序章 ASP.NET Core MVC

# 概述

MVC框架将一个应用分为三部分：模型、视图、控制器。这种模式有助于实现关注分离。

用户请求被路由到控制器，控制器负责与模型协作，以执行操作或\和返回请求结果。控制器选择视图显示给用户，并为其提供所需要的任何模型数据。

# 路由

ASP.NET Core MVC路由是建立在ASP.NET Core初级路由上的、一项强大的URL映射组件。

# 控制器

MVC中控制器是最初的入口，负责选择同哪一个模型类型协作，以及哪一类视图来呈现。控制器是承载用户交互。模型运转，并最终选择视图进行渲染的组件。

# 视图

视图负责在用户界面呈现内容。他们呢使用Razor视图引擎在HTML标记中嵌入C#代码。视图应仅包含少量逻辑，并且这些逻辑应该与呈现内容相关。

# 模型

模型代表了应用的状态和业务逻辑或其可以展现的一些操作。强类型视图一般使用特别设计的视图模型（ViewModel）类型，它包含了视图显示所需要的数据；控制器将创建并根据模型填充这些视图模型。

第二章 路由(Routing)

# 概述

Creedon: URL匹配其实是把一串URL转化为路由值(RouteValues)的过程。

分为常规路由(Conventional Routing)和特性路由(Attribute Routing)。

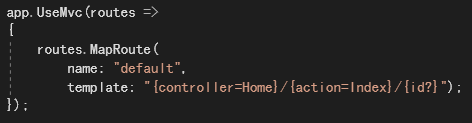
# 常规路由

常规路由实例：

routes.MapRoute(“default”, “{controller=Home}/{action=Index}/{id?}”)

第一段映射控制器，第二段映射操作，第三段映射模型实体或者其他（可选）。

**设置路由中间件：**



**使用简便方法app.UseMvcWithDefaultRoute()可以替换掉以上。**

{controller = Home}将Home定义为默认控制器。

{action = Index}将Index定义为默认操作。

{id?}将id设置为可选。

**路由名称：**

**以上的“default”是一个路由名称。路由名称对URL匹配和请求处理没有影响，他们仅用于URL生成。应用程序范围内路由名称必须唯一。**

**多路由：**

**路由可以不止一个，这些路由是有序的，将会从上到下被尝试。如果路由不匹配，将会尝试下个路由，这个过程称为“回退”。**

**动作方法歧义：**

**有两个一致的动作方法对应路由时，会根据动作方法参数和约束特性(例如[HttpPost]、[HttpGet]等)选择最佳操作。**

# 特性路由

特性路由需要更多的输入来指定一个路由，而上轨路由处理路由时则更加简洁。然而，特性路由允许精确每个操作的路由模板。使用特性路由时，控制器名，动作方法名不会影响路由匹配。

示例：







特性路由也可以使用[HTTPVerb]特性：





**路由组合：**

控制器上的路由特性可以和操作上的路由特性进行组合。控制器撒谎给你的路由模板会作为动作方法路由模板的前缀。

**特性路由的顺序：**

特性路由会构建一个树形结构，同时匹配所有路由。最具体的路由会在一般的路由之前执行。

**标记(占位)替换：**

特性路由支持占位标记替换。[action]、[controller]、[area]会在构建特性路由的最后一步被替换成对应的动作方法名和控制器名和区域名。

# URL生成

**通过动作方法名：**

Url.Action(“动作方法名”)

**通过路由名：**

Url.RouteUrl(“路由名称”)

# 区域路由

**注意：**

1、区域路由应该放在路由表的前面。

2、需要给控制器加上[Area(“…”)]特性，表示控制器属于某个区域的一部分。

3、不同于传统的ASP.NET MVC，现在Core可以用”{area}”占位标识了。

**添加区域路由方式一：**



(当匹配URL”Areaone/Home/Index”时实际产生的路由值是{area=Area1, controller=Home, action=Index })

**添加区域路由方式二：**

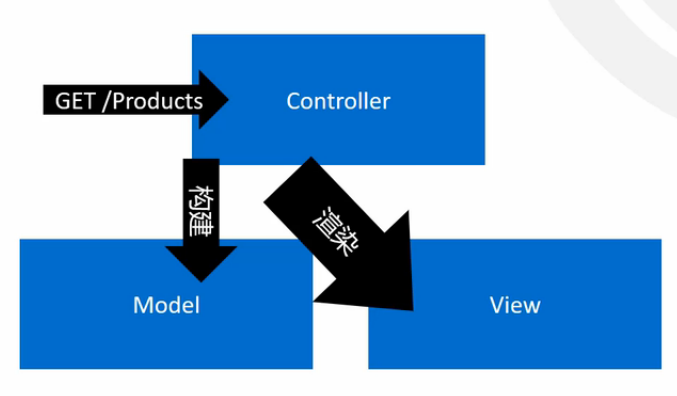


(exists后缀在ASP.NET Core MVC路由中有验证文件夹和文件是否存在的功能，只有当路由参数值对应的文件夹和文件在项目中确实存在时，路由匹配才算成功，否者路由匹配失败，会继续用Url去匹配 ASP.NET Core MVC路由表中的后续路由。)

第三章 控制器(Controller)

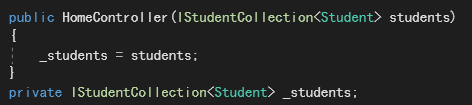
# 概述

控制器接受请求，构建Model(包含一定逻辑)，并渲染页面View(绑定Model)。控制器通常(不是必须的)位于Controllers文件夹类、继承自Controller基类且以”Controller”结尾。

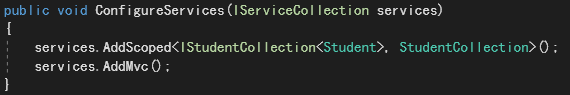


# 构造函数注入相关服务依赖

通过构造函数注入服务依赖：



在ConfigureServices方法中向IoC容器注册服务依赖：



# 控制器与动作方法可用的特性：

**HttpVerb特性：**

[HttpGet]只处理Get请求。

[HttpPost]只处理Post请求。

**路由特性：**

[Route(“…”)]特性路由

**过滤器特性**：

………………

# Controller成员(返回值)

**返回视图:**  this.View() ViewResult类型

this.View(“…”) （重载：返回名为…的视图）

this.View(model) （重载：强类型传值，把model传给视图）

**返回重定向:**  this.Redirect() RedirtResult类型

this.RedirctPermanent() RedirtResult类型

this.RedirectToAction() RedirtToActionResult类型

**返回状态:** this.Ok() StatusCodeResult类型（OkResult）

this.NotFound() StatusCodeResult类型（NotFoundResult）

**返回文本:** this.Content() ContentResult类型（比直接返回string更好）

**返回JSON:** this.Json() JsonResult类型

**返回文件:** this.File() FileContentResult类型

**※直接返回对象：** ObjectResult类型（一般默认会被序列化成Json）

# 控制器传值给视图

**ViewBag:** ViewBag.key = …; （无需类型转换）

**ViewData:** ViewData[“key”] = …; （用法与Session一样，需要类型转换）

**TempData:** TempData[“key”] = …; （可以跨方法传值）

**强类型传值**：return View(model); （传模型给视图）

# 控制器获取请求中的值

**通过动作方法的参数：(模型绑定)**

可以是简单的string、int等类型，关键在于参数名与表单元素的name属性名保持一致。也可以是复杂的模型对象，系统会自动赋值。还可以用来获取请求中的路由值。

public IActionResult Action(User user)

public IActionResult Action(string username, string pwd)

public IActionResult Action(int id)

**通过Request直接取值：**

与name属性名保持一致。

注意：与传统ASP.NET有些不同，Core中应该改用：

string username = Request.Query[“username”]; （GET）

string username = Request.Form[“username”]; （POST）

**（过期）FormCollection获取表单值在Core中已经不可用。**

# ※补充

**动作方法间传值**：

return RedirectToAction(string actionName, object routeValues);

常用非匿名的routeValues传值。

return Redirect(string url);

用查询字符串传值。

第四章 视图(View)

# Razor视图引擎

**Razor引擎：**

服务器首先分析请求的视图，然后运行视图中的服务端代码，最后动态生成可被客户端浏览器解析的HTML代码。

**Razor语法：**

@服务端代码、@(服务端代码)、@{服务端代码}

# ****HtmlHelper(Html帮助器):****

**普通帮助器示例：**

**Html.BeginForm()构建表单**

**Html.Label()构建标签**

**Html.TextBox()构建文本框**

**Html.Password()构建密码框**

**Html.DropDownList()构建下拉列表**

**Html.ActionLink()构建超链接**

**Html.ValidationSummary()构建错误信息汇总**

**Html.Raw()将Html内容自然呈现。**

**……**

**强类型帮助器示例：*(强类型帮助器会自动指定元素name等属性的值)***

**Html.TextBoxFor(lambda)**

**Html.LabelFor(lambda)**

**……**

**参数注释：**

**RouteValues：通常是一个包含路由值的匿名对象。new {…}**

(可以指定area、acton、controller、id这些)

**HtmlAttribute: 通常是一个包含html元素特性的匿名对象。new{…}**

(注意：value要大写为Value,class要加”@”)

# TagHelper (新增功能)

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

(在ViewImports中添加以上语句便可使用TagHelper)







（元素添加asp-属性后激活TagHelper）

# 强类型视图和弱类型视图

**强类型视图：**

强类型视图使用模型（Model）来呈现数据的视图。强类型视图可以通过”@model”指定模型，然后用@Model访问模型数据。

**弱类型视图：**

弱类型视图又称为动态类型视图。是通过ViewData、ViewBag等传递数据的视图。

# 补充：布局页(Layout)

**概述：**

布局页类似WebForm母版页，一般放在Shared文件夹。

**\_ViewStart.cshtml**

文件\_ViewStart.cshtml是启动文件，它的代码会在所有.cshtml被执行之前执行，主要用于一些放一些重复性代码和不适合在布局页中进行的操作，一般用它指定默认布局页。

**\_ViewImports.cshtml**

文件\_ViewImports.cshtml一般用于引入命名空间和启用TagHelper。

**@RenderBody()**

用于在布局页中某个位置定义视图页的占位。

**@RenderSection()**

在布局页某个位置定义section的占位。



在内容页定义section。



# 补充：分部视图

**概述：**

把某个.cshtml文件作为当前视图的一部分插入某个位置，称为分部视图。分部视图可以解决View代码复用。分部视图还可以嵌套。

**@Html.Partical(“…”)**

插入分部视图。其重载Html.Partial(“…”, model)还能把模型传给分部视图。

(注意：现在ASP.NET Core现在已经不支持Html.Action()方法了。但是新增；了ViewComponent功能)

**<partial>TagHelper**

除了Html.Partial帮助器方法外，还能使用TagHelper插入分部视图。

# 补充：视图组件（ViewComponent）

**概述：**

ViewComponent相当于一个小的MVC结构。

**添加xxxViewComponent类：**

相当于MVC控制器，继承自ViewComponent类，用Invoke方法返回一个视图。放在ViewComponents文件夹下。

**添加要返回的视图：**

一般放在Views/Shared/Components/xxx/目录下。名称是Invoke方法返回视图指定的名称，或者直接命名为Default.cshtml。

**视图中调用ViewComponent：**

@await Component.InvokeAsync(“xxx”);

第五章 模型(Model)

# 概述

模型分为默认模型和自定义模型。默认模型是指.NET提供的类模型，无需自定义类。自定义模型可以是EF的实体类，可以使用字段特性声明(数据注解)。

模型可以绑定视图来呈现数据，称为强类型视图。

# ※模型绑定 (Model Binding)

**模型绑定介绍：**

模型绑定其实是把HTTP请求中的值映射到动作方法参数的过程。

**参数可以是简单类型int、string等。也可以是复杂类型。**

**模型绑定查找数据源顺序：**

Form Values：通过POST发送的表单数据。

Route Values：路由数据。

QueryStrings：URL查询字符串的一部分。

**通过特性自定义模型绑定行为：**

[BindRequire]:如果不发生绑定则报错。

[BindNever]：从不绑定到此参数。

[From…]指定绑定源……

# 模型数据传递给视图

**1、在控制器实例化模型并传递给视图。**

如果有多行模型数据，可以用用一个泛型列表。

**2、在视图中用”@model”指定模型。**

如果有多行模型数据，则绑定一个List<>类型。

**3、绑定后就可以在视图中用”Model”调用模型数据了。**

如果绑定的是泛型列表List<>，可以用foreach语句遍历Model。

# 模型验证和ModelState

**验证机制：**

Model可以用来实现服务端验证，用模型对象作为动作方法参数来获取表单值，并用ModelState.IsValid判断是否验证通过。通常使用强类型的Html帮助器（如Html.TextBoxFor(lambda)、Html.LabelFor(lambda)）来构建用户输入表单。**ModelState:**

（是一个ModelStateDictionary类型，有以下成员：）

ModelState[] **键名索引**

ModelState.IsValid 验证是否通过

ModelState.Keys 所有字段的名称集合

ModelState.ErrorCount 错误数

ModelState.AddModelError(string key, string errorMessage) 添加错误信息

# 数据注解(Data Annotation)

**常用于验证的注解特性：**

[Display(Name = “…”)]该属性在页面显示的提示信息，常用于LabelFor。

[Require]必填字段。

[StringLength]字符长度。

[Range]用户输入数的范围。

……………

**自定义验证：**

继承ValidationAttribute，并且重写IsValid方法即可。