

# .Net 中的反应式编程 - 文章 - 伯乐在线



## 一、反应式编程(Reactive Programming)

1、什么是反应式编程：反应式编程(Reactive programming)简称Rx，他是一个使用LINQ风格编写基于观察者模式的异步编程模型。简单点说Rx = Observables + LINQ + Schedulers。

2、为什么会产生这种风格的编程模型？我在本系列文章开始的时候说过一个使用事件的例子：

```
1
2
3
4
5
6
7
8
9
var watch = new FileSystemWatcher();
    watch.Created += (s, e) =>
    {
        var fileType = Path.GetExtension(e.FullPath);
        if (fileType.ToLower() == ".jpg")
        {
            //do some thing
        }
    };
```

这个代码定义了一个FileSystemWatcher，然后在Watcher事件上注册了一个匿名函数。事件的使用是一种命令式代码风格，有没有办法写出生明性更强的代码风格？我们知道使用高阶函数可以让代码更具声明性，整个LINQ扩展就是一个高阶函数库，常见的LINQ风格代码如下：

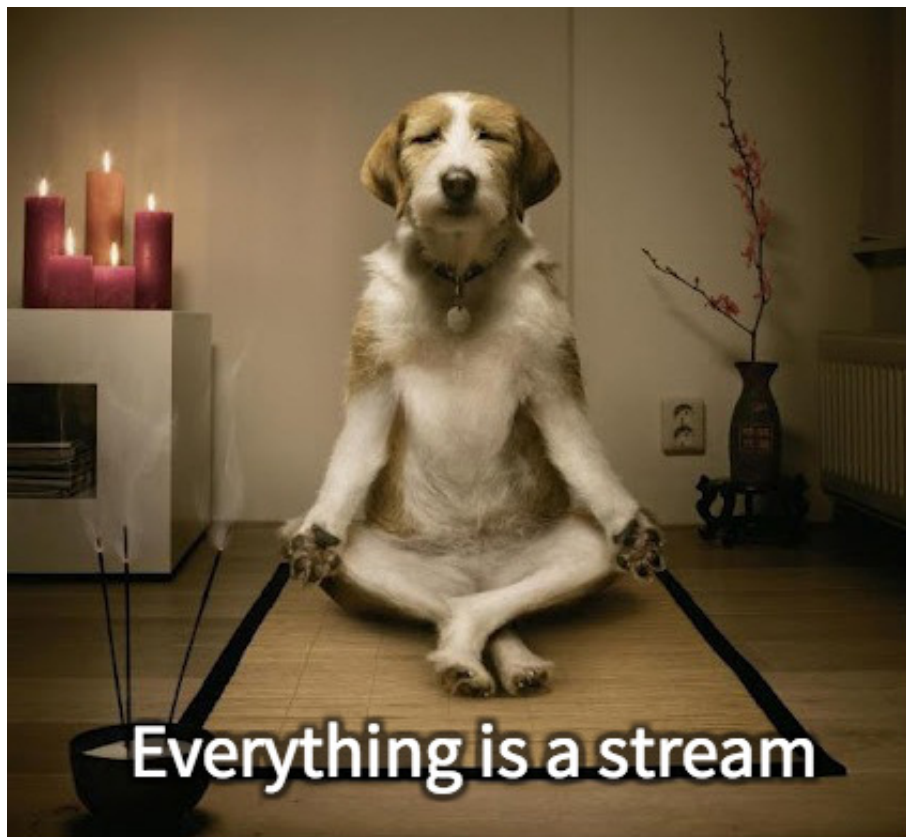
```
1
2
3
4
var list = Enumerable.Range(1, 10)
    .Where(x => x > 8)
```

```
.Select(x => x.ToString())  
.First();
```

能否使用这样的风格来编写事件呢？

### 3、事件流

LINQ是对IEnumerable的一系列扩展方法，我们可以简单的将IEnumerable认为是一个集合。当我们将事件放在一个时间范围内，事件也变成了集合。我们可以将这个事件集合理解为事件流。



事件流的出现给了我们一个能够对事件进行LINQ操作的灵感。

## 二、反应式编程中的两个重要类型

事件模型从本质上来说是观察者模式，所以IObservable和IObserver也是该模型的重头戏。让我们来看看这两个接口的定义：

```
1  
2  
3  
4  
5  
  
public interface IObservable  
{  
    //Notifies the provider that an observer is to receive notifications.  
    IDisposable Subscribe(IObserver observer);  
}  
  
public interface IObserver
```

```

{
    //Notifies the observer that the provider has finished sending push-based
    notifications.

    void OnCompleted();
    //Notifies the observer that the provider has experienced an error
    condition.

    void OnError(Exception error);
    //Provides the observer with new data.
    void OnNext(T value);
}

```

这两个名称准确的反应出了它两的职责：IObservable-可观察的事物，IObserver-观察者。

IObservable只有一个方法Subscribe(IObserver observer)，此方法用来对事件流注册一个观察者。

IObserver有三个回调方法。当事件流中有新的事件产生的时候会回调OnNext(T value)，观察者会得到事件中的数据。OnCompleted()和OnError(Exception error)则分别用来通知观察者事件流已结束，事件流发生错误。

显然事件流是可观察的事物，我们用Rx改写上面的例子：

```

1
2
3
4
5
6
Observable.FromEventPattern(watch, "Created")
    .Where(e =>
Path.GetExtension(e.EventArgs.FullPath).ToLower() == ".jpg")
    .Subscribe(e =>
    {
        //do some thing
    });

```

注：在.net下使用Rx编程需要安装以下Nuget组件：

```

1
Install-Package Rx-main

```

### 三、UI编程中使用Rx

Rx模型不但使得代码更加具有声明性，Rx还可以用在UI编程中。

#### 1、UI编程中的第一段Rx代码

为了简单的展示如何在UI编程中使用Rx，我们以Winform中的Button为例，看看事件模型和Rx有何不同。

```
1
2
3
4
5
6
7
8
9

private void BindFirstGroupButtons()
{
    btnFirstEventMode.Click += btnFirstEventMode_Click;
}

void btnFirstEventMode_Click(object sender, EventArgs e)
{
    MessageBox.Show("hello world");
}
```

添加了一个Button，点击Button的时候弹出一个对话框。使用Rx做同样的实现：

```
1
2
3
4

//得到了Button的Click事件流。
var clickedStream =
Observable.FromEventPattern(btnFirstReactiveMode, "Click");
//在事件流上注册了一个观察者。
clickedStream.Subscribe(e => MessageBox.Show("Hello world"));
```

有朋友指出字符串“Click”非常让人不爽，这确实是个问题。由于Click是一个event类型，无法用表达式树获取其名称，最终我想到使用扩展方法来实现：

```
1
2
3
4
5
6
7
8
9

public static IObservable<> FromClickEventPattern(this Button button)
```

```

    {
        return Observable.FromEventPattern(button, "Click");
    }
    public static IObservable> FromDoubleClickEventPattern(this Button button)
    {
        return Observable.FromEventPattern(button, "DoubleClick");
    }

```

我们平时常用的事件类型也就那么几个，可以暂时通过这种方案来实现，该方案算不上完美，但是比起直接使用字符串又能优雅不少。

```

1
2
btnFirstReactiveMode.FromClickEventPattern()
    .Subscribe(e => MessageBox.Show("hello world"));

```

2、UI编程中存在一个很常见的场景：当一个事件的注册者阻塞了线程时，整个界面都处于假死状态。.net中的异步模型也从APM，EAP，TPL不断演化直至async/await模型的出现才使得异步编程更加简单易用。我们来看看界面假死的代码：

```

1
2
3
4
5
6
void btnSecondEventMode_Click(object sender, EventArgs e)
{
    btnSecondEventMode.BackColor = Color.Coral;
    Thread.Sleep(2000);
    lblMessage.Text = "event mode";
}

```

Thread.Sleep(2000);模拟了一个长时间的操作，当你点下Button时整个界面处于假死状态并且此时的程序无法响应其他的界面事件。传统的解决方案是使用多线程来解决假死：

```

1
2
3
4
5
6
7
8
BtnSecondEventAsyncModel.BackColor = Color.Coral;
Task.Run(() =>

```

```

    {
        Thread.Sleep(2000);
        Action showMessage = () => lblMessage.Text = "async event
mode";

        lblMessage.Invoke(showMessage);
    });

```

这个代码的复杂点在于：普通的多线程无法对UI进行操作，在Winform中需要用

Control.BeginInvoke(Action action)经过包装后，多线程中的UI操作才能正确执行，WPF则使用Dispatcher.BeginInvoke(Action action)包装。

Rx方案：

```

btnSecondReactiveMode.FromClickEventPattern()
    .Subscribe(e =>
    {
        Observable.Start(() =>
        {
            btnSecondReactiveMode.BackColor =
Color.Coral;

            Thread.Sleep(2000);
            return "reactive mode";
        })
        .SubscribeOn(ThreadPoolScheduler.Instance)
        .ObserveOn(this)
        .Subscribe(x =>
        {
            lblMessage.Text = x;
        })
    });

```

一句SubscribeOn(ThreadPoolScheduler.Instance)将费时的操作跑在了新线程中，ObserveOn(this)让后面的观察者跑在了UI线程中。

注：使用ObserveOn(this)需要使用Rx-WinForms

1

Install-Package Rx-WinForms

这个例子虽然成功了，但是并没有比BeginInvoke(Action action)的方案有明显的进步之处。在一个事件流中再次使用Observable.Start()开启新的观察者让人更加摸不着头脑。这并不是Rx的问题，而是事件模型在UI编程中存在局限性：不方便使用异步，不具备可测试性等。以XMAL和MVVM为核心的UI编程模型将在未来处于主导地位，由于在MVVM中可以将UI绑定到一个Command，从而解耦了事件模型。

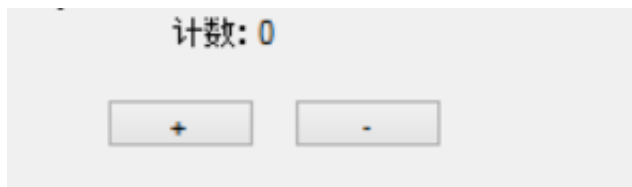
开源项目[ReactiveUI](http://reactiveui.net)提供了一个以Rx基础的UI编程方案，可以使用在XMAL和MVVM为核心的UI编程中，例如：Xamarin, WFP, Windows Phone8等开发中。

注：在WPF中使用ObserveOn() 需要安装Rx-WPF

1

Install-Package Rx-WPF

3、再来一个例子，让我们感受一下Rx的魅力



界面上有两个Button分别为+和-操作，点击+按钮则+1，点击-按钮则-1，最终的结果显示在一个Label中。

这样的需求使用经典事件模型只需要维护一个内部变量，两个按钮的Click事件分别对变量做加1或减1的操作即可。

Rx作为一种函数式编程模型讲求immutable-不可变性，即不使用变量来维护内部状态。

1

2

3

4

5

6

7

8

```
var increasedEventStream = btnIncrease.ClickEventPattern()
    .Select(_ => 1);
var decreasedEventStream = btnDecrease.ClickEventPattern()
    .Select(_ => -1);
increasedEventStream.Merge(decreasedEventStream)
    .Scan(0, (result, s) => result + s)
    .Subscribe(x => lblResult.Text = x.ToString());
```

这个例子使用了IObservable的”谓词”来对事件流做了一些操作。

- Select跟Linq操作有点类似，分别将两个按钮的事件变形为IObservable(1)和IObservable(-1)；
- Merge操作将两个事件流合并为一个；
- Scan稍显复杂，对事件流做了一个折叠操作，给定了一个初始值，并通过一个函数来对结果和下一个值进行累加；

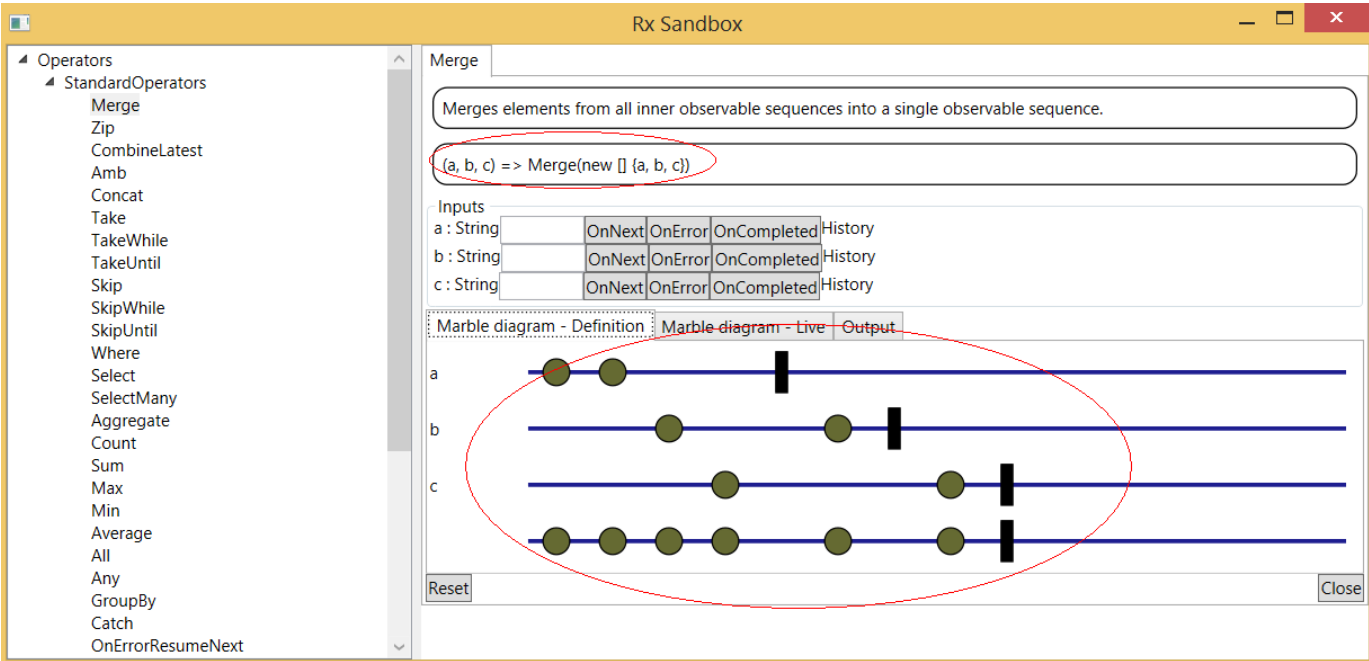
下面就让我们来看看IObservable中常用的“谓词”

## 四、IObservable中的谓词

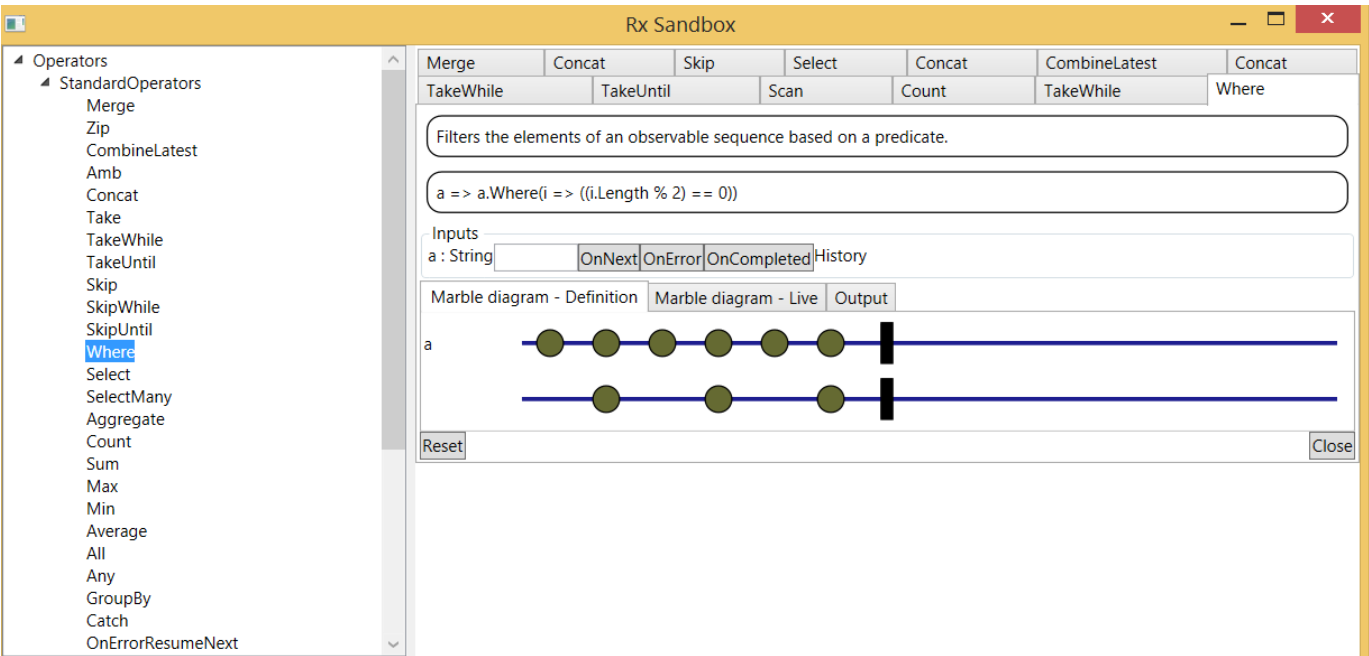
IObservable的灵感来源于LINQ，所以很多操作也跟LINQ中的操作差不多，例如Where、First、Last、Single、Max、Any。

还有一些“谓词”则是新出现的，例如上面提到的” Merge”、“Scan”等，为了理解这些“谓词”的含义，我们请出一个神器[RxSandbox](#)。

1、Merge操作，从下面的图中我们可以清晰的看出Merge操作将三个事件流中的事件合并在了同一个时间轴上。



2、Where操作则是根据指定的条件筛选出事件。



有了这个工具我们可以更加方便的了解这些“谓词”的用途。

## 五、IObservable的创建

Observable类提供了很多静态方法用来创建IObservable，之前的例子我们都使用FromEventPattern方法来将事件转化为IObservable，接下来再看看别的方法。

Return可以创建一个具体的IObservable:



1  
2  
3  
4  
5

```
public static void UsingReturn()
{
    var greeting = Observable.Return("Hello world");
    greeting.Subscribe(Console.WriteLine);
}
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

```
public static void UsingCreate()
{
    var greeting = Observable.Create(observer =>
    {
        observer.OnNext("Hello world");
        return Disposable.Create(() => Console.WriteLine("Observer
has unsubscribed"));
    });
    greeting.Subscribe(Console.WriteLine);
}
```

Range方法可以产生一个指定范围内的IObservable

1  
2

Generate方法是一个折叠操作的逆向操作，又称Unfold方法：

1  
2  
3  
4  
5

```
public static void UsingGenerate()
{
```

```
var range = Observable.Generate(0, x => x + 1, x => x);  
range.Subscribe(Console.WriteLine);  
}
```

Interval方法可以每隔一定时间产生一个IObservable:

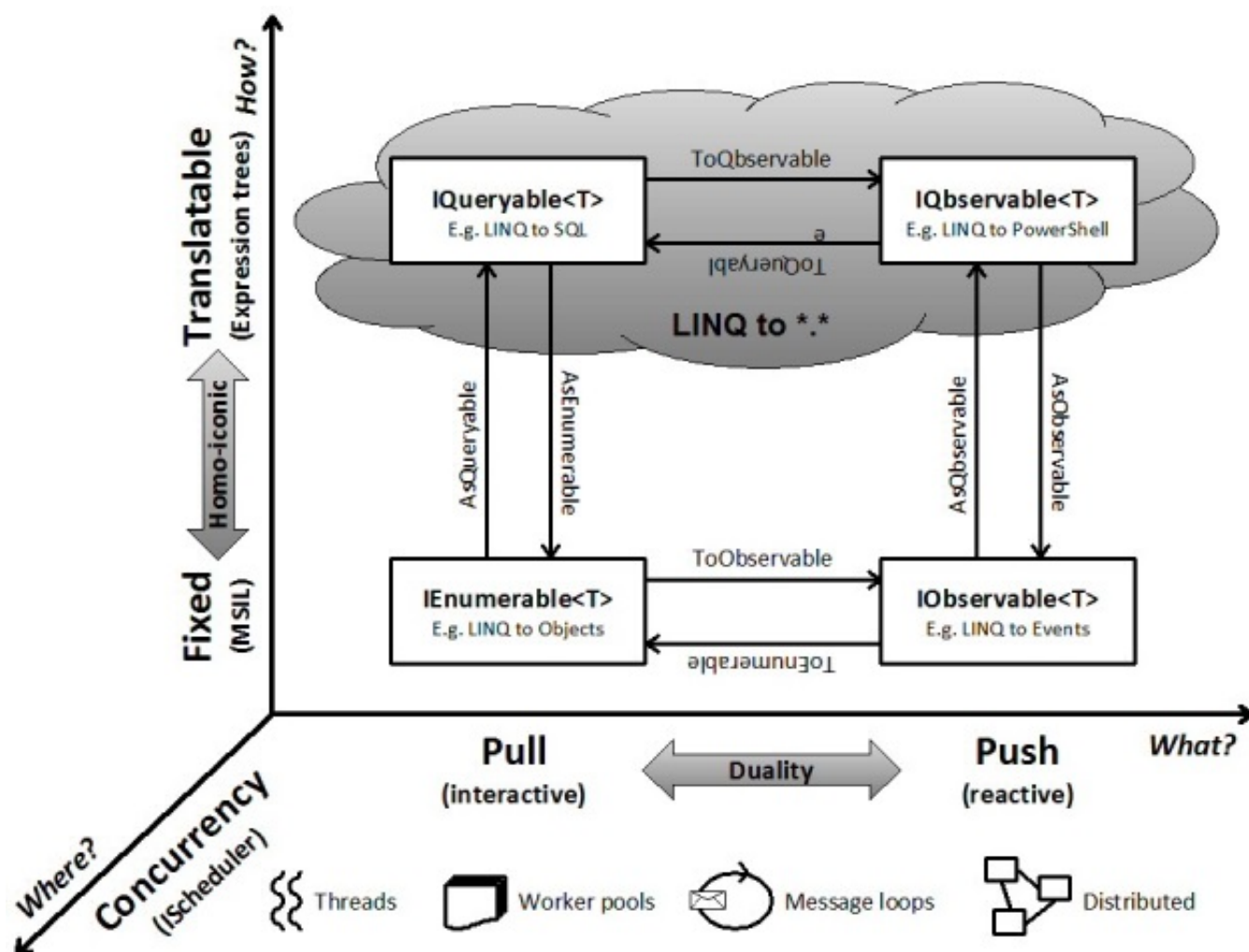
```
1  
2  
Observable.Interval(TimeSpan.FromSeconds(1))  
    .Subscribe(x => Console.WriteLine(x.ToString()));
```

Subscribe方法有一个重载，可以分别对Observable发生异常和Observable完成定义一个回调函数。

```
1  
2  
还可以将IEnumerable转化为IObservable类型:  
  
1  
2  
Enumerable.Range(1, 10).ToObservable()  
    .Subscribe(x => Console.WriteLine(x.ToString()));
```

也可以将IObservable转化为IEnumerable

```
1  
var list= Observable.Range(1, 10).ToEnumerable();
```



## 六、Scheduler

Rx的核心是观察者模式和异步，Scheduler正是为异步而生。我们在之前的例子中已经接触过一些具体的Scheduler了，那么他们都具体是做什么的呢？

1、先看下面的代码：

```

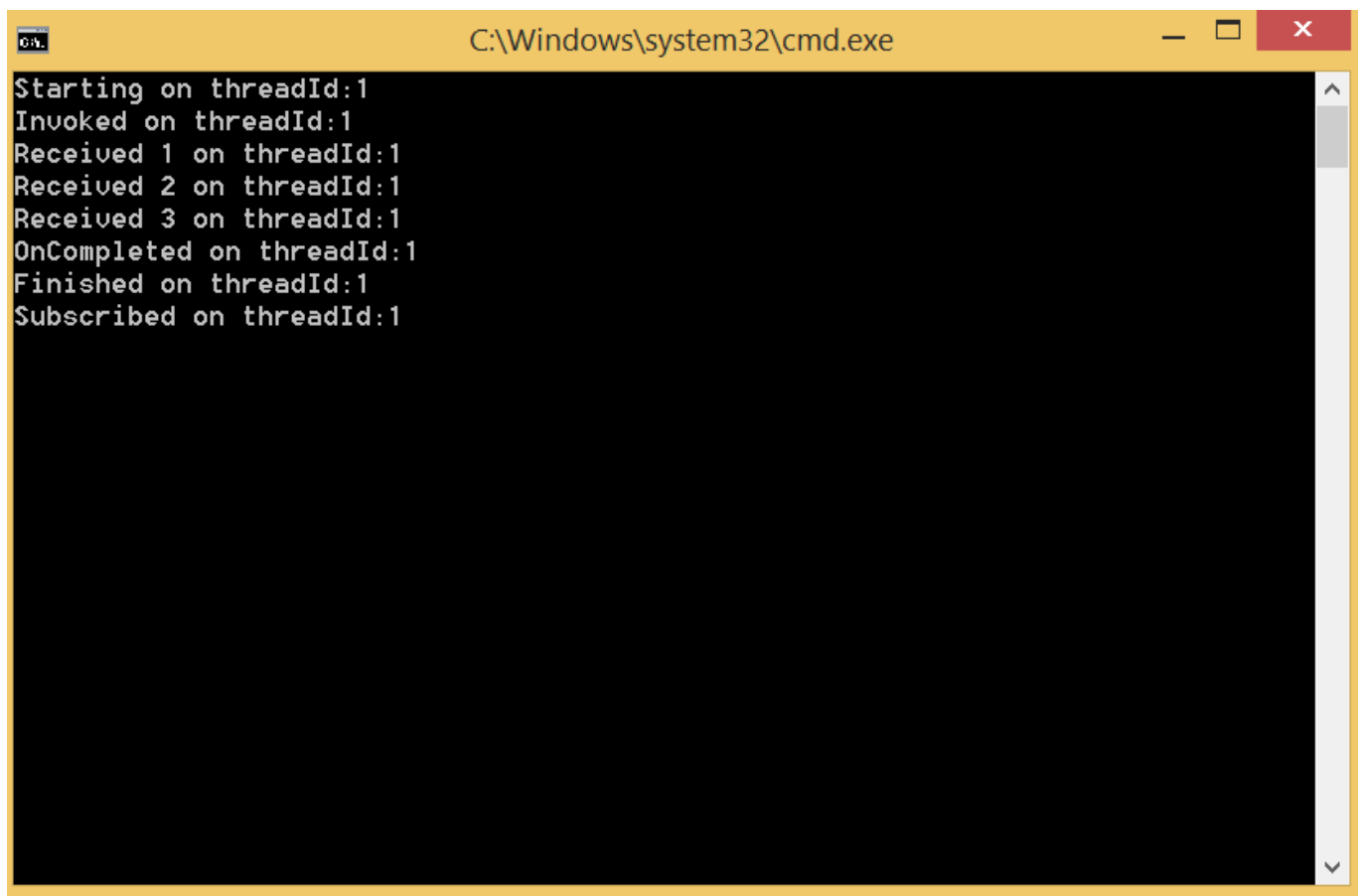
1
2
3
4
5
6
7
8
9
10
11
12
13
14

```

15  
16  
17  
18  
19  
20  
21  
22

```
public static void UsingScheduler()
{
    Console.WriteLine("Starting on threadId: {0}",
Thread.CurrentThread.ManagedThreadId);
    var source = Observable.Create(
o =>
{
        Console.WriteLine("Invoked on threadId: {0}",
Thread.CurrentThread.ManagedThreadId);
        o.OnNext(1);
        o.OnNext(2);
        o.OnNext(3);
        o.OnCompleted();
        Console.WriteLine("Finished on threadId:
{0}", Thread.CurrentThread.ManagedThreadId);
        return Disposable.Empty;
    });
    source
    //.SubscribeOn(NewThreadScheduler.Default)
    //.SubscribeOn(ThreadPoolScheduler.Instance)
    .Subscribe(
o => Console.WriteLine("Received {1} on threadId:
{0}", Thread.CurrentThread.ManagedThreadId, o),
    () => Console.WriteLine("OnCompleted on threadId:
{0}", Thread.CurrentThread.ManagedThreadId));
    Console.WriteLine("Subscribed on threadId: {0}",
Thread.CurrentThread.ManagedThreadId);
}
```

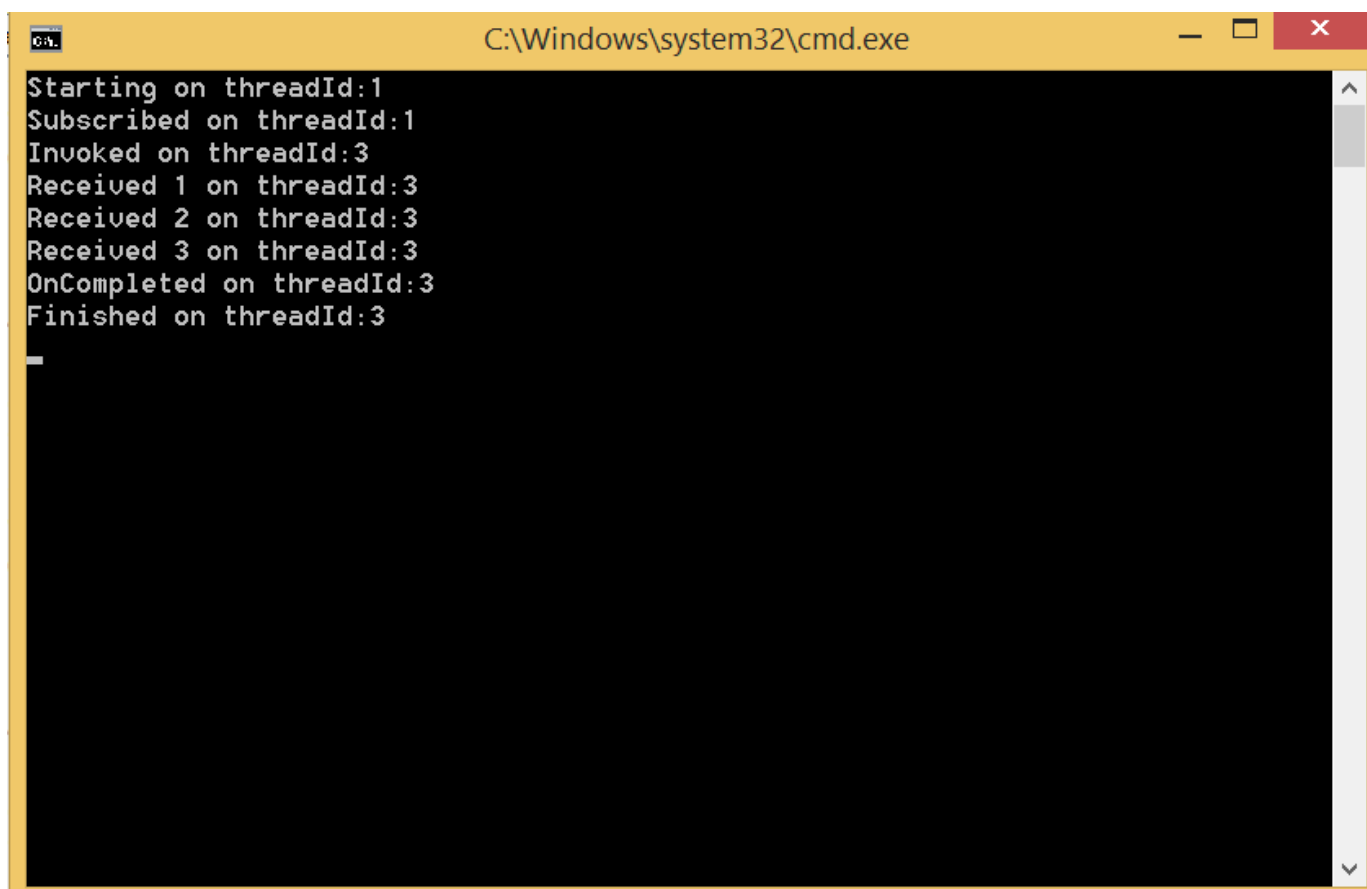
当我们不使用任何Scheduler的时候，整个Rx的观察者 and 主题都跑在主线程中，也就是说并没有异步执行。正如下面的截图，所有的操作都跑在threadId=1的线程中。



```
C:\Windows\system32\cmd.exe

Starting on threadId:1
Invoked on threadId:1
Received 1 on threadId:1
Received 2 on threadId:1
Received 3 on threadId:1
OnCompleted on threadId:1
Finished on threadId:1
Subscribed on threadId:1
```

当我们使用 `SubscribeOn(NewThreadScheduler.Default)` 或者 `SubscribeOn(ThreadPoolScheduler.Instance)` 的时候，观察者和主题都跑在了 `theadId=3` 的线程中。



```
C:\Windows\system32\cmd.exe

Starting on threadId:1
Subscribed on threadId:1
Invoked on threadId:3
Received 1 on threadId:3
Received 2 on threadId:3
Received 3 on threadId:3
OnCompleted on threadId:3
Finished on threadId:3
```

这两个 Scheduler 的区别在于：`NewThreadScheduler` 用于执行一个长时间的操作，`ThreadPoolScheduler` 用来执行短时间的操作。

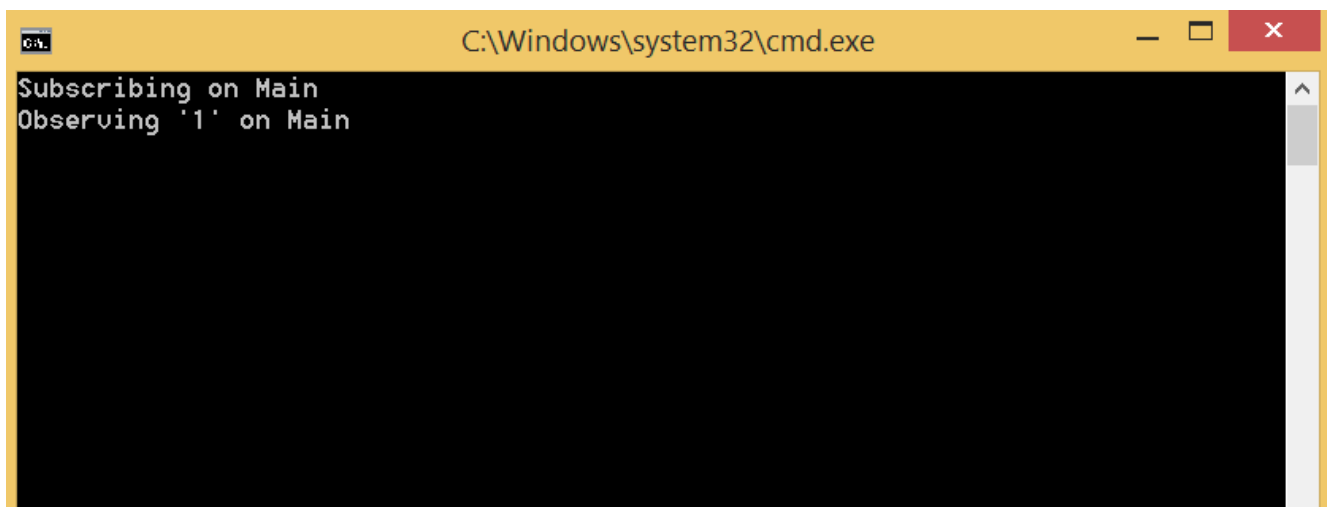
## 2、SubscribeOn和ObserveOn的区别

上面的例子仅仅展示了SubscribeOn()方法，Rx中还有一个ObserveOn()方法。stackoverflow上有一个这样的问题：[What's the difference between SubscribeOn and ObserveOn](#)，其中一个简单的例子很好的诠释了区别。

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

    public static void DifferenceBetweenSubscribeOnAndObserveOn()
    {
        Thread.CurrentThread.Name = "Main";
        IScheduler thread1 = new NewThreadScheduler(x => new Thread(x) {
Name = "Thread1" });
        IScheduler thread2 = new NewThreadScheduler(x => new Thread(x) {
Name = "Thread2" });
        Observable.Create(o =>
        {
            Console.WriteLine("Subscribing on " +
Thread.CurrentThread.Name);
            o.OnNext(1);
            return Disposable.Create(() => { });
        })
        //.SubscribeOn(thread1)
        //.ObserveOn(thread2)
        .Subscribe(x => Console.WriteLine("Observing '" + x + "' on " +
Thread.CurrentThread.Name));
    }
```

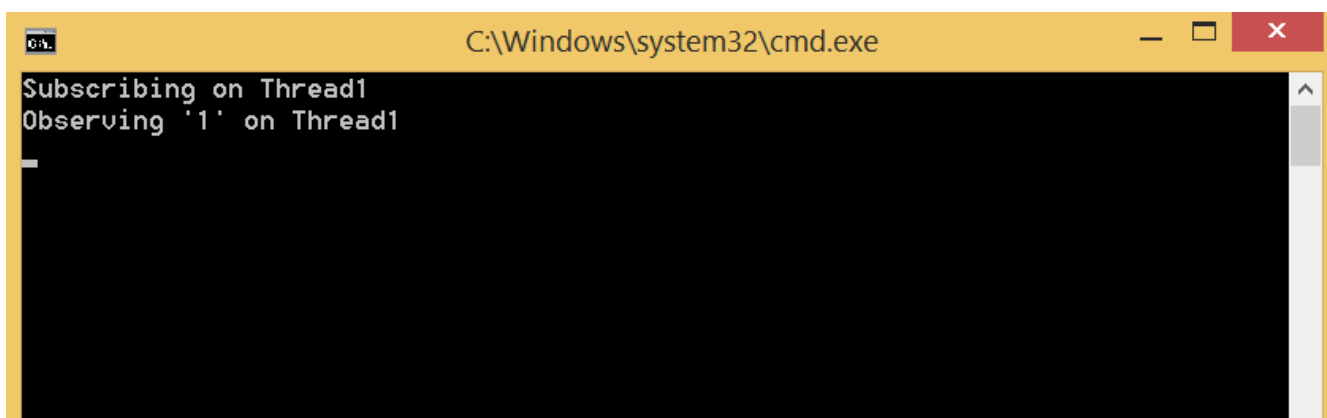
- 当我们注释掉：SubscribeOn(thread1)和ObserveOn(thread2)时的结果如下：



```
C:\Windows\system32\cmd.exe
Subscribing on Main
Observing '1' on Main
```

观察者和主题都跑在name为Main的thread中。

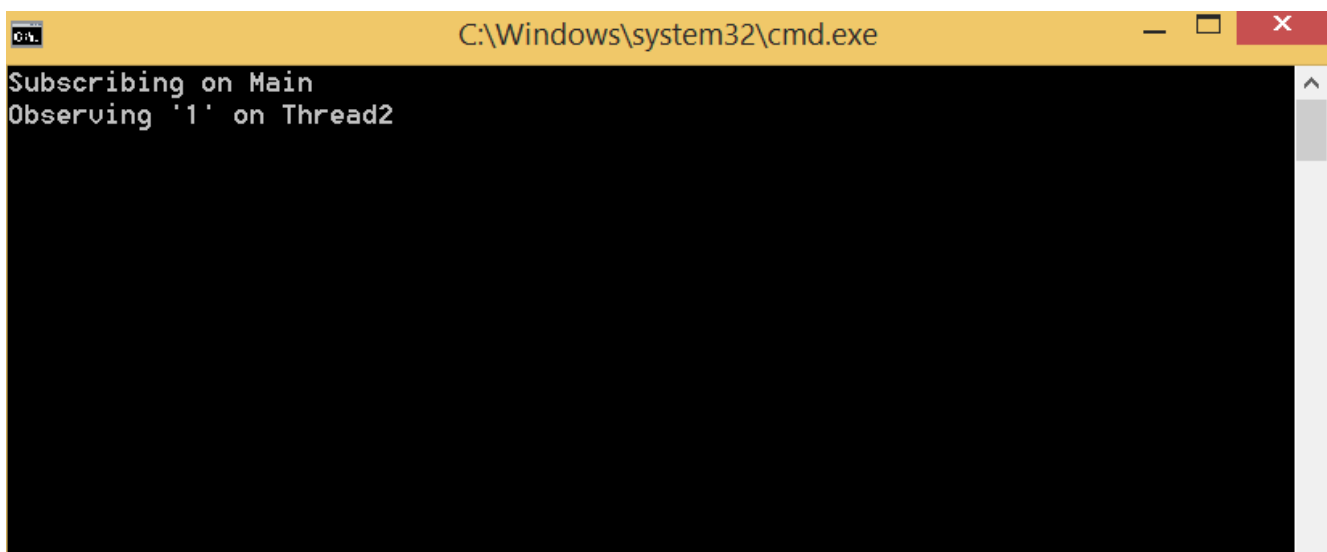
- 当我们放开SubscribeOn(thread1):



```
C:\Windows\system32\cmd.exe
Subscribing on Thread1
Observing '1' on Thread1
```

主题和观察者都跑在了name为Thread1的线程中

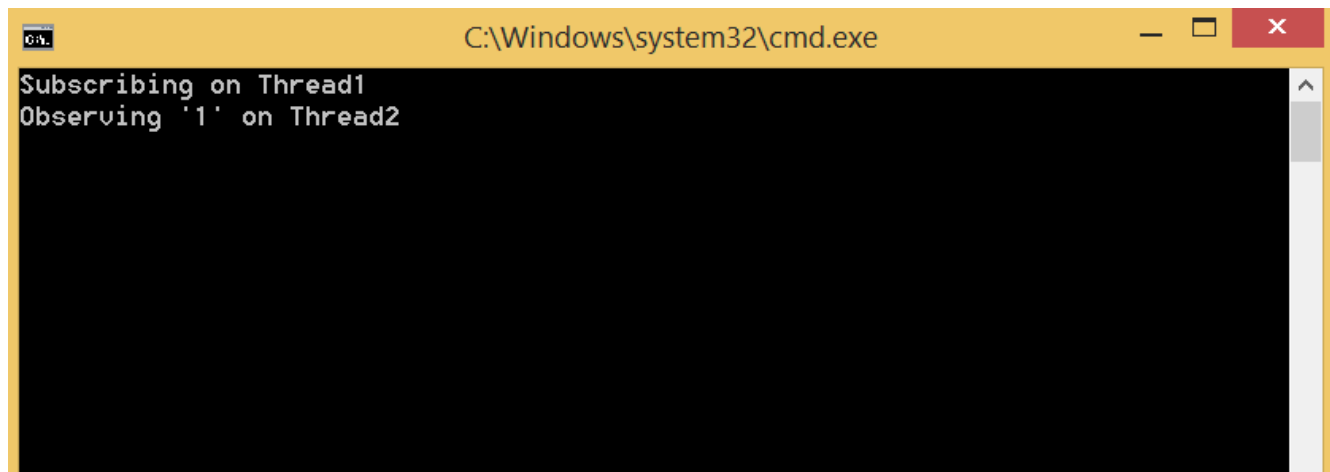
- 当我们注释掉: SubscribeOn(thread1), 放开ObserveOn(thread2)时的结果如下:



```
C:\Windows\system32\cmd.exe
Subscribing on Main
Observing '1' on Thread2
```

主题跑在name为Main的主线程中, 观察者跑在了name=Thread2的线程中。

- 当我们同时放开SubscribeOn(thread1)和ObserveOn(thread2)时的结果如下:



```
C:\Windows\system32\cmd.exe
Subscribing on Thread1
Observing '1' on Thread2
```

主题跑在name为Thread1的线程中，观察者跑在了name为Thread2的线程中。

至此结论应该非常清晰了：SubscribeOn() 和ObserveOn() 分别控制着主题和观察者的异步。

## 七、其他Rx资源

除了.net中的Rx.net，其他语言也纷纷推出了自己的Rx框架。

- [RxJS](#): Javascript中的Rx
- [RxCpp](#): C++中的Rx
- [Rx.rb](#): Ruby中的Rx
- [RxPy](#): python中的Rx

参考资源：