## **ASP.NET Core 中的用户身份**

### **Claim**

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

{

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**

ClaimsIdentity表示用户身份

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"));

最后我们还需要再将用户身份放到ClaimsPrincipal对象中。

### **ClaimsPrincipal**

ClaimsPrincipal代表一个用户

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)

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

var properties = new AuthenticationProperties();

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

// 加密 序列化

var token = Protect(ticket);

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。

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。

## **认证的使用**

下面就来介绍一下 ASP.NET Core 中的身份认证。

### **注册认证服务**

首先，在DI中注册服务认证所需的服务：

public void ConfigureServices(IServiceCollection services)

{

    services.AddAuthentication(options =>

    {

        // 指定认证动作使用的 Scheme，注：Scheme 是认证处理器的标识

        // 指定认证使用的 Scheme 为 Cookie认证 的 Scheme

        options.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;

        // 指定登录使用的 Scheme

        options.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;

        // 指定未认证使用的 Scheme

        options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;

    })

    // 注册 Cookie 认证处理器（AuthenticationHandler）

    .AddCookie()

    // 注册 OpenIdConnect 认证处理器

    .AddOpenIdConnect(o =>

    {

        o.ClientId = "server.hybrid";

        o.ClientSecret = "secret";

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

        o.ResponseType = OpenIdConnectResponseType.CodeIdToken;

    });

}

**添加认证处理中间件**

public void Configure(IApplicationBuilder app){

    app.UseAuthentication();

}

### UseAuthentication实际添加了AuthenticationMiddleware中间件，其源码如下：

public class AuthenticationMiddleware

{

    public async Task Invoke(HttpContext context)

    {

        ...

        // 获取认证处理器提供者

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

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

        {

            // 获取远程认证回调处理器

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

            // 如果是远程认证回调请求，则直接返回

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

            {

                return;

            }

        }

        // 获取默认 Scheme

        var defaultAuthenticate = await Schemes.GetDefaultAuthenticateSchemeAsync();

        if (defaultAuthenticate != null)

        {

            // 使用默认 Scheme 对应的认证处理器处理认证

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

            if (result?.Principal != null)

            {

                // 设置用户对象

                context.User = result.Principal;

            }

        }

        await \_next(context);

    }

}

## **认证处理器**

### **AuthenticationHandler**

AuthenticationHandler是所有认证Handler的抽象基类

AuthenticationHandler提供了IAuthenticationHandler的默认实现，其抽象方法HandleAuthenticateAsync

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

{

    // 处理认证

    protected abstract Task<AuthenticateResult> HandleAuthenticateAsync();

    ...

}

### **RemoteAuthenticationHandler**

RemoteAuthenticationHandler 便是所有远程认证的抽象基类了

RemoteAuthenticationHandler实现了IAuthenticationRequestHandler，其定义了一个抽象方法

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

where TOptions : RemoteAuthenticationOptions, new ()

{

    // 处理远程认证

    // 如：利用浏览器发送过来的令牌从认证服务器获取用户信息，生成结果 HandleRequestResult

    protected abstract Task<HandleRequestResult> HandleRemoteAuthenticateAsync ();

    protected override async Task<AuthenticateResult> HandleAuthenticateAsync () {

        var result = await Context.AuthenticateAsync (SignInScheme);

        ...

    }

}

**UML关系图看起来如下**

