RxDelegate代理UITableView事件

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

字号
字号
默认主题
金色主题
暗色主题

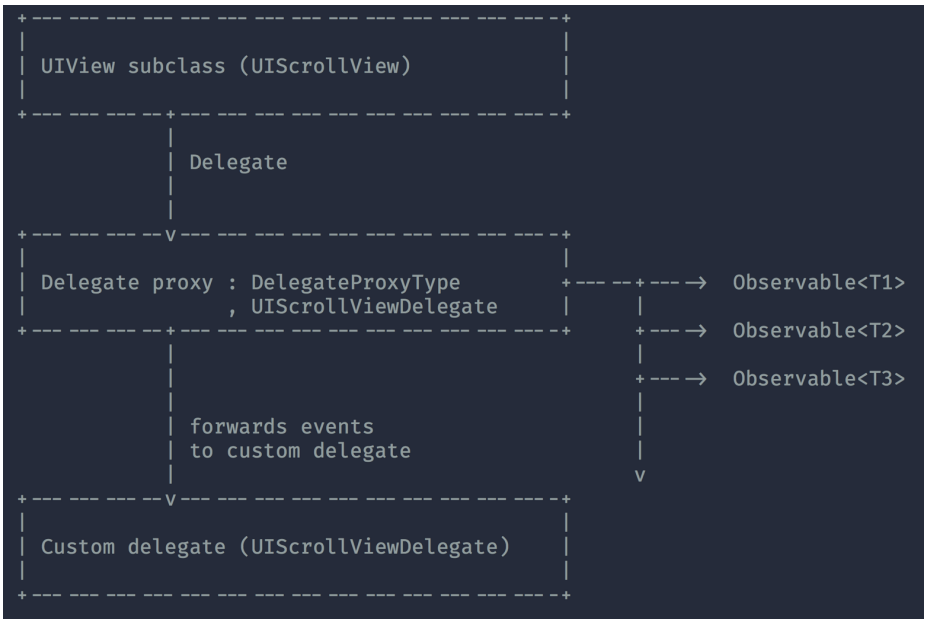
在上个视频的例子里，我们用Github的返回值，创建了一个Section UITableView。但是，我们还遗留了两个问题，一个是没有显示 `UITableView` 的section header，另一个是我们的Cell在点击之后，没有自动反选。用常规的 `UITableView` delegate实现它们很容易，定义两个方法就好了。但是，既然我们在讨论reactive programming，我们应该像下面这样来处理Cell点击事件：

```
self.searchResult
    .rxDidSelectRowAtIndexPath.subscribeNext {
    print("From delegate proxy")
    $0.deselectRowAtIndexPath($1, animated: true)
}.addDisposableTo(self.bag)
```

这看上去比直接定义delegate直观多了。RxSwift为我们提供了一种机制，叫做 `DelegateProxy` ，可以帮助我们实现上面例子中的 `rxDidSelectRowAtIndexPath` 。

什么是Delegate Proxy

在RxSwift的源代码里，我们可以找到一段关于 `DelegateProxy` 的解释。



简单来说，就是让只能注册一个delegate或data source的View，可以同时使用传统的delegate，以及observable事件序列来处理特定的事件。

在“接管”传统delegate的过程里，涉及到RxSwift中的两个类型：

- `DelegateProxyType`： 这是一个protocol，定义了为了创建proxy以及转发事件需要的方法；
- `DelegateProxy`： 这是一份 `DelegateProxyType` 的基础实现；

而我们要做的，就是从 `DelegateProxy` 派生一个遵从 `DelegateProxyType` 的类，然后根据自己的需要重定义必要的方法。别担心，这并不复杂。

接下来，我们就自定义一个delegate proxy来处理table cell被点击的事件，让它自动反选。

自定义delegate proxy class

我们像下面这样：

```
class MyRxTableViewDelegateProxy
    : DelegateProxy
    ,UITableViewDelegate
    ,DelegateProxyType {
}
```

在项目里新添加一个新的delegate proxy， `MyRxTableViewDelegateProxy` ，它除了从 `DelegateProxy` 派生并遵从 `DelegateProxyType` 之外，还要遵从它要“接管”的delegate，在我们的例子里，就是 `UITableViewDelegate` 。

然后，我们要实现 `DelegateProxyType` 中的两个方法。

第一个是让delegate proxy获取“原生delegate对象”的方法：

```
static func currentDelegateFor(
    object: AnyObject) -> AnyObject? {
    let tableView = object as! UITableView
    return tableView.delegate
}
```

`currentDelegate` 接受一个参数，返回它的 `delegate` 对象。

第二个是设置delegate proxy对象的方法：

```
static func setCurrentDelegate(
    delegate: AnyObject?,
    toObject object: AnyObject
) {
    let tableView = object as! UITableView
    tableView.delegate = delegate as? UITableViewDelegate
}
```

`setCurrentDelegate` 把 `toObject` 对象的 `delegate` 属性，设置成参数中delegate。

扩展UITableView

接下来，我们要通过 `UITableView` extension把 `MyRxTableViewDelegateProxy` 添加进来：

```
extension UITableView {
    var rxDelegate: MyRxTableViewDelegateProxy {
        return MyRxTableViewDelegateProxy
            .proxyForObject(self)
    }
}
```

`rxDelegate` 是一个computed property，这里要说明的是我们**不能直接生成一个MyRxTableViewDelegateProxy对象**，我们要使用 `proxyForObject` 方法来创建。把要“接管”delegate的对象传递给它，在我们的例子里，也就是 `UITableView` 自身。

接管 `UITableView` 事件

接下来，我们在 `UITableView` extension中添加另外一个computed property，它是一个Observable，我们希望当table cell被点击的时候，它可以发送事件。并且，为了能够处理点击事件，我们希望发送的事件中，包含被点击的 `UITableView` 对象，以及table cell的 `NSIndexPath` 。

```
var rxDidSelectRowAtIndexPath
    : Observable<(UITableView, NSIndexPath)>
```

接下来，我们通过监控tableview“原生delegate方法”来实现这个Observable。 `DelegateProxy` 提供了一个叫做 `observe` 方法，它接受一个 `#selector` 类型参数，表示要监控的方法。

一个更好的Selector

在继续之前，我们先介绍一个在Swift里更“安全”的使用 `#selector` 的方法。因为尽管Swift 2中 `#selector` 可以使用方法签名代替签名字符串了，但无论如何，面对一长串复杂的名字，它还是太“土”了。我们需要一个更优雅使用 `#selector` 的方法。

好在，一个叫Anddy Hope (https://medium.com/swift-programming/swift-selector-syntax-sugar-81c8a8b10df3#.unfgncty1)的人提出了一个好点子。我们先对 Selector 类型添加一个 private extension ，这样，我们所有对 Selector 的扩展就只在当前文件中生效了。然后，我们添加一个 static 属性，它的值就是我们需要的 #selector：

```
private extension Selector {
    static let didSelectRowAtIndexPath =
        #selector(UITableViewDelegate
            .tableView(_:didSelectRowAtIndexPath:))
}
```

这样，在任何一个需要 UITableViewDelegate.tableView(_:didSelectRowAtIndexPath:) 这个selector的地方，我们就可以直接使用 .didSelectorRowAtIndexPath 了。显然，这比它那个Objective-C签名要优雅多了。

把方法调用封装成Observable

在 rxDidSelectRowAtIndexPath 的实现里，添加下面的代码：

```
var rxDidSelectRowAtIndexPath:
    Observable<(UITableView, NSIndexPath)> {
    return rxDelegate.observe(.didSelectRowAtIndexPath)
        .map { params in
            return (params[0] as! UITableView,
                params[1] as! NSIndexPath)
        }
}
```

其中， observe 监控到方法被调用后，发送的事件值，是一个 [AnyObject] ，包含了调用selector指定的方法时，传递的所有参数。我们使用 map 把 .didSelectRowAtIndexPath 的两个参数变成了一个tuple。

至此， MyRxTableViewDelegateProxy 的实现和 UITableView 的改造就完成了。接下来，我们回到 ViewController ，来订阅table cell被选中的事件。

订阅rxDidSelectRowAtIndexPath

在 viewDidLoad 方法里，添加下面的代码：

```
self.searchResult.rxDidSelectRowAtIndexPath
    .subscribeNext {
    print("From delegate proxy")
    $0.deselectRowAtIndexPath($1, animated: true)
}.addDisposableTo(self.bag)
```

这样，我们就订阅到了table view被选中的事件，我们向控制台打印了一个消息，然后对当前选中的cell反选。

完成后，重新编译执行，就能看到反选的效果了。

“原生delegate”仍旧是可用的

接下来，我们实现section header的部分。当然，参照 rxDidSelectRowAtIndexPath 如法炮制一个 rxViewForHeaderInSection 固然没问题。但这次，我们通过“原生delegate”来演示它和delegate proxy的协同工作。

首先，我们让 ViewController 遵从 UITableViewDelegate：

```
class ViewController
        : UIViewController, UITableViewDelegate
```

并且在 viewDidLoad 方法里，设置 self.searchResult 的“原生delegate”：

```
self.searchResult.delegate = self
```

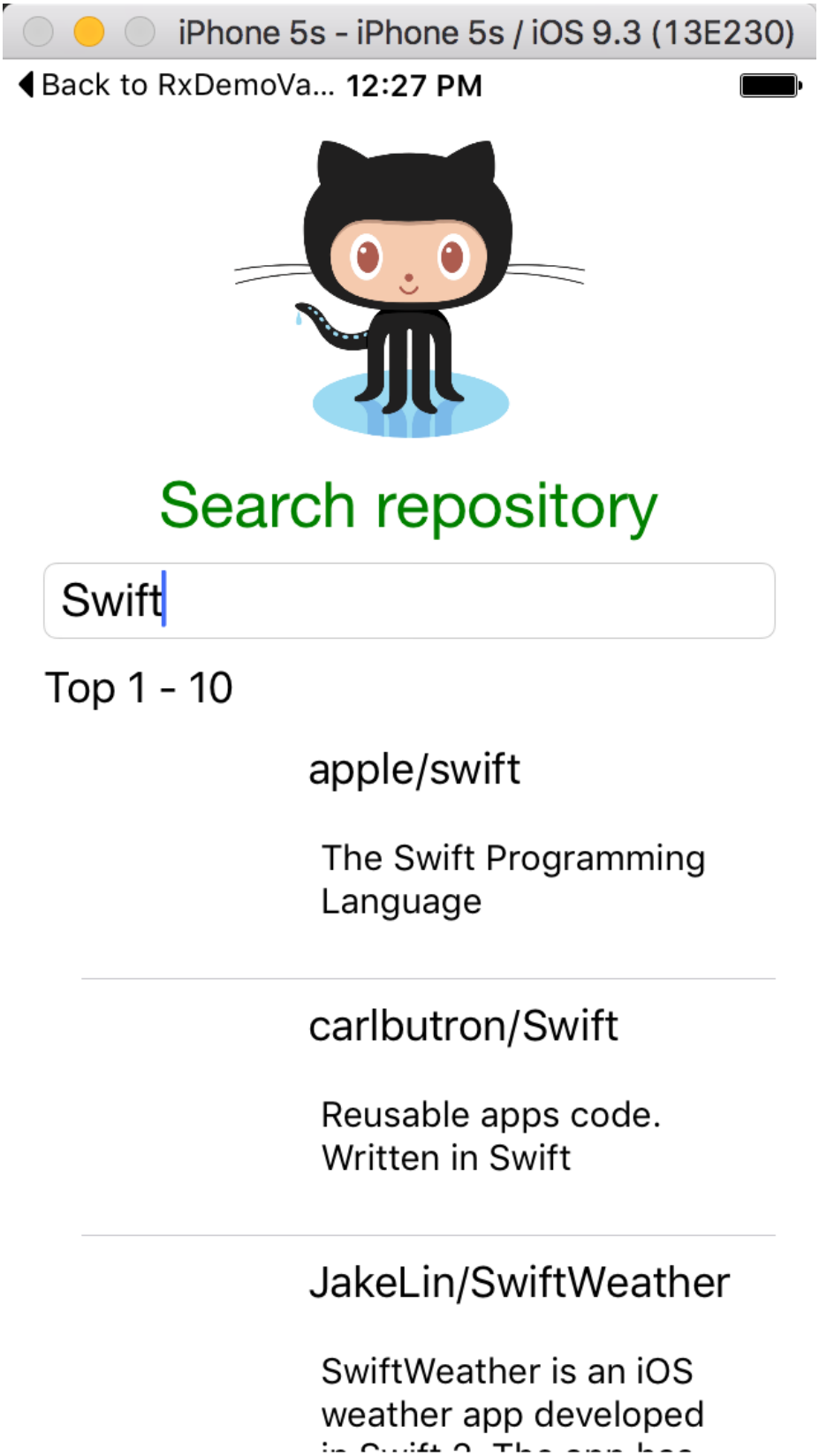
其次，新添加一个 ViewController extension专门设置 UITableViewDelegate：

```
func tableView(tableView: UITableView,
    viewForHeaderInSection section: Int) -> UIView? {
    let sectionCount = self.dataSource
        .numberOfSectionsInTableView(tableView)
    guard  sectionCount != 0 else {
        return nil
    }

    let label = UILabel(frame: CGRect.zero)
    label.text = self.dataSource
        .sectionAtIndex(section).model ?? ""

    return label
}
```

Command + R 重新编译执行，就可以看到table section header了。



“原生delegate”和”DelegateProxy”同时处理同一UI事件

最后，我们来看一种情况，如果我们让 UITableView 的“原生delegate”和“DelegateProxy”同时处理同一个UI事件时，就会看到事件会先被delegate pros处理，而后，会被“原生delegate”处理。

在 ViewController extension中，添加处理table cell被点击的代码：