# 视图

## 创建自定义视图引擎

一般情况下直接使用MVC框架自带的内建视图引擎即可，但如果想知道视图引擎是如何工作的，就需要从建立一个自定义视图引擎开始了。通过之前的学习我们都知道了内建视图引擎包括Razor和ASPX两种，ASPX是针对旧版本MVC程序的，他主要是维护旧版本MVC应用程序，保持系统的兼容性而保留的Web Form视图引擎；Razor是在MVC3引入的，它的语法更加简洁。

现在我们就先从自定义视图引擎开始，了解一下视图引擎的工作机制。视图引擎的接口是IViewEngine，其结构如下：

命名空间：System.Web.Mvc

方法：

1. FindPartialView

参数[类型]：

* controllerContext[ControllerContext]
* partialViewName[string]
* useCache[bool]

返回值：ViewEngineResult

1. FindView

参数[类型]：

* controllerContext[ControllerContext]
* viewName[string]
* masterName[string]
* useCache[bool]

返回值：ViewEngineResult

1. ReleaseView

参数[类型]：

* controllerContext[ControllerContext]
* view[IView]

返回值：ViewEngineResult

前两个方法（FindPartialView、FindView）接收的参数是描述请求的：处理该请求的控制器、视图名及布局。当框架对ViewResult进行处理时，会调用这两个方法。最后一个方法（ReleaseView）在视图不再需要时被调用，其功能就是要释放视图所占用的资源。

注：MVC框架对视图引擎的支持是由ControllerActionInvoker（控制器动作调用器）类实现的，这是IActionInvoker接口的内建实现。如果已经直接通过IActionInvoker或IControllerFactory接口实现了自己的动作调用器或控制器工厂，将无法自动地访问视图引擎特性。

当请求一个视图时，ViewEngineResult类使试图引擎能够对MVC框架作出响应。当视图引擎能够对请求提供视图时，将通过如下构造函数创建一个ViewEngineResult：

public ViewEngineResult(IView view, IViewEngine viewEngine)

当视图不能对请求提供视图时，则使用如下构造函数：

public ViewEngineResult(IEnumerable<string> searchedLocations)

该重载版本的构造函数是通过参数的视图位置的集合进行枚举查找并创建ViewEngineResult的，如果找不到视图，则该枚举的信息会显示给用户。

视图引擎系统的最后一个构造块是IVew接口：

namespace System.Web.Mvc

{

public interface IView

{

void Render(ViewContext viewContext, TextWriter writer);

}

}

该接口中定义的Render方法的ViewContext类型参数传递了客户端请求的信息，以及动作方法的输出。TextWriter类型参数则用于将输出写给客户端。

在了解了视图引擎的构造组成后，就来创建一个简单的视图引擎做一下深入的研究，我们的视图引擎简单到何种地步呢，我们只让其返回一个视图，该视图将渲染关于请求的信息，以及动作方法产生的视图数据。这样一来既能演示视图引擎的操作方式，也不会陷入解析视图模板的困境。

### 创建示例项目

项目模板：Empty

项目名称：Views

控制器：Home

Home控制器代码如下：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Views.Controllers

{

public class HomeController : Controller

{

//

// GET: /Home/

public ActionResult Index()

{

ViewData["Message"] = "Hello, World";

ViewData["Time"] = DateTime.Now.ToShortTimeString();

return View("DebugData");

}

public ActionResult List()

{

return View();

}

}

}

### 实现自定义的IView

自定义IView实现类：DebugDataView

位置：Infrastructure

代码清单：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.IO;

namespace Views.Infrastructure

{

public class DebugDataView : IView

{

public void Render(ViewContext viewContext, System.IO.TextWriter writer)

{

Write(writer, "---Routing Data（路由数据）---");

foreach (string key in viewContext.RouteData.Values.Keys)

{

Write(writer, "key: {0},value: {1}", key, viewContext.RouteData.Values[key]);

}

Write(writer, "---View Data（视图数据）---");

foreach (string key in viewContext.ViewData.Keys)

{

Write(writer, "key: {0},value: {1}", key, viewContext.ViewData[key]);

}

}

private void Write(TextWriter writer, string template, params object[] values)

{

writer.Write(string.Format(template, values) + "<p/>");

}

}

}

该演示代码中演示了Render方法的两个参数的用法：取得ViewContext，并用TextWriter向客户端写出响应。在后面的自定义视图引擎的实现中，我们慢慢地会明白该类的功能。

### 实现自定义的IViewEngine

一定要明白视图引擎的目的是产生一个ViewEngineResult对象，它或者包含一个IView，或是一个用于搜索适当视图的位置列表。

自定义IView实现类：DebugDataViewEngine

位置：Infrastructure

代码清单：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Views.Infrastructure

{

public class DebugDataViewEngine : IViewEngine

{

public ViewEngineResult FindPartialView(ControllerContext controllerContext, string partialViewName, bool useCache)

{

return new ViewEngineResult(new string[] { "No View (Debug Data View Engine)" });

}

public ViewEngineResult FindView(ControllerContext controllerContext, string viewName, string masterName, bool useCache)

{

if (viewName == "DebugData")

{

return new ViewEngineResult(new DebugDataView(), this);

}

else

{

return new ViewEngineResult(new string[] { "No View (Debug Data View Engine)" });

}

}

public void ReleaseView(ControllerContext controllerContext, IView view)

{

// do nothing...

}

}

}

本示例仅实现了针对单一的视图DebugData的支持，如果实现的是更严格的视图引擎，可以进行模板的搜索、考虑布局和提供缓存设置。

IviewEngine接口假设视图引擎有它需要查找的地方。但这里不需要查找任何地方，因此只返回一个哑元位置（Dummy Location），以表明不能交付视图。

该自定义视图还不支持分部视图，因此，通过FindPartialView方法返回一个结果，以表明其不能提供视图。

由于这里没有需要释放的资源，我们也就没有实现ReleaseView方法。

### 注册自定义视图引擎

视图引擎需要在Global.asax的Application\_Start方法中注册，如：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Routing;

using Views.Infrastructure;

namespace Views

{

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

**ViewEngines.Engines.Add(new DebugDataViewEngine());**

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

}

}

}

静态的ViewEngines.Engines集合中包含一组程序中按照的视图引擎。MVC框架也支持在一个程序中存在多个引擎。当处理一个ViewResult时，动作调用器获取这组已安装的视图引擎，并依次调用它们的FindView方法。

一旦动作调用器接收到一个含有IView的ViewEngineResult对象，便会停止调用FindView方法。如果有两个或多个引擎能够对同视图名的请求进行服务，这意味着在ViewEngines.Engines集合中添加引擎的顺序是重要的。如果希望引擎取得优先，可以将它插入在该集合的开始部分，如：

ViewEngines.Engines.Insert(0,new DebugDataViewEngine());

### 测试自定义视图引擎

此时启动程序，便可测试这个视图引擎了。效果如图：



这是Home控制器的Index方法通过View方法返回了指向DebugData视图的ViewResult产生的结果。但如果导航到：/Home/List，由于该动作方法返回了一个不受支持的默认视图，将会得到如下结果：



从上图红框的位置可以看出，消息是作为一条搜索视图的位置来报告的。注意Razor和ASPX视图也出现在列表中，这是因为这些视图引擎仍然起作用。如果只希望使用自定义的视图引擎，则必须在Global.asax的Application\_Start方法中将其清除，具体做法如下：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Routing;

using Views.Infrastructure;

namespace Views

{

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

**ViewEngines.Engines.Clear();**

ViewEngines.Engines.Add(new DebugDataViewEngine());

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

}

}

}

现在，重新导航到/Home/list将会看到下图的效果：



## 使用Razor引擎

前面实现的自定义视图仅仅是生成了一个十分简陋的视图，而且对于视图引擎的复杂性方面的实现一点都没有做，但是，这已经足够让我们明白视图引擎的工作机制了。

视图引擎的复杂度真正来源于视图模板系统，包括：代码片段、支持布局，以及为优化性能而对模板进行的编译等。

Razor几乎可以满足所有的MVC应用程序，只有十分罕见的项目需要创建自定义视图。

### 示例项目

对于后面想演示，需要再创建一个新的示例项目，使用的模板是Basic模板，项目名称为WorkingWithRazor，并创建一个Home控制器：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace WorkingWithRazor.Controllers

{

public class HomeController : Controller

{

public ActionResult Index()

{

string[] names = { "Apple", "Orange", "Pear" };

return View(names);

}

}

}

该Home控制器的Index动作方法对应的Index视图如下：

@model string[]

@{

ViewBag.Title = "Index";

}

This is a list of fruit names:

@foreach (string name in Model)

{

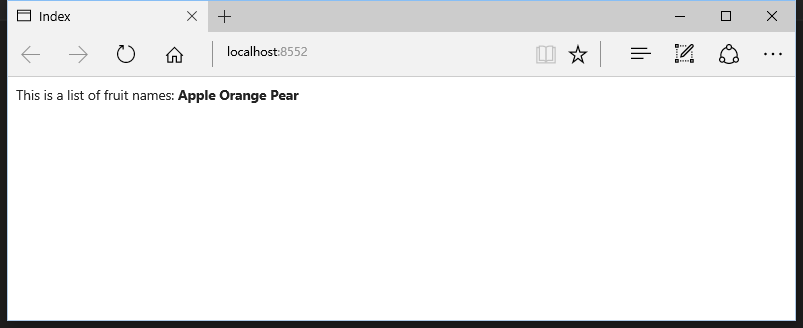
<span><b>@name</b></span>

}

### Razor视图的渲染

Razor视图引擎会将视图转换成C#类，然后将其进行编译。这样做的目的其一就是为了改善性能，同时这也是在视图中能够如此方便地包含C#代码片段的原因。

在程序启动之前，MVC中的视图不会被编译。因此，要查看Razor创建的类，需要启动程序，并导航到/Home/Index动作。发送给MVC程序的最初请求会触发所有视图的编译过程。下图中可以看出该请求的输出：



出于方便，会将视图文件生成的类写成磁盘上的C#代码文件，然后进行编译，也就是说我们可以在本机磁盘中找到这个文件，但是要想找到这个文件还是很不容易的——因为，需要通常为隐藏的文件夹，而且这些.cs文件名与它们所包含的类名不对应。但对于WIN10系统的存放位置一般是在：C:\Users\Administrator（这是我的机器登录用户名）\AppData\Local\Temp\Temporary ASP.NET Files目录下。对于该示例在本人机器中的路径为：root\0e20f253\3ae26c3f，其对应的视图为（为了方便阅读做了些整理）：

namespace ASP

{

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Net;

using System.Web;

using System.Web.Helpers;

using System.Web.Security;

using System.Web.UI;

using System.Web.WebPages;

using System.Web.Mvc;

using System.Web.Mvc.Ajax;

using System.Web.Mvc.Html;

using System.Web.Optimization;

using System.Web.Routing;

public class \_Page\_Views\_Home\_Index\_cshtml : System.Web.Mvc.WebViewPage<string[]>

{

public \_Page\_Views\_Home\_Index\_cshtml()

{

}

protected ASP.global\_asax ApplicationInstance

{

get

{

return ((ASP.global\_asax)(Context.ApplicationInstance));

}

}

public override void Execute()

{

ViewBag.Title = "Index";

WriteLiteral("\r\n\r\nThis is a list of fruit names:\r\n\r\n");

foreach (string name in Model)

{

WriteLiteral(" <span><b>");

Write(name);

WriteLiteral("</b></span>\r\n");

}

}

}

}

其实对于我在查找的时候，有一个让我很欣慰的是当我打开文件时，对于视图对应编译后的类文件都有原文件路径的指示，类似于这样：

#pragma checksum "**E:\XXX（你的项目路径）\WorkingWithRazor\Views\Home\Index.cshtml"** "{ff1816ec-aa5e-4d10-87f7-6f4963833460}" "0709F63862595E77163CDD7CA8667BF6CAC0664A"

从上面可以看出，这个类派生于WebViewPage<T>（这里的T为：string[]），而且从类名也能看出视图文件的路径已经被编译到类名之中（\_Page\_Views\_Home\_Index\_cshtml）。

在Execute方法中可以看出视图的语句和元素的处理时这样的：

* 以@符号为前缀的代码片段被直接表示成了C#语句。如：

@{

ViewBag.Title = "Index";

}

被转为了：

ViewBag.Title = "Index";

* HTML元素则以WriteLiteral方法处理，它将参数的内容写成了这些元素所给出的结果。这与Write方法相反，WriteLiteral方法用于C#变量并对字符串值进行编码，以使它们能够安全地用于HTML页面。

如：

This is a list of fruit names:

被转换成了：

WriteLiteral("\r\n\r\nThis is a list of fruit names:\r\n\r\n");

Write方法和WriteLiteral方法都是将内容写到一个TextWriter对象（这是传递给IView.Render方法的同一个对象）。编译Razor视图的目的是生成静态和动态内容，并通过TextWriter将内容发送给客户端。

### 配置视图搜索位置

Razor视图引擎在查找视图时遵循的是MVC框架早期版本建立约定。如Home控制器中的Index动作方法的视图，将会查找~/Views/Home/和~/Views/Shared/路径下的.cshtml（一个含有C#语句的模板）和.vbhtml（一个含有Visual Basic语句的模板）文件。Razor实际上不会在磁盘上查找这些视图文件（因为它们还没有被编译成C#类），而是查找表示这些视图的编译类。

通过实现RazorViewEngine类的子类可以改变Razor搜索的视图文件。该类是IViewEngine的Razor实现。它建立于一系列基类之上，这些类定义了一组用来确定搜索视图文件的属性，具体如下：

* 顶层视图的查找（相对于区域层的视图，位于项目的Views文件夹下）：
  1. 属性：
     + ViewLocationFormats
     + MasterLocationFormats
     + PartialViewLocationFormats
  2. 描述：查找视图、分部视图以及布局的位置
  3. 默认值：
     + ~/Views/{1}/{0}.cshtml
     + ~/Views/{1}/{0}.vbhtml
     + ~/Views/Shared/{0}.cshtml
     + ~/Views/Shared/{0}.vbhtml
* 区域层的视图查找：
  1. 属性：
     + AreaViewLocationFormats
     + AreaMasterLocationFormats
     + AreaPartialViewLocationFormats
  2. 描述：为一个区域查找视图、分部视图以及布局的位置
  3. 默认值：
     + ~/Areas/{2}/Views/{1}/{0}.cshtml
     + ~/Areas/{2}/Views/{1}/{0}.vbhtml
     + ~/Areas/{2}/Views/Shared/{0}.cshtml
     + ~/Areas/{2}/Views/Shared/{0}.vbhtml

这些属性早在Razor之前就存在了，其默认值中的占位符对应的参数值如下：

* {0}表示视图名；
* {1}表示控制器名；
* {2}表示区域名。

现在知道这些后就应该能猜到要想改变搜索位置，其实就是实现一个RazorViewEngine的子类，并修改上述属性的一个或多个属性值即可。下面在示例项目中添加一个Infrastructure文件夹，并添加一个视图引擎CustomLocationViewEngine来看看具体的操作，如：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace WorkingWithRazor.Infrastructure

{

public class CustomLocationViewEngine : RazorViewEngine

{

public CustomLocationViewEngine()

{

ViewLocationFormats = new string[]

{

"~/Views/{1}/{0}.cshtml"

,"~/Views/Common/{0}.cshtml"

};

}

}

}

现在需要做的是在Global.asax的Application\_Start方法中进行注册即可：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Optimization;

using System.Web.Routing;

using WorkingWithRazor.Infrastructure;

namespace WorkingWithRazor

{

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

**ViewEngines.Engines.Clear();**

**ViewEngines.Engines.Add(new CustomLocationViewEngine());**

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

}

}

如果想要仅使用自定义视图引擎，则先需要使用Clear方法将可能已被注册的视图引擎清除，然后使用Add方法添加自定义的实现。

为了能够使自定义视图引擎能够正常工作，需要在Views文件夹中创建一个Common文件夹，并在其中实现一个List.cshtml视图文件，如：

@{

ViewBag.Title = "List";

}

<h3>This is the /Views/Common/List.cshtml</h3>

然后继续在Home控制器中添加一个List动作方法来显示该视图：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace WorkingWithRazor.Controllers

{

public class HomeController : Controller

{

public ActionResult Index()

{

string[] names = { "Apple", "Orange", "Pear" };

return View(names);

}

**public ActionResult List()**

**{**

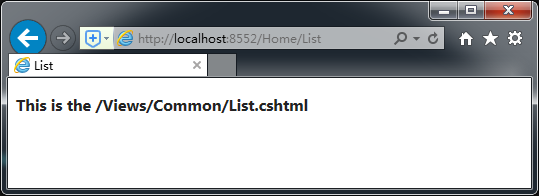
**return View();**

**}**

}

}

现在启动程序并导航至/Home/List时，将会使用自定义的位置查找/Views/Common文件夹中的List.cshtml视图文件，如图：



## 对Razor视图添加动态内容

所谓动态内容就是在运行时生成，并且随每一个请求而不同。这与静态内容恰好相反，在编写应用程序时，它的内容就已经生成了，且对每一次请求其内容都是一样的。添加动态内容的方式有多种，如下：

* 内联代码

用于小型的、自包含视图逻辑片段，如if和foreach语句。这是在视图中创建动态内容的基本手段，也是一些其他办法的基础。

* HTML辅助器方法

用于生成一个独立的HTML元素或小片元素集合，典型地，是基于视图模型或视图数据的值。MVC包含了许多有用的HTML辅助器方法，而且创建自己的辅助器方法也很容易。

* 分段

用于创建内容分段，这种分段用于插入到布局特定位置。

* 分部视图

用于在视图之间共享的子片段标记。分部视图也可以含有内联代码、HTML辅助器方法，以及引用其他分部视图。分部视图不调用动作方法，因此它们不能用来执行事务逻辑。

* 子动作

用于创建可重用的UI控件，或需要含有事务逻辑的小部件。当使用子动作时，它调用一个动作方法，返回一个视图，并把结果注入到响应流中。

### 使用分段

分段（Section）是用来在布局中提供内容区域的，它能灵活地控制将视图的哪一部分插入到布局中，以及将它们插入何处。请看下面一个示例：

在视图中定义分段：

@model string[]

@{

ViewBag.Title = "Index";

}

**@section Header{**

**<div class="view">**

**@foreach (string str in new[] { "Home", "List", "Edit" })**

**{**

**@Html.ActionLink(str, str, null, new { style = "margin:5px" })**

**}**

**</div>**

**}**

<div class="view">

This is a list of fruit names:

@foreach (string name in Model)

{

<span><b>@name</b></span>

}

</div>

**@section Footer{**

**<div class="view">**

**This is the footer**

**</div>**

**}**

这个示例修改了Index视图，其采用的定义格式为：@section <分段名称>（如示例中的“Header”和“Footer”分段）。分段的内容可以混用HTML标记和Razor标签。

还可以通过@RenderSection辅助器方法指定分段要插入的位置。如对\_Layout布局的修改：

提示：此时使用的仍是自定义视图引擎，尽管共享视图放在了/Views/Common文件夹，但共享布局仍位于/Views/Shared文件夹。

在布局中使用分段：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width" />

<style type="text/css">

div.layout {

background-color: lightgray;

}

div.view {

border: thin solid black;

margin: 10px 0;

}

</style>

<title>@ViewBag.Title</title>

</head>

<body>

**@RenderSection("Header")**

<div class="layout">

This is part of the layout

</div>

**@RenderBody()**

<div class="layout">

This is part of the layout

</div>

**@RenderSection("Footer")**

<div class="layout">

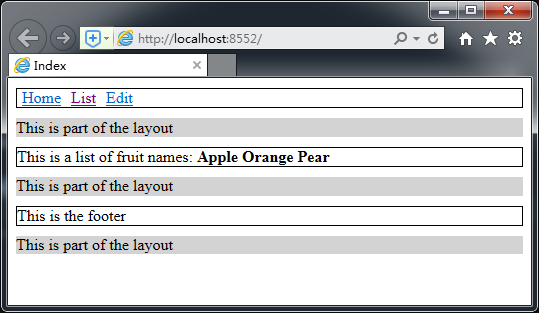
This is part of the layout

</div>

</body>

</html>

在Razor对布局进行解析时，RenderSection辅助器方法会显示视图中指定名称的分段内容。视图中未包含分段的内容，会插入在布局中使用RenderBody辅助器的地方。效果如图：



注：一个视图只能定义在布局中被引用的分段。如果试图在视图中定义布局中无对应的@ RenderSection辅助器调用的分段，MVC框架将会抛出异常。同时，默认情况下，视图必须含有布局中调用@RenderSection的所有分段，如果缺少，MVC框架同样会抛出异常。

一般情况下，不用把分段和视图的其余部分混杂在一起。约定是在视图的开始或结尾部分定义分段，以便更容易看到哪些内容区域被处理成分段，以及哪些将要由RenderBody辅助器来捕捉。推荐一种做法：把视图定义成一个个独立的分段，并包括一个体分段，如：

@model string[]

@{

ViewBag.Title = "Index";

}

@section Header{

<div class="view">

@foreach (string str in new[] { "Home", "List", "Edit" })

{

@Html.ActionLink(str, str, null, new { style = "margin:5px" })

}

</div>

}

<!--下面的方式将视图定义到了体分段中，即此时视图被定义成了一个独立的分段-->

@section Body{

<div class="view">

This is a list of fruit names:

@foreach (string name in Model)

{

<span><b>@name</b></span>

}

</div>

}

@section Footer{

<div class="view">

This is the footer

</div>

}

这种做法有利于建立更清晰的视图，并减少RenderBody捕捉无关内容的情况。下面是对这种方式的使用示例：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width" />

<style type="text/css">

div.layout {

background-color: lightgray;

}

div.view {

border: thin solid black;

margin: 10px 0;

}

</style>

<title>@ViewBag.Title</title>

</head>

<body>

@RenderSection("Header")

<div class="layout">

This is part of the layout

</div>

**@RenderSection("Body")**

<div class="layout">

This is part of the layout

</div>

@RenderSection("Footer")

<div class="layout">

This is part of the layout

</div>

</body>

</html>

1. 对分段进行测试

如果一个视图不需要或不希望提供特定内容，那么可以采取对一个分段提供默认内容的方式，如：

**@if (IsSectionDefined("Footer"))**

{

@RenderSection("Footer")

}

else

{

<h4>This is the default footer</h4>

}

IsSectionDefined辅助器可以使用要检查的分段名来判断是否定义了这个分段，如果定义了则返回一个真值（true）。

1. 渲染可选分段

由于默认情况下，视图必须含有布局中调用RenderSection的所有分段，否则将会抛出异常。为了查看这种异常的出现，现在对\_Layout布局进行修改，如：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width" />

<style type="text/css">

div.layout {

background-color: lightgray;

}

div.view {

border: thin solid black;

margin: 10px 0;

}

</style>

<title>@ViewBag.Title</title>

</head>

<body>

@RenderSection("Header")

<div class="layout">

This is part of the layout

</div>

@RenderSection("Body")

<div class="layout">

This is part of the layout

</div>

@if (IsSectionDefined("Footer"))

{

@RenderSection("Footer")

}

else

{

<h4>This is the default footer</h4>

}

**@RenderSection("scripts")**

<div class="layout">

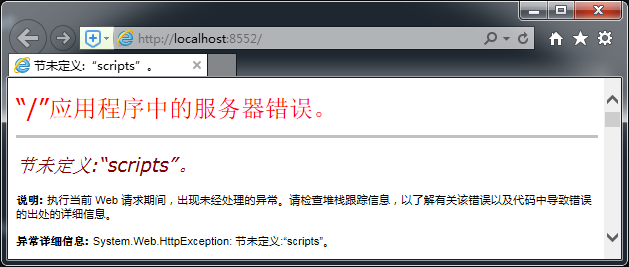
This is part of the layout

</div>

</body>

</html>

这样，将会看到如下异常：



当然可以使用前面提到的IsSectionDefined方法来避免这种情况的发送，但还有一个更好的办法，就是给RenderSection方法传递一个附加的false值，即使用可选分段，如：

@RenderSection("scripts", **false**)

### 使用分部视图

使用分部视图可以实现在程序中的不同地方使用同样的Razor标签和HTML标记片段，这一点在前面已经有了对应的介绍，就不多说了。分部视图具有以下几个特点：

* 含有标签
* 含有标记片段
* 具有独立性，是独立的视图文件
* 可以被包含在其他视图之中

1. 创建分部视图

首先，在/Views/Shared文件夹创建一个名为MyPartial的分部视图：

<div>

This is the message from the partial view.

@Html.ActionLink("This is a link to the Index action", "Index")

</div>

其次，通过HTML辅助器在另一个视图中调用这个分部视图：

@{

ViewBag.Title = "List";

**Layout = null;**

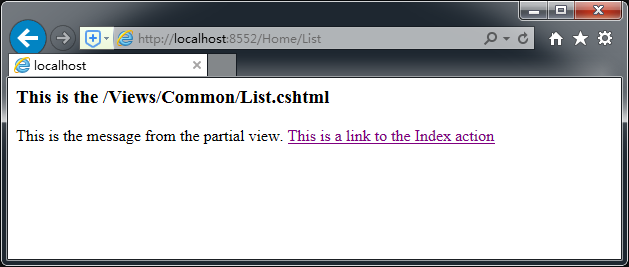
}

<h3>This is the /Views/Common/List.cshtml</h3>

**@Html.Partial("MyPartial")**

像这样对分部视图不指定扩展名的方式使用时，视图引擎会在常规位置处查找分部视图，即“/Views/Home”和“/Views/Shared”文件夹下查找。这里将Layout设置为null是为了避免使用之前布局中定义的分段。

下面来看一下效果：



提示：在上述分部视图中，对ActionLink辅助器方法的调用会根据所处理的请求，采用其控制器器的信息。也就是说会根据让该分部视图进行渲染的控制器生成对应的引用。

1. 使用强类型分部视图

现在看一下强类型的分部视图的创建和使用，分部视图名：MyStrongTypedPartial.cshtml。内容如下：

@model IEnumerable<string>

<div>

This is the message from the partial view.

<ul>

@foreach (string str in Model)

{

<li>

@str

</li>

}

</ul>

</div>

更新一下Common下的List视图，以便查看这一效果：

@{

ViewBag.Title = "List";

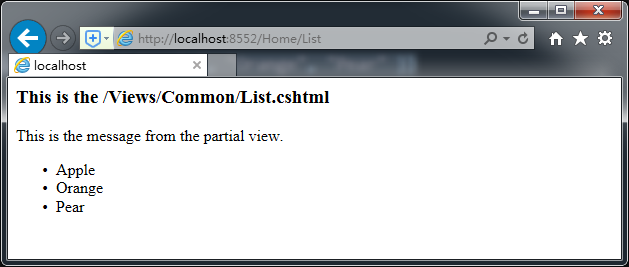
Layout = null;

}

<h3>This is the /Views/Common/List.cshtml</h3>

**@Html.Partial("MyStrongTypedPartial", new[] { "Apple", "Orange", "Pear" })**

效果图：



### 使用子动作

子动作（Child Action）是通过视图调用的动作方法。如果需要将某种控制器逻辑用于应用程序的多个地方，子动作可以使我们避免重复的控制器逻辑。它与动作之间的关系就像分部视图和视图一样。

比如，如果要显示某些数据驱动的“小部件”，这些“小部件”要出现在多个页面上，且含有与主动作无关的数据时，可能就要用到子动作。

1. 创建子动作

任何动作都可以作为一个子动作。现在通过在Home控制器添加一个新的动作方法来进行演示：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace WorkingWithRazor.Controllers

{

public class HomeController : Controller

{

//

// GET: /Home/

public ActionResult Index()

{

string[] names = { "Apple", "Orange", "Pear" };

return View(names);

}

public ActionResult List()

{

return View();

}

**[ChildActionOnly]**

**public ActionResult Time()**

**{**

**return PartialView(DateTime.Now);**

**}**

}

}

上面代码中动作方法Time通过调用PartialView方法渲染一个分部视图。ChildActionOnly注解属性可以确保一个动作方法只能在一个视图中作为一个子动作进行调用。动作方法并不一定非要使用这种方式才能成为一个子动作，但建议这么做，原因是这样可以防止该动作方法作为用户请求的一个结果被调用。

现在就可以看看如何调用了。注意，子动作典型地与分布视图相关联，但这不是必须的。下面是该动作方法的视图：

@model DateTime

<p>The time is: @Model.ToShortTimeString()</p>

1. 渲染子动作

可以用Html.Action辅助器调用子动作。我们修改一下List视图文件，用来渲染刚刚做的子动作：

@{

ViewBag.Title = "List";

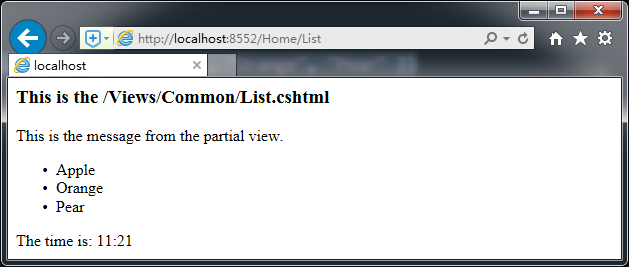
Layout = null;

}

<h3>This is the /Views/Common/List.cshtml</h3>

@Html.Partial("MyStrongTypedPartial", new[] { "Apple", "Orange", "Pear" })

**@Html.Action("Time")**



如果要调用的是其他控制器中的动作方法，需要提供控制器名称，如：

**@Html.Action("Time", "MyController")**

通过提供一个匿名类的对象，其属性对应于子动作方法的参数名，可以将参数传递给动作方法。如：

子动作方法：

**[ChildActionOnly]**

**public ActionResult Time(DateTime time)**

**{**

**return PartialView(time);**

**}**

调用方法：

**@Html.Action("Time", new { time = DateTime.Now })**