Action执行包含内容比较多，主要有同步/异步Action的概念和执行过程，Authorationfilter, ActionFiltor, ResultFilter, ExceptionFilter等四个主要过滤器类型的执行过程。首先介绍异步的Action，之前学习Controller的时候已经知道默认情况下Controller的执行是异步的，在不继承异步Controller的情况，我们代码中的方法一般是同步的Action，我们可以通过使用Task<ActionResult>类型的返回值和在Action方法使用Task.Factory.StartNew()等方法来调用异步的Action，主要用于I/O绑定等操作。ASP.NET是通过线程池的机制来处理并发的HTTP的请求的，这种方式的优点是：工作线程的重用，减少线程的创建和释放；限制工作线程数量，避免高并发时服务器的崩溃。这里省略MVC4版本前的老式异步Action调用，Task返回值的Action如下所示：

|  |
| --- |
| public Task<ActionResult> Article(string name)  {  return Task.Factory.StartNew(() =>  {  string path = ControllerContext.HttpContext.Server.MapPath(string.Format(@"\articles\{0}.html", name));  using (StreamReader reader = new StreamReader(path))  {  AsyncManager.Parameters["content"] = reader.ReadToEnd();  }  }).ContinueWith<ActionResult>(task =>  {  string content = (string)AsyncManager.Parameters["content"];  return Content(content);  });  } |

在上代码中，可以看到一个AsyncManager类，它起到了在异步操作和回调操作间传递参数的作用。这是一个关于异步操作很重要的类型，其属性OutstandingOperatons是一个异步操作计数器，类似信号量的概念，用Increment设置初始值，当一个或多个异步操作完成时递减，为0时表示有所操作已完成，出发Completed事件，调用Finish方法。需要注意的细节是设置初始值的方法需要放在异步操作的外部，异步操作的超时时间可以通过AsyncTimeoutAttribute特性的Duration属性来设置。

接下来，介绍Action的执行过程，在Controller中，包括Model绑定和验证在内的整个Action的执行是通过一个名为ActionInvoker的组件来完成的，也包含同步异步两个版本，实现类为ControllerActionInvoker和AsyncControllerActionInvoker。这个简单介绍一下Controller在选择ActionInvoker时的步骤：通过DependencyResolver以IAsyncAcionInvoker查；以IActionInvoker查；创建异步类型作为默认。不同的ControllerActionInvoker会创建其对应的ControllerDescriptor实现类，包含对应类型的ActionDescriptor。还有一点需要注意的是，Dependency默认使用会将反射创建的对象缓存到CurrentCache属性中，而不会使用当前新设置的映射重新获取。若想在程序中修改，需要手动的清空CurrentCache所对应类型中的\_cache字段，部分代码如下所示：

|  |
| --- |
| private void ClearCachedActionInvokers()  {  var property = typeof(DependencyResolver).GetProperty("CurrentCache", BindingFlags.NonPublic|BindingFlags.Static);  var cacheActionInvoker = property.GetValue(null, null);  FieldInfo field = cacheActionInvoker.GetType().GetField("\_cache", BindingFlags.NonPublic|BindingFlags.Instance);  var dictionary = field.GetValue(cacheActionInvoker) as ConcurrentDictionary<Type, object>;  dictionary.Clear();  } |

在介绍筛选器的执行之前，再回顾一下相关过程，目标Action方法的最终执行是由被激活Controller的ActionInvoker决定，ActionInvoker通过调用对应的ActionDesciptor来执行被它描述的Action。筛选器使用面向切面概念(AOP)的实现，它会在在Action方法执行的前后自动执行，主要包含非业务逻辑的实现，例如授权，异常处理等。Filter作为基类包含FilterScope和Order属性，Scope包括First、Global、Controller、Action和Last，Order越小优先级越高，默认值为-1。同样Filter也有相应的Provider类，框架中原生的有FilterAttributeFilterProvider，ControllerInstanceFilterProvider和GlobalFilterCollection，简介如下表所示：

|  |  |
| --- | --- |
| 类型 | 简介 |
| FilterAttribute, FilterAttributeFilterProvider | 方法GetFilter得到的每个Filter,对应的FilterAttribute特性作为其Instance属性，Scope属性取决于FilterAttribute特性是应用在Controller类型上还是Action方法上。 |
| Controller, ControllerInstanceFilterProvider | Controller实现了IActionFilter,IAuthorization等四接口，本身就是一个筛选器，通过ControllerInstanceFilterProvider类型来表示针对Controller对象这种特殊筛选器的Filter。它的GetFilter方法根据ControllerContext获得对应Controller，并作为Filter的Instance属性，其Scope为First，Order为Int32.MinValue，默认最先执行。 |
| GlobalFilterCollection | 全局的Filter通过GlobalFilter.Filters.Add方式来添加，默认Scope为Global。 |

在筛选器的执行顺序上，遵循先Order排序，再Scope排序，若同一筛选器特性标注在不同Scope上且AllowMultiple为false时，会选中最后的一个执行。框架使用一个FilterInfo类型统一管理内置的筛选器，之后开始按照执行顺序详细介绍各个内置的筛选器。

**AuthorizationFilter**，实现IAuthorizationFilter的OnAuthoration方法用于实现授权操作，成功后继续Action后续工作（Model绑定，验证，Action的执行），失败后AuthorizationContext对象的Result属性回复一个”401，Unauthorized”相应或者重定向到错误页面。它所对应的几个实现IAuthorizationFilter接口的如下表所示：

|  |  |
| --- | --- |
| 类型 | 简述与例子 |
| AuthrizeAttribute | 多个Authorize特性间是“逻辑与”得关系,如下代码任何用户均无法访问。  [Authorize(Users = "Xixi", Roles="Admin")]  [Authorize(Users = "XiongEr", Roles = "Admin")]  public void CannotCall() { } |
| RequiredHttpsAttribute | 要求请求为HTTPS形式(HttpRequest.IsSecureConnection)，不满足时，如果是GET方式请求返回RedirectResult重定向请求，如果是其他方式抛出异常。 |
| ValidateInputAttribute | 在Controller, Action级别上针对整个请求决定输入参数是否进行验证。例如在QueryString中放入”<script></script>” |
| ValidateAntiForgeryTokenAttribute | 防止CSRF(Cross-Site Request Forgery)跨站请求伪造网络攻击，如果说XSS(Cross Site Script)是利用了用户对网站的信任，那个CSRF就是利用了站点对认证用户的信任。在之后的内容中，将继续介绍CSRF的原理和框架的预防方法。 |
| ChildActionOnlyAttribute | 一般用于生成组成页面的某部分HTML，若非子Action则抛出异常。（通过DataTokens中是否包含ParentActionViewContext判断） |

接下来用蒋老师介绍的简单例子来解释CSRF的原理，假设我们奖励一个博客应用，作为博主的我们可以发表博文，而一般用户（包括匿名）可以评论。除此之外注册用户可以修改自己的绑定Email，我们将授权特性加在该Action上，看起来应该OK了，但仍然有漏洞可钻，如下图所示



从上图可知，通过跳转攻击者获得用户安全令牌，通过了授权验证，说明CSRF是一种隐蔽且危害巨大的攻击，框架通过ValidateAntiForgeryTokenAttribute结合HtmlHelper的AntiForgeryToken方法有效解决了这个问题。在View中通过调用AntiForgeryToken方法，在页面中生一个值为防伪令牌字符串的hidden类型的<input>元素，并且设置一个具有HttpOnly的Cookie。防伪令牌值通过Salt，Creation，Username等内容计算得出。Cookie的名称通过应用路径base64编码值加上\_RequestVerificationToken组合而成。对于加入防伪令牌的View在第一次访问或者Cookie不存在时，创建Cookie并设置HttpOnly标签，这样浏览器就无法通过脚本获得Cookie，保证了Cookie的安全。再次请求时，解密和反序列化Hidden与Cookie中相关值，比较属性即可。

**ActionFilter**之前介绍过的实现类包括AsyncTimeoutAttribute等，允许我们对Action执行前后添加一些额外操作，通过Result属性相应气请求。筛选器中OnActionExecuting与OnActionExecuted的执行顺序相反。正向执行时，一旦某一个ActionFilter将AcionExecuteingContext的Result设置为ActionResult对象，后续ActionFilter和目标Action将不会执行。而在逆向执行ActionFilter链时在ActionExecutedContext中设置Result不受影响，如下图所示：



ActionFilter链的异常处理过程通过对应的上下文类的Exception对象传递，ExceptionHanlded属性表明异常是否已被处理。

**ExceptionFilter**既可以处理ActionFilter最终抛出的异常，还可以处理ResultFilter抛出的异常。其中实现类HanldeErrorAttribute用于针对具体的异常类型来呈现对应的错误页面。同时由于ExceptionFilter链的反向执行特性，需要设置Order属性使得具体的HanldeErrorAttribute优先执行。例子如下：

|  |
| --- |
| [HandleError(ExceptionType = typeof(Exception1), View = "ErrorView1", Order = 3)]  [HandleError(ExceptionType = typeof(Exception2), View = "ErrorView1", Order = 2)]  [HandleError(Order = 1)]  public ActionResult Bar() |

需要注意的一点是，HandleErrorAttribute只有在允许自定义错误时才有效。

|  |
| --- |
| <customErrors mode="On"/> |

蒋老师在书中提到，异常处理是程序员最熟悉也最难掌握的一块概念了，我确实也有这样的感受，比如说一个异常类型到底“谁来管，该怎么管，管不住怎么办“，很像法制建设，需要一定的规定，但软件开发中还未有相关的通用规则。由于异常处理往往是场景驱动的，就需要一个灵活可配置的处理框架进行管理，例如微软企业库Entlib的Exception Handling Application Block（EHAB）。该库提供一种基于“策略”的异常处理方式。之后还提供了一个自动化处理异常的思路，即通过配置，自动生成try/catch过程。

简单来说就是：

异常处理策略=异常类型+异常处理器+异常后续处理方式，例子如下所示：

|  |
| --- |
| <configSections>  <section name="exceptionHandling" type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Configuration.ExceptionHandlingSettings, Microsoft.Practices.EnterpriseLibrary.ExceptionHandling"/>  </configSections>  <exceptionHandling>  <exceptionPolicies>  <add name="data access policy">  <exceptionTypes>  <add type="System.Data.SqlClient.SqlException, System.Data" postHandlingAction="ThrowNewException" name="SqlException">  <exceptionHandlers>  <add name="Logging.Handler"  type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Logging.LoggingExceptionHandler"/>  <add name="Replace Handler"  type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.ReplaceHandler"  exceptionMessage="Encounter data access error."  replaceExceptionType="Xixi.DbException"/>  </exceptionHandlers>  </add>  </exceptionTypes>  </add>  </exceptionPolicies>  </exceptionHandling> |

相应的调用方式为：

|  |
| --- |
| try { test001(); }  catch (Exception ex)  {  if (ExceptionPolicy.HandleException(ex, "data access policy"))  {  throw;  }  } |

**ResultFilter**用于控制ActionResult的执行，属于在Action方法执行过后对ActionResult执行过程的控制，也就是对视图渲染的控制了，内容与ActionFilter相似，就不介绍了。

注：本文主要供自己学习，不妥之处望见谅。

参考资料：

[1]蒋金楠. ASP.NET MVC4框架揭秘[M]. 上海:电子工业出版社, 2012. 320-389