

ASP.NET core 安全基础

北京理工大学计算机学院
金旭亮

Web安全概述

Web安全的两个核心问题

Authentication (认证)

- 用户是谁?

Authorization (授权)

- 用户能干什么?
- 用户能访问哪些资源?

Web的根基——HTTP的安全性问题

通信使用明文，内容可能会被窃听

应对

加密（信道加密，
内容加密）

不验证通信方的身份，因此有可能遭遇
伪装

应对

使用证明身份的证书
（客户端或服务端）

无法证明数据报文的完整性，所以有可
能已经遭到篡改

应对

MD5、SHA-1散列值
检验，数字签名

HTTPS = HTTP over TLS = HTTP + 加密 + 认证 + 完整性保护

与安全相关的HTTP的状态码和首部



HTTP协议规定**401 (Unauthroized)** 为未授权状态码，另有一个**WWW-Authenticate**首部 (Header) 提供身份认证相关信息。



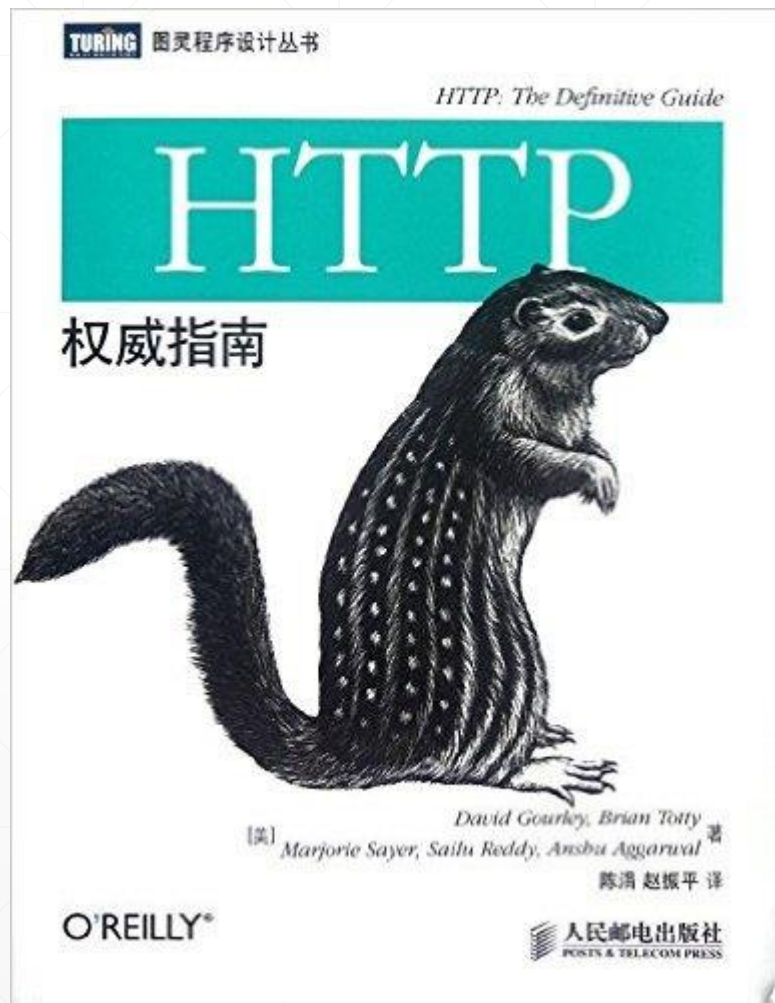
简单地说，如果Web Server发现客户端发来的访问某受保护资源的HTTP请求没有提供有访问权的有效身份凭据时，向客户端返回一个**401**响应，同时搭配一个**WWW-Authenticate**首部，提供一些附加的认证信息。

示例：

HTTP Digest Authentication认证方式的请求与响应：

```
GET /account HTTP/1.1  
Host: example.com  
...
```

```
HTTP/1.1 401 Unauthorized  
WWW-Authenticate: Digest realm="example company",  
nonce="dcd8...c083",  
opaque="5cc...0e41  
...
```

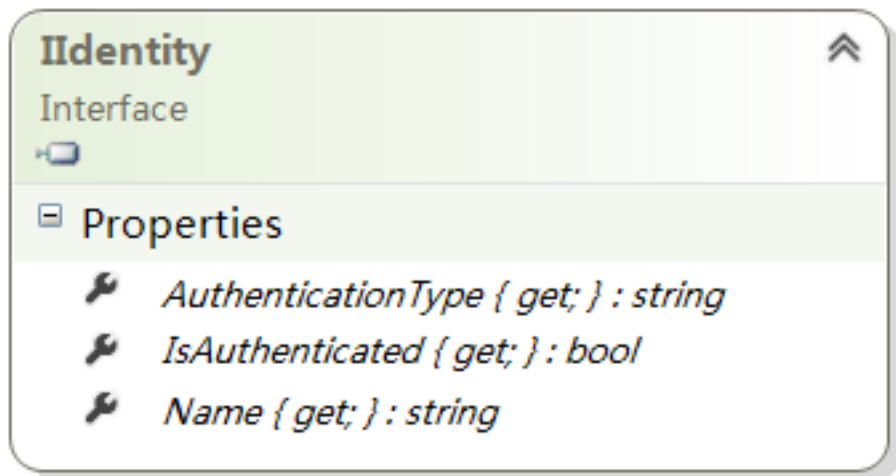


在这本HTTP的经典书籍中，有对HTTP 1.X协议安全机制的详细描述，可供参考。

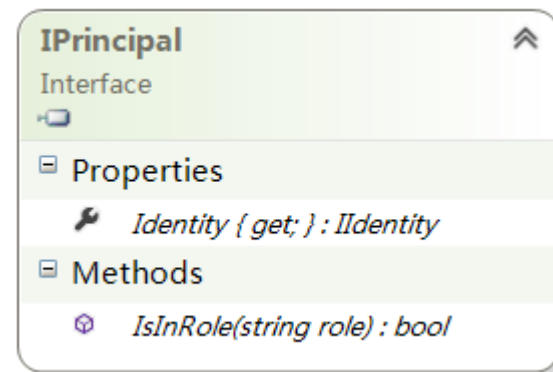
Principal和Identity

示例：PrincipalAndIdentity

两个最基本的安全接口

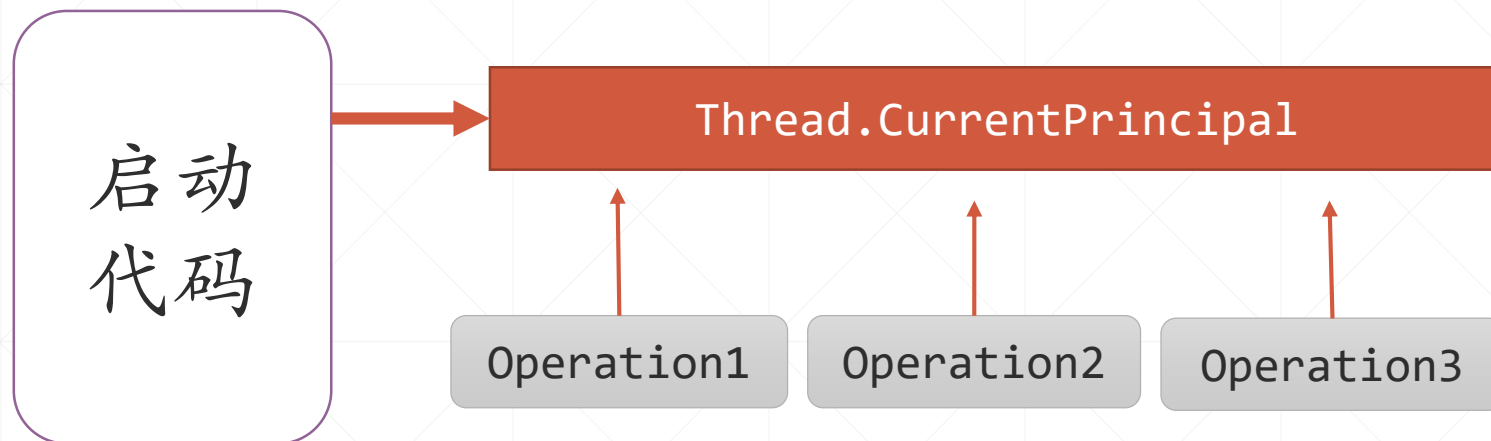


最早的时候，我们只需知道用户的名字，他是否已经通过认证，是通过哪种方式认证的，因此，这个接口就够了。



当引入“角色”这个概念后，**IIdentity**接口就不够用了，因为它没有包容角色信息，因此，.NET又扩展了一个**IPrincipal**接口，此接口定义了方法可以判断某用户是否属于某角色。

实现代码级别的安全特性



.NET中，使用Thread对象的**CurrentPrincipal**属性保存Principal信息。

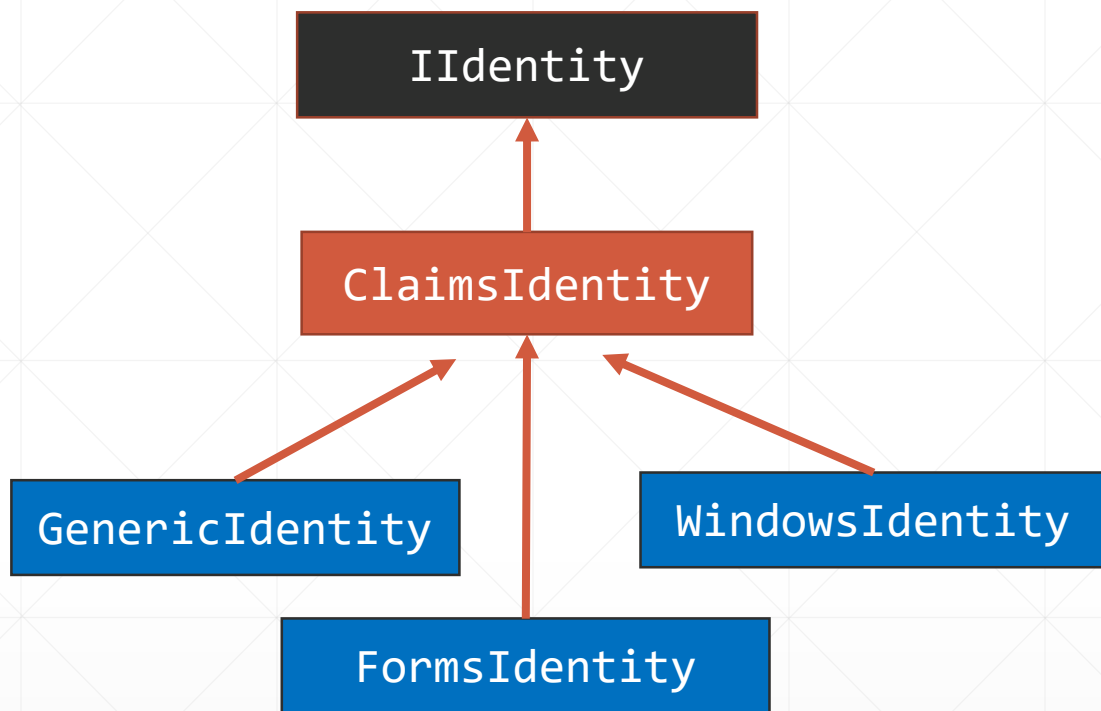


通常是在启动阶段设置好Principal，这样，当程序执行时，特定的方法就可以从**Thread.CurrentPrincipal**中提取出相关信息，以确定这个用户是否有权执行特定的代码。

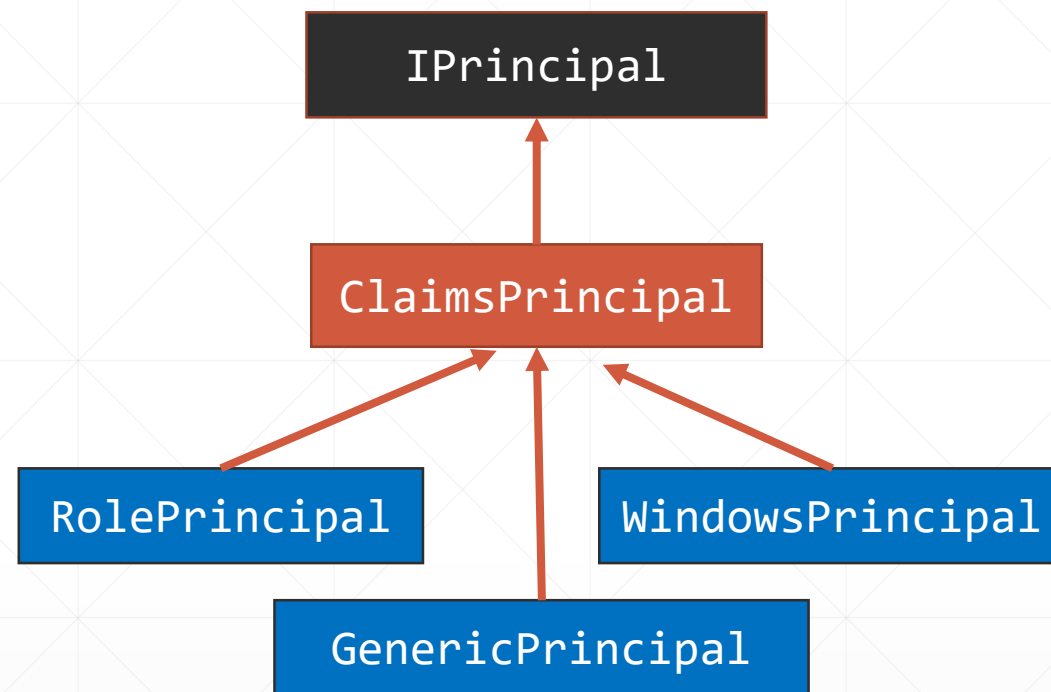
代表一个普通的用户身份
(想想人的身份证)

代表一个普通的用户 (想
想一个拥有身份证的人)

.NET 4.5 以后，基类库中的类型继承树如下所示：



每一个子类，都扩充了新的功能



实际开发中，真正起作用的是 `Principal`，系统基于它完成“授权（Authorize）”。

```
var id = new GenericIdentity("JinXuLiang");  
//JinXuLiang拥有两个角色  
var roles = new string[] { "Student", "Teacher" };  
//给用户以合适的身份，以便授权访问特定的资源  
var teacher = new GenericPrincipal(id, roles);  
//设定线程所使用的用户身份  
Thread.CurrentPrincipal = teacher;
```

使用IPrincipal接口所定义的**IsInRole()**方法，我们可以实现“基于角色”的权限控制。

```
public static void InputScore()  
{  
    new PrincipalPermission(null, "Teacher").Demand();  
    Console.WriteLine("只有教师才能输入考试成绩");  
}
```

没有权限的用户尝试执行代码时.....

An unhandled exception of type
'**System.Security.SecurityException**' occurred in mscorlib.dll

Additional information: 对主体权限的请求失败。

也可以直接使用Code Attribute设定代码权限:

```
[PrincipalPermission(SecurityAction.Demand, Role = "Teacher")]
```

0 references

```
public static void ModifyScore()  
{  
    Console.WriteLine("只有教师才能修改考试成绩");  
}
```

Claims

示例：UseClaim

什么是Claim ?

Claim是一个“命题 (Statement)”，或者称为一个“断言”，其结果只能是true或false。

- 张三是一个学生
- 李四是系统管理员
- 王五的电子邮件是wangwu@example.com
- 赵六只能读这个文档，不能修改它
- 牛七的密码是12345678



Claim实际上是代表了与用户关联的一些信息，就像身份证上所提供的“姓名”，“住址”等信息

Claim
Class

Properties

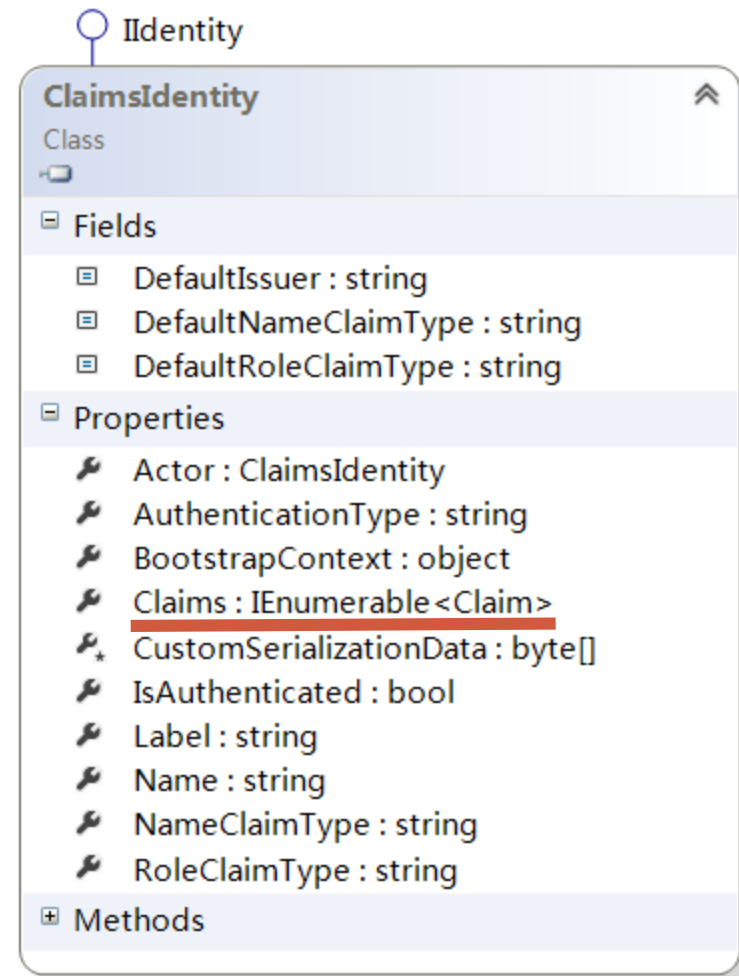
- CustomSerializationData : byte[]
- Issuer : string
- OriginalIssuer : string
- Properties : IDictionary<string, string>
- Subject : ClaimsIdentity
- Type : string
- Value : string
- ValueType : string

Methods

- Claim() (+ 8 overloads)
- Clone() : Claim (+ 1 overload)
- ToString() : string
- WriteTo() : void (+ 1 overload)

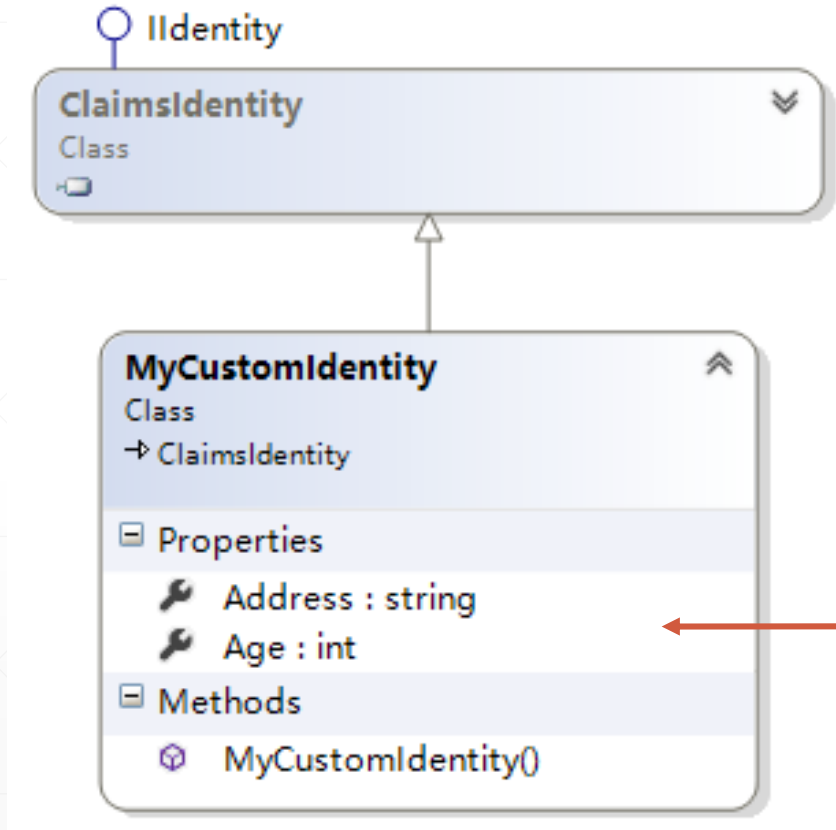
ClaimsIdentity

Identity代表某种“身份”，特定的Identity可以声明多个“claim”，表明“我是XXX”，我“能干XXX”之类的信息，软件系统就可以依据这些信息，确定拥有此Identity的用户是否能执行特定的代码，访问特定的资源。



一个ClaimsIdentity对象可以包容多个Claims

自定义ClaimsIdentity



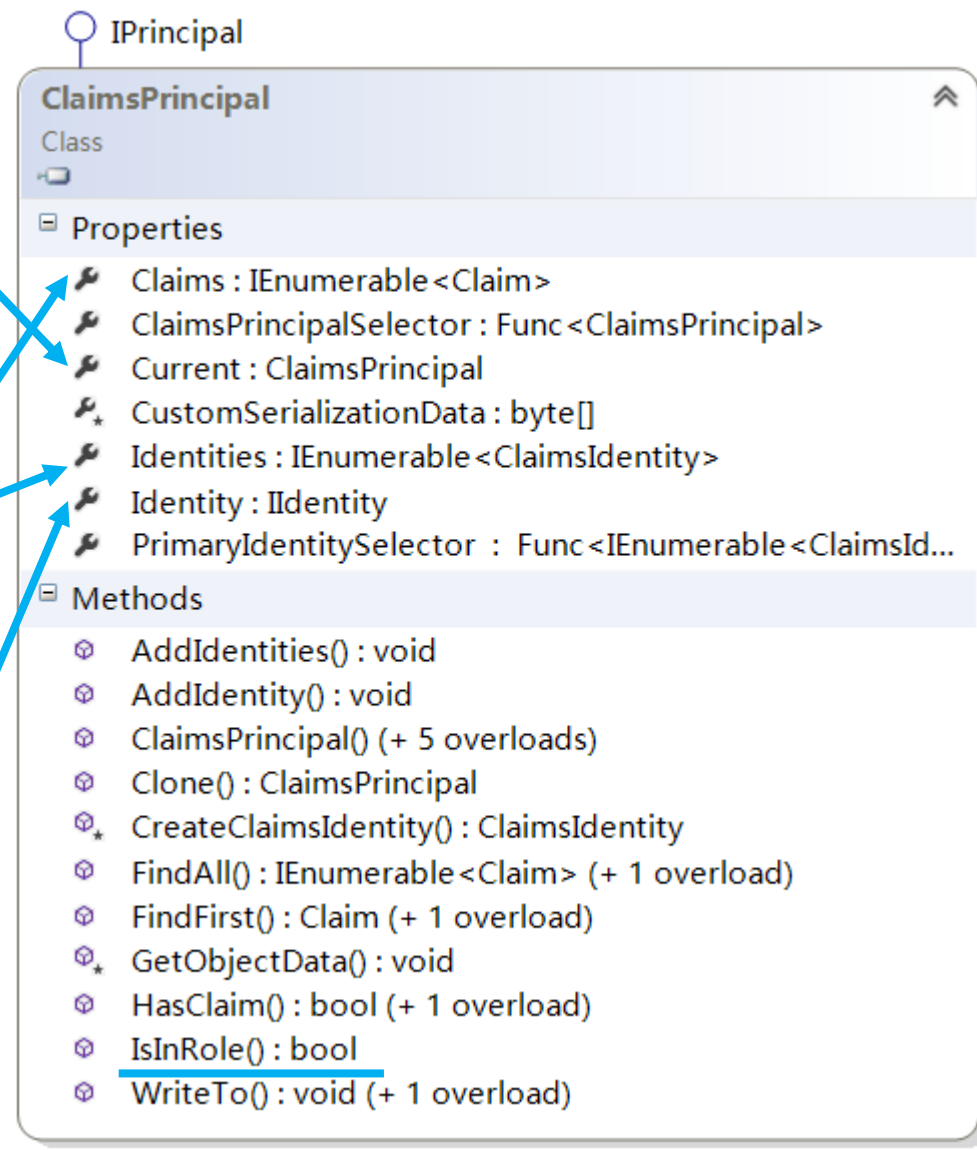
我们可以定义自己的Identity，封装特定的信息，从而简化信息的读取

ClaimsPrincipal

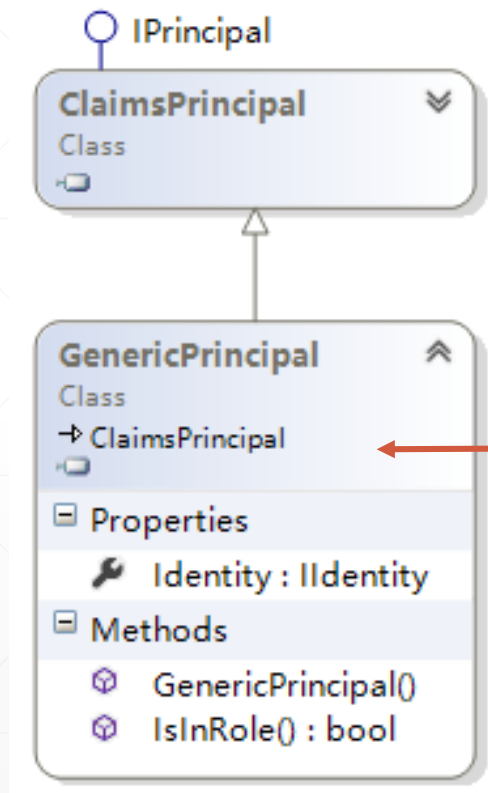
Current是一个静态属性，获取当前的 claims principal.

一个ClaimsPrincipal包容多个ClaimIdentity对象。因为每个Identity对象又各自包容着多个Claim，所以Claims属性把所有Identity对象的Claims全部包容了起来以方便使用。

Identity属性则引用本Principal的主要身份 (the primary claims identity)

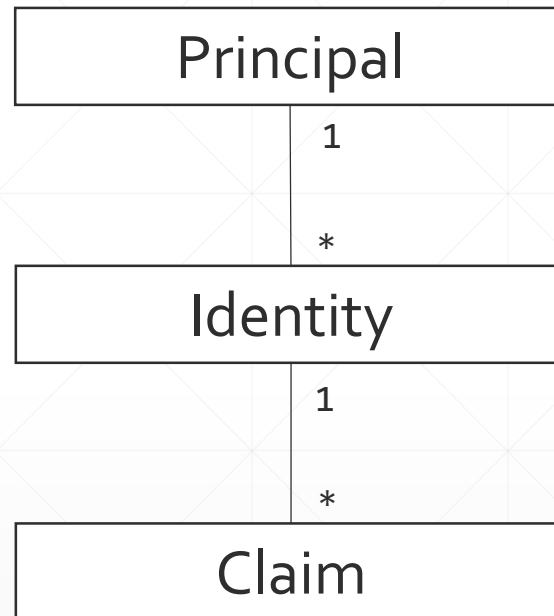


GenericPrincipal



在实例化此类时，通过构造方法，可以动态地给它添加多个“角色 (Role)”。

小结



ASP.NET core中的身份认证

使用Cookie保存认证信息

ASP.NET core 中的安全机制



ASP.NET core应用采用中间件的方式完成认证与授权工作。



对于认证，最简单的是使用Cookie，复杂一些的是采用OAuth/OpenId方式。



对于授权，最简单的是使用Role，复杂一些的是采用Policy方式。

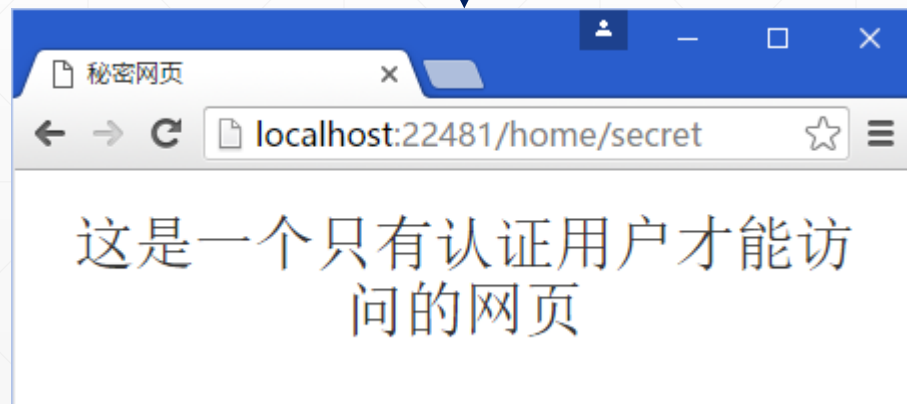
引例：ASPcoreCookieDemo

```
0 references
public class HomeController : Controller
{
    0 references
    public IActionResult Index()
    {
        return View();
    }
    [Authorize]
    0 references
    public IActionResult Secret()
    {
        return View();
    }
}
```

/home/index

ASP.NET core安全示例

访问秘密网页



查看HTTP请求与响应的细节

▼ Request Headers view parsed


GET /home/secret HTTP/1.1

Host: localhost:22481

Connection: keep-alive

由于Secret()方法上添加了
[Authorize]，所以未认证用户
访问时，会得到一个**401**响应。

[Authorize]

Filter		
<input type="checkbox"/> Regex <input type="checkbox"/> Hide data U		
Name	Status	Type
 secret	401	document

▼ Response Headers view parsed

HTTP/1.1 401 Unauthorized

Content-Length: 0

Server: Kestrel

X-SourceFiles: =?UTF-8?B?SjpcTU9PQ1zlvZXc

TUC5ORVQgY29yZeuiewFqOacuuWItlxTZWN1cm10

X-Powered-By: ASP.NET

Date: Sat, 18 Jun 2016 11:19:55 GMT

测试身份验证机制

在Startup.cs中定义一方法，创建一个虚拟用户.....

```
//此方法创建一个“虚拟的”用户
1 reference
private ClaimsPrincipal MockPrincipal()
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.Name, "jxl"),
        new Claim(ClaimTypes.Email, "jinxuliang@bit.edu.cn"),
        new Claim(ClaimTypes.Role, "User")
    };
    var id = new ClaimsIdentity(claims,
        authenticationType: "ApplicationCookie",
        nameType: ClaimTypes.Name,
        roleType: ClaimTypes.Role);
    return new ClaimsPrincipal(id);
}
```

```
public void Configure(IApplicationBuilder app)
{
    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();

    app.Use(async (context, next) => {
        var principle = MockPrincipal();
        context.User = principle;
        await next();
    });

    app.UseMvcWithDefaultRoute();
}
```

在HTTP管线中添加一个中间件，调用前述方法人工地设置一个“已登录”的用户。

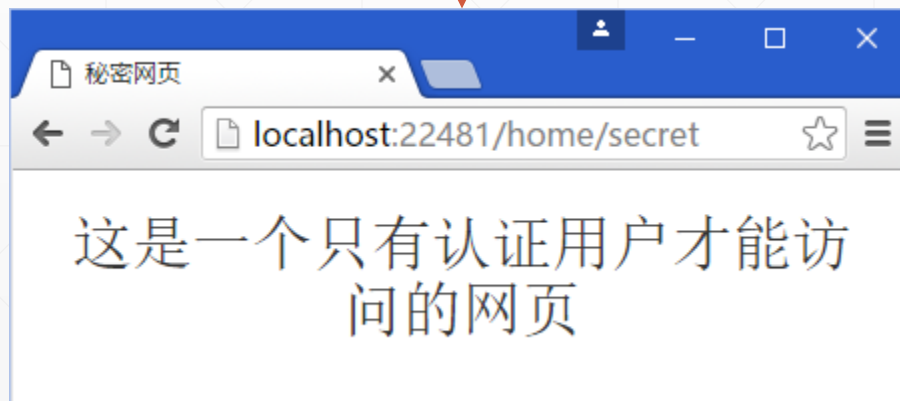
现在可以访问“秘密网页”.....

/home/index

ASP.NET core安全示例

访问秘密网页

```
var principle = MockPrincipal();  
context.User = principle;
```



在实际Web项目中.....

用户输入用户名与密码登录网站



```
graph TD; A[用户输入用户名与密码登录网站] --> B[保存登录凭证到Cookies中]; B --> C[检查凭证，授权访问受保护的资源];
```

保存登录凭证到
Cookies 中

检查凭证，授权访问
受保护的资源

使用Cookie保存用户凭据

添加两个依赖：

```
"Microsoft.AspNetCore.Authentication"
```

```
"Microsoft.AspNetCore.Authentication.Cookies"
```

构建HTTP请求处理管线

```
public void Configure(IApplicationBuilder app)
{
    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();

    //设置使用Cookie来保存用户凭据
    var options = new CookieAuthenticationOptions();
    //此名字用于标识中间件
    options.AuthenticationScheme = "MyCookieMiddlewareInstance";
    //此路径用于登录
    options.LoginPath = new PathString("/Account/Login/");
    //自动参与身份验证
    optionsAutomaticAuthenticate = true;
    //当身份验证失败时,重定向到LoginPath
    optionsAutomaticChallenge = true;

    //启用Cookie中间件
    app.UseCookieAuthentication(options);

    app.UseMvcWithDefaultRoute();
}
```

启用Cookie
认证中间件

设计一个登录页面



A screenshot of a web browser window showing a login page. The browser's address bar displays 'localhost:22481/Account/Login/'. The page has a blue header with the text '用户登录'. The main content area is white and contains the title '请登录' in large black characters. Below the title are two input fields: '用户名' (Username) with the value 'jinxuliang' and '密码' (Password) with masked characters '.....'. A '登录' (Login) button is positioned at the bottom left of the form.

登录之后，用户将可以访问秘密网页.....



ASP.NET core所保存Cookie

The screenshot shows the Chrome DevTools Resources tab. On the left sidebar, the 'Cookies' folder is expanded, and 'localhost' is selected. The main panel displays a table of cookies. The first row is highlighted in red and contains the following data:

Name	Value	Domain	Path	Expires	Size	HTTP
.AspNetCore.MyCookieMiddlewareInstance	CfDJ8A8x04x36pIMj44Gw5cu...	localhost	/	2016-...	513	✓

用Cookie保存登录凭据的关键代码.....

```
[HttpPost]
0 references
public async Task<IActionResult> Login(UserLoginViewModel user)
{
    //依据用户提交的登录信息, 创建Principal
    var principal = CreatePrincipal(user);

    //使用指定名字的Cookie中间件, 将用户凭据持久化保存到Cookie中
    await HttpContext.Authentication.SignInAsync(
        "MyCookieMiddlewareInstance",
        principal, new AuthenticationProperties
        {
            IsPersistent = true
        });

    //重定向到网站主页
    return RedirectToAction("Index", "Home");
}
```

ASP.NET core中的授权

示例：ExploreASPCoreSecurity

ASP.NET core 应用中实现授权的两种方式

Roles (角色)

- 基于“角色”实现
- 本应用有哪些人使用，每种类型的人能做什么？

Policy (策略)

- 设计几套安全方案
- 给特定的资源分配应用特定的安全策略

使用[Authorize]设定Web资源访问权限



[Authorize]其实是一个ActionFilter，在Action方法执行前执行。如果不能通过它的权限检查，Action方法将不会有被执行的机会。



[Authorize]可以在Action或Controller上针对具体用户名或角色进行授权：

```
//单用户名，单角色
[Authorize(Users="jxl")]
[Authorize(Roles="Admin")]

//多用户名，多角色
[Authorize(Users="jxl,bitfan")]
[Authorize(Roles="Admin,User")]
```

ExploreASPCoreSecurity示例解析

示例中定义了两种角色：“Admin（管理员）”和“User（普通用户）”，规定Home控制器中的Secret()方法只允许Admin角色访问：

```
[Authorize(Roles = "Admin")]  
0 references  
public IActionResult Secret()  
{  
    return View(User);  
}
```

示例中：

1. 用户“jxl”属于“Admin”
2. 用户“bitfan”属于“User”

现在我们需要区分出两种情况：

1. 未登录用户访问Secret方法。
2. 是已登录用户，但此用户不属于“Admin”角色。

授权相关设置

当未登录用户访问受保护资源时，重定向到此URL

```
//设定Cookie的相关参数
var options = new CookieAuthenticationOptions();
options.AuthenticationScheme = "MyCookieMiddlewareInstance";
options.LoginPath = new PathString("/Account/Login/");
options.AccessDeniedPath = new PathString("/Account/Forbidden/");
optionsAutomaticAuthenticate = true;
optionsAutomaticChallenge = true;

app.UseCookieAuthentication(options);
```

当权限不足的用户尝试访问受保护资源时，重定向到此URL

示例的运行结果-未认证用户场景

MVC示例

您是游客，要访问秘密页面请登录

登录为Admin

登录为用户

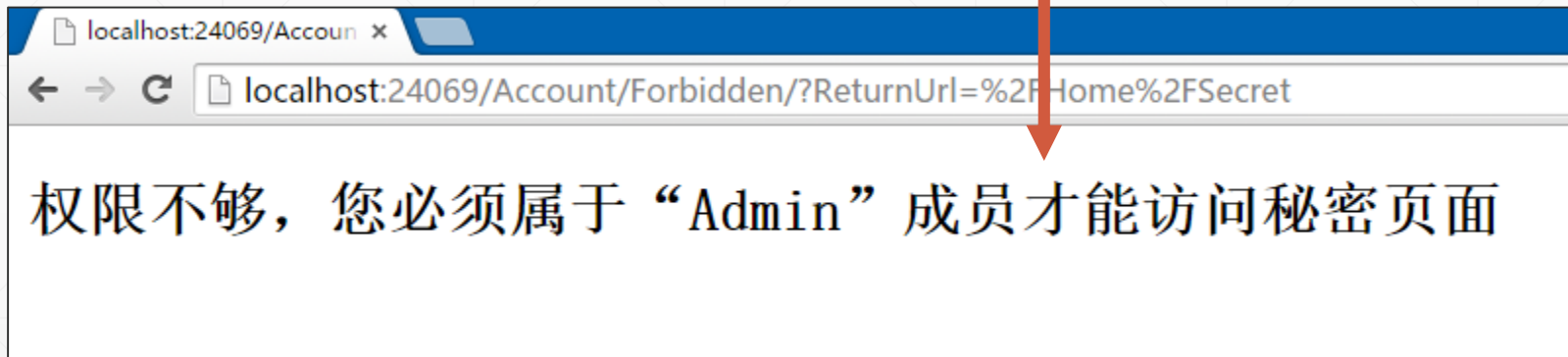
访问秘密页面

localhost:24069/Account x

localhost:24069/Account/Login/?ReturnUrl=%2FHome%2FSecret

未登录用户，您需要登录才能访问秘密页面

“权限不够”场景



“授权成功”场景

MVC示例

您是游客，要访问秘密页面请登录

登录为Admin

登录为User

访问秘密页面

jxl登录成功

返回首页

访问秘密页面



授权策略

可以在Startup.cs中定义一个或多个授权策略 (Policy)

0 references

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    //定义授权策略
    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdministratorRole", policy => policy.RequireRole("Admin"));
    });
}
```

应用授权策略

现在即可指定控制器或Action方法应用特定的授权策略。

```
[Authorize(Policy = "RequireAdministratorRole")]  
0 references  
public IActionResult Secret()  
{  
    |   return View(User);  
}
```

开发建议

与直接指定角色相比，授权策略要灵活得多，并且我们可以随意地依据不同场景加载不同的授权策略，或者使用新的策略替换掉原有的策略，推荐使用。

单页面应用中的授权

概述



单页面应用中，我们通常使用AngularJS等前端框架访问Web API，与MVC类似，某些Web API Service也是只允许拥有相应权限的用户调用。



ASP.NET core中，MVC与Web API其实是一个东西，不同之处在于Web API的路由习惯上以“/api”开始，并且Web API不返回HTML网页，而是返回“纯”的数据（通常是Json格式的）。

示例说明

```
[Route("api/[controller]")]
0 references
public class MyServiceController : Controller
{
    [HttpGet]
    [Route("open")]
    0 references
    public IActionResult GetInfo()
    {
        var result= new { info = "Hello,this is open service" };
        return Json(result);
    }

    [HttpGet]
    [Route("secret")]
    [Authorize(Roles ="Admin")]
    0 references
    public IActionResult GetSecretInfo()
    {
        var result = new { info = "this is secret info." };
        return Json(result);
    }
}
```

示例中定义了两个Web API方法，一个是开放的，另一个则指明只有管理员才能访问。

示例截图

Web API示例（AngularJS访问）

访问普通的Web API

访问秘密的Web API

Hello,this is open service

对于开放的API，无需登录即可访问

Web API示例（AngularJS访问）

访问普通的Web API

访问秘密的Web API

{"data":"","status":401}

未登录情况下，访问秘密的API，将得到一个401响应

示例截图



以管理员身份登录之后，可以看到，
能顺利地访问秘密的Web API。

技术关键点

通过URL区分开是Web API还是MVC请求，对Web API请求做特殊处理，仅返回401，不显示“拒绝访问”的Web网页

```
//设定Cookie的相关参数
var options = new CookieAuthenticationOptions();
options.AuthenticationScheme = "MyCookieMiddlewareInstance";
options.LoginPath = new PathString("/Account/Login/");
options.AccessDeniedPath = new PathString("/Account/Forbidden/");
optionsAutomaticAuthenticate = true;
optionsAutomaticChallenge = true;
//区分对待Web API和MVC，当未认证时，MVC应该显示“登录”对话框
//而Web API只需返回401即可。
Func<CookieRedirectContext, Task> OnRedirect = ctx =>
{
    if (ctx.Request.Path.StartsWithSegments("/api"))
    {
        ctx.Response.StatusCode = (int)HttpStatusCode.Unauthorized;
    }
    else
    {
        ctx.Response.StatusCode = 302;
        ctx.Response.Headers["Location"] = ctx.RedirectUri;
    }

    return Task.FromResult(0);
};
//挂接“重定向”回调方法
options.Events = new CookieAuthenticationEvents
{
    OnRedirectToLogin = OnRedirect,
    OnRedirectToAccessDenied = OnRedirect
};
app.UseCookieAuthentication(options);
```

小结



在ASP.NET core中，Web API与MVC可以共用一套基于Cookie 的身份验证与授权机制，大大地方便了开发。



如果需要的话，我们也可以将Web API与MVC分开，Web API可以采用Bearer Token进行身份验证，这时，通常会采用Oauth 2.0。可以在自己的项目中集成开源Identity Server做到这点。



ASP.NET core还支持将Cookie保存于硬盘上的指定位置，从而让多个ASP.NET core应用（甚至是混杂着早期ASP.NET MVC/Web API）实现“单点登录”的功能。