Reactive Extensions入门(4)：Rx实战

yycoding.xyz/post/2012/6/27/introducting-linq-and-reactiveextensions-practical-rx

Reactive Extensions(Rx)的优点在于能够将传统的异步编程方式从支离破碎的代码调用中解放出来。传统的采用回调的异步编程方式会使得代码很零散，尤其是异步嵌套异步的时候，代码块很难管理。这个和一般的编程语言不推荐使用goto语句的原因是一样的。Rx能够使的我们可以将异步代码写到一个单独的方法中，使得代码可读性和可维护性大大增强。

前一篇文章中我们了解了Rx中的一些比较重要的操作符。本文中我们将会学习如何将这些应用到我们的应用程序中。

1. 异步调用

在开始讨论异步调用之前，我们来看看同步调用。下面的代码是一个简单的Console程序。

```
static void Main(string[] args)
{
    int x = 4;
    int y = 5;
    int z = PlusTwoNumber(x, y);
    Console.WriteLine(z);
}
private static int PlusTwoNumber(int x, int y)
{
    return x + y;
}
```

Main方法中，对PlusTwoNumber的调用时同步的，也就是Main方法在执行下一条命令之前会一直等待PlusTwoNumber方法返回值。

使用Observable.Start

同步方法调用是阻塞式的，在很多场景下这是不合适的。我们能够将上面的同步方法调用改造成异步调用。一个最简单的方法就是使用IObservable.Start方法，使得Rx为我们来管理这些异步调用。

```
static void Main(string[] args)
{
    PlusTwoNumberAsync(5, 4).Subscribe(Console.WriteLine);
    Console.ReadKey();
}
private static IObservable<int> PlusTwoNumberAsync(int x, int y)
{
    return Observable.Start(()=>PlusTwoNumber(x,y));
}
private static int PlusTwoNumber(int x, int y)
{
    Thread.Sleep(5000);
    return x + y;
}
```

使用Observable.Return

除了Observable.Start外也可以使用Observable.Return来将同步方法改造为异步方法。只需要将上面的PlusTwoNumberAsync方法改为下面即可，运行程序的效果相同。

```
private static IObservable<int> PlusTwoNumberAsync(int x, int y)
{
    return Observable.Return(PlusTwoNumber(x, y));
}
```

使用SelectMany

使用SelectMany可以很方便的实现诸如在一个异步方法中调用另外一个异步方法的功能。

```
static void Main(string[] args)
{
    PlusTwoNumberAsync(5,
4).SelectMany(aPlusB=>MultiplyByFiveAsync(aPlusB)).Subscribe(Console.WriteLine);
    Console.ReadKey();
}
private static IObservable<int> PlusTwoNumberAsync(int x, int y)
{
    return Observable.Start(() => PlusTwoNumber(x, y));
}
private static IObservable<int> MultiplyByFiveAsync(int x)
{
    return Observable.Return(MultiplyByFive(x));
}
private static int PlusTwoNumber(int x, int y)
{
    Thread.Sleep(5000);
    return x + y;
}
private static int MultiplyByFive(int x)
{
    Thread.Sleep(5000);
    return x * 5;
}
```

使用FromAsyncPattern

FromAsyncPattern方法可以将传统的Begin/End的异步编程模式转换为单独的一个方法。为了展示如何使用FromAsyncPattern，下面演示将微软的Bing搜索引擎接口包装成Rx方法。该方法的同步方法签名如下：

public SearchResponse Search(SearchRequest parameters)

相对应的异步方法如下：

public IAsyncResult BeginSearch(SearchRequest parameters, AsyncCallback callback, object asyncState)

public SearchResponse EndSearch(IAsyncResult result)

如果需要进行查询，我们可以将上面的异步调用封装为一个简单的方法，使得签名看起来类似：

```
IObservable<SearchResponse> BingSearch(this BingPortTypeClient client, string searchText)
{
    var subjet = new AsyncSubject<SearchResponse>();
    SearchRequest searchRequest = new SearchRequest();
    searchRequest.AppId = App.BingSearchKey;
    searchRequest.Query = searchText;
    searchRequest.Sources = new SourceType[1];
    searchRequest.Sources[0] = SourceType.Image;
    client.BeginSearch(searchRequest, asynchResult => {
        try
        {
            SearchResponse response = client.EndSearch(asynchResult);
            subjet.OnNext(response);
            subjet.OnCompleted();
        }
        catch (Exception ex)
        {
            subjet.OnError(ex);
        }
    },null);
    return subjet;
}
```

在方法内部我们需要调用BeginSearch和EndSearch方法，然后将值填充到IObservable对象中。更重要地，我们要用到subject这一AsyncSubject实例对象。Subject的看起来是我们需要注册到Observable对象上的对象，但是该对象在等到值返回时就会调用，我们会丢失结果，因为这是一个Hot Observable对象。AsyncSubject对象只返回最后的那个结果，从而解决了竞争情况。它创建了一个Cold Observable对象。

将Begin/End方法转换为基于Observable的方法在Rx中非常常见，所以Rx的Observable对象提供了一个名为FromAsyncPattern的方法。该方法返回一个Func类型，Func的返回值类型为IObservable<TResult>

我们可以从Begin/End异步方法调用对应的同步方法来确定改Func的参数。以上面的例子来说，Search的同步方法为:

public SearchResponse Search(SearchRequest parameters)

所以返回的Func的输入参数为SearchRequest。当Func返回时，Rx为我们处理Begin/End异步调用，并返回IObservable对象，就像我们上面手动处理的那样，这极大的减少代码量。使用Rx将上面的BingSearch方法重写为：

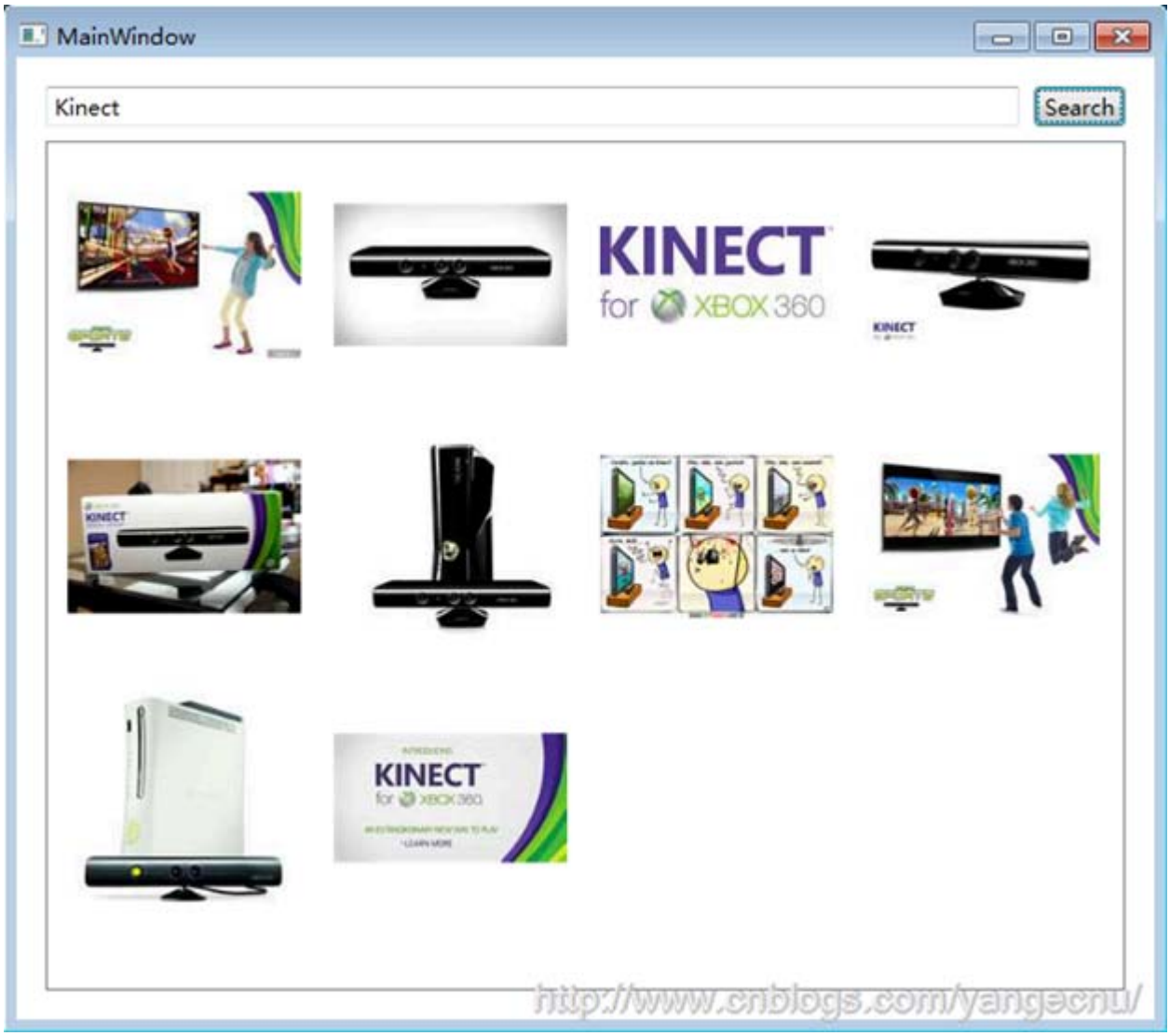
```
IObservable<SearchResponse> BingSearch(this BingPortTypeClient client, string searchText)
{
     Func<SearchRequest, IObservable<SearchResponse>> observableSearchFunc =
                Observable.FromAsyncPattern<SearchRequest, SearchResponse>
(client.BeginSearch, client.EndSearch);

    SearchRequest searchRequest = new SearchRequest();
    searchRequest.AppId = App.BingSearchKey;
    searchRequest.Query = searchText;
    searchRequest.Sources = new SourceType[1];
    searchRequest.Sources[0] = SourceType.Image;
    return observableSearchFunc(searchRequest);
}
```

下面我们以一个实际的例子来说明在实际编程中如何使用FromAsyncPattern

2. 实例：使用Rx来进行异步操作

下面用一个使用Bing来查询图片的例子来展示如何在实际应用开发中使用Rx来简化异步操作。

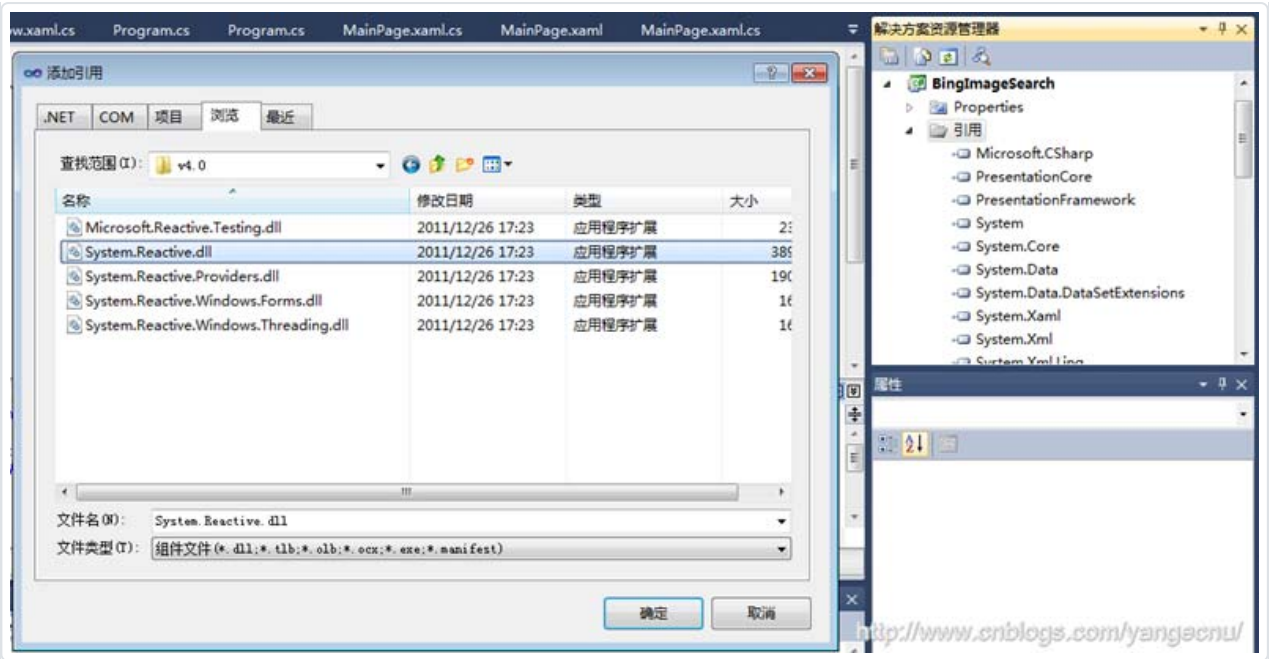
我们的这个应用叫BingImageSearch。程序可以帮助用户使用Bing搜索图片，并将结果显示出来。查询到的结果是异步返回的，并存储到Observable集合中。



创建应用程序，我们需要建立和Bing搜索引擎的连接，然后需要处理异步查询操作，获取异步返回的结果，这个场景很适合使用Rx。下面将一步一步演示如何建立。步骤如下：

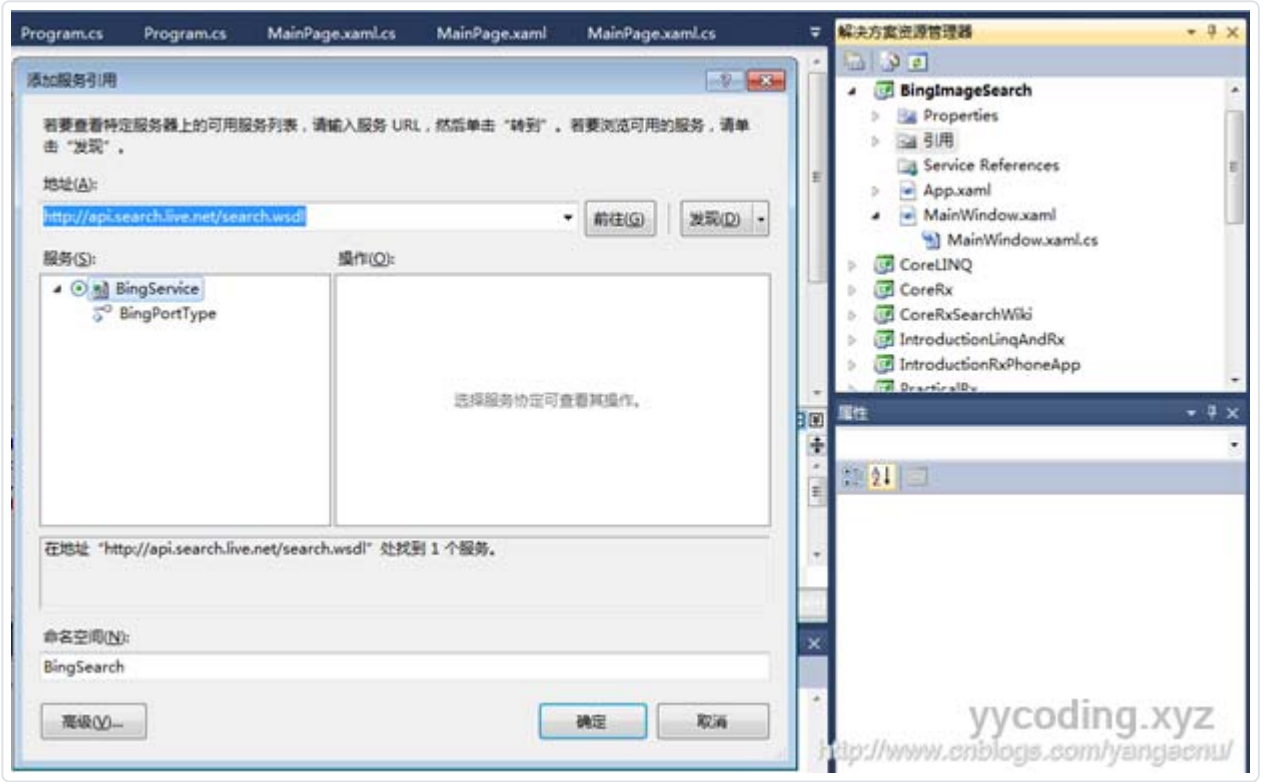
- 创建一个新的WPF应用程序。
- 添加对Reactive Extension dll的引用。
- 添加Bing搜索引擎服务。
- 获取Bing 搜索许可API。
- 创建UI界面。
- 创建Rx函数的原型。
- 实现基于Rx的SearchBingAPI
- 添加查询按钮的时间处理。
- 完成CreateImageFromURL逻辑

首先创建一个WPF应用程序，然后添加Rx相关的dll，如下图：

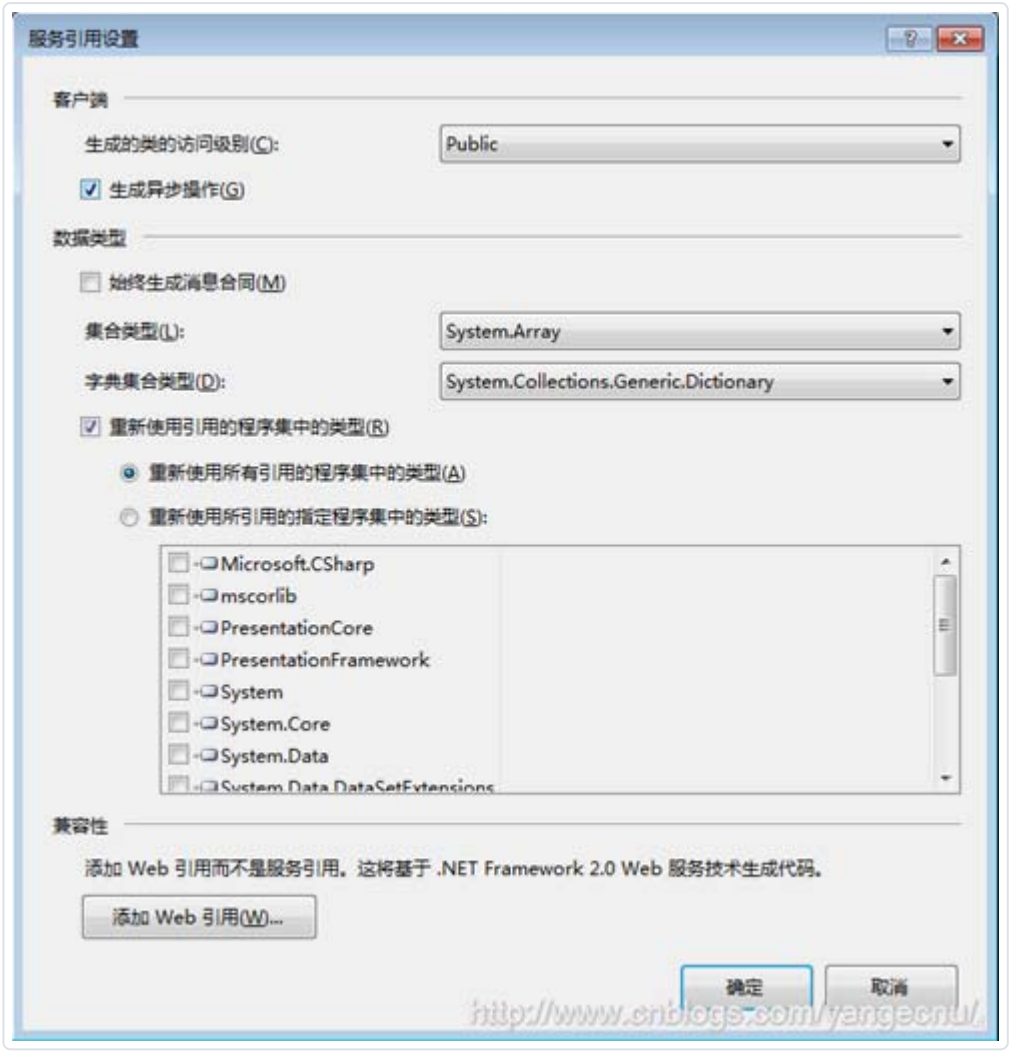


添加Bing搜索服务引用

添加完会Rx的引用之后，需要添加Bing API的引用，如图，在Add Service Reference中输入**http://api.search.live.net/search.wsdl**. 然后点击“前往”，找到BingService。



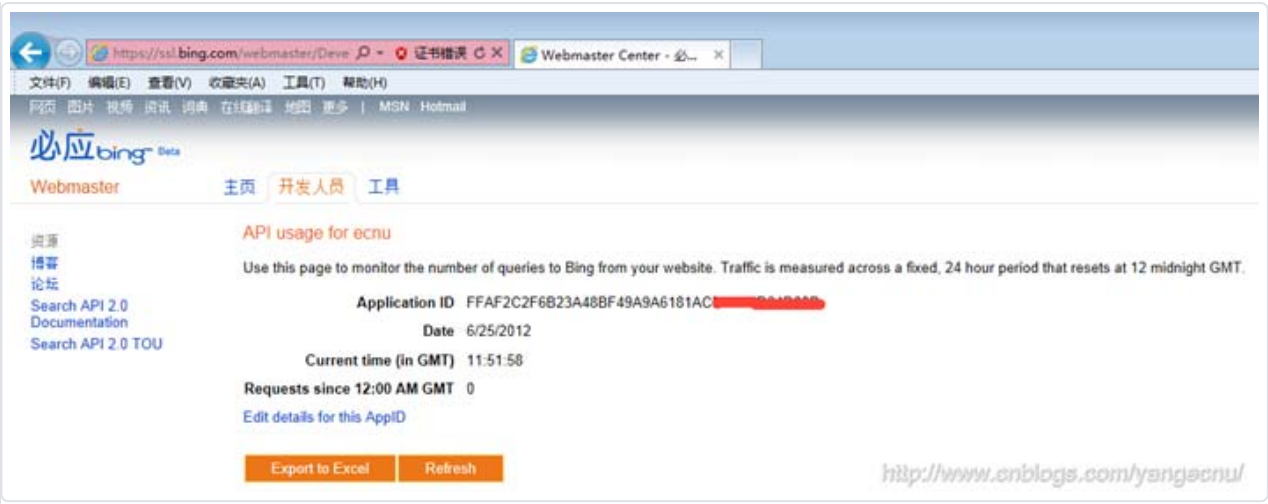
要注意的是，要点击”高级……”按钮，然后选择”生成异步操作”复选框。



WebService是使用异步回调的经典场景，从WebService返回的结果，通常受限于用户的网络连接情况。阻塞WebService的调用是不允许的，比如在Windows Phone开发中。

之所以需要产生异步调用的方法是因为我们在Observable.FromAsyncPattern中创建一个Rx风格的方法中需要用到。Rx风格的方法中不使用回调，他不需要一个线程来等待WebService的结果返回。简言之。我们需要编译器帮我们生成这些异步方法，只是为了在将这些方法转变为Rx方法时更为方便。

我们将该服务命名为BingSearch。有了服务之外，我们还需要到Bing开发者中心www.bing.com/developers. 申请一个Key，将Key放在程序中的合适位置才能使用Bing提供的API功能。



将申请到的Key放到App.xaml.cs中，如下：

```
public partial class App : Application
{
    public const string BingSearchKey = "FFAF2C2F6B23A48BF49A9A6181AC**********";
}
```

创建UI

UI界面非常简单，一个输入搜索的TextBox，一个Button和一个用来显示查询结果的Image的Listbox。

```
<Window x:Class="BingImageSearchDemo.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">
    <Grid Margin="16">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <TextBox x:Name="SearchBox" />
        <Button x:Name="SearchButton" Grid.Column="1" Content="Search" Margin="8,0,0,0"
Width="50" />
        <ListBox x:Name="Results" Grid.Row="1" Grid.ColumnSpan="2" Margin="0,8,0,0" />
    </Grid>
</Window>
```

界面构造好了之后就要实现逻辑了。

构造Rx函数原型

我们要构造的函数原型如下，

```
IObservable<String> SearchBingImageApi(String query)
{
}

BitmapImage CreateImageFromUrl(String url)
{
}
```

SearchBingImageApi方法用来根据输入的关键字通过Bing查找对应的所有图片的Url地址。CreateImageFromUrl通过前面获取的url产生图片。

实现Rx函数原型

首先来看第一个函数SearchBingImageApi。该函数的主要目的是产生一个未来的所有查询结果的url集合。也就是查询结果会在异步调用的未来的某一时刻到来。在Rx中我们通过返回一个IObservable集合来实现。IObservable集合中的数据将会通过异步填充。

该方法返回的IObservable会被接下来的CreateImageFromUrl方法使用。

a. 实现SearchBingImageApi

要使用Bing搜索，首先需要创建一个SearchRequest对象，然后将我们之前申请的API Key复制给该对象的AppId属性，最后我们将要查询的关键字复制给该对象的Query属性：

```
SearchRequest searchRequest = new SearchRequest();
searchRequest.AppId = App.BingSearchKey;
searchRequest.Query = query;
```

接下来，我们需要设置查询的类型，也就是我们需要定义是什么查询，是查询文本，图片还是视频等等，最后我们需要实例化一个BingPortTypeClient对象：

```
searchRequest.Sources = new SourceType[1];
searchRequest.Sources[0] = SourceType.Image;
BingPortTypeClient client = new BingPortTypeClient();
```

到目前位置，如果没有Rx，我们就需要通过异步回调的方式来获取查询的结果。使用Rx，我们可以通过Rx中的FromAsyncPattern方法由异步方法创建一个Iobservable集合。

```
Func<SearchRequest, IObservable<SearchResponse>> observableSearchFunc =
    Observable.FromAsyncPattern<SearchRequest, SearchResponse>(client.BeginSearch,
client.EndSearch);
return observableSearchFunc(searchRequest)
    .SelectMany(response => GetUrlsFromSearchResult(response)
    .ToObservable());
```

我们的目标是使用一个方法返回一个未来的SearchResponse的数据集合。我们可以从函数的返回类型看出来：

Func<SearchRequest, IObservable<SearchResponse>>

Rx的FromAsyncPattern将Begin/End对转换为一个单独的方法，并返回一个Func类型。我们通过给函数一个名词，然后将Observable.FromAsyncPattern的结果赋给该Func。

Func<SearchRequest, IObservable<SearchResponse>> observableSearchFunc =

Observable.FromAsyncPattern<……>(……);

如果我们查看非异步的WebService调用，我们可以发现方法传入SearchRequest，返回**SearchResponse**。这些成为了FromAsyncPattern的类型参数。

Func<SearchRequest, IObservable<SearchResponse>> observableSearchFunc =

Observable.FromAsyncPattern<**SearchRequest, SearchResponse**>(……);

然后我们查看异步的Webservice调用方法，我们会看到client.BeginSearch, client.EndSearch这两个方法，然后将这两个方法作为FromAsyncPattern的参数。

Func<SearchRequest, IObservable<SearchResponse>> observableSearchFunc =

Observable.FromAsyncPattern<**SearchRequest, SearchResponse**>(
client.BeginSearch, client.EndSearch);

我们可以这样理解该方法observableSearchFunc是一个Func类型，该类型接受一个SearchRequest类型参数，返回IObservable<SearchResponse> 类型。然后通过调用接受SearchRequest和SearchResponse两个泛型参数的Observable.FromAsyncPattern方法接受client.BeginSearch, client.EndSearch产生。

由于查询可能返回多个数据集合，所以返回的是一个Observable集合。然后我们对返回的Observable数据集进行SelectMany操作

```
return observableSearchFunc(searchRequest)
        .SelectMany(response => GetUrlsFromSearchResult (response)
        .ToObservable());
```

b. 实现GetUrlsFromSearchResult方法

```
private String[] GetUrlsFromSearchResult(SearchResponse response)
{
    return response.Image.Results.Select(x => x.Thumbnail.Url).ToArray();
}
```

我们也可以直接返回一个Observable集合，将上面的方法改为：

```
return observableSearchFunc(searchRequest)
    .SelectMany(response => GetUrlsFromSearchResult(response));
private IObservable<String> GetUrlsFromSearchResult(SearchResponse response)
{
    return response.Image.Results.Select(x => x.Thumbnail.Url).ToObservable();
}
```

这两种方式产生的结果是一致的。方法返回的结果是缩略图的url。

c.为按钮绑定事件

在这个例子中，我们通过在MainWindows的构造函数中创建一个Subject对象来实现Button的点击事件。

```
Subject<RoutedEventArgs> clickObservable = new Subject<RoutedEventArgs>();
SearchButton.Click += (o, e) => clickObservable.OnNext(e);
```

Subject对象和Observable对象除了下面两个重要的区别之外，他们很相似。

- Subject也是一种Observer,它能够观察其他对象也能够被观察
- 我们可以通过编程的方式控制subject对象。也就是说，我们可以通过一个方法调用，使得subject的内容能够被其他观察者所使用。我们能够手动的通过调用OnNext方法发布Subject中的内容。例如每当我们调用Subject的OnNext方法时，所有注册该对象的方法就会收到这一对象。

上面的代码第一行创建了一个名为clickObservable的以RoutedEventArgs为泛型参数的Subject对象。第二行将该对象的OnNext方法注册到button的click事件上。

我们通过事件获取textbox中用户输入的内容，然后将用户查询的内容输出到控制台上：

```
IObservable<String> searchTermObservable = clickObservable
    .Select(args => SearchBox.Text)
    .Where(term => String.IsNullOrWhiteSpace(term) == false);
searchTermObservable.Subscribe(term => Console.WriteLine(@"Search for '{0}'", term));
```

第一行代码通过clickObservable对象查询SearchBox的Text属性产生一个IObservable对象。第二行将用户查询的内容输出到控制台上。

上面所做的其实是将用户的点击流转为查询内容流，然后我们构造一个Observable对象来在查询框中部为空时通知注册的用户。

这里关键在于我们可以从一些列事件中找出我们需要关注的时间。在这里我们关注的事件时“用户想搜索一些东西”—我们需要的是一个简单的事件以及一些有用的关于查询内容的信息。Rx使得我们能够在更高的抽象层面上将我们关心的这些事件和信息组合起来。

d. 实现CreateImageFromUrl方法

为了在ListBox中显示查询到的结果图片集，我们需要创建一个ObservableCollection<BitmapImage>对象。

然后再主窗体构造函数中初始化该属性：

```
public ObservableCollection<BitmapImage> ImagesToDisplay { get; protected set; }
```

最后在点击查询事件中清空该集合以准备存储查询到的集合。

```
ImagesToDisplay = new ObservableCollection<BitmapImage>();
searchTermObservable.Subscribe(term =>
    {
        Console.WriteLine(@"Search for '{0}'", term);
        ImagesToDisplay.Clear();
    });
```

然后完成CreateImageFromUrl函数的主体：

```
BitmapImage CreateImageFromUrl(String url)
{
    return new BitmapImage(new Uri(url));
}
```

最后在构造函数中，通过SearchBox输入查询关键字产生的集合产生一个新的IObservable结果集合：

```
IObservable<BitmapImage> bitmapImagesToAdd = searchTermObservable
    .SelectMany(term => SearchBingImageApi(term))
    .ObserveOnDispatcher()
    .Select(url => CreateImageFromUrl(url));
bitmapImagesToAdd.Subscribe(images => ImagesToDisplay.Add(images));
```

注意到ObserveOnDispatcher方法与前篇文章中的目的一样，就是强制更新UI线程。

到此为止，所有后台逻辑代码已经完成。现在回头将前台代码与后面的集合对象进行绑定：

```
<Window x:Class="BingImageSearch.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" x:Name="Window" Height="568" Width="640">
    <Window.Resources>
        <ItemsPanelTemplate x:Key="ItemsPanelTemplate">
            <WrapPanel Orientation="Horizontal" IsItemsHost="True" />
        </ItemsPanelTemplate>
        <DataTemplate x:Key="DataTemplate">
            <Image Source="{Binding}" Width="128" Height="128"  Margin="8" />
        </DataTemplate>
    </Window.Resources>


    <Grid  Margin="16">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <TextBox x:Name="SearchBox" />
        <Button x:Name="SearchButton" Grid.Column="1" Content="Search" Margin="8,0,0,0"
Width="50" />
        <ListBox x:Name="Results" Grid.Row="1" Grid.ColumnSpan="2" Margin="0,8,0,0"
                ScrollViewer.HorizontalScrollBarVisibility="Disabled"
                 ItemsSource="{Binding  ImagesToDisplay,ElementName=Window}"
                 ItemsPanel="{DynamicResource  ItemsPanelTemplate}"
                 ItemTemplate="{DynamicResource DataTemplate}"/>
    </Grid>
</Window>
```

前台代码中，我们将ListBox绑定到ItemsSource，ItemsPanel和ItemsTemplate，并将它们分别绑定到ImagesToDisplay，ItemsPanelTemplate和DataTemplate资源文件上。

下面是所有后台代码：

MainWindow.xaml.cs

BingImageSearch.MainWindow

```
26  public partial class MainWindow : Window
27  {
28      public ObservableCollection<BitmapImage> ImagesToDisplay { get; set; }
29      public MainWindow()
30      {
31          ImagesToDisplay = new ObservableCollection<BitmapImage>();
32
33          InitializeComponent();
34
35          Subject<RoutedEventArgs> clickObservable = new Subject<RoutedEventArgs>();
36          SearchButton.Click += (o, e) => clickObservable.OnNext(e);
37
38          IObservable<String> searchTermObservable = clickObservable
39              .Select(args => SearchBox.Text)
40              .Where(term => String.IsNullOrWhiteSpace(term) == false);
41
42          searchTermObservable.Subscribe(term =>
43              {
44                  Console.WriteLine(@"Search for '{0}'", term);
45                  ImagesToDisplay.Clear();
46              });
47
48          IObservable<BitmapImage> bitmapImagesToAdd = searchTermObservable
49              .SelectMany(term => SearchBingImageApi(term))
50              .ObserveOnDispatcher()
51              .Select(url => CreateImageFromUrl(url));
52
53          bitmapImagesToAdd.Subscribe(images => ImagesToDisplay.Add(images));
54      }
55
56      IObservable<String> SearchBingImageApi(String query)
57      {
58          SearchRequest searchRequest = new SearchRequest();
59          searchRequest.AppId = App.BingSearchKey;
60          searchRequest.Query = query;
61          searchRequest.Sources = new SourceType[1];
62          searchRequest.Sources[0] = SourceType.Image;
63          BingPortTypeClient client = new BingPortTypeClient();
64          Func<SearchRequest, IObservable<SearchResponse>> observableSearchFunc =
65              Observable.FromAsyncPattern<SearchRequest, SearchResponse>(client.BeginSearch, client.EndSearch);
66          return observableSearchFunc(searchRequest)
67              .SelectMany(response => GetUrlsFromSearchResult(response));
68      }
69
70      private IObservable<String> GetUrlsFromSearchResult(SearchResponse response)
71      {
72          return response.Image.Results.Select(x => x.Thumbnail.Url).ToObservable();
73      }
74
75      BitmapImage CreateImageFromUrl(String url)
76      {
77          return new BitmapImage(new Uri(url));
78      }
79  }
80  }
```


从上面的代码可以看出Rx的FromAsyncPattern使得整个代码的逻辑更加清晰，可读性更强。下面我们使用传统的Begin/End异步回调的方式来实现这一逻辑。您可以对比看一下他们之间的区别。

3. 实例：使用Begin/End异步方式实现

使用Rx方式来进行异步操作可以使得代码更加容易维护，现在我们使用传统的异步回调的方式来实现上面的逻辑，对比我们可以看到他们之间的差别。

传统的方法中SearchBingImageAPI方法需要接受第二个参数，那就是一个委托类型的回调方法。

```
private void SearchBingImageApi(String query, Action<String[]> callBack)
```

接下来实例化searchRequest对象。

```
SearchRequest searchRequest = new SearchRequest();
searchRequest.AppId = App.BingSearchKey;
searchRequest.Query = query;
searchRequest.Sources = new SourceType[1];
searchRequest.Sources[0] = SourceType.Image;
BingPortTypeClient client = new BingPortTypeClient();
```

然后开始调用BeginSearch方法，在传统的Begin/End模式中，方法接受三个参数：一个SearchRequest参数，一个Callback方法，一个AsyncState对象。我们以内联的方式定义回调方法, AsyncState我们传进去null。

```
client.BeginSearch(searchRequest,
                   (asyncResult) =>
                   {
                       var result = client.EndSearch(asyncResult);
                       callBack(GetUrlsFromSearchResult(result));
                   }, null);
```

当方法完成时，调用GetUrlsFromSearchResult方法。

上面只是一个Begin/End回调，而且这里采用的是内联匿名的方式减少了代码量和破碎度。可以看到传统的方式很难维护和理解。尤其是当回调方法中如果再一步回调其他方法的话，整个代码就会变得难以控制。

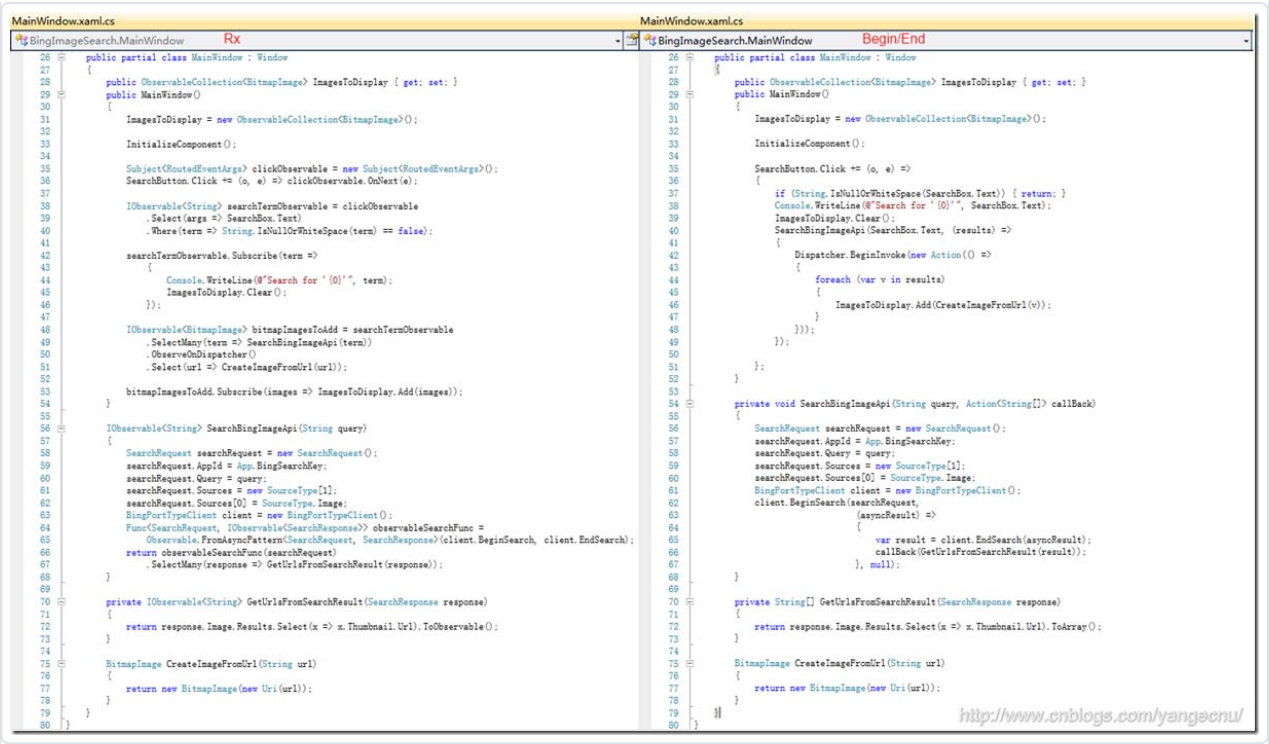
下面是使用传统的Begin/End方式实现上面逻辑的代码。

MainWindow.xaml.cs

BingImageSearch.MainWindow

```
26  public partial class MainWindow : Window
27  {
28      public ObservableCollection<BitmapImage> ImagesToDisplay { get; set; }
29      public MainWindow()
30      {
31          ImagesToDisplay = new ObservableCollection<BitmapImage>();
32
33          InitializeComponent();
34
35          SearchButton.Click += (o, e) =>
36          {
37              if (String.IsNullOrWhiteSpace(SearchBox.Text)) { return; }
38              Console.WriteLine(@"Search for '{0}'", SearchBox.Text);
39              ImagesToDisplay.Clear();
40              SearchBingImageApi(SearchBox.Text, (results) =>
41              {
42                  Dispatcher.BeginInvoke(new Action(() =>
43                  {
44                      foreach (var v in results)
45                      {
46                          ImagesToDisplay.Add(CreateImageFromUrl(v));
47                      }
48                  }));
49              });
50
51          };
52      }
53
54      private void SearchBingImageApi(String query, Action<String[]> callBack)
55      {
56          SearchRequest searchRequest = new SearchRequest();
57          searchRequest.AppId = App.BingSearchKey;
58          searchRequest.Query = query;
59          searchRequest.Sources = new SourceType[1];
60          searchRequest.Sources[0] = SourceType.Image;
61          BingPortTypeClient client = new BingPortTypeClient();
62          client.BeginSearch(searchRequest,
63                             (asyncResult) =>
64                             {
65                                 var result = client.EndSearch(asyncResult);
66                                 callBack(GetUrlsFromSearchResult(result));
67                             }, null);
68      }
69
70      private String[] GetUrlsFromSearchResult(SearchResponse response)
71      {
72          return response.Image.Results.Select(x => x.Thumbnail.Url).ToArray();
73      }
74
75      BitmapImage CreateImageFromUrl(String url)
76      {
77          return new BitmapImage(new Uri(url));
78      }
79  }
80  }
```


下面是两种方法代码的比较：



4. 结语

本文在前文的基础上进一步讨论了Rx中的一些操作符并以一个例子展示了Rx在异步方法调用中的作用。本文开始以一个Bing搜索图片的例子展示了如何使用Rx中的FromAsyncPattern来简化异步操作，然后使用传统的Begin/End方式实现了这一部分逻辑，对比可以看出Rx使得异步操作的代码更加可读和维护性更换好，本文所有的代码都在文中贴出来了，您可以自己拷贝到程序中测试，需要注意的是Bing API Key需要您自己到Bing开发者中心申请一个Key，替换文中的Key。

希望文章对于您了解Rx在异步操作中的作用有所帮助。