ASP.NET MVC和WebAPI已经是.NET Web部分的主流，刚开始时两个公用同一个管道，之后为了更加的轻量化(WebAPI是对WCF Restful的轻量化)，WebAPI使用了新的管道，因此两者相关类的命名空间有细微差异，在使用时需要注意。

Tip:文中是Pxx表示PDF页码，请无视哈

* WebAPI与ASP.NET路由的异同

1. ASP.NET MVC的路由：Routes(RouteCollection)的线程安全，读写锁，GetReadLock, GetWriteLock。RouteTable.Routes.MapPageRoute(…);

命名空间为System.Web.Routing中，P63

1. WebAPI的路由：首先介绍其相关类型，他们均是对Http报文的简易封装，System.Net.Http(HttpRequestMessage, HttpResponseMessage)。

命名空间为System.Web.**Http**.Routing中, P94

1. 两个路由的衔接，例如在Web Host模式中将WebAPI寄宿于一个Web应用时，其最终的URL路由还是通过ASP.NET本身的路由系统完成，几个主要的类型，HttpControllerRouteHandler,

HttpRouteHandler。

* 消息处理管道

还记的ASP.NET MVC中的核心是HttpHandler，而在WebAPI中其管道处理器是HttpMessageHandler。在实际中其通过职责链模式将委托通过InnerHandler(DelegationHandler)方式进行处理。其中，其中这个管道最开始的是httpServer，最末端的是HttpRoutingDispatcher(均在System.Web.Http命名空间下，支持异步模型)，P108



Tip:额外想想也能理解WebAPI管道为什么更加轻量化，因而它只需要处理Json等类型数据，不需要考虑如页面、JS、静态资源等内容。

* 常见特性

Class: [RoutePrefix(“api/demo”)]，针对具体类的路由设置，相对RouteConfig，粒度更细。

Method: [Route(“action”)]

[HttpGet], [HttpPost]

关于Web服务，其中比较难的概念一般都集中在安全，其相关概念非常的多，包括Windows相关认证模式、Forms认证、第三方认证、跨域访问等，接下来一一介绍。此外还会附加HttpClient、IOC框架的选择、服务幂等性、SignalR、EntLib中的EHAB等概念。

.NET安全模型：

Identiy表示用户身份， Identity AuthenticationType, IsAuthenticated, Name}，常见的Identity有WindowsIdentity, FormsIdentity, GenericIdentity。P556

IPrincipal, 被成功实施授权的实体，等价于身份加角色，包括WindowsPrincipal（windows的权限组）,GenericPrincipal, RolePrincipal(Membership组件和Roles组件) 。常见的认证方式通过“质询-应答”(challenge-Response)方式。

常见http认证方式，Basic和Digest，前者使用将认证凭证（用户名+密码）通过base64编码而未加密，但我们可以使用https传输来解决机密性问题。与此相关的两个过滤器， AuthenticationFilter和AuthorizationFilter。

补充ActionFilter概念，比如请求涉及大量运算，并且输入和输出一一对应（即相同的输入有相同的输出），那么可以考虑缓存Action。P585

Windows认证模式（均通过在IIS中设置身份认证模式）

WebHost寄宿下的安全：Windows认证模式，通过Basic, Digest认证方案，最终采用NTLM或者Kerberos协议。认证用户的Principal体现在HttpContext、当前线程、ApiControlelr。Keep-Alive，Fidder查看调用。

|  |  |  |
| --- | --- | --- |
| 名称 | 状态 | 响应类型 |
| Active Directory客户端证书身份验证 | 已禁用 | HTTP 401 质询 |
| ASP.NET 模式 | 已禁用 |  |
| Forms身份验证 | 已禁用 | HTTP 302 登录/重定向 |
| Windows身份验证 | 已禁用 | HTTP 401 质询 |
| 基本身份验证(Windows/Basic) | 已禁用 | HTTP 401 质询 |
| 匿名身份验证 | 已禁用 |  |
| 摘要式身份验证（Windows/digest） | 已启用 | HTTP 401 质询 |

* Basic认证, P602

现在都是HTTP401 质询模型，只有forms是http 302 登录/重定向。这个关于basic的质询方式很有意思，就是当你请求时，出现http 401，会要求你输入用户名密码，输入后你输入的用户名和密码会被base64编码发送的服务器，形式是Basic YWRtaW46YWRtaW4=，这部分的head就是authentication。查看windows的凭据管理器，账号密码木有问题，但仍然不能通过验证，非常的伤感，自己试着加上域cn1\，结果OK了，感觉棒棒哒，哈哈，说明asp.net安全模型和windows有很好的整合性。

Basic模式的流程是，浏览器向服务器IIS以匿名的方式发送GET请求，IIS回复一个401 Unauthorized的响应，该响应用“www-authenticate”报头告诉客户端采用的认证方案(basic)和对应的领域(Realm, localhost)。浏览器收到响应弹出登录对话框，收集输入的账号密码组成字符串作为认证凭证，接下来，浏览器再次发送请求，在authorization包头中携带认证的方案和用户的凭证Basic YWRtaW46YWRtaW4=，IIS解密后认证，action顺利执行。

Base64：是网络常见的用于传输8bit字节代码的编码方式，用在http表单（包括隐藏的表单域）和http GET url中，base64编码的信息具有不可读性，但不具有机密性，使用时需要注意应用场景。

* Digest认证, P607

Digest认证仅仅适用于Domain模式，如果基于WorkGroup模式，也无法使用，接下来通过fiddler看看相应的HTTP消息头。

HTTP 401的响应：

WWW-Authenticate: **Digest** **qop**="auth",**algorithm**=MD5-sess,**nonce**="+Upgraded+v1e4fcae181b935afc3d94f30f5033141a25e3c7e4b83bd101c60cf10ea425a352c8959c8d47e643e5fc38f90cffe411be5f7a99033900ae4d",charset=utf-8,**realm**="Digest"

Digest认证传输用户凭证的哈希码，而不是明文。客户端首先匿名向服务器发送GET请求，服务器返回一个401响应，这个响应包含一个”WWW-Authenticate”报头，携带的信息包括。

|  |  |
| --- | --- |
| 参数 | 解释 |
| Digest | 认证方案 |
| Realm=”Digest” | 领域 |
| Algorithm | 表示服务端支持的哈希算法，MD5-sess |
| Nonce | 服务端生成的唯一性支付穿，一般来说，IIS会利用当前时间戳以及请求的Etag(被请求变量的实体值)报头值来生成这个nonce |
| Qop(Quanlity of Protection) | IIS采用qop通知客户端采用的消息保护等级，可选值包括auth(authentication), auth-int(authentication-integrity)，前者仅限于基本的认证，后者还确保传输内容的一致性。 |

输入账号密码的再一次请求（响应为200成功）：

Authorization: **Digest** **username**="cn1\\she\_s", **realm**="Digest", **nonce**="+Upgraded+v1e4fcae181b935afc3d94f30f5033141a25e3c7e4b83bd101c60cf10ea425a352c8959c8d47e643e5fc38f90cffe411be5f7a99033900ae4d", uri="/Sory.Entertainment.WebAPI/", **algorithm**=MD5-sess, **response**="9a6cb99fad4404cdd521e5db432f6b09", **qop**="auth", nc=00000001, **cnonce**="c77c05d93544b363"

其相关参数为：

|  |  |
| --- | --- |
| 参数 | 解释 |
| Username | 代表客户端的用户名，看来用户名还是可以截取的 |
| Qop | 最终采用的消息保护等级， qop=”auth” |
| Algorithm | 最终采用的加密算法，MD5-sess |
| Nonce | 实际就是服务器端生成的nonce |
| cnonce | 客户端生成的nonce(c代表client)，它可以对请求内容签名以确保内容未被篡改，可以帮助客户端对服务端实施认证（服务端能够证明知道该nonce，这些很适合防御跨站请求伪造的防御） |
| Nc(nonce count) | 它表示客户端针对同一个nonce发送请求的数量，一位着这是随着请求不断增加的数字，IIS可以通过nc代表的数字来防止“重放攻击”，它会维护每个nonce的nc，如果请求携带的nc比这个少，会被认为是不合法的请求。 |

IIS在接受到第二次请求后，它先对请求进行合法性校验（比如nc的合法性），然后从Authentication报头提取用户名、nonce和加密算法计算出针对用户名真正的Digest，最终利用它与请求中提供的Digest进行比较确认密码的正确性，完成客户端认证。

Tip:

<script type="application/json" id="\_\_browserLink\_initializationData">

{"appName":"Firefox","requestId":"ee1fc1d3f30e4b4cba937703bee3ce10"}

</script>

<script type="text/javascript" src="http://localhost:13820/69ea419b05a141aaa5111affa4bb02fe/browserLink" async="async"></script>

这部分如何理解，与jsonP相关？

* 集成Windows认证， P610

无论问basic还是Digest认证，如果使用浏览器做客户端，第一次访问总需要在弹出框中输入，非常繁琐，并且密码在网络中传输，有安全风险，一般采用**加盐**的方式避免。集成Windows认证可以很好解决该问题，它默认以登录机器的Windows账号的名义来访问被授权的资源没，用户的密码被包含在请求携带的安全令牌中，非常的方便，该方式最终使用NTLM和Kerberos协议来完成。

NTLM协议（比较陈旧）：采用质询/应答（Challenge/Response）消息交换模式，DC域控制器保存所用用户的相关信息。基本流程为：步骤1，用户输入账户密码登录主机，主机会缓存输入密码的哈希值，原始密码会丢失。如果视图访问服务器资源，需要向对方发送请求，请求中包含一个明文表示的用户名；步骤2，服务端接受请求，生成16位随机数（称为质询challenge），存储起来后以明文的形式发送给客户端；和Digest请求中nonce的意图完全一致；步骤3，客户端收到服务端的质询后，用在步骤1中保存的密码哈希值对其加密，然后将加密后的质询发送给服务端；步骤4，服务端收到加密质询后，会向DC发送针对客户端的验证请求（请求中包括，用户名、客户端密码加密后的质询和原始的质询）；步骤5、6，DC根据用户名获得密码哈希值，对原始质询加密，再与服务端发送的质询比较，一致就为验证通过，否则失败。

Kerberos:这东西也算是困扰了小弟很多年，老看到，尤其每次注册windows时，呵呵，你懂得。实际上它是一种更搞笑安全的认证协议，过程更加的复杂，与NTLM相似，也包含三部分，客户端、服务器和KDC（Key Distribution Center，在windows域中，KDC有DC担当）。Kerberos实际是一种基于票据(Ticket)的认证方式，客户端要访问服务端的资源，首先要购买服务端认可的票据。也就是说，客户端在访问服务器前要先买好票，等待服务器验票后才能入场，但这票不能直接购买，首先需要认购权证（和粮票，股票认购权证相似）。这个认购权证和进入服务器的入场券均由KDC发售，感觉各种绕……相关详细内容暂时放一放。

在IIS中使用windows集成验证时，会看到provider的设置，有”negotiate”和“NTLM”两个选项，默认使用前者，其Provider包括”Negotiate: Kerberos”,当然也可以自定义。此外，客户端需要在IE设置-》高级中，开启Windows集成认证，默认是开启的。在使用HttpClient时，可以使用以下方式，简化调用。

|  |
| --- |
| HttpClientHandler handler = new HttpClientHanlder();  方式1: Handler.UseDefaultCredentials = true;  方式2: handler.Credentials = new NetworkCredentials(“username”, “password”, “domainname”);  Using(HttpClient client = new HttpClient(handler)){} |

可与独立于windows系统的认证方式，以前做webform时，forms认证是用的最多的，当时还一直以为forms验证也需要和windows相关，尤其是和webForm中的form相关，现在想想挺幼稚的，同时这个验证可以和membership很好的搭配在一起。但实际上这种验证方式是独立的，适合自行维护用户账号和密码的场景，也是绝大部分项目的场景。那么接下来介绍forms认证是如何进行的，努力使自己真正的走出误区。

Forms认证的流程设计4次的消息交换，其具体步骤如下所示。

步骤1：用户通过浏览器匿名向IIS发起请求，假设地址为“/home”，它会收到状态为”302, Found”的相应，这是一个用于实现“重定向”的http响应，它通过location报头表示的重定向地址指向登录的页面，之前访问的地址将作为查询字符串returnURL的值。

步骤2：浏览器接受到该请求后，针对重定向的地址发送请求，登录页面最终被呈现在浏览器。

步骤3：用户输入正确的用户名密码后提交表单，服务器在接受到请求之后提取它们对用户实施认证，认证成功后，它会生成一个安全令牌或者认证票据。接下来，通过查询字符串**returnURL**表示的原始请求地址，作为另一个状态为”302, Found”响应的Location报头，而经过加密/签名的**安全令牌作为该响应的Cookie**。

步骤4：这个**代表安全令牌的Cookie**将自动附加到浏览器后续的请求中，服务器直接利用它对请求实施认证。**Cookie的名称、过期策略以及采用的保护等级**均可以通过配置来控制。在禁用Cookie的情况下，安全令牌会直接作为URL的一部分传送。

Tip：

首先想补充补充的是原来的forms认证的配置通过如下配置，加上在login相关方法上加上[AllowAnonymous]，然后IIS中设置启用匿名认证和forms认证即可。

|  |
| --- |
| <authentication mode="Forms" >  <forms loginUrl="~/account/login"/>  </authentication>  <authorization>  <deny users="?"/>  </authorization> |

在ASP.NET 5之后的版本配置方式有一些变化，为了和第三方认证OAuth集成，不需要配置文件的配置，而是通过如下代码配置，如果添加往往会出错。

app.UseCookieAuthentication(new CookieAuthenticationOptions

{

AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,

LoginPath = new PathString("/Account/Login")

});

Cookie采用的保护等级，在IE设置中包括6个**隐私**等级，对cookie的管理程度各不相同，从cookie完全不可读写，到完全可读写，默认的等级为中，阻止没有精简隐私策略的第三方cookie。

昨天和同事聊天时，还注意到有的项目的logoff是直接跳转页面，而不是action，因此缺少清空session等服务器端信息的操作，在实际开发中也算是个易错点。

之前介绍的认证方式，都要求密码(token)在网络中进行传输，为了确保密码不被窃取，需要用SSL\TLS对传输的内容实施保护。其中涉及很多安全相关的基础知识点，这儿只做简要介绍。

1. 非对称加密：保证消息机密性，涉及有一个公钥和密钥组成的密钥对。
2. 数字签名：保证身份认证、防止抵赖Non-repudiation、消息一致性
3. 数字证书：Digital Certificate, 也称为公钥证书Public Key Certificate，比如12306要求在客户端安装的服务器根证书，大部分基于X.509 V3证书标准，还可以称为X.509证书。其实际上就是将某个**密钥对**中的公钥与某个主体Subject进行绑定的文件，其内容包括版本号V3,序列号，**签名算法**（md5WithRSAEncryption）、**颁发者**（Issuer）、有效日期、主体信息、**主题公钥和公钥算法**，以及**颁发者的数字签名**。

Tip:对于数字证书想说的是，一定要把它才分开了理解，大体包含三部分，公钥信息、签名信息和其他信息。并且后面两者都是为前者的安全送达服务的，简而言之(如12306购买火车票场景，祝愿大家都能买到过个幸福年，哈哈)，网站通过要求用户安装根证书的方式将网站通信密钥对中的公钥发送给我，但为了保证这个过程的安全，就需要提供数字签名的过程。就像将情报通讯密码给我，并且签上了FBI一样，之后就可以用这个密码进行通讯了。这个说的比较粗略，有些简化，省略了认证权威机构和认证树的概念。

关于SSL/TLS的概念，后者TLS（Transport Layer Security）其实是前者SSL(Secure Sockets Layer)的升级版本，TLS1.0就是SSL3.1，在IE的设置中，可以看到默认支持SSL 3.0和TLS1.0。而HTTPS是指HTTP与SSL/TLS的结合，像之前介绍的12306的安全，就是https的，也称为弱安全模型，那么强安全模型是什么？那就是我们使用网银时，大家都经历安装安全控件甚至使用U盾的过程，这儿就是强安全。简单来说，强安全，指服务器端和客户端都要安装对方的证书，相互认证；弱安全，指客户端安装服务器证书，客户端认证服务器。接下来介绍请求Https网站的过程。

步骤1：客户端向https站点发送协商请求，包括客户端所支持的加密算法列表

步骤2：Https站点从算法列表中选择所能支持最合适安全级别的算法（安全性和效率折衷），连同**绑定到该站点的数字证书**一并发送给客户端。

步骤3：客户端接受证书后，通过验证确认站点身份，成功后，生成一个随机数，作为会话密钥(Session key)缓存在客户端。客户端采用发回的加密算法，利用证书中的公钥对该密钥(Session Key)加密，加密后发送给站点，站点解密获得Session key。

步骤4：客户端和服务端使用该Session key使用**对称加密算法**进行加解密。（对称加密效率高，但密钥管理难，因此采用结合两者的方式，用非对称加密管理密钥，用密钥来对称加密，棒棒哒）

* SSL/TLS在IIS中的应用

IIS对多种传输协议提供支持，包括http、Tcph和MSMQ等，**站点绑定数字证书**的方式哟很多，最方便的是用iis管理器，其步骤如下。

步骤1：在未目标站点添加https绑定之前，我们需要为它准备一张证书，可以用makeCert.exe工具，也可以使用iis管理器来创建自我签名的证书。在IIS的特性列表中选择“服务器证书”，之后选择“创建自我签名证书”，命名和站点名称相同即可。

步骤2：在IIS中，选择我们指定的站点（Web Site），右键选择编辑绑定，在网站绑定页面添加https类型并选择相应的证书，在浏览网站栏就可以看到http, 和https了。这时你就可以浏览网页通过两种不同的方式，当然你自定义的证书未被加入根证书，因此用https时，浏览器会显示一个小红叉。之后在httpclient部分，你也会发现，我们可以通过设置，跳过客户端对服务器证书的验证，方便调用，不过不推荐。

网站的常见调用可以通过http和https两种方式，但具体到某一个调用的时候，需要在“安全”和“性能”间权衡，但是认证过程必须采用https，将指定的action设置为[RequireHttps]，那么它就只能通过https协议访问到。该特性实际是MVC提供的一个AuthenticationFilter，如果是一个普通请求，则会把该请求重定向到https的相应地址。这儿大家会注意到一个问题就是requiredHttps是MVC下的概念，那么WebAPI中有对应概念么？这个可以通过自定义的认证过滤器来处理。

|  |
| --- |
| public override void OnAuthorization(System.Web.Http.Controllers.HttpActionContext actionContext)  {  //如果为https请求，授权通过  if (Uri.UriSchemeHttps == actionContext.Request.RequestUri.Scheme)  {  base.OnAuthorization(actionContext);  return;  }  //对于http-get请求，将schema替换成https进行重定向  if (HttpMethod.Get == actionContext.Request.Method)  {  var requestUri = actionContext.Request.RequestUri;  string location = string.Format("https://{0}/{1}", requestUri.Host, requestUri.LocalPath.TrimStart('/'));  var actionResult = new RedirectResult(new Uri(location), actionContext.Request);  actionContext.Response = actionResult.ExecuteAsync(new CancellationToken()).Result;  }  //采用其他http方法的请求直接认为是bad request  actionContext.Response = new HttpResponseMessage(HttpStatusCode.BadRequest) { ReasonPhrase = "SSL Required" };  } |

一般来说，web应用的用户认证均由自身完成，通过存储用户名和密码并进行验证，但这种方式在当前的互联网场景下会有一下两个主要问题：用户需要注册不同的账号，记住和使用非常的麻烦了；对于应用提供者，大量认证系统会花费大量的精力。基于OAuth2.0(Open Authentication 2.0)的第三方认证模型的出现正好可以解决这个痛点，该模型借助了Google, 微信等值得信赖的IT服务提供商。OAuth是定义一种协议帮助资源的拥有者在**不提供自身凭证**的前提下授权第三方应用以他的名义存储保护的资源。~~简单来说就是你是某个会所的会员，由于你想参加一些活动（你懂得）但又不想暴露自己，那么会所会直接与活动提供商联络，比如将你作为至尊001，到时你去活动时，只要说自己是至尊001即可，不用透露你的相关信息即可享受服务。~~

获得资源拥有者授权的第三方应用获取受保护的资源采用的不是授权者凭证，而是一个被称为Access Token的安全令牌，Access Token颁发过程会涉及若干不同的角色。接下来通过一个蒋大师提到的简单例子做相应的介绍。例如我们开发了一个集成了新浪微博认证用于发布打折商品信息的App，经过用户授权之后它可以调用新浪微博的WebAPI获取用户的电子邮箱地址并发布相应的打折消息。那么OAuth在该场景下的作用是，用户授权该应用以自己名义调用新浪微博的webAPI获取自己的邮箱地址，涉及4个角色：资源拥有者，一般为最终用户；客户端应用，需要获得资源拥有者授权并最终访问受保护资源的应用；资源服务器，最终承载资源的服务器，一本为一个webAPI;授权服务器，它对用户和客户端实施认证，并在用户授权的情况下向客户端应用颁发Access Token，在之前介绍的场景下，两者合一，均为新浪微博。

一般来说，如果需要针对某个第三方认证服务来开发应用，需要向对应的认证服务提供商对应用进行注册，成功后获得AppID/AppSecet(名称不一定一样)，实际工作中其实每个项目往往也有AppID。常见的，我们可以申请windows服务<https://account.live.com/developers/applications/>， 申请应用后可以获取clientID和clientSecret，并且设置重定向的域名。这儿想提醒大家的一点，就是这个**重定向设置**可以是多个，并且一定要和你每一个请求的重定向设置对应起来，一旦没有设置，windows live会报无法提供服务的错误。



这儿流程类似于Kerberos认证，首先客户端获取授权凭证，之后再购买访问凭证，最后访问资源。Authorization grant（缩写AG）包含4中类型：Implicit，省略了获取AG过程；Authorization Code, 标准模式，这个AG是一个授权码;之后的两种不太有价值，就不介绍了。

* Implicit Authorization Grant简写IMP，例子如下（需要修改host）, P640,比较麻烦的部分。

该模型中，通过获取当前请求的AccessToken，之后调用Windows Live Connect提供的API(<https://apis.live.net/v5.0/me>)。不过在此之前，如果用户未登陆到Windows Live，那么首先会跳转到登陆页面，完成GetProfile调用后将json格式字符串显示在浏览器中。这儿的核心是，我们通过AuthenticateAttribute将AccessToken写入Cookie中，这与Forms认证相似，不过从安全角度讲，利用Cookie携带安全令牌会引起被称为“跨站请求伪造CSRF， Cross-Site Request Forgery”的安全问题，所以用htto报头来作为安全令牌的载体比较合理。在IAuthenticationFilter接口的ChallengeAsync用于在认证过程中向客户端发送“质询”响应，如果AccessToken不存在，就像WindowLive授权页面重定向，参数(response-type, redirect\_uri, client, scope)以查询字符串形式提供。而IActionFilter中的方法ExecuteActionFilterAsync用于将AccessToken写入cookie。代码如下所示：

|  |
| --- |
| public class **DemoController** : ApiController  {  public HttpResponseMessage GetProfile()  {  string accessToken;  if (this.Request.TryGetAccessToken(out accessToken))  {  using (var client = new HttpClient())  {  string address = string.Format("https://apis.live.net/v5.0/me?access\_token={0}", accessToken);  return client.GetAsync(address).Result;  }  }  return new HttpResponseMessage(HttpStatusCode.BadRequest) { ReasonPhrase = "No access token" };  }  public class **AccountController** : Controller  {  [RequireHttps]  public ActionResult CaptureToken(string requestUri)  {  ViewBag.RequestUri = requestUri;  return View();  }  [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]  public class **AuthenticateAttribute** : FilterAttribute, IAuthenticationFilter, IActionFilter  {  public const string CookieName = "AccessToken";  public string CaptureTokenUri { get; private set; }  public AuthenticateAttribute(string captureTokenUri)  {  this.CaptureTokenUri = captureTokenUri;  }  public Task AuthenticateAsync(HttpAuthenticationContext context, System.Threading.CancellationToken cancellationToken)  {  return Task.FromResult<object>(null);  }  public Task ChallengeAsync(HttpAuthenticationChallengeContext context, System.Threading.CancellationToken cancellationToken)  {  string accessToken;  if (!context.Request.TryGetAccessToken(out accessToken))  {  string clientId = "00000000441745E6";  string redirectUri = string.Format("{0}?requestUri={1}", this.CaptureTokenUri, context.Request.RequestUri);  string scope = "wl.signin%20wl.basic";  var uri = new StringBuilder();  uri.Append("https://login.live.com/oauth20\_authorize.srf");  uri.Append("?response\_type=token");  uri.AppendFormat("&redirect\_uri={0}&client\_id={1}&scope={2}", redirectUri, clientId, scope);  context.Result = new RedirectResult(new Uri(uri.ToString()), context.Request);  }  return Task.FromResult<object>(null);  }  public Task<System.Net.Http.HttpResponseMessage> ExecuteActionFilterAsync(System.Web.Http.Controllers.HttpActionContext actionContext, System.Threading.CancellationToken cancellationToken, Func<Task<System.Net.Http.HttpResponseMessage>> continuation)  {  var response = continuation().Result;  string accessToken;  if (actionContext.Request.TryGetAccessToken(out accessToken))  {  response.SetAccessToken(actionContext.Request, accessToken);  }  return Task.FromResult<HttpResponseMessage>(response);  }  **CaptureToken.cshtml**  <html>  <head>  <title></title>  <script src="@Url.Content("~/scripts/jquery-1.10.2.js")"></script>  <script type="text/javascript">  $(document).ready(function () {  var redirectUri = '@ViewBag.RequestUri';  if (redirectUri.indexOf('?') >= 0) {  redirectUri += '&' + location.**hash**.slice(1)  }  else {  redirectUri += '?' + location.hash.slice(1)  }  location.href = redirectUri;  });  </script>  </head>  <body></body>  </html> |

Tip:

location是javascript里边管理地址栏的内置对象，比如location.href就管理页面的url，用location.href=url就可以直接将页面重定向url。而location.hash则可以用来获取或设置页面的标签值。比如http://domain/**#**admin的location.hash="#admin"。利用这个属性值可以做一个非常有意义的事情。简单来说就是”#”号后面跟着的内容，类似于查询字符串。

window.location.hash这个属性可读可写。读取时，可以用来判断网页状态是否改变；写入时，则会在不重载网页的前提下，创造一条访问历史记录。

* Authorization Code Authorization Grant（简写AC）

之前介绍的IMP存在两个问题，其一，授权服务器没有对客户端应用进行认证，因为获取Access Token的请求只提供了客户端应用的ClientID而没有ClientSecret；其二，Access Token是授权服务器单独颁发给客户端应用的，应该对于其他人是不可见（包括拥有被访问资源的授权者）。IMP类型授权的客户端运行于纯客户端上下文环境，AC类型的使用户运行于服务器的应用，比如MVC应用中的Controller。

步骤1：客户端向授权服务器发送一个获取Authentication Code（认购权证）的请求，请求的地址和参数和IMP相似。

|  |  |
| --- | --- |
| 参数名 | 解释 |
| Response\_type | 表示请求希望获取的对象类型，在此我们希望获取的是Authorization Code |
| Redirect\_uri | 表示授权服务器在获得用户授权并完成对用户认证后重定向的地址，AC就是以查询字符串方式附加 |
| Client\_id | 授权客户端应用的clint\_id |
| Scope | 表示授权的范围，根据具体需要的权限集而定 |

步骤2：客户端利用AC向授权服务器获取Access Token，一般为POST请求，参数包括：

|  |  |
| --- | --- |
| 参数名 | 解释 |
| Client | 授权客户端应用的clint\_id |
| Client\_secret | 该标识对应的ClientSecret |
| Redirect\_uri | 之前获取AC时指定的重定向地址 |
| Grant\_type | 采用Authorization Grant类型，值为”authorization\_code” |

授权服务器接受到请求后，除了利用clientID和Secret对客户端实施验证外，还会检查重定向地址是否一致，完成后，生成一个AccessToken发还。消息包括：token\_type,bearer;expires\_in,3600;scope,wl.signin wl.basic;access\_token;authentication\_token。

出于安全考虑，access token有一个过期时限，此外授权服务器还会返回一个长期有效的安全令牌，当ac token过期时，可以利用它再获取，使用它需要在scope中加入”wl.offline\_access”，相关代码如下所示。

对了，实际使用中，不需要这么麻烦，你可以看到在project的app\_start的StartupAuth中，可以看到微软的设置，只用输入对应的clientId和clientSecret。

Tip:

一个问题，为什么我设置的[www.sory.com](http://www.sory.com)可以访问到，而我并未申请该域名和绑定IP？

WebAPI采用REST风格，将浏览器作为执行上下文客户端js应用是主要消费者，但“同源策略”限制了js的跨站点调用，这将导致WebAPI不能跨域访问资源，那么它将“名不副实”，如何解决这个问题呢？将在后面意义道来，主要有jsonP和cros规范两种方式。

浏览器作为internet工具，它为客户端应用提供一个寄宿和运行的环境。这个应用就是javaScript程序，由于js脚本并非都值得信奈，所以对js运行限制在一个沙盒sandbox中。同源策略是一项最基本的安全策略，是浏览器安全的基础，它限制了来自A站点的脚本只能操作A的页面的DOM,跨域操作B站点的资源将会被拒绝。同源要求一下3方面相同：主机名称(域名/子域名或者IP地址)；端口号；网络协议(Schema)。需要注意的，对于一段js脚本来说，其“源”与它存储的地址无关，而取决于脚本被加载的页面，JSONP就利用了这个特性。除了<script>标签，html还有其他一些具有src属性的标签（<img><iframe><link>）均具有跨域加载资源的能力，对于这些标签，每次加载都涉及一个GET请求。同源策略主要针对Ajax请求，该策略主要限制了通过XMLHttpRequest发送的Ajax请求，如果是一个异源地址，浏览器将拒绝读取返回的内容。

一个跨域访问的小例子，一个MVC的应用去调用一个webAPI应用的服务，两者在不同的接口下时。

|  |
| --- |
| <head>  <title>联系人列表</title>  <script src="@Url.Content("~/scripts/jquery-1.10.2.js")"></script>  </head>  <body>  <ul> id="contacts"</ul>  <script type="text/javascript">  $(function () {  var url = "http://www.sory.com/Sory.Entertainment.WebAPI/api/contact";  $.getJSON(url, null, function (contacts) {  $.each(contacts, function (index, contact) {  var html = "<li><ul>";  html += "<li>Name: " + contact.Name + "</li>";  html += "<li>Phone No: " + contact.PhoneNo + "</li>";  html += "<li>Email Address: " + contact.EmailAddress + "</li>";  html += "</ul>";  $("#contacts").append($(html));  });  });  });  </script>  </body> |

跨域调用的错误信息：XMLHttpRequest cannot load http://www.sory.com/Sory.Entertainment.WebAPI/api/contact. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:26829' is therefore not allowed access.

* JSONP方式

之前说过，js脚本的源有加载页面决定，而不是存储地址。对于一段<script>标签src属性加在的js脚本，它与当前页面同源。对于之前的例子来说，可以将联系人列表的呈现单独定义在listContacts函数中，并将WebAPI的地址置于<script>标签的src属性中来间接调用。

命名空间：System.Net.Http,简化了我们原有的HttpWebRequest的调用模式。

基本使用：

HttpResponseMessage response = client.GetAsync(“http://...”).Result;

Cw(response.Content.ReadAsAsync<DateTime>().Result.ToString();(这儿的result与await很类似啊, Task<long>)

* 利用httpClient调用Basic认证下的WebAPI

这儿需要注意，首先在nuget中获取httpClient相关组件，其依赖于Microsoft.Bcl库，部分扩展方法需要添加相应命名空间,自己找了半天ReadAsync<T>这个泛型方法半天没找到，也可以自己写一个，比较简单方便。

|  |
| --- |
| using (var client = new HttpClient())  {  var identityString = string.Format("{0}:{1}", "cn1\\she\_s", "Wanliwang11");  byte[] credential = Encoding.Default.GetBytes(identityString);  client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Basic", Convert.ToBase64String(credential));  HttpResponseMessage response = client.GetAsync("http://localhost/Sory.Entertainment.WebAPI/json/Demo/GetUsers").Result;  var userNames = response.Content.ReadAsStringAsync().Result;  return userNames;  } |

* 利用httpClient调用forms认证下的WebAPI

在Froms认证时，我们首先需要请求login页面，将用户名密码作为token发送给服务器，之后获取服务器响应信息head中的”Set-Cookie”属性，接着获取其中key为”.ASPXAUTH”的cookie信息，这个也就是服务器端和客户端通信的token。之后再在自己真正需要的请求中，附加上该token(用FormUrlEncodeContent打包，这个类实际气的就是原来mediaType=”application/x-www-form-urlencoded”的作用)，还有要注意的是httpClientHandler中需要初始化CookieContainer。

|  |
| --- |
| public static string GetData()  {  var result = string.Empty;  string token = GetSecurityToken("xionger", "123456", \_loginUrl, ".ASPXAUTH");  if (!string.IsNullOrEmpty(token))  {  var handler = new HttpClientHandler() { **CookieContainer** = new CookieContainer() };  handler.CookieContainer.Add(new Uri(\_targetUrl), new Cookie(".ASPXAUTH", token));  using (var client = new HttpClient(handler))  {  var response = client.GetAsync(\_targetUrl).Result;  result = response.Content.ReadAsStringAsync().Result;  }  }  return result;  }  private static string GetSecurityToken(string userName, string password, string url, string cookieName)  {  using (var client = new HttpClient())  {  Dictionary<string, string> credential = new Dictionary<string, string>();  credential.Add("UserName", userName);  credential.Add("Password", password);  var response = client.PostAsync(url, new **FormUrlEncodedContent**(credential)).Result;  IEnumerable<string> cookies;  if (response.**Headers**.TryGetValues(**"Set-Cookie"**, out cookies))  {  string token = cookies.FirstOrDefault(value => value.StartsWith(cookieName));  if (null == token)  {  return null;  }  return token.Split(';')[0].Substring(cookieName.Length + 1);  }  return null;  }  } |

参考资料：

1. 蒋金楠. ASP.NET Web API 2框架揭秘[M]. 北京:电子工业出版社, 2014.
2. （美）加洛韦. ASP.NET MVC 5高级编程(第5版)[M]. 北京:清华大学出版社, 2015.