Reactive Programming in Swift

基于RxSwift的网络编程 - I
(https://www.boxueio.com/series/reactive-programming-in-swift/ebook/80)

RxDataSource创建UITableView - II
(https://www.boxueio.com/series/reactive-programming-in-swift/ebook/82)

RxDataSource创建UITableView - I

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

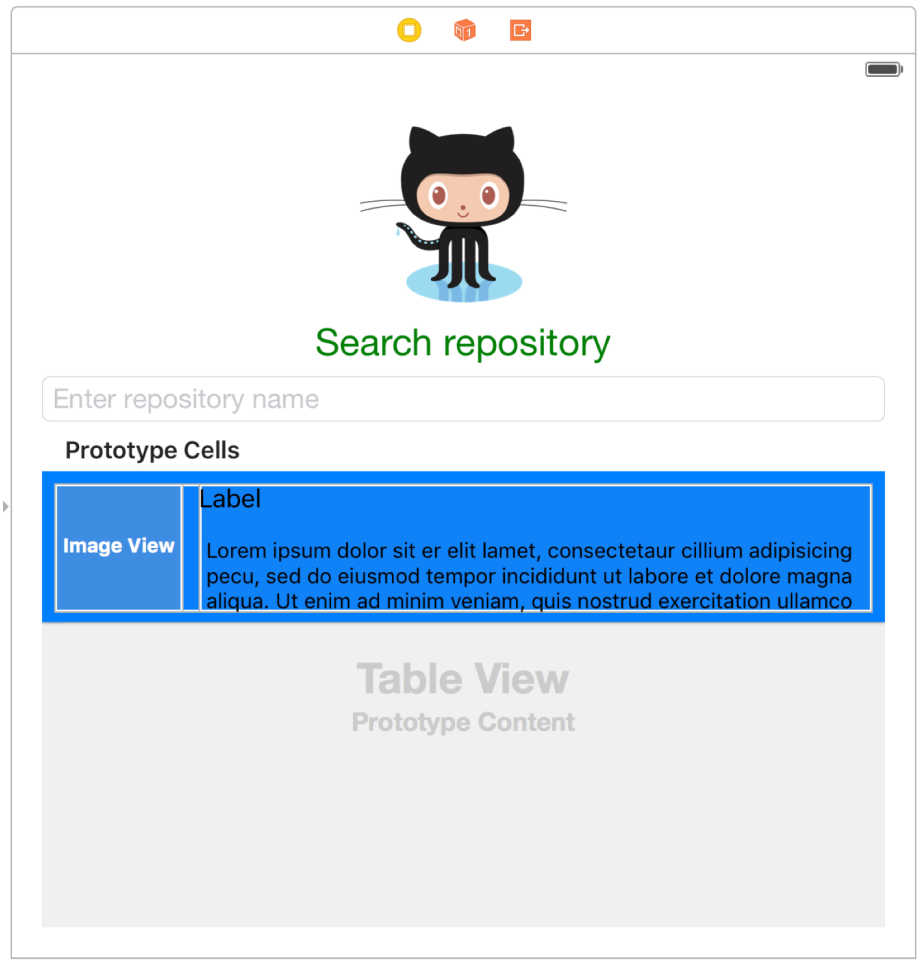
字号
字号
默认主题
金色主题
暗色主题

欢迎回来，在这段视频里，我们继续完成App的后半部分，基于RxDatasource，用reactive的方式处理 `UITableView` 。

准备工作

为了方便演示，基于上个视频完成的例子，我们做了一些额外的准备工作。大家可以在这里下载项目初始模板 (https://github.com/Boxue/episode-samples/tree/master/RxSwift/RxDataSource-I/RxDataSource-I-Starter)。

首先，我们给 `UITableView` 添加了一个Cell，在这个Cell里：



- 我们用一个 `UILabel` 和一个 `UITextView` 构成了一个垂直布局的StackView，表示项目的名字和描述；
- 用一个 `UIImageView` 和之前的StackView又构成了一个水平布局的StackView，最终形成了整个Cell的内容；

在这里，有一个技巧，我们可以在Stroyboard里，为Cell设置一个背景色，这样方便我们观察一个Cell里实际可以摆放内容的区域的大小；

其次，我们新建了一个叫做 `RepositoryInfoTableVieweCell` 的class，表示我们新创建的Cell。它定义我们需要访问的三个IBOutlet：

```
class RepositoryInfoTableViewCell: UITableViewCell {

    @IBOutlet weak var avatar: UIImageView!
    @IBOutlet weak var name: UILabel!
    @IBOutlet weak var detail: UITextView!

}
```

由于我们在Storyboard里设置了背景色，我们需要在 awakeFromNib 方法里，去掉它：

```
override func awakeFromNib() {
    super.awakeFromNib()
    // Initialization code
    self.backgroundColor = UIColor.clearColor()
}
```

第三，我们新建了一个 struct RepositoryModel ，表示Github返回的各种结果，我们将使用这个Model为 RepositoryInfoTableViewCell 赋值。并且，我们修改了 searchForGithub 和 parseGithubResponse ，使得最终我们可以直接订阅到一个 Observable<[RespositoryModel]> ；

第四，我们暂时去掉了 self.repositoryName.rx_text 的 subscribeNext 订阅，稍后，我们采用新的方式来订阅这个事件序列；

第五，我们在 ViewController extension 里，新添加了一个方法，用一个 UIAlertController 显示错误信息：

```
private func displayErrorAlert(error: NSError) {
    let alert = UIAlertController(
        title: "Network error",
        message: error.localizedDescription,
        preferredStyle: .Alert)

    alert.addAction(UIAlertAction(title: "OK",
        style: UIAlertActionStyle.Default,
        handler: nil))

    self.presentViewController(alert,
        animated: true, completion: nil)
}
```

最后，我们通过Cocoapods (https://cocoapods.org/)新安装了一个叫做RxDatasource的Swift模块：

```
# 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 'RxDataSources', '~> 0.7' # Our new swift module for data source
    pod 'SwiftyJSON', :git => 'https://github.com/SwiftyJSON/SwiftyJSON.gi
t'
end
```

准备完成之后，我们就可以开工了。

处理请求错误

在开始构建 UITableView 之前，我们先进一步完善网络请求的部分。当请求错误时，我们直接把Alamofire (https://github.com/Alamofire/Alamofire)返回的错误消息封装成了 .Error 事件。于是，我们可以这样来订阅请求成功和失败事件：

```
.subscribe(
    onNext: { repositoryModelArray in
        // We will create UITableView here later
    },
    onError: { error in
        let err = error as NSError
        self.displayErrorAlert(err)
    })
```

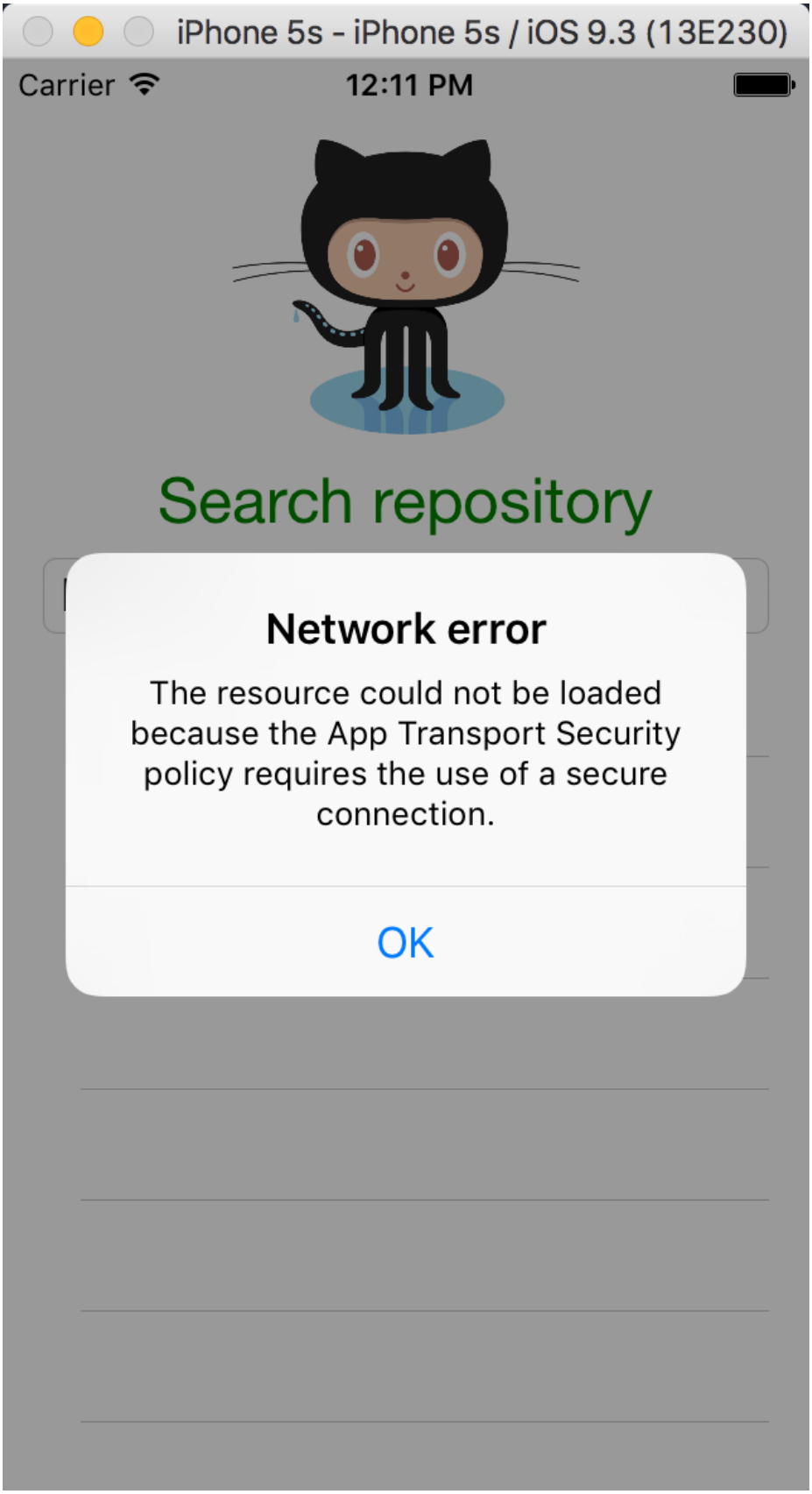
在上面这个例子里，我们分别通过 onNext 和 onError 参数传递了处理成功和失败事件的closure，其实这个版本的 subscribe 还有另外两个参数，是 onCompleted 和 onDisposed ，它们分别用于订阅事件序列的结束和回收。这样，要比我们在 subscribe 中使用 switch...case... 检查事件值方便一些。

但是，在我们这个例子里，由于我们使用了 .flatMap ，因此，我们是订阅不到网络请求事件序列中的 .Completed 事件的，它和 UITextField 输入事件序列的 .Completed 事件合并在一起了。

接下来，如果我们把 searchForGithub 中，请求的ur，由https改为http：

```
let url = "http://api.github.com/search/repositories"
```

重新编译执行，就可以看到相应的错误提示了：



用Rx的方式加载UITableView

接下来，我们使用RxSwift (https://github.com/ReactiveX/RxSwift)，把Github (https://github.com)的返回结果显示在下面的 `UITableView` 上。RxSwift (https://github.com/ReactiveX/RxSwift)允许我们通过几种不同的方式，通过订阅一个事件序列生成对应table cell对象，先来看最简单的一种。

首先，在网络请求成功的 `onNext` 部分，我们先重置 `searchResult` 的 `dataSource`：

```
self.searchResult.dataSource = nil
```

这是因为，每一次网络请求之后，我们需要重新订阅新的Observable来创建 UITableView ，如果不清空data source，RxSwift (https://github.com/ReactiveX/RxSwift)会报错。

一个复杂的bindTo

在订阅Github (https://github.com)返回结果之前，我们要先了解一个略显复杂的 bindTo 的用法，它的声明是这样的：

```
public func bindTo<R1, R2>(
    binder: Self -> R1 -> R2,
    curriedArgument: R1) -> R2
```

这里，参数 binder 仍旧用于指定一个订阅者，不同的是，这个订阅者可以接受一个closure做为参数，这个closure参数由 bindTo 的第二个参数，curriedArgument指定。简单来说， binder 可以调用 curriedArgument 指定的方法。

为什么要提到这个版本的 bindTo 呢？是因为我们要通过这样的方式来订阅 self.item 。为了简化代码，我们先定义一些 typealias：

```
typealias O = Observable<[RepositoryModal]>
typealias CC = (Int, RepositoryModel,
    RepositoryInfoTableViewCell) -> Void
```

然后，我们先来实现 binder：

```
let binder: O -> CC -> Disposable =
    self.searchResult.rx_itemsWithCellIdentifier(
        "RepositoryInfoCell",
        cellType:
            RepositoryInfoTableViewCell.self)
```

rx_itemsWithCellIdentifier 是RxSwift (https://github.com/ReactiveX/RxSwift)给 UITableView 添加的扩展，用于根据事件序列的值生成 UITableView 的每一行。它的返回值是一个Observer，也就是传递给 bindTo 方法的第一个参数。

但是， rx_itemsWithCellIdentifier 还需要调用另外一个Closure，用于执行具体的 UITableViewCell 的设置，这个Closure就是我们要传递给 bindTo 的第二个参数，curriedArgument：

```
let curriedArgument = { (
    rowIndex: Int,
    element: RepositoryModel,
    cell: RepositoryInfoTableViewCell) in

    cell.name?.text = element.name
    cell.detail?.text = element.detail
}
```

这个Closure有三个参数：

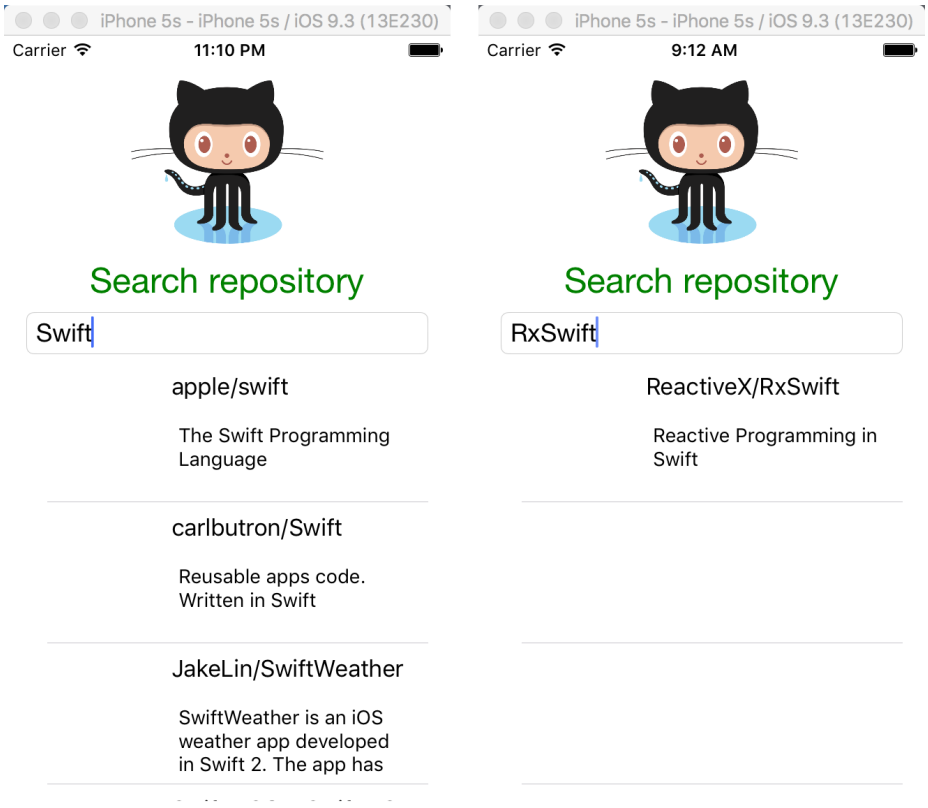
- 第一个参数是每一个Section里，row的索引。 rx_itemsWithCellIdentifier 只能生成只有一个Section的 UITableView ；
- 第二个参数，是用于生成每一个Cell需要的内容，在我们的例子里，就是一个 RepositoryModel 对象；
- 第三个参数，表示要生成的Cell对象，也就是 RepositoryInfoTableViewCell 对象；

接下来，在这个Closure内部，我们只是简单的设置了项目名称以及描述。

现在， binder 和 curriedArgument 都已齐备，我们可以调用 bindTo 通过订阅 self.item 创建 UITableView 了，和我们之前的代码相比，这反而是最简单的一步。我们先用 Observable.just 把Github的返回值包装成一个事件序列，然后，使用 bindTo 订阅它：

```
Observable.just(repositoryModelArray)
    .bindTo(binder,
        curriedArgument: curriedArgument)
    .addDisposableTo(self.bag)
```

然后， Command + R 编译执行，就可以看到结果了。并且，输入不同的内容， UITableView 可以自动更新：



Next?

这就是这段视频的内容，我们向大家介绍了如何通过 `subscribe` 订阅成功和失败事件，以及如何通过订阅事件，创建一个 `UITableView` 。在这里，使用 `rx_itemsWithCellIdentifier` 有一个小缺陷，就是无法创建包含多个Section的table，在下一段视频中，我们将通过自定义一个rx data source，解决这个问题。

⏮ 基于RxSwift的网络编程 - I
(https://www.boxueio.com/series/reactive-programming-in-swift/ebook/80)

RxDataSource创建UITableView - II ⏭
(https://www.boxueio.com/series/reactive-programming-in-swift/ebook/82)