**Android开发中的MVP架构以及性能优化**

[](https://www.jianshu.com/u/70c12759d4fe)

[小楠总](https://www.jianshu.com/u/70c12759d4fe) 关注

2017.02.28 16:33\* 字数 2741 阅读 6012评论 8喜欢 32

**前言**

为什么要做架构设计？  
一个APP越做越大的时候，随着业务、需求越来越复杂，为了系统的扩展性更好，这时候就需要考虑架构问题。  
当然，小公司里面基本不会涉及到这些，一是因为项目比较小，主要是完成功能为主，而且需求不多，需要修改的地方不多；二是很有可能你做完这个项目之后就不干了，根本就不用考虑以后的扩展。因此掌握诸如架构设计、性能优化、NDK（Java不能解决，效率低，安全性不好）、RN（兼顾了性能又能即使更新）等知识，是我们通往大公司或者在大公司中充分发挥所必须掌握的。  
我们要不断地扩展自己的编程视野，不能说什么都没搞过，这样的话即使天资再厉害也不行。另外我们还需要往底层研究，否则的话容易被淘汰。

**MVP概念篇**

聊到MVP的时候首先我们会聊到MVC，MVC的出现就是为了解决诸如Android开发之类的有界面的编程。随着程序的功能、需求不断增加，依然要保持架构的清晰、可扩展性。MVC最先是由微软提出来的，因此我放出下面这种图：

MVC.jpeg

在MVC中：Model和View代表着业务逻辑与展示方式，Model和View往往是互相引用，改变展示方式的时候很有可能也要修改业务逻辑层，即修改Controller。

因此为了去除这种弊端，需要做解耦，我们的MVP应运而生，至于后面还有MVVM之类的，不在我们的讨论范围之内。

在MVP中，我们先看一下这三者的概念：  
Model：业务逻辑，工作职责是：加载数据。  
View：视图，工作职责是：控制显示数据的方式。  
Presenter：中间者，绑定Model、View。

**注意：在Android中，Activity往往当成是View的实现类。**

MVP的架构图如下所示：

MVP.png

虽然MVP的使用比较麻烦了些，但是它的有点也是很明显的，Model和View充分解耦，修改一个不会牵扯到另外一个。  
另外，视图、业务逻辑也有可能会变，因此视图、业务逻辑抽取成接口，改变不同的实现类即可。 Presenter中只持有Model和VIew的引用，可以随时更换它们的实现类，从而实现对扩展是开放的。  
因此MVP相对于MVC来说，规范比较明确，在系统架构上扩展性更加强。

**栗子篇**

举个栗子.jpeg

好了，上面说了那么多，然并卵？没关系，我们举个栗子，上代码哈！

先瞄一眼我们整个栗子demo的项目架构，我们姑且把这个demo叫做栗子一号吧。

Demo的MVP架构.png

下面我们分步骤来介绍。

1、作为一个APP，界面的显示需要数据，因此我们需要先有数据。我们先创建一个包，专门放Model，如此类推。因为在MVP中，我们的数据以及显示都是通过接口的方式来实现的，因此我们需要创建接口：IMainModel.java，前面的大写字母I代表接口类型。  
public interface IMainModel {

void loadData(OnLoadCompleteListener listener);

interface OnLoadCompleteListener {

void onComplete(String data);

}

}

IMainModel接口主要负责加载数据，并且在加载完成的时候回调。

2、然后我们需要实现IMainModel接口，我们先做第一个版本，从本地加载数据。

public class MainModelImpl implements IMainModel {

@Override

public void loadData(OnLoadCompleteListener listener) {

String data = "我是从本地加载的数据";

listener.onComplete(data);

}

}

3、接着我们需要View的接口，在这里我们定义了两个比较简单的方法，一个是加载数据的时候显示的进度条，然后是显示加载出来的数据。

public interface IMainView {

void showLoading();

void showData(String data);

}

4、在上面的概念中提到，我们的Activity是作为View的实现类的，因此Activity需要实现View的接口，并且实现View接口的抽象方法。在这里，为了简单起见，假设我们的数据是在Model中通过网络加载的，加载中的时候我们显示一个Toast，加载成功的时候我们把数据显示在TextView上。

public class MainActivity extends AppCompatActivity implements IMainView {

private MainPresenter mPresenter;

private TextView tv\_test;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

tv\_test = (TextView) findViewById(R.id.tv\_test);

}

@Override

public void showLoading() {

Toast.makeText(this, "正在拼命加载中。。。", Toast.LENGTH\_SHORT).show();

}

@Override

public void showData(String data) {

tv\_test.setText(data);

}

}

5、走完上面的流程之后，我们需要有一个中间者Presenter，绑定View（Activity）以及我们的Model，如下所示。Presenter是一个类，它需要持有View以及Model的接口，而不是具体实现。在Presenter构造的时候初始化Model对象，接收View对象（Activity）。最后提供一个fetch方法，绑定二者，执行具体的业务逻辑，这里不再赘述。

public class MainPresenter {

private IMainModel mModel;

private IMainView mView;

public MainPresenter(IMainView view) {

mModel = new MainModelImpl();

mView = view;

}

public void fetch() {

mView.showLoading();

mModel.loadData(new IMainModel.OnLoadCompleteListener() {

@Override

public void onComplete(String data) {

mView.showData(data);

}

});

}

}

6、最后，在Activity的onCreate方法中实例化我们的Presenter对象。

public class MainActivity extends AppCompatActivity implements IMainView {

private MainPresenter mPresenter;

private TextView tv\_test;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

tv\_test = (TextView) findViewById(R.id.tv\_test);

//实例化Presenter

mPresenter = new MainPresenter(this);

mPresenter.fetch();

}

@Override

public void showLoading() {

Toast.makeText(this, "正在拼命加载中。。。", Toast.LENGTH\_SHORT).show();

}

@Override

public void showData(String data) {

tv\_test.setText(data);

}

}

下面瞄一下栗子一号的效果吧：

栗子一号.png

**MVP在系统扩展中的作用**

上面只是介绍了MVP的基本使用，下面需要介绍的才是在项目中使用MVP的时候最屌的地方。  
随着我们栗子一号的不断成长，迭代更新，发展壮大，我们的项目组提出了新的需求。

**需求1：我们的数据不能在本地加载了，需要改为通过在网络中加载。**

我们先分析一下MVP架构，我们加载数据的逻辑是通过Model的实现类来实现的，因此直接通过替换不同Model实现类就可以实现这一个需求。这就是面向对象的OCP原则，就是程序对修改关闭，扩展开放。  
好了，废话不多说，是时候对栗子一号进行手术了。

根据我们的分析，直接创建一个新的Model实现类即可，我们起名为MainModeNetlImpl，修改loadData的业务逻辑，改为从网络加载数据，。实际项目中可能是一些很复杂的代码，所以这里为了说面MVP的优点，通过简单的栗子来说明。如果项目中没有MVP或者MVC架构的话，需要修改的时候就会比较麻烦了，MVP（MVC）的分层思想使得我们的项目整体架构比较清晰。

public class MainModeNetImpl implements IMainModel {

@Override

public void loadData(OnLoadCompleteListener listener) {

String data = "我是从网络加载的数据";

listener.onComplete(data);

}

}

然后在我们的Presenter中替换掉我们的Model实现类即可，如下所示。

public class MainPresenter {

private IMainModel mModel;

private IMainView mView;

public MainPresenter(IMainView view) {

// mModel = new MainModelImpl();

mModel = new MainModeNetlImpl();

mView = view;

}

public void fetch() {

mView.showLoading();

mModel.loadData(new IMainModel.OnLoadCompleteListener() {

@Override

public void onComplete(String data) {

mView.showData(data);

}

});

}

}

然后这是栗子脱变之后的栗子二号：

栗子二号.png

**扩展：当然，如果你的项目经理脑抽了，改来改去，一时从网络加载，一时又觉得不好又从本地加载；或者程序在不同的情况之下需要从不同的地方加载数据。这里我们就可以使用策略模式，根据不同的情况来从不同的地方加载数据。所以我们可以在Presenter中作如下修改：**

public MainPresenter(IMainView view, boolean isFromNet) {

if (isFromNet) {

mModel = new MainModeNetImpl();

} else {

mModel = new MainModelImpl();

}

mView = view;

}

好吧，也许过了几天项目经理又抽风了（嘘，不要说太大声），又要改改改！

**需求2：我们的数据展示方法需要改变了，比如说要把数据展示到不同的控件上面等等，那么我们可以直接新建一个不同的Activity，实现View的接口，作出相应的修改即可，这里不再赘述了。**

**MVP提高篇**

在上面使用MVP中会有一些问题：

**问题1：每次使用都要手动newPresenter，写很多重复的代码，能否抽取出基类MVPBaseActivity，简化我们的代码？**

**问题2：上面的写法中会不会带来内存泄漏的问题？**

针对问题2，我在这里作出一些分析：我们的View（Activity）是持有Presenter的引用的，而Presenter中又持有Model的引用，这样就会形成一条引用链，如下图所示：

引用链.png

我们的Model去加载数据的时候可能是十分耗时间的，一旦加载过程中Activity销毁了，那么就会造成内存泄漏问题。比如说我们的栗子中，正在加载数据的时候，用户觉得不爽直接按下返回键销毁Activity了，但是因为存在上面的引用链，Activity是不能够被正常被回收的。

为了模拟这一个现象，我们在Model加载数据的时候通过Handler去做延时5秒，在加载过程中旋转屏幕，使得Activity重建，代码如下所示：

public class MainModeNetImpl implements IMainModel {

private Handler mHandler = new Handler();

@Override

public void loadData(final OnLoadCompleteListener listener) {

mHandler.postDelayed(new Runnable() {

@Override

public void run() {

String data = "我是从网络加载的数据";

listener.onComplete(data);

}

}, 5000);

}

}

在加载过程中旋转屏幕，Activity重建，通过Android Studio的Memory Monitor来Dump内存信息，我们可以看到，Activity的确不能够被正常回收，内存中存在两个MainActivity，如下图所示：

栗子内存快照.png

**解决办法**

前方高能，请系好安全带，老哥，稳！

为了解决内存泄漏的问题，我们在Activity创建的时候创建Presenter，在销毁的时候解除绑定。

public abstract class MVPBaseActivity<V, P extends BasePresenter<V>> extends AppCompatActivity {

protected P mPresenter;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

//创建Presenter

mPresenter = createPresenter();

//关联View

mPresenter.attachView((V) this);

}

@Override

protected void onDestroy() {

super.onDestroy();

mPresenter.detachView();

}

protected abstract P createPresenter();

}

对应的BasePresenter代码如下：

public abstract class BasePresenter<V> {

protected WeakReference<V> mViewRef;

public void attachView(V view) {

mViewRef = new WeakReference<>(view);

}

public void detachView() {

if (mViewRef != null) {

mViewRef.clear();

mViewRef = null;

}

}

protected V getView() {

return mViewRef.get();

}

}

通过上面的基类抽取就实现类Model与View的绑定以及解绑，即实现了两者生命周期的关联。并且在内存不足的时候，先干掉Model，后干掉View，给用户一个好的体验效果。

以上就是我们今天要讲述的MVP架构以及扩展，在这个过程当中，我们不当掌握了MVP的架构，还巩固了面向对象的OCP原则，设计模式中的策略模式，内存泄漏相关的知识等等。所以说知识是一个整体的体系，互相关联的，存在必有存在的意义。

另外，我在写博客的时候画了图，在面试的时候我们也许会经常被问到：源码，原理，机制性的东西，这时候我们该画图的时候画图，该比喻的比喻，画给面试官看，面试官一定会觉得焕然一新的。

如果觉得我的文字对你有所帮助的话，欢迎关注我的公众号：

公众号：Android开发进阶

我的群欢迎大家进来探讨各种技术与非技术的话题，有兴趣的朋友们加我私人微信**huannan88**，我拉你进群**交（♂）流（♀）**。