# 知识点总结

[知识点总结 1](#_Toc1281130045)

[Android基本知识点 6](#_Toc1281621493)

[1 Android类加载器 6](#_Toc951453441)

[2 Service 6](#_Toc914747325)

[2.1 IntentService 7](#_Toc322862402)

[2.2 生命周期 7](#_Toc1804698092)

[2.3 startService生命周期 8](#_Toc501802016)

[2.4 bindService生命周期 11](#_Toc618201143)

[3 fragemnt 12](#_Toc580726215)

[3.1创建方式 12](#_Toc2099803537)

[3.1.1 静态创建 12](#_Toc1799275208)

[3.1.2 动态创建具体步骤 12](#_Toc1701187449)

[3.2 Adapter对比 12](#_Toc260179185)

[3.3 生命周期 13](#_Toc554857003)

[3.1.1 动态加载： 13](#_Toc1107654147)

[3.1.2静态加载： 14](#_Toc1954996433)

[3.4 与Activity通信 14](#_Toc1125250331)

[3.4.1 Activity向Fragment传值： 14](#_Toc1341317635)

[3.4.2 Fragment向Activity传值： 14](#_Toc1389648886)

[3.4.3 Fragment与Fragment之间是如何传值的： 15](#_Toc1944165877)

[3.5 api区别 15](#_Toc1632205634)

[3.6懒加载 16](#_Toc523983860)

[4 Activity 17](#_Toc1913782320)

[4.1 Activity启动流程 17](#_Toc2076871121)

[4.1.1 点击Launcher图标来启动Activity 17](#_Toc773732309)

[4.1.2 Activity生命周期 21](#_Toc1105434778)

[5. view部分知识点 21](#_Toc1161283649)

[5.1 DecorView浅析 21](#_Toc1362904807)

[5.1.1 DecorView的作用 23](#_Toc1280512347)

[5.1.2 使用总结 23](#_Toc1637389442)

[5.2 View的事件分发 23](#_Toc1748899036)

[5.2.1 ViewGroup事件分发 24](#_Toc1137421563)

[5.2.2 View的事件分发 25](#_Toc1892267394)

[5.2.3 onTouch和onTouchEvent的区别 25](#_Toc1252762535)

[5.3 View的绘制 25](#_Toc1250250557)

[5.3.1 onMeasure(int widthMeasureSpec, int heightMeasureSpec) 25](#_Toc1981109251)

[5.3.2 onDraw 27](#_Toc1916718469)

[5.4 ViewGroup的绘制 28](#_Toc2032603483)

[5.4.1 onLayout 29](#_Toc1944365952)

[6 系统原理 30](#_Toc699898865)

[6.1 打包原理 30](#_Toc1432289436)

[6.1.1 问题复现 31](#_Toc1344351629)

[6.2 安装流程 32](#_Toc842378516)

[6.3 混淆 32](#_Toc1643517388)

[6.3.1 正常app混淆规则： 33](#_Toc1662072402)

[6.3.2 插件app混淆规则 33](#_Toc2131063885)

[6.3.3 插件app混淆方案 33](#_Toc1058450529)

[7. 第三方库解析 34](#_Toc1770992802)

[7.1 网络请求框架 34](#_Toc952675794)

[7.1.1 Retrofit 34](#_Toc2131481373)

[7.1.2 Okhttp 37](#_Toc1632720404)

[7.2 图片加载库对比 38](#_Toc585788662)

[7.2.1 介绍: 38](#_Toc1285004386)

[7.2.3 总结： 39](#_Toc1973161270)

[7.3 各种json解析库使用 39](#_Toc1478987916)

[7.3.1 Google的Gson 39](#_Toc226690187)

[7.3.2 阿里巴巴的FastJson 40](#_Toc345983131)

[8 热点技术 40](#_Toc1700250288)

[8.1 组件化 40](#_Toc1689183434)

[8.1.2 概念： 40](#_Toc372161898)

[8.1.3 由来： 41](#_Toc1452639622)

[8.1.4 优势： 41](#_Toc1920027858)

[8.1.5 考虑问题： 41](#_Toc1818929584)

[8.2 插件化 43](#_Toc1319803243)

[8.2.1概述 43](#_Toc574515238)

[8.2.2优点 43](#_Toc791128154)

[8.2.3缺点 44](#_Toc1419625701)

[8.3 多渠道打包 44](#_Toc1105838537)

[8.3.1 签名方式 44](#_Toc1504810221)

[9 屏幕适配 44](#_Toc430473628)

[9.1基本概念 44](#_Toc97859053)

[9.2 适配方法 45](#_Toc1892113816)

[10 性能优化 46](#_Toc819060736)

[10.1稳定——内存优化 47](#_Toc583612682)

[10.2 流畅——卡顿优化 47](#_Toc1220530525)

[10.3 节省——耗电优化 49](#_Toc692737531)

[10.4 安装包——APK瘦身 50](#_Toc1330833130)

[（1）安装包的组成结构 50](#_Toc1270232405)

[（2）减少安装包大小 51](#_Toc661096008)

[10.5 冷启动 51](#_Toc2107700525)

[11 模式架构 51](#_Toc1379966410)

[11.1 MVP模式 51](#_Toc272065270)

[12 虚拟机 52](#_Toc608308427)

[Android Dalvik虚拟机和ART虚拟机对比 52](#_Toc1817572869)

[12.1 Dalvik 52](#_Toc2139814355)

[12.2 ART 53](#_Toc2098711823)

[12.3 处理器 53](#_Toc630707186)

[12.3.1 so加载流程 54](#_Toc316393510)

[12.3.2 so加载算法 54](#_Toc456212598)

[13 Binder 54](#_Toc1048514796)

[13.1 Binder组成 54](#_Toc137369090)

[13.2 Binder通信过程 55](#_Toc217375105)

[14 AIDL 56](#_Toc553706188)

[需要了解的知识点 58](#_Toc1093259265)

[Java基本知识点 59](#_Toc538383123)

[1 Java的类加载过程 59](#_Toc1256543450)

[2 基础知识点 60](#_Toc371579552)

[2.1 String、StringBuilder、StringBuffer 60](#_Toc255084988)

[2.2 抽象类和接口 60](#_Toc836033904)

[3 JVM内存结构 61](#_Toc236322207)

[3.1、JVM基本结构 62](#_Toc1170069746)

[3.2 JVM源码分析 63](#_Toc854465443)

[4 GC机制 64](#_Toc777553012)

[5 类加载器 64](#_Toc895480689)

[5.1、双亲委派原理 65](#_Toc778541847)

[5.2 为什么使用双亲委托模型 65](#_Toc334961358)

[6、集合 65](#_Toc1140905119)

[6.1、区别 67](#_Toc310850970)

[6.2、List和Vector比较 67](#_Toc1792023286)

[6.3、HashSet如何保证不重复 67](#_Toc77218627)

[6.4、HashSet与Treeset的适用场景 68](#_Toc733341201)

[6.5、HashMap与TreeMap、HashTable的区别及适用场景 68](#_Toc856915074)

[7 常量池 69](#_Toc1146311936)

[7.1、Interger中的128(-128~127) 69](#_Toc988911115)

[7.2、为什么是-128-127? 69](#_Toc1253165672)

[8、泛型 70](#_Toc1583323175)

[8.1、泛型擦除 70](#_Toc1442732248)

[8.2、限定通配符 71](#_Toc763033859)

[8.3、泛型面试题 71](#_Toc1685211976)

[9、反射 72](#_Toc195860349)

[9.1、概念 72](#_Toc1879938439)

[9.2、作用 72](#_Toc198445962)

[10、代理 73](#_Toc239179543)

[11 注解 74](#_Toc1948675664)

[设计模式 74](#_Toc118784451)

[1 六大原则 75](#_Toc1397959894)

[2 单例模式 76](#_Toc2040840278)

[3 责任链模式 78](#_Toc793742462)

[4 建造者模式 79](#_Toc261143670)

[5 工厂模式 79](#_Toc1732570869)

[5.1 普通工厂模式 79](#_Toc1587825610)

[5.2 抽象工厂模式 81](#_Toc1953229648)

[跨平台研究 83](#_Toc1495665894)

[1. React Native 83](#_Toc1360592323)

[1.1 优点： 83](#_Toc1069299405)

[1.2 缺点 83](#_Toc1571941739)

[数据结构与算法 84](#_Toc1280981979)

[1、排序 84](#_Toc940559878)

[1.1 直接插入排序 85](#_Toc362743979)

[技术栈 85](#_Toc2079464867)

[1 插件化 86](#_Toc1417148391)

[2 MVP 86](#_Toc271878660)

[3 Retrofit 86](#_Toc1766921451)

[4 ButterKnife 86](#_Toc1244956241)

[5 UIL 87](#_Toc1046369766)

[6 fastJson 87](#_Toc593071879)

[7 eventbus 87](#_Toc1287464626)

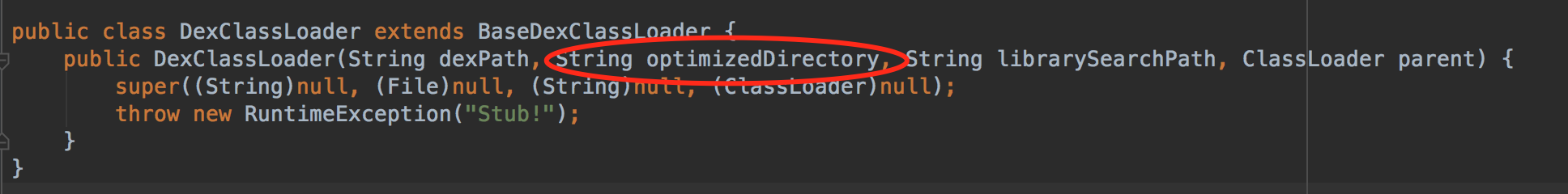
# Android基本知识点

## 1 基本概念

### 1.1 Android类加载器

我们知道不管是插件化还是组件化，都是基于系统的ClassLoader来设计的。只不过Android平台上虚拟机运行的是Dex字节码,一种对class文件优化的产物,传统Class文件是一个Java源码文件会生成一个.class文件，而Android是把所有Class文件进行合并，优化，然后生成一个最终的class.dex,目的是把不同class文件重复的东西只需保留一份,如果我们的Android应用不进行分dex处理,最后一个应用的apk只会有一个dex文件。

Android中常用的有两种类加载器，DexClassLoader和PathClassLoader，它们都继承于BaseDexClassLoader。区别在于调用父类构造器时，DexClassLoader多传了一个optimizedDirectory参数，这个目录必须是内部存储路径，用来缓存系统创建的Dex文件。而PathClassLoader该参数为null，只能加载内部存储目录的Dex文件。所以我们可以用DexClassLoader去加载外部的apk



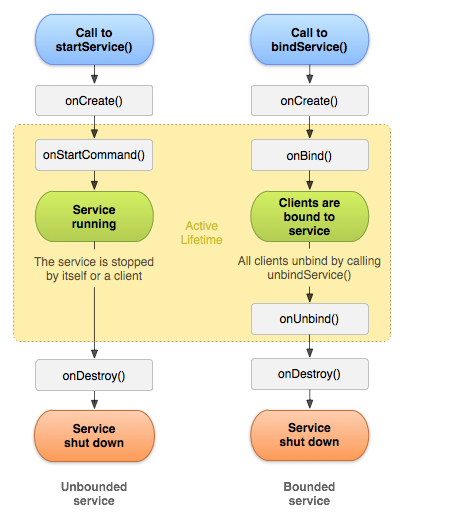
### 1.2 Service

* Service是在main Thread中执行，Service中不能执行耗时操作（网络请求，拷贝数据库，大文件）。
* 可以在Androidmanifest.xml中设置Service所在的进程，让Service在另外的进程中执行。
* Service执行的操作最多是20s，BroadcastReceiver是10s，Activity是5s。
* Activity通过bindService（Intent，ServiceConnection，flag）与Service绑定。
* Activity可以通过startService和bindService启动Service。

#### 1.2.1 IntentService

IntentService是一个抽象类，继承自Service，内部存在一个ServiceHandler（Handler）和HandlerThread（Thread）。IntentService是处理异步请求的一个类，在IntentService中有一个工作线程（HandlerThread）来处理耗时操作，启动IntentService的方式和普通的一样，不过当执行完任务之后，IntentService会自动停止。另外可以多次启动IntentService，每一个耗时操作都会以工作队列的形式在IntentService的onHandleIntent回调中执行，并且每次执行一个工作线程。它的本质是：**封装了一个HandlerThread和Handler的异步框架。**

#### 1.2.2 生命周期



#### 1.2.3 startService生命周期

当我们通过调用了Context的startService方法后，我们便启动了Service，通过startService方法启动的Service会一直无限期地运行下去，只有在外部调用Context的**stopService**或Service内部调用Service的**stopSelf**方法时，该Service才会停止运行并销毁。

* onCreate

onCreate: 执行startService方法时，如果Service没有运行的时候会创建该Service并执行Service的onCreate回调方法；如果Service已经处于运行中，那么执行startService方法不会执行Service的onCreate方法。也就是说如果多次执行了Context的startService方法启动Service，Service方法的onCreate方法只会在第一次创建Service的时候调用一次，以后均不会再次调用。我们可以在onCreate方法中完成一些Service初始化相关的操作。

* onStartCommand

onStartCommand: 在执行了startService方法之后，有可能会调用Service的onCreate方法，在这之后一定会执行Service的onStartCommand回调方法。也就是说，**如果多次执行了Context的startService方法，那么Service的onStartCommand方法也会相应的多次调用**。onStartCommand方法很重要，我们在该方法中根据传入的Intent参数进行实际的操作，比如会在此处创建一个线程用于下载数据或播放音乐等。

**public @StartResult int onStartCommand(Intent intent, @StartArgFlags int flags, int startId) {}**

当Android面临内存匮乏的时候，可能会销毁掉你当前运行的Service，然后待内存充足的时候可以重新创建Service，Service被Android系统强制销毁并再次重建的行为依赖于Service中onStartCommand方法的返回值。我们常用的返回值有三种值，START\_NOT\_STICKY、START\_STICKY和START\_REDELIVER\_INTENT，这三个值都是Service中的静态常量。

* **START\_NOT\_STICKY**

**如果返回START\_NOT\_STICKY，表示当Service运行的进程被Android系统强制杀掉之后，不会重新创建该Service，当然如果在其被杀掉之后一段时间又调用了startService，那么该Service又将被实例化。**那什么情境下返回该值比较恰当呢？如果我们某个Service执行的工作被中断几次无关紧要或者对Android内存紧张的情况下需要被杀掉且不会立即重新创建这种行为也可接受，那么我们便可将 onStartCommand的返回值设置为START\_NOT\_STICKY。举个例子，某个Service需要定时从服务器获取最新数据：通过一个定时器每隔指定的N分钟让定时器启动Service去获取服务端的最新数据。当执行到Service的onStartCommand时，在该方法内再规划一个N分钟后的定时器用于再次启动该Service并开辟一个新的线程去执行网络操作。假设Service在从服务器获取最新数据的过程中被Android系统强制杀掉，Service不会再重新创建，这也没关系，因为再过N分钟定时器就会再次启动该Service并重新获取数据。

* START\_STICKY

如果返回START\_STICKY，表示Service运行的进程被Android系统强制杀掉之后，Android系统会将该Service依然设置为started状态（即运行状态），**但是不再保存onStartCommand方法传入的intent对象，**然后Android系统会尝试再次重新创建该Service，并执行onStartCommand回调方法，但是onStartCommand回调方法的Intent参数为null，也就是onStartCommand方法虽然会执行但是获取不到intent信息。如果你的Service可以在任意时刻运行或结束都没什么问题，而且不需要intent信息，那么就可以在onStartCommand方法中返回START\_STICKY，比如一个用来播放背景音乐功能的Service就适合返回该值。

* START\_REDELIVER\_INTENT

如果返回START\_REDELIVER\_INTENT，表示Service运行的进程被Android系统强制杀掉之后，与返回START\_STICKY的情况类似，Android系统会将再次重新创建该Service，并执行onStartCommand回调方法，但是不同的是，**Android系统会再次将Service在被杀掉之前最后一次传入onStartCommand方法中的Intent再次保留下来并再次传入到重新创建后的Service的onStartCommand方法中，这样我们就能读取到intent参数。**只要返回START\_REDELIVER\_INTENT，那么onStartCommand重的intent一定不是null。如果我们的Service需要依赖具体的Intent才能运行（需要从Intent中读取相关数据信息等），并且在强制销毁后有必要重新创建运行，那么这样的Service就适合返回START\_REDELIVER\_INTENT。

* onBind

Service中的onBind方法是抽象方法，所以Service类本身就是抽象类，也就是onBind方法是必须重写的，即使我们用不到。在通过startService使用Service时，我们在重写onBind方法时，只需要将其返回null即可。onBind方法主要是用于给bindService方法调用Service时才会使用到。

* onDestroy

onDestroy: 通过startService方法启动的Service会无限期运行，只有当调用了Context的stopService或在Service内部调用stopSelf方法时，Service才会停止运行并销毁，在销毁的时候会执行Service回调函数。

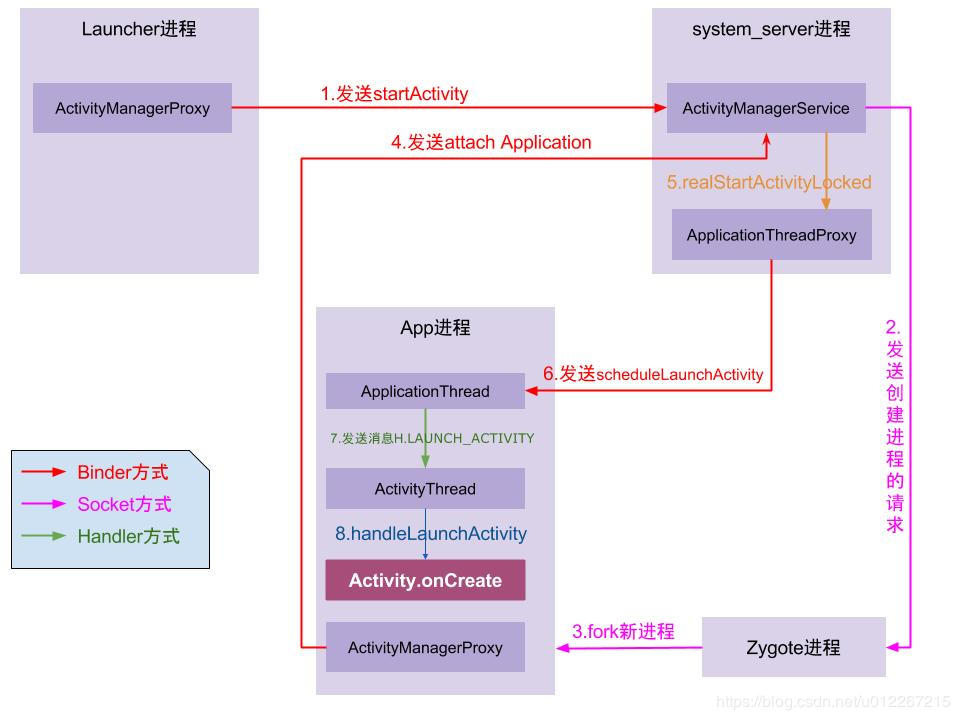
### 1.4 Activity

#### 1.4.1 Activity启动流程

用户点击屏幕上面的app icon，进入响应的app，背后经历了Activity和AMS的反反复复的通信过程。手机屏幕他就是一个activity，而这个activity所在的app被称之为Launcher，这是各个手机厂商提供的。

我们在开发app时，会在AndroidManifest.xml文件中定义默认启动的activity，设置activity的action和category属性标签。Launcher位每个app的icon图标提供了启动这个app时所需要的Intent信息。这些信息在app安装时由PackageManagerService从app的AndroidManifest.xml文件中读取。

#### 1.4.2 点击Launcher图标来启动Activity



**启动流程进程间简单分析：**

Zygote进程 –> SystemServer进程 –> 各种系统服务 –> 应用进程

在Actvity启动过程中，其实是应用进程与SystemServer进程相互配合启动Activity的过程，其中应用进程主要用于执行具体的Activity的启动过程，回调生命周期方法等操作，而SystemServer进程则主要是调用其中的各种服务，将Activity保存在栈中，协调各种系统资源等操作。

通过ActivityManagerNative –> ActivityManagerService实现了应用进程与SystemServer进程的通讯

通过AppicationThread <– IApplicationThread实现了SystemServer进程与应用进程的通讯

ActivityManagerProxy相当于Proxy

ActivityManagerNative就相当于Stub

ActivityManagerService是ActivityManagerNative的具体实现，换句话说，就是AMS才是服务端的具体实现！

ApplicationThreadProxy相当于Proxy

ApplicationThreadNative相当于Stub

ApplicationThread相当于服务器端，代码真正的实现者！

（1）点击桌面App图标，Launcher进程采用Binder IPC向system\_server进程发起startActivity请求；

（2）system\_server进程接收到请求后，向zygote进程发送创建进程的请求；

（3）Zygote进程fork出新的子进程，即App进程；

（4）App进程，通过Binder IPC向sytem\_server进程发起attachApplication请求；

（5）system\_server进程在收到请求后，进行一系列准备工作后，再通过binder IPC向App进程发送scheduleLaunchActivity请求；

（6）App进程的binder线程（ApplicationThread）在收到请求后，通过handler向主线程发送LAUNCH\_ACTIVITY消息；

（7）主线程在收到Message后，通过反射机制创建目标Activity，并回调Activity.onCreate()等方法。

（8）到此，App便正式启动，开始进入Activity生命周期，执行完onCreate/onStart/onResume方法，UI渲染结束后便可以看到App的主界面。

**涉及到的类有：**

1. **Instrumentation**
2. **ActivityThread**
3. **H**
4. **LoadedApk**
5. **AMS**
6. **ActivityManagerNative和ActivityManagerProxy**
7. **ApplicationThread和ApplicationThreadProxy**

**第一阶段：Launcher通知AMS**

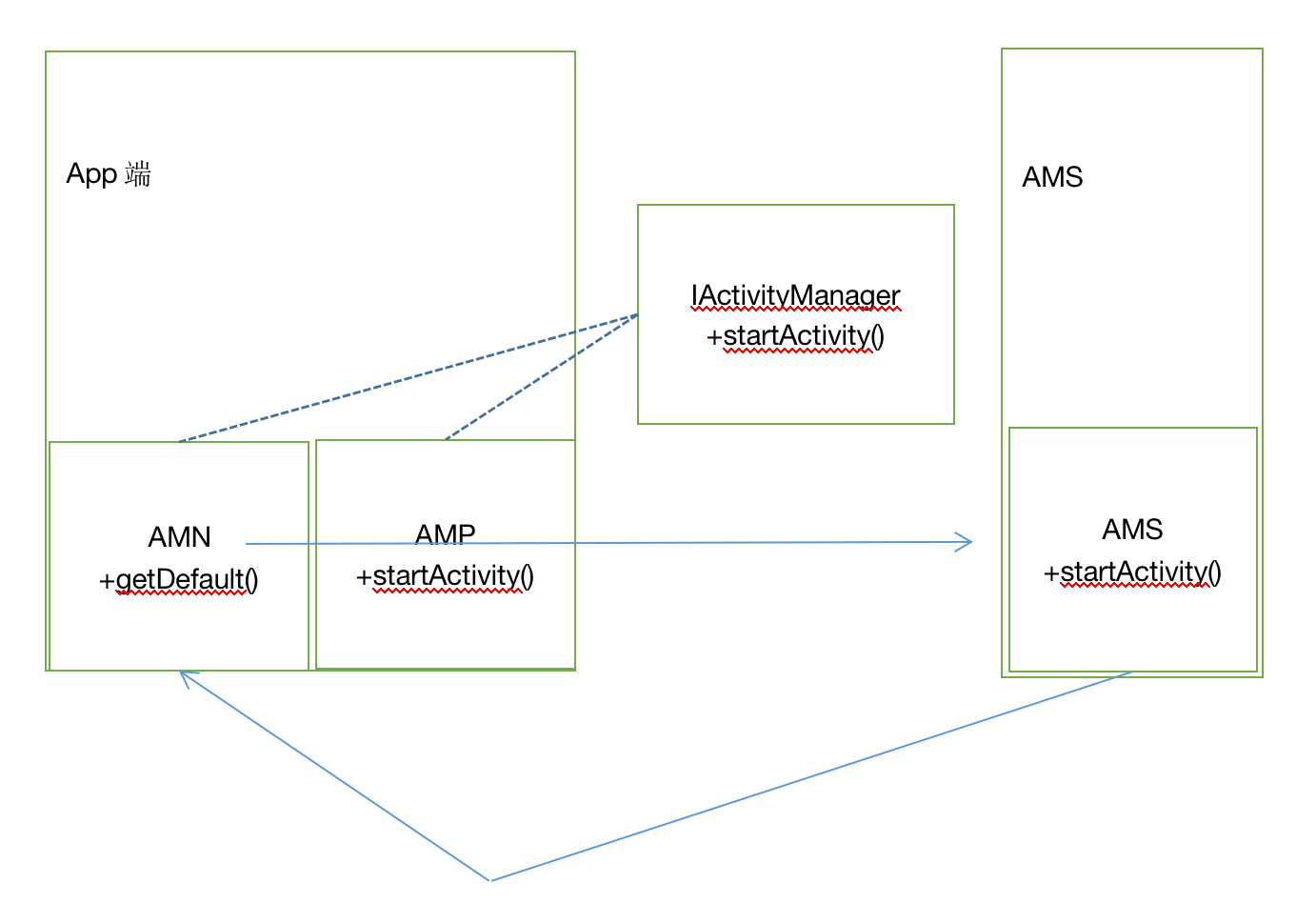
1. 启动activity调用startActivity方法，最后会调用startActivityForResult方法并传入code=-1表示不关心是否启动成功。
2. startActivityForResult内部会调用Instrumentation的execStartActivity方法，这个方法有几个参数需要注意：

****

**contextThread:**  位ActivityThread的getApplicationThread获取到一个IBinder对象，这个对象类型是ApplicationThread，代表Launcher所在的app进程。

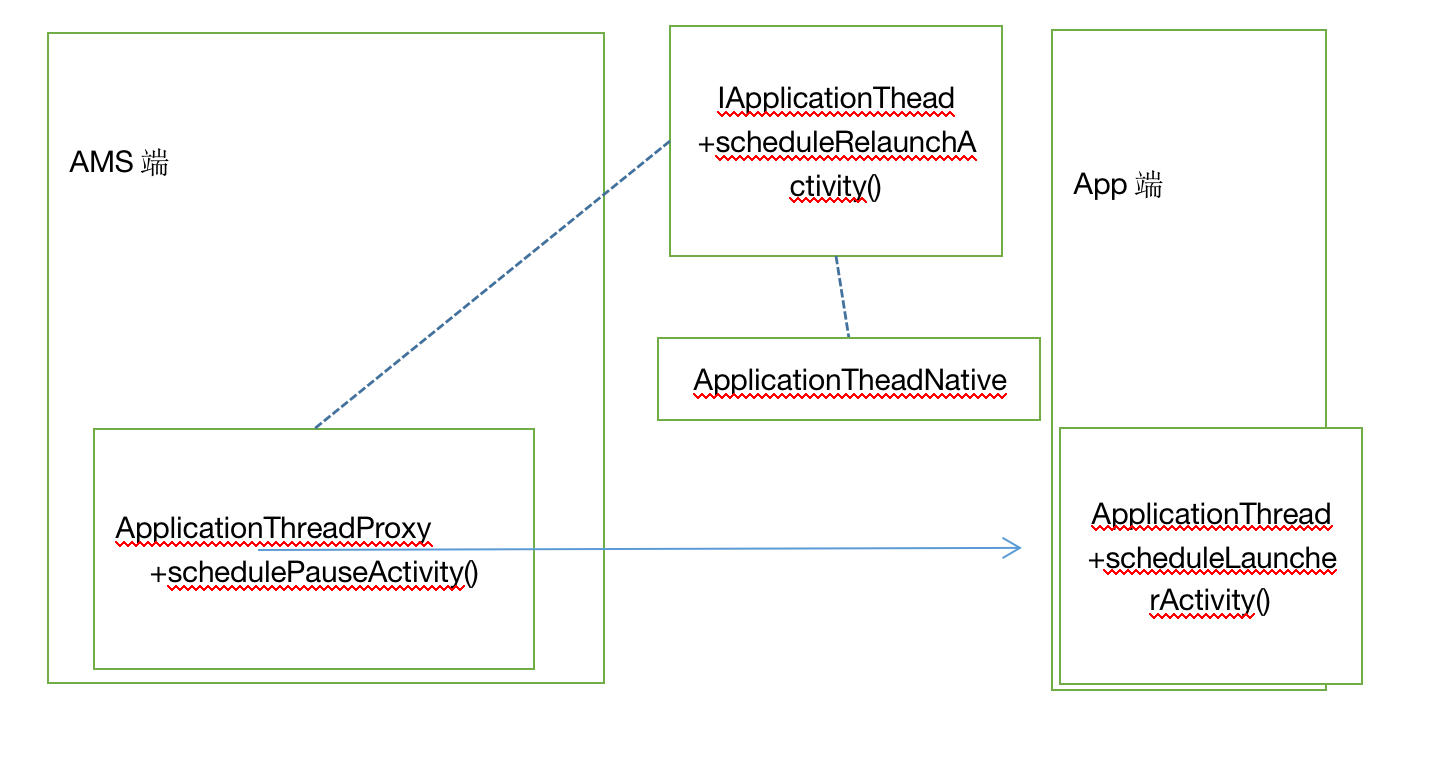
**Token：** 也是一个IBinder类型，代表Launcher这个activity也通过Instrumentation传给AMS，AMS就知道是谁向AMS发起了请求。

1. 接着，通过Instrumentation，Activity将数据传递给ActivityManagerNativ。
2. AMN通过getDefault方法，从ServiceManager中取得一个activity对象将其包装成ActivityManagerProxy(AMP)对象，他就是AMS的代理对象。
3. AMP调用startActivity方法就会将数据写入到另一个进程AMS然后等待AMS返回结果。



**第二阶段：AMS处理Launcher传过来的信息**

AMS给Launcher发送消息，它在AMS这边保存一个ActivityRecord对象，这个对象里面有个ApplicationThreadProxy，是一个Binder代理对象，就是ApplicationThread。



AMS通过ApplicationThreadProxy发送消息，在App端则通过ApplicationThread来接收消息。

**第三阶段：**

ApplicationThread接收到来自AMS的消息后，调用ActivityThread（UI主线程）的sendMessage方法，向Launcher的主线程消息队列发送一个PAUSE\_ACTIVITY的消息。

这里面涉及到一个H类，AMS给activity发送的所有消息，以及给三大组件发送的消息都是从H类经过，因此可以从这里做插件化。

**第四阶段：AMS启动新的进程**

新的进程以ActivityThread的main方法作为入口，启动新进程就是启动一个新的app。在这个方法中，创建一个主线程Looper也就是MainLooper，其次创建Application，其次在创建LoadedApk对象，然后创建ContextImpl通过反射创建目标Application调用其attach的方法最后再调用onCreate方法。

**第五阶段：启动相关Activity**

在Binder的另一端，App通过APT接收到AMS的消息，仍然在H的handleMessage中处理。在这个handleMessage中会调用ActivityThread的handleLaunchActivity方法。

流程梳理：

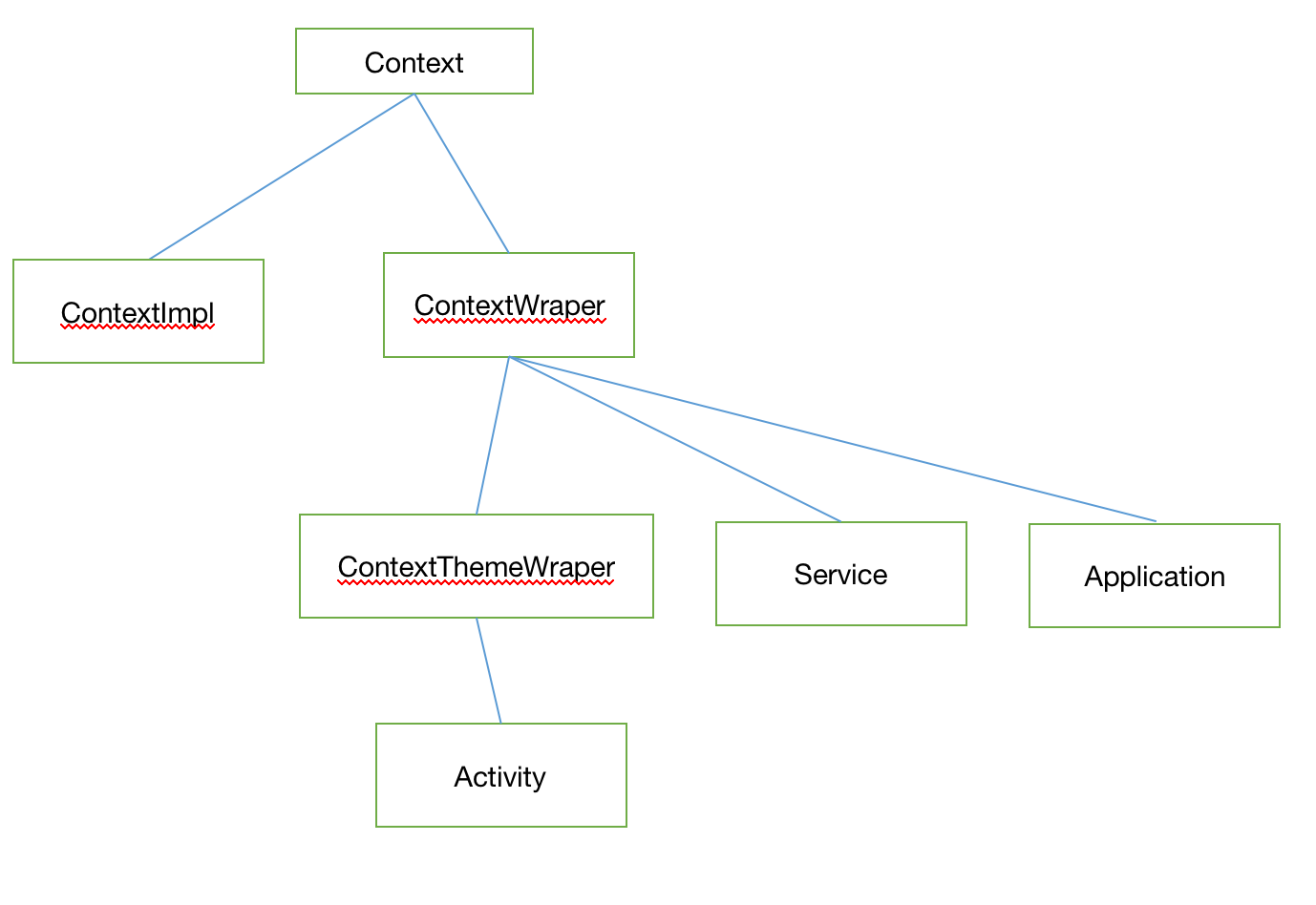
每次ApplicationThread执行ActivityThread的sendMessage方法，在这里拼装消息，丢给H的switch方法去执行，决定执行ActivityThread的哪个方法，每次都是这样。

handleLaunchActivity方法内容：

1. 通过Instrumentation的newActivity方法，创建要启动的Activity实例
2. 为这个Activity创建一个上下文Context，并与Activity关联
3. 通过Instrumentation的callActivityOnCreate方法执行Activity的onCreate方法从而启动Activity。

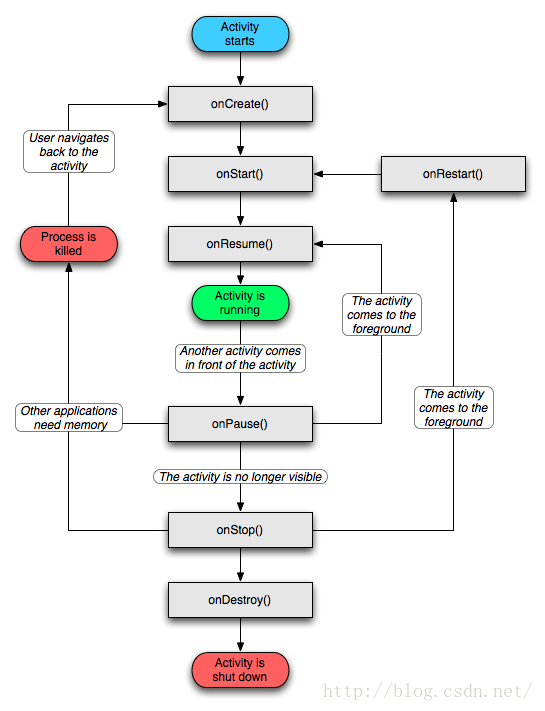
#### 1.4.2.1 ActivityThread.main





context关系图

#### 1.4.3 Activity生命周期

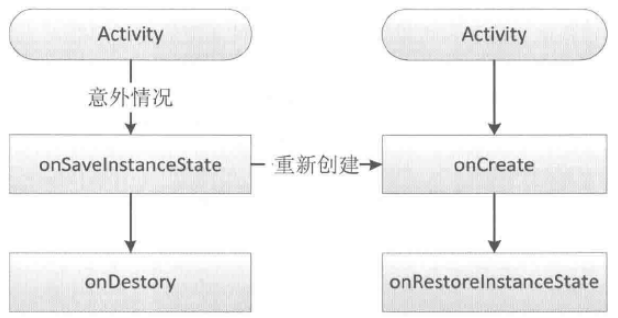




**onSaveInstanceState和onRestoreInstanceState调用的过程和时机**

**（1）调用时机**：Activity的异常情况下（例如转动屏幕或者被系统回收）的情况下，会调用到onSaveInstanceState和onRestoreInstanceState。其他情况不会触发这个过程。但是按Home键或者启动新Activity仍然会单独触发onSaveInstanceState的调用。

**（2）调用过程**：旧的Activity要被销毁时，由于是**异常情况**下的，所以除了正常调用onPause, onStop, onDestroy方法外，还会在**调用onStop方法前，调用onSaveInstanceState方法**。新的Activity重建时，我们就可以通过onRestoreInstanceState方法取出之前保存的数据并恢复，**onRestoreInstanceState的调用时机在onCreate之后**。



#### 1.4.4 启动模式

**（1）Standard模式:**

Activity可以有多个实例，每次启动Activity，无论任务栈中是否已经有这个Activity的实例，系统都会创建一个新的Activity实例

**（2）SingleTop模式:**

当一个singleTop模式的Activity已经位于任务栈的栈顶，再去启动它时，不会再创建新的实例,如果不位于栈顶，就会创建新的实例，会调用onNewIntent方法

**（3）SingleTask模式:**

如果Activity已经位于栈顶，系统不会创建新的Activity实例，和singleTop模式一样。但Activity已经存在但不位于栈顶时，系统就会把该Activity移到栈顶，并把它上面的activity出栈，onNewIntent方法

**（4）SingleInstance模式:**

singleInstance模式也是单例的，但和singleTask不同，singleTask只是任务栈内单例，系统里是可以有多个singleTaskActivity实例的，而singleInstance Activity在整个系统里只有一个实例，启动一singleInstanceActivity时，系统会创建一个新的任务栈，并且这个任务栈只有他一个Activity生命周期

**onNewIntent的作用和调用时机？**

* **调用时机**：如果Activity的启动模式是：singleTop, singleTask, singleInstance，在复用这些Acitivity时就会在调用onStart方法前调用onNewIntent方法
* **作用**：让已经创建的Activity处理新的Intent。

#### 1.4.5 两个Activity跳转的生命周期

1. **启动A**  
   onCreate - onStart - onResume
2. **在A中启动B**  
   ActivityA onPause  
   ActivityB onCreate  
   ActivityB onStart  
   ActivityB onResume  
   ActivityA onStop
3. **从B中返回A（按物理硬件返回键）**  
   ActivityB onPause  
   ActivityA onRestart  
   ActivityA onStart  
   ActivityA onResume  
   ActivityB onStop  
   ActivityB onDestroy

**4.继续返回**  
ActivityA onPause  
ActivityA onStop  
ActivityA onDestroy

#### 1.4.6 onRestart的调用场景

1. 按下home键之后，然后切换回来，会调用onRestart()。  
   **（2）** 从本Activity跳转到另一个Activity之后，按back键返回原来Activity，会调用onRestart()；  
   **（3）** 从本Activity切换到其他的应用，然后再从其他应用切换回来，会调用onRestart()；

#### 1.4.7 **横竖屏切换生命周期**

说下Activity的横竖屏的切换的生命周期，用那个方法来保存数据，两者的区别。触发在什么时候在那个方法里可以获取数据等。

1，默认情况下横竖屏切换，  
a，3.2之前的版本先执行onPause()，再执行onSaveInstanceState(Bundle outState)  
b，3.2版本开始先执行onSaveInstanceState(Bundle outState)，再执行onPause()

2，配置android:configChanges="orientation"，防止横竖屏切换重新创建Activity  
a，如果targetSdkVersion<=12，所有版本都有效  
b，如果targetSdkVersion>12，3.2版本开始需要多加个screenSize



### 1.5 Handler原理

Handler，Message，looper和MessageQueue构成了安卓的消息机制，handler创建后可以通过sendMessage将消息加入消息队列，然后looper不断的将消息从MessageQueue中取出来，回调到Hander的handleMessage方法，从而实现线程的通信。I0

**主线程：**

从两种情况来说，第一在UI线程创建Handler,此时我们不需要手动开启looper，因为在应用启动时，在ActivityThread的main方法中就创建了一个当前主线程的looper，并开启了消息队列，消息队列是一个无限循环，为什么无限循环不会ANR?因为可以说，应用的整个生命周期就是运行在这个消息循环中的，安卓是由事件驱动的，Looper.loop不断的接收处理事件，每一个点击触摸或者Activity每一个生命周期都是在Looper.loop的控制之下的，looper.loop一旦结束，应用程序的生命周期也就结束了。我们可以想想什么情况下会发生ANR，第一，事件没有得到处理，第二，事件正在处理，但是没有及时完成，而对事件进行处理的就是looper，所以只能说事件的处理如果阻塞会导致ANR，而不能说looper的无限循环会ANR

**子线程：**

另一种情况就是在子线程创建Handler,此时由于这个线程中没有默认开启的消息队列，所以我们需要手动调用looper.prepare(),并通过looper.loop开启消息  
主线程Looper从消息队列读取消息，当读完所有消息时，主线程阻塞。子线程往消息队列发送消息，并且往管道文件写数据，主线程即被唤醒，从管道文件读取数据，主线程被唤醒只是为了读取消息，当消息读取完毕，再次睡眠。因此loop的循环并不会对CPU性能有过多的消耗。

#### 1.5.1 主线程的消息循环机制是什么

**Activity 的生命周期都是依靠主线程的 Looper.loop**，当收到不同 Message 时

则采用相应措施：一旦退出消息循环，那么你的程序也就可以退出了。 从消息队列中取消息可能会阻塞，取到消息会做出相应的处理。如果某个消息处理时间 过长，就可能会影响 UI 线程的刷新速率，造成卡顿的现象。

thread.attach(false)方法函数中便会创建一个 Binder 线程（具体是指

ApplicationThread，Binder 的服务端，用于接收系统服务 AMS 发送来的事件），

该 Binder 线程通过 Handler 将 Message 发送给主线程。「Activity 启动过程」

比如收到 msg=H.LAUNCH\_ACTIVITY，则调用 ActivityThread.handleLaunchActivity()方法，最终会通过反射机制，创建 Activity 实例，然后再执行 Activity.onCreate()等方法； 再比如收到 msg=H.PAUSE\_ACTIVITY，则调用 ActivityThread.handlePauseActivity()方法，最终会执行 Activity.onPause()等方

法。主线程的消息又是哪来的呢？当然是app进程中的其他线程通过Handler发送给主线程

### 1.6 Requestlayout，onlayout，onDraw，DrawChild

requestLayout()方法 ：会导致调用measure()过程 和 layout()过程 。 说明：只是对View树重新布局layout过程包括measure()和layout()过程，如果view的l,t,r,b没有必变，那就不会触发onDraw；但是如果这次刷新是在动画里，mDirty非空，就会导致onDraw。

onLayout()方法(如果该View是ViewGroup对象，需要实现该方法，对每个子视图进行布局)

onDraw()方法绘制视图本身 (每个View都需要重载该方法，ViewGroup不需要实现该方法)

drawChild()去重新回调每个子视图的draw()方法

### 1.7 invalidate和postInvalidate的区别及使用

View.invalidate(): 层层上传到父级，直到传递到ViewRootImpl后触发了scheduleTraversals()，然后整个View树开始重新按照View绘制流程进行重绘任务。

invalidate:在ui线程刷新view

postInvalidate：在工作线程刷新view（底层还是handler）其实它的原理就是invalidate+handler  
View.postInvalidate最终会调用ViewRootImpl.dispatchInvalidateDelayed()方法

**public void dispatchInvalidateDelayed(View view, long delayMilliseconds) {**

**Message msg = mHandler.obtainMessage(MSG\_INVALIDATE, view);**

**mHandler.sendMessageDelayed(msg, delayMilliseconds);**

**}**

这里的mHandler是ViewRootHandler实例，在该Handler的handleMessage方法中调用了view.invalidate()方法。

**case MSG\_INVALIDATE:**

**((View) msg.obj).invalidate();**

**break;**

### 1.8 如何优化自定义view

1. 不要在onDraw或是onLayout()中去创建对象，因为onDraw()方法可能会被频繁调用，可以在view的构造函数中进行创建对象；
2. 降低view的刷新频率，尽可能减少不必要的调用invalidate()方法。或是调用带四种参数不同类型的invalidate()，而不是调用无参的方法。无参变量需要刷新整个view，而带参数的方法只需刷新指定部分的view。在onDraw()方法中减少冗余代码。
3. 使用硬件加速，GPU硬件加速可以带来性能增加。

4）状态保存与恢复，如果因内存不足，Activity置于后台被杀重启时，View应尽可能保存自己属性，可以重写onSaveInstanceState和onRestoreInstanceState方法，状态保存。

### 1.9 动画

#### 1.9.1 动画分类

（1）frame 帧动画：AnimationDrawable控制animation-list.xml布局。

（2）tween 补间动画：通过指定View的初末状态和变化方式，对View的内容完成一系列的图形变换来实现动画效果， Alpha, Scale ,Translate, Rotate。

（3）PropertyAnimation 属性动画：3.0引入，属性动画核心思想是对值的变化。

#### 1.9.2 属性动画&&补间动画的性能差异：

（1）属性动画操作的是对象的实例属性，例如translationX,然后反射调用set,geView动画:t方法，多个属性动画同时执行，会频繁反射调用类方法，降低性能。

（2）补间动画只产生了一个动画效果，其真实的坐标并没有发生改变，是效果一直在发生变化，没有频繁反射调用方法的耗费性能操作。

#### 1.9.3 原理及特点

**（1）帧动画：**

是在xml中定义好一系列图片之后，使用AnimatonDrawable来播放的动画。

**（2）View动画：**

只是影像变化，view的实际位置还在原来地方。

**（3）属性动画：**

插值器：作用是根据时间流逝的百分比来计算属性变化的百分比。

估值器：在1的基础上由这个东西来计算出属性到底变化了多少数值的类。

其实就是利用插值器和估值器，来计出各个时刻View的属性，然后通过改变View的属性来实现View的动画效果。

#### 1.9.4 区别

属性动画才是真正的实现了 view 的移动，补间动画对view 的移动更像是在不同地方绘制了一个影子，实际对象还是处于原来的地方。 当动画的 repeatCount 设置为无限循环时，如果在Activity退出时没有及时将动画停止，属性动画会导致Activity无法释放而导致内存泄漏，而补间动画却没问题。 xml 文件实现的补间动画，复用率极高。在 Activity切换，窗口弹出时等情景中有着很好的效果。 使用帧动画时需要注意，不要使用过多特别大的图，容导致内存不足。

#### 1.9.5 为什么属性动画移动后仍可点击？

播放补间动画的时候，我们所看到的变化，都只是临时的。而属性动画呢，它所改变的东西，却会更新到这个View所对应的矩阵中，所以当ViewGroup分派事件的时候，会正确的将当前触摸坐标，转换成矩阵变化后的坐标，这就是为什么播放补间动画不会改变触摸区域的原因了。

### 1.10 RecyclerView复用机制

**（1）在**RecyclerView**中，并不是每次绘制表项，都会重新创建**ViewHolder**对象，也不是每次都会重新绑定**ViewHolder**数据。**

（2）RecyclerView**通过**Recycler**获得下一个待绘制表项。**

（3） Recycler**有4个层次用于缓存**ViewHolder**对象，优先级从高到底依次为**ArrayList<ViewHolder> **mAttachedScrap、**ArrayList<ViewHolder> **mCachedViews、**ViewCacheExtension **mViewCacheExtension、RecycledViewPool** mRecyclerPool**。如果四层缓存都未命中，则重新创建并绑定**ViewHolder**对象**

（4） RecycledViewPool**对**ViewHolder**按**viewType**分类存储（通过**SparseArray**），同类**ViewHolder**存储在默认大小为5的**ArrayList**中**

（5） **从**mRecyclerPool**中复用的**ViewHolder**需要重新绑定数据，从**mAttachedScrap**中复用的**ViewHolder**不要重新出创建也不需要重新绑定数据**。

<https://www.cnblogs.com/dasusu/p/7746946.html>

## 2 Serializable与Parcable的区别

什么是序列化 —— 序列化，表示将一个对象转换成可存储或可传输的状态。序列化后的对象可以在网络上进行传输，也可以存储到本地。

由于在系统底层，数据的传输形式是简单的字节序列形式传递，即在底层，系统不认识对象，只认识字节序列，而为了达到进程通讯的目的，需要先将数据序列化，而**序列化就是将对象转化字节序列的过程**。相反地，当字节序列被运到相应的进程的时候，进程为了识别这些数据，就要将其**反序列化，即把字节序列转化为对象**。

**怎么通过序列化传输对象？**

Android中Intent如果要传递类对象，可以通过两种方式实现。

方式一：Serializable，要传递的类实现Serializable接口传递对象，

方式二：Parcelable，要传递的类实现Parcelable接口传递对象。

Serializable（Java自带）：  
Serializable是序列化的意思，表示将一个对象转换成可存储或可传输的状态。序列化后的对象可以在网络上进行传输，也可以存储到本地。

Parcelable（android 专用）：  
除了Serializable之外，使用Parcelable也可以实现相同的效果，  
不过不同于将对象进行序列化，Parcelable方式的实现原理是将一个完整的对象进行分解，  
而分解后的每一部分都是Intent所支持的数据类型，这样也就实现传递对象的功能了。

**实现序列化的作用**

1）永久性保存对象，保存对象的字节序列到本地文件中；

2）通过序列化对象在网络中传递对象；

3）通过序列化在进程间传递对象。

**选择序列化方法的原则**

1）在使用内存的时候，Parcelable比Serializable性能高，所以推荐使用Parcelable。

2）Serializable在序列化的时候会产生大量的临时变量，从而引起频繁的GC。

3）Parcelable不能使用在要将数据存储在磁盘上的情况，因为Parcelable不能很好的保证数据的持续性在外界有变化的情况下。尽管Serializable效率低点，但此时还是建议使用Serializable 。

android上应该尽量采用Parcelable，效率至上

编码上：

Serializable代码量少，写起来方便

Parcelable代码多一些

效率上：

Parcelable的速度比高十倍以上

serializable的迷人之处在于你只需要对某个类以及它的属性实现Serializable 接口即可。Serializable 接口是一种标识接口（marker interface），这意味着无需实现方法，Java便会对这个对象进行高效的序列化操作。

这种方法的缺点是使用了反射，序列化的过程较慢。这种机制会在序列化的时候创建许多的临时对象，容易触发垃圾回收。

Parcelable方式的实现原理是将一个完整的对象进行分解，而分解后的每一部分都是Intent所支持的数据类型，这样也就实现传递对象的功能了

## 3 三级缓存原理

### 3.1 三级缓存原理

当Android端需要获得数据时比如获取网络中的图片，首先从内存中查找（按键查找），内存中没有的再从磁盘文件或sqlite中去查找，若磁盘中也没有才通过网络获取

### 3.2 LruCache底层实现原理：

LruCache（Least Recently Used）中Lru算法的实现就是通过LinkedHashMap来实现的。LinkedHashMap继承于HashMap，它使用了一个双向链表来存储Map中的Entry顺序关系，  
对于get、put、remove等操作，LinkedHashMap除了要做HashMap做的事情，还做些调整Entry顺序链表的工作。

LruCache中将LinkedHashMap的顺序设置为LRU顺序来实现LRU缓存，每次调用get(也就是从内存缓存中取图片)，则将该对象移到链表的尾端。  
调用put插入新的对象也是存储在链表尾端，这样当内存缓存达到设定的最大值时，将链表头部的对象（近期最少用到的）移除。

## 4 进程保活

**（1）**Service设置成START\_STICKY kill 后会被重启(等待5秒左右)，重传Intent，保持与重启前一样

**（2）**通过 startForeground将进程设置为前台进程， 做前台服务，优先级和前台应用一个级别，除非在系统内存非常缺，否则此进程不会被 kill

**（3）**双进程Service： 让2个进程互相保护对方，其中一个Service被清理后，另外没被清理的进程可以立即重启进程

1. 用C编写守护进程(即子进程) : Android系统中当前进程(Process)fork出来的子进程，被系统认为是两个不同的进程。当父进程被杀死的时候，子进程仍然可以存活，并不受影响(Android5.0以上的版本不可行）联系厂商，加入白名单

**（5）**锁屏状态下，开启一个一像素Activity

## 5. view部分知识点

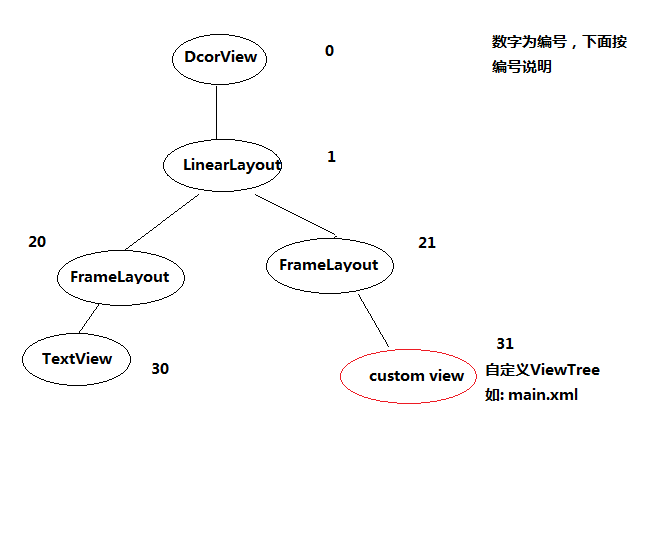
### 5.1 DecorView浅析

[Android View源码解读：浅谈DecorView与ViewRootImpl](https://www.jianshu.com/p/687010ccad66)

DecorView为整个Window界面的最顶层View，它只有一个子元素LinearLayout。代表整个Window界面，包含通知栏、标题栏、内容显示栏三块区域。其中LinearLayout中有两个FrameLayout子元素。



1



* （20）标题栏FrameLayout

其中（20）为标题栏显示界面，只有一个TextView显示应用的名称。

* （21）内容栏FrameLayout

其中（21）位内容栏显示界面，就是setContentView()方法载入的布局界面。

#### 5.1.1 DecorView的作用

1. DecorView是顶级View，本质是一个FrameLayout
2. 它包含两部分，标题栏和内容栏，都是FrameLayout
3. 内容栏id是content，也就是activity中设置setContentView的部分，最终将布局添加到id为content的FrameLayout中。
4. 获取content：ViewGroup content=findViewById（R.android.id.content）
5. 获取设置的View：content.getChildAt(0).

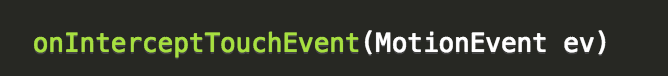
#### 5.1.2 使用总结

1. 每个Activity都包含一个Window对象，Window对象通常是由PhoneWindow实现的。
2. PhoneWindow：将DecorView设置为整个应用窗口的根View，是Window的实现类。它是Android中的最基本的窗口系统，每个Activity均会创建一个PhoneWindow对象，是Activity和整个View系统交互的接口。
3. DecorView：是顶层视图，将要显示的具体内容呈现在PhoneWindow上，DecorView是当前Activity所有View的祖先，它并不会向用户呈现任何东西。

### 5.2 View的事件分发

[图解 Android 事件分发机制](https://www.jianshu.com/p/e99b5e8bd67b)

#### 5.2.1 ViewGroup事件分发

****

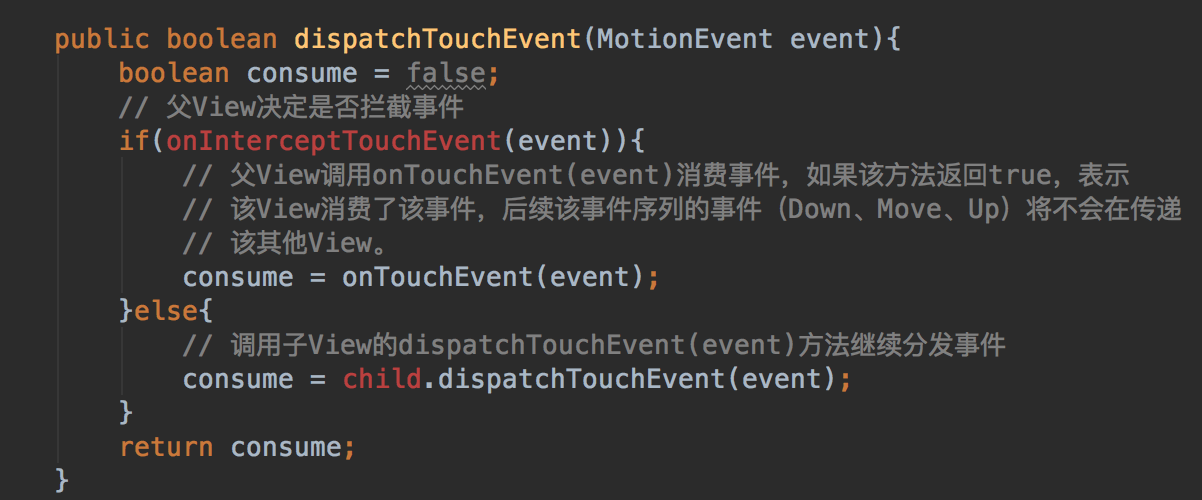
****

当一个点击事件产生后，它的传递过程将遵循如下顺序：

**Activity -> Window -> View**

事件总是会传递给Activity，之后Activity再传递给Window，最后Window再传递给顶级的View，顶级的View在接收到事件后就会按照事件分发机制去分发事件。如果一个View的onTouchEvent返回了FALSE，那么它的父容器的onTouchEvent将会被调用，依次类推，如果所有都不处理这个事件的话，那么Activity将会处理这个事件。

**对于ViewGroup的事件分发过程，大概是这样的：如果顶级的ViewGroup拦截事件即onInterceptTouchEvent返回true的话，则事件会交给ViewGroup处理，如果ViewGroup的onTouchListener被设置的话，则onTouch将会被调用，否则的话onTouchEvent将会被调用，也就是说：两者都设置的话，onTouch将会屏蔽掉onTouchEvent，在onTouchEvent中，如果设置了onClickerListener的话，那么onClick将会被调用。如果顶级ViewGroup不拦截的话，那么事件将会被传递给它所在的点击事件的子view，这时候子view的dispatchTouchEvent将会被调用**

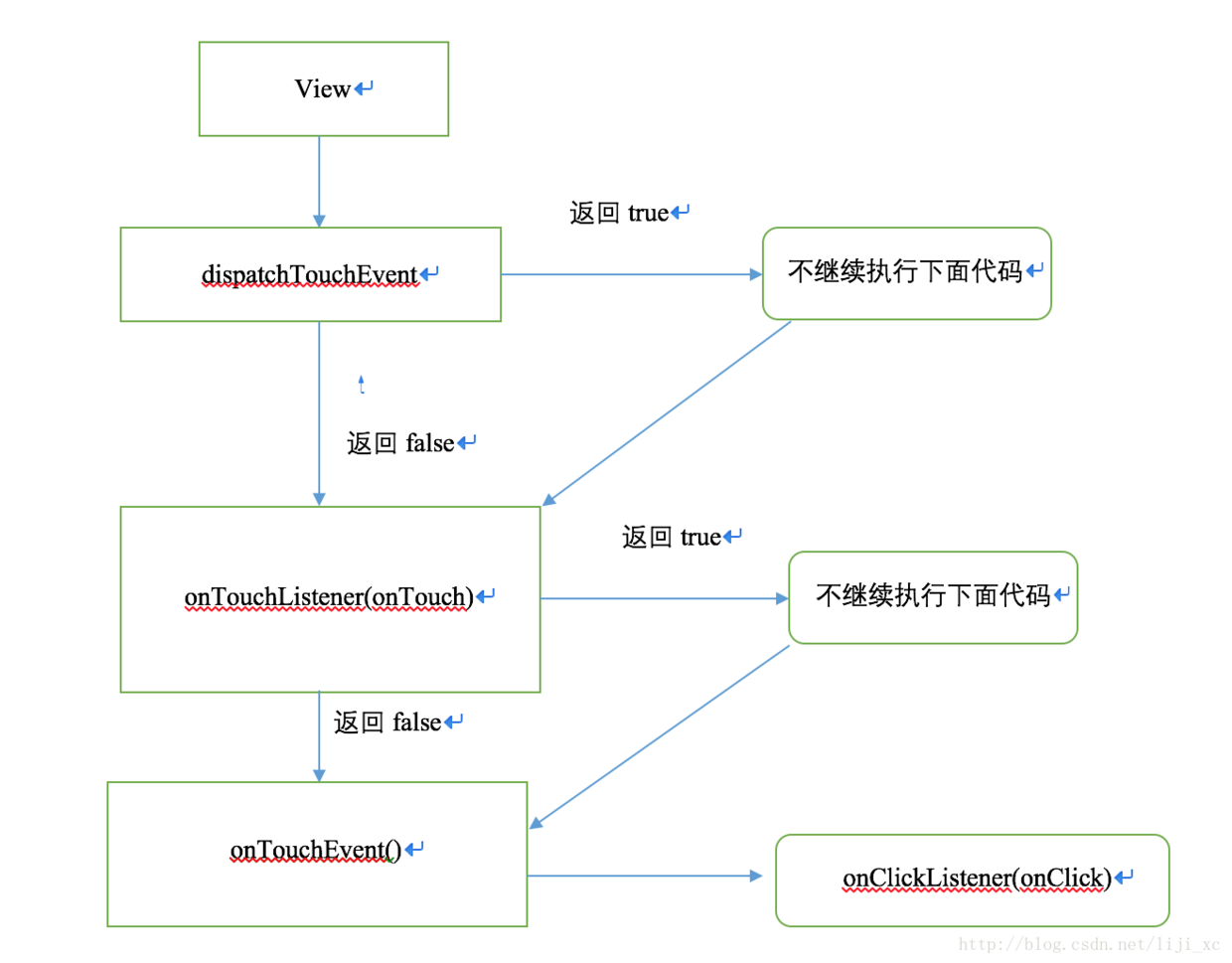


#### 5.2.2 View的事件分发

dispatchTouchEvent -> onTouch(setOnTouchListener) -> onTouchEvent -> onClick

#### 5.2.3 onTouch和onTouchEvent的区别

两者都是在dispatchTouchEvent中调用的，onTouch优先于onTouchEvent，如果onTouch返回true，那么onTouchEvent则不执行，及onClick也不执行。



#### 5.2.4 滑动冲突解决

**外部拦截法：**

(onInterceptTouchEvent )，如果父容器需要则拦截，如果不需要则不拦截，称为外部拦截法

**内部拦截法：**

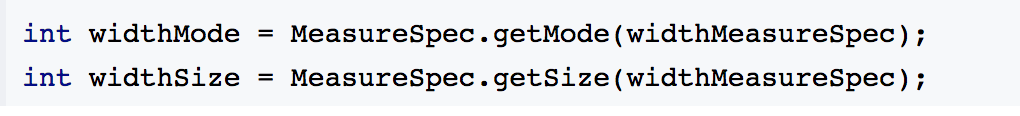
父容器不拦截任何事件，将所有事件传递给子元素，如果子元素需要则消耗掉，如果不需要则通过requestDisallowInterceptTouchEvent方法(请求父类不要拦截,返回值为true时不拦截,返回值为false时为拦截)交给父容器处理，称为内部拦截法，

### 5.3 View的绘制

#### 5.3.1 [onMeasure(int widthMeasureSpec, int heightMeasureSpec)](https://blog.csdn.net/huachao1001/article/details/51577291)

在xml布局文件中，我们的layout\_width和layout\_height参数可以不用写具体的尺寸，而是wrap\_content或者是match\_parent。这两个设置并没有指定真正的大小，可是我们绘制到屏幕上的View必须是要有具体的宽高的，正是因为这个原因，我们必须自己去处理和设置尺寸。当然了，View类给了默认的处理，但是如果View类的默认处理不满足我们的要求，我们就得重写onMeasure函数啦~。

一个int整数，里面放了测量模式和尺寸大小。int型数据占用32个bit，而google实现的是，将int数据的前面2个bit用于区分不同的布局模式，后面30个bit存放的是尺寸的数据。



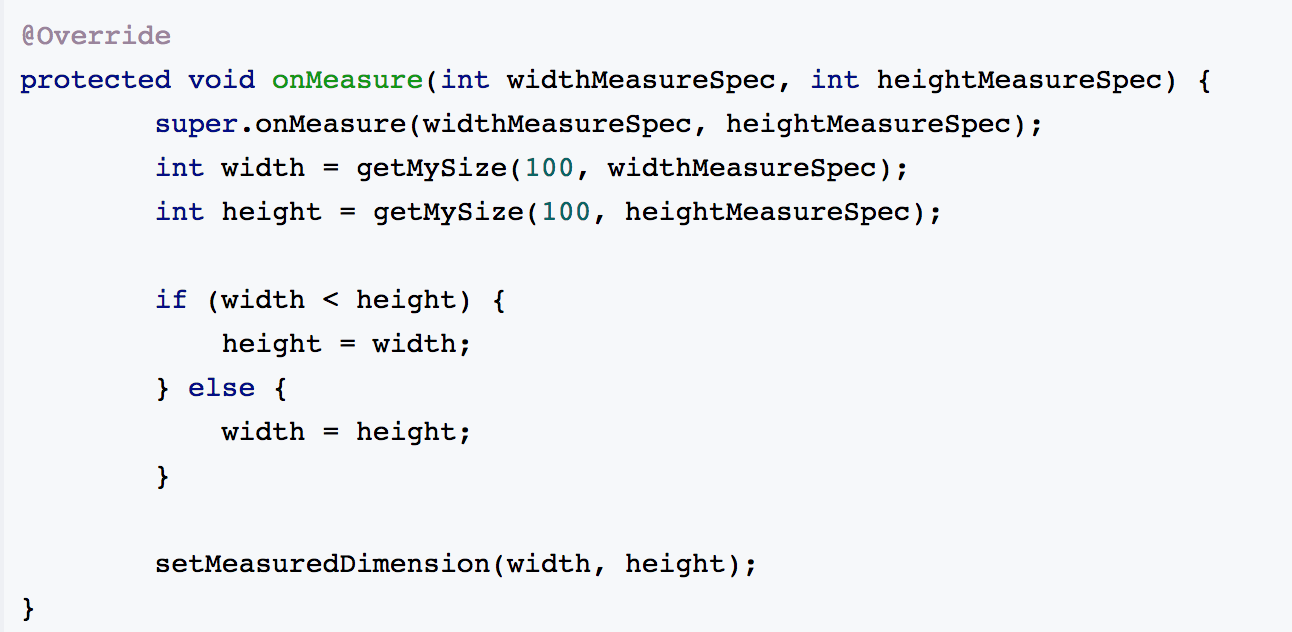


**match\_parent—>EXACTLY**。怎么理解呢？match\_parent就是要利用父View给我们提供的所有剩余空间，而父View剩余空间是确定的，也就是这个测量模式的整数里面存放的尺寸。

**wrap\_content—>AT\_MOST**。怎么理解：就是我们想要将大小设置为包裹我们的view内容，那么尺寸大小就是父View给我们作为参考的尺寸，只要不超过这个尺寸就可以啦，具体尺寸就根据我们的需求去设定。

**固定尺寸（如100dp）—>EXACTLY**。用户自己指定了尺寸大小，我们就不用再去干涉了，当然是以指定的大小为主啦。

****

****

#### 5.3.2 onDraw

**自定义属性**



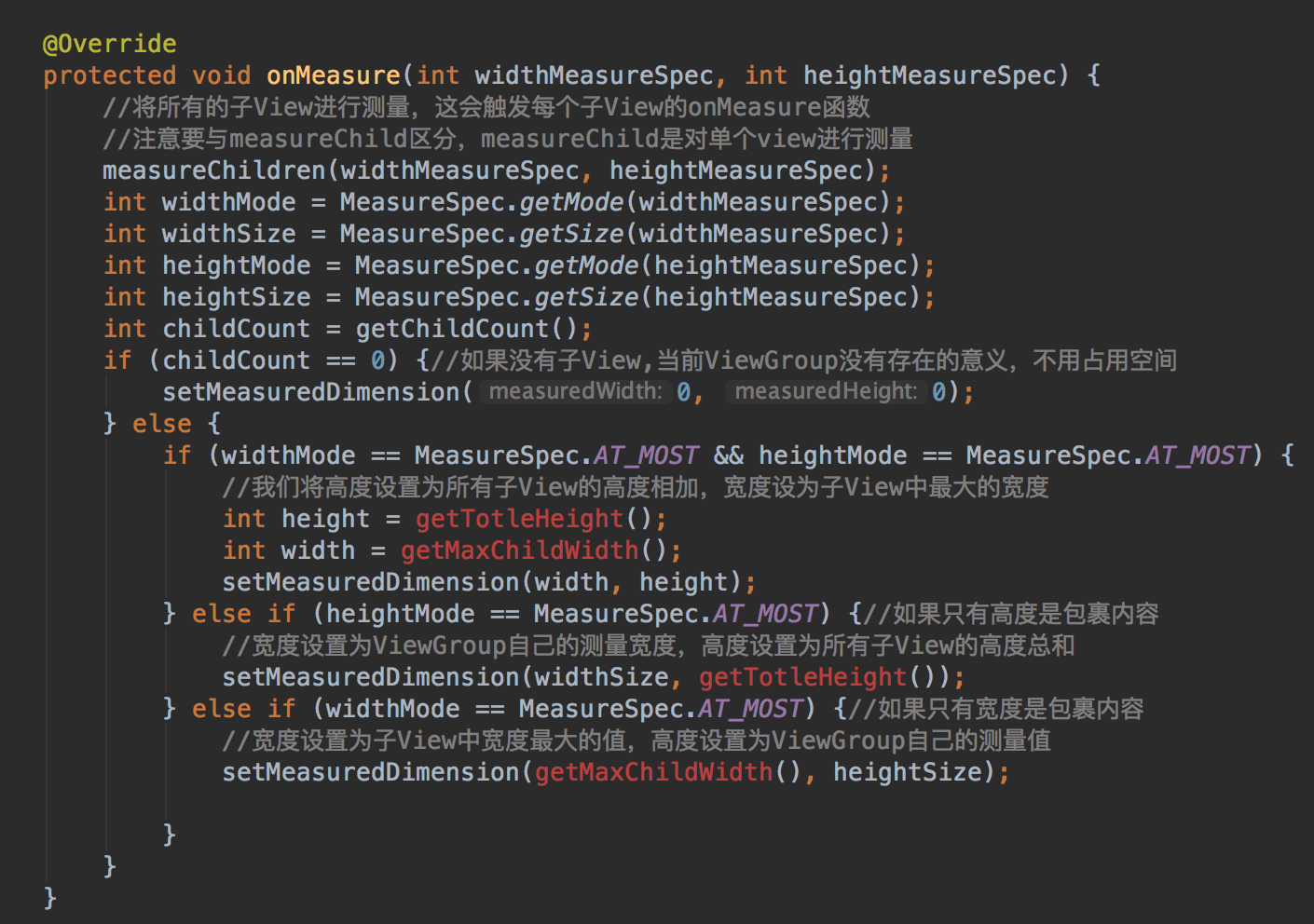
**命名空间："http://schemas.android.com/apk/res-auto"**



### 5.4 ViewGroup的绘制

自定义ViewGroup可就没那么简单啦~，因为它不仅要管好自己的，还要兼顾它的子View。我们都知道ViewGroup是个View容器，它装纳child View并且负责把child View放入指定的位置。

1. 首先，我们得知道各个子View的大小吧，只有先知道子View的大小，我们才知道当前的ViewGroup该设置为多大去容纳它们。
2. 根据子View的大小，以及我们的ViewGroup要实现的功能，决定出ViewGroup的大小
3. ViewGroup和子View的大小算出来了之后，接下来就是去摆放了吧，具体怎么去摆放呢？这得根据你定制的需求去摆放了，比如，你想让子View按照垂直顺序一个挨着一个放，或者是按照先后顺序一个叠一个去放，这是你自己决定的。
4. 已经知道怎么去摆放还不行啊，决定了怎么摆放就是相当于把已有的空间”分割”成大大小小的空间，每个空间对应一个子View，我们接下来就是把子View对号入座了，把它们放进它们该放的地方去。



#### 5.4.1 onLayout



### 5.5 SurfaceView

SurfaceView中采用了双缓冲机制，保证了UI界面的流畅性，同时SurfaceView不在主线程中绘制，而是另开辟一个线程去绘制，所以它不妨碍UI线程；

**SurfaceView继承于View，他和View主要有以下三点区别：**

(1)View底层没有双缓冲机制，SurfaceView有；

(2)view主要适用于主动更新，而SurfaceView适用与被动的更新，如频繁的刷新

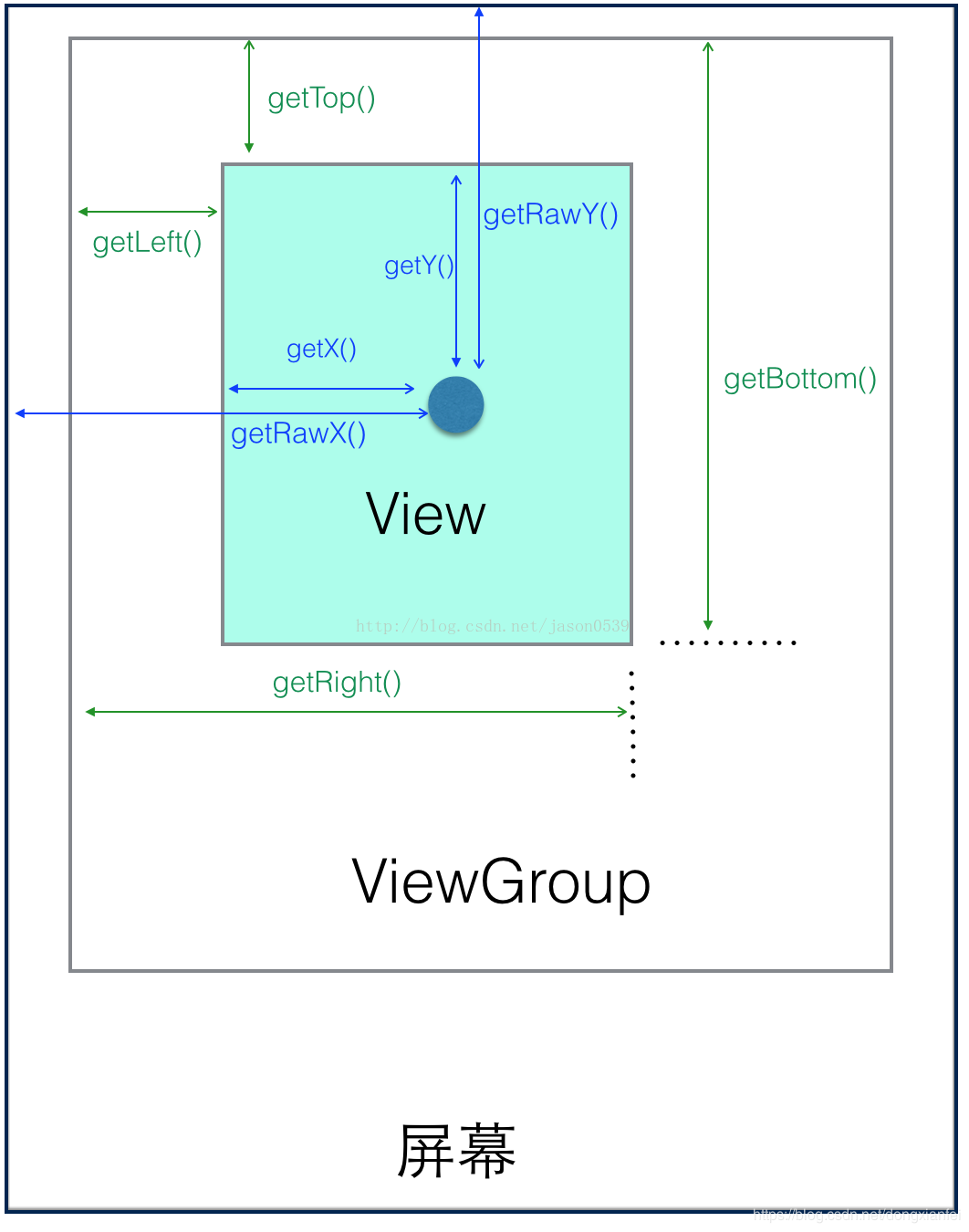
(3)view会在主线程中去更新UI，而SurfaceView则在子线程中刷新；SurfaceView的内容不在应用窗口上，所以不能使用变换（平移、缩放、旋转等）。也难以放在ListView或者ScrollView中，不能使用UI控件的一些特性比如View.setAlpha()

**View**：显示视图，内置画布，提供图形绘制函数、触屏事件、按键事件函数等；必须在UI主线程内更新画面，速度较慢。

**SurfaceView**：基于view视图进行拓展的视图类，更适合2D游戏的开发；是view的子类，类似使用双缓机制，在新的线程中更新画面所以刷新界面速度比view快，Camera预览界面使用SurfaceView。

**GLSurfaceView**：基于SurfaceView视图再次进行拓展的视图类，专用于3D游戏开发的视图；是SurfaceView的子类，openGL专用。

### 5.6 坐标系



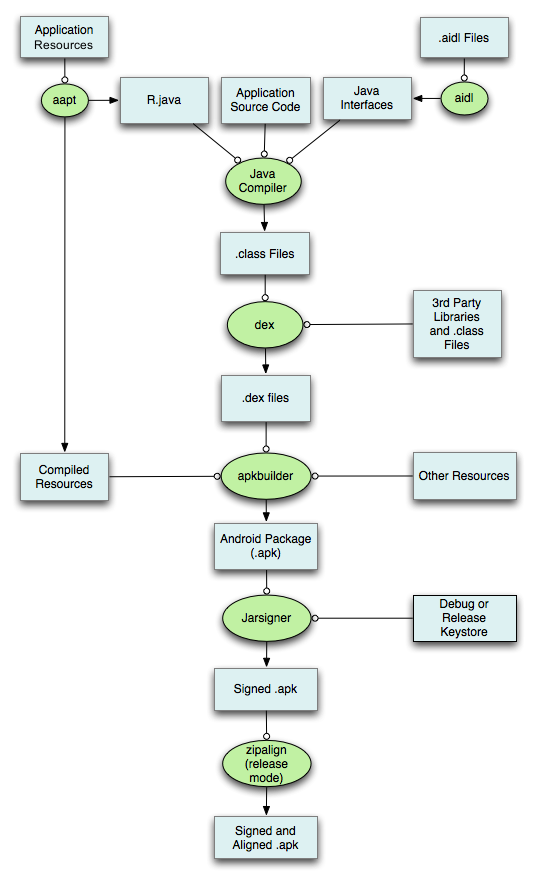
## 6 系统原理

### 6.1 打包原理

Android的包文件APK分为两个部分：代码和资源，所以打包方面也分为资源打包和代码打包两个方面，这 篇文章就来分析资源和代码的编译打包原理。

具体说来：

1. 通过AAPT工具进行资源文件（包括AndroidManifest.xml、布局文件、各种xml资源等）的打包，生成R.java文件。
2. 通过AIDL工具处理AIDL文件，生成相应的Java文件。
3. 通过Javac工具编译项目源码，生成Class文件。
4. 通过DX工具将所有的Class文件转换成DEX文件，该过程主要完成Java字节码转换成Dalvik字节码，压缩常量池以及清除冗余信息等工作。
5. 通过ApkBuilder工具将资源文件、DEX文件打包生成APK文件。
6. 利用KeyStore对生成的APK文件进行签名。
7. 如果是正式版的APK，还会利用ZipAlign工具进行对齐处理，对齐的过程就是将APK文件中所有的资源文件举例文件的起始距离都偏移4字节的整数倍，这样通过内存映射访问APK文件 的速度会更快。



#### 6.1.1 问题复现

1. **插件资源和host资源产生冲突解决方案**

**问题引出：**

每个资源都对应一个R中的16进制，由三部组成，packageId,TypeId,EntryId组成

packageId：apk包的id，默认为0x7f

TypeId:资源类型id，如layout，string，ID，drawable等

EntryId：类型TypeId下面的资源id，从0开始递增

**如：0x7f0b006d,**

packageId:0x7f

TypeId:0b

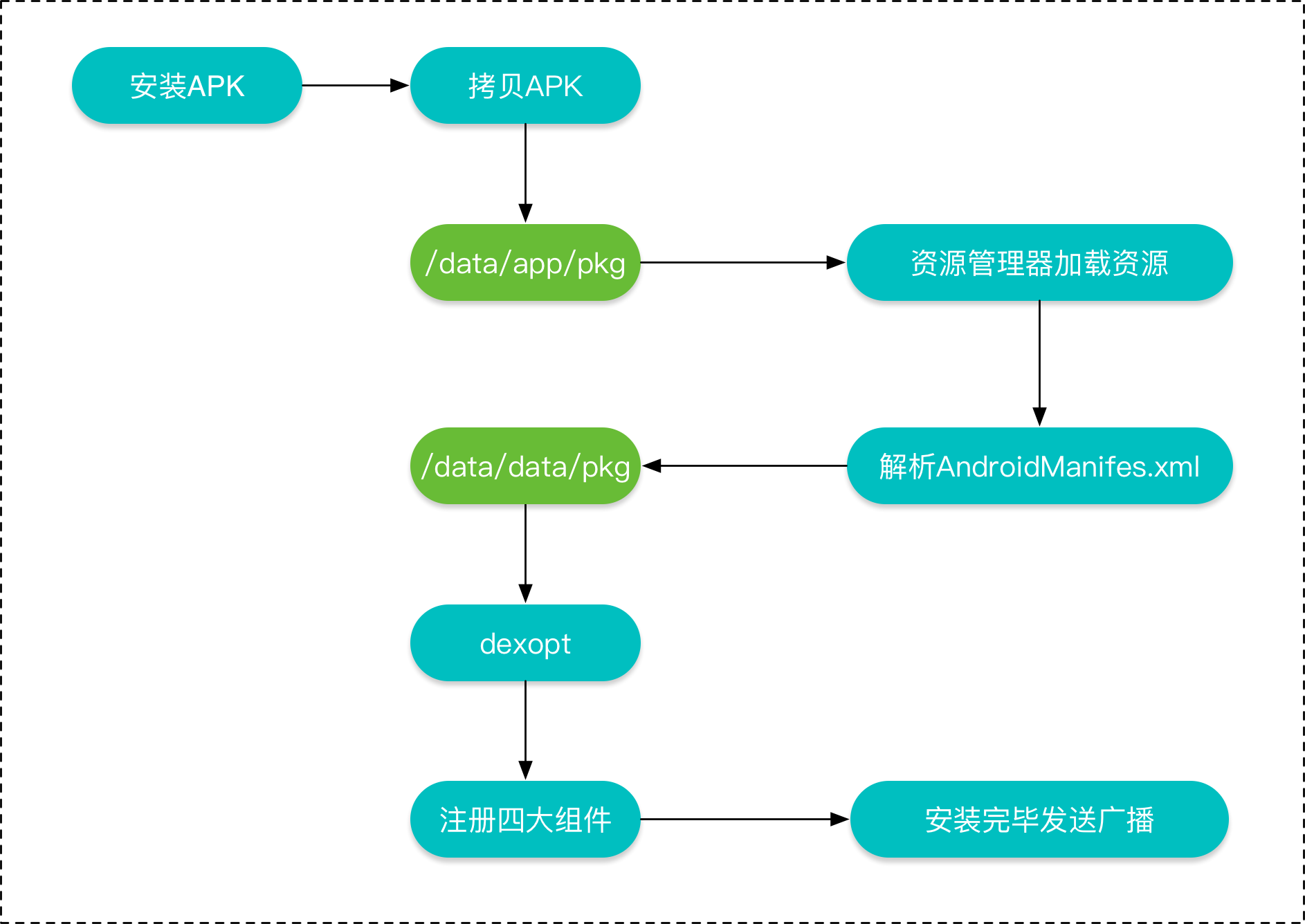
EntryId:006d

**解决方法：**

设置不同的packageId,它们是由aapt这个工具进行打包的，修改它的源码工具

### 6.2 安装流程

1. 复制APK到/data/app目录下，解压并扫描安装包。
2. 资源管理器解析APK里的资源文件。
3. 解析AndroidManifest文件，并在/data/data/目录下创建对应的应用数据目录。
4. 然后对dex文件进行优化，并保存在dalvik-cache目录下。
5. 将AndroidManifest文件解析出的四大组件信息注册到PackageManagerService中。
6. 安装完成后，发送广播。



### 6.3 混淆

**好处：**混淆将主项目及依赖库中未被使用的类，类成员，方法，属性移除，有助于规避64K方法数的瓶颈，会删除无用的资源，有效的减小apk包的大小，同时将类及其成员，方法重命名为无意义的简短名称，增加逆向工程的难度。

**混淆操作：**

1. 压缩（Shrinking）
2. 优化（Optimization）
3. 混淆（Obfuscation）
4. 预校验（Preverification）

#### 6.3.1 正常app混淆规则：

1. 四大组件和application不能混淆，需要在AndroidManifest中声明
2. R文件不能混淆，因为有时需要反射获取资源
3. support的v4和v7包中的类不能混淆，系统的东西不要动
4. 实现了Serializable和Parcelable的类不混淆，否则反序列化会出现问题
5. 泛型不能混淆
6. 注解不混淆，反射时会使用
7. 不能混淆枚举中的value和valueOf方法，会被反射使用
8. 自定义view不能混淆，否则在layout中使用的自定义view会找不到。

#### 6.3.2 插件app混淆规则

1. 遵守上述规则

#### 6.3.3 插件app混淆方案

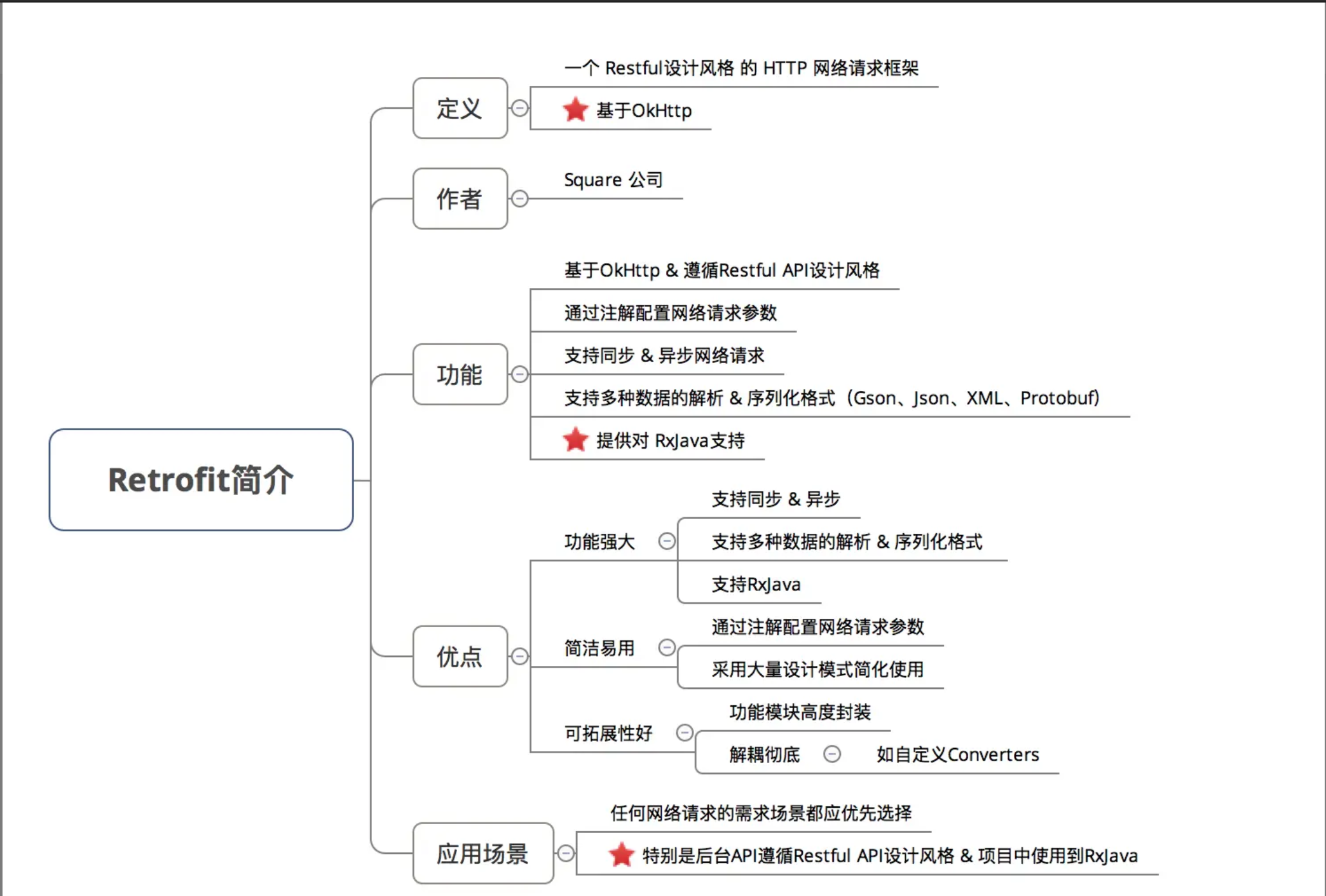
1. 不混淆公共库（provide引用方式）

## 第三方库解析

### 7.1 网络请求框架

#### 7.1.1 Retrofit

分析retrofit源码学习设计模式<https://www.jianshu.com/p/0c055ad46b6c>



**概念：**Retrofit是一个基于RESTful的HTTP网络请求框架的封装，其中网络请求的本质是由OKHttp完成的，而Retrofit仅仅负责网络请求接口的封装。

**原理：**App应用程序通过Retrofit请求网络，实际上是使用Retrofit接口层封装请求参数，Header、URL等信息，之后由OKHttp完成后续的请求，在服务器返回数据之后，OKHttp将原始的结果交给Retrofit，最后根据用户的需求对结果进行解析。

**请求过程：**

1. 通过解析网络请求接口的注解，配置网络请求参数
2. 通过动态代理生成网络请求对象
3. 通过网络请求适配器将网络请求对象进行平台（Android，RxJava，Guava，Java8）适配
4. 通过网络请求执行器发送网络请求
5. 通过数据转换器解析服务器返回的数据
6. 通过回调执行器切换线程（子线程到主线程）
7. 用户在主线程中返回结果

|  |  |  |  |
| --- | --- | --- | --- |
| **步骤** | **角色** | **流程** | **设计模式** |
| 1 | 接口-注解（BuildRequest） | 配置网络请求参数  （通过注解） | 建造者模式，（**Retrofit使用建造者模式通过Builder类创建一个实例**）  工厂方法模式  外观模式，代理模式（**Retrofit是通过外观和代理模式使用create方法创建网络请求接口的实例，同时通过网络接口请求里设置的注解进行了网络请求参数的配置**）  单例模式()  策略模式  装饰器模式 |
| 2 | 网络请求执行器  （Call） | 创建网络请求对象 |
| 3 | 网络请求适配器  （CallAdapter） | 适配到具体的call | 适配器模式 |
| 4 | 网络请求执行器  （Call） | 发送网络请求 | 代理模式 |
| 5 | 数据转换器  （Converter） | 服务器返回结果  解析数据 |  |
| 6 | 回调执行器  （Executor） | 切换线程  （子线程到主线程） | 适配器模式，装饰模式  **在Android extends Platform这个类中，有个MainThreadExecutor方法** |
| 7 |  | 处理返回的数据 |  |

总结：

Retrofit采用了外观模式统一调用创建网络请求接口实例和网络请求参数配置的方法：

1. 动态创建网络请求接口的实例（动态代理模式）
2. 创建ServiceMethod对象（建造者模式&单例模式（缓存机制））

（3）对ServiceMethod对象进行网络请求参数配置，通过解析网络请求接口的参数，返回值和注解类型，从从Retrofit对象中获取对应的网络请求的url地址、网络请求执行器、网络请求适配器 & 数据转换器。**（策略模式）**

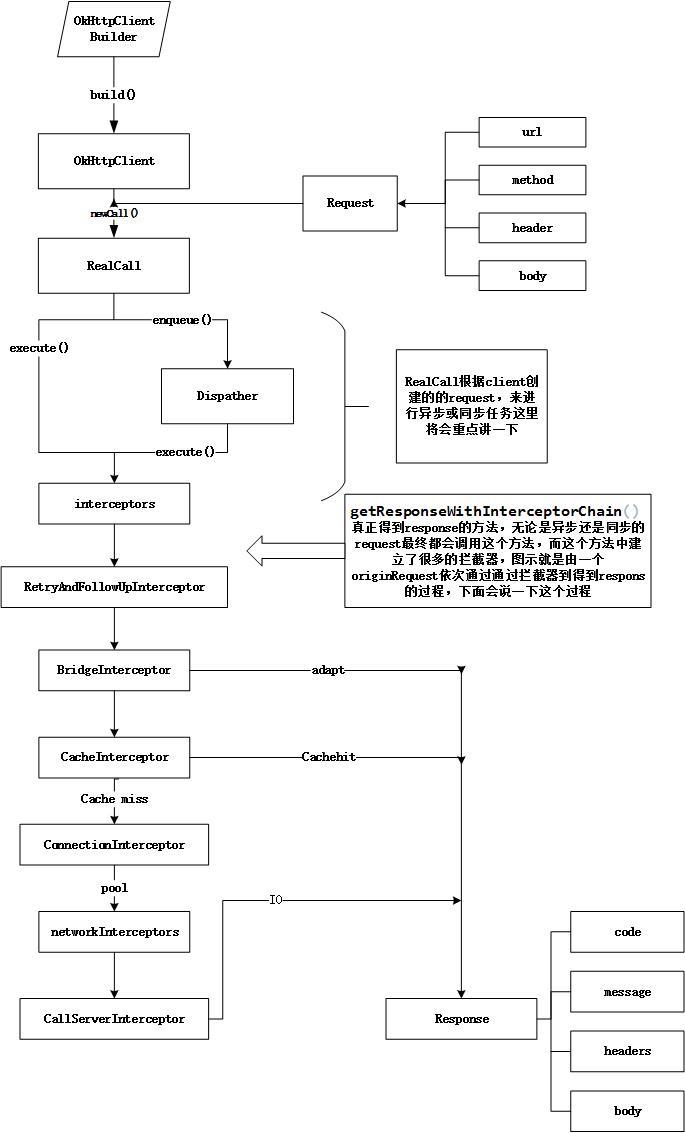
（4）对 serviceMethod 对象加入线程切换的操作，便于接收数据后通过Handler从子线程切换到主线程从而对返回数据结果进行处理**（装饰模式）**

（5）最终创建并返回一个OkHttpCall类型的网络请求对象



#### 7.1.2 Okhttp

<https://www.jianshu.com/p/196f7b2a703c>



**OKHttp优点**

支持HTTP2/SPDY（SPDY是Google开发的基于TCP的传输层协议，用以最小化网络延迟，提升网络速度，优化用户的网络使用体验。）

socket自动选择最好路线，并支持自动重连，拥有自动维护的socket连接池，减少握手次数，减少了请求延迟，共享Socket,减少对服务器的请求次数。

基于Headers的缓存策略减少重复的网络请求。

拥有Interceptors轻松处理请求与响应（自动处理GZip压缩）。

**keepalive connections**

当然大量的连接每次连接关闭都要三次握手四次分手的很显然会造成性能低下，因此http有一种叫做keepalive connections的机制，它可以在传输数据后仍然保持连接，当客户端需要再次获取数据时，直接使用刚刚空闲下来的连接而不需要再次握手。

Okhttp支持5个并发KeepAlive，默认链路生命为5分钟(链路空闲后，保持存活的时间)。

**缓存策略**

1、如果网络不可用并且无可用的有效缓存，则返回504错误；

2、继续，如果不需要网络请求，则直接使用缓存；

3、继续，如果需要网络可用，则进行网络请求；

4、继续，如果有缓存，并且网络请求返回HTTP\_NOT\_MODIFIED，说明缓存还是有效的，则合并网络响应和缓存结果。同时更新缓存；

5、继续，如果没有缓存，则写入新的缓存；

**五种拦截器**

看到这里，不禁会问，上面看到的那么多种拦截器到底分别是用来干啥的呢，在这里来总结一下（来自网络，只是做个归纳）：

**RetryAndFollowUpInterceptor**

用来实现连接失败的重试和重定向

**BridgeInterceptor**

用来修改请求和响应的 header 信息

**CacheInterceptor**

用来实现响应缓存。比如获取到的 Response 带有 Date，Expires，Last-Modified，Etag 等 header，表示该 Response 可以缓存一定的时间，下次请求就可以不需要发往服务端，直接拿缓存的

**ConnectInterceptor**

用来打开到服务端的连接。其实是调用了 StreamAllocation 的newStream 方法来打开连接的。建联的 TCP 握手，TLS 握手都发生该阶段。过了这个阶段，和服务端的 socket 连接打通

**CallServerInterceptor**

用来发起请求并且得到响应。上一个阶段已经握手成功，HttpStream 流已经打开，所以这个阶段把 Request 的请求信息传入流中，并且从流中读取数据封装成 Response 返回

**总结：**

OkHttp的底层是通过Java的Socket发送HTTP请求与接受响应的(这也好理解，HTTP就是基于TCP协议的)，但是OkHttp实现了连接池的概念，即对于同一主机的多个请求，其实可以公用一个Socket连接，而不是每次发送完HTTP请求就关闭底层的Socket，这样就实现了连接池的概念。而OkHttp对Socket的读写操作使用的OkIo库进行了一层封装。

### 7.2 图片加载库对比

Picasso：120K

Glide：475K

Fresco：3.4M

Android-Universal-Image-Loader：162K

图片函数库的选择需要根据APP的具体情况而定，对于严重依赖图片缓存的APP，例如壁纸类，图片社交类APP来说，可以选择最专业的Fresco。对于一般的APP，选择Fresco会显得比较重，毕竟Fresco3.4M的体量摆在这。根据APP对图片的显示和缓存的需求从低到高，我们可以对以上函数库做一个排序。

Picasso < Android-Universal-Image-Loader < Glide < Fresco

#### 7.2.1 介绍:

Picasso ：和Square的网络库一起能发挥最大作用，因为Picasso可以选择将网络请求的缓存部分交给了okhttp实现。

Glide：模仿了Picasso的API，而且在他的基础上加了很多的扩展(比如gif等支持)，Glide默认的Bitmap格式是RGB\_565，比 Picasso默认的ARGB\_8888格式的内存开销要小一半；Picasso缓存的是全尺寸的(只缓存一种)，而Glide缓存的是跟ImageView尺寸相同的(即56\*56和128\*128是两个缓存) 。

FB的图片加载框架Fresco：最大的优势在于5.0以下(最低2.3)的bitmap加载。在5.0以下系统，Fresco将图片放到一个特别的内存区域(Ashmem区)。当然，在图片不显示的时候，占用的内存会自动被释放。这会使得APP更加流畅，减少因图片内存占用而引发的OOM。为什么说是5.0以下，因为在5.0以后系统默认就是存储在Ashmem区了。

#### 7.2.3 总结：

Picasso所能实现的功能，Glide都能做，无非是所需的设置不同。但是Picasso体积比起Glide小太多如果项目中网络请求本身用的就是okhttp或者retrofit(本质还是okhttp)，那么建议用Picasso，体积会小很多(Square全家桶的干活)。Glide的好处是大型的图片流，比如gif、Video，如果你们是做美拍、爱拍这种视频类应用，建议使用。

Fresco在5.0以下的内存优化非常好，代价就是体积也非常的大，按体积算

Fresco>Glide>Picasso

不过在使用起来也有些不便（小建议：他只能用内置的一个ImageView来实现这些功能，用起来比较麻烦，我们通常是根据Fresco自己改改，直接使用他的Bitmap层）

### 7.3 各种json解析库使用

参考链接：<https://www.cnblogs.com/kunpengit/p/4001680.html>

#### 7.3.1 Google的Gson

Gson是目前功能最全的Json解析神器，Gson当初是为因应Google公司内部需求而由Google自行研发而来，但自从在2008年五月公开发布第一版后已被许多公司或用户应用。Gson的应用主要为toJson与fromJson两个转换函数，无依赖，不需要例外额外的jar，能够直接跑在JDK上。而在使用这种对象转换之前需先创建好对象的类型以及其成员才能成功的将JSON字符串成功转换成相对应的对象。类里面只要有get和set方法，Gson完全可以将复杂类型的json到bean或bean到json的转换，是JSON解析的神器。Gson在功能上面无可挑剔，但是性能上面比FastJson有所差距。

#### 7.3.2 阿里巴巴的FastJson

Fastjson是一个Java语言编写的高性能的JSON处理器,由阿里巴巴公司开发。

无依赖，不需要例外额外的jar，能够直接跑在JDK上。FastJson在复杂类型的Bean转换Json上会出现一些问题，可能会出现引用的类型，导致Json转换出错，需要制定引用。FastJson采用独创的算法，将parse的速度提升到极致，超过所有json库。

**综上Json技术的比较，在项目选型的时候可以使用Google的Gson和阿里巴巴的FastJson两种并行使用，如果只是功能要求，没有性能要求，可以使用google的Gson，如果有性能上面的要求可以使用Gson将bean转换json确保数据的正确，使用FastJson将Json转换Bean**

## 8 热点技术

参考链接- [Android组件化方案](https://blog.csdn.net/guiying712/article/details/55213884)

### 8.1 组件化

#### 8.1.2 概念：

**组件化：**是将一个APP分成多个module，每个module都是一个组件，也可以是一个基础库供组件依赖，开发中可以单独调试部分组件，组件中不需要相互依赖但是可以相互调用，最终发布的时候所有组件以lib的形式被主APP工程依赖打包成一个apk。

#### 8.1.3 由来：

1. APP版本迭代，新功能不断增加，业务变得复杂，维护成本高
2. 业务耦合度高，代码臃肿，团队内部多人协作开发困难
3. Android编译代码卡顿，单一工程下代码耦合严重，修改一处需要重新编译打包，耗时耗力。
4. 方便单元测试，单独改一个业务模块，不需要着重关注其他模块。

#### 8.1.4 优势：

1. 组件化将通用模块独立出来，统一管理，以提高复用，将页面拆分为粒度更小的组件，组件内部出了包含UI实现，还可以包含数据层和逻辑层
2. 每个组件度可以独立编译、加快编译速度、独立打包。
3. 每个工程内部的修改，不会影响其他工程。
4. 业务库工程可以快速拆分出来，集成到其他App中。
5. 迭代频繁的业务模块采用组件方式，业务线研发可以互不干扰、提升协作效率，并控制产品质量，加强稳定性。
6. 并行开发，团队成员只关注自己的开发的小模块，降低耦合性，后期维护方便等。

#### 8.1.5 考虑问题：

**模式切换：如何使得APP在单独调试跟整体调试自由切换**

组件化后的每一个业务的module都可以是一个单独的APP（isModuleRun=false）， release 包的时候各个业务module作为lib依赖，这里完全由一个变量控制，在根项目 gradle.properties里面isModuleRun=true。isModuleRun状态不同，加载application和AndroidManifest都不一样，以此来区分是独立的APK还是lib。

**在build.grade里面配置：**



**资源冲突：**当我们创建了多个Module的时候，如何解决相同资源文件名合并的冲突

业务Module和BaseModule资源文件名称重复会产生冲突，解决方案在

每个 module 都有 app\_name，为了不让资源名重名，在每个组件的 build.gradle 中增加 resourcePrefix “xxx\_强行检查资源名称前缀。固定每个组件的资源前缀。但是 resourcePrefix 这个值只能限定 xml 里面的资源，并不能限定图片资源。

**依赖关系：**多个Module之间如何引用一些共同的library以及工具类

**组件通信：**组件化之后，Module之间是相互隔离的，如何进行UI跳转以及方法调用

[阿里巴巴ARouter](https://github.com/alibaba/ARouter)

**[ARouter 核心实现思路是，我们在代码里加入的@Route 注解，会在编译时期通](https://github.com/alibaba/ARouter)**

**[过 apt 生成一些存储 path 和 activityClass 映射关系的类文件，然后 app 进程启](https://github.com/alibaba/ARouter)**

**[动的时候会拿到这些类文件，把保存这些映射关系的数据读到内存里(保存在](https://github.com/alibaba/ARouter)**

**[map 里)，然后在进行路由跳转的时候，通过 build()方法传入要到达页面的路由](https://github.com/alibaba/ARouter)**

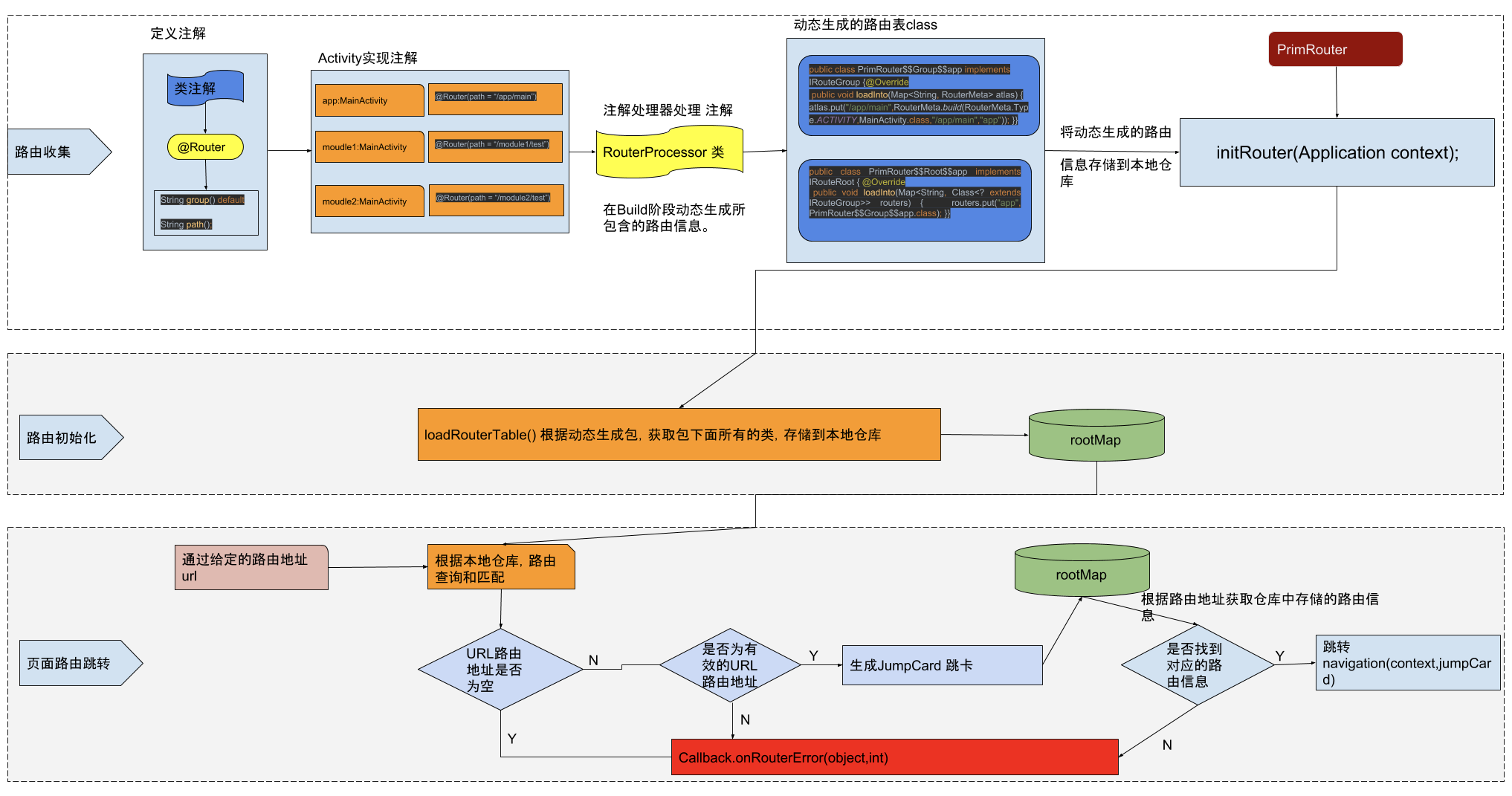
**[地址，ARouter 会通过它自己存储的路由表找到路由地址对应的](https://github.com/alibaba/ARouter)**

**[Activity.class(activity.class = map.get(path))，然后 new Intent()，当调用 ARouter](https://github.com/alibaba/ARouter)**

**[的 withString()方法它的内部会调用 intent.putExtra(String name, String value)，](https://github.com/alibaba/ARouter)**

**[调用 navigation()方法，它的内部会调用 startActivity(intent)进行跳转，这样便可](https://github.com/alibaba/ARouter)**

**[以实现两个相互没有依赖的 module 顺利的启动对方的 Activity 了。](https://github.com/alibaba/ARouter)**

[](https://github.com/alibaba/ARouter)

各业务Module之前不需要任何依赖可以通过路由跳转，完美解决业务之间耦合。

**入口参数：**我们知道组件之间是有联系的，所以在单独调试的时候如何拿到其它的Module传递过来的参数

**Application：**

当组件单独运行的时候，每个Module自成一个APK，那么就意味着会有多个Application，很显然我们不愿意重复写这么多代码，所以我们只需要定义一个BaseApplication即可，其它的Application直接继承此BaseApplication就OK了，BaseApplication里面还可定义公用的参数。

**[得到APP组件化](https://www.jianshu.com/p/1b1d77f58e84)**

### 8.2 插件化

**参考链接-** [插件化入门](https://www.jianshu.com/p/b6d0586aab9f)

<https://blog.csdn.net/github_37130188/article/details/89762543>

#### 8.2.1概述

提到插件化，就不得不提起方法数超过65535的问题，我们可以通过Dex分包来解决，同时也可以通过使用插件化开发来解决。插件化的概念就是由宿主APP去加载以及运行插件APP。

#### 8.2.2优点

1. 在一个大的项目里面，为了明确的分工，往往不同的团队负责不同的插件APP，这样分工更加明确。各个模块封装成不同的插件APK，不同模块可以单独编译，提高了开发效率。
2. 解决了上述的方法数超过限制的问题。
3. 可以通过上线新的插件来解决线上的BUG，达到“热修复”的效果。
4. 减小了宿主APK的体积。

#### 8.2.3缺点

插件化开发的APP不能在Google Play上线，也就是没有海外市场。

#### 8.2.4 插件化总结

HostApp：壳app

PluginApp：插件app

1. **插件中类的加载**

HostApp想要加载Pluginapp中的类，使用HostApp中的classloader是不行的，解决方案有以下三种：

1. 在反射插件中的类时可以使用插件的classloader
2. 宿主和插件它们各自的classloader都对应一个dex数组，把这些dex数组都合并到宿主的dex数组中，那么宿主app就可以通过反射加载任何插件中的类，
3. 自定义一个classloader，取代原先的宿主classloader，同时在自定义的classloader中放一个集合，承载所有插件的classloader，那么这个自定义classloader在加载任何一个类时都会在宿主中查找，如果没有的话再遍历内部的classloader集合，看哪个插件的classloader可以加载这个类。

**（2）哪些地方可以Hook**

a. app中可以使用的类，可以Hook，系统源码中标记了hide的类和方法不可以hook，但是可以通过反射去调用它们。但是在app中可以使用的类可以Hook，如Instrumentation和Callback

b. 实现了接口的类，可以通过动态代理的方式

1. **Activity的插件化解决方案**

Activity插件化解决方案主要有动态替换和静态代理两种。

1. 动态替换。占位思想，在HostApp中申明一个StubActivity，启动插件的ActivityA，但是告诉AMS启动的是StubActivity，欺骗成功后在即将启动Activity时再将StubActivity改为ActivityA，因此需要Hook一些系统方法。
2. 静态代理。在HostApp中设计一个ProxyActivity，插件中的Activity都是没有生命周期的，在ProxyActivity的生命周期中，调用插件Activity相应的生命周期函数。
3. **资源的插件化解决方案**

app是通过AssetManager来加载资源的，它通过addAssetPath方法加载指定位置的资源，默认是加载App自身的资源。从HostApp跳转到plugin只能夹杂HostApp下面的资源而不能加载插件的资源，解决方案如下：

1. 将HostApp和插件的资源都通过AssetManager的addAssetPath方法添加到一起，在插件Activity的基类中重写getResource方法。不过会产生资源ID冲突的问题：解决方法有：
2. 修改aapt，为每个插件的id指定不能的前缀，默认的0x7f，可以改为0x71等
3. 修改resource.arsc，在aapt执行后生成了R.java和resource.arsc ，可以把插件中R.java中的所有资源前缀都改为0x71，把resource.arsc中的0x7f也改为0x71即可。
4. **Service 的插件化解决方案**
5. 动态代理，事先在HostApp中的AndroidManifest.xml中申明多个StubService，分别对应不同的插件Service
6. 静态代理，创建一个ProxyService，有ProxyService来启动插件的Service，缺点是插件中有几个Service，HostApp中AndroidManifest.xml也要申明相同数量的ProxyService
7. **BroadCastReceiver的插件化解决方案**

同上

### 8.3 热修复

（1）PathClassLoader：只能加载已经安装到Android系统中的apk文件（/data/app目录），是Android默认使用的类加载器。

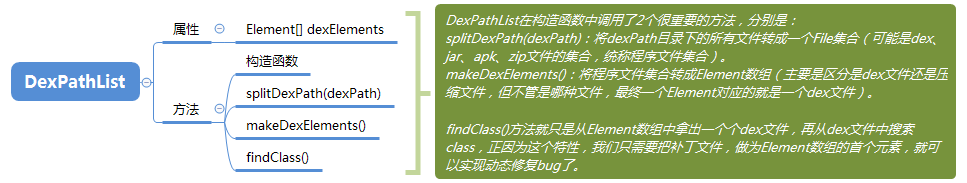
（2）DexClassLoader：可以加载任意目录下的dex/jar/apk/zip文件，比PathClassLoader更灵活，是实现热修复的重点。



类加载器肯定会提供有一个方法来供外界找到它所加载到的class，该方法就是findClass()，不过在PathClassLoader和DexClassLoader源码中都没有重写父类的findClass()方法，但它们的父类BaseDexClassLoader就有重写findClass()

BaseDexClassLoader的findClass()方法实际上是通过DexPathList对象（pathList）的findClass()方法来获取class的，而这个DexPathList对象恰好在之前的BaseDexClassLoader构造函数中就已经被创建好了。

DexPathList的构造函数是将一个个的程序文件（可能是dex、apk、jar、zip）封装成一个个Element对象，最后添加到Element集合中。



**热修复的实现原理：**

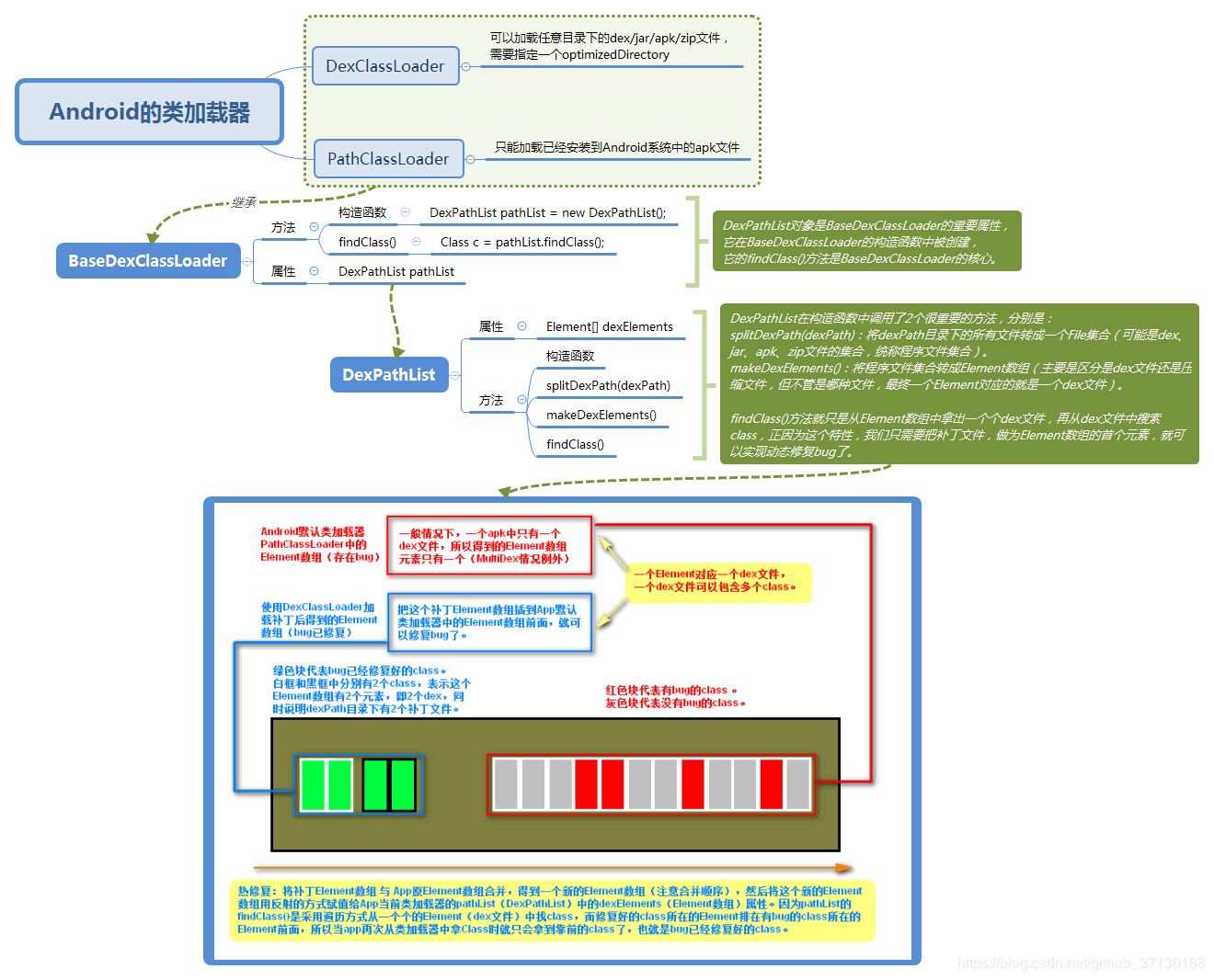
经过对PathClassLoader、DexClassLoader、BaseDexClassLoader、DexPathList的分析，我们知道，安卓的类加载器在加载一个类时会先从自身DexPathList对象中的Element数组中获取（Element[] dexElements）到对应的类，之后再加载。采用的是数组遍历的方式，不过注意，遍历出来的是一个个的dex文件。

在for循环中，首先遍历出来的是dex文件，然后再是从dex文件中获取class，所以，我们只要让修复好的class打包成一个dex文件，放于Element数组的第一个元素，这样就能保证获取到的class是最新修复好的class了（当然，有bug的class也是存在的，不过是放在了Element数组的最后一个元素中，所以没有机会被拿到而已）。

利用PathClassLoader和DexClassLoader去加载与bug类同名的类，替换掉bug类，进而达到修复bug的目的，原理是在app打包的时候阻止类打上**[CLASS\_ISPREVERIFIED](https://www.2cto.com/kf/201604/498017.html)**标志，然后在热修复的时候动态改变BaseDexClassLoader对象间接引用的dexElements，替换掉旧的类。



**总结：**



### 8.4 多渠道打包

多渠道打包关注两件事：

1. 将渠道信息写入apk中，
2. 将apk中的渠道信息传输到后台

#### 8.4.1 签名方式

1. Android7.0之前使用v1签名方式，是jar signature，源自于JDK。7.0和7.0以下的版本安装没有问题
2. Android7.0之后使用v2签名方式，是Android独有的apk signature。7.0以下的版本安装失败

## 9 屏幕适配

### 9.1基本概念

**屏幕尺寸**

含义：手机对角线的物理尺寸 单位：英寸（inch），1英寸=2.54cm

Android手机常见的尺寸有5寸、5.5寸、6寸等等

**屏幕分辨率**

含义：手机在横向、纵向上的像素点数总和

一般描述成屏幕的”宽x高”=AxB 含义：屏幕在横向方向（宽度）上有A个像素点，在纵向方向

（高）有B个像素点 例子：1080x1920，即宽度方向上有1080个像素点，在高度方向上有1920个像素点

单位：px（pixel），1px=1像素点

UI设计师的设计图会以px作为统一的计量单位

Android手机常见的分辨率：320x480、480x800、720x1280、1080x1920

**屏幕像素密度**

含义：每英寸的像素点数 单位：dpi（dots per ich）

假设设备内每英寸有160个像素，那么该设备的屏幕像素密度=160dpi

### 9.2 适配方法

1.支持各种屏幕尺寸： 使用wrap\_content, match\_parent, weight.要确保布局的灵活性并适应各种尺寸的屏幕，应使用 “wrap\_content”、“match\_parent” 控制某些视图组件的宽度和高度。

2.使用相对布局，禁用绝对布局。

**3.使用LinearLayout的weight属性**

假如我们的宽度不是0dp(**wrap\_content和0dp的效果相同**)，则是match\_parent呢？

android:layout\_weight的真实含义是:如果View设置了该属性并且有效，那么该 View的宽度等于原有宽度(android:layout\_width)加上剩余空间的占比。

从这个角度我们来解释一下上面的现象。在上面的代码中，我们设置每个Button的宽度都是match\_parent，假设屏幕宽度为L，那么每个Button的宽度也应该都为L，剩余宽度就等于L-（L+L）= -L。

Button1的weight=1，剩余宽度占比为1/(1+2)= 1/3，所以最终宽度为L+1/3\*(-L)=2/3L，Button2的计算类似，最终宽度为L+2/3(-L)=1/3L。

**4.使用.9图片**

## 10 性能优化

参考链接：[Android 性能监测工具，优化内存、卡顿、耗电、APK大小的方法](https://blog.csdn.net/csdn_aiyang/article/details/74989318)

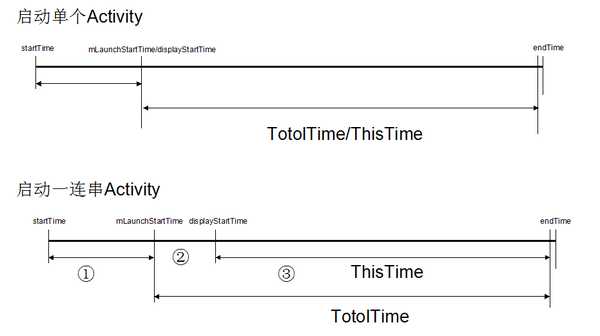
**参考博客：<http://mtancode.com/>**

1. 稳定（内存溢出、崩溃）
2. 流畅（卡顿）
3. 耗损（耗电、流量）
4. 安装包（APK瘦身）

影响稳定性的原因很多，比如内存使用不合理、代码异常场景考虑不周全、代码逻辑不合理等，都会对应用的稳定性造成影响。其中最常见的两个场景是：Crash 和 ANR，这两个错误将会使得程序无法使用。所以做好Crash全局监控，处理闪退同时把崩溃信息、异常信息收集记录起来，以便后续分析;合理使用主线程处理业务，不要在主线程中做耗时操作，防止ANR程序无响应发生。

### 10.0 启动app黑白屏优化

app启动时间计算：**adb shell am start -w packagename/activity,**



WaitTime 就是总的耗时，包括前一个应用 Activity pause 的时间和新应用启动的时间；

ThisTime 表示一连串启动 Activity 的最后一个 Activity 的启动耗时；

TotalTime 表示新应用启动的耗时，包括新进程的启动和 Activity 的启动，但不包括前一个应用 Activity pause 的耗时。也就是说，开发者一般只要关心TotalTime 即可，这个时间才是自己应用真正启动的耗时。

1. **窗口优化**

**Application的启动优化**

**Application#attachBaseContext()**

Application启动会经过attachBaseContext()–>onCreate()；这时大家从attachBaseContext的生命周期联想到什么？没错就是MultiDex分包机制。想必大家都会发现，自从我们方法数超出了65535处理了分包之后，启动白屏/黑屏的问题就出现了，分包机制是导致冷启动缓慢的重要原因，而现在部分应用采用插件化的方式来避免MultiDex带来的白屏问题，这虽然是一种方法，但是开发成本实在高，对于不少应用来说是不必要的。我们来聊一下MultiDex优化，首先MultiDex可分成运行时和编译时两个部分：

编译期：将App中的class以某种策略拆分在多个dex中，为了减少第一个dex也就主dex中包含的class数；

运行期： App启动时，虚拟机只加载主dex中的class。app启动以后，使用Multidex.install，通过反射机制修改ClassLoader中的dexElements来加载其他dex；

从网上的多篇实践分析中，他们主要采用的是异步方式。因为App起始会先加载主dex包，那么我们可以自主去处理分包的工作，我们将启动页和首页需要的库、组件等主要class分在主dex中，从而达到精分主dex包的大小，具体的操作写法，大家可以参考网上MultiDex启动优化文章，但是大家要注意在主dex的分包过程中，主dex经过我们一系列的优化操作减少了主dex的大小，因此也增大了NoClassDefFoundError的异常的可能，此时会导致我们的应用启动失败的风险，所以在优化后我们一定做好测试工作。

### 10.1稳定——内存优化

**内存抖动**

**内存抖动（代码注意事项）：**

内存抖动是由于短时间内有大量对象进出新生区导致的，它伴随着频繁的GC，gc会大量占用ui线程和cpu资源，会导致app整体卡顿。

**避免发生内存抖动的几点建议：**

（1）尽量避免在循环体内创建对象，应该把对象创建移到循环体外。

（2）注意自定义View的onDraw()方法会被频繁调用，所以在这里面不应该频繁的创建对象。

（3）当需要大量使用Bitmap的时候，试着把它们缓存在数组或容器中实现复用。

（4）对于能够复用的对象，同理可以使用对象池将它们缓存起来。

**（1）Memory Monitor 工具：**

它是Android Studio自带的一个内存监视工具，它可以很好地帮助我们进行内存实时分析。通过点击Android Studio右下角的Memory Monitor标签，打开工具可以看见较浅蓝色代表free的内存，而深色的部分代表使用的内存从内存变换的走势图变换，可以判断关于内存的使用状态，例如当内存持续增高时，可能发生内存泄漏；当内存突然减少时，可能发生GC等，如下图所示。

**（2）LeakCanary工具：**

这个工具是Square公司在Github开源的。

首先，笔者仔细查看了Leakcanary官方的github仓库，最重要的便是对**Leakcanary是如何起作用的**（即原理）这一问题进行了阐述，我自己把它翻译成了易于理解的文字，主要分为如下7个步骤：

* 1、RefWatcher.watch()创建了一个KeyedWeakReference用于去观察对象。
* 2、然后，在后台线程中，它会检测引用是否被清除了，并且是否没有触发GC。
* 3、如果引用仍然没有被清除，那么它将会把堆栈信息保存在文件系统中的.hprof文件里。
* 4、HeapAnalyzerService被开启在一个独立的进程中，并且HeapAnalyzer使用了HAHA开源库解析了指定时刻的堆栈快照文件heap dump。
* 5、从heap dump中，HeapAnalyzer根据一个独特的引用key找到了KeyedWeakReference，并且定位了泄露的引用。
* 6、HeapAnalyzer为了确定是否有泄露，计算了到GC Roots的最短强引用路径，然后建立了导致泄露的链式引用。
* 7、这个结果被传回到app进程中的DisplayLeakService，然后一个泄露通知便展现出来了。

官方的原理简单来解释就是这样的：**在一个Activity执行完onDestroy()之后，将它放入WeakReference中，然后将这个WeakReference类型的Activity对象与ReferenceQueque关联。这时再从ReferenceQueque中查看是否有没有该对象，如果没有，执行gc，再次查看，还是没有的话则判断发生内存泄露了。最后用HAHA这个开源库去分析dump之后的heap内存。**

**（3）Android Lint 工具：**

### 10.2 流畅——卡顿优化

View的绘制帧数保持60fps最佳，这要求没帧绘制时间不超过16ms，如果不能在16ms内完成界面的渲染，那么就会出现卡顿的现象。

· UI线程中做了耗时操作，导致UI线程卡顿

· 布局层次嵌套过多，过于复杂，无法在16ms内完成渲染

· 同一时间动画执行的次数过多，导致CPU和GPU负载过重

· overDraw，导致像素在同一帧的时间内被绘制多次

· view频繁的触发measure、layout，导致measure、layout类似耗时过多和整个view频繁重新渲染

· 频繁触发GC，使得16ms无法完成绘制

· ANR

**（1）布局优化**

在Android种系统对View进行测量、布局和绘制时，都是通过对View数的遍历来进行操作的。如果一个View数的高度太高就会严重影响测量、布局和绘制的速度。Google也在其API文档中建议View高度不宜哦过10层。现在版本种Google使用RelativeLayout替代LineraLayout作为默认根布局，目的就是降低LineraLayout嵌套产生布局树的高度，从而提高UI渲染的效率。

* 布局复用，使用<include>标签重用layout；
* 提高显示速度，使用<ViewStub>延迟View加载；
* 减少层级，使用<merge>标签替换父级布局；
* 注意使用wrap\_content，会增加measure计算成本；
* 删除控件中无用属性；

**（3）启动优化**

应用一般都有闪屏页SplashActivity，优化闪屏页的 UI 布局，可以通过 Profile GPU Rendering 检测丢帧情况。

1. **优化工具**

**CPU Profile**

### 10.3 节省——耗电优化

在 Android5.0 以前，关于应用电量消耗的测试即麻烦又不准确，而5.0 之后Google专门引入了一个获取设备上电量消耗信息的API—— Battery Historian。**Battery Historian** 是一款由 Google 提供的 Android 系统电量分析工具，直观地展示出手机的电量消耗过程，通过输入电量分析文件，显示消耗情况。

**最后提供一些可供参考耗电优化的方法：**

**（1）计算优化。算法、for循环优化、Switch..case替代if..else、避开浮点运算。**

浮点运算：计算机里整数和小数形式就是按普通格式进行存储，例如1024、3.1415926等等，这个没什么特点，但是这样的数精度不高，表达也不够全面，为了能够有一种数的通用表示法，就发明了浮点数。浮点数的表示形式有点像科学计数法（\*.\*\*\*\*\*×10^\*\*\*），它的表示形式是0.\*\*\*\*\*×10^\*\*\*，在计算机中的形式为 .\*\*\*\*\* e ±\*\*\*），其中前面的星号代表定点小数，也就是整数部分为0的纯小数，后面的指数部分是定点整数。利用这样的形式就能表示出任意一个整数和小数，例如1024就能表示成0.1024×10^4，也就是 .1024e+004，3.1415926就能表示成0.31415926×10^1，也就是 .31415926e+001，这就是浮点数。浮点数进行的运算就是浮点运算。浮点运算比常规运算更复杂，因此计算机进行浮点运算速度要比进行常规运算慢得多。

**（2）避免 Wake Lock 使用不当。**

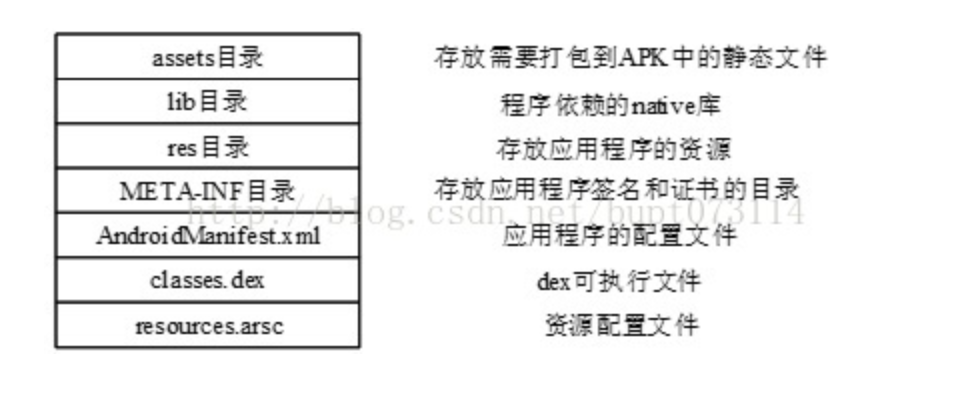
Wake Lock是一种锁的机制，主要是相对系统的休眠而言的，,只要有人拿着这个锁，系统就无法进入休眠意思就是我的程序给CPU加了这个锁那系统就不会休眠了，这样做的目的是为了全力配合我们程序的运行。有的情况如果不这么做就会出现一些问题，比如微信等及时通讯的心跳包会在熄屏不久后停止网络访问等问题。所以微信里面是有大量使用到了Wake\_Lock锁。系统为了节省电量，CPU在没有任务忙的时候就会自动进入休眠。有任务需要唤醒CPU高效执行的时候，就会给CPU加Wake\_Lock锁。大家经常犯的错误，我们很容易去唤醒CPU来工作，但是很容易忘记释放Wake\_Lock。

**（3）使用 Job Scheduler 管理后台任务。**

在Android 5.0 API 21 中，google提供了一个叫做JobScheduler API的组件，来处理当某个时间点或者当满足某个特定的条件时执行一个任务的场景，例如当用户在夜间休息时或设备接通电源适配器连接WiFi启动下载更新的任务。这样可以在减少资源消耗的同时提升应用的效率。

### 10.4 安装包——APK瘦身

#### （1）安装包的组成结构

****

**assets文件夹**。存放一些配置文件、资源文件，assets不会自动生成对应的 ID，而是通过 AssetManager 类的接口获取。

**res。**res 是 resource 的缩写，这个目录存放资源文件，会自动生成对应的 ID 并映射到 .R 文件中，访问直接使用资源 ID。

**META-INF。**保存应用的签名信息，签名信息可以验证 APK 文件的完整性。

**AndroidManifest.xml。**这个文件用来描述 Android 应用的配置信息，一些组件的注册信息、可使用权限等。

**classes.dex。**Dalvik 字节码程序，让 Dalvik 虚拟机可执行，一般情况下，Android 应用在打包时通过 Android SDK 中的 dx 工具将 Java 字节码转换为 Dalvik 字节码。

**resources.arsc。**记录着资源文件和资源 ID 之间的映射关系，用来根据资源 ID 寻找资源。

#### （2）减少安装包大小

* 代码混淆。使用IDE 自带的 proGuard 代码混淆器工具 ，它包括压缩、优化、混淆等功能。
* 资源优化。比如使用 Android Lint 删除冗余资源，资源文件最少化等。
* 图片优化。比如利用 PNG优化工具 对图片做压缩处理。推荐目前最先进的压缩工具Googlek开源库zopfli。如果应用在4.0版本以上，推荐使用 WebP图片格式。
* 避免重复或无用功能的第三方库。例如，百度地图接入基础地图即可、讯飞语音无需接入离线、图片库Glide\Picasso等。
* 插件化开发。比如功能模块放在服务器上，按需下载，可以减少安装包大小。
* 可以使用[微信开源资源文件混淆工具](https://github.com/shwenzhang/AndResGuard/blob/master/README.zh-cn.md" \t "_blank)——[AndResGuard](https://github.com/shwenzhang/AndResGuard" \t "_blank) 。一般可以压缩apk的1M左右大。

### 10.5 冷启动与热启动

**app冷启动**： 当应用启动时，后台没有该应用的进程，这时系统会重新创建一个新的进程分配给该应用， 这个启动方式就叫做冷启动（后台不存在该应用进程）。冷启动因为系统会重新创建一个新的进程分配给它，所以会先创建和初始化Application类，再创建和初始化MainActivity类（包括一系列的测量、布局、绘制），最后显示在界面上。

**app热启动**： 当应用已经被打开， 但是被按下返回键、Home键等按键时回到桌面或者是其他程序的时候，再重新打开该app时， 这个方式叫做热启动（后台已经存在该应用进程）。热启动因为会从已有的进程中来启动，所以热启动就不会走Application这步了，而是直接走MainActivity（包括一系列的测量、布局、绘制），所以热启动的过程只需要创建和初始化一个MainActivity就行了，而不必创建和初始化Application

**冷启动的流程**  
当点击app的启动图标时，安卓系统会从Zygote进程中fork创建出一个新的进程分配给该应用，之后会依次创建和初始化Application类、创建MainActivity类、加载主题样式Theme中的windowBackground等属性设置给MainActivity以及配置Activity层级上的一些属性、再inflate布局、当onCreate/onStart/onResume方法都走完了后最后才进行contentView的measure/layout/draw显示在界面上冷启动的生命周期简要流程：  
Application构造方法 –> attachBaseContext()–>onCreate –>Activity构造方法 –> onCreate() –> 配置主体中的背景等操作 –>onStart() –> onResume() –> 测量、布局、绘制显示

冷启动的优化主要是视觉上的优化，解决白屏问题，提高用户体验，所以通过上面冷启动的过程。能做的优化如下：

1、减少onCreate()方法的工作量  
2、不要让Application参与业务的操作  
3、不要在Application进行耗时操作  
4、不要以静态变量的方式在Application保存数据  
5、减少布局的复杂度和层级  
6、减少主线程耗时

### 10.6 内存泄漏的场景和解决办法

<https://blog.csdn.net/carson_ho/article/details/79407707>

#### 10.6.1非静态内部类的静态实例

非静态内部类会持有外部类的引用，如果非静态内部类的实例是静态的，就会长期的维持着外部类的引用，组织被系统回收，解决办法是使用静态内部类

#### 10.6.2多线程相关的匿名内部类和非静态内部类

匿名内部类同样会持有外部类的引用，如果在线程中执行耗时操作就有可能发生内存泄漏，导致外部类无法被回收，直到耗时任务结束，解决办法是在页面退出时结束线程中的任务

#### 10.6.3 Handler内存泄漏

Handler导致的内存泄漏也可以被归纳为非静态内部类导致的，Handler内部message是被存储在MessageQueue中的，有些message不能马上被处理，存在的时间会很长，导致handler无法被回收，如果handler是非静态的，就会导致它的外部类无法被回收，解决办法是1.使用静态handler，外部类引用使用弱引用处理2.在退出页面时移除消息队列中的消息

#### 10.6.4 Context导致内存泄漏

根据场景确定使用Activity的Context还是Application的Context,因为二者生命周期不同，对于不必须使用Activity的Context的场景（Dialog）,一律采用Application的Context,单例模式是最常见的发生此泄漏的场景，比如传入一个Activity的Context被静态类引用，导致无法回收

#### 10.6.5 静态View导致泄漏

使用静态View可以避免每次启动Activity都去读取并渲染View，但是静态View会持有Activity的引用，导致无法回收，解决办法是在Activity销毁的时候将静态View设置为null（View一旦被加载到界面中将会持有一个Context对象的引用，在这个例子中，这个context对象是我们的Activity，声明一个静态变量引用这个View，也就引用了activity）

#### 10.6.6 WebView导致的内存泄漏

WebView只要使用一次，内存就不会被释放，所以WebView都存在内存泄漏的问题，通常的解决办法是为WebView单开一个进程，使用AIDL进行通信，根据业务需求在合适的时机释放掉

#### 10.6.7 资源对象未关闭导致

如Cursor，File等，内部往往都使用了缓冲，会造成内存泄漏，一定要确保关闭它并将引用置为null

#### 10.6.8 集合中的对象未清理

集合用于保存对象，如果集合越来越大，不进行合理的清理，尤其是入股集合是静态的

#### 10.6.9 Bitmap导致内存泄漏

bitmap是比较占内存的，所以一定要在不使用的时候及时进行清理，避免静态变量持有大的bitmap对象

#### 10.6.10 监听器未关闭

很多需要register和unregister的系统服务要在合适的时候进行unregister,手动添加的listener也需要及时移除

### 10.7 Bitmap优化

<https://www.cnblogs.com/dasusu/p/9789389.html>

（1）要选择合适的图片规格（bitmap 类型）

通常我们优化 Bitmap 时，当需要做性能优化或者防止 OOM，我们通常会使用 RGB\_565，因为 ALPHA\_8 只有透明度，显示一般图片没有意义，Bitmap.Config.ARGB\_4444 显示图片不清楚， Bitmap.Config.ARGB\_8888 占用内存最多。：

 ALPHA\_8 每个像素占用 1byte 内存

 ARGB\_4444 每个像素占用 2byte 内存

 ARGB\_8888 每个像素占用 4byte 内存（默认）

 RGB\_565 每个像素占用 2byte 内存

（2）**降低采样率**

BitmapFactory.Options 参数 inSampleSize 的使用，先把 options.inJustDecodeBounds 设为 true，只是去读取图片的大小，在拿到图片的大小之后和要显示的大小做比较通过 calculateInSampleSize()函数计算 inSampleSize 的具体值，得到值之后。options.inJustDecodeBounds 设为 false读图片资源。

**（3）复用内存**：

即通过软引用(内存不够的时候才会回收掉)，复用内

存块，不需要再重新给这个 bitmap 申请一块新的内存，避免了

一次内存的分配和回收，从而改善了运行效率。

**（4）使用 recycle()方法及时回收内存**。

**（5）压缩图片**。

**Bitmap.recycle()会立即回收么？什么时候会回收？如果没有地方使用这**

**个 Bitmap，为什么垃圾回收不会直接回收？**

通过源码可以了解到，加载 Bitmap 到内存里以后，是包含**两部分内存区域**的。简单的说，一部分是 **Java 部分**的，一部分是 **C部分**的。这个 Bitmap 对象是由 Java 部分分配的，不用的时候系统就会自动回收了但是那个对应的 **C 可用**的内存区域，虚拟机是不能直接回收的，这个只能调用底层的功能释放。所以需要调用 recycle()方

法来释放 C 部分的内存bitmap.recycle()方法用于回收该 Bitmap 所占用的内存，接着将 bitmap 置空，最后使用 System.gc()调用一下系统的垃圾回收器进行回收，调用 System.gc()并不能保证立即开始进行回收过程，而只是为了加快回收的到来。

### 10.8 LRU 的原理

为减少流量消耗，可采用缓存策略。常用的缓存算法是LRU(Least Recently Used)：当缓存满时, 会优先淘汰那些近期最少使用的缓存对象。主要是两种方式：

**（1） LruCache(内存缓存)：**

LruCache 类是一个线程安全的泛型类：内部采用一个 LinkedHashMap 以强引用的方式存储外界的缓存对象，并提供 get 和 put 方法来完成缓存的获取和添加操作，当缓存满时会移除较早使用的缓存对象，再添加新的缓存对象。

（2）DiskLruCache(磁盘缓存)：

通过将缓存对象写入文件系统从

而实现缓存效果

### 10.9 webview优化

加载一个webview经历的过程如下：

1. webview的初始化
2. 从服务器下载html，css，js，image等文件
3. 解析html，构建dom树，布局绘制
4. 显示一个webview页面

**优化点：**

1. 将html，css，js，image等资源文件放到本地，设计增量更新策略，对需要更新的部分进行webpack和gzip压缩和cdn加速处理，提神拉取速度。并且在进行网络链接时可以让前端请求的域名跟API域名保持一致，减少DNS解析时间。
2. 通过setBlockNetworkImage(boolean)来设置预先加载非图片部分的内容，延迟图片加载，在onPageStart屏蔽图片加载，在onPageFinished开启图片加载

## 11 如何避免OOM?

11.1 使用更加轻量的数据结构：  
如使用ArrayMap/SparseArray替代HashMap,HashMap更耗内存，因为它需要额外的实例对象来记录Mapping操作，SparseArray更加高效，因为它避免了Key Value的自动装箱，和装箱后的解箱操作

#### 11.2 便面枚举的使用，可以用静态常量或者注解@IntDef替代

#### 11.3 Bitmap优化:

**（）1尺寸压缩**：通过InSampleSize设置合适的缩放

**（2）颜色质量**：设置合适的format，ARGB\_6666/RBG\_545/ARGB\_4444/ALPHA\_6，存在很大差异

**（3）inBitmap** :使用inBitmap属性可以告知Bitmap解码器去尝试使用已经存在的内存区域，新解码的Bitmap会尝试去使用之前那张Bitmap在Heap中所占据的pixel data内存区域，而不是去问内存重新申请一块区域来存放Bitmap。利用这种特性，即使是上千张的图片，也只会仅仅只需要占用屏幕所能够显示的图片数量的内存大小，但复用存在一些限制，具体体现在：在Android 4.4之前只能重用相同大小的Bitmap的内存，而Android 4.4及以后版本则只要后来的Bitmap比之前的小即可。使用inBitmap参数前，每创建一个Bitmap对象都会分配一块内存供其使用，而使用了inBitmap参数后，多个Bitmap可以复用一块内存，这样可以提高性能

11.4 StringBuilder替代String:  
在有些时候，代码中会需要使用到大量的字符串拼接的操作，这种时候有必要考虑使用StringBuilder来替代频繁的“+”

#### 11.5 避免在类似onDraw这样的方法中创建对象，因为它会迅速占用大量内存，引起频繁的GC甚至内存抖动

#### 11.6减少内存泄漏也是一种避免OOM的方法

## 11 模式架构

### 11.1 MVP模式

MVP架构由MVC发展而来。在MVP中，M代表Model，V代表View，P代表Presenter。

**模型层（Model）:**主要是获取数据功能，业务逻辑和实体模型。

**视图层（View）：**对应于Activity或Fragment，负责视图的部分展示和业务逻辑用户交互

**控制层（Presenter）:**负责完成View层与Model层间的交互，通过P层来获取M层中数据后返回给V层，使得V层与M层间没有耦合。

在MVP中 ，Presenter层完全将View层和Model层进行了分离，把主要程序逻辑放在Presenter层实现，Presenter与具体的View层（Activity）是没有直接的关联，是通过定义接口来进行交互的，从而使得当View层（Activity）发生改变时，Persenter依然可以保持不变。View层接口类只应该只有set/get方法，及一些界面显示内容和用户输入，除此之外不应该有多余的内容。绝不允许View层直接访问Model层，这是与MVC最大区别之处，也是MVP核心优点。

### 11.2 MVVM模式

· Model  
Model层就是职责数据的存储、读取网络数据、操作数据库数据以及I/O，一般会有一个ViewModel对象来调用获取这一部分的数据。

· View  
我感觉这里的View才是真正的View，为什么这么说？View层做的仅仅和UI相关的工作，我们只在XML、Activity、Fragment写View层的代码，View层不做和业务相关的事，也就是我们的Activity 不写和业务逻辑相关代码，一般Activity不写更新UI的代码，如果非得要写，那更新的UI必须和业务逻辑和数据是没有关系的，只是单纯UI逻辑来更新UI，比如：滑动时头部颜色渐变、editttext根据输入内容显示隐藏等，简单的说：View层不做任何业务逻辑、不涉及操作数据、不处理数据、UI和数据严格的分开。

· ViewModel  
ViewModel 只做和业务逻辑和业务数据相关的事，不做任何和UI、控件相关的事，ViewModel 层不会持有任何控件的引用，更不会在ViewModel中通过UI控件的引用去做更新UI的事情。ViewModel就是专注于业务的逻辑处理，操作的也都是对数据进行操作，这些个数据源绑定在相应的控件上会自动去更改UI，开发者不需要关心更新UI的事情。

## 12 虚拟机

### Android Dalvik虚拟机和ART虚拟机对比

### 12.1 Dalvik

Android4.4及以前使用的都是Dalvik虚拟机，我们知道Apk在打包的过程中会先将java等源码通过javac编译成.class文件，但是我们的Dalvik虚拟机只会执行.dex文件，这个时候dx会将.class文件转换成Dalvik虚拟机执行的.dex文件。Dalvik虚拟机在启动的时候会先将.dex文件转换成快速运行的机器码，又因为65535这个问题，导致我们在应用冷启动的时候有一个合包的过程，最后导致的一个结果就是我们的app启动慢，这就是Dalvik虚拟机的JIT特性（Just In Time）。

### 12.2 ART

ART虚拟机是在Android5.0才开始使用的Android虚拟机，ART虚拟机必须要兼容Dalvik虚拟机的特性，但是ART有一个很好的特性AOT（ahead of time），这个特性就是我们在安装APK的时候就将dex直接处理成可直接供ART虚拟机使用的机器码，ART虚拟机将.dex文件转换成可直接运行的.oat文件，ART虚拟机天生支持多dex，所以也不会有一个合包的过程，所以ART虚拟机会很大的提升APP冷启动速度。

**ART优点：**

加快APP冷启动速度

提升GC速度

提供功能全面的Debug特性

**ART缺点：**

APP安装速度慢，因为在APK安装的时候要生成可运行.oat文件

APK占用空间大，因为在APK安装的时候要生成可运行.oat文件

### 12.3 处理器

Android支持三种处理器类型：

1. x86体系，目前很少
2. arm体系，分为32位和64位，向下兼容，32位的可以运行在64位上面
3. mips体系，目前很少
4. armeabi/armeabi-v7a，主要用于Android4.0之后，cpu是32位的，缺少对浮点数计算的硬件支持，基本被淘汰。
5. Arm64-v8a，主要用于Android5.0之后，cpu是64位的。

**兼容性：**

Android上启动一个app，都会为它创建一个虚拟机。Android64位的系统加载32位的so文件或者app时，会在创建一个64位的虚拟机的同时还会创建一个32位的虚拟机，这样就能兼容32位的app应用。

#### 12.3.1 so加载流程

手机支持的cpu型号和种类存放在一个abiList集合中，有先后顺序，按照这个顺序遍历jniLib目录，如果这个目录下面有arm64-v8a子目录且里面有so文件，那么手机将加载这个目录下面的so文件就不会再去加载其他目录下面的so文件。

#### 12.3.2 so加载算法

假如jniLib目录下面存在arm64-v8a目录下有a.so文件，armeabi-v7a下面有a.so和b.so文件，手机只会加载arm64-v8a下的a.so文件，永远不会加载b.so文件，所以就会报找不到so文件的异常。

**方案：**

为了节省apk的体积，可以只保存一份so文件，那就是armeabi-v7a下的so文件：

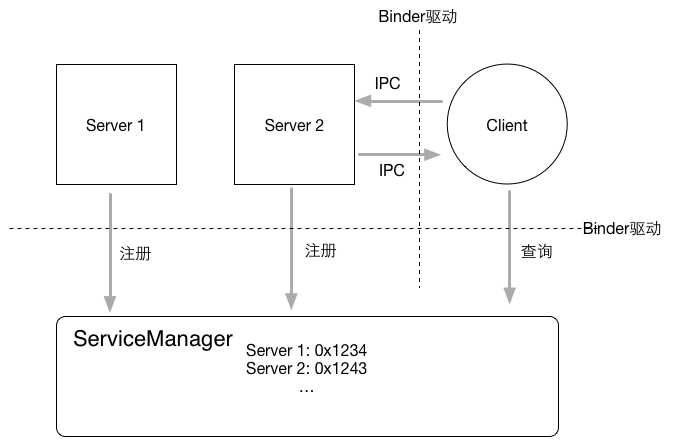
（1）32位的arm手机肯定能加载到armeabi-v7a下的so文件

（2）64位的arm手机要想加载32位的so文件，千万不要在arm64-v8a目录下面放置任何so文件。都放在armeabi-v7a下面即可。

## 13 Binder

### 13.1 Binder组成

Binder分为client进程和server进程。

****

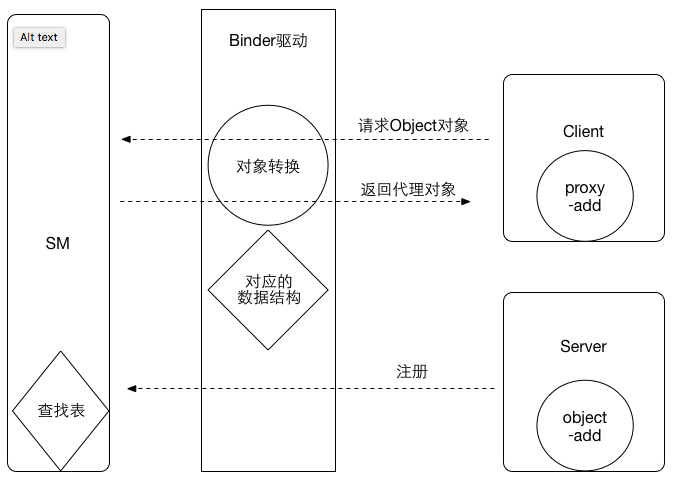
图中的IPC就是进程间通信的意思。

图中的ServiceManager，负责把Binder Server注册到一个容器中。

有人把ServiceManager比喻成电话局，存储着每个住宅的座机电话，还是很恰当的。张三给李四打电话，拨打电话号码，会先转接到电话局，电话局的接线员查到这个电话号码的地址，因为李四的电话号码之前在电话局注册过，所以就能拨通；没注册，就会提示该号码不存在。

对照着Android Binder机制，对着上面这图，张三就是Binder Client，李四就是Binder Server，电话局就是ServiceManager，电话局的接线员在这个过程中做了很多事情，对应着图中的Binder驱动

### 13.2 Binder通信过程

****

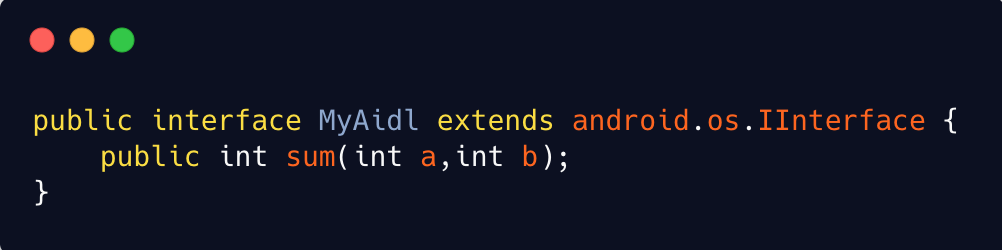
**通信流程：**

1. 首先Server需要在SM容器中注册
2. 其次，Client调用Server的add方法时，需要获取Server的对象，SM不会把Server的对象返回给Client，只是返回其一个代理对象Proxy
3. 最后，Client调用Proxy对象的add方法，SM会帮它调用Client的add方法并把结果返回给Client。

## 14 AIDL

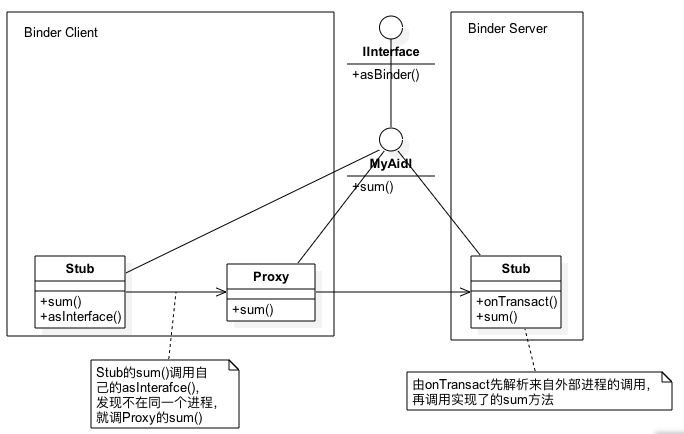
AIDL即Android Interface Definition Language，接口定义语言，用于定义服务器和客户端通信接口的一种描述语言。

1. IBinder
2. IInterface
3. Binder
4. Proxy
5. Stub









**调用方式：**

MyAidl.Stub.asInterface(IBinder对象).sum(1,2);

1. 判断上面的IBinder对象和自己是否在同一个进程，在的话直接使用，不在的话则需要把IBinder参数包装成一个Proxy对象，这时就相当于调用Proxy的sum方法。
2. 在Proxy的sum方法中，会使用Parcelable来准备数据将函数名称和函数参数写入\_data，让\_reply接收函数返回值，最后使用IBinder的transact方法把数据转给Binder的Server端。
3. Server则通过onTransact方法接收Client进程传过来的数据，包括函数名称，函数参数，然后调用相关方法，得到结果返回。

## 15 Linux进程间通信方式

### 15.1 管道

管道是由内核管理的一个缓冲区，它是一种环形的数据结构。大小有限，所以不适合Android大量的进程间通信。

### 15.2 消息队列

消息队列提供了一个进程向另外一个经常发送一个数据块的方法。

### 15.3 socket

### 15.4 共享内存

### 15.5 信号量

### 15.6 Android为何不采用Linux的通信方式而用binder

#### 15.6.1 安全性

传统的LinuxIPC通信方式无法获取进程的UID和PID，从而无法鉴别对方身份，Android 为每个安装好的应用程序分配了自己的 UID，故进程的 UID 是鉴别进程 身份的重要标志。前面提到 C/S 架构，Android 系统中对外只暴露 Client 端， Client 端将任务发送给 Server 端，Server 端会根据权限控制策略，判断 UID/PID 是否满足访问权限，目前权限控制很多时候是通过弹出权限询问对话框，让用 户选择是否运行。

**Android 系统中也依然采用了大量 Linux 现有的 IPC 机制，根据每类 IPC 的原理特性，因时制宜，不同场景特性往往会采用其下 最适宜的。比如在 Android OS 中的 Zygote 进程的 IPC 采用的是 Socket（套 接字）机制**，Android 中的 **Kill Process 采用的 signa（l 信号）机制**等等。而 **Binder 更多则用在 system\_server 进程与上层 App 层的 IPC 交互**。

#### 15.6.2 稳定性

Binder 是基于 C/S 架构的，简单解释下 C/S 架构，是指客 户端(Client)和服务端(Server)组成的架构，Client 端有什么需求，直接发送给 Server 端去完成，架构清晰明朗，Server 端与 Client 端相对独立，稳定性较好；

而共享内存实现方式复杂，没有客户与服务端之别， 需要充分考虑到访问临界

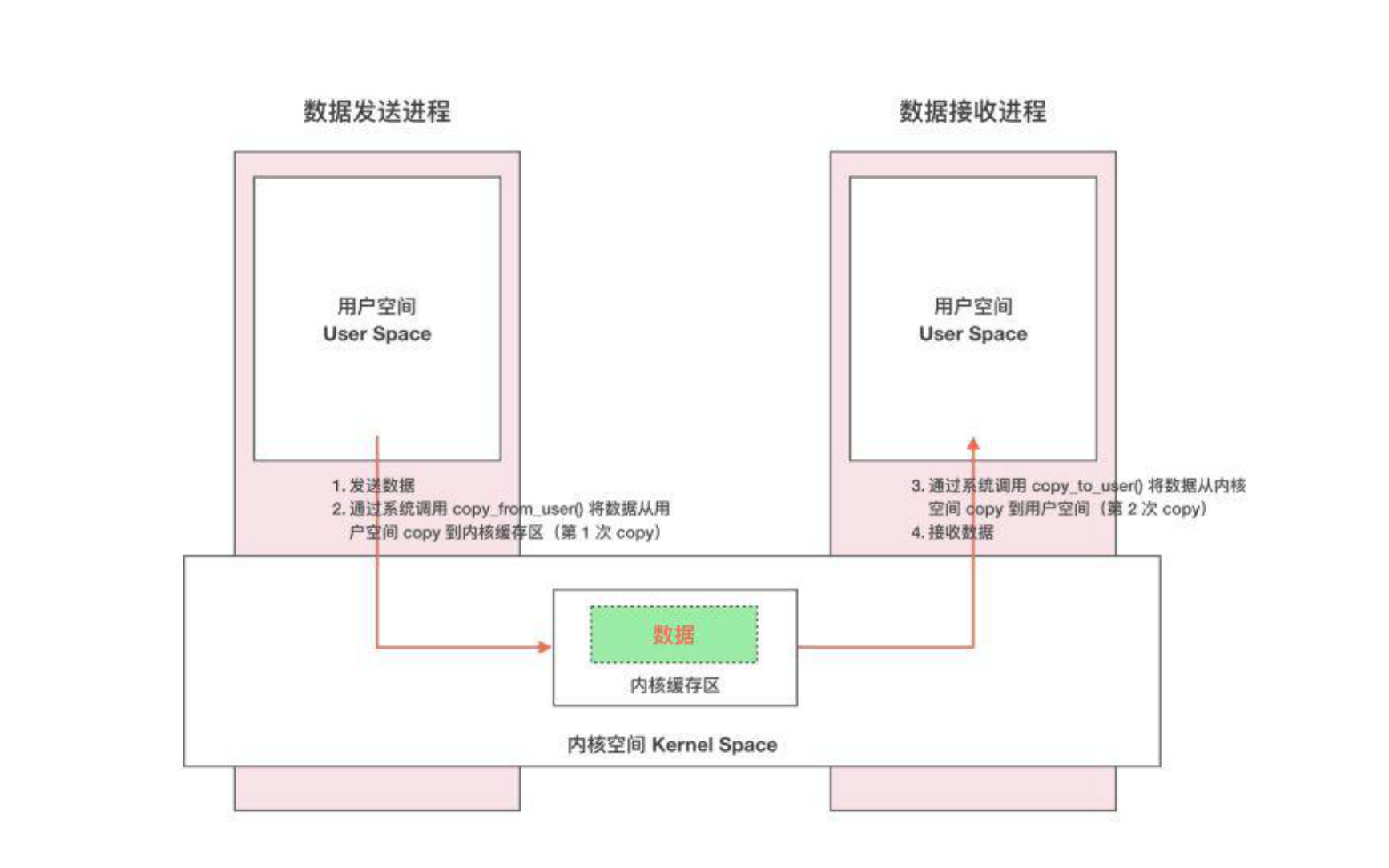
资源的并发同步问题，否则可能会出现死锁等问题；从这稳定性角度看，Binder

架构优越于共享内存。

#### 15.6.3 性能

binder性能仅次于共享内存，共享内存不需要拷贝，而管道，消息队列，socket都需要拷贝两次

### 15.7 linux 数据拷贝



#### 15.7.1 内存映射mmap

**一次完整的 Binder IPC 通信过程通常是这样：**

1. 首先 Binder 驱动在内核空间创建一个数据接收缓存区；

2. 接着在内核空间开辟一块内核缓存区，建立**内核缓存区**和**内核中数据接收**

**缓存区**之间的映射关系，以及**内核中数据接收缓存区**和**接收进程用户空间**

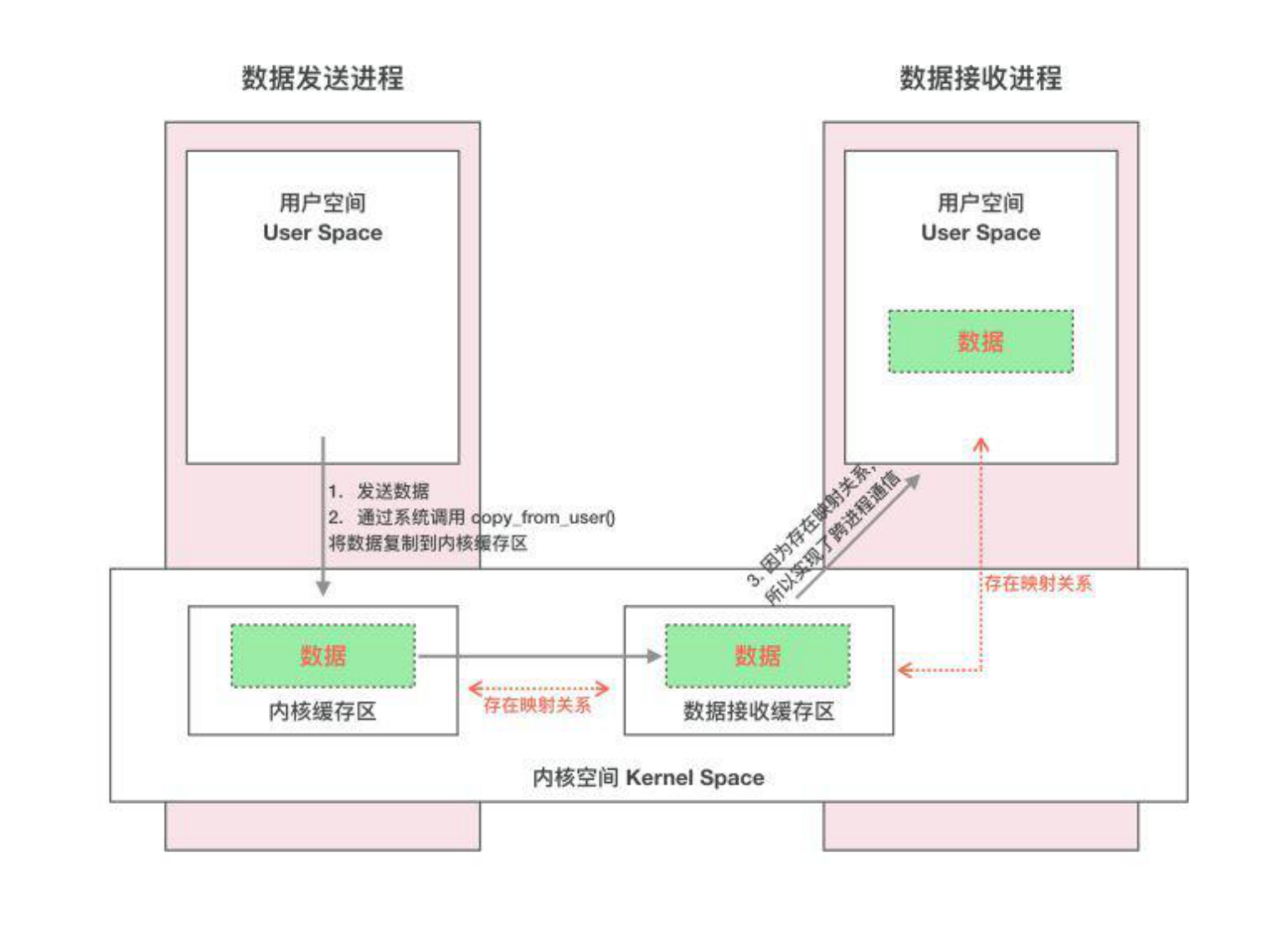
**地址**的映射关系；

3. 发送方进程通过系统调用 copyfromuser() 将数据 copy 到内核中的**内**

**核缓存区**，由于内核缓存区和接收进程的用户空间存在内存映射，因此也

就相当于把数据发送到了接收进程的用户空间，这样便完成了一次进程间

的通信。



## 16 API版本

android3.0 代号Honeycomb, 引入Fragments, ActionBar,属性动画，硬件加速

android4.0 代号Ice Cream，API14：截图功能，人脸识别，虚拟按键，3D优化驱动

android5.0 代号Lollipop API21：调整桌面图标及部件透明度等

android6.0 代号M Marshmallow API23，软件权限管理，安卓支付，指纹支持，App关联，

android7.0 代号N Preview API24，多窗口支持(不影响Activity生命周期)，增加了JIT编译器，引入了新的应用签名方案APK Signature Scheme v2（缩短应用安装时间和更多未授权APK文件更改保护）,严格了权限访问

android8.0 代号O API26，取消静态广播注册，限制后台进程调用手机资源，桌面图标自适应

android9.0 代号P API27，加强电池管理，系统界面添加了Home虚拟键，提供人工智能API，支持免打扰模式