Reactive Extensions入门(6)：使用Rx进行单元测试

yycoding.xyz/post/2012/7/10/introducting-linq-and-reactiveextensions-unittest

Rx本身就是一个功能强大的测试框架。本文将介绍如何使用Rx模拟异步方法调用返回预定的值来辅助针对一步方法调用的单元测试。Rx可以模拟经过一段预定时间返回结果的异步方法，从而简化了异步处理方法的测试。Rx也可以模拟异步方法返回错误的场景，使得代码测试可以覆盖多有的用例。本文将介绍Rx中的TestSheduler方法，他可以模拟耗时的异步操作，但是能在测试的时候能够立即执行。

1. 模拟异步方法

开发者可能会对测试与时间有关的异步方法会感到头疼。与测试同步方法不同，异步方法不仅需要返回需要的值，而且还要能够模拟出方法耗时以及延迟的效果。

有很多工具能够用来测试同步方法，但是测试异步方法则需要一定的技巧。我们可以使用Thread.Sleep()来模拟需要长时间执行的方法，但是该方法能用来测试的时间用例非常粗糙，在有些对执行时间要求比较高的场景，Thread.Sleep方法不能很好的满足我们的要求。

Rx的一个最大特色在于它能够很好的控制事件的执行时间。我们可以使用Rx中的Timeout，Delay，Windows和Buffer等操作符来灵活的控制事件的产生时间。

例如，如果我们需要调用一个名为FetchWebPage()的方法，签名如下：

```
public IObservable<string> FetchWebpage(string url);
```

那么我们可以实现一个如下的模拟方法：

```
public static IObservable<string> FetchWebpageStub(string url)
{
    // Check the URL and error out if the Url is invalid
    Uri testUrl;
    try
    {
        testUrl = new Uri(url);
    }
    catch (UriFormatException ex)
    {
        return Observable.Throw<string>(ex);
    }
    return Observable.Return(String.Format(@"<html><body><p>'{0}' not found</p></body>
</html>", testUrl));
}
```

我们也可以在传统的同步方法中使用这些方法，就像测试同步方法一样，只需要将所有的异步方法使用Rx来模拟即可。

2. 使用.First()方法来模拟异步方法

对于单元测试来说，其唯一目的就是测试从方法返回的实际值是否正确，调用异步方法的最简单方法就是在方法后面加上.First()，等待方法返回结果。下面这个例子就是用来测试上面的方法。我们可以对下面这个测试用例进行简单更改就可以达到我们想要的结果。

```
[TestMethod]
void FetchWebpageSuccessCase()
{
    var result = FetchWebpageStub("http://www.google.com").First();
    Assert.IsFalse(String.IsNullOrEmpty(result));
    Assert.IsTrue(result.ToLower().Contains("html"));
}
```

3. 模拟耗时

虽然上面的例子能够正确的返回结果，但是他仍没能有效的模拟处理的时间消耗。下面我们建立一个模拟方法来使这个方法需要一定时间才能返回我们想要的结果。要注意的是，任何一个不需要立即返回值的方法，都应该有一个Ischeduler类型的参数。后面我们将会看到这一点的重要性。

```
IObservable<string> FetchWebpageOnDialup(string url, IScheduler scheduler = null)
{
    // 模拟一个比较慢的操作，5秒后返回结果
    return FetchWebpageStub(url).Delay(TimeSpan.FromSeconds(5.0), scheduler);
}
```

或者在上面方法的基础上模拟一个抛出异常的方法：

```
IObservable<string> FetchWebpageOnAirportWifi(string url, IScheduler scheduler = null)
{
    // 模拟抛出异常，然后等待
    return FetchWebpageOnDialup(url, scheduler)
    .SelectMany(_ => Observable.Throw<string>(new WebException()));
}
```

4. 使用Schedulers对象

在Rx中，任何不直接通过计算后立即返回值的方法都是通过实现Ischeduler接口来实现的，换句话说，Rx从不直接创建一个Task<T>或者直接新建一个Thread对象。他委托Scheduler来执行实际的工作。

例如，我们可以创建一个Scheduler对象，在某一段时间内不执行任何操作，只是记录一下Scheduler运行的时间，然后在某一时间点之后开始执行一些操作。如果我们知道了事件执行的时间顺序，我们可以实现如“让程序运行10分钟，然后暂停”，Scheduler对象会运行10分钟，然后停止。既然我们知道时间线上的事件的执行顺序，我们不必关心某一个事件实际发生的时间，我们只需关心事件执行的顺序是否正确就可以了。这样我们就可以让测试方法尽快的按顺序执行这些事件，仍然可以得到正确的结果，因为我们的事件执行顺

利和真实环境中事件的执行顺序是一致的。当我们使用虚拟的Scheduler来做测试时，结果是确定性的，即每次运行相同的测试方法始终会得到相同的结果，这点和Thread.Sleep()不同。

如前所述，使用Scheduler方法的关键在于保证每一个接受Ischeduler的方法都提供了自己实现的Scheduler对象。在一些比较复杂的程序中，这一点可能不是特别明显。有时候，我们可以创建一个全局的静态的Scheduler对象来在单元测试中使用。下面的代码展示了我们使用TestScheduler来测试FetchWebpageOnDialup方法。TestScheduler对象继承与VirtualTimeScheduler。

```
[TestMethod]
void MakeSureFetchWebpageTakesTime()
{
    var sched = new TestScheduler();
    string result = null;
    // 不要使用.First()方法，否则起不到演示效果了
    var fixture = FetchWebpageOnDialup("http://www.yahoo.com", sched);
    fixture.Subscribe(x => result = x);
    // 向前推3秒，由于FetchWebpageOnDialup要等5秒才返回结果，所以现在的结果仍然是空
    sched.AdvanceTo(TimeSpan.FromSeconds(3.0).Ticks);
    Assert.AreEqual(null, result);
    // 向前推6秒，现在应该有结果了
    sched.AdvanceTo(TimeSpan.FromSeconds(6.0).Ticks);
    Assert.IsFalse(String.IsNullOrEmpty(result));
    Assert.IsTrue(result.ToLower().Contains("html"));
}
```

Scheduler对象可以使得我们构造自己的时间线，有了这个时间线，我们就可以用来精确的测试一些耗时的异步方法了。

5. 测试ReactiveUI应用程序

在上一篇文章中，我们介绍了与Rx结合比较紧密的MVVM框架ReactiveUI。和其他MVVM框架一样，ReactiveUI框架引入的ViewModel这一层使得单元测试变得更加容易。在上篇文章中我们展示了使用MVVM框架开发了一个搜索图片的应用程序。现在我们利用前面讲过的知识来为这个MVVM应用程序编写单元测试代码。

测试应用程序的UI交互逻辑是Rx测试框架的一大特色。传统的UI交互代码都是在后台执行的，要对UI界面的交互进行测试比较困难，一般的做法是，使用一些专门的自动化测试软件模拟屏幕鼠标或者键盘操作。现在使用ReactiveUI这个MVVM框架，我们就可以使用Rx来进行这些诸如模拟用户点击按钮，输入数据，然后点击查询这些交互了。使用Rx模拟用户的这些交互，然后测试程序的反应就可以测试应用程序的UI逻辑是否正确。

我们要测试的应用程序交互界面很简单，就是用户在界面上输入一些关键字，然后程序通过WebService进行查询最后返回查询结果。当然，我们测试的应用程序有几点:

只有当用户输入的内容产生变化，才会进行查询；当应用程序正在查询时spinner对象要显示出来；在同一时间不能执行多个查询。一旦深究起界面交互逻辑，你会发现要编写一个测试UI交互是否正确的单元测试还是比较难得，但至少RxUI及Rx能够简化不少工作。

由于我们的View是和ViewModel进行绑定的。所以我们的测试只需要对ViewModel即AppViewModel.cs 这个类进行测试就可以实现UI交互逻辑的测试了。

5.1 测试用户停止输入查询条件一段时间后才发出查询请求

我们的图片查询程序，并不会在用户正在输入查询条件的时候就马上触发查询，而是在用户暂停输入一小段时间后触发查询请求。我们将这个测试用例命名为SearchesShouldntRunOnEveryKeystroke，要测试这个用例，我们需要创建一个Scheduler对象，然后添加一系列的时间线以及对应的数据交互。

```
(new TestScheduler()).With(sched =>
{
    // 模拟用户在输入框中的输入
    var keyboardInput = sched.CreateColdObservable(
        sched.OnNextAt(10, "R"),
        sched.OnNextAt(20, "Ro"),
        sched.OnNextAt(30, "Robo"),
        sched.OnNextAt(40, "Robot"),
        sched.OnNextAt(2000, "Hat"));
```

在上面的代码中，我们模拟用户在输入框中输入的字符以及时间。在ReactiveUI中，框架使用两种类型的Scheduler对象，一种是Deferred Scheduler，比如UI线程，一种是Task Pool Sheduler比如后台执行的线程。在这里我们创建了一个TestScheduler，然后调用With方法。With中有一个匿名方法，方法执行完了之后TestScheduler也会释放，这样就可以保证本测试不会影响到其他测试。

接下来，我们要保证查询命令在我们发出请求时能够执行：

```
// 保证命令总是能够执行，模拟实际搜索的代码
var fixture = new AppViewModel(new ReactiveAsyncCommand(null, 1000),
Observable.Never<List<FlickrPhoto>>());
//包装键盘输入事件
keyboardInput.Subscribe(x => fixture.SearchTerm = x);

// 记录查询时间执行的次数
int numTimesCommandInvoked = 0;
fixture.ExecuteSearch.Subscribe(x => numTimesCommandInvoked++);
```

我们根据时间线上的时间来测试交互是否正确。我们调用Scheduler对象的RunToMilliseconds方法，来模拟执行到的时间点。

```
sched.RunToMilliseconds(25);
Assert.AreEqual(0, numTimesCommandInvoked);

sched.RunToMilliseconds(40);
Assert.AreEqual(0, numTimesCommandInvoked);

sched.RunToMilliseconds(1800);
Assert.AreEqual(1, numTimesCommandInvoked);

sched.RunToMilliseconds(2010);
Assert.AreEqual(1, numTimesCommandInvoked);

sched.RunToMilliseconds(5000);
Assert.AreEqual(2, numTimesCommandInvoked);
```

5.2 测试Spinner控件在程序执行时显示

在我们的应用程序中，当程序正在查询时，输入框右侧会显示一个Spinner控件，表示程序当前正在执行一次图片查询，查询结果返回时Spinner控件隐藏。

```
[TestMethod]
public void SpinnerShouldSpinWhileAppIsSearching()
{
    (new TestScheduler()).With(sched =>
    {
        // 我们需要创建一个模拟的Observable对象来模拟通过Webservice返回的数据
        // 在这里我们模拟Webservice在5秒钟后才返回数据
        var searchObservable = Observable.Return(createSampleResults())
            .Delay(TimeSpan.FromMilliseconds(5000), RxApp.TaskpoolScheduler);

        var command = new ReactiveAsyncCommand();
        command.RegisterAsyncObservable(x => searchObservable);

        var fixture = new AppViewModel(command, searchObservable);

        // 程序最开始的时候Spinner控件是隐藏的
        Assert.AreNotEqual(Visibility.Visible, fixture.SpinnerVisibility);

        // 执行查询命令
        fixture.ExecuteSearch.Execute("Robot");

        // 正在查询时Spinner控件应该隐藏
        sched.RunToMilliseconds(100);
        Assert.AreEqual(Visibility.Visible, fixture.SpinnerVisibility);

        // 6秒过后，程序应该查询完成，结果返回了，Spinner空间应该隐藏
        sched.RunToMilliseconds(6 * 1000);
        Assert.AreNotEqual(Visibility.Visible, fixture.SpinnerVisibility);
    });
}

List<FlickrPhoto> createSampleResults()
{
    return new List<FlickrPhoto>() {
        new FlickrPhoto() {
            Description = "A sample image description",
            Title = "Sample Image",
            Url = "http://www.example.com/image.gif",
        },
    };
}
```

上面的代码应该比较好理解。可以看出Scheduler能够灵活的控制当前程序执行到的时间点，只需要调用RunToMilliseconds 即可，如sched.RunToMilliseconds(6 * 1000)，该方法中我们并不需要等待着这6秒钟过去，我们可以直接跳到6秒这个时间点，然后测试Spinner控件是否显示。

5.3 测试连续搜索时只有搜索内容发生变化时才执行搜索

我们的图片查询程序中，只有当查询的关键字发生改变时才执行查询，这样可以避免重复请求。要测试这个场景我们也需要向前面那样模拟一个查询返回结果Observable表示查询返回结果。

```
var searchObservable = Observable.Return(createSampleResults())
    .Delay(TimeSpan.FromMilliseconds(5 * 1000), RxApp.TaskpoolScheduler);
```

然后我们模拟一个查询命令，传入到ViewModel对象，返回一个Observable。

```
// 创建一个查询命令，传入到ViewModel对象中，然后返回Observable
var command = new ReactiveAsyncCommand();
command.RegisterAsyncObservable(x => searchObservable);

var fixture = new AppViewModel(command, searchObservable);
Assert.IsTrue(fixture.SearchResults.Count == 0);

fixture.SearchTerm = "Foo";
```

现在我们来执行一些测试，在两秒时，没有结果返回，但是10秒后应该有结果返回：

```
// 2秒时间点上，还没有结果
sched.RunToMilliseconds(2 * 1000);
Assert.IsTrue(fixture.SearchResults.Count == 0);

// 10秒后，应该有结果返回
var sampleData = createSampleResults();
sched.RunToMilliseconds(10 * 1000);
Assert.AreEqual(sampleData.Count, fixture.SearchResults.Count);
```

最后我们要测试，返回的结果和实际的值是否一致:

```
// 保证这两个查询序列结果是一样的
foreach (var item in sampleData.Zip(fixture.SearchResults, (expected, actual) => new {
expected, actual }))
{
    Assert.AreEqual(item.expected.Title, item.actual.Title);
    Assert.AreEqual(item.expected.Description, item.actual.Description);
    Assert.AreEqual(item.expected.Url, item.actual.Url);
}
```

6. 结语

单元测试在软件开发中的作用日益重要，尤其是在测试驱动开发这样的软件开发模式中。Reactive Extension简化了针对异步方法进行单元测试的复杂性。使用Rx，对异步方法调用进行模拟变得非常直观，并且测试结果确定。

本文介绍了使用Rx的测试框架来模拟各种异步方法，并对前面一篇文章中我们使用ReactiveUI这个MVVM框架编写的WPF应用程序进行了单元测试。当然这些只是入门性的介绍，如果这些能让您感受到无论是使用Rx编写异步方法，或者是编写基于MVVM的应用程序，或者是编写针对异步方法进行单元测试所带来的简洁，那本文的目的就达到了。