# 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";

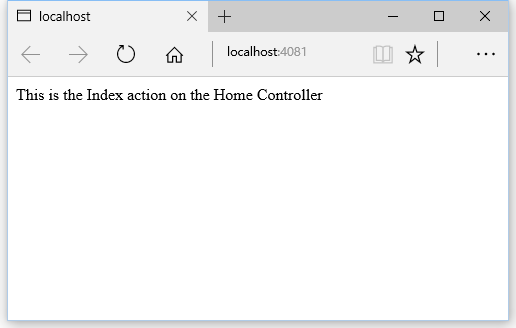
}

}

}

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。