Reactive Programming in Swift

RxSwift UI交互 - III
(https://www.boxueio.com/series/reactive-programming-in-swift/ebook/79)

RxDataSource创建UITableView - I
(https://www.boxueio.com/series/reactive-programming-in-swift/ebook/81)

基于RxSwift的网络编程 - I

Back to series (/series/reactive-programming-in-swift)

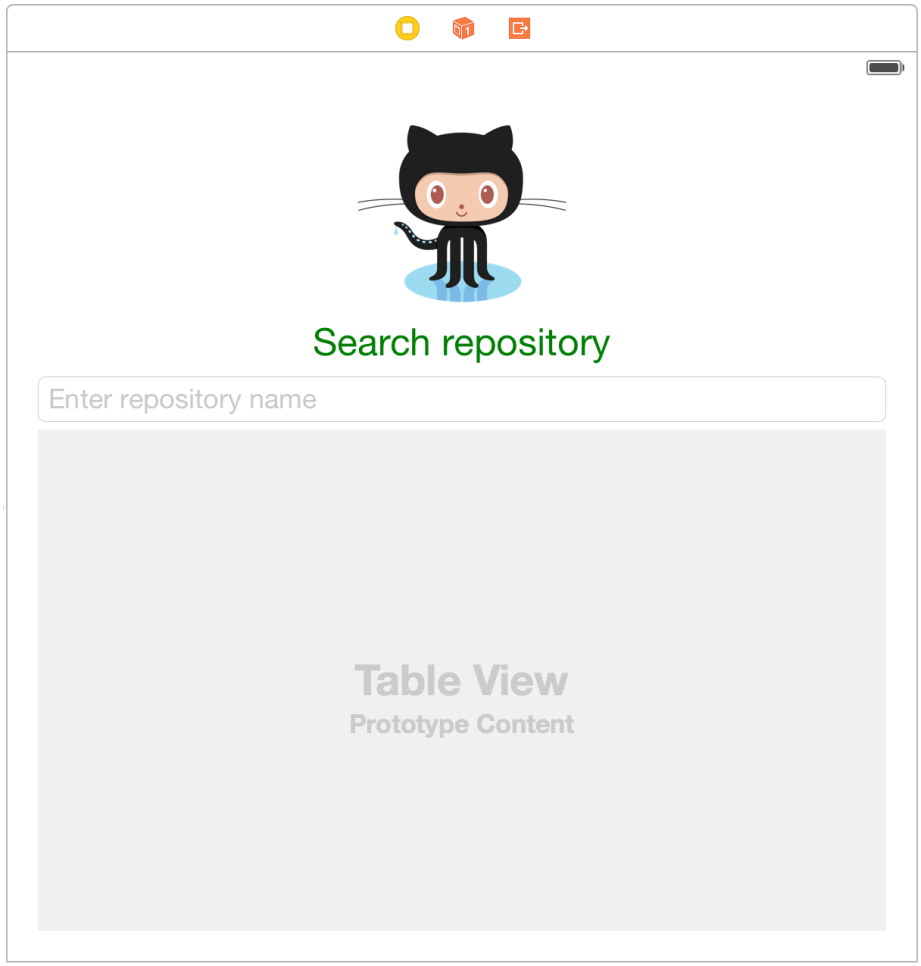
字号
字号
默认主题
金色主题
暗色主题

通过前面一系列视频，我们对RxSwift (https://github.com/ReactiveX/RxSwift)的各种用法已经有了一个比较具体的认识了。除了常见的UI交互之外，还有一大类异步事件是几乎我们一定要处理的，那就是网络编程。现如今，app通过HTTP API获取一段JSON信息，然后对UI做一些处理已经是非常平常的事情了。在这段视频的里，我们向大家介绍如何使用RxSwift (https://github.com/ReactiveX/RxSwift)简化网络编程的代码。

项目初始模版 (https://github.com/Boxue/episode-samples/tree/master/RxSwift/ReactiveNetwork-I/RxNetworkDemoStarter)

项目准备工作

我们的App会在Github上搜索特定名称的项目，在 `UITextField` 里输入项目名称，我们就自动在Github上搜索项目的名字，并在下面的 `UITableView` 中显示一些项目信息显示出来。



然后，在 `ViewController` 里，添加对应的IBOutlet：

```
@IBOutlet weak var repositoryName: UITextField!
@IBOutlet weak var searchResult: UITableView!
```

以及 `DisposeBag`：

```
var bag: DisposeBag! = DisposeBag()
```

最后，通过CocoaPods安装项目需要的组件：

```
# Uncomment this line to define a global platform for your project
platform :ios, '9.0'
# Uncomment this line if you're using Swift
use_frameworks!

target 'RxNetworkDemo' do
    pod 'Alamofire', '~> 3.4'
    pod 'RxSwift',    '~> 2.0'
    pod 'RxCocoa',    '~> 2.0'
    pod 'SwiftyJSON', :git => 'https://github.com/SwiftyJSON/SwiftyJSON.gi
t'
end
```

并且，在 ViewController 里，引入对应的组件：

```
import UIKit
import RxSwift
import RxCocoa
import Alamofire
import SwiftyJSON
```

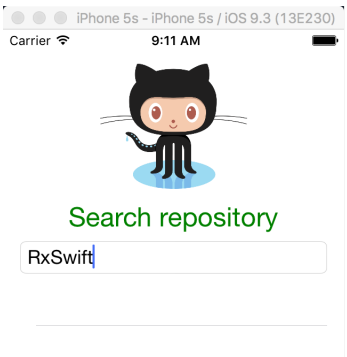
这样，我们就做好所有的准备工作了。

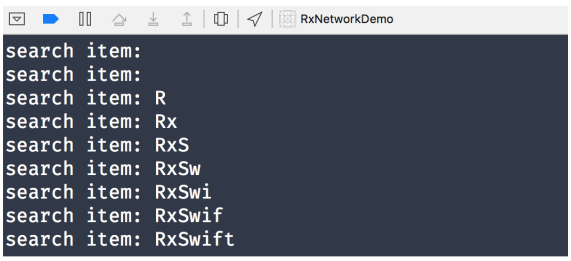
控制网络请求频度

发送请求之前，我们要先通过 UITextField 获取用户输入。很简单，直接订阅 UITextField 的 rx_text 就可以了，在 viewDidLoad 方法里，添加下面的代码：

```
self.repositoryName.rx_text
    .subscribeNext {
        print("Search item: \($0)")
    }.addDisposableTo(self.bag)
```

执行后，会发现，控制台里的结果是这样的：





第一次的空白字符串是UI加载的时候，监听到的事件值；第二次空白是 UITextField 获取输入事件的时候间听到的事件值；而后，我们每输入一个字符，就会监听到一个不同的事件。

如果我们用这样的结果来作为在Github上搜索的内容，会有一些问题：

- 我们用空的字符串进行了搜索，明显是错误的；
- 当输入只有1，2个字符时，发起的搜索明显是不精准的；
- 当输入的名称较长时，输入过程会发起大量无效的搜索（例如：仅仅是输入RxSwift，就发起了9次）；

首先，我们来解决前两个问题。

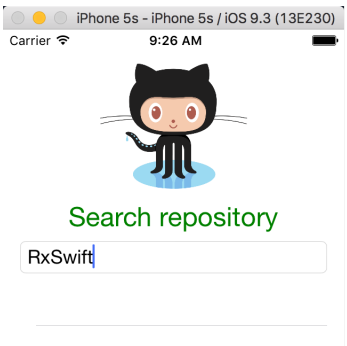
使用filter过滤事件内容

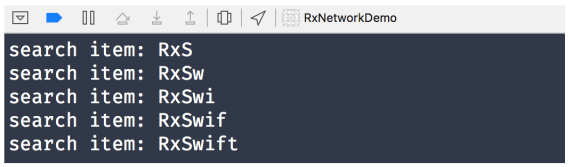
我们要先过滤掉过短的输入，例如，当用户输入2个以上的字符时才进行查询。很简单，在订阅前，使用 filter(n) 对事件值进行过滤就可以了：

```
self.repositoryName.rx_text
    .filter {
        return $0.characters.count > 2
    }
    .subscribeNext {
        print("search item: \($0)")
    }.addDisposableTo(self.bag)
```

`.filter` 的参数是事件值的类型，在我们的例子里，也就是 `String`，返回一个 `Bool`，表示是否要向订阅者发送事件。

重新运行，就会发现我们过滤掉了一些明显无效的输入：





尽管如此，我们还是订阅到了5次事件，如果每次订阅到都发起请求，还是太频繁了，我们希望进一步控制请求的频率。

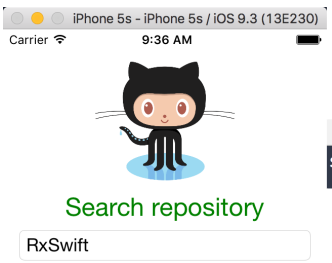
使用throttle控制请求频度

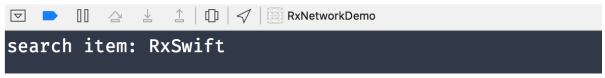
我们可以使用 `throttle` 在指定的时间间隔里，忽略掉发生的事件。这样，就不会每次输入都订阅到事件了。继续修改订阅代码：

```
self.repositoryName.rx_text
    .filter {
        return $0.characters.count > 2
    }
    .throttle(0.5, scheduler: MainScheduler.instance)
    .subscribeNext {
        print("search item: \($0)")
    }.addDisposableTo(self.bag)
```

`throttle` 的第一个参数表示希望忽略的时间间隔，第二个参数表示在主线程中运行计时器。

重新运行，这次，控制台里的结果基本就可用了：





接下来的思路就很简单了，我们直接在订阅到的事件里，调用Github API查询项目，并把查询结果更新到 `TableView` 里就好了。

思路虽然简单，却关联到了不少的实现细节，我们先来完成网络请求的部分。

包装Alamofire成Observable

我们先给 `ViewController` 添加一个 `extension`，所有和网络相关的代码，都放到这个 `extension` 里：

```
extension ViewController {
}
```

我们希望最终订阅到的事件值，是个包含我们需要内容的Key-Value集合，简单起见，我们添加一个类型的别名：

```
typealias RepositoryInfo = Dictionary<String, AnyObject>
```

在这个 Dictionary 中， String 用于索引结果中的内容，而值有可能是整数、有可能是字符串，因此我们定义成了 AnyObject 。

接下来，我们在 ViewController extension 中，添加一个方法：

```
private func searchForGithub(repositoryName: String)
    -> Observable<RepositoryInfo>
```

searchForGithub 接受一个表示，表示要查询的repository的名字，返回一个事件值类型是 RepositoryInfo 的事件序列。

怎么实现呢?

之前的视频里我们也提到过，RxSwift (https://github.com/ReactiveX/RxSwift)提供了一个叫做 create 的方法，可以让我们自定义事件序列。对于封装一个网络请求来说，它简直再合适不过了。

在 searchForGithub 里，添加下面的代码：

```
private func searchForGithub(repositoryName: String)
    -> Observable<RepositoryInfo> {

    return Observable.create {
        (observer: AnyObserver<RepositoryInfo>) -> Disposable in

        let url = "https://api.github.com/search/repositories"
            let parameters = [
                "q": repositoryName + " stars:>=2000"
            ]

        let request = Alamofire.request(.GET, url,
        parameters: parameters,
        encoding: .URLEncodedInURL)
    }
}
```

create 接受一个closure参数，这个closure参数本质上和我们之前用过的 subscribeNext 方法是类似的。它接受一个 AnyObserver ，并返回一个 Disposable 对象。

在这里， AnyObserver 表示要创建的事件序列的订阅者。稍候，我们要根据请求的不同结果，向这个订阅者发送事件。由于我们要返回的Observable的事件值类型是 RepositoryInfo ，因此，这里 AnyObserver 可以订阅到的事件值的类型，也是 RepositoryInfo 。

然后，在这个Clousre的实现里，我们先分别添加了请求的URL，以及附带的参数。其中：

- 参数q表示要查询的项目名；
- “ start:>=2000”是Github的项目查询语言，表示查询大于2000星的项目；

最后，直接用 Alamofire.request 请求了Github API。为了先了解下这个API的返回值，我们不妨先在浏览器里看一下调用结果：

```
https://api.github.com/search/repositories?q=RxSwift%20stars:>=2000
```

```
{
  "total_count": 1,
  "incomplete_results": false,
  "items": [
    {
      "id": 33569135,
      "name": "RxSwift",
      "full_name": "ReactiveX/RxSwift",
      "owner": {
        "login": "ReactiveX",
        "id": 6407041,
        "avatar_url": "https://avatars.githubusercontent.com/u/6407041?v=3",
        "gravatar_id": "",
        "url": "https://api.github.com/users/ReactiveX",
        "html_url": "https://github.com/ReactiveX",
        "followers_url": "https://api.github.com/users/ReactiveX/followers",
        "following_url": "https://api.github.com/users/ReactiveX/following{/other_user}",
        "gists_url": "https://api.github.com/users/ReactiveX/gists{/gist_id}",
        "starred_url": "https://api.github.com/users/ReactiveX/starred{/owner}{/repo}",
        "subscriptions_url": "https://api.github.com/users/ReactiveX/subscriptions",
        "organizations_url": "https://api.github.com/users/ReactiveX/orgs",
        "repos_url": "https://api.github.com/users/ReactiveX/repos",
        "events_url": "https://api.github.com/users/ReactiveX/events{/privacy}",
        "received_events_url": "https://api.github.com/users/ReactiveX/received_events",
        "type": "Organization",
        "site_admin": false
      },
      "private": false,
      "html_url": "https://github.com/ReactiveX/RxSwift",
      "description": "Reactive Programming in Swift",
      "fork": false,
      "url": "https://api.github.com/repos/ReactiveX/RxSwift",
      "forks_url": "https://api.github.com/repos/ReactiveX/RxSwift/forks",
      "keys_url": "https://api.github.com/repos/ReactiveX/RxSwift/keys{/key_id}",
      "collaborators_url": "https://api.github.com/repos/ReactiveX/RxSwift/collaborators{/
      "teams_url": "https://api.github.com/repos/ReactiveX/RxSwift/teams",
```


在返回的JSON里，大致分成几大部分：

- **total_count**：表示查询到的repository个数；
- **incomplete_results**：表示是否返回的是部分结果；
- **items**是一个JSON对象数组，包含了每一个查询到的repository的详细信息；

稍候，我们就会接收这个结果集，把他筛选成下面这样：

```
{
    "total_count": 1,
    "items": [
        {
            "full_name": "RxSwift",
            "description": "ReactiveX/RxSwift"
            "html_url": "Reactive Programming in Swift",
            "avatar_url": "https://avatars.githubusercontent.com/u/640704
1?v=3"
        }
     ]
}
```

然后，返回给事件的订阅者。至此，这一切都还不太难理解。接下来，重头戏就来了。

自定义向订阅者发送的事件

接下来我们要进行的工作，是使用 `create` 方法自定义Observable的重点，我们需要根据Github的返回值，来定义向订阅者返回的内容。

把 `Alamofire.request` 部分的代码，添加上结果处理：

```
let request = Alamofire.request(.GET, url,
    parameters: parameters,
    encoding: .URLEncodedInURL)
    .responseJSON { response in

        switch response.result {
        case .Success(let json):
        // How can we handle success event?
        case .Failure(let error):
            observer.on(.Error(error))
        }
    }
```

当请求失败的时候，我们的处理逻辑很简单：

1. 直接把返回的 `NSError` 对象封装在 `Event.Error` 里；
2. 通过 `on` 方法把事件发送给订阅者；

那成功的时候呢？发送事件的部分，当然也是如法炮制，用 on 方法就好了，我们发送些什么呢？

1. 我们首先要把返回的结果做一些筛选，只找出我们需要使用的数据；
2. 当请求成功时，我们要**先发送.Next事件，传递事件值，然后发送.Completed事件，表示结束**；

使用SwiftyJSON过滤返回结果

首先来实现第一步，对返回结果进行筛选，在 `ViewController extension` 中，添加一个新的方法：

```
private func parseGithubResponse(
    response: AnyObject) -> RepositoryInfo
```

它接受一个 `AnyObject` 作为参数，我们会传递请求成功时 `.Success` 的associated value，然后返回要发送给订阅者的 `RepositoryInfo` 。

在 `parseGithubResponse` 的实现里，我们使用SwiftyJSON (https://github.com/SwiftyJSON/SwiftyJSON)来简化JSON串的处理：

```
private func parseGithubResponse(
    response: AnyObject) -> RepositoryInfo {

    let json = JSON(response);
    let totalCount = json["total_count"].int!

    var ret: RepositoryInfo = [
        "total_count": totalCount,
        "items": []
    ];
}
```

在上面的代码里：

1. `JSON(response)` 用于初始化SwiftyJSON (https://github.com/SwiftyJSON/SwiftyJSON)，我们可以得到一个Swity.JSON对象 `json` ；
2. 然后，就可以像访问普通 `Dictionary` 一样去访问JSON串中的内容了，例如：`json["total_count"]` 。如果我们确信它是个整数，就直接访问它的 `int` 属性，读取optional的值就可以了；
3. 我们构建了一个最基本的返回值 `ret` ，初始化了 `total_count` ；

查询到了repository的个数之后，我们来处理返回结果中的“items”部分，它是一个JSON数组，数组中的每一个对象，都表示一个repository。同样，SwiftyJSON (https://github.com/SwiftyJSON/SwiftyJSON)也有方便我们处理数组的方法。在 `ret` 的定义后面，继续添加下面的代码：

```
if totalCount != 0 {
    let items = json["items"]
    var info: [RepositoryInfo] = []

    for (_, subJson):(String, JSON) in items {
        let fullName = subJson["full_name"].string!
        let description = subJson["description"].string!
        let htmlUrl = subJson["html_url"].string!
        let avatarUrl = subJson["owner"]["avatar_url"].string!

        info.append([
            "full_name": fullName,
            "description": description,
            "html_url": htmlUrl,
        "avatar_url": avatarUrl
        ])
    }

    ret["items"] = info
}
```

在上面的代码里，当查询到的repository不为0时：

首先，我们使用 `json["items"]` 读取到了JSON数组，它仍旧是一个 `Swity.JSON` 对象；

其次，我们定义了一个存储items信息的 `RepositoryInfo` 数组，用于保存筛选过的内容；

第三，尽管 `items` 是一个 `Swifty.JSON` 对象，我们仍旧可以使用 `for...in` 循环来遍历它。对于 `items` 中的每一个key-value，我们可以把它理解为是一个 `(String, JSON)` 类型的Tuple，于是，我们用这样的代码：

```
let fullName = subJson["full_name"].string!
let description = subJson["description"].string!
let htmlUrl = subJson["html_url"].string!
let avatarUrl = subJson["owner"]["avatar_url"].string!
```

分别读取了每一个项目的名称、描述、网址以及创始人头像。值得说明的是，当读取创始人头像时，由于owner索引的内容又是一个JSON对象，因此，我们可以使用串联索引的方式把嵌套的JSON串中的内容读取出来，很方便。

筛选出了所有需要的信息之后，我们就把内容添加到用于保存筛选结果的数组里。最后全部筛选结束之后，我们就把 `info` 更新到返回值的“items”字段里。

最后，别忘记让 `parseGithubResponse` 返回 `ret`：

```
return ret
```

这样我们就完成对结果的筛选了，最终我们得到了一个只包含我们感兴趣的 `RepositoryInfo` 对象。这时，我们回到主战场，处理Alamofire请求成功时的事件处理。

封装.Next()事件

在之前.Success的case里，添加下面的代码：

```
let request = Alamofire.request(.GET, url,
    parameters: parameters,
    encoding: .URLEncodedInURL)
    .responseJSON { response in

        switch response.result {
        case .Success(let json):
        // How can we handle success event?
            let info = self.parseGithubResponse(json)

            observer.on(.Next(info))
            observer.on(.Completed)
        case .Failure(let error):
            observer.on(.Error(error))
        }
    }
```

其实很简单，我们只要把 `parseGithubResponse` 的返回值，直接作为.Next的associated value就可以了。这里，再次提醒大家，不要忘记在 `.Next` 之后发送 `.Completed` 。

处理Observable.create参数的返回值

至此，我们已经完成了90%的工作，但是，现在还不是休息的时候。如果你记不清了，可以翻回头看看 `Observable.create` 的参数定义，它接受的closure参数还要返回一个 `Disposable` 对象呢。这个对象，用于对 `create` 返回的Observable进行“善后工作”。

在处理网络请求的时候，无论因为任何原因， `create` 创建的事件序列被销毁了，那么我们最好取消掉正在执行的网络请求。因此，我们要添加一个 `AnonymousDisposable` 对象，他唯一的工作，就是取消网络请求。在 `create` 的Closure方法最后，添加下面的代码：

```
return AnonymousDisposable {
    request.cancel()
}
```

如果 `request` 已经完成了，调用 `cancel()` 也不会带来任何问题。

如果我们创建的事件序列在被销毁时无需执行任何额外操作，我们也可以直接使用 `return NopDisposable` 返回一个“什么也不需要做的 `Disposable` 对象”。

这样，使用 `create` 封装网络请求的功能就全部完成了。我们把每一次网络请求，都封装成了一个可以被订阅的事件序列。

接下来，我们实现在 `UITextField` 中输入后，自动查询的功能。别急，看似简单的事情，仍旧有新要点要注意。

使用.flatMap转化Observable

基本思路是很简单的，把**要发送给订阅者**的每一次 UITextField 输入事件，在 map() 里调用 searchForGithub 方法，变成Github的查询结果就好了。按照想象的在订阅前添加下面的代码：

```
self.repositoryName.rx_text
    .filter {
        return $0.characters.count > 2
    }
    .throttle(0.5, scheduler: MainScheduler.instance)
    .map {
        self.searchForGithub($0)
    }
```

它可以正常工作，但是，执行的方式一定和我们想象中有点儿差别。我们希望 subscribeNext 可以订阅到一个事件值是 RepositoryInfo 的事件。

但是，由于 self.searchForGithub($0) 返回的是一个 Observable<RepositoryInfo> ，因此，我们订阅到的实际上是一个事件序列。我们还需要在 .subscribeNext 里继续订阅它，这显然不是我们想要的。

为了解决这样的问题，RxSwift (https://github.com/ReactiveX/RxSwift)提供了另外一个映射事件序列的方法 .flatMap ，在它的实现里，我们可以找到这样的注释：

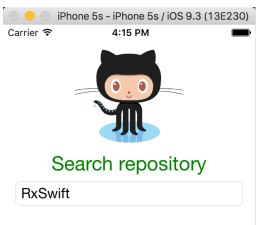
> Projects each element of an observable sequence to an observable sequence and merges the resulting observable sequences into one observable sequence.

简单来说，就是如果经过映射后的结果是一个新的事件序列，那么 flatMap 把**映射前的事件（在我们的例子里是 UITextField 的输入）**和**映射后的事件（在我们的例子里是一个网络请求）**合并成一个事件发送给订阅者。

这样，我们就可以直接在 subscribeNext 中订阅到 RepositoryInfo 了。我们可以把各种订阅到的值，输出到控制台上。

```
self.repositoryName.rx_text
    .filter {
        return $0.characters.count > 2
    }
    .throttle(0.5, scheduler: MainScheduler.instance)
    .flatMap {
        self.searchForGithub($0)
    }
.subscribeNext {
    let repoCount = $0["total_count"] as! Int;
    let repoItems = $0["items"] as! [RepositoryInfo];

    if repoCount != 0 {
        print("item count: \(repoCount)")

        for item in repoItems {
            print("---------------------------------")

            let name = item["full_name"]
            let description = item["description"]
            let avatarUrl = item["avatar_url"]


            print("full name: \(name)")
            print("description: \(description)")
            print("avatar_url: \(avatarUrl)")
        }
    }
}.addDisposableTo(self.bag)
```

这样，我们就实现实时响应查询的功能了，编译执行，我们就可以在控制台看到结果了：



Next?

终于到了一个可以稍事休息的阶段了。我们通过这段视频，实现了这个App的前半部分：

- 如何控制用户输入频度；
- 如何使用 `create` 封装网络请求；
- 如何使用SwiftyJSON (https://github.com/SwiftyJSON/SwiftyJSON)处理JSON结果；
- 如何使用 `flatMap` “合并”事件序列；

在下一段视频中，我们将完成这个App的后半部分，使用RxSwift (https://github.com/ReactiveX/RxSwift)，处理UITableView。

RxSwift UI交互 - III
(https://www.boxueio.com/series/reactive-programming-in-swift/ebook/79)

RxDataSource创建UITableView - I
(https://www.boxueio.com/series/reactive-programming-in-swift/ebook/81)