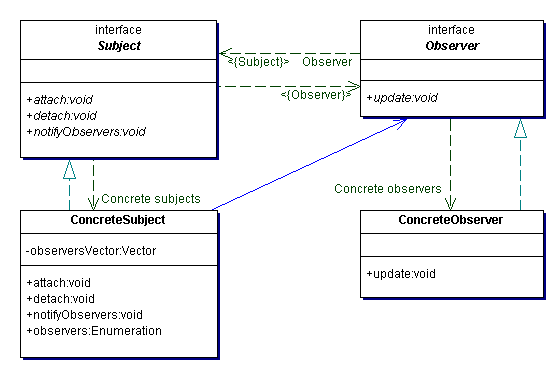
**观察者模式**

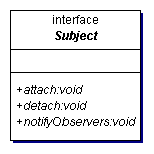
**一、观察者模式简介：**

观察者（Observer）模式是对象的行为型模式，又叫做发表-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-收听者（Source/Listener）模式或从属者（Dependents）模式。

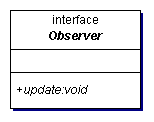
本模式的类图结构如下：



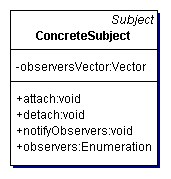
在观察者模式里有如下的角色：  
 **抽象主题（Subject）角色**：主题角色把所有的观察者对象的引用保存在一个列表里；每个主题都可以有任何数量的观察者。主题提供一个接口可以加上或撤销观察者对象；主题角色又叫做抽象被观察者(Observable)角色；



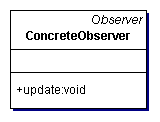
**抽象观察者（Observer）角色**：为所有的具体观察者定义一个接口，在得到通知时更新自己；



**具体主题（ConcreteSubject）角色**：保存对具体观察者对象有用的内部状态；在这种内部状态改变时给其观察者发出一个通知；具体主题角色又叫作具体被观察者角色；



**具体观察者（ConcreteObserver）角色**：保存一个指向具体主题对象的引用；和一个与主题的状态相符的状态。具体观察者角色实现抽象观察者角色所要求的更新自己的接口，以便使本身的状态与主题的状态自恰。



**二、开源代码链接**：

<https://github.com/youxin11544/MVP-RxJava-Hybride/blob/master/android_MVP_framwork/app/src/main/java/com/haibao/store/myapplication/base/BaseActivity.java>

1. **代码分析**：

**1.源代码：**

package com.haibao.store.myapplication.base;

import android.content.Intent;

import android.content.pm.ActivityInfo;

import android.os.Bundle;

import android.support.annotation.NonNull;

import android.support.v4.app.Fragment;

import android.support.v4.app.FragmentTransaction;

import android.support.v7.app.AppCompatActivity;

import android.view.LayoutInflater;

import android.view.View;

import com.google.gson.Gson;

import com.haibao.store.myapplication.api.Api;

import com.haibao.store.myapplication.api.ApiWrapper;

import com.haibao.store.myapplication.api.SimpleMyCallBack;

import com.haibao.store.myapplication.common.ActivityPageManager;

import com.haibao.store.myapplication.reponse.HttpExceptionBean;

import com.haibao.store.myapplication.utils.ToastUtils;

import com.haibao.store.myapplication.widget.dialog.DialogLoading;

import java.io.IOException;

import okhttp3.ResponseBody;

import retrofit2.HttpException;

import rx.Subscriber;

import rx.subscriptions.CompositeSubscription;

import static com.google.gson.internal.$Gson$Preconditions.checkNotNull;

/\*\*

\* Acivity 基类

\*/

public abstract class BaseActivity<T extends BasePresenter> extends AppCompatActivity implements View.OnClickListener {

protected AppCompatActivity mContext;

/\*\*

\* 使用CompositeSubscription来持有所有的Subscriptions

\*/

protected CompositeSubscription mCompositeSubscription;

/\*\*

\* 加载对话框

\*/

protected DialogLoading loading;

/\*\*

\* 来自哪个 页面

\*/

protected String fromWhere;

/\*\*

\* 页面布局的 根view

\*/

protected View mContentView;

/\*\*

\* Api类的包装 对象

\*/

protected ApiWrapper mApiWrapper;

public T presenter;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

// 设置不能横屏

setRequestedOrientation(ActivityInfo.SCREEN\_ORIENTATION\_NOSENSOR);

mContext = this;

//Activity管理

ActivityPageManager.getInstance().addActivity(this);

}

@Override

public void setContentView(int layoutResID) {

View view = LayoutInflater.from(this).inflate(layoutResID, null);

setContentView(view);

}

@Override

public void setContentView(View view) {

super.setContentView(view);

mContentView = view;

//初始化页面

init();

}

/\*\*

\* 初始化页面

\*/

public void init() {

initFromWhere();

initView();

bindEvent();

}

/\*\*

\* 初始化 Api 更具需要初始化

\*/

public void initApi() {

//创建 CompositeSubscription 对象 使用CompositeSubscription来持有所有的Subscriptions，然后在onDestroy()或者onDestroyView()里取消所有的订阅。

mCompositeSubscription = new CompositeSubscription();

// 构建 ApiWrapper 对象

mApiWrapper = new ApiWrapper();

}

public ApiWrapper getApiWrapper() {

if (mApiWrapper == null) {

mApiWrapper = new ApiWrapper();

}

return mApiWrapper;

}

public CompositeSubscription getCompositeSubscription() {

if (mCompositeSubscription == null) {

mCompositeSubscription = new CompositeSubscription();

}

return mCompositeSubscription;

}

/\*\*

\* 初始化view

\*/

public abstract void initView();

/\*\*

\* 绑定事件

\*/

public abstract void bindEvent();

/\*\*

\* 创建相应的 presenter

\*/

public void createPresenter(T presenter) {

if (presenter != null) {

this.presenter = presenter;

}

}

protected void initFromWhere() {

if (null != getIntent().getExtras()) {

if (getIntent().getExtras().containsKey("fromWhere")) {

fromWhere = getIntent().getExtras().getString("fromWhere").toString();

}

}

}

public String getFromWhere() {

return fromWhere;

}

/\*\*

\* 创建观察者 这里对观察着 过滤一次，过滤出我们想要的信息，错误的信息toast

\*

\* @param onNext

\* @param <T>

\* @return

\*/

protected <T> Subscriber newMySubscriber(final SimpleMyCallBack onNext) {

return new Subscriber<T>() {

@Override

public void onCompleted() {

hideLoadingDialog();

onNext.onCompleted();

}

@Override

public void onError(Throwable e) {

if (e instanceof Api.APIException) {

Api.APIException exception = (Api.APIException) e;

ToastUtils.showShort(exception.message);

} else if (e instanceof HttpException) {

if (e instanceof HttpException) {

ResponseBody body = ((HttpException) e).response().errorBody();

try {

String json = body.string();

Gson gson = new Gson();

HttpExceptionBean mHttpExceptionBean = gson.fromJson(json, HttpExceptionBean.class);

if (mHttpExceptionBean != null && mHttpExceptionBean.getMessage() != null) {

ToastUtils.showShort(mHttpExceptionBean.getMessage());

onNext.onError(mHttpExceptionBean);

}

} catch (IOException IOe) {

IOe.printStackTrace();

}

}

}

// e.printStackTrace();

hideLoadingDialog();

}

@Override

public void onNext(T t) {

if (!mCompositeSubscription.isUnsubscribed()) {

onNext.onNext(t);

}

}

};

}

/\*\*

\* 将 Fragment添加到Acitvtiy

\*

\* @param fragment

\* @param frameId

\*/

protected void addFragmentToActivity(@NonNull Fragment fragment, int frameId) {

checkNotNull(fragment);

FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();

transaction.add(frameId, fragment);

transaction.commit();

}

/\*\*

\* 显示一个Toast信息

\*/

public void showToast(String content) {

if (content != null) {

ToastUtils.showShort(content);

}

}

public void showLoadingDialog() {

if (loading == null) {

loading = new DialogLoading(this);

}

loading.show();

}

public void hideLoadingDialog() {

if (loading != null) {

loading.dismiss();

}

}

/\*\*

\* 跳转页面

\*

\* @param clazz

\*/

public void skipAct(Class clazz) {

Intent intent = new Intent(this, clazz);

intent.putExtra("fromWhere", getClass().getSimpleName());

startActivity(intent);

}

public void skipAct(Class clazz, Bundle bundle) {

Intent intent = new Intent(this, clazz);

intent.putExtras(bundle);

intent.putExtra("fromWhere", getClass().getSimpleName());

startActivity(intent);

}

public void skipAct(Class clazz, Bundle bundle, int flags) {

Intent intent = new Intent(this, clazz);

intent.putExtra("fromWhere", getClass().getSimpleName());

intent.setFlags(flags);

startActivity(intent);

}

@Override

protected void onDestroy() {

super.onDestroy();

//Acitvity 释放子view资源

ActivityPageManager.unbindReferences(mContentView);

ActivityPageManager.getInstance().removeActivity(this);

mContentView = null;

//一旦调用了 CompositeSubscription.unsubscribe()，这个CompositeSubscription对象就不可用了,

// 如果还想使用CompositeSubscription，就必须在创建一个新的对象了。

if(mCompositeSubscription != null){

mCompositeSubscription.unsubscribe();

}

//解绑 presenter

if (presenter != null) {

presenter.unsubscribe();

}

}

}

1. **代码任务：**

BaseAcitity为所有Activity的基类，此类为抽象类，采用模板方法模式， 在init（）方法中定义一个骨架，而将一些步骤延迟到子类中。使得子类acitivity可以在不改变结构的情况下，实现定义抽象中的某些步骤，这样所有的Activity都是这样的一种模式，看着简单易懂。另外包含了一些Activity常用的方法。

**3.代码段分析：**

创建观察者T（订阅）：

protected <T> Subscriber newMySubscriber(final SimpleMyCallBack onNext) {

return new Subscriber<T>() {

被观察者toast:

if (mHttpExceptionBean != null && mHttpExceptionBean.getMessage() != null) {

ToastUtils.showShort(mHttpExceptionBean.getMessage());

onNext.onError(mHttpExceptionBean);

}

公共空间的完成：

public void onCompleted() {

hideLoadingDialog();

onNext.onCompleted();

}

下一个公共空间：

public void onNext(T t) {

if (!mCompositeSubscription.isUnsubscribed()) {

onNext.onNext(t);

}

}

显示一个toast：

public void showToast(String content) {

if (content != null) {

ToastUtils.showShort(content);

}

}

public void showLoadingDialog() {

if (loading == null) {

loading = new DialogLoading(this);

}

loading.show();

}

public void hideLoadingDialog() {

if (loading != null) {

loading.dismiss();

}

}

1. **总结**

观察者模式的效果有以下的**优点**：

第一、观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。

由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

第二、观察者模式支持广播通讯。被观察者会向所有的登记过的观察者发出通知，

观察者模式有下面的**缺点**：

第一、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

第二、如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式是要特别注意这一点。

第三、如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的。

第四、虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎么发生变化的。