ASP.NET Core Identity 源码分析

Chaper1 : ASP.NET Core Identity 简介

第一章将会从一些简单的示例入手，从浅层面、多方面、以面向对象的视角展示ASP .NET Core Identity 的主要功能以及分析部分源码。

1.1 ASP .NET Core 是什么

ASP.NET Core 是一个跨平台的高性能开源Web框架，是由微软和社区开发的下一代ASP .NET。它是一个模块化框架，既可以Windows上的完整.NET Framework上运行，也可以在跨平台.NET Core上运行。

///介绍 .NET .NET CORE ASP.NET ASP.NET CORE

1.2 ASP .NET Core Identity 是什么

ASP.NET Core Identity 是将登录功能添加到 ASP.NET Core 应用的成员资格系统。它允许我们创建、读取、更新和删除账户。支持账号验证、身份验证、授权、恢复密码和SMS双因子身份验证，也可以使用外部登录提供程序。 支持的外部登录提供程序包括 Facebook、Google、Microsoft 帐户和 Twitter。

Identity 可以使用 SQL Server 数据库配置以存储用户名、密码和配置文件数据。 另外，还可以使用另一个永久性存储，例如 Azure 表存储。

ASP .NET Core Identity 官方默认提供的持久化库是Entity Framework 。

Entity Framework Core 是适用于 .NET 的新式对象数据库映射器。 它支持 LINQ 查询、更改跟踪、更新和架构迁移。 EF Core 适用于很多数据库，包括 SQL 数据库（本地和 Azure）、SQLite、MySQL、PostgreSQL 和 Azure Cosmos DB。

1.3 ASP .NET Core Identity 功能特性

·能够对用户资料很方便的扩展

·可以针对用户资料进行扩展。（是否使用邮件/电话号码认证等）

·持久化

·使用任意的关系型数据库，从sqllite到mysql、sqlserver等等，由Entity Framwork 支持

·角色机制

·提供角色机制，可以使用不同的角色来进行不同权限的限制，可以轻松的创建角色，向用户添加角色等。

·基于声明的认证模式（Claims Based Authentication）

·需要支持基于 Claims 的身份验证机制，其中用户身份是一组Claims，一组Claims可以比角色拥有更强的表现力。

·第三方社交登陆

·可以很方便的使用第三方登入，比如 Microsoft 账户，Facebook， Twitter，Google等，并且存储用户特定的数据。

chapter2 主要功能分析

本节的内容是选取主要功能进行需求建模和流程分析

identity的主要功能包括注册、登录、用户信息管理与存储、角色管理、基于声明的认证。下面选择**注册**、**登录**两个功能进行分析。

2.1 重要概念

要理解identity在整个用户管理系统的作用，首先要理解几个重要概念。

2.1.1 基于声明的认证

Claim（声明），是用于存储键值对的类。Claim中有两个属性，其中ClaimType为Key，ClaimValue为Value。

我们可以通俗地称Claim为“证件单元”，例如，居民身份证上的键值对“姓名：张三”，就是一个“证件单元”，也就是Claim。

ClaimIdentity，是用于存储“证件单元”和认证方式（AuthenticationType）的类。我们可以认为ClaimIdentity是“证件”，因为它是用来存储一个身份所拥有的“证件单元”的。

ClaimPrincipal，是代表用户的类，它内部存储某个用户所持有的所有“证件”。

Role（角色），角色包含一组Claim和该角色拥有的权限，与“证件”相对应，即持有某个“证件”的人拥有相应的角色。

“证件单元”构成“证件”，一个“证件”对应一个“角色”，持有“证件”的用户拥有相应的角色，而角色对应着一组资源权限，这几个概念构成了ASP.NET Core Identity中的主要身份管理机制。

2.1.2 认证与授权

认证 (Authentication)：识别来访者是谁、确认来访者的身份。

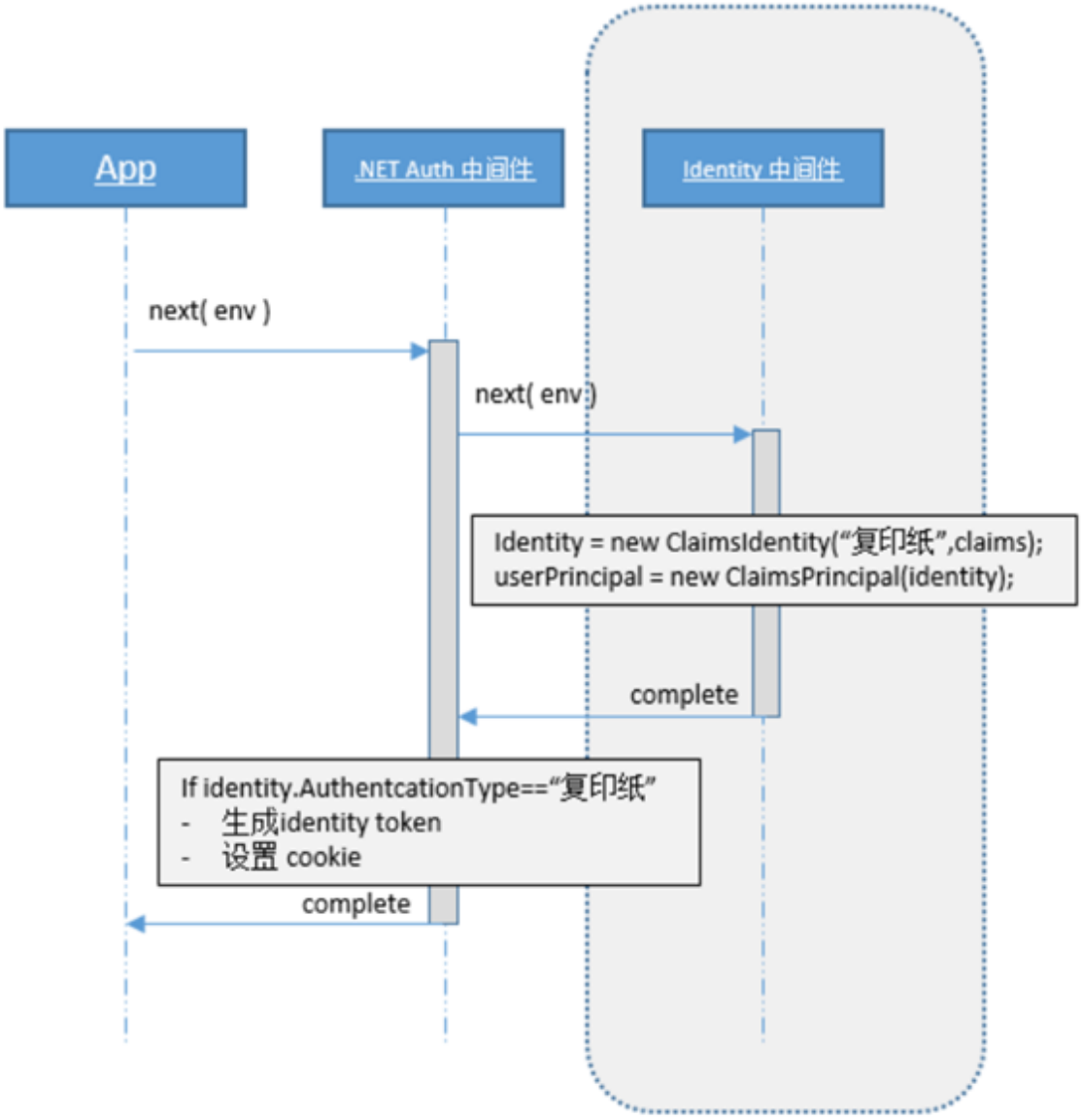
授权(Authorization)：确认**当前身份的来者**能不能访问当前请求的资源。

2.2 Identity的功能范围

需要特别注意的是，**认证**和**授权**与Identity负责的身份管理机制是独立的。Identity仅负责根据用户提供的信息生成Claim、将一组Claim组合成ClaimIdentity、以ClaimPrincipal的形式存储用户信息。

下面的示例图展示了Identity在登录中功能。

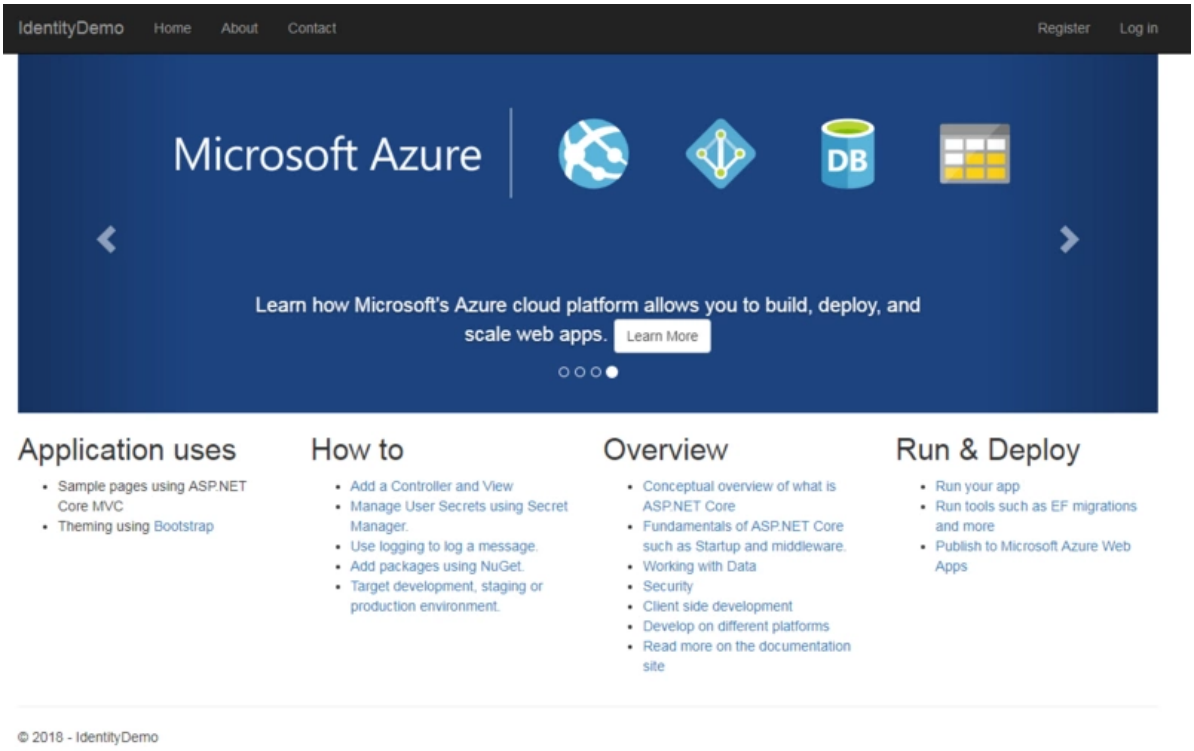
Identity仅负责Claim相关的逻辑，而具体这些信息如何在登录中被应用，这并不在Identity的功能范围内。



2.3 从创建新项目开始

在Visual Studio 2017的创建新项目面板依次选择 .net core -> asp.net core web 应用程序，选择 web 应用程序（模型视图控制器）->更改身份认证->个人用户账户。

运行新项目时，打开的界面如下图所示，在图片的右上角有Register和Login。



在我们选择**个人身份认证的时候** Identity被自动添加到项目中，并且生成了

- 账户控制器 *（`AccountController` 注册和登陆相关的代码）*
- 登陆注册页面 *（以及其他页面，如：确认邮件、访问受限等等）*
- 管理控制器 *（`ManageController` 注册用户相关逻辑，主要有两个功能，改密码和双因子验证）*

2.4 注册过程

打开刚才创建项目时自动生成的`AccountController.cs`文件，注册相关的代码如下图所示。

```
namespace IdentityDemo.Controllers
{
    public async Task<IActionResult> Register(RegisterViewModel model, string
returnUrl = null)
    {
        if (ModelState.IsValid)
        {
            //创建用户
            var user = new ApplicationUser {UserName = model.Email, Email =
model.Email};
            var result = await _userManager.CreateAsync(user, model.Password);
            //创建成功后的页面显示
            if (result.Succeeded)
            {
                var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
                var callbackUrl = Url.EmailConfirmationLink(user.Id, code,
Request.Scheme);
                await _emailSender.SendEmailConfirmationAsync(model.Email,
callbackUrl);

                await _signInManager.SignInAsync(user, isPersistent: false);
                return RedirectToLocal(returnUrl);
            }
            //创建失败时
            AddErrors(result);
```

```
        }
        // If we got this far, something failed, redisplay form
        return View(model);
    }
```

总结工作流程如下。

代码范围	类	属性，方法，或者属性的方法			工作	
我们的代码	AccountController	Register			从用户输入获取email、密码、确认密码	
					通过模型验证验证email合规性，密码复杂性，确认密码与密码是否一致	
					创建用户模型->ApplicationUser	
Identity	UserManager	CreateAsync			更新SecurityStamp	
			IUserValidator的默认实现UserValidator	ValidateUserName	验证用户名不是空白	
					验证用户名是否包含不允许的字符①	
					验证用户名是否已存在	
				如果要求邮件不能重复的话 ValidateEmail	验证邮件不是空白	
					验证邮件格式	
					验证邮件是否已经存在	
			如果支持用户锁定，并且允许新注册的用户被锁定		设置这个用户可以被锁定（即上一篇文章中提到的LockoutEnabled）	
			UpdateNormalizedUserNameAsync		更新规范化用户名	
			UpdateNormalizedEmailAsync		更新规范化的email	
			Store	CreateAsync	使用存储实现来存储用户数据	
我们的代码	AccountController	Register			生成邮件确认链接	这里比较简略因为不在这篇文章的讨论范围
					发送确认邮件	
					将用户登陆	

从工作流程可以看出，注册用户的功能主要由`UserManager`类负责，更具体的内容将在第二章进行讲解。

2.5 登录过程

登录过程与注册过程类似，主要代码在`AccountController.cs`的`public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)`方法中。

而实际的工作代码则只有一行。

```
var result = await _signInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, lockoutOnFailure: false);
```

因为登录过程需要另外两个模块HTTP、Security的配合，因此下面的表格只展示用户端代码和Identity代码的工作流程，而不是完整的登录流程。

代码范围	作用
我们的代码	从用户输入获取用户名、密码、记住我
identity	检查是否支持锁定用户以及此用户是否已被锁定
	检查用户密码是否正确
	如果支持锁定用户，并且支持在登陆失败超过指定次数锁定用户则增加AccessFailedCount计数，并且在到达设置的计数上限后清零计数设置LockoutEnd时间
	通过用户的基本信息生成Claims 及ClaimsIdentity
	添加额外的Claims
	生成ClaimsPrincipal
	添加认证方法Claim

从上面的表格我们注意到，Identity仅仅生成了用户信息和认证方法，至于登录所需要的认证和授权并不是Identity负责的。

实际上，认证过程是紧随在Identity工作内容之后的HTTP、Security模块完成的，而授权过程则是在用户请求资源时才发生。

chapter3 核心流程设计分析

本章我们将完成注册、登录过程在Identity中涉及的类和类间关系的分析。

在ASP.NET Core的框架中，类间的依赖关系广泛使用**依赖注入**(Dependency injection,**DI**)处理。在这当中，面向对象思想的核心思想被体现得淋漓尽致，包括高内聚低耦合、依赖倒置原则、里氏替换原则等。

3.1 什么是依赖注入

在ASP.NET Core的框架中，类间的依赖关系广泛使用**依赖注入**(Dependency injection,**DI**)处理。这是一种实现对象和依赖者之间松耦合的技术，将类用来执行其操作的这些对象以**注入**的方式提供给该类，而不是直接实例化依赖项或者使用静态引用。

当A类需要B类提供的方法来完成工作时，就产生了**依赖**。而根据依赖倒置原则，A类不应当依赖具体，而是应该依赖抽象。所以A类并不自己创建B类的实例而是由调用者通过接口参数传递。将依赖的创建委托给其他的类，并且由其他类通过传参的方法传递给自己，这就是**注入**。

在Identity设计中的DI服从显式依赖原则、依赖倒置原则，即类要求在它们构造时向构造函数提供抽象（通常是接口），而不是引用特定的实现，这种方法称为“构造函数注入”。

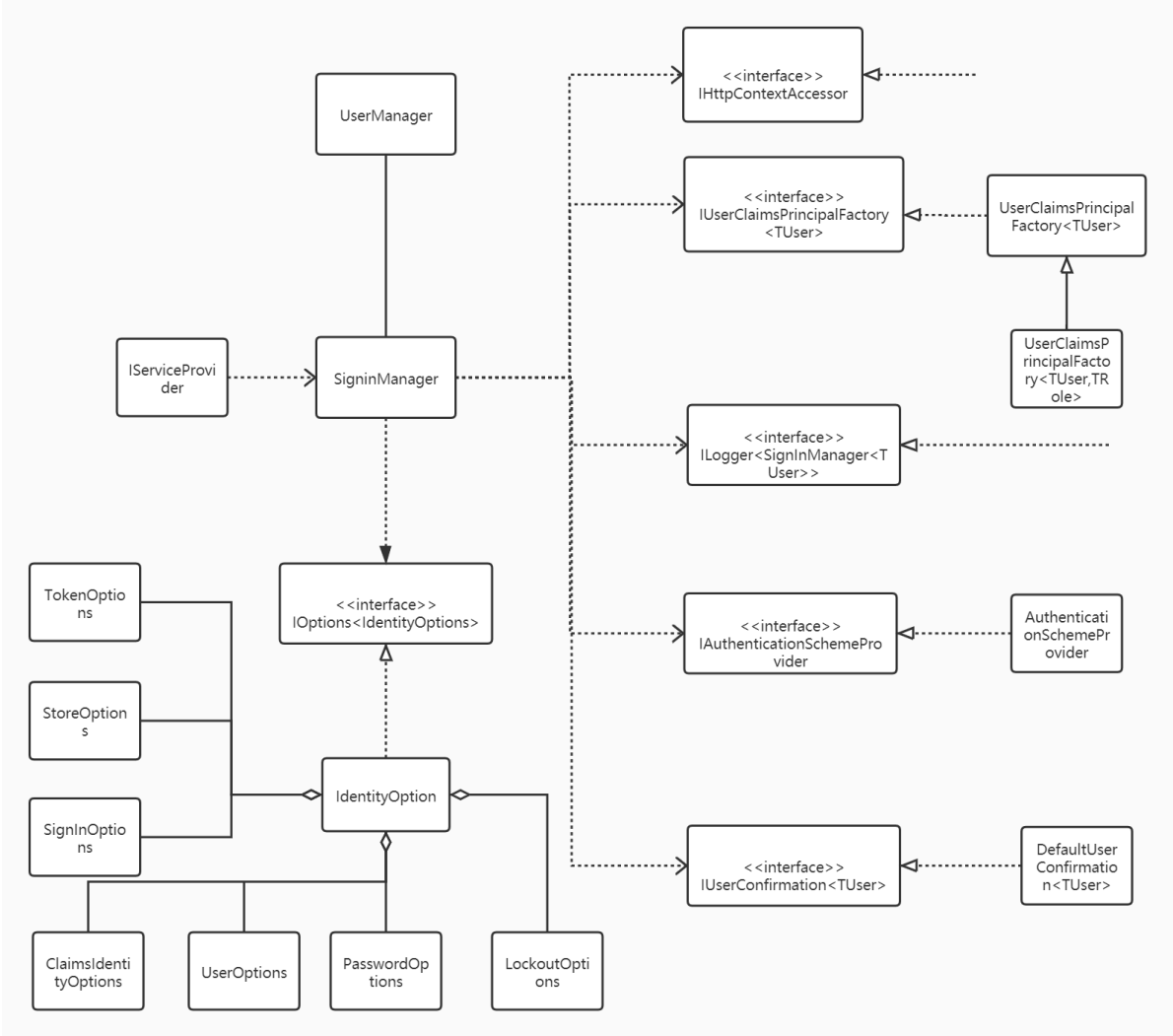
ASP.NET Core 提供的依赖注入（DI）的核心有两个组件：

- IServiceCollection 负责注册
- IServiceProvider 负责提供实例、绑定实例和服务的关系，是一个内置的服务容器，可以理解为统一的依赖管理中心，例如依赖B类有100处位置，这100处位置都从服务容器内获得实例。从另一个角度，容器本质上是一个工厂，负责提供向它请求的类型的实例。

当类的设计使用DI思想时，他们的耦合更加松散，因为他们没有对他们的合作者直接硬编码的依赖。这遵循“**依赖倒置原则**”，其中指出，高层模块不应该依赖于底层模块：两者都依赖于抽象。

3.2 类间关系总览

下图所示为登录过程涉及的类及类间关系，由于有些类不在本文探讨范围，故没有画出，同时，省略了UserManager类相关的关系。



从图中可以看出，SigninManager是登录过程的核心，与它的依赖项都是松耦合关系。SigninManager的构造方法遵循显式依赖原理，由调用者负责给出依赖项的抽象，服务容器提供依赖项的实例。

```
public SignInManager(UserManager<TUser> userManager,
            IHttpContextAccessor contextAccessor,
            IUserClaimsPrincipalFactory<TUser> claimsFactory,
            IOptions<IdentityOptions> optionsAccessor,
            ILogger<SignInManager<TUser>> logger,
            IAuthenticationSchemeProvider schemes,
            IUserConfirmation<TUser> confirmation)
```

SigninManager在登录过程中的功能与它的依赖项之间的关系如下表。

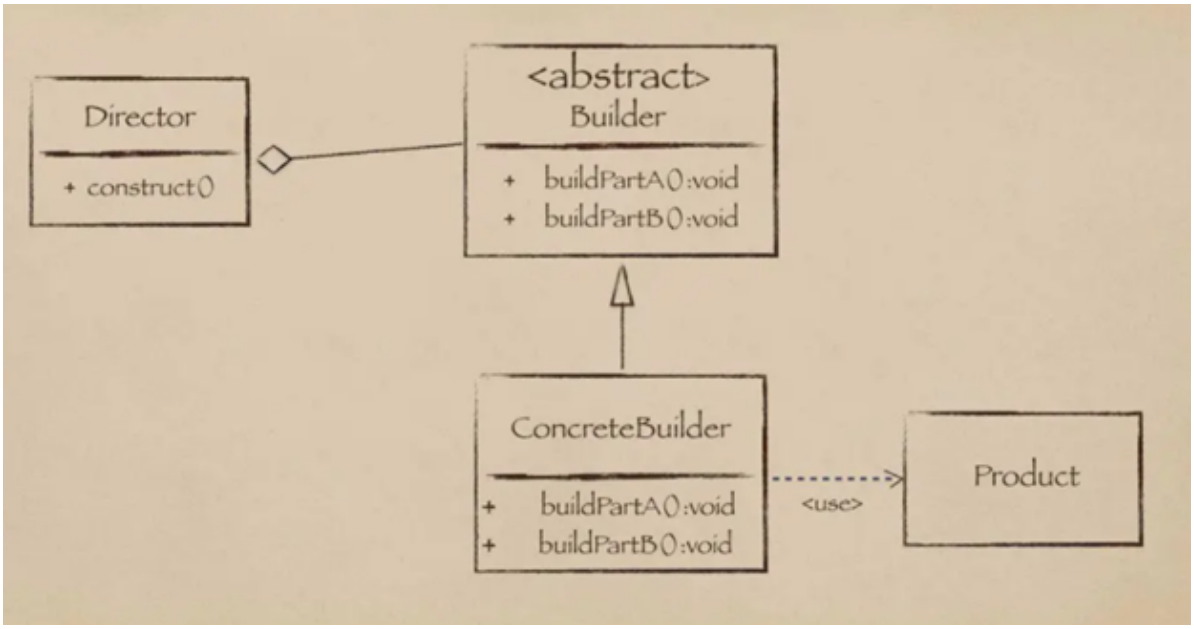
代码范围	作用
UserManager内的方法	检查是否支持锁定用户以及此用户是否已被锁定
UserManager内的方法	检查用户密码是否正确
ILogger提供的服务、UserManager内的方法	如果支持锁定用户，并且支持在登陆失败超过指定次数锁定用户则增加AccessFailedCount计数，并且在到达设置的计数上限后清零计数设置LockoutEnd时间
UserManager内的方法	通过用户的基本信息生成Claims 及ClaimsIdentity
UserManager内的方法	添加额外的Claims
IUserClaimsPrincipalFactory提供的服务	生成ClaimsPrincipal
IAuthenticationSchemeProvider提供的服务	添加认证方法Claim

chapter4 高级设计意图分析

4.1 Builder pattern

•设计模式中的Builder模式，又叫建造者模式

•主要的作用：分离一个复杂对象的**构建过程**和复杂对象的**表现形式**，抽象出构建过程，这样可以使用相同的构建过程，配合**依赖注入**构建出不同的产品。



4.1.1 简略版builder模式

identity系统所使用的builder模式与常规版本不同。

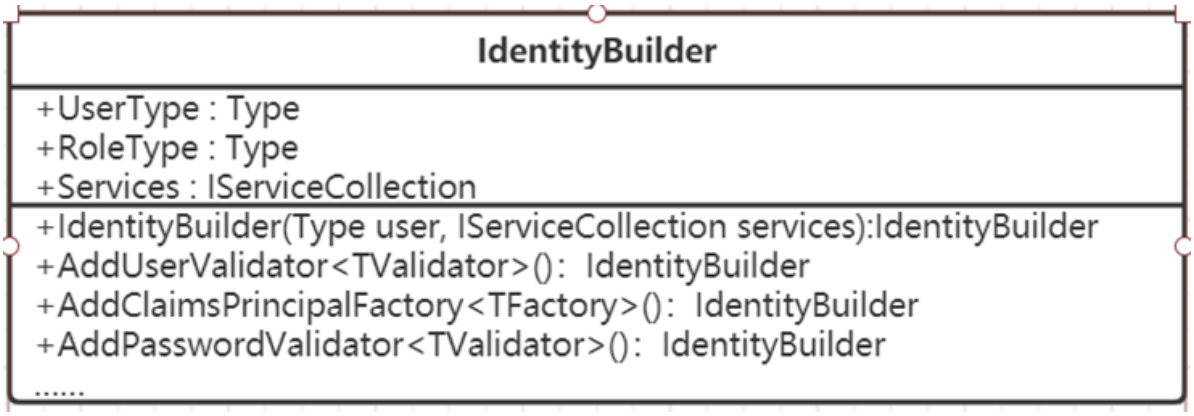
具体特点如下：

•没有Product类，只有builder类存储依赖关系，产品即为配置完成的identity系统

•没有director类，builder类的构建由**客户端**负责

•Builder类的创建和设值通过**依赖注入**完成

builder类的类图如下：



4.1.2 为什么使用builder模式?

首先，identity许多功能特性均为可配置的，如可以针对用户资料进行扩展、可以使用任意的关系型数据库、可以使用第三方登入等。每一个用户都可以搭建独特的identity系统，这就要求identity的构建过程支持个性配置，以便以同一个构建过程搭建出不同的产品。

其次，如果将构建过程完全交给用户，用户需要逐一配置依赖项，容易遗留、出错，对用户非常不友好。

4.1.3 为什么使用依赖注入来配合builder模式?

前面了解依赖注入的内涵之后，我们很自然地可以看出，使用依赖注入而非直接实例化依赖的优势:

- 可以轻松替换抽象的实现
- 方便进行单元测试
- 配置依赖项的代码简单、集中

4.1.4 具体实现

打开项目根目录下自动生成的`Startup.cs`文件，Identity相关的服务注册代码如下：

```
public class Startup
{
    //略...
    public void ConfigureServices(IServiceCollection services)
    {
        //略...
        services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultTokenProviders();
        //略...
    }
}
```

`AddIdentity`是在`Microsoft.Extensions.DependencyInjection`名字空间下`IdentityServiceCollectionExtensions`类内的方法。

这个方法负责将实现注入抽象，并创建`IdentityBuilder`的实例。

功能代码的形式如下：

`services.TryAddScoped<Interface, Implement>();`

`IUserValidator<TUser>`是抽象接口，而`UserValidator<TUser>`则是具体的实现类。

而`TryAddScoped`方法可以理解为如果检测到DI容器内已经有了当前要注册的Interface或Service就不会再次注册，没有才会注册进去。实际上这是优先使用用户的自定义实现，符合里氏替换原则。

4.2 策略模式

意图：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

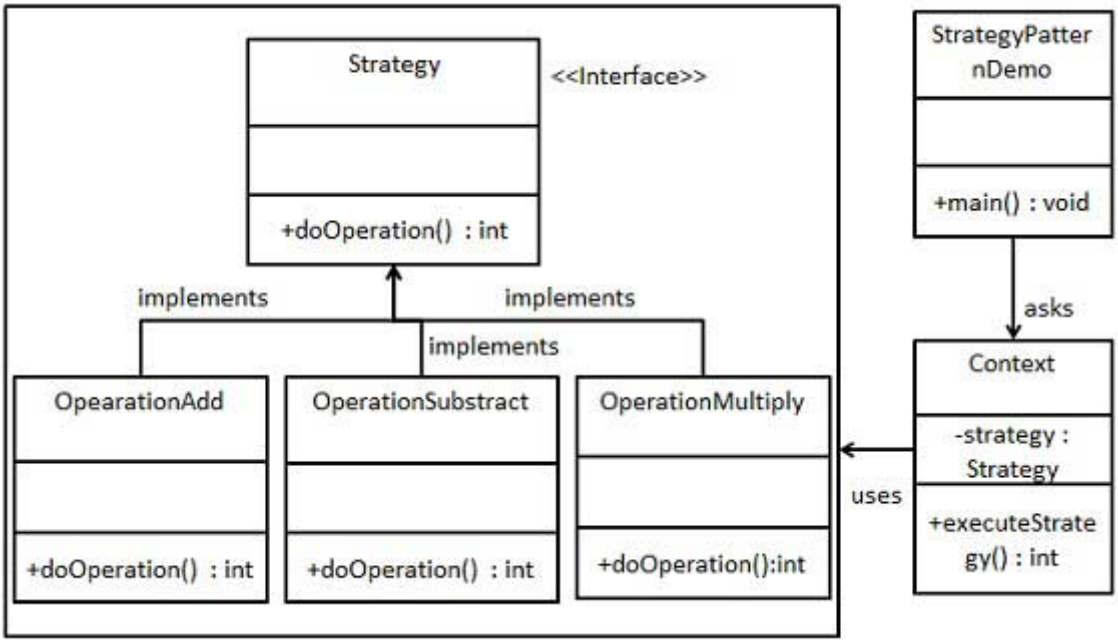
主要解决：在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。

何时使用：一个系统有许多许多类，而区分它们的只是他们直接的行为。

如何解决：将这些算法封装成一个一个的类，任意地替换。

关键代码：实现同一个接口。

示例类图关系如下。



4.2.1 高级设计意图

我们很容易在Identity的代码中找到Strategy模式。

因为依赖注入有两个特点：提取接口的依赖关系和提供接口的实现作为参数，而这显然又是策略模式的一个典型示例。

但我们需要特别注意DI和策略模式的共同点与区别：

DI和Strategy以相同的方式工作，但是Strategy用于更细粒度和短暂的依赖关系。

对象的依赖关系在它们的生命周期中发生变化，这在DI场景中通常不会发生，而在Strategy中却很常见。

策略专注于意图，而DI不仅管理抽象与实现，还负责提供实例、管理实例的生命周期。