目录

[MVVM实战 2](#_Toc10626985)

[课程目的 2](#_Toc10626986)

[MVVM基础知识 2](#_Toc10626987)

[**MVVM解决了什么问题？** 3](#_Toc10626988)

[**对比MVC与MVVM** 4](#_Toc10626989)

[**MVVM中的绑定** 5](#_Toc10626990)

[MVVM编程实战 6](#_Toc10626991)

[**1：简单列表展示（Demo 3）** 6](#_Toc10626992)

[**2：复杂列表页面展示（Demo 1）** 10](#_Toc10626993)

[**3：表单页面（Demo 2）** 15](#_Toc10626994)

[**其他细节** 20](#_Toc10626995)

[扩展阅读 21](#_Toc10626996)

**MVVM实战**

主讲人：李国庭

**课程目的**

本教程主要针对iOS研发人员，通过介绍大致了解我们架构方向和单测目的，掌握单测编写能力。

通过本教程，可以收获如下：

* 了解MVVM的原理、优势和不足
* 在典型的业务场景中使用MVVM模式
* 利用MVVM的优势，实现代码复用
* 使用MVVM进行单元测试

**MVVM基础知识**

MVVM是一种软件架构模式^1。最早由微软架构师Ken Cooper和Ted Peters开发。尽管微软大力推广的WPF和SilverLight并没有得到市场的充分认可，但MVVM架构的模式却非常有可取之处。

初期考虑是看中了MVVM模式容易进行单元测试的特点，因为ViewModel本质是一个普通OC对象，对它进行测试可以脱离UI环境，也不需要复杂的mock开发。

在iOS领域实践MVVM模式，通常需要使用RAC(ReactiveCocoa)去实施FRP(Functional Reactive Progamming[^2](https://en.wikipedia.org/wiki/Functional_reactive_programming))。这是一个伪命题，本质上讲实现MVVM并不需要使用RAC，但脱离了RAC或与之类似的框架，再实现MVVM的代码将变得非常丑陋。为什么说是伪命题呢？因为ReactiveCocoa和FRP和传统的OOP（面向对象编程）思路并非一脉相承。熟练的掌握RAC和FRP有较高的学习曲线。这次分享就是希望通过一些场景，帮助大家对MVVM有个系统认识和基本的规范。

**MVVM解决了什么问题？**

很多文章会说MVVM可以解决许多MVC中遇到的问题，比如**厚重的ViewController**、**遗失的网络逻辑**（没有属于它的位置）、**较差的可测试性**等！但这一点还有待商榷。MVC写得太重是因为解耦没有做好，层次模块没有划分清楚。如果没有很好的解除耦合和划分清楚层次模块，ViewModel一样会变重。

使用MVVM，可以解决两个问题:

1. 实现**代码复用**。有利于开发业务无关UI组件
2. 实现UI组件级别的**单元测试**，而不需要复杂的Mock。

**对比MVC与MVVM**

首先参考MVC与MVVM的逻辑示意图：

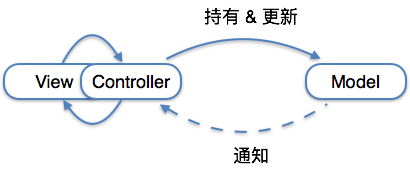


图1 MVC模式

* **View** — 显示用户界面
* **Model** — 实现业务逻辑
* **Controller** — 起到不同层面间的组织作用，用于控制应用程序的流程

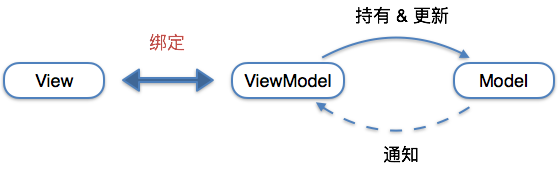
[](https://github.com/liangshixing/mvvmdemo/blob/master/images/MVVM.png)

图2 MVVM模式

* **View** — 显示用户界面
* **Model** — 实现业务逻辑
* **ViewModel** — 通过一系列属性和命令，对视图进行抽象

从示意图来看，MVVM本质的区别在与，ViewModel是通过“绑定”方式与View进行交互。这会带来两点好处：

1. 代码复用 - ViewModel与View之间的耦合关系更加松散，从而可以实现View的复用。后面做详细阐述。
2. 单元测试 - 由于“绑定”关系是通过三方框架建立的，绑定的结果是可信的。因此只需要测试到ViewModel一级，就可以覆盖业务逻辑在UI上的体现。又由于ViewModel本身都是普通Objective-C对象，不需要复杂的Mock就可以运行，所以更容易进行单元测试。

**总的来说，**viewModel主要是视图显示逻辑、验证逻辑、网络请求等代码存放的地方，唯一要注意的是，任何视图本身的引用都不应该放在VM中，换句话说就是VM中不要引入UIKit.h。这样，首先解决了VC臃肿的问题，将逻辑代码、网络请求等都写入了VM中，然后又由于VM中包含了所有的展示逻辑而且不会引用V，所以它是可以通过编程充分测试的。

**MVVM中的绑定**

在MVVM模式中，绑定非常重要，主要包含以下几种形式：

* 单向数据绑定
* 集合数据绑定
* 双向数据绑定
* 执行过程绑定
* 错误处理

接下来我们会通过具体的Demo体现这些绑定的用法。

**MVVM编程实战**

下面的几个Demo会体现MVVM模式在单元测试方面体现出来的优势，也会分别体现出上面列举的这些绑定场景。

**1：简单列表展示（Demo 3）**

Demo代码部分模拟了一个获取城市列表并展示的功能，通过本例，我们可以接触到：

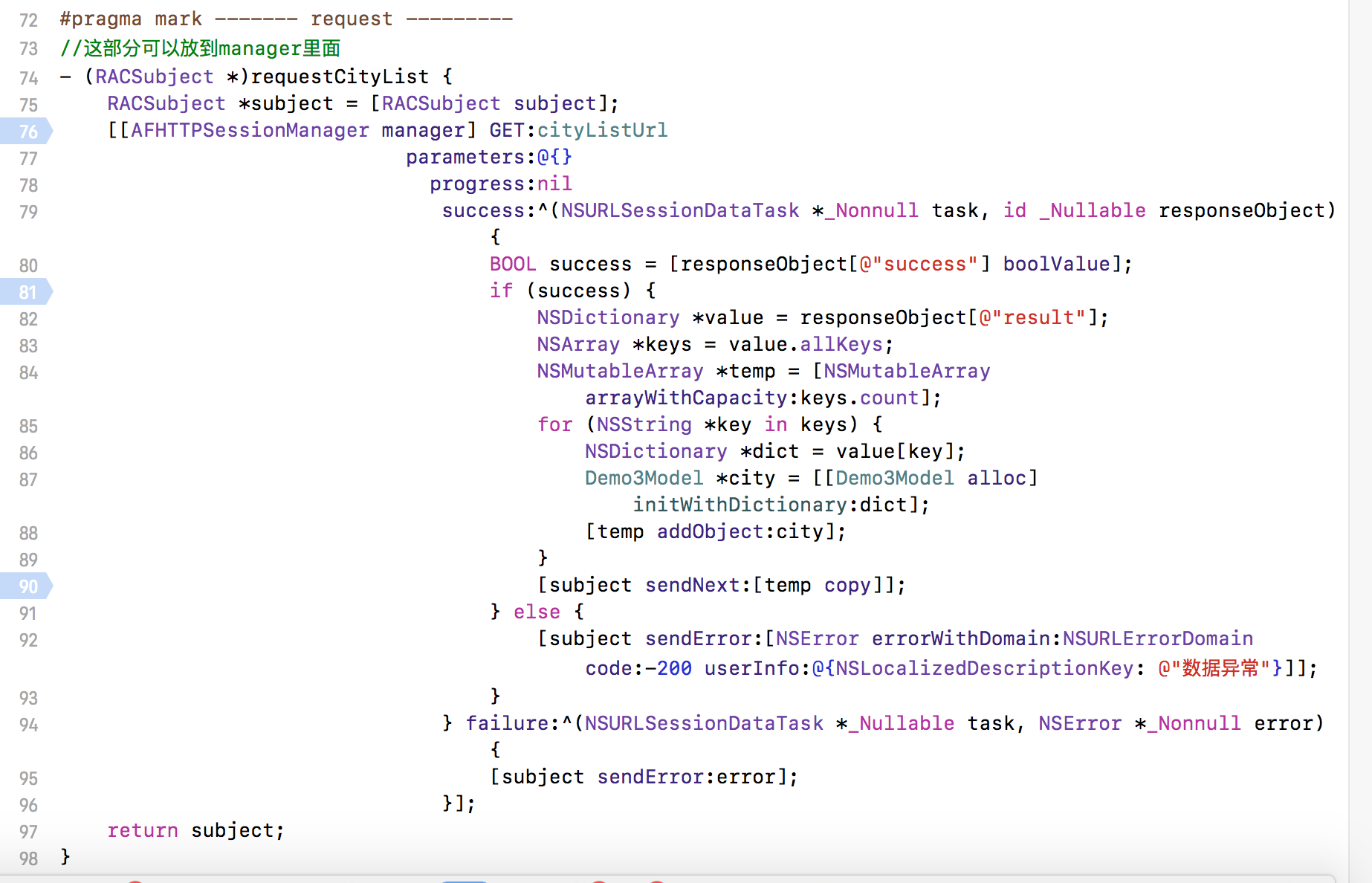
* 基于绑定实现的UI
  + 单向数据绑定
  + 执行过程绑定
  + 错误处理
* RAC的常见操作

本例中UI全部用代码实现，UI相关代码就不在这里体现。

之前我们提到过，ViewModel需要提供“UI属性”用于和View进行绑定。由于这个例子主要体现mvvm分工和结构思想，所以很简洁，对于这部分的绑定我们后面demo1中讲。接下来我们继续基于这个例子进行讲解；下面介绍相应的TableView。接下来这部分会涉及到：

* 执行过程绑定
* 错误处理

首先介绍TableView的Model层，通过网络拉取数据（这里相当于我们自己业务线封装的网络层，返回我们想要的model数组）。这里用**AF**网络请求：



**requestCityList**方法模拟了拉城市数据的过程，这里值得注意的是，由于这个方法构造了个冷信号，在执行这两个方法时，并不会真的触发网络请求。网络请求会在订阅信号的时候触发。同理，如果这个方法返回的信号被不同的地方进行了多次订阅，就会触发多次网络请求。希望了解更多冷热信号的相关知识，可以参考这几篇文章：

* <http://tech.meituan.com/talk-about-reactivecocoas-cold-signal-and-hot-signal-part-1.html>
* <http://tech.meituan.com/talk-about-reactivecocoas-cold-signal-and-hot-signal-part-2.html>
* [http://tech.meituan.com/talk-about-reactivecocoas-cold-signal-and-hot-signal-part-3.html](http://tech.meituan.com/talk-about-reactivecocoas-cold-signal-and-hot-signal-part-1.html)

TableView对应的ViewModel接口如下：

其中fetchCityListCommand用于执行返回一个信号创建的操作；其内容为将信号返回数据数组重新组装为cell需要的vm；并将其赋值给cityListViewModels，如果是返回错误信号就将错误数据赋值给cityListError。本例中，在监测到数组变化时reload整个tableView；从而完成数据流的获取和展示过程。

loadDataCommand是RACCommand类型，可以实现“执行过程绑定”。触发RACCommand，会执行一段代码逻辑。在本例中，会调用ViewModel的接口，拉取数据。

RACCommand中包含几个非常有用的属性：

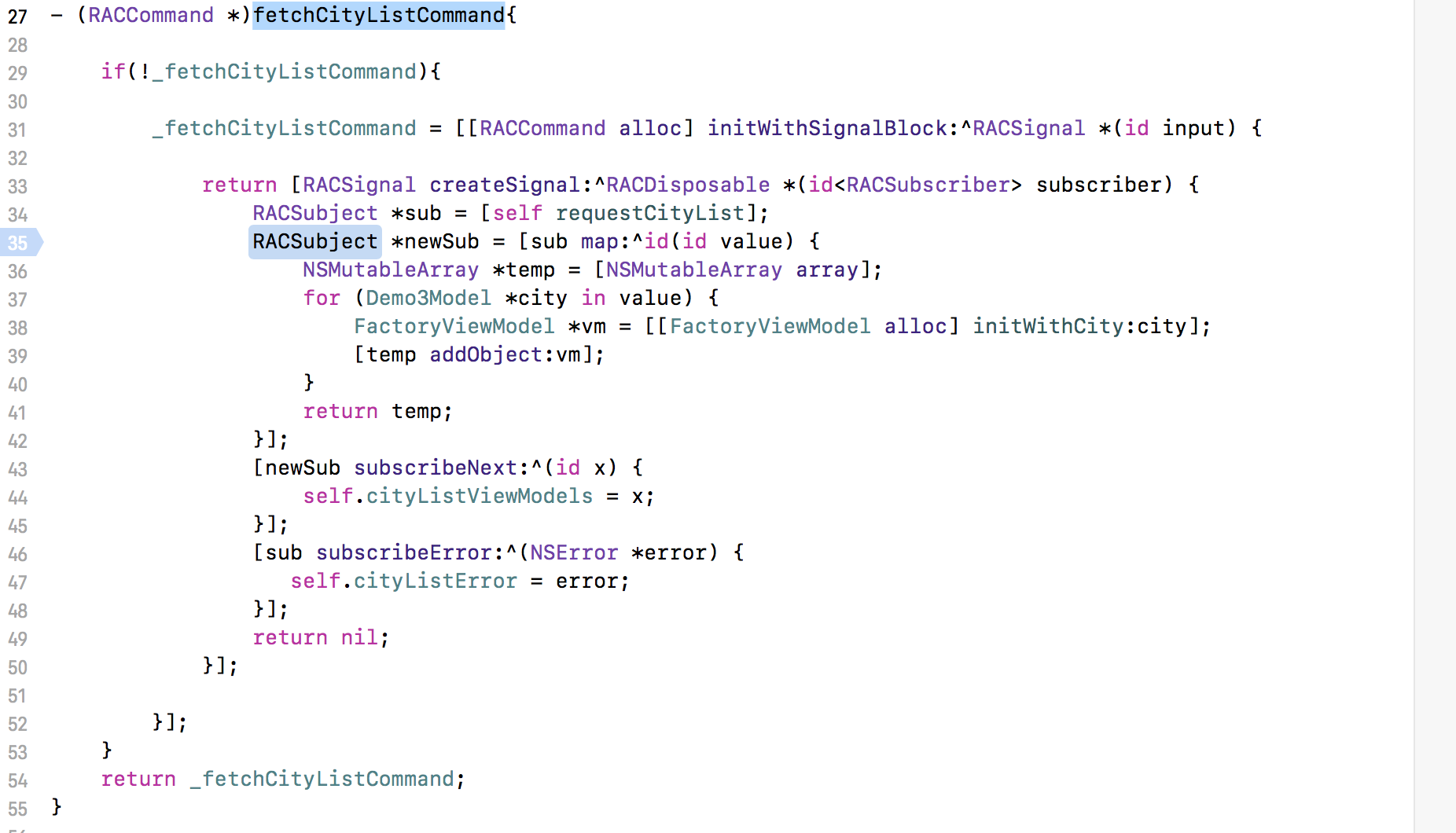
- @property (nonatomic, strong, readonly) RACSignal \*executionSignals;

- @property (nonatomic, strong, readonly) RACSignal \*executing;

- @property (nonatomic, strong, readonly) RACSignal \*errors;

其中，executionSignals是高阶信号（Next值是信号的信号），它里面包含了RACCommand每一次执行时返回的信号，本例中[viewModel.clickCityCommand.executionSignals switchToLatest]是指获取最近一次返回的信号；executing信号会返回当前的RACCommand是否处于执行状态；errors信号则会在RACCommand执行发生错误时，返回Error对象。

看一下fetchCityListCommand的内部实现：

这里会执行请求数据并将数据组装为cell需要的工厂类返回给数组，赋值给cityListViewModels；让后将它们打包成一个信号，再返回。

再看看VC的实现：



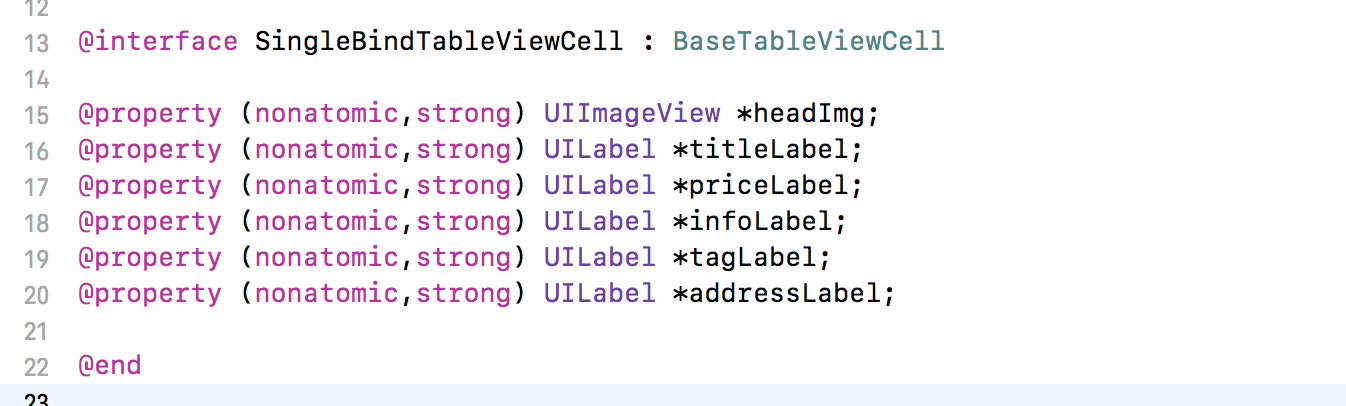
首先执行command操作；当监听到cityListViewModels有变化的时候执行table的reload操作；从而实现数据展示。

**2：复杂列表页面展示（Demo 1）**

通过本例，我们可以接触到：

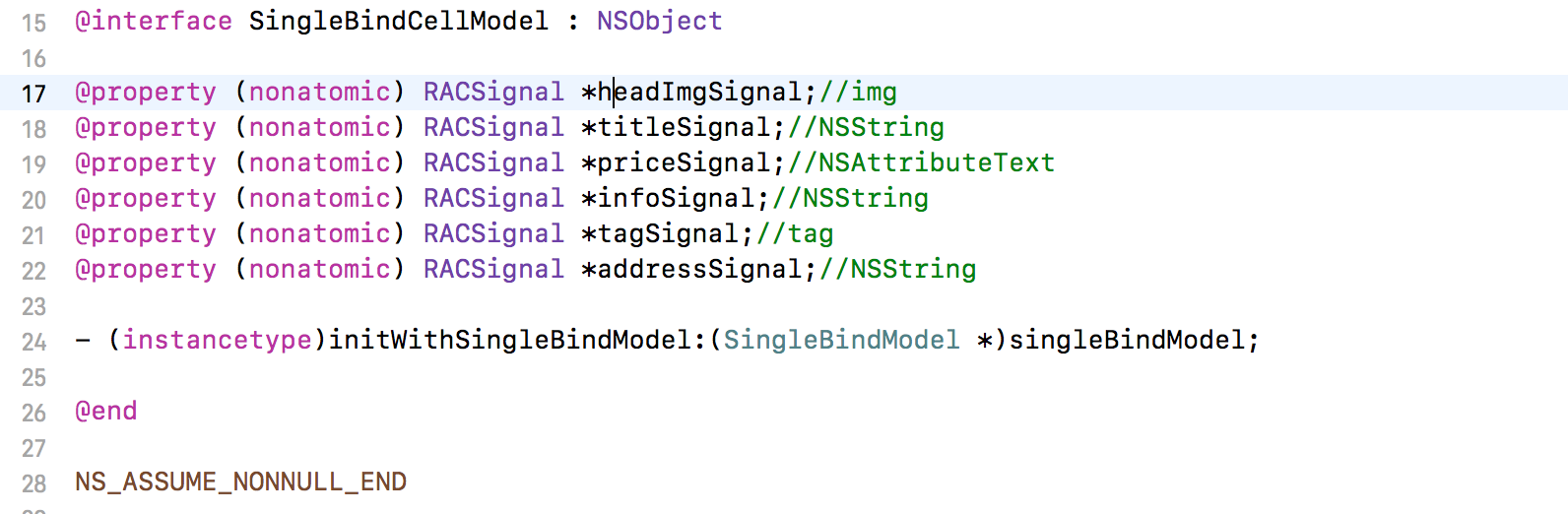
* 基于绑定实现的UI
  + 单向数据绑定
  + 集合数据绑定
  + 执行过程绑定
  + 错误处理
* MVVM模式下实现的代码复用
* RAC的常见操作

首先创建TableViewCell及其ViewModel。



按照前面的介绍，ViewModel中包含**UI属性**，用于与View进行绑定；并且通过Model中实现具体的业务逻辑

首先看包含**UI属性**的cellmodel：

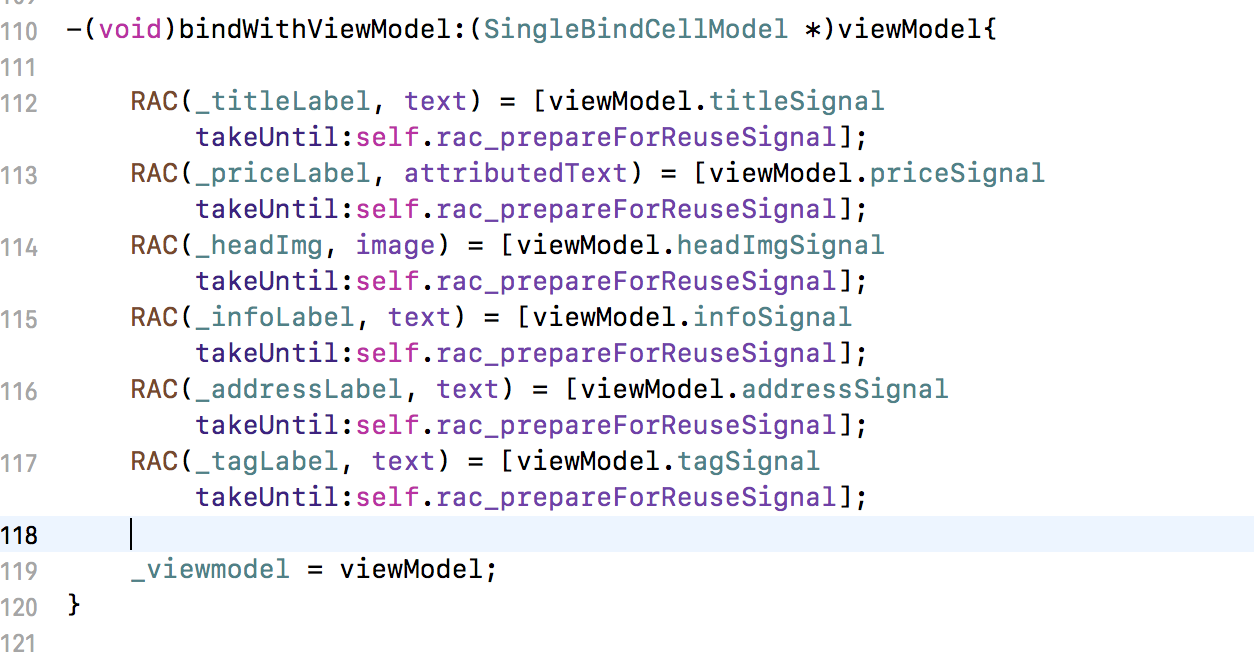


**.m这里面对数据进行了加工；渲染这部分可以放在cell中。**



不难看出，UI属性体现为一个或者多个信号，用于操作View上的对象。

下面将View于ViewModel进行绑定：

以上是对我们的cell和cellmodel（VM）的绑定和操作逻辑；接下来看看tableView和vm的绑定；

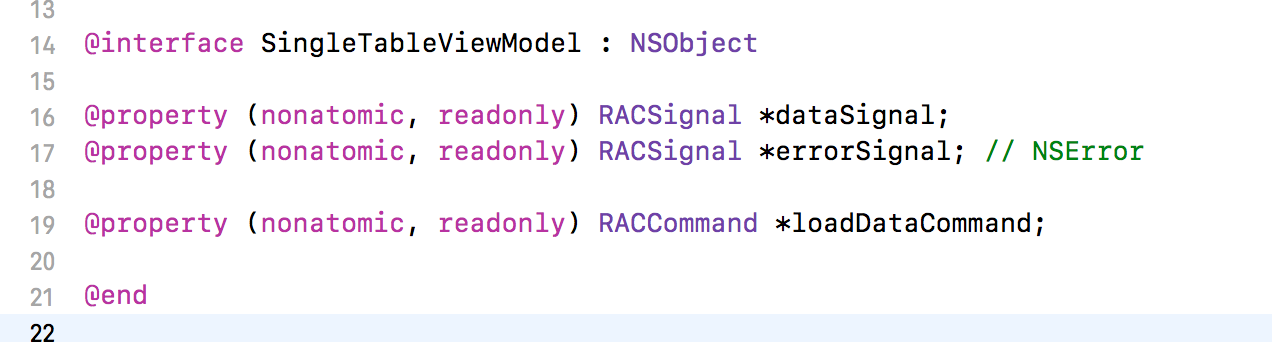
这部分会涉及到：

* 集合数据绑定
* 执行过程绑定
* 错误处理

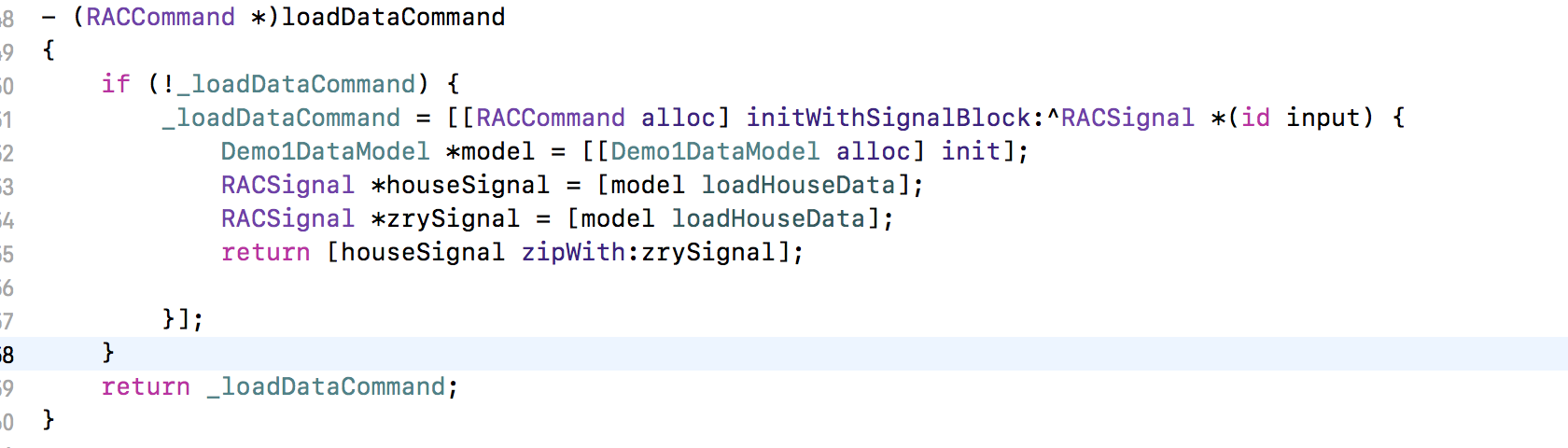
tableView的model层Demo1DataModel我们做了数据拉取的逻辑

-(RACSignal \*)loadHouseData； 并且构造了一个冷信号（需要订阅才可触发）。

TableView对应的ViewModel接口如下

其中dataSignal用于向View返回一个数组（集合），其内容为TableView的每一个Cell所对应的ViewModel，View需要根据数组中的ViewModel构造出TableView的Cell。这个过程就是所谓的“集合类型绑定”。当然，对于NSMutableArray，现实中可能还需要在数组内容发生变化时更新对应的Cell。本例中的代码采用了简单粗暴的方式，即不判断数组内容的变化，只处理数组整体的变化，在数组变化时reload整个tableView。

再看看loadDataCommand的实现



这里会分别请求两个数据源的数据（可以不同，这里为了方便请求的同一个数据源），让后将它们打包成一个信号，再返回。

接下来，需要实现dataSignal和errorSignal。很显然，它们可以通过loadDataCommand的executionSignals属性和errors属性变换而成。



errorSignal容易理解不比过多解释。这里重点讨论一下dataSignal。

首先解释一下flatternMap的含义：

假设有如下信号：

sourceSignal: RACSignal[ @1, @2, @3 ]

然后通过flatternMap得到新的信号：

newSignal = [sourceSignal flatternMap：^RACStream \*(id x) {

return [RACSignal return : @([x integerValue] \* 2)];

}]

则新信号的内容如下：

[ @2, @4, @6 ]

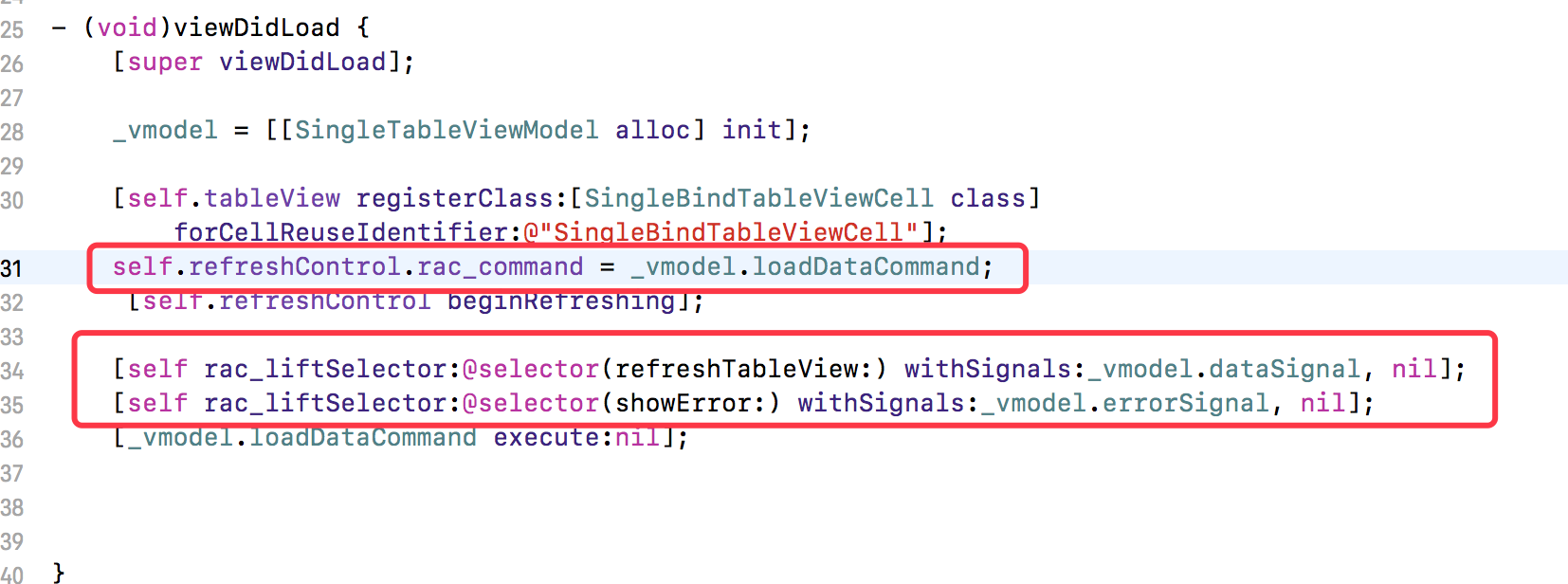
注意，新信号的内容并不是在flatternMap的闭包中返回的信号，而是信号的内容。因此，flatternMap非常有用，可以将高阶信号的内部信号展开，从而实现对高阶信号的降阶。

再回到dataSignal的逻辑中，我们介绍过executionSignals是一个高阶信号，里面每个值都是[houseSignal zipWith: zrySignal]。于是对这个信号进行映射，将其变换为房源和短租的ViewModel数组。

接下来VC中的逻辑就变得简单了；

将ViewModel的RACCommand与refreshControl进行绑定，绑定后在进行下拉刷新时会触发loadDataCommand的执行。

在dataSignal和errorSignal有next值的时候，触发refreshTableView:和showError:方法。



SingleBindModel就是我们的业务model。

**3：表单页面（Demo 2）**

本例将重点介绍Demo 1没有包含的**双向数据绑定**，通过一个表单例子给大家介绍清楚。本例运行效果如下：

[](https://github.com/liangshixing/mvvmdemo/blob/master/images/Demo2.png)

图5 Demo 2

在Demo2中定义的功能如下：

* 用户输入电话号码和密码时，输入框下面的label会同步展示输入的内容。
* 如果输入框中的内容不符合校验规则时，输入框中文本为红色；否则输入框中文本为黑色。
* 点击Clear按钮时，清空文本框。

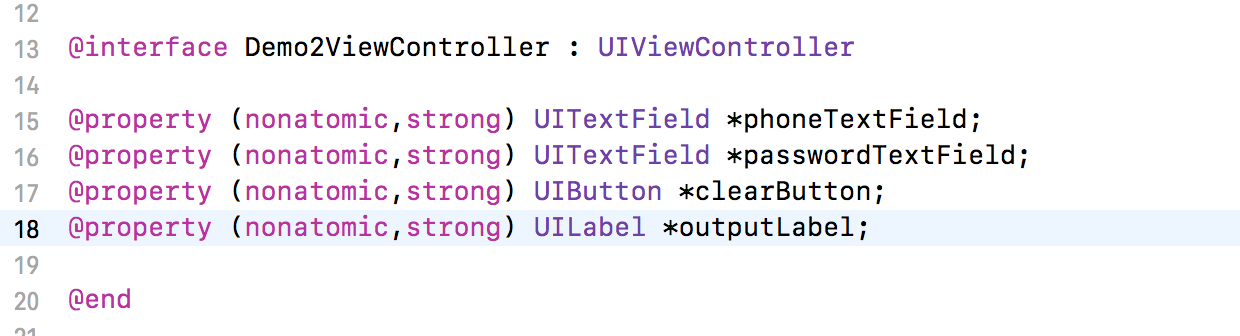
这里存在如下的双向绑定：

* 从输入框 -> ViewModel（ViewModel读取了输入框的内容）
* 从ViewModel -> 输入框 (设置初始值和清空输入框)

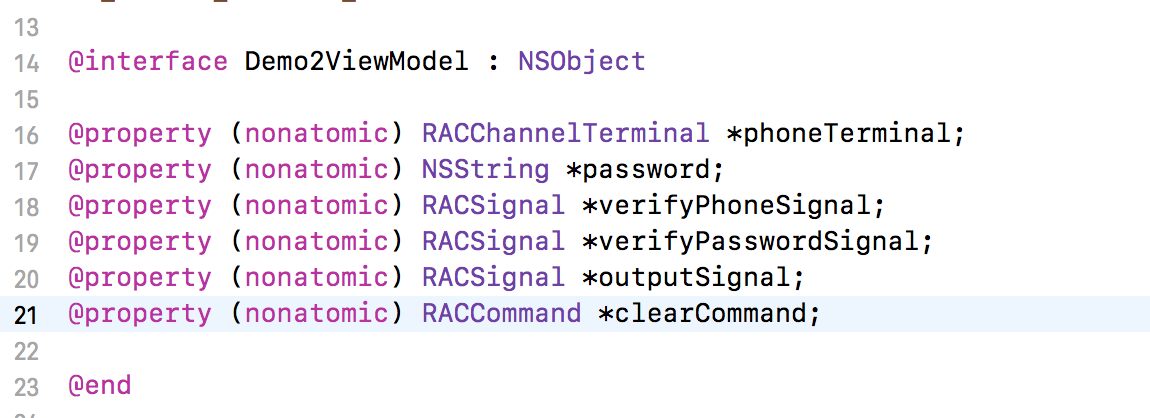
RAC提供了RACChannelTerminal，可以很好的支持双向绑定的需求。

废话少说，先看代码：

//UI部分，包含一些控件（两个文本框，一个标签，一个按钮）



显然，ViewModel有phone和password两个属性需要与UI进行双向绑定。这里提供了两种方式，一种是把RACChannelTerminal作为属性暴露出来，另一种方式是将原始的数据，如NSStirng\*暴露出来。为了体现这两种方式的异同，特意在同一个例子里提供了两种做法，可以根据各自喜好选择适合的实现。



接下来是ViewModel的实现逻辑：

@interface Demo2ViewModel ()

@property (nonatomic) NSString \*phone;

@end

@implementation Demo2ViewModel

- (instancetype)init

{

self = [super init];

if (self) {

\_phone = @"132";

\_password = @"";

\_phoneTerminal = RACChannelTo(self, phone); //①

\_outputSignal = [RACSignal combineLatest:@[RACObserve(self, phone), RACObserve(self, password)] reduce:^id (NSString \*phone, NSString \*password) {

return [NSString stringWithFormat:@"手机号：%@, 密码：%@", phone, password];

}];

\_verifyPhoneSignal = [RACObserve(self, phone) map:^id (NSString \*phone) {

NSString \*phoneRegexp = @"^1(3[0-9]|5[0-35-9]|8[0-25-9])\\d{8}$";

NSPredicate \*regextestmobile = [NSPredicate predicateWithFormat:@"SELF MATCHES %@", phoneRegexp];

return @((BOOL)[regextestmobile evaluateWithObject:phone]);

}];

\_verifyPasswordSignal = [RACObserve(self, password) map:^id (NSString \*password) {

return @((BOOL)([password length] >= 8));

}];

RACSignal \*verifyAllSignal = [RACSignal combineLatest:@[ \_verifyPhoneSignal, \_verifyPasswordSignal] reduce:^id (NSNumber \*phonePass, NSNumber \*passwordPass) {

return @([phonePass boolValue] && [passwordPass boolValue]);

}];

@weakify(self);

\_clearCommand = [[RACCommand alloc] initWithEnabled:verifyAllSignal signalBlock:^RACSignal \*(id input) {

@strongify(self);

self.phone = @"132";

self.password = @"";

return [RACSignal empty];

}];

}

return self;

}

@end

* ① 初始化了\_phoneTerminal
* \_verifyPhoneSignal和\_verifyPasswordSignal分别实现了电话号码和密码的校验逻辑。
* verifyAllSignal是基于\_verifyPhoneSignal和\_verifyPasswordSignal实现的，当电话号码和密码校验都通过，则校验通过。
* verifyAllSignal会决定\_clearCommand是否为enable，进而控制了按钮的可用状态。

ViewModel的实现逻辑简单又清晰，下面在ViewController中ViewModel进行绑定：

@implementation Demo2ViewController

- (void)viewDidLoad

{

[super viewDidLoad];

\_viewModel = [[Demo2ViewModel alloc] init];

RACChannelTerminal \*phoneTerminal = [\_phoneTextField rac\_newTextChannel];

[\_viewModel.phoneTerminal subscribe:phoneTerminal];

[phoneTerminal subscribe:\_viewModel.phoneTerminal]; // ①

RACChannelTerminal \*passwordTerminal = [\_passwordTextField rac\_newTextChannel];

RACChannelTerminal \*viewModelPasswordTerminal = RACChannelTo(\_viewModel, password);

[viewModelPasswordTerminal subscribe:passwordTerminal];

[passwordTerminal subscribe:viewModelPasswordTerminal];

// ②

RAC(\_outputLabel, text) = \_viewModel.outputSignal;

\_clearButton.rac\_command = \_viewModel.clearCommand; // 绑定执行过程

UIColor \*(^mapValidateToColor)(NSNumber \*) = ^(NSNumber \*pass) {

if (![pass boolValue]) {

return [UIColor redColor];

} else {

return [UIColor blackColor];

}

}; // ③

//根据校验结果更新输入框的颜色

RAC(\_passwordTextField, textColor) = [[\_viewModel.verifyPasswordSignal distinctUntilChanged]map:mapValidateToColor];

RAC(\_phoneTextField, textColor) = [[\_viewModel.verifyPhoneSignal distinctUntilChanged]map:mapValidateToColor];

}

@end

相比单向绑定，双向绑定看上去没有那么优雅，但也不麻烦。范式就是：

RACChannelTerminal\* aTerminal = xxxx;

RACChannelTerminal\* bTerminal = xxxx;

[aTerminal subscribe:bTerminal];

[bTerminal subscribe:aTerminal];

同时，RAC提供了一个好用的宏，可以帮助我们优雅的创建terminal：

RACChannelTerminal \*viewModelPasswordTerminal = RACChannelTo(\_viewModel, password);

此时，当另一个terminal与viewModelPasswordTerminal进行绑定，数据的变化会通过\_viewModel的passowrd属性反映出来。因此，当输入密码框的文本框内容发生变化时，会自动的更新\_viewModel的password属性；反过来，一旦修改\_viewModel的password属性值，文本框的内容也会发生改变。

本例在ViewModel的接口中提供了两种实现方法，一种是暴露Terminal属性，另一种直接暴露原始数据的属性。两种做法差异并不大，但我更推荐暴露Terminal的方法，这种做法使得原始数据属性不会直接的暴露出来，减少了代码出错的可能性。

**其他细节**

我们通过上面两个例子，覆盖了五种典型的绑定场景，理论上可以解决具体项目中遇到的绝大多数问题。接下来再讨论一些细节问题，帮助我们更好的实践MVVM。

**单元测试**

由于ViewModel是普通的Objective-C对象，对其进行单元测试会变得格外容易。另一方面，ViewModel承载了视图与业务逻辑的桥梁，可以体现出来从业务逻辑到UI呈现的细节信息，因而对ViewModel进行单元测试可以取得非常好的测试覆盖率，下面以Demo2的单元测试为例：

- (void)testVerifyPhone

{

Demo2ViewModel \*viewModel = [[Demo2ViewModel alloc] init];

RACChannel \*channel = [[RACChannel alloc] init];

[viewModel.phoneTerminal subscribe:channel.leadingTerminal];

[channel.leadingTerminal subscribe:viewModel.phoneTerminal];

[channel.followingTerminal sendNext:@"18612345678"]; // 模拟从文本框输入 18612345678

NSNumber \*verifyPhoneResult = [viewModel.verifyPhoneSignal first];

XCTAssertEqualObjects(verifyPhoneResult, @(YES));

[viewModel setValue:@"13810001000" forKey:@"phone"]; // 模拟ViewModel更新phone值为13810001000

XCTAssertEqualObjects([channel.followingTerminal first], @"13810001000"); // 检验文本框内容是否为13810001000

viewModel.password = @"1234";

NSNumber \*verifyPasswordResult = [viewModel.verifyPasswordSignal first];

XCTAssertEqualObjects(verifyPasswordResult, @(NO)); // 测试密码校验逻辑，少于8位，失败。

}

- (void)testClearCommand

{

Demo2ViewModel \*viewModel = [[Demo2ViewModel alloc] init];

[viewModel setValue:@"18612345678" forKey:@"phone"];

XCTAssertEqualObjects([viewModel valueForKey:@"phone"], @"18612345678");

[viewModel setValue:@"passw0rd" forKey:@"password"];

XCTAssertEqualObjects([viewModel valueForKey:@"password"], @"passw0rd");

[viewModel.clearCommand execute:nil];

XCTAssertEqualObjects([viewModel valueForKey:@"phone"], @"186");

XCTAssertEqualObjects([viewModel valueForKey:@"password"], @"");

}

只需要简单的几行代码，就可以对ViewModel的电话号码校验逻辑，以及ClearButton点击逻辑进行了测试。

**扩展阅读**

ReactiveCocoa为Objective-C插上了FRP的翅膀，为在iOS中实现MVVM模式起到了至关重要的作用。因此强烈建议多花些时间和精力去加深对RAC的学习和理解，这里推荐美团技术博客上的几篇文章：

* <http://tech.meituan.com/talk-about-reactivecocoas-cold-signal-and-hot-signal-part-1.html>
* <http://tech.meituan.com/talk-about-reactivecocoas-cold-signal-and-hot-signal-part-2.html>
* [http://tech.meituan.com/talk-about-reactivecocoas-cold-signal-and-hot-signal-part-3.html](http://tech.meituan.com/talk-about-reactivecocoas-cold-signal-and-hot-signal-part-1.html)
* <http://tech.meituan.com/potential-memory-leak-in-reactivecocoa.html>