MVP（Model-View-Presenter）模式

1. **简介**

MVP模式是MVC模式在Android上的一种变体，要介绍MVP就得先介绍MVC。在MVC模式中，Activity应该是属于View这一层。而实质上，Activity既承担了View，同时也包含一些Controller的东西在里面。这对于开发与维护来说不太友好，耦合度大高了。

把Activity的View和Controller抽离出来就变成了View和Presenter，这就是MVP模式。

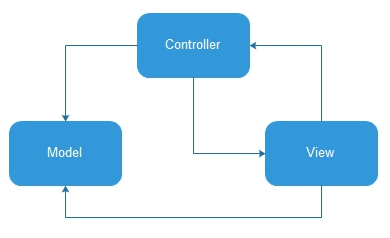
Android MVP Pattern：也不是什么新鲜的东西了，我在自己的项目里也普遍地使用了这个设计模式。当项目越来越庞大、复杂，参与的研发人员越来越多的时候，**MVP 模式**的优势就充分显示出来了。

1. **MVC模式**

MVP模式（Model-View-Presenter）可以说是MVC模式（Model-View-Controller）在Android开发上的一种变种、进化模式。MVC大家可能比较熟悉，就算不熟悉也可能或多或少地在自己的项目中用到过。要介绍MVP模式，就不得不先说说MVC模式。

MVC模式的结构分为三部分：

* 实体层的Model——JavaBean实体类，用于保存实例数据
* 视图层的View——就是程序的UI界面，用于向用户展示数据以及接收用户的输入。
* 控制层的Controller——更新UI界面和数据实例。

****

**例如：**View层接受用户的输入，然后通过Controller修改对应的Model实例；同时，当Model实例的数据发生变化的时候，需要修改UI界面，可以通过Controller更新界面。（View层也可以直接更新Model实例的数据，而不用每次都通过Controller，这样对于一些简单的数据更新工作会变得方便许多。）

**栗子：**现在要实现一个飘雪的动态壁纸，可以给雪花定义一个实体类Snow，里面存放XY轴坐标数据，View层当然就是SurfaceView（或者其他视图），为了实现雪花飘的效果，可以启动一个后台线程，在线程里不断更新Snow实例里的坐标值，这部分就是Controller的工作了，Controller里还要定时更新SurfaceView上面的雪花。进一步的话，可以在SurfaceView上监听用户的点击，如果用户点击，只通过Controller对触摸点周围的Snow的坐标值进行调整，从而实现雪花在用户点击后出现弹开等效果。具体的MVC模式请自行Google。

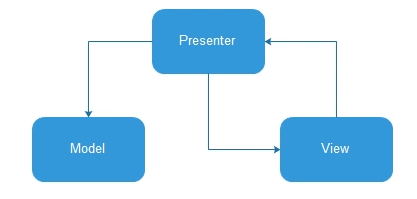
1. **MVP模式**

在Android项目中，Activity和Fragment占据了大部分的开发工作。如果有一种设计模式（或者说代码结构）专门是为优化Activity和Fragment的代码而产生的，你说这种模式重要不？这就是MVP设计模式。

按照MVC的分层，Activity和Fragment（后面只说Activity）应该属于View层，用于展示UI界面，以及接收用户的输入，此外还要承担一些生命周期的工作。Activity是在Android开发中充当非常重要的角色，特别是TA的生命周期的功能，所以开发的时候我们经常把一些业务逻辑直接写在Activity里面，这非常直观方便，代价就是Activity会越来越臃肿，超过1000行代码是常有的事，而且如果是一些可以通用的业务逻辑（比如用户登录），写在具体的Activity里就意味着这个逻辑不能复用了。如果有进行代码重构经验的人，看到1000+行的类肯定会有所顾虑。因此，Activity不仅承担了View的角色，还承担了一部分的Controller角色，这样一来V和C就耦合在一起了，虽然这样写方便，但是如果业务调整的话，要维护起来就难了，而且在一个臃肿的Activity类查找业务逻辑的代码也会非常蛋疼，所以看起来有必要在Activity中，把View和Controller抽离开来，而这就是MVP模式的工作了。

**MVP模式的核心思想：**

MVP把Activity中的UI逻辑抽象成View接口（IView），把业务逻辑抽象成Presenter接口(IPresenter)，Model类还是原来的Model。



这就是MVP模式，现在这样的话，Activity的工作的简单了，只用来响应生命周期，其他工作都丢到Presenter中去完成。

**MVP模式的好处：**

* 分离了视图逻辑（IView）和业务逻辑(IPresenter)，降低了耦合；（SL:如何区分视图逻辑和业务逻辑）
* Activity只处理生命周期的任务，代码变得更加简洁；

相信很多人阅读代码的时候，都是从Activity开始的，对着一个1000+行代码的Activity，看了都觉得难受。

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。

* 视图逻辑和业务逻辑分别抽象到了View和Presenter的接口中去，提高代码的可阅读性
* Presenter被抽象成接口，可以有多种具体的实现，所以方便进行单元测试
* 把业务逻辑抽到Presenter中去，避免后台线程引用着Activity导致Activity的资源无法被系统回收从而引起内存泄露和OOM

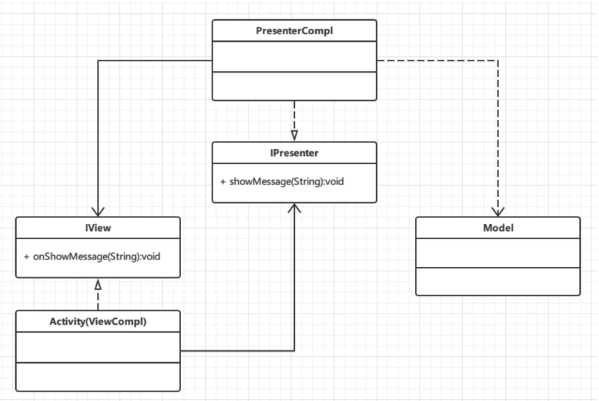
Android APP 发生OOM的最大原因就是出现内存泄露造成APP的内存不够用，而造成内存泄露的两大原因之一就是Activity泄露（Activity Leak）（另一个原因是Bitmap泄露（Bitmap Leak））。

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MV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。

1. **MVP的使用**



上面一张简单的MVP模式的UML图，从图中可以看出，使用MVP，至少需要经历以下步骤：

创建IPresenter接口，把所有业务逻辑的接口都放在这里，并创建它的实现PresenterCompl（在这里可以方便地查看业务功能，由于接口可以有多种实现所以也方便写单元测试）

创建IView接口，把所有视图逻辑的接口都放在这里，其实现类是当前的Activity/Fragment

由UML图可以看出，Activity里包含了一个IPresenter，而PresenterCompl里又包含了一个IView并且依赖了Model。Activity里只保留对IPresenter的调用，其它工作全部留到PresenterCompl中实现

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，就能熟悉MVP模式的写法，理解TA的意图，以及享受其带来的好处。

1. **问题：**

1.如果区分UI逻辑和业务逻辑？