# URL 路由

在Web Form 情况下，每一个 ASPX页面既是一个文件，又是一个队请求自包含的响应。而在 MVC 情况下，请求是由控制器类中的动作方法处理的，而且与硬盘上的文件没有一对一的相互关系。

ASP.NET 平台为了处理 MVC 的 URL，采用了路由系统，它主要有两个功能：

* 考查一个输入 URL（Incoming URL），并推出该请求想要的是哪一个控制器和动作。这正是接收到一个客户端请求时，希望路由系统去做的事情。
* 生成一个输出 URL（Outgoing URL），这些 URL 是在视图渲染的 HTML 中出现的 URL，以便使用户点击这些链接时，调用一个特定的动作（此时，它又变成了输入 URL）。

## URL 模式

路由系统用一组路由来实现它的功能。这些路由共同组成了应用程序的URL架构（Schema）或方案（Scheme），这种URL架构（或方案）是应用程序能够识别并能对之作出相应的一组URL。

每一条路由都包含一个URL模式（Pattern），用它与一个输入的URL进行比较，如果该模式与这个URL匹配，那么它（URL模式）便被路由系统用来对这个URL进行处理。

URL模式主要有连个关键行为：

* URL 模式是保守的（Conservative）：只匹配与模式具有相同片段数的URL。
* URL模式是宽松的（Liberal）：如果一个URL正好具有正确的片段数，该模式就会用来为片段变量提取值，而不管这个值可能是什么。

如示例：

|  |  |  |
| --- | --- | --- |
| <http://mysite.com> | /Admin | /Index |
|  | First Segment | Second Segment |
|  | 片段1 | 片段2 |

## 简单路由的创建及注册

定义路由的文件 RouteConfig.cs 文件是在 App\_Start 文件夹中的。在其中定义的静态 RegisterRoutes 是通过 Global.asax.cs 文件进行调用的，当启动应用程序时，它建立了一些核心的 MVC 特性。

基本流程如下：

底层的 ASP.NET 平台在 MVC 第一次启动时调用Global.asax.cs —>Application\_Start() **——>** RouteConfig.cs—> RegisterRoutes()。

定义函数类似如下样子：

public static void RegisterRoutes(RouteCollection routes)

{

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

}

Global.asax.cs中的Application\_Start()函数调用方式如下方式：

RouteConfig.RegisterRoutes(RouteTable.Routes);

注册路由的一个方便的方法是，使用 RouteCollection 类所定义的MapRoute方法。如：

routes.MapRoute("MyRoute", "{controller}/{action}");

也可以通过创建一个新的 Route 来实现，如：

Route myRoute = new Route("{controller}/{action}", new MvcRouteHandler());

routes.Add("MyRoute", myRoute);

上述两种方式的效果是一样的。

## 定义默认值

默认URL被表示成“~/”送给路由系统。由于URL模式是保守的（即它们只匹配指定片段数的URL），如果要改变这种默认行为，则需要使用默认值——当URL不包含与一个片段匹配的值时，便使用默认值。如下面粗体字部分：

public class RouteConfig

{

public static void RegisterRoutes(RouteCollection routes)

{

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

// 通过创建一个新的 Route 实现路由的注册

//Route myRoute = new Route("{controller}/{action}", new MvcRouteHandler());

//routes.Add("MyRoute", myRoute);

// 使用 RouteCollection 类所定义的MapRoute方法实现路由的注册（效果与上面方式相同）

**// 下面第三个参数提供了一个包含默认路由的值的对象，当 URL 片段无匹配的值时（如片段少于定义给定的片段数时），**

**// 便会使用默认值（片段数需要符合定义的路由片段数，太多时将不做匹配）。**

**routes.MapRoute("MyRoute", "{controller}/{action}", *new { controller = "Home", action = "Index" }*);**

}

}

## 使用静态URL片段

1、有时我们不仅需要URL模式的所有片段都是可变的，也会需要创建具体静态片段的模式。比如当我们需要支持带有某种前缀的URL，例如带有Public前缀的URL：http://mydomain.com/**Public**/Home/Index。

如：routes.MapRoute("", "**Public**/{controller}/{action}", new { controller = "Home", action = "Index" });

上面这句话的意思是：将匹配具有三个片段的URL，但第一个必须是Public，其他两个片段可以有任何值，并将被用于controller和action变了。如果没有后两个片段，则将使用默认值。

2、可以创建一个既有静态也有可变元素片段的 URL 模式，如：

// 创建一个既有静态也有可变元素片段的 URL 模式

// MapRoute 将在路由集合的末尾添加一条新的路由

// 该路由需要放在其他路由之前，原因是路由是按照他们在 RouteCollection 对象中出现的顺序被运用的。

// 我们可以将一条路由按照指定的位置添加，但一般不采用这种方式，原因是让路由以它们被定义的顺序来

// 运用更容易理解运用于一个应用程序的路由。

// 因此，路由系统是先匹配最前面定义的路由，如果不能匹配，则继续下一个，所以，最好先定义较具体的路

// 由，然后次之，以此类推。

routes.MapRoute("", "X{controller}/{action}");

假设如下颠倒顺序：

routes.MapRoute("MyRoute", "{controller}/{action}", new { controller = "Home", action = "Index" });

routes.MapRoute("", "X{controller}/{action}");

那么，第一条路由匹配任何具有0、1、2片段的URL，它将是被使用的一条。更具体的路由现在是列表的第二条，它将是不可到达的。新路由（第二条）去掉URL的前导“X”，但旧路由（第一条）却不会这么做。因此，像这样的一条URL：

http://mydomain.com/XHome/Index

将以名为“XHome”的控制器为目标，而这是不存在的，因此会导致一个“404——未找到”错误被发送给用户。

3、使用静态片段和默认值为特定的路由创建一个别名

如果已经公开发布了URL方案，并且它与用户形成了一种契约，此时创建别名是有用的。如果我们重构程序，则需要保留以前的URL格式。下面示例给出了如何保留旧式URL方案的路由：

// 结合静态片段和默认值为特定的路由创建一个别名

// 匹配第一个片段是 Shop 的任意两片段 URL，action 的值取自第二个 URL 片段。

// 由于此 URL 模式未提供 controler 的可变片段，所以会使用提供的默认值（“Home”）。即对

// Shop 控制器上的一个动作的请求会被转换成对 Home 控制器的请求。

routes.MapRoute("ShopSchema", "Shop/{action}", new { controller = "Home" });

如果更彻底的，我们可以为被重构且不再出现在控制器中的动作方法创建别名，如下：

routes.MapRoute("ShopSchema2", "Shop/OldAction", new { controller = "Home", action = "Index" });

## 定义自定义片段变量

在MVC中有三个片段变量的名称是被保留的，不能用于自定义片段变量名，除此之外均可命名为自定义片段变量。这三个片段变量分别是：controller（控制器片段变量）、action（动作方法片段变量）和area（区域片段变量）。

自定义片段变量示例：

// 定义一个名为“id”的自定义变量（粗体字部分）。如果没有与之对应的片段内容，则将使用默认值

routes.MapRoute("MyRoute", "{controller}/{action}/{id}",

new { controller = "Home", action = "Index", **id = "DefaultId"** });

我们可以在动作方法中通过RouteData.Values属性访问任何一个片段变量。如下：

/// <summary>

/// 获取 URL 模式中自定义变量（“id”）的值，并用 ViewBag 将它传递给视图。

/// </summary>

/// <returns></returns>

public ActionResult CustomVariable()

{

ViewBag.Controller = "Home";

ViewBag.Action = "CustomVariable";

ViewBag.CustomVariable = **RouteData.Values["id"]**;

return View();

}

### 用自定义变量作为动作方法参数

在开发过程中除了可以使用RouteData.Values属性访问自定义路由变量，还可以以URL模式中的变量相匹配的名称，来定义动作方法的参数。此时，MVC框架将把从URL获得的值作为参数传递给动作方法。

如下面的写法：

/// <summary>

/// 用自定义变量作为动作方法参数

/// </summary>

/// <param name="id">

/// ***MVC 框架会尝试将 URL 的值转换成所定义的参数类型，这***

/// ***里将转换成 string ，MVC 框的这以特性将方便开发者不必***

/// ***自行做转换。***

/// </param>

/// <returns></returns>

public ActionResult CustomVariable(**string id**)

{

ViewBag.Controller = "Home";

ViewBag.Action = "CustomVariable";

ViewBag.CustomVariable = **id**;

return View();

}

### 定义可选URL片段

可选URL片段是指用户不需要指定、但又未指定默认值的片段。

下面示例通过将默认值设置为“UrlParameter.Optional”，便指明了一个片段变量是可选的。***在下面实例中，只有当输入URL中存在相应片段时，id变量才被添加到变量集合中，当未为可选片段变量提供值时，对应的参数值将为null。***

// 定义可选 URL 片段。该路由的效果是：无论是否提供id，都将进行匹配

routes.MapRoute("MyRoute", "{controller}/{action}/{id}",

new { controller = "Home", action = "Index", id = **UrlParameter.Optional** });

对应的动作方法如下：

public ActionResult CustomVariable(string id)

{

ViewBag.Controller = "Home";

ViewBag.Action = "CustomVariable";

// 检查是否为一个可选片段变量提供了值

**ViewBag.CustomVariable =** **id == null ? "<no value>" : id;**

return View();

}

*注：该动作方法可以通过C#的可选参数的方式实现将片段变量的默认值从路由中分离，详见“****使用可选URL片段强制关注分离****”。*

* **使用可选URL片段强制关注分离**

使用C#的可选参数及路由中的可选片段变量定义动作方法参数的默认值，以实现将片段变量的默认值从应用程序的路由中分离。如：

/// <summary>

/// 使用 C# 的可选参数为动作方法参数定义默认值

/// </summary>

/// <param name="id"></param>

/// <returns></returns>

public ActionResult CustomVariable(**string id = "DefaultId"**)

{

ViewBag.Controller = "Home";

ViewBag.Action = "CustomVariable";

**ViewBag.CustomVariable = id;**

return View();

}

与之对应的可选URL片段路由就是前面介绍过的“可选URL片段”（通过将默认值设置为UrlParameter.Optional来实现）。其效果与这条路由相同： routes.MapRoute("MyRoute", "{controller}/{action}/{id}",new { controller = "Home", action = "Index", **id = "DefaultId"** });。

### 定义可变长路由

通过指定“全匹配（catchall）”片段变量，并以星号（\*）为前缀，便可以实现一个可变长路由。如：

// 定义可变长路由

routes.MapRoute("MyRoute", "*{controller}/{action}/{id}/****{\*catchall}***",

new { controller = "Home", action = "Index", id = UrlParameter.Optional });

上述代码中，斜体的片段变量定义中，前三个分别用于设置controller、action和id的值，后面的加粗斜体（***{\*catchall}***）实现了可变长路由的定义，可以匹配任何URL，无论有多少片段，也不管其值是什么。

由于由catchall捕获的片段是以“片段/片段/片段”的形式表示的，因此，需要对这个字符串进行处理——将其分解成一个个的片段，在处理catchall变量时和处理自定义变量是一样的，只是需要注意该变量的值可能是多个片段连成的一个单一的字符串，就像前面说的那种形式，当然是不需要担心字符串中会存在前导或后导的“/”字符。

### 按命名空间区分控制器优先顺序

如果在不同的命名空间存在同名控制器时，MVC框架将不知道该如何处理，即对象不明确。

假设在示例项目中添加名为AdditionalControllers的文件夹，并添加一个新的Home控制器，如下所示：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace UrlsAndRoutes.AdditionalControllers

{

public class HomeController : Controller

{

public ActionResult Index()

{

ViewBag.Controller = "Additional Controllers - Home";

ViewBag.Action = "Index";

return View("ActionName");

}

}

}

此时，运行程序，将会出现错误，原因就是在不同的命名空间下存在同名控制器，而MVC不会处理这种问题。但是，可以通过将这些命名空间表示成一个字符串数组的方式通知MVC框架要对指定的命名空间优先进行处理。如：

public static void RegisterRoutes(RouteCollection routes)

{

// 指定命名空间解析顺序，MVC 将在处理其他命名空间之前优先处理 UrlsAndRoutes.AdditionalControllers 命名空间

routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{\*catchall}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional }, **new string[] { "UrlsAndRoutes.AdditionalControllers" }**);

}

如果要对一个命名空间的某个控制器给予优先，但又要解析另一个命名空间中的所以其他控制器，就要创建多条路由，原因是添加到一条路由的同一组字符串数组中的命名空间具有同等的优先级。如：

public static void RegisterRoutes(RouteCollection routes)

{

**routes.MapRoute("AddControllerRoute", "{controller}/{action}/{id}/{\*catchall}",**

**new { controller = "Home", action = "Index", id = UrlParameter.Optional },**

**new string[] { "UrlsAndRoutes.AdditionalControllers" });**

**routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{\*catchall}",**

**new { controller = "Home", action = "Index", id = UrlParameter.Optional },**

**new string[] { "UrlsAndRoutes.Controllers" });**

}

当明确请求第一个片段为Home的URL时，会运用第一条路由，并且会以AdditionalControllers文件夹中的Home控制器为目标。其他所有请求，包括未指定第一片段的那些请求，会由Controllers文件夹中的控制器处理。

当然，也可以通知MVC框架只处理指定的命名空间。如果没有匹配的控制器，将不会查找其他命名空间下的控制器。如：

public static void RegisterRoutes(RouteCollection routes)

{

// 禁用备用命名空间

**Route myRoute =** routes.MapRoute("AddControllerRoute", "{controller}/{action}/{id}/{\*catchall}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional },

new string[] { "UrlsAndRoutes.AdditionalControllers" });

**myRoute.DataTokens["UseNamespaceFallback"] = false;**

}

为了禁止搜索其他命名空间的控制器，必须取得这个Route对象，并把DataTokens属性集中的UseNamespaceFallback键值设置为“false”。其效果是，不能满足AdditionalControllers文件夹中Home控制器的请求将失败。

## 约束路由

### 用正则表达式约束路由

/// <summary>

/// 使用正则表达式约束一条路由

/// </summary>

/// <param name="routes"></param>

public static void RegisterRoutes(RouteCollection routes)

{

routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{\*catchall}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional },

**new { controller = "^H.\*" }**,

new string[] { "UrlsAndRoutes.Controllers" });

}

通过把约束作为参数传递给MapRoute方法，可以定义约束。约束被表示成一个匿名类型，该类型的属性对应于想要进行约束的片段变量名。

*注意，在执行时，默认值是在约束被检测之前运用的。也就是说在匹配默认值的情况下，先匹配默认值，然后再进行约束的检查。*

### 将一条路由约束到一组指定的值

如果想限定URL片段只匹配一些指定的值，则可以通过竖线（|）字符来实现，如：

/// <summary>

/// 将一条路由约束到一组指定的值

/// </summary>

/// <param name="routes"></param>

public static void RegisterRoutes(RouteCollection routes)

{

routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{\*catchall}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional },

new { controller = "^H.\*", **action = "^Index$|^About&=$"** },

new string[] { "UrlsAndRoutes.Controllers" });

}

这条约束合起来就是施加于action变量值的约束与施加于controller变量的约束相组合。它只匹配这样的URL：controller变量以“H”字母打头，而且action变量是“Index”或“About”。

### 使用HTTP方法约束路由

可以对路由进行以使他们只匹配指定的HTTP方法进行请求的URL。如：

/// <summary>

/// 基于 HTTP 方法进行路由的约束

/// </summary>

/// <param name="routes"></param>

public static void RegisterRoutes(RouteCollection routes)

{

routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{\*catchall}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional },

new { controller = "^H.\*", action = "Index|About",

**httpMethod = new HttpMethodConstraint("GET")** },

new string[] { "UrlsAndRoutes.Controllers" });

}

可以像下面的方式方便的添加对HTTP其他方法的支持。如：

*httpMethod = new HttpMethodConstraint("GET","POST")*

### 定义自定义约束

可以通过实现IRouteConstraint接口，来定义自己的自定义约束。如：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Routing;

namespace UrlsAndRoutes.Infrastructure

{

public class UserAgentConstratint : IRouteConstraint

{

private string \_requiredUserAgent;

public UserAgentConstratint(string agentParam)

{

\_requiredUserAgent = agentParam;

}

public bool Match(HttpContextBase httpContext

, Route route

, string parameterName

, RouteValueDictionary values

, RouteDirection routeDirection)

{

return httpContext.Request.UserAgent != null && httpContext.Request.UserAgent.Contains(\_requiredUserAgent);

}

}

}

IRouteConstraint接口定义了Match方法，实现它可以用了实现对路由系统只是它的约束是否已得到满足。其中参数提供了这些对象的访问：客户端请求、待评估路由、约束的参数名、从URL提取的片段变量，以及该请求要检查的是输入URL还是输出URL的细节。上述自定义约束的使用方式见下列代码：

/// <summary>

/// 自定义路由约束的使用

/// </summary>

/// <param name="routes"></param>

public static void RegisterRoutes(RouteCollection routes)

{

routes.MapRoute("ChromeRoute", "{\*catchall}",

new { controller = "Home", action = "Index" },

**new**

**{**

**customConstraint = new UserAgentConstratint("Chrome")**

**}**,

new string[] { "UrlsAndRoutes.AdditionalControllers" });

routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{\*catchall}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional },

new string[] { "UrlsAndRoutes.Controllers" });

}

上例的约束的作用是：第一条约束路由时期只匹配来自用户代理字符串含有Chrome的浏览器的请求。如果此路由匹配，那么该请求将被发送给AdditionalControllers文件夹中定义的Home控制器的Index动作方法，而不管所请求的URL具有什么样的结构或内容。第二条路由将匹配其他所有请求，并以Controllers文件夹中的控制器为目标。

最终的效果是Chrome浏览器最终只能访问应用程序的同一个位置。*需要注意的是不建议对应用程序进行限制，以使他只支持一种浏览器，该示例只是提供了一种得到有关字母的办法。同时，该示例只是为了演示自定义路由约束。*

# 对磁盘文件的请求进行路由

MVC应用的请求除了针对控制器和动作的，还有需要对内容进行服务的方法，如对图像、静态HTML文件、JavaScript库等等。请看下面演示示例：

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Static HTML Content</title>

</head>

<body>

This is the static html file(~/content/StaticContent.html)

</body>

</html>

默认情况下，路由系统在评估应用程序的路由之前，会考察一个URL是否匹配一个磁盘文件，也就是说不必为对静态文件请求的URL添加路由。如果将RouteCollection的RouteExistingFiles属性设置为true，则可以实现在检查磁盘文件之前就进行路由评估。如：

routes.RouteExistingFiles = true;*（注：一般需将这句话放在紧靠RegisterRoutes方法的顶部，当然也可以放在其他位置。）*

## 为磁盘文件定义路由

下面，在RouteExistingFiles属性设置为true后，便可以定义对磁盘文件进行响应的URL相匹配的路由，如（下面加粗部分）：

public static void RegisterRoutes(RouteCollection routes)

{

// 下面这句话的作用是启用文件检查前的路由评估

routes.RouteExistingFiles = true;

**routes.MapRoute("DiskFile", "Content/StaticContent.html",**

**new { controller = "Customer", action = "List" });**

routes.MapRoute("ChromeRoute", "{\*catchall}",

new { controller = "Home", action = "Index" },

new

{

customConstraint = new UserAgentConstratint("Chrome")

},

new string[] { "UrlsAndRoutes.AdditionalControllers" });

routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{\*catchall}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional },

new string[] { "UrlsAndRoutes.Controllers" });

}

如果项对磁盘文件请求进行路由，需要慎重考虑，这不仅仅是因为URL模式会像其他情况一样，对这些种类的URL进行匹配，还可能得到一些奇怪的结果，并导致性能下降。所以，非不得已最好不要启用这种做法。

## 绕过路由系统

上面设置的RouteExistingFiles属性使得路由系统更具有包容性，但可以通过设置IgnoreRoute方法可以让路由系统少一些包容性。如：

public static void RegisterRoutes(RouteCollection routes)

{

// 下面这句话的作用是启用文件检查前的路由评估

routes.RouteExistingFiles = true;

**// 使用 IgnoreRoute 方法**

**routes.IgnoreRoute("Content/{filename}.html");**

routes.MapRoute("DiskFile", "Content/StaticContent.html",

new { controller = "Customer", action = "List" });

routes.MapRoute("ChromeRoute", "{\*catchall}",

new { controller = "Home", action = "Index" },

new

{

customConstraint = new UserAgentConstratint("Chrome")

},

new string[] { "UrlsAndRoutes.AdditionalControllers" });

routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{\*catchall}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional },

new string[] { "UrlsAndRoutes.Controllers" });

}

可以用{filename}这样的片段变量来匹配一组URL。此时，该URL模式将匹配任何两片段URL，其第一片段是content，第二片段内容具有.html扩展名。

IgnoreRoute方法在RouteCollection中创建了一个条目，其路由处理器是StopRoutingHandler类的一个实例，而不是MvcRouteHandler。如果传递给IgnoreRoute方法的URL模式得到匹配，那么将不在评估后续的路由，就像有一条唱歌路由已得到匹配时一样。（IgnoreRoute方法的作用是：忽略给定可用路由列表的指定 URL 路由）

# 高级路由特性

## 在视图中生成输出URL

使用路由系统来生成输出URL，能够确保URL方案动态地产生URL。

在视图中生成输出URL的最简单做法是在视图中调用Html.ActionLink辅助器方法，如：

<div>@Html.ActionLink("This is an outgoing URL", "CustomVariable")</div>

注意，ActionLink方法生成的HTML是基于当前路由配置的。如果路由方案是下面这样，

public static void RegisterRoutes(RouteCollection routes)

{

routes.MapRoute("MyRoute", "{controller}/{action}/{id}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional });

}

则将得到这样的：HTML：<a href=”/Home/CustomVarible”>This is an outgoing URL</a>

但是，如果将路由方案改为下面这种方式：

public static void RegisterRoutes(RouteCollection routes)

{

**routes.MapRoute("NewRoute", "App/Do{action}",**

**new { controller = "Home" });**

routes.MapRoute("MyRoute", "{controller}/{action}/{id}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional });

}

则将会得到：<a href=”/App/DoCustomVarible”>This is an outgoing URL</a>

因此，可以看出使用路由系统生成输出URL的方式可以方便的实现自动对路由配置的更改进行响应。这也方便了维护——通过修改路由方案，在视图中的输出链接会自动地反映出这种修改。

知识点：

路由系统是按照路由被添加的顺序来处理路由的，路由会被添加到传递给RegisterRoutes方法的RuteCollection对象中。每一条路由再被检测是否是一个匹配时，都需要满足三个条件：

* URL模式中定义的每一个片段变量都必须有一个可用的值。为了找到每个片段变量的值，路由系统首先查看已经提供的值（采用匿名类型的属性），然后查看当前请求的变量值，最后查看该路由中定义的默认值。
* 在给片段变量提供的值中，应当没有违背这条路由所定义的只默认变量的值。只默认变量是为其提供了默认值，但未在URL模式中出现的变量。如在匹配这条路由（routes.MapRoute("MyRoute", "{controller}/{action}",new { myVar = "True" });）时就要小心，不要给myVar（只默认变量：为其提供了默认值“true”，但未在URL模式中出现——如果未在URL模式中为其提供值）提供值，或要确保所提供的值与这个默认值是匹配的。
* 所有片段变量的值均必须满足路由约束。一定注意，路由系统不会试图查找最佳匹配路由，它只会找出**最先**匹配者，然后用这条路由来生成URL，任何后续的路由都会被忽略。所以，应该首先定义最具体的路由。

为各个片段参数所选择参数值将在其中被替换，即在Html.ActionLink给片段提供的值，会替换掉生成URL中的相关默认值——这些默认值是在路由配置中设定的。同时，将忽略**尾部**的默认值序列。如果明确地提供一些参数，但它们与片段参数或默认参数不符，那么方法将把这些参数以“名字/值”对的查询字符串形式进行追加。

## 以其他控制器为目标

ActionLink方法默认以当前控制器为目标创建输出URL，如果需要创建一个以不同控制器为目标的输出URL，则可以使用它的一个重载版本，它允许指定控制器名称。需要注意的是，路由系统不会对动作方法和控制器的值进行检验，不要指定不存在的目标。

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<title>ActionName</title>

</head>

<body>

<div>The controller is:@ViewBag.Controller</div>

<div>The action is:@ViewBag.Action</div>

<div>@Html.ActionLink("This is an outgoing URL", "CustomVariable")</div>

**<div>**

**<!--通过指定控制器来实现一个以不同控制器为目标的输出 URL -->**

**@Html.ActionLink("This targets another controller", "Index", "Admin")**

**</div>**

</body>

</html>

上述加粗部分使用了ActionLink的一个重载版本，其指定了目标控制器及动作，当视图渲染时，最终会生成以下HTML结果：<a href=”/Admin”>This targets another controller</a>。由于Index为默认动作方法，因此，可以省略默认的动作片段，这也是路由系统所允许的。

## 传递额外的值

可以使用一个匿名类型为一些片段变量传值，在这个匿名类型中一其属性表示片段。如下代码所示：

<div>

<!--使用匿名类型给片段变量提供值-->

@Html.ActionLink("This is an outgoing URL", "CustomVariable", new { id = "Hello" })

</div>

上面这段代码将会产生这样的HTML：

<a href="/App/**DoCustomVariable?id=Hello**">This is an outgoing URL</a>

根据前面的路由模式设置，为id提供的值被作为查询字符串添加到了URL，其原因是为了实现与路由设置的URL模式相匹配。前面的路由设置如下：

**routes.MapRoute("NewRoute", "App/Do{action}",new { controller = "Home" });**

因此，如果路由设置中只有一个与之匹配的路由，那将会得到不同的结果——复制给id属性的值被作为URL片段包括在URL中。即如果只有一条这样的与之匹配的路由：

routes.MapRoute("MyRoute", "{controller}/{action}/{id}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional });

HTML结果：<a href="/App/**CustomVariable/Hello**">This is an outgoing URL</a>

变化之处已用粗体标出。

知识点：

片段变量的重用

在为一条路由URL模式中的每一个片段变量查找值的过程中，路由系统将考查当前请求的值，就如前面谈到输出URL时所说的，路由系统为了找到每个片段变量的值，会首先查看已经提供的值（采用匿名类型的属性），然后查看当前请求的变量值，最后查看该路由定义的默认值。

假设只有一条路由，如下：

routes.MapRoute("MyRoute", "{controller}/{action}/{color}/{page}");

现在假设一个用户的当前URL为/Catalog/List/Purple/123，并渲染一个这样的链接：

@Html.Action(“Click me”,”List”,”Catalog”,new {page=789},null)

虽然，未提供color片段变量的值，同时也未定义默认值，但路由系统将会根据已经的定义的路由进行匹配，并生成这样的HTML：

<a href=”/Catalog/List/Purple/789”>Click me</a>

原因是，路由系统查找了当前请求的变量值——它重用了输入URL的片段变量值，color的值最终被赋成了Purple。

需要注意的是，在开发的过程中不要依赖路由系统的这种行为，因为，**路由系统将只对某些片段变量使用重用值，这些片段变量在URL模式中的出现早于提供给Html.ActionLink方法的参数**。如果下面这样的链接将不会被匹配：

@Html.Action(“Click me”,”List”,”Catalog”,new {color=”Aqua”},null)

这里未提供值的片段page在URL模式中的出现晚于已提供值的color片段，因此不能使用重用值，于是这条路由将不被匹配。

因此，这里强烈建议不要依赖与路由系统的这种行为，并应为URL模式中的所有片段变量都提供值。

## 指定HTML标签属性

通过提供一个匿名类型，可以为元素设置标签属性，该匿名类型的属性与所需要的标签属性相对应。如下面示例中对id标签属性的设置，并为其所在的a元素赋值了一个CSS的class：

<div>

<!--生成一个带有标签属性的锚点元素。class 之前使用@字符作为前缀，这是一条 C# 语言特性，它让用户能够用保留关键字作为 class 成员的名字（即以 class 作为该匿名类型的一个成员）-->

@Html.ActionLink("This is an outgoing URL", "Index", "Home", null, new { **id = "myAchorID"**, **@class = "myCSSClass"** })

</div>

上述这段语句将被渲染成这样的HTML：

<a **class=”myCSSClass”** href=”/” **id=”myAnchorID”**>This is an outgoing URL</a>

## 生成链接中的全限定URL

前面介绍的生成的链接都是相对URL，但我们还可以使用ActionLink辅助器方法生成全限定的URL。如：

<div>

<!--生成权限定 URL -->

@Html.ActionLink("This is an outgoing URL", "Index", "Home", "https", "myserver.mydomain.com", " myFragmentName", new { id = "myId" }

, new { id = "myAchorID", @class = "myCSSClass" })

</div>

最终会生成这样的HTML：

<a class=”myCSSClass” href=”https://myserver.mydomain.com/Home/Index/MyId#myFragmentName” id=”myAnchorID”>This is an outgoing URL</a>

***建议：在实际项目中尽可能使用相对URL，权限定URL会形成对应用程序的基础架构方式产生依赖性。***

## 生成URL（而不是链接）

使用Html.ActionLink辅助器方法可以生成完整的HTML的<a>元素，但如果需要，还可以使用Url.Action方法只生成URL，而不产生HTML元素。如下代码所示：

<div>

This is a URL:

@Url.Action("Index", "Home", new { id = "MyId" })

</div>

除了只生成URL外，它与Html.ActionLink方法的工作方式相同。

## 在动作方法中生成输出URL

有时我们需要在动作方法中输出URL，如果仅需生成URL，可以采取与在视图中同样的辅助器方法，如下：

/// <summary>

/// 在动作方法中生成输出 URL

/// </summary>

/// <returns></returns>

public ViewResult MyActionMethod()

{

string myActionUrl = Url.Action("Index", new { id = "MyID" });

string myRouteUrl = Url.RouteUrl(new { controller = "Home", action = "Index" });

// ... 用此 URL 做些事情...

return View();

}

对于上面这段代码，myActionUrl 的值将是“/Home/Index/MyID”，而myRouteUrl 将为“/”。

通常，需要将客户端浏览器重定向到另一个URL。使用RedirectToAction方法可以重定向到另一个动作，该方法指示MVC框架把一条重定向指令发布给一个URL，由这个URL调用指定的动作。如：

/// <summary>

/// 重定向到另一个动作

/// </summary>

/// <returns></returns>

public RedirectToRouteResult MyActionMethod()

{

return RedirectToAction ("Index");

}

该方法的一些重载版本可以给生成的URL中片段变量指定控制器和值。

如果需要用一个URL发送一个重定向，而这个URL只是根据对象的属性生成，则还可以使用RedirectToRoute方法，如：

/// <summary>

/// 重定向到根据匿名类型中的属性生成的URL

/// </summary>

/// <returns></returns>

public RedirectToRouteResult MyActionMethod()

{

return RedirectToRoute(new { controller = "Home", action = "Home", id = "MyID" });

}

## 根据指定路由生成URL

下面介绍一下如何通过指定的路由生成URL或链接，当然在实际的项目中根据指定路由生成URL是不可取的，主要是因为在视图或动作方法中使用指定的路由名生成URL，等同于在建立视图或动作方法与路由自己的依赖性，而这与MVC思想是不一致的。好了，这里咱们只需要通过这种方式来理解这一过程的实现原理：

public static void RegisterRoutes(RouteCollection routes)

{

routes.MapRoute("MyRoute", "{controller}/{action}");

routes.MapRoute("MyOtherRoute", "App/{action}", new { controller = "Home" });

}

上面为两条路由指定了名称——MyRoute和MyOtherRoute。对路由命名的原因主要有两个：

* 作为路由目的的一种说明
* 便于选择特点的路由，用以生成输出URL

由于前面路由配置已给定了路由顺序，所以，当使用下面的Html.ActionLink方法生成的链接将总是由MyRoute生成，如：

@Html.ActionLink(“Click me”,”Index”,”Customer”)

得到的结果：

<a href=”/Customer/Index”>Click me</a>

可以用Html.RouteLink方法（该方法可以指定路由生成链接）覆盖这种默认路由匹配行为，这种方法需要指定一条路由（如下面粗体的“MyOtherRoute”，而斜体的粗体字“Customer”被重写了），如：

@Html.RouteLink(“Click me”,”**MyOtherRoute**”,”Index”,”***Customer***”)

结果：

<a length=”8” href=”/App/Index?Length=5”>Click me</a>

说明：就像开头说明的这种以指定路由生成URL不是明智之举，但是如果我们需要明确路由的意图时最好还是改用注释来提示。

# 定制路由系统

路由系统是灵活可配置的，当然还可以通过下面这两种方式定制路由系统，来满足其他需求。

1. 通过创建自定义的RouteBase实现；
2. 通过创建自定义路由处理程序实现。

## 创建自定义的RouteBase实现

创建自定义的RouteBase实现，需要实现一个RouteBase的派生类，而这需要实现以下两个方法：

* GetRouteData(HttpContextBase httpContext)：这是入站URL进行匹配的工作机制。框架依次对RouteTable.Routes的每个条目调用这个方法，直到其中之一返回一个非空值。
* GetVirtualPath(RequestContext requestContext,RouteValueDictionary values)：这是出站URL生成的工作机制。框架依次对RouteTable.Routes的每一个条目调用这个方法，直到其中之一返回一个非空值。

为了演示这种自定义方式，这里创建了一个RouteBase的派生类。我们假设这样的一个需求环境：需要把一个现有的应用程序迁移到MVC框架，但不论出于什么原因，我们需要兼容之前的URL，那就可以通过这种方式来实现，当然可以通过规则的路由系统来处理——这里不对这种方式进行讨论。

首先，创建一个处理旧式路由请求的控制器，将其命名为：LegacyController，如：

using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers

{

/// <summary>

/// 用以处理旧式 URL 请求的控制器

/// </summary>

public class LegacyController : Controller

{

public ActionResult GetLegacyURL(string legacyURL)

{

// 应用程序迁移到 MVC 之前，请求是针对文件的，因此，实际上是需要在这里处理被请求的文件。但这里

// 只简单说明一下自定义 RouteBase 的实现原理，所以，此处仅在视图中显示这个 URL。

return View((object)legacyURL);

}

}

}

上面代码对View方法中的参数做了转换，如果不转换，则C#编译器会误认为要将参数作为要指定渲染的视图的名称的字符串（View方法的一个重载版本的实现）。下面是这个动作方法的视图GetLegacyURL.cshtml：

@model string

@{

ViewBag.Title = "GetLegacyURL";

Layout = null;

}

<h2>GetLegacyURL</h2>

The URL requested was:@Model

**1、对输入URL进行路由**

在Infrastructure文件夹中创建一个LegacyRoute类，其内容如下：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace UrlsAndRoutes.Infrastructure

{

public class LegacyRoute : RouteBase

{

private string[] urls;

public LegacyRoute(params string[] targetUrls)

{

}

public override RouteData GetRouteData(HttpContextBase httpContext)

{

RouteData result = null;

string requestedURL = httpContext.Request.AppRelativeCurrentExecutionFilePath;

if (urls.Contains(requestedURL, StringComparer.OrdinalIgnoreCase))

{

result = new RouteData(this, new MvcRouteHandler());

result.Values.Add("controller", "Legacy");

result.Values.Add("action", "GetLegacyURL");

result.Values.Add("legacyURL", "requestedURL");

}

return result;

}

public override VirtualPathData GetVirtualPath(RequestContext requestContext, RouteValueDictionary values)

{

return null;

}

}

}

注册一条路由，以使其使用新建的这个RouteBase派生类：

public static void RegisterRoutes(RouteCollection routes)

{

// 注册自定义的 RouteBase 实现

routes.Add(new LegacyRoute("~/articles/Windows\_3.1\_Overview.html", "~/old/.NET\_1.0\_Class\_Library"));

}

**2、生成输出URL**

在LegacyRoute中实现GetVirtualPath方法以使其能够支持输出URL的生成。如：

public override VirtualPathData GetVirtualPath(RequestContext requestContext, RouteValueDictionary values)

{

VirtualPathData result = null;

if (values.ContainsKey("legactURL") && urls.Contains((string)values["legacyURL"], StringComparer.OrdinalIgnoreCase))

{

// 如果存在一个匹配，将会创建一个 VirtualPathData 对象，在其中传递一个对当前对象的引用和出站 URL。由于路由系统已经预先将

// 字符“/”附加到了这个URL，因此，必须从生成的 URL 上删除这个前导字符。

result = new VirtualPathData(this, new UrlHelper(requestContext).Content((string)values["legacyURL"]).Substring(1));

}

return null;

}

在ActionName.cshtml视图中添加下面这段代码，以使其能礼仪自定义路由生成输出URL：

<div>

@\* 经由自定义路由生成一个输出 URL \*@

This is a URL:

@Html.ActionLink("Click me", "GetLegacyURL", new { legacyURL = "~/articles/Windows\_3.1\_Overview.html" })

</div>

上面代码将产生一个这样的a元素：

<a href=”/articles/Windows\_3.1\_Overview.html”>Click me</a>

用legacyURL属性创建的匿名类型被转换到了含有同名键的RouteValueDictionary类中。

## 创建自定义路由处理程序

路由已经依赖这个MvcRouteHandler了，因为MvcRouteHandler把路由系统连接到了MVC框架。但通过实现IRouteHandler接口，路由系统仍允许自定义自己的路由处理程序，如下面的示例：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Routing;

namespace UrlsAndRoutes.Infrastructure

{

public class CustomRouteHandler : IRouteHandler

{

public IHttpHandler GetHttpHandler(RequestContext requestContext)

{

return new CustomHttpHandler();

}

}

public class CustomHttpHandler : IHttpHandler

{

public bool IsReusable

{

get { return false; }

}

public void ProcessRequest(HttpContext context)

{

context.Response.Write("Hello");

}

}

}

IRouteHandler接口的目的是提供生成IHttpHandler接口的实现，且由它负责对请求进行处理。在该接口的MVC实现中，主要负责这几项工作：查找控制器、调用动作方法、渲染视图，并将结果写入到响应中。当然，这里的实现要简单的多，此处仅将单词“Hello”写到客户端，且只是文本形式。要想得到最终效果，需要在RouteConfig.cs文件中注册这个自定义处理程序：

public static void RegisterRoutes(RouteCollection routes)

{

// 注册自定义路由处理程序

routes.Add(new Route("SayHello", new CustomRouteHandler()));

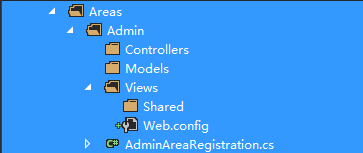
}

# 使用区域

MVC框架支持将Web应用程序组织成一些区域（Area），每个区域代表应用程序的一个功能端，如管理、结算、客户支持等等。这使得代码的管理很有用，尤其是大型项目，如果对所有控制器、视图和模型只使用一组文件夹，那将会是很难于管理的。

## 创建区域

可以直接对项目右键，选择“添加”->“区域”进行添加。还可以在当前的区域中创建其他区域。在刚刚的操作之后，项目中将会出现如下这样的区域文件夹结构：



通过Areas/Admin文件夹，可以看出这是一个小型的MVC项目。其中有“Controllers”、“Models”和“Views”的文件夹。前两个是空的，但“Views”文件夹含有一个“Shared”文件夹和一个Web.config视图引擎配置文件（这里暂不对视图引擎进行讨论）。

另外，这里还多了一个AdminAreaRegistration.cs文件，如：

using System.Web.Mvc;

namespace UrlsAndRoutes.Areas.Admin

{

public class AdminAreaRegistration : AreaRegistration

{

public override string AreaName

{

get

{

return "Admin";

}

}

public override void RegisterArea(AreaRegistrationContext context)

{

context.MapRoute(

"Admin\_default",

"Admin/{controller}/{action}/{id}",

new { action = "Index", id = UrlParameter.Optional }

);

}

}

}

从清单中可以看出，该类中的RegisterArea方法注册了一个URL模式为Admin/{controller}/{action}/{id}的路由。当然，也可以在该方法中定义该区域专用的其他路由。

注意：如果要给路由赋名，必须确保这些名称在整个应用程序而不仅仅是某一区域中是唯一的。

由于在Global.asax的Application\_Start方法中已经对路由的注册进行过处理，因此，不需要在开发的过程中采取其他措施来确保该注册方法会被调用：

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

**AreaRegistration.RegisterAllAreas();**

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

}

上面代码中对静态方法AreaRegistration.RegisterAllAreas的调用，会导致MVC框架对应用程序的所有类进行遍历，找出派生于AreaRegistration的所有类，并调用这些类上的RegisterArea方法。

注意：不用修改Application\_Start方法中与路由相关的语句顺序。如果在AreaRegistration.RegisterAllAreas之前调用RegisterRoutes，那么会在区域路由之前定义路由。由于路由系统是按顺序评估的，这意味着对区域控制器的请求有可能会用不正确的路由进行匹配。

注：AreaRegistrationContext类中的MapRoute方法会自动把注册的路由限制到包含该区域控制器的命名空间。也就是说，当某区域创建控制器时，必须把它放在其默认的命名空间中；否则，路由系统将无法找到它。

## 填充区域