iOS 架构模式--解密 MVC，MVP，MVVM以及VIPER架构
我们可以定义一个好的架构应该具备的特点：

- 任务均衡分摊给具有清晰角色的实体
- 可测试性通常都来自与上一条（对于一个合适的架构是非常容易）
- 易用性和低成本维护

前三种设计模式都把一个应用中的实体分为以下三类：

- **Models**--负责主要的数据或者操作数据的数据访问层，可以想象 Perspn 和 PersonDataProvider 类。
- **Views**--负责展示层（GUI），对于iOS环境可以联想一下以 UI 开头的所有类。
- **Controller/Presenter/ViewModel**--负责协调 Model 和 View，通常根据用户在View上的动作在Model上作出对应的更改，同时将更改的信息返回到View上。

1️⃣解释MVC的事实：
由于Controller是一个介于View 和 Model之间的协调器，所以View和Model之间没有任何直接的联系。Controller是一个最小可重用单元，这对我们来说是一个好消息，因为我们总要找一个地方来写逻辑复杂度较高的代码，而这些代码又不适合放在Model中。Cocoa的MVC模式驱使人们写出臃肿的视图控制器，因为它们经常被混杂到View的生命周期中，因此很难说View和ViewController是分离的。尽管仍可以将业务逻辑和数据转换到Model，但是大多数情况下当需要为View减负的时候我们却无能为力了，View的最大的任务就是向Controller传递用户动作事件。
ViewController最终会承担一切代理和数据源的职责，还负责一些分发和取消网络请求以及一些其他的任务，直到进行单元测试的时候才会发现问题越来越明显。因为你的ViewController和View是紧密耦合的，对它们进行测试就显得很艰难--你得有足够的创造性来模拟View和它们的生命周期，在以这样的方式来写View Controller的同时，业务逻辑的代码也逐渐被分散到View的布局代码中去。
以上所述，似乎Cocoa MVC 看起来是一个相当差的架构方案。我们来重新评估一下文章开头我们提出的MVC一系列的特征：

- **任务均摊**--View和Model确实是分开的，但是View和Controller却是紧密耦合的
- **可测试性**--由于糟糕的分散性，只能对Model进行测试
- **易用性**--与其他几种模式相比最小的代码量。熟悉的人很多，因而即使对于经验不那么丰富的开发者来讲维护起来也较为容易。

如果你不想在架构选择上投入更多精力，那么Cocoa MVC无疑是最好的方案，而且你会发现一些其他维护成本较高的模式对于你所开发的小的应用是一个致命的打击。
“就开发速度而言，Cocoa MVC是最好的架构选择方案。”
2️⃣我们来看下MVP模式下的三个特性的分析：

- 任务均摊--我们将最主要的任务划分到Presenter和Model，而View的功能较少（虽然上述例子中Model的任务也并不多）。

- 可测试性--非常好，由于一个功能简单的View层，所以测试大多数业务逻辑也变得简单

- 易用性--在我们上边不切实际的简单的例子中，代码量是MVC模式的2倍，但同时MVP的概念却非常清晰

还有一些其他形态的MVP--监控控制器的MVP。但是我们之前就了解到，模糊的职责划分是非常糟糕的，更何况将View和Model紧密的联系起来。这和Cocoa的桌面开发的原理有些相似。
和传统的MVC一样，写这样的例子没有什么价值，故不再给出。
3️⃣让我们再来看看MVVM关于三个特性的评估：

- 任务均摊 -- 在例子中并不是很清晰，但是事实上，MVVM的View要比MVP中的View承担的责任多。因为前者通过ViewModel的设置绑定来更新状态，而后者只监听Presenter的事件但并不会对自己有什么更新。

- 可测试性 -- ViewModel不知道关于View的任何事情，这允许我们可以轻易的测试ViewModel。同时View也可以被测试，但是由于属于UIKit的范畴，对他们的测试通常会被忽略。

- 易用性 -- 在我们例子中的代码量和MVP的差不多，但是在实际开发中，我们必须把View中的事件指向Presenter并且手动的来更新View，如果使用绑定的话(**在View层的绑定，它并不需要其他附加的代码来更新View**)，MVVM代码量将会小的多。

4️⃣**VIPER--把LEGO建筑经验迁移到iOS app的设计**

- **交互器** -- 包括关于数据和网络请求的业务逻辑，例如创建一个实体（数据），或者从服务器中获取一些数据。为了实现这些功能，需要使用服务、管理器，但是他们并不被认为是VIPER架构内的模块，而是外部依赖。
- **展示器** -- 包含UI层面的业务逻辑以及在交互器层面的方法调用。
- **实体** -- 普通的数据对象，不属于数据访问层次，因为数据访问属于交互器的职责。
- **路由器** -- 用来连接VIPER的各个模块。

当我们把VIPER和MV(X)系列作比较时，我们会在任务均摊性方面发现一些不同:

- **Model** 逻辑通过把实体作为最小的数据结构转换到交互器中。
- **Controller/Presenter/ViewModel**的UI展示方面的职责移到了Presenter中，但是并没有数据转换相关的操作。
- **VIPER**是第一个通过路由器实现明确的地址导航模式。

让我们再来评估一下特性:

- 任务均摊 -- 毫无疑问，VIPER是任务划分中的佼佼者。
- 可测试性 -- 不出意外地，更好的分布性就有更好的可测试性。
- 易用性 -- 最后你可能已经猜到了维护成本方面的问题。你必须为很小功能的类写出大量的接口。