# 自我介绍

2015年毕业于北京航空航天大学北海学院，读的是计算机科学与技术专业。在大三暑期实践时接触到了Android，当时觉得能在手机上运行自己开发的应用觉得挺有趣，还可以展示给亲戚朋友看，于是我就开始往Android开发方向发展了。

平时喜欢看郭霖、洪洋这些公众号来提升自己。

2015年就职于深圳市浩天投资有限公司，从事一些Android SDK和WEB前端的一些工作。为公司开发过一个官网，长期维护公司Android SDK的广告模块。后来因为部门转为游戏开发和自己职业规划冲突而离职。

2017年在乐逗游戏做Android开发，负责更新和维护公司的渠道计费SDK，支撑公司游戏接入SDK后上线。功能包括登录、支付、退出、闪屏、分享、推送、问卷调查。并根据bugly排查崩溃问题，根据自有上报数据排查登录不上及支付优化。支付过程打点，分析游戏支付丢失问题，通过顺序可知游戏是否闪退，收到什么支付回调。为游戏解决支付调单问题（轮询订单），为游戏playerId变化问题做udid持久化,androidId+imei/android+mac/null，只保存私有空间,根据权限变化会生成不同udid，卸载重装包会导致变化，将udid保存在SDK卡可防止卸载清理私有空间数据。因为支撑业务较多，工作时常被打断无法专心考虑离职。

职业规划：高级Android工程师，Android架构师。

# Okhttp

用于替代HttpUrlConnection和Apache HttpClient，android API23 6.0里已移除HttpClient。

## 优势

允许连接到同一个主机地址的所有请求,提高请求效率

共享Socket,减少对服务器的请求次数

通过连接池,减少了请求延迟

缓存响应数据来减少重复的网络请求

减少了对数据流量的消耗

自动处理GZip压缩

## 功能

get,post请求

文件的上传下载

加载图片(内部会图片大小自动压缩)

支持请求回调，直接返回对象、对象集合 

支持session的保持

# Volley

* 使用Volley 需要Volley.jar(120k)，加上自己的封装最多140k。
* 使用OkHttp需要 okio.jar (80k), okhttp.jar(330k)这2个jar包，总大小差不多400k,加上自己的封装，差不多得410k。

## ****Volley 的优点****

非常适合进行数据量不大，但通信频繁的网络操作

可直接在主线程调用服务端并处理返回结果

可以取消请求，容易扩展，面向接口编程

网络请求线程NetworkDispatcher默认开启了4个，可以优化，通过手机CPU数量

通过使用标准的HTTP缓存机制保持磁盘和内存响应的一致

## ****Volley 的缺点****

使用的是httpclient、HttpURLConnection

6.0不支持httpclient了，如果想支持得添加org.apache.http.legacy.jar

对大文件下载 Volley的表现非常糟糕

只支持http请求

图片加载性能一般

# 动画

# 图片优化

# 反射

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

# ****Map、Set、List****

## 1 HashMap、ConcurrentHashMap、HashTable

HashMap是支持null键和null值的，而HashTable在遇到null时，会抛出NullPointerException异常。HashTable是同步的，HashMap不是。

ConcurrentHashMap采用锁分段技术，将整个Hash桶进行了分段segment，也就是将这个大的数组分成了几个小的片段segment，而且每个小的片段segment上面都有锁存在，那么在插入元素的时候就需要先找到应该插入到哪一个片段segment，然后再在这个片段上面进行插入，而且这里还需要获取segment锁。

## 2 ArrayList、LinkedList

ArrayList是实现了基于动态数组的数据结构。

LinkedList基于链表的数据结构。

对于随机访问get和set，ArrayList优，因为LinkedList要移动指针。

对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

## 3 HashSet、TreeSet

HashSet是使用散列表进行存储，元素无序，元素允许为null。TreeSet是使用树结构来进行存储，元素按字符串顺序排序存储，元素不允许为null。

# ****线程****

## 1创建线程的方式

多进程是指操作系统能同时运行多个任务（程序）。

多线程是指在同一程序中有多个顺序流在执行。

通过Runnable接口创建线程类，Runnable是接口，可以再继承其他类。

继承Thread类创建线程

使用Callable和Future创建线程

## 2 Join、yeild、sleep、wait、notify、notifyall

## 3线程池

如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。根据获取cpu的核心来开线程。

使用单例线程池

// 获取单例的线程池对象

public static ThreadPool getThreadPool() {

if (mThreadPool == null) {

synchronized (ThreadManager.class) {

if (mThreadPool == null) {

int cpuNum = Runtime.getRuntime().availableProcessors();// 获取处理器数量

int threadNum = cpuNum \* 2 + 1;// 根据cpu数量,计算出合理的线程并发数

mThreadPool = new ThreadPool(threadNum, threadNum, 0L);

}

}

}

return mThreadPool;

}

// 获取单例的线程池对象

public static ThreadPool getThreadPool() {

if (mThreadPool == null) {

synchronized (ThreadManager.class) {

if (mThreadPool == null) {

int cpuNum = Runtime.getRuntime().availableProcessors();// 获取处理器数量

int threadNum = cpuNum \* 2 + 1;// 根据cpu数量,计算出合理的线程并发数

mThreadPool = new ThreadPool(threadNum, threadNum, 0L);

}

}

}

return mThreadPool;

}

# ****String****

方式一：String a = “aaa” ;

方式二：String b = new String(“aaa”);

两种方式都能创建字符串对象，但方式一要比方式二更优。

因为字符串是保存在常量池中的，而通过new创建的对象会存放在堆内存中。

# ****基本数据类型****

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 简单类型 | boolean | byte | char | short | Int | long | float | double | void |
| 二进制位数 | 1 | 8 | 16 | 16 | 32 | 64 | 32 | 64 | -- |
| 封装器类 | Boolean | Byte | Character | Short | Integer | Long | Float | Double | Void |

# ****http和https****

# ****tcp和udp****

# ****接口和抽象类****



# ****单例模式****

线程安全，保证在不同线程中不能同时实例化对象。

public class Single{

private static Single instance;

private Single(){}

public static synchronizedSingle getInstance(){

if(instance==null){

instance = new Single();

}

return instance;

}

}

# ANR

Activity 5秒内无法响应用户输入事件(例如键盘输入, 触摸屏幕等)

BroadcastReceiver在10秒内无法结束.

在主线程(UI线程)里面做了太多的阻塞耗时操作, 例如文件读写, 数据库读写, 网络查询

ANR产生时, 系统会生成一个traces.txt的文件放在/data/anr/下. 可以通过[adb](https://www.jianshu.com/p/5980c8c282ef" \t "_blank)命令将其导出到本地:

分析traces.txt文件中的日志有哪些耗时操作

# 判断主线程

方法一：使用 Looper 类判断   
Looper.myLooper() != Looper.getMainLooper()   
方法二：通过查看 Thread 类的当前线程   
Thread.currentThread() == Looper.getMainLooper().getThread()   
方法三：打印 Log，去看线程 id，看是否和进程号一样，一样是主线 程

# Singletop和SingleTask的不同

SingleTop 栈顶复用模式

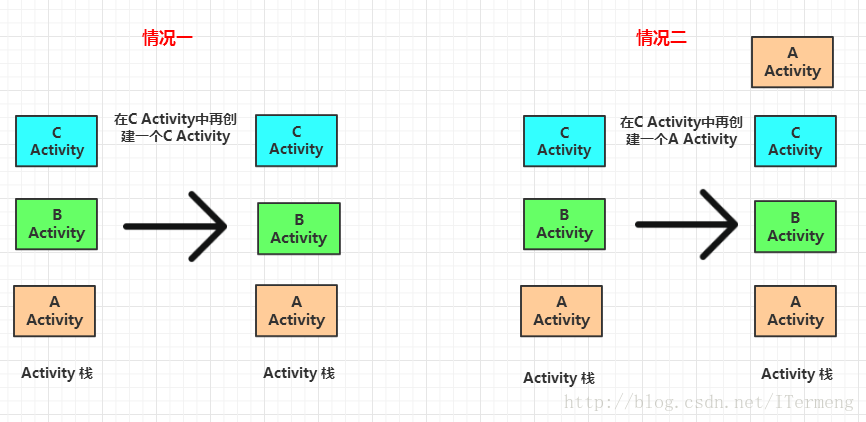
说明：分两种处理情况：须要创建的Activity已经处于栈顶时，此时会直接复用栈顶的Activity。不会再创建新的Activity；若须要创建的Activity不处于栈顶，此时会又一次创建一个新的Activity入栈，同Standard模式一样。

生命周期：若情况一中栈顶的Activity被直接复用时，它的onCreate、onStart不会被系统调用，由于它并没有发生改变。可是一个新的方法 onNewIntent会被回调（Activity被正常创建时不会回调此方法）。

举例：此时Activity 栈中以此有A、B、C三个Activity，此时C处于栈顶，启动模式为SingleTop 模式。

情况一：在C Activity中加入点击事件，须要跳转到还有一个同类型的C Activity。结果是直接复用栈顶的C Activity。

情况二：在C Activity中加入点击事件，须要跳转到还有一个A Activity。结果是创建一个新的Activity入栈。成为栈顶。



SingleTask 栈内复用模式

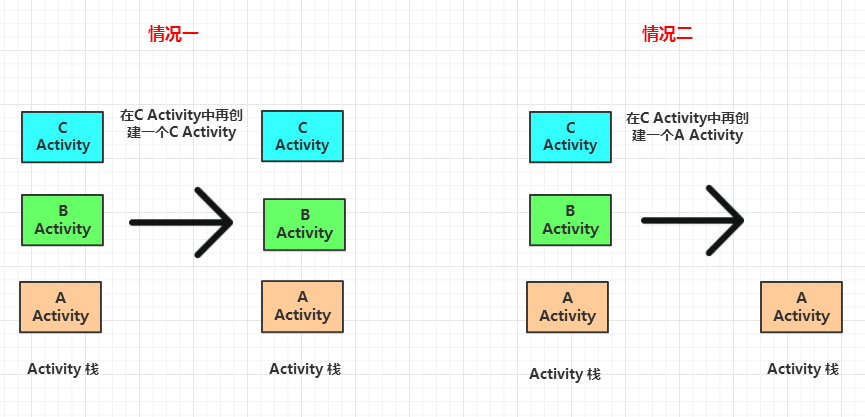
说明：若须要创建的Activity已经处于栈中时，此时不会创建新的Activity，而是将存在栈中的Activity上面的其他Activity所有销毁，使它成为栈顶。

生命周期：同SingleTop 模式中的情况一同样。仅仅会又一次回调Activity中的 onNewIntent方法

举例：此时Activity 栈中以此有A、B、C三个Activity。此时C处于栈顶，启动模式为SingleTask 模式。

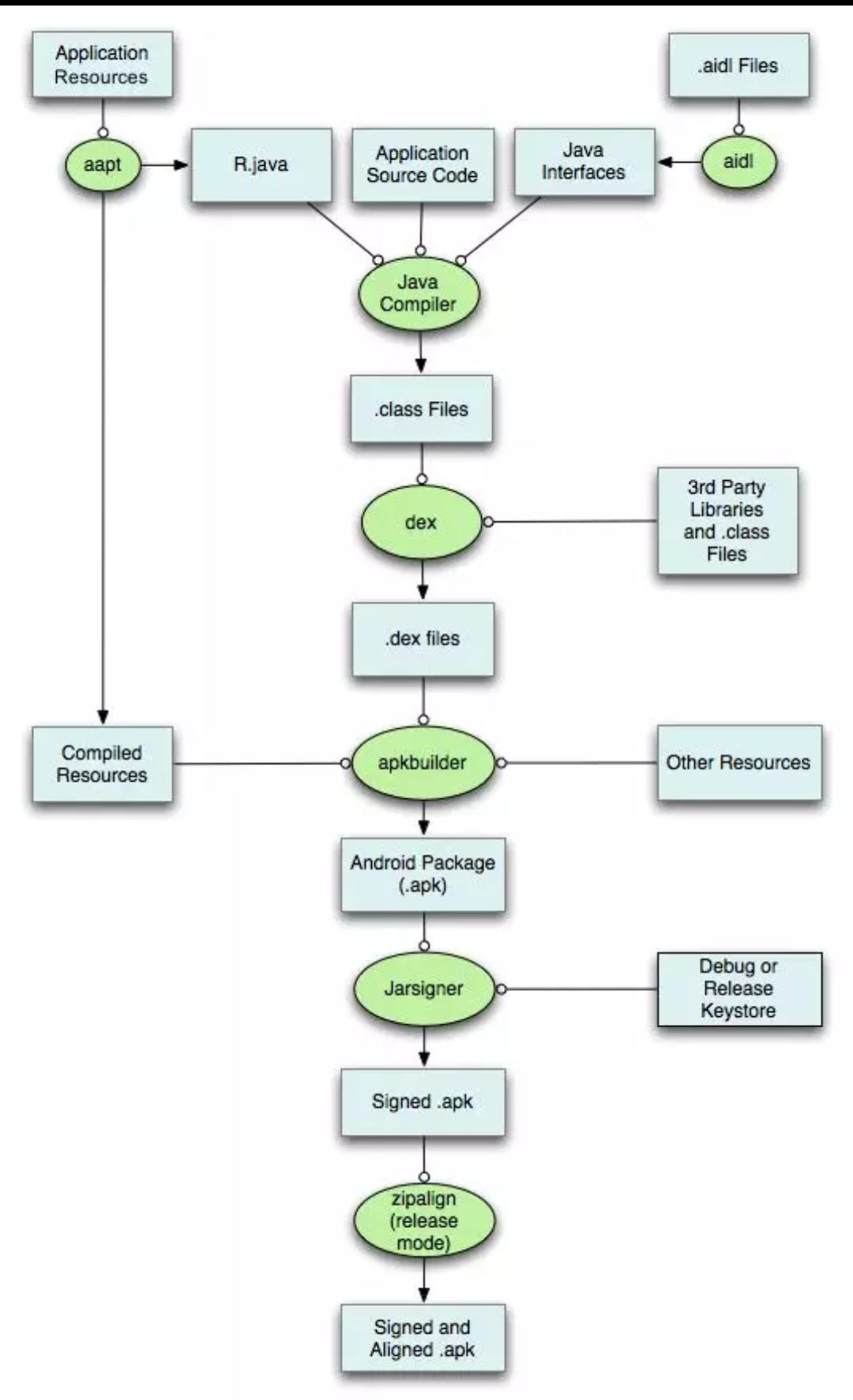
情况一：在C Activity中加入点击事件，须要跳转到还有一个同类型的C Activity。结果是直接用栈顶的C Activity。

情况二：在C Activity中加入点击事件，须要跳转到还有一个A Activity。结果是将A Activity上面的B、C所有销毁，使A Activity成为栈顶。



# APK编译流程

1. 工程的资源文件(res文件夹下的文件)，通过AAPT打包成R.java类(资源索引表)，以及.arsc资源文件 。Manifest被编译成二进制xml，arsc是资源索引，R.java是资源常量池。
2. 如果有aidl（跨进程通信），通过aidl工具，打包成java接口类
3. R.java和aidl.java通过java编译成想要的.class文件。
4. 源码class文件和第三方jar或者library通过dx工具打包成dex文件。dx工具的主要作用是将java字节码转换成Dalvik字节码，在此过程中会压缩常量池，消除一些冗余信息等。
5. apkbuilder工具会将所有没有编译的资源，.arsc资源，.dex文件打包到一个完成apk文件中中。
6. 签名，5中完成apk通过配置的签名文件(debug和release都有)，jarsigner工具会对齐签名。得到一个签名后的apk,signed.apk
7. zipAlign工具对6中的signed.apk进行对齐处理，所谓对齐，主要过程是将APK包中所有的资源文件距离文件起始偏移为4字节整数倍，这样通过内存映射访问apk文件时的速度会更快。对齐的作用主要是为了减少运行时内存的使用。



# AIDL

设计这门语言的目的是为了实现进程间通信，尤其是在涉及多进程并发情况下的进程间通信。

每一个进程都有自己的Dalvik VM实例，都有自己的一块独立的内存，都在自己的内存上存储自己的数据，执行着自己的操作，都在自己的那片狭小的空间里过完自己的一生。每个进程之间都你不知我，我不知你，就像是隔江相望的两座小岛一样，都在同一个世界里，但又各自有着自己的世界。而AIDL，就是两座小岛之间沟通的桥梁。相对于它们而言，我们就好像造物主一样，我们可以通过AIDL来制定一些规则，规定它们能进行哪些交流——比如，它们可以在我们制定的规则下传输一些特定规格的数据。

总之，通过这门语言，我们可以愉快的在一个进程访问另一个进程的数据，甚至调用它的一些方法，当然，只能是特定的方法。

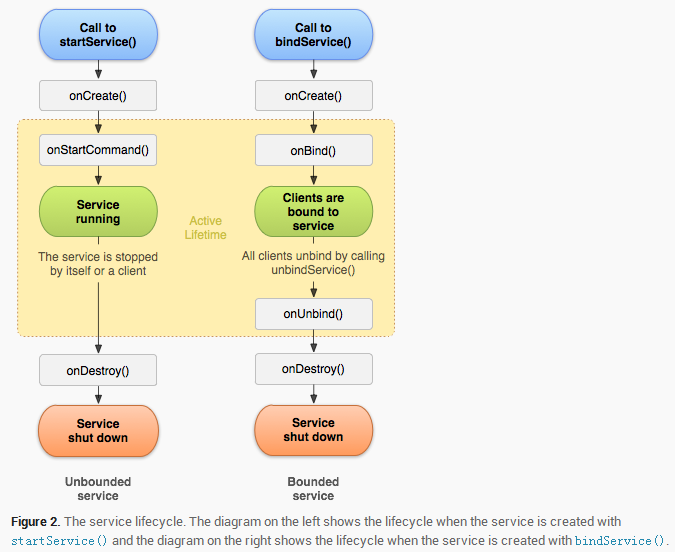
但是，如果仅仅是要进行跨进程通信的话，其实我们还有其他的一些选择，比如 BroadcastReceiver , Messenger 等，但是 BroadcastReceiver 占用的系统资源比较多，如果是频繁的跨进程通信的话显然是不可取的；Messenger 进行跨进程通信时请求队列是同步进行的，无法并发执行，

# Activity

1. onCreate：与onDestroy配对，表示Activity正在被创建，这是生命周期的第一个方法。在这个方法中可以做一些初始化的工作（加载布局资源、初始化Activity所需要的数据等），耗时的工作在异步线程上完成。
2. onRestart：表示Activity正在重新启动。一般情况下，在当前Activity从不可见重新变为可见的状态时onRestart就会被调用。这种情形一般是由于用户的行为所导致的，比如用户按下Home键切换到桌面或者打开了一个新的Activity（这时当前Activity会暂停，也就是onPause和onStop被执行），接着用户有回到了这个Activity，就会出现这种情况。
3. onStart：与onStop配对，表示Activity正在被启动，并且即将开始。但是这个时候要注意它与onResume的区别。两者都表示Activity可见，但是onStart时Activity还正在加载其他内容，正在向我们展示，用户还无法看到，即无法交互。
4. onResume：与onPause配对，表示Activity已经创建完成，并且可以开始活动了，这个时候用户已经可以看到界面了，并且即将与用户交互（完成该周期之后便可以响应用户的交互事件了）。
5. onPause：与onResume配对，表示Activity正在暂停，正常情况下，onStop接着就会被调用。在特殊情况下，如果这个时候用户快速地再回到当前的Activity,那么onResume会被调用（极端情况）。一般来说，在这个生命周期状态下，可以做一些存储数据、停止动画的工作，但是不能太耗时，如果是由于启动新的Activity而唤醒的该状态，那会影响到新Activity的显示，原因是onPause必须执行完，新的Activity的onResume才会执行。
6. onStop：与onStart配对，表示Activity即将停止，可以做一些稍微重量级的回收工作，同样也不能太耗时（可以比onPause稍微好一点）。
7. onDestroy：与onCreate配对，表示Activity即将被销毁，这是Activity生命周期的最后一个回调，我们可以做一些回收工作和最终的资源释放（如Service、BroadReceiver、Map等）。



# 广播



# oom

使用AS的monitors导出hporf文件（dump Java heap按钮），用sdk\platform-tools\ hprof-conv.exe转换成eclipse格式的hporf文件（hprof-conv n:\1.hprof n:\2.hprof），然后用eclipse的mat分析。。。

MAT集成方法：1.Help->Eclipse Marketplace->查找Memory Analysis->install->重启eclipse-> Window-> open perspective->打开Memory Analysis

充分挖掘特征。在挖掘特征时，需要多方面考虑，此过程更多的是猜测怀疑，所以可能的方面都要考虑到，包括但不限于代码改动、机器特征、时间特征等，必要时还需要做一定的统计分析。

根据掌握的特征寻找稳定的复现的途径。一般需要做内存压力测试，这样比较容易达到OOM的临界值，只是简单的一些正常操作难以触发OOM。

获取可分析的数据（内存dump文件）。利用MAT分析dump文件，MAT可以方便的按照大小排序实例，可以查看某些实例到GC ROOT的路径。

# AsyncTask

Android封装的异步线程类，AsyncTask的内部Handler和ThreadPoolExecutor都是进程范围内共享

# 版本适配

targetSdkVersion

## 6.0

动态申请权限

移除Appache的HTTP Client

## 7.0

应用间共享文件URI不能使用file://需要使用 content://，manifest文件中注册FileProvider

## 8.0

Android8.0中androidID算法和之前不一样，应用签署密钥、用户和设备的每个组合都具有唯一的 androidID 值

# handler消息机制

应用会把操作转换成消息让主线程处理，应用启动的时候会通过ActivityThread创建一个主线程，在主线程中关联一个Looper不断从消息队列MessageQueue中获取消息，消息队列对应了一个handler传送消息到消息队列中。

# 强更

1. 检查本地版本是否最新版本
2. dialog提示用户更新
3. dialog显示下载进度
4. 下载最新版本APK
5. 下载完成提示安装
6. Intent startActivity安装

静默安装：需要root权限，用pm install指令安装APK



# 静态内部类和非静态内部类

内部类是外部类的方法。

静态内部类可以有静态成员（方法和属性），而非静态内部类则不能有静态成员（成员或属性）

静态内部类只能够访问外部类的静态成员和静态方法，而非静态内部类则可以访问外部类的所有成员（方法和属性）

实例化一个非静态的内部类的方法：

OutClass.InnerClass  innerClass = new OutClass().new InnerClass();

实例化一个静态内部类的方法：

不依赖于外部类的实例，直接实例化静态内部类对象：

OutClass.InnerClass innerClass =  new OutClass.InnerClass();

内部类相对于外部类来说，前者是后者的方法，所以当前者是static修饰的时候，在前者中不能访问后者中的非static属性和方法；相反，当前者是非static修饰的时候，前者可以访问后者中的任何成员。

什么时候用静态内部类？什么时候用非静态内部类？

1）首先要弄清楚：为什么要用内部类？用内部类是因为内部类与所在外部类有一定的关系，往往只有该外部类调用此内部类，

所以没有必要专门用一个Java文件存放这个类。

2）生命周期不一样：静态内部类随着外部类的加载而加载，而不是随着外部类对象的产生而产生。

外部类实例与静态内部类实例是没有关系的。

外部内部类实例对应着不同的非静态内部类实例。

# 待更新知识点

泛型、重写、重载、锁synchronized、jni、全局变量线程池、、图片加载框架、自定义布局、不赋予读写权限怎么写文件