RxSwift - step by step

App demo II 使用map/flatMap简化代码

(https://www.boxueio.com/series/rxswift-101/ebook/270)

如何合并Observables中的事件

(https://www.boxueio.com/series/rxswift-101/ebook/272)

如何合并Observables

Back to series (/series/rxswift-101)

字号

字号

默认主题

金色主题

暗色主题

在前面和operators相关的内容里，filter operators也好，transform operators也好，我们介绍的内容都是和单个Observable相关的。但就如同这个纷繁复杂的世界一样，很多时候，我们需要把多个事件序列合并起来表达某个现实中的情况。为此，RxSwift提供了另外一大类operators完成这个工作，它们叫做combine operators。

处理事件的前置条件

我们要介绍的第一个场景，是为特定的事件序列，添加前置条件。例如，在处理网络请求之前，获得当前网络连接的情况。

首先，定义一个 enum 表示不同的网络连接情况：

```
enum Condition: String {
    case cellular = "Cellular"
    case wifi = "WiFi"
    case none = "none"
}
```

其次，自定义一个Observable，表示网络请求。这里我们只是直接返回一个 `String` 和completed事件表示服务器的成功返回：

```
example(of: "startWith") {
    let bag = DisposeBag()
    let request = Observable<String>.create { observer in

        observer.on(.next("Response from server."))
        observer.onCompleted()

        return Disposables.create()
    }
}
```

最后，在每次订阅这个“服务器返回值”的时候，我们就可以通过 `startWith` operator先得到当前的网络连接情况：

```
example(of: "startWith") {
    /// ...

    request.startWith(Condition.wifi.rawValue)
            .subscribe(onNext: { dump($0) })
            .addDisposableTo(bag)
}
```

这里，我们只是简单把订阅到的事件 dump 了出来，执行一下，就可以在控制台看到两个事件：第一个事件值是*WiFi*，也就是 `startWith` 发生的事件；第二个事件值是*Response from server.*，也就是服务器的返回结果。

有了这个思路之后，我们就可以在订阅的代码里，根据当前网络条件，以及服务器的返回值，做各种后续操作了。唯一需要注意的是 **`startWith` 中的事件值的类型，和它后续的事件值类型，必须是相同的**。在我们的例子例，也就是 `String` ，否则，会导致编译错误。

串行合并多个事件序列

除了把一个单一事件值和Observable串联，我们也可以把两个并行的Observable合并起来串行处理。直接来看个例子。首先，我们定义连个并行的事件序列：

```
let queueA = PublishSubject<String>()
let queueB = PublishSubject<String>()
```

其次，为了实现**先处理完 queueA 中的事件之后，再开始处理 queueB 中的事件**，我们可以用 concat operator把这两个序列串联起来：

```
let sequence = Observable
    .concat([queueA.asObservable(), queueB.asObservable()])
```

可以看到，这个全局的 concat operator接受一个 Array 参数，允许我们把要依次处理的Observable放在数组里传递给他。

第三，为了观察 concat 的效果，我们先用下面的代码试一下：

```
sequence.subscribe(onNext: {
    dump($0)
}).addDisposableTo(bag)

queueA.onNext("A1")
queueA.onNext("A2")
queueA.onNext("A3")
queueB.onNext("B1")

// We can only subscribe A1, A2, A3
```

其中， queueA 中发生了三次事件， queueB 中发生了一次。这时，我们之前的代码会订阅到什么呢？执行一下就会发现，只能订阅到“A1, A2, A3”。

这就是我们一开始说到的先处理完 queueA 中的事件之后，再开始处理 queueB 中的事件的含义。只要 queueA 没有结束，我们就无法在合并的结果中订阅到 queueB 的内容。这就是串行处理Observables的效果。为了订阅到 queueB 中的事件，我们得这样：

```
queueA.onNext("A1")
queueA.onNext("A2")
queueA.onNext("A3")
queueA.onCompleted()
queueB.onNext("B1")
```

重新执行下，就可以在控制台看到“B1”了。初次之外，除了接受数组的全局 concat operator，Observable自身也有一个 concat 方法，允许我们合并两个Observables：

```
let sequence = queueA.concat(queueB.asObservable())
```

当然，结果和之前是相同的。

第四，我们来看 concat 之后的Observable的 completed 和 disposed 事件。为此，我们把之前的订阅代码改成这样：

```
sequence.subscribe(onNext: {
        dump($0)
    }, onCompleted: {
        print("Completed")
    }, onDisposed:  {
        print("Disposed")
    }).addDisposableTo(bag)
```

还是之前的事件序列，执行一下就会看到，这次我们能订阅到*A1 -> A2 -> A3 -> B1 -> Disposed*事件。为什么没有看到*Completed*呢？这是因为，**只有 concat 中所有Sub observables都完成时，合成后的Observable才会完成**。因此，把之前的事件序列改成下面这样：

```
queueA.onNext("A1")
queueA.onNext("A2")
queueA.onNext("A3")
queueA.onCompleted()
queueB.onNext("B1")
queueB.onCompleted()
```

再执行一下，就会在*Disposed*事件之前，看到*Completed*事件了。这里，还要强调的一点是，无论是哪种形式的 concat operator，都只能合并事件值类型相同的Observables。否则，就会导致编译错误。

理解合成后Observable的生命周期

接下来，我们来了解合成后的Observable的“生死存亡”。如果所有Sub-observable都正常结束了，合成后的Observable就正常结束了。接下来，如果发生错误，存在着两种不同情况，为了方便观察，我们把订阅代码改成这样：

```
sequence.subscribe(onNext: {
        dump($0)
    }, onError: {
        print($0)
    }, onCompleted: {
        print("Completed")
    },onDisposed:  {
        print("Disposed")
    }).addDisposableTo(bag)
```

第一种情况，如果当前正在订阅的Sub-observable发生错误，合成后的Observable也会发生*Error*事件并结束，后续的Sub-observable中的事件都不会订阅到了：

```
enum E: Error {
    case demo
}

queueA.onNext("A1")
queueA.onNext("A2")
queueA.onError(E.demo)
queueB.onNext("B1")
```

在上面这段代码里，在订阅到A2之后， queueA 发生了错误，这时合并后的序列就结束了，于是我们只会订阅到*A1 -> A2 -> demo -> Disposed*。

第二种情况，如果在“排队中”的Sub-observable发生错误，这种情况其实和订阅中的Sub-observable中发生*Next*事件是没什么区别的，因为只要当前订阅的序列不结束，排队中的Sub-observable中的事件不会对合成的Observable有任何影响：

```
queueA.onNext("A1")
queueA.onNext("A2")
queueB.onNext("B1")
queueB.onError(E.demo)
queueA.onNext("A3")
```

这次，我们会订阅到*A1 -> A2 -> A3 -> Disposed*。因为，直到Observable离开作用域被回收的时候， queueA 也没有发生*Completed*或*Error*事件，因此 queueB 中的任何事件都不会对合成的Observable有影响。

并行合并多个事件序列

看过了刚才的例子，你可能会想，我们可以“并行”合并所有的Observable么？这样，只要合并进来的Observable中有事件发生，我们就可以订阅到，而无需等待前置的Observable结束。为此，我们需要使用 `merge` operator。

首先，把之前合并Observable的代码改成这样：

```
let sequence = Observable.merge(
    [queueA.asObservable(), queueB.asObservable()])
```

其次，把 queueA 和 queueB 的事件序列改成这样：

```
queueA.onNext("A1")
queueA.onNext("A2")
queueB.onNext("B1")
queueA.onNext("A3")
queueA.onCompleted()
queueB.onCompleted()
```

执行一下，我们就会单纯的按照事件发生的顺序，依次订阅到*A1 -> A2 -> B1 -> A3 -> Completed -> Disposed*事件了。

另外一点和 `concat` 不同的是，只要 `merge` 中的任何一个Sub-observable发生*Error*事件，合并后的Observable就会结束掉，大家可以自己试一下。

控制最大订阅数量

了解了 `merge` 的特性之后，我们来看如何控制合并的过程中同时订阅的Sub-observable数量，默认条件下， `merge` 当然就是同时订阅合并进来的所有Sub-observable。

例如，我们定义一个新的Observable，然后 merge 它：

```
let sequence1 = Observable.of(
    queueA.asObservable(),
    queueB.asObservable())
    .merge()

queueA.onNext("A1")
queueA.onNext("A2")
queueB.onNext("B1")
```

这时，订阅到的，就会是 queueA 和 queueB 中发生的所有事件，也就是*A1 -> A2 -> B1 -> Disposable*。

如果我们想只同时订阅到一个事件，就可以这样：

```
let sequence1 = Observable.of(
    queueA.asObservable(),
    queueB.asObservable())
    .merge(maxConcurrent: 1)
```

还是之前的事件序列，这次，我们就只能订阅到*A1 -> A2 -> Disposable*，因为 maxConcurrent 指定了只能“同时”订阅1个队列。如果我们在B1发生前，让 queueA 结束，就可以订阅到 queueB 中的事件了：

```
queueA.onNext("A1")
queueA.onNext("A2")
queueA.onCompleted()
queueB.onNext("B1")
```

> 理解了这个过程之后，我们可以再创建一个 queueC ，然后把并发订阅控制到2，就能进一步理解它的用法了。

What's next?

以上，就是这一节的内容，在了解了如何合并两个Observable之后，下一节，我们来看如何合并两个Observable中的事件。

◀ App demo II 使用map/flatMap简化代码 (https://www.boxueio.com/series/rxswift-101/ebook/270)

如何合并Observables中的事件 ▶ (https://www.boxueio.com/series/rxswift-101/ebook/272)

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)

版权声明 (HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT)

联系泊学

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

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