**android架构设计—mvp模式封装**

2018年01月21日 14:10:52 [LBJFxd](https://me.csdn.net/fanxudonggreat) 阅读数：6921 标签： [mvp](http://so.csdn.net/so/search/s.do?q=mvp&t=blog)[架构设计](http://so.csdn.net/so/search/s.do?q=%E6%9E%B6%E6%9E%84%E8%AE%BE%E8%AE%A1&t=blog)[Android架构](http://so.csdn.net/so/search/s.do?q=Android%E6%9E%B6%E6%9E%84&t=blog)[MVP封装](http://so.csdn.net/so/search/s.do?q=MVP%E5%B0%81%E8%A3%85&t=blog) 更多

个人分类： [Android开发](https://blog.csdn.net/fanxudonggreat/article/category/6067844)

版权声明：本文为博主原创文章，转载请注明出处！ https://blog.csdn.net/fanxudonggreat/article/details/79120011

**简介**

关于Android程序的构架, 当前比较成熟且使用最多的应该就是MVP架构了,当然还有其他的如：MVC和MVVM。MVC相对于较为落后，、耦合度太高、职责不明确，MVVM使用DataBind，普及性不如MVP，况且Google官方提供了Sample代码来展示MVP模式的用法。因此选择MVP架构毫无疑问。

**概念**

MVP即Model、View、Presenter

View：负责视图部分展示、视图事件处理。Activity、Fragment、Dialog、ViewGroup等呈现视图的组件都可以承担该角色。

Model：负责数据的请求、解析、过滤等数据层操作。

Presenter:View和Model交互的桥梁。

**优势**

单一职责

Model、View、Presenter只处理某一类逻辑

解耦

Model层修改和View层修改互不影响

面向接口编程，依赖抽象

Presenter和View互相持有抽象引用，对外隐藏内部实现细节

**可能存在的问题**

Model进行异步操作，获取结果通过Presenter回传到View时，出现View引用的空指针异常

Presenter和View互相持有引用，解除不及时造成的内存泄漏。

因此，在进行MVP架构设计时需要考虑Presenter对View进行回传时，View是否为空？Presenter与View何时解除引用即Presenter能否和View层进行生命周期同步?

**实现**

1.首先定义两个抽象

public interface MvpView {

}

*/\*\**

*\*定义P层生命周期与V层同步*

*\*/*

public interface MvpPresenter<V extends MvpView> {

void onMvpAttachView(V view, Bundle savedInstanceState);

void onMvpStart();

void onMvpResume();

void onMvpPause();

void onMvpStop();

void onMvpSaveInstanceState(Bundle savedInstanceState);

void onMvpDetachView(boolean retainInstance);

void onMvpDestroy();

}

*/\*\**

*\* Presenter生命周期包装、View的绑定和解除，P层实现的基类*

*\*/*

public class BaseMvpPresenter<V extends MvpView> implements MvpPresenter<V> {

private WeakReference<V> viewRef;

protected V getView() {

return viewRef.get();

}

protected boolean isViewAttached() {

return viewRef != null && viewRef.get() != null;

}

private void \_attach(V view, Bundle savedInstanceState) {

viewRef = new WeakReference<V>(view);

}

@Override

public void onMvpAttachView(V view, Bundle savedInstanceState) {

\_attach(view, savedInstanceState);

}

@Override

public void onMvpStart() {

}

@Override

public void onMvpResume() {

}

@Override

public void onMvpPause() {

}

@Override

public void onMvpStop() {

}

@Override

public void onMvpSaveInstanceState(Bundle savedInstanceState) {

}

private void \_detach(boolean retainInstance) {

if (viewRef != null) {

viewRef.clear();

viewRef = null;

}

}

@Override

public void onMvpDetachView(boolean retainInstance) {

\_detach(retainInstance);

}

@Override

public void onMvpDestroy() {

}

}

正常情况下，开发一个项目的时候我们都会建一个BaseActivity，然后在里面做一些共同的View层相关操作，此时如果我们将P层的初始化、与View生命周期同步放在BaseActivity里面，很明显会违背软件设计原则的单一职责，且会增加耦合，不利于软件升级、灵活性太低。因此新建一个BaseMvpActivity继承于BaseActivity来处理P层相关。

*/\*\**

*\* 纯粹的 MVP 包装，不要增加任何View层基础功能*

*\* 如果要添加基类功能，请在{@link BaseActivity} 中添加*

*\*/*

public abstract class BaseMvpActivity<P extends MvpPresenter> extends BaseActivity implements MvpView {

protected P presenter;

protected abstract P createPresenter();

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

presenter = createPresenter();

if (presenter == null) {

throw new NullPointerException("Presenter is null! Do you return null in createPresenter()?");

}

presenter.onMvpAttachView(this, savedInstanceState);

}

@Override

protected void onStart() {

super.onStart();

if (presenter != null) {

presenter.onMvpStart();

}

}

@Override

protected void onResume() {

super.onResume();

if (presenter != null) {

presenter.onMvpResume();

}

}

@Override

protected void onPause() {

super.onPause();

if (presenter != null) {

presenter.onMvpPause();

}

}

@Override

protected void onStop() {

super.onStop();

if (presenter != null) {

presenter.onMvpStop();

}

}

@Override

protected void onSaveInstanceState(Bundle outState) {

super.onSaveInstanceState(outState);

if (presenter != null) {

presenter.onMvpSaveInstanceState(outState);

}

}

@Override

protected void onDestroy() {

super.onDestroy();

if (presenter != null) {

presenter.onMvpDetachView(false);

presenter.onMvpDestroy();

}

}

}

使用

*/\*\**

*\* P层与V层接口定义*

*\*/*

public class MainContract {

public interface IMainView extends MvpView {

*/\*\**

*\* 测试*

*\*/*

void setTestContent();

}

public interface IMainPresenter extends MvpPresenter<IMainView> {

*/\*\**

*\* 测试*

*\*/*

void requestTestContent();

}

}

public class MainActivity extends BaseMvpActivity<MainContract.IMainPresenter> implements MainContract.IMainView {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_photo\_picker);

presenter.requestTestContent();

}

@Override

protected MainContract.IMainPresenter createPresenter() {

return new MainPresenterImpl();

}

@Override

public void setTestContent() {

Log.i(getClass().getSimpleName(), "测试成功");

}

}

public class MainPresenterImpl extends BaseMvpPresenter<MainContract.IMainView> implements MainContract.IMainPresenter {

@Override

public void requestTestContent() {

*//先进行非空判断*

if (isViewAttached()) {

getView().setTestContent();

}

}

@Override

public void onMvpAttachView(MainContract.IMainView view, Bundle savedInstanceState) {

super.onMvpAttachView(view, savedInstanceState);

}

*/\*\**

*\*重写P层需要的生命周期，进行相关逻辑操作*

*\*/*

@Override

public void onMvpResume() {

super.onMvpResume();

}

}

**小结**

Mvp模式很好的将View层和Presenter解耦，View层和Presenter层的修改互不影响，并且符合软件设计原则之单一职责，提高了代码的灵活性和可扩展性。最后二者之间通过抽象进行关联，使之可以互相访问。