TURING 图灵程序设计丛书

C#并发编程经典实例

Concurrency in C# Cookbook

[美] Stephen Cleary 著
相银初 译

人民邮电出版社
北　京

图书在版编目（CIP）数据

C#并发编程经典实例 /（美）克利里（Cleary,S.）著 ; 相银初译. -- 北京 : 人民邮电出版社, 2015.1
（图灵程序设计丛书）
ISBN 978-7-115-37427-1

Ⅰ. ①C… Ⅱ. ①克… ②相… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第260737号

内 容 提 要

本书全面讲解 C# 并发编程技术，侧重于.NET 平台上较新、较实用的方法。全书分为几大部分：首先介绍几种并发编程技术，包括异步编程、并行编程、TPL 数据流、响应式编程；然后阐述一些重要的知识点，包括测试技巧、互操作、取消并发、函数式编程与 OOP、同步、调度；最后介绍了几个实用技巧。全书共包含 70 多个有配套源码的实用方法，可用于服务器程序、桌面程序和移动应用的开发。

本书适合具有 .NET 基础，希望学习最新并发编程技术的开发人员阅读。

◆ 著 [美] Stephen Cleary
译 相银初
责任编辑 李松峰
执行编辑 李 静 曹静雯
责任印制 杨林杰

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：11.75
字数：237千字 2015年 1 月第 1 版
印数：1－3 000册 2015年 1 月北京第 1次印刷
著作权合同登记号 图字：01-2014-6523号

定价：49.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

译者序

关于并发编程的几个误解

关于并发编程，很多人都有一些误解。

误解一：并发就是多线程

实际上多线程只是并发编程的一种形式，在 C# 中还有很多更实用、更方便的并发编程技术，包括异步编程、并行编程、TPL 数据流、响应式编程等。

误解二：只有大型服务器程序才需要考虑并发

服务器端的大型程序要响应大量客户端的数据请求，当然要充分考虑并发。但是桌面程序和手机、平板等移动端应用同样需要考虑并发编程，因为它们是直接面向最终用户的，而现在用户对使用体验的要求越来越高。程序必须能随时响应用户的操作，尤其是在后台处理时（读写数据、与服务器通信等），这正是并发编程的目的之一。

误解三：并发编程很复杂，必须掌握很多底层技术

C# 和 .NET 提供了很多程序库，并发编程已经变得简单多了。尤其是 .NET 4.5 推出了全新的 `async` 和 `await` 关键字，使并发编程的代码减少到了最低限度。并行处理和异步开发已经不再是高手们的专利，只要使用本书中的方法，每个开发人员都能写出交互性良好、高效、可靠的并发程序。

本书的特色

本书全面讲解 C# 并发编程技术，侧重于 .NET 平台上较新、较实用的方法。全书分为几大

部分：首先介绍几种并发编程技术，包括异步编程、并行编程、TPL 数据流、响应式编程等；然后是一些重要的知识点，包括测试技巧、互操作、取消并发、函数式编程与 OOP、同步、调度等；最后介绍了几个实用技巧。书中包含 70 多个配有源码的实用方法，可用于服务器程序、桌面程序和移动端应用的开发。

本书填补了一个市场空白：它是一本用最新方法进行并发编程的入门指引和参考书。

本书作者 Stephen Cleary 是美国著名的软件开发者和技术书作家、C# MVP，在 C#/C++/JavaScript 等方面均有丰富的经验。我非常有幸能翻译他的著作。

翻译中的一点感受

过去的十多年我一直在从事软件开发和设计工作。相信国内很多开发人员都和我一样，心中存在着一个疑惑：我国的软件人员很多（绝对数量不会比美国少），但为什么软件技术总体上落后欧美国家那么多？确定翻译《C# 并发编程经典实例》这本书后，我一边仔细阅读原书，一边遵循作者的思路，逐渐发现作者思考问题的一个理念。这就是按软件的不同层次进行明确分工，我只负责我所实现的这个层次，底层技术是为上层服务的，我只负责选择和调用，不管内部的实现过程；同样，我负责的层次为更高一层的软件提供服务，供上层调用，也不需要上层关心我的内部实现。

由此想到，这正好反映出国内开发人员中的一个通病，即分工不够细、技术关注不够精。很多公司和团队在开发时都喜欢大包大揽，从底层到应用层全部自己实现；很多开发人员也热衷于“大而全”地学习技术，试图掌握软件开发中的各种技术，而不是精通某一方面。甚至流行这样一种观点，实现底层软件、写驱动的才是高级开发人员，做上层应用的人仅仅是“码农”。本书作者明确地反对了这种看法，书中强调如何利用好现成的库，而不是全部采用底层技术自己实现。利用现成的库开发出高质量的软件，对技术能力的考验并不低于开发底层库。

感谢

在本书的翻译过程中，得到了图灵公司李松峰老师的支持和帮助，在此表示感谢。由于本人水平有限，书中难免有疏忽和错误，恳请读者朋友们批评指正。

2014 年 10 月于深圳

第5章

Rx基础

LINQ是对序列数据进行查询的一系列语言功能。内置的LINQ to Objects（基于IEnumerable<T>）和LINQ to Entities（基于IQueryableT>）是两个最常用的LINQ提供者。另外还有很多提供者，并且大多数都采用相同的基本架构。查询是延后执行（lazily evaluated）的，只有在需要时才会从序列中获取数据。从概念上讲，这是一种拉取模式。在查询过程中数据项是被逐个拉取出来的。

Reactive Extensions（Rx）把事件看作是依次到达的数据序列。因此，将Rx认作是LINQ to events（基于IObservable<T>）也是可以的，它与其他LINQ提供者的主要区别在于，Rx采用“推送”模式。就是说，Rx的查询规定了在事件到达时程序该如何响应。Rx在LINQ的基础上构建，增加了一些功能强大的操作符，作为扩展方法。

本章介绍一些更常用的Rx操作。需要注意的是，所有的LINQ操作都可以在Rx中使用。从概念上看，过滤（Where）、投影（Select）等简单操作，和其他LINQ提供者的操作是一样的。本章不介绍那些常见的LINQ操作，而将重点放在Rx在LINQ基础上增加的新功能，尤其是与时间有关的功能。

要使用Rx，需要在应用中安装一个NuGet包Rx-Main。支持Reactive Extensions的平台非常丰富（参见表5-1）。

表5-1：支持Reactive Extensions的平台

平　台	Rx支持情况
.NET 4.5	✓
.NET 4.0	✓

（续）

平　台	Rx支持情况
Mono iOS/Droid	✓
Windows Store	✓
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	✓
Windows Phone SL 7.1	✓
Silverlight 5	✓

5.1　转换.NET事件

问题

把一个事件作为 Rx 输入流，每次事件发生时通过 OnNext 生成数据。

解决方案

Observable 类定义了一些事件转换器。大部分 .NET 框架事件与 FromEventPattern 兼容，对于不遵循通用模式的事件，需要改用 FromEvent。

FromEventPattern 最适合使用委托类型为 EventHandler<T> 的事件。很多较新框架类的事件都采用了这种委托类型。例如，Progress<T> 类定义了事件 ProgressChanged，这个事件的委托类型就是 EventHandler<T>，因此，它就很容易被封装到 FromEventPattern：

```
var progress = new Progress<int>();
var progressReports = Observable.FromEventPattern<int>(
    handler => progress.ProgressChanged += handler,
    handler => progress.ProgressChanged -= handler);
progressReports.Subscribe(data => Trace.WriteLine("OnNext:" + data.EventArgs));
```

请注意，data.EventArgs 是强类型的 int。FromEventPattern 的类型参数（上例中为 int）与 EventHandler<T> 的 T 相同。Rx 用 FromEventPattern 中的两个 Lambda 参数来实现订阅和退订事件。

较新的 UI 框架采用 EventHandler<T>，可以很方便地应用在 FromEventPattern 中。但是有些较旧的类常为每个事件定义不同的委托类型。这些事件也能在 FromEventPattern 中使用，但需要做一些额外的工作。例如，System.Timers.Timer 类有一个事件 Elapsed，它的类型是 ElapsedEventHandler。对此旧类事件，可以用下面的方法封装进 FromEventPattern：

```
var timer = new System.Timers.Timer(interval: 1000) { Enabled = true };
var ticks = Observable.FromEventPattern<ElapsedEventHandler, ElapsedEventArgs>(
    handler => (s, a) => handler(s, a),
    handler => timer.Elapsed += handler,
```

```
        handler => timer.Elapsed -= handler);
    ticks.Subscribe(data => Trace.WriteLine("OnNext: " + data.EventArgs.SignalTime));
```

注意，data.EventArgs 仍然是强类型的。现在 FromEventPattern 的类型参数是对应的事件处理程序和 EventArgs 的派生类。FromEventPattern 的第一个 Lambda 参数是一个转换器，它将 EventHandler<ElapsedEventArgs> 转换成 ElapsedEventHandler。除了传递事件，这个转换器不应该做其他处理。

上面代码的语法明显有些别扭。另一个方法是使用反射机制：

```
var timer = new System.Timers.Timer(interval: 1000) { Enabled = true };
var ticks = Observable.FromEventPattern(timer, "Elapsed");
ticks.Subscribe(data => Trace.WriteLine("OnNext: "
    + ((ElapsedEventArgs)data.EventArgs).SignalTime));
```

采用这种方法后，调用 FromEventPattern 就简单多了。但是这种方法也有缺点：出现了一个怪异的字符串（"Elapsed"），并且消息的使用者不是强类型了。就是说，这时 data.EventArgs 是 object 类型，需要人为地转换成 ElapsedEventArgs。

讨论

事件是 Rx 流数据的主要来源。本节介绍如何封装遵循标准模式的事件（标准事件模式：第一个参数是事件发送者，第二个参数是事件的类型参数）。对于不标准的事件类型，可以用重载 Observable.FromEvent 的办法，把事件封装进 Observable 对象。

把事件封装进 Observable 对象后，每次引发该事件都会调用 OnNext。在处理 AsyncCompletedEventArgs 时会发生令人奇怪的现象，所有的异常信息都是通过数据形式传递的（OnNext），而不是通过错误传递（OnError）。看一个封装 WebClient.DownloadStringCompleted 的例子：

```
var client = new WebClient();
var downloadedStrings = Observable.FromEventPattern(client,
    "DownloadStringCompleted");
downloadedStrings.Subscribe(
    data =>
    {
        var eventArgs = (DownloadStringCompletedEventArgs)data.EventArgs;
        if (eventArgs.Error != null)
            Trace.WriteLine("OnNext: (Error) " + eventArgs.Error);
        else
            Trace.WriteLine("OnNext: " + eventArgs.Result);
    },
    ex => Trace.WriteLine("OnError: " + ex.ToString()),
    () => Trace.WriteLine("OnCompleted"));
client.DownloadStringAsync(new Uri("http://invalid.example.com/"));
```

WebClient.DownloadStringAsync 出错并结束时，引发带有异常 AsyncCompletedEventArgs.Error

的事件。可惜 Rx 会把这作为一个数据事件，因此这个程序的结果是显示“OnNext:(Error)”，而不是“OnError:”。

有些事件的订阅和退订必须在特定的上下文中进行。例如，很多 UI 控件的事件必须在 UI 线程中订阅。Rx 提供了一个操作符 SubscribeOn，可以控制订阅和退订的上下文。大多数情况下没必要使用这个操作符，因为基于 UI 的事件订阅通常就是在 UI 线程中进行的。

参阅

5.2 节介绍如何修改引发事件的上下文。

5.4 节介绍如何对事件限流，以免订阅者因事件太多而崩溃。

5.2 发通知给上下文

问题

Rx 尽量做到了线程不可知（thread agnostic）。因此它会在任意一个活动线程中发出通知（例如 OnNext）。

但是我们通常希望通知只发给特定的上下文。例如 UI 元素只能被它所属的 UI 线程控制，因此，如果要根据 Rx 的通知来修改 UI，就应该把通知“转移”到 UI 线程。

解决方案

Rx 提供了 ObserveOn 操作符，用来把通知转移到其他线程调度器。

看下面的例子，使用 Interval，每秒钟产生一个 OnNext 通知：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Trace.WriteLine("UI thread is " + Environment.CurrentManagedThreadId);
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Subscribe(x => Trace.WriteLine("Interval " + x + " on thread " +
            Environment.CurrentManagedThreadId));
}
```

用我的电脑测试，显示结果为：

```
UI thread is 9
Interval 0 on thread 10
Interval 1 on thread 10
Interval 2 on thread 11
Interval 3 on thread 11
Interval 4 on thread 10
```

```
Interval 5 on thread 11
Interval 6 on thread 11
```

因为 Interval 基于一个定时器（没有指定的线程），通知会在线程池线程中引发，而不是在 UI 线程中。要更新 UI 元素，可以通过 ObserveOn 输送通知，并传递一个代表 UI 线程的同步上下文：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var uiContext = SynchronizationContext.Current;
    Trace.WriteLine("UI thread is " + Environment.CurrentManagedThreadId);
    Observable.Interval(TimeSpan.FromSeconds(1))
        .ObserveOn(uiContext)
        .Subscribe(x => Trace.WriteLine("Interval " + x + " on thread " +
            Environment.CurrentManagedThreadId));
}
```

ObserveOn 的另一个常用功能是可以在必要时离开 UI 线程。假设有这样的情况：鼠标一移动，就意味着需要进行一些 CPU 密集型的计算。默认情况下，所有的鼠标移动事件都发生在 UI 线程，因此可以使用 ObserveOn 把通知移动到一个线程池线程，在那里进行计算，然后再把表示结果的通知返回给 UI 线程：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var uiContext = SynchronizationContext.Current;
    Trace.WriteLine("UI thread is " + Environment.CurrentManagedThreadId);
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(evt => evt.EventArgs.GetPosition(this))
        .ObserveOn(Scheduler.Default)
        .Select(position =>
        {
            // 复杂的计算过程。
            Thread.Sleep(100);
            var result = position.X + position.Y;
            Trace.WriteLine("Calculated result " + result + " on thread " +
                Environment.CurrentManagedThreadId);
            return result;
        })
        .ObserveOn(uiContext)
        .Subscribe(x => Trace.WriteLine("Result " + x + " on thread " +
            Environment.CurrentManagedThreadId));
}
```

运行这段代码的话，就会发现计算过程是在线程池线程中进行的，计算结果在 UI 线程中显示。另外，还会发现计算和结果会滞后于输入，形成等待的队列，这种现象出现的原因在于，比起 100 秒 1 次的计算，鼠标移动的更新频率更高。Rx 中有几种技术可以处理这种情况，其中一个常用方法是对输入流速进行限制，具体会在 5.4 节介绍。

讨论

实际上，ObserveOn 是把通知转移到一个 Rx 调度器上了。本节介绍了默认调度器（即线程池）和一种创建 UI 调度器的方法。ObserveOn 最常用的功能是移到或移出 UI 线程，但调度器也能用于别的场合。6.6 节介绍高级测试时，将再次关注调度器。

参阅

5.1 节介绍如何利用事件创建序列。

5.4 节介绍如何限制事件流的流速。

6.6 节介绍测试 Rx 代码的特殊流程。

5.3 用窗口和缓冲对事件分组

问题

有一系列事件，需要在它们到达时进行分组。举个例子，需要对一些成对的输入作出响应。第二个例子，需要在 2 秒钟的窗口期内，对所有输入进行响应。

解决方案

Rx 提供了两个对到达的序列进行分组的操作：Buffer 和 Window。Buffer 会留住到达的事件，直到收完一组事件，然后会把这一组事件以一个集合的形式一次性地转送过去。Window 会在逻辑上对到达的事件进行分组，但会在每个事件到达时立即传递过去。Buffer 的返回类型是 IObservable<IList<T>>（由若干个集合组成的事件流）；Window 的返回类型是 IObservable<IObservable<T>>（由若干个事件流组成的事件流）。

下面的例子使用 Interval，每秒创建 1 个 OnNext 通知，然后每 2 个通知做一次缓冲：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Buffer(2)
        .Subscribe(x => Trace.WriteLine(
            DateTime.Now.Second + ": Got " + x[0] + " and " + x[1]));
}
```

用我的电脑测试，每 2 秒产生 2 个输出：

```
13: Got 0 and 1
15: Got 2 and 3
17: Got 4 and 5
```

```
19: Got 6 and 7
21: Got 8 and 9
```

下面的例子有些类似，使用 Window 创建一些事件组，每组包含 2 个事件：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Window(2)
        .Subscribe(group =>
        {
            Trace.WriteLine(DateTime.Now.Second + ": Starting new group");
            group.Subscribe(
                x => Trace.WriteLine(DateTime.Now.Second + ": Saw " + x),
                () => Trace.WriteLine(DateTime.Now.Second + ": Ending group"));
        });
}
```

用我的电脑测试，输出的结果就是这样：

```
17: Starting new group
18: Saw 0
19: Saw 1
19: Ending group
19: Starting new group
20: Saw 2
21: Saw 3
21: Ending group
21: Starting new group
22: Saw 4
23: Saw 5
23: Ending group
23: Starting new group
```

这几个例子说明了 Buffer 和 Window 的区别。Buffer 等待组内的所有事件，然后把所有事件作为一个集合发布。Window 用同样的方法进行分组，但它是在每个事件到达时就发布。

Buffer 和 Window 都可以使用时间段作为参数。在下面的例子中，所有的鼠标移动事件被收集进窗口，每秒一个窗口：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Buffer(TimeSpan.FromSeconds(1))
        .Subscribe(x => Trace.WriteLine(
            DateTime.Now.Second + ": Saw " + x.Count + " items."));
}
```

输出的结果依赖于怎么移动鼠标，类似于这样：

```
49: Saw 93 items.
50: Saw 98 items.
51: Saw 39 items.
52: Saw 0 items.
53: Saw 4 items.
54: Saw 0 items.
55: Saw 58 items.
```

讨论

Buffer 和 Window 可用来抑制输入信息，并把输入塑造成我们想要的样子。另一个实用技术是限流（throttling），将在 5.4 节介绍。

Buffer 和 Windows 都有其他重载，可用在更高级的场合。参数为 skip 和 timeShift 的重载能创建互相重合的组，还可跳过组之间的元素。还有一些重载可使用委托，可对组的边界进行动态定义。

参阅

5.1 节介绍如何利用事件创建序列。

5.4 节介绍对事件流进行限流。

5.4 用限流和抽样抑制事件流

问题

有时事件来得太快，这是编写响应式代码时经常碰到的问题。一个速度太快的事件流可导致程序的处理过程崩溃。

解决方案

Rx 专门提供了几个操作符，用来对付大量涌现的事件数据。Throttle 和 Sample 这两个操作符提供了两种不同方法来抑制快速涌来的输入事件。

Throttle 建立了一个超时窗口，超时期限可以设置。当一个事件到达时，它就重新开始计时。当超时期限到达时，它就把窗口内到达的最后一个事件发布出去。

下面的例子也是监视鼠标移动，但使用了 Throttle，在鼠标保持静止 1 秒后才报告最近一条移动事件。

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
```

```
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(x => x.EventArgs.GetPosition(this))
        .Throttle(TimeSpan.FromSeconds(1))
        .Subscribe(x => Trace.WriteLine(
            DateTime.Now.Second + ": Saw " + (x.X + x.Y)));
}
```

输出结果依赖于鼠标的实际动作，我的测试结果是这样：

```
47: Saw 139
49: Saw 137
51: Saw 424
56: Saw 226
```

Throttle 常用于类似“文本框自动填充”这样的场合，用户在文本框中输入文字，当他停止输入时，才需要进行真正的检索。

为抑制快速运动的事件序列，Sample 操作符使用了另一种方法。Sample 建立了一个有规律的超时时间段，每个时间段结束时，它就发布该时间段内最后的一条数据。如果这个时间段没有数据，就不发布。

下面的例子捕获鼠标移动，每隔一秒采样一次。 与 Throttle 不同，使用 Sample 的例子中，不需要让鼠标静止一段时间，就可要看到结果。

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(x => x.EventArgs.GetPosition(this))
        .Sample(TimeSpan.FromSeconds(1))
        .Subscribe(x => Trace.WriteLine(
            DateTime.Now.Second + ": Saw " + (x.X + x.Y)));
}
```

我先让鼠标静止几秒钟，然后连续移动，得到了下面的输出结果：

```
12: Saw 311
17: Saw 254
18: Saw 269
19: Saw 342
20: Saw 224
21: Saw 277
```

讨论

对于快速涌来的输入，限流和抽样是很重要的两种工具。别忘了还有一个过滤输入的简单

方法，就是采用标准 LINQ 的 Where 操作符。可以这样说，Throttle 和 Sample 操作符与 Where 基本差不多，唯一的区别是 Throttle、Sample 根据时间段过滤，而 Where 根据事件的数据过滤。在抑制快速涌来的输入流时，这三种操作符提供了三种不同的方法。

参阅

5.1 节介绍如何创建事件序列。

5.2 节介绍如何修改引发事件的上下文。

5.5 超时

问题

我们希望事件能在预定的时间内到达，即使事件不到达，也要确保程序能及时进行响应。通常此类事件是单一的异步操作（例如，等待 Web 服务请求的响应）。

解决方案

Timeout 操作符在输入流上建立一个可调节的超时窗口。一旦新的事件到达，就重置超时窗口。如果超过期限后事件仍没到达，Timeout 操作符就结束流，并产生一个包含 TimeoutException 的 OnError 通知。

下面的代码向一个域名发出 Web 请求，并使用 1 秒作为超时值：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var client = new HttpClient();
    client.GetStringAsync("http://www.example.com/").ToObservable()
        .Timeout(TimeSpan.FromSeconds(1))
        .Subscribe(
            x => Trace.WriteLine(DateTime.Now.Second + ": Saw " + x.Length),
            ex => Trace.WriteLine(ex));
}
```

Timeout 非常适用于异步操作，例如 Web 请求，但它也能用于任何事件流。下面的例子在监视鼠标移动时使用 Timeout，使用起来更加简单：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(x => x.EventArgs.GetPosition(this))
```

```
            .Timeout(TimeSpan.FromSeconds(1))
            .Subscribe(
                x => Trace.WriteLine(DateTime.Now.Second + ": Saw " + (x.X + x.Y)),
                ex => Trace.WriteLine(ex));
    }
```

我移动了一下鼠标，然后停止 1 秒，得到如下结果：

```
16: Saw 180
16: Saw 178
16: Saw 177
16: Saw 176
System.TimeoutException: The operation has timed out.
```

值得注意的是，一旦向 OnError 发送 TimeoutException，整个事件流就结束了，不会继续传来鼠标移动事件。为了阻止这种情况出现，Timeout 操作符具有重载方式，在超时发生时用另一个流来替代，而不是抛出异常并结束流。

下面的例子，在超时之前观察鼠标移动，超时发生后进行切换，观察鼠标点击：

```
    private void Button_Click(object sender, RoutedEventArgs e)
    {
        var clicks = Observable.FromEventPattern
            <MouseButtonEventHandler, MouseButtonEventArgs>(
                handler => (s, a) => handler(s, a),
                handler => MouseDown += handler,
                handler => MouseDown -= handler)
            .Select(x => x.EventArgs.GetPosition(this));

        Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
                handler => (s, a) => handler(s, a),
                handler => MouseMove += handler,
                handler => MouseMove -= handler)
            .Select(x => x.EventArgs.GetPosition(this))
            .Timeout(TimeSpan.FromSeconds(1), clicks)
            .Subscribe(
                x => Trace.WriteLine(
                    DateTime.Now.Second + ": Saw " + x.X + "," + x.Y),
                ex => Trace.WriteLine(ex));
    }
```

我先移动一下鼠标，停止 1 秒，然后在两个不同的位置点击。下面的输出表明，超时发生前鼠标移动事件在进行快速移动，超时后变成两个鼠标点击事件：

```
49: Saw 95,39
49: Saw 94,39
49: Saw 94,38
49: Saw 94,37
53: Saw 130,141
55: Saw 469,4
```

讨论

`Timeout` 操作符对优秀的程序来说是十分必要的，因为我们总是希望程序能及时响应，即使外部环境不理想。它可用于任何事件流，尤其是在异步操作时。需要注意，此时内部的操作并没有真正取消，操作将继续执行，直到成功或失败。

参阅

5.1 节介绍如何利用事件创建序列。

7.6 节介绍如何把异步代码封装成 Observable 对象事件流。

9.6 节介绍收到 `CancellationToken` 时如何从序列中退订。

9.3 节介绍用 `CancellationToken` 来实现超时功能。

第6章

测试技巧

测试是保证软件质量必不可少的环节。近年来，提倡单元测试的人越来越多，到处都能听到有关单元测试的讨论。有人提倡测试驱动型的开发模式，以保证软件测试和开发同步进行、同时完成。大家都知道单元测试在保证代码质量和整个开发过程中的作用，然而大多数开发人员直到今天都没有真正编写过单元测试。

我建议大家至少写一些单元测试，首先从自己觉得最没信心的代码开始。根据我个人的经验，单元测试主要有两大好处。

(1) 更好地理解代码。你是否遇到过这种情况：你了解程序的某个部分能正常运行，却对它的实现原理一无所知。当软件出现了令你不可思议的错误时，这种疑问常常占据你的内心深处。要理解那些特别“难”的代码的内部机理，编写单元测试就是一个很好的办法。编写描述代码行为的单元测试之后，就不会觉得这部分代码神秘了。编写一批单元测试后，最终就能搞清那些代码的行为，以及它们和其他代码之间的依赖关系。

(2) 修改代码时更有把握。迟早会有那么一天，你会因为有功能需求而必须修改那些“恐怖”的代码，你将无法继续假装它不存在。（我了解那种感觉。我经历过！）最好提前做好准备：在此类需求到来之前，为那些恐怖的代码编写单元测试。提前准备，以免以后麻烦。如果你的单元测试是完整的，你就相当于有了一个早期预警系统，如果修改后的代码影响到已有功能时，它就会立即发出警告。

不管是你自己还是其他人的代码，都能获得上述好处。我敢肯定单元测试还能带来其他好处。单元测试能减少错误出现的频率吗？很有可能。单元测试能减少项目的整体时间吗？有可能。但是我在上面列出的几条好处是肯定会有的。我每次编写单元测试时都能感受到。

因此，我强烈推荐单元测试。

本章的内容全部是关于测试的。很多开发人员（甚至包括经常编写单元测试的人）都逃避并发代码的单元测试，因为他们总觉得非常难。然而本章的内容将会告诉大家，并发代码的单元测试并没有想象中那么难。现在的语言功能和开发库，例如 async 和 Rx，在测试的方便性方面做了很多考虑，并且确实能体现出这点。我建议大家使用本章的方法编写单元测试，尤其是并发编程的新手（就是认为新并发代码“很难”或“可怕”的人）。

6.1 async方法的单元测试

问题

需要对 async 方法进行单元测试。

解决方案

现在大多数单元测试框架都支持 async Task 类型的单元测试，包括 MSTest、NUnit、xUnit。从 Visual Studio 2012 开始，MSTest 才支持 async Task 类型的单元测试，因此需要将老版本升级到最新版本。

下面是一个 async 类型 MSTest 单元测试的例子：

```
[TestMethod]
public async Task MyMethodAsync_ReturnsFalse()
{
    var objectUnderTest = ...;
    bool result = await objectUnderTest.MyMethodAsync();
    Assert.IsFalse(result);
}
```

单元测试框架检测到方法的返回类型是 Task，会自动 加上 await 等待任务完成，然后将测试结果标记为“成功”或“失败”。

如果单元测试框架不支持 async Task 类型的单元测试，就需要做一些额外的修改才能等待异步操作。其中一种做法是使用 Task.Wait，并在有错误时拆开 AggregateException 对象。我的建议是使用 NuGet 包 Nito.AsyncEx 中的 AsyncContext 类：

```
[TestMethod]
public void MyMethodAsync_ReturnsFalse()
{
    AsyncContext.Run(async () =>
    {
        var objectUnderTest = ...;
        bool result = await objectUnderTest.MyMethodAsync();
        Assert.IsFalse(result);
```

```
        });
    }
```

`AsyncContext.Run` 会等待所有异步方法完成。

讨论

模拟（mocking）异步方法间的依赖关系，虽说它给人的第一感觉是有点别扭，但至少可以测试某些方法如何响应同步成功（用 `Task.FromResult` 模拟）、同步出错（用 `TaskCompletionSource<T>` 模拟）以及异步成功（用 `Task.Yield` 模拟，并返回一个值），并且它在做这些测试时，是一个很好的办法。

跟同步代码相比，在测试异步代码时会出现更多的死锁和竞态条件。我发现，对每个测试进行超时设置很有用。在 Visual Studio 中，可以在解决方案中加一个测试设置文件，用来对每个测试设置独立的超时参数。这个参数的默认值是很大的，我通常将每一个测试的超时参数设成 2 秒。

`AsyncContext` 类在 NuGet 包的 `Nito.AsyncEx` 中。

参阅

6.2 节介绍对预计失败的异步方法进行单元测试。

6.2 预计失败的async方法的单元测试

问题

需要编写一个单元测试，用来检查 `async Task` 方法的一个特定错误。

解决方案

对于桌面程序或服务器程序，MSTest 就可以用常规的 `ExpectedExceptionAttribute` 进行错误测试：

```
// 不推荐用这种方法，原因在后面。
[TestMethod]
[ExpectedException(typeof(DivideByZeroException))]
public async Task Divide_WhenDenominatorIsZero_ThrowsDivideByZero()
{
    await MyClass.DivideAsync(4, 0);
}
```

但是这种方法并不是最好的。一方面，Windows 应用商店并没有 ExpectedException 支持单元测试。另一个本质的原因是 ExpectedException 的设计非常糟糕。单元测试中调用的任何方法都可以抛出这个异常。更好的设计是检查抛出异常的那段代码，而不是检查整个单元测试。

微软公司已经在向这个方向努力了，在 Windows 应用商店单元测试中去掉了 ExpectedException，改成用 Assert.ThrowsException<TException>，使用方法如下：

```
[TestMethod]
public async Task Divide_WhenDenominatorIsZero_ThrowsDivideByZero()
{
    await Assert.ThrowsException<DivideByZeroException>(async () =>
    {
        await MyClass.DivideAsync(4, 0);
    });
}
```

千万别忘了对 ThrowsException 返回的 Task 使用 await。这样才可以传递出所有监测到的出错信息。如果忘记使用 await 并且忽视了编译器的警告，那么不管被测试的方法是否真的正确，单元测试就会一直显示测试成功且不给任何提示。

可惜，微软只在 Windows 应用商店单元测试项目中加入了 ThrowsException，到目前为止其他几种单元测试框架并没有与 ThrowsException 等效的、兼容 async 的方法。这时我们可以自行创建这样的方法：

```
/// <summary>
/// 确保一个异步委托抛出异常。
/// </summary>
/// <typeparam name="TException">
/// 所预计异常的类型。
/// </typeparam>
/// <param name="action">被测试的异步委托</param>
/// <param name="allowDerivedTypes">
/// 是否接受派生的类。
/// </param>
public static async Task ThrowsExceptionAsync<TException>(Func<Task> action,
    bool allowDerivedTypes = true)
{
    try
    {
        await action();
        Assert.Fail("Delegate did not throw expected exception " +
            typeof(TException).Name + ".");
    }
    catch (Exception ex)
    {
        if (allowDerivedTypes && !(ex is TException))
```

```
                Assert.Fail("Delegate threw exception of type " + ex.GetType().Name +
                    ", but " + typeof(TException).Name +
                    " or a derived type was expected.");
            if (!allowDerivedTypes && ex.GetType() != typeof(TException))
                Assert.Fail("Delegate threw exception of type " + ex.GetType().Name +
                    ", but " + typeof(TException).Name + " was expected.");
        }
    }
```

调用这个方法跟 Windows 应用商店的 MSTest 方法 `Assert.ThrowsException<TException>` 一样。千万别忘记对返回值使用 `await`！

讨论

对错误处理进行测试，与测试正确的场景一样重要。甚至有人认为前者更重要，因为正确场景是每个人在软件发布前就试过的。如果软件的运行情况很怪异，可能是因为出现了以前没预料到的错误情形。

不过，我建议大家不要使用 `ExpectedException`。它更适用于测试某个特定点抛出的异常，而不是整个测试过程中随时会抛出的异常。不用 `ExpectedException` 的话，就可改用 `ThrowsException`（或者单元测试框架中类似的方法），或者使用它的另一种实现 `ThrowsExceptionAsync`。

参阅

6.1 节介绍异步方法单元测试的基础知识。

6.3 async void方法的单元测试

问题

需要对一个 `asnyc void` 类型的方法做单元测试。

解决方案

停。

要尽最大可能避免这个问题，而不是去解决它。只要有可能把 `async void` 方法改成 `async Task`，那就得改。

如果一个方法必须采用 `async void`（例如为满足某个接口方法的特征），那可考虑编写两个方法：一个包含所有逻辑的 `async Task` 方法和一个 `async void` 方法。这个 `async void` 方法只是做一个简单封装，即调用 `async Task` 方法，并用 `await` 等待结果。这样，`async void` 方法可满足格式要求，而 `async Task` 方法（包含所有逻辑）可用于测试。

如果真的不可能修改这个方法，并且确实必须对一个 async void 方法做单元测试，可试试这个方法，使用 Nito.AsyncEx 类库的 AsyncContext 类：

```
// 不推荐用这种方法，原因见前面。
[TestMethod]
public void MyMethodAsync_DoesNotThrow()
{
    AsyncContext.Run(() =>
    {
        var objectUnderTest = ...;
        objectUnderTest.MyMethodAsync();
    });
}
```

这个 AsyncContext 类会等待所有异步操作完成（包括 async void 方法），再将异常传递出去。

AsyncContext 在 NuGet 包 Nito.AsyncEx 中。

讨论

在 async 代码中，关键准则之一就是避免使用 async void。我非常建议大家在对 async void 方法做单元测试时进行代码重构，而不是使用 AsyncContext。

参阅

6.1 节介绍异步方法单元测试的基础知识。

6.4 数据流网格的单元测试

问题

程序中有一个数据流网格，需要对它进行正确性验证。

解决方案

数据流网格是独立的：有自己的生命周期，并且本质上就是异步的。自然而然，它的测试方法就是使用异步的单元测试。下面的单元测试验证 4.6 节中的自定义数据流块：

```
[TestMethod]
public async Task MyCustomBlock_AddsOneToDataItems()
{
```

```
    var myCustomBlock = CreateMyCustomBlock();

    myCustomBlock.Post(3);
    myCustomBlock.Post(13);
    myCustomBlock.Complete();

    Assert.AreEqual(4, myCustomBlock.Receive());
    Assert.AreEqual(14, myCustomBlock.Receive());
    await myCustomBlock.Completion;
}
```

可惜的是，对错误进行单元测试就没那么简单了。这是因为在数据流网格中，异常信息在块之间传递时会被一层一层地封装在另一个 AggregateException 中。下面的例子使用了一个辅助方法，以确保一个异常在丢弃数据之后，再在自定义块之间传递。

```
[TestMethod]
public async Task MyCustomBlock_Fault_DiscardsDataAndFaults()
{
    var myCustomBlock = CreateMyCustomBlock();

    myCustomBlock.Post(3);
    myCustomBlock.Post(13);
    myCustomBlock.Fault(new InvalidOperationException());

    try
    {
        await myCustomBlock.Completion;
    }
    catch (AggregateException ex)
    {
        AssertExceptionIs<InvalidOperationException>(
            ex.Flatten().InnerException, false);
    }
}

public static void AssertExceptionIs<TException>(Exception ex,
    bool allowDerivedTypes = true)
{
    if (allowDerivedTypes && !(ex is TException))
        Assert.Fail("Exception is of type " + ex.GetType().Name + ", but "
        + typeof(TException).Name + " or a derived type was expected.");
    if (!allowDerivedTypes && ex.GetType() != typeof(TException))
        Assert.Fail("Exception is of type " + ex.GetType().Name + ", but "
        + typeof(TException).Name + " was expected.");
}
```

讨论

直接对数据流网格做单元测试是可行的，但有些别扭。如果网格是一个大组件的组成部分，只对这个大组件做单元测试（隐式的测试网格），那样会比较简单。但如果开发可重用的自定义块或网格，那就应该像前面那样做单元测试。

参阅

6.1 节介绍异步方法单元测试的基础知识。

6.5 Rx Observable对象的单元测试

问题

程序中用到了 IObservable<T>，需要对这部分程序做单元测试。

解决方案

响应式扩展（Reactive Extension）有很多产生序列的操作符（如 Return），还有操作符可把响应式序列转换成普通集合或项目（如 SingleAsync）。我们可使用 Return 等操作符创建 Observable 对象依赖项的存根（stub），用 SingleAsync 等操作符来测试输出。

看下面的代码，它把一个 HTTP 服务作为依赖项，并且在调用 HTTP 时使用了一个超时：

```
public interface IHttpService
{
    IObservable<string> GetString(string url);
}

public class MyTimeoutClass
{
    private readonly IHttpService _httpService;

    public MyTimeoutClass(IHttpService httpService)
    {
        _httpService = httpService;
    }

    public IObservable<string> GetStringWithTimeout(string url)
    {
        return _httpService.GetString(url)
            .Timeout(TimeSpan.FromSeconds(1));
    }
}
```

我们要测试的代码是 MyTimeoutClass，它消耗一个 Observable 对象依赖项，生成一个 Observable 对象作为输出。

Return 操作符创建一个只有一个元素的冷序列（cold sequence），可用它来构建简单的存根（stub）。SingleAsync 操作符返回一个 Task<T> 对象，该对象在下一个事件到达时完成。SingleAsync 可用来做简单的单元测试，如下所示：

```
class SuccessHttpServiceStub : IHttpService
{
    public IObservable<string> GetString(string url)
```

```
        {
            return Observable.Return("stub");
        }
    }

    [TestMethod]
    public async Task MyTimeoutClass_SuccessfulGet_ReturnsResult()
    {
        var stub = new SuccessHttpServiceStub();
        var my = new MyTimeoutClass(stub);

        var result = await my.GetStringWithTimeout("http://www.example.com/")
            .SingleAsync();

        Assert.AreEqual("stub", result);
    }
```

存根代码中另一个重要操作符是 `Throw`，它返回一个以错误结束的 `Observable` 对象。这样我们也可对有错误的场景做单元测试。下面的例子使用了 6.2 节中的辅助方法 `ThrowsExceptionAsync`：

```
    private class FailureHttpServiceStub : IHttpService
    {
        public IObservable<string> GetString(string url)
        {
            return Observable.Throw<string>(new HttpRequestException());
        }
    }

    [TestMethod]
    public async Task MyTimeoutClass_FailedGet_PropagatesFailure()
    {
        var stub = new FailureHttpServiceStub();
        var my = new MyTimeoutClass(stub);

        await ThrowsExceptionAsync<HttpRequestException>(async () =>
        {
            await my.GetStringWithTimeout("http://www.example.com/")
                .SingleAsync();
        });
    }
```

讨论

`Return` 和 `Throw` 操作符很适合创建 observable 对象的存根，而要在 `async` 单元测试中测试 observable 对象，比较容易的方法就是使用 `SingleAsync`。对于简单的 observable 对象，这两个操作符结合起来使用效果很好。但如果 observable 对象与时间有关，它们就不那么管用了。例如要测试 `MyTimeoutClass` 类的超时能力，单元测试就必须真正地等待那么长时间。一旦增加更多的单元测试，这种方式就不大合适了。6.6 节介绍一种特殊的方法，Reactive Extensions 可以把时间本身排除在外。

参阅

6.1 节介绍对 async 方法做单元测试，这与用 await SingleAsync 进行单元测试非常相似。

6.6 节介绍对依赖于时间的 observable 序列做单元测试。

6.6 用虚拟时间测试Rx Observable对象

问题

需要写一个不依赖于时间的单元测试，来测试一个依赖于时间的 observable 对象。如 observable 对象中使用了超时、窗口 / 缓冲、限流 / 抽样等方法，那它就是依赖于时间的。我们要对它们做单元测试，但要求运行时间不能太长。

解决方案

我们当然可以让延迟函数在单元测试中运行。但是这样做会产生两个问题：1) 单元测试的运行时间会很长；2) 因为所有的单元测试是同时运行的，这样做会导致竞态条件，无法预测运行时机。

Rx 库在设计时就考虑到了测试问题。实际上，Rx 库本身就已经过大量单元测试。为了解决上面的问题，Rx 引入了调度器（scheduler）这一概念，每个与时间有关的 Rx 操作都在实现时使用了这个抽象的调度器。

要让 observable 对象便于测试，就要允许调用它的程序指定调度器。例如可以使用 6.5 节的 MyTimeoutClass，并加上一个调度器：

```
public interface IHttpService
{
    IObservable<string> GetString(string url);
}

public class MyTimeoutClass
{
    private readonly IHttpService _httpService;

    public MyTimeoutClass(IHttpService httpService)
    {
        _httpService = httpService;
    }

    public IObservable<string> GetStringWithTimeout(string url,
        IScheduler scheduler = null)
    {
        return _httpService.GetString(url)
            .Timeout(TimeSpan.FromSeconds(1), scheduler ?? Scheduler.Default);
```