Android开发笔记

# 序

本文档写给有一定android基础，独立开发app能力欠缺的新人。

开发者Java基本的语法要具备。面向对象设计思想要懂。反射注解高级应用需要了解。

假设app的packageName为 phoebe.frame.lib

# Appllication的功能设计

每一个App都应该有一个扩展的Application。这个类中主要写一些整个app广泛用到的方法或app相关配置操作

public class PhoebeApp extends android.app.Application

androidManifest.xml中配置

<application android:name=*"*phoebe.frame.lib*.PhoebeApp"*

application的生命周期是从app启动到退出。进程结束。所以在application中变量都定义为static, 也可以定义为非static。只是调用的时候稍微麻烦点

## Context

App的上线文引用，主要用于调用系统api

比如说 context.getSystemService(String name) 我们在app中调用的时候可以

PhoebeApp.getContext().getSystemService(Context.NOTIFICATION\_SREVICE)

## Handler

App中会频繁用到handler与线程的交互。有时候我们需要用Handler#postDelayed(Runnable r , int delayMills) 这个时候我们无需重新new handler(); 只需要调用全局的PhoebeApp#getHander() 即可

更高级的用法先不讲。后续再

至此一个基本的Application扩展类已经定义完成

# Log类的设计

正式代码设计之前我们先考虑一下日志类的使用。

日志主要用在调试app 一些复杂的业务流程，借助log 可以很容易问题所在

因为android.util.Log 定义为final类(不可继承) 所以我采用组合的方式才重写一个可以控制开关的工具类。考虑到Log应用的广泛性，所以定义为static，生命周期为全局

public class Log{

private staic boolean open = false; //默认关闭

public static boolean isOpen() {

return open;

}

public static void setOpen(boolean open) {

Log.open = open;

}

}

Log的开关控制，一般在app初始化的地方设置。后续再讨论

public static void d(String tag, String msg){

if(open == false){

return; // 不打印日志

}

android.util.Log.*d*(tag, msg);

}

其中tag和msg 考虑到传入的参数需要频繁转型

比如说像打印一个int型的变量值。这时候会很不方便。同理其他非String类型的都需要转化为String 。

因为优化一下当前方法

public static void d(Object tag, Object msg){

if(open == false){

return; // 不打印日志

}

android.util.Log.*d*(String.valueOf(tag), String.valueOf(msg));

}

这时候如果传入的 tag == null 或者 “” 或者 包含有” ” \n\r等字符 ，那么输出的结果可能不是我们想要的，

接着继续优化一下该方法

public static void d(Object tag, Object msg){

if(open == false){

return; // 不打印日志

}

tag = handleMsgOrTag(tag);

msg = handleMsgOrTag(msg);

android.util.Log.*d*(String.valueOf(tag), String.valueOf(msg));

}

private static Object handleMsgOrTag(Object msgOrTag) {

if (msgOrTag == null) {

msgOrTag = "[null]";

} else if (msgOrTag.toString().trim().length() == 0) {

msgOrTag = "[\"\"]";

} else {

msgOrTag = msgOrTag.toString().trim();

}

return msgOrTag;

}

# App的属性配置

一般app测试环境和线上环境好多配置都不同，比如说控制日志打印、服务器地址等

所以需要一个类专门进行app的属性配置。

至于为何不在PhoebeApp中写，一是减少PhoebeApp类的代码量，二分开写配置起来更自由。代码更清晰

因为是全局性的配置，所以类中的方法字段全部定义为static

public class PhoebeConfig

/\*\*

\* 默认开发环境 false表示线上正式环境

\*/

private static boolean debug = true;

/\*\*

\* app接口服务器地址(线上环境)

\*/

private static String serverUrl;

/\*\*

\* 同上 -- 测试环境

\*/

private static String serverUrl\_test;

/\*\*

\* 初始化进行app的配置 一般在{@link Application#onCreate()}中配置

\*/

public static void init() {

if (debug) {

Log.setOpen(true);

} else {

Log.setOpen(false);

}

}

/\*\*

\* 获取接口服务器的地址

\*

\* @return

\*/

public static String getServerUrl() {

if (debug) {

return serverUrl\_test;

} else {

return serverUrl;

}

}

定义完成以后 在PhoebeApp的onCreate中调用，使用

public void onCreate() {

super.onCreate();

onCreateHandler();

// 初始化app配置信息

PhoebeConfig.init();

}

做完以上工作，开始着手具体的功能设计。

# 功能设计

先把app常用的功能进行抽象，主要从Activity Fragment Service等控件进行考虑。

其次完善相关的工具类。主要从代码封装、迭代、可维护性等方面考虑。

## 抽象

### Activity的抽象

Activity抽象主要目的是简化具体功能界面的工作量。主要通过代码封装的方式来达到效果。

同时需要处理网络请求、UI刷新等操作

### Fragment的抽象

Fragment的抽象与activity类似。但不同的是需要处理多个Fragment之间的互相调用，传参方式也不太一样。

## 工具类

App需要一个管理Activity或者Fragment的工具类，对其做全局的操作(方法调用) 为了减少代码的耦合度。不建议在Activity中直接加static引用。可以通过写一些工具类间接的实现。

Activity中访问网络请求以及回调刷新UI是一件比较繁琐的事儿。常规的做法是new一个Thread是传入一个Handler对象，网络请求完成以后sendMessage刷新。但是这种做法耦合度比较强，而且好多业务逻辑都需要在Activity中处理，代码臃肿

private Handler handler;

protected void onCreate(Bundle savedInstanceState){

super.onCreate(savedInstanceState);

handler = new handler(){

public void handlerMessage(){

if(msg.what == 1){

// do sth

}

}

}

}

public class NetRunnable implements Runnable{

private Handler handler;

public NetRunnable(Handler handler){

this.handler = handler;

}

public void run(){

// do sth 处理网络等

Message msg = Message.obtain();

msg.what = 1;

msg.obj = “网络请求结果”;

this.handler.sendMessage(msg);

}

}

建议将网络请求的代码封装一下,

### ActivityMgr

这个类的作用主要是保存app中所有正在运行的Activity的引用。

private static Stack<BaseActivity> activities = new Stack< BaseActivity >;

public static void push(BaseActivity activity){

activities.push(activity);

}

public static void remove(BaseActivity activity){

activities.remove(activity);

}

public class BaseActivity extends Activity

public void onCreate(Bundle saveInstanceState){

ActivityMgr.push(this);

}

public void onDestroy(){

ActivityMgr.remove(this);

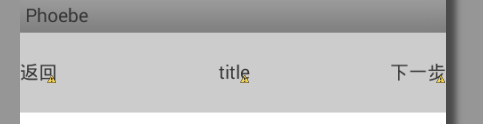
}

解析一下为什么onDestroy的时候采用remove方法 而不是pop()

防止acitivty AB, A 启动 B 如果此时A#finish 调用Stack#pop则会把B的引用移除

Fragment的工具类先略过

# 界面封装



假设app的title都是这种左中右结构的布局

先来继续完善一下BaseActivity

protected void onCreate(Bundle savedInstanceState){

super.onCreate(savedInstanceState);

ActivityMgr.push(this);

findViewById();

}

// 初始化app中通用的控件

protected void findViewById(){

}

// 设置标题栏

protected void setTitle(){

}

然后看一下BaseActivity的具体实现类

public class TitleDemoActivity extends BaseActivity{

protected void onCreate(Bundle savedInstanceState){

super.onCreate(savedInstanceState);

}

protected void findViewById(){

setContentView(R.layout.title\_demo);

super.findViewById();

super.setTitle()；// 设置标题栏

}

}

Title\_layout的源码略

## TitleBar封装

BaseActivity的设计初衷是所有的Activity的都继承该类。

首先定义一些通用的属性、以及方法