# 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)

{

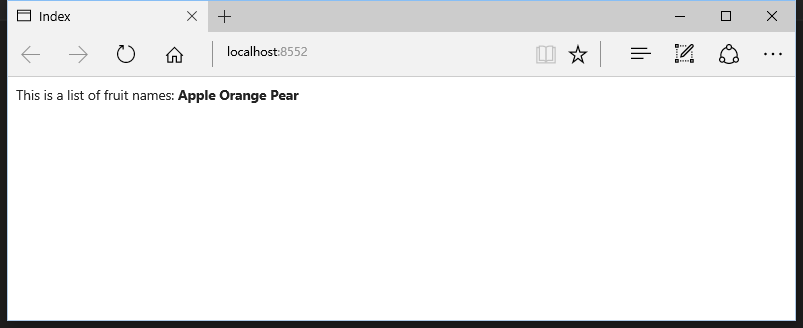
<span><b>@name</b></span>

}

### 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