# 用MVP架构开发Android应用

2015-11-09

## 摘要

本文原创，转载请注明地址：http://kymjs.com/code/2015/11/09/01 (http://kymjs.com/code/2015/11/09/01)
怎样从架构级别去搭建一个APP，怎样让他应对日益更改的界面与业务逻辑？今天为大家讲述一种在Android上实现MVP模式的方法。



今天为大家讲述一种在Android上实现MVP模式的方法。也是我从新项目中总结出来的一种新的架构模式，大家可以查看我的TheMVP项目：https://github.com/kymjs/TheMVP (https://github.com/kymjs/TheMVP)

# 为什么需要MVP

关于什么是MVP，以及MVC、MVP、MVVM有什么区别，这类问题网上已经有很多的讲解，你可以自行搜索或看看文末的参考文章，这里就只讲讲为什么需要MVP。
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。但是，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，以致很容易变得庞大而臃肿。
越小的类，bug越不容易出现，越容易调试，更容易测试，我相信这一点大家是都赞同的。在MVP模式下，View和Model是完全分离没有任何直接关联的(比如你在View层中完全不需要导Model的包，也不应该去关联它们)。
使用MVP模式能够更方便的帮助Activity(或Fragment)职责分离，减小类体积，使项目结构更加清晰。

# 现有的MVP方案

GitHub上有一个开源项目 `AndroidMVP` (https://github.com/antoniolg/androidmvp)，其思想是通过将Activity或Fragment看做View，并单独采用has...a...关系包含一个Presenter类的方式实现的，这是一种可行的技术方案。

**AndroidMVP使用示例**

完整Demo请查看这里
(https://github.com/antoniolg/androidmvp/tree/master/app/src/main/java/com/antonioleiva/mvpexample/app/Login)
首先需要定义一个View层接口，让View实现类Activity(Fragment)实现；
其次需要定义一个Presenter实现接口，让Presenter实现类实现；
在View实现类Activity(Fragment)中包含Presenter对象，并在Presenter创建的时候传一个View对象；
在Presenter中通过构造时传入的视图层对象操作View

```
public interface LoginView {
    public void showProgress();
    public void hideProgress();
    public void setUsernameError();
    public void setPasswordError();
}

public class A extends Activity implements LoginView, OnClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // ...
        // 省略初始化控件
        // ...
        presenter = new LoginPresenterImpl(this);
    }
    //...省略众多接口方法
}

public class LoginPresenterImpl implements LoginPresenter, OnLoginFinishedListener {
    private LoginView loginView;

    public LoginPresenterImpl(LoginView loginView) {
        this.loginView = loginView;
    }

    @Override public void validateCredentials(String username, String password) {
        loginView.showProgress();
        //做逻辑操作
    }
}
```

**AndroidMVP存在的问题**

但是在用的时候会出现一些问题：
1. 例如当应用进入后台且内存不足的时候，系统是会回收这个Activity的。通常我们都知道要用 `OnSaveInstanceState()` 去保存状态，用 `OnRestoreInstanceState()` 去恢复状态。 但是在我们的MVP中，View层是不应该去直接操作Model的，这样做不合理，同时也增大了M与V的耦合。
2. 界面复用问题。通常我们在APP最初版本中是无法预料到以后会有什么变动的，例如我们最初使用一个Fragment去作为界面的显示，后来在版本变动中发现这个Fragment越来越庞大，而Fragment的生命周期又太过复杂造成很多难以理解的BUG，我们需要把这个界面放到一个Activity中实现，这时候就麻烦了，要把Fragment转成Activity，这可不仅仅是改改类名的问题，更多的是一大堆生命周期需要去修改

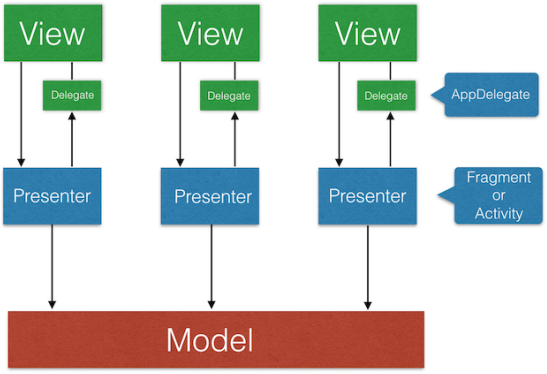
个Activity中实现。这时候就麻烦了，要把Fragment转成Activity，这可不仅仅是改改类名的问题，更多的是一个人维生命周期需要去修改。
例如参考文章2中的译者就遇到过这样的问题。
3. Activity本身就是Android中的一个Context。不论怎么去封装，都难以避免将业务逻辑代码写入到其中。

# 解决现有方案的问题

既然知道了这些问题，我们的解决办法自然是不要将Activity作为View层而去单独包含Presenter类进来。反过来，我们将Activity(Fragment)作为Presenter层的代码，包含一个View层的类来。如果你同时是一名IOS开发者，你一定会很熟悉，这不就是ViewController和APPDelegate吗。
使用Activity作为Presenter的优点就在于，可以原封不动的使用Activity本身的生命周期去处理项目逻辑，而不需要强加给另一个包含类，甚至记忆额外自定义的生命周期。
而同时作为独立的View层，我们的视图可以原封不动的传递给Presenter(不管是Activity或者Fragment)，而不需要改任何代码。对于一个开发团队，完全可以将View层的东西交给一个人编写，而将业务实现交给另一个人编写。而随着逻辑变化对View的更改，只需要通过Presenter层的包含一个代理对象————— `ViewDelegate` 来操作相应的更改方法就够了。

# TheMVP原理介绍

与传统androidMVP不同(原因上文已经说了)，TheMVP使用Activity作为Presenter层来处理代码逻辑，通过让Activity包含一个ViewDelegate对象来间接操作View层对外提供的方法，从而做到完全解耦视图层。如下图：




# TheMVP代码说明

要将Activity作为Presenter来写，需要让View变得可复用，必须解决的一个问题就是setContentView()如何调用，因为它是Activity(Fragment有类似)的方法。
我们需要把视图抽离出来独立实现。可以定义一个接口，来限定View层必须实现的方法(这个接口定义，也就是View层的代理对象)，例如：

```
public interface IDelegate {
    void create(LayoutInflater i, ViewGroup v, Bundle b);
    View getRootView();
}
```

首先通过inflater一个布局，将这个布局转换成View，再用getRootView()方法把这个View返回给Presenter层，让setContentView(view)去调用，这样就实现了rootView的独立。
所以，在Presenter层，我们的实现应该是：

```
protected void onCreate(Bundle savedInstanceState) {
    //获取到视图层对象
    IDelegate viewDelegate = xxx;
    //让视图层初始化(如果是Fragment，就需要传递onCreateView方法中的三个参数)
    viewDelegate.create(getLayoutInflater(), null, savedInstanceState);
    //拿到初始化以后的rootview，并设置content
    setContentView(viewDelegate.getRootView());
}
```

# 结合DataBinding

一个好的架构一定是对扩展开放，对修改关闭的，这是软件设计模式的开闭原则。
如果你之前有了解过Google的DataBinding，你一定知道ViewModel的概念。DataBinding 解决了 Android UI 编程中的一个痛点，就是要给一个控件设置内容，必须首先获取到控件的对象，并调用set方法(例如setText())，传一个数据进去。
DataBinding允许你使用这样的代码为控件设置内容。

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{user.lastName}" />
```

其中user.lastName表示在项目中的数据类的user对象的lastName属性。

**DataBinding存在的问题**

但是使用 Data Binding 之后，xml的布局文件就不再单纯地展示 UI 元素，还需要定义 UI 元素用到的变量。所以，它的根节点不再是一个 `ViewGroup` ，而是变成了 `layout` ，并且新增了一个节点 `data` 。

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
```

```
        <variable name="user" type="com.xxx.User" />
    </data>
    <!--原先的根节点（Root Element）-->
    <LinearLayout>
    ....
    </LinearLayout>
</layout>
```

然后还必须在 `onCreate()` 方法中，用 `DatabindingUtil.setContentView()` 来替换掉 `setContentView()` ，然后创建一个 user 对象，通过 `binding.setUser(user)` 与 variable 进行绑定。

```
protected void onCreate(Bundle savedInstanceState) {
    ActivityBasicBinding binding = DataBindingUtil.setContentView(
            this, R.layout.activity_basic);
    User user = new User();
    binding.setUser(user);
}
```

虽然简化了试图与数据绑定的代码，但同时也牺牲了布局文件的可复用性。例如我们经常会遇到的，一个Fragment与另一个Fragment，在布局上完全一样，而仅仅是数据不同，这时我们通常会用代码去控制在不同的界面显示不同的数据。然而，如果将数据写死在xml中，就失去了布局的复用性。
因此，我们可以尝试将视图与数据模型绑定的逻辑抽出，单独建立一个类来使用代码控制，这样如果发生相同的界面复用，只需要重写视图与数据绑定的逻辑就够了，其他的代码仍然不变。

**为Presenter添加ViewModel的功能**

定义一个ViewModel层的接口，其中包含两个泛型分别为View层的代理和Model层的代理：

```
public interface DataBinder<T extends IDelegate, D extends IModel> {
    /**
     * 将数据与View绑定，这样当数据改变的时候，框架就知道这个数据是和哪个View绑定在一起的，就可以自动改变ui
     * 当数据改变的时候，会回调本方法。
     *
     * @param viewDelegate 视图层代理
     * @param data         数据模型对象
     */
    void viewBindModel(T viewDelegate, D data);
}
```

为我们之前写好的Presenter添加扩展，使它能够支持DataBinder。其中使用getDataBinder()方法，得到开发中具体的某个界面的ViewModel层的扩展，然后我们只需要在数据改变的时候，手动调用notifyModelChanged()方法，即可使ViewModel中定义的绑定逻辑生效：

```
public abstract class DataBindActivity<T extends IDelegate> extends
        ActivityPresenter<T> {
    protected DataBinder binder;

    public abstract DataBinder getDataBinder();

    public <D extends IModel> void notifyModelChanged(D data) {
        binder.viewBindModel(viewDelegate, data);
    }
}
```

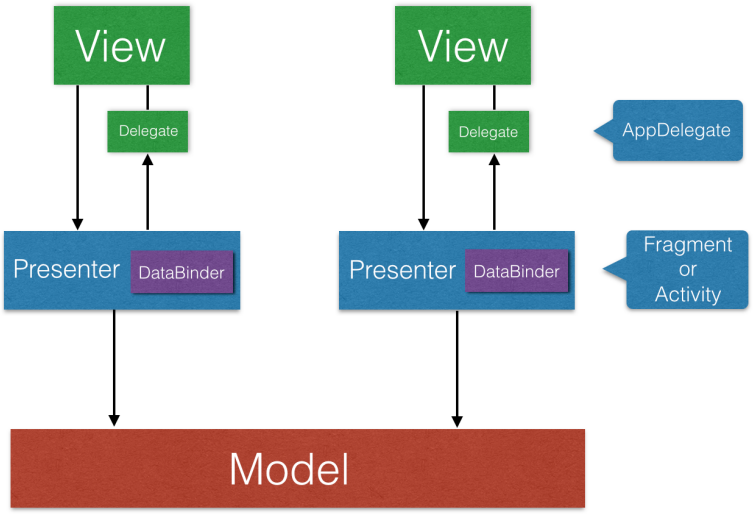
**双向绑定**

同时使用以代码控制的逻辑，能够轻松实现视图改变数据，数据改变视图的双向绑定。这是标准的ViewModel所一定会具备而现在的Beta版 `DataBinding` 没办法做到的功能。
刚刚的 `DataBinder` 接口已经实现的是 Model->View 的单向绑定，那么我们只需要为其添加一个 View-> Model 的方法，来让具体界面的ViewModel层实现类来解决其中的逻辑即可。

```
public interface DataBinder<T extends IDelegate, D extends IModel> {
    void viewBindModel(T viewDelegate, D data);

    /**
     * 将数据与View绑定，这样当view内容改变的时候，框架就知道这个View是和哪个数据绑定在一起的，就可以自动改变数据
     * 当ui改变的时候，会回调本方法。
     *
     * @param viewDelegate 视图层代理
     * @param data         数据模型对象
     */
    void modelBindView(T viewDelegate, D data);
}
```

也许你会有疑问，既然是MVP模式的项目，还又加ViewModel层，岂不是不伦不类？这里需要说明的是我们的架构目前仍旧是MVP模式的，你可以看做是在Presenter中，我们额外添加了一个方法，然后我们只在这个方法中写 `setText()`、 `setImageResource()` 这类对控件设值的方法。
此时，我们的项目结构如下图：




# 让MVP变得好用

**使用泛型解耦**

现在我们是实现了View与Presenter的解耦，在onCreate中包含了一个接口对象来实现我们固定的一些必须方法。但是又引入了问题：一些

特定方法没办法引用了。比如某个界面的设值、控件的修改显示逻辑对Presenter层的接口，接口对象必须强转成具体子类才能调用。
解决办法：可以通过泛型来解决直接引用具体对象的问题。比如我们可以在子类定义以后确定一个Presenter中所引用的Delegate的具体类型。例如：

```
public abstract class ActivityPresenter<T extends IDeLegate> extends Activity {
    protected T viewDelegate;

    protected void onCreate(Bundle savedInstanceState) {
        viewDelegate = getDelegateClass().newInstance();
    }

    protected abstract Class<T> getDelegateClass();
}
```

这样我们在ActivityPresenter的继承类中就可以通过动态设置getDelegateClass()的返回值来确定Delegate的具体类型了。

**注解初始化控件**

遗憾的是没办法使用编译时注解绑定控件，例如 `Butterknife` ；不过你依然可以使用运行时注解，例如 `afinal` 。不过也不推荐使用运行时注解，毕竟通过反射去初始化控件会很费时间。
当然，解决办法也是有的：就是通过定义findViewById()泛型类类型返回值，这样我们就不用写那又臭又长的函数名加强转了。

```
public <T extends View> T bindView(int id) {
    T view = (T) mViews.get(id);
    if (view == null) {
        view = (T) rootView.findViewById(id);
        mViews.put(id, view);
    }
    return view;
}

public <T extends View> T get(int id) {
    return (T) bindView(id);
}
```

**设置监听**

同时你也可以一次对多个控件设置监听事件，例如这样同时对button1,button2,button3设置监听器listener

```
viewDelegate.setOnClickListener(listener, R.id.button1, R.id.button2, R.id.button3);
```

它的内部实现也很简单，就是利用了变参函数

```
public void setOnClickListener(OnClickListener l, int... ids) {
        if (ids == null) {
            return;
        }
        for (int id : ids) {
            get(id).setOnClickListener(l);
        }
    }
```

**利用变长数组构建View集合**

由于Presenter在使用访问View的时候并不是直接调用，而是通过代理对象间接调用，如果我们在实现View层代码的时候有太多的控件需要被引用，可能就必须定义一大堆控件声明，会造成记忆负担。
这时候显然通过id去记忆更方便一些。我们可以使用 `SparseArray` 它是由两个数组来替代Map操作的类(如果你还是不知道他是干嘛的，可以简单的当成HashMap)。

结合上面的全部例子，可以为IDelegate接口定义一个抽象类，将全部的工具方法都集成进来

```
public abstract class AppDelegate implements IDelegate {
    protected final SparseArray<View> mViews = new SparseArray<View>();

    protected View rootView;

    public abstract int getRootLayoutId();

    @Override
    public void create(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        int rootLayoutId = getRootLayoutId();
        rootView = inflater.inflate(rootLayoutId, container, false);
    }

    @Override
    public View getRootView() {
        return rootView;
    }

    public <T extends View> T bindView(int id) {
        T view = (T) mViews.get(id);
        if (view == null) {
            view = (T) rootView.findViewById(id);
            mViews.put(id, view);
        }
        return view;
    }

    public <T extends View> T get(int id) {
        return (T) bindView(id);
    }

    public void setOnClickListener(View.OnClickListener listener, int... ids) {
        if (ids == null) {
            return;
        }
        for (int id : ids) {
            get(id).setOnClickListener(listener);
        }
    }
}
```

# 简单Demo

- 完整的Demo源码已经提交在了项目中，你可以在这里查看 (https://github.com/kymjs/TheMVP)，运行名为demo的module。
  这里仅取一个简单的示例。首先是View层的实现

```
public class SimpleDelegate extends AppDelegate {
    @Override
    public int getRootLayoutId() {
        return R.layout.delegate_simple;
    }
```

```
    @Override
    public void initWidget() {
        super.initWidget();
        TextView textView = get(R.id.text);
        textView.setText("在视图代理层创建布局");
    }

    public void setText(String text) {
        //get(id)等同于bindview(id)，从上文就可以看到了，get方法调用了bindview方法
        TextView textView = get(R.id.text);
        textView.setText(text);
    }
}
```

接着是Presenter层的实现

```
/**
 * 在这里做业务逻辑操作，通过viewDelegate操作View层控件
 */
public class SimpleActivity extends ActivityPresenter<SimpleDelegate> implements OnClickListener {

    @Override
    protected Class<SimpleDelegate> getDelegateClass() {
        return SimpleDelegate.class;
    }

    /**
     * 在这里写绑定事件监听相关方法
     */
    @Override
    protected void bindEvenListener() {
        super.bindEvenListener();
        viewDelegate.get(R.id.button1).setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.button1:
                viewDelegate.setText("你点击了button");
                break;
        }
    }
}
```

# 参考文章

《MVC, MVP, MVVM比较以及区别》 (http://www.cnblogs.com/JustRun1983/p/3727560.html)作者Justrun (http://weibo.com/1439307110/profile?s=6cm7D0)
《android实现MVP的新思路》 (https://github.com/bboyfeiyu/android-tech-frontier/tree/master/androidweekly/%E4%B8%80%E7%A7%8D%E5%9C%A8android%E4%B8%AD%E5%AE%9E%E7%8E%B0MVP%E6%A8%A1%E5%BC%8F%E7%9A%84%E6%96%B0%E6%80%9D%E8%B7%AF) 译者FTExplore (https://github.com/FTExplore)

# 补充

> 11月10日补充
> 感谢Rocko指出的问题，就是一个 View 需要几个 Model 的时候。这的确会有点麻烦，目前的想法是通过集合来解决。