

RxSwift中的那些“术语”到底在说什么？

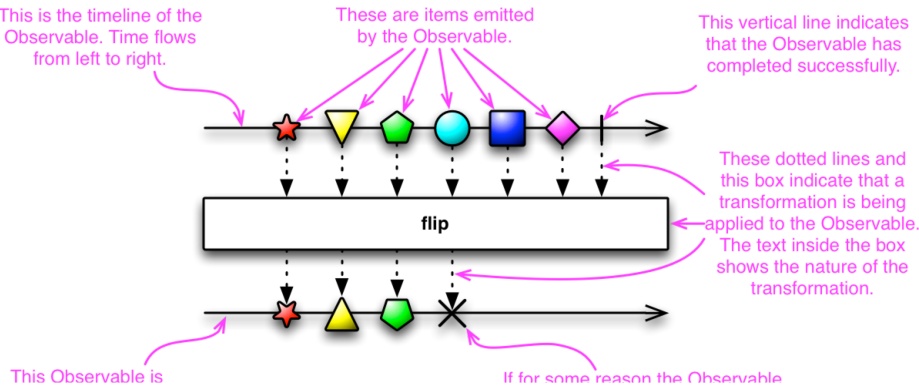
⌕ Back to series (</series/rxswift-101/>)

在这一节里，我们先来熟悉一下reactive programming中的常用术语，这些术语表达的大多是我们熟悉的概念，却有一个我们不太熟悉的名字，从某种意义上说，这给我们入门RxSwift也带来了不少麻烦。

- 🔍 字号
- 🔍 字号
- 🖌 默认主题
- 🖌 金色主题
- 🖌 暗色主题

从“以时间为索引的常量队列”开始 - Observable

第一个要介绍的，就是我们在之前的例子中提到的“以时间为索引的常量队列”。在RxSwift里，这种概念叫做**Observable**。在ReactiveX对Observable的说明中 (<http://reactivex.io/documentation/observable.html>)，我们可以看到一张这样的图：



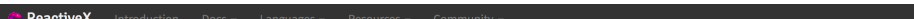
其中，最上面的一排，就是一个**Observable**。从左到右，表示时间由远及近的流动过程。上面的每一个形状，就表示在“某个时间点发生的事件”，而最右边的竖线则表示事件成功结束。因此，我们之前过滤用户输入的例子，也可以表示成这样：



看到这里，你可能已经有点儿着急了，你说的这些我都明白，代码呢？该怎么写呢？先别急，我们再来看一个概念：**Operator**。

理解operators

在ReactiveX官网可以看到 (<http://reactivex.io/documentation/operators.html>)，**operators**分为两大类：一类用于创建Observable；



这些不同的创建方法可以针对不同的事件流类型生成Observable。另一类是接受Observable作为参数，并返回一个新的Observable；

Transforming Observables

Operators that transform items that are emitted by an Observable.

- **Buffer** — periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time
- **FlatMap** — transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable
- **GroupBy** — divide an Observable into a set of Observables that each emit a different group of items from the original Observable, organized by key
- **Map** — transform the items emitted by an Observable by applying a function to each item
- **Scan** — apply a function to each item emitted by an Observable, sequentially, and emit each successive value
- **Window** — periodically subdivide items from an Observable into Observable windows and emit these windows rather than emitting the items one at a time

它们有点儿类似我们对集合类型使用的各种变形方法，用于在序列中找到我们希望处理的事件，稍后，我们就会看到一个更具体的例子。

创建一个事件序列

接下来，我们先看一些简单的创建Observable的方法。用任何一种之前我们提过的方法创建一个包含RxSwift的项目，这里我们用了SPM。

为了创建一个包含字符1-9的序列，我们可以在Sources/main.swift中添加下面的代码：

```
import RxSwift

Observable.of("1", "2", "3", "4", "5", "6", "7", "8", "9")

// Or

Observable.from(["1", "2", "3", "4", "5", "6", "7", "8", "9"])
```

这里，我们就用了两个operator：

- **of**：用固定数量的元素生成一个Observable；
- **from**：用一个 Sequence 类型的对象创建一个Observable；

但这两个operator返回的结果是一样的，都是一个包含字符1-9的Observable：



对事件序列进行处理

定义好事件序列之后，我们就可以处理过滤偶数的需求了。之前我们提到过，对序列的加工是通过另外一类operator完成的。思路，和我们之前过滤数组是一样的，先把Observable中的元素都变成整数：

```
_ = Observable.of("1", "2", "3", "4", "5", "6", "7", "8", "9")
    .map { Int($0) }
```

这里，**map** 就是一个可以对Observable中的元素变形的operator，它返回一个新的Observable对象。因此，我们可以把多个这种加工Observable的方法串联起来。

例如，进一步在变形后的Observable中，找出所有偶数：

```
_ = Observable.of("1", "2", "3", "4", "5", "6", "7", "8", "9")
    .map { Int($0) }
    .filter { $0 != nil && $0! % 2 == 0 }
```

这样，**filter** 这个operator返回的，就是一个只包含偶数的Observable了。

尽管上面 **map** 和 **filter** 这两个operator和集合中的 **map** 和 **filter** 方法非常类似，但它们执行的逻辑却截然不同。

调用集合类型的 **map** 和 **filter** 方法，表达的是同步执行的概念，在调用方法的同时，集合就被立即加工处理了。

但我们创建的Observable，表达的是异步操作。**Observable**中的每一个元素，都可以理解为一个异步发生的事件。因此，当我们对Observable调用 **map** 和 **filter** 方法时，只表示我们要对事件序列中的元素进行处理的逻辑，而并不会立即对Observable中的元素进行处理。

为了验证这个结论，我们可以在筛选偶数的时候打印个消息：

```
- = Observable.of("1", "2", "3", "4", "5", "6", "7", "8", "9")
    .map { Int($0) }
    .filter {
        if let item = $0, item % 2 == 0 {
            print("Even: \(item)")
            return true
        }

        return false
    }
}
```

然后, 执行 `swift build` 编译, 当我们执行编译结果的时候, 不会在控制台上打印任何消息。也就是说, 我们没有实际执行任何的筛选逻辑。

如何“订阅”事件?

那么, 真正的筛选是什么时候发生的呢? 答案是, 真的有人对这个事件感兴趣的时候。也就是说, 有人“订阅”这个Observable中的事件的时候, 像这样:

```
var evenNumberObservable =
    Observable.of("1", "2", "3", "4", "5", "6", "7", "8", "9")
        .map { Int($0) }
        .filter {
            if let item = $0, item % 2 == 0 {
                print("Even: \(item)")
                return true
            }

            return false
        }

evenNumberObservable.subscribe { event in
    print("Event: \(event)")
}
```

这表示的, 就是我们“从头至尾”关注了 `evenNumberObservable` 这个序列中的所有事件。重新编译执行一下, 就可以看到筛选的过程和结果了, 我们关注到了这个筛选事件中的所有偶数:

```
➔ ObservableDemo .build/debug/ObservableDemo
Event: 2
Event: next(Optional(2))
Event: 4
Event: next(Optional(4))
Event: 6
Event: next(Optional(6))
Event: 8
Event: next(Optional(8))
Event: completed
```

那么, 除了“全程关注”的人之外, 还有一类是“半路来的人”, 对于这些人, 就只能看到他们关注到 `evenNumberObservable` 之后, 发生的事件了, 我们可以用下面的代码, 来理解这个场景:

```
evenNumberObservable.skip(2).subscribe { event in
    print("Event: \(event)")
}
```

这里, 我们用了另外一个operator `skip` 来模拟“半路关注”的情况。 `skip(2)` 可以让订阅者“错过”前2次发生的事件。此时, 对这个订阅者而言, 他就完全不知道之前还过滤出了2个偶数, 他看到的结果就是这样的:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
➔ ObservableDemo .build/debug/ObservableDemo
Event: 2
Event: 4
Event: 6
Event: next(Optional(6))
Event: 8
Event: next(Optional(8))
Event: completed
```

把这两次订阅放在一个图里, 就是这样:

通过这两个例子，我们要表达的最重要的一个思想，就是Observable中的每一个元素都表示一个“异步发生的事件”这样的概念，operator对Observable的加工是在订阅的时候发生的。这种只有在订阅的时候才emit事件的Observable，有一个专有的名字，叫做**Cold Observable**。

言外之意，就是也有一种Observable是只要创建了就会自动emit事件的，它们叫做**Hot Observable**。在后面的内容中，我们会看到这类事件队列的用法。

subscribe也是一个operator

在刚才的例子里，还有一点值得我们注意的就是在结尾，会有一个 Event: `completed`。这就是在我们之前Observable示意图中的竖线，表示这个Observable事件流成功结束了。

实际上，我们使用的 `subscribe`，也是一个operator，用于把事件的订阅者（Observer）和事件的产生者（Observable）关联起来。而Observer和Observable之间，有着下面的约定：

- 当Observable正常发送事件时，会调用Observer提供的 `onNext` 方法，这个过程习惯上叫做 **emissions**；
- 当Observable成功结束时，会调用Observer提供的 `onCompleted` 方法；因此，在最后一次调用 `onNext` 之后，就会调用 `onCompleted`；
- 当Observable发生错误时，就会调用Observer提供的 `onError` 方法，并且，一旦发生错误，就不会再继续发送其它事件了。对于调用 `onComplete` 和 `onNext` 的过程，习惯上叫做 **notifications**；

在RxSwift里，还有一个约定，叫做 `onDisposed`，指的是Observable使用的资源被回收的时候，会调用Observer提供的 `onDisposed` 方法。那么，究竟什么是dispose呢？为什么我们需要它？这都要从Observable的类型说起。

理解Observable dispose

一直以来，我们使用的Observable都是有限序列，对 `evenNumberObservable` 来说，当它emit了所有的数字之后，就自动结束了，此时，它占用的资源就会被回收，这很好理解。

但事情并不总是如此，有些事件队列是无限的，例如像下面这样：

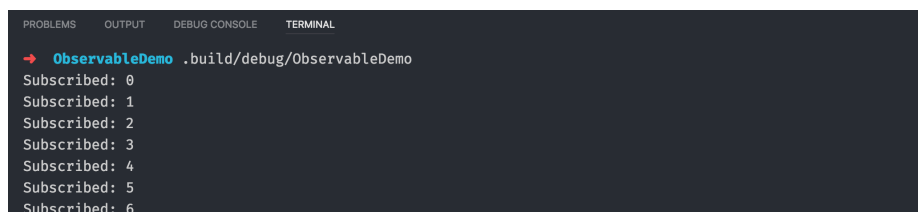
```
_ = Observable<Int>.interval(1, scheduler: MainScheduler.instance)
    .subscribe(onNext: { print("Subscribed: \($0)") })

dispatchMain()
```

我们用 `interval` 在主线程定义了一个每隔1秒发送一个整数的事件序列。为了让它可以一直emit事件，最后，我们调用了 `dispatchMain` 让程序保持不退出。

为了调用 `dispatchMain`，要在文件开始 `import Foundation`。

然后，把之前订阅 `evenNumberObservable` 的代码注释掉，重新编译执行，就能得到类似下面这样的结果：



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
→ ObservableDemo .build/debug/ObservableDemo
Subscribed: 0
Subscribed: 1
Subscribed: 2
Subscribed: 3
Subscribed: 4
Subscribed: 5
Subscribed: 6
```

可以想到,如果我们不强制退出,这会是一个一直执行下去的事件序列。它在程序退出之前,会一直占用着系统资源,在绝大多数时候,这不是我们想要的结果。对于这种无限事件序列,如果我们要在不用的时候回收掉它的资源,就得这样。

例如,假设5秒后,这个事件序列就不再需要了:

```
public func delay(_ delay: Double,
                 closure: @escaping () -> Void) {

    DispatchQueue.main.asyncAfter(
        deadline: .now() + delay) {
        closure()
    }
}

let disposable =
    Observable<Int>.interval(1, scheduler: MainScheduler.instance)
        .subscribe(
            onNext: { print("Subscribed: \($0)") },
            onDisposed: { print("The queue was disposed.") })

delay(5) {
    disposable.dispose()
}
```

在上面的代码里:

- 首先,我们定义了一个helper function `delay`,它可以在特定时间之后,执行我们指定的 closure;
- 其次,我们把 `subscribe` 的返回值保存在了一个叫做 `disposable` 的变量里,我们可以把它理解为是一个订阅对象,我们可以通过这个对象,来取消订阅。同时,我们还指定了 `onDisposed` 参数,用来在事件序列被回收时,获得通知;
- 最后,当我们使用 `disposable.dispose()` 的时候,就表示我们要“退订”这个计时事件了。退订后,原来的事件序列就不再继续emit事件了,它占用的资源也会被回收;

编译执行一下,就会看到这样的结果:

```
ObservableDemo .build/debug/ObservableDemo
Subscribed: 0
Subscribed: 1
Subscribed: 2
Subscribed: 3
Subscribed: 4
The queue was disposed.
```

但通常,这样单独对 `subscribe` 的返回值调用 `dispose()` 方法并不是一个好的编码习惯。RxSwift提供了一个类似RAII的机制,叫做 `DisposeBag`,我们可以把所有的订阅对象放在一个 `DisposeBag` 里,当 `DisposeBag` 对象被销毁的时候,它里面“装”的所有订阅对象就会自动取消订阅,对应的事件序列的资源也就被自动回收了。为了理解这个用法,我们可以把之前的代码改成这样:

```
var bag = DisposeBag()

Observable<Int>.interval(1, scheduler: MainScheduler.instance)
    .subscribe(
        onNext: { print("Subscribed: \($0)") },
        onDisposed: { print("The queue was disposed.") })
    .disposed(by: bag)

delay(5) {
    bag = DisposeBag()
}
```

这次,我们直接串联了 `subscribe` 的返回值,调用 `disposed(by:)` 方法,把返回的订阅对象直接“装”在了 `bag` 里。并且,在 `delay` 的 closure 参数里,我们通过让 `bag` 等于一个新的 `DisposeBag` 对象,模拟了 `DisposeBag` 对象被销毁的场景。编译执行一下,就能看到和之前同样的结果了:

```
ObservableDemo .build/debug/ObservableDemo
Subscribed: 0
Subscribed: 1
Subscribed: 2
Subscribed: 3
Subscribed: 4
The queue was disposed.
```

以上,就是和 `Observable` `dispose` 相关的概念。

What's next?

看到这里，我们对RxSwift中的基本概念就有了一个比较全面的了解了。什么是Observable，它和普通序列的区别是什么？Operators的功能是什么？什么是Observer，如何订阅各种事件？Dispose回收的究竟是什么，该如何正确使用它？当我们对这些问题的答案成竹在胸的时候，就可以自己在RxSwift的官方Playground中去学习各种相关的细节了。接下来，我们将在一些具体的开发场景里，来体会不同的operators的用法。

◀ 安装RxSwift的三种不同方式

(<https://www.boxueio.com/series/rxswift-101/ebook/205>)

理解create和debug operator ▶

(<https://www.boxueio.com/series/rxswift-101/ebook/213>)



职场漂泊的你，每天多学一点。

从开发、测试到运维，让技术不再成为你成长的绊脚石。我们用打磨产品的精神去传播知识，把最新的移动开发技术，通过简单的图表，清晰的视频，简明的文字和切实可行的例子一一向你呈现。让学习不仅是一种需求，也是一种享受。

泊学动态

一个工作十年PM终创业的故事（二）(<https://www.boxueio.com/after-the-full-upgrade-to-swift3>)
Mar 4, 2017

人生中第一次创业的"10有"(<https://www.boxueio.com/founder-chat>)
Jan 9, 2016

猎云网采访报道泊学(<http://www.lieyunwang.com/archives/144329>)
Dec 31, 2015

What most schools do not teach(<https://www.boxueio.com/what-most-schools-do-not-teach>)
Dec 21, 2015

一个工作十年PM终创业的故事（一）(<https://www.boxueio.com/founder-story>)
May 8, 2015

泊学相关

关于泊学 >

加入泊学 >

泊学用户隐私以及服务条款([HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE](https://www.boxueio.com/terms-of-service))

版权声明([HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT](https://www.boxueio.com/copyright-statement))

联系泊学

Email: 10[AT]boxue.io (mailto:10@boxue.io)
QQ: 2085489246