理解create和debug operator

RxSwift - step by step

← 返回视频列表 | 预计阅读时间: *13分钟* | ‹ PREVIOUS | NEXT ›

有时候，Observable中的事件值并不像整数或字符串这么简单，当我们需要精确控制发送给订阅者的成功、错误和结束事件时，就可以使用RxSwift提供的`create` operator。

创建自定义事件的序列

在Observable+Creation.swift里，可以看到`create`的签名是这样的：

```
public static func create(
    _ subscribe: @escaping (AnyObserver<E>) -> Disposable
) -> Observable<E>
```

单看这堆复杂又相似的名字，就会觉得这个函数不太好理解，`create`究竟是如何用自定义的方式创建Observable的呢？理解这个问题，我们得从它的参数入手。

这里，`subscribe`并不是指事件真正的订阅者，而是用来定义当有人订阅Observable中的事件时，应该如何向订阅者发送不同情况的事件，理解这个问题，是使用`create`的关键。

于是，`create`调用大体的结构，就是这样的：

```
let customOb = Observable<Int>.create {
    // Hanle next, error, completed event here
}
```

然后，再来看`subscribe`自身，当然，它是一个closure，此时，`AnyObserver<E>`在这个closure里，表示任意一个订阅的“替身”，我们要用这个“替身”来表达向订阅者发送各种事件的行为。理解了这个概念，我们的`create`调用就可以进一步细化成这样：

```
let customOb = Observable<Int>.create { observer in
    // next event
    observer.onNext(10)

    observer.onNext(11)

    // complete event
    observer.onCompleted()
}
```

表示，只要有人订阅了costomOb中的事件，我们就先向订阅者发送两次.next事件，值分别是10和11，然后，发送.completed表示Observable结束。

最后，subscribe还要返回一个Disposable对象，我们可以用这个对象取消订阅进而回收customOb占用的资源：

```
let customOb = Observable<Int>.create { observer in
    // The same as above...

    return Disposables.create()
}
```

这样，我们就自定义了一个Observable，它会向每个订阅者发送10和11，然后结束。我们可以用下面的代码来试一下：

```
let disposeBag = DisposeBag()

customOb.subscribe(
    onNext: { print($0) },
    onCompleted: { print("Completed") },
    onDisposed: { print("Game over") }
).addDisposableTo(disposeBag)

// 10
// 11
// Completed
// Game over
```

就可以在控制台看到上面注释中的结果了。至此，我们就完成90%的工作了，但你可能还记得，我们还没自定义发生错误时，向订阅者发送的内容。其实很简单，先定义一个表示具体错误的类型：

```
enum CustomError: Error {
    case somethingWrong
}
```

然后，在create的定义里，要发生错误的地方，直接通知订阅者就好了：

```
let customOb = Observable<Int>.create { observer in
    // next event
    observer.onNext(10)

    observer.onError(CustomError.somethingWrong)

    observer.onNext(11)

    // complete event
    observer.onCompleted()

    return Disposables.create()
}
```

最后，在订阅的时候，我们可以直接通过onError来得到错误通知：

```
customOb.subscribe(
    onNext: { print($0) },
    onError: { print($0) },
    onCompleted: { print("Completed") },
    onDisposed: { print("Game over") }
).addDisposableTo(disposeBag)

// 10
// somethingWrong
// Game over
```

重新执行之前的订阅，我们就只能在结果中看到上面注释中的结果了。

事件序列的调试

在实际的编程中，有时我们会串联多个operator对事件序列进行处理，虽然这样写起来很方便，但发生问题调试时就很麻烦了，因为紧密串联在一起的代码让我们很难方便的洞察每一个环节的状态。为此，RxSwift提供了一个类似“旁路”功能的operator：do。它的用法和subscribe类似，只不过事件会“穿过”do，继续发给后续的环节。这样，如果我们怀疑某个串联的环节发生了问题，就可以插入一个do operator进行观察：

```
customOb.do(
    onNext: { print("Intercepted: \($0)") },
    onError: { print("Intercepted: \($0)") },
    onCompleted: { print("Intercepted: Completed") },
    onDispose: { print("Intercepted: Game over") }
)
.subscribe(
    onNext: { print($0) },
    onError: { print($0) },
    onCompleted: { print("Completed") },
    onDisposed: { print("Game over") }
).addDisposableTo(disposeBag)
```

这样，customOb中的事件就会“流经”do之后，继续发送给subscribe，方便我们观察订阅到的每一个内容。重新执行一下，就能看到下面这样的结果：

```
// Intercepted: 10
// 10
// Intercepted: somethingWrong
// somethingWrong
// Game over
// Intercepted: Game over
```

此时，如果你足够眼尖，可能已经发现上面例子中的一个小细节了。在do里，用于处理取消订阅事件的参数是onDispose，但subscribe中对应的则是onDisposed，甚至，这个微小的差异还影响了最终打印的结果。

看到这里，你可能会想，用do进行调试并不方便，毕竟还要写一堆的on，再配上各自的closure，应该有一个专门可以穿插在各种operator之间进行调试的operator。实际上，do也的确不是为了调试而生的，我们只是借用了它的“旁路”特性而已。RxSwift提供了一个调试专属的operator，叫做debug，它可以安插在任意一个需要确认事件值的地方，像这样：

```
customOb.debug()
.subscribe(
    onNext: { print($0) },
    onError: { print($0) },
    onCompleted: { print("Completed") },
    onDisposed: { print("Game over") }
).addDisposableTo(disposeBag)
```

在上面的例子里，我们把debug安插在了订阅前取代了do operator。这样，debug就会在不影响subscribe的同时，自动打印customOb发出的所有事件。执行一下，就可以得到类似下面这样的结果：

```
2017-04-06 18:56:25.348: main.swift:23 (RxSwiftInSPM) -> subscribed
2017-04-06 18:56:25.355: main.swift:23 (RxSwiftInSPM) -> Event next(10)
10
2017-04-06 18:56:25.356: main.swift:23 (RxSwiftInSPM) -> Event error(somethingWrong)
somethingWrong
Game over
2017-04-06 18:56:25.356: main.swift:23 (RxSwiftInSPM) -> isDisposed
```

可以看到，带有时间和源代码信息的行，就是debug operator提供的详细信息，包括了从订阅开始，收到.next，收到.error一直到最后.disposed的全过程，这样，调试起来，就方便多了。

What's next?

至此，可以说，你已经步入RxSwift的大门了。但你可能也还有一个感觉：尽管一直以来我们在不断强调着Observable表达的异步概念，但使用的例子总让我们没有异步操作的感觉。例如，一连串随机点击的按钮事件、或者处理一个网络请求才是异步的，像这样提前设计好一个Observable中的所有值，然后发送出来怎么能叫异步的呢?

实际上，这些带给我们异步感觉的操作，都需要在运行时动态给Observable添加内容。这也就意味着，这种Observable要有双重身份：一方面，它自身得是一个订阅者以获得系统事件的通知；另一方面，它也得是一个Observable，供我们编写的客户端代码进行订阅。在Rx的世界里，这样具有双重身份的对象，有一个专属的名字，叫做**Subject**。因此，在

用更真实的方式学习RxSwift之前，我们还需要额外做些工作。

‹ Prev: RxSwift中的那些“术语”到底在说什么?

理解create和debug operator

Next: 四种Subject的基本用法 ›