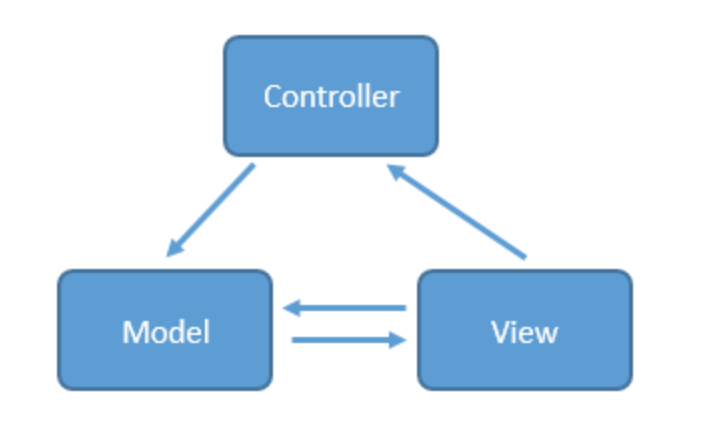
Mvc，MVP和MVVM

## 1.MVC框架



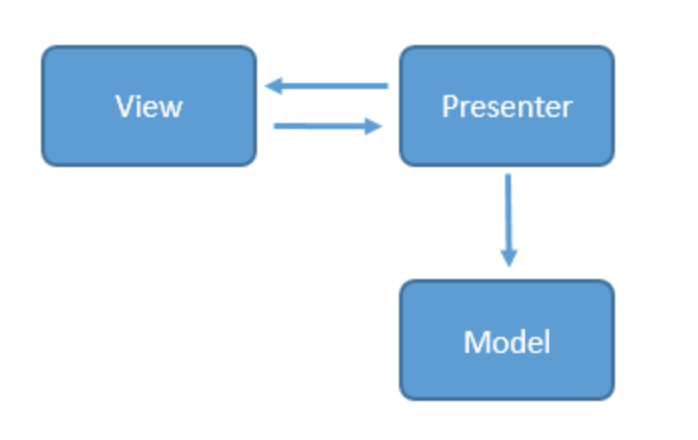
对于原生的Android项目来说，layout.xml里面的xml文件就对应于MVC的view层，里面都是一些view的布局代码，而各种java bean，还有一些类似repository类就对应于model层，至于controller层嘛，当然就是各种activity咯。大家可以试着套用我上面说的MVC的工作原理是理解。比如你的界面有一个按钮，按下这个按钮去网络上下载一个文件，这个按钮是view层的，是使用xml来写的，而那些和网络连接相关的代码写在其他类里，比如你可以写一个专门的networkHelper类，这个就是model层，那怎么连接这两层呢？是通过button.setOnClickListener()这个函数，这个函数就写在了activity中，对应于controller层。是不是很清晰。

为什么会有MVP和MVVM：因为xml作为view层，控制能力实在太弱了，你想去动态的改变一个页面的背景，或者动态的隐藏/显示一个按钮，这些都没办法在xml中做，只能把代码写在activity中， 了activity既是controller层，又是view层的这样一个窘境。大家回想一下自己写的代码，如果是一个逻辑很复杂的页面，activity或者fragment是不是动辄上千行呢？这样导致activity和fragment不仅代码很多，而且后期维护也很麻烦。

MVC还有一个重要的缺陷，大家看上面那幅图，view层和model层是相互可知的，这意味着两层之间存在耦合，耦合对于一个大型程序来说是非常致命的，因为这表示开发，测试，维护都需要花大量的精力。

## 2.MVP框架

MVP作为MVC的演化，解决了MVC不少的缺点，对于Android来说，MVP的model层相对于MVC是一样的，而activity和fragment不再是controller层，而是纯粹的view层，所有关于用户事件的转发全部交由presenter层处理。下面还是让我们看图



View：对应于Activity/Fragment/自定义View，主要负责UI渲染。  
Model：数据获取模块  
Presenter： 负责数据处理以及View和Model的交互等，持有Model和View的引用。从而使view和mode分离

从图中就可以看出，MVP和MVVM最明显的差别就是view层和model层不再相互可知，完全的解耦，取而代之的presenter层充当了桥梁的作用，Activity的功能就是响应生命周期和显示界面，具体其他的工作都丢到了Presenter层中进行完成。用于操作view层发出的事件传递到presenter层中，presenter层去操作model层，并且将数据返回给view层，整个过程中view层和model层完全没有联系。看到这里大家可能会问，虽然view层和model层解耦了，但是view层和presenter层不是耦合在一起了吗？其实不是的，对于view层和presenter层的通信，我们是可以通过接口实现的，具体的意思就是说我们的activity，fragment可以去实现实现定义好的接口，而在对应的presenter中通过接口调用方法。不仅如此，我们还可以编写测试用的View，模拟用户的各种操作，从而实现对Presenter的测试。这就解决了MVC模式中测试，维护难的问题。

当然，其实最好的方式是使用fragment作为view层，而activity则是用于创建view层(fragment)和presenter层(presenter)的一个控制器。

接口的话，Model并不是必须有的，但是一定会有View和Presenter。

通过上面的介绍，MVP的主要特点就是把Activity里的许多逻辑都抽离到View和Presenter接口中去，并由具体的实现类来完成。这种写法多了许多IView和IPresenter的接口， 在某种程度上加大了开发的工作量，刚开始使用MVP的小伙伴可能会觉得这种写法比较别扭，而且难以记住。其实一开始想太多也没有什么卵用，只要在具体项目中多写几次。

MVP模式也会存在一系列的缺点：

1、Presenter层要处理的业务逻辑过多，复杂的业务逻辑会使P层非常庞大和臃肿。

2、Presenter通过接口方式持有View层引用，接口及接口中声明的方法粒度无法把握，可能需要声明大量接口以及接口中需要声明太多方法，而其中有些方法是否会用到以及是否会增加或删减还需要后续进一步确认。

3、Activity中需要声明大量跟UI相关的方法，而相应的事件通过Presenter调用相关方法来实现。两者互相引用和调用，存在耦合。一旦View层的UI视图发生改变，接口中的方法就需要改变，View层和P层同时都需要修改。

2.1 MVP的优点

1、使Activity中的代码更加简洁   
在传统的项目中Activity兼顾着Controller和View，这使得其代码分分钟上千行（本人深受其害），这使得代码难以理解难以维护，看到这样的Activity就想吐。   
使用MVP之后，Activity就能瘦身许多了，基本上只有FindView、SetListener以及Init的代码。其他的就是对Presenter的调用，还有对View接口的实现。这种情形下阅读代码就容易多了，   
而且你只要看Presenter的接口，就能明白这个模块都有哪些业务，很快就能定位到具体代码。Activity变得容易看懂，容易维护，以后要调整业务、删减功能也就变得简单许多。

2、方便进行单元测试   
一般单元测试都是用来测试某些新加的业务逻辑有没有问题，如果采用传统的代码风格，我们可能要先在Activity里写一段测试代码，测试完了再把测试代码删掉换成正式代码，   
这时如果发现业务有问题又得换回测试代码，咦，测试代码已经删掉了！好吧重新写吧……   
MVP中，由于业务逻辑都在Presenter里，我们完全可以写一个PresenterTest的实现类继承Presenter的接口，现在只要在Activity里把 Presenter的创建换成PresenterTest，   
就能进行单元测试了，测试完再换回来即可。万一发现还得进行测试，那就再换成PresenterTest吧。

~~3、避免Activity的内存泄露~~   
APP发生OOM的最大原因就是出现内存泄露造成APP的内存不够用，而造成内存泄露的两大原因之一就是Activity泄露（Activity Leak）（另一个原因是Bitmap泄露（Bitmap Leak））。   
Java一个强大的功能就是其虚拟机的内存回收机制，这个功能使得Java用户在设计代码的时候，不用像 C++ 用户那样考虑对象的回收问题。然而，Java用户总是喜欢随便写一大堆对象，   
然后幻想着虚拟机能帮他们处理好内存的回收工作。可是虚拟机在回收内存的时候，只会回收那些没有被引用的对象，被引用着的对象因为还可能会被调用，所以不能回收。   
Activity 是有生命周期的，用户随时可能切换Activity，当APP的内存不够用的时候，系统会回收处于后台的Activity的资源以避免OOM。 采用传统的模式，一大堆异步任务和对UI的操作都放在Activity里面，比如你可能从网络下载一张图片，在下载成功的回调里把图片加载到Activity的ImageView里面， 所以异步任务保留着对Activity的引用。这样一来，即使Activity已经被切换到后台（onDestroy 已经执行），这些异步任务仍然保留着对Activity实例的引用， 所以系统就无法回收这个Activity实例了，结果就是Activity Leak。Android 的组件中，Activity 对象往往是在堆（Java Heap）里占最多内存的，所以系统会优先回收Activity对象， 如果有Activity Leak，APP很容易因为内存不够而 OOM。 采用MVP模式，只要在当前的Activity的onDestroy里，分离异步任务对Activity的引用，就能避免Activity Leak。（这些需要自己去注意以及优化的）

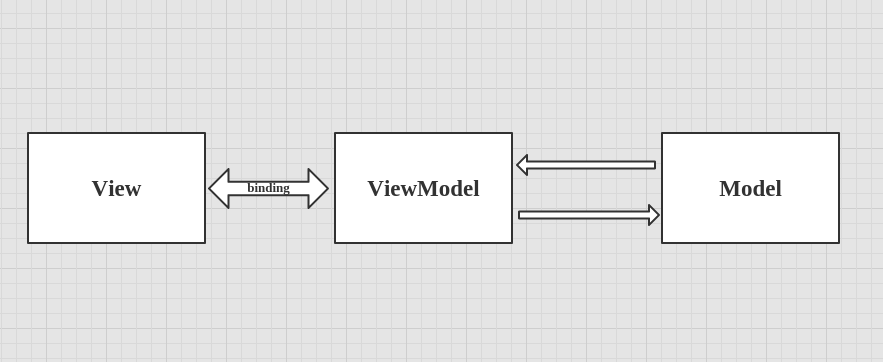
4.降低耦合，方便维护  
MVP的使用，使Activity中的网络请求剥离出来 成为model、presenter，model只负责网络的请求、pesenter负责处理请求网络后的数据处理：加载中 成功 or 失败 取消加载；最后View进行界面的展示

Activity和present相互持有对象，当activity销毁的时候注意释放，防止内存泄漏

缺点：(1)构架存在漏洞 (2)代码冗余量大 (3)通用性差 (4)异步任务时， 调用View可能引发的空指针异常（需要判断view是否存在）

## 3.MVVM

MVVM最早是由微软提出的



MVVM 模式将 Presenter 改名为 ViewModel，基本上与 MVP 模式完全一致。唯一的区别是，它采用双向绑定（data-binding）：View的变动，自动反映在 ViewModel，反之亦然。

View：对应于Activity/Fragment/自定义View，主要负责UI渲染。

Model：数据获取模块

ViewModel： 负责业务逻辑处理，负责View和Model的交互。和View层双向绑定。

Mvvm模式是通过将View层和ViewModel层进行双向绑定， View层的变化会自动通知给ViewModel层，而ViewModel层的数据变化也会通知给View层进行相应的UI的更新。这样，Model层只负责暴露获取数据的方法，View层只负责监听数据的变化更新，而ViewModel负责接收View层的事件指令以及获取并处理数据。从而实现业务逻辑和Ui的隔离。

这里要感谢[泡在网上的日子](http://www.jcodecraeer.com/plus/list.php?tid=16)，因为前面看到的三张图我都是从它的博客中摘取的，如果有人知道不允许这样做的话请告诉我，我会从我的博客中删除的，谢谢。

从图中看出，它和MVP的区别貌似不大，只不过是presenter层换成了viewmodel层，还有一点就是view层和viewmodel层是相互绑定的关系，这意味着当你更新viewmodel层的数据的时候，view层会相应的变动ui。

使用MVVM模式的优点：

1、低耦合度：

在MVVM模式中，数据处理逻辑是独立于UI层的。ViewModel只负责提供数据和处理数据，不会持有View层的引用。而View层只负责对数据变化的监听，不会处理任何跟数据相关的逻辑。在View层的UI发生变化时，也不需要像MVP模式那样，修改对应接口和方法实现，一般情况下ViewModel不需要做太多的改动。

2、数据驱动：

MVVM模式的另外一个特点就是数据驱动。UI的展现是依赖于数据的，数据的变化会自然的引发UI的变化，而UI的改变也会使数据Model进行对应的更新。ViewModel只需要处理数据，而View层只需要监听并使用数据进行UI更新。

3、异步线程更新Model：

Model数据可以在异步线程中发生变化，此时调用者不需要做额外的处理，数据绑定框架会将异步线程中数据的变化通知到UI线程中交给View去更新。

4、方便协作：

View层和逻辑层几乎没有耦合，在团队协作的过程中，可以一个人负责Ui 一个人负责数据处理。并行开发，保证开发进度。

5、易于单元测试：

MVVM模式比较易于进行单元测试。ViewModel层只负责处理数据，在进行单元测试时，测试不需要构造一个fragment/Activity/TextView等等来进行数据层的测试。同理View层也一样，只需要输入指定格式的数据即可进行测试，而且两者相互独立，不会互相影响。

6、数据复用：

ViewModel层对数据的获取和处理逻辑，尤其是使用Repository模式时，获取数据的逻辑完全是可以复用的。开发者可以在不同的模块，多次方便的获取同一份来源的数据。同样的一份数据，在版本功能迭代时，逻辑层不需要改变，只需要改变View层即可。

后面用LiveData + ViewModel + Repository做的MVVM的框架，但是使用的时候，每次还是需要开发者通过LiveData 来实现View 和ViewModel的数据绑定和命令绑定，用起来还是感觉麻烦不爽。 建议你可以看看其他Reactive的框架，比如美团有个Router框架，还有FaceBook的 AndroidFlux框架，看看能否满足你的需求