# Android App的设计架构：MVC,MVP,MVVM与架构经验谈-android,mvp,mvc 相关文章-天码营

和MVC框架模式一样，Model模型处理数据代码不变在Android的App开发中，很多人经常会头疼于App的架构如何设计：

- 我的App需要应用这些设计架构吗？
- MVC,MVP等架构讲的是什么？区别是什么？

本文就来带你分析一下这几个架构的特性，优缺点，以及App架构设计中应该注意的问题。

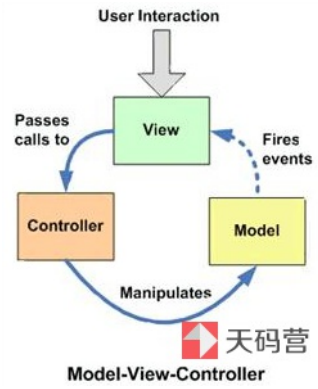
# 1.架构设计的目的

通过设计使程序模块化，做到模块内部的高聚合和模块之间的低耦合。这样做的好处是使得程序在开发的过程中，开发人员只需要专注于一点，提高程序开发的效率，并且更容易进行后续的测试以及定位问题。但设计不能违背目的，对于不同量级的工程，具体架构的实现方式必然是不同的，切忌犯为了设计而设计，为了架构而架构的毛病。

举个简单的例子：

> 一个Android App如果只有3个Java文件，那只需要做点模块和层次的划分就可以，引入框架或者架构反而提高了工作量，降低了生产力；
> 但如果当前开发的App最终代码量在**10W**行以上，本地需要进行**复杂操作**，同时也需要考虑到与其余的Android开发者以及后台开发人员之间的**同步配合**，那就需要在架构上进行一些思考！

# 2.MVC设计架构




## MVC简介

MVC全名是Model View Controller，如图，是模型(model)－视图(view)－控制器(controller)的缩写，一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。

其中M层处理数据，业务逻辑等；V层处理界面的显示结果；C层起到桥梁的作用，来控制V层和M层通信以此来达到分离视图显示和业务逻辑层。

## Android中的MVC

Android中界面部分也采用了当前比较流行的MVC框架，在Android中：

- 视图层(View)

一般采用XML文件进行界面的描述，这些XML可以理解为AndroidApp的View。使用的时候可以非常方便的引入。同时便于后期界面的修改。逻辑中与界面对应的id不变化则代码不用修改，大大增强了代码的可维护性。

- 控制层(Controller)

Android的控制层的重任通常落在了众多的Activity的肩上。这句话也就暗含了不要在Activity中写代码，要通过Activity交割Model业务逻辑层处理，这样做的另外一个原因是Android中的Actiivity的响应时间是5s，如果耗时的操作放在这里，程序就很容易被回收掉。

- 模型层(Model)

我们针对业务模型，建立的数据结构和相关的类，就可以理解为AndroidApp的Model，Model是与View无关，而与业务相关的（感谢@Xander的讲解）。对数据库的操作、对网络等的操作都应该在Model里面处理，当然对业务计算等操作也是必须放在的该层的。就是应用程序中二进制的数据。

## MVC代码实例

我们来看看MVC在Android开发中是怎么应用的吧！

先上界面图


**Controller控制器&View**

```
public class MainActivity extends ActionBarActivity implements OnWeatherListener, View.OnClickListener {    private WeatherModel weatherModel;    private EditText cityNOInput;    private TextView city;    ...    @Override    protected void onCreate(Bundle savedInstanceState) {        super.onCreate(savedInstanceState);        setContentView(R.layout.activity_main);        weatherModel = new WeatherModelImpl();        initView();    }    //初始化View    private void initView() {        cityNOInput = findView(R.id.et_city_no);        city = findView(R.id.tv_city);        ...        findView(R.id.btn_go).setOnClickListener(this);    }    //显示结果    public void displayResult(Weather weather) {        WeatherInfo weatherInfo = weather.getWeatherinfo();        city.setText(weatherInfo.getCity());        ...    }    @Override    public void onClick(View v) {        switch (v.getId()) {            case R.id.btn_go:                weatherModel.getWeather(cityNOInput.getText().toString().trim(), this);                break;        }    }    @Override    public void onSuccess(Weather weather) {        displayResult(weather);    }    @Override    public void onError() {        Toast.makeText(this, 获取天气信息失败, Toast.LENGTH_SHORT).show();    }    private T findView(int id) {        return (T) findViewById(id);    }}
```

从上面代码可以看到，Activity持有了WeatherModel模型的对象，当用户有点击Button交互的时候，Activity作为Controller控制层读取View视图层EditTextView的数据，然后向Model模型发起数据请求，也就是调用WeatherModel对象的方法 getWeather（）方法。当Model模型处理数据结束后，通过接口OnWeatherListener通知View视图层数据处理完毕，View视图层该更新界面UI了。然后View视图层调用displayResult（）方法更新UI。至此，整个MVC框架流程就在Activity中体现出来了。

**Model模型**

来看看WeatherModelImpl代码实现

```
public interface WeatherModel {    void getWeather(String cityNumber, OnWeatherListener listener);}.....public class WeatherModelImpl implements WeatherModel {    /*这部分代码范例有问题，网络访问不应该在Model中，应该把网络访问换成从数据库读取*/    @Override    public void getWeather(String cityNumber, final OnWeatherListener listener) {        /*数据层操作*/        VolleyRequest.newInstance().newGsonRequest(http://www.weather.com.cn/data/sk/ + cityNumber + .html,                Weather.class, new Response.Listener() {                    @Override                    public void onResponse(Weather weather) {                        if (weather != null) {                            listener.onSuccess(weather);                        } else {                            listener.onError();                        }                    }                }, new Response.ErrorListener() {                    @Override                    public void onErrorResponse(VolleyError error) {                        listener.onError();                    }                });    }}
```

以上代码看出，这里设计了一个WeatherModel模型接口，然后实现了接口WeatherModelImpl类。controller控制器activity调用WeatherModelImpl类中的方法发起网络请求，然后通过实现OnWeatherListener接口来获得网络请求的结果通知View视图层更新UI 。至此，Activity就将View视图显示和Model模型数据处理隔离开了。activity担当contronller完成了model和view之间的协调作用。

至于这里为什么不直接设计成类里面的一个getWeather（）方法直接请求网络数据？你考虑下这种情况：现在代码中的网络请求是使用Volley框架来实现的，如果哪天老板非要你使用Afinal框架实现网络请求，你怎么解决问题？难道是修改 getWeather（）方法的实现？ no no no，这样修改不仅破坏了以前的代码，而且还不利于维护， 考虑到以后代码的扩展和维护性，我们选择设计接口的方式来解决着一个问题，我们实现另外一个WeatherModelWithAfinalImpl类，继承自WeatherModel，重写里面的方法，这样不仅保留了以前的WeatherModelImpl类请求网络方式，还增加了WeatherModelWithAfinalImpl类的请求方式。Activity调用代码无需要任何修改。

# 3.MVP设计架构

在App开发过程中，经常出现的问题就是某一部分的代码量过大，虽然做了模块划分和接口隔离，但也很难完全避免。从实践中看到，这更多的出现在UI部分，也就是Activity里。想象一下，一个2000+行以上基本不带注释的Activity，我的第一反应就是想吐。Activity内容过多的原因其实很好解释，因为Activity本身需要担负与用户之间的操作交互，界面的展示，不是单纯的Controller或View。而且**现在大部分的Activity还对整个App起到类似IOS中的【ViewController】的作用**，这又带入了大量的逻辑代码，造成Activity的臃肿。为了解决这个问题，让我们引入MVP框架。

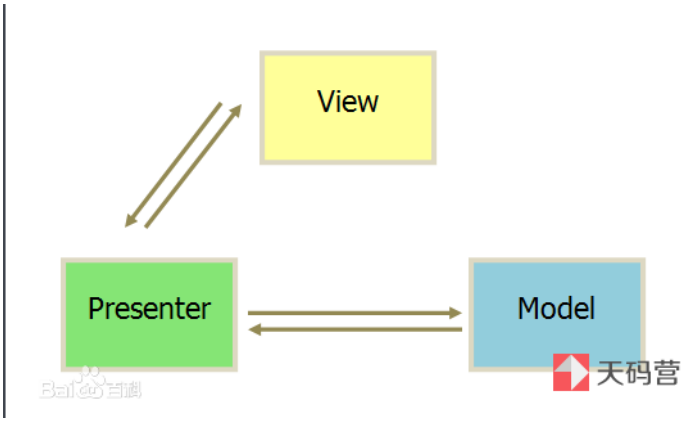
## MVC的缺点

在Android开发中，Activity并不是一个标准的MVC模式中的Controller，它的首要职责是加载应用的布局和初始化用户 界面，并接受并处理来自用户的操作请求，进而作出

响应。随着界面及其逻辑的复杂度不断提升，Activity类的职责不断增加，以致变得庞大臃肿。

## 什么是MVP?

MVP从更早的MVC框架演变过来，与MVC有一定的相似性：Controller/Presenter负责逻辑的处理，Model提供数据，View负责显示。




MVP框架由3部分组成：View负责显示，Presenter负责逻辑处理，Model提供数据。在MVP模式里通常包含3个要素（加上View interface是4个）：

- View:负责绘制UI元素、与用户进行交互(在Android中体现为Activity)
- Model:负责存储、检索、操纵数据(有时也实现一个Model interface用来降低耦合)
- Presenter:作为View与Model交互的中间纽带，处理与用户交互的负责逻辑。
- *View interface:需要View实现的接口，View通过View interface与Presenter进行交互，降低耦合，方便进行单元测试

**Tips：*View interface的必要性**

回想一下你在开发Android应用时是如何对代码逻辑进行单元测试的？是否每次都要将应用部署到Android模拟器或真机上，然后通过模拟用 户操作进行测试？然而由于Android平台的特性，每次部署都耗费了大量的时间，这直接导致开发效率的降低。而在MVP模式中，处理复杂逻辑的Presenter是通过interface与View(Activity)进行交互的，这说明我们可以通过自定义类实现这个interface来模拟Activity的行为对Presenter进行单元测试，省去了大量的部署及测试的时间。

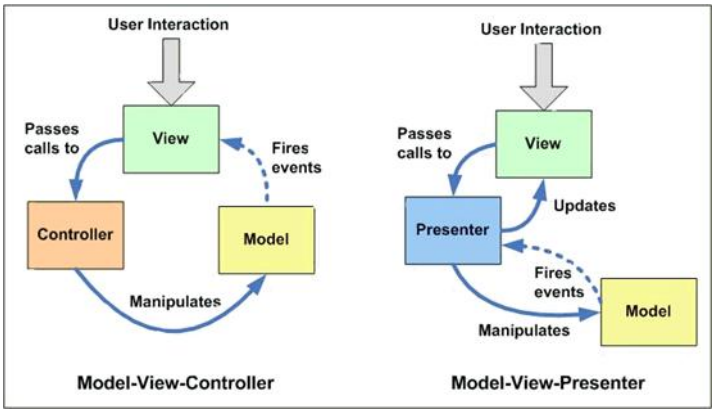
## MVC → MVP

当我们将Activity复杂的逻辑处理移至另外的一个类（Presenter）中时，Activity其实就是MVP模式中的View，它负责UI元素的初始化，建立UI元素与Presenter的关联（Listener之类），同时自己也会处理一些简单的逻辑（复杂的逻辑交由 Presenter处理）。

MVP的Presenter是框架的控制者，承担了大量的逻辑操作，而MVC的Controller更多时候承担一种转发的作用。因此在App中引入MVP的原因，是为了将此前在Activty中包含的大量逻辑操作放到控制层中，避免Activity的臃肿。

两种模式的**主要区别**：

- **（最主要区别）View与Model并不直接交互**，而是通过与Presenter交互来与Model间接交互。而在MVC中View可以与Model直接交互
- 通常View与Presenter是一对一的，但复杂的View可能绑定多个Presenter来处理逻辑。而Controller是基于行为的，并且可以被多个View共享，Controller可以负责决定显示哪个View
- Presenter与View的交互是通过接口来进行的，更有利于添加单元测试。




因此我们可以发现**MVP的优点**如下：

1、模型与视图完全分离，我们可以修改视图而不影响模型；

2、可以更高效地使用模型，因为所有的交互都发生在一个地方——Presenter内部；

3、我们可以将一个Presenter用于多个视图，而不需要改变Presenter的逻辑。这个特性非常的有用，因为视图的变化总是比模型的变化频繁；

4、如果我们把逻辑放在Presenter中，那么我们就可以脱离用户接口来测试这些逻辑（单元测试）。

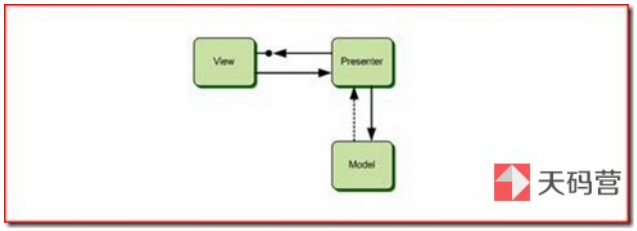
具体到Android App中，一般可以将App根据程序的结构进行纵向划分，根据MVP可以将App分别为模型层(M)，UI层(V)和逻辑层(P)。

UI层一般包括Activity，Fragment，Adapter等直接和UI相关的类，UI层的Activity在启动之后实例化相应的Presenter，App的控制权后移，由UI转移到Presenter，两者之间的通信通过BroadCast、Handler或者接口完成，只传递事件和结果。

举个简单的例子，UI层通知逻辑层（Presenter）用户点击了一个Button，逻辑层（Presenter）自己决定应该用什么行为进行响应，该找哪个模型（Model）去做这件事，最后逻辑层（Presenter）将完成的结果更新到UI层。

## **MVP的变种：Passive View

MVP的变种有很多，其中使用最广泛的是Passive View模式，即被动视图。在这种模式下，View和Model之间不能直接交互，View通过Presenter与Model打交道。Presenter接受View的UI请求，完成简单的UI处理逻辑，并调用Model进行业务处理，并调用View将相应的结果反映出来。View直接依赖Presenter，但是Presenter间接依赖View，它直接依赖的是View实现的接口。




相对于View的被动，那Presenter就是主动的一方。对于Presenter的主动，有如下的理解：

- Presenter是整个MVP体系的控制中心，而不是单纯的处理View请求的人；
- View仅仅是用户交互请求的汇报者，对于响应用户交互相关的逻辑和流程，View不参与决策，真正的决策者是Presenter；
- View向Presenter发送用户交互请求应该采用这样的口吻：“我现在将用户交互请求发送给你，你看着办，需要我的时候我会协助你”，不应该是这样：“我现在处理用户交互请求了，我知道该怎么办，但是我需要你的支持，因为实现业务逻辑的Model只信任你”；
- 对于绑定到View上的数据，不应该是View从Presenter上“拉”回来的，应该是Presenter主动“推”给View的；
- View尽可能不维护数据状态，因为其本身仅仅实现单纯的、独立的UI操作；Presenter才是整个体系的协调者，它根据处理用于交互的逻辑给View和Model安排工作。

## MVP架构存在的问题与解决办法

- 加入模板方法（Template Method）

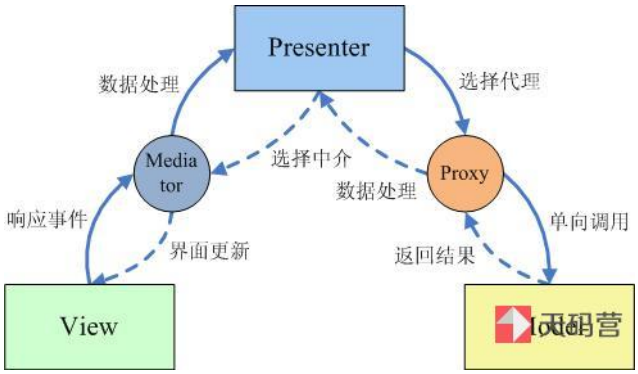
转移逻辑操作之后可能部分较为复杂的Activity内代码量还是不少，于是需要在分层的基础上再加入模板方法（Template Method）。
具体做法是在Activity内部分层。其中最顶层为BaseActivity，不做具体显示，而是提供一些基础样式，Dialog，ActionBar在内的内容，展现给用户的Activity继承BaseActivity，重写BaseActivity预留的方法。如有必要再进行二次继承，App中Activity之间的继承次数最多不超过3次。

- Model内部分层

模型层（Model）中的整体代码量是最大的，一般由大量的Package组成，针对这部分需要做的就是在程序设计的过程中，做好模块的划分，进行接口隔离，在内部进行分层。

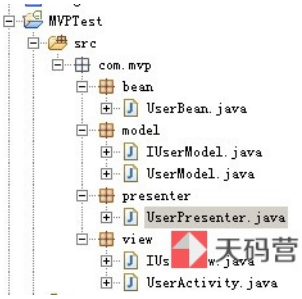
- 强化Presenter

强化Presenter的作用，将所有逻辑操作都放在Presenter内也容易造成Presenter内的代码量过大，对于这点，有一个方法是在UI层和Presenter之间设置中介者Mediator，将例如数据校验、组装在内的轻量级逻辑操作放在Mediator中；在Presenter和Model之间使用代理Proxy；通过上述两者分担一部分Presenter的逻辑操作，但整体框架的控制权还是在Presenter手中。Mediator和Proxy不是必须的，只在Presenter负担过大时才建议使用。
最终的架构如下图所示：




## MVP代码实例

我们来看看MVP在Android开发中是怎么应用的吧！！
我们用另一个例子来解释。
先来看包结构图




**建立Bean**

```
public class UserBean {     private String mFirstName;     private String mLastName;     public UserBean(String firstName, String lastName) {            this. mFirstName = firstName;            this. mLastName = lastName;     }     public String getFirstName() {            return mFirstName;     }     public String getLastName() {            return mLastName;     }}
```

**建立Model**

（处理业务逻辑，这里指数据读写），先写接口，后写实现

```
public interface IUserModel {     void setID(int id);     void setFirstName(String firstName);     void setLastName(String lastName);     int getID();     UserBean load(int id);// 通过id读取user信息,返回一个UserBean}
```

实现不在这里写了

**Presenter控制器**

建立presenter（主导器，通过iView和iModel接口操作model和view），activity可以把所有逻辑给presenter处理，这样java逻辑就从手机的activity中分离出来。

```
public class UserPresenter {     private IUserView mUserView;     private IUserModel mUserModel;     public UserPresenter(IUserView view) {            mUserView = view;            mUserModel = new UserModel();     }     public void saveUser( int id, String firstName, String lastName) {            mUserModel.setID(id);
mUserModel.setFirstName(firstName);            mUserModel.setLastName(lastName);     }     public void loadUser( int id) {
UserBean user = mUserModel.load(id);            mUserView.setFirstName(user.getFirstName()); // 通过调用IUserView的方法来更新显示
mUserView.setLastName(user.getLastName());     }}
```

**View视图**

建立view（更新ui中的view状态），这里列出需要操作当前view的方法，也是接口

```
public interface IUserView {     int getID();     String getFristName();     String getLastName();     void setFirstName(String firstName);     void setLastName(String lastName);}
```

activity中实现iview接口，在其中操作view，实例化一个presenter变量。

```
public class MainActivity extends Activity implements OnClickListener,IUserView {     UserPresenter presenter;     EditText id,first,last;     @Override     protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);           setContentView(R.layout. activity_main);           findViewById(R.id. save).setOnClickListener( this);           findViewById(R.id. load).setOnClickListener( this);           id = (EditText) findViewById(R.id. id);           first = (EditText) findViewById(R.id. first);           last = (EditText) findViewById(R.id. last);           presenter = new UserPresenter( this);     }     @Override     public void onClick(View v) {           switch (v.getId()) {           case R.id. save:                presenter.saveUser(getID(), getFristName(), getLastName());
break;           case R.id. load:                presenter.loadUser(getID());                break;           default:
break;          }     }     @Override     public int getID() {           return new Integer( id.getText().toString());     }     @Override     public String getFristName() {           return first.getText().toString();     }     @Override     public String getLastName() {           return last.getText().toString();     }     @Override     public void setFirstName(String firstName) {
first.setText(firstName);     }     @Override     public void setLastName(String lastName) {           last.setText(lastName);
}}
```

因此，Activity及从MVC中的Controller中解放出来了，这会Activity主要做显示View的作用和用户交互。每个Activity可以根据自己显示View的不同实现View视图接口IUserView。

通过对比同一实例的MVC与MVP的代码，可以证实MVP模式的一些优点：

- 在MVP中，Activity的代码不臃肿；
- 在MVP中，Model(IUserModel的实现类)的改动不会影响Activity(View)，两者也互不干涉，而在MVC中会；
- 在MVP中，IUserView这个接口可以实现方便地对Presenter的测试；
- 在MVP中，UserPresenter可以用于多个视图，但是在MVC中的Activity就不行。

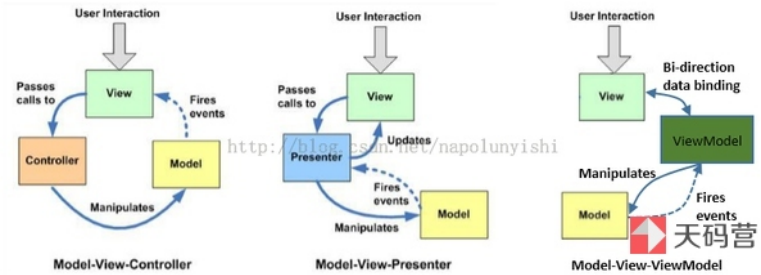
# 4.MVC、MVP与MVVM的关系

首先介绍下MVVM。

## MVVM

MVVM可以算是MVP的升级版，其中的VM是ViewModel的缩写，ViewModel可以理解成是View的数据模型和Presenter的合体，ViewModel和View之间的交互通过Data Binding完成，而Data Binding可以实现双向的交互，这就使得视图和控制层之间的耦合程度进一步降低，关注点分离更为彻底，同时减轻了Activity的压力。

在比较之前，先从图上看看三者的异同。




刚开始理解这些概念的时候认为这几种模式虽然都是要将view和model解耦，但是非此即彼，没有关系，一个应用只会用一种模式。后来慢慢发现世界绝对不是只有黑白两面，中间最大的一块其实是灰色地带，同样，这几种模式的边界并非那么明显，可能你在自己的应用中都会用到。实际上也根本没必要去纠结自己到底用的是MVC、MVP还是MVVP，不管黑猫白猫，捉住老鼠就是好猫。

## MVC->MVP->MVVM演进过程

MVC -> MVP -> MVVM 这几个软件设计模式是一步步演化发展的，MVVM 是从 MVP 的进一步发展与规范，MVP 隔离了MVC中的 M 与 V 的直接联系后，靠 Presenter 来中转，所以使用 MVP 时 P 是直接调用 View 的接口来实现对视图的操作的，这个 View 接口的东西一般来说是 showData、showLoading等等。M 与 V已经隔离了，方便测试了，但代码还不够优雅简洁，所以 MVVM 就弥补了这些缺陷。在 MVVM 中就出现的 Data Binding 这个概念，意思就是 View 接口的 showData 这些实现方法可以不写了，通过 Binding 来实现。

## 同

如果把这三者放在一起比较，先说一下三者的共同点，也就是Model和View：

- Model：数据对象，同时，提供本应用外部对应用程序数据的操作的接口，也可能在数据变化时发出变更通知。**Model不依赖于View的实现**，只要外部程序调用Model的接口就能够实现对数据的增删改查。
- View：UI层，提供对最终用户的交互操作功能，包括UI展现代码及一些相关的界面逻辑代码。

## 异

三者的差异在于如何粘合View和Model，实现用户的交互操作以及变更通知

- Controller

Controller接收View的操作事件，根据事件不同，或者调用Model的接口进行数据操作，或者进行View的跳转，从而也意味着一个Controller可以对应多个View。Controller对View的实现不太关心，只会被动地接收，Model的数据变更不通过Controller直接通知View，通常View采用观察者模式监听Model的变化。

- Presenter

Presenter与Controller一样，接收View的命令，对Model进行操作；与Controller不同的是Presenter会反作用于View，Model的变更通知首先被Presenter获得，然后Presenter再去更新View。一个Presenter只对应于一个View。根据Presenter和View对逻辑代码分担的程度不同，这种模式又有两种情况：Passive View和Supervisor Controller。

- ViewModel

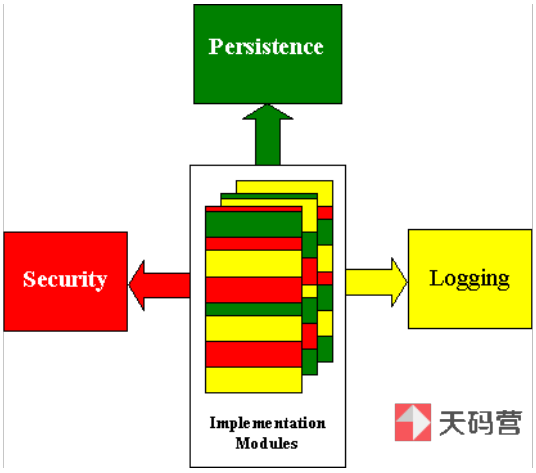
注意这里的“Model”指的是View的Model，跟MVVM中的一个Model不是一回事。所谓View的Model就是包含View的一些数据属性和操作的这么一个东东，这种模式的关键技术就是数据绑定（data binding），View的变化会直接影响ViewModel，ViewModel的变化或者内容也会直接体现在View上。这种模式实际上是框架替应用开发者做了一些工作，开发者只需要较少的代码就能实现比较复杂的交互。

## 一点心得

MVP和MVVM完全隔离了Model和View，但是在有些情况下，数据从Model到ViewModel或者Presenter的拷贝开销很大，可能也会结合MVC的方式，Model直接通知View进行变更。在实际的应用中很有可能你已经在不知不觉中将几种模式融合在一起，但是为了代码的可扩展、可测试性，必须做到模块的解耦，不相关的代码不要放在一起。网上有一个故事讲，一个人在一家公司做一个新产品时，一名外包公司的新员工直接在View中做了数据库持久化操作，而且一个hibernate代码展开后发现竟然有几百行的SQL语句，搞得他们惊讶不已，一时成为笑谈。
个人理解，在广义地谈论MVC架构时，并非指本文中严格定义的MVC，而是指的MV*，也就是视图和模型的分离，只要一个框架提供了视图和模型分离的功能，我们就可以认为它是一个MVC框架。在开发深入之后，可以再体会用到的框架到底是MVC、MVP还是MVVM。

# 5. 基于AOP的框架设计

AOP(Aspect-Oriented Programming, 面向切面编程)，诞生于上个世纪90年代，是对OOP(Object-Oriented Programming, 面向对象编程)的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（Cross-Cutting）代码，在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。而AOP技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。




## 5.1 AOP在Android中的使用

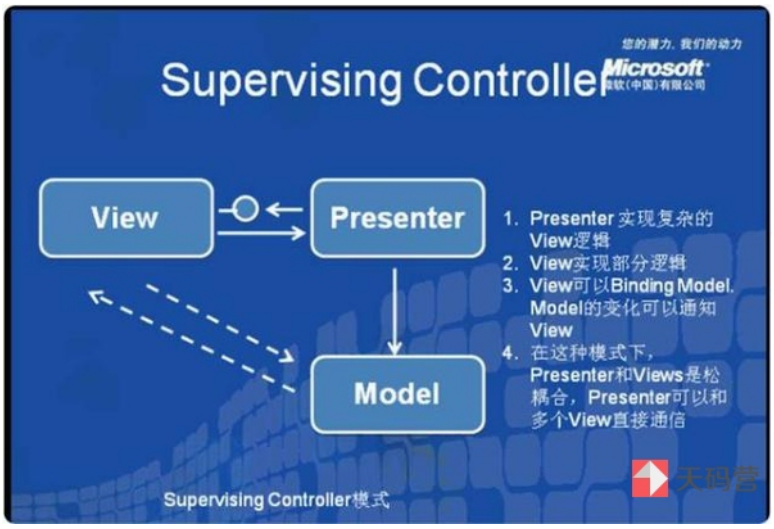
AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。在Android App中，哪些是我们需要的横切关注点？个人认为主要包括以下几个方面：Http, SharedPreferences, Json, Xml, File, Device, System, Log, 格式转换等。Android App的需求差别很大，不同的需求横切关注点必然是不一样的。一般的App工程中应该有一个Util Package来存放相关的切面操作，在项目多了之后可以将其中使用较多的Util封装为一个Jar包供工程调用。
在使用MVP和AOP对App进行纵向和横向的切割之后，能够使得App整体的结构更清晰合理，避免局部的代码臃肿，方便开发、测试以及后续的维护。

# 6. 干货：AndroidApp架构的设计经验

首先是作者最最喜欢的一句话，也是对创业公司特别适用的一句话，也是对不要过度设计的一种诠释：
先实现，再重构吧。直接考虑代码不臃肿得话，不知道什么时候才能写好了
先实现，再重构吧。直接考虑代码不臃肿得话，不知道什么时候才能写好了
先实现，再重构吧。直接考虑代码不臃肿得话，不知道什么时候才能写好了
（重要的事情说三遍）

## 6.1 整体架构

代码和文档规范，根据需求进行模块划分，确定交互方式，形成接口文档，这些较为通用的内容不再细说。做Android App时，一般将App进行纵向和横向的划分。纵向的App由UI层，逻辑层和模型层构成，整体结构基于MVP思想(图片来自网络)。




UI层内部多用模板方法，以Activity为例一般有BaseActivity，提供包括一些基础样式，Dialog，ActionBar在内的内容，展现的Activity都会继承BaseActivity并实现预留的接口，Activity之间的继承不超过3次；为避免Activity内代码过多，将App的整体控制权后移，也借鉴了IOC做法，大量的逻辑操作放在逻辑层中，逻辑层和UI层通过接口或者Broadcast等实现通信，只传递结果。一般Activity里的代码量都很大，通过这两种方式一般我写的单个Activity内代码量不超过400行。

逻辑层实现的是绝大部分的逻辑操作，由UI层启动，在内部通过接口调用模型层的方法，在逻辑层内大量使用了代理。打个比方，UI层告诉逻辑层我需要做的事，逻辑层去找相应的人(模型层)去做，最后只告诉UI这件事做的结果。

模型层没什么好说的，这部分一般由大量的Package组成，代码量是三层中最大的，需要在内部进行分层。

横向的分割依据AOP面向切面的思想，主要是提取出共用方法作为一个单独的Util，这些Util会在App整体中穿插使用。很多人的App都会引入自己封装的Jar包，封装了包括文件、JSON、SharedPreference等在内的常用操作，自己写的用起来顺手，也大幅度降低了重复作业。

这样纵，横两次对于App代码的分割已经能使得程序不会过多堆积在一个Java文件里，但靠一次开发过程就写出高质量的代码是很困难的，趁着项目的间歇期，对代码进行重构很有必要。

## 6.2 类库的使用

现在有很多帮助快速开发的类库，活用这些类库也是避免代码臃肿和混乱的好方法，下面给题主推荐几个常用类库。

减少Activity代码量的依赖注入框架ButterKnife:

https://github.com/JakeWharton/butterknife

简化对于SQlite操作的对象关系映射框架OrmLite:

https://github.com/j256/ormlite-android

图片缓存类库Fresco(by facebook):

https://github.com/facebook/fresco

能用第三方库就用第三方库。别管是否稳定，是否被持续维护，因为，任何第三方库的作者，都能碾压刚入门的菜鸟，你绝对写不出比别人更好的代码了。

最后附上知乎上面点赞次数很高的一段话：

> 如果“从零开始”，用什么设计架构的问题属于想得太多做得太少的问题。
> 从零开始意味着一个项目的主要技术难点是基本功能实现。当每一个功能都需要考虑如何做到的时候，我觉得一般人都没办法考虑如何做好。
> 因为，所有的优化都是站在最上层进行统筹规划。在这之前，你必须对下层的每一个模块都非常熟悉，进而提炼可复用的代码、规划逻辑流程。
> 所以，如果真的是从零开始，别想太多了

参考文献：


1、http://blog.csdn.net/luyi325xyz/article/details/43085409
2、http://blog.csdn.net/napolunyishi/article/details/22722345
3、https://www.zhihu.com/question/27645587/answer/37579829
4、http://www.jcodecraeer.com/a/anzhuokaifa/androidkaifa/2015/0202/2397.html
5、http://www.tuicool.com/articles/VJZrYb
6、https://www.zhihu.com/question/27645587
7、http://blog.csdn.net/luyi325xyz/article/details/43482123
8、https://www.zhihu.com/question/30976423
9、http://www.jcodecraeer.com/a/anzhuokaifa/androidkaifa/2015/0313/2599.html
10、http://www.2cto.com/kf/201506/405766.html
11、https://www.zhihu.com/question/19766132
12、http://www.cnblogs.com/artech/archive/2010/03/25/1696205.html
13、https://segmentfault.com/a/1190000003927200
14、http://blog.csdn.net/knxw0001/article/details/39637273