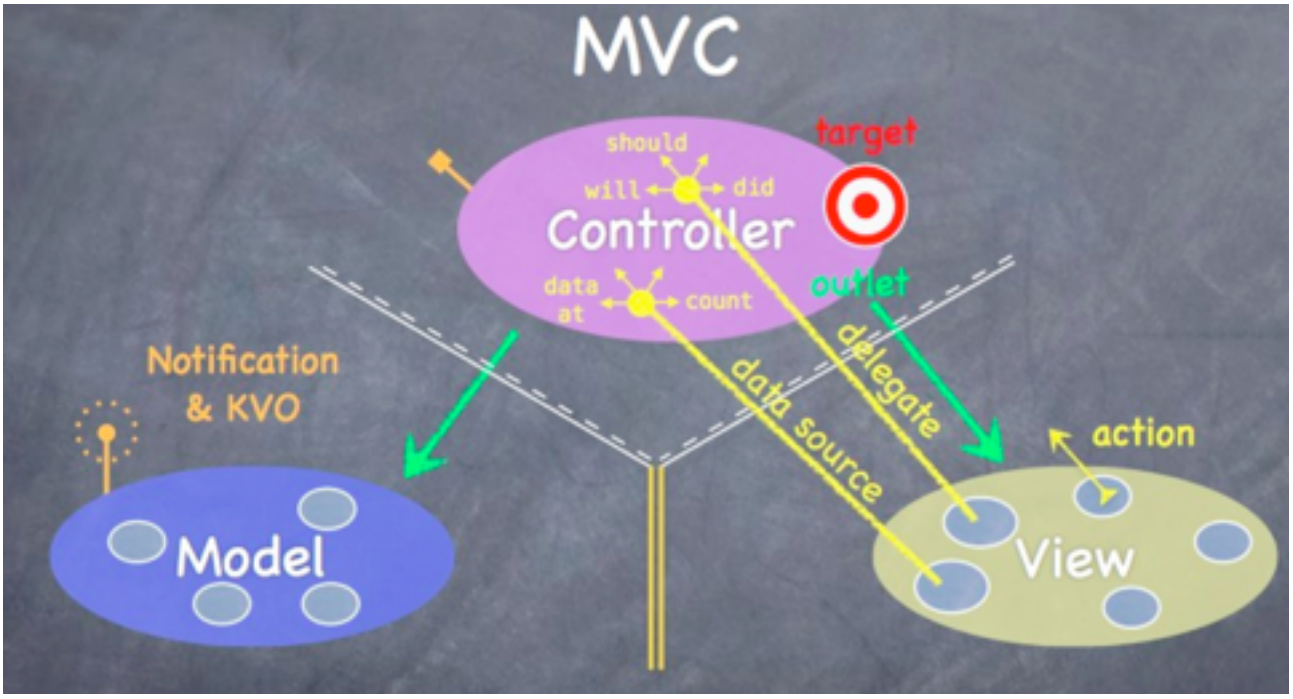
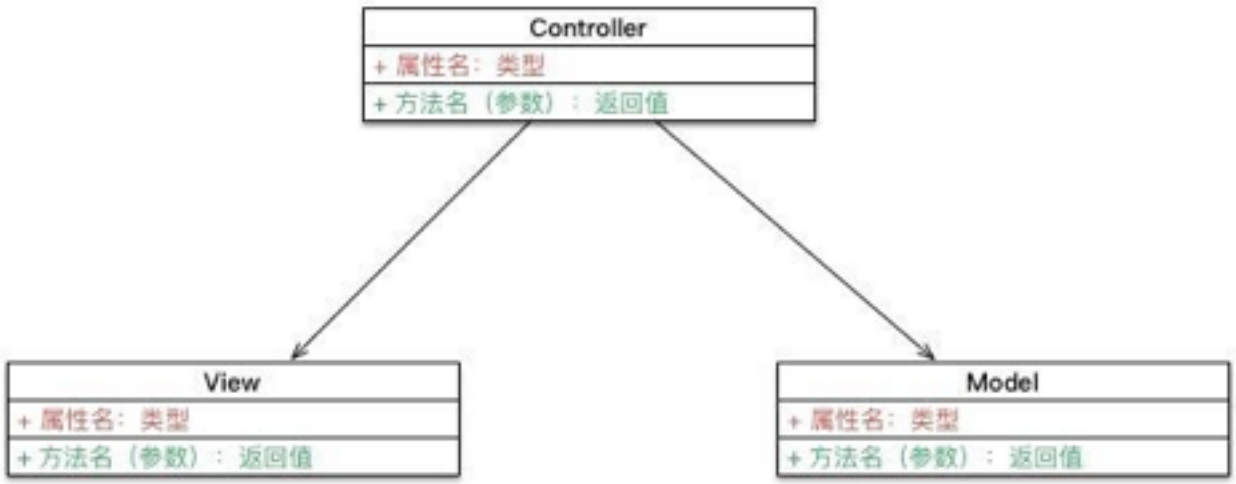
iOS架构的认识过程

MVC

经典就是经典，没有之一。iOS中MVC架构，看懂斯坦福大学白胡子老头这张图基本上就可以了



简单理解，就是Controller对象拥有View和Model对象，两者通过Controller进行沟通。对于单个页面，三个类就搞定了，感觉很简单。



网络连接应该放在哪里？Model中吗？感觉很有道理？实际上，很多的网络连接的发起和接收后的处理都放在了Controller中，因为方便嘛。Model一般只有属性定义，没有实现。

View应该是独立一块了吧？实际上呢，View大多都放在了Controller中，有个loadView函数，很方便啊。有几个人会单独写个类来作为view?

本来，xib和Storyboard是很好的分离view的方式。但是，由于“不合适多人合作，版本管理”，非要代码写界面，还振振有词：“性能高，对培养新人有好处”。“谎言说100次都能成真话”，何况这些理由听上去还那么有理。

像“检查用户名是否合法，检查密码对不对”应该放在哪里呢？有几个人会像斯坦福大学白胡子老头那样新起一个类来写？基本上都是Controller中搞定。

BaseController，BaseView，BaseModel一定见过不少吧？有的还有好几层呢

公共View，各种名字带common或者类似的类常见吧？里面网络连接，数据库，逻辑等等往往比View本身很多，俨然一个小模块了，功能比Controller都强大了。这还是view吗?

本来MVC理论上是最简单的架构，但是实际结果呢，变成了最难懂的架构。Controller成了上帝类，什么都干。“只知道那一坨东西有用，但看不出那是简单的MVC”。

MVC也被称之为 Massive View Controller（重量级视图控制器）。其实这不是MVC的错，只是没有程序员承认自己懒惰，编程习惯不好罢了。如果能够像斯坦福大学白胡子老爷爷那样好的编程习惯，那么大部分的iOS程序都能有清晰的MVC架构。

MVVM

认识MVVM的起点是@objc上文章MVVM 介绍

MVVM来自MVC，一张经典的图就是下面这张，在好多文章中看到过。

“稍微考虑一下，虽然 View 和 View Controller 是技术上不同的组件，但它们几乎总是手牵手在一起，成对的。你什么时候看到一个 View 能够与不同 View Controller 配对？或者反过来？所以，为什么不正规化它们的连接呢？” ------ 这段话当时给我的印象很深刻，这个观点到现在我都认可。

Controller代表了一个场景（Scene）的生命周期，是一个调度者。什么都是，因为什么都离不开它。又好像什么都不是，因为它代表不了任何具体的东西。让它和View合在一起，作为广义的view就有了具体的意义，并限制了它无所不能的印象。这点值得肯定。

“显示逻辑（presentation logic）”可以从Controller中移到ViewModel中，从而给Controller减负。这个观点我也是支持的。并且我以此认为ViewModel就是用来做“显示逻辑”的，一个页面一个，随页面而变化。在Swift中，我用结构体来做ViewModel。

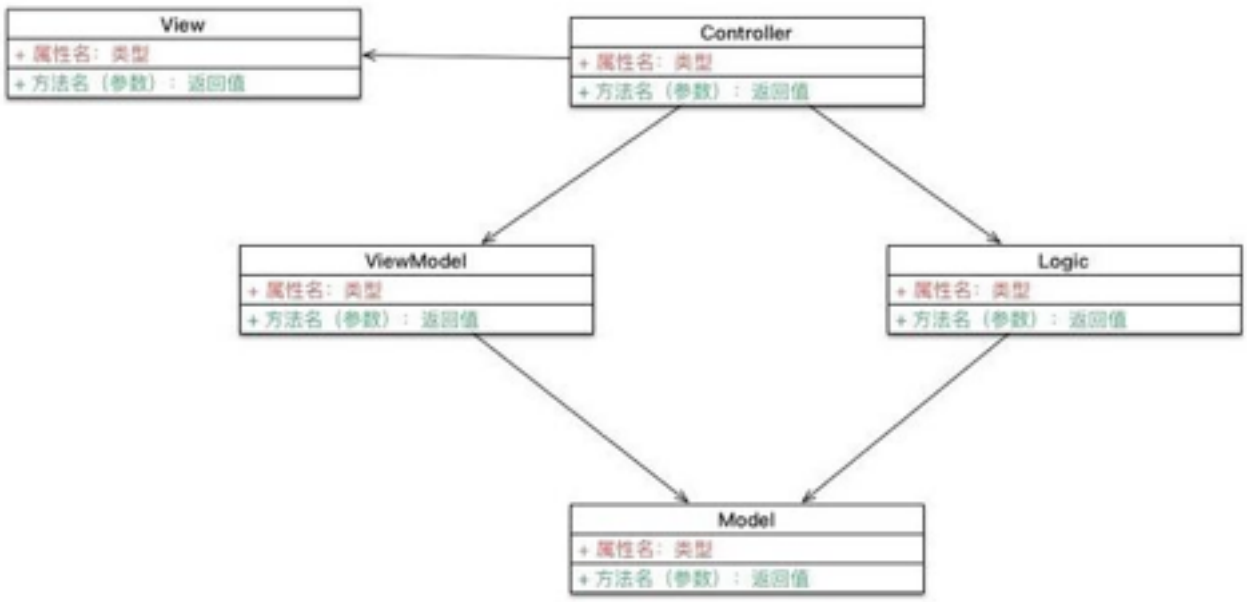
关于绑定机制，文章中推荐ReactiveCocoa。我去大致看了一下，主要是将KVO，Block，notification，delegate等各种通讯机制统一为RACSignal，将界面和数据进行双向绑定，功能确实强大。但是这个风格和普通iOS的开发习惯差距比较大，一下子很难变过来。文章中也说只是推荐，不强求，所以我也就一直没有采用。被误解的MVC和被神化的MVVM

至于绑定机制，在Swift中可以使用属性观察者。ViewModel一般作为Controller的一个属性，对它进行观察，一旦变化，就用ViewModel新的值设置界面元素，感觉挺好用的。还有一些相隔很远或者一对多的变化，一般可以采用NSNotification来达到目的。

从网络取数据，业务逻辑（相对于显示逻辑），应该放在那里呢？文章中没有说，看意思是保留在Controller中。还有的观点认为应该放在ViewModel中，这当然有道理，而且这是主流的理解。但是这样会让ViewModel变成另外一个上帝类。

我理解的MVVM

这是本人的理解，仅仅一家之言。主流的观点没有Logic那个类，从图中删除基本上就是了。ViewModel将是替代Controller的一个上帝类。



Controller主要作为调度者，居于中心位置。客串部分View相关功能：比如动画里面关于view的位置改变，这些代码是要放在Controller里面的。这也符合Controller+View实现view功能的概念。

ViewModel专门做“显示逻辑”，并且用属性观察者做绑定，必要的时候用Notification。正向的绑定比如“action-target”响应就保留在Controller中，具体事情交个其他类做就可以了。

在Swift中，ViewModel和Model推荐用Struct；Logic倾向于用class。从一个简单直观的概念来说，ViewModel需要保持轻量级，跟随页面走，随时准备修改。Model也是轻量级，跟随后台API定义走，只是个数据结构，随时准备修改。而Logic就显得比较大，考虑稳定，考虑复用。

增加Logic类，负责业务逻辑，比如从网络取数据，修改数据库，检查用户名合法性，具体的响应逻辑，监听后的具体处理等等
猿题库 iOS 客户端架构设计

文章中的DataController相当于这里的Logic

重点是Controller减负，尽量起调度者职能，具体工作都放到Logic中处理。

Logic考虑复用，可以对应单个页面，也可以多个页面共用。按照业务逻辑的思路去划分模块。划分标准可以和页面分类标准不一样。

对于复杂页面，View和ViewModel可以多个，按照组件的模式去考虑。

对于表格，ViewModel对应的是表格的cell，dataSource数组中放ViewModel的序列。

表格的delegate和dataSource，目前来看，放在Controller中是最方便的。当然，为了给Controller减负，再新增一个类TableDelegate也是很不错的方法。

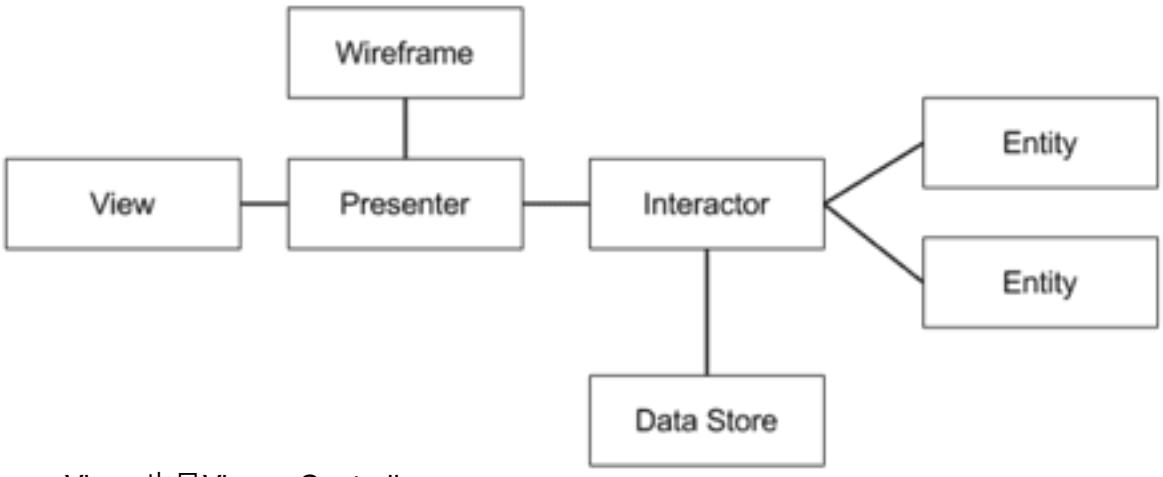
如果表格包含在一个组件中，用容器view做delegate和dataSource是一个不错的选择。

主要想法就是想方设法“架空”Controller，让它只做一个调度者，管理页面的生命周期就可以了。实在非它不可的时候，才让它做具体的事情。

不引入ReactiveCocoa等庞大的第三方库。这里有一篇文章不错，值得学习一下。
ReactiveCocoa 和 MVVM 入门

VIPER

这是比MVVM分类更细的一种模式。
经典图形



View: 也是View + Controller

Present：相当于ViewModel，叫展示器

Interactor：交互器，侧重于业务逻辑；从网络取数据，数据库等功能都在这里。

Entity:就是Model，仅仅是数据定义

WireFrame：就是Router，是页面跳转

值得借鉴的地方

将页面跳转独立出来，做成公共模块

将业务逻辑独立出来，做成公共模块

如果只是对于单个页面，分这么多类，感觉有点啰嗦了。不够对于多页面的模块来说，还是有借鉴意义的

其他架构

一些实际在用，但是没有通用缩写名称的架构
分层模式
分层架构.png

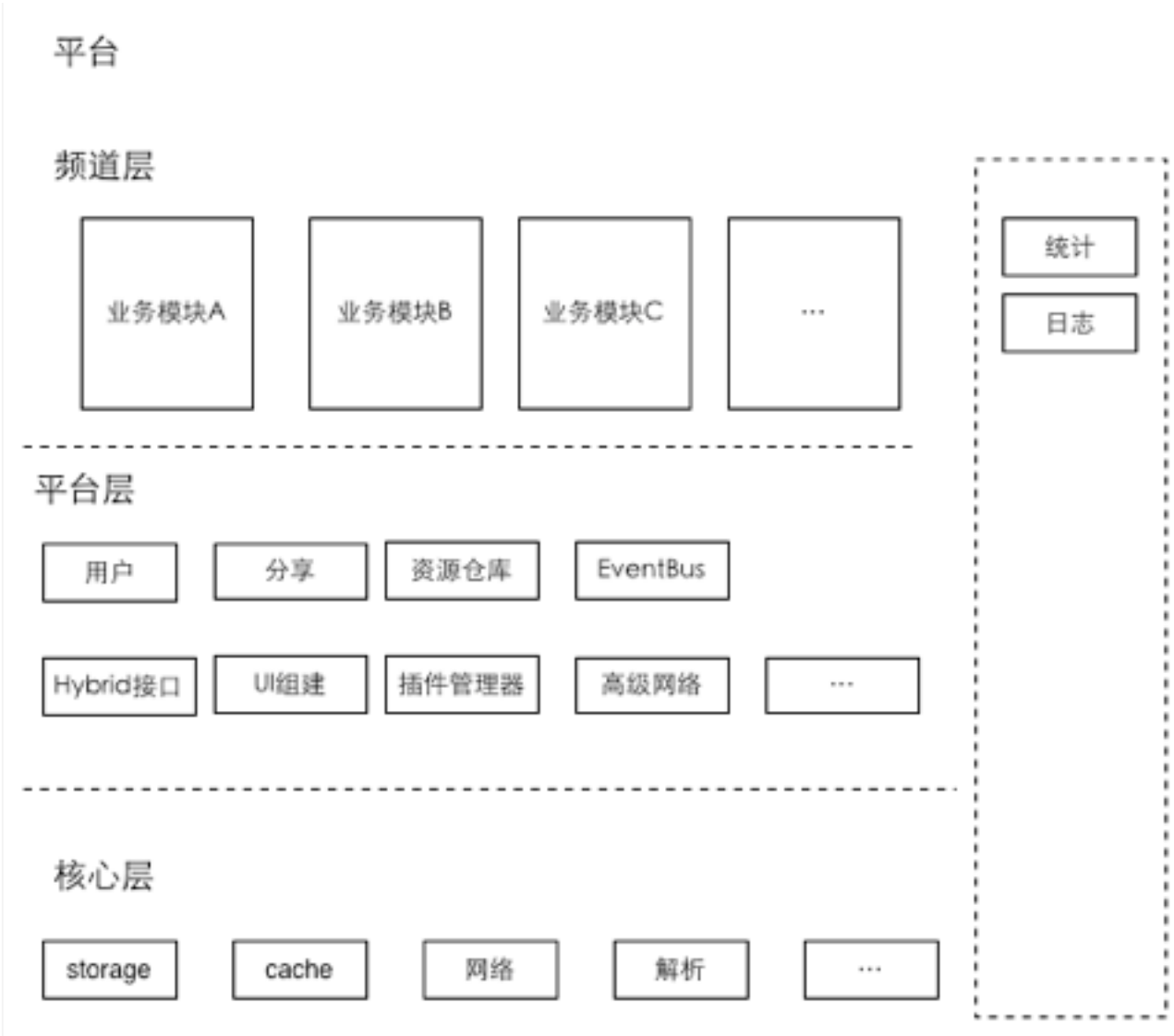
将服务service的概念引入客户端，作为一个中间层，进行隔离

业务逻辑作为服务模块，通过服务的方式进行访问

数据访问跟业务逻辑，表现层分开，是跟后台的接口层

表现层只关注UI，相当于VIPER中的V和P；或者是MVVM中的ViewModel（仅显示逻辑）和View，但是没有双向绑定；

平台模式



当前的APP，大多数是Native和H5的混合，将两者的接口代码统一成通用模块是比较好的做法

插件化也是一个越来越普遍的趋势，比如分享，第三方登录，支付等等，都由第三方以插件的形式提供。对这些插件集中管理也是好的做法

APP随着公司发展壮大，分出不同的事业部，在同一公司多个APP或者不同业务；这样就有两个相反的发展趋势：一方面，想共用模块，让多个业务共享；另一方面，各自业务又要隔离，独立发展。公司也有可能成立公共的平台部。各业务部门之间是纵向拆分；业务和平台之间是横向拆分。这就导致二维划分的立体架构。

参考文章

移动App架构设计

分离出界面层，尽量薄，和UI同学协作，快速应变
分离数据层，尽量薄，与后台合作，快速应变

一些思考

架构设计没有统一的标准，上面接触到的架构模型，都有积极的参考意义，但是都不能照搬。需要根据自己的实际情况进行一定的权衡取舍
Step0：平台型应用

以URL的方式，由主App调用子App

形式类似于调用打电话，发短信，发邮件

URL的定义需要统筹考虑

Step1：纵向划分

分Native，H5，插件三部分

Native和H5之间提供统一的桥接模块

Native和插件之间提供统一的桥架模块

如果加入ReactNative，那么也要提供Native和ReactNative之间的桥接模块。这个可以先预留，也可以以后再添加。

Step2：横向划分

Native部分进行横划分，因为这一块是最耗资源的部分

最上层是界面层（名字可以叫表现层或者UI层），这里可以借用MVVM的思想。M不用考虑，由下层以服务的形式提供。VM仅仅做显示逻辑，在Swift中用struct。这一层是跟产品的交流层，尽量薄，并且能够快速应对变化。业务逻辑等能分出去的功能，一律分出去。核心和重点就是让Controller只做调度者，万不得已可以酌情参与很少一部分的view工作。

最底层是微服务层（micro service）。这一层提供基本的功能，比如网络，缓存，加解密，系统信息，日志，统计等等。微服务的概念是只能供其他模块调用，不能调用其他模块的服务。本层中的模块之间也不能相互调用。这里是一些基础的组件，按照功能划分，相互间的隔离是第一考虑要素。要求高内聚。

中间是服务层（service），这里的服务可以调用微服务，也可以相互之间调用。

分三层相对简单一点。当然也可以分出一些接口层，服务层还可以分出公共服务层，业务逻辑层等。这个可以根据需要灵活配置。但是总体上分三层（界面、服务、微服务）。

不要跨层调用，界面层只能调用服务层提供的服务。服务层可以自己完成工作，也可以调用其他服务或者微服务完成工作。

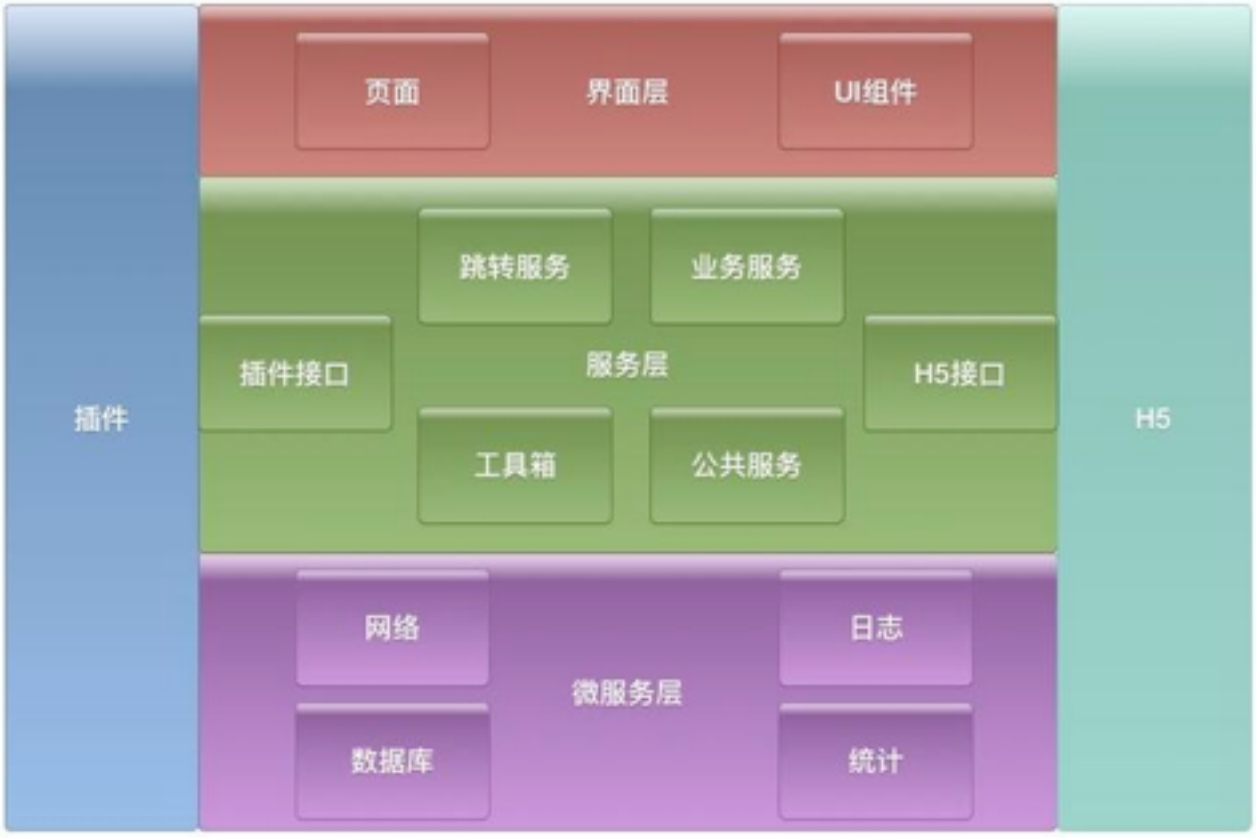
Step3：层内划分

界面层：按照页面进行组织，提供公共的UI组件，可以理解为（M）VVM。VM作为将“界面显示”转换为“数据操作”的媒介，利用Swift的属性观察者特性，进行一级绑定。不引入RxSwift等函数式编程的大型第三方库。

服务层：分为公共服务，跳转逻辑，业务逻辑等模块，按照逻辑功能划分。跟具体页面不必相关，跟界面层的接口为各ViewModel种定义的协议。

微服务层：按照功能划分，不设计业务逻辑，分网络、数据库、加解密，日志，统计等功能

框架图



语言选择Swift，最低支持版本iOS8，有条件的从iOS9开始

服务和微服务都以framework的形式提供，模块间的隔离需要重点考虑。

服务service和微服务仅仅是逻辑上的层次结构，在具体的工程组织上，都采用一级framework封装，相互间的层级和调用采用相互间的依赖隐含表示。

提供一个界面隔离的service.framework，界面层只调用它完成所有任务。作用相当于Foundation。

以workspace的方式组织工程，第三方管理工具采用Carthage。有条件的情况下，微服务以及部分服务可以采用私有Carthage的形式，更方便复用。

插件也要求以framework的形式提供，不接受.a的静态库。

如果暂时需要用到Object-C、C、C++，都统一成framework的形式，以后逐步用Swift替换。

类图

界面层

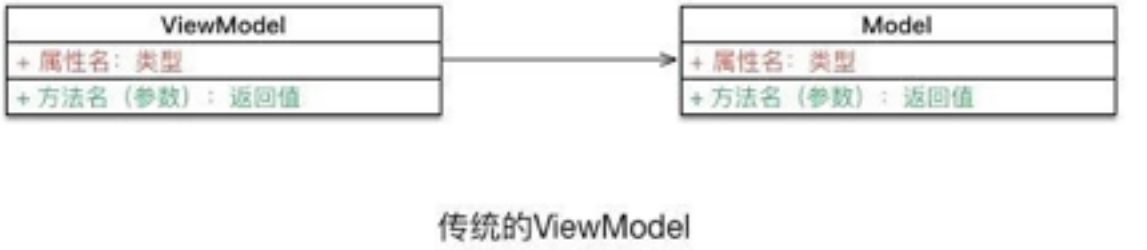
AppDelegate、ViewController仅仅作为调度者存在，不做任何具体的事情。

ViewModel仅仅做显示逻辑相关的事情，仅仅起到将界面转换为数据的作用。用结构体struct，每个成员都是普通变量，并且都有默认值，代表了页面的确定性。ViewModel是一种数据结构，做显示逻辑的事情。

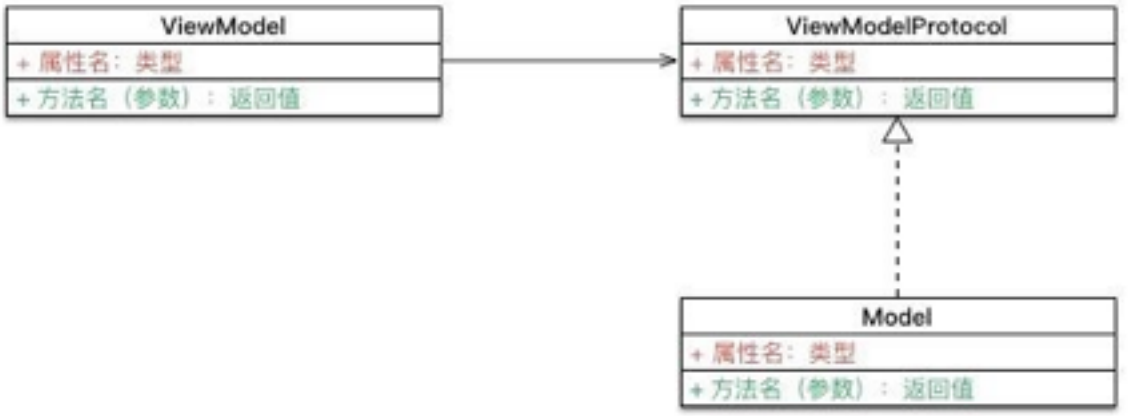
UI组件仅由View和ViewModel组成，只能包含显示逻辑，不能包含跳转逻辑，业务逻辑等。

ViewModel放UI层和Service层都有一定道理：放UI层表示显示逻辑；放service层表示UI和service之间的数据接口。考虑再三，觉得还是应该放UI层。ViewModel最大的作用还是在于将UI变化转变为数据操作，本质上离UI应该更近一些。

至于UI层和Service层之间的接口，还是定义相关的的ViewModel protocol比较好。这些protocol的定义放在service中（由于framework的影响），将ViewModel的一些基本需求放在protocol中。service中用Model或者还是用其他来满足这个protocol，就不做要求了。用protocol作为接口比单纯用Model做接口要好。因为Model随着后台API定义而变，而protocol只相当于一个基类，类型更灵活



传统的ViewModel



修改后的ViewModel

除了定义一个ViewModel的protocol之外，再定义一个是service的protocol，（既然引入service概念，就可以淡化logic和data的概念）。

界面层保持最轻量级。页面跳转逻辑，具体业务逻辑等工作全部下沉到服务层来做。

ViewModel是一个struct，主要做显示逻辑，概念相对比较小，一般一个页面一个或者多个。而service是一个类，概念比较大，可以多个页面共用一个。尺度可以根据具体情况灵活掌握。不同的页面，通过扩展遵循不同的协议区分开来。类本身可能比较大，但是每一部分都是相对比较小的。

服务层

service.framework作为一个粘合层存在，AppDelegate、ViewController只要import service就可以调用相关服务了。

金融.framework、保险.framework等属于业务特有的逻辑

Router、用户、分享等属于业务无关的公共逻辑

层内的各模块间可以相互调用

具体存在形式，单例、类、或者framework等，可根据具体情况灵活决定。

为了结构清晰，图中的调用关系线只画出了很少的一部分，大部分线都没有画出来。

微服务层

从开发的角度，按照功能分类；是工程师之间交流的技术语言，而不是跟产品交流的业务语言

功能高内聚，作为被调用的基础模块

模块之间不要存在相互调用的关系

以framework的形式存在。高内聚，高复用。