Reactive Extensions (Rx) 入门(1) —— Reactive Extensions 概要

blog.csdn.net/fangxing80/article/details/7381619

原文：http://www.atmarkit.co.jp/fdotnet/introrx/introrx_01/introrx_01_01.html

作者：河合 宜文

众所周知，从 C# 3.0 开始 Linq 大大改变了以往的编程风格，尤其是大幅度简化了大量数据加工这样麻烦的操作。对于各种数据(比如：数组，XML，SQL数据库等)提供了一种统一的语法，也是 Linq 的一个特征。

Reactive Extensions (下面简称 Rx) 是在 Linq 可操作的数据源上针对 "异步" (BeginXXX/EndXXX) 和 "事件"(XXXCompleted) 上的扩展，也可以被称为 "Linq To Asynchronous" 和 " Linq To Events"。相比以前复杂的异步处理或者事件处理，Timer的处理等，结合Linq 形式的Rx编程模型更加简洁。

支持 Rx 的平台有：

- .NET Framework 3.5
- .NET Framework 4.0 Client Profile
- Silverlight 3
- Silverlight 4
- Windows Phone 7
- XNA 4.0

尤其是像 Silverlight 和 Windows Phone 7 这样只提供异步API 的平台上更能发挥价值。而且，还有面向 JavaScript 的 "RxJS" 库，可以使得 JavaScript 中也能像在 .NET Framework 中一样，使用 Rx 强大的异步或者事件的编程模型。

Reactive Extenstions (Rx) 的历史

Rx 是在 2009年11月18日，微软的研发部门在 Microsoft DevLabs 上发布的实验性项目，之后1年半里又陆陆续续发布了一些试用版。另外也会因为某个实验项目可能会升级成正式的产品而中止在实验性网站上的发布。Rx 就是在 2011年1月21日被认定为正式的 API，此后该项目的主页就转移到了 Data Developer Center 中去了。其实在发布正式的 API 之前，Rx里主要的接口 IObservable<T> 和 IObserver<T> 就已经集成在 .NET Framework 4/ Silverlight 4 里了，而在 Windows Phone 7 SDK 里在 Microsoft.Phone.Reactive 命名空间下也已经集成进来了。

Interface 已经在 .NET 的类库里集成了，而且也是作为 Windows Phone 7 的标配API，虽然抱着 ".NET Framework 4.5"里是否会发布呢？" 这样的期待，但是现在作为 Developer Preview 发布的 Visual Studio 11 里的 .NET Framework 4.5 里并没有集成 Rx。但不管怎

么说，因为它已经不是 DevLabs 的实验项目而是正式项目了，微软是不会停止对于 Rx 框架的维护，所以放心使用也没有问题。

Reactive Extenstions (Rx) 可以做的事

Rx 可以处理多种内容，而异步和事件是最基本，其他的还有定时处理等 ， Linq to Objects 通常操作的集合对象(实现 IEnumerable<T> 接口的对象等)也可以被操作。
为什么说这是可能的呢？打个比方：想象 MouseClick 事件，点击鼠标，1秒后再点击一次，然后3秒后再点击一次。根据点击过程中的 MouseEventArgs 对象，被点击坐标等的值，可以看成是分布在时间轴上的点。
那么关于时间的处理，例如 Timer 会是怎么样呢？它可以看成是指定时间间隔发生的值。
异步处理又是怎么样的呢？这种场合则可以看成是在某个时间点上开始进行处理，而处理结束时得到某个值。
数组如何？这种场合也可以看成是一瞬间，例如：0.0000001秒间隔发生的数值。

因此，Rx 是一个基于时间轴的异步编程模型，在这基础上可以通过Linq进行映射，过滤和合成等。

Rx 和 Linq

Rx 中最基本的接口是 IObservable<T>，它与 Linq 中常用的IEnumerable<T> 是不一样的。但和以往使用的Linq是一样的，并能使用一样的结构进行Query。

```
1. using System.Reactive.Linq;
2. // LINQ to Objects的Query写法
3. var ix = from x in Enumerable.Range(1, 10)
4.          where x % 2 == 0
5.          select x * x;
6. // Rx的Query写法
7. var rx = from x in Observable.Range(1, 10)
8.          where x % 2 == 0
9.          select x * x;
```

为了解决更多的问题，Rx 里追加了不少新的方法。虽然不能说全部都一样，但大部分的同名方法都和原来一样，这大大降低了学习的成本。
虽然是完全不同的接口，但都可以使用 Linq 表达式来实现相同的效果。这就是 IObservable<T> / IObserver<T> 和 IEnumerable<T> / IEnumerator<T>，
你可以认为前者是后者的反转。

下面的代码中说明 IObserver<T> 接口是如何反转 IEnumerator<T> 接口的：
(换句话说：IEnumerator<T> 是如何演变出 IObserver<T> 的)

```
// 简化的 IEnumerator 接口
public interface IEnumerator<T>
{
  T Current { get; }
  bool MoveNext();
  // void Reset(); Reset现在一般不使用
}
// MoveNext方法返回bool值，再调用Current属性
public interface IEnumerator<T>{
  // MoveNext 方法返回 T 的实例
  // 结束的话则什么也不返回(相当于void)
  // 另外加上有可能抛出的异常，那么算起来有3种类型的返回形式
  T|void|Exception GetNext(void);
}
// 参数和返回值进行掉换
// Pull 类型取下一个值用 Get 作为谓词
// 因此 Push 类型则使用 Got
public interface IEnumeratorDual<T>
{
  void GotNext(T|void|Exception);
}
// 进而按照 Pull 的3种类型的返回形式分开定义
public interface IEnumeratorDual<T>
{
  void GotNext(T);
  void GotVoid(void); // void也就是完成
  void GotException(Exception);
}
// 最后推演出的 IObserver<T> 接口
```

```
30. public interface IObserver<T>
31. {
32.   void OnNext(T value);
33.   void OnCompleted();
34.   void OnError(Exception error);
35. }
```

这里虽然没有更详细的介绍，在IEnumerable<T>接口上的操作，在IObservable<T>也是能够实现的。综上所述，IEnumerable<T>和IObservable<T>完完全全可以相互转换。实际上IEnumerable<T>接口可以通过ToObservable扩展方法、IObservable<T>接口通过ToEnumerable扩展方法相互转换，而这些扩展方法在 Rx 里已经定义了。另外，通过下图，您也可以清楚的看出：IEnumerable<T>是Pull型，IObservable<T>是Push型的

Push型的场景，程序在最初需要 IObserver<T> (观察者) 需要对 IObservable<T>(被观察者)进行注册(Subscribe)后，进行消息的传递。而Push型的场景，程序是需要取得IEnumerable<T> 对象，然后主动的取值。

事件的使用例

事件有多种多样，比如GUI的鼠标事件，WindowsPhone的手势、传感器，实现INotifyPropertyChanged 接口的对象自身的数据变化通知，以及数据绑定等。

使用 Rx 可以将事件进行组合。比如：鼠标的按下/移动/放开的事件进行组合，生成一个drag/drop事件，代码如下：

```
1. using System.Reactive.Linq;
2. // Winform的drag(等于按下左键鼠标移动直到放开)
3. var drag = from down in this.MouseDownAsObservable()
4.            from move in
   this.MouseMoveAsObservable().TakeUntil(this.MouseUpAsObservable())
5.            select new { move.X, move.Y };
6. // 利用扩展方法将Winform原有的事件变换为 IObservable<T> 对象
7. public static class FormExtensions
8. {
9.   public static IObservable<MouseEventArgs> MouseMoveAsObservable(this Form form)
10.  {
11.    return Observable.FromEventPattern<MouseEventArgs>(form, "MouseMove").Select(e
   => e.EventArgs);
12.  }
13.  public static IObservable<MouseEventArgs> MouseDownAsObservable(this Form form)
14.  {
15.    return Observable.FromEventPattern<MouseEventArgs>(form, "MouseDown").Select(e
   => e.EventArgs);
16.  }
17.  public static IObservable<MouseEventArgs> MouseUpAsObservable(this Form form)
18.  {
19.    return Observable.FromEventPattern<MouseEventArgs>(form, "MouseUp").Select(e =>
   e.EventArgs);
20.  }
21. }
```

除此之外，结合时间进行过滤也可以利用 Rx 实现。如下例：

```
1. // 取得TextBox控件的TextChanged事件
2. Observable.FromEventPattern<EventArgs>(textBox, "TextChanged")
3.   .Select(_ => textBox.Text)
4.   .Throttle(TimeSpan.FromSeconds(1)); // 等待1秒钟如果没有别的变化则使用最近的变化值
```

如果你想实现根据输入的值进行实时通信的场景，比如增量式搜索等，试图发送所有的值则可能造成不小的浪费，在这种情况下设定一定的阀值以过滤一些不必要的输入。比如上面的例子中，输入的值不断变化，在1秒内无论发生多少次修改总是最后一次的值才会Push出去(之前的值会被过滤掉)。因此利用这样处理，可以控制不必要的网络通信。
这样的针对时间上的操作，在以往的编程里总是非常绕而且麻烦的，而使用 Rx 来实现则非常简单，基本上一个方法就能搞定了。

异步操作的使用例

Rx 在异步操作方面也非常强大。比如考虑一下(WPF里)WebRequest类(System.Net命名空间下)的异步处理的场景。Begin-End的异步编程模型使用Lambda表达式基本上需要进行嵌套，另外，在处理多个请求或者复杂的处理时，需要注意异常处理的范围，你需要在每一个嵌套的Lambda表达式里使用try-catch语句。另外UI操作的BeginInvoke也显得很麻烦。

```
using System.Net;
using System.IO;
var req = WebRequest.Create("http://hoge/");
req.BeginGetResponse(ar =>
{
  try
  {
    var res = req.EndGetResponse(ar);
    var url = new StreamReader(res.GetResponseStream()).ReadToEnd();
    var req2 = WebRequest.Create(url); // 在前面请求结果的基础上，再次发起请求
    req2.BeginGetResponse(ar2 =>
    {
      try
      {
        var res2 = req2.EndGetResponse(ar2);
        var str = new StreamReader(res2.GetResponseStream()).ReadToEnd();
        Dispatcher.BeginInvoke(new Action(() => MessageBox.Show(str)));
      }
      catch (WebException e) { Dispatcher.BeginInvoke(new Action(() =>
MessageBox.Show(e.ToString()))); }
    }, null);
  }
  catch (WebException e)
  {
    Dispatcher.BeginInvoke(new Action(() => MessageBox.Show(e.ToString())));
  }
}, null);
```

使用 Rx，代码如下：

```
WebRequest.Create("http://hoge/")
  .DownloadStringAsync()
  .SelectMany(url => WebRequest.Create(url).DownloadStringAsync())
  .ObserveOnDispatcher() // 在UI线程里响应
  .Subscribe(
    str => MessageBox.Show(str), // 请求成功时的处理
    e => MessageBox.Show(e.ToString())); // 请求失败时的处理
public static class WebRequestExtensions
{
  // 利用 Rx 可以通过扩展方法来分离处理的主体，代码更加简洁
  public static IObservable<string> DownloadStringAsync(this WebRequest request)
  {
    return Observable.FromAsyncPattern<WebResponse>(request.BeginGetResponse,
request.EndGetResponse)()
      .Select(res =>
      {
        using (var stream = res.GetResponseStream())
        using (var sr = new StreamReader(stream))
        {
          return sr.ReadToEnd();
        }
      });
  }
}
```

使用 Rx 可以将 Lambda 表达式的嵌套改写为 Method Chain，另外异常处理和UI刷新都可以集中写在一起，从而大幅度减少代码量。对于扩展方法里分离出去的处理主体，或许会感觉到有些不爽，但正因为很容易的将这些代码作为组件分离出去而成为 Rx 的主要优势之一。

谈到这里，可能会有人提起 async/await (.NET 4.5里引入的新异步编程模型)，那么 Rx 还有什么优势么？首先，Rx 在 .NET 4.0就可以使用的(当然 async/await 在.NET CTP里也发布了)，而且在对于 async/await 的使用上 .NET 5.0 里将会新增 Task<T> 的转换方法。最后，Rx 和 async/await 在关注领域上各有偏重，因此 Rx 的自身价值也不会因为 async/await 的出现而削弱。