本文主要介绍在本移动端项目当中相对重要的一些代码的业务实现方法, 主要包含以下内容 :

1.启动配置

在本项目中, 除了特定平台的启动配置 (如Syncfusion组件在iOS当中需要显示声明每个Renderer组件的初始化),

Android位于Xamarin.Android项目当中, iOS位于Xamarin.iOS当中。下文将主要介绍共享项目当中的初始化过程。

1.注册容器服务

本移动端项目中, 使用了Prism.Forms框架作为应用程序的核心, 与桌面应用程序相同, 应用程序首先会向容器当中注册依赖的服务, 其中包括(Web服务、View/ViewModel模块、实体验证器、导航服务)。还有一些容器的扩展, 其中包括(AutoMapper)。如下所示:

```
        protected override void RegisterTypes(IContainerRegistry
containerRegistry)
        {
            //注册应用程序服务
            containerRegistry.ConfigureSerivces();

            //注册Prism区域服务及导航页
            containerRegistry.RegisterRegionServices();
            containerRegistry.RegisterForNavigation<NavigationPage>();
        }
```

2.应用程序初始化

容器服务以及扩展注册完成后, 接下来就是应用程序初始化过程, 该过程主要设置应用的系统配置, 如(Syncfusion 授权、主题颜色设置)。如下所示:

```
            //设置Syncfusion授权

 SyncfusionLicenseProvider.RegisterLicense(AppConsts.SyncfusionLicenseKey);

            //设置系统主题
            ApplyThemes();
```

紧接着, 应用程序会直接显示初始化屏幕, 该过程主要通过Prism.Forms的导航服务实现。

备注: 在这一步, 也可以自行添加启动应用程序欢迎页, 如下一步等。

```
await NavigationService.NavigateAsync(AppViewManager.InitialScreen);
```

应用程序将直接打开 AppViewManager.InitialScreen 名称所定义的页面, 该页面是由一个ContentPage组成。

备注: 在Xamarin.Forms当中, 对于继承于Page类的实现, 都具备 Appearing 事件, 该事件主要用于当页面显示至屏幕时触发, 可以用于处理一些数据加载动作。

在本项目当中, 则用于加载一些本地的缓存数据, 包括授权信息、应用程序资源数据(多语言、应用程序本地化配置、多租户配置、默认显示语言等)。当应用程序的授权有效, 则直接导航显示至应用程序首页, 否则导航至登录页面。InitialScreenViewModel 当中Appearing方法定义如下:

```
        private async void Appearing()
        {
            //加载本地的缓存信息
            accessTokenManager.AuthenticateResult =
dataStorageService.RetrieveAuthenticateResult();
            applicationContext.Load(dataStorageService.RetrieveTenantInfo(),
dataStorageService.RetrieveLoginInfo());

            //获取应用程序资源数据(本地化资源、设置、用户信息权限等...)
            await AppConfigurationManager.GetIfNeedsAsync();

            if (accessTokenManager.IsUserLoggedIn)
            {
                await navigationService.NavigateAsync(AppViewManager.Main);
            }
            else
            {
                await navigationService.NavigateAsync(AppViewManager.Login);
            }
        }
```

3.登录页

当应用程序第一次启动或者未保存任何本地的用户授权信息时(注销用户)，将导航至登录页面, 加载登录页数据, 包括(可选语言列表、租户选项)及用户信息(存在时)。

4.首页

当应用程登录成功或存在缓存的授权信息时, 将导航至首页，加载首页数据，包括(应用程序信息、用户资料、功能模块列表) 及设置首页的默认页。

2.Web服务及授权

AppFramework.Application.Client 项目主要是负责提供Web服务的中间层, 它负责移动端到WebApi之间的数据交换。其中包含了每个模块的核心服务, 诸如: 用户、角色、组织机构、版本信息、多租户、缓存、语言列表等功能。

1.服务配置 (ApiUrlConfig)

该文件类主要包含用于配置主机服务地址配置、以及用于转换URL的方法。

注意: 主机服务地址需要根据实际情况进行修改。

2.授权 (AccessTokenManager)

AppFramework.Application.Client 项目当中的 AccessTokenManager 服务包含了用于登录身份授权、注销、刷新密钥功能。

3.Web服务 (ApiClient)

ApiClient 中包含了Api的配置相关以及封装了用于HTTP通信的常用方法, 如: Get、Post、Put、Delete 等。

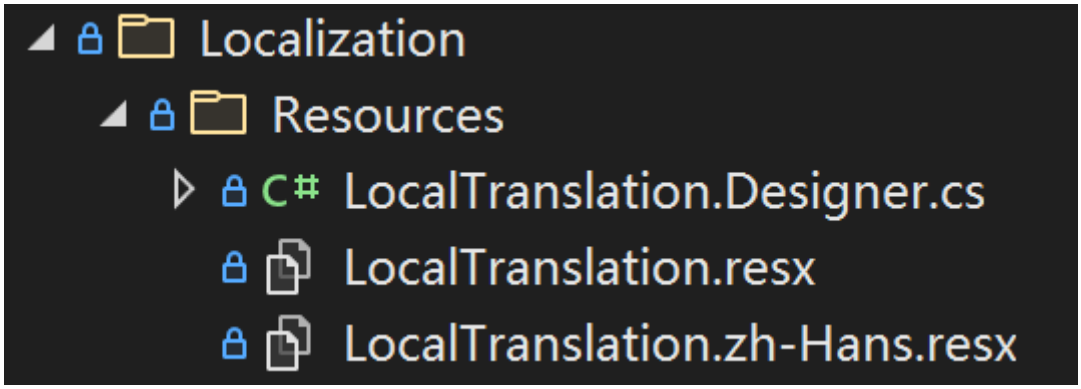
4.基础服务 (ProxyAppServiceBase)

ProxyAppServiceBase 类定义了服务层的基础协定, 包含API请求标题头以及资源服务的转换方法。通过AbpApiClient即可请求不同类型的HTTP方法获取资源。

3.本地化

该项目中主要包含两个部分

1.应用程序本地的多语言配置 (通过 .resx 文件定义)



2.服务端的多语言资源 (通过Web服务获取, 包含在 IApplicationContext 接口当中)

首先, 应用程序通过Web服务获取定义的多语言资源以及当前的语言类型, 如果不存在, 则使用应用程序的默认配置。如果存在, 那么直接生成对应的本地化信息并应用设置。如下所示:

```
        public void SetLocale(CultureInfo ci)
        {
            Thread.CurrentThread.CurrentCulture = ci;
            Thread.CurrentThread.CurrentUICulture = ci;
        }
```

并且, 将用于查找本地化的字符串资源类的Culture属性设置为当前选项:

```
LocalTranslation.Culture = userCulture;
```

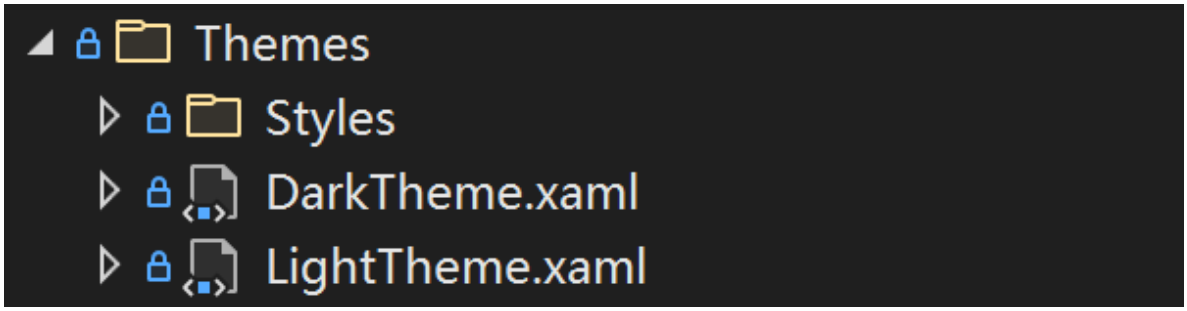
> 应用程序可配置多种本地化资源, 如: zh-Hans, en-US等等, 注意需要按照Web服务当中的约定进行命名。

4.应用主题

该项目中, 提供了两组主题的切换: 浅/深色模式。

主题适用于大部分常用的UI组件, 如: Button、Entry、CheckBox、ComboBox 等等。

Themes文件夹中分别定义了 Light(浅色) 及 Dark(深色) 模式下的不同资源。



同时, 提供了用于应用不同主题的扩展方法, ThemePalette类中包含用于设置主题的具体实现方法。

- ApplyLightTheme (浅色主题)
- ApplyDarkTheme (深色主题)

SkinView 内容视图包含的具体的设置以及功能实现。

5.模块导航

应用程序的模块导航, 主要由Prism.Forms.Regions实现, 首先, 需要在容器中注册对应的区域服务扩展。

```
        protected override void RegisterTypes(IContainerRegistry
containerRegistry)
        {
            //...

            //注册Prism区域服务及导航页
            containerRegistry.RegisterRegionServices();
            //...
        }
```

接着, 需要将用于区域导航的页面注册在容器当中, 主要通过RegisterForRegionNavigation扩展方法实现 :

```
//示例: 向容器中注册用于区域导航服务的用户视图
services.RegisterForRegionNavigation<UserView, UserViewModel>();
```

通过IRegionService 接口, 即可向指定的区域当中导航不同的视图, 如下所示:

```
regionManager.RequestNavigate(regionName, pageName);
```

页面中, 如果需要接收导航传递的参数, 可用通过实现 IRegionAware 接口

```
    public interface IRegionAware
    {
        //导航接收
        void OnNavigatedTo(INavigationContext navigationContext);

        //是否保存原导航对象
        bool IsNavigationTarget(INavigationContext navigationContext);

        //导航更改时
        void OnNavigatedFrom(INavigationContext navigationContext);
    }
```

6.系统模块

INavigationItemService 接口中定义了用于获取模块列表的方法实现, 该方法主要通过本地定义的模块列表与Web服务中返回的权限定义进行比较, 符合权限的模块将被定义在可用列表当中, 如下所示:

```
public ObservableCollection<NavigationItem> GetAuthMenuItems(Dictionary<string,
string> grantedPermissions)
        {
            var authorizedMenuItems = new ObservableCollection<NavigationItem>
();
            foreach (var menuItem in menuItems)
            {
                menuItem.Title = Local.Localize(menuItem.Title);

                if (menuItem.RequiredPermissionName == null)
                {
                    authorizedMenuItems.Add(menuItem);
                    continue;
                }

                if (grantedPermissions != null &&

 grantedPermissions.ContainsKey(menuItem.RequiredPermissionName))
                    authorizedMenuItems.Add(menuItem);
            }
            return authorizedMenuItems;
        }
```

> **说明:** 关于模块的图标、名称、导航页面名称、权限定义名都在应用程中定义。名称需要按照Web服务当中的约定进行定义, 这样便于多语言实现。

7.验证器及实体映射

- 验证器 (FluentValidation)

该组件主要用于提交表单(数据)时的实体验证, 如: 字符长度、是否必填、邮箱、正则表达式等。

验证器主要通过FluentValidation组件进行扩展, 通过定义全局的验证器IGlobalValidator, 通过依赖注入的形式, 在应用程序的任何地方进行数据验证。

- 实体映射 (AutoMapper)

该组件主要用于与Web服务及本地服务进行数据交换时, 实体模块的一些转换工作。

例如Dto->Model的转换, Dto负责从Web服务获取数据, Model则用于在Xamarin.Forms当中进行数据绑定, 通知更改等作用。

实体映射器主要通过AutoMapper组件进行扩展, 与验证器类似, 可用在任何地方进行数据转换。

8.其它的一些说明

除以上内容之外, 还存在一些常用的功能及方法, 例如:

- Converter (常用的转换器)
- Behaviors (事件命令转换)
- Controls (常用的控件封装)
- Web请求帮助类、异步类、异常处理类、本地资源访问类
- 弹窗帮助类 (用于提示错误、信息等)