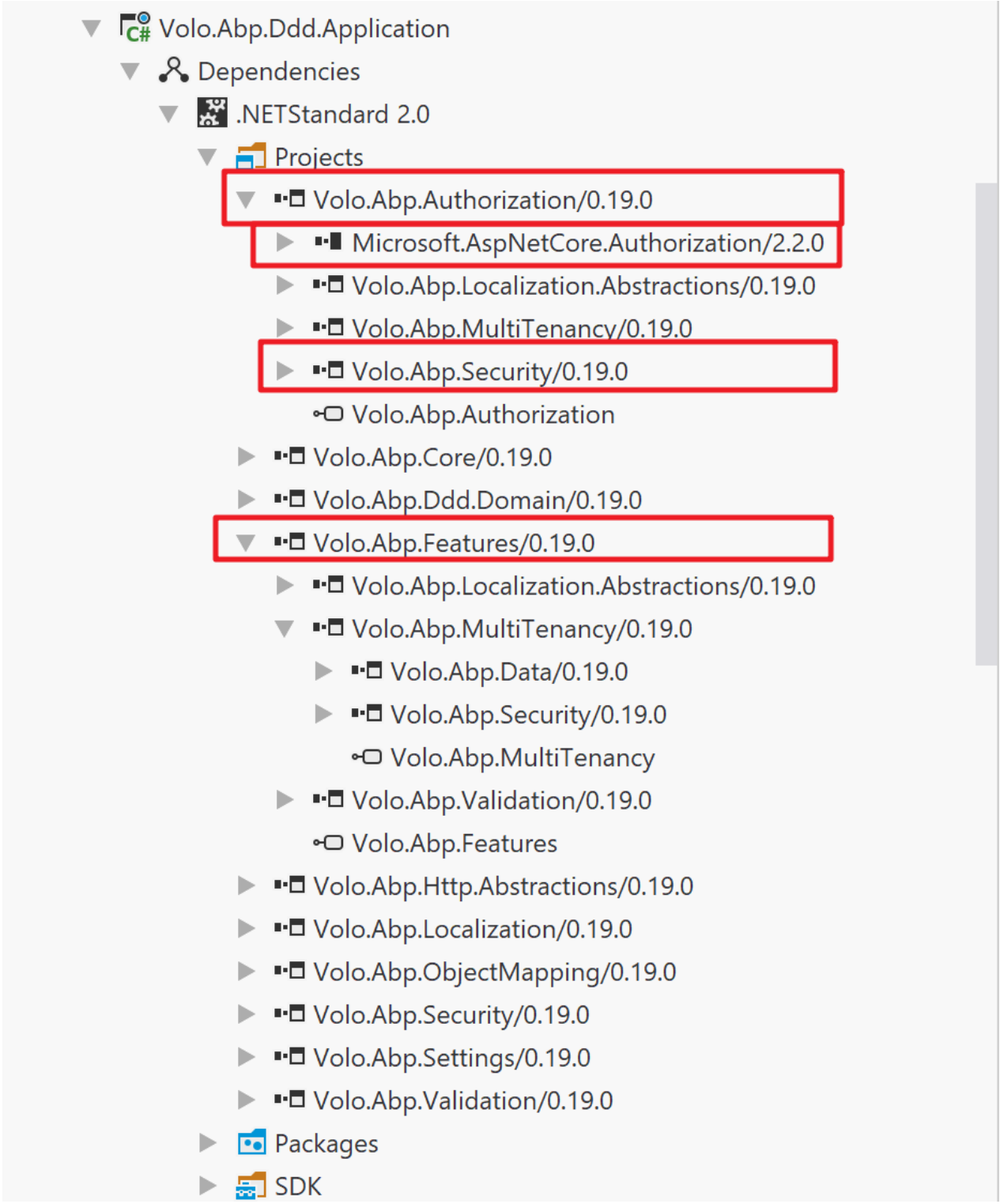
一、简要说明

在上篇文章里面，我们在 `ApplicationService` 当中看到了权限检测代码，通过注入 `IAuthorizationService` 就可以实现权限检测。不过跳转到源码才发现，这个接口是 ASP.NET Core 原生提供的 “基于策略” 的权限验证接口，这就说明 ABP vNext 基于原生的授权验证框架进行了自定义扩展。

让我们来看一下 **Volo.Abp.Ddd.Application** 项目的依赖结构(权限相关)。



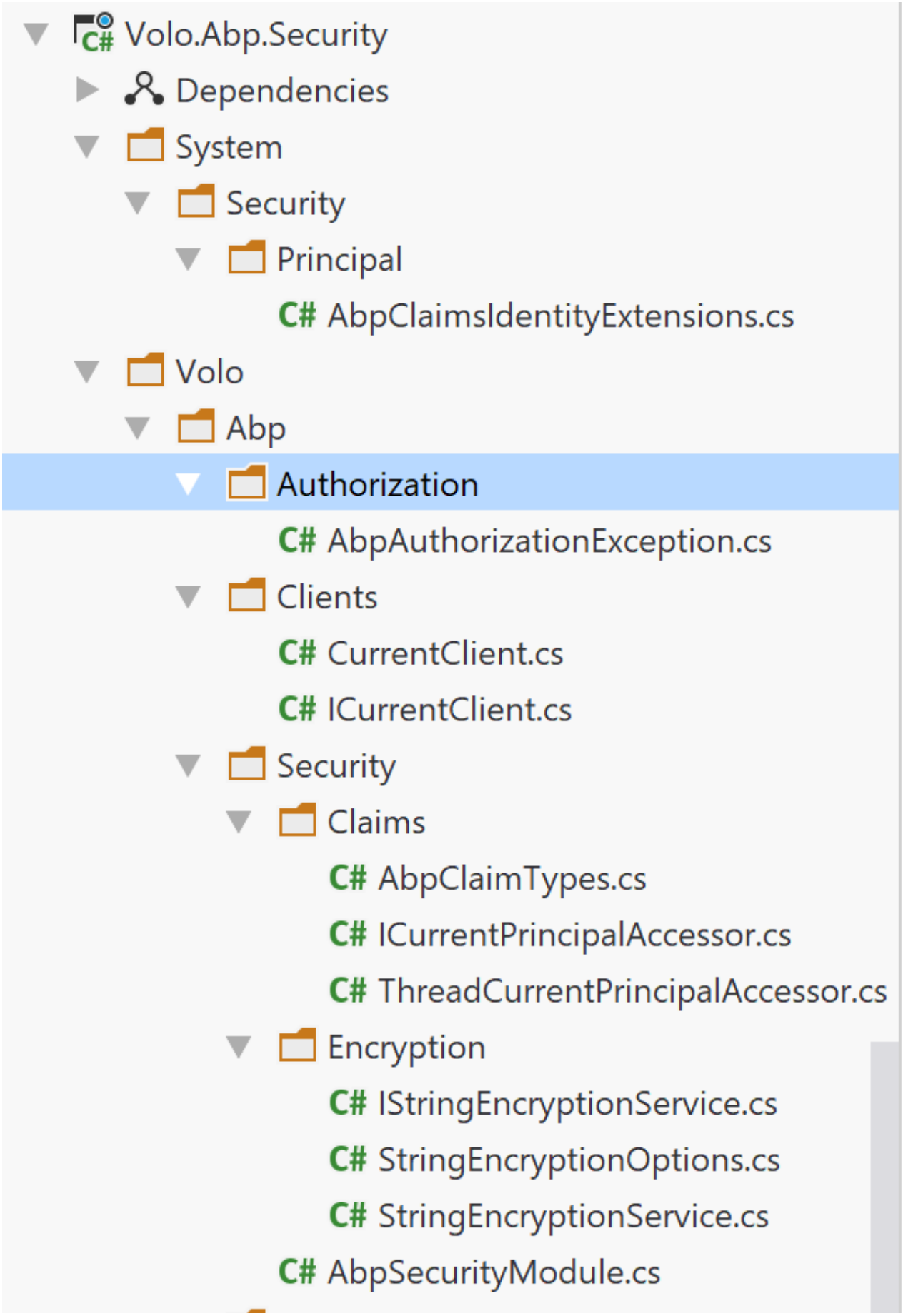
本篇文章下面的内容基本就会围绕上述框架模块展开，本篇文章通篇较长，因为还涉及到 .NET Core Identity 与 IdentityServer4 这两部分。关于这两部分的内容，我会在本篇文章大概讲述 ABP vNext 的实现，关于更加详细的内容，请查阅官方文档或其他博主的博客。

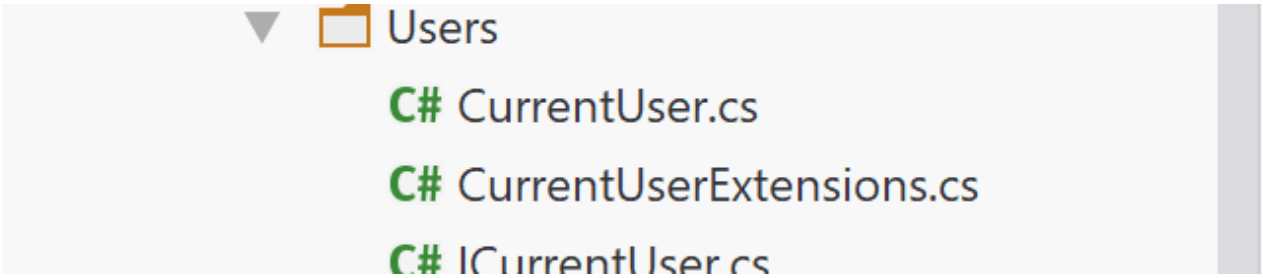
二、源码分析

ABP vNext 关于权限验证和权限定义的部分，都存放在 **Volo.Abp.Authorization** 和 **Volo.Abp.Security** 模块内部。源码分析我都比较喜欢倒推，即通过实际的使用场景，**反向推导** 基础实现，所以后面文章编写的顺序也将会以这种方式进行。

2.1 Security 基础组件库

这里我们先来到 **Volo.Abp.Security**，因为这个模块代码和类型都是最少的。这个项目都没有模块定义，说明里面的东西都是定义的一些基础组件。





2.1.1 Claims 与 Identity 的快捷访问

先从第一个扩展方法开始，这个扩展方法里面比较简单，它主要是提供对 `ClaimsPrincipal` 和 `IIdentity` 的快捷访问方法。比如我要从 `ClaimsPrincipal` / `IIdentity` 获取租户 Id、用户 Id 等。

```
public static class AbpClaimsIdentityExtensions
{
  public static Guid? FindUserId([NotNull] this ClaimsPrincipal principal)
  {
    Check.NotNull(principal, nameof(principal));

        // 根据 AbpClaimTypes.UserId 查找对应的值。
    var userIdOrNull = principal.Claims?.FirstOrDefault(c => c.Type ==
AbpClaimTypes.UserId);
    if (userIdOrNull == null || userIdOrNull.Value.IsNullOrWhiteSpace())
    {
      return null;
    }

        // 返回 Guid 对象。
    return Guid.Parse(userIdOrNull.Value);
  }
```

2.1.2 未授权异常的定义

这个异常我们在老版本 ABP 里面也见到过，它就是 `AbpAuthorizationException` 。只要有任何未授权的操作，都会导致该异常被抛出。后面我们在讲解 ASP.NET Core MVC 的时候就会知道，在默认的错误码处理中，针对于程序抛出的 `AbpAuthorizationException` ，都会视为 403 或者 401 错误。

```
public class DefaultHttpExceptionStatusCodeFinder : IHttpExceptionStatusCodeFinder,
ITransientDependency
{
  // ... 其他代码

  public virtual HttpStatusCode GetStatusCode(HttpContext httpContext, Exception
exception)
  {
    // ... 其他代码

    // 根据 HTTP 协议对于状态码的定义，401 表示的是没有登录的用于尝试访问受保护的资源。而 403 则表
示用户已经登录，但他没有目标资源的访问权限。
    if (exception is AbpAuthorizationException)
    {
```

```
            return httpContext.User.Identity.IsAuthenticated
                ? HttpStatusCode.Forbidden
                : HttpStatusCode.Unauthorized;
        }

        // ... 其他代码
    }

    // ... 其他代码
}
```

就 `AbpAuthorizationException` 异常来说，它本身并不复杂，只是一个简单的异常而已。只是因为它的特殊含义，在 ABP vNext 处理异常时都会进行特殊处理。

只是在这里我说明一下，ABP vNext 将它所有的异常都设置为可序列化的，这里的可序列化不仅仅是将 `Serialzable` 标签打在类上就行了。ABP vNext 还创建了基于 `StreamingContext` 的构造函数，方便我们后续对序列化操作进行定制化处理。

关于运行时序列化的相关文章，可以参考 《CLR Via C#》第 24 章，我也编写了相应的 **读书笔记** 。

2.1.3 当前用户与客户端

开发人员经常会在各种地方需要获取当前的用户信息，ABP vNext 将当前用户封装到 `ICurrentUser` 与其实现 `CurrentUser` 当中，使用时只需要注入 `ICurrentUser` 接口即可。

我们首先康康 `ICurrentUser` 接口的定义：

```
public interface ICurrentUser
{
    bool IsAuthenticated { get; }

    [CanBeNull]
    Guid? Id { get; }

    [CanBeNull]
    string UserName { get; }

    [CanBeNull]
    string PhoneNumber { get; }

    bool PhoneNumberVerified { get; }

    [CanBeNull]
    string Email { get; }

    bool EmailVerified { get; }

    Guid? TenantId { get; }

    [NotNull]
    string[] Roles { get; }

    [CanBeNull]
```

```
  Claim FindClaim(string claimType);

  [NotNull]
  Claim[] FindClaims(string claimType);

  [NotNull]
  Claim[] GetAllClaims();

  bool IsInRole(string roleName);
}
```

那么这些值是从哪儿来的呢？从带有 `Claim` 返回值的方法来看，肯定就是从 `HttpContext.User` 或者 `Thread.CurrentPrincipal` 里面拿到的。

那么它的实现就非常简单了，只需要注入 ABP vNext 为我们提供的 `ICurrentPrincipalAccessor` 访问器，我们就能够拿到这个身份容器(`ClaimsPrincipal`)。

```
public class CurrentUser : ICurrentUser, ITransientDependency
{
  // ... 其他代码

  public virtual string[] Roles => FindClaims(AbpClaimTypes.Role).Select(c =>
c.Value).ToArray();

  private readonly ICurrentPrincipalAccessor _principalAccessor;

  public CurrentUser(ICurrentPrincipalAccessor principalAccessor)
  {
    _principalAccessor = principalAccessor;
  }

  // ... 其他代码

  public virtual Claim[] FindClaims(string claimType)
  {
    // 直接使用 LINQ 查询对应的 Type 就能拿到上述信息。
    return _principalAccessor.Principal?.Claims.Where(c => c.Type ==
claimType).ToArray() ?? EmptyClaimsArray;
  }

  // ... 其他代码
}
```

至于 `CurrentUserExtensions` 扩展类，里面只是对 `ClaimsPrincipal` 的搜索方法进行了多种封装而已。

> PS:
>
> 除了 `ICurrentUser` 与 `ICurrentClient` 之外，在 ABP vNext 里面还有 `ICurrentTenant` 来获取当前租户信息。通过这三个组件，取代了老 ABP 框架的 `IAbpSession` 组件，三个组件都没有 `IAbpSession.Use()` 扩展方法帮助我们临时更改当前用户/租户。

2.1.4 ClaimsPrincipal 访问器

关于 ClaimsPrincipal 的内容，可以参考杨总的 《ASP.NET Core 之 Identity 入门》 进行了解，大致来说就是存有 `Claim` 信息的聚合对象。

关于 ABP vNext 框架预定义的 Claim Type 都存放在 `AbpClaimTypes` 类型里面的，包括租户 Id、用户 Id 等数据，这些玩意儿最终会被放在 JWT(JSON Web Token) 里面去。

一般来说 `ClaimsPrincipal` 里面都是从 `HttpContext.User` 或者 `Thread.CurrentPrincipal` 得到的，ABP vNext 为我们抽象出了一个快速访问接口 `ICurrentPrincipalAccessor` 。开发人员注入之后，就可以获得当前用户的 `ClaimsPrincipal` 对象。

```
public interface ICurrentPrincipalAccessor
{
  ClaimsPrincipal Principal { get; }
}
```

对于 `Thread.CurrentPrincipal` 的实现：

```
public class ThreadCurrentPrincipalAccessor : ICurrentPrincipalAccessor,
ISingletonDependency
{
  public virtual ClaimsPrincipal Principal => Thread.CurrentPrincipal as
ClaimsPrincipal;
}
```

而针对于 Http 上下文的实现，则是放在 **Volo.Abp.AspNetCore** 模块里面的。

```
public class HttpContextCurrentPrincipalAccessor : ThreadCurrentPrincipalAccessor
{
  // 如果没有获取到数据，则使用 Thread.CurrentPrincipal。
  public override ClaimsPrincipal Principal => _httpContextAccessor.HttpContext?.User
?? base.Principal;

  private readonly IHttpContextAccessor _httpContextAccessor;

  public HttpContextCurrentPrincipalAccessor(IHttpContextAccessor httpContextAccessor)
  {
    _httpContextAccessor = httpContextAccessor;
  }
}
```

扩展知识：两者的区别?

`Thread.CurrentPrincipal` 可以设置/获得当前线程的 `ClaimsPrincipal` 数据，而 `HttpContext?.User` 一般都是被 ASP.NET Core 中间件所填充的。

最新的 ASP.NET Core 开发建议是不要使用 `Thread.CurrentPrincipal` 和 `ClaimsPrincipal.Current` (内部实现还是使用的前者)。这是因为 `Thread.CurrentPrincipal` 是一个静态成员...而这个静态成员在异步代码中会出现各种问题，例如有以下代码：

```
// Create a ClaimsPrincipal and set Thread.CurrentPrincipal
var identity = new ClaimsIdentity();
identity.AddClaim(new Claim(ClaimTypes.Name, "User1"));
Thread.CurrentPrincipal = new ClaimsPrincipal(identity);

// Check the current user
Console.WriteLine($"Current user: {Thread.CurrentPrincipal?.Identity.Name}");

// For the method to complete asynchronously
await Task.Yield();

// Check the current user after
Console.WriteLine($"Current user: {Thread.CurrentPrincipal?.Identity.Name}");
```

当 `await` 执行完成之后会产生线程切换，这个时候 Thread.CurrentPrincipal 的值就是 null 了，这就会产生不可预料的后果。

如果你还想了解更多信息，可以参考以下两篇博文：

- DAVID PINE - **《WHAT HAPPENED TO MY THREAD.CURRENTPRINCIPAL》**
- SCOTT HANSELMAN - **《System.Threading.Thread.CurrentPrincipal vs. System.Web.HttpContext.Current.User or why FormsAuthentication can be subtle》**

2.1.5 字符串加密工具

这一套东西就比较简单了，是 ABP vNext 为我们提供的一套开箱即用组件。开发人员可以使用 `IStringEncryptionService` 来加密/解密你的字符串，默认实现是基于 `Rfc2898DeriveBytes` 的。关于详细信息，你可以阅读具体的代码，这里不再赘述。

2.2 权限与校验

在 **Volo.Abp.Authorization** 模块里面就对权限进行了具体定义，并且基于 ASP.NET Core Authentication 进行无缝集成。如果读者对于 ASP.NET Core 认证和授权不太了解，可以去学习一下 **雨夜朦胧** 大神的 **《ASP.NET Core 认证于授权》** 系列文章，这里就不再赘述。

2.2.1 权限的注册

在 ABP vNext 框架里面，所有用户定义的权限都是通过继承 `PermissionDefinitionProvider` ，在其内部进行注册的。

```
public abstract class PermissionDefinitionProvider : IPermissionDefinitionProvider,
ITransientDependency
{
    public abstract void Define(IPermissionDefinitionContext context);
}
```

开发人员继承了这个 Provider 之后，在 `Define()` 方法里面就可以注册自己的权限了，这里我以 Blog 模块的简化 Provider 为例。

```
public class BloggingPermissionDefinitionProvider : PermissionDefinitionProvider
{
    public override void Define(IPermissionDefinitionContext context)
```

```
    {
        var bloggingGroup = context.AddGroup(BloggingPermissions.GroupName,
L("Permission:Blogging"));

        // ... 其他代码。

        var tags = bloggingGroup.AddPermission(BloggingPermissions.Tags.Default,
L("Permission:Tags"));
        tags.AddChild(BloggingPermissions.Tags.Update, L("Permission:Edit"));
        tags.AddChild(BloggingPermissions.Tags.Delete, L("Permission:Delete"));
        tags.AddChild(BloggingPermissions.Tags.Create, L("Permission:Create"));

        var comments =
bloggingGroup.AddPermission(BloggingPermissions.Comments.Default,
L("Permission:Comments"));
        comments.AddChild(BloggingPermissions.Comments.Update, L("Permission:Edit"));
        comments.AddChild(BloggingPermissions.Comments.Delete,
L("Permission:Delete"));
        comments.AddChild(BloggingPermissions.Comments.Create,
L("Permission:Create"));
    }

    // 使用本地化字符串进行文本显示。
    private static LocalizableString L(string name)
    {
        return LocalizableString.Create<BloggingResource>(name);
    }
}
```

从上面的代码就可以看出来，权限被 ABP vNext 分成了 **权限组定义** 和 **权限定义**，这两个东西我们后面进行重点讲述。那么这些 Provider 在什么时候被执行呢？找到权限模块的定义，可以看到如下代码：

```
[DependsOn(
    typeof(AbpSecurityModule),
    typeof(AbpLocalizationAbstractionsModule),
    typeof(AbpMultiTenancyModule)
    )]
public class AbpAuthorizationModule : AbpModule
{
    public override void PreConfigureServices(ServiceConfigurationContext context)
    {
        // 在 AutoFac 进行组件注册的时候，根据组件的类型定义视情况绑定拦截器。

 context.Services.OnRegistred(AuthorizationInterceptorRegistrar.RegisterIfNeeded);

        // 在 AutoFac 进行组件注册的时候，根据组件的类型，判断是否是 Provider。
        AutoAddDefinitionProviders(context.Services);
    }

    public override void ConfigureServices(ServiceConfigurationContext context)
    {
        // 注册认证授权服务。
```

```
        context.Services.AddAuthorization();

        // 替换掉 ASP.NET Core 提供的权限处理器，转而使用 ABP vNext 提供的权限处理器。
        context.Services.AddSingleton<IAuthorizationHandler,
PermissionRequirementHandler>();

        // 这一部分是添加内置的一些权限值检查，后面我们在将 PermissionChecker 的时候会提到。
        Configure<PermissionOptions>(options =>
        {
            options.ValueProviders.Add<UserPermissionValueProvider>();
            options.ValueProviders.Add<RolePermissionValueProvider>();
            options.ValueProviders.Add<ClientPermissionValueProvider>();
        });
    }

    private static void AutoAddDefinitionProviders(IServiceCollection services)
    {
        var definitionProviders = new List<Type>();

        services.OnRegistred(context =>
        {
            if
(typeof(IPermissionDefinitionProvider).IsAssignableFrom(context.ImplementationType))
            {
                definitionProviders.Add(context.ImplementationType);
            }
        });

        // 将获取到的 Provider 传递给 PermissionOptions 。
        services.Configure<PermissionOptions>(options =>
        {
            options.DefinitionProviders.AddIfNotContains(definitionProviders);
        });
    }
}
```

可以看到在注册组件的时候，ABP vNext 就会将这些 Provider 传递给 `PermissionOptions` ，我们根据 `DefinitionProviders` 字段找到有一个地方会使用到它，就是 `PermissionDefinitionManager` 类型的 `CreatePermissionGroupDefinitions()` 方法。

```
protected virtual Dictionary<string, PermissionGroupDefinition>
CreatePermissionGroupDefinitions()
{
    //  创建一个权限定义上下文。
    var context = new PermissionDefinitionContext();

    // 创建一个临时范围用于解析 Provider，Provider 解析完成之后即被释放。
    using (var scope = _serviceProvider.CreateScope())
    {
        // 根据之前的类型，通过 IoC 进行解析出实例，指定各个 Provider 的 Define() 方法，会向权限
上下文填充权限。
        var providers = Options
```

```
            .DefinitionProviders
            .Select(p => scope.ServiceProvider.GetRequiredService(p) as
IPermissionDefinitionProvider)
            .ToList();

        foreach (var provider in providers)
        {
            provider.Define(context);
        }
    }

    // 返回权限组名称 - 权限组定义的字典。
    return context.Groups;
}
```

你可能会奇怪，为什么返回的是一个权限组名字和定义的键值对，而不是返回的权限数据，我们之前添加的权限去哪儿了呢?

2.2.2 权限和权限组的定义

要搞清楚这个问题，我们首先要知道权限与权限组之间的关系是怎样的。回想我们之前在 Provider 里面添加权限的代码，首先我们是构建了一个权限组，然后往权限组里面添加的权限。权限组的作用就是将权限按照组的形式进行划分，方便代码进行访问于管理。

```
public class PermissionGroupDefinition
{
    /// <summary>
    /// 唯一的权限组标识名称。
    /// </summary>
    public string Name { get; }

    // 开发人员针对权限组的一些自定义属性。
    public Dictionary<string, object> Properties { get; }

    // 权限所对应的本地化名称。
    public ILocalizableString DisplayName
    {
        get => _displayName;
        set => _displayName = Check.NotNull(value, nameof(value));
    }
    private ILocalizableString _displayName;

    /// <summary>
    /// 权限的适用范围，默认是租户/租主都适用。
    /// 默认值: <see cref="MultiTenancySides.Both"/>
    /// </summary>
    public MultiTenancySides MultiTenancySide { get; set; }

    // 权限组下面的所属权限。
    public IReadOnlyList<PermissionDefinition> Permissions =>
_permissions.ToImmutableList();
    private readonly List<PermissionDefinition> _permissions;
```

```
    // 针对于自定义属性的快捷索引器。
    public object this[string name]
    {
        get => Properties.GetOrDefault(name);
        set => Properties[name] = value;
    }

    protected internal PermissionGroupDefinition(
        string name,
        ILocalizableString displayName = null,
        MultiTenancySides multiTenancySide = MultiTenancySides.Both)
    {
        Name = name;
        // 没有传递多语言串，则使用权限组的唯一标识作为显示内容。
        DisplayName = displayName ?? new FixedLocalizableString(Name);
        MultiTenancySide = multiTenancySide;

        Properties = new Dictionary<string, object>();
        _permissions = new List<PermissionDefinition>();
    }

    // 像权限组添加属于它的权限。
    public virtual PermissionDefinition AddPermission(
        string name,
        ILocalizableString displayName = null,
        MultiTenancySides multiTenancySide = MultiTenancySides.Both)
    {
        var permission = new PermissionDefinition(name, displayName,
multiTenancySide);

        _permissions.Add(permission);

        return permission;
    }

    // 递归构建权限集合，因为定义的某个权限内部还拥有子权限。
    public virtual List<PermissionDefinition> GetPermissionsWithChildren()
    {
        var permissions = new List<PermissionDefinition>();

        foreach (var permission in _permissions)
        {
            AddPermissionToListRecursively(permissions, permission);
        }

        return permissions;
    }

    // 递归构建方法。
    private void AddPermissionToListRecursively(List<PermissionDefinition>
permissions, PermissionDefinition permission)
```

```
    {
        permissions.Add(permission);

        foreach (var child in permission.Children)
        {
            AddPermissionToListRecursively(permissions, child);
        }
    }

    public override string ToString()
    {
        return $"[{nameof(PermissionGroupDefinition)} {Name}]";
    }
}
```

通过权限组的定义代码你就会知道，现在我们的所有权限都会归属于某个权限组，这一点从之前 Provider 的 `IPermissionDefinitionContext` 就可以看出来。在权限上下文内部只允许我们通过 `AddGroup()` 来添加一个权限组，之后再通过权限组的 `AddPermission()` 方法添加它里面的权限。

权限的定义类叫做 `PermissionDefinition`，这个类型的构造与权限组定义类似，没有什么好说的。

```
public class PermissionDefinition
{
    /// <summary>
    /// 唯一的权限标识名称。
    /// </summary>
    public string Name { get; }

    /// <summary>
    /// 当前权限的父级权限，这个属性的值只可以通过 AddChild() 方法进行设置。
    /// </summary>
    public PermissionDefinition Parent { get; private set; }

    /// <summary>
    /// 权限的适用范围，默认是租户/租主都适用。
    /// 默认值: <see cref="MultiTenancySides.Both"/>
    /// </summary>
    public MultiTenancySides MultiTenancySide { get; set; }

    /// <summary>
    /// 适用的权限值提供者，这块我们会在后面进行讲解，为空的时候则使用所有的提供者进行校验。
    /// </summary>
    public List<string> Providers { get; } //TODO: Rename to AllowedProviders?

    // 权限的多语言名称。
    public ILocalizableString DisplayName
    {
        get => _displayName;
        set => _displayName = Check.NotNull(value, nameof(value));
    }
    private ILocalizableString _displayName;
```

```
    // 获取权限的子级权限。
    public IReadOnlyList<PermissionDefinition> Children =>
_children.ToImmutableList();
    private readonly List<PermissionDefinition> _children;

    /// <summary>
    /// 开发人员针对权限的一些自定义属性。
    /// </summary>
    public Dictionary<string, object> Properties { get; }

    // 针对于自定义属性的快捷索引器。
    public object this[string name]
    {
        get => Properties.GetOrDefault(name);
        set => Properties[name] = value;
    }

    protected internal PermissionDefinition(
        [NotNull] string name,
        ILocalizableString displayName = null,
        MultiTenancySides multiTenancySide = MultiTenancySides.Both)
    {
        Name = Check.NotNull(name, nameof(name));
        DisplayName = displayName ?? new FixedLocalizableString(name);
        MultiTenancySide = multiTenancySide;

        Properties = new Dictionary<string, object>();
        Providers = new List<string>();
        _children = new List<PermissionDefinition>();
    }

    public virtual PermissionDefinition AddChild(
        [NotNull] string name,
        ILocalizableString displayName = null,
        MultiTenancySides multiTenancySide = MultiTenancySides.Both)
    {
        var child = new PermissionDefinition(
            name,
            displayName,
            multiTenancySide)
        {
            Parent = this
        };

        _children.Add(child);

        return child;
    }

    /// <summary>
    /// 设置指定的自定义属性。
    /// </summary>
```

```
    public virtual PermissionDefinition WithProperty(string key, object value)
    {
        Properties[key] = value;
        return this;
    }

    /// <summary>
    /// 添加一组权限值提供者集合。
    /// </summary>
    public virtual PermissionDefinition WithProviders(params string[] providers)
    {
        if (!providers.IsNullOrEmpty())
        {
            Providers.AddRange(providers);
        }

        return this;
    }

    public override string ToString()
    {
        return $"[{nameof(PermissionDefinition)} {Name}]";
    }
}
```

2.2.3 权限管理器

继续回到权限管理器，权限管理器的接口定义是 `IPermissionDefinitionManager` ，从接口的方法定义来看，都是获取权限的方法，说明权限管理器主要提供给其他组件进行权限校验操作。

```
public interface IPermissionDefinitionManager
{
    // 根据权限定义的唯一标识获取权限，一旦不存在就会抛出 AbpException 异常。
    [NotNull]
    PermissionDefinition Get([NotNull] string name);

    // 根据权限定义的唯一标识获取权限，如果权限不存在，则返回 null。
    [CanBeNull]
    PermissionDefinition GetOrNull([NotNull] string name);

    // 获取所有的权限。
    IReadOnlyList<PermissionDefinition> GetPermissions();

    // 获取所有的权限组。
    IReadOnlyList<PermissionGroupDefinition> GetGroups();
}
```

接着我们来回答 2.2.1 末尾提出的问题，权限组是根据 Provider 自动创建了，那么权限呢？其实我们在权限管理器里面拿到了权限组，权限定义就很好构建了，直接遍历所有权限组拿它们的 `Permissions` 属性构建即可。

```
protected virtual Dictionary<string, PermissionDefinition>
CreatePermissionDefinitions()
{
    var permissions = new Dictionary<string, PermissionDefinition>();

    // 遍历权限定义组，这个东西在之前就已经构建好了。
    foreach (var groupDefinition in PermissionGroupDefinitions.Values)
    {
        // 递归子级权限。
        foreach (var permission in groupDefinition.Permissions)
        {
            AddPermissionToDictionaryRecursively(permissions, permission);
        }
    }

    // 返回权限唯一标识 - 权限定义 的字典。
    return permissions;
}

protected virtual void AddPermissionToDictionaryRecursively(
    Dictionary<string, PermissionDefinition> permissions,
    PermissionDefinition permission)
{
    if (permissions.ContainsKey(permission.Name))
    {
        throw new AbpException("Duplicate permission name: " + permission.Name);
    }

    permissions[permission.Name] = permission;

    foreach (var child in permission.Children)
    {
        AddPermissionToDictionaryRecursively(permissions, child);
    }
}
```

2.2.4 授权策略提供者的实现

我们发现 ABP vNext 自己实现了 `IAbpAuthorizationPolicyProvider` 接口，实现的类型就是 `AbpAuthorizationPolicyProvider` 。

这个类型它是继承的 `DefaultAuthorizationPolicyProvider` ，重写了 `GetPolicyAsync()` 方法，目的就是将 `PermissionDefinition` 转换为 `AuthorizationPolicy` 。

如果去看了 **雨夜朦胧** 大神的博客，就知道我们一个授权策略可以由多个条件构成。也就是说某一个 `AuthorizationPolicy` 可以拥有多个限定条件，当所有限定条件被满足之后，才能算是通过权限验证，例如以下代码。

```
public void ConfigureService(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
```

```
        options.AddPolicy("User", policy => policy
            .RequireAssertion(context => context.User.HasClaim(c => (c.Type ==
"EmployeeNumber" || c.Type == "Role")))
        );

        // 这里的意思是，用户角色必须是 Admin，并且他的用户名是 Alice，并且必须要有类型为
EmployeeNumber 的 Claim。
        options.AddPolicy("Employee", policy => policy
            .RequireRole("Admin")
            .RequireUserName("Alice")
            .RequireClaim("EmployeeNumber")
            .Combine(commonPolicy));
    });
}
```

这里的 `RequireRole()` 、 `RequireUserName()` 、 `RequireClaim()` 都会生成一个 `IAuthorizationRequirement` 对象，它们在内部有不同的实现规则。

```
public AuthorizationPolicyBuilder RequireClaim(string claimType)
{
    if (claimType == null)
    {
        throw new ArgumentNullException(nameof(claimType));
    }

    // 构建了一个 ClaimsAuthorizationRequirement 对象，并添加到策略的 Requirements 组。
    Requirements.Add(new ClaimsAuthorizationRequirement(claimType, allowedValues:
null));
    return this;
}
```

这里我们 ABP vNext 则是使用的 `PermissionRequirement` 作为一个限定条件。

```
public override async Task<AuthorizationPolicy> GetPolicyAsync(string policyName)
{
    var policy = await base.GetPolicyAsync(policyName);
    if (policy != null)
    {
        return policy;
    }

    var permission = _permissionDefinitionManager.GetOrNull(policyName);
    if (permission != null)
    {
        // TODO: 可以使用缓存进行优化。
        // 通过 Builder 构建一个策略。
        var policyBuilder = new AuthorizationPolicyBuilder(Array.Empty<string>());
        // 创建一个 PermissionRequirement 对象添加到限定条件组中。
        policyBuilder.Requirements.Add(new PermissionRequirement(policyName));
        return policyBuilder.Build();
    }
```

```
    return null;
}
```

与 `ClaimsAuthorizationRequirement` 不同的是，ABP vNext 并没有将限定条件处理器和限定条件定义放在一起实现，而是分开的，分别构成了 `PermissionRequirement` 和 `PermissionRequirementHandler` ，后者在模块配置的时候被注入到 IoC 里面。

> PS：
>
> 对于 Handler 来说，我们可以编写多个 Handler 注入到 IoC 容器内部，如下代码：
>
> ```
> services.AddSingleton<IAuthorizationHandler, BadgeEntryHandler>();
> services.AddSingleton<IAuthorizationHandler, HasTemporaryStickerHandler>();
> ```

首先看限定条件 `PermissionRequirement` 的定义，非常简单。

```
public class PermissionRequirement : IAuthorizationRequirement
{
    public string PermissionName { get; }

    public PermissionRequirement([NotNull]string permissionName)
    {
        Check.NotNull(permissionName, nameof(permissionName));

        PermissionName = permissionName;
    }
}
```

在限定条件内部，我们只用了权限的唯一标识来进行处理，接下来看一下权限处理器。

```
public class PermissionRequirementHandler :
AuthorizationHandler<PermissionRequirement>
{
    // 这里通过权限检查器来确定当前用户是否拥有某个权限。
    private readonly IPermissionChecker _permissionChecker;

    public PermissionRequirementHandler(IPermissionChecker permissionChecker)
    {
        _permissionChecker = permissionChecker;
    }

    protected override async Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        PermissionRequirement requirement)
    {
        // 如果当前用户拥有某个权限，则通过 Contxt.Succeed() 通过授权验证。
        if (await _permissionChecker.IsGrantedAsync(context.User,
requirement.PermissionName))
        {
            context.Succeed(requirement);
        }
```

```
    }
}
```

2.2.5 权限检查器

在上面的处理器我们看到了，ABP vNext 是通过权限检查器来校验某个用户是否满足某个授权策略，先看一下 `IPermissionChecker` 接口的定义，基本都是传入身份证(`ClaimsPrincipal`)和需要校验的权限进行处理。

```
public interface IPermissionChecker
{
    Task<bool> IsGrantedAsync([NotNull]string name);

    Task<bool> IsGrantedAsync([CanBeNull] ClaimsPrincipal claimsPrincipal,
[NotNull]string name);
}
```

第一个方法内部就是调用的第二个方法，只不过传递的身份证是通过 `ICurrentPrincipalAccessor` 拿到的，所以我们的核心还是看第二个方法的实现。

```
public virtual async Task<bool> IsGrantedAsync(ClaimsPrincipal claimsPrincipal, string
name)
{
    Check.NotNull(name, nameof(name));

    var permission = PermissionDefinitionManager.Get(name);

    var multiTenancySide = claimsPrincipal?.GetMultiTenancySide()
                           ?? CurrentTenant.GetMultiTenancySide();

    // 检查传入的权限是否允许当前的用户模式（租户/租主）进行访问。
    if (!permission.MultiTenancySide.HasFlag(multiTenancySide))
    {
        return false;
    }

    var isGranted = false;
    // 这里是重点哦，这个权限值检测上下文是之前没有说过的东西，说白了就是针对不同维度的权限检测。
    // 之前这部分东西是通过权限策略下面的 Requirement 提供的，这里 ABP vNext 将其抽象为
PermissionValueProvider。
    var context = new PermissionValueCheckContext(permission, claimsPrincipal);
    foreach (var provider in PermissionValueProviderManager.ValueProviders)
    {
        // 如果指定的权限允许的权限值提供者集合不包含当前的 Provider，则跳过处理。
        if (context.Permission.Providers.Any() &&
            !context.Permission.Providers.Contains(provider.Name))
        {
            continue;
        }

        // 调用 Provider 的检测方法，传入身份证明和权限定义进行具体校验。
        var result = await provider.CheckAsync(context);
```

```
        // 根据返回的结果，判断是否通过了权限校验。
        if (result == PermissionGrantResult.Granted)
        {
            isGranted = true;
        }
        else if (result == PermissionGrantResult.Prohibited)
        {
            return false;
        }
    }

    // 返回 true 说明已经授权，返回 false 说明是没有授权的。
    return isGranted;
}
```

2.2.6 权限值校验提供者

在模块配置方法内部，可以看到通过 `Configure<PermissionOptions>()` 方法添加了三个权限值校验提供者，即 `UserPermissionValueProvider` 、 `RolePermissionValueProvider` 、 `ClientPermissionValueProvider` 。在它们的内部实现，都是通过 `IPermissionStore` 从持久化存储 **检查传入的用户是否拥有某个权限**。

这里我们以 `UserPermissionValueProvider` 为例，来看看它的实现方法。

```
public class UserPermissionValueProvider : PermissionValueProvider
{
    // 提供者的名称。
    public const string ProviderName = "User";

    public override string Name => ProviderName;

    public UserPermissionValueProvider(IPermissionStore permissionStore)
        : base(permissionStore)
    {

    }

    public override async Task<PermissionGrantResult>
CheckAsync(PermissionValueCheckContext context)
    {
        // 从传入的 Principal 中查找 UserId，不存在则说明没有定义，视为未授权。
        var userId = context.Principal?.FindFirst(AbpClaimTypes.UserId)?.Value;

        if (userId == null)
        {
            return PermissionGrantResult.Undefined;
        }

        // 调用 IPermissionStore 从持久化存储中，检测指定权限在某个提供者下面是否已经被授予了权
限。
```

```
        // 如果被授予了权限， 则返回 true，没有则返回 false。
        return await PermissionStore.IsGrantedAsync(context.Permission.Name, Name,
userId)
            ? PermissionGrantResult.Granted
            : PermissionGrantResult.Undefined;
    }
}
```

这里我们先不讲 `IPermissionStore` 的具体实现，就上述代码来看，ABP vNext 是将权限定义放在了一个管理容器(`IPermissionDeftiionManager`)。然后又实现了自定义的策略处理器和策略，在处理器的内部又通过 `IPermissionChecker` 根据不同的 `PermissionValueProvider` 结合 `IPermissionStore` 实现了指定用户标识到权限的检测功能。

2.2.7 权限验证拦截器

权限验证拦截器的注册都是在 `AuthorizationInterceptorRegistrar` 的 `RegisterIfNeeded()` 方法内实现的，只要类型的任何一个方法标注了 `AuthorizeAttribute` 特性，就会被关联拦截器。

```
private static bool AnyMethodHasAuthorizeAttribute(Type implementationType)
{
    return implementationType
        .GetMethods(BindingFlags.Instance | BindingFlags.Public |
BindingFlags.NonPublic)
        .Any(HasAuthorizeAttribute);
}

private static bool HasAuthorizeAttribute(MemberInfo methodInfo)
{
    return methodInfo.IsDefined(typeof(AuthorizeAttribute), true);
}
```

拦截器和类型关联之后，会通过 `IMethodInvocationAuthorizationService` 的 `CheckAsync()` 方法校验调用者是否拥有指定权限。

```
public override async Task InterceptAsync(IAbpMethodInvocation invocation)
{
    // 防止重复检测。
    if (AbpCrossCuttingConcerns.IsApplied(invocation.TargetObject,
AbpCrossCuttingConcerns.Authorization))
    {
        await invocation.ProceedAsync();
        return;
    }

    // 将被调用的方法传入，验证是否允许访问。
    await AuthorizeAsync(invocation);
    await invocation.ProceedAsync();
}

protected virtual async Task AuthorizeAsync(IAbpMethodInvocation invocation)
{
    await _methodInvocationAuthorizationService.CheckAsync(
```

```
            new MethodInvocationAuthorizationContext(
                invocation.Method
            )
        );
    }
```

在具体的实现当中，首先检测方法是否标注了 `IAllowAnonymous` 特性，标注了则说明允许匿名访问，直接返回不做任何处理。否则就会从方法获取实现了 `IAuthorizeData` 接口的特性，从里面拿到 `Policy` 值，并通过 `IAuthorizationService` 进行验证。

```
protected async Task CheckAsync(IAuthorizeData authorizationAttribute)
{
    if (authorizationAttribute.Policy == null)
    {
        // 如果当前调用者没有进行认证，则抛出未登录的异常。
        if (!_currentUser.IsAuthenticated && !_currentClient.IsAuthenticated)
        {
            throw new AbpAuthorizationException("Authorization failed! User has not
logged in.");
        }
    }
    else
    {
        // 通过 IAuthorizationService 校验当前用户是否拥有 authorizationAttribute.Policy
权限。
        await _authorizationService.CheckAsync(authorizationAttribute.Policy);
    }
}
```

针对于 `IAuthorizationService` ，ABP vNext 还是提供了自己的实现 `AbpAuthorizationService` ，里面没有重写什么方法，而是提供了两个新的属性，这两个属性是为了方便实现 `AbpAuthorizationServiceExtensions` 提供的扩展方法，这里不再赘述。

三、总结

关于权限与验证部分我就先讲到这儿，后续文章我会更加详细地为大家分析 ABP vNext 是如何进行权限管理，又是如何将 ABP vNext 和 ASP.NET Identity 、IdentityServer4 进行集成的。