# 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。*