# 控制器和动作

到达应用程序的每一个请求都是由控制器处理的。但要注意，不要把事务或数据存储逻辑放到控制器中，也不要生成用户界面。

在ASP.NET MVC框架中，控制器是含有请求处理逻辑的.NET类。其作用是封装应用程序逻辑。也就是说，控制器要负责处理输入请求、执行域模型上的操作，并选择渲染给用户的视图。

## 控制器的介绍

为了能够详细的说明控制器和动作的功能，这里使用“空（Empty）”模板创建一个名为“ControllersAndActions”的新的MVC项目（记得选择“创建单元测试项目（Create a unit test project）”）。

在MVC框架中，必须实现System.Web.Mvc命名空间的IController接口。这个接口很简单，只有唯一的一个方法：Execute，其在请求以控制器类为目标时被调用。MVC框架通过读取路由数据生成的controller属性值，便会指定请求的目标是哪一个控制器。

由于IController接口是一个相当低级的接口，因此必须做大量的工作才能达到预期效果。如下面所示的一个相当简单的用于演示的控制器类：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace ControllersAndActions.Controllers

{

public class BasicController : IController

{

public void Execute(RequestContext requestContext)

{

string controller = (string)requestContext.RouteData.Values["controller"];

string action = (string)requestContext.RouteData.Values["action"];

requestContext.HttpContext.Response.Write(string.Format("Controller: {0},Action: {1}", controller, action));

}

}

}

上面代码仅仅演示了通过与请求相关联的RouteData对象读取controller和action变量的值，并将其显示出来。MVC框架并未指出控制器应该如何处理请求，也就是说可以采用任何方式来处理。需要注意的是，MVC框架并未在这个Basic控制器上强加视图引擎。如何产生响应是控制器本身要做的事，MVC框架不会对生成响应所用的技术做任何假设。

MVC框架可以无限定制和扩展。我们可以通过实现IController接口，来创建自己的控制器类，根据自己的需求来决定该如何处理请求。也可以通过System.Web.Mvc.Controller类来派生控制器。

该类提供了三个关键特性：

* **动作方法**（Action Method）：一个控制器的行为被分解为多个方法（而不是只有单一的Execute方法）。每个动作方法被暴露给不同的URL，并通过输入请求提取的参数进行调用。
* **动作结果**（Action Result）：可以返回一个描述动作结果的对象（如，渲染一个视图，或重定向到一个不同的URL或动作方法），然后通过该对象实现目的。这种指定结果和执行它们之间的分离简化了单元测试。
* **过滤器**（Filter）：可以把可重用的行为（如认证）封装成过滤器，然后通过在源代码中放置一个[Attribute]（注解属性）的办法，把这种行为标注到一个或多个控制器或动作方法上。

除非已经有了一个非常明确的需求，否则创建控制器最好的办法是通过Controller类进行派生，这也是Visual Studio创建控制器的默认方式。如下面通过这种方式创建的一个简单控制器：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllersAndActions.Controllers

{

public class DerivedController : Controller

{

public ActionResult Index()

{

ViewBag.Message = "Hello from the DerivedController Index method";

return View("MyView");

}

}

}

Controller基类会实现Execute方法并负责调用动作方法，动作方法名与路由数据中的action的值匹配。

Controller类也连接到Razor视图系统。上面代码中返回的View方法的结果，在其中传递了希望渲染给客户端的视图名。下面是该视图的内容：

@{

ViewBag.Title = "MyView";

}

<h2>MyView</h2>

Message:@ViewBag.Message

## 接收输入

控制器经常要访问输入数据，如查询字符串值、表单值及路由系统根据输入URL解析所得到的参数。

而控制器访问输入数据的主要途径有以下三个：

* 通过一组**上下文对象**（context objects）进行提取；
* 作为参数（Parameters）被传递给动作方法而形成的数据；
* 明确地调用框架模型绑定（Model Binding）特性。

下面先重点针对上下文对象和动作方法参数的方式进行介绍，模型绑定的方式将在“模型绑定”的章节中介绍。

1. 通过上下文对象获取数据

当控制器是通过Controller基类派生而来的时候，便得到了一组便利属性（Convenience Property），可以用来访问与请求相关的信息。包括Request、Response、RouteData、HttpContexty以及Server。之所以将这些属性叫做便利属性，是因为它们每一个都从请求的ControllerContext实例（可以通过Controller.ControllerContext属性对其进行访问）接受了不同类型的数据。

下表是一些常见的上下位对象：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 描述 |
| Request.QueryString | NameValueCollection | 随该请求发送的GET变量 |
| Request.Form | NameValueCollection | 随该请求发送的POST变量 |
| Request.Cookies | HttpCookieCollection | 由浏览器随该请求发送的Cookies |
| Request.HttpMethod | string | 用于该请求的HTTP方法（动词，如GET或POST） |
| Request.Headers | NameValueCollection | 随该请求发送的整个HTTP报头 |
| Request.Url | Uri | 所请求的URL |
| Request.UserHostAddress | string | 形成该请求的用户的IP地址 |
| RouteData.Route | RouteBase | 为该请求所选择RouteTable.Routes条目 |
| RouteData.Values | RouteValueDictionary | 当前路由的参数（从URL或默认值提取） |
| HttpContext.Application | HttpApplicationStateBase | 应用程序状态库 |
| HttpContext.Cache | Cache | 应用程序缓存库 |
| HttpContext.Items | IDictionary | 当前请求的状态库 |
| HttpContext.Session | HttpSessionStateBase | 访问者的会话状态库 |
| User | IPrincipal | 已登录用户的认证信息 |
| TempData | TempDataDictionary | 为当前用户存储的临时数据项 |

在一个动作方法中，可以用这些上下文（Context）对象的任意一个，来获取与请求相关的信息，下面这段代码给出了简单获取这些信息的方法：

public ActionResult RenameProduct()

{

// 访问上下文对象的各个属性

string userName = User.Identity.Name;

string serverName = Server.MachineName;

string clientIP = Request.UserHostAddress;

DateTime dateStamp = HttpContext.Timestamp;

AuditRequest(userName, serverName, clientIP, dateStamp, "Renaming product");

// 接收 Request.Form 所递交的数据

string oldProductName = Request.Form["OldName"];

string newProductName = Request.Form["NewName"];

bool result = AttemptProductRename(oldProductName, newProductName);

ViewData["RenameResult"] = result;

return View("ProductRenamed");

}

1. 使用动作方法参数

使用动作方法参数的方式提取数据比通过上下文对象手工提取更加灵活，且可读性更强。而且，这种方式还利于单元测试——不需要模仿控制器类的便利属性。如：

public ActionResult ShowWeatherForecast()

{

string city = (string)RouteData.Values["city"];

DateTime forDate = DateTime.Parse(Request.Form["forDate"]);

// …在这里实现天气预报…

return View();

}

可以把它重写成使用参数的形式，如：

public ActionResult ShowWeatherForecast(string city, DateTime forDate)

{

// …在这里实现天气预报…

return View();

}

需要注意的是在动作方法中不允许使用out或ref这样的出参，一是这么做没有任何意义，而且在ASP.NET MVC中，如果出现这种参数将会直接抛出异常。MVC框架会自动对动作方法的参数进行赋值，这是通过检查上下文对象来完成的。对参数名的处理不区分大小写，因此，像上面那样的“city”的动作方法参数能够被Request.Form["City"]的值所填充。

* 参数对象实例化

Controller基类使用叫作“**值提供器**（Value Provider）”和“**模型绑定器**（Model Binder）”的MVC框架组件来获取动作方法的参数值。

**值提供器**：表现一组可用于控制器的数据项。有一组内建的值提供器，它们可以抓取Request.Form、Request.QueryString、Request.Files，以及RouteData.Values的数据项。然后这些值被传递给模型绑定器，模型绑定器会尝试将这些数据映射成动作方法参数的数据类型。

默认的**模型绑定器**能创建并填充任何.NET类型的对象，包括集合和项目专用的自定义类型。

* 可选参数与强制参数

MVC框架如果找不到引用类型参数（如string或object）的值，动作方法仍然会被调用，但对该参数会使用一个null值。若找不到值类型参数（如int或double）的值，则会抛出一个异常，并且不会调用该动作方法。

值类型参数是强制的，为了使它们可选，可以为其指定一个默认值，或将该参数的类型改为可空（nullable）类型（如int?或DateTime?），这样，MVC框架在无值可用时会传递null值。

引用类型参数是可选的，为了使其为必需的（如以保证传递一个非空值），可以把一些代码添加到该动作方法的顶部，以拒绝null值。如，当值为null时抛出一个ArgumentNullException异常。

* 指定默认参数值

如果希望处理不含动作方法参数值的请求，但又不想在代码中检查null值或抛出异常，可以使用C#的可选参数特性来代替。如：

public ActionResult Search(string query = "all", int page = 1)

{

// …处理请求…

return View();

}

这时，如果MVC框架发现无可用的值，则将使用指定的默认值代替。

可选参数可以用于字面类型，字面类型（LiteralType）是不需要用new关键字定义的类型，包括string、int和double等。

注意：如果一个请求确实包含了一个参数的值，但该值无法转换为正确的类型，那么框架会传递该参数类型的默认值，并在一个名为“ModelState（模型状态）”的特殊上下文对象中将这个尝试值注册为一个验证错误。除非检查ModelState中的验证错误，否则，当用户在表单中输入了不良数据的情况下，可能会得到奇怪的境况：该请求还是被处理了，就好像用户没有输入任何数据，或输入的是这个默认值一样。

## 产生输出

当控制器完成一个请求之后，通常要生成一个响应。通过实现IController接口创建“裸机控制器（Bare-metal Controller）”（单纯继承控制器接口的原始的控制器——需要手动实现接口的功能及各种必须功能等）时，需要负责处理请求的各个方面，包括生成对客户端的响应。如：要想发送一个HTML响应，必须创建并装配HTML数据，并用Response.Write方法把它发送至客户端。类似地，若想将用户浏览器重定向到另一个URL，则需要调用Response.Redirect方法，并直接传递所需的URL。如下演示了这种需求（加粗部分）：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace ControllersAndActions.Controllers

{

public class BasicController : IController

{

public void Execute(RequestContext requestContext)

{

string controller = (string)requestContext.RouteData.Values["controller"];

string action = (string)requestContext.RouteData.Values["action"];

**if (action.ToLower() == "redirect")**

**{**

**requestContext.HttpContext.Response.Redirect("/Derived/Index");**

**}**

**else**

**{**

**requestContext.HttpContext.Response.Write(string.Format("Controller: {0}, Action: {1}", controller, action));**

**}**

}

}

}

当控制器派生于Controller类时，可以使用同样的办法。在Execute方法中读取requestContext.HttpContext.Reponse属性时，返回的是HttpResponseBase类，这个类派生控制器中可直接通过Controller.Response属性进行使用。如下所示（加粗部分）：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllersAndActions.Controllers

{

public class DerivedController : Controller

{

public ActionResult Index()

{

ViewBag.Message = "Hello from the DerivedController Index method";

return View("MyView");

}

**public void ProduceOutput()**

**{**

**if (Server.MachineName == "TINY")**

**{**

**Response.Redirect("/Derived/Index");**

**}**

**else**

**{**

**Response.Write("Controller: Derived, Action: ProduceOutput");**

**}**

**}**

}

}

上面示例代码中，ProduceOutput方法使用Server.Machine.Name属性的值来决定发送给客户端的响应（TINY是其中一个开放机器的名称）。

这种方式是可行的，但仍存在以下几个问题：

* 控制器类必须包含详细的HTML或URL结构，这些使得控制器类更加难以阅读和维护。
* 将响应直接生成为输出的控制器难以进行单元测试。为了确定输出表示的是什么，需要创建Response对象的模仿实现，然后才能处理从控制器接收到的输出。如，这可能意味着要解析HTML关键字，这是费时而痛苦的。
* 这种处理每个响应微小细节的方式是乏味而易错的。

还好，MVC框架提供了一个很好的特性来解决这种问题——**动作结果**（ActionResult），这将在后续内容中逐一介绍。

1. 理解动作结果

MVC框架通过使用动作结果把指明意图和执行意图分离开。此处不直接使用Response对象，而是返回一个ActionResult类的对象，它描述控制器响应要完成的功能，如渲染一个视图、重定向到另一个URL或动作方法等。

注：动作结果系统是一种**命令模式**（Command Pattern）。该模式描述你所处的场景，并发送一些对象，这些对象描述了要执行的操作。

当MVC框架从动作方法接收到一个ActionResult对象时，它调用有这个对象定义的ExecuteResult方法。然后在该动作结果的实现中处理Response对象，生成符合你意图的输出。（严格上说，MVC框架在接到动作结果对象时，是调用该对象类型对应的动作结果处理类（如：RedirectResult、ViewResult类等——这些类都继成于ActionResult类（这是一个抽象类，作用就想其描述的那样：封装一个操作方法的结果并用于代表该操作方法执行框架级操作。）），然后执行该类的ExecuteResult方法——这是动作结果的一个实现方法，主要负责处理Response对象，最终生成所期望的输出）。如下述清单所示（在项目中创建一个Infrastructure文件夹，然后在这里创建演示类：CustomRedirectResult类）：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllersAndActions.Infrastructure

{

public class CustomRedirectResult : ActionResult

{

public string Url { get; set; }

public override void ExecuteResult(ControllerContext context)

{

string fullUrl = UrlHelper.GenerateContentUrl(Url,context.HttpContext);

context.HttpContext.Response.Redirect(fullUrl);

}

}

}

下面是如何使用该类：

/// <summary>

/// 演示对 CustomRedirectResult 对象调用的使用方法

/// </summary>

/// <returns></returns>

public ActionResult ProduceOutput()

{

if (Server.MachineName == "TINY")

{

return new CustomRedirectResult { Url="/Basic/Index"};

}

else

{

Response.Write("Controller: Derived, Action: ProduceOutput");

return null;

}

}

上面介绍了如何自定义实现动作结果，并了解了其基本的工作模式。一般我们直接使用微软提供的已经过完全测试的动作结果，而且，很多动作结果都有其便利的辅助器方法。如下表中展示了一些常用的内建ActionResult类型：

|  |  |  |
| --- | --- | --- |
| 类型 | 描述 | 辅助器方法 |
| ViewResult | 返回指定的或默认的视图模板 | View |
| PartialViewResult | 返回指定的默认的分部视图模板 | PartialView |
| RedirectToRouteResult | 将HTTP301（或302）重定向发送给一个动作方法或特定的路由条目，根据路由配置生成一个URL | RedirectToAction  RedirectToActionPermanent  RedirectToRoute  RedirectToRoutePermanent |
| RedirectResult | 将HTTP301或302重定向发送给一个特定的URL | Redirect  RedirectPermanent |
| HttpUnauthorizedResult | 将响应的HTTP状态码设置为401（意为“未授权”），这会引发当前的认证机制（表单认证或Windows认证）要求访问者进行登录 | None |
| HttpNotFoundResult | 返回一个HTTP的“404——未找到”的错误 | HttpNotFound |
| HttpStatusCodeResult | 返回一个指定的HTTP码 | None |
| EmptyResult | 什么也不做 | None |

1. 通过渲染视图返回HTML

动作方法最常用的一种响应形式是生成HTML，并将其发送给浏览器。如下面示例使用ViewResult指定了一个要被渲染的视图（通过View辅助器方法创建了一个ViewResult实例对象）：

public ViewResult Index()

{

return View("Homepage");

}

当MVC框架调用ViewResult对象的ExecuteResult方法时，将开始搜索已经指定的视图。如果在项目中使用了区域，则框架将查找以下位置：

* /Areas/<AreaName（区域名）>/Views/<ControllerName（控制器名）>/<ViewName（视图名）>.aspx
* /Areas/<AreaName>/Views/<ControllerName>/<ViewName>.ascx
* /Areas/<AreaName>/Views/Shared/<ViewName>.aspx
* /Areas/<AreaName>/Views/Shared/<ViewName>.ascx
* /Areas/<AreaName>/Views/<ControllerName>/<ViewName>.cshtml
* /Areas/<AreaName>/Views/<ControllerName>/<ViewName>.vbhtml
* /Areas/<AreaName>/Views/Shared/<ViewName>.cshtml
* /Areas/<AreaName>/Views/Shared/<ViewName>.vbhtml

从上可见，即时在创建项目时指定的是Razor，框架也会查找遗留视图引擎创建的视图（文件扩展名为.aspx和.ascx）。框架也查找了C#和VB的.NET Razor模板（.cshtml文件属于C#模板，.vbhtml属于VB模板，Razor语法在这些文件中是一样的，而代码使用的是不同的语言）。MVC框架会依次检测这些文件是否存在，且，只要找到一个匹配的对象，便会用这个视图来渲染该动作方法的结果。

如果上述区域目录下未找到适当的文件，或未使用区域，则框架便会查找以下的位置：

* /Views/<ControllerName（控制器名）>/<ViewName（视图名）>.aspx
* /Views/<ControllerName>/<ViewName>.ascx
* /Views/Shared/<ViewName>.aspx
* /Views/Shared/<ViewName>.ascx
* /Views/<ControllerName>/<ViewName>.cshtml
* /Views/<ControllerName>/<ViewName>.vbhtml
* /Views/Shared/<ViewName>.cshtml
* /Views/Shared/<ViewName>.vbhtml

同样，只要MVC框架查找到合适的文件，便会停止搜索，且使用已经找到的这个视图将响应渲染给客户端。

在框架搜索相应位置的视图文件时，对于控制器的部分，将会忽略Controller，如控制器：ExampleController将会以Example作为控制器名进行搜索。

**单元测试：渲染一个视图**

为了测试动作方法渲染的视图，可以检测它返回的ViewResult对象。这当然不完全是一回事——毕竟，这并不是通过检查最终生成的HTML来跟踪这一过程——但也十分密切，只要能够充分确信MVC框架的视图系统会恰当工作。下面是我们的单元测试类：

*using System;*

*using Microsoft.VisualStudio.TestTools.UnitTesting;*

*using ControllersAndActions.Controllers;*

*using System.Web.Mvc;*

*namespace ControllersAndActions.Tests*

*{*

*[TestClass]*

*public class ActionTests*

*{*

*[TestMethod]*

*public void ViewSelectionTest()*

*{*

*// 准备——创建控制器*

*ExampleController target = new ExampleController();*

*// 动作——调用动作方法*

*ViewResult result = target.Index();*

*// 断言——检测结果*

*Assert.AreEqual("Homepage", result.ViewName);*

*}*

*}*

*}*

当对选择的默认视图的动作方法进行测试时，有些区别，如动作方法：

*public ViewResult Index()*

*{*

*return View();*

*}*

此时，需要对视图名采用空字符串（""），如：

*Assert.AreEqual("", result.ViewName);*

MVC框架搜索视图的目录序列是“约定优于配置”这一规则的另一个例子。不需要用框架注册视图文件，只需要把它们放在一组已知的位置即可。如，假设动作方法未指定视图，MVC框架则将会假设要渲染一个与动作方法同名的视图。

View方法有多种重载版本，它们对应于在ViewResult对象上设置的不同属性。如，通过明确地命名一个布局，可以重写一个视图所使用的（默认）布局，如：

public ViewResult Index()

{

return View("Index","\_AlternateLayoutPage");

}

注意，上述指定的布局文件名不需要带扩展名。

**通过路径指定视图**

命名约定办法虽然简单，但是很方便，且其确实能够限制我们能够渲染的视图。如果要渲染的一个特定的视图，可以通过提供一个明确的路径并绕过搜索阶段来完成。如：

*public ViewResult Index()*

*{*

*return View("~/Views/Other/Index.cshtml");*

*}*

注意，指定的路径必须以“/”或“~/”开始，并包括文件扩展名。但是，让我们要这么做的时候需要谨慎一些，因为这样指定路径的方式不利于应用程序的进一步扩展和维护，这是一种绑定或耦合。最好的做法是通过控制器的动作方法来重定向要渲染的视图。

1. 将数据从动作方法传递给视图

在实际的项目中经常会需要将数据从一个动作方法传给视图。而MVC框架对此提供了多种不同的方法，下面将给出描述：

* 提供视图模型对象

将一个对象作为View方法的参数发送给视图，如下面示例：

public ViewResult Index()

{

DateTime date = DateTime.Now;

return View(date);

}

上述示例传递了一个DateTime对象作为视图模型。可以在视图中用Razor的Model关键字来访问这个对象，如：

@{

// 演示如何获取动作方法通过 View 方法的参数发送给视图的视图模型信息

ViewBag.Title = "Index";

}

<h2>Index</h2>

The day is: @(((DateTime)Model).DayOfWeek)

需要注意的是，非类型或弱类型视图易产生杂乱的视图，这可以通过强类型视图来加以调整。

上面的视图就是一个非类型视图（或称为弱类型视图），该视图不知道关于视图模型对象的任何情况，而把它作为object的一个实例来看待。为了得到DayOfWeek属性的值，需要将其转换成DateTime的一个实例。然即使这样，这种做法仍会产生杂乱的视图。下面通过创建强类型视图加以调整，如：

@\*改用强类型视图，以避免产生杂乱的视图\*@

@model DateTime

@{

// 演示如何获取动作方法通过 View 方法的参数发送给视图的视图模型信息

ViewBag.Title = "Index";

}

<h2>Index</h2>

@\*弱类型视图方式：

The day is: @(((DateTime)Model).DayOfWeek)\*@

@\*强类型视图方式：\*@

The day is: @Model.DayOfWeek

注意：当指定模型类型时，要使用小写的“m”（如：@**m**odel DateTime

），而在读取模型值时，要用大写的“M”（如：@**M**odel.DayOfWeek）。

视图模型对象的单元测试

对于视图模型对象的单元测试，我们可以通过ViewResult.ViewData.Model 属性访问从动作方法传递给视图的视图模型对象。下面是一个简单的动作方法实例：

public ViewResult Index()

{

return View((object)"Hello,World");

}

该动作方法传递了一个字符串作为视图模型对象。该字符串被转换为object，以使编译器不会认为我们要用的是指定视图名称的那个View重载版本。对应的具体测试方法如下：

/// <summary>

/// 通过 ViewResult.ViewData.Model 属性访问从动作方法传递给视图的视图模型对象。

/// </summary>

[TestMethod]

public void ViewSelectionTest()

{

// 准备——创建控制器

ExampleController target = new ExampleController();

// 动作——调用动作方法

ViewResult result = target.Index();

// 断言——检查结果

Assert.AreEqual("Hello,World", result.ViewData.Model);

}

* 用ViewBag传递数据

ViewBag特性允许在一个动态对象上定义任意属性，并在视图中访问它们。这个动态对象可以通过Controller.ViewBag属性进行访问，如：

/// <summary>

/// 使用视图包特性：ViewBag

/// </summary>

/// <returns></returns>

public ViewResult Index()

{

ViewBag.Message = "Hello";

ViewBag.Date = DateTime.Now;

return View();

}

就像上面演示的那个，在动态定义属性：Massage和Date之前，它们是不存在的，不需要任何准备就可以创建它们。这是很方便的一个特性。要做视图中读取这些数据时，只有简单地采用在动作方法中设置的同样属性即可。如：

@{

ViewBag.Title = "Index";

}

<h2>Index</h2>

The day is: **@ViewBag.Date.DayOfWeek**

<p>

The message is: **@ViewBag.Message**

</p>

这样做的好处是：ViewBag便于将多个对象发送给视图。当用动态对象进行工作时，可以在视图中键入属性和方法调用的任意序列，如：

The day is: @ViewBag.Date.DayOfWeek.***Blah.Blah.Blah***（这里Blah是虚字，表示“一系列调用”）。

然而，这么做有一个不足，那就是：Visual Studio不能对包括ViewBag在内的任何动态对象提供智能感应支持，而在视图被渲染之前，不支持诸如“对此无法展示”之类的错误提示。

所以，在平常的项目中经常需要使用ViewBag的灵活性和强类型视图相结合的方式来满足我们的需求，这两种结合使用不会有任何问题。

ViewBag的单元测试

可以通过ViewResult.ViewBag属性来读取ViewBag的值，下面是测试方法：

/// <summary>

/// 通过 ViewResult.ViewBag 来读取 ViewBag，并对其进行测试

/// </summary>

[TestMethod]

public void ViewSelectionTest()

{

// 准备——创建控制器

ExampleController target = new ExampleController();

// 动作——调用动作方法

ViewResult result = target.Index();

// 断言——检查结果

Assert.AreEqual("Hello", result.ViewBag.Message);

}

1. 执行重定向

不是所有的动作方法都是或不是任何时候都需要直接产生输出，有时候我们需要把用户的浏览器重定向到另一个RUL，而且，大多数情况，这个URL是应用程序中的另一个动作方法，它可以生成希望用户看到的输出。

POST/Redirect/GET模式

重定向最频繁的使用是在处理HTTP POST请求的动作方法中。当想修改应用程序的程序状态时，才会使用POST请求。如果在请求处理之后简单地返回HTML，会陷入这样的风险：用户点击浏览器的刷新按钮，并再次递交该表单，这会引发异常和不符合需求的结果。

为了解决这一问题，可以遵循一种“Post/Redirect/Get”的模式。在该模式中，先接受一个POST请求（**POST**）、对该请求进行处理，然后重定向（**Redirect**）浏览器，以便由浏览器形成另一个GET请求（**GET**）的URL。GET请求不会修改应用程序的状态，因此，该请求的任何不经意的再次递交都不会引起任何问题。

上面的描述也体现了动作方法处理POST请求的安全工作流程：Post/Redirect/Get。即，用一个POST动作方法接受用户递交的POST请求，在该方法中对请求进行处理，然后用重定向方法把用户重定向到另一个GET方法。

在执行重定向时，给浏览器发送的是以下两个HTTP代码之一：

* 发送HTTP代码302，这是一个**临时**重定向。它是最常用的重定向类型，而且，当使用Post/Redirect/Get模式时，这就是要发送的代码。
* 发送HTTP代码301，它表示一个**永久**重定向。要小心使用它，因为它指示HTTP代码接收器不要请求原先的URL，而使用包含在重定向代码中的新URL。如果拿不准，则使用临时重定向，即发送302代码。
* **重定向到字面URL**

一般最常用最基本的方法是通过调用Redirect方法进行重定向，它返回RedirectResult类的一个实例，如：

/// <summary>

/// 重定向到一个字面 URL

/// </summary>

/// <returns></returns>

public RedirectResult Redirect()

{

return Redirect("/Example/Index");

}

Redirect方法发送的是一个临时重定向（HTTP代码：302），可以使用RedirectPermanent方法发送的是一个永久重定向（HTTP代码：301）。如：

/// <summary>

/// 重定向到一个字面 URL

/// </summary>

/// <returns></returns>

public RedirectResult Redirect()

{

// 永久重定向

return RedirectPermanent("/Example/Index");

}

通过Redirect的重载方法，通过一个布尔型参数指定是否永久重定向。

单元测试：字面重定向

字面重定向易于测试，可用RedirectResult类的Url和Permanent属性来读取URL和永久或临时重定向。如：

[TestMethod]

public void RedirectTest()

{

// 准备——创建控制器

ExampleController target = new ExampleController();

// 动作——调用动作方法

RedirectResult result = target.Redirect();

// 断言——检查结果

Assert.IsFalse(result.Permanent);

Assert.AreEqual("/Example/Index", result.Url);

}

* **重定向到路由系统的URL**

用字面URL重定向的问题是，对路由方案的任何修改，都意味着你需要检查代码，并对这些URL进行更新。

对于这种问题，可以使用路由系统来解决，通过RedirectToRoute方法来生成有效的URL，如：

/// <summary>

/// 重定向到一个路由系统的 URL

/// </summary>

/// <returns></returns>

public RedirectToRouteResult Redirect()

{

// 重定向到一个路由系统的 URL

return RedirectToRoute(new

{

controller = "Example",

action = "Index",

ID = "MyID"

});

}

该方法会发布一个临时重定向。对于永久重定向可以使用RedirectToRoutePermanent方法。这两个方法都以一个匿名类型作为参数，然后其属性被传递给路由系统，以生成一个URL。

单元测试：路由重定向

测试代码如下：

/// <summary>

/// 测试： 路由重定向

/// </summary>

[TestMethod]

public void RedirectValueTest()

{

// 准备——创建控制器

ExampleController target = new ExampleController();

// 动作——调用动作方法

RedirectToRouteResult result = target.RedirectToRoute();

// 断言——检查结果

Assert.IsFalse(result.Permanent);

Assert.AreEqual("Example", result.RouteValues["controller"]);

Assert.AreEqual("Index", result.RouteValues["action"]);

Assert.AreEqual("MyID", result.RouteValues["ID"]);

}

* **重定向到一个动作方法**

可以通过RedirectToAction方法方便的重定向到一个动作方法。该方法是RedirectToRoute的一个封装程序，让用户指定动作方法和控制器的值，而不需要创建一个匿名类型，如：

/// <summary>

/// 用 RedirectToAction 方法重定向

/// </summary>

/// <returns></returns>

public RedirectToRouteResult RedirectToAction()

{

return RedirectToAction("Index");

}

上述示例代码中，只指定了一个动作方法，此时，系统默认该方法为当前控制器的动作方法。如果需要重定向到另一个控制器，则需要以参数的方式提供其名称，如：

// 重定向到指定控制器（另一个控制器）的动作方法

return RedirectToAction("Index", "Basic");

还有一些其他的重载版本，可以用来为URL的生成提供额外的值，虽然其采用的匿名类型表示的方式破坏了一定的便利性，但仍具有很好的易读性。

需要注意的是：为控制器和动作方法提供的值，在被传递给路由系统之前是不会被检验的。在开发的时候需要确保指定的目标是实际存在的。

同样，该方法提供的是一个临时重定向，其永久重定向的方法是：RedirectToActionPermanent。

知识点：保留重定向数据

重定向是跨请求的，ViewBag不能用于跨请求的情况下控制器与视图之间的数据传递。重定向将导致浏览器递交一个新的HTTP请求，这样会失去对原先请求细节的访问。此时，如果需要把当前请求的数据保传递给下一个请求，可以使用TempData（临时数据）特性，

TempData类似于Session数据，只不过TempData的值在被读取之后仅被标记为删除状态，当请求被处理完成后才真正删除。下面是一个示例：

public RedirectToRouteResult RedirectToAction()

{

// 使用 TempData 保存重定向数据，以实现跨请求的重定向情况下控制器与视图之间的数据传递

TempData["Message"] = "Hello";

TempData["Date"] = DateTime.Now;

// 重定向到当前控制器的动作方法

return RedirectToAction("Index");

}

上述示例方法中，在处理请求时，在TempData集合中设置了一些值，然后把用户的浏览器重定向到同一个控制器中的Index动作方法。开发时，可以在母版动作方法中读回的数据，然后把它们传递给视图，如：

public ViewResult Index()

{

// 读取重定向至该方法之前设置的临时数据的值

ViewBag.Message = TempData["Message"];

ViewBag.Date = TempData["Date"];

return View();

}

更直接的做法是直接在视图中读取，如：

The day is: @**(((DateTime)TempData["Date"]).DayOfWeek)**

<p/>

The message is:@**TempData["Message"]**

如果在视图中直接读取了这些值，就不需要在动作方法中使用ViewBag特性获取值了。然而，必须把TempData结果转换成相应的类型。

利用Peek方法，可以得到TempData的值，而不把它标记为删除，如：

// 使用 Peek 方法实现读取 TempData 中的值但不将其标记为删除的方式

DateTime time = (DateTime)TempData.Peek("Date");

利用Keep方法可以保留一个否则将被删除的值，如：

TempData.Keep("Date");

但是，Keep方法是临时保留，并非永久保留，如果该值被再次读取，则它将被标记为删除。如果想存储一些数据，以使它们在请求被处理后不会被自动删除，请使用session数据（注意，session数据会占用服务器资源，只有在会话过期后才会被删除）。

1. 返回错误及HTTP代码