1、Web是依赖IIS部署，ASP.NET中是没有Program的，而Core中有了Program.cs文件。这是为什么？因为ASP.NET Core有一个自带的服务器，叫做Kestrel（支持反向代理服务器：IIS、Nginx、Apache 结合使用：【反向代理服务器】先接收来自网络的HTTP请求，然后再将这些请求转发但Kestrel，最后由Kestrel来实现调用）。所以Program.cs是用来启动Kestrel这个服务器的。

2、就静态文件而言，ASP.NET Core运行时会区分内容根文件夹和Web根文件夹。

内容根文件夹一般是项目当前的目录，在生产中就是部署的根文件夹。它代表代码需要执行的所有文件搜索和访问的基础路径。

与之相对，Web根文件夹是应用程序可能提供给Web客户端的所有静态文件的基础路径。

一般来说，Web根文件夹是内容根文件夹的子文件夹，被命名为wwwroot。

3、网站文件目录

wwwroot：网站的静态文件目录，是ASP.NET Core Web应用程序唯一可以访问的资源文件夹。

appsetings.json：（变量）配置文件，比如数据库连接字符串等等配置信息

Program.cs：程序入口文件（里面有个Main方法）

Startup.cs：启动配置文件

依赖项：管理项目所依赖的第三方组件的安装，配置，升级

Controller：控制器

Models：实体

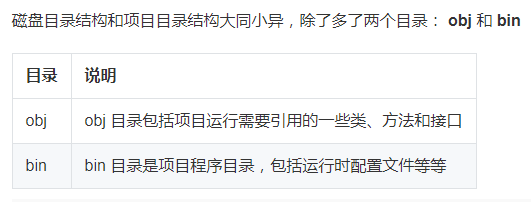
Views：视图



Program.cs 时讲到 Program 类会实例化 Startup 类

而ASP.NET Core 中大大小小的各个组件和中间件都会和 Startup 类打交道





**Startup 类**

Startup 类可以用来定义请求处理管道和配置应用程序需要的服务。

Startup 类必须是公开的，且必须包含以下两个方法

1.ConfigureServices() 方法

public void ConfigureServices(IServiceCollection services){}

ConfigureServices() 方法用于定义应用程序所需要的服务，例如 ASP.NET Core MVC 、 Entity Framework Core 和 Identity 等等

2.Configure() 方法

public void Configure(IApplicationBuilder app, IHostingEnvironment env){}

Configure() 用于定义请求管道中的中间件

Startup 类中的 Configure() 方法用于配置和构建 HTTP 请求管道的地方。其实也就是说，Configure() 方法可以用来定义我们的应用程序如何响应请求其实也就是说，Configure() 方法可以用来定义我们的应用程序如何响应请求

如果我们希望应用程序的行为不同，我们需要在 Configure() 方法中添加其他代码来更改管道

例如，如果我们想要为诸如 index.html 之类的静态文件提供服务，则需要向 Configure() 方法添加一些代码

例如，如果想要给 ASP.NET MVC 控制器发送错误页面或路由请求，都需要在这个 Configure() 方法中做一些工作

## 动态响应内容

Startup 类中的 Configure() 方法用于定义请求管道中的中间件

ASP.NET Core 中的中间件控制我们的应用程序如何响应 HTTP 请求，它还可以控制我们的应用程序在发生错误时的显示的内容，它是我们认证和授权用户执行特定操作的关键部分

**中间件**

那么，什么是中间件呢？

中间件是一种装配到应用程序管道以处理请求和响应的组件。

每个组件：

1. 可以选择是否将请求传递到管道中的下一个组件。
2. 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。

可以使用 Run、Map 和 Use 扩展方法来配置请求委托。

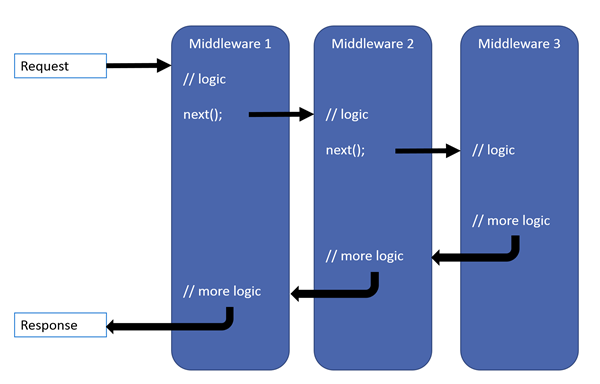
可将一个单独的请求委托并行指定为匿名方法 ( 称为并行中间件 ) ，或在可重用的类中对其进行定义

这些可重用的类和并行匿名方法即为中间件或中间件组件

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，或在适当情况下使链发生短路

## 中间件处理流程

ASP.NET Core 请求管道包含一系列相继调用的请求委托，如下图所示（执行过程遵循黑色箭头）



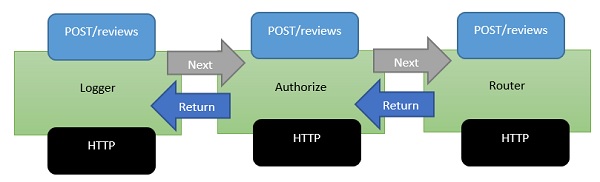
每个委托均可在下一个委托前后执行操作。

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。

我们假设想要记录每个请求的信息，而且想要验证每个请求是否被授权，然后才发送到相应的控制器中去处理

1. 在这种情况下，安装到应用程序中的第一件中间件可能就是日志记录组件
2. 记录器可以看到关于传入请求的所有信息，但可能只是记录一些信息， 然后将这个请求传递给下一个中间件



1. 中间件是这个处理流水线中存在的一系列组件
2. 安装到应用程序中的下一个中间件是授权组件
3. 授权组件可能会在 HTTP 标头中寻找特定的 Cookie 信息或访问令牌
4. 如果授权组件发现令牌有效，则继续传递请求给下一个中间件路由器。如果无效，则可以授权组件本身可能会用HTTP 错误代码或重定向代码来响应请求以将用户引导到到登录页面
5. 路由器中间件查看 URL 并确定下一步动作
6. 路由器中间件在应用程序中查找响应的内容，如果路由器没有找到任何响应，则路由器本身可能会返回 **404 Not Found** 错误

controllers文件夹：用于存放所有的控制器

HomeController类为默认控制器，也就是访问 / 时默认使用HomeController来处理

## 路由

如何将请求路由到不同的控制器

首先，ASP.NET Core 中间件需要一个方法来确定给定的 HTTP 请求是否应该发送给控制器进行处理，我们将这个过程称之为路由匹配

MVC 中间件将根据我们提供的 URL 和一些配置信息做出此决定

定义这些配置信息，或者当我们添加 MVC 中间件时，可以在 Startup.cs 中说明路由信息

这种方法通常被称为基于约定的路由。

以下代码是常规路由的代码片段

routeBuilder.MapRoute("Default", "{controller=Home}/{action=Index}/{id?}");

上面的代码中，我们定义了一个路由正则，告诉 MVC 如何查看 URL 并找到控制器名称和操作名称，其中控制器是 C＃ 类，操作是该类上的公共方法

( HomeController ) ，它是一个 C# 类，不需要从基类派生或实现接口或具有任何特殊属性。 它是一个纯 C#类，名称为 HomeController，它包含值的 Index() 方法

路由的目标是描述 ASP.NET Core MVC 用于处理 HTTP 请求并找到可响应该请求的控制器的规则

1. 我们使用一条路由规则将请求映射到不同的控制器
2. 我们告诉 routeBuilder 我们想要映射一个新的路由，它的名字是 Default，然后提供最重要的路由信息​​，路由模板
3. 路由模板是一个字符串，它用于向 ASP.NET Core MVC 描述如何拆分 URL
4. 在前面的章节中，我们添加了一个 HomeController，因此我们可以请求以下任何 URL，并且它们也将被定向到 HomeController 上的 Index 操作
   1. https://localhost:5001
   2. https://localhost:5001/Home
   3. https://localhost:5001/Home/Index
5. 当浏览器请求 http://mysite/ 或 http://mysite/ Home 时，它将得到 HomeController 的 Index 方法的输出的内容
6. 我们可以通过更改浏览器中的 URL 来尝试此操作。在这个例子中，它是 https:// localhost:5001/ ，可能端口号有所不同
7. 如果我们将 /Home 或 /Home/Index 追加到 URL 并按下 Enter 按钮，也会看到相同的结果
8. id 末尾的问号表示该参数是可选的。换句话说，ASP.NET Core MVC 在这里不需要看到某种类型的 id，可能是一个数字，也可能是一个字符串或 GUID

**属性路由**

通过基于属性的路由，我们可以在控制器类和这些类的内部方法上使用 C# 属性。 这些属性携带了告诉 ASP.NET Core 何时调用特定控制器的元数据

1. 属性路由是基于约定的路由的替代方案
2. 路由按照它们出现的顺序进行评估，也就是我们注册它们的顺序，映射多个路由的情况相当普遍，特别是如果我们想在 URL 中使用不同的参数或者如果要在 URL 中使用不同的文字

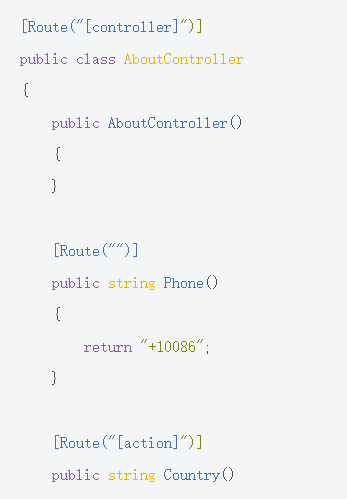


在这里，我们给 Phone() 方法添加了一个空的路由属性，这意味用户只需要访问 /about，而不需要指定操作就可以访问此方法。对于 country 方法，我们在路由属性中指定了 country

保存下 AboutController.cs ，刷新浏览器并访问 /about，我们可以看到正常输出了电话号码

如果我们访问 /about/country ，那么这将访问 AboutController 控制器中的 Country() 方法

如果希望 URL 的一段包含我们的控制器的名称，那么我们可以不直接显示指定控制器的名称，取而代之的是在方括号内使用令牌控制器，这用于告诉 ASP.NET MVC 在此位置使用此控制器的名称



使用这种方式，即使我们重命名路由器，也不必去更改路由。 对于一个动作也是一样，并且隐式地在控制器和动作之间有一个斜杠 ( / ) 。 它是控制器和动作之间的层次关系，就像它在 URL 中一样

## 动作结果 ( Action Result )

Microsoft.AspNetCore.Mvc 命名空间下的基类 Controller 让我们能够访问很多关于 HTTP 请求的上下文信息，以及提供了一些方法帮助我​​们构建返回给回客户端的结果

返回的响应的结果中，我们可以发送简单的字符串或者整数，或者发送像对象这样的复杂数据来表示学生或大学或餐馆等信息，以及与该对象关联的所有数据

这些结果通常被封装到实现 IActionResult 接口的对象中，有大量的不同类型的结果实现了该接口，这些结果类型可以包含模型或文件的内容以供下载

这些不同的结果类型即可以返回 JSON，也可以返回 XML，或者 HTML 视图

动作基本上可以返回任意不同类型的动作结果。它们都有一个共同的基类：ActionResult

下表列出了不同种类的动作结果及其行为

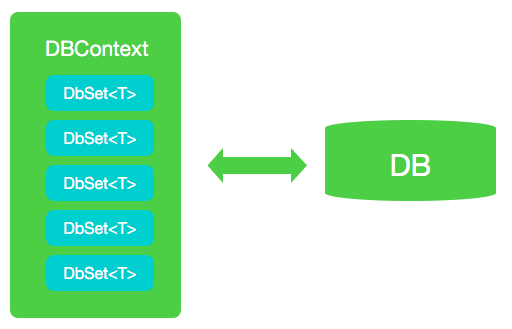


ASP.NET Core数据库上下文

## DBContext

EF 框架中，将数据视为对象并负责与之进行交互的主类是 DbContext

使用 DbContext 上下文的时候，推荐你定义一个从 DbContext 派生的类，并且定义一个公开的 DbSet 属性用于表示上下文中指定的实体集合



从逻辑上讲，DBContext 映射到具有 DBContext 可理解的表的特定数据库

在 DBContext 子类中，我们可以创建类型为 DbSet<T> 的属性。 泛型类型参数 T 将是一种类型的实体，如 Employee 是 HelloWorld 应用程序中的一个实体

ASP.NET Core设置和初始化数据库

## 初始化数据库

初始化数据库的方法之一是使用 EF 框架来创建数据库，仅仅需要两步就能完成

1. 第一步，给我们的 HelloWorld 项目添加迁移 ( migration ) 代码

迁移代码是 C# 代码，用来在数据库系统中创建数据库

当然了，EF 框架可以自动帮我们生成迁移代码

EF 框架通过对比数据库和我们的模型，并计算出所需的数据库表的更改

当我们添加其它的模型或对现有模型进行更改 ( 如 Employee 类 )时，我们可以使用 EF 框架创建新的迁移来保持我们的数据库表同步

1. 第二步，执行迁移代码，更新数据库，也就是说，我们需要明确地应用这些迁移来更新数据库

这两步任务都可以通过在 **控制台**窗口 ( 终端或 Power Shell 或 **命令提示符** ) 中使用一些简单的命令来实现

好吧，开始吧

1. 首先打开 **控制台**窗口 ( 终端或 Power Shell 或 **命令提示符** )，定位到我们的 HelloWorld 项目下

$ cd ~/Developer/aspnetcore/HelloWorld/HelloWorld

然后使用 ls 命名就可以列出当前目录下的所有文件和子目录

$ dir

AppSettings.json Controllers HelloWorld.csproj Models Program.cs Properties Startup.cs Views bin obj wwwroot

如果能看到 HelloWorld.csproj 则说明处于正确的目录，否则你应该继续 cd 到该目录下

然后可以运行下列命令创建迁移代码

dotnet ef migrations add InitialCreate -v

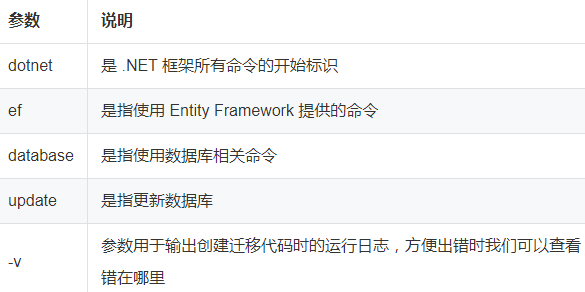


3、创建迁移成功后，我们就可以使用下面的命令查看当前有多少迁移代码和它们的状态

$ dotnet ef migrations list

4、接下来我们就要开始应用这些迁移来更新数据库，执行下面的命令来应用迁移代码

$ dotnet ef database update -v



## 布局视图 ( Layout View )

首先，我们来了解下 ASP.NET Core 中的布局视图到底是什么：

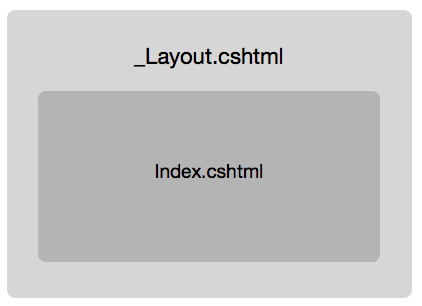
1. 布局视图是带有 .cshtml 扩展名的 Razor 视图

可以随意给布局视图命名，一般情况下，默认的约定的布局视图名字是 \_Layout.cshtml

这是布局视图的通用名称，可以不需要前导下划线。因为这只是许多开发者遵循的一个约定

用来区分布局视图与普通视图

1. 布局视图是一种特殊的视图，一旦我们有了布局视图，我们就可以设置我们的控制器视图，如 Home 的 Index 视图



我们可以将控制器视图设置为在布局视图内的特定位置显示

这种视图布局方法意味着 Index.cshtml 不需要知道有关 logo 或顶级导航的任何信息

Index.cshtml 视图只需要显示控制器操提供模型的特定内容，其它内容则由 布局视图来负责处理

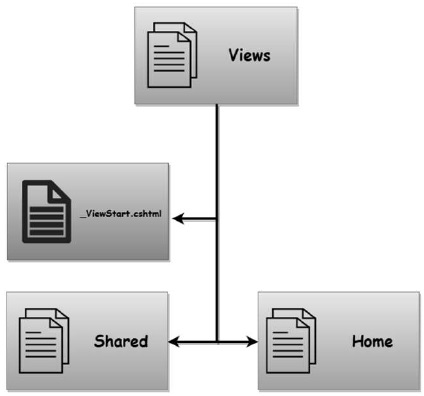
@RenderBody 表达式用于包含普通视图生成的内容

ViewBag 是一种数据结构，可以添加任何想要放入 ViewBag 的属性或数据。 例如可以在 ViewBag 上添加 ViewBag.Title，ViewBag.CurrentDate 或我们想要的任何属性

## 视图起始页 \_ViewStart.cshtml

MVC 中的 Razor 视图引擎有一个约定，就是在渲染单个控制器视图前，会优先查找名称为 \_ViewStart.cshtml 的文件，并执行该文件中的代码

于是，Razor 视图引擎的渲染流程就变成了下图这样



当然了，\_ViewStart.cshtml 也不是万能的，就是它不能包含任何 HTML 标记，但可用于从各个视图内的代码块中删除重复的代码

这也是遵守单一职责原则，输出 HTML 代码的工作都交给了布局视图和普通视图去搞定

## \_ViewStart.cshtml 的运行机制

1. 当 MVC 框架中的 Razor 开始渲染视图前，它会递归检查文件夹结构层次中是否存在 ViewStart 文件
2. 我们已将 \_ViewStart 直接放入了 Views 文件夹中。这将影响 Views 文件夹内的所有文件夹中的所有视图以及 Home 文件夹内的视图以及 Shared 文件夹以及将来可能添加的其它任何控制器文件夹
3. 如果我们使用 ViewStart 并将其 **仅** 放置在 Home 文件夹中，那么只有在 Home 文件夹中的视图被渲染时才会执行这一小段代码
4. 我们甚至可以拥有多个 ViewStart 文件，因此我们可以在 Views 文件夹中的 ViewStart.cshtml 中设置所有视图的布局视图
5. 如果我们想要改变 Home 文件夹中所有视图的默认设置，可以在 Home 文件夹中设置另一个 ViewStart，将布局设置为其它内容

**区别**

那么下面两个内容有区别吗？

Layout = "\_Layout";

和

Layout = "~/Views/Shared/\_Layout.cshtml";

答案是有，当然如果只有一个 \_Layout.cshtml 的情况下，就没有区别

1. 前者的搜索范围比后者更广，可以在 Views 目录也可以在 Views/Shared 目录
2. 前者的文件扩展名可以随意，但后者只能是 .cshtml

我们为了引入 HomePageViewModel ，竟然是用来全限定类名 HelloWorld.Controllers.HomePageViewModel

一种最简单的做法就是先引入 HelloWorld.Controllers 再使用 HomePageViewModel。 我们可以使用 Razor 提供了 @using 指令来引入命名空间

@using HelloWorld.Controllers

修改下我们的 Index.cshtml 为如下内容

@using HelloWorld.Controllers

@model HomePageViewModel

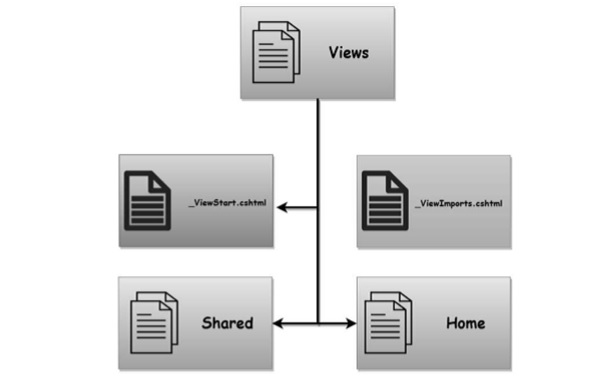
这样做的好处就是，如果要引入一个命名空间下的多个类，就不要全部都使用全限定类名了

## 视图导入 ( View Import )

视图导入是除了视图起始页之外，另外一种 Razor 在渲染任何其它视图前会查找的视图

视图导入的文件名为 \_ViewImports.cshtml

跟视图起始页一样，我们可以把视图导入 \_ViewImports.cshtml 放到一个文件夹中，那么视图导入中的代码则会影响当前文件夹和所有的递归子文件夹中的视图



在视图导入文件中 \_ViewImports.cshtml，我们可以编写一些 C# 代码，和放置常规的指令来引入视图所需要的命名空间

如果某个文件夹下的大部分视图都会使用到某个命名空间，那么我们可以在视图导入中使用一次 @using 导入即可，而不是在每个视图中使用 @using 指令分别导入或者输入全限定类名

## Razor 标签助手

标签助手允许服务器端代码 ( 也就是 C# 代码) 参与到 Razor 文件中创建和渲染 HTML 元素

简单的说，标签助手是一项新功能，类似于 HTML 助手，可以帮助我们渲染 [HTML](https://www.twle.cn/l/yufei/html/html-basic-index.html)

Razor 内置了许多标签助手用来处理常见的任务。例如创建表单，链接，加载资源等

标签助手是用 [C#](https://www.twle.cn/l/yufei/csharp/csharp-basic-index.html) 语言编写的，它们根据元素名称，属性名称或父标签创建 HTML 元素

例如，可以使用内置的 LabelTagHelper 并传递相应的属性来生成 HTML <label> 元素

如果你对 HTML 助手很熟悉，那么就可以轻松的使用标签助手来简化在 Razor 视图中生成 HTML

## 检查当前项目是否安装标签助手

ASP.NET Core MVC 中的 Razor 标签助手的命名空间是 Microsoft.AspNetCore.Mvc.TagHelpers 一般情况下系统会自动安装

## 安装 Razor 标签助手

如果你没有在 Microsoft.AspNetCore.App 包中找到 Microsoft.AspNetCore.Mvc.TagHelpers

## 关于标签助手

关于标签助手，我多说几句

1. 现在任何人都可以创作一个标签助手，所以如果你需要的标签助手不存在，你可以编写自己的标签助手
2. 你可以把自己创作的标签助手放在你的应用程序项目中，但需要把该标签助手关联到 Razor 视图引擎
3. 默认情况下，标签助手不会一起被渲染到客户端，即使这些标签助手看起来像融入了 HTML
4. Razor 会调用一些代码来处理标签助手，标签助手可以从 HTML 中删除自己，也可以添加额外的 HTML

**导入标签助手**

可以用标签助手做很多很棒的事情，但前提是需要向 Razor 注册你的标签助手，甚至微软官方的标签助手，以便 Razor 能够在标记中发现这些标签助手并且能够 调用处理标签助手的代码

注册标签助手的指令是 @addTagHelper， 我们可以把这个指令放置到单个视图中，或者如果您计划在整个应用程序中使用标签助手，则可以放在视图导入 \_ViewImports.cshtml

其实，\_ViewImports.cshtml 的默认代码就导入了所有官方提供的标签助手

@namespace HelloWorld.Views

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

可以看到 @addTagHelper 指令有两个参数

1. 第一个参数，星号 ( \* ) 用于导入程序集的所有的标签助手
2. 第二个参数，Microsoft.AspNet.Mvc.TagHelpers ，用于指定哪个程序集

第一个参数其实是一个类型名称，所以如果只想使用某个标签助手，那么我们可以列出这个名称，但如果想导入所有的标签助手，则可以使用星号 ( \* )

两个参数合起来的意思导入程序集 Microsoft.AspNet.Mvc.TagHelpers 中的所有标签助手

## 使用标签助手

因为我们已经在视图导入 \_ViewImports.cshtml 中导入了标签助手，所以我们可以在 Index.cshtml 视图中使用它们

我们可以使用 @Html.ActionLink URL 链接生成助手来创建自动关联到当前控制器的某个方法的 <a> 标签

@Html.ActionLink 标签有三个参数，分别是 <a> 标签的显示文本，当前控制器下的动作 ( action ) 和其它的附加参数组成的一个对象

如：@Html.ActionLink(employee.ID.ToString(), "Detail", new { id = employee.ID })

## HTML 助手

一般情况下，在视图中我们都更喜欢类 HTML 语法，也就是 [XML](https://www.twle.cn/l/yufei/xml/xml-basic-index.html) 语法

这样看起来更直观也更统一

而 Razor 也支持另外一种 HTML 助手，也就是 XML 语法的标签助手，这种助手直接在原来的 HTML 基础上添加一些 asp- 开头的属性来生成 HTML 标签

例如，对于 Index.cshtml 中的 @Html.ActionLink，我们还可以这么写

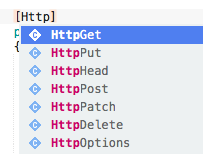
<a asp-action="Detail" asp-route-id="@employee.ID" >详情</a>

其中，asp-action="Detail" 是我们想要获得的动作的名称，如果想传递任何参数，可以使用 asp-route-\* 标签助手，在这里我们想包含 ID 作为参数，则可以使用 asp-route-id

<a asp-action="Detail" asp-route-Id="@employee.ID">详情</a>

ASP.NET Core 所有以 Http 开头的特性用于标识该方法只接受特定的 HTTP 请求方法

ASP.NET Core 支持下图七个 HTTP 请求方法特性





方法的返回值 IActionResult ， IActionResult 是所有返回结果的必须实现的接口，用这个作为返回值，也就是说该方法可以接受任意返回值。

RedirectTo 家族非常庞大，数量之多，几乎涵盖了各种情况下的重定向



**模型绑定** 是将 HTTP 请求的各种数据映射到动作方法参数的一种方法，参数即可以是简单类型，如字符串、整数或浮点数，也可以是复杂类型，如一个类。

**模型绑定** 查找数据源的顺序是

1. Form values: 通过 HTTP POST 方法提交的数据，例如 Name
2. Route values: 路由器解析的参数，例如 id
3. Query String: URI 中的查询字符串部分的数据，也就是 ? 后面那一串

而模型绑定按照上面的顺序，只要找到了数据，就会停止继续查找。假设我们我们的 URI 为 /home/edit/1 ，而我们的表单有一个字段 id 值为 2 ，那么 Edit 方法的值为 2 而不是一，这是因为已经在 Form 表单中找到了 id 值，所以就不会继续查找了

对于复杂的数据，比如 HomePageViewModel，它的属性也是有要求的

1. 要求接收数据的属性必须是 public 的，如果不是，那么就没法注入值
2. 要求必须有一个无参数的构造函数

当发生绑定时，类只会使用公共默认构造函数来实例化，然后在设置属性值。

### ModelState

当一个参数绑定了数据后，模型绑定就会停止查找该名称的值，并继续绑定下一个参数。

如果某个参数绑定失败，那么 MVC 框架也不会抛出任何错误，而是设置 ModelState 的属性为 false，仅此而已

因此，我们可以通过检查 ModelState 的值来检查数据格式是否正确

### \_context.SaveChanges();

\_context.SaveChanges(); 用于保存验证后的数据到数据库

Identity 框架 ，也就是所谓的身份验证框架

ASP.NET Core Identity 框架用于实现表单身份验证， 有许多验证方式可供选择，包括 Windows 身份验证和所有第三方身份验证，如 Google，Microsoft，Facebook 和 GitHub 等

Identity 框架还支持双因素身份验证，也就是所谓的两步验证，当然也支持第三方身份提供商和其它功能

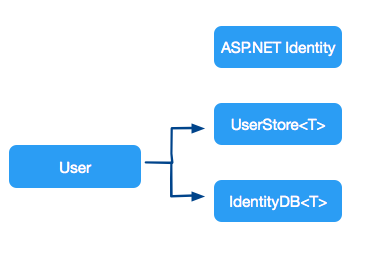
我们可以使用 Identity 框架 给我们的应用添加注册登陆功能，这样用户可以注册并使用本地密码登录

## 检查是否安装了 Identity 框架

ASP.NET Core 一般内置了 Identity 框架，我们可以展开依赖项 Microsoft.AspNetCore.App 包，一路查找名为 Microsoft.AspNetCore.Identity 的程序集

## Identity 框架注册与登录

为了使用 Identity 框架实现注册与登录，我们需要创建一个 User 实体，实体类将从 Identity 框架中的基类继承，该基类提供了标准用户属性，如用户名和电子邮件地址



1. 可以给 User 添加其它属性来保持我们想要的保存的用户信息
2. 然后，需要把这个 User 类加入到 Identity 框架提供的 UserStore 类中
3. 最后，使用 UserStore 与数据库进行交互， Identity 框架支持 EF 框架和可以与 EF 框架一起工作的所有数据库

当然了，我们也可以自定义 UserStore 来支持任何其它数据源

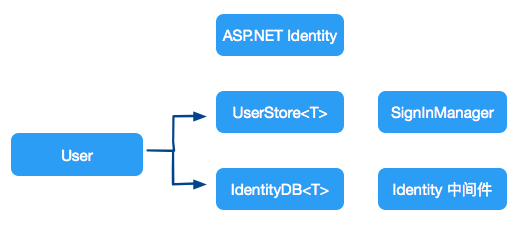
为了正确地使用 EF 框架，我们需要将 User 类插入到 IdentityDb 类，该类使用 EF 框架的 DBContext 完成实际数据库操作

当然了，使用 IdentityDb 类的方式是让现有的 DbContext 类从 IdentityDb 继承，而不是继承 EF 框架的 DBContext

IdentityDb 和 UserStore 类一起用于存储用户信息并验证用户密码

## Identity 框架最重要的两个组件

Identity 框架提供了很多组件，但我觉得最重要的莫过于两个：SignInManager 和 Identity 中间件



### 1. SignInManager ( 登录管理器 )

顾名思义，一旦密码验证通过，SignInManager 就允许用户登录

当然了，SignInManager 还可以用于登出一个用户

如果使用表单身份验证，那么登录和注销通过管理 cookie 来实现的。当我们告诉 SignInManager 允许某个用户登录时， SignInManager 会向用户的浏览器返回一个 cookie，浏览器接下来的每个后续请求中都会发送该 cookie，直到 cookie 过期，我们可以使用该 cookie 来识别用户

### 2. Identity 中间件

Identity 中间件读取 SignInManager发送的 cookie 并识别用户，一般情况下，Identity 中间件都是在排在所有其它中间件之后才运行的

要使用该中间件，我们需要将它配置到我们的应用程序管道中，才能处理 SignInManager 设置的 Cookie

验证用户身份的第一步就是要标识那些控制器哪些操作需要验证用户，而这一步操作可以使用 Identity 框架提供的 [Authorize] 特性来解决



## [Authorize] 与 [AllowAnonymous]

可以把 [Authorize] 特性放在控制器上，那么该控制器的所有方法都必须需要授权才能访问，也可以放在控制器内的动作方法上，那么只有该动作方法需要授权访问



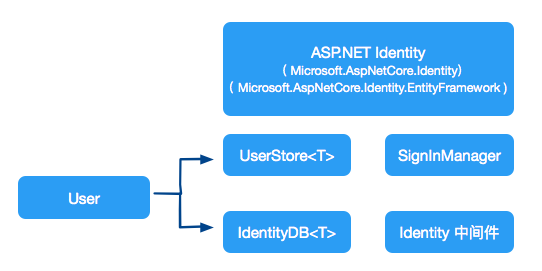
默认情况下，如果不传递其它参数给 Authorize，授权检查的唯一项目就是确认用户是否已经登录，我们也可以使用参数来指定任何自定义授权策略

除了 [Authorize] 外，还有一个特性 [AllowAnonymous]。当想要在使用了 [Authorize] 特性的某个控制器的动作方法上取消保护并允许匿名用户访问时，可以使用该特性

## 配置 Identity 事项

首先，我们要安装两个依赖项目 Microsoft.AspNetCore.Identity 和 Microsoft.AspNetCore.Identity.EntityFramework，前者一般情况下都内置了，后者会随着安装 Entity Frawework 一起安装

对于我们的应用程序，我们将继承 Identity 框架提供的基类，使用该类提供所有基本要素，如用户名属性和存储散列密码



我们也会修改 DBContext 类继承自 Identity 框架的 IdentityDb 类而非 Microsoft.EntityFrameworkCore 命名空间下的 DbContext。IdentityDb 类提供了 EF 框架存储用户信息所需的所有东西

一旦我们设置好了 User 类和 DBContext，接着就需要在 Startup 类的 ConfigureServices 方法中添加 Identity 服务，就像使用 MVC 时需要添加服务一样，使用 Identity 时也需要添加服务

我们需要注册的 Identity 服务有两项：UserStore 服务和 SignInManager 服务，这些服务会在需要的时候注入我们的控制器，以便在需要的时候创建用户和发送 cookies

最后，还需要在 Startup 类中的 Configure 方法中添加中间件 Identity，这个中间件可以读取 cookie 信息并创建用户身份，而且可以保证未授权时返回的是一个普通的页面和不适 401 空白页面

## 配置 Identity

刚刚，我们已经把配置 Identity 需要做的事情了解清楚了，接下来我们就一步一步来实现吧

1. 安装依赖 Microsoft.AspNetCore.Identity 和 Microsoft.AspNetCore.Identity.EntityFramework，这个就不多讲了，忽略
2. 在 Models 目录下添加一个用户类 User，添加过程也不详细介绍了，添加完成后默认的代码如下
3. 修改 User.cs 引入命名空间 Microsoft.AspNetCore.Identity，并修改 User 类继承自 IdentityUser

ASP.NET Core Identity迁移数据

1. 首先停止我们的应用程序，不然会报错，因为我们使用的是文件系统数据库
2. 其次打开 **控制台**窗口 ( 终端或 Power Shell 或 **命令提示符** )，定位到我们的 HelloWorld 项目下

$ cd ~/Developer/aspnetcore/HelloWorld/HelloWorld

然后使用 dir 命名就可以列出当前目录下的所有文件和子目录.

如果能看到 HelloWorld.csproj 则说明处于正确的目录，否则你应该继续 cd 到该目录下

3.然后可以运行下列命令创建迁移代码

dotnet ef migrations add InitialIdentity -v



4.创建迁移成功后，我们就可以使用下面的命令查看当前有多少迁移代码和它们的状态

$ dotnet ef migrations list

5、接下来我们就要开始应用这些迁移来更新数据库，执行下面的命令来应用迁移代码

$ dotnet ef database update -v

