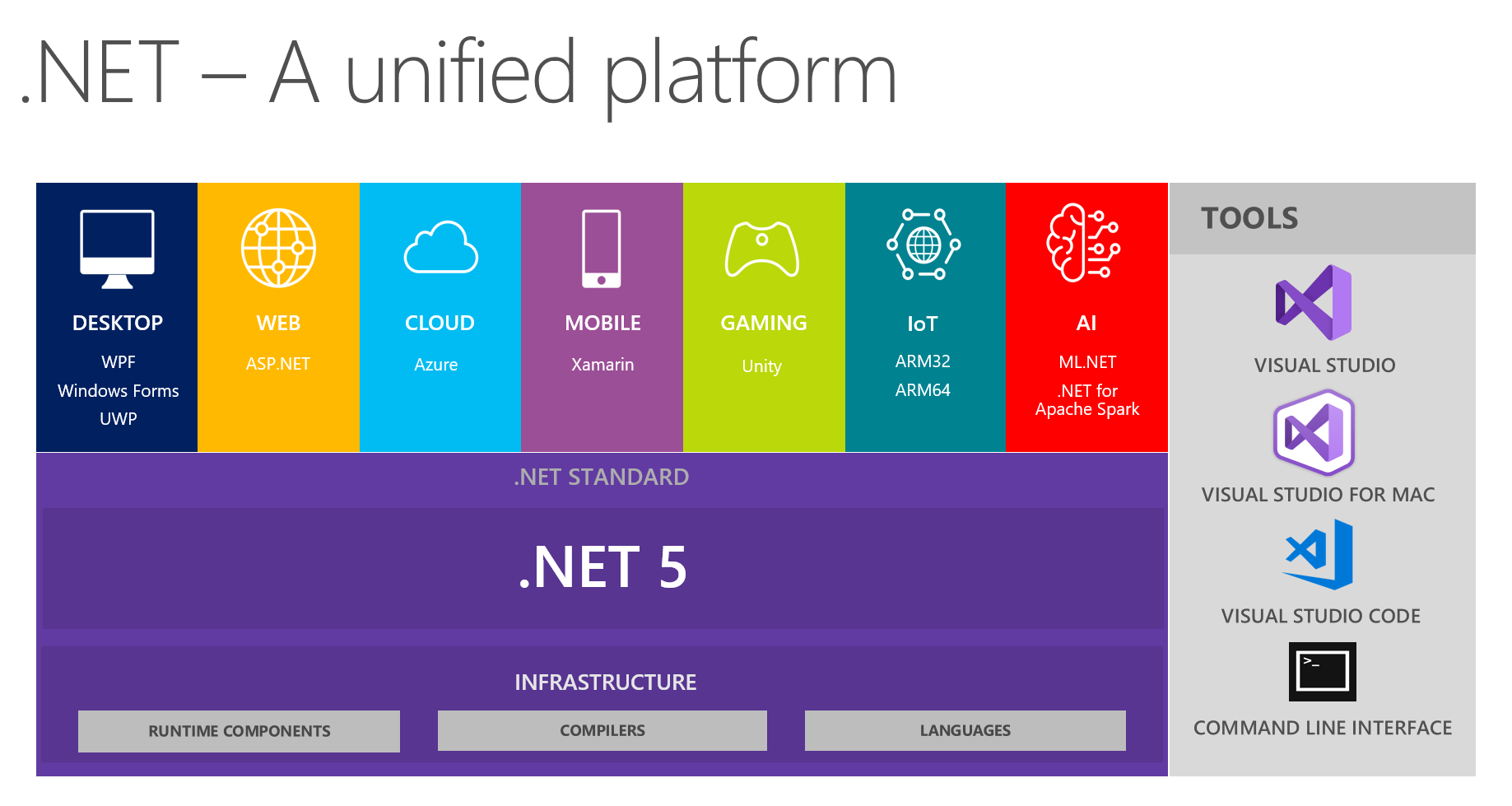
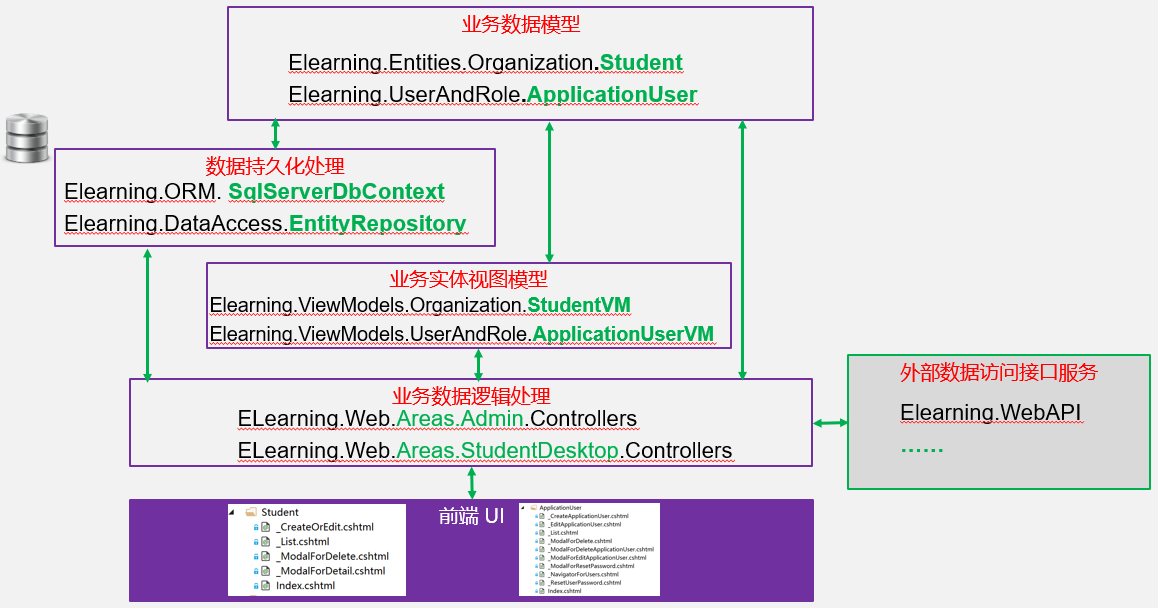
课程项目说明

# 软件架构

## .Net 平台技术栈



## 课程项目软件架构



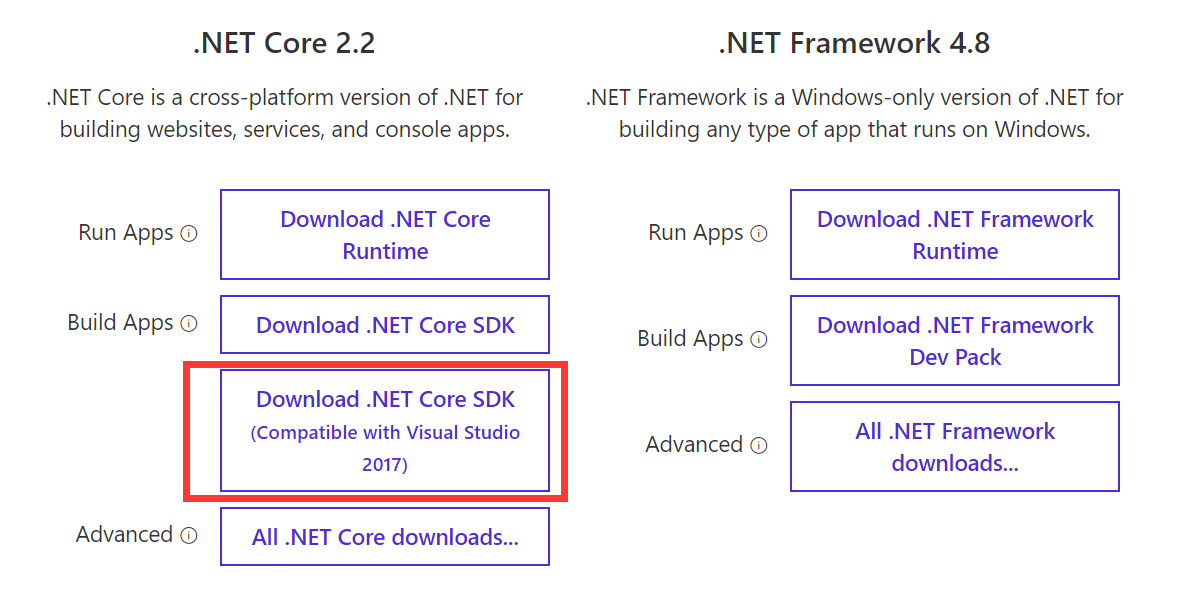
## 课程项目分层说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 层次 | 说明 | 项目类型 |
| 1 |  | **系统角色与用户模型**  自定义用户  自定义角色  角色类型 | **.net**  **standard类库** |
| 2 |  | **业务实体模型**  业务公用实体  业务实体约束接口 | **.net**  **standard类库** |
| 3 |  | **ORM**  业务与认证数据库上下文合二为一 | **.net**  **standard类库** |
| 4 |  | **数据持久化处理**  种子数据  业务实体访问泛型接口  业务实体访问实现泛型类 | **.net**  **standard类库** |
|  |  | **业务实体视图模型**  视图模型接口  视图模型抽象工厂类  视图模型实现基类  各业务视图模型类  公有模型  控制模型 | **.net**  **standard类库** |
|  |  | 前端UI层 | **asp.net core**  **应用程序**  **MVC** |

## 开发及运行环境

https://dotnet.microsoft.com/download

ASP.NET Core 2.0

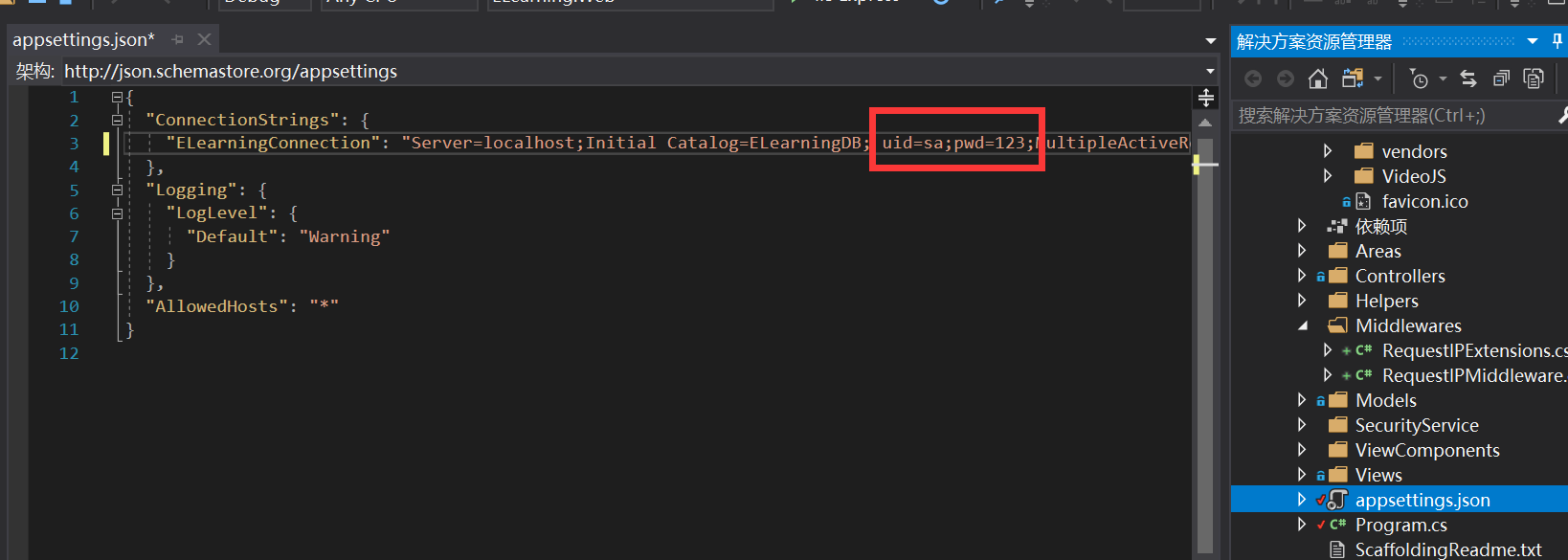


## 数据库迁移

参考资料

<https://docs.microsoft.com/zh-cn/ef/core/miscellaneous/cli/powershell>

（1）修改数据库连接字符串



（2）迁移命令参考

其中，红色部分是迁移名称，新迁移时修改名称

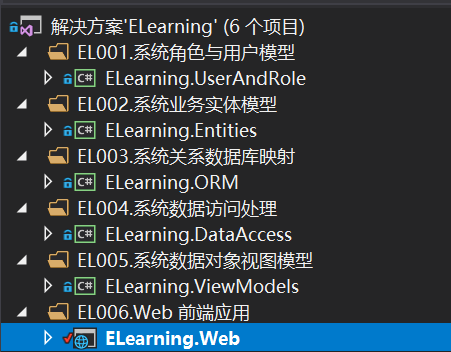
|  |
| --- |
| Add-Migration -Name "ELearning001" -Project "ELearning.ORM" -StartupProject "ELearning.Web"  Update-Database -Migration "ELearning001" -Project "ELearning.ORM" -StartupProject "ELearning.Web" |

# 项目构建说明

## 搭建框架

1. 创建空解决方案
2. 创建解决方案文件夹
3. 创建各层.net standard类库
4. 创建asp.net core Web MVC项目

效果如下图：

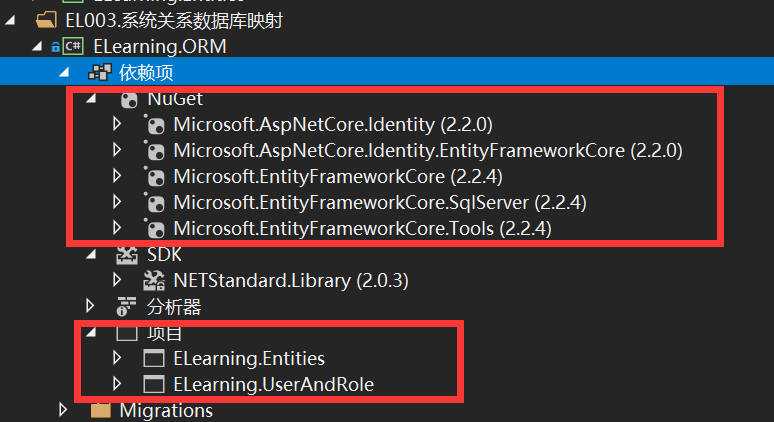


## 添加依赖项

为各项目添加依赖项

1. 添加项目引用
2. 添加NuGet包

效果如下图：



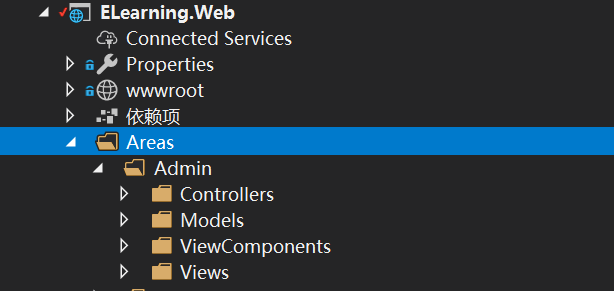
## 业务分区

参考

<https://docs.microsoft.com/zh-cn/aspnet/core/mvc/controllers/areas?view=aspnetcore-2.2>

区域用于将相关功能以单独的名称空间（用于路由）和文件夹结构（用于视图）的形式组织到一个组中。 带区域的应用路由为area\controller\action。

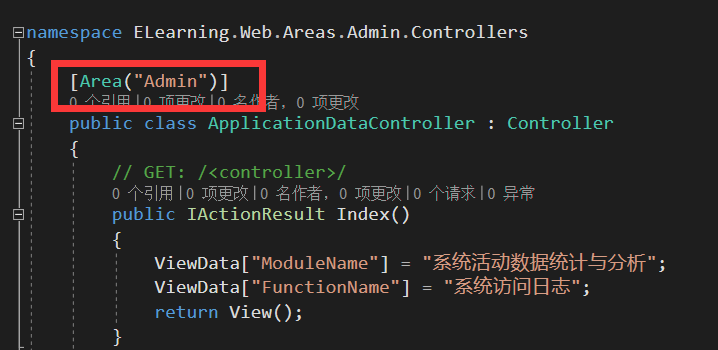
### 视图文件夹结构



### 使用区域属性将控制器与区域关联

|  |
| --- |
| [Area("区域名称")]  public class 控制器名: Controller  {…… } |

例

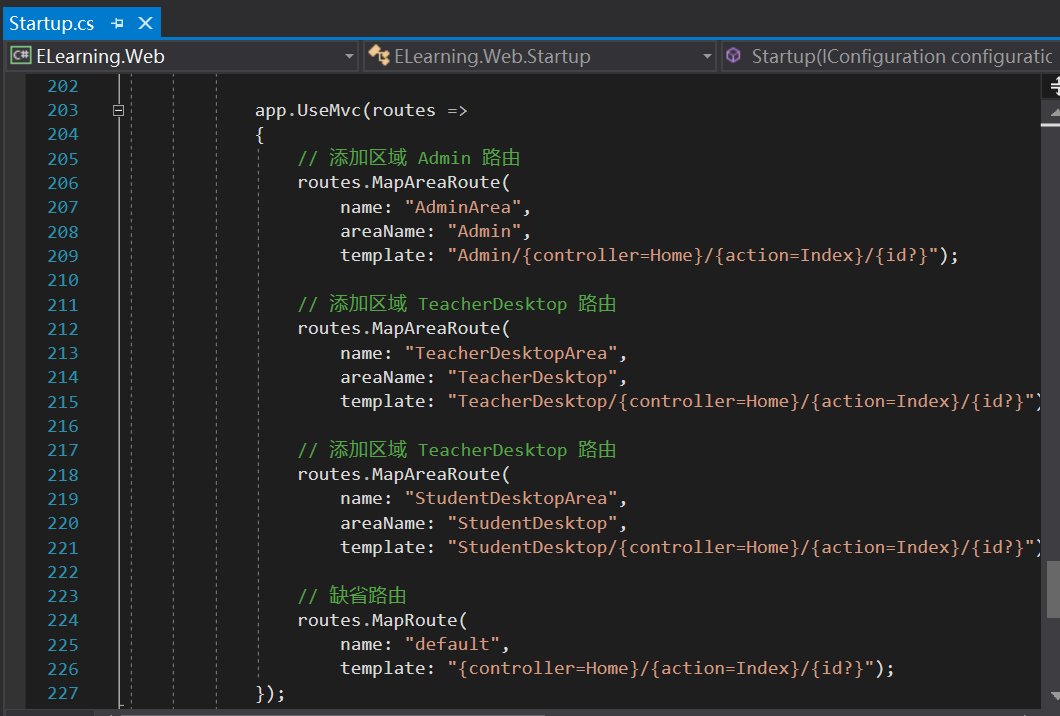


### 添加区域路由

在startup类Configure方法内，app.UseMvc中添加区域路由

|  |
| --- |
| app.UseMvc(routes =>  {  routes.MapRoute(  name: "MyArea",  template: "{area:exists}/{controller=Home}/{action=Index}/{id?}");  routes.MapRoute(  name: "default",  template: "{controller=Home}/{action=Index}/{id?}");  }); |

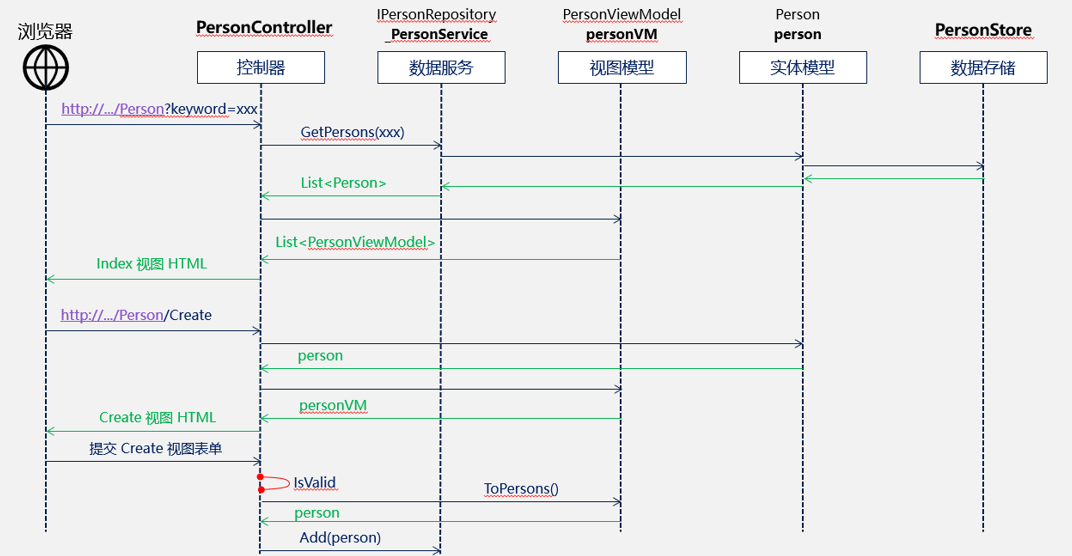
例子：



## 业务实现

根据迭代开发计划，项目引用顺序实现各层级类库相关类/方法及单元测试

业务数据访问过程参考



1. 实现认证与授权

根据业务需求构建用户与角色认证及授权

参考认证与授权.doc

## 前端编程

用NPM安装前端库

参考：

<https://www.yuque.com/yuejiangliu/dotnet/zlrv9a#75501e51>

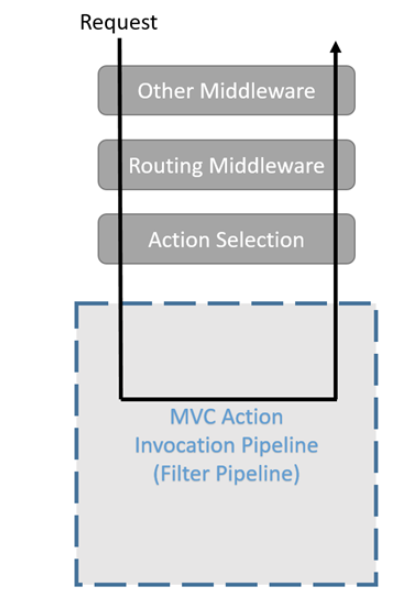
# 认证与授权

## 什么是认证和授权

**认证负责验证用户是谁**。**授权负责检查当前用户是否有权限访问资源**。授权方式包括对角色角色，对具体资源访问以及操作的授权

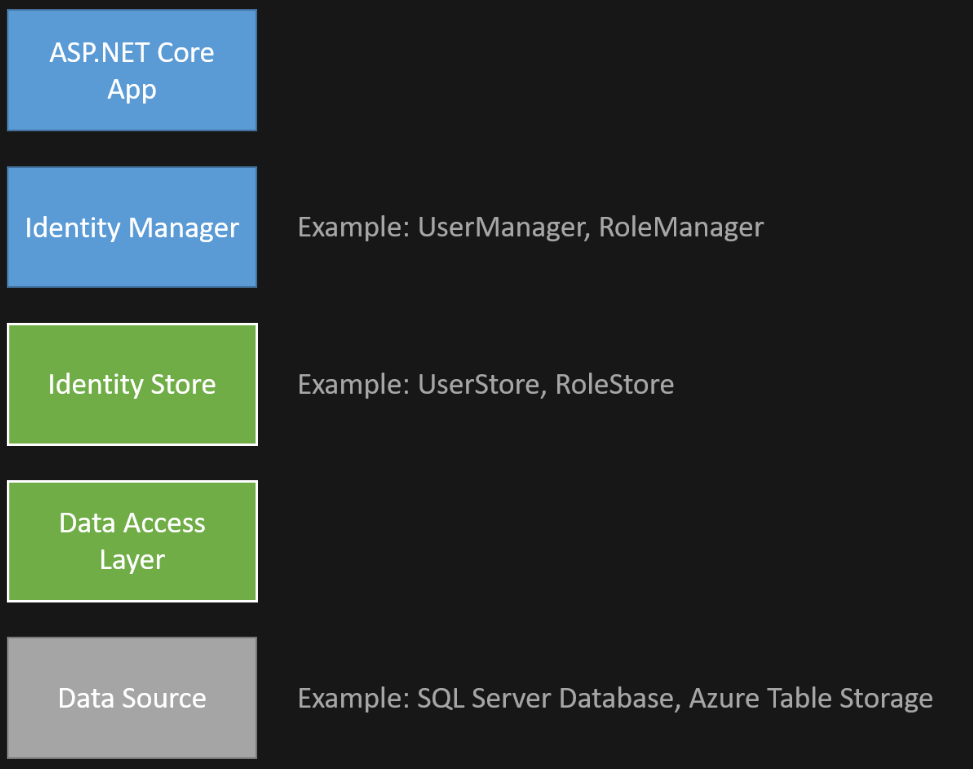
在ASP.NET Core中，MVC以中间件的形式加入到整个HTTP管道。在MVC在此之前有路由中间件。MVCRouteHandler被路由中间件加载出来处理请求。在MVC路由之前加入认证中间件可处理认证

如果我们要在MVC Middleware执行之请拦截请求只要在MVC中间件之前加入一个中间件即可。如验证中间件app.UseAuthentication()



## ASP.NET Core认证体系结构

ASP.NET Core标识包含类称为管理器和存储区。 管理器用于创建认证用户、角色等。 存储是指定如何保持实体，例如用户和角色。 存储和管理分离，通过依赖注入配置服务



### Identity模型

参考：

<https://docs.microsoft.com/zh-cn/aspnet/core/security/authentication/customize-identity-model?view=aspnetcore-2.2#the-identity-model>

1. 实体

|  |  |
| --- | --- |
| 实体类型 | 描述 |
| User | 表示的用户。 |
| UserRole | 联接实体相关联的用户和角色。 |
| UserClaim | 表示一个用户拥有的声明。 |
| UserToken | 表示用户的身份验证令牌。 |
| UserLogin | 将用户与登录名相关联。 |
| Role | 表示一个角色。 |
| RoleClaim | 表示一个角色内的所有用户授予的声明。 |

1. 实体间的关系

每个User都可以具有许多UserClaims。

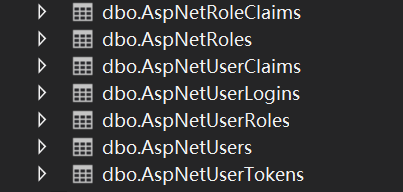
每个User都可以具有许多关联Roles，和每个Role可与许多关联Users。 这是需要在数据库中的联接表的多对多关系。 联接表由UserRole实体。

每个User都可以具有许多UserLogins。

每个User都可以具有许多UserTokens。

每个Role都可以具有许多关联RoleClaims。

1. 对应数据库中的表



1. 默认实体类

IdentityUser

IdentityRole

IdentityUserClaim

IdentityUserToken

IdentityUserLogin

IdentityRoleClaim

IdentityUserRole

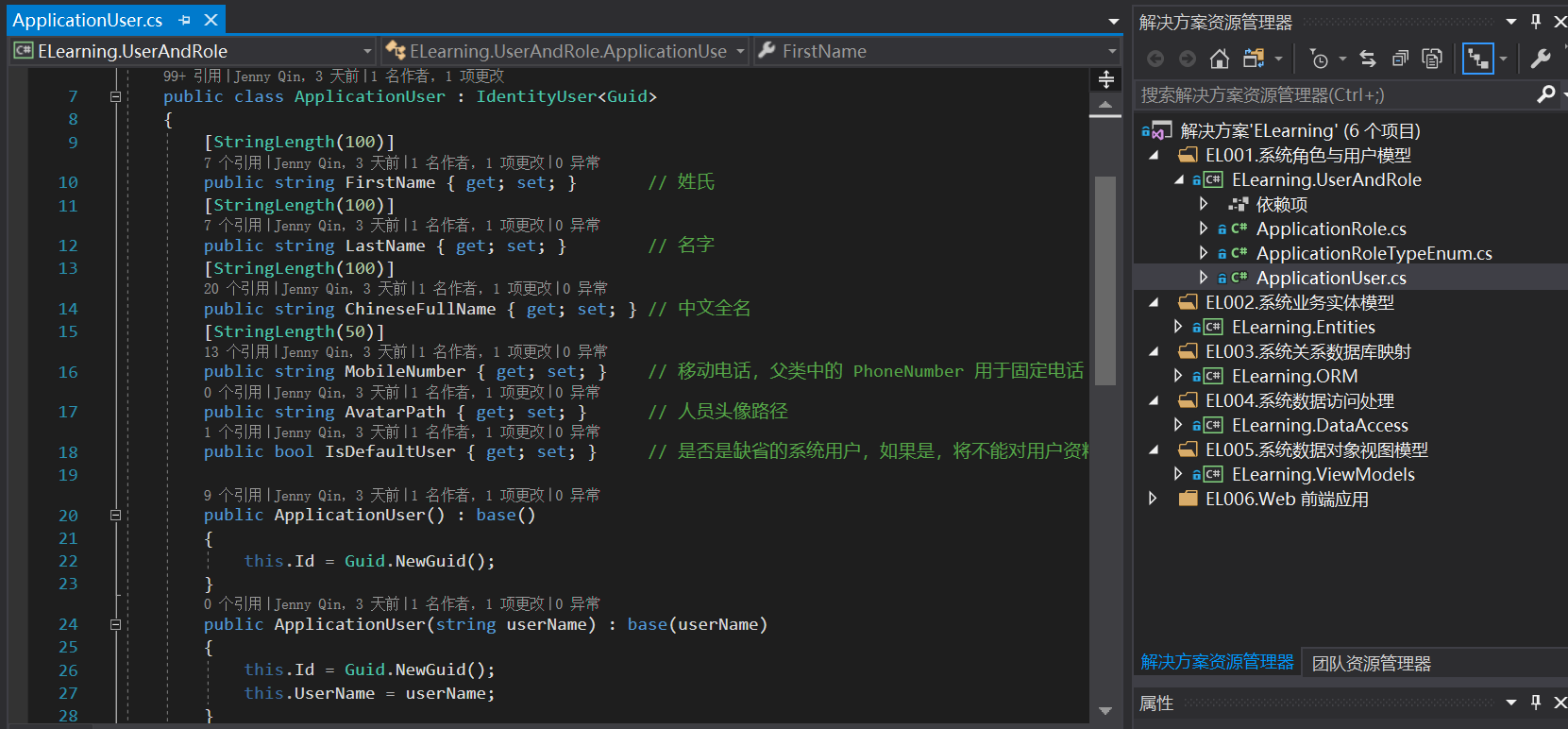
**使用时可以更改主键类型，可以使用无角色形式或有角色形式等。默认实体类通常作为实际应用的基类。**

#### 自定义模型（更改主键类型）

自定义用户

|  |
| --- |
| using System;  using Microsoft.AspNetCore.Identity;  public class **ApplicationUser** : **IdentityUser**<Guid>  {  public string CustomTag { get; set; }//自定义用户新增属性  } |

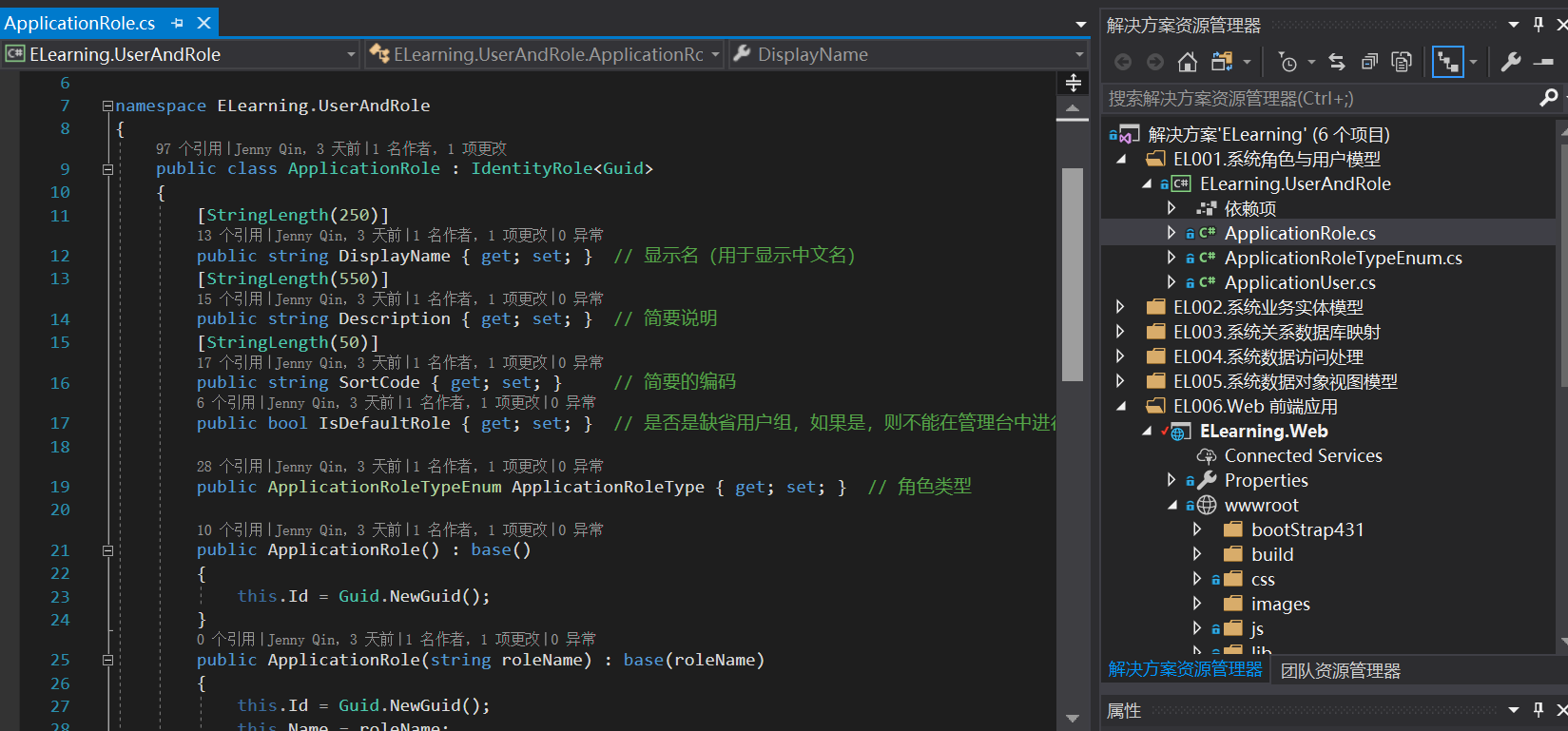
例



#### 自定义角色

|  |
| --- |
| using System;  using Microsoft.AspNetCore.Identity;  public class **ApplicationRole** : **IdentityRole**<Guid>  {  public string Description { get; set; }  } |

例子：



## 设置数据库上下文

|  |
| --- |
| public class ApplicationDbContext  **: IdentityDbContext<ApplicationUser, IdentityRole<Guid>, Guid>**  {  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)  : base(options)  {  }  } |

## 注册 Identity 服务

Startup.ConfigureServices

将上下文添加到identity

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  //……  **services.AddDbContext<ApplicationDbContext>(options =>**  **options.UseSqlServer(**  **Configuration.GetConnectionString("DefaultConnection")));**  **services.AddIdentity<ApplicationUser, ApplicationRole>()**  **.AddEntityFrameworkStores<ApplicationDbContext>()**  **.AddDefaultTokenProviders();**  } |

## 加入认证中间件

Startup.Configure

|  |
| --- |
| public void Configure(){  ……  **app.UseAuthentication();**  app.UseMvc（）；  ……  } |

## 用户与角色管理

类：

（1）RoleManager—角色CRUD等

（2）UserManager—用户CRUD等

一个用户可以属于不同的角色，一个角色可以拥有不同用户

在控制器/自定义业务类中通过在构造函数调用角色管理和用户管理实例

|  |
| --- |
| public class ApplicationUserVMService  {  private readonly UserManager<ApplicationUser> \_userManager;  private readonly RoleManager<ApplicationRole> \_roleManager;  public ApplicationRoleVMService(  UserManager<ApplicationUser> userManager,  RoleManager<ApplicationRole> roleManager  )  {  \_userManager = userManager;  \_roleManager = roleManager;  }  //在各方法中根据需要调用\_userManager或\_roleManager  } |

如：创建角色

|  |
| --- |
| \_roleManager.CreateAsync(角色) |

如：创建用户

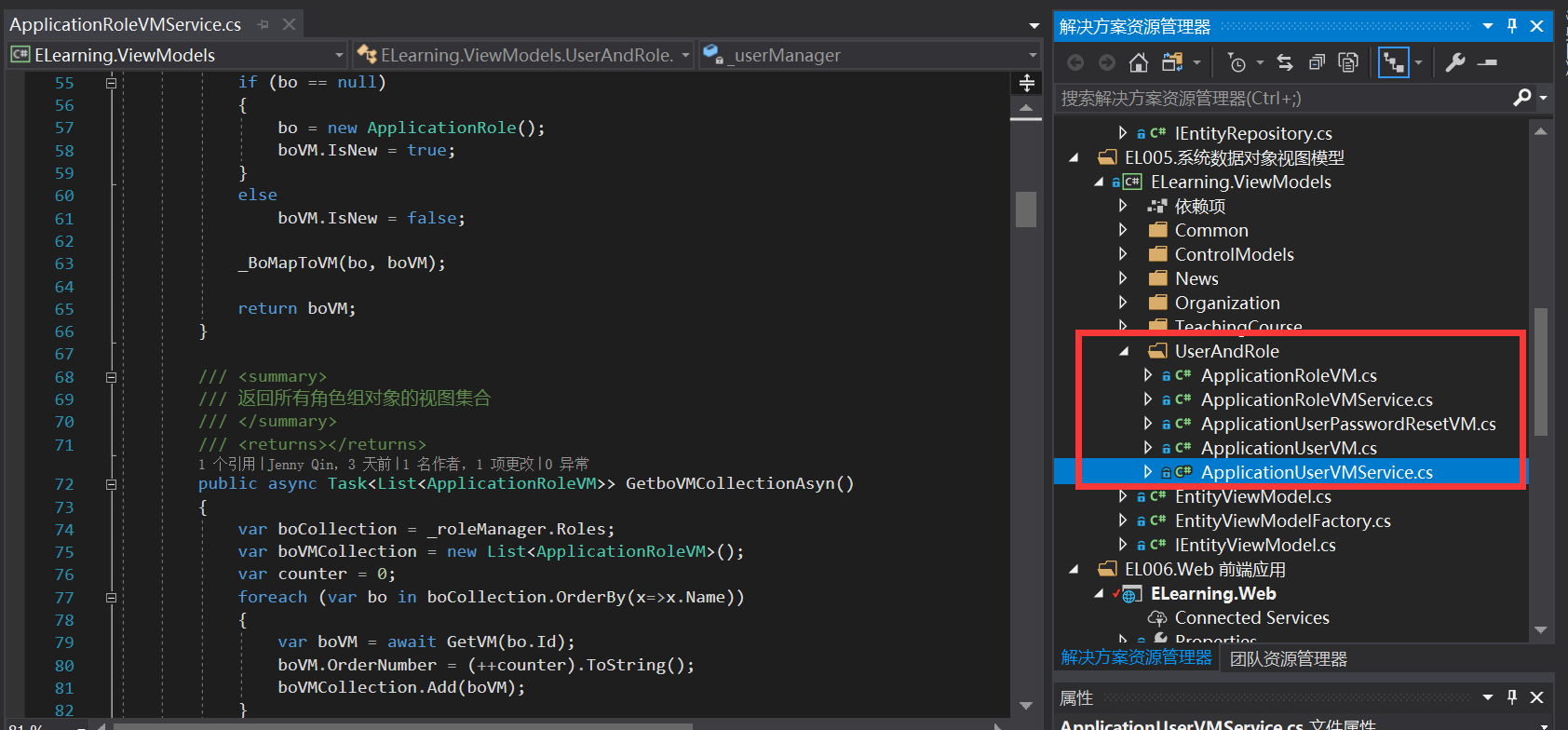
|  |
| --- |
| \_userManager. CreateAsync(用户，密码); |

如：为用户指定角色

|  |
| --- |
| \_userManager.AddToRoleAsync(用户,角色); |

## 认证管理业务类

在本项目中使用用户与角色业务类调用认证体系IdentityManager实现用户和角色管理



## 授权Authorization

参考文档

<https://docs.microsoft.com/zh-cn/aspnet/core/security/authorization/introduction?view=aspnetcore-2.1>

微软授权实验室源码

<https://github.com/blowdart/AspNetAuthorizationWorkshop>

### 简单授权

验证用户是否通过认证

#### 控制器级授权

例如

|  |
| --- |
| **[Authorize]**  public class AccountController : Controller  {  public ActionResult Login()  {  }  public ActionResult Logout()  {  }  } |

#### 行为级授权

|  |
| --- |
| public class AccountController : Controller  {  public ActionResult Login()  {  }  **[Authorize]**  public ActionResult Logout()  {  }  } |

#### 在控制器级授权中使用匿名授权

|  |
| --- |
| **[Authorize]**  public class AccountController : Controller  {  **[AllowAnonymous]**  public ActionResult Login()  {  }  public ActionResult Logout()  {  }  } |

### 基于角色的授权

通过指明 AuthorizeAttribute 的角色参数，限制认证用户归属角色访问控制器或者Action的授权；或者通过配置访问策略的方式来添加进行授权，限制访问控制器或者Action。

#### 对角色进行简单授权

（前提，已在数据库中创建Administrator和PowerUser角色，并将某些用户指定为Administrator，PowerUser角色

例1，单一角色限制

|  |
| --- |
| [Authorize(Roles = "Administrator")]  public class AdministrationController : Controller  {  } |

例2，多角色限制，逗号间隔

|  |
| --- |
| [Authorize(Roles = "Administrator, PowerUser")]  public class ControlPanelController : Controller  {  public ActionResult SetTime()  {  }  **[Authorize(Roles = "Administrator")]**  public ActionResult ShutDown()  {  }  } |

#### 对既定角色应用策略授权

（1）在Startup. ConfigureServices方法中添加授权策略

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddMvc();  ……  services.AddAuthorization(options =>  {  options.AddPolicy("RequireAdministratorRole",  policy => policy.RequireRole("Administrator"));  });  } |

（2）使用应用策略Policy属性上的AuthorizeAttribute属性授权

|  |
| --- |
| [Authorize(Policy = "RequireAdministratorRole")]  public IActionResult Shutdown()  {  return View();  } |

### 基于声明claims的授权

1. 创建用户时创建声明

### 基于策略policy的授权

1. 注册策略

授权策略包含一个或多个要求。 授权策略包含一个或多个要求，并在 Startup.ConfigureServices 方法中作为授权服务配置的一部分注册：

例创建了一个“AtLeast21”策略 年龄至少需要21岁

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  ……  services.AddAuthorization(options =>  {  options.AddPolicy("**AtLeast21**", policy =>  policy.**Requirements.Add(new MinimumAgeRequirement(21)**));  });  ……  } |

1. 将策略应用到 MVC 控制器

例AlcoholPurchaseController控制器受控于策略**AtLeast21**

|  |
| --- |
| using Microsoft.AspNetCore.Authorization;  using Microsoft.AspNetCore.Mvc;  [Authorize(**Policy = "AtLeast21**")]  public class AlcoholPurchaseController : Controller  {  public IActionResult Index() => View();  } |

1. 创建授权要求类

授权要求是一个可供策略用来评估当前用户主体的数据参数的集合。 在 “AtLeast21”策略中，此要求是单个参数—最低年龄。 要求可以实现 IAuthorizationRequirement，这是一个空标记接口。 参数化的最低年龄要求可以按如下方式实现：

|  |
| --- |
| using Microsoft.AspNetCore.Authorization;  public class **MinimumAgeRequirement** : **IAuthorizationRequirement**  {  //授权要求在创建实例时最小年龄属性获得值  **public int MinimumAge { get; }**  public MinimumAgeRequirement(int minimumAge)  {  MinimumAge = minimumAge;  }  } |

1. 创建授权要求处理程序

例

|  |
| --- |
| using System;  using System.Security.Claims;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Authorization;  using PoliciesAuthApp1.Services.Requirements;  public class MinimumAgeHandler : AuthorizationHandler<**MinimumAgeRequirement**>  {    protected override Task HandleRequirementAsync  (AuthorizationHandlerContext context,  **MinimumAgeRequirement** requirement)  {  //若声明中没有生日类型，则结束任务  if (!context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth &&  c.Issuer == "http://contoso.com"))  {  return Task.CompletedTask;  }  //从上下文中获取生日声明值  var dateOfBirth = Convert.ToDateTime(  context.User.FindFirst(c => c.Type == ClaimTypes.DateOfBirth &&  c.Issuer == "http://contoso.com").Value);  //计算年龄  int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;  if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))  {  calculatedAge--;  }  //符合授权需求年龄则，对应授权上下文置入成功  if (calculatedAge >= **requirement.MinimumAge**)  {  context.Succeed(requirement);  }  //任务结束  return Task.CompletedTask;  }  } |

1. 注册授权要求处理程序

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);  services.AddAuthorization(options =>  {  options.AddPolicy("AtLeast21", policy =>  policy.Requirements.Add(new MinimumAgeRequirement(21)));  });  // **AddSingleton服务在整个**  **services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();**  **}** |