# 过滤器

**过滤器**（Filter）把附加逻辑注入到MVC框的请求处理，实现了交叉关注。所谓**交叉关注**（Cross-Cutting Concerns），是指可以用于整个应用程序，而又不适合放置在某个局部位置的功能，否则会打破关注分离模式。典型的例子有：登录、授权、缓存等等。

## 使用过滤器

如果希望动作方法只能被认证用户所使用，可以在每个动作方法中检查请求的授权状态。如在动作方法中使用Request.IsAuthenticated方法明确地检查授权：if(!Request.IsAuthenticated){如果未被授权，则…}。

但是，如果在项目中这么做，那会非常繁琐，他需要在每一个需要有授权认证的动作方法中进行判断。所以，采用过滤器才是最好的办法，如清单1：

// 说明：

// 过滤器是 .NET 的注解属性，可以把它们运用于动作方法或控制器类。

// 当被用于控制器类时，其作用效果将覆盖当前控制器中的每一个方法。

[Authorize]

public class AdminController : Controller

{

public ViewResult Index()

{

…

}

public ViewResult Edit(int productId)

{

…

}

}

.NET的注解属性，一个新鲜的事物

**注解属性**（Attribute）是派生于System.Attribute的特殊的.NET类。它们可以被附加到其他代码元素上，包括类、方法、属性以及字段等。目的是把附加信息嵌入到已编译的代码中，以便在运行时读回这些信息。

在C#中，注解属性用方括号进行附加，而且可以用已命名参数语法给它们的public属性赋值（如：[MyAttribute(SomeProperty=value)]）。在C#的编译器命名约定中，如果这些注解属性类的名称以单词Attribute结尾，则可以忽略这一部分（如，对于AuthorizeAttribute注解属性，可以写成[Authorize]）。

### 过滤器的四种基本类型

MVC 框架支持四种不同类型的过滤器。分别是：认证过滤器、动作过滤器、结果过滤器、异常过滤器，如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| 过滤器类型 | 接口 | 默认实现 | 描述 |
| Authorization  (认证过滤器) | IAuthorizationFilter | AuthorizeAtrribute | 最先运行，在任何其他过滤器或动作方法之前 |
| Action  (动作过滤器) | IActionFilter | ActionFilterAtrribute | 在动作方法之前及之后运行 |
| Result  (结果过滤器) | IResultFilter | ActionFilterAtrribute | 在动作结果被执行之前和之后运行 |
| Exception  (异常过滤器) | IExceptionFilter | HandleErrorAtrribute | 仅在另一个过滤器、动作方法、或动作结果抛出异常时运行 |

在框架调用一个动作之前，会首先检测该方法的定义，以查看它是否具有这些过滤器设置。如果有，那么便会在请求管道的相应点上调用这些接口所定义的方法。当然，框架默认实现了这些接口。

注：ActionFilterAtrribute类即实现了IActionFilter，也实现了IResultFilter接口，但这是一个抽象类，要求我们必须提供一个实现。

### 将过滤器运用于控制器和动作方法

过滤器可以应用于动作方法，也可以运用于整个控制器。清单1,将Authorize过滤器运用于AdminController类，其效果与将其运用于控制器中的每一个方法相同，如清单2：

public class AdminController : Controller

{

**[Authorize]**

public ViewResult Index()

{

…

}

**[Authorize]**

public ViewResult Edit(int productId)

{

…

}

}

可以运用多个过滤器，也可以混搭它们运用的层级——即，将它们运用于整个控制器或某个动作方法。如清单3演示了三个不同过滤器的使用方式：

**[Authorize(Roles="trader")]** // applies to all actions(运用于所有动作)

public class ExampleController : Controller

{

**[ShowMessage]** // applies to just this action(仅用于本动作)

**[OutputCache(Duration=60)]** // applies to just this action(仅用于本动作)

public ViewResult Index()

{

…

}

}

注：如果为控制器定义了一个自定义基类，那么运用于基类上的任何过滤器都会影响其派生类。

## 创建示例项目

为了后面内容的介绍，我们这里利用空模板创建一个名为Filters的项目。在项目中添加一个Home控制器，并将其Index动作方法修改成如下这样：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Controllers

{

public class HomeController : Controller

{

/// <summary>

/// 一个返回字符串值的动作方法，这样可以使 MVC 绕过 Razor 视图引擎，直接将字符串值发送给浏览器

/// （这只是为了简化，在实际的项目中还是应该使用视图——这里只关注控制器）

/// </summary>

/// <returns></returns>

public string Index()

{

return "This is the Index action on the Home Controller";

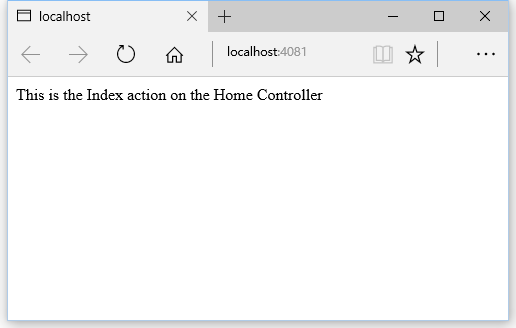
}

}

}

上面的动作方法的返回值修改为字符串型，这样可以是MVC框架绕过Razor视图引擎，直接将字符串发送给浏览器，这样做只是为了简化，因为现在我们只关注控制器。

该示例的运行结果如图所示：



## 使用授权过滤器

授权过滤器是首先运行的过滤器，而且也在动作方法被调用之前。过滤器通过执行设定的授权策略以确保动作方法只被一人在用户所调用。授权过滤器需要实现IAuthorizationFilter接口。如：

namespace System.Web.Mvc

{

public interface IAuthorizationFilter

{

void OnAuthorization(AuthorizationContext filterContext);

}

}

当然也可以通过创建实现IAuthorizationFilter接口的类，实现一个自定义的授权过滤器，使用自己的安全逻辑。但是，不建议这样做，主要原因如下：

警告：编写安全性代码的安全性

一般情况下，自行编写的安全代码总会存在一些缺陷，或未经测试的角落，从而留下了一些安全漏洞。

所以，只要可能，可以使用经过广泛测试并得到验证的安全代码。而且框架也提供了特性完备的授权过滤器，并能够扩展实现自定义授权策略。

一个更安全的办法是创建一个AuthorizeAttribute类的子类，让它照管所有棘手的事情，而且编写自定义的授权代码是很容易的。这里为了演示这种方式的实现，在示例项目中添加了一个Infrastructure文件夹，在这个文件夹中我创建了一个新的类文件：CustomAuthAttribute.cs，其内容如下：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Infrastructure

{

public class CustomAuthAttribute : AuthorizeAttribute

{

private bool \_localAllowed;

public CustomAuthAttribute(bool allowedParam)

{

this.\_localAllowed = allowedParam;

}

protected override bool AuthorizeCore(HttpContextBase httpContext)

{

if (httpContext.Request.IsLocal)

{

return this.\_localAllowed;

}

else

{

return true;

}

}

}

}

这是一个简单的授权过滤器，它实现了阻止本地请求的访问（所谓本地请求就是一种浏览器与应用程序服务器在同一设备上运行而形成的请求，如开发用机）。

在这个示例中继承了AuthorizeAttribute类，并重写了AuthorizeCore方法。这样即实现了自定义授权的目的，也保证了能够获益AuthorizeAttribute的其他内建特性。在构造函数中，使用了一个布尔值，用以指示是否允许本地请求。

这里重写的这个AuthorizeCore方法，是MVC框架用以检查过滤器，是否队请求进行授权访问的方式。其接收的参数是HttpContexBase对象，通过该参数可以获得待处理请求的信息。通过利用AuthorizeAttribute基类的内建特性，只需要关注授权逻辑，并在想要对请求进行授权时，从AuthorizeCore方法中返回true，而再不想授权时返回false。

保持授权注解属性简单

以上对AuthorizeCore方法传递了一个HttpContextBase对象，该对象所提供的是对请求信息进行访问的方法，而不是访问运用该注解属性的控制器和方法的信息。开发人员直接实现IAuthorizationFilter接口的主要原因，是为了获得对传递给OnAuthorization方法的AuthorizationContext的访问，通过它可以得到更广范的信息，包括路由细节，以及当前控制器和动作方法的信息。

一般是不建议自行实现IAuthorizationFilter接口的，主要是因为不仅编写自己的安全代码是不危险的。虽然授权是一种交叉关注，但会在授权注解属性中建立一些逻辑，这些逻辑会与控制器紧密地耦合在一起，这会破坏关注分离，并导致测试和维护的问题。要尽可能保持授权注解属性简单，并关注基于请求的授权——让授权的上下文来自于运用该注解属性的地方。

### 运用自定义授权过滤器

可以像默认的过滤器那样使用自定义的授权过滤器，如下演示的这样（加粗部分）：

**using Filters.Infrastructure;**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Controllers

{

public class HomeController : Controller

{

/// <summary>

/// 一个返回字符串值的动作方法，这样可以使 MVC 绕过 Razor 视图引擎，直接将字符串值发送给浏览器

/// （这只是为了简化，在实际的项目中还是应该使用视图——这里只关注控制器）

/// </summary>

/// <returns></returns>

**[CustomAuth(false)]**

public string Index()

{

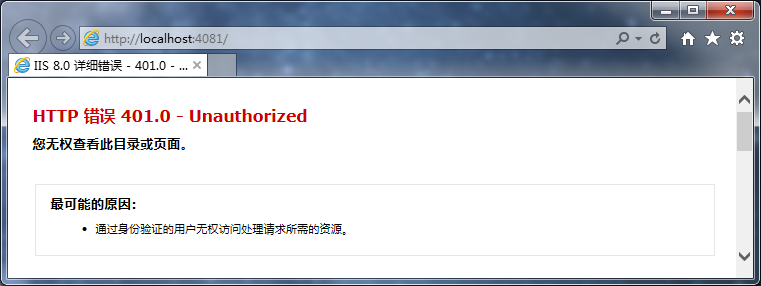
return "This is the Index action on the Home Controller";

}

}

}

示例中将过滤器构造函数的参数设置为false，表示本地请求将被拒绝访问Index动作方法。如果运行程序，并在本机浏览器进行访问，则将看到授权被拒绝的结果，如图：



本地请求被自定义授权过滤器拒绝访问

### 使用内建的授权过滤器

内建的授权过滤器：AuthorizeAttribute拥有自己的实现，它将通过AuthorizeCore方法实现常规的授权任务。

当直接使用AuthorizeAttribute时，可以用这个类的两个公共属性来指定授权策略，下表给出了这两个属性的简单介绍：

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| Users | string | 一个逗号分割的用户名称列表，允许这些用户访问该动作方法 |
| Roles | string | 一个逗号分割的角色列表。为了访问该动作方法，用户必须至少是这就角色之一。 |

下面是这两个属性的使用示例（加粗）：

using Filters.Infrastructure;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Controllers

{

public class HomeController : Controller

{

/// <summary>

/// 一个返回字符串值的动作方法，这样可以使 MVC 绕过 Razor 视图引擎，直接将字符串值发送给浏览器

/// （这只是为了简化，在实际的项目中还是应该使用视图——这里只关注控制器）

/// </summary>

/// <returns></returns>

//[CustomAuth(false)]

**[Authorize(Users = "adam,steve,jacqui", Roles = "admin")]**

public string Index()

{

return "This is the Index action on the Home Controller";

}

}

}

上述示例中同时指定了用户和角色，这表示除非两个条件都满足，否则将不授予权限。这里还有一个隐含条件，即该请求已被认证。如果未指定任何用户或角色，那么任何已被认证的用户都可以使用这个动作方法。

提示：AuthorizeAttribute处理授权，但不负责认证。但我们可以使用任何ASP.NET内建的认证系统或开发自己的认证系统（建议最好使用内建认证系统）。只要认证系统使用标准的ASP.NET的API，那么，AuthorizeAttribute就能限制对控制器和动作的访问。

对于大多数情况，AuthorizeAttribute提供的授权策略已经足够了。但如果希望实现一些特殊功能，可以通过这个类进行派生。虽然这样比直接实现IAuthorizationFilter接口的风险要小很多，但在开发的过程中还要非常谨慎的考虑策略的影响，并对它进行充分的测试。

## 使用异常过滤器

只有在调用一个动作方法时，如果抛出未处理的异常，异常过滤器才会运行。这种异常的来源主要有以下几种：

* 另一个过滤器（授权、动作或结构过滤器）；
* 动作方法本身；
* 当动作结果被执行时。

### 创建异常过滤器

异常过滤器必须实现IExceptionFilter接口，该接口的命名空间为：System.Web.Mvc，其中OnException方法是在发送异常时被调用的。该方法的参数是一个继承自ControllerContext的ExceptionContext类型的对象，它提供了很多有用的属性可以用来获取关于请求的信息，如下表：

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| Controller | ControllerBase | 返回请求的控制器对象 |
| HttpContext | HttpContextBase | 提供对请求细节的访问，以及对响应的访问 |
| IsChildAction | bool | 若是子动作，便返回true |
| RequestContext | RequestContext | 提供对HttpContext和路由数据的访问，通过其他属性，两者都是可用的 |
| RouteData | RouteData | 返回请求的路由数据 |

上表中的属性都是继承自ControllerContext的，除此之外，它还定义了一些附加属性，见下表：

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| ActionDescriptior | ActionDescriptior | 提供动作方法的细节 |
| Result | ActionResult | 用于动作方法的结果：通过将该属性设置为一个非空值过滤器可以取消这个请求 |
| Exception | Exception | 未处理异常 |
| ExceptionHandled | bool | 如果另一个过滤器已经把这个异常标记为“已处理”，则返回true |

被抛出的异常可以通过Exception属性进行操作。将ExceptionHandled属性设置为“true”，异常过滤器可以报告它已经处理了该异常。但即便如此，应用于一个动作的所有异常过滤器还是会被调用，所以，一个比较好的处理方式是检测另一个过滤器是否已经处理了这个问题，以免恢复另一个过滤器已经解决了的问题。

注：如果一个动作方法的所有异常过滤器均为将ExceptionHandled属性设置为“true”，MVC框架将使用默认的ASP.NET异常处理器。这将会显示恐怖的“黄色屏幕”。

Result属性由异常过滤器使用，已告诉MVC框架要做什么。异常过滤器的两个主要用途是对异常进行日志，并将适当的消息显示给用户。下面我们通过 类来演示该如何使用：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Infrastructure

{

public class RangeExceptionAttribute : **FilterAttribute**, IExceptionFilter

{

public void OnException(ExceptionContext filterContext)

{

**if (!filterContext.ExceptionHandled &&**

**filterContext.Exception is ArgumentOutOfRangeException)**

**{**

**filterContext.Result = new RedirectResult("~/Content/RangerErrorPage.html");**

**filterContext.ExceptionHandled = true;**

**}**

}

}

}

这里对ArgumentOutOfRangeException实例进行了处理，采取的办法是将用户浏览器重定向到Content文件夹中名为RangerErrorPage.html文件。

需要注意该类继承了FilterAttribute类，此外还实现了IExceptionFilter接口。为了让一个.NET注解属性类被视为一个MVC过滤器，该类必须实现IMvcFilter接口。我们可以直接实现该接口，但最简单的方式是通过FilterAttribute来派生自己的类，它已经实现了所以需要的接口，并提供一些有用的基本特性，如处理过滤器执行的默认处理顺序。

### 运用异常过滤器

在使用异常过滤器之前还需要前面提到的Content文件夹和RangerErrorPage.html文件（静态的HTML文件）。在本示例中将使用该文件显示一些简单的消息。代码如下：

<!DOCTYPE html>

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

<head>

<title>Rang Error</title>

</head>

<body>

<h2>Sorry</h2>

<span>One of the arguments was out of the expected range.</span>

</body>

</html>

下面，需要对Home控制器添加一个动作方法，它将抛出我们感兴趣的异常。添加的内容如下：

…

public string RangeTest(int id)

{

if (id > 100)

{

return string.Format("The id value is:{0}", id);

}

else

{

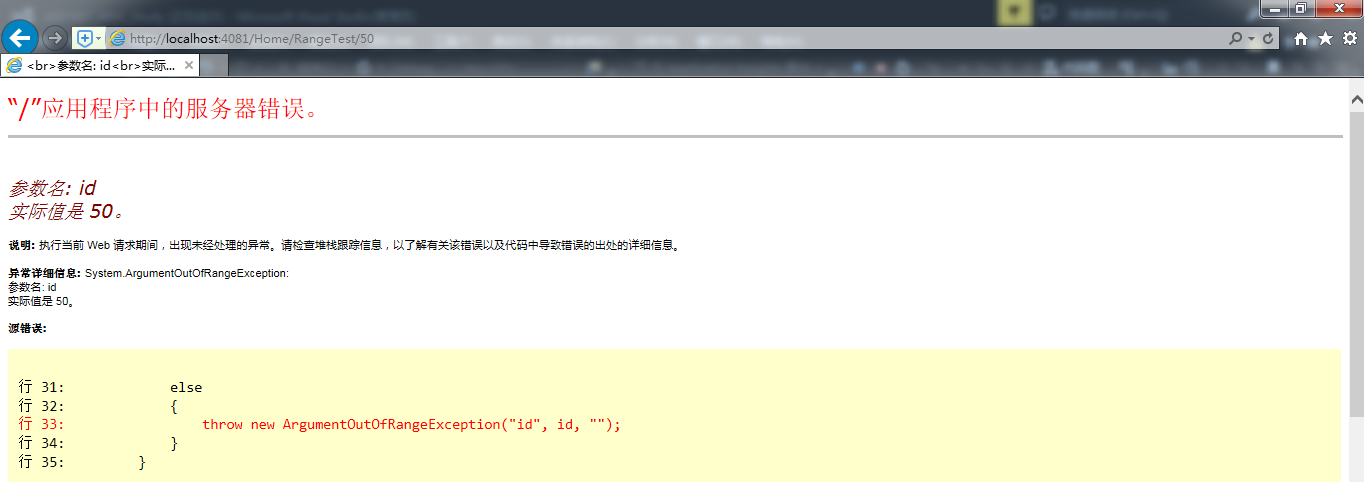
throw new ArgumentOutOfRangeException("id", id, "");

}

}

…

启动程序，并导航至/Home/RangeTest/50（如我的完整URL是：http://localhost:4081**/Home/RangeTest/50**），便可以看到默认的异常处理结果。（Visual Studio为MVC项目创建的默认路由中，具有一个名为id的片段变量，针对这一URL，它的值将被设为50，这回得到下图的响应结果）。



可以将异常过滤器运用于控制器或个别动作，如：

…

**[RangeException]**

public string RangeTest(int id)

{

if (id > 100)

{

return string.Format("The id value is:{0}", id);

}

else

{

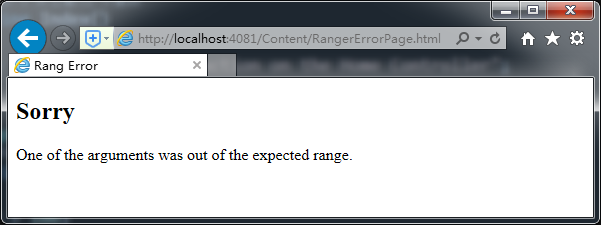
throw new ArgumentOutOfRangeException("id", id, "");

}

}

…

在此导航至/Home/RangeTest/50，将得到如下结果：



### 使用视图来响应异常

通过显示静态内容的页面来处理异常是简单且安全的，但这对用户来说没有什么用处。——最多也就能得到一个泛泛的警告，最后选择退出程序。最好的办法是使用视图来显示问题的细节，并为其提供一些上下文信息和选项，以帮助用户对异常采取处理。

对于该示例，这里做出了一些调整，将RangeExceptionAttribute类修改成如下这样（加粗部分和注释部分为修改内容）：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Infrastructure

{

public class RangeExceptionAttribute : FilterAttribute, IExceptionFilter

{

public void OnException(ExceptionContext filterContext)

{

if (!filterContext.ExceptionHandled &&

filterContext.Exception is ArgumentOutOfRangeException)

{

**int val = (int)(((ArgumentOutOfRangeException)filterContext.Exception).ActualValue);**

**filterContext.Result = new ViewResult**

**{**

**ViewName = "RangeError",**

**ViewData = new ViewDataDictionary<int>(val)**

**};**

// 下面注释掉的方式显示了一个静态页面内容

//filterContext.Result = new RedirectResult("~/Content/RangerErrorPage.html");

filterContext.ExceptionHandled = true;

}

}

}

}

这里创建了一个ViewResult对象，并设置了ViewName和ViewData属性的值，以指定视图名称和要传递给视图的模型对象。这样写显得代码有些凌乱是直接使用ViewResult对象，而不是在Controller类定义的动作方法中使用View方法的原因。这里我们没必要关注这些，只要明白如何实现这一效果即可。后面我们会通过内建的异常过滤器更好的实现同样效果。

这里我们指定了一个名为RangeError的视图，并传了一个int型的参数值，以这个引起异常的参数值作为视图模型对象。后面，继续在项目中添加一个Views/Shared文件夹，并创建RangeError.cshtml文件：

@model int

<!DOCTYPE html>

<html>

<head>

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

<title>Range Error</title>

</head>

<body>

<h2>The value @Model was out of the expected range.</h2>

<div>

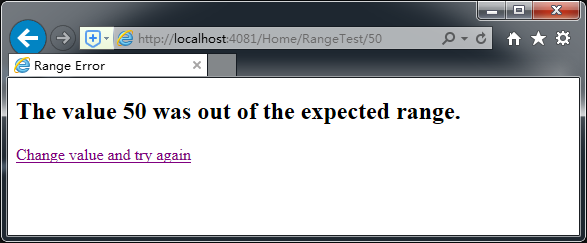
@Html.ActionLink("Change value and try again", "Index")

</div>

</body>

</html>

由于示例功能有限，未向用户指示出解决这一问题的任何有用信息，但这里使用ActionLink辅助器方法创建了一个指向另一个动作方法的连接，目的是演示可以使用一整套视图特性。再次启动程序，并导航至/Home/RangeTest/50，将看到如下效果：



**避免异常错误陷阱**

使用视图来显示错误消息的好处是可以使用布局是错误消息与应用程序的其余部分一致，并生成动态的内容，帮助用户了解发生了什么错误，以及提示他们可以如何处理问题。

但是这么做也存在一些缺陷：这就要求我们在开发的过程当中必须彻底的测试视图，以确保不会产生其他异常。作为简单示例，这里对RangeError.cshtml视图添加了一个Razor代码块，很明显它将抛出一个异常，如：

@model int

**@{**

**// 用来演示在异常视图信息中抛出异常的情况**

**var count = 0;**

**var number = Model / count;**

**}**

<!DOCTYPE html>

<html>

<head>

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

<title>Range Error</title>

</head>

<body>

<h2>The value @Model was out of the expected range.</h2>

<div>

@Html.ActionLink("Change value and try again", "Index")

</div>

</body>

</html>

当该视图被渲染时，将会生成一个DivideByZeroException（被零除异常）。如果启动程序，并导航至/Home/RangeTest/50，将会看到抛出的异常，该异常是在视图渲染期间抛出的，且不由控制器抛出。如下图：



这个演示虽不是真是场景，但它说明了当视图有问题时发送的情况——用户看到了一个困惑的错误提示，甚至与他们在程序中遇到的问题毫不相关。所以，在使用一个依赖于视图的异常过滤器时，必须小心谨慎地对视图进行充分的测试。

### 使用内建的异常过滤器

在实际的项目中一般不需要向前面将的那样去创建自己的过滤器，MVC框架包含了HandleErrorAttribute异常过滤器，它是内建IExceptionFilter接口的实现。该异常过滤器有一些常用的属性，可以用来指定一个异常以及视图和布局名称，详见下表：

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| ExceptionType | Type | 由过滤器处理的异常类型。它也处理通过给定值继承而来的异常类型，但会忽略所有其他类型。其默认值是System.Exception，其含义为，默认地处理所有标准异常 |
| View | string | 该过滤器渲染的视图模板名。如果未指定一个值，则采用默认的Error值，因此，默认情况下会渲染/Views/<currentCotrollerName>/Error.cshtml或/Views/Shared/Error.cshtml |
| Master | string | 在渲染这过滤器的视图时所使用的布局名称。如果未指定一个值，该视图使用其默认布局页面 |

当遇到由ExceptionType指定类型的未处理异常时，此过滤器将渲染由View属性指定的视图（使用默认布局，或有Master属性指定的布局）。

1. 使用内建异常过滤器要做的准备

只有在Web.config文件中启用了自定义错误时，HandleErrorAttribute过滤器才会生效，这可以在<system.web>节点中添加一个customErrors属性即可，如：

…

<system.web>

<httpRuntime targetFramework="4.5" />

<compilation debug="true" targetFramework="4.5" />

<pages>

<namespaces>

<add namespace="System.Web.Helpers" />

<add namespace="System.Web.Mvc" />

<add namespace="System.Web.Mvc.Ajax" />

<add namespace="System.Web.Mvc.Html" />

<add namespace="System.Web.Routing" />

<add namespace="System.Web.WebPages" />

</namespaces>

</pages>

<customErrors mode="On" defaultRedirect="/Content/RangeErrorPage.html"/>

</system.web>

…

model属性的默认值是RemoteOnly，意为在开发期间，HandleErrorAttribute将不会拦截异常，但将应用程序部署到产品服务器，并从另一台计算机发出请求时才会生效。为了看到用户最终将看到的情况，要确保已经将这个自定义错误模式设置为“On”。defaultRedirect属性指定了一个内容页面，在其他情况下都无法显示异常消息时，便会使用该页面。

1. 运营内建的异常过滤器

下面我们看看该如何使用这一内建的异常过滤器：

…

**[HandleError(ExceptionType = typeof(ArgumentOutOfRangeException), View = "RangeError")]**

public string RangeTest(int id)

{

if (id > 100)

{

return string.Format("The id value is:{0}", id);

}

else

{

throw new ArgumentOutOfRangeException("id", id, "");

}

}

…

该示例重建了前面自定义过滤器一样的情况，即通过将视图显示给用户的方式来处理ArgumentOutOfRangeException异常。

在渲染视图时，HandleErrorAttribute过滤器会传递一个HandleErrorInfo视图模型对象，这是一个封装了异常细节的封装程序，它提供了可在视图中使用的附加信息，下表给出了HandleErrorInfo类定义的属性：

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| ActionName | string | 返回生成异常的动作方法名称 |
| ControllerName | string | 返回生成异常的控制器名称 |
| Exception | Exception | 返回此异常 |

下面将演示如何使用这个模型对象来更新RangeError.cshtml视图：

<!--使用 HandleErrorInfo 模型对象-->

@model HandleErrorInfo

<!--model int-->

@{

//// 用来演示在异常视图信息中抛出异常的情况

//var count = 0;

//var number = Model / count;

// 使用 HandleErrorInfo 模型对象

ViewBag.Title = "Sorry,there was a problem!";

}

<!DOCTYPE html>

<html>

<head>

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

<title>Range Error</title>

</head>

<body>

@\*<h2>The value @Model was out of the expected range.</h2>

<div>

@Html.ActionLink("Change value and try again", "Index")

</div>\*@

<!--使用 HandleErrorInfo 模型对象-->

<h2>Sorry</h2>

<span>

The value @(((ArgumentOutOfRangeException)Model.Exception).ActualValue)

was out of the expected range.

</span>

<div>

@Html.ActionLink("Change value and try again", "Index")

</div>

<!--放在 div 中且将 display 设为 none 可将堆栈的跟踪情况隐藏-->

<div style="display:none">

<!--必须包含该属性的值，否则将不能显示异常视图-->

@Model.Exception.StackTrace

</div>

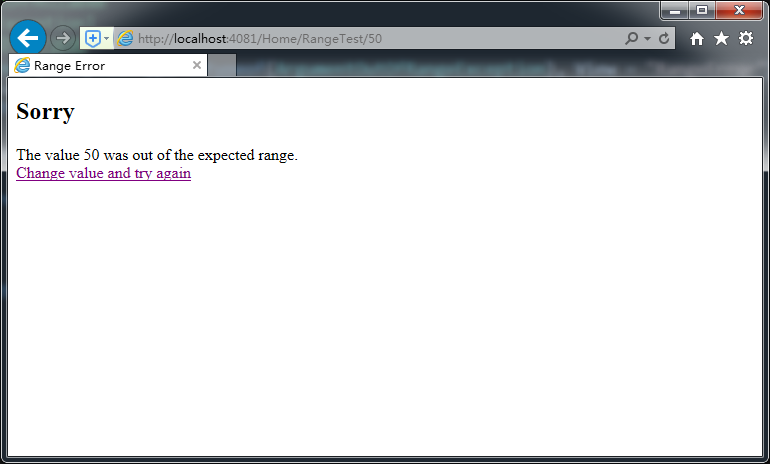
</body>

</html>

该视图中必须将Model.Exception属性值转为ArgumentOutOfRangeException类型，以便能够读取ActualValue属性，因为HandleErrorInfo类是一个用来将任何异常传递给视图的一个通用的模型对象。

注意：在使用HandleError过滤器时有一个奇怪的行为，即视图中必须包含Model. Exception. StackTrace属性的值，否则视图便不会显示给用户。但因为不想显示堆栈的跟踪情况，所以将该值的输出放在一个div元素中，并将其CSS的display属性设置为none，使之对用户是不可见的。

其效果和使用前述自定义异常过滤器一样，如图：



## 使用动作过滤器

动作过滤器是可以被用于任何目的的多用途过滤器。创建这种过滤器需要实现接口IActionFilter，在MSDN中给出的描述是这样的：

命名空间：System.Web.Mvc

语法：public interface IActionFilter

公开成员（方法）：

|  |  |  |
| --- | --- | --- |
| 名称 | 说明 | 语法 |
| OnActionExecuted | 在执行操作方法后调用。 | void OnActionExecuted(  ActionExecutedContext filterContext  ) |
| OnActionExecuting | 在执行操作方法之前调用。 | void OnActionExecuting(  ActionExecutingContext filterContext  ) |

### 实现OnActionExecuting方法

OnActionExecuting方法在调用动作方法之前被调用。可以利用这个机会来检测请求，并可以在这里取消请求、修改请求，或启动一些跨越动作调用期间的活动。该方法的参数参加下表的描述：

**ActionExecutingContext属性**

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| ActionDescriptor | ActionDescriptor | 提供动作方法的细节 |
| Result | ActionResult | 动作方法的结果：通过将该属性设置为非空值，过滤器可以取消该请求 |

可以用过滤器取消一个请求，这只需将Result参数属性设置成一个动作结果即可。下面来研究一下该如何实现这一功能，首先在Infrastructure文件夹中创建一个自己的动作过滤器类，名为：CustomActionArribute，如：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Infrastructure

{

public class CustomActionArribute : FilterAttribute, IActionFilter

{

public void OnActionExecuting(ActionExecutingContext filterContext)

{

if (filterContext.HttpContext.Request.IsLocal)

{

filterContext.Result = new HttpNotFoundResult();

}

}

/// <summary>

///

/// </summary>

/// <param name="filterContex"></param>

/// <remarks>

/// 如果不需要实现任何逻辑，则空着即可。注意不要抛出 NotImplementedExcetion 异常 —— 否则 MVC 将触发异常过滤器

/// </remarks>

public void OnActionExecuted (ActionExecutedContext filterContex)

{

// 尚未实现

}

}

}

上面示例代码中，用OnActionExceuting方法检查请求是否来自本地机器。如果是，便对用户返回一个“404 —— 未找到”的响应。

动作过滤器的运用和其他注解属性一样。为了演示刚刚创建的自定义的动作过滤器，我们需要在Home控制器添加一个新的动作方法，如：

…

**[CustomAction]**

public string FilterTest()

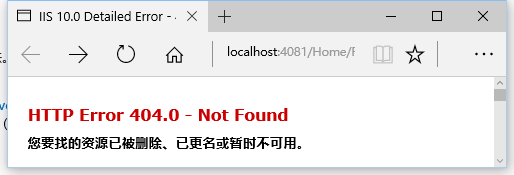
{

return "This is the FilterTest action";

}

…

此时，启动程序并导航至/Home/FilterTest，将得到预期的效果，如果是本地请求将看到如下图所示结果：



### 实现OnActionExecuted方法

用动过滤器还可以执行一些跨动作方法执行的任务，下面就看看该如何实现这一效果（需要新建一个动作过滤器类：ProfileActionAttribute）：

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Infrastructure

{

public class ProfileActionAttribute : FilterAttribute, IActionFilter

{

private Stopwatch timer;

public void OnActionExecuting(ActionExecutingContext filterContext)

{

timer = Stopwatch.StartNew();

}

public void OnActionExecuted(ActionExecutedContext filterContex)

{

timer.Stop();

if (filterContex.Exception == null)

{

filterContex.HttpContext.Response.Write(

string.Format("<div>Action method elapsed time: {0}</div>", timer.Elapsed.TotalSeconds));

}

}

}

}

这里，在OnActionExecuting方法中启动了一个计时器，在动作方法完成之后MVC框架会调用OnActionExecuted方法——其中执行了关闭计时器并打印执行动作方法所耗时间的信息。

下面是使用这一过滤器的实例：

…

// ProfileAction：演示 ProfileAction 过滤器对于耗时测量的效果

[ProfileAction]

//[CustomAction]

public string FilterTest()

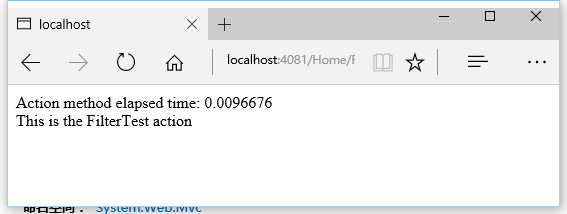
{

return "This is the FilterTest action";

}

…

现在就可以看到这一过滤器的效果了：



提示：页面中显示的耗时信息在动作方法结果被处理之前，是因为**动作过滤器是在动作方法完成之后，但在结果被处理之前执行的**。也就是说这一时间信息不包括对动作结果处理的时间。

ActionExecutedContext类型参数的一些属性的描述详见下表（其中Exception属性返回动作方法抛出的异常，而ExceptionHandled属性只是另一个过滤器是否已经处理了这个异常）：

**ActionExecutedContext属性**

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| ActionDescriptor | ActionDescriptor | 提供动作方法的细节 |
| Canceled | bool | 如果该动作方法已经被另一个过滤器取消，则返回true |
| Exception | Exception | 返回由另一个过滤器或动作方法抛出的异常 |
| ExceptionHandled | bool | 如果异常已经被处理，则返回true |
| Result | ActionResult | 动作方法的结果：通过把这个属性设置为一个非空值，过滤器可以取消这个请求 |

如果另一个过滤器已经取消了这个请求（通过对Result属性设置一个值的办法），从OnActionExecuting方法被调用的时刻开始，Canceled属性便会返回“true”。这仍会调用OnActionExecuted方法，但只是为了清理和释放已被占用的资源。

## 使用结果过滤器

结果过滤器顾名思义，就是要对动作方法产生的结果进行操作，同时它也是多用途过滤器。结果过滤器需要实现IResultFilter接口：

命名空间：System.Web.Mvc

语法：public interface IResultFilter

方法：

|  |  |  |
| --- | --- | --- |
| 名称 | 说明 | 语法 |
| OnResultExecuted | 在操作结果执行后调用。 | void OnResultExecuted(  ResultExecutedContext filterContext  ) |
| OnResultExecuting | 在操作结果执行之前调用。 | void OnResultExecuting(  ResultExecutingContext filterContext  ) |

可以将动作方法的意图与动作方法的执行分离开来。将结果过滤器运用于一个动作方法时，会在动作方法返回动作结果之时、但在执行该动作结果之前，调用（结果过滤器的）OnResultExecuting方法。动作结果被执行之后，会调用OnResultExecuted方法。

下面我们通过在Infrastructure文件夹中创建一个名为ProfileResultAttribute的类来演示一个简单的过滤器：

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Infrastructure

{

public class ProfileResultAttribute : FilterAttribute, IResultFilter

{

private Stopwatch timer;

public void OnResultExecuting(ResultExecutingContext filterContext)

{

timer = Stopwatch.StartNew();

}

public void OnResultExecuted(ResultExecutedContext filterContext)

{

timer.Stop();

filterContext.HttpContext.Response.Write(

string.Format("<div>Result elapsed time: {0}</div>", timer.Elapsed.TotalSeconds));

}

}

}

现在可以将其用在动作方法上了：

…

// ProfileAction：演示 ProfileAction 过滤器对于耗时测量的效果

[ProfileAction]

// 对 ProfileAction 动作方法过滤器作为了一个补充

**[ProfileResult]**

//[CustomAction]

public string FilterTest()

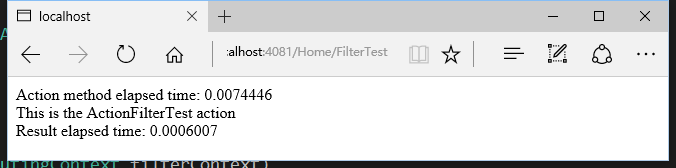
{

return "This is the ActionFilterTest action";

}

…

启动程序，并导航至/Home/FilterTest，查看效果。可以看出结果过滤器的输出显示在动作方法产生的结果之后。因为直到结果被适当地处理之后，OnResultExecuted方法才会被MVC框架执行。这与前面的动作过滤器的OnActionExecuted方法不一样，请注意这一区别。



**使用内建的动作过滤器和结果过滤器类**

MVC框架包含一个内建的类，可以用来创建动作过滤器和结果过滤器。这个类的名称为：ActionFilterAttribute，如：

public abstract class ActionFilterAttribute : FilterAttribute, IActionFilter, IResultFilter

{

protected ActionFilterAttribute();

public virtual void OnActionExecuted(ActionExecutedContext filterContext);

public virtual void OnActionExecuting(ActionExecutingContext filterContext);

public virtual void OnResultExecuted(ResultExecutedContext filterContext);

public virtual void OnResultExecuting(ResultExecutingContext filterContext);

}

如果使用这个类有个好处就是不需要重写和实现不打算使用的方法。除此之外，直接实现过滤器接口没有任何好处。

作为演示，在Infrastructure文件夹中创建了一个名为的新类：

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Infrastructure

{

public class ProfileAllAttribute : ActionFilterAttribute

{

private Stopwatch timer;

public override void OnActionExecuting(ActionExecutingContext filterContext)

{

timer = Stopwatch.StartNew();

}

public override void OnResultExecuted(ResultExecutedContext filterContext)

{

timer.Stop();

filterContext.HttpContext.Response.Write(

string.Format("<div>Total elapsed time: {0}</div>", timer.Elapsed.TotalSeconds));

}

}

}

ActionFilterAttribute过滤器实现了IActionFilter和 IResultFilter接口，这意味着即时未重写所有的方法，MVC框架也会把派生类作为两种过滤器的类型来处理。如上述示例仅实现了IActionFilter接口的OnActionExecuting和IResultFilter接口的OnResultExecuted方法，从而能够继续描述前面的主题，以测量动作方法所消耗的时间，并将结果作为一个单一的单元进行处理。下面我们就查看一下该如何使用该过滤器：

…

// ProfileAction：演示 ProfileAction 过滤器对于耗时测量的效果

[ProfileAction]

// 对 ProfileAction 动作方法过滤器作为了一个补充

[ProfileResult]

// 使用内建的动作过滤器和结果过滤器

**[ProfileAll]**

//[CustomAction]

public string FilterTest()

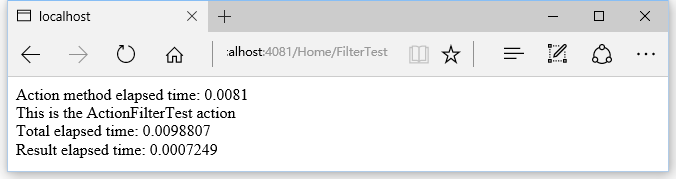
{

return "This is the ActionFilterTest action";

}

…

现在就可以启动程序并查看效果了：



## 使用其他过滤器特性

MVC框架除了前面介绍的过滤器，还有一些高级的过滤功能，这些过滤器更有趣，但用途不广。下面就来看看这些过滤器吧。

### 无注解属性的过滤器

常见的使用过滤器的方式是运用注解属性，但还有一种方式——Controller类也实现了IActionFilter、IAuthorizationFilter、IResultFilter和IExceptionFilter接口。它还对前面看到的每一个OnXXX方法提供了空白虚拟实现，因此，我们可以不用使用注解属性进行标注，而是直接重写OnXXX方法来实现过滤器功能。下面是一个演示示例：

using Filters.Infrastructure;

using System;

using System.Collections.Generic;

**using System.Diagnostics;**

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Controllers

{

public class HomeController : Controller

{

/// <summary>

/// 一个返回字符串值的动作方法，这样可以使 MVC 绕过 Razor 视图引擎，直接将字符串值发送给浏览器

/// （这只是为了简化，在实际的项目中还是应该使用视图——这里只关注控制器）

/// </summary>

/// <returns></returns>

//[CustomAuth(false)]

[Authorize(Users = "adam,steve,jacqui", Roles = "admin")]

public string Index()

{

return "This is the Index action on the Home Controller";

}

//自定义的异常过滤器

//[RangeException]

//内建的异常过滤器

[HandleError(ExceptionType = typeof(ArgumentOutOfRangeException), View = "RangeError")]

public string RangeTest(int id)

{

if (id > 100)

{

return string.Format("The id value is:{0}", id);

}

else

{

throw new ArgumentOutOfRangeException("id", id, "");

}

}

**//// ProfileAction：演示 ProfileAction 过滤器对于耗时测量的效果**

**//[ProfileAction]**

**//// 对 ProfileAction 动作方法过滤器作为了一个补充**

**//[ProfileResult]**

**//// 使用内建的动作过滤器和结果过滤器**

**//[ProfileAll]**

//[CustomAction]

public string FilterTest()

{

return "This is the ActionFilterTest action";

}

**private Stopwatch timer;**

**protected override void OnActionExecuting(ActionExecutingContext filterContext)**

**{**

**timer = Stopwatch.StartNew();**

**}**

**protected override void OnResultExecuted(ResultExecutedContext filterContext)**

**{**

**timer.Stop();**

**filterContext.HttpContext.Response.Write(**

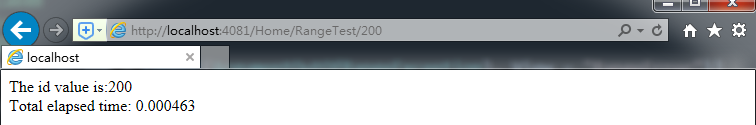
**string.Format("<div>Total elapsed time: {0}</div>", timer.Elapsed.TotalSeconds));**

**}**

}

}

上面示例中删除了FilterTest动作方法上的过滤器（已经注释的部分），因为下面实现的（重写的）动作方法已经实现了这些功能。下面看看效果（导航地址：/Home/RangeTest/200，该URL的目标是RangeTest动作，但它不会发生在演示HandleError过滤器时可能出现的那种异常）：



当要创建一个基类，以派生项目中多个控制器时，这项技术是最有用的。整个过滤点是在一个可重用的位置，放置整个应用程序需要的代码（还有一种方式是使用全局过滤器）。

注：建议使用注解属性，这样可以将控制器逻辑与过滤器逻辑进行分离。如果需要将一个过滤器运用于所有控制器，可以使用全局过滤器。

### 全局过滤器

全局过滤器（Global filter）被用于应用程序的所有动作方法。通过App\_Start/ FilterConfig.cs文件中定义的RegisterGlobalFilters方法，可以让一个常规过滤器成为一个全局过滤器。下面示例代码演示了如何把前面创建的ProfileAll过滤器改成一个全局过滤器：

**using Filters.Infrastructure;**

using System.Web;

using System.Web.Mvc;

namespace Filters

{

public class FilterConfig

{

public static void RegisterGlobalFilters(GlobalFilterCollection filters)

{

filters.Add(new HandleErrorAttribute());

**filters.Add(new ProfileAllAttribute());**

}

}

}

上述示例代码中第一条语句是MVC框架默认创建的异常处理策略，但这一策略默认是禁用的，后面在讲解“创建异常过滤器”时将描述该如何启用这一异常处理策略。

在注册全局过滤器时要注意的是过滤器必须为全名（如：filters.Add(new ProfileAllAttribute());）。一旦注册为全局过滤器，它将被运用于每个动作方法。为了演示这一效果，我们来创建一个新控制器Customer：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Controllers

{

public class CustomerController : Controller

{

public string Index()

{

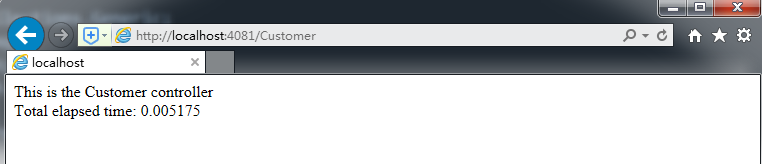
return "This is the Customer controller";

}

}

}

启动并导航至/Customer将看到如下效果（即时未在控制器上直接运用过滤器，全局过滤器也会起作用）：



### 对过滤器执的行进行排序

过滤器的执行顺序：授权过滤器→动作过滤器→结果过滤器。如果有未处理异常，在任何一阶段都会执行异常过滤器。然而，在每一种类型中，都可以控制过滤器的使用顺序。

后面使用一个简单的动作过滤器类来演示对过滤器的执行进行排序，示例过滤器SimpleMessageAttribute如下所示：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Infrastructure

{

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

public class SimpleMessageAttribute : FilterAttribute, IActionFilter

{

public string Message { get; set; }

public void OnActionExecuted(ActionExecutedContext filterContext)

{

filterContext.HttpContext.Response.Write(

string.Format("<div>[Before Action: {0}]</div>", Message));

}

public void OnActionExecuting(ActionExecutingContext filterContext)

{

filterContext.HttpContext.Response.Write(

string.Format("<div>[After Action: {0}]</div>", Message));

}

}

}

可以将该过滤器的多个实例可以运用于一个动作方法（注意AllowMultiple已被设置为true，所以可以实现这一功能）：

using Filters.Infrastructure;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace Filters.Controllers

{

public class CustomerController : Controller

{

**[SimpleMessage(Message = "A")]**

**[SimpleMessage(Message = "B")]**

public string Index()

{

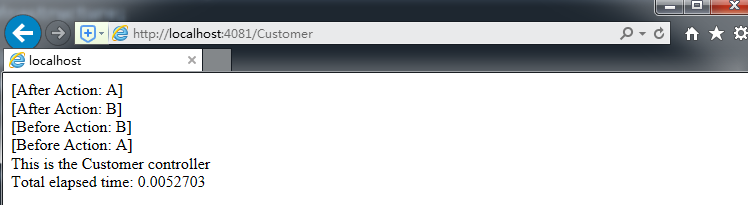
return "This is the Customer controller";

}

}

}

下面是这一过滤器的运行效果（/Customer）：



在该示例运行时，MVC框架在B过滤器在B过滤器之前执行A过滤器，但也可以按另一种方式执行。MVC框架不会保证任何特定的顺序或执行过程。大多数情况下，执行顺序是无关紧要的。但当有必要时，可以使用Order属性进行设置：

…

[SimpleMessage(Message = "A", Order = 2)]

[SimpleMessage(Message = "B", Order = 1)]

public string Index()

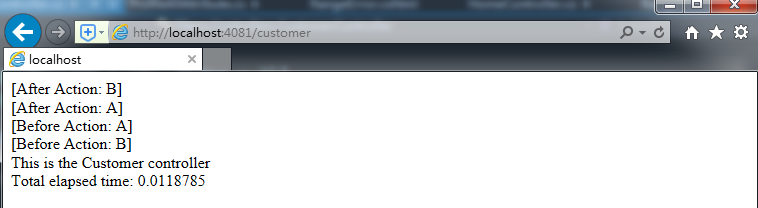
{

return "This is the Customer controller";

}

…

Order参数为一个int值，MVC框架以升序方式执行这些过滤器。比如示例中的给B指定了一个最小的值，所以将首先执行B过滤器，如图：



注：OnActionExecuting方法是按指定顺序执行的，但OnActionExecuted方法却以相反的顺序执行。在动作方法之前，MVC框架执行过滤器时会建立一个过滤器堆栈，并在随后释放这个堆栈。这种释放行为是不可改变的。

如果不指定Order值，将会为其设置一个-1的默认值。含义为：如果把有Order值和没有Order的过滤器混在一起，那些没有值的过滤器将优先执行，因为它们的Order值最低。

如果同类型的多个过滤器（如动作过滤器）具有相同的Order值（比如“1”），那么MVC框架会基于过滤器被运用的位置来决定顺序：全局过滤器 **>** 运用于控制器类的过滤器 **>** 运用于动作方法的过滤器。

注：异常过滤器的执行顺序是倒过来的。如果控制器和动作方法上以同样的Order值运用异常过滤器，动作方法上的（异常）过滤器将首先被执行。带有同样的Order值的全局异常过滤器最后被执行。

## 使用内建过滤器