**[Web APi之认证（Authentication）及授权（Authorization）【一】（十二）](http://www.cnblogs.com/CreateMyself/p/4856133.html)**

**http://www.cnblogs.com/CreateMyself/p/4856133.html**

**前言**

无论是ASP.NET MVC还是Web API框架，在从请求到响应这一过程中对于请求信息的认证以及认证成功过后对于访问页面的授权是极其重要的，用两节来重点来讲述这二者，这一节首先讲述一下关于这二者的一些基本信息，下一节将通过实战以及不同的实现方式来加深对这二者深刻的认识，希望此文对你有所收获。

**Identity**

Identity代表认证用户的身份，下面我们来看看此接口的定义

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public interface IIdentity  {      // Properties      string AuthenticationType { get; }        bool IsAuthenticated { get; }        string Name {get; }  } |

该接口定义了三个只读属性， AuthenticationType  代表认证身份所使用的类型， IsAuthenticated 代表是否已经通过认证， Name 代表身份的名称。对于AuthenticationType认证身份类型，不同的认证身份类型对应不同的Identity，若采用Windows集成认证，则其Identity为WindowsIdentity，反之对于Form表单认证，则其Identity为FormsIdentity，除却这二者之外，我们还能利用GenericIdentity对象来表示一般意义的Identity。

* **WindowsIdentity**

在WindowIdentity对象中的属性Groups返回Windows账号所在的用户组，而属性IsGuest则用于判断此账号是否位于Guest用户组中，最后还有一个IsSystem属性很显然表示该账号是否是一个系统账号。在对于匿名登录中，该对象有一个IsAnonymous来表示该账号是否是一个匿名账号。并且其方法中有一个GetAnonymous方法来返回一个匿名对象的WindowsIdentity对象，但是此WindowsIdentity仅仅只是一个空对象，无法确定对应的Windows账号。

* **FormsIdentity**

我们来看看此对象的定义

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class FormsIdentity : ClaimsIdentity  {        public FormsIdentity(FormsAuthenticationTicket ticket);        protected FormsIdentity(FormsIdentity identity);        public override string AuthenticationType { get; }        public override IEnumerable<Claim> Claims { get; }        public override bool IsAuthenticated { get; }        public override string Name { get; }        public FormsAuthenticationTicket Ticket { get; }        public override ClaimsIdentity Clone();  } |

一个FormsIdentity对象是通过加密过的认证票据（Authentication Ticket）或者是安全令牌（Security Token）来创建，被加密的内容或者是Cookie或者是请求的URl，下述就是通过FormsIdentity来对Cookie进行加密。

var ticket = new FormsAuthenticationTicket(1, "cookie", DateTime.Now, DateTime.Now.AddMinutes(20), true, "userData", FormsAuthentication.FormsCookiePath);

var encriptData = FormsAuthentication.Encrypt(ticket);

* **GenericIdentity**

以上两者都有其对应的Identity类型，如果想自定义认证方式只需继承该类即可，它表示一般性的安全身份。该类继承于IIdentity接口。至于如何判断一个匿名身份只需通过用户名即可，若用户名为空则对象的属性IsAuthenticated为true，否则为false。

**Principal**

**这个对象包含两个基本的要素即基于用户的安全身份以及用户所具有的权限，而授权即所谓的权限都是基于角色而绑定，所以可以将此对象描述为：【身份】+【角色】。**

首先我们来看看此接口

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public interface IPrincipal  {        bool IsInRole(string role);        IIdentity Identity { get; }  } |

上述基于IIdentity接口的实现即WindowsIdentity和GenericIdentity，当然也就对应着Principal类型即WindowsPrincipal和GenericPrincipal，除此之外还有RolePrincipal，关于这三者就不再叙述，我们重点来看看下APiController中的IPrincipal属性。

**APiController中User**

我们看看此User属性

|  |  |
| --- | --- |
| 1 | public IPrincipal User { get; } |

继续看看此属性的获取

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public IPrincipal User  {      get      {          return Thread.CurrentPrincipal;      }  } |

到这里还是不能看出什么，即使你用VS编译器查看也不能查看出什么，此时就得看官方的源码了。如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public HttpRequestContext RequestContext  {      get      {          return ControllerContext.RequestContext;      }      set      {......}  }    public IPrincipal User  {      get { return RequestContext.Principal; }      set { RequestContext.Principal = value; }  } |

到这里我们看出一点眉目了

**IPrincipal的属性User显然为当前请求的用户并且与HttpRequestContext中的属性Principal具有相同的引用。**

**那么问题来了，HttpRequestContext又是来源于哪里呢？**

我们知道寄宿模式有两种，一者是Web Host，另一者是Self Host，所以根据寄宿模式的不同则请求上下文就不同，我们来看看Web Host中的请求上下文。

* **Web Host**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | internal class WebHostHttpRequestContext : HttpRequestContext   {       private readonly HttpContextBase \_contextBase;       private readonly HttpRequestBase \_requestBase;       private readonly HttpRequestMessage \_request;           public override IPrincipal Principal       {           get           {               return \_contextBase.User;           }           set           {               \_contextBase.User = value;               Thread.CurrentPrincipal = value;           }       }  } |

从这里我们可以得出一个结论：

**Web Host模式下的Principal与当前请求上下文中的User具有相同的引用，与此同时，当我们将属性Principal进行修改时，则当前线程的Principal也会一同进行修改。**

* **Self Host**

我们看看在此寄宿模式下的对于Principal的实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | internal class SelfHostHttpRequestContext : HttpRequestContext  {      private readonly RequestContext \_requestContext;      private readonly HttpRequestMessage \_request;         public override IPrincipal Principal      {          get          {              return Thread.CurrentPrincipal;          }          set          {              Thread.CurrentPrincipal = value;          }      }    } |

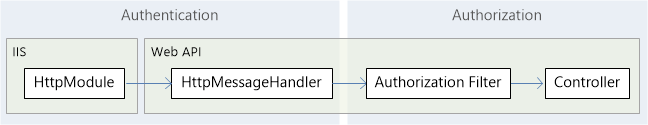
在此模式我们可以得出结论：

**Self Host模式下的Principal默认是返回当前线程使用的Principal。**

接下来我们来看看认证（Authentication）以及授权（Authorization）。

**AuthenticationFilter**

AuthenticationFilter是第一个执行过滤器Filter，因为任何发送到服务器请求Action方法首先得认证其身份，而认证成功后的授权即Authorization当然也就在此过滤器之后了，它被MVC5和Web API 2.0所支持。下面用一张图片来说明这二者在管道中的位置及关系



接下来我们首先来看看第一个过滤器AuthenticationFilter的接口IAuthenticationFilter的定义：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public interface IAuthenticationFilter : IFilter  {        Task AuthenticateAsync(HttpAuthenticationContext context, CancellationToken cancellationToken);          Task ChallengeAsync(HttpAuthenticationChallengeContext context, CancellationToken cancellationToken);  } |

**该接口定义了两个方法，一个是 AuthenticateAsync ，它主要认证用户的凭证。另一个则是 ChallengeAsync ，它主要针对于在认证失败的情况下，向客户端发送一个质询（Chanllenge）。**

以上两者关于认证的方法都分别对应定义在Http协议的RFC2612以及RFC2617中，并且两者都是基于以下两点：

* **如果客户端没有发送任何凭证到服务器，那么将返回一个401（unauthorized）响应到客户端，在返回到客户端的响应中包括一个WWW-Authenticate头，在这个头中包含一个或多个质询，并且每个质询将指定被服务器识别的认证组合。**
* **在从服务器响应返回一个401到客户端后，客户端将在认证头里发送所有的凭证。**

下面我们详细查看每个方法的内容：

**AuthenticateAsync**

此方法中的参数类型为HttpAuthenticationContext，表示为认证上下文，我们看看此类的实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public class HttpAuthenticationContext  {        public HttpAuthenticationContext(HttpActionContext actionContext, IPrincipal principal)      {          if (actionContext == null)          {              throw new ArgumentNullException("actionContext");          }            ActionContext = actionContext;          Principal = principal;      }          public HttpActionContext ActionContext { get; private set; }        public IPrincipal Principal { get; set; }        public IHttpActionResult ErrorResult { get; set; }        public HttpRequestMessage Request      {          get          {              Contract.Assert(ActionContext != null);              return ActionContext.Request;          }      }  } |

**在构造函数中通过Action上下文和认证的用户的Principal属性进行初始化，而属性ErrorResult则返回一个HttpActionResult对象，它是在认证失败的情况直接将错误消息返回给客户端。我们应该能想到请求到Action方法上的AuthenticationFilter可能不止一个，此时Web API会通过FilterScope进行排序而形成一个AuthenticationFilter管道，紧接着认证上下文会通过当前的Action请求上下文以及通过APiController的User属性返回的Principal而被创建，最终认证上下文会作为AuthenticationFilter中的AuthenticateAsync方法的参数并进行调用。**

当执行为AuthenticateAsync方法被成功执行并返回一个具体的HttpActionResult，此时后续操作将终止，接下来进入第二个方法即【发送认证质询】阶段。

**ChallengeAsync**

**绝大多数认证基本上都是采用【质询-应答】方式，服务器向端客户端发出质询要求来提供凭证，若客户端在执行AuthenticateAsync方法后，认证未成功，此时服务器端将通过ChallengeAsync方法发送认证质询。**

接下来我们来看看此方法的具体实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | public class HttpAuthenticationChallengeContext  {      private IHttpActionResult \_result;        public HttpAuthenticationChallengeContext(HttpActionContext actionContext, IHttpActionResult result)      {          if (actionContext == null)          {              throw new ArgumentNullException("actionContext");          }            if (result == null)          {              throw new ArgumentNullException("result");          }            ActionContext = actionContext;          Result = result;      }        public HttpActionContext ActionContext { get; private set; }        public IHttpActionResult Result      {          get          {              return \_result;          }          set          {              if (value == null)              {                  throw new ArgumentNullException("value");              }                \_result = value;          }      }        public HttpRequestMessage Request      {          get          {              Contract.Assert(ActionContext != null);              return ActionContext.Request;          }      }  } |

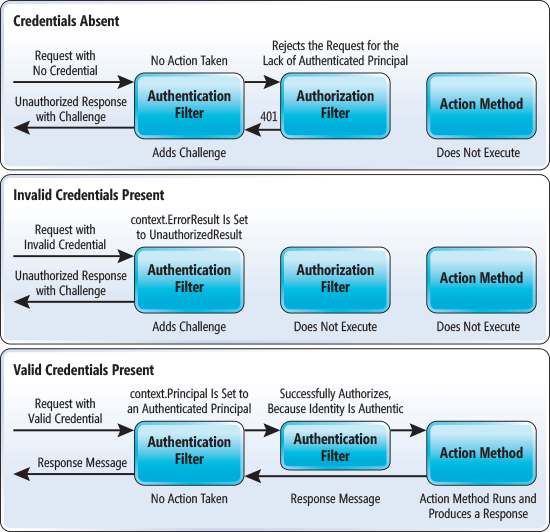
**很显然，当调用AuthenticateAsync方法完成认证工作后，此时ErrorResult将返回一个具体的HttpActionResult，然会将Action上下文以及具体的HttpActionResult传递到构造函数中进行初始化HttpAuthenticationChallgeContext，最后依次调用ChallengeAsync方法，当此方法都被执行后，此类中的Result属性也就返回一个具体的HttpActionResult对象，此对象将创建相应的HttpResponseMessage对象，并回传到消息处理管道中并作出响应从而发出认证质询。**

**总结**

（1）在Web API中使用AuthenticationFilter进行认证主要是以下三步

* **Web API会为每个需要被调用Action方法创建所有可能的AuthenticationFilter列表，若有多个则通过FilterScope来进行排序，最终形成AuthenticationFilter管道。**
* **Web API将为AuthenticationFilter管道中的每一个过滤器依次调用AuthenticateAsync方法，在此方法中每个AuthenticationFilter将验证来自客户端的Http请求凭证，即使在认证过程中触发到了错误，此时进程也不会终止。**
* **若认证成功，Web API将调用每个AuthenticationFilter的ChallengeAsync方法，接下来每一个AuthenticationFilter将通过此方法做出质询响应。**

（2）通过上述描述我们用三张示意图来对照着看



**认证方案**

我们知道Http协议中的认证方案有两种，一种是Basic基础认证，一种是Digest摘要认证

**Basic基础认证**

**此认证是在客户端将用户名和密码以冒号的形式并用Base64明文编码的方式进行发送，但是不太安全，因为未被加密，在此基础上采用Https信息通道加密则是不错的认证方案。**

**Digest摘要认证**

**此认证可谓是Basic基础认证的升级版，默认是采用MD5加密的方式，在一定程度上算是比较安全的，其执行流程和Basic基础认证一样，只是生成的算法不同而已。**

**未完待续：接下来将通过认证方案手动通过不同的方式来实现认证。。。。。。**