Reactive Extensions入门(5)：ReactiveUI MVVM框架

yycoding.xyz/post/2012/7/5/introducting-linq-and-reactiveextensions-reactiveui

从前面几篇文章可以了解到，Rx作为LINQ的一种扩展，极大地简化了异步编程。但Rx的用法不仅如此，由于其可高的扩展性，在其他很多方面也有所应用。

在前面例子中，我们使用代码和UI界面上的元素打交道，这种方式在传统的Winfom编程中很常见，但是在基于XAML构造的界面这种应用程序中，这样显得不是非常友好，XAML中声明式编程可以使得程序更加简洁，传统的方式没有利用到XAML中强大的绑定功能。之前，我们大量使用了诸如Observable.FromEvent这样的操作，然后来使用后台代码来设置控件的属性，这都是传统的编程方式。

当然，对于规模较小的程序来说，这种方式无可厚非。这种方式的最大的缺点在于，对于测试很不友好，要测试这样的应用程序很困难，我们需要创建UI控件并模拟输入，这样效率不高而且不可靠。另一个缺点是，这种方式使得代码高度耦合而且脆弱。针对这些问题，一种称为Model-View-ViewModel(MVVM)的设计模式逐渐发展起来。

结合MVVM模式和Rx类库，发展出了ReactiveUI这个MVVM框架。他能够使得应用程序可以管理，并能使用声明式、函数式的方式来表达复杂的对象间的交互。换句话说，ReactiveUI能够帮助我们描述属性之间是如何联系起来的，即使有些属性与异步方法调用有关。

1. MVVM模式

Model-View-ViewModel模式是充分利用XAML设计平台上的数据绑定功能而产生的一种设计模式。在该模式中，Model是用来表现应用程序的数据以及与界面独立的逻辑的核心对象。View就是UI控件及界面，比如窗体控件或者用户自定义控件。值得注意的是对于同一数据对象，可能有多种表现形式，比如对于同一数据源，有的视图用来显示统计或者全局的信息，有的显示每一项的详细信息。

一般我们很熟悉的是MVC模型，所以MVVM模式中的ViewModel是其特别的地方。从名字上看，ViewModel是一种针对视图的模型。可能有点不好理解。举个例子来说：在用户注册页面(View)中，一般有输入用户名，密码，重复输入密码这几个输入框。在这个视图中，用户名和密码显然是存在于Model中，但是 重复输入密码 这一项并不属于Model，它显然不应该存在于真实的数据模型中，该项只是用在View中。

在传统的以XAML为界面的程序中，开发者一般使用页面(View)的后台的代码来存储这个重复输入密码值，但是这样同样存在可测试性和紧耦合的问题。例如，如果我们要测试“只有当密码和重复密码输入的值匹配，提交按钮才可以使用”这样一个场景就变得有点困难。现在，我们将这个字段存在另一个称之为ViewModel的对象中，这个ViewModel对象只是一个普通的类，他并不需要继承自UI控件，我们可以将该对象看做是与View的交互逻辑。在我们的例子中，验证两次密码是否匹配以及在匹配时让提交按钮可用，这些逻辑都应该写在ViewModel对象中。对于每一个View，都应该有一个对应的ViewModel对象。

1.1 ViewModeld的理念

MVVM最强大的一方面在于它的目标是将一个命令(command)或者属性(property)是什么和如何执行分开来。ViewModel是对属性和命令的一种思考。在传统的基于Codebehind的用户交互框架中，开发者需要思考控件的事件和属性。当以这种方式编写代码时，意味着事件和相应的控件紧紧的联系在一起。使得测试变得困难，因为我们需要模拟出控件的动作才能测试控件对应的事件及功能是否正常。

当使用MVVM的ViewModels时，最重要的是将这两部分逻辑分开来。在View中决定了这些控件如何被触发，同时，控件对应的一些属性利用XAML的绑定技术和ViewModel绑定起来。

1.2 MVVM框架的作用

现在有很多开源的MVVM框架可以使用了，如MVVMLight、Prism，这些框架框架各有优点。但是他们都提供了实现MVVM模式的最基本要素。首先，这些框架为ViewModel对象提供了一个基类，当这些对象的属性在属性值发生改变时会得到通知，这个是通过实现INotifyPropertyChanged接口来完成的，这个接口很关键，因为他通知View需要更新绑定到界面上的数据。MVVM提供了处理命令的一套系统，当用户发出一些命令时它能够很好的处理。这是通过实现ICommand接口来实现的，这个接口通常包含在UI控件中。

2.ReactiveUI库

ReactiveUI类库是实现了MVVM模式的框架，他移除了一些Rx和用户界面进行交互的代码。ReactiveUI的核心思想是使开发者能够将属性变更以及事件转换为IObservable对象，然后在需要的时候使用IObservable对象将这些对象转换到属性中来。他的另一个核心目标是可以在ViewModel中相关属性发生变化时可以可执行相应的命令。虽然其他的框架也允许这么做，但是ReactiveUI会在依赖属性变更时自动的去更新结果，而不需要通过拉或者调用类似UpdateTheUI之类的方法。

2.1 核心类

ReactiveObject：它是ViewModel对象，该对象实现了INotifyPropertyChanged接口。除此之外，该对象也提供了一个称之为Changed的IObservable接口，允许其他对象来注册，从而使得该对象属性变更时能够得到通知。使用Rx中强大的操作符，我们还可以追踪到一些状态是如何改变的。

ReactiveValidateObject:该对象继承自ReactiveObject对象，它通过实现IDataErrorInfo接口，利用DataAnnotations来验证对象。因此属性的值可以使用一些限制标记，UI界面能够自动的在属性的值违反这些限制时显示出这些错误。

ObservableAsPropertyHelper<T>:该类可以很容易的将IObservable对想转换为一个属性，该属性存储该对象的最新值，并且在属性值发生改变时能够触发NofityPropertyChanged事件。使用该类，我们能够从IObservable中派生出一些新的属性。

ReactiveCommand：该类实现了ICommand和IObservable接口，并且当Execute执行时OnNext方法就会被执行。该对象的CanExecute可以通过IObservable<bool>来定义。

ReactiveAsyncCommand：该对象继承自ReactiveCommand，并且封装了一种通用的模式。即“触发一步命令，然后将结果封送到dispather线程中”该对象也允许设置最大并行值。当达到最大值时，CanExecute方法返回false。

3.使用ReactiveObject实现ViewModels

和其他MVVM框架一样，ReactiveUI框架有一个对象来作为ViewModel类。该对象和基于传统的实现了ViewModel对象的MVVM框架如Foundation，Cliburn.Micro类似。但是最大的不同在于，ReactiveUI能够很容易的通过名为Changed的IObservable接口注册事件变化。在任何一个属性发生变化时，都会触发通知，客户端通常只需要关注感兴趣的一两个变化了的属性。使用ReactiveUI，可以通过WhenAny扩展方法很容易的获取这些属性值：

```
var newLoginVm = new NewUserLoginViewModel();

newLoginVm.WhenAny(x => x.User, x => x.Value)
.Where(x => x.Name == "Bob")
.Subscribe(x => MessageBox.Show("Bob is already a user!"));

IObservable<bool> passwordIsValid = newLoginVm.WhenAny(
    x => x.Password, x => x.PasswordConfirm,
    (pass, passConf) => (pass.Value == passConf.Value));
```

WhenAny语法看起来过有点奇怪。方法中第一个参数是通过匿名方法定义的一系列属性。在上面的例子中，我们关心的是神马时候Password或者PasswordConfirm发生变化。最后一个参数和Zip操作符中的类似，他使用一个匿名方法来将两个结果结合起来，然后返回结果。当这两个属性中的任何一个发生变化时，方法就会执行，并以IObservable的形式返回执行结果，在上面的例子中就是passwordIsValid这个对象。

对于ReactiveObject，值得注意的是，属性必须明确的使用特定的语法进行定义。因为简单的get，set并没有实现INotifyPropertyChanged，从而不会通知ReactiveObject对象该属性发生了改变。唯一例外的就是，如果一个属性在构造器中初始化了，在以后的程序中不会发生改变。在ReactiveObject中，属性的命名也需要注意，用作属性的私有字段必须为属性名称前面加上下划线。下面的例子展示了如何使用ReactiveObject声明一个可读写的属性。

```
public class AppViewModel : ReactiveObject
{
    int _SomeProp;
    public int SomeProp
    {
        get { return _SomeProp; }
        set { this.RaiseAndSetIfChanged(x => x.SomeProp, value); }
    }
}
```

传统的实现IpropertyChangeNofity接口的实现方法如下：

```
public class AppViewModel : INotifyPropertyChanged
{
    int _SomeProp;
    public int SomeProp
    {
        get { return _SomeProp; }
        set
        {
            if (_SomeProp == value)
                return;
            _SomeProp = value;
            RaisePropertyChanged("SomeProp");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void RaisePropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = this.PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

WhenAny实现了ReactiveUI的核心功能之一，它使得开发者能够很容易将相关属性变化用IObservable表示。该功能使得可以直接使用Rx以声明的方式创建状态机。

除了使用Rx来描述复杂的异步操作事件之外，Rx和ReactiveUI结合可以使得对象在某个特定的状态下可以得到通知，即使这种状态涉及到多个不同的对象或者属性。

4. ReactiveCommand

ReactiveCommand实现了ICommand接口，他可以模拟简单的ICommand实现。我们可以将它看做是一种ICommand，可以使用Create静态方法创建。

```
var cmd = ReactiveCommand.Create(x => true, x => Console.WriteLine(x));
cmd.CanExecute(null);    //方法输出true
cmd.CanExecute("Hello"); //方法输出"Hello"
```

下面构造了一个Command，该Command只在鼠标松开时触发。

```
var mouseIsUp = Observable.Merge(
    Observable.FromEvent<MouseButtonEventArgs>(window, "MouseDown").Select(_ => false),
    Observable.FromEvent<MouseButtonEventArgs>(window, "MouseUp").Select(_ => true))
    .StartWith(true);
var cmd = new ReactiveCommand(mouseIsUp);
cmd.Subscribe(x => Console.WriteLine(x));
```

上面的例子演示了如何使用IObservable构造Command。通常我们使用WhenAny创建IObservable然后构造Command对象。大多数情况下，只有当特定的属性被设置或者取消设置时会触发Command。例如在之前的NewUserLoginViewModel中。

```
IObservable<bool> passwordIsValid = newLoginVm.WhenAny(
x => x.Password, x => x.PasswordConfirm,
(pass, passConf) => (pass.Value == passConf.Value));
var confirmCommand = new ReactiveCommand(passwordIsValid);
```

View通过按钮或者菜单绑定confirmCommand，当在两次密码不匹配时，按钮或者菜单就会呈现出灰色。当密码或者重复密码输入框中的值发生变化时，ReactiveCommand就会重新求值，来决定是否使得按钮或者菜单可用。

值得注意的是，当属性发生变化时，Command的CanExecute会立即自动更新，而不依赖于CommandManager.RequerySuggested。在WPF或者Silverlight中存在这个bug，除非你切换焦点或者点击，按钮不会重新改变其状态。使用IObservable意味着Commanding框架确切的知道在状态发生改变时，不需要重新手动执行页面上的每一个Command对象。

ReactiveCommand对象本身可以被注册，并且在执行Exectue方法时，提供一些有用的信息。这表明，订阅者可以执行一些Reactive可以执行的一些动作，使得我们能够更好的进行控制。如下：

```
var cmd = new ReactiveCommand();
cmd.Where(x => ((Int32)x % 2 == 0)).Subscribe(x => Console.WriteLine("{0} is Even numbers
.", x));
cmd.Where(x => ((Int32)x % 2 != 0))
    .Timestamp()
    .Subscribe(x => Console.WriteLine("{0} is Odd,{1}", x.Value, x.Timestamp));

cmd.Execute(2);//输出“2 is Even numbers.
cmd.Execute(3);//输出 3 is Odd,2012/3/4 20:38:51 +08:00
```

4.1使用ObservableAsPropertyHelper将Observables转化为Properties

使用WhenAny方法，可以监视对象属性的变化，并针对这些变化生成IObservable对象。但是有时候，我们想将这些生成的IObservable对象设置为一种输出属性。想象一下有这样一个场景，有一个取色器，用户能够通过3个Slide分别设置R,G,B值。每一个Slide可以使用ViewModel对象来表示，取值范围为0到1。为了显示结果，我们需要将RGB合成为一个XAML颜色对象。当RGB中的任何一个发生变化时，我们需要更新颜色属性。

我们可以常简单的通过WhenAny创建一个IObservable<Color>对象，但是我们想将这个值存回到属性中。ReactiveUI提供了一个称之为ObservableAsPropertyHelper的对象，该对象可以存储IObservable中的最新值。为了演示这一操作，我们需要创建一个“输出属性”

```
ObservableAsPropertyHelper<Color> finalColor;
public Color FinalColor
{
    get {return finalColor.Value;}
}
```

注意到属性并没有set方法，这是因为属性是由IObservable生成的，而不需要手动设定。在ViewModel的构造函数中，我们将描述如何从RGB产生FinalColor：

```
IObservable<Color> color = this.WhenAny(x => x.Red, x => x.Green, x => x.Blue, (r, g, b)
=> new Color(r.Value, g.Value, b.Value));
finalColor = color.ToProperty(this, x => x.FinalColor);
```

这一步只需在构造函数中执行一次。现在只要Red，Green，或者Blue中的任何一个发生变化，FinalColor对象都会更新以反映最新的变化值。

ReactiveObject和ReactiveCommad是创建ViewModel对象的两个核心工具。使用它们我们可以使用属性和命令以及通过描述属性和命令之间的动态关系来构建一个View。当我们关心状态变化，以及某一个属性的变化对另外一个变化产生的影像时时，我们可以将属性转换为IObservable对象。这一点可以帮助我们很好地测试ViewModel对象。

ReactiveUI还有一些功能能够帮助我们在用户界面上优雅的处理异步方法调用。几乎大部分的应用程序都需要运行后台程序，Reactive的灵活方便的异步操作能力使得ReactiveUI在获取这些异步计算结果时变得很容易。

4.2使用ReactiveAsyncCommand处理异步方法调用

在Winform或者WPF应用程序中，如果事件执行需要耗费很长时间，比如读取一个很大的文件，那么UI很容易卡死。这是因为程序在忙于处理文件读写操作或者在等待网络数据传输而不能够刷新用户界面。在Silverligh或者Windows Phone中通过规定UI线程不允许阻塞来解决了这一问题。

通常解决这一问题的办法是另外开一个线程或者使用线程池来处理这些耗时操作，但是这又带来了第二个问题，那就是所有基于XAML的框架都是线程关联的(thread affinity)，这意味着，我们只能够从创建该对象的那个线程访问该对象。所以如果您在非UI线程中更新UI，比如执行完了一些操作后直接进行类似textbox.text=results这类的更新就会抛出错误。因为非UI线程不能够更新UI上的对象。

传统的解决这一方法是在更新UI操作时调用Dispatcher.BeginInvoke方法，该方法要求代码在UI线程中运行，大致代码如下：

```
void OnSomeUIEvent(object o, EventArgs e)
{
    var someData = this.SomePropertyICanOnlyGetOnTheUIThread;
    var t = new Task(() => {
        var result = DoSomethingInTheBackground(someData);
        Dispatcher.BeginInvoke(new Action(() => {
            this.UIPropertyThatWantsTheCalculation = result;
        }));
    };
    t.Start();
}
```

ReactiveAsyncCommand将这一模式进行了一定的封装，使得我们编写代码更加容易。例如：用户界面上有时需要某个异步方法在某一段时间运行，在异步方法运行的过程中让一些按钮或者控件处于Disable状态。稍微友好一些的用户界面在后台正在进行的操作时给UI界面一些提示，比如在界面上显示，“程序正在进行xxx……”的提示，这样显得更加友好。

由于ReactiveAsyncCommand直接继承自ReactiveCommand，所以它能做基类的所有功能。使用Execute，使得Command开始在后台执行时并可以通知用户。ReactiveAsyncCommand和ReactiveCommand不同之处在于，它内建了能够自动跟踪后台线程中运行的任务的数量。

下面是一个简单的使用Command的例子，它在后台线程的Task中运行，并且只运行一次。

```
var cmd = new ReactiveAsyncCommand();
cmd.RegisterAsyncAction(i => {
    Thread.Sleep((int)i * 1000);
});

cmd.Execute(5);
cmd.CanExecute(5);//False
```

ReactiveAsyncCommand对象中是使用RegisterAsyncAction来注册异步执行操作的。它能够注册异步方法和同步方法，这些方法将会在后台线程中执行，并返回IObservable数据表示执行结果会在未来的某一时刻到来。IObservale通常对应Command调用。每一次执行Execute方法将会将结果存入到IObservable对象中。

4.3构造一个ViewModel例子

讲了这麽多ReactiveUI框架的几个重要对象。现在用一个简单的View以及与之相关联的VeiwModel来展示如何使用ViewModel。本例子将展示如何执行一些简单的和按钮相关的命令，并模拟在后台执行一些费时的操作。然后将结果显示在UI界面上。

首先来看看我们的前台页面，也就是View，在这里我建立的是一个简单的WPF应用程序。

```
<Window x:Class="RxUI.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525" x:Name="Window">
    <Grid DataContext="{Binding ViewModel, ElementName=Window}">
        <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
            <TextBlock Text="{Binding DataFromTheInternet}" FontSize="18"/>
            <Button Content="Click me!" Command="{Binding GetDataFromTheInternet}"
                    CommandParameter="5" MinWidth="75" Margin="0,6,0,0"/>
        </StackPanel>
    </Grid>
</Window>
```

View中有几个地方我们需要注意。首先我们将顶级Grid容器的DataContext参数绑定到我们的ViewModel对象上。这样，当我们使用XAML数据绑定时，这些元素相对于ViewModel而不是View来进行绑定。然后我们定义了一个TextBlock，将其内容绑定到DataFromTheInternet属性上。最后，我们绑定Button的Command属性到我们再ViewModel中定义的一个称之为GetDataFromTheInternet的Command对象上。相关的定义以及ViewModel代码如下：

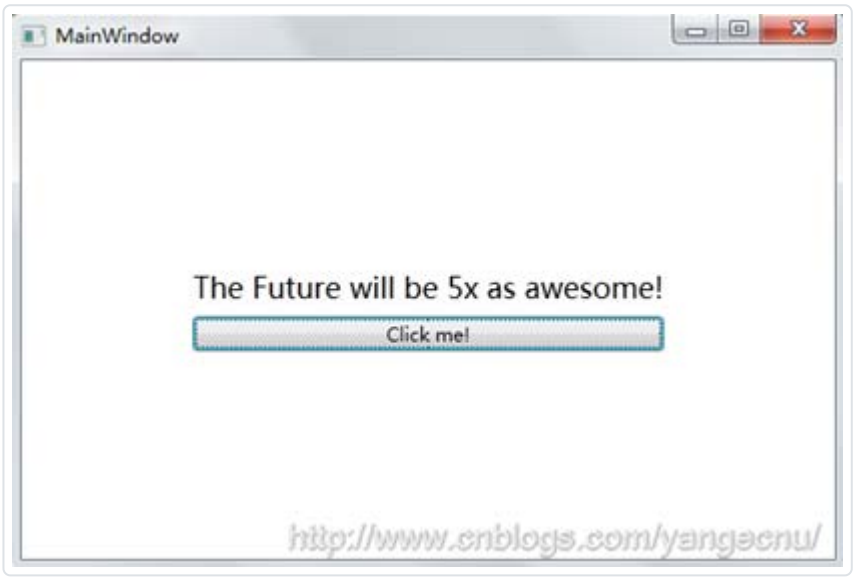
```
public partial class MainWindow : Window
{
    public AppViewModel ViewModel { get; protected set; }
    public MainWindow()
    {
        ViewModel = new AppViewModel();
        InitializeComponent();
    }
}

class AppViewModel:ReactiveObject
{
    ObservableAsPropertyHelper<String> dataFromTheInternet;
    public string DataFromTheInternet
    {
        get { return dataFromTheInternet.Value; }
    }

    public ReactiveAsyncCommand GetDataFromTheInternet { get; protected set; }
}
```

在View中，我们通过get set方法创建了一个名为ViewModel的普通属性，然后我们再构造函数的InitializeComponet方法之前初始化了该属性。接着，我们定义了一个ViewModel类来对我们的View进行建模。通过ObservableAsPropertyHelper以及ReactiveAsyncCommand定义了一个输出属性。必须是属性才在XAML中绑定，属性的setter是Protected的，因为我们只需要在构造函数中进行实例化，之后就不会再对其进行设置了。

接下来就到了比较关键的部分了-ViewModel的构造函数。ReactiveUI关注定义和描述属性和命令之间的相关关系，所以最重要的代码就在ViewModel的构造函数中，可以将这部分工作看作是属性之间的相互关联。这种方法的好处是，所有交互的代码都在这里，而不是分散在后台代码的事件处理和回调方法中。对于很多ViewModel来说，可能只有在构造函数中有一些代码。

```
public AppViewModel()
{
    GetDataFromTheInternet = new ReactiveAsyncCommand();
    var futureData = GetDataFromTheInternet.RegisterAsyncAction(I => {
        Thread.Sleep(5 * 1000);
        return String.Format("The Future will be {0}x  as awesome!", i);
    });
    dataFromTheInternet = futureData.ToProperty(this, x => x.DataFromTheInternet);
}
```

每一次用户点击按钮的时候,Command的Execute方法就会被执行一次，每5分钟就会向futureData这个Observable对象中传入一个数据。程序运行结果如下：



上面的代码很简洁，我们没有任何显示的定义异步方法比如声明一个Task或者开一个线程，也没有将返回的结果进行封送然后调用Dispatcher.BeginInvoke来更新UI界面的代码。整个代码看起来像是一个简单的单线程的应用程序。而且进一步，这种方式极大的提高了可测试性。

使用Dispatcher.BeginInvoke意味着我们假定Dispatcher存在并且起作用。但是在一个单元测试中，这个是不存在的。ReactiveUI会自动的删除这些代码并将他们换成默认的IScheduler而不使用Dispatcher.

使用ReactiveAsyncCommad，代码可以在后台线程中运行，前台UI依旧能够响应用户的操作。但是，一些长时间运行的操作，比如Web请求，并不需要频繁的进行重复。这些数据应该缓存起来，使得不同的请求只请求一次。

5.ReactiveUI中的缓存

缓存在实际开发中应用的很广泛。最常用的做法是在本地维护一个查找表，以存储最近获取的数据，当再次请求这些数据时，先查看查找表中是否存在，如果存在就直接读取，而不用再一次请求。每一种缓存方案都应该有缓存机制，例如规定缓存何时过期，如何移

除过期的数据等等。有时候不恰当的机制，比如只往缓存中添加数据，而不移除过期的数据，会导致内存泄露。

在ReactiveUI中，引入了一个称之为MemorizingMRUCache的对象，如名字所示，他是一种以最近最常使用过的数据来作为缓存方案，它会移除一些在一定时间内没有请求的数据，从而保证缓存集在一定的大小范围内。

5.1使用MemorizingMRUCache

调用MemorizingMRUCache的Get方法就可以从缓存中获取对应的值，构造缓存时需要在其构造函数中出传入缓存函数，该缓存函数必须是一种数学形态的，也就是说对于任何一个相同的给定参数，其返回值时也应该是相同的。另外一个需要注意的地方是他和QueuedAsyncMRUCache不同，他不是线程安全的。如果在多线程中使用该缓存对象，则需要加锁。下面的例子简单演示了MemorizingMRUCache的使用方法。

```
var cache = new MemoizingMRUCache<Int32, Int32>((x, ctx) => {
    Thread.Sleep(5 * 1000);
    return x * 100;
},20);

cache.Get(10);//第一次获取，需要5秒
cache.Get(10);//第二次取值，立即返回
cache.Get(15);//也需要5秒
```

5.2维护磁盘缓存

MemorizingMRUCache也可以将缓存数据从内存中存储到磁盘上供以后使用，缓存的键可以是一个URL，值可以是该URL对应的临时文件。当缓存文件不再需要时，调用OnRelease方法可以删除这些临时文件，下面是一些比较有用的函数。

- TryGet:视图从缓存中获取某一个键对应的值
- Invalidate：将某一个键对应的值的缓存进行清除，内部调用Release函数。
- InvalidateAll:清空所有缓存。

5.3 异步缓存结果

ObservableMemorizingMRUCache是一种线程安全的MemorizingMRUCache异步版本。如上所述，MemorizingMRUCache可以缓存一些需要大量计算的结果，但是它具有的缺点是其本身是单线程的结构，如果使用多线程访问或者试图缓存同时多个web请求的结果，就会产生问题。

ObservableMemorizingMRUCache解决了这一问题，同时提供了称之为AsyncGet的方法，该方法返回一个IObservable对象。该对象在异步命令返回时返回，而且只执行一次。

例如，假设我们要写一个微博客户端，需要获取每条信息发布者的人物图像，如果用传统的foreach方法的话，可能会比较慢。即使采用传统的异步方式获取，仍然存在有获取相同信息发布者的相同的人物图像的情况。

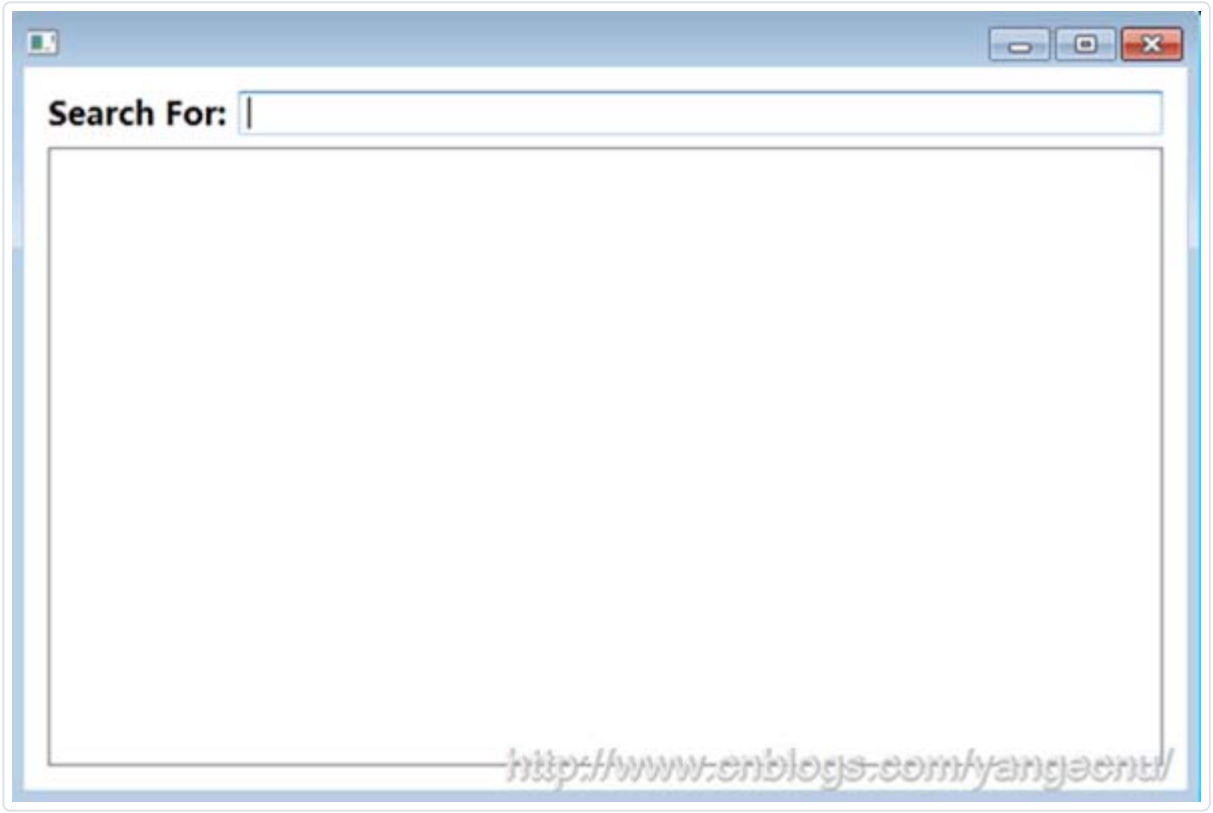
ObservableMemorizingMRUCache解决了这个问题。在前面的例子中，我们获取所有的微博信息集合，然后异步的请求发布者图像信息。对以第一条记录，我们发出WebRequest请求的时候，缓存中为空。然后我们请求第二条数据，这是时候，第一条数据可能还没有返回，我们又请求了同一个图像。如果某一个人发了50条微博信息，那么这样的请求就会产生50次。

当我们调用AsyncGet方法时，我们检查缓存，而且也需要检查请求列表。对于每一个可能的输入，我们可以认为他有三种状态，要么处于cache中，要么正在请求中，要么是全新的一个请求。ObservableAsyncMRUCache可以保证这三种状态能够以一种线程安全的方式正确处理。由于AsyncGet是一个异步方法，它能够和ReactiveAsyncCommand很好的协同工作，我们可以将他作为RegisterAsyncObservable方法的一个参数。最后的结果是一个Command对象，该对象从后台获取数据，然后自动的维持最小的请求数据，减轻并发量，而且缓存了重复的请求数据。

讲了这么多，最后我们将以一个例子展示ReactiveUI的应用。

6.使用ReactiveUI开发一个异步图片搜索工具

这是一个使用Flickr来进行照片搜索的例子，当然您也可以使用Bing等搜索引擎。当用户停止在输入框输入内容时，系统使用用户输入的关键字进行查询，然后将查询结果显示出来。界面如下：



6.1 设计MVVM

使用ReactiveUI框架的最主要目的是使用MVVM模式来开发程序，整个应用程序包含两个类。MainWindow这个是View，对应的AppViewModel是ViewModel。

在MainWindow中，我们需要创建一个AppViewModel简单属性，然后在MainWindows的构造函数的InitializeComponet()方法之前实例化AppViewModel对象。

```
public partial class MainWindow : Window
{
    public AppViewModel ViewModel { get; protected set; }
    public MainWindow()
    {
        ViewModel = new AppViewModel();
        InitializeComponent();
    }
}
```

对于AppViewModel类，使其继承自ReactiveObject对象，然后定义一个SearchTerm属性和ExecuteSearch命令，如下：

```
public class AppViewModel:ReactiveObject
{
    String _SearchTerm;
    public String SearchTerm {
        get { return _SearchTerm; }
        set { this.RaiseAndSetIfChanged(x => x.SearchTerm, value); }
    }

    public ReactiveAsyncCommand ExecuteSearch { get; protected set; }
}
```

6.2将IObservable对象转换为属性

在ReactiveUI中，我们可以将IObservable转换为属性，当Observable对象有新的值加入时，就会通知ReactiveObject对象更新其属性值。

前面讲到，要实现这个转换需要用到ObservableAsPropertyHelper类，这个类注册一个Observable对象并存储其最新值的一份拷贝。一般在ReactiveObject对象的RaisePropertyChanged方法调用时就会执行相应的操作。

```
ObservableAsPropertyHelper<List<FlickrPhoto>>
_SearchResults;
public List<FlickrPhoto> SearchResults {get { return this._SearchResults.Value; }}
ObservableAsPropertyHelper<Visibility> _SpinnerVisibility;
public Visibility SpinnerVisibility { get { return _SpinnerVisibility.Value; } }
```

上面创建一个属性，用来控制Spinner控件的显示，在应用程序忙时给出提示。然后，我们创建一个构造函数，定义两个可选属性，来方便测试。

```
    public AppViewModel(ReactiveAsyncCommand testExecuteSearchCommand = null,
IObservable<List<FlickrPhoto>> testSearchResults = null)
    {
        ExecuteSearch = testExecuteSearchCommand ?? new ReactiveAsyncCommand();
        ……
    }
```

ViewModel中的属性是彼此相互联系的，传统的方法很难简洁的描述他们之间的关系，如“当程序正在搜索时，显示Spinner”，这个简单的关系通常会涉及到好几个事件处理。使用ReactiveUI能够以一种很整洁清晰的方式定义各个属性之间的关系。

我们需要将属性转换为Observable对象，当搜索的关键字发生变化时，Observable就会返回一个对象。和之前的例子一样，我们使用Throttle操作符来忽略一些不必要的频繁的操作。我们并不想监听键盘每一次按下事件，我们监听变化的值，忽略两次相同的查询以及为空的查询。

最后，使用RxUI的InvoleCommand方法，该方法接受String类型，然后调用ExecuteSearch的Execute方法。

```
this.ObservableForProperty(x => x.SearchTerm)
        .Throttle(TimeSpan.FromMilliseconds(800), RxApp.DeferredScheduler)
        .Select(x => x.Value)
        .DistinctUntilChanged()
        .Where(x => !String.IsNullOrWhiteSpace(x))
        .InvokeCommand(ExecuteSearch);
```

当正在运行查询时，我们需要显示Spinner控件，ReactiveUI能够描述这种状态。ExecuteSearch有一个称之为ItemsInFlight的IObservable<int>属性，当有新的值产生或者移除时，会触发该属性发生变化，我们可以将这些信息和Visibility属性结合起来，当该值等于0时隐藏，大于0时显示。然后使用ToProperty操作符来创建ObservableAsPropertyHelper对象。

```
spinnerVisibility = ExecuteSearch.ItemsInflight
    .Select(x => x > 0 ? Visibility.Visible : Visibility.Collapsed)
    .ToProperty(this, x => x.SpinnerVisibility, Visibility.Hidden);
```

然后，我们需要定义当命令触发时应该执行的操作。在命令执行时，我们需要调用GetSearchResultsFromFlicker方法。值得注意的是，该方法的返回结果是一个Observable集合，每一次执行操作时，都会返回一个List类型的FlickerPhoto的Observable对象。

下面是构造函数中的方法和GetSearchResultsFromFlicker函数。

```
IObservable<List<FlickrPhoto>> results;
    if (testSearchResults != null)
    {
        results = testSearchResults;
    }
    else
    {
        results = ExecuteSearch.RegisterAsyncFunction(term =>
GetSearchResultsFromFlickr((String)term));
    }
    _SearchResults = results.ToProperty(this, x => x.SearchResults, new List<FlickrPhoto>
());

private static List<FlickrPhoto> GetSearchResultsFromFlickr(string searchTerm)
{
    var doc = XDocument.Load(String.Format(CultureInfo.InvariantCulture,
        "http://api.flickr.com/services/feeds/photos_public.gne?tags={0}&format=rss_200",
        HttpUtility.UrlEncode(searchTerm)));
    if (doc.Root == null)
        return null;
    var titles = doc.Root.Descendants("{http://search.yahoo.com/mrss/}title").Select(x =>
x.Value);
    var tagRegex = new Regex("<[^>]+>", RegexOptions.IgnoreCase);
    var descriptions =doc.Root.Descendants("{http://search.yahoo.com/mrss/}description")
                        .Select(x => tagRegex.Replace(HttpUtility.HtmlDecode(x.Value),
""));
    var items = titles.Zip(descriptions,
                            (t, d) => new FlickrPhoto { Title = t, Description = d
}).ToArray();
    var urls = doc.Root.Descendants("{http://search.yahoo.com/mrss/}thumbnail")
                        .Select(x => x.Attributes("url").First().Value);
    var ret = items.Zip(urls, (item, url) =>
    {
        item.Url = url; return item;
    }).ToList();
    return ret;
}
```

程序的后台代码写好了，前台代码如下，图中红色方框部分就是绑定ViewModel数据部分。可以看到UI界面上的控件都以声明的方式基本都和ViewModel部分的数据绑定好了，使得View的后台页面基本上没有什么代码，您是否体会到了XAML的强大的数据绑定能力呢。

```
<Window x:Class="ReactiveUIImageSearch.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Name="Window" Height="350" Width="525">
    <Window.Resources>
        <DataTemplate x:Key="PhotoDataTemplate">
            <Grid MaxHeight="100">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>

                <Image Source="{Binding Url, IsAsync=True}" Margin="6" MaxWidth="128"
                       HorizontalAlignment="Center" VerticalAlignment="Center" />

                <StackPanel Grid.Column="1" Margin="6">
                    <TextBlock FontSize="14" FontWeight="Bold" Text="{Binding Title}" />
                    <TextBlock FontStyle="Italic" Text="{Binding Description}"
                               TextWrapping="WrapWithOverflow" Margin="6" />
                </StackPanel>
            </Grid>
        </DataTemplate>
    </Window.Resources>
    <Grid DataContext="{Binding ViewModel, ElementName=Window}" Margin="12">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <TextBlock FontSize="16" FontWeight="Bold" VerticalAlignment="Center">Search For:</TextBlock>
        <TextBox Grid.Column="1" Margin="6,0,0,0" Text="{Binding SearchTerm, UpdateSourceTrigger=PropertyChanged}"/>
        <TextBlock Grid.Column="2" Margin="6,0,0,0" FontSize="16" FontWeight="Bold" Text="..." Visibility="{Binding SpinnerVisibility}" />

        <ListBox Grid.ColumnSpan="3" Grid.Row="1" Margin="0,6,0,0"  ScrollViewer.HorizontalScrollBarVisibility="Disabled"
                 ItemsSource="{Binding SearchResults}" ItemTemplate="{DynamicResource PhotoDataTemplate}"  />
    </Grid>
</Window>
```


编译运行，下面是程序运行结果。

7.总结

本文介绍了ReactiveUI这个和Rx结合紧密的MVVM框架，它使得我们开发的基于XAML的程序更加直观，简洁和可维护。另外使用Rx和ReactiveUI，使得程序的能够很方便的进行测试，可以使用Rx和ReactiveUI来模拟整个流程，当然这也是所有MVVM框架要达到的目的。 希望本文对您了解ReactiveUI及MVVM有所帮助。