RxSwift - step by step

App demo I 一个Alamofire router的实现 (https://www.boxueio.com/series/rxswift-101/ebook/269)

如何合并Observables (https://www.boxueio.com/series/rxswift-101/ebook/271)

App demo II 使用map/flatMap简化代码

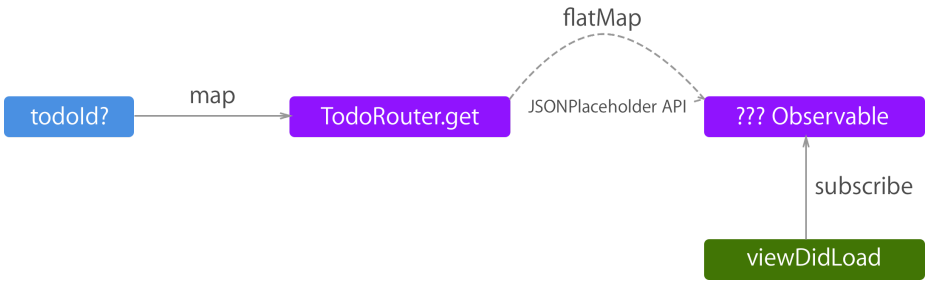
Back to series (/series/rxswift-101)

字号 字号 默认主题 金色主题 暗色主题

了解刚才实现的Todo Demo之后，这一节，我们通过*Transform operators*改进 `viewDidLoad` 的实现方式。

访问源代码 (https://github.com/puretears/my-todo-rx-demo/tree/master/Finished)

整体的实现思路，是这样的：



- 首先，由于 `GET /todos` 方法可以接受一个参数，因此，我们对一个 `Int?` 使用 `map` ，把它变成一个 `Observable<TudoRouter>` ；
- 其次，我们要把 `Observable<TudoRouter>` 变成某种表示网络请求结果的 `Observable` ；
- 第三，在 `viewDidLoad` 方法里，我们直接订阅上一步得到的结果，然后根据订阅到的事件更新UI就好了；

这样， `viewDidLoad` 方法里，就不会再有包含网络请求细节的代码了，而只体现了为了展示UI而执行的逻辑。有了这个思路之后，我们把 `viewDidLoad` 之前的代码删掉，来实现它。

第一步要实现的内容很简单，直接在 `viewDidLoad` 方法里，添加下面的代码：

```
override func viewDidLoad() {
    super.viewDidLoad()

    let todoId: Int? = nil
    Observable.just(todoId)
        .map { tid in
            return TodoRouter.get(tid)
        }
}
```

第二步，为了把网络请求的结果变成一个Observable，我们只能自己用 `create` operator定制一个。为此，我们添加了一个*Todo+Alamofire.swift*的文件。在这里，给 Todo 添加一个 `extension` 。这个extension中只有一个方法，它接受 TodoRouter 为参数，并返回 `Observable<[[String: Any]]>` ：

```
extension Todo {
    class func getList(from router: TodoRouter)
        -> Observable<[[String: Any]]> {

    }
}
```

在它的实现里，我们直接使用 `create` ，大体的逻辑，和之前我们写在 `viewDidLoad` 方法里的代码是相同的：

```
class func getList(from router: TodoRouter)
    -> Observable<[[String: Any]]> {
    return Observable.create {
        (observer) -> Disposable in
        let request = Alamofire.request(router)
            .responseJSON { response in
                guard response.result.error == nil else {
                    observer.on(
                        .error(response.result.error!))
                    return
                }

                guard let todos =
                    response.result.value as? [[String: Any]] else {
                    observer.on(
                        .error(GetTodoListError.cannotConvertServerRespons
e))
                    return
                }

                observer.on(.next(todos))
                observer.onCompleted()
            }

        return Disposables.create {
            request.cancel()
        }
    }
}
```

可以看到，同样，我们给 `request` 传递了一个 `TodoRouter` 对象，然后在 `responseJSON` 里处理了各种情况。不同的是，这次，我们通过 `observer.on()` 像订阅者发送了对应的错误和成功的事件，而没有在这里直接处理业务逻辑。最后，当创建的Observable被回收的时候，我们就取消网络请求。

第三步，有了这个自建的Observable，我们就可以继续编写之前 `viewDidLoad` 中的代码了，先来看Observable变换的部分：

```
override func viewDidLoad() {
    super.viewDidLoad()

    let todoId: Int? = nil
    Observable.of(todoId)
        .map { tid in
            return TodoRouter.get(tid)
        }
        .flatMap { route in
            return Todo.getList(from: route)
        }

    /// ...
}
```

这就是我们一开始在图中，展示的虚线的部分。在调用 `flatMap` 前，序列类型是 `Observable<TodoRouter>` ，之前我们说过， `flatMap` 会把原序列中的每一个事件，变成一个新的Observable。于是，在变换后，我们就可以直接订阅网络请求返回的结果了：

> 如果这里我们使用 `map` 而不是 `flatMap` ，就会变换出一个 `Observable<Observable<[[String: Any]]>>` 类型，而这就是flat的含义。

```
override func viewDidLoad() {
    super.viewDidLoad()

    let todoId: Int? = nil
    Observable.of(todoId)
        .map { tid in
            return TodoRouter.get(tid)
        }
        .flatMap { route in
            return Todo.getList(from: route)
        }
        .subscribe(onNext: { (todos: [[String: Any]]) in
            self.todoList = todos.flatMap { Todo(json: $0 ) }
            self.tableView.reloadData()
        }, onError: { error in
            print(error.localizedDescription)
        })
        .addDisposableTo(bag)
}
```

在订阅的代码里，我们再次使用了 `flatMap` ，不过这次，就和RxSwift没什么关系了，由于 `Todo.init(json:)` 返回的是 `Todo?` ，我们使用Array的 `flatMap` 方法，去掉了数组中所有的 `nil` 。另外，由于订阅的代码是发生在主线程中的，因此，订阅的closure也会在主线程中执行，这样，我们也就无需再使用 `DispatchQueue` 了。

现在，这段代码看上去，“拿到数据，更新UI”的意味就更明确了。重新执行一下，结果和之前应该是一样的。

What's next?

以上，就是*Transform operators* demo的全部内容。实际上，我们所有的重点，都围绕着 `flatMap` 展开。它最主要的应用，就是在优化这类异步事件的处理上。理解了这一点，几乎就可以拿下*Transform operators*用法的大半江山了。

至此，我们已经介绍了两大类operator，它们分别是*Filter operators*和*Transform operators*。通过对这些概念和应用的理解，相信现在你应该对RxSwift越发找到感觉了。接下来，我们来看另外一类operators，它们用来组合不同的Observables，叫做*Combine operators*。

◀ App demo I 一个Alamofire router的实现
(https://www.boxueio.com/series/rxswift-101/ebook/269)

如何合并Observables ▶
(https://www.boxueio.com/series/rxswift-101/ebook/271)

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/)