[ASP.NET Core 认证与授权[1]:初识认证](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html)

在ASP.NET 4.X 中，我们最常用的是Forms认证，它既可以用于局域网环境，也可用于互联网环境，有着非常广泛的使用。但是它很难进行扩展，更无法与第三方认证集成，因此，在 ASP.NET Core 中对认证与授权进行了全新的设计，并使用基于声明的认证(**claims-based authentication**)，以适应现代化应用的需求。在[运行原理解剖[5]:Authentication](http://www.cnblogs.com/RainingNight/p/authentication-in-asp-net-core.html)中介绍了一下**HttpContext**与认证系统的集成，本系列文章则来详细介绍一下 ASP.NET Core 中认证与授权。

**目录**

1. [基于声明的认证](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#%E5%9F%BA%E4%BA%8E%E5%A3%B0%E6%98%8E%E7%9A%84%E8%AE%A4%E8%AF%81)
2. [ASP.NET Core 中的用户身份](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#asp.net-core-%E4%B8%AD%E7%9A%84%E7%94%A8%E6%88%B7%E8%BA%AB%E4%BB%BD)
   * [Claim](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#claim)
   * [ClaimsIdentity](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#claimsidentity)
   * [ClaimsPrincipal](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#claimsprincipal)
   * [AuthenticationTicket](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#authenticationticket)
3. [Microsoft.AspNetCore.Authentication](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#microsoft.aspnetcore.authentication)
   * [Usage](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#usage)
   * [AddAuthentication](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#addauthentication)
   * [AddScheme](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#addscheme)
   * [AddRemoteScheme](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#addremotescheme)
   * [UseAuthentication](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#useauthentication)
4. [认证Handler](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#%E8%AE%A4%E8%AF%81handler)
   * [AuthenticationHandler](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#authenticationhandler)
   * [RemoteAuthenticationHandler](https://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#remoteauthenticationhandler)

**基于声明的认证**

Claim 通常被翻译成声明，但是感觉过于生硬，还是使用**Claim**来称呼更加自然一些。记得是在MVC5中，第一次接触到 “Claim" 的概念。在MVC5之前，我们所熟悉的是Windows认证和Forms认证，Windows认证通常用于企业内部，我们使用最多的还是Forms认证，先来回顾一下，以前是怎么使用的：

首先我们会在web.config中配置认证模式：

<authentication mode="Forms">

<forms loginUrl="~/Account/LogOn" timeout="2880" />

</authentication>

认证票据的生成是使用FormsAuthentication来完成的：

FormsAuthentication.SetAuthCookie("bob", true);

然后便可以通过HttpContext.User.Identity.Name获取到当前登录用户的名称："bob"，那么它是如何来完成认证的呢？

在 ASP.NET 4.x 中，我们应该都对 HttpModule 比较了解，它类似于 ASP.NET Core 中的中件间，ASP.NET 默认会在全局的 *administration.config* 文件中注册一大堆HttpModule，其中就包括WindowsAuthentication和FormsAuthentication，用来实现Windows认证和Forms认证：

<moduleProviders>

<!-- Server Modules-->

<add name="Authentication" type="Microsoft.Web.Management.Iis.Authentication.AuthenticationModuleProvider, Microsoft.Web.Management.Iis, Version=10.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />

<add name="AnonymousAuthentication" type="Microsoft.Web.Management.Iis.Authentication.AnonymousAuthenticationModuleProvider, Microsoft.Web.Management.Iis, Version=10.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />

<add name="BasicAuthentication" type="Microsoft.Web.Management.Iis.Authentication.BasicAuthenticationModuleProvider, Microsoft.Web.Management.Iis, Version=10.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />

<add name="ActiveDirectoryAuthentication" type="Microsoft.Web.Management.Iis.Authentication.ActiveDirectoryAuthenticationModuleProvider, Microsoft.Web.Management.Iis, Version=10.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />

<add name="WindowsAuthentication" type="Microsoft.Web.Management.Iis.Authentication.WindowsAuthenticationModuleProvider, Microsoft.Web.Management.Iis, Version=10.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />

<add name="DigestAuthentication" type="Microsoft.Web.Management.Iis.Authentication.DigestAuthenticationModuleProvider, Microsoft.Web.Management.Iis, Version=10.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />

<!-- ASP.NET Modules-->

<add name="FormsAuthentication" type="Microsoft.Web.Management.AspNet.Authentication.FormsAuthenticationModuleProvider, Microsoft.Web.Management.Aspnet, Version=10.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />

可能大多人都不知道有这些Module，这也是微软技术的一大弊端，总想着封装成傻瓜化，造成入门容易，精通太难的局面。

如上，我们可以看到生成票据时，默认只能转入一个Name，当然也可以通过手动创建FormsAuthenticationTicket来附带一些额外的信息，但是都太过麻烦。

在传统的身份认证中，每个应用程序都有它自己的验证用户身份的方式，以及它自己的用户数据库。这种方式有很大的局限性，因为它很难集成多种认证方式以支持用户使用不同的方式来访问我们的应用程序，比如组织内的用户（Windows-baseed 认证），其它组织的用户（Identity federation）或者是来自互联网的用户（Forms-based 认证）等等。

而*Claim* 是关于一个人或组织的某个主题的陈述，比如：一个人的名称，角色，个人喜好，种族，特权，社团，能力等等。它本质上就是一个键值对，是一种非常通用的保存用户信息的方式，可以很容易的将认证和授权分离开来，前者用来表示用户是/不是什么，后者用来表示用户能/不能做什么。

因此基于声明的认证有两个主要的特点：

* 将认证与授权拆分成两个独立的服务。
* 在需要授权的服务中，不用再去关心你是如何认证的，你用Windows认证也好，Forms认证也行，只要你出示你的 *Claims* 就行了。

**ASP.NET Core 中的用户身份**

**Claim**

在 ASP.NET Core 中，使用Cliam类来表示用户身份中的一项信息，它由核心的Type和Value属性构成：

public class Claim

{

private readonly string \_type;

private readonly string \_value;

public Claim(string type, string value)

: this(type, value, ClaimValueTypes.String, ClaimsIdentity.DefaultIssuer, ClaimsIdentity.DefaultIssuer, null, null, null)

{

}

internal Claim(string type, string value, string valueType, string issuer, string originalIssuer, ClaimsIdentity subject, string propertyKey, string propertyValue)

{

...

}

public string Type => \_type;

public string Value => \_value;

}

一个Claim可以是“用户的姓名”，“邮箱地址”，“电话”，等等，而多个Claim构成一个用户的身份，使用ClaimsIdentity类来表示：

**ClaimsIdentity**

public class ClaimsIdentity : IIdentity

{

public virtual IEnumerable<Claim> Claims {get;}

public virtual string AuthenticationType => \_authenticationType;

public virtual bool IsAuthenticated => !string.IsNullOrEmpty(\_authenticationType);

public virtual string Name

{

get

{

Claim claim = FindFirst(\_nameClaimType);

if (claim != null) return claim.Value;

return null;

}

}

}

如上，其Name属性用来查找Claims中，第一个Type为我们创建ClaimsIdentity时指定的NameClaimType的Claim的值，若未指定Type时则使用默认的ClaimTypes.Name。而IsAuthenticated只是判断\_authenticationType是否为空，\_authenticationType则对应[上一章](http://www.cnblogs.com/RainingNight/p/authentication-in-asp-net-core.html)中介绍的**Scheme**。

下面，我们演示一下用户身份的创建：

// 创建一个用户身份，注意需要指定AuthenticationType，否则IsAuthenticated将为false。

var claimIdentity = new ClaimsIdentity("myAuthenticationType");

// 添加几个Claim

claimIdentity.AddClaim(new Claim(ClaimTypes.Name, "bob"));

claimIdentity.AddClaim(new Claim(ClaimTypes.Email, "bob@gmail.com"));

claimIdentity.AddClaim(new Claim(ClaimTypes.MobilePhone, "18888888888"));

如上，我们可以根据需要添加任意个的Claim，最后我们还需要再将用户身份放到ClaimsPrincipal对象中。

**ClaimsPrincipal**

那么，ClaimsPrincipal是什么呢？在 ASP.NET 4.x 中我们可能对IPrincipal接口比较熟悉，在Controller中的User属性便是IPrincipal类型：

public interface IPrincipal

{

IIdentity Identity { get; }

bool IsInRole(string role);

}

可以看到IPrincipal除了包含用户身份外，还有一个IsInRole方法，用于判断用户是否属于指定角色，在基于角色的授权当中便是调用此方法来实现的。

而在 ASP.NET Core 中，HttpContext直接使用的就是ClaimsPrincipal类型，而不再使用IPrincipal。

public abstract class HttpContext

{

public abstract ClaimsPrincipal User { get; set; }

}

而在ClaimsPrincipal中，可以包含多个用户身份(ClaimsIdentity)，除了对用户身份的操作，还提供了针对Claims的查询：

public class ClaimsPrincipal : IPrincipal

{

private readonly List<ClaimsIdentity> \_identities = new List<ClaimsIdentity>();

public ClaimsPrincipal(IEnumerable<ClaimsIdentity> identities)

{

\_identities.AddRange(identities);

}

// 默认从\_identities中查找第一个不为空的ClaimsIdentity，也可以自定义查找方式。

public virtual System.Security.Principal.IIdentity Identity {}

// 查找\_identities中是否包含类型为RoleClaimType(在创建ClaimsIdentity时指定，或者默认的ClaimTypes.Role)的Claim。

public virtual bool IsInRole(string role) {}

// 获取所有身份的Claim集合

public virtual IEnumerable<Claim> Claims

{

get

{

foreach (ClaimsIdentity identity in Identities)

{

foreach (Claim claim in identity.Claims)

{

yield return claim;

}

}

}

}

}

ClaimsPrincipal的创建非常简单，只需传入我们上面创建的用户身份即可：

var principal = new ClaimsPrincipal(claimIdentity);

由于HTTP是无状态的，我们通常使用Cookie，请求头或请求参数等方式来附加用户的信息，在网络上进行传输，这就涉及到序列化和安全方面的问题。因此，还需要将principal对象包装成AuthenticationTicket对象。

**AuthenticationTicket**

当我们创建完ClaimsPrincipal对象后，需要将它生成一个用户票据并颁发给用户，然后用户拿着这个票据，便可以访问受保持的资源，而在 ASP.NET Core 中，用户票据用AuthenticationTicket来表示，如在Cookie认证中，其认证后的Cookie值便是对该对象序列化后的结果，它的定义如下：

public class AuthenticationTicket

{

public AuthenticationTicket(ClaimsPrincipal principal, AuthenticationProperties properties, string authenticationScheme)

{

AuthenticationScheme = authenticationScheme;

Principal = principal;

Properties = properties ?? new AuthenticationProperties();

}

public AuthenticationTicket(ClaimsPrincipal principal, string authenticationScheme)

: this(principal, properties: null, authenticationScheme: authenticationScheme) { }

public string AuthenticationScheme { get; private set; }

public ClaimsPrincipal Principal { get; private set; }

public AuthenticationProperties Properties { get; private set; }

}

用户票据除了包含上面创建的principal对象外，还需要指定一个AuthenticationScheme (通常在授权中用来验证Scheme)，并且还包含一个AuthenticationProperties对象，它主要是一些用户票据安全方面的一些配置，如过期时间，是否持久等。

var properties = new AuthenticationProperties();

var ticket = new AuthenticationTicket(principal, properties, "myScheme");

// 加密 序列化

var token = Protect(ticket);

最后，我们可以将票据(token)写入到Cookie中，或是也可以以JSON的形式返回让客户端自行保存，由于我们对票据进行了加密，可以保证在网络中安全的传输而不会被篡改。

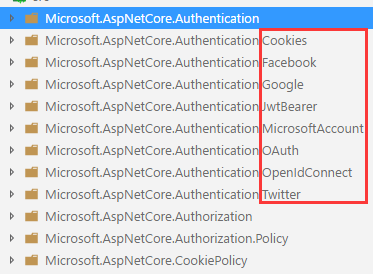
最终身份令牌的结构大概是这样的：



**Microsoft.AspNetCore.Authentication**

上面，我们介绍了身份票据的创建过程，下面就来介绍一下 ASP.NET Core 中的身份认证。

ASP.NET Core 中的认证系统具体实现在 [Security](https://github.com/aspnet/Security) 项目中，它包含 Cookie, JwtBearer, OAuth, OpenIdConnect 等：



认证系统提供了非常灵活的扩展，可以让我们很容易的实现自定义认证方式。

**Usage**

而对于认证系统的配置，分为两步，也是我们所熟悉的注册服务和配置中间件：

首先，在DI中注册服务认证所需的服务：

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;

})

.AddCookie()

.AddOpenIdConnect(o =>

{

o.ClientId = "server.hybrid";

o.ClientSecret = "secret";

o.Authority = "https://demo.identityserver.io/";

o.ResponseType = OpenIdConnectResponseType.CodeIdToken;

});

}

最后，注册认证中间件：

public void Configure(IApplicationBuilder app)

{

app.UseAuthentication();

}

如上，我们的系统便支持了Cookie和JwtBearer两种认证方式，是不是非常简单，在我们的应用程序中使用认证系统时，只需要调用 [上一章](http://www.cnblogs.com/RainingNight/p/authentication-in-asp-net-core.html)介绍的 **HttpContext** 中认证相关的扩展方法即可。

[Microsoft.AspNetCore.Authentication](https://github.com/aspnet/Security/tree/dev/src/Microsoft.AspNetCore.Authentication)，是所有认证实现的公共抽象类，它定义了实现认证Handler的规范，并包含一些共用的方法，如令牌加密，序列化等，AddAuthentication 便是其提供的统一的注册认证服务的扩展方法：

**AddAuthentication**

public static AuthenticationBuilder AddAuthentication(this IServiceCollection services)

{

services.AddAuthenticationCore();

services.AddDataProtection();

services.AddWebEncoders();

services.TryAddSingleton<ISystemClock, SystemClock>();

return new AuthenticationBuilder(services);

}

public static AuthenticationBuilder AddAuthentication(this IServiceCollection services, Action<AuthenticationOptions> configureOptions)

{

var builder = services.AddAuthentication();

services.Configure(configureOptions);

return builder;

}

如上，它首先会调用[上一章](http://www.cnblogs.com/RainingNight/p/authentication-in-asp-net-core.html)中介绍的AddAuthenticationCore方法，然后注册了DataProtection和WebEncoders两个服务。而对 AuthenticationOptions 我们之前在[IAuthenticationSchemeProvider](http://www.cnblogs.com/RainingNight/p/authentication-in-asp-net-core.html#iauthenticationschemeprovider)也介绍过，它用来配置Scheme。

**AddScheme**

在上面的 *AddAuthentication* 中返回的是一个AuthenticationBuilder类型，所有认证Handler的注册都是以它的扩展形式来实现的，它同时也提供了AddScheme扩展方法，使我们可以更加方便的来配置Scheme：

public class AuthenticationBuilder

{

public AuthenticationBuilder(IServiceCollection services)

=> Services = services;

public virtual IServiceCollection Services { get; }

public virtual AuthenticationBuilder AddScheme<TOptions, THandler>(string authenticationScheme, Action<TOptions> configureOptions)

where TOptions : AuthenticationSchemeOptions, new()

where THandler : AuthenticationHandler<TOptions>

=> AddScheme<TOptions, THandler>(authenticationScheme, displayName: null, configureOptions: configureOptions);

public virtual AuthenticationBuilder AddScheme<TOptions, THandler>(string authenticationScheme, string displayName, Action<TOptions> configureOptions)

where TOptions : AuthenticationSchemeOptions, new()

where THandler : AuthenticationHandler<TOptions>

{

Services.Configure<AuthenticationOptions>(o =>

{

o.AddScheme(authenticationScheme, scheme => {

scheme.HandlerType = typeof(THandler);

scheme.DisplayName = displayName;

});

});

if (configureOptions != null)

{

Services.Configure(authenticationScheme, configureOptions);

}

Services.AddTransient<THandler>();

return this;

}

}

在这里的*AddScheme* 扩展方法只是封装了对AuthenticationOptions中AddScheme的调用，如上面示例中的AddCookie便是调用该扩展方法来实现的。

**AddRemoteScheme**

看到 *Remote* 我们应该就可以猜到它是一种远程验证方式，先看一下它的定义：

public class AuthenticationBuilder

{

public virtual AuthenticationBuilder AddRemoteScheme<TOptions, THandler>(string authenticationScheme, string displayName, Action<TOptions> configureOptions)

where TOptions : RemoteAuthenticationOptions, new()

where THandler : RemoteAuthenticationHandler<TOptions>

{

Services.TryAddEnumerable(ServiceDescriptor.Singleton<IPostConfigureOptions<TOptions>, EnsureSignInScheme<TOptions>>());

return AddScheme<TOptions, THandler>(authenticationScheme, displayName, configureOptions: configureOptions);

}

private class EnsureSignInScheme<TOptions> : IPostConfigureOptions<TOptions> where TOptions : RemoteAuthenticationOptions

{

private readonly AuthenticationOptions \_authOptions;

public EnsureSignInScheme(IOptions<AuthenticationOptions> authOptions)

{

\_authOptions = authOptions.Value;

}

public void PostConfigure(string name, TOptions options)

{

options.SignInScheme = options.SignInScheme ?? \_authOptions.DefaultSignInScheme ?? \_authOptions.DefaultScheme;

if (string.Equals(options.SignInScheme, name, StringComparison.Ordinal))

{

throw new InvalidOperationException(Resources.Exception\_RemoteSignInSchemeCannotBeSelf);

}

}

}

}

首先使用**PostConfigure**模式（参见：[Options[1]:Configure](http://www.cnblogs.com/RainingNight/p/strongly-typed-options-configure-in-asp-net-core.html#postconfigure%E6%96%B9%E6%B3%95)），对RemoteAuthenticationOptions进行验证，要求远程验证中指定的SignInScheme不能为自身，这是为什么呢？后文再来解释。然后便是直接调用上面介绍的 *AddScheme* 方法。

关于远程验证相对比较复杂，在本章中并不会太过深入的来介绍，在后续其它文章中会逐渐深入。

**UseAuthentication**

在上面，注册认证中间件时，我们只需调用一个UseAuthentication扩展方法，因为它会执行我们注册的所有认证Handler：

public static IApplicationBuilder UseAuthentication(this IApplicationBuilder app)

{

return app.UseMiddleware<AuthenticationMiddleware>();

}

咦，它的代码好简单，只是注册了一个 **AuthenticationMiddleware** 而已，迫不及待的想看看它的实现：

public class AuthenticationMiddleware

{

private readonly RequestDelegate \_next;

public IAuthenticationSchemeProvider Schemes { get; set; }

public async Task Invoke(HttpContext context)

{

context.Features.Set<IAuthenticationFeature>(new AuthenticationFeature

{

OriginalPath = context.Request.Path,

OriginalPathBase = context.Request.PathBase

});

var handlers = context.RequestServices.GetRequiredService<IAuthenticationHandlerProvider>();

foreach (var scheme in await Schemes.GetRequestHandlerSchemesAsync())

{

var handler = await handlers.GetHandlerAsync(context, scheme.Name) as IAuthenticationRequestHandler;

if (handler != null && await handler.HandleRequestAsync())

{

return;

}

}

var defaultAuthenticate = await Schemes.GetDefaultAuthenticateSchemeAsync();

if (defaultAuthenticate != null)

{

var result = await context.AuthenticateAsync(defaultAuthenticate.Name);

if (result?.Principal != null)

{

context.User = result.Principal;

}

}

await \_next(context);

}

}

很简单，但是很强大，不管我们是使用Cookie认证，还是Bearer认证，等等，都只需要这一个中间件，因为它会解析所有的Handler来执行。

不过，在这里，这会先判断是否具体实现了IAuthenticationRequestHandler的Hander，优先来执行，这个是什么鬼?

查了一下，发现IAuthenticationRequestHandler是在[HttpAbstractions](https://github.com/aspnet/HttpAbstractions)中定义的，只是在[运行原理解剖[5]:Authentication](http://www.cnblogs.com/RainingNight/p/authentication-in-asp-net-core.html#iauthenticationschemeprovider)中没有介绍到它：

public interface IAuthenticationRequestHandler : IAuthenticationHandler

{

Task<bool> HandleRequestAsync();

}

它多了一个HandleRequestAsync方法，那么它存在的意义是什么呢？其实在Cookie认证中并没有用到它，它通常在远程认证（如：OAuth, OIDC等）中使用，下文再来介绍。

继续分析上面代码，通过调用Schemes.GetDefaultAuthenticateSchemeAsync来获取到认证的Scheme，也就是上文提到的问题，我们必须指定默认的Scheme。

最后，调用AuthenticateAsync方法进行认证，认证成功后，为HttpContext.User赋值，至于如何解析身份令牌生成ClaimsPrincipal对象，则交给相应的Handler来处理。

**认证Handler**

上文中多次提到认证Handler，它由统一的AuthenticationMiddleware来调用，负责具体的认证实现，并分为本地认证与远程认证两种方式。

在本地验证中，身份令牌的发放与认证通常是由同一个服务器来完成，这也是我们比较熟悉的场景，对于Cookie, JwtBearer等认证来说，都属于是本地验证。而当我们使用OAuth, OIDC等验证方式时，身份令牌的发放则是由独立的服务或是第三方（QQ, Weibo 等）认证来提供，此时在我们的应用程序中获取身份令牌时需要请求远程服务器，因此称之为远程验证。

**AuthenticationHandler**

AuthenticationHandler是所有认证Handler的抽象基类，对于本地认证直接实现该类即可，定义如下：

public abstract class AuthenticationHandler<TOptions> : IAuthenticationHandler where TOptions : AuthenticationSchemeOptions, new()

{

...

public async Task InitializeAsync(AuthenticationScheme scheme, HttpContext context)

{

...

await InitializeEventsAsync();

await InitializeHandlerAsync();

}

protected virtual async Task InitializeEventsAsync() { }

protected virtual Task<object> CreateEventsAsync() => Task.FromResult(new object());

protected virtual Task InitializeHandlerAsync() => Task.CompletedTask;

public async Task<AuthenticateResult> AuthenticateAsync()

{

var result = await HandleAuthenticateOnceAsync();

...

}

protected Task<AuthenticateResult> HandleAuthenticateOnceAsync()

{

if (\_authenticateTask == null)

{

\_authenticateTask = HandleAuthenticateAsync();

}

return \_authenticateTask;

}

protected abstract Task<AuthenticateResult> HandleAuthenticateAsync();

protected virtual Task HandleForbiddenAsync(AuthenticationProperties properties)

{

Response.StatusCode = 403;

return Task.CompletedTask;

}

protected virtual Task HandleChallengeAsync(AuthenticationProperties properties)

{

Response.StatusCode = 401;

return Task.CompletedTask;

}

...

}

如上，它定义一个抽象方法HandleAuthenticateAsync，并使用HandleAuthenticateOnceAsync方法来保证其在每次认证只执行一次。而HandleAuthenticateAsync是认证的核心，交给具体的认证Handler负责实现。而对于 ChallengeAsync, ForbidAsync 等方法也提供了默认的实现。

而对于HandleAuthenticateAsync的实现，大致的逻辑就是从请求中获取上面发放的身份令牌，然后解析成AuthenticationTicket，并经过一系列的验证，最终返回ClaimsPrincipal对象。

**RemoteAuthenticationHandler**

RemoteAuthenticationHandler 便是所有远程认证的抽象基类了，它继承自AuthenticationHandler，并实现了IAuthenticationRequestHandler接口：

public abstract class RemoteAuthenticationHandler<TOptions> : AuthenticationHandler<TOptions>, IAuthenticationRequestHandler

where TOptions : RemoteAuthenticationOptions, new()

{

public virtual Task<bool> ShouldHandleRequestAsync() => Task.FromResult(Options.CallbackPath == Request.Path);

public virtual async Task<bool> HandleRequestAsync()

{

if (!await ShouldHandleRequestAsync())

{

return false;

}

var authResult = await HandleRemoteAuthenticateAsync();

...

await Context.SignInAsync(SignInScheme, ticketContext.Principal, ticketContext.Properties);

if (string.IsNullOrEmpty(ticketContext.ReturnUri)) ticketContext.ReturnUri = "/";

Response.Redirect(ticketContext.ReturnUri);

return true;

}

protected abstract Task<HandleRequestResult> HandleRemoteAuthenticateAsync();

protected override async Task<AuthenticateResult> HandleAuthenticateAsync()

{

var result = await Context.AuthenticateAsync(SignInScheme);

...

}

protected override Task HandleForbiddenAsync(AuthenticationProperties properties)

=> Context.ForbidAsync(SignInScheme);

protected virtual void GenerateCorrelationId(AuthenticationProperties properties) {}

protected virtual bool ValidateCorrelationId(AuthenticationProperties properties) {}

}

在上面介绍的AuthenticationMiddleware中，提到它会先执行实现了IAuthenticationRequestHandler 接口的Handler（远程认证），之后（若未完成认证）再执行本地认证Handler。

而RemoteAuthenticationHandler中核心的认证逻辑便是 HandleRequestAsync 方法，它主要包含2个步骤：

1. 首先执行一个抽象方法HandleRemoteAuthenticateAsync，由具体的Handler来实现，该方法返回的HandleRequestResult对象包含验证的结果（跳过，失败，成功等），在成功时会包含一个ticket对象。
2. 若上一步验证成功，则根据返回的ticket，获取到ClaimsPrincipal对象，并调用其它认证Handler的Context.SignInAsync方法。

也就是说，远程Hander会在用户未登录时，指引用户跳转到认证服务器，登录成功后，解析认证服务器传回的凭证，最终依赖于本地Handler来保存身份令牌。当用户再次访问则无需经过远程Handler，直接交给本地Handler来处理。

由此也可以知道，远程认证中本身并不具备**SignIn**的能力，所以必须通过指定其它SignInScheme交给本地认证来完成 *SignIn*。

对于其父类的HandleAuthenticateAsync抽象方法则定义了一个默认实现：“直接转交给本地验证来处理”。当我们需要定义自己的远程认证方式时，通常只需实现 HandleRemoteAuthenticateAsync 即可，而不用再去处理 HandleAuthenticateAsync 。

**总结**

基于声明的认证并不是微软所特有的，它在国外被广泛的使用，如微软的ADFS，Google，Facebook，Twitter等等。在基于声明的认证中，对认证和授权进行了明确的区分，认证用来颁发一个用户的身份标识，其包含这个用户的基本信息，而对于这个身份的颁发则由我们信任的第三方机构来（STS）颁发（当然，你也可以自己来颁发）。而授权，则是通过获取身份标识中的信息，来判断该用户能做什么，不能做什么。

本文对 ASP.NET Core 中认证系统的整个流程做了一个简要的介绍，可能会比较苦涩难懂，不过没关系，大致有个印象就好，下一章则详细介绍一下最常用的本地认证方式：Cookie认证，后续也会详细介绍 *OIDC* 的用法与实现，到时再回头来看本文或许会豁然开朗。

# [ASP.NET Core 认证与授权[2]:Cookie认证](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html)

由于HTTP协议是无状态的，但对于认证来说，必然要通过一种机制来保存用户状态，而最常用，也最简单的就是Cookie了，它由浏览器自动保存并在发送请求时自动附加到请求头中。尽管在现代Web应用中，Cookie已略显笨重，但它依然是最为重要的用户身份保存方式。在 [上一章](http://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html) 中整体的介绍了一下 ASP.NET Core 中的认证流程，而未提及具体的实现方式，较为抽象，那本章就通过一个完整的示例，以及对其原理的解剖，来详细介绍一下Cookie认证，希望能帮助大家对 ASP.NET Core 认证系统有一个更深入的了解。

**目录**

1. [示例](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#%E7%A4%BA%E4%BE%8B)
   * [创建项目](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#%E5%88%9B%E5%BB%BA%E9%A1%B9%E7%9B%AE)
   * [配置Cookie认证](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#%E9%85%8D%E7%BD%AEcookie%E8%AE%A4%E8%AF%81)
   * [准备工作](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#%E5%87%86%E5%A4%87%E5%B7%A5%E4%BD%9C)
   * [认证流程](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#%E8%AE%A4%E8%AF%81%E6%B5%81%E7%A8%8B)
2. [补充](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#%E8%A1%A5%E5%85%85)
   * [JwtClaimTypes](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#jwtclaimtypes)
   * [SessionStore](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#sessionstore)
   * [Reacting to back-end changes](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#reacting-to-back-end-changes)
   * [Persistent and ExpiresUtc](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#persistent-and-expiresutc)
3. [源码解析](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90)
   * [AddCookie](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#addcookie)
   * [CookieAuthenticationOptions](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#cookieauthenticationoptions)
   * [CookieAuthenticationHandler](https://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#cookieauthenticationhandler)

## 示例

我们从零开始，一步一步来创建一个完整的 ASP.NET Core Cookie认证的详细示例。

### 创建项目

我们首先创建一个空的 .NET Core 2.0 Web 项目：



在创建的空项目中，默认具有如下引用：

<ItemGroup>

<PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />

</ItemGroup>

Microsoft.AspNetCore.All 是 ASP.NET Core 的全家桶，包含 Mvc, EFCore, Identity, NodeService, AzureAppServe 等等，并集成到了 .NET Core SDK 当中，个人感觉略坑，有违 .NET Core 轻量模块化的理念，虽然使用看似更加方便了，却也让我们变得更加傻瓜化。在本文中为了更好的演示，就移除掉 Microsoft.AspNetCore.All 的引用，然后手动安装需要的Nuget包，当然，你可以不这么做，并略过此小节。

在本章中，会用到以下几个项目中的Nuget包：

* [HttpAbstractions](https://github.com/aspnet/HttpAbstractions), ASP.NET Core的核心项目，我们的每一个应用程序都需要引用该项目。
* [Hosting](https://github.com/aspnet/Hosting), ASP.NET Core应用程序的宿主程序，必须引用。
* [KestrelHttpServer](https://github.com/aspnet/KestrelHttpServer), 最常用的Web服务器，支持跨平台，在Windows下还可以使用HttpSysServer或IISIntegration。
* [Logging](https://github.com/aspnet/Logging)，用来记录日志，可将日志输出到控制台，调试窗口，Windows事件日志等。
* [Security](https://github.com/aspnet/Security)，ASP.NET Core 认证系统，包括Cookies, JwtBearer, OAuth, OpenIdConnect等。

我们使用 **DotNet CLI** 来安装Nuget包：

dotnet add package Microsoft.AspNetCore.Hosting --version 2.0.0

dotnet add package Microsoft.AspNetCore.Server.Kestrel --version 2.0.0

dotnet add package Microsoft.Extensions.Logging.Console --version 2.0.0

dotnet add package Microsoft.AspNetCore.Authentication.Cookies --version 2.0.0

由于WebHost.CreateDefaultBuilder包含在Microsoft.AspNetCore.All中，需要对Program做如下修改：

public static IWebHost BuildWebHost(string[] args) =>

new WebHostBuilder()

.UseKestrel()

.UseUrls("http://localhost:5000")

.UseContentRoot(Directory.GetCurrentDirectory())

.ConfigureLogging((hostingContext, logging) =>

{

logging.AddConsole();

})

.UseStartup<Startup>()

.Build();

如上，我们使用Kestrel服务器，监听5000端口，并将日志打印到控制台，需要注意的是我们并没有使用UseIISIntegration，因此不支持在IIS下运行，需要使用控制台的方式来运行，修改Properties/launchSettings.json文件:

{

"profiles": {

"Console": {

"commandName": "Project",

"launchBrowser": false,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

}

}

}

如上，将IIS的相关配置删除掉，只保留控制台的启动配置，然后在Starup文件的Configure方法中添加如下代码：

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World!");

});

}

最后按下F5，启动程序，在浏览器中访问访问：<http://localhost:5000/>，输出：

Hello World!

### 配置Cookie认证

在 ASP.NET Core 中，有一个非常重要的依赖注入系统，它贯穿于所有项目中。对于认证系统，同样要先进行注册：

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(options =>

{

options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;

})

.AddCookie(options =>

{

// 在这里可以根据需要添加一些Cookie认证相关的配置，在本次示例中使用默认值就可以了。

});

}

如上，我们只配置了DefaultScheme，这样，DefaultSignInScheme, DefaultSignOutScheme, DefaultChallengeScheme, DefaultForbidScheme 等都会使用该 Scheme 作为默认值。

**AddCookie** 用来注册 CookieAuthenticationHandler，由它来完成身份认证的主要逻辑。

在注册完服务之后，接下来便是注册中间件，在 ASP.NET Core 中都是这个套路：

public void Configure(IApplicationBuilder app)

{

app.UseAuthentication();

}

如上，使用UseAuthentication方法注册了AuthenticationMiddleware中间件，它会负责调用对应的Handler，在[上一章](http://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html)中有详细的介绍。

### 准备工作

既然是身份认证，那首先要有用户，我们在这里模拟一个用户仓储，用来实现用户登录时的用户名和密码的检查。

定义用户类：

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public string Email { get; set; }

public string PhoneNumber { get; set; }

public string Password { get; set; }

public DateTime Birthday { get; set; }

}

定义用户仓储：

public class UserStore

{

private static List<User> \_users = new List<User>() {

new User { Id=1, Name="alice", Password="alice", Email="alice@gmail.com", PhoneNumber="18800000001" },

new User { Id=1, Name="bob", Password="bob", Email="bob@gmail.com", PhoneNumber="18800000002" }

};

public User FindUser(string userName, string password)

{

return \_users.FirstOrDefault(\_ => \_.Name == userName && \_.Password == password);

}

}

将UserStore注册到DI系统中：

public void ConfigureServices(IServiceCollection services)

{

...

services.AddSingleton<UserStore>();

}

由于我们并没有使用MVC，而使用字符串拼接的形式返回HTML较为费劲，在这里定义几个生成HTML的扩展方法：

public static class HttpResponseExtensions

{

public static async Task WriteHtmlAsync(this HttpResponse response, Func<HttpResponse, Task> writeContent)

{

var bootstrap = "<link rel=\"stylesheet\" href=\"https://cdn.bootcss.com/bootstrap/3.3.7/css/bootstrap.min.css\" integrity=\"sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u\" crossorigin=\"anonymous\">";

response.ContentType = "text/html";

await response.WriteAsync($"<!DOCTYPE html><html lang=\"zh-CN\"><head><meta charset=\"UTF-8\">{bootstrap}</head><body><div class=\"container\">");

await writeContent(response);

await response.WriteAsync("</div></body></html>");

}

public static async Task WriteTableHeader(this HttpResponse response, IEnumerable<string> columns, IEnumerable<IEnumerable<string>> data)

{

await response.WriteAsync("<table class=\"table table-condensed\">");

await response.WriteAsync("<tr>");

foreach (var column in columns)

{

await response.WriteAsync($"<th>{HtmlEncode(column)}</th>");

}

await response.WriteAsync("</tr>");

foreach (var row in data)

{

await response.WriteAsync("<tr>");

foreach (var column in row)

{

await response.WriteAsync($"<td>{HtmlEncode(column)}</td>");

}

await response.WriteAsync("</tr>");

}

await response.WriteAsync("</table>");

}

public static string HtmlEncode(string content) =>

string.IsNullOrEmpty(content) ? string.Empty : HtmlEncoder.Default.Encode(content);

}

### 认证流程

接下来，便可以在我们的应用程序中愉快的使用认证系统了。在本文中只是最简单的演示，便不使用MVC了，而是在Configure中通过中间件的形式来实现。

#### 登录

首先，我们定义一个登录的页面以及登录成功后身体令牌的发放：

app.Map("/Account/Login", builder => builder.Run(async context =>

{

if (context.Request.Method == "GET")

{

await context.Response.WriteHtmlAsync(async res =>

{

await res.WriteAsync($"<form method=\"post\">");

await res.WriteAsync($"<input type=\"hidden\" name=\"returnUrl\" value=\"{HttpResponseExtensions.HtmlEncode(context.Request.Query["ReturnUrl"])}\"/>");

await res.WriteAsync($"<div class=\"form-group\"><label>用户名：<input type=\"text\" name=\"userName\" class=\"form-control\"></label></div>");

await res.WriteAsync($"<div class=\"form-group\"><label>密码：<input type=\"password\" name=\"password\" class=\"form-control\"></label></div>");

await res.WriteAsync($"<button type=\"submit\" class=\"btn btn-default\">登录</button>");

await res.WriteAsync($"</form>");

});

}

else

{

var userStore = context.RequestServices.GetService<UserStore>();

var user = userStore.FindUser(context.Request.Form["userName"], context.Request.Form["password"]);

if (user == null)

{

await context.Response.WriteHtmlAsync(async res =>

{

await res.WriteAsync($"<h1>用户名或密码错误。</h1>");

await res.WriteAsync("<a class=\"btn btn-default\" href=\"/Account/Login\">返回</a>");

});

}

else

{

var claimIdentity = new ClaimsIdentity("Cookie");

claimIdentity.AddClaim(new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()));

claimIdentity.AddClaim(new Claim(ClaimTypes.Name, user.Name));

claimIdentity.AddClaim(new Claim(ClaimTypes.Email, user.Email));

claimIdentity.AddClaim(new Claim(ClaimTypes.MobilePhone, user.PhoneNumber));

claimIdentity.AddClaim(new Claim(ClaimTypes.DateOfBirth, user.Birthday.ToString()));

var claimsPrincipal = new ClaimsPrincipal(claimIdentity);

// 在上面注册AddAuthentication时，指定了默认的Scheme，在这里便可以不再指定Scheme。

await context.SignInAsync(claimsPrincipal);

if (string.IsNullOrEmpty(context.Request.Form["ReturnUrl"])) context.Response.Redirect("/");

else context.Response.Redirect(context.Request.Form["ReturnUrl"]);

}

}

}));

如上，我们在Get请求中返回登录页面，在Post请求中验证用户名密码，匹配成功后，创建用户Claim, ClaimsIdentity, ClaimsPrincipal 最终通过SignInAsync方法将用户身份写入到响应Cookie中，完成身份令牌的发放。

#### 授权

我们在登录中间件后面添加一个自定义的授权中间件，用来禁用匿名用户的访问：

app.UseAuthorize();

UseAuthorize的实现很简单，就是判断用户是否已通过认证，并跳过对首页的验证：

public static IApplicationBuilder UseAuthorize(this IApplicationBuilder app)

{

return app.Use(async (context, next) =>

{

if (context.Request.Path == "/")

{

await next();

}

else

{

var user = context.User;

if (user?.Identity?.IsAuthenticated ?? false)

{

await next();

}

else

{

await context.ChallengeAsync();

}

}

});

}

其实上面的实现和我们在MVC5中常用的[Authorize]特性非常相似。

#### 个人信息

再定义一个认证后才能访问的页面，并把当前登录用户的信息展示出来：

app.Map("/profile", builder => builder.Run(async context =>

{

await context.Response.WriteHtmlAsync(async res =>

{

await res.WriteAsync($"<h1>你好，当前登录用户： {HttpResponseExtensions.HtmlEncode(context.User.Identity.Name)}</h1>");

await res.WriteAsync("<a class=\"btn btn-default\" href=\"/Account/Logout\">退出</a>");

await res.WriteAsync($"<h2>AuthenticationType：{context.User.Identity.AuthenticationType}</h2>");

await res.WriteAsync("<h2>Claims:</h2>");

await res.WriteTableHeader(new string[] { "Claim Type", "Value" },

context.User.Claims.Select(c => new string[] { c.Type, c.Value }));

});

}));

#### 退出

退出则直接调用SignOutAsync方法即可：

app.Map("/Account/Logout", builder => builder.Run(async context =>

{

await context.SignOutAsync();

context.Response.Redirect("/");

}));

#### 首页

最后，添加一个简单的首页，方便测试：

app.Run(async context =>

{

await context.Response.WriteHtmlAsync(async res =>

{

await res.WriteAsync($"<h2>Hello Cookie Authentication</h2>");

await res.WriteAsync("<a class=\"btn btn-default\" href=\"/profile\">我的信息</a>");

});

});

### 运行

在浏览器中打开<http://localhost:5000/>，显示 "Hello Cookie Authentication"，点击 “我的信息” 按钮：

请求：

GET http://localhost:5000/profile HTTP/1.1

Host: localhost:5000

响应：

HTTP/1.1 302 Found

Location: http://localhost:5000/Account/Login?ReturnUrl=%2Fprofile

因为我们没有登录，会在授权中间件中会执行context.ChallengeAsync();方法，最终会跳转到登录页面，然后输入用户名密码(alice/alice)，登录成功：

请求：

POST http://localhost:5000/Account/Login?ReturnUrl=%2Fprofile HTTP/1.1

Host: localhost:5000

Content-Type: application/x-www-form-urlencoded

returnUrl=%2Fprofile&userName=alice&password=alice

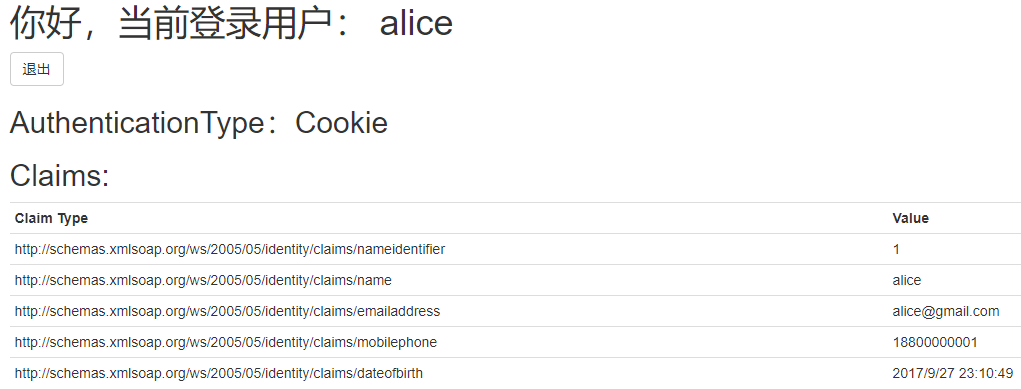
响应：

HTTP/1.1 302 Found

Location: /profile

Set-Cookie: .AspNetCore.Cookies=CfDJ8B4XRZETkRhMt3mT9VduB8KTzFhbcuszdNDXZpaQI3zSOcOD8uAzjr-iHzNCPVgXrKqxfK-MrP5d5r9X1zfKOgg2\_j54t0ccAQ5nshSmXnRvjIZ6id3GD5fDP9v2x1iV0JE7X9IdoA458DZjx6qm6971GeY5HYVnT7odwgQR8eRaHo0-Wacmt95QuC9IVSapqsShHOeu5ZowFmDAPXrlUHOSwBPAjiLkf8mNbu8U4ZcWFlaBXC9-H-2\_ts5wyi-90zw6jGxX3o7tRiQB4qq8IDmIJbZtN4Nl8TKHHcTbyFl5Z\_\_MrgrjJ7s4cGdnIoDJWB9ENw1IGRgF3Rib8KmhkwhlUyO2VMnuVI8vSP2PcwrkUGtudJwHMHrA8cuS021xpmIhkhgW3e82r\_0\_jxAh1nqG4zwTP5i8iLU6FsOLLWatveSWB441Ntqw-L-pYczsBAYFRT0Hh56ofUAxGd7aaGtDx0jvuuxW5gK245Pf0TKG-4G46yDwLrFtjNcN\_GREbpwtHAz-I7XqiDZgS3nbzjik5s05NxB7d6X3aOFc5JHCwFxW-i-xW-ToJLZrp3Jo8W0bAxVwxZIW2PwZlVtyeYSkqByFRaDS4qcBywE2Bmar\_TyJm9UpVWaL2s9KxpU\_DHN6meYne5E5dH4-k1DoABl6FyNPn6xYfMWxzu0\_7ZFhVJjCycScy1jggCv4Hk5nkltj9A3QrFpNb\_HCk21Uek9g-7Zi150EKfDzhGjMto5\_hbWcmQtUsHuLbZlnYTHXZ-7zELZOepAUts2ZGoUnEaI; path=/; samesite=lax; httponly

可以看到，响应报文在Cookie中附加了身份令牌，并会跳转到之前未登录时访问的页面/profile，跳转后显示如下：



如上，因为浏览器会自动附带上刚才写入的Cookie，所以授权通过，并展示出我们在登录时设置的Claim。

最后，点击 “退出” ，响应报文中会将Cookie设置为空（清除Cookie）:

请求：

GET http://localhost:5000/Account/Logout HTTP/1.1

Host: localhost:5000

响应：

HTTP/1.1 302 Found

Location: /

Set-Cookie: .AspNetCore.Cookies=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/; samesite=lax

## 补充

### JwtClaimTypes

在上面的演示中，我们可以看到Cookie中的值非常的长，而我们设置的Claim并不多，这是因为微软内置的ClaimTypes都是一大串的ULR地址。而对于 ASP.NET Core 本身来说，它并不关心你使用的ClaimType是什么，只要你读取与保存时使用的ClaimType保持一致就没有问题。我们可以使用简短的字符串name来代替ClaimTypes.Name，但是推荐的做法是直接使用JwtClaimTypes，因为它够简短而且通用。

首先添加ClaimTypes的Package引用：

dotnet add package IdentityModel --version 2.12.0

然后，将之前的添加Claim的代码修改如下：

var claimIdentity = new ClaimsIdentity("Cookie", JwtClaimTypes.Name, JwtClaimTypes.Role);

claimIdentity.AddClaim(new Claim(JwtClaimTypes.Id, user.Id.ToString()));

claimIdentity.AddClaim(new Claim(JwtClaimTypes.Name, user.Name));

claimIdentity.AddClaim(new Claim(JwtClaimTypes.Email, user.Email));

claimIdentity.AddClaim(new Claim(JwtClaimTypes.PhoneNumber, user.PhoneNumber));

claimIdentity.AddClaim(new Claim(JwtClaimTypes.BirthDate, user.Birthday.ToString()));

需要注意的是在创建ClaimsIdentity时需要手动指定它的NameType和RoleType，否则它将会使用默认的ClaimTypes.Name和ClaimTypes.Role，这样会导致我们从ClaimsPrincipal中获取Identity.Name属性和执行IsInRole检查时失败。

运行，重新登录，看看效果如何：

请求：

POST http://localhost:5000/Account/Login?ReturnUrl=%2Fprofile HTTP/1.1

Host: localhost:5000

Content-Type: application/x-www-form-urlencoded

returnUrl=%2Fprofile&userName=alice&password=alice

响应：

HTTP/1.1 302 Found

Date: Sun, 24 Sep 2017 06:04:28 GMT

Location: /profile

Set-Cookie: .AspNetCore.Cookies=CfDJ8B4XRZETkRhMt3mT9VduB8JV9NE2zJ9YVQmTpAU3E9Op9rQHvJ7WvdcrarbTGWE7c\_e2aLpoZCdDJ7-0fTFZGUwuLVMC0vD\_eeE9ct2Vj7gHCPCVeK3qQPsQ2lNmKvPwPf82-CURFXGgFC1y-N17tXdT7RoZhLHskIHx7qNcxeicS7wiSDhQD3l3mgOgq0bdjWJTk3LnpHk8zS0fDhKp6Vd6vFvCyzzRJu1ax5Y27Bg3dZp4Zsa3I9HAp5wXmyp51de8scS25nyaV0FEd1YUWgC1LsuwOODrSPqMkokv7XQXQc8W212O2dHbuuJ1xYEr1i5\_Gl1syIX3ZuPj1\_wqcnAKu5keY0ZVJz45iGYIRC09hd4n8j1SEA8dDlhbslCtyZ6xMt6MdRFv1D7fhbt\_g4RGDk7ZkjpnT6z9q3dTWNzkS3gSd9AekBNbUNw9ojZmTWoCFhZgxz-6Wwtcp9z7vIo; path=/; samesite=lax; httponly

是不是短了很多？好吧，其实感觉还是有点长，没关系，下面再介绍一种更加彻底的优化方式。

而最终页面上展示的Claims信息如下：



### SessionStore

终极的解决方案就是参考Session的原理，把Claims信息则保存在服务端，并为其设置一个ID，Cookie中则只保存该ID，这样就可以在服务端通过该ID来检索出完整的Claims信息。不过注意，这并不是在使用 ASP.NET Core 中的Session，只是参考其存储方式。

那么怎么做呢？在前面注册Cookie认证时，使用的AddCookie方法中，其CookieAuthenticationOptions参数还可以设置一个ITicketStore类型的SessionStore属性，我们可以通过实现该接口来自定义Cookie的存取方式，在这里，使用本地缓存来实现：

首先添加Microsoft.Extensions.Caching.Memory的Package引用：

dotnet add package Microsoft.Extensions.Caching.Memory --version 2.0.0

然后，定义MemoryCacheTicketStore类：

public class MemoryCacheTicketStore : ITicketStore

{

private const string KeyPrefix = "CSS-";

private IMemoryCache \_cache;

public MemoryCacheTicketStore()

{

\_cache = new MemoryCache(new MemoryCacheOptions());

}

public async Task<string> StoreAsync(AuthenticationTicket ticket)

{

var key = KeyPrefix + Guid.NewGuid().ToString("N");

await RenewAsync(key, ticket);

return key;

}

public Task RenewAsync(string key, AuthenticationTicket ticket)

{

var options = new MemoryCacheEntryOptions();

var expiresUtc = ticket.Properties.ExpiresUtc;

if (expiresUtc.HasValue)

{

options.SetAbsoluteExpiration(expiresUtc.Value);

}

options.SetSlidingExpiration(TimeSpan.FromHours(1));

\_cache.Set(key, ticket, options);

return Task.FromResult(0);

}

public Task<AuthenticationTicket> RetrieveAsync(string key)

{

\_cache.TryGetValue(key, out AuthenticationTicket ticket);

return Task.FromResult(ticket);

}

public Task RemoveAsync(string key)

{

\_cache.Remove(key);

return Task.FromResult(0);

}

}

将MemoryCacheTicketStore配置到CookieAuthenticationOptions中：

.AddCookie(options =>

{

options.SessionStore = new MemoryCacheTicketStore();

});

再次重新登录，响应如下:

HTTP/1.1 302 Found

Location: /profile

Set-Cookie: .AspNetCore.Cookies=CfDJ8B4XRZETkRhMt3mT9VduB8JOI85seEY347RswRzSiL\_BQlJTb4JeFqpJzXNW8xOH1CwUKjwsx4CJWyMV5Wwq61IV0Kz4If0LmmJpEicZi2uxmyE2jcCXw\_IRaPOaP0eJYM-DkpsjlA\_Qu9knFxrpGQaI\_BuRbUbbVhy62V5vjwMzoSewmQiPblS1PbPiqXfjAGmF\_ZaSM40kwNOboAP\_SMoJjX0AtEzmsUqECWFPZLxLoOJJ10Kz16cnSjtxha\_KXY7i8f95jVbnX3cj79-GQ5iXnRePBBR\_2LsXI5eDW\_6E; path=/; samesite=lax; httponly

这样，Cookie中的值就非常简短了（由于其还包含AuthenticationProperties序列化后的值，并没有想象中的短），并且Cookie中的值不会再随着我们设置的Claims的增加而变长，在分布式环境下则可以使用分布式缓存来保存。

### Reacting to back-end changes

对于认证系统，身份令牌都会有一个有效期的概念，而Cookie认证中默认有效期是14天，因此只要浏览器没有清除Cookie，并且Cookie没有过期，便么就一直可以验证通过。但是，如果用户修改了密码，我们希望该Cookie失效，或者是用户更新了Claims的信息时，我们希望重新生成Cookie，否则我们取到的还是旧的Claims信息。那么，该怎么做呢？

对此，网上比较流行的做法是在用户数据库中添加一个安全字段，当用户修改了一些安全性信息时，便更新该字段，并在Claim中加入此字段，一起写入到Cookie中，验证时便可以判断该字段是否与数据库一致，若不一致则验证失败或重新生成：

public static class LastChangedValidator

{

public static async Task ValidateAsync(CookieValidatePrincipalContext context)

{

var userRepository = context.HttpContext.RequestServices.GetRequiredService<IUserRepository>();

var userPrincipal = context.Principal;

string lastChanged = (from c in userPrincipal.Claims where c.Type == "LastUpdated" select c.Value).FirstOrDefault();

if (string.IsNullOrEmpty(lastChanged) || !userRepository.ValidateLastChanged(userPrincipal, lastChanged))

{

// 1. 验证失败 等同于 Principal = principal;

context.RejectPrincipal();

// 2. 验证通过，并会重新生成Cookie。

context.ShouldRenew = true;

}

}

}

如上，1 和 2 两种方式，我们可以根据实际情况选择一种，而不应该同时存在。

在Cookie认证的配置中，提供了一系列的事件，其中便有一个OnValidatePrincipal事件，用来附加服务端的验证逻辑：

.AddCookie(options =>

{

options.Events = new CookieAuthenticationEvents

{

OnValidatePrincipal = LastChangedValidator.ValidateAsync

};

});

如上，便完成了该事件的注册，不过该验证通常会查询数据库，损耗较大，可以通过设置验证周期来提高性能，如：每5分钟执行验证一次（在MVC5中是有该配置的，Core中暂未发现）。

### Persistent and ExpiresUtc

对于Cookie来说，默认的过期时间为Session，即关闭浏览器后就清除。通常在用户登录时会提供一个记住我的选项，用来保证在关闭浏览时不清除Cookie。而在SignInAsync方法中，还接收一个AuthenticationProperties类型的参数，可以用来指定Cookie是否持久化以及过期时间：

await HttpContext.SignInAsync("MyCookieAuthenticationScheme", principal, new AuthenticationProperties

{

// 持久保存

IsPersistent = true

// 指定过期时间

ExpiresUtc = DateTime.UtcNow.AddMinutes(20)

});

看一下CookieAuthenticationHandler中SignInAsync方法关于该配置的实现：

if (!signInContext.Properties.ExpiresUtc.HasValue)

{

signInContext.Properties.ExpiresUtc = issuedUtc.Add(Options.ExpireTimeSpan);

}

if (signInContext.Properties.IsPersistent)

{

var expiresUtc = signInContext.Properties.ExpiresUtc ?? issuedUtc.Add(Options.ExpireTimeSpan);

signInContext.CookieOptions.Expires = expiresUtc.ToUniversalTime();

}

只有在IsPersistent为True时，才会在写入Cookie指定Expires。需要注意的是浏览器中的Cookie过期时间仅仅是用来指定浏览器是否删除Cookie，而在Cookie存储的值中，也会包含该Cookie认证的发布时间和过期时间等，并在HandleAuthenticateAsync方法中对会其进行验证，并不是说只要你有Cookie就能验证通过。

## 源码解析

### AddCookie

AddCookie已多次用过，无需多说，直接看源码：

public static AuthenticationBuilder AddCookie(this AuthenticationBuilder builder)

=> builder.AddCookie(CookieAuthenticationDefaults.AuthenticationScheme, null, null);

public static AuthenticationBuilder AddCookie(this AuthenticationBuilder builder, string authenticationScheme, string displayName, Action<CookieAuthenticationOptions> configureOptions)

{

builder.Services.TryAddEnumerable(ServiceDescriptor.Singleton<IPostConfigureOptions<CookieAuthenticationOptions>, PostConfigureCookieAuthenticationOptions>());

return builder.AddScheme<CookieAuthenticationOptions, CookieAuthenticationHandler>(authenticationScheme, displayName, configureOptions);

}

其实现非常简单，首先注册了Cookie认证的配置项CookieAuthenticationOptions，而authenticationScheme参数用来指定当前认证的唯一的标识，不能重复。通常，使用默认的CookieAuthenticationDefaults.AuthenticationScheme就可以了，但是当我们同时使用多个Cookie认证方式时，需要手动为他们指定不同的Scheme。

最后，直接调用上一章中介绍的[AddScheme](http://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#addscheme)，完成对CookieAuthenticationHandler的注册。

### CookieAuthenticationOptions

CookieAuthenticationOptions是针对Cookie认证的各种配置，如重定向地址，认证阶段事件的注册，Cookie名，过期时间等等，首先看一下它的定义：

public class CookieAuthenticationOptions : AuthenticationSchemeOptions

{

private CookieBuilder \_cookieBuilder = new RequestPathBaseCookieBuilder

{

SameSite = SameSiteMode.Lax,

HttpOnly = true,

SecurePolicy = CookieSecurePolicy.SameAsRequest,

};

public CookieAuthenticationOptions()

{

ExpireTimeSpan = TimeSpan.FromDays(14);

ReturnUrlParameter = CookieAuthenticationDefaults.ReturnUrlParameter;

SlidingExpiration = true;

Events = new CookieAuthenticationEvents();

}

public CookieBuilder Cookie

{

get => \_cookieBuilder;

set => \_cookieBuilder = value ?? throw new ArgumentNullException(nameof(value));

}

public new CookieAuthenticationEvents Events

{

get => (CookieAuthenticationEvents)base.Events;

set => base.Events = value;

}

public ITicketStore SessionStore { get; set; }

// 当用户未登录时，重定向到该路径，默认：/Account/Login

public PathString LoginPath { get; set; }

// 指定登出的路径，默认：/Account/Logout

public PathString LogoutPath { get; set; }

// 当用户无权访问时，重定向到该路径，默认：/Account/AccessDenied

public PathString AccessDeniedPath { get; set; }

// 返回地址参数名，默认：ReturnUrl

public string ReturnUrlParameter { get; set; }

// 指定Cookie的过期时间

public TimeSpan ExpireTimeSpan { get; set; }

// 当Cookie过期时间已达一半时，是否重置为ExpireTimeSpan

public bool SlidingExpiration { get; set; }

// 用来将Cookie写入到浏览器或删除

public ICookieManager CookieManager { get; set; }

public IDataProtectionProvider DataProtectionProvider { get; set; }

public ISecureDataFormat<AuthenticationTicket> TicketDataFormat { get; set; }

}

#### CookieBuilder

在 ASP.NET Core 2.0 中对针对Cookie的配置集中放到CookieBuilder类型当中，相比之前更加清晰：

public class CookieBuilder : object

{

public virtual string Name { get; set; }

public virtual string Path { get; set; }

public virtual string Domain { get; set; }

public virtual bool HttpOnly { get; set; }

public virtual SameSiteMode SameSite { get; set; }

public virtual CookieSecurePolicy SecurePolicy { get; set; }

public virtual TimeSpan? Expiration { get; set; }

public virtual TimeSpan? MaxAge { get; set; }

public CookieOptions Build(HttpContext context);

}

都是一些针对Cookie配置的标准用法，无需多说。

#### CookieAuthenticationEvents

CookieAuthenticationEvents为我们提供了在Cookie认证的各个阶段（如，登录前后，退出前后，重定向等）注册事件的机会，以便我们拦截一些默认行为，来自定义处理逻辑。

public class CookieAuthenticationEvents

{

public virtual Task ValidatePrincipal(CookieValidatePrincipalContext context) => OnValidatePrincipal(context);

public virtual Task SigningIn(CookieSigningInContext context) => OnSigningIn(context);

public virtual Task SignedIn(CookieSignedInContext context) => OnSignedIn(context);

public virtual Task SigningOut(CookieSigningOutContext context) => OnSigningOut(context);

public virtual Task RedirectToLogout(RedirectContext<CookieAuthenticationOptions> context) => OnRedirectToLogout(context);

public virtual Task RedirectToLogin(RedirectContext<CookieAuthenticationOptions> context) => OnRedirectToLogin(context);

public virtual Task RedirectToReturnUrl(RedirectContext<CookieAuthenticationOptions> context) => OnRedirectToReturnUrl(context);

public virtual Task RedirectToAccessDenied(RedirectContext<CookieAuthenticationOptions> context) => OnRedirectToAccessDenied(context);

}

每一个事件都有它的默认实现，这里就不再多说，我们可以根据实际情况进行注册。

### CookieAuthenticationHandler

CookieAuthenticationHandler便是Cookie认证的具体实现：

public class CookieAuthenticationHandler : AuthenticationHandler<CookieAuthenticationOptions>, IAuthenticationSignInHandler, IAuthenticationSignOutHandler

{

...

protected override async Task<AuthenticateResult> HandleAuthenticateAsync()

{

var result = await EnsureCookieTicket();

if (!result.Succeeded)

{

return result;

}

var context = new CookieValidatePrincipalContext(Context, Scheme, Options, result.Ticket);

// 执行前而介绍的服务端验证

await Events.ValidatePrincipal(context);

if (context.ShouldRenew)

{

// 重新生成Cookie

RequestRefresh(result.Ticket);

}

return AuthenticateResult.Success(new AuthenticationTicket(context.Principal, context.Properties, Scheme.Name));

}

public async virtual Task SignInAsync(ClaimsPrincipal user, AuthenticationProperties properties)

{

...

var ticket = new AuthenticationTicket(signInContext.Principal, signInContext.Properties, signInContext.Scheme.Name);

....

var cookieValue = Options.TicketDataFormat.Protect(ticket, GetTlsTokenBinding());

Options.CookieManager.AppendResponseCookie(Context, Options.Cookie.Name, cookieValue, signInContext.CookieOptions);

var signedInContext = new CookieSignedInContext(Context, Scheme, signInContext.Principal, signInContext.Properties, Options);

await Events.SignedIn(signedInContext);

var shouldRedirect = Options.LoginPath.HasValue && OriginalPath == Options.LoginPath;

await ApplyHeaders(shouldRedirect, signedInContext.Properties);

Logger.SignedIn(Scheme.Name);

}

}

其核心方法HandleAuthenticateAsync会检查请求Cookie，查找与CookieBuilder.Name对应的Cookie值，解密反序列化成AuthenticationTicket对象，最后在上一章介绍的[AuthenticationMiddleware](http://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#useauthentication)中间件中将Principal赋予给**HttpContext**。

而CookieAuthenticationHandler还实现了IAuthenticationSignInHandler和IAuthenticationSignOutHandler，这也是ASP.NET Core中内置的唯一支持登录和退出的认证方式。在SignInAsync方法中使用ClaimsPrincipal来创建一个AuthenticationTicket对象，然后将其加密，写入到Cookie中，便完成了登录（身份令牌的发放），而SignOutAsync方法则只是简单的删除Cookie。

篇幅有限，就不再多说，感兴趣的可以去看一下完整代码：[CookieAuthenticationHandler](https://github.com/aspnet/Security/blob/dev/src/Microsoft.AspNetCore.Authentication.Cookies/CookieAuthenticationHandler.cs)。

## 总结

Cookie认证是一种本地认证方式，也是最为简单，最为常用的认证方式。其认证逻辑也很简单，总结一下就是获取请求中指定的Cookie，解密成功后，反序列生成 AuthenticationTicket 对象，并进行一系列的验证，而登录方法与之对应：根据用户信息创建 AuthenticationTicket 对象，并加密后序列化，写入到Cookie中。在下一章中，就来介绍一下最为流行的远程认证方式：**OAuth** 和 **OpenID Connect**。

# [ASP.NET Core 认证与授权[3]:OAuth & OpenID Connect认证](https://www.cnblogs.com/RainingNight/p/oidc-authentication-in-asp-net-core.html)

在[上一章](http://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html)中，我们了解到，Cookie认证是一种本地认证方式，通常认证与授权都在同一个服务中，也可以使用Cookie共享的方式分开部署，但局限性较大，而如今随着微服务的流行，更加偏向于将以前的单体应用拆分为多个服务并独立部署，而此时，就需要一个统一的认证中心，以及一种远程认证方式，本文就来介绍一下如今最为流行的远程认证方式：OAuth 和 OpenID Connect。

**目录**

1. [OAuth 2.0](https://www.cnblogs.com/RainingNight/p/oidc-authentication-in-asp-net-core.html#oauth-2.0)
   * [用例](https://www.cnblogs.com/RainingNight/p/oidc-authentication-in-asp-net-core.html#%E7%94%A8%E4%BE%8B)
   * [源码分析](https://www.cnblogs.com/RainingNight/p/oidc-authentication-in-asp-net-core.html#%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90)
2. [OpenID Connect](https://www.cnblogs.com/RainingNight/p/oidc-authentication-in-asp-net-core.html#openid-connect)
   * [示例](https://www.cnblogs.com/RainingNight/p/oidc-authentication-in-asp-net-core.html#%E7%A4%BA%E4%BE%8B)
   * [运行流程](https://www.cnblogs.com/RainingNight/p/oidc-authentication-in-asp-net-core.html#%E8%BF%90%E8%A1%8C%E6%B5%81%E7%A8%8B)

## OAuth 2.0

在介绍OAuth之前，我们先简单介绍一下OpenID。OpenID 是一个以用户为中心的数字身份识别框架，它具有开放、分散性。OpenID 的创建基于这样一个概念：我们可以通过 URI （又叫URL或网站地址）来认证一个网站的唯一身份，同理，我们也可以通过这种方式来作为用户的身份认证。

OpenID的认证非常简单，当你访问需要认证的A网站时，A网站要求你输入你的OpenID用户名，然后会跳转你的OpenID服务网站，输入用户名密码验证通过后，再跳回A网站，而些时已经显示认证成功。除了一处注册，到处通行外，OpenID还可以使所有支持OpenID的网站共享用户资源，而用户可以控制哪些信息可以被共享，例如姓名、地址、电话号码等。

而OAuth是一个关于授权（authorization）的开放网络标准，在全世界得到广泛应用，在官网对其是这样定义的：

An open protocol to allow secure API authorization in a simple and standard method from desktop and web applications.

OAuth关注的是第三方应用访问其受保护资源的能力，而OpenID关注的是第三方应用获取用户身份的能力。

如今大多网站都已不再支持OpenID，最为流行的是OAuth 2.0 (在本文中提到OAuth也均指2.0版本)，而OpenID的最新版是OpenID Connect，是OpenID的第三代技术，下文会来介绍。

关于OAuth的介绍，网上非常之多，本文就不再过多叙述，而是主要讲解如何在 ASP.NET Core 中使用 OAuth 认证。如果你对 OAuth 并不了解，那么建议先去网上查看一下这方面的资料，再来阅读本文。而在本文中提到的OAuth认证指的是 ASP.NET Core 中的一种认证方式，而OAuth本身只是一种授权协议，希望不要混淆。

在 OAuth 协议中包含以下四种授权模式：

* 授权码模式（authorization code）
* 简化模式（implicit）
* 密码模式（resource owner password credentials）
* 客户端模式（client credentials）

在以上四种模式中，只有第一种Code模式需要服务端参与，其它的只在客户端就可完成，因此，在 ASP.NET Core 的 OAuth 认证中，也就只有Code模式。

### 用例

先来看一下具体的用法：

对于项目的创建可以参考[上一章](http://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#%E5%88%9B%E5%BB%BA%E9%A1%B9%E7%9B%AE)，然后添OAuth的Nuget包引用：

dotnet add package Microsoft.AspNetCore.Authentication.OAuth --version 2.0.0

然后在ConfigureServices配置服务：

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = OAuthDefaults.DisplayName;

})

.AddCookie()

.AddOAuth(OAuthDefaults.DisplayName, options =>

{

options.ClientId = "oauth.code";

options.ClientSecret = "secret";

options.AuthorizationEndpoint = "https://oidc.faasx.com/connect/authorize";

options.TokenEndpoint = "https://oidc.faasx.com/connect/token";

options.CallbackPath = "/signin-oauth";

options.Scope.Add("openid");

options.Scope.Add("profile");

options.Scope.Add("email");

options.SaveTokens = true;

// 事件执行顺序 ：

// 1.创建Ticket之前触发

options.Events.OnCreatingTicket = context => Task.CompletedTask;

// 2.创建Ticket失败时触发

options.Events.OnRemoteFailure = context => Task.CompletedTask;

// 3.Ticket接收完成之后触发

options.Events.OnTicketReceived = context => Task.CompletedTask;

// 4.Challenge时触发，默认跳转到OAuth服务器

// options.Events.OnRedirectToAuthorizationEndpoint = context => context.Response.Redirect(context.RedirectUri);

});

}

上面前六个参数是都必填的（在IdentityServer中Scope必须包含openid），否则会报错，SaveTokens属性用来设置是否将OAuth服务器返回的Token信息保存到AuthenticationProperties中。

[https://oidc.faasx.com](https://oidc.faasx.com/) 是我使用 [IdentityServer4](https://github.com/IdentityServer/IdentityServer4) 搭建的一个OIDC服务，源码地址在 [IdentityServerSample](https://github.com/RainingNight/AspNetCoreSample/tree/master/src/Functional/Identity/IdentityServerSample) ，而本文中并不会涉及到IdentityServer的相关知识，后续有机会再来单独介绍一下它。

最后，注册中间件：

public void Configure(IApplicationBuilder app)

{

app.UseAuthentication();

// 授权，与上一章Cookie认证中的实现一样

app.UseAuthorize();

// 我的信息

app.Map("/profile", builder => builder.Run(async context =>

{

await context.Response.WriteHtmlAsync(async res =>

{

await res.WriteAsync($"<h1>你好，当前登录用户： {HttpResponseExtensions.HtmlEncode(context.User.Identity.Name)}</h1>");

await res.WriteAsync("<a class=\"btn btn-default\" href=\"/Account/Logout\">退出</a>");

await res.WriteAsync($"<h2>AuthenticationType：{context.User.Identity.AuthenticationType}</h2>");

await res.WriteAsync("<h2>Claims:</h2>");

await res.WriteTableHeader(new string[] { "Claim Type", "Value" }, context.User.Claims.Select(c => new string[] { c.Type, c.Value }));

// 在第一章中介绍过HandleAuthenticateOnceAsync方法，在此调用并不会有多余的性能损耗。

var result = await context.AuthenticateAsync();

await res.WriteAsync("<h2>Tokens:</h2>");

await res.WriteTableHeader(new string[] { "Token Type", "Value" }, result.Properties.GetTokens().Select(token => new string[] { token.Name, token.Value }));

});

}));

// 退出

app.Map("/Account/Logout", builder => builder.Run(async context =>

{

await context.SignOutAsync();

context.Response.Redirect("/");

}));

// 首页

app.Run(async context =>

{

await context.Response.WriteHtmlAsync(async res =>

{

await res.WriteAsync($"<h2>Hello OAuth Authentication</h2>");

await res.WriteAsync("<a class=\"btn btn-default\" href=\"/profile\">我的信息</a>");

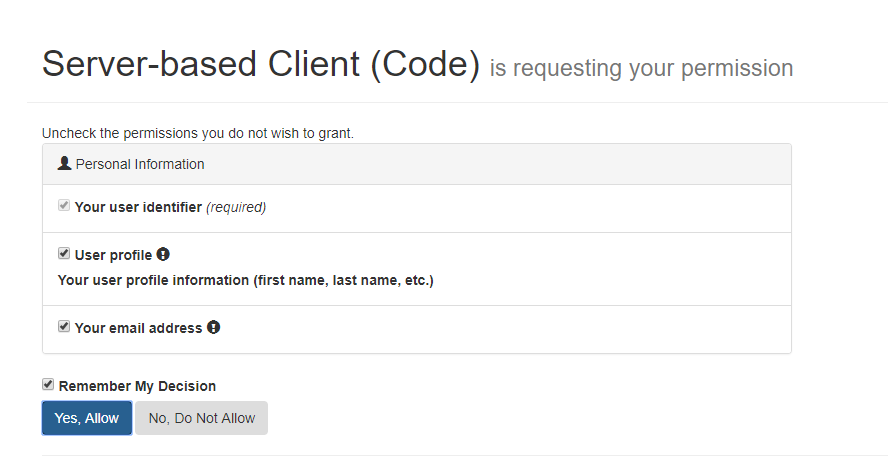
});

});

}

在 [上一章](http://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#remoteauthenticationhandler) 中有介绍到远程认证并不具备SignIn/SignOut的功能，而在这里的context.SignOutAsync()方法是由 [CookieHandler](http://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#cookieauthenticationhandler) 来执行的，因为我们指定了options.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme ，而DefaultSignOutScheme默认会使用DefaultSignInScheme中指定的值。

然后运行，访问：[http://localhost:5001](http://localhost:5001/)，点击 “我的信息” 按钮，将会跳转到OAuth服务器，登录成功后则显示授权页面：



点击允许，跳回我们的网站，并已登录成功，显示如下：



如图，**Cliams**中是空的，只有Token的相关信息，包含：access\_token, token\_type, expires\_at 三个值。后续，可以使用access\_token来访问OAuth服务方提供的受保护资源。这也就解释了为什么说OAuth只是授权，因为我们得到的只有一个本身无法识别的access\_token，而没有关于用户身份的任何信息，这也正是OAuth的本意。而国内却大多使用OAuth来做认证，以至于大多人都认为OAuth指的是认证，而非授权。虽然OAuth后来补充了 RFC7662 - OAuth2 Token Introspection 协议，让我们可以获取到用户的身份，但是并不建议使用，而是使用下面要介绍的OpenID Connect来做身份的认证。

下面再来简单介绍一下其运行流程。

### 源码分析

AddOAuth与上一章中介绍的[AddCookie](http://www.cnblogs.com/RainingNight/p/cookie-authentication-in-asp-net-core.html#addcookie)实现逻辑类似，而OAuthOptions中的参数，都是OAuth中的标准参数，不用多说。主要来介绍一下OAuthHandler，其用来完成获取Code，再使用Code获取AccessToken的整个流程：

public class OAuthHandler<TOptions> : RemoteAuthenticationHandler<TOptions> where TOptions : OAuthOptions, new()

{

...

protected override async Task<HandleRequestResult> HandleRemoteAuthenticateAsync()

{

// 获取从OAuth服务器返回的state

var state = query["state"];

var properties = Options.StateDataFormat.Unprotect(state);

// 获取从OAuth服务器返回的code

var code = query["code"];

var tokens = await ExchangeCodeAsync(code, BuildRedirectUri(Options.CallbackPath));

// ClaimsIssuer参数继承自父类：protected virtual string ClaimsIssuer => Options.ClaimsIssuer ?? Scheme.Name;

var identity = new ClaimsIdentity(ClaimsIssuer);

if (Options.SaveTokens)

{

// 保存Token信息到properties中，包括access\_token refresh\_token token\_type expires\_at

properties.StoreTokens(authTokens);

}

var ticket = await CreateTicketAsync(identity, properties, tokens);

return HandleRequestResult.Success(ticket);

}

// 使用上面获取到的授权码，拼装请求参数，然后调用TokenEndpoint，获取到Token。

protected virtual async Task<OAuthTokenResponse> ExchangeCodeAsync(string code, string redirectUri) { }

// 调用BuildChallengeUrl方法拼装请求参数，然后跳转。

protected override async Task HandleChallengeAsync(AuthenticationProperties properties) { }

}

补充一点，对于远程认证Handler，都只有请求路径（通常是认证服务器回调）与我们指定的CallbackPath一致时，才会执行，这一点在 [第一章](http://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#remoteauthenticationhandler)中也有介绍过。

以上代码简单展示了OAuth授权码模式的基本实现，完整的代码在：[OAuthHandler](https://github.com/aspnet/Security/blob/dev/src/Microsoft.AspNetCore.Authentication.OAuth/OAuthHandler.cs)，总的来说，OAuth认证还是比较简单的，我在这里再简单叙述一下。

授权码模式的整个流程如下：

+----------+

| Resource |

| Owner |

| |

+----------+

^

|

(B)

+----|-----+ Client Identifier +---------------+

| -+----(A)-- & Redirection URI ---->| |

| User- | | Authorization |

| Agent -+----(B)-- User authenticates --->| Server |

| | | |

| -+----(C)-- Authorization Code ---<| |

+-|----|---+ +---------------+

| | ^ v

(A) (C) | |

| | | |

^ v | |

+---------+ | |

| |>---(D)-- Authorization Code ---------' |

| Client | & Redirection URI |

| | |

| |<---(E)----- Access Token -------------------'

+---------+ (w/ Optional Refresh Token)

第一次未登录时访问，将会跳转到认证服务器并带上returnUrl参数，其附带 client\_id, scope, response\_type, redirect\_uri, state 五个参数，请求报文如下(为方便展示，都会使用URLDecode解码)：

GET /connect/authorize?client\_id=oauth.code&scope=openid profile email&response\_type=code&redirect\_uri=http://localhost:5001/signin-oauth&state=CfDJ8B4XRZETkRhMt3mT9VduB8K32v-jJapr\_X1RhEIiixwkk7L8krUsn32tBnyn3D0NX8PjPPpGtiAEG6O0bWI9ke42XhA0hrk-nI5nM86Fj9BDVQMoUwFJlrmT3QWBV7qTHWwPVWIXsK6lZR00owdKOqAL7g-9LjVv150V3NeBHD1P\_Jp9xiK1sN\_WywIbEUSwE\_ut\_c6w4V5nilEe6MqU-4JUoz5BTiqXDGG5kTd36ivGal4ihisn07csWFdodvC61A HTTP/1.1

Referer: http://localhost:5001/

其ReturnUrl的拼装代码如下：

protected virtual string BuildChallengeUrl(AuthenticationProperties properties, string redirectUri)

{

var scope = FormatScope();

var state = Options.StateDataFormat.Protect(properties);

var parameters = new Dictionary<string, string>

{

{ "client\_id", Options.ClientId },

{ "scope", scope },

{ "response\_type", "code" },

{ "redirect\_uri", redirectUri },

{ "state", state },

};

return QueryHelpers.AddQueryString(Options.AuthorizationEndpoint, parameters);

}

在OAuth服务登录成功后，返回如下：

HTTP/1.1 302 Found

Location: http://localhost:5001/signin-oauth?code=011e45b0f509969ac85aa69ab199636ddb33c13a06c711672b1be99509a5e205&scope=openid profile email&state=CfDJ8B4XRZETkRhMt3mT9VduB8K32v-jJapr\_X1RhEIiixwkk7L8krUsn32tBnyn3D0NX8PjPPpGtiAEG6O0bWI9ke42XhA0hrk-nI5nM86Fj9BDVQMoUwFJlrmT3QWBV7qTHWwPVWIXsK6lZR00owdKOqAL7g-9LjVv150V3NeBHD1P\_Jp9xiK1sN\_WywIbEUSwE\_ut\_c6w4V5nilEe6MqU-4JUoz5BTiqXDGG5kTd36ivGal4ihisn07csWFdodvC61A&session\_state=AwD762ldo1cgW58P0qKbK20amkXDIsIm\_GMm8oTas0Q.05716f0d6ff235cbc38da1c1d0508fa5

其 state 是我们在上一步中保存的 properties，code则是最重要的参数，用来获取 access\_token，由服务端来执行：

protected virtual async Task<OAuthTokenResponse> ExchangeCodeAsync(string code, string redirectUri)

{

var tokenRequestParameters = new Dictionary<string, string>()

{

{ "client\_id", Options.ClientId },

{ "redirect\_uri", redirectUri },

{ "client\_secret", Options.ClientSecret },

{ "code", code },

{ "grant\_type", "authorization\_code" },

};

var requestContent = new FormUrlEncodedContent(tokenRequestParameters);

var requestMessage = new HttpRequestMessage(HttpMethod.Post, Options.TokenEndpoint);

requestMessage.Headers.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

requestMessage.Content = requestContent;

var response = await Backchannel.SendAsync(requestMessage, Context.RequestAborted);

var payload = JObject.Parse(await response.Content.ReadAsStringAsync());

// payload：

// {

// "id\_token": "xxx",

// "access\_token": "xxx",

// "expires\_in": 3600,

// "token\_type": "Bearer"

// }

return OAuthTokenResponse.Success(payload);

}

最后调用Cookie认证的Sign方法写入Cookie，并跳转到我的信息页面：

HTTP/1.1 302 Found

Location: http://localhost:5001/profile

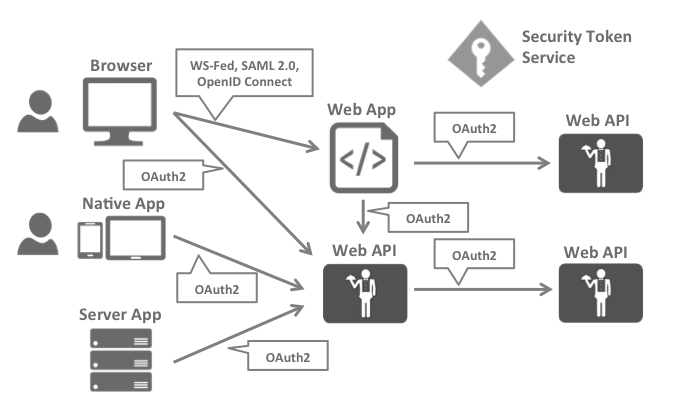
Set-Cookie: .AspNetCore.Correlation.OAuth.In32Oho-aTNH4EOvCWNZYSs--sYgA3eRfoJC9tZkgBY=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/signin-oauth; samesite=lax

Set-Cookie: .AspNetCore.Cookies=CfDJ8B4XRZETkRhMt3mT9VduB8JypeVkaj-mkVB8iXvf2tG2d8Xfs5CoX2wugjdlUkBv2DY73FAvPwPJOo81GpKqRJjjYzgwemGkB98ZTN7dbKI9rT\_\_Xwi-xsPZ-8gPCBtoeSnn1RfagM2kcprjk4djhBTrrJK1AVh3qufyu195Nju4Fqrmv91NKfelpztX0qaeVWS4y5cgbpKJwfeqJ3AfSSGkwnMFRIbKcX-TgJHKIDxQsP8OhAxm572GGv02X5WvpYZZF7Tc90zvNyH5HEzwv1nJ2yNuRUIgmSx6425M5RM684fq1fvaVIN29sORJCEj69gmw3xt7wpY3BsMYDyRXH1XSoa\_n7WEjvnT6lfy1zYqSLDM0MAMOlkBzAXRQ5Vjr8IVhTPkGOEsxT0jqHeJrVOzYD5PcuveP-oey3-n7OJyrKOtByu5qUfzI2Gs\_isUBiQSWcSfyQo9sLuB1Jj5dHtakCXOYUt8Hu2ysxUKugQiRSyh5WmxUm\_RBuV8\_QFzpD-3Ke5Brd1Kjl95Yo6iik66rfoHm1rcUOjKFGFvBl0be4uYz6-vhgauvhb-sa-gq-6uGLzNk\_Qm9l-vf1nJk7h\_qQ8OgwhjgNScrhMMt7r8DET3pX-\_gYg7Dl56KBGCpVX6nrBFgjjrdpN6kvDh6v26zrEpgYW6IHqVkje8HMgZbWe2PdNflrA9DmV6XtOxncoLc3EusWmpkUk-FCXG3lIzd4VlC8iim7fbCJHd5Z6Cn6b9cRGTTf4juvUcOWvTbHXi6HkT2f1Ym9eFZ-7BBghRWwc2fERPgxPZEcwgkSdEgoHPq\_eZtnshgHVTSM1e\_FUyiZxh8miVJbRRhzWgdR5xNW--lOD6ShdNH6-22dKKOZbPdxTkxraZl9SXslTdR1ILoD4Z23Jyi-rRZ42uPzCIrX4PnJIzm9HjFvjGQJedL8mm2tDaIuYQ2\_LBvyz8Wms9e0T\_VXJCaf53IE2rqAKahwxAV7kRDudEPNIp4y7pJ7djdhEahtdVUwiIh4Pz9y1p74zA42HeI2lUn6pTTetH\_npKn\_dqu1puge\_lXTncSH2yNqFZZTCQsNO6INjQjDMLRTkLGptQrjrMPz17MpFnb4lA1eVAo0R9EdlDIOAep2f2PuPzc-fVub5olnb8NjGUWy9J4rW6C-HBMj3sAlpZz9eHCYAOkRElKxeEgpyS1yOLA3469neukGKYFUySsTfDdEvU3JqsViFO3v\_8EwCe80eiePQv1l4SF7AWwIqQhZuXf-n4\_TXnoKFlZz9QqesA\_npYg1LgnrhHhUXEAhHvyehEKLRXFUDMQqIls6b\_WxWQ8d-9FBthl-WlZqMippZ1TJzGKLntsSXximbSnkGMuQfKNdESgIdUfvD1Dx8zbPVs\_2U87slOiCAwbrXPi9oVIFj8OuTBFKaf6NY8hh2aQ9ywDlekGCujnnh6EzZoPHuCVCNQ1bqKzAxnvAZ6Eg; path=/; samesite=lax; httponly

## OpenID Connect

OpenID Connect是OpenID的升级版，简称**OIDC**，是2014年初发布的开放标准，定义了一种基于OAuth2的可互操作的方式来来提供用户身份认证。在OIDC中，应用程序不必再为每个客户端构建不同的协议，而是可以将一个协议提供给多个客户端，它还使用了JOSN签名和加密规范，用来在传递携带签名和加密的信息，并使用简单的REST/JSON消息流来实现，和之前任何一种身份认证协议相比，开发者都可以轻松的集成。

上文中介绍到OAuth2是一个授权协议，它无法提供完善的身份认证功能。而OIDC使用OAuth2的授权服务器来为第三方客户端提供用户的身份认证，并把对应的身份认证信息传递给客户端，可以适用于各种类型的客户端（比如服务端应用，移动APP，JS应用），并完全兼容OAuth2，也就是说你搭建了一个OIDC的服务后，也可以当作一个OAuth2的服务来用（上面的OAuth服务器其实就是使用的OIDC服务器），应用场景如图：



OIDC在OAuth的基础上扩展了一些新的概念，避免了OAuth中的很多误区：

**ID Tokens**

OpenID Connect Id Token是一个签名的JSON Web Token（JWT：RFC7519），它包含一组关于用户身份的声明（claim），如：用户的标识（sub）、颁发令牌的提供程序的标识符（iss）、创建此标识的Client标识（aud），还包含token的有效期以及其他相关的上下文信息。

由于ID Token使用的是JWT签名，客户端可以直接解析出Token中的内容而无需依赖外部服务，因此我们可以使用ID Token来做身份认证，而不需要使用access\_token。不过，OIDC为了保持于OAuth的兼容，会同时提供Id token和access\_token。

**UserInfo Endpoint**

OIDC还提供了一个包含当前用户信息的标准的受保护的资源。UserInfo Endpoint不是身份认证的一部分，而是提供附加的标识信息，它提供了一组标准化的属性：比如profile、email、phone和address。OIDC中定义了一个特殊的openidscope，并且是必须的，它包含对Id token和UserInfo Endpoint的访问权限。

**服务发现和客户端注册**

OIDC定义了一个发现协议，客户端可以自动的获取有关如何与身份认证提供者进行交互的信息，还定义了一个客户端注册协议，允许客户端引入新的身份提供程序（identity providers）。通过这两种机制和一个通用的身份API，OIDC可以在互联网规模上良好的运行，而不需要任何一方事先知道对方的存在。

**Hybrid Flow**

混合流模式是授权码模式与隐式模式的组合，在一次请求中可以同时获取Code和ID Token，ResponseType可以是：code id\_token, code id\_token token 和 code token。

简单的介绍了一下OIDC的基本概念（网上对于OIDC的介绍很多，本文就不多叙述），下面就来介绍一下 ASP.NET Core 中的OIDC认证。

### 示例

首先添OpenIdConnect的Nuget包引用：

dotnet add package Microsoft.AspNetCore.Authentication.OpenIdConnect --version 2.0.0

OIDC的注册其实要比OAuth还简单些：

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;

})

.AddCookie()

.AddOpenIdConnect(o =>

{

o.ClientId = "oidc.hybrid";

o.ClientSecret = "secret";

// 若不设置Authority，就必须指定MetadataAddress

o.Authority = "https://oidc.faasx.com/";

// 默认为Authority+".well-known/openid-configuration"

//o.MetadataAddress = "https://oidc.faasx.com/.well-known/openid-configuration";

o.RequireHttpsMetadata = false;

// 使用混合流

o.ResponseType = OpenIdConnectResponseType.CodeIdToken;

// 是否将Tokens保存到AuthenticationProperties中

o.SaveTokens = true;

// 是否从UserInfoEndpoint获取Claims

o.GetClaimsFromUserInfoEndpoint = true;

// 在本示例中，使用的是IdentityServer，而它的ClaimType使用的是JwtClaimTypes。

o.TokenValidationParameters.NameClaimType = "name"; //JwtClaimTypes.Name;

// 以下参数均有对应的默认值，通常无需设置。

//o.CallbackPath = new PathString("/signin-oidc");

//o.SignedOutCallbackPath = new PathString("/signout-callback-oidc");

//o.RemoteSignOutPath = new PathString("/signout-oidc");

//o.Scope.Add("openid");

//o.Scope.Add("profile");

//o.ResponseMode = OpenIdConnectResponseMode.FormPost;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*相关事件\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// 未授权时，重定向到OIDC服务器时触发

//o.Events.OnRedirectToIdentityProvider = context => Task.CompletedTask;

// 获取到授权码时触发

//o.Events.OnAuthorizationCodeReceived = context => Task.CompletedTask;

// 接收到OIDC服务器返回的认证信息（包含Code, ID Token等）时触发

//o.Events.OnMessageReceived = context => Task.CompletedTask;

// 接收到TokenEndpoint返回的信息时触发

//o.Events.OnTokenResponseReceived = context => Task.CompletedTask;

// 验证Token时触发

//o.Events.OnTokenValidated = context => Task.CompletedTask;

// 接收到UserInfoEndpoint返回的信息时触发

//o.Events.OnUserInformationReceived = context => Task.CompletedTask;

// 出现异常时触发

//o.Events.OnAuthenticationFailed = context => Task.CompletedTask;

// 退出时，重定向到OIDC服务器时触发

//o.Events.OnRedirectToIdentityProviderForSignOut = context => Task.CompletedTask;

// OIDC服务器退出后，服务端回调时触发

//o.Events.OnRemoteSignOut = context => Task.CompletedTask;

// OIDC服务器退出后，客户端重定向时触发

//o.Events.OnSignedOutCallbackRedirect = context => Task.CompletedTask;

});

如上，ClientId，ClientSecret与在OAuth中的作用一样，而在这里不需要再分别指定各种Endpoint，是通过指定一个MetadataAddress地址来自动发现，可访问 <https://oidc.faasx.com/.well-known/openid-configuration> 来了解一下MetadataAddress中包含的信息。

然后，在上文OAuth认证示例中的Configure方法基础上添加signout和signout-remote：

// 本地退出

app.Map("/signout", builder => builder.Run(async context =>

{

await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);

await context.Response.WriteHtmlAsync(async res =>

{

await res.WriteAsync($"<h1>Signed out {HttpResponseExtensions.HtmlEncode(context.User.Identity.Name)}</h1>");

await res.WriteAsync("<a class=\"btn btn-default\" href=\"/\">Home</a>");

});

}));

// 远程退出

app.Map("/signout-remote", builder => builder.Run(async context =>

{

await context.SignOutAsync(OpenIdConnectDefaults.AuthenticationScheme, new AuthenticationProperties()

{

RedirectUri = "/signout"

});

}));

如上，远程退出使用的是OpenIdConnectDefaults.AuthenticationScheme，而本地退出则使用的CookieAuthenticationDefaults.AuthenticationScheme。

### 运行流程

#### 登录

来探索一下OIDC的运行流程，首先运行，点击一下我的信息：

请求：

GET http://localhost:5002/profile HTTP/1.1

响应：

HTTP/1.1 302 Found

Location: https://oidc.faasx.com/connect/authorize?client\_id=oidc.hybrid&redirect\_uri=http://localhost:5002/signin-oidc&response\_type=code id\_token&scope=openid profile&response\_mode=form\_post&nonce=636428853279287956.N2IxZmFlZDgtNDNmZC00OTRmLTljMWItNTVjMmQwOTVjNTQ3MWUyMDcxNTctZDg4Yy00MDRiLThmNmQtYjE1YTdjOGE4MThm&state=CfDJ8B4XRZETkRhMt3mT9VduB8LSACJO9seruKlM3kYPxaRyWcUSt0BvPMd6RUGAiay8qraTWLdMh9B3ClRJDE-BtMRYTmzGJSHegueIW-fyq2G9TpUtSQCd23BxAYrdB4SeGQte2IXaQ82cKMz-aSHQ7TTzhPO\_fgDtIVlwDJBtwgKQzEkEyyLsfH2DHxwr\_Ojn3M-uRHId2bi9RF2gR\_1hqoTdYlv-CZodFKuUGSMCqJO4cZLsuuAb-PrSnamz7h7MOpPixIOgQq5gd25sxF8avpSTsoT5HbU2fCiqX7g3rbCLzMG-rTnDftN8uZRiqc-JcyGkLPGIoj-FLNoW\_yfZbGk&x-client-SKU=ID\_NET&x-client-ver=2.1.4.0

Set-Cookie: .AspNetCore.OpenIdConnect.Nonce.CfDJ8B4XRZETkRhMt3mT9VduB8IHI9Q\_BeT6uPrlI72UUqej78UfqAdiczZsPOxF3Gy7bSm7Swh9Jh0\_haVi-UnQxUTGq-9xQLcaMX-PuXMjf6-6sLqc15NUgoLQ1w3KWqjkt7-3NlYW4qka6LqDPRtWJxT7vICtPhjx8ecNWtW\_ijqBg\_W8osmZLvFGS3PzPipP1UF14AkaIp48dFV1qNqt67Yta8ebXH7SHGkUhfcA5R-O0B8t-Q7sWL4NTN8AdKQF02HMqs\_duf-4FLf2p7Rbpsc=N; expires=Fri, 06 Oct 2017 11:30:27 GMT; path=/signin-oidc; httponly

Set-Cookie: .AspNetCore.Correlation.OpenIdConnect.BqW1filmAVCStL92ghXFQZBoLQjG5-Gl\_m60zxCW9BA=N; expires=Fri, 06 Oct 2017 11:30:27 GMT; path=/signin-oidc; httponly

由于我们还没有登录，会执行context.ChallengeAsync()方法，而我们在上面指定了DefaultChallengeScheme为：OpenIdConnectDefaults.AuthenticationScheme，因此会进入到**OpenIdConnectHandler**中来。

如上，重定向到了OIDC服务器，并写入了.AspNetCore.OpenIdConnect.Nonce.xxx和.AspNetCore.Correlation.OpenIdConnect.xxx两个Cookie，前者是OIDC的标准，会包含在ID Token中，用来减缓重放攻击，后者由Options.CorrelationCookie.Name + Scheme.Name + "." + correlationId组成， 用于防止CSRF。

而重定向地址中包含如下几个参数：

* **client\_id** 客户端标识，对应于OpenIdConnectOptions.ClientId。
* **redirect\_uri** 回调地址，对应于OpenIdConnectOptions.CallbackPath。
* **response\_type** 授权类型，对应于OpenIdConnectOptions.ResponseType。
* **scope** 权限范围，对应于OpenIdConnectOptions.Scope。
* **response\_mode** 响应模式，对应于OpenIdConnectOptions.ResponseMode，表示OIDC服务器来跳转到我们的应用时传递参数的方式。
* **nonce** OIDC服务器会在identity token中包含此参数，在认证时与Cookie中的.AspNetCore.OpenIdConnect.Nonce.xxx对比验证。
* **state** 用于保存状态，会原封不动地返回，在 ASP.NET Core 中，用AuthenticationProperties对象来表示。
* **x-client-SKU/x-client-ver** IdentityServer附加信息。

在OIDC服务器登录成功后，connect/authorize输出的是一个自动提交的表单：

<form method='post' action='http://localhost:5002/signin-oidc'>

<input type='hidden' name='code' value='eb53860906da276a1bb5318c5d539db085c6ca1fd3467da9d918aa7524f20f63' />

<input type='hidden' name='id\_token' value='eyJhbGciOiJSUzI1NiIsImtpZCI6IjdlYzk5MjVlMmUzMTA2NmY2ZmU2ODgzMDRhZjU1ZmM0IiwidHlwIjoiSldUIn0.eyJuYmYiOjE1MDcyODg1MjcsImV4cCI6MTUwNzI4ODgyNywiaXNzIjoiaHR0cDovL29pZGMuZmFhc3guY29tIiwiYXVkIjoib2lkYy5oeWJyaWQiLCJub25jZSI6IjYzNjQyODg1MzI3OTI4Nzk1Ni5OMkl4Wm1GbFpEZ3RORE5tWkMwME9UUm1MVGxqTVdJdE5UVmpNbVF3T1RWak5UUTNNV1V5TURjeE5UY3RaRGc0WXkwME1EUmlMVGhtTm1RdFlqRTFZVGRqT0dFNE1UaG0iLCJpYXQiOjE1MDcyODg1MjcsImNfaGFzaCI6ImJNT1FNVGdDV3VMX25EbHo5MDU4M3ciLCJzaWQiOiJjZTVjODg2ZmFhZDFiNTc4MDVkNzExYjkzZTliYWQ0ZiIsInN1YiI6IjAwMSIsImF1dGhfdGltZSI6MTUwNzI4ODM5NSwiaWRwIjoibG9jYWwiLCJhbXIiOlsicHdkIl19.BBn0SigvOW9USk-Mi1WP\_lJPWI9I06gsuomhqp69Ip5y3kqFyiBCVanzULsR4fBa0tOOBOtcPJzAfivzLqsMRwW1QfRamfVIlXuuzcRsR8WP1pFxvFPekwKi-D6-RLmmQzUT-\_78WvboiAu\_dZtwe0cm4ZLDCJH6LLCPs2xXTHYuNI7YoyAgeGKDAhWle0VrsbdlrcubPPgQFfFXPdLDInnLr8eEMpUZ7nru0FJgxm3Ah4hGXPKMud8jhLUMXDcSaKseL8tDgxIowmhpXOknU-y9x5FlrZUFOReDxaBZe7DG5V0xsPrdhMxMkZQbHHz8cJoaYrqcwHClm8rScEPxVA' />

<input type='hidden' name='scope' value='openid profile' />

<input type='hidden' name='state' value='CfDJ8B4XRZETkRhMt3mT9VduB8LSACJO9seruKlM3kYPxaRyWcUSt0BvPMd6RUGAiay8qraTWLdMh9B3ClRJDE-BtMRYTmzGJSHegueIW-fyq2G9TpUtSQCd23BxAYrdB4SeGQte2IXaQ82cKMz-aSHQ7TTzhPO\_fgDtIVlwDJBtwgKQzEkEyyLsfH2DHxwr\_Ojn3M-uRHId2bi9RF2gR\_1hqoTdYlv-CZodFKuUGSMCqJO4cZLsuuAb-PrSnamz7h7MOpPixIOgQq5gd25sxF8avpSTsoT5HbU2fCiqX7g3rbCLzMG-rTnDftN8uZRiqc-JcyGkLPGIoj-FLNoW\_yfZbGk' />

<input type='hidden' name='session\_state' value='g0m\_0W2scFKHxVQP\_8jdNt48r2XV8wDlF\_\_-ST9aYtk.28e1d34baad6d122a92c667329084600' />

</form>

<script>(function(){document.forms[0].submit();})();</script>

如上，可以看到，表单中包含有id\_token，因为我们使用的是code id\_token类型，然后便进入到了我们应用程序的OIDC认证逻辑中：

首先通过IdToken，可以来解析出AuthenticationProperties，ClaimsPrincipal等信息，然后使用Code，调用TokenEndpoint，获取access\_token等信息：

{

"id\_token": "....",

"access\_token": "...",

"expires\_in": 3600,

"token\_type": "Bearer"

}

因为我们将SaveTokens设置为true，则会将token信息保存到AuthenticationProperties中来：

public class OpenIdConnectHandler : RemoteAuthenticationHandler<OpenIdConnectOptions>, IAuthenticationSignOutHandler

{

protected override async Task<HandleRequestResult> HandleRemoteAuthenticateAsync()

{

...

if (Options.SaveTokens)

{

SaveTokens(properties, tokenEndpointResponse ?? authorizationResponse);

}

...

}

private void SaveTokens(AuthenticationProperties properties, OpenIdConnectMessage message)

{

var tokens = new List<AuthenticationToken>();

if (!string.IsNullOrEmpty(message.AccessToken))

{

tokens.Add(new AuthenticationToken { Name = OpenIdConnectParameterNames.AccessToken, Value = message.AccessToken });

}

if (!string.IsNullOrEmpty(message.IdToken))

{

tokens.Add(new AuthenticationToken { Name = OpenIdConnectParameterNames.IdToken, Value = message.IdToken });

}

if (!string.IsNullOrEmpty(message.RefreshToken))

{

tokens.Add(new AuthenticationToken { Name = OpenIdConnectParameterNames.RefreshToken, Value = message.RefreshToken });

}

if (!string.IsNullOrEmpty(message.TokenType))

{

tokens.Add(new AuthenticationToken { Name = OpenIdConnectParameterNames.TokenType, Value = message.TokenType });

}

if (!string.IsNullOrEmpty(message.ExpiresIn))

{

if (int.TryParse(message.ExpiresIn, NumberStyles.Integer, CultureInfo.InvariantCulture, out int value))

{

var expiresAt = Clock.UtcNow + TimeSpan.FromSeconds(value);

tokens.Add(new AuthenticationToken { Name = "expires\_at", Value = expiresAt.ToString("o", CultureInfo.InvariantCulture) });

}

}

properties.StoreTokens(tokens);

}

}

当我们做身份验证时，可能会需要更详细的用户Claims，可以将GetClaimsFromUserInfoEndpoint设置为True，使用UserInfoEndpoint返回的信息来重新创建ClaimsPrincipal对象：

protected override async Task<HandleRequestResult> HandleRemoteAuthenticateAsync()

{

...

if (Options.GetClaimsFromUserInfoEndpoint)

{

return await GetUserInformationAsync(tokenEndpointResponse ?? authorizationResponse, jwt, user, properties);

// UserInfoEndpoin返回的信息如下：

// {

// "name": "Alice Smith",

// "given\_name": "Alice",

// "family\_name": "Smith",

// "website": "http://alice.com",

// "sub": "001"

// }

}

else

{

var identity = (ClaimsIdentity)user.Identity;

foreach (var action in Options.ClaimActions)

{

action.Run(null, identity, ClaimsIssuer);

}

}

return HandleRequestResult.Success(new AuthenticationTicket(user, properties, Scheme.Name));

}

最后调用CookieHandler的SignInAsync方法，将AuthenticationTicket写入到Cookie中，响应如下：

HTTP/1.1 302 Found

Location: http://localhost:5002/profile

Set-Cookie: .AspNetCore.Correlation.OpenIdConnect.02q09RpgJBAi3rGZwy0WiyHUWGgLuDQIbVRUJuEXYBw=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/signin-oidc; samesite=lax

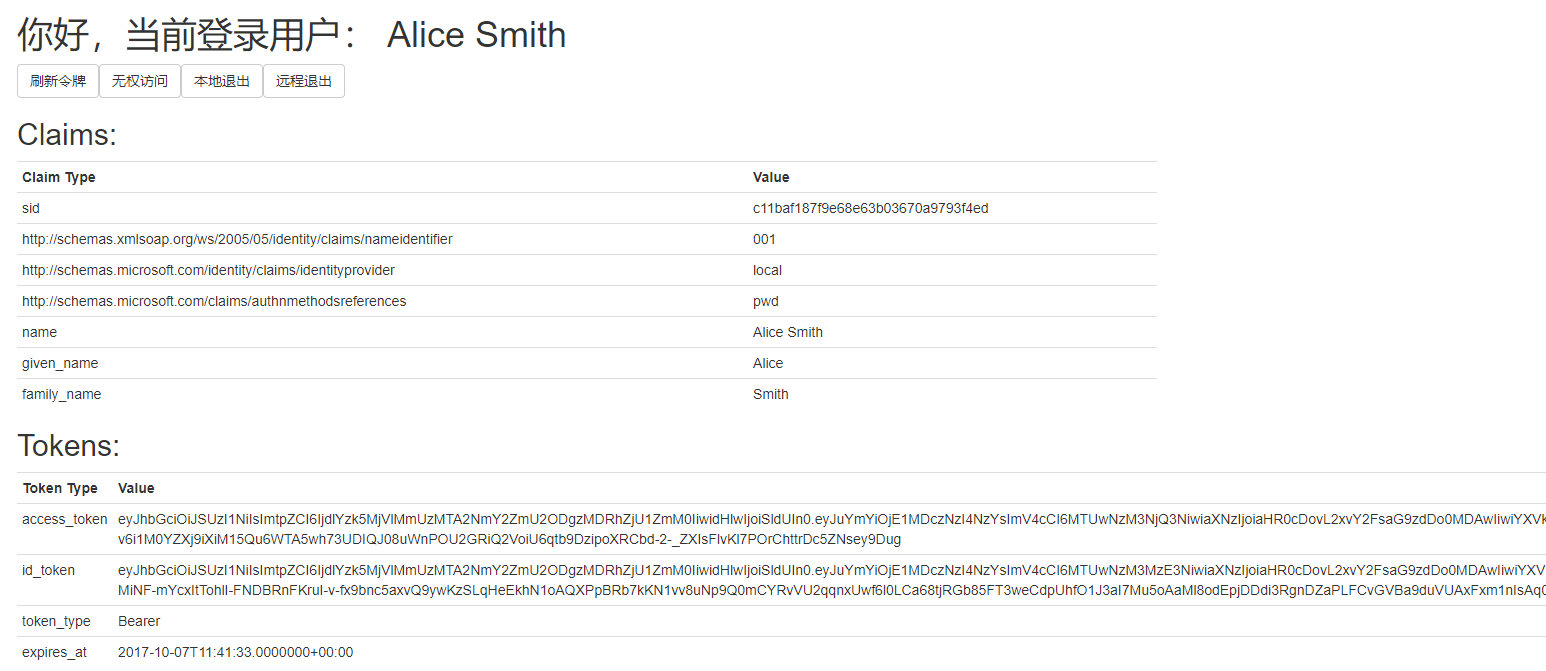
Set-Cookie: .AspNetCore.OpenIdConnect.Nonce.xxx=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/signin-oidc; samesite=lax

Set-Cookie: .AspNetCore.Cookies=chunks-2; path=/; samesite=lax; httponly

Set-Cookie: .AspNetCore.CookiesC1=xxx; path=/; samesite=lax; httponly

Set-Cookie: .AspNetCore.CookiesC2=xxx; path=/; samesite=lax; httponly

浏览器中显示如下：



接下来可以使用Token中的access\_token访问OIDC中的受保护资源（与OAuth用法一样），也可用使用Claims进行授权。可以看出，OIDC的认证流程比OAuth更加便捷和严谨，感兴趣的可以查看 [OpenIdConnectHandler](https://github.com/aspnet/Security/blob/dev/src/Microsoft.AspNetCore.Authentication.OpenIdConnect/OpenIdConnectHandler.cs) 的源码来更深一步的了解。

#### 退出

退出分为两种，一种是本地的退出，并不会使OIDC服务器退出，只需要简单的调用CookieHandler的SignOutAsync即可，无需多说。

而远程退出则会同时退出本地应用和OIDC服务器，大致逻辑是先跳转到OIDC服务器，退出后，OIDC服务器会回调本地应用，完成本地的退出，该回调地址是通过OpenIdConnectOptions.RemoteSignOutPath指定的。

当我们点击退出时，首先执行OpenIdConnectHandler的SignOutAsync方法，Http报文如下：

请求报文：

GET http://localhost:5002/signout-remote HTTP/1.1

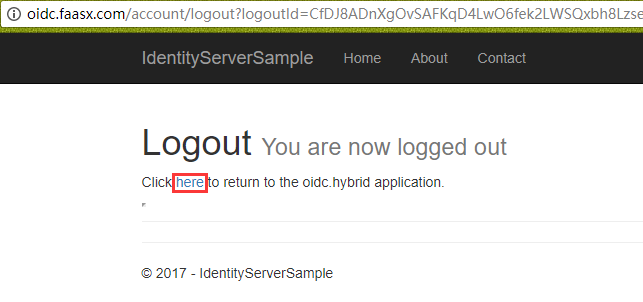
Cookie: .AspNetCore.Cookies=xxx

响应报文：

HTTP/1.1 302 Found

Location: http://oidc.faasx.com/connect/endsession?post\_logout\_redirect\_uri=http://localhost:5002/signout-callback-oidc&id\_token\_hint=eyJhbGciOiJSUzI1NiIsImtpZCI6IjdlYzk5MjVlMmUzMTA2NmY2ZmU2ODgzMDRhZjU1ZmM0IiwidHlwIjoiSldUIn0.eyJuYmYiOjE1MDc0MjgzMTAsImV4cCI6MTUwNzQyODYxMCwiaXNzIjoiaHR0cDovL29pZGMuZmFhc3guY29tIiwiYXVkIjoib2lkYy5oeWJyaWQiLCJub25jZSI6IjYzNjQzMDI1MTExOTUyNjYxNS5NVFU0WXpNMFl6RXRZemcwWmkwME5UWTFMVGcyTW1ZdE5XUmtZbUZtTUdabU5UWXlaalF6TkRReE9HWXROalEwWXkwMFpHTmtMVGhqTlRFdE1XWTROV0V3TW1NM09UZG0iLCJpYXQiOjE1MDc0MjgzMTAsImF0X2hhc2giOiJQSHRvd1JrUHhYLWNsclBzRnYxMkpnIiwic2lkIjoiMzFmMWJmMTA1MTU4MmZjMGE3ZTZhZjFjY2I0Y2RlMzUiLCJzdWIiOiIwMDEiLCJhdXRoX3RpbWUiOjE1MDc0MjgzMTAsImlkcCI6ImxvY2FsIiwiYW1yIjpbInB3ZCJdfQ.hIxkDhsx\_WE6IxM68O7uqkqdQquXXnOtxhlnrYiBJuU7Ex\_aApVXoKUdHS8HMx1nLswntr6SRsrygyMJnGMdzP5JutGsmfO\_i1WYGqk3BlTD7ry0wfBd\_U9OaVFcJhcVZq4q5u3SA47Wxqex9vifiHrTBQFT\_l6JqpevRLn-y91IxTl9rnXKfrHowhPsHJjdLzda3Lyj0wWWtb2N\_ng19mRChmDd4RXucP9mBHdQDyLZtvIJ5iIzV4pqtL7VCylFzV4RBLbCzUeuRnHI3E\_MqTaGWvVyFbUAKpr55TmVoc6lAOS4ie4CPzilR52KLWZ4l9eQh-WeIOA3NBgzHlJigg&state=CfDJ8B4XRZETkRhMt3mT9VduB8IbG5c9um6S\_maiHLsMKlIFRVtKMyLuXfqgB6e1OhWWVVJDYbgedt6hmyi1ny2aMbKW-SgHTru73YezUAZpre2ELXM3trlnX3YW\_FTkcGE\_RUyaR3hQ3eEYFmMdgdZdf0M&x-client-SKU=ID\_NET&x-client-ver=2.1.4.0

然后浏览器跳转到OIDC服务器，显示如下：



其HTML中还嵌套一个IFrame：

<iframe width="0" height="0" class="signout" src="http://oidc.faasx.com/connect/endsession/callback?endSessionId=CfDJ8ADnXgOvSAFKqD4LwO6fek3\_sYV1rgqtFD-CM4iSTjIo5wVq7lP0euy9tskf5BZ5hJHGweIMBQcnOcc4UR35xe94aaywrULsbUfA8n\_qNkVvtJbU0-EKMG2cadbhc6AHm06yyr8WpPEhZUvcVwlBFWNYnU9X6KErwyTE3oEe0yx-mOxWacIwWUbblQRjElil6PXICoR-0J6I4GfkPFRyHyja4EJz4IK\_Ik-vr1Lw\_CoAbpSQij2eIj54HfeE41TBJceoMNGJq8hJO\_ybL-CKma1hNJ\_sQ3Jc9h5uS8Y5oEig"></iframe>

该IFrame中的内容如下：

<!DOCTYPE html>

<html>

<style>iframe{display:none;width:0;height:0;}</style>

<body>

<iframe src='http://localhost:5002/signout-oidc?sid=854b9825a8cf571f7995e1ebafde8d37&iss=http%3A%2F%2Foidc.faasx.com'>

</iframe>

</body></html>

在用户毫无感觉的情况下，调用我们的应用服务器中配置的回调地址OpenIdConnectOptions.RemoteSignOutPath，清除本地的Cookie，实现同步退出：

protected virtual async Task<bool> HandleRemoteSignOutAsync()

{

OpenIdConnectMessage message = null;

if (string.Equals(Request.Method, "GET", StringComparison.OrdinalIgnoreCase))

{

message = new OpenIdConnectMessage(Request.Query.Select(pair => new KeyValuePair<string, string[]>(pair.Key, pair.Value)));

}

...

var remoteSignOutContext = new RemoteSignOutContext(Context, Scheme, Options, message);

await Events.RemoteSignOut(remoteSignOutContext);

...

await Context.SignOutAsync(Options.SignOutScheme);

return true;

}

而上图中的here链接则跳转到OpenIdConnectOptions.SignedOutCallbackPath，执行HandleSignOutCallbackAsync方法，我们可以通过注册事件的方式来附加一些业务逻辑：

protected async virtual Task<bool> HandleSignOutCallbackAsync()

{

var message = new OpenIdConnectMessage(Request.Query.Select(pair => new KeyValuePair<string, string[]>(pair.Key, pair.Value)));

...

var signOut = new RemoteSignOutContext(Context, Scheme, Options, message) { Properties = properties };

await Events.SignedOutCallbackRedirect(signOut);

...

if (!string.IsNullOrEmpty(properties?.RedirectUri))

{

Response.Redirect(properties.RedirectUri);

}

return true;

}

## 总结

本文简单介绍了OAuth和OpenID Connect的基本概念以及它们在 ASP.NET Core 中作为认证客户端的实现，如果我们只需要 "访问第三方资源" 的授权，使用OAuth认证即可。而在我们需要对自己的多个应用进行统一的身份验证时，应该使用OpenID Connect来实现，OpenID Connect不仅包含身份验证，还包含OAuth的授权协议，是更加推荐的做法。在下一章中来介绍一下另一种本地认证方式：**JWTBearer**，也是在现代Web应用中比较流行的认证方式。

[ASP.NET Core 认证与授权[4]:JwtBearer认证](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html)

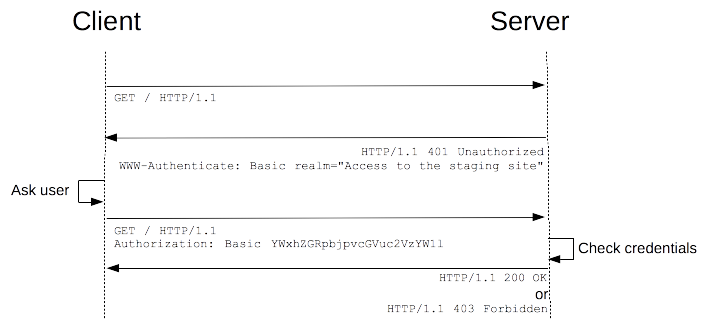
在现代Web应用程序中，通常会使用Web, WebApp, NativeApp等多种呈现方式，而后端也由以前的Razor渲染HTML，转变为Stateless的RESTFulAPI，因此，我们需要一种标准的，通用的，无状态的，与语言无关的认证方式，也就是本文要介绍的**JwtBearer**认证。

**目录**

1. [Bearer认证](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#bearer%E8%AE%A4%E8%AF%81)
2. [JWT(JSON WEB TOKEN)](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#jwtjson-web-token)
   * [头部(Header)](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E5%A4%B4%E9%83%A8header)
   * [载荷(Payload)](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E8%BD%BD%E8%8D%B7payload)
   * [签名(Signature)](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E7%AD%BE%E5%90%8Dsignature)
3. [示例](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E7%A4%BA%E4%BE%8B)
   * [模拟Token](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E6%A8%A1%E6%8B%9Ftoken)
   * [注册JwtBearer认证](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E6%B3%A8%E5%86%8Cjwtbearer%E8%AE%A4%E8%AF%81)
   * [添加受保护资源](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E6%B7%BB%E5%8A%A0%E5%8F%97%E4%BF%9D%E6%8A%A4%E8%B5%84%E6%BA%90)
   * [运行](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E8%BF%90%E8%A1%8C)
4. [扩展](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E6%89%A9%E5%B1%95)
   * [自定义Token获取方式](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E8%87%AA%E5%AE%9A%E4%B9%89token%E8%8E%B7%E5%8F%96%E6%96%B9%E5%BC%8F)
   * [使用OIDC服务](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#%E4%BD%BF%E7%94%A8oidc%E6%9C%8D%E5%8A%A1)
5. [源码探索](https://www.cnblogs.com/RainingNight/p/%E6%BA%90%E7%A0%81%E6%8E%A2%E7%B4%A2)
   * [JwtBearerPostConfigureOptions](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#jwtbearerpostconfigureoptions)
   * [JwtBearerHandler](https://www.cnblogs.com/RainingNight/p/jwtbearer-authentication-in-asp-net-core.html#jwtbearerhandler)

**Bearer认证**

HTTP提供了一套标准的身份验证框架：服务器可以用来针对客户端的请求发送质询(challenge)，客户端根据质询提供身份验证凭证。质询与应答的工作流程如下：服务器端向客户端返回401（Unauthorized，未授权）状态码，并在WWW-Authenticate头中添加如何进行验证的信息，其中至少包含有一种质询方式。然后客户端可以在请求中添加Authorization头进行验证，其Value为身份验证的凭证信息。



在HTTP标准验证方案中，我们比较熟悉的是"Basic"和"Digest"，前者将用户名密码使用BASE64编码后作为验证凭证，后者是Basic的升级版，更加安全，因为Basic是明文传输密码信息，而Digest是加密后传输。在前文介绍的Cookie认证属于Form认证，并不属于HTTP标准验证。

本文要介绍的**Bearer**验证也属于HTTP协议标准验证，它随着OAuth协议而开始流行，详细定义见： [RFC 6570](https://tools.ietf.org/html/rfc6750#section-1.2)。

A security token with the property that any party in possession of the token (a "bearer") can use the token in any way that any other party in possession of it can. Using a bearer token does not require a bearer to prove possession of cryptographic key material (proof-of-possession).

Bearer验证中的凭证称为BEARER\_TOKEN，或者是access\_token，它的颁发和验证完全由我们自己的应用程序来控制，而不依赖于系统和Web服务器，Bearer验证的标准请求方式如下：

Authorization: Bearer [BEARER\_TOKEN]

那么使用Bearer验证有什么好处呢？

* CORS: cookies + CORS 并不能跨不同的域名。而Bearer验证在任何域名下都可以使用HTTP header头部来传输用户信息。
* 对移动端友好: 当你在一个原生平台(iOS, Android, WindowsPhone等)时，使用Cookie验证并不是一个好主意，因为你得和Cookie容器打交道，而使用Bearer验证则简单的多。
* CSRF: 因为Bearer验证不再依赖于cookies, 也就避免了跨站请求攻击。
* 标准：在Cookie认证中，用户未登录时，返回一个302到登录页面，这在非浏览器情况下很难处理，而Bearer验证则返回的是标准的401 challenge。

**JWT(JSON WEB TOKEN)**

上面介绍的Bearer认证，其核心便是**BEARER\_TOKEN**，而最流行的Token编码方式便是：JSON WEB TOKEN。

Json web token (JWT), 是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准（[RFC 7519](https://tools.ietf.org/html/rfc7519)）。该token被设计为紧凑且安全的，特别适用于分布式站点的单点登录（SSO）场景。JWT的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源，也可以增加一些额外的其它业务逻辑所必须的声明信息，该token也可直接被用于认证，也可被加密。

JWT是由.分割的如下三部分组成：

**头部(Header)**

Header 一般由两个部分组成：

* alg
* typ

alg是是所使用的hash算法，如：HMAC SHA256或RSA，typ是Token的类型，在这里就是：**JWT**。

{

"alg": "HS256",

"typ": "JWT"

}

然后使用Base64Url编码成第一部分：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.<second part>.<third part>

**载荷(Payload)**

这一部分是JWT主要的信息存储部分，其中包含了许多种的声明（claims）。

Claims的实体一般包含用户和一些元数据，这些claims分成三种类型：

* **reserved claims**：预定义的 一些声明，并不是强制的但是推荐，它们包括 iss (issuer), exp (expiration time), sub (subject),aud(audience) 等（这里都使用三个字母的原因是保证 JWT 的紧凑）。
* **public claims**: 公有声明，这个部分可以随便定义，但是要注意和 IANA JSON Web Token 冲突。
* **private claims**: 私有声明，这个部分是共享被认定信息中自定义部分。

一个简单的Pyload可以是这样子的：

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

这部分同样使用Base64Url编码成第二部分：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.<third part>

**签名(Signature)**

Signature是用来验证发送者的JWT的同时也能确保在期间不被篡改。

在创建该部分时候你应该已经有了编码后的Header和Payload，然后使用保存在服务端的秘钥对其签名，一个完整的JWT如下：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

因此使用JWT具有如下好处：

* **通用**：因为json的通用性，所以JWT是可以进行跨语言支持的，像JAVA,JavaScript,NodeJS,PHP等很多语言都可以使用。
* **紧凑**：JWT的构成非常简单，字节占用很小，可以通过 GET、POST 等放在 HTTP 的 header 中，非常便于传输。
* **扩展**：JWT是自我包涵的，包含了必要的所有信息，不需要在服务端保存会话信息, 非常易于应用的扩展。

关于更多JWT的介绍，网上非常多，这里就不再多做介绍。下面，演示一下 ASP.NET Core 中 JwtBearer 认证的使用方式。

**示例**

**模拟Token**

ASP.NET Core 内置的JwtBearer验证，并不包含Token的发放，我们先模拟一个简单的实现：

[HttpPost("authenticate")]

public IActionResult Authenticate([FromBody]UserDto userDto)

{

var user = \_store.FindUser(userDto.UserName, userDto.Password);

if (user == null) return Unauthorized();

var tokenHandler = new JwtSecurityTokenHandler();

var key = Encoding.ASCII.GetBytes(Consts.Secret);

var authTime = DateTime.UtcNow;

var expiresAt = authTime.AddDays(7);

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(new Claim[]

{

new Claim(JwtClaimTypes.Audience,"api"),

new Claim(JwtClaimTypes.Issuer,"http://localhost:5200"),

new Claim(JwtClaimTypes.Id, user.Id.ToString()),

new Claim(JwtClaimTypes.Name, user.Name),

new Claim(JwtClaimTypes.Email, user.Email),

new Claim(JwtClaimTypes.PhoneNumber, user.PhoneNumber)

}),

Expires = expiresAt,

SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)

};

var token = tokenHandler.CreateToken(tokenDescriptor);

var tokenString = tokenHandler.WriteToken(token);

return Ok(new

{

access\_token = tokenString,

token\_type = "Bearer",

profile = new

{

sid = user.Id,

name = user.Name,

auth\_time = new DateTimeOffset(authTime).ToUnixTimeSeconds(),

expires\_at = new DateTimeOffset(expiresAt).ToUnixTimeSeconds()

}

});

}

如上，使用微软提供的*Microsoft.IdentityModel.Tokens*帮助类(源码地址：[azure-activedirectory-identitymodel-extensions-for-dotnet](https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet))，可以很容易的创建出JwtToen，就不再多说。

**注册JwtBearer认证**

首先添加JwtBearer包引用:

dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer --version 2.0.0

然后在Startup类中添加如下配置：

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(x =>

{

x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(o =>

{

o.TokenValidationParameters = new TokenValidationParameters

{

NameClaimType = JwtClaimTypes.Name,

RoleClaimType = JwtClaimTypes.Role,

ValidIssuer = "http://localhost:5200",

ValidAudience = "api",

IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(Consts.Secret))

/\*\*\*TokenValidationParameters的参数默认值\*\*\*\*\*\*\*/

// RequireSignedTokens = true,

// SaveSigninToken = false,

// ValidateActor = false,

// 将下面两个参数设置为false，可以不验证Issuer和Audience，但是不建议这样做。

// ValidateAudience = true,

// ValidateIssuer = true,

// ValidateIssuerSigningKey = false,

// 是否要求Token的Claims中必须包含Expires

// RequireExpirationTime = true,

// 允许的服务器时间偏移量

// ClockSkew = TimeSpan.FromSeconds(300),

// 是否验证Token有效期，使用当前时间与Token的Claims中的NotBefore和Expires对比

// ValidateLifetime = true

};

});

}

public void Configure(IApplicationBuilder app)

{

app.UseAuthentication();

}

在JwtBearerOptions的配置中，通常IssuerSigningKey(签名秘钥), ValidIssuer(Token颁发机构), ValidAudience(颁发给谁) 三个参数是必须的，后两者用于与TokenClaims中的Issuer和Audience进行对比，不一致则验证失败（与上面发放Token中的Claims对应）。

而NameClaimType和RoleClaimType需与Token中的ClaimType一致，在IdentityServer中也是使用的JwtClaimTypes，否则会造成User.Identity.Name为空等问题。

**添加受保护资源**

创建一个需要授权的控制器，直接使用Authorize即可：

[Authorize]

[Route("api/[controller]")]

public class SampleDataController : Controller

{

[HttpGet("[action]")]

public IEnumerable<WeatherForecast> WeatherForecasts()

{

return ...

}

}

**运行**

最后运行，直接访问/api/SampleData/WeatherForecasts，将返回一个401:

HTTP/1.1 401 Unauthorized

Server: Kestrel

Content-Length: 0

WWW-Authenticate: Bearer

让我们调用api/oauth/authenticate，获取一个JWT:

请求：

POST http://localhost:5200/api/oauth/authenticate HTTP/1.1

content-type: application/json

{

"username": "alice",

"password": "alice"

}

响应：

HTTP/1.1 200 OK

{"access\_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjEiLCJuYW1lIjoiYWxpY2UiLCJlbWFpbCI6ImFsaWNlQGdtYWlsLmNvbSIsInBob25lX251bWJlciI6IjE4ODAwMDAwMDAxIiwibmJmIjoxNTA5NDY0MzQwLCJleHAiOjE1MTAwNjkxNDAsImlhdCI6MTUwOTQ2NDM0MH0.Y1TDz8KjLRh\_vjQ\_3iYP4oJw-fmhoboiAGPqIZ-ooNc","token\_type":"Bearer","profile":{"sid":1,"name":"alice","auth\_time":1509464340,"expires\_at":1510069140}}

最后使用该Token，再次调用受保护资源：

GET http://localhost:5200/api/SampleData/WeatherForecasts HTTP/1.1

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjEiLCJuYW1lIjoiYWxpY2UiLCJlbWFpbCI6ImFsaWNlQGdtYWlsLmNvbSIsInBob25lX251bWJlciI6IjE4ODAwMDAwMDAxIiwibmJmIjoxNTA5NDY0MzQwLCJleHAiOjE1MTAwNjkxNDAsImlhdCI6MTUwOTQ2NDM0MH0.Y1TDz8KjLRh\_vjQ\_3iYP4oJw-fmhoboiAGPqIZ-ooNc

授权成功，返回了预期的数据：

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

[{"dateFormatted":"2017/11/3","temperatureC":35,"summary":"Chilly","temperatureF":94}]

**扩展**

**自定义Token获取方式**

JwtBearer认证中，默认是通过Http的Authorization头来获取的，这也是最推荐的做法，但是在某些场景下，我们可能会使用Url或者是Cookie来传递Token，那要怎么来实现呢？

其实实现起来非常简单，如前几章介绍的一样，JwtBearer也在认证的各个阶段为我们提供了事件，来执行我们的自定义逻辑：

.AddJwtBearer(o =>

{

o.Events = new JwtBearerEvents()

{

OnMessageReceived = context =>

{

context.Token = context.Request.Query["access\_token"];

return Task.CompletedTask;

}

};

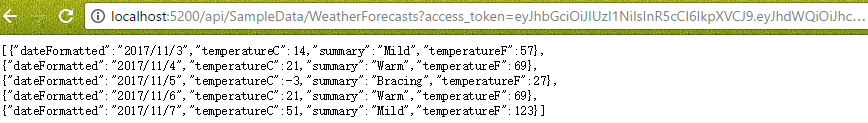
o.TokenValidationParameters = new TokenValidationParameters

{

...

};

然后在Url中添加access\_token=[token]，直接在浏览器中访问：



同样的，我们也可以很容易的在Cookie中读取Token，就不再演示。

除了OnMessageReceived外，还提供了如下几个事件：

* **TokenValidated**：在Token验证通过后调用。
* **AuthenticationFailed**: 认证失败时调用。
* **Challenge**: 未授权时调用。

**使用OIDC服务**

在上面的示例中，我们简单模拟的Token颁发，功能非常简单，并不适合在生产环境中使用，可是微软也没有提供OIDC服务的实现，好在.NET社区中提供了几种实现，可供我们选择：

| **Name** | **Description** |
| --- | --- |
| [AspNet.Security.OpenIdConnect.Server (ASOS)](https://github.com/aspnet-contrib/AspNet.Security.OpenIdConnect.Server) | Low-level/protocol-first OpenID Connect server framework for ASP.NET Core and OWIN/Katana |
| [IdentityServer4](https://identityserver.io/) | OpenID Connect and OAuth 2.0 framework for ASP.NET Core - officially certified by the OpenID Foundation and under governance of the .NET Foundation |
| [OpenIddict](https://github.com/openiddict/openiddict-core) | Easy-to-use OpenID Connect server for ASP.NET Core |
| [PwdLess](https://github.com/pwdless/pwdless) | Simple, stateless, passwordless authentication for ASP.NET Core |

我们在这里使用IdentityServer4来搭建一个OIDC服务器，并添加如下配置：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*OIDC服务器代码片段\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

// 配置IdentitryServer

services.AddIdentityServer()

.AddInMemoryPersistedGrants()

.AddInMemoryApiResources(Config.GetApis())

.AddInMemoryIdentityResources(Config.GetIdentityResources())

.AddInMemoryClients(Config.GetClients())

.AddTestUsers(Config.GetUsers())

.AddDeveloperSigningCredential();

}

new Client

{

ClientId = "jwt.implicit",

ClientName = "Implicit Client (Web)",

AllowedGrantTypes = GrantTypes.Implicit,

AllowAccessTokensViaBrowser = true,

RedirectUris = { "http://localhost:5200/callback" },

PostLogoutRedirectUris = { "http://localhost:5200/home" },

AllowedCorsOrigins = { "http://localhost:5200" },

AllowedScopes = { "openid", "profile", "email", "api" },

}

而JwtBearer客户端的配置就更加简单了，因为OIDC具有配置发现的功能：

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(x =>

{

x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(o =>

{

o.Authority = "https://oidc.faasx.com/";

o.Audience = "api";

o.TokenValidationParameters = new TokenValidationParameters

{

NameClaimType = JwtClaimTypes.Name,

RoleClaimType = JwtClaimTypes.Role,

};

});

}

如上，最重要的是Authority参数，用来表示OIDC服务的地址，然后便可以自动发现Issuer, IssuerSigningKey等配置，而o.Audience与o.TokenValidationParameters = new TokenValidationParameters { ValidAudience = "api" }是等效的，后面分析源码时会介绍。

OIDC兼容OAuth2协议，我们可以使用[上一章](http://www.cnblogs.com/RainingNight/p/oidc-authentication-in-asp-net-core.html)介绍的授权码模式来获取Token，也可以直接用户名密码模式来获取Token：

请求：

POST https://oidc.faasx.com/connect/token HTTP/1.1

Content-Type: application/x-www-form-urlencoded

client\_id=client.rop&client\_secret=secret&grant\_type=password&scope=api&username=alice&password=alice

响应：

HTTP/1.1 200 OK

Content-Type: application/json

{"access\_token":"eyJhbGciOiJSUzI1NiIsImtpZCI6IjdlYzk5MjVlMmUzMTA2NmY2ZmU2ODgzMDRhZjU1ZmM0IiwidHlwIjoiSldUIn0.eyJuYmYiOjE1MDk2NzI1NjksImV4cCI6MTUwOTY3NjE2OSwiaXNzIjoiaHR0cHM6Ly9vaWRjLmZhYXN4LmNvbSIsImF1ZCI6WyJodHRwczovL29pZGMuZmFhc3guY29tL3Jlc291cmNlcyIsImFwaSJdLCJjbGllbnRfaWQiOiJjbGllbnQucm9wIiwic3ViIjoiMDAxIiwiYXV0aF90aW1lIjoxNTA5NjcyNTY5LCJpZHAiOiJsb2NhbCIsIm5hbWUiOiJBbGljZSBTbWl0aCIsImVtYWlsIjoiQWxpY2VTbWl0aEBlbWFpbC5jb20iLCJzY29wZSI6WyJhcGkiXSwiYW1yIjpbInB3ZCJdfQ.PM93LThOZA3lkgPFVwieqGQQQtgmYDCY0oSFVmudv1hpKO6UaaZsmnn4ci9QjbGl5g2433JkDks5UIZsZ0xE62Qqq8PicPBBuaNoYrCf6dxR7j-0uZcoa7-FCKGu-0TrM8OL-NuMvN6\_KEpbWa3jlkwibCK9YDIwJZilVoWUOrbbIEsKTa-DdLScmzHLUzksT8GBr0PAVhge9PRFiGqg8cgMLjsA62ZeDsR35f55BucSV5Pj0SAj26anYvrBNTHKOF7ze1DGW51Dbz6DRu1X7uEIxSzWiNi4cRVJ6Totjkwk5F78R9R38o\_mYEdehZBjRHFe6zLd91hXcCKqOEh5eQ","expires\_in":3600,"token\_type":"Bearer"}

我们使用[https://jwt.io](https://jwt.io/)解析一下OIDC服务器颁发的Token中的Claims:

{

"nbf": 1509672569, // 2017/11/3 1:29:29 NotBefore Token生效时间，在此之前不可用

"exp": 1509676169, // 2017/11/3 2:29:29 Expiration Token过期时间，在此之后不可用

"iss": "https://oidc.faasx.com", // Issuer 颁发者，通常为STS服务器地址

"aud": [ // Audience Token的作用对象，也就是被访问的资源服务器授权标识

"https://oidc.faasx.com/resources",

"api"

],

"client\_id": "client.rop", // 客户端标识

"sub": "001",

"auth\_time": 1509672569, // Token颁发时间

"idp": "local",

"name": "Alice Smith",

"email": "AliceSmith@email.com",

"scope": [

"api"

],

"amr": [

"pwd"

]

}

我在本章的示例代码中，使用前端Angular框架演示了如何从本地登录获取Tokek或使用简化模式（implicit）从OIDC服务器获取Token，然后保存到sesstionStorage，在发送请求时附加到请求头中的示例，可供大家参考：[JwtBearerSample](https://github.com/RainingNight/AspNetCoreSample/tree/master/src/Functional/Authentication/JwtBearerSample)。

**源码探索**

**JwtBearerPostConfigureOptions**

在ASP.NET Core 2.0 Options框架中，新增了一种[PostConfigure](http://www.cnblogs.com/RainingNight/p/strongly-typed-options-configure-in-asp-net-core.html)模式，用来在我们所注册的Options配置执行完之后，再对Options做一些修改。

JwtBearerPostConfigureOptions用来实现配置发现：

public class JwtBearerPostConfigureOptions : IPostConfigureOptions<JwtBearerOptions>

{

public void PostConfigure(string name, JwtBearerOptions options)

{

// 如果未设置options.TokenValidationParameters.ValidAudience，则使用options.Audience

if (string.IsNullOrEmpty(options.TokenValidationParameters.ValidAudience) && !string.IsNullOrEmpty(options.Audience))

{

options.TokenValidationParameters.ValidAudience = options.Audience;

}

if (options.ConfigurationManager == null)

{

// 如果未设置MetadataAddress，则使用options.Authority+.well-known/openid-configuration

....

options.ConfigurationManager = new ConfigurationManager<OpenIdConnectConfiguration>(options.MetadataAddress, new OpenIdConnectConfigurationRetriever(), new HttpDocumentRetriever(httpClient) { RequireHttps = options.RequireHttpsMetadata });

}

}

}

}

**JwtBearerHandler**

JwtBearerHandler相对于前几章介绍的CookieHandler, OpenIdConnectHandler等，都简单的多。

首先便是从请求中获取Token：

protected override async Task<AuthenticateResult> HandleAuthenticateAsync()

{

var messageReceivedContext = new MessageReceivedContext(Context, Scheme, Options);

// 先触发MessageReceived事件，来获取Token

await Events.MessageReceived(messageReceivedContext);

if (messageReceivedContext.Result != null)

{

return messageReceivedContext.Result;

}

token = messageReceivedContext.Token;

// Token为空时，从Authorization头中获取

if (string.IsNullOrEmpty(token))

{

string authorization = Request.Headers["Authorization"];

if (string.IsNullOrEmpty(authorization))

{

return AuthenticateResult.NoResult();

}

if (authorization.StartsWith("Bearer ", StringComparison.OrdinalIgnoreCase))

{

token = authorization.Substring("Bearer ".Length).Trim();

}

if (string.IsNullOrEmpty(token))

{

return AuthenticateResult.NoResult();

}

}

...

}

然后初始化TokenValidationParameters参数，为Token验证做准备：

if (\_configuration == null && Options.ConfigurationManager != null)

{

\_configuration = await Options.ConfigurationManager.GetConfigurationAsync(Context.RequestAborted);

}

var validationParameters = Options.TokenValidationParameters.Clone();

if (\_configuration != null)

{

var issuers = new[] { \_configuration.Issuer };

validationParameters.ValidIssuers = validationParameters.ValidIssuers?.Concat(issuers) ?? issuers;

validationParameters.IssuerSigningKeys = validationParameters.IssuerSigningKeys?.Concat(\_configuration.SigningKeys)

?? \_configuration.SigningKeys;

}

可以看到，从OIDC服务器提供的配置发现中，获取ValidIssuers和IssuerSigningKeys。

最后对Token进行验证：

// Options.SecurityTokenValidators 默认为： new List<ISecurityTokenValidator> { new JwtSecurityTokenHandler() }

foreach (var validator in Options.SecurityTokenValidators)

{

if (validator.CanReadToken(token))

{

ClaimsPrincipal principal;

try

{

principal = validator.ValidateToken(token, validationParameters, out validatedToken);

}

catch (Exception ex)

{

// RefreshOnIssuerKeyNotFound默认为True, 在SignatureKey未找到时，重新从OIDC服务器获取

if (Options.RefreshOnIssuerKeyNotFound && Options.ConfigurationManager != null

&& ex is SecurityTokenSignatureKeyNotFoundException)

{

Options.ConfigurationManager.RequestRefresh();

}

continue;

}

...

// 触发TokenValidated事件

await Events.TokenValidated(tokenValidatedContext);

// 默认为true，保存Token到`AuthenticationProperties`中，可以通过`context.AuthenticateAsync()`来获取，在我们需要在服务端使用用户Token调用其他资源是非常有用。

if (Options.SaveToken)

{

tokenValidatedContext.Properties.StoreTokens(new[]

{

new AuthenticationToken { Name = "access\_token", Value = token }

});

}

// 验证成功

tokenValidatedContext.Success();

return tokenValidatedContext.Result;

}

}

其核心的验证也是在Microsoft.IdentityModel.Tokens中，就不在深究。

当使用JwtBearer认证时，我们肯定不希望在未登录时返回一个302，因此在前面的示例中，我们配置了x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;，对应的，会执行JwtBearerHandler的HandleChallengeAsync方法：

protected override async Task HandleChallengeAsync(AuthenticationProperties properties)

{

var authResult = await HandleAuthenticateOnceSafeAsync();

var eventContext = new JwtBearerChallengeContext(Context, Scheme, Options, properties)

{

AuthenticateFailure = authResult?.Failure

};

if (Options.IncludeErrorDetails && eventContext.AuthenticateFailure != null)

{

eventContext.Error = "invalid\_token";

eventContext.ErrorDescription = CreateErrorDescription(eventContext.AuthenticateFailure);

}

await Events.Challenge(eventContext);

if (eventContext.Handled)

{

return;

}

Response.StatusCode = 401;

// 最终将相应报文拼接成如下：

// https://tools.ietf.org/html/rfc6750#section-3.1

// WWW-Authenticate: Bearer realm="example", error="invalid\_token", error\_description="The access token expired"

}

ASP.NET Core JwtBearer认证的完整源码地址：[Microsoft.AspNetCore.Authentication.JwtBearer](https://github.com/aspnet/Security/tree/master/src/Microsoft.AspNetCore.Authentication.JwtBearer)。

本文示例代码地址：<https://github.com/RainingNight/AspNetCoreSample/tree/master/src/Functional/Authentication/JwtBearerSample>

**总结**

JwtToken其实与Cookie认证中加密后的Cookie值很像，他们都是基于Claim的，认证时无需STS(Security token service)的参与，这在分布式环境下提供了极大的便利。而他们的本质上的区别是：Cookie是微软式的，很难与其他语言集成，而JwtToken则是开放再开放，与平台，语言无关，在前端也可以直接解析出Claims。

PS: 在使用在Bearer认证时，通常还需与刷新Token配合来使用，因为JwtToken的验证是无需经过STS的，而当用户执行了退出，修改密码等操作时，是无法使该Token失效的。所以，通常会给access\_token设置一个较短的有效期(JwtBearer认证默认会验证有效期，通过notBefore和expires来验证)，当access\_token过期后，可以在用户无感知的情况下，使用refresh\_token自动从STS重新获取access\_token，但这就不属于Bearer认证的范畴了，在后续介绍IdentityServer时再来详细介绍一下。

# [ASP.NET Core 认证与授权[5]:初识授权](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html)

经过前面几章的姗姗学步，我们了解了在 ASP.NET Core 中是如何认证的，终于来到了授权阶段。在认证阶段我们通过用户令牌获取到用户的Claims，而授权便是对这些的Claims的验证，如：是否拥有Admin的角色，姓名是否叫XXX等等。本章就来介绍一下 ASP.NET Core 的授权系统的简单使用。

**目录**

1. [简单授权](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#%E7%AE%80%E5%8D%95%E6%8E%88%E6%9D%83)
   * [IAllowAnonymous](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#iallowanonymous)
   * [IAuthorizeData](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#iauthorizedata)
2. [授权策略详解](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#%E6%8E%88%E6%9D%83%E7%AD%96%E7%95%A5%E8%AF%A6%E8%A7%A3)
   * [AddAuthorization](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#addauthorization)
   * [AuthorizationOptions](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#authorizationoptions)
   * [AuthorizationPolicy](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#authorizationpolicy)
   * [IAuthorizationRequirement](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#iauthorizationrequirement)
   * [AuthorizationPolicyBuilder](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#authorizationpolicybuilder)
3. [基于策略的授权进阶](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#%E5%9F%BA%E4%BA%8E%E7%AD%96%E7%95%A5%E7%9A%84%E6%8E%88%E6%9D%83%E8%BF%9B%E9%98%B6)
   * [自定义策略](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#%E8%87%AA%E5%AE%9A%E4%B9%89%E7%AD%96%E7%95%A5)
   * [多Handler模式](https://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#%E5%A4%9Ahandler%E6%A8%A1%E5%BC%8F)

## 简单授权

在ASP.NET 4.x中，我们通常使用Authorize过滤器来进行授权，它可以作用在Controller和Action上面，也可以添加到全局过滤器中。而在ASP.NET Core中也有一个Authorize特性（但不是过滤器），用法类似：

[Authorize] // Controller级别

public class SampleDataController : Controller

{

[Authorize] // Action级别

public IActionResult SampleAction()

{

}

}

### IAllowAnonymous

在ASP.NET 4.x中，我们最常用的另一个特性便是AllowAnonymous，用来设置某个Controller或者Action跳过授权，它在 ASP.NET Core 中同样适用：

[Authorize]

public class AccountController : Controller

{

[AllowAnonymous]

public ActionResult Login()

{

}

public ActionResult Logout()

{

}

}

如上，LoginAction便不再需要授权，同样，在 ASP.NET Core 中提供了一个统一的IAllowAnonymous接口，在授权逻辑中都是通过该接口来判断是否跳过授权验证的。

public interface IAllowAnonymous

{

}

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false, Inherited = true)]

public class AllowAnonymousAttribute : Attribute, IAllowAnonymous

{

}

### IAuthorizeData

上面提到，在 ASP.NET Core 中，AuthorizeAttribute不再是一个MVC中的Filter了，而只是一个简单的实现了IAuthorizeData接口的**Attribute**：

public interface IAuthorizeData

{

string Policy { get; set; }

string Roles { get; set; }

string AuthenticationSchemes { get; set; }

}

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

public class AuthorizeAttribute : Attribute, IAuthorizeData

{

public AuthorizeAttribute() { }

public AuthorizeAttribute(string policy)

{

Policy = policy;

}

public string Policy { get; set; }

public string Roles { get; set; }

public string AuthenticationSchemes { get; set; }

}

记得第一次在ASP.NET Core中实现自定义授权时，按照以前的经验，直接继承自AuthorizeAttribute，然后准备重写OnAuthorization方法，结果懵逼了。然后在MVC的源码中，苦苦搜寻AuthorizeAttribute的踪迹，却毫无所获，后来才注意到它实现了IAuthorizeData接口，该接口才是认证的源头，而Authorize特性只是认证信息的载体，并不包含任何逻辑。IAuthorizeData中定义的Policy, Roles, AuthenticationSchemes三个属性分别代表着 ASP.NET Core 授权系统中的三种授权方式。

#### 基于角色的授权

基于角色的授权，我们都比较熟悉，使用方式如下：

[Authorize(Roles = "Admin")] // 多个Role可以使用,分割

public class SampleDataController : Controller

{

...

}

基于角色的授权的逻辑与ASP.NET 4.x类似，都是使用我在《[初识认证](http://www.cnblogs.com/RainingNight/p/introduce-basic-authentication-in-asp-net-core.html#claimsprincipal)》中介绍的IsInRole方法来实现的。

#### 基于Scheme的授权

对于AuthenticationScheme我在前面几章也都介绍过，比如Cookie认证默认使用的AuthenticationScheme就是Cookies，在JwtBearer认证中，默认的Scheme就是Bearer。

当初在学习认证时，还在疑惑，如何在使用Cookie认证的同时又支持Bearer认证呢？在认证中明明只能设置一个Scheme来执行。当看到这里时，豁然开朗，后面会详细介绍。

[Authorize(AuthenticationSchemes = "Cookies")] // 多个Scheme可以使用,分割

public class SampleDataController : Controller

{

...

}

当我们的应用程序中，同时使用了多种认证Scheme时，AuthenticationScheme授权就非常有用，在该授权模式下，会通过context.AuthenticateAsync(scheme)重新获取Claims。

#### 基于策略的授权

在ASP.NET Core中，重新设计了一种更加灵活的授权方式：基于策略的授权，也是授权的核心。

在使用基于策略的授权时，首先要定义授权策略，而授权策略本质上就是对Claims的一系列断言。

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.AddAuthorization(options =>

{

options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("EmployeeNumber"));

});

}

如上，我们定义了一个名称为EmployeeOnly的授权策略，它要求用户的Claims中必须包含类型为EmployeeNumber的Claim。

其实，基于角色的授权和基于Scheme的授权，只是一种语法上的便捷，最终都会生成授权策略，后文会详解介绍。

然后便可以在Authorize特性中通过Policy属性来指定授权策略：

[Authorize(Policy = "EmployeeOnly")]

public class SampleDataController : Controller

{

}

## 授权策略详解

### AddAuthorization

授权策略的定义使用了AddAuthorization扩展方法，我们来看看它的源码：

public static class AuthorizationServiceCollectionExtensions

{

public static IServiceCollection AddAuthorization(this IServiceCollection services)

{

services.TryAdd(ServiceDescriptor.Transient<IAuthorizationService, DefaultAuthorizationService>());

services.TryAdd(ServiceDescriptor.Transient<IAuthorizationPolicyProvider, DefaultAuthorizationPolicyProvider>());

services.TryAdd(ServiceDescriptor.Transient<IAuthorizationHandlerProvider, DefaultAuthorizationHandlerProvider>());

services.TryAdd(ServiceDescriptor.Transient<IAuthorizationEvaluator, DefaultAuthorizationEvaluator>());

services.TryAdd(ServiceDescriptor.Transient<IAuthorizationHandlerContextFactory, DefaultAuthorizationHandlerContextFactory>());

services.TryAddEnumerable(ServiceDescriptor.Transient<IAuthorizationHandler, PassThroughAuthorizationHandler>());

return services;

}

public static IServiceCollection AddAuthorization(this IServiceCollection services, Action<AuthorizationOptions> configure)

{

services.Configure(configure);

return services.AddAuthorization();

}

}

首先，是对授权进行配置的AuthorizationOptions，然后在DI系统中注册了几个核心对象的默认实现，我们一一来看。

### AuthorizationOptions

对于**Options**模式，大家应该都比较熟悉了，AuthorizationOptions是添加和获取授权策略的入口点：

public class AuthorizationOptions

{

private IDictionary<string, AuthorizationPolicy> PolicyMap { get; } = new Dictionary<string, AuthorizationPolicy>(StringComparer.OrdinalIgnoreCase);

// 在上一个策略验证失败后，是否继续执行下一个授权策略

public bool InvokeHandlersAfterFailure { get; set; } = true;

public AuthorizationPolicy DefaultPolicy { get; set; } = new AuthorizationPolicyBuilder().RequireAuthenticatedUser().Build();

public void AddPolicy(string name, AuthorizationPolicy policy)

{

PolicyMap[name] = policy;

}

public void AddPolicy(string name, Action<AuthorizationPolicyBuilder> configurePolicy)

{

var policyBuilder = new AuthorizationPolicyBuilder();

configurePolicy(policyBuilder);

AddPolicy(name,policyBuilder.Build());

}

public AuthorizationPolicy GetPolicy(string name)

{

return PolicyMap.ContainsKey(name) ? PolicyMap[name] : null;

}

}

首先是一个PolicyMap字典，我们定义的策略都保存在其中，AddPolicy方法只是简单的将策略添加到该字典中，而其DefaultPolicy属性表示默认策略，初始值为：“已认证用户”。

在AuthorizationOptions中主要涉及到AuthorizationPolicy和AuthorizationPolicyBuilder两个对象。

### AuthorizationPolicy

在 ASP.NET Core 中，授权策略具体表现为一个AuthorizationPolicy对象：

public class AuthorizationPolicy

{

public AuthorizationPolicy(IEnumerable<IAuthorizationRequirement> requirements, IEnumerable<string> authenticationSchemes) {}

public IReadOnlyList<IAuthorizationRequirement> Requirements { get; }

public IReadOnlyList<string> AuthenticationSchemes { get; }

public static AuthorizationPolicy Combine(params AuthorizationPolicy[] policies)

{

return Combine((IEnumerable<AuthorizationPolicy>)policies);

}

public static AuthorizationPolicy Combine(IEnumerable<AuthorizationPolicy> policies)

{

foreach (var policy in policies)

{

builder.Combine(policy);

}

return builder.Build();

}

public static async Task<AuthorizationPolicy> CombineAsync(IAuthorizationPolicyProvider policyProvider, IEnumerable<IAuthorizeData> authorizeData)

{

foreach (var authorizeDatum in authorizeData)

{

any = true;

var useDefaultPolicy = true;

if (!string.IsNullOrWhiteSpace(authorizeDatum.Policy))

{

policyBuilder.Combine(await policyProvider.GetPolicyAsync(authorizeDatum.Policy));

useDefaultPolicy = false;

}

var rolesSplit = authorizeDatum.Roles?.Split(',');

if (rolesSplit != null && rolesSplit.Any())

{

policyBuilder.RequireRole(rolesSplit.Where(r => !string.IsNullOrWhiteSpace(r)).Select(r => r.Trim()));

useDefaultPolicy = false;

}

var authTypesSplit = authorizeDatum.AuthenticationSchemes?.Split(',');

if (authTypesSplit != null && authTypesSplit.Any())

{

foreach (var authType in authTypesSplit)

{

if (!string.IsNullOrWhiteSpace(authType))

{

policyBuilder.AuthenticationSchemes.Add(authType.Trim());

}

}

}

if (useDefaultPolicy)

{

policyBuilder.Combine(await policyProvider.GetDefaultPolicyAsync());

}

}

return any ? policyBuilder.Build() : null;

}

}

如上，Combine方法通过调用AuthorizationPolicyBuilder来完成授权策略的合并，而CombineAsync则是将我们上面介绍的IAuthorizeData转换为授权策略，因此上面说基于角色/Scheme的授权本质上都是基于策略的授权。

对于AuthenticationSchemes属性，我们在前几章介绍认证时经常看到，用来表示我们使用哪个认证**Scheme**来获取用户的Claims，如果指定多个，则会合并它们的Claims，其实现《[下一章](http://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html)》中再来详细介绍。

而Requirements属性则是策略的核心了，每一个**Requirement**都代表一个授权条件，我们就先来了解一下它。

### IAuthorizationRequirement

**Requirement**使用IAuthorizationRequirement接口来表示：

public interface IAuthorizationRequirement

{

}

IAuthorizationRequirement接口中并没有任何成员，在 ASP.NET Core 中内置了一些常用的实现：

* **AssertionRequirement** ：使用最原始的断言形式来声明授权策略。
* **DenyAnonymousAuthorizationRequirement** ：用于表示禁止匿名用户访问的授权策略，并在AuthorizationOptions中将其设置为默认策略。
* **ClaimsAuthorizationRequirement** ：用于表示判断Cliams中是否包含预期的Claims的授权策略。
* **RolesAuthorizationRequirement** ：用于表示使用ClaimsPrincipal.IsInRole来判断是否包含预期的Role的授权策略。
* **NameAuthorizationRequirement**：用于表示使用ClaimsPrincipal.Identities.Name来判断是否包含预期的Name的授权策略。
* **OperationAuthorizationRequirement**：用于表示基于操作的授权策略。

其逻辑也都非常简单，我就不再一一介绍，只展示一下RolesAuthorizationRequirement的代码片段：

public class RolesAuthorizationRequirement : AuthorizationHandler<RolesAuthorizationRequirement>, IAuthorizationRequirement

{

public IEnumerable<string> AllowedRoles { get; }

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, RolesAuthorizationRequirement requirement)

{

...

if (requirement.AllowedRoles.Any(r => context.User.IsInRole(r)))

{

context.Succeed(requirement);

}

return Task.CompletedTask;

}

}

其AllowedRoles表示允许授权通过的角色，而它还实现了IAuthorizationHandler接口，用来完成授权的逻辑。

public interface IAuthorizationHandler

{

Task HandleAsync(AuthorizationHandlerContext context);

}

AuthorizationRequirement并不是一定要实现IAuthorizationHandler接口，后文会详细介绍。

### AuthorizationPolicyBuilder

在上面已经多次用到AuthorizationPolicyBuilder，它提供了一系列创建AuthorizationPolicy的快捷方法：

public class AuthorizationPolicyBuilder

{

public AuthorizationPolicyBuilder(params string[] authenticationSchemes);

public AuthorizationPolicyBuilder(AuthorizationPolicy policy);

public IList<IAuthorizationRequirement> Requirements { get; set; }

public IList<string> AuthenticationSchemes { get; set; }

public AuthorizationPolicyBuilder AddAuthenticationSchemes(params string[] schemes);

public AuthorizationPolicyBuilder AddRequirements(params IAuthorizationRequirement[] requirements);

public AuthorizationPolicyBuilder RequireAssertion(Func<AuthorizationHandlerContext, bool> handler);

public AuthorizationPolicyBuilder RequireAssertion(Func<AuthorizationHandlerContext, Task<bool>> handler)

{

Requirements.Add(new AssertionRequirement(handler));

return this;

}

public AuthorizationPolicyBuilder RequireAuthenticatedUser()

{

Requirements.Add(new DenyAnonymousAuthorizationRequirement());

return this;

}

public AuthorizationPolicyBuilder RequireClaim(string claimType);

public AuthorizationPolicyBuilder RequireClaim(string claimType, params string[] requiredValues);

public AuthorizationPolicyBuilder RequireClaim(string claimType, IEnumerable<string> requiredValues)

{

Requirements.Add(new ClaimsAuthorizationRequirement(claimType, requiredValues));

return this;

}

public AuthorizationPolicyBuilder RequireRole(params string[] roles);

public AuthorizationPolicyBuilder RequireRole(IEnumerable<string> roles)

{

Requirements.Add(new RolesAuthorizationRequirement(roles));

return this;

}

public AuthorizationPolicyBuilder RequireUserName(string userName)

{

Requirements.Add(new NameAuthorizationRequirement(userName));

return this;

}

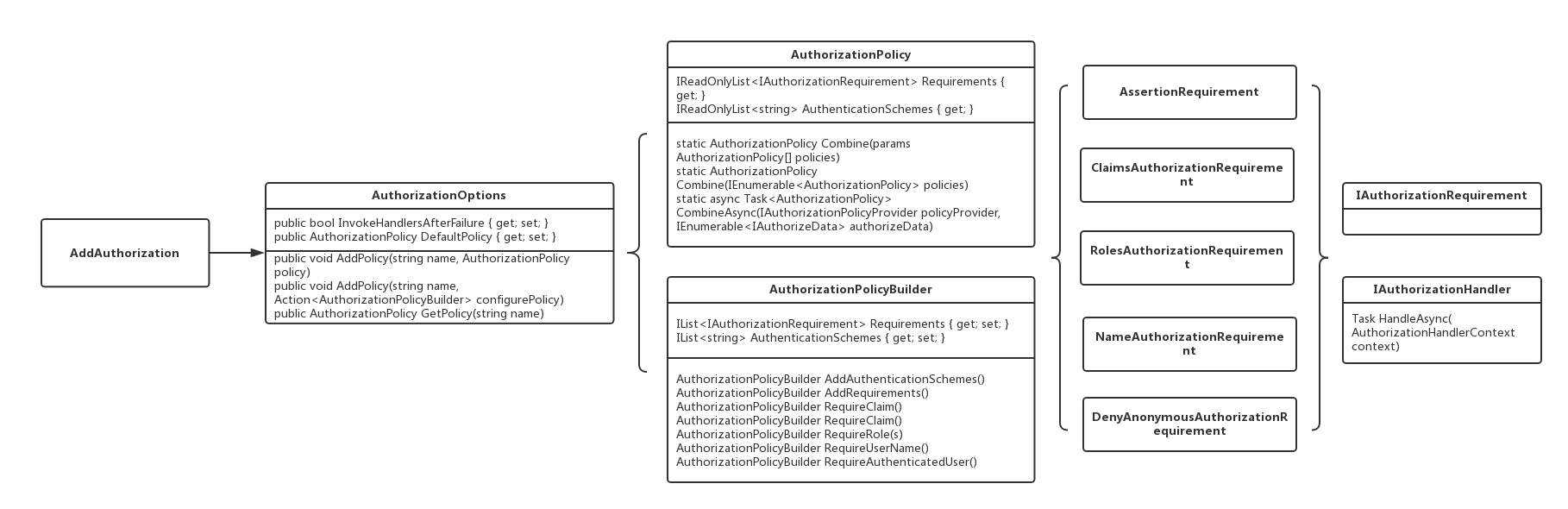
public AuthorizationPolicy Build();

public AuthorizationPolicyBuilder Combine(AuthorizationPolicy policy);

}

在上面介绍的几个**Requirement**，除了OperationAuthorizationRequirement外，都有对应的快捷添加方法，由于OperationAuthorizationRequirement并不属于基于资源的授权，所以不在这里，其用法留在其后续章节再来介绍。

整个授权策略的内容也就这么多，并不复杂，整个结构大致如下：



## 基于策略的授权进阶

在上一小节，我们探索了一下授权策略的源码，现在就来实战一下。

我们使用AuthorizationPolicyBuilder可以很容易的在策略定义中组合我们需要的**Requirement**：

public void ConfigureServices(IServiceCollection services)

{

var commonPolicy = new AuthorizationPolicyBuilder().RequireClaim("MyType").Build();

services.AddAuthorization(options =>

{

options.AddPolicy("User", policy => policy

.RequireAssertion(context => context.User.HasClaim(c => (c.Type == "EmployeeNumber" || c.Type == "Role")))

);

options.AddPolicy("Employee", policy => policy

.RequireRole("Admin")

.RequireUserName("Alice")

.RequireClaim("EmployeeNumber")

.Combine(commonPolicy));

});

}

如上，如果需要，我们还可以定义一个公共的策略对象，然后在策略定义中直接将其合并进来。

### 自定义策略

当内置的**Requirement**不能满足我们的需求时，我们也可以很容易的定义自己的**Requirement**：

public class MinimumAgeRequirement : AuthorizationHandler<NameAuthorizationRequirement>, IAuthorizationRequirement

{

public MinimumAgeRequirement(int minimumAge)

{

MinimumAge = minimumAge;

}

public int MinimumAge { get; private set; }

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, NameAuthorizationRequirement requirement)

{

if (context.User != null && context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth)

{

var dateOfBirth = Convert.ToDateTime(context.User.FindFirst(c => c.Type == ClaimTypes.DateOfBirth).Value);

int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;

if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))

{

calculatedAge--;

}

if (calculatedAge >= requirement.MinimumAge)

{

context.Succeed(requirement);

}

}

return Task.CompletedTask;

}

}

然后就可以直接在AddPolicy中使用了：

services.AddAuthorization(options =>

{

options.AddPolicy("Over21", policy => policy.Requirements.Add(new MinimumAgeRequirement(21)));

});

我们自定义的 Requirement 若想得到 ASP.NET Core 授权系统的执行，除了上面示例中的实现IAuthorizationHandler接口外，也可以单独定义**AuthorizationHandler**，这样可以更好的使用DI系统，并且还可以定义多个Handler，下面就来演示一下。

### 多Handler模式

授权策略中的多个Requirement，它们属于 **&** 的关系，只用全部验证通过，才能最终授权成功。但是在有些场景下，我们可能希望一个授权策略可以适用多种情况，比如，我们进入公司时需要出示员工卡才可以被授权进入，但是如果我们忘了带员工卡，可以去申请一个临时卡，同样可以授权成功:

public class EnterBuildingRequirement : IAuthorizationRequirement

{

}

public class BadgeEntryHandler : AuthorizationHandler<EnterBuildingRequirement>

{

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, EnterBuildingRequirement requirement)

{

if (context.User.HasClaim(c => c.Type == ClaimTypes.BadgeId)

{

context.Succeed(requirement);

}

else

{

// context.Fail();

}

return Task.CompletedTask;

}

}

public class HasTemporaryStickerHandler : AuthorizationHandler<EnterBuildingRequirement>

{

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, EnterBuildingRequirement requirement)

{

if (context.User.HasClaim(c => c.Type == ClaimTypes.TemporaryBadgeId)

{

context.Succeed(requirement);

}

return Task.CompletedTask;

}

}

如上，我们定义了两个Handler，但是想让它们得到执行，还需要将其注册到DI系统中：

services.AddSingleton<IAuthorizationHandler, BadgeEntryHandler>();

services.AddSingleton<IAuthorizationHandler, HasTemporaryStickerHandler>();

此时，在我们的应该程序中使用EnterBuildingRequirement的授权时，将会依次执行这两个Handler。而在上面介绍AuthorizationOptions时，提到它还有一个InvokeHandlersAfterFailure属性，在这里就派上用场了，只有其为true时（默认为True），才会在当前 AuthorizationHandler 授权失败时，继续执行下一个 AuthorizationHandler。

在上面的示例中，我们使用context.Succeed(requirement)将授权结果设置为成功，而失败时并没有做任何标记，正常情况下都是这样做的。但是如果需要，我们可以通过调用context.Fail()方法显式的将授权结果设置为失败，那么，不管其他 AuthorizationHandler 是成功还是失败，最终结果都将是授权失败。

## 总结

ASP.NET Core 授权策略是一种非常强大、灵活的权限验证方案，提供了更丰富、更易表达的验证模型，能够满足大部分的授权场景。通过本文对授权策略的详细介绍，我们应该能够灵活的使用基于策略的授权了，但是授权策略到底是怎么执行的呢？在《[下一章](http://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html)》中，就来完整的探索一下 ASP.NET Core 授权系统的执行流程。

[ASP.NET Core 认证与授权[6]:授权策略是怎么执行的？](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html)

在上一章中，详细介绍了 ASP.NET Core 中的授权策略，在需要授权时，只需要在对应的Controler或者Action上面打上[Authorize]特性，并指定要执行的策略名称即可，但是，授权策略是怎么执行的呢？怀着一颗好奇的心，忍不住来探索一下它的执行流程。

**目录**

1. [MVC中的授权](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#mvc%E4%B8%AD%E7%9A%84%E6%8E%88%E6%9D%83)
   * [AuthorizationApplicationModelProvider](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#authorizationapplicationmodelprovider)
   * [AuthorizeFilter](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#authorizefilter)
2. [IPolicyEvaluator](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#ipolicyevaluator)
   * [AuthenticateAsync(AuthenticationSchemes)](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#authenticateasyncauthenticationschemes)
   * [AuthorizeAsync(Requirements)](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#authorizeasyncrequirements)
3. [IAuthorizationService](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#iauthorizationservice)
   * [IAuthorizationPolicyProvider](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#iauthorizationpolicyprovider)
   * [IAuthorizationHandlerContextFactory](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#iauthorizationhandlercontextfactory)
   * [IAuthorizationHandlerProvider](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#iauthorizationhandlerprovider)
   * [IAuthorizationEvaluator](https://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#iauthorizationevaluator)

在《([上一章](http://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html)》中提到，AuthorizeAttribute只是一个简单的实现了IAuthorizeData接口的特性，并且在 ASP.NET Core 授权系统中并没有使用到它。我们知道在认证中，还有一个UseAuthentication扩展方法来激活认证系统，但是在授权中并没有类似的机制。

这是因为当我们使用[Authorize]通常是在MVC中，由MVC来负责激活授权系统。本来在这个系列的文章中，我并不想涉及到MVC的知识，但是为了能更好的理解授权系统的执行，就来简单介绍一下MVC中与授权相关的知识。

**MVC中的授权**

当我们使用MVC时，首先会调用MVC的AddMvc扩展方法，用来注册一些MVC相关的服务：

public static IMvcBuilder AddMvc(this IServiceCollection services)

{

var builder = services.AddMvcCore();

builder.AddAuthorization();

...

}

public static IMvcCoreBuilder AddAuthorization(this IMvcCoreBuilder builder)

{

AddAuthorizationServices(builder.Services);

return builder;

}

internal static void AddAuthorizationServices(IServiceCollection services)

{

services.AddAuthenticationCore();

services.AddAuthorization();

services.AddAuthorizationPolicyEvaluator();

services.TryAddEnumerable(

ServiceDescriptor.Transient<IApplicationModelProvider, AuthorizationApplicationModelProvider>());

}

在上面AddAuthorizationServices中的前三个方法都属于 ASP.NET Core 《[Security](https://github.com/aspnet/Security)》项目中提供的扩展方法，其中前两个在前面几章已经介绍过了，对于AddAuthorizationPolicyEvaluator放到后面再来介绍，我们先来看一下MVC中的AuthorizationApplicationModelProvider。

**AuthorizationApplicationModelProvider**

在MVC中有一个ApplicationModel的概念，它用来封装Controller, Filter, ApiExplorer等。对应的，在MVC中还提供了一系列的*ApplicationModelProvider*来初始化ApplicationModel的各个部分，而AuthorizationApplicationModelProvider就是用来初始化与授权相关的部分。

public class AuthorizationApplicationModelProvider : IApplicationModelProvider

{

public void OnProvidersExecuting(ApplicationModelProviderContext context)

{

foreach (var controllerModel in context.Result.Controllers)

{

var controllerModelAuthData = controllerModel.Attributes.OfType<IAuthorizeData>().ToArray();

if (controllerModelAuthData.Length > 0)

{

controllerModel.Filters.Add(GetFilter(\_policyProvider, controllerModelAuthData));

}

foreach (var attribute in controllerModel.Attributes.OfType<IAllowAnonymous>())

{

controllerModel.Filters.Add(new AllowAnonymousFilter());

}

foreach (var actionModel in controllerModel.Actions)

{

var actionModelAuthData = actionModel.Attributes.OfType<IAuthorizeData>().ToArray();

if (actionModelAuthData.Length > 0)

{

actionModel.Filters.Add(GetFilter(\_policyProvider, actionModelAuthData));

}

foreach (var attribute in actionModel.Attributes.OfType<IAllowAnonymous>())

{

actionModel.Filters.Add(new AllowAnonymousFilter());

}

}

}

}

}

如上，首先查找每个Controller中实现了IAuthorizeData接口的特性，然后将其转化为AuthorizeFilter并添加到*Controller*的Filter集合中，紧接着再查找实现了IAllowAnonymous接口的特性，将其转化为AllowAnonymousFilter过滤器也添加到Filter集合中，然后以同样的逻辑查找Action上的特性并添加到*Action*的Filter集合中。

其中的关键点就是将IAuthorizeData（也就是通过我们熟悉的[Authorize]特性）转化为MVC中的AuthorizeFilter过滤器：

public static AuthorizeFilter GetFilter(IAuthorizationPolicyProvider policyProvider, IEnumerable<IAuthorizeData> authData)

{

if (policyProvider.GetType() == typeof(DefaultAuthorizationPolicyProvider))

{

var policy = AuthorizationPolicy.CombineAsync(policyProvider, authData).GetAwaiter().GetResult();

return new AuthorizeFilter(policy);

}

else

{

return new AuthorizeFilter(policyProvider, authData);

}

}

CombineAsync在上一章的《[AuthorizationPolicy](http://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#authorizationpolicy)》中已经介绍过了，我们往下看看*AuthorizeFilter*的实现。

**AuthorizeFilter**

在MVC中有一个AuthorizeFilter过滤器，类似我们在ASP.NET 4.x中所熟悉的[Authorize]，它实现了IAsyncAuthorizationFilter接口，定义如下：

public class AuthorizeFilter : IAsyncAuthorizationFilter, IFilterFactory

{

public AuthorizeFilter(AuthorizationPolicy policy) {}

public AuthorizeFilter(IAuthorizationPolicyProvider policyProvider, IEnumerable<IAuthorizeData> authorizeData) : this(authorizeData) {}

public AuthorizeFilter(IEnumerable<IAuthorizeData> authorizeData) {}

public IEnumerable<IAuthorizeData> AuthorizeData { get; }

public AuthorizationPolicy Policy { get; }

public virtual async Task OnAuthorizationAsync(AuthorizationFilterContext context)

{

var effectivePolicy = Policy;

if (effectivePolicy == null)

{

effectivePolicy = await AuthorizationPolicy.CombineAsync(PolicyProvider, AuthorizeData);

}

var policyEvaluator = context.HttpContext.RequestServices.GetRequiredService<IPolicyEvaluator>();

var authenticateResult = await policyEvaluator.AuthenticateAsync(effectivePolicy, context.HttpContext);

if (context.Filters.Any(item => item is IAllowAnonymousFilter))

{

return;

}

var authorizeResult = await policyEvaluator.AuthorizeAsync(effectivePolicy, authenticateResult, context.HttpContext, context);

... // 如果授权失败，返回ChallengeResult或ForbidResult

}

}

AuthorizeFilter的OnAuthorizationAsync方法会在Action执行之前触发，其调用IPolicyEvaluator来完成授权，将执行流程切回到 ASP.NET Core 授权系统中。关于MVC中IApplicationModelProvider以及Filter的概念，在以后MVC系列的文章中再来详细介绍，下面就继续介绍 ASP.NET Core 的授权系统，也就是《[Security](https://github.com/aspnet/Security)》项目。

**IPolicyEvaluator**

IPolicyEvaluator是MVC调用授权系统的入口点，其定义如下：

public interface IPolicyEvaluator

{

Task<AuthenticateResult> AuthenticateAsync(AuthorizationPolicy policy, HttpContext context);

Task<PolicyAuthorizationResult> AuthorizeAsync(AuthorizationPolicy policy, AuthenticateResult authenticationResult, HttpContext context, object resource);

}

在上面介绍的AddMVC中，调用了AddAuthorizationPolicyEvaluator扩展方法，它有如下定义：

public static class PolicyServiceCollectionExtensions

{

public static IServiceCollection AddAuthorizationPolicyEvaluator(this IServiceCollection services)

{

services.TryAdd(ServiceDescriptor.Transient<IPolicyEvaluator, PolicyEvaluator>());

return services;

}

}

由此可知IPolicyEvaluator的默认实现为PolicyEvaluator，我们就从它入手，来一步一步解剖 ASP.NET Core 授权系统的执行步骤。

在AuthorizeFilter中，依次调到了AuthenticateAsync和AuthorizeAsync方法，我们就一一来看。

**AuthenticateAsync(AuthenticationSchemes)**

为什么还有一个AuthenticateAsync方法呢，这不是在认证阶段执行的吗？我们看下它的实现：

public class PolicyEvaluator : IPolicyEvaluator

{

public virtual async Task<AuthenticateResult> AuthenticateAsync(AuthorizationPolicy policy, HttpContext context)

{

if (policy.AuthenticationSchemes != null && policy.AuthenticationSchemes.Count > 0)

{

ClaimsPrincipal newPrincipal = null;

foreach (var scheme in policy.AuthenticationSchemes)

{

var result = await context.AuthenticateAsync(scheme);

if (result != null && result.Succeeded)

{

newPrincipal = SecurityHelper.MergeUserPrincipal(newPrincipal, result.Principal);

}

}

if (newPrincipal != null)

{

context.User = newPrincipal;

return AuthenticateResult.Success(new AuthenticationTicket(newPrincipal, string.Join(";", policy.AuthenticationSchemes)));

}

else

{

context.User = new ClaimsPrincipal(new ClaimsIdentity());

return AuthenticateResult.NoResult();

}

}

return (context.User?.Identity?.IsAuthenticated ?? false)

? AuthenticateResult.Success(new AuthenticationTicket(context.User, "context.User"))

: AuthenticateResult.NoResult();

}

}

在《[上一章](http://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#authorizationpolicy)》中，我们知道在*AuthorizationPolicy*中有**AuthenticationSchemes**和**IAuthorizationRequirement**两个属性，并详细介绍介绍了*Requirement*，但是没有提到*AuthenticationSchemes*的调用。

那么，看到这里，也就大概明白了，它与*Requirements*的执行是完全独立的，并在它之前执行，用于重置**Claims**，那么为什么要重置呢？

在认证的章节介绍过，在认证阶段，只会执行默认的认证Scheme，context.User就是使用context.AuthenticateAsync(DefaultAuthenticateScheme)来赋值的，当我们希望使用非默认的Scheme，或者是想合并多个认证Scheme的Claims时，就需要使用基于Scheme的授权来重置Claims了。

它的实现也很简单，直接使用我们在授权策略中指定的*Schemes*来依次调用认证服务的AuthenticateAsync方法，并将生成的*Claims*合并，最后返回我们熟悉的AuthenticateResult认证结果。

**AuthorizeAsync(Requirements)**

接下来再看一下*PolicyEvaluator*的AuthorizeAsync方法：

public class PolicyEvaluator : IPolicyEvaluator

{

private readonly IAuthorizationService \_authorization;

public PolicyEvaluator(IAuthorizationService authorization)

{

\_authorization = authorization;

}

public virtual async Task<PolicyAuthorizationResult> AuthorizeAsync(AuthorizationPolicy policy, AuthenticateResult authenticationResult, HttpContext context, object resource)

{

var result = await \_authorization.AuthorizeAsync(context.User, resource, policy);

if (result.Succeeded) return PolicyAuthorizationResult.Success();

return (authenticationResult.Succeeded) ? PolicyAuthorizationResult.Forbid() : PolicyAuthorizationResult.Challenge();

}

}

该方法会根据**Requirements**来完成授权，具体的实现是通过调用IAuthorizationService来实现的。

最终返回的是一个PolicyAuthorizationResult对象，并在授权失败时，根据认证结果来返回Forbid(未授权)或Challenge(未登录)。

public class PolicyAuthorizationResult

{

private PolicyAuthorizationResult() { }

public bool Challenged { get; private set; }

public bool Forbidden { get; private set; }

public bool Succeeded { get; private set; }

}

**IAuthorizationService**

然后就到了授权的核心对象AuthorizationService，也可以称为授权的外交官，我们也可以直接在应用代码中调用该对象来实现授权，它有如下定义：

public interface IAuthorizationService

{

Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user, object resource, string policyName);

Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user, object resource, IEnumerable<IAuthorizationRequirement> requirements);

}

在AuthorizeAsync中还涉及到一个resource对象，用来实现面向资源的授权，放在《[下一章](http://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html)》中再来介绍，而在本章与《[前一章](http://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html)》的示例中，该值均为null。

ASP.NET Core 中还为IAuthorizationService提供了几个扩展方法：

public static class AuthorizationServiceExtensions

{

public static Task<AuthorizationResult> AuthorizeAsync(this IAuthorizationService service, ClaimsPrincipal user, string policyName) {}

public static Task<AuthorizationResult> AuthorizeAsync(this IAuthorizationService service, ClaimsPrincipal user, AuthorizationPolicy policy) {}

public static Task<AuthorizationResult> AuthorizeAsync(this IAuthorizationService service, ClaimsPrincipal user, object resource, IAuthorizationRequirement requirement) {}

public static Task<AuthorizationResult> AuthorizeAsync(this IAuthorizationService service, ClaimsPrincipal user, object resource, AuthorizationPolicy policy) {}

}

其默认实现为DefaultAuthorizationService:

public class DefaultAuthorizationService : IAuthorizationService

{

private readonly AuthorizationOptions \_options;

private readonly IAuthorizationHandlerContextFactory \_contextFactory;

private readonly IAuthorizationHandlerProvider \_handlers;

private readonly IAuthorizationEvaluator \_evaluator;

private readonly IAuthorizationPolicyProvider \_policyProvider;

public async Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user, object resource, string policyName)

{

var policy = await \_policyProvider.GetPolicyAsync(policyName);

return await this.AuthorizeAsync(user, resource, policy);

}

public async Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user, object resource, IEnumerable<IAuthorizationRequirement> requirements)

{

var authContext = \_contextFactory.CreateContext(requirements, user, resource);

var handlers = await \_handlers.GetHandlersAsync(authContext);

foreach (var handler in handlers)

{

await handler.HandleAsync(authContext);

if (!\_options.InvokeHandlersAfterFailure && authContext.HasFailed)

{

break;

}

}

return \_evaluator.Evaluate(authContext);

}

}

通过上面代码可以看出，在《[上一章](http://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html)》中介绍的授权策略，在这里获取到它的*Requirements*，后续便不再需要了。而在AuthorizationService中是通过调用四大核心对象来完成授权，我们一一来看。

**IAuthorizationPolicyProvider**

由于在[Authorize]中，我们指定的是策略的名称，因此需要使用IAuthorizationPolicyProvider来根据名称获取到策略对象，默认实现为DefaultAuthorizationPolicyProvider：

public class DefaultAuthorizationPolicyProvider : IAuthorizationPolicyProvider

{

private readonly AuthorizationOptions \_options;

public Task<AuthorizationPolicy> GetDefaultPolicyAsync()

{

return Task.FromResult(\_options.DefaultPolicy);

}

public virtual Task<AuthorizationPolicy> GetPolicyAsync(string policyName)

{

return Task.FromResult(\_options.GetPolicy(policyName));

}

}

在上一章中介绍过，我们定义的策略都保存在《[AuthorizationOptions](http://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#authorizationoptions)》的字典中，因此在这里只是简单的将AuthorizationOptions中的同名方法异步化。

**IAuthorizationHandlerContextFactory**

授权上下文是我们接触较多的对象，当我们自定义授权Handler时就会用到它，它是使用简单工厂模式来创建的：

public class DefaultAuthorizationHandlerContextFactory : IAuthorizationHandlerContextFactory

{

public virtual AuthorizationHandlerContext CreateContext(IEnumerable<IAuthorizationRequirement> requirements, ClaimsPrincipal user, object resource)

{

return new AuthorizationHandlerContext(requirements, user, resource);

}

}

授权上下文中主要包含用户的Claims和授权策略的*Requirements*：

public class AuthorizationHandlerContext

{

private HashSet<IAuthorizationRequirement> \_pendingRequirements;

private bool \_failCalled;

private bool \_succeedCalled;

public AuthorizationHandlerContext(IEnumerable<IAuthorizationRequirement> requirements, ClaimsPrincipal user, object resource)

{

Requirements = requirements; User = user; Resource = resource;

\_pendingRequirements = new HashSet<IAuthorizationRequirement>(requirements);

}

public virtual bool HasFailed { get { return \_failCalled; } }

public virtual bool HasSucceeded => !\_failCalled && \_succeedCalled && !\_pendingRequirements.Any();

public virtual void Fail()

{

\_failCalled = true;

}

public virtual void Succeed(IAuthorizationRequirement requirement)

{

\_succeedCalled = true;

\_pendingRequirements.Remove(requirement);

}

}

如上，\_pendingRequirements中保存着所有待验证的*Requirements*，验证成功的*Requirement*则从中移除。

**IAuthorizationHandlerProvider**

兜兜转转，终于进入到了授权的最终验证逻辑中了，首先，使用IAuthorizationHandlerProvider来获取到所有的授权**Handler**。

IAuthorizationHandlerProvider的默认实现为DefaultAuthorizationHandlerProvider:

public class DefaultAuthorizationHandlerProvider : IAuthorizationHandlerProvider

{

private readonly IEnumerable<IAuthorizationHandler> \_handlers;

public DefaultAuthorizationHandlerProvider(IEnumerable<IAuthorizationHandler> handlers)

{

\_handlers = handlers;

}

public Task<IEnumerable<IAuthorizationHandler>> GetHandlersAsync(AuthorizationHandlerContext context)

=> Task.FromResult(\_handlers);

}

在《[上一章](http://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#%E8%87%AA%E5%AE%9A%E4%B9%89%E7%AD%96%E7%95%A5)》中，我们还介绍到，我们定义的*Requirement*，可以直接实现IAuthorizationHandler接口，也可以单独定义Handler，但是需要注册到DI系统中去。

在默认的*AuthorizationHandlerProvider*中，会从DI系统中获取到我们注册的所有Handler，最终调用其HandleAsync方法。

我们在实现IAuthorizationHandler接口时，通常是继承自AuthorizationHandler<TRequirement>来实现，它有如下定义：

public abstract class AuthorizationHandler<TRequirement> : IAuthorizationHandler where TRequirement : IAuthorizationRequirement

{

public virtual async Task HandleAsync(AuthorizationHandlerContext context)

{

foreach (var req in context.Requirements.OfType<TRequirement>())

{

await HandleRequirementAsync(context, req);

}

}

protected abstract Task HandleRequirementAsync(AuthorizationHandlerContext context, TRequirement requirement);

}

如上，首先会在HandleAsync过滤出与*Requirement*对匹配的*Handler*，然后再调用其HandleRequirementAsync方法。

那我们定义的直接实现IAuthorizationHandler了接口的*Requirement*又是如何执行的呢？

在AddAuthorization扩展方法中可以看到，默认还为IAuthorizationHandler注册了一个PassThroughAuthorizationHandler，定义如下：

public class PassThroughAuthorizationHandler : IAuthorizationHandler

{

public async Task HandleAsync(AuthorizationHandlerContext context)

{

foreach (var handler in context.Requirements.OfType<IAuthorizationHandler>())

{

await handler.HandleAsync(context);

}

}

}

它负责调用该策略中所有实现了IAuthorizationHandler接口的*Requirement*。

**IAuthorizationEvaluator**

最后，通过调用IAuthorizationEvaluator接口，来完成最终的授权结果，默认实现为DefaultAuthorizationEvaluator:

public class DefaultAuthorizationEvaluator : IAuthorizationEvaluator

{

public AuthorizationResult Evaluate(AuthorizationHandlerContext context)

=> context.HasSucceeded

? AuthorizationResult.Success()

: AuthorizationResult.Failed(context.HasFailed

? AuthorizationFailure.ExplicitFail()

: AuthorizationFailure.Failed(context.PendingRequirements));

}

当我们在一个策略中指定多个*Requirement*时，只有全部验证通过时，授权上下文中的HasSucceeded才会为True，而HasFailed代表授权结果的显式失败。

这里根据授权上下文的验证结果来生成授权结果：

public class AuthorizationResult

{

public bool Succeeded { get; private set; }

public AuthorizationFailure Failure { get; private set; }

public static AuthorizationResult Success() => new AuthorizationResult { Succeeded = true };

public static AuthorizationResult Failed(AuthorizationFailure failure) => new AuthorizationResult { Failure = failure };

public static AuthorizationResult Failed() => new AuthorizationResult { Failure = AuthorizationFailure.ExplicitFail() };

}

public class AuthorizationFailure

{

private AuthorizationFailure() { }

public bool FailCalled { get; private set; }

public IEnumerable<IAuthorizationRequirement> FailedRequirements { get; private set; }

public static AuthorizationFailure ExplicitFail()

{

return new AuthorizationFailure { FailCalled = true, FailedRequirements = new IAuthorizationRequirement[0] };

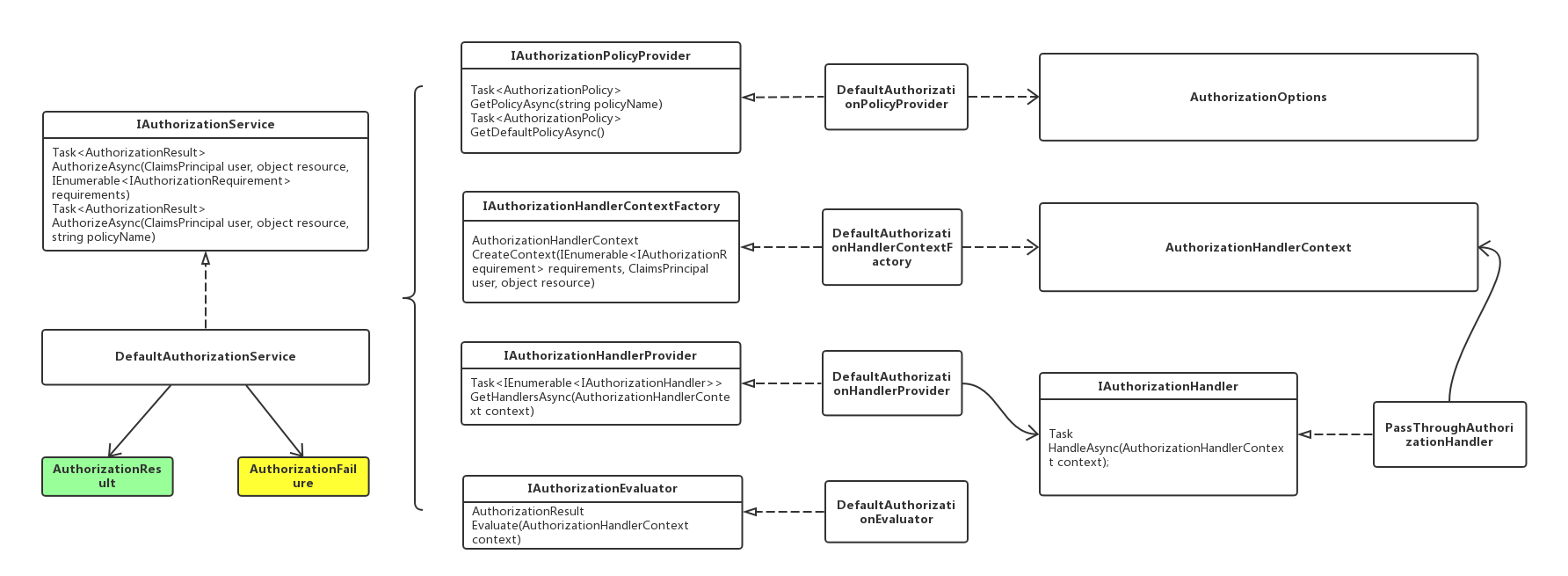
}

public static AuthorizationFailure Failed(IEnumerable<IAuthorizationRequirement> failed)

=> new AuthorizationFailure { FailedRequirements = failed };

}

整个授权流程的结构大致如下：



**总结**

通过对 ASP.NET Core 授权系统执行流程的探索，可以看出授权是主要是通过调用IAuthorizationService来完成的，而授权策略的本质是提供 **Requirement** ，我们完全可以使用它们两个来完成各种灵活的授权方式，而不用局限于策略。在 ASP.NET Core 中，还提供了基于资源的授权，放在《[下一章](http://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html)》中来介绍，并会简单演示一下在一个通用权限管理系统中如何来授权。

[ASP.NET Core 认证与授权[7]:动态授权](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html)

ASP.NET Core 中基于策略的授权旨在分离授权与应用程序逻辑，它提供了灵活的策略定义模型，在一些权限固定的系统中，使用起来非常方便。但是，当要授权的资源无法预先确定，或需要将权限控制到每一个具体的操作当中时，基于策略的授权便不再适用，本章就来介绍一下如何进行动态的授权。

**目录**

1. [基于资源的授权](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E5%9F%BA%E4%BA%8E%E8%B5%84%E6%BA%90%E7%9A%84%E6%8E%88%E6%9D%83)
   * [定义资源Requirement](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E5%AE%9A%E4%B9%89%E8%B5%84%E6%BA%90requirement)
   * [实现资源授权Handler](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E5%AE%9E%E7%8E%B0%E8%B5%84%E6%BA%90%E6%8E%88%E6%9D%83handler)
   * [调用AuthorizationService](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E8%B0%83%E7%94%A8authorizationservice)
2. [基于权限的授权](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E5%9F%BA%E4%BA%8E%E6%9D%83%E9%99%90%E7%9A%84%E6%8E%88%E6%9D%83)
   * [定义权限项](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E5%AE%9A%E4%B9%89%E6%9D%83%E9%99%90%E9%A1%B9)
   * [定义权限Requirement](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E5%AE%9A%E4%B9%89%E6%9D%83%E9%99%90requirement)
   * [实现权限授权Handler](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E5%AE%9E%E7%8E%B0%E6%9D%83%E9%99%90%E6%8E%88%E6%9D%83handler)
   * [使用策略授权](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E4%BD%BF%E7%94%A8%E7%AD%96%E7%95%A5%E6%8E%88%E6%9D%83)
   * [自定义授权过滤器](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E8%87%AA%E5%AE%9A%E4%B9%89%E6%8E%88%E6%9D%83%E8%BF%87%E6%BB%A4%E5%99%A8)
   * [在视图中使用授权](https://www.cnblogs.com/RainingNight/p/dynamic-authorization-in-asp-net-core.html#%E5%9C%A8%E8%A7%86%E5%9B%BE%E4%B8%AD%E4%BD%BF%E7%94%A8%E6%8E%88%E6%9D%83)

**基于资源的授权**

有些场景下，授权需要依赖于要访问的资源，例如：每个资源通常会有一个创建者属性，我们只允许该资源的创建者才可以对其进行编辑，删除等操作，这就无法通过[Authorize]特性来指定授权了。因为授权过滤器会在我们的应用代码，以及MVC的模型绑定之前执行，无法确定所访问的资源。此时，我们需要使用基于资源的授权，下面就来演示一下具体是如何操作的。

**定义资源Requirement**

在基于资源的授权中，我们要判断的是用户是否具有针对该资源的某项操作，因此，我们先定义一个代表操作的**Requirement**：

public class MyRequirement : IAuthorizationRequirement

{

public string Name { get; set; }

}

可以根据实际场景来定义需要的属性，在本示例中，只需要一个Name属性，用来表示针对资源的操作名称（如：增查改删等）。

然后，我们预定义一些常用的操作，方便业务中的调用：

public static class Operations

{

public static MyRequirement Create = new MyRequirement { Name = "Create" };

public static MyRequirement Read = new MyRequirement { Name = "Read" };

public static MyRequirement Update = new MyRequirement { Name = "Update" };

public static MyRequirement Delete = new MyRequirement { Name = "Delete" };

}

上面定义的 *MyRequirement* 虽然很简单，但是非常通用，因此，在 ASP.NET Core 中也内置了一个OperationAuthorizationRequirement：

public class OperationAuthorizationRequirement : IAuthorizationRequirement

{

public string Name { get; set; }

}

在实际应用中，我们可以直接使用OperationAuthorizationRequirement，而不需要再自定义 *Requirement*，而在这里只是为了方便理解，后续也继续使用 *MyRequirement* 来演示。

**实现资源授权Handler**

每一个 *Requirement* 都需要有一个对应的 *Handler*，来完成授权逻辑，可以直接让 *Requirement* 实现IAuthorizationHandler接口，也可以单独定义授权Handler，在这里使用后者。

在本示例中，我们是根据资源的创建者来判断用户是否具有操作权限，因此，我们定义一个资源创建者的接口，而不是直接依赖于具体的资源：

public interface IDocument

{

string Creator { get; set; }

}

然后实现我们的授权Handler:

public class DocumentAuthorizationHandler : AuthorizationHandler<OperationAuthorizationRequirement, IDocument>

{

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, OperationAuthorizationRequirement requirement, IDocument resource)

{

// 如果是Admin角色就直接授权成功

if (context.User.IsInRole("admin"))

{

context.Succeed(requirement);

}

else

{

// 允许任何人创建或读取资源

if (requirement == Operations.Create || requirement == Operations.Read)

{

context.Succeed(requirement);

}

else

{

// 只有资源的创建者才可以修改和删除

if (context.User.Identity.Name == resource.Creator)

{

context.Succeed(requirement);

}

else

{

context.Fail();

}

}

}

return Task.CompletedTask;

}

}

在前面章节的《[自定义策略](http://www.cnblogs.com/RainingNight/p/authorization-in-asp-net-core.html#%E8%87%AA%E5%AE%9A%E4%B9%89%E7%AD%96%E7%95%A5)》示例中，我们继承的是AuthorizationHandler<NameAuthorizationRequirement>，而这里继承了AuthorizationHandler<OperationAuthorizationRequirement, Document>，很明显，比之前的多了resource参数，以便用来实现基于资源的授权。

如上，我们并没有验证用户是否已登录，以及context.User是否为空等。这是因为在 ASP.NET Core 的默认授权中，已经对这些进行了判断，我们只需要在要授权的控制器上添加[Authorize]特性即可，无需重复性的工作。

最后，不要忘了，还需要将DocumentAuthorizationHandler注册到DI系统中：

services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationHandler>();

**调用AuthorizationService**

现在就可以在我们的应用代码中调用IAuthorizationService来完成授权了，不过在此之前，我们再来回顾一下IAuthorizationService接口：

public interface IAuthorizationService

{

Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user, object resource, IEnumerable<IAuthorizationRequirement> requirements);

Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user, object resource, string policyName);

}

在《[上一章](http://www.cnblogs.com/RainingNight/p/authorization-hot-to-work-in-asp-net-core.html#iauthorizationservice)》中，我们提到，使用[Authorize]设置授权时，其AuthorizationHandlerContext中的resource字段被设置为空，现在，我们将要授权的资源传进去即可：

[Authorize]

public class DocumentsController : Controller

{

public async Task<ActionResult> Details(int? id)

{

var document = \_docStore.Find(id.Value);

if (document == null)

{

return NotFound();

}

if ((await \_authorizationService.AuthorizeAsync(User, document, Operations.Read)).Succeeded)

{

return View(document);

}

else

{

return new ForbidResult();

}

}

public async Task<IActionResult> Edit(int? id)

{

var document = \_docStore.Find(id.Value);

if (document == null)

{

return NotFound();

}

if ((await \_authorizationService.AuthorizeAsync(User, document, Operations.Update)).Succeeded)

{

return View(document);

}

else

{

return new ForbidReuslt();

}

}

}

如上，在授权失败时，我们返回了ForbidResult，建议不要返回ChallengeResult，因为我们要明确的告诉用户是无权访问，而不是未登录。

基于资源的权限非常简单，但是每次都要在应用代码中显示调用IAuthorizationService，显然比较繁琐，我们也可以使用AOP模式，或者使用EF Core拦截器来实现，将授权验证与业务代码分离。

**基于权限的授权**

在一个通用的用户权限管理系统中，通常每一个Action都代表一种权限，用户拥有哪些权限也是可以动态分配的。本小节就来介绍一下在 ASP.NET Core 中，如何实现一个简单权限管理系统。

**定义权限项**

首先，我们要确定我们的系统分为哪些权限项，这通常是由业务所决定的，并且是预先确定的，我们可以硬编码在代码中，方便统一调用：

public static class Permissions

{

public const string User = "User";

public const string UserCreate = "User.Create";

public const string UserRead = "User.Read";

public const string UserUpdate = "User.Update";

public const string UserDelete = "User.Delete";

}

如上，我们简单定义了“创建用户”，“查询用户”，“更新用户”，“删除用户”四个权限。通常会对权限项进行分组，构成一个树形结构，这样在展示和配置权限时，都会方便很多。在这里，使用.来表示层级进行分组，其中User权限项包含所有以User.开头的权限。

**定义权限Requirement**

与基于资源的授权类似，我们同样需要定义一个权限*Requirement*：

public class PermissionAuthorizationRequirement : IAuthorizationRequirement

{

public PermissionAuthorizationRequirement(string name)

{

Name = name;

}

public string Name { get; set; }

}

使用Name属性来表示权限的名称，与上面Permissions的常量对应。

**实现权限授权Handler**

然后实现与上面定义的 *Requirement* 对应的授权Handler：

public class PermissionAuthorizationHandler : AuthorizationHandler<PermissionAuthorizationRequirement>

{

private readonly UserStore \_userStore;

public PermissionAuthorizationHandler(UserStore userStore)

{

\_userStore = userStore;

}

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, PermissionAuthorizationRequirement requirement)

{

if (context.User != null)

{

if (context.User.IsInRole("admin"))

{

context.Succeed(requirement);

}

else

{

var userIdClaim = context.User.FindFirst(\_ => \_.Type == ClaimTypes.NameIdentifier);

if (userIdClaim != null)

{

if (\_userStore.CheckPermission(int.Parse(userIdClaim.Value), requirement.Name))

{

context.Succeed(requirement);

}

}

}

}

return Task.CompletedTask;

}

}

如上，把admin角色设置为内部固定角色，直接跳过授权检查。其他角色则从Claims中取出用户Id，然后调用CheckPermission完成授权。

权限检查的具体逻辑就属于业务层面的了，通常会从数据库中查找用的的权限列表进行验证，这里就不在多说，简单模拟了一下：

public class UserStore

{

private static List<User> \_users = new List<User>() {

new User { Id=1, Name="admin", Password="111111", Role="admin", Email="admin@gmail.com", PhoneNumber="18800000000"},

new User { Id=2, Name="alice", Password="111111", Role="user", Email="alice@gmail.com", PhoneNumber="18800000001", Permissions = new List<UserPermission> {

new UserPermission { UserId = 1, PermissionName = Permissions.User },

new UserPermission { UserId = 1, PermissionName = Permissions.Role }

}

},

new User { Id=3, Name="bob", Password="111111", Role = "user", Email="bob@gmail.com", PhoneNumber="18800000002", Permissions = new List<UserPermission> {

new UserPermission { UserId = 2, PermissionName = Permissions.UserRead },

new UserPermission { UserId = 2, PermissionName = Permissions.RoleRead }

}

},

};

public bool CheckPermission(int userId, string permissionName)

{

var user = Find(userId);

if (user == null) return false;

return user.Permissions.Any(p => permissionName.StartsWith(p.PermissionName));

}

}

最后，与上面示例一样，将Handler注册到DI系统中：

services.AddSingleton<IAuthorizationHandler, PermissionAuthorizationHandler>();

**使用策略授权**

那么，怎么在应用代码中使用基于权限的授权呢？

最为简单的，我们可以直接借助于 ASP.NET Core 的授权策略来实现基于权限的授权，因为此时并不需要资源。

services.AddAuthorization(options =>

{

options.AddPolicy(Permissions.UserCreate, policy => policy.AddRequirements(new PermissionAuthorizationRequirement(Permissions.UserCreate)));

options.AddPolicy(Permissions.UserRead, policy => policy.AddRequirements(new PermissionAuthorizationRequirement(Permissions.UserRead)));

options.AddPolicy(Permissions.UserUpdate, policy => policy.AddRequirements(new PermissionAuthorizationRequirement(Permissions.UserUpdate)));

options.AddPolicy(Permissions.UserDelete, policy => policy.AddRequirements(new PermissionAuthorizationRequirement(Permissions.UserDelete)));

});

如上，针对每一个权限项都定义一个对应的授权策略，然后，就可以在控制器中直接使用[Authorize]来完成授权：

[Authorize]

public class UserController : Controller

{

[Authorize(Policy = Permissions.UserRead)]

public ActionResult Index()

{

}

[Authorize(Policy = Permissions.UserRead)]

public ActionResult Details(int? id)

{

}

[Authorize(Policy = Permissions.UserCreate)]

public ActionResult Create()

{

return View();

}

[Authorize(Policy = Permissions.UserCreate)]

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Create([Bind("Title")] User user)

{

}

}

当然，我们也可以像基于资源的授权那样，在应用代码中调用IAuthorizationService完成授权，这样做的好处是无需定义策略，但是，显然一个一个来定义策略太过于繁琐。

还有一种更好方式，就是使用MVC过滤器来完成对IAuthorizationService的调用，下面就来演示一下。

**自定义授权过滤器**

我们可以参考上一章中介绍的《[AuthorizeFilter](http://www.cnblogs.com/RainingNight/p/authorize-how-to-work-in-asp-net-core.html#authorizefilter)》来自定义一个权限过滤器：

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

public class PermissionFilter : Attribute, IAsyncAuthorizationFilter

{

public PermissionFilter(string name)

{

Name = name;

}

public string Name { get; set; }

public async Task OnAuthorizationAsync(AuthorizationFilterContext context)

{

var authorizationService = context.HttpContext.RequestServices.GetRequiredService<IAuthorizationService>();

var authorizationResult = await authorizationService.AuthorizeAsync(context.HttpContext.User, null, new PermissionAuthorizationRequirement(Name));

if (!authorizationResult.Succeeded)

{

context.Result = new ForbidResult();

}

}

}

上面的实现非常简单，我们接受一个name参数，代表权限的名称，然后将权限名称转化为PermissionAuthorizationRequirement，最后直接调用 *authorizationService* 来完成授权。

接下来，我们就可以直接在控制器中使用PermissionFilter过滤器来完成基于权限的授权了：

[Authorize]

public class UserController : Controller

{

[PermissionFilter(Permissions.UserRead)]

public ActionResult Index()

{

return View(\_userStore.GetAll());

}

[PermissionFilter(Permissions.UserCreate)]

public ActionResult Create()

{

}

[PermissionFilter(Permissions.UserCreate)]

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Create([Bind("Title")] User user)

{

}

[PermissionFilter(Permissions.UserUpdate)]

public IActionResult Edit(int? id)

{

}

[PermissionFilter(Permissions.UserUpdate)]

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Edit(int id, [Bind("Id,Title")] User user)

{

}

}

**在视图中使用授权**

通常，在前端页面当中，我们也需要根据用户的权限来判断是否显示“添加”，“删除”等按钮，而不是让用户点击“添加”，再提示用户没有权限，这在 ASP.NET Core 中实现起来也非常简单。

我们可以直接在Razor视图中注入IAuthorizationService来检查用户权限：

@inject IAuthorizationService AuthorizationService

@if ((await AuthorizationService.AuthorizeAsync(User, AuthorizationSample.Authorization.Permissions.UserCreate)).Succeeded)

{

<p>

<a asp-action="Create">创建</a>

</p>

}

不过，上面的代码是通过策略名称来授权的，如果我们使用了上面创建的授权过滤器，而没有定义授权策略的话，需要使用如下方式来实现：

@inject IAuthorizationService AuthorizationService

@if ((await AuthorizationService.AuthorizeAsync(User, new PermissionAuthorizationRequirement(AuthorizationSample.Authorization.Permissions.UserCreate))).Succeeded)

{

<p>

<a asp-action="Create">创建</a>

</p>

}

我们也可以定义一个AuthorizationService的扩展方法，实现通过权限名称进行授权，这里就不再多说。

我们不能因为隐藏了操作按钮，就不在后端进行授权验证了，就像JS的验证一样，前端的验证就为了提升用户的体验，后端的验证在任何时候都是必不可少的。

**总结**

在大多数场景下，我们只需要使用授权策略就可以应对，而在授权策略不能满足我们的需求时，由于 ASP.NET Core 提供了一个统一的 IAuthorizationService 授权接口，这就使我们扩展起来也非常方便。ASP.NET Core 的授权部分到这来也就介绍完了，总的来说，要比ASP.NET 4.x的时候，简单，灵活很多，可见 ASP.NET Core 不仅仅是为了跨平台，而是为了适应现代应用程序的开发方式而做出的全新的设计，我们也应该用全新的思维去学习.NET Core，踏上时代的浪潮。