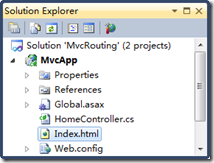
ASP.NET MVC的路由系统通过对HTTP请求的解析得到表示Controller、Action和其他相关的数据，并以此为依据激活Controller对象，调用相应的Action方法，并将方法返回的ActionResult写入HTTP回复中。为了更好的演示其实现原理，我创建一个简单的ASP.NET Web应用来模拟ASP.NET MVC的路由机制。这个例子中的相关组件基本上就是根据ASP.NET MVC的同名组件设计的，只是我将它们进行了最大限度的简化，因为我们只需要用它来演示大致的实现原理而已。[源代码从[这里](http://files.cnblogs.com/artech/MvcRouting.rar)下载]

目录：   
一、一个通过查询字符串表示Controller和Action的“MVC”程序   
二、通过Route解析HTTP请求获得路由信息   
三、在Global.asax中注册Route   
四、Route的执行   
五、通过MvcHandler处理请求   
六、将ActionResult写入Http回复   
七、实例的配置和定义

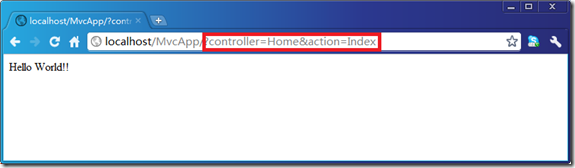
**一、一个通过查询字符串表示Controller和Action的“MVC”程序**

[[](http://images.cnblogs.com/cnblogs_com/artech/201112/201112051951565019.png)](http://images.cnblogs.com/cnblogs_com/artech/201112/201112051951565019.png)

如右图所示，我们的Web应用非常简单。HomeController.cs为定义Controller类型的文件，而Index.html表示HomeController中名称为Index的Action对应的View。我们按照ASP.NET MVC的原理，通过解析请求URL得到Controller和Action的名称。如果Controller为Home，则激活HomeController，如果当前的Action为Index，则将Index.html这个静态文件的内容作为HTTP回复返回。

我不想定义复杂的解析Controller和Action的逻辑，再这里我直接通过请求URL相应的查询字符串controler和action表示Controller和Action的名称。也就是说如果通过浏览器访问地址<http://localhost/mvcapp/?controller=Home&action=Index> 可以访问到Index.html中的内容（注：我们并没有将Index.html作为站点的默认页面）。

注：在本地站点中可以用http://localhost:8009/?controller=Home&action=Index

[](http://images.cnblogs.com/cnblogs_com/artech/201112/201112051952034849.png)

接下来我简单的介绍一下是哪些组建促使这个简单的ASP.NET Web应用能够按照MVC的模式来执行。为了使你能够在真正的ASP.NET MVC找到匹配的组件，我们采用了相同的接口和类型名称。

**二、通过Route解析HTTP请求获得路由信息**

我定义了如下一个RouteData类型表示解析HTTP请求得到的Controller和Action等信息。Assemblies和Namespaces表示需要引入的命名空间和程序集，这是因为URL中只能解析出Controller的类型名称，需要相应的命名空间采用得到它的类型全名。如果对应的程序集不曾加载，还需要加载相应的程序集。

1: public class RouteData

2: {

3: public string Controller { get; set; }

4: public string Action { get; set; }

5: public IList<string> Assemblies { get; private set; }

6: public IList<string> Namespaces { get; private set; }

7: public IRouteHandler RouteHandler { get; set; }

8:

9: public RouteData(string controller, string action, IRouteHandler routeHandler)

10: {

11: this.Controller = controller;

12: this.Action = action;

13: this.RouteHandler = routeHandler;

14: this.Namespaces = RouteTable.Namespaces;

15: this.Assemblies = RouteTable.Assemblies;

16: }

17: }

真正实现对HTTP请求进行解析并得到RouteData的Route继承自基类RouteBase。我们还定义个了一个表示Route集合的RouteCollection类型，它的GetRouteData方法对集合的所有Route对象进行遍历，并调用其GetRouteData方法。如果得到的RouteData不为空，则返回之。

1: public abstract class RouteBase

2: {

3: public abstract RouteData GetRouteData(HttpContextBase httpContext);

4: }

5:

6: public class RouteCollection: Collection<RouteBase>

7: {

8: public RouteData GetRouteData(HttpContextBase httpContext)

9: {

10: foreach (RouteBase route in this)

11: {

12: var routeData = route.GetRouteData(httpContext);

13: if (null != routeData)

14: {

15: return routeData;

16: }

17: }

18: return null;

19: }

20: }

和ASP.NET MVC一样，我们定义了如下一个RouteTable对象，其静态属性正是一个RouteCollection对象。两个静态属性Namespaces和Assemblies为命名空间和程序集名称的全局维护。

1: public class RouteTable

2: {

3: static RouteTable()

4: {

5: Routes = new RouteCollection();

6: Namespaces = new List<string>();

7: Assemblies = new List<string>();

8: }

9: public static RouteCollection Routes { get; private set; }

10: public static IList<string> Namespaces { get; private set; }

11: public static IList<string> Assemblies { get; private set; }

12: }

而我们实例中完成基于查询字符串的Controller和Action解析的QueryStringRoute对应如下。在GetRouteData方法中，除了根据查询字符解析并初始化Controller和Action名称之外，还将RouteHandler指定为MvcRouteHandler。而MvcRouteHandler得GetHttpHandler方法直接返回的是根据RequestContext创建的MvcHandler对象。

1: public class QueryStringRoute : RouteBase

2: {

3: public override RouteData GetRouteData(HttpContextBase httpContext)

4: {

5: if (httpContext.Request.QueryString.AllKeys.Contains("controller") &&

6: httpContext.Request.QueryString.AllKeys.Contains("controller") )

7: {

8: string controller = httpContext.Request.QueryString["controller"];

9: string action = httpContext.Request.QueryString["action"];

10: IRouteHandler routeHandler = new MvcRouteHandler();

11: return new RouteData(controller, action, routeHandler);

12: }

13: return null;

14: }

15: }

16:

17: public class MvcRouteHandler: IRouteHandler

18: {

19: public IHttpHandler GetHttpHandler(RequestContext requestContext)

20: {

21: return new MvcHandler(requestContext);

22: }

23: }

**三、在Global.asax中注册Route**

通过上面定义的RouteTable类型，我们在Global.asax中按照如下的方式在应用启动的时候QueryStringRoute对象添加到RouteTable的静态属性Routes表示的Route列表中。同时为需要的命名空间和程序集名称进行初始化，以辅助后续步骤中对Controller的创建。

1: public class Global : System.Web.HttpApplication

2: {

3: protected void Application\_Start(object sender, EventArgs e)

4: {

5: RouteTable.Routes.Add(new QueryStringRoute());

6: RouteTable.Assemblies.Add("MvcApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null");

7: RouteTable.Namespaces.Add("Artech.MvcApp");

8: }

9: }

**四、Route的执行**

通过RouteTable的Routes属性表示的Route列表对请求的解析和路由信息的获取是通过自定义的HttpModule来实现的，它的类型为UrlRoutingModule。如下面的代码片断所示，UrlRoutingModule注册了HttpApplication的PostResolveRequestCache事件，并在该事件触发的时候调用Route列表的GetRouteData方法，并根据得到RouteData创建RequestContext。最后通过RouteData的RouteHandler得到真正用于处理该请求的HttpHandler对象，并对其进行映射。这意味着后续将会采用这个映射的HttpHandler进行请求的处理。

1: public class UrlRoutingModule: IHttpModule

2: {

3: public void Dispose() { }

4: public void Init(HttpApplication context)

5: {

6: context.PostResolveRequestCache += (sender, args) =>

7: {

8: HttpContextWrapper contextWrapper = new HttpContextWrapper(context.Context);

9: HttpContextBase httpContext = (HttpContextBase)contextWrapper;

10: RouteData routeData = RouteTable.Routes.GetRouteData(httpContext);

11: if (null == routeData)

12: {

13: return;

14: }

15: RequestContext requestContext = new RequestContext { HttpContext = httpContext, RouteData = routeData };

16: httpContext.RemapHandler(routeData.RouteHandler.GetHttpHandler(requestContext));

17: };

18: }

19: }

UrlRoutingModule在mvcapp的web.config中也需要添加：

<system.webServer>

<modules>

<add name="UrlRoutingModule" type="Artech.MvcRouting.UrlRoutingModule, Artech.MvcRouting"/>

</modules>

</system.webServer>

**五、通过MvcHandler处理请求**

在UrlRoutingModule映射的实际上是具有如下定义的MvcHandler，它具有一个RequestContext属性通过构造函数进行初始化。在ASP.NET MVC中，真正的请求处理体现在根据路由信息创建Controller，并执行相应的Action方法。这两个步骤体现的ProcessRequest方法中。

1: public class MvcHandler: IHttpHandler

2: {

3: public RequestContext RequestContext{get; private set;}

4: public IControllerFactory ControllerFactory

5: {

6: get { return ControllerBuilder.Current.GetControllerFactory(); }

7: }

8: public MvcHandler(RequestContext requestContext)

9: {

10: this.RequestContext = requestContext;

11: }

12: public bool IsReusable

13: {

14: get { return false; }

15: }

16: public void ProcessRequest(HttpContext context)

17: {

18: RouteData routeData = this.RequestContext.RouteData;

19: var controller = this.ControllerFactory.CreateController(this.RequestContext, routeData.Controller);

20: controller.Execute(this.RequestContext);

21: }

22: }

Controller实现了具有如下定义的接口IController，所有Action方法都通过Execute方法执行，该方法的参数的表示当前请求上下文的RequestContext对象。IController通过相应的Controller工厂创建，下面的代码同时也定义了Controller工厂接口的定义。

1: public interface IController

2: {

3: void Execute(RequestContext requestContext);

4: }

5: public interface IControllerFactory

6: {

7: IController CreateController(RequestContext requestContext, string controllerName);

8: }

我们定义了如下一个简单名称为DefaultController，它的Execute方法定义很简单：通过包含在RequestContext的RouteData得到当前的Action，并将它作为方法名得到相应的MethodInfo对象，滨个通过反射调用它得到一个ActionResult对象，最后执行ActionResult的ExecuteResult方法。该方法的参数是基于RequestContext创建的另一个上下文ControllerContext。

1: public class DefaultController : IController

2: {

3: public void Execute(RequestContext requestContext)

4: {

5: string action = requestContext.RouteData.Action;

6: MethodInfo method = this.GetType().GetMethod(action);

7: ActionResult result = (ActionResult)method.Invoke(this, null);

8: ControllerContext controllerContext = new ControllerContext

9: {

10: RequestContext = requestContext

11: };

12: result.ExecuteResult(controllerContext);

13: }

14: }

我们定义了具有如下定义的Controller工厂类DefaultControllerFactory。创建Controller的逻辑也不复杂：通过RouteData表示的Controller名称得到相应的Controller类型，通过反射创建Controller对象。由于RouteData中只包含Controller的名称，所以需要通过命名空间和程序集的辅助才能解析出真正的类型。

1: class DefaultControllerFactory : IControllerFactory

2: {

3: public IController CreateController(RequestContext requestContext, string controllerName)

4: {

5: RouteData routeData = requestContext.RouteData;

6: string controllerType = string.Format("{0}Controller", controllerName);

7: IController controller;

8: controller = this.CreateControler(controllerType);

9: if (null != controller)

10: {

11: return controller;

12: }

13: foreach (string assembly in routeData.Assemblies)

14: {

15: controller = this.CreateControler(controllerType, assembly);

16: if (null != controller)

17: {

18: return controller;

19: }

20:

21: foreach (string ns in routeData.Namespaces)

22: {

23: controllerType = string.Format("{0}.{1}Controller", ns, controllerName);

24: controller = this.CreateControler(controllerType, assembly);

25: if (null != controller)

26: {

27: return controller;

28: }

29: }

30: }

31:

32: throw new InvalidOperationException("Cannot locate the controller");

33: }

34: private IController CreateControler(string controllerType, string assembly = null)

35: {

36: Type type = null;

37: if (null == assembly)

38: {

39: type = Type.GetType(controllerType);

40: }

41: else

42: {

43: type = Assembly.Load(assembly).GetType(controllerType);

44: }

45: if (null == type)

46: {

47: return null;

48: }

49: return Activator.CreateInstance(type) as IController;

50: }

51: }

**六、将ActionResult写入Http回复**

Controller的Action方法的返回值为具有如下定义的ActionResult类型，通过ExecuteResult方法将相应的执行结果写入HTTP回复中。我定义了如下一个StaticViewResult，它根据RouteData中的Action信息找到匹配的.html静态文件，并将文件的内容写入HttpResponse。

1: public abstract class ActionResult

2: {

3: public abstract void ExecuteResult(ControllerContext context);

4: }

5:

6: public class StaticViewResult: ActionResult

7: {

8: public override void ExecuteResult(ControllerContext context)

9: {

10: context.RequestContext.HttpContext.Response.WriteFile(context.RequestContext.RouteData.Action + ".html");

11: }

12: }

是否我可以把ExecuteResult方法理解成ActionResult执行view渲染的关键方法。

**七、实例的配置和定义**

在我们的实例中定义的HomeController定义如下，在表示Action的Index方法中，直接返回一个StaticViewResult对象。

1: public class HomeController : DefaultController

2: {

3: public ActionResult Index()

4: {

5: return new StaticViewResult();

6: }

7: }

然后在配置中进行了针对UrlRoutingModule的注册，仅此而已。

1: <configuration>

2: <system.webServer>

3: <modules>

4: <add name="UrlRoutingModule" type="Artech.MvcRouting.UrlRoutingModule, Artech.MvcRouting"/>

5: </modules>

6: </system.webServer>

7: </configuration>