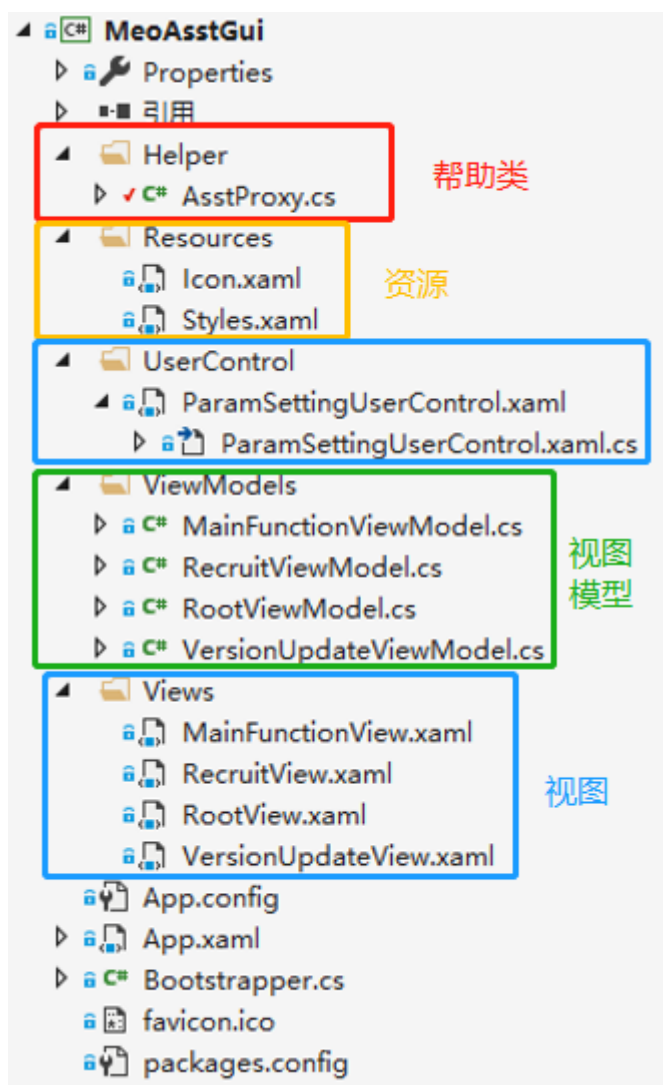


MeoAsstGui开发指南

项目中使用了Stylet框架，一个轻量级的WPF MVVM框架，源于Caliburn.Micro。不太习惯Caliburn.Micro通过命名约束来绑定命令、事件的做法，Stylet保留了Caliburn.Micro简化的命令、事件绑定，体积更小，文档详细，所以选择了它。文档目录<https://github.com/canton7/Stylet/wiki>。样式控件库暂时使用了HandyControl <https://handyorg.github.io/handycontrol/>。这些库NuGet里都可以直接安装到项目。

这里假设你懂一些WPF的初级知识（知道xaml是啥，懂些基本控件，布局）。推荐学习葵花宝典：WPF自学手册。可以想解些WPF MVVM的知识可以b站搜索MVVM 刘铁猛。猛哥的视频还是比较简洁适合新人的，C#不懂也可以学他的哈。

文件目录结构



总体结构如图1所示，下面会分别介绍各部分及现有的文件。未来可能还会有Model，Converter、Validation等部分。

Helper

存放一些包含静态方法的帮助类。

AsstProxy.cs

AsstProxy.cs里有AsstProxy类的实现。其实这个放在别的地方可能更好，暂时放在Helper里吧。这个类用来负责一切调用C++库相关接口的事务。原先分散在各个窗体下的C++调用都交由它来调用，里面对各个函数进行了封装，例如：

```
1 1个引用
---public void AsstStop()
---{
---    AsstStop(_ptr);
---}

3 个引用
---public bool AsstCatchEmulator()
---{
---    if (_ptr == null)
---    {
---        return false;
---    }
---    return AsstCatchEmulator(_ptr);
---}

4 个引用
---public void AsstStart(string task)
---{
---    AsstStart(_ptr, task);
---}
```

后面会介绍如何通过容器来对AsstProxy初始化、调用。

Resources

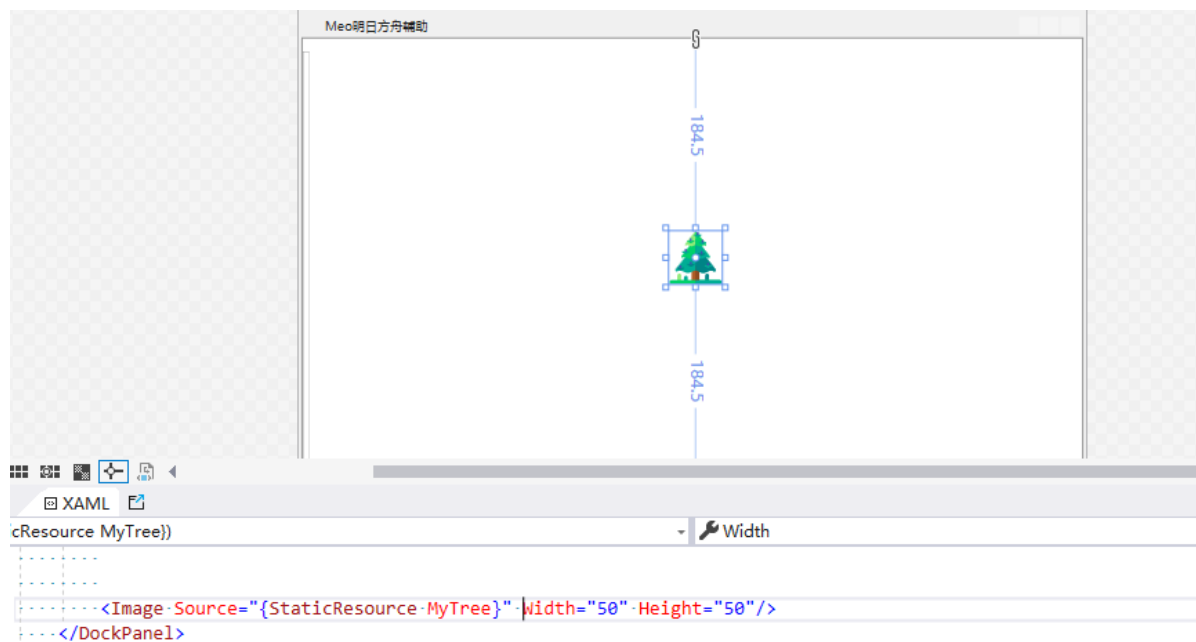
存放WPF中使用的图标、样式、多语言资源等按照xaml资源字典格式的文件。

Icon.xaml

里面暂时只存放了DrawingImage格式的图标资源。

可以在阿里矢量图库或者别的图库中找到svg格式的图标（xaml格式的也太少了），通过SharpVector将svg格式转化为xaml中的格式，添加图标资源。

使用这些资源也很简单，在启动的适合，我们在App.xaml中引用了这个资源字典，所以使用的时候只需要放个Image控件，Source引用静态资源就行了。如图：



这棵树只是举个例子，暂时也没用到，当然Image可以放在别的ContentControl里，比如

```

<Button Width="50" Height="50" Background="Transparent">
  <Image Source="{StaticResource MyTree}" />
</Button>
<Label Width="50" Height="50">
  <Image Source="{StaticResource MyTree}" />
</Label>

```

这样就可以创建样子为一棵树的按钮/标签了。

Styles.xaml

放了常用的一些样式资源，这个使用起来更简单了，直接对目标控件

```

<TabControl Style="{StaticResource StyletConductorTabControl}" Margin="0,10,0,0"/>

```

就能引用样式。

注意!!!

开头提到我们使用了HandyControl的样式控件库，如果不设置样式的话，默认会使用里面的一些样式。这也意味着使用自己写的样式的时候会覆盖掉HandyControl提供的样式。例如，我们在TabControl中使用了StyletConductorTabControl样式，这玩意儿是用来简化绑定的，在Stylet文档中有。

通过它，我们不需要自己去设置TabControl如何显示标签页与内容。但是直接按照文档那样写，就覆盖掉了HandyControl中的样式，所以我们需要继承一下样式：

```

<Style x:Key="StyletConductorTabControl" TargetType="TabControl" BasedOn="{StaticResource TabControlInline}">
  <Setter Property="ItemsSource" Value="{Binding Items}" />
  <Setter Property="SelectedItem" Value="{s:RethrowingBinding.ActiveItem}" />
  <Setter Property="DisplayMemberPath" Value="DisplayName" />
  <Setter Property="ContentTemplate" />
  <Setter Value>
    <DataTemplate>
      <ContentControl s:View.Model="{Binding}" VerticalContentAlignment="Stretch" HorizontalContentAlignment="Stretch" IsTabStop="False" />
    </DataTemplate>
  </Setter.Value>
</Setter>
</Style>

```

这个BasedOn后面的就是HandyControl中的静态资源。样式的继承只能静态资源，不能继承动态资源（这里不懂的可以百度WPF的动态、静态资源区别，我们暂时只用到了静态资源）。这里的就是设置属性的，效果等同于在创建空间时候

ControlTemplate、DataTemplate分别是用来描述如何显示控件、数据的，也是WPF提供的，有兴趣可以百度，不懂也没关系。

视图

视图这里分了两部分，一部分是Views，会直接与ViewModel绑定的页面，另一部分是作为某个View中一部分的UserControl，看个例子就懂了。

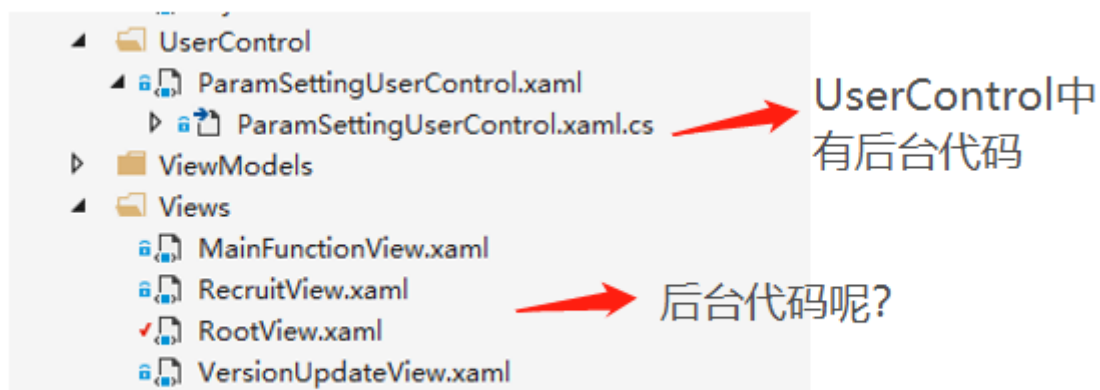
我们可以将刷理智界面分为多个区域，那么刷理智界面就是一个View，里面用到的区域就可以抽离成用户控件，放在UserControl下面。功能交互、运行状态也可以抽离成用户控件，因为没多少内容，所以直接写在View中了。

分成两个部分主要是为了方便维护，都写一块儿也没事。



后台文件

有WPF经验，但是没有Stylet框架经验的人可能要问了，Views中的文件咋都没有后台代码呢？



在MVVM下，后台代码除了在写一些依赖属性，或者纯页面交互的时候会需要（比如拖动窗体之类的，ViewModel不知道View中的内容，写起来很麻烦），别的时候都用不到，逻辑都写到了ViewModel中。我们可以找个空的Window或者UserControl看下，里面只在构造方法里面写了个InitializeComponent();

在Stylet中，大部分场景都是激活ViewModel来显示View的，所以它把InitializeComponent()的通过框架完成了。但是UserControl中的内容不是框架使用ViewModel时去使用的View；而是我们在View里写的，把后台文件删了，这个类创建时没有InitializeComponent()了，运行后这部分啥也不显示。

这里看不懂的话也没关系，只要记住Views里面**可以**把后台代码删了。（当然不删也行(o°ω°o)）

视图模型及MVVM

这里主要是写给MVVM新手看的，懂的大佬们可以忽略了~

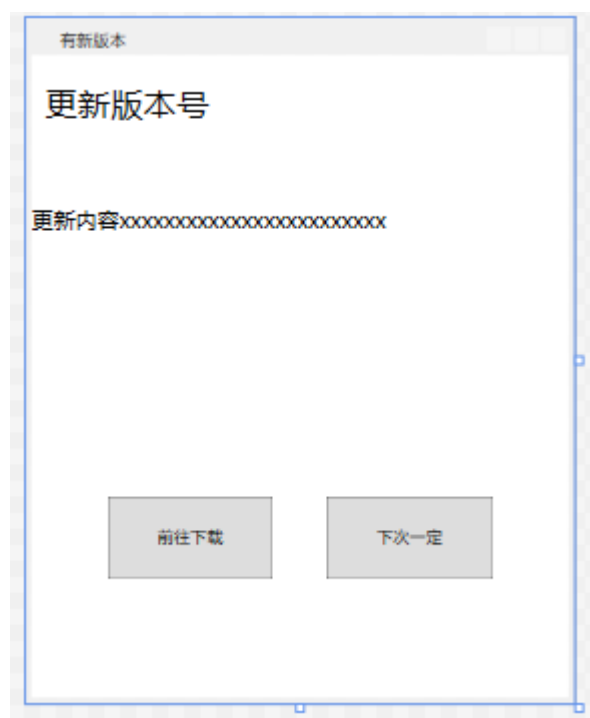
众所周知，WPF很牛逼的一个地方就是天生支持MVVM（不用MV*，那还不如直接上Winform呢、(o`v`)ノ）。



简单来说，在MVVM中，View无法直接访问Model/Service/数据库等内容，只能通过ViewModel来访问。ViewModel一般不直接使用View中内容，而是通过绑定去完成内容显示；View的交互（输入内容、点击按钮等）也通过绑定来通知ViewModel。

例子

我们拿VersionUpdate举例子。（例子中暂时省去布局相关的代码，使用伪代码写）



页面非常简单，主要有四个部分：

1. 显示更新的版本号
2. 显示更新的内容
3. 前往下载的按钮
4. 关闭按钮

那么我们按照非MVVM的写法

```
<StackPanel>
    <TextBlock x:Name="tb_UpdateTag"/>
    <TextBlock x:Name="tb_UpdateInfo"/>

    <Button Click="下载"/>
    <Button Click="关闭"/>
</StackPanel>
```

通过设置tb_UpdateTag.Text属性去修改显示的版本号（用Label的话就是Content属性），通过tb_UpdateInfo.Text属性去修改显示的更新内容。

在Button的点击事件处理程序中，完成下载、关闭等操作。

这么做当然可以，只是项目大了后，页面后台文件的内容会变得很多，在当页面发生变动时，维护起来比较困难。

MVVM的写法一般是

```
<StackPanel DataContext="某个ViewModel">
    <TextBlock Text="{Binding UpdateTag}"/>
    <TextBlock Text="{Binding UpdateInfo}"/>

    <Button Command="{Binding 下载命令}"/>
    <Button Command="{Binding 关闭命令}"/>
</StackPanel>
```

不熟悉WPF的朋友可能要疑惑了，这个DataContext是啥，有什么用。DataContext是WPF中FrameworkElement提供的一个依赖属性（这里不详细解释了，依赖属性一般就是在写控件时可以设置的属性，例如上面例子中的DataContext，Text，Command都是依赖属性）。

这个DataContext可有用了，如果不设置的话，会默认找它上一级的DataContext。例如在TextBlock中没有设置DataContext，那么它会寻找上级，也就是StackPanel的DataContext作为它的DataContext。

这样写之后，我们需要修改版本号的话，只要去修改“某个ViewModel”的UpdateTag属性就行了，这就是**绑定**的意义。我们有双向绑定、单向绑定、单次等不同的绑定。这里只介绍用的多的双向和单向。

```
<TextBox Text="{Binding aaa}" />
```

在双向绑定中，修改TextBox的内容，ViewModel中的aaa属性也会跟着变化；修改ViewModel中的aaa属性，TextBox里的内容也会跟着改变。

单向绑定顾名思义就是单个方向的，修改TextBox内容，ViewModel中aaa属性会变化；修改aaa内容，TextBox不变化（反过来也行，方向可以设置）。

ViewModel中原生的属性、Command写法比较冗余，这里不赘述了，只介绍使用Stylet后的写法。

Stylet中写法

```
<StackPanel>
    <TextBlock Text="{Binding UpdateTag}"/>
    <TextBlock Text="{Binding UpdateInfo}"/>

    <Button Command="{s:Action Download}"/>
    <Button Command="{s:Action Close}"/>
</StackPanel>
```

s是stylet所在的命名控件，只要在最外边加上

```
xmlns:s="https://github.com/canton7/stylet"
```

之前也提过了，Stylet是ViewModel-First的框架，所以不需要主动设置DataContext：激活VersionUpdateViewModel时，框架会找到对应文件名的为VersionUpdateView的页面，并将页面的DataContext设置为VersionUpdateViewModel（在上面章节提到的UserControl中我们也不需要设置，它的上级是某个View，已经有DataContext了，可以直接绑定属性）。

在ViewModel中，我们可以继承Stylet中提供的基类Screen（建议看下Stylet文档里的Screen和Conductor那章），我们要写和View对应的属性，例如：

```
private string _updateTag;

public string UpdateTag
{
    get
    {
        return _updateTag;
    }
    set
    {
        SetAndNotify(ref _updateTag, value);
    }
}
```

这个SetAndNotify是stylet封装好的方法，它会在修改属性时，通知"绑定"去更新View中的属性。

还要写对应的处理函数：

```
public void Download()
{
    System.Diagnostics.Process.Start(_updateUrl);
    RequestClose();
}
```

框架约束了只要名字一样，s:Action就可以为你创建一个对应的命令。这里提一下，如果有CanDownload()方法，那么会默认作用于Command，可以用来控制按钮的使能。

这里讲的比较简洁，完整的代码在项目中可以找到，体会一下写两个例子就明白了。

Bootstrapper.cs

这个文件需要单独讲一下，可以设置一些应用启动、退出时响应的函数，也可以配置IoC（见下章），未被捕获的异常也会在这里被捕获。在App.xaml中我们设置了指定的Bootstrapper为这个文件。

IOC容器

Stylet提供了轻量快速的默认IoC容器。项目里也是使用了这个默认的。<https://github.com/canton7/Stylet/wiki/StyletIoC-Introduction>

对于没有接触过IoC容器的朋友，也有的把它叫依赖注入容器，可以理解为一个容器，里面存放着各种各样的类（例如我们上面说的ViewModel, AsstProxy）。在别的地方需要的时候，我们可以从容器里拿出来（感觉像废话（￣▽￣）ノ），这样类之间就没有直接关联关系了，都关联了容器。也不用new去创建实例，运行时的绑定耦合比编译时要松。

依赖注入容器通常有下面几种用法：

构造函数注入

在Stylet中，我们通过WindowManager去弹窗，弹对话框等操作。

当然，我们要从类中获取这个容器。所以我们需要IContainer和IWindowManager两个成员。

我们看RootViewModel中有写到：

```
private IContainer _container;
private IWindowManager _windowManager;

public RootViewModel(IContainer container, IWindowManager windowManager)
{
    _container = container;
    _windowManager = windowManager;
}
```

我们在项目没有找到任何显式调用这个构造方法，这里不是框架写的构造会默认调用这两个参数，而是通过依赖注入实现的，我们当然可以自己加一些参数去做初始化。

我们在需要IContainer的地方，会自动使用容器中注册的IContainer对象，在需要IWindowManager的地方，会自动使用容器中注册的IContainer对象。

我们在Bootstrapper的protected override void ConfigureIoC(IStyletIoCBuilder builder)方法中可以放自己需要的东西。

按照上面讲的，我们需要在这里写：

```
protected override void ConfigureIoC(IStyletIoCBuilder builder)
{
    builder.Bind<IWindowManager>().To<WindowManager>();
}
```

这样我们每次需要IWindowManager的时候，就会传一个WindowManager给它。有的人可能要问了，这样我们每次请求来的IWindowManager是不是同一个WindowManager呢？这么写确实不是，这样写的话每次请求的时候都会创建一个新的WindowManager，所以Stylet是这么写的，加了InSingletonScope()后每次请求返回的都是同一个对象了。（AsWeakBinding可以看看文档，不需要了解）


```
builder.Bind<IWindowManager>().To<WindowManager>()  
().InSingletonScope().AsWeakBinding();
```

这些配置已经在框架里配置好了，所以我们也不需要写到ConfigureIOC里去。

直接从容器拿去需要的东西

最开始的时候提到了AsstProxy用来负责所有的C++库调用，所以我们的ConfigureIOC里需要添加一个

```
builder.Bind<AsstProxy>().ToSelf().InSingletonScope();
```

这个就是将AsstProxy绑定到自己头上，就是请求AsstProxy的时候，返回一个AsstProxy。当然，也需要是全局单例的。

我们别的类在需要它的时候，只要写：

```
var asstProxy = _container.Get<AsstProxy>();
```

然后再使用asstProxy进行调用就可以了，非常方便。

同样的，假设在AsstProxy调用全自动公开招聘时，会修改公开招聘中的招募信息。在AsstProxy也需要从容器获取自动公开招聘的ViewModel，然后通过上面所说的MVVM绑定，显示到界面上。

属性注入

属性注入我不习惯用，不讲啦。

StyletIOC别的用法还是看文档吧，真的挺详细的