## **简单授权**

[Authorize] // Controller级别

public class SampleDataController : Controller

{

[Authorize] // Action级别

public IActionResult SampleAction()

{

}

}

### **IAllowAnonymous**

[Authorize]

public class AccountController : Controller

{

[AllowAnonymous]

public ActionResult Login()

{

}

public ActionResult Logout()

{

}

}

#### **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。

[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。

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。

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>());

......

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**

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);

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

public void AddPolicy(string name, AuthorizationPolicy policy)

{

}

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

{

}

public AuthorizationPolicy GetPolicy(string name)

{

}

}

PolicyMap字典

我们添加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

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)

{

}

public static AuthorizationPolicy Combine(IEnumerable<AuthorizationPolicy> policies)

{

}

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

{

}

}

eequirements属性是策略的核心了，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。

## **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。

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));

});

}

### **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)

{

........

if (calculatedAge >= requirement.MinimumAge)

{

// 授权通过

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)));

});

### **多Handler模式**

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>();