# 控制器可扩展性

这部分主要研究一下如何配置控制器工厂和动作调用器，以便了解如何控制它们的行为；另外，也看看如何能够替代这些组件，并使用自己的逻辑。

下图是一个请求经过各组件时的基本流程，它演示了请求的处理过程：



调用一个动作方法

（一个请求在调用动作方法时的处理管道）

## 准备示例项目

为了解控制器的可扩展性，需要创建一个新项目。

项目名称：ControllerExtensibility

项目模板：Empty

基本结构：

* Models文件夹：Result.cs类——定义Result Model对象
* Views/ Shared文件夹：Result.cshtml视图——一个渲染控制器类中所有动作方法的视图（以Result类作为其模型，简单地显示ControllerName和ActionName）
* 控制器：Product、Customer

下面看看上述基本结构中各项的代码示例：

1、Result Model对象：

namespace ControllerExtensibility.Models

{

public class Result

{

public string ControllerName { get; set; }

public string ActionName { get; set; }

}

}

2、Result.cshtml视图：

@model ControllerExtensibility.Models.Result

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<title>Result</title>

</head>

<body>

<div>Controller: @Model.ControllerName</div>

<div>Action: @Model.ActionName</div>

</body>

</html>

3、Product控制器：

using ControllerExtensibility.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Controllers

{

public class ProductController : Controller

{

public ViewResult Index()

{

return View("Result", new Result { ControllerName = "Product", ActionName = "Index" });

}

public ViewResult List()

{

return View("Result", new Result { ControllerName = "Product", ActionName = "List" });

}

}

}

4、Customer控制器

using ControllerExtensibility.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Controllers

{

public class CustomerController : Controller

{

public ViewResult Index()

{

return View("Result", new Result { ControllerName = "Customer", ActionName = "Index" });

}

public ViewResult List()

{

return View("Result", new Result { ControllerName = "Customer", ActionName = "List" });

}

}

}

主要目的：研究MVC框架提供的对控制器和动作的管理进行定制的方式。

## 创建自定义控制器工厂

虽然实际项目中建议通过对内建的控制器工厂进行扩展，但现在需要理解其工作原理，所以，我们通过创建一个自定义控制器工厂进行探究。

控制器工厂是由IControllerFactory接口定义的，下面是接口的定义：

using System;

using System.Web.Routing;

using System.Web.SessionState;

namespace System.Web.Mvc

{

public interface IControllerFactory

{

IController CreateController(RequestContext requestContext, string controllerName);

SessionStateBehavior GetControllerSessionBehavior(RequestContext requestContext, string controllerName);

void ReleaseController(IController controller);

}

}

现在创建一个Infrastructure文件夹，并在其中建立一个简单的控制器工厂CustomControllerFactory：

using ControllerExtensibility.Controllers;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

using System.Web.SessionState;

namespace ControllerExtensibility.Infrastructure

{

public class CustomControllerFactory : IControllerFactory

{

public IController CreateController(RequestContext requestContext, string controllerName)

{

Type targetType = null;

switch (controllerName)

{

case "Product":

targetType = typeof(ProductController);

break;

case "Customer":

targetType = typeof(CustomerController);

break;

default:

requestContext.RouteData.Values["controller"] = "Product";

targetType = typeof(ProductController);

break;

}

return targetType == null ? null : (IController)DependencyResolver.Current.GetService(targetType);

}

public SessionStateBehavior GetControllerSessionBehavior(RequestContext requestContext, string controllerName)

{

return SessionStateBehavior.Default;

}

public void ReleaseController(IController controller)

{

IDisposable disposable = controller as IDisposable;

if (controller != null)

{

disposable.Dispose();

}

}

}

}

上面代码中最重要的是CreateController方法，在MVC需要控制器对请求进行服务时调用。该方法的RequestContext类型参数能提供请求的细节内容；另一个字符串参数则提供的是controller值，该值是从URL那里得到的。

RequestContext属性：

* HttpContext：类型为HttpContextBase，用来提供HTTP请求的信息
* RouteData：类型为RouteData，提供与请求匹配的路由信息

如果在开发的时候像上面那样自行创建控制器工厂将是一件很麻烦的事——在Web程序中查找控制器类并对它们实例化是复杂的，如需要能够动态并一致地定位控制器类，并能处理其中各种潜在的问题（如：消除不同命名空间中同名类之间的歧义、构造器异常，以及其他一些问题）。

我们的示例项目中只有两个控制器，且打算直接对它们进行实例化，即将类名强行写入控制器工厂——这在实际项目中是极其不明智的。

CreateController方法的目的是，创建能够对当前请求进行处理的控制器实例。至于具体做法是没有任何限制的。唯一的规则，作为该方法的结果，必修返回一个实现IController接口的对象。

在创建控制器工厂时可以遵循MVC框架的约定，也可以放弃这些约定而创建适合于自己项目需要的约定。如果只是单纯为了创建自己的约定，只能有助于理解MVC框架的灵活性，但这不是我们提倡的做法。

### 处理备用控制器

在示例项目中，当请求的控制器与任一控制器都不匹配，将以ProductController类为目标。这在实际项目中可能不是最好的做法，但它表明控制器工厂对请求的解析有充分的灵活性。但这需要我们了解MVC框架中的其他切入点是如何操作的。

默认情况下，MVC框架会根据路由数据中controller的值来选择视图，而不是控制器类的名称。所以，如果希望备用位置按照控制器名称组织的约定来使用视图，就需要改变controller路由属性的值，就像这样：

requestContext.RouteData.Values["controller"] = "Product";

这一改变将导致MVC框架搜索备用的控制器相关的视图，而不是用户请求的控制器。

这里有两个重要的切入点：

1. 控制器工厂不仅要独自负责将请求与控制器进行匹配，而且它还可以对请求进行修改，以改变请求处理管道中后续步骤的行为。这是MVC框架强有力的要素和关键特征。
2. 尽管在控制器工厂中可以自由选择尊重哪种约定，但仍需要了解MVC框架其他部分的约定。并且，因为其他组件也可以改用自定义代码。因此，遵循尽可能多的约定，以允许组件彼此独立地开发和使用是有意义的。

### 实例化控制器类

对控制器类的实例化没有硬性的规则，其中比较好的方法如通过依赖性解析器实例化。使用这种方式可以让我们在开发自定义控制器工厂时专注于请求与控制器类之间的映射，而将依赖性注入这样的问题留下来单独处理，并用于整个程序。如下面使用DependencyResolver类实现控制器的实例化的示例：

return targetType == null ? null : (IController)DependencyResolver.Current.GetService(targetType);

注意，上面示例代码没有检查最终返回的对象是否是IController的实现，但在实际项目中最好进行类型的检测。

### 实现其他接口方法

IcontrollerFactory接口中的另外两个方法如下：

* GetControllerSessionBehavior方法由MVC框架用来确定是否应该为控制器维护会话数据
* 当不再需要CreateController方法创建的控制器对象时，会调用ReleaseController方法。前面示例中检查了这个类是否实现IDisposable接口。如果是，则调用Dispose方法以释放那些可以释放的资源。

这两个方法的实现适用于大多数项目，且可以直接照搬。（一些具体的实现请看后续介绍）

### 注册自定义控制器工厂

通过ControllerBuilder类注册自定义控制器工厂。在程序启动时必须注册自定义工厂控制器，即在Global.asax.cs中使用Application\_Start方法，如：

**using ControllerExtensibility.Infrastructure;**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Routing;

namespace ControllerExtensibility

{

// 注意: 有关启用 IIS6 或 IIS7 经典模式的说明，

// 请访问 http://go.microsoft.com/?LinkId=9394801

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

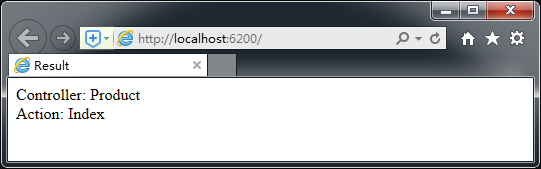
**ControllerBuilder.Current.SetControllerFactory(new CustomControllerFactory());**

}

}

}

此时启动程序将得到预期效果——浏览器将请求跟URL，并被路由系统映射到Home控制器。自定义工厂会通过创建ProductController类的实例来处理对Home控制器的请求，如图：



## 使用内建的控制器工厂

在了解了控制器工厂的工作机制后接着来看看内建的控制器工厂，对于大多数程序，内建的控制器工厂类——DefaultControllerFactory是完全足够的。当从路由系统收到一个请求时，该工厂会考察路由数据，找到controller属性的值，并试图在这个Web应用程序中找到满足以下条件的类（控制器类的条件）：

* 这个类必须是一个公共类（public类）。
* 这个类必须是具体类。
* 这个类必须没有泛型（Generic）参数。
* 类名必须以Controller结尾。
* 这个类必须实现IController接口。

DefaultControllerFactory类维护着程序中这些类的一个列表，且在请求到达时不需要每次都执行一次搜索。如果找到一个合适的类，便用控制器激活器（Controller Activator）创建一个实例，控制器的工作便完成了。如果没有匹配的控制器，那么便不能对该请求作进一步处理。

因为DefaultControllerFactory是遵循“约定优于配置”的，所以不需要在配置文件中注册控制器，需要做的全部工作仅仅是创建满足这个工厂查寻条件的类。

如果希望创建自定义控制器工厂的行为，可以对默认工厂的设置进行配置，或重写它的一些方法。这样，便能建立有用的“约定优于配置”行为，而不必向之前那样重新创建它。

### 命名空间优先排序

在创建路由时，做好对一个或多个命名空间的优先级设置可以解决控制器的多义性问题。且对命名空间列表处理并对其优先排序的就是这个控制器工厂：DefaultControllerFactory。

提示：Global优先级会被路由优先级所重写。即，可以定义一个全局策略，然后在必要时定制个别路由。

如果程序或项目中存在多个路由，那么全局性地指定命名空间优先级可能会更方便一些，以使这种优先级能够应用所有的这些路由。下面演示了如何在Global.asax文件的Application\_Start方法中实现这一功能（当然也可以使用App\_Start文件夹中的RouteConfig.cs文件）：

using ControllerExtensibility.Infrastructure;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Routing;

namespace ControllerExtensibility

{

// 注意: 有关启用 IIS6 或 IIS7 经典模式的说明，

// 请访问 http://go.microsoft.com/?LinkId=9394801

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

**ControllerBuilder.Current.DefaultNamespaces.Add("MyControllerNamespace");**

**ControllerBuilder.Current.DefaultNamespaces.Add("MyProbject.\*");**

}

}

}

上面示例代码中加粗部分添加给予优先的命名空间，这些被添加的命名空间并不分优先级，所谓优先级只是相对未被添加的命名空间而言的。也就是说当控制器工厂在Add方法定义的命名空间中找不到合适的控制器类，那么将搜索整个程序。

提示：上面的第二条设置命名空间的语句使用了星号（\*）。这表明控制器工厂应该查找MyProject命名空间及其所包含的任意子命名空间。这里可以用“.\*”作为命名空间的结尾，但不能在Add方法中使用任何正则表达式语法。

### 定制DefaultControllerFactory的控制器实例化

定制DefaultControllerFactory类如何实例化控制器对象有许多方式。一般来讲对控制器工厂进行定制普遍的原因是为了添加对DI的支持。这些方法中哪种最合适这要取决于在程序中的其他地方如何使用DI。

1. 使用依赖性解析器

在依赖性解析器（Dependency Resolver）可用时，DefaultControllerFactory类将用它来创建控制器。

DefaultControllerFactory会调用IDependencyResolver. GetService方法，以请求控制器实例，这为解析并注入依赖性提供了机会。

1. 使用控制器激活器

也可以通过创建一个控制器激活器（Controller Activator）的方法，将DI引入到控制器中。通过实现IControllerActivator接口创建激活器，IControllerActivator接口如下：

using System;

using System.Web.Routing;

namespace System.Web.Mvc

{

public interface IControllerActivator

{

IController Create(RequestContext requestContext, Type controllerType);

}

}

该接口的Create方法需要一个描述请求的RequestContext对象和一个指定应该对哪个控制器类进行实例化的类型（Type）。如下示例：

using ControllerExtensibility.Controllers;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace ControllerExtensibility.Infrastructure

{

public class CustomControllerAtivator : IControllerActivator

{

public IController Create(RequestContext requestContext, Type controllerType)

{

if (controllerType == typeof(ProductController))

{

controllerType = typeof(CustomerController);

}

return (IController)DependencyResolver.Current.GetService(controllerType);

}

}

}

该示例很简单，如果请求的是ProductController类，将以CustomerController类的实例作为其响应。这里这么做仅仅是为了演示如何利用IControllerActivator接口在控制器工厂和依赖性解析器之间截取请求。

为了使用这个自定义的激活器，需要为DefaultControllerFactory的构造函数传递一个实现类的实例，而且需要在Global.asax文件的Application\_Start方法中进行注册，如：

using ControllerExtensibility.Infrastructure;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Routing;

namespace ControllerExtensibility

{

// 注意: 有关启用 IIS6 或 IIS7 经典模式的说明，

// 请访问 http://go.microsoft.com/?LinkId=9394801

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

**ControllerBuilder.Current.SetControllerFactory(**

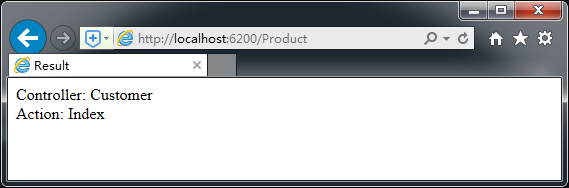
**new DefaultControllerFactory(new CustomControllerAtivator()));**

}

}

}

启动程序，并导航至/Product，就好看到这一自定义激活器的效果。路由将以Product控制器为目标，而DefaultControllerFactory会要求激活器对ProductController类进行实例化，但激活器截取了这一请求，反而创建了一个CustomerController实例对象进行替代。结果图如下：



1. 重写DefaultControllerFactory方法

还可以通过重写DefaultControllerFactory类中的方法来定制控制器的创建。下表给出了可以重写的三个方法，具体见表：

|  |  |  |
| --- | --- | --- |
| 方法 | 结果 | 描述 |
| CreateController | IController | IcontrollerFactory接口的CreateController方法的实现。默认情况下，这个方法调用GetControllerType来确定应该实例化哪一个类，然后通过将结果传递给GetControllerInstance方法，来获得一个控制器对象 |
| GetControllerType | Type | 检索指定名称和请求上下文的控制器类型 |
| GetControllerInstance | IController | 创建指定类型的一个实例 |

## 创建自定义动作调用器

控制器是通过Controller派生而来，而动作方法是由动作调用器（Action Invoker）调用的。动作调用器的接口为：

namespace System.Web.Mvc

{

public interface IActionInvoker

{

bool InvokeAction(ControllerContext controllerContext, string actionName);

}

}

该接口唯一成员InvokeAction的参数是一个ControllerContext对象和一个含有待调用动作名称的字符串。该成员方法返回的是一个布尔型结果，如果是“true”则表示找到并调用了这个动作；“false”表示控制器没有匹配的动作。

请注意，上面说的是“动作”，并未使用“方法”或“动作方法”这样的词汇来描述，是因为动作与方法之间的关联是严格可选的。虽然这是内建的动作调用器所采取的办法，但可以采取选择的任何方式随意地处理动作。如下面的实现：

注：动作是一种行为，而动作方法是实现这种行为的代码。动作调用器的作用是实现对一个动作的调用，而控制器中才是实现这个动作的动作方法。也就是说动作与动作方法的名是可以不同的，后面介绍到“使用自定动作名”的内容时将会进一步了解这些区别。

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Infrastructure

{

public class CustomActionInvoker : IActionInvoker

{

public bool InvokeAction(ControllerContext controllerContext, string actionName)

{

if (actionName == "Index")

{

controllerContext.HttpContext.Response.Write("This is output from the Index action");

return true;

}

else

{

return false;

}

}

}

}

动作调用器实际上并不关心控制器类中的方法，具体的方法是其自行处理的。在示例代码中，如果是对Index动作的请求，那么该调用器直接将一条消息写到Response。如果是对其他动作的请求，则返回false，这样MVC框架将报告一个“404——未找到”的错误消息显示给用户。

与一个控制器相关联的动作调用器是通过Controller.ActionInvoker属性获得的。这样表明不同的控制器可以使用不同的动作调用器。现在添加一个名为ActionInvoker的新控制器来进行演示：

using ControllerExtensibility.Infrastructure;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Controllers

{

public class ActionInvokerController : Controller

{

//

// GET: /ActionInvoker/

public ActionInvokerController()

{

this.ActionInvoker = new CustomActionInvoker();

}

}

}

在这个示例中我们依靠动作调用器去处理请求而非动作方法。启动程序，并导航至/ActionInvoker/Index，可以看到下面的效果，如果是导航至该控制的其他方法将会看到404的错误页面：



还是那句话，不建议自行实现动作调用器，而且，如果这么做也最好别安装示例这种做法。我们这里主要是为了探究其工作原理，另外内建的支持有一些非常有用的特性，最后在示例代码中存在的问题有缺乏可扩展性、贫乏的职责分离，而且缺乏对各种视图的支持。（当然在请求处理管道的几乎每一个方面都是可定制或完全可替换的）

## 使用内建的动作调用器

默认的内建动作调用器ControllerActionInvoker类非常完善，而且它与之前的实现不同，它依靠方法进行操作。

一个实现动作的方法必须满足如下几个条件：

* 必须是public的
* 必须不是static的
* 必须不在System.Web.Mvc. Controller或它的任何基类中
* 必须没有专用名

上述条件中前两个很简单。第三个排除了在Controller类或其基类中实现的方法，这意味着不包括IController接口实现的方法。原因很简单，谁也不希望把控制器的内部工作暴露给外部世界。最后一个条件意味着排除了构造器、属性以及事件访问器。事实上，不可以采用具有System.Reflection.MethodBase的IsSpeciaName标志的类成员来处理一个动作。

注：具有泛型参数的方法（MyMethod<T>()）满足所有条件，但如果视图调用这样的方法来处理一个请求，MVC框架会抛出一个异常。

默认情况下，ControllerActionInvoker查找一个具有与请求的动作同名的方法。如果找到一个这样的方法，便调用它来处理这个请求。但实际上MVC也提供了一些微调这一过程的机会。

### 自定义动作名

通常，动作方法的名称确定了它所表示的动作。Index动作方法对Index动作进行服务。但也可以用ActionName注解属性来重写这一行为：

using ControllerExtensibility.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Controllers

{

public class CustomerController : Controller

{

public ViewResult Index()

{

return View("Result", new Result { ControllerName = "Customer", ActionName = "Index" });

}

**[ActionName("Enumerate")]**

public ViewResult List()

{

return View("Result", new Result { ControllerName = "Customer", ActionName = "List" });

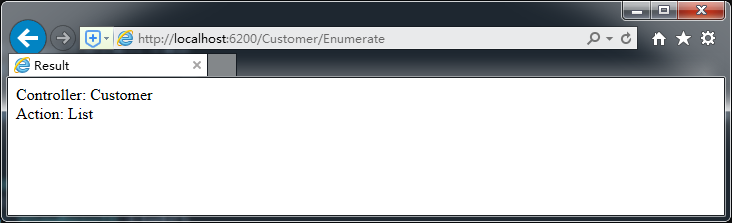
}

}

}

上面示例中我们将动作方法List重写为Enumerate，这意味着，当动作调用器接收到一个对Enumerate动作请求时，它将会使用List方法来实现这一动作，这有点像是将Enumerate动作绑定到了List方法实现上。

下面就导航到/Customer/Enumerate来看看其最终的结果：



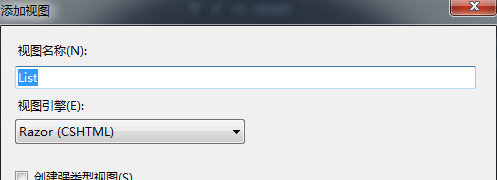
通过ActionName注解属性重写了动作的名称，也同时意味着直接以List方法为目标的URL不再工作，如图：



通常在下面这两种情况下需要用到这种方式重写动作名：

* 可以接收一个作为C#方法名不合法的动作名（如[ActionName(“User-Registration”)]）。
* 如果希望有两个不同的C#方法接收同一组参数，并运用同样的动作名（具有同样的参数的方法不能实现重载），但却要对不同HTTP请求类型进行响应（如一个是[HttpGet]，另一个是[HttpPost]），那么可以对这些方法用不同的C#名来满足编译器的要求，然后用[ActionName]将它们映射到同一个动作名。

需要注意的是，使用这种方式后会出现一个奇怪的现象，就是当添加视图时，Visual Studio将会使用原始的方法名。即当右击List方法添加视图时将看到如下对话框：



这是因为MVC框架是根据动作名查找其默认视图的，所以，在对其使用ActionName注解属性的动作方法创建默认视图时，必须确保该名称与此注解属性的值匹配，而不是与C#的方法名匹配。

### 使用动作方法选择

很多时候，一个控制器中有多个同名动作，这可能是因为有多个方法的重载（它们的参数不同）；或是因为使用了ActionName注解属性使多个方法表示同一个动作。

在这情况下MVC框架需要选择一些相应的动作，已处理一个请求的辅助办法。做这种事情的机制叫做“动作方法选择（Action Method Selection）”。它允许对一个动作定义其乐于处理的请求的种类。如下面就是这一情况的示例：

using SportsStore.Domain.Abstract;

using SportsStore.Domain.Entities;

using SportsStore.WebUI.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace SportsStore.WebUI.Controllers

{

public class CartController : Controller

{

// …为简化，此处省略其他代码

**[HttpPost]**

public ViewResult Checkout(Cart cart, ShippingDetails shippingDetails)

{

if (cart.Lines.Count() == 0)

{

ModelState.AddModelError("", "Sorry, your cart is empty!");

}

if (ModelState.IsValid)

{

\_orderProcessor.ProcessOrder(cart, shippingDetails);

cart.Clear();

return View("Completed");

}

else

{

return View(shippingDetails);

}

}

public ViewResult Checkout()

{

return View(new ShippingDetails());

}

}

}

动作调用器在选择一个动作时，会利用动作方法选择器来消除不明确性。例如上面示例中有两个候选方法。调用器会优先选择具有选择器的动作（[HttpPost]就是一种动作方法选择器）。这样MVC将优先评估HttpPost选择器，以此来确定是否可以处理该请求，如果可以将使用该方法，否则使用另一个方法。

有些内疚的注解属性可以做不同HTTP请求的选择器：HttpPost用于POST请求，HttpGet用于GET请求，HttpPut用于PUT请求等等。另一个内建注解属性是NonAction（非动作），它向动作调用器指示：这个否则会被视为动作方法的方法，不应该作为动作方法来使用（即该方法不作为动作方法）。下面示例给出了这一注解属性的用法：

using ControllerExtensibility.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Controllers

{

public class CustomerController : Controller

{

**[NonAction]**

public ActionResult MyAction()

{

return View();

}

}

}

该示例代码中的MyAction方法将不会被作为动作方法，即便它符合调用器查找的所有准则。这么做的意义是可以确保不会将控制器的工作暴露成动作。当然通常应该把这种方法简单地标记为private，以防止它们作为动作被调用。然而，如果出于某些原因，必须将方法标记为public时，则[NonAction]是非常有用的。

以NonAction方法为目标的URL请求会生成一个404——NotFound错误。如图：



1. 创建自定义动作方法选择器

动作方法选择器派生于ActionMethodSelectorAttribute类如：

using System;

using System.Reflection;

namespace System.Web.Mvc

{

[AttributeUsage(AttributeTargets.Method, AllowMultiple = false, Inherited = true)]

public abstract class ActionMethodSelectorAttribute : Attribute

{

protected ActionMethodSelectorAttribute();

public abstract bool IsValidForRequest(ControllerContext controllerContext, MethodInfo methodInfo);

}

}

从上面代码可以看出这是一个抽象类，它定义了一个抽象方法：IsValidForRequest。该方法的参数ControllerContext对象，用来对请求进行检测；另一个MethodInfo类型参数用来获取运用了选择器的方法的信息。如果该方法能够处理请求，便通过IsValidForRequest返回true，否则返回false。下面示例演示了一个简单的自定义动作方法选择器：

using System.Web.Mvc;

namespace ControllerExtensibility.Infrastructure

{

public class LocalAttribute : ActionMethodSelectorAttribute

{

public override bool IsValidForRequest(ControllerContext controllerContext, System.Reflection.MethodInfo methodInfo)

{

return controllerContext.HttpContext.Request.IsLocal;

}

}

}

该自定义动作选择器的使用演示如下，首先创建一个Home控制器用以进行此项演示，该Home控制器如下：

using ControllerExtensibility.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Controllers

{

public class HomeController : Controller

{

//

// GET: /Home/

public ActionResult Index()

{

return View("Result", new Result { ControllerName = "Home", ActionName = "Index" });

}

[ActionName("Index")]

public ActionResult Index()

{

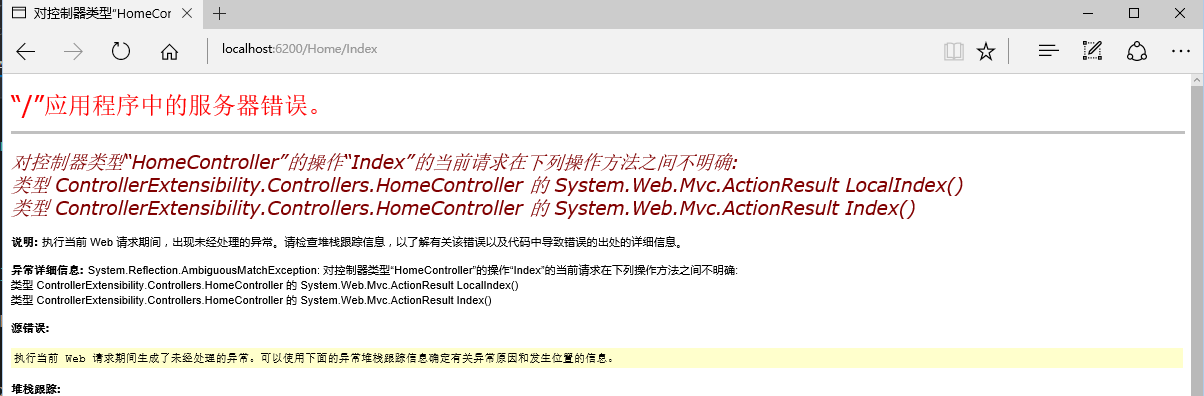
return View("Result", new Result { ControllerName = "Home", ActionName = "LocalIndex" });

}

}

}

本例用ActionName注解属性创建了具有两个Index动作方法的情况。此时，当到达的请求为/Home/Index时，动作调用器将无法猜出应该使用哪一个方法。因此，当接收到这样的请求时，会产生错误信息，如下图所示：



为了解决这一问题，可以对其中的一个歧义方法运用一个方法选择注解属性，如：

**[Local]**

[ActionName("Index")]

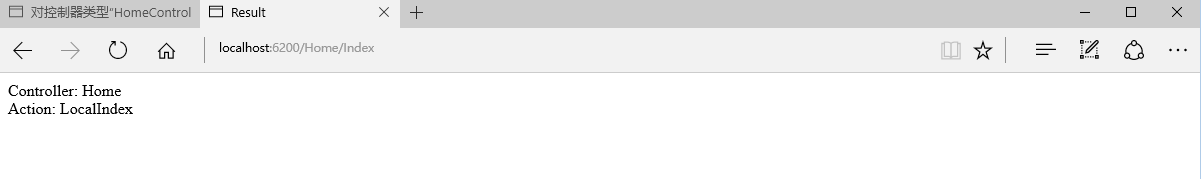
public ActionResult LocalIndex()

{

return View("Result", new Result { ControllerName = "Home", ActionName = "LocalIndex" });

}

再次启动将会看到如下图的效果：



动作方法的歧义过程

动作调用器从一个可能的候选方法列表开始进行处理（满足动作条件的控制器方法）。然后经理如下几个过程：

首先，调用器会根据名称尽可能丢弃掉一些方法。只有与目标动作同名，或与ActionName注解属性相配的方法被保留在这个列表中。

其次，调用器丢弃那些动作方法选择器注解属性对当前请求返回false的动作方法。

如果恰好只留下一个带有选择器的动作方法，那么就是要调用的方法。如果有多个带有选择器的方法，那么便抛出一个异常，因为该动作调用器不能消除可用方法之间的歧义。

如果不存在带有选择器的动作方法，那么该调用器便查找不带选择器的那些方法。如果恰好有一个这样的方法，那么这就是要调用的方法。如果有多个不带选择器的方法，便抛出一个异常，因为调用器不能在它们之间做出选择。

1. 处理未知动作

如果动作调用器找不到一个合适的可调用的动作方法，便从它的InvokeAction方法返回false。发送这种情况时Controller类会调用它的HandleUnknownAction方法。默认情况下，这个方法会返回一个“404——未找到”响应返回到客户端。但是，如果想做一些特殊的事情，可以在控制器类中选择重写这个方法。下面是在Home控制器中重写该方法的示例：

using ControllerExtensibility.Infrastructure;

using ControllerExtensibility.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Controllers

{

public class HomeController : Controller

{

//

// GET: /Home/

// …此处省去其他代码

**protected override void HandleUnknownAction(string actionName)**

**{**

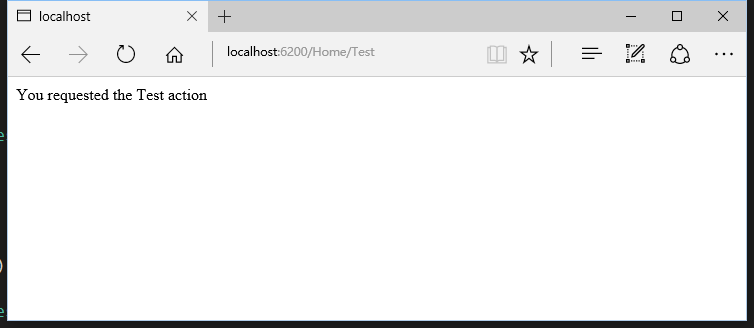
**Response.Write(string.Format("You requested the {0} action", actionName));**

**}**

}

}

此时启动程序，并导航至一个不存在的地址，就能看的这种修改的结果：



## 用特殊控制器改善性能

MVC框架提供了两种可以改善MVC Web应用程序性能的特殊控制器。但这和其他性能优化一样，它也表现出了一些折中，如在易用性方面或降低性能方面。下面我们就来看看这两种控制器的优缺点。

### 使用无会话控制器

通过名字我们也能猜出它是通过取消会话缓存来实现性能优化的。一般来说，控制器支持会话状态，这可以用来跨请求地存取数据值，是MVC开发人员的工作更轻松。由于会话数据会消耗服务器内存或一些其他存储单元空间，而且，多个Web服务器之间数据同步的需求，使得在服务器场（Server Farm）上运行应用程序更加困难（服务器场即联合运行一个大型应用程序的多个服务器）。

为了简化会话状态，ASP.NET对一个给定的会话在某一时刻只处理一个查询。如果客户端形成了多个重叠的请求，它们将被排成队列，并由服务器依序处理。这样做的好处是不需要担忧多个请求对同一数据进行修改的情况。缺点是得不到所希望的请求吞吐量。

然而不是所有情况下，不是所有的控制器都需要这种会话状态特性的。此时，就可以使用无会话的控制器来优化，这既优化了性能，也避免了棘手的会话维护工作。它们与规则的控制器之间的区别在于把它们用于处理一个请求时，MVC框架不加载或不存储会话的状态；以及重叠请求可以同时处理。

1. 在自定义的控制器器工厂中管理会话状态

之前介绍的“创建自定义控制器工厂”一节中实现了一个CustomControllerFactory类，该类继承于IControllerFactory接口，该接口中有一个GetControllerSessionBehavior方法，此方法的SessionStateBehavior类型返回值可以实现对会话状态的管理。该返回值是一个枚举，含有四个值，分别为：

* Default：使用默认的ASP.NET行为，它会根据HttpConext来决定会话状态的配置
* Required：启用完全的读写会话状态
* ReadOnly：启用只读会话状态
* Disabled：完全禁用会话状态

通过返回GetControllerSessionBehavior方法的SessionStateBehavior的值，实现IControllerFactory接口的控制器工厂会直接设置控制器会话状态的行为。下面演示的代码对之前创建的CustomControllerFactory控制器工厂中GetControllerSessionBehavior方法的实现做了修改，具体如下：

…

public SessionStateBehavior GetControllerSessionBehavior(RequestContext requestContext, string controllerName)

{

switch (controllerName)

{

case "Home":

return SessionStateBehavior.ReadOnly;

case "Product":

return SessionStateBehavior.Required;

default:

return SessionStateBehavior.Default;

}

}

…

1. 用DefaultControllerFactory管理会话状态

如果使用的是内建的控制器工厂，则可以将SessionState注解属性运用于每个控制器类，以便对控制器的会话状态进行控制，如：

using ControllerExtensibility.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.SessionState;

namespace ControllerExtensibility.Controllers

{

**[SessionState(SessionStateBehavior.Disabled)]**

public class FastController : Controller

{

public ActionResult Index()

{

return View("Result", new Result { ControllerName = "Fast", ActionName = "Index" });

}

}

}

该控制器运用的SessionState注解属性影响着该控制器中的所有动作。其唯一的参数是SessionStateBehavior枚举中的一个值。由于此处设置为会话的完全禁止状态，如果此时在控制器中设置一个会话值，如：

Seesion["Message"]="Hello";

或在一个视图中试图从会话状态读取一个值，如：

Message:@ Seesion["Message"]

那么，在这个动作被调用或这个视图被渲染时，MVC框架会抛出一个异常。（当会话状态设置为Disabled时，HttpContext.Session属性将会返回null值）

如果设置为ReadOnly，可以读取由其他控制器设置的值，但此时如果视图设置或修改一个值，仍然会得到一个运行时异常，但可以通过HttpContext.Session属性获取该会话的细节。

提示：如果只是简单的把数据从控制器传递给视图，可以考虑使用ViewBag特性，它不受SessionState注解属性的影响。

### 使用异步控制器

核心ASP.NET平台维护着一个用来处理客户端请求的.NET线程池。这个线程池叫作“工作线程池（Worker Thread Pool）”，而这些线程成为“工作线程（Worker Thread）”。当接收到一个请求时，将占用线程池中的一个工作线程，以用来进行该请求的处理。当请求处理完后，该工作线程会被返回给线程池，以便用于新请求的处理。这么做有两个好处：

* 通过重用工作线程，避免了每次处理一个请求时，都要创建一个新线程的开销。
* 通过具有固定数目的可以工作线程，避免了超出服务器处理能力的并发请求情况。

如果请求可以被短时间处理，则工作线程池会很正常。但是，如果有一些依赖于其他服务器，且占用长时间才能完成的动作，那么会可能遇到所有工作线程都被绑定于等待其他系统完成其工作的情况。

此时，将会出现线程池所有线程的等待状态，而这只占用服务器很少一部分资源，其他后续传入的请求将被排成队列。这将陷入程序处理停顿，但服务器大片闲置的奇怪状态。

为了解决这一问题，我们可以使用异步控制器。这会提高程序的整体性能，但不利于执行异步操作。

注：异步控制器只能对占用I/O或占用网络带宽，而非CPU密集型的动作是有用的。异步控制器视图解决的问题应当是线程池与所处理的请求类型之间搭配不当的状况。线程池意在确保每个请求得到一片服务器资源，但可能最终停滞于一组无所事事的工作线程上。如果对CPU密集型动作使用额外的后台线程，那么会因为涉及太多的并发请求而消弱服务器资源。

现在就来看看异步控制器是如何工作的。

首先，先新建一个常规的同步控制器：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers

{

public class RemoteDataController : Controller

{

//

// GET: /RemoteData/

public ActionResult Data()

{

RemoteService service = new RemoteService();

string data = service.GetRemoteData();

return View((object)data);

}

}

}

上面控制器动作方法中调用了一个耗时但低CPU的活动。下面是该控制器中使用的模型类：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading;

using System.Web;

namespace ControllerExtensibility.Models

{

public class RemoteService

{

public string GetRemoteData()

{

Thread.Sleep(2000);

return "Hello from the other side of the world";

}

}

}

为了简单描述问题，这里使用Thread.Sleep模拟了一个两秒的延时。接下来是一个简单的视图，名为Data.cshtml，如：

@model string

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<title>Data</title>

</head>

<body>

<div>

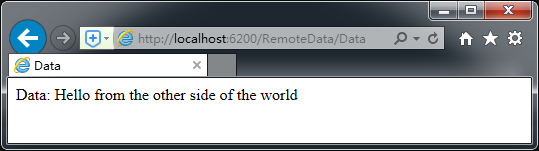
Data: @Model

</div>

</body>

</html>

运行并导航到/ RemoteData/Data这个地址，该动作方法被调用、创建RemoteService对象，并执行GetRemoteData方法。两秒之后将会得到一个如下图所示的信息：



这样就模拟了之前说的那种现象。

现在模拟了这个问题，也该看看如何通过创建异步控制器来解决这个问题了。创建异步控制器有两种方式。一种是实现System.Web.Mvc.Async. IAsyncController接口，这是一个与IController对等的异步接口。由于实现该接口需要过多的.NET并发编程知识，并不易于实现，所以，这里打算通过另一种方式来实现。

提示：异步控制器中不是所有的动作都需要异步，也可以包含同步方法，它们的行为和预期的一样。

通过System.Web.Mvc.AsyncController对控制器进行派生，可以很方便的实现一个异步控制器，其他IAsyncController接口的方法将由其自行实现，我们就不需要对此做过多的关注了。下面是对RemoteDataController做出的修改，粗体字部分给出了修改的高亮显示：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using ControllerExtensibility.Models;

using System.Threading.Tasks;

namespace ControllerExtensibility.Controllers

{

public class RemoteDataController : **AsyncController**

{

//

// GET: /RemoteData/

public **async Task<ActionResult>** Data()

{

string data = **await Task<string>.Factory.StartNew(() =>**

**{**

return new RemoteService().GetRemoteData();

**});**

return View((object)data);

}

}

}

也可以在程序的其他地方，通过异步控制器来使用异步方法。下面是对RemoteService类进行的修改：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading;

using System.Threading.Tasks;

using System.Web;

namespace ControllerExtensibility.Models

{

public class RemoteService

{

public string GetRemoteData()

{

Thread.Sleep(2000);

return "Hello from the other side of the world";

}

**public async Task<string> GetRemoteDataAsync()**

**{**

**return await Task<string>.Factory.StartNew(() =>**

**{**

**Thread.Sleep(2000);**

**return "Hello from the other side of the world";**

**});**

**}**

}

}

其中异步的方法与前面的同步方法的返回结果是相同的。下面是在异步控制器中对该异步方法的调用示例：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using ControllerExtensibility.Models;

using System.Threading.Tasks;

namespace ControllerExtensibility.Controllers

{

public class RemoteDataController : AsyncController

{

//

// GET: /RemoteData/

public async Task<ActionResult> Data()

{

string data = await Task<string>.Factory.StartNew(() =>

{

return new RemoteService().GetRemoteData();

});

return View((object)data);

}

**public async Task<ActionResult> ConsumeAsyncMethod()**

**{**

**string data = await new RemoteService().GetRemoteDataAsync();**

**return View("Data", (object)data);**

**}**

}

}

这两种方式的区别在于是在异步控制器创建Task对象，还是在异步方法中创建。其效果都是在等待GetRemoteData调用完成期间，不会绑定工作线程。于是，线程可用于处理其他请求，这可以极大地改善MVC框架系统的性能。