域性和 UI 区域性。对于使用 Cookie 或接受语言标题方法的应用，向 URL 添加查询字符串有助于调试和测试代码。默认情况下，`QueryStringRequestCultureProvider` 注册为 `RequestCultureProvider` 列表中的第一个本地化提供程序。传递查询字符串参数 `culture` 和 `ui-culture`。下面的示例将特定区域性（语言和区域）设置为"西班牙语/墨西哥"：

`http://localhost:5000/?culture=es-MX&ui-culture=es-MX`

如果仅传入两种区域性之一（`culture` 或 `ui-culture`，查询字符串提供程序将使用你传入的区域性设置这两个值。例如，仅设置区域性将同时设置 `Culture` 和 `UICulture`：

`http://localhost:5000/?culture=es-MX`

CookieRequestCultureProvider

通常，生产应用将提供一种机制来使用 ASP.NET Core 区域性 Cookie 设置区域性。若要创建 Cookie，请使用 `MakeCookieValue` 方法。

`CookieRequestCultureProvider` `DefaultCookieName` 将返回用来跟踪用户首选区域性信息的默认 Cookie 名称。默认 Cookie 名称是 `.AspNetCore.Culture`。

Cookie 格式为 `c=%LANGCODE%|uic=%LANGCODE%`，其中 `c` 是 `Culture`，`uic` 是 `UICulture`，例如：

```
c=en-UK|uic=en-US
```

如果仅指定其中一个区域性信息和 UI 区域性，则指定的区域性将同时用于区域性信息和 UI 区域性。

接受语言 HTTP 标题

接受语言标题在大多数浏览器中可设置，最初用于指定用户的语言。此设置指示浏览器已设置为发送或已从基础操作系统继承的内容。浏览器请求的接受语言 HTTP 标题不是检测用户首选语言的可靠方法（请参阅 Setting language preferences in a browser（在浏览器中设置首选项）。生产应用应包括一种用户可以自定义区域性选择的方法。

在 IE 中设置接受语言 HTTP 标题

1. 在齿轮图标中，点击"Internet 选项"。
2. 点击"语言"。

3. 点击“设置语言首选项”。

4. 点击“添加语言”。

5. 添加语言。

6. 点击语言，然后点击“向上移动”。

使用自定义提供程序

假设你想要让客户在数据库中存储其语言和区域性。你可以编写一个提供程序来查找用户的这些值。下面的代码演示如何添加自定义提供程序：

```
private const string enUSCulture = "en-US";

services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[]
    {
        new CultureInfo(enUSCulture),
        new CultureInfo("fr")
    };

    options.DefaultRequestCulture = new RequestCulture(culture: enUSCulture, uiCulture: enUSCulture);
    options.SupportedCultures = supportedCultures;
    options.SupportedUICultures = supportedCultures;

    options.RequestCultureProviders.Insert(0, new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return new ProviderCultureResult("en");
    }));
});
```

使用 `RequestLocalizationOptions` 添加或删除本地化提供程序。

以编程方式设置区域性

GitHub 上的示例项目 Localization.StarterWeb 包含设置 `Culture` 的 UI。Views/Shared/_SelectLanguagePartial.cshtml 文件允许你从支持的区域性列表中选择区域性：

```
@using Microsoft.AspNetCore.Builder
@using Microsoft.AspNetCore.Http.Features
@using Microsoft.AspNetCore.Localization
@using Microsoft.AspNetCore.Mvc.Localization
@using Microsoft.Extensions.Options

@inject IViewLocalizer Localizer
@inject IOptions<RequestLocalizationOptions> LocOptions

@{
    var requestCulture = Context.Features.Get<IRequestCultureFeature>();
    var cultureItems = LocOptions.Value.SupportedUICultures
        .Select(c => new SelectListItem { Value = c.Name, Text = c.DisplayName })
        .ToList();
    var returnUrl = string.IsNullOrEmpty(Context.Request.Path) ? "~/" : $"~{Context.Request.Path.Value}";
}

<div title="@Localizer["Request culture provider:"] @requestCulture?.Provider?.GetType().Name">
    <form id="selectLanguage" asp-controller="Home"
          asp-action="SetLanguage" asp-route-returnUrl="@returnUrl"
          method="post" class="form-horizontal" role="form">
        <label asp-for="@requestCulture.RequestCulture.UICulture.Name">@Localizer["Language:"]</label>
<select name="culture"
          onchange="this.form.submit();"
          asp-for="@requestCulture.RequestCulture.UICulture.Name" asp-items="cultureItems">
        </select>
    </form>
</div>
```

Views/Shared/_SelectLanguagePartial.cshtml 文件添加到了布局文件的 `footer` 部分，使它将可供所有视图使用：

```
<div class="container body-content" style="margin-top:60px">
    @RenderBody()
    <hr>
    <footer>
        <div class="row">
            <div class="col-md-6">
                <p>&copy; @System.DateTime.Now.Year - Localization</p>
            </div>
            <div class="col-md-6 text-right">
                @await Html.PartialAsync("_SelectLanguagePartial")
            </div>
        </div>
    </footer>
</div>
```

`SetLanguage` 方法可设置区域性 Cookie。

```
[HttpPost]
public IActionResult SetLanguage(string culture, string returnUrl)
{
    Response.Cookies.Append(
        CookieRequestCultureProvider.DefaultCookieName,
        CookieRequestCultureProvider.MakeCookieValue(new RequestCulture(culture)),
        new CookieOptions { Expires = DateTimeOffset.UtcNow.AddYears(1) }
    );

    return LocalRedirect(returnUrl);
}
```

不能将 _SelectLanguagePartial.cshtml 插入此项目的示例代码。GitHub 上的项目 Localization.StarterWeb 包含代码，可部分通过依赖关系注入容器将 `RequestLocalizationOptions` 流到 Razor。

全球化和本地化术语

本地化应用的过程还要求基本了解现代软件开发中常用的相关字符集，以及与之相关的问题。尽管所有计算机将文本都存储为数字（代码），但不同的系统使用不同的数字存储相同的文本。本地化过程是指针对特定区域性/区域设置转换应用的用户界面 (UI)。

本地化性是一个中间过程，用于验证全球化应用是否准备好进行本地化。

区域性名称的 RFC 4646 格式为 `<languagecode2>-<country/regioncode2>`，其中 `<languagecode2>` 是语言代码，`<country/regioncode2>` 是子区域性代码。例如，`es-CL` 表示西班牙语（智利），`en-US` 表示英语（美国），而 `en-AU` 表示英语（澳大利亚）。RFC 4646 是一个与语言相关的 ISO 639 双小写字母的区域性代码和一个与国家/地区相关的 ISO 3166 双大写字母子区域性代码的组合。请参阅语言区域性名称。

国际化常缩写为“I18N”。缩写采用第一个和最后一个字母以及它们之间的字母数，因此 18 代表第一个字母“I”和最后一个“N”之间的字母数。这同样适用于全球化 (G11N) 和本地化 (L10N)。

术语：

- 全球化 (G11N)：使应用支持不同语言和区域的过程。
- 本地化 (L10N)：针对给定语言和区域，自定义应用的过程。
- 国际化 (I18N)：同时描绘全球化和本地化。
- 区域性：它是一种语言和区域（可选）。
- 非特定区域性：具有指定语言但不具有区域的区域性。（例如，“en”，“es”）
- 特定区域性：具有指定语言和区域的区域性。（例如，“en-US”，“en-GB”，“es-CL”）
- 父区域性：包含特定区域性的非特定区域性。（例如，“en”是“en-US”和“en-GB”的父区域性）
- 区域设置：区域设置与区域性相同。

> **NOTE**
>
> 可能无法在小数字段中输入十进制逗号。若要使 jQuery 验证支持使用逗号（“,”）表示小数点的的非英语区域设置，以及支持非美国英语日期格式，必须执行使应用全球化的步骤。有关添加十进制逗号的说明，请参阅 GitHub 问题 4076。

其他资源

- 本文所用的 Localization.StarterWeb 项目。
- 对 .NET 应用程序进行全球化和本地化
- .resx 文件中的资源
- Microsoft 多语言应用工具包

在 ASP.NET Core 中配置可移植对象本地化

2018/10/30 • • Edit Online

作者：Sébastien Ros 和 Scott Addie

本文演示通过 Orchard Core 框架在 ASP.NET Core 应用程序中使用可移植对象 (PO) 文件的步骤。

请**注意**：Orchard Core 不是 Microsoft 产品。因此，Microsoft 不提供针对此功能的支持。

查看或下载示例代码（如何下载）

什么是 PO 文件？

PO 文件作为包含给定语言的已转换字符串的文本文件分发。使用 PO 文件替代 .resx 文件的一些优势包括：

- PO 文件支持复数形式；而 .resx 文件不支持复数形式。
- PO 文件的编译方法与 .resx 文件不同。同样，无需专用工具和生成步骤。
- PO 文件可很好地与协作联机编辑工具结合使用。

示例

下面是一个包含两个法语字符串（其中一个具有复数形式）转换的示例 PO 文件：

fr.po

```
#: Services/EmailService.cs:29
msgid "Enter a comma separated list of email addresses."
msgstr "Entrez une liste d'emails séparés par une virgule."

#: Views/Email.cshtml:112
msgid "The email address is \"{0}\"."
msgid_plural "The email addresses are \"{0}\"."
msgstr[0] "L'adresse email est \"{0}\"."
msgstr[1] "Les adresses email sont \"{0}\""
```

此示例使用下列语法：

- `#:`：注释，用于指示要转换的字符串的上下文。根据使用的位置，可对相同字符串进行不同转换。
- `msgid`：未转换的字符串。
- `msgstr`：已转换的字符串。

在支持复数形式的情况下，可定义多个条目。

- `msgid_plural`：未转换的复数形式字符串。
- `msgstr[0]`：针对事例 0 的已转换的字符串。
- `msgstr[N]`：针对事例 N 的已转换的字符串。

可在此处找到 PO 文件规范。

在 ASP.NET Core 中配置 PO 文件支持

此示例基于从 Visual Studio 2017 项目模板中生成的 ASP.NET Core MVC 应用程序。

引用包

添加对 `OrchardCore.Localization.Core` NuGet 包的引用。它可在 MyGet 上的以下包源中获得：

https://www.myget.org/F/orchardcore-preview/api/v3/index.json

.csproj 文件现在包含类似于以下内容的行（版本号可能不同）：

```
<PackageReference Include="OrchardCore.Localization.Core" Version="1.0.0-beta1-3187" />
```

注册服务

将所需服务添加到 Startup.cs 的 `ConfigureServices` 方法：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix);

    services.AddPortableObjectLocalization();

    services.Configure<RequestLocalizationOptions>(options =>
        {
            var supportedCultures = new List<CultureInfo>
            {
                new CultureInfo("en-US"),
                new CultureInfo("en"),
                new CultureInfo("fr-FR"),
                new CultureInfo("fr")
            };

            options.DefaultRequestCulture = new RequestCulture("en-US");
            options.SupportedCultures = supportedCultures;
            options.SupportedUICultures = supportedCultures;
        });
}
```

将所需中间件添加到 Startup.cs 的 `Configure` 方法：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseRequestLocalization();

    app.UseMvcWithDefaultRoute();
}
```

将以下代码添加到所选的 Razor 视图中。在此示例中，使用了 About.cshtml。

```
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer

<p>@Localizer["Hello world!"]</p>
```

注入了 `IViewLocalizer` 实例并将其用于转换文本"Hello world！"。

创建 PO 文件

在应用程序根文件夹中创建名为 .po 的文件。在此示例中，文件名为 fr.po，因为使用了法语：

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

此文件存储了要转换的字符串和已转换为法语的字符串。如有必要，转换将还原为其父级区域性。在此示例中，如果请求的区域性为 `fr-FR` 或 `fr-CA`，则使用 fr.po 文件。

测试应用程序

运行应用程序并导航到 URL `/Home/About`。此时将显示文本 Hello world! 。

导航到 URL `/Home/About?culture=fr-FR`。此时将显示文本 Bonjour le monde! 。

复数形式

PO 文件支持复数形式，在相同字符串需要基于基数以不同方式进行转换时，这非常有用。此任务较为复杂，因为每种语言均定义了自定义规则，以基于基数选择要使用的字符串。

Orchard 本地化包提供了一个 API 以自动调用这些不同的复数形式。

创建复数形式 PO 文件

将以下内容添加到前面所述的 fr.po 文件：

```
msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Il y a un élément."
msgstr[1] "Il y a {0} éléments."
```

有关此示例中每个条目所表示的内容的说明，请参阅什么是 PO 文件？。

使用不同的复数形式添加语言

前面的示例中使用了英语和法语字符串。英语和法语只有两种复数形式且拥有相同形式规则，即一的基数映射到第一种复数形式。任何其他基数映射到第二种复数形式。

并非所有语言都拥有相同的规则。捷克语就是一个例子，它具有三种复数形式。

如下所示，创建 `cs.po` 文件，并记下复数形式如何需要三种不同转换：

```
msgid "Hello world!"
msgstr "Ahoj světe!!"

msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Existuje jedna položka."
msgstr[1] "Existují {0} položky."
msgstr[2] "Existuje {0} položek."
```

若要接受捷克语本地化，请将 `"cs"` 添加到 `ConfigureServices` 方法中受支持的区域性列表：

```
var supportedCultures = new List<CultureInfo>
{
    new CultureInfo("en-US"),
    new CultureInfo("en"),
    new CultureInfo("fr-FR"),
    new CultureInfo("fr"),
    new CultureInfo("cs")
};
```

编辑 Views/Home/About.cshtml 文件以呈现一些基数的已本地化复数形式字符串：

```
<p>@Localizer.Plural(1, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(2, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(5, "There is one item.", "There are {0} items.")</p>
```

注意：在实际方案中，变量将用于表示计数。此处，我们通过三个不同的值重复相同代码，以公开非常特定的事例。

切换区域性时，将显示如下内容：

对于 `/Home/About`：

```
There is one item.
There are 2 items.
There are 5 items.
```

对于 `/Home/About?culture=fr`：

```
Il y a un élément.
Il y a 2 éléments.
Il y a 5 éléments.
```

对于 `/Home/About?culture=cs`：

```
Existuje jedna položka.
Existují 2 položky.
Existuje 5 položek.
```

请注意，对于捷克语区域性，这三种转换各不相同。对于最后两个已转换字符串，法语和英语区域性具有相同构造。

高级任务

将字符串置于上下文中理解

应用程序通常包含要在多个位置中进行转换的字符串。在应用中的特定位置（Razor 视图或类文件），相同字符串可能具有不同转换。PO 文件支持文件上下文概念，此概念可用于对所表示的字符串进行分类。使用文件上下文，可将字符串进行不同转换，具体取决于文件上下文（或缺乏文件上下文）。

PO 本地化服务使用完整类的名称或转换字符串时使用的视图。这通过在 `msgctxt` 条目上设置值来完成。

请考虑对以前的 fr.po 示例作一点小小的补充。可通过设置保留的 `msgctxt` 条目的值将位于 Views/Home/About.cshtml 的 Razor 视图定义为文件上下文：

```
msgctxt "Views.Home.About"
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

这样设置 `msgctxt` 后，导航到 `/Home/About?culture=fr-FR` 时将发生文本转换。而导航到 `/Home/Contact?culture=fr-FR` 时，则不发生转换。

当没有特定条目与给定文件上下文相匹配时，Orchard Core 的回退机制将在没有上下文的情况下查找适当的 PO 文件。假设不存在针对 Views/Home/Contact.cshtml 定义的特定文件上下文，导航到 `/Home/Contact?culture=fr-FR`，加载 PO 文件，如：

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

更改 PO 文件的位置

可以在 `ConfigureServices` 中更改 PO 文件的默认位置：

```
services.AddPortableObjectLocalization(options => options.ResourcesPath = "Localization");
```

在此示例中，从本地化文件夹加载 PO 文件。

实现用于查找本地化文件的自定义逻辑

当需要更复杂的逻辑以查找 PO 文件时，可实现 `OrchardCore.Localization.PortableObject.ILocalizationFileLocationProvider` 接口并将其注册为服务。在可将 PO 文件存储于不同位置或在文件夹层次结构中找到文件时，这非常有用。

使用不同默认复数形式语言

此包包含特定于两种复数形式的 `Plural` 扩展方法。对于需要更多复数形式的语言，请创建扩展方法。通过扩展方法，无需提供默认语言的任何本地化文件 — 可在代码中直接使用原始字符串。

可使用更加广泛的、接受转换的字符串数组的 `Plural(int count, string[] pluralForms, params object[] arguments)` 重载。

ASP.NET Core 中的 URL 重写中间件

2019/1/10 • • Edit Online

作者：Luke Latham 和 Mikael Mengistu

对于本主题的 1.1 版本，请下载 ASP.NET Core（版本 1.1，PDF）中的 URL 重写中间件。

本文档介绍 URL 重写并说明如何在 ASP.NET Core 应用中使用 URL 重写中间件。

URL 重写是根据一个或多个预定义规则修改请求 URL 的行为。URL 重写会在资源位置和地址之间创建一个抽象，使位置和地址不紧密相连。在以下几种方案中，URL 重写很有价值：

- 暂时或永久移动或替换服务器资源，并维护这些资源的稳定定位符。
- 拆分在不同应用或同一应用的不同区域中处理的请求。
- 删除、添加或重新组织传入请求上的 URL 段。
- 优化搜索引擎优化 (SEO) 的公共 URL。
- 允许使用友好的公共 URL 来帮助访问者预测请求资源后返回的内容。
- 将不安全请求重定向到安全终结点。
- 防止热链接，外部站点会通过热链接将其他站点的资产链接到其自己的内容，从而利用托管在其他站点上的静态资产。

> **NOTE**
>
> URL 重写可能会降低应用的性能。如果可行，应限制规则的数量和复杂度。

查看或下载示例代码（如何下载）

URL 重定向和 URL 重写

URL 重定向和 URL 重写之间的用词差异很细微，但这对于向客户端提供资源具有重要意义。ASP.NET Core 的 URL 重写中间件能够满足两者的需求。

URL 重定向涉及客户端操作，指示客户端访问与客户端最初请求地址不同的资源。这需要往返服务器。客户端对资源发出新请求时，返回客户端的重定向 URL 会出现在浏览器地址栏。

如果 `/resource` 被重定向到 `/different-resource`，则服务器作出响应，指示客户端应在 `/different-resource` 获取资源，所提供的状态代码指示重定向是临时的还是永久的。

将请求重定向到不同 URL 时，通过使用响应指定状态代码来指示重定向是永久还是临时：

- 如果资源有一个新的永久性 URL，并且你希望指示客户端所有将来的资源请求都使用新 URL，则使

用“301 (永久移动)”状态代码。收到 301 状态代码时，客户端可能会缓存响应并重用这段代码。

- “302 (找到)”状态代码用于后列情况：重定向操作是临时的或通常会发生变化。302 状态代码向客户端指示不存储 URL 并在将来使用。

有关状态代码的详细信息，请参阅 RFC 2616：状态代码定义。

URL 重写是服务器端操作，它从与客户端请求的资源地址不同的资源地址提供资源。重写 URL 不需要往返服务器。重写的 URL 不会返回客户端，也不会出现在浏览器地址栏。

如果 `/resource` 重写到 `/different-resource`，服务器会在内部提取并返回 `/different-resource` 处的资源。

尽管客户端可能能够检索已重写 URL 处的资源，但是，客户端发出请求并收到响应时，并不知道已重写 URL 处存在的资源。

URL 重写示例应用

可使用示例应用了解 URL 重写中间件的功能。该应用程序应用重定向和重写规则，并显示多个方案的重定向或重写的 URL。

何时使用 URL 重写中间件

如果无法使用以下方法，请使用 URL 重写中间件：

- 在 Windows Server 上使用带 IIS 的 URL 重写模块
- 在 Apache 服务器上使用 Apache mod_rewrite 模块
- Nginx 上的 URL 重写

此外，如果应用程序在 HTTP.sys 服务器（旧称 WebListener）上托管，请使用中间件。

使用 IIS、Apache 和 Nginx 中的基于服务器的 URL 重写技术的主要原因：

- 中间件不支持这些模块的完整功能。

  服务器模块的一些功能不适用于 ASP.NET Core 项目，例如 IIS 重写模块的 `IsFile` 和 `IsDirectory` 约束。在这些情况下，请改为使用中间件。

- 中间件性能与模块性能不匹配。

  基准测试是确定哪种方法会最大程度降低性能或降低的性能是否可忽略不计的唯一方法。

Package

要在项目中包含中间件，请在项目文件中添加对 Microsoft.AspNetCore.App 元数据包的包引用，该文件包含 Microsoft.AspNetCore.Rewrite 包。

不使用 `Microsoft.AspNetCore.App` 元包时，向 `Microsoft.AspNetCore.Rewrite` 包添加项目引用。

扩展和选项

通过使用扩展方法为每条重写规则创建 RewriteOptions 类的实例，建立 URL 重写和重写定向规则。按所需的处理顺序链接多个规则。使用 UseRewriter 将 `RewriteOptions` 添加到请求管道时，它会被传递到 URL 重写中间

件：

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

将非 www 重定向到 www

三个选项允许应用将非 `www` 重新定向到 `www`：

- AddRedirectToWwwPermanent – 如果请求是非 `www`，则将请求永久重定向到 `www` 子域。使用 Status308PermanentRedirect 状态代码进行重定向。
- AddRedirectToWww – 如果传入请求是非 `www`，则将请求重定向到 `www` 子域。使用 Status307TemporaryRedirect 状态代码进行重定向。重载允许提供响应状态代码。使用 StatusCodes 类的字段实现状态代码分配。

URL 重定向

使用 AddRedirect 将请求重定向。第一个参数包含用于匹配传入 URL 路径的正则表达式。第二个参数是替换字符串。第三个参数（如有）指定状态代码。如不指定状态代码，则状态代码默认为“302 (已找到)”，指示资源暂时移动或替换。

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

在启用了开发人员工具的浏览器中，向路径为 `/redirect-rule/1234/5678` 的示例应用发出请求。正则表达式匹配 `redirect-rule/(.*)` 上的请求路径，且该路径会被 `/redirected/1234/5678` 替代。重定向 URL 以"302 (已找到)"状态代码发回客户端。浏览器会在浏览器地址栏中出现的重定向 URL 上发出新请求。由于示例应用中的规则都不匹配重定向 URL：

- 第二个请求从应用程序收到"200 (正常)"响应。
- 响应正文显示了重定向 URL。

重定向 URL 时，系统将向服务器进行一次往返。

> **WARNING**
>
> 建立重定向规则时务必小心。系统会根据应用的每个请求(包括重定向后的请求)对重定向规则进行评估。很容易便会意外创建无限重定向循环。

原始请求：`/redirect-rule/1234/5678`

括号内的表达式部分称为"捕获组"。表达式的点 (`.`) 表示匹配任何字符。星号 (`*`) 表示零次或多次匹配前面的

字符。因此，URL 的最后两个路径段 `1234/5678` 由捕获组 `(.*)` 捕获。在请求 URL 中提供的位于 `redirect-rule/` 之后的任何值均由此单个捕获组捕获。

在替换字符串中，将捕获组注入带有美元符号（`$`）、后跟捕获序列号的字符串中。获取的第一个捕获组值为 `$1`，第二个为 `$2`，并且正则表达式中的其他捕获组值将依次继续排列。示例应用的重定向规则正则表达式中只有一个捕获组，因此替换字符串中只有一个注入组，即 `$1`。如果应用此规则，URL 将变为 `/redirected/1234/5678`。

URL 重定向到安全的终结点

使用 AddRedirectToHttps 将 HTTP 请求重定向到采用 HTTPS 协议的相同主机和路径。如不提供状态代码，则中间件默认为"302(已找到)"。如果不提供端口：

- 中间件默认为 `null`。
- 方案更改为 `https`（HTTPS 协议），客户端访问端口 443 上的资源。

下面的示例演示如何将状态代码设置为"301(永久移动)"并将端口更改为 5001。

```
public void Configure(IApplicationBuilder app)
{
    var options = new RewriteOptions()
        .AddRedirectToHttps(301, 5001);

    app.UseRewriter(options);
}
```

使用 AddRedirectToHttpsPermanent 将不安全的请求重定向到端口 443 上的采用安全 HTTPS 协议的相同主机和路径。中间件将状态代码设置为"301 (永久移动)"。

```
public void Configure(IApplicationBuilder app)
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent();

    app.UseRewriter(options);
}
```

> **NOTE**
> 当重定向到安全的终结点并且不需要其他重定向规则时，建议使用 HTTPS 重定向中间件。有关详细信息，请参阅强制使用 HTTPS主题。

示例应用能够演示如何使用 `AddRedirectToHttps` 或 `AddRedirectToHttpsPermanent`。将扩展方法添加到 `RewriteOptions`。在任何 URL 上向应用发出不安全的请求。消除自签名证书不受信任的浏览器安全警告，或创建例外以信任证书。

使用 `AddRedirectToHttps(301, 5001)` 的原始请求：`http://localhost:5000/secure`

使用 `AddRedirectToHttpsPermanent` 的原始请求：`http://localhost:5000/secure`

URL 重写

使用 AddRewrite 创建重写 URL 的规则。第一个参数包含用于匹配传入 URL 路径的正则表达式。第二个参数是替换字符串。第三个参数 `skipRemainingRules: {true|false}` 指示如果当前规则适用，中间件是否要跳过其他重写规则。

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

原始请求：`/rewrite-rule/1234/5678`

表达式开头的脱字号（`^`）意味着匹配从 URL 路径的开头处开始。

在前面的重定向规则 `redirect-rule/(.*)` 的示例中，正则表达式的开头没有脱字号（`^`）。因此，路径中 `redirect-rule/` 之前的任何字符都可能成功匹配。

路径	匹配
`/redirect-rule/1234/5678`	是
`/my-cool-redirect-rule/1234/5678`	是
`/anotherredirect-rule/1234/5678`	是

重写规则 `^rewrite-rule/(\d+)/(\d+)` 只能与以 `rewrite-rule/` 开头的路径匹配。注意下表中的匹配差异。

路径	匹配
`/rewrite-rule/1234/5678`	是
`/my-cool-rewrite-rule/1234/5678`	No
`/anotherrewrite-rule/1234/5678`	No

在表达式的 `^rewrite-rule/` 部分之后，有两个捕获组 `(\d+)/(\d+)`。`\d` 表示与数字匹配。加号（`+`）表示与前面的一个或多个字符匹配。因此，URL 必须包含数字加正斜杠加另一个数字的形式。这些捕获组以 `$1` 和 `$2` 的形式注入重写 URL 中。重写规则替换字符串将捕获组放入查询字符串中。重写 `/rewrite-rule/1234/5678` 的请求路径，获取 `/rewritten?var1=1234&var2=5678` 处的资源。如果原始请求中存在查询字符串，则重写 URL 时会保留此字符串。

无需往返服务器来获取资源。如果资源存在，系统会提取资源并以“200（正常）”状态代码返回给客户端。因为客户端不会被重定向，所以浏览器地址栏中的 URL 不会发生更改。客户端无法检测到服务器上发生的 URL 重写操作。

NOTE

尽可能使用 `skipRemainingRules: true`，因为匹配规则在计算上很昂贵并且增加了应用响应时间。对于最快的应用响应：

- 按照从最频繁匹配的规则到最不频繁匹配的规则排列重写规则。
- 如果出现匹配项且无需处理任何其他规则，则跳过剩余规则的处理。

Apache mod_rewrite

使用 AddApacheModRewrite 应用 Apache mod_rewrite 规则。请确保将规则文件与应用一起部署。有关 mod_rewrite 规则的详细信息和示例，请参阅 Apache mod_rewrite。

StreamReader 用于读取 ApacheModRewrite.txt 规则文件中的规则：

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

示例应用将请求从 `/apache-mod-rules-redirect/(.\*)` 重定向到 `/redirected?id=$1`。响应状态代码为“302 (已找到)”。

```
# Rewrite path with additional sub directory
RewriteRule ^/apache-mod-rules-redirect/(.*) /redirected?id=$1 [L,R=302]
```

原始请求：`/apache-mod-rules-redirect/1234`

中间件支持下列 Apache mod_rewrite 服务器变量：

- CONN_REMOTE_ADDR
- HTTP_ACCEPT
- HTTP_CONNECTION

- HTTP_COOKIE
- HTTP_FORWARDED
- HTTP_HOST
- HTTP_REFERER
- HTTP_USER_AGENT
- HTTPS
- IPV6
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_METHOD
- REQUEST_SCHEME
- REQUEST_URI
- SCRIPT_FILENAME
- SERVER_ADDR
- SERVER_PORT
- SERVER_PROTOCOL
- TIME
- TIME_DAY
- TIME_HOUR
- TIME_MIN
- TIME_MON
- TIME_SEC
- TIME_WDAY
- TIME_YEAR

IIS URL 重写模块规则

若要使用适用于 IIS URL 重写模块的同一规则集，使用 AddIISUrlRewrite。请确保将规则文件与应用一起部署。当在 Windows Server IIS 上运行时，请勿指示中间件使用应用的 web.config 文件。使用 IIS 时，应将这些规则存储在应用的 web.config 文件之外，以避免与 IIS 重写模块发生冲突。有关 IIS URL 重写模块规则的详细信息和示例，请参阅 Using Url Rewrite Module 2.0（使用 URL 重写模块 2.0）和 URL Rewrite Module Configuration Reference（URL 重写模块配置引用）。

StreamReader 用于读取 IISUrlRewrite.xml 规则文件中的规则：

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

示例应用将请求从 `/iis-rules-rewrite/(.*)` 重写为 `/rewritten?id=$1`。以“200 (正常)”状态代码作为响应发送到客户端。

```
<rewrite>
  <rules>
    <rule name="Rewrite segment to id querystring" stopProcessing="true">
      <match url="^iis-rules-rewrite/(.*)$" />
      <action type="Rewrite" url="rewritten?id={R:1}" appendQueryString="false"/>
    </rule>
  </rules>
</rewrite>
```

原始请求：`/iis-rules-rewrite/1234`

如果有配置了服务器级别规则(可对应用产生不利影响)的活动 IIS 重写模块，则可禁用应用的 IIS 重写模块。有关详细信息，请参阅禁用 IIS 模块。

不支持的功能

与 ASP.NET Core 2.x 一同发布的中间件不支持以下 IIS URL 重写模块功能：

- 出站规则
- 自定义服务器变量
- 通配符
- LogRewrittenUrl

受支持的服务器变量

中间件支持下列 IIS URL 重写模块服务器变量：

- CONTENT_LENGTH
- CONTENT_TYPE
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_HOST
- HTTP_REFERER
- HTTP_URL
- HTTP_USER_AGENT
- HTTPS
- LOCAL_ADDR
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_URI

> **NOTE**
>
> 也可通过 PhysicalFileProvider 获取 IFileProvider。这种方法可为重写规则文件的位置提供更大的灵活性。请确保将重写规则文件部署到所提供路径的服务器中。
>
> ```
> PhysicalFileProvider fileProvider = new PhysicalFileProvider(Directory.GetCurrentDirectory());
> ```

基于方法的规则

使用 Add 在方法中实现自己的规则逻辑。`Add` 公开 RewriteContext，这使 HttpContext 可用于方法中。RewriteContext.Result 决定如何处理其他管道进程。将值设置为下表中的 RuleResult 字段之一。

`REWRITECONTEXT.RESULT`	**操作**
`RuleResult.ContinueRules`（默认值）	继续应用规则。
`RuleResult.EndResponse`	停止应用规则并发送响应。
`RuleResult.SkipRemainingRules`	停止应用规则并将上下文发送给下一个中间件。

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

示例应用演示了如何对以 .xml 结尾的路径的请求进行重定向。如果发出针对 `/file.xml` 的请求，请求将重定向到 `/xmlfiles/file.xml` 。状态代码设置为“301 (永久移动)”。当浏览器发出针对 /xmlfiles/file.xml 的新请求后，静态文件中间件会将文件从 wwwroot / xmlfiles 文件夹提供给客户端。对于重定向，请显式设置响应的状态代码。否则，将会返回“200 (正常)”状态代码，且客户端上不会发生重写。

RewriteRules.cs:

```
public static void RedirectXmlFileRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    // Because the client is redirecting back to the same app, stop
    // processing if the request has already been redirected.
    if (request.Path.StartsWithSegments(new PathString("/xmlfiles")))
    {
        return;
    }

    if (request.Path.Value.EndsWith(".xml", StringComparison.OrdinalIgnoreCase))
    {
        var response = context.HttpContext.Response;
        response.StatusCode = StatusCodes.Status301MovedPermanently;
        context.Result = RuleResult.EndResponse;
        response.Headers[HeaderNames.Location] =
            "/xmlfiles" + request.Path + request.QueryString;
    }
}
```

此方法还可以重写请求。示例应用演示了如何重写任何文本文件请求的路径以从 wwwroot 文件夹中提供 file.txt 文本文件。静态文件中间件基于更新的请求路径来提供文件：

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

RewriteRules.cs:

```
public static void RewriteTextFileRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    if (request.Path.Value.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
    {
        context.Result = RuleResult.SkipRemainingRules;
        request.Path = "/file.txt";
    }
}
```

基于 IRule 的规则

使用 Add 在实现 IRule 接口的类中使用规则逻辑。与使用基于方法的规则方法相比，`IRule` 提供了更大的灵活性。实现类可能包含构造函数，可在其中传入 ApplyRule 方法的参数。

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

检查示例应用中 `extension` 和 `newPath` 的参数值是否符合多个条件。`extension` 须包含一个值，并且该值必须是 .png、.jpg 或 .gif。如果 `newPath` 无效，则会引发 ArgumentException。如果发出针对 image.png 的请求，请求将重定向到 `/png-images/image.png`。如果发出针对 image.png 的请求，请求将重定向到 `/jpg-images/image.jpg`。状态代码设置为“301 (永久移动)”，`context.Result` 设置为停止处理规则并发送响应。

```
public class RedirectImageRequests : IRule
{
    private readonly string _extension;
    private readonly PathString _newPath;

    public RedirectImageRequests(string extension, string newPath)
    {
        if (string.IsNullOrEmpty(extension))
        {
            throw new ArgumentException(nameof(extension));
        }

        if (!Regex.IsMatch(extension, @"^\.(png|jpg|gif)$"))
        {
            throw new ArgumentException("Invalid extension", nameof(extension));
        }

        if (!Regex.IsMatch(newPath, @"(/[A-Za-z0-9]+)+?"))
        {
            throw new ArgumentException("Invalid path", nameof(newPath));
        }

        _extension = extension;
        _newPath = new PathString(newPath);
    }

    public void ApplyRule(RewriteContext context)
    {
        var request = context.HttpContext.Request;

        // Because we're redirecting back to the same app, stop
        // processing if the request has already been redirected
        if (request.Path.StartsWithSegments(new PathString(_newPath)))
        {
            return;
        }

        if (request.Path.Value.EndsWith(_extension, StringComparison.OrdinalIgnoreCase))
        {
            var response = context.HttpContext.Response;
            response.StatusCode = StatusCodes.Status301MovedPermanently;
            context.Result = RuleResult.EndResponse;
            response.Headers[HeaderNames.Location] =
                _newPath + request.Path + request.QueryString;
        }
    }
}
```

原始请求：`/image.png`

原始请求：`/image.jpg`

正则表达式示例

目标	正则表达式字符串和匹配示例	替换字符串和输出示例
将路径重写为查询字符串	`^path/(.*)/(.*)` `/path/abc/123`	`path?var1=$1&var2=$2` `/path?var1=abc&var2=123`
去掉尾部反斜杠	`(.*)/$` `/path/`	`$1` `/path`
强制添加尾部反斜杠	`(.*[^/])$` `/path`	`$1/` `/path/`

避免重写特定请求	`^(.*)(?<!\.axd)$` 或
	`^(?!.*\.axd$)(.*)$`
	正确：`/resource.htm`
	错误：`/resource.axd`
		`rewritten/$1`
		`/rewritten/resource.htm`
		`/resource.axd`

重新排列 URL 段	`path/(.*)/(.*)/(.*)` `path/1/2/3`	`path/$3/$2/$1` `path/3/2/1`
替换 URL 段	`^(.*)/segment2/(.*)` `/segment1/segment2/segment3`	`$1/replaced/$2` `/segment1/replaced/segment3`

其他资源

- 应用程序启动
- 中间件
- .NET 中的正则表达式
- 正则表达式语言 - 快速参考
- Apache mod_rewrite
- 使用 URL 重写模块 2.0（适用于 IIS）
- URL Rewrite Module Configuration Reference（URL 重写模块配置引用）
- IIS URL 重写模块论坛
- Keep a simple URL structure（保持简单的 URL 结构）
- 10 URL Rewriting Tips and Tricks（10 个 URL 重写提示和技巧）
- To slash or not to slash（用斜杠或不用斜杠）

ASP.NET Core 中的文件提供程序

2018/11/21 • • Edit Online

作者：Steve Smith 和 Luke Latham

ASP.NET Core 通过文件提供程序来抽象化文件系统访问。在 ASP.NET Core 框架中使用文件提供程序：

- IHostingEnvironment 将应用的内容根和 Web 根作为 `IFileProvider` 类型公开。
- 静态文件中间件使用文件提供程序来查找静态文件。
- Razor 使用文件提供程序来查找页面和视图。
- .NET Core 工具使用文件提供程序和 glob 模式来指定应该发布哪些文件。

查看或下载示例代码（如何下载）

文件提供程序接口

主接口为 IFileProvider。`IFileProvider` 公开方法以实现以下目的：

- 获取文件信息 (IFileInfo)。
- 获取目录信息 (IDirectoryContents)。
- 设置更改通知（使用 IChangeToken）。

`IFileInfo` 提供用于处理文件的方法和属性：

- Exists
- IsDirectory
- 名称
- Length（以字节为单位）
- LastModified 日期

可以使用 IFileInfo.CreateReadStream 方法从文件读取内容。

示例应用演示了如何在 `Startup.ConfigureServices` 中配置文件提供程序，以便通过依赖关系注入在应用中使用。

文件提供程序实现

`IFileProvider` 有三种实现。

实现	描述
PhysicalFileProvider	物理提供程序用于访问系统的物理文件。
ManifestEmbeddedFileProvider	清单嵌入式提供程序用于访问程序集中嵌入的文件。
CompositeFileProvider	复合式提供程序提供对一个或多个其他提供程序中的文件和目录的合并访问。

实现	描述
PhysicalFileProvider	物理提供程序用于访问系统的物理文件。

实现	描述
EmbeddedFileProvider	嵌入式提供程序用于访问嵌入在程序集中的文件。
CompositeFileProvider	复合式提供程序提供对一个或多个其他提供程序中的文件和目录的合并访问。

PhysicalFileProvider

PhysicalFileProvider 提供对物理文件系统的访问。`PhysicalFileProvider` 使用(物理提供程序的)System.IO.File 类型并将所有路径范围限制到目录及其子目录。此范围界定可防止访问指定目录和子目录之外的文件系统。实例化此提供程序时，必须提供目录路径并将其作为使用提供程序发出的所有请求的基路径。可以直接实例化 `PhysicalFileProvider` 提供程序，也可以通过依赖关系注入请求构造函数中的 `IFileProvider`。

静态类型

下面的代码演示如何创建 `PhysicalFileProvider` 并用它来获取目录内容和文件信息：

```
var provider = new PhysicalFileProvider(applicationRoot);
var contents = provider.GetDirectoryContents(string.Empty);
var fileInfo = provider.GetFileInfo("wwwroot/js/site.js");
```

前面的示例中的类型：

- `provider` 是 `IFileProvider`。
- `contents` 是 `IDirectoryContents`。
- `fileInfo` 是 `IFileInfo`。

文件提供程序可用于循环访问 `applicationRoot` 指定的目录或调用 `GetFileInfo` 来获取文件信息。该文件提供程序无法访问 `applicationRoot` 目录外部的任何内容。

示例应用使用 IHostingEnvironment.ContentRootFileProvider 在应用的 `Startup.ConfigureServices` 类中创建提供程序：

```
var physicalProvider = _env.ContentRootFileProvider;
```

使用依赖关系注入获取文件提供程序类型

将提供程序注入任何类构造函数，并将其分配给本地字段。在整个类的方法中使用该字段来访问文件。

在示例应用中，`IndexModel` 类接收 `IFileProvider` 实例，获取应用的基路径的目录内容。

Pages/Index.cshtml.cs：

```
public class IndexModel : PageModel
{
    private readonly IFileProvider _fileProvider;

    public IndexModel(IFileProvider fileProvider)
    {
        _fileProvider = fileProvider;
    }

    public IDirectoryContents DirectoryContents { get; private set; }

    public void OnGet()
    {
        DirectoryContents = _fileProvider.GetDirectoryContents(string.Empty);
    }
}
```

在页中循环访问 `IDirectoryContents`。

Pages/Index.cshtml：

```
<ul>
    @foreach (var item in Model.DirectoryContents)
    {
        if (item.IsDirectory)
        {
            <li><strong>@item.Name</strong></li>
        }
        else
        {
            <li>@item.Name - @item.Length bytes</li>
        }
    }
</ul>
```

在示例应用中，`HomeController` 类接收 `IFileProvider` 实例，获取应用的基路径的目录内容。

Controllers/HomeController.cs：

```
public class HomeController : Controller
{
    private readonly IFileProvider _fileProvider;

    public HomeController(IFileProvider fileProvider)
    {
        _fileProvider = fileProvider;
    }

    public IActionResult Index()
    {
        var contents = _fileProvider.GetDirectoryContents(string.Empty);

        return View(contents);
    }
}
```

在视图中循环访问 `IDirectoryContents`。

Views/Home/Index.cshtml：

```
<ul>
    @foreach (IFileInfo item in Model)
    {
        if (item.IsDirectory)
        {
            <li><strong>@item.Name</strong></li>
        }
        else
        {
            <li>@item.Name - @item.Length bytes</li>
        }
    }
</ul>
```

ManifestEmbeddedFileProvider

ManifestEmbeddedFileProvider 用于访问程序集内嵌入的文件。`ManifestEmbeddedFileProvider` 使用编译到程序集中的某个清单来重建嵌入的文件的原始路径。

> **NOTE**
> ASP.NET Core 2.1 或更高版本中提供了 `ManifestEmbeddedFileProvider`。若要访问在 ASP.NET Core 2.0 或更早版本 的程序集中嵌入的文件，请参阅本主题的 ASP.NET Core 1.x 版本。

若要生成嵌入的文件清单，请将 `<GenerateEmbeddedFilesManifest>` 属性设置为 `true`。指定要使用 <EmbeddedResource> 嵌入的文件：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

  <ItemGroup>
    <EmbeddedResource Include="Resource.txt" />
  </ItemGroup>

</Project>
```

使用 glob 模式指定要嵌入到程序集中的一个或多个文件。

示例应用创建 `ManifestEmbeddedFileProvider` 并将当前正在执行的程序集传递给其构造函数。

Startup.cs：

```
var manifestEmbeddedProvider =
    new ManifestEmbeddedFileProvider(Assembly.GetEntryAssembly());
```

其他重载使你能够：

- 指定相对文件路径。
- 将文件范围限制到上次修改日期。
- 为包含嵌入文件清单的嵌入资源命名。

重载	描述
ManifestEmbeddedFileProvider(Assembly, String)	接受一个可选的 `root` 相对路径参数。指定 `root` 将对 GetDirectoryContents 的调用范围限制为提供的路径下的那些资源。
ManifestEmbeddedFileProvider(Assembly, String, DateTimeOffset)	接受一个可选的 `root` 相对路径参数和一个 `lastModified` 日期 (DateTimeOffset) 参数。`lastModified` 日期限制 IFileProvider 返回的 IFileInfo 实例的上次修改日期范围。
ManifestEmbeddedFileProvider(Assembly, String, String, DateTimeOffset)	接受一个可选的 `root` 相对路径、`lastModified` 日期和 `manifestName` 参数。`manifestName` 表示包含清单的嵌入资源的名称。

EmbeddedFileProvider

EmbeddedFileProvider用于访问程序集内嵌入的文件。指定要使用项目文件中的 <EmbeddedResource> 属性嵌入的文件：

```
<ItemGroup>
  <EmbeddedResource Include="Resource.txt" />
</ItemGroup>
```

使用 glob 模式指定要嵌入到程序集中的一个或多个文件。

示例应用创建 `EmbeddedFileProvider` 并将当前正在执行的程序集传递给其构造函数。

Startup.cs：

```
var embeddedProvider = new EmbeddedFileProvider(Assembly.GetEntryAssembly());
```

嵌入的资源不会公开目录。而是使用 `.` 分隔符将指向资源的路径（通过其命名空间）嵌入其文件名中。在示例应用中，`baseNamespace` 是 `FileProviderSample.`。

EmbeddedFileProvider(Assembly, String) 构造函数接受一个可选的 `baseNamespace` 参数。指定基命名空间，将对 GetDirectoryContents 的调用范围限制为提供的命名空间下的那些资源。

CompositeFileProvider

CompositeFileProvider 结合了 `IFileProvider` 实例，以便公开一个接口来处理来自多个提供程序的文件。创建 `CompositeFileProvider` 时，将一个或多个 `IFileProvider` 实例传递给其构造函数。

在示例应用中，`PhysicalFileProvider` 和 `ManifestEmbeddedFileProvider` 向在应用的服务容器中注册的 `CompositeFileProvider` 提供文件：

```
var physicalProvider = _env.ContentRootFileProvider;
var manifestEmbeddedProvider =
    new ManifestEmbeddedFileProvider(Assembly.GetEntryAssembly());
var compositeProvider =
    new CompositeFileProvider(physicalProvider, manifestEmbeddedProvider);

services.AddSingleton<IFileProvider>(compositeProvider);
```

在示例应用中，`PhysicalFileProvider` 和 `EmbeddedFileProvider` 向在应用的服务容器中注册的 `CompositeFileProvider` 提供文件：

```
var physicalProvider = _hostingEnvironment.ContentRootFileProvider;
var embeddedProvider = new EmbeddedFileProvider(Assembly.GetEntryAssembly());
var compositeProvider = new CompositeFileProvider(physicalProvider, embeddedProvider);

services.AddSingleton<IFileProvider>(compositeProvider);
```

监视更改

IFileProvider.Watch 方法提供了一个方案来监视一个或多个文件或目录是否发生更改。`Watch` 接受路径字符串，它可以使用 glob 模式指定多个文件。`Watch` 返回 IChangeToken。更改令牌会公开以下内容：

- HasChanged：可检查此属性以确定是否已发生更改。
- RegisterChangeCallback：检测到指定的路径字符串发生更改时调用此属性。每个更改令牌仅调用其关联的回调来响应单个更改。若要启用持续监视，请使用 TaskCompletionSource（如下所示）或重新创建 `IChangeToken` 实例以响应更改。

在示例应用中，WatchConsole 控制台应用配置为每次修改了文本文件时显示一条消息：

```
private static PhysicalFileProvider _fileProvider =
    new PhysicalFileProvider(Directory.GetCurrentDirectory());

public static void Main(string[] args)
{
    Console.WriteLine("Monitoring quotes.txt for changes (Ctrl-c to quit)...");

    while (true)
    {
        MainAsync().GetAwaiter().GetResult();
    }
}

private static async Task MainAsync()
{
    IChangeToken token = _fileProvider.Watch("quotes.txt");
    var tcs = new TaskCompletionSource<object>();

    token.RegisterChangeCallback(state =>
        ((TaskCompletionSource<object>)state).TrySetResult(null), tcs);

    await tcs.Task.ConfigureAwait(false);

    Console.WriteLine("quotes.txt changed");
}
```

```
private static PhysicalFileProvider _fileProvider =
    new PhysicalFileProvider(Directory.GetCurrentDirectory());

public static void Main(string[] args)
{
    Console.WriteLine("Monitoring quotes.txt for changes (Ctrl-c to quit)...");

    while (true)
    {
        MainAsync().GetAwaiter().GetResult();
    }
}

private static async Task MainAsync()
{
    IChangeToken token = _fileProvider.Watch("quotes.txt");
    var tcs = new TaskCompletionSource<object>();

    token.RegisterChangeCallback(state =>
        ((TaskCompletionSource<object>)state).TrySetResult(null), tcs);

    await tcs.Task.ConfigureAwait(false);

    Console.WriteLine("quotes.txt changed");
}
```

某些文件系统(例如 Docker 容器和网络共享)可能无法可靠地发送更改通知。将 `DOTNET_USE_POLLING_FILE_WATCHER` 环境变量设置为 `1` 或 `true` 以每隔四秒轮询一次文件系统，确认是否发生更改(不可配置)。

glob 模式

文件系统路径使用名为 glob(或通配)模式的通配符模式。使用这些模式指定文件的组。两个通配符分别是 `*` 和 `**`:

`*`
匹配当前文件夹级别的任何内容、任何文件名或任何文件扩展名。匹配由文件路径中的 `/` 和 `.` 字符终止。

`**`
匹配多个目录级别的任何内容。可用于以递归方式匹配目录层次结构中的许多文件。

glob 模式示例

`directory/file.txt`
匹配特定目录中的特定文件。

`directory/*.txt`
匹配特定目录中带 .txt 扩展名的所有文件。

`directory/*/appsettings.json`
匹配正好位于"目录"文件夹中下一级目录中的所有 `appsettings.json` 文件。

`directory/**/*.txt`
匹配在"目录"文件夹下任何位置找到的带 .txt 扩展名的所有文件。

ASP.NET Core 中的请求功能

2018/6/25 • • Edit Online

作者：Steve Smith

与 HTTP 请求和响应相关的 Web 服务器实现详细信息在接口中定义。服务器实现和中间件使用这些接口来创建和修改应用程序的托管管道。

功能接口

ASP.NET Core 在 `Microsoft.AspNetCore.Http.Features` 中定义了许多 HTTP 功能接口，服务器使用这些接口来标识其支持的功能。以下功能接口处理请求并返回响应：

`IHttpRequestFeature` 定义 HTTP 请求的结构，包括协议、路径、查询字符串、标头和正文。

`IHttpResponseFeature` 定义 HTTP 响应的结构，包括状态代码、标头和响应的正文。

`IHttpAuthenticationFeature` 定义支持基于 `ClaimsPrincipal` 来标识用户并指定身份验证处理程序。

`IHttpUpgradeFeature` 定义对 HTTP 升级的支持，允许客户端指定在服务器需要切换协议时要使用的其他协议。

`IHttpBufferingFeature` 定义禁用请求和/或响应缓冲的方法。

`IHttpConnectionFeature` 为本地和远程地址以及端口定义属性。

`IHttpRequestLifetimeFeature` 定义支持中止连接，或者检测是否已提前终止请求（如由于客户端断开连接）。

`IHttpSendFileFeature` 定义异步发送文件的方法。

`IHttpWebSocketFeature` 定义支持 Web 套接字的 API。

`IHttpRequestIdentifierFeature` 添加一个可以实现的属性来唯一标识请求。

`ISessionFeature` 为支持用户会话定义 `ISessionFactory` 和 `ISession` 抽象。

`ITlsConnectionFeature` 定义用于检索客户端证书的 API。

`ITlsTokenBindingFeature` 定义使用 TLS 令牌绑定参数的方法。

> **NOTE**
>
> `ISessionFeature` 不是服务器功能，而是由 `SessionMiddleware` 实现（请参阅管理应用程序状态）。

功能集合

`HttpContext` 的 `Features` 属性为获取和设置当前请求的可用 HTTP 功能提供了一个接口。由于功能集合即使在请求的上下文中也是可变的，所以可使用中间件来修改集合并添加对其他功能的支持。

中间件和请求功能

虽然服务器负责创建功能集合，但中间件既可以添加到该集合中，也可以使用集合中的功能。例如，`StaticFileMiddleware` 访问 `IHttpSendFileFeature` 功能。如果该功能存在，则用于从其物理路径发送所请求的静态文件。否则，使用较慢的替代方法来发送文件。如果可用，`IHttpSendFileFeature` 允许操作系统打开文件并执行直接内核模式复制到网卡。

另外，中间件可以添加到由服务器建立的功能集合中。中间件甚至可以取代现有的功能，以便增加服务器的功能。添加到集合中的功能稍后将在请求管道中立即用于其他中间件或基础应用程序本身。

通过结合自定义服务器实现和特定的中间件增强功能，可构造应用程序所需的精确功能集。这样一来，无需更改服务器即可添加缺少的功能，并确保只公开最少的功能，从而限制攻击外围应用并提高性能。

总结

功能接口定义给定请求可能支持的特定 HTTP 功能。服务器定义功能的集合，以及该服务器支持的初始功能集，但中间件可用于增强这些功能。

其他资源

- 服务器
- 中间件
- .NET 的开放 Web 接口 (OWIN)

在 ASP.NET Core 中访问 HttpContext

2018/8/2 • • Edit Online

ASP.NET Core 应用通过 IHttpContextAccessor 接口及其默认实现 HttpContextAccessor 来访问 `HttpContext`。只有在需要访问服务内的 `HttpContext` 时，才有必要使用 `IHttpContextAccessor`。

通过 Razor Pages 使用 HttpContext

Razor Pages PageModel 公开 HttpContext 属性：

```
public class AboutModel : PageModel
{
    public string Message { get; set; }

    public void OnGet()
    {
        Message = HttpContext.Request.PathBase;
    }
}
```

通过 Razor 视图使用 HttpContext

Razor 视图通过视图上的 RazorPage.Context 属性直接公开 `HttpContext`。下面的示例使用 Windows 身份验证检索 Intranet 应用中的当前用户名：

```
@{
    var username = Context.User.Identity.Name;
}
```

通过控制器使用 HttpContext

控制器公开 ControllerBase.HttpContext 属性：

```
public class HomeController : Controller
{
    public IActionResult About()
    {
        var pathBase = HttpContext.Request.PathBase;
        // Do something with the PathBase.

        return View();
    }
}
```

通过中间件使用 HttpContext

使用自定义中间件组件时，`HttpContext` 传递到 `Invoke` 或 `InvokeAsync` 方法，在中间件配置后可供访问：

```
public class MyCustomMiddleware
{
    public Task InvokeAsync(HttpContext context)
    {
        // Middleware initialization optionally using HttpContext
    }
}
```

通过自定义组件使用 HttpContext

对于需要访问 `HttpContext` 的其他框架和自定义组件，建议使用内置的依赖项注入容器来注册依赖项。依赖项注入容器向任意类提供 `IHttpContextAccessor`，以供类在自己的构造函数中将它声明为依赖项。

```
public void ConfigureServices(IServiceCollection services)
{
     services.AddMvc()
         .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
     services.AddHttpContextAccessor();
     services.AddTransient<IUserRepository, UserRepository>();
}
```

```
public void ConfigureServices(IServiceCollection services)
{
     services.AddMvc();
     services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
     services.AddTransient<IUserRepository, UserRepository>();
}
```

在上面的示例中：

- `UserRepository` 声明自己对 `IHttpContextAccessor` 的依赖。
- 当依赖项注入容器解析依赖链并创建 `UserRepository` 实例时，就会注入依赖项。

```
public class UserRepository : IUserRepository
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    public UserRepository(IHttpContextAccessor httpContextAccessor)
    {
        _httpContextAccessor = httpContextAccessor;
    }

    public void LogCurrentUser()
    {
        var username = _httpContextAccessor.HttpContext.User.Identity.Name;
        service.LogAccessRequest(username);
    }
}
```

使用 ASP.NET Core 中的更改令牌检测更改

2018/10/30 • • Edit Online

作者：Luke Latham

更改令牌是用于跟踪更改的通用、低级别构建基块。

查看或下载示例代码（如何下载）

IChangeToken 接口

IChangeToken 传播已发生更改的通知。`IChangeToken` 位于 Microsoft.Extensions.Primitives 命名空间中。对于不使用 Microsoft.AspNetCore.App 元包（ASP.NET Core 2.1 及更高版本）的应用，将在项目文件中引用 Microsoft.Extensions.Primitives NuGet 包。

`IChangeToken` 具有以下两个属性：

- ActiveChangedCallbacks，指示令牌是否主动引发回调。如果将 `ActiveChangedCallbacks` 设置为 `false`，则不会调用回调，并且应用必须轮询 `HasChanged` 获取更改。如果未发生任何更改，或者丢弃或禁用基础更改侦听器，还可能永远不会取消令牌。
- HasChanged，获取一个指示是否发生更改的值。

此接口具有一种方法，即 RegisterChangeCallback(Action<Object>, Object)，用于注册在令牌更改时调用的回调。调用回调之前，必须设置 `HasChanged`。

ChangeToken 类

`ChangeToken` 是静态类，用于传播已发生更改的通知。`ChangeToken` 位于 Microsoft.Extensions.Primitives 命名空间中。对于不使用 Microsoft.AspNetCore.App 元包的应用，将在项目文件中引用 Microsoft.Extensions.Primitives NuGet 包。

`ChangeToken` OnChange(Func<IChangeToken>, Action) 方法注册令牌更改时要调用的 `Action`：

- `Func<IChangeToken>` 生成令牌。
- 令牌更改时，调用 `Action`。

`ChangeToken` 具有 OnChange<TState>(Func<IChangeToken>, Action<TState>, TState) 重载，它接受传递到令牌使用者 `Action` 的附加 `TState` 参数。

`OnChange` 返回 IDisposable。调用 Dispose 将使令牌停止侦听更多更改并释放令牌的资源。

ASP.NET Core 中更改令牌的使用示例

更改令牌主要用于 ASP.NET Core 监视对象更改：

- 为了监视文件更改，IFileProvider 的 Watch 方法将为要监视的指定文件或文件夹创建 `IChangeToken`。
- 可以将 `IChangeToken` 令牌添加到缓存条目，以在更改时触发缓存逐出。
- 对于 `TOptions` 更改，IOptionsMonitor 的默认 OptionsMonitor 实现具有重载，该重载接受一个或多个 IOptionsChangeTokenSource 实例。每个实例返回 `IChangeToken`，以注册用于跟踪选项更改的更改通知回调。

监视配置更改

默认情况下，ASP.NET Core 模板使用 JSON 配置文件（appsettings.json、appsettings.Development.json 和 appsettings.Production.json）来加载应用配置设置。

使用接受 `reloadOnChange` 参数（ASP.NET Core 1.1 以及更高版本）的 ConfigurationBuilder 上的 AddJsonFile(IConfigurationBuilder, String, Boolean, Boolean) 扩展方法配置这些文件。`reloadOnChange` 指示文件更改时是否应该重载配置。请参阅 WebHost 便捷方法 CreateDefaultBuilder 中的此设置：

```
config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
      .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);
```

基于文件的配置由 FileConfigurationSource 表示。`FileConfigurationSource` 使用 IFileProvider 监视文件。

默认情况下，由使用 FileSystemWatcher 来监视配置文件更改的 PhysicalFileProvider 提供 `IFileMonitor`。

示例应用演示监视配置更改的两个实现。如果 appsettings.json 文件更改或者文件的 Environment 版本更改，则每个实现执行自定义代码。示例应用向控制台写入消息。

配置文件的 `FileSystemWatcher` 可以触发多个令牌回调，以用于单个配置文件更改。该示例的实现通过检查配置文件上的文件哈希来防范此问题。检查文件哈希可确保在运行自定义代码之前，至少已更改了其中一个配置文件。本示例使用 SHA1 文件哈希 (Utilities/Utilities.cs)：

```
public static byte[] ComputeHash(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fs = File.OpenRead(filePath))
                {
                    return System.Security.Cryptography.SHA1.Create().ComputeHash(fs);
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3 || ex.HResult != -2147024864)
            {
                throw;
            }
            else
            {
                Thread.Sleep(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
                runCount++;
            }
        }
    }

    return new byte[20];
}
```

使用指数回退实现重试。进行重试是因为文件锁定可能暂时阻止对其中一个文件计算新的哈希。

简单启动更改令牌

将用于更改通知的令牌使用者 `Action` 回调注册到配置重载令牌 (Startup.cs)：

```
ChangeToken.OnChange(
    () => config.GetReloadToken(),
    (state) => InvokeChanged(state),
    env);
```

`config.GetReloadToken()` 提供令牌。回调是 `InvokeChanged` 方法：

```
private void InvokeChanged(IHostingEnvironment env)
{
    byte[] appsettingsHash = ComputeHash("appSettings.json");
    byte[] appsettingsEnvHash =
        ComputeHash($"appSettings.{env.EnvironmentName}.json");

    if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
        !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
    {
        _appsettingsHash = appsettingsHash;
        _appsettingsEnvHash = appsettingsEnvHash;

        WriteConsole("Configuration changed (Simple Startup Change Token)");
    }
}
```

回调的 `state` 用于在 `IHostingEnvironment` 中传递。这有助于确定要监视的正确 appsettings 配置 JSON 文件，即 appsettings.<Environment>.json。只更改了一次配置文件时，文件哈希用于防止 `WriteConsole` 语句由于多个令牌回调而多次运行。

只要应用正在运行，该系统就会运行，并且用户不能禁用。

将配置更改作为服务进行监视

示例实现：

- 基本启动令牌监视。
- 作为服务监视。
- 启用和禁用监视的机制。

示例建立 `IConfigurationMonitor` 接口 (Extensions/ConfigurationMonitor.cs)：

```
public interface IConfigurationMonitor
{
    bool MonitoringEnabled { get; set; }
    string CurrentState { get; set; }
}
```

已实现的类的构造函数 `ConfigurationMonitor` 注册用于更改通知的回调：

```
public ConfigurationMonitor(IConfiguration config, IHostingEnvironment env)
{
    _env = env;

    ChangeToken.OnChange<IConfigurationMonitor>(
        () => config.GetReloadToken(),
        InvokeChanged,
        this);
}

public bool MonitoringEnabled { get; set; } = false;
public string CurrentState { get; set; } = "Not monitoring";
```

`config.GetReloadToken()` 提供令牌。`InvokeChanged` 是回调方法。此实例中的 `state` 是对 `IConfigurationMonitor` 实例的引用，用于访问监视状态。使用了以下两个属性：

- `MonitoringEnabled`，指示回调是否应该运行其自定义代码。
- `CurrentState`，描述在 UI 中使用的当前监视状态。

`InvokeChanged` 方法类似于前面的方法，不同之处在于：

- 不运行其代码，除非 `MonitoringEnabled` 为 `true`。
- 注意其 `WriteConsole` 输出中的当前 `state`。

```
private void InvokeChanged(IConfigurationMonitor state)
{
    if (MonitoringEnabled)
    {
        byte[] appsettingsHash = ComputeHash("appSettings.json");
        byte[] appsettingsEnvHash =
            ComputeHash($"appSettings.{_env.EnvironmentName}.json");

        if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
            !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
        {
            string message = $"State updated at {DateTime.Now}";


            _appsettingsHash = appsettingsHash;
            _appsettingsEnvHash = appsettingsEnvHash;

            WriteConsole($"Configuration changed (ConfigurationMonitor Class) {message}, state:
{state.CurrentState}");
        }
    }
}
```

将实例 `ConfigurationMonitor` 作为服务注册在 Startup.cs 的 `ConfigureServices` 中：

```
services.AddSingleton<IConfigurationMonitor, ConfigurationMonitor>();
```

索引页提供对配置监视的用户控制。将 `IConfigurationMonitor` 的实例注入到 `IndexModel` 中：

```
public IndexModel(
    IConfiguration config,
    IConfigurationMonitor monitor,
    FileService fileService)
{
    _config = config;
    _monitor = monitor;
    _fileService = fileService;
}
```

按钮启用和禁用监视：

```
<button class="btn btn-danger" asp-page-handler="StopMonitoring">Stop Monitoring</button>
```

```
public IActionResult OnPostStartMonitoring()
{
    _monitor.MonitoringEnabled = true;
    _monitor.CurrentState = "Monitoring!";

    return RedirectToPage();
}

public IActionResult OnPostStopMonitoring()
{
    _monitor.MonitoringEnabled = false;
    _monitor.CurrentState = "Not monitoring";

    return RedirectToPage();
}
```

触发 `OnPostStartMonitoring` 时，会启用监视并清除当前状态。触发 `OnPostStopMonitoring` 时，会禁用监视并设置状态以反映未进行监视。

监视缓存文件更改

可以使用 IMemoryCache 将文件内容缓存在内存中。内存中缓存主题中介绍了在内存中缓存。无需采取其他步骤（如下面所述的实现），如果源数据更改，将从缓存返回陈旧（过时）数据。

当续订可调过期时段造成陈旧缓存数据时，不考虑所缓存源文件的状态。数据的每个请求续订可调过期时段，但不会将文件重载到缓存中。任何使用文件缓存内容的应用功能都可能会收到陈旧的内容。

在文件缓存方案中使用更改令牌可防止缓存中出现陈旧的文件内容。示例应用演示方法的实现。

示例使用 `GetFileContent` 来完成以下操作：

- 返回文件内容。
- 实现具有指数回退的重试算法，以涵盖文件锁暂时阻止读取文件的情况。

Utilities/Utilities.cs：

```
public async static Task<string> GetFileContent(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fileStreamReader = File.OpenText(filePath))
                {
                    return await fileStreamReader.ReadToEndAsync();
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3 || ex.HResult != -2147024864)
            {
                throw;
            }
            else
            {
                await Task.Delay(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
                runCount++;
            }
        }
    }

    return null;
}
```

创建 `FileService` 以处理缓存文件查找。服务的 `GetFileContent` 方法调用尝试从内存中缓存获取文件内容并将其返回到调用方 (Services/FileService.cs)。

如果使用缓存键未找到缓存的内容，则将执行以下操作：

1. 使用 `GetFileContent` 获取文件内容。
2. 使用 IFileProviders.Watch 从文件提供程序获取更改令牌。修改该文件时，会触发令牌的回调。
3. 可调过期时段将缓存文件内容。如果缓存文件时发生了更改，则将更改令牌与 MemoryCacheEntryExtensions.AddExpirationToken 连接在一起，以逐出缓存条目。

```
public class FileService
{
    private readonly IMemoryCache _cache;
    private readonly IFileProvider _fileProvider;
    private List<string> _tokens = new List<string>();

    public FileService(IMemoryCache cache, IHostingEnvironment env)
    {
        _cache = cache;
        _fileProvider = env.ContentRootFileProvider;
    }

    public async Task<string> GetFileContents(string fileName)
    {
        // For the purposes of this example, files are stored
        // in the content root of the app. To obtain the physical
        // path to a file at the content root, use the
        // ContentRootFileProvider on IHostingEnvironment.
        var filePath = _fileProvider.GetFileInfo(fileName).PhysicalPath;
        string fileContent;

        // Try to obtain the file contents from the cache.
        if (_cache.TryGetValue(filePath, out fileContent))
        {
            return fileContent;
        }

        // The cache doesn't have the entry, so obtain the file
        // contents from the file itself.
        fileContent = await GetFileContent(filePath);

        if (fileContent != null)
        {
            // Obtain a change token from the file provider whose
            // callback is triggered when the file is modified.
            var changeToken = _fileProvider.Watch(fileName);

            // Configure the cache entry options for a five minute
            // sliding expiration and use the change token to
            // expire the file in the cache if the file is
            // modified.
            var cacheEntryOptions = new MemoryCacheEntryOptions()
                .SetSlidingExpiration(TimeSpan.FromMinutes(5))
                .AddExpirationToken(changeToken);

            // Put the file contents into the cache.
            _cache.Set(filePath, fileContent, cacheEntryOptions);

            return fileContent;
        }

        return string.Empty;
    }
}
```

将 `FileService` 和内存缓存服务 (Startup.cs) 一起注册在服务容器中：

```
services.AddMemoryCache();
services.AddSingleton<FileService>();
```

页面模型使用服务 (Pages/Index.cshtml.cs) 加载文件内容：

```
var fileContent = await _fileService.GetFileContents("poem.txt");
```

CompositeChangeToken 类

要在单个对象中表示一个或多个 `IChangeToken` 实例，请使用 CompositeChangeToken 类。

```
var firstCancellationTokenSource = new CancellationTokenSource();
var secondCancellationTokenSource = new CancellationTokenSource();

var firstCancellationToken = firstCancellationTokenSource.Token;
var secondCancellationToken = secondCancellationTokenSource.Token;

var firstCancellationChangeToken = new CancellationChangeToken(firstCancellationToken);
var secondCancellationChangeToken = new CancellationChangeToken(secondCancellationToken);

var compositeChangeToken =
    new CompositeChangeToken(
        new List<IChangeToken>
        {
            firstCancellationChangeToken,
            secondCancellationChangeToken
        });
```

如果任何表示的令牌 `HasChanged` 为 `true`，则复合令牌上的 `HasChanged` 报告 `true`。如果任何表示的令牌 `ActiveChangeCallbacks` 为 `true`，则复合令牌上的 `ActiveChangeCallbacks` 报告 `true`。如果发生多个并发更改事件，则调用一次复合更改回调。

其他资源

- 缓存在内存中 ASP.NET Core
- 分布式缓存在 ASP.NET Core 中
- 响应缓存在 ASP.NET Core
- 响应缓存在 ASP.NET Core 中的中间件
- ASP.NET Core MVC 中的缓存标记帮助程序
- ASP.NET Core 中的分布式缓存标记帮助程序

ASP.NET Core 中 .NET 的开放 Web 接口 (OWIN)

2019/1/10 • • Edit Online

作者：Steve Smith 和 Rick Anderson

ASP.NET Core 支持 .NET 的开放 Web 接口 (OWIN)。OWIN 允许 Web 应用从 Web 服务器分离。它定义了在管道中使用中间件来处理请求和相关响应的标准方法。ASP.NET Core 应用程序和中间件可以与基于 OWIN 的应用程序、服务器和中间件进行互操作。

OWIN 提供了一个分离层，可一起使用具有不同对象模型的两个框架。`Microsoft.AspNetCore.Owin` 包提供了两个适配器实现：

- ASP.NET Core 到 OWIN
- OWIN 到 ASP.NET Core

此方法可将 ASP.NET Core 托管在兼容 OWIN 的服务器/主机上，或在 ASP.NET Core 上运行其他兼容 OWIN 的组件。

> **NOTE**
> 使用这些适配器会带来性能成本。仅使用 ASP.NET Core 组件的应用不应使用 `Microsoft.AspNetCore.Owin` 包或适配器。

查看或下载示例代码（如何下载）

在 ASP.NET Core 管道中运行 OWIN 中间件

ASP.NET Core 的 OWIN 支持作为 `Microsoft.AspNetCore.Owin` 包的一部分进行部署。可通过安装此包将 OWIN 支持导入到项目中。

OWIN 中间件符合 OWIN 规范，该规范要求使用 `Func<IDictionary<string, object>, Task>` 接口，并设置特定的键（如 `owin.ResponseBody`）。以下简单的 OWIN 中间件显示“Hello World”：

```
public Task OwinHello(IDictionary<string, object> environment)
{
    string responseText = "Hello World via OWIN";
    byte[] responseBytes = Encoding.UTF8.GetBytes(responseText);

    // OWIN Environment Keys: http://owin.org/spec/spec/owin-1.0.0.html
    var responseStream = (Stream)environment["owin.ResponseBody"];
    var responseHeaders = (IDictionary<string, string[]>)environment["owin.ResponseHeaders"];

    responseHeaders["Content-Length"] = new string[] {
responseBytes.Length.ToString(CultureInfo.InvariantCulture) };
    responseHeaders["Content-Type"] = new string[] { "text/plain" };

    return responseStream.WriteAsync(responseBytes, 0, responseBytes.Length);
}
```

示例签名返回 `Task`，并接受 OWIN 所要求的 `IDictionary<string, object>`。

以下代码显示了如何使用 `UseOwin` 扩展方法将 `OwinHello` 中间件（如上所示）添加到 ASP.NET Core 管道。

```
public void Configure(IApplicationBuilder app)
{
    app.UseOwin(pipeline =>
    {
        pipeline(next => OwinHello);
    });
}
```

可配置在 OWIN 管道中要进行的其他操作。

> **NOTE**
>
> 响应标头只能在首次写入响应流之前进行修改。

> **NOTE**
>
> 由于性能原因，不建议多次调用 `UseOwin`。组合在一起时 OWIN 组件的性能最佳。

```
app.UseOwin(pipeline =>
{
    pipeline(async (next) =>
    {
        // do something before
        await OwinHello(new OwinEnvironment(HttpContext));
        // do something after
    });
});
```

在基于 OWIN 的服务器中使用 ASP.NET Core 托管

基于 OWIN 的服务器可托管 ASP.NET Core 应用。Nowin(.NET OWIN Web 服务器)属于这种服务器。本文的示例中包含了一个引用 Nowin 的项目，并使用它创建了一个自托管 ASP.NET Core 的 `IServer`。

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;

namespace NowinSample
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseNowin()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}
```

`IServer` 是需要 `Features` 属性和 `Start` 方法的接口。

`Start` 负责配置和启动服务器，在此情况下，此操作通过一系列 Fluent API 调用完成，这些调用设置从 IServerAddressesFeature 分析的地址。请注意，`_builder` 变量的 Fluent 配置指定请求将由方法中之前定义的 `appFunc` 来处理。对于每个请求，都会调用此 `Func` 以处理传入请求。

我们还将添加一个 `IWebHostBuilder` 扩展，以便添加和配置 Nowin 服务器。

```
using System;
using Microsoft.AspNetCore.Hosting.Server;
using Microsoft.Extensions.DependencyInjection;
using Nowin;
using NowinSample;

namespace Microsoft.AspNetCore.Hosting
{
    public static class NowinWebHostBuilderExtensions
    {
        public static IWebHostBuilder UseNowin(this IWebHostBuilder builder)
        {
            return builder.ConfigureServices(services =>
            {
                services.AddSingleton<IServer, NowinServer>();
            });
        }

        public static IWebHostBuilder UseNowin(this IWebHostBuilder builder, Action<ServerBuilder> configure)
        {
            builder.ConfigureServices(services =>
            {
                services.Configure(configure);
            });
            return builder.UseNowin();
        }
    }
}
```

完成此操作后，调用 Program.cs 中的扩展以使用此自定义服务器运行 ASP.NET Core 应用：

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;

namespace NowinSample
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseNowin()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}
```

了解有关 ASP.NET Core 服务器的更多信息。

在基于 OWIN 的服务器上运行 ASP.NET Core 并使用其 WebSocket 支持

ASP.NET Core 如何利用基于 OWIN 的服务器功能的另一个示例是访问 WebSocket 等功能。前面示例中使用的 .NET OWIN Web 服务器支持内置的 Web 套接字，可由 ASP.NET Core 应用程序利用。下面的示例显示了简单的 Web 应用，它支持 Web 套接字并回显通过 WebSocket 发送到服务器的所有内容。

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            if (context.WebSockets.IsWebSocketRequest)
            {
                WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync();
                await EchoWebSocket(webSocket);
            }
            else
            {
                await next();
            }
        });

        app.Run(context =>
        {
            return context.Response.WriteAsync("Hello World");
        });
    }

    private async Task EchoWebSocket(WebSocket webSocket)
    {
        byte[] buffer = new byte[1024];
        WebSocketReceiveResult received = await webSocket.ReceiveAsync(
            new ArraySegment<byte>(buffer), CancellationToken.None);

        while (!webSocket.CloseStatus.HasValue)
        {
            // Echo anything we receive
            await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, received.Count),
                received.MessageType, received.EndOfMessage, CancellationToken.None);

            received = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
                CancellationToken.None);
        }

        await webSocket.CloseAsync(webSocket.CloseStatus.Value,
            webSocket.CloseStatusDescription, CancellationToken.None);
    }
}
```

使用与前一个相同的 `NowinServer` 来配置此示例 - 唯一的区别是如何在其 `Configure` 方法中配置应用程序。使用简单的 websocket 客户端的测试演示应用程序：

OWIN 环境

可使用 `HttpContext` 来构造 OWIN 环境。

```
var environment = new OwinEnvironment(HttpContext);
var features = new OwinFeatureCollection(environment);
```

OWIN 键

OWIN 依赖于 `IDictionary<string,object>` 对象，以在整个 HTTP请求/响应交换中传达信息。ASP.NET Core 实现以下所列的键。请参阅主规范、扩展和 OWIN Key Guidelines and Common Keys（OWIN 键指南和常用键）。

请求数据 (OWIN v1.0.0)

键	值（类型）	说明
owin.RequestScheme	`String`	
owin.RequestMethod	`String`	
owin.RequestPathBase	`String`	
owin.RequestPath	`String`	
owin.RequestQueryString	`String`	
owin.RequestProtocol	`String`	
owin.RequestHeaders	`IDictionary<string,string[]>`	

键	值(类型)	说明
owin.RequestBody	`Stream`	

请求数据 (OWIN v1.1.0)

键	值(类型)	说明
owin.RequestId	`String`	Optional

响应数据 (OWIN v1.0.0)

键	值(类型)	说明
owin.ResponseStatusCode	`int`	Optional
owin.ResponseReasonPhrase	`String`	Optional
owin.ResponseHeaders	`IDictionary<string,string[]>`	
owin.ResponseBody	`Stream`	

其他数据 (OWIN v1.0.0)

键	值(类型)	说明
owin.CallCancelled	`CancellationToken`	
owin.Version	`String`	

常用键

键	值(类型)	说明
ssl.ClientCertificate	`X509Certificate`	
ssl.LoadClientCertAsync	`Func<Task>`	
server.RemoteIpAddress	`String`	
server.RemotePort	`String`	
server.LocalIpAddress	`String`	
server.LocalPort	`String`	
server.IsLocal	`bool`	
server.OnSendingHeaders	`Action<Action<object>,object>`	

SendFiles v0.3.0

键	值(类型)	说明
sendfile.SendAsync	请参阅委托签名	每请求

Opaque v0.3.0

键	值(类型)	说明
opaque.Version	`String`	
opaque.Upgrade	`OpaqueUpgrade`	请参阅委托签名
opaque.Stream	`Stream`	
opaque.CallCancelled	`CancellationToken`	

WebSocket v0.3.0

键	值(类型)	说明
websocket.Version	`String`	
websocket.Accept	`WebSocketAccept`	请参阅委托签名
websocket.AcceptAlt		非规范
websocket.SubProtocol	`String`	请参阅 RFC6455 4.2.2 节步骤 5.5
websocket.SendAsync	`WebSocketSendAsync`	请参阅委托签名
websocket.ReceiveAsync	`WebSocketReceiveAsync`	请参阅委托签名
websocket.CloseAsync	`WebSocketCloseAsync`	请参阅委托签名
websocket.CallCancelled	`CancellationToken`	
websocket.ClientCloseStatus	`int`	Optional
websocket.ClientCloseDescription	`String`	Optional

其他资源

- 中间件
- 服务器

在 ASP.NET Core 中使用托管服务实现后台任务

2018/12/12 • • Edit Online

作者：Luke Latham

在 ASP.NET Core 中，后台任务作为托管服务实现。托管服务是一个类，具有实现 IHostedService 接口的后台任务逻辑。本主题提供了三个托管服务示例：

- 在计时器上运行的后台任务。
- 激活有作用域的服务的托管服务。有作用域的服务可使用依赖项注入。
- 按顺序运行的已排队后台任务。

查看或下载示例代码（如何下载）

此示例应用分为两个版本：

- Web 主机 – Web 主机可用于托管 Web 应用。本主题中所示的示例代码来自示例的 Web 主机版本。有关详细信息，请参阅 Web 主机主题。
- 通用主机 – 通用主机是 ASP.NET Core 2.1 中的新增功能。有关详细信息，请参阅通用主机主题。

Package

引用 Microsoft.AspNetCore.App 元包或将包引用添加到 Microsoft.Extensions.Hosting 包。

IHostedService 接口

托管服务实现 IHostedService 接口。该接口为主机托管的对象定义了两种方法：

- StartAsync(CancellationToken) – `StartAsync` 包含启动后台任务的逻辑。当使用 Web 主机时，会在启动服务器并触发 IApplicationLifetime.ApplicationStarted 后调用 `StartAsync`。当使用通用主机时，会在触发 `ApplicationStarted` 之前调用 `StartAsync`。
- StopAsync(CancellationToken) – 主机正常关闭时触发。`StopAsync` 包含结束后台任务的逻辑。实现 IDisposable 和终结器（析构函数）以处置任何非托管资源。

  默认情况下，取消令牌会有五秒超时，以指示关闭进程不再正常。在令牌上请求取消时：

  - 应中止应用正在执行的任何剩余后台操作。
  - `StopAsync` 中调用的任何方法都应及时返回。

  但是，在请求取消后，将不会放弃任务 — 调用方等待所有任务完成。

  如果应用意外关闭（例如，应用的进程失败），则可能不会调用 `StopAsync`。因此，在 `StopAsync` 中执行的任何方法或操作都可能不会发生。

  若要延长默认值为 5 秒的关闭超时值，请设置：

  - 使用泛型主机时为 ShutdownTimeout。有关更多信息，请参见.NET 通用主机。
  - 使用 Web 主机时为关闭超时值主机配置设置。有关更多信息，请参见ASP.NET Core Web 主机。

托管服务在应用启动时激活一次，在应用关闭时正常关闭。如果在执行后台任务期间引发错误，即使未调用 `StopAsync`，也应调用 `Dispose`。

计时的后台任务

定时后台任务使用 System.Threading.Timer 类。计时器触发任务的 `DoWork` 方法。在 `StopAsync` 上禁用计时器，并在 `Dispose` 上处置服务容器时处置计时器：

```
internal class TimedHostedService : IHostedService, IDisposable
{
    private readonly ILogger _logger;
    private Timer _timer;

    public TimedHostedService(ILogger<TimedHostedService> logger)
    {
        _logger = logger;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("Timed Background Service is starting.");

        _timer = new Timer(DoWork, null, TimeSpan.Zero,
            TimeSpan.FromSeconds(5));

        return Task.CompletedTask;
    }

    private void DoWork(object state)
    {
        _logger.LogInformation("Timed Background Service is working.");
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("Timed Background Service is stopping.");

        _timer?.Change(Timeout.Infinite, 0);

        return Task.CompletedTask;
    }

    public void Dispose()
    {
        _timer?.Dispose();
    }
}
```

已使用 `AddHostedService` 扩展方法在 `Startup.ConfigureServices` 中注册该服务：

```
services.AddHostedService<TimedHostedService>();
```

在后台任务中使用有作用域的服务

要在 `IHostedService` 中使用有作用域的服务，请创建一个作用域。默认情况下，不会为托管服务创建作用域。

作用域后台任务服务包含后台任务的逻辑。在以下示例中，将 ILogger 注入到服务中：

```
internal interface IScopedProcessingService
{
    void DoWork();
}

internal class ScopedProcessingService : IScopedProcessingService
{
    private readonly ILogger _logger;

    public ScopedProcessingService(ILogger<ScopedProcessingService> logger)
    {
        _logger = logger;
    }

    public void DoWork()
    {
        _logger.LogInformation("Scoped Processing Service is working.");
    }
}
```

托管服务创建一个作用域来解决作用域后台任务服务以调用其 `DoWork` 方法：

```
internal class ConsumeScopedServiceHostedService : IHostedService
{
    private readonly ILogger _logger;

    public ConsumeScopedServiceHostedService(IServiceProvider services,
        ILogger<ConsumeScopedServiceHostedService> logger)
    {
        Services = services;
        _logger = logger;
    }

    public IServiceProvider Services { get; }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is starting.");

        DoWork();

        return Task.CompletedTask;
    }

    private void DoWork()
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is working.");

        using (var scope = Services.CreateScope())
        {
            var scopedProcessingService =
                scope.ServiceProvider
                    .GetRequiredService<IScopedProcessingService>();

            scopedProcessingService.DoWork();
        }
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is stopping.");

        return Task.CompletedTask;
    }
}
```

已在 `Startup.ConfigureServices` 中注册这些服务。已使用 `AddHostedService` 扩展方法注册 `IHostedService` 实现：

```
services.AddHostedService<ConsumeScopedServiceHostedService>();
services.AddScoped<IScopedProcessingService, ScopedProcessingService>();
```

排队的后台任务

后台任务队列基于 .NET 4.x QueueBackgroundWorkItem（暂定为 ASP.NET Core 3.0 内置版本）：

```
public interface IBackgroundTaskQueue
{
    void QueueBackgroundWorkItem(Func<CancellationToken, Task> workItem);

    Task<Func<CancellationToken, Task>> DequeueAsync(
        CancellationToken cancellationToken);
}

public class BackgroundTaskQueue : IBackgroundTaskQueue
{
    private ConcurrentQueue<Func<CancellationToken, Task>> _workItems =
        new ConcurrentQueue<Func<CancellationToken, Task>>();
    private SemaphoreSlim _signal = new SemaphoreSlim(0);

    public void QueueBackgroundWorkItem(
        Func<CancellationToken, Task> workItem)
    {
        if (workItem == null)
        {
            throw new ArgumentNullException(nameof(workItem));
        }

        _workItems.Enqueue(workItem);
        _signal.Release();
    }

    public async Task<Func<CancellationToken, Task>> DequeueAsync(
        CancellationToken cancellationToken)
    {
        await _signal.WaitAsync(cancellationToken);
        _workItems.TryDequeue(out var workItem);

        return workItem;
    }
}
```

在 `QueueHostedService` 中，队列中的后台任务会取消排队，并作为 BackgroundService 执行，此类是用于实现长时间运行 `IHostedService` 的基类：

```
public class QueuedHostedService : BackgroundService
{
    private readonly ILogger _logger;

    public QueuedHostedService(IBackgroundTaskQueue taskQueue,
        ILoggerFactory loggerFactory)
    {
        TaskQueue = taskQueue;
        _logger = loggerFactory.CreateLogger<QueuedHostedService>();
    }

    public IBackgroundTaskQueue TaskQueue { get; }

    protected async override Task ExecuteAsync(
        CancellationToken cancellationToken)
    {
        _logger.LogInformation("Queued Hosted Service is starting.");

        while (!cancellationToken.IsCancellationRequested)
        {
            var workItem = await TaskQueue.DequeueAsync(cancellationToken);

            try
            {
                await workItem(cancellationToken);
            }
            catch (Exception ex)
            {
                _logger.LogError(ex,
                   $"Error occurred executing {nameof(workItem)}.");
            }
        }

        _logger.LogInformation("Queued Hosted Service is stopping.");
    }
}
```

已在 `Startup.ConfigureServices` 中注册这些服务。已使用 `AddHostedService` 扩展方法注册 `IHostedService` 实现：

```
services.AddHostedService<QueuedHostedService>();
services.AddSingleton<IBackgroundTaskQueue, BackgroundTaskQueue>();
```

在索引页模型类中，将 `IBackgroundTaskQueue` 注入构造函数并分配给 `Queue`：

```
public IndexModel(IBackgroundTaskQueue queue,
    IApplicationLifetime appLifetime,
    ILogger<IndexModel> logger)
{
    Queue = queue;
    _appLifetime = appLifetime;
    _logger = logger;
}

public IBackgroundTaskQueue Queue { get; }
```

在索引页上选择“添加任务”按钮时，会执行 `OnPostAddTask` 方法。调用 `QueueBackgroundWorkItem` 来将工作项排入队列：

```
public IActionResult OnPostAddTask()
{
    Queue.QueueBackgroundWorkItem(async token =>
    {
        var guid = Guid.NewGuid().ToString();

        for (int delayLoop = 0; delayLoop < 3; delayLoop++)
        {
            _logger.LogInformation(
                $"Queued Background Task {guid} is running. {delayLoop}/3");
            await Task.Delay(TimeSpan.FromSeconds(5), token);
        }

        _logger.LogInformation(
            $"Queued Background Task {guid} is complete. 3/3");
    });

    return RedirectToPage();
}
```

其他资源

- 使用 IHostedService 和 BackgroundService 类在微服务中实现后台任务
- System.Threading.Timer

在 ASP.NET Core 中使用承载启动程序集

2019/1/10 • • Edit Online

作者：Luke Latham 和 Pavel Krymets

通过 IHostingStartup（承载启动）实现，在启动时从外部程序集向应用添加增强功能。例如，外部库可使用承载启动实现为应用提供其他配置提供程序或服务。ASP.NET Core 2.0 或更高版本中提供了 `IHostingStartup`。

查看或下载示例代码（如何下载）

HostingStartup 属性

HostingStartup 属性表示存在要在运行时激活的承载启动程序集。

将自动扫描输入程序集或包含 `Startup` 类的程序集以查找 `HostingStartup` 属性。用于搜索 `HostingStartup` 属性的程序集列表在运行时从 WebHostDefaults.HostingStartupAssembliesKey 中的配置加载。要从发现中排除的程序集列表从 WebHostDefaults.HostingStartupExcludeAssembliesKey 加载。有关详细信息，请参阅 Web 主机：承载启动程序集和 Web 主机：承载启动排除程序集。

在以下示例中，承载启动程序集的命名空间为 `StartupEnhancement`。包含承载启动代码的类是 `StartupEnhancementHostingStartup`：

```
[assembly: HostingStartup(typeof(StartupEnhancement.StartupEnhancementHostingStartup))]
```

`HostingStartup` 属性通常位于承载启动程序集的 `IHostingStartup` 实现类文件中。

发现加载的承载启动程序集

要发现加载的承载启动程序集，请启用日志记录并检查应用的日志。记录加载程序集时发生的错误。会在调试级别记录加载的承载启动程序集，并记录所有错误。

禁用承载启动程序集的自动加载

要禁用承载启动程序集的自动加载，请使用以下方法之一：

- 要阻止加载所有承载启动程序集，请将以下内容之一设置为 `true` 或 `1`：
  - 阻止承载启动主机配置设置。
  - `ASPNETCORE_PREVENTHOSTINGSTARTUP` 环境变量。
- 要阻止加载特定的承载启动程序集，请将以下之一设置为以分号分隔的承载启动程序集字符串，以便在启动时排除：
  - 承载启动排除程序集承载配置设置。
  - `ASPNETCORE_HOSTINGSTARTUPEXCLUDEASSEMBLIES` 环境变量。

要禁用承载启动程序集的自动加载，请将以下之一设置为 `true` 或 `1`：

- 阻止承载启动主机配置设置。
- `ASPNETCORE_PREVENTHOSTINGSTARTUP` 环境变量。

如果同时设置了主机配置设置和环境变量，则主机设置将控制行为。

如果使用主机设置或环境变量来禁用承载启动程序集，将全局禁用程序集，并可能会禁用应用的多个特征。

项目

使用以下任一项目类型创建承载启动：

- 类库
- 无入口点的控制台应用

类库

可在类库中提供承载启动增强。库包含 `HostingStartup` 属性。

示例代码包括 Razor Pages 应用、HostingStartupApp 和类库 HostingStartupLibrary。类库：

- 包含承载启动类 `ServiceKeyInjection`，用于实现 `IHostingStartup`。`ServiceKeyInjection` 使用内存中配置提供程序 (AddInMemoryCollection) 将一对服务字符串添加到应用的配置中。
- 包含 `HostingStartup` 属性，用于标识承载启动的命名空间和类。

`ServiceKeyInjection` 类的 Configure 方法使用 IWebHostBuilder 将增强功能添加到应用。用户代码中 `Startup.Configure` 之前的运行时调用托管启动程序集中的 `IHostingStartup.Configure`，允许用户代码覆盖托管启动程序集提供的任何配置。

HostingStartupLibrary/ServiceKeyInjection.cs：

```
[assembly: HostingStartup(typeof(HostingStartupLibrary.ServiceKeyInjection))]

namespace HostingStartupLibrary
{
    public class ServiceKeyInjection : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureAppConfiguration(config =>
            {
                var dict = new Dictionary<string, string>
                {
                    {"DevAccount_FromLibrary", "DEV_1111111-1111"},
                    {"ProdAccount_FromLibrary", "PROD_2222222-2222"}
                };

                config.AddInMemoryCollection(dict);
            });
        }
    }
}
```

应用的索引页读取并呈现类库承载启动程序集设置的两个键的配置值：

HostingStartupApp/Pages/Index.cshtml.cs：

```
public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        ServiceKey_Development_Library = config["DevAccount_FromLibrary"];
        ServiceKey_Production_Library = config["ProdAccount_FromLibrary"];
        ServiceKey_Development_Package = config["DevAccount_FromPackage"];
        ServiceKey_Production_Package = config["ProdAccount_FromPackage"];
    }

    public string ServiceKey_Development_Library { get; private set; }
    public string ServiceKey_Production_Library { get; private set; }
    public string ServiceKey_Development_Package { get; private set; }
    public string ServiceKey_Production_Package { get; private set; }

    public void OnGet()
    {
    }
}
```

示例代码还包括一个 NuGet 包项目，该项目提供单独的承载启动 HostingStartupPackage。该包具有与前述类库相同的特征。包：

- 包含承载启动类 `ServiceKeyInjection`，用于实现 `IHostingStartup`。`ServiceKeyInjection` 将一对服务字符串添加到应用的配置中。
- 包含 `HostingStartup` 属性。

HostingStartupPackage/ServiceKeyInjection.cs：

```
[assembly: HostingStartup(typeof(HostingStartupPackage.ServiceKeyInjection))]

namespace HostingStartupPackage
{
    public class ServiceKeyInjection : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureAppConfiguration(config =>
            {
                var dict = new Dictionary<string, string>
                {
                    {"DevAccount_FromPackage", "DEV_3333333-3333"},
                    {"ProdAccount_FromPackage", "PROD_4444444-4444"}
                };

                config.AddInMemoryCollection(dict);
            });
        }
    }
}
```

应用的索引页读取并呈现包承载启动程序集设置的两个键的配置值：

HostingStartupApp/Pages/Index.cshtml.cs：

```
public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        ServiceKey_Development_Library = config["DevAccount_FromLibrary"];
        ServiceKey_Production_Library = config["ProdAccount_FromLibrary"];
        ServiceKey_Development_Package = config["DevAccount_FromPackage"];
        ServiceKey_Production_Package = config["ProdAccount_FromPackage"];
    }

    public string ServiceKey_Development_Library { get; private set; }
    public string ServiceKey_Production_Library { get; private set; }
    public string ServiceKey_Development_Package { get; private set; }
    public string ServiceKey_Production_Package { get; private set; }

    public void OnGet()
    {
    }
}
```

无入口点的控制台应用

此方法仅适用于 .NET Core 应用，不适用于 .NET Framework。

可在包含 `HostingStartup` 属性的无入口点的控制台应用中提供动态承载启动增强功能，该功能无需编译时引用进行激活。发布控制台应用会生成可从运行时存储中使用的承载启动程序集。

此过程中使用没有入口点的控制台应用，因为：

- 需要依赖项文件来使用承载启动程序集中的承载启动。依赖项文件是一种可运行的应用资产，它通过发布应用而不是库来生成。
- 无法直接将库添加到运行时包存储，该过程需要一个以已共享运行时为目标的可运行项目。

创建动态承载启动过程中：

- 从控制台应用创建承载启动程序集，无需以下入口点：
  - 包含 `IHostingStartup` 实现的类。
  - 包含用于识别 `IHostingStartup` 实现类的 HostingStartup 属性。
- 发布控制台应用，获取承载启动的依赖项。发布控制台应用的结果是从依赖项文件中删除了未使用的依赖项。
- 修改依赖项文件以设置承载启动程序集的运行时位置。
- 承载启动程序集及其依赖项文件位于运行时包存储中。要发现承载启动程序集及其依赖项文件，它们将在一对环境变量中列出。

控制台应用引用 Microsoft.AspNetCore.Hosting.Abstractions 包：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Hosting.Abstractions"
                      Version="2.1.1" />
  </ItemGroup>

</Project>
```

生成 IWebHost 时，HostingStartup 属性将类标识为 `IHostingStartup` 的实现，用于加载和执行。在下面的示例

中，命名空间为 `StartupEnhancement`，类为 `StartupEnhancementHostingStartup`：

```
[assembly: HostingStartup(typeof(StartupEnhancement.StartupEnhancementHostingStartup))]
```

类实现 `IHostingStartup`。类的 Configure 方法使用 IWebHostBuilder 将增强功能添加到应用。用户代码中 `Startup.Configure` 之前的运行时调用托管启动程序集中的 `IHostingStartup.Configure`，允许用户代码覆盖托管启动程序集提供的任何配置。

```
namespace StartupEnhancement
{
    public class StartupEnhancementHostingStartup : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            // Use the IWebHostBuilder to add app enhancements.
        }
    }
}
```

生成 `IHostingStartup` 项目时，依赖项文件 (*.deps.json) 将程序集的 `runtime` 位置设为 bin 文件夹：

```
"targets": {
  ".NETCoreApp,Version=v2.1": {
    "StartupEnhancement/1.0.0": {
      "dependencies": {
        "Microsoft.AspNetCore.Hosting.Abstractions": "2.1.1"
      },
      "runtime": {
        "StartupEnhancement.dll": {}
      }
    }
  }
}
```

仅显示部分文件。示例中程序集的名称是 `StartupEnhancement`。

指定承载启动程序集

对于类库或控制台应用提供的承载启动，请在 `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` 环境变量中指定承载启动程序集的名称。环境变量是以分号分隔的程序集列表。

仅扫描承载启动程序集以查找 `HostingStartup` 属性。对于示例应用 HostingStartupApp，要发现前述的承载启动，请将环境变量设置为以下值：

```
HostingStartupLibrary;HostingStartupPackage;StartupDiagnostics
```

还可使用承载启动程序集主机配置设置来设置此承载启动程序集。

存在多个托管启动程序集时，将按列出程序集的顺序执行 Configure 方法。

激活

承载启动激活的选项包括：

- 运行时存储 – 激活无需用于激活的编译时引用。示例应用将承载启动程序集和依赖项文件放入文件夹“deployment”，以便在多计算机环境中部署承载启动。“deployment”文件夹还包括 PowerShell 脚本，该脚本可在部署系统上创建或修改环境变量以启用承载启动。

- 激活所需的编译时引用
  - NuGet 包
  - 项目 bin 文件夹

运行时存储

承载启动实现位于运行时存储中。增强型应用无需对程序集进行编译时引用。

构建承载启动后，使用清单项目文件和 dotnet store 命令生成运行时存储。

```
dotnet store --manifest {MANIFEST FILE} --runtime {RUNTIME IDENTIFIER} --output {OUTPUT LOCATION} --skip-optimization
```

在示例应用（*RuntimeStore* 项目）中，使用以下命令：

```
dotnet store --manifest store.manifest.csproj --runtime win7-x64 --output ./deployment/store --skip-optimization
```

对于发现运行时存储的运行时，运行时存储的位置将添加到 `DOTNET_SHARED_STORE` 环境变量中。

修改并放置承载启动的依赖项文件

要在未包引用增强功能的情况下激活增强功能，请使用 `additionalDeps` 为运行时指定附加依赖项。使用 `additionalDeps`，你可以：

- 通过提供一组附加的 **.deps.json* 文件来扩展应用的库图，以便在启动时与应用自身的 **.deps.json* 文件合并。
- 使承载启动程序集可被发现并可加载。

生成附加依赖项文件的推荐方法是：

1. 对上一节中引用的运行时存储清单文件执行 `dotnet publish`。
2. 从库中删除清单引用，以及生成的 **deps.json* 文件的 `runtime` 部分。

在示例项目中，`store.manifest/1.0.0` 属性已从 `targets` 和 `libraries` 部分中删除：

```
{
  "runtimeTarget": {
    "name": ".NETCoreApp,Version=v2.1",
    "signature": "4ea77c7b75ad1895ae1ea65e6ba2399010514f99"
  },
  "compilationOptions": {},
  "targets": {
    ".NETCoreApp,Version=v2.1": {
      "store.manifest/1.0.0": {
        "dependencies": {
          "StartupDiagnostics": "1.0.0"
        },
        "runtime": {
          "store.manifest.dll": {}
        }
      },
      "StartupDiagnostics/1.0.0": {
        "runtime": {
          "lib/netcoreapp2.1/StartupDiagnostics.dll": {
            "assemblyVersion": "1.0.0.0",
            "fileVersion": "1.0.0.0"
          }
        }
      }
    }
  },
  "libraries": {
    "store.manifest/1.0.0": {
      "type": "project",
      "serviceable": false,
      "sha512": ""
    },
    "StartupDiagnostics/1.0.0": {
      "type": "package",
      "serviceable": true,
      "sha512": "sha512-
oiQr60vBQW7+nBTmgKLSldj06WNLRTdhOZpAdEbCuapoZ+M2DJH2uQbRLvFT8EGAAv4TAKzNtcztpx5YOgBXQQ==",
      "path": "startupdiagnostics/1.0.0",
      "hashPath": "startupdiagnostics.1.0.0.nupkg.sha512"
    }
  }
}
```

将 **.deps.json* 文件放入以下位置：

```
{ADDITIONAL DEPENDENCIES PATH}/shared/{SHARED FRAMEWORK NAME}/{SHARED FRAMEWORK VERSION}/{ENHANCEMENT
ASSEMBLY NAME}.deps.json
```

- `{ADDITIONAL DEPENDENCIES PATH}` – 添加到 `DOTNET_ADDITIONAL_DEPS` 环境变量中的位置。
- `{SHARED FRAMEWORK NAME}` – 此附加依赖项文件所需的共享框架。
- `{SHARED FRAMEWORK VERSION}` – 最小共享框架版本。
- `{ENHANCEMENT ASSEMBLY NAME}` – 增强功能的程序集名称。

在示例应用（*RuntimeStore* 项目）中，附加依赖项文件放于以下位置：

```
additionalDeps/shared/Microsoft.AspNetCore.App/2.1.0/StartupDiagnostics.deps.json
```

对于发现运行时存储位置的运行时，附加依赖项文件位置将添加到 `DOTNET_ADDITIONAL_DEPS` 环境变量中。

在示例应用（*RuntimeStore* 项目）中，使用 PowerShell 脚本完成构建运行时存储并生成附加依赖项文件。

有关如何设置各种操作系统的环境变量的示例，请参阅使用多个环境。

部署

为了便于在多计算机环境中部署托承载启动，示例应用在已发布的输出中创建一个“deployment”文件夹，其中包含：

- 承载启动运行时存储。
- 承载启动依赖项文件。
- PowerShell 脚本，用于创建或修改 `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES`、`DOTNET_SHARED_STORE` 和 `DOTNET_ADDITIONAL_DEPS` 以支持激活承载启动。从部署系统上的管理 PowerShell 命令提示符运行脚本。

NuGet 程序包

可在 NuGet 包中提供承载启动增强。该包含有 `HostingStartup` 属性。包提供的承载启动类型使用以下任一方法提供给应用：

- 增强型应用的项目文件在应用的项目文件（编译时引用）中为承载启动提供了包引用。有了编译时引用，承载启动程序集及其所有依赖项都会合并到应用的依赖项文件 (*.deps.json) 中。此方法适用于已发布到 nuget.org 的承载启动程序集包。
- 承载启动的依赖项文件可用于增强型应用，如运行时存储部分所述（无编译时引用）。

有关 NuGet 包和运行时存储的详细信息，请参阅以下主题：

- 如何使用跨平台工具创建 NuGet 包
- 发布包
- 运行时包存储

项目 bin 文件夹

承载启动增强功能可通过增强型应用中的 bin -deployed 程序集提供。程序集提供的承载启动类型使用以下任一方法提供给应用：

- 增强型应用的项目文件对承载启动进行了程序集引用（编译时引用）。有了编译时引用，承载启动程序集及其所有依赖项都会合并到应用的依赖项文件 (*.deps.json) 中。如果部署方案要求将已编译承载启动库的程序集（DLL 文件）移动到使用项目中或使用项目可访问的位置，并且对承载启动程序集进行编译时引用，此方法适用。
- 承载启动的依赖项文件可用于增强型应用，如运行时存储部分所述（无编译时引用）。

示例代码

示例代码（如何下载）演示了承载启动实现方案：

- 两个承载启动程序集（类库）分别设置一对内存中配置键值对：
  - NuGet 包 (HostingStartupPackage)
  - 类库 (HostingStartupLibrary)
- 从运行时存储部署的程序集 (StartupDiagnostics) 激活承载启动。该程序集在启动时将两个中间件添加到应用，用于提供以下内容的诊断信息：
  - 已注册服务
  - 地址（方案、主机、基路径、路径、查询字符串）
  - 连接（远程 IP、远程端口、本地 IP、本地端口、客户端证书）
  - 请求标头
  - 环境变量

运行示例：

从 NuGet 包激活

1. 使用 dotnet pack 命令编译 HostingStartupPackage 包。
2. 将包的程序集名称 HostingStartupPackage 添加到 `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` 环境变量中。
3. 编译并运行应用。增强型应用中存在包引用(编译时引用)。应用项目文件中的 `<PropertyGroup>` 指定包项目的输出 (../HostingStartupPackage/bin/Debug) 作为包源。这允许应用使用该包而无需将包上传到 nuget.org。有关详细信息，请参阅 HostingStartupApp 项目文件中的说明。

```
<PropertyGroup>

<RestoreSources>$(RestoreSources);https://api.nuget.org/v3/index.json;../HostingStartupPackage/bin/De
bug</RestoreSources>
</PropertyGroup>
```

4. 观察到索引页呈现的服务配置键值与包的 `ServiceKeyInjection.Configure` 方法设置的值匹配。

如果更改并重新编译 HostingStartupPackage 项目，请清除本地 NuGet 包缓存，确保 HostingStartupApp 从本地缓存中收到更新后的包而不是旧包。要清除本地 NuGet 缓存，请执行以下 dotnet nuget locals 命令：

```
dotnet nuget locals all --clear
```

从类库激活

1. 使用 dotnet build 命令编译 HostingStartupLibrary 类库。
2. 将类库的程序集名称 HostingStartupLibrary 添加到 `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` 环境变量中。
3. bin - 通过将类库编译输出中的 HostingStartupLibrary.dll 文件复制到应用的 bin/Debug 文件夹，将类库程序集部署到应用。
4. 编译并运行应用。应用项目文件中的 `<ItemGroup>` 引用类库的程序集 (.\bin\Debug\netcoreapp2.1\HostingStartupLibrary.dll)(编译时引用)。有关详细信息，请参阅 HostingStartupApp 项目文件中的说明。

```
<ItemGroup>
  <Reference Include=".\bin\Debug\netcoreapp2.1\HostingStartupLibrary.dll">
    <HintPath>.\bin\Debug\netcoreapp2.1\HostingStartupLibrary.dll</HintPath>
    <SpecificVersion>False</SpecificVersion>
  </Reference>
</ItemGroup>
```

5. 观察到索引页呈现的服务配置键值与类库的 `ServiceKeyInjection.Configure` 方法设置的值匹配。

从运行时存储部署的程序集激活

1. StartupDiagnostics 项目使用 PowerShell 修改其 StartupDiagnostics.deps.json 文件。默认情况下，Windows 7 SP1 和 Windows Server 2008 R2 SP1 及以后版本的 Windows 上安装有 PowerShell。若要在其他平台上获取 PowerShell，请参阅安装 Windows PowerShell。
2. 构建 StartupDiagnostics 项目。构建项目后，会自动生成项目文件中的构建目标：
   - 触发 PowerShell 脚本以修改 StartupDiagnostics.deps.json 文件。
   - 将 StartupDiagnostics.deps.json 文件移动到用户配置文件的 additionalDeps 文件夹。
3. 在承载启动目录的命令提示符处执行 `dotnet store` 命令，将程序集及其依赖项存储在用户配置文件的运行时存储中：

```
dotnet store --manifest StartupDiagnostics.csproj --runtime <RID>
```

对于 Windows，该命令使用 `win7-x64` 运行时标识符 (RID)。为其他运行时提供承载启动时，请替换为正确的 RID。

4. 设置环境变量：
   - 将程序集名称 StartupDiagnostics 的添加到 `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` 环境变量中。
   - 在 Windows 上，将 `DOTNET_ADDITIONAL_DEPS` 环境变量设置为 `%UserProfile%\.dotnet\x64\additionalDeps\StartupDiagnostics\`。在 macOS/Linux 上，将 `DOTNET_ADDITIONAL_DEPS` 环境变量设置为 `/Users/<USER>/.dotnet/x64/additionalDeps/StartupDiagnostics/`，其中 `<USER>` 是包含承载启动的用户配置文件。
5. 运行示例应用。
6. 请求 `/services` 终结点以查看应用的注册服务。请求 `/diag` 终结点以查看诊断信息。

ASP.NET Core 2.1 的 Microsoft.AspNetCore.App 元包

2018/10/12 • • Edit Online

此功能需要面向 .NET Core 2.1 及更高版本的 ASP.NET Core 2.1。

ASP.NET Core 的 Microsoft.AspNetCore.App 元包：

- 不包括第三方依赖项，Json.NET、Remotion.Linq 和 IX-Async 除外。为确保正常使用主要框架功能，这些第三方依赖项均为必要条件。
- 包括 ASP.NET Core 团队支持的所有软件包，包含第三方依赖项的软件包（不包括上文所述）除外。
- 包括 Entity Framework Core 团队支持的所有软件包，包含第三方依赖项的软件包（不包括上文所述）除外。

`Microsoft.AspNetCore.App` 包中包含了 ASP.NET Core 2.1 及更高版本和 Entity Framework Core 2.1 及更高版本的所有功能。面向 ASP.NET Core 2.1 及更高版本的默认项目模板使用此包。建议面向 ASP.NET Core 2.1 及更高版本和 Entity Framework Core 2.1 及更高版本的应用程序使用 `Microsoft.AspNetCore.App` 包。

`Microsoft.AspNetCore.App` 元包的版本号表示 ASP.NET Core 版本和 Entity Framework Core 版本。

使用 `Microsoft.AspNetCore.App` 元包可提供用于保护应用的版本限制：

- 如果包含的包对 `Microsoft.AspNetCore.App` 中的包具有可传递的（非直接）依赖项，且这些版本号不同，则 NuGet 将生成错误。
- 添加到应用的其他包无法更改 `Microsoft.AspNetCore.App` 中包含的包版本。
- 版本一致性能确保可靠的体验。`Microsoft.AspNetCore.App` 旨在防止在同一应用中结合使用未经测试的相关位版本组合。

使用 `Microsoft.AspNetCore.App` 元包的应用程序会自动使用 ASP.NET Core 共享框架。使用 `Microsoft.AspNetCore.App` 元包时，应用程序不部署所引用的 ASP.NET Core NuGet 包中的任何资产 — .NET Core 共享框架包含这些资产。共享框架中的资产已经过预编译，以便缩短应用程序启动时间。有关详细信息，请参阅 .NET Core 分发打包中的“共享框架”。

以下项目文件为 ASP.NET Core 引用 `Microsoft.AspNetCore.App` 元包，该文件是一种典型的 ASP.NET Core 2.1 模板：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

前面的标记表示典型的 ASP.NET Core 2.1 及更高版本模板。它不会为

`Microsoft.AspNetCore.App` 包引用指定版本号。如果未指定版本，SDK 会指定隐式版本，即 `Microsoft.NET.Sdk.Web`。建议使用 SDK 指定的隐式版本，而不是在包引用上显式设置版本号。如果对这种方法有任何疑问，可以在 Microsoft.AspNetCore.App 隐式版本讨论上发表 GitHub 评论。

对于便携式应用，隐式版本设置为 `major.minor.0`。共享框架前滚机制将在安装的共享框架的最新兼容版本上运行应用。为确保在开发、测试和生产中使用相同的版本，请确保在所有环境中都安装相同版本的共享框架。对于独立应用，将隐式版本号设置为在已安装的 SDK 中捆绑的共享框架的 `major.minor.patch`。

在 `Microsoft.AspNetCore.App` 引用上指定版本号，不能保证将选择该共享框架版本。例如，假设指定的版本是“2.1.1”，但安装的是“2.1.3”。这种情况下，应用将使用"2.1.3"。不过不建议这样做，你可以禁用前滚（修补程序和/或次要版本）。有关 dotnet 主机前滚以及如何配置其行为的详细信息，请参阅 dotnet 主机前滚。

`<Project Sdk` 必须设置为 `Microsoft.NET.Sdk.Web` 以使用隐式版本 `Microsoft.AspNetCore.App`。使用 `<Project Sdk="Microsoft.NET.Sdk">`（不带尾随 `.Web`）时：

- 生成以下警告：

  警告 NU1604：项目依赖项 Microsoft.AspNetCore.App 不包括包含下限。请在依赖项版本中包括下限，以确保一致的还原结果。

- 这是 .NET Core 2.1 SDK 的一个已知问题，将在 .NET Core 2.2 SDK 中修复。

更新 ASP.NET Core

`Microsoft.AspNetCore.App` 元包不是从 NuGet 更新的传统包。类似于 `Microsoft.NETCore.App`，`Microsoft.AspNetCore.App` 表示共享运行时，它具有在 NuGet 之外处理的特殊版本控制语义。有关详细信息，请参阅包、元包和框架。

更新 ASP.NET Core：

- 在开发计算机和构建服务器上：下载并安装 .NET Core SDK。
- 在部署服务器上：下载并安装 .NET Core 运行时。

在应用程序重启时，应用程序将前滚到最新安装的版本。无需更新项目文件中的 `Microsoft.AspNetCore.App` 版本号。有关详细信息，请参阅依赖于框架的应用会前滚。

如果应用程序之前使用 `Microsoft.AspNetCore.All`，请参阅从 Microsoft.AspNetCore.All 迁移到 Microsoft.AspNetCore.App。

ASP.NET Core 2.0 的 Microsoft.AspNetCore.All 元包

2018/11/5 • • Edit Online

> **NOTE**
> 对于面向 ASP.NET Core 2.1 及更高版本的应用程序，建议使用 Microsoft.AspNetCore.App 元包而不是此包。请参阅本文中的从 Microsoft.AspNetCore.All 迁移到 Microsoft.AspNetCore.App。

此功能需要面向 .NET Core 2.x 的 ASP.NET Core 2.x。

ASP.NET Core 的 Microsoft.AspNetCore.All 元包包括：

- ASP.NET Core 团队支持的所有包。
- Entity Framework Core 支持的所有包。
- ASP.NET Core 和 Entity Framework Core 使用的内部和第三方依赖关系。

`Microsoft.AspNetCore.All` 包中包含了 ASP.NET Core 2.x 和 Entity Framework Core 2.x 的所有功能。定目标到 ASP.NET Core 2.0 的默认项目模板使用此包。

`Microsoft.AspNetCore.All` 元包的版本号表示 ASP.NET Core 版本和 Entity Framework Core 版本。

使用 `Microsoft.AspNetCore.All` 元包的应用程序会自动使用 .NET Core 运行时存储。此运行时存储包含运行 ASP.NET Core 2.x 应用程序所需的所有运行时资产。使用 `Microsoft.AspNetCore.All` 元包时，应用程序不会部署引用的 ASP.NET Core NuGet 包中的任何资产—.NET Core 运行时存储包含这些资产。运行时存储中的资产已经过预编译，以便缩短应用程序启动时间。

可使用包修整过程来删除不使用的包。已发布的应用程序输出中排除了修整的包。

以下 .csproj 文件引用了 ASP.NET Core 的 `Microsoft.AspNetCore.All` 元包：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
  </ItemGroup>

</Project>
```

隐式版本控制

在 ASP.NET Core 2.1 或更高版本中，可以指定 `Microsoft.AspNetCore.All` 包引用但不指定版本。如果未指定版本，SDK 会指定隐式版本（`Microsoft.NET.Sdk.Web`）。建议使用 SDK 指定的隐式版本，而不是在包引用上显式设置版本号。如果对这种方法有任何疑问，可以在 Microsoft.AspNetCore.App 隐式版本讨论上发表 GitHub 评论。

对于便携式应用，隐式版本设置为 `major.minor.0`。共享框架前滚机制在安装的共享框架的最新兼容版本上运行应用。为确保在开发、测试和生产中使用相同的版本，请确保在所有环境中都安装相同版本的共享

框架。对于独立应用，将隐式版本号设置为在已安装的 SDK 中捆绑的共享框架的 `major.minor.patch`。

在 `Microsoft.AspNetCore.All` 包引用上指定版本号，不能保证会选择该共享框架版本。例如，假设指定的版本是“2.1.1”，但安装的是“2.1.3”。这种情况下，应用将使用"2.1.3"。不过不建议这样做，你可以禁用前滚（修补程序和/或次要版本）。有关 dotnet 主机前滚以及如何配置其行为的详细信息，请参阅 dotnet 主机前滚。

必须在项目文件中将项目的 SDK 设置为 `Microsoft.NET.Sdk.Web`，这样才能使用隐式版本的 `Microsoft.AspNetCore.All`。指定 `Microsoft.NET.Sdk` SDK（项目文件顶部的 `<Project Sdk="Microsoft.NET.Sdk">`）时，将生成以下警告：

警告 NU1604：项目依赖项 Microsoft.AspNetCore.All 不包括包含下限。请在依赖项版本中包括下限，以确保一致的还原结果。

这是 .NET Core 2.1 SDK 的一个已知问题，将在 .NET Core 2.2 SDK 中修复。

从 Microsoft.AspNetCore.All 迁移到 Microsoft.AspNetCore.App

`Microsoft.AspNetCore.All` 包中包含以下包，而 `Microsoft.AspNetCore.App` 包中不包含以下包。

- `Microsoft.AspNetCore.ApplicationInsights.HostingStartup`
- `Microsoft.AspNetCore.AzureAppServices.HostingStartup`
- `Microsoft.AspNetCore.AzureAppServicesIntegration`
- `Microsoft.AspNetCore.DataProtection.AzureKeyVault`
- `Microsoft.AspNetCore.DataProtection.AzureStorage`
- `Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv`
- `Microsoft.AspNetCore.SignalR.Redis`
- `Microsoft.Data.Sqlite`
- `Microsoft.Data.Sqlite.Core`
- `Microsoft.EntityFrameworkCore.Sqlite`
- `Microsoft.EntityFrameworkCore.Sqlite.Core`
- `Microsoft.Extensions.Caching.Redis`
- `Microsoft.Extensions.Configuration.AzureKeyVault`
- `Microsoft.Extensions.Logging.AzureAppServices`
- `Microsoft.VisualStudio.Web.BrowserLink`

要从 `Microsoft.AspNetCore.All` 移到 `Microsoft.AspNetCore.App`，如果应用使用上述包或由这些包引入的包中的任何 API，请在项目中添加对这些包的引用。

不会隐式包含非 `Microsoft.AspNetCore.App` 依赖项的上述包的任何依赖项。例如：

- `StackExchange.Redis` 作为 `Microsoft.Extensions.Caching.Redis` 的依赖项
- `Microsoft.ApplicationInsights` 作为 `Microsoft.AspNetCore.ApplicationInsights.HostingStartup` 的依赖项

更新 ASP.NET Core 2.1

建议迁移到 2.1 及更高版本的 `Microsoft.AspNetCore.App` 元数据包。要继续使用 `Microsoft.AspNetCore.All` 元数据包并确保部署最新的补丁版本，请执行以下操作：

- 在开发计算机和生成服务器上：安装最新的 .NET Core SDK。
- 在部署服务器上：安装最新的 .NET Core 运行时。在应用程序重启时，应用将前滚到最新安装的版本。

在 ASP.NET Core 中使用 LoggerMessage 的高性能日志记录

2018/10/30 • • Edit Online

作者：Luke Latham

LoggerMessage 功能创建可缓存的委托，该功能比记录器扩展方法（例如 `LogInformation`、`LogDebug` 和 `LogError`）需要的对象分配和计算开销少。对于高性能日志记录方案，请使用 `LoggerMessage` 模式。

与记录器扩展方法相比，`LoggerMessage` 具有以下性能优势：

- 记录器扩展方法需要将值类型（例如 `int`）"装箱"（转换）到 `object` 中。`LoggerMessage` 模式使用带强类型参数的静态 `Action` 字段和扩展方法来避免装箱。
- 记录器扩展方法每次写入日志消息时必须分析消息模板（命名的格式字符串）。如果已定义消息，那么 `LoggerMessage` 只需分析一次模板即可。

查看或下载示例代码（如何下载）

此示例应用通过基本引号跟踪系统演示 `LoggerMessage` 功能。此应用使用内存中数据库添加和删除引号。发生这些操作时，通过 `LoggerMessage` 模式生成日志消息。

LoggerMessage.Define

Define（LogLevel、EventId、字符串）创建用于记录消息的 `Action` 委托。`Define` 重载允许向命名的格式字符串（模板）传递最多六个类型参数。

提供给 `Define` 方法的字符串是一个模板，而不是内插字符串。占位符按照指定类型的顺序填充。模板中的占位符名称在各个模板中应当具备描述性和一致性。它们在结构化的日志数据中充当属性名称。对于占位符名称，建议使用帕斯卡拼写法。例如：`{Count}`、`{FirstName}`。

每条日志消息都是一个 `Action`，保存在由 `LoggerMessage.Define` 创建的静态字段中。例如，示例应用创建一个字段来为索引页 (Internal/LoggerExtensions.cs) 描述 GET 请求的日志消息：

```
private static readonly Action<ILogger, Exception> _indexPageRequested;
```

对于 `Action`，指定：

- 日志级别。
- 具有静态扩展方法名称的唯一事件标识符 (EventId)。
- 消息模板（命名的格式字符串）。

对示例应用的索引页的请求设置：

- 将日志级别设置为 `Information`。
- 将事件 ID 设置为具有 `IndexPageRequested` 方法名称的 `1`。
- 将消息模板（命名的格式字符串）设置为字符串。

```
_indexPageRequested = LoggerMessage.Define(
    LogLevel.Information,
    new EventId(1, nameof(IndexPageRequested)),
    "GET request for Index page");
```

结构化日志记录存储可以使用事件名称(当它获得事件 ID 时)来丰富日志记录。例如，Serilog 使用该事件名称。

通过强类型扩展方法调用 `Action`。`IndexPageRequested` 方法在示例应用中记录索引页 GET 请求的消息：

```
public static void IndexPageRequested(this ILogger logger)
{
    _indexPageRequested(logger, null);
}
```

在 Pages/Index.cshtml.cs 的 `OnGetAsync` 方法中，在记录器上调用 `IndexPageRequested`：

```
public async Task OnGetAsync()
{
    _logger.IndexPageRequested();

    Quotes = await _db.Quotes.AsNoTracking().ToListAsync();
}
```

检查应用的控制台输出：

```
info: LoggerMessageSample.Pages.IndexModel[1]
      => RequestId:0HL90M6E7PHK4:00000001 RequestPath:/ => /Index
      GET request for Index page
```

要将参数传递给日志消息，创建静态字段时最多定义六种类型。通过为 `Action` 字段定义 `string` 类型来添加引号时，示例应用会记录一个字符串：

```
private static readonly Action<ILogger, string, Exception> _quoteAdded;
```

委托的日志消息模板从提供的类型接收其占位符值。示例应用定义一个委托，用于在 quote 参数是 `string` 的位置添加引号：

```
_quoteAdded = LoggerMessage.Define<string>(
    LogLevel.Information,
    new EventId(2, nameof(QuoteAdded)),
    "Quote added (Quote = '{Quote}')");
```

用于添加引号的静态扩展方法 `QuoteAdded` 接收 quote 参数值并将其传递给 `Action` 委托：

```
public static void QuoteAdded(this ILogger logger, string quote)
{
    _quoteAdded(logger, quote, null);
}
```

在索引页的页面模型 (Pages/Index.cshtml.cs) 中，调用 `QuoteAdded` 来记录消息：

```
public async Task<IActionResult> OnPostAddQuoteAsync()
{
    _db.Quotes.Add(Quote);
    await _db.SaveChangesAsync();

    _logger.QuoteAdded(Quote.Text);

    return RedirectToPage();
}
```

检查应用的控制台输出：

```
info: LoggerMessageSample.Pages.IndexModel[2]
      => RequestId:0HL90M6E7PHK5:0000000A RequestPath:/ => /Index
      Quote added (Quote = 'You can avoid reality, but you cannot avoid the consequences of avoiding reality.
- Ayn Rand')
```

本示例应用实现用于删除引号的 `try` – `catch` 模式。为成功的删除操作记录提示性信息。引发异常时，为删除操作记录错误消息。针对未成功的删除操作，日志消息包括异常堆栈跟踪 (Internal/LoggerExtensions.cs)：

```
private static readonly Action<ILogger, string, int, Exception> _quoteDeleted;
private static readonly Action<ILogger, int, Exception> _quoteDeleteFailed;
```

```
_quoteDeleted = LoggerMessage.Define<string, int>(
    LogLevel.Information,
    new EventId(4, nameof(QuoteDeleted)),
    "Quote deleted (Quote = '{Quote}' Id = {Id})");

_quoteDeleteFailed = LoggerMessage.Define<int>(
    LogLevel.Error,
    new EventId(5, nameof(QuoteDeleteFailed)),
    "Quote delete failed (Id = {Id})");
```

请注意异常如何传递到 `QuoteDeleteFailed` 中的委托：

```
public static void QuoteDeleted(this ILogger logger, string quote, int id)
{
    _quoteDeleted(logger, quote, id, null);
}

public static void QuoteDeleteFailed(this ILogger logger, int id, Exception ex)
{
    _quoteDeleteFailed(logger, id, ex);
}
```

在索引页的页面模型中，成功删除引号时会在记录器上调用 `QuoteDeleted` 方法。如果找不到要删除的引号，则会引发 `ArgumentNullException`。通过 `try` – `catch` 语句捕获异常，并在 `catch` 块 (Pages/Index.cshtml.cs) 中调用记录器上的 `QuoteDeleteFailed` 方法来记录异常：

```
public async Task<IActionResult> OnPostDeleteQuoteAsync(int id)
{
    var quote = await _db.Quotes.FindAsync(id);

    // DO NOT use this approach in production code!
    // You should check quote to see if it's null before removing
    // it and saving changes to the database. A try-catch is used
    // here for demonstration purposes of LoggerMessage features.
    try
    {
        _db.Quotes.Remove(quote);
        await _db.SaveChangesAsync();

        _logger.QuoteDeleted(quote.Text, id);
    }
    catch (ArgumentNullException ex)
    {
        _logger.QuoteDeleteFailed(id, ex);
    }

    return RedirectToPage();
}
```

成功删除引号时，检查应用的控制台输出：

```
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:00000016 RequestPath:/ => /Index
      Quote deleted (Quote = 'You can avoid reality, but you cannot avoid the consequences of avoiding
reality. - Ayn Rand' Id = 1)
```

引号删除失败时，检查应用的控制台输出。请注意，异常包括在日志消息中：

```
fail: LoggerMessageSample.Pages.IndexModel[5]
      => RequestId:0HL90M6E7PHK5:00000010 RequestPath:/ => /Index
      Quote delete failed (Id = 999)
System.ArgumentNullException: Value cannot be null.
Parameter name: entity
   at Microsoft.EntityFrameworkCore.Utilities.Check.NotNull[T](T value, String parameterName)
   at Microsoft.EntityFrameworkCore.DbContext.Remove[TEntity](TEntity entity)
   at Microsoft.EntityFrameworkCore.Internal.InternalDbSet`1.Remove(TEntity entity)
   at LoggerMessageSample.Pages.IndexModel.<OnPostDeleteQuoteAsync>d__14.MoveNext() in
      <PATH>\sample\Pages\Index.cshtml.cs:line 87
```

LoggerMessage.DefineScope

DefineScope(字符串)创建一个用于定义日志作用域的 `Func` 委托。`DefineScope` 重载允许向命名的格式字符串(模板)传递最多三个类型参数。

`Define` 方法也一样，提供给 `DefineScope` 方法的字符串是一个模板，而不是内插字符串。占位符按照指定类型的顺序填充。模板中的占位符名称在各个模板中应当具备描述性和一致性。它们在结构化的日志数据中充当属性名称。对于占位符名称，建议使用帕斯卡拼写法。例如：`{Count}`、`{FirstName}`。

使用 DefineScope(字符串)方法定义一个日志作用域，以应用到一系列日志消息中。

示例应用含有一个"全部清除"按钮，用于删除数据库中的所有引号。通过一次删除一个引号来将其删除。每当删除一个引号时，都会在记录器上调用 `QuoteDeleted` 方法。在这些日志消息中会添加一个日志作用域。

在 appsettings.json 的控制台记录器部分启用 `IncludeScopes`：

```
{
  "Logging": {
    "Console": {
      "IncludeScopes": true
    },
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

要创建日志作用域，请添加一个字段来保存该作用域的 `Func` 委托。示例应用创建一个名为 `_allQuotesDeletedScope` (Internal/LoggerExtensions.cs) 的字段：

```
private static Func<ILogger, int, IDisposable> _allQuotesDeletedScope;
```

使用 `DefineScope` 来创建委托。调用委托时最多可以指定三种类型作为模板参数使用。示例应用使用包含删除的引号数量的消息模板（`int` 类型）：

```
_allQuotesDeletedScope = LoggerMessage.DefineScope<int>("All quotes deleted (Count = {Count})");
```

为日志消息提供一种静态扩展方法。包含已命名属性的任何类型参数（这些参数出现在消息模板中）。示例应用采用引号的 `count`，以删除并返回 `_allQuotesDeletedScope`：

```
public static IDisposable AllQuotesDeletedScope(this ILogger logger, int count)
{
    return _allQuotesDeletedScope(logger, count);
}
```

该作用域将日志记录扩展调用包装在 `using` 块中：

```
public async Task<IActionResult> OnPostDeleteAllQuotesAsync()
{
    var quoteCount = await _db.Quotes.CountAsync();

    using (_logger.AllQuotesDeletedScope(quoteCount))
    {
        foreach (Quote quote in _db.Quotes)
        {
            _db.Quotes.Remove(quote);

            _logger.QuoteDeleted(quote.Text, quote.Id);
        }
        await _db.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

检查应用控制台输出中的日志消息。以下结果显示删除的三个引号，以及包含的日志作用域消息：

```
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index => All quotes deleted (Count = 3)
      Quote deleted (Quote = 'Quote 1' Id = 2)
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index => All quotes deleted (Count = 3)
      Quote deleted (Quote = 'Quote 2' Id = 3)
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index => All quotes deleted (Count = 3)
      Quote deleted (Quote = 'Quote 3' Id = 4)
```

其他资源

- 日志记录

使用文件观察程序开发 ASP.NET Core 应用

2018/7/16 • • Edit Online

作者：Rick Anderson 和 Victor Hurdugaci

`dotnet watch` 是一种在源文件更改时运行 .NET Core CLI 命令的工具。例如，文件更改可能触发编译、测试执行或部署。

本教程使用现有 Web API，它具有两个终结点：分别返回总和和产品。该产品方法有一个 bug，本教程已修复。

下载示例应用。它包含两个项目：WebApp（一种 ASP.NET Core Web API）和 WebAppTests（Web API 的单元测试）。

在命令行界面中，导航到 WebApp 文件夹。运行下面的命令：

```
dotnet run
```

控制台输出会显示如下类似的消息（表示应用正在运行且正在等待请求）：

```
$ dotnet run
Hosting environment: Development
Content root path: C:/Docs/aspnetcore/tutorials/dotnet-watch/sample/WebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

在 Web 浏览器中，导航到 `http://localhost:<port number>/api/math/sum?a=4&b=5`。应该会显示结果 `9`。

导航到产品 API（`http://localhost:<port number>/api/math/product?a=4&b=5`）。它会返回 `9`，而不是所预期的 `20`。本教程的后面部分将解决该问题。

将 `dotnet watch` 添加到项目

`dotnet watch` 文件观察程序工具随 2.1.300 版本的 .NET Core SDK 一同提供。使用早期版本的 .NET Core SDK 时，需要执行以下步骤。

1. 将 `Microsoft.DotNet.Watcher.Tools` 包引用添加到 .csproj 文件：

   ```
   <ItemGroup>
       <DotNetCliToolReference Include="Microsoft.DotNet.Watcher.Tools" Version="2.0.0" />
   </ItemGroup>
   ```

2. 运行以下命令，安装 `Microsoft.DotNet.Watcher.Tools` 包：

   ```
   dotnet restore
   ```

使用 `dotnet watch` 运行 .NET Core CLI 命令

`dotnet watch` 可用于运行任何 .NET Core CLI 命令 例如:

命令	带 WATCH 的命令
dotnet 运行	dotnet watch run
dotnet run -f netcoreapp2.0	dotnet watch run -f netcoreapp2.0
dotnet run -f netcoreapp2.0 -- --arg1	dotnet watch run -f netcoreapp2.0 -- --arg1
dotnet test	dotnet watch test

运行"WebApp"文件夹中的 `dotnet watch run`。控制台输出指示 `watch` 已启动。

使用 `dotnet watch` 执行更改

确保 `dotnet watch` 正在运行。

修复 MathController 的 `Product` 方法中的 bug，使其返回产品而非总和。

```
public static int Product(int a, int b)
{
  return a * b;
}
```

保存该文件。控制台输出指示 `dotnet watch` 已检测到文件更改并已重启应用。

验证 `http://localhost:<port number>/api/math/product?a=4&b=5` 是否返回正确结果。

使用 `dotnet watch` 运行测试

1. 将 MathController.cs 的 `Product` 方法改回返回总和。保存该文件。
2. 在命令外壳中，导航到"WebAppTests"文件夹。
3. 运行 dotnet restore。
4. 运行 `dotnet watch test`。其输出指示测试失败且观察程序正在等待文件更改：

   ```
   Total tests: 2. Passed: 1. Failed: 1. Skipped: 0.
   Test Run Failed.
   ```

5. 修复 `Product` 方法代码，使其返回产品。保存该文件。

`dotnet watch` 检测到文件更改并重新运行测试。控制台输出指示测试通过。

自定义要监视的文件列表

默认情况下，`dotnet-watch` 跟踪与以下 glob 模式匹配的所有文件：

- `**/*.cs`
- `*.csproj`
- `**/*.resx`

通过编辑 .csproj 文件，可向监视列表添加更多项。可以单独指定项或使用 glob 模式指定。

```
<ItemGroup>
    <!-- extends watching group to include *.js files -->
    <Watch Include="**\*.js" Exclude="node_modules\**\*;**\*.js.map;obj\**\*;bin\**\*" />
</ItemGroup>
```

选择退出要监视的文件

`dotnet-watch` 可配置为忽略其默认设置。要忽略特定文件，请在 .csproj 文件中向某项的定义中添加 `Watch="false"` 特性：

```
<ItemGroup>
    <!-- exclude Generated.cs from dotnet-watch -->
    <Compile Include="Generated.cs" Watch="false" />

    <!-- exclude Strings.resx from dotnet-watch -->
    <EmbeddedResource Include="Strings.resx" Watch="false" />

    <!-- exclude changes in this referenced project -->
    <ProjectReference Include="..\ClassLibrary1\ClassLibrary1.csproj" Watch="false" />
</ItemGroup>
```

自定义监视项目

`dotnet-watch` 不仅限于 C# 项目。可以创建自定义监视项目来处理不同的方案。假设项目布局如下：

- **test/**
  - *UnitTests/UnitTests.csproj*
  - *IntegrationTests/IntegrationTests.csproj*

如果目标是监视这两个项目，请创建配置为监视这两个项目的自定义项目文件：

```
<Project>
    <ItemGroup>
        <TestProjects Include="**\*.csproj" />
        <Watch Include="**\*.cs" />
    </ItemGroup>

    <Target Name="Test">
        <MSBuild Targets="VSTest" Projects="@(TestProjects)" />
    </Target>

    <Import Project="$(MSBuildExtensionsPath)\Microsoft.Common.targets" />
</Project>
```

要对这两个项目启动文件监视，请更改为 test 文件夹。请执行以下命令：

```
dotnet watch msbuild /t:Test
```

当任一测试项目中的任何文件发生更改时，将会执行 VSTest。

GitHub 中的 `dotnet-watch`

`dotnet-watch` 是 GitHub DotNetTools 存储库的一部分。

从 ASP.NET Core 迁移 2.2 到 3.0 预览

2019/1/8 • • Edit Online

作者：Scott Addie

本文介绍如何更新现有的 ASP.NET Core 2.2 项目到 ASP.NET Core 3.0。

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 已安装“ASP.NET 和 Web 开发”工作负载的 [Visual Studio 2017 版本 15.9 或更高版本](* Visual Studio 2017 version 15.9 or later with the **ASP.NET and web development** workload)
- .NET core SDK 3.0

更新项目文件

- 设置目标框架名字对象 (TFM)到 `netcoreapp3.0`：

  ```
  <TargetFramework>netcoreapp3.0</TargetFramework>
  ```

- 删除任何 `<PackageReference>` 到Microsoft.AspNetCore.All或Microsoft.AspNetCore.App元包。

  预览 1 中没有的已知的问题：不使用启动项目 `<Project Sdk="Microsoft.NET.Sdk.Web">` 收到由于缺少编译器或运行时错误 `Microsoft.AspNetCore.*` 程序集。这通常是用于测试项目和类库用例。解决方法是以下内容添加至 *.csproj*文件。

  ```
  <ItemGroup>
     <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
  ```

  有关详细信息，请参阅NuGet/Home 问题 #7342。

- 更新 `Version` 上的剩余 `<PackageReference>` 元素 `Microsoft.AspNetCore.*` 程序包添加到当前的预览版中 (例如，3.0.0-preview-18579-0053)。

  如果没有版本 3.0 的包，包可能会弃用在 3.0 中。其中许多都属于 `Microsoft.AspNetCore.App` 和不应单独不再引用。初步不再生成在 3.0 中的包的列表，请参阅aspnet/AspNetCore #3756。

- 某些程序集已从 `Microsoft.AspNetCore.App` 2.x 和 3.0 之间。可能需要添加 `<PackageReference>` 如果你使用的 Api 中列出的包中的项aspnet/AspNetCore #3755

  例如，`Microsoft.EntityFrameworkCore` 并 `System.Data.SqlClient` 是否不再属于 `Microsoft.AspNetCore.App`。在进行发布的程序集列表 `Microsoft.AspNetCore.App` 尚未尚未完成，并且在 3.0 RTM 之前会更改。

从 ASP.NET Core 2.1 迁移到 2.2

2019/1/12 • • Edit Online

作者：Scott Addie

本文介绍如何将现有的 ASP.NET Core 2.1 项目更新到 ASP.NET Core 2.2。

系统必备

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

更新目标框架名字对象 (TFM)

面向 .NET Core 的项目应使用版本大于或等于 .NET Core 2.2 的 TFM。使用 `netcoreapp2.2` 更新 `<TargetFramework>` 节点：

```
<TargetFramework>netcoreapp2.2</TargetFramework>
```

面向.NET Framework 的项目可以继续使用大于或等于.NET Framework 4.6.1 版本的 TFM:

```
<TargetFramework>net461</TargetFramework>
```

采用的 IIS 进程内承载模型

若要采用IIS 进程内承载模型，添加 `<AspNetCoreHostingModel>` 属性值为 `InProcess` 到 `<PropertyGroup>` 项目文件中：

```
<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
```

对于面向.NET Framework 的 ASP.NET Core 应用不支持进程内承载模型。

有关详细信息，请参阅 ASP.NET Core 模块。

更新包引用

如果以面向 .NET Core，请删除元包引用的 `Version` 属性。包含 `Version` 属性将导致以下警告：

```
A PackageReference to 'Microsoft.AspNetCore.App' specified a Version of `2.2.0`. Specifying the version of this
package is not recommended. For more information, see https://aka.ms/sdkimplicitrefs
```

有关详细信息，请参阅 ASP.NET Core 2.1 及更高版本的 Microsoft.AspNetCore.App 元包。

元包引用应与下方的 `<PackageReference />` 节点相似：

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

如果面向 .NET Framework，请将每个包引用的 `Version` 属性更新到 2.2.0 或更高版本。下面是面向 .NET Framework 的典型 ASP.NET Core 2.2 项目中的包引用：

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.CookiePolicy" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.HttpsPolicy" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.2.0" />
</ItemGroup>
```

如果在引用Microsoft.AspNetCore.Razor.Design包，更新其 `Version` 属性到 2.2.0 或更高版本。如果不这样做会导致以下错误：

```
Detected package downgrade: Microsoft.AspNetCore.Razor.Design from 2.2.0 to 2.1.2. Reference the package
directly from the project to select a different version.
```

在 global.json 中更新 .NET Core SDK 版本

如果解决方案依靠 global.json 文件来面向特定 .NET Core SDK 的版本，请将其 `version` 属性更新为你的计算机上安装的 2.2 版本：

```
{
  "sdk": {
    "version": "2.2.100"
  }
}
```

更新启动设置

如果使用 Visual Studio Code 中更新项目的启动设置文件 (*.vscode/launch.json*)。`program` 路径应引用的新 TFM:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            // If you have changed target frameworks, make sure to update the program path.
            "program": "${workspaceFolder}/bin/Debug/netcoreapp2.2/test-app.dll",
            "args": [],
            "cwd": "${workspaceFolder}",
            "stopAtEntry": false,
            "internalConsoleOptions": "openOnSessionStart",
            "launchBrowser": {
                "enabled": true,
                "args": "${auto-detect-url}",
                "windows": {
                    "command": "cmd.exe",
                    "args": "/C start ${auto-detect-url}"
                },
                "osx": {
                    "command": "open"
                },
                "linux": {
                    "command": "xdg-open"
                }
            },
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            },
            "sourceFileMap": {
                "/Views": "${workspaceFolder}/Views"
            }
        },
        {
            "name": ".NET Core Attach",
            "type": "coreclr",
            "request": "attach",
            "processId": "${command:pickProcess}"
        }
    ]
}
```

Kestrel 配置

如果该应用程序调用UseKestrel中 `CreateDefaultBuilder` 方法 `Program` 类中，调用 `ConfigureKestrel` 若要配置 Kestrel 服务器，而不是 `UseKestrel` 以避免与冲突IIS 进程托管模型:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            // Set properties and call methods on options
        });
```

如果应用程序不会调用 `CreateDefaultBuilder` ，调用UseKestrel之前调用 `ConfigureKestrel` :

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseKestrel()
        .UseIISIntegration()
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            // Set properties and call methods on options
        })
        .Build();

    host.Run();
}
```

有关详细信息，请参阅 ASP.NET Core 中的 Kestrel Web 服务器实现。

更新兼容性版本

将 `Startup.ConfigureServices` 中的兼容性版本更新为 `Version_2_2`：

```
services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

更新 CORS 策略

ASP.NET Core 2.2 中使用通配符源的进行响应 CORS 中间件（`*`）如果策略允许任何源，并允许凭据。凭据不支持通配符源时（`*`）指定，则，浏览器将不允许 CORS 请求。有关详细信息，包括用于更正该问题在客户端，请参阅MDN web 文档。

若要更正此问题在服务器上的，执行以下操作之一：

- 修改要不再允许凭据的 CORS 策略。也就是说，删除对调用AllowCredentials时配置的策略。
- 如果凭据所需的 CORS 请求成功执行，修改策略来指定允许的主机。例如，使用 `builder.WithOrigins("https://api.example1.com", "https://example2.com")` 而不是使用AllowAnyOrigin。

更新 Docker 映像

下表显示了 Docker 映像标记更改：

2.1	2.2
`microsoft/dotnet:2.1-aspnetcore-runtime`	`microsoft/dotnet:2.2-aspnetcore-runtime`
`microsoft/dotnet:2.1-sdk`	`microsoft/dotnet:2.2-sdk`

更改 `FROM` 行中您*Dockerfile*以使用上表中的 2.2 列中的新图像标记。

使用 IIS 进程内承载时，手动构建 Visual Studio 中

Visual Studio浏览器请求的自动生成不起作用体验IIS 进程托管模型。使用进程内承载时，必须手动重新生成项目。此体验的改进计划未来版本的 Visual Studio。

更新日志记录代码

- 移动日志记录提供程序初始化到 `Startup.ConfigureServices` 。

  2.1 示例：

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();
}
```

  2.2 示例：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddLogging(builder => builder.AddConsole());
}
```

- 更改筛选，以使用AddLogging生成器。

  2.1 示例：

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(LogLevel.Information);
    // or
    loggerFactory.AddConsole((category, level) =>
        category == "A" || level == LogLevel.Critical);
}
```

  2.2 示例：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddLogging(builder => builder
        .AddConsole()
        .AddFilter<ConsoleLoggerProvider>
            (category: null, level: LogLevel.Information)
        // or
        .AddFilter<ConsoleLoggerProvider>
            ((category, level) => category == "A" || level == LogLevel.Critical)
        );
}
```

- 更改配置加载使用 `AddLogging` 生成器。

  2.1 示例：

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(configuration);
}
```

  2.2 示例：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddLogging(builder => builder
        .AddConfiguration(configuration)
        .AddConsole());
}
```

其他资源

- ASP.NET Core MVC 的兼容性版本
- ASP.NET Core 2.1 及更高版本的 Microsoft.AspNetCore.App 元包
- 隐式包引用

从 ASP.NET Core 2.0 到 2.1 迁移

2018/12/19 • • Edit Online

作者：Rick Anderson

请参阅ASP.NET Core 2.1 中的新增有关 ASP.NET Core 2.1 中的新功能的概述。

这篇文章：

- 介绍如何将 ASP.NET Core 2.0 应用迁移到 2.1 版的基础知识。
- 提供对 ASP.NET Core web 应用程序模板的更改的概述。

若要获取的变动概述 2.1 中的快速方法是：

- 创建名为 WebApp1 的 ASP.NET Core 2.0 web 应用。
- 源代码管理系统中提交 WebApp1。
- 删除 WebApp1 并创建名为 WebApp1 放在同一位置的 ASP.NET Core 2.1 web 应用。
- 查看 2.1 版本中的更改。

本文概述在迁移到 ASP.NET Core 2.1。它不包含将迁移到版本 2.1 所需的所有更改的完整列表。某些项目可能需要更多的步骤，具体取决于创建项目时选择的选项和项目所做的修改。

更新项目文件以使用 2.1 版本

更新项目文件：

- 通过更新项目文件中的将目标框架更改为.NET Core 2.1 `<TargetFramework>netcoreapp2.1</TargetFramework>`。
- 替换的包引用 `Microsoft.AspNetCore.All` 使用的包引用 `Microsoft.AspNetCore.App`。可能需要添加依赖项已从 `Microsoft.AspNetCore.All`。有关详细信息，请参阅 ASP.NET Core 2.0 的 Microsoft.AspNetCore.All 元包 和 ASP.NET Core 2.1 及更高版本的 Microsoft.AspNetCore.App 元包。
- 删除的包引用的"版本"属性 `Microsoft.AspNetCore.App`。项目可使用该技术 `<Project Sdk="Microsoft.NET.Sdk.Web">` 不需要设置的版本。版本将权限隐含的目标框架，并被选为最佳匹配项 ASP.NET Core 2.1 的工作的方式。（请参阅下面的详细信息。）
- 对于面向.NET Framework 的应用程序，每个包引用更新为 2.1。
- 删除对引用 **<DotNetCliToolReference>** 以下包元素。这些工具被捆绑在.NET Core CLI 的默认情况下，并且无需单独安装。
  - Microsoft.DotNet.Watcher.Tools（`dotnet watch`）
  - Microsoft.EntityFrameworkCore.Tools.DotNet（`dotnet ef`）
  - Microsoft.Extensions.Caching.SqlConfig.Tools（`dotnet sql-cache`）
  - 包含 Microsoft.Extensions.SecretManager.Tools（`dotnet user-secrets`）
- 可选：你可以删除 **<DotNetCliToolReference>** 元素 `Microsoft.VisualStudio.Web.CodeGeneration.Tools`。可以使用全局安装的版本替换此工具，通过运行 `dotnet tool install -g dotnet-aspnet-codegenerator`。
- 对于 2.1，Razor 类库是建议的解决方案，用于将 Razor 文件。如果您的应用程序使用嵌入的视图或否则依赖于运行时编译的 Razor 文件，添加 `<CopyRefAssembliesToPublishDirectory>true</CopyRefAssembliesToPublishDirectory>` 到 `<PropertyGroup>` 项目文件中。

以下标记显示了模板生成 2.0 项目文件：

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <UserSecretsId>aspnet-{Project Name}-{GUID}</UserSecretsId>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.3" PrivateAssets="All" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.4"
PrivateAssets="All" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.3" />
    <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.2" />
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.4" />
  </ItemGroup>
</Project>
```

以下标记显示了模板生成 2.1 项目文件：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <UserSecretsId>aspnet-{Project Name}-{GUID}</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.1.1"
PrivateAssets="All" />
  </ItemGroup>

</Project>
```

面向共享的框架的项目的规则

ASP.NET Core 2.1 包括以下共享的框架：

- Microsoft.AspNetCore.App
- Microsoft.AspNetCore.All

指定的包引用的版本是*所需的最小*版本。例如，一个项目引用 2.1.1 仅 2.1.0 一台计算机运行这些包的版本不会安装运行时。

面向共享的框架的项目的已知的问题：

- .NET Core SDK（首先包括在 Visual Studio 15.6）设置的隐式版本 2.1.300 `Microsoft.AspNetCore.App` 到 2.1.0 使用 Entity Framework Core 2.1.1 导致冲突。建议的解决方案是升级到 2.1.301 或更高版本.NET Core SDK。有关详细信息，请参阅Microsoft.AspNetCore.App 与共享依赖项的包不能引用的修补程序版本。
- 必须使用的所有项目 `Microsoft.AspNetCore.All` 或 `Microsoft.AspNetCore.App` 应添加包引用的包在项目文件中，即使它们包含对另一个项目使用的项目引用 `Microsoft.AspNetCore.All` 或 `Microsoft.AspNetCore.App` 。

  示例：

  - `MyApp` 包引用 `Microsoft.AspNetCore.App` 。
  - `MyApp.Tests` 项目引用 `MyApp.csproj` 。

  添加的包引用 `Microsoft.AspNetCore.App` 到 `MyApp.Tests` 。有关详细信息，请参阅集成测试是难以设置，可能会中断上共享的框架服务。

更新到 2.1 的 Docker 映像

在 ASP.NET Core 2.1 中的 Docker 映像迁移到dotnet/dotnet-docker GitHub 存储库。下表显示了 Docker 映像和标记的更改：

2.0	2.1
microsoft / aspnetcore:2.0	microsoft/dotnet:2.1-aspnetcore-runtime
microsoft/aspnetcore 的生成：2.0	microsoft / dotnet:2.1-sdk

更改 `FROM` 行中您*Dockerfile*以使用上表 2.1 列中的新映像名称和标记。有关详细信息，请参阅aspnetcore docker 存储库从迁移到 dotnet。

若要利用 ASP.NET Core 2.1 中建议的新基于代码的惯例的更改

要保留的更改

下图显示的模板化生成所做的更改*Program.cs*文件。

```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

上图中显示为红色的 2.0 版本与删除。

下图显示了 2.1 的代码。绿色中的代码替换为 2.0 版：

```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();

    }
}
```

下面的代码演示的 2.1 版本*Program.cs*:

```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

新 `Main` 替换为调用 `BuildWebHost` 与CreateWebHostBuilder。IWebHostBuilder添加了以支持新集成测试基础结构。

对启动的更改

下面的代码显示了对 2.1 模板生成代码。所有更改新都添加的代码中的，不同之处在于 `UseBrowserLink` 已删除：

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace WebApp1
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>(options =>
            {
                // This lambda determines whether user consent for non-essential cookies is needed for a given
request.
                options.CheckConsentNeeded = context => true;
                options.MinimumSameSitePolicy = SameSiteMode.None;
            });


            services.AddMvc()
                .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            // If the app uses Session or TempData based on Session:
            // app.UseSession();

            app.UseMvc();
        }
    }
}
```

中详细介绍了前面的代码更改:

- 在 ASP.NET Core 支持 GDPR为 `CookiePolicyOptions` 和 `UseCookiePolicy` 。
- HTTP 严格传输安全协议 (HSTS)为 `UseHsts` 。
- 要求使用 HTTPS为 `UseHttpsRedirection` 。
- SetCompatibilityVersion为 `SetCompatibilityVersion(CompatibilityVersion.Version_2_1)` 。

更改身份验证代码

ASP.NET Core 2.1 提供ASP.NET Core 标识作为Razor 类库(RCL)。

默认 2.1 标识 UI 不会对 2.0 版中当前提供重要的新功能。标识替换 RCL 包是可选的。替换模板的优点生成的标识与 RCL 版本的代码包括：

- 多个文件会移出源树。
- 中包含任何 bug 修补程序或标识的新功能Microsoft.AspNetCore.App 元包。你可以自动获取更新后的标识时 `Microsoft.AspNetCore.App` 更新。

如果进行了重要的对模板的更改生成的标识代码：

- 前面的优点却很不证明将转换为 RCL 版。
- 你可以将 ASP.NET Core 2.0 标识代码，完全支持。

标识 2.1 公开终结点与 `Identity` 区域。例如下，表显示将更改从 2.0 到 2.1 的标识终结点的示例：

2.0 URL	2.1 URL
/ 帐户/登录名	/ 标识/Account/Login
/ 帐户/注销	/ 标识/帐户/注销
/ 帐户/管理	/ Identity/帐户/管理

应用程序使用标识的代码并替换 2.0 2.1 标识库需要考虑到标识 Url 标识 UI 具有 `/Identity` 预置到 Uri 的段。一种方法来处理新标识终结点是设置了重定向，例如从 `/Account/Login` 到 `/Identity/Account/Login`。

标识更新到版本 2.1

以下选项可供更新到 2.1 的标识。

- 使用无需更改标识 UI 2.0 代码。完全支持使用标识 UI 2.0 代码。为生成的标识代码进行了重大更改，这是一个不错的方法。
- 删除现有的标识 2.0 代码和基架标识到你的项目。你的项目将使用ASP.NET Core 标识 Razor 类库。你可以生成代码和 UI 的任何修改标识 UI 代码。适用于基架新生成的 UI 代码的代码更改。
- 删除现有的标识 2.0 代码和基架标识到你的项目选项**重写的所有文件**。

替换标识 2.0 使用标识 2.1 Razor 类库的 UI

本部分概述的步骤，以替换与的 ASP.NET Core 2.0 模板生成标识代码ASP.NET Core 标识 Razor 类库。以下步骤适用于 Razor 页面项目，但对于 MVC 项目方法非常相似。

- 验证项目文件更新为使用 2.1 版本
- 删除以下文件夹和在其中的所有文件：
  - *控制器*
  - *页/帐户 /*
  - *扩展插件*
- 生成项目。
- 创建基架标识到你的项目：
  - 选择退出的项目 *_Layout.cshtml*文件。
  - 选择 **+** 右侧的图标 **数据上下文**类。接受默认名称。
  - 选择**添加**创建新的数据上下文类。创建新的数据上下文是必需的搭建基架。下一节中删除新的数据上下文。

标识的基架后更新

- 删除标识基架生成 `IdentityDbContext` 派生类中的 *领域/Identity/Data/* 文件夹。
- 删除*Areas/Identity/IdentityHostingStartup.cs*
- 更新 *_LoginPartial.cshtml*文件：
  - 移动*Pages /_LoginPartial.cshtml*到*页/Shared/_LoginPartial.cshtml*
  - 添加 `asp-area="Identity"` 窗体和定位点的链接。
  - 更新 `<form />` 元素

    ```
    <form asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Page("/Index", new {
    area = "" })" method="post" id="logoutForm" class="navbar-right">
    ```

  下面的代码显示更新后 *_LoginPartial.cshtml*文件：

```
@using Microsoft.AspNetCore.Identity

@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Page("/Index", new {
area = "" })" method="post" id="logoutForm" class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="Identity" asp-page="/Account/Manage/Index" title="Manage">Hello
@UserManager.GetUserName(User)!</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn navbar-link">Log out</button>
            </li>
        </ul>
    </form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="Identity" asp-page="/Account/Register">Register</a></li>
        <li><a asp-area="Identity" asp-page="/Account/Login">Log in</a></li>
    </ul>
}
```

更新 `ConfigureServices` 使用以下代码：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Register no-op EmailSender used by account confirmation and password reset
    // during development
    services.AddSingleton<IEmailSender, EmailSender>();
}
```

更改为 Razor 页面项目 Razor 文件

布局文件

- 移动*Pages /_Layout.cshtml*到*页/Shared/_Layout.cshtml*
- *Layout.cshtml*文件具有以下更改：
  - `<partial name="_CookieConsentPartial" />` 新增了。有关详细信息，请参阅 ASP.NET Core 中的 GDPR 支持。
  - jQuery 从 2.2.0 更改为 3.3.1

_ValidationScriptsPartial.cshtml

- *页/_ValidationScriptsPartial.cshtml*转为*页面/Shared/_ValidationScriptsPartial.cshtml*
- *jquery.validate/1.14.0*更改为*jquery.validate/1.17.0*

新的文件

添加以下文件：

- *Privacy.cshtml*
- *Privacy.cshtml.cs*

请参阅GDPR 支持在 ASP.NET Core 有关前面提到的文件的信息。

对 MVC 项目 Razor 文件的更改

布局文件

*Layout.cshtml*文件具有以下更改：

- `<partial name="_CookieConsentPartial" />` 新增了。
- jQuery 从 2.2.0 更改为 3.3.1

_ValidationScriptsPartial.cshtml

*jquery.validate/1.14.0*更改为*jquery.validate/1.17.0*

新的文件和操作方法

添加以下项：

- *Views/Home/Privacy.cshtml*
- `Privacy` 操作方法添加到 Home 控制器。

请参阅GDPR 支持在 ASP.NET Core 有关前面提到的文件的信息。

对 launchSettings.json 文件的更改

由于 ASP.NET Core 应用程序现在默认情况下都使用 HTTPS *Properties/launchSettings.json*文件已更改。

以下 JSON 显示模板生成的早期 2.0 *launchSettings.json*文件：

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:1799/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApp1": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:1798/"
    }
  }
}
```

以下 JSON 显示模板生成了新 2.1 *launchSettings.json*文件：

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:39191",
      "sslPort": 44390
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApp1": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

有关详细信息，请参阅 强制实施 HTTPS 在 ASP.NET Core 。

其他更改

- 如果承载有 IIS、Windows 上的应用程序安装最新.NET Core 托管捆绑包。

- SetCompatibilityVersion
- 传输配置

从 ASP.NET Core 1.x 迁移到 2.0

2018/10/26 • • Edit Online

作者：Scott Addie

本文演示如何将现有 ASP.NET Core 1.x 项目更新到 ASP.NET Core 2.0。通过将应用程序迁移到 ASP.NET Core 2.0，可利用大量新功能和改进功能。

现有 ASP.NET Core 1.x 应用程序基于版本特定的项目模板。随着 ASP.NET Core 框架不断演变，其中的项目模板和起始代码也在变化。除了更新 ASP.NET Core 框架外，还需要为应用程序更新代码。

系统必备

请参阅 ASP.NET Core 入门。

更新目标框架名字对象 (TFM)

面向 .NET Core 的项目需使用大于或等于 .NET Core 2.0 版本的 TFM。在".csproj"文件中搜索 `<TargetFramework>` 节点，并将其内部文本替换为 `netcoreapp2.0`：

```
<TargetFramework>netcoreapp2.0</TargetFramework>
```

面向 .NET Framework 的项目需使用大于或等于 .NET Framework 4.6.1 版本的 TFM。在".csproj"文件中搜索 `<TargetFramework>` 节点，并将其内部文本替换为 `net461`：

```
<TargetFramework>net461</TargetFramework>
```

> **NOTE**
> 相比于 .NET Core 1.x，.NET Core 2.0 提供更多的外围应用。如果仅因为 .NET Core 1.x 中缺少 API 而要面向 .NET Framework，则定向于 .NET Core 2.0 可能有用。

在 global.json 中更新 .NET Core SDK 版本

如果解决方案依靠 global.json 文件来定向于特定 .NET Core SDK 版本，请更新其 `version` 属性以使用计算机上安装的 2.0 版本：

```
{
  "sdk": {
    "version": "2.0.0"
  }
}
```

更新包引用

1.x 项目中的".csproj"文件列出了该项目使用的每个 NuGet 包。

在面向 .NET Core 2.0 的 ASP.NET Core 2.0 项目中，".csproj"文件中的单个 metapackage 引用将替换包的集合：

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
</ItemGroup>
```

该元包中具备 ASP.NET Core 2.0 和 Entity Framework Core 2.0 的所有功能。

面向 .NET Framework 的 ASP.NET Core 2.0 项目应继续引用单个 NuGet 包。将每个 `<PackageReference />` 节点的 `Version` 特性更新至 2.0.0。

例如，下述列表列出了面向 .NET Framework 的典型 ASP.NET Core 2.0 项目中使用的 `<PackageReference />` 节点：

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Authentication.Cookies" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.0"
PrivateAssets="All" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="2.0.0" />
  <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0"
PrivateAssets="All" />
</ItemGroup>
```

更新 .NET Core CLI 工具

在“.csproj”文件中，将每个 `<DotNetCliToolReference />` 节点的 `Version` 特性更新至 2.0.0。

例如，下述列表列出了面向 .NET Core 2.0 的典型 ASP.NET Core 2.0 项目中使用的 CLI 工具：

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
</ItemGroup>
```

重命名“包目标回退”属性

1.x 项目的“.csproj”文件使用了 `PackageTargetFallback` 节点和变量：

```
<PackageTargetFallback>$(PackageTargetFallback);portable-net45+win8+wp8+wpa81;</PackageTargetFallback>
```

将节点和变量重命名为 `AssetTargetFallback`：

```
<AssetTargetFallback>$(AssetTargetFallback);portable-net45+win8+wp8+wpa81;</AssetTargetFallback>
```

更新 Program.cs 中的 Main 方法

在 1.x 项目中，“Program.cs”的 `Main` 方法如下所示：

```
using System.IO;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore1App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .UseApplicationInsights()
                .Build();

            host.Run();
        }
    }
}
```

在 2.0 项目中，简化了“Program.cs”的 `Main` 方法：

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore2App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

强烈建议采用新的 2.0 模式，Entity Framework (EF) Core 迁移等产品功能需要此模式才能正常运行。例如，从“包管理器控制台”窗口运行 `Update-Database`，或从命令行（位于转换为 ASP.NET Core 2.0 的项目上）运行 `dotnet ef database update` 时，都会生成以下错误：

```
Unable to create an object of type '<Context>'. Add an implementation of
'IDesignTimeDbContextFactory<Context>' to the project, or see https://go.microsoft.com/fwlink/?linkid=851728
for additional patterns supported at design time.
```

添加配置提供程序

在 1.x 项目中，已通过 `Startup` 构造函数将配置提供程序添加到了某个应用。涉及的步骤包括创建 `ConfigurationBuilder` 实例、加载适用的提供程序（环境变量、应用设置等）以及初始化 `IConfigurationRoot` 的成员。

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}

public IConfigurationRoot Configuration { get; }
```

上例使用 psettings.json 以及任何与 `IHostingEnvironment.EnvironmentName` 属性匹配的 appsettings.<EnvironmentName>.json 文件中的配置设置加载 `Configuration` 成员。这些文件所在位置与 Startup.cs 的路径相同。

在 2.0 项目中，样板配置代码会继承在幕后运行的 1.x 代码。例如，启动时就加载环境变量和应用设置。等效的 Startup.cs 代码减少到 `IConfiguration` 初始化设置并包括插入的实例：

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

若要删除由 `WebHostBuilder.CreateDefaultBuilder` 添加的默认提供程序，请对 `ConfigureAppConfiguration` 内的 `IConfigurationBuilder.Sources` 属性调用 `Clear` 方法。若要添加回提供程序，请使用 Program.cs 中的 `ConfigureAppConfiguration` 方法：

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureAppConfiguration((hostContext, config) =>
        {
            // delete all default configuration providers
            config.Sources.Clear();
            config.AddJsonFile("myconfig.json", optional: true);
        })
        .Build();
```

若要查看上一代码片段中 `CreateDefaultBuilder` 方法使用的配置，请参阅此处。

有关详细信息，请参阅 ASP.NET Core 中的配置。

移动数据库初始化代码

在 1.x 项目中使用 EF Core 1.x（类似 `dotnet ef migrations add` 的命令）执行下列任务：

1. 实例化 `Startup` 实例
2. 调用 `ConfigureServices` 方法，为所有服务注册依赖关系注入（包括 `DbContext` 类型）
3. 执行其必要任务

在 2.0 项目中使用 EF Core 2.0，调用 `Program.BuildWebHost` 以获取应用程序服务。与 1.x 不同，这有调用 `Startup.Configure` 的副作用。如果 1.x 应用在其 `Configure` 方法中调用了数据库初始化代码，可能出现意外问题。例如，如果数据库尚不存在，种子设定代码将在 EF Core 迁移命令执行前运行。如果数据库尚不存在，此问题将导致 `dotnet ef migrations list` 命令失败。

请考虑在 *Startup.cs* 的 `Configure` 方法中使用以下 1.x 种子初始化代码。

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});

SeedData.Initialize(app.ApplicationServices);
```

在 2.0 项目中，将 `SeedData.Initialize` 调用移动到 *Program.cs* 的 `Main` 方法：

```
var host = BuildWebHost(args);

using (var scope = host.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    try
    {
        // Requires using RazorPagesMovie.Models;
        SeedData.Initialize(services);
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred seeding the DB.");
    }
}

host.Run();
```

从 2.0 开始，`BuildWebHost` 只应用于生成和配置 Web 主机。有关运行应用程序的任何内容都应在 `BuildWebHost` — 外部处理，通常是在 Program.cs 的 `Main` 方法中。

查看 Razor 视图编译设置

加快应用程序启动速度和缩小已发布的捆绑包至关重要。为此，ASP.NET Core 2.0 中默认启用 Razor 视图编译。

无需再将 `MvcRazorCompileOnPublish` 属性设置为 true。若不禁用视图编译，可能会从“.csproj”文件中删除此属性。

以 .NET Framework 为目标时，仍需显式引用“.csproj”文件中的 Microsoft.AspNetCore.Mvc.Razor.ViewCompilation NuGet 包：

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.0" PrivateAssets="All"
/>
```

依靠 Application Insights“启动”功能

能够轻松设置应用程序性能检测非常重要。现可依靠 Visual Studio 2017 工具中推出的新的 Application Insights“启动”功能。

Visual Studio 2017 中创建的 ASP.NET Core 1.1 项目默认添加 Application Insights。若不直接使用 Application Insights SDK，则除了执行“Program.cs”和“Startup.cs”，还请执行以下步骤：

1. 如果定目标到 .NET Core，请从 .csproj 文件中删除以下 `<PackageReference />` 节点：

```
<PackageReference Include="Microsoft.ApplicationInsights.AspNetCore" Version="2.0.0" />
```

2. 如果定目标到 .NET Core，请从 Program.cs 中删除 `UseApplicationInsights` 扩展方法调用：

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .UseApplicationInsights()
        .Build();

    host.Run();
}
```

3. 从“_Layout.cshtml”中删除 Application Insights 客户端 API 调用。它会比较以下两行代码：

```
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
@Html.Raw(JavaScriptSnippet.FullScript)
```

若要直接使用 Application Insights SDK，请继续此操作。2.0 元包中具备最新版本的 Application Insights，因此如果引用较旧版本，将出现包降级错误。

采用身份验证/标识改进

ASP.NET Core 2.0 具有新的身份验证模型和大量针对 ASP.NET Core 标识的重大更改。如果在启用个人用户帐户的情况下创建了项目，或者已手动添加身份验证或标识，请参阅将身份验证和标识迁移到 ASP.NET Core 2.0。

其他资源

- ASP.NET Core 2.0 中的重大更改

将身份验证和标识迁移到 ASP.NET Core 2.0

2018/12/27 • • Edit Online

通过Scott Addie和Hao 永远

ASP.NET Core 2.0 具有用于身份验证的新模型和标识这简化了使用服务的配置。ASP.NET Core 1.x 应用程序使用身份验证或标识可以更新以使用新的模型，如下所述。

身份验证中间件和服务

在 1.x 项目中，通过中间件配置身份验证。中间件方法调用每个你想要支持的身份验证方案。

下面的 1.x 示例中的标识配置 Facebook 身份验证*Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>();
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory)
{
    app.UseIdentity();
    app.UseFacebookAuthentication(new FacebookOptions {
        AppId = Configuration["auth:facebook:appid"],
        AppSecret = Configuration["auth:facebook:appsecret"]
    });
}
```

在 2.0 项目中，通过服务配置身份验证。在中注册每个身份验证方案 `ConfigureServices` 方法*Startup.cs*。`UseIdentity` 方法替换 `UseAuthentication`。

下面的 2.0 示例中的标识配置 Facebook 身份验证*Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>();

    // If you want to tweak Identity cookies, they're no longer part of IdentityOptions.
    services.ConfigureApplicationCookie(options => options.LoginPath = "/Account/LogIn");
    services.AddAuthentication()
            .AddFacebook(options =>
            {
                options.AppId = Configuration["auth:facebook:appid"];
                options.AppSecret = Configuration["auth:facebook:appsecret"];
            });
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory) {
    app.UseAuthentication();
}
```

`UseAuthentication` 方法将添加一个单一的身份验证中间件组件，它负责进行自动身份验证和远程身份验证请求的处理。它会替换所有单独的中间件组件共有的中间件组件。

下面是有关每个主要身份验证方案的 2.0 迁移说明。

基于 cookie 的身份验证

选择以下两个选项之一并进行必要的更改在*Startup.cs*:

1. 标识与使用 cookie

   - 替换 `UseIdentity` 与 `UseAuthentication` 中 `Configure` 方法:

     ```
     app.UseAuthentication();
     ```

   - 调用 `AddIdentity` 中的方法 `ConfigureServices` 方法添加 cookie 身份验证服务。

   - （可选）调用 `ConfigureApplicationCookie` 或 `ConfigureExternalCookie` 中的方法 `ConfigureServices` 标识 cookie 设置进行调整的方法。

     ```
     services.AddIdentity<ApplicationUser, IdentityRole>()
             .AddEntityFrameworkStores<ApplicationDbContext>()
             .AddDefaultTokenProviders();

     services.ConfigureApplicationCookie(options => options.LoginPath = "/Account/LogIn");
     ```

2. 使用 cookie，而无需标识

   - 替换 `UseCookieAuthentication` 方法中调用 `Configure` 方法替换 `UseAuthentication`:

     ```
     app.UseAuthentication();
     ```

   - 调用 `AddAuthentication` 并 `AddCookie` 中的方法 `ConfigureServices` 方法:

     ```
     // If you don't want the cookie to be automatically authenticated and assigned to
     HttpContext.User,
     // remove the CookieAuthenticationDefaults.AuthenticationScheme parameter passed to
     AddAuthentication.
     services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
             .AddCookie(options =>
             {
                 options.LoginPath = "/Account/LogIn";
                 options.LogoutPath = "/Account/LogOff";
             });
     ```

JWT 持有者身份验证

进行以下更改中的*Startup.cs*:

- 替换 `UseJwtBearerAuthentication` 方法中调用 `Configure` 方法替换 `UseAuthentication`:

  ```
  app.UseAuthentication();
  ```

- 调用 `AddJwtBearer` 中的方法 `ConfigureServices` 方法:

  ```
  services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
          .AddJwtBearer(options =>
          {
              options.Audience = "http://localhost:5001/";
              options.Authority = "http://localhost:5000/";
          });
  ```

此代码片段不使用标识，因此应通过将传递设置的默认方案 `JwtBearerDefaults.AuthenticationScheme` 到 `AddAuthentication` 方法。

OpenID Connect (OIDC) 身份验证

进行以下更改中的*Startup.cs*:

- 替换 `UseOpenIdConnectAuthentication` 方法中调用 `Configure` 方法替换 `UseAuthentication`:

```
app.UseAuthentication();
```

- 调用 `AddOpenIdConnect` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication(options =>
{
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.Authority = Configuration["auth:oidc:authority"];
    options.ClientId = Configuration["auth:oidc:clientid"];
});
```

facebook 身份验证

进行以下更改中的*Startup.cs*:

- 替换 `UseFacebookAuthentication` 方法中调用 `Configure` 方法替换 `UseAuthentication`:

```
app.UseAuthentication();
```

- 调用 `AddFacebook` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication()
        .AddFacebook(options =>
        {
            options.AppId = Configuration["auth:facebook:appid"];
            options.AppSecret = Configuration["auth:facebook:appsecret"];
        });
```

Google 身份验证

进行以下更改中的*Startup.cs*:

- 替换 `UseGoogleAuthentication` 方法中调用 `Configure` 方法替换 `UseAuthentication`:

```
app.UseAuthentication();
```

- 调用 `AddGoogle` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication()
        .AddGoogle(options =>
        {
            options.ClientId = Configuration["auth:google:clientid"];
            options.ClientSecret = Configuration["auth:google:clientsecret"];
        });
```

Microsoft 帐户身份验证

进行以下更改中的*Startup.cs*:

- 替换 `UseMicrosoftAccountAuthentication` 方法中调用 `Configure` 方法替换 `UseAuthentication`:

```
app.UseAuthentication();
```

- 调用 `AddMicrosoftAccount` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication()
        .AddMicrosoftAccount(options =>
        {
            options.ClientId = Configuration["auth:microsoft:clientid"];
            options.ClientSecret = Configuration["auth:microsoft:clientsecret"];
        });
```

Twitter 身份验证

进行以下更改中的*Startup.cs*:

- 替换 `UseTwitterAuthentication` 方法中调用 `Configure` 方法替换 `UseAuthentication`:

```
app.UseAuthentication();
```

- 调用 `AddTwitter` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication()
        .AddTwitter(options =>
        {
            options.ConsumerKey = Configuration["auth:twitter:consumerkey"];
            options.ConsumerSecret = Configuration["auth:twitter:consumersecret"];
        });
```

设置默认身份验证方案

在 1.x 中，`AutomaticAuthenticate` 并 `AutomaticChallenge` 的属性AuthenticationOptions基类用于设置单一身份验证方案。强制实施此没有很好方法。

在 2.0 中，这两个属性已删除作为单个属性 `AuthenticationOptions` 实例。它们可以在中配置 `AddAuthentication` 中的方法调用 `ConfigureServices` 方法*Startup.cs*:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme);
```

在上述代码段中，默认方案设置为 `CookieAuthenticationDefaults.AuthenticationScheme` ("Cookie")。

或者，使用的重载的版本 `AddAuthentication` 方法设置多个属性。在下面的重载的方法示例，默认方案设置为 `CookieAuthenticationDefaults.AuthenticationScheme` 。身份验证方案可能还可以指定在个人 `[Authorize]` 属性或授权策略。

```
services.AddAuthentication(options =>
{
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
});
```

在 2.0 中定义默认方案，如果以下条件之一为 true:

- 您希望用户以进行自动签名
- 您使用 `[Authorize]` 而无需指定方案的属性或授权策略

此规则的例外是 `AddIdentity` 方法。此方法将为你和设置默认值进行身份验证并为应用程序 cookie 质询方案添加 cookie `IdentityConstants.ApplicationScheme` 。此外，它将默认登录方案设置为外部 cookie `IdentityConstants.ExternalScheme` 。

使用 HttpContext 身份验证扩展插件

`IAuthenticationManager` 接口是 1.x 身份验证系统的主入口点。它已替换为一组新的 `HttpContext` 中的扩展方法 `Microsoft.AspNetCore.Authentication` 命名空间。

例如，1.x 项目引用 `Authentication` 属性：

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.Authentication.SignOutAsync(_externalCookieScheme);
```

在 2.0 项目中，导入 `Microsoft.AspNetCore.Authentication` 命名空间，并删除 `Authentication` 属性引用：

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

Windows 身份验证 (HTTP.sys / IISIntegration)

有两种变体的 Windows 身份验证：

1. 主机只允许经过身份验证的用户
2. 主机允许同时匿名和身份验证的用户

上面所述的第一种变化形式是不受 2.0 更改的影响。

2.0 更改的情况下，会影响上面所述的第二个变体。例如，你可能会将以允许匿名用户到你的应用在 IIS 或HTTP.sys层在控制器级别但授权用户。在此方案中，将默认方案设置为 `IISDefaults.AuthenticationScheme` 在 `Startup.ConfigureServices` 方法：

```
services.AddAuthentication(IISDefaults.AuthenticationScheme);
```

相应地设置默认方案失败会阻止在授权请求，以来自工作的挑战。

IdentityCookieOptions 实例

2.0 更改一个副作用是切换到使用名为选项而不是 cookie 选项实例。删除自定义标识 cookie 方案名称的功能。

例如，1.x 项目使用构造函数注入传递 `IdentityCookieOptions` 到参数*AccountController.cs*。从提供的实例访问的外部 cookie 身份验证方案：

```
public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IOptions<IdentityCookieOptions> identityCookieOptions,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _externalCookieScheme = identityCookieOptions.Value.ExternalCookieAuthenticationScheme;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger<AccountController>();
}
```

前面提到的构造函数注入将成为在 2.0 项目中，不必要和 `_externalCookieScheme` 删除字段，可以：

```
public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger<AccountController>();
}
```

`IdentityConstants.ExternalScheme` 可以直接使用常量：

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

添加 POCO IdentityUser 导航属性

Entity Framework (EF) Core 导航属性的基本 `IdentityUser` POCO（普通旧 CLR 对象）已被删除。如果在 1.x 项目使用这些属性，手动将它们添加回 2.0 项目：

```
/// <summary>
/// Navigation property for the roles this user belongs to.
/// </summary>
public virtual ICollection<IdentityUserRole<int>> Roles { get; } = new List<IdentityUserRole<int>>();

/// <summary>
/// Navigation property for the claims this user possesses.
/// </summary>
public virtual ICollection<IdentityUserClaim<int>> Claims { get; } = new List<IdentityUserClaim<int>>();

/// <summary>
/// Navigation property for this users login accounts.
/// </summary>
public virtual ICollection<IdentityUserLogin<int>> Logins { get; } = new List<IdentityUserLogin<int>>();
```

若要防止重复的外键，运行 EF Core 迁移时，将以下代码添加到你 `IdentityDbContext` 类的 `OnModelCreating` 方法（后 `base.OnModelCreating();` 调用）：

```
protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);
    // Customize the ASP.NET Core Identity model and override the defaults if needed.
    // For example, you can rename the ASP.NET Core Identity table names and more.
    // Add your customizations after calling base.OnModelCreating(builder);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Claims)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Logins)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Roles)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);
}
```

替换为 GetExternalAuthenticationSchemes

同步方法 `GetExternalAuthenticationSchemes` 的异步版本，于是取消了。在 1.x 项目具有以下代码*ManageController.cs*:

```
var otherLogins = _signInManager.GetExternalAuthenticationSchemes().Where(auth => userLogins.All(ul =>
auth.AuthenticationScheme != ul.LoginProvider)).ToList();
```

此方法将出现在*Login.cshtml*过：

```
var loginProviders = SignInManager.GetExternalAuthenticationSchemes().ToList();
        <div>
            <p>
                @foreach (var provider in loginProviders)
                {
                    <button type="submit" class="btn btn-default" name="provider"
value="@provider.AuthenticationScheme" title="Log in using your @provider.DisplayName
account">@provider.AuthenticationScheme</button>
                }
            </p>
        </div>
    </form>
}
```

在 2.0 项目中，使用 `GetExternalAuthenticationSchemesAsync` 方法：

```
var schemes = await _signInManager.GetExternalAuthenticationSchemesAsync();
var otherLogins = schemes.Where(auth => userLogins.All(ul => auth.Name != ul.LoginProvider)).ToList();
```

在中*Login.cshtml*，则 `AuthenticationScheme` 中访问属性 `foreach` 循环更改为 `Name`：

```
var loginProviders = (await SignInManager.GetExternalAuthenticationSchemesAsync()).ToList();
        <div>
            <p>
                @foreach (var provider in loginProviders)
                {
                    <button type="submit" class="btn btn-default" name="provider" value="@provider.Name"
title="Log in using your @provider.DisplayName account">@provider.DisplayName</button>
                }
            </p>
        </div>
    </form>
}
```

ManageLoginsViewModel 属性更改

一个 `ManageLoginsViewModel` 中使用对象 `ManageLogins` 的操作*ManageController.cs*。在 1.x 项目中，该对象的 `OtherLogins` 属性的返回类型是 `IList<AuthenticationDescription>` 。此返回类型时需要导入 `Microsoft.AspNetCore.Http.Authentication`：

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Http.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore1App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationDescription> OtherLogins { get; set; }
    }
}
```

在 2.0 项目中，返回类型更改为 `IList<AuthenticationScheme>` 。此新的返回类型需要替换 `Microsoft.AspNetCore.Http.Authentication` 导入具有 `Microsoft.AspNetCore.Authentication` 导入。

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore2App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationScheme> OtherLogins { get; set; }
    }
}
```

其他资源

有关更多详细信息和讨论，请参阅讨论身份验证 2.0 GitHub 上的问题。

从 ASP.NET 迁移到 ASP.NET Core

2018/12/13 • • Edit Online

作者：Isaac Levin

本文可作为从 ASP.NET 应用迁移到 ASP.NET Core 的参考指南。

系统必备

.NET Core SDK 2.2 或更高版本

目标框架

ASP.NET Core 项目为开发人员提供了面向 .NET Core 和/或 .NET Framework 的灵活性。若要确定最合适的目标框架，请参阅为服务器应用选择 .NET Core 或 .NET Framework。

面向 .NET Framework 时，项目需要引用单个 NuGet 包。

得益于有 ASP.NET Core 元包，面向 .NET Core 时可以避免进行大量的显式包引用。在项目中安装 `Microsoft.AspNetCore.App` 元包：

```
<ItemGroup>
   <PackageReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

使用此元包时，应用不会部署元包中引用的任何包。.NET Core 运行时存储中包含这些资产，并已预编译，旨在提升性能。如需了解更多详情，请参阅用于 ASP.NET Core 的 Microsoft.AspNetCore.App 元包。

项目结构差异

ASP.NET Core 中简化了 .csproj 文件格式。下面是一些显著的更改：

- 无需显式添加，即可将文件视作项目的一部分。服务于大型团队时，这可减少出现 XML 合并冲突的风险。
- 没有对其他项目的基于 GUID 的引用，这可以提高文件的可读性。
- 无需在 Visual Studio 中卸载文件即可对它进行编辑：

Global.asax 文件替换

ASP.NET Core 引入了启动应用的新机制。ASP.NET 应用程序的入口点是 Global.asax 文件。路由配置及筛选器和区域注册等任务在 Global.asax 文件中进行处理。

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

此方法会将应用程序和应用程序要部署到的服务器耦合在一起，并且它们的耦合方式会干扰实现。为了将它们分离，引入了 OWIN 来提供一种更为简便的同时使用多个框架的方法。OWIN 提供了一个管道，可以只添加所需的模块。托管环境使用 Startup 函数配置服务和应用的请求管道。`Startup` 在应用程序中注册一组中间件。对于每个请求，应用程序都使用现有处理程序集的链接列表的头指针调用各个中间件组件。每个中间件组件可以向请求处理管道添加一个或多个处理程序。为此，需要返回对成为列表新头的处理程序的引用。每个处理程序负责记住并调用列表中的下一个处理程序。使用 ASP.NET Core 时，应用程序的入口点是 `Startup`，不再具有 Global.asax 的依赖关系。结合使用 OWIN 和 .NET Framework 时，使用的管道应如下所示：

```
using Owin;
using System.Web.Http;

namespace WebApi
{
    // Note: By default all requests go through this OWIN pipeline. Alternatively you can turn this off by
adding an appSetting owin:AutomaticAppStartup with value “false”.
    // With this turned off you can still have OWIN apps listening on specific routes by adding routes in
global.asax file using MapOwinPath or MapOwinRoute extensions on RouteTable.Routes
    public class Startup
    {
        // Invoked once at startup to configure your application.
        public void Configuration(IAppBuilder builder)
        {
            HttpConfiguration config = new HttpConfiguration();
            config.Routes.MapHttpRoute("Default", "{controller}/{customerID}", new { controller = "Customer",
customerID = RouteParameter.Optional });

            config.Formatters.XmlFormatter.UseXmlSerializer = true;
            config.Formatters.Remove(config.Formatters.JsonFormatter);
            // config.Formatters.JsonFormatter.UseDataContractJsonSerializer = true;

            builder.UseWebApi(config);
        }
    }
}
```

这会配置默认路由，默认为 XmlSerialization 而不是 Json。根据需要向此管道添加其他中间件（加载服务、配置设置、静态文件等）。

ASP.NET Core 使用相似的方法，但是不依赖 OWIN 处理条目。相反，这是通过 Program.cs `Main` 方法（类似于控制台应用程序）完成，其中加载了 `Startup`。

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace WebApplication2
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

`Startup` 必须包含 `Configure` 方法。在 `Configure` 中，向管道添加必要的中间件。在下面的示例（来自默认网站模板）中，扩展方法为管道配置以下支持：

- 错误页
- HTTP 严格传输安全
- 从 HTTP 重定向到 HTTPS
- ASP.NET Core MVC

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseMvc();
}
```

现在主机和应用程序已分离，这样将来就可以灵活地迁移到其他平台。

> **NOTE**
>
> 若要获取 ASP.NET Core Startup 和中间件的更深入的参考信息，请参阅 ASP.NET Core 中的 Startup

存储配置

ASP.NET 支持存储设置。这些设置可用于支持应用程序已部署到的环境（以此用途为例）。常见做法是将所有的自定义键值对存储在 Web.config 文件的 `<appSettings>` 部分中：

```
<appSettings>
  <add key="UserName" value="User" />
  <add key="Password" value="Password" />
</appSettings>
```

应用程序使用 `System.Configuration` 命名空间中的 `ConfigurationManager.AppSettings` 集合读取这些设置：

```
string userName = System.Web.Configuration.ConfigurationManager.AppSettings["UserName"];
string password = System.Web.Configuration.ConfigurationManager.AppSettings["Password"];
```

ASP.NET Core 可以将应用程序的配置数据存储在任何文件中，并可在启动中间件的过程中加载它们。项目模板中使用的默认文件是 appsettings.json：

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  // Here is where you can supply custom configuration settings, Since it is is JSON, everything is represented
as key: value pairs
  // Name of section is your choice
  "AppConfiguration": {
    "UserName": "UserName",
    "Password": "Password"
  }
}
```

将此文件加载到应用程序内的 `IConfiguration` 的实例的过程在 Startup.cs 中完成：

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

应用读取 `Configuration` 来获得这些设置：

```
string userName = Configuration.GetSection("AppConfiguration")["UserName"];
string password = Configuration.GetSection("AppConfiguration")["Password"];
```

此方法有扩展项，它们可使此过程更加可靠，例如使用依存关系注入 (DI) 来加载使用这些值的服务。DI 方法提供了一组强类型的配置对象。

```
// Assume AppConfiguration is a class representing a strongly-typed version of AppConfiguration section
services.Configure<AppConfiguration>(Configuration.GetSection("AppConfiguration"));
```

> **NOTE**
>
> 若要获取 ASP.NET Core 配置的更深入的参考信息，请参阅 ASP.NET Core 中的配置。

本机依存关系注入

生成大型可缩放应用程序时，一个重要的目标是将组件和服务松散耦合。依赖项注入不仅是可实现此目标的常用技术，还是 ASP.NET Core 的本机组件。

在 ASP.NET 应用中，开发人员依赖第三方库实现依存关系注入。其中的一个库是 Microsoft 模式和做法提供的 Unity。

实现打包 `UnityContainer` 的 `IDependencyResolver` 是使用 Unity 设置依存关系注入的一个示例：

```
using Microsoft.Practices.Unity;
using System;
using System.Collections.Generic;
using System.Web.Http.Dependencies;

public class UnityResolver : IDependencyResolver
{
    protected IUnityContainer container;

    public UnityResolver(IUnityContainer container)
    {
        if (container == null)
        {
            throw new ArgumentNullException("container");
        }
        this.container = container;
    }

    public object GetService(Type serviceType)
    {
        try
        {
            return container.Resolve(serviceType);
        }
        catch (ResolutionFailedException)
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        try
        {
            return container.ResolveAll(serviceType);
        }
        catch (ResolutionFailedException)
        {
            return new List<object>();
        }
    }

    public IDependencyScope BeginScope()
    {
        var child = container.CreateChildContainer();
        return new UnityResolver(child);
    }

    public void Dispose()
    {
        Dispose(true);
    }

    protected virtual void Dispose(bool disposing)
    {
        container.Dispose();
    }
}
```

创建 `UnityContainer` 的实例，注册服务，然后将 `HttpConfiguration` 的依赖关系解析程序设置为容器的 `UnityResolver` 新实例：

```
public static void Register(HttpConfiguration config)
{
    var container = new UnityContainer();
    container.RegisterType<IProductRepository, ProductRepository>(new HierarchicalLifetimeManager());
    config.DependencyResolver = new UnityResolver(container);

    // Other Web API configuration not shown.
}
```

在必要时注入 `IProductRepository`：

```
public class ProductsController : ApiController
{
    private IProductRepository _repository;

    public ProductsController(IProductRepository repository)
    {
        _repository = repository;
    }

    // Other controller methods not shown.
}
```

由于依存关系注入是 ASP.NET Core 的组成部分，因此可以在 Startup.cs 的 `ConfigureServices` 方法中添加你的服务：

```
public void ConfigureServices(IServiceCollection services)
{
    // Add application services.
    services.AddTransient<IProductRepository, ProductRepository>();
}
```

可在任意位置注入存储库，Unity 亦是如此。

> **NOTE**
> 有关依赖关系注入的详细信息，请参阅依赖关系注入。

提供静态文件

Web 开发的一个重要环节是提供客户端静态资产的功能。HTML、CSS、Javascript 和图像是最常见的静态文件示例。这些文件需要保存在应用（或 CDN）的发布位置中，并且需要引用它们，以便请求可以加载这些文件。在 ASP.NET Core 中，此过程发生了变化。

在 ASP.NET 中，静态文件存储在各种目录中，并在视图中进行引用。

在 ASP.NET Core 中，静态文件存储在“Web 根”（<内容根>/wwwroot）中，除非另有配置。通过从 `Startup.Configure` 调用 `UseStaticFiles` 扩展方法将这些文件加载到请求管道中：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

> **NOTE**
> 如果面向 .NET Framework，请安装 NuGet 包 `Microsoft.AspNetCore.StaticFiles`。

例如，可以通过浏览器从类似 `http://<app>/images/<imageFileName>` 的位置访问 wwwroot/images 文件夹中的图像资产。

> **NOTE**
> 若要获取在 ASP.NET Core 中提供静态文件的更深入的参考信息，请参阅静态文件。

其他资源

- 将库移植到 .NET Core

将从 ASP.NET MVC 迁移到 ASP.NET Core MVC

2018/11/10 • • Edit Online

通过Rick Anderson，Daniel Roth，Steve Smith，和Scott Addie

这篇文章演示如何开始迁移到 ASP.NET MVC 项目ASP.NET Core MVC。在过程中，它重点介绍了许多的 ASP.NET MVC 中已更改的事项。从 ASP.NET MVC 迁移是一个多步骤过程，这篇文章介绍初始设置、基本控制器和视图、静态内容和客户端的依赖项。其他文章介绍了迁移配置和多个 ASP.NET MVC 项目中的标识代码。

> **NOTE**
>
> 这些示例中的版本号可能不是最新。您可能需要相应地更新你的项目。

创建入门级的 ASP.NET MVC 项目

若要演示升级，我们将首先创建一个 ASP.NET MVC 应用。创建同名*WebApp1*使命名空间匹配我们在下一步中创建 ASP.NET Core 项目。

可选：更改从解决方案的名称*WebApp1*到*Mvc5*。Visual Studio 将显示新的解决方案名称 (*Mvc5*)，从而更轻松地判断此项目从下一个项目。

创建 ASP.NET Core 项目

创建一个新空ASP.NET Core 与以前的项目同名的 web 应用 (*WebApp1*) 以便将两个项目中的命名空间匹配。具有相同的命名空间，可以更轻松地将代码在两个项目之间复制。您必须在与上一个项目以使用相同的名称不同的目录中创建此项目。

- *可选*：创建新的 ASP.NET Core 应用使用*Web 应用程序*项目模板。将项目命名*WebApp1*，再选择身份验证选项的单个用户帐户。重命名此应用*FullAspNetCore*。在转换过程中创建此项目为您节省时间。您可以看一下若要查看最终结果，或将代码复制到转换项目模板生成代码。如果您遇到与模板生成项目进行比较的转换步骤，该技术还有助于。

将站点配置为使用 MVC

- 当面向.NET Core Microsoft.AspNetCore.App 元包默认情况下引用。此包包含常用包的包 MVC 应用程序。如果面向.NET Framework，包引用必须单独列出项目文件中。
- 当面向.NET Core Microsoft.AspNetCore.All 元包默认情况下引用。此包包含常用包的包 MVC 应用程序。如果面向.NET Framework，包引用必须单独列出项目文件中。
- 当目标.NET Core 或.NET Framework，包通常用于 MVC 应用程序列出包单独项目文件中。

`Microsoft.AspNetCore.Mvc` 是 ASP.NET Core MVC 框架。`Microsoft.AspNetCore.StaticFiles` 是静态文件处理程序。ASP.NET Core 运行时是一个模块化，和中，你必须显式选择要为静态文件服务（请参阅静态文件）。

- 打开*Startup.cs*文件，并更改代码以匹配以下内容：

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;

namespace WebApp1
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add services to the container.
        // For more information on how to configure your application, visit
https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseStaticFiles();

            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

`UseStaticFiles` 扩展方法将添加静态文件处理程序。正如前面提到的是模块化，ASP.NET 运行时，你必须显式选择中提供静态文件。 `UseMvc` 扩展方法将添加路由。有关详细信息，请参阅应用程序启动并路由。

添加控制器和视图

在本部分中，您将添加一个最小控制器和视图，以用作占位符的 ASP.NET MVC 控制器和视图将在下一节中迁移。

- 添加*控制器*文件夹。
- 添加**控制器**类名为*HomeController.cs*到*控制器*文件夹。

- 添加*视图*文件夹。
- 添加*Views/Home*文件夹。
- 添加**Razor** 视图名为*Index.cshtml*到*Views/Home*文件夹。

项目结构如下所示：

内容替换为*Views/Home/Index.cshtml*具有以下文件：

```
<h1>Hello world!</h1>
```

运行应用。

请参阅控制器并视图有关详细信息。

现在，我们已最小的工作 ASP.NET Core 项目，我们可以开始从 ASP.NET MVC 项目迁移功能。我们需要移动以下：

- 客户端的内容（CSS、字体和脚本）
- 控制器
- 视图
- 模型
- 捆绑
- 筛选器
- 输入/输出的日志，标识（这是在下一步的教程。）

控制器和视图

- 复制每个方法从 ASP.NET MVC `HomeController` 对新 `HomeController` 。请注意，在 ASP.NET MVC 内置模板

的控制器操作方法的返回类型ActionResult; 在 ASP.NET Core MVC，操作方法返回 `IActionResult` 相反。`ActionResult` 实现 `IActionResult`，因此无需更改您的操作方法的返回类型。

- 复制*About.cshtml*，*Contact.cshtml*，和*Index.cshtml* Razor 视图文件从 ASP.NET MVC 项目添加到 ASP.NET Core 项目。
- 运行 ASP.NET Core 应用和测试每个方法。我们还没有迁移样式的布局文件，因此呈现的视图仅包含在视图文件中的内容。不会有的布局文件生成链接 `About` 并 `Contact` 视图，因此您必须调用，从浏览器（替换**4492**与项目中使用的端口号）。
  - `http://localhost:4492/home/about`
  - `http://localhost:4492/home/contact`

请注意，没有样式设置和菜单项。此问题将在下一部分得以解决。

静态内容

在以前版本的 ASP.NET MVC 中，静态内容托管从 web 项目的根目录和服务器端文件混合。在 ASP.NET Core 静态内容承载于*wwwroot*文件夹。你将想要复制的静态内容从旧 ASP.NET MVC 应用程序到*wwwroot* ASP.NET Core 项目文件夹中的。在此示例转换：

- 复制*favicon.ico*文件从旧的 MVC 项目到*wwwroot* ASP.NET Core 项目文件夹中的。

旧 ASP.NET MVC 项目使用Bootstrap Bootstrap 文件在其样式以及存储区*内容*并*脚本*文件夹。模板，用于生成旧的 ASP.NET MVC 项目，引用布局文件中的 Bootstrap（*Views/Shared/_Layout.cshtml*）。无法复制*bootstrap.js*并*bootstrap.css*文件从 ASP.NET MVC 项目到*wwwroot*新项目文件夹中的。相反，我们将添加对 Bootstrap 的支持（和其他客户端库），它在下一节中使用 Cdn。

将布局文件中迁移

- 复制 *_ViewStart.cshtml*文件从旧的 ASP.NET MVC 项目*视图*文件夹导入到 ASP.NET Core项目*视图*文件夹。*_ViewStart.cshtml*文件未更改 ASP.NET Core mvc。
- 创建*Views/Shared*文件夹。
- *可选：*复制 *_ViewImports.cshtml*从*FullAspNetCore* MVC 项目*视图*文件夹导入到 ASP.NET Core 项目*视图*文件夹。中的任何命名空间声明中删除 *_ViewImports.cshtml*文件。*_ViewImports.cshtml*文件的所有视图文件提供了命名空间和带来标记帮助程序。新的布局文件中使用标记帮助程序。*_ViewImports.cshtml*文件是用于 ASP.NET Core 新功能。

- 复制 *_Layout.cshtml*文件从旧的 ASP.NET MVC 项目 *视图/共享*文件夹导入到 ASP.NET Core 项目 *视图/共享*文件夹。

打开 *_Layout.cshtml*文件，然后进行以下更改（如下所示的已完成的代码）：

- 替换 `@Styles.Render("~/Content/css")` 与 `<link>` 元素来加载*bootstrap.css*（见下文）。
- 删除 `@Scripts.Render("~/bundles/modernizr")` 。
- 注释掉 `@Html.Partial("_LoginPartial")` 行（括在行的 `@*...*@` ）。有关详细信息请参阅迁移身份验证和标识到 ASP.NET Core
- 替换 `@Scripts.Render("~/bundles/jquery")` 与 `<script>` 元素（见下文）。
- 替换 `@Scripts.Render("~/bundles/bootstrap")` 与 `<script>` 元素（见下文）。

Bootstrap CSS 包含替换标记：

```
<link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous">
```

JQuery 和 Bootstrap JavaScript 包含的替换标记：

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
    integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
crossorigin="anonymous"></script>
```

已更新 *_Layout.cshtml*文件如下所示：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link rel="stylesheet"
          href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
          integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
          crossorigin="anonymous">
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class =
"navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @*@Html.Partial("_LoginPartial")*@
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    <script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
            integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
            crossorigin="anonymous"></script>
    @RenderSection("scripts", required: false)
</body>
</html>
```

在浏览器中查看该站点。它应现在正确加载，以就地预期样式。

- *可选：*可能想要尝试使用新的布局文件。对于此项目中，你可以复制中的布局文件*FullAspNetCore*项目。新的布局文件使用标记帮助程序并且具有其他改进。

配置捆绑和缩减

有关如何配置绑定和缩减的信息，请参阅捆绑和缩减。

解决 HTTP 500 错误

有许多可能会导致 HTTP 500 错误消息包含没有关于此问题的源的信息的问题。例如，如果*views/_viewimports.cshtml*文件包含在项目中不存在的命名空间，您将收到 HTTP 500 错误。默认情况下，在

ASP.NET Core 应用中，`UseDeveloperExceptionPage` 扩展添加到 `IApplicationBuilder` 和执行在配置后开发。这将在下面的代码中详细说明：

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;

namespace WebApp1
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add services to the container.
        // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?
LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseStaticFiles();

            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

ASP.NET Core 将在 web 应用程序中未经处理的异常转换为 HTTP 500 错误响应。通常情况下，错误详细信息不包括在这些响应来防止泄露有关服务器的潜在敏感信息。请参阅使用开发人员异常页中处理错误有关详细信息。

其他资源

- 客户端开发
- 标记帮助程序

将从 ASP.NET Web API 迁移到 ASP.NET Core

2018/12/11 • • Edit Online

通过Scott Addie和Steve Smith

ASP.NET 4.x Web API 是到达范围广泛的客户端，包括浏览器和移动设备的 HTTP 服务。ASP.NET Core 统一了 ASP.NET 4.x 的 MVC 和 Web API 应用模型到名为 ASP.NET Core MVC 更简单的编程模型。本文演示如何将 ASP.NET 4.x Web API 迁移到 ASP.NET Core MVC 所需的步骤。

查看或下载示例代码（如何下载）

系统必备

- 已安装“ASP.NET 和 Web 开发”工作负载的 Visual Studio 2017 版本 15.9 或更高版本
- .NET Core SDK 2.2 或更高版本

检查 ASP.NET 4.x Web API 项目

作为起点，本文使用*ProductsApp*中创建项目Getting Started with ASP.NET Web API 2。该项目中，一个简单的 ASP.NET 4.x Web API 项目配置，如下所示。

在中*Global.asax.cs*，调用`WebApiConfig.Register`：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Routing;

namespace ProductsApp
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);
        }
    }
}
```

`WebApiConfig`类位于*App_Start*文件夹并且具有静态`Register`方法：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

namespace ProductsApp
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services

            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

此类配置的属性路由，但它实际上未使用项目中。它还可以配置 ASP.NET Web API 使用的路由表。在这种情况下，ASP.NET 4.x Web API 需要 Url 以匹配格式 `/api/{controller}/{id}`，使用 `{id}` 可选的。

*ProductsApp*项目包括一个控制器。在控制器继承 `ApiController` 和包含两个操作：

```
using ProductsApp.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Web.Http;

namespace ProductsApp.Controllers
{
    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product
            {
                Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1
            },
            new Product
            {
                Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M
            },
            new Product
            {
                Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M
            }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
    }
}
```

`Product` 使用模型 `ProductsController` 是一个简单的类：

```
namespace ProductsApp.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

以下部分演示了迁移到 ASP.NET Core MVC Web API 项目。

创建目标项目

完成 Visual Studio 中的以下步骤：

- 转到**文件** > **新** > 项目 > **其他项目类型** > **Visual Studio 解决方案**。选择**空白解决方案**，并将解决方案命

名*WebAPIMigration*。单击**确定**按钮。

- 添加现有*ProductsApp*到解决方案。
- 添加一个新**ASP.NET Core Web 应用程序**到解决方案。选择 **.NET Core**从下拉列表中的框架为目标，然后选择**API**项目模板。将项目命名*ProductsCore*，然后单击**确定**按钮。

该解决方案现在包含两个项目。以下各节介绍了迁移*ProductsApp*项目的内容与*ProductsCore*项目。

迁移配置

不使用 ASP.NET Core *App_Start*文件夹或*Global.asax*文件，并*web.config*文件添加在发布时间。*Startup.cs*将取代*Global.asax*和位于项目根目录中。`Startup`类负责处理所有应用程序启动任务。有关详细信息，请参阅 ASP.NET Core 中的应用启动 。

在 ASP.NET Core MVC 中，在默认情况下包含的属性路由时UseMvc名为`Startup.Configure`。以下`UseMvc`调用替换*ProductsApp*项目的*app_start/webapiconfig.cs*文件：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseMvc();
}
```

迁移模型和控制器

通过复制*ProductApp*项目的控制器和它所使用的模型。请执行以下步骤：

1. 复制*Controllers/ProductsController.cs*从原始到新项目。
2. 复制整个*模型*从原始项目到新的文件夹。
3. 更改复制的文件的命名空间以匹配新的项目名称 (*ProductsCore*)。调整`using ProductsApp.Models;`中的语句*ProductsController.cs*过。

此时，生成中的多个编译错误的应用程序结果。发生错误的原因在 ASP.NET Core 中不存在以下组件：

- `ApiController` 类
- `System.Web.Http` 命名空间
- `IHttpActionResult` 接口

修复错误，如下所示：

1. 更改`ApiController`到ControllerBase。添加`using Microsoft.AspNetCore.Mvc;`若要解决`ControllerBase`引用。
2. 删除`using System.Web.Http;`。
3. 更改`GetProduct`操作的返回类型从`IHttpActionResult`到`ActionResult<Product>`。

简化`GetProduct`操作的`return`与以下语句：

```
return product;
```

配置路由

配置路由，如下所示：

1. 修饰 `ProductsController` 类具有以下属性：

   ```
   [Route("api/[controller]")]
   [ApiController]
   ```

   在前面[Route]属性所配置控制器的属性路由模式。[ApiController]特性使此控制器中的路由所必需的所有操作的属性。

   属性路由支持令牌，如 `[controller]` 和 `[action]` 。在运行时，每个令牌是分别替换为控制器或操作的名称，已向其应用了该属性。令牌减少的项目中的魔幻字符串。路由保持与相应的控制器同步，并将应用时自动重命名重构操作，还请确保令牌。

2. 将项目的兼容性模式设置为 ASP.NET Core 2.2:

   ```
   public void ConfigureServices(IServiceCollection services)
   {
       services.AddMvc()
               .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
   }
   ```

   上述更改：

   - 使用所需 `[ApiController]` 在控制器级别属性。
   - 选择启用可能出现的中断 ASP.NET Core 2.2 中引入的行为。

3. 启用 HTTP Get 请求到 `ProductController` 操作：

   - 将应用[HttpGet]属性为 `GetAllProducts` 操作。
   - 将应用 `[HttpGet("{id}")]` 属性为 `GetProduct` 操作。

上述更改和删除未使用之后 `using` 语句，*ProductsController.cs*文件如下所示：

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using ProductsCore.Models;

namespace ProductsCore.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : ControllerBase
    {
        Product[] products = new Product[]
        {
            new Product
            {
                Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1
            },
            new Product
            {
                Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M
            },
            new Product
            {
                Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M
            }
        };

        [HttpGet]
        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        [HttpGet("{id}")]
        public ActionResult<Product> GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return product;
        }
    }
}
```

运行已迁移的项目，并浏览到 `/api/products` 。将显示三种产品的完整列表。浏览到 `/api/products/1` 。第一个产品将出现。

兼容性修补程序

Microsoft.AspNetCore.Mvc.WebApiCompatShim库提供了兼容性填充程序移动到 ASP.NET Core 的 ASP.NET 4.x Web API 项目。兼容性填充程序扩展了 ASP.NET Core 支持数量从 ASP.NET 4.x Web API 2 的约定。移植之前在本文档中的示例是基本兼容性填充程序是不需要的。对于大型项目，使用兼容性修补程序也可用于临时之间的隔阂 API ASP.NET Core 和 ASP.NET 4.x Web API 2。

旨在用作临时度量值以支持大型 ASP.NET 4.x Web API 项目迁移到 ASP.NET Core Web API 兼容性修补程序。随着时间推移，应更新项目以使用 ASP.NET Core 模式，而不是依靠兼容性填充码。

中包含的兼容性功能 `Microsoft.AspNetCore.Mvc.WebApiCompatShim` 包括：

- 添加 `ApiController` 类型，以便控制器的基类型不需要更新。
- 启用 Web API 样式模型绑定。ASP.NET Core MVC 模型绑定函数类似于的 ASP.NET 4.x MVC 5，默认情况下。

兼容性填充程序更改模型绑定是更类似于 ASP.NET 4.x Web API 2 模型绑定约定。例如，复杂类型自动绑定从请求正文。

- 扩展模型绑定，以便控制器操作可以采取的类型参数 `HttpRequestMessage`。
- 添加消息格式化程序，允许的操作以返回类型的结果 `HttpResponseMessage`。
- 添加 Web API 2 操作可能具有用于为响应提供服务的其他响应方法：
  - `HttpResponseMessage` 生成器：
    - `CreateResponse<T>`
    - `CreateErrorResponse`
  - 结果的操作方法：
    - `BadRequestErrorMessageResult`
    - `ExceptionResult`
    - `InternalServerErrorResult`
    - `InvalidModelStateResult`
    - `NegotiatedContentResult`
    - `ResponseMessageResult`
- 添加的一个实例 `IContentNegotiator` 向应用程序的依赖关系注入 (DI) 容器，并使可从内容协商相关类型Microsoft.AspNet.WebApi.Client。此类类型的示例包括 `DefaultContentNegotiator` 和 `MediaTypeFormatter`。

若要使用兼容性修补程序：

1. 安装Microsoft.AspNetCore.Mvc.WebApiCompatShim NuGet 包。
2. 兼容性修补程序的服务注册应用程序的 DI 容器通过调用 `services.AddMvc().AddWebApiConventions()` 在 `Startup.ConfigureServices`。
3. 定义特定于 API 的路由使用的 web `MapWebApiRoute` 上 `IRouteBuilder` 在应用的 `IApplicationBuilder.UseMvc` 调用。

其他资源

- 使用 ASP.NET Core 构建 Web API
- ASP.NET Core Web API 中控制器操作的返回类型
- ASP.NET Core MVC 的兼容性版本

将配置迁移到 ASP.NET Core

2018/10/30 • • Edit Online

作者：Steve Smith 和 Scott Addie

在以前的文章中，我们就已着手将 ASP.NET MVC 项目迁移到 ASP.NET Core MVC。在本文中，我们将迁移配置。

查看或下载示例代码（如何下载）

安装程序配置

ASP.NET Core 不再使用*Global.asax*和*web.config* ASP.NET 的早期版本使用的文件。在早期版本的 ASP.NET，应用程序启动逻辑已放置在 `Application_StartUp` 方法内的*Global.asax*。更高版本，在 ASP.NET MVC 中，*Startup.cs*文件包含在项目；的根目录，并当应用程序启动时调用。ASP.NET Core 已完全采用这种方法，通过将中的所有启动逻辑*Startup.cs*文件。

*Web.config*还在 ASP.NET Core 替换文件。配置本身现在可以配置，应用程序启动过程中所述*Startup.cs*。配置仍然可以利用 XML 文件，但通常 ASP.NET Core 项目将置于配置值的 JSON 格式文件，如*appsettings.json*。ASP.NET Core 配置系统可以方便地访问环境变量，可以提供更安全、极其可靠的位置特定于环境的值。这是不应签入源代码管理的 API 密钥连接字符串等机密的尤其如此。请参阅配置若要了解有关 ASP.NET Core 中配置的详细信息。

有关本文中，我们开始使用从部分已迁移的 ASP.NET Core 项目上一篇文章。若要设置配置，请添加以下构造函数和属性设置为*Startup.cs*文件位于项目根目录中：

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

请注意，此时，*Startup.cs*文件将无法编译，因为我们仍需要添加以下 `using` 语句：

```
using Microsoft.Extensions.Configuration;
```

添加*appsettings.json*到使用合适的项目模板的项目的根目录的文件：

从 web.config 中迁移的配置设置

我们的 ASP.NET MVC 项目包含中的所需的数据库连接字符串*web.config*，请在 `<connectionStrings>` 元素。在我们 ASP.NET Core 项目中，我们将存储此信息在*appsettings.json*文件。打开*appsettings.json*，并记下它已包含以下：

```
{
    "Data": {
        "DefaultConnection": {
            "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;Trusted_Connection=True;"
        }
    }
}
```

突出显示的行上面所示，在将更改从数据库的名称 **_CHANGE_ME**到数据库的名称。

总结

ASP.NET Core 将置于单个文件，在其中的必要的服务和依赖项可以定义和配置应用程序的所有启动逻辑。它将替换*web.config*具有一种灵活的配置功能，可以利用各种文件格式，如 JSON，以及环境变量文件。

将身份验证和标识迁移到 ASP.NET Core

2018/8/22 • • Edit Online

作者：Steve Smith

在以前的文章中，我们配置从 ASP.NET MVC 项目迁移到 ASP.NET Core MVC。在本文中，我们将迁移的注册、登录名和用户管理功能。

配置标识和成员身份

在 ASP.NET MVC 中，使用 ASP.NET 标识中配置身份验证和标识功能*Startup.Auth.cs*并*IdentityConfig.cs*，它位于*App_Start*文件夹。在 ASP.NET Core MVC，这些功能在中配置*Startup.cs*。

安装 `Microsoft.AspNetCore.Identity.EntityFrameworkCore` 和 `Microsoft.AspNetCore.Authentication.Cookies` NuGet 包。

然后，打开*Startup.cs*，并更新 `Startup.ConfigureServices` 要使用实体框架和标识服务的方法：

```
public void ConfigureServices(IServiceCollection services)
{
    // Add EF services to the services container.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

     services.AddMvc();
}
```

此时，有我们尚未尚未迁移 ASP.NET MVC 项目中的上述代码中引用的两种类型：`ApplicationDbContext` 和 `ApplicationUser`。创建一个新*模型*文件夹中 ASP.NET Core 项目，并将两个类添加到它对应于这些类型。你将找到 ASP.NET MVC 中这些类的版本 */Models/IdentityModels.cs*，但我们将使用每个迁移项目中的类的一个文件，因为这是更清楚。

ApplicationUser.cs:

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

namespace NewMvcProject.Models
{
  public class ApplicationUser : IdentityUser
  {
  }
}
```

ApplicationDbContext.cs:

```
using Microsoft.AspNetCore.Identity.EntityFramework;
using Microsoft.Data.Entity;

namespace NewMvcProject.Models
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
            // Customize the ASP.NET Core Identity model and override the defaults if needed.
            // For example, you can rename the ASP.NET Core Identity table names and more.
            // Add your customizations after calling base.OnModelCreating(builder);
        }
    }
}
```

ASP.NET Core MVC 初学者 Web 项目不包括多的自定义的用户，或 `ApplicationDbContext`。迁移时真正的应用程序，还需要迁移的所有自定义属性和方法的应用程序的用户和 `DbContext` 类，以及您的应用程序利用任何其他模型类。例如，如果你 `DbContext` 已 `DbSet<Album>`，您需要迁移 `Album` 类。

使用这些文件，*Startup.cs*文件可以进行编译，方法是更新其 `using` 语句：

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
```

我们的应用程序现在已准备好支持身份验证和标识服务。它只需要具有向用户公开这些功能。

迁移注册和登录逻辑

使用应用配置的标识服务和数据访问使用 Entity Framework 和 SQL Server 配置，即可向应用添加注册和登录的支持。请记住，迁移过程中更早我们注释掉对的引用 *_LoginPartial*中 *_Layout.cshtml*。现在就返回到该代码中，取消注释，并在必要的控制器和视图以便支持登录功能中添加。

取消注释 `@Html.Partial` 行中 *_Layout.cshtml*:

```
      <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
    </ul>
    @*@Html.Partial("_LoginPartial")*@
  </div>
</div>
```

现在，添加一个名为的新 Razor 视图 *_LoginPartial*到*Views/Shared*文件夹：

更新 *_LoginPartial.cshtml*用下面的代码（替换所有内容）：

```
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="" asp-controller="Account" asp-action="Logout" method="post" id="logoutForm"
class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="" asp-controller="Manage" asp-action="Index" title="Manage">Hello
@UserManager.GetUserName(User)!</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn navbar-link">Log out</button>
            </li>
        </ul>
    </form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="" asp-controller="Account" asp-action="Register">Register</a></li>
        <li><a asp-area="" asp-controller="Account" asp-action="Login">Log in</a></li>
    </ul>
}
```

此时，您应能够刷新您的浏览器中的站点。

总结

ASP.NET Core 引入 ASP.NET 标识功能更改。在本文中，你看到了如何将 ASP.NET 标识的身份验证和用户管理功能迁移到 ASP.NET Core。

迁移 ClaimsPrincipal.Current

2018/8/23 • • Edit Online

在 ASP.NET 4.x 项目中，常见的做法是使用ClaimsPrincipal.Current来检索当前身份验证的用户的标识和声明。在 ASP.NET Core 中，不能再设置此属性。取决于它的代码需要更新，以通过不同方式获取当前经过身份验证的用户的标识。

特定于上下文的数据，而不是静态数据

使用 ASP.NET Core，这两者的值时 `ClaimsPrincipal.Current` 和 `Thread.CurrentPrincipal` 未设置。这些属性都表示静态状态，通常避免 ASP.NET Core。相反，ASP.NET Core 体系结构是检索特定于上下文的服务集合（如当前用户的标识）的依赖关系（使用其依赖关系注入(DI) 模型）。此外，`Thread.CurrentPrincipal` 是线程静态的因此它可能不会持续某些异步方案中的更改（并 `ClaimsPrincipal.Current` 只需调用 `Thread.CurrentPrincipal` 默认情况下）。

若要了解的问题线程类型的静态成员可能会导致在异步方案中，请考虑下面的代码段：

```
// Create a ClaimsPrincipal and set Thread.CurrentPrincipal
var identity = new ClaimsIdentity();
identity.AddClaim(new Claim(ClaimTypes.Name, "User1"));
Thread.CurrentPrincipal = new ClaimsPrincipal(identity);

// Check the current user
Console.WriteLine($"Current user: {Thread.CurrentPrincipal?.Identity.Name}");

// For the method to complete asynchronously
await Task.Yield();

// Check the current user after
Console.WriteLine($"Current user: {Thread.CurrentPrincipal?.Identity.Name}");
```

前面的示例代码集 `Thread.CurrentPrincipal`，并检查其值之前和之后等待的异步调用。`Thread.CurrentPrincipal` 特定于*线程*在其上设置，而且方法是可能会继续在另一个线程在等待后的执行。因此，`Thread.CurrentPrincipal` 存在时将首先检查，但在调用后为 null `await Task.Yield()`。

从应用程序的 DI 服务集合获取当前用户的标识太多可测试性，因为可以轻松地注入测试的标识。

检索当前用户在 ASP.NET Core 应用程序

有几个选项用于检索当前经过身份验证的用户的 `ClaimsPrincipal` 中代替了 ASP.NET Core `ClaimsPrincipal.Current`：

- **ControllerBase.User**。MVC 控制器可以访问当前经过身份验证的用户使用其用户属性。
- **HttpContext.User**。有权访问当前组件 `HttpContext`（中间件）可以获取当前用户的 `ClaimsPrincipal` 从HttpContext.User。
- **从调用方传入**。而无需访问当前库 `HttpContext` 通常称为从控制器或中间件组件，并且能与当前用户的标识作为参数传递。
- **IHttpContextAccessor**。正在迁移到 ASP.NET Core 项目可能太大，无法轻松地将当前用户的标识传递到所有必要的位置。在这种情况下，IHttpContextAccessor可用作一种解决方法。`IHttpContextAccessor` 能够访问当前 `HttpContext`（如果存在）。将获取尚未尚未更新可以使用 ASP.NET Core DI 驱动体系结构的代码中的当前用户的标识的短期解决方案：

- 请 `IHttpContextAccessor` DI 容器通过调用中提供AddHttpContextAccessor中 `Startup.ConfigureServices` 。
- 获取的实例 `IHttpContextAccessor` 在启动过程并将其存储在静态变量中。实例将提供对以前已从静态属性检索当前用户的代码。
- 检索当前用户的 `ClaimsPrincipal` 使用 `HttpContextAccessor.HttpContext?.User` 。如果此代码使用的 HTTP 请求上下文之外 `HttpContext` 为 null。

最后一个选项，请使用 `IHttpContextAccessor` ，行为是违背 ASP.NET Core 原则（首选静态依赖项的插入依赖关系）。计划将最终删除依赖于静态 `IHttpContextAccessor` 帮助器。不过，可以迁移之前使用的大型现有 ASP.NET 应用时它是一个有用的桥， `ClaimsPrincipal.Current` 。

ASP.NET 成员资格身份验证从迁移到 ASP.NET Core 2.0 标识

2019/1/11 • • Edit Online

作者：Isaac Levin

本文演示如何迁移使用成员资格到 ASP.NET Core 2.0 标识的身份验证的 ASP.NET 应用程序的数据库架构。

> **NOTE**
>
> 本文档提供将基于 ASP.NET 成员资格的应用程序的数据库架构迁移到用于 ASP.NET Core 标识的数据库架构所需的步骤。有关从基于 ASP.NET 成员资格的身份验证迁移到 ASP.NET 标识的详细信息，请参阅将现有应用程序从 SQL 成员身份迁移到 ASP.NET 标识。有关 ASP.NET Core 标识的详细信息，请参阅ASP.NET Core 上的标识简介。

查看的成员身份架构

在 ASP.NET 2.0 之前开发人员被负责创建自己的应用的整个身份验证和授权过程。使用 ASP.NET 2.0 成员身份引入时，提供处理 ASP.NET 应用程序中的安全性的样本解决方案。开发人员现在可以启动到与 SQL Server 数据库架构aspnet_regsql.exe命令。运行此命令后，已在数据库中创建以下表。

若要将现有应用迁移到 ASP.NET Core 2.0 标识，这些表中的数据需要迁移到使用新的标识架构的表。

ASP.NET Core 标识 2.0 架构

ASP.NET Core 2.0 遵循标识ASP.NET 4.5 中引入的原则。尽管共享的原则，框架之间的实现是不同的甚至 ASP.NET Core 的版本之间 (请参阅将身份验证和标识迁移到 ASP.NET Core 2.0)。

ASP.NET Core 2.0 标识查看架构的最快方法是创建新的 ASP.NET Core 2.0 应用程序。请按照 Visual Studio 2017 中的执行以下步骤操作：

1. 选择“文件” > “新建” > “项目”。
2. 创建一个新**ASP.NET Core Web** 应用程序名为项目*CoreIdentitySample*。
3. 选择**ASP.NET Core 2.0**在下拉列表中，然后选择**Web** 应用程序。此模板会生成Razor 页面应用。然后再单击**确定**，单击**更改身份**验证。

4. 选择**单个用户帐户**标识模板。最后，单击**确定**，然后**确定**。Visual Studio 创建项目时使用 ASP.NET Core 标识模板。

5. 选择**工具** > **NuGet 包管理器** > **程序包管理器控制台**打开包管理器控制台(PMC) 窗口。

6. 导航到项目根目录中 PMC 中，然后运行Entity Framework (EF) Core `Update-Database`命令。

   ASP.NET Core 2.0 标识使用 EF Core 与存储身份验证数据的数据库进行交互。为了使新创建的应用程序能够，那里需要使用数据库来存储此数据。创建新的应用程序之后，检查的数据库环境中的架构的最快方法是创建数据库使用EF Core 迁移。此过程创建数据库、在本地或其他位置，模拟该架构。查看上述文档，有关详细信息。

   EF Core 命令中指定的数据库使用连接字符串*appsettings.json*。下面的连接字符串上的目标数据库*localhost*名为*asp net core 标识*。在此设置中，EF Core 配置为使用`DefaultConnection`连接字符串。

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=aspnet-core-
identity;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

7. 选择视图 > **SQL Server 对象资源管理器**。展开中指定的数据库名称与对应的节点`ConnectionStrings:DefaultConnection`的属性*appsettings.json*。

   `Update-Database`命令创建与架构指定的数据库和任何所需的应用程序初始化的数据。下图描绘了使用上述步骤创建的表结构。

迁移架构

有细微的差别的表结构和成员身份和 ASP.NET Core 标识字段。模式已显著更改身份验证/授权与 ASP.NET 和 ASP.NET Core 应用程序。仍用于标识的关键对象是*用户*并*角色*。以下是有关映射表*用户*，*角色*，并*UserRoles*。

用户

标识 (DBO。ASPNETUSERS)		*成员资格 (DBO.ASPNET_USERS / DBO.ASPNET_MEMBERSHIP)*	
字段名称	**Type**	**字段名称**	**Type**
`Id`	字符串	`aspnet_Users.UserId`	字符串
`UserName`	字符串	`aspnet_Users.UserName`	字符串

标识 ***(DBO。ASPNETUSERS)***		*成员资格* ***(DBO.ASPNET_USERS / DBO.ASPNET_MEMBERSHIP)***	
`Email`	字符串	`aspnet_Membership.Email`	字符串
`NormalizedUserName`	字符串	`aspnet_Users.LoweredUserName`	字符串
`NormalizedEmail`	字符串	`aspnet_Membership.LoweredEmail`	字符串
`PhoneNumber`	字符串	`aspnet_Users.MobileAlias`	字符串
`LockoutEnabled`	位	`aspnet_Membership.IsLockedOut`	位

> **NOTE**
> 并非所有字段映射类似都于从成员资格到 ASP.NET Core 标识的一对一关系。前面的表使用默认成员资格用户架构，并将其映射到 ASP.NET Core 标识架构。用于成员资格的任何其他自定义字段需要手动映射。在此映射中，则没有密码的映射因为两者之间不迁移密码条件和密码 salt。**建议可保留为 null 的密码，并要求用户重置其密码。**在 ASP.NET Core 标识 `LockoutEnd` 应设置为在将来的某个日期中，如果用户被锁定。迁移脚本所示。

角色

标识 ***(DBO。ASPNETROLES)***		*成员资格* ***(DBO.ASPNET_ROLES)***	
字段名称	**Type**	**字段名称**	**Type**
`Id`	字符串	`RoleId`	字符串
`Name`	字符串	`RoleName`	字符串
`NormalizedName`	字符串	`LoweredRoleName`	字符串

用户角色

标识 ***(DBO。ASPNETUSERROLES)***		*成员资格* ***(DBO.ASPNET_USERSINROLES)***	
字段名称	**Type**	**字段名称**	**Type**
`RoleId`	字符串	`RoleId`	字符串
`UserId`	字符串	`UserId`	字符串

创建的迁移脚本时，引用前面的映射表*用户并角色*。下面的示例假定数据库服务器上有两个数据库。一个数据库包含的现有 ASP.NET 成员身份架构和数据。另*CoreIdentitySample*使用前面所述的步骤创建数据库。注释是以内联形式包含有关更多详细信息。

```
-- THIS SCRIPT NEEDS TO RUN FROM THE CONTEXT OF THE MEMBERSHIP DB
BEGIN TRANSACTION MigrateUsersAndRoles
USE aspnetdb

-- INSERT USERS
INSERT INTO CoreIdentitySample.dbo.AspNetUsers
```

```
INSERT INTO CoreIdentitySample.dbo.AspNetUsers
            (Id,
             UserName,
             NormalizedUserName,
             PasswordHash,
             SecurityStamp,
             EmailConfirmed,
             PhoneNumber,
             PhoneNumberConfirmed,
             TwoFactorEnabled,
             LockoutEnd,
             LockoutEnabled,
             AccessFailedCount,
             Email,
             NormalizedEmail)
SELECT aspnet_Users.UserId,
       aspnet_Users.UserName,
       -- The NormalizedUserName value is upper case in ASP.NET Core Identity
       UPPER(aspnet_Users.UserName),
       -- Creates an empty password since passwords don't map between the 2 schemas
       '',
       /*
        The SecurityStamp token is used to verify the state of an account and
        is subject to change at any time. It should be initialized as a new ID.
       */
       NewID(),
       /*
        EmailConfirmed is set when a new user is created and confirmed via email.
        Users must have this set during migration to reset passwords.
       */
       1,
       aspnet_Users.MobileAlias,
       CASE
         WHEN aspnet_Users.MobileAlias IS NULL THEN 0
         ELSE 1
       END,
       -- 2FA likely wasn't setup in Membership for users, so setting as false.
       0,
       CASE
         -- Setting lockout date to time in the future (1,000 years)
         WHEN aspnet_Membership.IsLockedOut = 1 THEN Dateadd(year, 1000,
                                                     Sysutcdatetime())
         ELSE NULL
       END,
       aspnet_Membership.IsLockedOut,
       /*
        AccessFailedAccount is used to track failed logins. This is stored in
        Membership in multiple columns. Setting to 0 arbitrarily.
       */
       0,
       aspnet_Membership.Email,
       -- The NormalizedEmail value is upper case in ASP.NET Core Identity
       UPPER(aspnet_Membership.Email)
FROM   aspnet_Users
       LEFT OUTER JOIN aspnet_Membership
                    ON aspnet_Membership.ApplicationId =
                       aspnet_Users.ApplicationId
                       AND aspnet_Users.UserId = aspnet_Membership.UserId
       LEFT OUTER JOIN CoreIdentitySample.dbo.AspNetUsers
                    ON aspnet_Membership.UserId = AspNetUsers.Id
WHERE  AspNetUsers.Id IS NULL

-- INSERT ROLES
INSERT INTO CoreIdentitySample.dbo.AspNetRoles(Id, Name)
SELECT RoleId, RoleName
FROM aspnet_Roles;

-- INSERT USER ROLES
INSERT INTO CoreIdentitySample.dbo.AspNetUserRoles(UserId, RoleId)
SELECT UserId, RoleId
```

```
SELECT UserId, RoleId
FROM aspnet_UsersInRoles;

IF @@ERROR <> 0
  BEGIN
    ROLLBACK TRANSACTION MigrateUsersAndRoles
    RETURN
  END

COMMIT TRANSACTION MigrateUsersAndRoles
```

完成操作后的脚本，使用成员资格用户填充前面创建的 ASP.NET Core 标识应用。用户需要在登录之前更改其密码。

> **NOTE**
>
> 如果成员资格系统必须具有不匹配其电子邮件地址的用户名的用户，不需要对创建早期为解决此问题的应用更改。默认模板需要 `UserName` 和 `Email` 相同。它们是不同的情况下，登录过程需要进行修改以使用 `UserName` 而不是 `Email` 。

在中 `PageModel` 的登录页上，位于*Pages\Account\Login.cshtml.cs*，删除 `[EmailAddress]` 属性从*电子邮件*属性。其重命名为*用户名*。这需要更改无论在何处 `EmailAddress` 所述，在*视图*并*PageModel*。结果看起来如下所示：

后续步骤

在本教程中，您学习了如何移植到 ASP.NET Core 2.0 标识从 SQL 成员资格用户。有关 ASP.NET Core 标识的详细信息，请参阅标识简介。

将 HTTP 处理程序和模块迁移到 ASP.NET Core 中间件

2018/8/23 • • Edit Online

通过Matt Perdeck

这篇文章演示如何迁移现有的 ASP.NET HTTP 模块和处理程序 system.webserver到 ASP.NET Core 中间件。

模块和处理程序再次讨论

在继续之前到 ASP.NET Core 中间件，让我们首先会扼要重述 HTTP 模块和处理程序的工作原理：

处理程序是：

- 类实现IHttpHandler
- 用于处理请求的给定的文件名或扩展名，如 。*报表*
- 配置在*Web.config*

模块包括：

- 类实现IHttpModule
- 为每个请求调用
- 能够短路（停止进一步处理的请求）
- 无法添加到 HTTP 响应，或创建其自己
- 配置在*Web.config*

在其中模块处理传入的请求的顺序取决于：

1. 应用程序生命周期，这是由 ASP.NET 激发的系列事件：BeginRequest，AuthenticateRequest，等等。每个模块可以创建一个或多个事件的处理程序。
2. 对于同一个事件，在其中配置中的顺序*Web.config*。

除了模块，可以添加到生命周期事件的处理程序您*Global.asax.cs*文件。已配置的模块中的处理程序后运行这些处理程序。

从处理程序和模块到中间件

中间件是 HTTP 模块和处理程序比简单得多：

- 模块、处理程序、*Global.asax.cs*，*Web.config*（除 IIS 配置）和应用程序生命周期都消失了
- 模块和处理程序的角色具有接管中间件
- 中间件配置为使用代码而不在*Web.config*
- 管道分支允许将请求发送到特定的中间件，根据不仅 URL 还在请求标头、查询字符串，等等。

中间件是非常类似于模块：

- 在为每个请求的主体中调用
- 能够通过短路请求不将请求传递给下一个中间件
- 能够创建他们自己的 HTTP 响应

中间件和模块按不同顺序处理：

- 中间件的顺序基于在其中插入到请求管道，而模块的顺序主要基于的顺序应用程序生命周期事件
- 中间件的响应的顺序是反向从，对于请求，而模块的顺序是相同的请求和响应
- 请参阅使用 IApplicationBuilder 创建中间件管道

请注意如何在上图中中的身份验证中间件设置短路请求。

迁移到中间件模块代码

现有的 HTTP 模块将类似于此：

```
// ASP.NET 4 module

using System;
using System.Web;

namespace MyApp.Modules
{
    public class MyModule : IHttpModule
    {
        public void Dispose()
        {
        }

        public void Init(HttpApplication application)
        {
            application.BeginRequest += (new EventHandler(this.Application_BeginRequest));
            application.EndRequest += (new EventHandler(this.Application_EndRequest));
        }

        private void Application_BeginRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the beginning of request processing.
        }

        private void Application_EndRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the end of request processing.
        }
    }
}
```

中所示中间件页上，ASP.NET Core 中间件是公开的类 `Invoke` 方法拍摄 `HttpContext` 并返回 `Task` 。在新的中间件将如下所示：

```
// ASP.NET Core middleware

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyMiddleware
    {
        private readonly RequestDelegate _next;

        public MyMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task Invoke(HttpContext context)
        {
            // Do something with context near the beginning of request processing.

            await _next.Invoke(context);

            // Clean up.
        }
    }

    public static class MyMiddlewareExtensions
    {
        public static IApplicationBuilder UseMyMiddleware(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyMiddleware>();
        }
    }
}
```

上一部分中执行前面的中间件模板编写中间件。

*MyMiddlewareExtensions*帮助器类，可以更轻松地配置中间件中的你 `Startup` 类。 `UseMyMiddleware` 方法将中间件类添加到请求管道。在中间件的构造函数中注入获取所需的中间件服务。

你的模块可能会终止，例如，如果用户未经授权的请求：

```
// ASP.NET 4 module that may terminate the request

private void Application_BeginRequest(Object source, EventArgs e)
{
    HttpContext context = ((HttpApplication)source).Context;

    // Do something with context near the beginning of request processing.

    if (TerminateRequest())
    {
        context.Response.End();
        return;
    }
}
```

中间件处理这是通过不调用 `Invoke` 管道中的下一个中间件上。请注意这不会完全终止请求，因为响应其返回到管道的方式时，仍要调用前面中间件。

```
// ASP.NET Core middleware that may terminate the request

public async Task Invoke(HttpContext context)
{
    // Do something with context near the beginning of request processing.

    if (!TerminateRequest())
        await _next.Invoke(context);

    // Clean up.
}
```

当你迁移到新中间件模块的功能时，你可能会发现你的代码不会编译，因为 `HttpContext` 类中 ASP.NET Core 已显著更改。更高版本上，你将了解如何将迁移到新的 ASP.NET Core HttpContext。

迁移模块插入到请求管道

HTTP 模块通常会添加到请求管道使用*Web.config*:

```
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <modules>
      <add name="MyModule" type="MyApp.Modules.MyModule"/>
    </modules>
  </system.webServer>
</configuration>
```

将对此进行转换添加新中间件到请求管道中在 `Startup` 类:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions = Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>
();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

在其中插入新中间件在管道中的确切位置取决于它作为模块处理的事件（`BeginRequest`，`EndRequest`等）及其在列表中的模块中的顺序*Web.config*。

如前面所述，没有任何应用程序生命周期中 ASP.NET Core，中间件处理响应的顺序不同于使用模块的顺序。这可能使你订购的决策更具挑战性。

如果排序将为问题，无法将你的模块分为多个中间件组件可单独进行排序。

迁移到中间件的处理程序代码

HTTP 处理程序如下所示：

```
// ASP.NET 4 handler

using System.Web;

namespace MyApp.HttpHandlers
{
    public class MyHandler : IHttpHandler
    {
        public bool IsReusable { get { return true; } }

        public void ProcessRequest(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            context.Response.Output.Write(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.QueryString["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }
}
```

在 ASP.NET Core 项目中，你将翻译以下到中间件类似于以下内容：

```
// ASP.NET Core middleware migrated from a handler

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyHandlerMiddleware
    {

        // Must have constructor with this signature, otherwise exception at run time
        public MyHandlerMiddleware(RequestDelegate next)
        {
            // This is an HTTP Handler, so no need to store next
        }

        public async Task Invoke(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            await context.Response.WriteAsync(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.Query["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }

    public static class MyHandlerExtensions
    {
        public static IApplicationBuilder UseMyHandler(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyHandlerMiddleware>();
        }
    }
}
```

此中间件是非常类似于对应于模块的中间件。唯一的差别是，此处没有不需要调用 `_next.Invoke(context)` 。这样很合理，因为该处理不程序末尾的请求管道，因此将对任何要调用的下一个中间件。

迁移的处理程序插入到请求管道

配置 HTTP 处理程序中完成*Web.config*和外观如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <handlers>
      <add name="MyHandler" verb="*" path="*.report" type="MyApp.HttpHandlers.MyHandler"
resourceType="Unspecified" preCondition="integratedMode"/>
    </handlers>
  </system.webServer>
</configuration>
```

无法将此转换通过将新的处理程序中间件添加到请求管道中在 `Startup` 类，类似于从模块转换为中间件。该方法的问题是，它会将所有请求都发送到新的处理程序中间件。但是，你只想与给定扩展插件的请求来访问中间件。可让你已经有了 HTTP 处理程序使用相同的功能。

一种解决方案是分支的扩展名为给定的请求管道使用 `MapWhen` 扩展方法。执行此操作在同一个 `Configure` 方法，添加在其他中间件：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions = Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>
();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

`MapWhen` 使用这些参数:

1. 采用的 lambda `HttpContext` ，并返回 `true` 如果请求应停机分支。这意味着可以分支请求而不仅仅是基于其的扩展名，而且还在请求标头、查询字符串参数，等等。
2. 采用的 lambda `IApplicationBuilder` ，并将添加的分支的所有中间件。这意味着您可以添加其他中间件到分支处理程序中间件的前面。

将所有请求; 调用分支之前，中间件添加到管道分支会对其产生任何影响。

正在加载使用选项模式的中间件选项

一些模块和处理程序具有中存储的配置选项*Web.config*。但是，在 ASP.NET Core 中新的配置模型使用代替了*Web.config*。

新配置系统为你提供这些选项，若要解决此问题：

- 直接注入到中间件的选项，如中所示下一节。
- 使用选项模式：

1. 创建一个类来保存中间件的选项，例如：

```
public class MyMiddlewareOptions
{
    public string Param1 { get; set; }
    public string Param2 { get; set; }
}
```

2. 存储选项的值

   配置系统，可存储选项任意位置所需的值。但是，多数场所使用*appsettings.json*，因此我们将这种方法：

```
{
  "MyMiddlewareOptionsSection": {
    "Param1": "Param1Value",
    "Param2": "Param2Value"
  }
}
```

   *MyMiddlewareOptionsSection*下面是部分名称。它不一定要与选项类的名称相同。

3. 与选项类关联的选项值

   选项模式使用 ASP.NET Core 依赖关系注入框架将选项类型相关联（如`MyMiddlewareOptions`）与`MyMiddlewareOptions`具有实际选项对象。

   更新你`Startup`类：

   a. 如果您使用的*appsettings.json*，将其添加到中的配置生成器`Startup`构造函数：

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

   b. 配置选项服务：

```
public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
    services.AddMvc();
}
```

c. 将你的选项与选项类相关联：

```
public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
    services.AddMvc();
}
```

4. 注入到中间件构造函数的选项。这是类似于将注入到控制器的选项。

```
public class MyMiddlewareWithParams
{
    private readonly RequestDelegate _next;
    private readonly MyMiddlewareOptions _myMiddlewareOptions;

    public MyMiddlewareWithParams(RequestDelegate next,
        IOptions<MyMiddlewareOptions> optionsAccessor)
    {
        _next = next;
        _myMiddlewareOptions = optionsAccessor.Value;
    }

    public async Task Invoke(HttpContext context)
    {
        // Do something with context near the beginning of request processing
        // using configuration in _myMiddlewareOptions

        await _next.Invoke(context);

        // Do something with context near the end of request processing
        // using configuration in _myMiddlewareOptions
    }
}
```

UseMiddleware添加到中间件的扩展方法 `IApplicationBuilder` 负责处理的依赖关系注入。

这并不局限于 `IOptions` 对象。这种方式，可以注入中间件需要的任何其他对象。

加载通过直接注入的中间件选项

选项模式具有的优势在于它会创建松散耦合选项值和其使用者之间。一旦与实际选项值关联的选项类，其他任何

类可获得访问通过依赖关系注入框架的选项。没有无需传递选项值。

这将分解但如果想要使用不同选项两次，使用同一个中间件。例如授权中间件使用允许不同的角色的不同分支中。不能将两个不同的选项对象与一个选项类相关联。

解决方法是获取使用中的实际选项值的选项对象在 `Startup` 类，并将这些直接向中间件的每个实例。

1. 添加到第二个密钥*appsettings.json*

   若要添加另一组选项*appsettings.json*文件，请使用新密钥来唯一识别它：

   ```
   {
     "MyMiddlewareOptionsSection2": {
       "Param1": "Param1Value2",
       "Param2": "Param2Value2"
     },
     "MyMiddlewareOptionsSection": {
       "Param1": "Param1Value",
       "Param2": "Param2Value"
     }
   }
   ```

2. 检索选项值，并将其传递到中间件。`Use...`（其中向管道添加中间件）扩展方法是将选项值传递的合理位置：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions =
Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

3. 启用中间件需要包含 options 参数。提供的一个重载 `Use...` 扩展方法 (它接受的 options 参数并将其传递给 `UseMiddleware`)。当 `UseMiddleware` 调用带参数，它将参数传递给中间件构造函数实例化中间件对象时。

```
public static class MyMiddlewareWithParamsExtensions
{
    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>();
    }

    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder, MyMiddlewareOptions myMiddlewareOptions)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>(
            new OptionsWrapper<MyMiddlewareOptions>(myMiddlewareOptions));
    }
}
```

请注意这将中的选项对象的包装 `OptionsWrapper` 对象。这会实现 `IOptions`，如所需的中间件构造函数。

迁移到新的 HttpContext

你此前看到的 `Invoke` 中间件中的方法采用一个参数类型的 `HttpContext`：

```
public async Task Invoke(HttpContext context)
```

`HttpContext` 已显著更改 ASP.NET Core 中。本部分演示如何将最常使用的属性转换System.Web.HttpContext对新 `Microsoft.AspNetCore.Http.HttpContext`。

HttpContext

Httpcontext.items 进行更换会转换为：

```
IDictionary<object, object> items = httpContext.Items;
```

唯一请求 ID（没有 System.Web.HttpContext 对应）

为你的唯一 id 用于每个请求。包括在日志中非常有用。

```
string requestId = httpContext.TraceIdentifier;
```

HttpContext.Request

HttpContext.Request.HttpMethod会转换为：

```
string httpMethod = httpContext.Request.Method;
```

HttpContext.Request.QueryString会转换为：

```
IQueryCollection queryParameters = httpContext.Request.Query;

// If no query parameter "key" used, values will have 0 items
// If single value used for a key (...?key=v1), values will have 1 item ("v1")
// If key has multiple values (...?key=v1&key=v2), values will have 2 items ("v1" and "v2")
IList<string> values = queryParameters["key"];

// If no query parameter "key" used, value will be ""
// If single value used for a key (...?key=v1), value will be "v1"
// If key has multiple values (...?key=v1&key=v2), value will be "v1,v2"
string value = queryParameters["key"].ToString();
```

HttpContext.Request.Url并HttpContext.Request.RawUrl将转换为:

```
// using Microsoft.AspNetCore.Http.Extensions;
var url = httpContext.Request.GetDisplayUrl();
```

HttpContext.Request.IsSecureConnection会转换为:

```
var isSecureConnection = httpContext.Request.IsHttps;
```

HttpContext.Request.UserHostAddress会转换为:

```
var userHostAddress = httpContext.Connection.RemoteIpAddress?.ToString();
```

HttpContext.Request.Cookies会转换为:

```
IRequestCookieCollection cookies = httpContext.Request.Cookies;
string unknownCookieValue = cookies["unknownCookie"]; // will be null (no exception)
string knownCookieValue = cookies["cookie1name"];     // will be actual value
```

HttpContext.Request.RequestContext.RouteData会转换为:

```
var routeValue = httpContext.GetRouteValue("key");
```

HttpContext.Request.Headers会转换为:

```
// using Microsoft.AspNetCore.Http.Headers;
// using Microsoft.Net.Http.Headers;

IHeaderDictionary headersDictionary = httpContext.Request.Headers;

// GetTypedHeaders extension method provides strongly typed access to many headers
var requestHeaders = httpContext.Request.GetTypedHeaders();
CacheControlHeaderValue cacheControlHeaderValue = requestHeaders.CacheControl;

// For unknown header, unknownheaderValues has zero items and unknownheaderValue is ""
IList<string> unknownheaderValues = headersDictionary["unknownheader"];
string unknownheaderValue = headersDictionary["unknownheader"].ToString();

// For known header, knownheaderValues has 1 item and knownheaderValue is the value
IList<string> knownheaderValues = headersDictionary[HeaderNames.AcceptLanguage];
string knownheaderValue = headersDictionary[HeaderNames.AcceptLanguage].ToString();
```

HttpContext.Request.UserAgent会转换为:

```
string userAgent = headersDictionary[HeaderNames.UserAgent].ToString();
```

HttpContext.Request.UrlReferrer会转换为:

```
string urlReferrer = headersDictionary[HeaderNames.Referer].ToString();
```

HttpContext.Request.ContentType会转换为:

```
// using Microsoft.Net.Http.Headers;

MediaTypeHeaderValue mediaHeaderValue = requestHeaders.ContentType;
string contentType = mediaHeaderValue?.MediaType.ToString();   // ex. application/x-www-form-urlencoded
string contentMainType = mediaHeaderValue?.Type.ToString();    // ex. application
string contentSubType = mediaHeaderValue?.SubType.ToString();  // ex. x-www-form-urlencoded

System.Text.Encoding requestEncoding = mediaHeaderValue?.Encoding;
```

HttpContext.Request.Form会转换为:

```
if (httpContext.Request.HasFormContentType)
{
    IFormCollection form;

    form = httpContext.Request.Form; // sync
    // Or
    form = await httpContext.Request.ReadFormAsync(); // async

    string firstName = form["firstname"];
    string lastName = form["lastname"];
}
```

> **WARNING**
>
> 仅当内容的子类型为读取窗体值*x-www 的窗体的 urlencoded或窗体数据*。

HttpContext.Request.InputStream会转换为:

```
string inputBody;
using (var reader = new System.IO.StreamReader(
    httpContext.Request.Body, System.Text.Encoding.UTF8))
{
    inputBody = reader.ReadToEnd();
}
```

> **WARNING**
>
> 仅在处理程序类型中间件在管道末尾使用此代码。
>
> 每个请求仅一次如上所示，可以读取原始正文。尝试在第一次读取后读取正文的中间件将读取正文为空。
>
> 这不适用于读取窗体，如前文所述，因为这缓冲区中完成。

HttpContext.Response

HttpContext.Response.Status并HttpContext.Response.StatusDescription将转换为:

```
// using Microsoft.AspNetCore.Http;
httpContext.Response.StatusCode = StatusCodes.Status200OK;
```

HttpContext.Response.ContentEncoding并HttpContext.Response.ContentType将转换为:

```
// using Microsoft.Net.Http.Headers;
var mediaType = new MediaTypeHeaderValue("application/json");
mediaType.Encoding = System.Text.Encoding.UTF8;
httpContext.Response.ContentType = mediaType.ToString();
```

HttpContext.Response.ContentType上其自身还会转换为:

```
httpContext.Response.ContentType = "text/html";
```

HttpContext.Response.Output会转换为:

```
string responseContent = GetResponseContent();
await httpContext.Response.WriteAsync(responseContent);
```

HttpContext.Response.TransmitFile

提供一个文件进行了讨论此处。

HttpContext.Response.Headers

发送响应标头十分复杂，这一事实，如果您将它们设置到响应正文写入任何内容后，它们将不会发送。

解决方案是设置将写入到响应开始之前先调用右侧的回调方法。最好的做法是在开头`Invoke`中间件中的方法。它是此回调方法，设置响应标头。

下面的代码设置调用的回调方法`SetHeaders`:

```
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

`SetHeaders`回调方法将如下所示:

```
// using Microsoft.AspNet.Http.Headers;
// using Microsoft.Net.Http.Headers;

private Task SetHeaders(object context)
{
    var httpContext = (HttpContext)context;

    // Set header with single value
    httpContext.Response.Headers["ResponseHeaderName"] = "headerValue";

    // Set header with multiple values
    string[] responseHeaderValues = new string[] { "headerValue1", "headerValue1" };
    httpContext.Response.Headers["ResponseHeaderName"] = responseHeaderValues;

    // Translating ASP.NET 4's HttpContext.Response.RedirectLocation
    httpContext.Response.Headers[HeaderNames.Location] = "http://www.example.com";
    // Or
    httpContext.Response.Redirect("http://www.example.com");

    // GetTypedHeaders extension method provides strongly typed access to many headers
    var responseHeaders = httpContext.Response.GetTypedHeaders();

    // Translating ASP.NET 4's HttpContext.Response.CacheControl
    responseHeaders.CacheControl = new CacheControlHeaderValue
    {
        MaxAge = new System.TimeSpan(365, 0, 0, 0)
        // Many more properties available
    };

    // If you use .Net 4.6+, Task.CompletedTask will be a bit faster
    return Task.FromResult(0);
}
```

HttpContext.Response.Cookies

Cookie 的浏览器中为旅行*Set-cookie*响应标头。因此，发送 cookie 用于发送响应标头需要使用相同的回调：

```
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetCookies, state: httpContext);
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

`SetCookies` 回调方法将如下所示：

```
private Task SetCookies(object context)
{
    var httpContext = (HttpContext)context;

    IResponseCookies responseCookies = httpContext.Response.Cookies;

    responseCookies.Append("cookie1name", "cookie1value");
    responseCookies.Append("cookie2name", "cookie2value",
        new CookieOptions { Expires = System.DateTime.Now.AddDays(5), HttpOnly = true });

    // If you use .Net 4.6+, Task.CompletedTask will be a bit faster
    return Task.FromResult(0);
}
```

其他资源

- HTTP 处理程序和 HTTP 模块概述

- 配置
- 应用程序启动
- 中间件

将从 2.1 到 2.2 或 3.0 Microsoft.Extensions.Logging 迁移

2019/1/8 • • Edit Online

本文概述了用于迁移使用的非 ASP.NET Core 应用程序的常见步骤 `Microsoft.Extensions.Logging` 从 2.1 到 2.2 或 3.0。

2.1 到 2.2

手动创建 `ServiceCollection`，并调用 `AddLogging`。

2.1 示例：

```
using (var loggerFactory = new LoggerFactory())
{
    loggerFactory.AddConsole();

    // use loggerFactory
}
```

2.2 示例：

```
var serviceCollection = new ServiceCollection();
serviceCollection.AddLogging(builder => builder.AddConsole());

using (var serviceProvider = serviceCollection.BuildServiceProvider())
using (var loggerFactory = serviceProvider.GetService<ILoggerFactory>())
{
    // use loggerFactory
}
```

2.1 到 3.0

在 3.0 中，使用 `LoggingFactory.Create`。

2.1 示例：

```
using (var loggerFactory = new LoggerFactory())
{
    loggerFactory.AddConsole();

    // use loggerFactory
}
```

3.0 的示例：

```
using (var loggerFactory = LoggerFactory.Create(builder => builder.AddConsole()))
{
    // use loggerFactory
}
```

其他资源

ASP.NET Core 中的日志记录