前言

不知不觉，笔者也撸码也已经一年多了。随着撸码的数量疾速上涨，如何高效，简单的组织代码，经常引起笔者的思考。作为一个方法论及其实践者（这个定义是笔者自己胡诌的），始终希望能够找到一些简单、有效的方法来解决问题，由此，也开始了一段构建代码的实践体验。

这次要分享的，是自己在长期实践 MVVM 结构后，对 MVVM 框架的一些理解与自己的工作流程。其中或许还有一些地方拿捏欠妥，希望大家能一起相互交流。

前戏

ViewModel 这个概念是基于 MVVM 结构提出的，全称应该叫做 Model-View-ViewModel，从结构上来说，应该是 Model-ViewModel-ViewController-View。简单来说，就是在 MVC 结构的基础上，将 ViewController 中数据相关的职能剥离出来，单独形成一个结构层级。

关于 ViewModel 的详细定义，可以参考这篇 MVVM介绍。

此外，在工作流中，笔者在一定程度上参考了 BDD 的代码构建思路，虽然没有真正意义上的按照行为构建测试代码，但是其书写过程与 BDD 确实有相似之处。关于 BDD，可以参考这篇 行为驱动测试。

为本篇文章所编写的 Demo 已经传至 Github：传送门~

好吧，我们开始。

ViewModel 与 ViewController

基类

嗯，在这里，需要用到 OOP 的经典模式 —— 继承。

我们不打算把 ViewModel 的功能构建的太重，所以，它只需要一个指向拥有自己的 ViewController 指针，与一个赋值 ViewController 的工厂方法。 就像下面这段代码：

```
//BCBaseViewModel.h

@interface BCBaseViewModel : NSObject

@property (nonatomic,weak,readonly) UIViewController *viewController;

+(BCBaseViewModel *)modelWithViewController:(UIViewController *)viewController;

@end
```

ViewModel 只需要一个 weak 类型的 viewController 指针指向自己的 viewController，而 viewModel 则由 viewController 使用 strong 指针持有，用于规避循环引用。

这样，就足够了。

委托者与代理者

为了让 ViewModel 与 ViewController 的关系更加清晰，也为了能够批量化的生产 ViewModel，接下来要定义的，就是 ViewModel 与 ViewController 的结构特征了。

在分析了 ViewModel 划分层次的原因与主要承担的功能之后，我们大致可以总结出这么几个特征：

- ViewModel 与 ViewController 是一一对应的
- ViewModel 实现的功能是从 ViewController 中剥离出来的
- ViewModel 是 ViewController 的附属对象

根据上面几点特征，最容易想到的类间关系应该就是代理/委托关系了，把一眼就看到的关系说的复杂可能会招骂，但是对接下来的论述，上面多多少少会起到点决定性的作用。

比如，虽然确定了代理与委托，但究竟谁是代理者，谁是委托者呢？换句话说，谁是协议的制定方，而谁又是实现方呢？

笔者这里给出两个依据来确认。

1. 协议方法是被动调用方法，也就是反向调用。基于此，协议的实现方，应该同时是事件的响应方，以事件驱动正向调用，再由此触发反向调用。
2. 协议的实现方实现的方法是通行，且抽象的。反推之，协议的制定方需要实现更难抽象或是更为具体的方法。这个依据也可以从另外一个层面来理解，即协议的实现方的可替换性应该更强。

第一条依据相对毋庸置疑，毕竟 ViewController 是 View 的持有者与管理者，更是 View 与 ViewModel 相互影响的唯一渠道。让 ViewModel 作为 View 事件的响应方来驱动 UIViewController，从结构上有些说不通。

第二条则是实践得来的结论，在实际开发时，由外而内，视图的修改频度往往是大于数据的。因此，重构 ViewController 的概率也要大于重构 ViewModel 的概率。不过这种归纳性的结论无法一言蔽之，反而会建议诸位在实际的开发过程当中，应当针对这些开发诉求对结构做更灵活的调整和优化。

这次实践，则会以 ViewModel 作为协议的制定方，来构建代码。

让协议轻一点

在 OC 中，有 @protocol 相关的一系列语法专门用于声明与实现协议相关的所有功能。但是考虑到具体的 ViewModel 与 ViewController 之间的相互调用都各不相同，如果我们为每一组 ViewModel 与 ViewController 都声明一份协议，并且交由彼此实现和调用，代码量激增基本上是一种必然了。

为了让整个协议结构轻一点，这里并没有采用 @protocol 相关语法来实现，而是利用如下代码：

```
typedef NSUInteger BCViewControllerCallBackAction;

@interface UIViewController(ViewModel)

-(void)callBackAction:(BCViewControllerCallBackAction)action info:(id)info;

@end
```

这段代码做了这么几件事：

1. 利用分类，为 UIViewController 拓展了 ViewModel 相关的回调方法声明。功能类似于父类声明抽象接口，而交由子类去实现。
2. 接口支持传参，具体的类不再制定协议方法，而只需要协议参数。
3. 将该分类声明在 ViewModel 的基类中，即可保证对 ViewModel 可见的 UIViewController 都实现了协议方法，从而不需要再编写 @protocol 段落。

在具体的 ViewModel 与 ViewController 子类中，只需要根据具体的需求设计回调参数，构建一个 对应的枚举即可。

将整个协议结构轻质化，主要的原因是因为协议内容变动频繁。使用枚举而非 protocol，可以减小改动范围，且代码量较少，定制方便。

笔者曾经也尝试过双向抽象方法定义，即对 ViewModel 也做一些抽象方法，使双方仅根据基类约定的协议工作。但实践下来，ViewModel 的方法并不易于抽象，因为其公共方法往往直接体现了 ViewController 的数据需求。如果强行拟订抽象方法，反而会在构建具体类时产生归纳困惑，由此产生的最坏结果就是放弃遵守协议，整个代码反而会变的难以维护。

化需求为行为

在开发过程当中，最常见的开发流还是需求驱动型开发流。说白了，就是扔给你一张示意图，有时运气好点还有交互原型神马的（运气不好就是别人家的 App = =），然后就交由你任性的东一榔头西一棒槌的写写画画。

这个时候，还是建议适当的规划一下开发流程。主要是考虑这么几点：

- 开发层级与顺序；
- 单位时间内只关心尽可能少的东西；
- 易于构建和调试；
- 合理简省重复性工作。

其实说简单点，就是让整个工作流变的有规则和秩序，以确保开发有理有据且可控。另外，也能有效避免反错的频率和严重程度。

这里，笔者不要脸的分享自己的简易工作流。

整个过程并不复杂，其实就是先撸 ViewController 界面，遇到需要数据的地方，就在 ViewModel 中声明一个方法，然后伴装调用。撸的代码大概是这个样子的：

```
typedef NS_ENUM(BCViewControllerAction,BCTopViewCallBackAction){
    BCTopViewCallBackActionReloadTable = 1 < < 0,
    BCTopViewCallBackActionReloadResult = 1 << 1
};

@interface BCTopViewModel : BCBaseViewModel

- (NSString *)LEDString;

- (NSUInteger)operationCount;

- (NSString *)operationTextAtIndex:(NSUInteger)index;

- (void)undo;

- (void)clear;

@end

@interface BCTopViewController ()@property (nonatomic,strong) BCTopViewModel *model;
@property (nonatomic,weak) IBOutlet UITableView *operationTable;
@property (nonatomic,weak) IBOutlet UILabel *result;
@end


@implementation BCTopViewController

- (void)viewDidLoad{
    [super viewDidLoad];
    self.operationTable.tableFooterView = UIView.new;

}

#pragma mark - action

- (IBAction)undo:(UIButton *)sender{
    [self.model undo];
}

- (IBAction)clear:(UIButton *)sender{
    [self.model clear];
}

#pragma mark - call back

- (void)callBackAction:(BCViewControllerAction)action{
    if (action & BCTopViewCallBackActionReloadTable) {
        [self.operationTable reloadData];
    }
    if (action & BCTopViewCallBackActionReloadResult) {
        self.result.text = self.model.LEDString;
    }
}

#pragma mark - tableView datasource & delegate

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)sectic
    return self.model.operationCount;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexF
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"cell" forInc
    cell.textLabel.text = [self.model operationTextAtIndex:indexPath.row];
    return cell;
}
```

```
66
67  @end
```

这样开发利用了一个 Runtime trick，那就是 Nil 可以响应任何消息。

所以，虽然我们只声明了方法，并没有实现，上面的代码也是随时可以运行的。换言之，你可以随时运行来调试界面，而不用担心 ViewModel 的实现。

相对麻烦的是测试回调方法，笔者自己的建议是在编写好回调方法之后，在 ViewController 中对应的 ViewModel正向调用之后直接调用自己的回调，如果遇到可能的网络请求或者需要延时处理的回调，也可以考虑编写一个基于 dispatch_after 的测试宏来测试回调。

一般来说，视图界面层的开发总是所见即所得的，所以测试标准就是页面需求本身。当肉眼可见的所有需求实现，我们的界面编写也就告一段落了。当然了，此时的代码依旧是脆弱的，因为我们只做了正向实现，还没有做边界用例测试，所以并不知道在非正常情况下，是否会出现什么诡异的事情。

不过值得庆幸的是，我们已经成功的把 ViewController 中数据相关的部分成功的隔离了出去。在未来的测试中，发现的任何与数据相关的 BUG，我们都可以拍着胸脯说，它肯定和 ViewController 无关。

另外，一如我所说，需求本身就是页面的测试标准。也就是说，当你实现了需求，你的视图层就已经通过了测试。是的，我要开始套用 TDD 的思考方式了。我们已经拿着需求当了测试用例，并且一一 Pass。

而当我们开发完 ViewController 的同时，我们也已经为 ViewModel 声明好了所有公共方法，并且在对应的位置做了调用。BDD 的要点在于 It...when...should 的行为断言，在此时的环境下，It 就是 ViewModel，when 就是 ViewController 中的每次调用，而 should ，则对应着 ViewModel 所有数据接口所衍生出的变化。

换句话说，我们可能没办法从界面上看到所有的行为引发的变化，但是我们已经在 ViewModel 实现之前构建了一个可测试环境。如果时间充足的话，此时的第一件事应当是根据具体的调用环境，为每个公共方法编写足够强壮的测试代码，来避免数据错误。

顺便说几句风月场上的虚话。在构建程序的时候，面向接口是优于面向实现的，因为在任何一个系统中，比起信息的产生，信息的传递更决定着系统本身是否强大。而编写代码的时候，先将抽象功能方法具体化，再将数据逐步抽象化，经历一个类似梭型的过程，可以更完美的贴合“高内聚、低耦合”的目标。

Fat Model

如果单从 ViewModel 实践上来说，以上的内容已然解释的差不多了。不过鉴于笔者手贱撸了一整个 Demo，就额外解释下其它几个地方的设计了。

首先是关于胖 Model 的设计。关于胖瘦 Model 的概念笔者也是最近才从这篇 iOS 应用架构谈 view层的组织和调用方案 上看到。在此之前，只是凭直觉和朋友讨论过 Model 与 Model 之间也应该有所区分。

Model 的胖瘦是根据业务相关性来划分的。所以，笔者有时会直接将胖 Model 称之为业务层 Model 以区分瘦 Model。在示例代码中，CalculatorBrain 应该算是一个相对标准的业务层 Model 了。

如果遇到单个 ViewModel（或者 MVC 中的 Controller）无法解决的需求时，就需要整体业务下沉，交给一个相对独立的 Model 来解决问题。上层只持有该 Model 开放出来的接口，以此促成的业务层 Model，带有明显的业务痕迹，说白了，就是不容易复用。

不过，笔者自己的开发观点是，弱业务相关的模块复用性应该强，即功能应该尽量单元化。而强业务相关的模块则应该有更好的重构性和替换性能，即尽可能的功能内聚。说简单点，比如这个 Demo 不再是一个计算器，而需要变成一个计数器或者别的什么，需要重构的就只有 CalculatorBrain 这个类。（当然，这只是基于假设，界面不变底层数据狂变的需求不敢想象...）

从另外一方面来看，在整个 MVVM 框架中，也可以将每个单独的 ViewModel 视作一个管道。在整个业务链中做了双向的抽象，使整个业务链各个部分的替换性都有所提升，笔者个人倾向于将其解释为，通过设计中间层，均衡了上下层级的复杂度。

更轻量级的 ViewController

objccn.io 第一期的第一篇文章就是更轻量的 View Controllers。文章内曾提到，通过将各个 protocol 的实现挪到 ViewController 之外，来为 ViewController 瘦身。

笔者也曾是这个建议的实践者之一，甚至一度认为这也是 ViewModel 的主要功能。不过随着开发时间拉长，笔者不得不重新开始审视这个问题。

首先，这种方法会产生很多额外的接口。我们依旧用 UITableView 来举例。 假设我们让 ViewModel 实现了 UITableViewDelegate 与 UITableViewDataSource 协议。这个时候，如果 ViewController 的另一个控件想要根据 tableView 的滚动位置做出响应该怎么办呢？由于 ViewModel 才是 tableView 的 delegate，所以我们就需要为 ViewController 声明额外的公共方法，供 ViewModel 在回调方法中调用。

而我们不难发现，基本所有视图控件的 Delegate 协议都涉及到视图本身的响应，只要涉及到同界面下不同控件元素的互动，就不可避免的需要 ViewController 的参与。

笔者也尝试过将 UITableViewDelegate 实现在 ViewController 中，而把 UITableViewDataSource 托付给 ViewModel 的方式。蛋疼的事情发生在动态高度 Cell 的实现上，我们一方面在 ViewModel 内部给 tableView:cellForRowAtIndexPath 输入数据，一方面却又要为 tableView:heightForRowAtIndexPath: 开设接口提供相同的数据以供计算高度。

笔者最后总结了原因，是因为 View 层 与 ViewController 层本身是持有与被持有的依赖关系，所以任何类作为 ViewController 的类内实例来实现协议回调，实际上都是在跨层调用，所以，就注定要以额外的接口为代价，换言之，ViewController 的内聚性变差了。

而另外一方面的原因，则是关于测试。我们说 ViewController 难以测试的原因是因为在大部分情况下，它并没有几个像样的公共方法，且私有方法中还有一大部分方法是传参回调。如果我们将这些 protocol 实现在另一个类中，其实并不会提升它们的可测试性。更为行之有效的方式，应该是将 protocol 的实现与数据接口隔离开来，让实现方通过接口来填充数据，而非自身。

在 Demo 中，TopViewModel 便为 cell 的内容填充开设了 operationCount 与 operationTextAtIndex: 这样的数据接口。相信，为这样的数据接口构造测试环境，要比为 tableView:cellForRowAtIndexPath 这种方法构造测试环境要简单的多。从侧面来看，这样的接口反而更合适于测试覆盖。

基于以上这两点原因，在之后的开发中，笔者开始将越来越多的 protocol 又请回了 ViewController 中。并且，由于 ViewModel 的存在，笔者更倾向与将 ViewController 构建成为一个独立实现且只负责实现界面布局、逻辑的类，让一个类做更少的事，但做的更好。

后记

本文的相关 Demo，实现的功能并不复杂，甚至有些简陋的不好见人。见责于笔者想象力不周，本着以实践演示为主的心态，做个参考就好吧。

笔者自诩为方法论及其实践者，比较认同“构建代码的方法比代码更有价值”这个观点。写出一两句惊艳的代码或许是运气，掌握方法去构建代码本身才是战斗力吧。尽可能让自己每一句代码都有理有据，而不是随心所欲，也觉得会比较负责，起码写起来有个交待。

以上的总结见识有限，很多地方或许会有疏漏之处，希望能与诸位看官一起交流，如果能指出其中疏漏甚至错误的观点，那就不甚感激了。

另外，说点说出来就不嫌丢人的话。截至笔者写完这篇博文，虽然对“设计模式”的相关概念有各种旁敲侧击的求证与查询，但仍未系统学习过相关概念。说来惭愧，有时候自己花好大功夫才弄明白、想清楚的答案，突然发现某本书、某篇文章上早已几句话讲的明明白白，其实还挺挫败的。次数多了，甚至会对未知的知识产生抗拒，用来安慰自己很牛逼，这也是特地声明没有系统学习的原因吧。

不过开发路漫漫，其实大家都知道，我们只不过是爬到巨人肩上的搬砖工人而已。回头看看自己脚下的路，每一块砖都足以让自己自惭形秽，自欺欺人什么的，也只不过是浮躁上头，丢人现眼罢了。