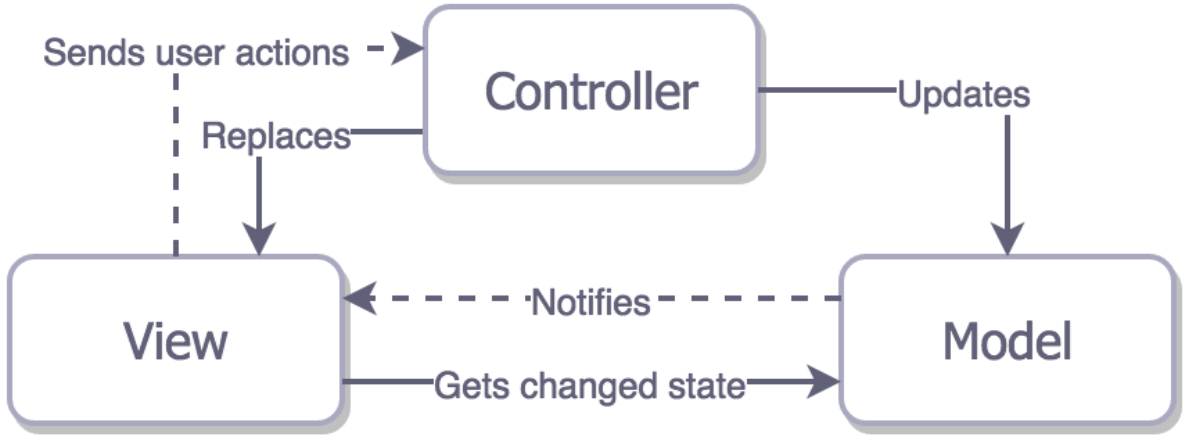
## iOS开发架构的探索——不再是MVC

MVC这个词，大家想必对它再熟悉不过了，尤其是开发iOS App的童鞋们。不过，大家都知道MVC可以解读为Model-View-Controller，大家可还知道它还可以被解读为Massive ViewController（重量级视图控制器）？让我们先从这个梗的来历说起。

MVC架构是苹果官方推荐使用的一种架构，而苹果也在它的语言中体现了这一点——精心为你准备的UIViewController、UIView类，你直接拿来用就可以了。这个架构应用广泛、便于理解、易于使用，而且它是Apple推荐使用的喂！然而这并不能说明它就是一个很棒的开发架构，这要从MVC的分工说起。




M(Model)——程序中要操纵的实际对象的抽象，包含了代表着实际对象属性(例如汽车的颜色)的属性(property)和操作这些属性的方法(method)，在MVC的相互协作中，View会通过Controller来向Model索要数据，经由Controller转换之后展示到View上，同时会将用户操作通过Controller反馈到Model中，更新Model的内容。而至于如何获取需要的数据，以及如何处理用户发出的数据请求，这些通常都定义在Model中。

V(View)——在标准的MVC模式中，View通常又被叫做“哑掉的View(Dumb View, 我自己翻译的＝＝)”。也就是说View只负责机械地展示来自Model的数据，并且将用户的操作反馈到Controller中，自己并不参与到整个数据的处理过程当中。在iOS开发中，标签、文本框、表单、图片、滑动页等等这些都属于View这一类。

C(Controller)——可以说Controller是MVC中最重要的一部分，

Controller负责从Model获取数据(Controller不直接持有Model，而是通过KVC或KVO获取数据)，经过处理(将raw data处理为可以直接展示的数据形式，这个处理过程叫做表示逻辑，presentation logic)后交给View(Controller直接持有View)展示；同时，Controller负责对用户在View上的操作进行响应，并相应地更新Model。

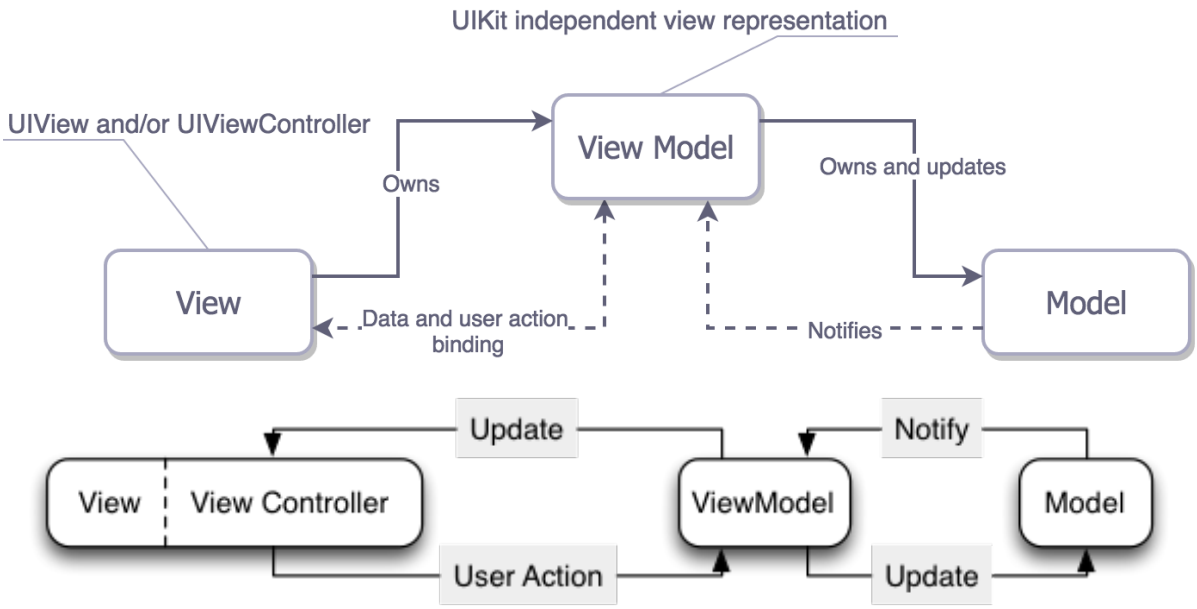
我们可以看到Controller的功能何其强大！它做的事情实在太多了，导致在App不断扩展的时候，Controller的体积也变得越来越大。在一个小型App(比如天气预报)里可能看不出来，但是对于一个大型App来说，上万行的Controller是常有的事。如果一个模块做了太多的事情，你就要考虑一下你做的是否正确了。另外，由于Model、View、Controller三个部分之间的耦合非常紧密(tight coupling)，导致对presentation logic的单元测试(Unit Test)几乎是不可行的——你只要实例化Controller就必须要同时实例化一堆与之相关联的View。从这几点我们可以看出，对于开发大型App来说，MVC已经不是一个合适的开发架构了。当然，如果你开发的是一个小型App，MVC仍然是你最好的选择，因为它够快、够简单。下面，我们来探索一些更佳的开发架构。

1、MV*架构

这种类型的开发架构实际上是在原有的Controller上下功夫，寻找Controller更佳的替代品。这种开发架构的代表是MVP(Model-View-Presenter)和MVVM(Model-View-ViewModel)，在这里我只讲MVVM，因为MVP和MVVM并没有什么本质上的区别。

MVVM(Model-View-ViewModel)

之所以选择讲MVVM，是因为MVVM是MV*系列里目前来说最出色的一种架构，而且它很能代表MV*的思想，以至于其他MV*架构(比如MVP)可以表示成MVVM的一个简单的变体。现在我们就来看一下这个架构的强大之处，我们仍然从各个模块的分工讲起。




M(Model)——Model部分和MVC中的Model没有什么区别。

V(View)——注意！在MVVM中，View不再是UIView的子类，而变成了UIViewController的子类。因此我们可以在上图中看到，View实际是和ViewController绑定在一起的。这种View实际上就是MVC中剥离了处理presentation logic部分的Controller——它仍然有各种UIView*的属性(例如UILabel、UITextField等等)、仍然有ViewController生命周期的各种方法(因此View部分负责将视图展示出来，也负责响应用户的操作)，但是它不知道该展示些什么数据(实际上MVC中的Controller也是不知道的)，它也不知道该用什么方法处理并展示来自Model的数据(这个是MVC中的Controller知道的)。少做了一大部分工作之后，View(这里也是Controller)终于不再臃肿了。

VM(ViewModel)——在MVVM中，扮演协调者(Interactor)角色的不再是Controller，而变成了ViewModel。ViewModel被View持有，同时也持有着Model。ViewModel中定义了如何从Model获取数据、如何更新Model、如何处理用户的数据请求以及何时以何种方式更新View等等的众多方法，可以说是MVC中Controller的一个精简版。

在MVVM中，除了ViewModel比Controller做了更少的事情之外，各个模块之间的耦合也没有MVC里那么紧密的耦合。在MVVM中，View持有ViewModel，ViewModel持有Model。但是ViewModel没有反过来持有View，Model也没有反过来持有ViewModel。那么这两个方向的通信

如何进行呢？更直接一点来说，我如何能让ViewModel告诉View它该更新了(以及以何种方式更新)，又如何能让Model告诉ViewModel它(Model)发生了变化呢？对于后者，我们仍然可以采用KVC或者KVO的方式来进行通信。而对于前者，我们推荐使用一种通过发射信号(Signal)的方式来进行通信。在这方面有许多第三方库做得非常好，其中应用最为广泛的当属ReactiveCocoa。ReactiveCocoa是一个非常棒的通信框架，但我这里不打算详细介绍，请大家自行Google使用方法。

另外，由于获取数据的部分改由ViewModel来做，不再和View绑定(在MVC中是由和View绑定的Controller做的)，因此单元测试变得十分容易，现在我们可以单独地测试对数据进行操作的部分，而不用担心会实例化一堆View了。

2、VIPER(View-Interactor-Presenter-Entity-Router)

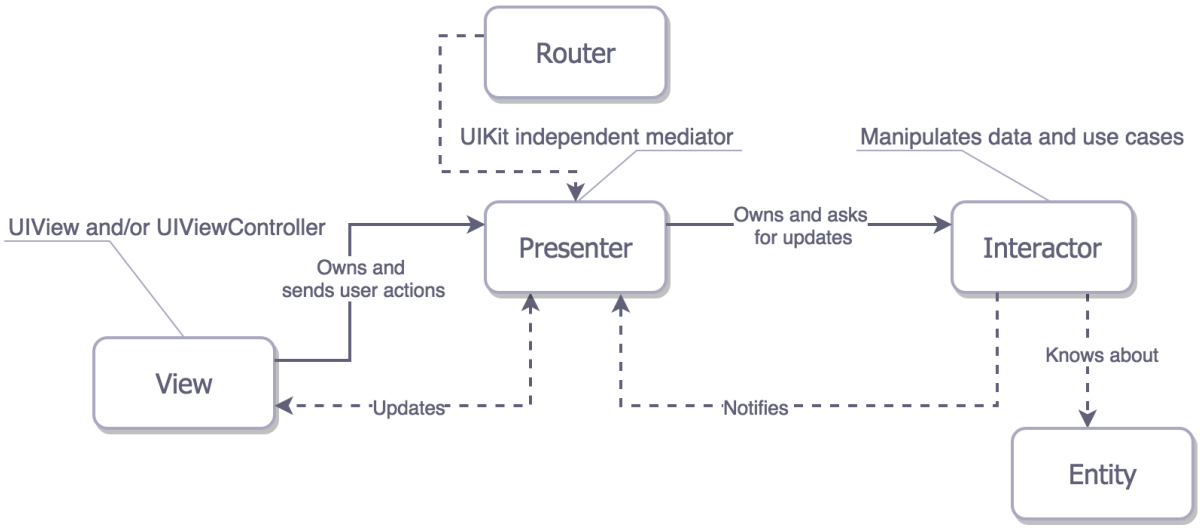
VIPER是一种很有意思的架构，因为它把要处理的职责划分成了五层。实际上VIPER是MVVM的一个更细致的划分，让各个模块有了更清晰单一的分工。

在详细介绍VIPER前，我需要先来介绍一下用例(Use Case)的概念。

“Apps are often implemented as a set of use cases. Use cases are also known as acceptance criteria, or behaviors, and describe what an app is meant to do. Maybe a list needs to be sortable by date, type, or name. That’s a use case. A use case is the layer of an application that is responsible for business logic. Use cases should be independent from the user interface implementation of them. They should also be small and well-defined.”——Jeff Gilbert & Conrad Stoll

通俗来讲，用例就是指用户会用你的程序做的事情，一个事情成为一个用例。比如我点击了这个按钮，程序会有什么样的反应，这就是一个用例。

介绍完用例的概念，我们再来看看VIPER中各个模块的分工：




View：和MVVM中的View一样，它也是UIViewController的子类，它仍然负责将各种UI组件展示到屏幕上。但是与MVVM不同的是，它现在不负责响应用户对UI的操作了，这个响应部分现在被移到了展示器(Presenter)里。

Interactor(交互器)：交互器负责处理用例中规定好的逻辑，它负责获取并处理数据，并将数据送给展示器(Presenter)。

Persenter(展示器)：展示器负责处理presentation logic，还负责响应各种用户事件(比如按钮的点击)。

Entity(实体)：实体仅仅是一个数据结构的定义，它定义了程序要处理的对象应该具有哪些属性，而没有定义处理这些数据的方法(这一点和MVC、MVVM中的Model不同)。用过Core Data的同学会更容易理解这个东西。

Router(路由)：路由负责处理各种转场(Transition)逻辑，也就是用于实现导航功能。以前写在Controller里的presentViewController、performSegue等方法就可以放在这里来实现了。

通过对VIPER中各个模块分工的介绍我们可以发现，VIPER实际上是对MVVM的各个部分进行了进一步的细化，但又不仅仅是对MVVM进行了更细致的分割：View和Presenter是对MVVM中的View的进一步细化，同时Presenter又包揽了一部分ViewModel的工作(例如presentation logic的处理)；I和E是对Model的细化，但Interactor又包含了一部分ViewModel的工作(例如根据用户的操作来更新Model/Entity)；Router则又分担了View的一部分工作。所以VIPER相比MVVM是更为细致的划

分，同时两者的各个模块又有着功能上的交叉。

另外，在VIPER中，Presenter对View的通信仍然可以使用ReactiveCocoa来进行，因为这个方向上的通信和MVVM实际上是一致的，都是在刷新UI。

讲到这里，我们这次对iOS开发架构的探索就告一段落啦～我虽然是以iOS应用为例来讲解这些架构的，但实际上这些架构可以用在其他各种客户端的开发上。Any idea or suggestion is welcome~

推荐文章：

Architecture：

https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52#.52hd9676g

MVVM：

https://www.objc.io/issues/13-architecture/mvvm/

https://www.raywenderlich.com/74106/mvvm-tutorial-with-reactivecocoa-part-1

http://www.teehanlax.com/blog/model-view-viewmodel-for-ios/

VIPER：

https://www.objc.io/issues/13-architecture/viper/

By Caesar