

## 为什么需要connectable operator

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

🔍 字号

● 字号

🖌️ 默认主题

🖌️ 金色主题

🖌️ 暗色主题

如果你回顾下之前我们讲过的所有Observable，就会发现它们有一个共同的特征，就是一旦有事件发生，订阅者就会立即订阅到。在RxSwift里，还有一类operator，可以给Observable添加一个特性，让所有的订阅者准备好之后，才可以订阅到其中的事件。

为了理解这些operators的用法，我们先从一个普通的Observable开始。

首先，为了方便延迟订阅Observable，我们添加了一个帮助函数：

```
public func delay(_ delay: Double,
    closure: @escaping () -> Void) {
    DispatchQueue.main.asyncAfter(deadline: .now() + delay) {
        closure()
    }
}
```

其次，我们定义一个每隔1秒钟触发一次事件的Observable，事件值是从1开始累加的整数：

```
let interval = Observable<Int>.interval(1,
    scheduler: MainScheduler.instance)
```

第三，我们用下面的代码，先后两次订阅 interval：

```
_ = interval.subscribe(onNext: {
    print("Subscriber 1: \($0)") })

delay(2) {
    _ = interval.subscribe(onNext: {
        print("Subscriber 2: \($0)") })
}

dispatchMain()
```

为了方便观察结果，我们在最后调用了 dispatchMain 。执行一下，在控制台观察几秒，就会看到类似这样的结果：

```
Subscriber 1: 0 # START
Subscriber 1: 1 # second 1
Subscriber 1: 2 # second 2
Subscriber 2: 0
Subscriber 1: 3 # second 3
Subscriber 2: 1
Subscriber 1: 4 # second 4
Subscriber 2: 2
...
```

就像注释中标记的一样，前两秒，只有Subscribe 1，然后，从第3秒开始，Subscribe 1和Subscribe 2就可以同时订阅到事件了。这就是我们在一开始说的，订阅的时候，Observable中只要有事件，就会立即订阅到。

但是，一个重要的细节是，这两个订阅者并不共享Observable，不同订阅者订阅的，是自己的Observable。从Subscribe 2订阅到的第一个事件是0，就可以确认这一点了。

但是，如果我们希望订阅者在准备就绪后，统一订阅，该怎么办呢？其实，这个诉求里，蕴含着两个含义：

- 一个是，所有的订阅者此时要订阅的就是同一个Observable了，否则也就无从谈起统一订阅；
- 另一个是，我们需要一种方式，掐住Observable的喉咙，不让它发声，直到我们松手为止；

为此，RxSwift提供了几种不同的方式，我们先通过一个最简单的operator：publish 理解这个概念。

## 使用publish发布事件

就如同这个operator的名字，`publish` 用于向所有订阅者“统一”发布事件。我们直接通过代码来了解这个过程。首先，还是定义 `interval`，只不过这次，我们要使用 `publish operator`：

```
let interval = Observable<Int>
    .interval(1, scheduler: MainScheduler.instance)
    .publish()
```

此时，如果执行一下就会发现，我们订阅不到任何事件了。于是，我们就可以在定义好 `interval` 之后，定义各种subscriber了，同样，还是之前先后订阅的两个订阅者，我们不用做任何修改。

然后，我们在Subscriber 1定义好之后，调用 `connect()` 方法对外“发布事件”。然后，在发布之后的两秒，再添加第二个订阅者：

```
_ = interval.subscribe(onNext: {
    print("Subscriber 1: \($0)") })

_ = interval.connect()

delay(2) {
    _ = interval.subscribe(onNext: {
        print("Subscriber 2: \($0)") })
}
```

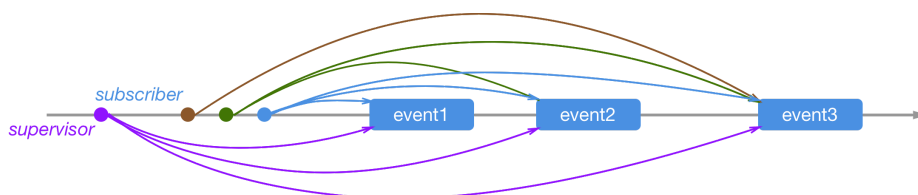
这时，重新执行一下，就会看到下面这样的结果了：

```
Subscriber 1: 0 # START
Subscriber 1: 1 # Second 1
Subscriber 1: 2 # Second 2
Subscriber 2: 2
Subscriber 1: 3 # Second 3
Subscriber 2: 3
Subscriber 1: 4 # Second 4
Subscriber 2: 4
...
```

和我们之前的例子一个本质的不同，就是使用了 `publish` 之后，`Subscribe 1`和`Subscribe 2`是共享同一个`Observable`的。这从`Subscribe 2`第一次订阅到的值是1而不是0就可以确认了。

## 使用multicast operator

除了使用 `connect` 方法控制开始订阅的时机之外，有时，我们还会面对另外一种需求，就是一方面随着事情的进展控制订阅者的数量，另一方面还要有一个“监管机构”一样的订阅者，在每件事件发生的时候，得到通知。我们用一个图来理解这个过程：



如图所示，图中，紫色的圆点表示监管者，它需要了解发生的每一个事件；而棕绿蓝色的三个圆点则表示三个订阅者。现在，假设我们希望控制`event1`只有1个人处理，如果他搞不定，就升级为`event2`，此时有两个人处理，如果还搞不定，就升级成`event3`，由三个人来处理。

理解了场景之后，该怎么办呢？

这时，我们就可以使用 `multicast operator`。它可以原事件序列中的事件通过另外一个`subject`对象代为传递。我们一步步通过代码来理解这个过程。

首先，当然还是之前使用的 `interval Observable`，它表示我们关注的原始事件序列：

```
let interval = Observable<Int>
    .interval(1, scheduler: MainScheduler.instance)
```

其次，我们定义一个事件值和 `interval` 相同的 `PublishSubject`，它用于向监管者报告事件：

```
let supervisor = PublishSubject<Int>()

_ = supervisor.subscribe(onNext: {
    print("Supervisor: event \($0)") })
```

第三, 对 interval 使用 multicast operator:

```
let interval = Observable<Int>
    .interval(1, scheduler: MainScheduler.instance)
    .multicast(supervisor)
```

第四, 我们来按照图中的顺序, 定义三个订阅者, 通过它们的订阅, 来模拟事件逐步升级, 被更多人订阅到的场景。这次, 为了方便观察, 我们在提示消息前面都打印了一个 \t :

```
_ = interval.subscribe(onNext: {
    print("\tSubscriber 1: \($0)") })

delay(2) {
    _ = interval.subscribe(onNext: {
        print("\tSubscriber 2: \($0)") })
}

delay(4) {
    _ = interval.subscribe(onNext: {
        print("\tSubscriber 3: \($0)") })
}
```

第五, 在第一个订阅后面, 我们调用 connect 方法启动 interval :

```
_ = interval.subscribe(onNext: {
    print("\tSubscriber 1: \($0)") })

_ = interval.connect()

delay(2) {
    _ = interval.subscribe(onNext: {
        print("\tSubscriber 2: \($0)") })
}
```

执行一下, 就能在控制台看到类似这样的结果了:

```
Supervisor: event 0 # START
Subscriber 1: 0
Supervisor: event 1 # Second 1
Subscriber 1: 1
Supervisor: event 2 # Second 2
Subscriber 1: 2
Subscriber 2: 2
Supervisor: event 3 # Second 3
Subscriber 1: 3
Subscriber 2: 3
Supervisor: event 4 # Second 4
Subscriber 1: 4
Subscriber 2: 4
Subscriber 3: 4
...
```

## What's next?

看到这, 你应该对connectable operator有一个比较清楚的了解了。这一节中我们提到的两个 operators: publish 和 multicast 有个特点, 就是对订阅者而言, 都只能获得从订阅开始之后的事件。下一节, 我们来看另外一组connectable operators, 它们可以给订阅者“回放”历史事件。



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@boxue.io](mailto:10@boxue.io) (<mailto:10@boxue.io>)

QQ: 2085489246

2017 © Boxue, All Rights Reserved. 京ICP备15057653号-1 (<http://www.miibeian.gov.cn/>) 京公网安备 11010802020752号 (<http://www.beian.gov.cn/portal/registerSystemInfo?recordcode=11010802020752>)

友情链接 [SwiftV](http://www.swiftv.cn/) (<http://www.swiftv.cn/>) | [Seay信息安全博客](http://www.cnseay.com/) (<http://www.cnseay.com/>) | [Swift.gg](http://swift.gg/) (<http://swift.gg/>) | [Laravist](http://laravist.com/) (<http://laravist.com/>) | [SegmentFault](https://segmentfault.com/) (<https://segmentfault.com/>) | [戴青K的博客](http://blog.dianqk.org/) (<http://blog.dianqk.org/>)