Todo I - 通过一个真实的App体会Rx的基本概念

RxSwift - step by step

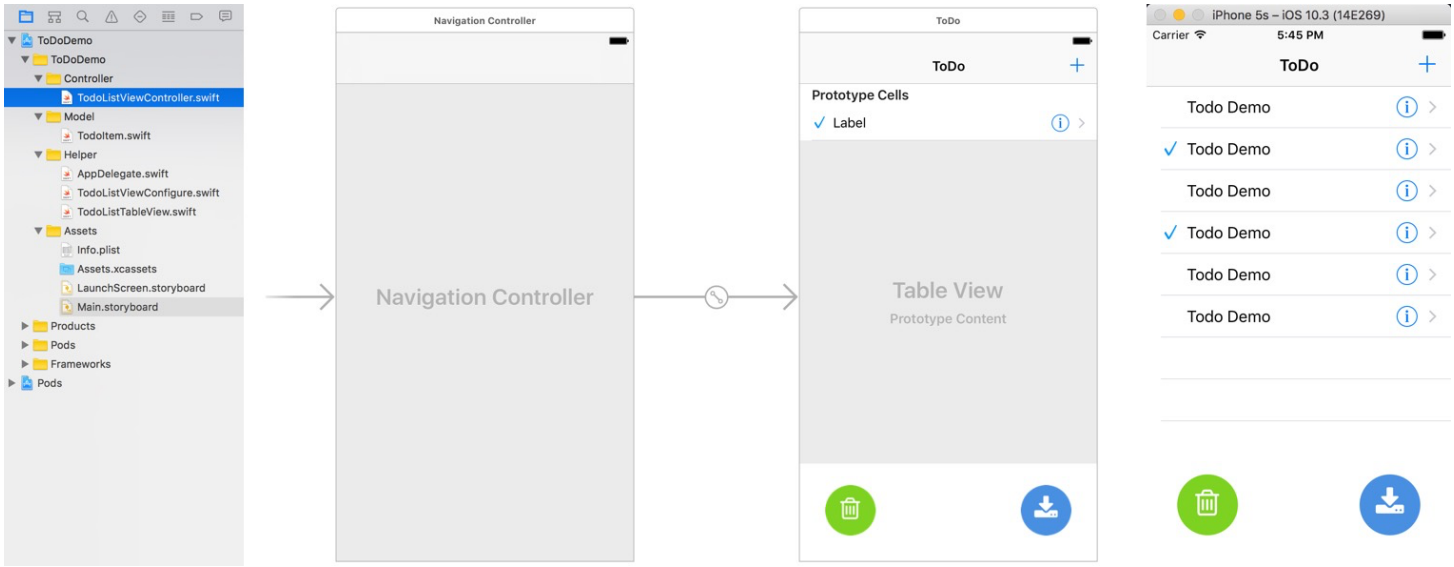
← 返回视频列表 | 预计阅读时间: *30分钟* | ‹ PREVIOUS | NEXT ›

经过了前面几节的内容铺垫后，现在，是时候在一个真的App里感受下这些Rx的概念了。在日常的开发中，如何通过RxSwift绑定UI和Model？如何在不同的Controller之间共享数据？通过对这些内容的实践，你就会更真实的感受到之前提到的那些基本概念的含义。

当然，作为开始，我们的目标还不是一个MVVM架构的App，那是最终的目标。在这一节，我们先从一个常规开发的App开始，用Rx的思想改造一些常规功能的实现，以此，加深对Observable，Subscribe，Subject，Dispose这些概念的理解。

ToDo App

首先，大家可以在Github上下载RxToDoDemo源代码，进入ToDoDemoStarter目录，这是项目的起始源代码。先执行`pod install`安装RxSwift，完成后，打开`ToDoDemo.xcworkspace`。



继续之前，我们先简单了解下这个项目：

首先，Model目录中，是App使用的数据，它是一个遵从`NSCoding`的类，方便我们序列化成plist保存和加载。其中，`name`表示ToDo的标题，`isFinished`表示是否完成；

```
class TodoItem: NSObject, NSCoding {
    var name: String = ""
    var isFinished: Bool = false

    // ...
}
```

其次，Assets目录中，是App的UI。在Main.storyboard中，我们希望点击App右上角的+添加新的todo，点击todo内容所在行可以用一个蓝色的对勾切换完成状态，下面的绿色按钮清空整个todo列表；蓝色按钮保存当前所有的todo内容和完成状态；

第三，Controllers目录中是目前App唯一的view controller。它包含了App的Model、`@IBOutlet`以及`@IBAction`。在最开始的这个版本里，为了简单起见，我们让所有添加的todo内容和状态都是相同的。

```
class TodoListViewController: UIViewController {
    var todoItems: [TodoItem] = []
    @IBOutlet weak var tableView: UITableView!

    required init?(coder aDecoder: NSCoder) {
        // ...
    }

    @IBAction func addTodoItem(_ sender: Any) {
        // ...
    }

    @IBAction func saveTodoList(_ sender: Any) {
        // ...
    }

    @IBAction func clearTodoList(_ sender: Any) {
        // ...
    }
}
```

最后，为了实现`TodoListViewController`中的功能，我们把一些具体的功能代码放在了*Helper*目录，其中*TodoListTableView.swift*中存放的是table view的data source以及delegate，*TodoListViewConfigure.swift*中存放的，则是保存和加载todo model相关的代码。

对Todo的响应式改造

Variable

对这个App的第一个改造，是让`TodoListViewController`中的model变成响应式的，为此，我们把之前的`todoItems`变成一个`Variable`，并添加一个用于回收取消订阅的`DisposeBag`对象：

```
// In TodoListViewController.swift

class TodoListViewController: UIViewController {
    let todoItems = Variable<[TodoItem]>([])
    let bag = DisposeBag()

    // ...
}
```

这样一来，所有之前直接访问`todoItems`数据的部分，都要改成访问`todoItems.value`。首先，是显示Todo列表的`UITableView`，打开TodoListTableView.swift，修改对应的data source和delegate方法。唯一需要注意的是，在左滑删除的代码里，我们只是删除了`todoItems`的数据，而没有执行删除cell UI的代码。稍后就会看到，在`todoItems`变成`Variable`之后，所有UI相关的代码，将会在对其的订阅中统一处理。

```
// UITableView delegate
extension TodoListViewController: UITableViewDelegate {
    func tableView(_ tableView: UITableView,
                   didSelectRowAt indexPath: IndexPath) {
        if let cell = tableView.cellForRow(at: indexPath) {
            let todo = todoItems.value[indexPath.row]

            todo.toggleFinished()
            configureStatus(for: cell, with: todo)
        }

        tableView.deselectRow(at: indexPath, animated: true)
    }

    func tableView(_ tableView: UITableView,
                   commit editingStyle: UITableViewCellEditingStyle,
                   forRowAt indexPath: IndexPath) {
        todoItems.value.remove(at: indexPath.row)
    }
}

// UITableView data source
extension TodoListViewController: UITableViewDataSource {
    func tableView(_ tableView: UITableView,
                   numberOfRowsInSection section: Int) -> Int {
        return self.todoItems.value.count
    }

    func tableView(_ tableView: UITableView,
                   cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(
            withIdentifier: "TodoItem", for: indexPath)
        let todo = todoItems.value[indexPath.row]

        configureLabel(for: cell, with: todo)
        configureStatus(for: cell, with: todo)

        return cell
    }
}
```

其次，修改通过storyboard初始化的`init?`方法，此时，我们已经不需要在这里初始化`todoItems`了：

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    loadTodoItems()
}
```

第三，是序列化`todoItems`的时候，要改成访问`todoItems.value`属性。在TodoListViewConfigure.swift里，把`saveTodoItems`和`loadTodoItems`修改成下面这样：

```
func saveTodoItems() {
    let data = NSMutableData()
    let archiver = NSKeyedArchiver(forWritingWith: data)

    archiver.encode(todoItems.value, forKey: "TodoItems")
    archiver.finishEncoding()

    data.write(to: dataFilePath(), atomically: true)
}

func loadTodoItems() {
    let path = dataFilePath()

    if let data = try? Data(contentsOf: path) {
        let unarchiver = NSKeyedUnarchiver(forReadingWith: data)
        todoItems.value =
            unarchiver.decodeObject(forKey: "TodoItems") as! [TodoItem]

        unarchiver.finishDecoding()
    }
}
```

第四，把TodoListViewController.swift中，保存和清除Todo列表的代码改成这样：

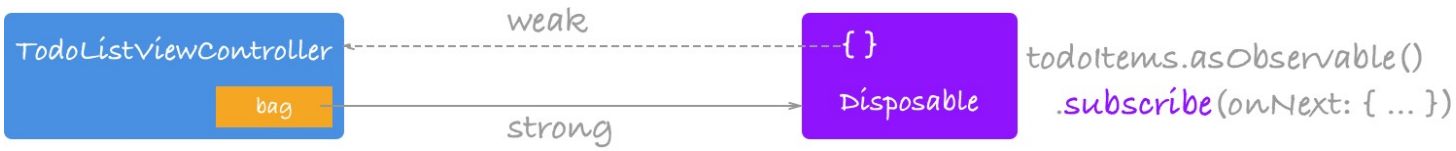
```
class TodoListViewController: ViewController {
    @IBAction func addTodoItem(_ sender: Any) {
        let todoItem = TodoItem(name: "Todo Demo", isFinished: false)
        todoItems.value.append(todoItem)
    }

    @IBAction func clearTodoList(_ sender: Any) {
        todoItems.value.removeAll()
    }
}
```

可以看到，此时，这两部分代码也只是在处理todoItems自身，而没有UI相关的代码了。都修改完成之后，按Cmd + B构建一次，确认没有错误。现在，我们来着手处理当todoItems的值更新时，对应UI的修改。

由于todoItems是一个Subject，作为一个observer，我们修改它的值，就相当于它自己订阅到了事件。而要响应值的修改，我们就把它当作一个observable直接订阅就好了。在viewDidLoad里，添加下面的代码：

```
todoItems.asObservable().subscribe(
    onNext: { [weak self] todos in
    self?.updateUI(todos: todos)
}).addDisposableTo(bag)
```

很简单，当发现todoItems的值发生变化的时候，调用TodoListViewController中的updateUI方法更新界面，稍后，我们就来实现这个方法。现在，先来看onNext closure中捕获的self，为什么要用weak呢?



如上图所示，`subscribe`方法返回的`Disposable`对象被`bag`管理，因此`bag`持有一个strong reference；此时，如果`Disposable`对象的`onNext` clousre中持有指回`self`的strong reference，`TodoListViewController`对象和`Disposable`对象之间就会形成引用循环了。因此，这里要使用`weak self`。

理解了这个问题之后，我们来实现`updateUI`方法：

```
func updateUI(todos: [TodoItem]) {
    self.tableView.reloadData()
}
```

很简单对不对？我们只要让`tableView`对象重新加载数据就好了，尽管这不是一个高效的方法，也还有很多交互细节可以改进，但至少你可以感觉到，通过Subject，我们把根据`todoItems`的值更新UI的代码，都放到了一起。

绑定更多和UI相关的操作

看到这里，你可能会觉得，这一点点小改进没什么，至少不足以激起Rx对你的兴趣。接下来，我们再对UI进行一点约束，例如：

- 顶部的标题应该显示当前todo的个数；
- 清空列表后应该禁用绿色按钮；
- 限制最多只能存在4个未完成的todo，否则就禁用添加按钮；

传统的方式怎么办呢？你可能会想到针对`todoItems`利用KVO的机制来解决问题，但毕竟我们在使用Swift，一来，KVO只能处理有限类型的属性；二来，我们似乎一下子又回到了披着Swift外衣的OC世界；最后，KVO的使用在Swift中也真的非常不方便，单就那一长串`#selector`就会让代码看上去并不那么Swift。

现在，有了RxSwift，`todoItems`变成了一个Subject，为了实现上面的功能，我们只要在`updateUI`中添加几行代码就搞定了：

```
func updateUI(todos: [TodoItem]) {
    clearTodoBtn.isEnabled = !todos.isEmpty
    addTodo.isEnabled =
        todos.filter { !$0.isFinished }.count < 4
    title = todos.isEmpty ? "Todo" : "\(todos.count) ToDos"

    self.tableView.reloadData()
}
```

怎么样？是不是看着就很Swift。执行一下就会发现，前两个功能都好用，限制未完成todo个数的功能并不好用。这是因为，我们订阅的`todoItems`并不会响应数组中成员的属性被修改的事件，因此，编辑已有todo的完成状态并不会给`todoItems`发送通知。这里，一个简单的办法就是，在TodoListTableView.swift中，把处理cell自动反选的代码改成这样：

```
func tableView(_ tableView: UITableView,
               didSelectRowAt indexPath: IndexPath) {
    if let cell = tableView.cellForRow(at: indexPath) {
        let todo = todoItems.value[indexPath.row]
        todo.toggleFinished()

        // Trigger event
        todoItems.value[indexPath.row] = todo

        configureStatus(for: cell, with: todo)
    }

    tableView.deselectRow(at: indexPath, animated: true)
}
```

通过给对应位置的`todoItems`赋值，我们就可以变相触发事件，进而订阅到`todoItems`的值了。

What's next?

在这个简单的例子里，我们开始把一个用传统方式编写的App，进行一点改进，初步体会了如何通过RxSwift把更新Model和更新UI的代码进行分离。但此时，添加新Todo的功能还没有实现，在下一节，我们就来看如何通过Subject简化在不同的Controller之间传递数据，并实现新建和编辑Todo的功能。

‹ Prev: 四种Subject的基本用法 | Todo I - 通过一个真实的App体会Rx的基本概念 | Next: Todo II - 如何通过Subject传递数据 ›