

Table of Contents

| | |
|------------------------------------|--------|
| Introduction | 0 |
| Android基础 | 1 |
| AsyncTask | 1.1 |
| AsyncTask源码剖析-API-23 | 1.1.1 |
| 你真的了解AsyncTask | 1.1.2 |
| DownloadManager | 1.2 |
| Android中DownloadManager的使用 | 1.2.1 |
| Fragment | 1.3 |
| Fragment之我的解决方案-Fragmentation | 1.3.1 |
| Fragment全解析系列1-那些年踩过的坑 | 1.3.2 |
| Fragment全解析系列2-正确的使用姿势 | 1.3.3 |
| View | 1.4 |
| Android中View的getViewTreeObserver方法 | 1.4.1 |
| 你造么,Android中程序的停止状态 | 1.5 |
| 动画 | 1.6 |
| 一种新的Activity转换动画实现方式 | 1.6.1 |
| 实现Activity间的共享控件转场动画 | 1.6.2 |
| 动画基础 | 1.7 |
| 常识 | 1.8 |
| 移动开发需要知道的像素知识 | 1.8.1 |
| 特殊效果 | 1.9 |
| Android悬浮窗TYPE_TOAST小结-源码分析 | 1.9.1 |
| Android无需权限显示悬浮窗---兼谈逆向分析app | 1.9.2 |
| Android逆向实践-使用Smali注入改造YD词典悬浮窗 | 1.9.3 |
| 理解Android中垃圾回收日志信息 | 1.10 |
| 组件 | 1.11 |
| Android中Notification捕捉点击事件的替代方式 | 1.11.1 |

| | |
|---|----------|
| 经验 | 1.12 |
| Android开发中，那些让你相见恨晚的方法、类或接口 | 1.12.1 |
| Android日常开发总结的技术经验60条 | 1.12.2 |
| 你应该知道的那些Android小经验 | 1.12.3 |
| 开发中必须避免的基础问题 | 1.12.4 |
| 绘图 | 1.13 |
| Xfermode | 1.13.1 |
| Xfermode in android | 1.13.1.1 |
| Memory | 2 |
| Basic | 2.1 |
| android系统不释放内存吗 | 2.1.1 |
| booking.com-android客户端的bitmap复用 | 2.1.2 |
| java里的totalMemory、maxMemory、freeMemory究竟是什么 | 2.1.3 |
| 如何偷Android的内存－Tricking Android MemoryFile | 2.1.4 |
| 简析Android的垃圾回收与内存泄露 | 2.1.5 |
| Bitmap | 2.2 |
| Android开发绕不过的坑-你的 Bitmap 究竟占多大内存 | 2.2.1 |
| Android性能优化之Bitmap的内存优化 | 2.2.2 |
| Android高清加载巨图方案-拒绝压缩图片 | 2.2.3 |
| MemoryLeak | 2.3 |
| aliyun-Android内存泄漏总结 | 2.3.1 |
| Android Weak Handler：可以避免内存泄漏的Handler库 | 2.3.2 |
| Android5.1Webview内存泄漏新场景 | 2.3.3 |
| Android中导致内存泄漏的竟然是它----Dialog | 2.3.4 |
| Android内存泄漏之Handler | 2.3.5 |
| Android内存泄漏之Thread | 2.3.6 |
| Android内存泄漏之资源 | 2.3.7 |
| Android内存泄漏开篇 | 2.3.8 |
| GCRoot | 2.3.9 |
| 一个内存泄漏引发的血案-Square | 2.3.10 |

| | |
|--------------------------------------|-------|
| NDK | 3 |
| Android使用NDK-STACK来恢复Cocos2d-x错误堆栈信息 | 3.1 |
| 与原生代码通信 | 3.2 |
| 使用SWIG自动生成JNI代码 | 3.3 |
| 基础 | 3.4 |
| 优化 | 4 |
| Android性能优化典范 | 4.1 |
| Android性能优化典范-第5季 | 4.1.1 |
| ApkSize | 4.2 |
| Android APP终极瘦身指南 | 4.2.1 |
| APK瘦身实践 | 4.2.2 |
| Principle | 4.3 |
| 写出高质量代码的10个Tips | 4.3.1 |
| 资深谷歌安卓工程师对安卓应用开发的建议 | 4.3.2 |
| UI-Optimization | 4.4 |
| Android-UI性能优化详解 | 4.4.1 |
| facebook新闻页ListView优化 | 4.4.2 |
| Instagram是如何提升TextView渲染性能的 | 4.4.3 |
| 启动速度 | 4.5 |
| Android端应用秒开优化体验 | 4.5.1 |
| 怎么计算apk的启动时间 | 4.5.2 |
| 深入浅出RenderThread | 4.6 |
| 兼容_适配 | 5 |
| Android M 新的运行时权限开发者需要知道的一切 | 5.1 |
| Android6.0以下判断摄像头是否可用 | 5.2 |
| 你需要知道的Android拍照适配方案 | 5.3 |
| 如何检测程序是否获得了某项权限 | 5.4 |
| 那些值得你去细细研究的Drawable适配 | 5.5 |
| 内核 | 6 |
| Android 体系架构 | 6.1 |

| | |
|---|-------|
| Android子线程真的不能更新UI么 | 6.2 |
| View的工作原理 | 6.3 |
| 源码分析 | 6.4 |
| Activity的绘制流程简单分析 | 6.4.1 |
| 单元测试 | 7 |
| Android单元测试研究与实践 | 7.1 |
| 工具使用 | 8 |
| adb | 8.1 |
| Android通过adb命令查看内存，CPU，启动时间，电量等信息 | 8.1.1 |
| AllInOne | 8.2 |
| Android内存泄漏终极解决篇-上下篇 | 8.2.1 |
| Android客户端性能优化--魅族资深工程师毫无保留奉献 | 8.2.2 |
| Android性能分析工具整理汇总 | 8.2.3 |
| Android界面性能调优手册 | 8.2.4 |
| 使用StrictMode和MAT分析Android内存泄露 | 8.2.5 |
| 值得推荐的Android应用性能检测工具列表 | 8.2.6 |
| 加速你的Android应用 | 8.2.7 |
| AndroidStudio | 8.3 |
| 使用新版AndroidStudio检测内存泄露和性能 | 8.3.1 |
| BatteryHistorian | 8.4 |
| 使用BatteryHistorian分析应用的电量 | 8.4.1 |
| BlockCanary | 8.5 |
| BlockCanary-轻松找出AndroidApp界面卡顿元凶 | 8.5.1 |
| gradle | 8.6 |
| 更优雅的Android发布自动版本号方案 | 8.6.1 |
| Jenkins | 8.7 |
| Jenkins+Gradle+checkstyle对Android工程源码进行静态代码分析 | 8.7.1 |
| Jenkins+Gradle+findbugs对Android工程源码进行静态代码分析 | 8.7.2 |
| Jenkins+Gradle+Lint对Android工程源码进行静态代码分析 | 8.7.3 |
| Jenkins+Gradle+pmd对Android工程源码进行静态代码分析 | 8.7.4 |

| | |
|-----------------------------------|--------|
| Jenkins+Gradle实现Android开发持续集成问题汇总 | 8.7.5 |
| Jenkins中ConsoleOutput中文乱码问题 | 8.7.6 |
| Jenkins中Jelly邮件模板的配置 | 8.7.7 |
| Jenkins使用学习笔记 | 8.7.8 |
| jenkins如何做到触发远程构建 | 8.7.9 |
| Jenkins学习四-Jenkins邮件配置 | 8.7.10 |
| Jenkins构建的版本包如何发布成可下载的资源—IIS发布方式 | 8.7.11 |
| Jenkins的环境变量的使用 | 8.7.12 |
| Jenkins自编邮件模板 | 8.7.13 |
| Jenkins邮件配置 | 8.7.14 |
| Jenkins里邮件触发器配置SendtoDevelopers | 8.7.15 |
| jenkins里面使用批处理命令进行自动部署 | 8.7.16 |
| Jenkins集成APKSize与dexcount趋势图 | 8.7.17 |
| Mac配置Jenkins | 8.7.18 |
| 使用jenkins自动化构建Android和ios应用 | 8.7.19 |
| 如何在Jenkins发送的构建邮件中提供版本包的下载 | 8.7.20 |
| LeakCanary | 8.8 |
| 用LeakCanary检测内存泄漏 | 8.8.1 |
| Lint | 8.9 |
| AndroidStudio使用lint代码检查设置 | 8.9.1 |
| Android自定义Lint实践 | 8.9.2 |
| MAT | 8.10 |
| Android性能优化之使用MAT分析内存泄露问题 | 8.10.1 |
| Scalpel | 8.11 |
| Scalpel-Jake大神的第三把刀 | 8.11.1 |
| Systrace | 8.12 |
| 使用Systrace分析UI性能 | 8.12.1 |
| TraceView | 8.13 |
| TraceView性能优化工具使用 | 8.13.1 |
| 静态检查工具 | 8.14 |

| | |
|--------------------------------------|-----------|
| 如何更好地利用Pmd与Findbugs和CheckStyle分析结果 | 8.14.1 |
| 常用Java静态代码分析工具的分析与比较 | 8.14.2 |
| 开发Tips | 9 |
| Android开发的那些坑和小技巧 | 9.1 |
| ListView和SwipeRefreshView冲突解决 | 9.2 |
| 神奇的android:clipChildren属性 | 9.3 |
| 扩展知识 | 10 |
| 怎样阅读源代码 | 10.1 |
| 面试时问哪些问题能试出一个Android应用开发者真正的水平 | 10.2 |
| 批量打包 | 11 |
| Android打包的那些事 | 11.1 |
| 一种为Apk动态写入信息的方案 | 11.2 |
| 批量打包-gradle-配置Manifest | 11.3 |
| 批量打包-使用Gradle | 11.4 |
| 批量打包-使用Python | 11.5 |
| 批量打包-美团gradle | 11.6 |
| 插件式开发 | 12 |
| Android博客周刊专题之插件化开发 | 12.1 |
| kaedea系列 | 12.2 |
| 0-索引 | 12.2.1 |
| 1-简单易懂的介绍方式 | 12.2.2 |
| 2-ClassLoader的工作机制 | 12.2.3 |
| 3-加载SD卡的SO库 | 12.2.4 |
| 4-简单加载模式 | 12.2.5 |
| 5-代理Activity模式 | 12.2.6 |
| 6-动态创建Activity模式 | 12.2.7 |
| weishu系列 | 12.3 |
| 1.Android插件化原理解析——概要 | 12.3.1 |
| 10.Android插件化原理解析——DroidPlugin插件通信机制 | 12.3.2 |
| 11.Android插件化原理解析——插件机制之资源管理 | 12.3.3 |

| | |
|--|-------------|
| 12.Android插件化原理解析——不同插件框架方案对比 | 12.3.4 |
| 13.Android插件化原理解析——插件化的未来 | 12.3.5 |
| 2.Android插件化原理解析——Hook机制之动态代理 | 12.3.6 |
| 3.Android插件化原理解析——Hook机制之BinderHook | 12.3.7 |
| 4.Android插件化原理解析——Hook机制之AMS与PMS | 12.3.8 |
| 5.Android插件化原理解析——Activity生命周期管理 | 12.3.9 |
| 6.Android插件化原理解析——插件加载机制 | 12.3.10 |
| 7.Android插件化原理解析——广播的管理 | 12.3.11 |
| 8.Android插件化原理解析——Service的插件化 | 12.3.12 |
| 9.Android插件化原理解析——ContentProvider的插件化 | 12.3.13 |
| 入门 | 12.4 |
| Android动态加载dex技术初探 | 12.4.1 |
| Android动态加载技术三个关键问题详解 | 12.4.2 |
| Android插件化基础 | 12.4.3 |
| Jenkins集成APKSize与dexcount趋势图 | 12.4.4 |
| 插件化开发—动态加载技术加载已安装和未安装的apk | 12.4.5 |
| 其他 | 12.5 |
| Android关于Dex拆分技术详解 | 12.5.1 |
| Android拆分与加载Dex的多种方案对比 | 12.5.2 |
| CodeBoy微信抢红包外挂 | 12.5.3 |
| dexopt的源码跟踪 | 12.5.4 |
| 其实你不知道MultiDex到底有多坑 | 12.5.5 |
| 各大热补丁方案分析和比较 | 12.5.6 |
| 当Field邂逅65535 | 12.5.7 |
| 基础 | 12.6 |
| 代理模式及Java实现动态代理 | 12.6.1 |
| 反射原理 | 12.6.2 |
| 深入探讨Java类加载器 | 12.6.3 |
| 尼古拉斯 | 12.7 |
| Android中插件开发篇之----动态加载Activity-免安装运行程序 | 12.7.1 |

| | |
|--------------------------------|----------|
| Android中插件开发篇之----应用换肤原理解析 | 12.7.2 |
| Android中插件开发篇之----类加载器 | 12.7.3 |
| Android中的动态加载机制 | 12.7.4 |
| images | 12.7.5 |
| 进阶 | 12.8 |
| 5 | 12.8.1 |
| Android-apk动态加载机制的研究 | 12.8.2 |
| Android-APK资源加载和activity生命周期管理 | 12.8.3 |
| Android插件化的一种实现 | 12.8.4 |
| APK动态加载框架DL解析 | 12.8.5 |
| DynamicLoadApk源码解析 | 12.8.6 |
| 动态加载APK原理分享 | 12.8.7 |
| 携程Android应用插件化和动态加载实践 | 12.8.8 |
| 美团DEX自动拆包及动态加载简介 | 12.8.9 |
| 途牛AndroidApp的插件实现 | 12.8.10 |
| 那些年蘑菇街Android组件与插件化背后的故事--插件篇1 | 12.8.11 |
| 那些年蘑菇街Android组件与插件化背后的故事--插件篇2 | 12.8.12 |
| 架构模式 | 13 |
| 设计原则 | 13.1 |
| oop_design_priciple | 13.1.1 |
| S-代表着单一职责原则 | 13.1.2 |
| 软件架构 | 13.2 |
| mvp | 13.2.1 |
| Android应用架构 | 13.2.1.1 |
| mvvm | 13.2.2 |
| Android-MVVM到底是啥-看完就明白了 | 13.2.2.1 |
| 使用 | 13.2.3 |
| 选择恐惧症的福音！教你认清MVC， MVP和MVVM | 13.2.3.1 |
| 架构设计 | 13.2.4 |
| Android简洁架构设计 | 13.2.4.1 |

| | |
|--|----------|
| 解析Android架构设计原则 | 13.2.4.2 |
| 案例 | 13.2.5 |
| Repository设计模式 | 13.2.5.1 |
| 笔记-Android应用架构-Android-Dev-Summit-2015 | 13.2.6 |
| 重构 | 13.3 |
| 关于烂代码的那些事 | 13.3.1 |
| 关于烂代码的那些事情-上 | 13.3.1.1 |
| 关于烂代码的那些事情-下 | 13.3.1.2 |
| 关于烂代码的那些事情-中 | 13.3.1.3 |
| 进程保活 | 14 |
| Android-Persistent属性研究 | 14.1 |
| 关于Android进程保活你需要知道的一切 | 14.2 |
| 微信Android客户端后台保活经验分享 | 14.3 |
| 怎么让Android程序一直后台运行-像QQ一样不被杀死 | 14.4 |
| 问题解决 | 15 |
| Ubuntu64位系统无法使用Android命令问题解决 | 15.1 |
| 项目经验 | 16 |
| 聚划算客户端 | 16.1 |
| 聚划算android客户端1期教训总结 | 16.1.1 |
| 聚划算android客户端2期总结 | 16.1.2 |

开源项目

查看方式

编译依赖gitbook以及JDK 1.6环境

- 1、打开终端，进入到build目录下赋予可执行权限

```
chmod a+x ./build.sh
```

- 2、编译

```
./build.sh
```

- 3、查看

macos

```
open _book/index.html
```

或者直接进入到**_book**目录下打开**index.html**文件

Notes---所有文章均非原创

AsyncTask

AsyncTask源码剖析(API 23)

来源:[Spark Yuan/](#)

Android的UI是线程不安全的，想在子线程中更新UI就必须使用Android的异步操作机制，直接在主线程中更新UI会导致程序崩溃。

Android的异步操作主要有两种，`AsyncTask`和`Handler`。`AsyncTask`是一个轻量的异步类，简单、可控。本文主要结合API 23的源码讲解一下`AsyncTask`到底是什么。

基本用法

声明：Android不同API版本中同一个类的实现方法可能会有不同，本文是基于最新的API 23的源码进行讲解的。

```
public abstract class AsyncTask<Params, Progress, Result>
```

- `Params`: 执行时传入的参数
- `Progress`: 后台任务的执行进度
- `Result`: 返回值

`AsyncTask`是个抽象类，所以需要自己定义一个类继承他，比如

```
class MyAsyncTask extends AsyncTask<Void, Integer, Boolean>
```

AsyncTask的执行过程

- `execute(Params... params)`, 执行异步任务。
- `onPreExecute()`, 在`execute(Params... params)`被调用后执行，界面上的初始化操作，比如显示一个进度条对话框等。
- `doInBackground(Params... params)`, 在`onPreExecute()`完成后执行，用于执行较为费时的操作，如果`AsyncTask`的第三个泛型参数指定的是`Void`，就可以不返回任务执行结果。在执行过程中可以调用`publishProgress(Progress... values)`来更新进度信息。注意，此方法中不可以进行UI操作。
- `onProgressUpdate(Progress... values)`, 调用`publishProgress(Progress... values)`时，此方法被执行，将进度信息更新到UI组件上。

- `onPostExecute(Result result)`, 当后台操作结束时, 此方法将会被调用, 计算结果将做为参数传递到此方法中, 可以利用返回的数据来进行一些UI操作, 比如说提醒任务执行的结果, 以及关闭掉进度条对话框等。

示例

新建一个Activity, 一个Button和一个ProgressBar, 点击Button启动一个AsyncTask并实时更新ProgressBar的状态。

MyAsyncTask

```

class MyAsyncTask extends AsyncTask<Void, Integer, Boolean> {

    @Override
    protected void onPreExecute() {
        super.onPreExecute();
    }

    @Override
    protected void onPostExecute(Boolean aBoolean) {
        super.onPostExecute(aBoolean);
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        super.onProgressUpdate(values);
        progressBar.setProgress(values[0]);
    }

    @Override
    protected Boolean doInBackground(Void... params) {
        for (int i = 0; i < 100; i++) {
            //调用publishProgress,触发onProgressUpdate方法
            publishProgress(i);
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return true;
    }

    @Override
    protected void onCancelled() {
        super.onCancelled();
        progressBar.setProgress(0);
    }
}

```

Button的Click方法

```

startAsyncBtn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        myAsyncTask = new MyAsyncTask();
        myAsyncTask.execute();
    }
});

```

源码剖析

通过上面的例子可以发现，`AsyncTask`使用起来很简单，很方便的就可以在主线程中新建一个子线程进行UI的更新等操作。但是他的实现并不像使用起来那么简单，下面就是对`AsyncTask`的源码进行剖析。

AsyncTask的构造函数

```
public AsyncTask() {
    mWorker = new WorkerRunnable<Params, Result>() {
        public Result call() throws Exception {
            mTaskInvoked.set(true);
            Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
            //noinspection unchecked
            Result result = doInBackground(mParams);
            Binder.flushPendingCommands();
            return postResult(result);
        }
    };
    mFuture = new FutureTask<Result>(mWorker) {
        @Override
        protected void done() {
            try {
                postResultIfNotInvoked(get());
            } catch (InterruptedException e) {
                android.util.Log.w(LOG_TAG, e);
            } catch (ExecutionException e) {
                throw new RuntimeException("An error occurred while executing doInBackground()", e.getCause());
            } catch (CancellationException e) {
                postResultIfNotInvoked(null);
            }
        }
    };
}
```

在构造函数中只做了两件事，初始化`mWorker`和`mFuture`两个变量。`mWorker`是一个`Callable`对象，`mFuture`是一个`FutureTask`对象。`execute()`时会用到。

execute(Params... params)

```
public final AsyncTask<Params, Progress, Result> execute(Params... params) {  
    return executeOnExecutor(sDefaultExecutor, params);  
}
```

只有一行代码，调用了executeOnExecutor方法，sDefaultExecutor实际上是一个串行的线程池，一个进程中所有的AsyncTask全部在这个串行的线程池中排队执行。
executeOnExecutor源码如下。

```
public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec,  
    Params... params) {  
    if (mStatus != Status.PENDING) {  
        switch (mStatus) {  
            case RUNNING:  
                throw new IllegalStateException("Cannot execute task:"  
                    + " the task is already running.");  
            case FINISHED:  
                throw new IllegalStateException("Cannot execute task:"  
                    + " the task has already been executed "  
                    + "(a task can be executed only once)");  
        }  
    }  
  
    mStatus = Status.RUNNING;  
    onPreExecute();  
    mWorker.mParams = params;  
    exec.execute(mFuture);  
    return this;  
}
```

可以看到在这个方法里调用了onPreExecute()，接下来执行exec.execute(mFuture)下面分析一下线程池的执行过程。

```

private static class SerialExecutor implements Executor {
    final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
    Runnable mActive;

    public synchronized void execute(final Runnable r) {
        mTasks.offer(new Runnable() {
            public void run() {
                try {
                    r.run();
                } finally {
                    scheduleNext();
                }
            }
        });
        if (mActive == null) {
            scheduleNext();
        }
    }

    protected synchronized void scheduleNext() {
        if ((mActive = mTasks.poll()) != null) {
            THREAD_POOL_EXECUTOR.execute(mActive);
        }
    }
}

```

系统先把AsyncTask的Params参数封装为FutureTask对象， FutureTask是一个并发类，这里它相当于Runnable；接着将FutureTask交给SerialExecutor的execute方法， 它先把FutureTask插入到任务队列tasks中， 如果这个时候没有正在活动的AsyncTask任务， 那么就会执行下一个AsyncTask任务， 同时当一个AsyncTask任务执行完毕之后， AsyncTask会继续执行其他任务直到所有任务都被执行为止。从这里就可以看出， 默认情况下， AsyncTask是串行执行的

看一下AsyncTask的构造函数， mFuture构造时是把mWork作为参数传进去的， mFuture的run方法会调用mWork的call()方法， 因此call()最终会在线程池中执行。call()中调用了doInBackground()并把返回结果给了postResult。

```

private Result postResult(Result result) {
    @SuppressWarnings("unchecked")
    Message message = getHandler().obtainMessage(MESSAGE_POST_RESULT,
        new AsyncTaskResult<Result>(this, result));
    message.sendToTarget();
    return result;
}

```

可以看到在postResult中通过getHandler()获得一个Handler。

```

private static Handler getHandler() {
    synchronized (AsyncTask.class) {
        if (sHandler == null) {
            sHandler = new InternalHandler();
        }
        return sHandler;
    }
}

```

查看getHandler源码，可以发现getHandler返回的是一个InternalHandler，再来看看InternalHandler的源码。

```

private static class InternalHandler extends Handler {
    public InternalHandler() {
        super(Looper.getMainLooper());
    }

    @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
    @Override
    public void handleMessage(Message msg) {
        AsyncTaskResult<?> result = (AsyncTaskResult<?>) msg.obj;
        switch (msg.what) {
            case MESSAGE_POST_RESULT:
                // There is only one result
                result.mTask.finish(result mData[0]);
                break;
            case MESSAGE_POST_PROGRESS:
                result.mTask.onProgressUpdate(result mData);
                break;
        }
    }
}

```

看到这里已经豁然开朗了，InternalHandler是一个继承Handler的类，在他的handleMessage()方法中对msg进行了判断，如果是MESSAGE_POST_RESULT就执行finish()，如果是MESSAGE_POST_PROGRESS，就执行onProgressUpdate()。MESSAGE_POST_PROGRESS消息是在publishProgress里发出的，详情见源码。

```

protected final void publishProgress(Progress... values) {
    if (!isCancelled()) {
        getHandler().obtainMessage(MESSAGE_POST_PROGRESS,
            new AsyncTaskResult<Progress>(this, values)).sendToTarget();
    }
}

```

finish()

```
private void finish(Result result) {
    if (isCancelled()) {
        onCancelled(result);
    } else {
        onPostExecute(result);
    }
    mStatus = Status.FINISHED;
}
```

如果当前任务被取消，就调用onCancelled()方法，如果没有调用onPostExecute()。

注意事项

- AsyncTask的类必须在主线程中加载，这个过程在Android 4.1及以上版本中已经被系统自动完成。
- AsyncTask对象必须在主线程中创建，execute方法必须在UI线程中调用。
- 一个AsyncTask对象只能执行一次，即只能调用一次execute方法，否则会报运行时异常。

各个版本的区别

在Android 1.6之前，AsyncTask是串行执行任务的，Android 1.6的时候AsyncTask开始采用线程池并行处理任务，但是从Android 3.0开始，为了避免AsyncTask带来的并发错误，AsyncTask又采用一个线程来串行执行任务。尽管如此，在Android 3.0以及后续版本中，我们可以使用AsyncTask的executeOnExecutor方法来并行执行任务。但是这个方法是Android 3.0新添加的方法，并不能在低版本上使用。

总结

整个AsyncTask的源码已经剖析完了，在分析完真个源码后可以发现，AsyncTask并没有什么神秘的，他的本质就是Handler。

我们现在已经知道了AsyncTask如何使用，各个方法会在什么时候调用，有什么作用，相互之间有什么联系，相信大家以后在遇到AsyncTask的任何问题都不会再害怕了，因为AsyncTask的整个源码都翻了个底朝天，还有什么好怕的呢。

你真的了解AsyncTask？

来源:[Weishu's Notes](#)

虽说现在做网络请求有了Volley全家桶和OkHttp这样好用的库，但是在处理其他后台任务以及与UI交互上，还是需要用到AsyncTask。但是你真的了解AsyncTask吗？

AsyncTask的实现几经修改，因此在不同版本的Android系统上表现各异；我相信，任何一个用户量上千万的产品绝对不会在代码里面使用系统原生的AsyncTask，因为它糟糕的兼容性以及极高的崩溃率实在让人不敢恭维。本文将带你了解AsyncTask背后的原理，并给出一个久经考验的AsyncTask修改版。

AsyncTask是什么？

AsyncTask到底是什么呢？很简单，它不过是线程池和Handler的封装；用线程池来处理后台任务，用Handler来处理与UI的交互。线程池使用的是 Executor 接口，我们先了解一下线程池的特性。

线程池ThreadPoolExecutor

JDK5带来的一大改进就是Java的并发能力，它提供了三种并发武器：**并发框架 Executor**，**并发集合类型如 ConcurrentHashMap**，**并发控制类如 countDownLatch 等**；圣经《Effective Java》也说，尽量使用 Executor 而不是直接用 Thread 类进行并发编程。

AsyncTask 内部也使用了线程池处理并发；线程池通过 ThreadPoolExecutor 类构造，这个构造函数参数比较多，它允许开发者对线程池进行定制，我们先看看这每个参数是什么意思，然后看看Android是以何种方式定制的。

ThreadPoolExecutor 的其他构造函数最终都会调用如下的构造函数完成对象创建工作：

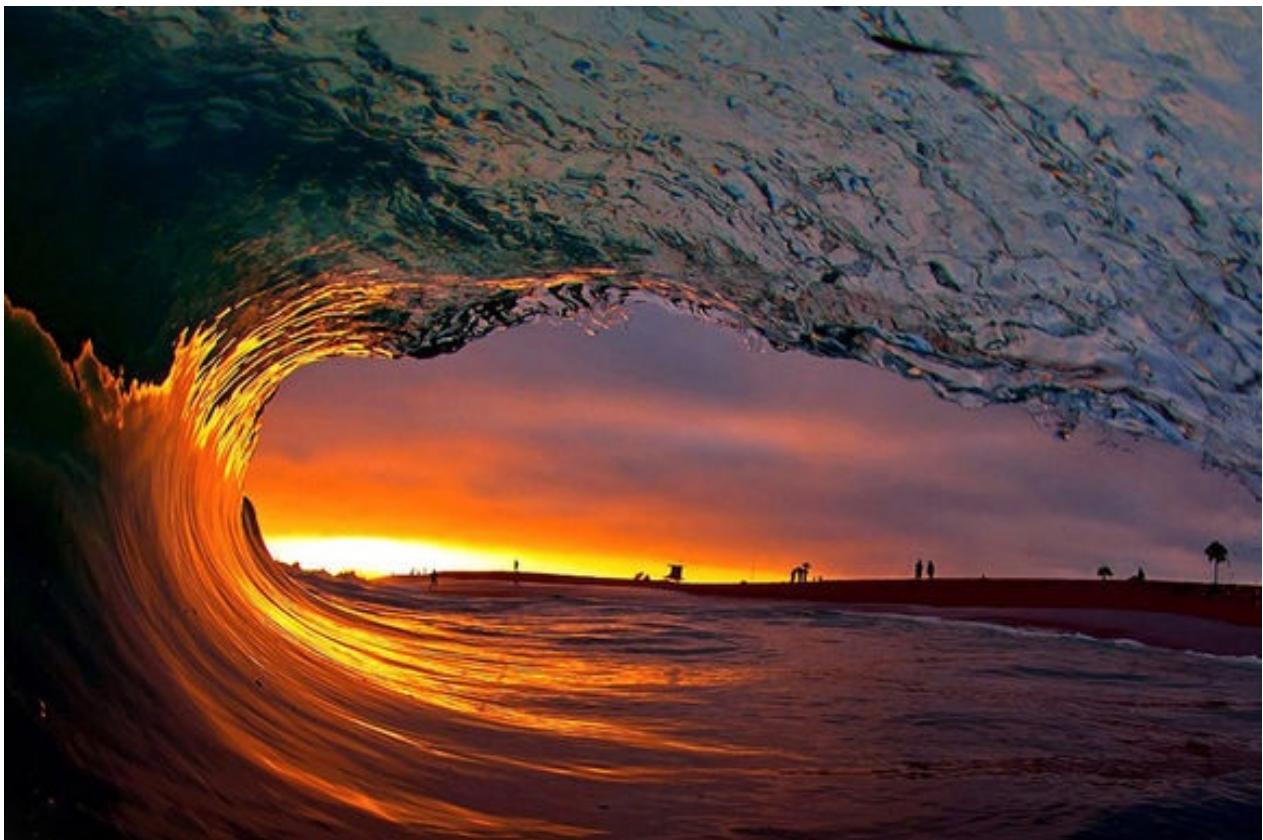
```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

- corePoolSize: 核心线程数目，即使线程池没有任务，核心线程也不会终止（除非设置了allowCoreThreadTimeOut参数）可以理解为“常驻线程”
- maximumPoolSize: 线程池中允许的最大线程数目；一般来说，线程越多，线程调度开销越大；因此一般都有这个限制。
- keepAliveTime: 当线程池中的线程数目比核心线程多的时候，如果超过这个keepAliveTime的时间，多余的线程会被回收；这些与核心线程相对的线程通常被称为缓存线程
- unit: keepAliveTime的时间单位
- workQueue: 任务执行前保存任务的队列；这个队列仅保存由execute提交的Runnable任务
- threadFactory: 用来构造线程池的工厂；一般都是使用默认的；
- handler: 当线程池由于线程数目和队列限制而导致后续任务阻塞的时候，线程池的处理方式。

那么，当一个新的任务到达的时候，线程池中的线程是如何调度的呢？（别慌，讲这么一大段线程池的知识，是为了理解AsyncTask；Be Patient）

- 1、如果线程池中线程的数目少于corePoolSize，就算线程池中有其他的没事做的核心线程，线程池还是会重新创建一个核心线程；直到核心线程数目到达corePoolSize（常驻线程就位）
- 2、如果线程池中线程的数目大于或者等于corePoolSize，但是工作队列workQueue没有满，那么新的任务会放在队列workQueue中，按照FIFO的原则依次等待执行；（当有核心线程处理完任务空闲出来后，会检查这个工作队列然后取出任务默默执行去）
- 3、如果线程池中线程数目大于等于corePoolSize，并且工作队列workQueue满了，但是总线程数目小于maximumPoolSize，那么直接创建一个线程处理被添加的任务。
- 4、如果工作队列满了，并且线程池中线程的数目到达了最大数目maximumPoolSize，那么就会用最后一个构造参数handler处理；**默认的处理方式是直接丢掉任务，然后抛出一个异常。

总结起来，也即是说，当有新的任务要处理时，先看线程池中的线程数量是否大于corePoolSize，再看缓冲队列 workQueue 是否满，最后看线程池中的线程数量是否大于 maximumPoolSize。另外，当线程池中的线程数量大于 corePoolSize 时，如果有线程的空闲时间超过了 keepAliveTime，就将其移除线程池，这样，可以动态地调整线程池中线程的数量。



我们以API 22为例，看一看AsyncTask里面的线程池是以什么参数构造的；AsyncTask里面有“两个”线程池；一个 `THREAD_POOL_EXECUTOR` 一个 `SERIAL_EXECUTOR`；之所以打引号，是因为其实 `SERIAL_EXECUTOR` 也使用 `THREAD_POOL_EXECUTOR` 实现的，只不过加了一个队列弄成了串行而已，那么这个 `THREAD_POOL_EXECUTOR` 是如何构造的呢？

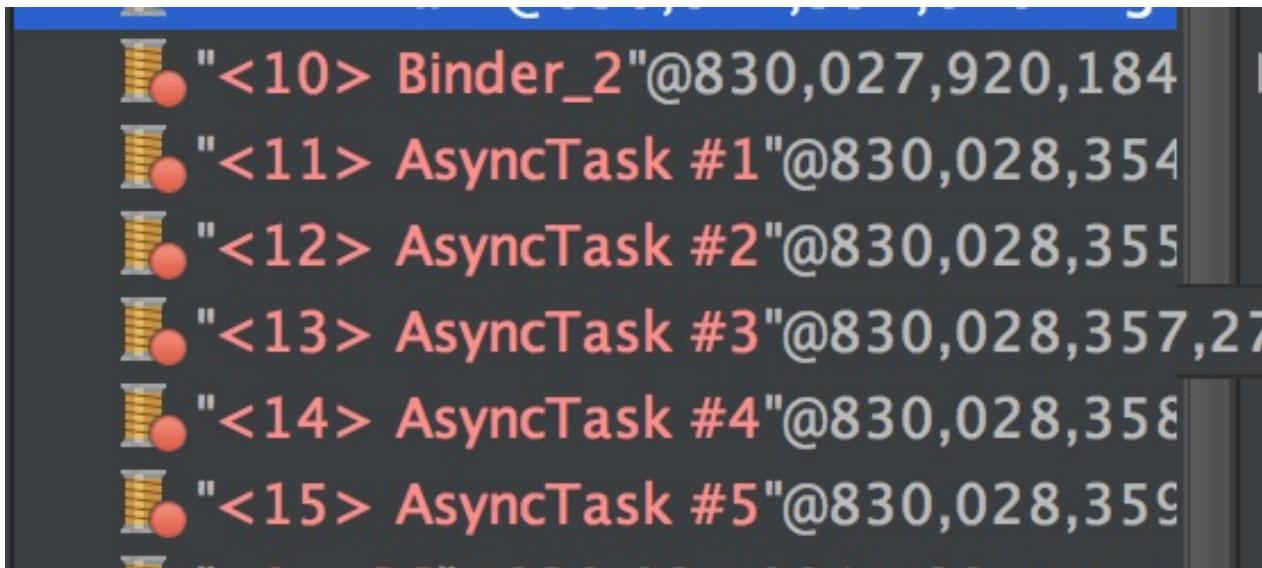
```
private static final int CORE_POOL_SIZE = CPU_COUNT + 1;
private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;
private static final int KEEP_ALIVE = 1;
private static final BlockingQueue<Runnable> sPoolWorkQueue =
    new LinkedBlockingQueue<Runnable>(128);

public static final Executor THREAD_POOL_EXECUTOR
    = new ThreadPoolExecutor(CORE_POOL_SIZE, MAXIMUM_POOL_SIZE, KEEP_ALIVE,
        TimeUnit.SECONDS, sPoolWorkQueue, sThreadFactory);
```

可以看到，AsyncTask里面线程池是一个核心线程数为**CPU + 1**，最大线程数为**CPU * 2 + 1**，工作队列长度为**128**的线程池；并且没有传递handler参数，那么使用的就是默认的Handler（拒绝执行）。

那么问题来了：

- 如果任务过多，那么超过了工作队列以及线程数目的限制导致这个线程池发生阻塞，那么悲剧发生，默认的处理方式会直接抛出一个异常导致进程挂掉。假设你自己写一个异步图片加载的框架，然后用AsyncTask实现的话，当你快速滑动ListView的时候很容易发生这种异常；这也是为什么各大ImageLoader都是自己写线程池和Handler的原因。
- 这个线程池是一个静态变量；那么在同一个进程之内，所有地方使用到的AsyncTask默认构造函数构造出来的AsyncTask都使用的是同一个线程池，如果App模块比较多并且不加控制的话，很容易满足第一条的崩溃条件；如果你不幸在不同的AsyncTask的doInBackground里面访问了共享资源，那么就会发生各种并发编程问题。
- 在AsyncTask全部执行完毕之后，进程中还是会常驻corePoolSize个线程；在Android 4.4 (API 19) 以下，这个corePoolSize是hardcode的，数值是5；API 19改成了cpu + 1；也就是说，在Android 4.4以前；如果你执行了超过五个AsyncTask；然后啥也不干了，进程中还是会有5个AsyncTask线程；不信，你看：



Handler

AsyncTask里面的handler很简单，如下 (API 22代码)：

```
private static final InternalHandler sHandler = new InternalHandler();

public InternalHandler() {
    super(Looper.getMainLooper());
}
```

注意，这里直接用的主线程的Looper；如果去看API 22以下的代码，会发现它没有这个构造函数，而是使用默认的；默认情况下，Handler会使用当前线程的Looper，如果你的AsyncTask是在子线程创建的，那么很不幸，你的onPreExecute和onPostExecute并非在UI线程执行，而是被Handler post到创建它的那个线程执行；如果你在这两个线程更新了UI，那么直接导致崩溃。这也是大家口口相传的**AsyncTask必须在主线程创建**的原因。

另外，AsyncTask里面的这个Handler是一个静态变量，也就是说它是在类加载的时候创建的；如果在你的APP进程里面，以前从来没有使用过AsyncTask，然后在子线程使用AsyncTask的相关变量，那么导致静态Handler初始化，如果在API 16以下，那么会出现上面同样的问题；这就是**AsyncTask必须在主线程初始化**的原因。

事实上，在Android 4.1(API 16)以后，在APP主线程ActivityThread的main函数里面，直接调用了`AsynTask.init`函数确保这个类是在主线程初始化的；另外，init这个函数里面获取了`InternalHandler`的Looper，由于是在主线程执行的，因此，AsyncTask的Handler用的也是主线程的Looper。这个问题从而得到彻底的解决。

AsyncTask是并行执行的吗？

现在知道AsyncTask内部有一个线程池，那么派发给AsyncTask的任务是并行执行的吗？

答案是不确定。在Android 1.5刚引入的时候，AsyncTask的`execute`是串行执行的；到了Android 1.6直到Android 2.3.2，又被修改为并行执行了，这个执行任务的线程池就是`THREAD_POOL_EXECUTOR`，因此在一个进程中，所有的AsyncTask都是并行执行的；但是在Android 3.0以后，如果你使用`execute`函数直接执行AsyncTask，那么这些任务是串行执行的；（你说蛋疼不）源代码如下：

```
public final AsyncTask<Params, Progress, Result> execute(Params... params) {
    return executeOnExecutor(sDefaultExecutor, params);
}
```

这个`sDefaultExecutor`就是用来执行任务的线程池，那么它的值是什么呢？继续看代码：

```
private static volatile Executor sDefaultExecutor = SERIAL_EXECUTOR;
```

因此结论就来了：**Android 3.0以上，AsyncTask默认并不是并行执行的；**

为什么默认不并行执行？

也许你不理解，为什么`AsyncTask`默认把它设计为串行执行的呢？

由于一个进程内所有的`AsyncTask`都是使用的同一个线程池执行任务；如果同时有几个`AsyncTask`一起并行执行的话，恰好`AsyncTask`的使用者在`doInbackground`里面访问了相同的资源，但是自己没有处理同步问题；那么就有可能导致灾难性的后果！

由于开发者通常不会意识到需要对他们创建的所有`AsyncTask`对象里面的`doInbackground`做同步处理，因此，API的设计者为了避免这种无意中访问并发资源的问题，干脆把这个API设置为默认所有串行执行的了。如果你明确知道自己需要并行处理任务，那么你需要使用`executeOnExecutor(Executor exec, Params... params)`这个函数来指定你用来执行任务的线程池，同时为自己的行为负责。（处理同步问题）

实际上《Effective Java》里面有一条原则说的就是这种情况：**不要在同步块里面调用不可信的外来函数**。这里明显违背了这个原则：`AsyncTask`这个类并不知道使用者会在`doInBackground`这个函数里面做什么，但是对它的行为做了某种假设。

如何让`AsyncTask`并行执行？

正如上面所说，如果你确定自己做好了同步处理，或者你没有在不同的`AsyncTask`里面访问共享资源，需要`AsyncTask`能够并行处理任务的话，你可以用带有两个参数的`executeOnExecutor`执行任务：

```
new AsyncTask<Void, Void, Vo
    @Override
    protected Void doInBackground(Void... params) {
        // do something
        return null;
    }
}.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
```

更好的`AsyncTask`

从上面的分析得知，`AsyncTask`有如下问题：

- 默认的`AsyncTask`如果处理的任务过多，会导致程序直接崩溃；
- `AsyncTask`类必须在主线程初始化，必须在主线程创建，不然在API 16以下很概率崩溃。
- 如果你曾经使用过`AsyncTask`，以后不用了；在Android 4.4以下，进程中也默认有5个`AsyncTask`线程；在Android 4.4以上，默认有**CPU + 1**个线程。
- Android 3.0以上的`AsyncTask`默认是串行执行任务的；如果要并行执行需要调用低版

本没有的API，处理麻烦。

因此我们对系统的AsyncTask做了一些修改，在不同Android版本提供一致的行为，并且提高了使用此类的安全性，主要改动如下：

- 添加对于任务过多导致崩溃的异常保护；在这里进行必要的数据统计上报工作；如果出现这个问题，说明AsyncTask不适合这种场景了，需要考虑重构；
- 移植API 22对于Handler的处理；这样就算在线程创建异步任务，也不会有任何问题；
- 提供串行执行和并行执行的 `execute` 方法；默认串行执行，如果明确知道自己在干什么，可以使用 `executeParallel` 并行执行。
- 在`dolnbackgroud`里面频繁崩溃的地方加上 `try..catch`；自己处理数据上报工作。

完整代码见gist，[Better AsyncTask](#)

DownloadManager

Android DownloadManager 的使用

来源:<http://blog.csdn.net/carrey1989/article/details/8060155#>

从Android 2.3 (API level 9) 开始Android用系统服务(Service)的方式提供了Download Manager来优化处理长时间的下载操作。Download Manager处理HTTP

连接并监控连接中的状态变化以及系统重启来确保每一个下载任务顺利完成。

在大多数涉及到下载的情况下使用Download Manager都是不错的选择，特别是当用户切换不同的应用以后下载需要在后台继续进行，以及当下载任务顺利完成

成非常重要的情况 (DownloadManager对于断点续传功能支持很好)。

要想使用Download Manager，使用 `getSystemService` 方法请求系统的 `DOWNLOAD_SERVICE` 服务，代码片段如下：

```
String serviceString = Context.DOWNLOAD_SERVICE;
DownloadManager downloadManager;
downloadManager = (DownloadManager) getSystemService(serviceString);
```

下载文件

要请求一个下载操作，需要创建一个`DownloadManager.Request`对象，将要请求下载的文件的Uri传递给Download Manager的 `enqueue` 方法，代码片段如下所示：

```
String serviceString = Context.DOWNLOAD_SERVICE;
DownloadManager downloadManager;
downloadManager = (DownloadManager) getSystemService(serviceString);

Uri uri = Uri.parse("http://developer.android.com/shareables/icon_templates-v4.0.zip");
DownloadManager.Request request = new Request(uri);
long reference = downloadManager.enqueue(request);
```

在这里返回的reference变量是系统为当前的下载请求分配的一个唯一的**ID**，我们可以通过这个**ID**重新获得这个下载任务，进行一些自己想要进行的操作或者查询下载的状态以及取消下载等等。

我们可以通过 `addRequestHeader` 方法为 `DownloadManager.Request` 对象 `request` 添加 HTTP 头，也可以通过 `setMimeType` 方法重写从服务器返回的 `mime type`。

我们还可以指定在什么连接状态下执行下载操作。`setAllowedNetworkTypes` 方法可以用来限定在 WiFi 还是手机网络下进行下载，`setAllowedOverRoaming` 方法可以用来阻止手机在漫游状态下下载。

下面的代码片段用于指定一个较大的文件只能在 WiFi 下进行下载：

```
request.setAllowedNetworkTypes(Request.NETWORK_WIFI);
```

Android API level 11 介绍了 `getRecommendedMaxBytesOverMobile` 类方法（静态方法），返回一个当前手机网络连接下的最大建议字节数，可以来判断下载是否应该限定在 WiFi 条件下。

调用 `enqueue` 方法之后，只要数据连接可用并且 `Download Manager` 可用，下载就会开始。

要在下载完成的时候获得一个系统通知（notification），注册一个广播接受者来接收 `ACTION_DOWNLOAD_COMPLETE` 广播，这个广播会包含一个 `EXTRA_DOWNLOAD_ID` 信息在 `intent` 中包含了已经完成的这个下载的 ID，代码片段如下所示：

```
IntentFilter filter = new IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE);

BroadcastReceiver receiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        long reference = intent.getLongExtra(DownloadManager.EXTRA_DOWNLOAD_ID, -1);
        if (myDownloadReference == reference) {

        }
    }
};
registerReceiver(receiver, filter);
```

使用 `Download Manager` 的 `openDownloadedFile` 方法可以打开一个已经下载完成的文件，返回一个 `ParcelFileDescriptor` 对象。我们可以通过 `Download Manager` 来查询下载文件的保存地址，如果在下载时制定了路径和文件名，我们也可以直接操作文件。

我们可以为**ACTION_NOTIFICATION_CLICKED** action注册一个广播接受者，当用户从通知栏点击了一个下载项目或者从Downloads app点击可一个下载的项目的时候，系统就会发出一个点击下载项的广播。

代码片段如下：

```
IntentFilter filter = new IntentFilter(DownloadManager.ACTION_NOTIFICATION_CLICKED);

BroadcastReceiver receiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String extraID = DownloadManager.EXTRA_NOTIFICATION_CLICK_DOWNLOAD_IDS;
        long[] references = intent.getLongArrayExtra(extraID);
        for (long reference : references)
            if (reference == myDownloadReference) {
                // Do something with downloading file.
            }
    }
};

registerReceiver(receiver, filter);
```

定制Download Manager Notifications的样式

默认情况下，通知栏中会显示被Download Manager管理的每一个download每一个Notification会显示当前的下载进度和文件的名字，如下图所示(图没了)：

通过Download Manager可以为每一个download request定制**Notification**的样式，包括完全隐藏Notification。下面的代码片段显示了通过 `setTitle` 和 `setDescription` 方法来定制显示在文件下载**Notification**中显示的文字。

```
request.setTitle("Earthquakes");
request.setDescription("Earthquake XML");
```

`setNotificationVisibility` 方法可以用来控制什么时候显示Notification，甚至是隐藏该request的Notification。有以下几个参数：

- **Request.VISIBILITY_VISIBLE**: 在下载进行的过程中，通知栏中会一直显示该下载的Notification，当下载完成时，该Notification会被移除，这是默认的参数值。
- **Request.VISIBILITY_VISIBLE_NOTIFY_COMPLETED**: 在下载过程中通知栏会一直显示该下载的Notification，在下载完成后该Notification会继续显示，直到用户点击

该 Notification或者消除该Notification。

- **Request.VISIBILITY_VISIBLE_NOTIFY_ONLY_COMPLETION**: 只有在下载完成后该Notification才会被显示。
- **Request.VISIBILITY_HIDDEN**: 不显示该下载请求的Notification。如果要使用这个参数，需要在应用的清单文件中加上`DOWNLOAD_WITHOUT_NOTIFICATION`权限。

指定下载保存地址

默认情况下，所有通过Download Manager下载的文件都保存在一个共享下载缓存中，使用系统生成的文件名每一个Request对象都可以制定一个下载保存的地址，通常情况下，所有的下载文件都应该保存在外部存储中，所以我们需要在应用清单文件中加上`WRITE_EXTERNAL_STORAGE`权限：

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

下面的代码片段是在外部存储中指定一个任意的保存位置的方法：

```
request.setDestinationUri(Uri.fromFile(f));
```

f是一个File对象。

如果下载的这个文件是你的应用所专用的，你可能会希望把这个文件放在你的应用在外部存储中的一个专有文件夹中。注意这个文件夹不提供访问控制，所以其他的应用也可以访问这个文件夹。在这种情况下，如果你的应用卸载了，那么在这个文件夹也会被删除。

下面的代码片段是指定存储文件的路径是应用在外部存储中的专用文件夹的方法：

```
request.setDestinationInExternalFilesDir(this,  
    Environment.DIRECTORY_DOWNLOADS, "Bugdroid.png");
```

如果下载的文件希望被其他的应用共享，特别是那些你下载下来希望被Media Scanner扫描到的文件（比如音乐文件），那么你可以指定你的下载路径在外部存储的公共文件夹之下，下面的代码片段是将文件存放到外部存储中的公共音乐文件夹的方法：

```
request.setDestinationInExternalPublicDir(Environment.DIRECTORY_MUSIC,  
    "Android_Rock.mp3");
```

在默认的情况下，通过Download Manager下载的文件是不能被Media Scanner扫描到的，进而这些下载的文件（音乐、视频等）就不会在Gallery和Music Player这样的应用中看到。

为了让下载的音乐文件可以被其他应用扫描到，我们需要调用Request对象的allowScaningByMediaScanner方法。

如果我们希望下载的文件可以被系统的Downloads应用扫描到并管理，我们需要调用Request对象的 setVisibleInDownloadsUi 方法，传递参数true。

取消和删除下载

Download Manager的remove方法可以用来取消一个准备进行的下载，中止一个正在进行的下载，或者删除一个已经完成的下载。

remove方法接受若干个download 的ID作为参数，你可以设置一个或者几个你想要取消的下载的ID，如下代码段所示：

```
downloadManager.remove(REFERENCE_1, REFERENCE_2, REFERENCE_3);
```

该方法返回成功取消的下载的个数，如果一个下载被取消了，所有相关联的文件，部分下载的文件和完全下载的文件都会被删除。

查询Download Manager

你可以通过查询Download Manager来获得下载任务的状态，进度，以及各种细节，通过query方法返回一个包含了下载任务细节的Cursor。

query方法传递一个DownloadManager.Query对象作为参数，通过DownloadManager.Query对象的setFilterById方法可以筛选我们希望查询的下载任务的ID。也可以使用setFilterByStatus方法筛选我们希望查询的某一种状态的下载任务，传递的参数是**DownloadManager.STATUS_***常量，可以指定正在进行、暂停、失败、完成四种状态。

Download Manager包含了一系列COLUMN_*静态String常量，可以用来查询Cursor中的结果列索引。我们可以查询到下载任务的各种细节，包括状态，文件大小，已经下载的字节数，标题，描述，URI，本地文件名和URI，媒体类型以及Media Provider download URI。

下面的代码段是通过注册监听下载完成事件的广播接受者来查询下载完成文件的本地文件名和URI的实现方法：

```

@Override
public void onReceive(Context context, Intent intent) {
    long reference = intent.getLongExtra(DownloadManager.EXTRA_DOWNLOAD_ID, -1);
    if (myDownloadReference == reference) {

        Query myDownloadQuery = new Query();
        myDownloadQuery.setFilterById(reference);

        Cursor myDownload = downloadManager.query(myDownloadQuery);
        if (myDownload.moveToFirst()) {
            int fileNameIdx =
                myDownload.getColumnIndex(DownloadManager.COLUMN_LOCAL_FILENAME);
            int fileUriIdx =
                myDownload.getColumnIndex(DownloadManager.COLUMN_LOCAL_URI);

            String fileName = myDownload.getString(fileNameIdx);
            String fileUri = myDownload.getString(fileUriIdx);

            // TODO Do something with the file.
            Log.d(TAG, fileName + " : " + fileUri);
        }
        myDownload.close();
    }
}

```

对于暂停和失败的下载，我们可以通过查询**COLUMN_REASON**列查询出原因的整数码。

对于**STATUS_PAUSED**状态的下载，可以通过**DownloadManager.PAUSED_***静态常量来翻译出原因的整数码，进而判断出下载是由于等待网络连接还是等待WiFi连接还是准备重新下载三种原因而暂停。

对于**STATUS_FAILED**状态的下载，我们可以通过**DownloadManager.ERROR_***来判断失败的原因，可能是错误码（失败原因）包括没有存储设备，存储空间不足，重复的文件名，或者HTTP errors。

下面的代码是如何查询出当前所有的暂停的下载任务，提取出暂停的原因以及文件名称，下载标题以及当前进度的实现方法：

```

// Obtain the Download Manager Service.
String serviceString = Context.DOWNLOAD_SERVICE;
DownloadManager downloadManager;
downloadManager = (DownloadManager) getSystemService(serviceString);

```

```
// Create a query for paused downloads.  
Query pausedDownloadQuery = new Query();  
pausedDownloadQuery.setFilterByStatus(DownloadManager.STATUS_PAUSED);  
  
// Query the Download Manager for paused downloads.  
Cursor pausedDownloads = downloadManager.query(pausedDownloadQuery);  
  
// Find the column indexes for the data we require.  
int reasonIdx = pausedDownloads.getColumnIndex(DownloadManager.COLUMN_REASON);  
int titleIdx = pausedDownloads.getColumnIndex(DownloadManager.COLUMN_TITLE);  
int fileSizeIdx =  
    pausedDownloads.getColumnIndex(DownloadManager.COLUMN_TOTAL_SIZE_BYTES);  
int bytesDLIdx =  
    pausedDownloads.getColumnIndex(DownloadManager.COLUMN_BYTES_DOWNLOADED_SO_FAR);  
  
// Iterate over the result Cursor.  
while (pausedDownloads.moveToNext()) {  
    // Extract the data we require from the Cursor.  
    String title = pausedDownloads.getString(titleIdx);  
    int fileSize = pausedDownloads.getInt(fileSizeIdx);  
    int bytesDL = pausedDownloads.getInt(bytesDLIdx);  
  
    // Translate the pause reason to friendly text.  
    int reason = pausedDownloads.getInt(reasonIdx);  
    String reasonString = "Unknown";  
    switch (reason) {  
        case DownloadManager.PAUSED_QUEUED_FOR_WIFI :  
            reasonString = "Waiting for WiFi"; break;  
        case DownloadManager.PAUSED_WAITING_FOR_NETWORK :  
            reasonString = "Waiting for connectivity"; break;  
        case DownloadManager.PAUSED_WAITING_TO_RETRY :  
            reasonString = "Waiting to retry"; break;  
        default : break;  
    }  
  
    // Construct a status summary  
    StringBuilder sb = new StringBuilder();  
    sb.append(title).append("\n");  
    sb.append(reasonString).append("\n");  
    sb.append("Downloaded ").append(bytesDL).append(" / ").append(fileSize);  
  
    // Display the status  
    Log.d("DOWNLOAD", sb.toString());  
}  
  
// Close the result Cursor.  
pausedDownloads.close();
```

更新的例子

```
package com.test.update;

import android.annotation.SuppressLint;
import android.app.Activity;
import android.app.DownloadManager;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.net.Uri;
import android.os.Environment;
import android.text.Html;
import android.view.Gravity;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;

import org.json.JSONObject;

import java.io.File;

import okhttp3.Call;

/**
 * 升级管理器
 *
 * @author wangheng
 */
public class UpdateManager {

    public static final String KEY_UPDATE_DOWNLOAD_ID = "keyUpdateDownloadId";
    public static final int INVALID_UPDATE_DOWNLOAD_ID = -1;

    private static final String KEY_HAS_NEW_VERSION = "keyHasNewVersion";

    private UpdateManager(){
        }

    private static class Generator{
        private static final UpdateManager INSTANCE = new UpdateManager();
    }

    public static UpdateManager getInstance(){
        return Generator.INSTANCE;
    }

    /**

```

```

    * 下载
    * @param info 更新信息
    */
public void download(UpdateInfo info){

    DownloadManager dm = (DownloadManager) App.getInstance().getContext()
        .getSystemService(Context.DOWNLOAD_SERVICE);

    DownloadManager.Request request = new DownloadManager.Request(Uri.parse(info.url));
    request.setAllowedNetworkTypes(DownloadManager.Request.NETWORK_MOBILE
        | DownloadManager.Request.NETWORK_WIFI);

    File dir = null;
    if(Environment.MEDIA_MOUNTED.equals(Environment.getExternalStorageState())){
        //dir = App.getInstance().getContext().getExternalCacheDir();
        String path = FileManager.getInstance().getApkSaveDir();
        if(path == null){

            Toast.makeText(App.getInstance().getContext(),
                R.string.no_sdcard,Toast.LENGTH_LONG).show();
            return;
        }
        dir = new File(path);
        if(!dir.exists()){
            Toast.makeText(App.getInstance().getContext(),
                R.string.no_sdcard,Toast.LENGTH_LONG).show();
            return;
        }
        Logger.i("wangheng","#####file exists :" + path);
    }else{
        Toast.makeText(App.getInstance().getContext(),
            R.string.no_sdcard,Toast.LENGTH_LONG).show();
        return;
    }

    File file = new File(dir,info.getVersionCode() + ".apk");
    //if(file.exists()){
    //    file.delete();
    //}

    Logger.i("wangheng",file.getAbsolutePath() + "#####file exists :" + file);
    if(file.exists()){
        String md5 = MD5.getMD5(file);
        if(md5 != null && md5.equalsIgnoreCase(info.getCheckCode())) {
            Intent installIntent = new Intent(Intent.ACTION_VIEW);
            installIntent.setDataAndType(Uri.fromFile(file),
                "application/vnd.android.package-archive");
            installIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            App.getInstance().getContext().startActivity(installIntent);
            return;
        }
    }
}

```

```

    }

    request.setDestinationUri(Uri.fromFile(file));

    request.setMimeType("application/vnd.android.package-archive");
    request.allowScanningByMediaScanner();
    // 设置为可见和可管理
    request.setVisibleInDownloadsUi(true);
    // 自定义标题和描述
    request.setTitle(App.getInstance().getString(R.string.update_statusbar_title))

    String appName = App.getInstance().getString(R.string.app_name);
    String downloading = App.getInstance().getString(R.string.update_statusbar_de
    request.setDescription(appName + "V" + info.getVersionName() + downloading);

    // 在下载过程中通知栏会一直显示该下载的Notification，在下载完成后该Notification会继续显
    // 直到用户点击该Notification或者消除该Notification
    request.setNotificationVisibility(DownloadManager.Request.VISIBILITY_VISIBLE_

    // 返回值是系统为当前的下载请求分配的一个唯一的ID,
    // 我们可以通过这个ID重新获得这个下载任务，进行一些自己想要进行的操作或者查询
    long downloadId = dm.enqueue(request);
    SPManger.getInstance().putLong(KEY_UPDATE_DOWNLOAD_ID, downloadId);

    IntentFilter filter = new IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLE
    App.getInstance().getContext().registerReceiver(new UpdateReceiver(), filter);
}

/**
 * 检查新版本
 * @param callback callback
 */
public void checkUpdate(final ICheckUpdateCallback callback){

    if(null == callback){
        return;
    }

    String channel = App.getInstance().getChannel();
    int versionCode = PackageUtils.getVersionCode(App.getInstance().getContext())
    String versionName = PackageUtils.getVersionName(App.getInstance().getContext()

    PhoneLiveApi.checkUpdate(versionCode, versionName, channel, new StringCallbac
        @Override
        public void onError(Call call, Exception e) {

            SPManger.getInstance().putBoolean(KEY_HAS_NEW_VERSION, false);

            callback.onFailure(e);
        }

    @Override

```

```

public void onResponse(String response) {

    UpdateInfo info = new UpdateInfo();
    JSONObject jsonObject = null;
    try {
        jsonObject = new JSONObject(response);
        JSONObject dataObject = new JSONObject(jsonObject.getString("data"));
        JSONObject infoObject = new JSONObject(dataObject.getString("info"));

        info.setNeedUpdate(infoObject.getInt("needUpdate"));
        info.setVersionName(infoObject.getString("versionName"));
        info.setContent(infoObject.getString("content"));
        info.setVersionCode(infoObject.getInt("versionCode"));
        info.setSize(infoObject.getString("size"));
        info.setDownloadUrl(infoObject.getString("downloadUrl"));
        info.setCheckCode(infoObject.getString("checkCode"));

        SPManger.getInstance().putBoolean(KEY_HAS_NEW_VERSION, info.hasNewVersion());
        callback.onSuccess(info);
    } catch (Exception e) {

        e.printStackTrace();
        info.setNeedUpdate(UpdateInfo.NOT_HAS_NEW_VERSION);
        SPManger.getInstance().putBoolean(KEY_HAS_NEW_VERSION, false);

        callback.onSuccess(info);
    }
}

public DialogPlus showUpdateDialog(final Activity activity, final UpdateInfo info) {
    if(null == info){
        return null;
    }

    @SuppressLint("InflateParams")
    View contentView = App.getInstance().getLayoutInflater()
        .inflate(R.layout.dialog_update,null);

    // 更新版本
    TextView versionNameTextView = (TextView) contentView.findViewById(R.id.tvUpdateName);
    String versionNameMode = App.getInstance().getString(R.string.update_version_mode);
    versionNameTextView.setText(String.format(versionNameMode,info.getVersionName()));

    // 更新大小
    TextView sizeTextView = (TextView) contentView.findViewById(R.id.tvUpdateSize);
    String sizeMode = App.getInstance().getString(R.string.update_size_mode);
    sizeTextView.setText(String.format(sizeMode,info.getSize())));
}

```

```

// 更新内容
TextView contentTextView = (TextView) contentView.findViewById(R.id.tvUpdateContent);
contentTextView.setText(Html.fromHtml(String.valueOf(info.getContent())));

return DialogHelp.showNormalDialog(activity, contentView, Gravity.CENTER,
        new OnClickListener() {
            @Override
            public void onClick(DialogPlus dialog, View view) {
                switch (view.getId()) {
                    case R.id.tvUpdateCancel:
                        dialog.dismiss();
                        break;
                    case R.id.tvUpdateCommit:
                        dialog.dismiss();
                        download(info);
                        break;
                    default:
                        break;
                }
            }
        },
        true, R.drawable.dialog_phone_background);
}

public interface ICheckUpdateCallback{
    void onSuccess(UpdateInfo info);
    void onFailure(Exception e);
}

/**
 * 是否有新版本
 * @return
 */
public boolean hasNewVersion(){
    return SPManger.getInstance().getBoolean(KEY_HAS_NEW_VERSION, false);
}
}

```

Receiver

```

package com.test.update;

import android.app.DownloadManager;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.net.Uri;

import java.io.File;

```

```

/**
 * 升级Receiver
 *
 * @author wangheng
 */
public class UpdateReceiver extends BroadcastReceiver {

    private static final long INVALID_DOWNLOAD_ID = -1;

    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        // 下载完成广播
        if(DownloadManager.ACTION_DOWNLOAD_COMPLETE.equals(action)){

            // 得到DownloadId
            long downloadId = intent.getLongExtra(DownloadManager.EXTRA_DOWNLOAD_ID, INVALID_DOWNLOAD_ID);
            long currentId = SPManager.getInstance().getLong(UpdateManager.KEY_UPDATE_DOWNLOAD_ID,
                UpdateManager.INVALID_UPDATE_DOWNLOAD_ID);

            if(downloadId == currentId){
                Logger.i("wangheng", "auto update when download completed");
                installApk(context, downloadId);
            }
            }else if(DownloadManager.ACTION_NOTIFICATION_CLICKED.equals(action)){
                long ids[] = intent.getLongArrayExtra(DownloadManager.EXTRA_NOTIFICATION_CLICK_ID);
                if(null == ids){
                    return;
                }
                long currentId = SPManager.getInstance().getLong(UpdateManager.KEY_UPDATE_DOWNLOAD_ID,
                    UpdateManager.INVALID_UPDATE_DOWNLOAD_ID);
                for(long id : ids){
                    if(id == currentId){
                        Logger.i("wangheng", "click status bar notification");
                        installApk(context, currentId);
                    }
                }
            }
        }

        private void installApk(Context context, long downloadId) {
            DownloadManager dm = (DownloadManager) App.getInstance().getContext()
                .getSystemService(Context.DOWNLOAD_SERVICE);
            Uri uri = dm.getUriForDownloadedFile(downloadId);
            String path = uri.getPath();

            Intent installIntent = new Intent(Intent.ACTION_VIEW);
            installIntent.setDataAndType(Uri.fromFile(new File(path)), "application/vnd.android.package-archive");
            installIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            context.startActivity(installIntent);
        }
    }
}

```

```
}

package com.test.update;

import java.io.Serializable;

/**
 *
 * 更新信息
 *
 * <pre>
response:

needUpdate:int 1 (1:有新版本, 其他值没有新版本)
versionCode:int 版本号
versionName:String 版本名(不带V)
size:String 大小(如15.4M)
downloadUrl:String 下载地址
content:String 更新内容
// forceUpdate:int 1(1表示必须更新, 其他值表示非必须更新)

request:

versionCode:int 版本号
versionName:String 版本名(不带V)
channel:String 渠道号
osType:int 1(1表示andrsoid, 2表示ios)
</pre>
*
* @author wangheng
*/
public class UpdateInfo implements Serializable{

    private static final long serialVersionUID = -3613164738336297527L;

    // 操作系统类型
    public static final String UPDATE_OS_TYPE = "1";

    // 有新版本
    private static final int HAS_NEW_VERSION = 1;
    // 没有新版本
    public static final int NOT_HAS_NEW_VERSION = 0;

    // 1 代表有新版本 ; 其他值代表没有新版本
    private int needUpdate;
    // 版本号 int 类型
    private int versionCode;
```

```
// 版本名(不带V)
private String versionName;
// 更新内容
private String content;

// 下载地址
private String downloadUrl;

// APK大小, 比如1.5M
private String size;

// md5值
private String checkCode;

public String getCheckCode() {
    return checkCode;
}

public void setCheckCode(String checkCode) {
    this.checkCode = checkCode;
}

public boolean hasNewVersion(){
    return needUpdate == HAS_NEW_VERSION;
}

public int getNeedUpdate() {
    return needUpdate;
}

public void setNeedUpdate(int needUpdate) {
    this.needUpdate = needUpdate;
}

public int getVersionCode() {
    return versionCode;
}

public void setVersionCode(int versionCode) {
    this.versionCode = versionCode;
}

public String getVersionName() {
    return versionName;
}

public void setVersionName(String versionName) {
    this.versionName = versionName;
}

public String getContent() {
    return content;
```

```
}

public void setContent(String content) {
    this.content = content;
}

public String getDownloadUrl() {
    return downloadUrl;
}

public void setDownloadUrl(String downloadUrl) {
    this.downloadUrl = downloadUrl;
}

public String getSize() {
    return size;
}

public void setSize(String size) {
    this.size = size;
}

@Override
public String toString() {
    return "UpdateInfo{" +
        "needUpdate=" + needUpdate +
        ", versionCode=" + versionCode +
        ", versionName='" + versionName + '\'' +
        ", content='" + content + '\'' +
        ", downloadUrl='" + downloadUrl + '\'' +
        ", size='" + size + '\'' +
        ", checkCode='" + checkCode + '\'' +
        '}';
}
}
```

Fragment

Fragment之我的解决方案：Fragmentation

来源:www.jianshu.com

- 1、[Fragment全解析系列（一）：那些年踩过的坑](#)
- 2、[Fragment全解析系列（二）：正确的使用姿势](#)
- 3、[Fragment之我的解决方案：Fragmentation](#)

附：[SwipeBackFragment的实现分析](#)

如果你通读了本系列的前两篇，我相信你可以写出大部分场景都能正常运行的Fragment了。如果你想了解更多，那么你可以看看我封装的这个库：Fragmentation。

本篇主要介绍这个库，解决了一些BUG，使用简单，提供实时查看栈视图等实用功能。

源码地址：[Github](#)，欢迎Fork，提Issues。

[Demo网盘下载](#)

Demo演示：[单Activity+多Fragment](#)



Fragmentation

为"单Activity + 多Fragment的架构","多模块Activity + 多Fragment的架构"而生，帮你简化使用过程，修复了官方Fragment库存在的一些BUG。

特性

- 1、为重度使用Fragment而生
- 2、提供了方便的管理Fragment的方法
- 3、有效解决Fragment重叠问题
- 4、实时查看Fragment的(包括嵌套Fragment)栈视图，方便Fragment嵌套时的调试
- 5、增加启动模式、startForResult等类似Activity方法
- 6、修复官方库里pop(tag/id)出栈多个Fragment时的一些BUG
- 7、完美解决进出栈动画的一些BUG，更自由的管理Fragment的动画
- 8、支持SwipeBack滑动边缘退出(需要使用Fragmentation_SwipeBack库, 详情[README](#))

如何使用

- 1、项目下app的build.gradle中依赖：

```
compile 'me.yokeyword:fragmentation:0.5.4'  
// appcompat v7包也是必须的  
// 如果想使用SwipeBack 滑动边缘退出Fragment/Activity功能, 请再添加下面的库  
// compile 'me.yokeyword:fragmentation-swipeback:0.3.0'
```

- 2、Activity继承SupportActivity：

```

public class MainActivity extends SupportActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(...);
        if (savedInstanceState == null) {
            start(HomeFragment.newInstance());
        }
    }

    /**
     * 设置Container的id, 必须实现
     */
    @Override
    public int setContainerId() {
        return R.id.fl_container;
    }

    /**
     * 设置全局动画, 在SupportFragment可以自由更改其动画
     */
    @Override
    protected FragmentAnimator onCreateFragmentAnimator() {
        // 默认竖向(和安卓5.0以上的动画相同)
        return super.onCreateFragmentAnimator();
        // 设置横向(和安卓4.x动画相同)
        // return new DefaultHorizontalAnimator();
        // 设置自定义动画
        // return new FragmentAnimator(enter,exit,popEnter,popExit);
    }
}

```

- 3、Fragment继承SupportFragment

API

SupportActivity

打开 栈视图 的提示框，在复杂嵌套的时候，可以通过这个来清洗的理清不同阶级的栈视图。

```
// 弹出 栈视图 提示框
showFragmentStackHierarchyView();
```

除此之外包含大部分SupportFragment的方法，请自行查看。

SupportFragment

1、启动相关：

```
// 启动新的Fragment  
start(SupportFragment fragment)  
// 以某种启动模式，启动新的Fragment  
start(SupportFragment fragment, int launchMode)  
// 启动新的Fragment，并能接收到新Fragment的数据返回  
startForResult(SupportFragment fragment, int requestCode)  
// 启动目标Fragment，并关闭当前Fragment  
startWithFinish(SupportFragment fragment)
```

2、关闭Fragment：

```
// 关闭当前Fragment  
pop();  
  
// 关闭某一个Fragment栈内之上的Fragments  
popTo(Class fragmentClass, boolean includeSelf);  
  
// 如果想出栈后，紧接着开始.beginTransaction()开始一个事务，请使用下面的方法：  
// 在第二篇文章内的Fragment事务部分的问题有解释原因  
popTo(Class fragmentClass, boolean includeSelf, Runnable afterTransaction)
```

3、在SupportFragment内，支持监听返回按钮事件：

```
@Override  
public boolean onBackPressedSupport() {  
    // 返回false，则继续传递返回事件； 返回true则不继续传递  
    return false;  
}
```

4、定义当前Fragment的动画，复写onCreateFragmentAnimation方法：

```
@Override  
protected FragmentAnimator onCreateFragmentAnimation() {  
    // 获取在SupportActivity里设置的全局动画对象  
    FragmentAnimator fragmentAnimator = _mActivity.getFragmentAnimator();  
    fragmentAnimator.setEnter(0);  
    fragmentAnimator.setExit(0);  
    return fragmentAnimator;  
  
    // 也可以直接通过  
    // return new FragmentAnimator(enter, exit, popEnter, popExit)设置一个全新的动画  
}
```

5、获取当前Activity/Fragment内栈顶(子)Fragment:

```
getTopFragment();
```

6、获取栈内某个Fragment对象：

```
findFragment(Class fragmentClass);  
  
// 获取某个子Fragment对象  
findChildFragment(Class fragmentClass);
```

更多

隐藏/显示 输入法:

```
// 隐藏软键盘 一般用在onHidden里  
hideSoftInput();  
// 显示软键盘  
showSoftInput(View view);
```

下面是DetailFragment startForResult ModifyTitleFragment的代码：

DetailFragment.class里：

```
startForResult(ModifyDetailFragment.newInstance(mTitle), REQ_CODE);
@Override
public void onFragmentResult(int requestCode, int resultCode, Bundle data) {
    super.onFragmentResult(requestCode, resultCode, data);
    if (requestCode == REQ_CODE && resultCode == RESULT_OK ) {
        // 在此通过Bundle data 获取返回的数据
    }
}
```

ModifyTitleFragment.class里：

```
Bundle bundle = new Bundle();
bundle.putString("title", "xxxx");
setFramgentResult(RESULT_OK, bundle);
```

下面是以一个singleTask模式start一个Fragment的标准代码：

```
HomeFragment fragment = findFragment(HomeFragment.class);
if (fragment == null) {
    fragment = HomeFragment.newInstance();
} else{
    Bundle newBundle = new Bundle();
    // 传递的bundle数据，会调用目标Fragment的onNewBundle(Bundle newBundle)方法
    fragment.putNewBundle(newBundle);
}
// homeFragment以SingleTask方式启动
start(fragment, SupportFragment.SINGLETASK);

// 在HomeFragment.class中：
@Override
protected void onNewBundle(Bundle newBundle){
    // 在此可以接收到数据
}
```

关于Fragmentation帮你恢复Fragment，你需要知道的2个概念：

“同级”式：比如QQ的主界面，“消息”、“联系人”、“动态”，这3个Fragment属于同级关系

“流程”式：比如登录->注册/忘记密码->填写信息->跳转到主页Activity

对于Activity内的“流程”式Fragments（比如登录->注册/忘记密码->填写信息->跳转到主页Activity），Fragmentation帮助你处理了栈内的恢复，保证Fragment不会重叠，你不需要再自己处理了。

但是如果你的Activity内的Fragments是“同级”的，那么需要你复写onHandleSaveInstanceState()使用findFragmentByTag(tag)或getFragments()去恢复处理。

```
@Override  
protected void onHandleSaveInstancState(Bundle savedInstanceState) {  
    // 复写的时候 下面的super一定要删掉  
    // super.onHandleSaveInstancState(savedInstanceState);  
    // 在此处 通过findFragmentByTag或getFraments来恢复，详情参考第二篇文章  
}
```

而如果你有Fragment嵌套，那么不管是“同级”式还是“流程”式，你都需要自己去恢复处理。

Fragment全解析系列（一）：那些年踩过的坑

来源:www.jianshu.com

Fragment系列文章：

- 1、[Fragment全解析系列（一）：那些年踩过的坑](#)
- 2、[Fragment全解析系列（二）：正确的使用姿势](#)
- 3、[Fragment之我的解决方案：Fragmentation](#)

本篇主要介绍一些最常见的Fragment的坑以及官方Fragment库的那些自身的BUG，这些BUG在你深度使用时会遇到，比如Fragment嵌套时或者单Activity + 多Fragment架构时遇到的坑。

如果想看较为实用的技巧，请直接看[第二篇](#)

Fragment是可以让你的app纵享丝滑的设计，如果你的app想在现在基础上性能大幅度提高，并且占用内存降低，同样的界面Activity占用内存比Fragment要多，响应速度Fragment比Activity在中低端手机上快了很多，甚至能达到好几倍！如果你的app当前或以后有移植平板等平台时，可以让你节省大量时间和精力。

简陋的目录

- 1、`getActivity()`空指针
- 2、Fragment重叠异常----正确使用`hide`、`show`的姿势
- 3、Fragment嵌套的那些坑
- 4、不靠谱的出栈方法`remove()`
- 5、多个Fragment同时出栈的那些深坑BUG
- 6、超级深坑 Fragment转场动画

开始之前

最新版知乎，单Activity多Fragment的架构，响应可以说非常“丝滑”，非要说缺点的话，就是没有转场动画，并且转场会有类似闪屏现象。我猜测可能和Fragment转场动画的一些BUG有关。（这系列的最后一篇文章我会给出我的解决方案，可以自定义转场动画，并能在各种特殊情况下正常运行。）

但是！Fragment相比较Activity要难用很多，在多Fragment以及嵌套Fragment的情况下更是如此。

更重要的是Fragment的坑真的太多了，看Square公司的这篇文章吧，[Square：从今天开始抛弃Fragment吧！](#)

当然，不能说不再用Fragment，Fragment的这些坑都是有解决办法的，官方也在逐步修复一些BUG。

下面罗列一些，有常见的，也有极度隐蔽的一些坑，也是我在用单Activity多Fragment时遇到的坑，可能有更多坑可以挖掘…

在这之前为了方便后面文章的介绍，先规定一个“术语”，安卓app有一种特殊情况，就是app运行在后台的时候，系统资源紧张的时候导致把app的资源全部回收（杀死app的进程），这时把app再从后台返回到前台时，app会重启。这种情况下文简称为：“**内存重启**”。

在系统要把app回收之前，系统会把Activity的状态保存下来，Activity的FragmentManager负责把Activity中的Fragment保存起来。在“内存重启”后，Activity的恢复是从栈顶逐步恢复，Fragment会在宿主Activity的 `onCreate` 方法调用后紧接着恢复（从`onAttach`生命周期开始）。

getActivity()空指针

可能你遇到过`getActivity()`返回null，或者平时运行完好的代码，在“内存重启”之后，调用`getActivity()`的地方却返回null，报了空指针异常。

大多数情况下的原因：你在调用了`getActivity()`时，当前的Fragment已经`onDetach()`了宿主Activity。比如：你在pop了Fragment之后，该Fragment的异步任务仍然在执行，并且在执行完成后调用了`getActivity()`方法，这样就会空指针。

- 解决办法：

更“安全”的方法：(对于Fragment已经 `onDetach` 这种情况，我们应该避免在这之后再去调用宿主Activity对象，比如取消这些异步任务，但我们的团队可能会有粗心大意的情况，所以下面给出的这个方案会保证安全)

在Fragment基类里设置一个 `Activity mActivity` 的全局变量，在 `onAttach(Activity activity)` 里赋值，使用`mActivity`代替 `getActivity()`，保证Fragment即使在 `onDetach` 后，仍持有Activity的引用（有引起内存泄露的风险，但是相比空指针闪退，这种做法“安全”些），即：

```

protected Activity mActivity;
@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    this.mActivity = activity;
}

/**
 * 如果你用了support 23的库，上面的方法会提示过时，有强迫症的小伙伴，可以用下面的方法代替
 */
@Override
public void onAttach(Context context) {
    super.onAttach(context);
    this.mActivity = (Activity)context;
}

```

Fragment重叠异常-----正确使用hide、show的姿势

如果你 `add()` 了几个Fragment，使用 `show()`、`hide()` 方法控制，比如微信、QQ的底部tab等情景，如果你什么都不做的话，在“内存重启”后回到前台，app的这几个Fragment界面会重叠。

原因是FragmentManager帮我们管理Fragment,每当我们离开该Activity,FragmentManager都会保存它的Fragments，当发生“内存重启”，他会从栈底向栈顶的顺序恢复Fragments，并且全都都是以 `show()` 的方式，所以我们看到了界面重叠。（如果是replace，恢复顺序和Activity一致：栈顶先恢复，当pop返回上一个Fragment时，再恢复这个Fragment）

这里给出2个解决方案：（为方便描述，以下皆不考虑Fragment嵌套的情况）

1、是大家比较熟悉的 `findFragmentByTag`：

即在 `add()` 或者 `replace()` 时绑定一个tag，一般我们是用fragment的类名作为tag，然后在发生“内存重启”时，通过 `findFragmentByTag` 找到对应的Fragment，并 `hide()` 需要隐藏的fragment。

下面是个标准恢复写法：

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity);  
  
    TargetFragment targetFragment;  
    HideFragment hideFragment;  
  
    if (savedInstanceState != null) { // “内存重启”时调用  
        targetFragment = getSupportFragmentManager().findFragmentByTag(targetFragment.  
        hideFragment = getSupportFragmentManager().findFragmentByTag(hideFragment.get  
        // 解决重叠问题  
        getSupportFragmentManager().beginTransaction()  
            .show(targetFragment)  
            .hide(hideFragment)  
            .commit();  
    }else{ // 正常时  
        targetFragment = TargetFragment.newInstance();  
        hideFragment = HideFragment.newInstance();  
  
        getSupportFragmentManager().beginTransaction()  
            .add(R.id.container, targetFragment, targetFragment.getClass().getNam  
            .add(R.id.container, hideFragment, hideFragment.getClass().getName())  
            .hide(hideFragment)  
            .commit();  
    }  
}
```

如果你想恢复到用户离开时的那个Fragment的界面，你还需要
在 `onSaveInstanceState(Bundle outState)` 里 保存离开时的那个见面的tag或下标，
在 `onCreate` “内存重启”代码块中，取出 tag/下标，进行恢复。

2、使 用 `getSupportFragmentManager().getFragments()` 恢复

通过 `getFragments()` 可以获取到当前FragmentManager管理的栈内所有Fragment。

标准写法如下：

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity);  
  
    TargetFragment targetFragment;  
    HideFragment hideFragment;  
  
    if (savedInstanceState != null) { // “内存重启”时调用  
        List<Fragment> fragmentList = getSupportFragmentManager().getFragments();  
        for (Fragment fragment : fragmentList) {  
            if(fragment instanceof TargetFragment){  
                targetFragment = (TargetFragment)fragment;  
            }else if(fragment instanceof HideFragment){  
                hideFragment = (HideFragment)fragment;  
            }  
        }  
        // 解决重叠问题  
        getSupportFragmentManager().beginTransaction()  
            .show(targetFragment)  
            .hide(hideFragment)  
            .commit();  
    }else{ // 正常时  
        targetFragment = TargetFragment.newInstance();  
        hideFragment = HideFragment.newInstance();  
  
        // 这里add时, tag可传可不传  
        getSupportFragmentManager().beginTransaction()  
            .add(R.id.container)  
            .add(R.id.container, hideFragment)  
            .hide(hideFragment)  
            .commit();  
    }  
}
```

从代码看起来，这种方式比较复杂，但是这种方式在一些场景下比第一种方式更加简便有效。

我会在[下一篇](#)中介绍在不同场景下如果选择，何时用 `findFragmentByTag()`，何时用 `getFragments()` 恢复。

顺便一提，有些小伙伴会用一种并不合适的方法恢复Fragment，虽然效果也能达到，但并不恰当。即：

```

// 保存
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    getSupportFragmentManager().putFragment(outState, KEY, targetFragment);
}

// 恢复
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_scrolling);

    if (savedInstanceState != null) {
        Fragment targetFragment = getSupportFragmentManager().getFragment(savedInstanceState, KEY);
    }
}

```

如果仅仅为了找回栈内的Fragment，使用 `putFragment(bundle, key, fragment)` 保存 fragment，是完全没有必要的；因为FragmentManager在任何情况都会帮你存储 Fragment，你要做的仅仅是在“内存重启”后，找回这些Fragment即可。。

Fragment嵌套的那些坑

其实一些小伙伴遇到的很多嵌套的坑，大部分都是由于对嵌套的栈视图产生混乱，只要理清栈视图关系，做好恢复相关工作以及正确选择是使用 `getFragmentManager()` 还是 `getChildFragmentManager()` 就可以避免这些问题。

这部分内容是我们感觉Fragment非常难用的一个点，我会在[下一篇](#)中，详细介绍使用 Fragment嵌套的一些技巧，以及如何清晰分析各个层级的栈视图。

- 附：`startActivityForResult`接收返回问题

在support 23.2.0以下的支持库中，对于在嵌套子Fragment 的 `startActivityForResult ()`，会发现无论如何都不能 在 `onActivityResult()` 中接收到返回值，只有最顶层的父Fragment才能接收到，这是一个support v4库的一个BUG，不过在前两天发布的 support 23.2.0 库中，已经修复了该问题，嵌套的子Fragment也能正常接收到返回数据了！

不靠谱的出栈方法`remove()`

如果你想让某一个Fragment出栈，使用 `remove()` 并不靠谱。它并不能真正将Fragment从栈内移除，如果你在2秒后（确保Fragment事务已经完成）打印 `getSupportFragmentManager().getFragments()`，会发现该Fragment依然存在。并且依然可以返回到被`remove`的Fragment，而且是空白页面。

`popBackStack()` 系列方法才能真正出栈，这也就引入下一个深坑，`popBackStack(String tag, int flags)` 等系列方法的BUG。

多个Fragment同时出栈的那些深坑BUG

在Fragment库中如下4个方法是有BUG的：

- 1、`popBackStack(String tag,int flags)`
- 2、`popBackStack(int id,int flags)`
- 3、`popBackStackImmediate(String tag,int flags)`
- 4、`popBackStackImmediate(int id,int flags)`

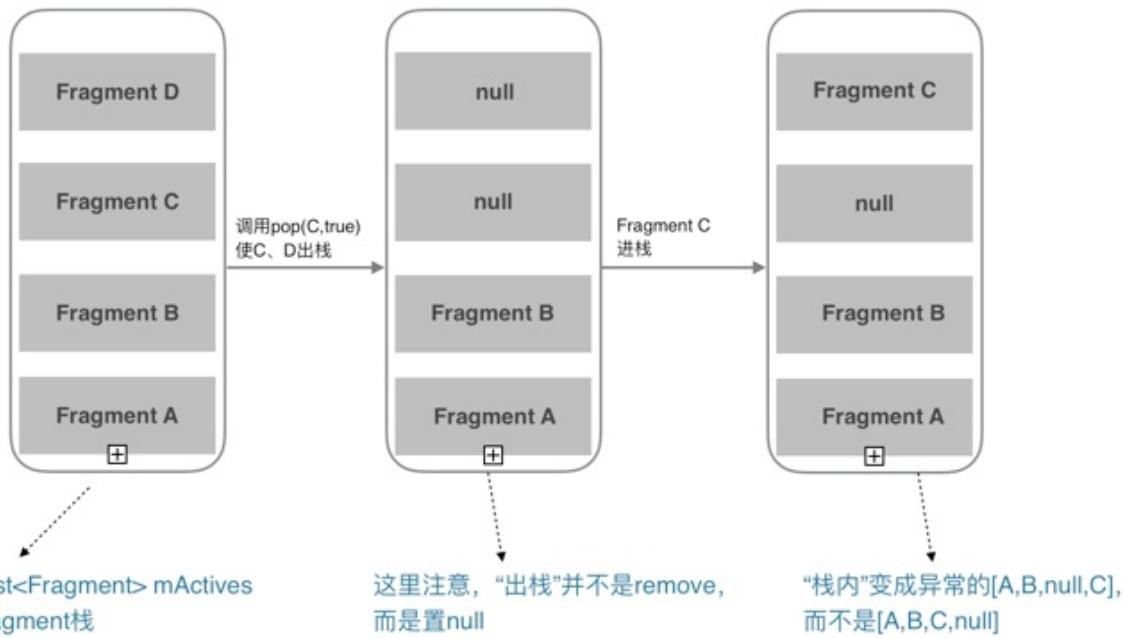
上面4个方法作用是，出栈到tag/id的fragment，即一次多个Fragment被出栈。

1、FragmentManager栈中管理fragment下标位置的数组 ArrayList mAvailIndices 的BUG

下面的方法`FragmentManagerImpl`类方法，产生BUG的罪魁祸首是管理Fragment栈下标的 `mAvailIndices` 属性：

```
void makeActive(Fragment f) {
    if (f.mIndex >= 0) {
        return;
    }
    if (mAvailIndices == null || mAvailIndices.size() <= 0) {
        if (mActive == null) {
            mActive = new ArrayList<Fragment>();
        }
        f.setIndex(mActive.size(), mParent);
        mActive.add(f);
    } else {
        f.setIndex(mAvailIndices.remove(mAvailIndices.size()-1), mParent);
        mActive.set(f.mIndex, f);
    }
    if (DEBUG) Log.v(TAG, "Allocated fragment index " + f);
}
```

上面代码最终导致了栈内顺序不正确的问题，如下图



上面的这个情况，会一次异常，一次正常。带来的问题就是“内存重启”后，各种异常甚至 Crash。

我发现这BUG的时候，我也懵比了，幸好，[stackoverflow](#)上有大神给出了[解决方案](#)！ hack FragmentManagerImpl 的 mAvailIndices，对其进行一次 collections.reverseOrder() 降序排序，保证栈内Fragment的index的正确。

```
public class FragmentTransactionBugFixHack {

    public static void reorderIndices(FragmentManager fragmentManager) {
        if (!(fragmentManager instanceof FragmentManagerImpl))
            return;
        FragmentManagerImpl fragmentManagerImpl = (FragmentManagerImpl) fragmentManager;
        if (fragmentManagerImpl.mAvailIndices != null && fragmentManagerImpl.mAvailIndices.size() > 0)
            Collections.sort(fragmentManagerImpl.mAvailIndices, Collections.reverseOrder());
    }
}
```

使用方法就是通过 `popBackStackImmediate(tag/id)` 多个Fragment后，调用

```

hanler.post(new Runnable(){
    @Override
    public void run() {
        FragmentTransactionBugFixHack.reorderIndices(fragmentManager);
    }
});

```

2、popBackStack的BUG

`popBackStack` 和 `popBackStackImmediate` 的区别在于前者是加入到主线队列的末尾，等其它任务完成后才开始出栈，后者是立刻出栈。

但是！！！在我使用的过程中我发现，`popBackStack(tag/id)` 这2个方法是有BUG的！

先不说有转场动画情况下的各种深坑BUG...

如果你 `popBackStack` 多个Fragment后，紧接着 `beginTransaction()` add一个新的一个 Fragment，接着发生了“内存重启”后，你再执行 `popBackStack()`，app就会Crash。

所以，如果你想出栈多个Fragment，你应尽量使用 `popBackStackImmediate(tag/id)`，而不是 `popBackStack(tag/id)`，如果你想在出栈后，立刻 `beginTransaction()` 开始一项事务，你应该把事务的代码post到主线程的消息队列里，下一篇有详细描述。

超级深坑 Fragment转场动画

如果你的Fragment没有转场动画，或者使用 `setCustomAnimations(enter, exit)` 的话，那么上面的那些坑解决后，你可以愉快的玩耍了。

```

getFragmentManager().beginTransaction()
    .setCustomAnimations(enter, exit)
    // 如果你有通过tag/id同时出栈多个Fragment的情况时,
    // 请谨慎使用.setCustomAnimations(enter, exit, popEnter, popExit)
    // 因为在出栈多Fragment时，伴随出栈动画，会在某些情况下发生异常
    // 你还需要搭配Fragment的onCreateAnimation()临时取消出栈动画

```

总结起来就是Fragment没有出栈动画的话，可以避免很多坑。

如果想让出栈动画运作正常的话，需要使用 Fragment的`onCreateAnimation` 中控制动画。

```

@Override
public Animation onCreateAnimation(int transit, boolean enter, int nextAnim) {
    // 此处设置动画
}

```

但是用代价也是有的，你需要解决出栈动画带来的几个坑。

1、pop多个Fragment时转场动画 带来的问题

在使用 `pop(tag/id)` 出栈多个Fragment的这种情况下，务必不能设定转场动画；

原因在于这种情景下，如果发生“内存重启”后，Fragment并不会被FragmentManager正常保存下来。

2、进入新的Fragment并立刻关闭当前Fragment 时的一些问题

- (1) 如果你想从当前Fragment进入一个新的Fragment，并且同时要关闭当前Fragment。由于数据结构是栈，所以正确做法是先pop，再add，但是转场动画会有覆盖的不正常现象，你需要特殊处理，不然会闪屏！
- (2) Fragment的根布局要设置 `android:clickable = true`，原因是在pop后又立刻add新的Fragment时，在转场动画过程中，如果你的手速太快，在动画结束前你多点击了一下，上一个Fragment的可点击区域可能会在下一个Fragment上依然可用。

总结

看了上面的介绍，你可能会觉得Fragment有点可怕。

但是我想说，如果你只是浅度使用，比如一个Activity容器包含列表Fragment + 详情Fragment这种简单情景下，不涉及到 `popBackStack/Immediate(tag/id)` 这些的方法，还是比较轻松使用的，出现的问题，网上都可以找到解决方案。

但是如果你的Fragment逻辑比较复杂，有特殊需求，或者你的app架构是仅有一个Activity + 多个Fragment，上面说的这些坑，你都应该全部解决。

在[下一篇](#)中，介绍了一些非常实用的使用技巧，包括如何解决Fragment嵌套、各种环境、组件下Fragment的使用等技巧，推荐阅读！

还有一些比较隐蔽的问题，不影响app的正常运行，仅仅是一些显示的BUG，并没有在上面介绍，在本系列的[最后一篇](#)，我给出了我的解决方案，一个我封装的[Fragmentation](#)库，解决了所有动画问题，非常适合**单Activity+多Fragment** 或者 **多模块Activity + 多Fragment**的架构。有兴趣的可以看看 :)

Fragment全解析系列（二）：正确的使用姿势

来源:www.jianshu.com

- 1、[Fragment全解析系列（一）：那些年踩过的坑](#)
- 2、[Fragment全解析系列（二）：正确的使用姿势](#)
- 3、[Fragment之我的解决方案：Fragmentation](#)

本篇主要介绍一些Fragment使用技巧。

Fragment是可以让你的app纵享丝滑的设计，如果你的app想在现在基础上**性能大幅度提高，并且占用内存降低**，同样的界面Activity占用内存比Fragment要多，响应速度Fragment比Activty在中低端手机上快了很多，甚至能达到好几倍！如果你的app当前或以后有移植平板等平台时，可以让你节省大量时间和精力。

简陋的目录

- 1、一些使用建议
- 2、add(), show(), hide(), replace()的那点事
- 3、关于FragmentManager你需要知道的
- 4、使用ViewPager+Fragment的注意事项
- 5、Fragment事务，你可能不知道的坑
- 6、是使用单Activity + 多Fragment的架构，还是多模块Activity + 多Fragment的架构？

作为一个稳定的app，从后台且回到前台，一定会在任何情况都能恢复到离开前的页面，并且保证数据的完整性。

如果你没看过本系列的[第一篇](#)，为了方便后面文章的介绍，先规定一个“术语”，安卓app有一种特殊情况，就是 app运行在后台的时候，系统资源紧张的时候导致把app的资源全部回收（杀死app的进程），这时把app再从后台返回到前台时，app会重启。这种情况下文简称为：“内存重启”。

一些使用建议

- 1、对Fragment传递数据，建议使用 `setArguments(Bundle args)`，而后在`onCreate`中使用 `getArguments()`取出，在“内存重启”前，系统会帮你保存数据，不会造成数据的丢失。和Activity的Intent原理一致。
- 2、使用`newInstance(参数)`创建Fragment对象，优点是调用者只需要关心传递的哪些数据，而无需关心传递数据的Key是什么。
- 3、如果你需要在Fragment中用到宿主Activity对象，建议在你的基类Fragment定义一个Activity的全局变量，在 `onAttach` 中初始化。原因参考第一篇的“`getActivity()`空指针”部分，在`onCreateView()` 内出现`getActivity()`的代码 很可能是危险的。

```
protected Activity mActivity;
@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    this.mActivity = activity;
}
```

add(), show(), hide(), replace()的那点事

- 1、区别

`show()`，`hide()` 最终是让Fragment的View `setVisibility (true还是false)`，不会调用生命周期；`replace()`的话会销毁视图，即调用`onDestoryView`、`onCreateView`等一系列生命周期；

`add()` 和 `replace()` 不要在同一个阶级的FragmentManager里混搭使用。

- 2、使用场景

如果你有一个很高的概率会再次使用当前的Fragment，建议使用 `show()`，`hide()`，可以提高性能。

在我使用Fragment过程中，大部分情况下都是用`show()`，`hide()`，而不是`replace()`。

- 3、`onHiddenChanged`的回调时机

当使用`add() + show()`，`hide()`跳转新的Fragment时，旧的Fragment回调`onHiddenChanged()`，不会回调`onStop()`等生命周期方法，而新的Fragment在创建时是不会回调`onHiddenChanged()`，这点要切记。

- 4、Fragment重叠问题

使用show(), hide()带来的一个问题就是，如果你不做任何处理，在“内存重启”后，Fragment会重叠；

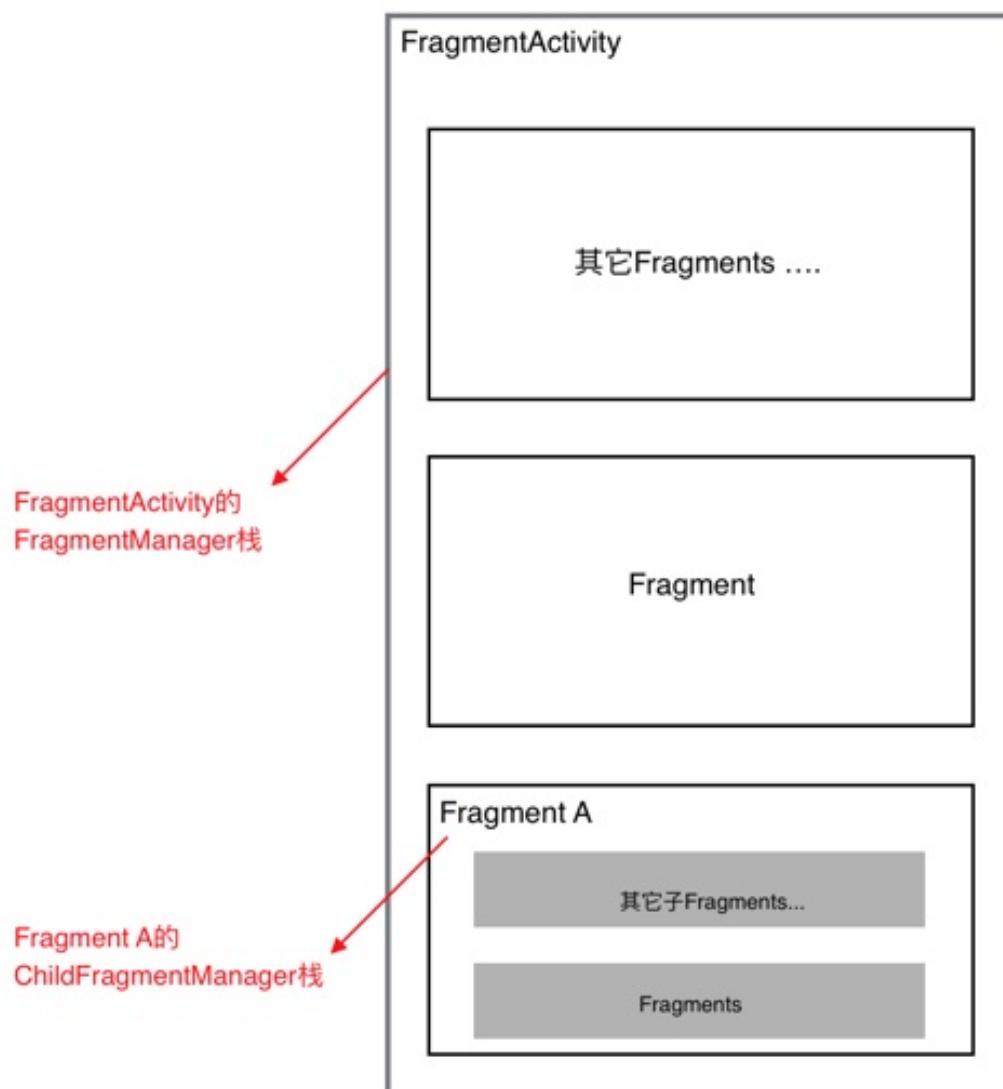
有些小伙伴可能就是为了避免Fragment重叠问题，而选择使用replace()，但是使用show(), hide()时，重叠问题是完全可以解决的，有两种方式解决，详情参考上一篇。

关于FragmentManager你需要知道的

- 1、FragmentManager栈视图

(1) 每个Fragment以及宿主Activity(继承自FragmentActivity)都会在创建时，初始化一个FragmentManager对象，处理好Fragment嵌套问题的关键，就是理清这些不同阶级的栈视图。

下面给出一个简要的关系图：



(2) 对于宿主Activity, `getSupportFragmentManager()`获取的`FragmentActivity`的`FragmentManager`对象;

对于`Fragment`, `getFragmentManager()`是获取的是父`Fragment`(如果没有, 则是`FragmentActivity`)的`FragmentManager`对象, 而`getChildFragmentManager()`是获取自己的`FragmentManager`对象。

- 2、恢复`Fragment`时 (同时防止`Fragment`重叠), 选择`getFragments()`还是`findFragmentByTag()`

(1) 选择`getFragments()`

对于一个Activity内的多个`Fragment`, 如果`Fragment`的关系是“流程”, 比如登录->注册/忘记密码->填写信息->跳转到主页Activity。这种情况下, 用`getFragments()`的方式是最合适的, 在你的Activity内 (更好的方式是在你的所有“流程”基类Activity里), 写下如下代码:

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (savedInstanceState != null) {
        List<Fragment> fragments = getSupportFragmentManager().getFragments();

        if (fragments != null && fragments.size() > 0) {
            boolean showFlag = false;

            FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
            for (int i = fragments.size() - 1; i >= 0; i--) {
                Fragment fragment = fragments.get(i);
                if (fragment != null) {
                    if (!showFlag) {
                        ft.show(fragments.get(i));
                        showFlag = true;
                    } else {
                        ft.hide(fragments.get(i));
                    }
                }
            }
            ft.commit();
        }
    }
}
```

(2) 选择`findFragmentByTag()`恢复

如果你的Activity的Fragments，不是“流程”关系，而是“同级”关系，比如QQ的主界面，“消息”、“联系人”、“动态”，这3个Fragment属于同级关系，用上面的代码就不合适了，恢复的时候总会恢复最后一个，即“动态Fragment”。

正确的做法是在onSaveInstanceState()内保存当前所在Fragment的tag或者下标，在onCreate()是恢复的时候，隐藏其它2个Fragment。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity);

    MsgFragment msgFragment;
    ContactFragment contactFragment;
    MeFragment meFragment;

    if (savedInstanceState != null) { // “内存重启”时调用
        msgFragment = getSupportFragmentManager().findFragmentByTag(msgFragment.getClass().getName());
        contactFragment = getSupportFragmentManager().findFragmentByTag(contactFragment.getClass().getName());
        meFragment = getSupportFragmentManager().findFragmentByTag(meFragment.getClass().getName());

        index = savedInstanceState.getInt(KEY_INDEX);
        // 根据下标判断离开前是显示哪个Fragment,
        // 这里省略判断代码, 假设离开前是ConactFragment
        // 解决重叠问题
        getFragmentManager().beginTransaction()
            .show(contactFragment)
            .hide(msgFragment)
            .hide(meFragment)
            .commit();
    }else{ // 正常时
        msgFragment = MsgFragment.newInstance();
        contactFragment = ContactFragment.newInstance();
        meFragment = MeFragment.newInstance();

        getFragmentManager().beginTransaction()
            .add(R.id.container, msgFragment, msgFragment.getClass().getName())
            .add(R.id.container, contactFragment, contactFragment.getClass().getName())
            .add(R.id.container, meFragment, meFragment.getClass().getName())
            .hide(contactFragment)
            .hide(meFragment)
            .commit();
    }
}

@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // 保存当前Fragment的下标
    outState.putInt(KEY_INDEX, index);
}
```

当然在“同级”关系中，使用getFragments()恢复也是可以的。

使用ViewPager+Fragment的注意事项

- 1、使用ViewPager + Fragment时，切换不同ViewPager页面，不会回调任何生命周期方法以及onHiddenChanged()，只有setUserVisibleHint(boolean isVisibleToUser)会被回调，所以如果你想进行一些懒加载，需要在这里处理。
- 2、在给ViewPager绑定FragmentPagerAdapter时，new FragmentPagerAdapter(fragmentManager)的FragmentManager，一定要保证正确，如果ViewPager是Activity内的控件，则传递getSupportFragmentManager()，如果是Fragment的控件中，则应该传递getChildFragmentManager()。只要记住ViewPager内的Fragments是当前组件的子Fragment这个原则即可。
- 3、如果使用ViewPager+Fragment，不需要在“内存重启”的情况下，去恢复的Fragments，有FragmentPagerAdapter的存在，不需要你去做恢复工作。

Fragment事务，你可能不知道的坑

- 1、如果你在使用popBackStackImmediate()方法后，紧接着直接调用类似如下事务的方法，因为他们运行在消息队列的问题，还没来得及出栈就运行事务的方法了，这可能会导致不正常现象。

```
getSupportFragmentManager().popBackStackImmediate();
getSupportFragmentManager().beginTransaction()
    .add(R.id.container, fragment, tag)
    .hide(currentFragment)
    .commit;
```

正确的做法是使用主线程的Handler，将事务放到Runnable里运行。

```
getSupportFragmentManager().popBackStackImmediate();
new Handler().post(new Runnable(){
    @Override
    public void run() {
        // 在这里执行Fragment事务
    }
});
```

- 2、给Fragment设定Fragment转场动画时，如果你没有一整套解决方案，应避免使用.setTransition(transit) 以及.setCustomAnimations(enter, exit, popEnter, popExit)，而只使用.setCustomAnimations(enter, exit) 这个方法。

其它2个方法会在某种情况下的“内存重启”中会出现BUG。

本系列最后一篇给出了我的解决方案，解决了该问题，有兴趣可以自行查看 :)

另外一提：谨慎使用 `popStackBack(String tag/int id, int flag)` 系列的方法，原因在[上一篇](#)中已经描述。

是使用单Activity + 多Fragment的架构，还是多模块Activity + 多Fragment的架构？

- 单Activity + 多Fragment：

一个app仅有一个Activity，界面皆是Fragment，Activity作为app容器使用。

优点：性能高，速度最快。参考：新版知乎、google系app

缺点：逻辑比较复杂，尤其当Fragment之间联动较多或者嵌套较深时，比较复杂。

- 多模块Activity + 多Fragment：

一个模块用一个Activity，比如

1、登录注册流程：LoginActivity + 登录Fragment + 注册Fragment + 填写信息Fragment + 忘记密码Fragment

2、或者常见的数据展示流程：DataActivity + 数据列表Fragment + 数据详情Fragment + ...

优点：速度快，相比较单Activity+多Fragment，更易维护。

我的观点：

权衡利弊，我认为多模块Activity + 多Fragment是最合适的架构，开发起来不是很复杂，app的性能又很高效。

当然。Fragment只是官方提供的灵活组件，请优先遵从你的项目设计！真的特别复杂的界面，或者单个Activity就可以完成一个流程的界面，使用Activity可能是更好的方案。

最后

如果你读完了[第一篇](#)和这篇文章，那么我相信你使用多模块Activity+多Fragment的架构所遇到的坑，大部分都应该能找到解决办法。

但是如果流程较为复杂，比如Fragment A需要启动一个新的Fragment B并且关闭当前A，或者A启动B，B在获取数据后，想在返回到A时把数据交给A（类似Activity的startActivityForResult），又或者你保证在Fragment转场动画的情况下，使用pop(tag\id)从栈内退出多个Fragment，或者你甚至想Fragment有一个类似Activity的SingleTask启动模式，那么你可以参考[下一篇](#)，我的解决方案库，[Fragmentation](#)。它甚至提供了一个让你在开发时，可以随时查看所有阶级的栈视图的UI界面。

文 / YoKey（简书作者） 原文链接：<http://www.jianshu.com/p/fd71d65f0ec6> 著作权归作者所有，转载请联系作者获得授权，并标注“简书作者”。

View

Android View中 getViewTreeObserver().addOnGlobalLayoutListener()

来源:http://blog.csdn.net/lingshu_java/article/details/46544811

我们知道在 `onCreate` 中 `View.getWidth` 和 `View.getHeight` 无法获得一个view的高度和宽度，这是因为View组件布局要在 `onResume` 回调后完成。所以现在需要使用 `getViewTreeObserver().addOnGlobalLayoutListener()` 来获得宽度或者高度。这是获得一个view的宽度和高度的方法之一。

`OnGlobalLayoutListener` 是 `ViewTreeObserver` 的内部类，当一个视图树的布局发生改变时，可以被 `ViewTreeObserver` 监听到，这是一个注册监听视图树的观察者(observer)，在视图树的全局事件改变时得到通知。`ViewTreeObserver` 不能直接实例化，而是通过 `getViewTreeObserver()` 获得。

除了 `OnGlobalLayoutListener`，`ViewTreeObserver` 还有如下内部类：

- `interface ViewTreeObserver.OnGlobalFocusChangeListener`

当在一个视图树中的焦点状态发生改变时，所要调用的回调函数的接口类

- `interface ViewTreeObserver.OnGlobalLayoutListener*`

当在一个视图树中全局布局发生改变或者视图树中的某个视图的可视状态发生改变时，所要调用的回调函数的接口类

- `interface ViewTreeObserver.OnPreDrawListener`

当一个视图树将要绘制时，所要调用的回调函数的接口类

- `interface ViewTreeObserver.OnScrollChangedListener`

当一个视图树中的一些组件发生滚动时，所要调用的回调函数的接口类

- `interface ViewTreeObserver.OnTouchModeChangeListener`

当一个视图树的触摸模式发生改变时，所要调用的回调函数的接口类

其中，我们可以利用 `OnGlobalLayoutListener` 来获得一个视图的真实高度。

```

int mHeaderViewHeight;
mHeaderView.getViewTreeObserver().addOnGlobalLayoutListener(
    new OnGlobalLayoutListener() {
        @Override
        public void onGlobalLayout() {

            mHeaderViewHeight = mHeaderView.getHeight();
            getViewTreeObserver()
                .removeGlobalOnLayoutListener(this);
        }
    });

```

但是需要注意的是 `onGlobalLayoutListener` 可能会被多次触发，因此在得到了高度之后，要将 `OnGlobalLayoutListener` 注销掉。

有时候需要在 `onCreate` 方法中知道某个View组件的宽度和高度等信息，而直接调用View组件

的 `getWidth()`、`getHeight()`、`getMeasuredWidth()`、`getMeasuredHeight()`、`getTop()`、`getLeft()` 等方法是无法获取到真实值的，只会得到0。这是因为View组件布局要在 `onResume` 回调后完成。下面提供实现方法，`onGlobalLayout` 回调会在view布局完成时自动调用：

类似：

```

// This listener is used to get the final width of the GridView and then calculate the
// number of columns and the width of each column. The width of each column is variable
// as the GridView has stretchMode=columnWidth. The column width is used to set the height
// of each view so we get nice square thumbnails.
mGridView.getViewTreeObserver().addOnGlobalLayoutListener( //view 布局完成时调用, 每次view
    new ViewTreeObserver.OnGlobalLayoutListener() {
        @Override
        public void onGlobalLayout() {
            if (mAdapter.getNumColumns() == 0) {
                final int numColumns = (int) Math.floor(
                    mGridView.getWidth() / (mImageThumbSize + mImageThumbSpacing));
                if (numColumns > 0) {
                    final int columnWidth =
                        (mGridView.getWidth() / numColumns) - mImageThumbSpacing;
                    mAdapter.setNumColumns(numColumns); //设置 列数
                    mAdapter.setItemHeight(columnWidth); //设置 高度
                }
            }
        }
    });

```

在gridview布局完成后，根据gridview的宽和高设置adapter列数和每个item高度

你造么,Android中程序的停止状态

来源:[你造么,Android中程序的停止状态 - 技术黑屋](#)

很多人遇到过广播收不到的问题,比如Google Play推广安装广播没有收到等,诸如这些问题,又都是什么原因呢,这篇文章将进行回答.

从Android 3.1(HoneyComb) 也就是API 12开始,Android引入了一套新的启动控制,这就是程序的停止状态.那让我们看一下Google对于程序的停止状态的描述.

什么是程序的停止状态

Starting from Android 3.1, the system's package manager keeps track of applications that are in a stopped state and provides a means of controlling their launch from background processes and other applications.

从Android 3.1开始,系统的包管理器开始跟踪处理停止状态的程序.并且提供了方法来控制从后台进程或者其他程序对它们的启动.

Note that an application's stopped state is not the same as an Activity's stopped state. The system manages those two stopped states separately.

注意 程序的停止状态和Activity的停止状态不同,系统会单独处理这两种状态.

The platform defines two new intent flags that let a sender specify whether the Intent should be allowed to activate components in stopped application.

Android平台提供了两个intent flags,用来让发送广播的一方决定广播是否需要同时发送给已经停止的程序.

FLAG_INCLUDE_STOPPED_PACKAGES — Include intent filters of stopped applications in the list of potential targets to resolve against. 将已经支持的程序加入到能处理intent的目标处理器.

FLAG_EXCLUDE_STOPPED_PACKAGES — Exclude intent filters of stopped applications from the list of potential targets. 在能处理intent的目标处理器中不包含已经停止的程序.

当如果intnet中没有或者设置了上面两个flag,在目标处理器中是包含已经处于停止的程序.但是注意,系统会为所有的广播intent增加 **FLAG_EXCLUDE_STOPPED_PACKAGES** 这个flag.

为什么Android要引入这一状态

Note that the system adds FLAG_EXCLUDE_STOPPED_PACKAGES to all broadcast intents. It does this to prevent broadcasts from background services from inadvertently or unnecessarily launching components of stopped applications. A background service or application can override this behavior by adding the FLAG_INCLUDE_STOPPED_PACKAGES flag to broadcast intents that should be allowed to activate stopped applications.

需要注意的是,系统会默认地对所有的广播intent增加一个 FLAG_EXCLUDE_STOPPED_PACKAGES 的flag,这样做的目的是为了阻止来自后台服务的广播不慎或者启动处于停止状态的程序的不必要的组件.

通常的intnet广播,处于停止状态的程序的receiver是无法接受到的.那么怎么才能让这些停止状态的程序接受到呢?可以这样做,在后台服务或者应用中发送广播时,增加一个 FLAG_INCLUDE_STOPPED_PACKAGES 的flag,意思是包含处于停止状态的程序.这样就可以激活停止状态的程序.

正如上述引用指出,系统默认阻止广播intent发送给处于停止状态的程序包,实际上这是为了保证安全和省电需要.比如说网络变化的广播,如果某些程序注册监听,并且它在得到广播时,做一系列的网络操作,这样必然是很耗能源的.

激活状态和停止状态的切换

当程序第一次安装并且没有启动,或者用户手动从程序管理将其停止后,程序都会处于停止状态.

如何变为停止状态

- 在设置应用管理中的应用详情页点击强制停止
- 使用 adb shell adb shell am force-stop package-name
- 使用ActivityManager的隐藏方法 forceStopPackages ,并且向 manifest 加入申请权限 <uses-permission android:name="android.permission.FORCE_STOP_PACKAGES"/>

如何脱离停止状态

- 手动启动程序
- 使用adb激活应用组件,如activity或者receiver

发送广播intent给处于停止状态的应用

- 在Java代码发送Intent时,加入flag FLAG_INCLUDE_STOPPED_PACKAGES
- 如果使用adb,同样是加入 FLAG_INCLUDE_STOPPED_PACKAGES (其具体值为32),如 adb shell am broadcast -a com.android.vending.INSTALL_REFERRER -f 32

检查是否处于停止状态

- 进入设置—应用管理—某个应用的详细页,如果强制停止按钮不可用,则说明程序已经处于停止状态.
- 进入设备终端,查看系统文件cat /data/system/packages-stopped.xml

问答环节

提问:如果我的程序没有activity只有一个receiver,我该如何激活才能接收到正常的广播intent呢 **回答:**实际上,如果是上面所述的情况,该应用在安装之后不是处于停止状态,因为它没有任何用户可以直接点击的行为去将它移除停止状态.你可以正常接收广播intent,除非你人为地将它强制停止.

提问:系统的程序刚安装会处于停止状态么? **回答:**系统的程序通常会存放在 /system/app目录下,在一开始安装之后不会处于停止状态.

提问:Google Play的推广广播据说是在程序安装完成之后发送,是不是3.1之后受影响么 **回答:**不受影响的.Google文档说INSTALL_REFERRER会在程序安装完成之后发送,据实际查看日志观察,从3.1之后,是在程序安装后第一次打开时发送.

动画

一种新的Activity转换动画实现方式

来源:codethink.me

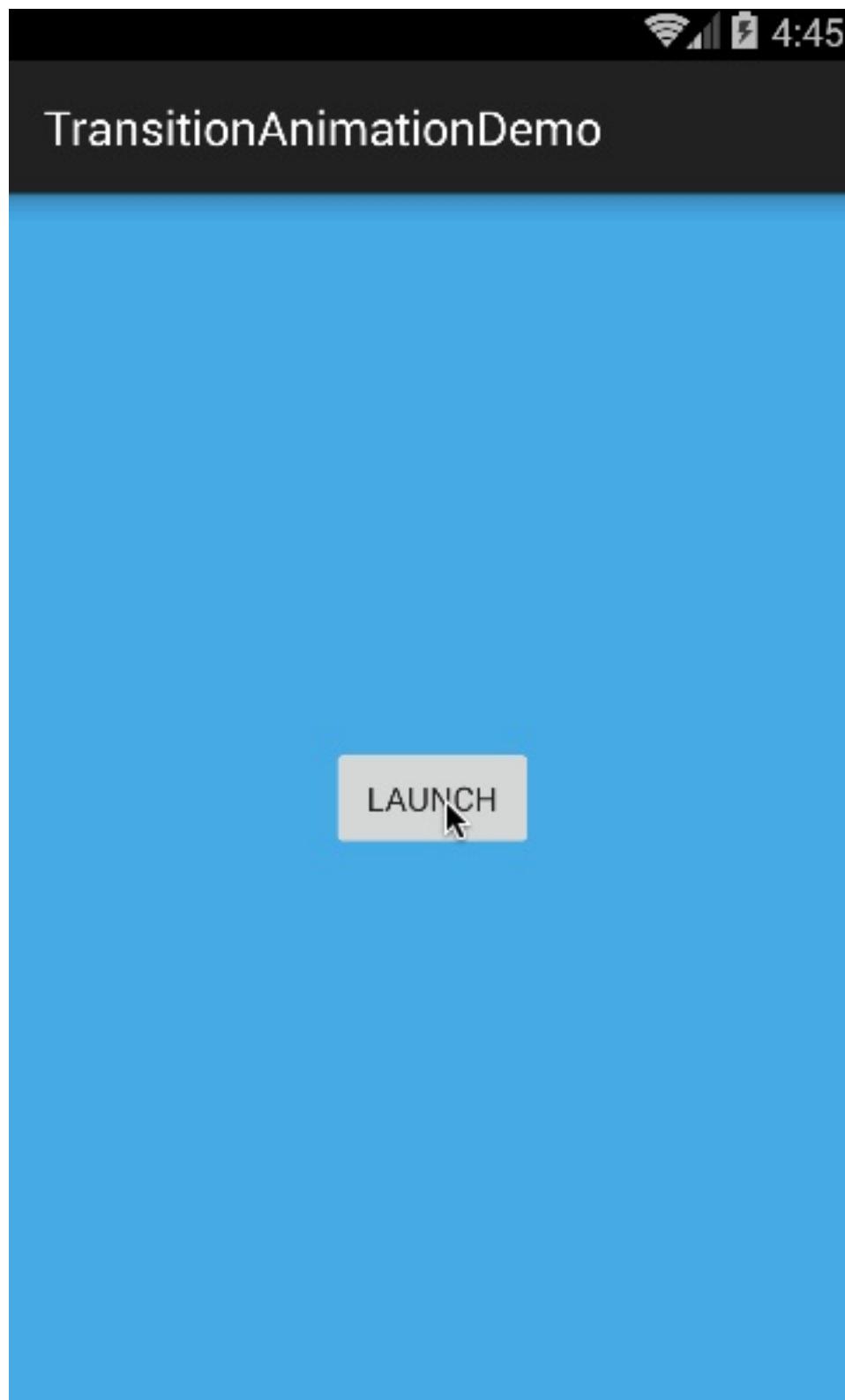
0. 前言

为Android中基本的View组建Activity设置转换动画的方式一般有两种：通过overridePendingTransitions设置，以及使用TransitionManager实现。

overridePendingTransitions只能使用XML来设置Activity的进入和退出动画，局限性很大。而使用TransitionManager只兼容API level 19及以上的设备。最近在[InstaMaterial concept](#)中发现其利用addOnPreDrawListener方法，提供了一种新的Activity转换动画实现方式，这里详细记录下这种基于addOnPreDrawListener()的实现方式。

1. 实现展开动画

首先创建一个基本的Activity转换场景，去掉默认的转换动画，没有任何动画的Activity转换效果如下。



修改第二个Activity的布局activity_second，设置最顶层的布局id为root。之后，在SecondActivity的onCreate中，为root设置动画，核心部分代码如下：

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_second);

    rootView = findViewById(R.id.root);

    if (savedInstanceState == null) {
        rootView.getViewTreeObserver().addOnPreDrawListener(
            new ViewTreeObserver.OnPreDrawListener() {
                @Override
                public boolean onPreDraw() {
                    rootView.getViewTreeObserver().removeOnPreDrawListener(this);
                    startRootAnimation();
                    return true;
                }
            });
    }
}

```

这里需要注意的是：

- 1) 只需要在首次创建时执行动画，因此需要满足条件`savedInstanceState == null`；
- 2) 在`onPreDraw`中要首先移除`OnPreDrawListener`，否则在动画过程中会多次调用，导致死循环。

最后，`startRootAnimation`的实现如下：

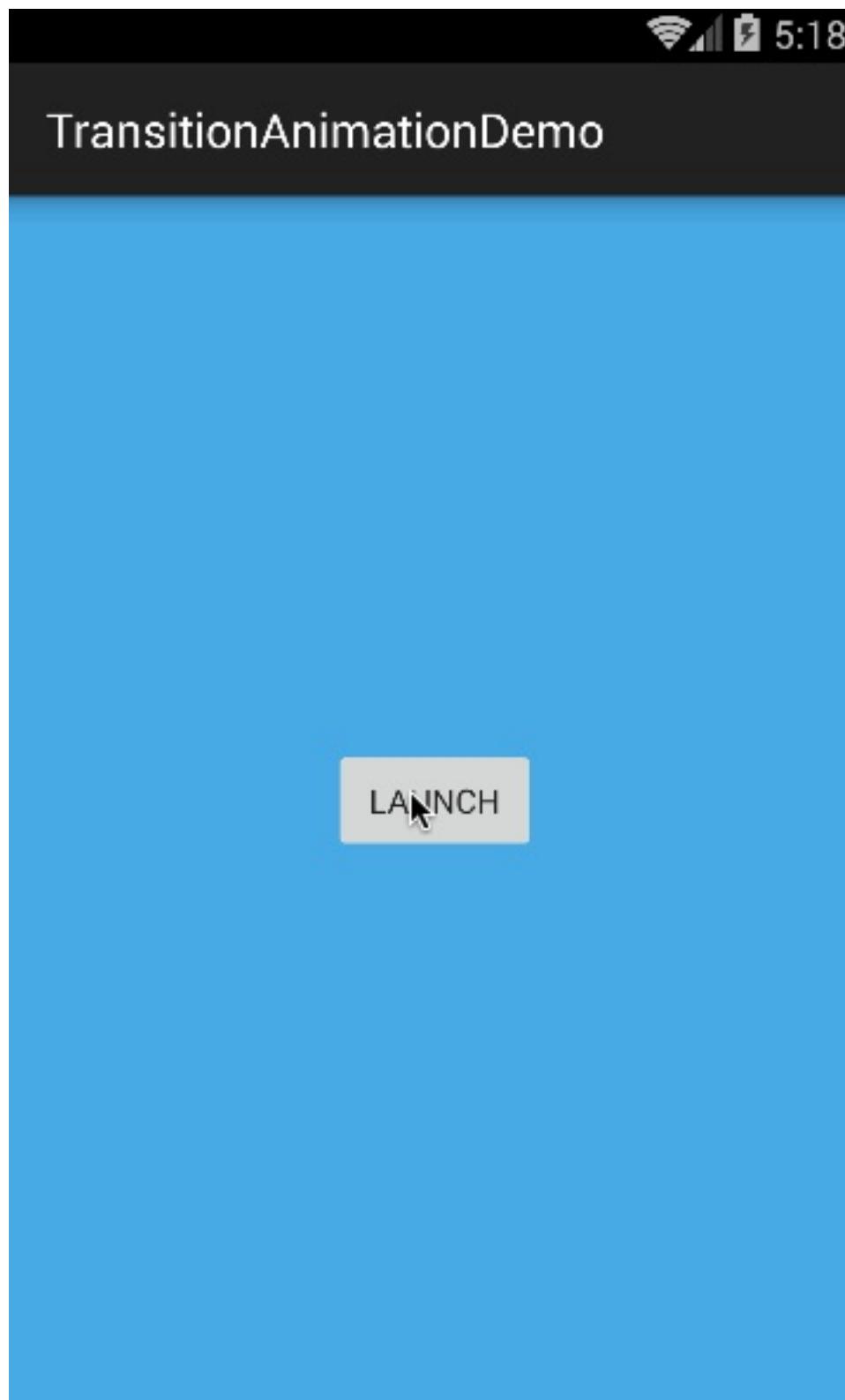
```

private void startRootAnimation() {
    rootView.setScaleY(0.1f);
    rootView.setPivotY(rootView.getY() + rootView.getHeight() / 2);

    rootView.animate()
        .scaleY(1)
        .setDuration(1000)
        .setInterpolator(new AccelerateInterpolator())
        .start();
}

```

此时的动画效果如下：



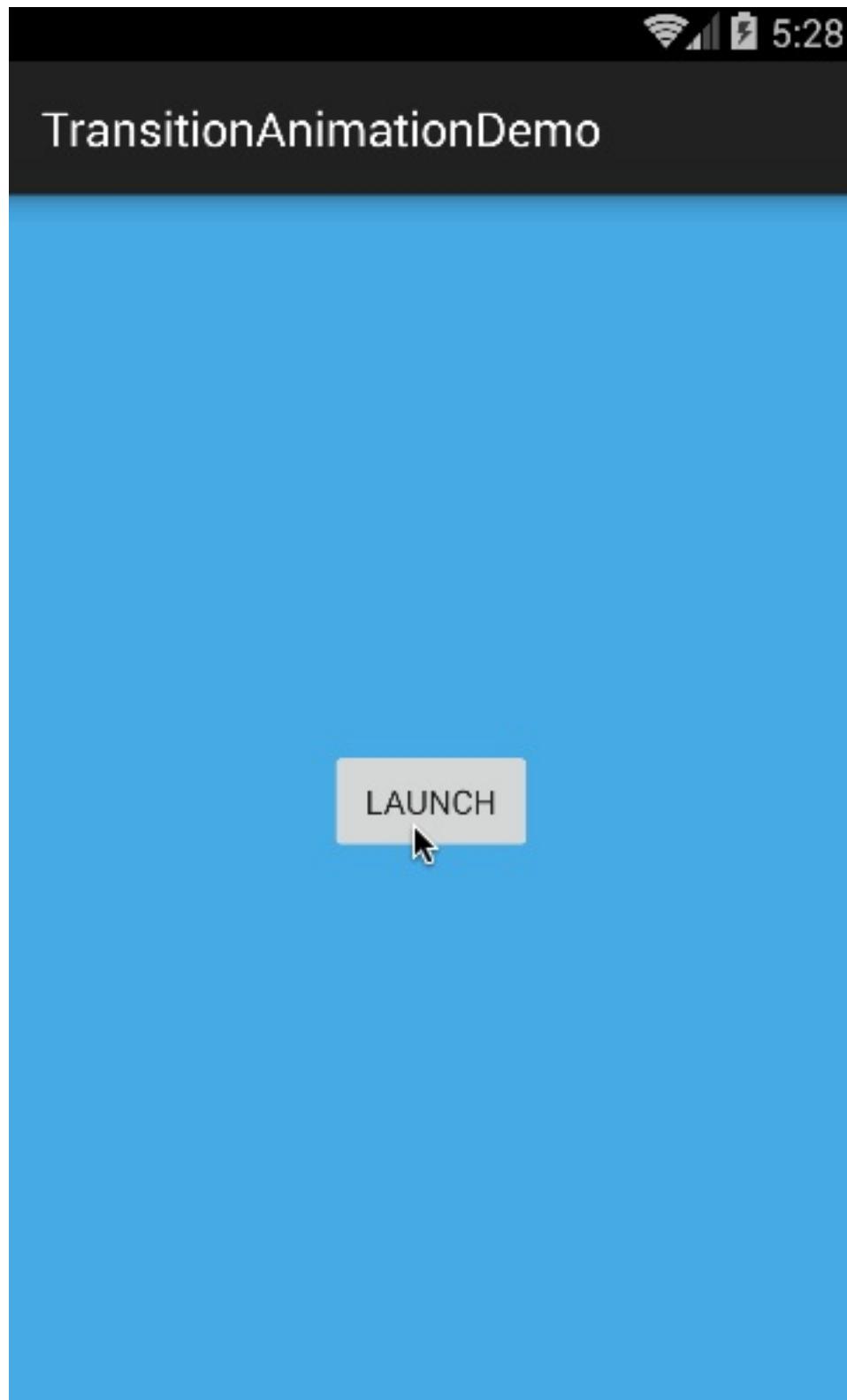
2. 设置Activity透明背景

上面的动画还有一个问题：第二个Activity展开的时候，它的背景不是第一个Activity，而是白色背景，这是因为默认的主题为每个Activity设置了白色作为窗口的背景，因此需要在style中创建一个背景为透明的主题，并在AndroidManifest.xml中设置SecondActivty的主题。

题为透明背景主题。透明背景主题代码如下：

```
<style name="AppTheme.TransparentActivity">
    <item name="android:windowBackground">@android:color/transparent</item>
    <item name="android:windowIsTranslucent">true</item>
    <item name="android:windowAnimationStyle">@null</item>
</style>
```

最后的效果如下：



3. 小结

至此实现了一个简单的基于`addOnPreDrawListener`的转换动画，这种转换动画相对于`overridePendingTransitions`更为灵活，提供了更多想象空间，同时相比于`TransitionManager`有更好的兼容性。

这种方式目前存在的问题是需要为顶级view以及各层子view分别设置动画，使得顶级view和子view同时展开，或子view延后展开（否则会出现子view已经绘制到目标位置，顶级view仍然在执行动画的情况），对于复杂的布局而言实现有些繁琐。

评论内容

参考链接：<http://frogermcs.github.io/Instagram-with-Material-Design-concept-is-getting-real/>

不需要rootview，直接

```
rootView = ((ViewGroup) findViewById(android.R.id.content)).getChildAt(0);
```

实现Activity间的共享控件转场动画

来源:www.jianshu.com

Lollipop中有 `shared_element` 可以进行元素在activity之间进行共享，网上已经有很多介绍了，然而目前还有大量的kitkat设备，所以说啊，兼容更重要。

如下的方法，可以实现在旧的手机上实现动画效果。采用了类似于豌豆荚的 `开眼` 项目使用的技术。github上可能有在5.0以下的兼容包，但是个人并不推荐使用第三方的UI工具。



实现原理

最近逛业界良心酷安网，发现了豌豆荚的一款叫做[开眼](#)的项目，这个项目的意义就是每天把墙外的东西搬运回来让村里的阿Q们开开眼，这款app至少目前看还没有那么毒瘤，还是比较小而美的，于是下载试用了一下。

有个转场动画使用了类似上面gif的效果，第一感觉是自定义了一个 `popwindow`，于是起床，打开电脑，对两个效果的始末进行查询

```
adb shell dumpsys window windows | grep -E 'mCurrentFocus'
```

结果如下，可以发现是2个activity，而不是自定义view实现的

```
mCurrentFocus=xx/xxxx.ui.activity.FeedActivity}
mCurrentFocus=xx/xxxx.ui.activity.DetailActivity}
```

反复研究，最后发现原理如下

- FeedActivity将view的 `top/width/height`，内容等信息通过intent进行发送
- DetailActivity设置为背景 透明模式，转场动画关闭，这里关闭是非常重要的
- DetailActivity接着读取 `intent`，并根据高度等信息进行动画绘制，由于 `DetailActivity` 的背景是透明的，所以用户会误认为是进行了“放大”操作

在UED中，“放大”与“右转”是两种常见的场景切换操作，安利一本叫做《[Learn iOS Design](#)》的书，Android开发者也值得借鉴一下

步骤

步骤非常简单，主要时间是耗在了调试动画上

1. 设置DetailActivity的主题

```
<style name="DetailedTheme" parent="AppTheme">
    <!-- 背景透明 -->
    <item name="android:windowIsTranslucent">true</item>
    <item name="android:windowBackground">@android:color/transparent</item>
    <!-- 无进入动画 -->
    <item name="android:windowAnimationStyle">@null</item>
</style>
```

2. 获取FeedActivity中的itemview

楼主在RecyclerView的Adapter中的viewholder中手动加了一个接口，在主界面实现。这里使用的是ObjectAnimator，它在动画完成后，可以保存动画结束的布局，而且最后测试机(1080P/2G/4.4)的渲染时间均在警戒线(60fps)的一半左右，性能可以满足。

```
@Override public void onItemClick(View v, int position) {
    Parcelable imgInfo = ((CardAdapter) mRvFragCard.getAdapter()).getData().get(position);
    //封装了width/top等信息
    Position viewPosition = Position.from(v);
    DetailedActivity.startActivity(v.getContext(), viewPosition, imgInfo);
}
```

3. 在DetailActivity中进行动画

动画没有什么技巧，跟老司机开车一样，属于熟练工种，我们这里主要是使用了AnimationSet进行并发动画

```
void anim(final Position position, final boolean in,
          final Animator.AnimatorListener listener, View... views) {
    if (isPlaying) {
        return;
    }
    //记住括号哦，我这里调试了一小时
    float delta = ((float) (position.width)) / ((float) (position.height));

    float[] y_img = { position.top - views[0].getTop(), 0 };
    float[] s_img = { 1f, delta };

    float[] y_icn = { views[1].getHeight() * 4, 0 };
    float[] s_icn = { 3f, 1f };

    views[0].setPivotX(views[0].getWidth() / 2);
    views[1].setPivotX(views[1].getWidth() / 2);

    Animator trans_Y =
```

```
        ObjectAnimator.ofFloat(view[0], View.TRANSLATION_Y, in ? y_img[0] : y_img[1],
                               in ? y_img[1] : y_img[0]));
    Animator scale_X =
        ObjectAnimator.ofFloat(view[0], View.SCALE_X, in ? s_img[0] : s_img[1],
                               in ? s_img[1] : s_img[0]);
    Animator scale_Y =
        ObjectAnimator.ofFloat(view[0], View.SCALE_Y, in ? s_img[0] : s_img[1],
                               in ? s_img[1] : s_img[0]);
    Animator scale_icn_X =
        ObjectAnimator.ofFloat(view[1], View.SCALE_X, in ? s_icn[0] : s_icn[1],
                               in ? s_icn[1] : s_icn[0]);
    Animator scale_icn_Y =
        ObjectAnimator.ofFloat(view[1], View.SCALE_Y, in ? s_icn[0] : s_icn[1],
                               in ? s_icn[1] : s_icn[0]);

    Animator trans_icn_Y =
        ObjectAnimator.ofFloat(view[1], View.TRANSLATION_Y, in ? y_icn[0] : y_icn[1],
                               in ? y_icn[1] : y_icn[0]));

    AnimatorSet set = new AnimatorSet();

    set.playTogether(trans_Y, scale_X, scale_Y);
    set.playTogether(scale_icn_X, scale_icn_Y, trans_icn_Y);
    set.setDuration(400);
    set.addListener(new Animator.AnimatorListener() {
        @Override public void onAnimationStart(Animator animation) {
            listener.onAnimationStart(animation);
            isPlaying = true;
        }

        @Override public void onAnimationEnd(Animator animation) {
            listener.onAnimationEnd(animation);
            isPlaying = false;
        }

        @Override public void onAnimationCancel(Animator animation) {
            listener.onAnimationCancel(animation);
            isPlaying = false;
        }

        @Override public void onAnimationRepeat(Animator animation) {
            listener.onAnimationRepeat(animation);
        }
    });
    set.setInterpolator(new AccelerateInterpolator());
    set.start();
}
```

这样，动画就搞定了，是不是很简单？

总结

源码仍在更新中，后期将加入大量仿iOS的组件

<https://github.com/miao1007/AnimeWallpaper>

最后，总结一下，这个东西实现不难，但是

- 调试动画太费时间了，暗坑多
- 属性`android:windowIsTranslucent`暗坑多，大部分手机运行状态未知
- 编码改动量大，回调接口多，很难把所有业务封装在一个文件中，源码乱，后期维护困难

所以除了情怀，很少有专门的人愿意这样写，我建议在个人app中使用，或者提起商量好了再合作编码。另外关于状态栏适配，可以[参考这里](#)

谢谢大家的观看，如果觉得本文有意义的话，不妨点个赞或者分享吧！

动画

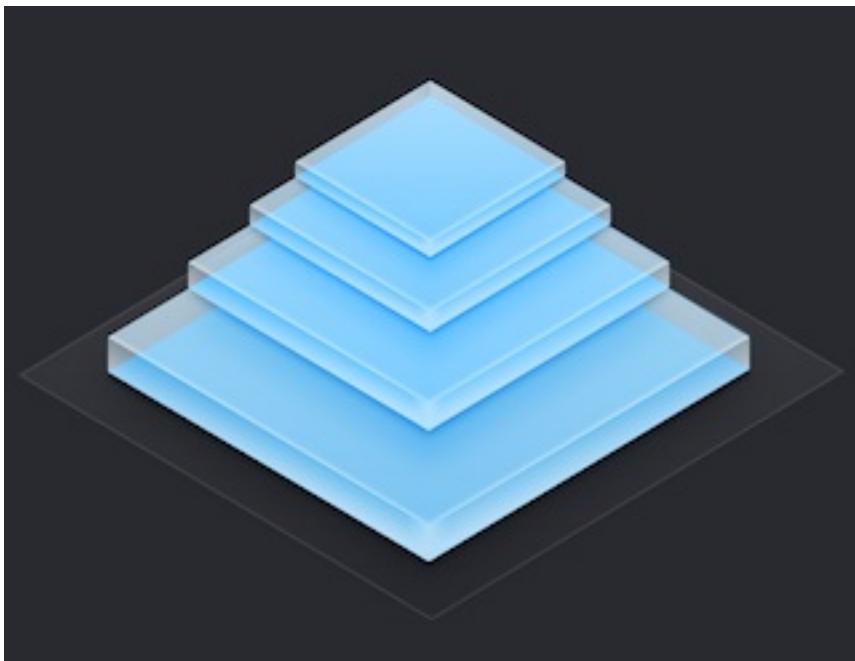
常识

移动开发需要知道的像素知识『多图』

来源:weizhifeng.net

作者: JeremyWei | 可以转载, 但必须以超链接形式标明文章原始出处和作者信息及
版权声明

网址: <http://weizhifeng.net/you-should-know-about-dpi.html>

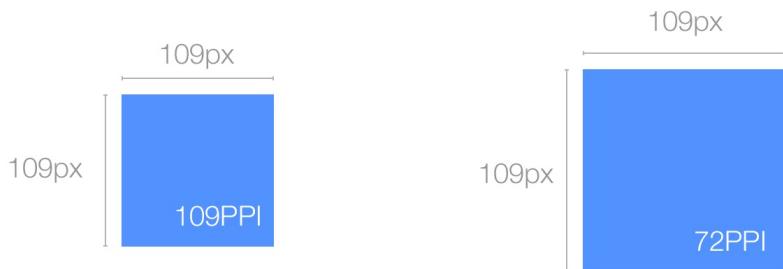


像素 (Pixel) 对于WEB开发者来说很是熟悉, 在PC互联网时代没少与其打交道。进入移动互联网之后, 随着移动设备屏幕的解析度越来越高, 衍生了一些关于屏幕和像素的一些新概念, 比如DPI, DP, PT, Retina, 4K等等, 本文对这些概念做一个简单的介绍。

DPI与PPI

DPI (Dots Per Inch) 是印刷行业中用来度量空间点密度用的, 这个值是打印机每英寸可以喷的墨汁点数。计算机显示设备从打印机中借鉴了DPI的概念, 由于计算机显示设备中的原子单位不是墨汁点而是像素, 所以就创造了PPI (Pixels Per Inch), 这个值是屏幕每英寸的像素数量, 即像素密度 (Screen density)。由于各种原因, 目前PPI(主要是iOS)和DPI(比如在Android中)都会用在计算机显示设备的参数描述中, 不过二者的意思是一样的, 都是代表像素密度。

高PPI屏幕显示的元素会比较精细 (看起来会比较小), 低PPI屏幕显示的元素相对来说就比粗糙 (看起来会比较大), 我们通过一幅图来看看在不同PPI下元素显示的区别:



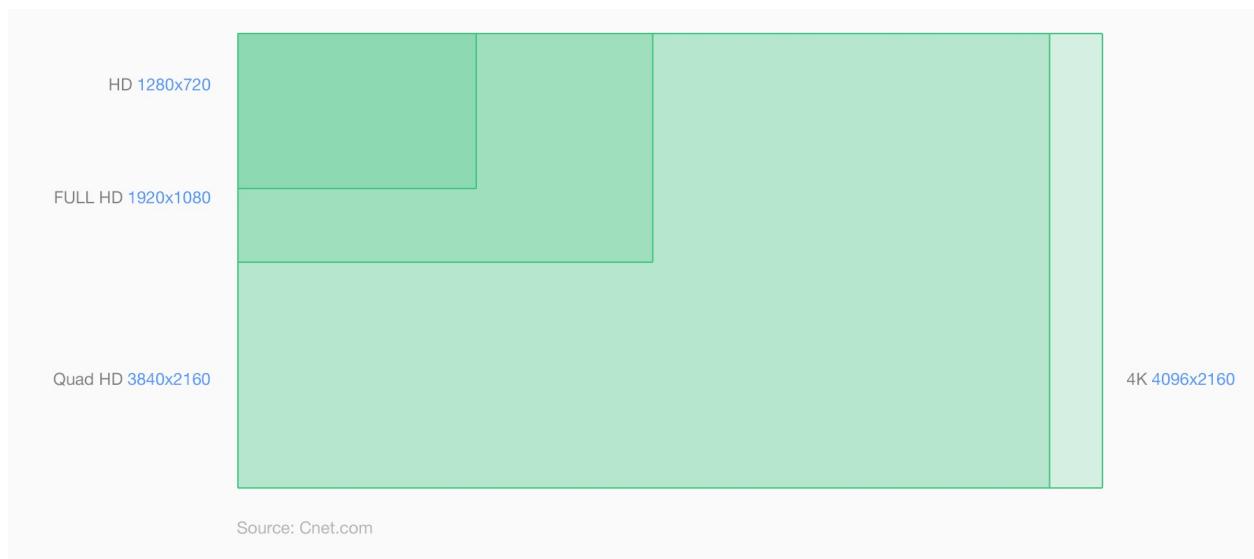
HD与4K

现在移动设备、智能电视宣传最多的两个关键词估计就是HD、4K，这二者都是用来描述显示设备分辨率的标准，到底二者之间有什么区别？

HD(High-Definition)的分辨率要高于1280x720px或者720p。

Full HD的分辨率要高于1920x1080px，目前是主流电视以及高端手机（比如Galaxy SIV, HTC one, Sony Xperia Z, Nexus5等）采用的是这个分辨率。

4K（也叫做Quad HD或者Ultra HD）的分辨率从3840x2160起步，主要是现在高端电视的分辨率；其还有一个更高的4096x2160的标准，主要用于电影放映机或者专业相机。

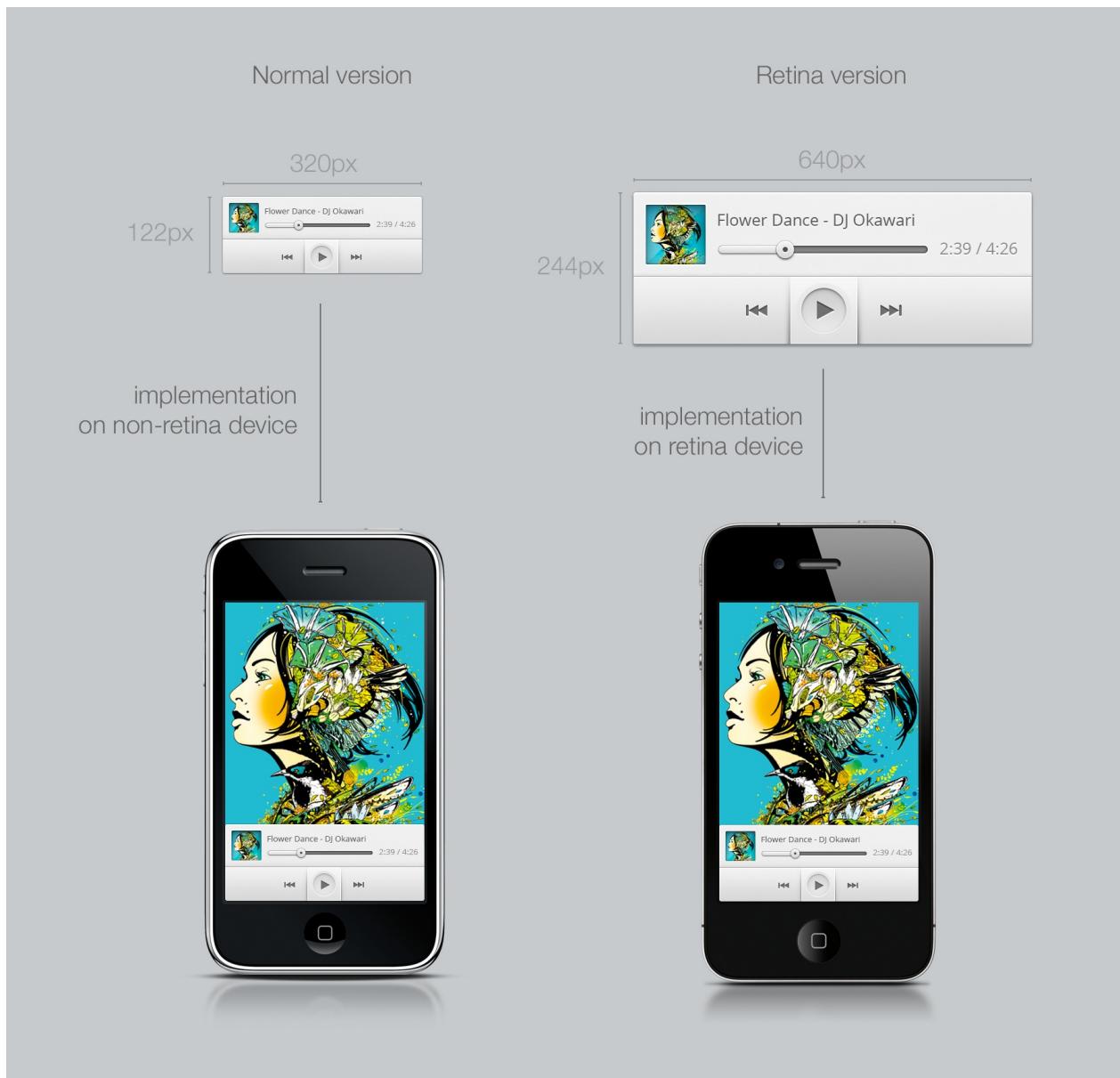


Retina

[Retina display](#)即视网膜屏幕，是苹果发布iPhone 4时候提出的，之所以叫做视网膜屏幕，是因为屏幕的PPI太高，人的视网膜无法分辨出屏幕上的像素点。iPhone 4/S的PPI为326，是iPhone 3G/S的两倍，如下图：



由于屏幕在宽和高的像素数量提高了整整一倍，所以之前非Retina屏幕上的一个像素渲染的内容在Retina屏幕上会用4个像素去渲染： $1\times 1\text{px}(\text{non Retina}) = 2\times 2\text{px}(\text{Retina})$ ，这样元素的内容就会变得精细，比如：

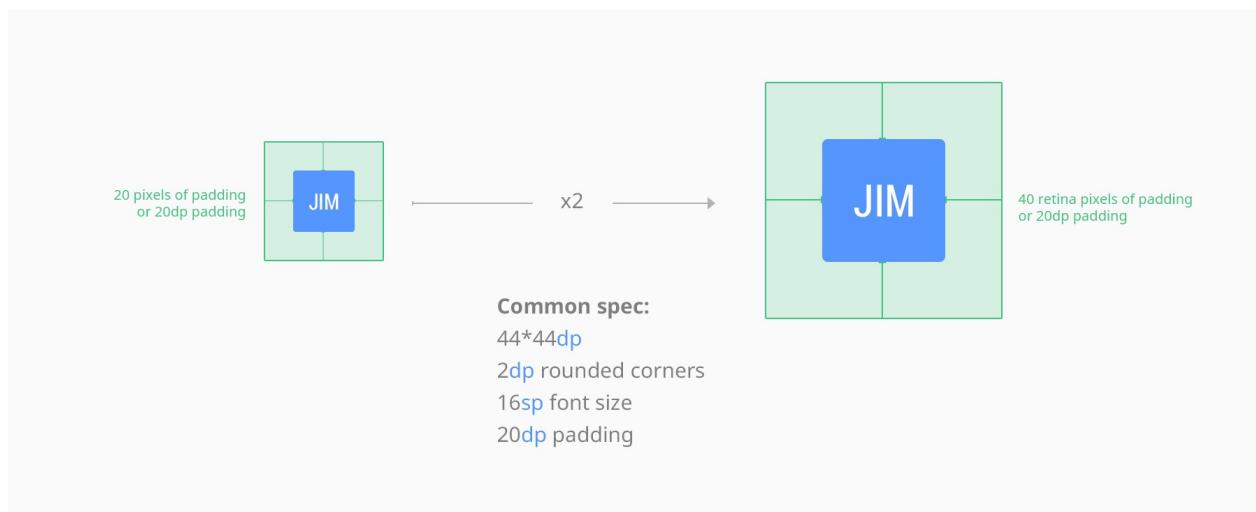


注意， Retina display 是苹果注册的命名方式，其他厂商只能使用 HI-DPI 或者其他的命名方式，不过意思都是一样的，就是屏幕的PPI非常高。

DP/PT/SP

随着移动设备屏幕PPI的不断提高，对于开发者来说以前用物理像素(Physical Pixel)来度量显示元素的方法已经不奏效了。为了解决这个问题，两大平台都提出了抽象像素的概念：iOS叫做PT (Point, 显示点)，Android中叫做DP/DIP (Device independent Pixel, 设备无关像素)，如果没有特殊说明，以下统一用DP来进行描述。

举个例子，44x44dp的元素在非Retina屏幕中等于44x44px，在Retina屏幕中等于88x88px（变为4倍）。

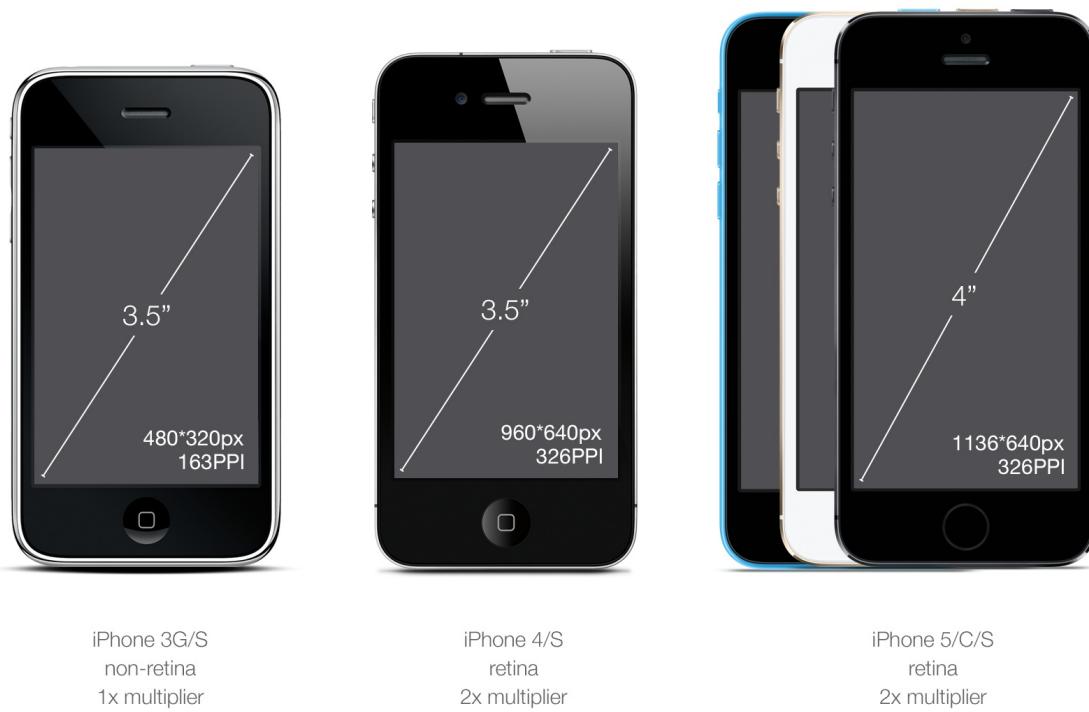


SP (Scale-independent pixel) 是缩放无关的像素，与DP和PT一样都是抽象像素，只不过用于描述字体的大小。

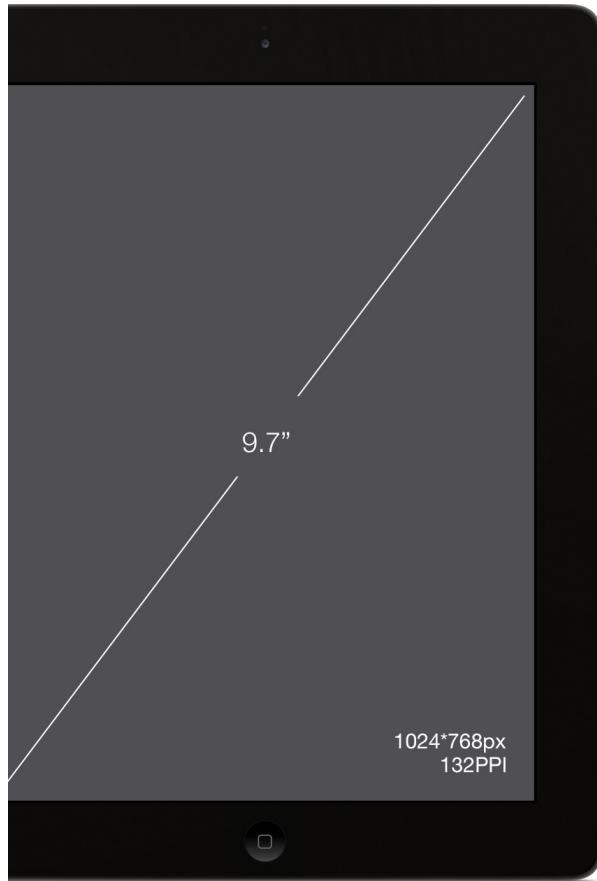
iOS中处理PPI

iOS中处理不同PPI显示的方法很简单：首先规定在多高的PPI下1DP等于1px，并以这个PPI作为基准（1x multiplier），如果显示设备的PPI是基准PPI的2倍，那么1DP等于2px（2x multiplier），其实就是简单的小学乘法。

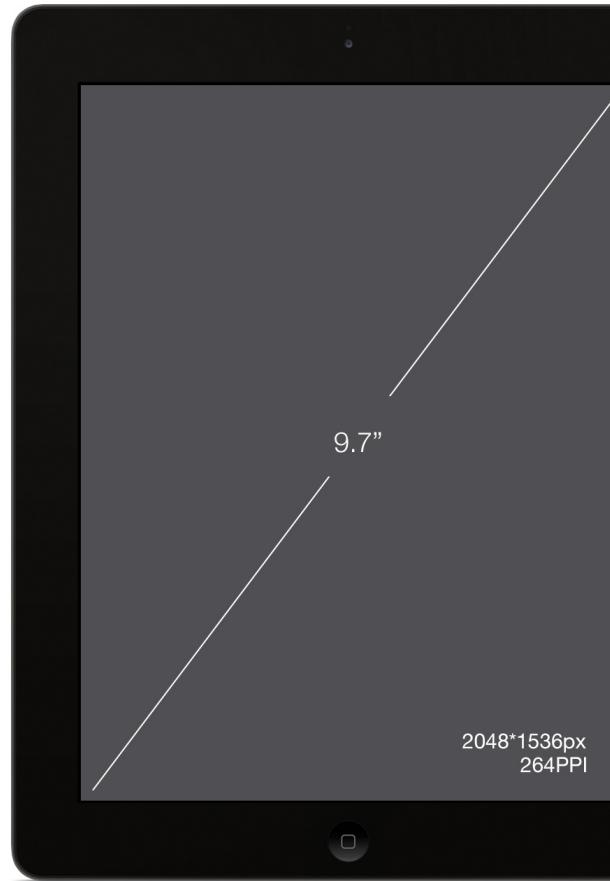
在iPhone系列中，3G/S为1x multiplier，其他为2x multiplier：



在iPad系列中， iPad 1代和2代为1x multiplier， 其他为2x multiplier：

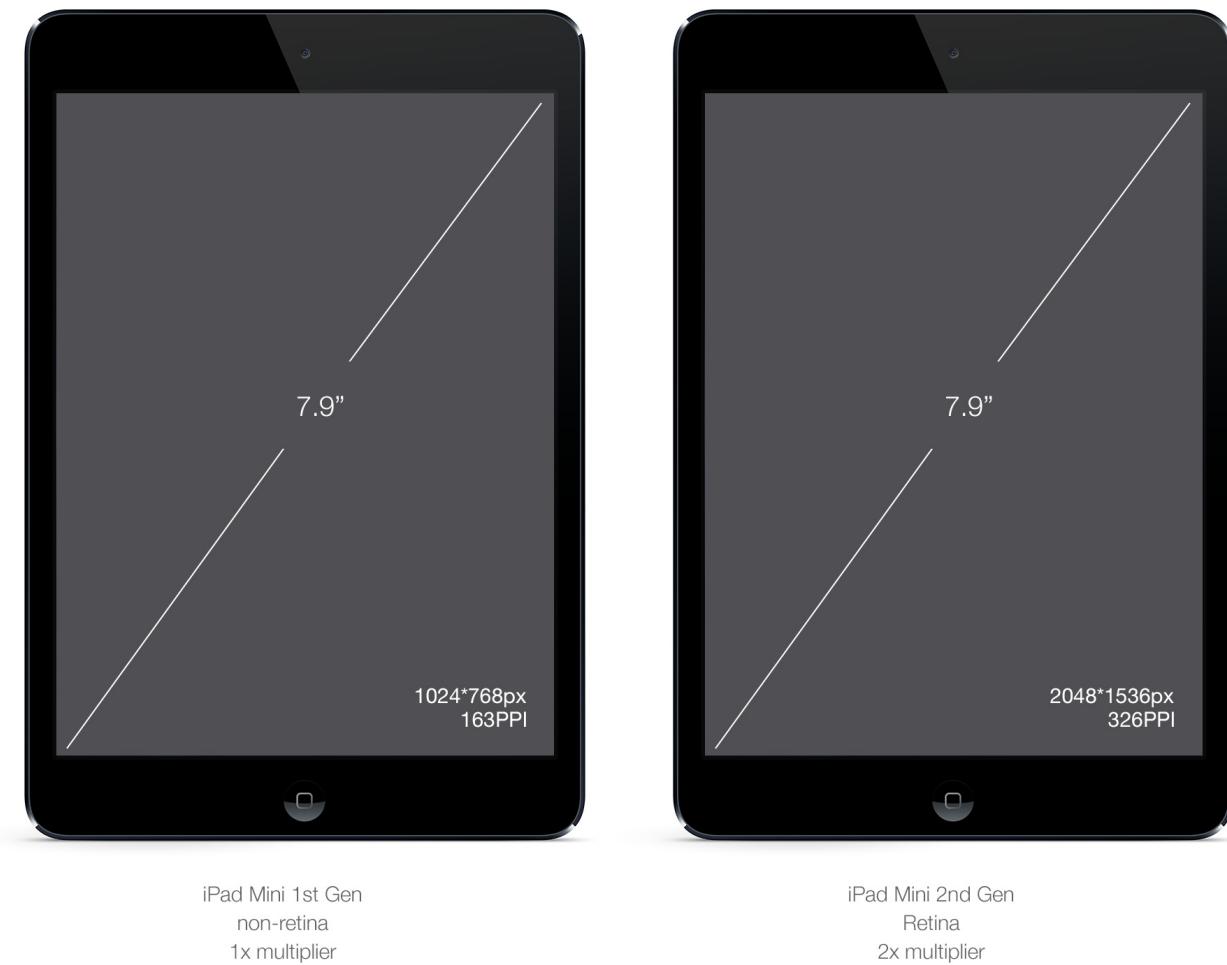


iPad 1 & 2
non-retina
1x multiplier

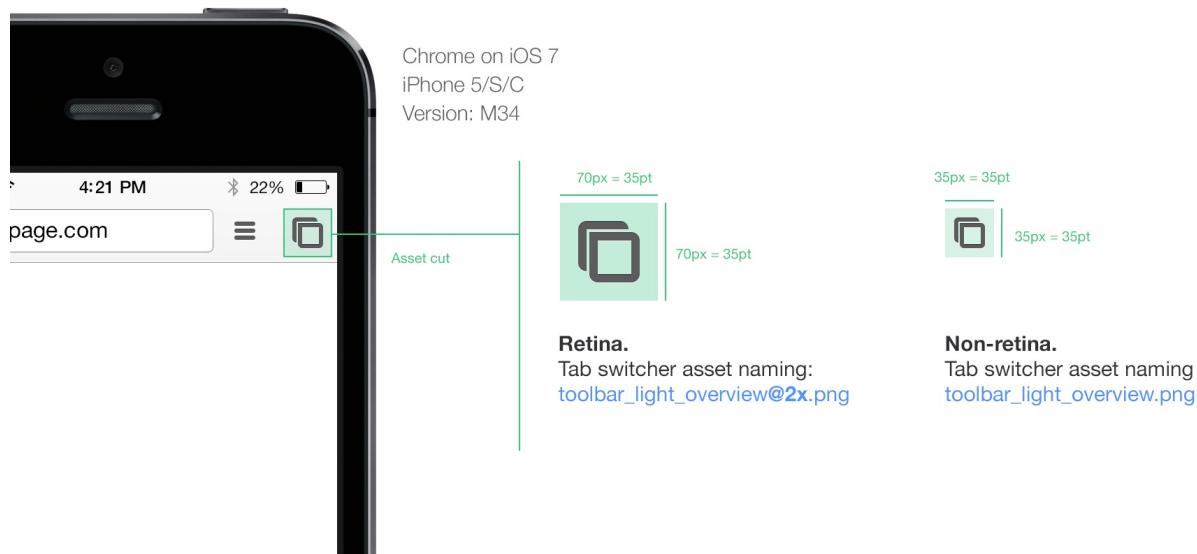


iPad 3 & 4 / Air
Retina
2x multiplier

在iPad Mini系列中， iPad Mini一代为1x multiplier， 其他为2x multiplier：



在iOS中，同一个应用在非Retina屏幕和Retina屏幕显示的资源是不同的，其规则是：
name.png为非Retina资源， name@2x.png 为Retina资源，所以对于设计人员来说，在你设计的时候需要考虑到Retina屏幕和非Retina屏幕，看下面这个例子：



Android中处理PPI

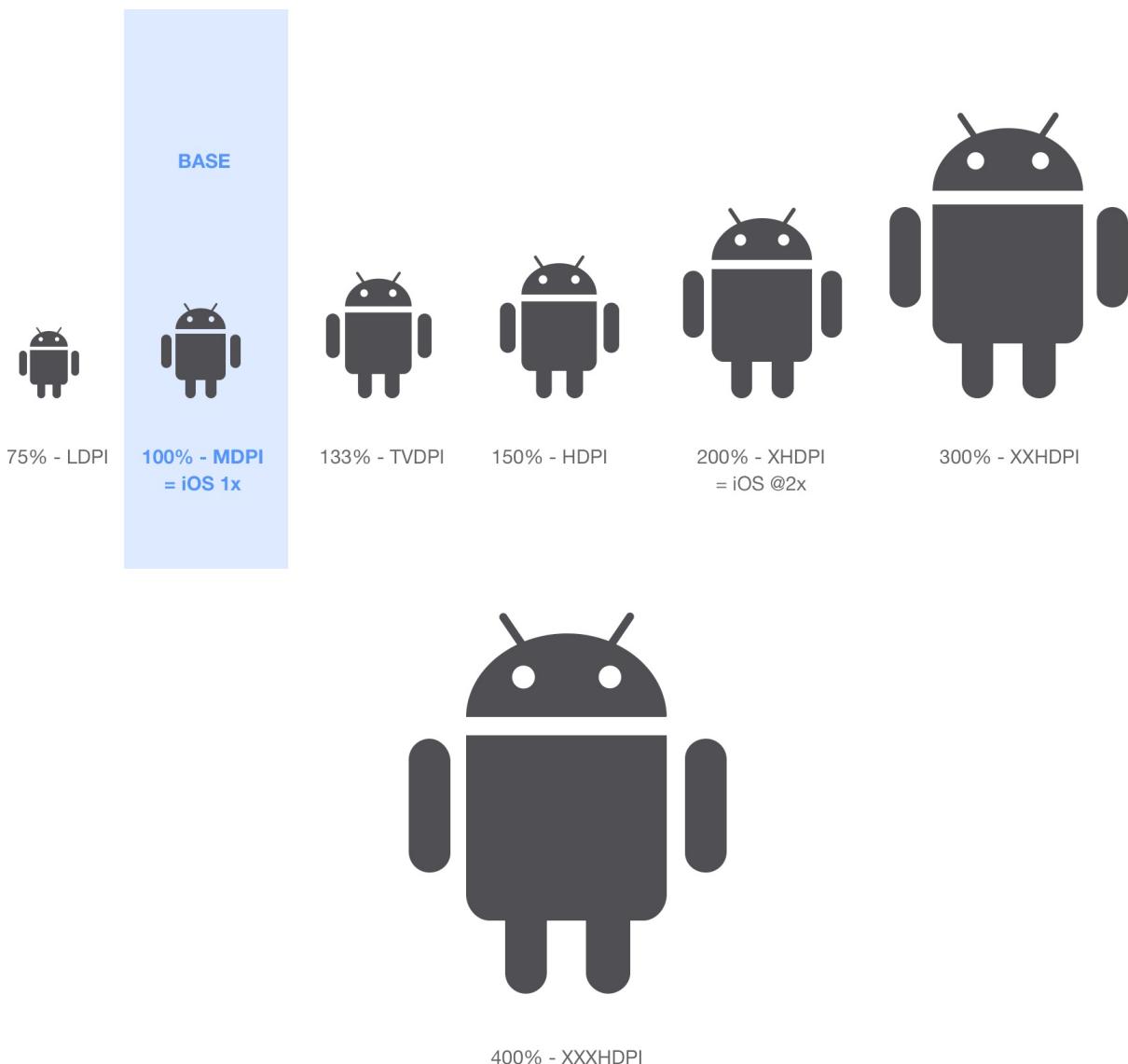
由于Android系统是开放的系统，要适配的PPI非常多，所以它对PPI划分的非常细：

- ldpi (low) ~120dpi
- mdpi (medium) ~160dpi
- hdpi (high) ~240dpi
- xhdpi (extra-high) ~320dpi
- xxhdpi (extra-extra-high) ~480dpi
- xxxhdpi (extra-extra-extra-high) ~640dpi

你需要把对应dpi的资源放到对应的目录就可以了，Android会根据dpi自动选择资源，目录规则如下：

- drawable-mdpi/asset.png
- drawable-hdpi/asset.png
- drawable-xhdpi/asset.png
- ...

可以看出Android中mdpi与iOS中的1x multiplier所代表的PPI是一样的，xhdpi与iOS的2x multiplier所代表的PPI一样，如图：



参考

- <http://sebastien-gabriel.com/designers-guide-to-dpi/home>
- http://developer.android.com/guide/practices/screens_support.html

特殊效果

Android悬浮窗TYPE_TOAST小结: 源码分析

来源:[简书](#)

前言

[Android无需权限显示悬浮窗, 兼谈逆向分析app](#)这篇文章阅读量很大, 这篇文章是通过逆向分析UC浏览器的实现和兼容性处理来得到一个悬浮窗的实现小技巧, 但有很多问题没有弄明白, 比如为什么在API 18及以下**TYPE_TOAST**的悬浮窗无法接受触摸事件, 为什么使用**TYPE_TOAST**就不需要权限.

期间[@廖祜秋liaohuqiu_秋百万](#)和我有较多探讨, 原文贴的一个[demo android-UCToast](#)也是他做的, 他也有写[Android 悬浮窗的小结](#). 这几篇关于悬浮窗的文章, 是我和他共同探索的结果, 非常感谢.

思路

老实说一开始我是想看看整个事件的传播过程, 从EventHub开始, 到View.onTouchEvent, 想看看Android系统内事件分发, 不过由于绝大部分代码在Native层, 我并没有搞清楚.

其实要想知道原因很简单, 只要grep一下**TYPE_TOAST**, 把每个用到的地方看一看, 自然就知道了, 但是恰好周末我手上没有源码, 只能在grepcode上面一个一个的查, 所以也花了不少时间.

正文

还是从最简单的地方开始, 我们调用了 `WindowManager.addView`, **WindowManager**是个接口, 我们使用的是他的实现类**WindowManagerImpl**, 看看它的addView方法:

```
@Override  
public void addView(View view, ViewGroup.LayoutParams params) {  
    mGlobal.addView(view, params, mDisplay, mParentWindow);  
}
```

mGlobal是**WindowManagerGlobal**的实例, 再看看 `WindowManagerGlobal.addView` :

```
public void addView(View view, ViewGroup.LayoutParams params,
    Display display, Window parentWindow) {
    .....
    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)params;
    .....
    synchronized (mLock) {
        .....
        root = new ViewRootImpl(view.getContext(), display);
        view.setLayoutParams(wparams);
        .....
    }
    // do this last because it fires off messages to start doing things
    try {
        root.setView(view, wparams, panelParentView);
    } catch (RuntimeException e) {
        // BadTokenException or InvalidDisplayException, clean up.
        .....
    }
}
```

代码中创建了一个 `ViewRootImpl`, 调用了它的 `setView`, 将我们要添加的view传入. 继续看 `ViewRootImpl.setView` :

```

public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView
synchronized (this) {
    if (mView == null) {
        .....
        mWindowAttributes.copyFrom(attrs);
        if (mWindowAttributes.packageName == null) {
            mWindowAttributes.packageName = mBasePackageName;
        }
        .....
        try {
            .....
            res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
                getHostVisibility(), mDisplay.getDisplayId(),
                mAttachInfo.mContentInsets, mInputChannel);
        } catch (RemoteException e) {
            .....
            throw new RuntimeException("Adding window failed", e);
        } finally {
            .....
        }
        .....
    }
}
}

```

对我们的分析来说最关键的代码是

```

res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
    getHostVisibility(), mDisplay.getDisplayId(),
    mAttachInfo.mContentInsets, mInputChannel);

```

mWindowSession 的类型是 IwindowSession , mWindow的类型是 Iwindow.Stub , 这句代码就是利用AIDL进行IPC, 实际被调用的是 Session.addToDisplay :

```

@Override
public int addToDisplay(Iwindow window, int seq, WindowManager.LayoutParams attrs,
    int viewVisibility, int displayId, Rect outContentInsets,
    InputChannel outInputChannel) {
    return mService.addWindow(this, window, seq, attrs, viewVisibility, displayId,
        outContentInsets, outInputChannel);
}

```

mService 是 WindowManagerService e, 继续往下跟:

```
public int addWindow(Session session, IWindow client, int seq,
                     WindowManager.LayoutParams attrs, int viewVisibility, int display
                     Rect outContentInsets, InputChannel outInputChannel) {
    int[] appOp = new int[1];
    int res = mPolicy.checkAddPermission(attrs, appOp);
    if (res != WindowManagerGlobal.ADD_OKAY) {
        return res;
    }
    .....
    final int type = attrs.type;

    synchronized(mWindowMap) {
        .....
        mPolicy.adjustWindowParamsLw(win.mAttrs);
        .....
    }
    .....
    return res;
}
```

mPolicy 是标记为final的成员变量：

```
final WindowManagerPolicy mPolicy = PolicyManager.makeNewWindowManager();
```

继续看 PolicyManager.makeNewWindowManager：

```

public final class PolicyManager {
    private static final String POLICY_IMPL_CLASS_NAME =
        "com.android.internal.policy.impl.Policy";

    private static final IPolicy sPolicy;

    static {
        // Pull in the actual implementation of the policy at run-time
        try {
            Class policyClass = Class.forName(POLICY_IMPL_CLASS_NAME);
            sPolicy = (IPolicy)policyClass.newInstance();
        } catch (ClassNotFoundException ex) {
            throw new RuntimeException(
                POLICY_IMPL_CLASS_NAME + " could not be loaded", ex);
        } catch (InstantiationException ex) {
            throw new RuntimeException(
                POLICY_IMPL_CLASS_NAME + " could not be instantiated", ex);
        } catch (IllegalAccessException ex) {
            throw new RuntimeException(
                POLICY_IMPL_CLASS_NAME + " could not be instantiated", ex);
        }
    }

    // Cannot instantiate this class
    private PolicyManager() {}

    .....
    public static WindowManagerPolicy makeNewWindowManager() {
        return sPolicy.makeNewWindowManager();
    }
    .....
}

```

这里 `sPolicy` 是 `com.android.internal.policy.impl.Policy` 对象, 再看看它的 `makeNewWindowManager` 方法返回的是什么:

```

public WindowManagerPolicy makeNewWindowManager() {
    return new PhoneWindowManager();
}

```

现在我们知道 `mPolicy` 实际上是 `PhoneWindowManager`, 那么

```
int res = mPolicy.checkAddPermission(attrs, appOp);
```

实际调用的代码是:

```

@Override
public int checkAddPermission(WindowManager.LayoutParams attrs, int[] outAppOp) {
    int type = attrs.type;

    outAppOp[0] = AppOpsManager.OP_NONE;

    if (type < WindowManager.LayoutParams.FIRST_SYSTEM_WINDOW
        || type > WindowManager.LayoutParams.LAST_SYSTEM_WINDOW) {
        return WindowManagerGlobal.ADD_OKAY;
    }
    String permission = null;
    switch (type) {
        case TYPE_TOAST:
            // XXX right now the app process has complete control over
            // this... should introduce a token to let the system
            // monitor/control what they are doing.
            break;
        case TYPE_DREAM:
        case TYPE_INPUT_METHOD:
        case TYPE_WALLPAPER:
        case TYPE_PRIVATE_PRESENTATION:
            // The window manager will check these.
            break;
        case TYPE_PHONE:
        case TYPE_PRIORITY_PHONE:
        case TYPE_SYSTEM_ALERT:
        case TYPE_SYSTEM_ERROR:
        case TYPE_SYSTEM_OVERLAY:
            permission = android.Manifest.permission.SYSTEM_ALERT_WINDOW;
            outAppOp[0] = AppOpsManager.OP_SYSTEM_ALERT_WINDOW;
            break;
        default:
            permission = android.Manifest.permission.INTERNAL_SYSTEM_WINDOW;
    }
    if (permission != null) {
        if (mContext.checkSelfPermission(permission)
            != PackageManager.PERMISSION_GRANTED) {
            return WindowManagerGlobal.ADD_PERMISSION_DENIED;
        }
    }
    return WindowManagerGlobal.ADD_OKAY;
}

```

我截取的是4.4_r1的代码, 我们最关心的部分其实一直没有变, 那就是**TYPE_TOAST**根本没有做权限检查, 直接break出去了, 最后返回**WindowManagerGlobal.ADD_OKAY**.

不需要权限显示悬浮窗的原因已经找到了, 接着刚才addWindow方法的分析, 继续看下面一句:

```
`` mPolicy.adjustWindowParamsLw(win.mAttrs);
```

也就是`PhoneWindowManager.adjustWindowParamsLw`，注意这里我给出了三个版本的实现，一个是2.0到2.3.7

```
//Android 2.0 - 2.3.7 PhoneWindowManager public void
adjustWindowParamsLw(WindowManager.LayoutParams attrs) { switch (attrs.type) {
case TYPE_SYSTEM_OVERLAY: case TYPE_SECURE_SYSTEM_OVERLAY: case
TYPE_TOAST: // These types of windows can't receive input events. attrs.flags |=
WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE |
WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE; break; } }

//Android 4.0.1 - 4.3.1 PhoneWindowManager public void
adjustWindowParamsLw(WindowManager.LayoutParams attrs) { switch (attrs.type) {
case TYPE_SYSTEM_OVERLAY: case TYPE_SECURE_SYSTEM_OVERLAY: case
TYPE_TOAST: // These types of windows can't receive input events. attrs.flags |=
WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE |
WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE; attrs.flags &=
~WindowManager.LayoutParams.FLAG_WATCH_OUTSIDE_TOUCH; break; } }

//Android 4.4 PhoneWindowManager @Override public void
adjustWindowParamsLw(WindowManager.LayoutParams attrs) { switch (attrs.type) {
case TYPE_SYSTEM_OVERLAY: case TYPE_SECURE_SYSTEM_OVERLAY: //
These types of windows can't receive input events. attrs.flags |=
WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE |
WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE; attrs.flags &=
~WindowManager.LayoutParams.FLAG_WATCH_OUTSIDE_TOUCH; break; } } ``
```

grepcode上没有3.x的代码，我也没查具体是什么，没必要考虑3.x。

可以看到，在4.0.1以前，当我们使用**TYPE_TOAST**，Android会偷偷给我们加上**FLAG_NOT_FOCUSABLE**和**FLAG_NOT_TOUCHABLE**，4.0.1开始，会额外再去掉**FLAG_WATCH_OUTSIDE_TOUCH**，这样真的是什么事件都没了。而4.4开始，**TYPE_TOAST**被移除了，所以从4.4开始，使用**TYPE_TOAST**的同时还可以接收触摸事件和按键事件了，而4.4以前只能显示出来，不能交互。

API level 18及以下使用TYPE_TOAST无法接收触摸事件的原因也找到了。

尾声

原文发的时候很多事情没搞清楚, 后来文章编辑了十几次, 加上这篇文章, 基本上把所有的疑问都搞明白了. 嗯, 关于这个神奇的悬浮窗的事情应该到这里就结束了.

本人水平有限, 如有错误, 欢迎指正, 以免误导他人

Android无需权限显示悬浮窗, 兼谈逆向分析app

来源:[简书](#)

前言

最近UC浏览器中文版出了一个快速搜索的功能, 在使用其他app的时候, 如果复制了一些内容, 屏幕顶部会弹一个窗口, 提示一些操作, 点击后跳转到UC, 显示这个悬浮窗不需要申请 `android.permission.SYSTEM_ALERT_WINDOW` 权限.

如下图, 截图是在使用Chrome时截的, 但是屏幕顶部却有UC的view浮在屏幕上. 我使用的是小米, 我并没有给UC授悬浮窗权限, 所以我看到这个悬浮窗时是很震惊的.



快速搜索

网络同传

好，同传以后所有的电脑都装好操作系统了，很方便。

局域网内有一百多台电脑，全部都是linux操作系统，所有电脑配置相同，系统完全相同（包括用户名和密码），ip地址是自动分配的。现在有个任务是在这些电脑上执行某些命令，或者说进行某些操作，比如安装某些软件，拷贝某些文件，批量关机等。如果一台一台得手工去操作，费时又费力，如果要进行多个操作就更麻烦啦。

或许你会想到网络同传，网络同传是什么？就是在一台电脑上把电脑装好，配置好，然后利用某些软件，如“联想网络同传”把系统原样拷贝过去，在装系统时很有用，只要在一台电脑上装好，同传以后所有的电脑都装好操作系统了，很方便。同传要求所有电脑硬件完全相同，在联想的电脑上装的系统传到方正电脑上肯定会出问题的。传系统也是很费时间的，根据硬盘大小，如果30G硬盘，100多台电脑大约要传2个多小时，反正比一台一台地安装快！但是如果系统都传完了，发现忘了装一个软件，或者还需要做些小修改，再同传一次可以，但是太慢，传两次半天时间就没了。这时候我们可以利用ssh去控制每台电脑去执行某些命令。

先让我们回忆一下SSH远程登录的过程，首先执行命令



分享



官方微博

@程序员的万花筒
weibo.com/u/2951917192

悬浮窗原理

做过悬浮窗功能的人都知道, 要想显示悬浮窗, 要有一个服务运行在后台, 通过 `getSystemService(Context.WINDOW_SERVICE)` 拿到 `WindowManager`, 然后向其中`addView`, `addView`第二个参数是一个 `WindowManager.LayoutParams`, `WindowManager.LayoutParams` 中有一个成员`type`, 有各种值, 一般设置成`TYPE_PHONE`就可以悬浮在很多view的上方了, 但是调用这个方法需要申请 `android.permission.SYSTEM_ALERT_WINDOW` 权限, 在很多机型上, 这个权限的名字叫悬浮窗, 比如小米手机上默认是禁用这个权限的, 有些恶意app会用这个权限弹广告, 而且很难追查是哪个应用弹的. 如果这个权限被禁用, 那么结果就是悬浮窗无法展示, 比如有道词典的复制查词功能, 在小米手机上经常没用, 其实是用户没有授权, 而且应用也没有引导用户给它打开授权.

现在UC能突破这个限制, 我很好奇它是怎么做到的.

研究实现

Android开发有点蛋疼的地方就是太容易被反编译, 但有时这也成为我们研究别人app的一种手段.

反编译

使用apktool可以很轻松的反编译UC.

找代码

逆向别人的app, 比较关键的地方是怎么找代码, 因为代码基本上都是混淆的, 直接看肯定是看不懂的, 只能去找, 突破口一般在字符资源上, 比如我们看到上图中的**快速搜索**是UC的字符, 那么我们到 `res/values/strings.xml` 去找**快速搜索**, 就可以找到下面的内容

```
<string name="dark_search_banner_search">快速搜索</string>
```

这里我们拿到了**快速搜索**对应的名字 `dark_search_banner_search`, Android在编译时会给每个资源分配一个id, 我们grep一下这个字符资源的名字就能知道id是多少, 一般在 `R.java`, `res/values/public.xml` 中有定义, 我直接到 `public.xml` 中找到了它的id

```
<public type="string" name="dark_search_banner_search" id="0x7f070049" />
```

有了字符资源的id 0x7f070049, 我们再在代码里面grep一下这个id, 就能知道哪几个文件使用了这个字符资源.

之所以这么确定是在代码里, 是因为UC在我们复制的内容不同时, 悬浮窗标题会不一样, 一定是在代码里控制的, 结果如下

```
./com/uc/browser/b/f.smali
```

结果可能和大家不一样, 但是一定会找到一个被混淆的smali文件

看代码

这一部应该是最恶心的. smali代码和java代码的关系, 就像汇编代码和C++代码, 但是smali比汇编代码要容易理解的多, 不然也不会有那么多公司故意将代码写在C++层了.

虽然代码都被混淆了, 而且以我们不熟悉的方式出现, 但我们可以根据一些蛛丝马迹来判断代码的执行, 比如Framework的类和API是不能被混淆的, 这也是我们能看懂smali的原因之一, 我们可以结合这些面包屑来还原整个app代码, 当然这需要我们对smali很熟悉, 如果不熟悉smali, 至少要对Android的API熟悉. 因为有时实在看不懂, 我们要靠猜来还原一段代码的逻辑.

首先在代码里面找到 0x7f070049 , 发现了如下代码

```
(省略)
const v3, 0x7f070049

invoke-virtual {v1, v3}, Landroid/content/res/Resources;->getString(I)Ljava/lang/String;
move-result-object v1

input-object v1, v0, Lcom/uc/browser/b/a;->dpC:Ljava/lang/String;

:cond_9

(省略)

invoke-virtual {v0, v1}, Lcom/uc/browser/b/a;->o(Landroid/graphics/drawable/Drawable;
:try_end_2
.catch Ljava/lang/Exception; {:try_start_2 .. :try_end_2} :catch_0

goto/16 :goto_0
(省略)
```

这是 0x7f070049 出现之后的一部分代码, 一路看下来, 其实都是在取值赋值, 就拿 0x7f070049 来说:

```
#使v3寄存器的值为0x7f070049
const v3, 0x7f070049
#v1是Resources实例, 调用它的getString方法, 方法的参数是v3中的值
invoke-virtual {v1, v3}, Landroid/content/res/Resources;->getString(I)Ljava/lang/
#将结果存入v1寄存器
move-result-object v1
```

其实就是我们常用的 `getResources().getString`

其实如果一直这么看下去, 会发现毫无头绪, 剩下的代码一直在干差不多的事情, 所以我只截取了这部分, 注意最后一行

```
goto/16 :goto_0
```

也就是说, 有可能代码转到`goto_0`那儿去了, 那么看看`goto_0`那里又写了些什么

```
:goto_0
(省略)

const-string v1, "window"

invoke-virtual {v0, v1}, Landroid/content/Context;->getSystemService(Ljava/lang/S
move-result-object v0

check-cast v0, Landroid/view/WindowManager;

invoke-interface {v0}, Landroid/view/WindowManager;->getDefaultDisplay()Landroid/v
move-result-object v0

invoke-virtual {v0}, Landroid/view/Display;->getWidth()I

move-result v0

iget-object v1, v10, Lcom/uc/browser/b/a;->dpx:Landroid/view/WindowManager$Layout
input v0, v1, Landroid/view/WindowManager$LayoutParams;->width:I

iget-object v0, v10, Lcom/uc/browser/b/a;->dpx:Landroid/view/WindowManager$Layout
input v0, v1, Lcom/uc/browser/b/a;->mWindowManager:Landroid/view/WindowManager

invoke-virtual {v10}, Lcom/uc/browser/b/a;->getContext()Landroid/content/Context;

move-result-object v1

invoke-virtual {v1}, Landroid/content/Context;->getResources()Landroid/content/res
move-result-object v1

const v2, 0x7f0d0022

invoke-virtual {v1, v2}, Landroid/content/res/Resources;->getDimension(I)F

move-result v1

float-to-int v1, v1

input v1, v0, Landroid/view/WindowManager$LayoutParams;->height:I

iget-object v0, v10, Lcom/uc/browser/b/a;->mWindowManager:Landroid/view/WindowManag
input v0, v1, Lcom/uc/browser/b/a;->dpx:Landroid/view/WindowManager$Layout

invoke-interface {v0, v10, v1}, Landroid/view/WindowManager;->addView(Landroid/vi
```

其实看到 `const-string v1, "window"`, 我们就应该有所警惕了, 这可能是关键代码了. 为什么这么说? 因为悬浮窗的实现里面, 需要获取 `WindowManager`, 从而需要调用 `context.getSystemService(Context.WINDOW_SERVICE)`, 而官方文档写了 `Context.WINDOW_SERVICE` 就是常量 `window`. 而后我们看到代码中构造了 `WindowManager.LayoutParams``, 最终在 `addView` 时传入.

看到这里, 我也觉得很奇怪, 我在悬浮窗原理中写的是我知道的实现悬浮窗的方法, UC的实现好像跟我调用的是相同的API, 也没看到反射之类可能展示奇技淫巧的代码, 为什么UC就可以不需要权限直接显示悬浮窗呢?

猜测

我认为 `addView` 的第二个参数 `WindowManager.LayoutParams` 可能是关键, 所以我需要知道 UC是如何构造这个 `WindowManager.LayoutParams` 的.

由于是系统的类, 无法混淆, 直接搜索 `LayoutParams` 就找到了下面的代码

```
    igure-object v1, v10, Lcom/uc/browser/b/a; ->dpx:Landroid/view/WindowManager$LayoutParams
```

这句话就是把 `v10` 的值赋给 `v1`, `v10` 是 `com/uc/browser/b/a` 的成员 `dpx`, 那么打开 `com/uc/browser/b/a.smali` 看看 `dpx` 到底是怎么构造的.

(省略)

```
.field dpx:Landroid/view/WindowManager$LayoutParams;

(省略)
.line 68
new-instance v0, Landroid/view/WindowManager$LayoutParams;

invoke-direct {v0}, Landroid/view/WindowManager$LayoutParams; -><init>()

input-object v0, p0, Lcom/uc/browser/b/a; ->dpx:Landroid/view/WindowManager$LayoutParams;

.line 69
if-eqz p2, :cond_0

.line 70
iget-object v0, p0, Lcom/uc/browser/b/a; ->dpx:Landroid/view/WindowManager$LayoutParams;

const/16 v1, 0x7d5

input v1, v0, Landroid/view/WindowManager$LayoutParams; ->type:I

.line 74
:goto_0
iget-object v0, p0, Lcom/uc/browser/b/a; ->dpx:Landroid/view/WindowManager$LayoutParams;

const/4 v1, 0x1

input v1, v0, Landroid/view/WindowManager$LayoutParams; ->format:I
(省略)
```

这里的代码就很简单了，我最先看的是下面这段

```
const/16 v1, 0x7d5

input v1, v0, Landroid/view/WindowManager$LayoutParams; ->type:I
```

这两句代码就是把 `WindowManager.LayoutParams.type` 字段设成 `0x7d5`，官网上写了 `0x000007d5` 是 `WindowManager.LayoutParams.TYPE_TOAST` 的值。

验证

实际测试了一下，将 `type` 设置成 **TYPE_TOAST** 果然有效，不需要 `android.permission.SYSTEM_ALERT_WINDOW` 权限就能显示一个悬浮窗。

之前我一直以为调用了系统 `WindowManager.addView` 需要 `android.permission.SYSTEM_ALERT_WINDOW` 权限, 但实际上调用这个方法是不需要权限的, 在Android源码中有这么一段

```
public int checkAddPermission(WindowManager.LayoutParams attrs) {
    int type = attrs.type;

    if (type < WindowManager.LayoutParams.FIRST_SYSTEM_WINDOW
        || type > WindowManager.LayoutParams.LAST_SYSTEM_WINDOW) {
        return WindowManagerImpl.ADD_OKAY;
    }

    String permission = null;
    switch (type) {
        case TYPE_TOAST:
            // XXX right now the app process has complete control over
            // this... should introduce a token to let the system
            // monitor/control what they are doing.
            break;
        case TYPE_INPUT_METHOD:
        case TYPE_WALLPAPER:
            // The window manager will check these.
            break;
        case TYPE_PHONE:
        case TYPE_PRIORITY_PHONE:
        case TYPE_SYSTEM_ALERT:
        case TYPE_SYSTEM_ERROR:
        case TYPE_SYSTEM_OVERLAY:
            permission = android.Manifest.permission.SYSTEM_ALERT_WINDOW;
            break;
        default:
            permission = android.Manifest.permission.INTERNAL_SYSTEM_WINDOW;
    }
    if (permission != null) {
        if (!mContext.checkCallingOrSelfPermission(permission)
            != PackageManager.PERMISSION_GRANTED) {
            return WindowManagerImpl.ADD_PERMISSION_DENIED;
        }
    }
    return WindowManagerImpl.ADD_OKAY;
}
```

可以猜到这个方法是往系统的 `WindowManager` 里 `addView` 的时候做权限检查用的, 那个 `type` 就是我们在构造 `WindowManager.LayoutParams` 时赋值的 `type`, 可以看到, 除了 **TYPE_TOAST**, 其他都是要权限的, 而且非常喜感的是, 代码中的注释还说他们现在对这种 `type` 毫无限制, 应该引入标记来限制开发者.

处理兼容性

在这篇文章刚刚公布的时候, 就有同学反馈悬浮窗无法接收事件, 刚开始我并没有特别在意, 在廖祐秋大神做了一个demo之后, 这篇文章阅读量又涨了不少, 随即收到更多反馈事件的问题, 我今天晚上借了台MIUI V5 4.2.2实测了一下, 这台机器上UC的快速搜索功能也无法正常使用.

在这个ROM上表现为:

- 使用**TYPE_PHONE**这类需要权限的type时, 只有在app处于前台时能显示悬浮窗, 且能正常接受触摸事件. 如果在应用详情里面授悬浮窗权限, 则工作完全正常.(这里是MIUI V5对悬浮窗的特殊处理, 现在的ROM, 包括MIUI V6上, 如果不授权, 无法显示任何悬浮窗)
- 使用**TYPE_TOAST**这个不需要权限的type时, 悬浮窗正常显示, 但不能接受触摸事件.

我重新检查了一下smali代码, 发现UC是有分版本处理的, 不过因为smali代码的规则问题, 很难直接看出来, 我把分析过程写出来, 顺便解释一下smali的语法, 供大家以后逆向时拿来参考.

这次我是在OS X上反编译的, 所以变量名可能略有区别.

接着上面 `com/uc/browser/b/a.smali` 中查看dpx的构造过程, 代码如下:

```
.field dpx:Landroid/view/WindowManager$LayoutParams;

(省略)

# direct methods
.method public constructor <init>(Landroid/content/Context;Z)V
.locals 7

(省略)

.line 68
new-instance v0, Landroid/view/WindowManager$LayoutParams;

invoke-direct {v0}, Landroid/view/WindowManager$LayoutParams;-><init>()V

input-object v0, p0, Lcom/uc/browser/b/a;->dpx:Landroid/view/WindowManager$LayoutParams;

.line 69
if-eqz p2, :cond_0

.line 70
iget-object v0, p0, Lcom/uc/browser/b/a;->dpx:Landroid/view/WindowManager$LayoutParams;

const/16 v1, 0x7d5

input v1, v0, Landroid/view/WindowManager$LayoutParams;->type:I
```

为了方便说明, 我遵循smali的规则, 它用 .line xx , 我们就说这是第XX行的代码.

上面是我之前分析得到UC使用的是**TYPE_TOAST**的地方, 证据就是第70行的 const/16 v1 , 0x7d5 , 但是要知道, smali代码没有跳转的话, 就是从上往下执行, 我们看第69行的代码如下:

```
.line 69
if-eqz p2, :cond_0
```

这句话的意思是如果p2等于0, 控制流跳转到**cond_0**, 否则就是继续顺序往下执行. 也就是说UC只有在p2 != 0条件满足的时候才会使用TYPE_TOAST, 我们看看**cond_0**对应的代码.

```
.line 72
:cond_0
    ige-object v0, p0, Lcom/uc/browser/b/a;->dpx:Landroid/view/WindowManager$LayoutParams
    const/16 v1, 0x7d2

    irtual v1, v0, Landroid/view/WindowManager$LayoutParams;->type:I
```

这里很简单, 就是将0x7d2赋给了type, 官网写了0x000007d2是TYPE_PHONE, 也就是说UC在某种情况下还是会用需要权限的老方法展示悬浮窗.

现在问题是条件是什么, 关键在p2, 在smali里面, 有两种寄存器命名规则, 一种叫v命名规则, 另一种是p命名规则, 当然只是命名规则而已, 在使用apktool时是可以选的. 这里是p命名规则.

我刚才分析的赋值过程, 所有的方法是下面这个, 我在刚才的代码片段中也保留了这个部分.

```
# direct methods
.method public constructor <init>(Landroid/content/Context;Z)V
.locals 7
```

这就是 com/uc/browser/b/a 的构造方法, dpx就是在构造方法里初始化的, .locals 7 告诉我们这个方法中将出现7个局部寄存器(local register), 名字是v0, v1...v6, 而这个方法的参数有3个, 隐式告诉我们这个方法中将出现3个参数寄存器(parameter register), 名字分别是p0, p1, p2.

我是怎么知道这个方法有3个参数的呢. smali中非静态方法, 都隐含一个参数p0, 指向自身, 和Java中的this是一个意思, 而方法的参数写在括号里, 也就是 Landroid/content/Context;z , 其中 Landroid/content/Context; 很明显就是Android中的 Context, 值存储在p1里, 而Z对应的是Android中的boolean, p2就是他了.

也就是说, type是用TYPE_TOAST还是用TYPE_PHONE, 取决于这个构造方法的第二个参数, 那到底谁构造了 com/uc/browser/b/a 呢? 可以去代码里面搜形如 new-instance ***, Lcom/uc/browser/b/a; 的代码. 更保险的做法是搜 Lcom/uc/browser/b/a 然后一个一个的看.

我在 com/uc/browser/b/f.smali 里面找到了下面的代码:

```

.prologue
const/4 v0, 0x0

const/4 v1, 0x1

(省略)

new-instance v3, Lcom/uc/browser/b/a;

iget-object v4, v9, Lcom/uc/browser/b/e; ->mContext:Landroid/content/Context;

sget v5, Landroid/os/Build$VERSION; ->SDK_INT:I

const/16 v6, 0x13

if-lt v5, v6, :cond_0

move v0, v1

:cond_0
invoke-direct {v3, v4, v0}, Lcom/uc/browser/b/a; -><init>(Landroid/content/Context;Z)V

```

这段代码首先是创建了 `com/uc/browser/b/a` 的实例, 存储在 `v3` 中, 从另一处拿到了一个 `Context` 存储在 `v4` 中, 然后拿到了当前系统的 `android.os.Build.VERSION.SDK_INT` 存储在 `v5` 中, 此时将 `v6` 的值设为 `0x13`, 千万别粗心看成 13 了, 我好几次都觉得这是 13, 其实是十进制的 19, 接下来是一个条件分支, 如果 `v5` 的值小于 `v6`, 也就是说 `android.os.Build.VERSION.SDK_INT < 19`, 直接跳转到 `cond_0`, 否则先将 `v1` 的值赋给 `v0`, 再顺序执行.

这句代码

```
invoke-direct {v3, v4, v0}, Lcom/uc/browser/b/a; -><init>(Landroid/content/Context;Z)V
```

就是调用 `v3` 的构造方法, 参数是 `v4` 和 `v0`, 分析一下上面这段代码的逻辑就是:

如果当前系统 API level 小于 19, 那么第二个参数就是 0, 否则就是 1.

而这第二个参数的值就是之前我们分析的 `p2` 的值, UC 只有在 `p2 != 0` 条件满足的时候才会使用 **TYPE_TOAST**, 把整个逻辑串起来就是:

UC 在 `API level >= 19` 的时候, 使用 **TYPE_TOAST**, 其他情况使用 **TYPE_PHONE** (需要权限).

可能是为了规避在低版本**TYPE_TOAST**不能接受事件的问题.

关于针对源代码的分析, 请看[Android悬浮窗使用TYPE_TOAST的小结](#)

实测效果

我之前写的一个app有悬浮窗播放功能, 支持拖动窗口和点击暂停, 关闭窗口等等, 在4.4.4上实测功能正常.

20:31 0.00K/s 99%

< 分类浏览

音乐

 Fall Out Boy – Immortals (End Credit Version) ["From "Big Hero 6"] .mp3
7.40MB | 2015-9-21 14:02

 Fall Out Boy – My Songs Know What You Did In The Dark (Light Em)
7.25MB | 2015-9-21 14:01

 Fools Garden – Lemon Tree.mp3
7.33MB | 2015-9-21 14:02

 Goose house – 光るなら.mp3
10.14MB | 2015-9-21 14:00

 GUMI 鏡音リン – LUVORATORRRRY!.mp3
7.95MB | 2015-9-21 14:02

 胡歌 – 六月的雨.mp3
8.98MB | 2015-9-21 14:00

 胡歌 – 道遥叹.mp3
12.25MB | 2015-9-21 14:00

 Hans Zimmer – A Watchful Guardian.mp3

感谢微博上关注的大神[廖祜秋](#), 他做了个[demo](#), 虽然交互和UC不同, 可以参考一下实现.

The screenshot shows a mobile browser interface. At the top, there's a navigation bar with icons for signal strength, battery level, and time (6:21). Below this is the address bar, which displays a lock icon followed by the URL <https://github.com/liaohuqiu/android-UCToast>. To the right of the address bar are three vertical dots and a menu icon.

The main content area displays the GitHub repository page for 'liaohuqiu / android-UCToast'. The page title is 'liaohuqiu / android-UCToast'. Below the title, there's a brief description: 'Show how UC browser show a system overlay view without any permission'. The repository card shows a 'master' branch with a blue dropdown arrow icon. Below the branch name, it says 'Latest commit by liaohuqiu 9 minutes ago'. There are two buttons at the bottom of the card: 'View code' with a folder icon and 'Jump to file' with a magnifying glass icon.

Below the repository card, there's a section titled 'README.md' with a document icon. A large text overlay in Chinese reads: 'UC 浏览器复制，无需权限提示悬
浮窗实现' (Implementation of a floating window for copying in UC browser, no permission required).

At the very bottom of the screenshot, there's a black navigation bar with three white icons: a left arrow, a house-like symbol, and a square-like symbol.

关于这个,他也写了一篇[Android 悬浮窗的小结](#)

其他补充

评论区的浮海大虾同学有更多补充如下：

TYPE_TOAST一直都可以显示，但是用TYPE_TOAST显示出来的在2.3上无法接收点击事件，因此还是无法随意使用。

下面是我之前研究后台线程显示对话框的时候记得笔记，大家可以看看我们项目中有需求需要在后台任务中显示Dialog，项目最初的做法是用Activity模拟Dialog，一个Activity已经承载了近20种Dialog，代码混乱至极。后来我发现Dialog可以通过改变Window Type实现不依赖Activity显示，然后就很兴奋的要在使用这种方式来作为新的实现方式。

最初WindowType是**WindowManager.LayoutParams.TYPE_SYSTEM_ALERT**，可是这是悬浮窗了，MIUI会默认禁止(真他妈操蛋，也没有任何提示)最终放弃。后来试着换成了**WindowManager.LayoutParams.TYPE_TOAST**，起初效果很好，MIUI也不禁止了，哪里都能显示，这下开心了。可是后来又发现在2.3上不能接收点击事件，也就是说Dialog上的按钮不能点击，这他妈就很操蛋了，又放弃了。又试了试其他的Type都不能满足需求，结果如下：

- TYPE_SEARCH_BAR: 未知
- TYPE_ACCESSIBILITY_OVERLAY: 拒绝使用
- TYPE_APPLICATION: 只能配合Activity在当前APP使用
TYPE_APPLICATION_ATTACHED_DIALOG: 只能配合Activity在当前APP使用
- TYPE_APPLICATION_MEDIA: 无法使用(什么也不显示)
- TYPE_APPLICATION_PANEL: 只能配合Activity在当前APP使用(PopupWindow默认就是这个Type)
- TYPE_APPLICATION_STARTING: 无法使用(什么也不显示)
- TYPE_APPLICATION_SUB_PANEL: 只能配合Activity在当前APP使用
TYPE_BASE_APPLICATION: 无法使用(什么也不显示)
- TYPE_CHANGED: 只能配合Activity在当前APP使用
- TYPE_INPUT_METHOD: 无法使用(直接崩溃)
- TYPE_INPUT_METHOD_DIALOG: 无法使用(直接崩溃)
- TYPE_KEYGUARD_DIALOG: 拒绝使用
- TYPE_PHONE: 属于悬浮窗(并且给一个Activity的话按下HOME键会出现看不到桌面上的图标异常情况)
- TYPE_TOAST: 不属于悬浮窗，但有悬浮窗的功能，缺点是在Android2.3上无法接收点击事件
- TYPE_SYSTEM_ALERT: 属于悬浮窗，但是会被禁止 ``

尾声

现在我们都知道了如何在不申请权限的情况下显示悬浮窗, 我相信以中国Android开发者的脑洞, 一定会有很多有趣或恶心的功能被开发出来, 一方面我自己觉得这个东西很有用, 可以实现一些很神奇的功能, 另一方面又担心这个API被滥用, 最终不得不限制权限.

还有就是, 逆向分析仅用于学习, 不要干违法的事情.

本人技术有限, 如果文中有错误的欢迎指正, 以免误导他人

利益声明: 虽然我目前在UC实习, 但我并没有UC浏览器中文版的代码权限, 也不会将公司的代码分享给外人. 本文完全是靠我自己开发经验+逆向分析经验+Google完成, 在此之前没有看过UC浏览器的任何代码.

Android逆向实践：使用Smali注入改造YD词典悬浮窗

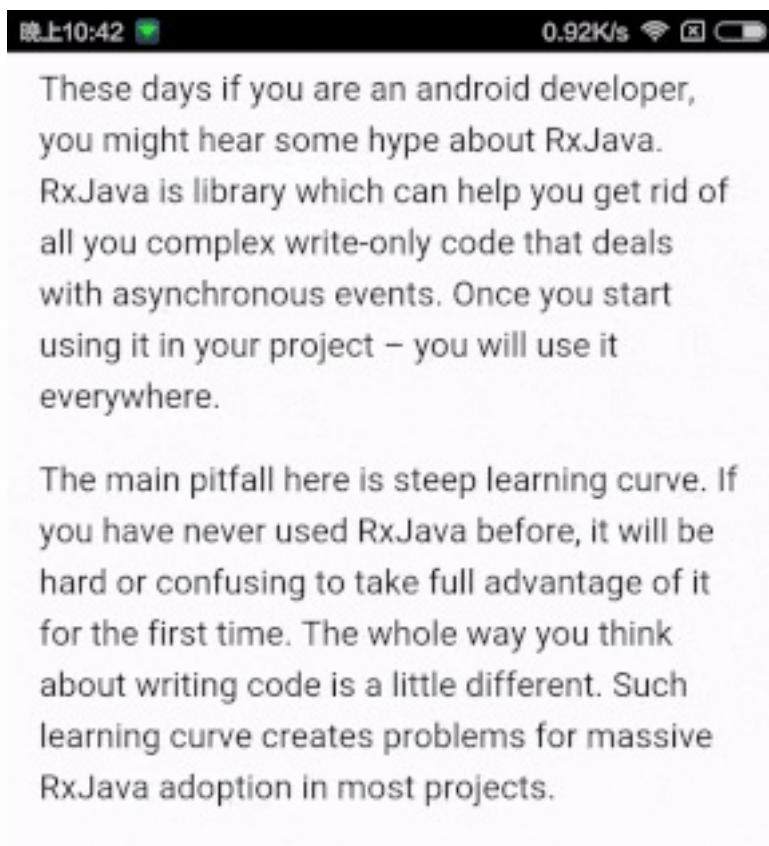
来源：[简书](#)

前言

最近有个开源APP[咕咚翻译](#). 参考我之前在[Android无需权限显示悬浮窗, 兼谈逆向分析app](#)中介绍的一个小的细节, 以悬浮窗的形式做了复制查词功能. 在我写那篇文章之后, 就一直想有这样一个能提供复制查词功能的APP, 无奈自己不知道怎么做一个词典APP, 也就一直没管(主要是懒). 自己平时一直用YD词典, 它也有复制查词功能, 但是YD悬浮窗的交互我觉得特别蛋疼, 每次安装还要把悬浮窗权限手动打开才能用.



几天前下载咕咚翻译试用, 发现了一个崩溃, 顺手改了一下发了pull request. 然后就在想怎么给咕咚翻译的悬浮窗加上交互, 至少能让我主动关闭悬浮窗, 参考了iOS上通知的交互方式, 也就是能往下拉一点点, 还能往上滑动关闭, 无奈好像遇到了Android的bug, 就用了一种奇怪的方式实现, 有一定副作用, 于是没有push到github, 就自己本地用了. 下面故意展示了副作用.



寝室每天都要断电，断电了就没网，咕咚翻译必须联网查词，一到晚上断电就没法用。而YD词典拥有离线复制查词功能，悬浮窗有点蛋疼，凑合凑合也能用。

需求

我的需求是：咕咚翻译能提供离线查词的功能。

这事说起来简单，实际上很复杂，例如离线词典数据从哪来？查词速度如何？怎么管理离线词典数据？如何实现功能？没有找过开源项目，一直用YD词典的复制查词功能，于是我就盯上了YD词典。

不知道怎么实现离线查词，必然需要研究YD词典的实现，在手机上粗略看了一下YD词典在/data/data下的目录结构，大概能确定YD词典的离线查词功能实现在native层，这要我研究到狗年马月。

今天突然来了个奇思妙想，既然YD词典过于庞大，无法剥离离线查词功能，何不将咕咚翻译的悬浮窗“赠与”YD词典，来个移花接木。之前从来没有做过这方面的尝试，但是凭着自己以往的经验，觉得难度不算大，可以在几个小时之内搞定。

可行性

想把咕咚翻译的悬浮窗"赠与"YD词典, 我只想到了一种方案: Smali注入.

我是个懒人, 一个事情太麻烦我就不想做了, Smali注入这个方案看起来很吓人, 实际想想可行性非常高.

观察一下Smali文件的结构:

```
# class信息  
  
# 注解信息  
  
# 实现的接口  
  
# static字段  
  
# 成员字段  
  
# 直接成员方法  
  
# 重写成员方法
```

Smali文件中对外部类的引用使用类似完全限定名的形式.

可以猜测: 如果一个类只依赖Android framework, 不依赖其他自定义类, 那么直接把这个类的Smali文件放到apktool反编译生成的目录中, 不会产生错误.

同时, 假设一个类依赖其他自定义类, 如果把整个依赖关系中涉及的所有非Framework的Smali文件都放入apktool生成的目录中, 同样不会产生错误.

基于以上猜测, 我们可以做到将一个APP中的类安全的添加到另一个APP中.

剩下的就是对被注入APP的Smali代码进行修改, 使得被注入APP调用注入的Smali代码, 且能将信息传递给注入的Smali代码.

观察实现

知道了思路, 就可以实际操作了. 操作前需要了解两个APP的逻辑, 这样才能选择合适的地方进行Smali注入, 减少工作量, 同时减少出错的可能.

咕咚翻译的实现

对咕咚翻译, 我们可以同时观察它的Java代码和Smali代码.

本文粘贴的咕咚翻译代码是我修改的版本, 与github上的源码有区别.

咕咚翻译中悬浮窗使用MVP的设计, 当View被创建时会使用Dagger 2创建Presenter, 同时进行双方的依赖注入.

View的实现类TipView关键代码如下所示:

```
public class TipView extends FrameLayout implements ITipView {

    @Inject
    protected ITIPresenter mPresenter;

    public TipView(Context context) {
        this(context, null);
    }

    public TipView(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }

    public TipView(Context context, AttributeSet attrs, int defStyleAttr) {
        .....
        DaggerTipComponent.builder().tipModule(new TipModule(this)).build().inject(this);
    }

}
```

Presenter的接口如下所示:

```
public interface ITIPresenter {
    void readyForShow();

    void tipShowFully();

    void tipHided();

    void favoriteClicked(Result result);

    void onUserTouch();

    void onTouchOver();
}
```

当TipView中的收藏按钮被点击时, favoriteClicked 方法会被调用. ITIPresenter 的实现类是 TipPresenterImpl

我们需要将TipView的相关类全部注入到YD词典中, 根据前面的分析, 我们的依赖应当越来越少, 否则一旦类的关系没设计好, 一个类可能带起一堆类, 特别麻烦, 所以我把 Dagger 2 相关的依赖注入改成自己直接手写, 同时把收藏按钮去掉, 因为这会导致 TipPresenterImpl 中依赖咕咚翻译Model层, 这会带起一堆依赖, 为了简单直接去掉. 悬浮窗的代码中还使用RxJava相关的东西, 同样需要把悬浮窗中涉及RxJava的部分去掉.

这一步做完, 我们需要直接注入的类就已经变得很清爽了.

此外, 我们还需要知道如何调用TipView, 咕咚翻译里相关代码如下所示:

```
public void show(Result result) {
    TipView tipView = new TipView(mContext);
    tipView.setContent(result);
    tipView.setViewManager(mWindowManager);
    LayoutParams params = getPopViewParams();
    tipView.setLayoutParams(params);
    mWindowManager.addView(tipView, params);
}
```

在这里我们知道如果要调用TipView, 我们需要创建它, 同时获取一个 LayoutParams , 一个 Result , 再通过 WindowManager 完成全部调用.

这是我们需要在YD词典中添加的smali代码的逻辑, 通过这段代码来调用我们注入的 TipView .

再观察上面方法中出现的Result是什么:

```
public class Result {
    .....

    public Result(IResult mIResult) {
        .....
    }

    .....
}
```

这个类可以通过传入一个IResult完成构造, IResult是个接口, 这其实对我们很有利. 假如 YD词典离线查词的结果实现了IResult, 我们就能直接构造一个Result传给TipView了. 因此我们的目的之一就是将YD词典离线查词的结果改造成实现IResult接口. 为此, 我们可以精简IResult, 只保留我们需要的方法, 精简结果如下:

```

public interface IResult {
    List<String> wrapExplains();

    String wrapQuery();

    int wrapErrorCode();

    String wrapPhAm();
}

```

由于我希望精简之后, 咕咚翻译还能正常编译运行, 所以保留了一个多余的 `wrapErrorCode` 方法, 这个方法在Smali注入中没用.

总结一下, 在Smali层面上, 我们需要将TipView的所有代码注入YD词典中, 需要在YD词典中合适的地方编写调用TipView的逻辑, 需要让YD词典离线查词结果实现IResult接口.

YD词典的实现

对YD词典, 我们可以观察的Smali代码, 也可以观察比较接近YD词典源码的Java代码.

关于复制查词的实现, 一个常识是开发者需要在Service中监听剪贴板信息变化, 在手机里观察YD词典的信息, 可以大概猜是哪个Service在监听剪贴板.



根据名字, 可以猜到在ClipboardWatcher中, 建议先用jadx或者dex2jar观察, 毕竟直接看Smali还是太不直观了, 不到必要的时候不用看Smali.

首先看到了如下代码:

```

private class ClipboardListener implements OnPrimaryClipChangedListener {
    .....
    public void onPrimaryClipChanged() {
        .....
        if (ClipboardWatcher.isValidText(clipboardText)) {
            .....
            ClipboardWatcher.this.queryWordViaService(clipboardText);
        }
    }
}

```

比较惊喜的是代码竟然没有混淆, 这下可以节省不少时间, 不用去猜变量的意思了. 接着看 `clipboardWatcher.queryWordViaService` :

```
private void queryWordViaService(String word) {
    if (...) {
        QuickQueryService.show(this, word, 0,
            Util.dip2px(this, BitmapDescriptorFactory.HUE_ORANGE),
            QuickQueryType.COPY_REQ_POPUP);
    }
}
```

非常简单, 还是方法调用, 我们接着看 `QuickQueryService.show`:

```
public static void show(Context context, String word,
                        int screenX, int screenY, QuickQueryType quickQueryType) {
    show(context, word, screenX, screenY, true, true, quickQueryType);
}

private static void show(Context context, String word, int screenX, int screenY,
                        boolean showCloseButton, boolean belowWord,
                        QuickQueryType quickQueryType) {
    Intent intent = new Intent(context, QuickQueryService.class);
    .....
    intent.putExtra(WORD, word);
    .....
    context.startService(intent);
}
```

整个过程很清晰, `clipboardWatcher` 负责监听剪贴板, 当剪贴板内容变化后, 获取其内容, 交给 `QuickQueryService` 处理. 直接去看 `QuickQueryService.onStartCommand`, 十有八九是在这里进行下一步逻辑:

```

public int onStartCommand(Intent intent, int flags, int startId) {
    .....
    try {
        if (...) {
            .....
            String word = intent.getStringExtra(WORD);
            .....
            this.handler.obtainMessage(0, Util.deleteRedundantSpace(word)).sendToTarget();
        } else if (...) {
            .....
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return 2;
}

```

这里不用管其他的逻辑, 只看关键代码, QuickQueryService从intent 中获取需要查询的单词, 交给handler处理, 再看 handler.handleMessage 的逻辑:

```

public void handleMessage(Message msg) {
    try {
        QuickQueryService.this.mainHandler.obtainMessage(0,
            QueryServerManager.getLocalQueryServer().queryWord(msg.obj)).sendToTarget();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

显然, handler应该是一个非主线程的handler, 在这里进行了本地查词相关的工作, 最后把结果交给mainHandler处理, mainHandler.handleMessage 逻辑如下:

```

public void handleMessage(Message msg) {
    try {
        QuickQueryService.this.view.setContent(msg.obj);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

可以猜到view就是YD词典的悬浮窗类, 服务只需要调用它的setContent方法, 剩下的由view自行处理, 这里是复制查词功能整个调用的终点. 如果我们要进行Smali注入, 这个方法是非常不错的注入点. 最后一个问题: 上面的msg.obj是什么?

我们知道这个对象是通过调用 `QueryServerManager.getLocalQueryServer().queryWord` 得到的, 查看这个类的代码可以很容易知道 `msg.obj` 的类型是 `YDLocalDictEntity`, 根据之前的讨论, 我们需要让它实现 `IResult`, 因此我们还要对这个类进行 Smali 注入.

总结一下, 我们需要对 `mainHandler.handleMessage` 注入代码, 让它调用我们的 `TipView`, 需要对 `YDLocalDictEntity` 注入代码, 让它实现 `IResult` 接口. 此外, 由于调用 `TipView` 还需要 `WindowManager` 支持, 因此我们可能还需要对 `QuickQueryService` 进行注入.

实施注入

先用 apktool 反编译 YD 词典 APK. 下面开始进行 Smali 注入.

直接复制 Smali

对于完整的类, 我们不需要手写 Smali 代码, 直接编译一个咕咚翻译 APK, 再用 apktool 反编译, 到对应的路径下把相应的 smali 文件复制到 YD 词典目录下.

注意复制 smali 文件的时候务必要复制全部, 一个 java 文件可能生成不止一个 smali 文件, 例如下面是 `TipView` 对应的全部 smali 文件.



所有引用到的类的 smali 都要复制进去, 且按照原 APK 的包名设置目录并对应放置.

这一步很简单, 仅仅是复制一下就完成了.

修改 Smali

完成了复制, 还需要添加调用代码, 注入代码必须要看 smali 了.

首先对 `mainHandler.handleMessage` 进行注入.

这里有同学可能会去 `QuickQueryService.smali` 里面找代码, 实际上这部分代码不在这个文件里, 而是在 `QuickQueryService$2.smali` 中, 这主要是因为 `mainHandler` 是一个内部类实例, 内部类实例都是在 `class$n.smali` 这种命名的文件里. 要知道具体是哪个文件, 可以看 jadx 中的初始化代码, 代码上都会有注释写清楚真正的代码在哪个文件里, 也可以在 `QuickQueryService.smali` 中直接找到答案, 例如 `QuickQueryService.onCreate` 方法中有如下一段:

```
.method public onCreate()V
    .....
    new-instance v1, Lcom/youdao/dict/services/QuickQueryService$2;
    invoke-direct {v1, p0}, Lcom/youdao/dict/services/QuickQueryService$2;-><init>(Lco
    input-object v1, p0, Lcom/youdao/dict/services/QuickQueryService;->mainHandler:Lan
    .....
    return-void
.end method
```

这就是典型的初始化操作, 创建一个实例 `QuickQueryService$2`, 由`v1`指向它. 随后调用`v1`的方法, 传入参数`p0`, 这个方法完成后对象就构造完毕了, `p0`就是java中的`this`, 之所以内部类能访问外部类的成员, 一部分原因是由于内部类隐式持有了外部类的引用, 这个引用就是在这里被传入的. 最后`v1`的值存入了`p0`的成员`mainHandler`中. 换句话说, 这三句就是初始化`mainHandler`用的, 可知 `mainHandler` 的代码在 `QuickQueryService$2.smali` 中. 直接到 `QuickQueryService$2.smali` 中找 `handleMessage` 方法, 代码如下(可以略过这段smali代码):

```
# virtual methods
.method public handleMessage(Landroid/os/Message;)V
    .locals 3
    .param p1, "msg"    # Landroid/os/Message;

    .prologue
    .line 85
    :try_start_0
    igure-object v1, p1, Landroid/os/Message;->obj:Ljava/lang/Object;

    check-cast v1, Lcom/youdao/dict/model/YDLocalDictEntity;

    .line 86
    .local v1, "entity":Lcom/youdao/dict/model/YDLocalDictEntity;
    igure-object v2, p0, Lcom/youdao/dict/services/QuickQueryService$2;->this$0:Lcom/you

    # getter for: Lcom/youdao/dict/services/QuickQueryService;->view:Lcom/youdao/dict/widget/QuickQueryView;
    invoke-static {v2}, Lcom/youdao/dict/services/QuickQueryService;->access$100(Lcom.you

move-result-object v2

invoke-virtual {v2, v1}, Lcom/youdao/dict/widget/QuickQueryView;->setContent(Lcom.you
:try_end_0
.catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0

.line 90
.end local v1    # "entity":Lcom/youdao/dict/model/YDLocalDictEntity;
:goto_0
return-void

.line 87
:catch_0
move-exception v0

.line 88
.local v0, "e":Ljava/lang/Exception;
invoke-virtual {v0}, Ljava/lang/Exception;->printStackTrace()V

goto :goto_0
.end method
```

别看这段代码这么长，实际上就是下面这段Java代码：

```

public void handleMessage(Message msg) {
    try {
        QuickQueryService.this.view.setContent(msg.obj);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

因为内部类访问外部类实例的本质, 是通过编译器给咱们加的各种合成方法(Synthetic Method)实现的, 所以转换成smali之后特别冗长.

我们要做的就是把这段smali代码改成类似下面的Java代码的效果

```

public void handleMessage(Message msg) {
    TipView tipView = new TipView(mContext);
    Result result = new Result((YDLocalDictEntity) msg.obj);
    tipView.setContent(result);
    tipView.setViewManager(QuickQueryService.this.mWindowManager);
    LayoutParams params = QuickQueryService.getPopViewParams();
    tipView.setLayoutParams(params);
    QuickQueryService.this.mWindowManager.addView(tipView, params);
}

```

但是这段代码的能跑的前提是YDLocalDictEntity实现了IResult, 以及QuickQueryService有一个成员变量mWindowManager和一个静态方法getPopViewParams.

YDLocalDictEntity注入

打开YDLocalDictEntity.smali, 和写Java一样, 一个类要实现一个接口, 需要写implements interface_name, 同时实现方法, smali也类似, 修改后smali如下所示, 添加的代码我重点标出来了:

```

.class public Lcom/youdao/dict/model/YDLocalDictEntity;
.super Ljava/lang/Object;
.source "YDLocalDictEntity.java"

# interfaces
.implements Ljava/io/Serializable;
=====在这里添加要实现的接口=====
.implements Lname/gudong/translate/mvp/model/entity/IResult;
=====

.....

```

```
#=====下面是IResult四个方法的实现=====

.method public wrapQuery()Ljava/lang/String;
.locals 1

.prologue
.iget-object v0, p0, Lcom/youdao/dict/model/YDLocalDictEntity;->word:Ljava/lang/St

return-object v0
.end method

.method public wrapExplains()Ljava/util/List;
.locals 1
.annotation system Ldalvik/annotation/Signature;
    value = {
        "()", 
        "Ljava/util/List", 
        "<", 
        "Ljava/lang/String;", 
        ">;"
    }
.end annotation

.prologue
.iget-object v0, p0, Lcom/youdao/dict/model/YDLocalDictEntity;->translations:Ljava.

return-object v0
.end method

.method public wrapErrorCode()I
.locals 1

.prologue
const v0, 0x0

return v0
.end method

.method public wrapPhAm()Ljava/lang/String;
.locals 1

.prologue
.iget-object v0, p0, Lcom/youdao/dict/model/YDLocalDictEntity;->phoneticUS:Ljava/l

return-object v0
.end method
#=====
```

`iget-object v0, p0, field` 可以理解为 `v0 = p0.field` .

这里 `wrapQuery`, `wrapExplains` 和 `wrapPhAm` 三个方法都只是取当前对象中的一个成员返回, `wrapErrorCode` 纯粹是为了兼容才写入接口的, 直接返回0. 这些代码可以从类似的Java代码对应的smali中修改得到, 也可以直接写, 毕竟这些代码很简单.

QuickQueryService注入

我们需要给 `QuickQueryService` 添加一个 `WindowManager` 成员和一个静态方法, 为了方便代码书写, 全部使用`public`修饰.

`QuickQueryService.smali` 修改后如下:

```

.....
.field private view:Lcom/youdao/dict/widget/QuickQueryView;

#=====添加一个成员 mWindowManager=====
.field public mWindowManager:Landroid/view/WindowManager;
#=====

.....
.method public onCreate()V
    .locals 3
    .....
#=====初始化 mWindowManager=====
    const-string/jumbo v0, "window"

    invoke-virtual {p0, v0}, Landroid/content/Context;.>getSystemService(Ljava/lang/S
    move-result-object v0

    check-cast v0, Landroid/view/WindowManager;

    input-object v0, p0, Lcom/youdao/dict/services/QuickQueryService;.>mWindowManager:
#=====

    .line 108
    return-void
.end method

.....
#====添加静态方法 getPopViewParams=====
.method public static getPopViewParams()Landroid/view/WindowManager$LayoutParams;
    .locals 8

    ...(建议用Java写了反编译复制过来)...
.end method
#=====

```

添加成员可以说是依葫芦画瓢，初始化也很容易写，注意把初始化的代码放到onCreate方法的最下面，因为这个方法无返回值，因此在方法末尾可以随意使用寄存器，不需要操心破坏寄存器里的原始值。静态方法的声明很容易，但是这个方法代码量大，建议用Java写了反编译了复制，这里就不贴了，实在太长了，光看到.locals 8就够吓人了。

mainHandler注入

我们只需要注入mainHandler.handleMessage, 但是因为代码较多, 需要仔细写, 这里我们没有动外层的try-catch, 直接在内层做修改, 注释标明了这块区域:

```

# virtual methods
.method public handleMessage(Landroid/os/Message;)V
    .locals 3
    .param p1, "msg"    # Landroid/os/Message;

    .prologue
    .line 85
:try_start_0
    ige-object v1, p1, Landroid/os/Message;->obj:Ljava/lang/Object;

    check-cast v1, Lcom/youdao/dict/model/YDLocalDictEntity;

    .line 86
    .local v1, "entity":Lcom/youdao/dict/model/YDLocalDictEntity;
    ige-object v2, p0, Lcom/youdao/dict/services/QuickQueryService$2;->this$0:Lcom/y

#前面的代码使得v1是YDLocalDictEntity, v2是QuickQueryService
=====下面是注入代码=====
#v0指向一个Result, 使用v1做参数初始化, v1是YDLocalDictEntity
new-instance v0, Lname/gudong/translate/mvp/model/entity/Result;

invoke-direct {v0, v1}, Lname/gudong/translate/mvp/model/entity/Result;-><init>(L
#将v1改为指向一个TipView, 使用v2做采纳数初始化, v2是QuickQueryService
new-instance v1, Lname/gudong/translate/listener/view/TipView;

invoke-direct {v1, v2}, Lname/gudong/translate/listener/view/TipView;-><init>(L
#下面这句等于v1.setContent(v0)
invoke-virtual {v1, v0}, Lname/gudong/translate/listener/view/TipView;->setContent
#下面这句将v0指向QuickQueryService.this.mWindowManager
ige-object v0, v2, Lcom/youdao/dict/services/QuickQueryService;->mWindowManager:
#等于v1.setViewManager(v0)
invoke-virtual {v1, v0}, Lname/gudong/translate/listener/view/TipView;->setViewMa

invoke-static {}, Lcom/youdao/dict/services/QuickQueryService;->getPopViewParams(
#下面这句将上面方法得到的结果存到v0, 也就是说v0此时是LayoutParams
move-result-object v0
#v1.saveLayoutParams(v0)
invoke-virtual {v1, v0}, Lname/gudong/translate/listener/view/TipView;->saveLayout
# v2是QuickQueryService, 下面这句等于v2 = v2.mWindowManager, 此时v2是mWindowManager
ige-object v2, v2, Lcom/youdao/dict/services/QuickQueryService;->mWindowManager:
#等于v2.addView(v1, v0)
invoke-interface {v2, v1, v0}, Landroid/view/WindowManager;->addView(Landroid/vie

=====上面是注入代码=====

:try_end_0
.catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0

```

```

.line 90
.end local v1    # "entity":Lcom/youdao/dict/model/YDLocalDictEntity;
:goto_0
return-void

.line 87
:catch_0
move-exception v0

.line 88
.local v0, "e":Ljava/lang/Exception;
invoke-virtual {v0}, Ljava/lang/Exception;->printStackTrace()V

goto :goto_0
.end method

```

如果不明白注入的代码, 可以看我写的注释, 总体上来看还是很简单的. 当然, 我手写的代码的效率没有生成的高.

添加资源

由于TipView中会使用layout, 所以还需要把layout下的文件放到YD词典目录的对应位置. 而且需要自己在res/values/ids.xml和res/values/public.xml中添加一些内容. 如果layout中引用了drawable, 还需要把对应的drawable放到YD词典目录的对应位置. 引用了color, dimen等的, 都需要添加对应的定义.

apktool在打包的时候会用aapt来帮助生成id, 但是实际上smali文件中已经没有对R文件的引用了, 全是常量, 所以对于代码中直接使用的R.id.name, 需要我们自己到ids.xml中添加id, 然后到public.xml中指定好唯一的值, 再把smali中的常量替换成我们定义的, 对于layout, 只需要到public.xml中指定好值, 把smali中R.layout.name换成我们指定的值就行. 其他的如color, dimen的, aapt会自动帮我们生成id. 但如果直接在代码中使用了, 还是要和layout一样, 自己去定义.

例如我在ids.xml中添加了如下内容:

```

<item type="id" name="pop_view_content_all">false</item>
<item type="id" name="pop_view_content_without_shadow">false</item>
<item type="id" name="ll_pop_src">false</item>
<item type="id" name="tv_pop_src">false</item>
<item type="id" name="tv_pop_phonetic">false</item>
<item type="id" name="ll_pop_dst">false</item>
<item type="id" name="tv_point">false</item>

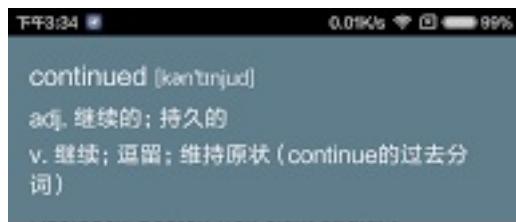
```

在public.xml中添加了如下内容：

```
<public type="layout" name="pop_view" id="0x7f0301a7" />
<public type="id" name="pop_view_content_all" id="0x7f0d0629" />
<public type="id" name="pop_view_content_without_shadow" id="0x7f0d062a" />
<public type="id" name="ll_pop_src" id="0x7f0d062b" />
<public type="id" name="tv_pop_src" id="0x7f0d062c" />
<public type="id" name="tv_pop_phonetic" id="0x7f0d062d" />
<public type="id" name="ll_pop_dst" id="0x7f0d062e" />
<public type="id" name="tv_point" id="0x7f0d062f" />
```

签名与安装

最后使用apktool打包，使用jarsigner签名，就可以安装到手机上了，YD词典的功能均可用，同时悬浮窗被替换成了咕咚翻译的悬浮窗。



championships in the NBA from 1959 to 1966. This championship streak is the longest in NBA history. They did not win the title in 1966–67, but regained it in the 1967–68 season and repeated in 1969. The domination totaled nine of the ten championship banners of the 1960s.^[14]

Through this period, the NBA continued to strengthen with the shift of the Minneapolis Lakers to Los Angeles, the Philadelphia Warriors to San Francisco, the Syracuse Nationals to Philadelphia to become the Philadelphia 76ers, and the St. Louis Hawks moving to Atlanta, as well as the addition of its first expansion franchises. The Chicago Packers (now Washington Wizards) became the ninth NBA team in 1961. From 1966 to 1968, the league expanded from 9 to 14



这个YD词典给我的印象一直是卡卡的, 用着还行, 这次逆向顺便把它的硬件加速开了, 流畅很多, 也不知道这个APP还有没有人维护, 怎么连硬件加速都不愿意开. 用Smali注入给它换个悬浮窗本来只是一个想法, 感觉这个想法挺有意思的, 就试了一下, 花了8小时才做出来, 现在手机复制查词爽多了.

理解Android中垃圾回收日志信息

链接地址:[技术小黑屋](#)

如果你是一名Android开发者并且常常看程序日志的话，那么下面的这些信息对你来说可能一点都不陌生。

```
GC_CONCURRENT freed 178K, 41% free 3673K/6151K, external 0K/0K, paused 2ms+2ms  
GC_EXPLICIT freed 6K, 41% free 3667K/6151K, external 0K/0K, paused 29ms  
GC_CONCURRENT freed 379K, 42% free 3856K/6535K, external 0K/0K, paused 2ms+3ms  
GC_EXPLICIT freed 144K, 41% free 3898K/6535K, external 0K/0K, paused 32ms  
GC_CONCURRENT freed 334K, 40% free 4091K/6727K, external 0K/0K, paused 2ms+3ms
```

但是这些到底是什么，又有什么含义呢？

上面的这几行就是Android系统垃圾回收的部分输出信息。每当垃圾回收被触发的时候，你就可以通过logcat查看到这样的信息。这样短短的一行的日志有着很大的信息量。比如通过日志我们可以发现程序可能有内存（泄露）问题。本文将具体介绍这些日志信息的每一部分的含义来帮助帮助大家更好地了解垃圾回收的运行情况。

原因

GC_CONCURRENT freed 178K, 41% free 3673K/6151K, external 0K/0K, paused 2ms+2ms **GC_EXPLICIT** freed 6K, 41% free 3667K/6151K, external 0K/0K, paused 29ms

红颜色标出的部分就是垃圾回收触发的原因。在Android中有五种类型的垃圾回收触发原因。

- **GC_CONCURRENT** 当堆内存增长到一定程度时会触发。此时触发可以对堆中的没有用的对象及时进行回收，腾出空间供新的对象申请，避免进行不必要的增大堆内存的操作。
- **GC_EXPLICIT** 当程序中调用System.gc()方法触发。这个方法应避免出现在程序中调用。因为JVM有足够的能力来控制垃圾回收。
- **GC_EXTERNAL_MALLOC** 当Bitmap和NIO Direct ByteBuffer对象分配外部存储

(机器内存，非Dalvik堆内存) 触发。这个日志只有在2.3之前存在，从2.3系统开始，垃圾回收进行了调整，前面的对象都会存储到Dalvik堆内存中。所以在2.3系统之后，你就再也不会看到这种信息了。

- **GC_FOR_MALLOC** 当堆内存已满，系统需要更多内存的时候触发。这条日志出现后意味着JVM要暂停你的程序进行垃圾回收操作。
- **GC_HPROF_DUMP_HEAP** 当创建一个内存分析文件HPROF时触发。

结果

JVM内存

```
GC_CONCURRENT freed 178K, 41% free 3673K/6151K, external 0K/0K, paused  
2ms+2ms GC_EXPLICIT freed 6K, 41% free 3667K/6151K, external 0K/0K, paused  
29ms
```

这部分数据告诉我们JVM进行垃圾回收释放了多少空间。

堆内存数据

```
GC_CONCURRENT freed 178K, 41% free 3673K/6151K, external 0K/0K, paused  
2ms+2ms GC_EXPLICIT freed 6K, 41% free 3667K/6151K, external 0K/0K, paused  
29ms
```

这部分告诉我们堆内存中可用内存占的比例，当前活跃的对象总的空间，以及当前堆的总大小。所以这里的数据就是41%的堆内存可用，已经使用了3673K，总的堆内存大小为6151K。

外部存储数据

```
GC_EXTERNAL_ALLOC freed 1125K, 47% free 6310K/11847K, external  
1051K/1103K, paused 46ms GC_EXTERNAL_ALLOC freed 295K, 47% free  
6335K/11847K, external 1613K/1651K, paused 41ms
```

这部分数据告诉我们外部存储（位于机器内存）对象的数据。在2.3之前，bitmap对象存放在机器内存。因此在第一条数据中我们可以看到以有1051K使用，外部存储为1103K。

上面两行数据相差100毫秒，我们可以看到第一条数据表明外部存储快满了，由于GC_EXTERNAL_ALLOC被触发，外部存储空间扩大到了1651K。

垃圾回收暂停时间

GC_CONCURRENT freed 178K, 41% free 3673K/6151K, external 0K/0K, **paused 2ms+2ms** GC_EXPLICIT freed 6K, 41% free 3667K/6151K, external 0K/0K, **paused 29ms**

这部分数据表明垃圾回收消耗的时间。在GC_CONCURRENT回收时，你会发现两个暂停时间。一个是在回收开始的暂停时间，另一个时在回收结束的暂停时间。

GC_CONCURRENT从2.3开始引入，相比之前的程序全部暂停的垃圾回收机制，它的暂停时间要小的多。一般少于5毫秒。因为GC_CONCURRENT的绝大多数操作在一个单独的线程中进行。

组件

Android中Notification捕捉点击事件的替代方式

来源:<http://blog.csdn.net/liuweiballack/article/details/48009109>

在处理程序中的通知消息时，一般都是用Notification类来处理，通过设置PendingIntent来处理点击通知之后的动作。与一般的Intent不同，PendingIntent表示即将要执行的动作，是在用户点击消息之后才进行处理，它里面保存了一个Intent用来执行跳转的操作。

但是有一些需求，要求在用户点击通知之后，还需要执行一些其他的操作，并非单纯的进行activity之间的跳转。因此需要对notification的点击事件进行捕捉，但是Android中并没有提供这样的API供开发者使用。我们无法对系统通知的点击事件进行捕捉，这就需要找到一种其他的方式解决这个问题。

在PendingIntent中提供了一个方法getBroadcast()，我们可以通过广播的方式进行处理：当用户点击通知时，发送一个广播，当广播接收者收到这个广播时，在进行对应的逻辑处理。

具体代码如下：

定义方法showNotification

```
private void showNotification() {
    // 创建一个NotificationManager的引用
    NotificationManager notificationManager = (NotificationManager) this.getSystemService(Context.NOTIFICATION_SERVICE);

    // 定义Notification的各种属性
    Notification notification = new Notification(R.drawable.ic_launcher, "新消息",
        NotificationCompat.Builder builder = new NotificationCompat.Builder(mContext)
        builder.setSmallIcon(R.drawable.ic_launcher);
    notification.flags |= Notification.FLAG_AUTO_CANCEL;

    // 设置通知的事件消息
    CharSequence contentTitle = "Title"; // 通知栏标题
    CharSequence contentText = "Text"; // 通知栏内容

    Intent clickIntent = new Intent(mContext, NotificationClickReceiver.class);
    clickIntent.putExtra(Notification.EXTRA_NOTIFICATION_ID, id);
    PendingIntent contentIntent = PendingIntent.getBroadcast(this.getApplicationContext(), id, clickIntent, 0);
    notification.setLatestEventInfo(this, contentTitle, contentText, contentIntent);
    notificationManager.notify(id, notification);
}
```

定义NotificationClickReceiver继承 BroadcastReceiver， NotificationClickReceiver需要在 AndroidManifest.xml中注册或者使用其他方法注册

```
public class NotificationClickReceiver extends BroadcastReceiver {  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        //todo 跳转之前要处理的逻辑  
        Intent newIntent = new Intent(context, ActivityMain.class).addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);  
        context.startActivity(newIntent);  
    }  
}
```

在用户点击通知之后， NotificationClickReceiver 会接受到通知的点击广播，执行 onReceive()方法。

注意：

- 1、onReceive () 方法中不能执行耗时操作。
- 2、在onReceive () 的context中启动activity， 需要加上 Intent.FLAG_ACTIVITY_NEW_TASK， 否则会崩溃。

经验

Android开发中，那些让你相见恨晚的方法、类或接口

来源:[liukun的个人博客](#)

PS:本文类容来自我在[知乎上对Android开发中，有哪些让你觉得相见恨晚的方法、类或接口？](#)这一问题的回答，目前就总结这些，日后若有新的发现，随时补充。欢淫点赞。

- `getParent().requestDisallowInterceptTouchEvent(true);`剥夺父view 对touch 事件的处理权，谁用谁知道。
- `ArgbEvaluator.evaluate(float fraction, Object startValue, Object endValue);` 用于根据一个起始颜色值和一个结束颜色值以及一个偏移量生成一个新的颜色，分分钟实现类似于微信底部栏滑动颜色渐变。
- Canvas中`clipRect`、`clipPath`和`clipRegion` 剪切区域的API。

第3个有个坑，就是有些clip貌似不支持硬件加速...(也有可能我测试机不多，安卓api demo里面的canvas clip那个Activity是在Activity注册的时候加了使用软件绘图的属性，当年照着demo写clipPath一直没有效果...后来发现要关闭硬件加速才行，被这个坑了好久... 还有就是刚出的design包里的FAB，手动设置成软件绘图在4.x下才会正常显示阴影...)

- `Bitmap.extractAlpha();`返回一个新的Bitmap，capture原始图片的alpha 值。有的时候我们需要动态的修改一个元素的背景图片又不希望使用多张图片的时候，通过这个方法，结合Canvas 和Paint 可以动态的修改一个纯色Bitmap的颜色。
- HandlerThread，代替不停new Thread 开子线程的重复体力写法。
- IntentService,一个可以干完活后自己去死且不需要我们去管理子线程的Service。
- Palette，5.0加入的可以提取一个Bitmap 中突出颜色的类，结合上面的 `Bitmap.extractAlpha`，你懂的。
- Executors. newSingleThreadExecutor();这个是java 的，之前不知道它，自己花很大功夫去研究了单线程顺序执行的任务队列。。
- `android:animateLayoutChanges="true"`， LinearLayout中添加View 的动画的办法，支持通过`setLayoutTransition()`自定义动画。

- ViewDragHelper，自定义一个子View可拖拽的ViewGroup时，处理各种事件很累吧，嗯？what the fuck!!
- GradientDrawable，之前接手公司的项目，发现有个阴影效果还不错，以为是切的图片，一看代码，什么鬼= =！
- AsyncQueryHandler，如果做系统工具类的开发，比如联系人短信辅助工具等，肯定免不了和ContentProvider打交道，如果数据量不是很大的情况下，随便搞，如果数据量大的情况下，了解下这个类是很有必要的，需要注意的是，这玩意儿吃异常..
- ViewFlipper，实现多个view的切换(循环)，可自定义动画效果，且可针对单个切换指定动画。
- 有朋友提到了在自定义View时有些方法在开启硬件加速的时候没有效果的问题，在API16之后确实有很多方法不支持硬件加速，通常我们关闭硬件加速都是在清单文件中通过，其实android也提供了针对特定View关闭硬件加速的方法，调用View.setLayerType(View.LAYER_TYPE_SOFTWARE, null);即可。
- android util包中的Pair类，可以方便的用来存储一”组”数据。注意不是key value。
- PointF，graphics包中的一个类，我们经常见到在处理Touch事件的时候分别定义一个downX，一个downY用来存储一个坐标，如果坐标少还好，如果要记录的坐标过多那代码就不好看了。用PointF(float x, float y);来描述一个坐标点会清楚很多。
- StateListDrawable，定义Selector通常的办法都是xml文件，但是有的时候我们的图片资源可能是从服务器动态获取的，比如很多app所谓的皮肤，这种时候就只能通过StateListDrawable来完成了，各种addState即可。
- android:descendantFocusability，ListView的item中CheckBox等元素抢焦点导致item点击事件无法响应时，除了给对应的元素设置focusable，更简单的是在item根布局加上android:descendantFocusability="blocksDescendants"
- android:duplicateParentState="true"，让子View跟随其Parent的状态，如pressed等。常见的使用场景是某些时候一个按钮很小，我们想要扩大其点击区域的时候通常会再给其包裹一层布局，将点击事件写到Parent上，这时候如果希望被包裹按钮的点击效果对应的Selector继续生效的话，这时候duplicateParentState就派上用场了。
- includeFontPadding="false"，TextView默认上下是有一定的padding的，有时候我们可能不需要上下这部分留白，加上它即可。
- Messenger，面试的时候通常都会被问到进程间通信，一般情况下大家都是开始背书，AIDL巴拉巴拉。。有一天在鸿神的博客看到这个，嗯，如他所说，又可以装一下了。

- TextView.setError();用于验证用户输入。
 - ViewConfiguration.getScaledTouchSlop();触发移动事件的最小距离，自定义View处理touch事件的时候，有的时候需要判断用户是否真的存在move，系统提供了这样的方法。
 - ValueAnimator.reverse(); 顺畅的取消动画效果。
 - ViewStub，有的时候一块区域需要根据情况显示不同的布局，通常我们都会通过setVisibility的方法来显示和隐藏不同的布局，但是这样默认是全部加载的，用ViewStub可以更好的提升性能。
 - onTrimMemory，在Activity中重写此方法，会在内存紧张的时候回调（支持多个级别），便于我们主动的进行资源释放，避免OOM。
 - EditTxt.setImeOptions，使用EditText弹出软键盘时，修改回车键的显示内容(一直很讨厌用回车键来交互，所以之前一直不知道这玩意儿)
 - TextView.setCompoundDrawablePadding，代码设置TextView的drawable padding。
 - ImageSwitcher，可以用来做图片切换的一个类，类似于幻灯片。
 - WeakHashMap，直接使用HashMap有时候会带来内存溢出的风险，使用WeakHashMap实例化Map。当使用者不再有对象引用的时候，WeakHashMap将自动被移除对应Key值的对象。
-

评论内容1

作者：StephenLee 链接：<https://www.zhihu.com/question/33636939/answer/57171337>
来源：知乎 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

抛砖引玉一下，怎么都没什么人？

- 1、Throwable接口中的getStackTrace()方法（或者Thread类的getStackTrace()方法），根据这个方法可以得到函数的逐层调用地址，其返回值为StackTraceElement[]；
- 2、StackTraceElement类，其中四个方法getClassName(), getFileNames(), getLineNumber(), getMethodName()在调试程序打印Log时非常有用；

- 3、UncaughtExceptionHandler接口，再好的代码异常难免，利用此接口可以对未捕获的异常善后； 使用参见：Android使用UncaughtExceptionHandler捕获全局异常
- 4、Resources类中的getIdentifier(name, defType, defPackage)方法，根据资源名称获取其ID，做UI时经常用到；
- 5、View中的isShown()方法，以前都是用view.getVisibility() == View.VISIBLE来判断的(╯□╰)；（谢评论提醒，这里面其实有一个坑：【android】 view.isShown ()的用法）
- 6、Arrays类中的一系列关于数组操作的工具方法：binarySearch(), asList(), equals(), sort(), toString(), copyOfRange()等； Collections类中的一系列关于集合操作的工具方法：sort(), reverse()等；
- 7、android.text.format.Formatter类中formatFileSize(Context, long)方法，用来格式化文件Size (B → KB → MB → GB)；
- 8、android.media.ThumbnailUtils类，用来获取媒体（图片、视频）缩略图；
- 9、String类中的format(String, Object...)方法，用来格式化strings.xml中的字符串
(多谢 @droider An 提示：Context类中getString(int, Object...)方法用起来更加方便)；
- 10、View类中的三个方法：callOnClick(), performClick(), performLongClick(), 用于触发View的点击事件；
- 11、TextUtils类中的isEmpty(CharSequence)方法，判断字符串是否为null或""；
- 12、TextView类中的append(CharSequence)方法，添加文本。一些特殊文本直接用+连接会变成String；
- 13、View类中的getDrawingCache()等一系列方法，目前只知道可以用来截图；
- 14、DecimalFormat类，用于字串格式化包括指定位数、百分数、科学计数法等；
- 15、System类中的arraycopy(src, srcPos, dest, destPos, length)方法，用来copy数组；
- 16、Fragment类中的onHiddenChanged(boolean)方法，使用FragmentTransaction中的hide(), show()时貌似Fragment的其它生命周期方法都不会被调用，太坑爹！
- 17、Activity类中的onWindowFocusChanged(boolean), onNewIntent(intent)等回调方法；

- 18、View类中的getLocationInWindow(int[])方法和getLocationOnScreen(int[])方法，获取View在窗口/屏幕中的位置；
 - 19、TextView类中的setTransformationMethod(TransformationMethod)方法，可用 来实现“显示密码”功能；
 - 20、TextWatcher接口，用来监听文本输入框内容的改变，可用来实现一系列具有特 殊功能的文本输入框；
 - 21、View类中的setSelected(boolean)方法结合android:state_selected=""用来实现图 片选中效果；
 - 22、Surface设置透明：SurfaceView.setZOrderOnTop(true);
SurfaceView.getHolder().setFormat(PixelFormat.TRANSLUCENT);但是会挡住其它 控件；
 - 23、ListView或GridView类中的setFastScrollEnabled(boolean)方法，用来设置快速 滚动滑块是否可见，当然前提是item够多；
 - 24、PageTransformer接口，来自定义ViewPager页面切换动画，用 setPageTransformer(boolean, PageTransformer)方法来进行设置；
 - 25、apache提供的一系列jar包：commons-lang.jar, commons-collections.jar, commons-beanutils.jar等，里面很多方法可能是你曾经用几十几百行代码实现过的，但是执行效率或许要差很多，比如：ArrayUtils, StringUtils.....；
 - 26、AndroidTestCase类，Android单元测试，在AndroidStudio中使用非常方便；
 - 27、TextView类的setKeyListener(KeyListener)方法；其中DigitsKeyListener类，使 用getInstance(String accepted)方法即可指定EditText可输入字符集；
 - 28、ActivityLifecycleCallbacks接口，用于在Application类中监听各Activity的状态变 化；
 - 29、Context类中的createPackageContext(packageName, flags)方法，可用来获取 指定包名应用程序的Context对象。
-

评论内容2

作者：Rocko

链接：<https://www.zhihu.com/question/33636939/answer/57297329>

来源：知乎

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

首先呼应题目，`Log.wtf()`

Part 1：

- `Activity.startActivities()` 常用于在应用程序中间启动其他的Activity。
- `TextUtils.isEmpty()` 简单的工具类,用于检测是否为空。
- `Html.fromHtml()` 用于生成一个Html,参数可以是一个字符串.个人认为它不是很快,所以我不怎么经常去用. (我说不经常用它是为了重点突出这句话：请多手动构建 Spannable 来替换 `Html.fromHtml()`) , 但是它对渲染从 web 上获取的文字还是很不错的。
- `TextView.setError()` 在验证用户输入的时候很棒。
- `Build.VERSION_CODES` 这个标明了当前的版本号,在处理兼容性问题的时候经常会用到.点进去可以看到各个版本的不同特性。
- `Log.getStackTraceString()` 方便的日志类工具,方法`Log.v()`、`Log.d()`、`Log.i()`、`Log.w()`和`Log.e()`都是将信息打印到LogCat中, 有时候需要将出错的信息插入到数据库或一个自定义的日志文件中, 那么这种情况就需要将出错的信息以字符串的形式返回来, 也就是使用`static String getStackTraceString(Throwable tr)`方法的时候。
- `LayoutInflater.from()` 顾名思义,用于Inflate一个layout,参数是layout的id.这个经常写Adapter的人会用的比较多。
- `ViewConfiguration.getScaledTouchSlop()` 使用 `ViewConfiguration` 中提供的值以保证所有触摸的交互都是统一的。这个方法获取的值表示:用户的手滑动这个距离后,才判定为正在进行滑动.当然这个值也可以自己来决定.但是为了一致性,还是使用标准的值较好。
- `PhoneNumberUtils.convertKeypadLettersToDigits` 顾名思义.将字母转换为数字,类似于T9输入法,
- `Context.getCacheDir()` 获取缓存数据文件夹的路径,很简单但是知道的人不多,这个路径通常在SD卡上(这里的SD卡指的是广义上的SD卡,包括外部存储和内部存储)Android/data/您的应用程序包名/cache/ 下面.测试的时候,可以去这里面看是否缓存成功.缓存在这里的好处是:不用自己再去手动创建文件夹,不用担心用户把自己创建的文件夹删掉,在应用程序卸载的时候,这里会被清空,使用第三方的清理工具的时候,这里也会被清空。
- `ArgbEvaluator` 用于处理颜色的渐变。就像 Chris Banes 说的一样, 这个类会进行很多自动装箱的操作, 所以最好还是去掉它的逻辑自己去实现它。这个没用过,不明其所以然,回头再补充.

- ContextThemeWrapper 方便在运行的时候修改主题。
- Space space是Android 4.0中新增的一个控件，它实际上可以用来分隔不同的控件，其中形成一个空白的区域。这是一个轻量级的视图组件，它可以跳过Draw，对于需要占位符的任何场景来说都是很棒的。
- ValueAnimator.reverse() 这个方法可以很顺利地取消正在运行的动画。

Part 2:

- DateUtils.formatDateTime() 用来进行区域格式化工作，输出格式化和本地化的时间或者日期。
- AlarmManager.setInexactRepeating 通过闹铃分组的方式省电，即使你只调用了一个闹钟，这也是一个好的选择，（可以确保在使用完毕时自动调用 AlarmManager.cancel ()）。原文说的比较抽象，这里详细说一下：setInexactRepeating 指的是设置非准确闹钟，使用方法：alarmManager.setInexactRepeating(AlarmManager.RTC, startTime, intervalL, pendingIntent)，非准确闹钟只能保证大致的时间间隔，但是不一定准确，可能出现设置间隔为30分钟，但是实际上一次间隔20分钟，另一次间隔40分钟。它的最大的好处是可以合并闹钟事件，比如间隔设置每30分钟一次，不唤醒休眠，在休眠8小时后已经积累了16个闹钟事件，而在手机被唤醒的时候，非准时闹钟可以把16个事件合并为一个，所以这么看来，非准时闹钟一般来说比较节约能源。
- Formatter.formatFileSize() 一个区域化的文件大小格式化工具。通俗来说就是把大小转换为MB, G, KB之类的字符串。
- ActionBar.hide()/show() 顾名思义，隐藏和显示ActionBar，可以优雅地在全屏和带ActionBar之间转换。
- Linkify.addLinks() 在Text上添加链接。很实用。
- StaticLayout 在自定义 View 中渲染文字的时候很实用。
- Activity.onBackPressed() 很方便的管理back键的方法，有时候需要自己控制返回键的事件的时候，可以重写一下。比如加入“点两下back键退出”功能。
- GestureDetector 用来监听和相对应的手势事件，比如点击，长按，慢滑动，快滑动，用起来很简单，比你自己实现要方便许多。
- DrawFilter 可以让你在不调用onDraw方法的情况下，操作canvas，比了个如，你可以在创建自定义 View 的时候设置一个 DrawFilter，给父 View 里面的所有 View 设置反别名。
- ActivityManager.getMemoryClass() 告诉你你的机器还有多少内存，在计算缓存大小的时候会比较有用。
- ViewStub 它是一个初始化不做任何事情的 View，但是之后可以载入一个布局文件。在慢加载 View 中很适合做占位符。唯一的缺点就是不支持标签，所以如果你不太小

心的话，可能会在视图结构中加入不需要的嵌套。

- `SystemClock.sleep()` 这个方法在保证一定时间的 `sleep` 时很方便，通常我用来进行 `debug` 和模拟网络延时。
- `DisplayMetrics.density` 这个方法你可以获取设备像素密度，大部分时候最好让系统来自动进行缩放资源之类的操作，但是有时候控制的效果会更好一些。(尤其是在自定义 `View` 的时候)。
- `Pair.create()` 方便构建类和构造器的方法。

Part 3:

- `UrlQuerySanitizer`——使用这个工具可以方便对 URL 进行检查。
- `Fragment.setArguments`——因为在构建 `Fragment` 的时候不能加参数，所以这是个很好的东西，可以在创建 `Fragment` 之前设置参数（即使在 `configuration` 改变的时候仍然会导致销毁/重建）。
- `DialogFragment.setShowsDialog()`——这是一个很巧妙的方式，`DialogFragment` 可以作为正常的 `Fragment` 显示！这里可以让 `Fragment` 承担双重任务。我通常在创建 `Fragment` 的时候把 `onCreateView()` 和 `onCreateDialog()` 都加上，就可以创建一个具有双重目的的 `Fragment`。
- `FragmentManager.enableDebugLogging()`——在需要观察 `Fragment` 状态的时候会有帮助。
- `LocalBroadcastManager`——这个会比全局的 `broadcast` 更加安全，简单，快速。像 `otto` 这样的 `Event buses` 机制对你的应用场景更加有用。
- `PhoneNumberUtils.formatNumber()`——顾名思义，这是对数字进行格式化操作的时候用的。
- `Region.op()`——我现在对比两个渲染之前的区域的时候很实用，如果你有两条路径，那么怎么知道它们是不是会重叠呢？使用这个方法就可以做到。
- `Application.registerActivityLifecycleCallbacks`——虽然缺少官方文档解释，不过我想它就是注册 `Activity` 的生命周期的一些回调方法（顾名思义），就是一个方便的工具。
- `versionNameSuffix`——这个 `gradle` 设置可以让你在基于不同构建类型的 `manifest` 中修改版本名这个属性，例如，如果需要在在 `debug` 版本中以“-SNAPSHOT”结尾，那么就可以轻松的看出当前是 `debug` 版还是 `release` 版。
- `CursorJoiner`——如果你是只使用一个数据库的话，使用 SQL 中的 `join` 就可以了，但是如果收到的数据是来自两个独立的 `ContentProvider`，那么 `CursorJoiner` 就很实用了。
- `Genymotion`——一个非常快的 Android 模拟器，本人一直在用。
- `-nodpi`——在没有特别定义的情况下，很多修饰符(`-mdpi,-hdpi,-xdpi`等等)都会默认自

动缩放 assets/dimensions，有时候我们需要保持显示一致，这种情况下就可以使用 -nodpi。

- BroadcastRecevier.setDebugUnregister ()——又一个方便的调试工具。
- Activity.recreate ()——强制让 Activity 重建。
- PackageManager.checkSignatures ()——如果同时安装了两个 app 的话，可以用这个方法检查。如果不进行签名检查的话，其他人可以轻易通过使用一样的包名来模仿你的 app。

Part 4:

- Activity.isChangingConfigurations ()——如果在 Activity 中 configuration 会经常改变的话，使用这个方法就可以不用手动做保存状态的工作了。
- SearchRecentSuggestionsProvider——可以创建最近提示效果的 provider，是一个简单快速的方法。
- ViewTreeObserver——这是一个很棒的工具。可以进入到 View 里面，并监控 View 结构的各种状态，通常我都用来做 View 的测量操作（自定义视图中经常用到）。
- org.gradle.daemon=true——这句话可以帮助减少 Gradle 构建的时间，仅在命令行编译的时候用到，因为 Android Studio 已经这样使用了。
- DatabaseUtils——一个包含各种数据库操作的使用工具。
- android:weightSum (LinearLayout)——如果想使用 layout weights，但是却不想填充整个 LinearLayout 的话，就可以用 weightSum 来定义总的 weight 大小。
- android:duplicateParentState (View)——此方法可以使得子 View 可以复制父 View 的状态。比如如果一个 ViewGroup 是可点击的，那么可以用这个方法在它被点击的时候让它的子 View 都改变状态。
- android:clipChildren (ViewGroup)——如果此属性设置为不可用，那么 ViewGroup 的子 View 在绘制的时候会超出它的范围，在做动画的时候需要用到。
- android:fillViewport (ScrollView)——在这片文章中有详细介绍文章链接，可以解决在 ScrollView 中当内容不足的时候填不满屏幕的问题。
- android:tileMode (BitmapDrawable)——可以指定图片使用重复填充的模式。
- android:enterFadeDuration/android:exitFadeDuration (Drawables)——此属性在 Drawable 具有多种状态的时候，可以定义它展示前的淡入淡出效果。
- android:scaleType (ImageView)——定义在 ImageView 中怎么缩放/剪裁图片，一般用的比较多的是“centerCrop”和“centerInside”。
- Merge——此标签可以在另一个布局文件中包含别的布局文件，而不用再新建一个 ViewGroup，对于自定义 ViewGroup 的时候也需要用到；可以通过载入一个带有标签的布局文件来自动定义它的子部件。
- AtomicFile——通过使用备份文件进行文件的原子化操作。这个知识点之前我也写

过，不过最好还是有出一个官方的版本比较好。

Part 5:

- `ViewDragHelper`——视图拖动是一个比较复杂的问题。这个类可以帮助解决不少问题。如果你需要一个例子，`DrawerLayout`就是利用它实现扫滑。Flavient Laurent 还写了一些关于这方面的优秀文章。
- `PopupWindow`——Android到处都在使用`PopupWindow`，甚至你都没有意识到（标题导航条`ActionBar`，自动补全`AutoComplete`，编辑框错误提醒`Edittext Errors`）。这个类是创建浮层内容的主要方法。
- `Actionbar.getThemrContext()`——导航栏的主题化是很复杂的（不同于Activity其他部分的主题化）。你可以得到一个上下文（`Context`），用这个上下文创建的自定义组件可以得到正确的主题。
- `ThumbnailUtils`——帮助创建缩略图。通常我都是用现有的图片加载库（比如，`Picasso`或者`Volley`），不过这个`ThumbnailUtils`可以创建视频缩略图。译者注：该API从V8才开始支持。
- `Context.getExternalFilesDir()`——申请了SD卡写权限后，你可以在SD的任何地方写数据，把你的数据写在设计好的合适位置会更加有礼貌。这样数据可以及时被清理，也会有更好的用户体验。此外，Android 4.0 Kitkat中在这个文件夹下写数据是不需要权限的，每个用户有自己的独立的数据存储路径。译者注：该API从V8才开始支持。
- `SparseArray`——`Map`的高效优化版本。推荐了解姐妹类`SparseBooleanArray`、`SparseIntArray`和`SparseLongArray`。
- `PackageManager.setComponentEnabledSetting()`——可以用来启动或者禁用程序清单中的组件。对于关闭不需要的功能组件是非常赞的，比如关掉一个当前不用的广播接收器。
- `SQLiteDatabase.yieldIfContendedSafely()`——让你暂时停止一个数据库事务，这样你就可以就不会占用太多的系统资源。
- `Environment.getExternalStoragePublicDirectory()`——还是那句话，用户期望在SD卡上得到统一的用户体验。用这个方法可以获得在用户设备上放置指定类型文件（音乐、图片等）的正确目录。
- `View.generateViewId()`——每次我都想要推荐动态生成控件的ID。需要注意的是，不要和已经存在的控件ID或者其他已经生成的控件ID重复。
- `ActivityManager.clearApplicationUserData()`——一键清理你的app产生的用户数据，可能是做用户退出登录功能，有史以来最简单的方式了。
- `Context.createConfigurationContext()`——自定义你的配置环境信息。我通常会遇到这样的问题：强制让一部分显示在某个特定的环境下（倒不是我一直这样瞎整，说来

话长，你很难理解）。用这个实现起来可以稍微简单一点。

- ActivityOptions ——方便的定义两个Activity切换的动画。使用ActivityOptionsCompat 可以很好解决旧版本的兼容问题。
- AdapterViewFlipper.fyiWillBeAdvancedByHostKThx()——仅仅因为很好玩，没有其他原因。在整个安卓开源项目中（AOSP the Android ——open Source Project Android开放源代码项目）中还有其他很有意思的东西（比如GRAVITY_DEATH_STAR_I）。不过，都不像这个这样，这个确实有用
- ViewParent.requestDisallowInterceptTouchEvent() ——Android系统触摸事件机制大多时候能够默认处理，不过有时候你需要使用这个方法来剥夺父级控件的控制权。

译文出自 [@Gracker](#) 的博客，[Android Performance](#)：

Part1: [\[译\]Android小技巧\(1\)](#)

Part2: [\[译\]Android小技巧\(2\)](#)

Part3: [\[译\]Android小技巧\(3\)](#)

Part4: [\[译\]Android小技巧\(4\)](#)

Part5: [\[译\]Android小技巧\(5\)](#)

原文出自 Dan Lew 的博客，有 5 篇，强烈推荐。

[Android Tips Round-Up, Part 1](#)

[Android Tips Round-Up, Part 2](#)

[Android Tips Round-Up, Part 3](#)

[Android Tips Round-Up, Part 4](#)

[Android Tips Round-Up, Part 5](#)

最后做个福利广告 [zhengxiaopeng/android-dev-bookmarks · GitHub](#)

补充

- 1、android:clipChildren 和 android:clipToPadding： clipToPadding就是说控件的绘制区域是否在padding里面的，true的情况下如果你设置了padding那么绘制的区域就往里缩，clipChildren是指子控件是否超过padding区域，这两个属性默认是true的，所以在设置了padding情况下，默认滚动是在 padding内部的，要达到上面的效果主要把这两个属性设置了false那么这样子控件就能画到padding的区域了。使用场景如：ActionBar（透明）下显示Listview而第一项要在actionbar下。参见[android:clipToPadding和android:clipChildren](#)。
- 2、Fragment 的 setUserVisibleHint 方法，可实现 fragment 对用户可见时才加载资源（延迟加载）。

- 3、自定义 View 时重写 hasOverlappingRendering 方法指定 View 是否有 Overlapping 的情况，提高渲染性能。
 - 4、AutoScrollHelper，在可滚动视图中长按边缘实现滚动，[Android View.OnTouchListener 的子类](#)。
 - 5、TouchSlop，系统所能识别出的被认为是最小的滑动距离，`ViewConfiguration.get(context).getScaledTouchSlop()`。
 - 6、VelocityTracker，可用于 View 滑动事件速度跟踪。
 - 7、AlphabetIndexer，字母索引辅助类。
 - 8、Messenger，AIDL 实现的封装，比手写 AIDL 更方便。
 - 9、ArrayMap，比 HashMap 更高的内存效率，但比 HashMap 慢，不适合有大量数据的场景。
 - 10、Property，抽象类，封装出对象中的一个易变的属性值，使用场景如在使用属性动画时对动画属性的操作。
 - 11、SortedList，v7 包中，见名知意。
-

技术小黑屋，<http://droidyue.com/>

- HandlerThread 单一线程 + 任务队列 处理轻量的异步任务 详见 [详解 Android 中的 HandlerThread](#)
 - Proguard assumenosideeffects 编译器屏蔽日志 详见 [关于Android Log的一些思考](#)
 - [Android性能调优利器StrictMode](#)
 - Android中线程优先级控制 `android.os.Process.setThreadPriority` 详见 [剖析Android中进程与线程调度之nice](#)
 - Android Lint 一个静态分析工具 详见 [使用Android lint发现并解决高版本API问题](#)
 - (更新) 使用UncaughtExceptionHandler 处理应用崩溃，收集信息甚至可以不显示崩溃对话框 [详细：Android处理崩溃的一些实践](#)
-

张明云，<http://zmywly8866.github.io/>

- 清除画布上的内容：`canvas.drawColor(Color.TRANSPARENT, PorterDuff.Mode.CLEAR);`
- 在自定义View的onDetachedFromWindow方法中清理与View相关的资源；
- Fragment在onAttach方法中接回调：

```
@Override  
public void onAttach(Activity activity) {  
    super.onAttach(activity);  
    try {  
        mPageSelectedListener = (PageSelectedListener) activity;  
        mMenuBtnOnclickListener = (MenuBtnOnClickListener) activity;  
        mCommitBtnOnClickListener = (CommitBtnOnClickListener) activity;  
    } catch (ClassCastException e) {  
        throw new ClassCastException(activity.toString()  
            + "must implements listener");  
    }  
}
```

- 使用[ClipDrawable](#)实现进度条功能；
- 自定义view中的getContext(), 再也不需要专门创建一个mContext全局对象了；
- 自定义手写view的时候，在手指移动的过程中通过[MotionEvent | Android Developers](#) 对象的getHistorySize()获得缓存的历史点，绘制出来的曲线要平滑很多。
- 复写Activity的onUserLeaveHint方法，确保用户离开界面时能够立即暂停界面中的一些任务，关于onUserLeaveHint的更多作用可以谷歌：[android - Google 搜索](#)
- 值得借鉴的点击两次退出应用的实现：[Android关于双击退出应用的问题](#)

没那么麻烦，直接用toast的getView().getParent() 判断是不是空就ok了。API 16 测试通过

```
public class MainActivity extends Activity {  
  
    private Toast toast;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        toast = Toast.makeText(getApplicationContext(), "确定退出?", 0);  
  
    }  
    public void onBackPressed() {  
        quitToast();  
    }  
    /*  
     *  
     *  
     *  
     *  
     *  
     *  
     *  
     */  
    public boolean onKeyDown(int keyCode, KeyEvent event) {  
        System.out.println(keyCode + "...." + event.getKeyCode());  
        if(keyCode == KeyEvent.KEYCODE_BACK){  
            quitToast();  
        }  
        return super.onKeyDown(keyCode, event);  
    }  
    /*  
     *  
     */  
    private void quitToast() {  
        if(null == toast.getView().getParent()){  
            toast.show();  
        }else{  
            System.exit(0);  
        }  
    }  
}
```

Allan, Android 爱好者 -> 开发者

- android:animateLayoutChanges

一直以为 Lollipop Dialer 接通画面里面那些酷炫的动画（文字部分）是很复杂的做出来的，后来发现其实只有一行。[视频演示](#)

只需要加好 android:animateLayoutChanges="true" 然后 setVisibility 就可以了

林申:魅族科技 (中国) 有限公司 软件工程师

竟然没有人说这个:

LocalBroadcastManager

- You know that the data you are broadcasting won't leave your app, so don't need to worry about leaking private data.
- It is not possible for other applications to send these broadcasts to your app, so you don't need to worry about having security holes they can exploit.
- It is more efficient than sending a global broadcast through the system.

简单来说，就是 LocalBroadcastManager 可以在App的范围内发广播和收广播，不会被 global 的 receiver 收到，对数据比较敏感且不用共享的可以用这个。

果然 google support 包里面藏了太多的好东西。

夏青，Android工程师 谣言粉碎机

说几个简单但是基础的吧

- selector用这个来做样式的多态真是没有太方便了，以前傻傻的自己分析事件来变换
- HierarchyViewer这个工具用来了解界面实现方式，找到每个view布局和对应id实在太方便了，还可以配合dumpsys命令用来调试
- ListView ViewHolder的使用，虽说现在RecyclerView已经把ViewHolder包含进去了，但是还是要说说这个方法对于复用的意义
- moveTaskToBack 看到很多论坛写模拟home按键的方式是用 Intent.setAction(Intent.ACTION_MAIN) 来实现所谓模拟home键的方式(实际上是调用 launcher)，其实很多场景里面要这么做的主要目的就是为了让当前的APP隐藏而非退出，但是用Intent方法其实并不是那么优雅，刚才说了这个方法的实质是调用 launcher，会导致所有应用全部转到后台，最近在做Android多任务相关工作，这么做对于系统开发者和其他应用造成不小的困扰，其实在activity层面调用 moveTaskToBack 就可以搞定了
- setSystemUiVisibility 和 setStatusBarColor 要实现status bar的透明或者颜色用这两个接口就可以了，透明只需要设置SystemUiVisibility为 View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN |

View.SYSTEM_UI_FLAG_LAYOUT_STABLE, 颜色的话调用后面那个接口就可以, 看到某某云音乐用系统id status_bar_height高度各种计算, 自己绘制来实现对于status bar的染色也是最了

- loader和ContentObserver 实现数据于控制机制分离的非常好的结构, Android原生邮件系统使用了大量的这样的模式, 来处理繁杂的邮件相关的信息内容, 加载邮件服务器内容到数据库以及UI上的显示更新完全是两条路
-

尹東, INTP

TimingLogger,SDK自带打印时间戳工具, 简直神器。

摘自官方文档:

```
A utility class to help log timings splits throughout a method call. Typical usage is

TimingLogger timings = new TimingLogger(TAG, "methodA");
// ... do some work A ... timings.addSplit("work A");
// ... do some work B ... timings.addSplit("work B");
// ... do some work C ... timings.addSplit("work C"); timings.dumpToLog();

The dumpToLog call would add the following to the log:

D/TAG ( 3459): methodA: begin
D/TAG ( 3459): methodA: 9 ms, work A
D/TAG ( 3459): methodA: 1 ms, work B
D/TAG ( 3459): methodA: 6 ms, work C
D/TAG ( 3459): methodA: end, 16 ms
```

原文 <http://www.ydcool.me/archives/11/>

master郑, 做好产品

我也添一些, 如有雷同纯属巧合~

- 1.通过 WindowManager.addView 在其他app界面添加一个view时, 经常会无法显示, 特别在miui, emui固件上, 需要指定type为LayoutParams.TYPE_TOAST。
 - 2.View.getLocationOnScreen(new int[]), 获取view在屏幕上的位置
-

- 3.Paint.setXfermode(porterDuffXfermode)，在ApiDemo里面有专门的介绍，实现了穿透，叠加，覆盖等多种绘制效果，非常实用
- 4.直接获取当前系统壁纸的fd

```
Ibinder binder = ServiceManager.getService("wallpaper");
IWallpaperManager wm = IWallpaperManager.Stub.asInterface(binder);
Bundle params = new Bundle();
ParcelFileDescriptor fd = wm.getWallpaper(stub, params);
```

直接获取当前系统壁纸的fd，避免壁纸过大造成oom问题。这种方式有适配问题，需注意。

- 5.通过View.getDrawingCache()可以获取截图，但是需要setDrawingCacheEnabled(true)频繁使用可能会oom，还有一种方法直接用canvas

```
Bitmap bm = Bitmap.createBitmap((int)(w * scale), (int)(h * scale), Bitmap.Config.ARGB_8888);
Canvas canvas = new Canvas();
canvas.setBitmap(bm);
View.draw(canvas);
return bm;
```

- 6.说到几个oom，顺带说下有一种偷懒又有效的解决办法，在manifest上加 android:largeHeap="true"
- 7.用一个牛逼的来结尾，AccessibilityService。由于强大所以需要手动

咕咚，热爱代码，喜欢倒腾的程序员 ...

AtomicInteger，一个提供原子操作的Integer的类。在Java语言中，++i和i++操作并不是线程安全的，在使用的时候，不可避免的会用到synchronized关键字。而AtomicInteger则通过一种线程安全的加减操作接口。这个类在AsyncTask中用到了。

田元，算不上程序员。

- 1、android.support.design.widget.TextInputLayout，给EditText带个套吧⊙▽⊙

- 2、AndroidManifest.xml activity的一些标签，比如`android:windowSoftInputMode`

```
<activity android:name=".Main"
    android:label="@string/app_name"
    android:windowSoftInputMode="stateHidden" >
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
```

activity launch后默认隐藏键盘，这在activity里面有EditText等元素又不想一开始就弹出软键盘的情况下有用，在此之前就知道`android:name` 和`android:label`这两个属性(:3)∠)

- 3、`getSystemService`函数，获取各种系统service,而且不用担心性能问题，都是直接返回各种manager。
- 4、`Parcelable`接口 原来受MFC等c++类库影响，比较习惯继承`serializable`接口这种方式，但后来知道了`Parcelable`的实现方式就喜欢上了。
- 5、`android.support.v4.widget.DrawerLayout` 原生大方的抽屉控件。
- 6、`android.support.v7.widget.Toolbar` 定制性极强的viewGroup

peerless2012

- 5、`View`中的`isShown()`方法，以前都是用`view.getVisibility() == View.VISIBLE`来判断的

这个方法是个坑，并不是像描述的那样，如果view是个`ViewGroup`，如果某个子view是不可见的，其他可见，这个方法照样会返回fasle

[陈启超]

```
sendOrderedBroadcast (Intent intent, String receiverPermission,
    BroadcastReceiver resultReceiver, Handler scheduler,
    int initialCode, String initialData, Bundle initialExtras)
```

可以设置一个最终广播接收者`resultReceiver`。即使有优先级高的广播接收者使用了`abortBroadcast()`拦截了广播，最终的广播接收着依然可以接收到广播。

参考：[Context | Android Developers](#)(`android.content.Intent`, `java.lang.String`,
`android.content.BroadcastReceiver`, `android.os.Handler`, `int`, `java.lang.String`,
`android.os.Bundle`)

李照

再补充一个监听主线程是否空闲，在View丢帧的情况下使用，效果立竿见影，在主线程中
`Looper.myQueue().addIdleHandler(mIdleHandler)`

千里山南，Android Dever

- `LinearLayout#setDividerDrawable`
- `View#inflate()`
- `URLUtil#isNetworkUrl`及其他URL校验的方法
- `Debug#startMethodTracing`
- `Debug#waitForDebug`
- `TimeUnit.SECONDS#sleep`
- `View.addOnAttachStateChangeListener`

还有一些稍后补充

另外推荐一本小册子 [《50 Android Hacks》摘要如下](#)50 Android Hack 读书笔记

其他

- `Throwable`类中的`getStackTrace()`配合`Arrays.toString()`方法，可打印函数的逐层调用，方便找出引发异常的代码。
- [Cool Android Apis 整理（一）](#)
- [Cool Android Apis 整理（二）](#)
- [Cool Android Apis 整理（三）](#)

Android 日常开发总结的技术经验 60 条

来源:[Liter's Blog](#)

- 1.全部Activity可继承自 BaseActivity，便于统一风格与处理公共事件，构建对话框统一构建器的建立，万一需要整体变动，一处修改到处有效。
- 2.数据库表段字段常量和SQL逻辑分离，更清晰，建议使用Lite系列框架[LiteOrm](#)库，超级清晰且重心可以放在业务上不用关心数据库细节。
- 3.全局变量放全局类中，模块私有放自己的管理类中，让常量清晰且集中。
- 4.不要相信庞大的管理类的东西会带来什么好处，可能是一场灾难，而要时刻注意单一职责原则，一个类专心做好一件事情更为清晰。
- 5.如果数据没有必要加载，数据请务必延迟初始化，谨记为用户节省内存，总不会有坏处。
- 6.异常抛出，在合适的位置处理或者集中处理，不要搞的到处是catch，混乱且性能低，尽量不要在循环体中捕获异常，以提升性能。
- 7.地址引用链长时（3个以上指向）小心内存泄漏，和警惕堆栈地址指向，典型的易发事件是：数据更新了，ListView视图却没有刷新，这时Adapter很可能指向并的并不是你更新的数据容器地址（一般为List）。
- 8.信息同步：不管是数据库还是网络操作，新插入的数据注意返回ID（如果没有赋予唯一ID），否则相当于没有同步。
- 9.多线程操作数据库时，db关闭了会报错，也很可能出现互锁的问题，推荐使用事务，推荐使用自动化的[LiteOrm](#)库操作。
- 10.做之前先考虑那些可以公用，资源，layout，类，做一个结构、架构分析以加快开发，提升代码可复用度。
- 11.有序队列操作add、delete操作时注意保持排序，否则你会比较难堪喔。
- 12.数据库删除数据时，要注意级联操作避免出现永远删不掉的脏数据喔。
- 13.关于形参实参：调用函数时参数为基本类型传的是值，即传值；参数为对象传递的是引用，即传址。
- 14.listView在数据未满一屏时，setSelection函数不起作用；ListView批量操作时各子项和视图正确对应，可见即所选。

15.控制Activity的代码量，保持主要逻辑清晰。其他类遵守SRP（单一职能），ISP（接口隔离）原则。

16.arraylist执行remove时注意移除int和Integer的区别。你懂得。

17.Log请打上Tag，调试打印一定要做标记，能定位打印位置，否则尴尬是：不知道是哪里在打印。

18.码块/常量/资源可以集中公用的一定共用，即使共用逻辑稍复杂一点也会值得，修改起来很轻松，修改一种，到处有效。

19.setSelection不起作用，尝试smoothScrollToPosition。ListView的LastVisiblePosition（最后一个可见子项）会随着getView方法执行位置不同变动而变。

20.与Activity通讯使用Handler更方便；如果你的框架回调链变长，考虑监听者模式简化回调。

21.监听者模式不方便使用时，推荐EventBus框架库，使用事件总线，没接触过的同学可以自行脑补一下哦。

22.Handler在子线程线程使用Looper.prepare，或者new的时候给构造函数传入MainLooper来确保在主线程run。

23.timepicker 点击确定后需要clearFocus才能获取手动输入的时间。

24.构造函数里面极度不推荐启动异步线程，会埋下隐患。比如：异步线程调用了本类的实例，就会悲剧等着崩溃吧。

25.千万不要理所当然的以为一个对象不会为空，充分的做好容错处理；另外注意null也可以插入ArrayList等容器中。

26.ExpandableListView的子列表不能点击（禁用）要把Adapter的isChildSelectable方法返回true。

27.UI显示注意内容过长的情形要提前使用ScrollView否则在小手机上尴尬你懂得。

28.注意按钮的感应范围不小于9mm否则不易点击；输入框注意光标的位置更易用户输入。

29.服务器和客户端尽量统一唯一标识（有可能是ID），否则多少会有歧义和问题。

30.注释，尽量去写足够的注释，去描述一下思路，达到看了可以明白某一块代码的效果。

31.完整型数据一定要用Sqlite的Transaction，大数据一定要用。粗略测试插入100个数据有20倍的提速，插入1000个数据就有100多倍的提速。

32. 避免String="null"的情况出现String = null,""都可以。避免出现title="无主题"这样的数据提交到数据库浪费空间。

33. 存在多个不同的dbhelper实例情况下，sqlitedatabase对象必然存在不同的实例，多线程同时写入数据，轮流写入数据时会不定时的报db is locked，引起崩溃，不管是操作同张表还是异表。读和写可以同时并发，轮流无规律的交替执行。同时写入数据时解决方案是用并发的每个线程都用事务，db则不会lock，按次整体写入。

34. 建议整个应用维护一个dbhelper实例，只要db没有关闭，全局就只有一个db实例，多线程并发写入db不会lock，严格交替进行写入：123123123。。。 (123代表不同线程，轮流插入一个记录)，读和写均不会锁住db，读写交替并没有规律，执行次数和程度看cpu分配给哪个线程的时间片长。

35. 一个任务使用事务嵌套N个事务，N个事务中有一个失败，这个任务整体失败，全部成功后，数据才写入，具有安全性，整体性。并且事务写入大批量数据的效率经实际测试成百上千倍的高于一般的单个写入。数据库大量数据、多线程操作建议使用LiteOrm数据库框架，更稳定简单。

36. 经常需要用ListView或者其它显示大量Items的控件实时跟踪或者查看信息，并且希望最新的条目可以自动滚动到可视范围内。通过设置的控件transcriptMode属性可以将Android平台的控件（支持ScrollBar）自动滑动到底部。

37. Long a; 判断a有没有赋值，if(a == 0)在a没有赋值情况下会报错。应该if(a == null), Integer、Float等也一样，原因你懂，只是提醒你要小心喔。

38. 编码遇到读写、出入等逻辑要双向考虑，文件导入导出，字符字节相互转换都要两边转码。

39. 一个int值与一个Integer对象（能包含int值的最小对象）的大小比率约为1:4（32位和64位机器有不同）。额外的开销源于JVM用于描述Java对象的元数据也就是Integer, (Long、Double等也是)。

40. 对象由元数据和数据组成。元数据包括类（指向类的指针，描述了类的类型），标记（描述了对象状态，如散列码、形状等），锁（对象同步信息）。数组对象还包括大小的元数据。

41. 一个在32位Java运行时中使用1GB Java堆的Java应用程序在迁移到64位Java运行时之后，通常需要使用1.7GB的Java堆。

42. Hash集合的访问性能比任何List的性能都要高，但每条目的成本也要更高。由于访问性能方面的原因，如果您正在创建大集合（例如，用于实现缓存），那么最好使用基于Hash的集合，而不必考虑额外的开销。

43.对于并不那么注重访问性能的较小集合而言，List 则是合理的选择。ArrayList 和 LinkedList 集合的性能大体相同，但其内存占用完全不同：ArrayList 的每条目大小要比 LinkedList 小得多，但它不是准确设置大小的。List 要使用的正确实现是 ArrayList 还是 LinkedList 取决于 List 长度的可预测性。如果长度未知，那么正确的选择可能是 LinkedList，因为集合包含的空白空间更少。如果大小已知或可预知或比较小，那么 ArrayList 的内存开销会更低一些。

43.选择正确的集合类型使你能够在集合性能与内存占用之间达到合理的平衡。除此之外，你可以通过正确调整集合大小来最大化填充率、最小化未得到利用的空间，从而最大限度地减少内存占用。

44.充分利用封装（提供接口类来控制访问数据）和委托（helper对象来实施任务）两种理念。

45.延迟分配 Hashtable：如果 Hashtable 为空是经常发生的普遍现象，那么仅在存在需要存储的数据时分配 Hashtable 应该是一种合理的做法。将 Hashtable 分配为准确的大小：虽然会有默认大小，但建议使用更为准确的初始大小。

46.EditText在setText时不要忘记是否需要setSelection。在大多数情况下是需要设置的。

47.XML两种情况要注意：1 属性名字时候有重复；2 注意文本是否包含非法字符，注意使用CDATA包裹。

48.当逻辑没有明显问题时考虑对象属性、函数参数、网络传输参数是否全部了解，是否设置正确。

49.当出现编译或者运行时错误，别人那没问题时，考虑你的编译环境和环境版本是否有问题。

50.由于String类的immutable性质，当String变量需要经常变换其值时，应该考虑使用 StringBuilder提升性能，多线程使用 StringBuffer操作string提高程序效率。

51.java 栈的优势是比堆速度快，可共享，主要存放临时变量、参数等，堆的优势是可动态分配内存大小。

52.只要是用new()来新建对象的，都会在堆中创建，而且其数据是单独存值的，即使与栈中的数据（值）相同，也不会与栈中的数据共享。

53.基本数据类型定义的变量称自动变量，存的是‘字面值’，存在于栈中，可共享（存在即不新建）。

54.多个RandomAccessFile对象指向同一个文件，可使用多个线程一起写入无需再自己加锁，经试验结论：三个线程分别写入100万次数据，使用锁约12秒，不使用约8.5秒。100个线程分别写入1万次数据使用锁耗时约4.2秒，不使用锁耗时约3秒。

55.XmlPullParser解析慎用nextText()方法，xml比较复杂，含有空标签、重复名字标签时容易出现异常问题；TEXT中使用getText()方法代替START_TAG中使用nextText()方法；START_TAG, TEXT, END_TAG三个事件配合使用。注意每个xml节点之间（不管是开始节点还是结束节点）都会出现TEXT事件。

56.改变逻辑的时候考虑全部用到这项功能的地方，分散的地方多了，容易大意。

57.当系统原生组件出现问题时，查看错误栈信息，自己写一个该组件的子类，并在合适的地方将出错方法复写一下，加上try catch保证不崩溃掉。不要扰乱了该系统控件的正常逻辑。

58.输入控件注意对空格、换行等符号的控制；输入框里内容注意和左右控件的空间，防止误点击。

59.注意函数参数里的++或者-操作。是++c 还是 c++，区别很大。

60.各种地方、永远的不要小看null指针问题，甚至有些场合宁可错杀（try catch），不可放过。

你应该知道的那些Android小经验

来源:www.jayfeng.com

做Android久了，就会踩很多坑，被坑的多了就有经验了，闲暇之余整理了部分，现挑选一些重要或者偏门的“小”经验做个记录。

查看SQLite日志

```
adb shell setprop log.tag.SQLiteLog V  
adb shell setprop log.tag.SQLiteStatements V
```

因为实现里用了 `Log.isLoggable(TAG, Log.VERBOSE)` 做了判断，`LessCode` 的 `LogLess` 中也参考了这种机制：[LogLess](#)。

使用这种方法就可以在Release版本也能做到查看应用的打印日志了。

PNG优化

APK打包会自动对PNG进行无损压缩，如果自行无损压缩是无效的。

当然进行有损压缩是可以的：<https://tinypng.com/>

Tcpdump抓包

有些模拟器比如genymotion自带了tcpdump，如果没有的话，需要下载tcpdump：
<http://www.strazzere.com/android/tcpdump>

把tcpdump push到/data/local下，抓包命令：

```
adb shell /data/local/tcpdump -i any -p -s 0 -w /sdcard/capture.pcap
```

查看签名

很多开发者服务都需要绑定签名信息，用下面的命令可以查看签名：

```
keytool -list -v -keystore release.jks
```

注意，这个是需要密码的，可以查看MD5, SHA1, SHA256等等。

单例模式(懒汉式)的更好的写法

特别说到这个问题，是因为网上很多这样的代码：

```
public class Singleton {  
    private static Singleton instance;  
    private Singleton (){}  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

这种写法线程不安全，改进一下，加一个同步锁：

```
public class Singleton {  
    private static Singleton instance;  
    private Singleton (){}  
    public static synchronized Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

网上这样的代码更多，可以很好的工作，但是缺点是效率低。

实际上，早在JDK1.5就引入volatile关键字，所以又有了一种更好的双重校验锁写法：

```

public class Singleton {
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}

```

注意，别忘记`volatile`关键字哦，否则就是10重，100重也可能还是会出问题。

上面是用的最多的，还有一种静态内部类写法更推荐：

```

public class Singleton {
    private Singleton() {}
    private static class SingletonLoader {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonLoader.INSTANCE;
    }
}

```

多进程Application

是不是经常发现Application里的方法执行了多次？百思不得其解。

因为当有多个进程的时候，Application会执行多次，可以通过pid来判断那些方法只执行一次，避免浪费资源。

隐式启动Service

这是Android5.0的一个改动，不支持隐式的Service调用。下面的代码在Android 5.0+上会报错： `Service Intent must be explicit :`

```
Intent serviceIntent = new Intent();
serviceIntent.setAction("com.jayfeng.MyService");
context.startService(serviceIntent);
```

可改成如下：

```
// 指定具体Service类，或者有packageName也行
Intent serviceIntent = new Intent(context, MyService.class);
context.startService(serviceIntent);
```

fill_parent的寿命

在Android2.2之后，支持使用match_parent。你的布局文件里是不是既有fill_parent和match_parent显得很乱？

如果你现在的minSdkVersion是8+的话，就可以忽略fill_parent，统一使用match_parent了，否则请使用fill_parent。

ListView的局部刷新

有的列表可能 `notifyDataSetChanged()` 代价有点高，最好能局部刷新。

局部刷新的重点是，找到要更新的那项的View，然后再根据业务逻辑更新数据即可。

```
private void updateItem(int index) {
    int visiblePosition = listView.getFirstVisiblePosition();
    if (index - visiblePosition >= 0) {
        //得到要更新的item的view
        View view = listView.getChildAt(index - visiblePosition);

        // 更新界面（示例参考）
        // TextView nameView = ViewLess.$(view, R.id.name);
        // nameView.setText("update " + index);
        // 更新列表数据（示例参考）
        // list.get(index).setName("Update " + index);
    }
}
```

强调一下，最后那个列表数据别忘记更新，不然数据源不变，一滚动可能又还原了。

系统日志中几个重要的TAG

```
// 查看Activity跳转  
adb logcat -v time | grep ActivityManager  
// 查看崩溃信息  
adb logcat -v time | grep AndroidRuntime  
// 查看Dalvik信息, 比如GC  
adb logcat -v time | grep "D\\Dalvik"  
// 查看art信息, 比如GC  
adb logcat -v time | grep "I\\art"
```

一行居中, 多行居左的TextView

这个一般用于提示信息对话框, 如果文字是一行就居中, 多行就居左。

在TextView外套一层wrap_content的ViewGroup即可简单实现:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
    <!-- 套一层wrap_content的ViewGroup -->  
    <LinearLayout  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_centerInParent="true">  
  
        <TextView  
            android:layout_width="wrap_content"  
            android:layout_height="wrap_content"  
            android:text="@string/hello_world" />  
    </LinearLayout>  
</RelativeLayout>
```

setCompoundDrawablesWithIntrinsicBounds()

网上一大堆setCompoundDrawables()方法无效不显示的问题, 然后解决方法是setBounds, 需要计算大小... 不用这么麻烦, 用setCompoundDrawablesWithIntrinsicBounds()这个方法最简单!

计算程序运行时间

为了计算一段代码运行时间，一般的做法是，在代码的前面加个startTime，在代码的后面把当前时间减去startTime，这个时间差就是运行时间。

这里提供一种写起来更方便的方法，完全无时间逻辑，只是加一个打印log就够了。

```
// 测试setContentView()的时间
Log.d("TAG", "Start");
setContentView(R.layout.activity_http);
Log.d("TAG", "End");
```

没有计算时间的逻辑，这能测出来？

把日志过滤出来，运行命令“adb logcat -v time | grep TAG”：

```
03-18 14:47:25.477 D/TAG      (14600): Start
03-18 14:47:25.478 D/TAG      (14600): End
```

通过 `-v time` 参数，可以比较日志左边的时间来算出中间的代码运行的时间。

JAVA引用类型一览表

对象引用：强引用 > 软引用 > 弱引用 > 虚引用。

| 引用类型 | 回收时机 | 用途 | 生存时间 |
|------|--------|---------|------------|
| 强引用 | 从来不会 | 对象的一般状态 | JVM停止运行时终止 |
| 软引用 | 在内存不足时 | 对象缓存 | 内存不足时终止 |
| 弱引用 | 在垃圾回收时 | 对象缓存 | GC运行后终止 |
| 虚引用 | 在垃圾回收时 | 对象跟踪 | GC运行后终止 |

Context使用场景

为了防止Activity，Service等这样的Context泄漏于一些生命周期更长的对象，可以使用生命周期更长的ApplicationContext，但是不是所有的Context的都能替换为 `ApplicationContext`，这是网上流传的一份表格：

| | Application | Activity | Service | ContentProvider |
|--------------------------|--------------------|-----------------|----------------|------------------------|
| Show Dialog | 否 | 是 | 否 | 否 |
| Start Activity | 否 | 是 | 否 | 否 |
| Layout Inflation | 否 | 是 | 否 | 否 |
| Start Service | 是 | 是 | 是 | 是 |
| Bind Service | 是 | 是 | 是 | 是 |
| Send Broadcast | 是 | 是 | 是 | 是 |
| Regist BroadcastReceiver | 是 | 是 | 是 | 是 |
| Load Resource Value | 是 | 是 | 是 | 是 |

图片缓存大小

现在很多图片库需要给图片设置一个最大缓存，但是这个值设置多少合适呢？

高端机和低端机的配置显然应该不同，可以考虑设置一个动态值。

建议设置为应用可用内存的1/8:

```
int memoryCache = (int) (Runtime.getRuntime().maxMemory() / 8);
```

系统内置的一些工具类

在AOSP源码全局搜了一下包含Util关键字的类，整理出这个列表供大家参考：

```
// 系统
./android/database/DatabaseUtils.java
./android/transition/TransitionUtils.java
./android/view/animation/AnimationUtils.java
./android/view/ViewAnimationUtils.java
./android/webkit/URLUtil.java
./android/bluetooth/le/BluetoothLeUtils.java
./android/gesture/GestureUtils.java
./android/text/TextUtils.java
./android/text/format/DateUtils.java
./android/os/FileUtils.java
```

```
./android/os/CommonTimeUtils.java  
./android/net/NetworkUtils.java  
./android/util/MathUtils.java  
./android/util/TimeUtils.java  
./android/util/ExceptionUtils.java  
./android/util/DebugUtils.java  
./android/drm/DrmUtils.java  
./android/media/ThumbnailUtils.java  
./android/media/ImageUtils.java  
./android/media/Utils.java  
./android/opengl/GLUtils.java  
./android/opengl/ETC1Util.java  
./android/telephony/PhoneNumberUtils.java  
// 设计和支持库  
./design/src/android/support/design/widget/ViewGroupUtils.java  
./design/src/android/support/design/widget/ThemeUtils.java  
./design/src/android/support/design/widget/ViewUtils.java  
./design/lollipop/android/support/design/widget/ViewUtilsLollipop.java  
./design/base/android/support/design/widget/AnimationUtils.java  
./design/base/android/support/design/widget/MathUtils.java  
./design/honeycomb/android/support/design/widget/ViewGroupUtilsHoneycomb.java  
.v7/recyclerview/src/android/support/v7/widget/helper/ItemTouchUIUtil.java  
.v7/recyclerview/src/android/support/v7/widget/helper/ItemTouchUIUtilImpl.java  
.v7/recyclerview/src/android/support/v7/util/MessageThreadUtil.java  
.v7/recyclerview/src/android/support/v7/util/AsyncListUtil.java  
.v7/recyclerview/src/android/support/v7/util/ThreadUtil.java  
.v7/recyclerview/tests/src/android/support/v7/widget/AsyncListUtilLayoutTest.java  
.v7/recyclerview/tests/src/android/support/v7/util/AsyncListUtilTest.java  
.v7/recyclerview/tests/src/android/support/v7/util/ThreadUtilTest.java  
.v7/appcompat/src/android/support/v7/graphics/drawable/DrawableUtils.java  
.v7/appcompat/src/android/support/v7/widget/DrawableUtils.java  
.v7/appcompat/src/android/support/v7/widget/ThemeUtils.java  
.v7/appcompat/src/android/support/v7/widget/ViewUtils.java  
.v4/tests/java/android/support/v4/graphics/ColorUtilsTest.java  
.v4/jellybean-mr1/android/support/v4/text/TextUtilsCompatJellybeanMr1.java  
.v4/jellybean/android/support/v4/app/BundleUtil.java  
.v4/jellybean/android/support/v4/app/NavUtilsJB.java  
.v4/java/android/support/v4/app/NavUtils.java  
.v4/java/android/support/v4/database/DatabaseUtilsCompat.java  
.v4/java/android/support/v4/graphics/ColorUtils.java  
.v4/java/android/support/v4/text/TextUtilsCompat.java  
.v4/java/android/support/v4/util/TimeUtils.java  
.v4/java/android/support/v4/util/DebugUtils.java  
.v4/java/android/support/v4/content/res/TypedArrayUtils.java
```

这么多工具类，一定可以找到对你有用的。

ClipPadding

这个不多说， ListView的ClipPadding设为false， 就能为ListView设置各种padding而不会出现丑陋的滑动“禁区”了。

强大的dumpsys

dumpsys可以查看系统服务和状态， 非常强大， 可通过如下查看所有支持的子命令：

```
dumpsys | grep "DUMP OF SERVICE"
```

这里列举几个稍微常用的：

| 子命令 | 备注 |
|------------|--|
| activity | 显示所有的activities的信息 |
| cpuinfo | 显示CPU信息 |
| window | 显示键盘， 窗口和它们的关系 |
| meminfo | 内存信息 (meminfo \$package_name or \$pid 使用包名或者进程id 显示内存信息) |
| alarm | 显示Alarm信息 |
| statusbar | 显示状态栏相关的信息 (找出广告通知属于哪个应用) |
| usagestats | 每个界面启动的时间 |

bugreport命令

很多人都用过 adb logcat ， 但是如果想要更详细的信息， logcat则无能为力。

所以大多数手机厂商测试更多的是用 adb bugreport 来抓log给开发人员分析。

```
// 除了log， 还包括启动后的系统状态， 包括进程列表， 内存信息， VM信息等等
// 而且不像logcat是一直打印的， bugreport命令输出到当前时间就停止结束了。
adb bugreport > main.log
```

dpi文件夹的换算比例

之前的ldpi基本可以抛弃了，主流的dpi已经从很早之前的mdip转移到了xhdpi了，特别提醒。

| PPI | RESOLUTION | DP | PX |
|----------------|------------|----|-----|
| mdpi(160dp) | 320P | 1 | 1 |
| hdpi(240dp) | 480P | 1 | 1.5 |
| xhdpi(320dp) | 720P | 1 | 2 |
| xxhdpi(480dpi) | 1080P | 1 | 3 |

更新媒体库文件

以前做ROM的时候经常碰到一些第三方软件（某音乐APP）下载了新文件或删除文件之后，但是媒体库并没有更新，因为这个是需要第三方软件主动触发。

```
// 通知媒体库更新单个文件状态
Uri fileUri = Uri.fromFile(file);
sendBroadcast(new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE, fileUri));
```

媒体库会在手机启动，SD卡插拔的情况下进行全盘扫描，不是实时的而且代价比较大，所以单个文件的刷新很有必要。

Monkey参数

大家都知道，跑monkey的参数设置有一些要注意的地方，比如太快了不行不切实际，太慢了也不行等等，这里给出一个参考：

```
adb shell monkey -p <packageName> -s 1000 --ignore-crashes --ignore-timeouts --ignore
```

一边跑monkey，一边抓log吧。

小结

无论是大经验还是小经验，有用就是好经验。

开发中必须避免的基础问题

来源:[Liter's Blog](#)

有些东西可能是老生常谈，这并不重要，重要的是有些东西分享出来，发扬下去，即便是非常非常基础的东西。

日常编码中注意避免以下“低级错误”。

1.空指针异常：第一就要说这个，必须的，避免了它，大概意味着避免80、90%的错误吧，对方法的调用不进行空指针判断而造成针异常（原则是千万不要想当然认为一个对象就不会空），举个简单例子就是equals操作时没有将常量放在equals操作符的左边（字符串变量与常量比较时，先写常量可以避免空指针异常）。记得有危险的地方要么if判断要么try catch。

2.命名与注释：方法和变量命名随意而不规范，没有类注释，方法注释或者注释不规范，代码修改后，不同步修改注释，注释与代码不符。

3.数据类重载：数据类不重载toString()方法，log打印时会对信息展示有所帮助。（说明：编程规范要求‘所有的数据类必须重载toString()方法，返回该类有意义的内容’）。

4.重量级资源释放：数据库操作，IO操作的资源没有及时释放，释放顺序不正确，或者使用没有必要的预处理，数据库操作，IO操作等需要使用结束close()的对象必须在try-catch-finally的finally中close()。

5.无视循环体效率：循环体内包含了大量没有必要在循环中处理的语句，循环体内循环获取数据库连接，垂体内进行不必要的try-catch操作，（说明：编程规范中建议不要在循环体内调用同步方法和使用try-catch块）。

6.小问题大伤害：不对数组下标作范围校验；用“==”比较两个字符串内容相等；对list做foreach轮回时，循环代码中修改list的结构。

7.异常处理：字符串转化为数字丰数据库时没有做异常处理；捕捉异常后没有在打印异常栈。

8.日志：没有打印异常栈消息，日志没有定位，没有实际作用；日志打印乱糟糟或者无意义；日志和实际情形不一致。

9.万恶的魔鬼字符，魔鬼数字。

10. 可复用性低：差不多的功能，到处都是冗杂的看似不同的代码（实际上很多可以抽离复用）。（建议1：码块/常量/资源可以集中公用的一定共用，即使共用逻辑复杂一点（防止逻辑太过复杂）也值得，修改起来很轻松，修改一种，到处有效。）（建议2：充分利用继承、多态、封装等面向对象机制来提高可复用性）

11. 可读性不高：

- a 代码不分组，不合理使用空行，相似作用的函数没有聚集在一块；
- b 逻辑嵌套不优化，嵌套if else层级过多；
- c 代码与数据耦合（常量或者SQL语句和逻辑耦合）；（建议：设计从简，遵循KIS原则；胆大兼顾心细，平衡稳定性和重构之间的矛盾，使代码以至架构和模块越来越合理）

12. 编程思想：

- a 模块分离：举个小例子，有不少同学在Activity里做了很多事，甚至做了DAO、网络操作、数据解析，这是不是很合理的，导致一个UI和逻辑之间的‘门面’挂载了过多的伤不起的‘难以承受之重’，阅读困难，逻辑庞大。（建议模块和代码遵循MVC模式，建议View视图、控制相关（内存管理、核心逻辑），数据相关（文件操作、数据库操作、网络操作、数据组装与解析、数据模型）各分一个相对独立大、小模块，模块内分层级架构（积极合理使用继承与实现等面向对象机制））。
- b 单一职责：有的同学写类啥都可以干，管得了内存，控得住文件，做得了解析，搞得了组装，上得了天堂，下得了厨房。这个是模块分离的基础。
- c 接口隔离：举个例子一个水果类在这里是卖水果功能，在那里却还可以买水果，这样是不太合理的，一个类对另一个类的依赖性建立在最小接口之上。一个接口一个角色，一种客户一种接口。

Xfermode

Xfermode in android

来源:[Weishu's Notes](#)

Xfermode有三个实现类：`AvoidXfermode`，`PixelXorXfermode` 以及 `PorterDuffXfermode`。

前两个类因为不支持硬件加速在 API level 16 被标记为 `Deprecated` 了，用也可以，但是需要关闭硬件加速，简单说下。

AvoidXfermode

`AvoidXfermode` xfermode will draw the src everywhere except on top of the opColor or, depending on the Mode, draw only on top of the opColor.

这话翻译成中文太别扭了，自己理解吧。举个栗子，如果你想对原来图像进行处理，把红色换成绿色，可以使用这个；或者，你想把不是红色的换成某个颜色，也行。这里有一个容差值的概念，比如红色是`0xff0000`但是在一定范围内也都是红色，如果你设定一个容差值，那么“各种符合要求的红色”都会被处理。

PixelXorfermode

文档都说这种模式对于操作混合色没有什么用，还不支持硬件加速，pass，说说重头戏。

PorterDuffXfermode

Porter-Duff的由来

说来说去，这个到处都是PorterDuff的玩意儿到底是什么意思？

Porter-Duff 操作是 1 组 12 项用于描述数字图像合成的基本手法，包括 Clear、Source Only、Destination Only、Source Over、Source In、Source Out、Source Atop、Destination Over、Destination In、Destination Out、Destination Atop、XOR。通过组合使用 Porter-Duff 操作，可完成任意 2D 图像的合成。

Thomas Porter 和 Tom Duff 发表于 1984年原始论文的扫描版本

看到没！可以完成任意2D图像的合成，理论支撑，所以说是核武器～

对文档的解释

如果去查阅文档这个模式怎么用，相信你一定会fuck：

```
public enum Mode {
    /** [0, 0] */
    CLEAR      (0),
    /** [Sa, Sc] */
    SRC        (1),
    /** [Da, Dc] */
    DST        (2),
    /** [Sa + (1 - Sa)*Da, Rc = Sc + (1 - Sa)*Dc] */
    SRC_OVER   (3),
    /** [Sa + (1 - Sa)*Da, Rc = Dc + (1 - Da)*Sc] */
    DST_OVER   (4),
    /** [Sa * Da, Sc * Da] */
    SRC_IN     (5),
    /** [Sa * Da, Sa * Dc] */
    DST_IN     (6),
    // ...以下省略
}
```

这尼玛是什么意思。。好了，别慌。如果懂些图形学，大致就知道：

- Sa = Source alpha
- Da = Dest alpha
- Sc = Source color
- Dc = Dst color

如果用叠加的形式看，**Dst**是下面的图，也就是先画的图；**Source**是上面的图，也就是后面要画的图。

要说明的是，使用porterduffxfermode绘制的时候，由于窗口时透明的，如果出现黑色结果，那么就是这个原因，stackoverflow上有很多这样的问题答案说需要之前画一个bitmap，原因是正确的，但是不应该这么处理，使用layer保存图层即可。

```
int count = canvas.saveLayer
paint.setXfermode()
canvas.drawXXX
canvas.restoreLayer()
```

实际效果测试以及mode含义

往上很多对于Xfermode的解释，使用API demo望文生义，并没有考虑到alpha通道的情况，实际上是错误的。典型的解释类似这种：

4. PorterDuff.Mode.SRC_OVER 正常绘制显示，上下层绘制叠盖。

5. PorterDuff.Mode.DST_OVER 上下层都显示。下层居上显示。

6. PorterDuff.Mode.SRC_IN 取两层绘制交集。显示上层。

7. PorterDuff.Mode.DST_IN 取两层绘制交集。显示下层。

只能说太肤浅了，我们根据上面的图像学的解释，alpha通道的存在意味着事情没那么简单，我们用实际的例子验证一下。

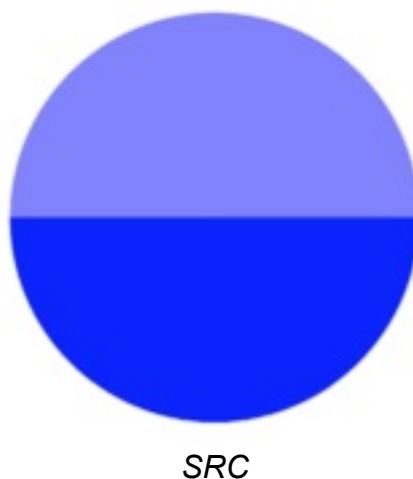
验证的代码如下：

```
@Override  
protected void onDraw(Canvas canvas) {  
    super.onDraw(canvas);  
    // draw background  
    canvas.drawColor(Color.WHITE);  
    int count = canvas.saveLayer(0, 0, width, height, p, Canvas.ALL_SAVE_FLAG);  
    canvas.save();  
    canvas.scale(0.5f, 0.5f, width / 2, height / 2);  
  
    canvas.drawBitmap(mDst, 0, 0, p);  
    p.setXfermode(xfermode);  
    canvas.drawBitmap(mSrc, 0, 0, p);  
    p.setXfermode(null);  
    canvas.restore();  
  
    canvas.restoreToCount(count);  
}
```

这里的mSrc以及mDst分别对应我们绘制的SRC bitmap以及DST bitmap；理论已经解释过了，DST代表先画的，下层图像；SRC是后画的上层图像。测试的图像用代码画出来的：

```
@Override  
protected void onSizeChanged(int w, int h, int oldw, int oldh) {  
    super.onSizeChanged(w, h, oldw, oldh);  
    width = w;  
    height = h;  
    float halfWidth = width / 2;  
  
    // DST Rect  
    Paint p = new Paint(Paint.ANTI_ALIAS_FLAG);  
    mSrc = Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888);  
    mDst = Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888);  
  
    Canvas canvas = new Canvas(mDst);  
    p.setColor(Color.RED);  
    canvas.drawRect(0, 0, halfWidth, height, p);  
    p.setAlpha(1 << 7);  
    canvas.drawRect(halfWidth, 0, width, height, p);  
  
    canvas = new Canvas(mSrc);  
    // SRC circle  
    p.setColor(Color.BLUE);  
    p.setAlpha((1 << 8) - 1);  
  
    float radius = Math.min(height, width) / 2;  
    mSrcRect.set(width / 2 - radius, height / 2 - radius,  
                 width / 2 + radius, height / 2 + radius);  
    canvas.drawArc(mSrcRect, 0, 180, true, p);  
    p.setAlpha(1 << 7);  
    canvas.drawArc(mSrcRect, 180, 180, true, p);  
}
```

注意到，每一个图都有半透明和全透明的两周状态，画出来，肉眼看到效果如下：



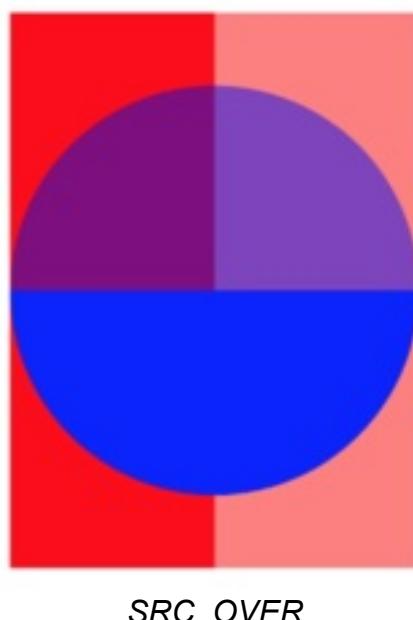


SRC和DST

这个就不是解释了，SRC - [Sa, Sc]只有源图像的alpha和颜色，因此只保留源图像；DST也是一样。

SRC_OVER

[$Sa + (1-Sa) Da , Rc = Sc + (1- Sa) Dc$]。从名字上看，从DST上面绘制SRC图像（透明度的叠加）：



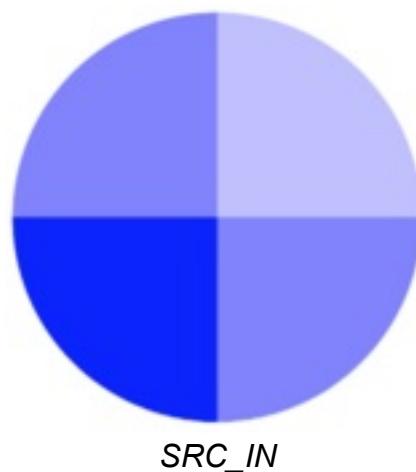
DST_OVER

$[Sa + (1 - Sa) Da, Rc = Dc + (1 - Da) Sc]$ 。与上面情况差不多，看看效果：



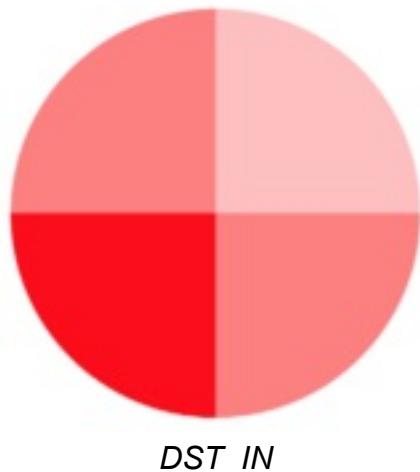
SRC_IN

$[Sa Da, Sc Da]$ 。注意，alpha通道是SRC和DSTalpha的乘法叠加；颜色是SRC颜色与DSTalpha通道的叠加；考虑一下，我们的图像应该是个什么样子；首先确定图像范围。什么时候才会有图像呢，rgb应该有分量，alpha不能为0；所以rgb分量里面只有SRC，说明图像里面区域里面只有源图像；alpha通道只有DST，当DSTalpha为0的地方没有图像（这句话有两个意思，在DST完全透明的地方不存在源图像）简而言之，就是在相交的地方绘制源图像；但是绘制的alpha通道受DST影响：



DST_IN

$[Sa Da, Sa Dc]$ 。按照上面的理解。在相交的地方绘制DST，但是alpha受SRC影响：



SRC_OUT

很多地方解释说：

- 取上层绘制非交集部分。
- 在不相交的地方绘制 源图像。

我们看看是不是这样：



说好的在不相交的地方绘制源图像呢？如果是这个意思，因为DST包含SRC，那么应该整个应该是什么都没有（待商榷，下面说）。为什么相交的地方还是有源图像？

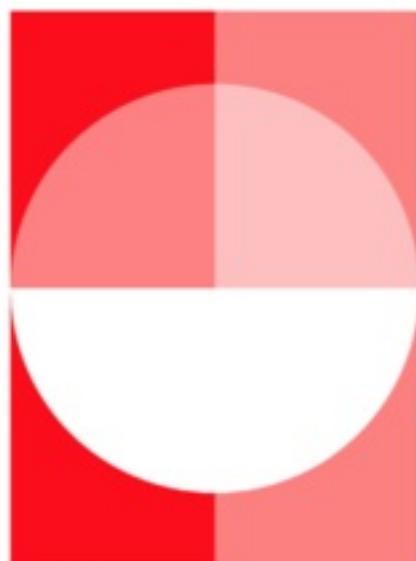
看看这个porterduff公式：[$Sa(1 - Da), Sc(1 - Da)$] 这里对应的rgb是 $Sc * (1 - Da)$ ，

- 1、在不相交的地方 Da 肯定是0，那么不相交的地方就是 Sc 也就是完全是SRC图像；
- 2、在相交的地方是 $Sc * (1 - Da)$ 也就是虽然是SRC的颜色，但是受到DST的alpha通道的影响。
- 3、在相交地方的特殊情况，如果DST完全不透明，那么 $Da = 1$ ；这时候这个表达式值就是0；也就是通常的解释“绘制非交集部分（交集部分没有图像）”

总结一下，这种模式应该是：在不相交的地方绘制源图像，在相交处根据DST的alpha进行过滤；特殊情况下相交处DST不透明，那么相交处没有颜色，如果完全透明（相当于没有DST图像）

DST_OUT

[Da (1 - Sa) , Dc (1 - Sa)] 与上面解释类似，不赘述。

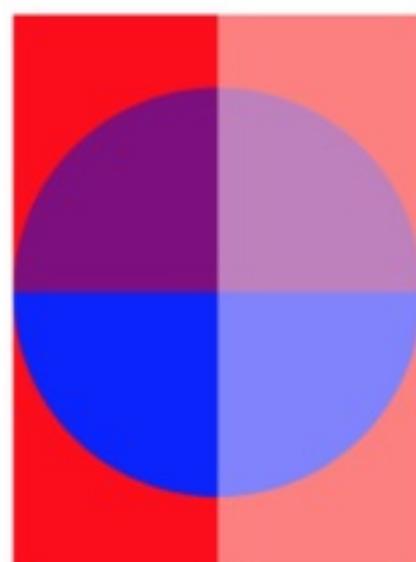


DST_OUT

SRC_ATOP

[Da , Sc Da + (1 - Sa) Dc]。与上面两种模式解释差不多，有一点不同。

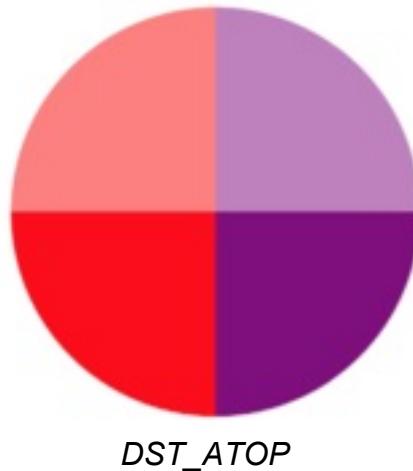
源图像和目标图像相交处绘制源图像，不相交的地方绘制目标图像，并且相交处的效果会受到源图像和目标图像alpha的影响；



*SRC_ATOP***DST_ATOP**

[$S_a, S_a D_c + S_c (1 - D_a)$], 直接上解释。

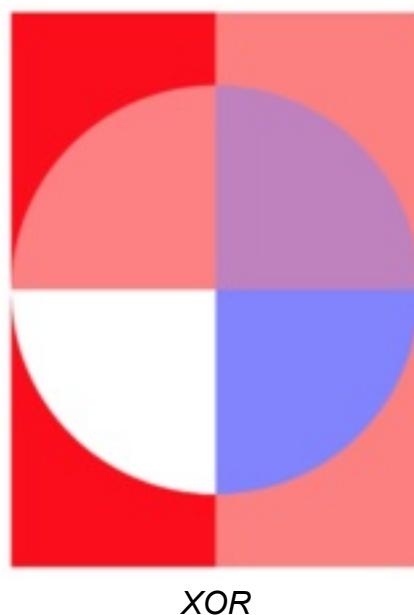
源图像和目标图像相交处绘制目标图像，不相交的地方绘制源图像，并且相交处的效果会受到源图像和目标图像alpha的影响；

**XOR**

[$S_a + D_a - 2 S_a D_a, S_c (1 - D_a) + (1 - S_a) D_c$]

在不相交的地方按原样绘制源图像和目标图像，相交的地方受到对应alpha和色值影响。

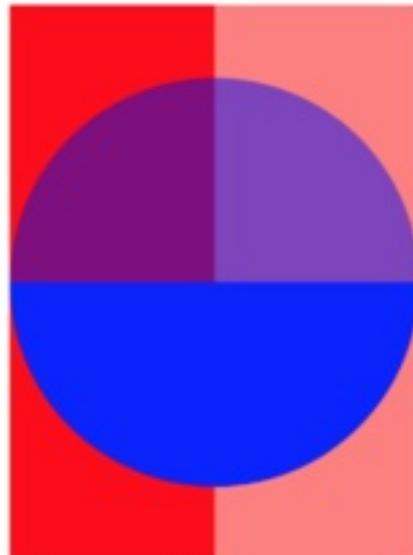
按上面公式进行计算，如果都完全不透明则相交处完全不绘制；



DARKEN

$$[Sa + Da - Sa Da , Sc (1 - Da) + Dc * (1 - Sa) + \min(Sc, Dc)]$$

从算法上看，alpha值变大，色值上如果不透明则取较暗值，非完全不透明情况下使用上面算法进行计算，受到源图和目标图对应色值和alpha值影响；正如名字所说，会感觉效果变暗，即进行对应像素的比较，取较暗值，如果色值相同则进行混合；

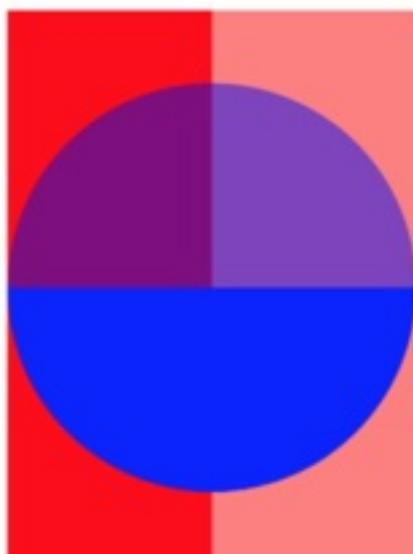


DARKEN

LIGHTEN

$$[Sa + Da - Sa Da , Sc (1 - Da) + Dc * (1 - Sa) + \max(Sc , Dc)]$$

与DARKEN相反，LIGHTEN 的目的则是变亮，如果在均完全不透明的情况下，色值取源色值和目标色值中的较大值，否则按上面算法进行计算；



LIGHTEN

接下来四种 **MULTIFY**, **SCREEN**, **ADD**, **OVERLAY**就不说了，产生的结果不太确定。
自己查阅文档吧。

Memory

Basic

android系统不释放内存吗？

来源:[知乎](#)

问题

Android系统下关闭程序后，系统内存并不释放。即使关掉后台进程，内存也增加不多。据说即使前台关掉进程，其实该进程在后台还在运行（休眠）。why？有人说是因为智能手机无需将程序彻底关掉，可以减少再启动的时间。是这样吗？

回答

高爷

我来逐条回答你的问题把

1.android系统下关闭程序后，系统内存并不释放。

这个是不准确的,只能说对了一半. 你所描述的"android系统下关闭程序",指的是怎么个关闭法呢?目前阶段有好几种关闭程序的方法:

点击Back键退出. 这种退出的方法, 进程是否被杀掉, 取决于这个应用程序的实现. 举个栗子, 如果你创建一个空的应用, 这时候查看系统内存信息(包名为com.example.gaojianwu.myapplication, pid为5708, 内存为13910kb):

```
0000 kB: com.android.providers.settings (pid 1050)
40925 kB: Foreground
    27015 kB: com.meizu.safe (pid 2348)
    13910 kB: com.example.gaojianwu.myapplication (pid 5708 / activities)
39317 kB: Visible
```

可以看到,这个应用程序的pid为5708 , 其优先级为Foreground,即前台程序.

这时候我们点击Back键退出,然后再查看系统的内存信息(`adb shell dumpsys meminfo`)

```
50516 kB: Cached
    14147 kB: android.process.acore (pid 5968)
    12337 kB: com.example.gaojianwu.myapplication (pid 5708)
    10424 kB: com.android.calendar (pid 5790)
        7548 kB: com.android.providers.calendar (pid 5766)
        6060 kB: com.android.providers.usagestats (pid 5942)
```

我们看到,这个程序在Back键之后,其进程5708依旧是存在的.只是其进程优先级变成了Cache.其占用内存变成了12337kb,和之前的13910kb相比是变小了一些.但是大部分内存是没有被释放掉的.

在任务管理器中杀掉应用:

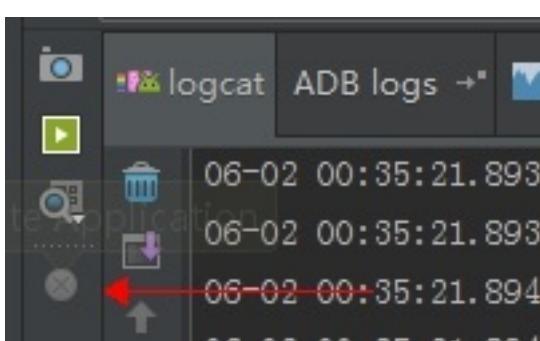
在任务管理器中杀掉应用,这个结果是不一致的,其取决于这个OS的任务管理器的实现,大部分国内的厂家都会对任务管理器进行定制,以达到更有效的杀掉应用的效果.一般来说厂家定制的任务管理器都会比较暴力,除了少数白名单,其他的应用一概直接将进程杀掉.

我们以上面的那个测试程序为例,打开这个程序之后, 其进程优先级为Foreground,这时候我们直接调用任务管理器杀掉改程序(以魅族MX4 Pro为栗子):

```
48138 kB: Cached
    13957 kB: android.process.acore (pid 5968)
    10495 kB: com.android.dialer (pid 6148)
    10317 kB: com.android.calendar (pid 5790)
        7442 kB: com.android.providers.calendar (pid 5766)
        5927 kB: com.android.providers.usagestats (pid 5942)
```

可以看到用任务管理器杀掉之后, 整个应用程序的进程都被杀掉了.

通过命令行或者开发者工具杀掉应用. 我们可以通过 `adb shell am force-stop` 包名来杀掉这个程序,其结果也是进程直接被杀掉. IDE(比如Android Studio)选择一个进程后,点击:



也是可以干掉这个进程的.

2.即使关掉后台进程, 内存也增加不多。

这个不对,一个进程被杀死后,其内存会被释放掉的.

我们以知乎App Android客户端为栗子:

打开这个程序之前,系统剩余内存,以(MX4 Pro为栗子):

```
Free RAM: 1837160 kB (48172 cached pss + 1586368 cached + 202620 free)
```

打开这个程序之后,系统剩余内存:

```
Free RAM: 1805784 kB (123856 cached pss + 1644064 cached + 37864 free)
```

知乎占用的内存:

```
49907 kB: com.zhihu.android (pid 6389 / activities)
```

使用任务管理器杀掉知乎,系统剩余内存:

```
Free RAM: 1862296 kB (139596 cached pss + 1639200 cached + 83500 free)
```

可以看到,杀掉进程之后,内存是会增加的.

3.据说即使前台关掉进程, 其实该进程在后台还在运行(休眠)。why?

这个和第一条一样,取决于你关掉进程的方法.

另外像豌豆荚这样的应用,他会起好几个进程:

```
90270 kB: Foreground  
65914 kB: com.wandoujia.phoenix2 (pid 7304 / activities)  
18846 kB: com.wandoujia.phoenix2.usbproxy (pid 5584)  
  
16159 kB: com.wandoujia.framework.webdownload (pid 7430)
```

当我们用任务管理器杀掉他后,

```
119134 kB: B Services  
20285 kB: com.wandoujia.phoenix2.usbproxy (pid 5584)
```

这个进程不会被杀掉. 因为人家就是需要在后台跑一个Services来维持usb的链接.休眠?

NO NO NO, 人家还是要干活的...

想想也是哈,如果我把这个进程也干掉了,那手机不就连不上电脑了么???

以此类推:

要是我把QQ的

```
115610 kB: Perceptible
    36497 kB: com.tencent.mobileqq (pid 8257)
    36489 kB: com.baidu.input_mz (pid 1184)
    22685 kB: com.tencent.mobileqq:MSF (pid 8290)
```

干掉的话,不就收不到推送信息了?不行不行,得留着.

要是.....?不行不行,得留着.
要是.....?不行不行,得留着.
要是.....?不行不行,得留着.
要是.....?不行不行,得留着.
要是.....?不行不行,得留着.

警告:系统内存不足!!!!!!!!!!

系统:杀杀杀!

用户:啥破手机,老是杀我后台!!!

手机:怪我咯?

APP:怪我咯?

程序员:怪我咯?产品狗说要加这么多功能的,还要一直后台接受消息的..

产品狗:怪我咯?自己技术不行...

4. 有人说是因为智能手机无需将程序彻底关掉, 可以减少再启动的时间。是这样吗?

这个说法前半句是不对的,后半句是对的.

先说前半句:Android设计的时候,确实是想让大家不去关心内存问题,Android会有一套自己的内存管理机制,在内存不足的时候通过优先级干掉一些应用,这个[@monkey code](#)已经说了. 每个应用在接收到内存不足的信号(之前是onLowMemory,现在一般用onTrimMemory, onLowMemory的级别相当于onTrimMemory中的最严重的哪个等级). 会根据内存不足的程度,来释放掉一部分内存.以保持自己的进程不被杀死,这样下次启动的时候就不用去fork zygote. 但是.....凡是总有个但是, 理想是丰满的,现实是骨干的. 严格按照Google想的那一套去做的应用不多,国内开发者对内存的敏感程度很低,导致很多应用程序跑起来分分钟就100-200MB了,墨迹天气这样的应用,400m妥妥的(不好意思又黑了墨迹天气). 所以手机低内存的情况非常常见,这时候大部分应用并没有重写onTrimMemory方法, 所以低内存的情况会很频繁. 这时候你再起一个应用,申请内存的时候发现内存不够,就开始杀应用了. 所以经常会出现你在看电子书,突然这时候微信来了个消息,你切过去回了个消息,打开相机拍了个照,然后发给朋友, 又发了条微博,再回来

看书的时候发现电子书已经挂了,正在重新加载程序....WLGQ...

这时候你就发现杀掉进程的重要性了, 把不重要的进程直接干掉, 保证重要的进程不会被系统杀掉.

所以说不重要的程序是需要在使用结束后直接干掉的. 一劳永逸, 麻麻再也不用担心这货偷跑流量/后台安装程序/占内存/占CPU了....

再说后半句: 可以减少启动的时间. 这个是对的, 如果一个应用程序的进程没有被杀死, 那么下一次启动这个应用程序的时候, 就不需要去创建这个进程了(fork zygote, 这个耗时还是蛮多的), 而是直接在这个进程中创建对应的组件即可(Android四大组件).

update 2015-6-3

补充:

- 1.关于墨迹天气

下面是我抓的墨迹天气的内存使用:

```
Total RSS by OOM adjustment:  
226158 kB: Native  
    187226 kB: com.meizu.mjweather (pid 6642) ←  
    61511 kB: surfaceflinger (pid 2350)  
    15666 kB: mediaserver (pid 2367)  
    11472 kB: zygote (pid 2380)  
    6989 kB: myEnOcean (pid 2391)  
  
383338 kB: Foreground  
    383338 kB: com.meizu.mjweather (pid 6362 / activities) ←  
    95299 kB: Visible  
        43189 kB: com.android.launcher3 (pid 3678 / activities)  
        33289 kB: com.meizu.net.search (pid 3922)  
        6229 kB: com.meizu.mjweather (pid 6362)  
  
66146 kB: A Services  
    18721 kB: com.meizu.mjweather:pushservice (pid 6653) ←  
    12280 kB: com.meizu.media.music (pid 4171)  
    10369 kB: com.android.process.media (pid 4425)
```

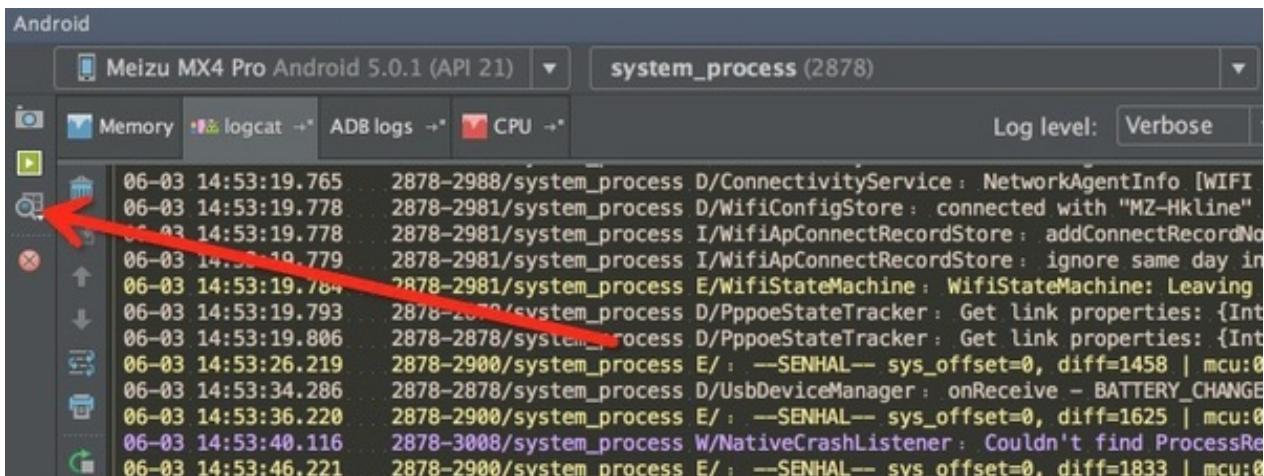
三个进程, 一个在Native, 一个是前台进程, 还有一个推送的Service。

- 2.内存工具

另外有人问我是怎么抓的, 其实就是上面提到的那个命令:

```
|  adb shell dumpsys meminfo
```

另外Android Studio提供了简单的图形操作方式:



弹出的框选第三个：



然后就会有一个报告自动打开。

monkey code

简单点说，这样做好处就是再次打开这个应用的时候速度会变快，因为节省了fork，重新解析加载布局等操作，直接resume就可以了，这个也就是冷启动和热启动的区别。

这样做的是基于OOM_ADJ的，当系统内存的压力值达到某个地步的时候，就会开始杀对应优先级的进程。

一个很好的例子就是当你进入大游戏后回到主菜单，主菜单会有一个重新加载的过程，但是状态栏就没有，这个过程简单点说可以是这样的：

进入大游戏，逐渐申请内存，前台大游戏的OOM_ADJ值是0，后台的launcher是6，当逐渐内存压力变大的时候，lmk就开始杀进程，于是从最低优先级的进程

(CACHED_APP_MAX_ADJ = 15) 杀起，一路杀到launcher，如果发现够了，就不再杀。而从大游戏退出到launcher后，重新创建launcher，于是有了重新加载的过程。

而systemui由于是persist进程优先级为-12，因此不会杀到。

希望你能理解~

一转倾心

我的愚见是：之前使用过的手机基本上都遵循这个规律

1.一键清理后台之后，你可以到“系统设置-所有应用程序”的地方看看，正在运行的程序之中会有一些没有成功清理，这就是传说中的不清理后台的程序列表，大概就是系统自愿或者不自愿地把它们放进了白名单。例如微信，微博，音乐播放器。通常，我会选择点击这些程序手动停止，遇到部分比较顽强的抵抗，我也只有从了。

2.同样在正在运行的界面，点击显示“缓存后台程序”，这些程序和前面的不同，一般是不会占用系统内存的，也就是非活跃的内存区域，简而言之可以忽略，而且你也无法手动停止他们。

3.根据以上两点现象，我们讨论一下android的后台机制，和WP，ios那种假后台不同，安卓系统和windows比较类似，采用了多任务的机制，所以原则上内存越大，能够并行的多任务越多。如果有一键清理，强迫症的用户，例如我，就会经常点击一下，这是手动释放内存的方法。安卓自动释放内存的规律是，等到内存池溢出，安卓系统才会强制根据算法进行筛选，将最不常用的进程，或者最早的进程砍掉，腾出空间。

4.这就是你总感觉系统不流畅，尤其是切换多任务的时候，出现了部分应用假死也是常有的事，这好比windows，我们也要经常到资源管理器杀进程

5.一键清理后台也有弊病，当我们过于频繁清理后台之后，我们会发现部分应用打开的时间，切换应用的时间同样需要等待，这时候不是内存空间不足导致，而是重新加载到内存所导致。

6.所以即使是升级到android4.4之后，我也不建议用户购买1GB内存以下的机型，因为随着我们并行的任务逐渐增多，谷歌再优化也是无补于是，倒不如根据我们的使用习惯，适当地选择大内存的手机更划算，2GB机型已经开始普及到千元以下的价位，3GB基本上成为旗舰机标配，今年，华硕，乐视的机型也上了4GB的配置，所以只要不是ios和WP阵营，大内存还是很有必要的。

booking.com android客户端的bitmap复用

来源:blog.aaapei.com

前言

被鞭炮吵得睡不着觉，rss中找一篇简单的文档翻译下，原文链接：

<http://blog.booking.com/android-reuse-bitmaps.html>, 大部分团队应该都做过这个bitmap优化，不过估计设置过BitmapFactory.Options.inTempStorage参数的应该不多 :)

booking.com android客户端在新版本的增加了一个新功能：酒店的图片集合

The screenshot shows a mobile application interface for a hotel search. At the top, there's a blue header bar with the 'B. Hotel' logo, a folder icon with a heart, a share icon, and a more options icon. The main content area features a large image of a city skyline with a prominent church tower, labeled '1/28'. Below the image, the hotel name 'Swissôtel Amsterdam' is displayed in white text, along with a five-star rating icon.

**Damrak 96, Amsterdam City Center,
Amsterdam, 1012 LP**

Rooms available from: **€205**

Value Deal

Price for 2 guests for 1 night (May 27 - May 28)
No booking or credit card fees
Pay Later on selected rooms

We speak your language!
English spoken here.

Very good

Select rooms

不幸的是，增加了这个新功能后，发现这个应用的内存消耗增长了20%。图片集的界面的滑动有明显的卡顿，经定位，我们发现viewpager加载图片时的gc问题造成了以上的问题。由于应用的图片资源多；控件布局层次复杂；数据量较大，造成内存的申请很容易触发GC。

当申请bitmap内存时，logcat输出信息如下：

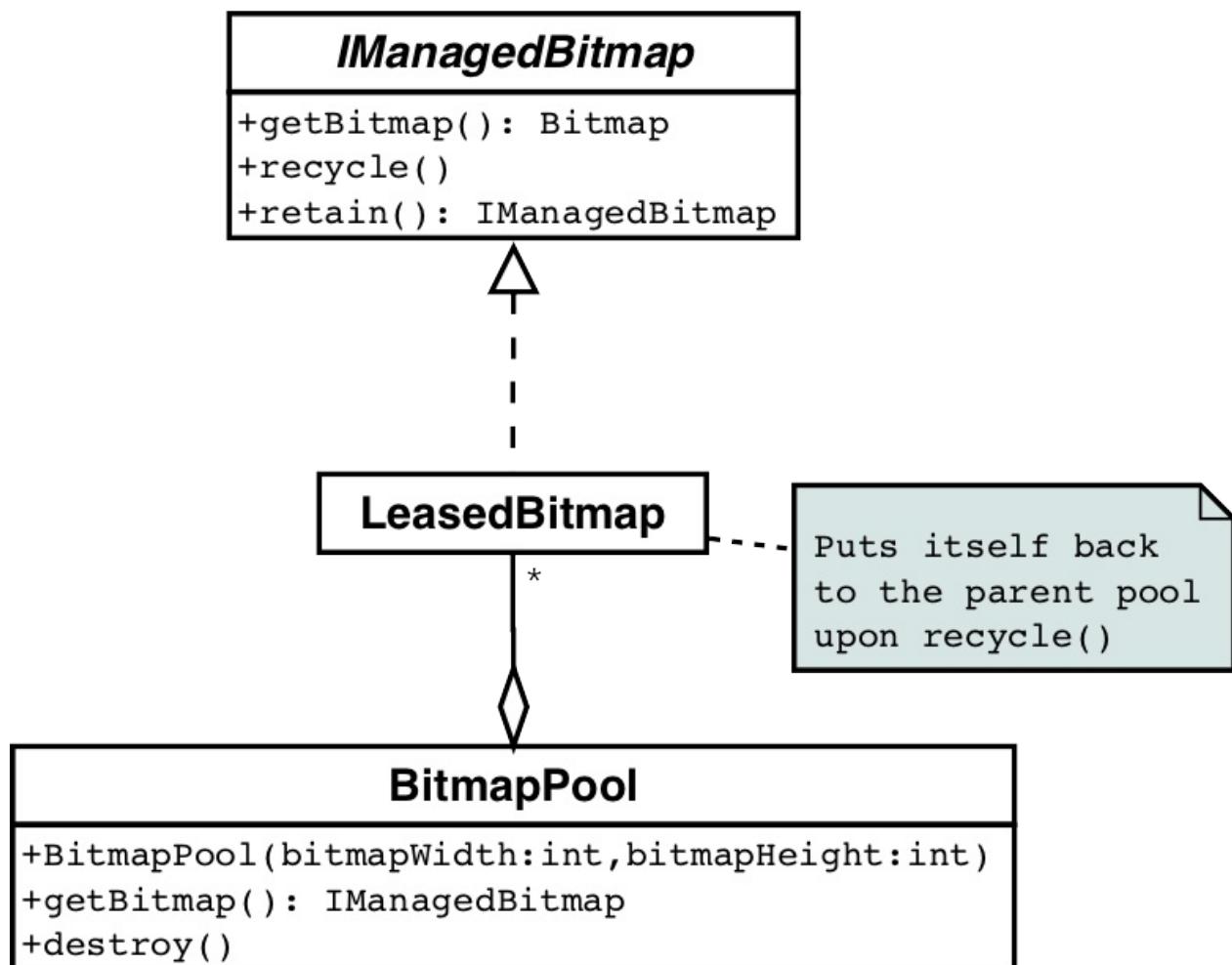
```
GC_FOR_ALLOC freed 3255K, 20% free 21813K/26980K, paused 62ms, total 62ms
GC_FOR_ALLOC freed 710K, 20% free 30242K/37740K, paused 72ms, total 72ms
GC_FOR_ALLOC freed <1K, 20% free 31778K/39280K, paused 74ms, total 74ms
```

通过日志信息可以知道，一个bitmap图片的申请，造成应用约70ms的gc停顿，导致应用程序掉5次左右的帧。为了保证应用的流畅体验，必须保证gc停顿的时间降到16ms以下。我们决定利用inBitmap参数进行图片资源的复用，不过这个参数必须保证图片的大小一致；幸好，在android4.4之后，二次复用的图片不需要严格遵守这个规则，只需保证不比原图片资源大即可。

基于这个api，我们在viewpager adapter中抽象了一个图片池管理图片复用。当一个imageview移除屏幕以外时，apater管理bitmap的生命周期，将其关联的bitmap buffer内存放置到图片池中而不是直接销毁。

bitmap的生命周期管理

为了管理bitmap内存，需要为bitmap进行引用计数，引用技术的接口是这样的；



```

package com.booking.util.bitmap;

import android.graphics.Bitmap;

/**
 * A reference-counted Bitmap object. The Bitmap is not really recycled
 * until the reference counter drops to zero.
 */
public interface IManagedBitmap {

    /**
     * Get the underlying {@link Bitmap} object.
     * NEVER call Bitmap.recycle() on this object.
     */
    Bitmap getBitmap();

    /**
     * Decrease the reference counter and recycle the underlying Bitmap
     * if there are no more references.
     */
    void recycle();

    /**
     * Increase the reference counter.
     * @return self
     */
    IManagedBitmap retain();
}

```

其中bitmappool类管理bitmap集合，当不存在bitmap内存时，或新申请，或复用已有内存。 bitmapPool类不直接引用bitmap，而通过IManagedBitmap进行bitmap的引用计数。

由于我们只在主线程进行imageview的创建和销毁，我们尚未对BitmapPool进行线程安全同步，如果你需要在后台线程申请位图资源，请自行进行线程同步。

BitmapPool的代码片段是这样的：

```

package com.booking.util.bitmap;

import java.util.Stack;

import android.graphics.Bitmap;
import android.os.Handler;

/**
 * A pool of fixed-size Bitmaps. Leases a managed Bitmap object
 * which is tied to this pool. Bitmaps are put back to the pool
 * instead of actual recycling.
 */

```

```

 * WARNING: This class is NOT thread safe, intended for use
 *           from the main thread only.
 */
public class BitmapPool {
    private final int width;
    private final int height;
    private final Bitmap.Config config;
    private final Stack<Bitmap> bitmaps = new Stack<Bitmap>();
    private boolean isRecycled;

    private final Handler handler = new Handler();

    /**
     * Construct a Bitmap pool with desired Bitmap parameters
     */
    public BitmapPool(int bitmapWidth,
                      int bitmapHeight,
                      Bitmap.Config config)
    {
        this.width = bitmapWidth;
        this.height = bitmapHeight;
        this.config = config;
    }

    /**
     * Destroy the pool. Any leased IManagedBitmap items remain valid
     * until they are recycled.
     */
    public void recycle() {
        isRecycled = true;
        for (Bitmap bitmap : bitmaps) {
            bitmap.recycle();
        }
        bitmaps.clear();
    }

    /**
     * Get a Bitmap from the pool or create a new one.
     * @return a managed Bitmap tied to this pool
     */
    public IManagedBitmap getBitmap() {
        return new LeasedBitmap(bitmaps.isEmpty()
                               ? Bitmap.createBitmap(width, height, config) : bitmaps.pop());
    }

    private class LeasedBitmap implements IManagedBitmap {
        private int referenceCounter = 1;
        private final Bitmap bitmap;

        private LeasedBitmap(Bitmap bitmap) {
            this.bitmap = bitmap;
        }
    }
}

```

```
    @Override
    public Bitmap getBitmap() {
        return bitmap;
    }

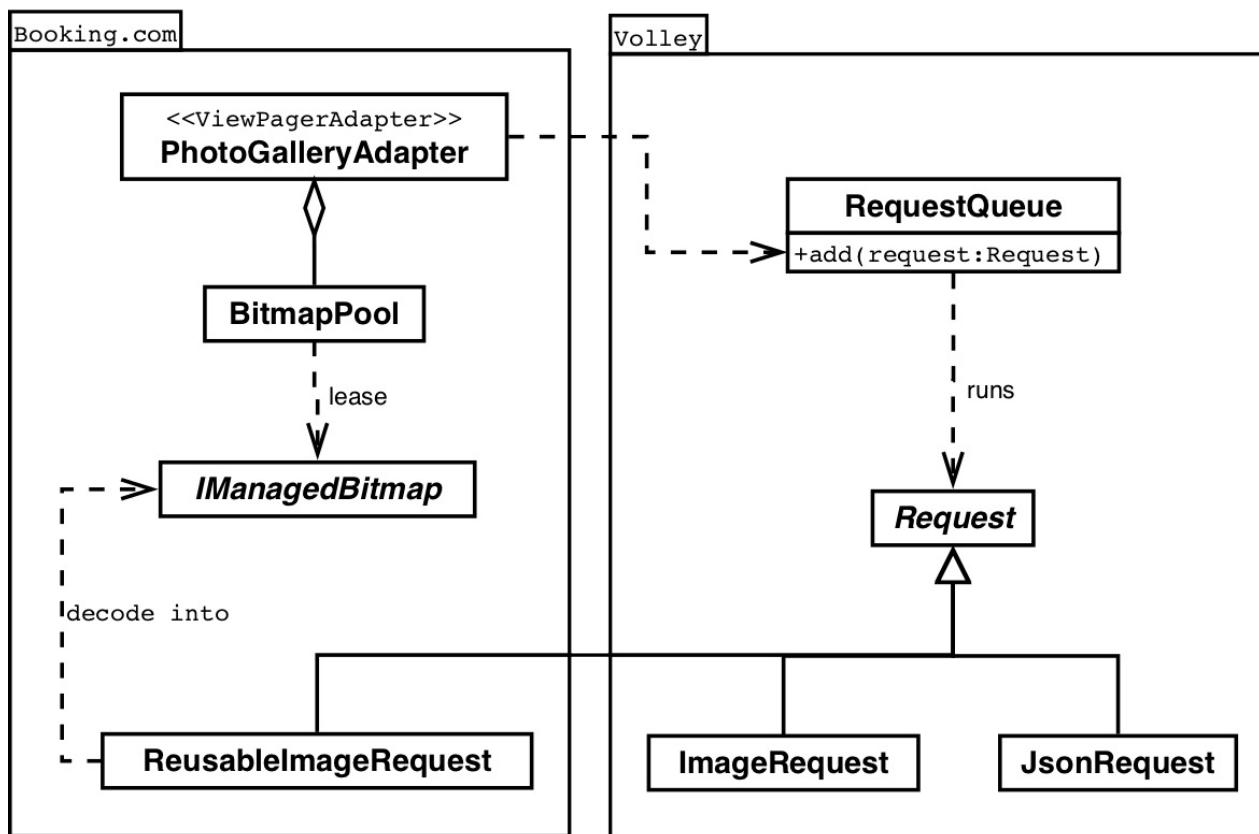
    @Override
    public void recycle() {
        handler.post(new Runnable() {
            @Override
            public void run() {
                if (--referenceCounter == 0) {
                    if (isRecycled) {
                        bitmap.recycle();
                    } else {
                        bitmaps.push(bitmap);
                    }
                }
            }
        });
    }

    @Override
    public IManagedBitmap retain() {
        ++referenceCounter;
        return this;
    }
}
```

网络层

Booking.com客户端的网络通信层使用Volley框架，默认情况下，Volley通过ImageRequest进行网络图片的bitmap的获取，为了和ImagePool集成，我们实现了一个自定义的ImageRequest（ReusableImageRequest），ReusableImageRequest内部持有一个IManagedBitmap进行bitmap的解码。为了避免内存泄漏，当ReusableImageRequest被取消时，需要有机制通知IManagedBitmap进行引用释放，因此，我们为ReusableImageRequest扩展了一个onFinished方法。

与volley的结构图是这样的：



其他工作

当我们实现了一个自定义的ImageRequest时，我们还利用 `BitmapFactory.Options.inTempStorage`. 参数进行了图片解码的优化。inTempStorage可以预申请一块内存，对所有解码过程中指定相同的内存，以达到减少临时内存的目的。

java里的totalMemory()、maxMemory()、freeMemory()究竟是什么

来源:www.jcodecraeer.com

- totalMemory() : 返回 Java 虚拟机中的内存总量。
- maxMemory() : 返回 Java 虚拟机试图使用的最大内存量。
- freeMemory() : 返回 Java 虚拟机中的空闲内存量。

这是API的解释。

我写了这么一段代码

```
public class RuntimeDemo{  
    public static void main(String[] args) throws Exception{  
        //构造方法被私有化了。  
        Runtime myRun = Runtime.getRuntime();  
        System.out.println("已用内存" + myRun.totalMemory());  
        System.out.println("最大内存" + myRun.maxMemory());  
        System.out.println("可用内存" + myRun.freeMemory());  
        String i = "";  
        long start = System.currentTimeMillis();  
        System.out.println("浪费内存中.....");  
        for(int j = 0;j < 10000;j++){  
            i += j;  
        }  
        long end = System.currentTimeMillis();  
        System.out.println("执行此程序总共花费了" + (end - start) + "毫秒");  
        System.out.println("已用内存" + myRun.totalMemory());  
        System.out.println("最大内存" + myRun.maxMemory());  
        System.out.println("可用内存" + myRun.freeMemory());  
        myRun.gc();  
        System.out.println("清理垃圾后");  
        System.out.println("已用内存" + myRun.totalMemory());  
        System.out.println("最大内存" + myRun.maxMemory());  
        System.out.println("可用内存" + myRun.freeMemory());  
  
    }  
}
```

下面是打印的结果

```
已用内存31064064  
最大内存460455936  
可用内存30714320  
浪费内存中.....  
执行此程序总共花费了659毫秒  
已用内存193593344  
最大内存460455936  
可用内存150298760  
清理垃圾后  
已用内存193724416  
最大内存460455936  
可用内存192511928
```

关于可用内存的解释：

我们知道，JAVA程序本身是不能直接在计算机上运行的，它需要依赖于硬件基础之上的操作系统和JVM（JAVA虚拟机）。JAVA程序启动时JVM都会分配一个初始内存和最大内存给这个应用程序。这个初始内存和最大内存一定程度上会影响应用程序的性能。

JVM其实就是操作系统上的一个普通程序（进程名叫java，这个程序可以解释执行class文件，系统中当前运行了多少个java程序就会有多少个java进程）。当java进程启动时会首先分配一块堆内存（最小内存），以后每当java程序要求JVM（java进程）分配内存时，JVM就会在预先分配的那块内存上为java程序分配内存，当预先分配的那块内存用完时，JVM会再向操作系统要内存（物理内存），但是JVM不会无限制的向操作系统要内存，当它占用的实际内存达到一个预定值（最大可用内存）时，如果java程序还向JVM要内存，并且JVM无法通过垃圾回收机制回收当前堆中的内存来为java程序服务时，它就会给程序抛出异常：

`java.lang.OutOfMemoryError`。其中内存回收时机并不是在用掉内存达到最大可用内存时才进行，它的运行时机是不确定的。可见JVM的最大可用内存就是java程序能够使用的最大内存。例如：我们把某JAVA程序的JVM最大可用内存设为200M，而我们的物理内存是1G。这种情况下，我们的java程序最多能使用200M内存，虽然我们可能还有800M的内存可用，但是当我们的程序用掉200M后，如果再要内存，JVM不会因为我们还有800M的内存而为我们分配内存，它会抛出`java.lang.OutOfMemoryError`异常。

如何偷Android的内存 – Tricking Android MemoryFile

来源:www.jcodecraeer.com

编辑推荐：[稀土掘金](#)，这是一个高质量的技术干货分享社区，web前端、Android、iOS、设计资源和产品，满足你的学习欲望。

原文标题：[Tricking Android MemoryFile](#)

原文作者：[Ragnarok](#)

之前在做一个内存优化的时候，使用到了MemoryFile，由此发现了MemoryFile的一些特性以及一个非常tricky的使用方法，因此在这里记录一下

What is it

MemoryFile是android在最开始就引入的一套框架，其内部实际上是封装了android特有的内存共享机制[Ashmem](#)匿名共享内存，简单来说，Ashmem在Android内核中是被注册成一个特殊的字符设备，Ashmem驱动通过在内核的一个自定义[slab](#)缓冲区中初始化一段内存区域，然后通过mmap把申请的内存映射到用户的进程空间中（通过[tmpfs](#)），这样子就可以在用户进程中使用这里申请的内存了，另外，Ashmem的一个特性就是可以在系统内存不足的时候，回收掉被标记为"unpin"的内存，这个后面会讲到，另外，MemoryFile也可以通过Binder跨进程调用来让两个进程共享一段内存区域。由于整个申请内存的过程并不在Java层上，可以很明显的看出使用MemoryFile申请的内存实际上是并不会占用Java堆内存的。

MemoryFile暴露出来的用户接口可以说跟他的名字一样，基本上跟我们平时的文件的读写基本一致，也可以使用InputStream和OutputStream来对其进行读写等操作：

```
MemoryFile memoryFile = new MemoryFile(null, inputStream.available());
memoryFile.allowPurging(false);
OutputStream outputStream = memoryFile.getOutputStream();
outputStream.write(1024);
```

上面可以看到allowPurging这个调用，这个就是之前说的"pin"和"unpin"，在设置了allowPurging为false之后，这个MemoryFile对应的Ashmem就会被标记成"pin"，那么即使在android系统内存不足的时候，也不会对这段内存进行回收。另外，由于Ashmem默认

都是"unpin"的，因此申请的内存 在某个时间点内都可能会被回收掉，这个时候是不可以再读写了

Tricks

MemoryFile是一个非常tricky的东西，由于并不占用Java堆内存，我们可以将一些对象用MemoryFile来保存起来避免GC，另外，这里可能android上有个BUG：

在4.4及其以上的系统中，如果在应用中使用了MemoryFile，那么在dumpsyst meminfo的时候，可以看到多了一项Ashmem的值：

```

ragnarok@ragnarok-MacBook-Pro:~/Works/MMSource/micromessenger_android(unstable/RB-6.5-v3/sns_memory) * adb shell dumpsys meminfo com.ragnarok.memoryfiletest
Applications Memory Usage (kB):
Uptime: 93423614 Realtime: 611817961

** MEMINFO in pid 7947 [com.ragnarok.memoryfiletest] **
              Pss   Private  Private  Swapped    Heap    Heap    Heap
              Total    Dirty    Clean    Dirty   Size   Alloc   Free
-----+-----+-----+-----+-----+-----+-----+-----+
Native Heap      0       0       0       0     5056    3689      6
Dalvik Heap  3041    2348       0       0    18276   18235     41
Dalvik Other   234     224       0       0
Stack          92      92       0       0
Ashmem         980     980       0       0
Other dev     13292   3656    9616       0
.so mmap       1736     544      28       0
.apk mmap        72      0      24       0
.ttf mmap        2      0      0       0
.dex mmap     2139     0     2052       0
code mmap      689     0      68       0
image mmap    1069     792      4       0
Other mmap        6      4       0       0
Unknown        1966   1956       0       0
TOTAL        25318   10596   11792       0    23332   21924      47

Objects
  Views:           15  ViewRootImpl:           1
  AppContexts:      3  Activities:            1
  Assets:           3  AssetManagers:        3
  Local Binders:     9  Proxy Binders:       27
  Death Recipients: 12
  OpenSSL Sockets:  0

SQL
  MEMORY_USED:      0
  PAGECACHE_OVERFLOW: 0
  MALLOC_SIZE:        0

```

可以看出来虽然MemoryFile申请的内存不计入Java堆也不计入Native堆中，但是占用了Ashmem的内存，这个实际上是算入了app当前占用的内存当中

但是在4.4以下的机器中时，使用MemoryFile申请的内存居然是不算入app的内存中的：

```

ragnarok@ragnarokMBP:~/Exercise/android/MemoryFileTest/app/src/main/assets » adb shell dumpsys meminfo com.ragnarok.memoryfiletest
Applications Memory Usage (kB):
Uptime: 921872 Realtime: 921861

** MEMINFO in pid 9383 [com.ragnarok.memoryfiletest] **
              Shared    Private   Heap   Heap
              Pss      Dirty     Dirty   Size   Alloc   Free
-----+-----+-----+-----+-----+-----+
Native      20       16      20    4176    3544    355
Dalvik    2023     5928    1876  10916    2893   8023
Cursor        0       0       0
Ashmem       0       4       0
Other dev   1852    1168    1284
.so mmap    2310    2336    992
.jar mmap     0       0       0
.apk mmap     71       0       0
.ttf mmap      2       0       0
.dex mmap   1240       0      32
Other mmap    477     16      88
Unknown     1856    492    1852
TOTAL      9851    9960    6144   15092    6437   8378

Objects
Views:          14   ViewRootImpl:           1
AppContexts:      4   Activities:            1
Assets:          3   AssetManagers:         3
Local Binders:    7   Proxy Binders:        15
Death Recipients: 0
OpenSSL Sockets: 0

SQL
MEMORY_USED:    0
PAGECACHE_OVERFLOW: 0
MALLOC_SIZE:     0

```

而且这里我也算过，也是不算入Native Heap中的，另外，这个时候去系统设置里面看进程的内存占用，也可以看出来其实并没有计入Ashmem的内存的

这个应该是android的一个BUG，但是我搜了一下并没有搜到对应的issue，搞不好这里也可能是一个feature

而在大名鼎鼎的Fresco当中，他们也有用到这个bug来避免在decode bitmap的时候，将文件的字节读到Java堆中，使用了MemoryFile，并利用了这个BUG然这部分内存不算入app中，这里分别对应了Fresco中的[GingerbreadPurgeableDecoder](#)和[KitKatPurgeableDecoder](#)，Fresco在decode图片的时候会在4.4和4.4以下的系统中分别使用这两个不同的decoder

从这个地方可以看出来，使用MemoryFile，在4.4以下的系统当中，可以帮我们的app额外"偷"一些内存，并且可以不计入app的内存当中

Summary

这里主要是简单介绍了MemoryFile的基本原理和用法，并且阐述了一个MemoryFile中一个可以帮助开发者"偷"内存的地方，这个是一个非常tricky的方法，虽然4.4以下使用这块的内存并不计入进程当中，但是并不推荐大量使用，因为当设置了allowPurging为false的时候，这个对应的Ashmem内存区域是被"pin"了，那么在android系统内存不足的时候，是不能够把这段内存区域回收的，如果长时间没有释放的话，这样子相当于无端端占用了大量手机内存而又无法回收，那对系统的稳定性肯定会造成影响

References

- [Android系统匿名共享内存Ashmem（Anonymous Shared Memory）驱动程序源代码分析](#)
- [Android Kernel Features\(Ashmem\)](#)

简析Android的垃圾回收与内存泄露

来源:[简书](#)

Android系统是运行在Java虚拟机上的，作为嵌入式设备，内存往往非常有限，了解Android的垃圾回收机制，可以有效的防止内存泄露问题或者OOM问题。本文作为入门文章，将浅显的讨论垃圾回收与内存泄露的原理，不讨论Dalvik虚拟机底层机制或者native层面的问题。

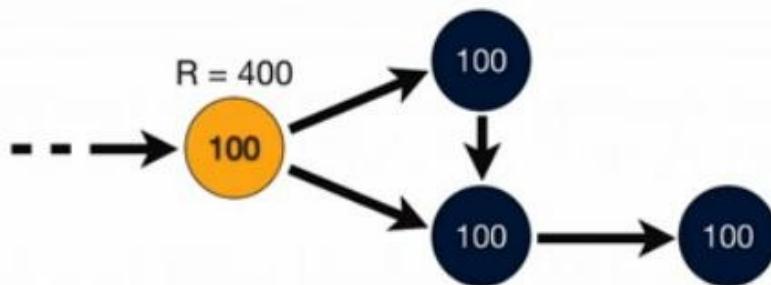
1. 基础

在分析垃圾回收前，我们要复习Java与离散数学的基础。

- **实例化：**对象是类的一个实例，创建对象的过程也叫类的实例化。对象是以类为模板来创建的。比如`Car car = new Car();`，我们就创造了一个Car的实例（Create new class instance of Car）
- **引用：**某些对象的实例化需要其它的对象实例，比如`ImageView`的实例化就需要`Context`对象，就是表示`ImageView`对于`Context`持有引用（`ImageView holds a reference to Context`）。
- **有向图：**在每条边上都标有有向线段的图称为有向图，Java中的garbage collection采用有向图的方式进行内存管理，箭头的方向表示引用关系，比如`B ← A`，就是B中需要A，B引用A。
- **可达：**有向图 $D=\{S,R\}$ 中，对于 $S_i, S_j \in S$ ，如果从 S_i 到 S_j 有任何一条通路存在，则可称 S_i 可达 S_j 。也就是说，当 $B \leftarrow A$ 中间箭头断了，就称作不可达，这时A就不可达B了。
- **Shallow heap与Retain heap的对比**
 - **Shallow heap**表示当前对象所消耗的内存
 - **Retained heap**表示当前对象所消耗的内存加上它引用的内存总合

Eclipse Memory Analyzer (MAT)

- Download from <http://eclipse.org/mat/>
- “Shallow heap” and “retained heap”



Google I/O

Google I/O 2011: Memory management for Android Apps

上图的橙色的Object是该有向图的起点，它的Shallow heap是100，而它的Retained heap是 $100 + 300 = 400$ 。

2. 什么是垃圾回收

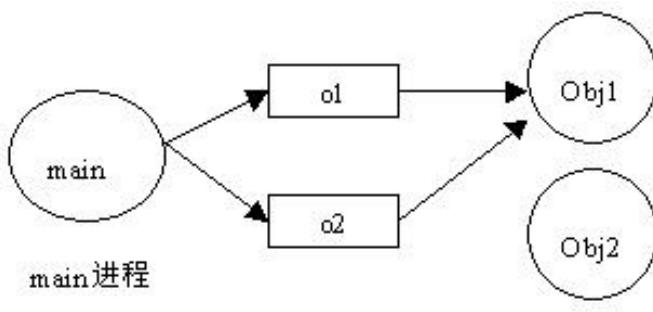
Java GC (Garbage Collection, 垃圾收集, 垃圾回收) 机制，是Java与C++/C的主要区别之一，作为Java开发者，一般不需要专门编写内存回收和垃圾清理代码，对内存泄露和溢出的问题，也不需要像C程序员那样战战兢兢。这是因为在Java虚拟机中，存在自动内存管理和垃圾清扫机制。概括地说，该机制对虚拟机中的内存进行标记，并确定哪些内存需要回收，根据一定的回收策略，自动的回收内存，永不停息（Never Stop）的保证虚拟机中的内存空间，防止出现内存泄露和溢出问题。

3. 什么情况需要垃圾回收

对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常GC采用有向图的方式记录并管理堆中的所有对象，通过这种方式确定哪些对象时“可达”，哪些对象时“不可达”。当对象不可达的时候，即对象不再被引用的时候，就会被垃圾回收。

网上有很多文档介绍可达的关系了，如图，在第六行的时候，o2改变了指向，Obj2就不再引用main的了，即它们是不可达的，Obj2就可能在下次的GC中被回收。

```
class test{
    Public static void main(String a[]){
        Object o1 =new Object();
        Object o2 =new Object();
        o2=o1;
        //此行为第 6 行
    }
}
```



该图描述了第 6 行的内存管理的有向图，
Obj2 是第二次申请的对象，此时为可回收对象

developerWorks Java technology

- 什么是内存泄露 当你不再需要某个实例后，但是这个对象却仍然被引用，**防止被垃圾回收**(Prevent from being garbage collected)。这个情况就叫做**内存泄露**(Memory Leak)。

下面将以[How to Leak a Context: Handlers & Inner Classes](#)这篇文章翻译为例，介绍一个内存泄露。

看如下的代码

```
public class SampleActivity extends Activity {

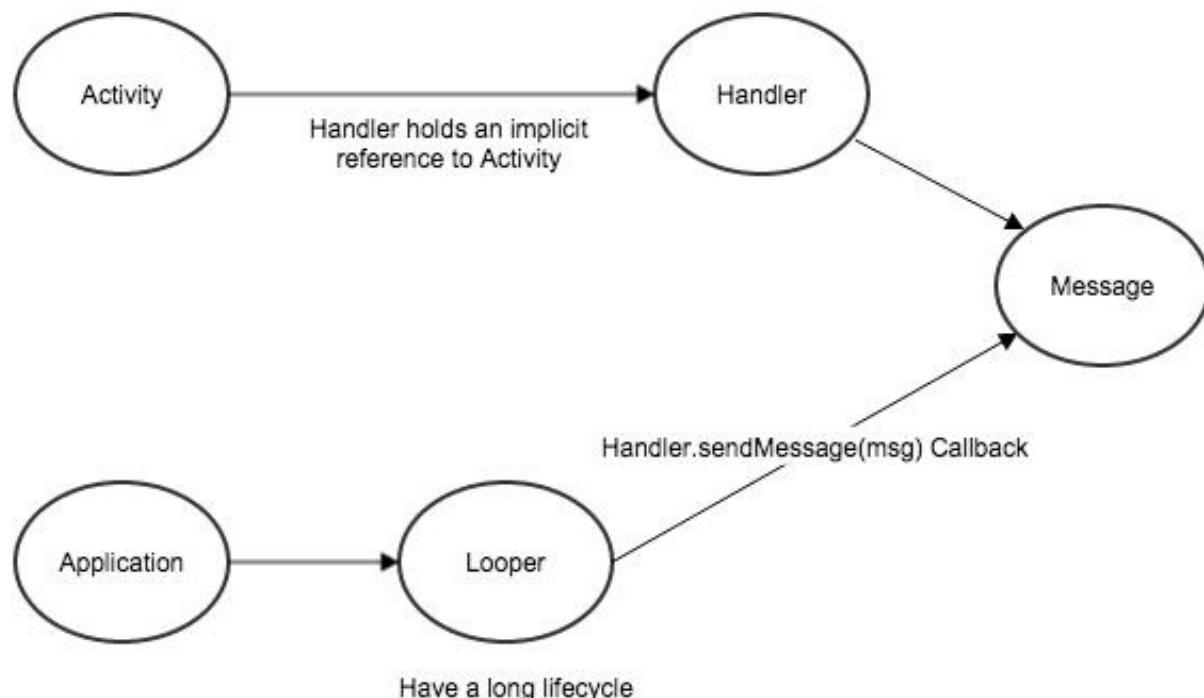
    private final Handler mLeakyHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            // ...
        }
    }
}
```

当你打完这个代码后，IDE应该就会提醒你

In Android, Handler classes should be static or leaks might occur.

- 当你启动一个application时，它会自动在主线程创建一个Looper对象，用于处理Handler中的message。Looper实现了简单的消息队列，在循环中一个接一个的处理Message对象。大多数Application框架事件（比如Activity生命周期调用，按钮点击等）都在Message中，它们在Looper的消息队列中一个接一个的处理。注意Looper是存在于application整个生命周期中。
- 当你新建了一个handler对象后，它会被分配给Looper的消息队列。被发送到消息队列的Message将保持对Handler的引用，因为当消息队列处理到这个消息时，需要使用[Handler#handleMessage\(Message\)](#)这个方法。（也就是说，只要没有处理到这个Message，Handler就一直在队列中被引用）

- 在java中，非静态的内部Class与匿名Class对它们外部的Class有强引用。static inner class除外。



现在，我们尝试运行如下代码

```

public class SampleActivity extends Activity {

    private final Handler mLeakyHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            // ...
        }
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Post a message and delay its execution for 10 minutes.
        mLeakyHandler.postDelayed(new Runnable() {
            @Override
            public void run() { /* ... */ }
        }, 1000 * 60 * 10);

        // Go back to the previous Activity.
        finish();
    }
}

```

这个程序很简单，我们可以脑补一下，它应该是启动了又瞬间关闭，但是事实真的是关闭了吗？

稍有常识的人可以看出，它发送了一个Message，将在十分钟后运行，也就是说Message将被保持引用达到10分钟，这就造成了至少10分钟的内存泄露。

最后正确的代码如下

```
public class SampleActivity extends Activity {

    /**
     * Instances of static inner classes do not hold an implicit
     * reference to their outer class.
     */
    private static class MyHandler extends Handler {
        private final WeakReference<SampleActivity> mActivity;

        public MyHandler(SampleActivity activity) {
            mActivity = new WeakReference<SampleActivity>(activity);
        }

        @Override
        public void handleMessage(Message msg) {
            SampleActivity activity = mActivity.get();
            if (activity != null) {
                // ...
            }
        }
    }

    private final MyHandler mHandler = new MyHandler(this);

    /**
     * Instances of anonymous classes do not hold an implicit
     * reference to their outer class when they are "static".
     */
    private static final Runnable sRunnable = new Runnable() {
        @Override
        public void run() { /* ... */ }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Post a message and delay its execution for 10 minutes.
        mHandler.postDelayed(sRunnable, 1000 * 60 * 10);

        // Go back to the previous Activity.
        finish();
    }
}
```

结论

- GC是按照有向图是否可达来判断对象实例是否有用

- 如果不在需要某个实例，却仍然被引用，这个情况叫做内存泄露
- 匿名类/非静态类内部class会保持对它所在Activity的引用，使用时要注意它们的生命周期不能超过Activity，否则要用static inner class
- 善于在Activy中的生命周期(比如onPause)中**手动控制**其他类的生命周期
- 最后再补充一下iOS的情况，iOS在新版的OC与Swift中，已经引入了新的内存管理体系ARC(auto reference counting，引用自动计数)，C代码在编译时，编译器自动适时的添加释放内存的代码。

References

- <http://www.jianshu.com/p/22e73e80e950>
- <http://www.ibm.com/developerworks/cn/java/l-JavaMemoryLeak/>
- <http://stackoverflow.com/a/70358/4016014>
- <http://blog.csdn.net/luoshengyang/article/details/8852432>
- <http://developer.android.com/training/articles/perf-tips.html>
- <https://techblog.badoo.com/blog/2014/08/28/android-handler-memory-leaks/>

Bitmap

Android 开发绕不过的坑：你的 Bitmap 究竟占多大内存？

来源:bugly.qq.com

0、写在前面

本文涉及到屏幕密度的讨论，这里先要搞清楚 `DisplayMetrics` 的两个变量，摘录官方文档的解释：

- **density**: The logical density of the display. This is a scaling factor for the Density Independent Pixel unit, where one DIP is one pixel on an approximately 160 dpi screen (for example a 240x320, 1.5"x2" screen), providing the baseline of the system's display. Thus on a 160dpi screen this density value will be 1; on a 120 dpi screen it would be .75; etc.

This value does not exactly follow the real screen size (as given by `xdpi` and `ydpi`), but rather is used to scale the size of the overall UI in steps based on gross changes in the display dpi. For example, a 240x320 screen will have a density of 1 even if its width is 1.8", 1.3", etc. However, if the screen resolution is increased to 320x480 but the screen size remained 1.5"x2" then the density would be increased (probably to 1.5).

- **densityDpi**: The screen density expressed as dots-per-inch.

简单来说，可以理解为 `density` 的数值是 `1dp=density px`；`densityDpi` 是屏幕每英寸对应多少个点（不是像素点），在 `DisplayMetrics` 当中，这两个的关系是线性的：

| density | densityDpi |
|----------------|-------------------|
| 1 | 160 |
| 1.5 | 240 |
| 2 | 320 |
| 3 | 480 |
| 3.5 | 560 |
| 4 | 640 |

为了不引起混淆，本文所有提到的密度除非特别说明，都指的是 `densityDpi`，当然如果你愿意，也可以用 `density` 来说明问题。

另外，本文的依据主要来自 *android 5.0* 的源码，其他版本可能略有出入。文章难免疏漏，欢迎指正～

1、占了多大内存？

做移动客户端开发的朋友们肯定都因为图头疼过，说起来曾经还有过 leader 因为组里面一哥们在工程里面加了一张 jpg 的图发脾气的事儿，哈哈。

为什么头疼呢？吃内存呗，时不时还给你来个 OOM 冲冲喜，让你的每一天过得有滋有味（真是没救了）。那每次工程里面增加一张图片的时候，我们都需要关心这货究竟要占多大的坑，占多大呢？Android API 有个方便的方法，

```
public final int getByteCount() {
    // int result permits bitmaps up to 46,340 x 46,340
    return getRowBytes() * getHeight();
}
```

通过这个方法，我们就可以获取到一张 Bitmap 在运行时到底占用多大内存了。

举个例子

一张 522x686 的 PNG 图片，我把它放到 `drawable-xxhdpi` 目录下，在三星s6上加载，占用内存2547360B，就可以用这个方法获取到。

2、给我一张图我告诉你占多大内存

每次都问 Bitmap 你到底多大啦。。感觉怪怪的，毕竟我们不能总是去问，而不去搞清楚它为嘛介么大吧。能不能给它算个命，算算它究竟多大呢？当然是可以的，很简单嘛，我们直接顺藤摸瓜，找出真凶，哦不，找出答案。

2.1 getByteCount

`getByteCount` 的源码我们刚刚已经认识了，当我们问 Bitmap 大小的时候，这孩子也是先拿到出生年月日，然后算出来的，那么问题来了，`getHeight` 就是图片的高度（单位：px），`getRowBytes` 是什么？

```
public final int getRowBytes() {
    if (mRecycled) {
        Log.w(TAG, "Called getRowBytes() on a recycle()'d bitmap! This is undefined");
    }
    return nativeRowBytes(mFinalizer.mNativeBitmap);
}
```

额，感觉太对了啊，要 JNI 了。由于在下 C++ 实在用得少，每次想起 JNI 都请想象脑门磕墙的场景，不过呢，毛爷爷说过，一切反动派都是纸老虎~与

`nativeRowBytes` 对应的函数如下：

Bitmap.cpp

```
static jint Bitmap_rowBytes(JNIEnv* env, jobject, jlong bitmapHandle) {
    SkBitmap* bitmap = reinterpret_cast<SkBitmap*>(bitmapHandle)
    return static_cast<jint>(bitmap->rowBytes());
}
```

等等，我们好像发现了什么，原来 `Bitmap` 本质上就是一个 `SkBitmap`。。而这个 `SkBitmap` 也是大有来头，不信你瞧：[Skia](#)。啥也别说了，赶紧瞅瞅 `SkBitmap`。

SkBitmap.h

```
/** Return the number of bytes between subsequent rows of the bitmap. */
size_t rowBytes() const { return fRowBytes; }
```

SkBitmap.cpp

```
size_t SkBitmap::ComputeRowBytes(Config c, int width) {
    return SkColorTypeMinRowBytes(SkBitmapConfigToColorType(c), width);
}
```

SkImageInfo.h

```
static int SkColorTypeBytesPerPixel(SkColorType ct) {
    static const uint8_t gSize[] = {
        0, // Unknown
        1, // Alpha_8
        2, // RGB_565
        2, // ARGB_4444
        4, // RGBA_8888
        4, // BGRA_8888
        1, // kIndex_8
    };
    SK_COMPILE_ASSERT(SK_ARRAY_COUNT(gSize) == (size_t)(kLastEnum_SkColorType + 1),
                      size_mismatch_with_SkColorType_enum);

    SkASSERT((size_t)ct < SK_ARRAY_COUNT(gSize));
    return gSize[ct];
}

static inline size_t SkColorTypeMinRowBytes(SkColorType ct, int width) {
    return width * SkColorTypeBytesPerPixel(ct);
}
```

好，跟踪到这里，我们发现 ARGB_8888 也就是我们最常用的 Bitmap 的格式) 的一个像素占用 4byte，那么 rowBytes 实际上就是 $4 \times \text{width bytes}$ 。

那么结论出来了，一张 ARGB_8888 的 Bitmap 占用内存的计算公式 $\text{bitmapInRam} = \text{bitmapWidth} \times \text{bitmapHeight} \times 4 \text{ bytes}$

说到这儿你以为故事就结束了么？有本事你拿去试，算出来的和你获取到的总是会差个倍数，为啥呢？

还记得我们最开始给出的那个例子么？

一张522*686的 PNG 图片，我把它放到`drawable-xxhdpi`目录下，在三星s6上加载，占用内存2547360B，就可以用这个方法获取到。

然而公式计算出来的可是1432368B。。。

2.2 Density

知道我为什么在举例的时候那么费劲的说放到xxx目录下，还要说用xxx手机么？你以为 Bitmap 加载只跟宽高有关么？Naive。

还是先看代码，我们读取的是 `drawable` 目录下面的图片，用的是 `decodeResource` 方法，该方法本质上就两步：

- 读取原始资源，这个调用了 `Resource.openRawResource` 方法，这个方法调用完成之后会对 `TypedValue` 进行赋值，其中包含了原始资源的 `density` 等信息；
- 调用 `decodeResourceStream` 对原始资源进行解码和适配。这个过程实际上就是原始资源的 `density` 到屏幕 `density` 的一个映射。

原始资源的 `density` 其实取决于资源存放的目录（比如 `xxhdpi` 对应的是480），而屏幕 `density` 的赋值，请看下面这段代码：

BitmapFactory.java

```
public static Bitmap decodeResourceStream(Resources res, TypedValue value,
    InputStream is, Rect pad, Options opts) {

    //实际上，我们这里的opts是null的，所以在这里初始化。
    if (opts == null) {
        opts = new Options();
    }

    if (opts.inDensity == 0 && value != null) {
        final int density = value.density;
        if (density == TypedValue.DENSITY_DEFAULT) {
            opts.inDensity = DisplayMetrics.DENSITY_DEFAULT;
        } else if (density != TypedValue.DENSITY_NONE) {
            opts.inDensity = density; //这里density的值如果对应资源目录为hdpi的话，就是240
        }
    }

    if (opts.inTargetDensity == 0 && res != null) {
        //请注意，inTargetDensity就是当前的显示密度，比如三星s6时就是640
        opts.inTargetDensity = res.getDisplayMetrics().densityDpi;
    }

    return decodeStream(is, pad, opts);
}
```

我们看到 `opts` 这个值被初始化，而它的构造居然如此简单：

```

public Options() {
    inDither = false;
    inScaled = true;
    inPremultiplied = true;
}

```

所以我们就很容易的看到，`Option.inScreenDensity` 这个值没有被初始化，而实际上后面我们也会看到这个值根本不会用到；我们最应该关心的是什么呢？

是 `inDensity` 和 `inTargetDensity`，这两个值与下面 `cpp` 文件里面的 `density` 和 `targetDensity` 相对应——重复一下，`inDensity` 就是原始资源的 `density`，`inTargetDensity` 就是屏幕的 `density`。紧接着，用到了 `nativeDecodeStream` 方法，不重要的代码直接略过，直接给出最关键的 `doDecode` 函数的代码：

BitmapFactory.cpp

```

static jobject doDecode(JNIEnv* env, SkStreamRewindable* stream, jobject padding, jobject
.....
if (env->GetBooleanField(options, gOptions_scaledFieldID)) {
    const int density = env->GetIntField(options, gOptions_densityFieldID); // 对应h
    const int targetDensity = env->GetIntField(options, gOptions_targetDensityFie
    const int screenDensity = env->GetIntField(options, gOptions_screenDensityFie
    if (density != 0 && targetDensity != 0 && density != screenDensity) {
        scale = (float) targetDensity / density;
    }
}
}

const bool willScale = scale != 1.0f;
.....
SkBitmap decodingBitmap;
if (!decoder->decode(stream, &decodingBitmap, prefColorType, decodeMode)) {
    return nullObjectReturn("decoder->decode returned false");
}
// 这里这个 decodingBitmap 就是解码出来的 bitmap，大小是图片原始的大小
int scaledWidth = decodingBitmap.width();
int scaledHeight = decodingBitmap.height();
if (willScale && decodeMode != SkImageDecoder::kDecodeBounds_Mode) {
    scaledWidth = int(scaledWidth * scale + 0.5f);
    scaledHeight = int(scaledHeight * scale + 0.5f);
}
if (willScale) {
    const float sx = scaledWidth / float(decodingBitmap.width());
    const float sy = scaledHeight / float(decodingBitmap.height());
    // TODO: avoid copying when scaled size equals decodingBitmap size
}

```

```

SkColorType colorType = colorTypeForScaledOutput(decodingBitmap.colorType());
// FIXME: If the alphaType is kUnpremul and the image has alpha, the
// colors may not be correct, since Skia does not yet support drawing
// to/from unpremultiplied bitmaps.
outputBitmap->setInfo(SkImageInfo::Make(scaledWidth, scaledHeight,
    colorType, decodingBitmap.alphaType()));
if (!outputBitmap->allocPixels(outputAllocator, NULL)) {
    return nullObjectReturn("allocation failed for scaled bitmap");
}

// If outputBitmap's pixels are newly allocated by Java, there is no need
// to erase to 0, since the pixels were initialized to 0.
if (outputAllocator != &javaAllocator) {
    outputBitmap->eraseColor(0);
}

SkPaint paint;
paint.setFilterLevel(SkPaint::kLow_FilterLevel);

SkCanvas canvas(*outputBitmap);
canvas.scale(sx, sy);
canvas.drawBitmap(decodingBitmap, 0.0f, 0.0f, &paint);
}
.....
}

```

注意到其中有个 `density` 和 `targetDensity`，前者是 `decodingBitmap` 的 `density`，这个值跟这张图片的放置的目录有关（比如 `hdpi` 是240，`xxhdpi` 是480），这部分代码我跟了一下，太长了，就不列出来了；`targetDensity` 实际上是我们加载图片的目标 `density`，这个值的来源我们已经在前面给出了，就是 `DisplayMetrics` 的 `densityDpi`，如果是三星s6那么这个数值就是640。`sx` 和 `sy` 实际上是约等于 `scale` 的，因为 `scaledWidth` 和 `scaledHeight` 是由 `width` 和 `height` 乘以 `scale` 得到的。我们看到 `canvas` 放大了 `scale` 倍，然后又把读到内存的这张 `bitmap` 画上去，相当于把这张 `bitmap` 放大了 `scale` 倍。

再来看我们的例子：

一张522686的PNG图片，我把它放到 `drawable-xxhdpi` 目录下，在三星s6上加载，占用内存2547360B，其中 `density` 对应 `xxhdpi` 为480，`targetDensity` 对应三星s6的密度为640： $522/480 \times 640 = 686/480 \times 640 * 4 = 2546432B$

2.3 精度

越来越有趣了是不是，你肯定会发现我们这么细致的计算还是跟获取到的数值

不！一！样！

为什么呢？由于结果已经非常接近，我们很自然地想到精度问题。来，再把上面这段代码中的一句拿出来看看：

```
outputBitmap->setInfo(SkImageInfo::Make(scaledWidth, scaledHeight,
                                         colorType, decodingBitmap.alphaType()));
```

我们看到最终输出的 `outputBitmap` 的大小是 `scaledWidth*scaledHeight`，我们把这个变量计算的片段拿出来给大家一看就明白了：

```
if (willScale && decodeMode != SkImageDecoder::kDecodeBounds_Mode) {
    scaledWidth = int(scaledWidth * scale + 0.5f);
    scaledHeight = int(scaledHeight * scale + 0.5f);
}
```

在我们的例子中，

$$\text{scaledWidth} = \text{int}(522.640 / 480f + 0.5) = \text{int}(696.5) = 696$$

$$\text{scaledHeight} = \text{int}(686.640 / 480f + 0.5) = \text{int}(915.16666...) = 915$$

下面就是见证奇迹的时刻：

$$915 \cdot 696 \cdot 4 = 2547360$$

有木有很兴奋！有木有很激动！！

写到这里，突然想起《STL源码剖析》一书的扉页，侯捷先生只写了一句话：

“源码之前，了无秘密”。

2.4 小结

其实，通过前面的代码跟踪，我们就不难知道，`Bitmap` 在内存当中占用的大小其实取决于：

- 色彩格式，前面我们已经提到，如果是 ARGB8888 那么就是一个像素4个字节，如果是 RGB565 那就是2个字节
- 原始文件存放的资源目录（是 hdpi 还是 xxhdpi 可不能傻傻分不清楚哈）
- 目标屏幕的密度（所以同等条件下，红米在资源方面消耗的内存肯定是要小于三星S6的）

3、想办法减少 Bitmap 内存占用

3.1 Jpg 和 Png

说到这里，肯定会有人会说，我们用 jpg 吧，jpg 格式的图片不应该比 png 小么？

这确实是个好问题，因为同样一张图片，jpg 确实比 png 会多少小一些（甚至很多），原因很简单，jpg 是一种有损压缩的图片存储格式，而 png 则是无损压缩的图片存储格式，显而易见，jpg 会比 png 小，代价也是显而易见的。

可是，这说的是文件存储范畴的事情，它们只存在于文件系统，而非内存或者显存。说得简单一点儿，我有一个极品飞车的免安装硬盘版的压缩包放在我的磁盘里面，这个游戏是不能玩的，我需要先解压，才能玩——jpg 也好，png 也好就是个压缩包的概念，而我们讨论的内存占用则是从使用角度来讨论的。

所以，jpg 格式的图片与 png 格式的图片在内存当中不应该有什么不同。

『啪！！！』

『谁这么缺德！！打人不打脸好么！』

肯定有人有意见，jpg 图片读到内存就是会小，还会给我拿出例子。当然，他说的不一定是错的。因为 jpg 的图片没有 alpha 通道！！所以读到内存的时候如果用 RGB565 的格式存到内存，这下大小只有 ARGB8888 的一半，能不小么。。。

不过，抛开 Android 这个平台不谈，从出图的角度来看的话，jpg 格式的图片大小也不一定比 png 的小，这要取决于图像信息的内容：

JPG 不适用于所含颜色很少、具有大块颜色相近的区域或亮度差异十分明显的较简单的图片。对于需要高保真的较复杂的图像，PNG 虽然能无损压缩，但图片文件较大。

如果仅仅是为了 Bitmap 读到内存中的大小而考虑的话，jpg 也好 png 也好，没有什么实质的差别；二者的差别主要体现在：

- alpha 你是否真的需要？如果需要 alpha 通道，那么没有别的选择，用 png。
- 你的图色值丰富还是单调？就像刚才提到的，如果色值丰富，那么用 jpg，如果作为按钮的背景，请用 png。
- 对安装包大小的要求是否非常严格？如果你的 app 资源很少，安装包大小问题不是很凸显，看情况选择 jpg 或者 png（不过，我想现在对资源文件没有苛求的应用会很少吧。。）
- 目标用户的 cpu 是否强劲？jpg 的图像压缩算法比 png 耗时。这方面还是要酌情选择，前几年做了一段时间 Cocos2dx，由于资源非常多，项目组要求统一使用 png，可能就是出于这方面的考虑。

嗯，跑题了，我们其实想说的是怎么减少内存占用。。这一小节只是想说，休想通过这个方法来减少内存占用。。。XD

3.2 使用 inSampleSize

有些朋友一看到这个肯定就笑了。采样嘛，我以前是学信号处理的，一看到 Sample 就抽抽。。哈哈开个玩笑，这个采样其实就跟统计学里面的采样是一样的，在保证最终效果满足要求的前提下减少样本规模，方便后续的数据采集和处理。

这个方法主要用在图片资源本身较大，或者适当地采样并不会影响视觉效果的条件下，这时候我们输出地目标可能相对较小，对图片分辨率、大小要求不是非常的严格。

举个例子

我们现在有个需求，要求将一张图片进行模糊，然后作为 ImageView 的 src 呈现给用户，而我们的原始图片大小为 1080*1920，如果我们直接拿来模糊的话，一方面模糊的过程费时费力，另一方面生成的图片又占用内存，实际上在模糊运算过程中可能会存在输入和输出并存的情况，此时内存将会有一个短暂的峰值。

这时候你一定会想到三个字母在你的脑海里挥之不去，它们就是『OOM』。

既然图片最终是要被模糊的，也看不太情况，还不如直接用一张采样后的图片，如果采样率为 2，那么读出来的图片只有原始图片的 1/4 大小，真是何乐而不为呢？？

```
BitmapFactory.Options options = new Options();
options.inSampleSize = 2;
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), resId, options);
```

3.3 使用矩阵

用到 Bitmap 的地方，总会见到 Matrix。这时候你会想到什么？

『基友』『是在下输了。。。』

其实想想，Bitmap 的像素点阵，还不就是个矩阵，真是你中有我，我中有你的交情啊。那么什么时候用矩阵呢？

大图小用用采样，小图大用用矩阵。

还是用前面模糊图片的例子，我们不是采样了么？内存是小了，可是图的尺寸也小了啊，我要用 Canvas 绘制这张图可怎么办？当然是用矩阵了：

方式一：

```
Matrix matrix = new Matrix();
matrix.preScale(2, 2, 0f, 0f);
//如果使用直接替换矩阵的话，在Nexus6 5.1.1上必须关闭硬件加速
canvas.concat(matrix);
canvas.drawBitmap(bitmap, 0, 0, paint);
```

需要注意的是，在使用搭载 5.1.1 原生系统的 Nexus6 进行测试时发现，如果使用 `canvas` 的 `setMatrix` 方法，可能会导致与矩阵相关的元素的绘制存在问题，本例当中如果使用 `setMatrix` 方法，`bitmap` 将不会出现在屏幕上。因此请尽量使用 `canvas` 的 `scale`、`rotate` 这样的方法，或者使用 `concat` 方法。

方式二：

```
Matrix matrix = new Matrix();
matrix.setScale(2, 2, 0, 0);
canvas.drawBitmap(bitmap, matrix, paint);
```

这样，绘制出来的图就是放大以后的效果了，不过占用的内存却仍然是我们采样出来的大小。

如果我要把图片放到 `ImageView` 当中呢？一样可以，请看：

```
Matrix matrix = new Matrix();
matrix.postScale(2, 2, 0, 0);
imageView.setImageMatrix(matrix);
imageView.setScaleType(ScaleType.MATRIX);
imageView.setImageBitmap(bitmap);
```

3.4 合理选择Bitmap的像素格式

其实前面我们已经多次提到这个问题。ARGB8888格式的图片，每像素占用 4 Byte，而 RGB565则是 2 Byte。我们先看下有多少种格式可选：

| 格式 | 描述 |
|-----------|--------------------------------------|
| ALPHA_8 | 只有一个alpha通道 |
| ARGB_4444 | 这个从API 13开始不建议使用，因为质量太差 |
| ARGB_8888 | ARGB四个通道，每个通道8bit |
| RGB_565 | 每个像素占2Byte，其中红色占5bit，绿色占6bit，蓝色占5bit |

这几个当中，

ALPHA8 没必要用，因为我们随便用个颜色就可以搞定的。

ARGB4444 虽然占用内存只有 ARGB8888 的一半，不过已经被官方嫌弃，失宠了。。
『又要占省内存，又要看着爽，臣妾做不到啊T T』。

ARGB8888 是最常用的，大家应该最熟悉了。

RGB565 看到这个，我就看到了资源优化配置无处不在，这个绿色。。。 (不行了，突然好邪恶XD)，其实如果不需要 alpha 通道，特别是资源本身为 jpg 格式的情况下，用这个格式比较理想。

3.5 高能：索引位图(Indexed Bitmap)

索引位图，每个像素只占 1 Byte，不仅支持 RGB，还支持 alpha，而且看上去效果还不错！等等，请收起你的口水，Android 官方并不支持这个。是的，你没看错，官方并不支持。

```
public enum Config {
    // these native values must match up with the enum in SkBitmap.h

    ALPHA_8      (2),
    RGB_565      (4),
    ARGB_4444    (5),
    ARGB_8888    (6);

    final int nativeInt;
}
```

不过，Skia 引擎是支持的，不信你再看：

```
enum Config {
    kNo_Config,    //!< bitmap has not been configured
    kA8_Config,   //!< 8-bits per pixel, with only alpha specified (0 is transparent

    //看这里看这里！！↓↓↓↓↓
    kIndex8_Config, //!< 8-bits per pixel, using SkColorTable to specify the colors
    kRGB_565_Config, //!< 16-bits per pixel, (see SkColorPriv.h for packing)
    kARGB_4444_Config, //!< 16-bits per pixel, (see SkColorPriv.h for packing)
    kARGB_8888_Config, //!< 32-bits per pixel, (see SkColorPriv.h for packing)
    kRLE_Index8_Config,

    kConfigCount
};
```

其实 Java 层的枚举变量的 nativeInt 对应的就是 Skia 库当中枚举的索引值，所以，如果我们能够拿到这个索引是不是就可以了？对不起，拿不到。

不行了，废话这么多，肯定要挨板砖了 T T。

不过呢，在 png 的解码库里面有这么一段代码：

```

bool SkPNGImageDecoder::getBitmapColorType(png_structp png_ptr, png_infop info_ptr,
                                             SkColorType* colorTypep,
                                             bool* hasAlphap,
                                             SkPMColor* SK_RESTRICT theTranspColorp) {
    png_uint_32 origWidth, origHeight;
    int bitDepth, colorType;
    png_get_IHDR(png_ptr, info_ptr, &origWidth, &origHeight, &bitDepth,
                 &colorType, int_p_NULL, int_p_NULL, int_p_NULL);

    #ifdef PNG_sBIT_SUPPORTED
        // check for sBIT chunk data, in case we should disable dithering because
        // our data is not truly 8bits per component
        png_color_8p sig_bit;
        if (this->getDitherImage() && png_get_sBIT(png_ptr, info_ptr, &sig_bit)) {
            #if 0
                SkDebugf("----- sBIT %d %d %d\n",
                         sig_bit->red, sig_bit->green,
                         sig_bit->blue, sig_bit->alpha);
            #endif
            // 0 seems to indicate no information available
            if (pos_le(sig_bit->red, SK_R16_BITS) &&
                pos_le(sig_bit->green, SK_G16_BITS) &&
                pos_le(sig_bit->blue, SK_B16_BITS)) {
                this->setDitherImage(false);
            }
        }
    #endif

    if (colorType == PNG_COLOR_TYPE_PALETTE) {
        bool paletteHasAlpha = hasTransparencyInPalette(png_ptr, info_ptr);
        *colorTypep = this->getPrefColorType(kIndex_SrcDepth, paletteHasAlpha);
        // now see if we can upscale to their requested colortype
        // 这段代码，如果返回false，那么colorType就被置为索引了，那么我们看看如何返回false
        if (!canUpscalePaletteToConfig(*colorTypep, paletteHasAlpha)) {
            *colorTypep = kIndex_8_SkColorType;
        }
    } else {
        .....
    }
    return true;
}

```

`canUpscalePaletteToConfig` 函数如果返回`false`, 那么 `colorType` 就被置为 `kIndex_8_SkColorType` 了。

```
static bool canUpscalePaletteToConfig(SkColorType dstColorType, bool srcHasAlpha) {
    switch (dstColorType) {
        case kN32_SkColorType:
        case kARGB_4444_SkColorType:
            return true;
        case kRGB_565_SkColorType:
            // only return true if the src is opaque (since 565 is opaque)
            return !srcHasAlpha;
        default:
            return false;
    }
}
```

如果传入的 `dstColorType` 是 `kRGB_565_SkColorType`, 同时图片还有 `alpha` 通道, 那么返回 `false` ~~咳咳, 那么问题来了, 这个 `dstColorType` 是哪儿来的? ? 就是我们 在 `decode` 的时候, 传入的 `options` 的 `inPreferredConfig`。

下面是实验时间~

准备: 在 `assets` 目录当中放了一个叫 `index.png` 的文件, 大小`192*192`, 这个文件是通过 PhotoShop 编辑之后生成的索引格式的图片。

代码:

```
try {
    Options options = new Options();
    options.inPreferredConfig = Config.RGB_565;
    Bitmap bitmap = BitmapFactory.decodeStream(getResources().getAssets().open("index.png"));
    Log.d(TAG, "bitmap.getConfig() = " + bitmap.getConfig());
    Log.d(TAG, "scaled bitmap.getByteCount() = " + bitmap.getByteCount());
    imageView.setImageBitmap(bitmap);
} catch (IOException e) {
    e.printStackTrace();
}
```

程序运行在 Nexus6上, 由于从 `assets` 中读取不涉及前面讨论到的 `scale` 的问题, 所以这张图片读到内存以后的大小理论值 (ARGB8888) :

192 192 4=147456

好, 运行我们的代码, 看输出的 `Config` 和 `ByteCount`:

```
D/MainActivity: bitmap.getConfig() = null  
D/MainActivity: scaled bitmap.getByteCount() = 36864
```

先说大小为什么只有 36864，我们知道如果前面的讨论是没有问题的话，那么这次解码出来的 Bitmap 应该是索引格式，那么占用的内存只有 ARGB 8888 的 1/4 是意料之中的；再说 Config 为什么为 null。。额。。。黑户。。。官方说：

```
public final Bitmap.Config getConfig ()  
Added in API level 1  
If the bitmap's internal config is in one of the public formats, return that config,  
otherwise return null.
```

再说一遍，黑户。。。XD。

看来这个法子还真行啊，占用内存一下小很多。不过由于官方并未做出支持，因此这个方法有诸多限制，比如不能在 xml 中直接配置，，生成的 Bitmap 不能用于构建 Canvas 等等。

3.6 不要辜负。。。『哦，不要姑父！』

其实我们一直在抱怨资源大，有时候有些场景其实不需要图片也能完成的。比如在开发中我们会经常遇到 Loading，这些 Loading 通常就是几帧图片，图片也比较简单，只需要黑白灰加 alpha 就齐了。

『排期太紧了，这些给我出一系列图吧』

『好，不过每张图都是 300*300 的 png 哈，总共 5 张，为了适配不同的分辨率，需要出 xxhdpi 和 xxxhdpi 的两套图。。。』

Orz。。。

如果是这样，你还是自定义一个 View，覆写 onDraw 自己画一下好了。。。

4、结语

写了这么多，我们来稍稍理一理，本文主要讨论了如何运行时获取 Bitmap 占用内存的大小，如果事先根据 Bitmap 的格式、读取方式等算出其占用内存的大小，后面又整理了一些常见的 Bitmap 使用建议。突然好像说，是时候研究一下 Skia 引擎了。

怎么办，看来扔了好几年的 C++ 还是要捡回来么。。嘆。。。

评论：

文章不错，不过讲的有点多，Bitmap占用内存的大小，其实可以直接根据Bitmap的宽高计算得出，这里面讲的C++这些的，其实只是来计算最终生成的Bitmap的大小。Bitmap的Config.ARGB_8888 说的很清楚，Each pixel is stored on 4 bytes,所以其实可以直接拿Bitmap的宽x高x4就得出了。而不是原始图片的宽高。

Android性能优化之Bitmap的内存优化

来源:<http://blog.csdn.net/u010687392/article/details/50721437>

[TOC]

1、BitmapFactory解析Bitmap的原理

BitmapFactory提供的解析Bitmap的静态工厂方法有以下五种：

```
Bitmap decodeFile(...)  
Bitmap decodeResource(...)  
Bitmap decodeByteArray(...)  
Bitmap decodeStream(...)  
Bitmap decodeFileDescriptor(...)
```

其中常用的三个：`decodeFile`、`decodeResource`、`decodeStream`。

`decodeFile` 和 `decodeResource` 其实最终都是调用 `decodeStream` 方法来解析 Bitmap, `decodeStream` 的内部则是调用两个native方法解析Bitmap的：

```
nativeDecodeAsset()  
nativeDecodeStream()
```

这两个native方法只是对应 `decodeFile` 和 `decodeResource`、`decodeStream` 来解析的，像 `decodeByteArray`、`decodeFileDescriptor` 也有专门的native方法负责解析Bitmap。

接下来就是看看这两个方法在解析Bitmap时究竟有什么区

别 `decodeFile`、`decodeResource`，查看后发现它们调用路径如下：

```
decodeFile->decodeStream  
decodeResource->decodeResourceStream->decodeStream
```

`decodeResource` 在解析时多调用了一个 `decodeResourceStream` 方法，而这个 `decodeResourceStream` 方法代码如下：

```

public static Bitmap decodeResourceStream(Resources res, TypedValue value,
    InputStream is, Rect pad, Options opts) {

    if (opts == null) {
        opts = new Options();
    }

    if (opts.inDensity == 0 && value != null) {
        final int density = value.density;
        if (density == TypedValue.DENSITY_DEFAULT) {
            opts.inDensity = DisplayMetrics.DENSITY_DEFAULT;
        } else if (density != TypedValue.DENSITY_NONE) {
            opts.inDensity = density;
        }
    }

    if (opts.inTargetDensity == 0 && res != null) {
        opts.inTargetDensity = res.getDisplayMetrics().densityDpi;
    }

    return decodeStream(is, pad, opts);
}

```

它主要是对**Options**进行处理了，在得到 `opts.inDensity` 属性的前提下，如果我们没有对该属性设定值，那么将`opts.inDensity = DisplayMetrics.DENSITY_DEFAULT;`赋定这个默认的Density值，这个默认值为160，为标准的dpi比例，即在**Density=160**的设备上**1dp=1px**，这个方法中还有这么一行

```
opts.inTargetDensity = res.getDisplayMetrics().densityDpi;
```

对 `opts.inTargetDensity` 进行了赋值，该值为当前设备的 `densityDpi` 值，所以说在 `decodeResourceStream` 方法中主要做了两件事：

- 1、对`opts.inDensity`赋值，没有则赋默认值160
- 2、对`opts.inTargetDensity`赋值，没有则赋当前设备的`densityDpi`值

之后重点来了，之后参数将传入`decodeStream`方法，该方法中在调用native方法进行解析Bitmap后会调用这个方法 `setDensityFromOptions(bm, opts);`：

```

private static void setDensityFromOptions(Bitmap outputBitmap, Options opts) {
    if (outputBitmap == null || opts == null) return;

    final int density = opts.inDensity;
    if (density != 0) {
        outputBitmap.setDensity(density);
        final int targetDensity = opts.inTargetDensity;
        if (targetDensity == 0 || density == targetDensity || density == opts.inScaled)
            return;
    }

    byte[] np = outputBitmap.getNinePatchChunk();
    final boolean isNinePatch = np != null && NinePatch.isNinePatchChunk(np);
    if (opts.inScaled || isNinePatch) {
        outputBitmap.setDensity(targetDensity);
    }
} else if (opts.inBitmap != null) {
    // bitmap was reused, ensure density is reset
    outputBitmap.setDensity(Bitmap.getDefaultDensity());
}
}
}

```

该方法主要就是把刚刚赋值过的两个属性`inDensity`和`inTargetDensity`给`Bitmap`进行赋值，不过并不是直接赋给`Bitmap`就完了，中间有个判断，当`inDensity`的值与`inTargetDensity`或与设备的屏幕`Density`不相等时，则将应用`inTargetDensity`的值，如果相等则应用`inDensity`的值。

所以总结来说，`setDensityFromOptions` 方法就是把 `inTargetDensity` 的值赋给`Bitmap`，不过前提是 `opts.inScaled = true`；

进过上面的分析，可以得出这样一个结论：

在不配置`Options`的情况下：

- 1、`decodeFile`、`decodeStream`在解析时不会对`Bitmap`进行一系列的屏幕适配，解析出来的将是原始大小的图
- 2、`decodeResource`在解析时会对`Bitmap`根据当前设备屏幕像素密度`densityDpi`的值进行缩放适配操作，使得解析出来的`Bitmap`与当前设备的分辨率匹配，达到一个最佳的显示效果，并且`Bitmap`的大小将比原始的大

1.1、关于Density、分辨率、-hdpi等res目录之间的关系

| Density Dpi | 分辨率 | res | Density |
|-------------|------------|---------|---------|
| 160dpi | 320x533 | mdpi | 1 |
| 240dpi | 480x800 | hdpi | 1.5 |
| 320dpi | 720x1280 x | hdpi | 2 |
| 480dpi | 1080x1920 | xxhdpi | 3 |
| 560dpi | 1440x2560 | xxxhdpi | 3.5 |

dp与px的换算公式为：

```
px = dp * Density
```

1.2、DisplayMetrics::densityDpi与density的区别

```
getResources().getDisplayMetrics().densityDpi—表示屏幕的像素密度  
getResources().getDisplayMetrics().density—1dp等于多少个像素(px)
```

举个栗子：在屏幕密度为160的设备下，1dp=1px。在屏幕密度为320的设备下，1dp=2px。

所以这就为什么在安卓中布局建议使用dp为单位，因为可以根据当前设备的屏幕密度动态的调整进行适配

2、Bitmap的优化策略

2.1、BitmapFactory.Options的属性解析

BitmapFactory.Options中有以下属性：

inBitmap—在解析Bitmap时重用该Bitmap，不过必须等大的Bitmap而且inMutable须为true
 inMutable—配置Bitmap是否可以更改，比如：在Bitmap上隔几个像素加一条线段
 inJustDecodeBounds—为true仅返回Bitmap的宽高等属性
 inSampleSize—须 $>=1$, 表示Bitmap的压缩比例，如：inSampleSize=4, 将返回一个原始图的1/16大小的Bitmap
 inPreferredConfig—Bitmap.Config.ARGB_8888等
 inDither—是否抖动，默認為false
 inPremultiplied—默認為true，一般不改变它的值
 inDensity—Bitmap的像素密度
 inTargetDensity—Bitmap最终的像素密度
 inScreenDensity—当前屏幕的像素密度
 inScaled—是否支持缩放，默認為true，当设置了这个，Bitmap将会以inTargetDensity的值进行缩放
 inPurgeable—当存储Pixel的内存空间在系统内存不足时是否可以被回收
 inInputShareable—inPurgeable为true情况下才生效，是否可以共享一个InputStream
 inPreferQualityOverSpeed—为true则优先保证Bitmap质量其次是解码速度
 outWidth—返回的Bitmap的宽
 outHeight—返回的Bitmap的高
 inTempStorage—解码时的临时空间，建议16*1024

2.2、优化策略

- 1、BitmapConfig的配置
- 2、使用decodeFile、decodeResource、decodeStream进行解析Bitmap时，配置inDensity和inTargetDensity
- 3、使用inJustDecodeBounds预判断Bitmap的大小及使用inSampleSize进行压缩
- 4、对Density>240的设备进行Bitmap的适配（缩放Density）
- 5、2.3版本inNativeAlloc的使用
- 6、4.4以下版本inPurgeable、inInputShareable的使用
- 7、Bitmap的回收

针对上面方案，把Bitmap解码的代码封装成了一个工具类，如下：

```

public class BitmapDecodeUtil {
    private static final int DEFAULT_DENSITY = 240;
    private static final float SCALE_FACTOR = 0.75f;
    private static final Bitmap.Config DEFAULT_BITMAP_CONFIG = Bitmap.Config.RGB_565;

    private static BitmapFactory.Options getBitmapOptions(Context context) {
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inScaled = true;
        options.inPreferredConfig = DEFAULT_BITMAP_CONFIG;
        options.inPurgeable = true;
        options.inInputShareable = true;
        options.inJustDecodeBounds = false;
        if (Build.VERSION.SDK_INT <= Build.VERSION_CODES.GINGERBREAD_MR1) {
            Field field = null;
            try {

```

```
        field = BitmapFactory.Options.class.getDeclaredField("inNativeAlloc")
        field.setAccessible(true);
        field.setBoolean(options, true);
    } catch (NoSuchFieldException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}

int displayDensityDpi = context.getResources().getDisplayMetrics().densityDpi
float displayDensity = context.getResources().getDisplayMetrics().density;
if (displayDensityDpi > DEFAULT_DENSITY && displayDensity > 1.5f) {
    int density = (int) (displayDensityDpi * SCALE_FACTOR);
    options.inDensity = density;
    options.inTargetDensity = density;
}
return options;
}

public static Bitmap decodeBitmap(Context context, int resId) {
    checkParam(context);
    return BitmapFactory.decodeResource(context.getResources(), resId, getBitmapOptions(context));
}

public static Bitmap decodeBitmap(Context context, String pathName) {
    checkParam(context);
    return BitmapFactory.decodeFile(pathName, getBitmapOptions(context));
}

public static Bitmap decodeBitmap(Context context, InputStream is) {
    checkParam(context);
    checkParam(is);
    return BitmapFactory.decodeStream(is, null, getBitmapOptions(context));
}

public static Bitmap compressBitmap(Context context,int resId, int maxWidth, int maxHeight) {
    checkParam(context);
    final TypedValue value = new TypedValue();
    InputStream is = null;
    try {
        is = context.getResources().openRawResource(resId, value);
        return compressBitmap(context, is, maxWidth, maxHeight);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (is != null) {
            try {
                is.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

    }

    return null;
}

public static Bitmap compressBitmap(Context context, String pathName, int maxWidth,
checkParam(context);
InputStream is = null;
try {
    is = new FileInputStream(pathName);
    return compressBitmap(context, is, maxWidth, maxHeight);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (is != null) {
        try {
            is.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
return null;
}

public static Bitmap compressBitmap(Context context, InputStream is, int maxWidth,
checkParam(context);
checkParam(is);
BitmapFactory.Options opt = new BitmapFactory.Options();
opt.inJustDecodeBounds = true;
BitmapFactory.decodeStream(is, null, opt);
int height = opt.outHeight;
int width = opt.outWidth;
int sampleSize = computeSampleSize(width, height, maxWidth, maxHeight);
BitmapFactory.Options options = getBitmapOptions(context);
options.inSampleSize = sampleSize;
return BitmapFactory.decodeStream(is, null, options);
}

private static int computeSampleSize(int width, int height, int maxWidth, int maxHeight) {
    int inSampleSize = 1;
    if (height > maxHeight || width > maxWidth) {
        final int heightRate = Math.round((float) height / (float) maxHeight);
        final int widthRate = Math.round((float) width / (float) maxWidth);
        inSampleSize = heightRate < widthRate ? heightRate : widthRate;
    }
    if (inSampleSize % 2 != 0) {
        inSampleSize -= 1;
    }
    return inSampleSize <= 1 ? 1 : inSampleSize;
}

private static <T> void checkParam(T param){
    if(param == null)
}

```

```
        throw new NullPointerException();
    }
}
```

主要有两类方法：

- 一、decodeBitmap:对Bitmap不压缩，但是会根据屏幕的密度合适的进行缩放压缩
- 二、compressBimtap:对Bitmap进行超过最大宽高的压缩，同时也会根据屏幕的密度合适的进行缩放压缩。

3、Bitmap优化前后性能对比

针对上面方案，做一下性能对比,图片大小为3.26M,分辨率为2048*2048

有两台设备：

3.1、density为320的设备

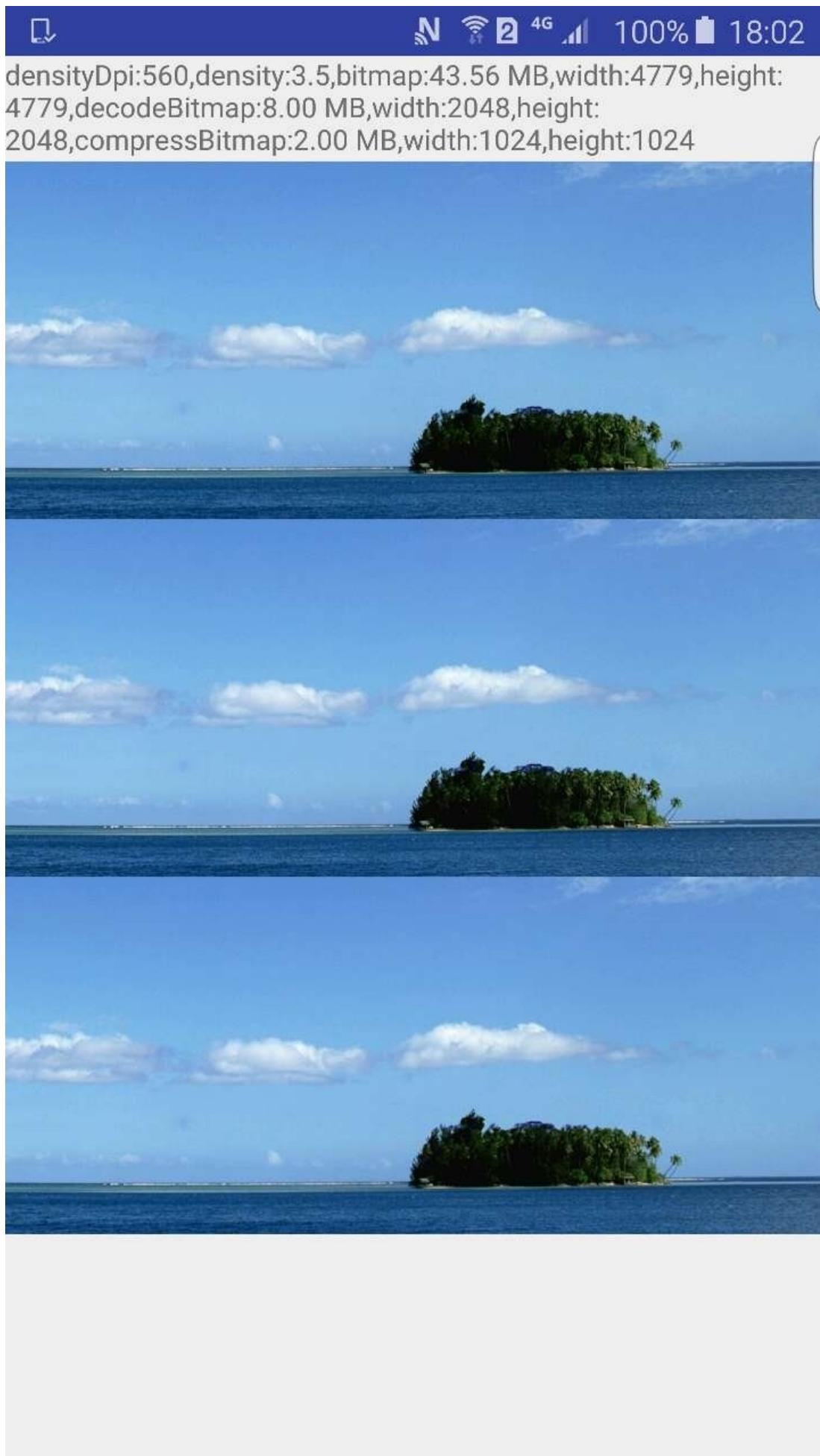
仅限紧急呼叫

18:01

densityDpi:320,density:2.0,bitmap:14.23MB,width:2731,height:2731,decodeBitmap:8.00MB,width:2048,height:2048,compressBitmap:2.00MB,width:1024,height:1024



3.2、density为560的设备



可以看到，都是加载同一图片，在高屏幕像素密度的设备下所需要的内存需要很大、载入内存中的Bitmap的宽高也因设备的屏幕像素密度也改变，正如上面分析的一样，使用decodeResource会自动适配当前设备的分辨率达到一个最佳效果，而只有这个方法会自动适配其它方法将不会，依次思路，我们在封装的工具类中在每一个方法都加入了依屏幕像素密度来自动适配，而在实际中并不需要那么高清的图片，所以我们可以根据设备的density来进行缩放，比如：在 $400 \geq \text{density} > 240$ 的情况下 $\times 0.8$, 在 $\text{density} > 400$ 的情况下 $\times 0.7$ ，这样Bitmap所占用的内存将减少非常多，可以对面上面两个图片中bitmap和decodeBitmap两个值的大小，decodeBitmap只是对density进行了一定的缩放，而占用内存却减少非常多，而且显示效果也和原先的并无区别。之后对比我们进行了inSampleSize压缩的图片，进行压缩后的效果也看不出太大区别，而占用内存也减少了很多。

4、Bitmap的回收

4.1、Android 2.3.3(API 10)及以下的系统

在2.3以下的系统中，Bitmap的像素数据是存储在native中，Bitmap对象是存储在java堆中的，所以在回收Bitmap时，需要回收两个部分的空间：native和java堆。即先调用recycle()释放native中Bitmap的像素数据，再对Bitmap对象置null，保证GC对Bitmap对象的回收

4.2、Android 3.0(API 11)及以上的系统

在3.0以上的系统中，Bitmap的像素数据和对象本身都是存储在java堆中的，无需主动调用recycle()，只需将对象置null，由GC自动管理

Android 高清加载巨图方案 拒绝压缩图片

来源：[鸿洋_](#)

一、概述

距离上一篇博客有段时间没更新了，主要是最近有些私事导致的，那么就先来一篇简单一点的博客脉动回来。

对于加载图片，大家都不陌生，一般为了尽可能避免OOM都会按照如下做法：

- 对于图片显示：根据需要显示图片控件的大小对图片进行压缩显示。
- 如果图片数量非常多：则会使用LruCache等缓存机制，将所有图片占据的内容维持在一个范围内。

其实对于图片加载还有种情况，就是单个图片非常巨大，并且还不允许压缩。比如显示：世界地图、清明上河图、微博长图等。

那么对于这种需求，该如何做呢？

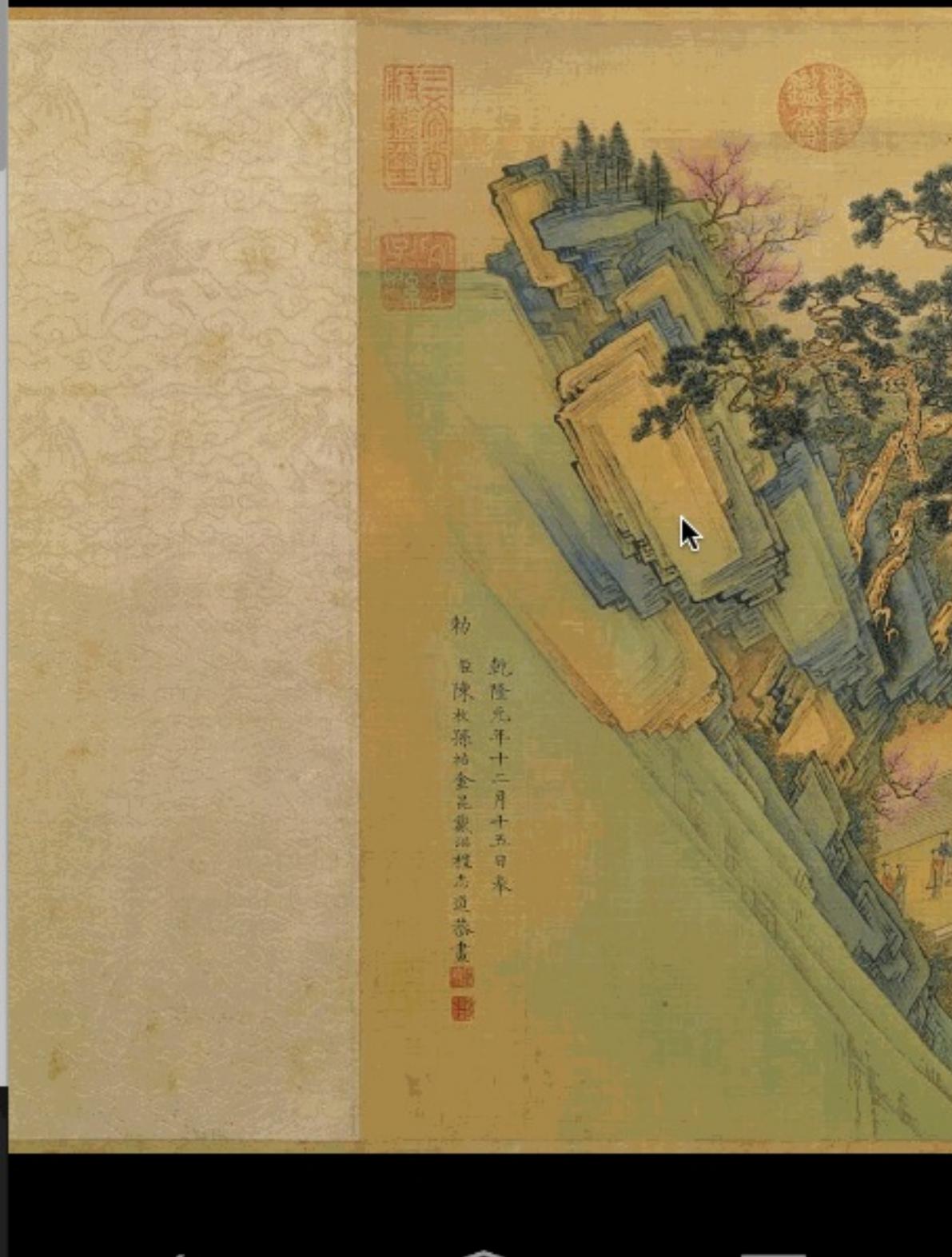
首先不压缩，按照原图尺寸加载，那么屏幕肯定是不够大的，并且考虑到内存的情况，不可能一次性整图加载到内存中，所以肯定是局部加载，那么就需要用到一个类：

BitmapRegionDecoder

其次，既然屏幕显示不完，那么最起码要添加一个上下左右拖动的手势，让用户可以拖动查看。

那么综上，本篇博文的目的就是去自定义一个显示巨图的View，支持用户去拖动查看，大概的效果图如下：

LargeImageviewDemo



好吧，这清明上河图太长了，想要观看全图，文末下载，图片在assets目录。当然如果你的图，高度也很大，肯定也是可以上下拖动的。

二、初识BitmapRegionDecoder

`BitmapRegionDecoder` 主要用于显示图片的某一块矩形区域，如果你需要显示某个图片的指定区域，那么这个类非常合适。

对于该类的用法，非常简单，既然是显示图片的某一块区域，那么至少只需要一个方法去设置图片；一个方法传入显示的区域即可；详见：

`BitmapRegionDecoder` 提供了一系列的 `newInstance` 方法来构造对象，支持传入文件路径，文件描述符，文件的`inputstream`等。

例如：

```
BitmapRegionDecoder bitmapRegionDecoder = BitmapRegionDecoder.newInstance(inputStream);
```

上述解决了传入我们需要处理的图片，那么接下来就是显示指定的区域。

```
bitmapRegionDecoder.decodeRegion(rect, options);
```

参数一很明显是一个`rect`，参数二是 `BitmapFactory.Options`，你可以控制图片的 `inSampleSize`，`inPreferredConfig` 等。

那么下面看一个超级简单的例子：

```
package com.zhy.blogcodes.largeImage;

import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.BitmapRegionDecoder;
import android.graphics.Rect;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.widget.ImageView;

import com.zhy.blogcodes.R;

import java.io.IOException;
import java.io.InputStream;

public class LargeImageViewActivity extends AppCompatActivity{
    private ImageView mImageView;

    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_large_image_view);

        mImageView = (ImageView) findViewById(R.id.id_imageview);
        try{
            InputStream inputStream = getAssets().open("tangyan.jpg");

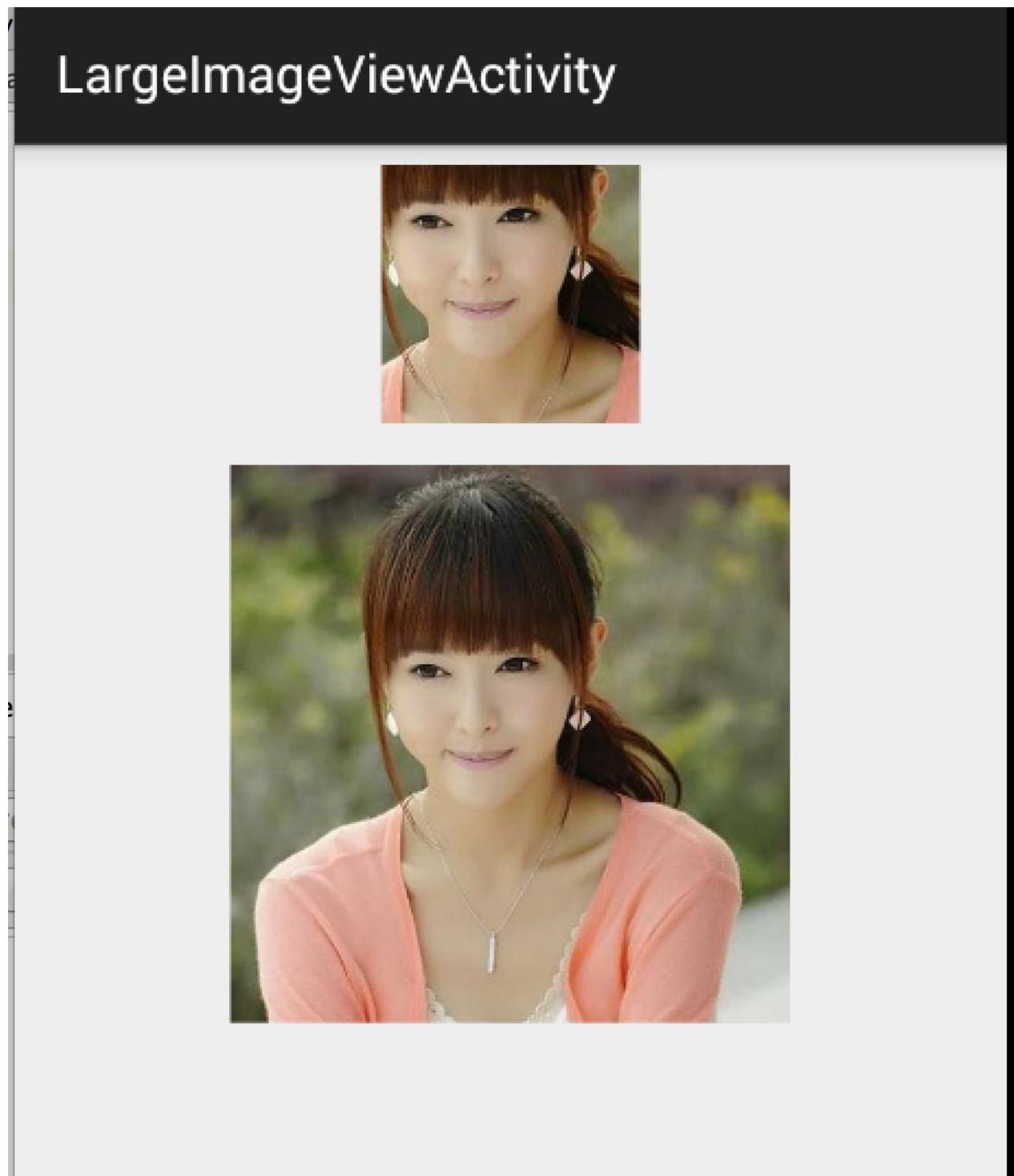
            //获得图片的宽、高
            BitmapFactory.Options tmpOptions = new BitmapFactory.Options();
            tmpOptions.inJustDecodeBounds = true;
            BitmapFactory.decodeStream(inputStream, null, tmpOptions);
            int width = tmpOptions.outWidth;
            int height = tmpOptions.outHeight;

            //设置显示图片的中心区域
            BitmapRegionDecoder bitmapRegionDecoder = BitmapRegionDecoder.newInstance
                BitmapFactory.Options options = new BitmapFactory.Options();
            options.inPreferredConfig = Bitmap.Config.RGB_565;
            Bitmap bitmap = bitmapRegionDecoder.decodeRegion(new Rect(width / 2 - 100
                mImageView.setImageBitmap(bitmap);

        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

上述代码，就是使用 `BitmapRegionDecoder` 去加载assets中的图片，调用 `bitmapRegionDecoder.decodeRegion` 解析图片的中间矩形区域，返回bitmap，最终显示在ImageView上。

效果图：



上面的小图显示的即为下面的大图的中间区域。

ok, 那么目前我们已经了解了 `BitmapRegionDecoder` 的基本用法，那么往外扩散，我们需要自定义一个控件去显示巨图就很简单了，首先Rect的范围就是我们View的大小，然后根据用户的移动手势，不断去更新我们的Rect的参数即可。

三、自定义显示大图控件

根据上面的分析呢，我们这个自定义控件思路就非常清晰了：

- 提供一个设置图片的入口
- 重写`onTouchEvent`, 在里面根据用户移动的手势，去更新显示区域的参数
- 每次更新区域参数后，调用 `invalidate`，`onDraw` 里面去 `regionDecoder.decodeRegion` 拿到 `bitmap`，去 `draw`

理清了，发现so easy，下面上代码：

```
package com.zhy.blogcodes.largeImage.view;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.BitmapRegionDecoder;
import android.graphics.Canvas;
import android.graphics.Rect;
import android.util.AttributeSet;
import android.view.MotionEvent;
import android.view.View;

import java.io.IOException;
import java.io.InputStream;

/**
 * Created by zhy on 15/5/16.
 */
public class LargeImageView extends View {
    private BitmapRegionDecoder mDecoder;
    /**
     * 图片的宽度和高度
     */
    private int mImageWidth, mImageHeight;
    /**
     * 绘制的区域
     */
    private volatile Rect mRect = new Rect();
    private MoveGestureDetector mDetector;
```

```
private static final BitmapFactory.Options options = new BitmapFactory.Options();

static {
    options.inPreferredConfig = Bitmap.Config.RGB_565;
}

public void setInputStream(InputStream is) {
    try {
        mDecoder = BitmapRegionDecoder.newInstance(is, false);
        BitmapFactory.Options tmpOptions = new BitmapFactory.Options();
        // Grab the bounds for the scene dimensions
        tmpOptions.inJustDecodeBounds = true;
        BitmapFactory.decodeStream(is, null, tmpOptions);
        mImageWidth = tmpOptions.outWidth;
        mImageHeight = tmpOptions.outHeight;

        requestLayout();
        invalidate();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {

        try {
            if (is != null) is.close();
        } catch (Exception e) {
        }
    }
}

public void init() {
    mDetector = new MoveGestureDetector(getContext(), new MoveGestureDetector.SimpleOnMoveListener() {
        @Override
        public boolean onMove(MoveGestureDetector detector) {
            int moveX = (int) detector.getMoveX();
            int moveY = (int) detector.getMoveY();

            if (mImageWidth > getWidth()) {
                mRect.offset(-moveX, 0);
                checkWidth();
                invalidate();
            }
            if (mImageHeight > getHeight()) {
                mRect.offset(0, -moveY);
                checkHeight();
                invalidate();
            }

            return true;
        }
    });
}
```

```
private void checkWidth() {  
  
    Rect rect = mRect;  
    int imageWidth = mImageWidth;  
    int imageHeight = mImageHeight;  
  
    if (rect.right > imageWidth) {  
        rect.right = imageWidth;  
        rect.left = imageWidth - getWidth();  
    }  
  
    if (rect.left < 0) {  
        rect.left = 0;  
        rect.right = getWidth();  
    }  
}  
  
private void checkHeight() {  
  
    Rect rect = mRect;  
    int imageWidth = mImageWidth;  
    int imageHeight = mImageHeight;  
  
    if (rect.bottom > imageHeight) {  
        rect.bottom = imageHeight;  
        rect.top = imageHeight - getHeight();  
    }  
  
    if (rect.top < 0) {  
        rect.top = 0;  
        rect.bottom = getHeight();  
    }  
}  
  
public LargeImageView(Context context, AttributeSet attrs) {  
    super(context, attrs);  
    init();  
}  
  
@Override  
public boolean onTouchEvent(MotionEvent event) {  
    mDetector.onTouchEvent(event);  
    return true;  
}  
  
@Override  
protected void onDraw(Canvas canvas) {
```

```

        Bitmap bm = mDecoder.decodeRegion(mRect, options);
        canvas.drawBitmap(bm, 0, 0, null);
    }

    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        super.onMeasure(widthMeasureSpec, heightMeasureSpec);

        int width = getMeasuredWidth();
        int height = getMeasuredHeight();

        int imageWidth = mImageWidth;
        int imageHeight = mImageHeight;

        //默认直接显示图片的中心区域，可以自己去调节
        mRect.left = imageWidth / 2 - width / 2;
        mRect.top = imageHeight / 2 - height / 2;
        mRect.right = mRect.left + width;
        mRect.bottom = mRect.top + height;
    }
}

```

根据上述源码：

- setInputStream里面去获得图片的真实的宽度和高度，以及初始化我们的mDecoder
- onMeasure里面为我们的显示区域的rect赋值，大小为view的尺寸
- onTouchEvent里面我们监听move的手势，在监听的回调里面去改变rect的参数，以及做边界检查，最后invalidate
- 在onDraw里面就是根据rect拿到bitmap，然后draw了

ok，上面并不复杂，不过大家有没有注意到，这个监听用户move手势的代码写的有点奇怪，恩，这里模仿了系统的 ScaleGestureDetector，编写了 MoveGestureDetector，代码如下：

- MoveGestureDetector:

```

package com.zhy.blogcodes.largeImage.view;

import android.content.Context;
import android.graphics.PointF;
import android.view.MotionEvent;

public class MoveGestureDetector extends BaseGestureDetector {

    private PointF mCurrentPointer;
    private PointF mPrePointer;
    //仅仅为了减少创建内存
}

```

```

private PointF mDeltaPointer = new PointF();

//用于记录最终结果，并返回
private PointF mExternalPointer = new PointF();

private OnMoveGestureListener mListenter;

public MoveGestureDetector(Context context, OnMoveGestureListener listener) {
    super(context);
    mListenter = listener;
}

@Override
protected void handleInProgressEvent(MotionEvent event) {
    int actionCode = event.getAction() & MotionEvent.ACTION_MASK;
    switch (actionCode) {
        case MotionEvent.ACTION_CANCEL:
        case MotionEvent.ACTION_UP:
            mListenter.onMoveEnd(this);
            resetState();
            break;
        case MotionEvent.ACTION_MOVE:
            updateStateByEvent(event);
            boolean update = mListenter.onMove(this);
            if (update) {
                mPreMotionEvent.recycle();
                mPreMotionEvent = MotionEvent.obtain(event);
            }
            break;
    }
}

@Override
protected void handleStartProgressEvent(MotionEvent event) {
    int actionCode = event.getAction() & MotionEvent.ACTION_MASK;
    switch (actionCode) {
        case MotionEvent.ACTION_DOWN:
            resetState(); //防止没有接收到CANCEL or UP，保险起见
            mPreMotionEvent = MotionEvent.obtain(event);
            updateStateByEvent(event);
            break;
        case MotionEvent.ACTION_MOVE:
            mGestureInProgress = mListenter.onMoveBegin(this);
            break;
    }
}

protected void updateStateByEvent(MotionEvent event) {
    final MotionEvent prev = mPreMotionEvent;
}

```

```
mPrePointer = caculateFocalPointer(prev);
mCurrentPointer = caculateFocalPointer(event);

//Log.e("TAG", mPrePointer.toString() + " , " + mCurrentPointer);

boolean mSkipThisMoveEvent = prev.getPointerCount() != event.getPointerCount();

//Log.e("TAG", "mSkipThisMoveEvent = " + mSkipThisMoveEvent);
mExtenalPointer.x = mSkipThisMoveEvent ? 0 : mCurrentPointer.x - mPrePointer.x;
mExtenalPointer.y = mSkipThisMoveEvent ? 0 : mCurrentPointer.y - mPrePointer.y;

}

/**
 * 根据event计算多指中心点
 *
 * @param event
 * @return
 */
private PointF caculateFocalPointer(MotionEvent event) {
    final int count = event.getPointerCount();
    float x = 0, y = 0;
    for (int i = 0; i < count; i++) {
        x += event.getX(i);
        y += event.getY(i);
    }

    x /= count;
    y /= count;

    return new PointF(x, y);
}

public float getMoveX() {
    return mExtenalPointer.x;
}

public float getMoveY() {
    return mExtenalPointer.y;
}

public interface OnMoveGestureListener {
    public boolean onMoveBegin(MoveGestureDetector detector);

    public boolean onMove(MoveGestureDetector detector);

    public void onMoveEnd(MoveGestureDetector detector);
}
```

```
public static class SimpleMoveGestureDetector implements OnMoveGestureListener {  
  
    @Override  
    public boolean onMoveBegin(MoveGestureDetector detector) {  
        return true;  
    }  
  
    @Override  
    public boolean onMove(MoveGestureDetector detector) {  
        return false;  
    }  
  
    @Override  
    public void onMoveEnd(MoveGestureDetector detector) {  
    }  
}
```

- BaseGestureDetector

```
package com.zhy.blogcodes.largeImage.view;

import android.content.Context;
import android.view.MotionEvent;

public abstract class BaseGestureDetector {

    protected boolean mGestureInProgress;

    protected MotionEvent mPreMotionEvent;
    protected MotionEvent mCurrentMotionEvent;

    protected Context mContext;

    public BaseGestureDetector(Context context) {
        mContext = context;
    }

    public boolean onTouchEvent(MotionEvent event) {

        if (!mGestureInProgress) {
            handleStartProgressEvent(event);
        } else {
            handleInProgressEvent(event);
        }

        return true;
    }

    protected abstract void handleInProgressEvent(MotionEvent event);

    protected abstract void handleStartProgressEvent(MotionEvent event);

    protected abstract void updateStateByEvent(MotionEvent event);

    protected void resetState() {
        if (mPreMotionEvent != null) {
            mPreMotionEvent.recycle();
            mPreMotionEvent = null;
        }
        if (mCurrentMotionEvent != null) {
            mCurrentMotionEvent.recycle();
            mCurrentMotionEvent = null;
        }
        mGestureInProgress = false;
    }
}
```

你可能会说，一个move手势搞这么多代码，太麻烦了。的确是的，move手势的检测非常简单，那么之所以这么写呢，主要是为了可以复用，比如现在有一堆的 `xxxGestureDetector`，当我们需要监听什么手势，就直接拿个detector来检测多方便。我相信大家肯定也郁闷过Google，为什么只有 `ScaleGestureDetector` 而没有 `RotateGestureDetector` 呢。

根据上述，大家应该理解了为什么要这么做，当时不强制，每个人都有个性。

不过值得一提的是：上面这个手势检测的写法，不是我想的，而是一个开源的项目 <https://github.com/rharder/android-gesture-detectors>，里面包含很多的手势检测。对应的博文是：<http://code.almeros.com/android-multitouch-gesture-detectors#.VibzzhArJXg>那面上面两个类就是我偷学了的~ 哈

四、测试

测试其实没撒好说的了，就是把我们的LargeImageView放入布局文件，然后Activity里面去设置inputstream了。

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.zhy.blogcodes.largeImage.view.LargeImageView
        android:id="@+id/id_largetImageview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</RelativeLayout>
```

然后在Activity里面去设置图片：

```
package com.zhy.blogcodes.largeImage;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;

import com.zhy.blogcodes.R;
import com.zhy.blogcodes.largeImage.view.LargeImageView;

import java.io.IOException;
import java.io.InputStream;

public class LargeImageViewActivity extends AppCompatActivity{
    private LargeImageView mLARGEImageView;

    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_large_image_view);

        mLARGEImageView = (LargeImageView) findViewById(R.id.id_largetImageview);
        try{
            InputStream inputStream = getAssets().open("world.jpg");
            mLARGEImageView.setInputStream(inputStream);

        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

效果图：

LargeImageActivity



ok, 那么到此，显示巨图的方案以及详细的代码就描述完成了，总体还是非常简单的。

但是，在实际的项目中，可能会有更多的需求，比如增加放大、缩小；增加快滑手势等等，那么大家可以去参考这个库：<https://github.com/johnnylambda/WorldMap>，该库基本实现了绝大多数的需求，大家根据本文这个思路再去看这个库，也会简单很多，定制起来也容易。我这个地图的图就是该库里面提供的。

哈，掌握了这个，以后面试过程中也可以悄悄的装一把了，当你优雅的答完Android加载图片的方案以后，然后接一句，其实还有一种情况，就是高清显示巨图，那么我们应该...相信面试官对你的印象会好很多~ have a nice day ~

[源码点击下载](#)

MemoryLeak

Android内存泄漏总结

来源:yq.aliyun.com

Android 内存泄漏总结

内存管理的目的就是让我们在开发中怎么有效的避免我们的应用出现内存泄漏的问题。内存泄漏大家都不陌生了，简单粗俗的讲，就是该被释放的对象没有释放，一直被某个或某些实例所持有却不再被使用导致 GC 不能回收。最近自己阅读了大量相关的文档资料，打算做个 总结 沉淀下来跟大家一起分享和学习，也给自己一个警示，以后 coding 时怎么避免这些情况，提高应用的体验和质量。

我会从 java 内存泄漏的基础知识开始，并通过具体例子来说明 Android 引起内存泄漏的各种原因，以及如何利用工具来分析应用内存泄漏，最后再做总结。

篇幅有些长，大家可以分几节来看！

Java 内存分配策略

Java 程序运行时的内存分配策略有三种，分别是静态分配、栈式分配、和堆式分配，对应的，三种存储策略使用的内存空间主要分别是静态存储区（也称方法区）、栈区和堆区。

- **静态存储区（方法区）**：主要存放静态数据、全局 static 数据和常量。这块内存在程序编译时就已经分配好，并且在程序整个运行期间都存在。
- **栈区**：当方法被执行时，方法体内的局部变量都在栈上创建，并在方法执行结束时这些局部变量所持有的内存将会自动被释放。因为栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
- **堆区**：又称动态内存分配，通常就是指在程序运行时直接 new 出来的内存。这部分内存不使用时将会由 Java 垃圾回收器来负责回收。

栈与堆的区别

在方法体内定义的（局部变量）一些基本类型的变量和对象的引用变量都是在方法的栈内存中分配的。当在一段方法块中定义一个变量时，Java 就会在栈中为该变量分配内存空间，当超过该变量的作用域后，该变量也就无效了，分配给它的内存空间也将被释放掉，该内存空间可以被重新使用。

堆内存用来存放所有由 new 创建的对象（包括该对象其中的所有成员变量）和数组。在堆中分配的内存，将由 Java 垃圾回收器来自动管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，这个特殊的变量就是我们上面说的引用变量。我们可以通过这个引用变量来访问堆中的对象或者数组。

举个例子：

```
public class Sample() {
    int s1 = 0;
    Sample mSample1 = new Sample();

    public void method() {
        int s2 = 1;
        Sample mSample2 = new Sample();
    }
}

Sample mSample3 = new Sample();
```

Sample 类的局部变量 s2 和引用变量 mSample2 都是存在于栈中，但 mSample2 指向的对象是存在于堆上的。

mSample3 指向的对象实体存放在堆上，包括这个对象的所有成员变量 s1 和 mSample1，而它自己存在于栈中。

结论

- 局部变量的基本数据类型和引用存储于栈中，引用的对象实体存储于堆中。—— 因为它们属于方法中的变量，生命周期随方法而结束。
- 成员变量全部存储与堆中（包括基本数据类型，引用和引用的对象实体）—— 因为它们属于类，类对象终究是要被new出来使用的。

了解了 Java 的内存分配之后，我们再来看看 Java 是怎么管理内存的。

Java是如何管理内存

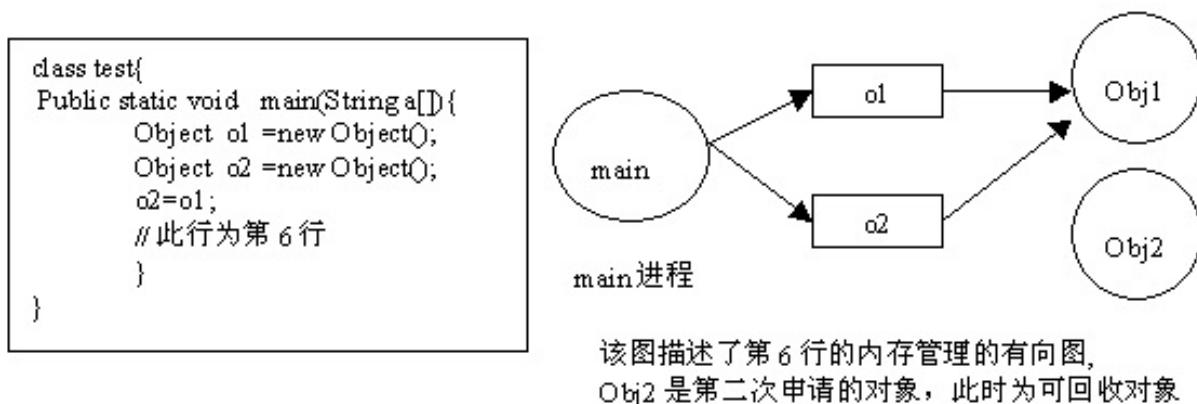
Java的内存管理就是对象的分配和释放问题。在 Java 中，程序员需要通过关键字 new 为每个对象申请内存空间 (基本类型除外)，所有的对象都在堆 (Heap)中分配空间。另外，对象的释放是由 GC 决定和执行的。在 Java 中，内存的分配是由程序完成的，而内存的释放是由 GC 完成的，这种收支两条线的方法确实简化了程序员的工作。但同时，它也加

重了JVM的工作。这也是Java程序运行速度较慢的原因之一。因为，GC为了能够正确释放对象，GC必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等，GC都需要进行监控。

监视对象状态是为了更加准确地、及时地释放对象，而释放对象的根本原则就是该对象不再被引用。

为了更好理解GC的工作原理，我们可以将对象考虑为有向图的顶点，将引用关系考虑为图的有向边，有向边从引用者指向被引对象。另外，每个线程对象可以作为一个图的起始顶点，例如大多程序从main进程开始执行，那么该图就是以main进程顶点开始的一棵根树。在这个有向图中，根顶点可达的对象都是有效对象，GC将不回收这些对象。如果某个对象(连通子图)与这个根顶点不可达(注意，该图为有向图)，那么我们认为这个(这些)对象不再被引用，可以被GC回收。

以下，我们举一个例子说明如何用有向图表示内存管理。对于程序的每一个时刻，我们都有一个有向图表示JVM的内存分配情况。以下右图，就是左边程序运行到第6行的示意图。



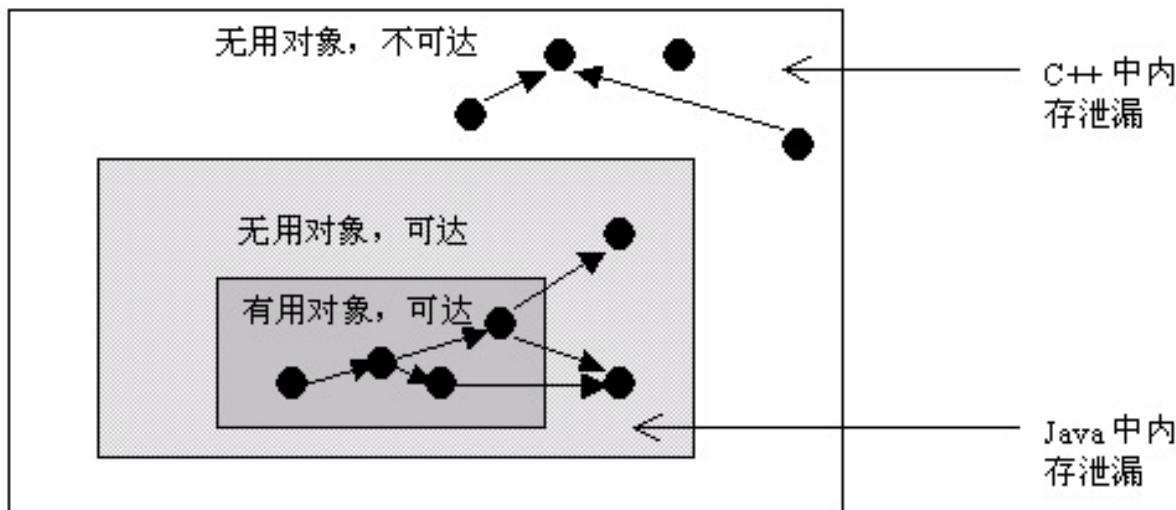
Java使用有向图的方式进行内存管理，可以消除引用循环的问题，例如有三个对象，相互引用，只要它们和根进程不可达的，那么GC也是可以回收它们的。这种方式的优点是管理内存的精度很高，但是效率较低。另外一种常用的内存管理技术是使用计数器，例如COM模型采用计数器方式管理构件，它与有向图相比，精度行低(很难处理循环引用的问题)，但执行效率很高。

什么是Java中的内存泄露

在Java中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，首先，这些对象是可达的，即在有向图中，存在通路可以与其相连；其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为Java中的内存泄漏，这些对象不会被GC所回收，然而它却占用内存。

在C++中，内存泄漏的范围更大一些。有些对象被分配了内存空间，然后却不可达，由于C++中没有GC，这些内存将永远收不回来。在Java中，这些不可达的对象都由GC负责回收，因此程序员不需要考虑这部分的内存泄露。

通过分析，我们得知，对于C++，程序员需要自己管理边和顶点，而对于Java程序员只需要管理边就可以了(不需要管理顶点的释放)。通过这种方式，Java提高了编程的效率。



因此，通过以上分析，我们知道在Java中也有内存泄漏，但范围比C++要小一些。因为Java从语言上保证，任何对象都是可达的，所有的不可达对象都由GC管理。

对于程序员来说，GC基本是透明的，不可见的。虽然，我们只有几个函数可以访问GC，例如运行GC的函数System.gc()，但是根据Java语言规范定义，该函数不保证JVM的垃圾收集器一定会执行。因为，不同的JVM实现者可能使用不同的算法管理GC。通常，GC的线程的优先级别较低。JVM调用GC的策略也有很多种，有的是内存使用到达一定程度时，GC才开始工作，也有定时执行的，有的是平缓执行GC，有的是中断式执行GC。但通常来说，我们不需要关心这些。除非在一些特定的场合，GC的执行影响应用程序的性能，例如对于基于Web的实时系统，如网络游戏等，用户不希望GC突然中断应用程序执行而进行垃圾回收，那么我们需要调整GC的参数，让GC能够通过平缓的方式释放内存，例如将垃圾回收分解为一系列的小步骤执行，Sun提供的HotSpot JVM就支持这一特性。

同样给出一个Java内存泄漏的典型例子，

```
Vector v = new Vector(10);
for (int i = 1; i < 100; i++) {
    Object o = new Object();
    v.add(o);
    o = null;
}
```

在这个例子中，我们循环申请Object对象，并将所申请的对象放入一个Vector中，如果我们仅仅释放引用本身，那么Vector仍然引用该对象，所以这个对象对GC来说是不可回收的。因此，如果对象加入到Vector后，还必须从Vector中删除，最简单的方法就是将Vector对象设置为null。

Android中常见的内存泄漏汇总

- 集合类泄漏

集合类如果仅仅有添加元素的方法，而没有相应的删除机制，导致内存被占用。如果这个集合类是全局性的变量（比如类中的静态属性，全局性的map等即有静态引用或final一直指向它），那么没有相应的删除机制，很可能导致集合所占用的内存只增不减。比如上面的典型例子就是其中一种情况，当然实际上我们在项目中肯定不会写这么2B的代码，但稍不注意还是很容易出现这种情况，比如我们都喜欢通过HashMap做一些缓存之类的事，这种情况就要多留一些心眼。

- 单例造成的内存泄漏

由于单例的静态特性使得其生命周期跟应用的生命周期一样长，所以如果使用不恰当的话，很容易造成内存泄漏。比如下面一个典型的例子，

```
public class AppManager {
    private static AppManager instance;
    private Context context;
    private AppManager(Context context) {
        this.context = context;
    }
    public static AppManager getInstance(Context context) {
        if (instance == null) {
            instance = new AppManager(context);
        }
        return instance;
    }
}
```

这是一个普通的单例模式，当创建这个单例的时候，由于需要传入一个Context，所以这个Context的生命周期的长短至关重要：

- 1、如果此时传入的是Application的Context，因为Application的生命周期就是整个应用的生命周期，所以这将没有任何问题。
- 2、如果此时传入的是Activity的Context，当这个Context所对应的Activity退出时，由于该Context的引用被单例对象所持有，其生命周期等于整个应用程序的生命

周期，所以当前 Activity 退出时它的内存并不会被回收，这就造成泄漏了。

正确的方式应该改为下面这种方式：

```
public class AppManager {  
    private static AppManager instance;  
    private Context context;  
    private AppManager(Context context) {  
        this.context = context.getApplicationContext(); // 使用Application 的context  
    }  
    public static AppManager getInstance(Context context) {  
        if (instance == null) {  
            instance = new AppManager(context);  
        }  
        return instance;  
    }  
}
```

或者这样写，连 Context 都不用传进来了：

在你的 Application 中添加一个静态方法，`getContext()` 返回 Application 的 context，

```
...  
  
context = getApplicationContext();  
  
...  
/**  
 * 获取全局的context  
 * @return 返回全局context对象  
 */  
public static Context getContext(){  
    return context;  
}  
  
public class AppManager {  
    private static AppManager instance;  
    private Context context;  
    private AppManager() {  
        this.context = MyApplication.getContext(); // 使用Application 的context  
    }  
    public static AppManager getInstance() {  
        if (instance == null) {  
            instance = new AppManager();  
        }  
        return instance;  
    }  
}
```

- 匿名内部类/非静态内部类和异步线程

1、非静态内部类创建静态实例造成的内存泄漏

有的时候我们可能会在启动频繁的Activity中，为了避免重复创建相同的数据资源，可能会出现这种写法：

```
public class MainActivity extends AppCompatActivity {
    private static TestResource mResource = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        if(mManager == null){
            mManager = new TestResource();
        }
        //...
    }
    class TestResource {
        //...
    }
}
```

这样就在Activity内部创建了一个非静态内部类的单例，每次启动Activity时都会使用该单例的数据，这样虽然避免了资源的重复创建，不过这种写法却会造成内存泄漏，因为非静态内部类默认会持有外部类的引用，而该非静态内部类又创建了一个静态的实例，该实例的生命周期和应用的一样长，这就导致了该静态实例一直会持有该Activity的引用，导致Activity的内存资源不能正常回收。正确的做法为：

将该内部类设为静态内部类或将该内部类抽取出来封装成一个单例，如果需要使用Context，请按照上面推荐的使用Application的Context。当然，Application的context不是万能的，所以也不能随便乱用，对于有些地方则必须使用Activity的Context，对于Application，Service，Activity三者的Context的应用场景如下：

| 功能 | Application | Service | Activity |
|----------------------------|-------------|---------|----------|
| Start an Activity | NO1 | NO1 | YES |
| Show a Dialog | NO | NO | YES |
| Layout Inflation | YES | YES | YES |
| Start a Service | YES | YES | YES |
| Bind to a Service | YES | YES | YES |
| Send a Broadcast | YES | YES | YES |
| Register BroadcastReceiver | YES | YES | YES |
| Load Resource Values | YES | YES | YES |

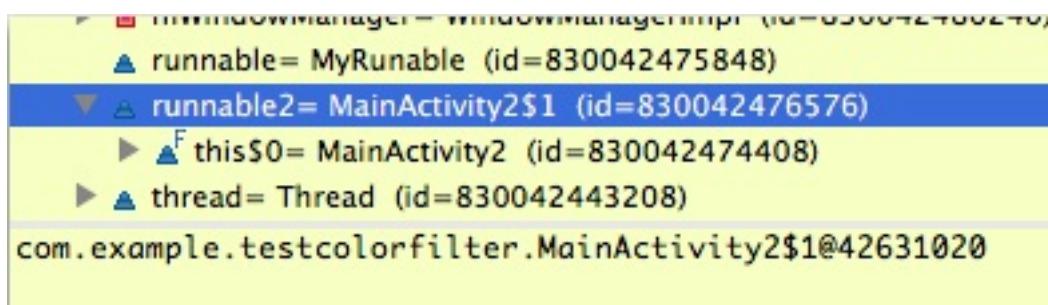
其中：NO1表示 Application 和 Service 可以启动一个 Activity，不过需要创建一个新的 task 任务队列。而对于 Dialog 而言，只有在 Activity 中才能创建

2、匿名内部类

android开发经常会继承实现Activity/Fragment/View，此时如果你使用了匿名类，并被异步线程持有了，那要小心了，如果没有任何措施这样一定会导致泄露

```
public class MainActivity extends Activity {
    ...
    Runnable ref1 = new MyRunnable();
    Runnable ref2 = new Runnable() {
        @Override
        public void run() {
            ...
        }
    };
    ...
}
```

ref1和ref2的区别是，ref2使用了匿名内部类。我们来看看运行时这两个引用的内存：



可以看到，ref1没什么特别的。

但ref2这个匿名类的实现对象里面多了一个引用：

this\$0这个引用指向MainActivity.this，也就是说当前的MainActivity实例会被ref2持有，如果将这个引用再传入一个异步线程，此线程和此Activity生命周期不一致的时候，就造成了Activity的泄露。

- Handler 造成的内存泄漏

Handler 的使用造成的内存泄漏问题应该说是最为常见了，很多时候我们为了避免 ANR 而不在主线程进行耗时操作，在处理网络任务或者封装一些请求回调等api都借助Handler 来处理，但 Handler 不是万能的，对于 Handler 的使用代码编写一不规范即有可能造成内存泄漏。另外，我们知道 Handler、Message 和 MessageQueue 都是相互关联在一起的，万一 Handler 发送的 Message 尚未被处理，则该 Message 及发送它的 Handler 对

象将被线程 MessageQueue 一直持有。由于 Handler 属于 TLS(Thread Local Storage) 变量, 生命周期和 Activity 是不一致的。因此这种实现方式一般很难保证跟 View 或者 Activity 的生命周期保持一致, 故很容易导致无法正确释放。

举个例子:

```
public class SampleActivity extends Activity {

    private final Handler mLeakyHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            // ...
        }
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Post a message and delay its execution for 10 minutes.
        mLeakyHandler.postDelayed(new Runnable() {
            @Override
            public void run() { /* ... */ }
        }, 1000 * 60 * 10);

        // Go back to the previous Activity.
        finish();
    }
}
```

在该 SampleActivity 中声明了一个延迟10分钟执行的消息 Message, mLeakyHandler 将其 push 进了消息队列 MessageQueue 里。当该 Activity 被 finish() 掉时, 延迟执行任务的 Message 还会继续存在于主线程中, 它持有该 Activity 的 Handler 引用, 所以此时 finish() 掉的 Activity 就不会被回收了从而造成内存泄漏 (因 Handler 为非静态内部类, 它会持有外部类的引用, 在这里就是指 SampleActivity) 。

修复方法: 在 Activity 中避免使用非静态内部类, 比如上面我们将 Handler 声明为静态的, 则其存活期跟 Activity 的生命周期就无关了。同时通过弱引用的方式引入 Activity, 避免直接将 Activity 作为 context 传进去, 见下面代码:

```

public class SampleActivity extends Activity {

    /**
     * Instances of static inner classes do not hold an implicit
     * reference to their outer class.
     */
    private static class MyHandler extends Handler {
        private final WeakReference<SampleActivity> mActivity;

        public MyHandler(SampleActivity activity) {
            mActivity = new WeakReference<SampleActivity>(activity);
        }

        @Override
        public void handleMessage(Message msg) {
            SampleActivity activity = mActivity.get();
            if (activity != null) {
                // ...
            }
        }
    }

    private final MyHandler mHandler = new MyHandler(this);

    /**
     * Instances of anonymous classes do not hold an implicit
     * reference to their outer class when they are "static".
     */
    private static final Runnable sRunnable = new Runnable() {
        @Override
        public void run() { /* ... */ }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Post a message and delay its execution for 10 minutes.
        mHandler.postDelayed(sRunnable, 1000 * 60 * 10);

        // Go back to the previous Activity.
        finish();
    }
}

```

综述，即推荐使用静态内部类 + WeakReference 这种方式。每次使用前注意判空。

前面提到了 WeakReference，所以这里就简单的说一下 Java 对象的几种引用类型。

Java对引用的分类有 Strong reference, SoftReference, WeakReference, PhatomReference 四种。

| 级别 | 回收时机 | 用途 | 生存时间 |
|----|--------|--|------------|
| 强 | 从来不会 | 对象的一般状态 | JVM停止运行时终止 |
| 软 | 在内存不足时 | 联合ReferenceQueue构造有效期短/占内存大/生命周期长的对象的二级高速缓冲器 (内存不足才清空) | 内存不足时终止 |
| 弱 | 在垃圾回收时 | 联合ReferenceQueue构造有效期短/占内存大/生命周期长的对象的一级高速缓冲器 (系统发生gc则清空) | gc运行后终止 |
| 虚 | 在垃圾回收时 | 联合ReferenceQueue来跟踪对象被垃圾回收器回收的活动 | gc运行后终止 |

在Android应用的开发中，为了防止内存溢出，在处理一些占用内存大而且声明周期较长的对象时候，可以尽量应用软引用和弱引用技术。

软/弱引用可以和一个引用队列 (ReferenceQueue) 联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。利用这个队列可以得知被回收的软/弱引用的对象列表，从而为缓冲器清除已失效的软/弱引用。

假设我们的应用会用到大量的默认图片，比如应用中有默认的头像，默认游戏图标等等，这些图片很多地方会用到。如果每次都去读取图片，由于读取文件需要硬件操作，速度较慢，会导致性能较低。所以我们考虑将图片缓存起来，需要的时候直接从内存中读取。但是，由于图片占用内存空间比较大，缓存很多图片需要很多的内存，就可能比较容易发生 OutOfMemory 异常。这时，我们可以考虑使用软/弱引用技术来避免这个问题发生。以下就是高速缓冲器的雏形：

首先定义一个HashMap，保存软引用对象。

```
private Map <String, SoftReference<Bitmap>> imageCache = new HashMap <String, SoftRef
```

再来定义一个方法，保存Bitmap的软引用到HashMap。

```
public class CacheBySoftRef {  
    // 首先定义一个HashMap，保存软引用对象。  
    private Map<String, SoftReference<Bitmap>> imageCache = new HashMap<String,  
    SoftReference<Bitmap>>();  
    // 再来定义一个方法，保存Bitmap的软引用到HashMap。  
    public void addBitmapToCache(String path) {  
        // 强引用的Bitmap对象  
        Bitmap bitmap = BitmapFactory.decodeFile(path);  
        // 软引用的Bitmap对象  
        SoftReference<Bitmap> softBitmap = new SoftReference<Bitmap>(bitmap);  
        // 添加该对象到Map中使其缓存  
        imageCache.put(path, softBitmap);  
    }  
    // 获取的时候，可以通过SoftReference的get()方法得到Bitmap对象。  
    public Bitmap getBitmapByPath(String path) {  
        // 从缓存中取软引用的Bitmap对象  
        SoftReference<Bitmap> softBitmap = imageCache.get(path);  
        // 判断是否存在软引用  
        if (softBitmap == null) {  
            return null;  
        }  
        // 通过软引用取出Bitmap对象，如果由于内存不足Bitmap被回收，将取得空，如果未被回收，则可重复使  
用，提高速度。  
        Bitmap bitmap = softBitmap.get();  
        return bitmap;  
    }  
}
```

使用软引用以后，在OutOfMemory异常发生之前，这些缓存的图片资源的内存空间可以被释放掉的，从而避免内存达到上限，避免Crash发生。

如果只是想避免OutOfMemory异常的发生，则可以使用软引用。如果对于应用的性能更在意，想尽快回收一些占用内存比较大的对象，则可以使用弱引用。

另外可以根据对象是否经常使用来判断选择软引用还是弱引用。如果该对象可能会经常使用的，就尽量用软引用。如果该对象不被使用的可能性更大些，就可以用弱引用。

ok，继续回到主题。前面所说的，创建一个静态Handler内部类，然后对 Handler 持有的对象使用弱引用，这样在回收时也可以回收 Handler 持有的对象，但是这样做虽然避免了 Activity 泄漏，不过 Looper 线程的消息队列中还是可能会有待处理的消息，所以我们在 Activity 的 Destroy 时或者 Stop 时应该移除消息队列 MessageQueue 中的消息。

下面几个方法都可以移除 Message：

```
public final void removeCallbacks(Runnable r);  
  
public final void removeCallbacks(Runnable r, Object token);  
  
public final void removeCallbacksAndMessages(Object token);  
  
public final void removeMessages(int what);  
  
public final void removeMessages(int what, Object object);
```

- 尽量避免使用 static 成员变量

如果成员变量被声明为 static，那我们都知道其生命周期将与整个app进程生命周期一样。

这会导致一系列问题，如果你的app进程设计上是长驻内存的，那即使app切到后台，这部分内存也不会被释放。按照现在手机app内存管理机制，占内存较大的后台进程将优先回收，因为如果此app做过进程互保保活，那会造成app在后台频繁重启。当手机安装了你参与开发的app以后一夜时间手机被消耗空了电量、流量，你的app不得不被用户卸载或者静默。

这里修复的方法是：

- a.不要在类初始时初始化静态成员。可以考虑lazy初始化。
- b.架构设计上要思考是否真的有必要这样做，尽量避免。如果架构需要这么设计，那么此对象的生命周期你有责任管理起来。

- 避免 override finalize()

1、finalize 方法被执行的时间不确定，不能依赖与它来释放紧缺的资源。时间不确定的原因是：虚拟机调用GC的时间不确定

Finalize daemon线程被调度到的时间不确定

2、`finalize` 方法只会被执行一次，即使对象被复活，如果已经执行过了 `finalize` 方法，再次被 GC 时也不会再执行了，原因是：

含有 `finalize` 方法的 object 是在 `new` 的时候由虚拟机生成了一个 `finalize reference` 在来引用到该Object的，而在 `finalize` 方法执行的时候，该 object 所对应的 `finalize Reference` 会被释放掉，即使在这个时候把该 object 复活(即用强引用引用住该 object)，再第二次被 GC 的时候由于没有了 `finalize reference` 与之对应，所以 `finalize` 方法不会再执行。

3、含有Finalize方法的object需要至少经过两轮GC才有可能被释放。

详情见[这里](#) [深入分析过dalvik的代码](#)

- 资源未关闭造成的内存泄漏

对于使用了BroadcastReceiver, ContentObserver, File, 游标 Cursor, Stream, Bitmap等资源的使用，应该在Activity销毁时及时关闭或者注销，否则这些资源将不会被回收，造成内存泄漏。

- 一些不良代码造成的内存压力

有些代码并不造成内存泄露，但是它们，或是对没使用的内存没进行有效及时的释放，或是没有有效的利用已有的对象而是频繁的申请新内存。

比如：

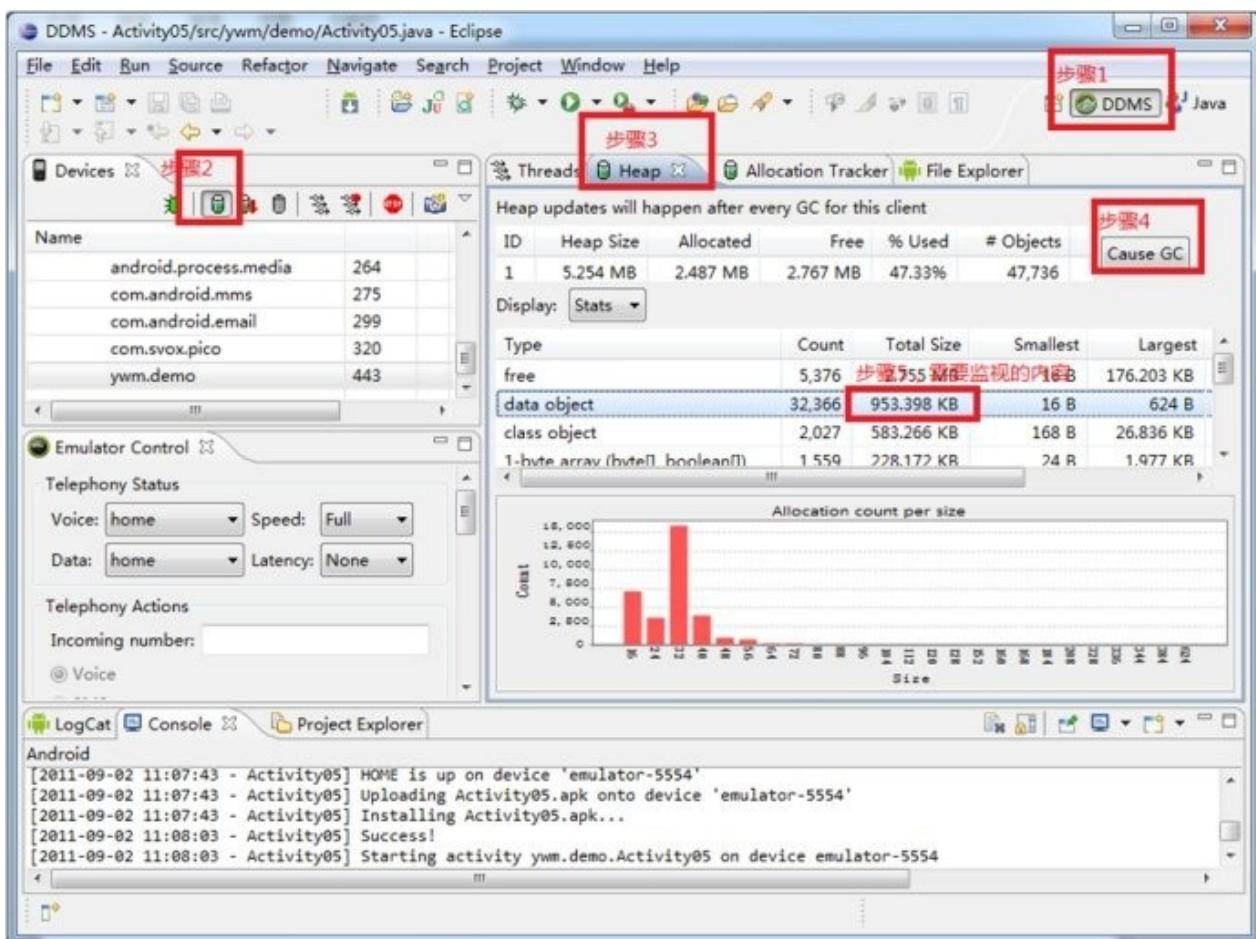
- Bitmap 没调用 `recycle()`方法，对于 Bitmap 对象在不使用时,我们应该先调用 `recycle()` 释放内存，然后才设置为 null. 因为加载 Bitmap 对象的内存空间，一部分是 java 的，一部分 C 的 (因为 Bitmap 分配的底层是通过 JNI 调用的)。而这个 `recycle()` 就是针对 C 部分的内存释放。
- 构造 Adapter 时，没有使用缓存的 `convertView` ,每次都在创建新的 `converView`。这里推荐使用 `ViewHolder`。

工具分析

Java 内存泄漏的分析工具有很多，但众所周知的要数 MAT(Memory Analysis Tools) 和 YourKit 了。由于篇幅问题，我这里就只对 [MAT](#) 的使用做一下介绍。--> [MAT 的安装](#)

- MAT分析heap的总内存占用大小来初步判断是否存在泄露

打开 DDMS 工具，在左边 Devices 视图页面选中“Update Heap”图标，然后在右边切换到 Heap 视图，点击 Heap 视图中的“Cause GC”按钮，到此为止需检测的进程就可以被监视。



Heap视图中部有一个Type叫做data object，即数据对象，也就是我们的程序中大量存在的类类型的对象。在data object一行中有一列是“Total Size”，其值就是当前进程中所有Java数据对象的内存总量，一般情况下，这个值的大小决定了是否会有内存泄漏。可以这样判断：

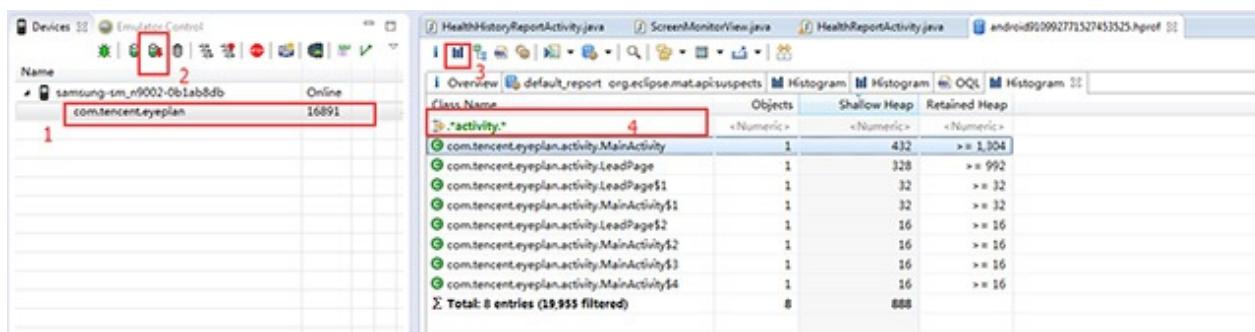
进入某应用，不断的操作该应用，同时注意观察**data object**的**Total Size**值，正常情况下**Total Size**值都会稳定在一个有限的范围内，也就是说由于程序中的代码良好，没有造成对象不被垃圾回收的情况。

所以说虽然我们不断的操作会不断的生成很多对象，而在虚拟机不断的进行GC的过程中，这些对象都被回收了，内存占用量会落到一个稳定的水平；反之如果代码中存在没有释放对象引用的情况，则**data object**的**Total Size**值在每次GC后不会有明显的回落。随着操作次数的增多**Total Size**的值会越来越大，直到到达一个上限后导致进程被杀掉。

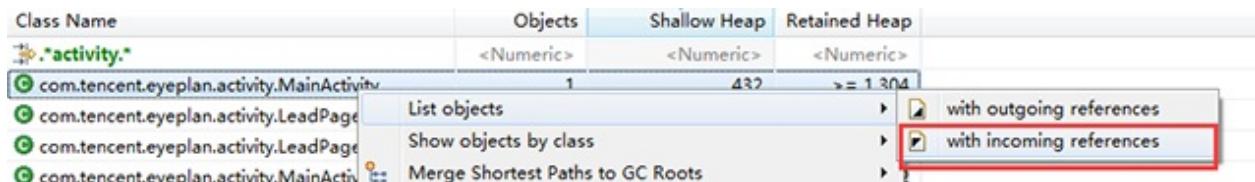
- MAT分析hprof来定位内存泄露的原因所在

这是出现内存泄露后使用MAT进行问题定位的有效手段。

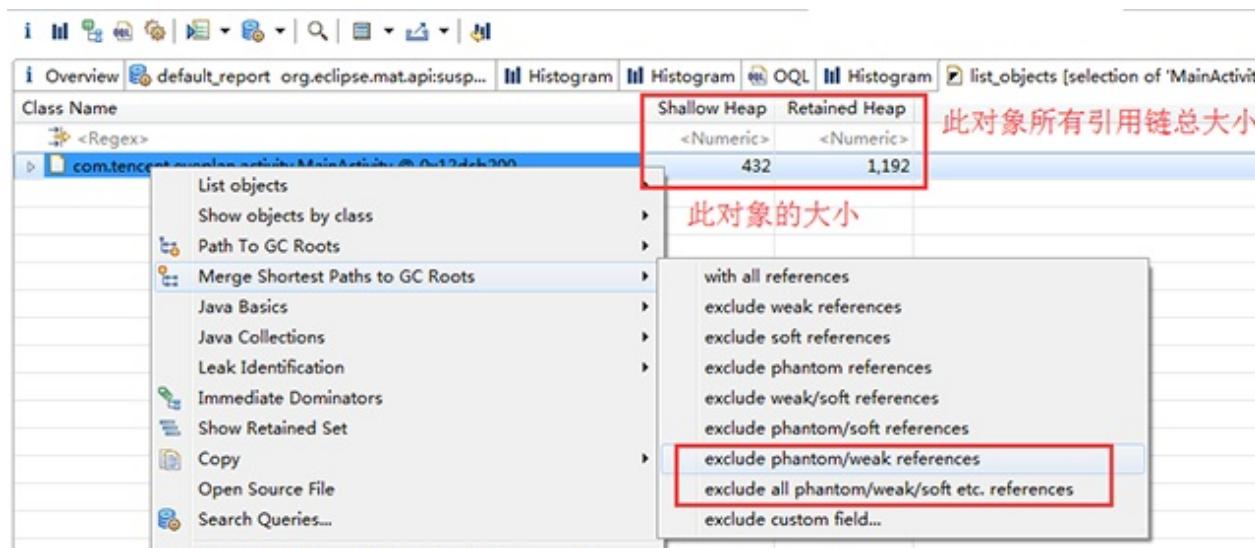
A)Dump出内存泄露当时的内存镜像hprof，分析怀疑泄露的类：



B) 分析持有此类对象引用的外部对象



C) 分析这些持有引用的对象的GC路径



D) 逐个分析每个对象的GC路径是否正常

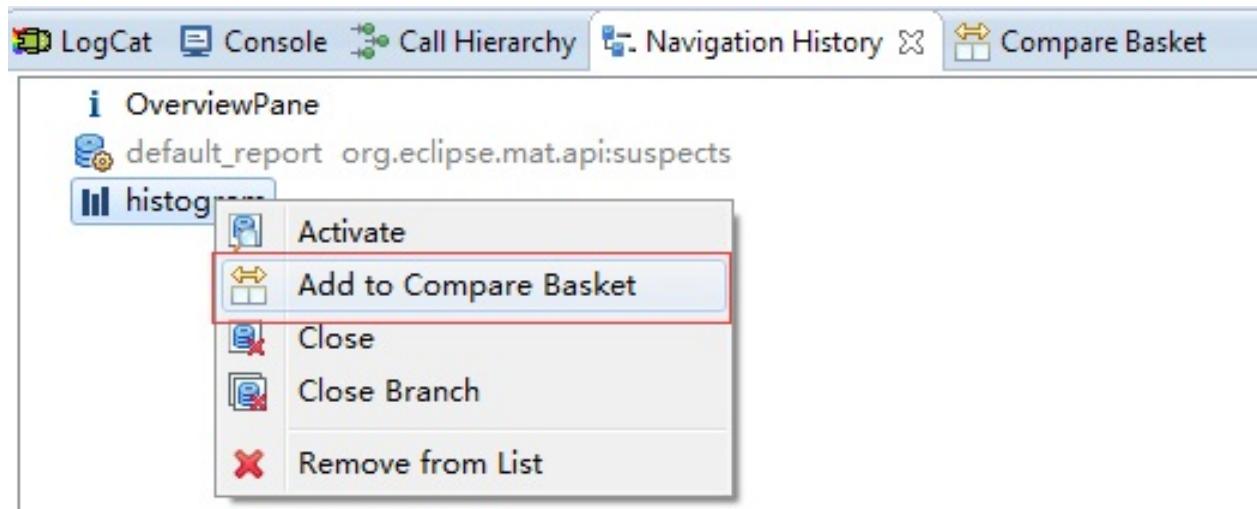
| Class Name | Ref. Objects | Sh. |
|--|--------------|-----|
| <Regex> | <Numeric> | |
| com.tencent.eyeplan.util.AntiRadiationUtil @ 0x12c14400 System Class | 1 | |
| mContext com.tencent.eyeplan.activity.MainActivity @ 0x12dcbb00 | 1 | |

从这个路径可以看出是一个antiRadiationUtil工具类对象持有了MainActivity的引用导致MainActivity无法释放。此时就要进入代码分析此时antiRadiationUtil的引用持有是否合理（如果antiRadiationUtil持有了MainActivity的context导致节目退出后MainActivity无法销毁，那一般都属于内存泄露了）。

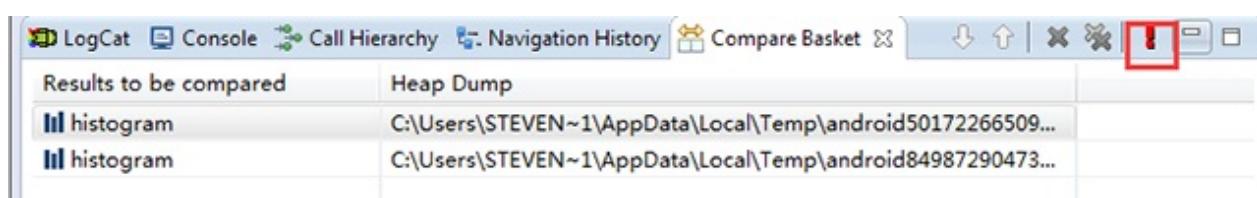
- MAT对比操作前后的hprof来定位内存泄露的根因所在

为查找内存泄漏，通常需要两个 Dump结果作对比，打开 Navigator History面板，将两个表的 Histogram结果都添加到 Compare Basket中去

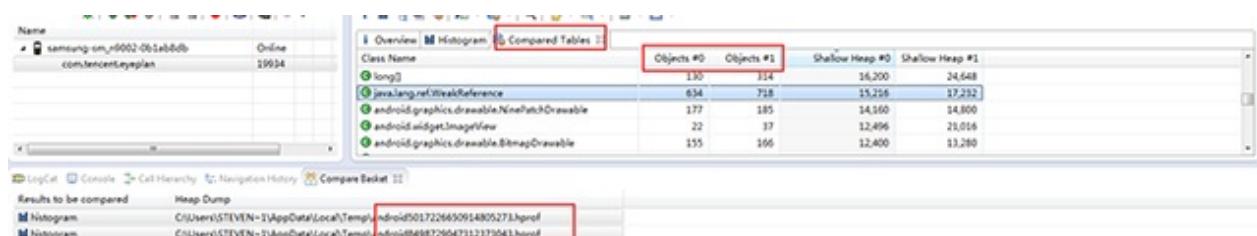
- A) 第一个HPROF 文件(usingFile > Open Heap Dump).
- B) 打开Histogram view.
- C) 在NavigationHistory view里 (如果看不到就从Window >show view>MAT- Navigation History), 右击histogram然后选择Add to Compare Basket .



- D) 打开第二个HPROF 文件然后重做步骤2和3。
- E) 切换到Compare Basket view, 然后点击Compare the Results (视图右上角的红色"!"图标)。



- F) 分析对比结果



可以看出两个hprof的数据对象对比结果。

通过这种方式可以快速定位到操作前后所持有的对象增量，从而进一步定位出当前操作导致内存泄露的具体原因是泄露了什么数据对象。

注意：

如果是用 MAT Eclipse 插件获取的 Dump文件，不需要经过转换则可在MAT中打开，Adt会自动进行转换。

而手机SDK Dump 出的文件要经过转换才能被 MAT识别，Android SDK提供了这个工具 hprof-conv (位于 sdk/tools下)

首先，要通过控制台进入到你的 android sdk tools 目录下执行以下命令： ./hprof-conv xxx-a.hprof xxx-b.hprof

例如 hprof-conv input.hprof out.hprof

此时才能将out.hprof放在eclipse的MAT中打开。

Ok，下面将给大家介绍一个屌炸天的工具 -- LeakCanary 。

使用 LeakCanary 检测 Android 的内存泄漏

什么是 [LeakCanary](#) 呢？为什么选择它来检测 Android 的内存泄漏呢？

别急，让我来慢慢告诉大家！

LeakCanary 是国外一位大神 Pierre-Yves Ricau 开发的一个用于检测内存泄露的开源类库。一般情况下，在对战内存泄露中，我们都会经过以下几个关键步骤：

- 1、了解 OutOfMemoryError 情况。
- 2、重现问题。
- 3、在发生内存泄露的时候，把内存 Dump 出来。
- 4、在发生内存泄露的时候，把内存 Dump 出来。
- 5、计算这个对象到 GC roots 的最短强引用路径。
- 6、确定引用路径中的哪个引用是不该有的，然后修复问题。

很复杂对吧？

如果有一个类库能在发生 OOM 之前把这些事情全部都搞定，然后你只要修复这些问题就好了。LeakCanary 做的就是这件事情。你可以在 debug 包中轻松检测内存泄露。

一起来看这个例子（摘自 LeakCanary 中文使用说明，下面会附上所有的参考文档链接）：

```
class Cat {  
}  
  
class Box {  
    Cat hiddenCat;  
}  
class Docker {  
    // 静态变量，将不会被回收，除非加载 Docker 类的 ClassLoader 被回收。  
    static Box container;  
}  
  
// ...  
  
Box box = new Box();  
  
// 薛定谔之猫  
Cat schrodingerCat = new Cat();  
box.hiddenCat = schrodingerCat;  
Docker.container = box;
```

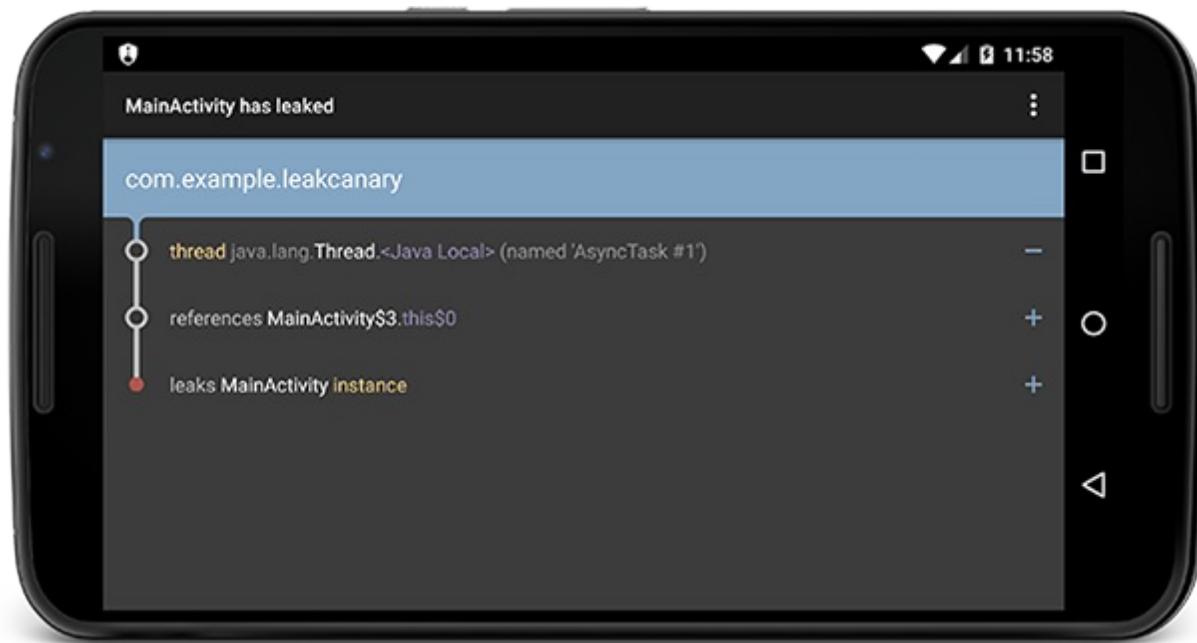
当发现有内存泄露的时候，你会看到一个很漂亮的 leak trace 报告：

- GC ROOT static Docker.container
- references Box.hiddenCat
- leaks Cat instance

我们知道，你很忙，每天都有一大堆需求。所以我们把这个事情弄得很简单，你只需要添加一行代码就行了。然后 LeakCanary 就会自动侦测 activity 的内存泄露了。

```
public class ExampleApplication extends Application {  
    @Override public void onCreate() {  
        super.onCreate();  
        LeakCanary.install(this);  
    }  
}
```

然后你会在通知栏看到这样很漂亮的一个界面：



以很直白的方式将内存泄露展现在我们的面前。

- Demo

一个非常简单的 LeakCanary demo: [一个非常简单的 LeakCanary demo](#)

- 接入

在 build.gradle 中加入引用，不同的编译使用不同的引用：

```
dependencies {
    debugCompile 'com.squareup.leakcanary:leakcanary-android:1.3'
    releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.3'
}
```

* 如何使用

使用 RefWatcher 监控那些本该被回收的对象。

```
RefWatcher refWatcher = {...};
```

```
// 监控 refWatcher.watch(schrodingerCat);
```

`LeakCanary.install()` 会返回一个预定义的 `RefWatcher`，同时也会启用一个 `ActivityRefWatcher`，
在 Application 中进行配置：

```
public class ExampleApplication extends Application {
```

```
public static RefWatcher getRefWatcher(Context context) {  
    ExampleApplication application = (ExampleApplication) context.getApplicationContentResolver().getSystemService(Context.APPLICATION_SERVICE);  
    return application.refWatcher;  
}  
  
private RefWatcher refWatcher;  
  
@Override public void onCreate() {  
    super.onCreate();  
    refWatcher = LeakCanary.install(this);  
}  
  
}
```

使用 RefWatcher 监控 Fragment:

```
public abstract class BaseFragment extends Fragment {  
  
    @Override public void onDestroy() {  
        super.onDestroy();  
        RefWatcher refWatcher = ExampleApplication.getRefWatcher(getActivity());  
        refWatcher.watch(this);  
    }  
  
}
```

使用 RefWatcher 监控 Activity:

```
public class MainActivity extends AppCompatActivity {
```

```
.....
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    //在自己的应用初始Activity中加入如下两行代码
    RefWatcher refWatcher = ExampleApplication.getRefWatcher(this);
    refWatcher.watch(this);

    textView = (TextView) findViewById(R.id.tv);
    textView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            startAsyncTask();
        }
    });
}

private void async() {

    startAsyncTask();
}

private void startAsyncTask() {
    // This async task is an anonymous class and
    // therefore has a hidden reference to the outer
    // class MainActivity. If the activity gets destroyed
    // before the task finishes (e.g. rotation),
    // the activity instance will leak.
    new AsyncTask<Void, Void, Void>() {
        @Override
        protected Void doInBackground(Void... params) {
            // Do some slow work in background
            SystemClock.sleep(20000);
            return null;
        }
    }.execute();
}

}
}
````
```

## ● 工作机制

1. RefWatcher.watch() 创建一个 KeyedWeakReference 到要被监控的对象。
2. 然后在后台线程检查引用是否被清除，如果没有，调用GC。
3. 如果引用还是未被清除，把 heap 内存 dump 到 APP 对应的文件系统中的一个 .hprof 文件中。
4. 在另外一个进程中的 HeapAnalyzerService 有一个 HeapAnalyzer 使用HAHA 解析这个文件。
5. 得益于唯一的 reference key, HeapAnalyzer 找到 KeyedWeakReference, 定位内存泄露。
6. HeapAnalyzer 计算 到 GC roots 的最短强引用路径，并确定是否是泄露。如果是的话，建立导致泄露的引用链。
7. 引用链传递到 APP 进程中的 DisplayLeakService， 并以通知的形式展示出来。

ok,这里就不再深入了，想要了解更多就到 [作者 github 主页](#) 这去哈。

## 总结

- 对 Activity 等组件的引用应该控制在 Activity 的生命周期之内；如果不能就考虑使用 getApplicationContext 或者 getApplication，以避免 Activity 被外部长生命周期的对象引用而泄露。
- 尽量不要在静态变量或者静态内部类中使用非静态外部成员变量（包括context），即使要使用，也要考虑适时把外部成员变量置空；也可以在内部类中使用弱引用来引用外部类的变量。
- 对于生命周期比Activity长的内部类对象，并且内部类中使用了外部类的成员变量，可以这样做避免内存泄漏：
- 将内部类改为静态内部类
- 静态内部类中使用弱引用来引用外部类的成员变量
- Handler 的持有的引用对象最好使用弱引用，资源释放时也可以清空 Handler 里面的消息。比如在 Activity onStop 或者 onDestroy 的时候，取消掉该 Handler 对象的 Message和 Runnable.
- 在 Java 的实现过程中，也要考虑其对象释放，最好的方法是在不使用某对象时，显式地将此对象赋值为 null，比如使用完Bitmap 后先调用 recycle(), 再赋为null,清空对图片等资源有直接引用或者间接引用的数组（使用 array.clear(); array = null）等，最好遵循谁创建谁释放的原则。
- 正确关闭资源，对于使用了BroadcastReceiver, ContentObserver, File, 游标 Cursor, Stream, Bitmap等资源的使用，应该在Activity销毁时及时关闭或者注销。
- 保持对对象生命周期的敏感，特别注意单例、静态对象、全局性集合等的生命周期。

以上部分图片、实例代码和文段都摘自或参考以下文章：

支付宝：

[Android怎样coding避免内存泄露](#)

[支付宝钱包Android内存治理](#)

IBM：

[Java的内存泄漏](#)

Android Design Patterns：

[How to Leak a Context: Handlers & Inner Classes](#)

伯乐在线团队：

[Android性能优化之常见的内存泄漏](#)

我厂同学：

[Dalvik虚拟机 Finalize 方法执行分析](#)

腾讯bugly：

[内存泄露从入门到精通三部曲之基础知识篇](#)

[内存泄露从入门到精通三部曲之排查方法篇](#)

[内存泄露从入门到精通三部曲之常见原因与用户实践](#)

LeakCanary：

[LeakCanary 中文使用说明](#)

[LeakCanary: 让内存泄露无所遁形](#)

<https://github.com/square/leakcanary>

# Android Weak Handler：可以避免内存泄漏的Handler库

来源:[www.jcodecraeer.com](http://www.jcodecraeer.com)

android使用java作为其开发环境。java的跨平台和垃圾回收机制已经帮助我们解决了底层的一些问题。但是尽管有了垃圾回收机制，在开发android的时候仍然时不时的遇到 out of memory 的问题，这个时候我们不禁要问，垃圾回收机器去哪儿了？

我们主要讲的是handler引起的泄漏，并给出三种解决办法，其中最后一种方法就是我们想介绍的 WeakHandler 库。

可能导致泄漏问题的handler一般会被提示 Lint警告：

```
This Handler class should be static or leaks might occur
意思：class使用静态声明否则可能出现内存泄露。
```

```
public class MainActivity extends ActionBarActivity {

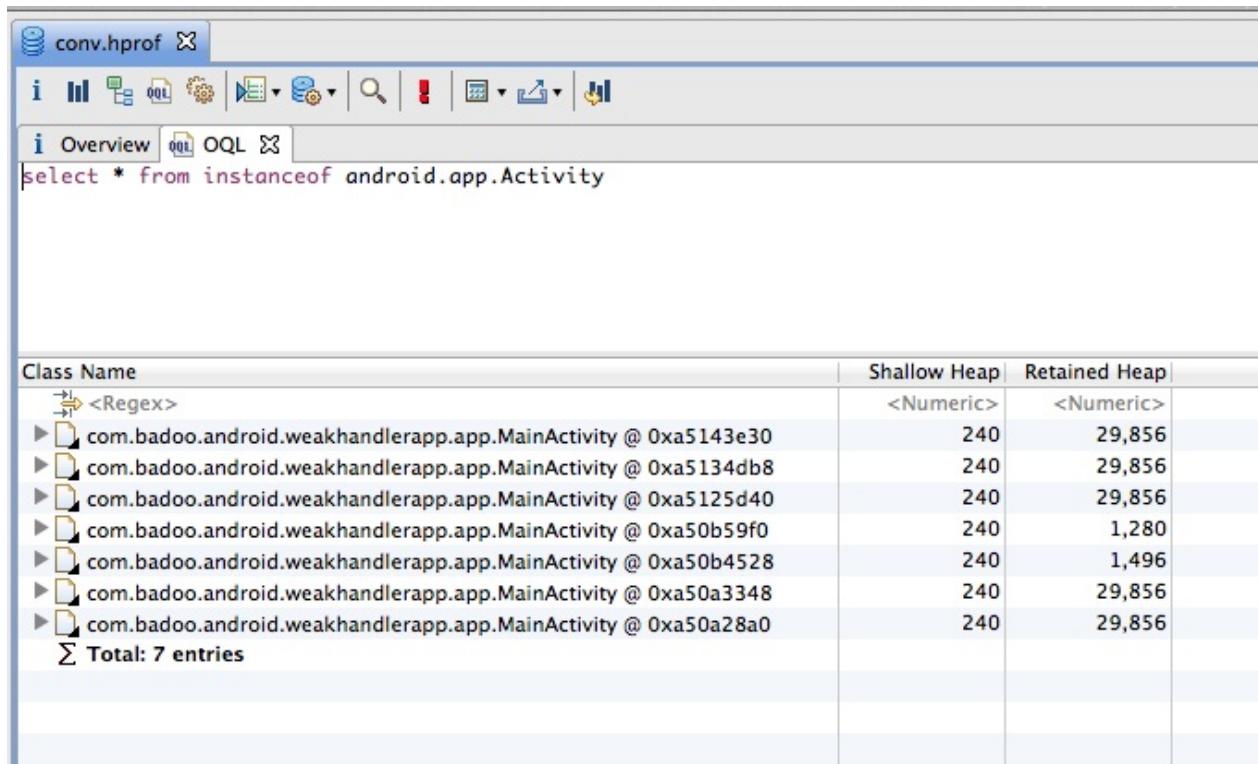
 private Handler mHandler = new Handler();
 private TextView mTextView;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);

 mTextView = (TextView) findViewById(R.id.hello_text);
 mHandler.postDelayed(new Runnable() {

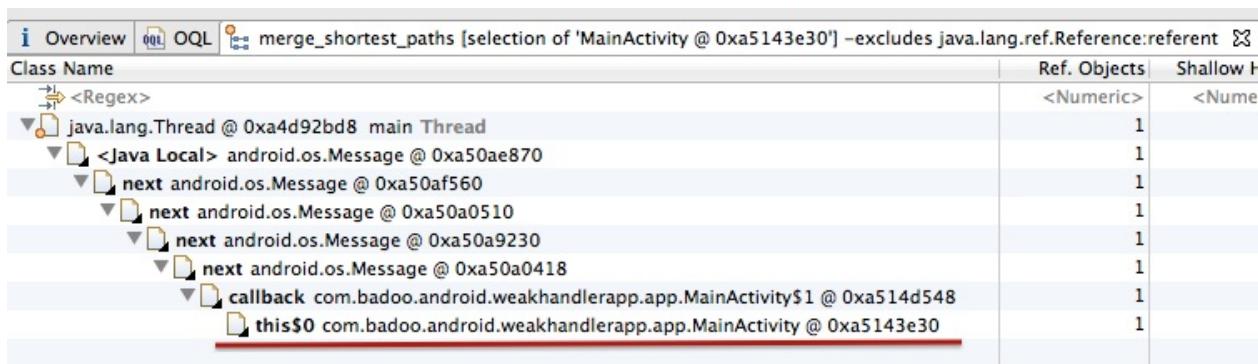
 @Override
 public void run() {
 mTextView.setText("Done");
 }
 }, 800000);
 }
}
```

这是一个基本的activity。在handler的post方法中我们加入了一个匿名的Runnable，同时我将其执行延迟了整整80秒。我们运行这个程序，并且旋转几次手机，然后分析内存。



现在内存中有7个activity了，这太不靠谱了，所以我们来研究下为什么GC没有清理它。

(上图中我查询内存中activity列表时用的是oql(对象查询语言)，简单强大的工具，ps 怎么用的，谁能告诉我？)



从上图中我们可以看到其中一个对mainactivity的引用是来自 this\$0，this\$0 是什么呢？以下是关于 this\$0 的解释：

-----什么是 this\$0 -----

非static的 inner class 里面都会有一个 this\$0 的字段保存它的父对象。在Java中，**非静态(匿名)内部类会默认隐性引用外部类对象。而静态内部类不会引用外部类对象。**一个编译后的inner class 很可能是这样的：

```
class parent$inner{
 synthetic parent this$0;
 parent$inner(parent this$0){
 this.this$0 = this$0;
 this$0.foo();
 }
}
```

-----什么是 this\$0 结束-----

在我们的代码中，匿名的Runnable是一个非静态的内部类，因此他会使用 this\$0 来保存 MainActivity，然后Runnable会继续被它的callback引用，而callback又接着被接下来一连串的message引用，这样主线程的引用就太他妈多了。当Activity finish后，延时消息会继续存在主线程消息队列中80秒，然后处理消息，因此handler在继续存在于内存中，而handler引用了Activity，在我们旋转手机的时候，Activity 不停的重建和finish，导致多个activity的引用出现。

一旦将Runnable或者是Message 发送进handler，将保存一连串的引用了主线程（这里是MainActivity吧）的Message命令，直到message执行完。如果发送Runnable设置了延迟时间，那么至少在这段延迟时间内内存泄漏是肯定的，如果是直接发送，在Message比较大的情况下，也是有可能发生暂时的泄漏的。

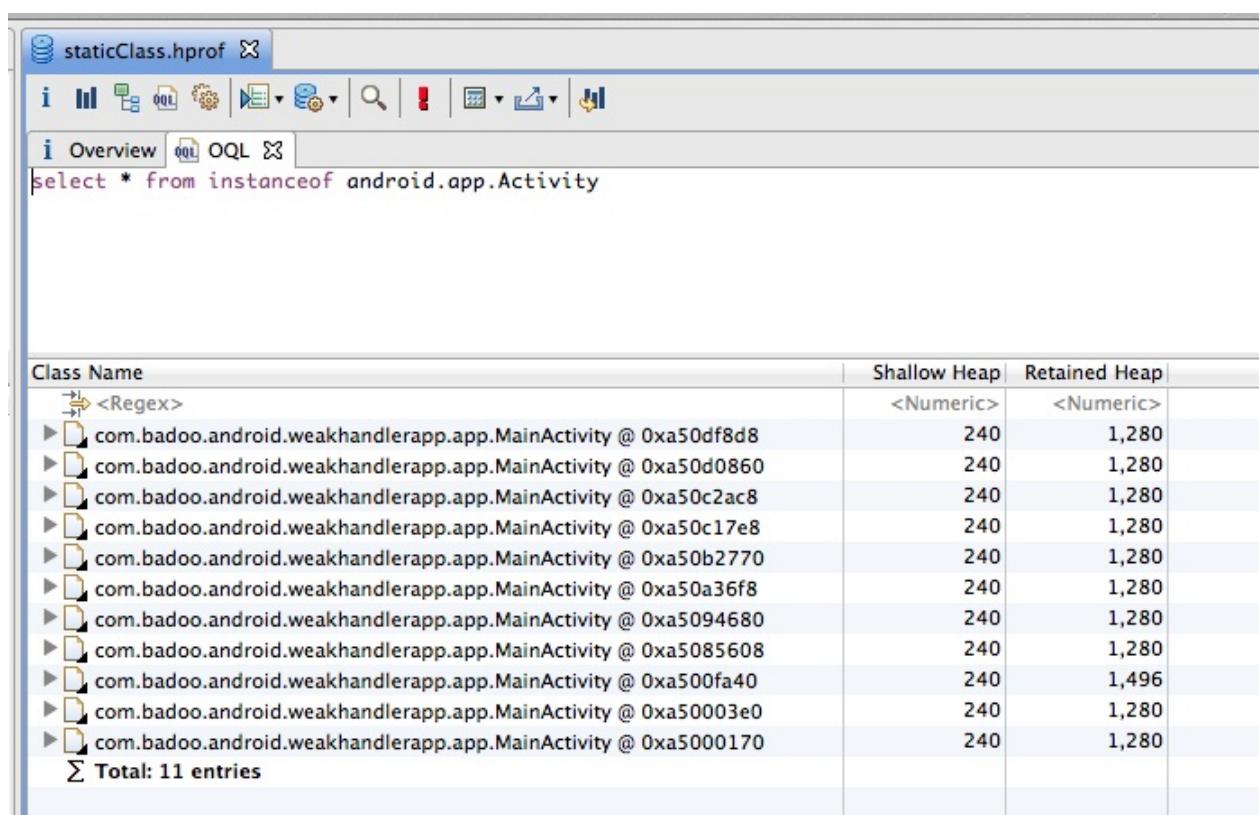
## 解决办法一：使用Static

```

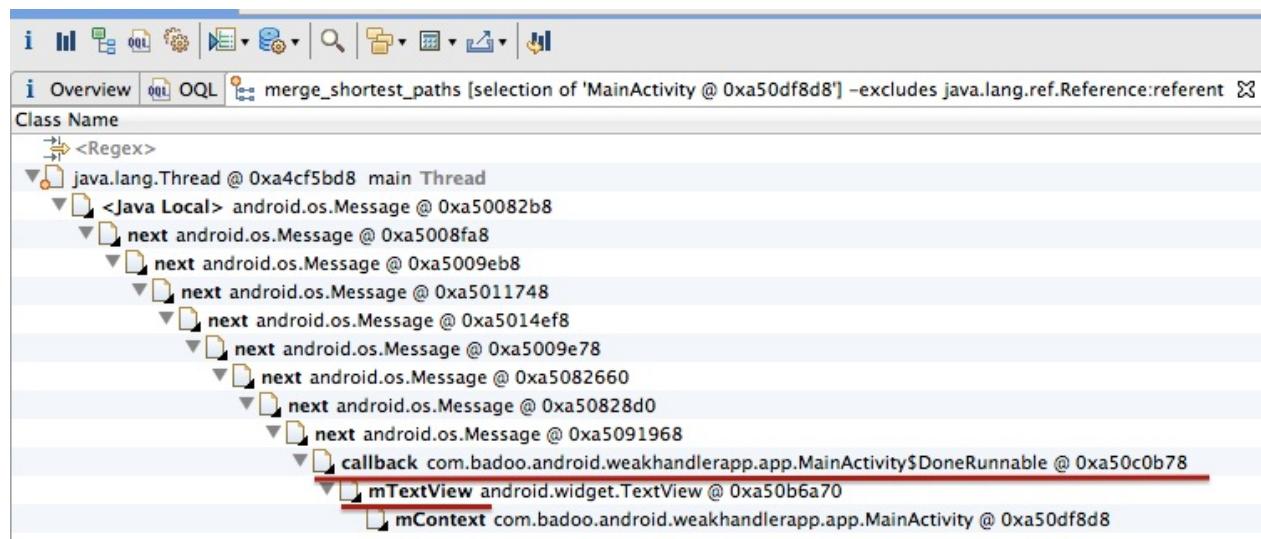
9 public class MainActivity extends ActionBarActivity {
10
11 private Handler mHandler = new Handler();
12 private TextView mTextView;
13
14 @Override
15 protected void onCreate(Bundle savedInstanceState) {
16 super.onCreate(savedInstanceState);
17 setContentView(R.layout.activity_main);
18
19 mTextView = (TextView) findViewById(R.id.hello_text);
20 mHandler.postDelayed(new DoneRunnable(mTextView), 800000);
21 }
22
23 private static final class DoneRunnable implements Runnable {
24 private final TextView mTextView;
25
26 protected DoneRunnable(TextView textView) {
27 mTextView = textView;
28 }
29
30 @Override
31 public void run() {
32 mTextView.setText("Done");
33 }
34 }
35 }

```

再次执行，同时旋转手机，分析内存如下：



尼玛，还是一样的。我们看看是谁还拉着activity不放：



在最底下我们发现activity继续被DoneRunnable里面mTextView中的mContext引用着。看来在这种情况下，看来仅仅使用static并没有解决问题啊。还需要做点工作才行。

### 静态的Runnable加WeakReference

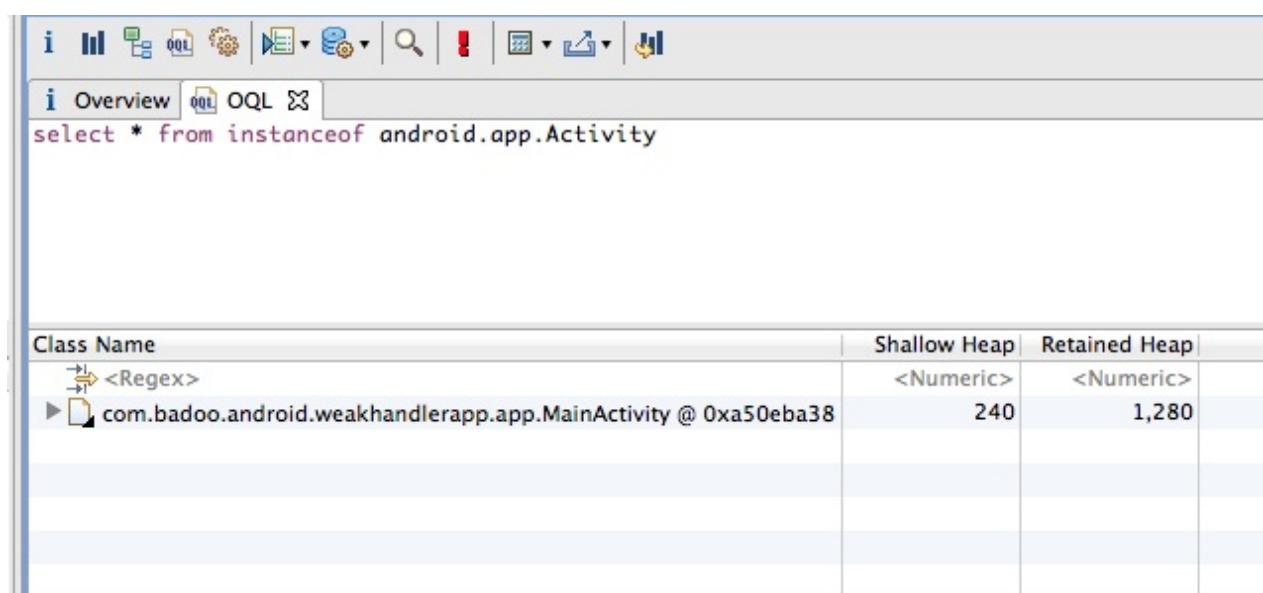
既然是因为mTextView引起的，那我们把mTextView换成弱引用好了：

```

10
11 public class MainActivity extends ActionBarActivity {
12
13 private Handler mHandler = new Handler();
14 private TextView mTextView;
15
16 @Override
17 protected void onCreate(Bundle savedInstanceState) {
18 super.onCreate(savedInstanceState);
19 setContentView(R.layout.activity_main);
20
21 mTextView = (TextView) findViewById(R.id.hello_text);
22 mHandler.postDelayed(new DoneRunnable(mTextView), 800000);
23 }
24
25 private static final class DoneRunnable implements Runnable {
26 private final WeakReference<TextView> mTextViewRef;
27
28 protected DoneRunnable(TextView textView) {
29 mTextViewRef = new WeakReference<TextView>(textView);
30 }
31
32 @Override
33 public void run() {
34 final TextView textView = mTextViewRef.get();
35 if (textView != null) {
36 textView.setText("Done");
37 }
38 }
39 }
40 }

```

需要注意的，既然mTextView是弱引用，所以随时都可能为null，因此需要在使用前判断是否为空。好了继续看看内存的情况：



all right,我想我们已经完美的解决问题了。总结一下我们做了哪些工作：

## 使用静态的内部类

对所有handler/Runnable中的变量都用弱引用。

但是这种方式代码是不是很多，而且还必须得小心翼翼。

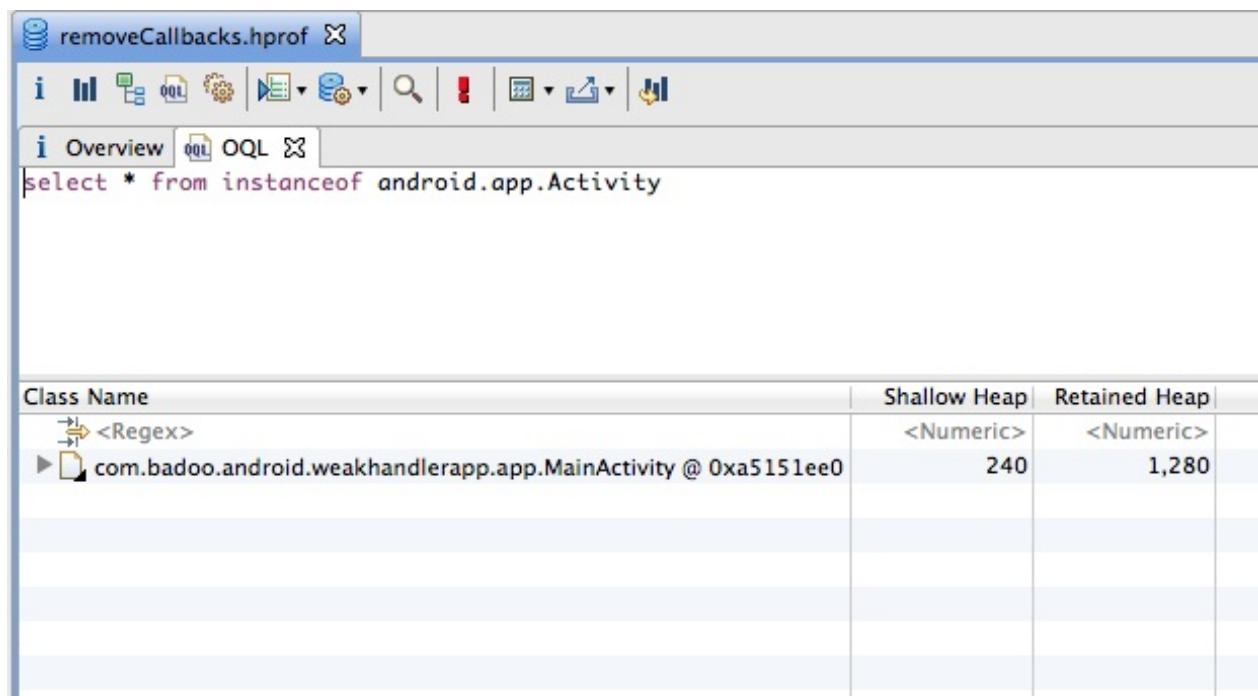
## 在onDestroy中清理掉所有Messages

Handler有个很方便的方法：removeCallbacksAndMessages,当参数为null的时候，可以清除掉所有跟次handler相关的Runnable和Message，我们在onDestroy中调用次方法也就不会发生内存泄漏了。

```

8
9 public class MainActivity extends ActionBarActivity {
10
11 private Handler mHandler = new Handler();
12 private TextView mTextView;
13
14 @Override
15 protected void onCreate(Bundle savedInstanceState) {
16 super.onCreate(savedInstanceState);
17 setContentView(R.layout.activity_main);
18
19 mTextView = (TextView) findViewById(R.id.hello_text);
20 mHandler.postDelayed(new Runnable() {
21
22 @Override
23 public void run() {
24 mTextView.setText("Done");
25 }
26 }, 800000);
27 }
28
29 @Override
30 protected void onDestroy() {
31 super.onDestroy();
32 mHandler.removeCallbacksAndMessages(null);
33 }
34 }
```

运行，旋转手机



但是如果你对代码有更高的要求，觉得这样还不方便可以使用作者提供的WeakHandler库

## WeakHandler

WeakHandler使用起来和handler一模一样，但是他是安全的，WeakHandler使用如下：

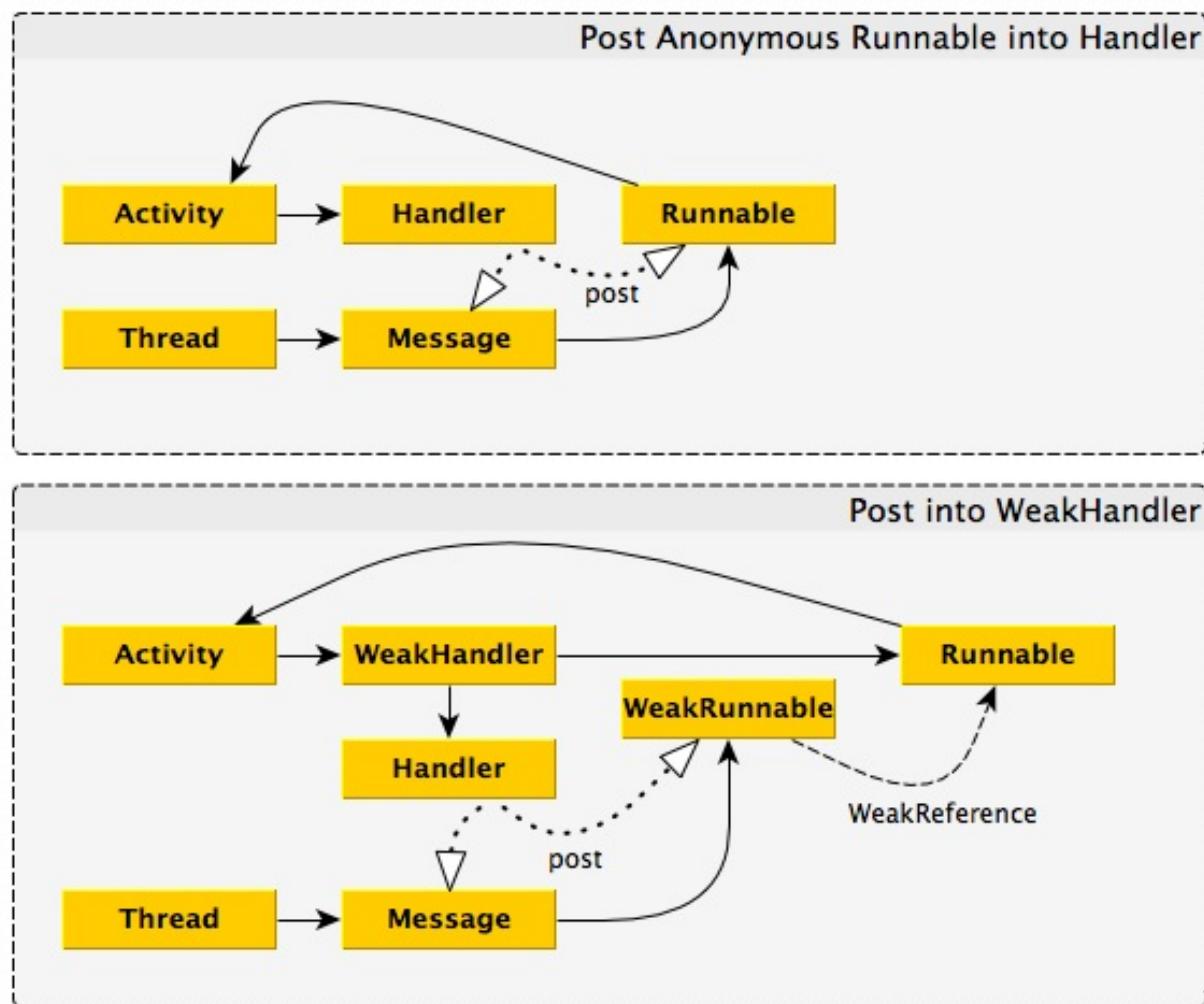
```

7
8 public class MainActivity extends ActionBarActivity {
9
10 private WeakHandler mHandler = new WeakHandler();
11 private TextView mTextView;
12
13 @Override
14 protected void onCreate(Bundle savedInstanceState) {
15 super.onCreate(savedInstanceState);
16 setContentView(R.layout.activity_main);
17
18 mTextView = (TextView) findViewById(R.id.hello_text);
19 mHandler.postDelayed(new Runnable() {
20
21 @Override
22 public void run() {
23 mTextView.setText("Done");
24 }
25 }, 800000);
26 }
27 }
```

你只需要把以前的Handler替换成WeakHandler就行了。

## WeakHandler的实现原理

WeakHandler的思想是将Handler和Runnable做一次封装，我们使用的是封装后的WeakHandler，但其实真正起到handler作用的是封装的内部，而封装的内部对handler和runnable都是用的弱引用。



第一幅图是普通handler的引用关系图

第二幅图是使用WeakHandler的引用关系图

其实原文有对WeakHandler更多的解释，但是表述起来也挺复杂的。

- 原文地址：<https://techblog.badoo.com/blog/2014/10/09/calabash-android-query/>
- github项目地址：<https://github.com/badoo/android-weak-handler>

# Android 5.1 Webview内存泄漏新场景

来源:<https://coolpers.github.io/webview/memory/leak/2015/07/16/android-5.1-webview-memory-leak.html>

## 问题现象

今天发现App存在 WebView 泄漏情况，还比较严重。并且只是发生在 Android 5.1 系统。

GC roots 如下：

|                                                                                           |   |
|-------------------------------------------------------------------------------------------|---|
| com.baidu.appsearch.AppSearch @ 0x12c71900 Native Stack                                   | 2 |
| └ mComponentCallbacks java.util.ArrayList @ 0x12c76980                                    | 2 |
| └ array java.lang.Object[12] @ 0x13c5b8c0                                                 | 2 |
| └ [3] com.android.org.chromium.android_webview.AwContents\$AwComponentCallbacks @ 0x12ec4 | 1 |
| └ this\$0 com.android.org.chromium.android_webview.AwContents @ 0x13b8f3a0                | 1 |
| └ mContainerView com.baidu.appsearch.webview.AppSearchWebView @ 0x134ca400                | 1 |
| └ [2] com.android.org.chromium.android_webview.AwContents\$AwComponentCallbacks @ 0x12c03 | 1 |
| └ this\$0 com.android.org.chromium.android_webview.AwContents @ 0x13b916a0                | 1 |
| └ mContainerView com.baidu.appsearch.webview.AppSearchWebView @ 0x13bd8c00                | 1 |
| Σ Total: 2 entries                                                                        |   |

每新打开一次这个WebViewActivity，就会发生就会发生一次改Webview实例无法释放，新增一个对象。

上图中的两个 AppSearchWebView 实例，就是由于打开了两次导致。

## 问题分析

出现了这个问题分析起来还是比较简单的，根据这个引用关系，我们可以直观的看到是由于 Appsearch (extends Application) 的 mComponentCallbacks 一直在强引用 AwComponentCallbacks，导致无法释放。然后 AwComponentCallbacks -> AwContents > AppSearchWebView。

通过分析代码发现关键在于 AwContents 这里的 AwComponentsCallbacks 为什么没有释放。

```
@Override
public void onAttachedToWindow() {
 if (isDestroyed()) return;
 if (mIsAttachedToWindow) {
 Log.w(TAG, "onAttachedToWindow called when already attached. Ignoring");
 return;
 }

 mComponentCallbacks = new AwComponentCallbacks();
 mContext.registerComponentCallbacks(mComponentCallbacks);
}

@Override
public void onDetachedFromWindow() {
 if (isDestroyed()) return;
 if (!mIsAttachedToWindow) {
 Log.w(TAG, "onDetachedFromWindow called when already detached. Ignoring");
 return;
 }

 if (mComponentCallbacks != null) {
 mContext.unregisterComponentCallbacks(mComponentCallbacks);
 mComponentCallbacks = null;
 }

}
```

看这段代码看不出来什么问题，onAttach的时候 register，detach的时候 unregister，不会存在问题。

但是为什么呢？

难道是由于 if (isDestroyed()) return 这条return引起的？

当调用 Webview.destroy() 后这个判断返回true。

我们看下哪里调用了 webview.destroy()

```
// source from WebViewActivity

@Override
protected void onDestroy() {
 super.onDestroy();

 if (mWebView != null) {
 mWebView.destroy();
 }
}
```

很多应用应该都是这么做的，包括系统浏览器，在Activity destroy的时候，调用 webview 的destroy。并且一直工作的很好。

通过调试发现，确实是由于此调用导致的。onDestroy 发生在 onDetach 之前。

那为什么 android 5.1 之前的代码没有问题呢？

看下代码：

```
// AwContents.java

@Override
public void onDetachedFromWindow() {
 if (!mIsAttachedToWindow) {
 Log.w(TAG, "onDetachedFromWindow called when already detached. Ignoring");
 return;
 }

 if (mComponentCallbacks != null) {
 mContext.unregisterComponentCallbacks(mComponentCallbacks);
 mComponentCallbacks = null;
 }

}
```

相对于 5.1 的代码少了那句 `if (isDestroyed()) return;`

## 规避方法

三水哥的解决方案可以：在destroy之前，把webview 从 parent 中 remove 掉，同样可以提前detach。

```
protected void onDestroy() {
 if (mWebView != null) {
 ((ViewGroup) mWebView.getParent()).removeView(mWebView);
 mWebView.destroy();
 mWebView = null;
 }
 super.onDestroy();
}
```

# Android中导致内存泄漏的竟然是它----Dialog

来源:[腾讯Bugly-微信](#)



## 一， 内存泄漏的 Bug 猛增

最近接入了公司组件分析云，在 memory-leak-dialog/memory-leak-dialogApp 进行 mokey 测试的时候顺便检测内存泄漏问题。于是，就在前天的测试中，楼主一瞬间收到了 4 个这样的 Bug 单，瞬间心理无比纠结，真有千万只羊驼向我奔来。

- [Magnifier分析云][activity\_leak][no\_used]com.example.tribe.account.login.LoginActivity leaked(1 time)
- [Magnifier分析云][activity\_leak][no\_used]com.example.tribe.account.login.LoginActivity leaked(5 times)
- [Magnifier分析云][activity\_leak][no\_used]com.example.tribe.account.login.LoginActivity leaked(1 time)
- [Magnifier分析云][activity\_leak][no\_used]com.example.tribe.account.login.LoginActivity leaked(10 times) ↗

memory-leak-dialog/memory-leak-dialog

登录页面出现内存泄漏？？！！ 楼主的代码是如此的完美而无懈可击，这么可能出现这么多泄漏的问题？分析云测漏的工具有问题吧！？

插播什么是 Activity 泄漏：Android 中 Activity 代表一个页面，拥有一段生命周期，生命周期结束后，Activity 对象应当在之后某个合适的时机被 VM 回收内存。出现了泄漏就意味着 Activity 生命周期结束后，VM 发现 Activity 一直被持有，没有回收这些无用的内存。

按照以往的经验，大部分 Activity 泄漏的原因都是由于 Handler 内部类长时间挂在线程中导致的。而这块我们 App 已经考虑并处理了。究竟是哪泄漏了？

## 二， WebView导致内存泄漏众所周知

带着怀疑的心态并且为了证明清白，我一个个点进去看了，总共有三条不同的引用链。为了后续说明，这里取了个名字：

- ① AuthDialog 引用链

GC Path:

```
com.example.tribe.account.login.LoginActivity@42b67c68
|-com.example.connect.auth.AuthDialog@42f6b2d8
 |-com.example.connect.auth.AuthDialog$3@42cda2d0
 |-android.os.Message@42c74b70
 |-java.lang.Thread(CookieSyncManager)@42b6dad0
```

- ② BrowserFrame 引用链

GC Path:

```
com.example.tribe.account.login.LoginActivity@42c793e0
|-android.webkit.BrowserFrame@43236680
 |-android.os.Message@42c65f48
 |-android.os.Message@42c98778
 |-android.os.Message@42c80b60
 |-android.os.Message@42d173d0
 |-android.os.Message@42eec548
 |-android.os.MessageQueue@42ba6980
 |-java.lang.Thread(WebViewCoreThread)@42beb5e0
```

- ③ IClipboardDataPaste 引用链

## GC Path:

```
com.example.tribe.account.login.LoginActivity@42c87720
|-android.webkit.WebViewClassic@42bfff50
 |-android.webkit.WebViewClassic$IClipboardDataPasteEventImpl@42d19ef0
 |-android.webkit.WebViewClassic$IClipboardDataPasteEventImpl$1@42cabf20
```

看来这次情况有点不同！由于 Monkey 测试的机型比较少，这里所有的 Bug 都来自一部三星 GT-I9300@android+4.3 手机。

为了快速解决问题，楼主询问了其他同事和 StackOverflow，发现这其中这三个类 CookieSyncManager, WebView, WebViewClassic 已经被很多人提起过，它们会导致内存泄漏！初步有如下的结论如下：

- 1.CookieSyncManager 是个全局静态单例，操作系统内部使用了 App 的 Activity 作为 Context 构造了它的实例。我们应该在 App 启动的时候，抢先帮系统创建这个单例，而且要用 applicationContext，让它不会引用到 Activity。

```
/**
 * Create a singleton CookieSyncManager within a context
 * @param context
 * @return CookieSyncManager
 */
public static synchronized CookieSyncManager createInstance(
 Context context) {
 if (context == null) {
 throw new IllegalArgumentException("Invalid context argument");
 }

 if (sRef == null) {
 sRef = new CookieSyncManager(context);
 }
 return sRef;
}
```

- 2. 使用 WebView 的页面（Activity），在生命周期结束页面退出（onDestory）的时候，需要主动调用 `WebView.onPause()` 以及 `WebView.onDestroy()` 以便让系统释放 WebView 相关资源。



You should be able to stop / resume these threads by calling the onPause / onResume on the webview.

34



Those are however hidden, so you will need to do it through reflection. The following code worked for me:



```
Class.forName("android.webkit.WebView").getMethod("onPause", (Class[]) null).invoke(webView,
```

Where webView is the instance of WebView.

- 3. WebView 内存泄漏是众所周知的，建议另外启动一个进程专门运行 WebView。不要9998，不要9999，我们要100%！WebView 用完之后就把进程杀死，即使泄漏了也无碍。

按照以上的种种结论，我们都认定了这里面就是 WebView 引起的。

但是！我们的应用主进程 LoginActivity 根本没有用到 WebView 啊！！

### 三. 第三方 jar 包使用 WebView 这可如何是好

根据以上的 AuthDialog 引用链，楼主把目标锁定了某sdk：

翻了一阵子恶心的混淆后的代码，找到下面这么一段。SDK 确实创建了 WebView 实例，并且用的是客户程序的 Activity 对象作为 WebView 的 Context 如下：

c 跟 j 都是 SDK 中继承于 WebView 的一个子类，k 是登录接口的输入参数 Activity。这里创建了 c 对象之后向上塑形赋给了 j。

```

129 return var2;
130 }
131
132 private void b() {
133 this.c();
134 LayoutParams var1 = new LayoutParams(-1, -1);
135 this.j = new c(this.k); k 是 LoginActivity 引用
136 this.j.setLayoutParams(var1);
137 this.e = new FrameLayout(this.k);
138 var1.gravity = 17;
139 this.e.setLayoutParams(var1);
140 this.e.addView(this.j);
141 this.e.addView(this.g);
142 this.setContentView(this.e);
143 }

```

网上已经有很多例子表明，直接用 Activity 作为参数构建 WebView 就非常有可能导致 Activity 泄漏。

▲
38
▼

I conclude from above comments and further tests, that the problem is a bug in the SDK: when creating a WebView via XML layout, the activity is passed as the context for the WebView, not the application context. When finishing the activity, the WebView still keeps references to the activity, therefore the activity doesn't get removed from the memory. I filed a bug report for that, see the link in the comment above.

✓ webView = new WebView(getApplicationContext());

不过也看到了代码中，有调用了 WebView 的 `destroy()` 方法释放资源。但是这里似乎无法保证 `dismiss()` 一定会被执行。

```

AuthDialog.class

 Decompiled .class file, bytecode version: 50.0 (Java 6)

public void dismiss() {
 this.s.clear();
 this.d.removeCallbacksAndMessages((Object)null);
 if(this.isShowing()) {
 super.dismiss();
 }

 if(this.j != null) {
 this.j.destroy();
 this.j = null;
 }
}

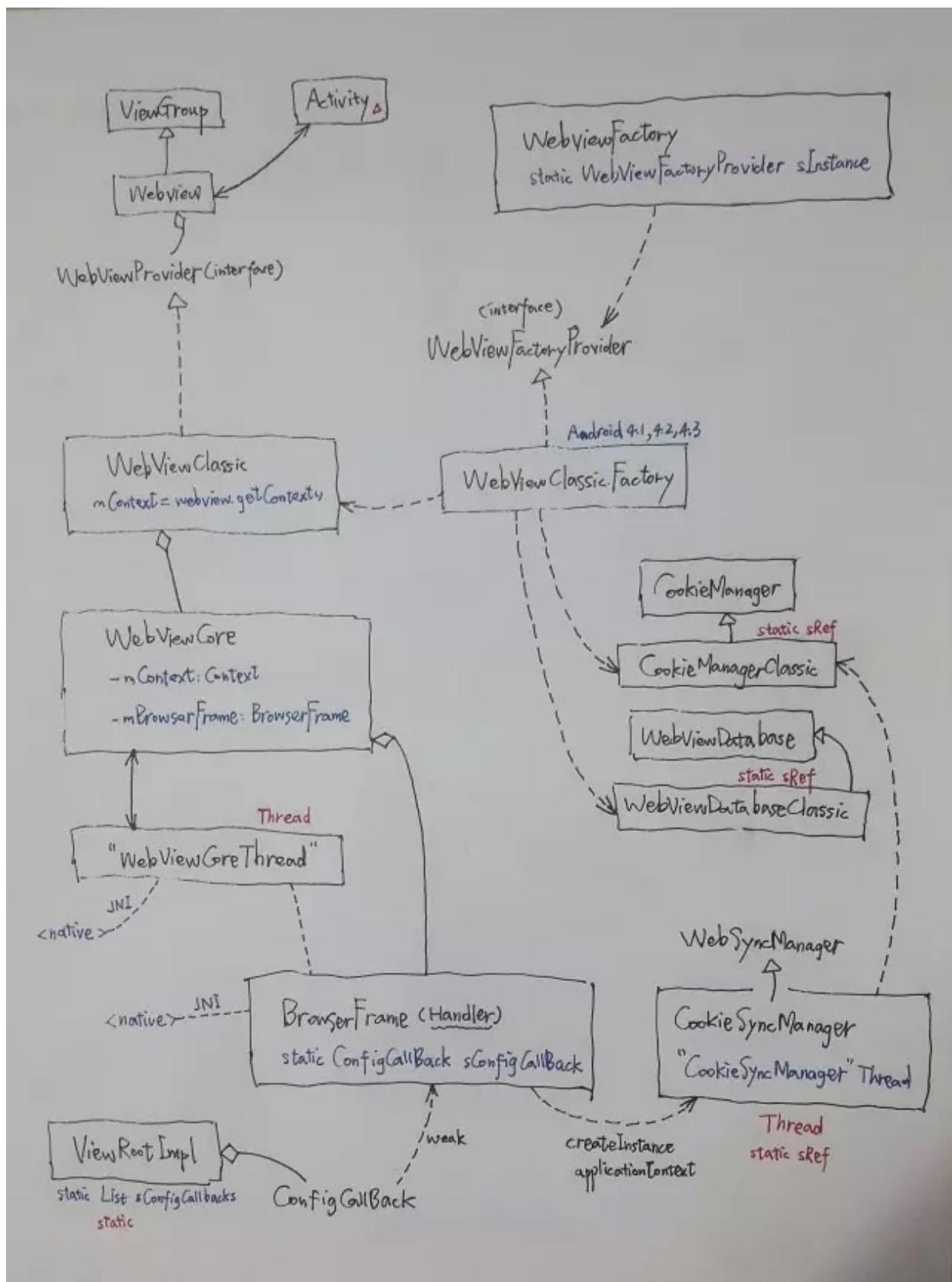
```

问题到这里发现比较麻烦了，SDK 对我们来说是第三方包，我们没法让第三方包不用 WebView，或者让第三方包把 WebView 放在另外一个进程中运行啊！于是，在 App 上面做规避暂时不好实现。于是找了 SDK 的童鞋一起分析了。

最终，大家都有了一个初步的共识，在 Android4.3 以下的旧版本，使用 Activity 对象创建 WebView，确实有可能导致内存泄漏。非常高兴能得到 SDK 童鞋的大力支持，一起分析，问题到这里有了初步的进展。

#### 四. 心结未解，翻看WebView源码了解根源

不过，问题到这里楼主心理还是有个很严重的疑惑没有解开（是什么疑惑呢？）。于是拿了 Android4.3 的源码又翻了一遍希望找寻这里头的根本原因，做了一点记录，针对 WebView 在 Java 层的结构画了一个不严谨的类图：源码来源：[http://androidxref.com/4.3\\_r2.1/](http://androidxref.com/4.3_r2.1/)



大概情况是这样：WebView 这套结构中，有一个工厂类 WebViewFactory 提供静态方法。

Android4.3(JellyBean) 版本通过 WebViewFactory 工厂类创建了一个全局单例对象 WebViewClassic\$Factory，然后使用这个 Factory 创建了一整套实现的代码 (XXXClassic): WebViewClassic, CookieManagerClassic, WebViewDatabaseClassic。

WebViewClassic 才是真正地实现 WebView 的各种 API。WebViewClassic 创建并维护了 WebViewCore 对象。

WebViewCore 创建了一个子线程“WebViewCoreThread”，这里是一个全局的单例的而且一旦启动就不会停止的 Thread！WebViewCore 会在这个子线程中创建维护并调用 BrowserFrame 的方法。

BrowserFrame 本身是一个属于“WebViewCoreThread”线程的 Handler 子类。

BrowserFrame 会被 native(c++) 层调用，然后将这些调用切换到“WebViewCoreThread”线程中去执行，比如刷新进度或者处理屏幕旋转事件等等。

BrowserFrame 还会调用 CookieSyncManager.createInstance()，这也是系统框架中唯一一处调用的地方！

```
public BrowserFrame(Context context, WebViewCore w, CallbackProxy proxy,
 WebSettingsClassic settings, Map<String, Object> javascriptInterfaces) {
 Context appContext = context.getApplicationContext();

 // Create a global JWebCoreJavaBridge to handle timers and
 // cookies in the WebCore thread.
 if (sJavaBridge == null) {
 sJavaBridge = new JWebCoreJavaBridge();
 // set WebCore native cache size
 ActivityManager am = (ActivityManager) context
 .getSystemService(Context.ACTIVITY_SERVICE);
 if (am.getMemoryClass() > 16) {
 sJavaBridge.setCacheSize(8 * 1024 * 1024);
 } else {
 sJavaBridge.setCacheSize(4 * 1024 * 1024);
 }
 // create CookieSyncManager with current Context
 CookieSyncManager.createInstance(appContext);
 // create PluginManager with current Context
 PluginManager.getInstance(appContext);
 }
}
```

看到这里之后，楼主发现以上所说的，提前帮系统调用

CookieSyncManager.createInstance(context.getApplicationContext()) 可能是没有效果的，因为系统本来就是这么做的。手机厂商修改这里的可能性不大。

CookieSyncManager 又是什么东西？同样的，它自己也创建一个子线程，线程名就叫“CookieSyncManager”，又是全局单例不会停！这个线程每过5分钟就会把缓存在内存中的 Cookie 进行持久化 syncFromRamToFlash()。

这里我们比较关心为什么 Activity 会泄漏，所以关键看看哪些类对象中持有了 Activity(Context) 引用： WebViewClassic, WebViewCore, BrowserFrame。

这套结构中有很多静态单例，还有子线程，想想也挺恶心的。而且三个关键的类都持有 Activity 引用。不过我们发现，其实 WebViewClassic, WebViewCore 这两个对象跟 WebView 对象的生命周期是一致的，Activity 销毁于是 WebView 销毁了，WebView 销毁了另外两个对象也跟着销毁。烟消云散。。。

留下两个孤独的子线程还在跑，还有全局静态的钉子户对象。

但是！ BrowserFrame 本身是 Handler，假如它因为 native 层的调用往“WebViewCoreThread”挂了一个消息，那么便可以建立一条引用链：

```
Thread->MessageQueue->Message->Handler(BrowserFrame)->Activity
```

**GC Path:**

```
com.example.tribe.account.login.LoginActivity@42c793e0
 |-android.webkit.BrowserFrame@43236680
 |-android.os.Message@42c65f48
 |-android.os.Message@42c98778
 |-android.os.Message@42c80b60
 |-android.os.Message@42d173d0
 |-android.os.Message@42eec548
 |-android.os.MessageQueue@42ba6980
 |-java.lang.Thread(WebViewCoreThread)@42beb5e0
```

好了，楼主的疑惑是什么？

## 五. 最后的疑惑

我们再来看看 AuthDialog 的引用链。

GC Path:

```
com.example.tribe.account.login.LoginActivity@42b67c68
|-com.example.connect.auth.AuthDialog@42f6b2d8
|-com.example.connect.auth.AuthDialog$3@42cda2d0
|-android.os.Message@42c74b70
|-java.lang.Thread(CookieSyncManager)@42b6dad0
```

换成 MAT 看会比较清晰：



楼主发现，这里 `CookieSyncManager` 线程，居然直接引用了 `Message` 对象！这是什么鬼？一般情况下，`HandlerThread` 持有一个 `MessageQueue` 对象，`MessageQueue` 才持有 `Message` 队列。

Java Local : A local variable. For example, input parameters, or locally created objects of methods that are still in the stack of a thread. Native stack. Input or output parameters in native code, for example user-defined JNI code or JVM internal code. Many methods have native parts, and the objects that are handled as method parameters become garbage collection roots. For example, parameters used for file, network, I/O, or reflection operations.

这里表明，`CookieSyncManager` 线程中存在某个 `Message` 的局部变量，而由于线程一直没有结束，所以局部变量一直没有被释放。而这个 `Message.obj` 成员引用了 `AuthDialog$3` 对象。

这是一个内部类，楼主发现内部类混淆之后的命名规则就是：第几个出现就命名为几。

`AuthDialog` 里面有很多内部类：

```
189 private void d() {
190 this.j.setVerticalScrollBarEnabled(false);
191 this.j.setHorizontalScrollBarEnabled(false);
192 this.j.setWebViewClient(new AuthDialog.LoginWebViewClient(null));
193 this.j.setWebChromeClient(new WebChromeClient());
194 this.j.clearFormData();
195 this.j.clearSslPreferences();
196 this.j.setOnLongClickListener(new OnLongClickListener() {
197 • 1
198 public boolean onLongClick(View var1) { 第1个
199 return true;
200 }
201 });
202 • 2
203 this.j.setOnTouchListener(new OnTouchListener() {
204 public boolean onTouch(View var1, MotionEvent var2) {
205 switch(var2.getAction()) {
206 case 0:
207 case 1:
208 if(!var1.hasFocus()) {
209 var1.requestFocus();
210 }
211 default:
212 return false;
213 }
214 }
215 });
216 });
217 }
```

```

228 this.j.loadUrl(this.a);
229 this.j.setVisibility(4);
230 this.j.getSettings().setSavePassword(false);
231 this.l.a(new SecureJsInterface(), "SecureJsInterface");
232 SecureJsInterface.isPWDEdit = false;
233 super.setOnDismissListener(new OnDismissListener() {
234 public void onDismiss(DialogInterface var1) { 第3个！
235 try {
236 JniInterface.clearAllPWD(); 就是它！
237 } catch (Exception var3) {
238 ;
239 }
240 }
241 });
242 });

```

如上图，MAT中的引用链中的AuthDialog\$3指的就是这里的OnDismissListener匿名内部类！接着我们来看看Dialog.setOnDismissListener里面做了什么勾搭：

```

public void setOnDismissListener(final OnDismissListener listener) {
 if (mCancelAndDismissTaken != null) {
 throw new IllegalStateException(
 "OnDismissListener is already taken by "
 + mCancelAndDismissTaken + " and can not be replaced.");
 }
 if (listener != null) {
 mDismissMessage = mListenersHandler.obtainMessage(DISMISS, listener);
 } else {
 mDismissMessage = null;
 }
}

```

纳尼！OnDismissListener居然被赋给了Message.obj成员！

于是，我们心中生成的一条引用链是这样的：

```
Thread(main) -> MessageQueue->Message -> obj(OnDismissListener) -> AuthDialog -> Activity
```

```

62 public class AuthDialog extends Dialog {
63 private String a;
64 private AuthDialog.OnTimeListener b;
65 private IUiListener c;
66 private Handler d;
67 private FrameLayout e;
68 private LinearLayout f;
69 private FrameLayout g;
70 private ProgressBar h;
71 private String i;
72 private c j; LoginActivity
73 private Context k; private Context k;
74 private b l;
75 private boolean m = false;
76 private int n;

```

可是不对啊，我们所能找到的引用链跟 CookieSyncManager 子线程一点关系都没有！

再对比一下：

**GC Path:**

```

com.example.tribe.account.login.LoginActivity@42b67c68
|-com.example.connect.auth.AuthDialog@42f6b2d8
 |-com.example.connect.auth.AuthDialog$3@42cda2d0
 |-android.os.Message@42c74b70
 |-java.lang.Thread(CookieSyncManager)@42b6dad0

```

子线程 CookieSyncManager 拿到了主线程的 Message！！ Oh no !! 这是什么情况？？？这个 Message 被某处地方错误引用了？子线程通过 JNI 在 native 中拿到 Java 层的对象？

好吧，楼主承认研究了一个晚上没有任何进展。。。

## 六. 原来是它！ —Dialog

注：以下的分析感悟来自Github上面的一篇文章：[《一个内存泄漏引发的血案》](#)

这里简要说明一下，作者的结论是：在 Android Lollipop 之前使用 AlertDialog 可能会导致内存泄漏！

作者发现，局部变量的生命周期在 Dalvik VM 跟 ART/JVM 中有区别。在 DVM 中，假如线程死循环或者阻塞，那么线程栈帧中的局部变量假如没有被置为 null，那么就不会被回收。

如下代码使用阻塞队列说明问题：

```
static class MyMessage {
 final String message;
 MyMessage(String message) {
 this.message = message;
 }
}

static void startThread() {
 final BlockingQueue<MyMessage> queue = new LinkedBlockingQueue<>();
 MyMessage message = new MyMessage("Hello Leaking World");
 queue.offer(message);
 new Thread() {
 @Override public void run() {
 try {
 loop(queue);
 } catch (InterruptedException e) {
 throw new RuntimeException(e);
 }
 }
 }.start();
}

static void loop(BlockingQueue<MyMessage> queue) throws InterruptedException {
 while (true) {
 MyMessage message = queue.take();
 System.out.println("Received: " + message);
 }
}
```

子线程中调用 loop()死循环，不停地从阻塞队列中取出一个 MyMessage 对象并且将对象的引用赋值给局部变量 message，一次 while 循环之后，虚拟机应当结束 while 花括号中的局部变量的生命周期，并且释放对应的堆内存中的 MyMessage 对象。可是，DVM 没有这么做！！

在 VM 中，每一个栈帧都是本地变量的集合，而垃圾回收器是保守的：只要存在一个存活的引用，就不会回收它。在每次循环结束后，本地变量不再可访问，然而本地变量仍持有对 Message 的引用，interpreter/JIT 理论上应该在本地变量不可访问时将其引用置为 null，然而它们并没有这样做，引用仍然存活，而且不会被置为 null，使得它不会被回收！！

这种场景不就是 Android Handler 消息机制的处理方式么？！

```

0011: public static void loop() {
0012: final Looper me = myLooper();
0013: if (me == null) {
0014: throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
0015: }
0016: final MessageQueue queue = me.mQueue;
0017:
0018: // Make sure the identity of this thread is that of the local process,
0019: // and keep track of what that identity token actually is.
0020: Binder.clearCallingIdentity();
0021: final long ident = Binder.clearCallingIdentity();
0022:
0023: for (;;) {
0024: Message msg = queue.next(); // might block
0025: if (msg == null) {
0026: // No message indicates that the message queue is quitting.
0027: return;
0028: }
0029:
0030: msg.target.dispatchMessage(msg);
0031:
0032: msg.recycle();
0033: }
0034: } ? end loop ?
0035:

```

Looper 不停地从阻塞队列 MessageQueue 中取出下一条消息 Message 并将引用赋给本地变量 msg。一旦一次循环结束了，msg 没有被置为 null，对应的 Message 对象没有被回收，于是就泄漏了。

不过，Message 是自带回收机制的，而且任何线程共用，从上面源码可以看到，每个 Message 被 Handler 处理完之后都会 recycle()，置空所有的成员变量，并且放到回收池中。

好了，被 CookieSyncManager 子线程的 Looper 轮过一次的 Message 对象也跟其他人一样，被回收并放在了回收池中。这个时候，刚好遇到了 Dialog！！

```

public void setOnDismissListener(final OnDismissListener listener) {
 if (mCancelAndDismissTaken != null) {
 throw new IllegalStateException(
 "OnDismissListener is already taken by "
 + mCancelAndDismissTaken + " and can not be replaced.");
 }
 if (listener != null) {
 mDismissMessage = mListenerHandler.obtainMessage(DISMISS, listener);
 } else {
 mDismissMessage = null;
 }
}

```

这家伙刚刚好通过 obtainMessage() 从回收池中拿到了这个 Message (被 CookieSyncManager 线程的本地变量引用住了), 而且 Message.obj 变量就是 OnDismissListener。

拿到之后, Dialog 居然据为己有!! 作为一个成员宠爱着!

```

00082: public class Dialog implements DialogInterface, Window.Callback,
00083: KeyEvent.Callback, OnCreateContextMenuListener {
00084: private static final String TAG = "Dialog";
00085: private Activity mOwnerActivity;
00086:
00087: final Context mContext;
00088: final WindowManager m WindowManager;
00089: Window mWindow;
00090: View mDecor;
00091: private ActionBarImpl mActionBar;
00092: /**
00093: * This field should be made private, so it is hidden from the SDK.
00094: * {@hide}
00095: */
00096: protected boolean mCancelable = true;
00097:
00098: private String mCancelAndDismissTaken;
00099: private Message mCancelMessage;
00100: private Message mDismissMessage; ←
00101: private Message mShowMessage;
00000

```

Dialog 自从拥有了 mDismissMessage 对象之后就不会让它挂到消息队列中了, 每次要用都是拷贝一份而已。Message.obtain(mDismissMessage), 所以这个 Message 再也不会回到回收池中, 直到 Dialog 被销毁, mDismissMessage 变量也被置为 null 了。

```

00343: private void sendDismissMessage() {
00344: if (mDismissMessage != null) {
00345: // Obtain a new message so this dialog can be re-used
00346: Message.obtain(mDismissMessage).sendToTarget();
00347: }
00348: }
00349:

```

但是，这个 Message 依然占据着堆内存，而且被一个“游离”着的子线程局部变量 msg 引用着！！于是有了这条引用链：

```
Thread(CookieSyncManager) -> Message -> AuthDialog$3(OnDismissListener) -> AuthDialog
```

## 七. 总结一些注意点

针对 Android4.3 及以下版本，或者使用 DVM 的 Android 版本

- 使用 WebView 的时候，需要注意确保调用 destroy()
- 考虑是否使用 applicationContext() 来构建 WebView 实例
- 调用 Dialog 设置 OnShowListener、OnDismissListener、OnCancelListener 的时候，注意内部类是否泄漏 Activity 对象
- 尽量不要自己持有 Message 对象。

# Android内存泄露之Handler

来源:[www.jcodecraeer.com](http://www.jcodecraeer.com)

Handler也是造成内存泄露的一个重要的源头，主要Handler属于TLS(Thread Local Storage)变量,生命周期和Activity是不一致的，Handler引用Activity会存在内存泄露。

看一下如下代码

```
/*
 * 实现的主要功能。
 * @version 1.0.0
 * @author Abay Zhuang

 * Create at 2014-7-28
 */
public class HandlerActivity extends Activity {

 private final Handler mHandler = new Handler() {
 @Override
 public void handleMessage(Message msg) {
 // ...
 }
 };

 @Override
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 mHandler.sendMessageDelayed(Message.obtain(), 60000);

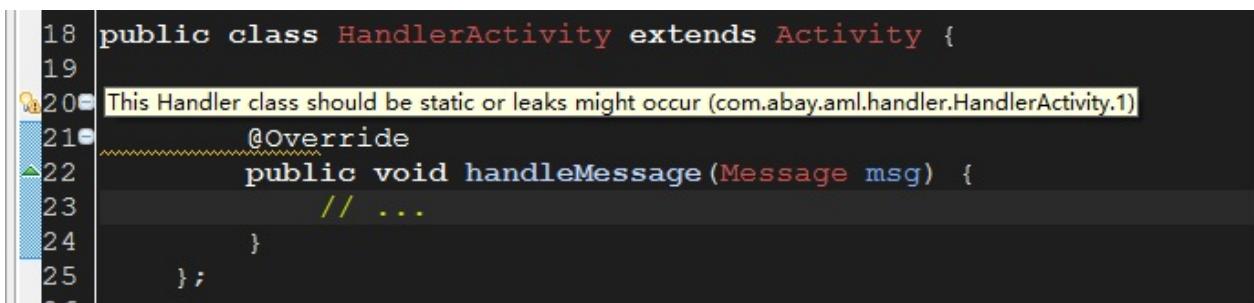
 //just finish this activity
 finish();
 }

}
```

是否您以前也是这样用的呢。

## 没有问题？

Eclipse 工具有这样的警告提示 警告:



```

18 public class HandlerActivity extends Activity {
19
20 This Handler class should be static or leaks might occur (com.abay.aml.handler.HandlerActivity.1)
21 @Override
22 public void handleMessage(Message msg) {
23 // ...
24 }
25 }

```

This Handler class should be static or leaks might occur  
 (com.example.ta.HandlerActivity.1), 意思： class 使用静态声明否者可能出现内存泄露。

## 为啥出现这样的问题呢

### Handler 的生命周期与Activity 不一致

- 当Android应用启动的时候，会先创建一个UI主线程的Looper对象，Looper实现了一个简单的消息队列，一个一个的处理里面的Message对象。主线程Looper对象在整个应用生命周期中存在。
- 当在主线程中初始化Handler时，该Handler和Looper的消息队列关联（没有关联会报错的）。发送到消息队列的Message会引用发送该消息的Handler对象，这样系统可以调用 Handler#handleMessage(Message) 来分发处理该消息。

### handler 引用 Activity 阻止了GC对Activity的回收

- 在Java中，非静态(匿名)内部类会默认隐性引用外部类对象。而静态内部类不会引用外部类对象。
- 如果外部类是Activity，则会引起Activity泄露。

当Activity finish后，延时消息会继续存在主线程消息队列中1分钟，然后处理消息。而该消息引用了Activity的Handler对象，然后这个Handler又引用了这个Activity。这些引用对象会保持到该消息被处理完，这样就导致该Activity对象无法被回收，从而导致了上面说的Activity泄露。

## 如何避免修？

- 使用显形的引用，1.静态内部类。2.外部类
- 使用弱引用：WeakReference

修改代码如下：

```
/*
 *
 * 实现的主要功能。
 *
 * @version 1.0.0
 * @author Abay Zhuang

 * Create at 2014-7-28
 */
public class HandlerActivity2 extends Activity {

 private static final int MESSAGE_1 = 1;
 private static final int MESSAGE_2 = 2;
 private static final int MESSAGE_3 = 3;
 private final Handler mHandler = new MyHandler(this);

 @Override
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 mHandler.sendMessageDelayed(Message.obtain(), 60000);

 // just finish this activity
 finish();
 }

 public void todo() {
 };

 private static class MyHandler extends Handler {
 private final WeakReference<HandlerActivity2> mActivity;

 public MyHandler(HandlerActivity2 activity) {
 mActivity = new WeakReference<HandlerActivity2>(activity);
 }

 @Override
 public void handleMessage(Message msg) {
 System.out.println(msg);
 if (mActivity.get() == null) {
 return;
 }
 mActivity.get().todo();
 }
 }
}
```

# 上面这样就可以了吗？

- 当Activity finish后 handler对象还是在Message中排队。还是会处理消息，这些处理有必要？
- 正常Activitiy finish后，已经没有必要对消息处理，那需要怎么做呢？

解决方案也很简单，在Activity onStop或者onDestroy的时候，取消掉该Handler对象的Message和Runnable。

通过查看Handler的API，它有几个方法：removeCallbacks(Runnable r)和removeMessages(int what)等。

代码如下：

```
/*
 * 一切都是为了不要让mHandler拖泥带水
 */
@Override
public void onDestroy() {
 mHandler.removeMessages(MESSAGE_1);
 mHandler.removeMessages(MESSAGE_2);
 mHandler.removeMessages(MESSAGE_3);

 // ...
 mHandler.removeCallbacks(mRunnable);

 // ...
}
```

如果上面觉得麻烦，也可以如下面：

```
@Override
public void onDestroy() {
 // If null, all callbacks and messages will be removed.
 mHandler.removeCallbacksAndMessages(null);
}
```

敬请期待下一章(^\_\_^) 嘻嘻.....

也可以关注[我的github](#)

# Android内存泄露之Thread

来源：[CSDN](#)

线程也是造成内存泄露的一个重要的源头。线程产生内存泄露的主要原因在于线程生命周期的不可控。

## 1.看一下下面是否存在问题

```
/*
 *
 * @version 1.0.0
 * @author Abay Zhuang

 * Create at 2014-7-17
 */
public class ThreadActivity extends Activity {
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 new MyThread().start();
 }

 private class MyThread extends Thread {
 @Override
 public void run() {
 super.run();
 dosomthing();
 }
 }
 private void dosomthing(){
 }
}
```

这段代码很平常也很简单，是我们经常使用的形式。

## 真的没有问题吗

我们思考一个问题：假设MyThread的run函数是一个很费时的操作，当我们开启该线程后，将设备的横屏变为了竖屏，一般情况下当屏幕转换时会重新创建Activity，按照我们的想法，老的Activity应该会被销毁才对，然而事实上并非如此。由于我们的线程是Activity

的内部类，所以MyThread中保存了Activity的一个引用，当MyThread的run函数没有结束时，MyThread是不会被销毁的，因此它所引用的老的Activity也不会被销毁，因此就出现了内存泄露的问题。

## 2.这种线程导致的内存泄露问题应该如何解决呢？

- 第一、将线程的内部类，改为静态内部类。
- 第二、在线程内部采用弱引用保存Context引用。代码如下：

```
/*
 *
 * @version 1.0.0
 * @author Abay Zhuang

 * Create at 2014-7-17
 */
public class ThreadAvoidActivity extends Activity {
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 new MyThread(this).start();
 }
 private void dosomthing() {
 }
 private static class MyThread extends Thread {
 WeakReference<ThreadAvoidActivity> mThreadActivityRef;
 public MyThread(ThreadAvoidActivity activity) {
 mThreadActivityRef = new WeakReference<ThreadAvoidActivity>(
 activity);
 }
 @Override
 public void run() {
 super.run();
 if (mThreadActivityRef == null)
 return;
 if (mThreadActivityRef.get() != null)
 mThreadActivityRef.get().dosomthing();
 // dosomthing
 }
 }
}
```

### 上面的两个步骤其实是切换两个对象的双向强引用链接

- 静态内部类：切断Activity对于MyThread的强引用。

- 弱引用：切断MyThread对于Activity 的强引用。

### 3.AsyncTask 内部类会如何呢？

有些人喜欢用Android提供的AsyncTask，但事实上AsyncTask的问题更加严重， Thread 只有在run函数不结束时才出现这种内存泄露问题，然而AsyncTask内部的实现机制是运用了ThreadPoolExecutor，该类产生的Thread对象的生命周期是不确定的，是应用程序无法控制的，因此如果AsyncTask作为Activity的内部类，就更容易出现内存泄露的问题。

代码如下：

```
/*
 *
 * 弱引用
 * @version 1.0.0
 * @author Abay Zhuang

 * Create at 2014-7-17
 */
public abstract class WeakAsyncTask<Params, Progress, Result, WeakTarget>
 extends AsyncTask<Params, Progress, Result> {
 protected WeakReference<WeakTarget> mTarget;
 public WeakAsyncTask(WeakTarget target) {
 mTarget = new WeakReference<WeakTarget>(target);
 }
 @Override
 protected final void onPreExecute() {
 final WeakTarget target = mTarget.get();
 if (target != null) {
 this.onPreExecute(target);
 }
 }
 @Override
 protected final Result doInBackground(Params... params) {
 final WeakTarget target = mTarget.get();
 if (target != null) {
 return this.doInBackground(target, params);
 } else {
 return null;
 }
 }
 @Override
 protected final void onPostExecute(Result result) {
 final WeakTarget target = mTarget.get();
 if (target != null) {
 this.onPostExecute(target, result);
 }
 }
 protected void onPreExecute(WeakTarget target) {
 // Nodefaultaction
 }
 protected abstract Result doInBackground(WeakTarget target,
 Params... params);
 protected void onPostExecute(WeakTarget target, Result result) {
 // Nodefaultaction
 }
}
```

# Android内存泄露之资源

来源:[CSDN](#)

资源内存泄露主要是资源申请未释放，还有资源没有重复使用。

- 第一种解决这部分问题的关键在于申请资源后能保证能释放资源
- 第二种利用复用机制优化,如池的概念

## 1.引用资源没有释放

代码如下：

```
private final class SettingsObserver implements Observer {
 public void update(Observable o, Object arg) {
 // todo ...
 }
}

ContentQueryMap.getInstance().addObserver(new SettingsObserver());
```

你觉得这段代码正常吗

答案是否定

存在的问题是没办法注销观察者对象（SettingsObserver），这样带来的问题是没办法释放该观察者。

那么这个对象将伴随整个单例生命周期，无形中就泄露一个SettingsObserver的内存。

### 1.1 注册未取消造成的内存泄露

这种Android的内存泄露比纯Java的内存泄露还要严重，

因为其他一些Android程序可能引用我们的Android程序的对象（比如注册机制）。

即使我们的Android程序已经结束了，但是别的引用程序仍然还有对我们的Android程序的某个对象的引用，泄露的内存依然不能被垃圾回收。

还是我们的对象周期不一致引起的。

例如：

- BroadcastReceiver对象注册

```
... has leaked IntentReceiver ... Are you missing a call to unregisterReceiver()?
```

- Observer对象

被观察对象生命周期和观察者的生命周期不一致 不观察的时候需要注销

## 1.2 集合中对象没清理造成的内存泄露

我们通常把一些对象的引用加入到了集合中，当我们不需要该对象时，并没有把它的引用从集合中清理掉，这样这个集合就会越来越大。

如果这个集合是static的话，那情况就更严重了

在add 情况也要记得 remove

## 2. 资源对象没关闭造成的内存泄露

资源性对象比如（Cursor, File文件等）往往都用了一些缓冲，我们在不使用的时候，应该及时关闭它们，以便它们的缓冲及时回收内存。它们的缓冲不仅存在于java虚拟机内，还存在于java虚拟机外。

如果我们仅仅是把它的引用设置为null,而不关闭它们，往往会造成内存泄露。因为有些资源性对象，比如SQLiteDatabase（在析构函数finalize () ,如果我们没有关闭它，它自己会调close()关闭），如果我们没有关闭它，系统在回收它时也会关闭它，但是这样的效率太低了。因此对于资源性对象在不使用的时候，应该调用它的close()函数，将其关闭掉，然后才置为null.

在我们的程序退出时一定要确保我们的资源性对象已经关闭。

程序中经常会进行查询数据库的操作，但是经常会有使用完毕Cursor后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在常时间大量操作的情况下才会复现内存问题，这样就会给以后的测试和问题排查带来困难和风险

代码如下：

```
try {
 Cursor c = queryCursor();
 int a = c.getInt(1);

 // 如果出错,后面的cursor.close()将不会执行

 c.close();
} catch (Exception e) {
}
```

合理的写法：

```
Cursor c;
try {
 c = queryCursor();
 int a = c.getInt(1);

 // 如果出错,后面的cursor.close()将不会执行
 //c.close();
} catch (Exception e) {
} finally{
 if (c != null) {
 c.close();
 }
}
```

### ## 3.一些不良代码造成的内存压力

有些代码并不造成内存泄露，但是它们， 或是对没使用的内存没进行有效及时的释放，或是没有有效的利用已有

- > 如何优化呢？
- >
- > \* 频繁的申请内存对象和和释放对象，可以考虑Pool 池。
- > \* 多线程可以考虑线程池
- > \* 频繁的申请对象释放对象。可以考虑对象池。 例如：AbsListView 中RecycleBin 类是view对象池
- > \* 频繁的链接资源和释放资源。可以考虑链接资源池。

#### ### 3.1 Bitmap没调用recycle()

Bitmap对象在不使用时，我们应该先调用recycle()释放内存，然后才它设置为null. 虽然recycle()从源码上看

#### ### 3.2 构造Adapter时，没有使用缓存的 convertView

以构造ListView的BaseAdapter为例，在BaseAdapter中提供了方法：

```
public View getView(int position, View convertView, ViewGroup parent)
```

来向ListView提供每一个item所需要的view对象。初始时ListView会从BaseAdapter中根据当前的屏幕布局实

如下图：



由上图可以看出，如果我们不去使用convertView，而是每次都在getView()中重新实例化一个View对象的话，即

Listview回收list item的view对象的过程可以查看：

android.widget.AbsListView.java --> obtainView 方法。

在使用过程中java代码如下：

```
@Override public View getView(int position, View convertView, ViewGroup parent) {
 Log.d(TAG, "Position:" + position + "___"
```

```
 + String.valueOf(System.currentTimeMillis()));
 ViewHolder holder;
 if (convertView == null) {
 final LayoutInflater inflater = (LayoutInflater) mContext
 .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
 convertView = inflater.inflate(R.layout.list_item_icon_text, null);
 holder = new ViewHolder();
 holder.icon = (ImageView) convertView.findViewById(R.id.icon);
 holder.text = (TextView) convertView.findViewById(R.id.text);
 convertView.setTag(holder);
 } else {
 holder = (ViewHolder) convertView.getTag();
 }
 holder.icon.setImageDrawable(mDefaultDrawable);
 holder.text.setText(mData[position]);
 return convertView;
```

}

```
static class ViewHolder { ImageView icon; TextView text; }`
```

## 优化点

- 1.复用convertView,convertView在ListView有RecycleBin的对象池维护。
- 2.ViewHolder出现减少findViewById 的调用

代码位置：

<https://github.com/loyabe/Docs/tree/master/%E5%86%85%E5%AD%98%E6%B3%84%E9%9C%B2>

敬请期待下一章(^\_\_^) 嘻嘻.....

# Android内存泄露开篇

来源:[CSDN](#)

## 先来想这三个问题

1. 内存泄露是怎么回事
2. 内存会泄露的原因
3. 避免内存泄露

## 1.内存泄露怎么回事

一个程序中，已经不需要使用某个对象，但是因为仍然有引用指向它垃圾回收器就无法回收它，当然该对象占用的内存就无法被使用，这就造成了内存泄露。

Android的一个应用程序的内存泄露对别的应用程序影响不大。

为了能够使得Android应用程序安全且快速的运行，Android的每个应用程序都会使用一个专有的Dalvik虚拟机实例来运行，它是由Zygote服务进程孵化出来的，也就是说每个应用程序都是在属于自己的进程中运行的。

Android为不同类型的进程分配了不同的内存使用上限，如果程序在运行过程中出现了内存泄漏而造成应用进程使用的内存超过了这个上限，则会被系统视为内存泄漏，从而被kill掉，这使得仅仅自己的进程被kill掉，而不会影响其他进程（如果是system\_process等系统进程出问题的话，则会引起系统重启）

## 2.出现内存泄露原因

### 1.资源对象没关闭造成的内存泄露

资源性对象比如（Cursor，File文件等）往往都用了一些缓冲，我们在不使用的时候，应该及时关闭它们，以便它们的缓冲及时回收内存

### 2.变量的作用域不一样导致

| 变量                      | 作用域   |
|-------------------------|-------|
| 函数变量                    | 函数内   |
| 成员变量                    | 整个对象内 |
| TLS(ThreadLocalStorage) | 整个线程  |
| 静态变量                    | 整个进场内 |
| Binder (IPC)            | 进程间   |

因为作用域的不同，作用域大引用到对象都可能不会马上销毁，所以会内存泄露。

handle 的内存泄露主要 TLS变量和 activity的生命周期不一样，。

Thread 引用其他对象也容易出现对象泄露。

### 3. 内存压力过大

- 1.图片资源加载过多，超过内存使用空间，例如Bitmap 的使用
- 2.重复创建view，没有重复使用 listview，的使用

### 3. 如何避免内存泄露

- 1.良好的代码规范，清晰代码逻辑
- 2.对于引用生命不一样的对象，可以用弱引用WeakReferer
- 3.对于资源对象 使用finally 强制关闭
- 4.内存压力过大就要统一的管理内存
- 5.对象重复并且频繁调用可以考虑对象池。

# GC Root

来源:[www.xuebuyuan.com](http://www.xuebuyuan.com)

常说的**GC(Garbage Collector) roots**, 特指的是垃圾收集器 (*Garbage Collector*) 的对象, GC会收集那些不是GC roots且没有被GC roots引用的对象。

一个对象可以属于多个root, GC root有几下种:

- **Class** - 由系统类加载器(system class loader)加载的对象, 这些类是不能够被回收的, 他们可以以静态字段的方式保存持有其它对象。我们需要注意的一点就是, 通过用户自定义的类加载器加载的类, 除非相应的 `java.lang.Class` 实例以其它的某种(或多种)方式成为roots, 否则它们并不是roots.
- **Thread** - 活着的线程
- **Stack Local** - Java方法的local变量或参数
- **JNI Local** - JNI方法的local变量或参数
- **JNI Global** - 全局JNI引用
- **Monitor Used** - 用于同步的监控对象
- **Held by JVM** - 用于JVM特殊目的由GC保留的对象, 但实际上这个与JVM的实现是有关的。可能已知的一些类型是: 系统类加载器、一些JVM知道的重要的异常类、一些用于处理异常的预分配对象以及一些自定义的类加载器等。然而, **JVM并没有为这些对象提供其它的信息**, 因此就只有留给分析分员去确定哪些是属于"JVM持有"的了。

以下是一张由Java Profiler的标示出哪些是GC roots的示例图:



译自: [http://www.yourkit.com/docs/80/help/gc\\_roots.jsp](http://www.yourkit.com/docs/80/help/gc_roots.jsp)

# 一个内存泄漏引发的血案-Square

- 原文链接：[A small leak will sink a great ship](#)
- 原文作者：[Pierre-Yves Ricau](#)
- 译文出自：[开发技术前线 www.devtf.cn](#)。未经允许，不得转载！
- 译者：[chaossss](#)
- 校对者：[这里校对者的github用户名](#)
- 状态：完成

在开发 LeakCanary 时我发现一处奇怪的内存泄漏，为了搞清楚到底是什么原因导致这个问题我一边 Debug，一边在邮件中和小伙伴们沟通，最后形成了这篇博文。

嫌弃篇幅太长懒得看的，概述在此：在 Android Lollipop 之前使用 AlertDialog 可能会导致内存泄漏。

## The Artist

LeakCanary 通知我存在内存泄漏：

```
* GC ROOT thread com.squareup.picasso.Dispatcher.DispatcherThread.<Java Local>
* references android.os.Message.obj
* references com.example.MainActivity$MyDialogClickListener.this$0
* leaks com.example.MainActivity instance
```

简单来说就是：一个 Picasso 线程正在站内持有一个 Message 实例的本地变量，而 Message 持有 DialogInterface.OnClickListener 的引用，而 DialogInterface.OnClickListener 又持有一个被销毁 Activity 的引用。

本地变量通常由于他们仅存在于栈内存活时间较短，当线程调用某个方法，系统就会为其分配栈帧。当方法返回，栈帧也会随之被销毁，栈内所有本地变量都会被回收。如果本地变量导致了内存泄漏，一般意味着线程要么死循环，要么阻塞了，而且线程在这种状态时持有 Message 实例的引用。

于是 Dimitris 和我都去 Picasso 源码中一探究竟：

Dispatcher.DispatcherThread 是一个简单的 HandlerThread：

```
static class DispatcherThread extends HandlerThread {
 DispatcherThread() {
 super(Utils.THREAD_PREFIX + DISPATCHER_THREAD_NAME, THREAD_PRIORITY_BACKGROUND);
 }
}
```

这个线程用标准的方式通过 Handler 接收 Message：

```
private static class DispatcherHandler extends Handler {
 private final Dispatcher dispatcher;

 public DispatcherHandler(Looper looper, Dispatcher dispatcher) {
 super(looper);
 this.dispatcher = dispatcher;
 }

 @Override public void handleMessage(final Message msg) {
 switch (msg.what) {
 case REQUEST_SUBMIT: {
 Action action = (Action) msg.obj;
 dispatcher.performSubmit(action);
 break;
 }
 // ... handles other types of messages
 }
 }
}
```

显然 `Dispatcher.DispatcherHandler.handleMessage()` 里面没有明显会让本地变量持有 `Message` 引用的 Bug。

## Queue Tips

Let's look at how HandlerThread works:

后来越来越多内存泄漏的通知出现了，这些通知不仅仅来自 Picasso，各种各样线程中的本地变量都存在内存泄漏，而且这些内存泄漏往往和 Dialog 的 OnClickListener 有关。发生内存泄漏的线程有一个共同的特性：他们都是工作者线程，而且通过某种阻塞队列接收各自的工作。

```

for (;;) {
 Message msg = queue.next(); // might block
 if (msg == null) {
 return;
 }
 msg.target.dispatchMessage(msg);
 msg.recycleUnchecked();
}

```

通过源码可以发现，肯定存在本地变量持有 `Message` 的引用，然而它的生命周期本应很短，而且在循环结束时被清除。

我们尝试通过利用阻塞队列实现一个简单的工作者线程来重现这个 Bug，它只发送一个 `Message`：

```

static class MyMessage {
 final String message;
 MyMessage(String message) {
 this.message = message;
 }
}

static void startThread() {
 final BlockingQueue<MyMessage> queue = new LinkedBlockingQueue<>();
 MyMessage message = new MyMessage("Hello Leaking World");
 queue.offer(message);
 new Thread() {
 @Override public void run() {
 try {
 loop(queue);
 } catch (InterruptedException e) {
 throw new RuntimeException(e);
 }
 }
 }.start();
}

static void loop(BlockingQueue<MyMessage> queue) throws InterruptedException {
 while (true) {
 MyMessage message = queue.take();
 System.out.println("Received: " + message);
 }
}

```

一旦 `Message` 被打印到 Log 中，`MyMessage` 实例应该被回收，然而还是发生了内存泄漏：

- GC ROOT thread com.example.MyActivity\$2. (named 'Thread-110')
- leaks com.example.MyActivity\$MyMessage instance

我们发送新的 Message 到阻塞队列的瞬间，前一个 Message 就被回收，而新的 Message 就泄漏了。

在 VM 中，每一个栈帧都是本地变量的集合，而垃圾回收器是保守的：只要存在一个存活的引用，就不会回收它。

在循环结束后，本地变量不再可访问，然而本地变量仍持有对 Message 的引用，interpreter/JIT 理论上应该在本地变量不可访问时将其引用置为 null，然而它们并没有这样做，引用仍然存活，而且不会被置为 null，使得它不会被回收。

为了验证我们的结论，我们手动将引用设为 null，并打印它，使得 null 不会是最优化办法：

```
static void loop(BlockingQueue<MyMessage> queue) throws InterruptedException {
 while (true) {
 MyMessage message = queue.take();
 System.out.println("Received: " + message);
 message = null;
 System.out.println("Now null: " + message);
 }
}
```

在测试上面的代码时，我们发现 MyMessage 实例在 Message 被设为 null 时立刻被回收。也就是说我们的结论似乎是正确的。

因为这样的内存泄漏会在各种各样的线程和阻塞队列的实现中发生，我们现在确定这是一个存在于 VM 中的 Bug。基于这个结论，我们只能在 Dalvik VM 中复现这个 Bug，在 ART VM 或 JVM 中则无法复现。

## Message in a (recycled) bottle

我们发现了一个会导致内存泄漏的 Bug，但这会导致严重的内存泄漏吗？不妨看看我们最初的泄漏信息：

```
* GC ROOT thread com.squareup.picasso.Dispatcher.DispatcherThread.<Java Local>
* references android.os.Message.obj
* references com.example.MyActivity$MyDialogClickListener.this$0
* leaks com.example.MyActivity.MainActivity instance
```

我们发送给 Picasso Dispatcher 线程的 Message，我们从未将 Message.obj 设为 DialogInterface.OnClickListener，那它是怎么结束的？

此外，当 Message 被处理，它应该立刻被回收，而且 Message.obj 应该被设为 null。只有那样 HandlerThread 才会等待下一个 Message，并暂时泄漏前一个 Message：

```
for (;;) {
 Message msg = queue.next(); // might block
 if (msg == null) {
 return;
 }
 msg.target.dispatchMessage(msg);
 msg.recycleUnchecked();
}
```

因而我们知道泄漏的 Message 会被回收，因此不会持有之前的内容。

一旦被回收，Message 就会回到常量池中：

```

void recycleUnchecked() {
 // Mark the message as in use while it remains in the recycled object pool.
 // Clear out all other details.
 flags = FLAG_IN_USE;
 what = 0;
 arg1 = 0;
 arg2 = 0;
 obj = null;
 replyTo = null;
 sendingUid = -1;
 when = 0;
 target = null;
 callback = null;
 data = null;

 synchronized (sPoolSync) {
 if (sPoolSize < MAX_POOL_SIZE) {
 next = sPool;
 sPool = this;
 sPoolSize++;
 }
 }
}
}

```

我们此时拥有一个泄漏的空 Message，它可能会被重用，并填充不同的内容。Message 常常以相同的方式被使用：在池中被调用，填充内容，放入 MessageQueue，然后被处理，最后被回收，并置回到池中。

它理应在很长一段时间内不会持有它的内容，那我们为什么总会在 DialogInterface.OnClickListener 实例中发生内存泄漏呢？

## Alert Dialogs

我们先创建一个简单的 AlertDialog：

```

new AlertDialog.Builder(this)
 .setPositiveButton("Baguette", new DialogInterface.OnClickListener() {
 @Override public void onClick(DialogInterface dialog, int which) {
 MyActivity.this.makeBread();
 }
 })
 .show();

```

注意到 ClickListener 持有 Activity 的引用，该匿名内部类实际完成的工作与下面的代码是一样的：

```
// First anonymous class of MyActivity.
class MyActivity$0 implements DialogInterface.OnClickListener {
 final MyActivity this$0;
 MyActivity$0(MyActivity this$0) {
 this.this$0 = this$0;
 }
 @Override public void onClick(DialogInterface dialog, int which) {
 this$0.makeBread();
 }
}

new AlertDialog.Builder(this)
 .setPositiveButton("Baguette", new MyActivity$0(this));
 .show();
Internally, AlertDialog delegates the work to AlertController:

/**
 * Sets a click listener or a message to be sent when the button is clicked.
 * You only need to pass one of {@code listener} or {@code msg}.
 */
public void setButton(int whichButton, CharSequence text,
 DialogInterface.OnClickListener listener, Message msg) {
 if (msg == null && listener != null) {
 msg = mHandler.obtainMessage(whichButton, listener);
 }
 switch (whichButton) {
 case DialogInterface.BUTTON_POSITIVE:
 mButtonPositiveText = text;
 mButtonPositiveMessage = msg;
 break;
 case DialogInterface.BUTTON_NEGATIVE:
 mButtonNegativeText = text;
 mButtonNegativeMessage = msg;
 break;
 case DialogInterface.BUTTON_NEUTRAL:
 mButtonNeutralText = text;
 mButtonNeutralMessage = msg;
 break;
 }
}
```

所以 OnClickListener 被包裹到 Message 中，并被设置到 AlertController.mButtonPositiveMessage，现在我们看看该 Message 在什么时候被使用：

```

private final View.OnClickListener mButtonHandler = new View.OnClickListener() {
 @Override public void onClick(View v) {
 final Message m;
 if (v == mButtonPositive && mButtonPositiveMessage != null) {
 m = Message.obtain(mButtonPositiveMessage);
 } else if (v == mButtonNegative && mButtonNegativeMessage != null) {
 m = Message.obtain(mButtonNegativeMessage);
 } else if (v == mButtonNeutral && mButtonNeutralMessage != null) {
 m = Message.obtain(mButtonNeutralMessage);
 } else {
 m = null;
 }
 if (m != null) {
 m.sendToTarget();
 }
 // Post a message so we dismiss after the above handlers are executed.
 mHandler.obtainMessage(ButtonHandler.MSG_DISMISS_DIALOG, mDialogInterface)
 .sendToTarget();
 }
};

```

注意这： `m = Message.obtain(mButtonPositiveMessage);`

Message 被克隆，也就是说后面使用的是它的拷贝。这就以为着原始的 Message 从没有被发送，因此不会被回收，所以永久保存着它的内容，直到发生垃圾回收。

现在假设 Message 已经由于 HandlerThread 本地引用在被回收池调用之前发生内存泄漏，Dialog 最终会被垃圾回收，并释放由 mButtonPositiveMessage 持有的 Message 的引用。

然而，由于 Message 已经泄漏，它并不会被垃圾回收。同样的，它持有的内容也不会被回收，而 OnClickListener 持有对 Activity 的引用，导致 Activity 不会被回收。

## Smoking Gun

我们能证明这个结论么？

我们需要发送一个 Message 到 HandlerThread 中，让他被处理和回收，并不要再发送任何 Message 到那个线程中，使最后一个 Message 发生内存泄漏。然后，我们需要显示一个带 Button 的 Dialog，并希望它会从池中获取到相同的 Message。这确实很可能会发生，因为一旦被回收，Message 就会变成池内的第一个可调用的 Message。

```

HandlerThread background = new HandlerThread("BackgroundThread");
background.start();
Handler backgroundhandler = new Handler(background.getLooper());
final DialogInterface.OnClickListener clickListener = new DialogInterface.OnClickListener() {
 @Override public void onClick(DialogInterface dialog, int which) {
 MyActivity.this.makeCroissants();
 }
};
backgroundhandler.post(new Runnable() {
 @Override public void run() {
 runOnUiThread(new Runnable() {
 @Override public void run() {
 new AlertDialog.Builder(MyActivity.this) //
 .setPositiveButton("Baguette", clickListener) //
 .show();
 }
 });
 }
});

```

如果我们运行上面的代码，然后旋转屏幕销毁当前 Activity，就很有可能会使该 Activity 发生内存泄漏。

LeakCanary 准确地检测到了内存泄漏：

```

* GC ROOT thread android.os.HandlerThread.<Java Local> (named 'BackgroundThread')
* references android.os.Message.obj
* references com.example.MyActivity$1.this$0 (anonymous class implements android.con
* leaks com.example.MyActivity instance

```

现在我们成功地重现了这个 Bug，不妨看看该怎么修复它。

## The Startup Fix

只支持 ART VM 的设备当然不会有这个 Bug，当然，也没多少人用.....

## The Won't Fix

你可能会觉得这些内存泄漏没啥影响，而且你有其他更值得做的事情，或许更简单的内存泄漏需要修复。LeakCanary 默认无视了所有 Message 泄漏。但要注意的是，Activity 持有其整个 View 的资源，那可是有好几兆的。

## The App fix

确保 DialogInterface.OnClickListener 不会持有对 Activity 实例的强引用，例如在 Dialog 退出后清除对 Listener 的引用，下面是一个简化它的包裹类：

```
public final class DetachableClickListener implements DialogInterface.OnClickListener {

 public static DetachableClickListener wrap(DialogInterface.OnClickListener delegate) {
 return new DetachableClickListener(delegate);
 }

 private DialogInterface.OnClickListener delegateOrNull;

 private DetachableClickListener(DialogInterface.OnClickListener delegate) {
 this.delegateOrNull = delegate;
 }

 @Override public void onClick(DialogInterface dialog, int which) {
 if (delegateOrNull != null) {
 delegateOrNull.onClick(dialog, which);
 }
 }

 public void clearOnDetach(DialogInterface dialog) {
 dialog.getWindow()
 .getDecorView()
 .getViewTreeObserver()
 .addOnWindowAttachListener(new OnWindowAttachListener() {
 @Override public void onWindowAttached() { }
 @Override public void onWindowDetached() {
 delegateOrNull = null;
 }
 });
 }
}
```

然后你可以包裹所有 OnClickListener 实例：

```

 DetachableClickListener clickListener = wrap(new DialogInterface.OnClickListener() {
 @Override public void onClick(DialogInterface dialog, int which) {
 MyActivity.this.makeCroissants();
 }
 });

 AlertDialog dialog = new AlertDialog.Builder(this) //
 .setPositiveButton("Baguette", clickListener) //
 .create();
 clickListener.clearOnDetach(dialog);
 dialog.show();

```

## The Plumber's Fix

在一个常用的基础清晰你的工作者线程：当 Handler 闲置就向它发送空 Message，以确保不会发生 Message 的内存泄漏。

```

static void flushStackLocalLeaks(Looper looper) {
 final Handler handler = new Handler(looper);
 handler.post(new Runnable() {
 @Override public void run() {
 Looper.myQueue().addIdleHandler(new MessageQueue.IdleHandler() {
 @Override public boolean queueIdle() {
 handler.sendMessageDelayed(handler.obtainMessage(), 1000);
 return true;
 }
 });
 }
 });
}

```

在使用库时这个办法很好似用，因为你不能控制开发者在 Dialog 中写的代码。我们在 Picasso 中就用这个办法解决了这个 Bug。

## Conclusion

Many thanks to [Josh Humphries](#), [Jesse Wilson](#), [Manik Surtani](#), and [Wouter Coekaerts](#) for their help in our internal email thread.

正如我们所见，一个小小的，难以发现的 VM 行为会导致内存泄漏，而这又会导致大量内存被泄漏，最终使 App 崩溃。

感谢 [Josh Humphries](#), [Jesse Wilson](#), [Manik Surtani](#), 和 [Wouter Coekaerts](#) 在邮件沟通中帮助我解决了这个 Bug。



# Android使用NDK-STACK来恢复Cocos2d-x错误堆栈信息

来源:blog.csdn.net

之前在网上偶然看到可以使用ndk-stack来恢复cocos2d-x的错误堆栈信息，这虽不能算是通过Android调试C++代码，可还是能定位到具体哪一行代码出错，这样比起之前用输出log来定位错误来说，已经算是是从地狱走到天堂的大门了。

好了，下面是具体操作，其实很简单。

1.首先把logcat中的错误信息复制保存在文本中作为ndk-stack的输入，比如H:\logcat.txt。复制信息时要从有一行\*号的信息开始，即从

开始复制全部绿色的错误信息，然后粘贴在H:\logcat.txt中。

2 打开cmd 输入 \$NDK/ndk-stack -sym \$PROJECT PATH/obj/local/armeabi -dump H:logcat.txt

-dump选项将指定logcat保存为文件作为输入：

\$NDK是NDK安裝目錄..

\$PROJECT\_PATH是项目路径

3.最后是显示出C++的错误堆栈信息，从中可以找出错误代码的具体行号。

# JNI与原生代码通信

## 什么是JNI?

JNI: Java Native Interface

## JNI的能力

- 允许Java类的默写方法原生实现
- 能够像普通Java方法一样调用原生方法
- 原生方法可以使用Java对象，使用方法与java代码使用对象的方法一致
- 原生方法可以创建新的Java对象或者使用Java应用程序创建的对象

## JNI开发流程 - 示例(ndk的hello-jni示例)

### 原生方法声明

使用 `native` 关键字声明一个native方法，但是不能有方法体(因为方法需要让jni层实现)

```
/* 原生方法由'hello-jni'原生库实现
 *
 * 'hello-jni'，该原生库与本应用程序一起打包
 */
public native String stringFromJNI();
```

此时还不能直接调用这个方法，因为虚拟机不知道到哪里去找这个方法的实现

### 加载共享库

原生方法被编译成一个共享库。需要先加载共享库，虚拟机才能找到原生方法的实现。`java.lang.System`类提供了两个静态方法，`load` 和 `loadLibrary`，用于在运行时加载共享库。

```
/*
 * 这段代码用于在应用程序启动的时候加载'hello-jni'库
 * 该库在安装时由包管理器解压到/data/data/包名/lib/libhello-jni.so中
 */
static{
 System.loadLibrary("hello-jni");
}
```

Java的设计目标是平台独立，作为Java框架API的一部分，`loadLibrary`也要保持平台独立性。尽管Android NDK生成的实际共享库被命名为`libhello-jni.so`，但是`loadLibrary`方法只能使用`hello-jni`这个库名，在按照所使用的具体操作系统加上必要的前缀或者后缀。库名与`Android.mk`文件中使用`LOCAL_MODULE`构建系统变量定义的模块名一致。

`loadLibrary`的参数也不包含共享库的位置。Java库路径，也就是系统属性`java.library.path`保存`loadLibrary`方法在共享库搜索的目录列表，Android上的Java库路径包含`/vendor/lib`和`/system/lib`。

需要强调的是，`loadLibrary`在扫描Java库路径时，一旦返现同名的库，立即加载共享库。因为Java库路径的第一组目录是Android系统目录，为了避免与系统库命名冲突，强烈建议开发人员为每个共享库选择唯一的名字。

## 实现原生方法

### C/C++头文件生成器:javah

让原生函数名以及参数列表和Java类文件的原始定义一致是繁杂而多余的，因为JDK自带一个名为`javah`的命令行工具来执行任务，`javah`工具可以为原生方法解析Java类文件并生成由原生方法声明组成的头文件。

- 在命令行方式下运行

将当前工作目录改为`HelloJni`项目的导入目录，即`<Eclispe Workspace>/HelloJni`，`javah`工具对编译过的Java类文件进行操作，用编译过的类文件所在的位置和要解析的Java类名为参数调用`javah`，命令格式如下：

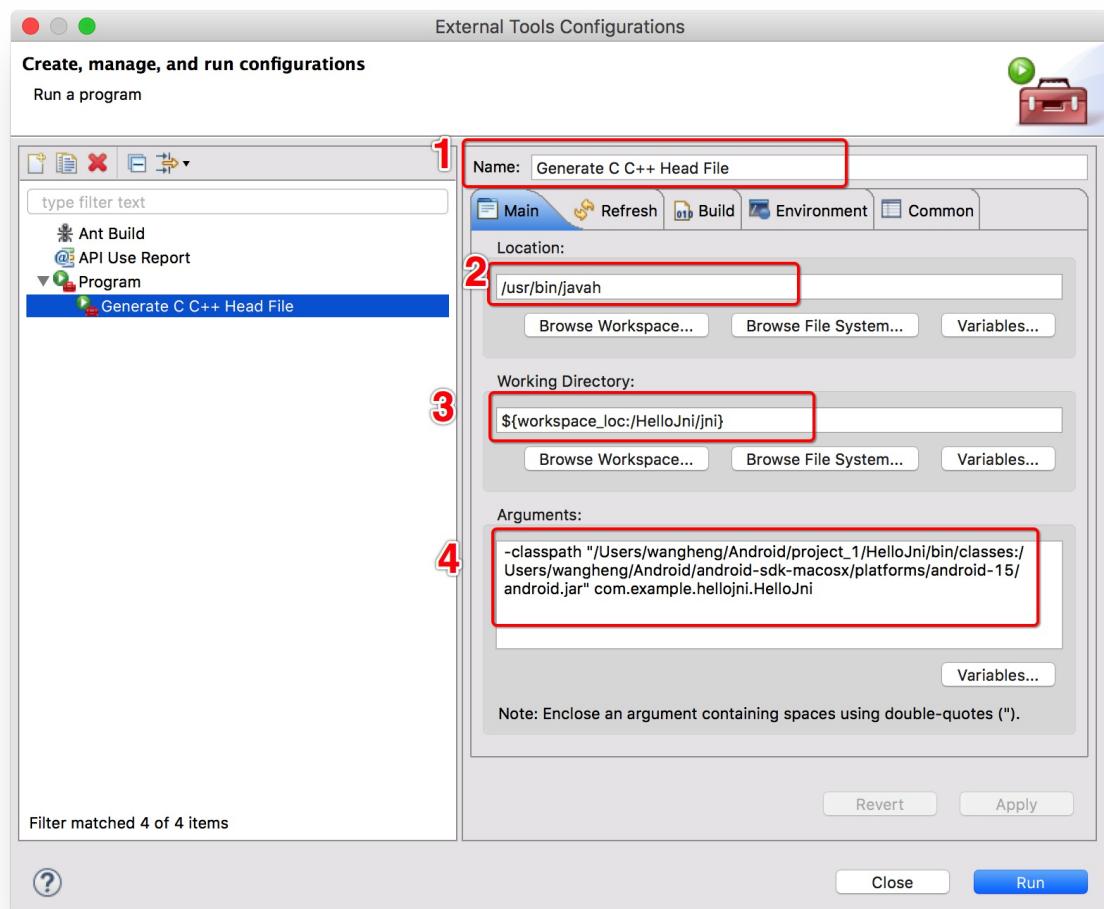
```
javah -classpath bin/classes com.example.hello.HelloJni
```

`javah`工具讲解析`com.example.hellojni.HelloJni`类文件，且生成`com_example_hellojni_HelloJni.h`的C/C++头文件，生成的方法签名内容如下：

```
Java_com_example_hellojni_HelloJni_stringFromJNI(JNIEnv*, jobject)
```

- 在Eclipse中运行

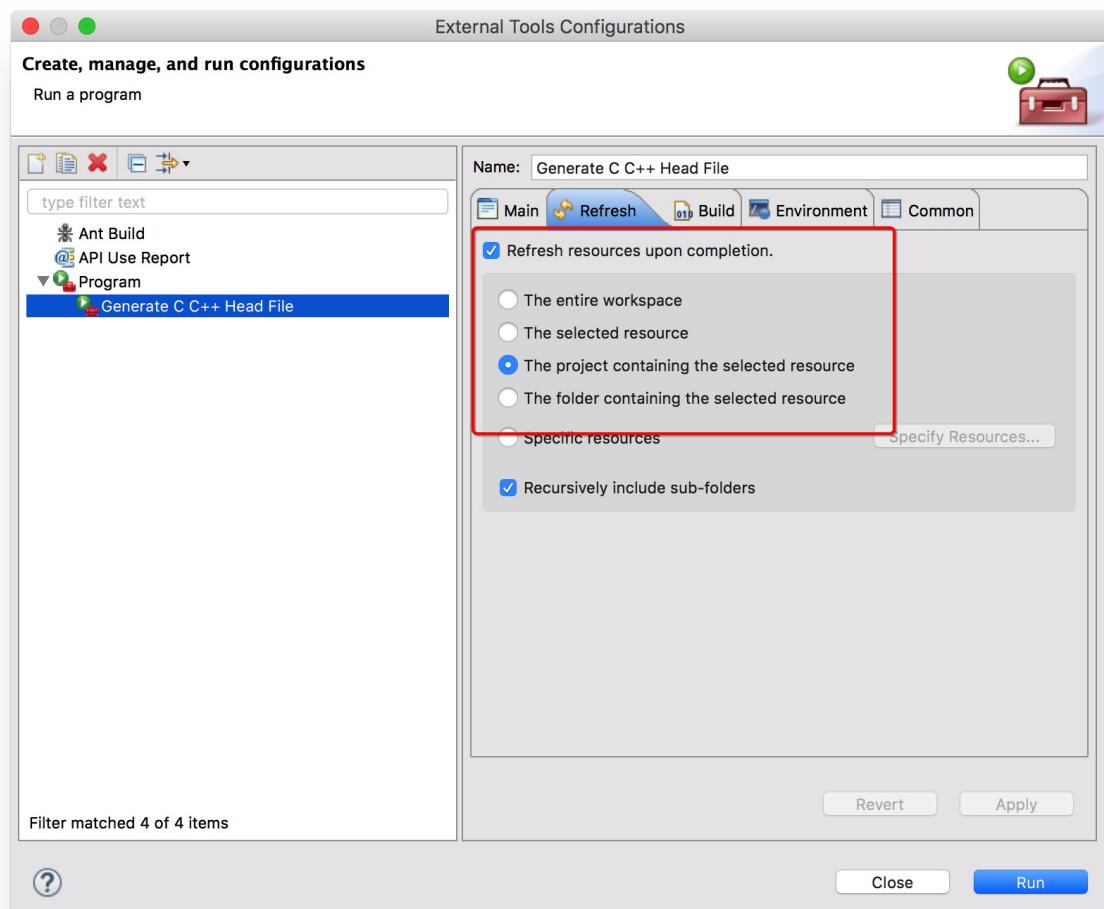
打开Eclipse,菜单栏选择Run->External Tools -> External Tools Configurations. 在打开的 External Tools Configurations 对话框中选择Program, 单击 New launch configuration 按钮, 单击Main选项卡, 按照下面的内容进行填写:



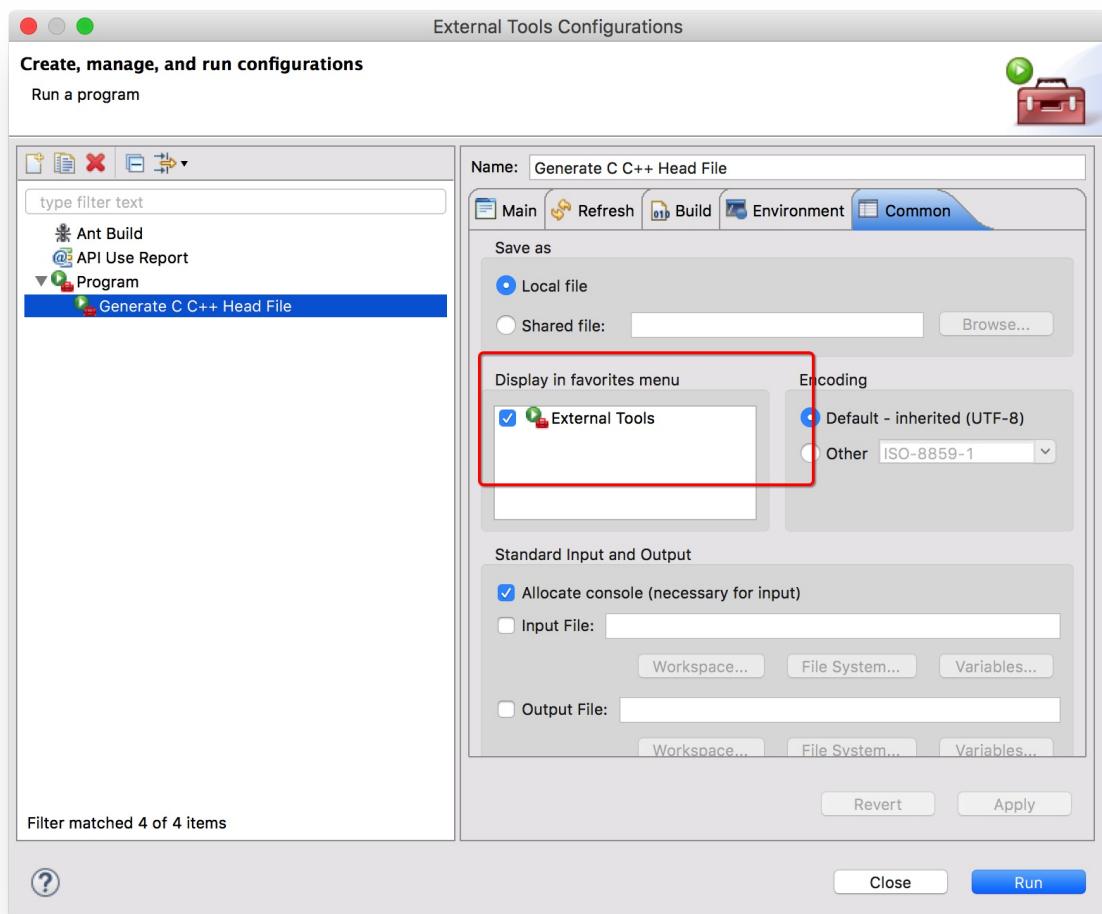
```
// MacOS / Linux
-classpath "${project_classpath}:${env_var:ANDROID_SDK_HOME}/platforms/android-15/and

// Windows
-classpath "${project_classpath};${env_var:ANDROID_SDK_HOME}/platforms/android-15/and
```

切换到Refresh选项卡，勾选 Refresh resource upon completion，并在列表中选择 The project containint the selected resource



切换到Common选项卡，选中 Display in favorites menu 组下的复选框 External Tools



点击OK保存配置。

macos下配置Android SDK和Android NDK环境变量

```
// 配置SDK
echo export ANDROID_SDK_HOME=/Users/wangheng/Android/android-sdk-macosx >> ~/.bash_profile
echo export PATH=$ANDROID_SDK_HOME/tools:$ANDROID_SDK_HOME/platform-tools:$PATH >> ~/.bash_profile

// 配置NDK
echo export ANDROID_NDK_HOME=/Users/wangheng/Android/android-ndk-r10e >> ~/.bash_profile
echo export PATH=$ANDROID_NDK_HOME:$PATH >> ~/.bash_profile
```

运行新创建的这个Program,即可生成指定.h文件

## 方法声明

尽管java方法 `stringFromJNI` 不带任何参数，但是原生方法带有两个参数：

```
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_stringFromJNI
 (JNIEnv *, jobject);
```

第一个参数 `JNIEnv` 是指向可用JNI函数表的接口指针； 第二个参数 `jobject` 是HelloJni类的Java对象引用

- `JNIEnv`接口指针

原生代码通过 `JNIEnv` 接口指针提供的各种函数来使用虚拟机功能。`JNIEnv` 是要给指向线程-局部数据的指针，而线程-局部数据中包含指向函数表的指针。实现原生方法的函数讲 `JNIEnv` 接口指针作为它们的第一个参数。

注意，传递给每一个原生方法调用的`JNIEnv`接口指针在与方法调用相关的线程中也有效，但是它不能被缓存以及被其他线程使用

原生代码是C与原生代码是C++其调用JNI函数的语法不通。C语言中，`JNIEnv` 是指向 `JNINativeInterface` 结构的指针，为了访问任何一个JNI函数，该指针需要首先被解引用。因为C代码的JNI函数不了解当前的JNI环境，`JNIEnv` 实例应该作为第一个参数传递给每一个JNI函数，格式如下：

```
return (*env)->NewStringUTF(env, "Hello from JNI!!!!");
```

在C++中，`JNIEnv` 实际上是C++类实例，JNI函数以成员函数的形式存在。因为JNI方法已经访问了当前JNI环境，因此JNI方法调用不要求`JNIEnv`实例作为参数，在C++中，完成同样功能的函数调用格式：

```
return env->NewStringUTF("Hello from JNI!!!!");
```

- 实例方法和静态方法

实例方法和类实例有关，他们只能在类实例中调用；静态方法不与类实例相关，他们可以在静态上下文中直接调用。静态方法和实例方法均可以声明为原生的，可以通过JNI技术以代码的形式提供他们的实现。原生实例方法通过第二个参数获取实力引用，该参数是 `jobject`类型的。

因为静态方法没有与实例绑定，所以第二个参数是类引用而不是实例引用，第二个参数是 `jclass`类型的：

```
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_getStringFromJNI
 (JNIEnv *, jclass);
```

# 数据类型

## Java数据类型

- 基本数据类型:boolean、byte、char、short、int、long、float、double
- 引用类型:字符串型、数组类型等等

## 基本数据类型映射

| Java类型  | JNI类型    | C/C++类型        | 大小     |
|---------|----------|----------------|--------|
| boolean | jboolean | unsigned char  | 无符号8位  |
| byte    | jbyte    | char           | 有符号8位  |
| char    | jchar    | unsigned short | 无符号16位 |
| short   | jshort   | short          | 有符号16位 |
| int     | jint     | int            | 有符号32位 |
| long    | jlong    | long long      | 有符号64位 |
| float   | jfloat   | float          | 32位    |
| double  | jdouble  | double         | 64位    |

## 引用类型

与基本数据类型不同，引用类型对原生方法不透明；它们的内部数据结构并不直接向原生代码公开

| Java类型              | 原生类型          |
|---------------------|---------------|
| java.lang.Class     | jclass        |
| java.lang.Throwable | jthrowable    |
| java.lang.String    | jstring       |
| Other objects       | jobjects      |
| java.lang.Object[]  | jobjectArray  |
| boolean[]           | jbooleanArray |
| byte[]              | jbyteArray    |
| char[]              | jcharArray    |
| short[]             | jshortArray   |
| int[]               | jintArray     |
| long[]              | jlongArray    |
| float[]             | jfloatArray   |
| double[]            | jdoubleArray  |
| Other Arrays        | jarray        |

## 对引用类型的操作

引用类型以不透明的引用方式传递给原生代码，而不是以原生数据类型的方式呈现，因此引用类型不能直接使用和修改。JNI提供了与这些应用类型密切相关的一组API，这些API通过JNIEnv接口指针提供给原生函数。

## 字符串

JNI把Java字符串当成应用类型来处理。这些应用类型并不像原生C字符串一样可以直接使用。JNI提供了Java字符串和C字符串之间相互转换的必要函数，因为Java字符串不可变，所以JNI不提供任何修改现有Java字符串内容的函数；JNI支持Unicode编码格式和UTF-8编码格式的字符串，还提供两组函数通过JNIEnv接口指针处理这些字符串编码。

- 创建Java字符串

`NewString` 返回一个Unicode编码格式的Java字符串应用类型的`jstring` `NewStringUTF` 返回一个UTF-8编码格式的Java字符串应用类型的`jstring`

```
jstring javaString;
javaString = (*env)->NewStringUTF(env, "Hello World!");;
```

在内存溢出的情况下，这些函数返回 `NULL` 以通知原生代码在虚拟机中抛出异常，这样原生代码就会停止运行

- Java字符串转为C字符串

Java字符串不能直接使用，需要使用 `GetStringChars` 函数讲Unicode的Java字符串转为C字符串或者用 `GetStringUTFChars` 函数将UTF-8格式的Java字符串转为C字符串；这两个函数的第三个参数均为可选参数(参数名`isCopy`)，它让调用者确定返回的C字符串地址指向副本还是指向堆中的固定对象：

```
const jbyte* str;
jboolean isCopy;

str = (*env)->GetStringUTFChars(env, javaString, &isCopy);
if(0 != str){
 printf("Java string:%s", str);

 if(JNI_TRUE == isCopy){
 printf("C string is a copy of the java string.");
 }else{
 printf("C string points to actual string.");
 }
}
```

- 释放字符串

通过JNI `GetStringChars` 和 `GetStringUTFChars` 函数获得的C字符串在原生代码中使用完之后需要正确的释放，否则将会引起内存泄露。JNI提供了 `ReleaseStringChars` 函数释放 Unicode编码格式的字符串，提供了 `ReleaseStringUTFChars` 函数释放UTF-8编码格式的字符串：

```
(*env)->ReleaseStringUTFChars(env, javaString, str);
```

## 数组

JNI把Java数组当成引用类型处理，JNI提供必要的函数访问和处理Java数组。

- 创建数组

用 `New<Type>Array` 函数在原生代码中创建数组实例，其中 `<Type>` 可以是 `int` , `char` , `boolean` 等,例如 `NewIntArray` ,使用这些函数的时候，应该以参数形式给出数组大小:

```
jintArray javaArray;
javaArray = (*env)->NewIntArray(env, 10);

if(javaArray != 0){
 /** 可以使用数组了 */
}
```

与 `NewString` 函数一样，在内存溢出的情况下，`New<Type>Array` 函数将返回 `NULL` 以通知原生代码虚拟机中有异常抛出，这样原生代码就会停止运行。

- 访问数组元素

JNI提供两种访问Java数组元素的方法，可以将数组复制成C数组或者让JNI提供直接指向数组元素的指针。

- 对副本的操作

`Get<Type>ArrayRegion` 函数将给定的基本Java数组复制到给定的C数组中:

```
jint nativeArray[10];
(*env)->GetIntArrayRegion(env, javaArray, 0, 10, nativeArray);
```

原生代码可以像使用普通的C数组一样使用和修改数组元素。当原生代码想将所做的修改提交给Java数组时，可以使用 `Set<Type>ArrayRegion` 函数将C数组复制回Java数组:

```
(*env)->SetIntArrayRegion(env, javaArray, 0, 10, nativeArray);
```

当数组很大时，为了对数组进行操作而复制数组会引起性能问题。在这种情况下，如果可能的话，原生代码应该只获取或者设置数组元素区域而不是获取整个数组。另外JNI提供了不同的函数集以获得数组元素而非其副本的直接指针

- 对直接指针的操作

可能的话，原生代码可以用 `Get<Type>ArrayElements` 函数获得指向数组元素的直接指针:

```
jint *nativeDirectArray;
jboolean isCopy;

nativeDirectArray = (*env)->GetIntArrayElements(env, javaArray, &isCopy);
```

因为可以像普通的C数组一样访问和处理数组元素，因此JNI没有提供访问和处理数组元素的方法，JNI要求原生代码用完这些指针立即释放，否则会出现内存泄露。原生代码可以使用JNI提供的 `Release<Type>ArrayElements` 函数释放 `Get<Type>ArrayElements` 函数返回的C数组：

```
(*env)->ReleaseIntArrayElements(env, javaArray, nativeDirectArray, 0);
```

该函数带有四个参数，第四个参数是释放模式，参考如下：

| 释放模式       | 动作                                  |
|------------|-------------------------------------|
| 0          | 将内容赋值回来并释放原生数组                      |
| JNI_COMMIT | 将内容赋值回来但是不释放原生数组，一般用于周期性的更新一个Java数组 |
| JNI_ABORT  | 释放原生数组但不用将内容释放回来                    |

## NIO操作

原生IO(NIO)在缓冲管理区、大规模网络和文件IO及字符集支持方面的性能有所改进，JNI提供了在原生代码中使用NIO的函数。与数组操作相关，NIO混充去的数据传送性能较好，更适合在原生代码和Java应用程序之间传送大量数据。

- 创建直接字节缓冲区

原生代码可以创建Java使用的直接字节缓冲区，该过程是以提供一个原生C字节数组为基础的：

```
unsigned char* buffer = (unsigned char*)malloc(1024);
...
jobject directBuffer;
directBuffer = (*env)->NewDirectByteBuffer(env, buffer, 1024);
```

原生方法中的内存分配超出了虚拟机的管理范围，且不能用虚拟机的GC机制回收原生方法中的内存，原生方法应该通过释放未使用的内存分配以避免内存泄露来正确管理内存。

- 直接字节缓冲区获取

Java也可以创建直接字节缓冲区，在原生代码中调用 `GetDirectBufferAddress` 方法可以获得原生字节数组的内存地址：

```
unsigned char* buffer;
buffer = (unsigned char*) (*env)->GetDirectBufferAddress(env,directBuffer);
```

## 访问变量

JNI提供了访问静态变量和实例变量的方法：

```
public class JavaClass{
 /** 实例变量 */
 private String instanceField = "Instance Field";

 /** 静态变量 */
 private static String staticField = "Static Field";
}
```

- 获取变量Id

获取变量Id都是通过类的Class对象获取的

```
// 获取指定对象的class
jclass clazz;
clazz = (*env)->GetObjectClass(env,thiz);
```

获取Class之后，就可以通过 `GetFiledId` 获取实例变量，通过 `GetStaticFieldId` 获取静态变量：

```
// 获取实例变量Id
jfieldID instanceFieldId;
instanceFieldId = (*env)->GetFieldId(env,thiz,"instanceField","Ljava/lang/String;");

// 获取静态变量id
jfieldID staticFieldId;
staticFieldId = (*env)->GetStaticFieldId(env,clazz,"staticField","Ljava/lang/String;");
```

为了提高应用程序性能，可以缓存变量Id，一般总是缓存使用最频繁的变量Id

- 获取字段的值

得到字段Id之后，就可以通过 `GetObjectField` 获取实例变量的值或者使用 `GetStaticObjectFiled` 获取静态变量的值了：

```
// 获取实例变量的值
jstring instanceField = (*env)->GetObjectField(instanceFiledId);

// 获取静态变量的值
jstring staticField = (*env)->GetStaticObjectField(staticFieldId);
```

注意，如果内存溢出，这些函数返回 `NULL`，此时原生代码不会继续执行

获得单个变量需要调用两到三个JNI函数，原生代码回到Java中获取每个单独的变量，这给应用程序增加了额外的负担，进而导致性能下降。强烈建议将所有需要的参数传递给原生方法调用，而不是让原生代码到Java中取

## 调用方法

与字段一样，JNI提供访问静态方法和实例方法的函数

```
public class JavaClass{

 // 实例方法
 private String instanceMethod(){
 return "Instance Method";
 }

 // 静态方法
 private static String staticMethod(){
 return "Static Method";
 }

}
```

- 获取方法ID

可以通过 `GetMethodID` 获取实例方法ID，通过 `GetStaticMethodID` 获取静态方法的ID

```
// 首先获取指定类的class
jclass clazz = (*env)->GetObjectType(env, thiz);

// 获取实例方法的MethodID
jmethodID instanceMethodId = (*env)->GetMethodID(env, clazz, "instanceMethod", "()Ljava/;

// 获取静态方法的MethodID
jmethodID staticMethodId = (*env)->GetStaticMethodID(env, clazz, "staticMethod", "()Ljav;
```

与字段一样，最后一个参数表示方法描述符，在Java中他表示方法签名

为了提升应用性能，可以缓存方法ID，一般总是缓存使用最频繁的方法ID

- 调用方法

可以以方法ID为参数调用 `call<Type>Method` 函数来调用实例方法，或者调用 `callStatic<Type>Method` 函数调用静态方法：

```
jstring instanceMethodResult = (*env)->CallStringMethod(env, thiz, instanceMethodId);
jstring staticMethodResult = (*env)->CallStaticStringMethod(env, clazz, staticMethodId)
```

## 变量和方法描述符

| Java 类型               | 签 名                     |
|-----------------------|-------------------------|
| Boolean               | Z                       |
| Byte                  | B                       |
| Char                  | C                       |
| Short                 | S                       |
| Int                   | I                       |
| Long                  | J                       |
| Float                 | F                       |
| Double                | D                       |
| fully-qualified-class | Lfully-qualified-class; |
| type[]                | [type]                  |
| method type           | (arg-type)ret-type      |

注意几个比较特殊的：void对应的是V，boolean对应的是Z, long对应的是J

用类型签名映射手工生成变量和方法描述符并让它们跟Java代码同步是一件非常繁琐的任务，所以可以使用javap工具，他可以从class文件中得到变量和方法的描述符

- 命令行下运行

进入目录，然后运行：`javap -classpath bin/classes -p -s 包名.类名`，或者进入到bin/classes下运行 `javap -p -s 包名.类名`

例子：

```
gebilaowang$ javap -p -s com.example.hellojni.HelloJni
Compiled from "HelloJni.java"
public class com.example.hellojni.HelloJni extends android.app.Activity {
 private java.lang.String name;
 Signature: Ljava/lang/String;
 private static java.lang.String company;
 Signature: Ljava/lang/String;
 static {};
 Signature: ()V

 public com.example.hellojni.HelloJni();
 Signature: ()V

 public void onCreate(android.os.Bundle);
 Signature: (Landroid/os/Bundle;)V

 public native java.lang.String stringFromJNI();
 Signature: ()Ljava/lang/String;

 public native java.lang.String unimplementedStringFromJNI();
 Signature: ()Ljava/lang/String;

 public static native java.lang.String getStringFromJNI();
 Signature: ()Ljava/lang/String;

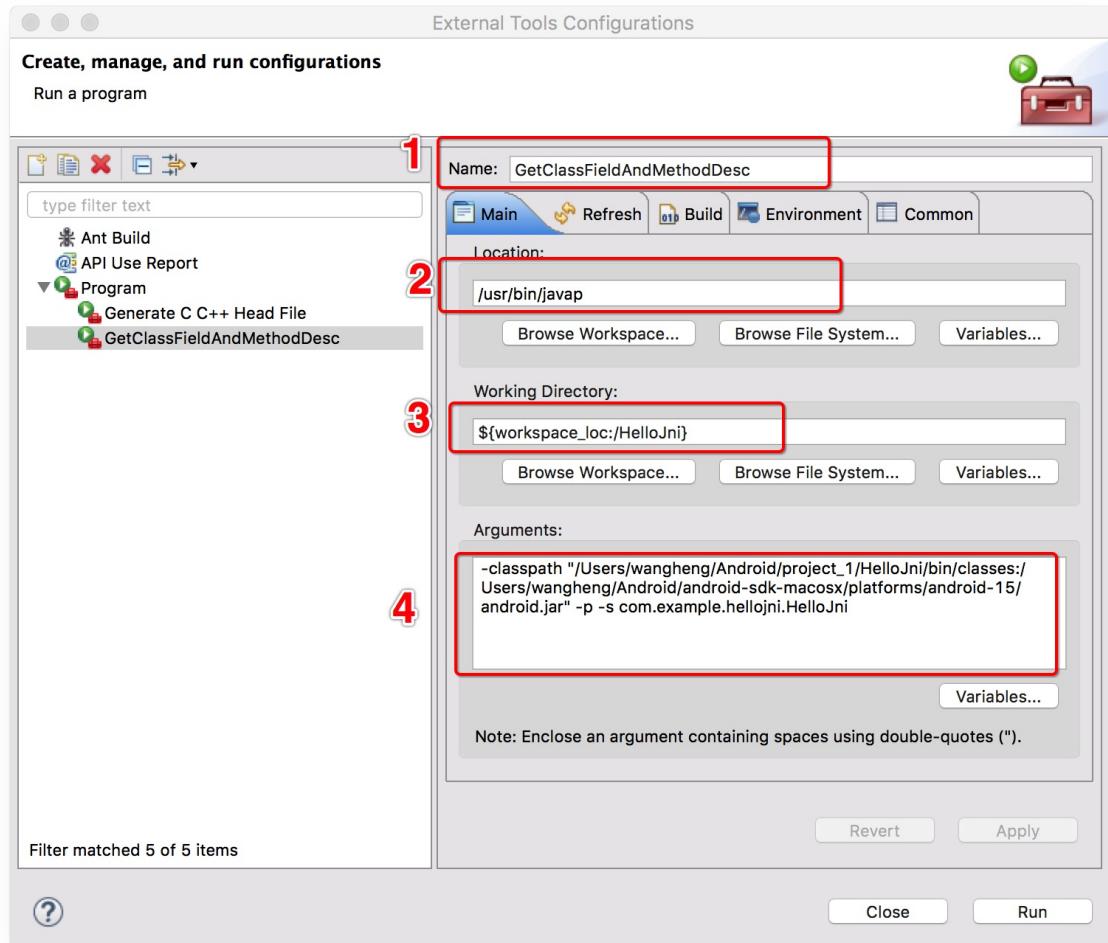
 public static native int getInt();
 Signature: ()I

 public native java.lang.String getString();
 Signature: ()Ljava/lang/String;

 private java.lang.String printByInstance(java.lang.String);
 Signature: (Ljava/lang/String;)Ljava/lang/String;

 private static void printByStatic();
 Signature: ()V
}
```

- Eclipse下调用，参考javah，并在第一步按照下图进行配置



然后执行这个程序就可以在Console中看到相同的输出

## 异常处理

异常处理是Java的重要功能，JNI的异常行为与Java中有所不同。在Java中，当抛出一个异常时，虚拟机停止执行代码块并进入调用栈反向检查能处理特定类型异常的异常处理程序代码块，这也叫异常捕获。虚拟机清除异常并将控制权交给异常处理程序。相比之下，JNI要求开发人员在异常发生后显式地实现异常处理流。

- 捕获异常

JNINv提供了一组与异常相关的函数集，在运行过程中可以使用Java类查看这些函数：

```
// 抛出方法
public class JavaClass{
 private void throwingMethod() throws NullPointerException{
 throw new NullPointerException("Null Pointer");
 }
}
// 访问方法(原生方法)
private native void accessMethods();
```

调用 `throwingMethod` 方式时，`accessMethods` 原生方法需要显式的做异常处理。JNI提供了 `ExceptionOccurred` 函数查询虚拟机中是否有挂起的异常。在使用完之后，异常处理程序需要用 `ExceptionClear` 函数显示的清除异常：

```
jthrowable ex;
...
(*env)->CallVoidMethod(env, thiz, throwingMethodID);
ex = (*env)->ExceptionOccurred(env);
if(0 != ex){
 (*env)->ExceptionClear(env);
 /* Exception处理 */
}
```

- 抛出异常

JNI也允许原生代码抛出异常。因为异常是Java类，应该先用 `FindClass` 函数找到异常类，用 `ThrowNew` 函数可以初始化且抛出新的异常：

```
jclass clazz;
...
clazz = (*env)->FindClass(env, "java/lang/NullPointerException");

if(0 != clazz){
 (*env)->ThrowNew(env, clazz, "Exception Message!");
}
```

因为原生代码执行不受虚拟机的控制，因此异常并不会停止原生代码执行并把控制权转交给异常处理程序。到抛出异常时，原生函数应该释放所有已分配的原生资源，例如内存以及合适的返回值等。通过JNIEnv接口获得的引用是局部引用且一旦返回原生函数，他们自动的被虚拟机释放。

## 局部和全局引用

虚拟机通过追踪类实例的引用并收回不再引用的垃圾来管理类实例的使用期限。因为原生代码不是一个管理环境，因此JNI提供了一组函数允许原生代码显式的管理对象引用和使用期间原生代码。JNI支持三种引用：局部引用、全局引用和弱全局引用

- 局部引用

大多数JNI函数返回局部引用。局部引用不能在后续的调用中被缓存和重用，主要因为它们的使用期限仅限于原生方法，一旦原生函数返回，局部引用即被释放。例如：`FindClass` 函数返回一个局部引用，当原生方法返回时，它被自动释放，也可以用 `DeleteLocalRef` 函数显式的释放原生代码：

```
jclass clazz;
clazz = (*env)->FindClass(env, "java/lang/String");
...
(*env)->DeleteLocalRef(env, clazz);
```

根据JNI规范，虚拟机应该允许原生代码创建至少16个局部引用，在单个方法调用时进行多个内存密集型操作的最佳实践是删除未用的局部变量。如果不可能，原生代码可以在使用之前用 `EnsureLocalCapacity` 方法请求更多的局部变量引用槽。

- 全局引用

全局引用在原生方法的后续调用过程中依然有效，除非它们被原生代码显式释放。

### 创建全局引用

可以使用 `NewGlobalRef` 函数将局部引用初始化为全局引用

```
jclass localClazz;
jclass globalClazz;
...
localClazz = (*env)->FindClass(env, "java/lang/String");
globalClazz = (*env)->NewGlobalRef(env, localClazz);
...
(*env)->DeleteLocalRef(env, localClazz);
```

### 删除全局引用

当原生代码不再需要一个全局引用时，可以随时用 `DeleteGlobalRef` 函数释放它：

```
(*env)->DeleteGlobalRef(env, globalClazz);
```

- 弱全局引用

全局引用的另一种类型是弱全局引用。与全局引用一样，弱全局引用在原生方法的后续调用中仍然有效。但是弱全局引用不阻止潜在的对象被GC回收

### 创建 - NewWeakGlobalRef

```
jclass weakGlobalClazz = (*env)->NewWeakGlobalRef(env, localClazz);
```

**有效性检查 - IsSameObject:**是否仍然指向活动的类实例

```
if(JNI_FLASe == (*env)->IsSameObject(env, weakGlobalClazz, NULL)){
 // 对象仍然处于活动状态且可以使用
}else{
 // 对象被GC回收，不能使用
}
```

### 删除 - DeleteWeakGlobalRef

```
(*env)->DeleteWeakGlobalRef(env, weakGlobalClazz);
```

全局应用释放前一直有效，它们可以被其他函数以及原生线程使用。

## 线程

作为多线程环境的一部分，虚拟机支持运行原生代码。在开发原生组件时要记住JNI技术的一些约束：

- 只在native方法执行期间以及在执行原生方法的线程环境下局部引用是有效的，局部变量不能在多线程间共享
- 被传递给每个native方法的JNIEnv接口指针在与方法调用相关的线程中也是有效的，它不能被其他线程缓存或者使用

## 同步

与Java的同步类似，JNI的监视器允许原生代码利用java对象同步，虚拟机保证存取监视器的线程能够安全执行，而其他线程等待监视器对象变成可用状态：

```

// java同步代码块
synchronized(obj){
 // 同步执行安全代码块
}

// JNI同步
if(JNI_OK == (*env)->MonitorEnter(env,obj)){
 /** 错误处理 **/
}

/** 同步线程安全代码块 **/

if(JNI_OK == (*env)->MonitorExit(env,obj)){
 /** 错误处理 **/
}

```

对MonitorEnter函数的调用应该与MonitorExit的调用相匹配，从而避免出现死锁

## 原生线程

为了执行特定任务，这些native组件可以并行使用native线程。因为VM不知道的native线程，因此它们不能与Java组件直接通信。为了与应用的依然活跃部分交互，原生线程应该先附着在VM上。

JNI通过JavaVM接口指针提供了 `AttachCurrentThread` 函数以便让native代码将native线程附着到VM上，JavaVM接口指针应该尽早被缓存，否则的话他不能被获取：

```

JavaVM* cachedJvm;
...
JNIEnv* env;
...
/** 将当前线程附着到VM **/
(*cachedJvm)->AttachCurrentThread(cachedJvm,&env,NULL);

/** 可以用JNIEnv接口实现与Java应用程序的通信 **/

/** 将当前线程与虚拟机分离 **/
(*cachedJvm)->DetachCurrentThread(cachedJvm);

```

对 `AttachCurrentThread` 函数的调用允许应用程序获得对当前线程的有效JNIEnv接口指针。将一个已经附着的原生线程再次附着不会有任何副作用。当原生线程完成时，可以用 `DetachCurrentThread` 函数将原生线程与VM分离。



# 使用SWIG自动生成JNI代码

## 什么是SWIG

SWIG:简化的包装器和接口生成器， Simplified Wrapper And Interface Generator

SWIG不是Android或者Java的专用工具， 它是一个可以生成许多其他编程语言代码的、 广泛使用的工具。

SWIG是一个编译时软件开发工具， 能生成将用C/C++编写的原生模块与包括Java在内的其他编程语言进行联接的必要代码。它不仅仅是一个代码生成器， 还是一个接口编译器。他不定义新的协议， 也不是一个组件框架或者一个特定的运行时库。SWIG把接口文件看做输入， 并生成必要的代码在Java中展示接口， 从而让Java能够理解原生代码中的接口定义。SWIG不是一个存根生成器； 它产生将要被编译和运行的代码。

## 在MAC上安装SWIG

SWIG 网站没有提供MAC OS X平台的安装包， 需要用Homebrew包管理器下载并安装 SWIG(需要安装Homebrew)。

然后执行命令: `brew install swig` ,然后执行 `swig -version` 验证安装

# Android NDK 基础

## 环境搭建

- 1、下载SDK并解压&配置环境变量
- 2、安装Eclipse ADT插件
- 3、下载NDK开发包并解压&配置环境变量
- 4、新建Android工程或者导入已存在的Android工程
- 5、Preferences->Android->NDK设置NDK目录
- 6、右键工程->Android Tools->Add Native Support
- 如果编译出错，提示以下信息

```
Type '*****' could not be resolved
Method '*****' could not be resolved
```

则按照以下步骤解决：

右键工程->Properties->C/C++ General->Paths and Symbols->Includes->Add->添加/platforms/android-19/arch-arm/usr/include

## 构建系统

- NDK的构建系统是基于GNU Make的。主要目的是让开发人员能够用很短的构建文档来描述原生Android应用；还处理了包括替代开发热暖指定工具链、平台、CPU和ABI等很多细节。封装该构建过程可以在不改变构建文件的情况下，使Android NDK的后续更新添加更多对工具链、平台以及系统接口的支持。
- Android NDK 构建系统是由多种GNU Makefile片段构成的。该构建系统包括基于渲染构建过程的不同类型NDK所需要的必
- 要片段。这些片段可以在/build/core子目录找到。一般开发人员并不会接触这些文件，但知道他们的位置对于构建系统相关的故障很有帮助
- 除了这些片段，NDK构建系统还依赖另外两个文件:Android.mk以及Application.mk

# Android.mk

Android.mk是一个向NDK构建系统描述NDK项目的GNU Makefile片段。它使每一个NDK项目的必备组件。构建系统希望它出现在jni子目录中。

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := hello-jni
LOCAL_SRC_FILES := hello-jni.c

include $(BUILD_SHARED_LIBRARY)
```

- 以#开头的是注释行
- 根据命名规范，变量名大写

## **LOCAL\_PATH := \$(call my-dir)**

第一行指令用来定义 `LOCAL_PATH` 变量，根据NDK构建系统的要求，`Android.mk` 文件必须以 `LOCAL_PATH` 变量的第一行开始,Android 构建系统利用 `LOCAL_PATH` 来定位源文件。因为该变量设置为硬编码值并不合适，所以Android构建系统提供了一个名为my-dir的宏功能。通过将该变量设置为my-dir宏功能的返回值，可以将其指定为当前目录

## **include \$(CLEAR\_VARS)**

Android构建系统将 `CLEAR_VARS` 变量设置为 `clear-vars.mk` 片段的位置。包含 `Makefile` 片段可以清除除了 `LOCAL_PATH` 以外的 `LOCAL_<name>` 变量，例如 `LOCAL_MODULE` 与 `LOCAL_SRC_FILES` 等；这样做是因为Android构建系统在单次执行中解析多个构建文件和模块定义，而 `LOCAL_<name>` 是全局变量，清除它们可以避免冲突，每一个原生组件被称为一个模块

## **LOCAL\_MODULE := hello-jni**

`LOCAL_MODULE` 变量用来给这些模块设定一个唯一的名称，因为模块名称也被用于给构建过程所生成的文件命名，所以构建系统给该文件添加了适当的前缀和后缀(如，前面加 `lib` ,后面加 `.so` )，本例中 `hello-jni` 模块会生成一个共享库文件并且构建系统将它命名为 `libhello-jni.so`

## **LOCAL\_SRC\_FILES := hello-jni.c**

`LOCAL_SRC_FILES` 变量定义用来简历和组装这个模块的源文件列表，这个变量可以是以空格分隔的多个源文件名称(如果需要换行，需要在结尾加上一个`\`)

以上，`Android.mk` 文件定义的构建系统变量简单描述了原生项目。编译和生成实际模块的构建系统还需要包含合适的构建系统模块，具体需要哪些片段取决于想要生成的模块类型(共享库和静态库)

## 构建变量

### 构建共享库

为了建立可供主应用程序使用的模块，必须将该模块变成共享库。Android NDK构建系统将`BUILD_SHARED_LIBRARY` 变量设置成`build-shared-library.mk` 文件的保存位置。这个`Makefile` 片段包含了将该源文件构建和组装成共享库的必要过程：

```
include $(BUILD_SHARED_LIBRARY)
```

### 构建多个共享库

基于不同的应用程序的体系架构，一个单独的`Android.mk` 文档可能产生多个共享库模块。为了达到这个目的，需要在`Android.mk` 文档中定义多个模块：

```
LOCAL_PATH := $(call-mydir)

模块1
include $(CLEAR_VARS)

LOCAL_MODULE := module1
LOCAL_SRC_FILES := module11.c module12.c module13.c
include $(BUILD_SHARED_LIBRARY)

模块2
include $(CLEAR_VARS)

LOCAL_MODULE := module2
LOCAL_SRC_FILES := module21.c module22.c module23.c
include $(BUILD_SHARED_LIBRARY)
```

在处理完这个`Android.mk` 构建文档之后，Android NDK构建系统会产生`libmodule1.so`和`libmodule2.so`两个共享库

## 构建静态库

Android NDK构建系统也支持静态库，实际的Android应用程序并不直接使用静态库，并且应用程序中也不包含静态库。静态库可以用来构建共享库，例如，在将第三方代码添加到现有原生项目中的时候，不直接使用第三方源码包括在原生项目中，而是将第三方代码编译成静态库然后并入共享库：

```
LOCAL_PATH := $(my-dir)

第三方AVI库
include $(CLEAR_VARS)

LOCAL_MODULE := avilib
LOCAL_SRC_FILES := avilib.c platform_posix.c

include $(BUILD_STATIC_LIBRARY)

原生模块
include $(CLEAR_VARS)

LOCAL_MODULE := module
LOCAL_SRC_FIELS := module.c

LOCAL_STATIC_LIBRARIES := avilib
include $(BUILD_SHARED_LIBRARY)
```

在将第三方模块生成静态库之后，共享库就可以通过将它的模块名添加到 `LOCAL_STATIC_LIBRARIES` 变量中来使用该模块

## 用共享库共享通用模块

静态库可以保证源码模块化，但是当将太苦与共享库连接时，它就变成了共享库的一部分。在多个共享库的情况下，多个共享库与同一个静态库连接时，需要将通用模块的多个副本与不同共享库重复连接，这样就增加了应用程序的大小。在这种情况下，不用构建静态库，而是将通用模块作为共享库建立起来，而公台连接依赖模块以便消除重复副本：

```

LOCAL_PATH := $(my-dir)

第三方AVI库
include $(CLEAR_VARS)

LOCAL_MODULE := avilib
LOCAL_SRC_FILES := avilib.c platform_posix.c

include $(BUILD_SHARED_LIBRARY)

原生模块1
include $(CLEAR_VARS)

LOCAL_MODULE := module1
LOCAL_SRC_FILES := module1.c

LOCAL_SHARED_LIBRARIES := avilib

include $(BUILD_SHARED_LIBRARY)

原生模块2
include $(CLEAR_VARS)

LOCAL_MODULE := module2
LOCAL_SRC_FILES := module2.c

LOCAL_SHARED_LIBRARIES := avilib

include $(BUILD_SHARED_LIBRARY)

```

## 在多个NDK项目间共享模块

同时使用静态库和共享库时，可以在模块间共享通用模块。但是这些模块必须属于同一个NDK项目。从R5版本开始，NDK允许在NDK项目间共享和重用模块。可以通过以下步骤在多个NDK项目间共享avilib模块：

- 首先，将avilib源代码移动到NDK项目外的位置，例如:C:\android\shared-modules\avilib.为了避免命名冲突，目录结构也可以包含模块提供者的名字，如:C:\android\shared-modules\gebilaowang\avilib

**注意，在Android NDK构建系统中，共享模块路径不能包含空格**

- 作为共享模块，avilib需要自己的Android.mk文件：

```

LOCAL_PATH := $(call-mydir)

第三方AVI库

include $(CLEAR_VARS)
LOCAL_MODULE := avilib
LOCAL_SRC_FILES := avilib.c platform_posix.c

include $(BUILD_SHARED_LIBRARY)

```

- 现在，可以讲avilib模块从NDK项目的Android.mk文件中移除。为了使用这个共享模块，将以gebilaowang/avilib为参数调用函数宏import-module部分添加在构建文档的末尾(为了避免构建系统的冲突，应该将import-module函数宏调用放在Android.mk文件的末尾):

```

原生模块
include $(CLEAR_VARS)

LOCAL_MODULE := module
LOCAL_SRC_FILES := module.c
LOCAL_SHARED_LIBRARIES := avilib

include $(BUILD_SHARED_LIBRARY)

$$(call import-module, gebilaowang/avilib)

```

- import-module 函数宏调用需要限定为共享模块，然后再将它导入到NDK项目中。默认情况下，import-module 函数宏至搜索 <NDK\_ROOT>/sources目录。为了搜索 c:\android\shared-modules 目录，定义一个名为 NDK\_MODULE\_PATH 的新环境变量，并将它设置为共享模板的根目录，如:C:\android\shared-modules

## 使用Prebuilt库

使用共享模块要求有共享模块的源代码，Android NDK构建系统简单的把这些源文件包含在NDK项目中并每次构建他们。自从R5版本以后，Android NDK也提供对Prebuilt库的支持。在下面的情况下，Prebuilt库是非常有用的：

- 想在不发布源代码的情况下将你的模块发布给其他人
- 想使用共享模块的预建版本来加速构建过程

尽管已经被编译了，但是预建模块仍需要一个 Android.mk 构建文档：

```

LOCAL_PATH := $(call my-dir)

第三方构建AVI库
include $(CLEAR_VARS)

LOCAL_MODULE := avilib
LOCAL_SRC_FILES := libavilib.so

include $(PREBUILT_SHARED_LIBRARY)

```

`LOCAL_SRC_FILES` 变量指向的不是源文件，而是实际Prebuilt库相对于 `LOCAL_PATH` 的位置

注意，Prebuilt库定义中不包含任何关于该库所构建的实际机器体系结构的信息。开发人员需要保证Prebuilt库是为与NDK项目相同的机器体系结构而构建的

`PREBUILT_SHARED_LIBRARY` 变量指向 `prebuilt-shared-library.mk` Makefile片段，它什么都没有构建，但是它将Prebuilt库复制到了NDK项目的libs目录下。通过使用 `PREBUILT_STATIC_LIBRARY` 变量，静态库可以像共享库一样被用作Prebuilt库，NDK项目可以像普通共享库一样使用Prebuilt库了。

## 构建独立的可执行文件

在Android平台上使用原生组件的推荐和支持的方法是将他们打包成共享库。但是为了方便测试和进行快速原型设计，Android NDK也支持构建独立的可执行文件。这些独立的可执行文件不用打包成APK文件就可以复制到Android设备上的常规Linux应用程序，而且他们可以直接执行，而不通过java程序加载。生成独立的可执行文件需要在 `Android.mk` 构建文档中导入 `BUILD_EXECUTABLE` 变量，而不是导入 `BUILD_SHARED_LIBRARY` 变量：

```

独立的可执行的原生模块
include $(CLEAR_VARS)

LOCAL_MODULE := module
LOCAL_SRC_FILES := module.c

LOCAL_STATIC_LIBRARIES := avilib

include $(BUILD_EXECUTABLE)

```

`BUILD_EXECUTABLE` 变量执行 `build-executable.mk` Makefile片段，该片段包含了在Android平台生成独立可执行文件的必要步骤。独立可执行文件以与模块相同的名称被放在 `libs/<machine architecture>` 目录下。尽管放在该目录下，但在打包阶段它并没有被包含在APK文件中。

## 其他构建系统变量

### 构建系统变量

- TARGET\_ARCH: 目标CPU体系架构的名称, 如arm
- TARGET\_PLATFORM: 目标Android平台的名称, 如android-3
- TARGET\_ARCH\_ABI: 目标CPU体系结构和ABI名称, 例如armeabi-v7a
- TARGET\_ABI: 目标平台和ABI的窗帘, 如android-3-armeabi-v7a

### 可被定义为模块说明部分的变量

- LOCAL\_MODULE\_FILENAME: 可选变量, 用来重新定义生成输出文件名称。默认情况下, 构建系统使用LOCAL\_MODULE的值作为生成的输出文件名称, 但是变量LOCAL\_MODULE\_FILENAME可以覆盖LOCAL\_MODULE的值
- LOCAL\_CPP\_EXTENSION: C++源文件的默认扩展名是.cpp, 这个变量可以用来为C++源文件指定一个或者多个文件扩展名: LOCAL\_CPP\_EXTENSION := .cpp .cxx
- LOCAL\_CPP\_FEATURES: 可选变量, 用来指明模块所以来的具体C++特性, 如RTTI, exceptions等: LOCAL\_CPP\_FEATURES := rtti
- LOCAL\_C\_INCLUDES: 可选目录列表, NDK安装目录的相对路径, 用来搜索头文件:

```
LOCAL_C_INCLUDES := sources/shared-module
LOCAL_C_INCLUDES := $(LOCAL_PATH)/includes
```

- LOCAL\_CFLAGS: 一组可选的编译器标志, 在编译C和C++源文件的事后会被传送给编译器: LOCAL\_CFLAGS := -DNDEBUG -DPORT=1234
- LOCAL\_CPP\_FLAGS: 一组可选的编译标志, 在只编译C++源文件时被传送给编译器
- LOCAL\_WHOLE\_STATIC\_LIBRARIES: LOCAL\_STATIC\_LIBRARIES的变体, 用来指明应该被包含在生成的共享库中的所有静态库内容

注意, 当几个静态库之间有循环依赖时, LOCAL\_WHOLE\_STATIC\_LIBRARIES很有用

- LOCAL\_LDLIBS: 链接标志的可选列表, 当对目标文件进行链接以生成输出文件时该标志将被传送给链接器。它主要用于传送要进行动态链接的系统库列表。例如, 要与Android NDK日志库链接, 使用以下代码: LOCAL\_LDLIBS := -llog
- LOCAL\_ALLOW\_UNDEFINED\_SYMBOLS: 可选参数, 它进制在生成的文件中进行缺失符号检查, 若没有定义, 链接器会在符号缺失时生成错误信息

- LOCAL\_ARM\_MODE: 可选参数, ARM机器体系架构特有变量, 用于指定要生成的ARM二进制类型, 默认情况下, 构建体系在拇指模式下用16位指令生成, 但是改变量可以被设置成arm来制定使用32位指令: LOCAL\_ARM\_MODE=arm ,该变量改变了整个模块的构建系统行为; 可以用 .arm 扩展名指定只在arm模式下构建特定文件 LOCAL\_SRC\_FILES :=file1.c file2.c.arm
- LOCAL\_ARM\_NEON:可选参数, ARM机器体系架构特有的变量, 用来指定在源文件中使用的ARM高级单指令流多数据流(Single Instruction Multiple Data,SIMD) (a.k.a. NEON) 内联函数, LOCAL\_ARM\_NEON :=true ,该变量改变了整个模块的构建系统行为, 可以用 .neon 扩展名指定只构建带有NEON内联函数的特定文件: LOCAL\_SRC\_FILES :=file1.c file.c.neon
- LOCAL\_DISABLE\_NO\_EXECUTE:可选变量, 用来禁用NX Bit安全特性, NX Bit代表Never Execute(永不执行), 它是在CPU中使用的一项技术, 用来隔离代码区和存储区。这样可以防止恶意软件通过将它的代码插入应用程序的存储区来控制应用程序: LOCAL\_DISABLE\_NO\_EXECUTE :=true
- LOCAL\_EXPORT\_CFLAGS:该变量记录一组编译器标志, 这些编译器标志会被添加到通过变量LOCAL\_STATIC\_LIBRARIES或LOCAL\_SHARED\_LIBRARIES使用本模块的其他模块的LOCAL\_CFLAGS定义中:

```

LOCAL_MODULE := avilib
...
LOCAL_EXPORT_CFLAGS := -DENABLE_AUDIO
...
LOCAL_MODULE := module1
LOCAL_CFLAGS := -DDEBUG
...
LOCAL_SHARED_LIBRARIES := avilib

```

编译器会在构建module1的时候以-DENABLE\_AUDIO -DDEBUG标志执行

- LOCAL\_EXPORT\_CPPFLAGS:和LOCAL\_EXPORT\_CFLAGS一样, 但是它是C++特定代码编译器标志
- LOCAL\_EXPORT\_LDFLAGS:和LOCAL\_EXPORT\_CFLAGS一样, 但用作链接器标志
- LOCAL\_EXPORT\_C\_INCLUDES:该变量允许记录路径集, 这些路径会被添加到通过变量LOCAL\_STATIC\_LIBRARIES或LOCAL\_SHARED\_LIBRARIES使用该模块的LOCAL\_C\_INCLUDES定义中
- LOCAL\_SHORT\_COMMANDS:对于有大量资源或者独立的静态、共享库的模块, 该变量应该被设置成true, 诸如windows之类的操作系统只允许命令行最多输入8191个字符; 该变量会分解构建命令使其长度小于8191个字符, 在较小的模块中不推荐使用该方法, 因为使用它会让构建过程变慢

- LOCAL\_FILTER\_ASM:该变量定义了用于过滤来自LOCAL\_SRC\_FILES变量的装配文件的应用程序

## 其他的构建系统函数宏

- all-subdir-makefiles:返回当前目录的所有子目录下的Android.mk构建文件列表。例如，调用以下命令可以将子目录下的所有Android.mk文件包含在构建过程中：`include $(call all-subdir-makefiles)`
- this-makefile:返回当前Android.mk构建文件路径
- parent-makefile:返回本航当前构建文件的父Android.mk构建文件的路径
- grand-parent-makefile:和parent-makefile一样但用于祖父目录

## 定义新变量

开发人员可以定义其他变量来简化他们的构建文件。以LOCAL和NDK前缀开头的名称预留给Android NDK构建系统，建议开发人员定义的变量以MY\_开头，如：

```
MY_SRC_FILES := avllib.c platform_posix.c
LOCAL_SRC_FILES := $(addprefix avllib/, $(MY_SRC_FILES))
```

## 条件操作

Andorid.mk构建文件也可以包含关于这些变量的条件判断，例如，在每个体系结构中包含一个不同个源文件集，如：

```
ifeq ($(TARGET_ARCH), arm)
 LOCAL_SRC_FILES += armonly.c
else
 LOCAL_SRC_FILES += generic.c
endif
```

## Application.mk

`Application.mk` 也是Android NDK构建系统中使用的一个可选构建文件。和 `Android.mk` 文件一样，它也被放在jni目中。`Application.mk` 也是一个GNU Makefile的片段。它的目的是描述应用程序需要哪些模块；它也定义所有模块的通用变量。以下是 `Application.mk` 构建文件支持的变量：

- APP\_MODULES:默认情况下，Android NDK构建系统构建Android.mk文件声明的所有模块

有模块。该变量可以覆盖上述行为并提供一个空格分开的、需要被构建的模块列表

- APP\_OPTIM: 该变量可以被设置为release或者debug以改变生成的二进制文件的优化级别。默认情况下是release模式，并且此时生成的二进制文件被高度优化。该变量可以被设置为debug模式以生成更容易调试的未优化的二进制文件
- APP\_CFLAGS: 该变量列出了一些编译器标志，在编译任何模块的C和C++源文件时，这些标记都被传给编译器
- APP\_CPPFLAGS: 该变量类除了一些编译器标志，在编译任何模块的C++源文件时，这些标志都会被传递给编译器
- APP\_BUILD\_SCRIPT: 默认情况下，Android NDK构建系统在项目的jni子目录下查找Android.mk文件。可以用该变量改变上述行为，并使用不同的生成文件
- APP\_ABI: 默认情况下，Android NDK构建系统为armabi ABI生成二进制文件。可以使用该变量改变上述行为，并为其他ABI生成二进制文件，如：
  - APP\_ABI := mips ## 设置一个ABI
  - APP\_ABI := mips armeabi ## 设置多个ABI
  - APP\_ABI := all ## 所有ABI
- APP\_STL: 默认情况下，Android NDK构建系统使用最小STL运行库，可被称为system库。可以用该变量选择不同个STL实现：APP\_STL := stlport\_shared
- APP\_GNUSTL\_FORCE\_CPP\_FEATURES: 与LOCAL\_CPP\_EXTENSION变量相似，该变量表明所有模块都依赖于具体的C++特性，如RTTI、exceptions等。
- APP\_SHORT\_COMMANDS: 与LOCAL\_SHORT\_COMMANDS变量相似，该变量使得构建系统在有大量源文件的情况下可以在项目中使用更短的命令

## NDK-Build脚本

可以通过执行ndk-build脚本启动Android NDK构建系统，该脚本用一组参数使维护和控制构建过程更容易。

- 默认情况下，ndk-build脚本应该在主项目中执行。-C参数可以用于指定命令行中NDK项目的位置，这样来ndk-build脚本可以从任何位置开始：ndk-build -C  
/path/to/the/project
- 如果源文件没被修改，Android NDK构建系统不会重新构建目标，可以用-B执行ndk-build脚本来强制重新构建所有的源代码：ndk-build -B
- 为了清理二进制文件和目标文件，可以执行 nek-build clean 命令
- Android NDK构建系统依赖于GNU Make工具对模块进行构建，默认情况下，GNU Make工具一次执行依据构建命令，等这一句执行完以后再执行下一句。如果在命令行使用-j参数，GNU Make就可以并行执行构建命令。另外可用功过指定该参数之后的数字来制定并行执行的线程数：nek-build -j 4
- Android NDK构建系统有大量的日志以支持系统相关的故障排除，在命令行输入 ndk-

`build NDK_LOG=1` 便可以启用Android NDK构建系统内部状态日志功能。Android NDK 构建系统会产生大量的日志，日志消息的前缀是 `Android NDK` ;如果执行看到实际执行的构建命令，可以在命令行输入 `ndk-build V=1`



# Android性能优化典范

# Android性能优化典范 - 第5季

来源:<http://hukai.me/android-performance-patterns-season-5/>

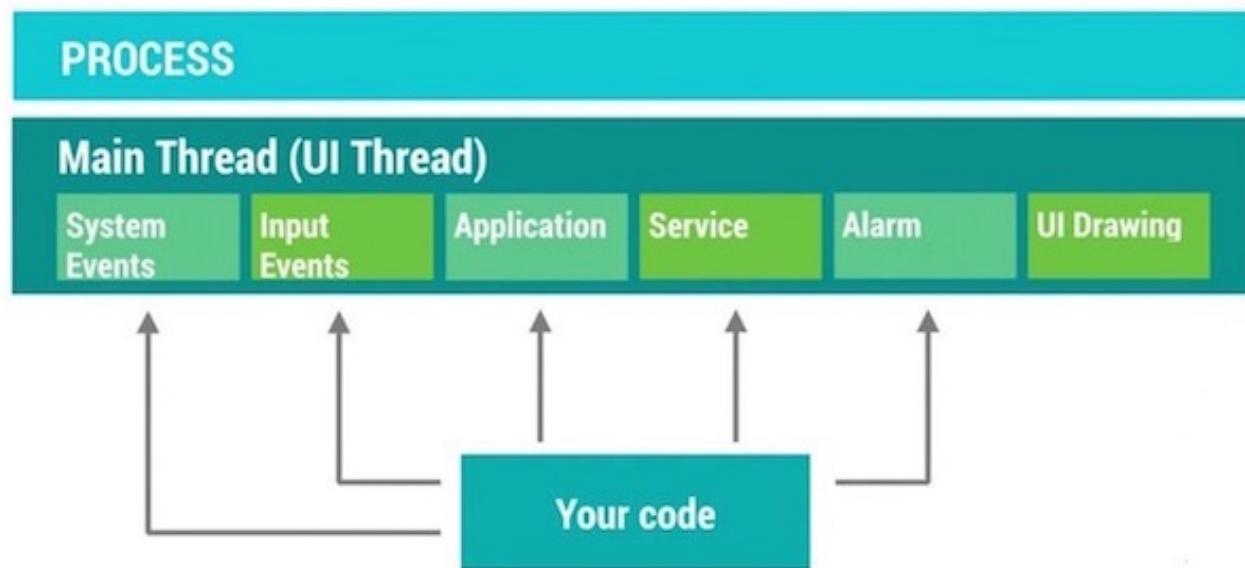


这是Android性能优化典范第5季的课程学习笔记，拖拖拉拉很久，记录分享给大家，请多多包涵担待指正！文章共10个段落，涉及的内容有：多线程并发的性能问题，介绍了AsyncTask, HandlerThread, IntentService与ThreadPool分别适合的使用场景以及各自的使用注意事项，这是一篇了解Android多线程编程不可多得的基础文章，清楚的了解这些Android系统提供的多线程基础组件之间的差异以及优缺点，才能够在项目实战中做出最恰当的选择。

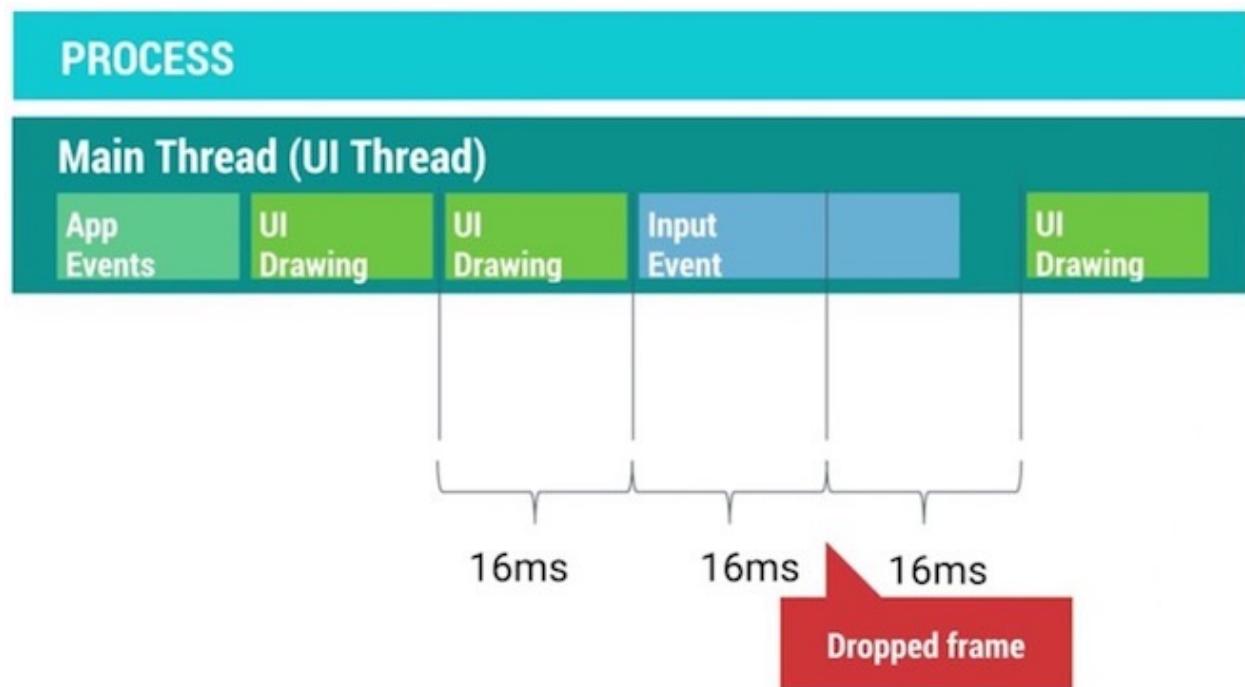
## 1) Threading Performance

在程序开发的实践当中，为了让程序表现得更加流畅，我们肯定会需要使用到多线程来提升程序的并发执行性能。但是编写多线程并发的代码一直以来都是一个相对棘手的问题，所以想要获得更佳的程序性能，我们非常有必要掌握多线程并发编程的基础技能。

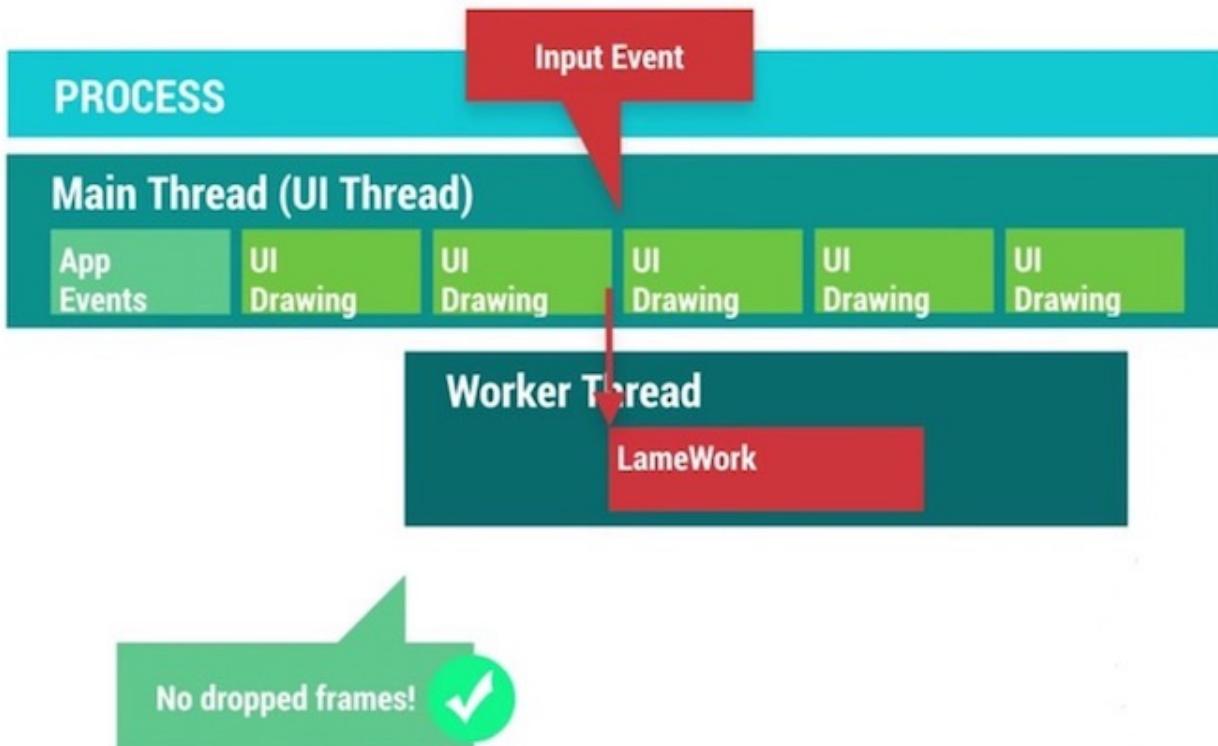
众所周知，Android程序的大多数代码操作都必须执行在主线程，例如系统事件(例如设备屏幕发生旋转)，输入事件(例如用户点击滑动等)，程序回调服务，UI绘制以及闹钟事件等等。那么我们在上述事件或者方法中插入的代码也将执行在主线程。



一旦我们在主线程里面添加了操作复杂的代码，这些代码就很可能阻碍主线程去响应点击/滑动事件，阻碍主线程的UI绘制等等。我们知道，为了让屏幕的刷新帧率达到60fps，我们需要确保16ms内完成单次刷新的操作。一旦我们在主线程里面执行的任务过于繁重就可能导致接收到刷新信号的时候因为资源被占用而无法完成这次刷新操作，这样就会产生掉帧的现象，刷新帧率自然也就跟着下降了(一旦刷新帧率降到20fps左右，用户就可以明显感知到卡顿不流畅了)。



为了避免上面提到的掉帧问题，我们需要使用多线程的技术方案，把那些操作复杂的任务移动到其他线程当中执行，这样就不容易阻塞主线程的操作，也就减小了出现掉帧的可能性。



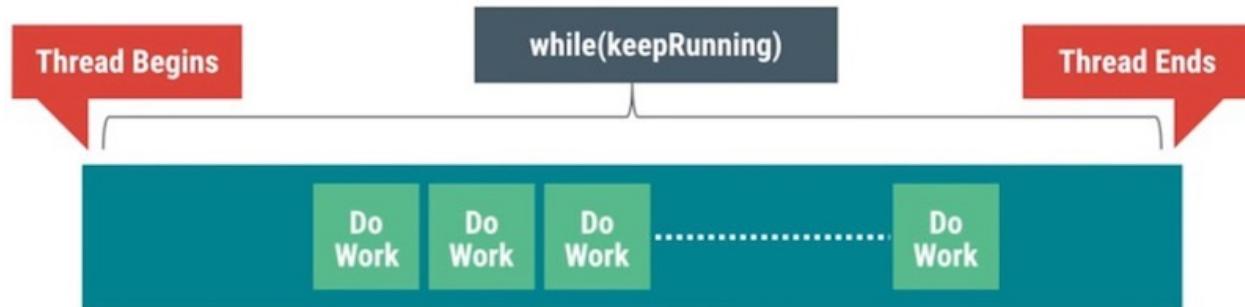
那么问题来了，为主线程减轻负的多线程方案有哪些呢？这些方案分别适合在什么场景下使用？Android系统为我们提供了若干组工具类来帮助解决这个问题。

- **AsyncTask**: 为UI线程与工作线程之间进行快速的切换提供一种简单便捷的机制。适用于当下立即需要启动，但是异步执行的生命周期短暂的使用场景。
- **HandlerThread**: 为某些回调方法或者等待某些任务的执行设置一个专属的线程，并提供线程任务的调度机制。
- **ThreadPool**: 把任务分解成不同的单元，分发到各个不同的线程上，进行同时并发处理。
- **IntentService**: 适合于执行由UI触发的后台Service任务，并可以把后台任务执行的情况通过一定的机制反馈给UI。

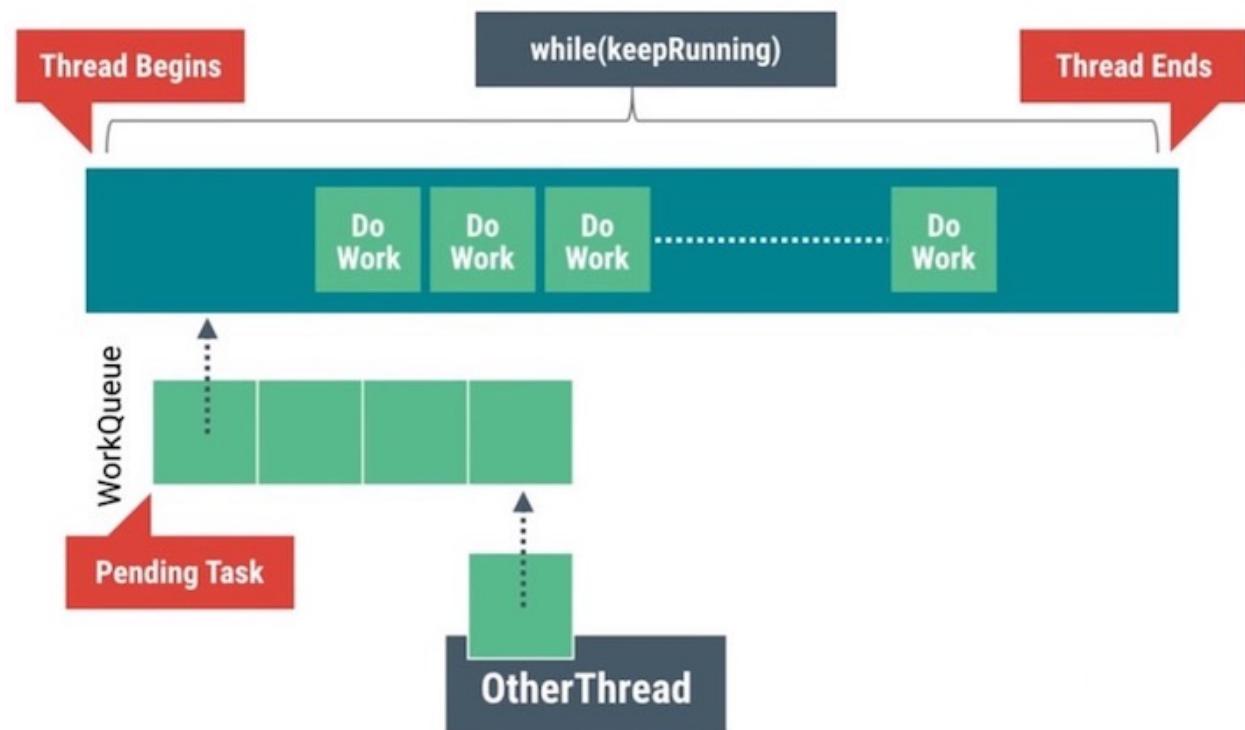
了解这些系统提供的多线程工具类分别适合在什么场景下，可以帮助我们选择合适的解决方案，避免出现不可预期的麻烦。虽然使用多线程可以提高程序的并发量，但是我们需要特别注意因为引入多线程而可能伴随而来的内存问题。举个例子，在Activity内部定义的一个AsyncTask，它属于一个内部类，该类本身和外面的Activity是有引用关系的，如果Activity要销毁的时候，AsyncTask还仍然在运行，这会导致Activity没有办法完全释放，从而引发内存泄漏。所以说，多线程是提升程序性能的有效手段之一，但是使用多线程却需要十分谨慎小心，如果不了解背后的执行机制以及使用的注意事项，很可能引起严重的问题。

## 2) Understanding Android Threading

通常来说，一个线程需要经历三个生命阶段：开始，执行，结束。线程会在任务执行完毕之后结束，那么为了确保线程的存活，我们会在执行阶段给线程赋予不同的任务，然后在里面添加退出的条件从而确保任务能够执行完毕后退出。



在很多时候，线程不仅仅是线性执行一系列的任务就结束那么简单的，我们会需要增加一个任务队列，让线程不断的从任务队列中获取任务去进行执行，另外我们还可能在线程执行的任务过程中与其他的线程进行协作。如果这些细节都交给我们自己来处理，这将会是件极其繁琐又容易出错的事情。

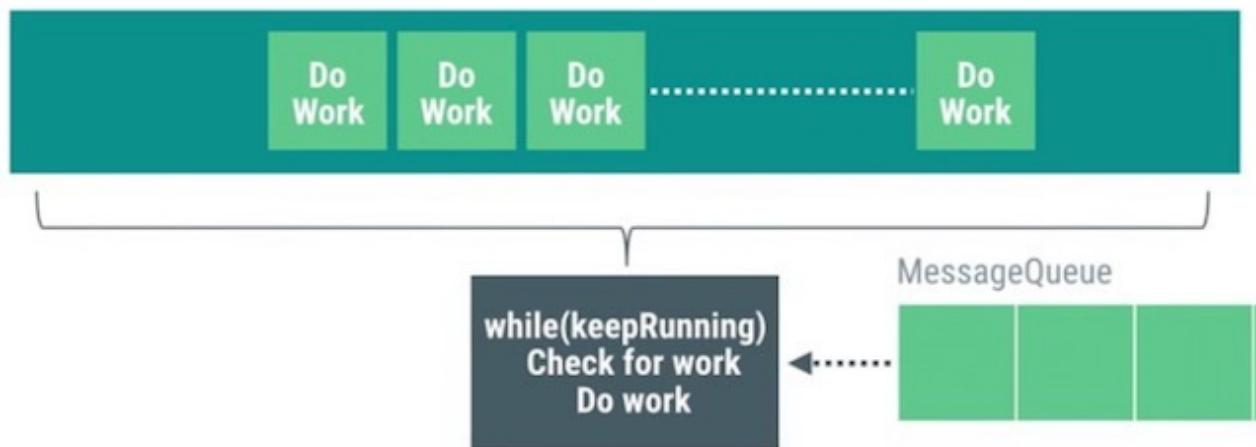


所幸的是，Android系统为我们提供了Looper, Handler, MessageQueue来帮助实现上面的线程任务模型：

**Looper:** 能够确保线程持续存活并且可以不断的从任务队列中获取任务并进行执行。

# Looper

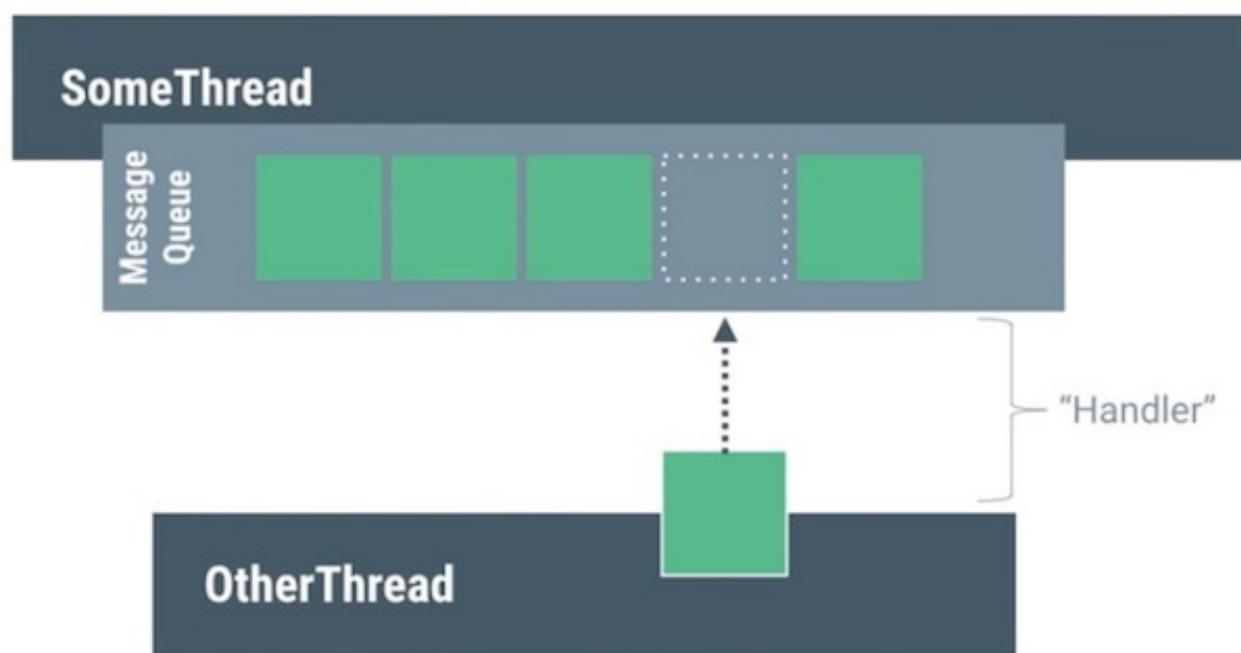
aka "Event Loop"



**Handler:** 能够帮助实现队列任务的管理，不仅仅能够把任务插入到队列的头部，尾部，还可以按照一定的时间延迟来确保任务从队列中能够来得及被取消掉。

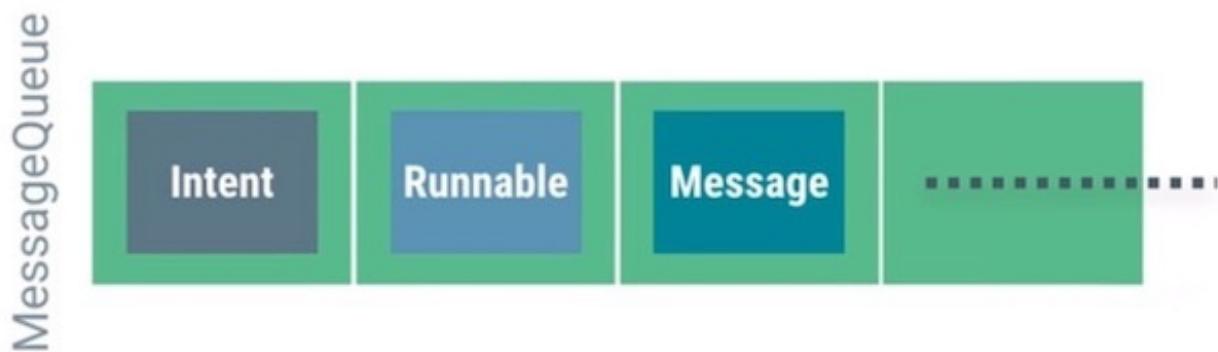
## Handler

Adds messages to a queue



**MessageQueue:** 使用Intent, Message, Runnable作为任务的载体在不同的线程之间进行传递。

# Intent / runnable / message



把上面三个组件打包到一起进行协作，这就是**HandlerThread**

## HandlerThread

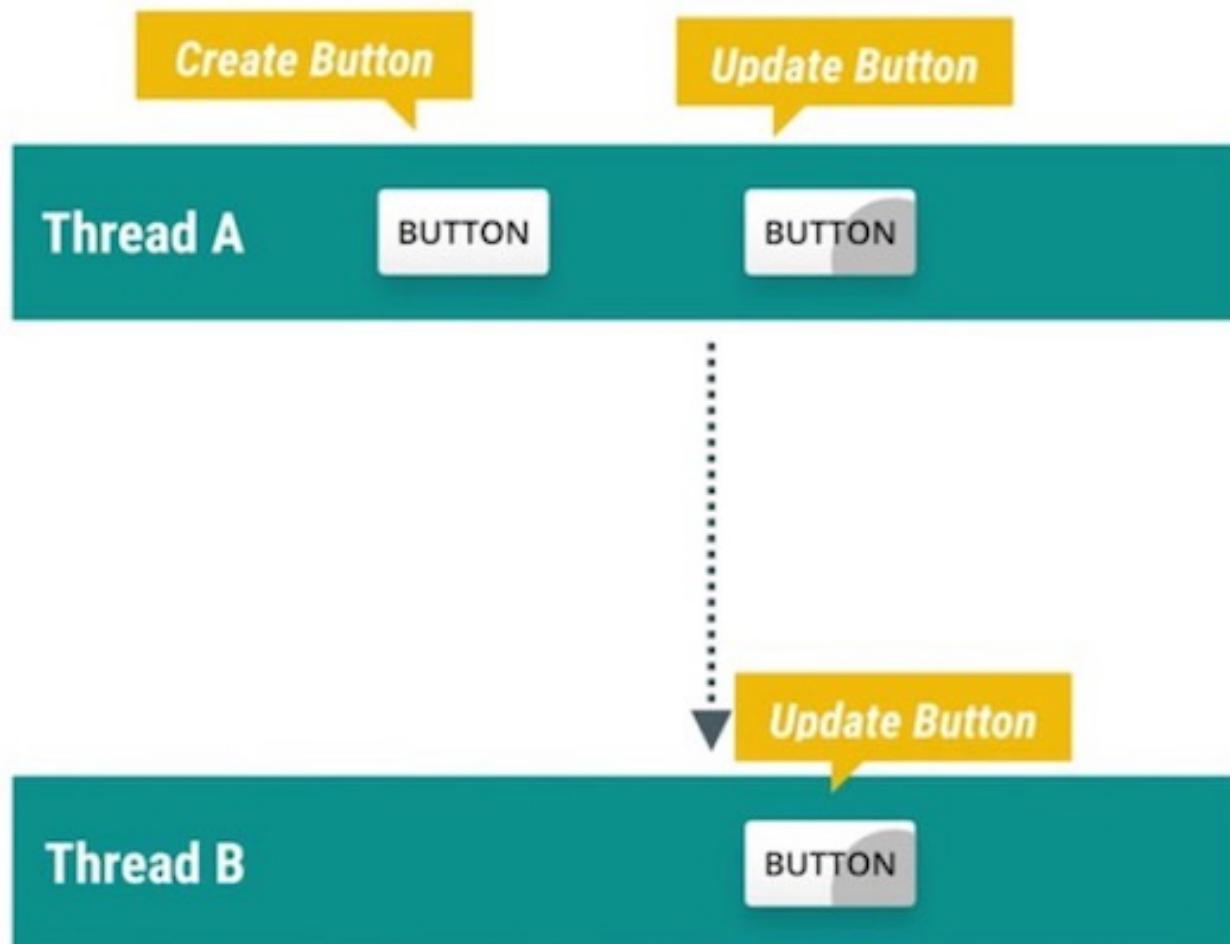


我们知道，当程序被启动，系统会帮忙创建进程以及相应的主线程，而这个主线程其实就是一个HandlerThread。这个主线程会需要处理系统事件，输入事件，系统回调的任务，UI绘制等等任务，为了避免主线程任务过重，我们就会需要不断的开启新的工作线程来处理那些子任务。

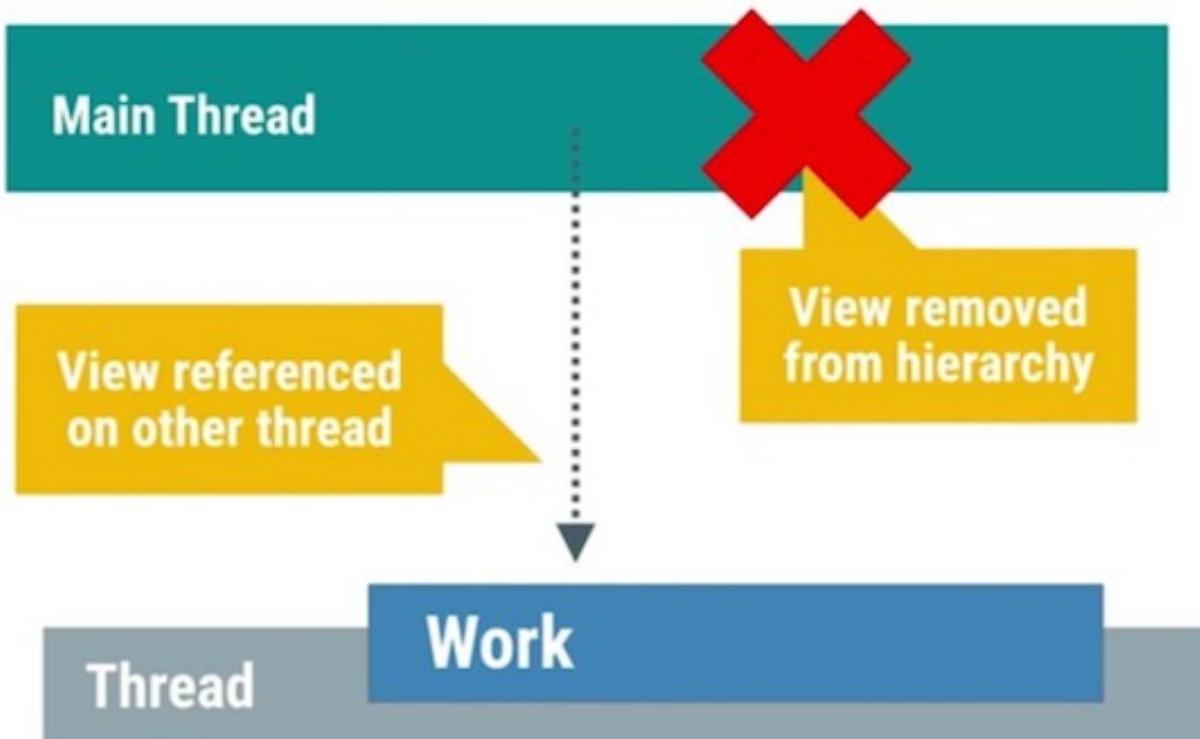
## 3)Memory & Threading

增加并发的线程数会导致内存消耗的增加，平衡好这两者的关系是非常重要的。我们知道，多线程并发访问同一块内存区域有可能带来很多问题，例如读写的权限争夺问题，[ABA问题](#)等等。为了解决这些问题，我们会需要引入锁的概念。

在Android系统中也无法避免因为多线程的引入而导致出现诸如上文提到的种种问题。Android UI对象的创建，更新，销毁等等操作都默认是执行在主线程，但是如果我们在非主线程对UI对象进行操作，程序将可能出现异常甚至是崩溃。



另外，在非UI线程中直接持有UI对象的引用也很可能出现问题。例如Work线程中持有某个UI对象的引用，在Work线程执行完毕之前，UI对象在主线程中被从ViewHierarchy中移除了，这个时候UI对象的任何属性都已经不再可用，另外对这个UI对象的更新操作也都没有任何意义了，因为它已经从ViewHierarchy中被移除，不再绘制到画面上了。



不仅如此，View对象本身对所属的Activity是有引用关系的，如果工作线程持续保有View的引用，这就可能导致Activity无法完全释放。除了直接显式的引用关系可能导致内存泄露之外，我们还需要特别留意隐式的引用关系也可能导致泄露。例如通常我们会看到在Activity里面定义的一个AsyncTask，这种类型的AsyncTask与外部的Activity是存在隐式引用关系的，只要Task没有结束，引用关系就会一直存在，这很容易导致Activity的泄漏。更糟糕的情况是，它不仅仅发生了内存泄漏，还可能导致程序异常或者崩溃。

```
public class MainActivity extends Activity
{
 public class MyAsyncTask extends AsyncTask<Void, Void, String>
 {
 @Override protected String doInBackground(Void... params)
 {
 ...
 }

 @Override protected void onPostExecute(String result)
 {
 ...
 }
 }
}
```

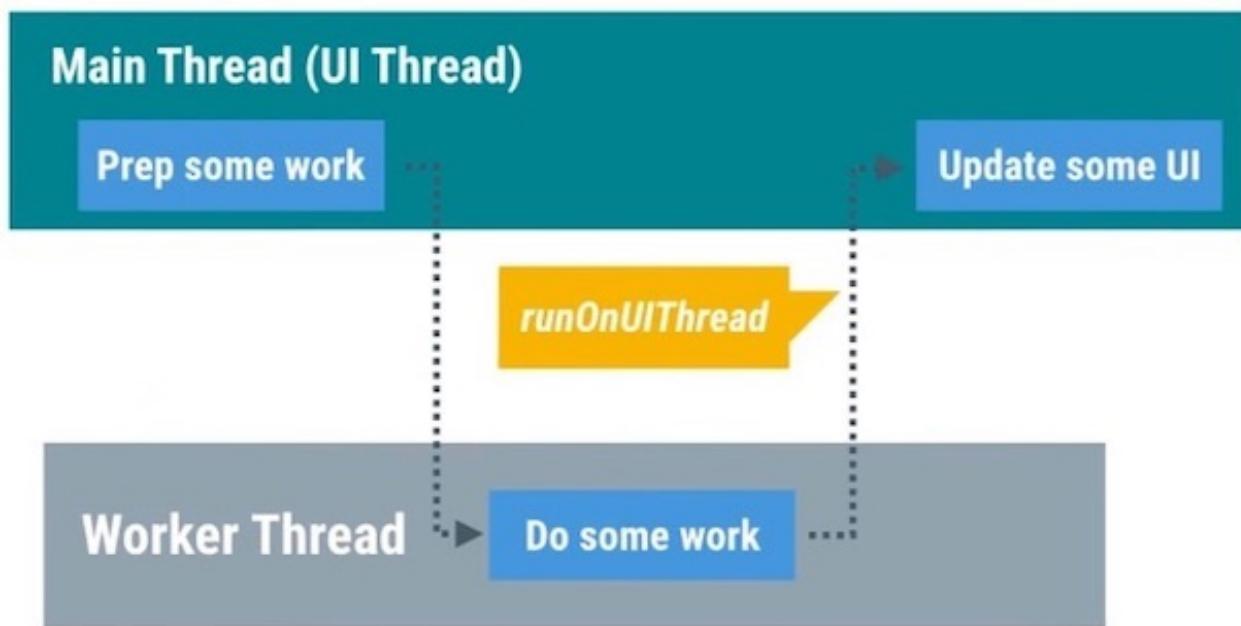
**Inner class contains Implicit reference to outer Activity**

为了解决上面的问题，我们需要谨记的原则就是：不要在任何非UI线程里面去持有UI对象的引用。系统为了确保所有的UI对象都只会被UI线程所进行创建，更新，销毁的操作，特地设计了对应的工作机制(当Activity被销毁的时候，由该Activity所触发的非UI线程都将无法对UI对象进行操作，否者就会抛出程序执行异常的错误)来防止UI对象被错误的使用。

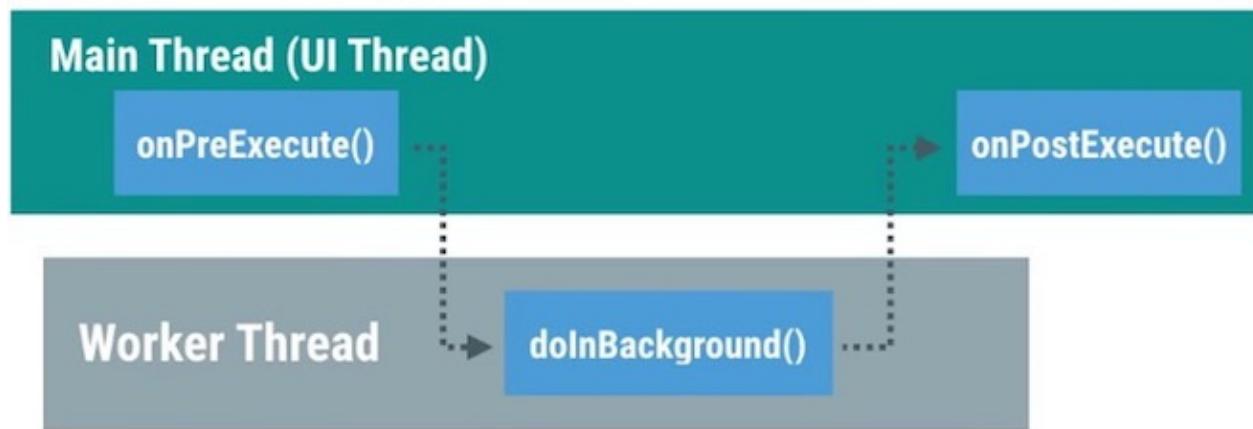
## 4) Good AsyncTask Hunting

AsyncTask是一个让人既爱又恨的组件，它提供了一种简便的异步处理机制，但是它又同时引入了一些令人厌恶的麻烦。一旦对AsyncTask使用不当，很可能对程序的性能带来负面影响，同时还可能导致内存泄露。

举个例子，常遇到的一个典型的使用场景：用户切换到某个界面，触发了界面上的图片的加载操作，因为图片的加载相对来说耗时比较长，我们需要在子线程中处理图片的加载，当图片在子线程中处理完成之后，再把处理好的图片返回给主线程，交给UI更新到画面上。

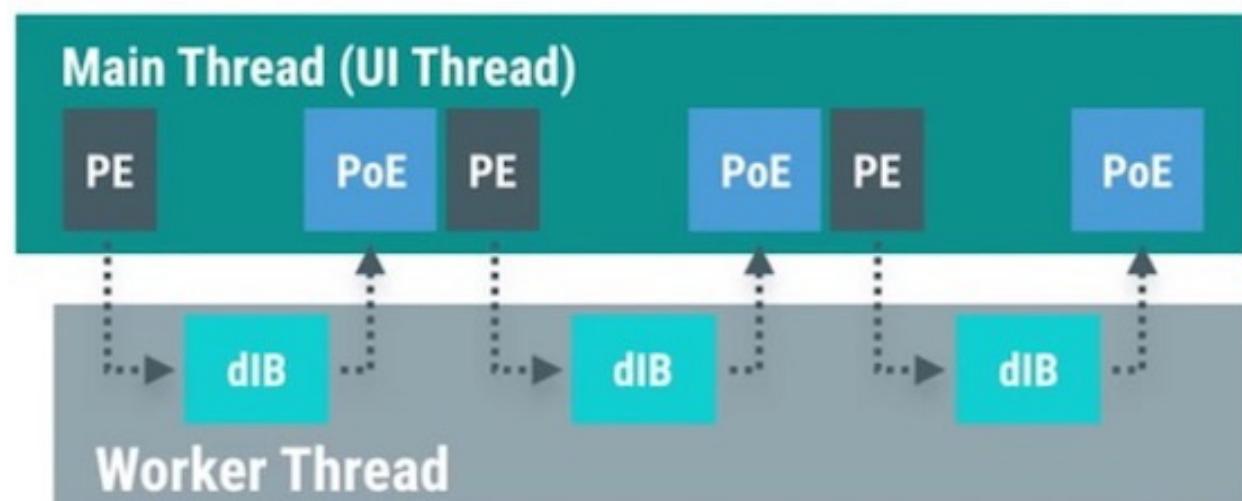


AsyncTask的出现就是为了快速的实现上面的使用场景，AsyncTask把在主线程里面的准备工作放到 `onPreExecute()` 方法里面进行执行，`doInBackground()` 方法执行在工作线程中，用来处理那些繁重的任务，一旦任务执行完毕，就会调用 `onPostExecute()` 方法返回到主线程。

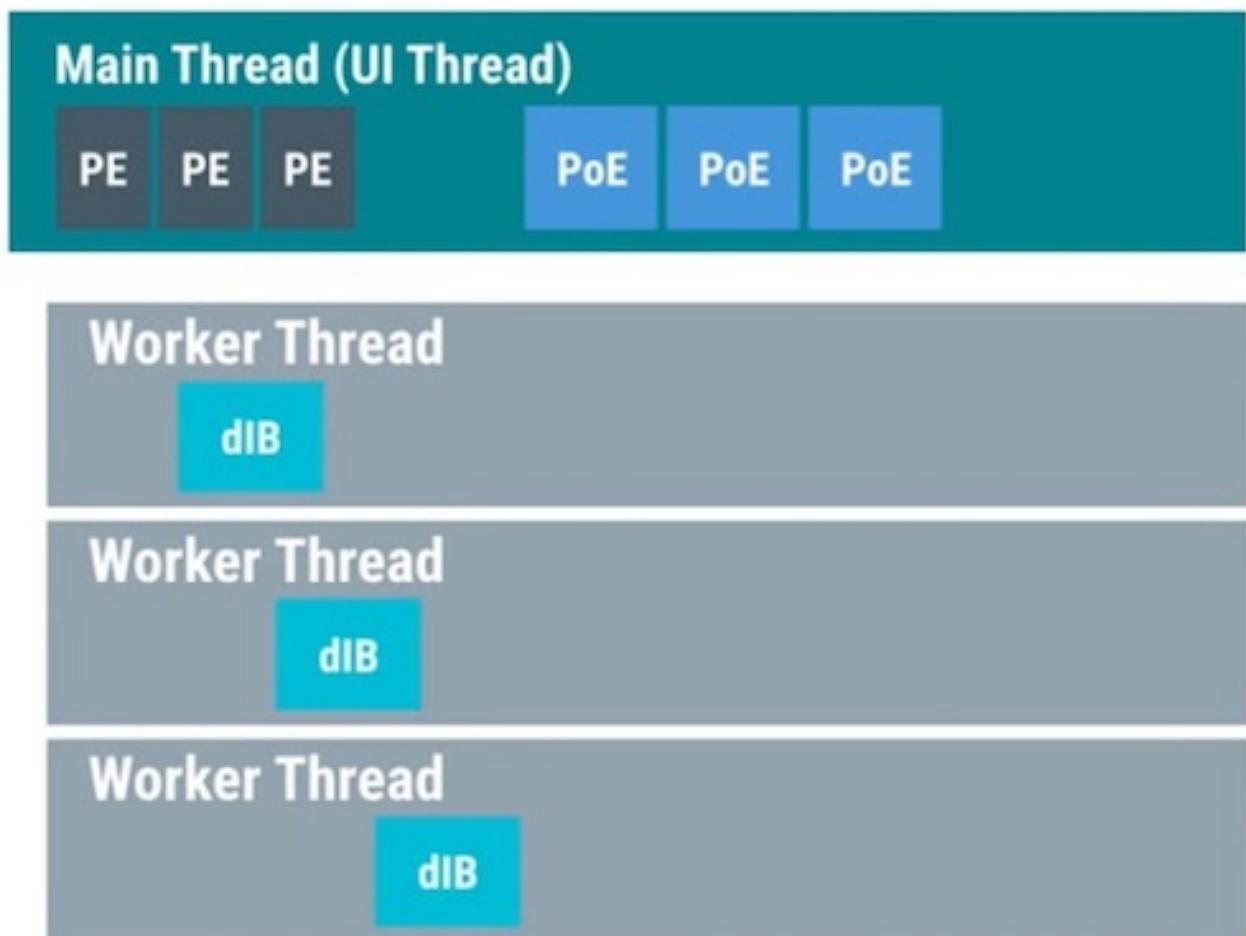


使用`AsyncTask`需要注意的问题有哪些呢？请关注以下几点：

- 首先，默认情况下，所有的`AsyncTask`任务都是被线性调度执行的，他们处在同一个任务队列当中，按顺序逐个执行。假设你按照顺序启动20个`AsyncTask`，一旦其中的某个`AsyncTask`执行时间过长，队列中的其他剩余`AsyncTask`都处于阻塞状态，必须等到该任务执行完毕之后才能够有机会执行下一个任务。情况如下图所示：



为了解决上面提到的线性队列等待的问题，我们可以使用 `AsyncTask.executeOnExecutor()` 强制指定`AsyncTask`使用线程池并发调度任务。



- 其次，如何才能够真正的取消一个AsyncTask的执行呢？我们知道AsyncTasks有提供cancel()的方法，但是这个方法实际上做了什么事情呢？线程本身并不具备中止正在执行的代码的能力，为了能够让一个线程更早的被销毁，我们需要在doInBackground()的代码中不断的添加程序是否被中止的判断逻辑，如下图所示：

```
DoInBackground(..)
{
 //Doing some stuff
 If (isCancelled()) {...} //Oh noez, we done, clean up

 For (i < objs.length)
 If (isCancelled()) //Oh noez, we done, clean up
 {...}

}
```

一旦任务被成功中止，AsyncTask就不会继续调用onPostExecute()，而是通过调用onCancelled()的回调方法反馈任务执行取消的结果。我们可以根据任务回调到哪个方法（是onPostExecute还是onCancelled）来决定是对UI进行正常的更新还是把对应的任務所占用的内存进行销毁等。

- 最后，使用AsyncTask很容易导致内存泄漏，一旦把AsyncTask写成Activity的内部类的形式就很容易因为AsyncTask生命周期的不确定而导致Activity发生泄漏。

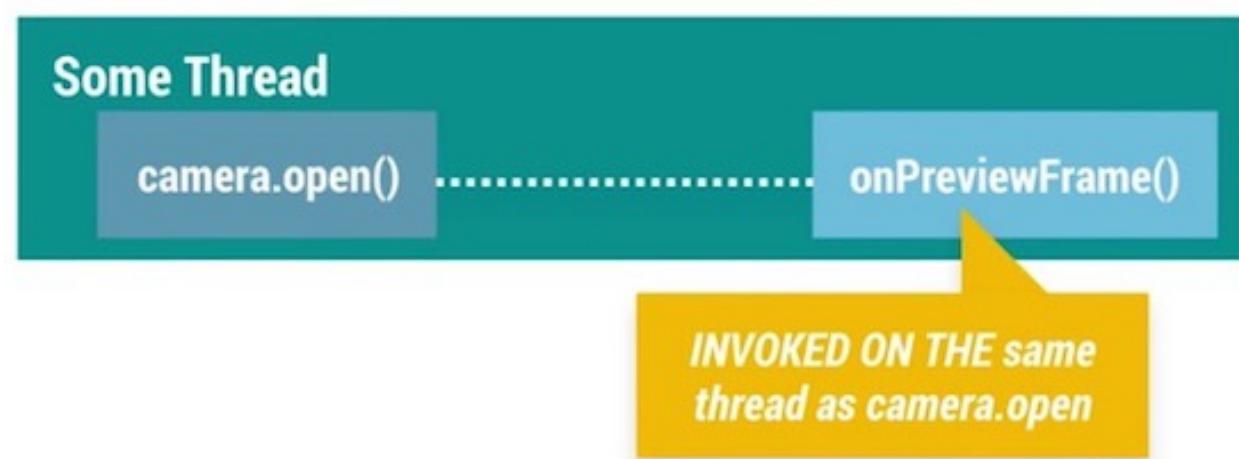
```
public class MainActivity extends Activity
{
 public class MyAsyncTask extends AsyncTask<Void, ...
 {
 @Override protected String doInBackground(Void ...
 {...}
 @Override protected void onPostExecute(String res ...
 }
}
```

Inner class contains  
Implicit reference to outer Activity

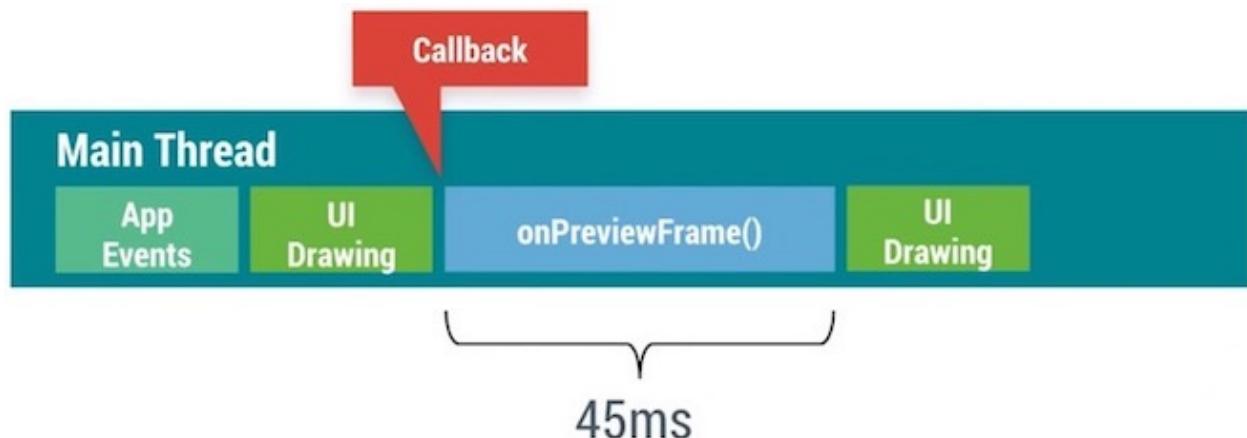
综上所述，AsyncTask虽然提供了一种简单便捷的异步机制，但是我们还是很有必要特别关注到他的缺点，避免出现因为使用错误而导致的严重系统性能问题。

## 5) Getting a HandlerThread

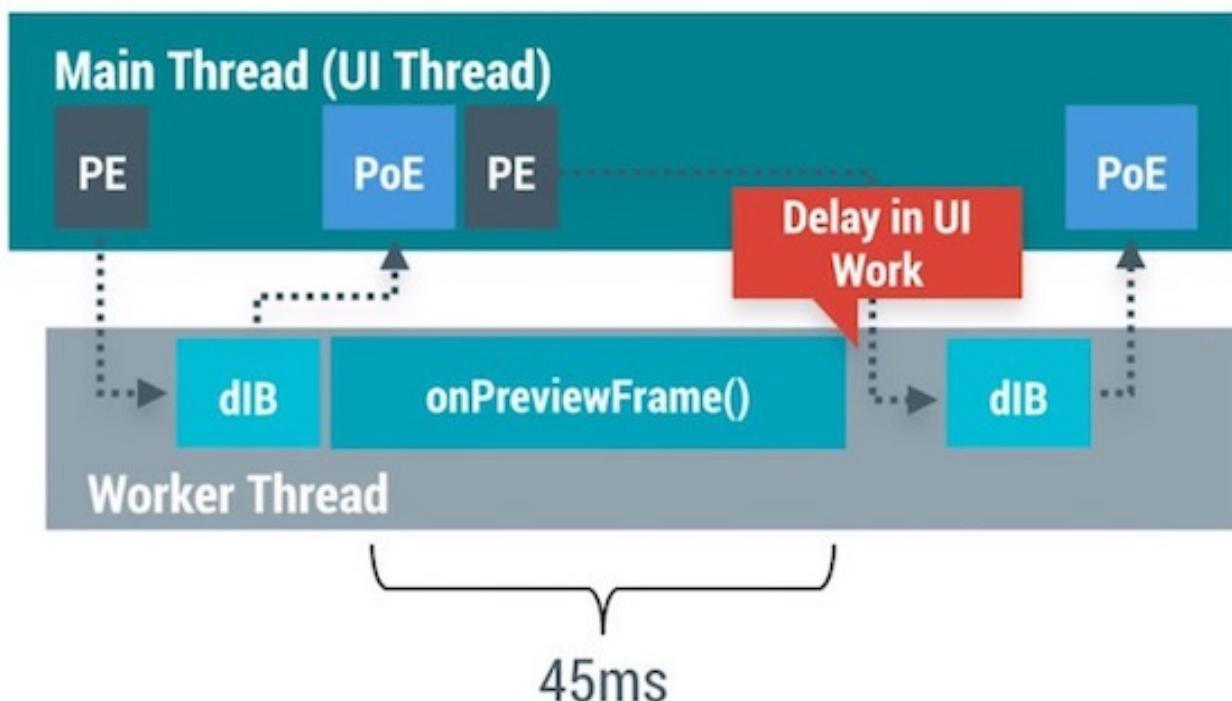
大多数情况下，AsyncTask都能够满足多线程并发的场景需要（在工作线程执行任务并返回结果到主线程），但是它并不是万能的。例如打开相机之后的预览帧数据是通过 `onPreviewFrame()` 的方法进行回调的，`onPreviewFrame()` 和 `open()` 相机的方法是执行在同一个线程的。



如果这个回调方法执行在UI线程，那么在 `onPreviewFrame()` 里面将要执行的数据转换操作将和主线程的界面绘制，事件传递等操作争抢系统资源，这就有可能影响到主界面的表现性能。



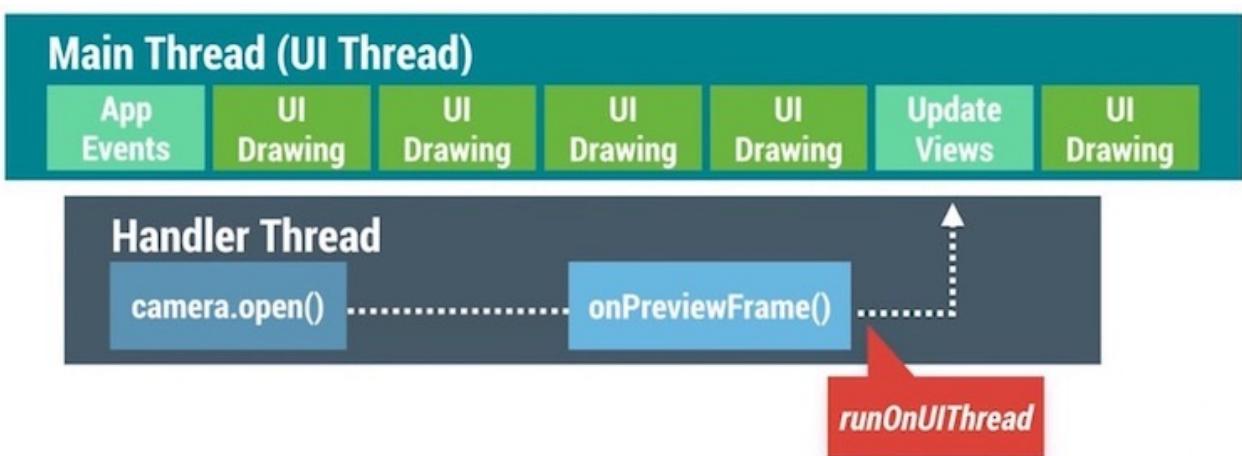
我们需要确保 `onPreviewFrame()` 执行在工作线程。如果使用`AsyncTask`, 会因为`AsyncTask`默认的线性执行的特性(即使换成并发执行)会导致因为无法把任务及时传递给工作线程而导致任务在主线程中被延迟，直到工作线程空闲，才可以把任务切换到工作线程中进行执行。



所以我们需要的是一个执行在工作线程，同时又能够处理队列中的复杂任务的功能，而`HandlerThread`的出现就是为了实现这个功能的，它组合了`Handler`, `MessageQueue`, `Looper`实现了一个长时间运行的线程，不断的从队列中获取任务进行执行的功能。

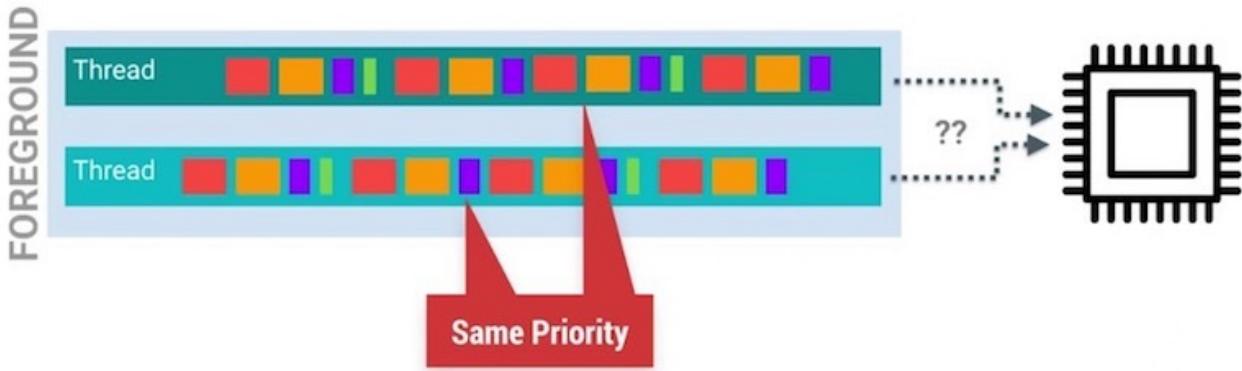


回到刚才的处理相机回调数据的例子，使用HandlerThread我们可以把 `open()` 操作与 `onPreviewFrame()` 的操作执行在同一个线程，同时还避免了AsyncTask的弊端。如果需要在 `onPreviewFrame()` 里面更新UI，只需要调用 `runOnUiThread()` 方法把任务回调给主线程就够了。



HandlerThread比较合适处理那些在工作线程执行，需要花费时间偏长的任务。我们只需要把任务发送给HandlerThread，然后就只需要等待任务执行结束的时候通知返回到主线程就好了。

另外很重要的一点是，一旦我们使用了HandlerThread，需要特别注意给HandlerThread设置不同的线程优先级，CPU会根据设置的不同线程优先级对所有的线程进行调度优化。



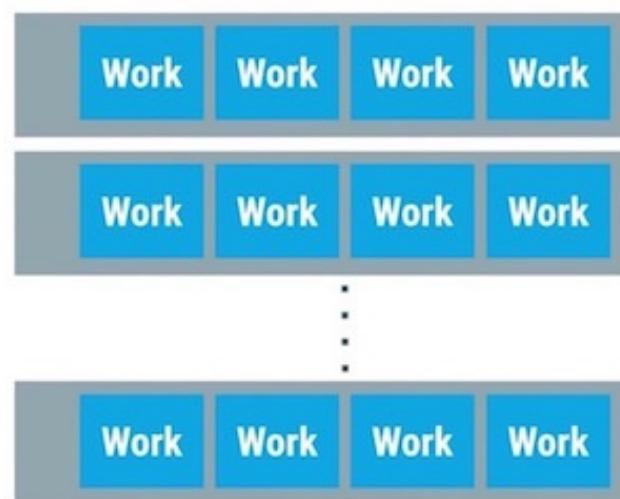
掌握HandlerThread与AsyncTask之间的优缺点，可以帮助我们选择合适的方案。

## 6) Swimming in Threadpools

线程池适合用在把任务进行分解，并发进行执行的场景。通常来说，系统里面会针对不同的任务设置一个单独的守护线程用来专门处理这项任务。例如使用Networking Thread用来专门处理网络请求的操作，使用IO Thread用来专门处理系统的I/O操作。针对那些场景，这样设计是没有问题的，因为对应的任务单次执行的时间并不长而且可以是顺序执行的。但是这种专属的单线程并不能满足所有的情况，例如我们需要一次性decode 40张图片，每个线程需要执行4ms的时间，如果我们使用专属单线程的方案，所有图片执行完毕会需要花费160ms( $40 \times 4$ )，但是如果我们创建10个线程，每个线程执行4个任务，那么我们就只需要16ms就能够把所有的图片处理完毕。

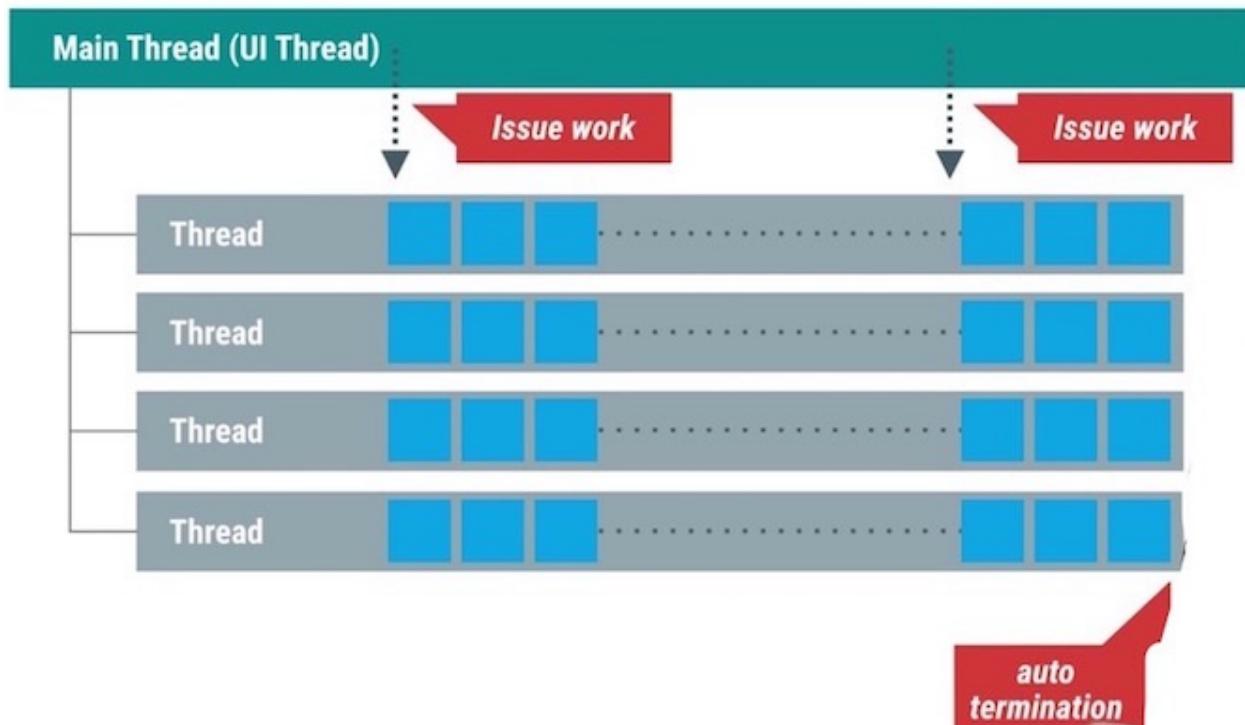


Parallel = 16ms

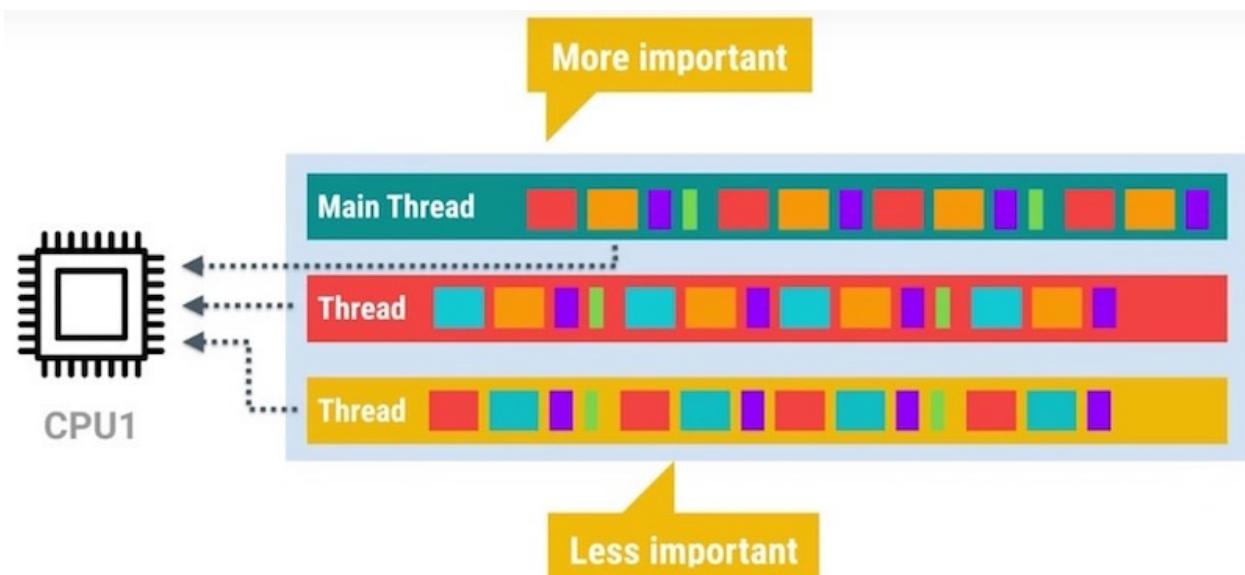


为了能够实现上面的线程池模型，系统为我们提供了ThreadPoolExecutor帮助类来简化实现，剩下需要做的就只是对任务进行分解就好了。

# ThreadPoolExecutor



使用线程池需要特别注意同时并发线程数量的控制，理论上来说，我们可以设置任意你想要的并发数量，但是这样做非常的不好。因为CPU只能同时执行固定数量的线程数，一旦同时并发的线程数量超过CPU能够同时执行的阈值，CPU就需要花费精力来判断到底哪些线程的优先级比较高，需要在不同的线程之间进行调度切换。



一旦同时并发的线程数量达到一定的量级，这个时候CPU在不同线程之间进行调度的时间就可能过长，反而导致性能严重下降。另外需要关注的一点是，每开一个新的线程，都会耗费至少64K+的内存。为了能够方便的对线程数量进行控制，ThreadPoolExecutor为我们提供了初始化的并发线程数量，以及最大的并发数量进行设置。

```

private static int NUMBER_OF_CORES = Runtime.getRuntime().availableProcessors();

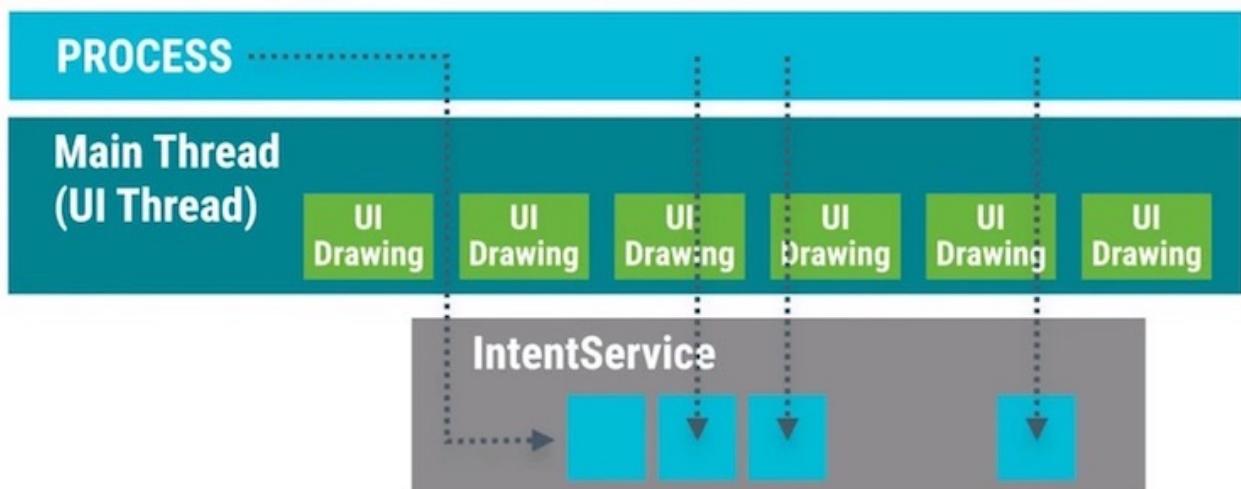
mDecodeThreadPool = new ThreadPoolExecutor(
 NUMBER_OF_CORES>>1, // Initial pool size
 NUMBER_OF_CORES, // Max pool size
 KEEP_ALIVE_TIME, // Keep alive time
 KEEP_ALIVE_TIME_UNIT, // Keep alive units (seconds etc)
 mDecodeWorkQueue);
}

```

另外需要关注的一个问题是：`Runtime.getRuntime().availableProcesser()` 方法并不可靠，他返回的值并不是真实的CPU核心数，因为CPU会在某些情况下选择对部分核心进行睡眠处理，在这种情况下，返回的数量就只能是激活的CPU核心数。

## 7) The Zen of IntentService

默认的Service是执行在主线程的，可是通常情况下，这很容易影响到程序的绘制性能(抢占了主线程的资源)。除了前面介绍过的AsyncTask与HandlerThread，我们还可以选择使用IntentService来实现异步操作。IntentService继承自普通Service同时又在内部创建了一个HandlerThread，在`onHandleIntent()` 的回调里面处理扔到IntentService的任务。所以IntentService就不仅仅具备了异步线程的特性，还同时保留了Service不受主页面生命周期影响的特点。



如此一来，我们可以在IntentService里面通过设置闹钟间隔性的触发异步任务，例如刷新数据，更新缓存的图片或者是分析用户操作行为等等，当然处理这些任务需要小心谨慎。

使用IntentService需要特别留意以下几点：

- 首先，因为IntentService内置的是HandlerThread作为异步线程，所以每一个交给IntentService的任务都将以队列的方式逐个被执行到，一旦队列中有某个任务执行时间过长，那么就会导致后续的任务都会被延迟处理。
- 其次，通常使用到IntentService的时候，我们会结合使用BroadcastReceiver把工作

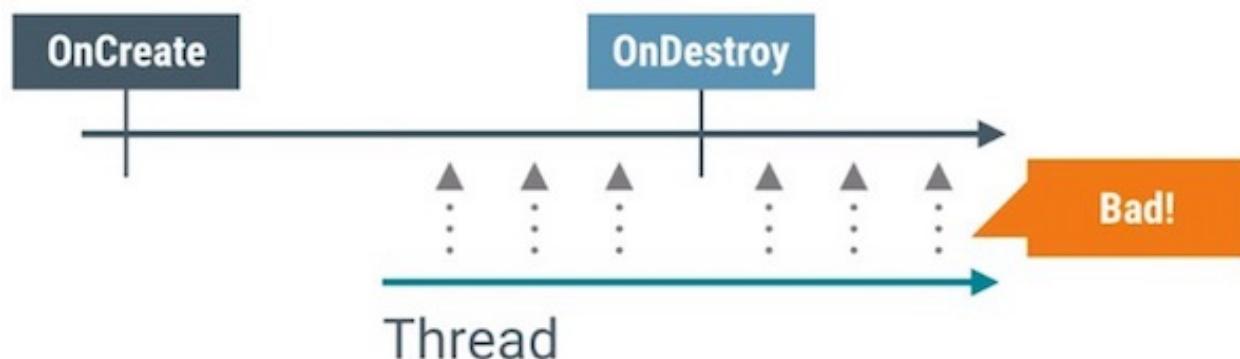
线程的任务执行结果返回给主UI线程。使用广播容易引起性能问题，我们可以使用 LocalBroadcastManager 来发送只在程序内部传递的广播，从而提升广播的性能。我们也可以使用 runOnUiThread() 快速回调到主UI线程。

- 最后，包含正在运行的IntentService的程序相比起纯粹的后台程序更不容易被系统杀死，该程序的优先级是介于前台程序与纯后台程序之间的。

## 8) Threading and Loaders

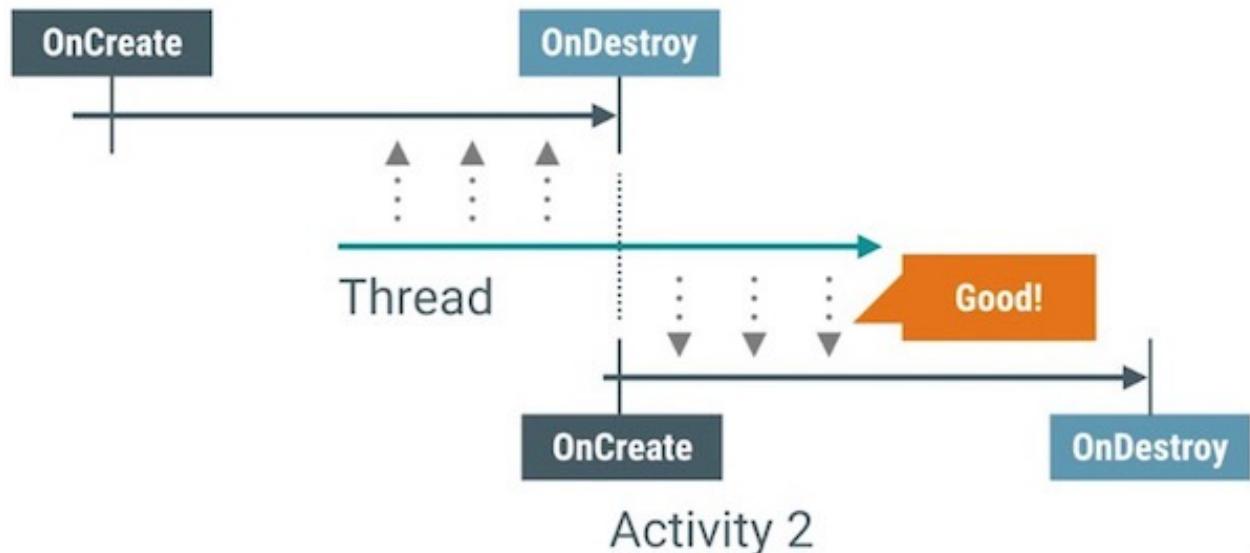
当启动工作线程的Activity被销毁的时候，我们应该做点什么呢？为了方便的控制工作线程的启动与结束，Android为我们引入了Loader来解决这个问题。我们知道Activity有可能因为用户的主动切换而频繁的被创建与销毁，也有可能是因为类似屏幕发生旋转等被动原因而销毁再重建。在Activity不停的创建与销毁的过程当中，很有可能因为工作线程持有Activity的View而导致内存泄漏(因为工作线程很可能持有View的强引用，另外工作线程的生命周期还无法保证和Activity的生命周期一致，这样就容易发生内存泄漏了)。除了可能引起内存泄漏之外，在Activity被销毁之后，工作线程还继续更新视图是没有意义的，因为此时视图已经不在界面上显示了。

### Activity 1



Loader的出现就是为了确保工作线程能够和Activity的生命周期保持一致，同时避免出现前面提到的问题。

## Activity 1

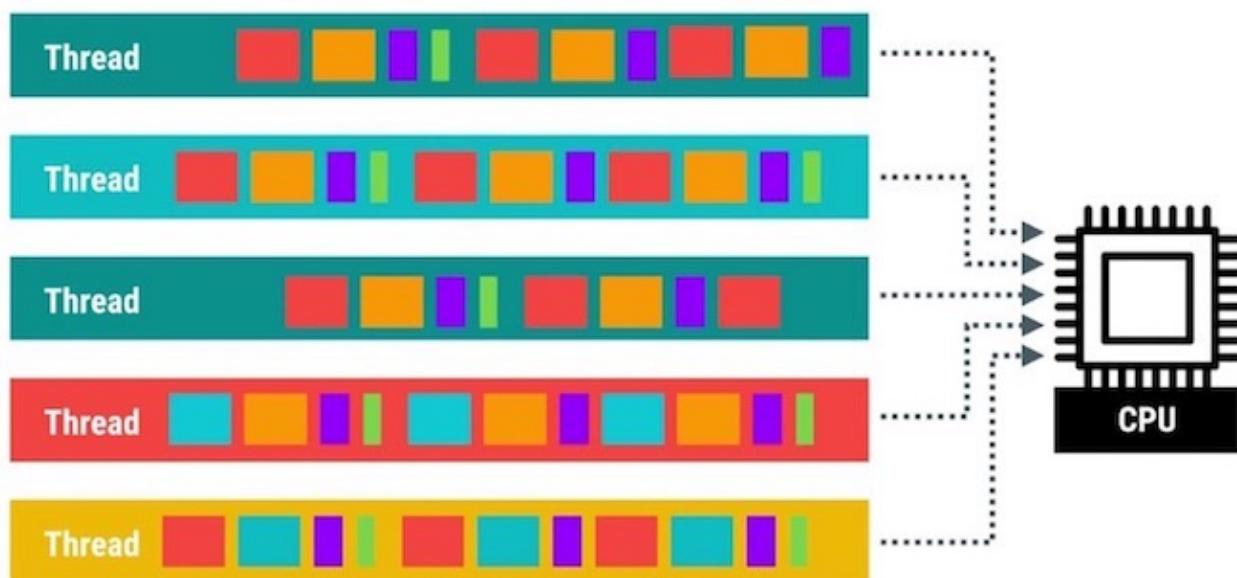


LoaderManager会对查询的操作进行缓存，只要对应Cursor上的数据源没有发生变化，在配置信息发生改变的时候(例如屏幕的旋转)，Loader可以直接把缓存的数据回调到 `onLoadFinished()`，从而避免重新查询数据。另外系统会在Loader不再需要使用到的时候(例如使用Back按钮退出当前页面)回调 `onLoaderReset()` 方法，我们可以在这里做数据的清除等等操作。

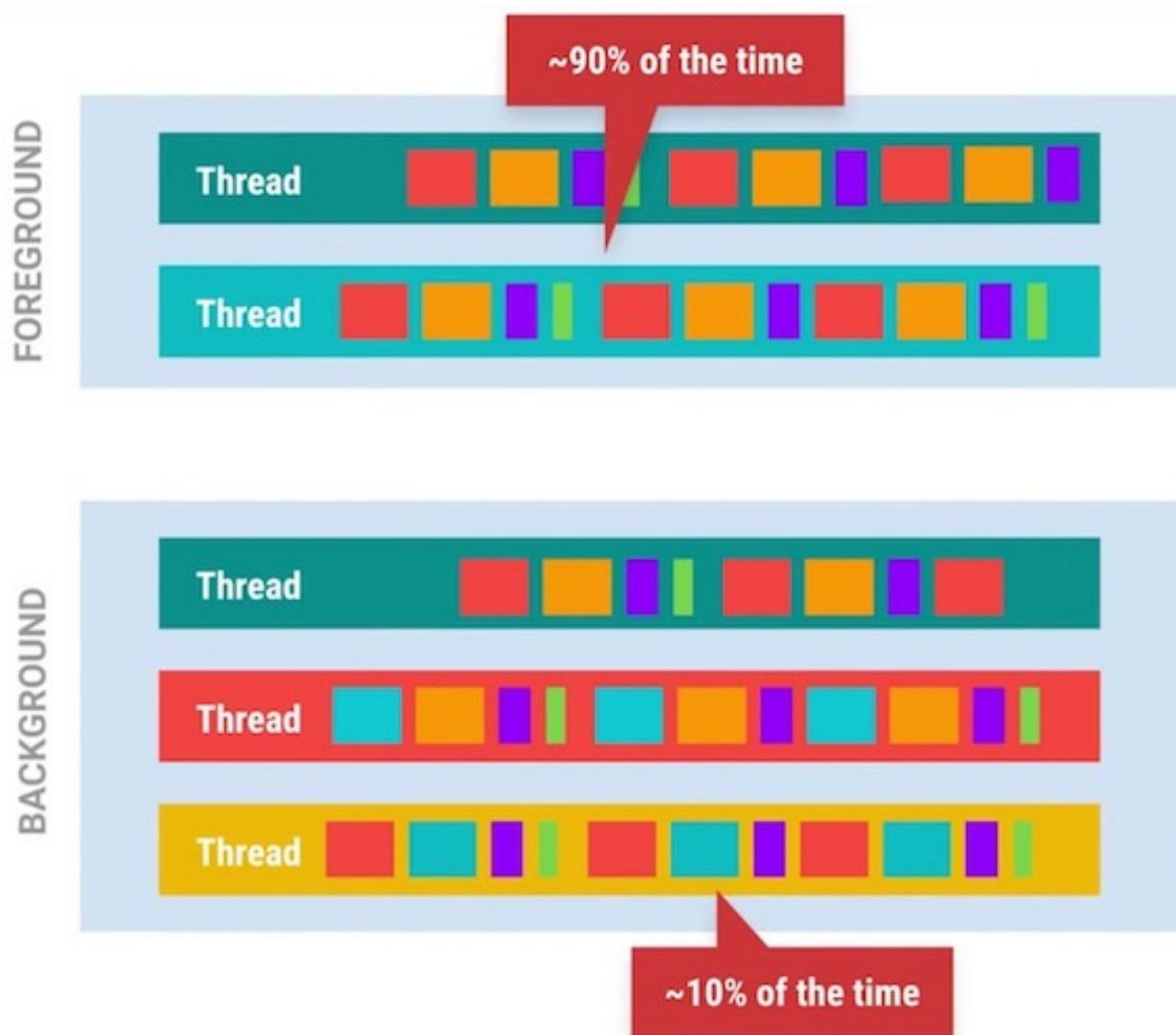
在Activity或者Fragment中使用Loader可以方便的实现异步加载的框架，Loader有诸多优点。但是实现Loader的这套代码还是稍微有点点复杂，Android官方为我们提供了使用Loader的[示例代码](#)进行参考学习。

## 9) The Importance of Thread Priority

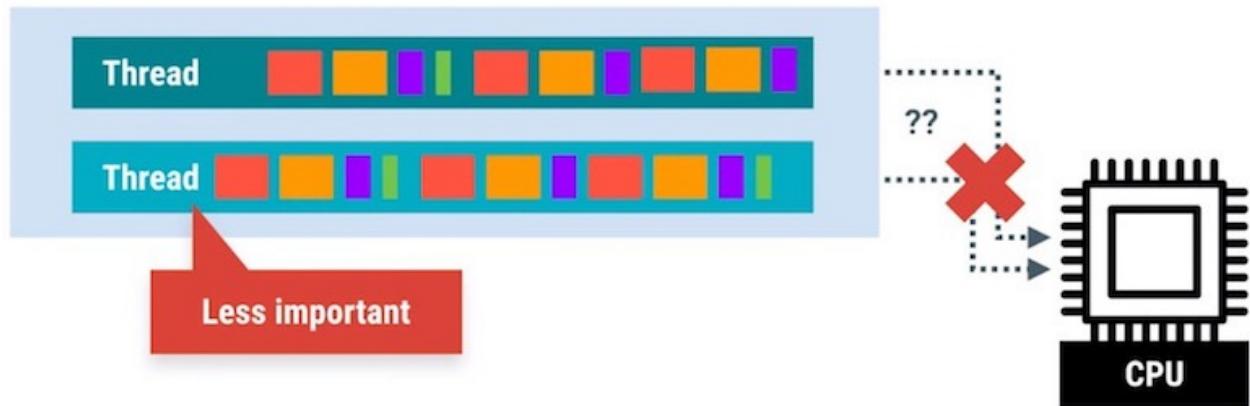
理论上来说，我们的程序可以创建出非常多的子线程一起并发执行的，可是基于CPU时间片轮转调度的机制，不可能所有的线程都可以同时被调度执行，CPU需要根据线程的优先级赋予不同的时间片。



Android系统会根据当前运行的可见的程序和不可见的后台程序对线程进行归类，划分为foreground的那部分线程会大致占用掉CPU的90%左右的时间片，background的那部分线程就总共只能分享到5%-10%左右的时间片。之所以设计成这样是因为foreground的程序本身的优先级就更高，理应得到更多的执行时间。



默认情况下，新创建的线程的优先级默认和创建它的母线程保持一致。如果主UI线程创建出了几十个工作线程，这些工作线程的优先级就默认和主线程保持一致了，为了不让新创建的工作线程和主线程抢占CPU资源，需要把这些线程的优先级进行降低处理，这样才能给帮助CPU识别主次，提高主线程所能得到的系统资源。



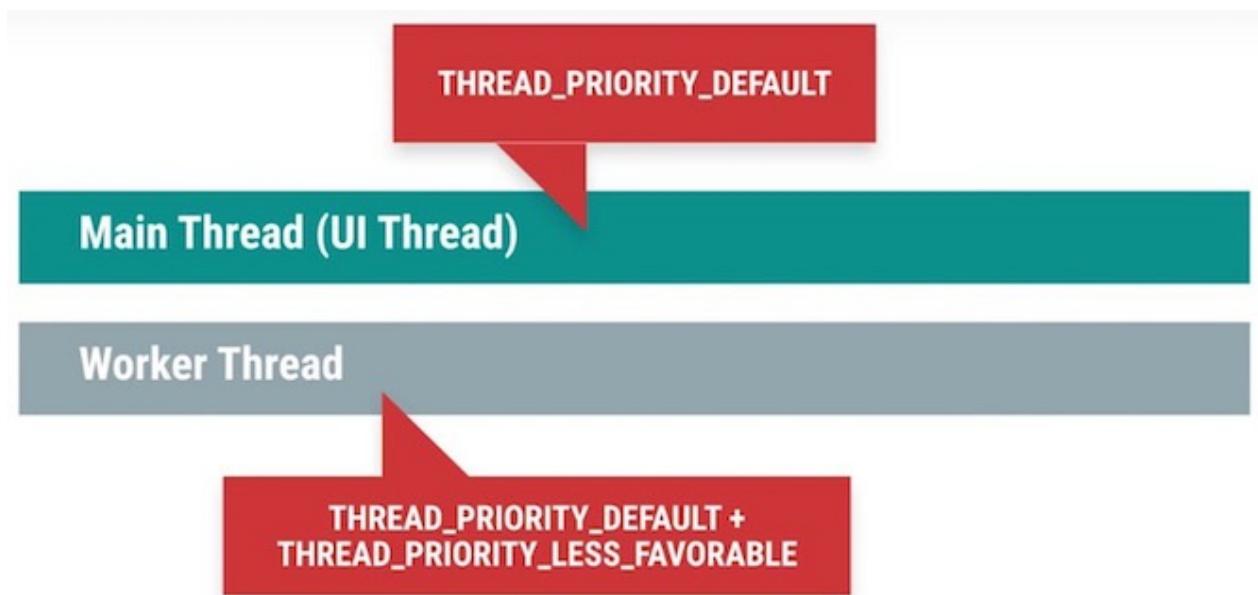
在Android系统里面，我们可以通过 `android.os.Process.setThreadPriority(int)` 设置线程的优先级，参数范围从-20到24，数值越小优先级越高。Android系统还为我们提供了以下的一些预设值，我们可以通过给不同的工作线程设置不同数值的优先级来达到更细粒度的控制。

| Base State enumeration         | Value |
|--------------------------------|-------|
| THREAD_PRIORITY_URGENT_AUDIO   | -19   |
| THREAD_PRIORITY_AUDIO          | -16   |
| THREAD_PRIORITY_URGENT_DISPLAY | -8    |
| THREAD_PRIORITY_DISPLAY        | -4    |
| THREAD_PRIORITY_FOREGROUND     | -2    |
| THREAD_PRIORITY_DEFAULT        | 0     |
| THREAD_PRIORITY_BACKGROUND     | 10    |
| THREAD_PRIORITY_LOWEST         | 19    |



| Modifiers                      | Value |
|--------------------------------|-------|
| THREAD_PRIORITY_MORE_FAVORABLE | -1    |
| THREAD_PRIORITY_LESS_FAVORABLE | +1    |

大多数情况下，新创建的线程优先级会被设置为默认的0，主线程设置为0的时候，新创建的线程还可以利用**THREAD\_PRIORITY\_LESS\_FAVORABLE**或者**THREAD\_PRIORITY\_MORE\_FAVORABLE**来控制线程的优先级。



Android系统里面的AsyncTask与IntentService已经默认帮助我们设置线程的优先级，但是对于那些非官方提供的多线程工具类，我们需要特别留意根据需要自己手动来设置线程的优先级。

```
/*
 * Creates a new asynchronous task. This constructor must be invoked on the UI thread.
 */
public AsyncTask() {
 mWorker = new WorkerRunnable<Params, Result>() {
 public Result call() throws Exception {
 mTaskInvoked.set(true);

 Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
 //noinspection unchecked
 return postResult(doInBackground(mParams));
 }
 };
}
```

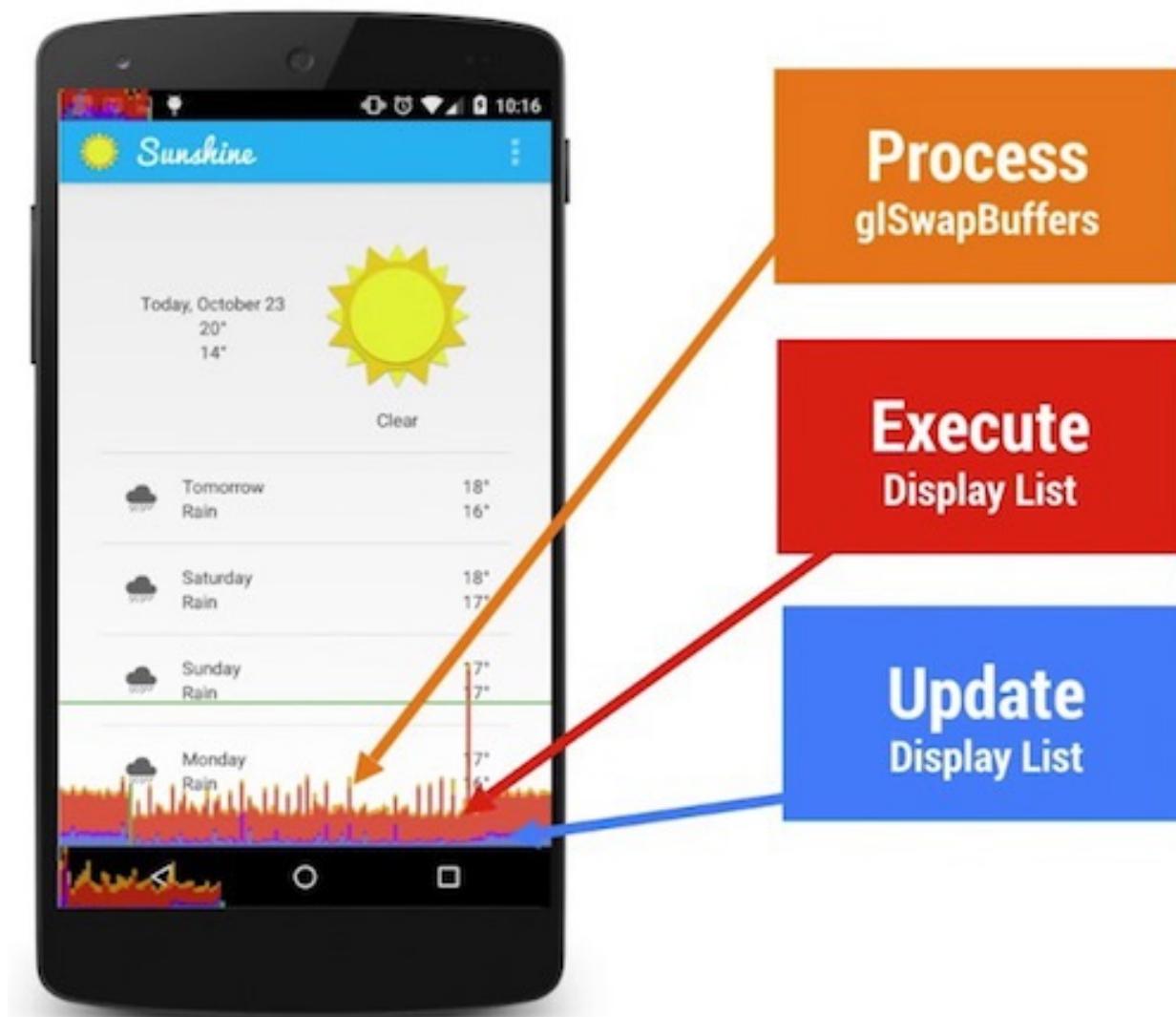
```
/**
 * Handy class for starting a new thread that has a looper. The looper can then be
 * used to create handler classes. Note that start() must still be called.
 */
public class HandlerThread extends Thread {
 int mPriority;
 int mTid = -1;
 Looper mLooper;

 public HandlerThread(String name) {
 super(name);
 mPriority = Process.THREAD_PRIORITY_DEFAULT;
 }

 /**
 * Constructs a HandlerThread.
 * @param name
 * @param priority The priority to run the thread at. The value supplied must be
 * {@link android.os.Process} and not from java.lang.Thread.
 */
 public HandlerThread(String name, int priority) {
 super(name);
 mPriority = priority;
 }
}
```

## 10) Profile GPU Rendering : M Update

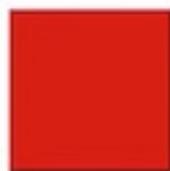
从Android M系统开始，系统更新了GPU Profiling的工具来帮助我们定位UI的渲染性能问题。早期的CPU Profiling工具只能粗略的显示出Process, Execute, Update三大步骤的时间耗费情况。



但是仅仅显示三大步骤的时间耗费情况，还是不太能够清晰帮助我们定位具体的程序代码问题，所以在Android M版本开始，GPU Profiling 工具把渲染操作拆解成如下8个详细的步骤进行显示。



Swap Buffers



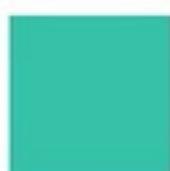
Command Issue



Sync & Upload



Draw



Measure / Layout



Animation

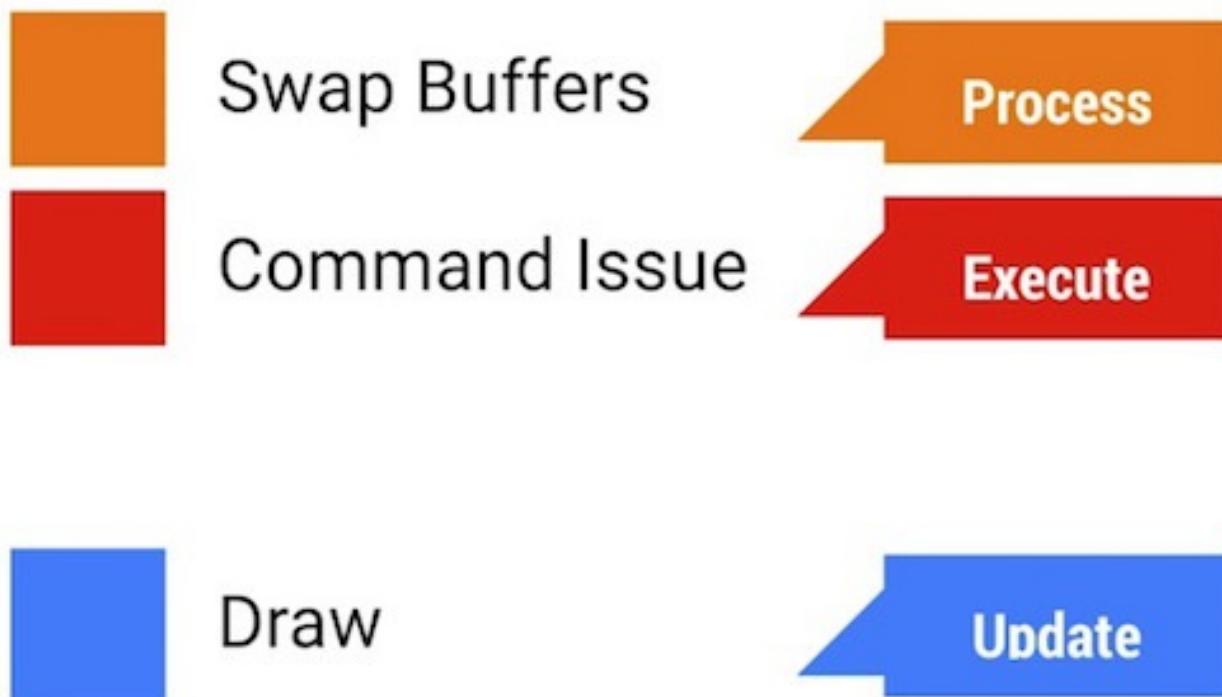


Input Handling



Misc Time / Vsync Delay

旧版本中提到的Proces, Execute, Update还是继续得到了保留，他们的对应关系如下：



接下去我们看下其他五个步骤分别代表了什么含义：

- **Sync & Upload:** 通常表示的是准备当前界面上有待绘制的图片所耗费的时间，为了减少该段区域的执行时间，我们可以减少屏幕上的图片数量或者是缩小图片本身的大  
小。
- **Measure & Layout:** 这里表示的是布局的onMeasure与onLayout所花费的时间，一  
旦时间过长，就需要仔细检查自己的布局是不是存在严重的性能问题。
- **Animation:** 表示的是计算执行动画所需要花费的时间，包含的动画有  
ObjectAnimator, ViewPropertyAnimator, Transition等等。一旦这里的执行时间过  
长，就需要检查是不是使用了非官方的动画工具或者是检查动画执行的过程中是不是  
触发了读写操作等等。
- **Input Handling:** 表示的是系统处理输入事件所耗费的时间，粗略等于对于的事件处  
理方法所执行的时间。一旦执行时间过长，意味着在处理用户的输入事件的地方执行  
了复杂的操作。
- **Misc/Vsync Delay:** 如果稍加注意，我们可以在开发应用的Log日志里面看到这样  
一行提示：`I/Choreographer(691): Skipped XXX frames! The application may be doing  
too much work on its main thread`。这意味着我们在主线程执行了太多的任务，导致  
UI渲染跟不上vSync的信号而出现掉帧的情况。

上面八种不同的颜色区分了不同的操作所耗费的时间，为了便于我们迅速找出那些有问题的步骤，GPU Profiling工具会显示16ms的阈值线，这样就很容易找出那些不合理的性能问题，再仔细看对应具体哪个步骤相对来说耗费时间比例更大，结合上面介绍的细化步  
骤，从而快速定位问题，修复问题。

首发于CSDN：Android性能优化典范（五）





# Android APP终极瘦身指南

来源:[jayfeng.com](http://jayfeng.com)

## 前言

之前写了一篇《APK瘦身实践》侧重于实践和效果对比，后来受徐川老师点拨，建议改写成一篇更全面的瘦身终极杀招大全，深以为然，思考良久，新开一篇。

## 指南条例

### 第1条：使用一套资源

这是最基本的一条规则，但非常重要。

对于绝大部分APP来说，只需要取一套设计图就足够了。鉴于现在分辨率的趋势，建议取720p的资源，放到xhdpi目录。

相对于多套资源，只使用720P的一套资源，在视觉上差别不大，很多大公司的产品也是如此，但却能显著的减少资源占用大小，顺便也能减轻设计师的出图工作量了。

注意，这里不是说把不是xhdpi的目录都删除，而是强调保留一套设计资源就够了。

### 第2条：开启minifyEnabled混淆代码

在gradle使用minifyEnabled进行Proguard混淆的配置，可大大减小APP大小：

```
android {
 buildTypes {
 release {
 minifyEnabled true
 }
 }
}
```

在proguard中，是否保留符号表对APP的大小是有显著的影响的，可酌情不保留，但是建议尽量保留用于调试。

详细proguard的相关的配置和原理可自行查阅。

## 第3条：开启shrinkResources去除无用资源

在gradle使用shrinkResources去除无用资源，效果非常好。

```
android {
 buildTypes {
 release {
 shrinkResources true
 }
 }
}
```

## 第4条：删除无用的语言资源

大部分应用其实并不需要支持几十种语言的国际化支持。还好强大的gradle支持语言的配置，比如国内应用只支持中文：

```
android {
 defaultConfig {
 resConfigs "zh"
 }
}
```

## 第5条：使用tinypng有损压缩

android打包本身会对png进行无损压缩，所以使用像tinypng这样的有损压缩是有必要的。

重点是Tinypng使用智能有损压缩技术，以尽量少的失真换来图片大小的锐减，效果非常好，强烈推荐。

Tinypng的官方网站：<http://tinypng.com/>

## 第6条：使用jpg格式

如果对于非透明的大图，jpg将会比png的大小有显著的优势，虽然不是绝对的，但是通常会减小到一半都不止。

在启动页，活动页等之类的大图展示区采用jpg将是非常明智的选择。

## 第7条：使用webp格式

webp支持透明度，压缩比比jpg更高但显示效果却不输于jpg，官方评测quality参数等于75均衡最佳。

相对于jpg、png，webp作为一种新的图片格式，限于android的支持情况暂时还没用在手机端广泛应用起来。从Android 4.0+开始原生支持，但是不支持包含透明度，直到Android 4.2.1+才支持显示含透明度的webp，使用的时候要特别注意。

官方介绍：<https://developers.google.com/speed/webp/docs/precompiled>

## 第8条：缩小大图

如果经过上述步骤之后，你的工程里面还有一些大图，考虑是否有必要维持这样的尺寸，是否能适当的缩小。

事实上，由于设计师出图的原因，我们拿到的很多图片完全可以适当的缩小而对视觉影响是极小的。

## 第9条：覆盖第三库里的大图

有些第三库里引用了一些大图但是实际上并不会被我们用到，就可以考虑用1x1的透明图片覆盖。

你可能会有点不舒服，因为你的drawable下竟然包含了一些莫名其妙的名称的1x1图片…

## 第10条：删除armable-v7包下的so

基本上armable的so也是兼容armable-v7的，armable-v7a的库会对图形渲染方面有很大的改进，如果没有这方面的要求，可以精简。

这里不排除有极少数设备会Crash，可能和不同的so有一定的关系，请大家务必测试周全后再发布。

## 第11条：删除x86包下的so

与第十条不同的是，x86包下的so在x86型号的手机是需要的，如果产品没用这方面的要求也可以精简。

建议实际工作的配置是只保留armable、armable-x86下的so文件，算是一个折中的方案。

## 第12条：使用微信资源压缩打包工具

微信资源压缩打包工具通过短资源名称，采用 7 zip对APP进行极致压缩实现减小APP的目标，效果非常的好，强烈推荐。

详情参考：[Android资源混淆工具使用说明](#)

原理介绍：[安装包立减1M—微信Android资源混淆打包工具](#)

建议开启7zip，注意白名单的配置，否则会导致有些资源找不到，粗略配置如下，

```
<?xml version="1.0" encoding="UTF-8"?>
<resproguard>
 <!--default property to set -->
 <issue id="property" >
 <seventzip value= "true" />
 <!-- ... -->
 </issue>

 <issue id="whitelist" isactive="true">
 <path value ="com.xxx.yyy.R.drawable.emoji_*" />
 <path value ="com.xxx.yyy.... />
 </issue>

 <issue id ="compress" isactive="true">
 <!-- ... -->
 </issue>
</resproguard>
```

## 第13条：使用provided编译

对于一些库是按照需要动态的加载，可能在某些版本并不需要，但是代码又不方便去除否则会编译不过。

使用provided可以保证代码编译通过，但是实际打包中并不引用此第三方库，实现了控制APP大小的目标。

但是也需要开发者自己判断不引用这个第三方库时就不要执行到相关的代码，避免APP崩溃。

## 第14条：使用shape背景

特别是在扁平化盛行的当下，很多纯色的渐变的圆角的图片都可以用shape实现，代码灵活可控，省去了大量的背景图片。

## 第15条：使用着色方案

相信你的工程里也有很多selector文件，也有很多相似的图片只是颜色不同，通过着色方案我们能大大减轻这样的工作量，减少这样的文件。

借助于android support库可实现一个全版本兼容的着色方案，参考代码：[DrawableLess.java](#)

## 第16条：在线化素材库

如果你的APP支持素材库(比如聊天表情库)的话，考虑在线加载模式，因为往往素材库都有不小的体积。

这一步需要开发者实现在线加载，一方面增加代码的复杂度，一方面提高了APP的流量消耗，建议酌情选择。

## 第17条：避免重复库

避免重复库看上去是理所当然的，但是秘密总是藏的很深，一定要当心你引用的第三方库又引用了哪个第三方库，这就很容易出现功能重复的库了，比如使用了两个图片加载库：Glide和Picasso。

通过查看exploded-aar目录和External Libraries或者反编译生成的APK，尽量避免重复库的大小，减小APP大小。

## 第18条：使用更小的库

同样功能的库在大小上是不同的，甚至会悬殊很大。

如果并无对某个库特别需求而又对APP大小有严格要求的话，比较这些相同功能第三方库的大小，选择更小的库会减小APP大小。

## 第19条：支持插件化

过去的一年，插件化技术雨后春笋一样的都冒了出来，这些技术支持动态的加载代码和动态的加载资源，把APP的一部分分离出来了，对于业务庞大的项目来说非常有用，极大的分解了APP大小。

因为插件化技术需要一定的技术保障和服务端系统支持，有一定的风险，如无必要（比如一些小型项目，也没什么扩展业务）就不需要了，建议酌情选择。

## 第20条：精简功能业务

这条完全取决于业务需求。

从统计数据分析砍掉一些没用的功能是完全有可能的，甚至干脆去掉一些花哨的功能出个轻聊版、极速版也不是不可以的。

## 第21条：重复执行第1到20条

多次执行上述步骤，你总能发现一些蛛丝马迹，这是一个好消息，不是吗？

## 在线评估

针对很多朋友的反馈，有必要对条例的适用范围、易用性和风险指数做个粗略的评估，汇总如下，方便大家执行。

| 指南条例              | 适用范围          | 易用性 | 风险指数 | 备注       |
|-------------------|---------------|-----|------|----------|
| 使用一套资源            | 非极高UI要求的APP   | 易   | 无    |          |
| 开启minifyEnabled   | 全部            | 易   | 无    |          |
| 开启shrinkResources | 全部            | 易   | 无    |          |
| 删除无用的语言资源         | 非全球国际化应用      | 易   | 无    |          |
| 使用tinypng有损压缩     | 非极高UI要求的APP   | 易   | 低    |          |
| 使用jpg格式           | 仅限非透明大图       | 易   | 中    |          |
| 使用webp格式          | 仅限4.0+,4.2+设备 | 中   | 中    |          |
| 缩小大图 限允许缩小的大图     | 易             | 中   |      |          |
| 覆盖第三库里的无用大图       | 全部            | 中   | 高    |          |
| 删除armable-v7包下的so | 限允许对极少数设备不兼容  | 易   | 中    |          |
| 删除x86包下的so        | 限允许对x86设备不兼容  | 易   | 高    |          |
| 使用微信资源压缩打包工具      | 全部            | 中   | 中    | 切记要配置白名单 |
| 使使用provided编译     | 全部            | 易   | 低    | 容错处理     |
| 使用shape背景         | 全部            | 易   | 无    |          |
| 使用着色方案            | 全部            | 易   | 低    |          |
| 表情在线化             | 限含表情包的APP     | 中   | 高    |          |
| 避免重复库             | 全部            | 中   | 中    |          |
| 使用更小的库            | 全部            | 中   | 高    |          |
| 支持插件化             | 限扩展性要求高的APP   | 难   | 高    |          |
| 精简功能业务            | 限允许精简的APP     | 难   | 高    |          |

## 小结

相信经过上述步骤，一定可以把你的Android APP极大的瘦身下去。

考虑到一定的风险性，建议挑选适合自己的方法就行；同时，我也会跟踪最新的瘦身技巧，及时补充更新。

# APK瘦身实践

来源:[杰风居](#)

因为推广的需要，公司需要把APK的大小再“减小”一下，4M以内！当达到4M以内之后，公司建议说，能否再压压？2M如何？

## 瘦身前

因为平时就考虑到大小的限制，所以很多工作已经做过了，如下列举现在的状态：

- 7.3M (Debug版本) 和6.5M (Release版本)
- 开启minifyEnabled
- 开启shrinkResources
- 已经去除不相关的大型库
- 图片和代码已经经历过粗略的一轮清理

## 开始魔鬼瘦身

### 1. tinypng有损压缩

android打包本身会对png进行无损压缩，不信大家可以看看apk中的图片的大小实际上比你代码工程里的图片要小（针对没进行过无损压缩的那些png图）。

所以，纯粹的进行无损压缩并不会对apk的减小有任何效果，这是我特别想在这里强调的一个经验。现在大家主流的比较喜欢用的tinypng其实是有损压缩：

<https://tinypng.com/>

[原文] TinyPNG uses smart lossy compression techniques to reduce the file size of your PNG files...

[翻译] TinyPNG使用智能有损压缩技术，来减少PNG文件的大小...

通过tinypng确实能在尽量少的损失下再减小apk，如果图片资源多或者大的话，效果还是很明显的。

具体减少多少，因为这个处理过程我们是间隔做的，无法准确给出结果，就按200k~500k算吧。

## 2. png换成jpg

经验发现，一些背景，启动页，宣传页的PNG图片比较大，这些图片图形比较复杂，如果转用有损JPG可能只有不到一半（当然是有损，不过通过设置压缩参数可以这种损失比较小到忽略）。

因为都是大图，所以这种方式能有效减小apk的大小。

这种情况下的apk的减小是不可估量的。

## 3. jpg换成webp

如果png大图转成jpg还是很大，或者想压的更小，而尽量不降低画质，那么可以考虑一下webp。

- android 4.0+才原生支持webp，但是我们的app是兼容2.3+，所以4.0以下的设备将无法看到图片。
- 考虑到我们4.0以下的所有设备比例之和大约在0.44%，非常少
- 4.0以下的设备不会崩溃

我们选择不对4.0以下做webp兼容处理，不显示就不显示。否则，要引入webp相关so文件增大apk大小。

通过把下面四张大图换成webp， webp的quality参数按50配置(据说官方评测75是最佳值)，清晰度勉强可以接受，这个值大家具体按产品要求来定。

|                     |          |    |                         |
|---------------------|----------|----|-------------------------|
| lt_start_bg.png     | 34.5 kB  | 图像 | 2016年01月18日 星期一 14时0分0秒 |
| lt_start_bg.webp    | 6.2 kB   | 音频 | 2016年01月18日 星期一 14时2分0秒 |
| 1_avatarcheck.png   | 62.5 kB  | 图像 | 2016年01月18日 星期一 13时0分0秒 |
| 1_avatarcheck.webp  | 6.4 kB   | 音频 | 2016年01月18日 星期一 13时2分0秒 |
| n_im_nophotobg.png  | 75.2 kB  | 图像 | 2016年01月14日 星期四 17时0分0秒 |
| n_im_nophotobg.webp | 7.2 kB   | 音频 | 2016年01月18日 星期一 13时2分0秒 |
| ME.txt              | 91 bytes | 文本 | 2016年01月18日 星期一 10时5分0秒 |
| _bg.png             | 88.3 kB  | 图像 | 2016年01月14日 星期四 17时2分0秒 |
| _bg.webp            | 7.6 kB   | 音频 | 2016年01月18日 星期一 12时5分0秒 |

一共webp化了这四张大图，  
文件本身减少227k

其中安装jpg转webp工具：

```
brew install webp
```

转换命令如下

```
cwebp -q <quality> input.jpg output.webp
// Example:
cwebp -q 50 a.jpg a.webp
```

更多下载：<https://developers.google.com/speed/webp/docs/precompiled>

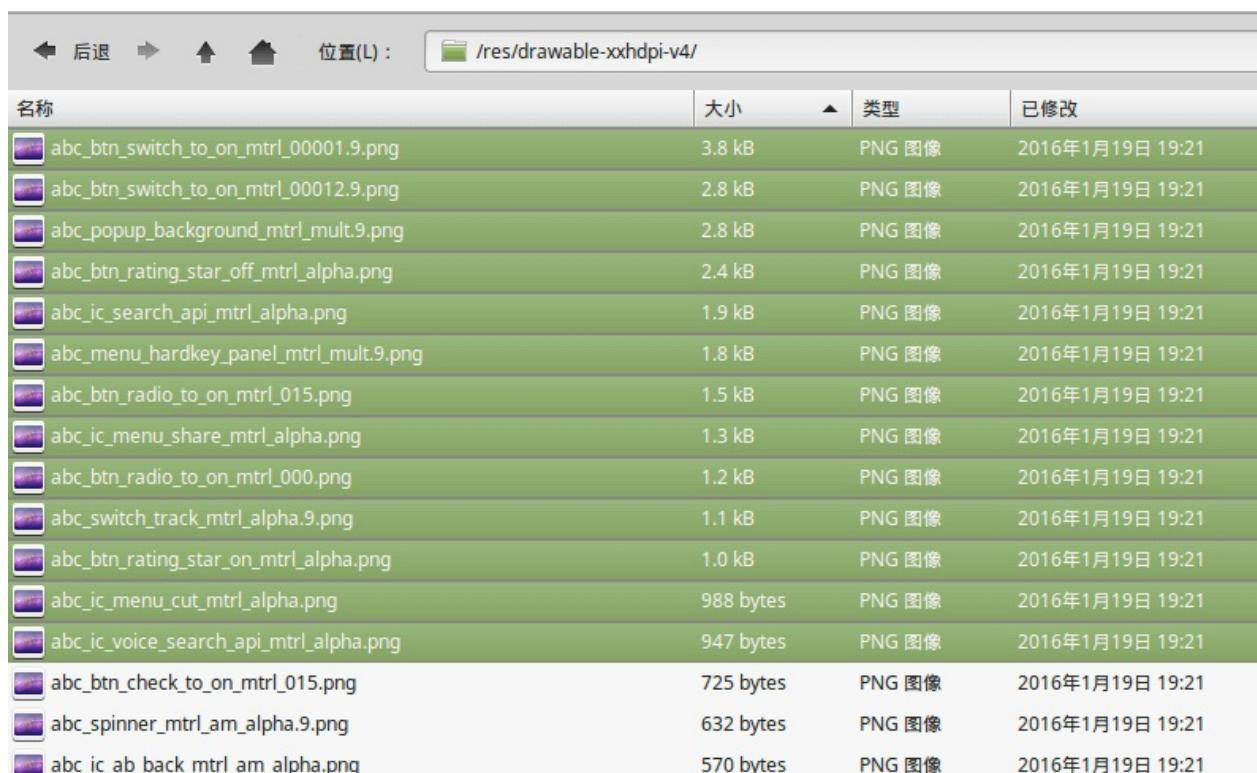
最终，apk减小了188k。

## 4. 大图缩小

如果经过上面的步骤，依然存在大图的话，说明确实图有点大了，可能真的有点大了！所以，要考虑的问题是，是否有必要保证如此的大小？能否缩小？如果这方面能减小的话，apk瘦身的效果必然又会上一个档次。这种情况下的apk的减小是不可估量的。

## 5. 覆盖aar里的一些默认的大图

一些aar库里面包含根本就没有用的图。最典型的是support-v4兼容库中包含一些“可能”用到的图片，实际上在你的app中不会用到。



| 名称                                     | 大小        | 类型     | 已修改              |
|----------------------------------------|-----------|--------|------------------|
| abc_btn_switch_to_on_mtrl_00001.9.png  | 3.8 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_btn_switch_to_on_mtrl_00012.9.png  | 2.8 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_popup_background_mtrl_mult.9.png   | 2.8 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_btn_rating_star_off_mtrl_alpha.png | 2.4 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_ic_search_api_mtrl_alpha.png       | 1.9 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_menu_hardkey_panel_mtrl_mult.9.png | 1.8 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_btn_radio_to_on_mtrl_015.png       | 1.5 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_ic_menu_share_mtrl_alpha.png       | 1.3 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_btn_radio_to_on_mtrl_000.png       | 1.2 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_switch_track_mtrl_alpha.9.png      | 1.1 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_btn_rating_star_on_mtrl_alpha.png  | 1.0 kB    | PNG 图像 | 2016年1月19日 19:21 |
| abc_ic_menu_cut_mtrl_alpha.png         | 988 bytes | PNG 图像 | 2016年1月19日 19:21 |
| abc_ic_voice_search_api_mtrl_alpha.png | 947 bytes | PNG 图像 | 2016年1月19日 19:21 |
| abc_btn_check_to_on_mtrl_015.png       | 725 bytes | PNG 图像 | 2016年1月19日 19:21 |
| abc_spinner_mtrl_am_alpha.9.png        | 632 bytes | PNG 图像 | 2016年1月19日 19:21 |
| abc_ic_ab_back_mtrl_am_alpha.png       | 570 bytes | PNG 图像 | 2016年1月19日 19:21 |

我没有把所有图都替换掉，只是把几张大一点点的图（选中的那些图）用1x1的图片替换，如果9patch图的话，要做成3x3的9patch图替换。

support库可能还算好的，就怕有些库引用了一些大图而不自知，可以在/build/intermediates/exploded-aar/下的各个aar库的res目录查找检验。

apk减小了18k。

## 6. 删除armable-v7包的so

感谢@杨辉\_\_ ,@kymjs张涛的提醒， armable-v7和armable文件夹可以只保留armable。

当然， armable-v7a的库会对图形渲染方面有很大的改进，因为我们主要是一些业务上动态库， 所以删掉无大碍。



apk减小了191k。

## 7. 微信资源压缩打包

这个方案网上一直在说，之前一直没有需求或者动力实践，在这里感谢一下@裸奔的凯子哥的推荐和交流，他那边的apk可以压小1M，效果还是比较惊人的。

这个步骤我是在后面很多步压缩之后测试的，每个阶段的压缩结果都会有些许出入，所以数据仅供参考。



微信压缩方案结果比较

- 通过正常压缩， apk包减小了464k。
- 如果开启7zip， apk包减小了594k。
- apk减小了594k。

PS: 关于这个压缩，我集成到了gradle脚本中了，新建了一个Task，大概代码如下：

```
task compressReleaseApp {
 // 在现有release的版本上生成到compressed目录下
 def appid = "appid"
 def channel = "abcdefghijkl"
 def guardJarFile = file('../AndResGuard/andresguard-1.1.jar')
 def guardConfigFile = file('../AndResGuard/config.xml')
 def originApkFile = file("../app.${appid}/build/outputs/apk/release/${appid}-rele
 def outputDir = file("../app.${appid}/build/outputs/apk/compressed/")
 def keystoreFile = file(RELEASE_STORE_FILE)
 // 开始执行压缩命令
 def proc = "java -jar ${guardJarFile} ${originApkFile} -config ${guardConfigFile}
proc.waitFor();
println "return code: ${ proc.exitValue()}" +
 ", stderr: ${proc.err.text}" +
 " stdout: ${proc.in.text}"
}
```

config开启了7zip, 部分配置如下:

```

<?xml version="1.0" encoding="UTF-8"?>

<resproguard>

 <!--default property to set -->

 <issue id="property" >

 <seventzip value= "true" />

 <!-- ... -->

 </issue>

 <issue id="whitelist" isactive="true">

 <path value ="com.xxx.yyy.R.drawable.emoji_*" />

 <path value ="com.xxx.yyy.... />
 </issue>

 <issue id ="compress" isactive="true">

 <!-- ... -->

 </issue>

</resproguard>

```

- 详情参考：<https://github.com/shwenzhang/AndResGuard>
- 原理介绍：[http://mp.weixin.qq.com/s?\\_\\_biz=MzAwNDY1ODY2OQ==∣=208135658&idx=1&sn=ac9bd6b4927e9e82f9fa14e396183a8f#rd](http://mp.weixin.qq.com/s?__biz=MzAwNDY1ODY2OQ==∣=208135658&idx=1&sn=ac9bd6b4927e9e82f9fa14e396183a8f#rd)

## 8. proguard深度混淆代码

之前为了简单起见，很多包都直接忽略了，现在启动严格模式，把能混淆的都混淆了：  
release版本混淆效果

| 非最终压缩版本比较  |   |     |     |       |                       |
|------------|---|-----|-----|-------|-----------------------|
| -rw-r--r-- | 1 | jay | jay | 4232K | 1月 18 13:36           |
| -rw-r--r-- | 1 | jay | jay | 4224K | 1月 18 13:42           |
| -rw-r--r-- | 1 | jay | jay | 4196K | 1月 18 13:50           |
| -rw-r--r-- | 1 | jay | jay | 4192K | 1月 18 13:57           |
| -rw-r--r-- | 1 | jay | jay | 4030K | 1月 18 14:03           |
| -rw-r--r-- | 1 | jay | jay | 4008K | 1月 18 14:10           |
| -rw-r--r-- | 1 | jay | jay | 4006K | 1月 18 14:14           |
|            |   |     |     |       | dev-0-origin.apk      |
|            |   |     |     |       | dev-1-picasso.apk     |
|            |   |     |     |       | dev-2-fasterxml.apk   |
|            |   |     |     |       | dev-3-lianlianpay.apk |
|            |   |     |     |       | dev-4-org.apk         |
|            |   |     |     |       | dev-5-octo.apk        |
|            |   |     |     |       | dev-6-uk.apk          |

采用微信压缩方案最终效果比较：

|                   |                                           |                   |                                          |        |
|-------------------|-------------------------------------------|-------------------|------------------------------------------|--------|
| 3604K 1月 18 13:37 | 2101-dev_signed_7zip_aligned-0-origin.apk | 3389K 1月 18 14:18 | 2101-dev_signed_7zip_aligned-1-after.apk | 最终压缩版本 |
|-------------------|-------------------------------------------|-------------------|------------------------------------------|--------|

最终压缩release版本混淆效果 apk减小了215k。

PS:混淆后，一定要经过严格测试，有时候甚至很难发现错误，比如我开启严格混淆，用了一段时间之后慢慢发现了两个bug，排除了两个包程序才正常。

## 9. 深度清理代码和资源

有意思的是，无论何时何地去清理代码和资源，总能有新的发现：

- 新发现或者新引入的无用图片
- 这几张图怎么一样
- 这个类好像没有用
- 没用的类相关的图片也没用
- 有些图片可以用着色方案替换
- 有些图片可以用shape来代替
- hdpi里的ic\_launcher.png好像也可以删掉
- ...

apk减小了66k。

## 10. proguard去符号表

之前为了保留调试信息，我们是在Proguard保留了符号表的：

```
-keepattributes SourceFile,LineNumberTable
```

官方渠道我觉得还是尽量保留这个，现在针对推广渠道，只能采用特殊手段，注释这一行。

apk减小了230k。

ps:以后友盟上看推广渠道的bug要辛苦一点，手动上传mapping.txt了。

## 11. provided关键字

可以对仅在运行时需要的库设置provided关键字，实际并不被打包：

```
provided 'com.android.support:support-annotations:22.0.0'
```

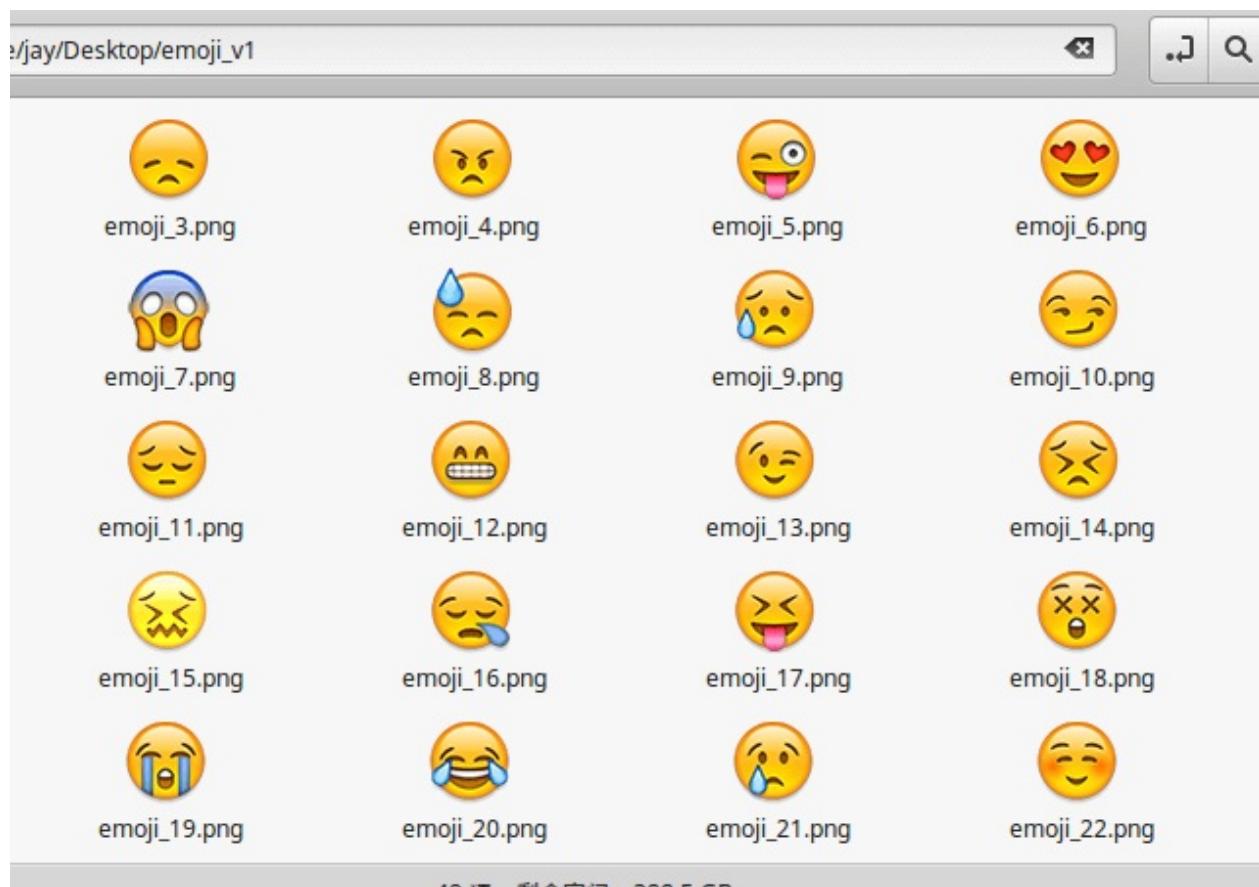
我没有发现这样的场景，如果说有的话，就是support-annotations，但是经过后来的测试验证，support-annotations本来就会在release版本中被minifyEnabled掉，所以对support-annotations设置provided是没有意义的。

如果有实际场景，欢迎留言说明，不甚感激。

apk没有减小。

## 12. 表情包在线化

虽然应用的表情不多，只有50来个，但是如果能把这部分表情放到网上，不仅能有效减小apk大小，还可以方便后期扩展支持：



打包成emoji\_v1.zip, 大小是202k。

现在把emoji\_v1.zip放到网上，按需下载后使用，最终对比结果如下：表情包在线化成效

|                     |                                         |
|---------------------|-----------------------------------------|
| y 3797K 1月 19 17:57 | 01-dev-0-origin.apk                     |
| y 3599K 1月 19 17:28 | 01-dev-1-after.apk                      |
| y 3185K 1月 19 17:58 | 01-dev_signed_7zip_aligned-0-origin.apk |
| y 2992K 1月 19 17:29 | 01-dev_signed_7zip_aligned-1-after.apk  |

apk减小了193k。

## 13. 全版本兼容的着色方案

考虑着色方案主要目的是更方便支持多主题，减轻UI工作量，减少工程里一大堆selector文件等，然后才是，顺便的减小一下apk大小。

通过着色方案，我们去除了10多张纯色的按下状态图片和对应的xml等等。

apk减小了15k。

PS: 具体实现可以参考 <http://www.race604.com/tint-drawable/>，而我也把它集成到了我的LessCode库中了：[DrawableLess.java](#)

## 14. 去除重复库

发现两个地方：

- 现在发现七牛的SDK引用了android-async-http-1.4.6.jar，虽然不大，只有95.4k，但是感觉完全可以写一个轻量级的jar，控制在10~20k就足够了，具体可以在现有的网络库上实现。
- 自己工程使用的是UIL，但是引入的第三方库引用了picasso，两个重复的图片下载库也是完全没用必要的。现在还没有处理这块，新任务介入，延期优化，敬请期待。

## 15. 去除无用库

这是一个很基本的点，但是很容易被人忽视，当你仔细回顾的时候，有一些鸡肋的功能或者库，是几无用处的。不如干脆去掉。

比如，在很早的时候，我就把我们app里的sharesdk删除了，因为对于我们的产品定位和推广来看，这毫无意义。这种情况下的apk的减小是不可估量的。

## 16. 去除百度统计

这个视具体情况决定。

因为我们的APP里面包含友盟和百度两套统计系统，早期老板要求，事实上后面已经很少看这方面的数据，百度统计的数据几乎没人去看，可以暂时先去除。

原本的百度统计的jar有130多k，去除之后的apk的减小会远远没有这么多。

apk减小了20k。

## 17. 使用更小的库

使用更小的库不应该成为你选择方案的决定性因素，但是可以作为参考因素（fresco确实太大了，这个大小也可以成为决定性因素）。

图片下载，网络请求，json解析等等的库和它的竞品都有多大，你心里有数吗？

以工具库为例，网上有很多工具库，但是往往它们的大小很难控制。

- xutils-3.2.6.aar – 843.8k
- lite-common-1.1.3.jar – 148.1k
- lesscode-core-0.8.2.aar – 64k
- ...

上面最后一个库LessCode是我自己收集的工具类集合，非常小：LessCode，混淆后只有不到50k大小。

不仅提高了开发效率，减少了冗余代码，而且能避免引用一些其他大型的库，有效避免包的增大。

比如，我们碰到过这样的一个bug，快速点击按钮多次触发跳转，现在RxJava结合RxBind有这样的一个场景解决方案，如果引入这些库的话必然会增大apk大小，实际上就几行代码，我把这样的解决方案集成到了LessCode，下次别的项目碰到这样的问题不用再犹豫是否要引入一个这么大的库了。

这些小的工具库，建议根据自己的经验人手总结一个，不求全，但求精！

这种情况下的apk的减小是不可估量的。

## 18. 插件化

尴尬的是，我们所呈现的功能大部分都是重要的不可分割的功能，很难从业务上分离出来。

今年预计要实践一个轻量级的插件化方案，用别人的也好，自己写也好，希望能解决或者优化一些安装包加载多模块，或者主题

切换，或者热修复的问题。

这里作为候选方案备用。

这种情况下的apk的减小是不可估量的。

## 19. 功能业务取舍

一开始考虑瘦身，领导是允许适当的砍掉一些功能，因为4M的目标我们已经实现了，所  
以现在还没有到砍功能的地步。这里作为候选方案备用。

这种情况下的apk的减小是不可估量的。

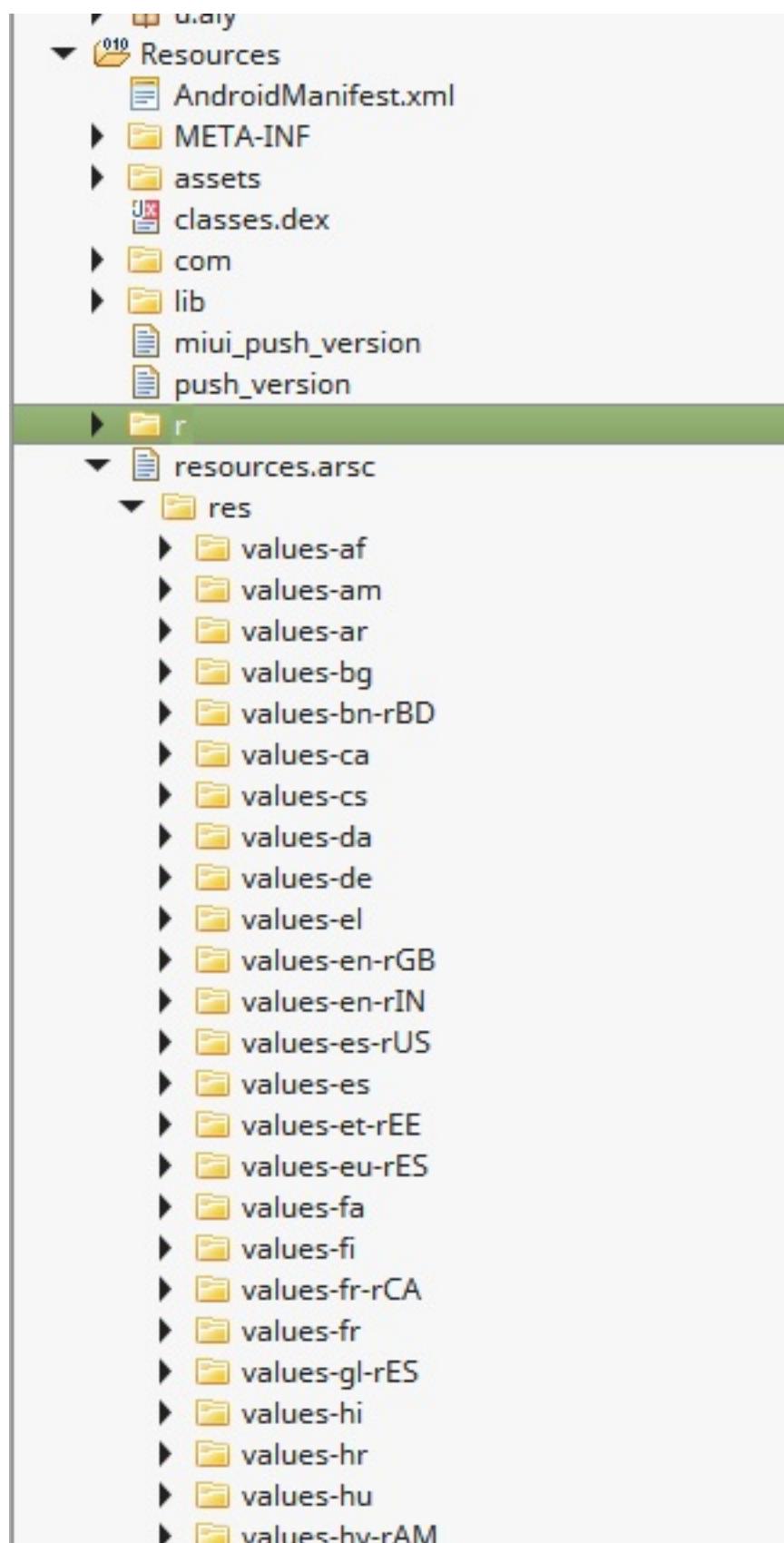
## 补充

文章发出后，收到了一些朋友的建议，补充几点。

- 

1. 去除无用的语言资源

感谢@牧志轩的建议，通过配置resConfigs可以选择只打包哪几种语言，进而去掉各种aar  
包中全世界的语言，尤其是support包中的。



选择保留什么语言要根据产品的用户和市场来定，如果只选择默认英语和中文语言，配置如下

```
 android {
 defaultConfig {
 resConfigs "zh"
 }
 }
```

看看效果：

```
→ vs_res ls -l --block-size=k
total 12964K
-rw-r--r-- 1 jay jay 3610K 1月 22 12:03 gouda-release-v3.2.1.2-68-0101-dev-0-origin.apk
-rw-r--r-- 1 jay jay 3413K 1月 22 12:06 gouda-release-v3.2.1.2-68-0101-dev-1-resConfig.apk
-rw-r--r-- 1 jay jay 2973K 1月 22 12:04 gouda-release-v3.2.1.2-68-0101-dev_signed_7zip_aligned-0-origin.apk
-rw-r--r-- 1 jay jay 2957K 1月 22 12:07 gouda-release-v3.2.1.2-68-0101-dev_signed_7zip_aligned-1-resConfig.apk
```

只选择默认英文和中文的语言包效果

如果不采用微信压缩方案结果对比，apk减小了197k。

如果采用微信压缩（开启7zip）对比结果，apk只减小了16k，因为微信对resources.arsc进行了强力压缩，厉害！apk减小了16k。



### 1. 删除x86包的so

再次感谢@杨辉\_\_的建议，x86的包删除了之后，测试反应好像有些机器容易崩溃，未能经过严格测试，所以主版本又复原了，只在个别渠道执行这条措施。

一般情况下不会有大问题，测试了一下效果，apk减小了78k。

这里作为候选方案备用。

## 小结

最终，我们成功的把apk压到了2.9M，如果把上面遗漏的步骤继续再做，应该还能再减小一点。客户反应压的好小，领导简直不敢相信～瘦身不难，难的是魔鬼瘦身！



# 写出高质量代码的10个Tips

来源:[gold.xitu.io](http://gold.xitu.io)

很长一段时间以来，我都在关注如何提高代码质量，也为此做过一些尝试，我想这个话题可能大家会比较感兴趣，在这里分享一下我关于如何提高代码质量的一些体会。

## 1. 打好基础

写出高质量代码，并不是搭建空中楼阁，需要有一定的基础，这里我重点强调与代码质量密切相关的几点：

- **掌握好开发语言**，比如做Android就必须对Java足够熟悉，《Effective Java》一书就是教授大家如何更好得掌握Java, 写出高质量Java代码。
- **熟悉开发平台**, 不同的开发平台, 有不同的API, 有不同的工作原理, 同样是Java代码, 在PC上写与Android上写很多地方不一样, 要去熟悉Android编程的一些特性, iOS编程的一些特性, 了解清楚这些, 才能写出更加地道的代码, 充分发挥各自平台的优势。
- **基础的数据结构与算法**, 掌握好这些在解决一些特定问题时, 可以以更加优雅有效的方式处理。
- **基础的设计原则**, 无需完全掌握23种经典设计模式, 只需要了解一些常用的设计原则即可, 甚至你也可以只了解什么是低耦合, 并在你的代码中坚持实践, 也能写出很不错的代码。

## 2. 代码标准

代码标准在团队合作中尤为重要，谁也不希望一个项目中代码风格各异，看得让人糟心，即便是个人开发者，现在也需要跟各种开源项目打交道。标准怎么定是一个老生常谈的话题，我个人职业生涯中经历过很多次的代码标准讨论会议，C++, C#, Java等等，大家有时会坚持自己的习惯不肯退让。可现如今时代不一样了，Google等大厂已经为我们制定好了各种标准，不用争了，就用这些业界标准吧。

## 3. 想好再写

除非你很清楚你要怎么做，否则我不建议边做边想。你真的搞清楚你要解决的问题是什么了吗？你的方案是否能有效？有没有更优雅简单的方案？准备怎么设计它，在必要的情况下，需要有设计文档，复杂一些的设计需要有同行评审，写代码其实是很简单的事情，前提是你要先想清楚。

## 4. 代码重构

重构对于代码质量的重要性不言而喻，反正我是很难一次把代码写得让自己满意、无可挑剔，《重构》这本书作为业内经典也理应人人必读，也有其他类似的教授重构技巧的书，有些也非常不错，遗憾的是我发现很多工作多年的学生甚至都没有了解过重构的概念。

## 5. 技术债务

知乎上最近有个热门问题《为什么有些大公司技术弱爆了？》，其实里面提到的很多归根结底都是技术债务问题，这在一些大公司尤为常见。技术债务话题太大，但就代码质量而言，我只想提一下不要因为这些债是前人留下的你就不去管，现实是没有多少机会让你从一个清爽清新的项目开始做起，你不得不去面对这些，你也没法完全不跟这些所谓的烂代码打交道。

因此我建议各位：当你负责一个小模块时，除了把它做好之外，也要顺便将与之纠缠在一起的技术债务还掉，因为这些债务最终将是整个团队来共同承担，任何一个人都别想独善其身，如果你还对高质量代码有追求的话。

作为团队的技术负责人，也要顶住压力，鼓励大家勇于做出尝试，引导大家不断改进代码质量，不要总是畏手畏脚，停滞不前，真要背锅也得上，要有担当。

## 6. 代码审查

我曾经听过一些较高级别的技术分享，竟然还不时听到一些呼吁大家要做代码审查的主题，我以为在这个级别的技术会议上，不应再讨论代码审查有什么好，为什么要做代码审查之类的问题。同时我接触过相当多所谓国内一线互联网公司，竟有许多是不做代码审查的，这一度让我颇为意外。

这里也不想多谈如何做好代码审查，只是就代码质量这点，不客气地说：没有过代码审查经历的同学，往往很难写出高质量的代码，尤其是在各种追求速度的糙快猛创业公司。

## 7. 静态检查

很多代码上的问题，都可以通过一些工具来找到，某些场景下，它比人要靠谱得多，至少不会出现某些细节上的遗漏，同时也能有效帮助大家减少代码审查的工作量。

Android开发中有Lint, Find bugs, PMD等优秀静态检查工具可用，通过改进这些工具找出的问题，就能对语法的细节，规范，编程的技巧有更多直观了解。

建议最好与持续集成(CI)，代码审查环境配套使用，每次提交的代码都能自动验证是否通过了工具的代码检查，通过才允许提交。

## 8. 单元测试

Android单元测试，一直备受争议，主要还是原生的测试框架不够方便，每跑一次用例需要在模拟器或者真机上运行，效率太低，也不方便在CI环境下自动构建单元测试，好在有Robolectric，能帮我们解决部分问题。

单元测试的一个非常显著的优点是，当你需要修改大量代码时，尽管放心修改，只需要保证单元测试用例通过即可，无需瞻前顾后。

## 9. 充分自测

有一种说法：程序员最害怕的是他自己写的代码，尤其是准备在众人面前show自己的工作成果时，因此在写完代码后，需要至少跑一遍基本的场景，一些简单的异常流。在把你工作成果提交给测试或用户前，充分自测是基本的职业素养，不要总想着让测试帮你找问题，随便用几下就Crash的东西，你好意思拿给别人吗？

## 10. 善用开源

并非开源的东西，质量就高，但至少关注度较高，使用人数较多，口碑较好的开源项目，质量是有一定保证的，这其中的道理很简单。即便存在一些问题，也可以通过提交反馈，不断改进。最重要的是，你自己花时间造的轮子，需要很多精力维护，而充分利用开源项目，能帮助你节省很多时间，把精力专注在最需要你关心的问题上。

从另一个方面来说，开源项目中的一些知名项目，往往是领域内的翘楚所写，学习这些高手的代码，能让你了解到好的代码应该是怎样的，培养出更灵敏的嗅觉，识别代码中的各种味道。

我最近运营了一个Android开发相关的微信公众号：Android程序员(AndroidTrending)，主要关注Android最佳实践，开发经验分享，原创内容一般在那里首发，欢迎感兴趣的朋友扫码关注。

---

很长一段时间以来，我都在关注如何提高代码质量，也为这做过一些尝试，我想这个话题可能大家会比较感兴趣，在这里分享一下我关于如何提高代码质量的一些体会。

## 作者

# 资深谷歌安卓工程师对安卓应用开发的建议

来源:[Realm](#)

擅长Java语言的资深开发者们，多年以来多是工作在网页，服务器，和桌面系统等开发领域。这些领域的经验帮助他们建立起来了自己使用Java语言的模式和自己的Java库的生态系统。但是移动应用的开发却和这些领域的java开发有着天壤之别。优秀的安卓应用开发者需要考虑到移动设备的限制，重新学习怎么样去使用java语言，怎么样去有效地使用实时环境和安卓平台，然后写出更好的安卓应用程序。

---

## About the Speaker: Romain Guy

Romain是谷歌的安卓工程师。在加入Robotics之前，他在安卓Framework组参与了安卓1.0到5.0的开发工作。他现在又重新加入了安卓的新UI和图形图像相关的项目。

[@romainguy](#)

## About the Speaker: Chet Haase

Chet也是谷歌的工程师。他现在是安卓UI Toolkit组的组长，他擅长于动画，图像，UI控件和其他能带来安卓更好的用户体验的UI组件的开发。他还擅长撰写和表演喜剧。

[@chethaase](#)

## 介绍 (0:00)

本次演讲是以安卓平台组写的近10篇文章为基础的。所有的文章都能够在Medium网站上看到，文章的第一部分请看[这里](#). 今天我们会讲到这些文章里面的一些东西，如果你对特定的话题感兴趣或者想深入了解它们，请去阅读原文。

## 为什么移动开发如此艰难？ (1:47)

### 有限的内存 (1:47)

我们发现谷歌公司里面的应用开发者有一个大问题，他们对口袋里每天都携带着的安卓手机的本质都有些误解。这些设备内存，CPU的处理能力和电池的待机能力都非常有限。开发者们必须理解你们的应用不是在设备上唯一运行的应用。内存是非常有限的资源，并且

被整个系统共享着。我所在的Android平台组非常小心地对待这个问题，这也是为什么有的时候我们建议的一些规则看起来会有一点极端的原因。我们需要有全局的眼光来看待这个问题，因为系统会同时运行20、30、40个的进程。当你只为单一应用开发的时候，牢记这些限制是比较困难的。

今天我们手上的设备往往会有1-3GB的内存，但不是所有的安卓设备都有这么多的内存的。安卓阵营中有14亿的设备，其中许多是两三年前的产品。事实上，他们才是着14亿设备的主力军。大部分的设备不是在美国和西欧，而是在中国和印度这样的国家。在这些国家里，安卓设备必须便宜才能吸引更多的消费者。

安卓在从4.3到4.4的演进过程中，我们使出浑身解数才使安卓系统能够成功地在内存受限的系统上运行起来。很长时间以来，因为人们想在便宜的500MB内存的设备上使用安卓系统，姜饼系统成了他们唯一的选择。当然，现在的情况已经大不相同，姜饼设备已经差不多消失殆尽了。但是，因为现在的安卓版本有着庞大的大内存消耗的应用群，这就需要更强大的框架和平台。

这是一件你必须持续思考的事情，但是这很难，因为它不总在你优先考虑的事情的范畴内。你可能知道Knuth的名言“过早的优化是一些罪恶的根源”。虽然我同意他的观点，可是当你写完你的应用，然后打算从应用中再减少50MB的内存的使用，也是一件非常困难的事情。我们不是说你需要毁掉你的应用系统架构或者牺牲你的测试，但是每一次你能做些改进来改善你的内存使用情况的时候，请马上动手。

为了整个设备有更好的整体用户体验而开发。当你的应用很大的时候，你会导致系统杀掉别的应用，这样你就影响到了别人应用的用户体验。当别人的应用不注意内存使用情况的时候，你的应用体验也不会好。你们需要互相交流然后找到和谐共存的方法，请牢记这条建议。

如果系统需要杀掉除了你的应用之外的所有应用，那么当你的应用退出以后，其他应用需要重新建立他们可运行的环境。这样你的应用会看起来像一个恶意的应用。如果你想要让用户重入你的应用时有一个良好的体验的话，请尽你所能将你的应用保持在后台运行并且消耗最小的内存。

在有些应用当中，实现内存回调接口是非常管用的。你能在API文档中非常容易的找到onTrim的内存回调函数，并且能实现多个级别的Trim操作。基于你的Trim操作的级别，你可以释放一些缓存和bitmaps，也可以退出一些Activity，或者其他任何能帮助你留在后台运行的措施。

## CPU 处理器 (8:07)

移动处理器显然比桌面和服务器的处理器能力要慢很多。虽然从外界不容易察觉，但是移动处理器大部分时候都是处在过热降频保护的状态。这意味着尽管CPU的频率很高，但是仍然不能像桌面和服务器的处理器一样快。当诸如用户在屏幕上拖拽的时候，CPU可以达到它的最快处理速度。可是在其他的情况下，CPU处理器是跑在待机频率上的，不然的话，电池不能保证用户的基本待机时间。

如果你买的是2GHz的台式机或者笔记本，CPU大部分时候是能跑在那个参数指标上的。可是你如果买的是一个2GHz的手机，CPU只能有时候跑在2GHz上。如果你买的是八核的手机，硬件上你是有八个处理核，但是他们大部分时候不会一起工作。基本上，手机盒子上的参数表和你实际上能达到的参数是有不一致的地方的。这主要是为了降频保护CPU，提高电池待机时间和减少发热量。

## GPU 图像处理器 (10:10)

和CPU一样，GPU也有保护降频的功能。纹理加载（Texture uploads）的开销特别大。所有bitmap的操作或者结果是bitmap的操作都会被加载到GPU。举个例子说，当你在绘制路径时，这些东西会转成bitmap并且作为纹理加载到GPU，这时系统的开销就会特别大。你的性能瓶颈就可能在这。

填充率和分辨率之间也是相关联的。Nexus 6P的分辨率非常高，换句话说，屏幕上有许多的像素点要填充。但是GPU的性能带宽却跟不上。系统为了填充所有的像素点而超负荷的工作。你的要求越高，系统需要的时间就越长。

一个提升UI性能的技巧就是避免过度刷屏。你需要系统填充屏幕上每次像素的次数越多，当屏幕越来越大和分辨率越来越高的时候，情况就会越来越糟。如果你有一个Playstation 4或者Xbox One的话，他们在运行那些1080p的游戏（每秒30帧）时十分卡顿。在我们的手机上，我们有更高的分辨率，而且我们以每秒60帧的速度完成所有的事情；我们尽可能的完成更多的事情并且消耗更少的电量。这就是为什么我们对图片的处理性能有着诸多的要求以及对应用优化的建议，因为我们没有你想象的那么强的处理能力和电池电量。

## 内存 = 性能 (12:29)

如果你有大的应用，那么就会发生更多的内存页的置换，更慢的内存分配。实际运行中，系统需要遍历来决定新的对象可以放到内存何处，这需要花费更多的时间。回收也会需要花费更多的时间；每次需要内存的时候都要遍历更多的东西。

整体上内存回收机制也被更多地触发。在我们的演讲中，我们详细阐述了内存回收是如何工作的，而且讨论了ART和Dalvik的改进。详情请看 [上面的视频 at 13:06;](#))

## 低端设备 (19:40)

你口袋里面的设备保证比你用户的设备要快很多。你认为的那些老旧设备并没有消失。A 它们依旧在用户的口袋中，而且用户打算尽可能长地使用它们。B 那些老旧的设备虽然慢但是便宜。这意味着它们能吸引哪些不能够购买最快设备的人们。

## 流畅的帧频率 (21:01)

找到一个流畅的帧频率是十分困难的。你只有16毫秒的时间去完成所有的事情。这些事情包括触屏事件的处理，计算，布局，绘制帧，然后交换缓存。16毫秒意味着每秒刷新60帧，这是我们在安卓系统上要求的，因为我们是V-Sync。我们不希望屏幕花屏。花屏在有些游戏中你能看到，因为它们在一秒内同时有两个buffer。那个看起来太可怕了，我们不想让你这么做。这就意味着如果你仅仅多花了一点时间，哪怕17毫秒，我们就会跳过一帧。然后跳过一次V-sync，这样系统就不是60fps了，而是30fps。我们叫这种现象为Jank。

系统在60fps和30fps中来回跳跃是件非常糟糕的事情。这会让你的用户觉得你的应用非常janky和不一致。也有可能你的应用会一直表现糟糕，一直都是30fps。这就是许多游戏当它们认为做不到60fps的时候，它们就一直都是30fps。

## 实时运行：语言 ≠ 平台 (22:48)

几周前，有人问我“当有新版本的JDK发布的时候，你一般做什么？”。然后我意识到他们并不理解语言和实时运行环境是不一样的。当有新的JDK发布的时候，我基本不关心。它和安卓运行时一点关系都没有。我们使用同一种语言，但那不意味着我们在运行的时候是一样的。

当人们使用Java语言的时候，有三个方面的因素构成了整个java体验。Java编程语言本身，实时运行环境（在有些server环境中叫HotSpot）然后是硬件设备。

对于那些擅长服务器的人来说，服务器的实时运行环境提供诸如移动，压缩收集器的功能，这些都意味着临时的内存分配带来的系统开销非常小。这些事情和移动指针一样快。在服务器环境中，的确如此。然后你会理所当然的认为移动设备有一样快的处理器，一样大的内存，所以处理能力就应该和服务器一样无限制。

在安卓系统中，情况是非常不一样的，特别是你习惯于无限的系统资源和完全不同的实时运行环境的时候。我们有Dalvik和ART。我们没有压缩，这意味着当你分配一个对象的时候，这个对象就会存在堆里面。堆会碎片化，也会使得寻找空余的内存空间变得开销更大

和更困难。在ART环境中，我们有空闲时的压缩。ART从来不能在应用是前台运行的时候压缩堆空间，但是当它在后台运行的时候是可以压缩堆空间的。在进程的生存周期里面，有的时候压缩是会发生的，这会帮助系统寻找空闲内存。

压缩在本地代码上的作用更为关键。如果你的应用使用JNI，并且分配一个指针给一个java对象的时候，那个指针通常会被系统认为一直有效。在ART环境中，如果你的堆被压缩了，那指针就会变成野指针。如果你使用JNI，在开始的时候你就必须对这种情况特别小心。

## UI 线程 (26:56)

安卓是一个单线程的UI系统。技术上，你可以有多个UI线程，但这会带来很多麻烦。正如你担心的那样，所有的UI控件都是在同一个线程里。因为是一个UI线程，所以你任何阻塞UI线程的操作都会带来性能上的影响，jankiness（闪屏）和不一致性。在UI线程上分配内存就更糟了。如果你有后台运行的线程分配内存，当虚拟机VM阻塞所有的线程（包括UI线程）十几毫秒的时候，你就会跳过一帧。这样，你就只有30fps，这会非常糟糕。

## 存储 (28:53)

存储的性能没有一个定论。有时候人们把数据存在SD卡上，这就会有各种各样的性能标准。当存储设备快要满的时候，即使在同一个存储设备上也是有不同的性能体现的。但是如果你的应用是依赖每个测试设备的存储速度的话，那总归是不好的。写flash存储的时候也意味着控制器需要做很多事情，因为它需要收集信息，记录哪些空间已用和哪些空间没用。换句话说，如果你的应用在后台做大量的IO操作的话，你会把这个系统拖慢。你应该有这样的经验，当Play Stoer 在后台安装应用更新的时候，你的设备会突然间感觉慢了起来。因为它在后台做磁盘写操作，所以前台的应用读它们自己资源也会变得很慢。

存储的大小也是各式各样的，所以不要让你的应用对特定的存储大小有依赖。APK的大小也很重要。你应用中资源的大小会影响到你的启动时间。

优化你的应用资源的办法有很多。现在在老的版本的安卓上，也有了矢量图。如果可以的话，请使用安卓的SVG库。虽然对于图标来说不太合适，但的确能对你的应用起好的作用。你也可以使用WebP格式，这会比PNG省下20%-30%的存储空间。[PNG Crush](#)也是一个很好地离线工具。APT做了一部分PNG crush的工作，但是还有很多工具会做的更好，更有效。

## 网络 (28:53)

你使用的网络肯定比大多数用户的网络要快。他们可能还在用着2G网络，并且流量很贵。所以你的应用不应该依赖持续的网络连接。也许你应该让你的应用能够智能地下载它需要的内容或者你的应用不需要依赖网络下载的内容就能使用，真正的内容可以晚点再去下载。

开车去Utah的路上，你可以测试在糟糕的网络环境下，你的应用的表现如何。或者你可以用一些模拟糟糕网络的工具。所有这些应用的测试都是手工进行的。

## 每一台设备都是一个村庄 (33:31)

每一台设备都是一个村庄，这意味着每个在村庄里的人都需要共同努力来维护一个好的用户体验。你可以让这个体验特别差，也可以特别好。如果你的应用想成为好的体验的支持者，你可以试着在你的manifest里面关闭Service和broadcast receivers。在代码里面，当你的应用发现用户打算使用你的应用的时候，你才动态的开启你的services和receivers。邮件客户端就是一个好的例子，当用户安装邮件客户端的时候，所有的一切都应该处于关闭的状态。一旦用户点击了该邮件客户端并且加入了账户的时候，你才开启你的services和receivers。然后下次用户重启设备的时候，Framework就会记录下来，你的服务就真正的运行起来了。这个方法很简单，但也很重要。因为你的services虽然运行起来了，但是没有人用，你会对别人带来不必要的坏的影响。

另外一个现象叫做“公共地带的悲剧”：每个人都认为他自己的应用是最重要的。如果每个人都是这样的观点，那么每个应用都会非常大而且尽可能被激活，这样设备会承受不了。勿以恶小而为之，勿以善小而不为。

## 技巧与建议 (35:53)

### 了解你使用的语言 (35:55)

开发者们可能有了很多年Java的经验，但是却不能最好地利用语言来发挥出移动设备的最大性能。

- 不要使用Java的序列化

序列化本身是非常有用的，这不是我们现在讨论的话题。如果你用过序列化的话，你会发现不太好用，因为你需要产生UUID，而且它还有限制。序列化也会慢，因为它用到了Reflection。

- 使用安卓的数据结构

其他场合Java开发者常用到的一些集合类和方法在Android上也许就不合适了。

Autoboxing 和 Primitive Java 类型的替代品在Java领域里使用的非常广泛。集合的迭代器也是非常常用的。在Android平台上，我们特别创建了一些集合类来替代这些模式。对于Key来说，我们使用基本类型，所以如果hash表里面是一个整形作为key的时候，我们没有autoboxing。你可以以使用SparseArray类，同样的，ArrayMap和SimpleArrayMap也是HashMap的替代者。

为了避免java package中基础类型的额外开销，使用Android的数据结构类型是个不错的选择。**HashMap里面的每个条目都会多用4倍的内存空间**，像你的int类型一样。你放在 hashmap里面的内容越多，你浪费的内存资源也就越多。看看你现在已经使用的数据结构，如果你打算在某些情况下重写你自己的数据类的话，不要犹豫。

- 小心使用XML和JSON

他们相对来说太大了，对于你的某些应用来说，他们或许太结构化了。如果你使用有线设备上的数据格式会更加简洁，但是至少你能在你做网络传输的时候gzip这些数据。当然，如果你打算序列化你的对象到磁盘上，你可能需要找找其他的替代方案了。

- 避免使用JNI

有些时候你需要JNI，但是如果不方便就不要使用它了。JNI有些有趣的事情：每一次你越过JAVA和Native的边界的时候，我们都需要检查参数的有效性，这会对GC的行为有影响，而且带来额外的消耗。有的时候这些消耗是非常昂贵的，所以当你使用了很多JNI的调用的时候，你可能在JNI调用本身上花的时间比在你native代码执行的时间都要多。如果你有些老的JNI代码，请仔细检查他们。

有一件你能提高JNI效率的事情就是尽可能的把多次调用集中到一次。避免在JAVA和Native中间来回调用，一次搞定。例如，在Android的Graphic Pipeline中，我们在每次调用JNI的时候传入了尽可能多的参数，从而避免了调用JNI的时候，JNI从JAVA对象中抽取参数的开销。我们使用了基础类型，避免了使用奇怪的对象，我们传入的是Left, bottom 和right参数，所以我们不需要又返回到JAVA层了

- 基础类型 vs. 封装后的基础类型

请使用基础类型来代替封装后的基础类型。一个不那么明显的事实是：如果你使用那些集合类并且比较它们是否相等的时候，我们每次都会做autobox。每次都需要分配一个对象去使用，因为它们会被强制转换成boxed equivalent。这里有个例外，你可以使用big boolean，因为它们只有两个。

- 避免使用反射（Reflection）

比避免使用JNI更进一步，我需要如我建议避免使用JNI一样指出来，animation framework使用了JNI来避免使用反射。这样当然多了双重内存分配的开销，因为我们需要分配这些对象并且每处都要autobox他们，同时从Java运行环境发起的函数调用采用的内部机制的开销特别大。

- 小心使用 finalizers

内部我们只在很有限的情况下才用。关于finalizers有个事实不太明显的是：它需要两次完整的GC才能收集完所有的事情。如果你在一个finalizer里面放置某些assert来收集信息的话，我们需要运行两次完整的GC才能回来处理。有些时候这是必要的，但是在其他的一些情况下，把这些处理放在离finalizer之外的近点的地方会更方便。不然的话开销太大了。

## 网络 (47:19)

- 不要过度同步

正如前面描述的那样，你的用户的网络可能不那么好，或者他们的网络会很贵。而且，你会给系统带来负担。也许你认为你的应用需要尽可能快地得到那些数据，但是事实上是你会给系统带来很多的负担，仅仅为了保持你的应用处于激活状态。

- 允许延时下载

这在信号不好的网络和收费很贵的网络下十分重要。你可以把数据打包起来。把所有需要下载的东西收集在一起，然后做一次同步。

- 谷歌云消息 Google Cloud Messaging

GCM收集了很多东西。他会使用传送层来和后台服务器传送数据，所以你也许可以重用这个系统来避免自己重复工作，也避免了每个应用都创建自己的socket。

- GCM 网络管理 (Job 调度)

这也是个很好的东西可以利用起来。Job 调度可以让你把你东西打包起来。你可以说“我想在空闲的时候才使用这些事务，或者当我被激活的，或者当我在wifi连上的时候”，然后在普通的时间间隙里面收集这些事情。这会更有效，对用户也不会太突兀。

- 不要轮询

永远不要轮询

只同步你需要的

你知道你的应用里面发生了什么，所以仅仅只要同步当前应用需要的数据，而不是把所有可能需要的数据都同步下来。

## 网络质量 (50:05)

不要对网络有任何的假设。为低端网络开发，然后保证你的应用在低端网络下测试过。即使是你在使用模拟器，你也需要保证你在我门称为“糟糕的网络”的环境下测试过。在这些情况下，你应用的表现会让你大吃一惊的。

## 数据 & 协议 (51:13)

如果你拥有服务器，做所有能帮助到设备的事情。比如改变数据格式，改变发送数据的类型。设备来告诉服务器它打算浏览图片的大小，也是个好方法。我们看到过内部的应用接收到了比屏幕大小大4倍的图片，然后在设备端重新处理图片的大小。这个工作明显服务器来做比较合适。

使用缩小和压缩的算法。gzip是个好的选择。如果你能在HTTP上传送GZIP的数据，请这样做。这会很有用，特别是在糟糕的网络上。GCM也可以帮忙。它会帮助你保持连接，所以，一个好处就是使用它会有更短的延时，因为不用在每次连接的时候都做一次握手而且在延时方面，移动网络是出了名的糟糕的。

## 存储 (52:21)

- 不要写死你的文件路径

路径会改变的，如果用户想把它们存到别处呢？

- 仅仅固定相对路径

你知道你的数据相对你的APK的路径，这是正确的，安卓有APIs得到那些路径。你不需要做hard code的事情，你只需要坚守和你APK实际安装位置的相对路径。

- 使用存储缓存来处理临时文件

有APIs可以调用，请使用它们。

- 简单情况下不要使用SQLite

SQLite的消耗在某种程度上也是很大的，所以如果你只需要些简单的方式，比如 key-value 存储会更加合适些。

- 避免使用太多的数据库

数据库整体上开销是很大的。也许你可以试试只用一个数据库，然后服务于多个不同的用户。

- 让用户选择内容存储位置

当用户有可移除的存储设备的时候这些尤为重要。或者他们可以使用adoptable storage，这是在棉花糖版本上的新的方法。如果用户打算采用新的存储设备，然后把数据都迁移到它上面，你的应用应该允许他们这样做，这样你的应用就不会乱掉了。

## 问和答 (54:00)

- 问：你提到了一个平滑的策略是降低你的帧频率，有可能一个应用说：“我需要30fps而不是60fps吗？”

**Romain**：某种程度上。有一些你可以使用的API来达到和屏幕同步的目的。你可以每隔一帧同步一次。在普通的应用中，这会是很困难的，除非你知道你会因为外面各种各样的设备而有许多的工作量。但是，如果你使用的是OpenGL或者Canvas，你又有自己的线程做渲染，并且你对渲染线程有完整的控制权的时候，情况就不一样了。在这种情况下，你可以控制你的帧，你可以等待，然后你记录V-Sync，诸如此类的动作。如果你想深入的了解这些高级的技术细节，你可以看看那些游戏开发者的文章，他们在做类似的事情。

**Chet**：一般情况下，你可以尝试60fps，然后你可以使用一个叫GPU Profile的工具，这个工具在Android Studio上就有。工具里有一个彩色的显示条，你需要保证你持续的呆在绿线以下。你可以在[developers.android.com](https://developer.android.com) 查到工具的详细信息。

**Romain**：还有，我们经常用 Systrace。Systrace是一个底层的，轻量级的，系统层面的监测工具。如果你的应用在任何地方有性能问题，它不需要嵌入在你的代码中。Systrace不但能显示你在那你花费了时间，而且还能显示你的线程是否被调度，CPU的运行频率，你是不是被从一个CPU转移到另一个CPU上（这也会影响到你的性能），是否后台线程导致了锁定，或者优先级倒置。文档在[developer.android.com](https://developer.android.com) 上也能找到。这也是个很有用的工具。

- 问：有一个类叫‘android.os.memfile’。它和linux 共享内存联系在一起。当我使用它的时候，我能从linux 内核中得到300-400MB的内存来共享。所以，我使用它来分享拍摄的视频并且存到了临时区域。同时我把它设为 non-purgable. 因为使用了400MB的内存，Android总会参与进来刷新内存。你对使用这个技术来存储奇怪的内存有什么建议吗，或者不要使用共享内存？

**Romain**：这是个非常有趣的问题。我没有什么想法。我也不知道系统有这样的行为，也许你应该问问内核组来理解这里的 behavior。如果是如设计一样，那么后果是什么。对不起，我没有一个更好的答案。

- 问：我想指出当Android在切换Activity状态的时候，会使用共享内存。他们发现如果你一直监视着你的Android设备中共享内存的使用状态，你可以准确地猜出Activity是处于什么状态，如：是从onResume到onStop，所以这看起来像是个安全漏洞。显

然，安卓再每次做Activity的状态转换的时候都使用了共享内存，有时候是12bytes。所以实时监测共享内存的状态，你就可以知道Activity是在什么状态。

**Romain:** 如果你一直监视着内存状态，你可能还可以知道其他的发生在设备上的事情。比如看着屏幕。但是，很高兴你能分享这点。

- 问：你之前讨论到应该避免使用JNI，也需要避免使用Reflection。然后你提到你在动画实现中忍受了许多开销来避免使用Reflection。我想知道如果使用了Reflection，会在动画对象中发生些什么呢？

**Chet:** 我们没有用Reflection，我们用的是JNI。JNI里面有Reflection的代码。它是这样工作的：他会调用到JNI说，“我需要一个方法”。所以，当你想使用一个动画的属性叫做“foo”。你传入了一个串，然后说“好吧，我想找到一个设置的方法叫做setFoo”。进入JNI层后，看起来JNI层会说：“有这样的方法的签名吗？”如果JNI找不到，就会返回false然后就会使用Reflection。我认为Reflection的代码应该永远都不被调用到。所以JNI需要返回false。我以前也使用过Reflection，只要是我写的代码，我会使用已经存在的接口来保证没有其他奇怪的事情发生。然后，在使用Reflection的时候我还会有些backup的机制。但是我不认为Reflection需要这样用。这种情况下可以使用JNI。在有些代码中，也许不太明显，但是如果你找到代码正确的地方，你会发现一个情况是“JNI够用吗？如果他够用，就使用JNI好了。”

**Romain:** 当你使用JNI的时候，你其实可以有效地访问到所有的成员和方法，这就是Reflection做的事情。我们发现使用JNI会比Reflection快上很多。

**Chet:** 而且部分原因是内存的消耗。因为至少算上运行的开销，JNI是不会额外再分配内存的。通过任何机制实现方法查找都不简单。但是，我们只会在你第一次调用的时候做方法查找。所以，当你创建动画对象的时候，第一次你运行的时候，就开始寻找setter或者getter，然后缓存它们。这样同样的情况不会再次发生。

**Romain:** 另一个事情是当你通过Reflection调用一个set alpha的函数的时候，你会在方法对象上调用该方法，这需要一个对象。所以当你需要传入一个Float时，你需要分配内存来封装（boxing）你的Float。但是如果你用的是JNI，你就不需要boxing。

**Chet:** 当然还有第三种机制，如果你打算习惯它的话，就是属性（properties）。这就是我们为什么很早前就加入属性（properties）。为一些特殊的视图我们创建了属性，alpha属性，translation X属性，rotation X属性等等。他们直接使用setters。所以当你使用属性的时候，他直接调用了静态的属性对象。你在视图的实例中传递它们，它会调用视图的seter，这样是开销最小的方法。

- 问：你刚才说你们不太关心Java的版本发布，因为Java语言和实时运行是分开的。但是我还是很好奇你们如何看待retrolambda，它让你使用lambdas，这是Java 8的功能，但是它把Lambdas编译成二进制代码。而且作为dexing的一部分，它把他们重编

译成匿名类。你有什么建议吗？

**Romain:** 我知道retrolambda，而且我喜欢它。我觉得它太棒了。使用它之前，我会反编译retrolambda的结果看看发生了什么。因为使用lambda，我敢肯定你会有些讨厌的惊喜。我非常肯定它们在某些地方会变成匿名类，所以在你会遇到大量的内存分配，分配情况取决于它们是如何介入的。例如，如果我在一个循环中传入一个lambda，内存分配会发生什么呢？所以，我会去看看反汇编代码，然后在我决定前看看发生了什么。你知道，匿名类可以被缓存，或者它会在每次使用的时候分配，我不知道具体会发生那种情况。我需要看看。作为一个软件工程师来说，在决定如何使用前，尽量的去理解背后到底发生了什么是非常重要的。因为这样做，就不会在之后的开发中遇到讨厌的惊喜。当然，在某些情况下，如果你是在用户点击一个按钮时使用了lambda，并不重要，对吗？当你点击一个按钮时，性能不是个特别大的问题。

- 问：我想知道的是什么时候你们会支持Java 8，然后你不需要把他们编译成匿名类。

**Romain:** 不知道，实话实说，这几个月我们确实讨论了不少关于桌面系统的JAVA 8的事情。我使用过streams，parallel streams和lambdas。这些都太棒了。代码看起来特别棒，写起来都特别有趣。但是当你意识到在某些情况下，他们确实会比老的循环更慢些的时候，感觉就不一样了。再说一遍，小心一些，尽量去理解你现在正在做的取舍。关于Java 8语言什么时候会被Android支持，我不知道，因为我们没有讨论过这个事情。而且即使我知道，我也不能告诉你。

**Chet:** 我有必要提一下，我最喜欢的一个工具就是DDMS里面的Allocation Tracker，它现在可能是在Monitor的什么地方。反编译二进制代码是一个好的方法，这样可以看看到底编译器干了什么，但是运行的时候看看发生了什么也是个好方法。当我们刚开始开发动画框架的时候，我们使用Allocation Tracker去找到我们在每一帧的时候分配了些什么，然后消除它们。所以，如果你用retrolambda，你需要确定在你应用的关键点，你不要分配那些从外部看不需要的内存。

- 问：我们日常的工作都是feature驱动的，对吗？内部，我们如何去维护一个性能的标杆？你们有专职的QA来跟踪你们的代码，然后你可以看栈的trace，然后时刻提醒你，或者作为一个feature的开发者在你们提交给QA前你们有什么标准吗？

**Romain:** 这是个混合的问题。对于那些非常关心他们应用的性能的工程师来说，这是个他们开发过程中非常小心的问题。也有QA会关心这个话题，但是这几年来，Android组写了很多自动化的测试用例来测试性能。例如，拿Butter项目来说，图像组为每一个CL做了一个jank的dashboard。他会运行一些像在Gmail中滚动滚动条的用例，然后计算我们错过的帧。每一次数字的变动，我们都会收到一封邮件，然后有人就会受到批评。当然应用的开发者会先收到批评，然后他们会发给我们框架组说：“你们太烂，你们的东西太慢”。然后我们会回复它，这样会来来回回讨论好长时间。

**Chet:** 我们也会写出许多工具并且改进它们。Systrace就是一个我们内部正在使用并且持续改进的一个很好的工具，之后我们提供给所有人使用。并且我们持续迭代地开发它。这样我们就像最近做的事情一样，在一些不明显的情况下，给你一些关于你的问题的提醒和信息。

**Romain:** 事实上，大部分的工具你可以通过工程的UI看到很多信息。所以诸如Hierarchy Viewer, devel-draw, debug tool 和 GPU profiler都在设备上可以运行。所有的这些工具都是我们内部需要的，所以我们开发出来并且提供给所有人使用。当然，还有很多的事情可以做，但是关于性能最重要的事情就是你们能写出自动的测试用例来捕捉它们。这是非常困难的一件事情。我们暴露了一些内部的计数器，我想我们有文档描述它们。在adb shell的dumpsys帮助下，你可以获取其中的一些信息，帧的时间戳，janks的次数，特别是在棉花糖和棒棒糖中我们加入了更多的方法。你可以做的事情就像UI automater一样，你可以创建一个用例，驱动你的应用，然后输出这些计数器看看能否告诉你些什么有用的信息。更有效的方式是有个dashboard。它抓取了这些命令的输出，然后给你画一个特别好看的图表。另一件事情是，当你做些事情的时候想想你的性能。说起来简单，做起来难，评判起来更难，理解你在量化什么还要困难一些。但是，这是唯一的途径。

- 问：你提到不要使用序列化。你能再详细的描述一下吗？在Activity和Fragment间使用Serializable和Parcelable传输数据包含在里面吗？

**Romain:** 我们是在讨论Serializable的接口。Parcelable是Android提供的Serialization的一个变种，是非常有效的，但是仅仅用来进程间的数据传输。现在我们有新的方式了，搜索“persistent parcel”，你就可以找到它了。但是对了，我们讨论的是Serializable接口。

- 问：你之前提到flat buffers。有一种说法是如果网络请求需要很长时间的话，节省100多毫秒不太重要。

**Romain:** 我之前给的例子比较有意思是因为他用了JSON作为磁盘上文件存储格式。这里的想法是你已经缓存了数据了，所以你已经承受了网络的开销，所以可以优化的地方在你多次读取这些数据的时候。在这种情况下，使用更快的方式是有意义的。但是你是对的，如果解析JSON需要10毫秒，解析你的flat buffer只需要5毫秒，但是网络传输需要200多毫秒，这样的优化还有作用吗？显然，我希望你的应用在后台线程处理所有的网络操作。所以这里讨论的更多的是UI的延时，而不是别的。如果你有什么东西是经常访问的，你当然需要在flat buffer这样的地方里面寻找它。

还有一个Cap'n Proto，也是protobuf v2的开发者开发的。他写了一些类似flat buffer的东西，但是更加深入。因为它可以被作为有线传输格式，可以被用作RPC机制，而且我相信它有Java封装层。这就回到我们之前的坚持，当你考虑性能的时候，你需要量化标准。确定你修改的是正确的问题。如果你修改的地方不是个问题可能还无所谓。作为我们来说，在你的应用里面到底什么会是个问题更难。因为这完全依赖于你的应用。我们只能分

享我们在应用中一次又一次看到的问题，再一次强调，这不意味着你的应用也有同样的问题。但是真的，如果你在磁盘上做serialization，看看其他的格式吧。总而言之，找到你最好的方式。



# Android UI性能优化详解

来源:<http://music4kid.github.io/>

设计师，开发人员，需求研究和测试都会影响到一个app最后的UI展示，所有人都很乐于去建议app应该怎么去展示UI。UI也是app和用户打交道的部分，直接对用户形成品牌意识，需要仔细的设计。无论你的app UI是简单还是复杂，重要的是性能一定要好。

## UI性能测试

性能优化都需要有一个目标，UI的性能优化也是一样。你可能会觉得“我的app加载很快”很重要，但我们还需要了解终端用户的期望，是否可以去量化这些期望呢？我们可以从人机交互心理学的角度来考虑这个问题。研究表明，0-100毫秒以内的延迟对人来说是瞬时的，100-300毫秒则会感觉明显卡顿，300-1000毫秒会让用户觉得“手机卡死了”，超过1000ms就会让用户想去干别的事情了。

这是人类心理学最基础的理论，我们可以从这个角度去优化页面 / view / app的加载时间。Ilya Grigorik 有一个很棒的演讲，是关于搭建1000毫秒内加载完成移动网站的。如果你的网页能在1秒内加载好，就超过了人类感知的预期，你的用户一定会感觉很满意。还有研究表明，如果网页在3-4秒内还没加载出任何内容，用户就会放弃了。把这些数据应用到app的加载，不难明白加载时间是越短越好。这篇文章主要关注UI的加载时间。当然UI性能优化还会涉及到其他方面，比如必需在后台运行到任务，要从服务器下载一个文件等等，这些我们在后面的文章再聊。

## 卡顿 (Jank)

内容的快速加载很重要，渲染的流畅性也很重要。android团队把滞缓，不流畅的动画定义为jank，一般是由于丢帧引起的。安卓设备的屏幕刷新率一般是60帧每秒 ( $1/60\text{fps} = 16.6\text{ms每帧}$ )，所以你想要渲染的内容能在16ms内完成十分关键。每丢一帧，用户就会感觉的动画在跳动，会出现违和感。为了保证动画的流畅性，我们接下来看下从哪些方面优化可以让内容在16ms内渲染完成，同时分析一些常见的导致UI卡顿的问题。

## android设备的UI渲染性能

早期android用户抱怨最多的就是UI，尤其是触碰反馈和动画流畅度，感觉都很卡。后来随着android系统逐渐成熟，开发人员也投入了大量的时间和精力让交互变的流畅起来。下面列举一些不同系统版本所带来的提升：

在Gingerbread或者更早的设备上，屏幕完全是由软件绘制（CPU绘制）的（不需要GPU的参与）。后来随着屏幕尺寸变大和像素的提升，纯粹靠软件绘制遇到了瓶颈。

Honeycomb加入了平板设备，进一步增加了屏幕尺寸。同时出于性能考虑，加入了GPU芯片，app在渲染内容的时候多了一个GPU硬件加速的选项。

对于针对Ice Cream Sandwich或者更高系统的设备，GPU硬件加速是默认打开的。将软件绘制（CPU）的压力大部分转移到了GPU上。

Jelly Bean 4.1 (and 4.2) “Project Butter” 做了近一步的提升来避免卡顿，通过引入 VSYNC机制和增加额外的frame buffer(vsync和frame buffer的解释可以参考[这篇文章](#))，运行 Jelly Bean的设备丢帧的概率变的更小。引入这些机制的同时，android开发团队还加入了一些优秀的工具来测量屏幕的绘制，开发者可以使用这些工具来检测VSYNC buffering和卡顿。

我们从普通开发者的角度，来逐一看下这些提升和相关的测量工具。我们的目标很明显：

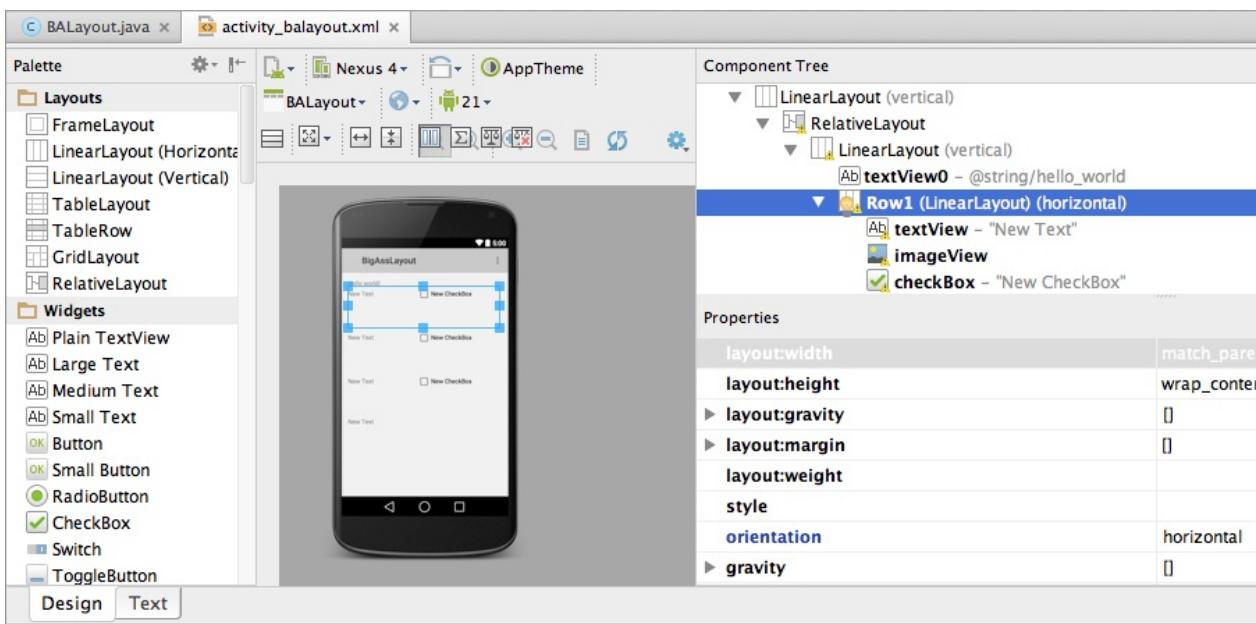
- 屏幕绘制低延迟
- 保证流程稳定的帧率来避免卡顿

当android开发团队引入这些UI流畅性的提升时，他们需要能量化这些提升的工具。经由他们的努力，这些工具都打包进了SDK以方便开发者们来检测UI相关的性能问题。接下来我们就使用这些工具来优化几个demo程序。

## 搭建Views

大家应该都对android studio里xml布局编辑器很熟悉了，知道怎么在android studio (Eclipse) 中搭建和检测View结构。下图是一个简单的app view，包含一些套嵌的子view。搭建这些view的时候，一定要留意屏幕右上角的组件树（Component Tree）。套嵌的子view越深，组件树就越复杂，渲染起来也就越费时间。

图4-1



对于app里的每一个view，android系统都会经过三部曲来渲染：measure, layout, draw。可以在脑中回想下你搭建的view的xml布局文件结构，measure从最顶部的节点开始，顺着layout树形结构依次往下：测量每个view需要在屏幕当中展示的尺寸大小（上图当中：LinearLayout; RelativeLayout, LinearLayout；然后是textView0和LinearLayout Row1点分支，该分支又有另外3个子节点）。每个子节点都需要向自己的父节点提供自己的尺寸来决定展示的位置，遇到冲突的时候，父节点可以强制子节点重新measure（由此可能导致measure的时间消耗为原来的2-3倍）。这就是为什么扁平的view结构会性能更好。节点所处位置越深，套嵌带来的measure越多，计算就会越费时。我们来看一些具体的例子，看measure是怎么影响渲染性能的。

## Remeasuring Views (重新测量views)

并不是只有发生错误的时候才会触发remeasure。RelativeLayouts经常需要measure所有子节点两次才能把子节点合理的布局。如果子节点设置了weights属性，LinearLayouts也需要measure这些节点两次，才能获得精确的展示尺寸。如果LinearLayouts或者RelativeLayouts被套嵌使用，measure所费时间可能会呈指数级增长（两个套嵌的views会有四次measure，三个套嵌的views会有8次的measure）。可以看下面图4-9里面一个夸张点的例子。

一旦view开始被measure，该view所有的子view都会被重新layout，再把该view传递给它的父view，如此重复一直到最顶部的根view。layout完成之后，所有的view都被渲染到屏幕上。需要特别注意到是，并不是只有用户看得见的view才会被渲染，所有的view都会。后面我们会看下“屏幕重复绘制”的问题。app拥有的views越多，measure, layout, draw所花费的时间就越久。要缩短这个时间，关键是保持view的树形结构尽量扁平，而且要移

除所有不需要渲染的view。移除这些view会对加速屏幕渲染产生明显的效果。理想情况下，总共的measure, layout, draw时间应该被很好的控制在16ms以内，以保证滑动屏幕时UI的流畅。

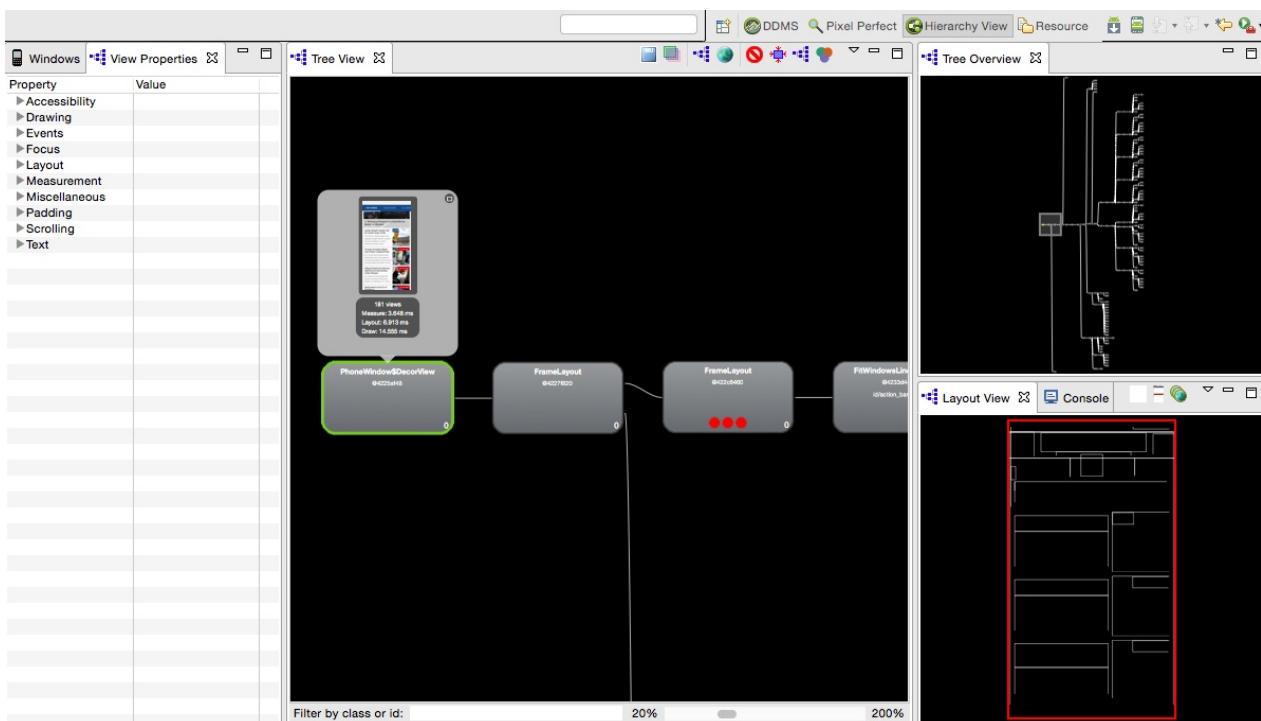
虽然可以通过xml文件查看所有的view，但不一定能轻易的查出哪些view是多余的。要找到那些多余的view（增加渲染延迟的view），可以用android studio monitor里的Hierarchy Viewer工具，可视化的查看所有的view。（monitor是个独立的app，下载android studio的时候会同时下载）

## Hierarchy Viewer

Hierarchy Viewer可以很方便可视化的查看屏幕上套嵌view结构，是查看你的view结构的实用工具。这个工具包含在android studio monitor当中，需要运行在带有开发者版本的android系统的设备上。后续所有的view和屏幕截图都来自一款三星的Note II设备，系统版本是Jelly Bean。在老的设备（处理器慢）上测试渲染性能，更容易发现问题。

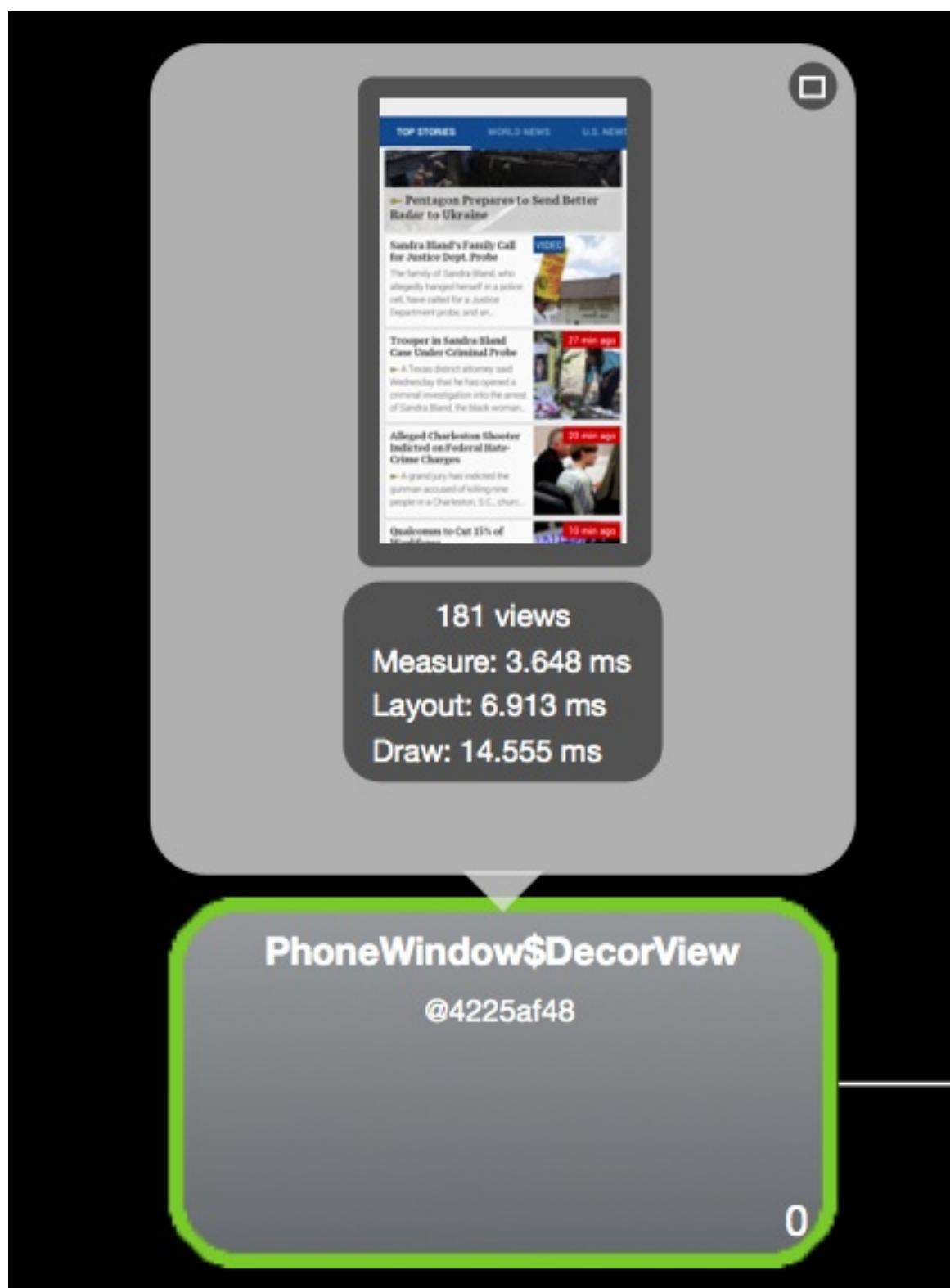
如图4-2所示，打开Hierarchy Viewer之后，会看到几个窗口：左边的窗口列出了连上你电脑的android设备和设备上所有运行的进程。活跃的进程是粗体展示的。第二个tab某一个编译版本的详情（后面细说）。中间的部分是可缩放的view的树形图。点击某一个view能看到在设备上展示的样子和一些额外的数据。右边有两个view：树形结构总览和布局view。树形结构总览显示了整个view的树形结构，里面有一个方块显示了中间窗口在整个树形结构当中所处的位置。布局view当中深红色高亮的区域表示所选中的view被绘制的部分（浅红色展示的是父view）。

图4-2



在中间的这个窗口，你可以点击任何一个view来查看该view在android设备屏幕上的展示。点击树形图工具栏里红绿紫三色的维恩图图标，还能展示子view的数量，和measure, layout, draw三部曲所花费的时间。这个时间是被选择的view及其所有子节点所花费时间的总和。（图4-3中，我选择了最顶部的view来获取整个view结构的时间）

图4-3



最顶部的view总共包含181个view，measure的总时间为3.6ms，layout是7ms，draw花了14.5ms（总共大约25ms）。要缩短渲染这些view的总时间，我们先看下app的树形结构图预览，看看所有的view是怎么拼凑到一起的。从树形结构图上可以看出屏幕里有非常多的view，树的结构比较扁平。前面说过，扁平的结构性能好，树的深度对渲染的性能会产生很大的影响。我们的结构虽然是扁平的，却依然花费了26ms的时间来渲染，说明扁平的结构也有可能会卡顿，也需要去考虑怎么优化。

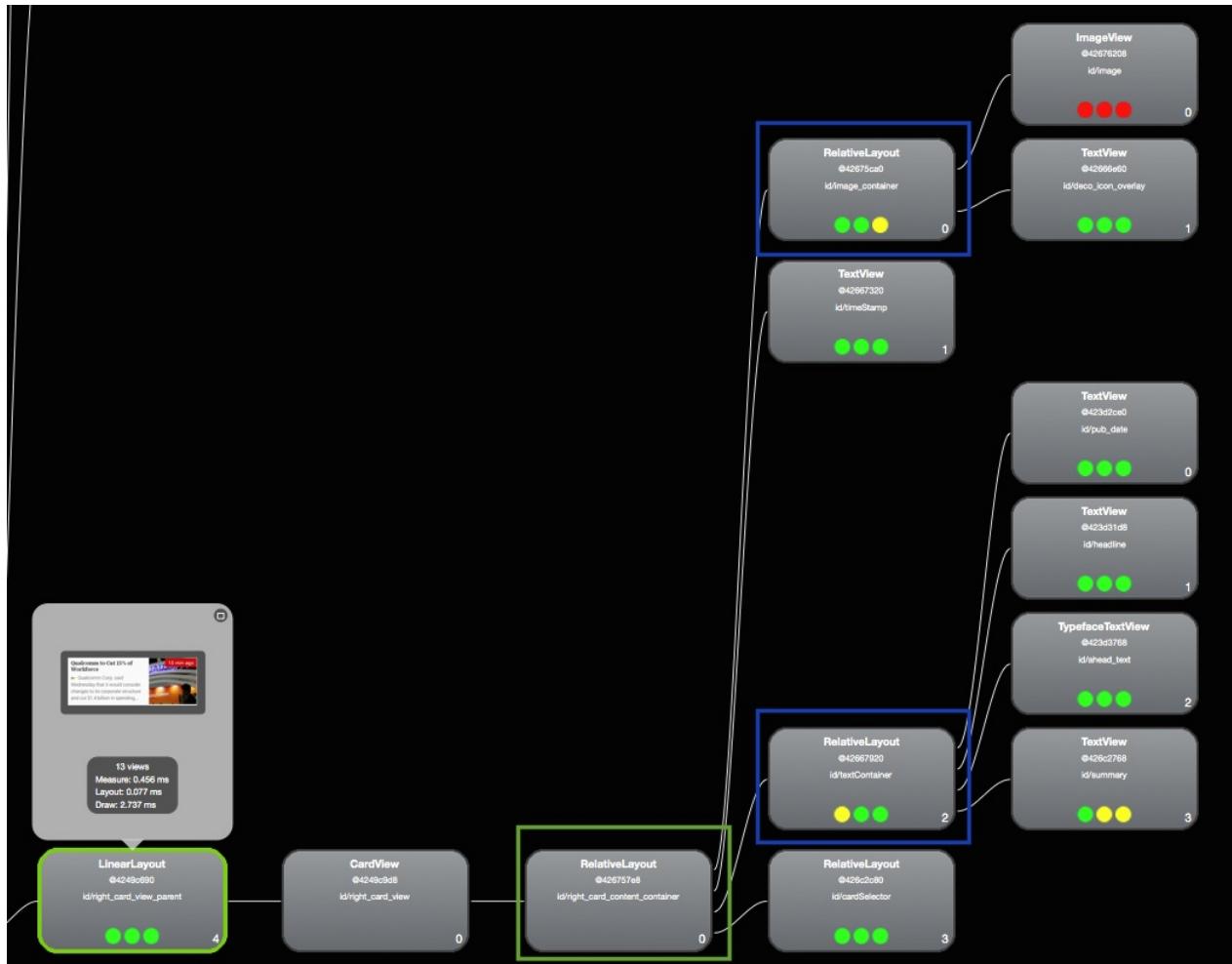
图4-4



排查一个新闻类app的树形结构，大致可以看三个区域：头部（底部蓝色的方框），文章列表（两个橙色的方框表示两个不同的tab），单篇文章的view是用红色方框来标注的。内部标题view的结构重复出现了九次，5个在上面橙色的方框内，4个在下面的方框内。最后，我们可以看到从边上拉出来的导航栏是用底部绿色的方框标出来的。头部用了22个view，两个文章列表个用了67和44个view（每个标题部分使用了13个view），导航抽屉

使用了20个。这样我们还剩下18个view没有计算在内。剩下的这些view其实是在滑动手势动画过程当中生成的。很显然，view的数量很多，要做到不卡顿要让view的绘制非常高效才行。

图4-5



仔细看下标题部分，一个标题是由13个view组成的。每个标题的结构有5层之深，一共花费0.456ms来measure, 0.077ms来layout, 2.737ms来draw。第五层是通过第四层的两个RelativeLayouts来连接的（蓝色高亮），这些又是通过第三层的另一个RelativeLayout来连接的（绿色高亮）。如果我们把第四第五层的view都移到第三层来，我们可以少渲染一整层。而且我之前解释过，RelativeLayout里的measure都会发生两次，套嵌的view会导致measure时间的增加。

现在，你可能已经注意到了每个view里红色、黄色和绿色的圆圈。它们表示该view在那一层树形结构里measure, layout和draw所花费的相对时间（从左到右）。绿色表示最快的前50%，黄色表示最慢的前50%，红色表示那一层里面最慢的view。显然，红色的部分是我们优先优化的对象。

再看下文章标题的树形结构，绘制最慢的view是右上角的ImageView。顺着ImageView一直找到文章父view，父view是通过两个RelativeLayouts来连接的（这里增加了measure的时间），然后是3个没有子节点的view（在最底部）。这3个view可以优化合并成一个view，这样能减少两个layer的渲染。

我们再看另一个新闻类app是怎么来减少标题view里面的子view数量的。从图4-6里能看到一个和图4-5类似的树形结构图。

图4-6

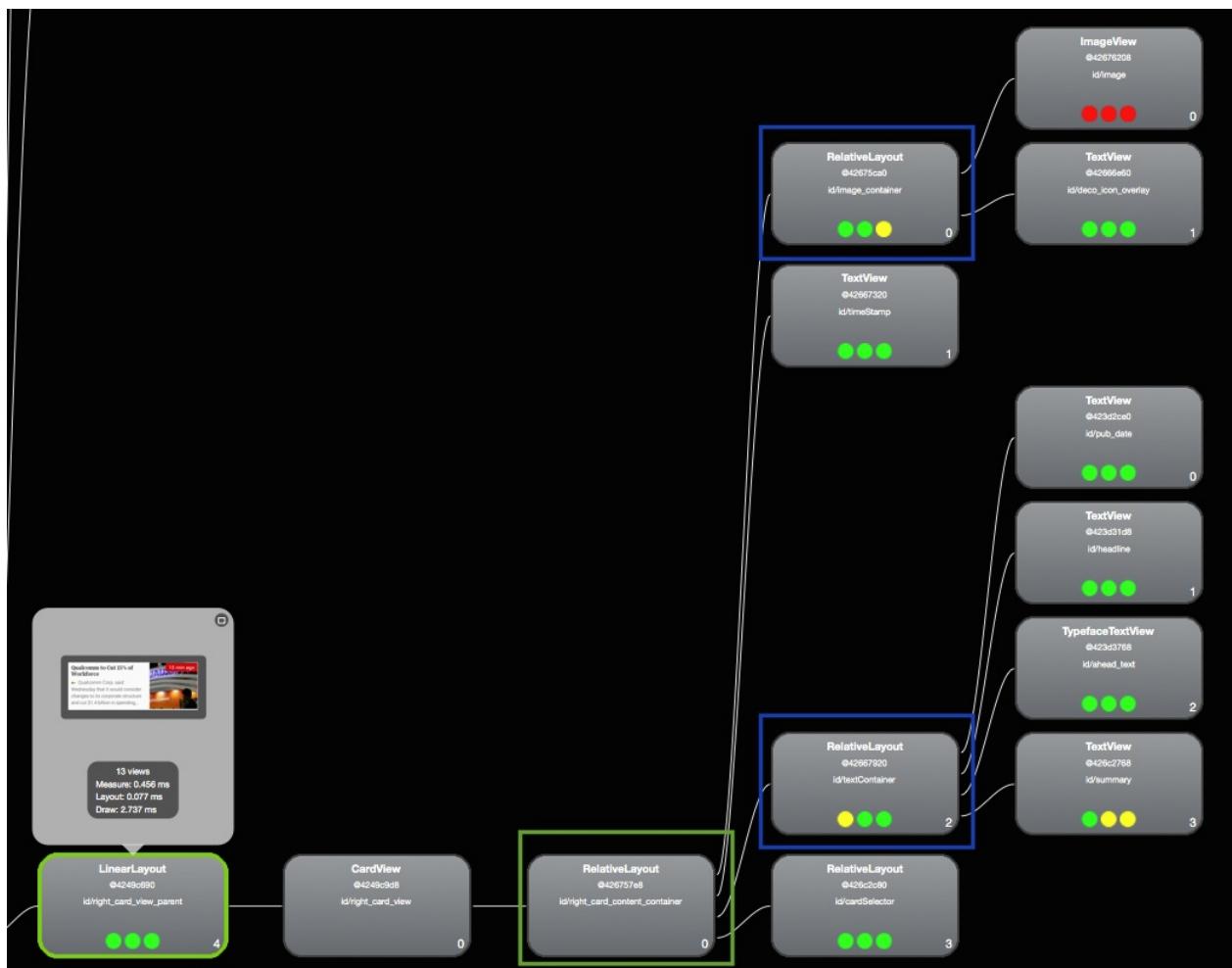
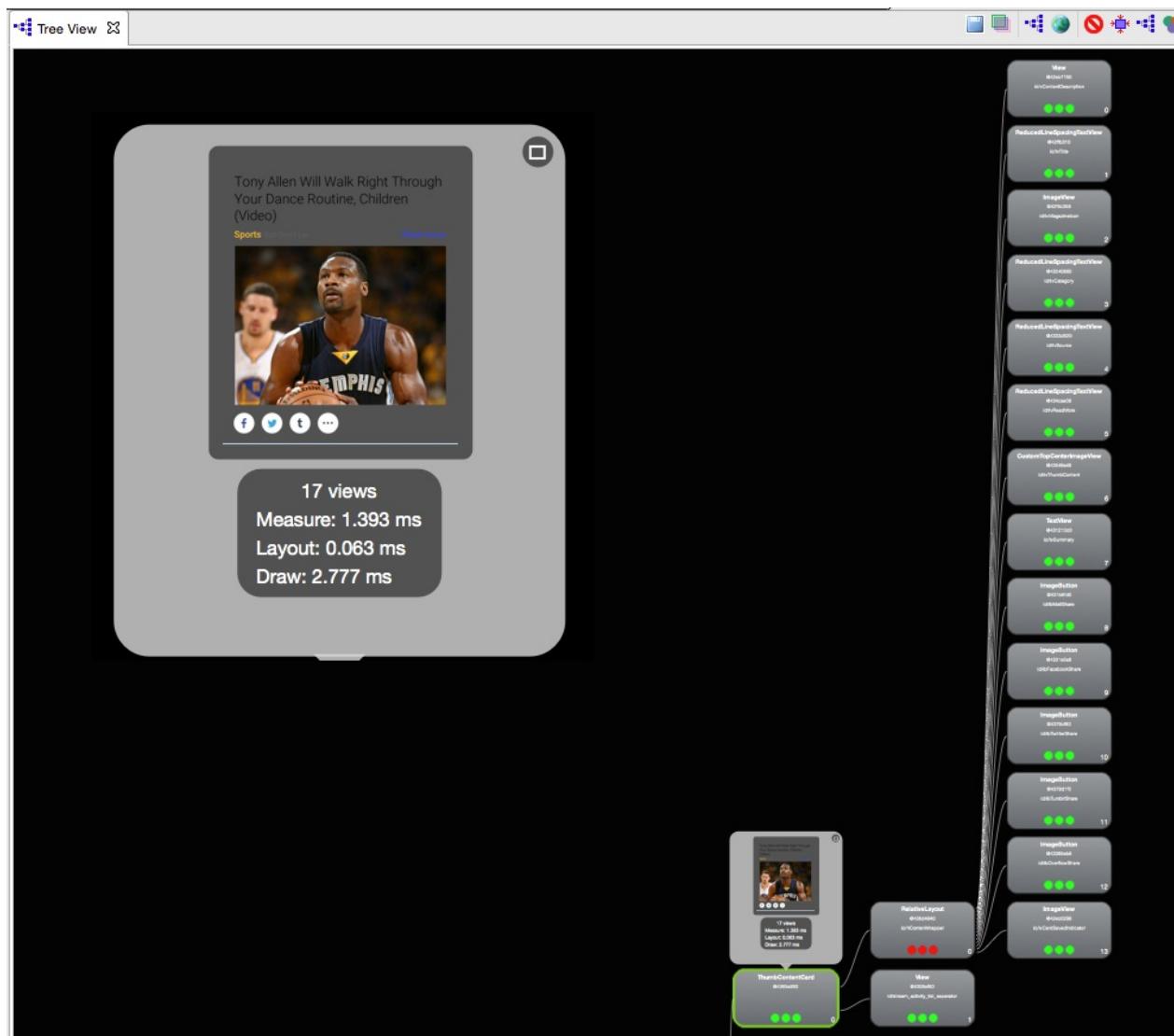


图4-6里的标题view也有RelativeLayouts（绿色的部分）的问题，一共消耗了1.275ms的measure时间，layout用了0.066ms，draw 3.24ms（总共是4.6ms）。在这些数据基础上，我们再做一些调整，加入一个更大的图片展示和分享按钮，但是整个树形结构变得扁平一点（如图4-7所示）。

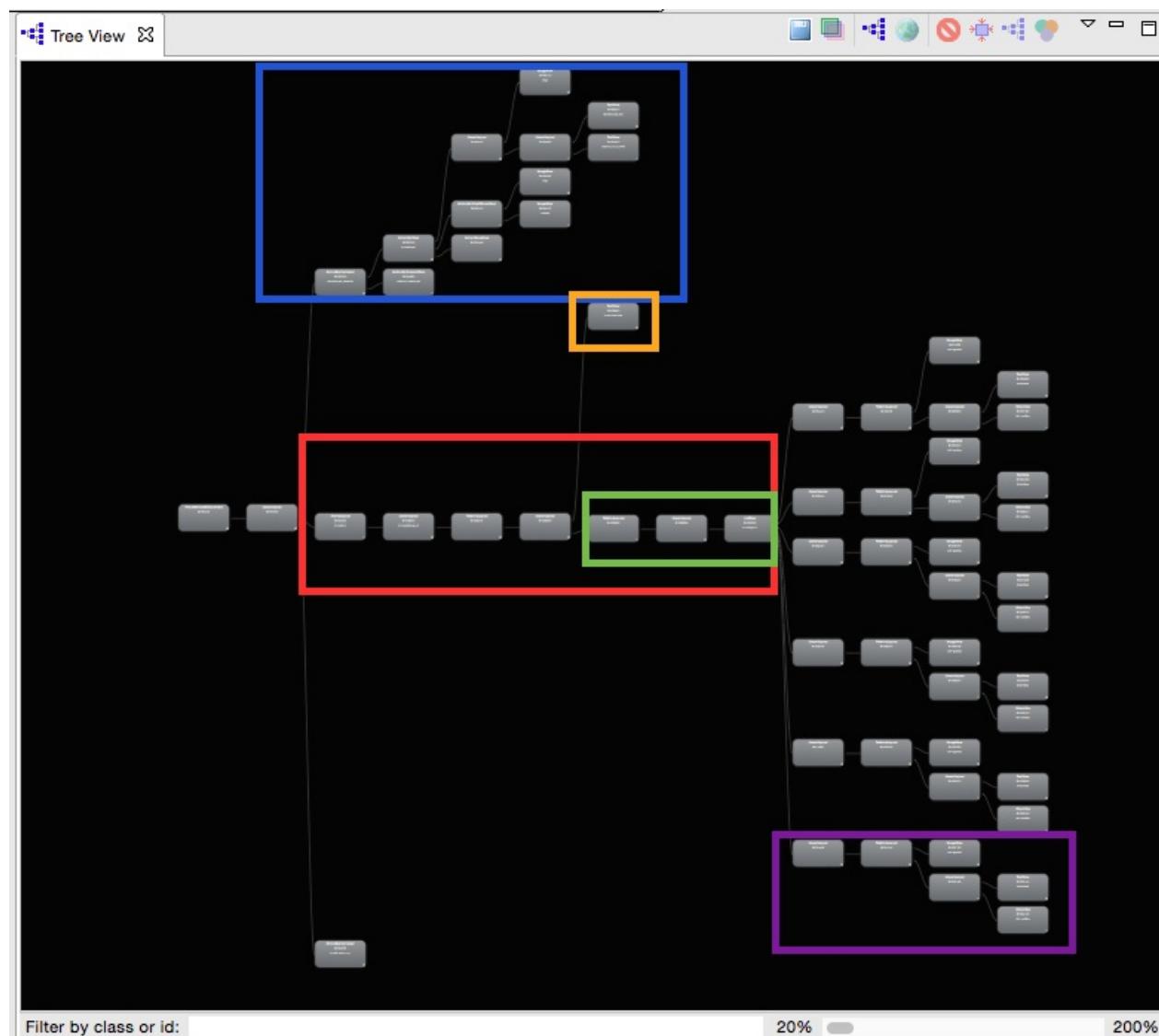
图4-7



容，但节省了400ms！

为了更好的了解这部分的优化，我们再看另一个例子app。这个例子会展示一个山羊图片等列表。界面使用了几种不同的layout方式，性能差的和性能好的都有。仔细的查看这些布局，然后一步步优化它们，我们就能清楚的理解怎么去优化一个app的渲染性能了。我们分几步来进行优化，每一步改变都可以通过Hierarchy View可视化的查看。每换一种 layout方式，xml渲染的性能要么变好，要么变差。我们先从性能差的布局方式开始。先快速的扫一眼图4-8里的Hierarchy View。

图4-8



这个简单的app里有59个view。但是和图4-4里的app不同，这个app的树形结构更扁平，水平方向的view更多一些。叠加的view越多，渲染就会越费时，减少view树形结构的深度，app每一帧的渲染就会变快。

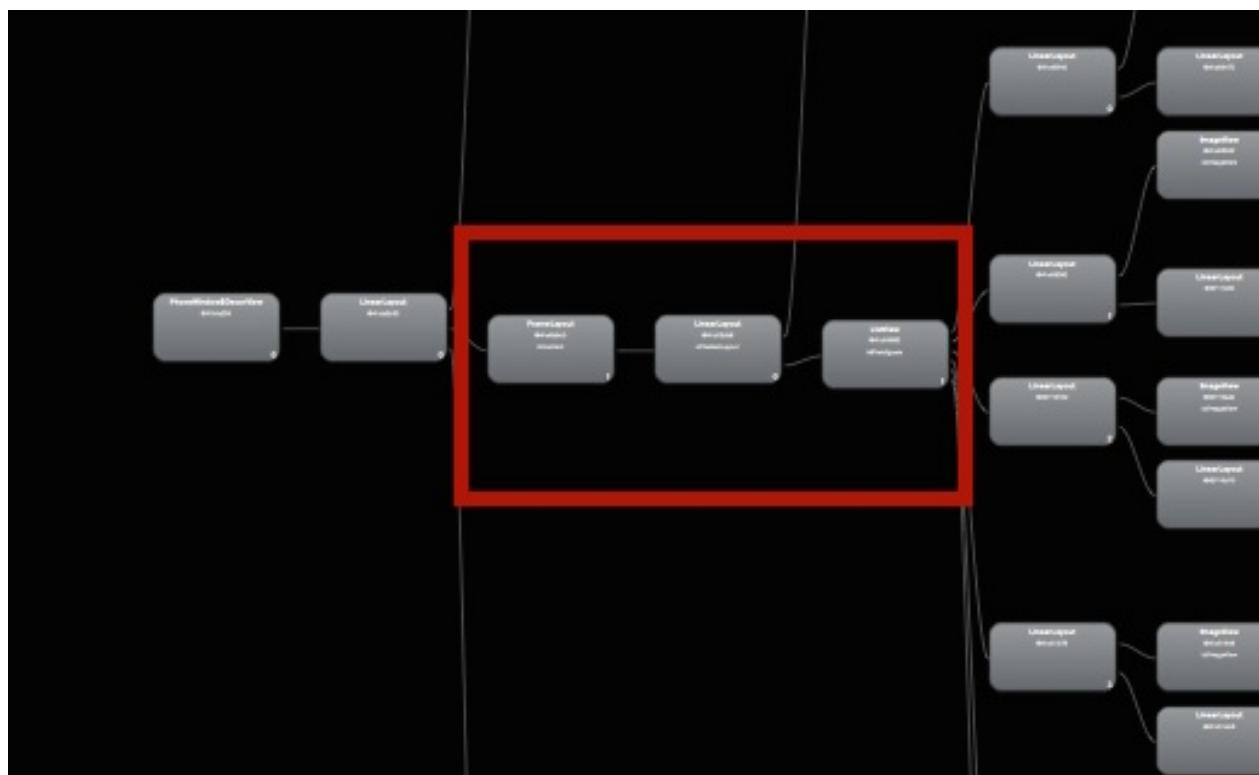
蓝色方框里面的view是action bar。橘色方框里的是屏幕顶部的text box，紫色方框里展示的是山羊的详细信息（有6个这种view）。红色方框标示了7个view，每个都增加了树形结构的深度。我们仔细看些这7个view其中三个的remeasure数据（图4-9）。

图4-9



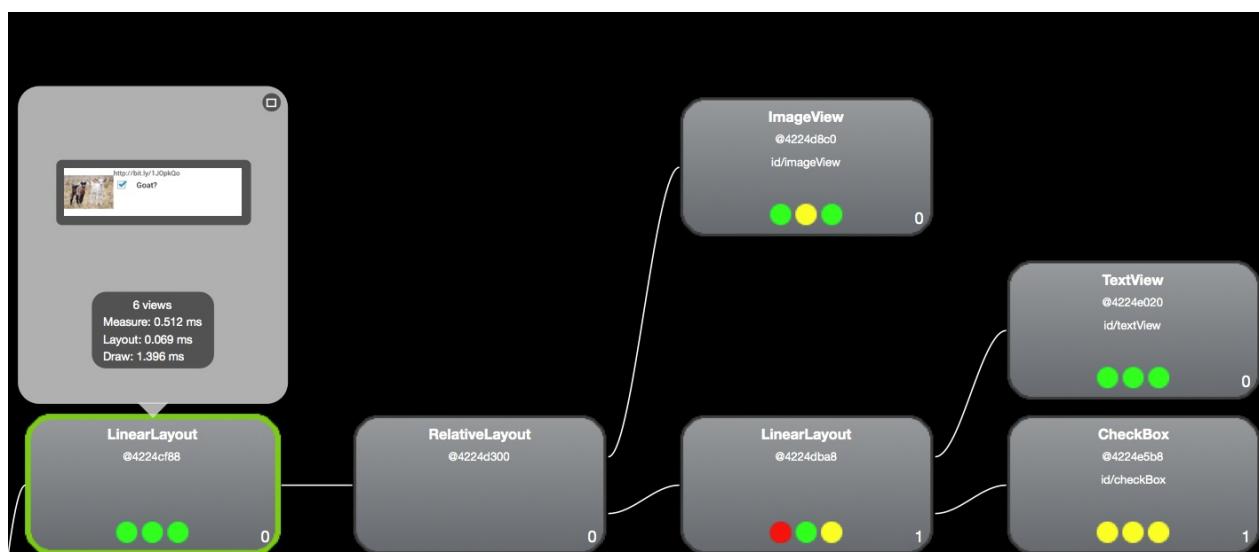
当设备开始measure views的时候，先从右边的子views开始，然后到左边的父views。右边ListView包含6行数据，一共37个view，花了0.012ms来measure。把这个ListView加到中间的LinearLayout之后，变成38个views。有意思的是，measure的时间由于remeasure被触发，瞬间跳到了18.109ms，是原来的三个数量级。LinearLayout左边的RelativeLayout使得measure的时间再次翻倍到33.739ms。再依次往左继续观察（图4-8里红色方框部分），measure的时间叠加到了68ms。但是只要移除上面的一个LinearLayout，measure的时间瞬间降到了1ms。我们可以移除更多的层让树形结构更扁平一些，这样我们可以得到图4-10里的结果，层数减少到了3层。

图4-10



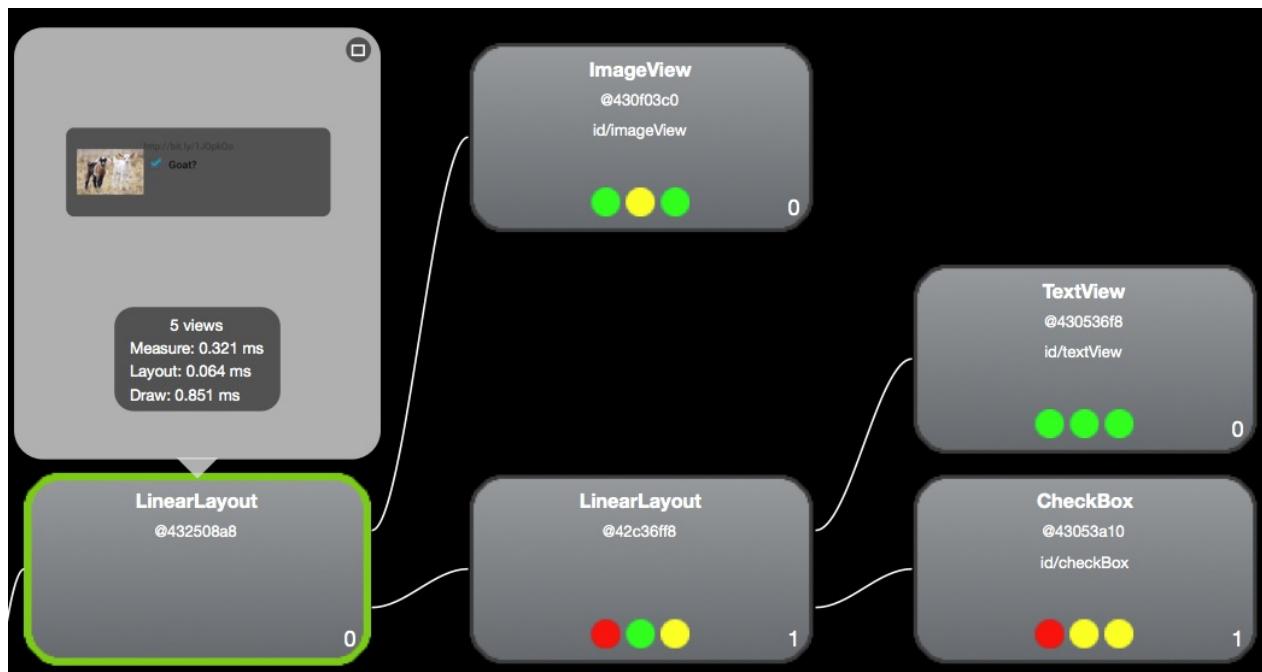
我们可以继续看下山羊信息到row展示部分，来继续减少view结构的深度。每一行山羊信息有6个view，一个有6行数据在屏幕中展示（图4-8中有一行数据是用紫色方框高亮的）。我们用Hierarchy View看下一行view的结构是怎么样的（图4-11），先看下左边两个view（一个`LinearLayout`，一个`RelativeLayout`），这两个view唯一的作用就是加深了树机构的深度。`LinearLayout`连接了`RelativeLayout`，但并没有展示其他什么内容。

图4-11



因为`RelativeLayout`会measure两次（我们现在关注优化measure的时间），我们先移除`RelativeLayout`（图4-12）。这样树形结构的深度从4减到了3，渲染立马快了一些。

图4-12



但效果还并不理想。我们继续移除`LinearLayout`，同时调整下`RelativeLayout`来展示整个row的信息（图4-13），这样深度进一步减少到了2。渲染又快了0.1ms。这样看来优化的途径有很多种，多尝试总是有好处的（看下表格4-1里的结果）。

图4-13

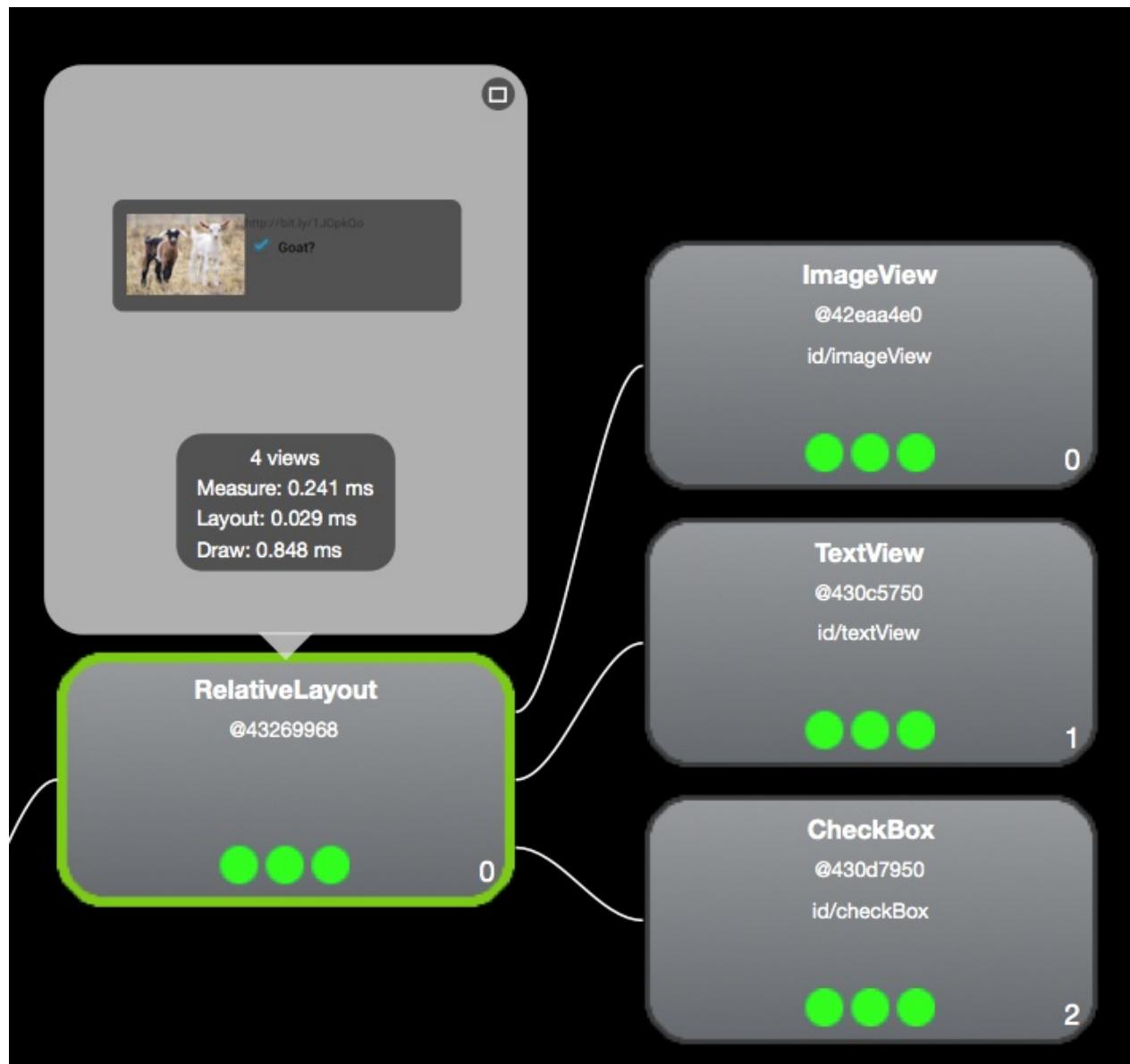


Table 4-1. View Tree optimization improvements

| Version                | View Count | View Depth | Measure | Layout | Draw  | Total    |
|------------------------|------------|------------|---------|--------|-------|----------|
| Unoptimized            | 6          | 4          | 0.570   | 0.068  | 1.477 | 2.115 ms |
| Remove Relative-Layout | 5          | 3          | 0.321   | 0.064  | 0.851 | 1.236 ms |
| Remove Linear-Layouts  | 4          | 2          | 0.241   | 0.029  | 0.848 | 1.118 ms |

每一行减少大约1ms的时间，我们一共可以节省6ms的渲染时间。如果你的app有卡顿，或者你通过工具检测到每次渲染接近16ms了，减少6ms的时间当然会让你的app更快一点。

## View的重用

如果一个程序员面向对象编程经验丰富，他就会尽可能重用创建的view（而不是每次都创建）。拿上面山羊app作为例子，其实每一行展示的layout都是重用的。如果xml文件里最外层的view只是用来承载子view的，那这个view只不过是增加了view结构的深度，这种情况下，我们可以移除这个view，用一个merge标签来代替。这种方式可以移除树形结构里多余的层。

大家可以从github上下载这个山羊app练习下，改变里面xml文件的布局方式，再用Hierarchy Viewer工具看下渲染时间的变化。

## Hierarchy Viewer（不止是树形结构图）

Hierarchy Viewer还有一个功能，可以帮助开发者发现overdraw（重复的绘制）。从左到右看下树形结构窗口的选项，可以发现这些功能：

- 把view的树形结构图保存为png图片。
- 导出为photoshop的格式。
- 重新加载一个view（第二个紫色树形按钮）。
- 在另一个窗口里打开较大的view结构图，还可以设置背景色来发现重复绘制。
- 让一个view的绘制失效（有条红线的按钮）。
- 让view重新layout。
- 让view生出draw命令到logcat（紫色树形按钮到第三个用处）。这样可以查看绘制到底触发了哪些opengl行为。这个功能对opengl的专家做深度优化比较有用。

Hierarchy Viewer对于优化app view的树形结构重要性不言而喻了，很可能会帮你节省几十毫秒的绘制时间。

## 资源缩减

在我们把app的view结构变扁平，view的总数量减少之后，我们还可以尝试减少每个view里面使用的资源数量。2014年的时候，Instagram把标题栏里的资源数量从29减少到了8个。他们测量后发现app的启动时间增加了10% – 20%（因设备而异）。主要是通过资源上色的方式来进行缩减。比如只加载一个资源，然后在运行的时候通过ColorFilter进行上色。我们看下下面的例子是怎么给一个drawable上色的。

```
public Drawable colorDrawable(Resources res,
 @DrawableRes int drawableResId, @ColorRes int colorResId) {
 Drawable drawable = res.getDrawable(drawableResId);
 int color = res.getColor(colorResId);
 drawable.setColorFilter(color, PorterDuff.Mode.SRC_IN);
 return drawable;
}
```

这样一个资源文件就可以表示几种不同的状态了（加星或者不加星，在线或者离线等等）。

## 屏幕的重复绘制

每过几年，就会有传闻说某个博物馆在用X光扫描一副无价的名画之后，发现画作的作者其实重用了老的画布，在名画的底下还藏着另一副没有被发现的画作。有时候，博物馆还能用高级的图像技术来还原画布上的原作。Android里面的view的绘制就是类似的情况。

当android系统绘制屏幕的时候，先画父view，然后子view，再是更深的子view等等。这会导致所有的view都被绘制到了屏幕上，就像画家的画布一样，这些view都被他们的子view覆盖住了。

文艺复兴时期，有很多伟大的画家要等画干了以后才能重用画布。但在我们的高科技触摸屏上，屏幕重画的速度要快几个数量级，但是多次的重新绘制屏幕会使得绘制延迟变大，最终导致布局的卡顿。重新绘制屏幕的行为叫做overdraw，下面我们会看下怎么检测overdraw。

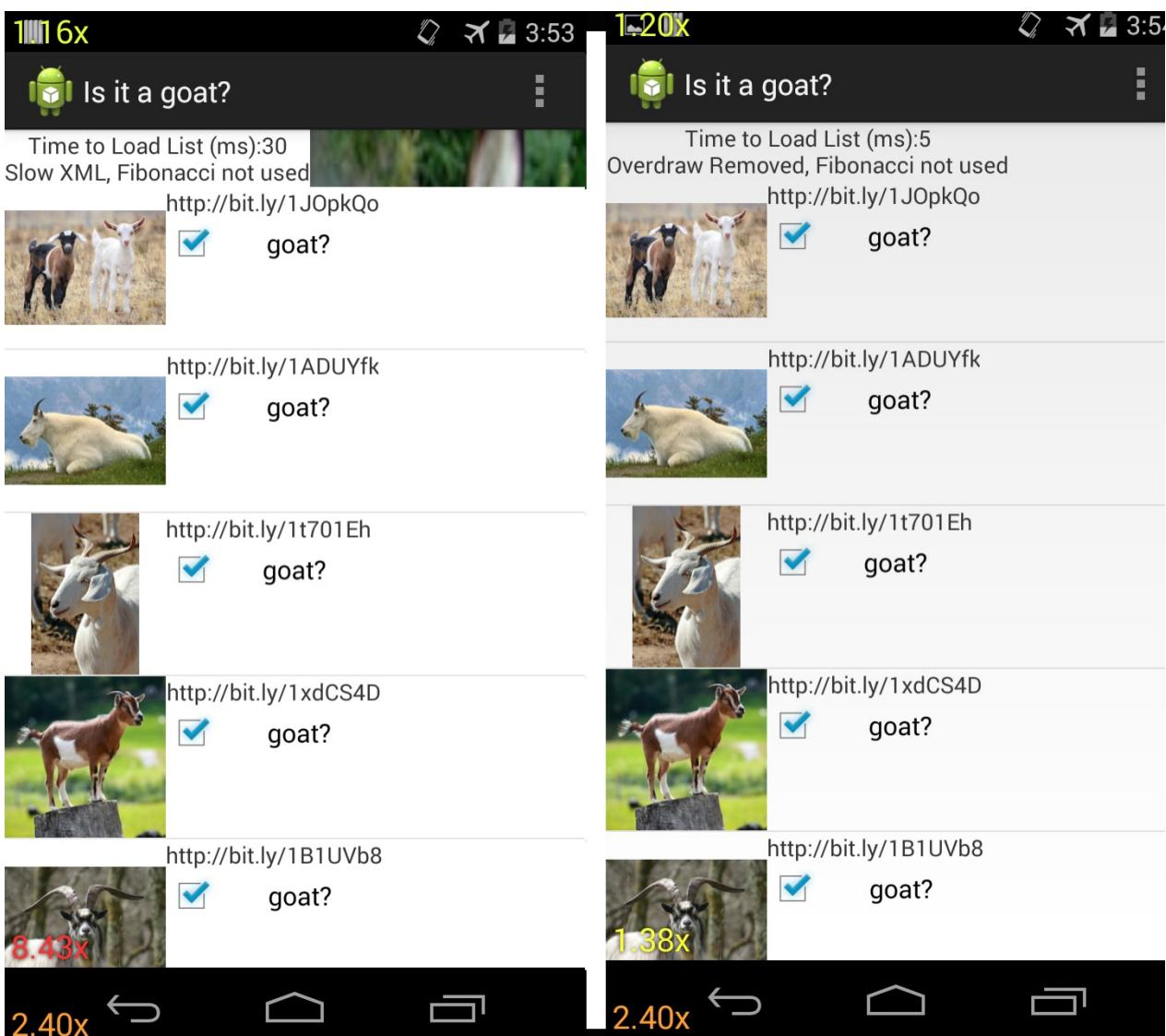
overdraw还带来的另一个问题，当view内容有更新的时候，之前绘制的view就失效了，view的每一个像素都需要重绘。android设备没法判断哪个view是可见的，所以只能绘制每个view的相关像素。类比上面画家的例子，画家只能把老画一幅幅还原出来，再一层层画到画布上，最后再画上最新的画。你的app如果有很多层，每一层的相关像素都需要绘制一遍。如果不小心，这些绘制就会带来性能问题。

## 检测overdraw

android提供了一些很好的工具来检测overdraw。Jelly Bean 4.2里，开发者选项菜单里增加了Debug GPU Overdraw的选项。如果你用的是Jelly Bean 4.3 或者 KitKat 设备，在屏幕的左下角会有一个计数展示屏幕overdraw的程度。我亲试过这个工具对检测overdraw十分有效。虽然有时候这个会多提示6-7次overdraw（发生的概率还不小）。

图4-14中的截图还是来自上面的山羊app。左下方可以看到overdraw的计数。屏幕中可以看到3个overdraw的计数，其中开发者能控制的是主窗口的计数。overdraw的计数是在左下方。没优化过的app overdraw的次数是8.43，我们优化过后可以降到1.38。导航栏overdraw的次数是1.2（菜单按钮是2.4），也就是说文字和图标的overdraw贡献了额外的20%。overdraw计数可以在不影响用户体验的前提下，快速便捷的比较不同app的overdraw，但没办法定位overdraw是哪里产生的。

图4-14

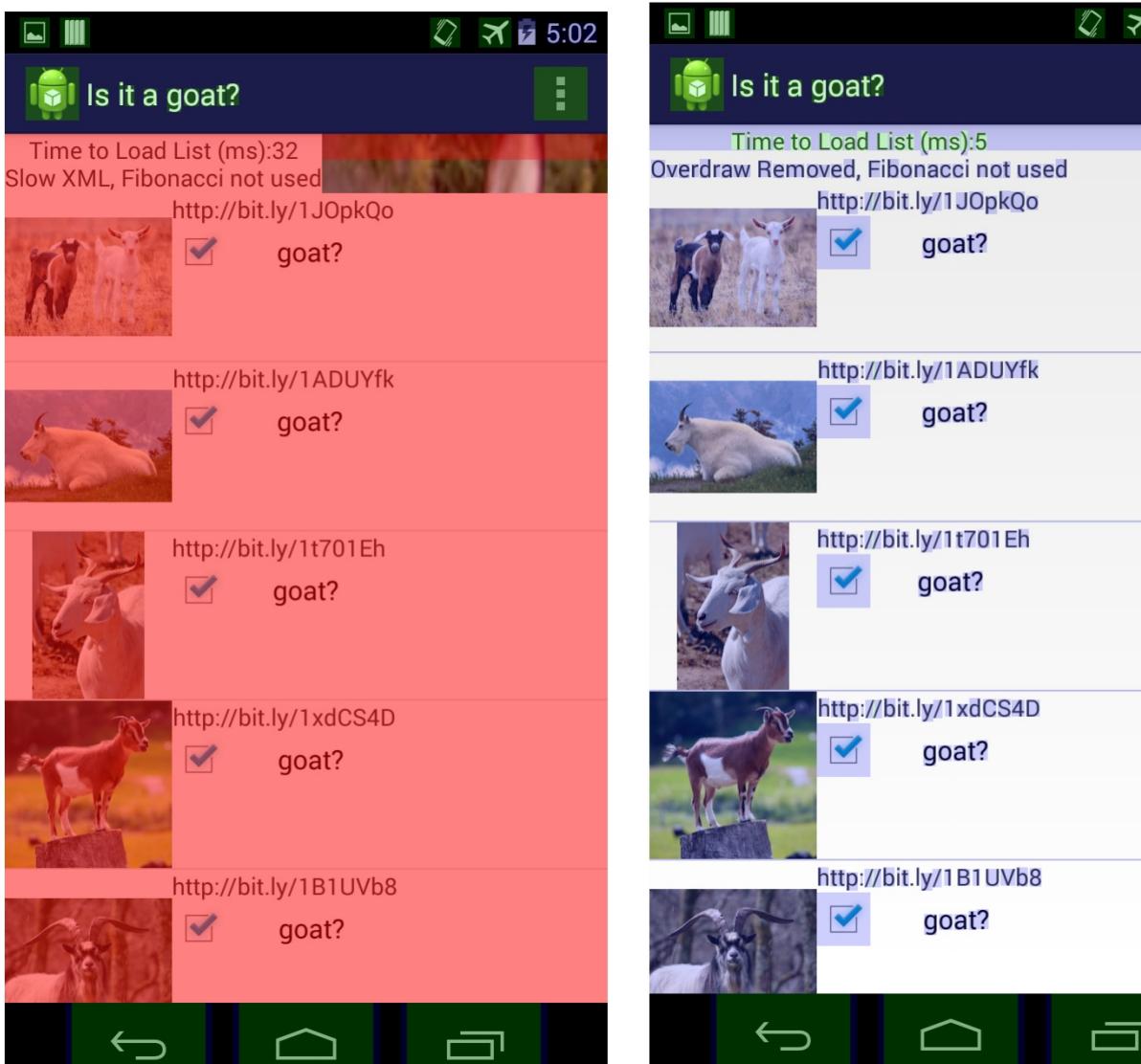


另一种查看overdraw的方式是在Debug GPU overdraw菜单里选择“Show Overdraw areas”选项。选择之后，会在app的不同区域覆盖不同的颜色来表示overdraw的次数。比较屏幕上这些不同的颜色，可以快速方便的定位overdraw问题：

白色：没有overdraw 蓝色：1x overdraw（屏幕绘制了2次） 绿色：2x overdraw 浅红色：3x overdraw 深红色：4x或者更多overdraw

在图4-15中，可以看到山羊app优化前后overdraw区域的变化。app的菜单栏优化前后都没有颜色（没有overdraw），但android图标和菜单按钮图标都是绿色的（2x overdraw）。山羊图片等列表在优化之前是深红色的（4x以上的overdraw）。优化app之后，只有checkbox和图片区域是蓝色（1x）的了，说明至少3层overdraw被消灭掉了！text和空白区域都没有overdraw了。

图4-15



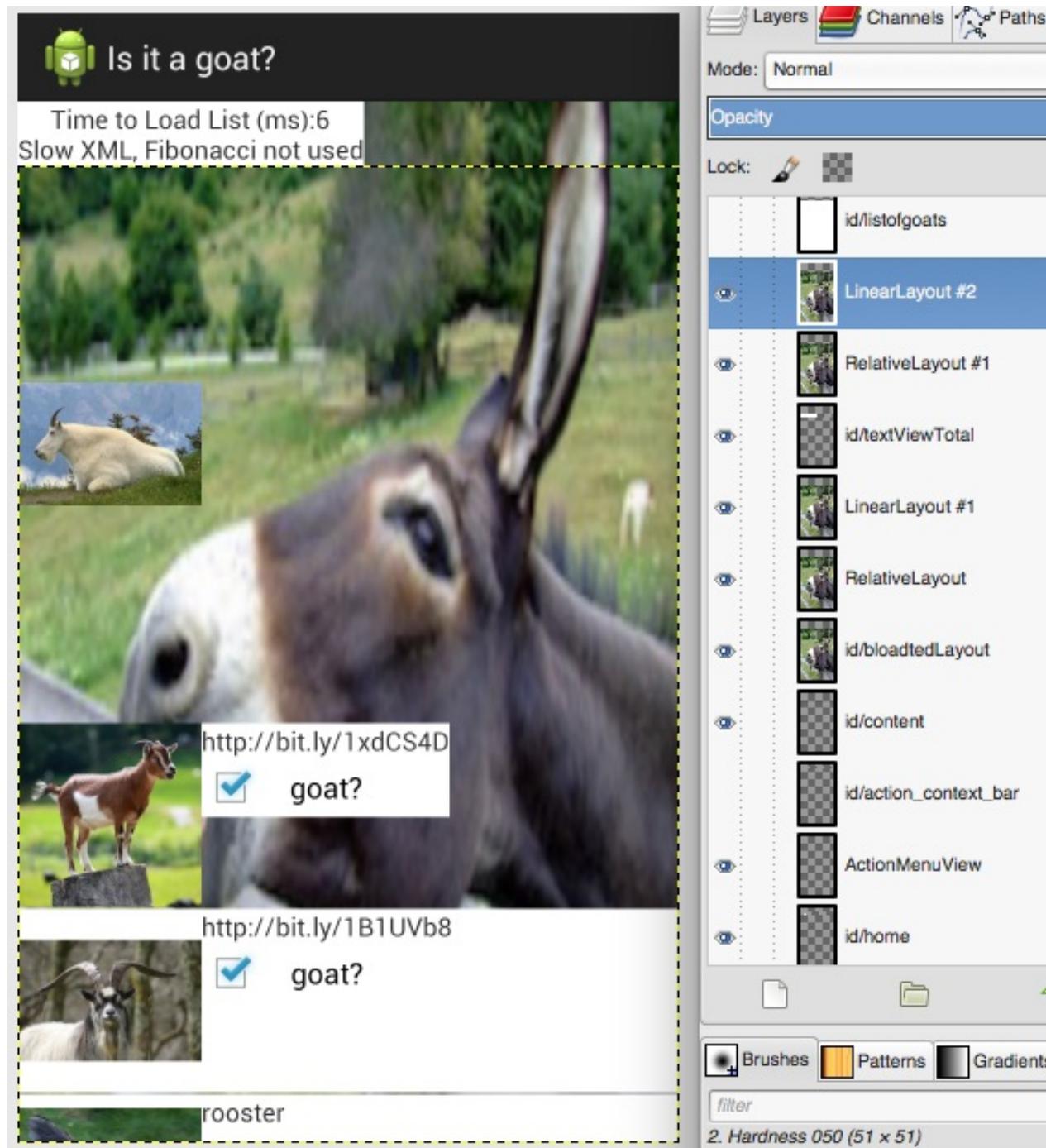
通过减少view的数量（或者去移除重复绘制的view），app的渲染会更快。通过比较父view在优化前后的绘制时间，可以发现优化后带来50%性能的提升，由13.5ms降到6.8ms。

## Hierarchy Viewer当中的overdraw

另一种查看app当中overdraw的方式是把Hierarchy Viewer中的view的树形结构保存成photoshop识别的文档（树形view里的第二个选项）。如果你没有安装photoshop，有几个其他的免费软件也可以打开这个文档。打开文档查看view，可以清楚看到不同layer里的overdraw。对于大部分的线上app，在一个白色背景上放上另一个白色背景很常见。听起来还好，但这里其实有一次绘制是多余的，完全可以避免的。我们再看下山羊app，所有overdraw图片区域都放在了一张驴子的背景图片上（替换了之前的白色背景）。之前的驴子看不到，是因为被白色背景图挡住了。移除掉之后就可以看到下面的驴子了，这样我们就可以快速的定位哪里出现了overdraw。用GIMP打开文档之后，app里所有可见的view

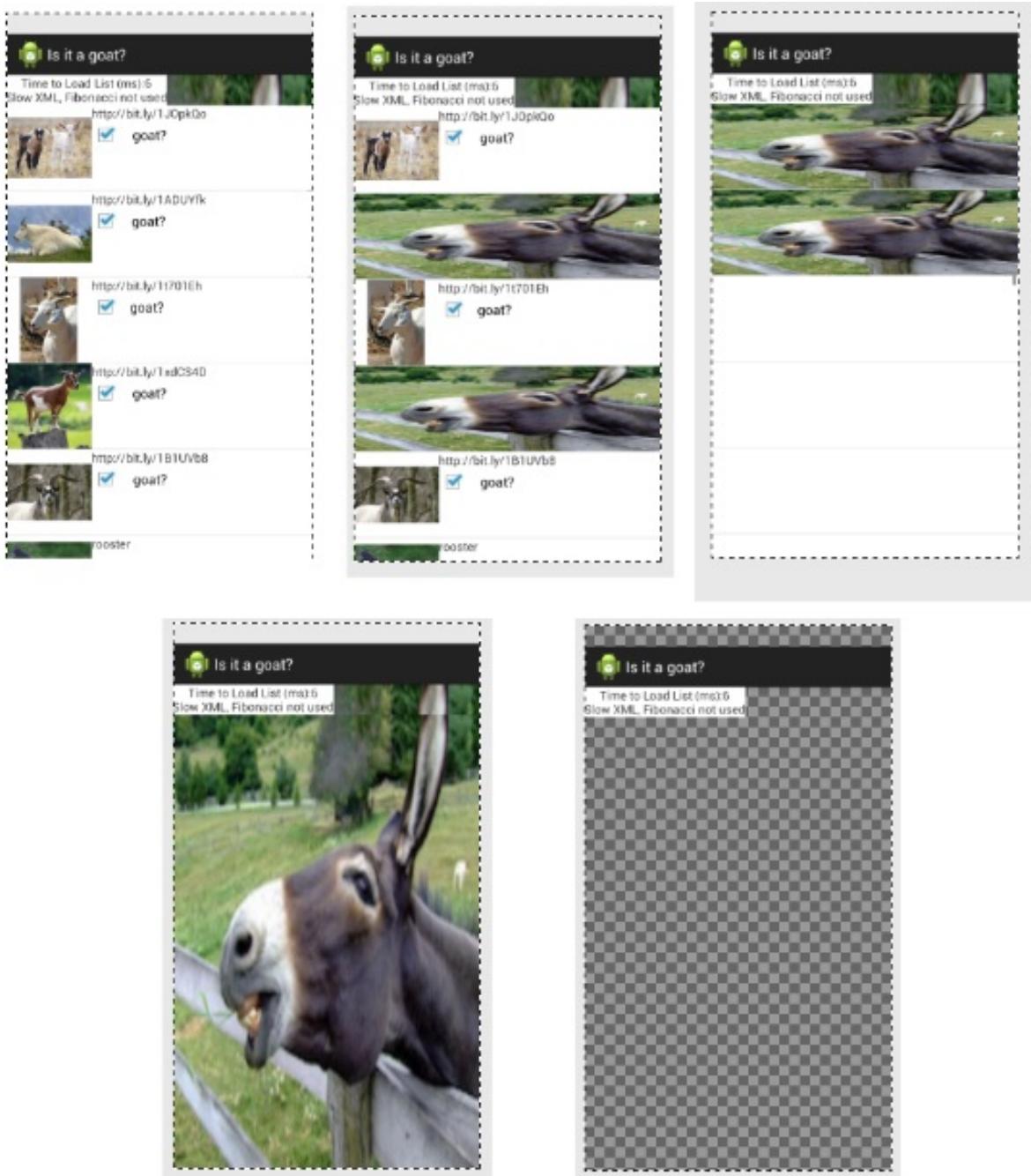
的左边都有一个小眼睛图标。在图4-16中，可以看到我从最上面开始把view一个个隐藏起来了。在右边的layout视图中，可以看到一些其他的全屏layout（都显示了驴子的图片）。

图4-16



在图4-17中可以看到另一个逐步隐藏view的办法。从最左边的全屏图片开始，到中间的图片，可以看到我们隐藏了两行山羊的图片展示，每一行下面的出现了一张拉伸的驴子的图片。在这些驴子图片的下面是一张白色的背景图（从最右边的图片可以看出）。再移除这张白色背景可以看到一张大的驴子的图片，在左下角。再往下是另一张白色的全屏背景图。

图4-17



## KitKat里的overdraw

在KitKat或者更新的设备里，overdraw被大幅度的削减了。这项技术叫overdraw avoidance，系统可以检测发现简单的overdraw场景（比如一个view完全盖住了另一个view），然后自动移除额外的绘制，应用到上面的例子，也就是说驴子那张大背景图就不会去绘制了。这很明显会极大的提高设备的绘制性能。但开发者还是要尽可能的避免额外的overdraw（为了更好的性能，也为了能兼容Jelly Bean及更老的设备）。

## Overdraw Avoidance和相关开发者工具

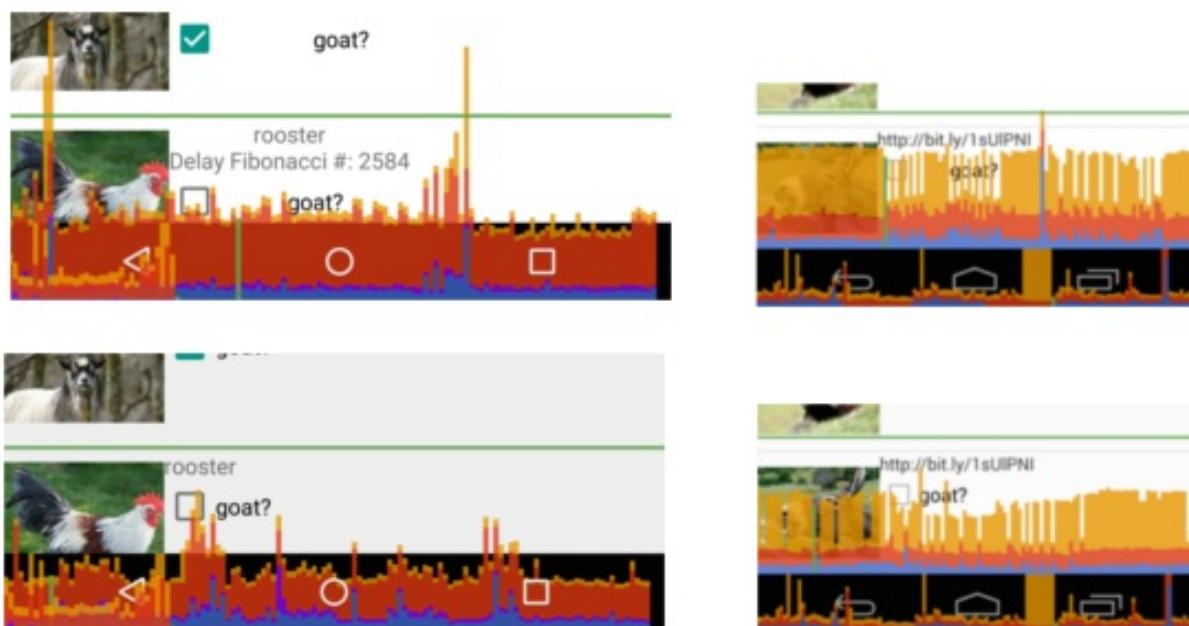
当用上面提到的overdraw检测工具时，KitKat的overdraw avoidance功能会被禁止，这只是为了方便你查看view的布局，和在设备上真正运行的情况并不一样。

## 分析卡顿（测量GPU的渲染性能）

在我们优化过view的树形结构和overdraw之后，你可能还是感觉自己的app有卡顿和丢帧，或者滑动慢：卡顿还是存在。可能高端机器上感觉不到卡顿，但低端机上还是可能会出现卡顿。为了能获取更全面的卡顿检测信息，android在Jelly Bean及更新的系统里加入了一个GPU绘制开发者选项。能够测出每一帧的绘制用了多少时间。你可以把测量出来的数据保存到一个logfile（adb shell dumpsys gfxinfo），或者在设备的屏幕上实时查看这些信息（只支持android 4.2+）。

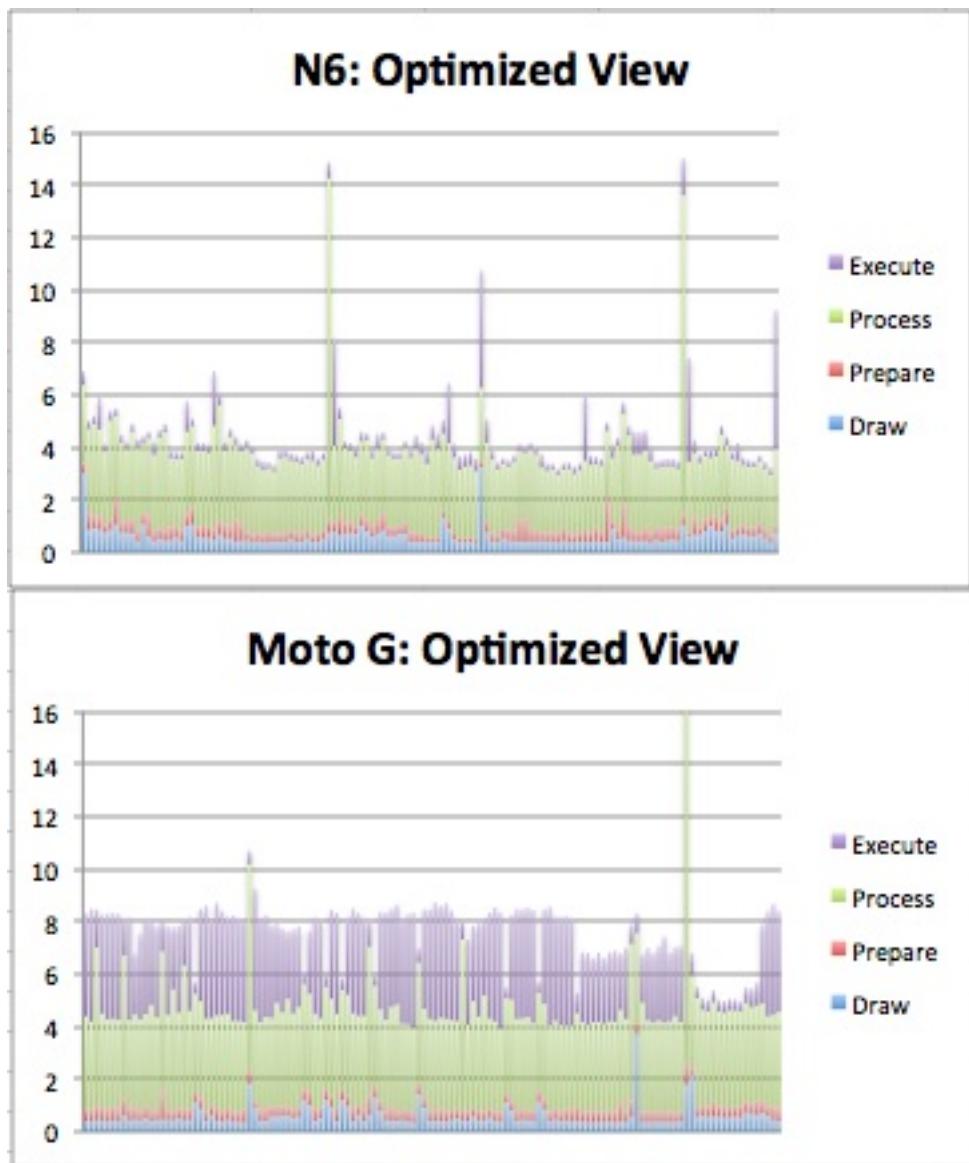
我们快速来看下怎么分析，我比较喜欢在屏幕上直接展示GPU的渲染数据，这样感觉更直观全面（logfile里面的数据很适合离线的详细分析）。我们最好在不同的设备上都试一下。图4-18展示的是Nexus 6运行Lollipop（左边）和Moto G运行KitKat（右边）同时跑山羊app的GPU渲染数据。重点看下GPU测量图表底部的水平绿条。它是设备16ms绘制一帧的分割线，如果你有很多帧都超过了这条绿线，那就表示有卡顿了。在下图里可以看到Nexus6上有偶尔的卡顿。出现在滑动到页面底部的时候，播放里一个反弹的动画。用户体验不算太糟。每一次屏幕绘制（竖线）被分成四种颜色来表示额外的测量数据：**draw**（蓝色），**prepare**（紫色），**process**（红色），**执行**（黄色）。在KitKat和更早的版本里，**prepare**的数据没有独立出来，包含在其他项里面（因此只有看到3种颜色）。

图4-18



对比下Nexus 6和Moto G的GPU数据可以看出真机测试的重要性。图4-18中，没有优化过的山羊app精确的表示Moto G绘制的时间是Nexus 6的两倍（比较两图中绿线的高度）。这一点可以通过数据采集（adb shell dumpsys gfxinfo）进一步说明。下一个例子当中，优化过的view绘制在Moto G上用了两倍多时间。对于两台设备来说，draw，prepare，process这几步都花了差不多的时间（少于4ms）。差别出现在execute阶段（紫色），Moto G比Nexus 6多用了差不多4ms。说明GPU渲染测试最好是在低端机器上来做，比较容易发现卡顿问题。

图4-19



一般来说，GPU Profiler可以帮助你发现问题。在山羊app里，如果我打开Fibonacci延迟（在创建view多时候进行耗时的递归计算），GPU profiler看不出任何卡顿，因为计算都发生在主线程而且完全阻止了渲染（在低端机上，可能会出现ANR消息）。

## Fibonacci算法

Fibonacci序列是这样一组数的集合：每个数字都是它前面两个数字的和。比如0, 1, 1, 2, 3, 5, 8等等。程序里一般用来表示递归，这里我用了最低效的方式来生成Fibonacci序列。

```
public class fibonacci {
 //recursive Fibonacci
 public static int fib(int n){
 if (n<=0)
 return 0;
 if (n==1)
 return 1;
 return fib(n-1) + fib(n-2);
 }
}
```

生成这些数字的计算次数呈指数级增长。这样做的目的是在渲染的时候增加CPU的压力，这样渲染事件就无法得到及时处理，出现延迟。计算n=40就把app变得很慢了（低端机上会crash）。这个例子虽然有点牵强，但我们定位卡顿是由Fibonacci产生的过程会很有意义。

## Android Marshmallow里的GPU渲染

在android marshmallow里，运行 `adb shell dumpsys gfxinfo` . 可以发现一些检测卡顿的新功能。首先，数据报告开头部分能看到每一帧渲染的信息了。

```
** Graphics info for pid 2612 [appname] **

Stats since: 1914100487809ns
Total frames rendered: 26400
Janky frames: 5125 (19.41%)
90th percentile: 20ms
95th percentile: 32ms
99th percentile: 36ms
Number Missed Vsync: 142
Number High input latency: 11
Number Slow UI thread: 2196
Number Slow bitmap uploads: 439
Number Slow draw: 3744
```

从app的启动开始，我们可以看到一共渲染了多少帧，其中多少帧的渲染时间是控制在理想值的90%以内，还能看到渲染比较慢的帧（90%，95%，99%）。最后五行列出的是没有在16ms内渲染完成的原因。注意，这里不止有卡顿的问题，帧率还收到了其他因素的

影响。

android marshmallow在gfxinfo库里增加了另一个好用的测试工具，`adb shell dumpsys gfxinfo framestats`。它能够输出每一帧里发生的某些事件耗时，格式是逗号分隔的一张大表。列名没有给出，但在Android Developer网站里有解释。为了算出渲染里每一步的耗时，我们要计算出报告里不同framestats的差异。下面是一些绘制事件：

- VSYNC-Intended\_VSYC (告诉你是否丢帧里，也就是卡顿)
- 处理输入事件的时间（一般要小于2ms）
- 动画计算（一般小于2ms）
- layout和measure
- view.draw()耗时
- Sync耗时（如果大于0.4ms，表示很多bitmap正在发送到GPU）
- GPU耗时（overdraw的时间会在这里面）
- 绘制一帧的总时间

有时候即使出现了超过16ms的绘制，但由于有vsync buffer的存在，也不会出现丢帧。对于没有额外buffer的低端设备，就可能会出现卡顿了。

## 不只是卡顿（丢帧）

有时候GPU Profile里看不到超过16ms的数据，但你从屏幕上看到明显的卡顿或跳动。出现这种情况可能是由于CPU在做别的事情被堵住了，从而导致丢帧。在Monitor或者Android Studio中，可以查看DDMS里的logfiles。通过过滤log更容易查看app的运行情况。可以重点看下类似下图中的log。

| Level | Time               | PID   | TID   | Text                                                                                       |
|-------|--------------------|-------|-------|--------------------------------------------------------------------------------------------|
| I     | 01-29 14:39:11.421 | 10497 | 10497 | Attempted to finish an input event but the input event receiver has already been disposed. |
| I     | 01-29 14:39:11.421 | 10497 | 10497 | Skipped 193 frames! The application may be doing too much work on its main thread.         |
| I     | 01-29 14:39:24.825 | 10497 | 10497 | Skipped 41 frames! The application may be doing too much work on its main thread.          |

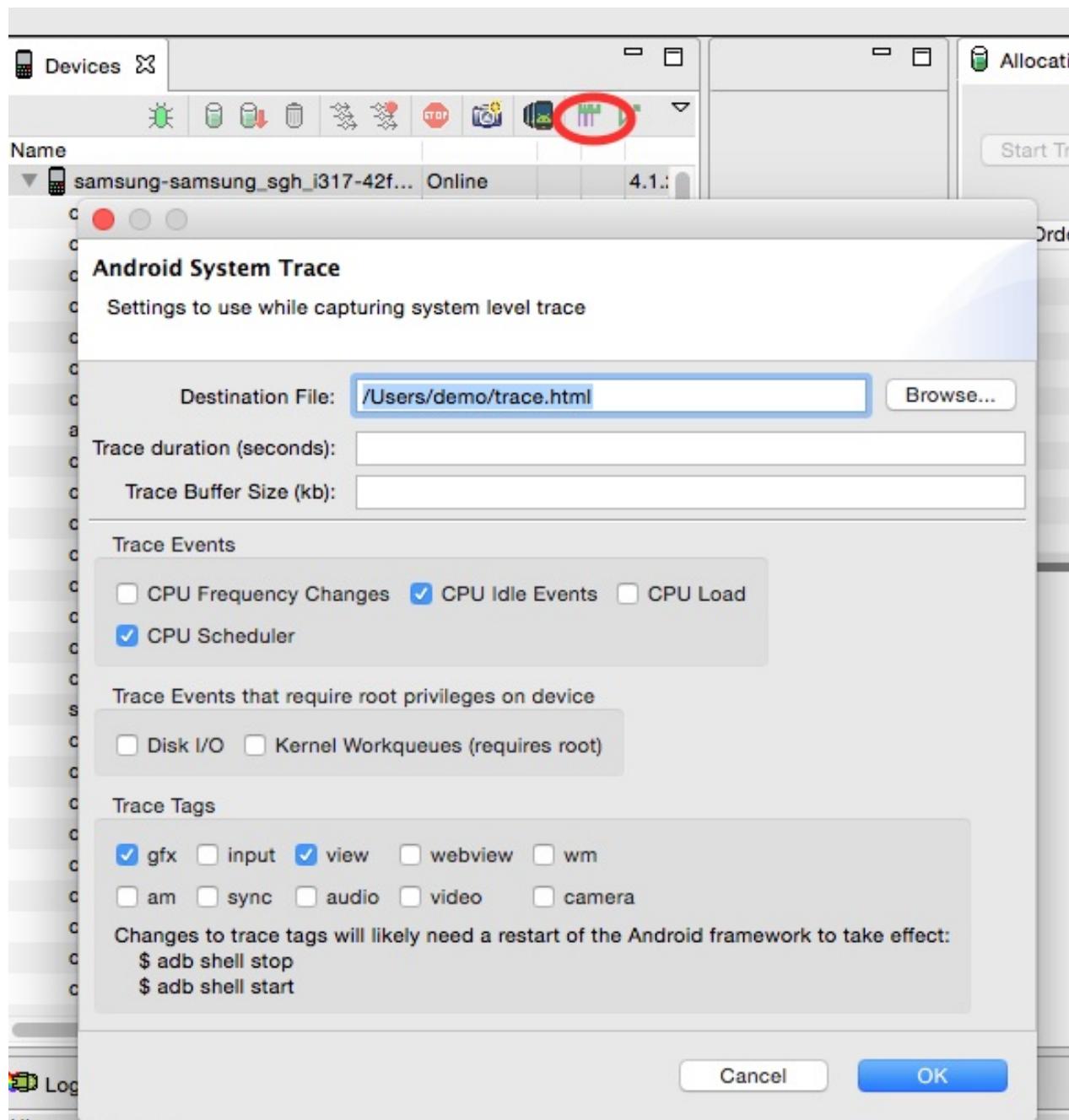
我们在后面的文章里会讲诉CPU导致的丢帧是怎么产生的。

## Systrace

在上面的这些优化之后，如果你的界面还有卡顿，我们还有办法。Systrace工具也可以测量你app的性能。甚至可以帮助你定位问题产生的位置。这个工具是作为“Project Butter”一部分同Jelly Bean一同发布的，它能够从内核级检测你设备的运行状态。Systrace可配置的参数很多。我们这里重点关注UI是怎么渲染的，用systrace检测卡顿问题。

Systrace和之前的工具不同的是，它记录的是整个android系统的状态，并不是针对某一个app的。所以最好是用运行app比较少的设备来做检测，这样就不会受到其他app的干扰了。Systrace图标是绿色和粉红色组成的（下图红色的椭圆里）。点击下，会弹出一个带几个选项的窗口。

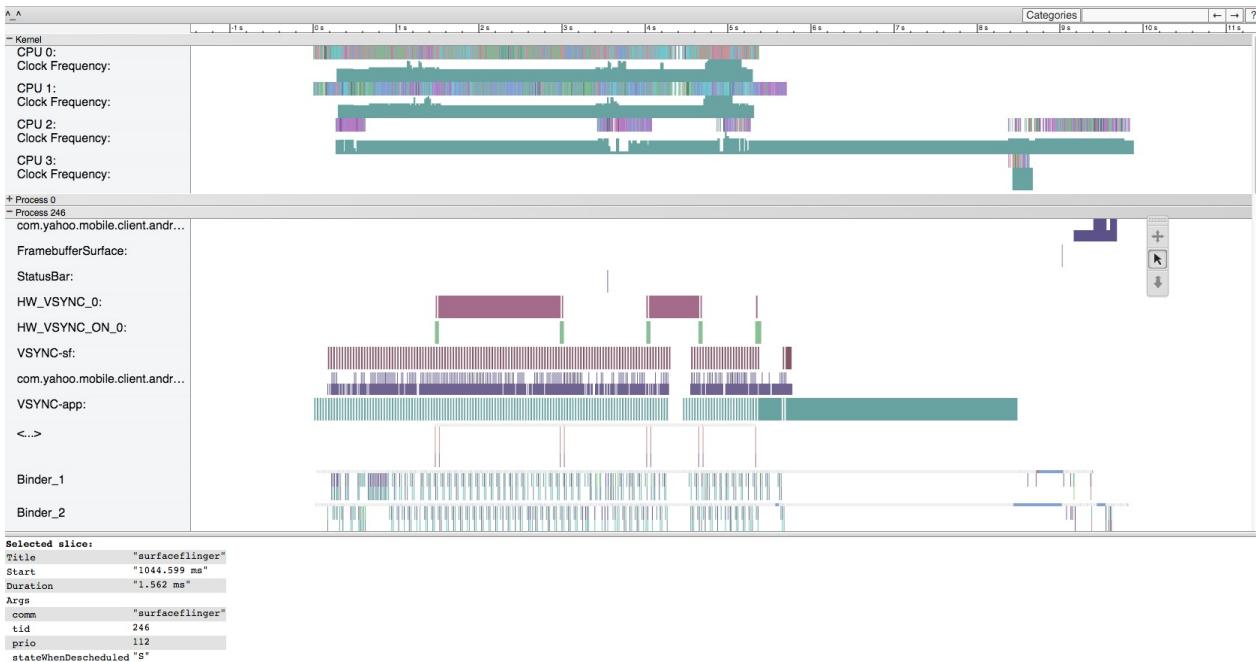
图4-22



trace数据记录在一个html文件里，可以用浏览器打开。这里主要研究屏幕的交互数据，主要收集CPU, graphics和view数据（如图4-22所示）。duration留空（默认是5秒）。点击OK之后，Systrace会马上开始采集设备上的数据（最好马上开始操作）。因为采集的数据非常之多，所以最好一次只针对一个问题。

traces里面的数据看着有点吓人（我们只是勾选里4个选项！）。鼠标可以控制滑动，WASD可以用来zoom in / out (W, S) 和左右滑动 (A, D)。在刚跑的trace数据最上面，能看到CPU的详细数据，CPU数据的下面是几个可折叠的区域，分别表示不同的活跃进程。每一个色条表示系统的一个行为，色条的长度表示该行为的耗时（放大可以看到更多细节）。选中屏幕底部的一个色条，第一眼看到的总览有点吓人，我们一条条分析看下这些数据。

图4-23



## Systrace进化史

就像android生态圈一样，Systrace在不同的系统版本里有不同的界面，展示，和输出结果。

- 在Jelly Bean设备，在设置的开发者选项里可以打开tracing。必须要同时打开电脑和手机上的该功能。
- 随着android系统版本的升级，trace生成的数据也更加详细，布局也有一些改变。
- 我建议通过Jelly Bean查看Systraces，然后喝Lollipop上的数据对比，收集到的数据会不一样。

在2015年的google io大会上，google发布了新版本的Systrace，新版本增加了一些新特性，下面会有更详细的介绍。

我们继续滑动Systrace的输出结果，运行期间每个进程的数据都可以看到。我们主要研究卡顿相关信息，查看屏幕刷新时可能有问题的绘制。只要刷新率和绘制都正常，屏幕的渲染应该就是流畅的。但只要一个出问题，就有可能会导致页面渲染的卡顿。

## Systrace Screen Painting

我们通过图4-24来看下屏幕绘制的步骤。最顶部一行的trace（蓝色高亮）时VSYNC，由一些均匀分布的蓝绿色宽条组成。VSYNC是操作系统发来的信号，表示此时该刷新屏幕了。每个宽条表示16ms（宽条之间的空白也是16ms）。当VSYNC事件发生的时候（在蓝绿色宽条的任意一侧），surface flinger（红色高亮方框包含几种颜色的长条）会从view buffer（没展示出来）里选一个view，然后绘制到屏幕上。理想情况下，surfaceflinger事件之间相距16ms（没有卡顿），因此如果出现长条空缺则表示surfaceflinger丢掉了一次VSYNC更新事件，屏幕就没有及时的刷新（此时就会有卡顿）。在trace文件2/3的位置可以看到这样的空缺（绿色高亮方框）。

图4-24



图4-24底部展示的是app的详情。第二行数据（绿色和紫色的线条）表示的app正在创建view，然后是底部的数据（绿色，蓝色，和一些紫色的条状），表示的是RenderThread，view的渲染和发送到buffer（图中没有画出来）都是在这个线程里做的。

注意看可以发现大概1/3的位置，这些条状在该区域集中变粗了，表示app此时由于某种原因发生了卡顿。不同app情况不一样，发生卡顿的原因也不同，但是我们可以根据一些共同的现象推测卡顿的发生。

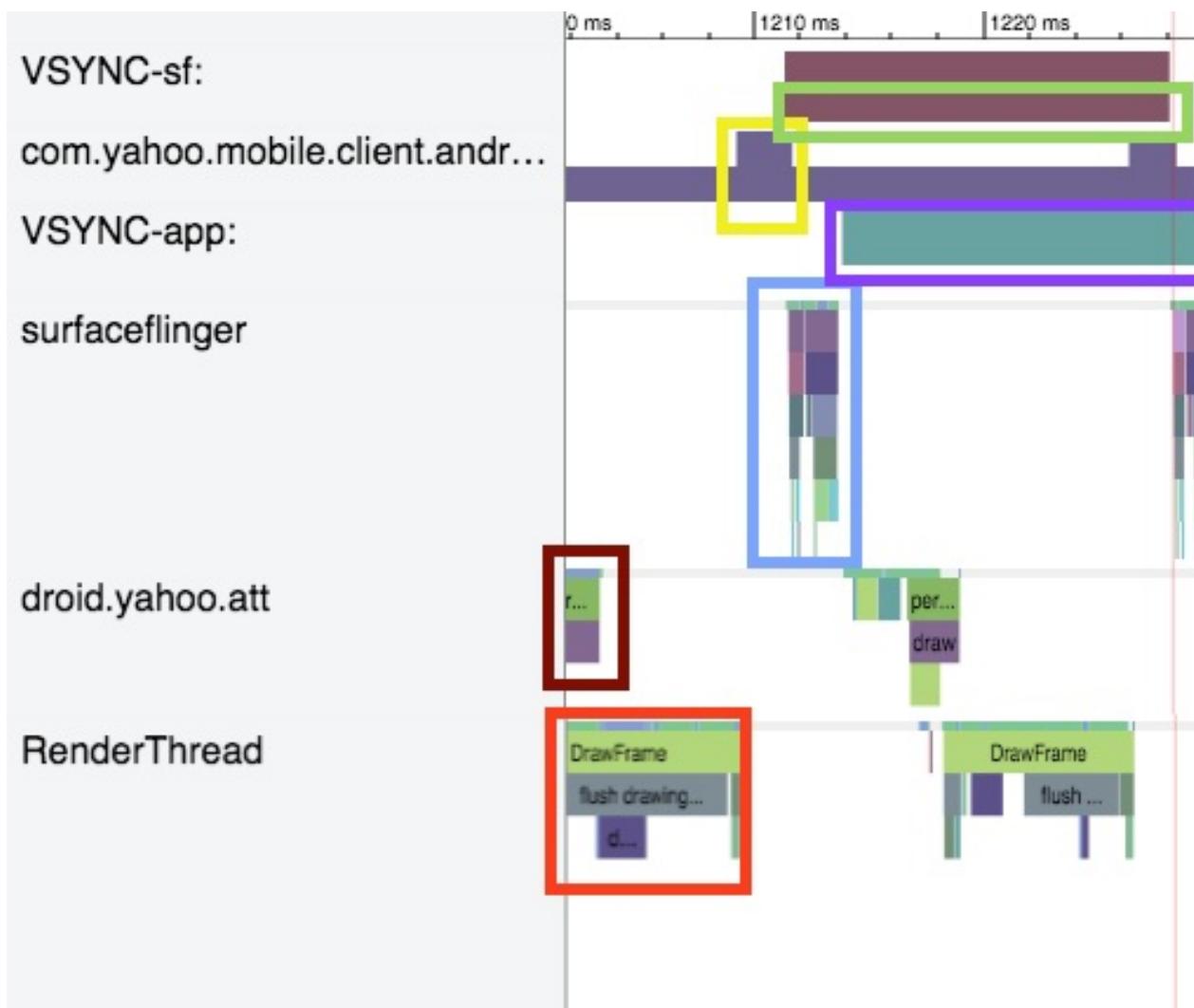
这种总览很适合查找卡顿，但要调查清楚原因需要放大仔细看下。要明白Systrace都记录了什么数据，最好搞明白Systrace到底是怎么进行测量的，app没有卡顿的时候Systrace输出又是什么样的。一旦弄明白了Systrace是怎么工作的，查找问题就方便多了。在图4-25中，我把app正常运行时Systrace纪录的相关线条放到了一起。我们从屏幕左边的droid.yahoo.com看起。我描述的时候在trace文件里会来回跳动到不同的位置。当绘制发生的时候：

- 红色方框：droid.yahoo.com完成了所有view的measure，然后把结果发送给RenderThread。
- 橙色方框：RenderThread，这里app会：
- 绘制frame（浅绿色）
- 显示buffer里的内容（灰色）
- 清空buffer（紫色）
- 发送给缓存的view列表。
- 黄色方框：com.yahoo.mobile.client.andr...

buffer里面有一些view，线条的高度表示了buffer当中view的数量。刚开始，只有一个，当新的view加入到buffer中之后，高度就变成了2倍。

- 绿色方框：VSYNC – sf 提示surface flinger有16ms的时间来渲染屏幕。里面棕色的条状表示16ms的长度。
- 蓝色方框：surfaceflinger从队列里抓取一个view（注意黄色方框里的buffer中view数量从2变为1）。完成之后，view被发送给GPU，屏幕就绘制被绘制了。
- 紫色方框：VSYNC – app告诉app去渲染新的view（这里有个16ms的timer）。
- 当VSYNC一开始，droid.yahoo.att就不停的重复这个过程，measure view,发送给RenderThread等等，不停的循环。

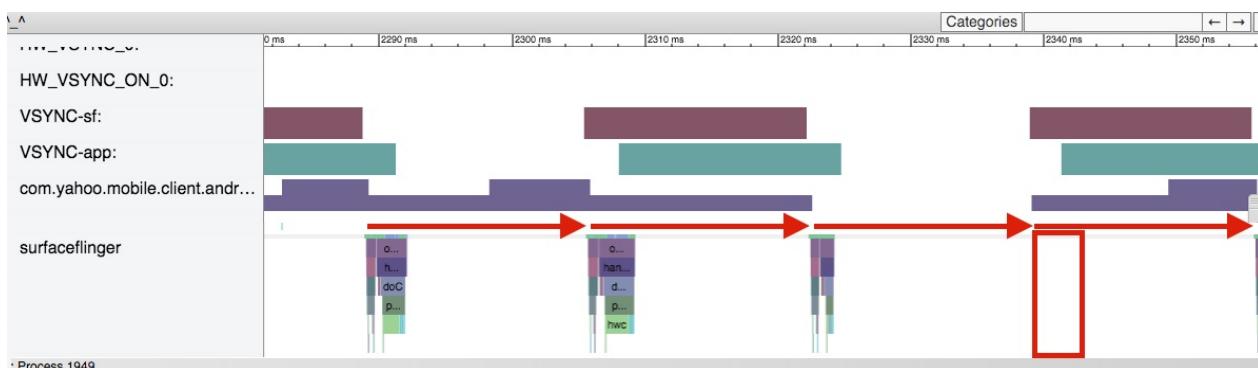
图4-25



再回过头想一下设备能这么短的时间内流畅的渲染屏幕，确实是件很神奇的事情。了解了渲染的过程，我们来找下卡顿的原因。

图4-26中，我们看下OS层的行为。我增加了一些箭头来表示16ms的间隔，红色的方框表示surfaceflinger的丢帧。

图4-26

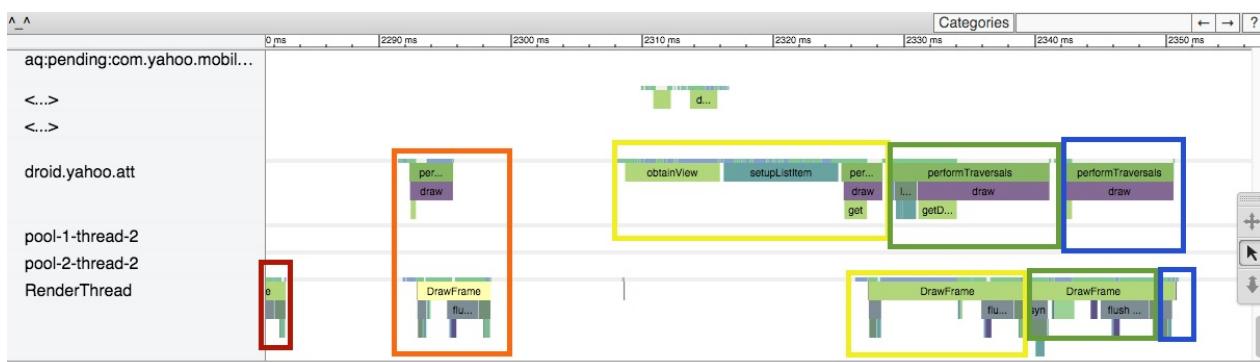


为什么会出现这种情况？箭头上方的一行是view buffer，行的高度表示有多少帧缓存在了buffer里面。trace开始的时候，buffer里缓存的数量是1到2交替出现。surfaceflinger每抓取一个view（buffer里的数量减一），又会马上从app里生成一个新的view来填充。但是当

surfaceflinger完成第三个动作之后，buffer被清空了，但是没有从app里及时填充新的view。所以，我们从app层面来检查下这期间发生了什么。

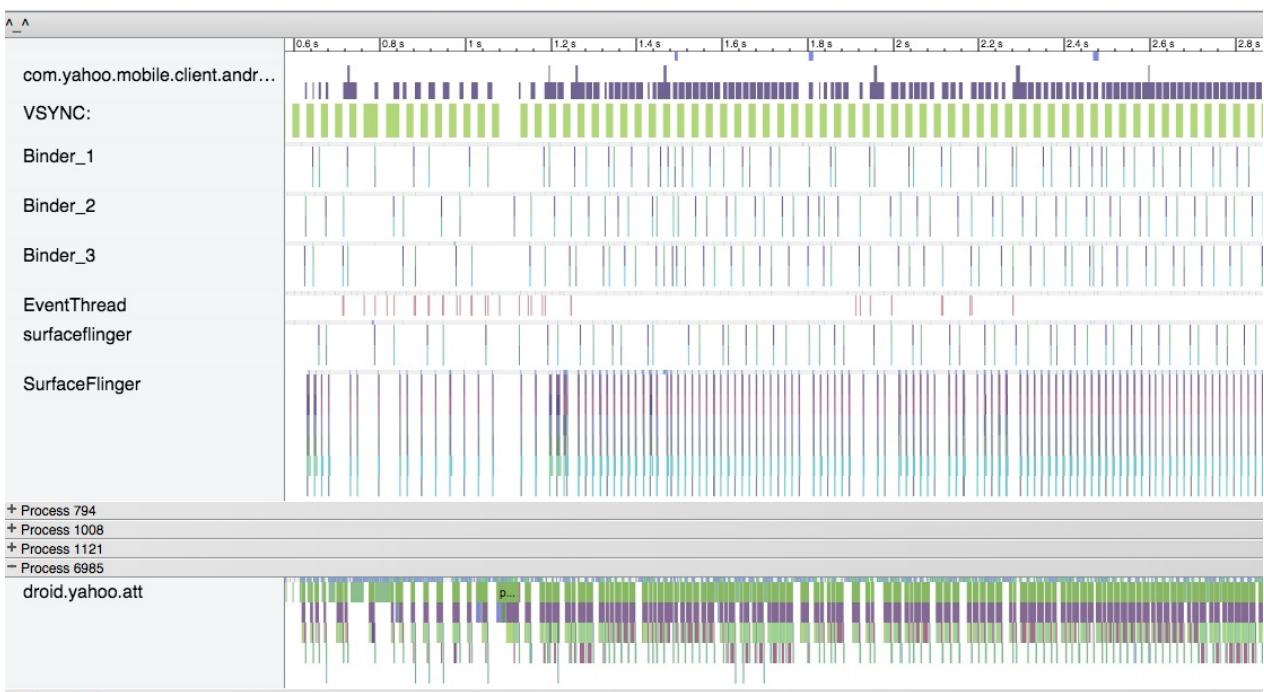
在图4-27中，我们可以看到开始的时候RenderThread发送了一个view到buffer（红色方框）。橘色方框表示app新建了另一个view，渲染，然后交给buffer（droid.yahoo.att measure,layout所有的view，RenderThread负责绘制）。不幸的是，app没来得及创建新view就被挂起了（黄色方框内）。为了创建下一个view，droid.yahoo.att app在运行暗绿色的“performTraversals”（3ms）之前，要先运行“obtainView”7ms，“setupListItem”8.7ms。app然后把数据交给RenderThread，这一步也比较慢（12ms）。创建这一帧总共用了近31ms（上一个只用了6ms）。当创建这一帧开始的时候，buffer里只有一帧的数据，但是设备需要两帧。buffer没有被填满，所以屏幕绘制出现了卡顿。

图4-27



有意思的是app后面马上就速度追了上来。黄色方框内延迟递交的view创建并交给buffer之后，后续的两帧紧接着创建好了（绿色和蓝色的方框）。通过快速的填充新的帧，app就只丢了一帧。这个trace结果是在Nexus 6上运行的（处理器比较快，能快速的跟上）。在三星S4 Mini，Jelly Bean 4.2.2上运行同样的结果得到图4-28。

图4-28



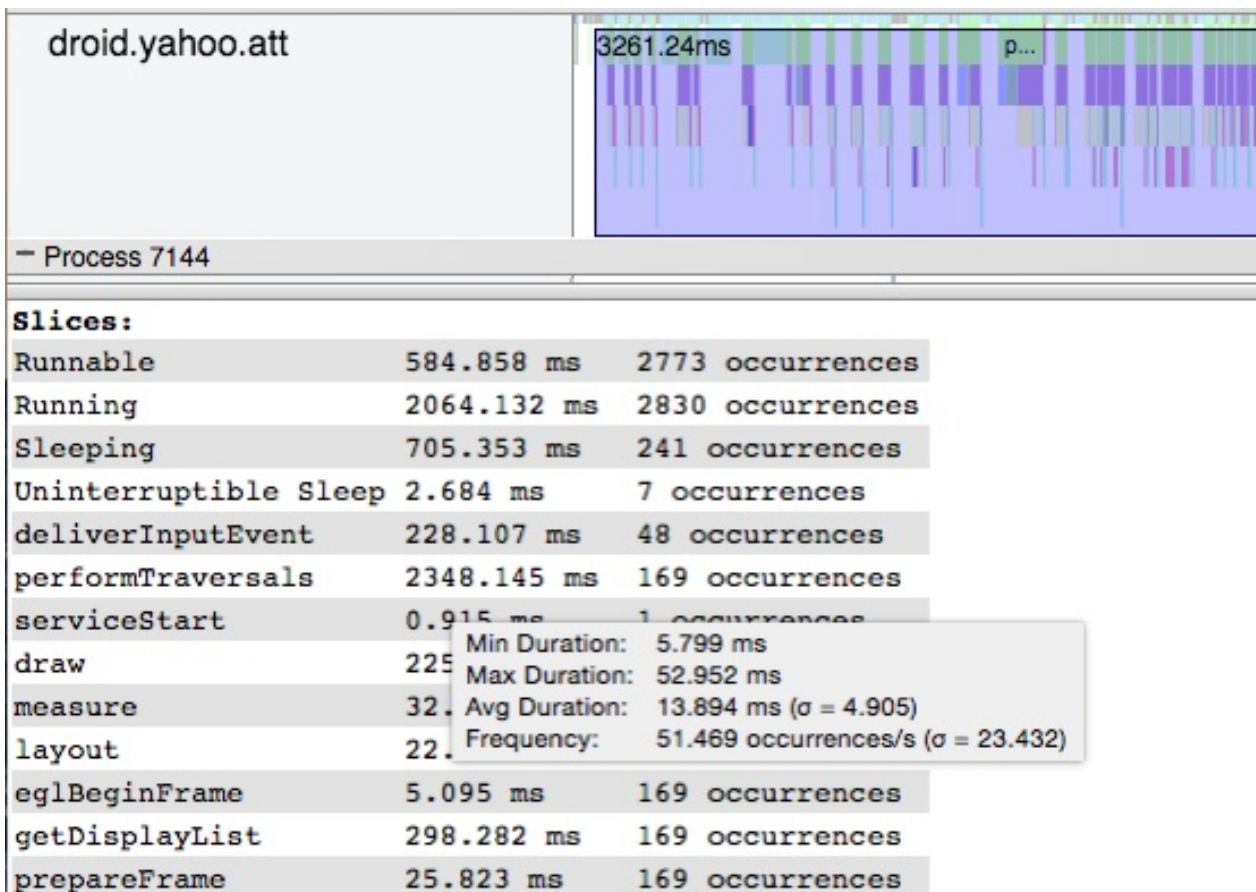
从总览图上可以清晰的看到有很多帧都丢掉了（trace开始的时候surfaceflinger部分有很多的空缺）。而且顶部那一行（view buffer）里的buffer经常是空的（导致卡顿），buffer里同时有两个view的情况非常少。对于一个GPU性能比较差的设备来说，app能够像Nexus 6一样赶上填满buffer的概率比较小。

小贴示：其实你可以偶尔渲染一帧超过16ms，因为buffer里面一般都有1到2帧准备好的view备用。但是如果超过2-3帧渲染很慢，用户就会感觉到卡顿了。

上面的trace是在运行Jelly Bean的手机上跑的，RenderThread的数据归到了droid.yahoo.att那一行（Lollipop之前measure,draw,layout都是和在一起的）。把每一行数据合在一起之后竖条变宽。每一次调用之间的细条空白说明手机在每帧的绘制之后，只剩下很少的时间处理其它任务。手机上的app只能稍稍领先surfaceflinger填满buffer的速度。如果app能够减小所绘制view的复杂度，也就是加快view的渲染，细条的空白就会变的宽一点，buffer填满的概率就更大，也就给低端设备在绘制之外更多的空间去处理其它任务。

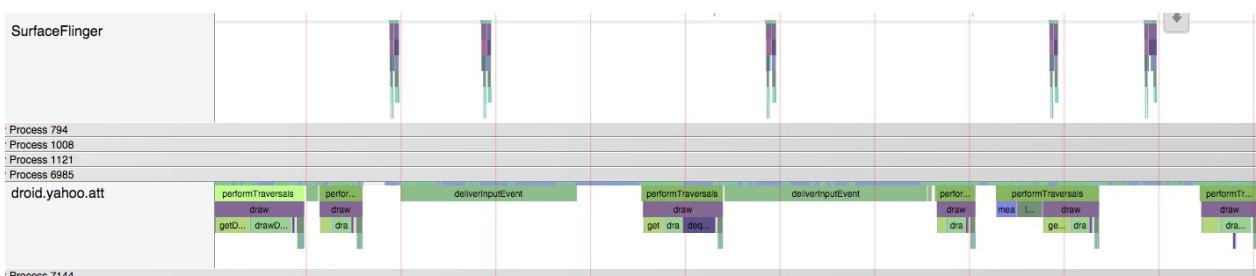
把这块区域加高亮之后，Systrace会把所有条状所占的时间计算出一个总和，用鼠标在上面依次移动就能看到基本的数据了。图4-29中，可以看到performtraversals（父view的draw命令）平均用了13.8ms，大概有5ms的波动。16ms的卡顿阈值在波动的范围之内，所以很有可能设备上会有卡顿。

图4-29



把这块放大能看到更多的细节（图4-30）。每个垂直的红线表示16ms。从图中可以看出，大概有5, 6次SurfaceFlinger错过了红线标记。绿色的“performtraversals”线条都几乎有16ms长（这一步是必须做的，有卡顿）。还有两个蓝绿色的 deliverInputEvents（每个都超过了16ms）也阻碍了app的屏幕绘制。

图4-30



那到底是什么触发了deliverInputEvents呢？这其实是用户在点击屏幕，强制ListView重绘所有的view。这部分影响是CPU，我们接下来简单看下这时候CPU都在干嘛。

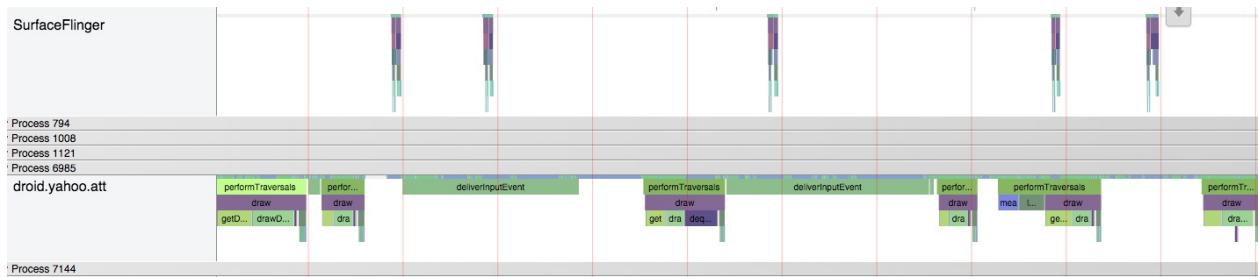
## Systrace和CPU对渲染的影响

如果你频繁的感觉到卡顿，但是在绘制或者surfaceflinger部分看不到什么明显的异常，这时候可以尝试看下CPU在处理什么事情，在Systrace的顶部可以看到这部分的数据。如果你能大概猜到是哪部分的逻辑影响了绘制，可以把这部分代码注释掉试试。山羊app里

有个选项可以开启Fibonacci延迟。打开之后，app在每一行数据渲染的时候都会计算一个很大的Fibonacci值。用膝盖想都知道这时CPU会变得很忙。由于计算是在主线程做的，会妨碍的view的渲染，理所当然就导致里丢帧，滑动也会变的很卡。图4-21里显示的log就能看到这种情况下的丢帧。我们再深挖一点看能不能通过Systrace定位到计算Fibonacci数的代码。

我们再重头看下trace数据，图4-31里是没有优化过的山羊app在Nexus 6上跑的数据。

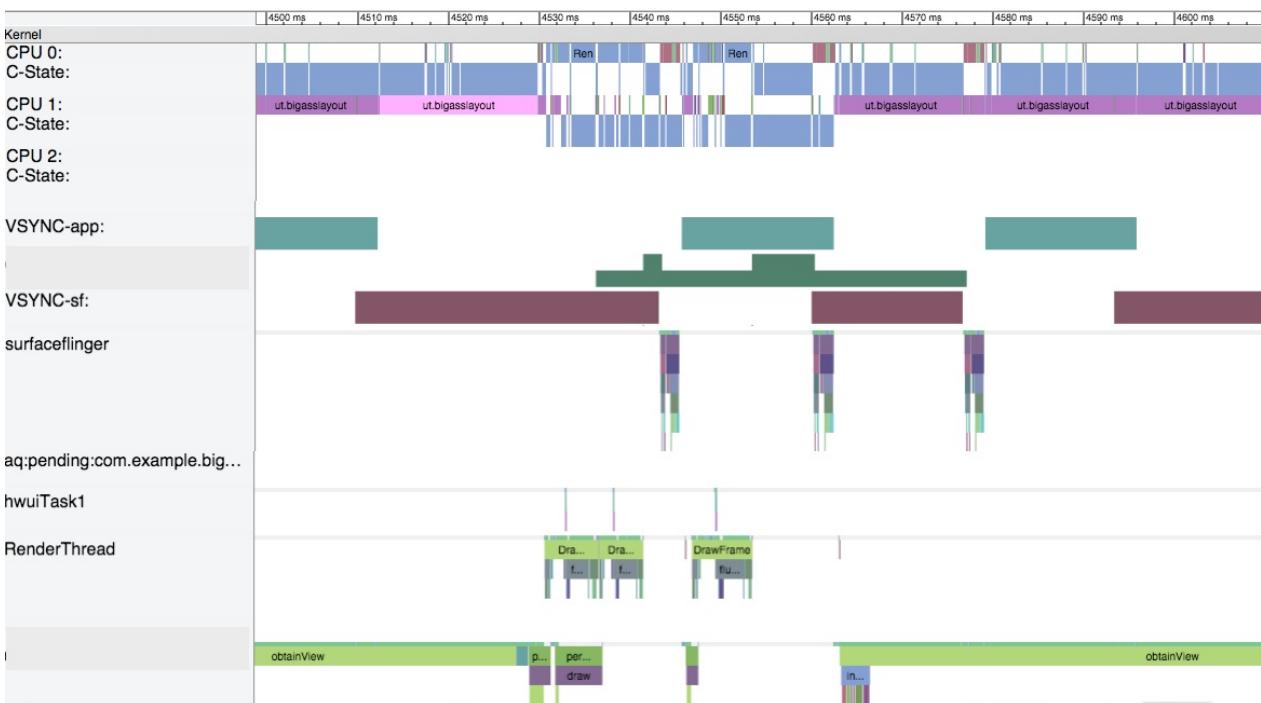
图4-31



展示做了一些修改，CPU和surfaceflinger之间的一些线被去掉了。这个trace里看不到卡顿，surfaceflingers每16ms的间隔很均匀。RenderThread和每一行view填满buffer的表现也很正常。和CPU那一行数据对比一下，可以发现一个新规律。当RenderThread在绘制layout的时候，CPU1正在运行一个蓝色的任务(注意我们看的是窄一点的CPU1，不是CPU1:C-State)。当山羊app的view正在被measure的时候，CPU0有一个相应的紫色的行为。view的layout和绘制是由两个CPU完成的。注意X轴上的点击是每隔10ms发生的，这里每个行为都没有超过2-4ms。

当我们加入费时的Fibonacci计算之后，Systrace的结果看起来就很不一样了。（图4-32）

图4-32

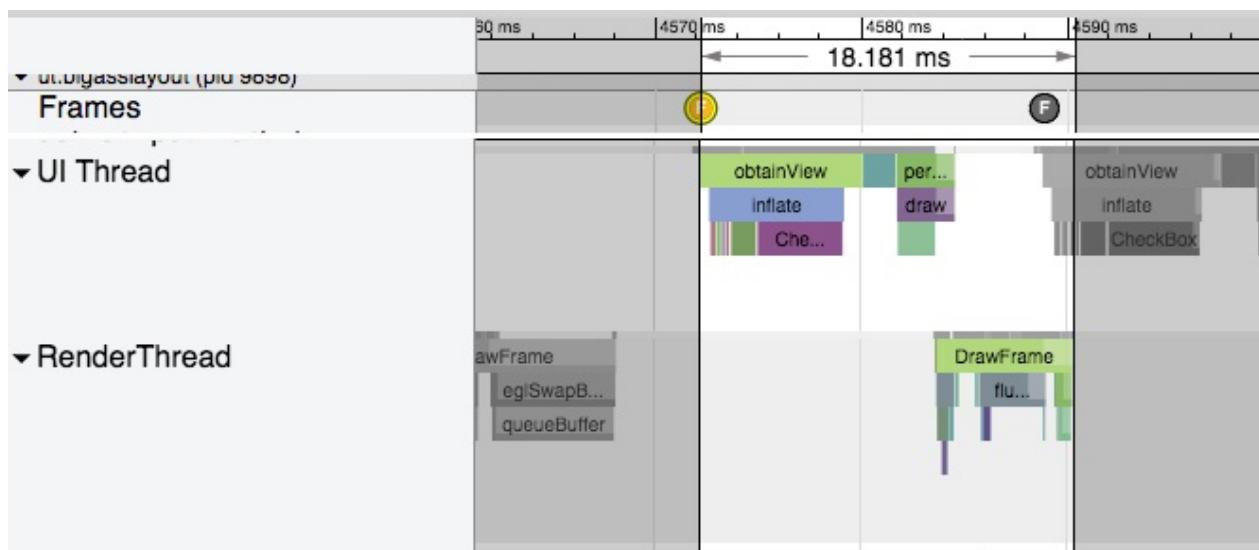


从Systrace里能看到很多卡顿，在相同的100ms时间范围内，surfaceflinger就画了三帧（上面不卡顿的情况画了7帧）。可以看到RenderThread绘制view还是很快的（从图中可以看出，蓝色的RenderThread是在CPU0上运行的）。但是，measure view的时候，Fibonacci的递归计算就导致了问题。山羊app进程那一行花了大部分的时间在obtainView的状态，而不是measure。同时可以看到CPU1上紫色对应的山羊进程不再是2-4ms宽了，变成了2-17ms宽。Fibonacci计算每次大概用了13-17ms，对app的绘制性能产生了很大的影响。

## Systrace更新 – I / O 2015

在2015年Google I / O大会上，google发布了新版本的systrace，上面提到的分析数据变的更简单了。在图4-27里，我把每一帧的更新都高亮出来了。在新版本的systrace（图4-33）里，每一帧都是由一个带F的小圆图标示的。正常渲染的帧会有绿点，慢帧则是黄色或者红色。选择一个点，然后按下m就可以高亮某一帧，分析起来更方便。

图4-33



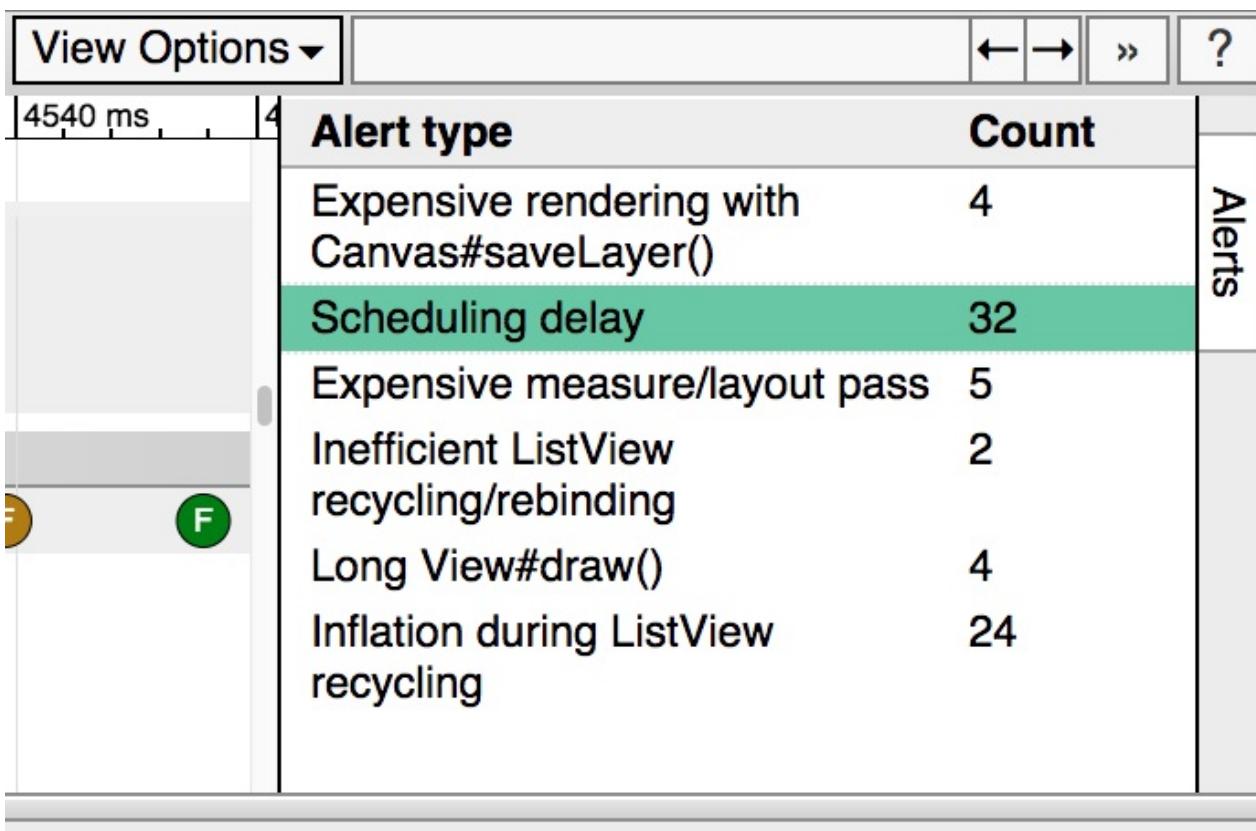
新版本的systrace对于正在发生的行为也有更清晰的描述了。在图4-33中，帧的渲染时间是18.181ms，是用黄色标示的，如果有很多帧超过了16ms就会导致卡顿了。在trace文件下方的描述信息面板上（图4-34），可以看到警告信息，说我的app在重用ListView的item，而不是创建新的item，这样拖慢了view inflation。

图4-34

| 1 item selected: Frame (1) |                                                                                                                                                |
|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| Alert                      | Inflation during ListView recycling                                                                                                            |
| Time spent                 | 9.339 ms                                                                                                                                       |
| ListView items inflated    | 1                                                                                                                                              |
| <u>obtainView</u>          | took 7.96ms                                                                                                                                    |
| <u>setupListItem</u>       | took 1.57ms                                                                                                                                    |
| <u>Frame</u>               |                                                                                                                                                |
| Description                | ListView item recycling involved inflating views. Ensure your Adapter#getView() recycles the incoming View, instead of constructing a new one. |

在systrace里可以看到其它类似的警告，形状像泡泡或是点，屏幕右边的警告面板也列出了这些信息（图4-35）。

图4-35



The screenshot shows the Systrace interface with the 'Alerts' tab selected. A table lists various UI performance alerts with their counts:

| Alert type                                  | Count |
|---------------------------------------------|-------|
| Expensive rendering with Canvas#saveLayer() | 4     |
| Scheduling delay                            | 32    |
| Expensive measure/layout pass               | 5     |
| Inefficient ListView recycling/rebinding    | 2     |
| Long View#draw()                            | 4     |
| Inflation during ListView recycling         | 24    |

这些新功能让Systrace诊断UI问题更加简单了。

## 第三方工具

每个大的芯片厂商都有自己的GPU评测工具，可以帮助发现更多渲染时遇到瓶颈的信息。这些工具对一些特定的芯片更有针对性，信息也更多。可以帮你针对不同的GPU做更深度的优化。Qualcomm, NVIDIA和Intel都提供了这些开发者工具，有兴趣的可以自己试下。

## 感知优化

上面的内容都是在讨论怎么通过测试，调试，优化布局来让UI的体验更快。其实还有另外一个办法让你的app UI更快：让用户感觉更快。当然作为开发者要尽可能优化自己的代码，view，overdraw和其它所有可能会影响渲染性能的地方，上面这些都做了之后，再考虑下面这些能让用户觉得你的app更快的方法。

人类大脑工作的方式很有意思，通过改变大脑对等待的感知，可以让你的用户感觉延迟变短了。杂货店的老板都会在走廊上放一些没用的杂志，就是为了让客户有东西可以看，感觉等待的时间就会短一些。如果在向用户展示内容的时候增加一些过渡效果，见效明显。

这就像一个小魔术一样让用户感觉体验变的更快了，归根结底重要的是用户觉得你的app有多快。这个技巧实现起来也有点取巧，有时候这种感知的优化甚至会得到相反的效果，做A / B test来确保你的优化对用户来说真的有效。

## loading菊花：优缺点

loading菊花，进度条，沙漏图标，和其它所有表示等待的方式都存在很久了。这些都可以让app的内容过渡变得更快。比如在app里加一个进度条，加载的时候播放一个进度的动画来让用户等待。研究表明使用一个带有动画的滑动条的时候用户会感觉更舒服。快速旋转的loading菊花也让用户感觉等待的时间更短。

但是，有延迟的时候，加个菊花并不总是有效的。iOS app Polar的开发者发现他们的app渲染一个view的时候有一点延迟。他们第一反应是在页面里加了一个菊花告诉用户页面正在渲染内容，但效果不如预期。用户开始反馈app变慢了，等待页面加载的时间变长了

（其实app没有变慢，不过是加了一个菊花）。加了个等待的标识之后让用户明显的感觉到他们在等。取消菊花之后，用户感觉app又变快了（开发者仅仅是改变了菊花）。通过改变用户对等待的感知，可以让用户觉得app变快了。Facebook也遇到过类似的问题：使用自己定制的菊花让用户感觉更慢，用默认菊花感觉更快。

增加菊花最好让用户测试下他们的真实感受。一般来说，当等待的时间稍微有点长的时候，增加菊花是可以接受的：比如打开一个新页面或者从网上下载一张图片。如果延迟很短（一般来说小于一秒），就应该考虑去掉菊花了。这种情况下应该让用户觉得他们并没有在等。

## 用动画来抵消等待的时间

点击后看到一个空白的屏幕会让用户感觉在等待。就是这个原因让浏览器在点击链接，新页面刷出来之前都是展示旧的页面。在手机app里，一般来说我们不希望让用户停留在老的页面上，一个快速的切换动画可以争取到足够的时间让下一个页面准备就绪。可以观察下你最常用的android app，当页面切换的时候有多少从边上或者底部出现的动画。

## 瞬时更新的小谎言

如果你的用户在页面上做了更新数据的操作，即使数据还没抵达服务器，可以马上把用户看到的数据更新掉（当然开发者要保证这些数据能100%抵达服务器）。比如说，你在Instagram上点了赞，页面上马上就更新了赞的状态，其实赞的状态甚至可能还没有更新到服务器。Instagram的开发者管这叫“行为最优化”，状态的更新要几秒后才能到服务器并对网站的用户可见（网速不好的时候可能要几分钟），但是更新最后都会成功，等待服务器返回成功其实是没必要的。移动端用户一般都不希望在等待，只要最后能成功就好。

瞬时更新的另一个好处是，用户会感觉你的app在网速或者信号不好（火车经过隧道）的时候也能正常工作。FlipBoard就做过一个离线发送网络请求的框架，可以很方便的应用到更新UI。

另一个优化的小技巧是提前上传。对于像Instagram这种app来说，上传大量的图片会增加主线程的延迟，提前开始上传这些图片会是个好办法。Instagram发现发一个新post是慢在上传图片这一步，所以Instagram就在用户在图片上添加文字的间隙开始上传图片了，图片被真正发布到服务器之前就已经传好了。用户只要一点击Post按钮，就只需要上传文本和创建post的命令了，这样就会让用户感知非常快。Instagram在遇到“是否要添加菊花”这个问题时，他们的答案是通过改变架构的方式永远的杜绝菊花。

## 提升感知体验的小提示

当app的速度通过优化代码或者view的优化提升之后，你可以用秒表来测试下结果。有些感知是可以用秒表测量的（Instagram的例子），有些则不能（菊花的例子）。当常规的分析或者测量工具不可靠的时候，需要让用户来真正的体验这些优化效果。可以做一些可用性测试，增加测试的范围，A / B测试，这些才能真正的让你确认你的优化是让用户更开心还是更沮丧。

## 总结

Android app的用户体验直接跟屏幕上展现的内容相关。如果app的内容加载很慢或者滑动不够流畅，用户的感知就是负面的。在这篇文章，我们讲了如何优化view树形结构，看是否扁平或者简化view等等。我们还讲了怎么检测解决overdraw的问题。还有一些需要深度分析的优化（像CPU导致的问题），systrace很适合发现和解决这种卡顿问题。最后是一些让你的app感觉更快的小技巧，比如把CPU或者网络相关的任务延后处理，不要影响绘制渲染。

[英文-原文链接](#)

# facebook新闻页ListView优化

来源:[blog.aaapei.com](http://blog.aaapei.com)

## 引言

原文链接: <https://code.facebook.com/posts/87949888759525/fast-rendering-news-feed-on-android/> 透漏的信息量不大，且大多数项目并不会遇到facebook这种ListView的场景，不过可以拓展下思路：逻辑单元不一定是视图单元；移动端不要死搬MVC的架构，在市场上仍是中低端机型为主时，还是应该多考虑性能；附上[rebbit的关于本文的讨论](#)，有些干货 :)

## 基础知识

android系统每隔16.7ms发出一个渲染信号，通知ui线程进行界面的渲染。为了达到流畅的体验，应用程序需要在这个时间内完成应用逻辑，使系统达到60fps。当一个Listview被添加到布局时，其关联的adapter的getView方法将会被回调。在16.7毫秒这样一个时间单元内，可见listitem单元的getView方法将被按照顺序执行。在大多数情况下，由于其他绘图行为的存在，例如measure和draw， getView实际分配到执行时间远低于16ms。一旦listview包含复杂控件时，在16毫秒内不能完成渲染，用户只能看到上一帧的结果，这时就发生了掉帧。

## Facebook新闻页介绍

Facebook的新闻页是一个复杂的listview控件，如何使它获得流畅的滚动体验一直困扰我们。首先，新闻页的每一条新闻的可见区域非常大，包含一系列的文本以及照片；其次，新闻的展现类型也很多样，除了文本以及照片，新闻的附件还可包含链接、音频、视频等。除此之外，新闻还可以被点赞、被转载，导致一个新闻会被其他新闻包含在内。当新闻被大量用户转载时，甚至会出现一条新闻占据两个屏幕的情况。加上android用户的机型多为中低端设备，这使我们在16.7ms内完成新闻页的渲染变的非常困难。



## 新闻页最初架构

在2012年，我们将新闻页从web-view转化成本地控件，在最初的那个版本中，基于View-Model-Binder设计模型，我们为新闻listitem创建了一个自定义StoryView类，这个类有一个bindModel方法，该方法用于和数据进行绑定。代码是这样的：

```

public class NewsFeedStoryView extends LinearLayout {
 // ...

 public void bindModel(NewsFeedStory story) {
 // ...
 mHeaderView.bindModel(story);
 mMessageView.bindModel(story);
 mAttachmentsView.bindModel(story);
 // ... and so on ...
 }

 // ...
}

public class HeaderView extends RelativeLayout {
 // ...

 public void bindModel(NewsFeedStory story) {
 // ...
 mTitle.setText(story.getTitle());
 // ...
 }
}

```

StoryView的包含的子控件都会有一个bindModel方法，例如HeaderView通过该方法与其相关的数据进行绑定。这种设计，代码非常直观清晰，但他的缺点也很明显：

- listview复用机制不能有效的工作,Android's recycling mechanism does not work well in this case: Every item in the ListView was usually a StoryView, but once bound to a story, two StoryViews would be radically different and recycling one into the other wasn't effective. (这一段存疑，直接放原文)
- 逻辑嵌套：采用bindModel绑定控件和数据，业务逻辑与视图逻辑耦合，导致逻辑类层次非常深；
- 布局嵌套非常深：不但导致低效的视图渲染，例如新闻被不停的转载的极端场景下还会导致栈溢出；
- bindModel方法逻辑过重：bindModel方法在当用户滚动列表时被ui线程回调，由于所有的数据解析都在这个方法内，导致该方法耗时

以上这些问题虽有他们单独的解决方法，例如我们可以自己设计一套回收机制解决storyView复用问题。但基于维护成本和开发时间考虑，我们决定进行一次重构。

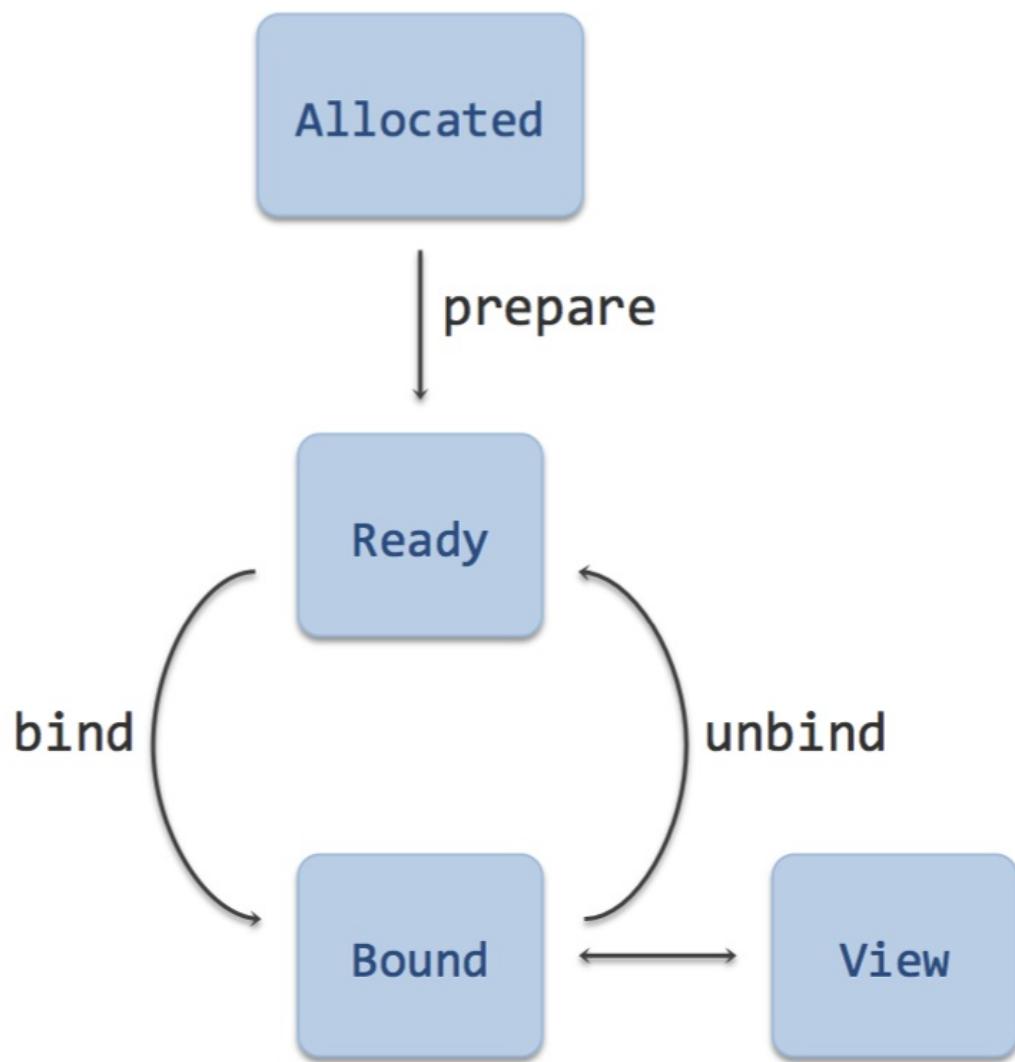
## 重构方案

重构工作大约是一年之前开始的，为了解决前一个架构的问题，首先我们决定将一条新闻分隔成多个listview item。例如，新闻的headerview将是一个独立的listitem。这样，我们可以利用android回收机制，HeaderView新闻子控件将被不同的新闻复用。另外，切分成

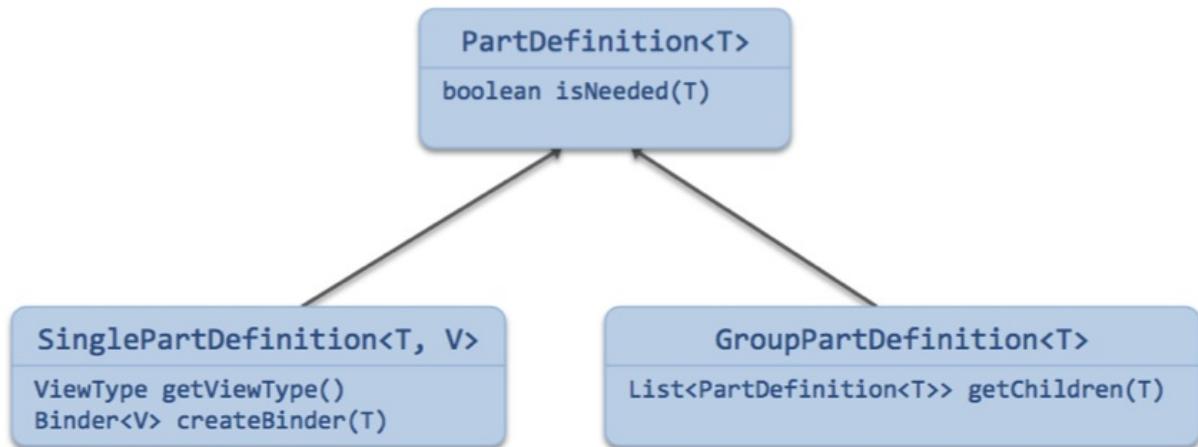
小view也使得内存占用更小，在之前的架构中，Storyview部分的可见会导致这个Storyview被加载到内存中，而现在，粒度更小，只有可见的子控件才会被加载。



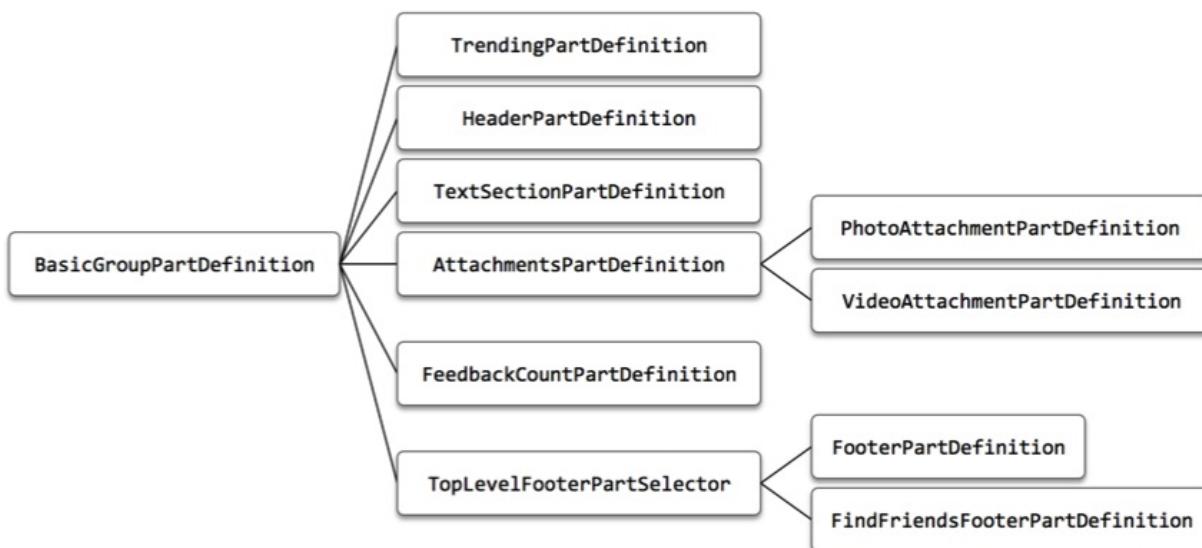
另一个大的修改是，我们将视图逻辑和数据逻辑分离，StoryView被分离成两个类：只负责展现的视图类，以及一个Binder类。视图类仅包含set方法（例如HeaderView包含了setTitle, setSubTitle, setProfilePic等等）。Binder类包含了原来的bindMethod的逻辑，binder类包含三个方法：prepare, bind, unbind。bind方法调用view的set方法设置数据，unbind清理视图数据，prepare方法在cpu空闲期间做一些预初始化工作，例如进行click事件绑定、数据格式化、创建Spannable等等，它会在getView方法之前被调用



我们遇到的技术难点是Binder的设计，由于StoryView被拆分不同的子控件，一条新闻可能会包含多个不同的Binder。而在之前，我们只需要根据视图的树结构进行结构化赋值。因此，我们引进了PartDefinition类，PartDefinition负责维护一条新闻包含哪些子控件、包含Binder的类型以及为新闻创建Binder类，有两种类型的PartDefinition：单个PartDefinition以及PartDefinition集合。



一个新闻在重构之后的PartDefinition结构是这样的：



## 结论

- 采取新的架构，内存错误减少了17%，总crash率减少了8%，彻底解决涨溢出问题
- 渲染时间减少了10%，大新闻场景不再掉帧
- 精简了原来的自定义回收机制，同时在重构过程中增加了单元测试

# Instagram是如何提升TextView渲染性能的

来源: [codethink.me](http://codethink.me)

原文链接: [Improving Comment Rendering on Android](https://medium.com/@iginternals/improving-comment-rendering-on-android-3e3a2a2a2a2a)

上周，成千上万来自全世界的IG用户齐聚在社区组织的先下聚会 Worldwide InstaMeet11 上。WWIM11 是历史上最大，最具地域多样性的Instagram聚会，从Muscat到 Bushwick，成千上万用户分享了大约10万张照片。

每月世界上有超过3亿用户每月使用IG，其中65%来自美国以外的国家，无论用户在哪，我们一致致力于让IG更快，更容易使用。自从去年夏天IG重新设计后，我们在继续努力提升性能。

我们最近的一项改进是关于渲染庞大复杂的文本以及如何通过改进它优化IG的feed滚动。我们希望你可以从我们的经验中找到提升自己app速度的方法。

## 产品需求和性能问题

在IG中，feed是由图片，视频和文字组成的。对于每个图片和视频，我们需要展示对应的图片说明和5条最近的评论。由于用户通常通过图片说明来讲述图片背后的故事，这些图片说明通常是大段复杂的文字，甚至可能包含链接和emoji表情。

# Measuring

20~30ms

# Drawing

30~60 ms



渲染这种复杂文本的主要问题在于它滚动时对性能的影响。在Android中，文本的渲染是很慢的。即使在一个像Nexus 5这样的新设备上，一段有十几行复杂文本的图片说明的初始绘制时间可能会达到50ms，而其文本的measure阶段就需要30ms。这些都发生在UI线程，在滚动时会导致app跳帧。

## 使用text.Layout，缓存text.Layout

Android有很多用于文字展示的控件，但实际上，他们都用text.Layout进行渲染。例如，TextView会将String转化为一个text.Layout对象，并通过canvas API将它绘制到屏幕上。

由于text.Layout需要在构造函数中测量文本的高度，因此它的创建效率不高。缓存text.Layout和复用text.Layout实例可以节省这部分时间。Android的TextView控件并没有提供设置TextLayout的方法，但是添加一个这样的方法并不困难：

```

public class TextLayoutView extends View {

 private Layout mLayout;

 public void setTextLayout(Layout layout) {
 mLayout = layout;
 }

 @Override
 protected void onDraw(Canvas canvas) {
 super.onDraw(canvas);

 canvas.save();

 if (mLayout != null) {
 canvas.translate(getPaddingLeft(), getPaddingTop());
 mLayout.draw(canvas);
 }

 canvas.restore();
 }
}

```

使用自定义的view来手动绘制text.Layout会提升其性能： TextView是一个包含大量特性的通用控件。如果我们只需要在屏幕上渲染静态的，可点击的文本，事情就简单多了：

- 我们可以不用从SpannableStringBuilder转化到String。根据你的文本中是否包含链接，底层的TextView可能会复制一份你的字符串，这需要分配一些内存。
- 我们可以一直使用StaticLayout，这比DynamicLayout要稍微快一些。
- 我们可以避免使用TextView中其他的逻辑：监听文本修改的逻辑，展示嵌入drawable的逻辑，绘制编辑器的逻辑以及弹出下拉列表的逻辑。

通过使用TextLayoutView，我们可以缓存和复用text.Layout，从而避免了每次调用 TextView的 `setText(CharSequence c)` 方法时都要花费20ms来创建它。

## 下载feed后准备好Layout缓存

由于我们确定会在下载评论后展示他们，一个简单的改进是在下载它们后就准备好text.Layout的缓存。

```

if (shouldWarmTextLayoutsCache()) {
 prepareCommentTextLayouts(feedResponse.getItems());
}

```

## 停止滚动后准备好TextLayoutCache

在可以设置text.Layout缓存后，我们的到来常数级的测量（measure）和绑定（binding）时间。但是初次绘制的时间仍然很长。50ms的绘制时间可能会导致明显的卡顿。

这50ms中的大部分被用于测量文本高度以及产生文字符号。这些都是CPU操作。为了提升文本渲染速度，Android在ICS中引入了TextLayoutCache用于缓存这些中间结果。

TextLayoutCache是一个LRU缓存，缓存的key是文本。如果查询缓存时命中，文本的绘制速度会有很大提升。

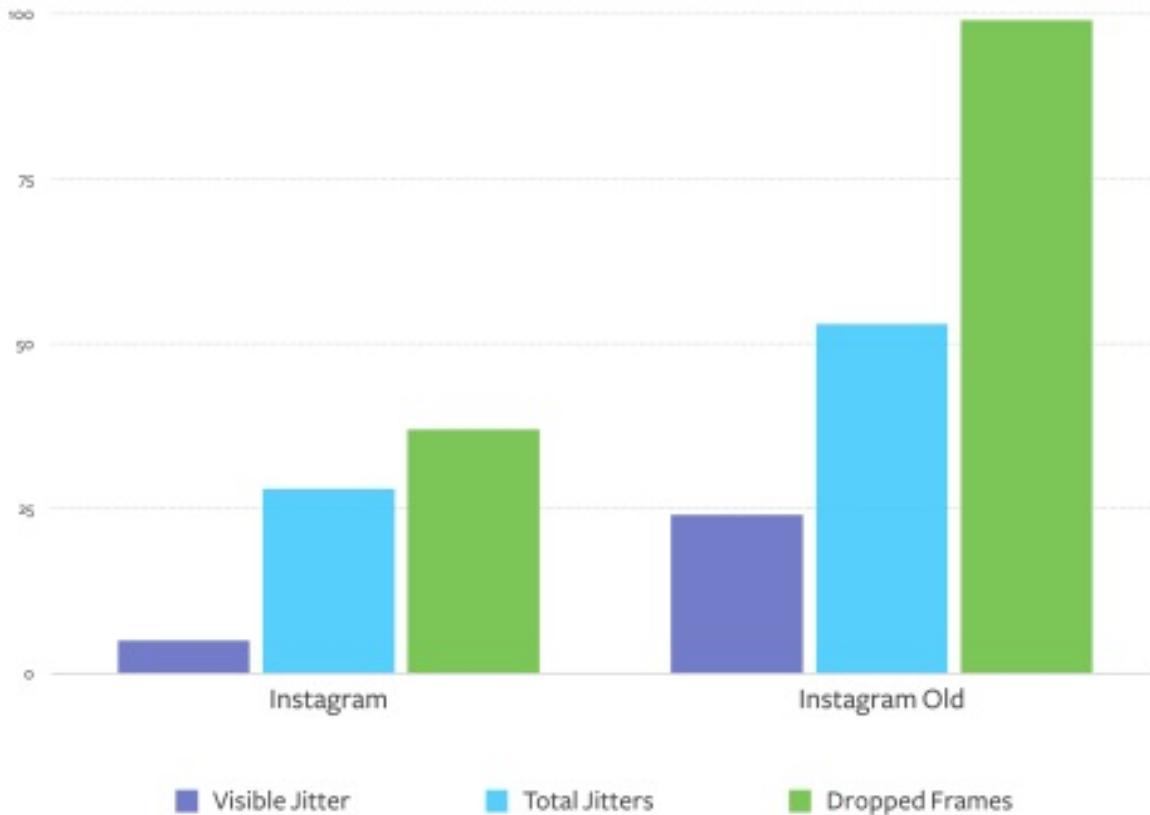
在我们的测试中，这种缓存可以将绘制时间从30ms-50ms减少到2ms-6ms。



为了更好的提升绘制性能，我们可以在绘制文本到屏幕前准备好这个缓存。我们的思路是在一块屏幕外的canvas上虚拟的绘制这些文本。这样在我们绘制文本到屏幕前，TextLayoutCache就已经在一个背景线程中被准备好了。

```
Layout textLayout = getTextLayoutForComment(comment);
textLayout.draw(mPicture.beginRecording(textLayout.getWidth(), textLayout.getHeight()));
mPicture.endRecording();
```

默认情况下，TextLayoutCache的大小为0.5M，这足以缓存十几张图片的评论。我们决定在用户停止滑动时准备缓存，我们向用户滑动的方向提前缓存5个图片的评论。任何时候，我们都至少在任何一个方向上缓存了5个图片的评论。



在应用了所有的这些优化后，掉帧情况减少了60%，而卡顿的情况减少了50%。我们希望这些能帮助你提升你app的速度和性能。告诉我们你的想法吧，我们期待听到你的经验。

# 启动速度

# Android端应用秒开优化体验

来源:[Android端应用秒开优化体验](#)

## 前言

最近部门内抛出了一个问题，应用启动很慢、卡图标？主要表现在中低端机型中。究其这个问题，由于对性能优化比较感兴趣，借了个低端机和一个中端机来一看究竟，对同一应用分别测了下它在中低端机的启动时间，下面为启动耗时情况：

```
192:~ Sunzxyong$ adb shell am start -W com.koudai.weidian.buyer/.activity.SplashActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.koudai.weidian.buyer/.activity.SplashActivity }
Status: ok
Activity: com.koudai.weidian.buyer/.activity.SplashActivity
ThisTime: 3974
TotalTime: 3974
Complete
192:~ Sunzxyong$ adb shell am start -W com.koudai.weidian.buyer/.activity.SplashActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.koudai.weidian.buyer/.activity.SplashActivity }
Status: ok
Activity: com.koudai.weidian.buyer/.activity.SplashActivity
ThisTime: 3887
TotalTime: 3887
Complete
192:~ Sunzxyong$ adb shell am start -W com.koudai.weidian.buyer/.activity.SplashActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.koudai.weidian.buyer/.activity.SplashActivity }
Status: ok
Activity: com.koudai.weidian.buyer/.activity.SplashActivity
ThisTime: 4119
TotalTime: 4119
Complete
```

启动了三次，基本都在4s左右。

## 原因

究其原因，主要因素是任务在界面绘制前过于集中化。

应用启动过程从用户点击launcher图标到看到第一帧这个过程中，主要会经过以下这些过程：

```
main()>Application:attachBaseContext()>onCreate()>Activity:onCreate()
>onStart()>onPostCreate()>onResume()>onPostResume()
```

而一般我们的初始化任务主要都会集中化在Application:onCreate()方法中，这就使得初始化任务在第一帧绘制之前得完成，这就造成了卡图标、应用启动慢。那么把任务打散呢？分散在LaunchActivity中去分段初始化？还是不行的，因为界面开始绘制是在onResume()方法开始后才开始绘制，所以，得从Activity的创建过程找办法。

main->Activity创建的这个过程会经过一系列framework层的操作，这些操作都是系统自动执行的，不易进行优化，不过可以在Activity创建这个过程前后来找一些蛛丝马迹，因为Activity的创建都会辗转到ActivityThread:performLaunchActivity()这个方法中，在这个方法中可以知道这么几件事：

- 1、先通过Instrumentation:newActivity()来创建一个Activity实例
- 2、再判断Application实例是否已创建，已创建则直接返回，否则调用Instrumentation:newApplication()来创建Application实例，在这个过程中会依次执行attachBaseContext()和\* onCreate()方法
- 3、之后Activity:attach()方法会创建一个PhoneWindow对象，它就是界面，它有一个DecorView，调用setContentView()时会给配置DecorView，其中就会设置一个背景：

```
if (getContainer() == null) {
 final Drawable background;
 if (mBackgroundResource != 0) {
 background = getContext().getDrawable(mBackgroundResource);
 } else {
 background = mBackgroundDrawable;
 }
 mDecor.setWindowBackground(background);
```

我们的View也是add进DecorView中显示，它作为RootView肯定是最先显示，所以可以给它设置个默认背景

- 4、最后依次调用Activity的onCreate、onStart等方法

## 措施

- 1、任务分级
- 2、任务并行
- 3、界面预显示

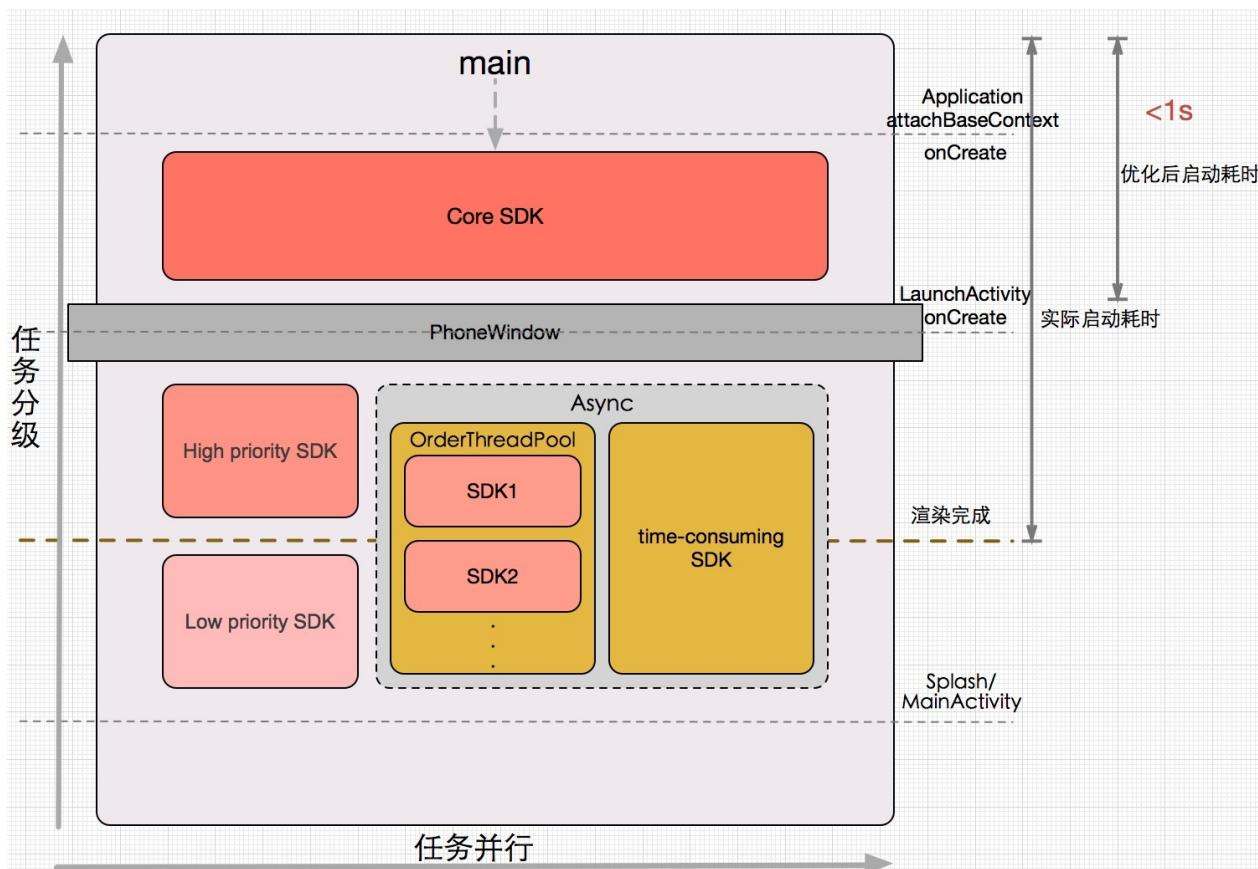
对于任务集中初始化化、耗时初始化原因导致应用在中低端机启动过慢，而Activity界面绘制的时机导致简单的将任务分给Activity初始化也不起作用，我们必须找一个切入点

界面的创建和界面的绘制，这两个过程第一个是Application的attachBaseContext和onCreate这两个方法影响的，第二个则是Application创建一直到界面绘制

所以，可以对任务进行分级的临界点可以这样分：

- 1、CoreSDK——Application的创建
- 2、HighPrioritySDK——Activity的创建
- 3、LowPrioritySDK——Activity界面完成绘制
- 4、AsyncSDK——Activity的创建

如图：



对任务这样分级后，测了一下，应用的启动即使在低端机上，也能秒开：

```
ThisTime: 928
TotalTime: 928
Complete
MacBook-Pro-2:~ Sunzxyong$ adb shell am start -W com.gmall.leadui/activity.SplashActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.gmall.leadui/.activity.SplashActivity }
Status: ok
Activity: com.gmall.leadui/.activity.SplashActivity
ThisTime: 868
TotalTime: 868
Complete
MacBook-Pro-2:~ Sunzxyong$ adb shell am start -W com.gmall.leadui/activity.SplashActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.gmall.leadui/.activity.SplashActivity }
Status: ok
Activity: com.gmall.leadui/.activity.SplashActivity
ThisTime: 956
TotalTime: 956
Complete
```

# 分级带来的问题

正常启动过程那肯定是没有问题的，不过有这么几种场景：

- 1、App切回后台，内存不足导致Application被回收，从最近任务列表中恢复界面时Application需重新创建
- 2、应用没挂起时，Push推送需从Notification跳入应用内某界面
- 3、应用没挂起时，浏览器外链需跳入应用内某界面

这些Case可能导致的问题是被跳入的界面使用到了未初始化的SDK，可能导致Crash或者数据异常，所以目标页面启动前必须确保SDK已经初始化，这个过程的原因是没有唤起启动页来初始化SDK，可以通过hook newActivity解决。

```
public Activity newActivity(ClassLoader cl, String className, Intent intent)
 throws InstantiationException,
 IllegalAccessException, ClassNotFoundException {
 if (InitializeOptimizer.isApplicationCreated()
 && (InitializeUtil.isOuterChainIntent(intent) ||
 InitializeUtil.isInnerChainIntent(intent)) ||
 && (!InitializeOptimizer.isHighSDKInitialized() ||
 !InitializeOptimizer.isLowSDKInitialized() ||
 !InitializeOptimizer.isAsyncSDKInitialized())) {
 InitializeOptimizer.setApplicationCreated(false);
 intent.addCategory(InitializeUtil.INITIALIZE_CATEGORY);
 return (Activity) cl.loadClass(
 InitializeOptimizer.getLaunchClassName()).newInstance();
 }
 InitializeOptimizer.setApplicationCreated(false);
 return super.newActivity(cl, className, intent);
}
```

# 怎么计算apk的启动时间

来源:[知乎](#)

## 问题

利用python或者直接用adb命令怎么计算apk的启动时间呢？就是计算从点击图标到apk完全启动所花费的时间。比如，对游戏来说就是点击游戏图标到进入到登录界面的这段时间。

已知的两种方法貌似可以获取，但是感觉结果不准确：一种是，`adb shell am start -w packagename/activity` ,这个可以得到两个值，`ThisTime` 和 `TotalTime`，不知道两个有什么区别，而且与实际启动时间不匹配，两者相加都可能比实际启动时间小（测试游戏的时候差别更大）；另外一种是通过adb logcat的方式，感觉获取的结果也与实际有差别。

## 回答

[高爷](#)

事实上是可以准确计算的.但是要分场景.

但是要分开游戏和应用. 大家都知道,在Android中,游戏开发和应用开发是两码事.所以我们需要分开来说.

## 1. 应用启动.

我们平时在写应用的时候,一般会指定一个mainActivity,用户在桌面上点击这个Activity的时候,系统会直接起这个Activity. 我们知道Activity在启动的时候会走onCreate/onStart.onResume.这几个回调函数.许多书里讲过,当执行完onResume函数之后,应用就显示出来了...其实这是一种不准确的说法,因为从系统层面来看,一个Activity走完onCreate/onStart.onResume这几个生命周期之后,只是完成了应用自身的一些配置,比如window的一些属性的设置/ View树的建立(只是建立,并没有显示,也就是说只是调用了inflate而已) . 后面ViewRootImpl还会调用两次performTraversals ,初始化Egl以及measure/layout/draw.等.

所以我们定义一个Android应用的启动时间, 肯定不能在Activity的回调函数上下手.而是以用户在手机屏幕上看到你在onCreate的setContentview中设置的layout完全显示为准,也就是我们常说的应用第一帧.

上面扯得有点远,不感兴趣的话可以不看,下面直接说方法.

题主说的adb shell am start -w packagename/activity,是可以完全应用的启动时间的.不过也要分场景.

### 1.1 应用第一次启动

也就是我们常说的冷启动,这时候你的应用程序的进程是没有创建的. 这也是大部分应用的使用场景. 用户在桌面上点击你应用的icon之后,首先要创建进程,然后才启动mainActivity.

这时候adb shell am start -w packagename/activity 返回的结果,就是标准的应用程序的启动时间:

```
adb shell am start -W com.meizu.media.painter/com.meizu.media.painter.PainterMainActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
Status: ok
Activity: com.meizu.media.painter/.PainterMainActivity
ThisTime: 355
TotalTime: 355
WaitTime: 365
Complete
```

总共返回了三个结果,我们以WaitTime为准. (关于ThisTime/TotalTime/WaitTime的区别,各位可以自己去看源码.这里就不详细说了). WaitTime会返回从startActivity到应用第一帧完全显示这段时间.

## 1.2 应用非第一次启动

如果是你按Back键, 并没有将应用进程杀掉的话, 那么执行上述命令就会快一些, 因为不用创建进程了, 只需要启动一个Activity即可。这也就是我们说的应用热启动。

## 2. 游戏启动

游戏启动的话, 就不适用用命令行的方法来启动了, 因为从用户点击桌面图标到登录界面, 既有系统的一部分也有游戏自己的部分。

### 2.1 系统部分

游戏也有一个Activity, 所以启动的时候还是会去启动这个Activity, 所以系统启动部分也就是用户点击桌面桌面响应到这个Activity启动。

### 2.2 游戏部分

一般游戏的主Activity启动后, 还会做一些比较耗时的事情, 这时候你看到的界面是不能操作的, 比如: 加载游戏数据、联网更新数据、读取和更新配置文件、游戏引擎初始化等操作。从游戏开发的角度来看, 到了真正用户能操作的界面才算是一个游戏真正加载完成的时间。

那么这个时间, 就得使用Log来记录了, 因为加载游戏数据、联网更新数据、读取和更新配置文件、游戏引擎初始化这些操作, 都是游戏自己的逻辑, 与系统无关, 所以得由游戏自己定义加载完成的点。

## 3. 准确性

计算机最让人着迷的一点就是其准确性, 1+1永远等于2, 启动耗时多久就是多久, 每一次可能不一样, 但每一次的时间都是这一次的准确时间。

一些愚见, 有想法可以一起交流

## Groffa

终于有空来填坑啦。关于应用启动速度, [@Gracker](#)已经回答的很完善了。我补充下“adb shell am start -W”这条命令得出的三个时间是如何计算出来的, 了解了这个也就清楚了究竟哪个时间更准了。

```
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.android.mms/.ui.ConversationList }
Status: ok
Activity: com.android.mms/.ui.ConversationList
ThisTime: 321
TotalTime: 321
WaitTime: 346
Complete: 346
 wcnWallpaper(WallpaperData, Remote)
 clearWallpaper() void WallpaperManager
 clearWallpaperLocked(boolean, int, Remote)
 out.endDocument();
 stream.close();
 journal.commit();
```

“adb shell am start -W”的实现在

frameworks\base\cmds\am\src\com\android\commands\am\Am.java文件中。其实就是跨Binder调用ActivityManagerService.startActivityAndWait()接口 (后面将

ActivityManagerService简称为AMS），这个接口返回的结果包含上面打印的ThisTime、TotalTime时间。

$$\text{WaitTime} = \text{endTime} - \text{startTime}$$

startTime记录的刚准备调用startActivityAndWait()的时间点，endTime记录的是startActivityAndWait()函数调用返回的时间点，WaitTime = startActivityAndWait()调用耗时。

ThisTime、TotalTime的计算在

frameworks\base\services\core\java\com\android\server\am\ActivityRecord.java文件的reportLaunchTimeLocked()函数中。

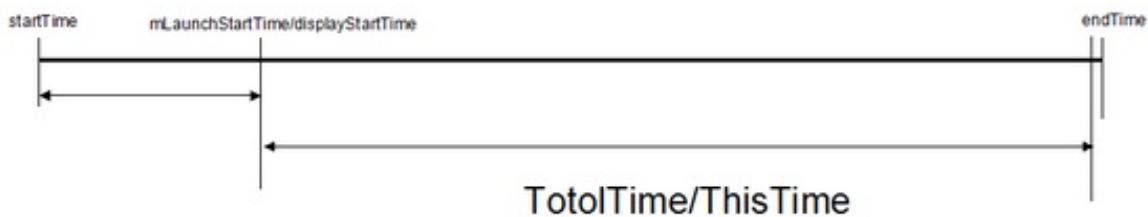
```
private void reportLaunchTimeLocked(final long curTime) {
 final ActivityStack stack = task.stack;
 final long thisTime = curTime - displayStartTime;
 final long totalTime = stack.mLaunchStartTime != 0
 ? (curTime - stack.mLaunchStartTime) : thisTime;
```

我们来解释下代码里curTime、displayStartTime、mLaunchStartTime三个时间变量。

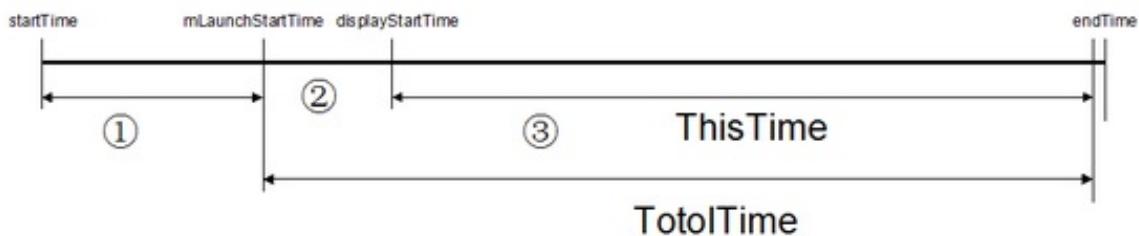
- curTime表示该函数调用的时间点.
- displayStartTime表示一连串启动Activity中的最后一个Activity的启动时间点.
- mLaunchStartTime表示一连串启动Activity中第一个Activity的启动时间点.

正常情况下点击桌面图标只启动一个有界面的Activity，此时displayStartTime与mLaunchStartTime便指向同一时间点，此时ThisTime=TotalTime。另一种情况是点击桌面图标应用会先启动一个无界面的Activity做逻辑处理，接着又启动一个有界面的Activity，在这种启动一连串Activity的情况下（知乎的启动就是属于这种情况），displayStartTime便指向最后一个Activity的开始启动时间点，mLaunchStartTime指向第一个无界面Activity的开始启动时间点，此时ThisTime! =TotalTime。这两种情况如下图：

## 启动单个Activity



## 启动一连串Activity



在上面的图中，我用①②③分别标注了三个时间段，在这三个时间段内分别干了什么事呢？

在第①个时间段内，AMS创建ActivityRecord记录块和选择合理的Task、将当前Resume的Activity进行pause；在第②个时间段内，启动进程、调用无界面Activity的onCreate()等、pause/finish无界面的Activity；在第③个时间段内，调用有界面Activity的onCreate、onResume；

看到这里应该清楚 `ThisTime`、`TotalTime`、`WaitTime` 三个时间的关系了吧。`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。

Event log中TAG=`am_activity_launch_time`中的两个值分表表示 `ThisTime`、`TotalTime`，跟通过“`adb shell am start -W`”得到的值是一致的。

最后再说下系统根据什么来判断应用启动结束。我们知道应用启动包括进程启动、走Activity生命周期onCreate/onResume等。在第一次onResume时添加窗口到WMS中，然后measure/layout/draw，窗口绘制完成后通知WMS，WMS在合适的时机控制界面开始显示(夹杂了界面切换动画逻辑)。记住是窗口界面显示出来后，WMS才调用 `reportLaunchTimeLocked()` 通知AMS Activity启动完成。

最后总结一下，如果只关心某个应用自身启动耗时，参考 `TotalTime`；如果关心系统启动应用耗时，参考 `WaitTime`；如果关心应用有界面Activity启动耗时，参考 `ThisTime`。



# 深入浅出RenderThread

来源:<http://blog.chengdazhi.com/>

[TOC]

原文链接：<https://medium.com/@workingkills/understanding-the-renderthread-4dc17bcaf979#.950cwydhj>

RenderThread是Android Lollipop中引入的新组件，相关文档很少。事实上，在我写这篇文章的时候，只找到三篇相关引用，以及下面这个很模糊的定义：

RenderThread是一个新的由系统控制的处理线程，它可以在UI线程阻塞时保持动画平滑。

为了理解其真实功能，我们需要先介绍几个概念。

---

当设备开启硬件加速时，Android不再在每一帧内都执行绘制任务，而是使用一个叫做“展示列表”的（隐藏的）组件，它通过RenderNode类（曾经是DisplayList类）记录绘制操作集合。

这种间接的方式可以带来诸多好处：

- 一个展示列表可以被多次绘制，而不需要重新执行业务逻辑。
- 特定的操作（如转换、放缩等等）可以覆盖整个列表，无需重新安排某个绘制操作。
- 一旦所有的绘制操作已知，就可以进行优化：比如，如果可能，所有的文字都一起绘制。
- 展示列表的处理工作可能可以分发给另一个线程执行。

上面的第四点正是RenderThread的工作之一：处理优化操作与GPU分发，减轻UI线程的压力。

在Lollipop之前你可能会注意到，进行如Activity切换等重量级工作时，想要使View属性动画平滑进行是不可能的。而在Lollipop以后，这些动画，包括如水波等其他效果在相同场景下竟可以流畅进行，其中就依靠RenderThread的帮助。

渲染的真正执行者是GPU，而它自己是不懂任何动画的。展示动画的唯一方法就是对于每一帧发布不同的绘制指令，这个逻辑不是GPU可以处理的。当这些逻辑需要在UI线程中执行时，重量级工作会妨碍新的绘制指令及时发布，于是产生卡顿现象，无论在进行哪种动画。

前面提到了， RenderThread可以负责展示列表流水线的部分工作，但要注意展示列表的创建与修改还是需要在UI线程中完成。

那么如何在子线程中更新动画呢？

当通过硬件加速进行绘制时， Canvas的实现类叫做DisplayListCanvas(曾经叫GLES20Canvas)，它有许多绘制方法的重载方法，这些方法不是接收一个直接的参数，而是一个CanvasProperty的引用，这个CanvasProperty封装了需要的参数值。这样一来在UI线程创建的展示列表仍然可以静态地调用绘制方法，而且这些调用的参数可以通过CanvasProperty映射被动态修改（在RenderThread中异步修改）。

之后还有一步：CanvasProperty的值需要通过RenderNodeAnimator来随时间变动，由此动画被配置并启动。

产生的动画有这些有趣的属性：

- 目标DisplayListCanvas：需要被人工设定，且之后不可以再修改。
- 即发即弃：一旦被启动就只能被取消，也就是说无法暂停/继续。而且不能知道当下的值。
- 可以提供一个自定义的Interpolator，其代码会在RenderThread中调用。
- 如有延迟启动，会在RenderThread中进行等待。

## 下面是能在RenderThread中操作的动画（到现在为止）：

View属性（可以通过View.animate访问）：

- 变换（X、Y、Z）
- 放缩（X、Y）
- 旋转（X、Y）
- 透明度Alpha
- 圆形展开动画Circular Reveal

## Canvas方法（通过Canvas属性）

- 画圈drawCircle(centerX, centerY, radius, paint)
- 画圆角矩形drawRoundRect(left, top, right, bottom, cornerRadiusX, cornerRadiusY, paint)

## Paint属性

- 透明度Alpha

- 宽度Stroke Width

看起来Google只是封装了所有实现Material Design动画所需的绘制操作。这看起来虽然很有限，但是只需要一点创造力就可以实现各种不同的动画，从各种水波动画到全新的效果。这样的动画操作的好处是可以提供不在UI线程运行的不卡顿的动画。

现在看起来在Android N中RenderThread的能力会被加强（比如AnimatedVectorDrawable将会在其中完成），或许有朝一日它会进入public API。

---

## 我可以让我自己的动画运行在RenderThread中吗？

官方的简短回答：除了View.animate与ViewAnimationUtils.createCircularReveal提供的动画都不可以。

非官方的长一些的回答：本文所说的每一个组件都是隐藏的，所以如果要使用哪个组件都需要通过反射获得所需类和方法的引用，进行封装以保证类型安全，提供获取失败的回调方法等等。详情见我的这个repo

或许这种方法不应该被实际应用，这一点需要你自己把握。

使用RenderThread很简单，一般有三步：

```
CanvasProperty<Float> centerXProperty;
CanvasProperty<Float> centerYProperty;
CanvasProperty<Float> radiusProperty;
CanvasProperty<Paint> paintProperty;

Animator radiusAnimator;
Animator alphaAnimator;

@Override
protected void onDraw(Canvas canvas) {

 if (!animationInitialised) {
 // 1. 创建绘制动画所需要的所有CanvasProperty
 centerXProperty = RenderThread.createCanvasProperty(canvas, initialCenterX);
 centerYProperty = RenderThread.createCanvasProperty(canvas, initialCenterY);
 radiusProperty = RenderThread.createCanvasProperty(canvas, initialRadius);
 paintProperty = RenderThread.createCanvasProperty(canvas, paint);

 // 2. 创建一个或多个Animator，与你想操作的属性对应
 radiusAnimator = RenderThread.createFloatAnimator(this, canvas, radiusProperty);
 alphaAnimator = RenderThread.createPaintAlphaAnimator(this, canvas, paintProperty);
 radiusAnimator.start();
 alphaAnimator.start();
 }

 // 3. 绘制到Canvas上
 RenderThread.drawCircle(canvas, centerXProperty, centerYProperty, radiusProperty,
}
```

在上面的Repo中有完整的示例。

欢迎关注我的公众号“androidway”，将零碎时间都用在刷干货上！



# 兼容性问题

# Android M 新的运行时权限开发者需要知道的一切

来源:[jijiaxin89.com](http://jijiaxin89.com)

android M 的名字官方刚发布不久， 最终正式版即将来临！

android在不断发展， 最近的更新 M 非常不同， 一些主要的变化例如运行时权限将有颠覆性影响。惊讶的是android社区鲜有谈论事儿， 尽管这事很重要或许在不远的将来会引发很严重的问题。

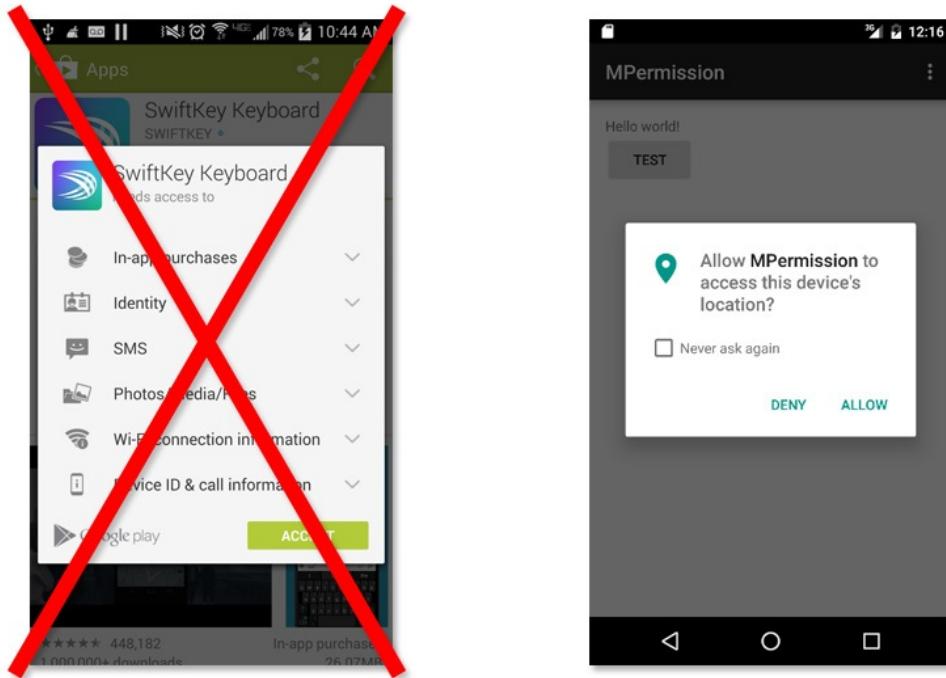
这是今天我写这篇博客的原因。这里有一切关于android运行时权限你需要知道的， 包括如何在代码中实现。现在亡羊补牢还不晚。

## 新运行时权限

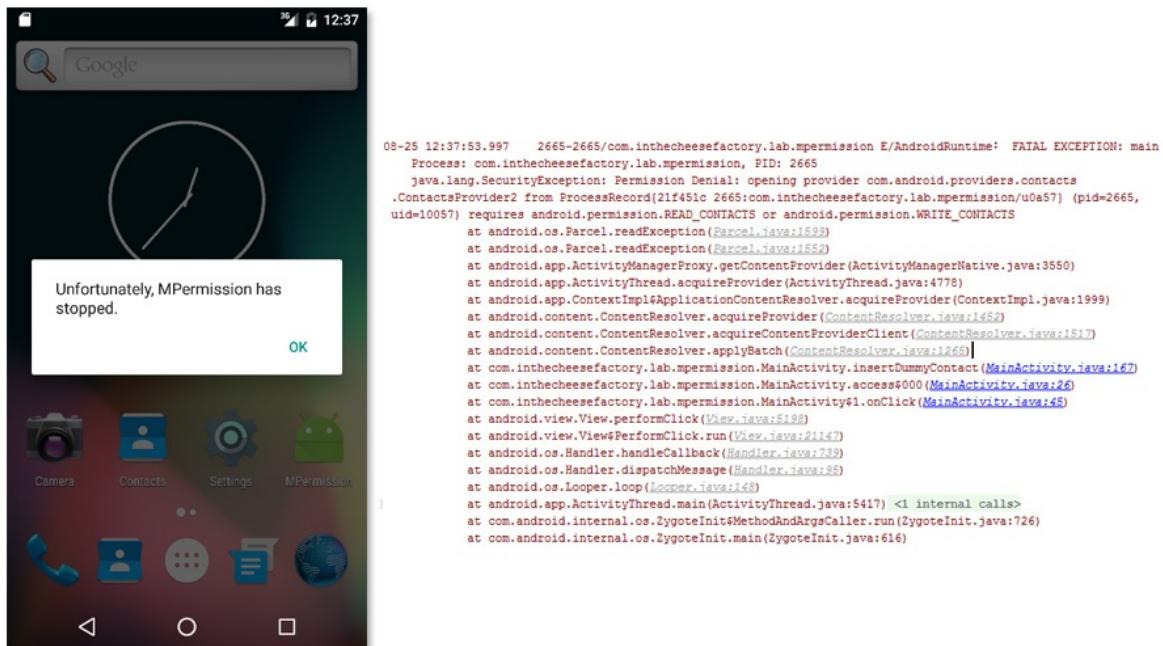
android的权限系统一直是首要的安全概念， 因为这些权限只在安装的时候被询问一次。一旦安装了， app可以在用户毫不知晓的情况下访问权限内的所有东西。

难怪一些坏蛋利用这个缺陷恶意收集用户数据用来做坏事了！

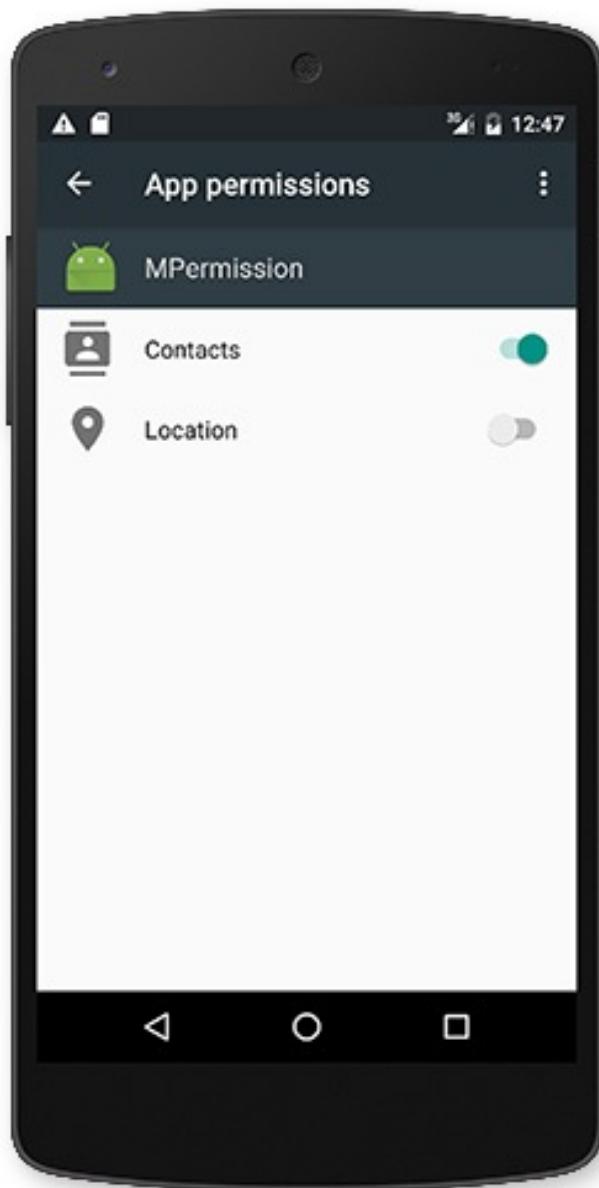
android小组也知道事儿。7年了！权限系统终于被重新设计了。在android6.0棉花糖， app将不会在安装的时候授予权限。取而代之的是， app不得不在运行时一个一个询问用户授予权限。



注意权限询问对话框不会自己弹出来。开发者不得不自己调用。如果开发者要调用的一些函数需要某权限而用户又拒绝授权的话，函数将抛出异常直接导致程序崩溃。



另外，用户也可以随时在设置里取消已经授权的权限。



或许已经感觉到背后生出一阵寒意。。。如果你是个Android开发者，意味着要完全改变你的程序逻辑。你不能像以前那样直接调用方法了，你不得不为每个需要的地方检察权限，否则app就崩溃了！

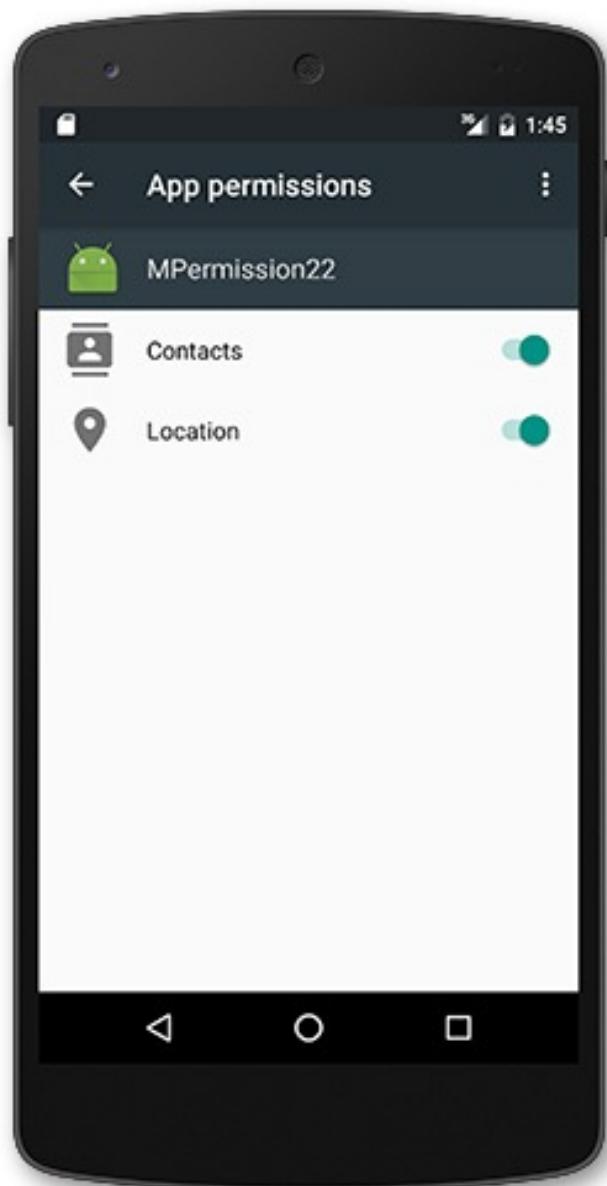
是的。我不能哄你说这是简单的事儿。尽管这对用户来说是好事，但是对开发者来说就是噩梦。我们不得不修改编码不然不论短期还是长远来看都是潜在的问题。

这个新的运行时权限仅当我们设置**targetSdkVersion to 23**（这意味着你已经在23上测试通过了）才起作用，当然还要是M系统的手机。app在6.0之前的设备依然使用旧的权限系统。

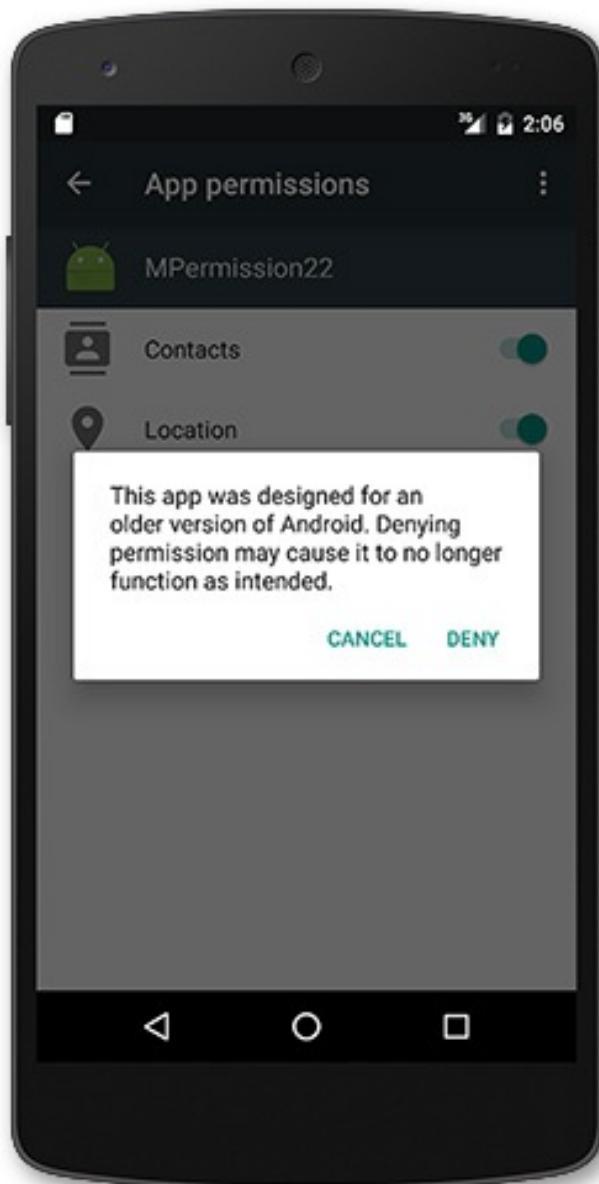
## 已经发布了的app会发生什么

新运行时权限可能已经让你开始恐慌了。“hey，伙计！我三年前发布的app可咋整呢。如果他被装到android 6.0上，我的app会崩溃吗？！？”

莫慌张，放轻松。android小队又不傻，肯定考虑到了这情况。如果app的**targetSdkVersion** 低于 23，那将被认为app没有用23新权限测试过，那将被继续使用旧有规则：用户在安装的时候不得不接受所有权限，安装后app就有了那些权限咯！

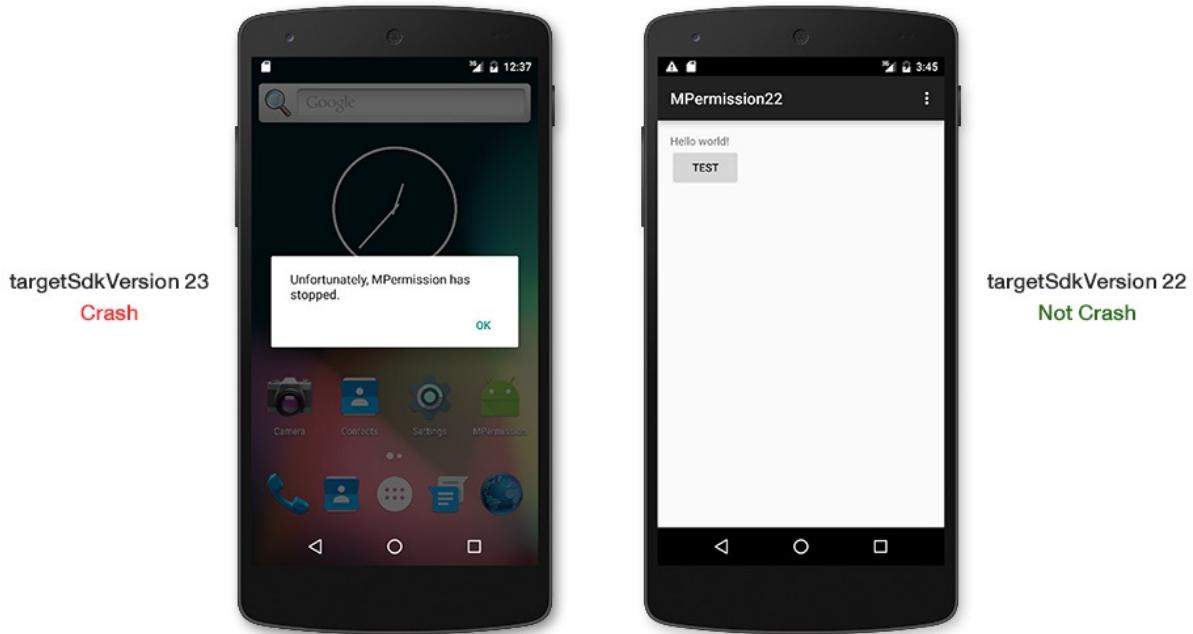


然后app像以前一样奔跑！注意，此时用户依然可以取消已经同意的授权！用户取消授权时，**android 6.0**系统会警告，但这不妨碍用户取消授权。



问题又来了，这时候你的app崩溃吗？

善意的主把这事也告诉了`android`小组，当我们在`targetSdkVersion`低于23的app调用一个需要权限的函数时，这个权限如果被用户取消授权了的话，不抛出异常。但是他将啥都不干，结果导致函数返回值是`null`或者0.



别高兴的太早。尽管app不会调用这个函数时崩溃，返回值null或者0可能接下来依然导致崩溃。

好消息（至少目前看来）是这类取消权限的情况比较少，我相信很少用户这么搞。如果他们这么办了，后果自负咯。

但从长远看来，我相信还是会有大量用户会关闭一些权限。我们app不能在新设备完美运行这是不可接受的。

怎样让他完美运行呢，你最好修改代码支持最新的权限系统，而且我建议你立刻着手搞起！

代码没有成功改为支持最新运行时权限的app,不要设置targetSdkVersion 23 发布，否则你就有麻烦了。只有当你测试过了，再改为targetSdkVersion 23 。

**警告：**现在你在android studio新建项目，targetSdkVersion 会自动设置为 23。如果你还没支持新运行时权限，我建议你首先把targetSdkVersion 降级到22

## PROTECTION\_NORMAL类权限

当用户安装或更新应用时，系统将授予应用所请求的属于 PROTECTION\_NORMAL 的所有权限（安装时授权的一类基本权限）。这类权限包括：

```
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS
android.permission.ACCESS_NETWORK_STATE
android.permission.ACCESS_NOTIFICATION_POLICY
android.permission.ACCESS_WIFI_STATE
android.permission.ACCESS_WIMAX_STATE
android.permission.BLUETOOTH
android.permission.BLUETOOTH_ADMIN
android.permission.BROADCAST_STICKY
android.permission.CHANGE_NETWORK_STATE
android.permission.CHANGE_WIFI_MULTICAST_STATE
android.permission.CHANGE_WIFI_STATE
android.permission.CHANGE_WIMAX_STATE
android.permission.DISABLE_KEYGUARD
android.permission.EXPAND_STATUS_BAR
android.permission.FLASHLIGHT
android.permission.GET_ACCOUNTS
android.permission.GET_PACKAGE_SIZE
android.permission.INTERNET
android.permission.KILL_BACKGROUND_PROCESSES
android.permission.MODIFY_AUDIO_SETTINGS
android.permission.NFC
android.permission.READ_SYNC_SETTINGS
android.permission.READ_SYNC_STATS
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.REORDER_TASKS
android.permission.REQUEST_INSTALL_PACKAGES
android.permission.SET_TIME_ZONE
android.permission.SET_WALLPAPER
android.permission.SET_WALLPAPER_HINTS
android.permission.SUBSCRIBED_FEEDS_READ
android.permission.TRANSMIT_IR
android.permission.USE_FINGERPRINT
android.permission.VIBRATE
android.permission.WAKE_LOCK
android.permission.WRITE_SYNC_SETTINGS
com.android.alarm.permission.SET_ALARM
com.android.launcher.permission.INSTALL_SHORTCUT
com.android.launcher.permission.UNINSTALL_SHORTCUT
```

只需要在AndroidManifest.xml中简单声明这些权限就好，安装时就授权。不需要每次使用时都检查权限，而且用户不能取消以上授权。

## 让你的app支持新运行时权限

是时候让我们的app支持新权限模型了，从设置 compileSdkVersion 和 targetSdkVersion 为23开始吧。

```

 android {
 compileSdkVersion 23
 ...
 }

 defaultConfig {
 ...
 targetSdkVersion 23
 ...
 }
}

```

例子，我想用以下方法添加联系人。

```

private static final String TAG = "Contacts";
private void insertDummyContact() {
 // Two operations are needed to insert a new contact.
 ArrayList<ContentProviderOperation> operations = new ArrayList<ContentProviderOpe

 // First, set up a new raw contact.
 ContentProviderOperation.Builder op =
 ContentProviderOperation.newInsert(ContactsContract.RawContacts.CONTENT_U
 .withValue(ContactsContract.RawContacts.ACCOUNT_TYPE, null)
 .withValue(ContactsContract.RawContacts.ACCOUNT_NAME, null);
 operations.add(op.build());

 // Next, set the name for the contact.
 op = ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
 .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0)
 .withValue(ContactsContract.Data.MIMETYPE,
 ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE
 .withValue(ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME,
 "__DUMMY CONTACT from runtime permissions sample");
 operations.add(op.build());

 // Apply the operations.
 ContentResolver resolver = getContentResolver();
 try {
 resolver.applyBatch(ContactsContract.AUTHORITY, operations);
 } catch (RemoteException e) {
 Log.d(TAG, "Could not add a new contact: " + e.getMessage());
 } catch (OperationApplicationException e) {
 Log.d(TAG, "Could not add a new contact: " + e.getMessage());
 }
}

```

上面代码需要 `WRITE_CONTACTS` 权限。如果不询问授权，app就崩了。

下一步像以前一样在 `AndroidManifest.xml` 添加声明权限。

```
<uses-permission android:name="android.permission.WRITE_CONTACTS"/>
```

下一步，不得不再写个方法检查有没有权限。如果没有弹个对话框询问用户授权。然后你才可以下一步创建联系人。

权限被分组了，如下表：

| Permission Group                    | Permissions                                                                                                                                                                                                                                                                                                                                                                    |
|-------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| android.permission-group.CALENDAR   | <ul style="list-style-type: none"> <li>• android.permission.READ_CALENDAR</li> <li>• android.permission.WRITE_CALENDAR</li> </ul>                                                                                                                                                                                                                                              |
| android.permission-group.CAMERA     | <ul style="list-style-type: none"> <li>• android.permission.CAMERA</li> </ul>                                                                                                                                                                                                                                                                                                  |
| android.permission-group.CONTACTS   | <ul style="list-style-type: none"> <li>• android.permission.READ_CONTACTS</li> <li>• android.permission.WRITE_CONTACTS</li> <li>• android.permission.GET_ACCOUNTS</li> </ul>                                                                                                                                                                                                   |
| android.permission-group.LOCATION   | <ul style="list-style-type: none"> <li>• android.permission.ACCESS_FINE_LOCATION</li> <li>• android.permission.ACCESS_COARSE_LOCATION</li> </ul>                                                                                                                                                                                                                               |
| android.permission-group.MICROPHONE | <ul style="list-style-type: none"> <li>• android.permission.RECORD_AUDIO</li> </ul>                                                                                                                                                                                                                                                                                            |
| android.permission-group.PHONE      | <ul style="list-style-type: none"> <li>• android.permission.READ_PHONE_STATE</li> <li>• android.permission.CALL_PHONE</li> <li>• android.permission.READ_CALL_LOG</li> <li>• android.permission.WRITE_CALL_LOG</li> <li>• com.android.voicemail.permission.ADD_VOICEMAIL</li> <li>• android.permission.USE_SIP</li> <li>• android.permission.PROCESS_OUTGOING_CALLS</li> </ul> |
| android.permission-group.SENSORS    | <ul style="list-style-type: none"> <li>• android.permission.BODY_SENSORS</li> </ul>                                                                                                                                                                                                                                                                                            |
| android.permission-group.SMS        | <ul style="list-style-type: none"> <li>• android.permission.SEND_SMS</li> <li>• android.permission.RECEIVE_SMS</li> <li>• android.permission.READ_SMS</li> <li>• android.permission.RECEIVE_WAP_PUSH</li> <li>• android.permission.RECEIVE_MMS</li> <li>• android.permission.READ_CELL_BROADCASTS</li> </ul>                                                                   |
| android.permission-group.STORAGE    | <ul style="list-style-type: none"> <li>• android.permission.READ_EXTERNAL_STORAGE</li> <li>• android.permission.WRITE_EXTERNAL_STORAGE</li> </ul>                                                                                                                                                                                                                              |

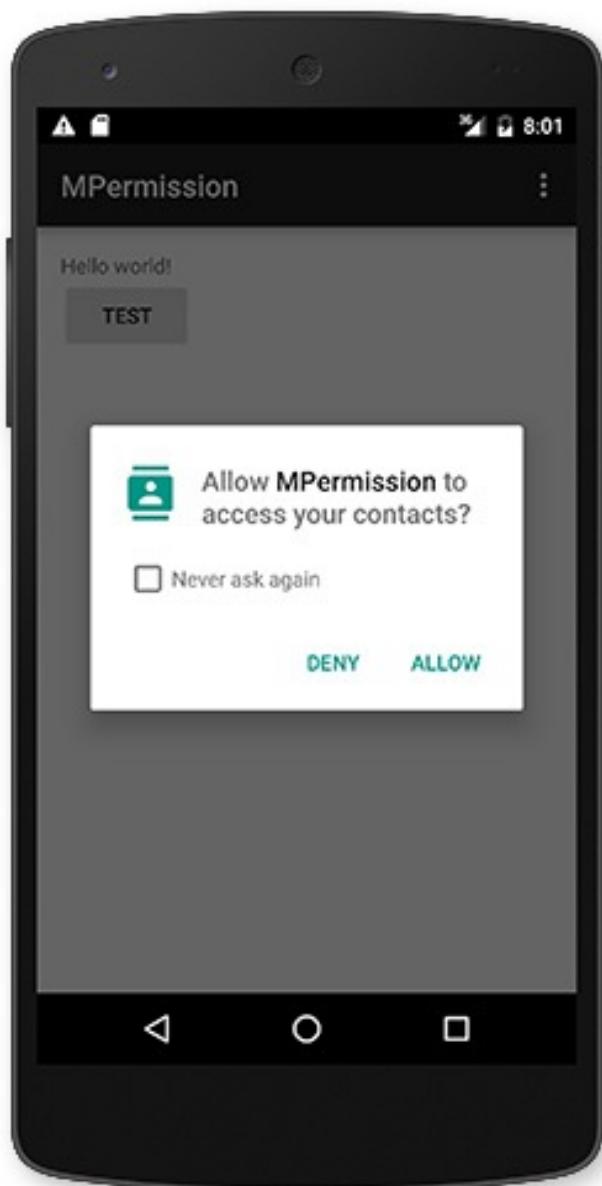
同一组的任何一个权限被授权了，其他权限也自动被授权。例如，一旦 `WRITE_CONTACTS` 被授权了，app也有 `READ_CONTACTS` 和 `GET_ACCOUNTS` 权限了。

源码中被用来检查和请求权限的方法分别是Activity的 `checkSelfPermission` 和 `requestPermissions`。这些方法在api23引入。

```
final private int REQUEST_CODE_ASK_PERMISSIONS = 123;

private void insertDummyContactWrapper() {
 int hasWriteContactsPermission = checkSelfPermission(Manifest.permission.WRITE_CONTACTS);
 if (hasWriteContactsPermission != PackageManager.PERMISSION_GRANTED) {
 requestPermissions(new String[] {Manifest.permission.WRITE_CONTACTS},
 REQUEST_CODE_ASK_PERMISSIONS);
 }
 insertDummyContact();
}
```

如果已有权限，`insertDummyContact()` 会执行。否则，`requestPermissions` 被执行来弹出请求授权对话框，如下：



不论用户同意还是拒绝，activity的 `onRequestPermissionsResult` 会被回调来通知结果（通过第三个参数），`grantResults`，如下：

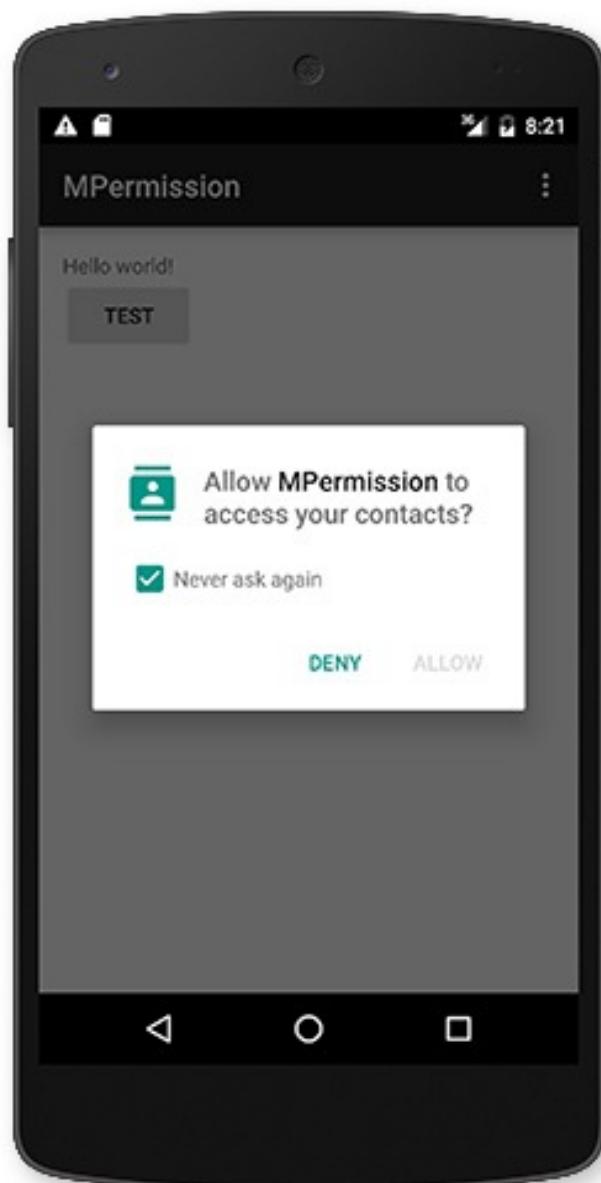
```
@Override
public void onRequestPermissionsResult(int requestCode,
 String[] permissions, int[] grantResults) {
 switch (requestCode) {
 case REQUEST_CODE_ASK_PERMISSIONS:
 if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
 // Permission Granted
 insertDummyContact();
 } else {
 // Permission Denied
 Toast.makeText(MainActivity.this,
 "WRITE_CONTACTS Denied", Toast.LENGTH_SHORT)
 .show();
 }
 break;
 default:
 super.onRequestPermissionsResult(requestCode,
 permissions, grantResults);
 }
}
```

这就是新权限模型工作过程。代码真复杂但是只能去习惯它。。。为了让app很好兼容新权限模型，你不得不用以上类似方法处理所有需要的情况。

如果你想捶墙，现在是时候了。。。

## 处理“不再提醒”

如果用户拒绝某授权。下一次弹框，用户会有一个“不再提醒”的选项的来防止app以后继续请求授权。



如果这个选项在拒绝授权前被用户勾选了。下次为这个权限请求 `requestPermissions` 时，对话框就不弹出来了，结果就是，app啥都不干。

这将是很差的用户体验，用户做了操作却得不到响应。这种情况需要好好处理一下。在请求 `requestPermissions` 前，我们通过 `activity` 的 `shouldShowRequestPermissionRationale` 方法来检查是否需要弹出请求权限的提示对话框，代码如下：

```
final private int REQUEST_CODE_ASK_PERMISSIONS = 123;

private void insertDummyContactWrapper() {
 int hasWriteContactsPermission =
 checkSelfPermission(Manifest.permission.WRITE_CONTACTS);

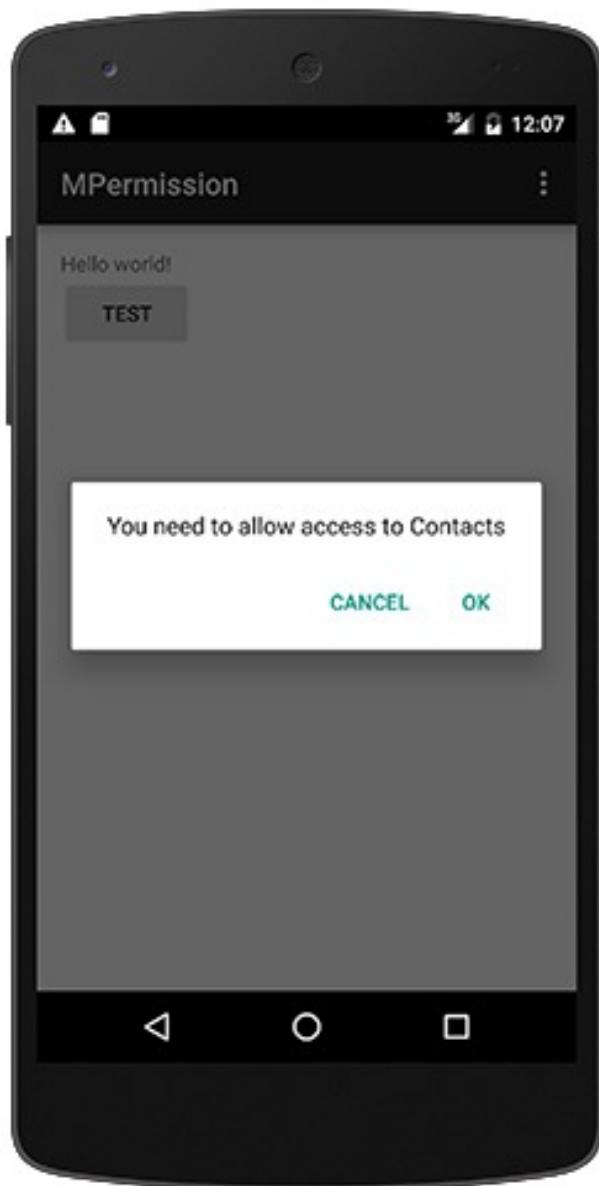
 if (hasWriteContactsPermission != PackageManager.PERMISSION_GRANTED) {
 if (!shouldShowRequestPermissionRationale(Manifest.permission.WRITE_CONTACTS))
 showMessageOKCancel("You need to allow access to Contacts",
 new DialogInterface.OnClickListener() {
 @Override
 public void onClick(DialogInterface dialog, int which) {
 requestPermissions(
 new String[]{Manifest.permission.WRITE_CONTACTS},
 REQUEST_CODE_ASK_PERMISSIONS);
 }
 });
 return;
 }
 requestPermissions(new String[] {Manifest.permission.WRITE_CONTACTS},
 REQUEST_CODE_ASK_PERMISSIONS);
 return;
}

insertDummyContact();

private void showMessageOKCancel(String message,
 DialogInterface.OnClickListener okListener) {
 new AlertDialog.Builder(MainActivity.this)
 .setMessage(message)
 .setPositiveButton("OK", okListener)
 .setNegativeButton("Cancel", null)
 .create()
 .show();
}
```

当一个权限第一次被请求和用户标记过不再提醒的时候,我们写的对话框被展示。

最后一种情况, `onRequestPermissionsResult` 会收到 `PERMISSION_DENIED`, 系统询问对话框不展示。



搞定！

## 一次请求多个权限

当然了有时候需要好多权限，可以用上面方法一次请求多个权限。不要忘了为每个权限检查“不再提醒”的设置。

修改后的代码：

```
final private int REQUEST_CODE_ASK_MULTIPLE_PERMISSIONS = 124;

private void insertDummyContactWrapper() {
 List<String> permissionsNeeded = new ArrayList<String>();

 final List<String> permissionsList = new ArrayList<String>();
 if (!addPermission(permissionsList, Manifest.permission.ACCESS_FINE_LOCATION))
 permissionsNeeded.add("GPS");
 if (!addPermission(permissionsList, Manifest.permission.READ_CONTACTS))
 permissionsNeeded.add("Read Contacts");
 if (!addPermission(permissionsList, Manifest.permission.WRITE_CONTACTS))
 permissionsNeeded.add("Write Contacts");

 if (permissionsList.size() > 0) {
 if (permissionsNeeded.size() > 0) {
 // Need Rationale
 String message = "You need to grant access to " + permissionsNeeded.get(0)
 for (int i = 1; i < permissionsNeeded.size(); i++)
 message = message + ", " + permissionsNeeded.get(i);
 showMessageOKCancel(message,
 new DialogInterface.OnClickListener() {
 @Override
 public void onClick(DialogInterface dialog, int which) {
 requestPermissions(permissionsList.toArray(new String[permissionsList.size()]),
 REQUEST_CODE_ASK_MULTIPLE_PERMISSIONS);
 }
 });
 }
 return;
 }
 requestPermissions(permissionsList.toArray(new String[permissionsList.size()]),
 REQUEST_CODE_ASK_MULTIPLE_PERMISSIONS);
 return;
}

insertDummyContact();
}

private boolean addPermission(List<String> permissionsList, String permission) {
 if (checkSelfPermission(permission) != PackageManager.PERMISSION_GRANTED) {
 permissionsList.add(permission);
 // Check for Rationale Option
 if (!shouldShowRequestPermissionRationale(permission))
 return false;
 }
 return true;
}
```

如果所有权限被授权，依然回调 `onRequestPermissionsResult`，我用hashmap让代码整洁便于阅读。

```
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
 switch (requestCode) {
 case REQUEST_CODE_ASK_MULTIPLE_PERMISSIONS:
 {
 Map<String, Integer> perms = new HashMap<String, Integer>();
 // Initial
 perms.put(Manifest.permission.ACCESS_FINE_LOCATION, PackageManager.PERMISSION_GRANTED);
 perms.put(Manifest.permission.READ_CONTACTS, PackageManager.PERMISSION_GRANTED);
 perms.put(Manifest.permission.WRITE_CONTACTS, PackageManager.PERMISSION_GRANTED);
 // Fill with results
 for (int i = 0; i < permissions.length; i++)
 perms.put(permissions[i], grantResults[i]);
 // Check for ACCESS_FINE_LOCATION
 if (perms.get(Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED
 && perms.get(Manifest.permission.READ_CONTACTS) == PackageManager.PERMISSION_GRANTED
 && perms.get(Manifest.permission.WRITE_CONTACTS) == PackageManager.PERMISSION_GRANTED)
 // All Permissions Granted
 insertDummyContact();
 } else {
 // Permission Denied
 Toast.makeText(MainActivity.this, "Some Permission is Denied", Toast.LENGTH_SHORT)
 .show();
 }
 }
 break;
 default:
 super.onRequestPermissionsResult(requestCode, permissions, grantResults);
 }
}
```

条件灵活的，你自己设置。有的情况，一个权限没有授权，就不可用；但是也有情况，能工作，但是表现的是有所限制的。对于这个我不做评价，你自己设计吧。

## 用兼容库使代码兼容旧版

以上代码在android 6.0以上运行没问题，但是23 api之前就不行了，因为没有那些方法。

粗暴的方法是检查版本

```
if (Build.VERSION.SDK_INT >= 23) {
 // Marshmallow+
} else {
 // Pre-Marshmallow
}
```

但是太复杂，我建议用v4兼容库，已对这个做过兼容，用这个方法代替：

- ContextCompat.checkSelfPermission()

被授权函数返回PERMISSION\_GRANTED，否则返回PERMISSION\_DENIED，在所有版本都是如此。

- ActivityCompat.requestPermissions()

这个方法在M之前版本调用，OnRequestPermissionsResultCallback直接被调用，带着正确的PERMISSION\_GRANTED或者PERMISSION\_DENIED。

- ActivityCompat.shouldShowRequestPermissionRationale()

在M之前版本调用，永远返回false。

用v4包的这三方法，完美兼容所有版本！这个方法需要额外的参数，Context or Activity。别的就没啥特别的了。下面是代码：

```

private void insertDummyContactWrapper() {
 int hasWriteContactsPermission = ContextCompat.checkSelfPermission(MainActivity.this,
 Manifest.permission.WRITE_CONTACTS);
 if (hasWriteContactsPermission != PackageManager.PERMISSION_GRANTED) {
 if (!ActivityCompat.shouldShowRequestPermissionRationale(MainActivity.this,
 Manifest.permission.WRITE_CONTACTS)) {
 showMessageOKCancel("You need to allow access to Contacts",
 new DialogInterface.OnClickListener() {
 @Override
 public void onClick(DialogInterface dialog, int which) {
 ActivityCompat.requestPermissions(MainActivity.this,
 new String[] {Manifest.permission.WRITE_CONTACTS},
 REQUEST_CODE_ASK_PERMISSIONS);
 }
 });
 }
 return;
 }
 ActivityCompat.requestPermissions(MainActivity.this,
 new String[] {Manifest.permission.WRITE_CONTACTS},
 REQUEST_CODE_ASK_PERMISSIONS);
 return;
}
insertDummyContact();
}

```

后两个方法，我们也可以在Fragment中使用，用v13兼容

包: FragmentCompat.requestPermissions() and  
 FragmentCompat.shouldShowRequestPermissionRationale() 和activity效果一样。

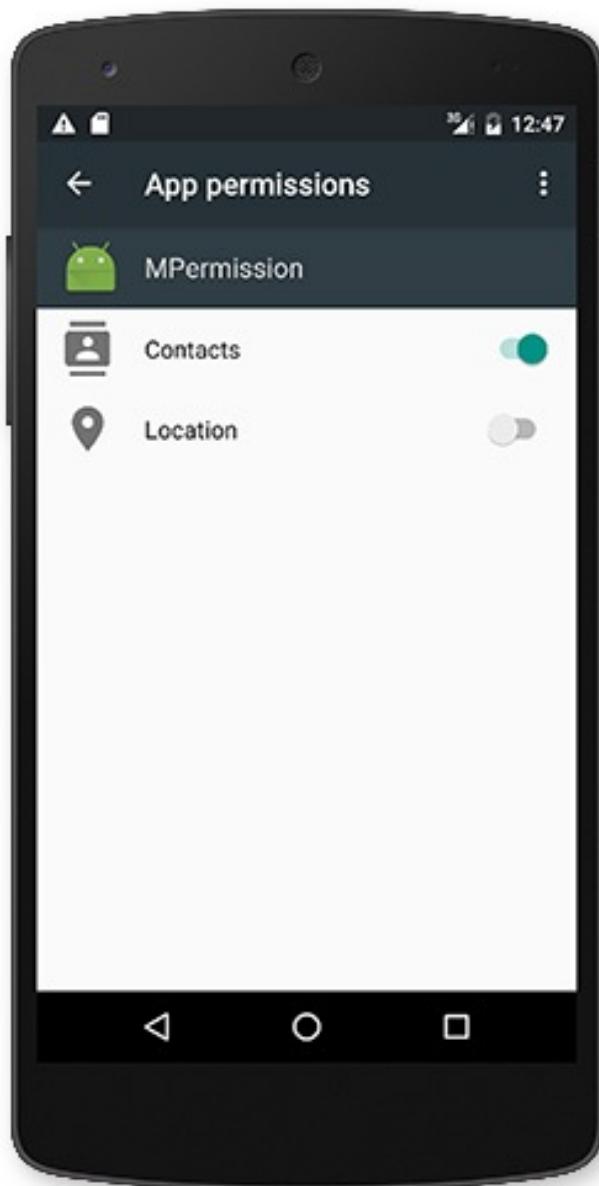
## 第三方库简化代码

以上代码真尼玛复杂。为解决这事，有许多第三方库已经问世了，真66溜真有速度。我试了很多最终找到了个满意的[hotchemi's PermissionsDispatcher](#)。

他和我上面做的一样，只是简化了代码。灵活易扩展，试一下吧。如果不满足你可以找些其他的。

**如果我的app还开着呢，权限被撤销了，会发生在什么**

权限随时可以被撤销。



当app开着的时候被撤消了会发生什么呢？我试过了发现这时app会突然终止 terminated。app中的一切都被简单粗暴的停止了，因为terminated！对我来说这可以理解，因为系统如果允许它继续运行（没有某权限），这会召唤弗雷迪到我的噩梦里。或许更糟...

## 结论建议

我相信你对新权限模型已经有了清晰的认识。我相信你也意识到了问题的严峻。

但是你没得选择。新运行时权限已经在棉花糖中被使用了。我们没有退路。我们现在唯一能做的就是保证app适配新权限模型。欣慰的是只有少数权限需要运行时权限模型。大多数常用的权限，例如，网络访问，属于Normal Permission 在安装时自动会授权，当然你要

声明，以后无需检查。因此，只有少部分代码你需要修改。

两个建议：

- 1. 严肃对待新权限模型
- 2. 如果你代码没支持新权限，不要设置targetSdkVersion 23。尤其是当你在Studio新建工程时，不要忘了修改！

说一下代码修改。这是大事，如果代码结构被设计的不够好，你需要一些很蛋疼的重构。每个app都要被修正。如上所说，我们没的选择。。。

列出所有你需要请求的权限所有情形，如果A被授权，B被拒绝，会发生什么。blah, blah。

祝重构顺利。把它列为你需要做的大事，从现在就开始着手做，以保证M正式发布的时候没有问题。

希望本文对你有用，快乐编码！

译文来自 <http://inthecheesefactory.com/blog/things-you-need-to-know-about-android-m-permission-developer-edition/en>

# android 判断摄像头是否可用（6.0以下）

来源:[CSDN](#)

## 问题概述

android 应用程序无法判定当前是否有摄像头的使用权限，是否可用。

## 问题描述

在做ocr 的时候遇到个问题，点击拍照/扫描页面的入口Button，弹出一个对话框“申请拍照和录像权限”“禁止”/“允许”。如果这个时候点击了“禁止”，或者自己在移动设备的权限设置里把APP的“拍照和摄像”权限改为拒绝（或者禁止）。那么当我们在app 中再想调用拍照和摄像的功能就不行了，有的手机没有任何反应，有的干脆就直接挂掉。

## 问题分析

android 6.0 Google 对Android的权限做了比较大的修改，比较好处理。问题主要是体现在6.0 以下的系统。这是我用的一款测试机打印出来的log，忘记是三星还是小米了。

Caused by: Camera permission has been disabled for current app

```
W/System.err: java.lang.RuntimeException: Camera permission has been disabled for current app
W/System.err: at com.lbe.security.service.core.client.MultimediaClient.newCamera(Unknown Source)
W/System.err: at android.hardware.Camera.native_setup(Native Method)
W/System.err: at android.hardware.Camera.<init>(Camera.java:329)
W/System.err: at android.hardware.Camera.open(Camera.java:302)
W/System.err: at com.lbe.security.service.core.client.MultimediaClient.openCamera(Unknown Source)
W/System.err: at com.lbe.security.service.core.client.MultimediaClient.access$000(Unknown Source)
W/System.err: at com.lbe.security.service.core.client.MultimediaClient$1.run(Thread.java:856)
```

而魅族MX5的比较怪，其他的魅族机没有测试

Caused by: java.lang.RuntimeException: Camera is being used after  
Camera.release() was called

```

E: FATAL EXCEPTION: main
Process: com.xiaozhu.xrdz, PID: 21473
java.lang.RuntimeException: Unable to start activity ComponentInfo{com.xiaozhu.xrdz/com.xiaozhu.xrdz.activity.Order_OcrActivity}: java.lang.RuntimeException: Camera is being used after Camera.release() was called
 at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2608)
 at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2677)
 at android.app.ActivityThread.access$200(ActivityThread.java:178)
 at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1539)
 at android.os.Handler.dispatchMessage(Handler.java:111)
 at android.os.Looper.loop(Looper.java:194)
 at android.app.ActivityThread.main(ActivityThread.java:5774) {2 internal calls}
 at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:1004)
 at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:799)
Caused by: java.lang.RuntimeException: Camera is being used after Camera.release() was called
 at android.hardware.Camera.native_getParameters(Native Method)
 at android.hardware.Camera.getParameters(Camera.java:3195)
 at com.xiaozhu.xrdz.CameraManager.setCameraFlashModel(Unknown Source)
 at com.xiaozhu.xrdz.activity.Order_OcrActivity.onCreate(Unknown Source)
 at android.app.Activity.performCreate(Activity.java:6104)
 at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1123)
 at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2561)
 at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2677)
 at android.app.ActivityThread.access$200(ActivityThread.java:178)
 at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1539)
 at android.os.Handler.dispatchMessage(Handler.java:111)
 at android.os.Looper.loop(Looper.java:194)
 at android.app.ActivityThread.main(ActivityThread.java:5774)
 at java.lang.reflect.Method.invoke(Native Method)
 at java.lang.reflect.Method.invoke(Method.java:372) {2 more...}

```

最开始的时候还以为是摄像头的资源没有释放掉之类的问题引起的，但是在“权限管理”中把摄像头的权限打开就没有这种问题，真心醉了，为什么会报这样的错误？！！！但是问题肯定是权限导致的。

## 问题解决

网上有种方法是根据 `checkPermission` 判断权限，但这是判断是否在清单文件中注册了权限，并不能判断当前的摄像头是否被禁止，可用！

```

PackageManager pm = getPackageManager();
boolean permission = (PackageManager.PERMISSION_GRANTED ==
 pm.checkPermission("android.permission.CAMERA ", "packageName"));
if (permission) {
 showToast("有这个权限");
} else {
 showToast("木有这个权限");
}

```

找到种可行的处理方法，是用抓取 `Camera.open()` 异常的方法来解决，比较暴力。

```

public class PermissionTool {
 /**
 * 判断摄像头是否可用
 * 主要针对6.0 之前的版本，现在主要是依靠try...catch... 报错信息，感觉不太好，
 * 以后有更好的方法的话可适当替换
 *
 * @return
 */
 public static boolean isCameraCanUse() {
 boolean canUse = true;
 Camera mCamera = null;
 try {
 mCamera = Camera.open();
 // setParameters 是针对魅族MX5 做的。MX5 通过Camera.open() 拿到的Camera
 // 对象不为null
 Camera.Parameters mParameters = mCamera.getParameters();
 mCamera.setParameters(mParameters);
 } catch (Exception e) {
 canUse = false;
 }
 if (mCamera != null) {
 mCamera.release();
 }
 return canUse;
 }
}

```

而我们可以在APP进入拍照/扫描 页面的入口处，会先针对权限进行检查。如果有摄像头权限，则正常使用；没有摄像头使用权限，可以弹出个权限弹层或者相关提示之类的告诉用户正在使用的app 没有摄像头权限，需要自己开启。

```

if (PermissionTool.isCameraCanUse()) {
 跳转到相关的拍照/扫描 页面
} else {
 当前APP没有摄像头权限弹层，或者其他相关提示
}

```

关于问题“android 6.0以下系统判断当前摄像头是否可用”目前我是这样处理的，感觉不太好，以后有更好的处理办法再替换吧！

## 补充

```
// 能否录音
public boolean canRecord(){
 boolean canRecorder = true;
 // isStart是为了处理魅族手机自己catch异常的问题
 boolean isStart = false;
 MediaRecorder mr = null;
 try{
 mr = new MediaRecorder();
 mr.setAudioSource(MediaRecorder.AudioSource.MIC);
 mr.setOutputFormat(MediaRecorder.OutputFormat.AMR_NB);

 File dir = Environment.getExternalStorageDirectory();
 mr.setOutputFile(new File(dir,"test.mp3").getPath());

 mr.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);

 mr.prepare();
 mr.start();
 isStart = true;
 }catch (Throwable e){
 canRecorder = false;
 e.printStackTrace();
 }finally {
 try{
 if(mr != null) {
 if(isStart) {
 mr.stop();
 }
 mr.release();
 }
 }catch (Throwable e){
 canRecorder = false;
 e.printStackTrace();
 }
 }
 return canRecorder;
}
```

# 你需要知道的Android拍照适配方案

来源:<http://www.jianshu.com/p/f269bcda335f#rd>

近段时间，家里陪自己度过大学四年的电脑坏了，挑选好的新电脑配件终于在本周全部到货，自己动手完成组装。从AMD到i7的CPU，6G内存到14G内存，打开 AndroidStudio 的速度终于杠杠的上去了，感动到泪流满面啊！！！！！！！扯了这么多，回归一下正题，还是来说说本篇文章要写什么吧！说起调用系统相机来拍照的功能，大家肯定不陌生，现在所有应用都具备这个功能。例如最基本的，用户拍照上传头像。Android开发的孩纸都知道，碎片化给拍照这个功能的实现带来挺多头疼的问题。所以，我决定写写一些网上不多见但又经常听到童鞋们吐槽的问题。

## 拍照功能实现

Android 程序上实现拍照功能的方式分为两种：第一种是利用相机的 API 来自定义相机，第二种是利用 Intent 调用系统指定的相机拍照。下面讲的内容都是针对第二种实现方式的适配。

通常情况下，我们调用拍照的业务场景是如下面这样的：

- A 界面，点击按钮调用相机拍照；
- A 界面得到拍完照片，跳转到 B 界面进行预览；
- B 界面有个按钮，点击后触发某个业务流程来处理这张照片；

实现的大体流程代码如下：

```

//1、调用相机
File mPhotoFile = new File(folder,filename);
Intent captureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
Uri fileUri = Uri.fromFile(mPhotoFile);
captureIntent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri);
mActivity.startActivityForResult(captureIntent, CAPTURE_PHOTO_REQUEST_CODE);

//2、拿到照片
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
 if (requestCode == CapturePhotoHelper.CAPTURE_PHOTO_REQUEST_CODE && resultCode ==
 File photoFile = mCapturePhotoHelper.getPhoto();//获取拍完的照片
 if (photoFile != null) {
 PhotoPreviewActivity.preview(this, photoFile);//跳转到预览界面
 }
 finish();
} else {
 super.onActivityResult(requestCode, resultCode, data);
}
}

//3、各种各样处理这张图片的业务代码

```

到这里基本科普完了如何调用系统相机拍照，相信这些网上一搜一大把的代码，很多童鞋都能看懂。

## 有没有相机可用？

前面讲到我们是调用系统指定的相机app来拍照，那么系统是否存在可以被我们调用的app呢？这个我们不敢确定，毕竟Android奇葩问题多，还真有遇到过这种极端的情况导致闪退的。虽然很极端，但作为客户端人员还是要进行处理，方式有二：

- 调用相机时，简单粗暴的 try-catch
- 调用相机前，检测系统有没有相机 app 可用

try-catch 这种粗暴的方式大家肯定很熟悉了，那么要如何检测系统有没有相机 app 可用呢？系统在 PackageManager 里为我们提供这样一个 API



通过这样一个 API，可以知道系统是否存在 action 为 **MediaStore.ACTION\_IMAGE\_CAPTURE** 的 intent 可以唤起的拍照界面，具体实现代码如下：

```
/**
 * 判断系统中是否存在可以启动的相机应用
 *
 * @return 存在返回true, 不存在返回false
 */
public boolean hasCamera() {
 PackageManager packageManager = mActivity.getPackageManager();
 Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
 List<ResolveInfo> list = packageManager.queryIntentActivities(intent, PackageManager.MATCH_DEFAULT_ONLY);
 return list.size() > 0;
}
```

## 拍出来的照片“歪了”！ ！ ！

经常会遇到一种情况，拍照时看到照片是正的，但是当我们的 app 获取到这张照片时，却发现旋转了 90 度（也有可能是 180、270，不过 90 度比较常见，貌似都是由于手机传感器导致的）。很多童鞋对此感到很困扰，因为不是所有手机都会出现这种情况，就算会是出现这种情况的手机上，也并非每次必现。要怎么解决这个问题呢？从解决的思路上看，只要获取到照片旋转的角度，利用 Matrix 来进行角度纠正即可。那么问题来了，要怎么知道照片旋转的角度呢？细心的童鞋可能会发现，拍完一张照片去到相册点击属性查看，能看到下面这样一堆关于照片的属性数据

< 20/248



拍摄时间: 2016年3月18日 晚上8:15

修改时间: 2016年3月18日 晚上8:15

地点: 广东省, 广州市, 东闸大街, 中国

[在地图上查看](#)

文件名: IMG\_20160318\_201526

尺寸: 2448\*3264px

旋转角度: 0

文件大小: 1.12MB

制造商: Xiaomi

设备: 2014501

闪光灯: 未使用闪光灯

焦距: 3.5 mm

白平衡: 自动

光圈: f/2.2

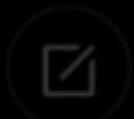
曝光时间: 1/33

ISO: 233

路径: /storage/emulated/0/DCIM/  
Camera/  
IMG\_20160318\_201526.jpg



发送



编辑



删除



更多

没错，这里面就有一个旋转角度，倘若拍照后保存的成像照片文件发生了角度旋转，这个图片的属性参数就能告诉我们到底旋转了多少度。只要获取到这个角度值，我们就能进行纠正的工作了。Android 系统提供了 ExifInterface 类来满足获取图片各个属性的操作

|                      |               |                        |
|----------------------|---------------|------------------------|
| AudioRecord          |               | Type is String.        |
| AudioRecord.Builder  |               |                        |
| AudioTimestamp       |               |                        |
| AudioTrack           |               |                        |
| AudioTrack.Builder   |               |                        |
| CamcorderProfile     |               |                        |
| CameraProfile        |               |                        |
| <b>ExifInterface</b> |               |                        |
| FaceDetector         |               |                        |
| FaceDetector.Face    |               |                        |
| Image                |               |                        |
|                      | <b>String</b> | <b>TAG_MODEL</b>       |
|                      |               | Type is String.        |
|                      | <b>String</b> | <b>TAG_ORIENTATION</b> |
|                      |               | Type is int.           |
|                      | <b>String</b> | <b>TAG_SUBSEC_TIME</b> |
|                      |               | Type is int.           |

通过 ExifInterface 类拿到 TAG\_ORIENTATION 属性对应的值，即为我们想要得到旋转角度。再根据利用 Matrix 进行旋转纠正即可。实现代码大致如下：

```
/*
 * 获取图片的旋转角度
 *
 * @param path 图片绝对路径
 * @return 图片的旋转角度
 */
public static int getBitmapDegree(String path) {
 int degree = 0;
 try {
 // 从指定路径下读取图片，并获取其EXIF信息
 ExifInterface exifInterface = new ExifInterface(path);
 // 获取图片的旋转信息
 int orientation = exifInterface.getAttributeInt(ExifInterface.TAG_ORIENTATION
 switch (orientation) {
 case ExifInterface.ORIENTATION_ROTATE_90:
 degree = 90;
 break;
 case ExifInterface.ORIENTATION_ROTATE_180:
 degree = 180;
 break;
 case ExifInterface.ORIENTATION_ROTATE_270:
 degree = 270;
 break;
 }
 } catch (IOException e) {
 e.printStackTrace();
 }
 return degree;
}

/**
 * 将图片按照指定的角度进行旋转
 *
 * @param bitmap 需要旋转的图片
 * @param degree 指定的旋转角度
 * @return 旋转后的图片
 */
public static Bitmap rotateBitmapByDegree(Bitmap bitmap, int degree) {
 // 根据旋转角度，生成旋转矩阵
 Matrix matrix = new Matrix();
 matrix.postRotate(degree);
 // 将原始图片按照旋转矩阵进行旋转，并得到新的图片
 Bitmap newBitmap = Bitmap.createBitmap(bitmap, 0, 0, bitmap.getWidth(), bitmap.getHeight());
 if (bitmap != null && !bitmap.isRecycled()) {
 bitmap.recycle();
 }
 return newBitmap;
}
```

# 拍完照怎么闪退了？

曾在小米和魅族的某些机型上遇到过这样的问题，调用系统相机拍照，拍完点击确定回到自己的app里面却莫名其妙的闪退了。这种闪退有两个特点：

- 没有什么错误日志（有些机子啥日志都没有，有些机子会出来个空异常错误日志）；
- 同个机子上非必现（有时候怎么拍都不闪退，有时候一拍就闪退）；

对待非必现问题往往比较头疼，当初遇到这样的问题也是非常不解。上网搜罗了一圈也没方案，后来留意到一个比较有意思信息：**有些系统厂商的 ROM 会给自带相机应用做优化，当某个 app 通过 intent 进入相机拍照界面时，系统会把这个 app 当前最上层的 Activity 销毁回收。**（注意：我遇到的情况是有时候很快就回收掉，有时候怎么等也不回收，没有什么必现规律）为了验证一下，便在启动相机的 Activity 中对 onDestory 方法进行加 log。果不其然，终于发现进入拍照界面的时候 onDestory 方法被执行了。所以，前面提到的闪退基本可以推测是 Activity 被回收导致某些非UI控件的成员变量为空导致的。

（有些机子会报出空异常错误日志，但是有些机子闪退了什么都不报，是不是觉得很奇葩！）

既然涉及到 Activity 被回收的问题，自然要想起 onSaveInstanceState 和 onRestoreInstanceState 这对方法。去到 onSaveInstanceState 把数据保存，并在 onRestoreInstanceState 方法中进行恢复即可。大体代码思路如下：

```

@Override
protected void onSaveInstanceState(Bundle outState) {
 super.onSaveInstanceState(outState);
 mRestorePhotoFile = mCapturePhotoHelper.getPhoto();
 if (mRestorePhotoFile != null) {
 outState.putSerializable(EXTRA_RESTORE_PHOTO, mRestorePhotoFile);
 }
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
 super.onRestoreInstanceState(savedInstanceState);
 mRestorePhotoFile = (File) savedInstanceState.getSerializable(EXTRA_RESTORE_PHOTO);
 mCapturePhotoHelper.setPhoto(mRestorePhotoFile);
}

```

对于 onSaveInstanceState 和 onRestoreInstanceState 方法的作用还不熟悉的童鞋，网上资料很多，可以自行搜索。

到这里，可能有童鞋要问，这种闪退并不能保证复现，我要怎么知道问题所在和是否修复了呢？我们可以去到开发者选项里开启**不保留活动**这一项进行调试验证

下午3:05

... 0.00K/s \*

< 开发者选项

## 显示 CPU 使用情况

屏幕上叠加层显示当前 CPU 使用情况

GPU 呈现模式分析



关闭

## 启用 OpenGL 跟踪



无

应用

不保留活动

用户离开后即销毁每个活动

## 后台进程限制



### 标准限制

## 系统内存优化级别



中级

显示所有“应用程序无响应”

为后台应用程序显示“应用程序无响应”(ANR)对话框

显示“强制关闭”

为应用程序显示“强制关闭”(FC)对话框

它作用是保留当前和用户接触的 Activity，并将目前无法和用户交互 Activity 进行销毁回收。打开这个调试选项就可以满足验证的需求，当你的 app 的某个 Activity 跳转到拍照的 Activity 后，这个 Activity 立马就会被系统销毁回收，这样就可以很好的完全复现闪退的场景，帮助开发者确认问题有没有修复了。

涉及到 Activity 被销毁，还想提一下代码实现上的问题。假设当前有两个 Activity，MainActivity 中有个 Button，点击可以调用系统相机拍照并显示到 PreviewActivity 进行预览。有下面两种实现方案：

- 方案一：MainActivity 中点击 Button 后，启动系统相机拍照，并在 MainActivity 的 onActivityResult 方法中获取拍下来的照片，并启动跳转到 PreviewActivity 界面进行效果预览；
- 方案二：MainActivity 中点击 Button 后，启动 PreviewActivity 界面，在 PreviewActivity 的 onCreate（或者 onStart、onResume）方法中启动系统相机拍照，然后在 PreviewActivity 的 onActivityResult 方法中获取拍下来的照片进行预览；

上面两种方案得到的实现效果是一模一样的，但是第二种方案却存在很大的问题。因为启动相机的代码放在 onCreate（或者 onStart、onResume）中，当进入拍照界面后，PreviewActivity 随即被销毁，拍完照确认后回到 PreviewActivity 时，被销毁的 PreviewActivity 需要重建，又要走一遍 onCreate、onStart、onResume，又调用了启动相机拍照的代码，周而复始的进入了死循环状态。为了避免让你的用户抓狂，果断明智的选择方案一。

以上这种情况提到调用系统拍照时，Activity 就回收的情况，在小米4S和小米4 LTE机子上（MIUI的版本是7.3，Android系统版本是6.0）出现的概率很高。所以，建议看到此文的童鞋也可以去验证适配一下。

## 图片无法显示

图片无法显示这个问题也是略坑，如何坑法？往下看，同样是在小米4S和小米4 LTE机子上（MIUI的版本是7.3，Android系统版本是6.0）出现概率很高的场景（当然，不保证其他机子没出现过）。按照我们前面提到的业务场景，调用相机拍照完成后，我们的 app 会有一个预览图片的界面。但是在用了小米的机子进行拍照后，自己 app 的预览界面却怎么也无法显示出照片来，同样是相当郁闷，郁闷完后还是要一步一步去排查解决问题的！为此，需要一步一步猜测验证问题所在。

- 猜测一：没有拿到照片路径，所以无法显示？  
直接断点打 log 跟踪，猜测一很快被推翻，路径是有的。

- 猜测二：Bitmap太大了，无法显示？

直接在 AS 的 log 控制台仔细的观察了一下系统 log，发现了一些蛛丝马迹

```
05-22 17:04:52.453 1773-1877/? I/ActivityManager: Displayed com.clock.study/.activity.PhotoPreviewActivity: +1s572ms
05-22 17:04:52.460 2962-2962/? V/PhoneStatusBarPolicy: updateManagedProfile: mManagedProfileFocused: false mKeyguardVisible: false
05-22 17:04:52.465 31976-32058/com.clock.study W/OpenGLRenderer: Bitmap too large to be uploaded into a texture (3120x4160, max=4096x4096)
05-22 17:04:52.465 31976-32058/com.clock.study W/OpenGLRenderer: Bitmap too large to be uploaded into a texture (3120x4160, max=4096x4096)
05-22 17:04:52.465 31976-32058/com.clock.study W/OpenGLRenderer: Bitmap too large to be uploaded into a texture (3120x4160, max=4096x4096)
05-22 17:04:52.746 1773-1877/? I/Timeline: Timeline: Activity_windows_visible id: ActivityRecord{ca5d6c9 u0 com.clock.study/.activity.Photo
05-22 17:04:52.760 31976-32058/com.clock.study W/OpenGLRenderer: Bitmap too large to be uploaded into a texture (3120x4160, max=4096x4096)
05-22 17:04:52.764 31976-32058/com.clock.study W/OpenGLRenderer: Bitmap too large to be uploaded into a texture (3120x4160, max=4096x4096)
05-22 17:04:52.764 31976-32058/com.clock.study W/OpenGLRenderer: Bitmap too large to be uploaded into a texture (3120x4160, max=4096x4096)
05-22 17:04:52.792 31976-32058/com.clock.study W/OpenGLRenderer: Bitmap too large to be uploaded into a texture (3120x4160, max=4096x4096)
05-22 17:04:52.796 1773-1877/? I/Timeline: Timeline: App_transition_stopped time:20604583
```

OpenGLRenderer: Bitmap too large to be uploaded into a texture

每次拍完照片，都会出现上面这样的 log，果然，因为图片太大而导致在 ImageView 上无法显示。到这里有童鞋要吐槽了，没对图片的采样率 **inSampleSize** 做处理？天地良心啊，绝对做处理了，直接看代码：

```
/**
 * 压缩Bitmap的大小
 *
 * @param imagePath 图片文件路径
 * @param requestWidth 压缩到想要的宽度
 * @param requestHeight 压缩到想要的高度
 * @return
 */
public static Bitmap decodeBitmapFromFile(String imagePath, int requestWidth, int req
 if (!TextUtils.isEmpty(imagePath)) {
 if (requestWidth <= 0 || requestHeight <= 0) {
 Bitmap bitmap = BitmapFactory.decodeFile(imagePath);
 return bitmap;
 }
 BitmapFactory.Options options = new BitmapFactory.Options();
 options.inJustDecodeBounds = true;//不加载图片到内存，仅获得图片宽高
 BitmapFactory.decodeFile(imagePath, options);
 options.inSampleSize = calculateInSampleSize(options, requestWidth, requestHei
 options.inJustDecodeBounds = false;
 return BitmapFactory.decodeFile(imagePath, options);
 } else {
 return null;
 }
}

public static int calculateInSampleSize(BitmapFactory.Options options, int reqWidth,
 final int height = options.outHeight;
 final int width = options.outWidth;
 int inSampleSize = 1;
 Log.i(TAG, "height: " + height);
 Log.i(TAG, "width: " + width);
 if (height > reqHeight || width > reqWidth) {
```

```
final int halfHeight = height / 2;
final int halfWidth = width / 2;

while ((halfHeight / inSampleSize) > reqHeight && (halfWidth / inSampleSize) > reqWidth)
 inSampleSize *= 2;
}

long totalPixels = width * height / inSampleSize;

final long totalReqPixelsCap = reqWidth * reqHeight * 2;

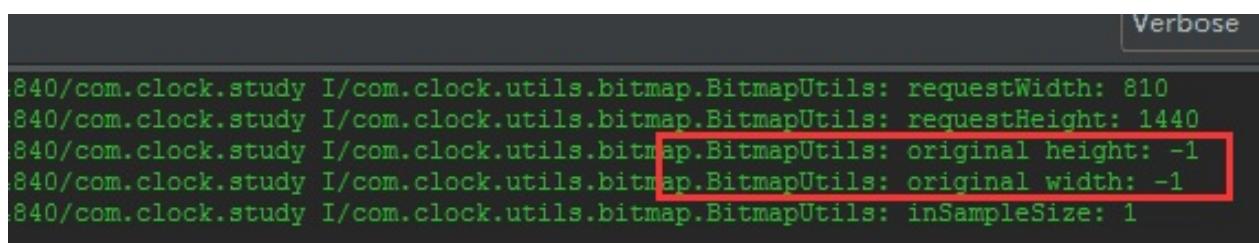
while (totalPixels > totalReqPixelsCap) {
 inSampleSize *= 2;
 totalPixels /= 2;
}
}

return inSampleSize;
}
```

瞄了代码后，是不是觉得没有问题了？没错，`inSampleSize` 确确实实经过处理，那为什么图片还是太大而显示不出来呢？`requestWidth`、`int requestHeight` 设置得太大导致`inSampleSize` 太小了？不可能啊，我都试着把长宽都设置成 100 了还是没法显示！干脆，直接打印`inSampleSize` 值，一打印，`inSampleSize` 值居然为 1。我去，彻底打脸了，明明说好的处理过了，居然还是 1！！！！！为了一探究竟，干脆加 log。

```
public static Bitmap decodeBitmapFromFile(String imagePath, int requestWidth, int requestHeight) {
 if (!TextUtils.isEmpty(imagePath)) {
 Log.i(TAG, "requestWidth: " + requestWidth);
 Log.i(TAG, "requestHeight: " + requestHeight);
 if (requestWidth <= 0 || requestHeight <= 0) {
 Bitmap bitmap = BitmapFactory.decodeFile(imagePath);
 return bitmap;
 }
 BitmapFactory.Options options = new BitmapFactory.Options();
 options.inJustDecodeBounds = true;//不加载图片到内存，仅获得图片宽高
 BitmapFactory.decodeFile(imagePath, options);
 Log.i(TAG, "original height: " + options.outHeight);
 Log.i(TAG, "original width: " + options.outWidth);
 options.inSampleSize = calculateInSampleSize(options, requestWidth, requestHeight);
 Log.i(TAG, "inSampleSize: " + options.inSampleSize);
 options.inJustDecodeBounds = false;
 return BitmapFactory.decodeFile(imagePath, options);
 } else {
 return null;
 }
}
```

运行打印出来的日志如下：



图片原来的宽高居然都是 -1，真是奇葩了！难怪，`inSampleSize` 经过处理之后结果还是 1。狠狠的吐槽了之后，总是要回来解决问题的。那么，图片的宽高信息都丢失了，我去哪里找啊？像下面这样？

```

public static Bitmap decodeBitmapFromFile(String imagePath,
 int requestWidth, int requestHeight) {
 ...
 BitmapFactory.Options options = new BitmapFactory.Options();
 options.inJustDecodeBounds = true; //不加载图片到内存，仅获得图片宽高
 Bitmap bitmap = BitmapFactory.decodeFile(imagePath, options);
 bitmap.getWidth();
 bitmap.getHeight();
 ...
} else {
 return null;
}
}

```

no, 此方案行不通, `inJustDecodeBounds = true` 时, `BitmapFactory` 获得 `Bitmap` 对象是 `null`; 那要怎样才能获图片的宽高呢? 前面提到的 `ExifInterface` 再次帮了我们大忙, 通过它的下面两个属性即可拿到图片真正的宽高

|                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| AudioFormat.Builder<br>AudioManager<br>AudioRecord<br>AudioRecord.Builder<br>AudioTimestamp<br>AudioTrack<br>AudioTrack.Builder<br>CamcorderProfile<br>CameraProfile<br><b>ExifInterface</b><br>FaceDetector<br>FaceDetector.Face<br>Image<br>Image.Plane<br>ImageReader<br>ImageWriter<br>JetPlayer<br>MediaActionSound<br>MediaCodec<br>MediaCodec.BufferInfo | <b>TAG_IMAGE_LENGTH</b><br><code>String TAG_IMAGE_LENGTH</code><br>Type is int.<br>Constant Value: "ImageLength" |
|                                                                                                                                                                                                                                                                                                                                                                 | <b>TAG_IMAGE_WIDTH</b><br><code>String TAG_IMAGE_WIDTH</code><br>Type is int.<br>Constant Value: "ImageWidth"    |

顺手吐槽一下, 为什么高不是 `TAG_IMAGE_HEIGHT` 而是 `TAG_IMAGE_LENGTH`。改良过后的代码实现如下:

```
public static Bitmap decodeBitmapFromFile(String imagePath,
 int requestWidth, int requestHeight) {
 if (!TextUtils.isEmpty(imagePath)) {
 Log.i(TAG, "requestWidth: " + requestWidth);
 Log.i(TAG, "requestHeight: " + requestHeight);
 if (requestWidth <= 0 || requestHeight <= 0) {
 Bitmap bitmap = BitmapFactory.decodeFile(imagePath);
 return bitmap;
 }
 BitmapFactory.Options options = new BitmapFactory.Options();
 options.inJustDecodeBounds = true;//不加载图片到内存，仅获得图片宽高
 BitmapFactory.decodeFile(imagePath, options);
 Log.i(TAG, "original height: " + options.outHeight);
 Log.i(TAG, "original width: " + options.outWidth);
 if (options.outHeight == -1 || options.outWidth == -1) {
 try {
 ExifInterface exifInterface = new ExifInterface(imagePath);
 int height = exifInterface.getAttributeInt(
 ExifInterface.TAG_IMAGE_LENGTH,
 ExifInterface.ORIENTATION_NORMAL);//获取图片的高度
 int width = exifInterface.getAttributeInt(
 ExifInterface.TAG_IMAGE_WIDTH,
 ExifInterface.ORIENTATION_NORMAL);//获取图片的宽度
 Log.i(TAG, "exif height: " + height);
 Log.i(TAG, "exif width: " + width);
 options.outWidth = width;
 options.outHeight = height;
 } catch (IOException e) {
 e.printStackTrace();
 }
 }
 options.inSampleSize = calculateInSampleSize(options,
 requestWidth, requestHeight); //计算获取新的采样率
 Log.i(TAG, "inSampleSize: " + options.inSampleSize);
 options.inJustDecodeBounds = false;
 return BitmapFactory.decodeFile(imagePath, options);
 } else {
 return null;
 }
}
```

再看一下，打印出来的log

```

clock.study I/com.clock.utils.bitmap.BitmapUtils: requestWidth: 810
clock.study I/com.clock.utils.bitmap.BitmapUtils: requestHeight: 1440
clock.study I/com.clock.utils.bitmap.BitmapUtils: original height: -1
clock.study I/com.clock.utils.bitmap.BitmapUtils: original width: -1
clock.study I/com.clock.utils.bitmap.BitmapUtils: exif height: 4160
clock.study I/com.clock.utils.bitmap.BitmapUtils: exif width: 3120
clock.study I/com.clock.utils.bitmap.BitmapUtils: inSampleSize: 8

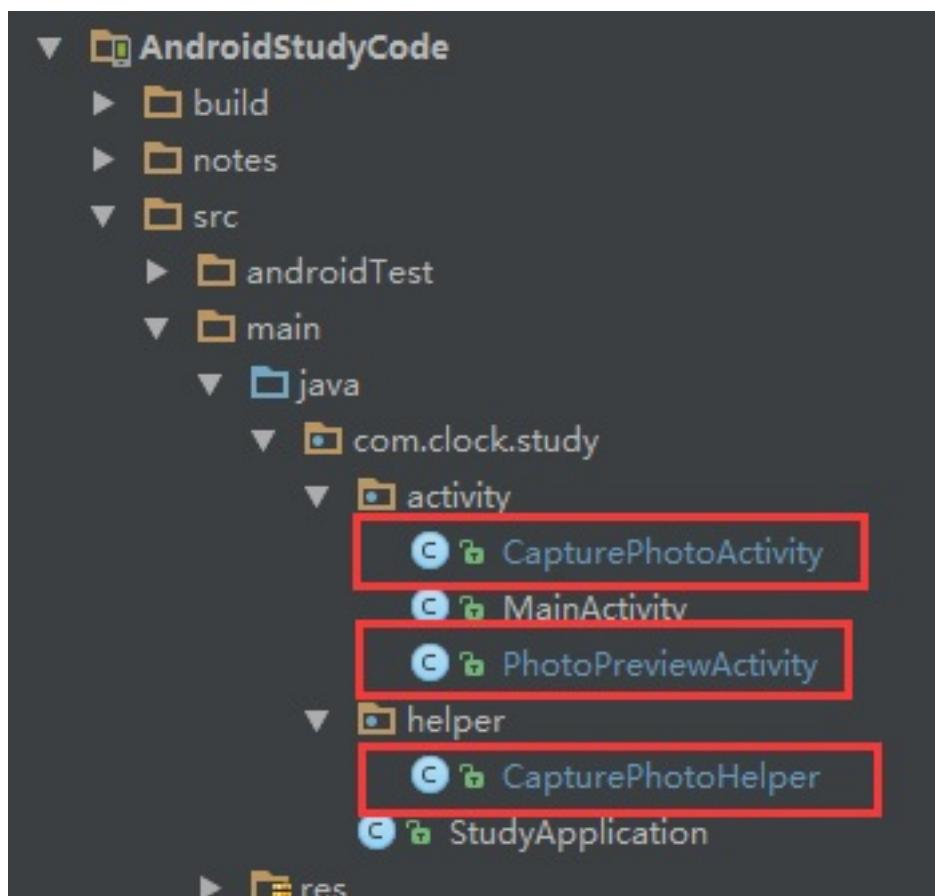
```

这样就可以解决问题啦。

## 总结

以上总结了这么些身边童鞋经常问起，但网上又不多见的适配问题，希望可以帮到一些开发童鞋少走弯路。文中多次提到小米的机子，并不代表只有MIUI上有这样的问题存在，仅仅只是因为我身边带的几部机子大都是小米的。对待适配问题，在搜索引擎都无法提供多少有效的信息时，我们只能靠断点、打log、观察控制台的日志、以及API文档来寻找一些蛛丝马迹作为突破口，相信办法总比困难多。

以上的示例代码已经整理到：<https://github.com/D-clock/AndroidStudyCode>，主要的代码在下面红圈部分



## 打个广告

最近 [diycode 社区](#) 上，越来越多开发小伙伴活跃在上面。看着有意思的提问，有意思的回答，我的感触还是蛮大的，曾经渴望过有这样一个优雅问答的技术社区存在，目前在一点点的实现！希望能够它能够越做越好！喜欢开发的小伙伴也可以一起上来 <http://diycode.cc/> 探讨交流！

# 如何检测程序是否获得了某项权限

来源:[如何检测程序是否获得了某项权限](#)

[start141](#) commented on 21 May 2015

众所周知，MIUI、Flyme等Android定制系统在程序请求打开相机、录音等操作时，系统会先弹出对话框提示用户是否允许程序执行这些操作，如果用户选择允许，则接下来的操作一切正常，如果用户选择拒绝之后，程序后面的工作将无法正常允许，甚至崩溃。

提问：在这种情况下，如何判断用户是拒绝还是允许了操作？目前我知道用try catch可以粗暴的判断，但我不确定所有的权限请求被拒绝后都会抛异常。

各位有更好的办法吗？

[t12x3456](#) commented on 21 May 2015

PackageManager有个checkPermission(String permName, String pkgName)方法可以判断.返回PackageManager.PERMISSION\_GRANTED即为已经授权,PackageManager.PERMISSION\_DENIED为被拒绝.当然这里要注意一点,这个api没办法在所有机型上通用,某些系统上把特定权限的名字已经修改掉了.另外定位权限比较特殊,没办法同步获取是否已授权.[@start141](#)

[start141](#) commented on 21 May 2015

checkPermission测试的是manifest有没有注册某项权限吧，并不能判断用户是否允许了某项权限。

[t12x3456](#) commented on 21 May 2015

**@start141**,`checkPermission`就是用来判断用户是否授予权限的，比如是否开启获取联系人，录音的权限，你可以看下api,并且检验下,这个是具体的api解释：

```
public abstract int checkPermission (String permName, String pkgName)
```

Added in API level 1

Check whether a particular package has been granted a particular permission.

#### Parameters

*permName* The name of the permission you are checking for,

*pkgName* The name of the package you are checking against.

#### Returns

If the package has the permission, `PERMISSION_GRANTED` is returned. If it does not have the permission, `PERMISSION_DENIED` is returned.

#### See Also

`PERMISSION_GRANTED`

`PERMISSION_DENIED`

[start141](#) commented on 21 May 2015

api的注释我的理解是指menifest注册了权限就返回`PERMISSION_GRANTED`，否则就返回`PERMISSION_DENIED`。也可能是我理解错了。

不过重要的是这个方法我测试过，无效。

[newtonker](#) commented on 24 Aug 2015

[@start141](#) 楼主这个问题，最后有没有什么解决方案？

[start141](#) commented on 24 Aug 2015

[@newtonker](#) 我目前知道的只有try catch，我们的app里面目前是用try catch来判断的。

[newtonker](#) commented on 24 Aug 2015 • edited

[@start141](#) 本来我也用的try catch，在其他手机上（华为，小米，魅族低版本）都是好的。但是在测试MX5 Android 5.1的系统时，在打开相机的那一刻，如果在弹出的权限对话框中拒绝了权限，会直接闪退。报的闪退居然是`RuntimeException`, `Method called after release()`。但是我表示都正常release了。

```
public void releaseCamera() {
 if (camera != null) {
 camera.setPreviewCallback(null);
 camera.stopPreview();
 camera.release();
 camera = null;
 }
}
```

[Qixingchen](#) commented on 24 Aug 2015

Method called after release() 的意思是，你在release后又进行了操作。可能是系统自动帮你release了，也可能是你的release函数在其他地方调用了，具体看报错栈才知道。

[@start141](#)

[start141](#) commented on 24 Aug 2015

这个问题可能是因为你在多个地方release()了，你可以在release()之后设置 camera=null，下次release()的时候先判断camera是不是等于null。  
还有可能是你try catch之后没有return后面的操作。

总之你看一次Log的错误行数吧，跟一下流程。[@Qixingchen](#) 已经说了原因。[@newtonker](#)

[newtonker](#) commented on 24 Aug 2015

[@Qixingchen](#), [@start141](#) 多谢你们两个人的帮助，我最终还是通过try catch解决了。MX 5 的相机权限和之前碰到的不一样。之前碰到华为，小米里弹出权限对话框时，如果禁止了相机权限，camera对象会返回null, MX 5 禁止了之后还是会返回一个相机对象，导致if(null != camera)方法判断无效了，所以会出现上面的问题。  
扩大了Exception的范围，捕获这个异常就行了。

[1207229280](#) commented on 12 Feb

我没做任何处理的话也不会出现异常 所以没办法catch 日志里只说明app不允许跳转

[1207229280](#) commented on 12 Feb

照相机权限

[dushu9247](#) commented on 10 Mar

[@newtonker](#) 多谢，魅族确实给开发带来了不少麻烦。。

I123456789jy commented on 14 Mar

手机里面不是有个应用权限列表吗？看下framework层的那个列表的权限状态，如何获取，能获取到这个我觉得就可以了！

Labmem003 commented on 12 May

各位，除了try catch 外有新的进展吗？我在尝试发短信，需要检查 android.permission.SEND\_SMS权限，但是发短信这个调用连异常都不会抛出。。。

imhet commented on 30 May

@Labmem003 小米手机上拒绝发送短信的权限异常确实没有抛出，可以加一个超时（比如3秒）没有返回短信是否发送成功的消息则判断发送短信权限被拒绝。

xilost commented on 15 Jun

@newtonker 你好，我也遇到了和你一样的问题，同样是魅族手机。你说的“扩大了Exception的范围，捕获这个异常就行了。”，想问一下具体是扩大到什么地方呢。

dushu9247 commented on 15 Jun

@xilost 魅族拒绝相机权限时Camera.open(i)返回的不是null，我是这么干的

```
try {
 mCamera = Camera.open(i); // 打开当前选中的摄像头
 Camera.Parameters mParameters = mCamera.getParameters();
 mCamera.setParameters(mParameters);
 mCamera.startPreview(); // 开始预览
} catch (Exception e) {
 mCamera = null;
} finally {
 if (mCamera == null) {
 finish();
 }
}
```

xilost commented on 15 Jun

@newtonker 懂了！太感谢了~

jmbeizi commented on 17 Jun

@xilost java.lang.RuntimeException: Camera is being used after Camera.release() was called 为什么我的是抱这样的错误。同样只是魅族MX5 有这样的问题

[luffykou](#) commented on 21 Jun

上面提到的坑都踩过。。。各大android手机厂商到底是要闹哪样啊！！！

新问题：

设备：VIVO X5Pro

操作：打开摄像头、录像、录音，会分别弹出权限申请，拍照、摄像、录音

现象：如果用户禁止了，可以得到camera对象，不抛任何异常，

只是预览没有画面、录像文件没有画面、没有声音（猜想是没有拿到data）

问题：这怎么搞？怎么才能知道用户禁止了权限？

[xilost](#) commented on 21 Jun

[@jmbeizi](#) 我的就是报这个错，用上面那个哥们的办法判断就好了。[@luffykou](#) 你的问题应该是一样的。

[luffykou](#) commented on 21 Jun

[@xilost](#) 跟上面提到的那个问题一样？

[jmbeizi](#) commented on 21 Jun

[@xilost](#) [@luffykou](#) 我的问题已经处理了。解决办法也是来自楼上。主要是在入口那里判断的，而像QQ，微信也同样是在拍照入口那里的，并没有在预览页。

```
mCamera = Camera.open(i); //打开当前选中的摄像头（这里对于大多数的设备是可以的）
Camera.Parameters mParameters = mCamera.getParameters();
mCamera.setParameters(mParameters); //这里是对于魅族MX5 处理的，其他的设备未知
mCamera.startPreview(); //开始预览（这里其实就可以不用了）
```

对于设备厂商的乱搞 真的是恶心透了

[jmbeizi](#) commented on 21 Jun

我把我的问题描述和解决方案放到博客里了

[http://blog.csdn.net/jm\\_beizi/article/details/51728495](http://blog.csdn.net/jm_beizi/article/details/51728495)

[CaptainJno](#) commented 28 days ago

我看了源码发现，Android内部其实也是通过捕获异常来判断摄像头权限是否开启的。大家可以看看 `android.hardware.Camera` 这个类，在第490行。

[drakeet](#) commented 28 days ago

魅族不按套路出牌，相机权限这里也是被坑过。



# 那些值得你去细细研究的Drawable适配 (上、下)

来源:[那些值得你去细细研究的Drawable适配-上](#)、[那些值得你去细细研究的Drawable适配-下](#)

郭霖

微信号: guolin\_blog

功能介绍:Android技术分享平台，在这里不仅可以学到各种Android相关的最新技术，还可以将你自己的技术总结分享给其他人，每周定期更新。

## 那些值得你去细细研究的Drawable适配(上)

话说前段时间我写了一篇drawable微技术的文章，也是引起了不小的反响，因为讲的东西是大家天天在使用，但却偏偏却没有深入了解的知识。而本篇文章的投稿者 王月半子 表示“不服”，看完我写的那篇之后自己也忍不住写了一篇，我阅读之后发现写的挺棒，于是邀请他来公众号上投稿。由于本篇文章内容较长，因此分为上下两篇发表。

王月半子的博客地址：[http://blog.csdn.net/wrg\\_20100512](http://blog.csdn.net/wrg_20100512)

Android适配的问题太多，有屏幕尺寸的适配、屏幕分辨率的适配以及不同系统版本的适配。反映在代码上，就是需要在资源文件上下功夫，主要是layout和drawable目录下的文件，这里主要研究一下drawable的适配。首先我们先熟悉一下基本的概念。

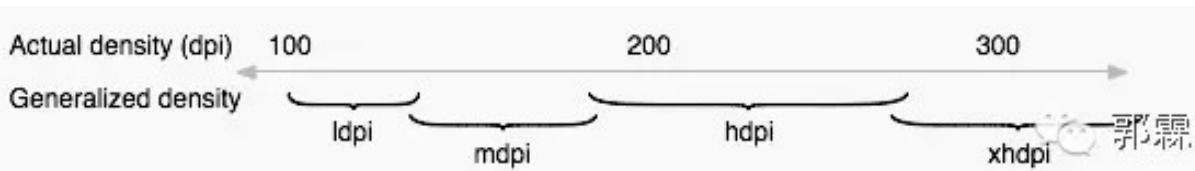
## Android中的长度单位

### px(pixel)

表示屏幕实际的像素。例如，1200×1920的屏幕在横向有1200个像素，在纵向有1920个像素。

### dpi(dot per inch)

表示屏幕密度是指每英寸上的像素点数。Android将根据不同的dpi将Android设备分成多个显示级别。具体如下：



正如drawable目录和mipmap目录有ldpi、mdpi、hdpi、xhdpi、xxhdpi之分。

- 这里解释一下mipmap和drawable的区别

在Android studio开发中，新建一个module的时候不同于Eclipse(会生成drawable、drawable-ldpi、drawable-mdpi、drawable-hdpi等等)，在资源文件中会生成mipmap-hdpi、mipmap-mdpi、mipmap-xhdpi、mipmap-xxhdpi和一个drawable目录。

对于这个问题谷歌官方有说法：

**drawable/**

For bitmap files (PNG, JPEG, or GIF), 9-Patch image files, and XML files that describe Drawable shapes or Drawable objects that contain multiple states (normal, pressed, or focused).

意思是说以drawable开头的目录存放的文件有png、jpeg、gif格式图片文件、.9图片以及一些XML文件。

**mipmap/**

For app launcher icons.

而以mipmap开头的目录存放的是App的图标。

## dp

也叫dip(density independent pixel)直译为密度无关的像素。我们猜测如果使用了这个单位，在不同屏幕密度的设备上显示的长度就会是相同的。那么在屏幕上显示图像时都是在屏幕上填充像素点，而使用这种与密度无关的像素（我们在布局文件中使用的 dp/dip 就是与密度无关的像素）是如何转换成像素的呢？其实在Android中，将屏幕密度为160dpi的中密度设备屏幕作为基准屏幕，在这个屏幕中，1dp=1px。其他屏幕密度的设备按照比例换算，具体如下表：

| 密度     | ldpi  | mdpi    | hdpi    | xhdpi   | xxhdpi  | xxxhdpi |
|--------|-------|---------|---------|---------|---------|---------|
| dpi 范围 | 0–120 | 120–160 | 160–240 | 240–320 | 320–480 | 480–640 |
| 比例     | 0.75  | 1       | 1.5     | 2       | 3       | 4       |
| 整数比例   | 3     | 4       | 6       | 8       | 12      | 16      |

由上表不难计算1dp在hdpi设备下等于1.5px，同样的在xxhdpi设备下1dp=3px。这里我们从dp到px解释了Android中不同屏幕密度之间的像素比例关系。

下面换一个角度，从px到dp的变化来说明这种比例关系。

这里我们选择在以mipmap开头的目录中设计一个icon，要求icon在屏幕中占据相同的dp。那么对于不同的屏幕密度（MDPI、HDPI、XHDPI、XXHDPI 和 XXXHDPI）应按照2:3:4:6:8 的比例进行缩放。比如尺寸为48x48dp，这表示在 MDPI 的屏幕上其实际尺寸应为 48x48px，在 HDPI 的屏幕上其实际大小是 MDPI 的 1.5 倍 (72x72 px)，在 XDPI 的屏幕上其实际大小是 MDPI 的 2 倍 (96x96 px)，依此类推。

- 图片的描述有两种：
  - 1、仅仅通过宽高的像素。
  - 2、通过图片分辨率（不同于屏幕分辨率，单位英寸中所包含的像素点数）和尺寸大小。

## sp (scale-independent pixels)

与dp类似，但是可以在设置里面调节字号的时候，文字会随之改变。当系统字号设为“普通”时，sp与px的尺寸换算和dp与px是一样的。

# 屏幕尺寸、屏幕分辨率、屏幕密度

## 屏幕尺寸

设备的物理屏幕尺寸，指屏幕的对角线的长度，单位是英寸，1 inch = 2.54 cm。比如“5寸大屏手机”，就是指对角线的尺寸，5寸×2.54厘米/寸=12.7厘米。

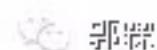
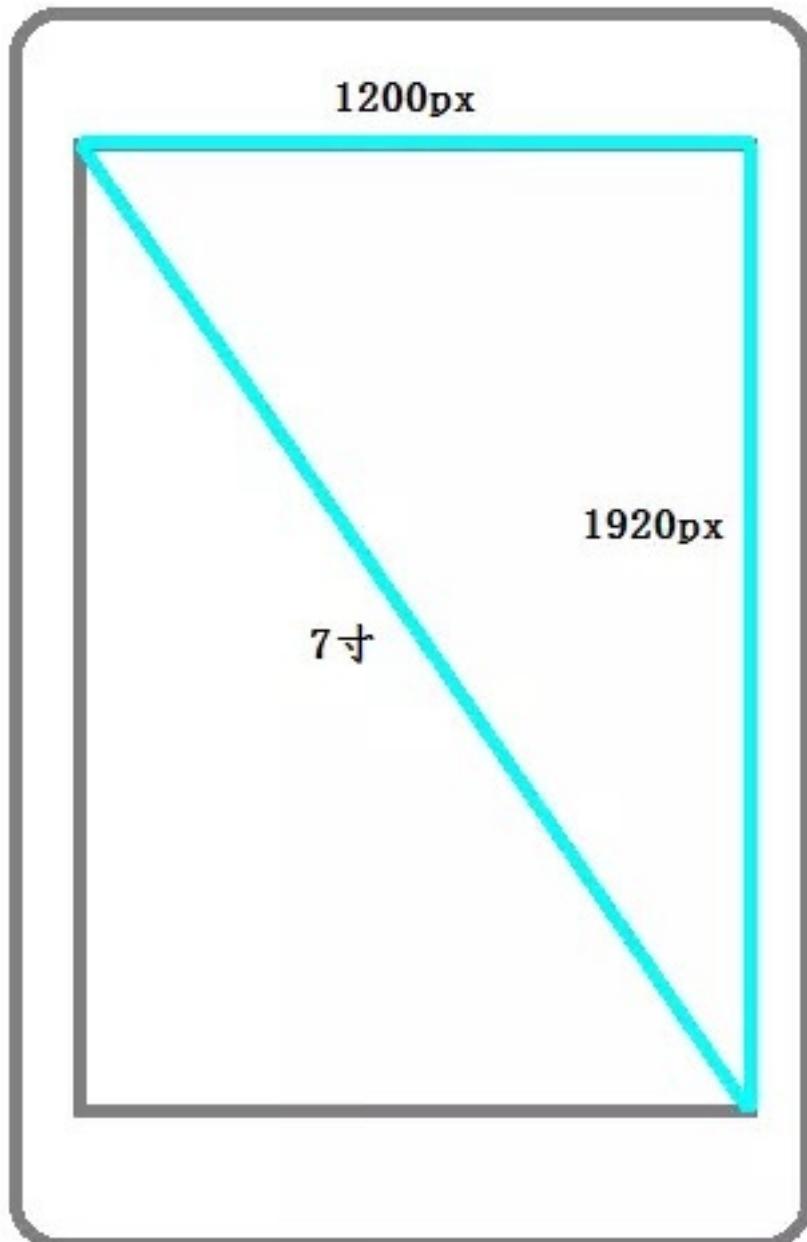
## 屏幕分辨率

也叫显示分辨率，是屏幕图像的精密度，是指屏幕或者显示器所能显示的像素点有多少。一般以横向像素×纵向像素表示分辨率，如 $1200\times 1920$ 表示此屏幕在宽度方向有1200个像素，在高度方向有1920个像素。

## 屏幕密度

是指每英寸上的像素点数，单位是dpi(dot per inch)或者ppi(pixels per inch)，数值越高显示越细腻。屏幕密度与屏幕尺寸和屏幕分辨率有关。例如在屏幕尺寸一定的条件下，屏幕分辨率越高屏幕密度越大，反之越小。同理在屏幕分辨率一定的条件下，屏幕尺寸越小屏幕密度越大，反之越小。

屏幕尺寸和屏幕分辨率，这两个值是可以直接得到的。屏幕密度需要我们计算得到。例如我的手机的分辨率是 $1200\times 1920$ ，屏幕尺寸是7寸的。根据屏幕尺寸、屏幕分辨率和屏幕密度定义不难看出他们之间的关系如下图：



根据勾股定理，我们得出对角线的像素数大约是2264，那么用2264除以7就是此屏幕的密度了，计算结果是323。

- 备注：上面的出现的0.75, 1, 1.5, 2, 3, 4才是屏幕密度(density)。而120, 160, 240, 320, 480, 640是屏幕密度dpi(densityDpi)。

## 实际密度与系统密度

实际密度就是我们自己算出来的密度，这个密度代表了屏幕真实的细腻程度，如上述例子中的323dpi就是实际密度，说明这块屏幕每寸有323个像素。7英寸 $1200\times 1920$ 的屏幕密度是323,5英寸 $1200\times 1920$ 的屏幕密度是452，而相同分辨率的4.5英寸屏幕密度是503。如此看来，屏幕密度将会出现很多数值，呈现严重的碎片化。而密度又是Android屏幕将界面进行缩放显示的依据，那么Android是如何适配这么多屏幕的呢？

其实，每部Android手机屏幕都有一个初始的固定密度，这些数值是120、160、240、320、480，这些就是Android为不同设备设定的系统密度。

得到实际密度以后，一般会选择一个最近的密度作为系统密度，系统密度是出厂预置的，如440dpi的系统密度就是和它最接近的480dpi；如果是330dpi的设备，它的系统密度就是320dpi。但是，现在很多手机不一定会选择这些值作为系统密度，而是选择实际的dpi作为系统密度，这就导致了很多手机的dpi也不是在这些值内。例如小米Note这样的xxhdpi的设备他的系统密度并不是480，而是它的实际密度440。

## 获取设备的上述属性

Android系统中有个DisplayMetrics的类，通过这个类就可以得到上述的所有属性。

```
DisplayMetrics displayMetrics = getResources().getDisplayMetrics();
float density = displayMetrics.density; //屏幕密度
int densityDpi = displayMetrics.densityDpi;//屏幕密度dpi
int heightPixels = displayMetrics.heightPixels;//屏幕高度的像素
int widthPixels = displayMetrics.widthPixels;//屏幕宽度的像素
String screenResolution = widthPixels + "X" + heightPixels;
float scaledDensity = displayMetrics.scaledDensity;//字体的放大系数
float xdpi = displayMetrics.xdpi;//宽度方向上的dpi
float ydpi = displayMetrics.ydpi;//高度方向上的dpi
```

其中xdpi = ydpi = densityDpi.打印结果如下：

```
8594-8594/? I/MainActivity: density = 2.0
8594-8594/? I/MainActivity: densityDpi = 320
8594-8594/? I/MainActivity: scaledDensity = 2.0
8594-8594/? I/MainActivity: Screen resolution = 1200×1920
8594-8594/? I/MainActivity: xdpi = 320.0
8594-8594/? I/MainActivity: ydpi = 320.0
```



上面计算的我的设备的dpi为323。这里系统给定的屏幕密度dpi为320。

OK，基础的东西介绍完了，那么回归正题：

## 1. android项目中那么多以drawable开头的文件夹，那应用的配图应该放在哪个文件夹之下呢？

一般的开发的话，都会做三套配图，对应着drawable-hdpi、drawable-xhdpi、drawable-xxhdpi三个文件夹。

### 2. 那要是做一套呢？应该做哪一套呢？

做一套也是可以的，放在drawable-xxhdpi文件夹中。

### 3. 为什么做一套配图要放在drawable-xxhdpi文件夹中？

这里先不给出答案，我在网上也看了一些说法，说是省内存，这里暂且不置可否。

明天我们就从内存的角度去分析问题3，使用同一张图片，放置在不同的drawable文件夹，在同一设备上运行，研究一下图片大小及内存的占用。

# 那些值得你去细细研究的Drawable适配(下)

本篇文章由 王月半子 投稿，继续承接上篇带你研究Drawable的适配问题。相比于上篇，下篇文章更加面向于实战，内容也更加实用。

王月半子的博客地址：[http://blog.csdn.net/wrg\\_20100512](http://blog.csdn.net/wrg_20100512)

上篇中我们讲了基本概念，今天我们就进入实战对上篇的问题做出解答，相对于昨天的开胃菜，今天这篇你可能需要花费一番工夫阅读。

首先我准备一张600×960像素的png图片，大小为248k，放在不同分辨率的drawable文件夹下，使用同一手机（1200X1920像素）测试。布局很简单，一个RelativeLayout里面包含一个ImageView， ImageView的宽高均为"wrap\_content"，Activity中关键的代码如下：

```
BitmapDrawable drawable = (BitmapDrawable) imageView.getDrawable();
if (drawable != null) {
 Bitmap bitmap = drawable.getBitmap();
 Log.i(TAG, "bitmap width = " + bitmap.getWidth()
 + " bitmap height = " + bitmap.getHeight());
 Log.i(TAG, "bitmap size = " + bitmap.getByteCount());//获取bitmap的占用内存
 Log.i(TAG, "imageView width = " + imageView.getWidth()
 + " imageView height = " + imageView.getHeight());
}
```

篇幅有限，这里我直接贴出部分结果以及汇总结果：

放在drawable-mdpi文件夹下：

```
9919-9919/? I/MainActivity: bitmap width = 1200 bitmap height = 1920
9919-9919/? I/MainActivity: bitmap size = 9216000
9919-9919/? I/MainActivity: imageView width = 1200 imageView height = 1920
```



放在drawable-xxhdpi文件夹下：

```
10120-10120/? I/MainActivity: bitmap width = 400 bitmap height = 640
10120-10120/? I/MainActivity: bitmap size = 1024000
10120-10120/? I/MainActivity: imageView width = 400 imageView height = 640
```



汇总结果（原图600×960像素）：

| 密度           | ldpi      | mdpi      | hdpi     | xhdpi   | xxhdpi  |
|--------------|-----------|-----------|----------|---------|---------|
| 位图大小         | 1600×2500 | 1200×1920 | 800×1280 | 600×960 | 400×640 |
| 占用内存         | 15.63M    | 8.79M     | 3.90M    | 2.20M   | 0.98M   |
| ImageView 大小 | 1200×1920 | 1200×1920 | 800×1280 | 600×960 | 400×640 |

接下来我们来分析两个问题：

## 同一张图片，放在不同目录下，生成了不同大小的Bitmap

要想回答这个问题，必须要深入理解Android系统加载drawable目录下图片的过程，用的是decodeResource方法：

```
final TypedValue value = new TypedValue();
is = res.openRawResource(id, value);
bm = decodeResourceStream(res, value, is, null, opts);
```

该方法本质上就两步：

- 1. 读取原始资源，这个调用了 Resource.openRawResource 方法，这个方法调用完成之后会对 TypedValue 进行赋值，其中包含了原始资源的 density 等信息；原始资源的 density 其实取决于资源存放的目录（比如 drawable-xxhdpi 对应的是480，drawable-hdpi对应的就是240，而drawable目录对应的是 TypedValue.DENSITY\_DEFAULT=0）。

- 2.调用 decodeResourceStream 对原始资源进行解码和适配。这个过程实际上就是原始资源的 density 到屏幕 density 的一个映射，代码如下：

```
public static Bitmap decodeResourceStream(Resources res, TypedValue value,
 InputStream is, Rect pad, Options opts) {
 if (opts == null) {
 opts = new Options();
 }
 if (opts.inDensity == 0 && value != null) {
 final int density = value.density;
 if (density == TypedValue.DENSITY_DEFAULT) {
 opts.inDensity = DisplayMetrics.DENSITY_DEFAULT;
 } else if (density != TypedValue.DENSITY_NONE) {
 opts.inDensity = density;
 }
 }
 if (opts.inTargetDensity == 0 && res != null) {
 opts.inTargetDensity = res.getDisplayMetrics().densityDpi;
 }
 return decodeStream(is, pad, opts);
}
```

该方法主要就是对opts对象中的属性进行赋值，代码不难理解。如果 **value.density=DisplayMetrics.DENSITY\_DEFAULT**也就是0的话，将 **opts.inDensity** 赋值为 **DisplayMetrics.DENSITY\_DEFAULT**默认值为160.否则就将 **opts.inDensity** 赋值为第一步获取到的值。此外将 **opts.inTargetDensity** 赋值为屏幕密度Dpi。**inDensity** 和 **inTargetDensity**要特别注意，这两个值与下面 cpp 文件里面的 **density** 和 **targetDensity** 相对应。

BitmapFactory.cpp:

```

static jobject doDecode(JNIEnv* env, SkStreamRewindable* stream,
 jobject padding, jobject options) {

 if (env->GetBooleanField(options, gOptions_scaledFieldID)) {
 //通过JNI获取opts.inDensity的值
 const int density = env->GetIntField(options, gOptions_densityFieldID);
 //通过JNI获取opts.inTargetDensity的值
 const int targetDensity = env->GetIntField(options, gOptions_targetDensityFieldID);
 if (density != 0 && targetDensity != 0 && density != screenDensity) {
 scale = (float) targetDensity / density;//求出缩放的倍数。
 }
 }
}

const bool willScale = scale != 1.0f;
.....
SkBitmap decodingBitmap;
if (!decoder->decode(stream, &decodingBitmap, prefColorType, decodeMode)) {
 return nullObjectReturn("decoder->decode returned false");
}
//这里这个deodingBitmap就是解码出来的bitmap，大小是图片原始的大小
int scaledWidth = decodingBitmap.width();
int scaledHeight = decodingBitmap.height();
if (willScale && decodeMode != SkImageDecoder::kDecodeBounds_Mode) {
 scaledWidth = int(scaledWidth * scale + 0.5f);//缩放后的宽
 scaledHeight = int(scaledHeight * scale + 0.5f); //缩放后的高
}
if (willScale) {
 const float sx = scaledWidth / float(decodingBitmap.width()); //宽的缩放倍数
 const float sy = scaledHeight / float(decodingBitmap.height()); //高的缩放倍数

 SkPaint paint;
 SkCanvas canvas(*outputBitmap);
 canvas.scale(sx, sy); //缩放画布
 canvas.drawBitmap(decodingBitmap, 0.0f, 0.0f, &paint); //画出图像
}

```

代码中的density 和 targetDensity 均是通过JNI获取的值，前者是 opts.inDensity, targetDensity 实际上是 opts.inTargetDensity 也就是 DisplayMetrics 的 densityDpi, 我的手机的densityDpi在上面已经打印过了320。最终我们看到 Canvas 放大了 scale 倍，然后又把读到内存的这张 bitmap 画上去，相当于把这张 bitmap 放大了 scale 倍。

Android中一张图片（BitMap）占用的内存主要和以下几个因数有关：图片长度，图片宽度，单位像素占用的字节数。这里我们需要知道bitmap中单位像素占据的内存大小，而单位像素占据的内存大小是与.Options的inPreferredConfig有关的：

```
public Bitmap.Config inPreferredConfig = Bitmap.Config.ARGB_8888;
```

inPreferredConfig的类型为Bitmap.Config默认值为Bitmap.Config.ARGB\_8888。ARGB指的是一种色彩模式，里面A代表Alpha，R表示red，G表示green，B表示blue。ARGB\_8888代表的就是这四个通道各占8位也就是一个字节，合起来就是4个字节。同理Bitmap.Config中还有ARGB\_4444、ALPHA\_8、RGB\_565，他们占用内存的大小和ARGB\_8888一样。

我们再来分析一下上面的那汇总表：

已知图片的大小为600×960，格式为png，测试手机的densityDpi为320（`opts.inTargetDensity=320`）。

当图片放在drawable-mdpi目录时，此时得到的`opts.inDensity=160`，那么放大的倍数就是 $320/160=2$ ，放大后图片的大小就是1200×1920，占用的内存就是：

$1200\times1920\times4=9216000\text{B}$ ,  $9216000\div1024\div1024\approx8.79\text{M}$ 。同样的当图片放在drawable-xxhdpi目录下时，此时得到的`opts.inDensity=480`，那么放大的倍数就是 $320/480=2/3$ ，放大后图片的大小就是400×640，占用的内存就是： $400\times640\times4=1024000\text{B}$ ,  $1024000\div1024\div1024\approx0.98\text{M}$ 。其他的类似，这里就不再赘述。至此问题分析完毕。

## 如果只做一套配图,为什么要放在**drawable-xxhdpi**文件夹中?

由汇总图可看出，同一张图片，放在不同的drawable目录下（从drawable-lpdi到drawable-xxhpdi）在同一手机上占用的内存越来越小。

因为同一部手机，它的densityDpi是固定的，而不同的drawable目录对应的原始密度不同，并且从drawable-lpdi到drawable-xxhpdi原始密度越来越大，而图片的放大倍数= $\text{densityDpi}\div\text{原始密度}$ ，所以放大倍数变小，自然占用的内存小了。

图片占用的内存主要和以下几个因数有关：

- 1.图片在内存中的像素。
- 2.单位像素占用的字节数。

而对于同一应用来说，单位像素占用的字节数一定是相同的，那么图片占用的内存只与图片在内存中的像素有关了，而图片在内存中的像素又由图片的原始像素和图片在内存中放大的倍数（ $\text{scale} = \text{densityDpi}\div\text{原始密度}$ ）。

综上所述，图片占用的内存和以下几个因素有关：

- 图片的原始像素
- 设备的densityDp
- 图片在哪个drawable目录下

设想一下，做一套适配xhdpi设备的配图，放在drawable-xhdpi目录下，和做一套适配mdpi设备的配图，放在drawable-mdpi目录下，这时候我用一个hdpi的设备来测试这两种方案，哪种方案更省内存呢？

咱们具体分析一下，就拿我的设备来说是xhdpi的，分辨率为 $1200 \times 1920$ ，仍旧用原来的测试代码，要让ImageView显示为全屏，并且图片放在drawable-xhdpi目录下，那么图片的原始像素就应该是 $1200 \times 1920$ 。对于相同尺寸的mdpi来说，他的分辨率是 $600 \times 960$ ，要让ImageView显示为全屏，并且图片放在drawable-mdpi目录下，那么图片的原始像素就应该是 $600 \times 960$ 。

这时候使用hdpi的设备来测试方案一，可以得到：图片原始的像素为 $1200 \times 1920$ ，设备的densityDpi为240，原始的dpi为320。所以放大倍数为 $2/3$ ，最终图片在内存中的大小为 $800 \times 1280$ 。

使用hdpi的设备来测试方案二，可以得到：图片原始的像素为 $600 \times 960$ ，设备的densityDpi为240，原始的dpi为160。所以放大倍数为 $3/2$ ，最终图片在内存中的大小也为 $800 \times 1280$ 。

当然在其他不同dpi的设备上这两种方案占用的内存也是一样的，这里就不再赘述。

所以如果只做一套图的话，不考虑app包的大小以及app内部配图的清晰度来说，只要图片所处的drawable目录适配该目录对应着dpi设备(例如做一套适配mdpi设备的图，将这些配图放在drawable-mdpi目录下)，再通过android系统加载图片的缩放机制后，不论哪种方案，对于同种dpi的设备，图片所占的内存是相同的。

但是这些都是不考虑app包的大小以及app内部配图的清晰度来说的，事实上不可能不考虑这些因素。在考虑这些因素之前，先说一下图片缩放的问题。

图片的像素、分辨率以及尺寸满足如下关系：

$$\text{分辨率} = \text{像素} \div \text{尺寸}$$

图片在进行缩放的时候，分辨率是不变的，变化的是像素和尺寸。比如将图片放大两倍，这时就会有白色像素插值补充原有的像素中，所以就会看起来模糊，清晰度不高。而缩小图片，会通过对图片像素采样生成缩略图，将会增强它的平滑度和清晰度。

如果制作适配xxhdpi设备的图片，同时放在drawable-xxhdpi目录中，其他除了xxxhdpi的设备在显示配图时都是缩小原图，不会出现模糊的情况。而这样的话App打包由于配图的质量高，自然App会相对大些。

相比较App的大小和用户体验来说，毫无疑问，用户至上。所以如果只有一套配图的话，制作高清大图适配xxhdpi的设备，将配图放置在drawable-xxhdpi目录下就可以了。

另外对于xxxhdpi来说，市场上的设备不多，没必要为了那么一点点特殊群体来加大app的容量（ $4 \div 3 \approx 1.3$ 倍，容量放大的倍数不小呀！！！）

至此，所有的问题都分析完毕，感谢您的阅读。

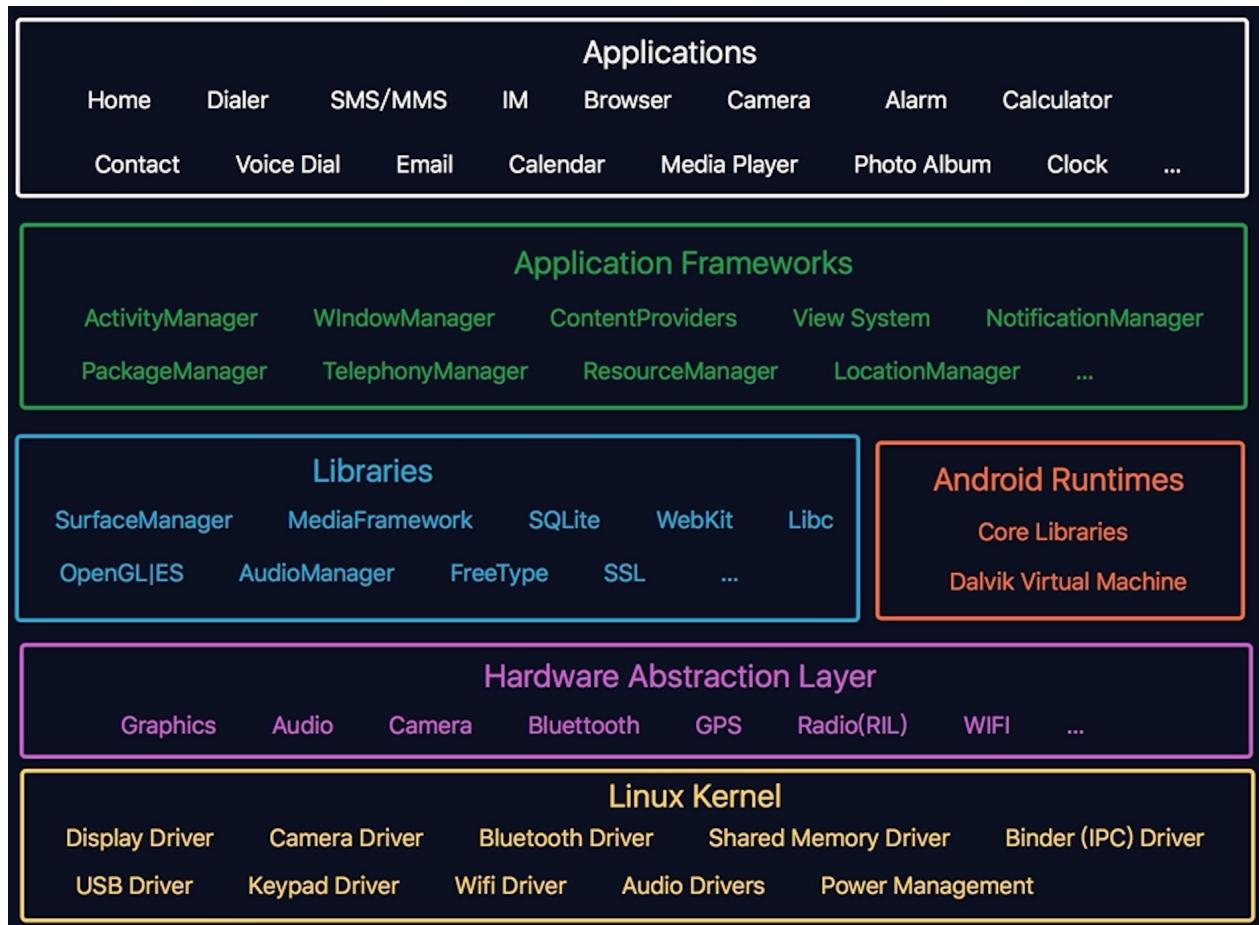
如果你有好的技术文章想和大家分享，欢迎向我的公众号投稿，投稿具体细节请在公众号主页点击“投稿”菜单查看。

欢迎长按下图 -> 识别图中二维码或者扫一扫关注我的公众号：



# 内核

# Android 体系架构图



## 内核层

Android底层是基于Linux操作系统的。严格意义上讲是Linux操作系统的一个变种。为什么使用Linux作为内核？

- 避开了与硬件直接打交道
  - Linux在与硬件打交道方面是强项，而且Linux开源
- 基于Linux的驱动开发可扩展性很强

## 硬件抽象层

HAL:Hardware Abstraction Layer,通过定义硬件驱动接口来进一步降低Android系统与硬件的耦合度。

- User space C/C++ library layer
- Defines the interface that Android requires hardware "drivers" to implement
- Separates the Android platform logic from the hardware interface

## 系统运行库

这一层包含支撑整个系统正常运行的基础库。由于使用C或者C++实现的，所以被称为C库

## 应用程序框架层

一般使用Java实现，所以也称为java库

## 应用层

系统应用和普通应用

# Android子线程真的不能更新UI么

来源:[www.cnblogs.com](http://www.cnblogs.com)

Android单线程模型是这样描述的：

Android UI操作并不是线程安全的，并且这些操作必须在UI线程执行

如果在其它线程访问UI线程，Android提供了以下的方式：

- Activity.runOnUiThread(Runnable)
- View.post(Runnable)
- View.postDelayed(Runnable, long)
- Handler

为什么呢？在子线程中就不能操作UI么？

当一个程序第一次启动的时候，Android会同时启动一个对应的主线程，这个主线程就是UI线程，也就是ActivityThread。UI线程主要负责处理与UI相关的事件，如用户的按键点击、用户触摸屏幕以及屏幕绘图等。系统不会为每个组件单独创建一个线程，在同一个进程里的UI组件都会在UI线程里实例化，系统对每一个组件的调用都从UI线程分发出去。所以，响应系统回调的方法永远都是在UI线程里运行，如响应用户动作的onKeyDown()的回调。

那为什么选择一个主线程干这些活呢？换个说法，Android为什么使用单线程模型，它有什么好处？

先让我们看下单线程化的事件队列模型是怎么定义的：

采用一个专门的线程从队列中抽取事件，并把他们转发给应用程序定义的事件处理器

这看起来就是Android的消息队列、Looper和Handler嘛。类似知识请参考：[深入理解Message, MessageQueue, Handler和Looper](#)

其实现代GUI框架就是使用了类似这样的模型：模型创建一个专门的线程，事件派发线程来处理GUI事件。单线程化也不单单存在Android中，Qt、XWindows等都是单线程化。当然，也有人试图用多线程的GUI，最终由于竞争条件和死锁导致的稳定性问题等，又回到单线程化的事件队列模型老路上来。单线程化的GUI框架通过限制来达到线程安全：所有GUI中的对象，包括可视组件和数据模型，都只能被事件线程访问。

这就解释了Android为什么使用单线程模型。

那Android的UI操作并不是线程安全的又是怎么回事？

Android实现View更新有两组方法，分别是invalidate和postInvalidate。前者在UI线程中使用，后者在非UI线程中使用。换句话说，Android的UI操作不是线程安全可以表述为invalidate在子线程中调用会导致线程不安全。作一个假设，现在我用invalidate在子线程中刷新界面，同时UI线程也在用invalidate刷新界面，这样会不会导致界面的刷新不能同步？既然刷新不同步，那么invalidate就不能在子线程中使用。这就是invalidate不能在子线程中使用的原因。

postInvalidate可以在子线程中使用，它是怎么做到的？

看看源码是怎么实现的：

```
public void postInvalidate() {
 postInvalidateDelayed(0);
}

public void postInvalidateDelayed(long delayMilliseconds) {
 // We try only with the AttachInfo because there's no point in invalidating
 // if we are not attached to our window
 if (mAttachInfo != null) {
 Message msg = Message.obtain();
 msg.what = AttachInfo.INVALIDATE_MSG;
 msg.obj = this;
 mAttachInfo.mHandler.sendMessageDelayed(msg, delayMilliseconds);
 }
}
```

说到底还是通过Handler的sendMessageDelayed啊，还是逃不过消息队列，最终还是交给UI线程处理。所以View的更新只能由UI线程处理。

如果我非要在子线程中更新UI，那会出现什么情况呢？

```
android.view.ViewRoot$CalledFromWrongThreadException: Only the original thread that c
```

抛了一个 CalledFromWrongThreadException 异常。

相信很多人遇到这个异常后，就会通过前面的四种方式中的其中一种解决：

```
* Activity.runOnUiThread(Runnable)
* View.post(Runnable)
* View.postDelayed(Runnable, long)
* Handler
```

说到底还没触发到根本，为什么会出现这个异常呢？这个异常在哪里抛出来的呢？

```
void checkThread() {
 if (mThread != Thread.currentThread()) {
 throw new CalledFromWrongThreadException(
 "Only the original thread that created a view hierarchy can touch its
 }
}
```

该代码出自 `framework/base/core/java/android/view/ViewRootImpl.java`

再看下`ViewRootImpl`的构造函数，`mThread`就是在这初始化的：

```
public ViewRootImpl(Context context, Display display) {
 mContext = context;
 mWindowSession = WindowManagerGlobal.getWindowSession();
 mDisplay = display;
 mBasePackageName = context.getBasePackageName();

 mDisplayAdjustments = display.getDisplayAdjustments();

 mThread = Thread.currentThread();

}
```

再研究一下这个`CalledFromWrongThreadException`异常的堆栈，会发现最后到了`invalidateChild`和`invalidateChildInParent`方法中：

```
@Override
public void invalidateChild(View child, Rect dirty) {
 invalidateChildInParent(null, dirty);
}

@Override
public ViewParent invalidateChildInParent(int[] location, Rect dirty) {
 checkThread();

}
```

最终通过`checkThread`形成了这个异常。说到底，非UI线程是可以刷新UI的呀，前提是它要拥有自己的`ViewRoot`。如果想直接创建`ViewRoot`实例，你会发现找不到这个类。那怎么做呢？通过`WindowManager`。

```

class NonUiThread extends Thread{
 @Override
 public void run() {
 Looper.prepare();
 TextView tx = new TextView(MainActivity.this);
 tx.setText("non-UiThread update textview");

 WindowManager windowManager = MainActivity.this.getWindowManager();
 WindowManager.LayoutParams params = new WindowManager.LayoutParams(
 200, 200, 200, 200, WindowManager.LayoutParams.FLAG_FULLSCREEN |
 WindowManager.LayoutParams.TYPE_TOAST, PixelFormat.OPAQUE);
 windowManager.addView(tx, params);
 Looper.loop();
 }
}

```

就是通过windowManager.addView创建了ViewRoot, WindowManagerImpl.java中的addView方法:

```

@Override
public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params) {
 applyDefaultToken(params);
 mGlobal.addView(view, params, mDisplay, mParentWindow);
}

```

```
private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();
```

mGlobal是一个WindowManagerGlobal实例, 代码在frameworks/base/core/java/android/view/WindowManagerGlobal.java中, 具体实现如下:

```

= 0; --i) {
 mRoots.get(i).loadSystemProperties();
 }
}
};

SystemProperties.addChangeListener(mSystemPropertyUpdater);
}

int index = findViewLocked(view, false);
if (index >= 0) {
 if (mDyingViews.contains(view)) {
 // Don't wait for MSG_DIE to make it's way through root's queue.
 mRoots.get(index).doDie();
 }
}

```

```

 } else {
 throw new IllegalStateException("View " + view
 + " has already been added to the window manager.");
 }
 // The previous removeView() had not completed executing. Now it has.
 }

 // If this is a panel window, then find the window it is being
 // attached to for future reference.
 if (wparams.type >= WindowManager.LayoutParams.FIRST_SUB_WINDOW &&
 wparams.type = 0) {
 removeViewLocked(index, true);
 }
}
throw e;
}

}" data-snippet-id="ext.cf3ddee221e8e32c9ad97dc73d590bf6" data-snippet-saved="false">
 Display display, Window parentWindow) {
 if (view == null) {
 throw new IllegalArgumentException("view must not be null");
 }
 if (display == null) {
 throw new IllegalArgumentException("display must not be null");
 }
 if (!(params instanceof WindowManager.LayoutParams)) {
 throw new IllegalArgumentException("Params must be WindowManager.LayoutParams");
 }

 final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams) params;
 if (parentWindow != null) {
 parentWindow.adjustLayoutParamsForSubWindow(wparams);
 } else {
 // If there's no parent, then hardware acceleration for this view is
 // set from the application's hardware acceleration setting.
 final Context context = view.getContext();
 if (context != null
 && (context.getApplicationInfo().flags
 & ApplicationInfo.FLAG_HARDWARE_ACCELERATED) != 0) {
 wparams.flags |= WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED
 }
 }

 ViewRootImpl root;
 View panelParentView = null;

 synchronized (mLock) {
 // Start watching for system property changes.
 if (mSystemPropertyUpdater == null) {
 mSystemPropertyUpdater = new Runnable() {
 @Override public void run() {
 synchronized (mLock) {
 for (int i = mRoots.size() - 1; i >= 0; --i) {

```

```

 mRoots.get(i).loadSystemProperties();
 }
}
}
};

SystemProperties.addChangeCallback(mSystemPropertyUpdater);
}

int index = findViewLocked(view, false);
if (index >= 0) {
 if (mDyingViews.contains(view)) {
 // Don't wait for MSG_DIE to make it's way through root's queue.
 mRoots.get(index).doDie();
 } else {
 throw new IllegalStateException("View " + view
 + " has already been added to the window manager.");
 }
 // The previous removeView() had not completed executing. Now it has.
}

// If this is a panel window, then find the window it is being
// attached to for future reference.
if (wparams.type >= WindowManager.LayoutParams.FIRST_SUB_WINDOW &&
 wparams.type <= WindowManager.LayoutParams.LAST_SUB_WINDOW) {
 final int count = mViews.size();
 for (int i = 0; i < count; i++) {
 if (mRoots.get(i).mWindow.asBinder() == wparams.token) {
 panelParentView = mViews.get(i);
 }
 }
}

root = new ViewRootImpl(view.getContext(), display);

view.setLayoutParams(wparams);

mViews.add(view);
mRoots.add(root);
mParams.add(wparams);
}

// do this last because it fires off messages to start doing things
try {
 root.setView(view, wparams, panelParentView);
} catch (RuntimeException e) {
 // BadTokenException or InvalidDisplayException, clean up.
 synchronized (mLock) {
 final int index = findViewLocked(view, false);
 if (index >= 0) {
 removeViewLocked(index, true);
 }
 }
}
}

```

```

 throw e;
 }
}

```

所以，非UI线程能更新UI，只要它有自己的ViewRoot。

延伸一下：Android Activity本身是在什么时候创建ViewRoot的呢？

既然是单线程模型，就要先找到这个UI线程实现类ActivityThread，看里面哪里addView了。没错，是在onResume里面，对应ActivityThread就是handleResumeActivity这个方法：

```

final void handleResumeActivity(IBinder token,
 boolean clearHide, boolean isForward, boolean reallyResume) {
 // If we are getting ready to gc after going to the background, well
 // we are back active so skip it.
 unscheduleGcIdler();
 mSomeActivitiesChanged = true;

 // TODO Push resumeArgs into the activity for consideration
 ActivityClientRecord r = performResumeActivity(token, clearHide);

 if (r.window == null && !a.mFinished && willBeVisible) {
 r.window = r.activity.getWindow();
 View decor = r.window.getDecorView();
 decor.setVisibility(View.INVISIBLE);
 WindowManager wm = a.getWindowManager();
 WindowManager.LayoutParams l = r.window.getAttributes();
 a.mDecor = decor;
 l.type = WindowManager.LayoutParams.TYPE_BASE_APPLICATION;
 l.softInputMode |= forwardBit;
 if (a.mVisibleFromClient) {
 a.mWindowAdded = true;
 wm.addView(decor, l);
 }
 }

 // If the window has already been added, but during resume
 // we started another activity, then don't yet make the
 // window visible.
} else if (!willBeVisible) {
 if (localLOGV) Slog.v(
 TAG, "Launch " + r + " mStartedActivity set");
 r.hideForNow = true;
}
.....
}

```

所以，如果在onCreate中通过子线程直接更新UI，并不会抛  
CalledFromWrongThreadException异常。但是一般情况下，我们不会在onCreate中做这  
样的事情。

这就是Android为我们设计的单线程模型，核心就是一句话：Android UI操作并不是线程安  
全的，并且这些操作必须在UI线程执行。但这一句话背后，却隐藏着我们平时看不见的代  
码实现，只有搞懂这些，我们才能知其然知其所以然。

参考：[Android子线程在没有ViewRoot的情况下能刷新UI吗？](#)

# View的工作原理

来源:[sparkyuan.me](http://sparkyuan.me)

View的绘制流程是从ViewRoot的performTraversals方法开始的，它经过measure、layout和draw三个过程才能最终将一个View绘制出来，其中measure用来测量View的宽和高，layout用来确定View在父容器中的放置位置，而draw则负责将View绘制在屏幕上。

## ViewRoot和DecorView

### ViewRoot

ViewRoot对应ViewRootImpl类，它是连接WindowManager和DecorView的纽带，View的三大流程均通过ViewRoot来完成。ActivityThread中，Activity创建完成后，会将DecorView添加到Window中，同时创建ViewRootImpl对象，并建立两者的关联。

### DecorView

DecorView作为顶级View，一般情况下它内部包含一个竖直方向的LinearLayout，在这个LinearLayout里面有上下两个部分（具体情况和Android版本及主体有关），上面的是标题栏，下面的是内容栏。在Activity中通过setContentView所设置的布局文件其实就是被加到内容栏之中的，而内容栏的id是content，在代码中可以通过 `ViewGroup content = (ViewGroup) findViewById(R.android.id.content)` 来得到content对应的layout。

DecorView其实是一个FrameLayout，View层的事件都先经过DecorView，然后才传递给我们的View。

### MeasureSpec

在测量过程中，系统会将View的LayoutParams根据父容器所施加的规则转换成对应的MeasureSpec，然后再根据这个MeasureSpec来测量出View的宽和高。测量出来的宽和高不一定等于View最终的宽和高。

MeasureSpec将SpecMode和SpecSize打包成一个int值来避免过多的对象内存分配，高2位代表SpecMode，低30位代表SpecSize，SpecMode是指测量模式，而SpecSize是指在某种测量模式下的规格大小。

SpecMode有三类：

- **UNSPECIFIED**: 父容器不对View有任何限制，要多大给多大，这种情况一般用于系统内部，表示一种测量状态
- **EXACTLY**: 父容器已经检测出View所需要的精确大小，这个时候View的最终大小就是SpecSize所指定的值。它对应于LayoutParams中的match\_parent和具体的数值这两种模式
- **AT\_MOST**: 父容器指定了一个可用大小即SpecSize，View的大小不能大于这个值，具体是什么值要看不同View的具体实现。它对应于LayoutParams中的wrap\_content。

## 普通MeasureSpec的创建规则

对于普通View，其MeasureSpec由父容器的MeasureSpec和自身的LayoutParams来共同决定。

- 子View为精确宽高，无论父容器的MeasureSpec，子View的MeasureSpec都为精确值且遵循LayoutParams中的值。
- 子View为match\_parent时，如果父容器是精确模式，则子View也为精确模式且为父容器的剩余空间大小；如果父容器是最大模式，则子View也是wrap\_content且不会超过父容器的剩余空间。
- 子View为wrap\_content时，无论父View是精确还是最大模式，子View的模式总是最大模式，且不会超过父容器的剩余空间。

## View的工作流程

### measure

ViewGroup的measure方法会遍历每个子元素，并调用子元素内部的measure方法，measure源码如下：

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
 setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec,
 getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));
}
```

注：

- `getDefaultSize()`返回`MeasureSpec`中的`specSize`, 也就是View测量后的大小。
- `getSuggestedMinimumWidth()`, View如果没有背景, 那么返回`android:minWidth`这个属性指定的值, 这个值可以为0; 如果设置了背景, 则返回背景的最小宽度和`minWidth`中的最大值。
- `getSuggestedMinimumHeight()`, 与`getSuggestedMinimumWidth()`类似。
- 直接继承View的自定义控件需要重写`onMeasure`方法并设置`wrap_content`时的自身大小, 否则在布局中使用`wrap_content`时就相当于使用`match_parent`。因为`LayoutParams=wrap_content`的情况下, `MeasureSpec`为`AT_MOST`, 所以View的宽和高为父容器当前剩余的空间, 这种效果与`match_parent`一致。具体处理方法要根据需求灵活决定。

## 如何得到View的宽和高

在Activity的`onCreate`、`onStart`、`onResume`方法中均无法正确得到某个View的宽/高信息, 这是因为View的measure过程和Activity的生命周期方法不是同步执行的, 因此无法保证Activity执行了`onCreate`、`onStart`、`onResume`时某个View就已经测量完毕了, 如果View还没有测量完毕, 那么获得的宽/高就是0。

可以通过如下四个方法来解决这个问题:

- Activity或者View的`onWindowFocusChanged`方法 (注意该方法会在Activity Pause和resume时被多次调用)
- `view.post(new Runnable( {@Overidde public void run(){}}))`, 在run方法中获取。
- `ViewTreeObserver`中的`onGlobalLayoutListener`中。
- 手动调用View的`measure`方法。

示例代码请参考原书P190页

## layout

layout的作用是用来确定子视图在父视图中的位置。源码如下:

```

public void layout(int l, int t, int r, int b) {
 int oldL = mLeft;
 int oldT = mTop;
 int oldB = mBottom;
 int oldR = mRight;
 boolean changed = setFrame(l, t, r, b);
 if (changed || (mPrivateFlags & LAYOUT_REQUIRED) == LAYOUT_REQUIRED) {
 if (ViewDebug.TRACE_HIERARCHY) {
 ViewDebug.trace(this, ViewDebug.HierarchyTraceType.ON_LAYOUT);
 }
 onLayout(changed, l, t, r, b);
 mPrivateFlags &= ~LAYOUT_REQUIRED;
 if (mOnLayoutChangeListeners != null) {
 ArrayList<OnLayoutChangeListener> listenersCopy =
 (ArrayList<OnLayoutChangeListener>) mOnLayoutChangeListeners.clone();
 int numListeners = listenersCopy.size();
 for (int i = 0; i < numListeners; ++i) {
 listenersCopy.get(i).onLayoutChange(this, l, t, r, b, oldL, oldT, oldB,
 oldR);
 }
 }
 mPrivateFlags &= ~FORCE_LAYOUT;
 }
}

```

通过setFrame()确定四个顶点的位置，进而确定View在父容器中的位置。

在View的默认实现中，View的测量宽/高和最终宽/高是相等的，只不过测量宽/高形成于View的measure过程，而最终宽/高形成于View的layout过程，即两者的赋值时机不同，测量宽/高的赋值时机稍微早一些。多数情况下可以认为View的测量宽/高就等于最终的宽/高，但对于在View的layout中改变了View的left、top、right、bottom四个属性时，得出的测量宽/高有可能和最终的宽/高不一致。

## draw

draw的过程很简单主要有以下几步：

- 绘制背景(background.draw)
- 绘制自己(onDraw)
- 绘制children(dispatchDraw)
- 绘制装饰(onDrawScrollBars)。

源码如下

```

public void draw(Canvas canvas) {

```

```
/ * Draw traversal performs several drawing steps which must be executed
 * in the appropriate order:
 *
 * 1. Draw the background if need
 * 2. If necessary, save the canvas' layers to prepare for fading
 * 3. Draw view's content
 * 4. Draw children (dispatchDraw)
 * 5. If necessary, draw the fading edges and restore layers
 * 6. Draw decorations (scrollbars for instance)
 */

//Step 1, draw the background, if needed
if (!dirtyOpaque) {
 drawBackground(canvas);
}

// skip step 2 & 5 if possible (common case)
final int viewFlags = mViewFlags;
if (!verticalEdges && !horizontalEdges) {
 // Step 3, draw the content
 if (!dirtyOpaque) onDraw(canvas);

 // Step 4, draw the children
 dispatchDraw(canvas);

 // Step 6, draw decorations (scrollbars)
 onDrawScrollBars(canvas);

 if (mOverlay != null && !mOverlay.isEmpty()) {
 mOverlay.getOverlayView().dispatchDraw(canvas);
 }

 // we're done...
 return;
}

// Step 2, save the canvas' layers
...

// Step 3, draw the content
if (!dirtyOpaque)
 onDraw(canvas);

// Step 4, draw the children
dispatchDraw(canvas);

// Step 5, draw the fade effect and restore layers

// Step 6, draw decorations (scrollbars)
onDrawScrollBars(canvas);
}
```

注：

- View有一个特殊的方法setWillNotDraw，如果一个View不需要绘制任何内容，设置这个标记位true后，系统会进行优化。默认情况下，View没有启用这个优化标记位，但是ViewGroup会默认启用这个优化标记位。
- 这个标记位对实际开发的意义是：如果自定义控件继承于ViewGroup并且本身不具备绘制功能时，就可以开启这个标记位从而便于系统进行后续的优化。当明确知道一个ViewGroup需要通过onDraw来绘制内容时，需要显示地关闭WILL\_NOT\_DRAW这个标记位。

欢迎转载，转载请注明出处<http://sparkyuan.me/>



# Activity的绘制流程简单分析

来源:apkbus

要明白这个流程，我们还得从第一部开始，大家都知道在 activity 里面 `setContentView` 调用结束以后 就可以看到程序加载好我们的布局文件了，从而让我们在手机上看到这个画面。那么我们来看一下这个源码是如何实现的。

```
/**
 * Set the activity content from a layout resource. The resource will be
 * inflated, adding all top-level views to the activity.
 *
 * @param layoutResID Resource ID to be inflated.
 *
 * [url=home.php?mod=space&uid=189949]@See[/url] #setContentView(android.view.View)
 * @see #setContentView(android.view.View, android.view.ViewGroup.LayoutParams)
 */
public void setContentView(int layoutResID) {
 getWindow().setContentView(layoutResID);
 initActionBar();
}
```

我们这里看到是调用了 `getWindow` 的返回值来调用 `setContentView` 方法的。

```
/** Retrieve the current {@link android.view.Window} for the activity.
 * This can be used to directly access parts of the Window API that
 * are not available through Activity/Screen.
 *
 * [url=home.php?mod=space&uid=309376]@return[/url] Window The current window, or
 * visual.
 */
public Window getWindow() {
 return mWindow;
}
```

们再来看看 `mWindow` 这个值是从哪来的。

```

final void attach(Context context, ActivityThread aThread, Instrumentation
 Application application, Intent intent, ActivityInfo info,
 CharSequence title, Activity parent, String id,
 NonConfigurationInstances lastNonConfigurationInstances,
 Configuration config) {
 attachBaseContext(context);

 mFragments.attachActivity(this);

 mWindow = PolicyManager.makeNewWindow(this);
 mWindow.setCallback(this);
 mWindow.getLayoutInflater().setPrivateFactory(this);
 if (info.softInputMode != WindowManager.LayoutParams.SOFT_INPUT_STATE_UNSPECIFIED)
 mWindow.setSoftInputMode(info.softInputMode);
}
if (info.uiOptions != 0) {
 mWindow.setUiOptions(info.uiOptions);
}
mUiThread = Thread.currentThread();

mMainThread = aThread;
mInstrumentation = instr;
mToken = token;
mIdent = ident;
mApplication = application;
mIntent = intent;
mComponent = intent.getComponent();
mActivityInfo = info;
mTitle = title;
mParent = parent;
mEmbeddedID = id;
mLastNonConfigurationInstances = lastNonConfigurationInstances;

mWindow.set.WindowManager(null, mToken, mComponent.flattenToString(),
 (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELERATED) != 0);
if (mParent != null) {
 mWindow.setContainer(mParent.getWindow());
}
m.WindowManager = mWindow.get.WindowManager();
mCurrentConfig = config;
}

```

注意看11行的代码 我们发现这个 `mWindow` 的值 是通过 `makeNewWindow` 这个方法来实现的。我们再来看看这个方法，当然了我们要先找到这个类，这个类位

于 `<source_code>/frameworks/base/core/java/com/android/internal/policy/PolicyManager.java`

```
/* * Copyright (C) 2008 The Android Open Source Project
```

```

/*
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.android.internal.policy;

import android.content.Context;
import android.view.FallbackEventHandler;
import android.view.LayoutInflater;
import android.view.Window;
import android.view.WindowManagerPolicy;

import com.android.internal.policy.IPolicy;

/**
 * {@hide}
 */
public final class PolicyManager {
 private static final String POLICY_IMPL_CLASS_NAME =
 "com.android.internal.policy.impl.Policy";

 private static final IPolicy sPolicy;

 static {
 // Pull in the actual implementation of the policy at run-time
 try {
 Class policyClass = Class.forName(POLICY_IMPL_CLASS_NAME);
 sPolicy = (IPolicy)policyClass.newInstance();
 } catch (ClassNotFoundException ex) {
 throw new RuntimeException(
 POLICY_IMPL_CLASS_NAME + " could not be loaded", ex);
 } catch (InstantiationException ex) {
 throw new RuntimeException(
 POLICY_IMPL_CLASS_NAME + " could not be instantiated", ex);
 } catch (IllegalAccessException ex) {
 throw new RuntimeException(
 POLICY_IMPL_CLASS_NAME + " could not be instantiated", ex);
 }
 }

 // Cannot instantiate this class
}

```

```

private PolicyManager() {}

// The static methods to spawn new policy-specific objects
public static Window makeNewWindow(Context context) {
 return sPolicy.makeNewWindow(context);
}

public static LayoutInflater makeNewLayoutInflater(Context context) {
 return sPolicy.makeNewLayoutInflater(context);
}

public static WindowManagerPolicy makeNewWindowManager() {
 return sPolicy.makeNewWindowManager();
}

public static FallbackEventHandler makeNewFallbackEventHandler(Context context) {
 return sPolicy.makeNewFallbackEventHandler(context);
}
}

```

这里发现是一个反射的动态加载，我们暂时不去深究他，继续看代码，找到 Policy 这个类，他位

于 <source\_code>/frameworks/base/policy/src/com/android/internal/policy/impl/Policy.java

```

/*
 * Copyright (C) 2008 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.android.internal.policy.impl;

import android.content.Context;
import android.util.Log;
import android.view.FallbackEventHandler;
import android.view.LayoutInflater;
import android.view.Window;
import android.view.WindowManagerPolicy;

```

```
import com.android.internal.policy.IPolicy;
import com.android.internal.policy.impl.PhoneLayoutInflater;
import com.android.internal.policy.impl.PhoneWindow;
import com.android.internal.policy.impl.PhoneWindowManager;

/**
 * {@hide}
 */

// Simple implementation of the policy interface that spawns the right
// set of objects
public class Policy implements IPolicy {
 private static final String TAG = "PhonePolicy";

 private static final String[] preload_classes = {
 "com.android.internal.policy.impl.PhoneLayoutInflater",
 "com.android.internal.policy.impl.PhoneWindow",
 "com.android.internal.policy.impl.PhoneWindow$1",
 "com.android.internal.policy.impl.PhoneWindow$ContextMenuCallback",
 "com.android.internal.policy.impl.PhoneWindow$DecorView",
 "com.android.internal.policy.impl.PhoneWindow$PanelFeatureState",
 "com.android.internal.policy.impl.PhoneWindow$PanelFeatureState$SavedState",
 };

 static {
 // For performance reasons, preload some policy specific classes when
 // the policy gets loaded.
 for (String s : preload_classes) {
 try {
 Class.forName(s);
 } catch (ClassNotFoundException ex) {
 Log.e(TAG, "Could not preload class for phone policy: " + s);
 }
 }
 }

 public Window makeNewWindow(Context context) {
 return new PhoneWindow(context);
 }

 public LayoutInflater makeNewLayoutInflater(Context context) {
 return new PhoneLayoutInflater(context);
 }

 public WindowManagerPolicy makeNewWindowManager() {
 return new PhoneWindowManager();
 }

 public FallbackEventHandler makeNewFallbackEventHandler(Context context) {
 return new PhoneFallbackEventHandler(context);
 }
}
```

```
}
```

看62行代码，到这里我们就发现了在 `activity` 里 `getQâindow` 返回的 实际上就是这个 `PhoneWindow` 对象！！！！！！！！！！！！！ 我们继续看这个 `PhoneWindow` 类,他位于 `<source_code>/frameworks/base/policy/java/com/android/internal/policy/impl/PhoneWindow.java` ,注意在这里我就不放这个类的源码了，因为2000多行。。。我只抽部分重要的说一下:

```
/**
 * Android-specific Window.
 * <p>
 * todo: need to pull the generic functionality out into a base class
 * in android.widget.
 */
public class PhoneWindow extends Window implements MenuBuilder.Callback {

 private final static String TAG = "PhoneWindow";

 private final static boolean SWEEP_OPEN_MENU = false;

 /**
 * Simple callback used by the context menu and its submenus. The options
 * menu submenus do not use this (their behavior is more complex).
 */
 final DialogMenuCallback mContextMenuCallback = new DialogMenuCallback(FEATURE_CO

 final TypedValue mMinWidthMajor = new TypedValue();
 final TypedValue mMinWidthMinor = new TypedValue();

 // This is the top-level view of the window, containing the window decor.
 private DecorView mDecor;

 // This is the view in which the window contents are placed. It is either
 // mDecor itself, or a child of mDecor where the contents go.
 private ViewGroup mContentParent;

 SurfaceHolder.Callback2 mTakeSurfaceCallback;

 InputQueue.Callback mTakeInputQueueCallback;

 private boolean mIsFloating;

 private LayoutInflater mLayoutInflater;

 private TextView mTitleView;

 private ActionBarView mActionBar;
 private ActionMenuPresenterCallback mActionMenuPresenterCallback;
 private PanelMenuPresenterCallback mPanelMenuPresenterCallback;

 private DrawableFeatureState[] mDrawables;

 private PanelFeatureState[] mPanels;
```

看22和23行代码,我们就知道这个 `DecorView` 就是我们绘制view的时候最顶层的那个 `view`。换句话说就是最根部的视图。而且再继续跟代码,我们会发现他就是 `phonewindow` 的一个内部类,注意看他是继承的 `FrameLayout`

```
private final class DecorView extends FrameLayout implements RootViewSurfaceTaker {
 /* package */ int mDefaultOpacity = PixelFormat.OPAQUE;

 /** The feature ID of the panel, or -1 if this is the application's DecorView
 * @see android.R.styleable.WindowFeature#FEATURE_ID
 */
 private final int mFeatureId;

 private final Rect mDrawingBounds = new Rect();

 private final Rect mBackgroundPadding = new Rect();

 private final Rect mFramePadding = new Rect();

 private final Rect mFrameOffsets = new Rect();

 private boolean mChanging;

 private Drawable mMenuBackground;
 private boolean mWatchingForMenu;
 private int mDownY;

 private ActionMode mActionMode;
 private ActionBarContextView mActionModeView;
 private PopupWindow mActionModePopup;
 private Runnable mShowActionModePopup;

 public DecorView(Context context, int featureId) {
 super(context);
 mFeatureId = featureId;
 }
}
```

所以到这里我们可以发现在 `activity` 里调用 `setConteview` 的时候 最终就是调用的 `PhoneWindow` 的这个方法

```

@Override
public void setContentView(int layoutResID) {
 if (mContentParent == null) {
 installDecor();
 } else {
 mContentParent.removeAllViews();
 }
 mLayoutInflater.inflate(layoutResID, mContentParent);
 final Callback cb = getCallback();
 if (cb != null && !isDestroyed()) {
 cb.onContentChanged();
 }
}

```

这里代码其实也很好理解，如果是第一次调用就 `installDecor` 否则就 `remove` 所有的 `view` 我们来看这个 `installDecor` 的代码

```

private void installDecor() {
 if(mDecor == null) {
 mDecor = generateDecor();
 mDecor.setDescendantFocusability(ViewGroup.FOCUS_AFTER_DESCENDANTS);
 mDecor.setIsRootNamespace(true);
 }
 if(mContentParent == null) {
 mContentParent = generateLayout(mDecor);

 mTitleView = (TextView)findViewById(com.android.internal.R.id.title);
 if(mTitleView != null) {
 if((getLocalFeatures() & (1 << FEATURE_NO_TITLE)) != 0) {
 View titleContainer = findViewById(com.android.internal.R.id.title_container);
 if(titleContainer != null) {
 titleContainer.setVisibility(View.GONE);
 } else {
 mTitleView.setVisibility(View.GONE);
 }
 if(mContentParent instanceof FrameLayout) {
 ((FrameLayout)mContentParent).setForeground(null);
 }
 } else {
 mTitleView.setText(mTitle);
 }
 } else {
 mActionBar = (ActionBarView)findViewById(com.android.internal.R.id.action_bar);
 if(mActionBar != null) {
 mActionBar.setWindowCallback(getCallback());
 if(mActionBar.getTitle() == null) {
 mActionBar.setWindowTitle(mTitle);
 }
 final int localFeatures = getLocalFeatures();
 }
 }
 }
}

```

```
 if((localFeatures & (1 << FEATURE_PROGRESS)) != 0) {
 mActionBar.initProgress();
 }
 if((localFeatures & (1 << FEATURE_INDETERMINATE_PROGRESS)) != 0) {
 mActionBar.initIndeterminateProgress();
 }

 boolean splitActionBar = false;
 final boolean splitWhenNarrow = (mUiOptions & ActivityInfo.UIOPTION_SPLIT_ACTION_BAR_WHEN_NARROW) != 0;
 if(splitWhenNarrow) {
 splitActionBar =
 getContext().getResources().getBoolean(com.android.internal.R.bool.config_activitySplitActionBar);
 } else {
 splitActionBar =
 getWindowStyle().getBoolean(com.android.internal.R.styleable.WindowSplitActionBar);
 }
 final ActionBarContainer splitView =
 (ActionBarContainer)findViewById(com.android.internal.R.id.split_view);
 if(splitView != null) {
 mActionBar.setSplitView(splitView);
 mActionBar.setSplitActionBar(splitActionBar);
 mActionBar.setSplitWhenNarrow(splitWhenNarrow);

 final ActionBarContextView cab =
 (ActionBarContextView)findViewById(com.android.internal.R.id.action_bar_context);
 cab.setSplitView(splitView);
 cab.setSplitActionBar(splitActionBar);
 cab.setSplitWhenNarrow(splitWhenNarrow);
 } else if(splitActionBar) {
 Log.e(TAG, "Requested split action bar with " + "incompatible window style");
 }

 // Post the panel invalidate for later; avoid application onCreateOptions()
 // being called in the middle of onCreate or similar.
 mDecor.post(new Runnable() {

 public void run() {
 // Invalidate if the panel menu hasn't been created before this point.
 PanelFeatureState st = getPanelState(FEATURE_OPTIONS_PANEL, false);
 if(! isDestroyed() && (st == null || st.menu == null)) {
 invalidatePanelMenu(FEATURE_ACTION_BAR);
 }
 }
 });
 }
}
```

注意看第八行代码 这个就是绘制 activity 根布局最关键的地方 这个函数一共有300行左右 我也不能全部放上来，有兴趣的同学可以自己看一下源码，我在这截取部分重要的说。其实重要的代码就是这么一些：

```

int layoutResource;
int features = getLocalFeatures();
// System.out.println("Features: 0x" + Integer.toHexString(features));
if((features & ((1 << FEATURE_LEFT_ICON) | (1 << FEATURE_RIGHT_ICON))) != 0) {
 if(mIsFloating) {
 TypedValue res = new TypedValue();
 getContext().getTheme().resolveAttribute(com.android.internal.R.attr.dialogTi
 layoutResource = res.resourceId;
 } else {
 layoutResource = com.android.internal.R.layout.screen_title_icons;
 }
 // XXX Remove this once action bar supports these features.
 removeFeature(FEATURE_ACTION_BAR);
 // System.out.println("Title Icons!");
} else if((features & ((1 << FEATURE_PROGRESS) | (1 << FEATURE_INDETERMINATE_PROGRESS
 && (features & (1 << FEATURE_ACTION_BAR)) == 0) {
 // Special case for a window with only a progress bar (and title).
 // XXX Need to have a no-title version of embedded windows.
 layoutResource = com.android.internal.R.layout.screen_progress;
 // System.out.println("Progress!");
} else if((features & (1 << FEATURE_CUSTOM_TITLE)) != 0) {
 // Special case for a window with a custom title.
 // If the window is floating, we need a dialog layout
 if(mIsFloating) {
 TypedValue res = new TypedValue();
 getContext().getTheme().resolveAttribute(com.android.internal.R.attr.dialogCu
 layoutResource = res.resourceId;
 } else {
 layoutResource = com.android.internal.R.layout.screen_custom_title;
 }
 // XXX Remove this once action bar supports these features.
 removeFeature(FEATURE_ACTION_BAR);
} else if((features & (1 << FEATURE_NO_TITLE)) == 0) {
 // If no other features and not embedded, only need a title.
 // If the window is floating, we need a dialog layout
 if(mIsFloating) {
 TypedValue res = new TypedValue();
 getContext().getTheme().resolveAttribute(com.android.internal.R.attr.dialogTi
 layoutResource = res.resourceId;
 } else if((features & (1 << FEATURE_ACTION_BAR)) != 0) {
 if((features & (1 << FEATURE_ACTION_BAR_OVERLAY)) != 0) {
 layoutResource = com.android.internal.R.layout.screen_action_bar_overlay;
 } else {
 layoutResource = com.android.internal.R.layout.screen_action_bar;
 }
 } else {

```

```
 layoutResource = com.android.internal.R.layout.screen_title;
 }
 // System.out.println("Title!");
} else if((features & (1 << FEATURE_ACTION_MODE_OVERLAY)) != 0) {
 layoutResource = com.android.internal.R.layout.screen_simple_overlay_action_mode;
} else {
 // Embedded, so no decoration is needed.
 layoutResource = com.android.internal.R.layout.screen_simple;
 // System.out.println("Simple!");
}

mDecor.startChanging();

View in = mLayoutInflater.inflate(layoutResource, null);
decor.addView(in, new ViewGroup.LayoutParams(MATCH_PARENT, MATCH_PARENT));
```

这个 `layoutResource` 的值 实际上就代表了窗口修饰的哪些布局文件，你看最后两行代码就知道，当我们确定了这个布局文件以后就把她 `add` 到 `decor` 这个对象里。所以我们就能想明白 为啥 我们的 `requestWindowFeature` 这个方法一定要在 `setContentview` 前面调用才有作用了~~然后给大家随便看下布局文件吧，就是系统自带的这些根布局。

```

File Edit Selection Find View Goto Tools Project Preferences Help
preference_list_content_single.xml
preference_list_fragment.xml
preference_widget_checkbox.xml
preference_widget_seekbar.xml
preference_widget_switch.xml
preferences.xml
progress_dialog.xml
progress_dialog_holo.xml
recent_apps_dialog.xml
recent_apps_icon.xml
remote_views_adapter_default_loading_view.xml
resolve_list_item.xml
safe_mode.xml
screen.xml
screen_action_bar.xml
screen_action_bar_overlay.xml
screen_custom_title.xml
screen_progress.xml
screen_simple.xml
screen_simple_overlay_action_mode.xml
screen_title.xml
screen_title_icons.xml
search_bar.xml
search_dropdown_item_1line.xml
search_dropdown_item_icons_2line.xml
search_view.xml
seekbar_dialog.xml
select_dialog.xml
select_dialog_holo.xml
select_dialog_item.xml
select_dialog_item_holo.xml
select_dialog_multichoice.xml
select_dialog_multichoice_holo.xml
select_dialog_singlechoice.xml
select_dialog_singlechoice_holo.xml
simple_dropdown_hint.xml
simple_dropdown_item_1line.xml
simple_dropdown_item_2line.xml
simple_expandable_list_item_1.xml
simple_expandable_list_item_2.xml
simple_gallery_item.xml
simple_list_item_1.xml
simple_list_item_2.xml
simple_list_item_2_single_choice.xml
simple_list_item_activated_1.xml
simple_list_item_activated_2.xml
simple_list_item_checked.xml
simple_list_item_multichoice.xml

```

screen\_title.xml    x    Policy.java    x    IPolicy.java    x    PolicyManager.java

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <!-- Copyright (C) 2006 The Android Open Source Project
3
4 Licensed under the Apache License, Version 2.0 (the "License");
5 you may not use this file except in compliance with the License.
6 You may obtain a copy of the License at
7
8 http://www.apache.org/licenses/LICENSE-2.0
9
10 Unless required by applicable law or agreed to in writing, software
11 distributed under the License is distributed on an "AS IS" BASIS,
12 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 See the License for the specific language governing permissions and
14 limitations under the License.
15 -->
16 <!--
17 This is an optimized layout for a screen, with the minimum set of features
18 enabled.
19 -->
20
21 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
22 android:orientation="vertical"
23 android:fitsSystemWindows="true">
24 <!-- Popout bar for action modes -->
25 <ViewStub android:id="@+id/action_mode_bar_stub"
26 android:inflatedId="@+id/action_mode_bar"
27 android:layout="@layout/action_mode_bar"
28 android:layout_width="match_parent"
29 android:layout_height="wrap_content" />
30
31 <FrameLayout
32 android:layout_width="match_parent"
33 android:layout_height="?android:attr/windowTitleSize"
34 style="?android:attr/windowTitleBackgroundStyle">
35 <TextView android:id="@+id/title"
36 style="?android:attr/windowTitleStyle"
37 android:background="@null"
38 android:fadingEdge="horizontal"
39 android:gravity="center_vertical"
40 android:layout_width="match_parent"
41 android:layout_height="match_parent" />
42 </FrameLayout>
43 <FrameLayout android:id="@+id/content"
44 android:layout_width="match_parent"
45 android:layout_height="0dip"
46 android:layout_weight="1"
47 android:foregroundGravity="fill_horizontal|top"
48 android:foreground="?android:attr/windowContentOverlay" />
49
50
51

```

apkbus.com  
安卓巴士出品

这种大家肯定经常用了，就是上面有个标题 然后下面就放我们自己的布局文件来展示内容,当然了还有人喜欢用全屏的 screen\_simple , 他的代码也是很简单的。这里不截图上代码:

```

<?xml version="1.0" encoding="utf-8"?><!--
/* //device/apps/common/assets/res/layout/screen_simple.xml
*/
**
** Copyright 2006, The Android Open Source Project
**
** Licensed under the Apache License, Version 2.0 (the "License");
** you may not use this file except in compliance with the License.
** You may obtain a copy of the License at
**
** http://www.apache.org/licenses/LICENSE-2.0
**
** Unless required by applicable law or agreed to in writing, software
** distributed under the License is distributed on an "AS IS" BASIS,
** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
** See the License for the specific language governing permissions and
** limitations under the License.
*/

```

This is an optimized layout for a screen, with the minimum set of features enabled.

-->

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:fitsSystemWindows="true"
 android:orientation="vertical">
 <ViewStub android:id="@+id/action_mode_bar_stub"
 android:inflatedId="@+id/action_mode_bar"
 android:layout="@layout/action_mode_bar"
 android:layout_width="match_parent"
 android:layout_height="wrap_content" />
 <FrameLayout
 android:id="@android:id/content"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:foregroundInsidePadding="false"
 android:foregroundGravity="fill_horizontal|top"
 android:foreground="?android:attr/windowContentOverlay" />
</LinearLayout>

```

那注意33行代码 `android:id="@android:id/content"` 这个地方跟我们上一张的博客那边有一样的地方，都是用的这个 `id=content` 为根布局的，有兴趣的同学可以看看我们view教程05的结尾部分，

两个串起来看就能明白了~~~

然后看一下这个函数 另外一个重要的代码

```

ViewGroup contentParent = (ViewGroup)findViewById(ID_ANDROID_CONTENT);
if(contentParent == null) {
 throw new RuntimeException("Window couldn't find content container view");
}

if((features & (1 << FEATURE_INDETERMINATE_PROGRESS)) != 0) {
 ProgressBar progress = getCircularProgressBar(false);
 if(progress != null) {
 progress.setIndeterminate(true);
 }
}

// Remaining setup -- of background and title -- that only applies
// to top-level windows.
if(getContainer() == null) {
 Drawable drawable = mBackgroundDrawable;
 if(mBackgroundResource != 0) {
 drawable = getContext().getResources().getDrawable(mBackgroundResource);
 }
 mDecor.setWindowBackground(drawable);
 drawable = null;
 if(mFrameResource != 0) {
 drawable = getContext().getResources().getDrawable(mFrameResource);
 }
 mDecor.setWindowFrame(drawable);

 // System.out.println("Text=" + Integer.toHexString(mTextColor) +
 // " Sel=" + Integer.toHexString(mTextSelectedColor) +
 // " Title=" + Integer.toHexString(mTitleColor));

 if(mTitleColor == 0) {
 mTitleColor = mTextColor;
 }

 if(mTitle != null) {
 setTitle(mTitle);
 }
 setTitleColor(mTitleColor);
}

mDecor.finishChanging();

return contentParent;

```

返回值是 contentParent 而他的值实际上就是我们那个布局文件里装内容的 android id content，很好理解吧 所以 generateLayout 这个函数的作用就是 确定一下我们 activity 的显示风格还有把 content 这个 framelayout 的值给 mContentParent ,然后通过第8行的代码就把我们的布局文件添加到这个 FrameLayout 里了。

```
@Override
public void setContentView(int layoutResID) {
 if(mContentParent == null) {
 installDecor();
 } else {
 mContentParent.removeAllViews();
 }
 mLayoutInflater.inflate(layoutResID, mContentParent);
 final Callback cb = getCallback();
 if(cb != null && !isDestroyed()) {
 cb.onContentChanged();
 }
}
```

最终由 `ActivityManagerService` 这个类还显示我们的 `decorview`。

**最后我们再把前面的流程简单复述一下**

启动一个 `activity` 的时候，我们最终是调用的 `PhoneWindow` 的 `setContentView` 方法，这个方法会创建一个 `DecorView` 对象，然后再过一遍窗口属性这个流程，最后取得 `android:id/content` 这个 `FrameLayout`，然后将布局文件添加到这个 `FrameLayout` 里面，最后由 `ActivityManagerService` 负责把这个最终的界面展示出来~~~

自定义view 07 将会讲一下view的绘制流程~

# UnitTest

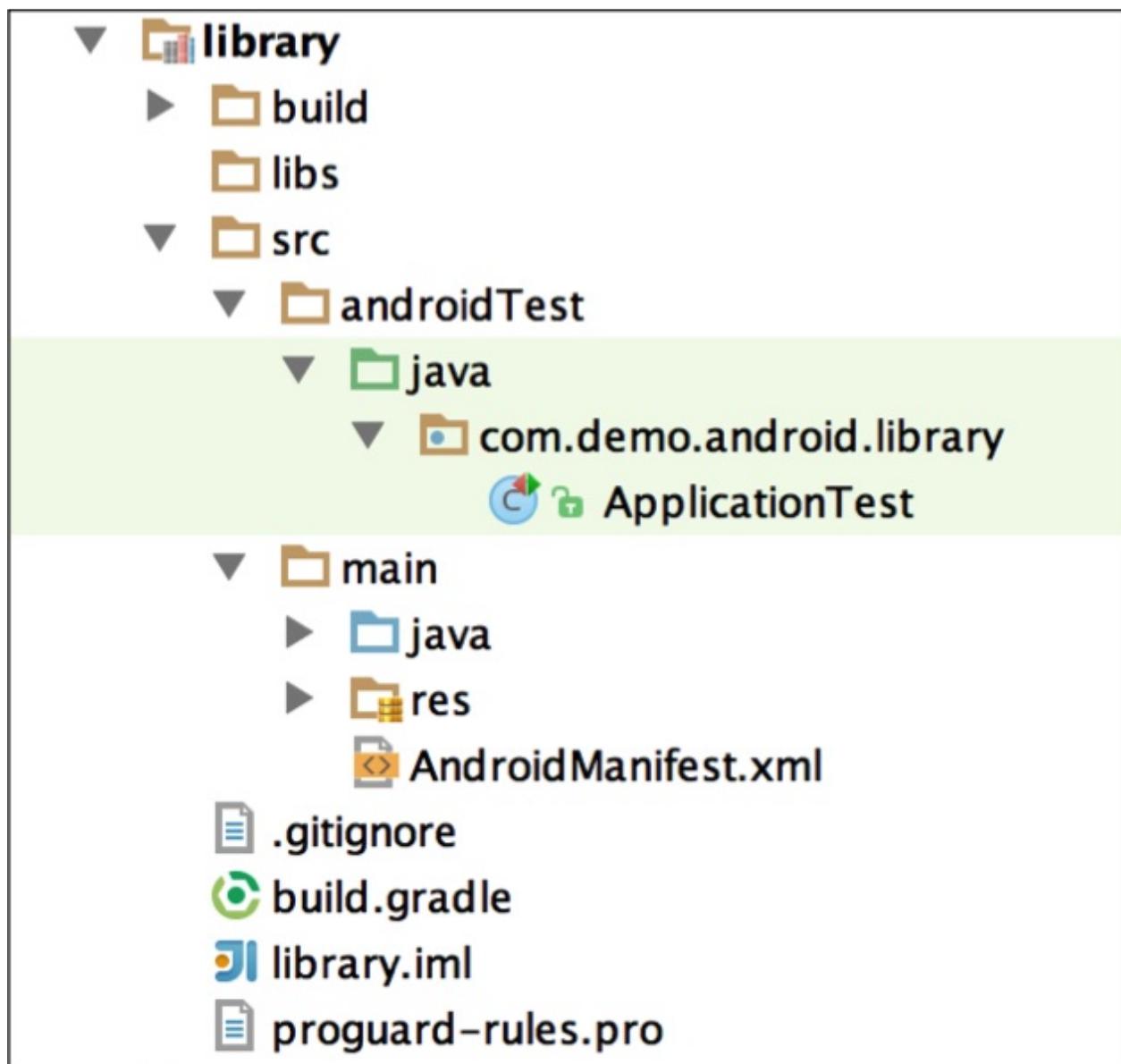
# Android单元测试研究与实践

来源:[美团技术团队](#)

## Android单元测试介绍

处于高速迭代开发中的Android项目往往需要除黑盒测试外更加可靠的质量保障，这正是单元测试的用武之地。单元测试周期性对项目进行函数级别的测试，在良好的覆盖率下，能够持续维护代码逻辑，从而支持项目从容应对快速的版本更新。

单元测试是参与项目开发的工程师在项目代码之外建立的白盒测试工程，用于执行项目中的目标函数并验证其状态或者结果，其中，单元指的是测试的最小模块，通常指函数。如图1所示的绿色文件夹即是单元测试工程。这些代码能够检测目标代码的正确性，打包时单元测试的代码不会被编译进入APK中。



与Java单元测试相同，Android单元测试也是维护代码逻辑的白盒工程，但由于Android运行环境的不同，Android单元测试的环境配置以及实施流程均有所不同。

## Java单元测试

在传统Java单元测试中，我们需要针对每个函数进行设计单元测试用例。如图2便是一个典型的单元测试的用例。

```
public class Obj {
 public Boolean dosomething(Boolean param){
 return param == Boolean.TRUE;
 }
}
```

```
public class ObjTest {
 Obj obj = new Obj();

 @Test
 public void testObj() {
 assertTrue(obj.dosomething(true));
 assertFalse(obj.dosomething(false));
 assertFalse(obj.dosomething(null));
 }
}
```

上述示例中，针对函数dosomething(Boolean param)的每个分支，我们都需要构造相应的参数并验证结果。单元测试的目标函数主要有三种：

- 有明确的返回值，如上图的dosomething(Boolean param)，做单元测试时，只需调用这个函数，然后验证函数的返回值是否符合预期结果。
- 这个函数只改变其对象内部的一些属性或者状态，函数本身没有返回值，就验证它所改变的属性和状态。
- 一些函数没有返回值，也没有直接改变哪个值的状态，这就需要验证其行为，比如点击事件。

既没有返回值，也没有改变状态，又没有触发行为的函数是不可测试的，在项目中不应该存在。当存在同时具备上述多种特性时，本文建议采用多个case来真对每一种特性逐一验证，或者采用一个case，逐一执行目标函数并验证其影响。

构造用例的原则是测试用例与函数一对一，实现条件覆盖与路径覆盖。Java单元测试中，良好的单元测试是需要保证所有函数执行正确的，即所有边界条件都验证过，一个用例只测一个函数，便于维护。在Android单元测试中，并不要求对所有函数都覆盖到，像Android SDK中的函数回调则不用测试。

## Android单元测试

在Android中，单元测试的本质依旧是验证函数的功能，测试框架也是JUnit。在Java中，编写代码面对的只有类、对象、函数，编写单元测试时可以在测试工程中创建一个对象出来然后执行其函数进行测试，而在Android中，编写代码需要面对的是组件、控件、生命周期、异步任务、消息传递等，虽然本质是SDK主动执行了一些实例的函数，但创建一个Activity并不能让它执行到resume的状态，因此需要JUnit之外的框架支持。

当前主流的单元测试框架AndroidTest和Robolectric，前者需要运行在Android环境上，后者可以直接运行在JVM上，速度也更快，可以直接由Jenkins周期性执行，无需准备Android环境。因此我们的单元测试基于Robolectric。对于一些测试对象依赖度较高而需要解除依赖的场景，我们可以借助Mock框架。

## Android单元测试环境配置

### Robolectric环境配置

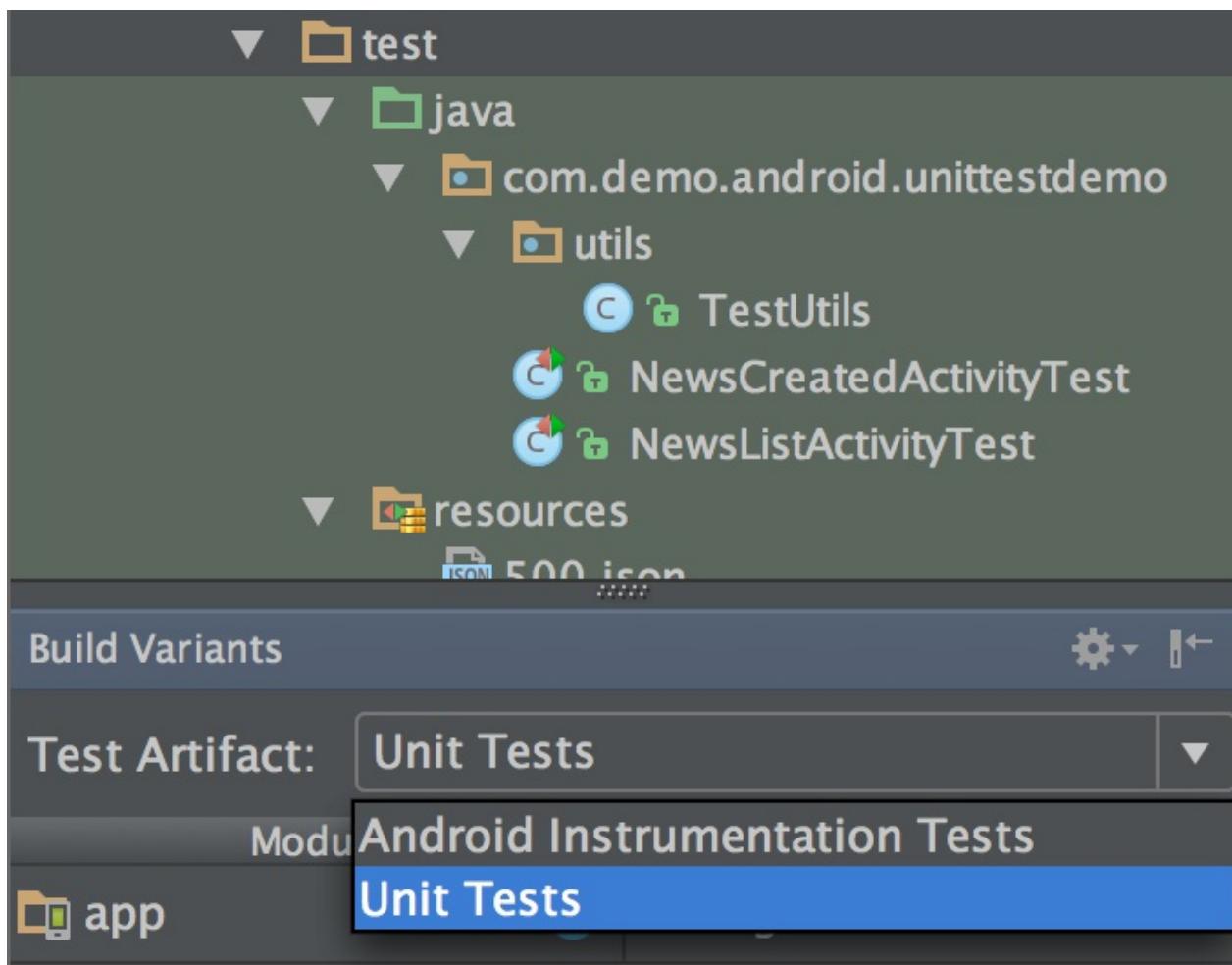
Android单元测试依旧需要JUnit框架的支持，Robolectric只是提供了Android代码的运行环境。如果使用Robolectric 3.0，依赖配置如下：

```
testCompile 'junit:junit:4.10'
testCompile 'org.robolectric:robolectric:3.0'
```

Gradle对Robolectric 2.4的支持并不像3.0这样好，但Robolectric 2.4所有的测试框架均在一个包里，另外参考资料也比较丰富，作者更习惯使用2.4。如果使用Robolectric 2.4，则需要如下配置：

```
classpath 'org.robolectric:robolectric-gradle-plugin:0.14.+' //这行配置在buildscript的dependencies中
apply plugin: 'robolectric'
androidTestCompile 'org.robolectric:robolectric:2.4'
```

上述配置中，本文将testCompile写成androidTest，并且常见的Android工程的单元测试目录名称有test也有androidTest，这两种写法并没有功能上的差别，只是Android单元测试Test Artifact不同而已。Test Artifact如图3所示：



在Gradle插件中，这两种Artifact执行的Task还是有些区别的，但是并不影响单元测试的写法与效果。虽然可以主动配置单元测试的项目路径，本文依旧建议采用与Test Artifact对应的项目路径和配置写法。

## Mock配置

如果要测试的目标对象依赖关系较多，需要解除依赖关系，以免测试用例过于复杂，用Robolectric的Shadow是个办法，但是推荐更加简单的Mock框架，比如Mockito，该框架可以模拟出对象来，而且本身提供了一些验证函数执行的功能。 Mockito配置如下：

```
repositories {
 jcenter()
}
dependencies {
 testCompile "org.mockito:mockito-core:1.+"
}
```

## Robolectric使用介绍

## Robolectric单元测试编写结构

单元测试代码写在项目的test（也可能是androidTest，该目录在项目中会呈浅绿色）目录下。单元测试也是一个标准的Java工程，以类为文件单位编写，执行的最小单位是函数，测试用例（以下简称case）是带有@Test注解的函数，单元测试里面带有case的类由Robolectric框架执行，需要为该类添加注解@RunWith(RobolectricTestRunner.class)。基于Robolectric的代码结构如下：

```
//省略一堆import
@RunWith(RobolectricTestRunner.class)
public class MainActivityTest {
 @Before
 public void setUp() {
 //执行初始化的操作
 }
 @Test
 public void testCase() {
 //执行各种测试逻辑判断
 }
}
```

上述结构中，带有@Before注解的函数在该类实例化后，会立即执行，通常用于执行一些初始化的操作，比如构造网络请求和构造Activity。带有@test注解的是单元测试的case，由Robolectric执行，这些case本身也是函数，可以在其他函数中调用，因此，case也是可以复用的。每个case都是独立的，case不会互相影响，即便是相互调用也不会存在多线程干扰的问题。

## 常见Robolectric用法

Robolectric支持单元测试范围从Activity的跳转、Activity展示View（包括菜单）和Fragment到View的点击触摸以及事件响应，同时Robolectric也能测试Toast和Dialog。对于需要网络请求数据的测试，Robolectric可以模拟网络请求的response。对于一些Robolectric不能测试的对象，比如ConcurrentTask，可以通过自定义Shadow的方式现实测试。下面将着重介绍Robolectric的常见用法。

## Robolectric 2.4模拟网络请求

由于商业App的多数Activity界面数据都是通过网络请求获取，因为网络请求是大多数App首要处理的模块，测试依赖网络数据的Activity时，可以在@Before标记的函数中准备网络数据，进行网络请求的模拟。准备网络请求的代码如下：

```

public void prepareHttpResponse(String filePath) throws IOException {
 String netData = FileUtils.readFileToString(FileUtils.
 toFile(getClass().getResource(filePath)), HTTP.UTF_8);
 Robolectric.setDefaultHttpResponse(200, netData);
} //代码适用于Robolectric 2.4, 3.0需要注意网络请求的包的位置

```

由于Robolectric 2.4并不会发送网络请求，因此需要本地创建网络请求所返回的数据，上述函数的filePath便是本地数据的文件的路径，setDefaultHttpResponse()则创建了该请求的Response。上述函数执行后，单元测试工程便拥有了与本地数据数据对应的网络请求，在这个函数执行后展示的Activity便是有数据的Activity。

在Robolectric 3.0环境下，单元测试可以发真的请求，并且能够请求到数据，本文依旧建议采用mock的办法构造网络请求，而不要依赖网络环境。

## Activity展示测试与跳转测试

创建网络请求后，便可以测试Activity了。测试代码如下：

```

@Test
public void testSampleActivity(){
 SampleActivity sampleActivity=Robolectric.buildActivity(SampleActivity.class).
 create().resume().get();
 assertNotNull(sampleActivity);
 assertEquals("Activity的标题", sampleActivity.getTitle());
}

```

Robolectric.buildActivity()用于构造Activity，create()函数执行后，该Activity会运行到onCreate周期，resume()则对应onResume周期。assertNotNull和assertEquals是JUnit中的断言，Robolectric只提供运行环境，逻辑判断还是需要依赖JUnit中的断言。

Activity跳转是Android开发的重要逻辑，其测试方法如下：

```

@Test
public void testActivityTurnActionBarActivity firstActivity, Class secondActivity) {
 Intent intent = new Intent(firstActivity.getApplicationContext(), secondActivity)
 assertEquals(intent, Robolectric.shadowOf(firstActivity).getNextStartedActivity())
}

```

## Fragment展示与切换

Fragment是Activity的一部分，在Robolectric模拟执行Activity过程中，如果触发了被测试的代码中的Fragment添加逻辑，Fragment会被添加到Activity中。

需要注意Fragment出现的时机，如果目标Activity中的Fragment的添加是执行在onResume阶段，在Activity被Robolectric执行resume()阶段前，该Activity中并不会出现该Fragment。采用Robolectric主动添加Fragment的方法如下：

```
@Test
public void addfragment(Activity activity, int fragmentContent){
 FragmentTestUtil.startFragment(activity.getSupportFragmentManager().findFragmentBy
 Fragment fragment = activity.getSupportFragmentManager().findFragmentById(fragment
 assertNotNull(fragment);
}
```

startFragment()函数的主体便是常用的添加fragment的代码。切换一个Fragment往往由Activity中的代码逻辑完成，需要Activity的引用。

## 控件的点击以及可视验证

```
@Test
public void testButtonClick(int buttonID){
 Button submitButton = (Button) activity.findViewById(buttonID);
 assertTrue(submitButton.isEnabled());
 submitButton.performClick();
 //验证控件的行为
}
```

对控件的点击验证是调用performClick()，然后断言验证其行为。对于ListView这类涉及到Adapter的控件的点击验证，写法如下：

```
//listView被展示之后
listView.performItemClick(listView.getAdapter().getView(position, null, null), 0, 0);
```

与button等控件稍有不同。

## Dialog和Toast测试

测试Dialog和Toast的方法如下：

```

public void testDialog(){
 Dialog dialog = ShadowDialog.getLatestDialog();
 assertNotNull(dialog);
}
public void testToast(String toastContent){
 ShadowHandler.idleMainLooper();
 assertEquals(toastContent, ShadowToast.getTextOfLatestToast());
}

```

上述函数均需要在Dialog或Toast产生之后执行，能够测试Dialog和Toast是否弹出。

## Shadow写法介绍

Robolectric的本质是在Java运行环境下，采用Shadow的方式对Android中的组件进行模拟测试，从而实现Android单元测试。对于一些Robolectric暂不支持的组件，可以采用自定义Shadow的方式扩展Robolectric的功能。

```

@Implements(Point.class)
public class ShadowPoint {
 @RealObject private Point realPoint;
 ...
 public void __constructor__(int x, int y) {
 realPoint.x = x;
 realPoint.y = y;
 }
}//样例来源于Robolectric官网

```

上述实例中，`@Implements`是声明Shadow的对象，`@RealObject`是获取一个Android对象，`constructor`则是该Shadow的构造函数，Shadow还可以修改一些函数的功能，只需要在重载该函数的时候添加`@Implementation`，这种方式可以有效扩展Robolectric的功能。

Shadow是通过对真实的Android对象进行函数重载、初始化等方式对Android对象进行扩展，Shadow出来的对象的功能接近Android对象，可以看成是对Android对象一种修复。自定义的Shadow需要在config中声明，声明写法是`@Config(shadows=ShadowPoint.class)`。

## Mock写法介绍

对于一些依赖关系复杂的测试对象，可以采用Mock框架解除依赖，常用的有Mockito。例如Mock一个List类型的对象实例，可以采用如下方式：

```
List list = mock(List.class); //mock得到一个对象，也可以用@mock注入一个对象
```

所得到的list对象实例便是List类型的实例，如果不采用mock，List其实只是个接口，我们需要构造或者借助ArrayList才能进行实例化。与Shadow不同，Mock构造的是一个虚拟的对象，用于解耦真实对象所需要的依赖。Mock得到的对象仅仅是具备测试对象的类型，并不是真实的对象，也就是并没有执行过真实对象的逻辑。

Mock也具备一些补充JUnit的验证函数，比如设置函数的执行结果，示例如下：

```
When(sample.dosomething()).thenReturn(someAction); //when(一个函数执行).thenReturn(一个可
//上述代码是设置sample.dosomething()的返回值，当执行了sample.dosomething()这个函数时，就会得到
```

上述代码为被测函数定义一个可替代真实函数的结果的返回值。当使用这个函数后，这个可验证的结果便会产生影响，从而代替函数的真实结果，这样便解除了对真实函数的依赖。

同时Mock框架也可以验证函数的执行次数，代码如下：

```
List list = mock(List.class); //Mock得到一个对象
list.add(1); //执行一个函数
verify(list).add(1); //验证这个函数的执行
verify(list, time(3)).add(1); //验证这个函数的执行次数
```

在一些需要解除网络依赖的场景中，多使用Mock。比如对retrofit框架的网络依赖解除如下：

```
//代码参考了参考文献[3]
public class MockClient implements Client {
 @Override
 public Response execute(Request request) throws IOException {
 Uri uri = Uri.parse(request.getUrl());
 String responseString = "";
 if(uri.getPath().equals("/path/of/interest")) {
 responseString = "返回的json1";//这里是设置返回值
 } else {
 responseString = "返回的json2";
 }
 return new Response(request.getUrl(), 200, "nothing", Collections.EMPTY_LIST,
 }
}
//MockClient使用方式如下：
RestAdapter.Builder builder = new RestAdapter.Builder();
builder.setClient(new MockClient());
```

这种方式下retrofit的response可以由单元测试编写者设置，而不来源于网络，从而解除了对网络环境的依赖。

## 在实际项目中使用Robolectric构建单元测试

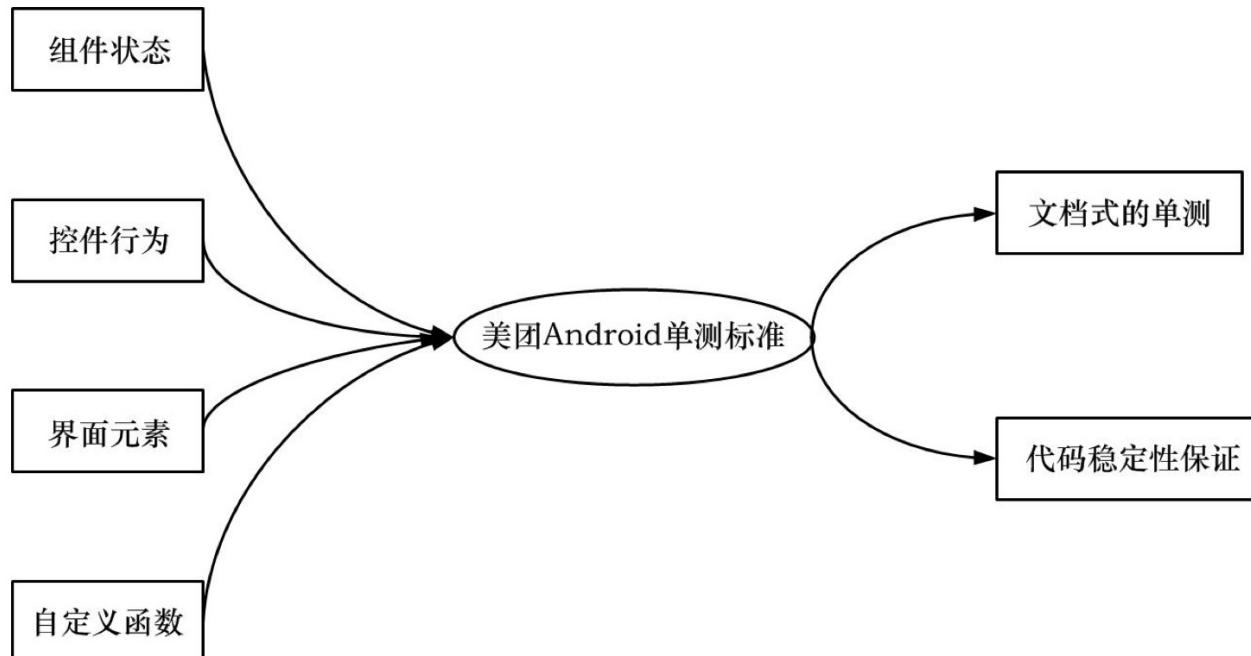
### 单元测试的范围

在Android项目中，单元测试的对象是组件状态、控件行为、界面元素和自定义函数。本文并不推荐对每个函数进行一对一的测试，像onStart()、onDestroy()这些周期函数并不需要全部覆盖到。商业项目多采用Scrum模式，要求快速迭代，有时候未必有较多的时间写单元测试，不再要求逐个函数写单元测试。

本文单元测试的case多来源于一个简短的业务逻辑，单元测试case需要对这段业务逻辑进行验证。在验证的过程中，开发人员可以深度了解业务流程，同时新人来了看一下项目单元测试就知道哪个逻辑跑了多少函数，需要注意哪些边界——是的，单元测试需要像文档一样具备业务指导能力。

在大型项目中，遇到需要改动基类中代码的需求时，往往不能准确快速地知道改动后的影  
响范围，紧急时多采用创建子类覆盖父类函数的办法，但这不是长久之计，在足够覆盖率的单元测试支持下，跑一下单元测试就知道某个函数改动后的影响，可以放心地修改基类。

美团的Android单元测试编写流程如图4所示。



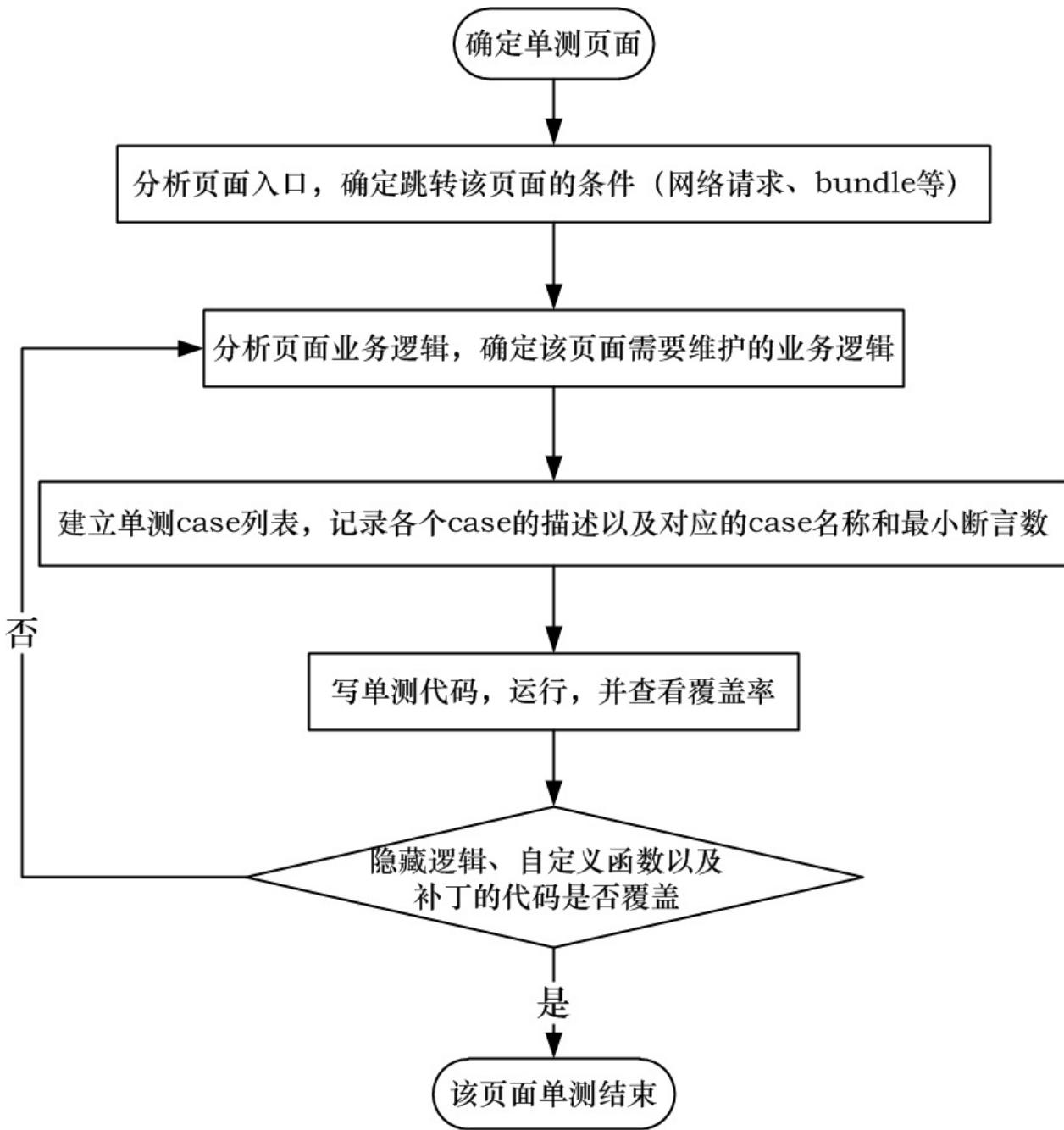
单元测试最终需要输出文档式的单元测试代码，为线上代码提供良好的代码稳定性保证。

## 单元测试的流程

实际项目中，单元测试对象与页面是一对一的，并不建议跨页面，这样的单元测试耦合度太大，维护困难。单元测试需要找到页面的入口，分析项目页面中的元素、业务逻辑，这里的逻辑不仅仅包括界面元素的展示以及控件组件的行为，还包括代码的处理逻辑。然后可以创建单元测试case列表（列表用于纪录项目中单元测试的范围，便于单元测试的管理以及新人了解业务流程），列表中记录单元测试对象的页面，对象中的case逻辑以及名称等。工程师可以根据这个列表开始写单元测试代码。

单元测试是工程师代码级别的质量保证工程，上述流程并不能完全覆盖重要的业务逻辑以及边界条件，因此，需要写完后，看覆盖率，找出单元测试中没有覆盖到的函数分支条件等，然后继续补充单元测试case列表，并在单元测试工程代码中补上case。

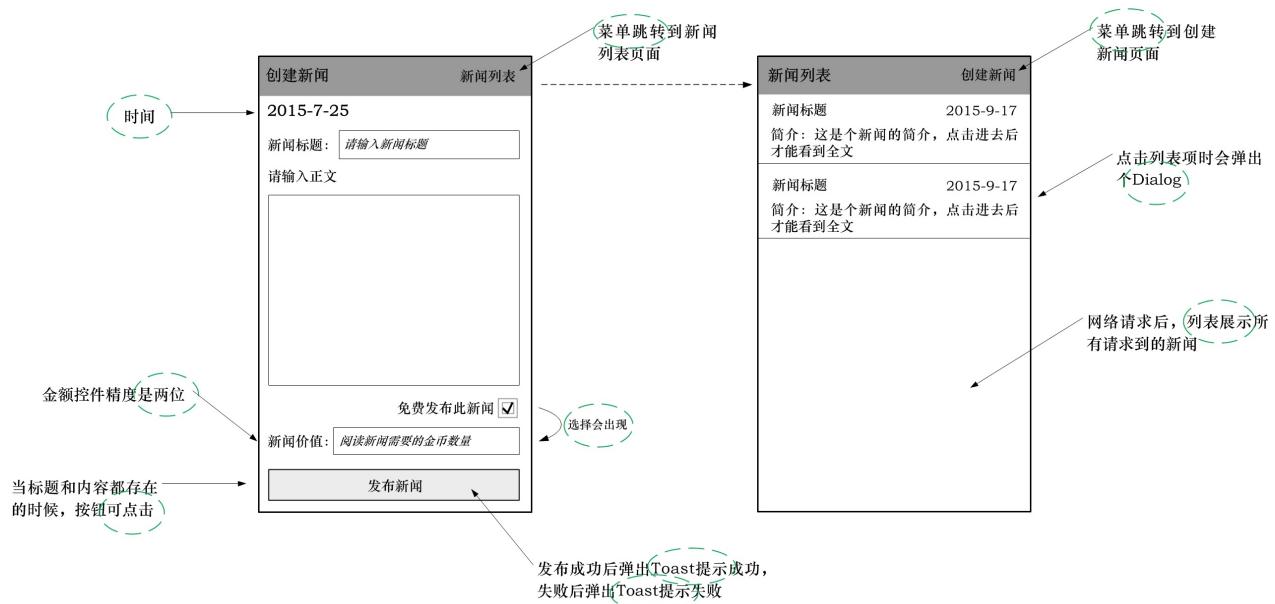
直到规划的页面中所有逻辑的重要分支、边界条件都被覆盖，该项目的单元测试结束。单元测试流程如图5所示。



上述分析页面入口所得到结果便是@Before标记的函数中的代码，之后的循环便是所有的case (@Test标记的函数）。

## 单元测试项目实践

为了系统的介绍单元测试的实施过程，本文创建了一个小型的demo项目作为测试对象。demo的功能是供用户发布所见的新闻到服务端，并浏览所有已经发表的新闻，是个典型的自媒体应用。该demo的开发和测试涉及到TextView、EditText、ListView、Button以及自定义View，包含了网络请求、多线程、异步任务以及界面跳转等。能够为多数商业项目提供参照样例。项目页面如图6所示。



首先需要分析App的每个页面，针对页面提取出简短的业务逻辑，提取出的业务逻辑如图6绿色圈图所示。根据这些逻辑来设计单元测试的case（带有@Test注解的那个函数），这里的业务逻辑不仅指需求中的业务，还包括其他需要维护的代码逻辑。业务流程不允许跨页面，以免增加单元测试case的维护成本。针对demo中界面的单元测试case设计如下：

表1 单元测试case列表

| 目标页面                          | 业务覆盖        | 界面元素                     | 逻辑描述                                                                                                             | 最小断言数 | case名称                                                               |
|-------------------------------|-------------|--------------------------|------------------------------------------------------------------------------------------------------------------|-------|----------------------------------------------------------------------|
| 创建新闻页面<br>NewsCreatedActivity | 编写新闻        | 1.标题框<br>2.内容框<br>3.发布按钮 | 1.向标题框输入内容<br>2.向内容框输入内容<br>3.当标题和内容都存在的时候，上传按钮可点击                                                               | 3     | testWriteNews()                                                      |
|                               | 输入新闻的金额     | 1.Checkbox<br>2.金额控件     | 1.选中免费发布时，金额输入框消失<br>2.不选免费时可以输入金额<br>3.金额输入框只接受小数点后最多两位                                                         | 3     | testValue()                                                          |
|                               | 菜单跳转至新闻列表   | 1.菜单按钮                   | 1.点击菜单跳转到新闻列表页面                                                                                                  | 1     | testMenuForTrunNewsList()                                            |
|                               | 发布新闻        | 1.发布按钮<br>2.Toast        | 1.当标题或者内容为空时，发布按钮不可点击<br>2.编写了新闻的前提下，点击发布按钮<br>3.新闻发布成功，弹出Toast提示“新闻已提交”<br>4.没有标题或者内容时，新闻发布失败，弹出Toast提示“新闻提交失败” | 5     | testNewsPush()、<br>testPushNewsFailed()                              |
| 新闻列表页面<br>NewsListActivity    | 浏览新闻列表      | 1.列表                     | 1.进入此页面后会出现新闻列表<br>2.有网络情况下，能发起网络请求<br>3.网络请求需要用Mock解除偶和，单独验证页面对数据的响应，后端返回一项时，列表只有一条数据                           | 6     | testNewsListNoNetwork()<br>testGetnewsWhenNetwork()<br>testSetNews() |
|                               | 菜单跳转至创建新闻页面 | 1.菜单按钮                   | 1.点击菜单跳转到创建新闻页面                                                                                                  | 1     | testMenuForTrunCreatNews()                                           |
|                               | 查看详细新闻      | 1.有内容的列表<br>2.Dialog     | 1.有新闻的前提下，列表可点击，点击弹出Dialog                                                                                       | 1     | testNewsDialog()                                                     |

接下来需要在单元测试工程中实现上述case，最小断言数是业务逻辑上的判断，并不是代码的边界条件，真实的case需要考虑代码的边界条件，比如数组为空等条件，因此，最终的断言数量会大于等于最小断言数。在需求业务上，最小断言数也是该需求的业务条件。

写完case后需要跑一遍单元测试并检查覆盖率报告，当覆盖率报告中缺少有些单元测试case列表中没有但是实际逻辑中会有的逻辑时，需要更新单元测试case列表，添加遗漏的逻辑，并将对应的代码补上。直到所有需要维护的逻辑都被覆盖，该项目中的单元测试才算完成。单元测试并不是QA的黑盒测试，需要保证对代码逻辑的覆盖。

对表1分析，第一个页面的“发布新闻”的case可以直接调用“编写新闻”的case，以满足条件“2.编写了新闻的前提下，点击发布按钮”，在JUnit框架下，case（带@Test注解的那个函数）也是个函数，直接调用这个函数就不是case，和case是无关的，两者并不会相互影响，可以直接调用以减少重复代码。第二个页面不同于第一个，一进入就需要网络请求，后续业务都需要依赖这个网络请求，单元测试不应该对某一个条件过度耦合，因此，需要用mock解除耦合，直接mock出网络请求得到的数据，单独验证页面对数据的响应。

## 总结

单元测试并不是一个能直接产生回报的工程，它的运行以及覆盖率也不能直接提升代码质量，但其带来的代码控制力能够大幅度降低大规模协同开发的风险。现在的商业App开发都是大型团队协作开发，不断会有新人加入，无论新人是刚入行的应届生还是工作多年，在代码存在一定业务耦合度的时候，修改代码就有一定风险，可能会影响之前比较隐蔽的业务逻辑，或者是丢失曾经的补丁，如果有高覆盖率的单元测试工程，就能很快定位到新增代码对现有项目的影响，与QA验收不同，这种影响是代码级的。

在本文所设计的单元测试流程中，单元测试的case和具体页面的具体业务流程以及该业务的代码逻辑紧密联系，单元测试如同技术文档一般，能够体现出一个业务逻辑运行了多少函数，需要注意什么样的条件。这是一种新人了解业务流程、对业务进行代码级别融入的好办法，看一下以前的单元测试case，就能知道与该case对应的那个页面上的那个业务逻辑会执行多少函数，以及这些函数可能出现的结果。

## 参考文献

- [1] <http://robolectric.org>
- [2] <https://github.com/square/okhttp/tree/master/mockwebserver>
- [3] <http://stackoverflow.com/questions/17544751/square-retrofit-server-mock-for-testing>
- [4] [https://en.wikipedia.org/wiki/Unit\\_testing](https://en.wikipedia.org/wiki/Unit_testing)



# adb

# Android 通过adb shell命令查看内存，CPU，启动时间，电量等信息

来源:[http://blog.sina.com.cn/s/blog\\_13cc013b50102vfwu.html](http://blog.sina.com.cn/s/blog_13cc013b50102vfwu.html)

## 1、查看内存信息

### 1) 查看所有内存信息

命令：

```
dumps sys meminfo
```

或者

```
adb shell dumps sys meminfo
```

例：

```
C:\Users\laiyu>adb shell
shell@android:/ $ dumps sys meminfo
dumps sys meminfo
Applications Memory Usage (kB):
Uptime: 80066272 Realtime: 226459939

Total PSS by process:
 90058 kB: com.tencent.mobileqq (pid 16731)
 57416 kB: system (pid 651)
 52052 kB: com.miui.home (pid 1121)
.....(篇幅问题, 略)

Total PSS by OOM adjustment:
 223177 kB: Persistent
 57416 kB: system (pid 651)
 50036 kB: com.android.deskclock (pid 1096)
.....
 252678 kB: Foreground
 90058 kB: com.tencent.mobileqq (pid 16731)
.....
 50944 kB: Visible
 20318 kB: com.miui.miwallpaper (pid 974)
.....
 90855 kB: Perceptible
```

```
36448 kB: com.google.android.inputmethod.pinyin (pid 987)
.....
39654 kB: A Services
 23320 kB: com.tencent.android.qqdownloader (pid 14080)
.....
49659 kB: B Services
 20085 kB: com.tencent.mobileqq:qzone (pid 19646)
.....
148413 kB: Background
 21457 kB: com.miui.weather2 (pid 14296)
.....
3453 kB: com.miui.providers.datahub (pid 14651)

Total PSS by category:
454627 kB: Dalvik
137206 kB: Unknown
100835 kB: .so mmap
62670 kB: .dex mmap
54208 kB: Other dev
30258 kB: Other mmap
8527 kB: .apk mmap
4752 kB: .ttf mmap
2216 kB: Ashmem
60 kB: Cursor
21 kB: .jar mmap
0 kB: Native

Total PSS: 855380 kB
KSM: 0 kB saved from shared 0 kB
 0 kB unshared; 0 kB volatile
```

## 2) 查看某个包的内存信息

命令：

```
| dumpsys meminfo pkg_name
```

或者

```
| adb shell dumpsys meminfo pkg_name
```

例：

```
wangheng:adb wangheng$ adb shell dumpsys meminfo com.reshow.rebo
Applications Memory Usage (kB):
Uptime: 1114931193 Realtime: 1493301728
```

| ** MEMINFO in pid 3473 [com.reshow.rebo] ** |           |           |           |           |           |           |           |
|---------------------------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
|                                             | Pss       | Private   | Private   | Swapped   | Heap      | Heap      | Heap      |
|                                             | Total     | Dirty     | Clean     | Dirty     | Size      | Alloc     | Free      |
|                                             | - - - - - | - - - - - | - - - - - | - - - - - | - - - - - | - - - - - | - - - - - |
| Native Heap                                 | 8417      | 1572      | 0         | 0         | 29655     | 29655     | 22568     |
| Dalvik Heap                                 | 19668     | 992       | 0         | 0         | 63981     | 48568     | 15413     |
| Dalvik Other                                | 702       | 200       | 0         | 0         |           |           |           |
| Stack                                       | 280       | 84        | 0         | 0         |           |           |           |
| Other dev                                   | 18250     | 1848      | 3236      | 0         |           |           |           |
| .so mmap                                    | 5478      | 28        | 3064      | 0         |           |           |           |
| .apk mmap                                   | 394       | 0         | 220       | 0         |           |           |           |
| .ttf mmap                                   | 29        | 0         | 0         | 0         |           |           |           |
| .dex mmap                                   | 9000      | 0         | 8996      | 0         |           |           |           |
| code mmap                                   | 2683      | 0         | 892       | 0         |           |           |           |
| image mmap                                  | 2246      | 72        | 64        | 0         |           |           |           |
| Other mmap                                  | 450       | 4         | 160       | 0         |           |           |           |
| Unknown                                     | 1812      | 368       | 0         | 0         |           |           |           |
| TOTAL                                       | 69409     | 5168      | 16632     | 0         | 93636     | 78223     | 37981     |

#### Objects

|                   |     |                |    |
|-------------------|-----|----------------|----|
| Views:            | 265 | ViewRootImpl:  | 1  |
| AppContexts:      | 7   | Activities:    | 1  |
| Assets:           | 2   | AssetManagers: | 2  |
| Local Binders:    | 23  | Proxy Binders: | 27 |
| Death Recipients: | 1   |                |    |
| OpenSSL Sockets:  | 0   |                |    |

#### SQL

|                     |     |              |    |
|---------------------|-----|--------------|----|
| MEMORY_USED:        | 435 | MALLOC_SIZE: | 62 |
| PAGECACHE_OVERFLOW: | 128 |              |    |

#### DATABASES

| pgsz | dbsz | Lookaside(b) | cache   | Dbname                              |
|------|------|--------------|---------|-------------------------------------|
| 4    | 20   | 25           | 2/20/3  | /storage/emulated/0/emlibs/libs/mo  |
| 4    | 20   | 37           | 76/19/5 | /data/data/com.reshow.rebo/databas  |
| 4    | 16   | 36           | 76/17/5 | /storage/emulated/0/Mob/comm/dbs/.i |

具体输出项含义请搜索网络

## 2、查看CPU信息

### 方法1：linux系统的top命令

```
top -d 1 | busybox grep "pkg_name"
```

或者

```
adb shell top -d 1 | busybox grep "pkg_name"
```

或者

```
top -d 1 | grep "pkg_name"
```

或者

```
adb shell top -d 1 | grep "pkg_name"
```

注：直接使用grep可能报错，提示找不到命令，这时如果busybox中有grep命令，可以如上，busybox grep

例子：

```
4425 0 0% S 1 1542328K 82204K fg u0_a362 com.reshow.rebo
3473 1 1% S 84 1667756K 222156K fg u0_a362 com.reshow.rebo
4425 0 0% S 1 1542328K 82204K fg u0_a362 com.reshow.rebo
3473 1 9% S 94 1680536K 226336K fg u0_a362 com.reshow.rebo
4425 0 0% S 1 1542328K 82204K fg u0_a362 com.reshow.rebo
3473 0 15% S 102 1738492K 232592K fg u0_a362 com.reshow.rebo
4425 0 0% S 1 1542328K 82204K fg u0_a362 com.reshow.rebo
3473 1 48% S 104 1770592K 251836K fg u0_a362 com.reshow.rebo
4425 0 0% S 1 1542328K 82204K fg u0_a362 com.reshow.rebo
3473 0 73% S 104 1773752K 269244K fg u0_a362 com.reshow.rebo
4425 0 0% S 1 1542328K 82204K fg u0_a362 com.reshow.rebo
3473 2 59% S 104 1774844K 271324K fg u0_a362 com.reshow.rebo
4425 0 0% S 1 1542328K 82204K fg u0_a362 com.reshow.rebo
3473 2 62% S 105 1775908K 272168K fg u0_a362 com.reshow.rebo
4425 0 0% S 1 1542328K 82204K fg u0_a362 com.reshow.rebo
3473 2 65% S 105 1775908K 274144K fg u0_a362 com.reshow.rebo
```

第三列的百分数就是CPU利用率

## 方法2：通过dummps sys cpuinfo命令

```
adb shell dumpsys cpuinfo
```

或者分成两部走(参考 查看电量信息)

先 adb shell，然后 dumpsys cpuinfo

例：

```
wangheng:adb wangheng$ adb shell dumpsys cpuinfo
Load: 13.72 / 9.22 / 6.14
CPU usage from 7969ms to 2155ms ago:
214% 3473/com.reshow.rebo: 194% user + 20% kernel / faults: 34850 minor
91% 10545/mediaserver: 81% user + 10% kernel / faults: 1292 minor
23% 4249/com.qihoo.gameassist: 17% user + 5.8% kernel / faults: 1498 minor
21% 29278/com.tencent.mm: 19% user + 1.5% kernel / faults: 854 minor
10% 10413/surfaceflinger: 4.9% user + 5.3% kernel
7.1% 237/logd: 6.3% user + 0.8% kernel
2.9% 10689/system_server: 2% user + 0.8% kernel / faults: 22 minor
2.5% 494/adbd: 0.1% user + 2.3% kernel / faults: 228 minor
1.8% 491/kdfrgx: 0% user + 1.8% kernel
1.8% 4054/kworker/u8:0: 0% user + 1.8% kernel
1.5% 4453/kworker/u8:1: 0% user + 1.3% kernel + 0.1% iowait
1.5% 4459/kworker/u8:3: 0% user + 1.5% kernel
1.3% 232/irq/104-ispirq: 0% user + 1.3% kernel
1.3% 294/dhd_dpc: 0% user + 1.3% kernel
1.1% 4560/kworker/u8:5: 0% user + 1.1% kernel
1% 4184/com.qihoo.daemon: 0.5% user + 0.5% kernel / faults: 11 minor
1% 11325/logcat: 0.3% user + 0.6% kernel
0.5% 128/mmcqd/0: 0% user + 0.5% kernel
0.5% 29472/com.tencent.mm:push: 0% user + 0.5% kernel / faults: 1069 minor
0.3% 153/irq/24-intel_ss: 0% user + 0.3% kernel
0.3% 5048/kworker/u8:6: 0% user + 0.3% kernel
0.3% 32435/kworker/2:2: 0% user + 0.3% kernel
0% 3/ksoftirqd/0: 0% user + 0% kernel
0.1% 8/rcu_preempt: 0% user + 0.1% kernel
0.1% 37/irq/47-intel_ps: 0% user + 0.1% kernel
0.1% 293/dhd_watchdog_th: 0% user + 0.1% kernel
0.1% 308/sensorhubd: 0% user + 0.1% kernel
0.1% 17315/com.eg.android.AlipayGphone:push: 0.1% user + 0% kernel / faults: 66 minor
0.1% 19699/wpa_supplicant: 0% user + 0.1% kernel
99% TOTAL: 81% user + 15% kernel + 1.8% irq + 0.5% softirq
```

### 3、查看应用启动时间

#### 方法1

命令：

```
adb logcat -c && adb logcat -f /mnt/sdcard/up.txt -s tag
```

选项说明

```
-c 清屏
-f 指定运行结果输出文件， 默认输出到标准设备（一般是显示器
-s 设置默认的过滤级别为Silent
tag 仅显示priority/tag
```

更多信息烦请参考 `adb logcat -help`

例：

先启动app，然后执行如下命令

```
例：先启动app，然后执行如下命令 C:\Users\laiyu>adb logcat -c && adb logcat -f /mnt/sdcard/up.txt -s ActivityManager
```

备注：I/ActivityManager：I代表优先级，ActivityManager代表tag

## 方法2

```
adb shell am start -W package_name/package_name.activity_name
```

例子：

```
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
cmp=com.reshow.rebo/.entry.splash.SplashActivity }
Status: ok
Activity: com.reshow.rebo/.entry.splash.SplashActivity
ThisTime: 192
TotalTime: 216
WaitTime: 229
Complete
```

## 4、查看电量信息

命令：

```
dumpsyst battery
```

例：

```
shell@android:/ $ dumpsys battery
dumpsys battery
Current Battery Service state:
 AC powered: false
 USB powered: true
 status: 5
 health: 2
 present: true
 level: 100
 scale: 100
 voltage:4211
 temperature: 297
 technology: Li-poly
shell@android:/ $
```

# AllInOne

# Android内存泄漏终极解决篇 上

来源:[Android内存泄漏终极解决篇 上](#)

## 一、概述

Android内存的文章详见：[http://blog.csdn.net/lingshu\\_java/article/details/39480761](http://blog.csdn.net/lingshu_java/article/details/39480761)

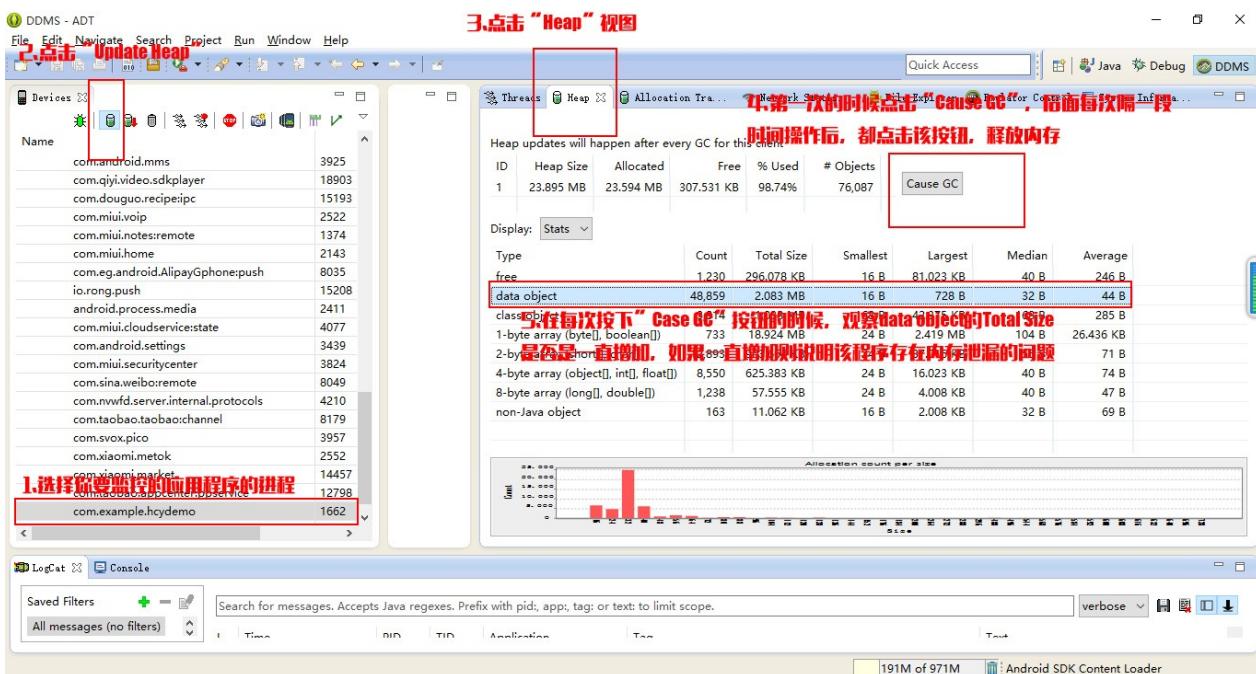
在Android的开发中，经常听到“内存泄漏”这个词。“内存泄漏”就是一个对象已经不需要再使用了，但是因为其它的对象持有该对象的引用，导致它的内存不能被回收。“内存泄漏”的慢慢积累，最终会导致OOM的发生，千里之堤，毁于蚁穴。所以在写代码的过程中，应该要注意规避会导致“内存泄漏”的代码写法，提高软件的健壮性。

本文将从发现问题、解决问题、总结问题的三个角度出发，循序渐进，彻底解决“内存泄漏”的问题。

## 二、内存泄漏的检查工具Heap

工欲善其事必先利其器，要检测“内存泄漏”的发生，需要借助DDMS中的Heap工具及MAT工具，Heap工具用于大致分析是否存在“内存泄漏”，而MAT工具则用于分析“内存泄漏”发生在哪。

Heap工具的使用介绍



## 具体操作

- 1.在Devices设备列表中，找到你所在的设备，点击你想要监控的进程。
- 2.点击“Update Heap”按钮更新堆内存的情况。
- 3.点击“Heap”视图，查看内存的情况。
- 4.每次在Activity的退出和进入的时候点击“Cause GC”，手动调用GC释放应用的内存。
- 5.观察data object那一行，每一次点击“Casue GC”的时候，观察Total Size的值，如果该值不断增加，则说明该应用程序存在“内存泄漏”。

我们先模拟一下内存泄漏，然后通过Heap工具来判断一下是否存在内存泄漏。

上一段存在内存泄漏的代码：

```
public class LeakAty extends Activity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.aty_leak);
 testLeak();

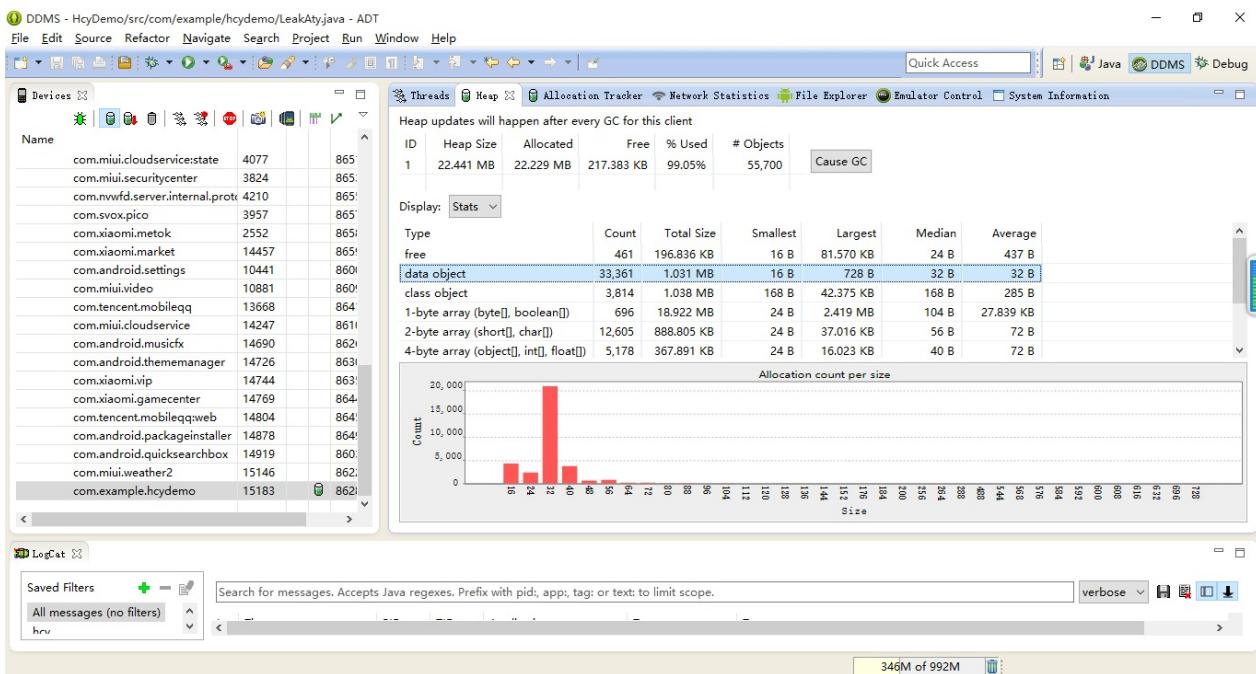
 }

 /**
 * 测试内存泄漏的代码
 */
 private void testLeak() {
 new Thread(new Runnable() {

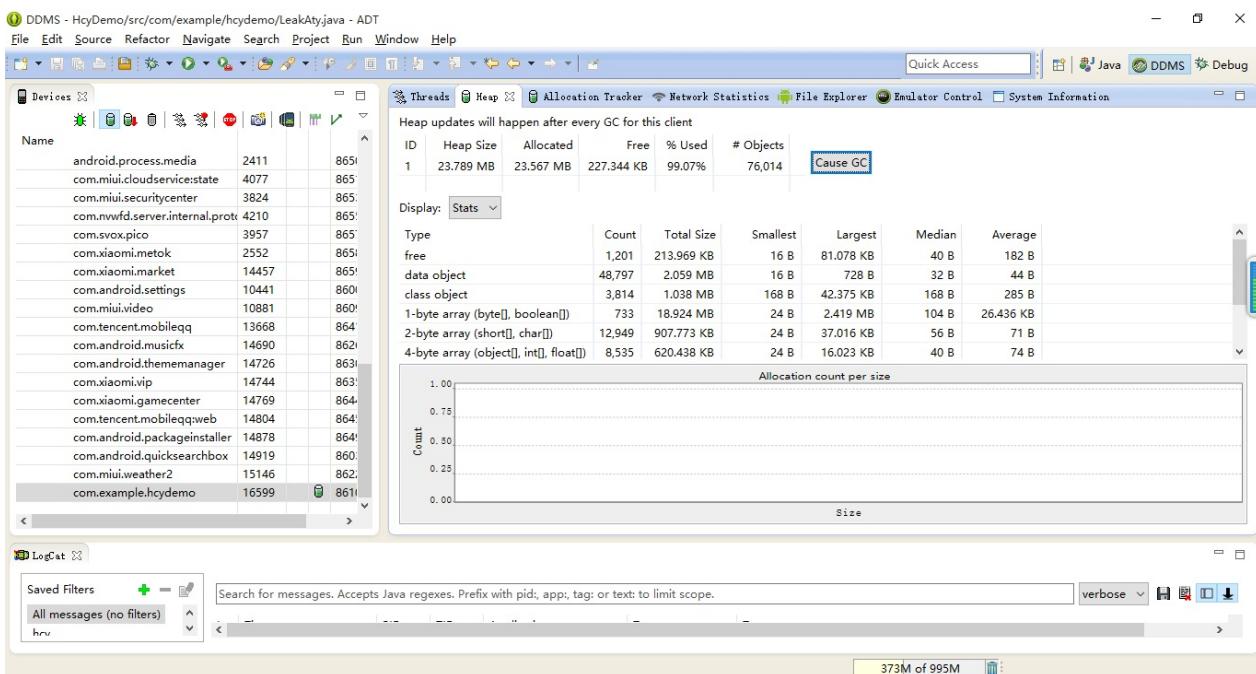
 @Override
 public void run() {
 while (true) {
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
 }).start();
 }
}
```

上述的代码存在内存泄漏，new Runnable(){}是一个非静态的匿名内部类，所以它会强引用创建它的外围对象LeakAty，我们来测试一下内存泄漏的过程，开启手机的方向旋转功能，不断地旋转手机，让LeakAty不断地创建新的实例。理论上如果不存在上述泄漏的代码，之前的Activity会在onDestory之后被回收内存。而一旦存在上述泄漏的代码，新创建的Ruannale实例会一直处于运行状态，它不会被回收，而它强引用的LeakAty当然也不会被回收，所以在屏幕不断旋转，之前创建的LeakAty就不会被释放，会导致旋转n次，内存中就存在n+1个的LeakAty实例。

Heap工具第一次按下Cause GC按钮的截图：



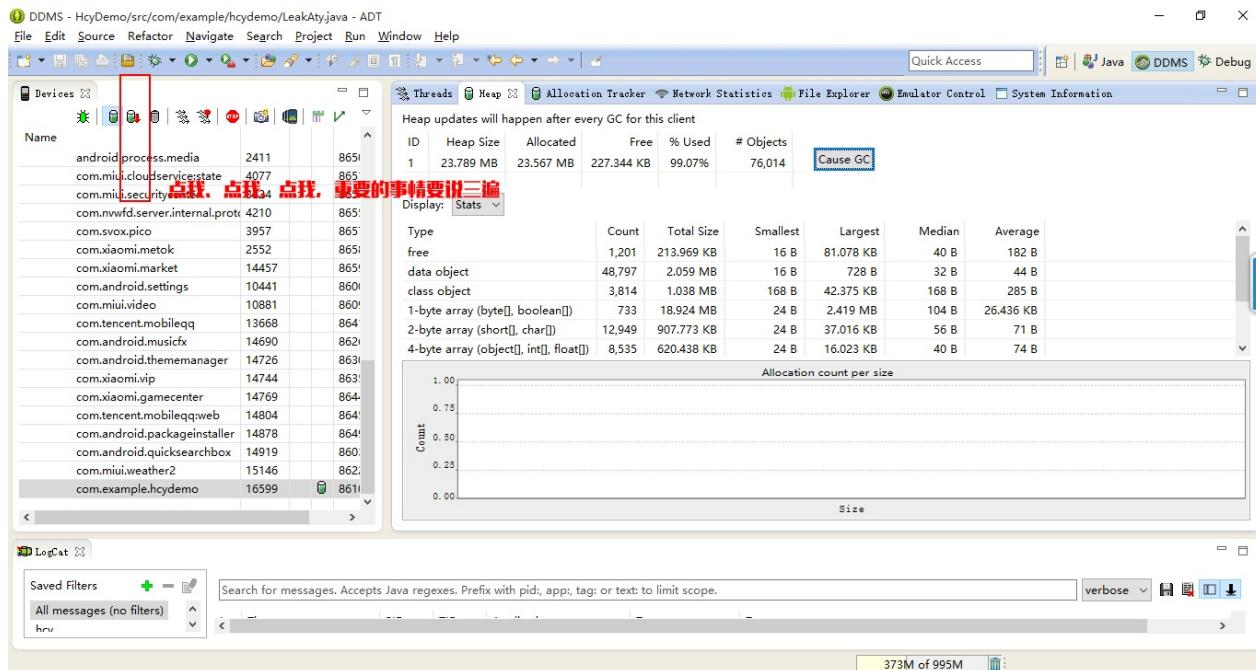
上图的data object的Total Size的大小为1.031M。经过多次的旋转屏幕之后，我们再看一下下截图



Total Size变成了2.059M，从1.031M到2.059M，每次调用GC的过程中data object的总大小没有回落，所以可以证实上面的代码确实是存在内存泄漏的问题，那么泄漏发生在哪里？答案可以通过MAT工具来分析得到。

### 三、内存泄漏的分析工具MAT

要通过MAT分析，需要提供一个.hprof文件。我们可以通过“Dump HPROF file”按钮转存当前的堆内存信息。我们将其保存为1.hprof。



导出的1.hprof的格式需要通过..\\sdk\\tools\\目录下的hprof-conv.exe工具进行转换才能被MAT成功导入，我们将其转换成out1.hprof

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

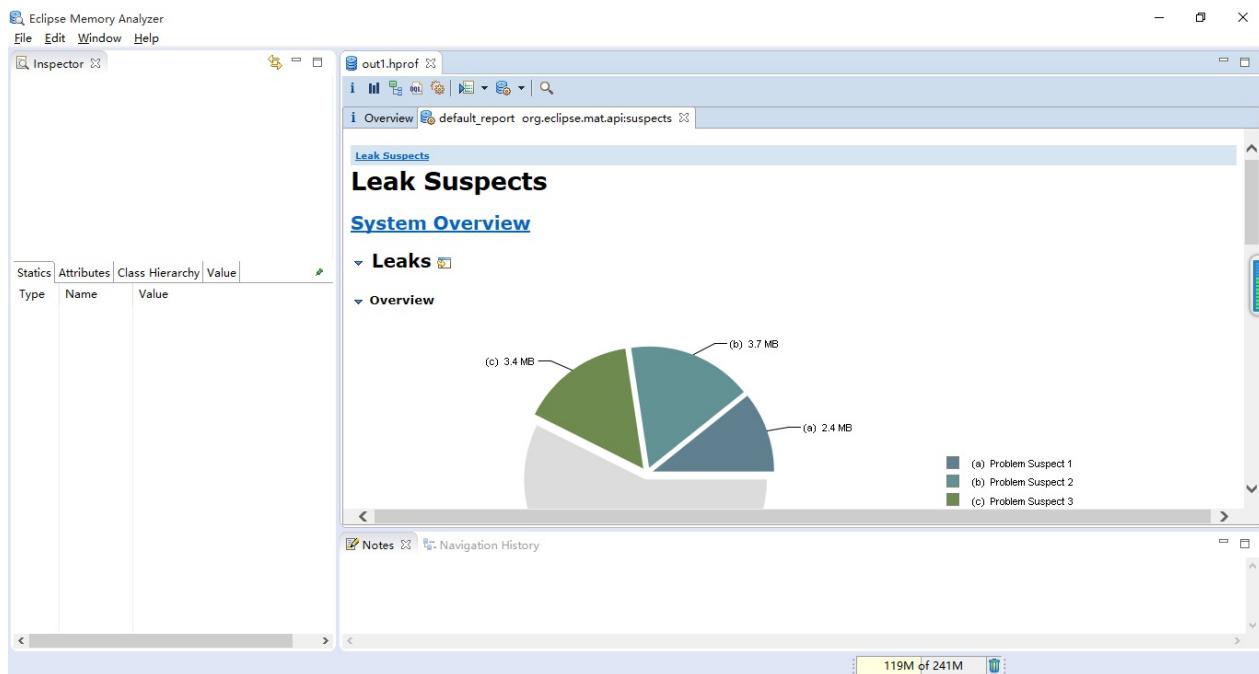
C:\Users\HuangCaiyuan>d:

D:>cd D:\Android\Soft\adt-bundle-windows-x86_64-20131030\sdk\tools

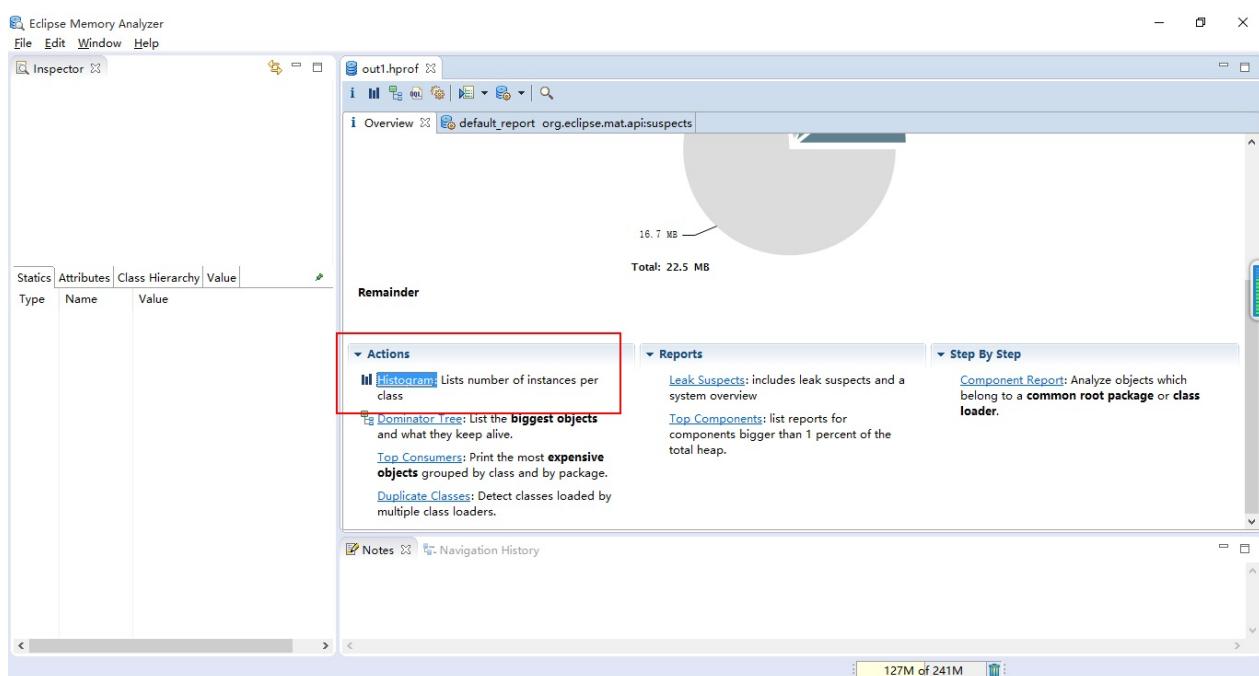
D:\Android\Soft\adt-bundle-windows-x86_64-20131030\sdk\tools>hprof-conv C:\Users\HuangCaiyuan\Desktop\内存泄漏分析\1.hprof C:\Users\HuangCaiyuan\Desktop\内存泄漏分析\out1.hprof

D:\Android\Soft\adt-bundle-windows-x86_64-20131030\sdk\tools>
```

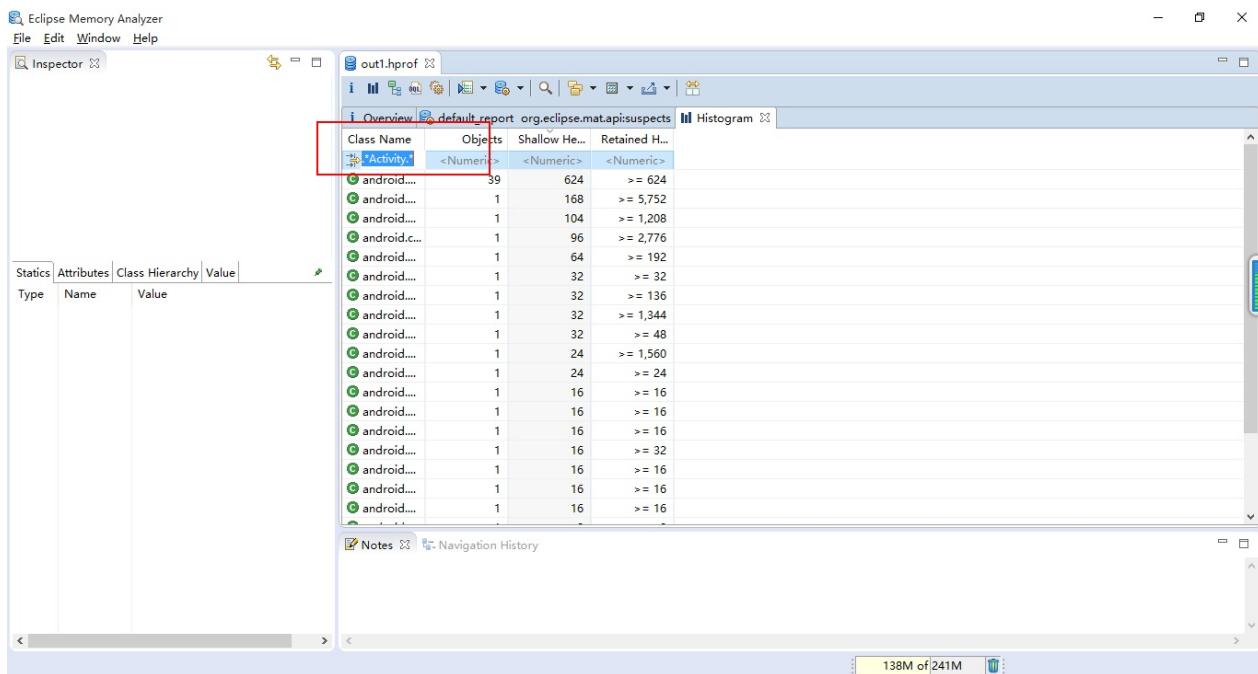
将out1.hprof导入到MAT工具中,File->Open Heap Dump...



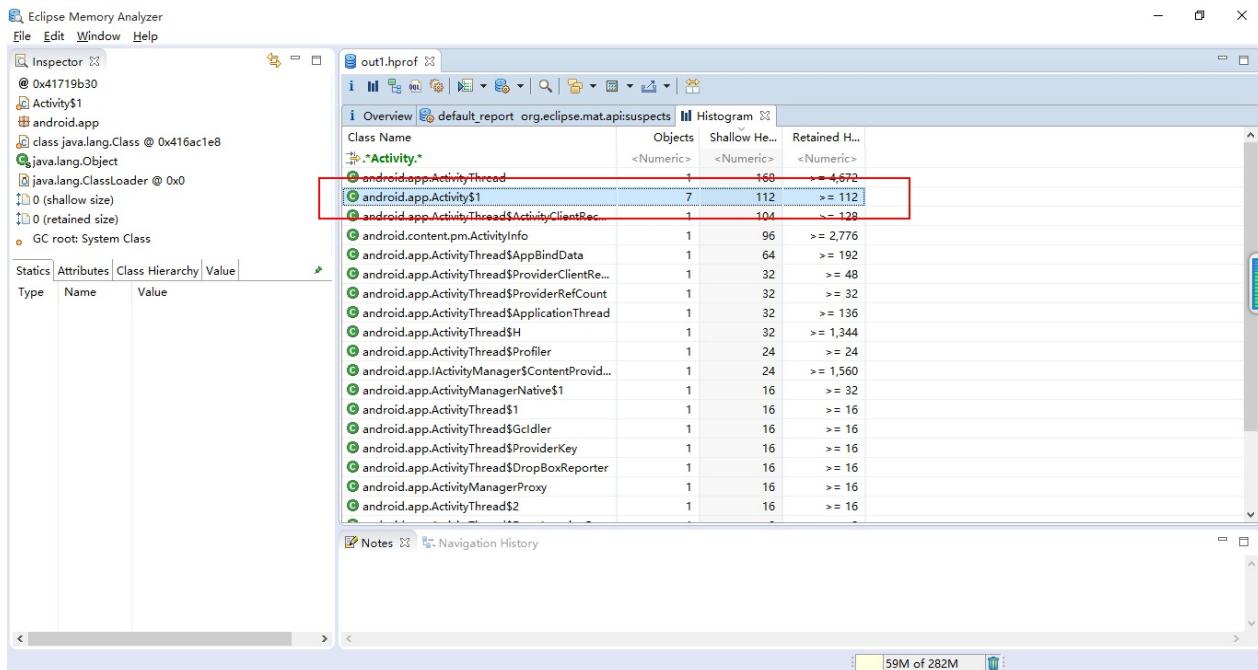
点击左边的标签Overview,Actions->Histogram[直方图]



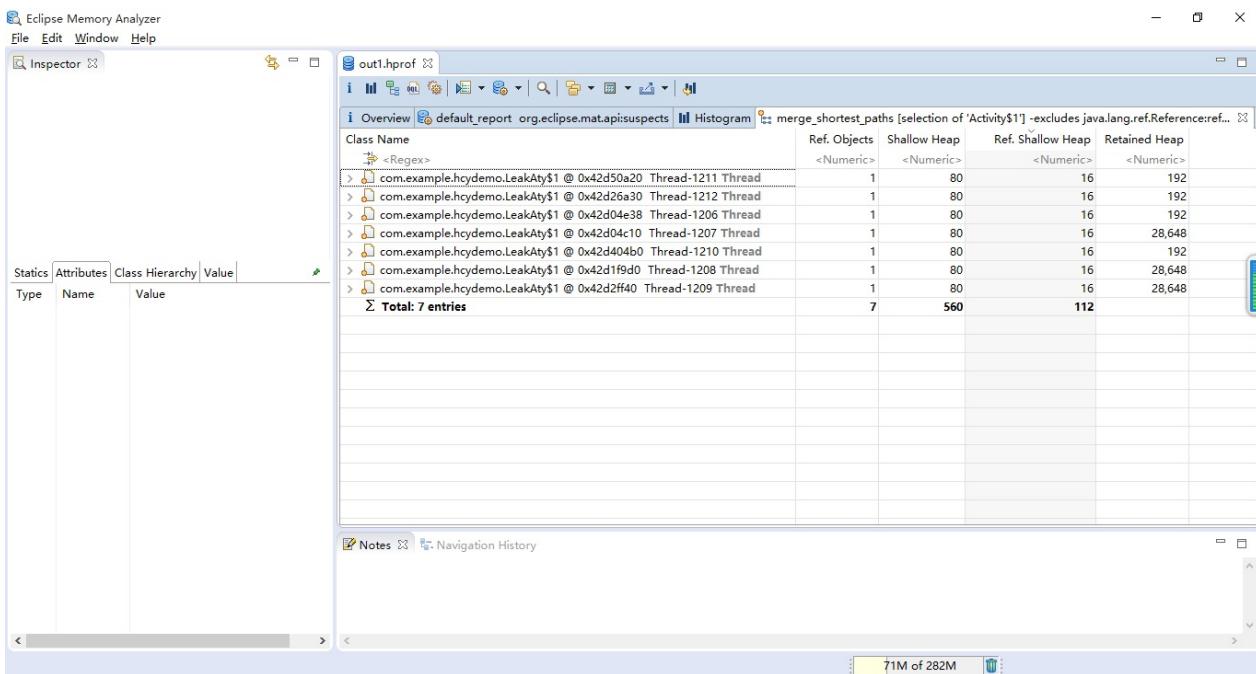
在Histogram界面中，因为我们想要知道Activity是否泄漏了，所以输入关键词Activity,然后按下回车键。



之后便可以得到Activity的相关的搜索结果,下图的搜索结果中Activity的实例有7个。点击选中下图标红色框框的地方,右键->Merge Shortest Paths to GC Roots->exclude all phantom/weak/soft etc. references。排除虚引用、弱引用、软引用的实例,剩下的都是强引用实例。



从过滤出来的强引用的列表中,我们可以看到这七个实例都是被Thread所引用了。所以证实上面的代码确实存在内存泄漏。



## 四、本文总结

内存泄漏检测可以使用Heap工具，内存分析可以使用MAT工具。本文的案例中提到了一种内存泄漏的情况，就是非静态内部类的对象会强引用其外围对象，一旦这个非静态内部类的实例没有释放，它的外围对象也不会释放，所以就会造成内存泄漏。下篇将具体探讨一下，在Android的开发过程中，哪些写法容易造成内存泄漏，该如何解决？请阅读[Android内存泄漏终极解决篇（下）](#)。

## 五、附件

MAT工具下载见：[MAT下载地址](#)

-----

# Android内存泄漏终极解决篇 下

来源:[Android内存泄漏终极解决篇 下](#)

## 一、概述

在 [Android内存泄漏终极解决篇（上）](#) 中我们介绍了如何检查一个App是否存在内存泄漏的问题，本篇将总结典型的内存泄漏的代码，并给出对应的解决方案。内存泄漏的主要问题可以分为以下几种类型：

- 静态变量引起的内存泄漏
- 非静态内部类引起的内存泄漏
- 资源未关闭引起的内存泄漏

## 二、静态变量引起的内存泄漏

在java中静态变量的生命周期是在类加载时开始，类卸载时结束。换句话说，在android中其生命周期是在进程启动时开始，进程死亡时结束。所以在程序的运行期间，如果进程没有被杀死，静态变量就会一直存在，不会被回收掉。如果静态变量强引用了某个Activity中变量，那么这个Activity就同样也不会被释放，即便是该Activity执行了onDestroy(不要将执行onDestroy和被回收划等号)。这类问题的解决方案为：

- 1. 寻找与该静态变量生命周期差不多的替代对象。
- 2. 若找不到，将强引用方式改成弱引用。

比较典型的例子如下：

- 单例引起的Context内存泄漏

```
public class IMManager {
 private Context context;
 private static IMManager mInstance;

 public static IMManager getInstance(Context context) {
 if (mInstance == null) {
 synchronized (IMManager.class) {
 if (mInstance == null)
 mInstance = new IMManager(context);
 }
 }
 return mInstance;
 }

 private IMManager(Context context) {
 this.context = context;
 }
}
```

当调用getInstance时，如果传入的context是Activity的context。只要这个单例没有被释放，这个Activity也不会被释放。

## 解决方案

传入Application的context,因为Application的context的生命周期比Activity长，可以理解为Application的context与单例的生命周期一样长，传入它是最合适的。

```
public class IMManager {
 private Context context;
 private static IMManager mInstance;

 public static IMManager getInstance(Context context) {
 if (mInstance == null) {
 synchronized (IMManager.class) {
 if (mInstance == null)
 //将传入的context转换成Application的context
 mInstance = new IMManager(context.getApplicationContext());
 }
 }
 return mInstance;
 }

 private IMManager(Context context) {
 this.context = context;
 }
}
```

## 三、非静态内部类引起的内存泄漏

在java中，创建一个非静态的内部类实例，就会引用它的外围实例。如果这个非静态内部类实例做了一些耗时的操作，就会造成外围对象不会被回收，从而导致内存泄漏。这类问题的解决方案为：

- 1.将内部类变成静态内部类
- 2.如果有强引用Activity中的属性，则将该属性的引用方式改为弱引用。
- 3.在业务允许的情况下，当Activity执行onDestory时，结束这些耗时任务。
- 内部线程造成的内存泄漏

```
public class LeakAty extends Activity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.aty_leak);
 test();
 }

 public void test() {
 //匿名内部类会引用其外围实例LeakAty.this,所以会导致内存泄漏
 new Thread(new Runnable() {

 @Override
 public void run() {
 while (true) {
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
 }).start();
 }
}
```

## 解决方案

将非静态匿名内部类修改为静态匿名内部类

```
public class LeakAty extends Activity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.aty_leak);
 test();
 }
 //加上static, 变成静态匿名内部类
 public static void test() {
 new Thread(new Runnable() {

 @Override
 public void run() {
 while (true) {
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
 }).start();
 }
}
```

- Handler引起的内存泄漏

```
public class LeakAty extends Activity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.aty_leak);
 fetchData();

 }

 private Handler mHandler = new Handler() {
 public void handleMessage(android.os.Message msg) {
 switch (msg.what) {
 case 0:
 // 刷新数据
 break;
 default:
 break;
 }
 };
 };

 private void fetchData() {
 //获取数据
 mHandler.sendMessage(0);
 }
}
```

mHandler 为匿名内部类实例，会引用外围对象LeakAty.this,如果该Handler在Activity退出时依然还有消息需要处理，那么这个Activity就不会被回收。

## 解决方案

```
public class LeakAty extends Activity {
 private TextView tvResult;
 private MyHandler handler;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.aty_leak);
 tvResult = (TextView) findViewById(R.id.tvResult);
 handler = new MyHandler(this);
 fetchData();

 }
 //第一步，将Handler改成静态内部类。
 private static class MyHandler extends Handler {
 //第二步，将需要引用Activity的地方，改成弱引用。
 private WeakReference<LeakAty> atyInstance;
 public MyHandler(LeakAty aty) {
 this.atyInstance = new WeakReference<LeakAty>(aty);
 }

 @Override
 public void handleMessage(Message msg) {
 super.handleMessage(msg);
 LeakAty aty = atyInstance == null ? null : atyInstance.get();
 //如果Activity被释放回收了，则不处理这些消息
 if (aty == null || aty.isFinishing()) {
 return;
 }
 aty.tvResult.setText("fetch data success");
 }
 }

 private void fetchData() {
 // 获取数据
 handler.sendEmptyMessage(0);
 }

 @Override
 protected void onDestroy() {
 //第三步，在Activity退出的时候移除回调
 super.onDestroy();
 handler.removeCallbacksAndMessages(null);
 }
}
```

## 四、资源未关闭引起的内存泄漏

当使用了BroadcastReceiver、Cursor、Bitmap等资源时，当不需要使用时，需要及时释放掉，若没有释放，则会引起内存泄漏。

## 五、总结

综上所述，内存泄漏的主要情况为上面的三大类型，最终归结为一点，就是资源在不需要的时候没有被释放掉。所以在编码的过程中要注意这些细节，提高程序的性能。

# Android客户端性能优化（魅族资深工程师毫无保留奉献）

来源:[听云](#)

本文由魅族科技有限公司资深Android开发工程师degao（嵌入式企鹅圈原创团队成员）撰写，是degao在嵌入式企鹅圈发表的第一篇原创文章，毫无保留地总结分享其在领导魅族多个项目开发中的Android客户端性能优化经验，极具实践价值！

众所周知，一个好的产品，除了功能强大，好的性能也必不可少。有调查显示，近90%的受访者会因为APP性能差而卸载，性能也是造成APP用户沮丧的头号原因。

那Android客户端性能的指标都有哪些？如何发现和定位客户端的性能问题？本文结合多个项目的开发实践，给出了要关注的重要指标项目，以及定位和解决性能问题的一般步骤。

性能优化应该贯穿于功能开发的全部周期，而不是做完一次后面便不再关注。每次发布版本前，最好能对照标准检查下性能是否达标。

记住：产品=性能×功能！

## 一、 性能检查项

### 1. 启动速度

- 1) 这里的启动速度指的是冷启动的速度，即杀掉应用后重新启动的速度，此项主要是和你的竞品对比。
- 2) 不应在Application以及Activity的生命周期回调中做任何费时操作，具体指标大概是你在onCreate, onResume, onStart等回调中所花费的总时间最好不要超过400ms，否则用户在桌面点击你的应用图标后，将感觉到明显的卡顿。

### 2. 界面切换

- 1) 应用操作时，界面和动画不应有明显卡顿；
- 2) 可通过在手机上打开 设置->开发者选项->调试GPU过度绘制，然后操作应用查看gpu是否超线进行初步判断；



### 3. 内存泄露

- 1) back 退出不应存在内存泄露，简单的检查办法是在退出应用后，用命令 `adb shell dumpsys meminfo 应用包名` 查看 `Activities Views` 是否为零；
- 2) 多次进入退出后的占用内存 `TOTAL` 不应变化太大；

| Applications Memory Usage (kB):                     |              |                  |                  |                  |              |               |
|-----------------------------------------------------|--------------|------------------|------------------|------------------|--------------|---------------|
| Uptime: 35361199 Realtime: 40390465                 |              |                  |                  |                  |              |               |
| ** MEMINFO in pid 22390 [com.meizu.media.reader] ** |              |                  |                  |                  |              |               |
|                                                     | Pss<br>Total | Private<br>Dirty | Private<br>Clean | Swapped<br>Dirty | Heap<br>Size | Heap<br>Alloc |
| Native Heap                                         | 0            | 0                | 0                | 0                | 32768        | 15321         |
| Dalvik Heap                                         | 16176        | 15568            | 0                | 0                | 33753        | 31737         |
| Dalvik Other                                        | 625          | 624              | 0                | 0                |              |               |
| Stack                                               | 597          | 596              | 0                | 0                |              |               |
| Ashmem                                              | 5            | 0                | 0                | 0                |              |               |
| Other dev                                           | 4            | 0                | 4                | 0                |              |               |
| .so mmap                                            | 1574         | 276              | 708              | 0                |              |               |
| .apk mmap                                           | 425          | 0                | 220              | 0                |              |               |
| .ttf mmap                                           | 176          | 0                | 36               | 0                |              |               |
| .dex mmap                                           | 4728         | 0                | 4728             | 0                |              |               |
| .oat mmap                                           | 1441         | 0                | 304              | 0                |              |               |
| .art mmap                                           | 1730         | 1188             | 96               | 0                |              |               |
| Other mmap                                          | 20           | 4                | 0                | 0                |              |               |
| Unknown                                             | 15524        | 15452            | 0                | 0                |              |               |
| TOTAL                                               | 43025        | 33708            | 6096             | 0                | 66521        | 47058         |

| Objects           |    |                  |    |
|-------------------|----|------------------|----|
| Views:            | 0  | ViewRootImpl:    | 0  |
| AppContexts:      | 2  | Activities:      | 0  |
| Assets:           | 4  | AssetManagers:   | 4  |
| Local Binders:    | 16 | Proxy Binders:   | 24 |
| Parcel memory:    | 11 | Parcel count:    | 46 |
| Death Recipients: | 0  | OpenSSL Sockets: | 0  |

## 4. onTrimMemory回调

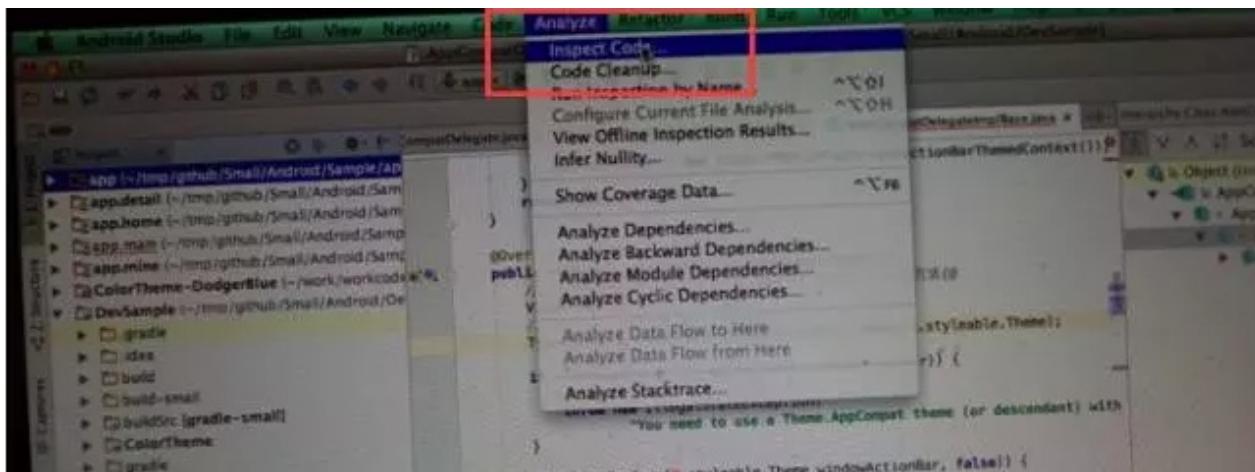
- 1) 应用响应此回调释放非必须内存；
- 2) 验证可通过命令 `adb shell dumpsys gfxinfo` 应用包名-cmd trim 5 后，再）用命令 `adb shell dumpsys meminfo` 应用包名 查看内存大小。

## 5. 过度绘制

打开设置中的GPU过度绘制开关，各界面过度绘制不应超过2.5x；也就是打开此调试开关后，界面整体呈现浅色，特别复杂的界面，红色区域也不应该超过全屏幕的四分之一；

## 6. lint检查

- 1) 通过Android Studio中的 Analyze->Inspect Code 对工程代码做静态扫描；找出潜在的问题代码并修改；
- 2) 0 error & 0 warning，如果确实不能解决，需给出原因。



## 7. 反射优化

- 1) 在代码中减少反射调用；
- 2) 对频繁调用的返回值进行Cache；

## 8. 稳定性

- 1) 连续48小时monkey不应出现闪退，anr问题。
- 2) 如果应用接入了数据埋点的sdk，比如百度统计sdk等，这些sdk都会将应用的崩溃信息上报回来，开发者应每天关注这些统计到的崩溃日志，严格控制应用的崩溃率；

## 9. 耗电

- 1) 应用进入后台后不应异常消耗电量；
- 2) 操作应用后，退出应用，让应用处于后台，一段时间后通过 `adb shell dumpsys batterystats` 查看电量消耗日志看是否存在异常。

# 二、性能问题常见原因

性能问题一般归结为三类：

- 1.UI卡顿和稳定性:这类问题用户可直接感知，最为重要；
- 2.内存问题：内存问题主要表现为内存泄露，或者内存使用不当导致的内存抖动。如果存在内存泄露，应用会不断消耗内存，易导致频繁gc使系统出现卡顿，或者出现OOM报错；内存抖动也会导致UI卡顿。
- 3.耗电问题：会影响续航，表现为不必要的自启动，不恰当持锁导致系统无法正常休眠，系统休眠后频繁唤醒系统等；

## 三、UI卡顿常见原因和分析方法

下面分别介绍出现这些问题的常见原因以及分析这些问题的一般步骤。

### 1. 卡顿常见原因

- 1) 人在UI线程中做轻微耗时操作，导致UI线程卡顿；
- 2) 布局Layout过于复杂，无法在16ms内完成渲染；
- 3) 同一时间动画执行的次数过多，导致CPU或GPU负载过重；
- 4) View过度绘制，导致某些像素在同一帧时间内被绘制多次，从而使CPU或GPU负载过重；
- 5) View频繁的触发measure、layout，导致measure、layout累计耗时过多及整个View频繁的重新渲染；
- 6) 内存频繁触发GC过多（同一帧中频繁创建内存），导致暂时阻塞渲染操作；
- 7) 冗余资源及逻辑等导致加载和执行缓慢；
- 8) 工作线程优先级未设置为

Process.THREAD\_PRIORITY\_BACKGROUND 导致后台线程抢占UI线程cpu时间片，阻塞渲染操作；

- 9) ANR；

### 2. 卡顿分析解决的一般步骤

- 1) 解决过度绘制问题

在设置->开发者选项->调试GPU过度绘制中打开调试，看对应界面是否有过度绘制，如果有先解决掉：



定位过渡绘制区域

利用Android提供的工具进行位置确认以及修改(HierarchyView , Tracer for OpenGL ES)

定位到具体的视图(xml文件或者View)

通过代码和xml文件分析过渡绘制的原因

结合具体情况进行优化

使用Lint工具进一步优化

## 2) 检查是否有主线程做了耗时操作：

严苛模式（StrictMode），是Android提供的一种运行时检测机制，用于检测代码运行时的一些不规范的操作，最常见的场景是用于发现主线程的IO操作。应用程序可以利用StrictMode尽可能的发现一些编码的疏漏。

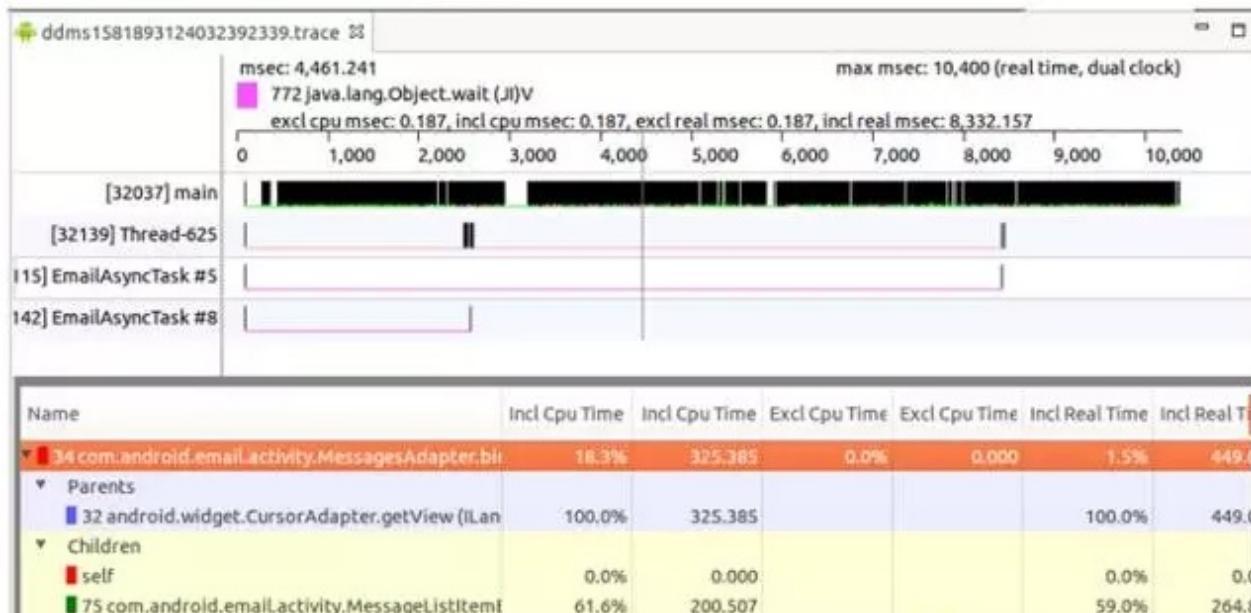
- 开启 StrictMode
- 对于应用程序而言，Android 提供了一个最佳使用实践：尽可能早的在 `android.app.Application` 或 `android.app.Activity` 的生命周期使能 StrictMode，`onCreate()`方法就是一个最佳的时机，越早开启就能在更多的代码执行路径上发现违规操作。
- 监控代码

```
public void onCreate() {
 if (DEVELOPER_MODE) {
 StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
 .detectAll() .penaltyLog() .build());
 StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
 .detectAll() .penaltyLog() .build());
 }
 super.onCreate();
}
```

如果主线程有网络或磁盘读写等操作，在logcat中会有"D/StrictMode"tag的日志输出，从而定位到耗时操作的代码。

- 3) 如果主线程无耗时操作，还存在卡顿，有很大可能是必须在UI线程操作的一些逻辑有问题，比如控件measure、layout耗时过多等，此时可通过Traceview以及systrace来进行分析。
- 4) Traceview：Traceview主要用做热点分析，找出最需要优化的点。

打开DDMS然后选择一个进程，接着点击上面的“Start Method Profiling”按钮（红色小点变为黑色即开始运行），然后操作我们的卡顿UI,然后点击"Stop Method Profiling",会打开如下界面：



图中展示了Trace期间各方法调用关系，调用次数以及耗时比例。通过分析可以找出可疑的耗时函数并进行优化；

### 5) systrace: 抓取trace:

执行如下命令：

```
$ cd android-sdk/platform-tools/systrace
$ python systrace.py --time=10 -o mynewtrace.html sched gfx view wm
> 操作APP, 然后会生成一个mynewtrace.html 文件, 用Chrome打开。
```

图示如下：



通过分析上面的图，可以找出明显存在的layout, measure, draw的超时问题。

### 6) 导入如下插件，可通过在方法上添加@DebugLog来打印方法的耗时：

build.gradle:

```
buildscript {
 dependencies {
```

//用于方便调试性能问题的打印插件。给访法加上@DebugLog，就能输出该方法的调用参数，以及执行时间；

```
 classpath 'com.jakewharton.hugo:hugo-plugin:1.2.1'
 }
}
```

//用于方便调试性能问题的打印插件。给访法加上@DebugLog，就能输出该方法的调用参数，以及执行时间；

```
apply plugin: 'com.jakewharton.hugo'
```

java：

```
@DebugLog public void test(int a){ int b=a*a; }
```

```
四、内存性能分析优化
```

```
1. 内存泄露
```

该问题目前在项目中一般用leakcanary基本就能搞定，配置起来也相当简单：

```
build.gradle: dependencies { debugCompile 'com.squareup.leakcanary:leakcanary-android:1.3.1' // or 1.4-beta1 releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.3.1' // or 1.4-beta1 testCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.3.1' // or 1.4-beta1 }
```

```
java: public class ExampleApplication extends Application {
```

```
 @Override public void onCreate() { super.onCreate(); LeakCanary.install(this); } }
```

一旦有内存泄露，将会在通知栏生成一条通知，点开可看到泄露的对象以及引用路径：

```

```

### ### 2. 内存抖动

如果代码中存在在`onDraw`或者`for`循环等多次执行的代码中分配对象的行为，会导致运行过程中gc次数增多，

### ## 五、耗电量优化建议

电量优化主要是注意尽量不要影响手机进入休眠，也就是正确申请和释放WakeLock，另外就是不要频繁唤醒手机，

### ## 六、一些好的代码实践

- \* 节制地使用Service

- \* 当界面不可见时释放内存

- \* 当内存紧张时释放内存

- \* 避免在Bitmap上浪费内存

对大图片，先获取图片的大小信息，根据实际需要展示大小计算inSampleSize，最后decode；

```
public static Bitmap decodeSampledBitmapFromFile(String filename, int reqWidth, int
reqHeight) { // First decode with inJustDecodeBounds=true to check dimensions final
BitmapFactory.Options options = new BitmapFactory.Options();
options.inJustDecodeBounds = true; BitmapFactory.decodeFile(filename, options);
```

```
// Calculate inSampleSize
options.inSampleSize = calculateInSampleSize(options, reqWidth, reqHeight);
// Decode bitmap with inSampleSize set
options.inJustDecodeBounds = false;
return BitmapFactory.decodeFile(filename, options);
```

```
}
```

```
public static int calculateInSampleSize(BitmapFactory.Options options, int reqWidth, int
reqHeight) { // Raw height and width of image final int height = options.outHeight; final int
width = options.outWidth; int inSampleSize = 1; if (height > reqHeight || width >
reqWidth) { if (width > height) { inSampleSize = Math.round((float) height / (float)
reqHeight); } else { inSampleSize = Math.round((float) width / (float) reqWidth); } } return
inSampleSize; }
```

- \* 使用优化过的数据集合

- \* 谨慎使用抽象编程

- \* 尽量避免使用依赖注入框架

很多依赖注入框架是基于反射的原理，虽然可以让代码看起来简洁，但是是有碍性能的。

- \* 谨慎使用external libraries

- \* 优化整体性能

- \* 使用ProGuard来剔除不需要的代码

```
android { buildTypes { release { minifyEnabled true shrinkResources true proguardFiles getDefaultProguardFile('proguard-android.txt'), 'src/main/proguard-project.txt' signingConfig signingConfigs.debug } } }
```

- \* 慎用异常，异常对性能不利

抛出异常首先要创建一个新的对象。`Throwable`接口的构造函数用名为`fillInStackTrace()`的本地方法，`1`

以下例子不好：

```
try { startActivity(intentA); } catch () { startActivity(intentB); }
```

应该用下面的语句判断：

```
if (getPackageManager().resolveActivity(intentA, 0) != null) ...
```

不要再循环中使用 try/catch 语句，应把其放在最外层，使用 System.arraycopy() 代替 for 循环复制。

更多Android、Linux、嵌入式和物联网原创技术分享敬请关注微信公众号：嵌入式企鹅圈。



# Android性能分析工具整理汇总

来源:[D\\_clock的简书](#)

把做Android开发以来碰到的一些不错的性能分析工具做个整理汇总...

## Debug GPU Overdraw

**类型：**系统自带功能UI渲染检测功能（打开Settings，然后到 Developer Options -> Debug GPU Overdraw 选择 Show overdraw areas，手机系统设置中文的孩纸，自行对照翻译进去哈）

**作用：**用来检测UI的重绘次数，开发者可以用来优化UI的性能。

**使用心得：**检测UI性能的利器，对于开发者做UI优化的帮助挺大的。因为大量的重绘容易让app造成卡顿或者直接导致丢帧的现象。开发者熟悉View的绘制原理可以结合对一些布局或者自定义控件做相应的优化。诸如：在ListView或GridView里面的item使用 layout\_weight设置就会造成多余重绘。其他情况还有很多，不一一例举。至于怎么用，可以自行Google

## Profile GPU Rendering

**类型：**系统自带功能UI渲染检测功能（打开Settings，然后到 Developer Options -> Profile GPU Rendering. 选择 On screen as bars ）

**作用：**用来检测UI绘制帧的速率和耗时，同样开发者可以用来优化UI的性能。

**使用心得：**跟Debug GPU Overdraw功能类似，但它反应的是UI绘制帧的速率，同样可以用来检测自己的app是否丢帧或者绘制过度，具体操作可以自行Google

## Hierarchy Viewer

**类型：**SDK自带工具（打开Settings，然后到 Developer Options -> Profile GPU Rendering. 选择 On screen as bars ）

**作用：**检测UI渲染用的

使用心得：老牌工具了，Google一下

## **Memory Monitor、Heap Viewer、Allocation Tracker**

类型：AndroidStudio自带的工具

作用：均是内存检测分析的工具

使用心得：不用多说，大家懂的...

## **Memory Analyzer Tool (MAT)**

类型：ADT时代的插件，也有独立的MAT版本

作用：内存详尽分析的神器啊！

使用心得：它是我在ADT下唯一的美好回忆啊，AS现在的工具就差它了，希望快点跟上。为了隆重介绍我的挚爱，果断献上它的官方文

档：<http://help.eclipse.org/mars/index.jsp>

## **Traceview、Systrace**

类型：SDK自带

作用：CPU使用分析的工具

使用心得：排除CPU性能瓶颈的利器，TraceView能让我知道个个函数调用的CPU耗时，以及总CPU耗时等，方便排查优化。Systrace能够让我了解各个AP子模块的使用情况，同样利于瓶颈排查，性能优化工作等，总之，很赞就是了。

## **Battery Historian**

类型：独立开源软件 (Google IO大会上的推荐的工具)

作用：耗电分析工具

**使用心得：**在耗电分析上Google亲自推的东西自然不用说，Battery Historian 1.0的基本使用在网上挺多，可以自行查看。2.0的功能更加perfect，但是国内资料少，国外的资料算还可以，so，翻墙吧，骚年！使用 Battery Historian 需要注意两点，一是它只对5.x及其以上的系统生效，二是搭建环境的时候注意要使用Python2.x的，不要使用Python3.x。因为两个版本的语法变法很大，Python 3.x下Battery Historian会报错。最后，这个是开源项目 <https://github.com/google/battery-historian>

-----分割线-----

上面主要都是官方的工具，下面是一些第三方apk工具...

## WakeLock Detector

**功能简介：**对手机的运行状态进行探测记录，能统计那些应用触发了CPU运行消耗cpu，那些应用触发了屏幕点亮。同时还可以对运行时间进行统计，可以查看应用内使用细节。

**使用心得：**之前做了一款app被用户投诉耗电太快。偶然发现了它，拿做电量损耗检测。同时，它也能够统计其他安装在手机上的app的电量消耗，方便做出对比，向顶级体验的应用看齐。

**使用前提：**手机需要root，该app需要获取root权限

## GSam Battery Monitor

**功能简介：**检测手机电池电量的消耗去向，能够用折线图进行统计展示。

**使用心得：**不错的产品，能够计算出你的电量被手机的哪部分功能所消耗的，可以追溯到这部分功能是哪些app在使用，从而定位到手机耗电过快的元凶。

**使用前提：**手机需要root，该app需要获取root权限

## Trepn Profiler

**功能简介：**高通出品的，杠杠的赞啊！分析检测手机CPU的消耗，而且能够分析特地的分析某个app。

**使用心得：**用来调试分析自己的app，实时的用折线图展示了app对CPU的消费情况，赞赞赞。

**使用前提：**手机需要root，该app需要获取root权限，且只支持手机的CPU是高通的。

## Root Explorer

**功能简介：**一款文件浏览器，可以查看app没有加密过的数据库，读取里面的数据，且支持简单的条件查询。

**使用心得：**在开发的时候，需要确认是否成功把数据插入数据库，有了它就可以直接打开database文件浏览查找了。

**使用前提：**手机需要root，该app需要获取root权限

-----分割线-----

除了上面这些apk工具外，最后是一些知名IT公司开发的工具（包含SDK），很好用...

## Bugly

揪BUG、揪ANR的SDK。腾讯出品的东东，杠杠的。对发布出去的产品你想准确定位各种闪退的BUG，用它准行。而且bugly的更新频率还挺快的，大公司的效率真是任性（只能说鹅厂越来越会用技术赚钱了~）

官网地址：<http://bugly.qq.com/>

## BugTags

官网说的：测试，从未如此简单！新一代的、专为移动测试而生的缺陷发现及管理工具。个人觉得很不错，同样推荐！

官网地址：<https://bugtags.com/>

## GT

这款神器，可能并不多人知道（我猜的）。腾讯MIG专项测试组开发出来的狂拽酷炫屌炸天的神器，只要多神，不多说了，直接进去官网看吧，我已泪奔（腾讯的技术真心叼）

官网地址：<http://gt.tencent.com/index.html>

## iTest

科大讯飞出品的测试工具，直接安装使用。是一款服务于Android测试人员的专业手机自动化性能监控工具。

官网地址：<http://itest.iflytesting.com/?p=1>

## Emmagee

网易出品的测试工具，和iTest差不多，最大的好处在于，能够对应用的常用性能指标进行检测，并以csv的格式保存方便查看应用的各项参数。测试结果看起来更加直观，还有很重要一点是，它开源!!!!

官网地址：<https://github.com/NetEase/Emmagee>

## 待续...

目前大体就这些了，后续有更好的工具也会接着更新，有些工具过时失效了，也会在这里移除...

文章同步归档到此：<https://github.com/D-clock/Doc>

# Android界面性能调优手册

来源:[测试者手册](#)

注：本文是我在 Android 界面性能调优知识的系统性总结，纯属个人碎碎念。秉持开源分享的原则发布本文出来，各位看官有需则取。

追加：本文发布 24 小时，已被开发者头条、推酷、图灵社区以及一些小刊小网转载，并在简书上收获 40 几个喜欢。并打破 [ANDROIDTEST.ORG](#) 博客历史最高的《走过人生的五分之一》文章时的 IP 日活（IP 694），创下新的历史新高（IP 792，「戳我查看」），以及收获了几十元的打赏零花钱，感谢大家的支持！

界面是 Android 应用中直接影响用户体验最关键的部分。如果代码实现得不好，界面容易发生卡顿且导致应用占用大量内存。

我司这类做 ROM 的公司更不一样，预装的应用一定要非常流畅，这样给客户或用户的第一感觉就是快。又卡又慢的应用体验，会影响客户或用户对产品的信心和评价，所以不可忽视。

## 目录

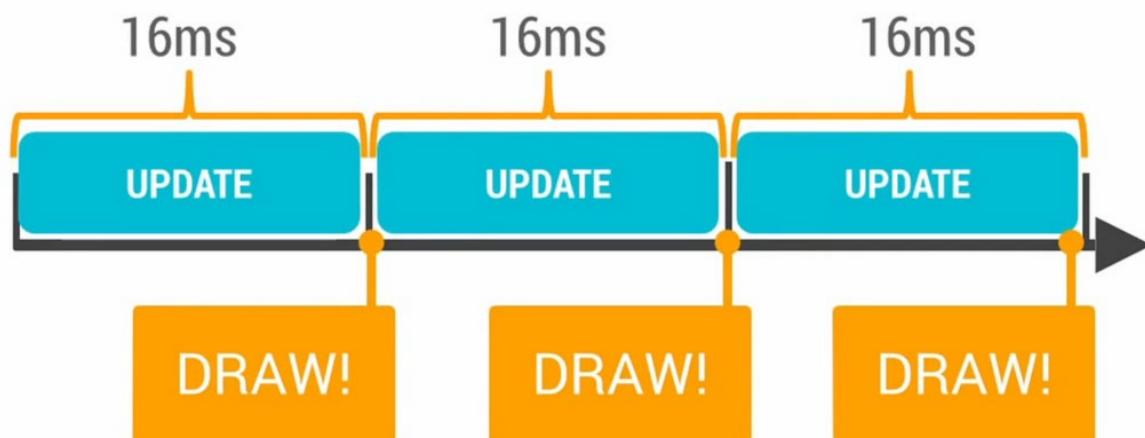
- 一. Android渲染知识
  - 1.1 绘制原理
  - 1.2 掉帧
  - 1.3 为什么是60Fps?
  - 1.4 垃圾回收
  - 1.5 UI 线程
  - 1.6 垂直同步
  - 1.7 UI 绘制机制与栅格化
- 二. 检测和解决
  - 2.1 检测维度
  - 2.2 调试工具
  - 2.3 如何解决
- 三. 界面过度绘制
  - 3.1 过度绘制概念
  - 3.2 追踪过度绘制
  - 3.3 过度绘制的根源

- 3.4 不合理的xml布局对绘制的影响
- 3.5 源码相关
- 四. 渲染性能
  - 4.1 渲染性能概念
  - 4.2 追踪渲染性能
  - 4.3 渲染性能差的根源
  - 4.4 检测说明
  - 4.5 UI绘制机制的补充说明
- 五. 布局边界合理性
- 六. 给开发的界面优化建议
  - 6.1 优化布局的结构
  - 6.2 优化处理逻辑
  - 6.3 善用 DEBUG 工具
- 附录

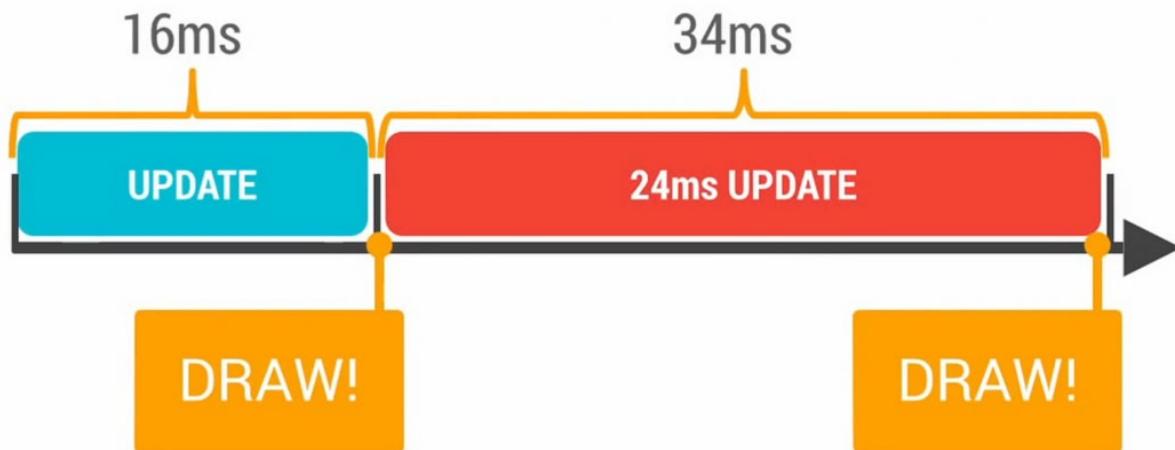
# 一. Android渲染知识

## 1.1 绘制原理

Android系统要求每一帧都要在 16ms 内绘制完成，平滑的完成一帧意味着任何特殊的帧需要执行所有的渲染代码（包括 framework 发送给 GPU 和 CPU 绘制到缓冲区的命令）都要在 16ms 内完成，保持流畅的体验。这个速度允许系统在动画和输入事件的过程中以约 60 帧每秒（ $1\text{秒} / 0.016\text{帧每秒} = 62.5\text{帧/秒}$ ）的平滑帧率来渲染。



如果你的应用没有在 16ms 内完成这一帧的绘制，假设你花了 24ms 来绘制这一帧，那么就会出现掉帧的情况。



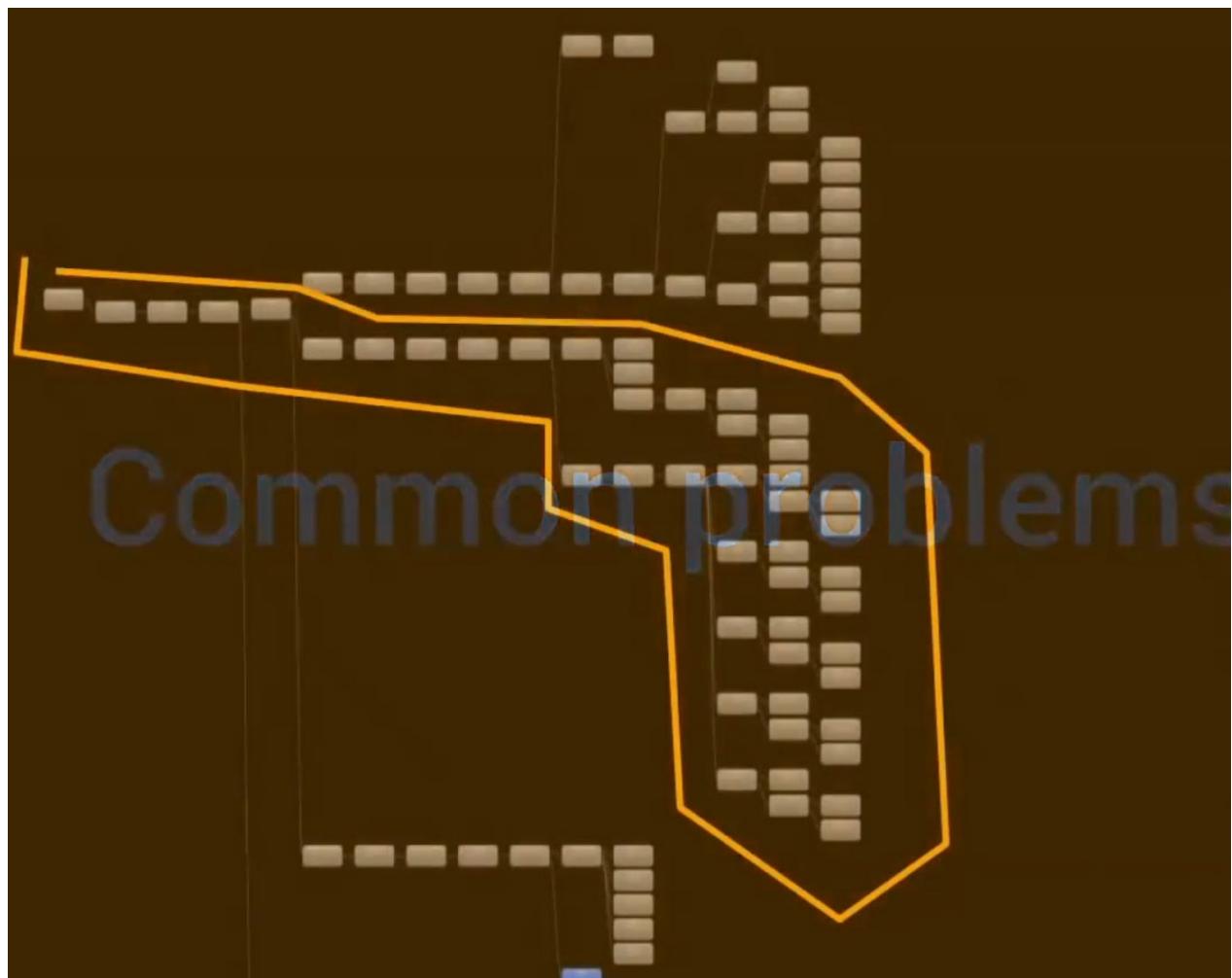
系统准备将新的一帧绘制到屏幕上，但是这一帧并没有准备好，所有就不会有绘制操作，画面也就不会刷新。反馈到用户身上，就是用户盯着同一张图看了 32ms 而不是 16ms，也就是说掉帧发生了。

## 1.2 掉帧

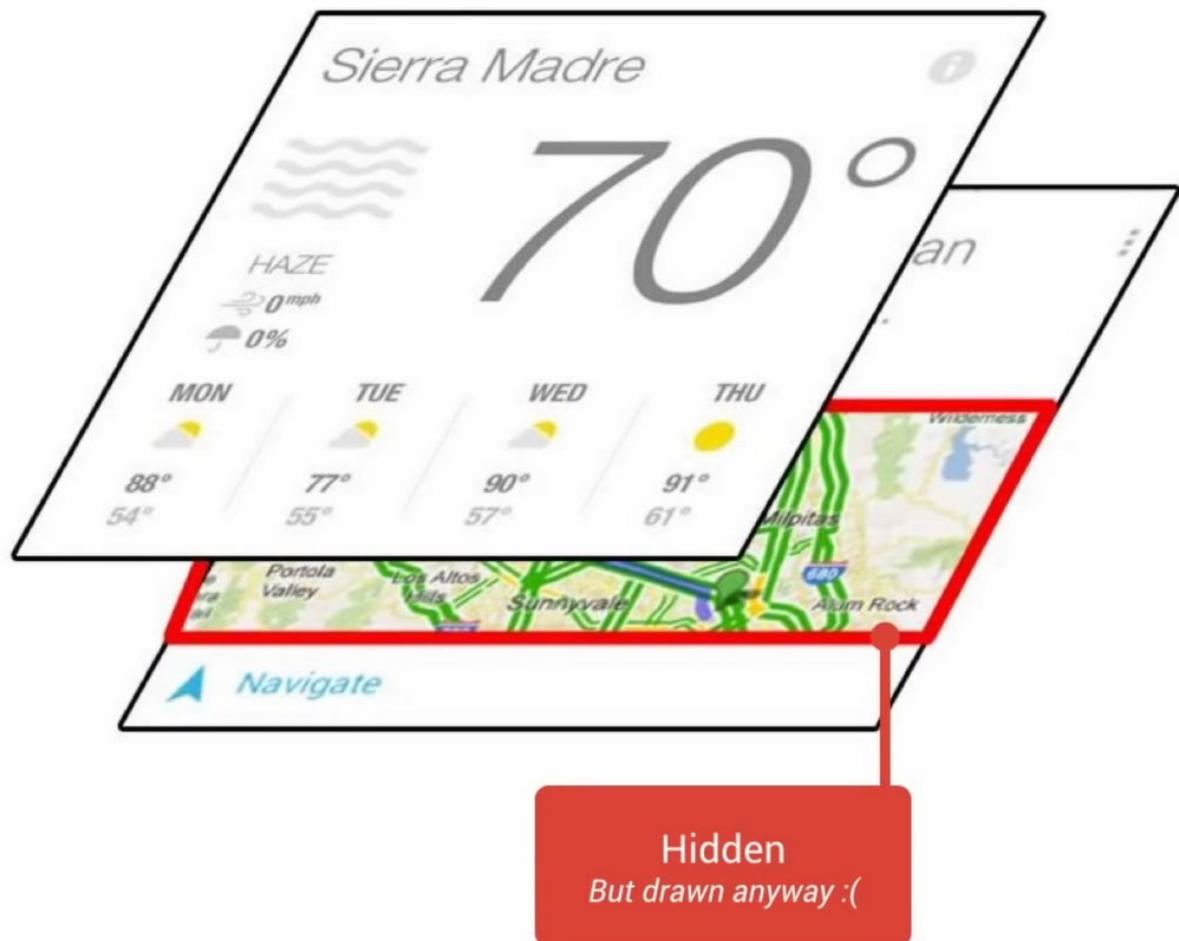
掉帧是用户体验中一个非常核心的问题。丢弃了当前帧，并且之后不能够延续之前的帧率，这种不连续的间隔会容易会引起用户的注意，也就是我们常说的卡顿、不流畅。

引起掉帧的原因非常多，比如：

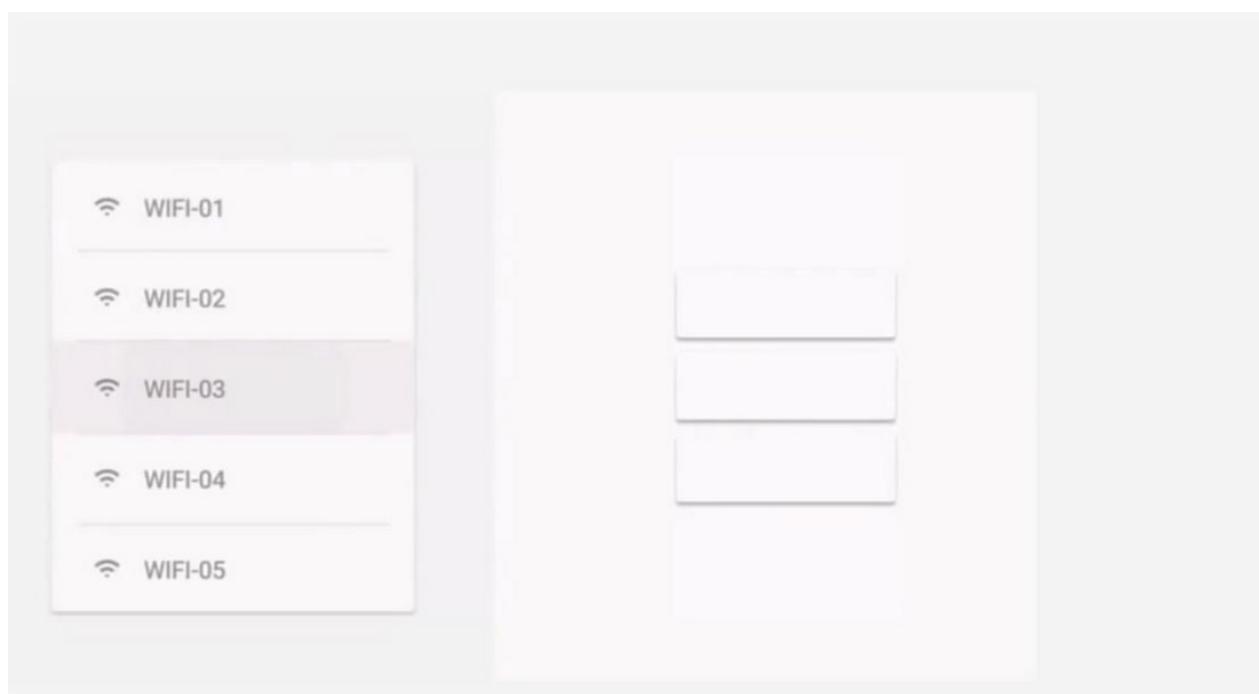
- 花了非常多时间重新绘制界面中的大部分东西，这样非常浪费CPU周期；



- 过度绘制严重，在绘制用户看不到的对象上花费了太多的时间；



- 有一大堆动画重复了一遍又一遍，消耗 CPU 、 GPU 资源；



- 频繁的触发垃圾回收；

## 1.3 为什么是60Fps?

Android系统要求每一帧都要在 16ms 内绘制完成，那么1秒的帧率就是约 60 帧每秒（ $1\text{秒} / 0.016\text{帧每秒} = 62.5\text{帧/秒}$ ），那为什么要以 60 Fps 来作为 App 性能的衡量标准呢？这是因为人眼和大脑之间的协作无法感知到超过 60 Fps 的画面更新。

市面上绝大多数Android设备的屏幕刷新频率是 60 HZ。当然，超过 60 Fps 是没有意义的，人眼感知不到区别。24 Fps 是人眼能感知的连续线性的运动，所以是电影胶圈的常用帧率，因为这个帧率已经足够支撑大部分电影画面所要表达的内容，同时能最大限度地减少费用支出。但是，低于 30 Fps 是无法顺畅表现绚丽的画面内容的，此时就需要用到 60 Fps 来达到想要表达的效果。了解更多Fps知识详见「[Wiki](#)」。

应用的界面性能目标就是保持 60 Fps，这意味着每一帧你只有 16 ms（1秒 / 60帧率）的时间来处理所有的任务。

## 1.4 垃圾回收

垃圾回收器是一个在应用运行期间自动释放那些不再引用的内存的机制，常称 GC。频繁的 GC 也是导致严重性能问题的罪魁祸首之一。

前面提到，平滑的完成一帧意味着所有渲染代码都必须在 16ms 内完成。频繁的 GC 会严重限制一帧时间内的剩余时间，如果 GC 所做的工作超过了那些必须的工作，那么留给应用平滑的帧率的时间就越少。越接近 16ms，在垃圾回收事件触发的时候，就越容易导致卡顿。

注意，Android4.4 引进了新的 ART 虚拟机来取代 Dalvik 虚拟机。它们的机制大有不同，简单而言：

- Dalvik 虚拟机的 GC 是非常耗资源的，并且在正常的情况下一个硬件性能不错的 Android设备也会很容易耗费掉 10 - 20 ms 的时间；
- ART 虚拟机的GC会动态提升垃圾回收的效率，在 ART 中的中断，通常在 2 - 3 ms 间。比 Dalvik 虚拟机有很大的性能提升；

ART 虚拟机相对于 Dalvik 虚拟机来说的垃圾回收来说有一个很大的性能提升，但 2 - 3 ms 的回收时间对于超过16ms帧率的界限也是足够的。因此，尽管垃圾回收在 Android 5.0 之后不再是耗资源的行为，但也是始终需要尽可能避免的，特别是在执行动画的情况下，可能会导致一些让用户明显感觉的丢帧。

想了解更多详细的 ART 和 Dalvik 虚拟机垃圾回收机制，可「[戳我](#)」和「[我](#)」进行深入了解。

## 1.5 UI 线程

UI 线程是应用的主线程，很多的性能和卡顿问题是由于我们在主线程中做了大量的工作。

所以，所有耗资源的操作，比如 IO 操作、网络操作、SQL 操作、列表刷新等，都应该用后台进程去实现，不能占用主线程，主线程是 UI 线程，是保持程序流畅的关键；

在 Android 5.0 版本里，Android 框架层引入了“ Render Thread ”，用于向 GPU 发送实际渲染的操作。这个线程减轻了一些 UI 线程减少的操作。但是输入、滚动和动画仍然在 UI thread，因为 Thread 必须能够响应操作。

## 1.6 垂直同步

垂直同步是 Android4.1 通过 Project Butter 在 UI 架构中引入的新技术，同期引入的还有 Triple Buffer 和 HWComposer 等技术，都是为提高 UI 的流畅性而生。

举个例子，你拍了一张照片，然后旋转5度再拍另外一张照片，将两照片的中间剪开并拼接在一起，得到下图：



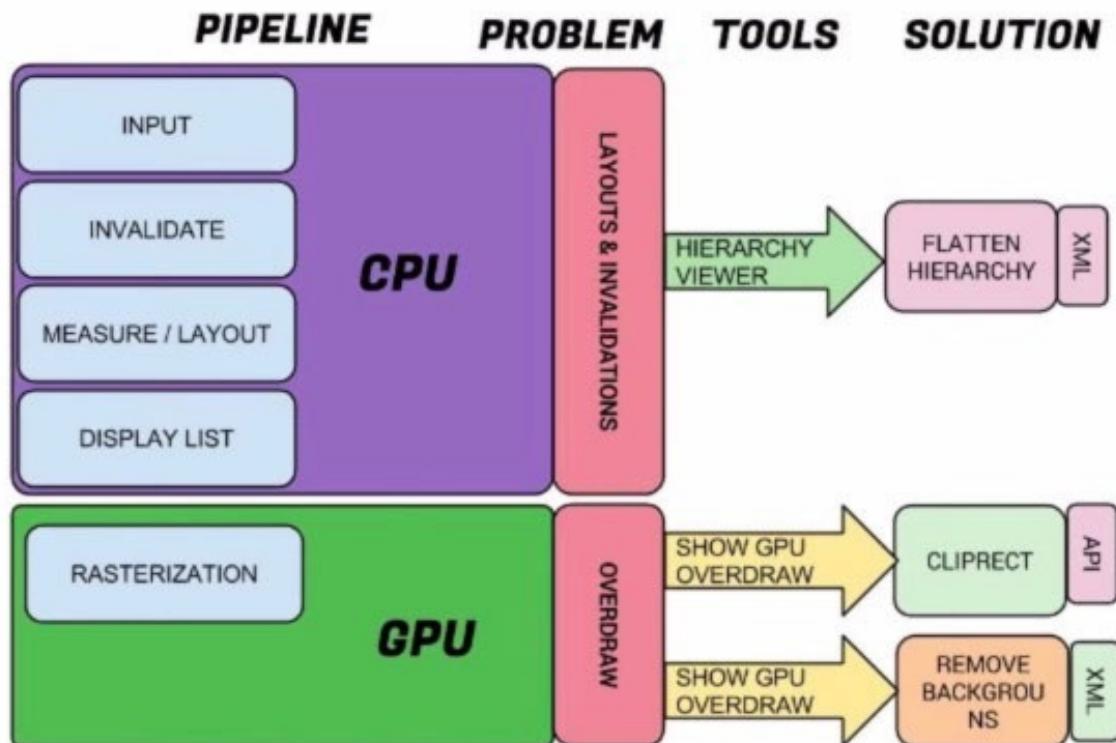
中间这部分有明显区别的部分，等价于设备刷新率和帧速率不一致的结果。

一般而言，GPU的帧速率应高于刷新率，才不会卡顿或掉帧。如果屏幕刷新率比帧速率还快，屏幕会在两帧中显示同一个画面，这种断断续续情况持续发生时，用户将会很明显地感觉到动画的卡顿或者掉帧，然后又恢复正常，我们常称之为闪屏、跳帧、延迟。

应用应避免这些帧率下降的情况，以确保GPU能在屏幕刷新之前完成数据的获取及写入，保证动画流畅。

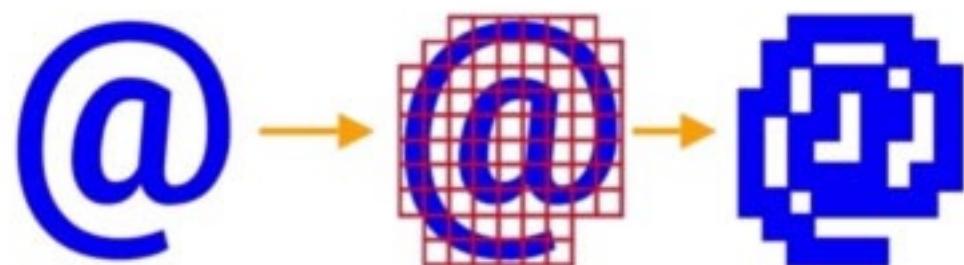
## 1.7 UI绘制机制与栅格化

绝大多数渲染操作都依赖两个硬件：CPU、GPU。CPU负责Measure、layout、Record、Execute的计算操作，GPU负责栅格化（Rasterization）操作。非必需的视图组件会带来多余的CPU计算操作，还会占用多余的GPU资源。

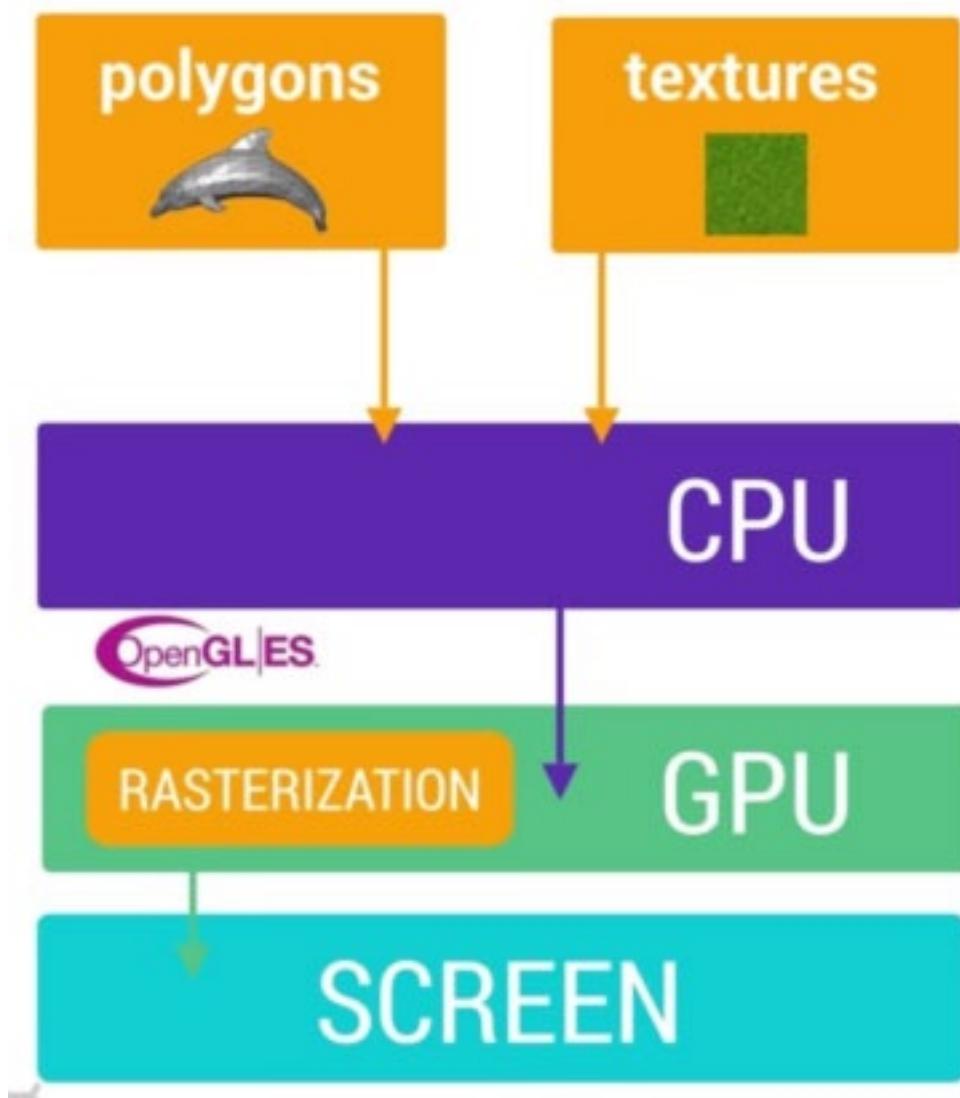


栅格化（Rasterization）能将Button、Shape、Path、Bitmap等资源组件拆分到不同的像素上进行显示。这个操作很费时，所以引入了GPU来加快栅格化的操作。

# Rasterization



CPU 负责把 UI 组件计算成多边形（ Polygons ），纹理（ Texture ），然后交给 GPU 进行栅格化渲染，再将处理结果传到屏幕上显示。



在 Android 里的那些资源组件的显示（比如 Bitmaps、Drawable），都是一起打包到统一的纹理（Texture）当中，然后再传递到 GPU 里面。

图片的显示，则是先经过 CPU 的计算加载到内存中，再传给 GPU 进行渲染。

文字的显示，则是先经过 CPU 换算成纹理（Texture），再传给 GPU 进行渲染，返回到 CPU 绘制单个字符的时候，再重新引用经过 GPU 渲染的内容。

动画的显示更加复杂，我们需要在 16 ms 内处理完所有 CPU 和 GPU 的计算、绘制、渲染等操作，才能获得应用的流畅体验。

## 二. To检测和解决

### 2.1 检测维度

根据业务的不同与所需要的测试粒度的不同，就会有不同的检测维度。目前我所在业务所需的界面性能检测维度如下：

- 界面过度绘制；（检测过度绘制）
- 渲染性能；（检测严格模式下的UI渲染性能呈现）
- 布局边界合理性；（检测元素显示的合理性）

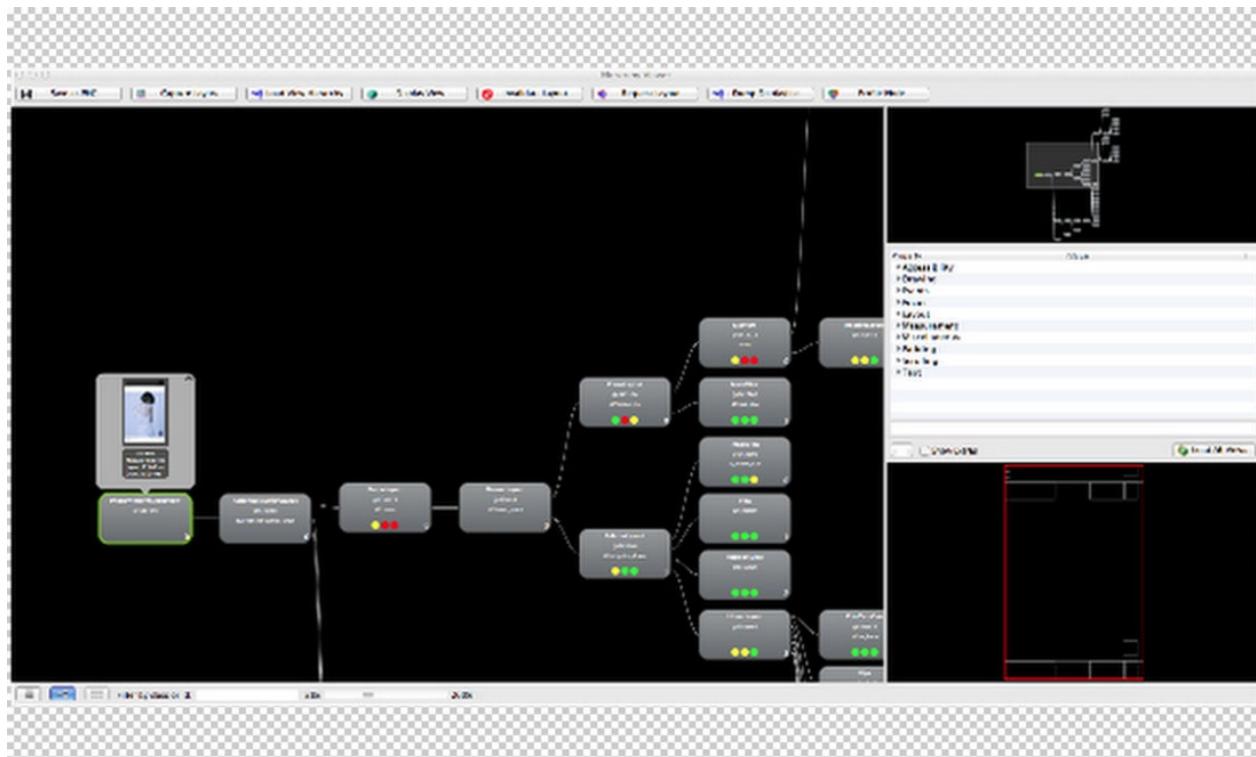
还有专项测试中某些用户场景可能还包含着另外一些隐形的检测维度，比如：

- OpenGL 跟踪分析；
- GPU 视图更新合理性；
- Flash 硬件层更新合理性；
- 动画加 / 减速状态问题点检测；
- .....

## 2.2 调试工具

检测和解决界面性能问题很大程度上依赖于你的应用程序架构，幸运的是，Andorid 提供了很多调试工具，知道并学会使用这些工具很重要，它们可以帮助我们调试和分析界面性能问题，以让应用拥有更好的性能体验。下面列举Android常见的界面性能调试工具：

### 2.2.1 Hierarchy View



Hierarchy View 在Android SDK里自带，常用来查看界面的视图结构是否过于复杂，用于了解哪些视图过度绘制，又该如何进行改进。详见官方使用教程（需要翻墙）：「[戳我](#)」，官方介绍「[戳我](#)」。

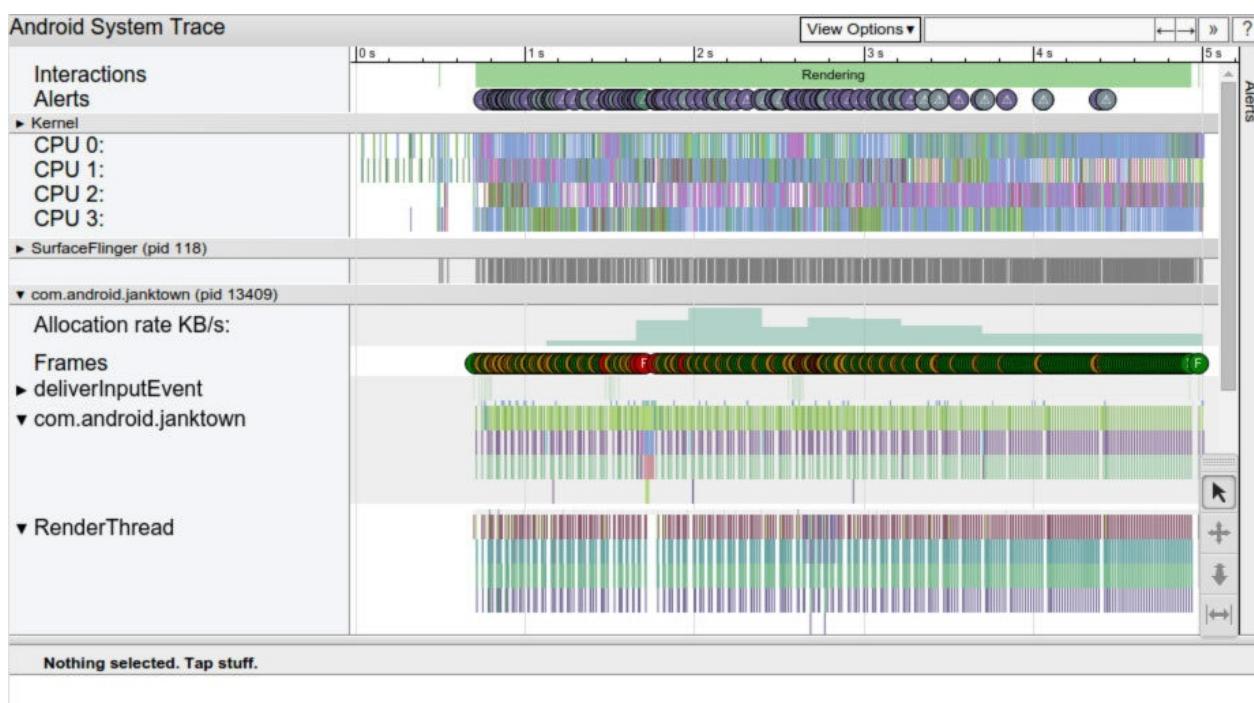
## 2.2.2 Lint

Lint 是 ADT 自带的静态代码扫描工具，可以给 XML 布局文件和 项目代码中不合理的或存在风险的模块提出改善性建议。官方关于 Lint 的实际使用的提示，列举几点如下：

- 包含无用的分支，建议去除；
- 包含无用的父控件，建议去除；
- 警告该布局深度过深；
- 建议使用 compound drawables；
- 建议使用 merge 标签；
- .....

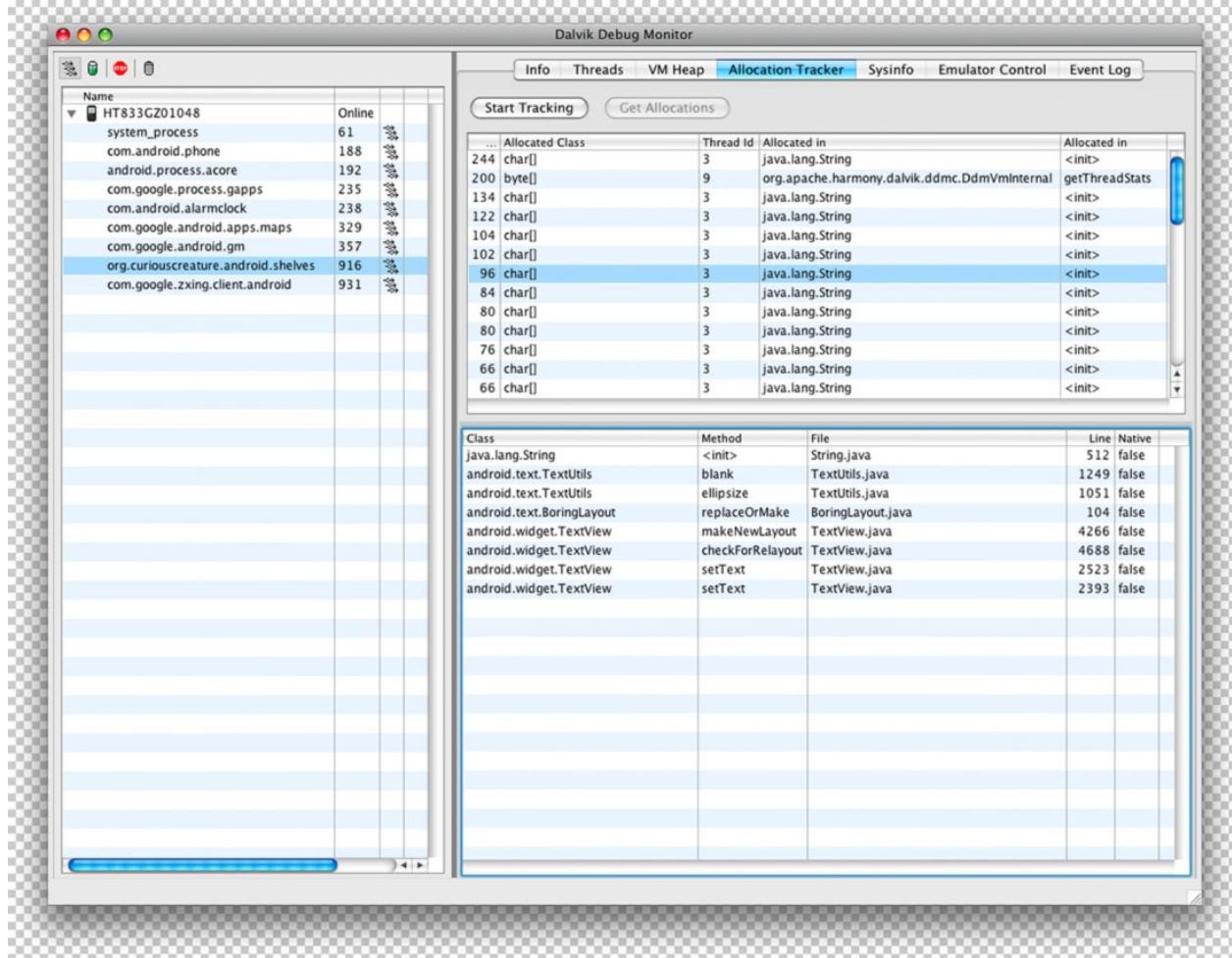
更多 Lint 的官方介绍「[戳我](#)」。

## 2.2.3 Systrace



Systrace 在Android DDMS 里自带，可以用来跟踪 graphics、view 和 window 的信息，发现一些深层次的问题。很麻烦，限制大，实际调试中我基本用不到。官方介绍「[戳我](#)」和「[我](#)」。

## 2.2.4 Track



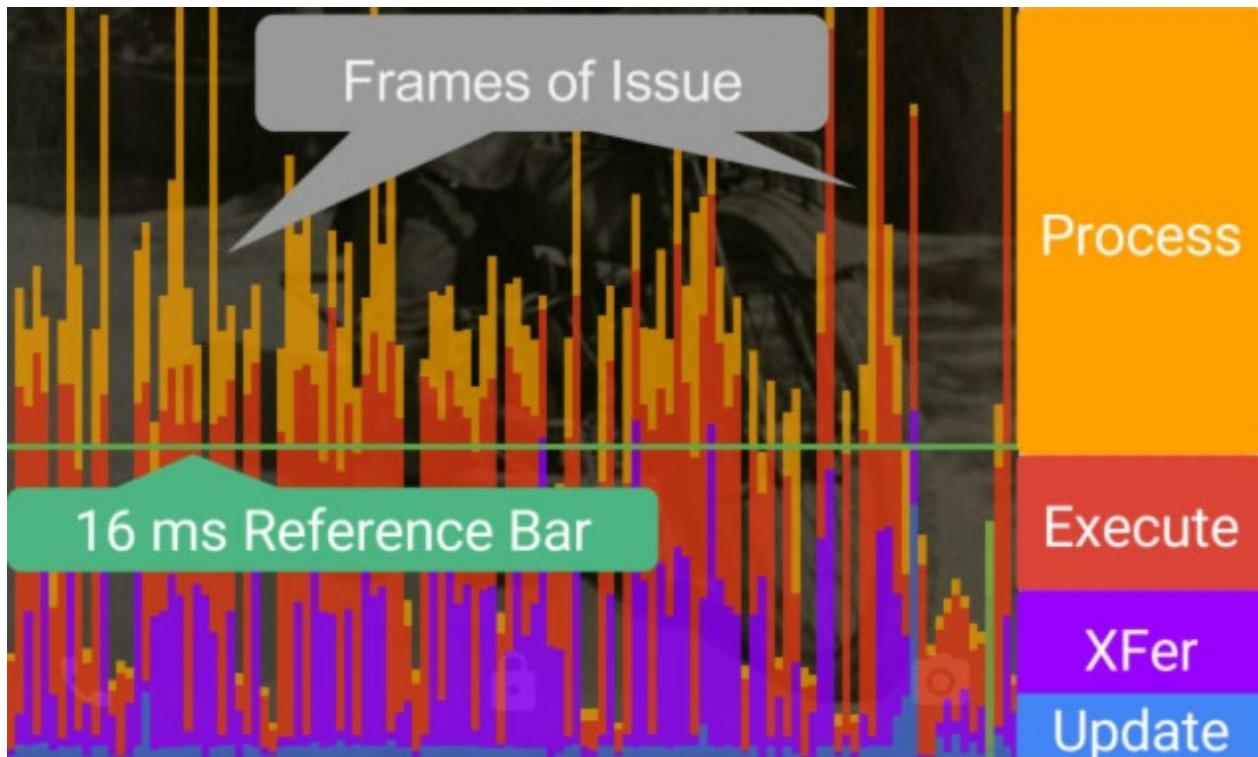
Track 在 Android DDMS里自带，是个很棒的用来跟踪构造视图的时候哪些方法费时，精确到每一个函数，无论是应用函数还是系统函数，我们可以很容易地看到掉帧的地方以及那一帧所有函数的调用情况，找出问题点进行优化。官方介绍 「[戳我](#)」。

## 2.2.5 OverDraw



通过在 Android 设备的设置 APP 的开发者选项里打开“调试 GPU 过度绘制”，来查看应用所有界面及分支界面下的过度绘制情况，方便进行优化。官方介绍「[戳我](#)」。

## 2.2.6 GPU 呈现模式分析



通过在 Android 设备的设置 APP 的开发者选项里启动“GPU 呈现模式分析”，可以得到最近 128 帧 每一帧渲染的时间，分析性能渲染的性能及性能瓶颈。官方介绍「[戳我](#)」。

## 2.2.7 StrictMode

通过在 Android 设备的设置 APP 的开发者选项里启动“严格模式”，来查看应用哪些操作在主线程上执行时间过长。当一些操作违背了严格模式时屏幕的四周边界会闪烁红色，同时输出 StrictMode 的相关信息到 LOGCAT 日志中。

## 2.2.8 Animator duration scale

通过在 Android 设备的设置 APP 的开发者选项里打开“窗口动画缩放” / “过渡动画缩放” / “动画程序时长缩放”，来加速或减慢动画的时间，以查看加速或减慢状态下的动画是否会有问题。

## 2.2.9 Show hardware layer updates

通过在 Android 设备的设置 APP 的开发者选项里启动“显示硬件层更新”，当 Flash 硬件层在进行更新时会显示为绿色。使用这个工具可以让你查看在动画期间哪些不期望更新的布局有更新，方便你进行优化，以获得应用更好的性能。实例《Optimizing Android Hardware Layers》（需要翻墙）：「[戳我](#)」。

## 2.3 如何解决

前面提到过我司的目前所需的测试维度如下：

- 界面过度绘制；（检测过度绘制）
- 渲染性能；（检测严格模式下的UI渲染性能呈现）
- 布局边界合理性；（检测元素显示的合理性）

故接下来将围绕这三两点，分别从概念、追踪、挖掘根源以及排查的工具来具体讲述如何解决，以及给开发的优化建议。

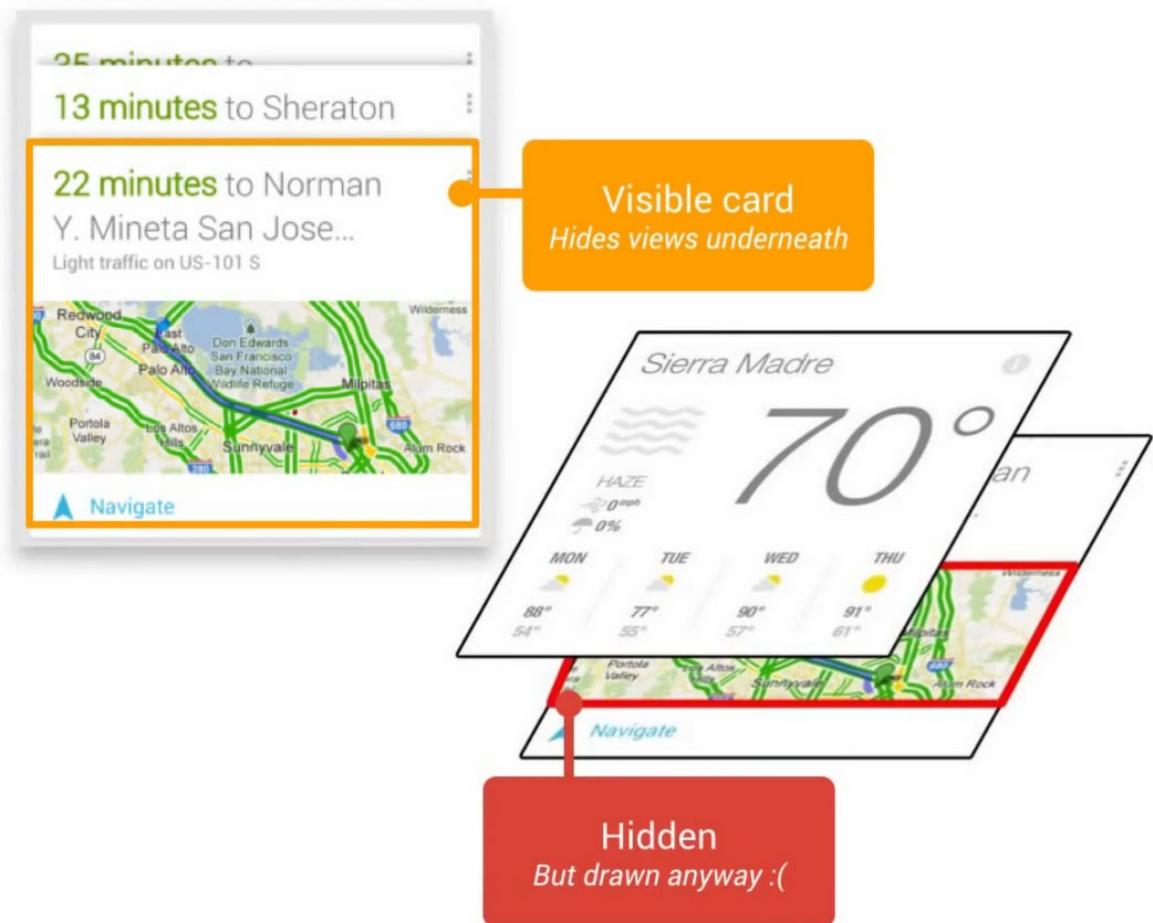
# 三. 界面过度绘制 (OverDraw)

## 3.1 过度绘制概念

过度绘制是一个术语，表示某些组件在屏幕上的一个像素点的绘制次数超过 1 次。

通俗来讲，绘制界面可以类比成一个涂鸦客涂鸦墙壁，涂鸦是一件工作量很大的事情，墙面的每个点在涂鸦过程中可能被涂了各种各样的颜色，但最终呈现的颜色却只可能是 1 种。这意味着我们花大力气涂鸦过程中那些非最终呈现的颜色对路人是不可见的，是一种

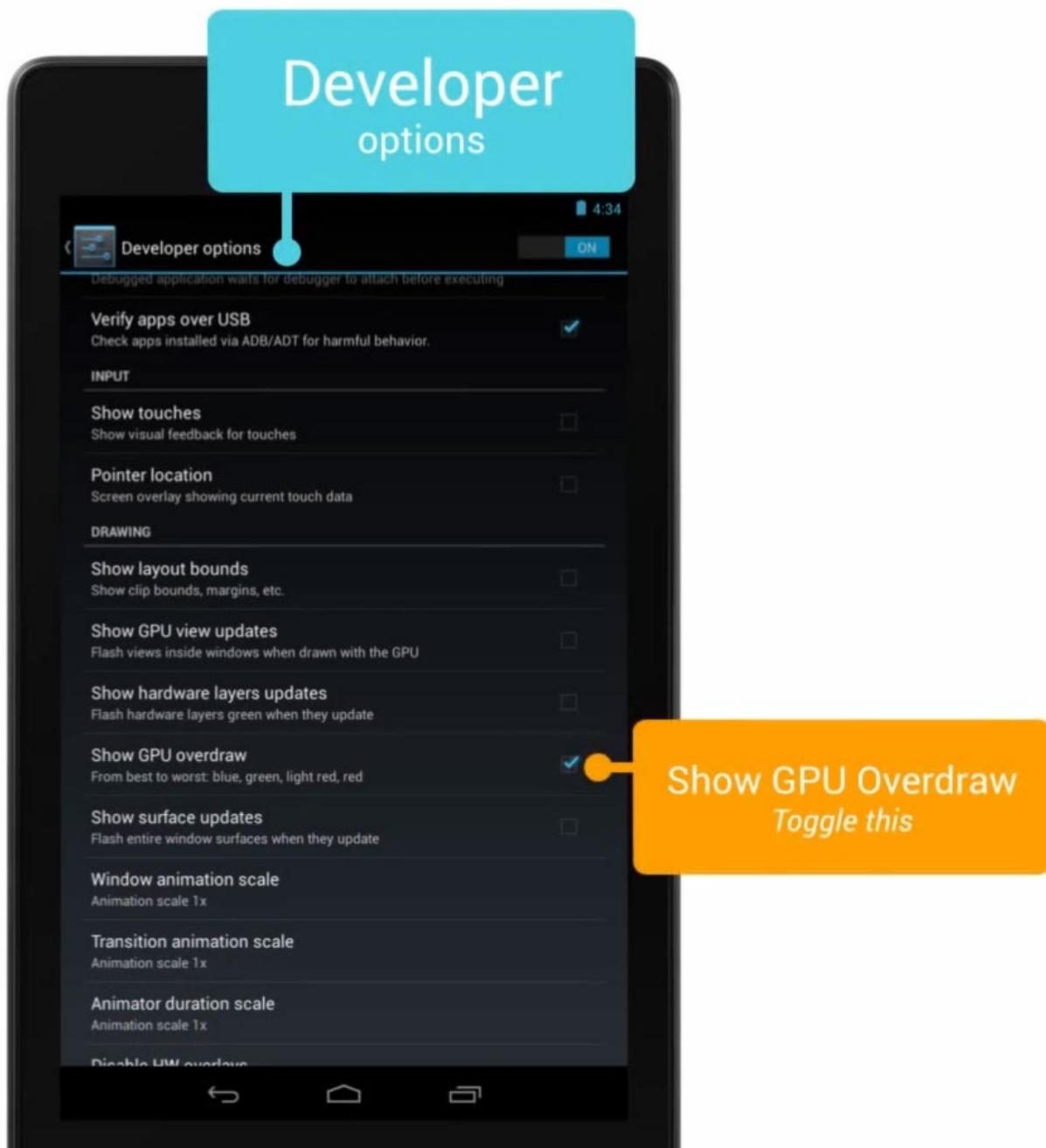
对时间、精力和资源的浪费，存在很大的改善空间。绘制界面同理，花了太多的时间去绘制那些堆叠在下面的、用户看不到的东西，这样是在浪费CPU周期和渲染时间！



官方例子，被用户激活的卡片在最上面，而那些没有激活的卡片在下面，在绘制用户看不到的对象上花费了太多的时间。

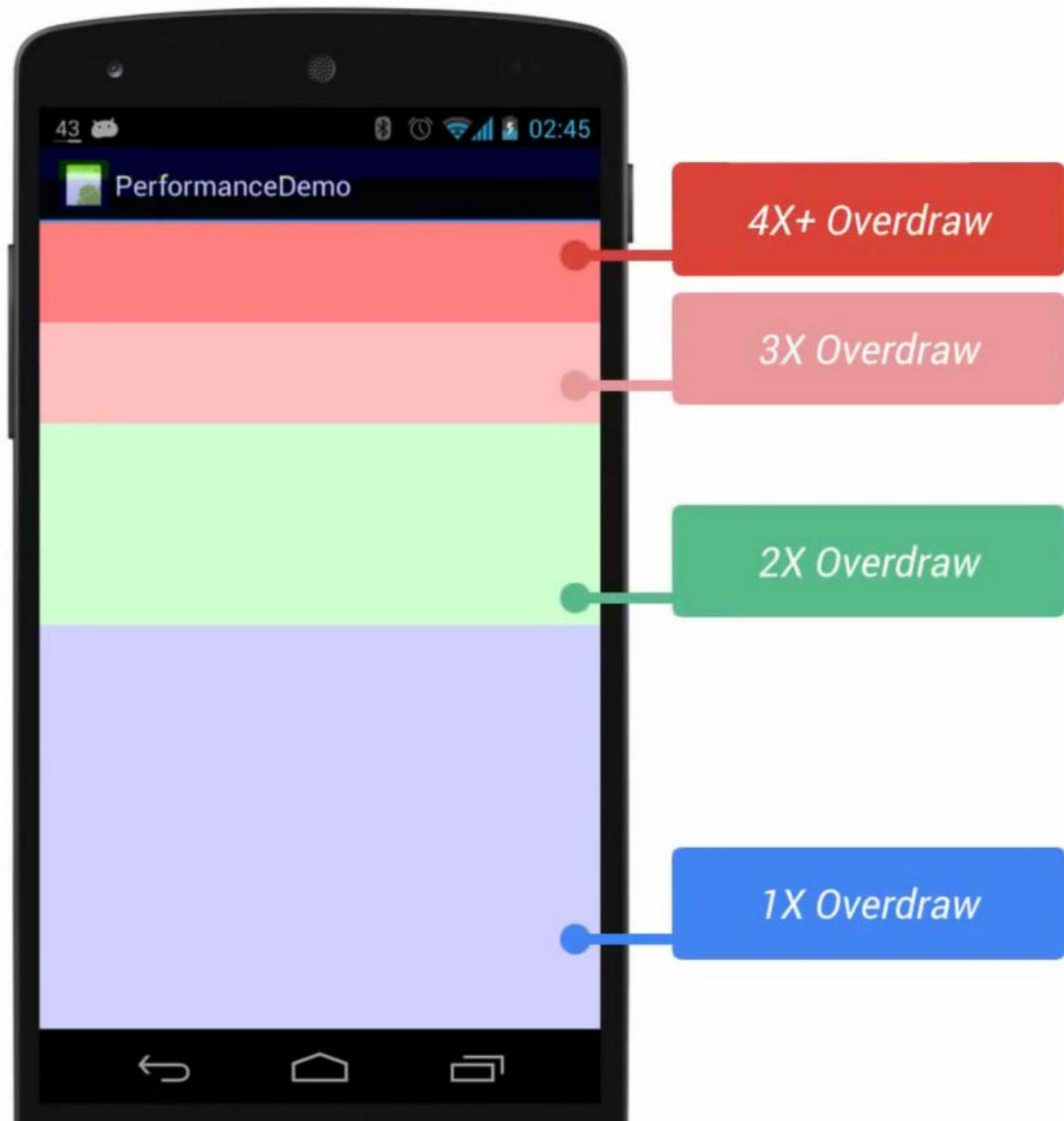
## 3.2 追踪过度绘制

通过在 Android 设备的设置 APP 的开发者选项里打开“调试 GPU 过度绘制”，来查看应用所有界面及分支界面下的过度绘制情况，方便进行优化。

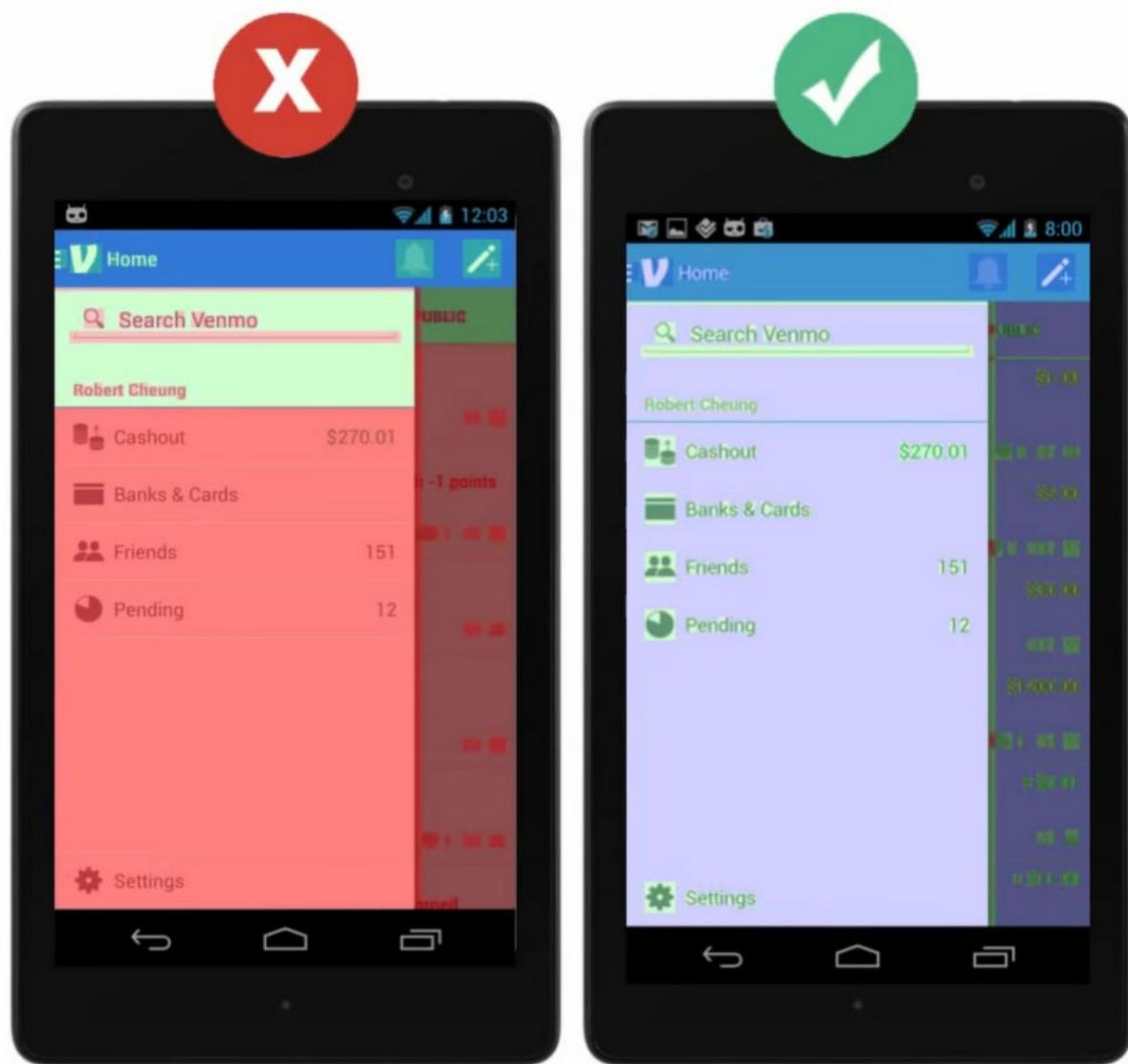


Android 会在屏幕上显示不同深浅的颜色来表示过度绘制：

- 没颜色：没有过度绘制，即一个像素点绘制了 1 次，显示应用本来的颜色；
- 蓝色：1倍过度绘制，即一个像素点绘制了 2 次；
- 绿色：2倍过度绘制，即一个像素点绘制了 3 次；
- 浅红色：3倍过度绘制，即一个像素点绘制了 4 次；
- 深红色：4倍过度绘制及以上，即一个像素点绘制了 5 次及以上；



设备的硬件性能是有限的，当过度绘制导致应用需要消耗更多资源（超过了可用资源）的时候性能就会降低，表现为卡顿、不流畅、ANR 等。为了最大限度地提高应用的性能和体验，就需要尽可能地减少过度绘制，即更多的蓝色色块而不是红色色块。

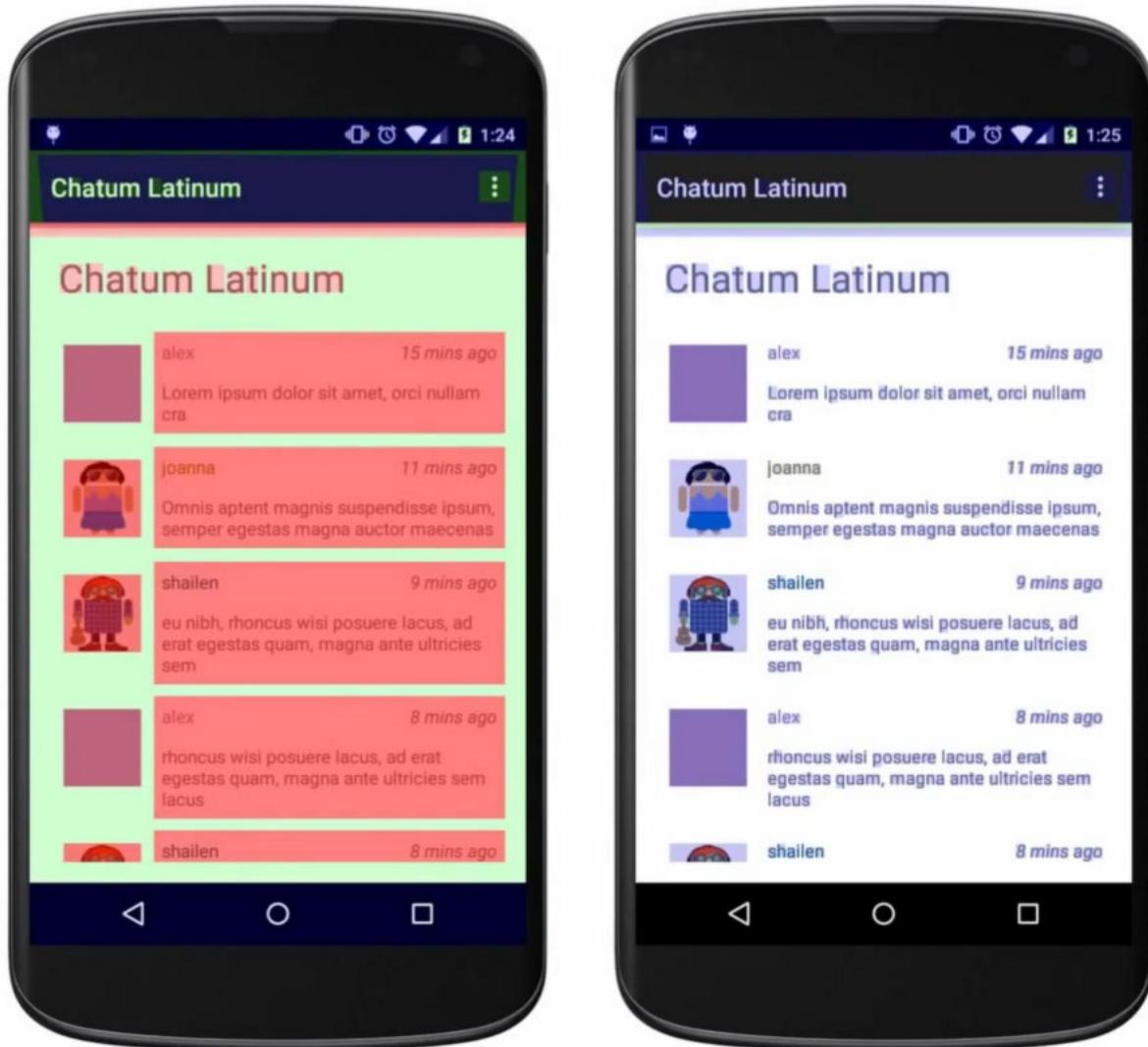


实际测试，常用以下两点来作为过度绘制的测试指标，将过度绘制控制在一个约定好的合理范围内：

- 应用所有界面以及分支界面均不存在超过4X过度绘制（深红色区域）；
- 应用所有界面以及分支界面下，3X过度绘制总面积（浅红色区域）不超过屏幕可视区域的1/4；

### 3.3 过度绘制的根源

过度绘制很大程度上来自于视图相互重叠的问题，其次还有不必要的背景重叠。



官方例子，比如一个应用所有的View都有背景的话，就会看起来像第一张图中那样，而在去除这些不必要的背景之后（指的是Window的默认背景、Layout的背景、文字以及图片的可能存在的背景），效果就像第二张图那样，基本没有过度绘制的情况。

### 3.4 不合理的xml布局对绘制的影响

当布局文件的节点树的深度越深，XML 中的标签和属性设置越多，对界面的显示有灾难性影响。

一个界面要显示出来，第一步会进行解析布局，在 requestLayout 之后还要进行一系列的 measure、layout、draw 操作，若布局文件嵌套过深、拥有的标签属性过于臃肿，每一步的执行时间都会受到影响，而界面的显示是进行完这些操作后才会显示的，所以每一步操作的时间增长，最终显示的时间就会越长。

## 3.5 源码相关

有能力且有兴趣看源码的童鞋，过度绘制的源码位置在：

/frameworks/base/libs/hwui/OpenGLRenderer.cpp，有兴趣的可以去研究查看。

```
if (Properties::debugOverdraw && getTargetFbo() == 0) {
 const Rect* clip = &mTilingClip;
 mRenderState.scissor().setEnabled(true);
 mRenderState.scissor().set(clip->left,
 mState.firstSnapshot()->getViewportHeight() - clip->bottom,
 clip->right - clip->left,
 clip->bottom - clip->top);

 // 1x overdraw
 mRenderState.stencil().enableDebugTest(2);
 drawColor(mCaches.getOverdrawColor(1), SkXfermode::kSrcOver_Mode);

 // 2x overdraw
 mRenderState.stencil().enableDebugTest(3);
 drawColor(mCaches.getOverdrawColor(2), SkXfermode::kSrcOver_Mode);

 // 3x overdraw
 mRenderState.stencil().enableDebugTest(4);
 drawColor(mCaches.getOverdrawColor(3), SkXfermode::kSrcOver_Mode);

 // 4x overdraw and higher
 mRenderState.stencil().enableDebugTest(4, true);
 drawColor(mCaches.getOverdrawColor(4), SkXfermode::kSrcOver_Mode);

 mRenderState.stencil().disable();
}
```

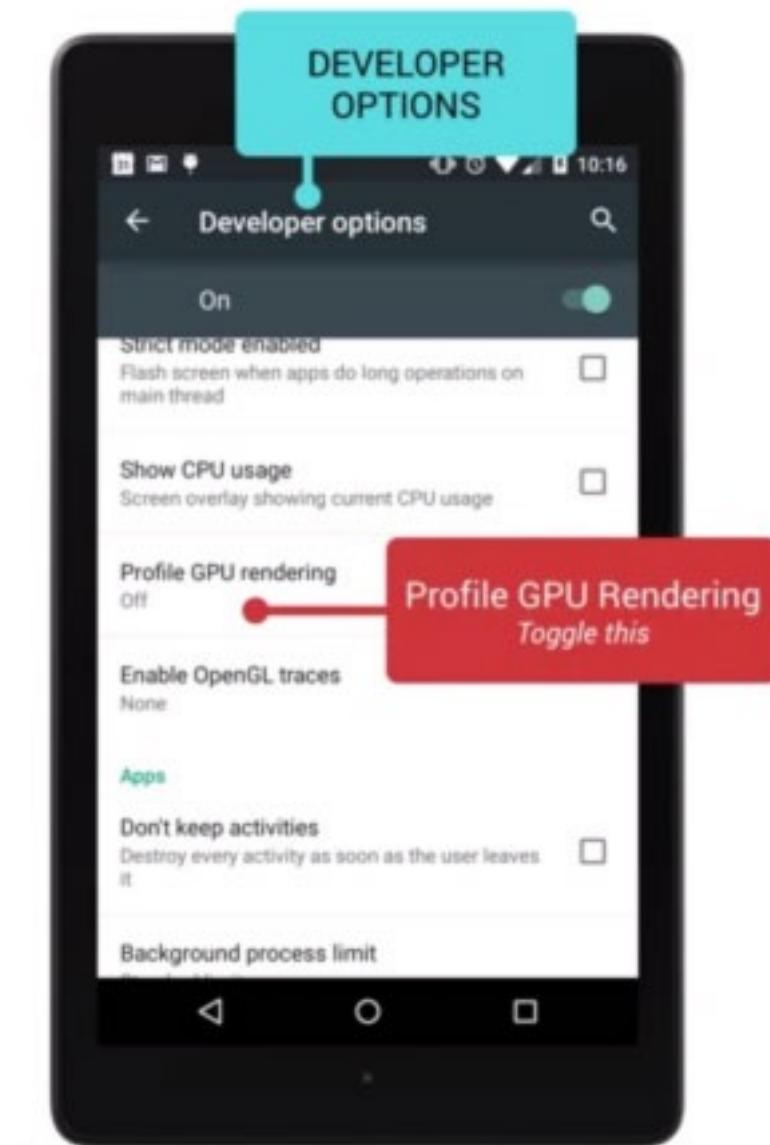
## 四. 渲染性能 (Rendering)

### 4.1 渲染性能概念

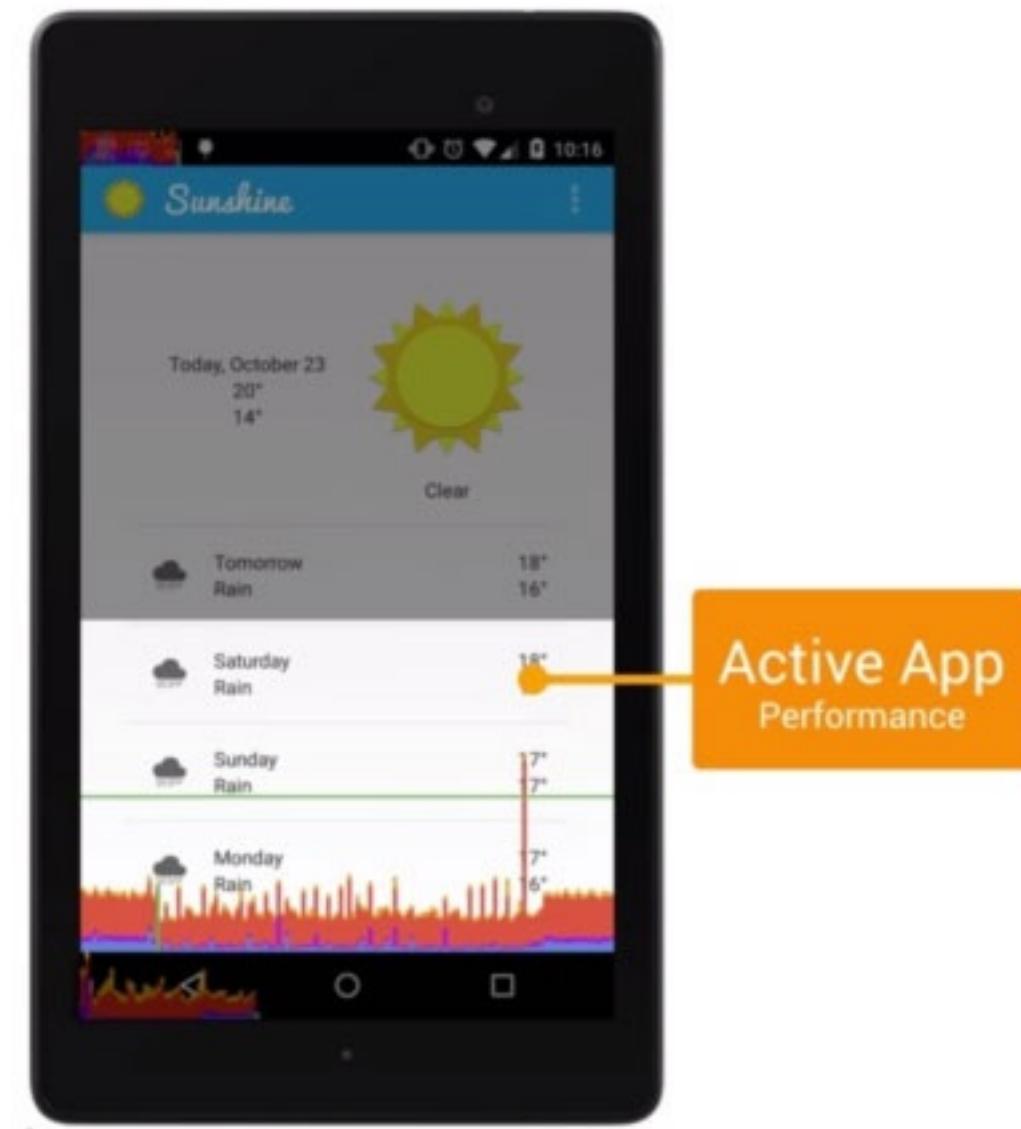
渲染性能往往是掉帧的罪魁祸首，这种问题很常见，让人头疼。好在 Android 给我们提供了一个强大的工具，帮助我们非常容易追踪性能渲染问题，看到究竟是什么导致你的应用出现卡顿、掉帧。

### 4.2 追踪渲染性能

通过在 Android 设备的设置 APP 的开发者选项里打开 “GPU 呈现模式分析” 选项，选择 “在屏幕上显示为条形图”。

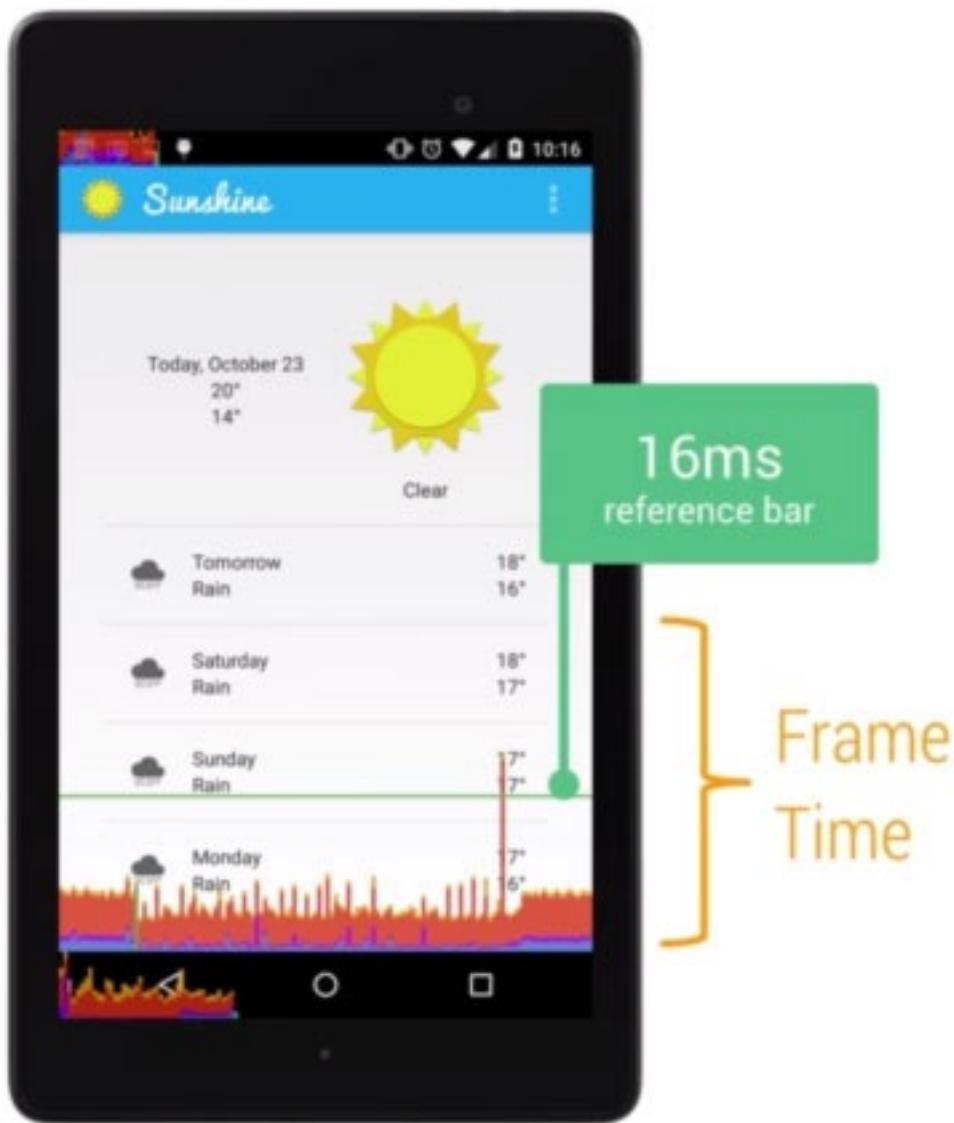


这个工具会在Android设备的屏幕上实时显示当前界面的最近128帧的GPU绘制图形数据，包括StatusBar、NavBar、当前界面的GPU绘制图形柱状图数据。我们一般只需关心当前界面的GPU绘制图形数据即可。



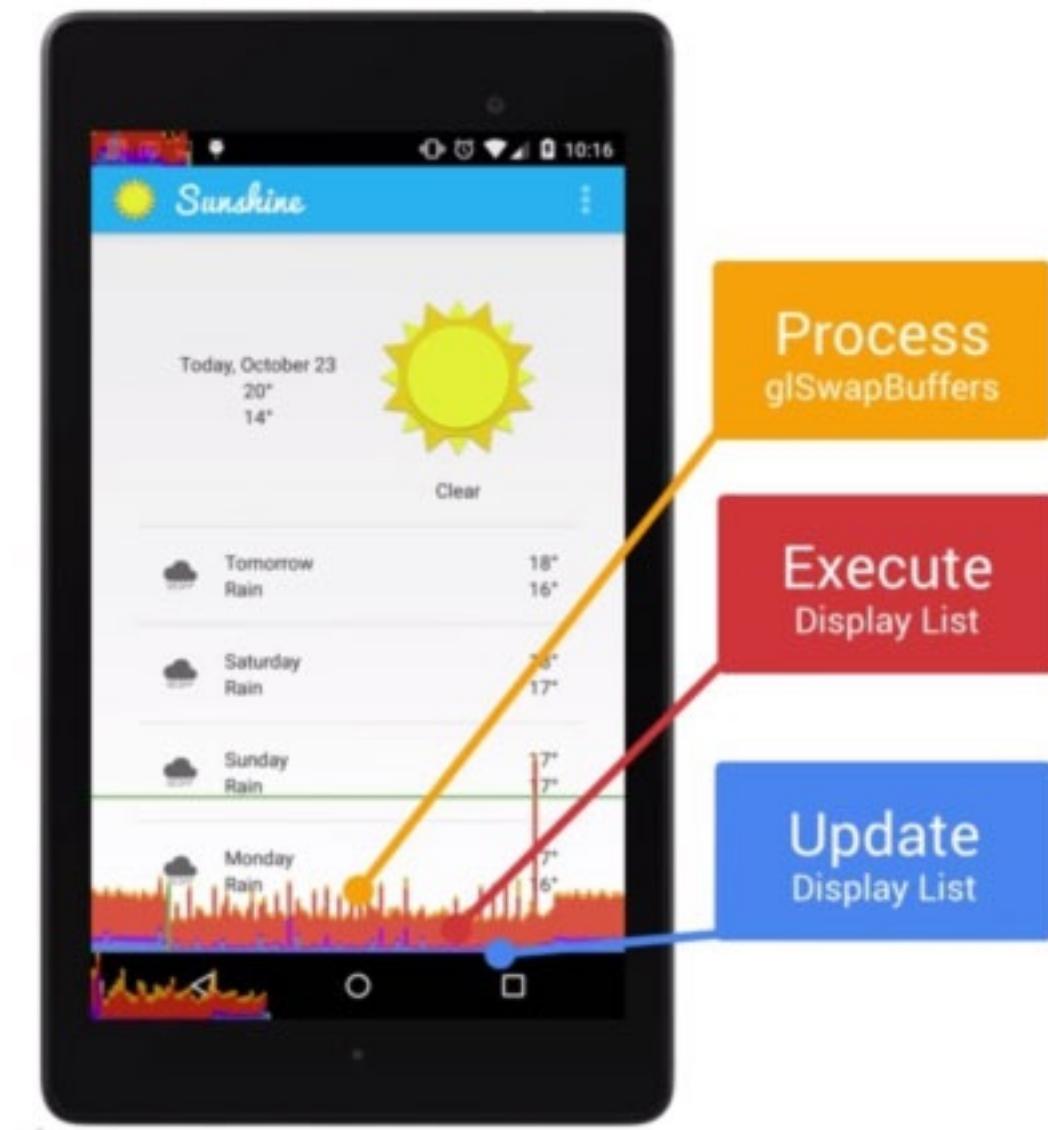
界面上一共有 128 个小柱状图，代表的是当前界面最近的 128 帧 GPU 绘制图形数据。一个小柱状图代表的这一帧画面渲染的耗时，柱状图越高代表耗时越长。随着界面的刷新，柱状图信息也会实时滚动刷新。

中间有一条绿线，代表 16 ms，保持动画流畅的关键就在于让这些垂直的柱状条尽可能地保持在绿线下面，任何时候超过绿线，你就有可能丢失一帧的内容。



每一个柱状图都是由三种颜色构成：蓝、红、黄。

- 蓝色代表的是这一帧绘制 Display List 的时间。通俗来说，就是记录了需要花费多长时间在屏幕上更新视图。用代码语言来说，就是执行视图的 `onDraw` 方法，创建或更新每一个视图的 Display List 的时间。
- 红色代表的是这一帧 OpenGL 渲染 Display List 所需要的时间。通俗来说，就是记录了执行视图绘制的耗时。用代码语言来说，就是 Android 用 OpenGL ES 的 API 接口进行 2D 渲染 Display List 的时间。
- 黄色代表的是这一帧 CPU 等待 GPU 处理的时间。通俗来说，就是 CPU 等待 GPU 发出接到命令的回复的等待时间。用代码语言来说，就是这是一个阻塞调用。



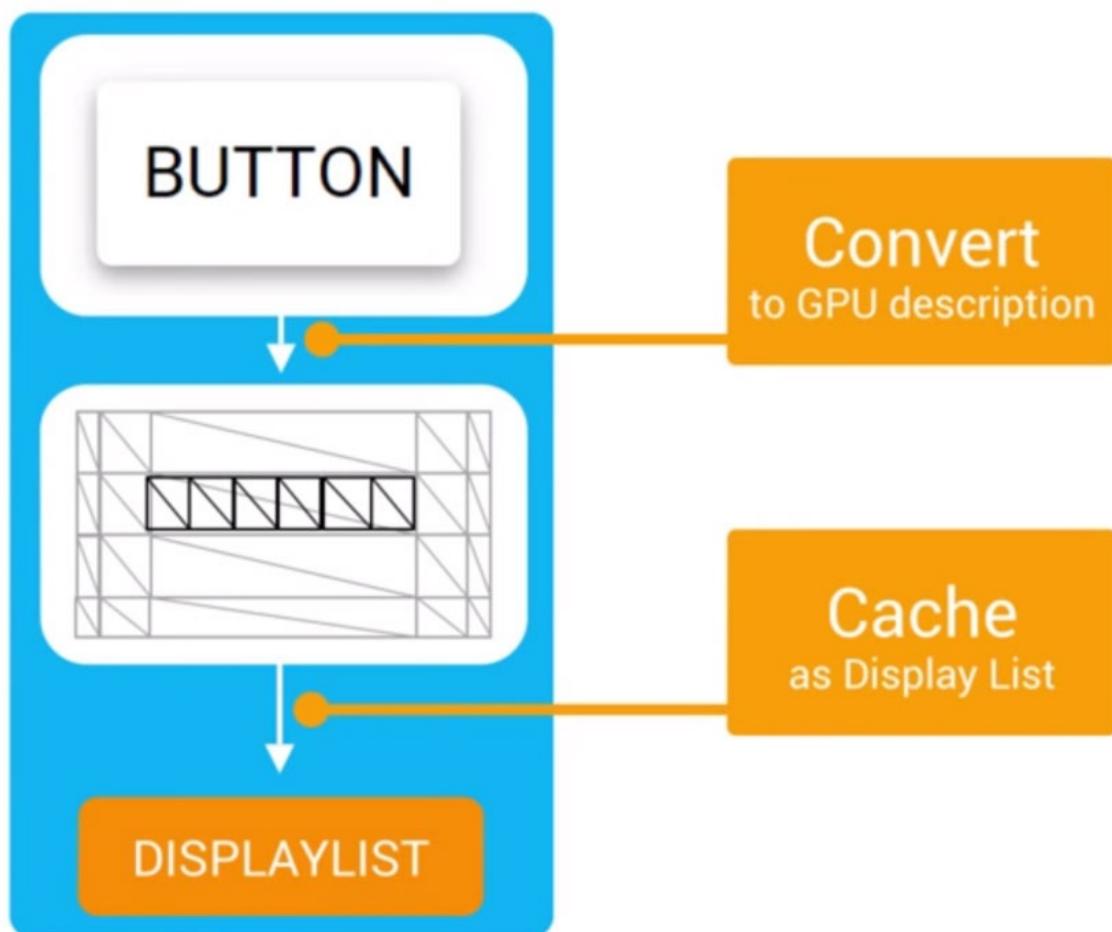
实际测试，常用以下两点来作为渲染性能的测试指标，将渲染性能控制在一个约定好的合理范围内：

- 执行应用的所有功能及分支功能，操作过程中涉及的柱状条区域应至少 90 % 保持到绿线下面；
- 从用户体验的角度主观判断应用在 512 M 内存的 Android 设备下所有操作过程中的卡顿感是否能接受，不会感觉突兀怪异；

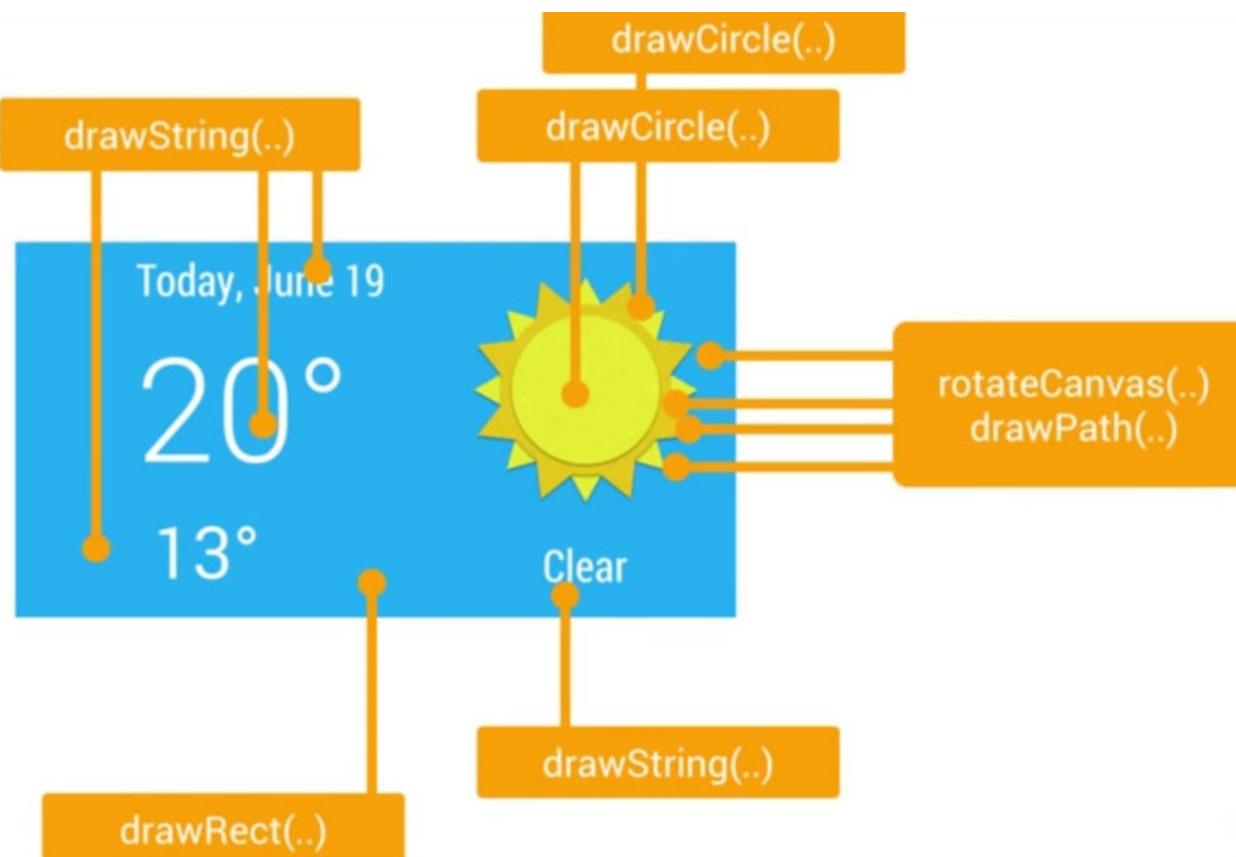
### 4.3 渲染性能差的根源

当你看到蓝色的线较高的时候，可能是由于你的视图突然无效了需要重新绘制，或者是自定义的视图过于复杂耗时过长。

# Draw Phase



当你看到红色的线较高的时候，可能是由于你的视图重新提交了需要重新绘制导致的（比如屏幕从竖屏旋转成横屏后当前界面重新创建），或者是自定义的视图很复杂，绘制起来很麻烦，导致耗时过长。比如下面这种视图：



当你看到黄色的线较高的时候，那就意味着你给 GPU 太多的工作，太多的负责视图需要 OpenGL 命令去绘制和处理，导致 CPU 迟迟没等到 GPU 发出接到命令的回复。

## 4.4 检测说明

这个工具能够很好地帮助你找到渲染相关的问题，帮助你找到卡顿的性能瓶颈，追踪究竟是什么导致被测应用出现卡顿、变慢的情况，以便在代码层面进行优化。甚至让负责产品设计的人去改善他的设计，以获得良好的用户体验。

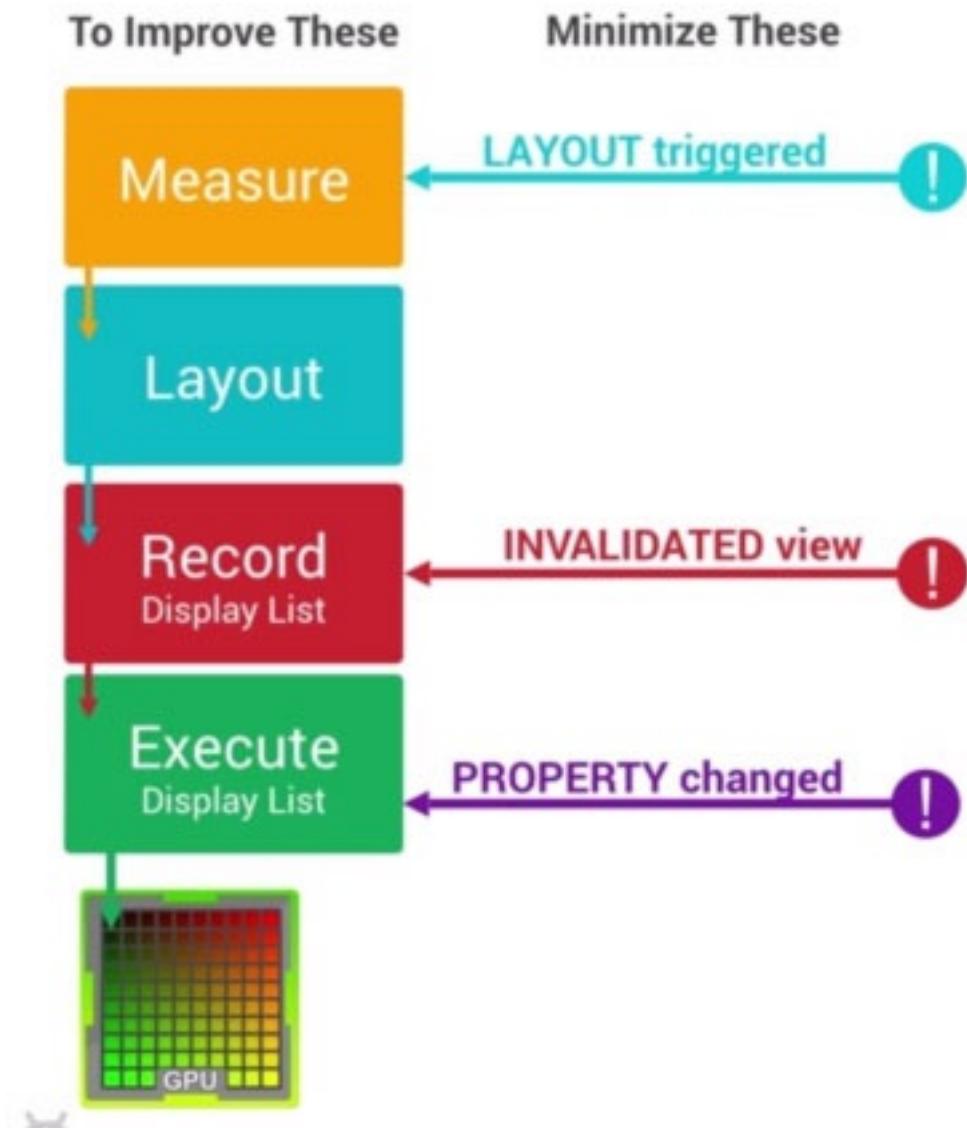
检测渲染性能时，常伴随着开启“严格模式”查看应用哪些情景在 UI 线程（主线程）上执行时间过长。

另外有些强大但可能少用的工具在测试性能渲染时辅助分析，比如：

- `HierarchyViewer`: 这个工具常用来查看界面的视图结构是否过于复杂，用于了解哪些视图过度绘制，又该如何进行改进；
- `Tracer for OpenGL`: 这个工具收集了所有UI界面发给GPU的绘制命令。常用于辅助开发人员 DEBUG 、定位一些 `HierarchyViewer` 工具定位不了的疑难渲染细节问题。

## 4.5 UI绘制机制的补充说明

如上面所说，布局和 UI 组件等都会先经过 CPU 计算成 GPU 能够识别并绘制的多边形（Polygons），纹理（Texture），然后交给 GPU 进行栅格化渲染，再将处理结果传到屏幕上显示。“CPU 计算成 GPU 能够识别并绘制的对象”这个操作是在 DisplayList 的帮助下完成的。DisplayList 拥有要交给 GPU 栅格化渲染到屏幕上的数据信息。



DisplayList 会在某个视图第一次需要渲染时创建。当该视图有类似位置被移动等变化而需要重新渲染这个视图的时候，则只需 GPU 额外执行一次渲染指令并更新到屏幕上就够了。但如果视图中的绘制内容发生变化时（比如不可见了），那之间的 DisplayList 就无法继续使用了，这时系统就会重新执行一次重新创建 DisplayList、渲染DisplayList 并更新到屏幕上。这个流程的表现性能取决于该视图的复杂程度。

## 五. 布局边界合理性

~~这一章节比较简单，考虑到应该没受众，故不展开讨论。~~

# 六. 给开发的界面优化 Advice

## 6.1 优化布局的结构

布局结构太复杂，会减慢渲染的速度，造成性能瓶颈。我们可以通过以下这些惯用、有效的布局原则来优化：

- 避免复杂的View层级。布局越复杂就越臃肿，就越容易出现性能问题，寻找最节省资源的方式去展示嵌套的内容；
- 尽量避免在视图层级的顶层使用相对布局 RelativeLayout。相对布局 RelativeLayout 比较耗资源，因为一个相对布局 RelativeLayout 需要两次度量来确保自己处理了所有的布局关系，而且这个问题会伴随着视图层级中的相对布局 RelativeLayout 的增多，而变得更严重；
- 布局层级一样的情况建议使用线性布局 LinearLayout 代替相对布局 RelativeLayout，因为线性布局 LinearLayout 性能要更高一些；确实需要对分支进行相对布局 RelativeLayout 的时候，可以考虑更优化的网格布局 GridLayout，它已经预处理了分支视图的关系，可以避免两次度量的问题；
- 相对复杂的布局建议采用相对布局 RelativeLayout，相对布局 RelativeLayout 可以简单实现线性布局 LinearLayout 嵌套才能实现的布局；
- 不要使用绝对布局 AbsoluteLayout；
- 将可重复使用的组件抽取出来并用 </include> 标签进行重用。如果应用多个地方的 UI 用到某个布局，就将其写成一个布局部件，便于各个 UI 重用。官方详解 「[戳我](#)」
- 使用 merge 标签减少布局的嵌套层次，官方详解 「[戳我](#)」；
- 去掉多余的不可见背景。有多层背景颜色的布局，只留最上层的对用户可见的颜色即可，其他用户不可见的底层颜色可以去掉，减少无效的绘制操作；
- 尽量避免使用 layoutweight 属性。使用包含 layoutweight 属性的线性布局 LinearLayout 每一个子组件都需要被测量两次，会消耗过多的系统资源。在使用 ListView 标签与 GridView 标签的时候，这个问题显的尤其重要，因为子组件会重复被创建。平分布局可以使用相对布局 RelativeLayout 里一个 0dp 的 view 做分割线来搞定，如果不，那就……；
- 合理的界面的布局结构应是宽而浅，而不是窄而深；

## 6.2 优化处理逻辑

按需载入视图。某些不怎么重用的耗资源视图，可以等到需要的时候再加载，提高UI渲染速度；

- 使用 ViewStub 标签来加载一些不常用的布局；

- 动态地 inflation view 性能要比用 ViewStub 标签的 setVisibility 性能要好，当然某些功能的实现采用 ViewStub 标签更合适；
- 尽量避免不必要的耗资源操作，节省宝贵的运算时间；
- 避免在 UI 线程进行繁重的操作。耗资源的操作（比如 IO 操作、网络操作、SQL 操作、列表刷新等）耗资源的操作应用后台进程去实现，不能占用 UI 线程，UI 线程是主线程，主线程是保持程序流畅的关键，应该只操作那些核心的 UI 操作，比如处理视图的属性和绘制；
- 最小化唤醒机制。我们常用广播来接收那些期望响应的消息和事件，但过多的响应超过本身需求的话，会消耗多余的Android 设备性能和资源。所以应该最小化唤醒机制，当应用不关心这些消失和事件时，就关闭广播，并慎重选择那些要响应的 Intent。
- 为低端设备考虑，比如 512M 内存、双核 CPU、低分辨率，确保你的应用可以满足不同水平的设备。
- 优化应用的启动速度。当应用启动一个应用时，界面的尽快反馈显示可以给用户一个良好的体验。为了启动更快，可以延迟加载一些 UI 以及避免在应用 Application 层级初始化代码。

## 6.3 善用 DEBUG 工具

- 多使用Android提供的一些调试工具去追踪应用主要功能的性能情况；
- 多使用Android提供的一些调试工具去追踪应用主要功能的内存分配情况；

# 附录Appendix

本文首发于[androidtest.org](#)：《Android界面性能调优手册》

## 参考文章

- [Developing for Android I Understanding the Mobile Context](#)
- [Developing for Android IX Tools](#)
- [Training for Android developers](#)
- [Android developers](#)
- [Google developers](#)
- [Google I/O 2013](#)

# 使用StrictMode和MAT分析Android内存泄露

来源:[七章](#)

## 涉及工具

- [Memory Analyzer \(MAT\)](#): Eclipse的内存分析工具，可选择独立版本(Stand-alone)或Eclipse插件(Update-site)。
- [Android Studio](#): Android开发的IDE，自带内存监视工具。
- [leakcanary](#): Square的内存泄露检测库。

## 文章由来

上周开发组内的同事介绍了Android自带的StrictMode，在使用过程发现有多处Activity泄露告警。通过结合Android Studio自带的内存监视工具和MAT，追踪到多处问题代码，并进行了修正，在此进行记录。

## 基本概念

### 垃圾对象

Android虚拟机GC采用的根搜索算法,GC从GC Roots出发，对heap进行遍历。最终没有直接或者间接引用GC Roots的对象就是需要回收的垃圾。

### 内存泄露

某些失去使用价值的对象仍然保持着对跟GC Roots的直接或间接应用，即可以视为发生了内存泄露的情况。Android应用可使用内存较少，发生内存泄露的情况使得内存的使用更加紧张，甚至可能发生OOM。

## 常见原因

- 类的静态变量持有大数据对象
- 静态变量长期维持到大数据对象的引用，阻止垃圾回收。
- 非静态内部类的静态实例

非静态内部类会维持一个到外部类实例的引用，如果非静态内部类的实例是静态的，就会间接长期维持着外部类的引用，阻止被回收掉。

- 资源对象未关闭

资源性对象如Cursor、File、Socket，应该在使用后及时关闭。未在finally中关闭，会导致异常情况下资源对象未被释放的隐患。

- 注册对象未反注册 未反注册会导致观察者列表里维持着对象的引用，阻止垃圾回收。

- Handler临时性内存泄露

- Handler通过发送Message与主线程交互，Message发出之后是存储在MessageQueue中的，有些Message也不是马上就被处理的。在Message中存在一个target，是Handler的一个引用，如果Message在Queue中存在的时间越长，就会导致Handler无法被回收。如果Handler是非静态的，则会导致Activity或者Service不会被回收。
- 由于AsyncTask内部也是Handler机制，同样存在内存泄漏的风险。此种内存泄露，一般是临时性的。

## 检测方式

### 静态检测

- Android Lint

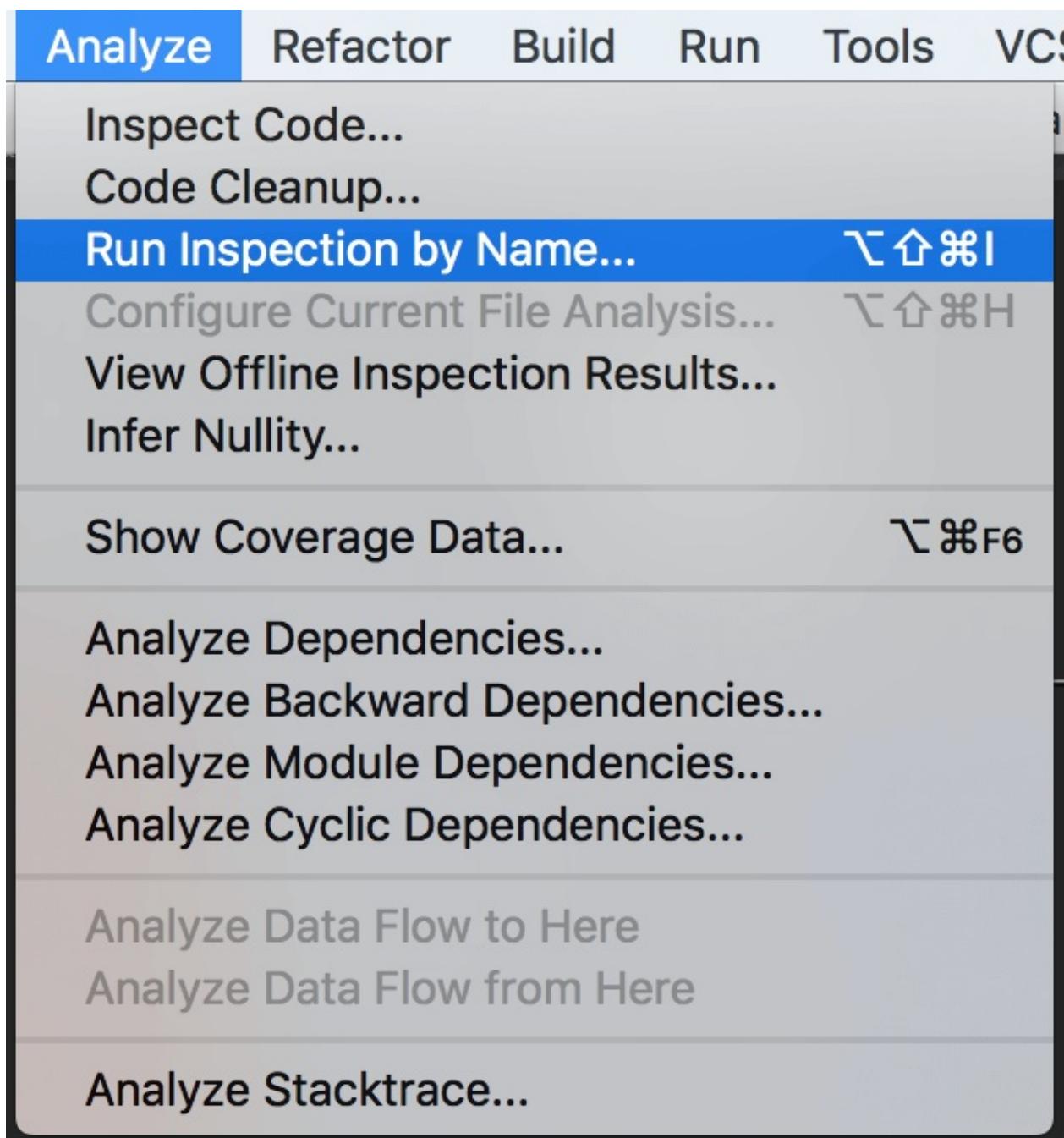
在开发过程中，Android Studio通常会提示我们可能会引起泄露的问题，比如图中所示的Handler。

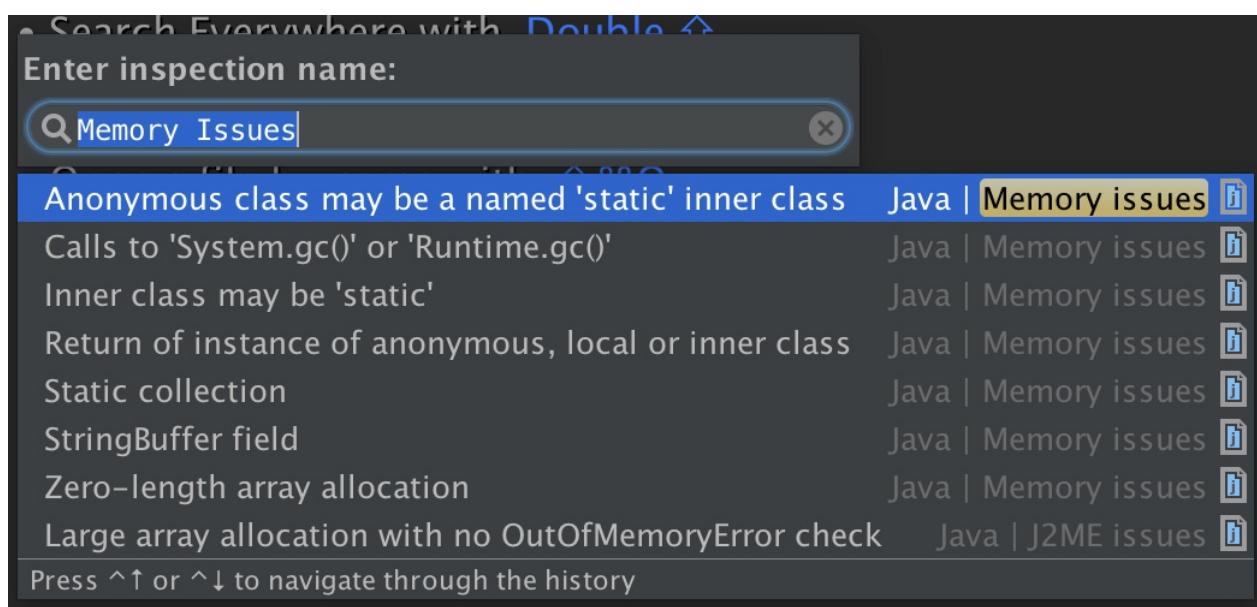
```
private Handler mHandler = new Handler() {
 public void handleMessage(Message msg) {
 switch (msg.what) {
 case 0:
 break;
 case MSG_CLOSE:
 break;
 case MSG_OPENSUBMENU:
 break;
 }
 }
};
```

c: This Handler class should be static or leaks might occur (null) [more...](#) (⌘F1)

- Insteption

也可通过Android Studio中的分析工具对整个项目进行静态分析。Analyze - Run Insteption by Name - 输入Memory Issues，选择需要进行分析的项目进行分析即可。





- StrictMode

StrictMode除了通常使用的检测UI线程中的阻塞性操作外，还能检测内存方面的问题，在发生内存泄露时输出Error的LogCat日志。可以在Application或Activity中进行如下的设置。

**注意请不要在线上版本使用，建议设置开关**

```

@Override
protected void onCreate(Bundle savedInstanceState) {
 if (BuildConfig.DEBUG) {
 StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()

 .detectActivityLeaks() //检测Activity泄露

 .detectLeakedSqlLiteObjects() //检测数据库对象泄露

 .detectLeakedClosableObjects() //检测Closable对象泄露

 .detectLeakedRegistrationObjects() //检测注册对象的泄露，需要API16及以上

 .penaltyLog() //在LogCat中打印

 .build());
 }
 super.onCreate(savedInstanceState);
}

```

- LeakCanary

LeakCanary是Square推出的开源内存泄露检测库，可以检测Fragment、Activity等，并支持上传追踪文件到服务器。最基本的使用方式如下。

1.在 build.gradle 中加入引用，不同的编译使用不同的引用：

```
dependencies {
 debugCompile 'com.squareup.leakcanary:leakcanary-android:1.3'
 releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.3'
}
```

2.在 Application 中：

```
public class ExampleApplication extends Application {

 @Override public void onCreate() {
 super.onCreate();
 LeakCanary.install(this);
 }
}
```

这样，如果检测到某个 activity 有内存泄露，LeakCanary 会自动地显示一个通知。

## 使用MAT进行分析

这里我们模拟一个使用Handler时常见的可能出现内存泄露的情况。

```

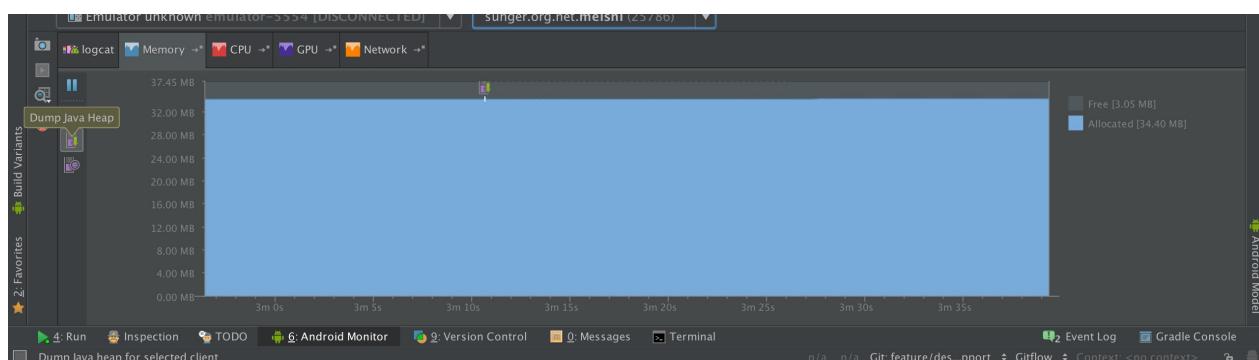
private TextView mCountTV;

private Handler mHandler = new Handler(new Handler.Callback() {
 @Override
 public boolean handleMessage(Message msg) {
 return false;
 }
});

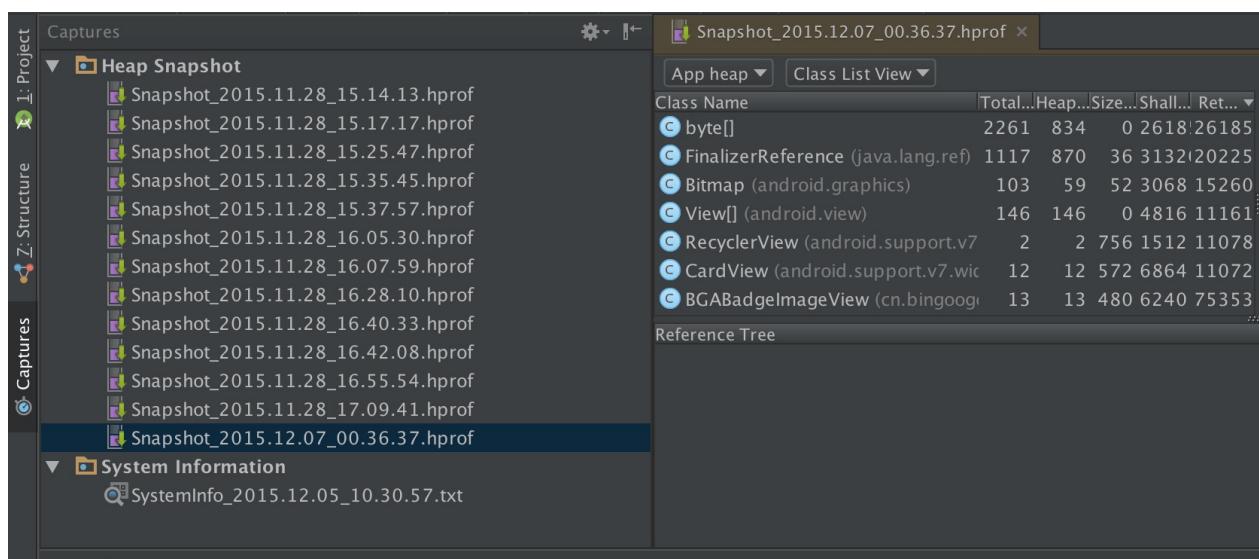
@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 mCountTV = (TextView) findViewById(R.id.tv_main_stop_times);
 mHandler.postDelayed(new Runnable() {
 @Override
 public void run() {
 mCountTV.setText("11111");
 }
 }, 10000);
}

```

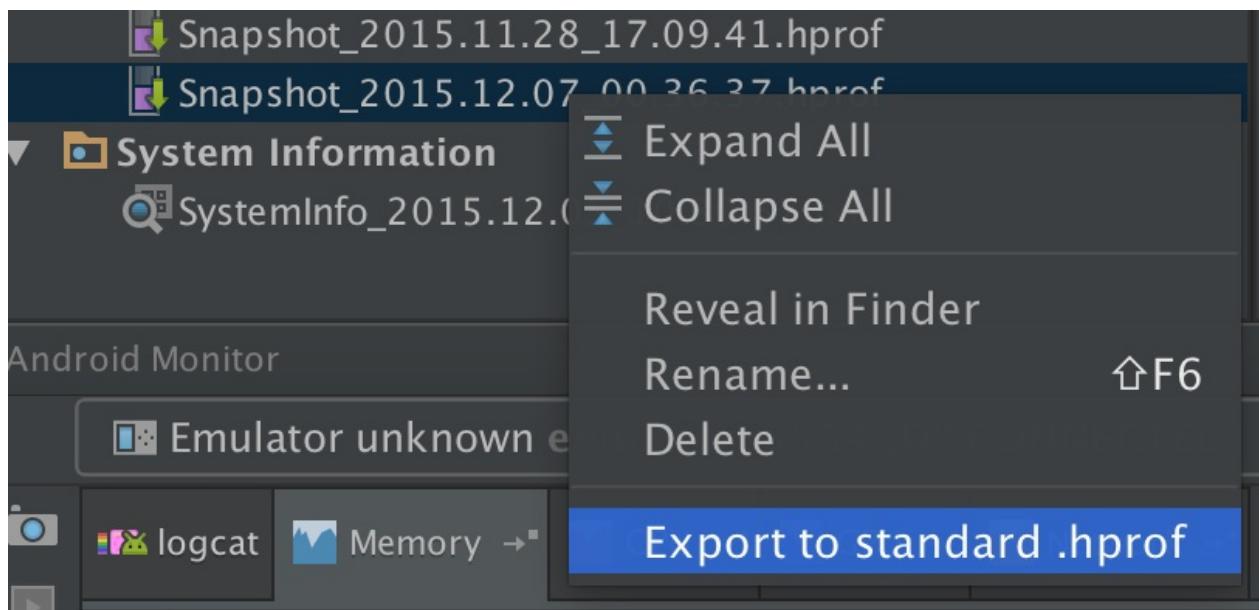
当StrictMode或LeakCanary提示我们发生内存泄露问题时，我们可以通过Android Studio自带的内存监视工具转存Java堆文件。



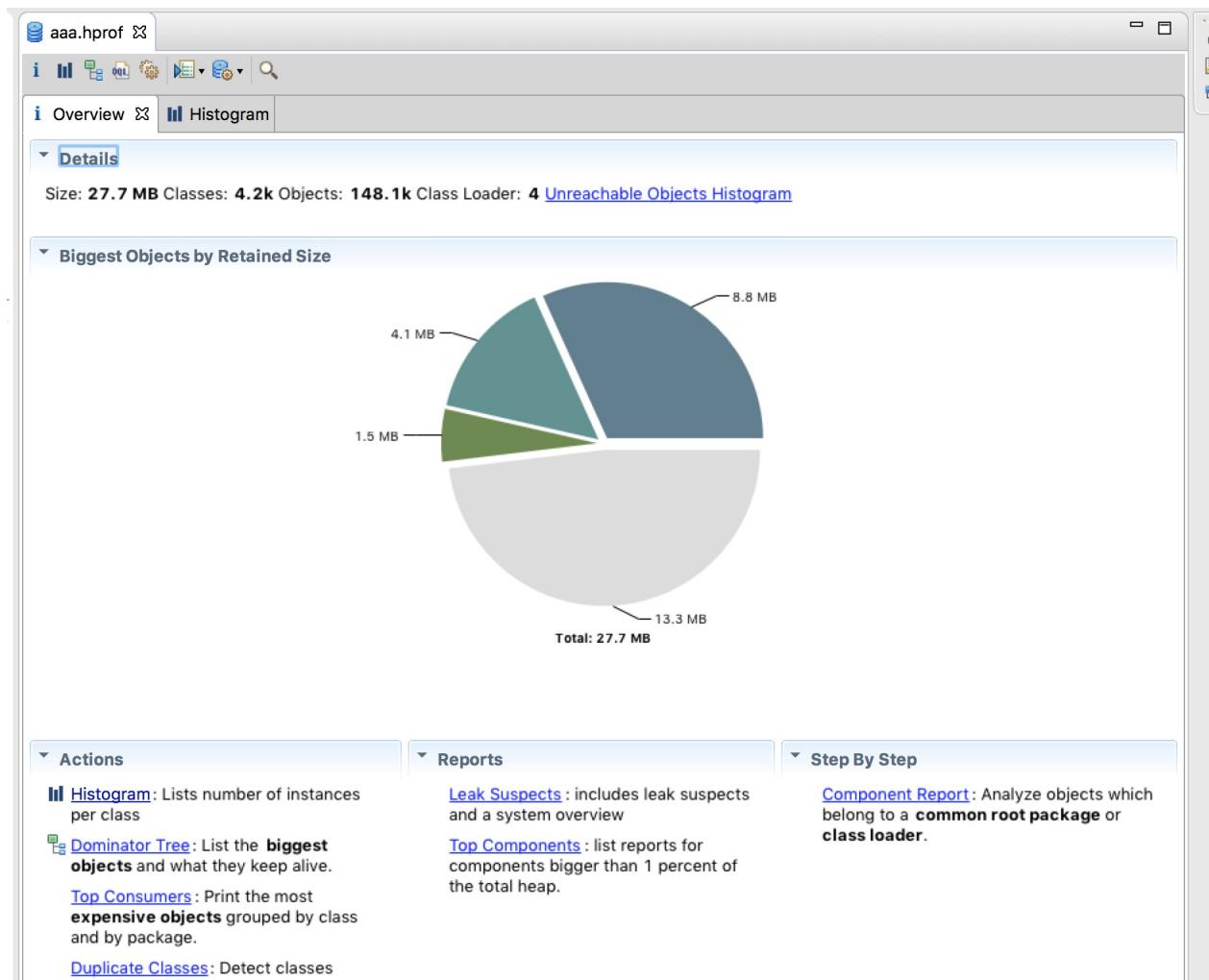
转存成功后可以在Captures中看到堆镜像文件。



由于Android Studio目前提供的内存分析功能有限，这里我们使用MAT工具进行分析。这里我们需要将文件导出为标准的hprof格式才能被MAT成功分析。

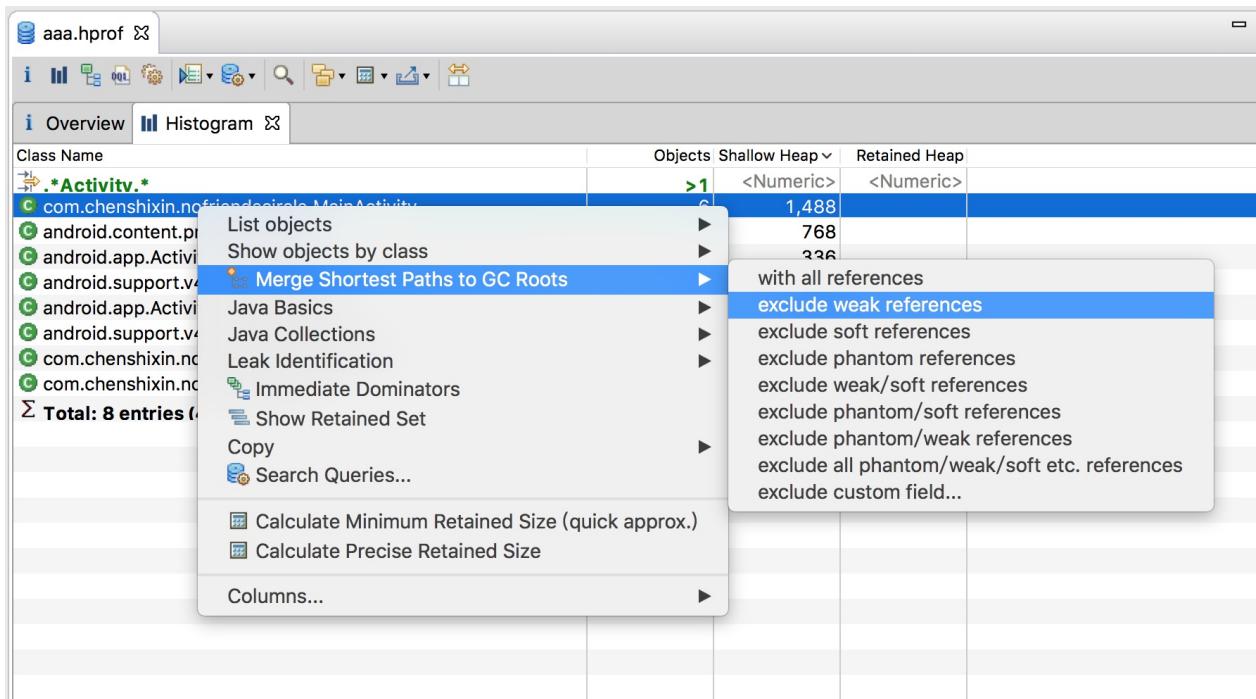


使用MAT打开导出的内存文件，点击下方的Histogram视图(立方图)，搜索警告中提及到的Activity。这里可以搜索ClassName为包含Activity，Objects个数大于1的，可以直观的罗列存在多个实例的Activity。

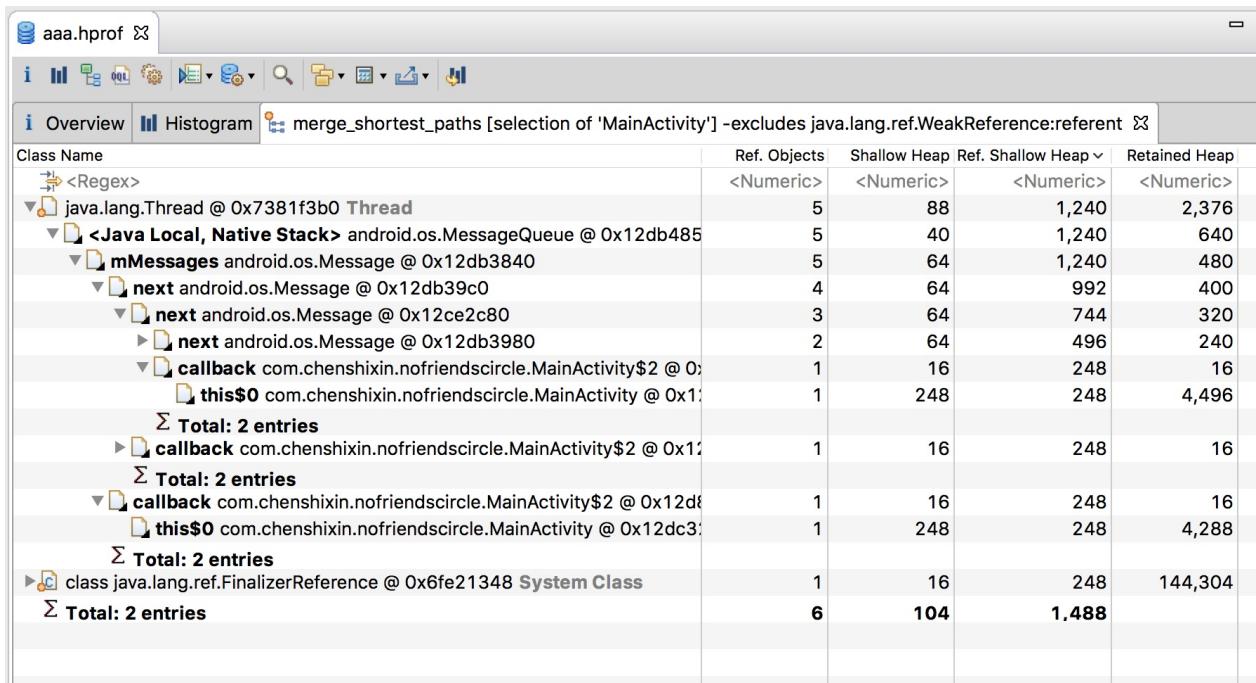


| aaa.hprof                                   |           |              |               |
|---------------------------------------------|-----------|--------------|---------------|
|                                             |           |              |               |
| Class Name                                  | Objects   | Shallow Heap | Retained Heap |
| .*MainActivity.*                            | <Numeric> | <Numeric>    | <Numeric>     |
| com.chenshixin.nofriendscircle.MainActivity | 6         |              | 1,488         |

找到疑似存在问题的Activity，我们需要知道是由于什么原因导致其之前的实例没有销毁，这里可以通过 **右键类名-Merge Shortest Paths to GC Roots-exclude weak references** 找到对GC根的引用。



通过展开可以分析出最终实例未销毁的原因是Activity被callback所引用，它又被Message中一系列的next所引用，最后到主线程才结束。



通过以上方法，可以分析出对象（特别是Activity）未销毁的原因，较为常见的是2中所列原因。为修正此类问题，可以通过避免非静态内部类或者使用弱引用等方式进行改造。

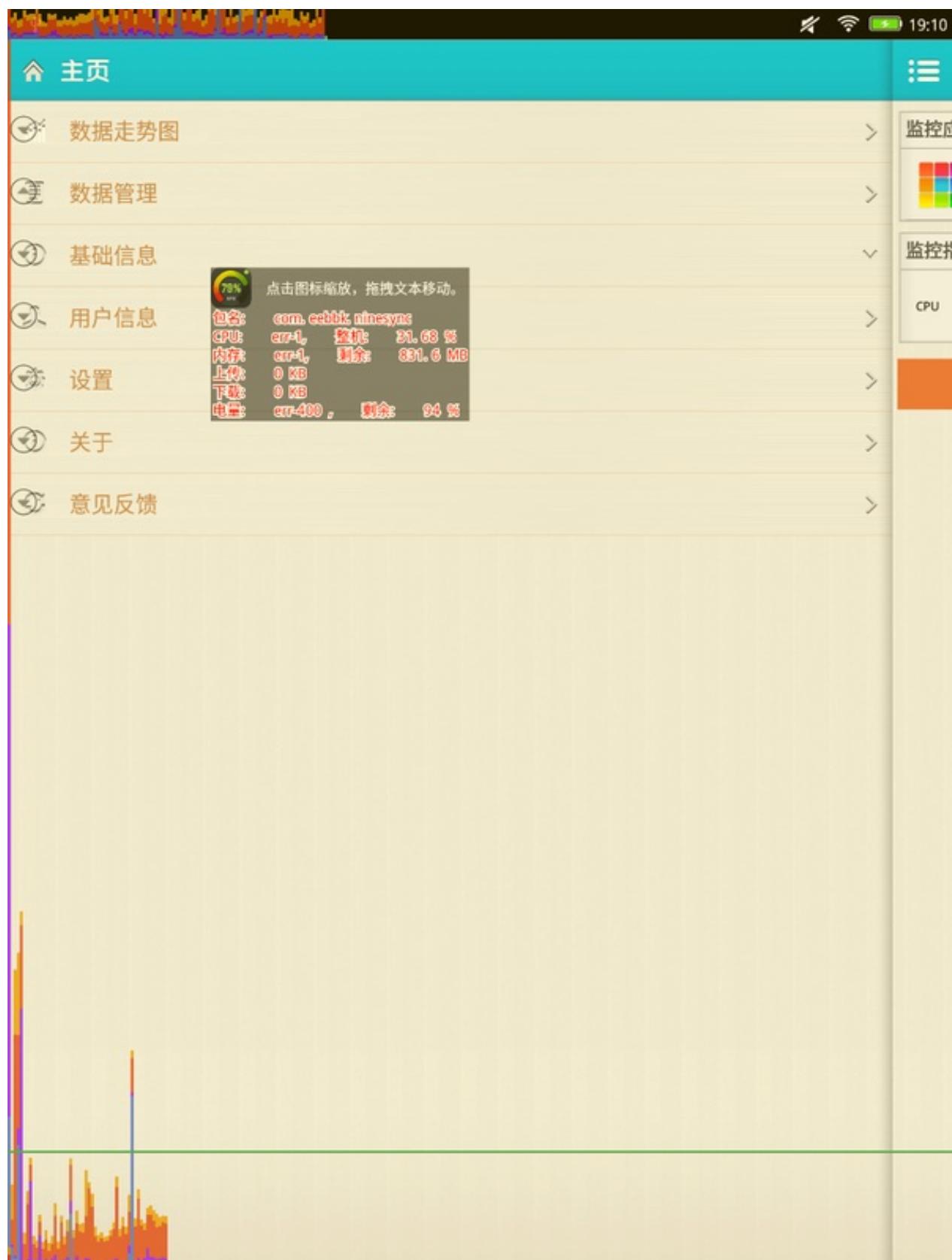
## 参考资料

- 使用Android studio分析内存泄露
- Android内存泄漏研究

- 避免Android中Context引起的内存泄露

# 值得推荐的Android应用性能检测工具列表

来源:[张明云的知乎专栏](#)



著作权归作者所有。

商业转载请联系作者获得授权，非商业转载请注明出处。

作者：张明云

链接：<http://zhuanlan.zhihu.com/zmywly8866/20416881>

来源：知乎

最近这段时间一直在做android应用的性能优化，一个应用性能的好坏并不能依靠我们的主观意识去评判，必须要看数据说话，因此必须要了解和学会使用各种性能测试工具才知道问题出在哪以及具体的优化方向。下面对我对性能优化的理解以及在性能优化过程中使用到的一些工具作个介绍。

## 零 性能指标

Android的应用性能的指标主要有：

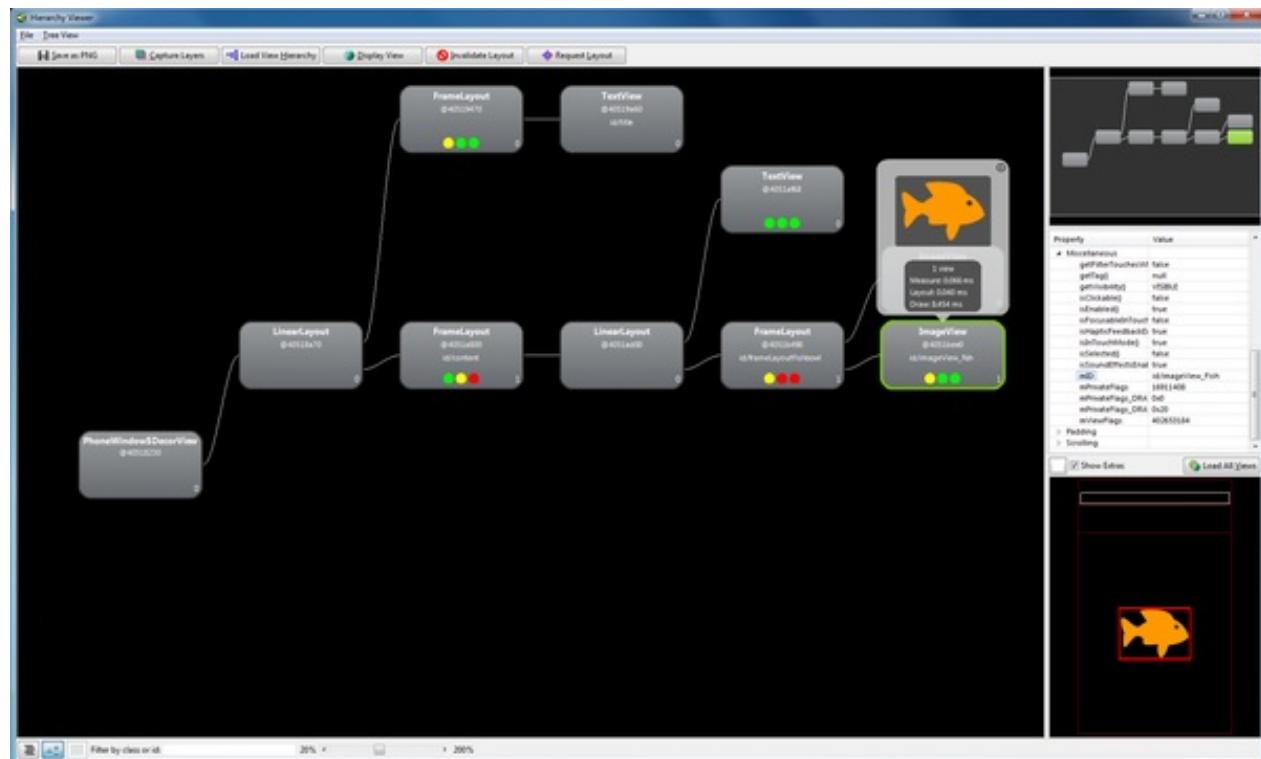
- **布局复杂度**：布局复杂会导致布局需要更长的时间，从而导致进入应用慢、页面切换慢；
- **耗电量**：耗电量大会导致机器发热、缩短机器的有效使用时长；
- **内存**：内存消耗大会导致频繁GC，GC时会暂停其它工作，导致页面卡顿；内存泄露会导致剩余可用内存越来越小；内存不足会导致应用异常；
- **网络**：频繁的网络访问会导致耗电和影响应用的性能；网络交互数据大小会影响网络传输的效率；
- **程序执行效率**：糟糕的代码会严重影响程序的运行效率，UI线程过多的任务会阻塞应用的正常运行，长时间持有某个对象会导致潜在的内存泄露，频繁的IO操作、网络操作而不用缓存会严重影响程序的运行效率。

## — Android官方性能优化工具介绍

android针对上面这些会影响到应用性能的情况提供了一些列的工具：

### 1、布局复杂度：

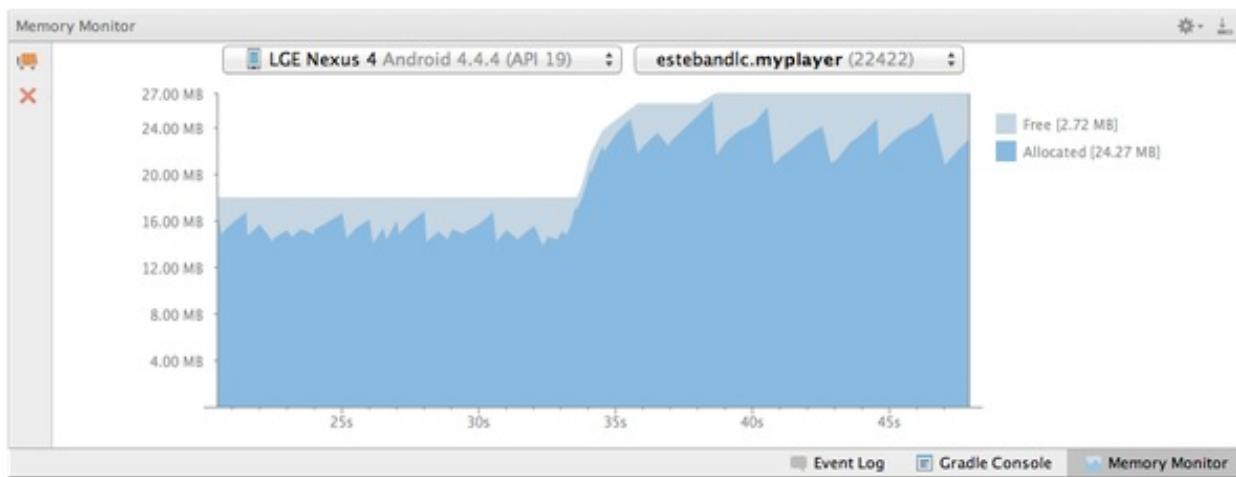
- **hierarchyviewer**：检测布局复杂度，各视图的布局耗时情况：



- **Android开发者模式—GPU过渡绘制：**



- 2、耗电量：Android开发者模式中的电量统计；
- 3、内存：应用运行时内存使用情况查看：Android Studio—Memory/CPU/GPU；



- 内存泄露检测工具：DDMS—MAT；

## 4、网络：Android Studio—NetWork；

## 5、程序执行效率：

- 静态代码检查工具：Android studio—Analyze—Inspect Code.../Code cleanup...，用于检测代码中潜在的问题、存在效率问题的代码段并提供改善方案；
- DDMS—TraceView，用于查找程序运行时具体耗时在哪；
- StrictMode：用于查找程序运行时具体耗时在哪，需要集成到代码中；
- Andorid开发者模式—GPU呈现模式分析。

## 6 程序稳定性：

- monkey，通过monkey对程序在提交测试前做自测，可以检测出明显的导致程序不稳定的问题，执行monkey只需要一行命令，提交测试前跑一次可以避免应用刚提交就被打回的问题。

说明：

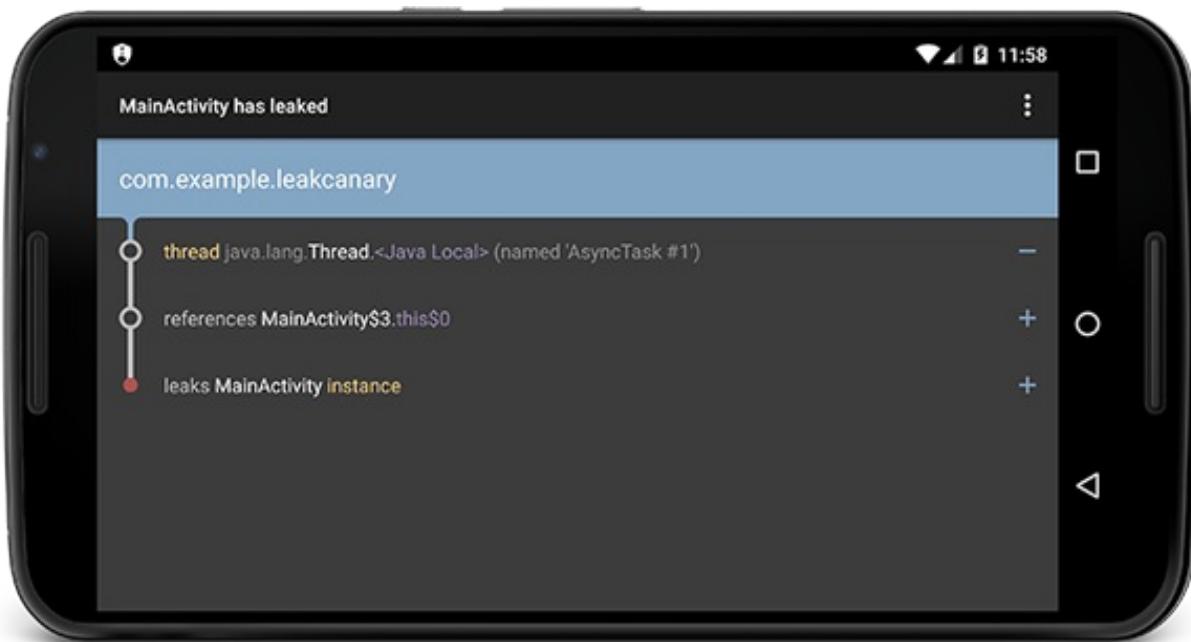
- 上面提到的这些工具可以进[Android开发者官网性能工具介绍](#)查看每个工具的介绍和使用说明；
- Android开发者选项中有很多测试应用性能的工具，对应用性能的检测非常有帮助，具体可以查看：[All about your phone's developer options](#)和[15个必知的Android开发者选项](#)对[Android开发者选项](#)中每一项的介绍；
- 针对Android应用性能的优化，Google官方提供了一系列的性能优化视频教程，对应应用性能优化具有非常好的指导作用，具体可以查看：[优酷Google Developers](#)或者[Android Performance Patterns](#)

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。作者：张明云 链接：<http://zhuanlan.zhihu.com/zmywly8866/20416881> 来源：知乎

## 二 第三方性能优化工具介绍

除了android官方提供的一系列性能检测工具，还有很多优秀的第三方性能检测工具使用起来更方便，比如对内存泄露的检测，使用leakcanary比MAT更人性化，能够快速查到具体是哪存在内存泄露。

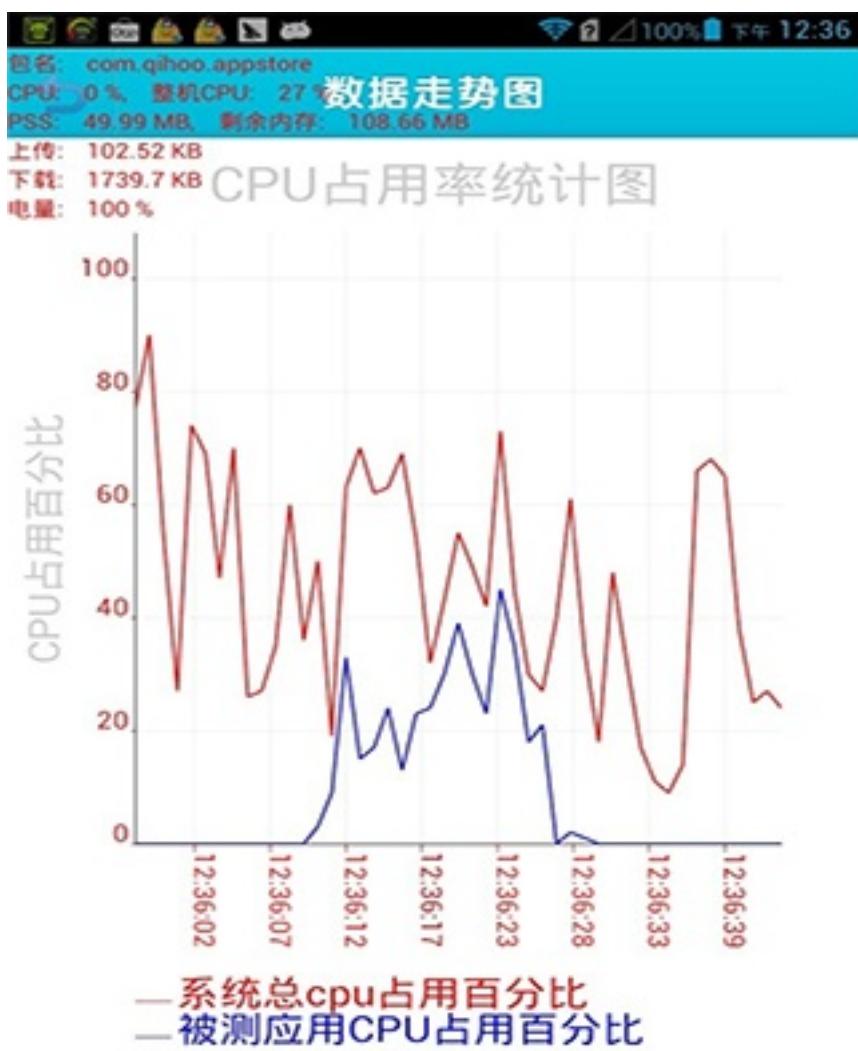
- leakcanary: [square/leakcanary · GitHub](https://github.com/square/leakcanary)，通过集成到程序中的方式，在程序运行时检测应用中存在的内存泄露，并在页面中显示，在应用中集成leakcanary后，程序运行时会存在卡顿的情况，这个是正常的，因为leakcanary就是通过gc操作来检测内存泄露的，gc会知道应用卡顿，说明文档：[LeakCanary 中文使用说明](#)、[LeakCanary：让内存泄露无所遁形](#)。



- GT: [GT Home](#)，GT是腾讯开发的一款APP的随身调测平台，利用GT，可以对CPU、内存、流量、点亮、帧率/流畅度进行测试，还可以查看开发日志、crash日志、抓取网络数据包、APP内部参数调试、真机代码耗时统计等等，需要说明的是，应用需要集成GT的sdk后，GT这个apk才能在应用运行时对各性能进行检测。



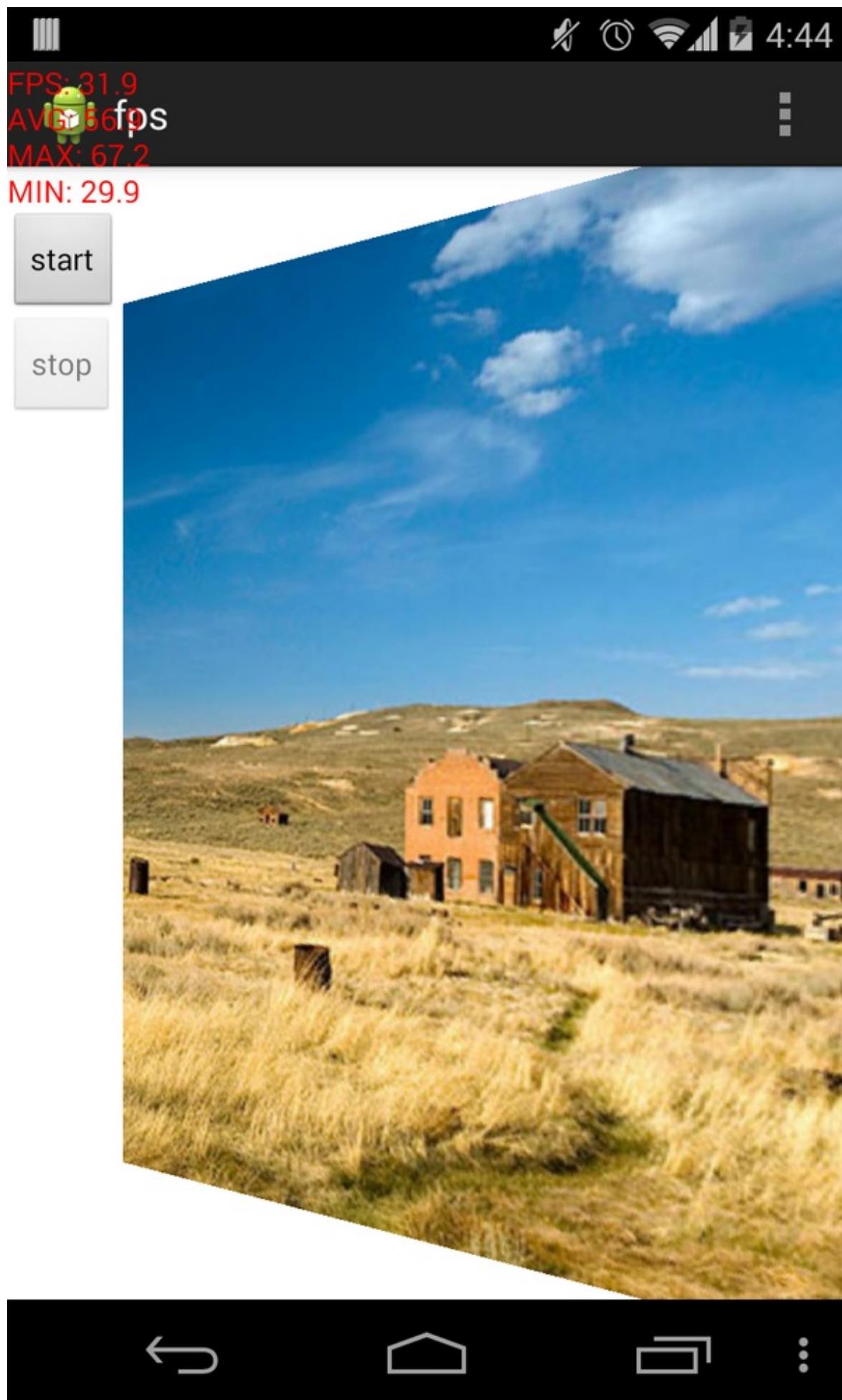
- iTetst: iTetst, 业内首创的Android自动化性能监控工具, 它能够记录特定应用的性能消耗情况, 包括cpu、内存、流量、电量等信息, 支持浮窗实时查看应用的具体信息, iTetst不需要集成sdk到应用中, 在itest中选中需要测试的应用即可进行测试;



- Emmagee: [Emmagee下载](#)、[NetEase/Emmagee · GitHub](#), 网易开发的性能检测工具, Emmagee和iTest一样, 不需要在应用中集成sdk, 能够对应用的常用性能指标进行检测, 并以csv的格式保存方便查看应用的各项参数;

| 指定应用的CPU内存监控情况 |                             |            |            |             |            |        |  |
|----------------|-----------------------------|------------|------------|-------------|------------|--------|--|
| 应用包名:          | com.netease.pris            |            |            |             |            |        |  |
| 应用名称:          | Pris                        |            |            |             |            |        |  |
| 应用PID:         | 675                         |            |            |             |            |        |  |
| 机器内存大小(MB):    | 336.36MB                    |            |            |             |            |        |  |
| 机器CPU型号:       | ARMv7 Processor rev 0 (v7l) |            |            |             |            |        |  |
| 机器android系统版本: | 4.2                         |            |            |             |            |        |  |
| 手机型号:          | sdk                         |            |            |             |            |        |  |
| UID:           | 10048                       |            |            |             |            |        |  |
| 时间             | 应用占用内存PSS(MB)               | 应用占用内存比(%) | 机器剩余内存(MB) | 应用占用CPU率(%) | CPU总使用率(%) | 流量(KB) |  |
| 11:42:28       | 19.81                       | 5.89       | 192.55     | 37.41       | 100        | 3      |  |
| 11:42:33       | 21.25                       | 6.32       | 192.7      | 20.99       | 42.15      | 4      |  |
| 11:42:38       | 23.44                       | 6.97       | 189.06     | 19.08       | 78.01      | 16     |  |
| 11:42:43       | 24.01                       | 7.14       | 188.57     | 59.94       | 99.06      | 16     |  |
| 11:42:48       | 13.71                       | 4.08       | 197.43     | 25.98       | 65.89      | 30     |  |
| 11:42:53       | 16.2                        | 4.82       | 196.69     | 54.21       | 99.66      | 62     |  |
| 11:42:58       | 18.35                       | 5.46       | 194.82     | 72.57       | 100        | 78     |  |
| 11:43:03       | 23.25                       | 6.91       | 192.5      | 65.89       | 100        | 103    |  |

- APT: [Tencent/apt](#) | [CODE](#), 腾讯出的, 暂时还没使用过, 无法评价。
- FPSService: 百度一位开发者写的帧率测试工具, 需要集成到应用中才可查看:



### 三、应用性能优化资料推荐

- 首页 - 专注安卓性能优化以及最佳实践的
- [Blog Archive](#)
- [Android Performance Patterns](#)
- [Best Practices for Performance](#)
- 另一个专注应用性能优化的博客



# 加速你的 Android 应用

来源：[稀土掘金](#)

原文链接：[Speed up your app](#)

原文作者：[UDI COHEN](#)

译者：[zijianwang90](#)

几周之前，我在Droidcon NYC上有过一次关于Android性能优化的演讲。

我在这个演讲中花费了大量的时间，因为我想通过真实的例子展现性能问题，以及我是通过什么样的工具去发掘这些问题的。因为时间原因，在演讲中我不得不舍弃一半的内容。在这篇文章中，我会总结在演讲中我所讨论的所有内容，并且给出实例

[点此链接进入演讲视频](#)

现在，我们来逐一讨论我在演讲中提及的一些重点内容，希望我的阐述足够的清晰。首先，在我进行性能优化的时候我遵循如下原则：

## 原则

每当我遇到性能问题，或者尝试发现性能问题的时候，我会遵循如下原则：

- **坚持性能测试** – 不要用你的眼睛去优化性能。也许在你盯着同一个动画看了几次之后，你会开始相信他运行的越来越流畅了。数据不会说谎。在你优化你的代码之前以及之后，使用我们将要介绍的一系列工具，去多次的测试你的app到底性能几何。
- **使用低端设备** – 如果你想要你想暴露你应用的性能问题，低端设备往往会更加的容易。性能强大的设备往往不会太在意你应用上面的一些优化问题，且不是所有用户都

在使用这些旗舰设备。

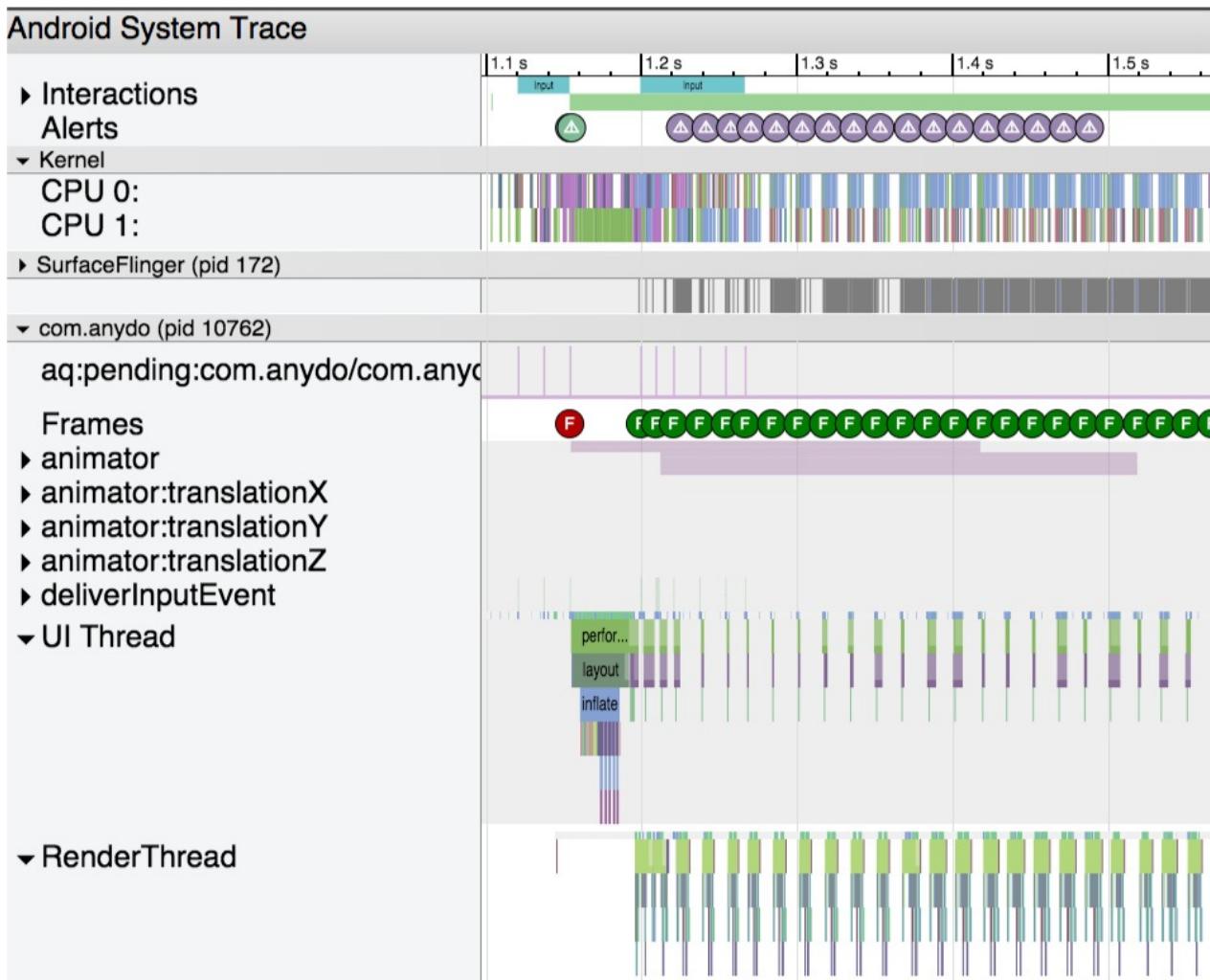
- **权衡** – 性能的优化始终围绕着权衡这两个字。你在某一个点上的优化可能会造成另一点上出现问题。在很多情况下，你会花大量的时间寻找并解决这些问题，但造成这些问题的原因也可能是因为例如bitmaps的质量，或是你没有使用正确的数据结构去存储你的数据。所以你要时刻准备好作出一定的牺牲

## Systrace

Systrace是一个非常好但却有可能被你忽视的工具，这是因为开发者们往往不确定Systrace能够为他们提供什么样的信息。

Systrace会展示一个运行在手机上程序状况的概览。这个工具提醒了我们手机其实是一个可以在同一时间完成很多工作的电脑。在最近的一次SDK更新中，这个工具在数据分析能力上得到了提升，用以帮助我们寻找性能问题之所在。

下面让我们来看看Systrace长什么样子：

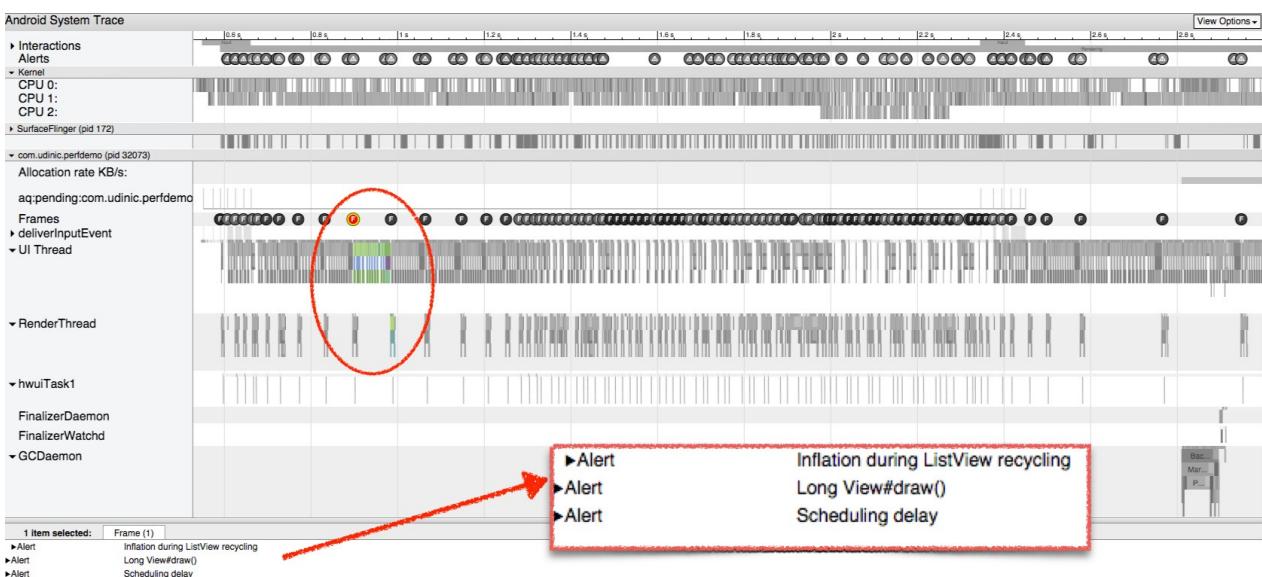


你可以通过 Android Device Monitor Tool 或者是命令行来生成 Systrace 文件，想了解更多[猛戳此处](#)。

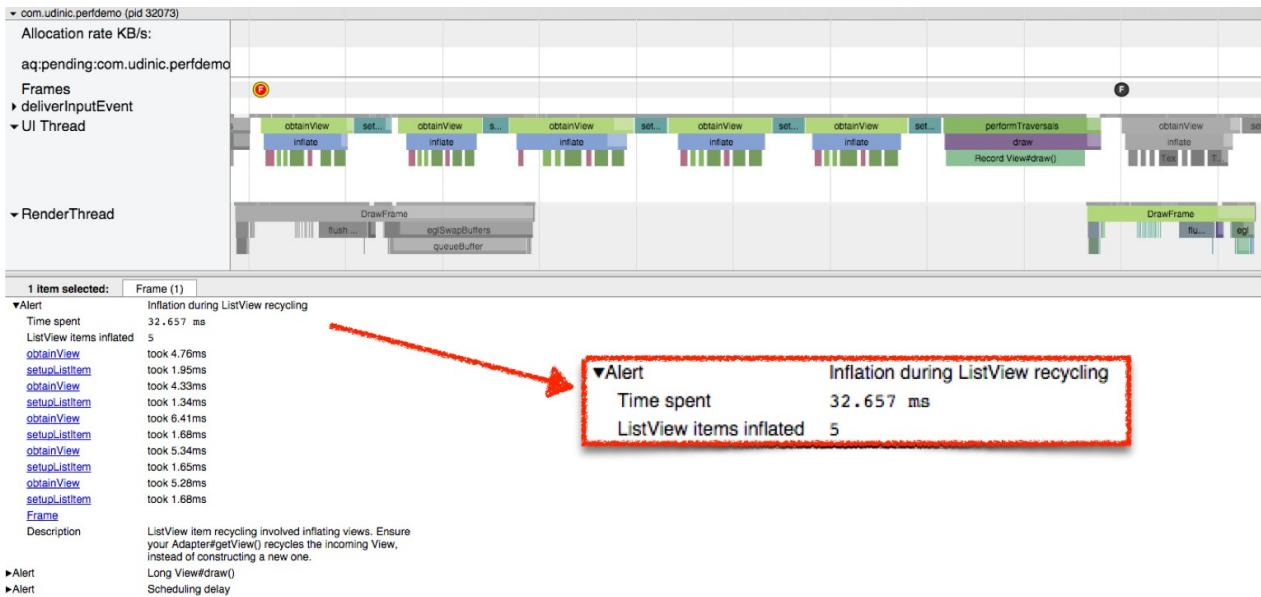
在视频中，我向大家介绍了Systrace中不同区域的功能。当然最有趣的还是Alerts和Frames两栏，它们展示了通过手机来的数据而生成出来的可视化分析结果。让我们来选择最上方的alerts瞧瞧：



这个警告指出了，有一个View#draw()方法执行了比较长的时间。我们可以在下面看到问题的描述，链接，甚至是相关的视频。下面我们看Frames这一行，可以看到这里展示了被绘制出来的每一帧，并且用绿、黄、红三颜色来区分它们在绘制时的性能。我们选一个红色帧来瞅瞅：

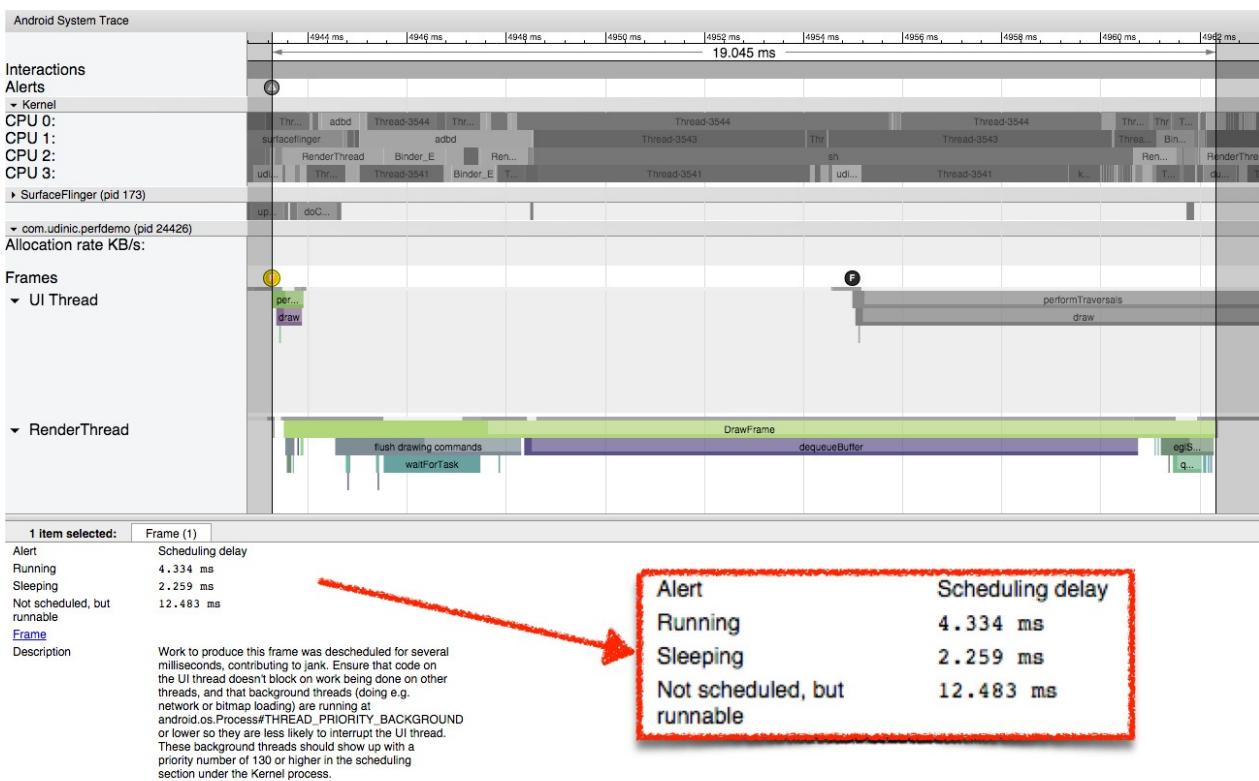


在最下方，我们看到了与这一帧所相关的一些警告。在这三个警告中，有一个是我们上面所提到的（View#draw()） 。接下来我们在这一帧处放大并在下方展开“Inflation during ListView recycling”这条警告：



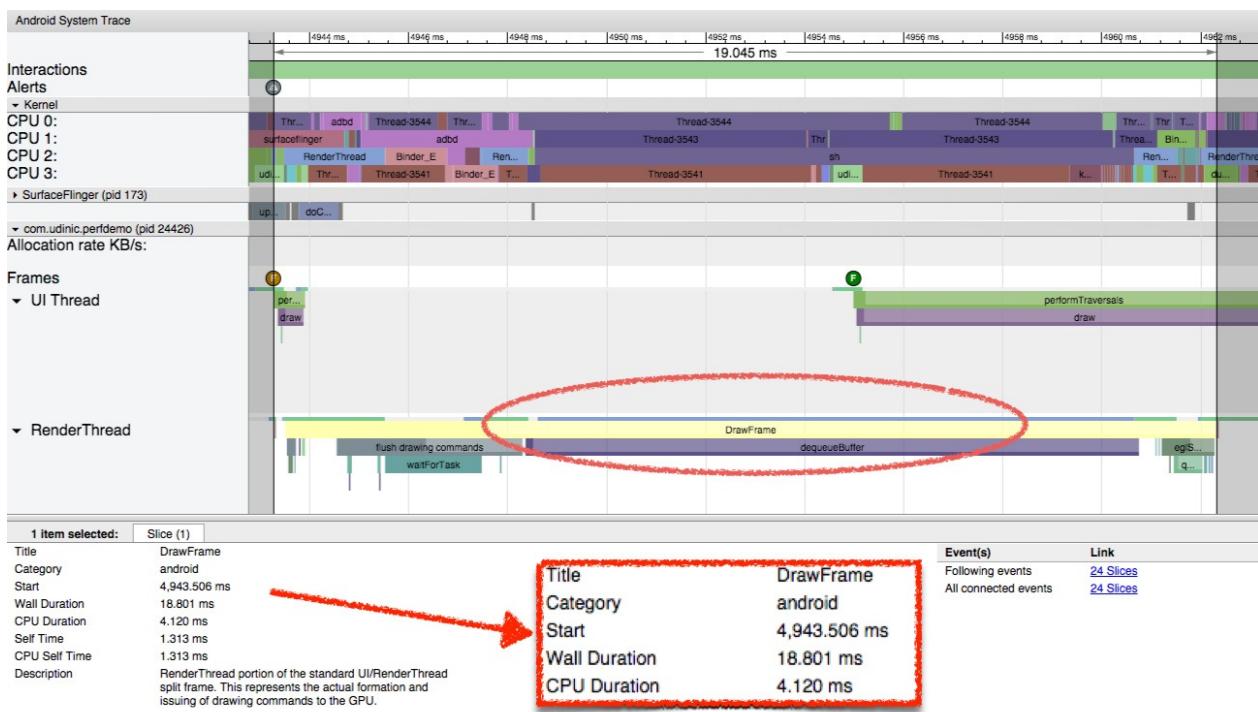
我们可以看到警告部分的总耗时，32毫秒，远高于了我们对保障60fps所需的16毫秒绘制时间。同时还有更多的ListView每个条目的绘制时间，大约是6毫秒每个条目，总共五个。而Description描述项中的内容会帮助我们理解问题，甚至提供问题的解决方案。回到我们上一张图片，我们可以在“inflate”这一个块区处放大，并且观察到底是哪些View在被填充过程中耗时比较严重。

下面是另外一个渲染过慢的实例：



在选择了某一帧之后，我们可以按“m”键来高亮这一帧，并且在上方看到了这一部分的耗时，如图，我们看到了这一阵的绘制总共耗时超过19毫秒。而当我们展开这一帧唯一的一个警告时，我们发现了“Scheduling delay”这条错误。

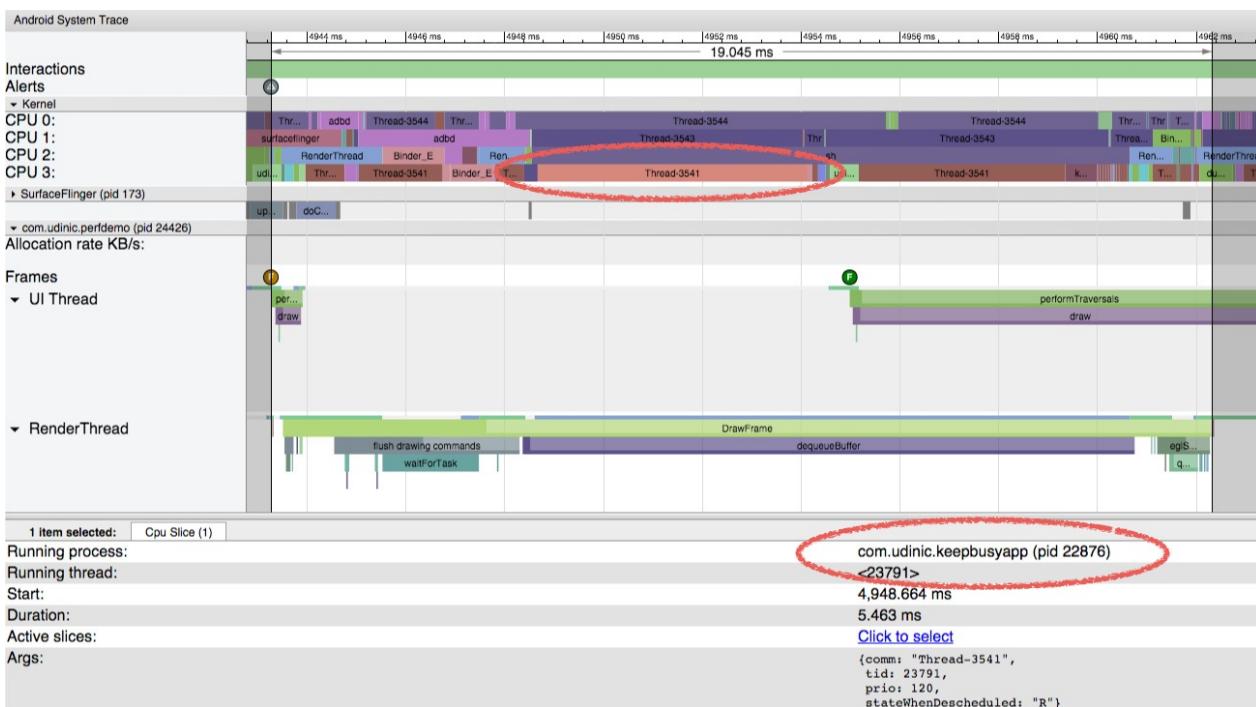
Scheduling delay (调度延迟) 的意思就是一个线程在处理一块运算的时候，在很长一段时间都没有被分配到CPU上面做运算，从而导致这个线程在很长一段时间都没有完成工作。我们选择这一帧中最长的一块，从而得到更加详细的信息：



在红框区域内，我们看到了“**Wall duration**”，他代表着这一区块的开始到结束的耗时。之所以叫作“Wall duration”，是因为他就像是墙上的一个时钟，从线程的一开始就为你计时。

但是，CPU Duration一项中显示了实际CPU在处理这一区块所消耗的时间。

很显然，两个时间的差距还是非常大的。整个区块耗时18毫秒，而在这之中CPU只消耗了4毫秒的时间去运算。这就有点奇怪了，所以我们应该看一下在这整个过程之中，CPU去干吗了。



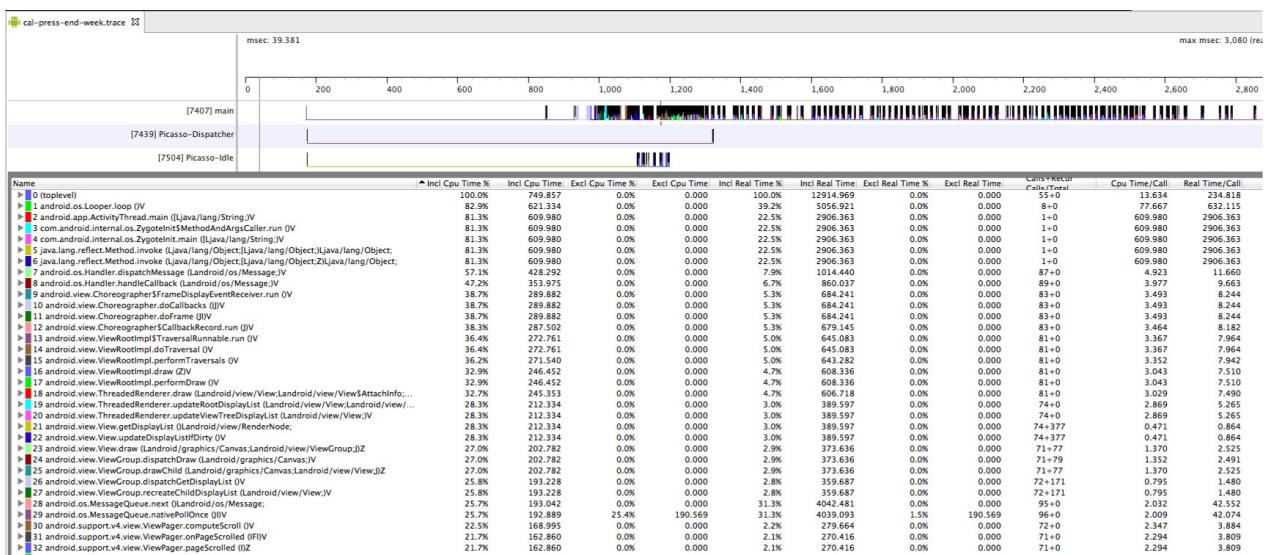
可以看到，所有四个线程都非常的繁忙。

选择其中的一个线程会告诉我们是哪个程序在占用他，在这里是一个包名为 com.udinic.keepbusyapp 的程序。在这里，由于另外一个程序占用CPU，导致了我们的程序未能获得足够的CPU资源。

但是这种情况其实是暂时的，因为被其他后台应用占用CPU的情况并不多见（--），但仍有其他应用的线程或是主线程占用CPU。而Traceview也只能为我们提供一个概览，他的深度是有限的。所以要找到我们app中到底是什么让我们的CPU繁忙，我们还要借助另一个工具——Traceview。

## TraceView

Traceview是一个性能测试工具，展示了所有方法的运行时间。下面让我们来瞅瞅他是啥样的：

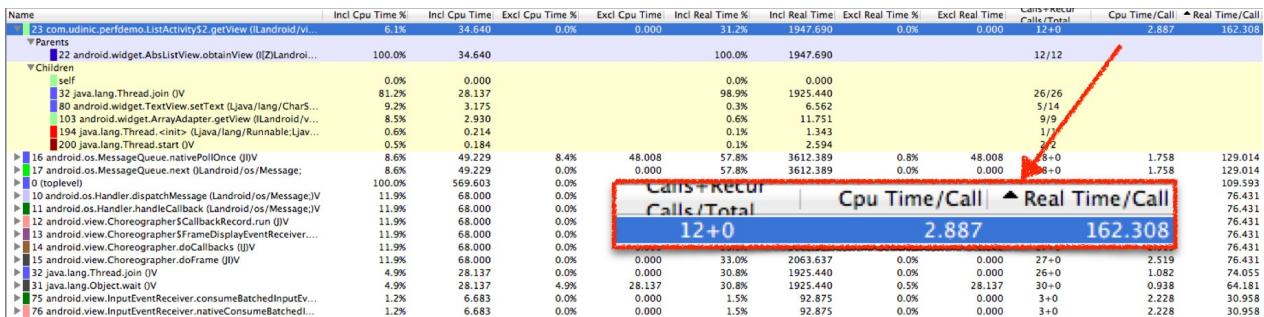


这个工具可以从·Android Device Monitor·中打开也可以通过代码打开。更多的消息信息[请看这里](#)。

下面让我们来看看每一列的含义：

- Name – 方法名，以及他们在上面图表中所对应的颜色。
- Inclusive CPU Time – CPU在处理这个方法以及所有子方法（如被他调用的所有方法）的总耗时。
- Exclusive CPU Time – CPU在处理这一个单独方法的总耗时。
- Inclusive/Exclusive Real Time – 从方法的开始执行到执行结束的总耗时，和Systrace中的“Wall duration”类似
- Calls+Recursion – 这个方法被调用的次数，以及被递归调用的次数。
- CPU/Real time per Call – 在处理这个方法时的CPU耗时的平均值以及实际耗时的平均值。另外的列展示了这个方法所有调用的累计耗时

我打开一个滑动不太顺滑的应用。开启记录，滑动一点后停止记录。展开getView()方法，如下图：

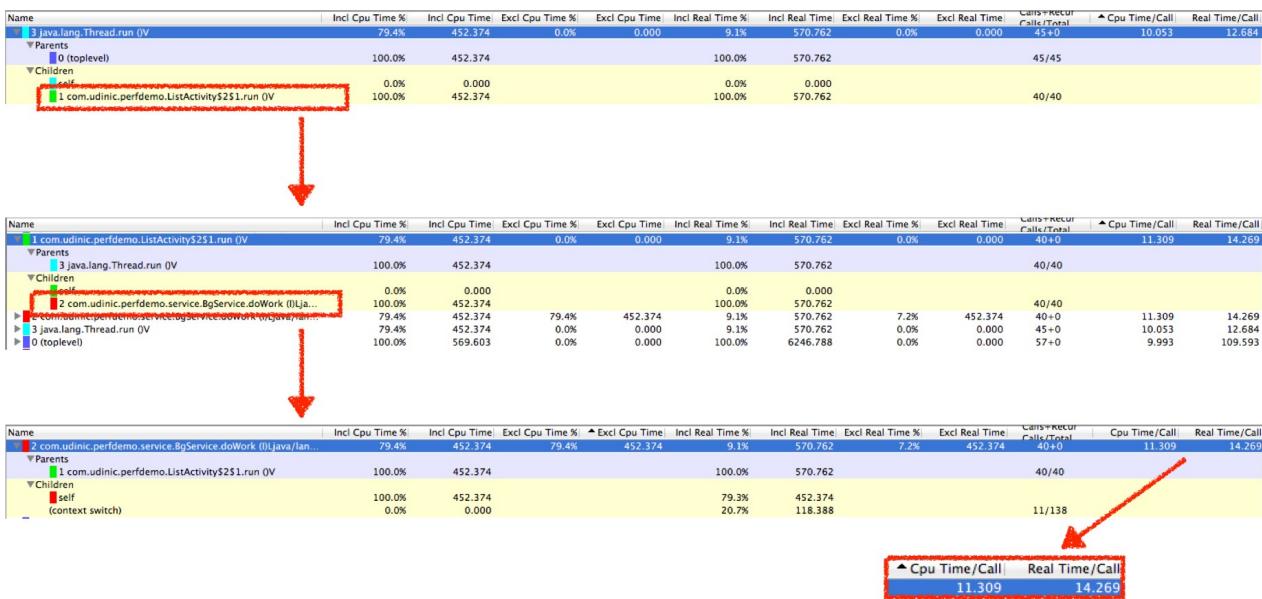


这个方法被调用了12次，每次CPU会消耗3毫秒左右，但是每次调用的总耗时却高达162毫秒！绝对有问题啊！

而看看这个方法的children，我们可以看到这其中的每个方法在耗时方面是如何分布的。Thread.join()方法占据了98%的inclusive real time。这个方法在等待另一个线程结束的时候被调用。在Children中另外一个方法就是Tread.start()方法，而之所以整个方法耗时很长，我猜测是因为在getView()方法中启动了线程并且在等待它的结束。

但是这个线程在哪儿？

我们在getView()方法中并不能看到这个线程做了什么，因为这段逻辑不在getView()方法之中。于是我找到了Thread.run()方法，就是在线程被创建出来时候所运行的方法。而跟随这个方法一路向下，我找到了问题的元凶。



我发现了BgService.doWork()方法的每次调用花费了将近14毫秒，并且有四十个这东西！而且getView()中还有可能调用多次这个方法，这就解释了为什么getView()方法执行时间如此之长。这个方法让CPU长时间的保持在了繁忙状态。而看看Exclusive CPU time，我们可以看到他占据了80%的CPU时间！此外，根据Exclusive CPU time排序，可以帮助我们更好的定位那些耗时很长的方法，而他们很有可能就是造成性能问题的罪魁祸首。

关注这些耗时方法，例如getView()，View#onDraw()等方法，可以很好的帮助我们寻找为什么应用运行缓慢的原因。但有些时候，还会有一些其他的东西来占用宝贵的CPU资源，而这些资源如果被运用在UI的绘制上，也许我们的应用会更加流畅。Garbage Collector垃圾回收机制会不时的运行，回收那些没用的对象，通常来讲这不会影响我们在前台运行的程序。但如果GC被运行的过于频繁，他同样可以影响我们应用的执行效率。而我们该如何知道回收的是否过于频繁了呢...

## 内存调优 Memory Profiling

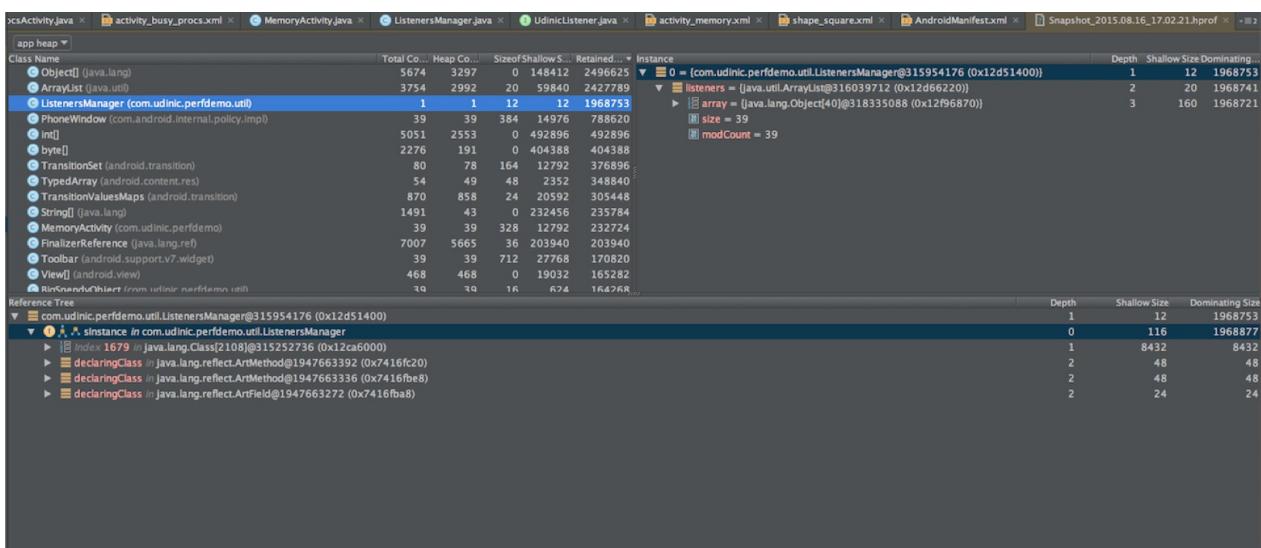
Android Studio在最近的更新中给予了我们更加强大的工具去分析性能问题。在底部Android选项中的Memory选项卡，会显示有多大的数据在什么时候被分配到了堆内存之中，他是长成这个样子的：



而当图表中出现一个小的下滑的时候，说明GC回收发生了，他清除了不必要的对象并且腾出了一定的堆空间。而在这张图表的左侧有两个工具供我们使用， Head dump 和 Allocation Tracker。

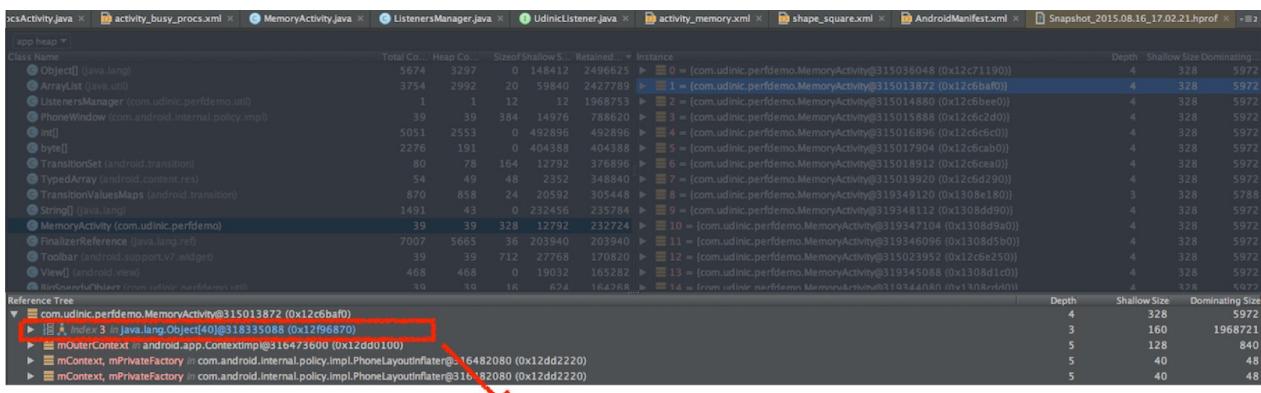
## Heap dump

为了找出到底是什么正在占用我们的堆内存，我们可以使用左边的heap dump按钮。他会提供一个堆内存占用情况的快照，并且会在Android Studio中打开一个单独的报告界面。



在左侧，我们看到一个图标展示了堆中所有的实例，按照类进行分组。而对于每一个实例，会展示有多少个实例的对象被分配到堆中，以及他们的所占用的空间（Shallow size 浅尺寸），以及这些对象在内存中仍然占用的空间，后者告诉了我们多少的内存空间将会被释放如果这些实例被释放。这个工具可以让我们直观的观察处内存是被如何占用的，帮助我们分析我们使用的数据结构和对象之间的关系，以便发现问题并使用更加高效的数据结构，解开和对象之间的关联，并且降低Ratained Memory的占用。而最终目的，就是尽可能的降低我们的内存占用。

回过头来看图表，我们发现MemoryActivity存在39个实例，这对于一个Activity来说有点奇怪。在右边选择其中的一个实例，会在下方看到所有的对这个实例的引用树状列表。



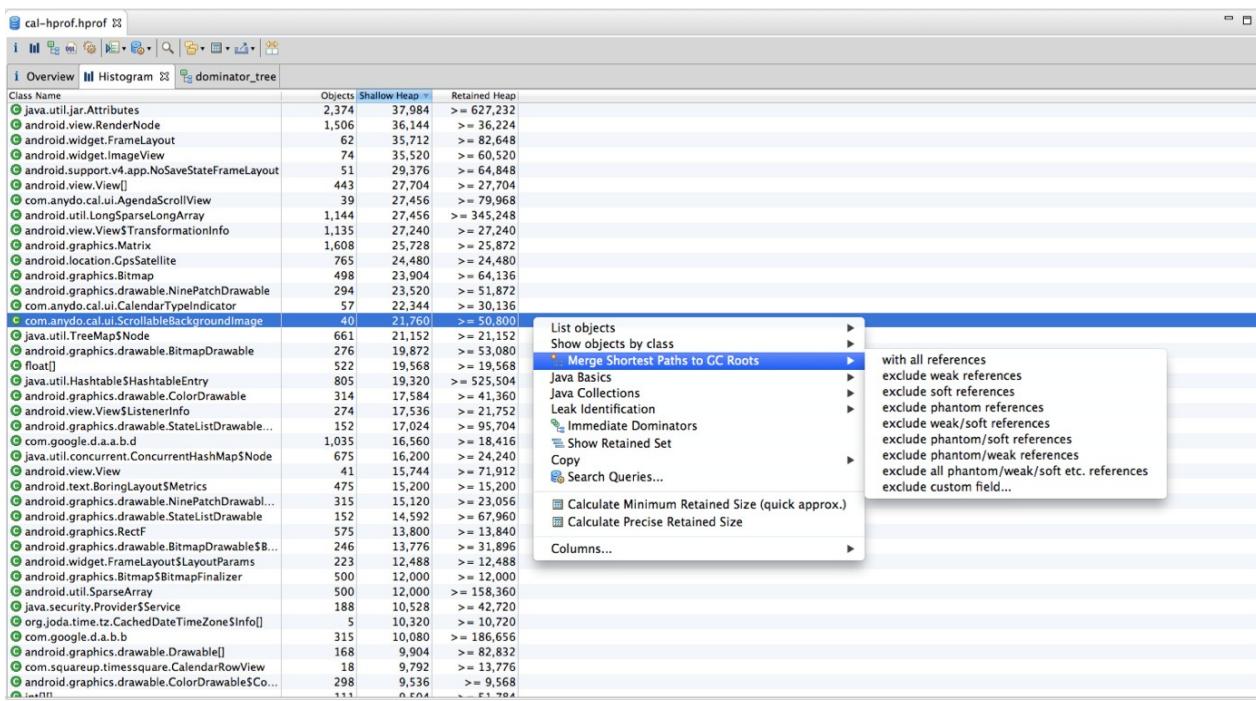
其中一个是ListenersManager对象中的一个集合。而观察这个activity的其他实例，就会他们都因为这个对象而被保留在了内存之中。这也解释了为什么这些对象占用了如此多的内存：



这个现象就叫做“内存泄露”，我们的activity已经被销毁，但是他们的对象却因为始终被引用着而无法被垃圾回收。我们可以避免这种情况，例如确保这些对象再被销毁后不会被其他对象一直引用着。在我们这个例子中，在Activity被销毁后，ListenersManager并不需要保持着对这些对象的引用。所以解决办法就是在onDestroy()回调方法中移除这些引用。

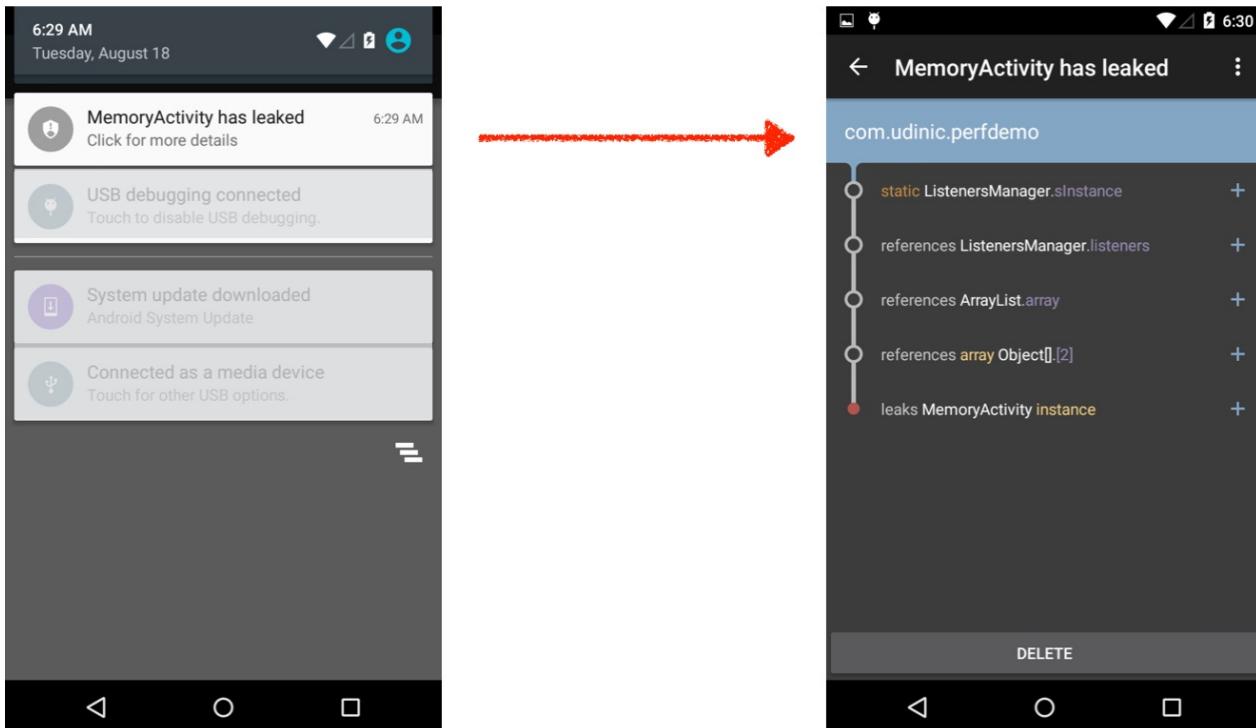
内存泄露以及其他较大的对象会在堆中占据很多的控件，它们减少着可用内存的同时也频繁的造成垃圾回收。而垃圾回收又会造CPU的繁忙，而堆内存并不会变得更大，最终就会导致更悲剧的结果发生：OutOfMemoryException内存溢出，并导致程序崩溃。

另外一个更先进的工具就是 Eclipse Memory Analyzer Tool (Eclipse MAT) :



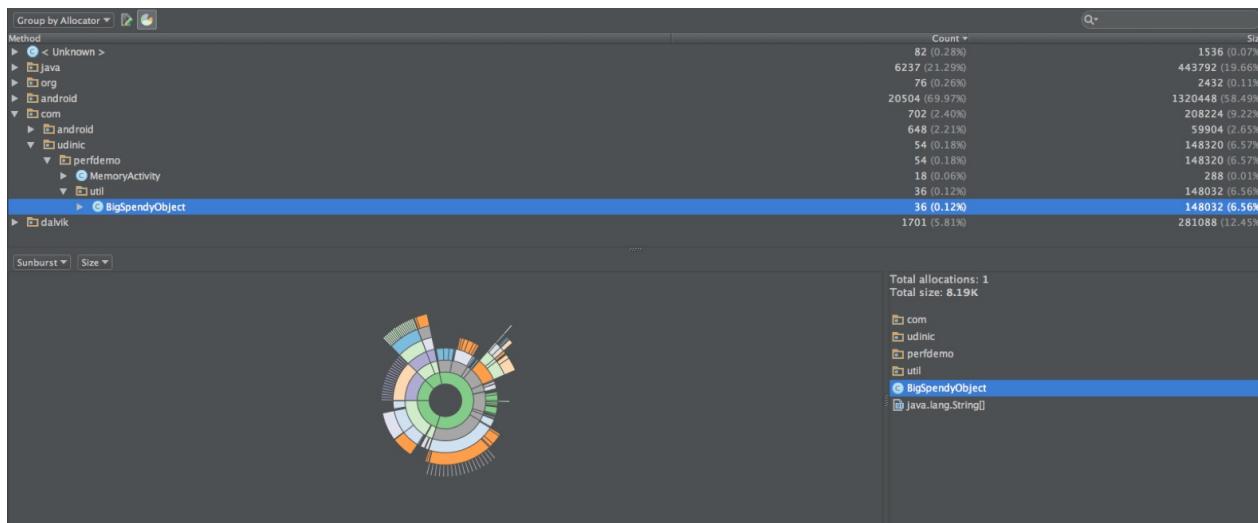
这个工具可以做所有Android Studio可以做的，并且辨别可能出现的内存泄露，以及提供更加高级的搜索功能，例如搜索所有大于2MB的Bitmap实例，或是搜索所有空的Rect对象。

另外一个很好的工具是LeakCanary，是一个第三方库，可以观察应用中的对象并且确保它们没有造成泄漏。而如果造成泄漏了，会有一个推送来提醒你在哪里发生了什么。

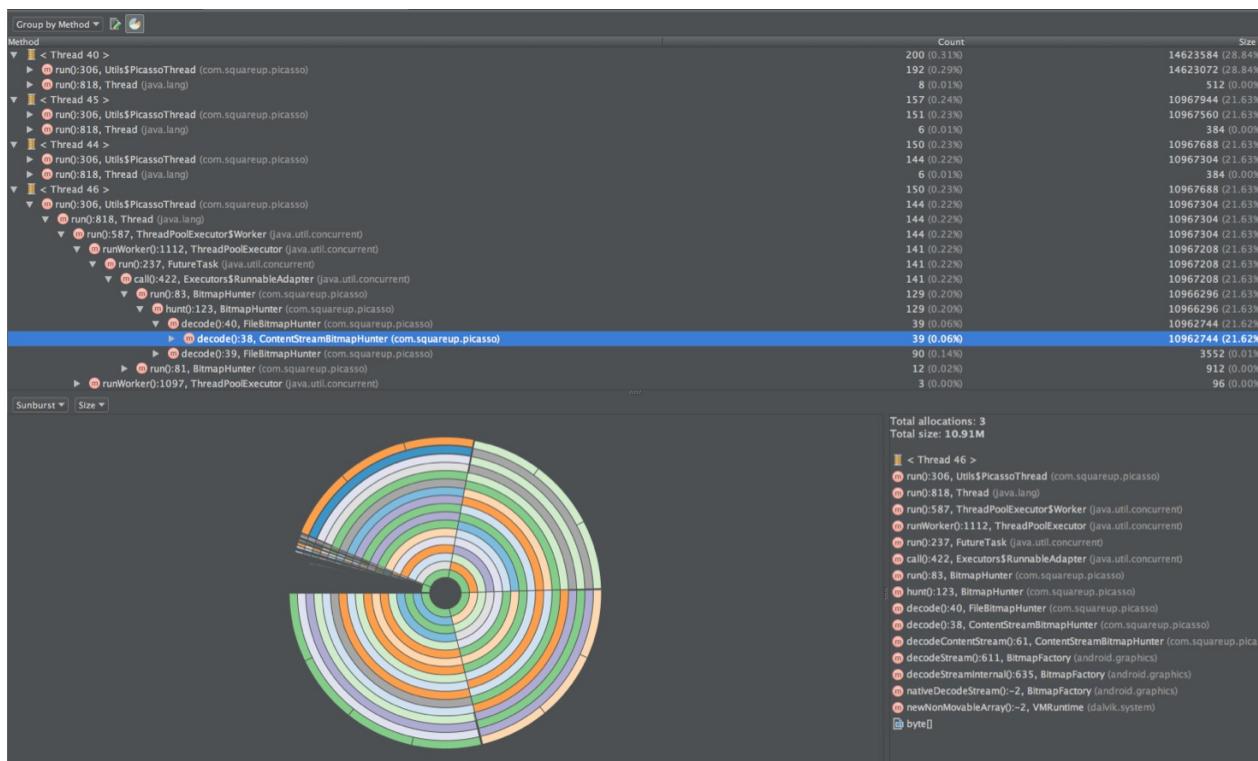


## Allocation Tracker

我们可以在内存图表的左侧找到Allocation Tracker的启动和停止按钮。他会生成一个在一定时间内被生成的所有实例的报告，并且按照类分型分组：



或者按照方法分组：



同时它还能通过美观的可视化界面，告诉我们哪些方法或类拥有最多的实例。

利用这些信息，我们可以找到哪些占用过多内存，引发过多次垃圾回收且又对耗时非常敏感的方法。我们也可以利用这个工具找到很多短命的相同类的实例，从而可以考虑使用对象池的思想去尽量的减少过多的实例被创建。

## 常见内存小技巧

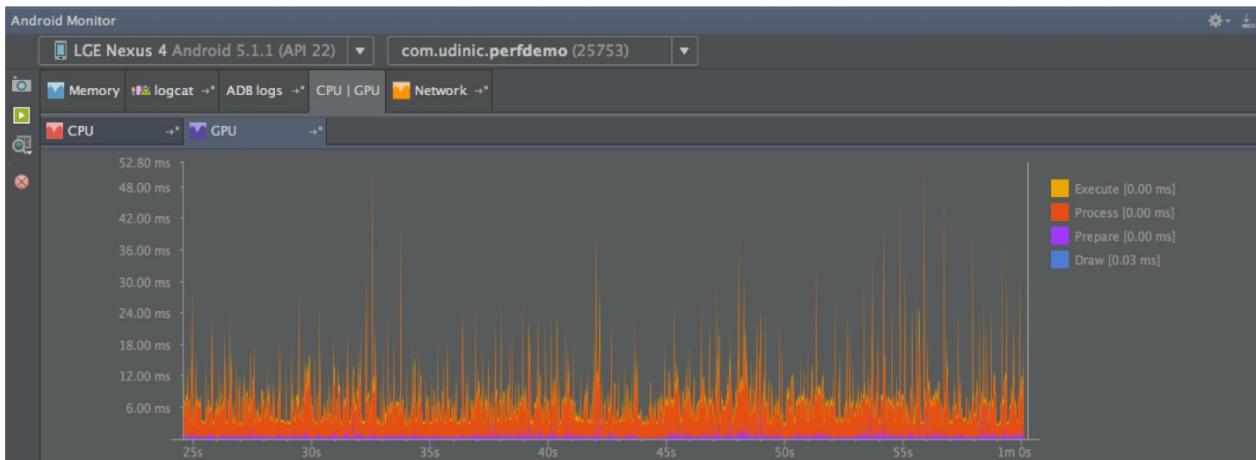
以下是一些我写代码时候遵循的规律或是技巧：

- 枚举在性能问题上一直是一个被经常讨论的话题。这里是一个[讨论枚举的视频](#)，指出了枚举所消耗的内存空间，这还有一段[关于这个视频的讨论](#)，当然其中存在着一些误导。但是回过头来，枚举真的比一般的常量更加占用空间吗？肯定的。但是这一定不好吗？未必。如果你在编写一个library库并且需要很强的类型安全性，那么也许可以使用枚举而非其他办法，例如[@IntDef](#)。而如果你只是一堆的常量，使用枚举也许就不能么明智了。还是那句话，在你做决定之前一定要权衡与取舍。
- 自动装箱 – 自动装箱是一个从原始数据类型到对象型数据的装箱过程（例如int到Integer）。每当一个原始类型数据被装箱到一个对象类型数据，一个新的对象就产生了（震惊吧。。。）。所以如果发生了很多次的自动装箱，势必会加快GC的执行频率，而且自动装箱是很容易被我们忽视的。而解决办法，在使用这些类型的时候尽量一致，如果你在应用中完全使用原始数据类型，那么尽量避免他被无缘无故的自动封装。你可以使用我们上面提到的memory profiling工具去寻找这些过于大量的对象类型数据，也可以通过Traceview去寻找类似Integer.valueOf(), Long.valueOf()这样的方法来判断是否发生了大量不必要的自动封装。
- **HashMap vs ArrayMap / Sparse\*Array** – 既然提到了自动装箱的问题，那么使用HashMap的话，就需要我们使用对象类型作为键。而如果我们在整个应用中使用的都是基本数据类型的“int”，那么在我们使用HashMap时候就会发生自动装箱，而这时也许我们就可以考虑使用SparseIntArray。而假如我们仍然需要键为对象类型，那么我们可以使用ArrayMap。ArrayMap和HashMap很类似，但是在[底层的实现原理却不尽相同](#)，这也会让我们更加高效的使用内存，但要付出一定的性能代价。两种方法都会比HashMap更加节省内存空间，但是相比于HashMap，查询和增删的速度上会有一定的牺牲。当然，除非你具有至少1000条的数据源，否则在运行时也不会对速度造成太大的影响，这也是你使用他们替代HashMap的原因之一。
- 注意**Context** – 在我们前面也看到了，Activity是非常容易造成内存泄露的。在Android中，最容易造成内存泄露的当属Activity。并且这些内存泄露会浪费大量的内存，因为他们持有者他们UI中所有的View，而这些View通常会占据很多的控件。在开发过程中的很多操作需要Context，而我通常也会使用Activity来传递。所以一定要搞清楚你对这个Activity做了什么。如果一个引用被缓存起来了，且这个对象的生命周期比你的Activity还要长，那么在我们解除这个引用之前，就会造成内存泄露了。
- 避免非静态 – 当我们创建非静态内部类，并且初始化它的时候，在其内部会创建一个外部类的隐式引用。而如果内部类的生命周期比外部类还要长，那么外部类也同样会被保留在内存之中，尽管我们已经完全不需要它了。例如，在Activity内创建了一个继

承自AsyncTask的内部类，完后在Activity运行的时候启动这个async task，再杀掉Activity。那么这时候这个async task会保持着这个Activity直到执行结束。而解决办法也很简单，不要这么做，尽量使用静态内部类。

## 监测GPU (GPU Profiling)

在AndroidStudio 1.4中的一个全新工具，就是可以查看GPU绘制。

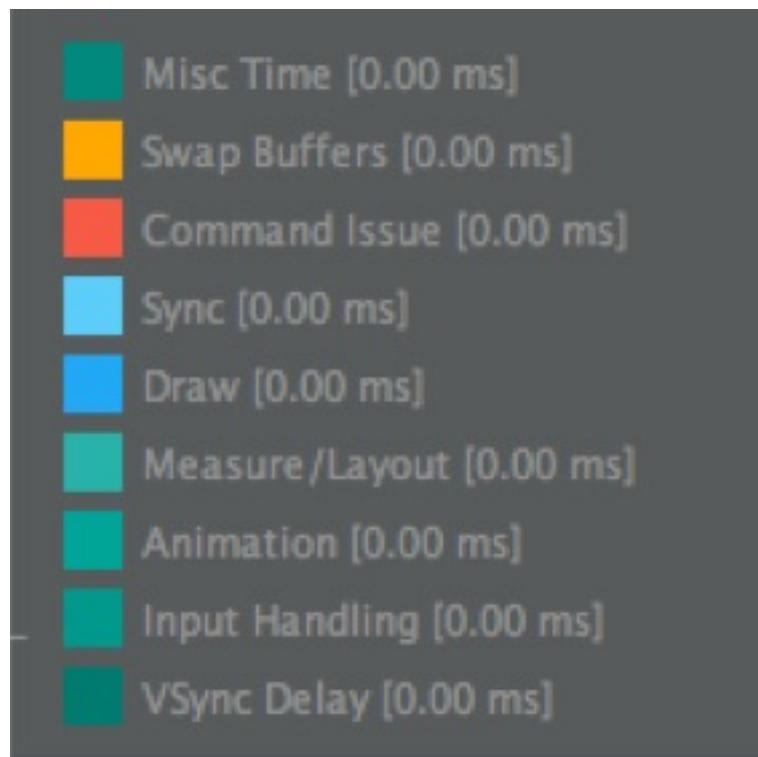


每一条线意味着一帧被绘制出来，而每条线中的不同颜色又代表着在绘制过程中的不同阶段：

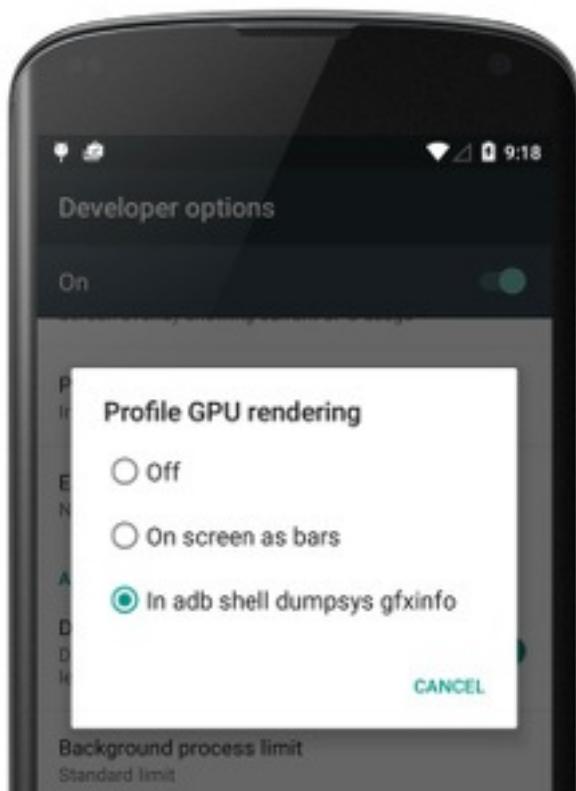
- **Draw (蓝色)** - 代表着View#onDraw()方法。在这个环节会创建/刷新DisplayList中的对象，这些对象在后面会被转换成GPU可以明白的OpenGL命令。而这个值比较高可能是因为view比较复杂，需要更多的时间去创建他们的display list，或者是因为有太多的view在很短的时间内被创建。
- **Prepare (紫色)** – 在Lollipop版本中，一个新的线程被加入到了UI线程中来帮助UI的绘制。这个线程叫作RenderThread。它负责转换display list到OpenGL命令并且送至GPU。在这过程中，UI线程可以继续开始处理后面的帧。而在UI线程将所有资源传递给RenderThread过程中所消耗的时间，就是紫色阶段所消耗的时间。如果在这过程中有很多的资源都要进行传递，display list会变得过多过于沉重，从而导致在这一阶段过长的耗时。
- **Process (红色)** – 执行Display list中的内容并创建OpenGL命令。如果有过多或者过于复杂的display list需要执行的话，那么这阶段会消耗较长的时间，因为这样的话会有很多的view被重绘。而重绘往往发生在界面的刷新或是被移动出了被覆盖的区域。
- **Execute (黄色)** – 发送OpenGL命令到GPU。这个阶段是一个阻塞调用，因为CPU在这里只会发送一个含有一些OpenGL命令的缓冲区给GPU，并且等待GPU返回空的缓冲区以便再次传递下一帧的OpenGL命令。而这些缓冲区的总量是一定的，如果GPU太过于繁忙，那么CPU则会去等待下一个空缓冲区。所以，如果我们看到这一阶段耗时比较长，那可能是因为GPU过于繁忙的绘制UI，而造成这个的原因则可能是在短时

间内绘制了过于复杂的view。

在Marshmallow版本中，有更多的颜色被加了进来，例如Measure/Layout阶段，input handling输入处理，以及一些其他的：



在使用这些功能之前，你需要在开发者选项中开启GPU rendering(GPU呈现模式分析)：



接下来我们就可以通过以下这条adb命令得到我们想要得到的所有信息：

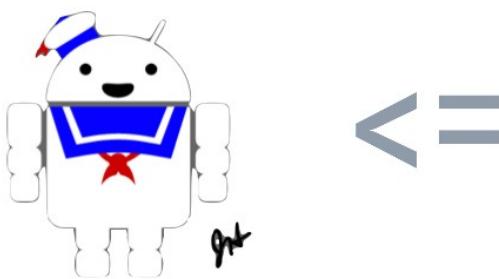
```
adb shell dumpsys gfxinfo<PACKAGE_NAME>
```

我们可以自己收集这些信息并创建图表。这个命令也会打印出一些其他有用的信息，例如view层级中的层数，display lists的大小等等。在Marshmallow中，我们也会得到更多的信息：

```
Profile data in ms:

com.udinic.perfdemo/com.udinic.perfdemo.BusyProcsActivity/android.view.ViewRootImpl@21bc4c3
Draw Prepare Process Execute
0.03 1.56 9.28 1.34
0.00 0.67 15.41 0.98
0.03 0.21 6.90 1.04
0.00 0.24 6.32 0.70
0.03 0.43 6.38 0.67
0.00 0.40 2.87 0.49
0.00 3.17 3.11 0.43
0.00 0.31 3.20 0.43
0.00 0.31 2.84 0.40
0.03 0.31 14.31 0.49

View hierarchy:
com.udinic.perfdemo/com.udinic.perfdemo.BusyProcsActivity/android.view.ViewRootImpl@21bc4c3
17 views, 17.86 kB of display lists
```

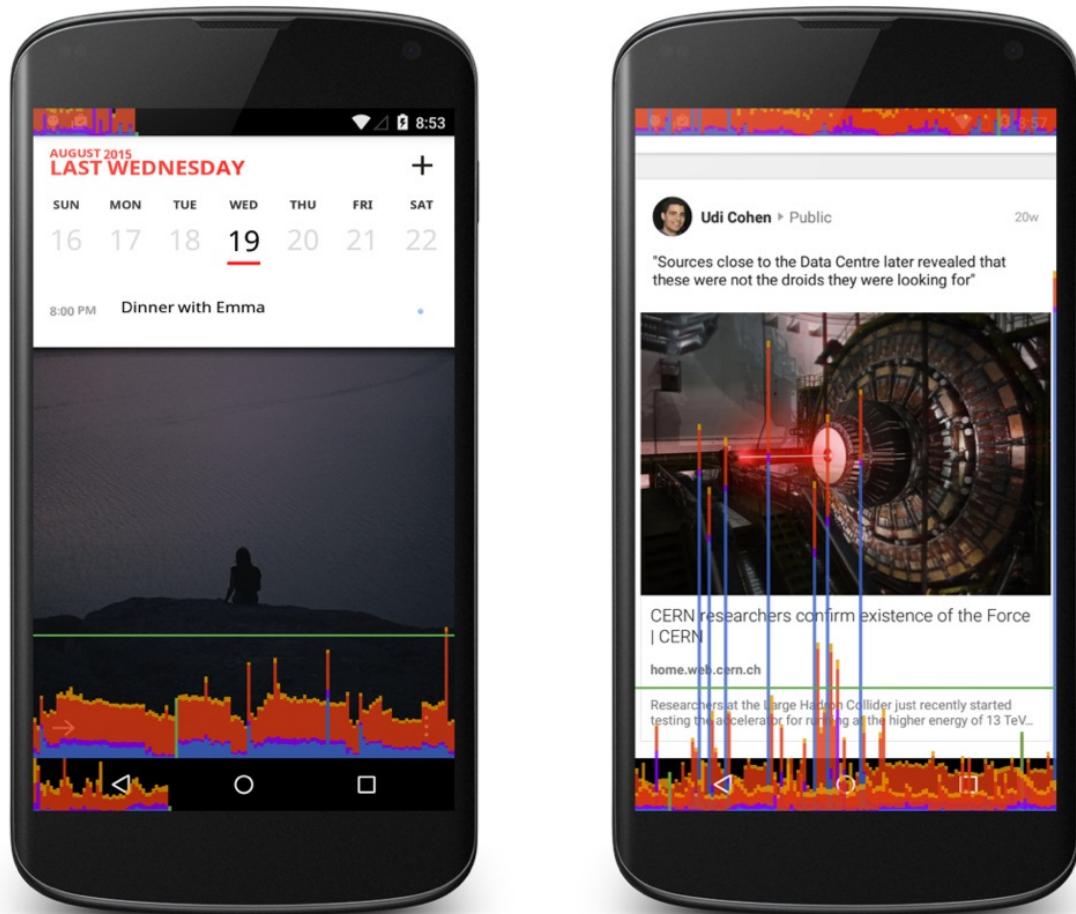


```
Total frames rendered: 2887
Janky frames: 151 (5.23%)
90th percentile: 10ms
95th percentile: 19ms
99th percentile: 26ms
Number Missed Vsync: 5
Number High input latency: 0
Number Slow UI thread: 137
Number Slow bitmap uploads: 2
Number Slow draw: 143
```

如果我们要自动化测试我们的app，那么我们可以自己创建服务器去运行在特定节点执行这些命令（如列表滚动，重度动画等），并观察这些数值的变动。这可以帮助我们找出在哪里出现了性能的下降，并且产品上线之前找到问题的所在。我们也能够通过“framestats”关键字来找到更多更加精确的数据，这里有[更详尽的解释](#)。

但这可不是获取GPU Rendering数据的唯一方式！

我们在开发者选项中看过了GPU呈现模式分析内的Profile GPU Rendering”选项后，还有另外一个选项就是”On screen as bars”（在屏幕上显示为条形图）。打开这个后，我们就可以直观的看到每一帧在绘制过程中所消耗的时间，绿色的横线则代表16ms的60fps零界值。

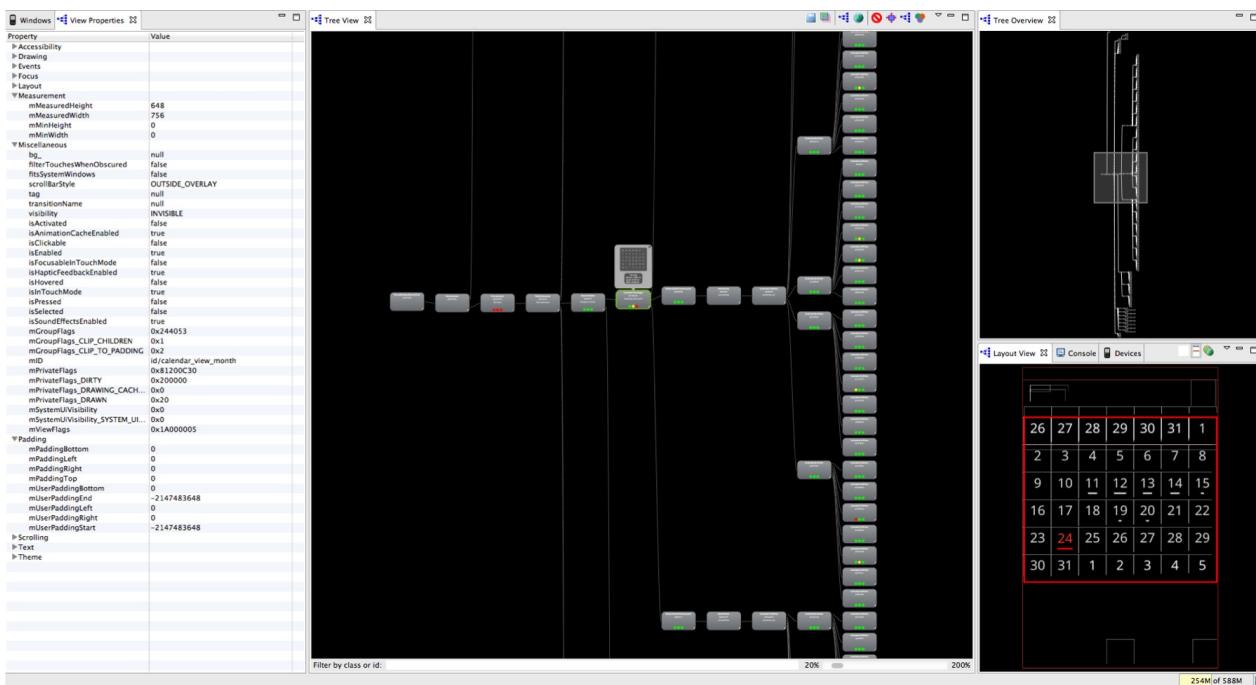


在右边的例子中，我们可以看到很多帧都超出了绿线，这也意味着它花了多余16毫秒的时间去绘制。而蓝色占据了这些线条的主体，我们知道这可能是因为过多或是过于复杂的view在被绘制。在这种情况下，当我滑动列表，因为列表中view的结构比较复杂，有一些view已经被绘制完成而一些因为过于复杂还处于绘制阶段，而这可能就是造成这些帧超过绿线的原因——绘制起来实在太复杂了。

## Hierarchy Viewer

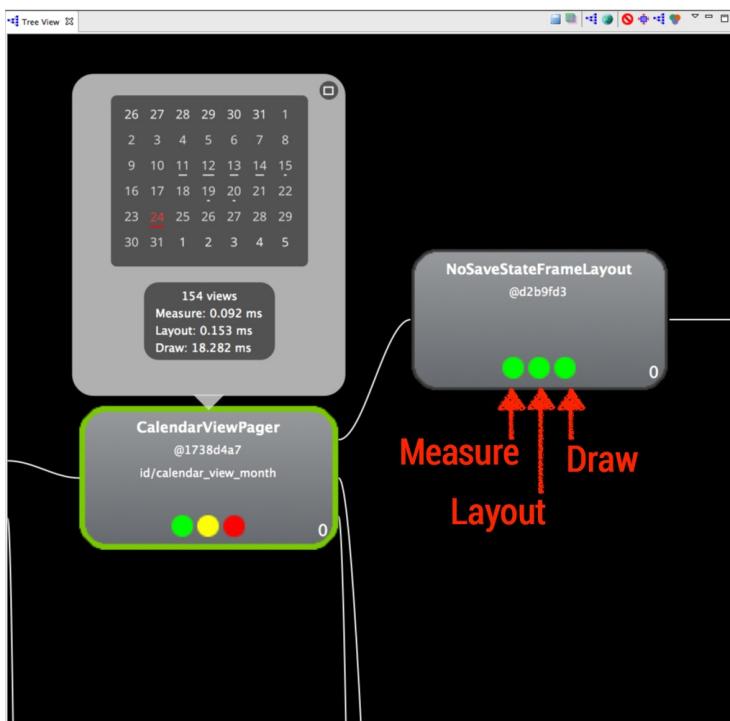
我非常喜欢这个工具，同时也因为那么多人完全不用而感到一丝的悲凉。

使用Hierarchy Viewer，我们可以获得性能数据，观察View层级中的每一个View，并且可以看到所有View的属性。我们同样可以导出theme数据，这样可以看到每一个style中的属性值，但是我们只能在单独运行Hierarchy Viewer的时候才能这么干，而非通过Android Monitor。通常在我进行布局设计以及布局优化的时候，我会使用到这个工具。



在正中间我们看到的树状结构就代表了View的层级。View的层级可以很宽，但如果太宽的话（10层左右），也许会在布局和测量阶段消耗大量的性能。在每一次View通过View#onMeasure()方法测量的时候，或是通过View#onLayout()方法布局他的所有子view的时候，这些方法又回传递到它所有的子view上面并且重头来过。有的布局会将上述步骤做两次，例如RelativeLayout以及某些通过配置的LinearLayout，而如果它们又层层嵌套，那么这些方法的传递会大量的增加。

在右下方，我们看到了一个我们布局的“蓝图”，标注了每一个view的位置。当我们点击这里（或者从树状结构中），我们会在左侧看到他所有的属性。在设计布局时候，有时候我不确定为什么一个view被摆在那里，而使用这个工具，我可以在树状图中找到这个view，选择，并观察他在预览窗口中的位置。我还通过view在屏幕上最终的绘制尺寸，来设计有趣的动画，并且使用这些信息让动画或者View的位置更加的精准。我也可以通过这个工具来寻找被其他View不小心盖住从而找不到的View，等等等等。

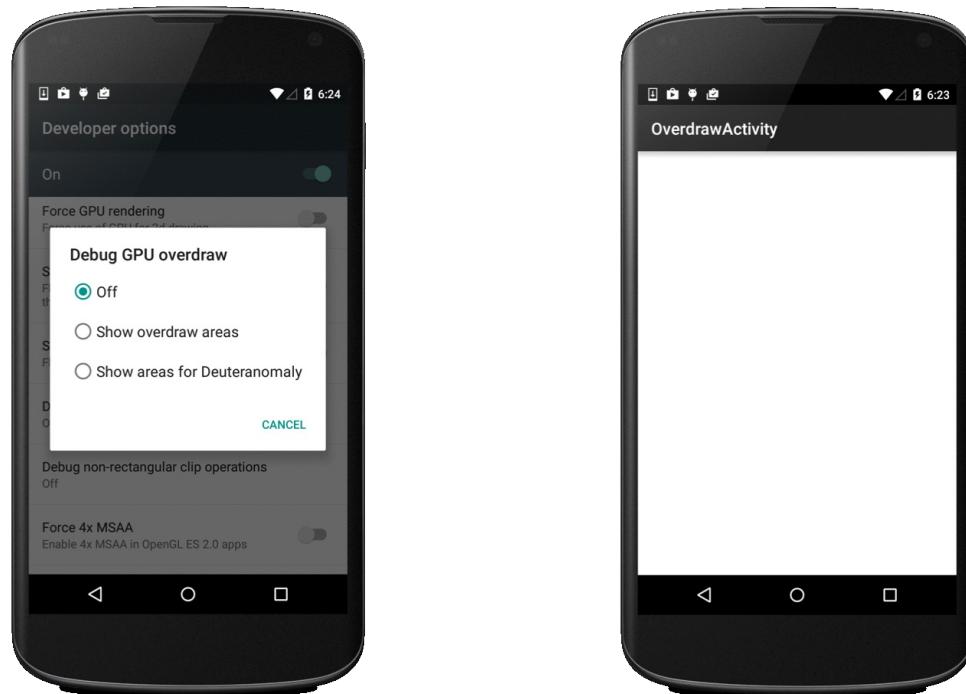


Green: view is in the faster 50%  
 Yellow: view is in the slower 50%  
 Red: view is the slowest in the tree

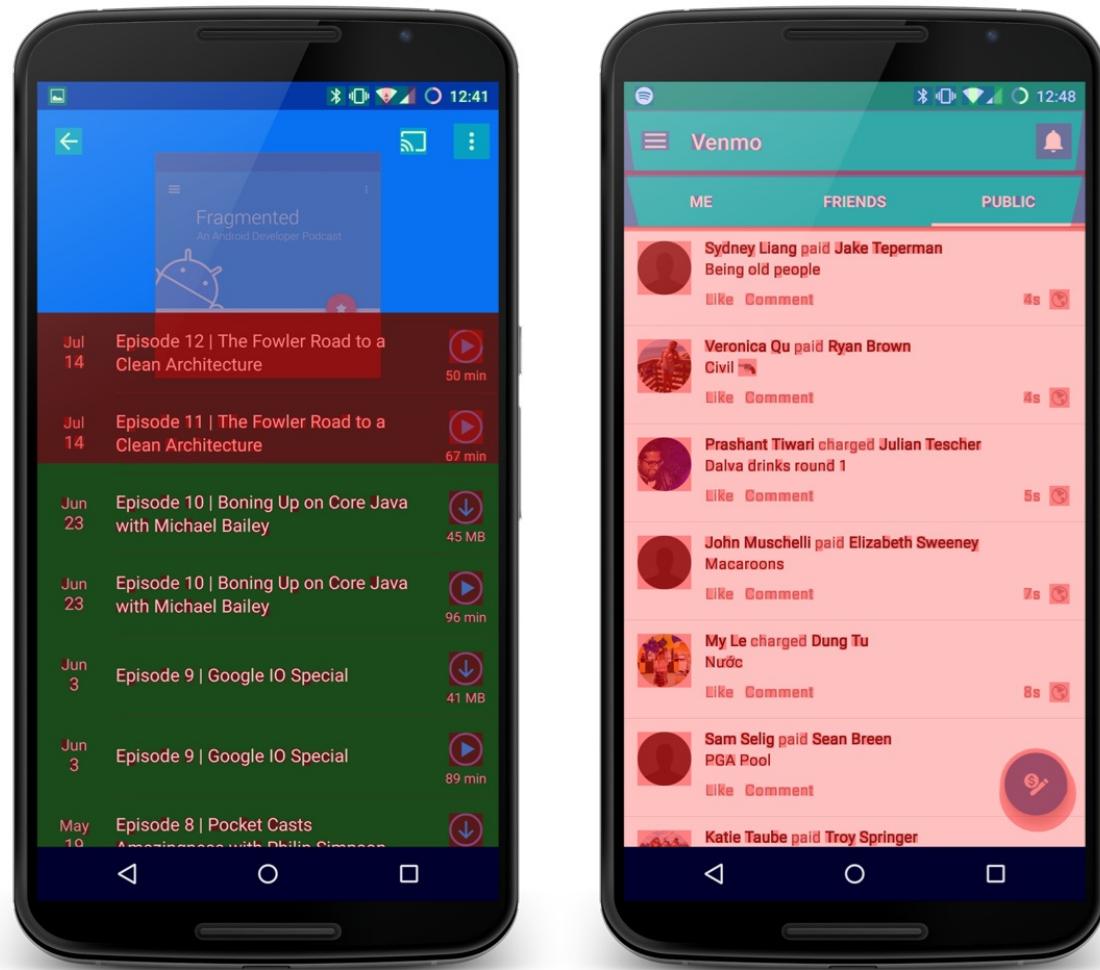
对于每一个view我们可以获得他测量、布局以及绘制的用时和它所包含的所有子view。在这里颜色代表了这个view在绘制过程中，相比树中其他view的性能表现，这是我们找到这些性能不足view的最佳途径。鉴于我们能够看到所有view的预览，我们可以沿着树状图，跟随view被创建的顺序，找寻那些可以被舍弃的多余步骤。而其中之一，也是对性能影响非常大的，就是过度绘制。

## 过度绘制

正如我们在GPU Profiling部分看到的，在Execute黄色阶段，如果GPU有过多的东西要在屏幕上绘制，整个阶段会消耗更多的时间，同事也增加了每一帧所消耗的时间。过度绘制往往发生在我们需要在一个东西上面绘制另外一个东西，例如在一个红色的背景上画一个黄色的按钮。那么GPU就需要先画出红色背景，再在他上面绘制黄色按钮，此时过度绘制就是不可避免的了。如果我们有太多层需要绘制，那么则会过度的占用GPU导致我们每帧消耗的时间超过16毫秒。



使用“Debug GPU Overdraw”（调试过度绘制）功能，所有的过度绘制会以不同颜色的形式展示在屏幕上。1x或是2x的过度绘制没啥问题，即便是一小块浅红色区域也不算太坏，但如果我们看到太多的红色区域在屏幕上，那可能就有问题了，让我们来看几个例子：



在左边的例子中，我们看到列表部分是绿色的，通常还OK，但是在上方发生覆盖的区域是一片红色，那就有问题了。在右边的例子中，整个列表都是浅红色。在两个例子中，都各有一个不透明的列表存在 $2x$ 或 $3x$ 的过度绘制。这些过度绘制可能发生在我们给Activity或Fragment设置了全屏的背景，同时又给ListView以及ListView的条目设置了背景色。而通过只设置一次背景色即可解决这样的问题。

注意：默认的主题会为你指定一个默认的全屏背景色，如果你的activity又一个不透平的背景盖住了默认的背景色，那么你可以移除主题默认的背景色，这样也会移除一层的过度绘制。这可以通过配置主题配置或是通过代码的方法，在onCreate()方法中调用getWindow().setBackgroundDrawable(null)方法来实现。

而使用Hierarchy Viewer，你可以导出一个所有view层级的PSD文件，在Photoshop中打开，并且调查不同的layout以及不同的层级，也能够发现一些在布局中存在的过度绘制。而使用这些信息可以移除不必要的过度绘制。而且，不要看到绿色就满足了，冲着蓝色去！

# 透明度

使用透明度可能会影响性能，但是要去理解为什么，让我们瞅瞅当我们给view设置透明度的时候到底发生了什么。我们来看一下下面这个布局：



我们看到这个layout中又三个ImageView并且重叠摆放。在使用最常规的设置透明度的方法setAlpha()时，方法会传递到每一个子view上面，在这里是每一个ImageView。而后，这些ImageView会携带新的透明值被绘制入帧缓冲区。而结果就是：



这并不是我们想要看到的结果。

因为每一个ImageView都被赋予了一个透明值，导致了本应覆盖的部分被融合在一起。幸运的是，系统为我们解决了这个问题。布局会被复制到一个非屏幕区域缓冲区中，并且以一个整体来接收透明度，其结果再被复制到帧缓冲区。结果就是：



但是，我们是要付出性能上面的代价的。

假如在帧缓冲区内绘制之前，还要在off-screen缓冲区中绘制一遍的话，相当于增加了一层不可见的绘制层。而系统并不知道我们是希望这个透明度以何种的形式去展现，所以系统通常会采用相对复杂的一种。但是也有很多设置透明度的方法能够避免在off-screen缓冲区中的复杂操作：

- TextView – 使用setTextColor()方法替代setAlpha()。这种方法使用Alpha通道来改变字色，字也会直接使用它进行绘制。
- ImageView – 使用setImageAlpha()方法替代setAlpha()。原理同上。
- 自定义控件 – 如果你的自定义控件并不支持相互覆盖，那就无所谓了。所有的子view并不会像上面的例子中一样，因为覆盖而相互融合。而通过复写hasOverlappingRendering()并将其返回false后，便会通知系统使用最直接的方式绘制view。同时我们也可以通过复写onSetAlpha()返回true来手动操控设置透明度后的逻辑。

## 硬件加速

在Honeycomb版本中引入了硬件加速（Hardware Acceleration）后，我们的应用在绘制的时候就有了全新的[绘制模型](#)。它引入了DisplayList结构，用来记录View的绘制命令，以便更快的进行渲染。但还有一些很好的功能开发者们往往会忽略或者使用不当——View layers。

使用View layers（硬件层），我们可以将view渲染入一个非屏幕区域缓冲区（off-screen buffer，前面透明度部分提到过），并且根据我们的需求来操控它。这个功能主要是针对动画，因为它能让复杂的动画效果更加的流畅。而不使用硬件层的话，View会在动画属性（例如coordinate, scale, alpha值等）改变之后进行一次刷新。而对于相对复杂的view，这一次刷新又会连带它所有的子view进行刷新，并各自重新绘制，相当的耗费性能。使用View layers，通过调用硬件层，GPU直接为我们的view创建一个结构，并且不会造成view的刷新。而我们可以在避免刷新的情况下对这个结构进行很多种的操作，例如

x/y位置变换，旋转，透明度等等。总之，这意味着我们可以对一个让一个复杂view执行动画的同时，又不会刷新！这会让动画看起来更加的流畅。下面这段代码我们该如何操作：

```
// Using the Object animator

view.setLayerType(View.LAYER_TYPE_HARDWARE, null);

ObjectAnimator objectAnimator=ObjectAnimator.ofFloat(view, View.TRANSLATION_X, 20f);

objectAnimator.addListener(new AnimatorListenerAdapter(){

@Override

public void onAnimationEnd(Animator animation){

view.setLayerType(View.LAYER_TYPE_NONE, null);

});

objectAnimator.start();

// Using the Property animator

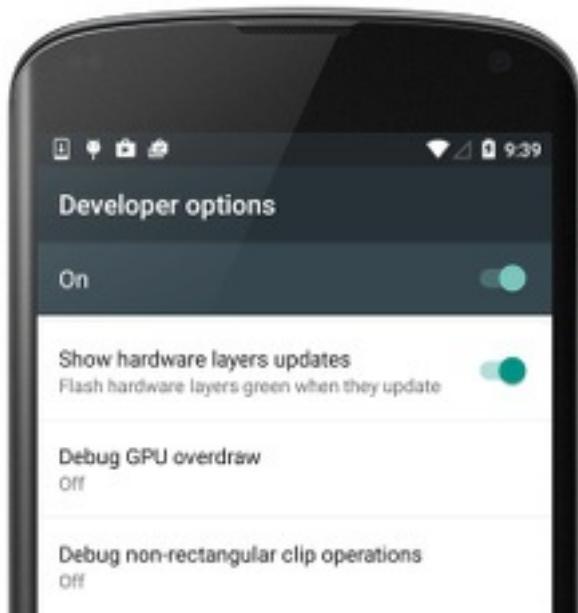
view.animate().translationX(20f).withLayer().start();
```

很简单，对吧？

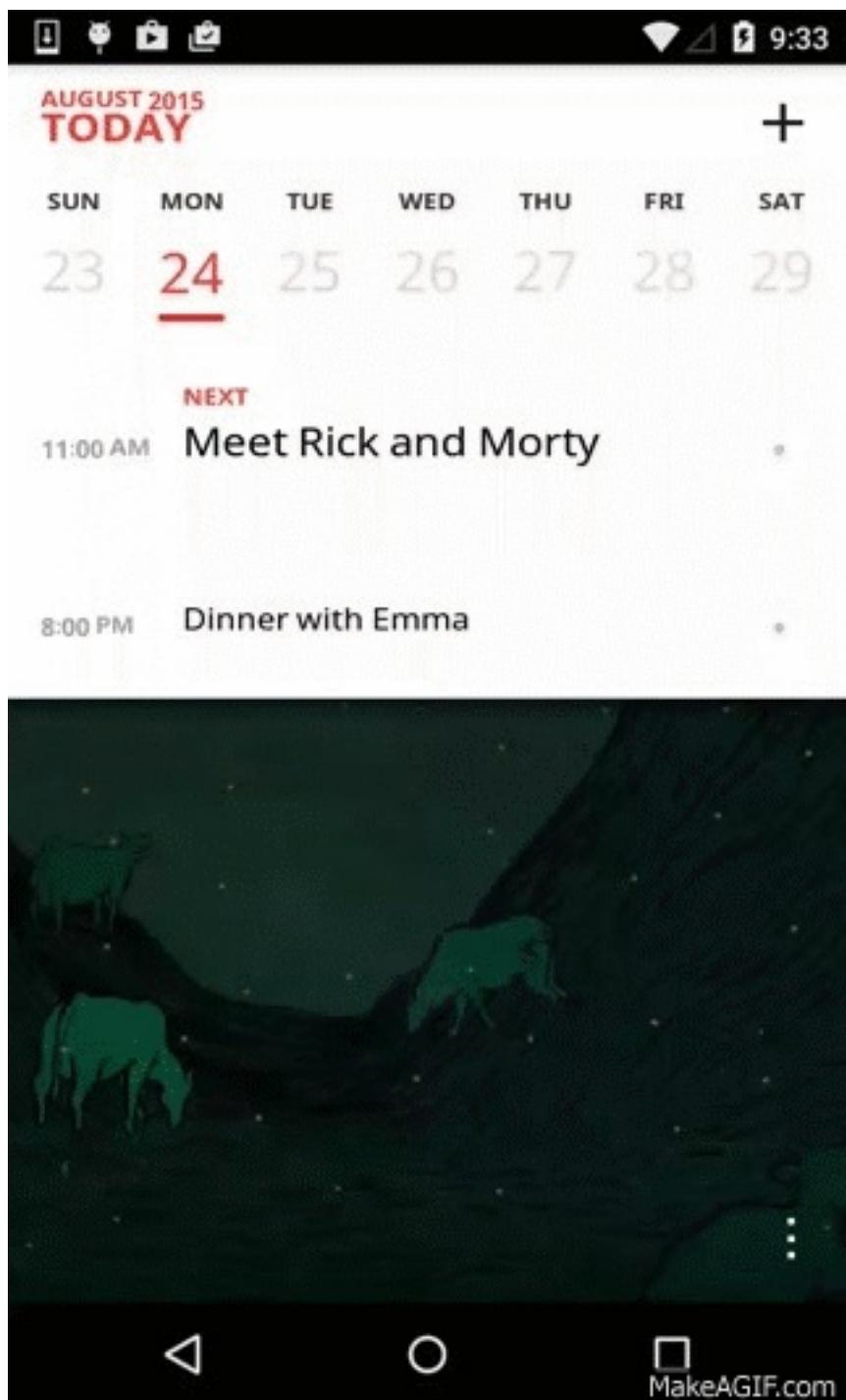
是的，但是再使用硬件layers的时候还是有几点要牢记在心：

- **回收** – 硬件层会占用GPU中的一块内存。只在必要的时候使用他们，比如动画，并且事后注意回收。例如在上面ObjectAnimator的例子中，我们增加了一个动画结束监听以便在动画结束后可以移除硬件层。而在Property animator的例子中，我们使用了withLayers()，这会在动画开始时候自动创建硬件层并且在结束的时候自动移除。
- 如果你在调用了硬件View layers后改变了View，那么会造成硬件硬件层的刷新并且再次重头渲染一遍view到非屏幕区域缓存中。这种情况通常发生在我们使用了硬件层暂时还不支持的属性（目前为止，硬件层只针对以下几种属性做了优化：rotation、scale、x/y、translation、pivot和alpha）。例如，如果你另一个view执行动画，并且使用硬件层，在屏幕滑动他们的同时改变他的背景颜色，这就会造成硬件层的持续刷新。而以硬件层的持续刷新所造成的性能消耗来说，可能让它在这里的使用变得并不那么值。

而对于第二个问题，我们也有一个可视化的办法来观察硬件层更新。使用开发者选项中的“Show hardware layers updates”（显示硬件层更新）



当打开该选项后，View会在硬件层刷新的时候闪烁绿色。在很久以前我有一个ViewPager在滑动的时候有点不流畅。在开发者模式启动这个选项后，我再次滑动ViewPager，发现了如下情况：



左右两页在滑动的时候完全变成了绿色！

这意味着他们在创建的时候使用了硬件层，而且在滑动的时候也界面也进行了刷新。而当我在背景上面使用时差效果并且让条目有一个动画效果的时候，这些处理确实会让它进行刷新，但是我并没有对ViewPager启动硬件层。在阅读了ViewPager的源码后，我发现了在滑动的时候会自动为左右两页启动一个硬件层，并且在滑动结束后移除掉。

在两页间滑动的时候创建硬件层也是可以理解的，但对我来说小有不幸。通常来讲加入硬件层是为了让ViewPager的滑动更加流畅，毕竟它们相对复杂。但这不是我的app所想要的，我不得不通过[一些编码来移除硬件层](#)。

硬件层其实并不是什么酷炫的东西。重要的是我们要理解他的原理并且合理的使用他们，要不然你确实会遇到一些麻烦。

## DIY

在准备上述这一系列例子的过程中，我进行了很多的编码去模拟这些情景。你可以在这个[Github项目](#)中找到这些代码，同时也可以在[Google Play](#)中找到。我用不同的Activity区分了不同的情景，并且尽量将他们的用文档解释清楚，以便于帮助大家理解不同的Activity中是出现哪种问题。大家可以边阅读各个Activity的javadoc的同时，利用我们前面讲到的工具去玩儿这个App。

## 更多信息

随着安卓系统的不断进化，你有话你的应用的手段也在不断变多。很多全新的工具被引入到了SDK中，以及一些新的特性被加入到了系统中（好比硬件层这东西）。所以与时俱进和懂得取舍是非常重要的。

这是一个非常棒的[油管播放列表](#)，叫Android Performance Patterns，一些谷歌出品的短视频，讲解了很多与性能相关的话题。你可以找到不同数据结构之间的对比（HashMap vs ArrayMap），Bitmap的优化，网络优化等等，吐血推荐！

加入[Android Performance Patterns的G+社群](#)，和大家一起讨论，分享心得，提出问题！

更多有意思的链接：

- 了解安卓中的[图形结构（Graphics Architecture）](#)。例如关于UI的渲染，不同的系统组件，比如SurfaceFlinger，以及他们之间是如何交互的。比较长，但是值得一看！ \* [Google IO 2012上的一段演讲](#)，展示了绘制模型（Drawing model）是如何工作的。
- 一段来自Devoxx 2013的关于Anrdroid性能的研讨，展示了一些在Anrdroid 4.4对绘制模型的一些优化，并且通过demo的形式展示了对不同优化工具的使用（Systrace, Overdraw等等）。
- 一篇非常好的关于“预防性优化”（Preventative Optimizations）的文章，阐述了他和“不成熟的优化”有和区别。很多的开发者并不优化他们的代码，因为他们认为这些影响并不明显。但是记住，问题也是积少成多的。如果你有机会去优化很小的一点，即便是非常微不足道的一点，也是应该的。
- [安卓中的内存管理](#) – 一个2011年的Google IO视频，仍然值得一看。视频展示了安卓是如何管理不同app的内存的，以及如何使用Eclipse MAT去发现问题。
- 一个叫做Romain Guy的谷歌工程师的[案例研究](#)，通过优化一个第三方的推特客户端。在这个研究中，Romain展示了他是如何发现问题的，并且建议了相应的解决方

案。[另一篇文章跟进了这个问题](#)，展示了这个app在重新制作后的一些其他问题。

我真心希望你通过这篇文章获得到了足够丰富的信息和信心，从今天开始优化你的应用吧！

尝试用工具去记录，并通过一些开发者选项中的选项，开搞吧。欢迎来G+上分享你在安卓性能优化上面的心得！

# AndroidStudio

# 使用新版 Android Studio 检测内存泄露和性能

来源:<http://gold.xitu.io/entry/5732900179bc44005c248910>

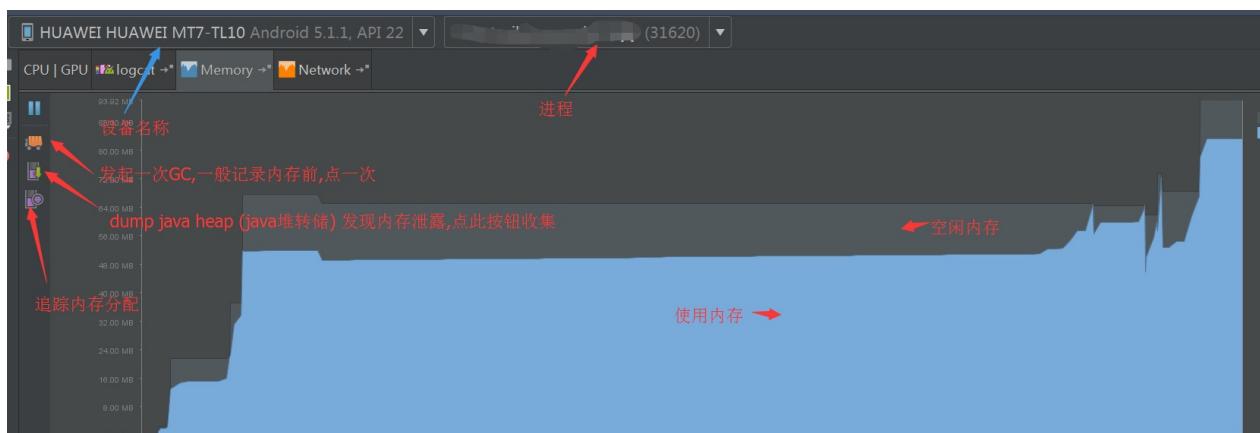
**NeXT\_ 推荐:** Android 工程师 @ 稀土 内存泄露，是 Android 开发者最头疼的事。可能一处小小的内存泄露，都可能是毁于千里之堤的蚁穴。怎么才能检测内存泄露呢？网上教程非常多，不过很多都是使用 Eclipse 检测的，其实 1.3 版本以后的 Android Studio 检测内存非常方便，如果结合上 MAT 工具，LeakCanary 插件，一切就变得 so easy 了。作者：@于连林 520wcf  
原文链接：<http://www.jianshu.com/p/216b03c22bb8>

内存泄露，是Android开发者最头疼的事。可能一处小小的内存泄露，都可能是毁于千里之堤的蚁穴。

怎么才能检测内存泄露呢？网上教程非常多，不过很多都是使用Eclipse检测的，其实1.3版本以后的Android Studio 检测内存非常方便，如果结合上MAT工具,LeakCanary插件,一切就变得so easy了。

## 熟悉Android Studio界面

工欲善其事,必先利其器。我们接下来先来熟悉下Android Studio的界面



一般分析内存泄露，首先运行程序，打开日志控制台，有一个标签Memory，我们可以在这个界面分析当前程序使用的内存情况，一目了然，我们再也不需要苦苦的在logcat中寻找内存的日志了。

图中蓝色区域，就是程序使用的内存，灰色区域就是空闲内存

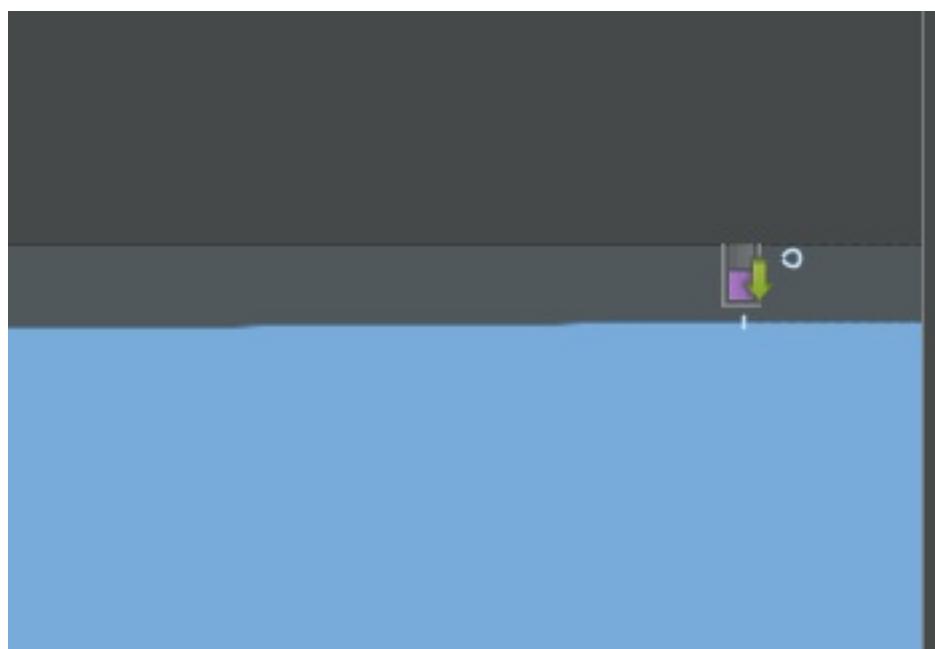
当然, Android内存分配机制是对每个应用程序逐步增加, 比如你程序当前使用30M内存, 系统可能会给你分配40M, 当前就有10M空闲, 如果程序使用了50M了, 系统会紧接着给当前程序增加一部分, 比如达到了80M, 当前你的空闲内存就是30M了。当然, 系统如果不能再给你分配额外的内存, 程序自然就会OOM(内存溢出)了。每个应用程序最高可以申请的内存和手机密切相关, 比如我当前使用的华为Mate7, 极限大概是200M, 算比较高的了, 一般128M就是极限了, 甚至有的手机只有可怜的16M或者32M, 这样的手机相对于内存溢出的概率非常大了。

## 我们怎么检测内存泄露呢

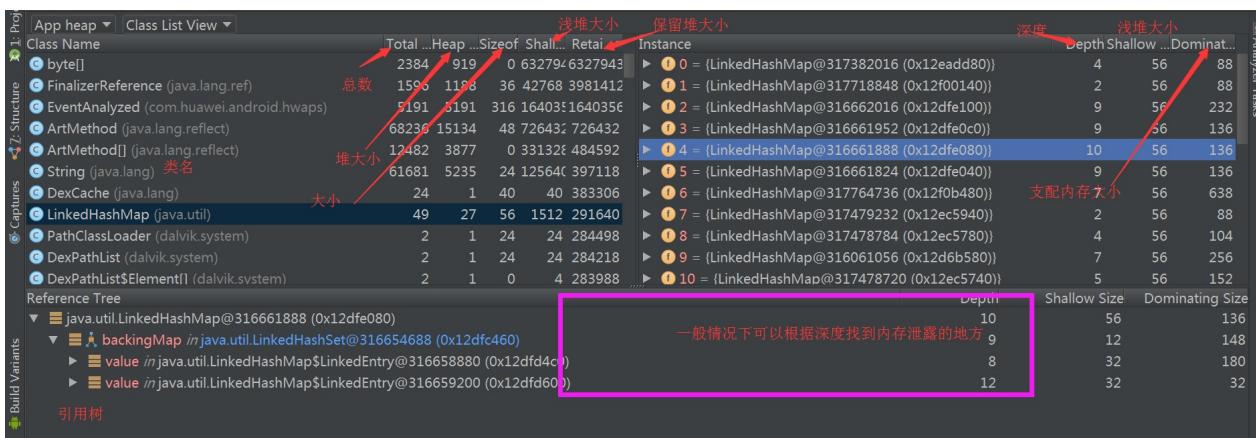
首先需要明白一个概念, 内存泄露就是指, 本应该回收的内存, 还驻留在内存中。

一般情况下, 高密度的手机, 一个页面大概就会消耗20M内存, 如果发现退出界面, 程序内存迟迟不降低的话, 可能就发生了严重的内存泄露。

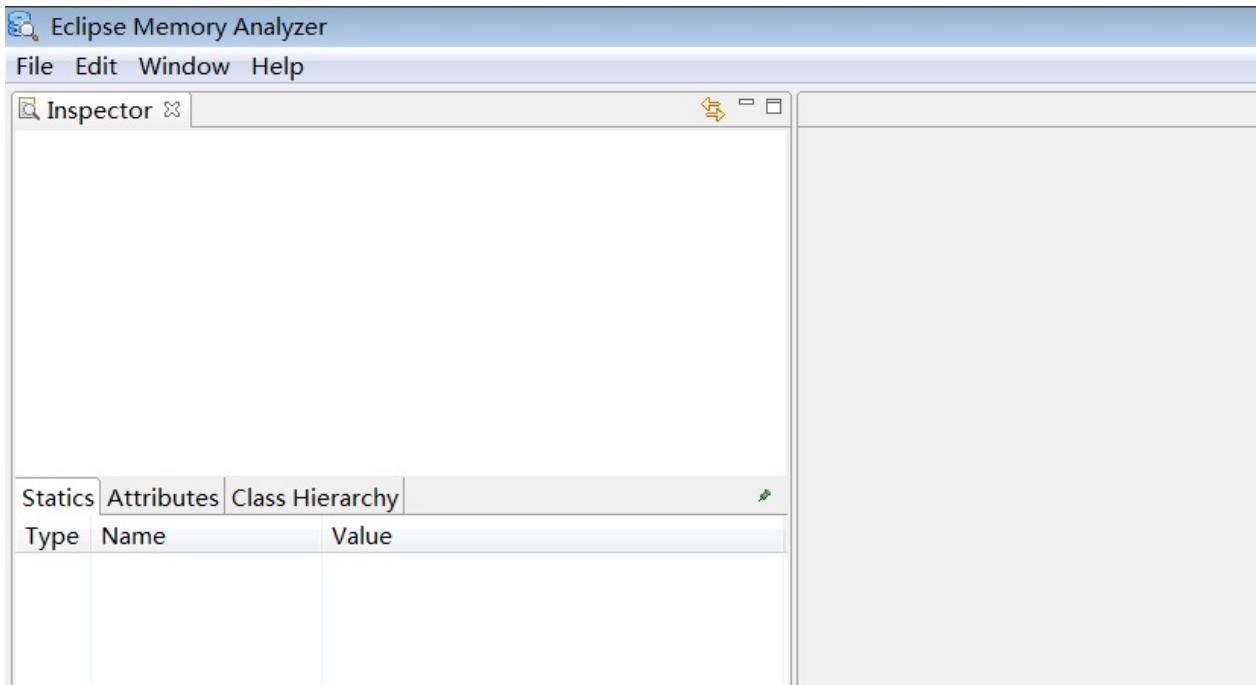
我们可以反复进入该界面, 然后点击dump Java heap这个按钮, 然后Android Studio就开始干活了, 下面的图就是正在dump



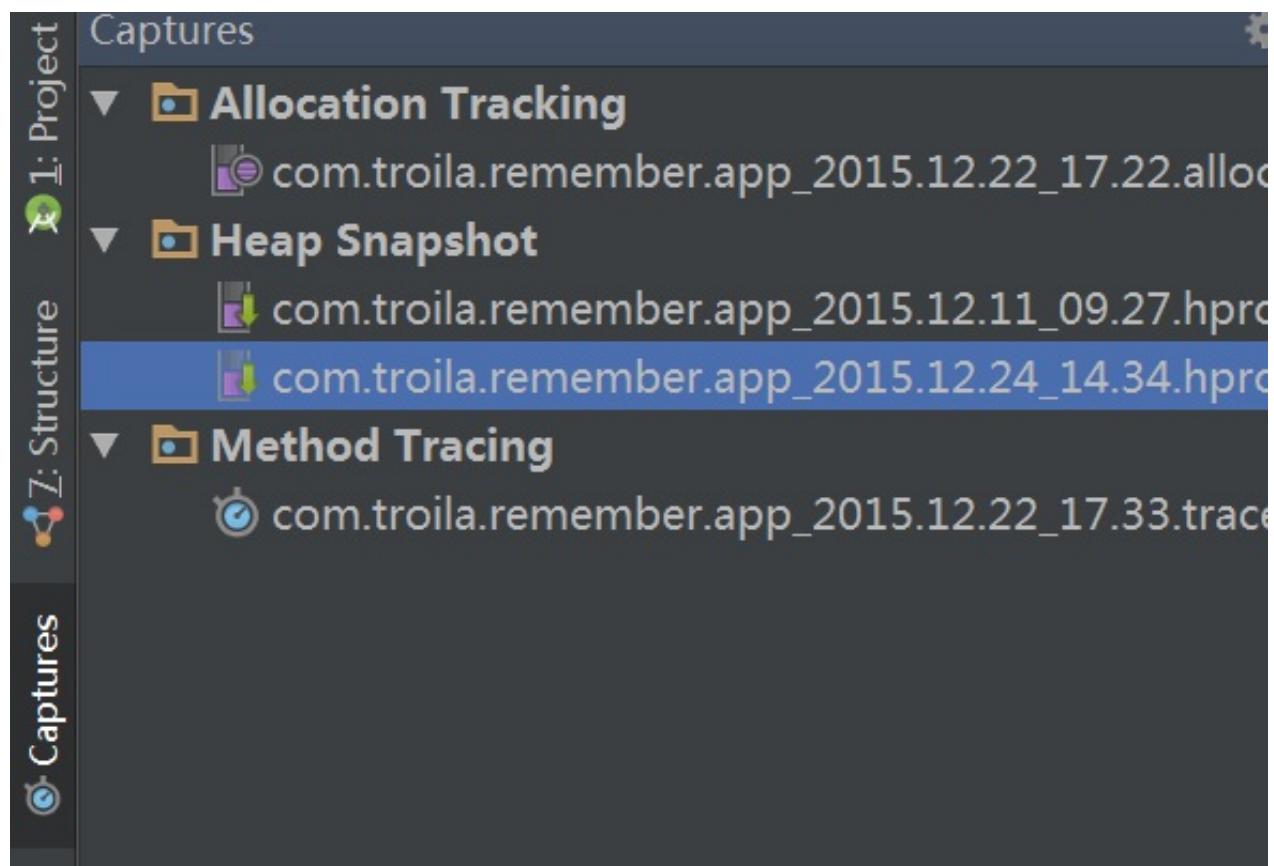
dump成功后会自动打开 hprof文件, 文件以Snapshot+时间来命名



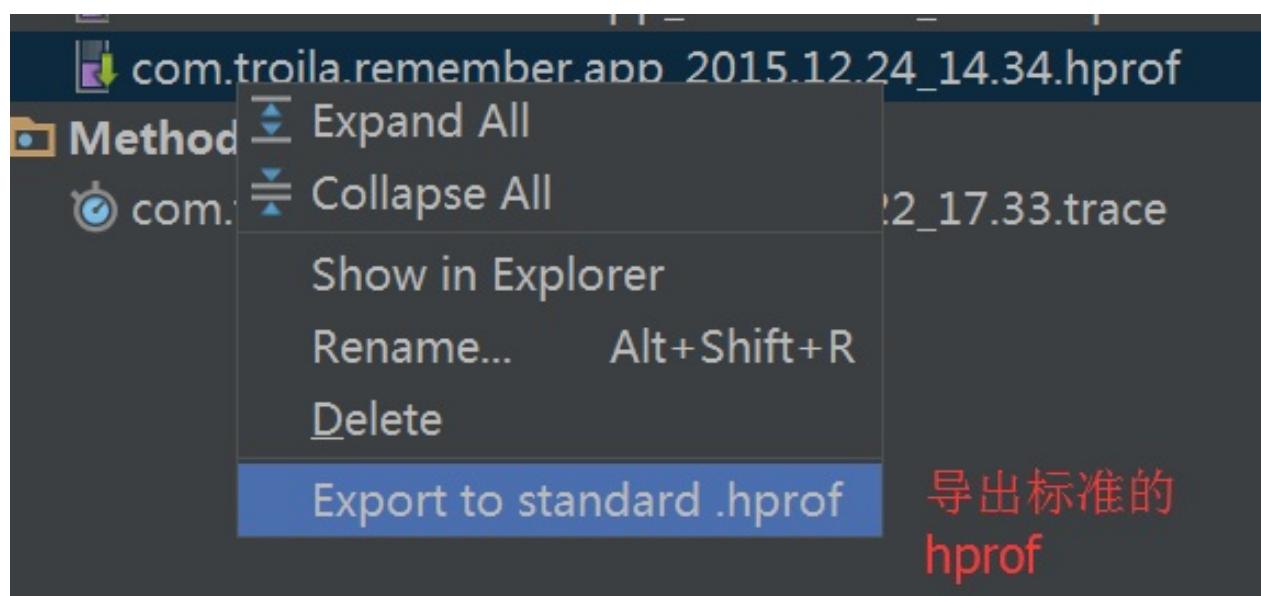
通过Android Studio自带的界面,查看内存泄露还不是很智能,我们可以借助第三方工具,常见的工具就是MAT了,下载地址 <http://eclipse.org/mat/downloads.php>,这里我们需要下载独立版的MAT. 下图是MAT一开始打开的界面, 这里需要提醒大家的是, MAT并不会准确地告诉我们哪里发生了内存泄漏, 而是会提供一大堆的数据和线索, 我们需要自己去分析这些数据来去判断到底是不是真的发生了内存泄漏。



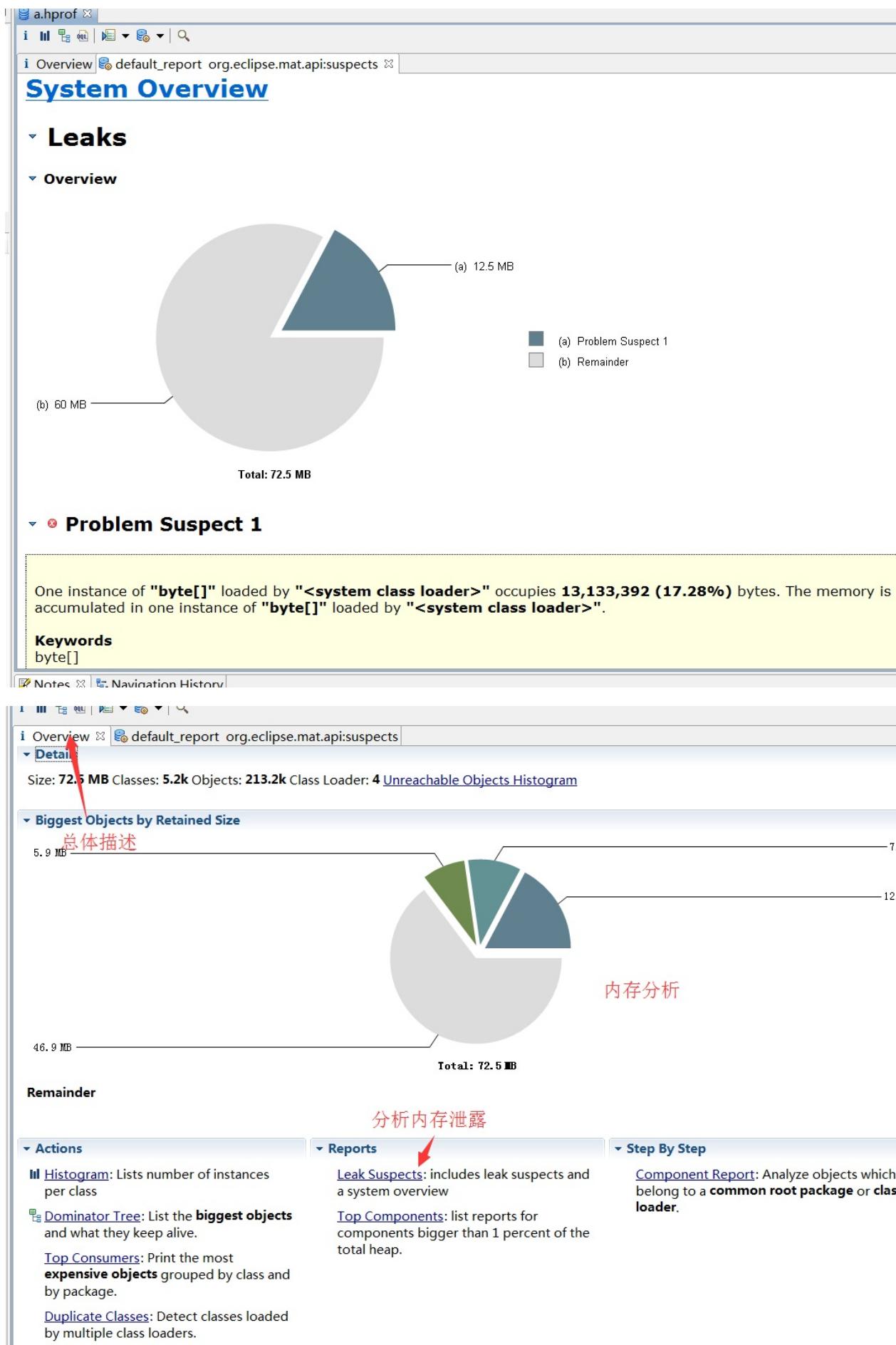
接下来我们需要用MAT打开内存分析的文件, 上文给大家介绍了使用Android Studio生成了hprof文件, 这个文件在呢, 在Android Studio中的Captures这个目录中,可以找到



注意,这个文件不能直接交给MAT, MAT是不识别的,我们需要右键点击这个文件,转换成MAT识别的。



然后用MAT打开导出的hprof(File->Open heap dump) MAT会帮我们分析内存泄露的原因

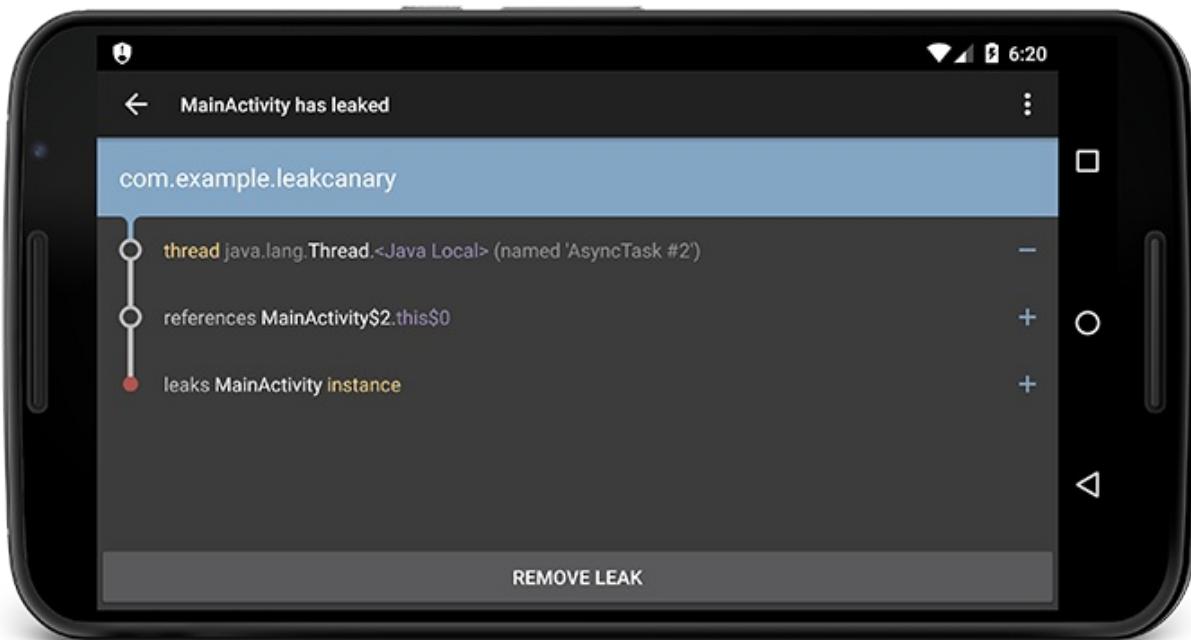


# LeakCanary

上面介绍了MAT检测内存泄露, 再给大家介绍LeakCanary。

项目地址: <https://github.com/square/leakcanary>

LeakCanary会检测应用的内存回收情况, 如果发现有垃圾对象没有被回收, 就会去分析当前的内存快照, 也就是上边MAT用到的.hprof文件, 找到对象的引用链, 并显示在页面上。这款插件的好处就是,可以在手机端直接查看内存泄露的地方,可以辅助我们检测内存泄露



使用:

在build.gradle文件中添加, 不同的编译使用不同的引用:

```
dependencies {
 debugCompile 'com.squareup.leakcanary:leakcanary-android:1.3'
 releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.3'
}
```

在应用的Application onCreate方法中添加LeakCanary.install(this), 如下

```
public class ExampleApplication extends Application
 @Override
 public void onCreate() {
 super.onCreate();
 LeakCanary.install(this);
 }
}
```

应用运行起来后，LeakCanary会自动去分析当前的内存状态，如果检测到泄漏会发送到通知栏，点击通知栏就可以跳转到具体的泄漏分析页面。

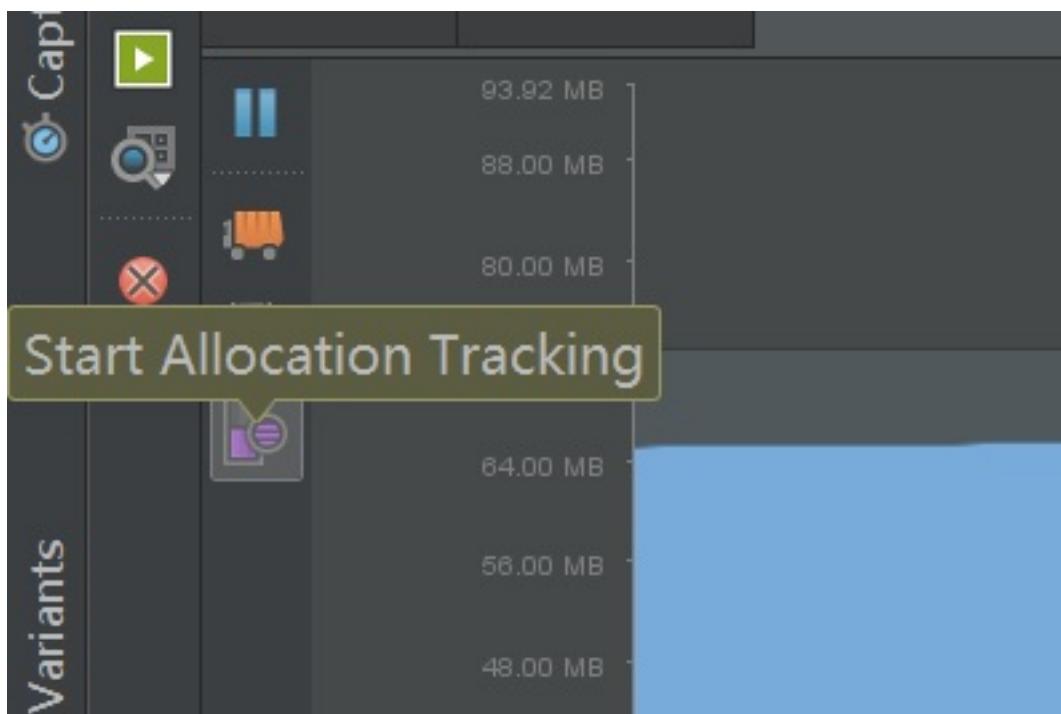
Tips：就目前使用的结果来看，绝大部分泄漏是由于使用单例模式hold住了Activity的引用，比如传入了context或者将Activity作为listener设置了进去，所以在使用单例模式的时候要特别注意，还有在Activity生命周期结束的时候将一些自定义监听器的Activity引用置空。

关于LeakCanary的更多分析可以看项目主页的介绍，还有这里  
<http://www.liaohuqiu.net/cn/posts/leak-canary-read-me/>

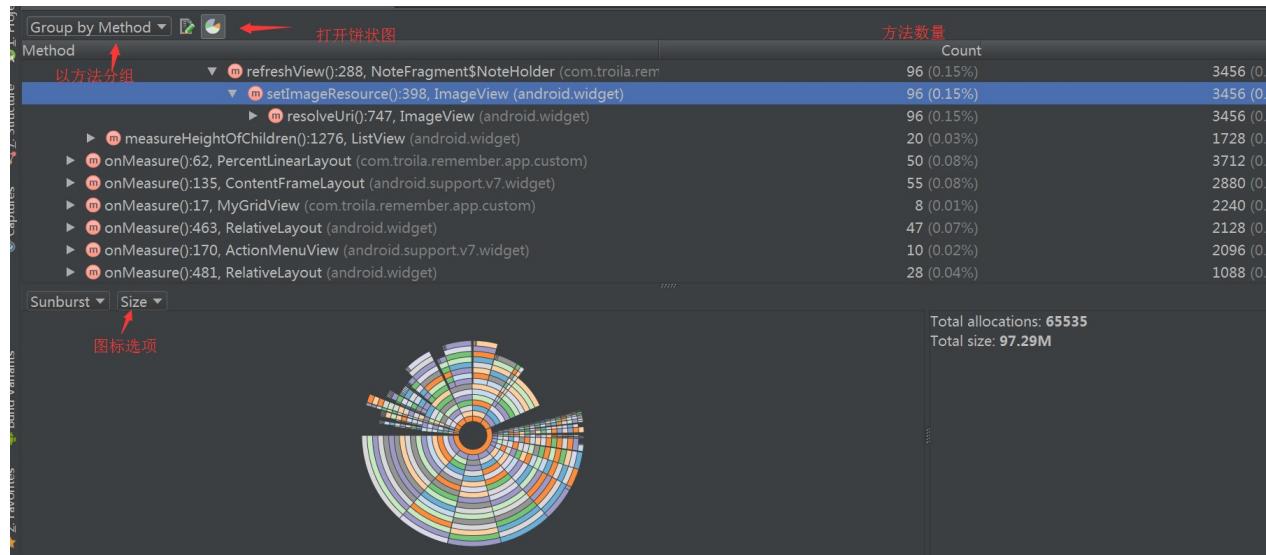
## 追踪内存分配

如果我们想了解内存分配更详细的情况，可以使用Allocation Traker来查看内存到底被什么占用了。

用法很简单：



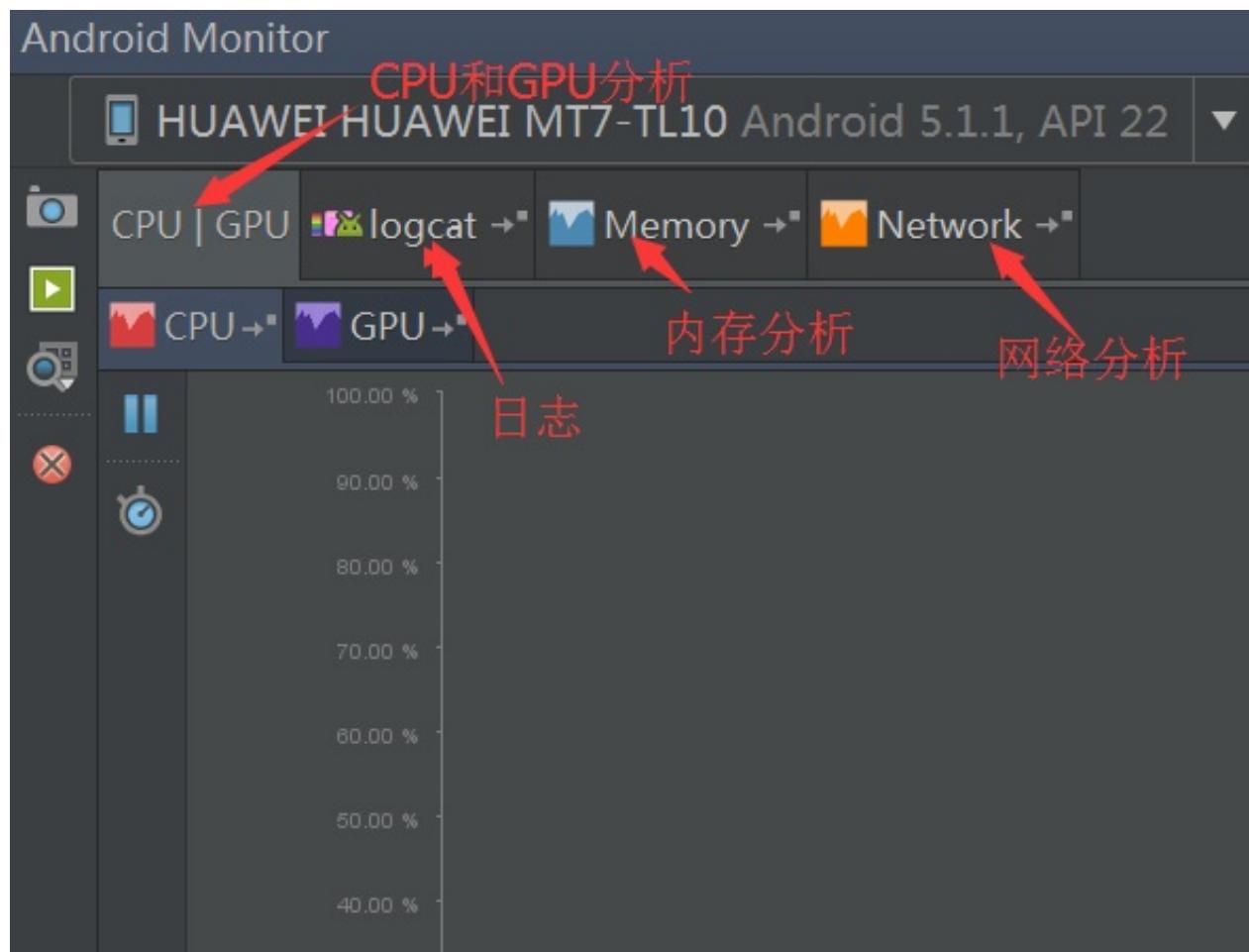
点一下是追踪，再点一下是停止追踪，停止追踪后.alloc文件会自动打开,打开后界面如下：



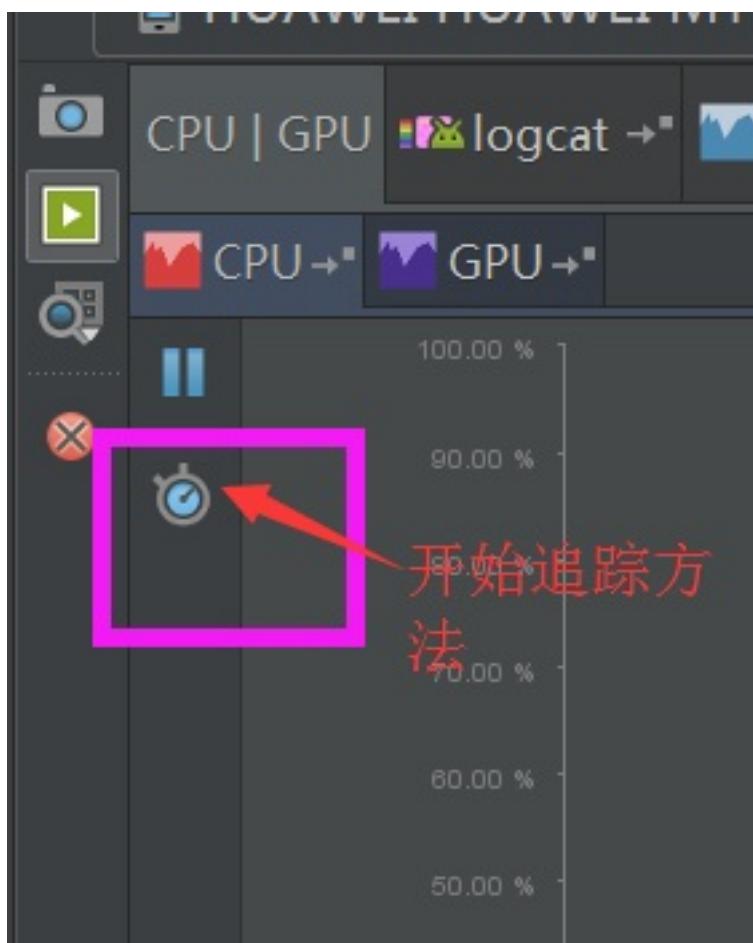
当你想查看某个方法的源码时,右键选择的方法,点击 [Jump to source](#) 就可以了

## 查询方法执行的时间

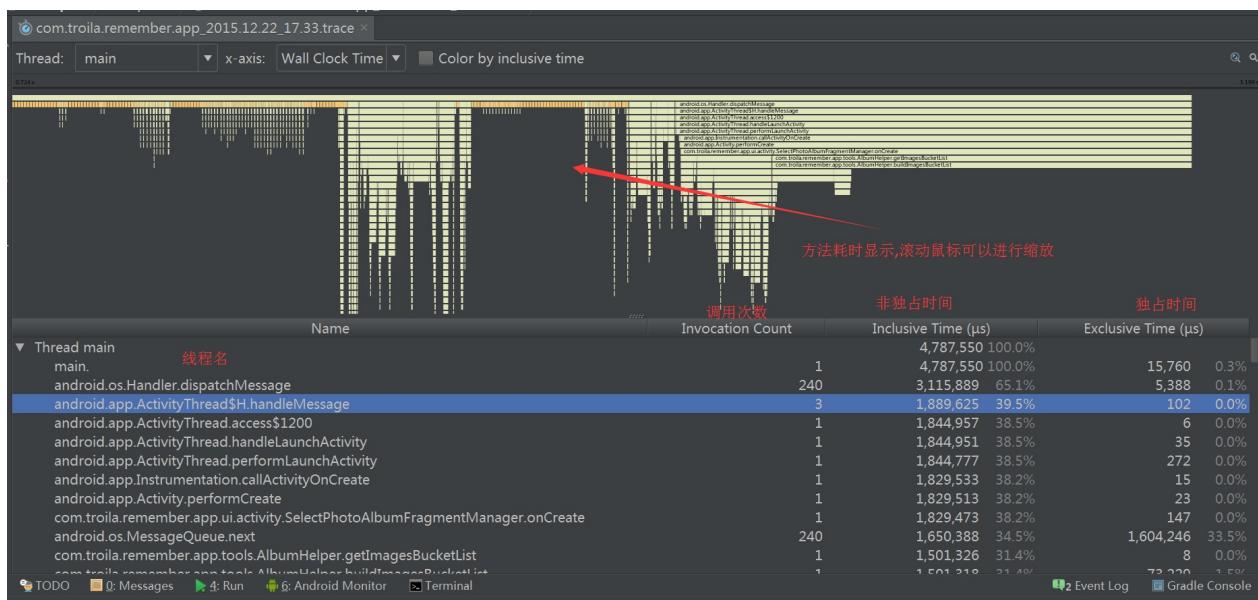
Android Studio 功能越来越强大了, 我们可以借助AS观测各种性能,如下图:



如果我们要观测方法执行的时间,就需要来到CPU界面



点击Start Method Tracking, 一段时间后再点击一次, trace文件被自动打开,



**非独占时间:** 某函数占用的CPU时间,包含内部调用其它函数的CPU时间。

**独占时间:**某函数占用CPU时间,但不含内部调用其它函数所占用的CPU时间。

**我们如何判断可能有问题的方法?**

通过方法的调用次数和独占时间来查看, 通常判断方法是:

- 如果方法调用次数不多，但每次调用却需要花费很长的时间的函数，可能会有问题。
- 如果自身占用时间不长，但调用却非常频繁的函数也可能有问题。

## 综述

上面给大家介绍了若干使用Android Studio检查程序性能的工具,工具永远是辅助,不要因为工具耽误太长时间。如果有问题，欢迎大家纠正。

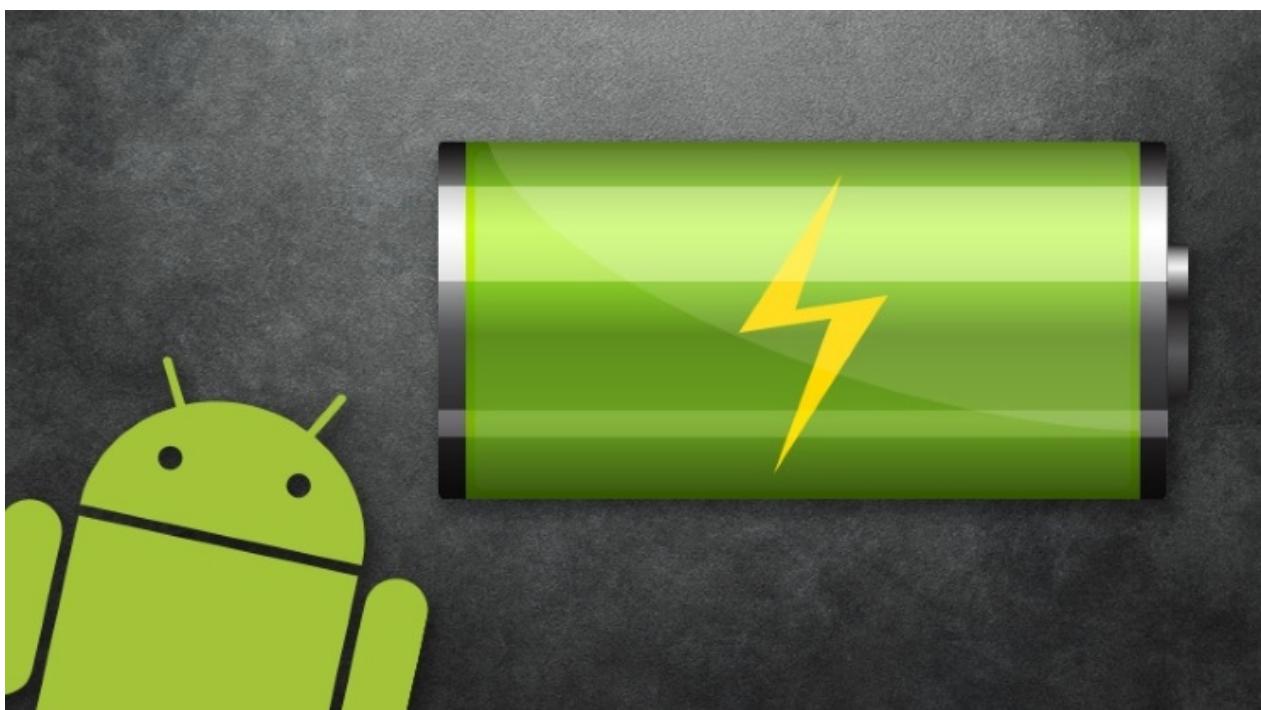
# BatteryHistorian

# 使用Battery Historian分析应用的电量

来源:[简书](#)

欢迎Follow我的[GitHub](#), 关注我的[简书](#). 其余参考[Android目录](#).

在Android项目中, 较难监控应用的电量消耗, 但是用户却非常关心手机的待机时间. 过度耗电的应用, 会遭到用户无情的卸载, 不要存在侥幸心理, 给竞品带来机会. 因此, 我们需要研究应用的耗电量, 并进行优化. 本文讲解一下[Battery Historian](#), 是一款由Google提供的Android系统电量分析工具. 在网页中展示手机的电量消耗过程, 输入电量分析文件, 显示消耗情况. 最后提供一些电量优化的方法, 可供参考.



## 1. 安装Go

Battery Historian是Go语言开发, 需要安装Go编译环境.

下载Mac版的[安装包](#), 执行完成, 检查Go版本

```
→ ~ go version
go version go1.6 darwin/amd64
```

在 `.bash_profile` 中, 设置Go语言变量

```
#Go Settings
export GOPATH=/Users/.../Workspace/GoWorkspace
export GOBIN=/Users/.../Workspace/GoWorkspace/bin
```

GOPATH源码地址. GOBIN生成地址, 推荐 \$GOPATH/bin .

执行 source .bash\_profile , 应用profile配置.

新建src文件夹, 添加 HelloWorld 文件 hello.go .

```
package main

import "fmt"

func main() {
 fmt.Printf("hello, world\n")
}
```

安装hello.go

```
go install hello.go
```

执行

```
$GOBIN/hello
```

如果显示hello, world, 即表示安装完成.

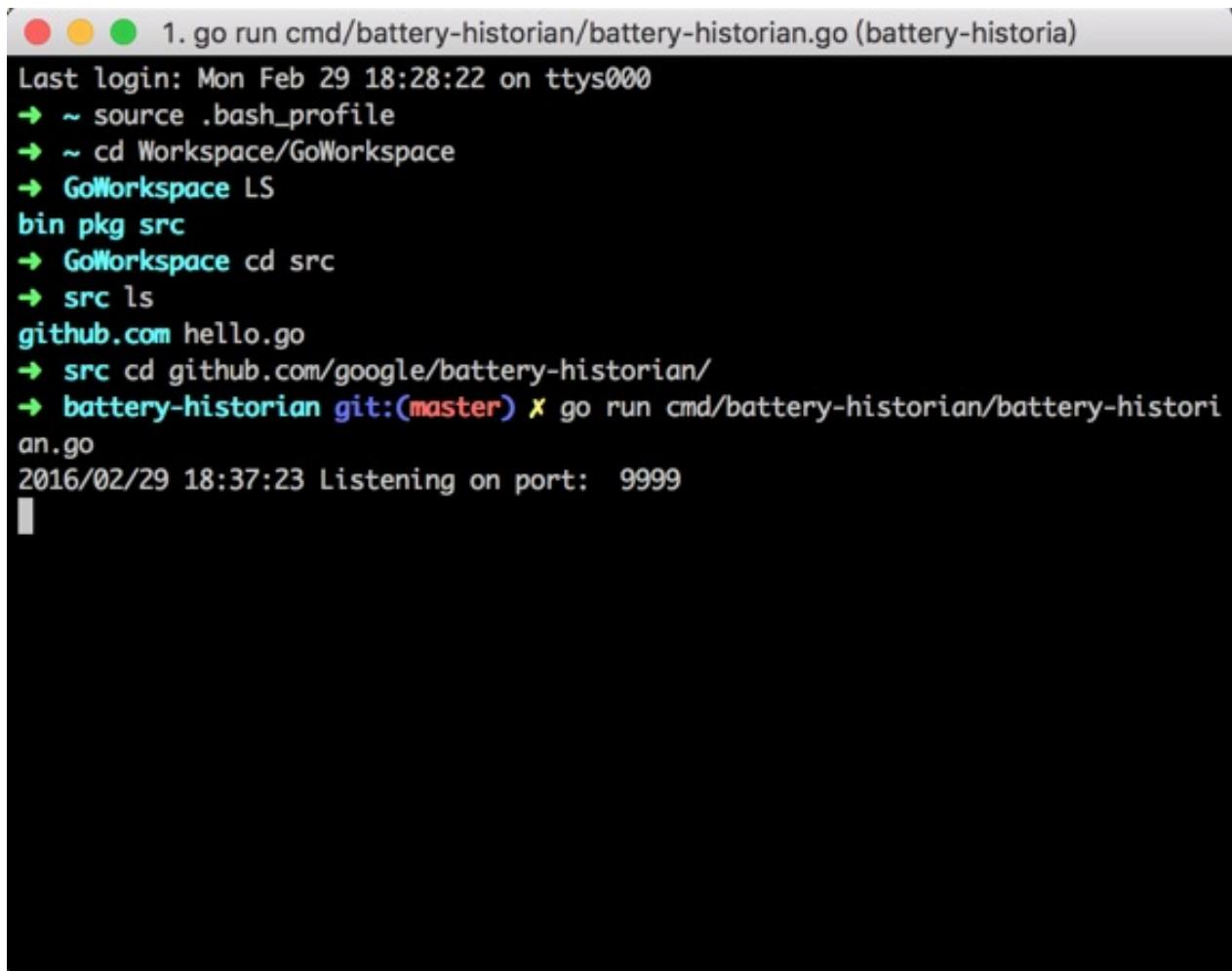
## 2. Battery Historian

在安装Battery Historian时, 需要提前安装wget.

```
sudo brew install wget
```

按照Battery Historian的[GitHub](#)文档执行操作即可.

```
go get -u github.com/golang/protobuf/proto
go get -u github.com/golang/protobuf/protoc-gen-go
go get -u github.com/google/battery-historian/...
cd $GOPATH/src/github.com/google/battery-historian
bash setup.sh
运行Go脚本， 默认端口9999
go run cmd/battery-historian/battery-historian.go [--port <default:9999>]
```



The screenshot shows a terminal window with a light gray header bar containing three colored circles (red, yellow, green) and the text "1. go run cmd/battery-historian/battery-historian.go (battery-historian)". Below the header, the terminal displays a series of command-line steps:

```
Last login: Mon Feb 29 18:28:22 on ttys000
→ ~ source .bash_profile
→ ~ cd Workspace/GoWorkspace
→ GoWorkspace LS
bin pkg src
→ GoWorkspace cd src
→ src ls
github.com hello.go
→ src cd github.com/google/battery-historian/
→ battery-historian git:(master) ✘ go run cmd/battery-historian/battery-historian.go
2016/02/29 18:37:23 Listening on port: 9999
```

## 启动命令

```
cd $GOPATH/src/github.com/google/battery-historian
go run cmd/battery-historian/battery-historian.go [--port <default:9999>]
```

在浏览器中，输入<http://localhost:9999/>，即可启动电量检测页面。

## Battery Historian 2.0

Upload Bugreport

未选择任何文件

在启动页面时, 可能需要连接VPN, 访问Google信息.

### 3. 分析

获取手机的电量文件, 导出到根目录, 以备**Battery Historian**使用.

```
adb bugreport > bugreport.txt
```

使用**Battery Historian**的网页加载**bugreport.txt**文件.

如遇到一些问题, 重新提交**bugreport.txt**文件.

系统状态

## Battery Historian 2.0

File: bugreport.txt  
 Device: Redmi Note 3  
 Build: Xiaomi/hennessy/hennessy:5.0.2/LRX22G/V7.1.8.0.LHNCNCK:user/release-keys

Analyze a new bugreport.  
[Warnings](#) [Show](#)

| System Stats                        | Historian 2.0 | Historian (legacy) | App Stats | Choose an application |
|-------------------------------------|---------------|--------------------|-----------|-----------------------|
| <a href="#">Clear app selection</a> |               |                    |           | ▼                     |

## Redmi Note 3 LRX22G

## Aggregated Stats:

| Metric                           | Value                  |
|----------------------------------|------------------------|
| Device                           | Redmi Note 3           |
| Build                            | LRX22G                 |
| Duration / Realtime              | 4h0m1.39s              |
| Screen Off Discharge Rate (%/hr) | 2.88 (Discharged: 10%) |
| Screen On Discharge Rate (%/hr)  | 13.37 (Discharged: 7%) |
| Screen On Time                   | 31m25.181s             |
| Screen Off Uptime                | 2h6m5.878s             |
| Userspace Wakelock Time          | 2h0m25.49s             |
| Kernel Overhead Time             | 5m40.388s              |
| Mobile KBs/hr                    | 2956.83                |
| WiFi KBs/hr                      | 2586.90                |
| Mobile Active Time               | 40.135s                |
| Signal Scanning Time             | 0                      |

## Top power consuming entities:

| Ranking | Name           | Uid  | Battery Percentage Consumed |
|---------|----------------|------|-----------------------------|
| 0       | UNACCOUNTED    | 0    | 8.79%                       |
| 1       | SCREEN         | 0    | 3.68%                       |
| 2       | CELL           | 0    | 0.40%                       |
| 3       | ANDROID_SYSTEM | 1000 | 0.34%                       |
| 4       | ROOT           | 0    | 0.34%                       |

## Userspace partial wakelocks:

| Ranking | Name                                 | Uid   | Duration | Count |
|---------|--------------------------------------|-------|----------|-------|
| 2       | com.tencent.mm : WakerLock:971815360 | 10108 | 2m7.241s | 12    |
| 3       | com.tencent.mm : WakerLock:881587341 | 10108 | 26s      | 5     |

## 应用状态(简书)

## Battery Historian 2.0

File: bugreport.txt  
 Device: Redmi Note 3  
 Build: Xiaomi/hennessy/hennessy:5.0.2/LRX22G/V7.1.8.0.LHNCNCK:user/release-keys

Analyze a new bugreport.  
[Warnings](#) [Show](#)

| System Stats                                       | Historian 2.0 | Historian (legacy) | App Stats | com.jianshu.haruki (Uid: 10759) | ▼                         |
|----------------------------------------------------|---------------|--------------------|-----------|---------------------------------|---------------------------|
| <a href="#">Clear app selection</a>                |               |                    |           |                                 | ▼                         |
| Application                                        |               |                    |           |                                 | com.jianshu.haruki        |
| Version Code                                       |               |                    |           |                                 | 16012901                  |
| UID                                                |               |                    |           |                                 | 10759                     |
| Computed power drain                               |               |                    |           |                                 | 0.10 %                    |
| Foreground                                         |               |                    |           |                                 | 21 times over 6m 9s 767ms |
| <b>Network information:</b>                        |               |                    |           |                                 |                           |
| Mobile KB transferred                              |               |                    |           |                                 |                           |
| 341.67 total (253.36 received, 88.31 transmitted)  |               |                    |           |                                 |                           |
| Wifi KB transferred                                |               |                    |           |                                 |                           |
| 372.22 total (268.61 received, 103.61 transmitted) |               |                    |           |                                 |                           |
| Mobile packets transferred                         |               |                    |           |                                 |                           |
| 1005 total (460 received, 545 transmitted)         |               |                    |           |                                 |                           |
| Wifi packets transferred                           |               |                    |           |                                 |                           |
| 901 total (386 received, 515 transmitted)          |               |                    |           |                                 |                           |
| Mobile active time                                 |               |                    |           |                                 |                           |
| 213.10ms                                           |               |                    |           |                                 |                           |
| Mobile active count                                |               |                    |           |                                 |                           |
| 2                                                  |               |                    |           |                                 |                           |
| <b>Wakelocks:</b>                                  |               |                    |           |                                 |                           |
| <b>Wakelock Name</b>                               |               |                    |           |                                 |                           |
| Full Time                                          |               |                    |           |                                 |                           |
| WindowManager                                      |               |                    |           |                                 |                           |
| 14s 527ms                                          |               |                    |           |                                 |                           |
| Full Count                                         |               |                    |           |                                 |                           |
| *alarm*                                            |               |                    |           |                                 |                           |
| 7                                                  |               |                    |           |                                 |                           |
| Partial Time                                       |               |                    |           |                                 |                           |
| 0ms                                                |               |                    |           |                                 |                           |
| Partial Count                                      |               |                    |           |                                 |                           |
| 48ms                                               |               |                    |           |                                 |                           |
| Window Time                                        |               |                    |           |                                 |                           |
| 0ms                                                |               |                    |           |                                 |                           |
| <b>Services:</b>                                   |               |                    |           |                                 |                           |
| <b>Processes:</b>                                  |               |                    |           |                                 |                           |
| <b>Sensor use:</b>                                 |               |                    |           |                                 |                           |
| <a href="#">Show</a>                               |               |                    |           |                                 |                           |

## 4. 电量优化

根据**Battery Historian**的电量提示信息, 消耗电量包含

**唤醒锁\SyncManager同步管理器\音视频\流量.**

优化方式:

- (1) 检查全部**唤醒锁**, 是否存在冗余或者无用的位置.
- (2) 集中相关的**数据请求**, 统一发送; 精简数据, 减少无用数据的传输.
- (3) 分析和统计等**非重要操作**, 可以在电量充足或连接WIFI时进行, 参考[JobScheduler](#).
- (4) 精简冗余的服务(**Service**), 避免长时间执行耗电操作.
- (5) 注意定位信息的获取, 使用后及时关闭.

电量优化并不是很难, 但需要对业务非常熟悉, 了解一些耗电操作的使用情况, 及时优化. 只有给用户精致的体验, 用户才能更加喜欢我们的应用, 这就是服务型社会的本质.

That's all! Enjoy it!

# BlockCanary

# BlockCanary — 轻松找出Android App界面卡顿元凶

来源：[markzhai's home](#)

BlockCanary是我利用个人时间开发的Android平台上的一个轻量的，非侵入式的性能监控组件，应用只需要简单地加几行，提供一些该组件需要的上下文环境就可以在使用应用的时候检测主线程上的各种卡顿问题，并通过组件提供的各种信息分析出原因并进行修复。

开源代码：[moduth/blockcanary](#)

## 背景

在复杂的项目环境中，由于历史代码庞大，业务复杂，包含各种第三方库，偶尔再来个jni调用，所以在出现了卡顿的时候，我们很难定位到底是哪里出现了问题，即便知道是哪一个Activity/Fragment，也仍然需要进去里面一行一行看，动辄数千行的类再加上跳来跳去调来调去的，结果就是不了了之随它去了，实在不行了再优化吧。于是一拖再拖，最后可能压根就改不动了，客户端越来越卡。

事实上，很多情况下卡顿不是必现的，它们可能与机型、环境、操作等有关，存在偶然性，即使发生了，再去查那如山般的logcat，也不一定能找到卡顿的原因，是我们自己的应用导致的还是其他应用抢占资源导致的？是哪些方法导致的？很难去回溯。有些机型自己修改了api导致的卡顿，还必须拿那台机器才能去调试找原因。

BlockCanary就是来解决这个问题的。告别打点，告别Debug，哪里卡顿，一目了然。

## 介绍

BlockCanary对主线程操作进行了完全透明的监控，并能输出有效的信息，帮助开发分析、定位到问题所在，迅速优化应用。其特点有：

- 非侵入式，简单的两行就打开监控，不需要到处打点，破坏代码优雅性。
- 精准，输出的信息可以帮助定位到问题所在（精确到行），不需要像Logcat一样，慢慢去找。

目前包括了核心监控输出文件，以及UI显示卡顿信息功能。仅支持Android端。

# 原理

熟悉Message/Looper/Handler系列的同学们一定知道Looper.java中这么一段：

```
private static Looper sMainLooper; // guarded by Looper.class
...

/***
 * Initialize the current thread as a looper, marking it as an
 * application's main looper. The main looper for your application
 * is created by the Android environment, so you should never need
 * to call this function yourself. See also: {@link #prepare()}
 */

public static void prepareMainLooper() {
 prepare(false);
 synchronized (Looper.class) {
 if (sMainLooper != null) {
 throw new IllegalStateException("The main Looper has already been prepared");
 }
 sMainLooper = myLooper();
 }
}

/***
 * Returns the application's main looper,
 * which lives in the main thread of the application.
 */
public static Looper getMainLooper() {
 synchronized (Looper.class) {
 return sMainLooper;
 }
}
```

即整个应用的主线程，只有这一个looper，不管有多少handler，最后都会回到这里。

如果再细心一点会发现在Looper的loop方法中有这么一段

```
public static void loop() {
 ...

 for (;;) {
 ...

 // This must be in a local variable, in case a UI event sets the logger
 Printer logging = me.mLogging;
 if (logging != null) {
 logging.println(">>>> Dispatching to " + msg.target + " " +
 msg.callback + ": " + msg.what);
 }

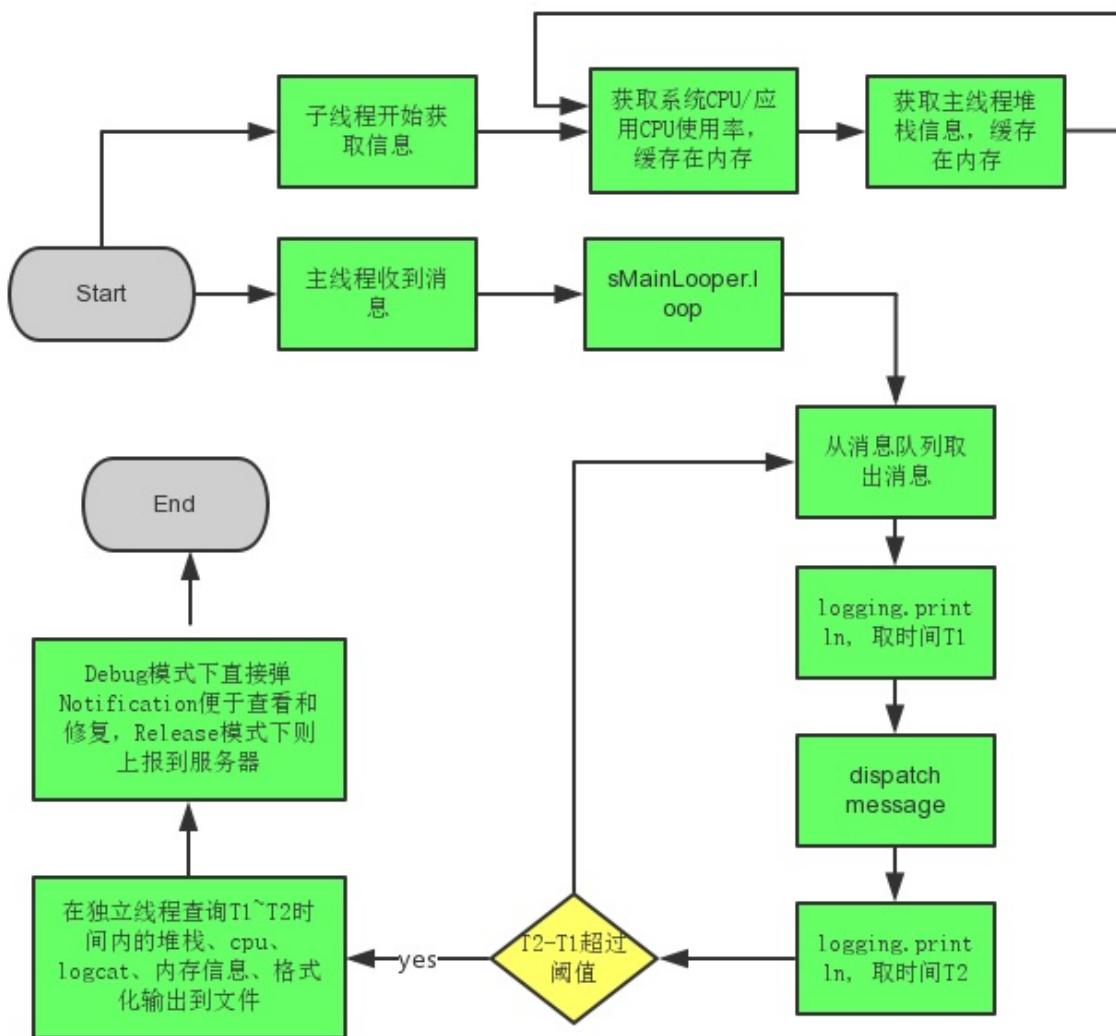
 msg.target.dispatchMessage(msg);

 if (logging != null) {
 logging.println("<<<< Finished to " + msg.target + " " + msg.callback);
 }

 ...
 }
}
```

是的，就是这个Printer - mLogging，它在每个message处理的前后被调用，而如果主线程卡住了，不就是在dispatchMessage里卡住了吗？

核心流程图：



该组件利用了主线程的消息队列处理机制，通过

```
Looper.getMainLooper().setMessageLogging(mainLooperPrinter);
```

并在 `mainLooperPrinter` 中判断 `start` 和 `end`，来获取主线程 `dispatch` 该 `message` 的开始和结束时间，并判定该时间超过阈值(如2000毫秒)为主线程卡慢发生，并dump出各种信息，提供开发者分析性能瓶颈。

```

...
@Override
public void println(String x) {
 if (!mStartedPrinting) {
 mStartTimeMillis = System.currentTimeMillis();
 mStartThreadTimeMillis = SystemClock.currentThreadTimeMillis();
 mStartedPrinting = true;
 } else {
 final long endTime = System.currentTimeMillis();
 mStartedPrinting = false;
 if (isBlock(endTime)) {
 notifyBlockEvent(endTime);
 }
 }
}

private boolean isBlock(long endTime) {
 return endTime - mStartTimeMillis > mBlockThresholdMillis;
}
...

```

说到此处，想到是不是可以用mainLooperPrinter来做更多事情呢？既然主线程都在这里，那只要parse出app包名的第一行，每次打印出来，是不是就不需要打点也能记录出用户操作路径？再者，比如想做onClick到页面创建后的耗时统计，是不是也能用这个原理呢？之后可以试试看这个思路（目前存在问题是获取线程堆栈是定时3秒取一次的，很可能一些比较快的方法操作一下子完成了没法在stacktrace里面反映出来）。

## 功能

BlockCanary会在发生卡顿（通过MonitorEnv的getConfigBlockThreshold设置）的时候记录各种信息，输出到配置目录下的文件，并弹出消息栏通知（可关闭）。

简单的使用如在开发、测试、Monkey的时候，Debug包启用

- 开发可以通过图形展示界面直接看信息，然后进行修复
- 测试可以把log丢给开发，也可以通过卡慢详情页右上角的更多按钮，分享到各种聊天软件（不要怀疑，就是抄的LeakCanary）
- Monkey生成一堆的log，找个专人慢慢过滤记录下重要的卡慢吧

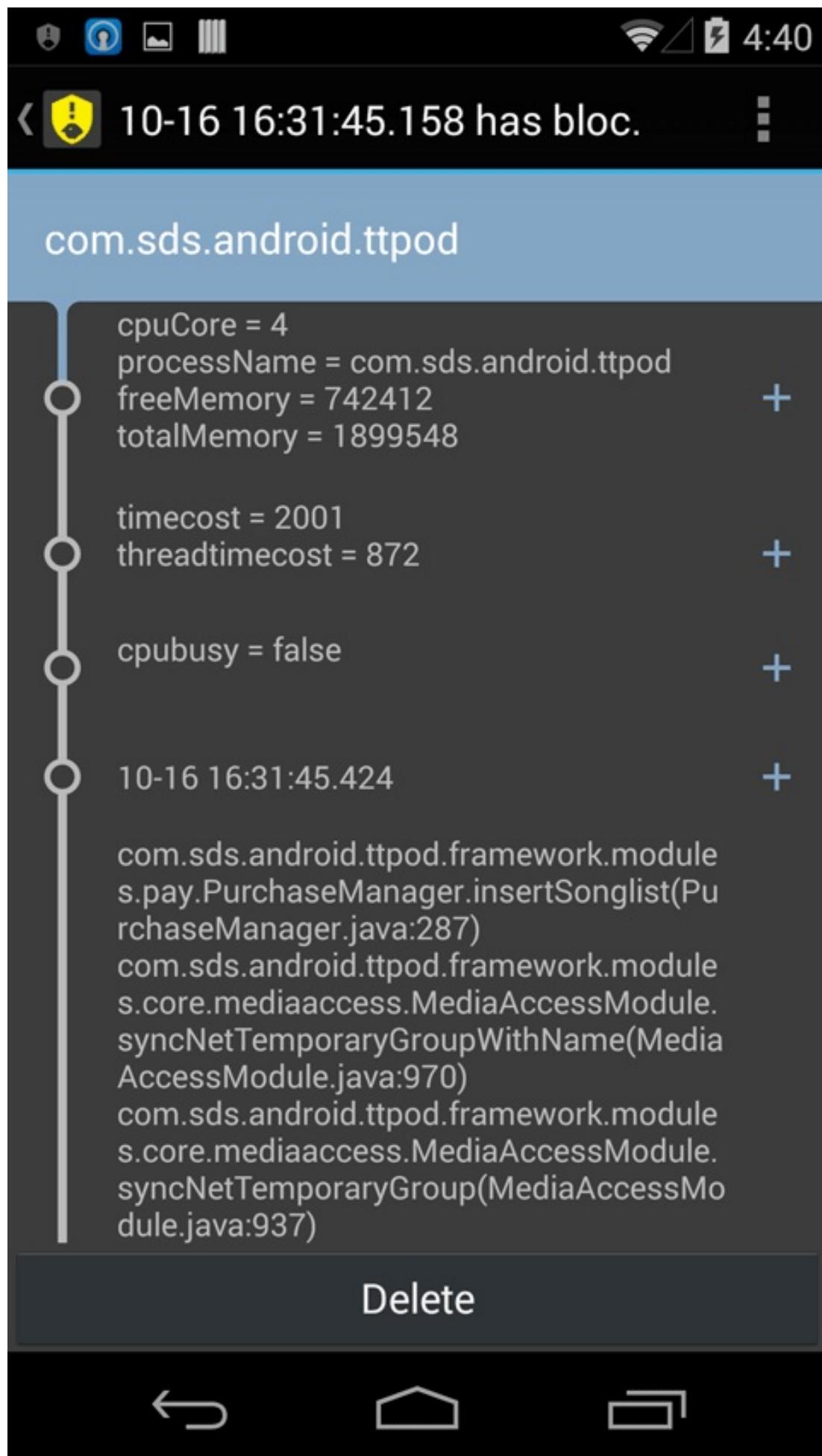
还可以通过Release包用户端定时开启监控并上报log，后台匹配堆栈过滤同类原因，提供给开发更大的样本环境来优化应用。

本项目提供了一个友好的展示界面，供开发测试直接查看卡慢信息（基于LeakCanary的界面修改）。

dump的信息包括：

- 基本信息：安装包标示、机型、api等级、uid、CPU内核数、进程名、内存、版本号等
- 耗时信息：实际耗时、主线程时钟耗时、卡顿开始时间和结束时间
- CPU信息：时间段内CPU是否忙，时间段内的系统CPU/应用CPU占比，I/O占CPU使用率
- 堆栈信息：发生卡慢前的最近堆栈，可以用来帮助定位卡慢发生的地方和重现路径

sample如下图，可以精确定位到代码中哪一个类的哪一行造成了卡慢。



## 总结

BlockCanary作为一个Android组件，目前还有局限性，因为其在一个完整的监控系统中只是一个生产者，还需要对应的消费者去分析日志，比如归类排序，以便看出哪些卡慢更有修复价值，需要优先处理；又比如需要过滤机型，有些奇葩机型的问题造成的卡慢，到底要不要去修复是要斟酌的。扯远一点的话，像是埋点除了统计外，完全还能用来做链路监控，比如一个完整的流程是A -> B -> D -> E, 但是某个时间节点突然A -> B -> D后没有到达E，这时候监控平台就可以发出预警，让开发人员及时定位。很多监控方案都需要C/S两端的配合。

目前阿里内多个Android项目接入并使用BlockCanary来优化Android应用的性能。

# gradle

# 更优雅的 Android 发布自动版本号方案

来源:[稀土掘金](#)

以前看到一些自动化版本号打包的文章。如果您的项目是用 Git 管理的，并且恰巧又是使用 Gradle 编译（应该绝大部分都是这样的了吧？），本文试图找到一种更加优雅的自动版本管理方法。

## 1 背景

我们都知道，Android 应用的版本管理是依赖 `AndroidManifest.xml` 中的两个属性：

- **android:versionCode**: 版本号，是一个大于 0 的整数，相当于 Build Number，随着版本的更新，这个必须是递增的。大的版本号，覆盖更新小的版本号；
- **android:versionName**: 版本名，是一个字符串，例如 "1.2.0"，这个是给人看的版本名，系统并不关心这个值，但是合理的版本名，对后期的维护和 bug 修复也非常重要。

在使用了 Android Studio 或者 Gradle 编译以后，我们通常是在 `build.gradle` 里面定义这两个值，如下：

```
android {
 ...
 defaultConfig {
 ...
 versionCode 1
 versionName "1.0"
 }
}
```

## 2 自动版本号

在这篇文章中[6 tips to speed up your Gradle build](#) 发现了，可以使用 Git 中 commit 的数量来作为版本号（`versionCode`）。方案如下：

```

def cmd = 'git rev-list HEAD --first-parent --count'
def gitVersion = cmd.execute().text.trim().toInteger()

android {
 defaultConfig {
 versionCode gitVersion
 }
}

```

这里关键是这一行 git 命令 `git rev-list HEAD --first-parent --count`，表示获取当前分支的 commit 数量。

这是一个绝妙的方案。因为在项目开发中，我们的往 Git 库中提交的 Commit 的数量应该是只增不减的（当然，在极少的情况下有例外），而且对应 Commit 的数量直接对应代码当前的版本状态，只要你做了代码修改，版本号就应该增加。有些解决方案中，每次 Build 就会增加一次版本号，个人感觉并不合适，如果是相同的代码，发布出去版本号应该保持一致，而不在乎你编译多少次。

另外，有些人可能会担心，每次版本发布，可能会包含几百个新的 commit，这样的话 `versionCode` 会不会增长太快了，最后导致不够用了。其实，完全没有必要担心，`versionCode` 是 int 类型，最大值是  $2^{31}-1$ ，也就是 21 亿多，Android 源码中，改动最活跃的 framework/base 所有分支到目前为止也就 20 万多个 commit，所以完全够用了。

### 3 自动版本名

前面通过一条简单的命令实现了自动化的 `versionCode`，现在我们看怎么自动化 `versionName`。

在正常的发布流程中，在发布新版本的时候，都会在版本库中打 tag。一般情况下，tag 名就是版本名，而且也建议这么做，因为如果某个版本出现 bug，也可以正好 checkout 这个 tag 来查看代码。所以，现在的问题就是怎么自动获得 git 库中最新的最新 tag？原来，git 早就提供了命令 `git describe`，它的功能就是获取从当期 commit 到距离它最近的 tag 的描述。默认都是 annotated tag，如果要指所有的类型的 tag 的话，就加 `--tags` 参数。

此命令的详细介绍在这里：[git-describe](#)。举例一个简单的例子，假如你的当前代码状态如下：

```

--A--B-...-C-->
| |
v1.0 v1.1

```

执行 `git describe` 的结果是: `v1.1`, 如果是如下的情况:

```
--A--B-...-C--D-->
| |
v1.0 v1.1
```

执行 `git describe` 的结果是: `v1.1-1-gxxxxxx`, 其中 1 表示当前代码距离最近的 tag `v1.1` 一个 commit, 最新的 commit 的 id 是 `xxxxxx`。

可见, `describe` 命令很好的描述了当前的分支的版本状态, 我们可以直接使用这个它的输出作为版本号。在 `build.gradle` 中的使用如下:

```
def cmd = 'git describe --tags'
def version = cmd.execute().text.trim()

android {
 defaultConfig {
 versionName version
 }
}
```

这样就可以自动抽取 git 中的 tag 为版本名了。有些同学可能接受不了这样版本名字 `v1.1-1-gxxxxxx`, 这里也可以稍微做一些修改, 使版本号更好看, 如下:

```
def pattern = "-(\d+)-g"
def matcher = version =~ pattern

if (matcher) {
 version = version.substring(0, matcher.start()) + "." + matcher[0][1]
} else {
 version = version + ".0"
}
```

这样的话, 上面的版本名就变为了 `v1.0.0` 和 `v1.1.1` 了。

## 4 优化

前面的那篇文章中说了, 为了尽可能减少 gradle 脚本的运算, 提高开发速度, 我们可以把这样的自动版本的计算放到 `release` 编译中去。最后的写法如下:

```
def gitVersionCode() {
 def cmd = 'git rev-list HEAD --first-parent --count'
```

```
 cmd.execute().text.trim().toInteger()
 }

def gitVersionTag() {
 def cmd = 'git describe --tags'
 def version = cmd.execute().text.trim()

 def pattern = "-(\\d+)-g"
 def matcher = version =~ pattern

 if (matcher) {
 version = version.substring(0, matcher.start()) + "." + matcher[0][1]
 } else {
 version = version + ".0"
 }
}

return version
}

android {
 compileSdkVersion 23
 buildToolsVersion "23.0.2"

 defaultConfig {
 applicationId "com.race604.example"
 minSdkVersion 15
 targetSdkVersion 23
 versionCode 1
 versionName '1.0'
 }
 buildTypes {
 debug {
 // 为了不和 release 版本冲突
 applicationIdSuffix ".debug"
 }
 release {
 minifyEnabled false
 proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
 }
 }
}

applicationVariants.all { variant ->
 if (variant.buildType.name.equals('release')) {
 variant.mergedFlavor.versionCode = gitVersionCode()
 variant.mergedFlavor.versionName = gitVersionTag()
 }
}
```

至此，结合 git 和 gradle 我们就实现了自动版本号。

# Jenkins

# Jenkins+Gradle+checkstyle对Android工程源码进行静态代码分析

来源:测试蜗牛,一步一个脚印

## 环境说明

```
Gradle 2.6.
OS: windows server 2008
Jenkins 1.620
checkstyle 6.11.6
```

## 前提

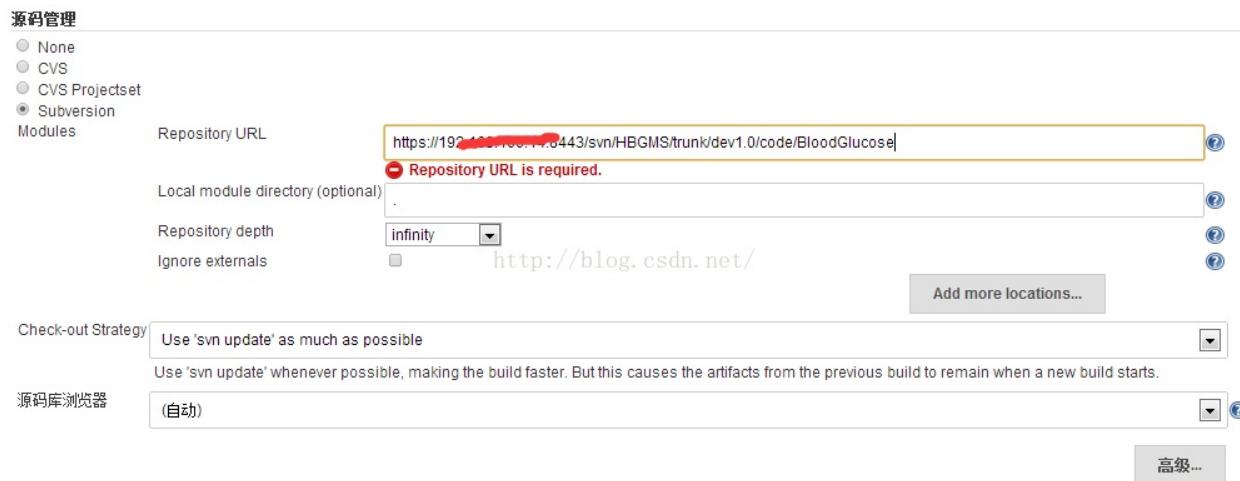
Jenkins需要提前安装好Checkstyle Plug-in插件

## 一、Jenkins配置如下:

- 1、新建job

The screenshot shows the Jenkins project configuration page for 'CodeCheck\_Style\_HBGMS\_BloodGlucose'. The 'Project Name' field is set to 'CodeCheck\_Style\_HBGMS\_BloodGlucose'. The 'Description' field contains a plain text link to 'http://blog.csdn.net/'. Under the 'Advanced Options' section, the 'Label Expression' is set to 'Master\_15' and 'Slaves in label' is set to '1'. The 'Build History' table shows four previous builds: #23 (2015-10-12下午5:10), #22 (2015-10-12下午5:09), #21 (2015-10-12下午5:08), and #20 (2015-10-12下午4:53). A '高级...' button is visible at the bottom right.

- 2、配置svn



## ● 3、配置构建操作

The screenshot shows the Jenkins build configuration under the 'Build' section. The 'Invoke Gradle script' section is expanded, showing the following fields:

- Gradle Version:** gradle2.6 [highlighted with a red box]
- Build step description:** [Empty field]
- Switches:** [Empty field]
- Tasks:** checkstyle --info [highlighted with a red box]
- Root Build script:** \${workspace}\BloodGlucose
- Build File:** build.gradle

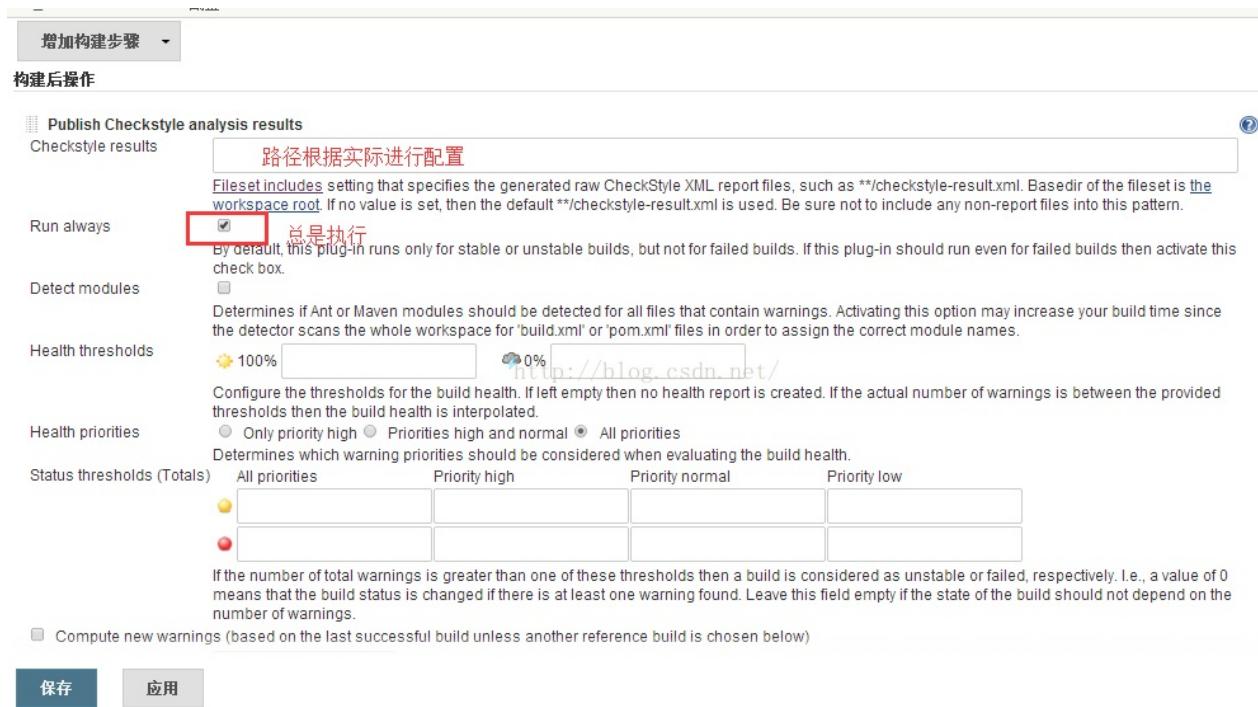
Below the build steps, there is a note: "Specify Gradle build file to run. Also, some environment variables are available to the build script". A 'Delete' button is also visible.

备注：

Tasks指的是build.gradle里面的task名称

配置info参数是用来查看调试日志，也可以配置debug级别。主要用来查看构建失败的原因。

## ● 4、配置分析报告



## 二、gradle.build的配置如下

- 1、添加checkstyle的依赖

```
buildscript {
 repositories {
 mavenCentral()
 }
 dependencies {
 classpath 'com.Android.tools.build:gradle:1.0.0+'
 //classpath 'io.fabric.tools:gradle:1.+'
 //classpath 'com.google.code.findbugs:findbugs:3.0.1'
 //classpath 'com.puppycrawl.tools:checkstyle:6.11.2'
 //classpath 'net.sourceforge.pmd:pmd:5.4.0'
 }
}
```

备注：版本包可以通过中央仓库 (<http://mvnrepository.com/artifact/>) 查看，如图

MVNREPOSITORY

Search for groups, artifacts, categories

Search

[Artifacts/Year](#)

Home » checkstyle » checkstyle

## Checkstyle

<http://blog.csdn.net/>

Note: This artifact was moved to:

|              |                      |
|--------------|----------------------|
| New Group    | com.puppycrawl.tools |
| New Artifact | checkstyle           |

[点击进入查看版本号](#)

Checkstyle 25 usages

## 版本列表：

Popular Categories

- Aspect Oriented
- Actor Frameworks
- Application Metrics
- Build Tools
- Bytecode Libraries
- Command Line Parsers
- Cache Implementations
- Cloud Computing
- Code Analyzers
- Collections
- Configuration Libraries
- Core Utilities
- Date and Time Utilities
- Dependency Injection
- Embedded SQL Databases
- HTML Parsers
- HTTP Clients

Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard

Tags: analysis

Find an Easier Path to Achieving USB Connectivity with Mixed Signal USB Solutions! [\(1\)](#) [X](#)

Learn how to easily add USB connectivity as embedded designs become increasingly complex.

[DOWNLOAD NOW!](#)

SILICON LABS

|        | Version | Usages | Type    | Date        |
|--------|---------|--------|---------|-------------|
|        | 6.11.2  | 1      | release | (Oct, 2015) |
| 6.11.x | 6.11.1  | 0      | release | (Oct, 2015) |
|        | 6.11    | 0      | release | (Sep, 2015) |
| 6.10.x | 6.10.1  | 0      | release | (Sep, 2015) |
|        | 6.10    | 0      | release | (Aug, 2015) |
| 6.9.x  | 6.9     | 0      | release | (Aug, 2015) |
|        | 6.8.2   | 0      | release | (Aug, 2015) |
| 6.8.x  | 6.8.1   | 2      | release | (Jul, 2015) |

## ● 2、增加checkstyle的task

```

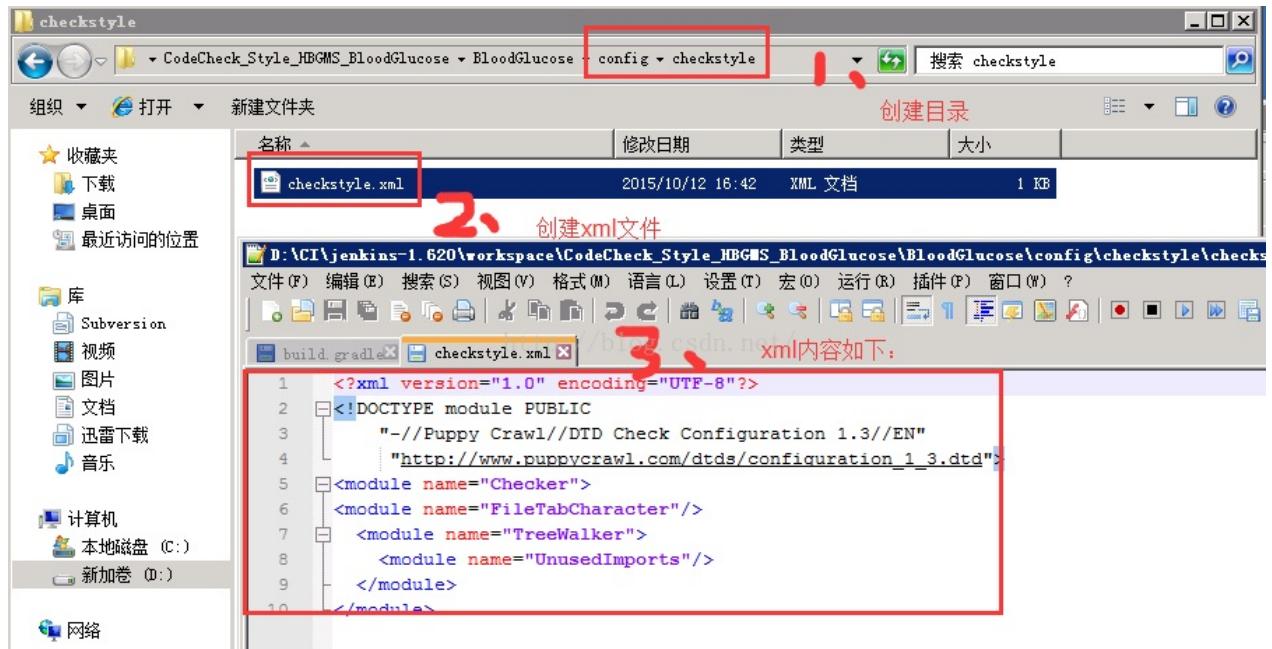
apply plugin: "checkstyle"

repositories {
 mavenCentral()
}

task checkstyle(type: Checkstyle) {
 //toolVersion = "2.0.1"
 ignoreFailures = true
 //config= files("$rootProject.projectDir/config/checkstyle/checkstyle.xml")
 source= fileTree('build/intermediates/classes/debug/com/sn/')
 classpath= files()
 reports{
 xml {
 destination "build/checkstyle-result.xml"
 }
 }
}

```

备注：其中检查规则文件 `checkstyle.xml` 需要创建，步骤如下



Xml内容代码：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC
"-//Puppy Crawl//DTD Check Configuration 1.3//EN"
"http://www.puppycrawl.com/dtds/configuration_1_3.dtd">
<module name="Checker">
<module name="FileTabCharacter"/>
<module name="TreeWalker">
<module name="UnusedImports"/>
</module>

```

## 三、构建结果查看

**Jenkins**

Jenkins > Code\_Check > CodeCheck\_Style\_HBGMs\_BloodGlucose > #23

回到工程 状态集 变更记录 Console Output 编辑编译信息 删 除本次生成 Checkstyle Warnings 前一次构建

**构建 #23 (2015-10-12 17:10:16)**

http://blog.csdn.net/ 没有变化。

启动用户董卫华

Checkstyle: 271 warnings from one analysis.

**Jenkins**

Jenkins > Code\_Check > CodeCheck\_Style\_HBGMs\_BloodGlucose > #23 > Checkstyle Warnings

回到工程 状态集 变更记录 Console Output 编辑编译信息 删 除本次生成 **Checkstyle Warnings** 前一次构建

**CheckStyle Result**

**Warnings Trend**

| All Warnings | New Warnings | Fixed Warnings |
|--------------|--------------|----------------|
| 271          | 0            | 0              |

**Summary**

| Total | High Priority | Normal Priority         | Low Priority |
|-------|---------------|-------------------------|--------------|
| 271   | 271           | http://blog.csdn.net/ 0 | 0            |

**Details**

| Folders                                                            | Files | Warnings           | Details                                                 |
|--------------------------------------------------------------------|-------|--------------------|---------------------------------------------------------|
| Source Folder                                                      |       | Total Distribution |                                                         |
| BloodGlucose/build/intermediates/classes/debug/com/sn/Fragment     |       | 20                 | <div style="width: 100%; background-color: red;"></div> |
| BloodGlucose/build/intermediates/classes/debug/com/sn/activity     |       | 13                 | <div style="width: 100%; background-color: red;"></div> |
| BloodGlucose/build/intermediates/classes/debug/com/sn/adapter      |       | 14                 | <div style="width: 100%; background-color: red;"></div> |
| BloodGlucose/build/intermediates/classes/debug/com/sn/bloodglucose |       | 32                 | <div style="width: 100%; background-color: red;"></div> |
| BloodGlucose/build/intermediates/classes/debug/com/sn/callback     |       | 2                  | <div style="width: 100%; background-color: red;"></div> |
| BloodGlucose/build/intermediates/classes/debug/com/sn/connection   |       | 3                  | <div style="width: 100%; background-color: red;"></div> |
| BloodGlucose/build/intermediates/classes/debug/com/sn/dentity      |       | 22                 | <div style="width: 100%; background-color: red;"></div> |
| BloodGlucose/build/intermediates/classes/debug/com/sn/dialog       |       | 13                 | <div style="width: 100%; background-color: red;"></div> |
| BloodGlucose/build/intermediates/classes/debug/com/sn/global       |       | 5                  | <div style="width: 100%; background-color: red;"></div> |

## 四、build.gradle的所有代码如下

```

buildscript {
 repositories {
 mavenCentral()
 }
 dependencies {
 classpath 'com.android.tools.build:gradle:1.0.0+'
 //classpath 'io.fabric.tools:gradle:1.+'
 //classpath 'com.google.code.findbugs:findbugs:3.0.1'
 //classpath 'com.puppycrawl.tools:checkstyle:6.11.2'
 //classpath 'net.sourceforge.pmd:pmd:5.4.0'
 }
}
apply plugin: 'android'

dependencies {
 compile fileTree(dir: 'libs', include: '*.jar')
}

```

```
android {
 compileSdkVersion 20
 buildToolsVersion "20.0.0"

 //忽略编码错误
 lintOptions {
 abortOnError false
 }

 //设置版本号
 defaultConfig {
 versionCode 1
 versionName "1.0"
 minSdkVersion 8
 targetSdkVersion 18
 }

 //引用so包
 sourceSets{
 main{
 jniLibs.srcDir(['libs'])
 jniLibs.srcDir(['obj'])
 }
 }

 //设置编译编码
 tasks.withType(JavaCompile) {
 options.encoding = 'UTF-8'
 }

 //autograph
 signingConfigs{
 //keystore info
 myConfig {
 storeFile file("bgkey")
 storePassword "sinocare@ydl"
 keyAlias "com.sn.bloodglucose"
 keyPassword "sinocare@ydl"
 }
 }

 //混淆
 buildTypes{
 release{
 signingConfig signingConfigs.myConfig
 minifyEnabled false
 }
 }

 sourceSets {
 main {
 }
```

```

 manifest.srcFile 'AndroidManifest.xml'
 java.srcDirs = ['src']
 resources.srcDirs = ['src']
 aidl.srcDirs = ['src']
 renderscript.srcDirs = ['src']
 res.srcDirs = ['res']
 assets.srcDirs = ['assets']
 }

 // Move the tests to tests/java, tests/res, etc...
 instrumentTest.setRoot('tests')

 // Move the build types to build-types/<type>
 // For instance, build-types/debug/java, build-types/debug/AndroidManifest.xml
 // This moves them out of them default location under src/<type>/... which would
 // conflict with src/ being used by the main source set.
 // Adding new build types or product flavors should be accompanied
 // by a similar customization.
 debug.setRoot('build-types/debug')
 release.setRoot('build-types/release')
}

apply plugin: "findbugs"

repositories {
 mavenCentral()
}

task findbugs(type: FindBugs) {
 //toolVersion = "2.0.1"
 ignoreFailures = true
 effort = "max"
 reportLevel = "low"
 classes = files("$project.buildDir/intermediates/classes")
 source = fileTree('build/intermediates/classes/debug/com/sn/')
 classpath = files()
 reports {
 xml {
 destination "build/findbugs.xml"
 }
 }
}

apply plugin: "checkstyle"

repositories {
 mavenCentral()
}
task checkstyle(type: Checkstyle) {
 ignoreFailures = true
}

```

```
//config = files("build/config/checkstyle/checkstyle.xml")
source = fileTree('build/intermediates/classes/debug/com/sn/')
classpath = files()
reports {
 xml {
 destination "build/checkstyle-result.xml"
 }
}
}

apply plugin: "pmd"

repositories {
 mavenCentral()
}
task pmd(type: Pmd) {
 ignoreFailures = true
 source = fileTree('src/com/sn/')
 //ruleSetConfig = resources.file("${project.rootDir}/config/pmd/PmdRuleSets.xml")
 //ruleSetFiles = files("config/pmd/PmdRuleSets.xml")
 ruleSetFiles = files("${project.rootDir}/config/pmd/PmdRuleSets.xml")
 ruleSets = ["java-android"]
 reports {
 xml {
 destination "build/pmd.xml"
 }
 }
}
```

# Jenkins+Gradle+findbugs对Android工程源码进行静态代码分析

来源:测试蜗牛,一步一个脚印

## 环境说明

```
Gradle 2.6.
OS: windows server 2008
Jenkins 1.620
Findbugs 3.0.1
```

## 前提

Jenkins需要提前安装好FindBugs Plug-in插件

## 一、Jenkins配置如下:

- 1、新建job

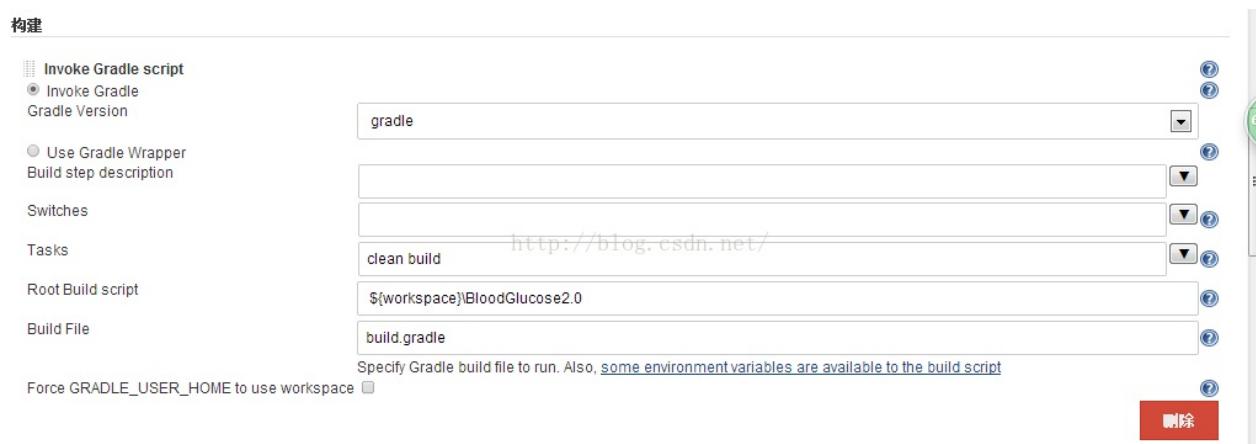
The screenshot shows the Jenkins job configuration page for 'CodeCheck\_HBGMS\_BloodGlucose'. The 'Project name' field contains 'CodeCheck\_HBGMS\_BloodGlucose'. The 'Description' field is empty. Under 'Build Triggers', the 'Restrict where this project can be run' checkbox is checked. The 'Label Expression' field contains 'Master\_15'. The 'Build History' section shows four recent builds: #169 (2015-10-12下午3:44), #168 (2015-10-12下午3:33), #167 (2015-10-12下午3:29), and #166 (2015-10-12下午3:25). A 'Advanced...' button is visible at the bottom right.

- 2、配置svn

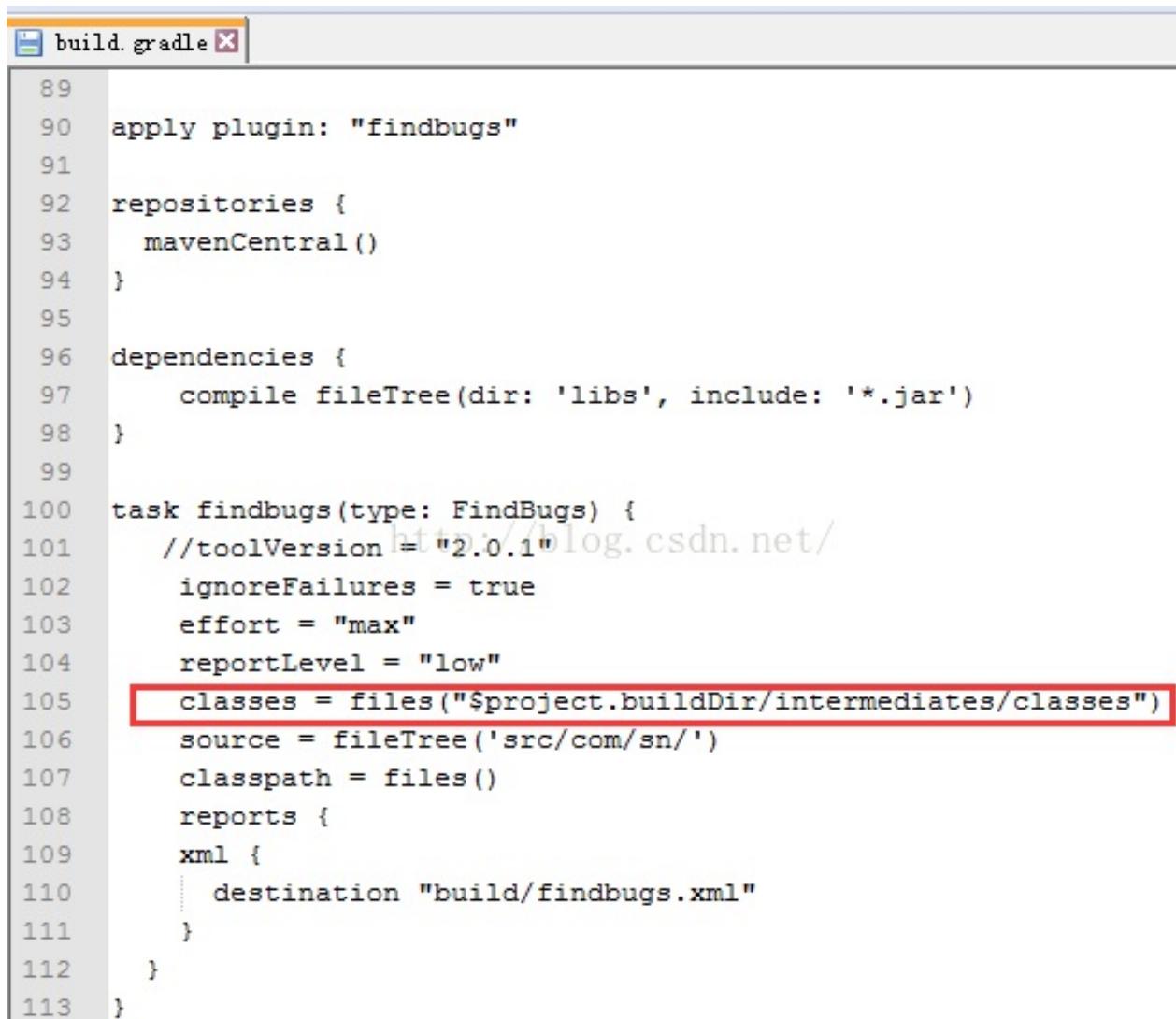


## 3、配置构建操作

### 先配置build构建操作



为什么要配置build任务呢？从下图的findbugs的task任务可以看出findbugs要检查class文件，class文件需要编译java文件才能生成。



```

89
90 apply plugin: "findbugs"
91
92 repositories {
93 mavenCentral()
94 }
95
96 dependencies {
97 compile fileTree(dir: 'libs', include: '*.jar')
98 }
99
100 task findbugs(type: FindBugs) {
101 //toolVersion = "2.0.1"
102 ignoreFailures = true
103 effort = "max"
104 reportLevel = "low"
105 classes = files("$project.buildDir/intermediates/classes")
106 source = fileTree('src/com/sn/')
107 classpath = files()
108 reports {
109 xml {
110 destination "build/findbugs.xml"
111 }
112 }
113 }

```

再配置findbugs检查任务。如下图：



备注：

Tasks指的是build.gradle里面的task名称

配置info参数是用来查看调试日志，也可以配置debug级别。主要用来查看构建失败的原因。

- 4、配置分析报告

构建后操作

Published FindBugs analysis results

FindBugs results

Fileset includes setting that specifies the generated raw FindBugs XML report files, such as \*\*/findbugs.xml or \*\*/findbugsXml.xml. Basedir of the fileset is [the workspace root](#). If no value is set, then the default \*\*/findbugsXml.xml or \*\*/findbugs.xml are used for maven or ant builds, respectively. Be sure not to include any non-report files into this pattern.

Use rank as priority

Files to include  Uses the bug rank when evaluating the priority of the warnings (otherwise the FindBugs priority is used).

Files to exclude  Comma separated list of [regular expressions](#) that specifies the files to include in the report (based on their absolute filename). If this field is empty then all files are included.

Run always  <http://blog.csdn.net/>

By default, this plug-in runs only for stable or unstable builds, but not for failed builds. If this plug-in should run even for failed builds then activate this check box.

Detect modules

Health thresholds  100%  0%

Configure the thresholds for the build health. If left empty then no health report is created. If the actual number of warnings is between the provided thresholds then the build health is interpolated.

Health priorities  Only priority high  Priorities high and normal  All priorities

Determines which warning priorities should be considered when evaluating the build health.

Status thresholds (Totals) All priorities Priority high Priority normal Priority low

**保存** **应用**

## 二、gradle.build的配置如下

- 1、添加依赖

```
buildscript {
 repositories {
 mavenCentral()
 }
 dependencies {
 classpath 'com.android.tools.build:gradle:1.0.0+'
 }
}
```

备注：版本包可以通过中央仓库

[<http://mvnrepository.com/artifact/>] [<http://mvnrepository.com/artifact/>) 查看，如图

## Group: com.google.code.findbugs

**Popular Categories**

- Aspect Oriented
- Actor Frameworks
- Application Metrics
- Build Tools
- Bytecode Libraries
- Command Line Parsers
- Cache Implementations
- Cloud Computing
- Code Analyzers
- Collections
- Configuration Libraries
- Core Utilities
- Date and Time Utilities
- Dependency Injection
- Embedded SQL Databases
- HTML Parsers
- HTTP Clients

**com.google.code.findbugs**  
Used by 2029 artifacts

**Google AdWords必备技巧**  
30天30个AdWords免费必备技巧学习，行业专家撰写，轻松从入门到进阶 >

| Artifact                                    | Usages                     | Last Version | Description                                                                    |
|---------------------------------------------|----------------------------|--------------|--------------------------------------------------------------------------------|
| jsr305 (9)<br>FindBugs Jsr305               | http://blog.csdn.net/1,498 | 3.0.1        | JSR305 Annotations for Findbugs<br>Defect Detection Metadata<br>Code Analyzers |
| annotations (11)<br>FindBugs Annotations    | 428                        | 3.0.1        | Annotation the FindBugs tool supports<br>Defect Detection Metadata             |
| findbugs (9)<br>FindBugs Project            | 78                         | 3.0.1        | Findbugs: Because it's easy!<br>Code Analyzers                                 |
| bcel (6)<br>FindBugs Bcel                   | 9                          | 2.0.2        | Modified BCEL for Findbugs                                                     |
| jFormatString (7)<br>FindBugs JFormatString | 8                          | 3.0.0        | jFormatString for Findbugs                                                     |
| findbugs-ant (8)<br>FindBugs AntTask        | 5                          | 3.0.0        | AntTask to run Findbugs                                                        |
| bcel-findbugs (1)<br>FindBugs Bcel          | 3                          | 6.0          | Modified BCEL for Findbugs                                                     |

- 2、增加findbugs的task

```
apply plugin: "findbugs"

repositories {
 mavenCentral()
}

task findbugs(type: FindBugs) {
 //toolVersion = "3.0.1"
 ignoreFailures= true
 effort= "max"
 reportLevel= "low"
 classes = files("$project.buildDir/intermediates/classes")
 source= fileTree('build/intermediates/classes/debug/com/sn/')
 classpath= files()
 reports{
 xml {
 destination "build/findbugs.xml"
 }
 }
}
```

## 三、构建结果查询



Jenkins > Code\_Check > CodeCheck\_HBGMS\_BloodGlucose > #169

构建 #169 (2015-10-12 15:44:46)

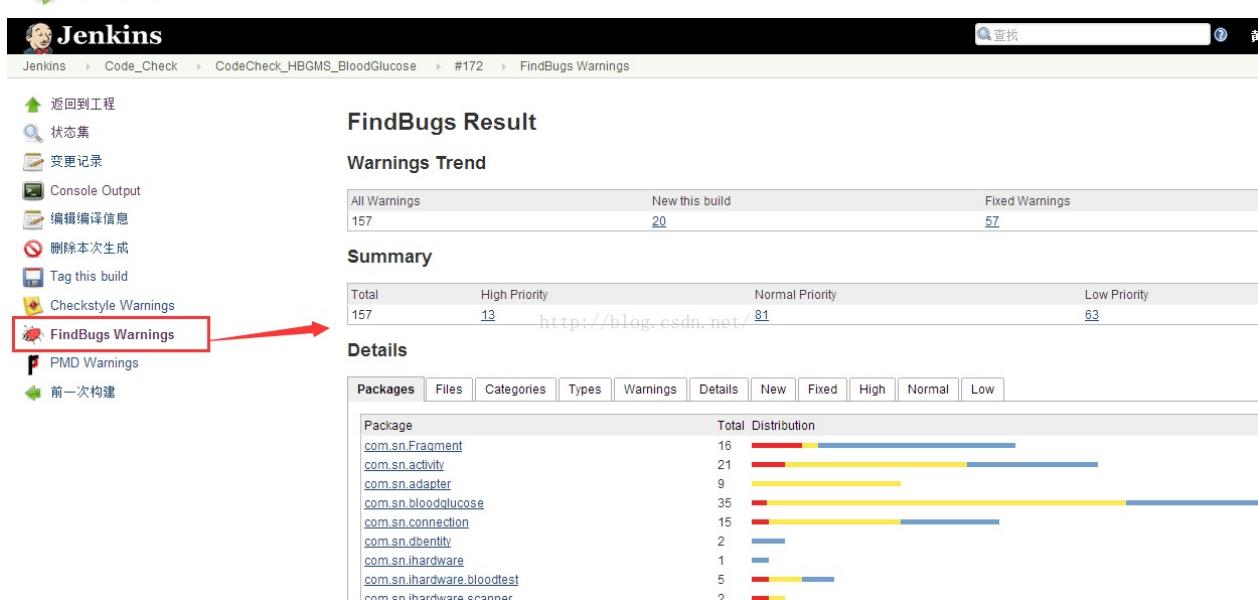
- 返回到工程
- 状态集
- 变更记录
- Console Output
- View as plain text
- 编辑编译信息
- 删除本次生成
- FindBugs Warnings**
- 前一次构建
- 后一次构建

http://blog.csdn.net/ 没有变化。

启动用户善卫化

FindBugs: 194 warnings from one analysis.

- 194 new warnings

Jenkins > Code\_Check > CodeCheck\_HBGMS\_BloodGlucose > #172 > FindBugs Warnings

### FindBugs Result

#### Warnings Trend

| All Warnings | New this build | Fixed Warnings |
|--------------|----------------|----------------|
| 157          | 20             | 57             |

#### Summary

| Total | High Priority | Normal Priority | Low Priority |
|-------|---------------|-----------------|--------------|
| 157   | 13            | 81              | 63           |

#### Details

| Packages                  | Files | Categories | Types | Warnings           | Details | New | Fixed | High | Normal | Low |
|---------------------------|-------|------------|-------|--------------------|---------|-----|-------|------|--------|-----|
| Package                   |       |            |       | Total Distribution |         |     |       |      |        |     |
| com.sn.Fragment           |       |            |       | 16                 | 16      |     |       |      |        |     |
| com.sn.activity           |       |            |       | 21                 | 21      |     |       |      |        |     |
| com.sn.adapter            |       |            |       | 9                  | 9       |     |       |      |        |     |
| com.sn.bloodglucose       |       |            |       | 35                 | 35      |     |       |      |        |     |
| com.sn.connection         |       |            |       | 15                 | 15      |     |       |      |        |     |
| com.sn.databind           |       |            |       | 2                  | 2       |     |       |      |        |     |
| com.sn.hardware           |       |            |       | 1                  | 1       |     |       |      |        |     |
| com.sn.hardware.bloodtest |       |            |       | 5                  | 5       |     |       |      |        |     |
| com.sn.hardware.scanner   |       |            |       | 2                  | 2       |     |       |      |        |     |

## 四、build.gradle的所有代码如下

```

buildscript {
 repositories {
 mavenCentral()
 }
 dependencies {
 classpath 'com.android.tools.build:gradle:1.0.0+'
 //classpath 'io.fabric.tools:gradle:1.+'
 //classpath 'com.google.code.findbugs:findbugs:3.0.1'
 //classpath 'com.puppycrawl.tools:checkstyle:6.11.2'
 //classpath 'net.sourceforge.pmd:pmd:5.4.0'
 }
}
apply plugin: 'android'

dependencies {
 compile fileTree(dir: 'libs', include: '*.jar')
}

```

```
}

android {
 compileSdkVersion 20
 buildToolsVersion "20.0.0"

 //忽略编码错误
 lintOptions {
 abortOnError false
 }

 //设置版本号
 defaultConfig {
 versionCode 1
 versionName "1.0"
 minSdkVersion 8
 targetSdkVersion 18
 }

 //引用so包
 sourceSets{
 main{
 jniLibs.srcDir(['libs'])
 jniLibs.srcDir(['obj'])
 }
 }

 //设置编译编码
 tasks.withType(JavaCompile) {
 options.encoding = 'UTF-8'
 }

 //autograph
 signingConfigs{
 //keystore info
 myConfig {
 storeFile file("bgkey")
 storePassword "sinocare@ydy1"
 keyAlias "com.sn.bloodglucose"
 keyPassword "sinocare@ydy1"
 }
 }

 //混淆
 buildTypes{
 release{
 signingConfig signingConfigs.myConfig
 minifyEnabled false
 }
 }

 sourceSets {
```

```

main {
 manifest.srcFile 'AndroidManifest.xml'
 java.srcDirs = ['src']
 resources.srcDirs = ['src']
 aidl.srcDirs = ['src']
 renderscript.srcDirs = ['src']
 res.srcDirs = ['res']
 assets.srcDirs = ['assets']
}

// Move the tests to tests/java, tests/res, etc...
instrumentTest.setRoot('tests')

// Move the build types to build-types/<type>
// For instance, build-types/debug/java, build-types/debug/AndroidManifest.xml
// This moves them out of their default location under src/<type>/... which would
// conflict with src/ being used by the main source set.
// Adding new build types or product flavors should be accompanied
// by a similar customization.
debug.setRoot('build-types/debug')
release.setRoot('build-types/release')
}

apply plugin: "findbugs"

repositories {
 mavenCentral()
}

task findbugs(type: FindBugs) {
 //toolVersion = "2.0.1"
 ignoreFailures = true
 effort = "max"
 reportLevel = "low"
 classes = files("$project.buildDir/intermediates/classes")
 source = fileTree('build/intermediates/classes/debug/com/sn/')
 classpath = files()
 reports {
 xml {
 destination "build/findbugs.xml"
 }
 }
}

apply plugin: "checkstyle"

repositories {
 mavenCentral()
}
task checkstyle(type: Checkstyle) {

```

```
ignoreFailures = true
//config = files("build/config/checkstyle/checkstyle.xml")
source = fileTree('build/intermediates/classes/debug/com/sn/')
classpath = files()
reports {
 xml {
 destination "build/checkstyle-result.xml"
 }
}
}

apply plugin: "pmd"

repositories {
 mavenCentral()
}
task pmd(type: Pmd) {
 ignoreFailures = true
 source = fileTree('src/com/sn/')
 //ruleSetConfig = resources.file("${project.rootDir}/config/pmd/PmdRuleSets.xml")
 //ruleSetFiles = files("config/pmd/PmdRuleSets.xml")
 ruleSetFiles = files("${project.rootDir}/config/pmd/PmdRuleSets.xml")
 ruleSets = ["java-android"]
 reports {
 xml {
 destination "build/pmd.xml"
 }
 }
}
```

## 备注

FindBugs Gradle任务详细配置：

- <https://docs.gradle.org/current/dsl/org.gradle.api.plugins.quality.FindBugsExtension.html#org.gradle.api.plugins.quality.FindBugsExtension:excludeFilter>
- [https://docs.gradle.org/current/userguide/findbugs\\_plugin.html#useFindBugsPlugin](https://docs.gradle.org/current/userguide/findbugs_plugin.html#useFindBugsPlugin)

# Jenkins+Gradle+Lint对Android工程源码进行静态代码分析

来源:测试蜗牛,一步一个脚印

## Lint的介绍

### 官网介绍

The Android linttool is a static code analysis tool that checks your Android project sourcefiles for potential bugs and optimization improvements for correctness, security, performance, usability, accessibility, and internationalization.

Android Lint工具是一个静态代码分析工具,检查你的Android项目源文件为潜在的bug和优化改进正确性,安全性、性能、可用性、可访问性和国际化。

## 静态检查原理

Figure 1 shows how the lint tool processes the application source files.

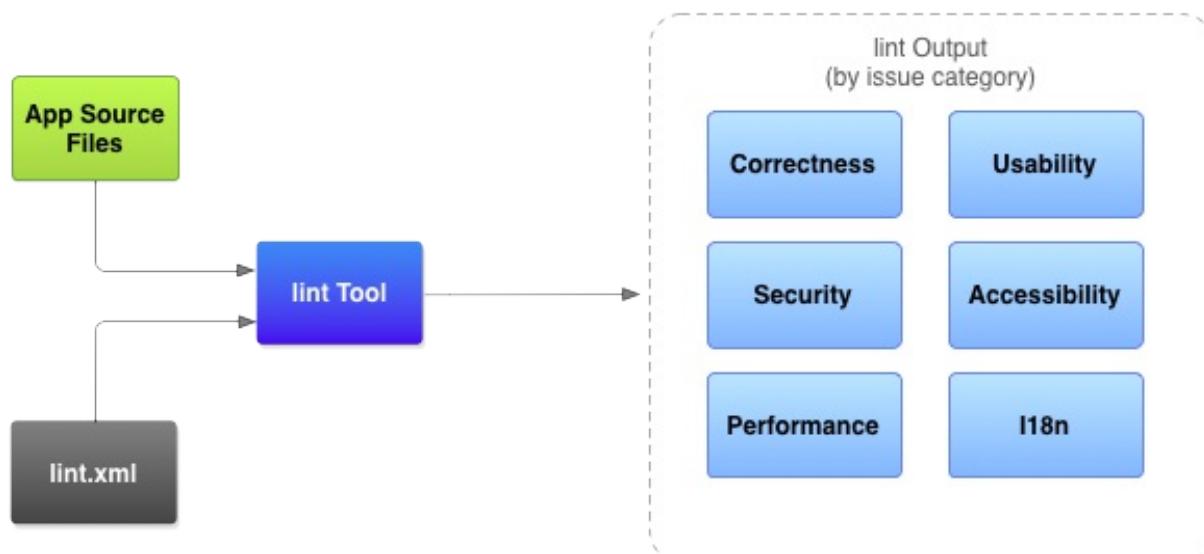


Figure 1. Code scanning workflow with the lint tool

- Application source files

The source files consist of files that make up your Android project, including Java and XML files, icons, and ProGuard configuration files.

- The lint.xml file

A configuration file that you can use to specify any lint checks that you want to exclude and to customize problem severity levels.

- The lint tool

A static code scanning tool that you can run on your Android project from the command-line or from Eclipse. The lint tool checks for structural code problems that could affect the quality and performance of your Android application. It is strongly recommended that you correct any errors that lint detects before publishing your application.

- Results of lint checking

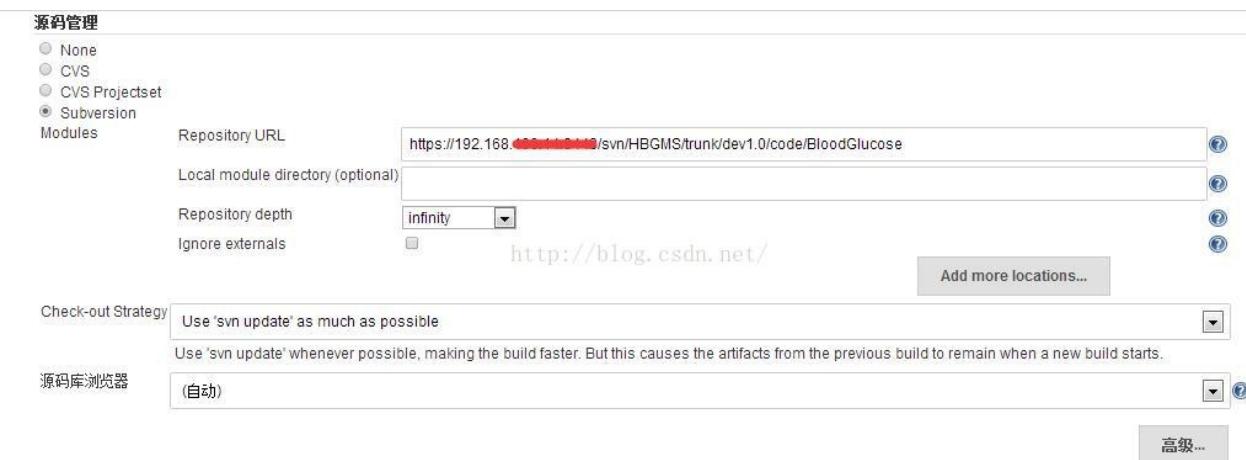
You can view the results from lint in the console or in the Lint Warnings view in Eclipse. Each issue is identified by the location in the source files where it occurred and a description of the issue.

## Jenkins配置如下：

### 新建job

The screenshot shows the Jenkins 'Configure' screen for a job named 'CodeCheck\_Lint\_HBGMS\_BloodGlucose'. On the left, there's a sidebar with links like '返回面板', '状态', '修改记录', '工作空间', '立即构建', '删除 Project', '配置', and 'Lint Issues'. The main area has fields for '项目名称' (set to 'CodeCheck\_Lint\_HBGMS\_BloodGlucose') and '描述'. Below these are several checkboxes: '丢弃旧的构建', '参数化构建过程', '关闭构建 (重新开启构建前不允许进行新的构建)', '在必要的时候并发构建', and 'Restrict where this project can be run' (which is checked). A 'Label Expression' field contains 'Master\_15'. Under '高级项目选项', there's a 'Slaves in label: 1' section. At the bottom, there are '高级...' and '取消' buttons. The build history section shows three builds: #3 (2015-11-27 上午10:05), #2 (2015-10-21 上午9:51), and #1 (2015-10-21 上午9:50). There are also 'RSS 全部' and 'RSS 失败' links.

### 配置svn



## 配置构建操作



## 配置分析报告生成步骤

**构建后操作**

**Publish Android Lint results**

**Lint files**

**\*\*/lint-results\*.xml**

执行构建命令 Lint 时，生成的结果文件为 lint-results.xml  
执行构建命令 Lint-debug 时，生成的结果文件为 lint-results-debug.xml  
此处用通配符\*来做即可。

Fileset includes setting that specifies the generated Lint XML report files, such as \*\*/lint-results.xml. Basedir of the fileset is the workspace root. If no value is set, then the default \*\*/lint-results.xml is used. Be sure not to include any non-report files into this pattern.

**Run always**

By default, this plug-in runs only for stable or unstable builds, but not for failed builds. If this plug-in should run even for failed builds then activate this check box.

**Detect modules**

Determines if Ant or Maven modules should be detected for all files that contain warnings. Activating this option may increase your build time since the detector scans the whole workspace for 'build.xml' or 'pom.xml' files in order to assign the correct module names.

**Health thresholds**

**100%** **0%**

Configure the thresholds for the build health. If left empty then no health report is created. If the actual number of warnings is between the provided thresholds then the build health is interpolated.

**Health priorities**

Only priority high  Priorities high and normal  All priorities

Determines which warning priorities should be considered when evaluating the build health.

**Status thresholds (Totals)**

| All priorities | Priority high | Priority normal | Priority low |
|----------------|---------------|-----------------|--------------|
| Yellow         |               |                 |              |
| Red            |               |                 |              |

If the number of total warnings is greater than one of these thresholds then a build is considered as unstable or failed, respectively. I.e., a value of 0 means that the build status is changed if there is at least one warning found. Leave this field empty if the state of the build should not depend on the number of warnings.

Compute new warnings (based on the last successful build unless another reference build is chosen below)

**Default Encoding**

Default encoding when parsing or showing files. Leave this field empty to use the default encoding of the platform

**保存** **应用**

## 构建结果查看

- 点击报告

Jenkins > Code\_Check > Project CodeCheck\_Lint\_HBGMS\_BloodGlucose

**Project CodeCheck\_Lint\_HBGMS\_BloodGlucose**

返回面板 状态 修改记录 工作空间 立即构建 删除 Project 配置 Lint Issues

工作区 最新修改记录

**相关连接**

- Last build(#3), 13 分之前
- Last stable build(#3), 13 分之前
- Last successful build(#3), 13 分之前
- Last failed build(#1), 1 月 7 days之前
- Last unsuccessful build(#1), 1 月 7 days之前

**Build History**

| #  | 构建时间               |
|----|--------------------|
| #3 | 2015-11-27 上午10:05 |
| #2 | 2015-10-21 上午9:51  |
| #1 | 2015-10-21 上午9:50  |

- 报告详细列表

Jenkins > Code\_Check > CodeCheck\_Lint\_HBGMs\_BloodGlucose #3 Lint Issues

返回到工程 状态集 变更记录 Console Output 编译错误信息 删除本次生成 Tag this build Lint Issues 前一次构建

## Lint Issues

### Warnings Trend

| All Warnings | New Warnings | Fixed Warnings |
|--------------|--------------|----------------|
| 841          | 51           | 30             |

### Summary

| Total | High Priority | Normal Priority | Low Priority |
|-------|---------------|-----------------|--------------|
| 841   | 46            | 795             | 0            |

### Details

| Folders                         | Files | Categories | Types | Warnings | Details | New          | Fixed | High | Normal |
|---------------------------------|-------|------------|-------|----------|---------|--------------|-------|------|--------|
| Source Folder                   |       |            |       |          | Total   | Distribution |       |      |        |
| BloodGlucose                    |       |            |       |          | 30      |              |       |      |        |
| BloodGlucose/gradle/wrapper     |       |            |       |          | 1       |              |       |      |        |
| BloodGlucose/libs               |       |            |       |          | 2       |              |       |      |        |
| BloodGlucose/res                |       |            |       |          | 2       |              |       |      |        |
| BloodGlucose/res/anim           |       |            |       |          | 5       |              |       |      |        |
| BloodGlucose/res/drawable       |       |            |       |          | 15      |              |       |      |        |
| BloodGlucose/res/drawable-hdpi  |       |            |       |          | 15      |              |       |      |        |
| BloodGlucose/res/drawable-nodpi |       |            |       |          | 76      |              |       |      |        |
| BloodGlucose/res/drawable-xhdpi |       |            |       |          | 1       |              |       |      |        |

# Jenkins+Gradle+pmd对Android工程源码进行静态代码分析

来源:测试蜗牛,一步一个脚印

## 环境说明

```
Gradle 2.6.
OS: windows server 2008
Jenkins 1.620
pmd 5.4.0
```

## 前提

Jenkins需要提前安装好PMD Plug-in插件

## 一、Jenkins配置如下:

- 1、新建job

The screenshot shows the Jenkins project configuration page for 'CodeCheck\_Pmd\_HBGMs\_BloodGlucose'. The left sidebar contains links for '返回面板', '状态', '修改记录', '工作空间', '立即构建', '删除 Project', '配置', and 'PMD Warnings'. The main configuration area has fields for '项目名称' (CodeCheck\_Pmd\_HBGMs\_BloodGlucose) and '描述'. Below these are several checkboxes: '丢弃旧的构建', '参数化构建过程', '关闭构建 (重新开启构建前不允许进行新的构建)', '在必要的时候并发构建', and 'Restrict where this project can be run'. A 'Label Expression' field is set to 'Master\_15'. The '高级项目选项' section includes a 'Slaves in label: 1' dropdown. At the bottom right is a '高级...' button.

- 2、配置svn



## ● 3、配置构建操作

### 构建

The screenshot shows the Jenkins build configuration. The 'Build' section is open, specifically the 'Invoke Gradle script' step. The configuration includes:

- Invoke Gradle script (radio button selected)
- Gradle Version: `gradle2.6` (highlighted with a red box)
- Use Gradle Wrapper
- Build step description: `pmd --debug http://blog.csdn.net/` (highlighted with a red box)
- Switches: `...`
- Tasks: `pmd --debug http://blog.csdn.net/`, `SB{workspace}!BloodGlucose`, `build.gradle` (highlighted with a red box)
- Root Build script: `...`
- Build File: `...`
- Specify Gradle build file to run. Also, [some environment variables are available to the build script](#)
- Force GRADLE\_USER\_HOME to use workspace
- Delete button

### 备注：

Tasks指的是build.gradle里面的task名称

配置info参数是用来查看调试日志，也可以配置debug级别。主要用来查看构建失败的原因。

## ● 4、配置分析报告

## 构建后操作

Publish PMD analysis results

PMD results

Fileset includes setting that specifies the generated raw PMD XML report files, such as `**/pmd.xml`. Basedir of the fileset is the workspace root. If no value is set, then the default `**/pmd.xml` is used. Be sure not to include any non-report files into this pattern.

Run always

By default, this plug-in runs only for stable or unstable builds, but not for failed builds. If this plug-in should run even for failed builds then activate this check box.

Detect modules

Determines if Ant or Maven modules should be detected for all files that contain warnings. Activating this option may increase your build time since the detector scans the whole workspace for 'build.xml' or 'pom.xml' files in order to assign the correct module names.

Health thresholds

100% 0%

Configure the thresholds for the build health. If left empty then no health report is created. If the actual number of warnings is between the provided thresholds then the build health is interpolated.

Only priority high  Priorities high and normal  All priorities

Determines which warning priorities should be considered when evaluating the build health.

Status thresholds (Totals)

| All priorities | Priority high | Priority normal | Priority low |
|----------------|---------------|-----------------|--------------|
|                |               |                 |              |
|                |               |                 |              |

If the number of total warnings is greater than one of these thresholds then a build is considered as unstable or failed, respectively. I.e., a value of 0 means that the build status is changed if there is at least one warning found. Leave this field empty if the state of the build should not depend on the number of warnings.

Compute new warnings (based on the last successful build unless another reference build is chosen below)

Default Encoding

UTF-8

Default encoding when parsing or showing files. Leave this field empty to use the default encoding of the platform.

**保存** **应用**

## 二、gradle.build的配置如下

- 1、添加checkstyle的依赖

```
buildscript {
 repositories {
 mavenCentral()
 }
 dependencies {
 classpath 'com.android.tools.build:gradle:1.0.0+'
 //classpath 'io.fabric.tools:gradle:1.+'
 //classpath 'com.google.code.findbugs:findbugs:3.0.1'
 //classpath 'com.puppycrawl.tools:checkstyle:6.11.2'
 //classpath 'net.sourceforge.pmd:pmd:5.4.0'
 }
}
```

备注：版本包可以通过中央仓库 (<http://mvnrepository.com/artifact/>) 查看，如图

MVNREPOSITORY

Search for groups, artifacts, categories

Artifacts/Year

Home > pmd > pmd

**PMD**

<http://blog.csdn.net/>

Note: This artifact was moved to:

|              |                     |
|--------------|---------------------|
| New Group    | net.sourceforge.pmd |
| New Artifact | pmd                 |

点击进入查看版本号

## 版本列表：

|       | Version      | Usages | Type    | Date                  |
|-------|--------------|--------|---------|-----------------------|
| 5.4.x | <b>5.4.0</b> | 0      | release |                       |
|       | 5.3.5        | 0      | release |                       |
|       | 5.3.4        | 0      | release | http://blog.csdn.net/ |
|       | 5.3.3        | 0      | release |                       |
|       | 5.3.2        | 0      | release |                       |
|       | 5.3.1        | 0      | release |                       |
|       | 5.3.0        | 0      | release |                       |
|       | 5.2.3        | 0      | release |                       |
|       | 5.2.2        | 0      | release |                       |
|       | 5.2.1        | 0      | release |                       |

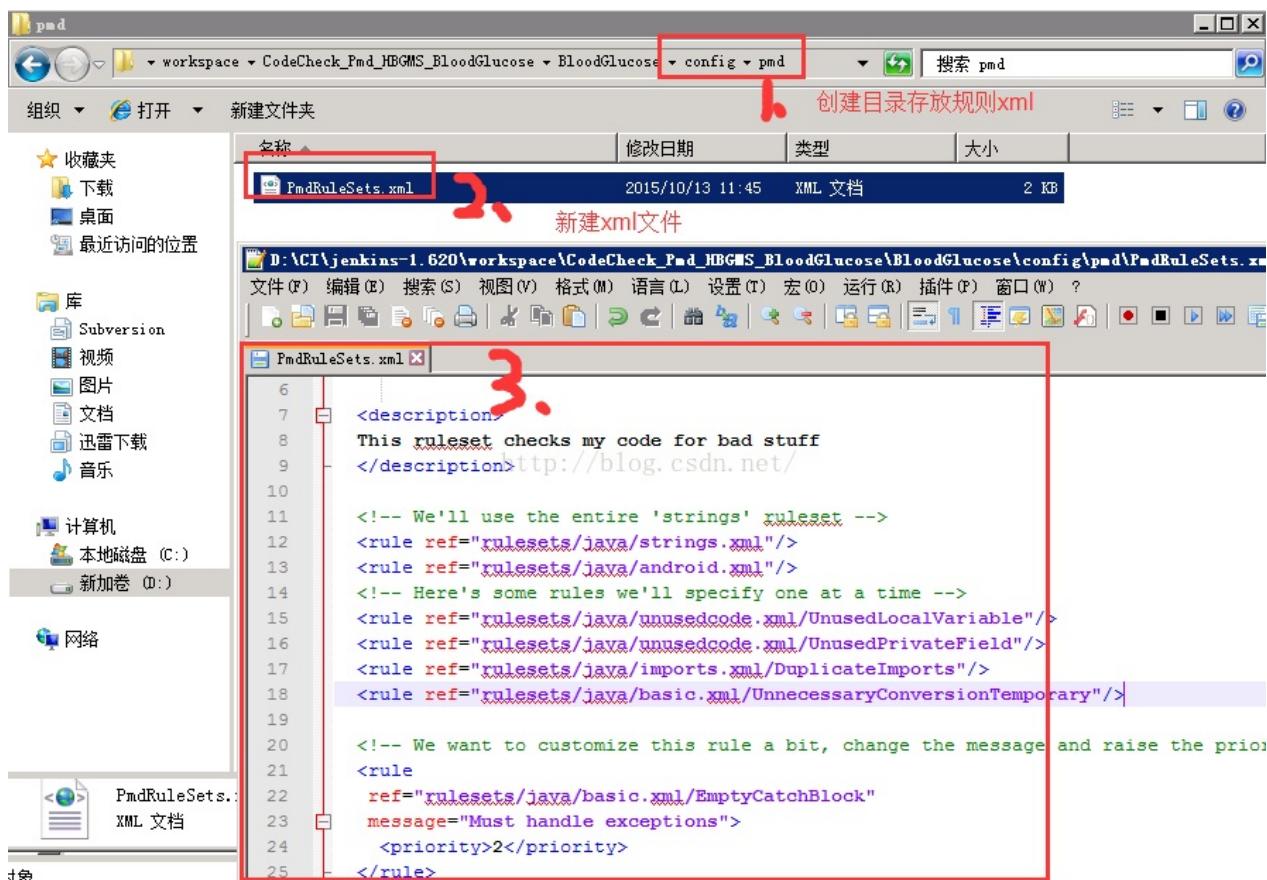
## ● 2、增加checkstyle的task

```

applyplugin: "pmd"
repositories{
 mavenCentral()
}
taskpmd(type: Pmd) {
 ignoreFailures = true
 source = fileTree('src/com/sn/')
 //ruleSetConfig =resources.file("${project.rootDir}/config/pmd/PmdRuleSets.xml")
 //ruleSetFiles =files("config/pmd/PmdRuleSets.xml")
 ruleSetFiles =files("${project.rootDir}/config/pmd/PmdRuleSets.xml")
 ruleSets = ["java-android"]
 reports {
 xml {
 destination "build/pmd.xml"
 }
 }
}

```

备注：其中检查规则文件PmdRuleSets.xml需要创建，步骤如下



Xml内容代码(可以参考官网配置):

PMD'官网地址: <http://pmd.sourceforge.net/pmd-5.1.1/howtomakearuleset.html>

```
<?xml version="1.0"?>
<ruleset name="Custom ruleset"
 xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0http://pmd.sourceforge.net/ruleset/2.0.0.xsd">

 <description>
 This ruleset checks my code for bad stuff
 </description>

 <!-- We'll use the entire 'strings' ruleset -->
 <rule ref="rulesets/java/strings.xml"/>
 <rule ref="rulesets/java/android.xml"/>
 <!-- Here's some rules we'll specify one at a time -->
 <ruleref="rulesets/java/unusedcode.xml/UnusedLocalVariable"/>
 <ruleref="rulesets/java/unusedcode.xml/UnusedPrivateField"/>
 <rule ref="rulesets/java/imports.xml/DuplicateImports"/>
 <rule ref="rulesets/java/basic.xml/UnnecessaryConversionTemporary"/>

 <!-- We want to customize this rule a bit, change the message and raise the priority
 <rule
 ref="rulesets/java/basic.xml/EmptyCatchBlock"
 message="Must handle exceptions">
 <priority>2</priority>
 </rule>

 <!-- Now we'll customize a rule's property value -->
 <ruleref="rulesets/java/codesize.xml/CyclomaticComplexity">
 <properties>
 <property name="reportLevel" value="5"/>
 </properties>
 </rule>

 <!-- We want everything from braces.xml except WhileLoopsMustUseBraces -->
 <rule ref="rulesets/java/braces.xml">
 <exclude name="WhileLoopsMustUseBraces"/>
 </rule>
</ruleset>
```

### 三、构建结果查看

The screenshot shows the Jenkins interface for a build named "Code\_Check" of a project "CodeCheck\_Pmd\_HBGMS\_BloodGlucose". The build number is #23, and it was triggered by user "黄卫华" at 2015-10-13 12:28:52. The main dashboard displays several status icons and links:

- 回到工程 (Return to Project)
- 状态集 (Status Set)
- 变更记录 (Change Log)
- Console Output
- View as plain text
- 编辑编译信息 (Edit Build Information)
- 删除本次生成 (Delete This Build)
- PMD Warnings (highlighted with a red box)
- 前一次构建 (Previous Build)

On the right, there's a summary for the URL <http://blog.csdn.net/>:

- 没有变化。 (No changes.)
- 启动用户 黄卫华 (Triggered by user Huang Weihua)
- PMD: 200 warnings from one analysis.
- 200 new warnings

**PMD Result**

**Warnings Trend**

| All Warnings | New Warnings | Fixed Warnings |
|--------------|--------------|----------------|
| 200          | 200          | 0              |

**Summary**

| Total | High Priority | Normal Priority                                               | Low Priority |
|-------|---------------|---------------------------------------------------------------|--------------|
| 200   | 1             | <a href="http://blog.csdn.net/">http://blog.csdn.net/</a> 199 | 0            |

**Details**

| Packages                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Files              | Categories | Types | Warnings | Details | New | High | Normal |         |                    |                 |    |                 |    |                |   |                     |    |                   |   |                 |   |               |   |                   |   |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------|------------|-------|----------|---------|-----|------|--------|---------|--------------------|-----------------|----|-----------------|----|----------------|---|---------------------|----|-------------------|---|-----------------|---|---------------|---|-------------------|---|
| <table border="1"> <thead> <tr> <th>Package</th> <th>Total Distribution</th> </tr> </thead> <tbody> <tr> <td>com.bn.Fragment</td> <td>22</td> </tr> <tr> <td>com.bn.activity</td> <td>15</td> </tr> <tr> <td>com.bn.adapter</td> <td>7</td> </tr> <tr> <td>com.bn.bloodglucose</td> <td>15</td> </tr> <tr> <td>com.bn.connection</td> <td>1</td> </tr> <tr> <td>com.bn.identity</td> <td>1</td> </tr> <tr> <td>com.bn.dialog</td> <td>6</td> </tr> <tr> <td>com.bn.thirdparty</td> <td>2</td> </tr> </tbody> </table> |                    |            |       |          |         |     |      |        | Package | Total Distribution | com.bn.Fragment | 22 | com.bn.activity | 15 | com.bn.adapter | 7 | com.bn.bloodglucose | 15 | com.bn.connection | 1 | com.bn.identity | 1 | com.bn.dialog | 6 | com.bn.thirdparty | 2 |
| Package                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | Total Distribution |            |       |          |         |     |      |        |         |                    |                 |    |                 |    |                |   |                     |    |                   |   |                 |   |               |   |                   |   |
| com.bn.Fragment                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | 22                 |            |       |          |         |     |      |        |         |                    |                 |    |                 |    |                |   |                     |    |                   |   |                 |   |               |   |                   |   |
| com.bn.activity                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | 15                 |            |       |          |         |     |      |        |         |                    |                 |    |                 |    |                |   |                     |    |                   |   |                 |   |               |   |                   |   |
| com.bn.adapter                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | 7                  |            |       |          |         |     |      |        |         |                    |                 |    |                 |    |                |   |                     |    |                   |   |                 |   |               |   |                   |   |
| com.bn.bloodglucose                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | 15                 |            |       |          |         |     |      |        |         |                    |                 |    |                 |    |                |   |                     |    |                   |   |                 |   |               |   |                   |   |
| com.bn.connection                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | 1                  |            |       |          |         |     |      |        |         |                    |                 |    |                 |    |                |   |                     |    |                   |   |                 |   |               |   |                   |   |
| com.bn.identity                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | 1                  |            |       |          |         |     |      |        |         |                    |                 |    |                 |    |                |   |                     |    |                   |   |                 |   |               |   |                   |   |
| com.bn.dialog                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | 6                  |            |       |          |         |     |      |        |         |                    |                 |    |                 |    |                |   |                     |    |                   |   |                 |   |               |   |                   |   |
| com.bn.thirdparty                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | 2                  |            |       |          |         |     |      |        |         |                    |                 |    |                 |    |                |   |                     |    |                   |   |                 |   |               |   |                   |   |

## 四、build.gradle的所有代码如下

```

buildscript {
 repositories {
 mavenCentral()
 }
 dependencies {
 classpath 'com.android.tools.build:gradle:1.0.0+'
 //classpath 'io.fabric.tools:gradle:1.+'
 //classpath 'com.google.code.findbugs:findbugs:3.0.1'
 //classpath 'com.puppycrawl.tools:checkstyle:6.11.2'
 //classpath 'net.sourceforge.pmd:pmd:5.4.0'
 }
}
apply plugin: 'android'

dependencies {
 compile fileTree(dir: 'libs', include: '*.jar')
}

```

```
android {
 compileSdkVersion 20
 buildToolsVersion "20.0.0"

 //忽略编码错误
 lintOptions {
 abortOnError false
 }

 //设置版本号
 defaultConfig {
 versionCode 1
 versionName "1.0"
 minSdkVersion 8
 targetSdkVersion 18
 }

 //引用so包
 sourceSets{
 main{
 jniLibs.srcDir(['libs'])
 jniLibs.srcDir(['obj'])
 }
 }

 //设置编译编码
 tasks.withType(JavaCompile) {
 options.encoding = 'UTF-8'
 }

 //autograph
 signingConfigs{
 //keystore info
 myConfig {
 storeFile file("bgkey")
 storePassword "sinocare@ydy1"
 keyAlias "com.sn.bloodglucose"
 keyPassword "sinocare@ydy1"
 }
 }

 //混淆
 buildTypes{
 release{
 signingConfig signingConfigs.myConfig
 minifyEnabled false
 }
 }

 sourceSets {
 main {
 manifest.srcFile 'AndroidManifest.xml'
 }
 }
}
```

```

 java.srcDirs = ['src']
 resources.srcDirs = ['src']
 aidl.srcDirs = ['src']
 renderscript.srcDirs = ['src']
 res.srcDirs = ['res']
 assets.srcDirs = ['assets']
 }

 // Move the tests to tests/java, tests/res, etc...
 instrumentTest.setRoot('tests')

 // Move the build types to build-types/<type>
 // For instance, build-types/debug/java, build-types/debug/AndroidManifest.xml
 // This moves them out of their default location under src/<type>/... which would
 // conflict with src/ being used by the main source set.
 // Adding new build types or product flavors should be accompanied
 // by a similar customization.
 debug.setRoot('build-types/debug')
 release.setRoot('build-types/release')
}

apply plugin: "findbugs"

repositories {
 mavenCentral()
}

task findbugs(type: FindBugs) {
 //toolVersion = "2.0.1"
 ignoreFailures = true
 effort = "max"
 reportLevel = "low"
 classes = files("$project.buildDir/intermediates/classes")
 source = fileTree('build/intermediates/classes/debug/com/sn/')
 classpath = files()
 reports {
 xml {
 destination "build/findbugs.xml"
 }
 }
}

apply plugin: "checkstyle"

repositories {
 mavenCentral()
}
task checkstyle(type: Checkstyle) {
 ignoreFailures = true
 //config = files("build/config/checkstyle/checkstyle.xml")
}

```

```

source = fileTree('build/intermediates/classes/debug/com/sn/')
classpath = files()
reports {
 xml {
 destination "build/checkstyle-result.xml"
 }
}
}

apply plugin: "pmd"

repositories {
 mavenCentral()
}

task pmd(type: Pmd) {
 ignoreFailures = true
 source = fileTree('src/com/sn/')
 //ruleSetConfig = resources.file("${project.rootDir}/config/pmd/PmdRuleSets.xml")
 //ruleSetFiles = files("config/pmd/PmdRuleSets.xml")
 ruleSetFiles = files("${project.rootDir}/config/pmd/PmdRuleSets.xml")
 ruleSets = ["java-android"]
 reports {
 xml {
 destination "build/pmd.xml"
 }
 }
}
}

```

## 问题列表：

- 1、Can't find resource 'null' for rule 'android'

```

12:21:37.470 [LIFECYCLE] [class org.gradle.TaskExecutionLogger] :pmd FAILED
12:21:37.472 [INFO] [org.gradle.execution.taskgraph.AbstractTaskPlanExecutor] :pmd (Thread[main,5,main]) completed. Took 2.959 secs.
12:21:37.472 [DEBUG] [org.gradle.execution.taskgraph.AbstractTaskPlanExecutor] Task worker [Thread[main,5,main]] finished, busy: 2.959 secs, idle: 0.001 secs
12:21:37.484 [ERROR] [org.gradle.BuildExceptionReporter]
12:21:37.485 [ERROR] [org.gradle.BuildExceptionReporter] FAILURE: Build failed with an exception.
12:21:37.486 [ERROR] [org.gradle.BuildExceptionReporter]
12:21:37.487 [ERROR] [org.gradle.BuildExceptionReporter] * What went wrong:
12:21:37.487 [ERROR] [org.gradle.BuildExceptionReporter] Execution failed for task ':pmd'.
12:21:37.488 [ERROR] [org.gradle.BuildExceptionReporter] > Can't find resource 'null' for rule 'android'. Make sure the resource is a valid file or URL and is on
the CLASSPATH. Here's the current classpath: D:\CL\gradle-2.6\bin\lib\gradle-launcher-2.6.jar\UT\
12:21:37.488 [ERROR] [org.gradle.BuildExceptionReporter]
12:21:37.489 [ERROR] [org.gradle.BuildExceptionReporter] * Try:
12:21:37.489 [ERROR] [org.gradle.BuildExceptionReporter] Run with --stacktrace option to get the stack trace.
12:21:37.490 [LIFECYCLE] [org.gradle.BuildResultLogger]
12:21:37.490 [LIFECYCLE] [org.gradle.BuildResultLogger] BUILD FAILED
12:21:37.491 [LIFECYCLE] [org.gradle.BuildResultLogger]
12:21:37.491 [LIFECYCLE] [org.gradle.BuildResultLogger] Total time: 33.19 secs

```

解决：

The reason is simple. In case of Gradle 2.0 you must add language prefix before name of rule. You must use java-strings instead of plain strings.

```
apply plugin: "pmd"

repositories {
 mavenCentral()
}

task pmd(type: Pmd) {
 ignoreFailures = true
 source = fileTree('src/com/sn/')
 //ruleSetConfig = resources.file("${project.rootDir}/config/pmd/PmdRuleSets.xml")
 //ruleSetFiles = files("config/pmd/PmdRuleSets.xml")/
 ruleSetFiles = files("${project.rootDir}/config/pmd/PmdRuleSets.xml")
 ruleSets = ["java-android"]
}
reports {
}
xml {
 destination "build/pmd.xml"
}
}
```

ruleSets = ["android"]需要修改成  
ruleSets = ["java-android"]

# Jenkins+Gradle实现android开发持续集成问题汇总

来源：[测试蜗牛，一步一个脚印](#)

## 问题1：Android系统的环境变量不能被jenkins调用导致编译失败，需要在jenkins里面配置环境变量

报错信息：

What went wrong:

A problem occurred configuring root project 'BloodGlucose'.

SDK location not found. Define location with sdk.dir in the local.properties file or with an ANDROID\_HOME environment variable.



### 控制台输出

```
Started by user 黄卫华
Building on master in workspace D:\CI\jenkins-1.620\workspace\Build_HBGMS_BloodGlucose
No emails were triggered.
[Gradle] - Launching build.
[BloodGlucose] $ cmd.exe /C '"D:\CI\gradle-2.2.1\bin\gradle.bat clean build -b build.gradle && exit %%ERRORLEVEL%%'
FAILURE: Build failed with an exception.

* What went wrong:
A problem occurred configuring root project 'BloodGlucose'.
> SDK location not found. Define location with sdk.dir in the local.properties file or with an ANDROID_HOME environment variable.

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.

BUILD FAILED

Total time: 13.063 secs
Build step 'Invoke Gradle script' changed build result to FAILURE
Build step 'Invoke Gradle script' marked build as failure
Email was triggered for: Failure - Any
Sending email for trigger: Failure - Any
Request made to compress build log
Sending email to: huangweihua@sinocare.com
Finished: FAILURE
```

解决：在系统管理->环境变量里面新增一个全局环境变量



## 问题2：Gradle版本不对，导致编译失败

报错信息：

What went wrong:  
A problem occurred evaluating root project 'BloodGlucose'.  
Failed to apply plugin [id 'android']  
Gradle version 2.1 is required. Current version is 2.6. If using the gradle wrapper,  
try editing the distributionUrl in D:\CI\jenkins-  
1.620\workspace\Build\_HBGMS\_BloodGlucose\BloodGlucose\gradle\wrapper\gra  
dle-wrapper.properties to gradle-2.1-all.zip

```
Downloaded https://repo1.maven.org/maven2/org/xerial/kzml/kzml/2.3.0/kzml-2.3.0.jar
Download https://repo1.maven.org/maven2/org/ow2/asm/4.0/asm-4.0.jar
Download https://repo1.maven.org/maven2/com/android/tools/external/lombok/lombok-ast/0.2.2/lombok-ast-0.2.2.jar
Download https://repo1.maven.org/maven2/org/ow2/asm/asm-tree/4.0/asm-tree-4.0.jar
Download https://repo1.maven.org/maven2/org/apache/httpcomponents/httpcore/4.1/httpcore-4.1.jar
Download https://repo1.maven.org/maven2/commons-logging/commons-logging/1.1.1/commons-logging-1.1.1.jar
Download https://repo1.maven.org/maven2/commons-codec/commons-codec/1.4/commons-codec-1.4.jar

FAILURE: Build failed with an exception.

* Where:
Build file 'D:\CI\jenkins-1.620\workspace\Build_HBGMS_BloodGlucose\BloodGlucose\build.gradle' line: 9

* What went wrong:
! problem occurred evaluating root project 'BloodGlucose'.
> Failed to apply plugin [id 'android']
 > Gradle version 2.1 is required. Current version is 2.6. If using the gradle wrapper, try editing the distributionUrl in
D:\CI\jenkins-1.620\workspace\Build_HBGMS_BloodGlucose\BloodGlucose\gradle\wrapper\gradle-wrapper.properties to gradle-2.1-all.zip

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.

BUILD FAILED

Total time: 9 mins 10.824 secs
Build step 'Invoke Gradle script' changed build result to FAILURE
Build step 'Invoke Gradle script' marked build as failure
Email was triggered for: Failure - Any
Sending email for trigger: Failure - Any
Request made to compress build log
Sending email to: huangweihua@sinocare.com
Finished: FAILURE
```

解决：在服务器上安装2.2.1版本的gradle，并在jenkins里面配置gradle。

(1)在系统管理->环境变量里面新增一个全局环境变量



(2)在job里面选择gradle版本



### 问题3： build.gradle文件路径配置错误，导致失败

报错信息：

What went wrong:  
Build file 'D:\CI\jenkins-  
1.620\workspace\Build\_HBGMs\_BloodGlucose\build.gradle' does not exist.

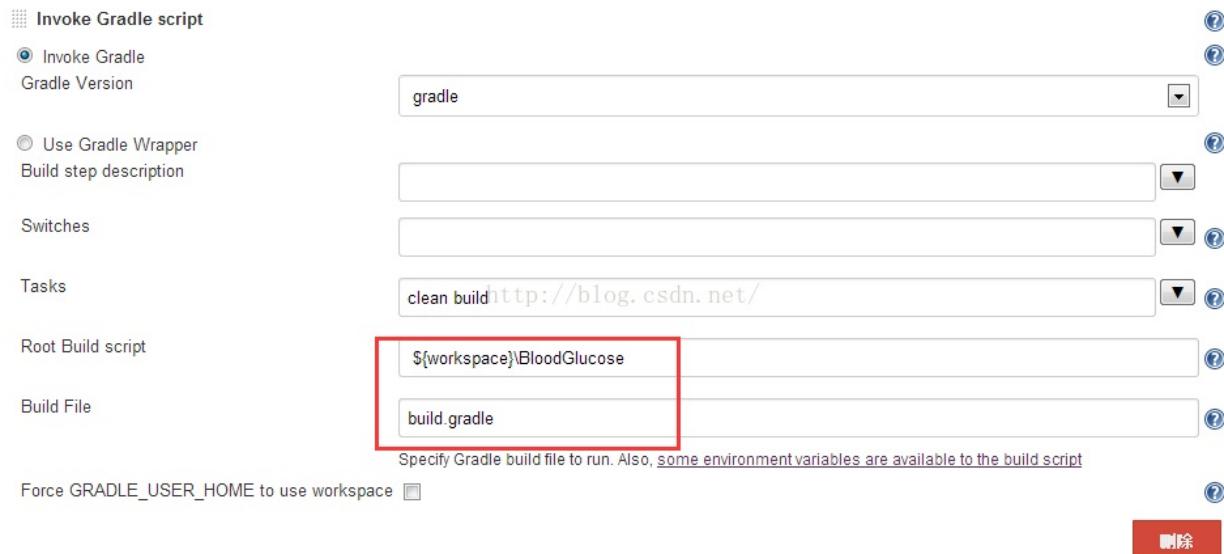
## 控制台输出

```

Started by user 黄卫华
Building on master in workspace D:\CI\jenkins-1.620\workspace\Build_HBGMS_BloodGlucose
Updating https://192.168.100.14:8443/svn/HBGMS/trunk/dev1.0/code/BloodGlucose at revision '2015-08-26T11:45:16.705 +0:
At revision 1032
no change for https://192.168.100.14:8443/svn/HBGMS/trunk/dev1.0/code/BloodGlucose since the previous build
No emails were triggered.
[Gradle] - Launching build.
[BloodGlucose] $ cmd.exe /C "gradle.bat clean build -b build.gradle && exit %%ERRORLEVEL%%"
The directory name is invalid
FATAL: command execution failed
java.io.IOException: Cannot run program "cmd.exe" (in directory "D:\CI\jenkins-1.620\workspace\Build_HBGMS_BloodGlucose\BloodGlucose\BloodGlucose"): CreateProcess error=267, 目录名称无效。
 at java.lang.ProcessBuilder.start(Unknown Source)
 at hudson.Proc$LocalProc.<init>(Proc.java:244)
 at hudson.Proc$LocalProc.<init>(Proc.java:216)
 at hudson.Launcher$LocalLauncher.launch(Launcher.java:816)
 at hudson.Launcher$ProcStarter.start(Launcher.java:382)
 at hudson.Launcher$ProcStarter.join(Launcher.java:389)
 at hudson.plugins.gradle.Gradle.performTask(Gradle.java:262)
 at hudson.plugins.gradle.Gradle.perform(Gradle.java:116)
 at hudson.tasks.BuildStepMonitor$1.perform(BuildStepMonitor.java:20)
 at hudson.model.AbstractBuild$AbstractBuildExecution.perform(AbstractBuild.java:779)
 at hudson.model.Build$BuildExecution.build(Build.java:205)
 at hudson.model.Build$BuildExecution.doRun(Build.java:162)
 at hudson.model.AbstractBuild$AbstractBuildExecution.run(AbstractBuild.java:537)
 at hudson.model.Run.execute(Run.java:1741)
 at hudson.model.FreeStyleBuild.run(FreeStyleBuild.java:43)
 at hudson.model.ResourceController.execute(ResourceController.java:98)
 at hudson.model.Executor.run(Executor.java:381)
Caused by: java.io.IOException: CreateProcess error=267, 目录名称无效。
 at java.lang.ProcessImpl.create(Native Method)
 at java.lang.ProcessImpl.<init>(Unknown Source)
 at java.lang.ProcessImpl.start(Unknown Source)

```

解决：根据实际情况配置build.gradle文件的路径及名称。



## 问题4：Sdk未安装导致编译失败，环境变量配置错误，导致编译失败

The screenshot shows the Jenkins interface with the 'Console Output' tab selected. The output log is displayed, highlighting an error message: 'Execution failed for task ':preBuild'. > failed to find target android-20 : D:\CI\ sdk\ tools'. This error is enclosed in a red box. The log also includes details about the build environment and failure triggers.

```
Started by user 黄卫华
Building on master in workspace D:\CI\jenkins-1.620\workspace\Build_HBGMs_BloodGlucose
No emails were triggered.
[Gradle] - Launching build.
[BloodGlucose] $ cmd.exe /C '"D:\CI\gradle-2.2.1\bin\gradle.bat clean build -b build.gradle && exit %ERRORLEVEL%"
:clean
:preBuild FAILED
FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':preBuild'.
> failed to find target android-20 : D:\CI\ sdk\ tools

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.

BUILD FAILED

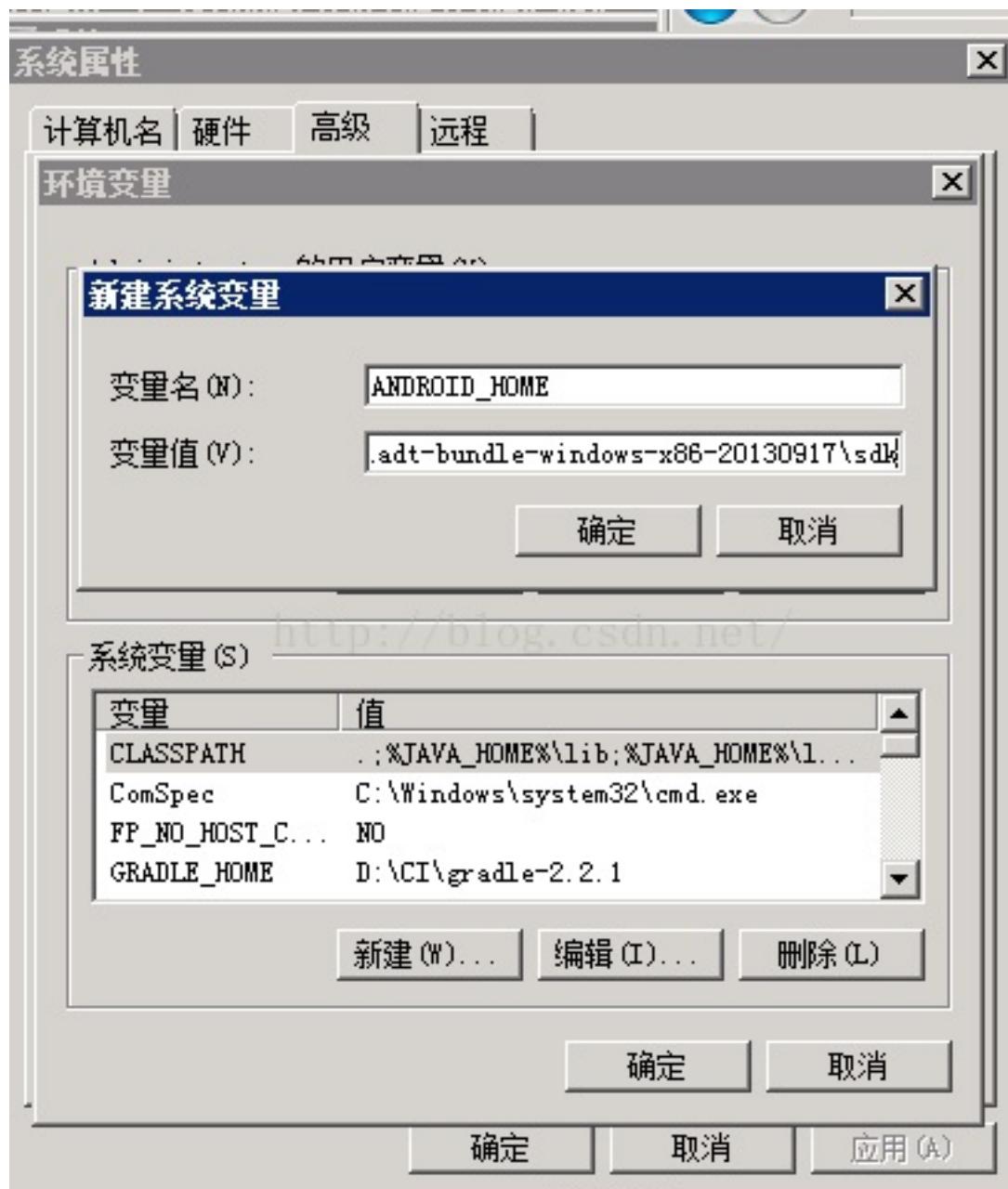
Total time: 17.202 secs
Build step 'Invoke Gradle script' changed build result to FAILURE
Build step 'Invoke Gradle script' marked build as failure
Email was triggered for: Failure - Any
Sending email for trigger: Failure - Any
Request made to compress build log
Sending email to: huangweihua@sinocare.com
Finished: FAILURE
```

## 解决：配置tools的环境变量

步骤1：下载并解压adt-bundle-windows-x86

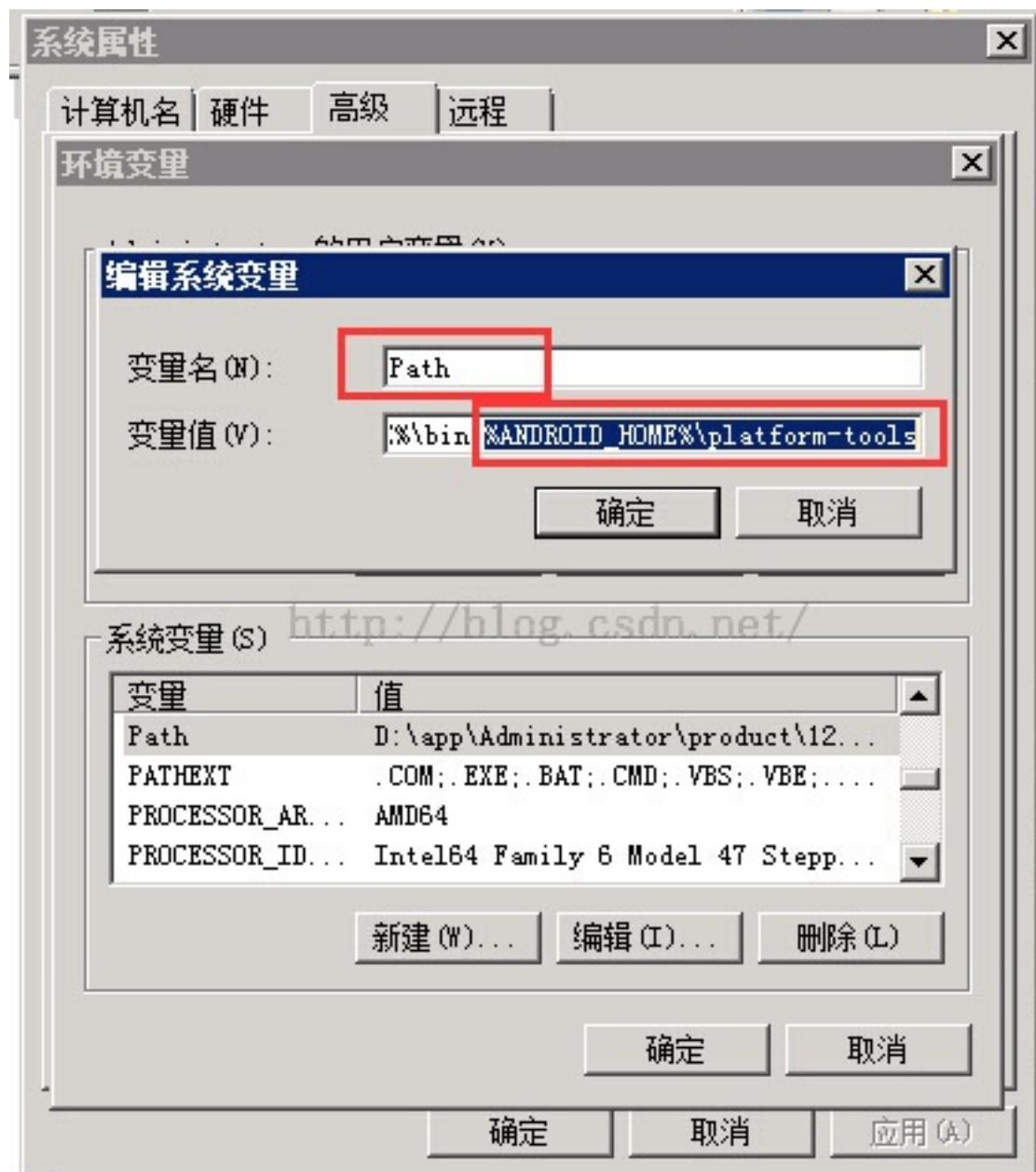
步骤2：设置环境变量

1、新增ANDROID\_HOME，值为sdk解压目录，如下图



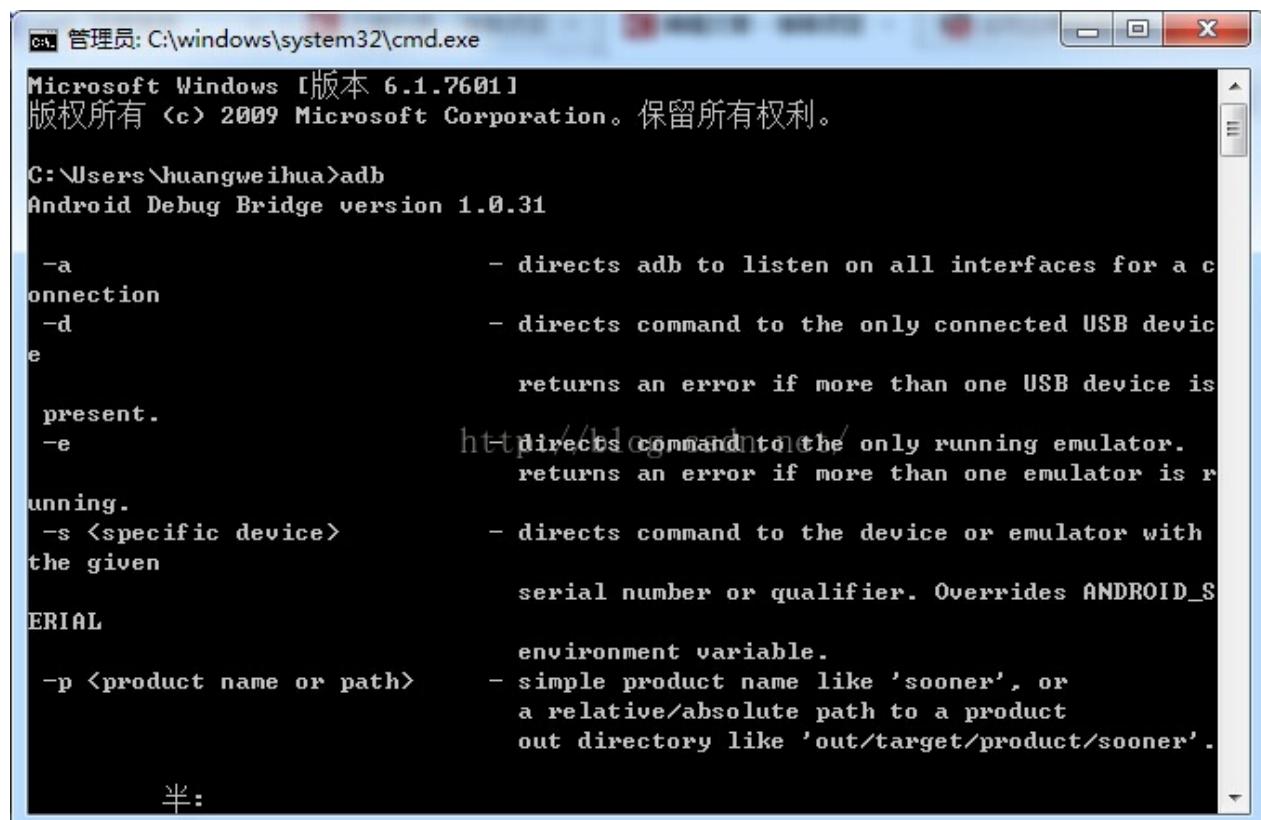
2、增加path路径，如下图

备注：2.2以上的sdk版本adb命令\工具在\platform-tools下：



### 环境变量的验证

在cmd命令里面输入“adb”，如下图



管理员: C:\windows\system32\cmd.exe

Microsoft Windows [版本 6.1.7601]  
版权所有 © 2009 Microsoft Corporation。保留所有权利。

```
C:\Users\huangweihua>adb
Android Debug Bridge version 1.0.31

-a - directs adb to listen on all interfaces for a connection
-d - directs command to the only connected USB device
-e - returns an error if more than one USB device is present.
-e http://[host]:[port]/
 directs command to the only running emulator.
 returns an error if more than one emulator is running.
-s <specific device> - directs command to the device or emulator with the given serial number or qualifier. Overrides ANDROID_SERIAL environment variable.
-p <product name or path> - simple product name like 'sooner', or a relative/absolute path to a product out directory like 'out/target/product/sooner'.
```

半:

# Jenkins中Console Output中文乱码问题

来源:测试蜗牛,一步一个脚印

### 解决方案：

在 `.jenkins/jenkins.xml` 中新增 `-Dfile.encoding=utf-8`，可解决jenkins信息乱码问题，如下：

```
-Xrs -Xmx256m -Dfile.encoding=utf-8 -
Dudson.lifecycle=hudson.lifecycle.WindowsServiceLifecycle -jar
"%BASE%\jenkins.war" --httpPort=8080
```

添加前的乱码，如下图：

```
:checkReleaseManifest
:prepareReleaseDependencies
:compileReleaseAidl
:compileReleaseRenderscript
:generateReleaseBuildConfig
:generateReleaseAssets UP-TO-DATE
:mergeReleaseAssets
:generateReleaseResValues UP-TO-DATE
:generateReleaseResources
:mergeReleaseResources
:processReleaseManifest
:processReleaseResources
:generateReleaseSources
:compileReleaseJava
:compileReleaseJavaWithJavac -Xlint:deprecation
:compileReleaseJavaWithJavac -Xlint:unchecked
```

添加后就无乱码了，如下图：

```
:preBuild
:compileDebugNdk
:preDebugBuild
:checkDebugManifest
:prepareDebugDependencies
:compileDebugAidl
:compileDebugRenderscript
:generateDebugBuildConfig
:generateDebugAssets UP-TO-DATE
:mergeDebugAssets
:generateDebugResValues
:generateDebugResources
:mergeDebugResources
:processDebugManifest
:processDebugResources
:generateDebugSources
:compileDebugJava注: 某些输入文件使用或覆盖了已过时的 API。
注: 有关详细信息, 请使用 -Xlint:deprecation 重新编译。
注: 某些输入文件使用了未经检查或不安全的操作。
注: 有关详细信息, 请使用 -Xlint:unchecked 重新编译。
```

# Jenkins中Jelly邮件模板的配置

来源:测试蜗牛, 一步一个脚印

1、找到email-ext.jar并解压, 路径为: D:\CI\jenkins-1.620\plugins\email-ext\WEB-INF\lib



2、在jenkins里的default content 里面设置\${JELLY\_SCRIPT,template="html"}, 或者 \${JELLY\_SCRIPT,template="text"}, 如下图

增加构建步骤 ▾

构建后操作

**Editable Email Notification**

Disable Extended Email Publisher

Project Recipient List: \$DEFAULT\_RECIPIENTS

Project Reply-To List: \$DEFAULT\_REPLYTO

Content Type: Default Content Type

Default Subject: \$DEFAULT SUBJECT

Default Content: \${JELLY\_SCRIPT,template='text'}

保存 应用

3、执行job并查看邮件

主题: test - Build # 20

# BUILD SUCCESS

Build URL <http://192.168.100.15:8080/job/test/20/>  
Project: test  
Date of build: Tue, 18 Aug 2015 14:42:36 +0800  
Build duration: 0.49 秒

## CHANGES

No Changes

# Jenkins 使用学习笔记

来源:[cnblogs](#)

[TOC]

## 一、 Jenkins安装

### 1. Jenkins下载

由于Jenkins是开源的，可以直接下载其源代码自行编译，也可以下载发布好的文件，下载地址为：<http://mirrors.jenkins-ci.org/war/latest/jenkins.war>

### 2.Jenkins 安装

Jenkins是用Java语言开发的系统，首先要确定服务器上已经安装JDK或者JRE。

- 安装方式一

直接运行 `java -jar Jenkins.war`，在浏览器中输入 <http://localhost:8080>即可。

- 安装方式二

安装Tomcat。

添加环境变量**JENKINS\_HOME**，该变量为jenkins系统的工作目录，如下图：



(注意：该目录对于jenkins极为重要，系统所有相关的配置、数据文件等都存放于此，所以一定要确保该目录有足够的空间）。

修改Tomcat 程序conf/server.xml文件，在Host结点下添加<Context path="/jenkins" docBase="G:\WWWRoot\jenkins\jenkins.war" reloadable="true" />，其中path表示jenkins系统的访问跨径，docBase表示jenkins程序文件所在位置。

启动Tomcat，输入<http://localhost:8080/jenkins>打开系统，如下图，（如能正常打开系统，表示系统安装成功）



### 3. Jenkins 目录结构

Jenkins 所有数据存放在JENKINS\_HOME所设置的目录下，如果没设置此变量，数据将会保存在我的文档/jenkins目录下。目录结构如下：

```

JENKINS_HOME
+- config.xml (配置文件)
+- *.xml (其它配置文件)
+- userContent (用户授权文件)
+- plugins (插件)
+- jobs
 +- [JOBNAME] (存放所有的Job的文件)
 +- config.xml (job 配置文件)
 +- workspace (存放系统代码)

```

由于Jenkins没有数据库存，所以备份、删除、移动数据非常方便。备份是要备份**JENKINS\_HOME**目录即可，恢复时要先停止jenkins。对于移动或删除jobs，只需要简单地移动或删除%**JENKINS\_HOME%**\jobs目录。对于修改jobs的名字，只需要简单地修

改`%JENKINS_HOME%\jobs`下对应job的文件夹的名字。对于不经常使用的job，只需要对`%JENKINS_HOME%\jobs`下对应的jobs的目录zip或tar后存储到其他的地方。

## 4.Jenkins系统管理

在使用系统之前，还需要对系统进行一些基本的配置，如下图：

## 5.系统配置

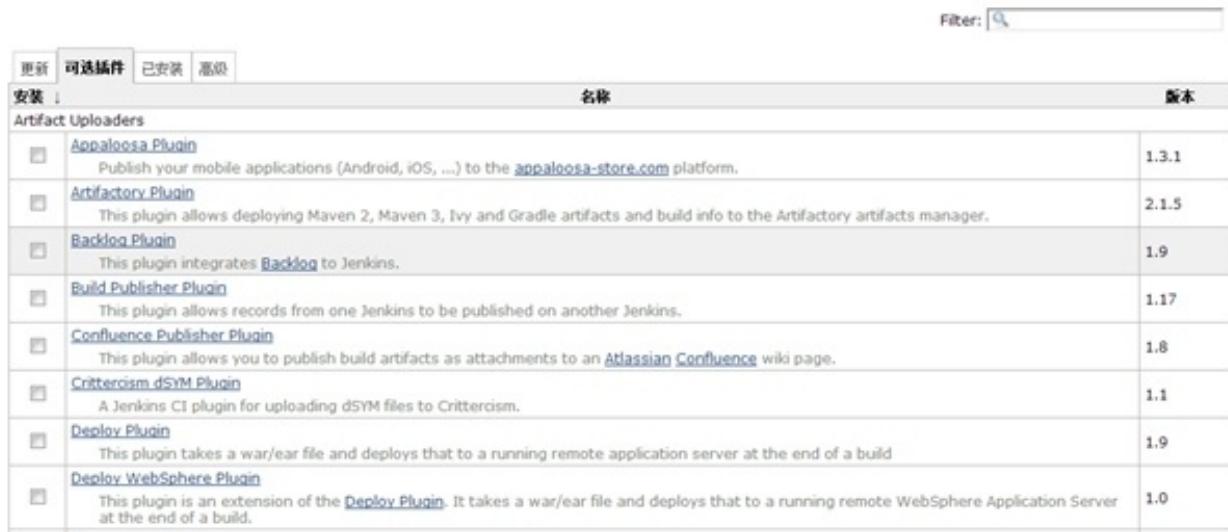
系统配置是Jenkins配置中非常重要的一个页面，如下图。

在系统配置页面，可以配置JDK、MSBuild、源代码控制等。几乎所有的插件安装后有需要配置的都会在这里呈现。

## 6.插件管理

Jenkins是一个可扩展的系统，其很多功能都得益于各式各样的插件，现Jenkins中已有超过200个各种不同功能的插件。所以插件对于Jenkins是非常重要的。

Jenkins的插件安装可离线安装，也可在线安装。打开插件管理页面，如下图：



| 安装                       | 名称                                                                                                                                                                                                                             | 版本    |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| <input type="checkbox"/> | <a href="#">Appaloosa Plugin</a><br>Publish your mobile applications (Android, iOS, ...) to the <a href="#">appaloosa-store.com</a> platform.                                                                                  | 1.3.1 |
| <input type="checkbox"/> | <a href="#">Artifactory Plugin</a><br>This plugin allows deploying Maven 2, Maven 3, Ivy and Gradle artifacts and build info to the Artifactory artifacts manager.                                                             | 2.1.5 |
| <input type="checkbox"/> | <a href="#">Backlog Plugin</a><br>This plugin integrates <a href="#">Backlog</a> to Jenkins.                                                                                                                                   | 1.9   |
| <input type="checkbox"/> | <a href="#">Build Publisher Plugin</a><br>This plugin allows records from one Jenkins to be published on another Jenkins.                                                                                                      | 1.17  |
| <input type="checkbox"/> | <a href="#">Confluence Publisher Plugin</a><br>This plugin allows you to publish build artifacts as attachments to an <a href="#">Atlassian Confluence</a> wiki page.                                                          | 1.8   |
| <input type="checkbox"/> | <a href="#">Crittercism dSYM Plugin</a><br>A Jenkins CI plugin for uploading dSYM files to Crittercism.                                                                                                                        | 1.1   |
| <input type="checkbox"/> | <a href="#">Deploy Plugin</a><br>This plugin takes a war/ear file and deploys that to a running remote application server at the end of a build                                                                                | 1.9   |
| <input type="checkbox"/> | <a href="#">Deploy WebSphere Plugin</a><br>This plugin is an extension of the <a href="#">Deploy Plugin</a> . It takes a war/ear file and deploys that to a running remote WebSphere Application Server at the end of a build. | 1.0   |

在这里可轻松的管理插件。

注意：插件安装完成后，一般需要重启系统，还需要到系统配置页面对插件进行配置，插件才能起作用。

## 二、创建一个任务

### 1. 安装MSBuild Plugin插件

在创建一个任务之前，需要添加一些必须的插件。由于例子是C#写的，首先需安装MSBuild插件



安装完MSBuild Plugin插件后，还需在系统配置里进行配置。进入系统配置目录，找到刚安装的插件



点击MSBuild安装，添加MSBuild的配置，如下图，分别添加了.NET FrameWork 4.0 32位和64位MSBuild.exe，在创建任务选择用MSBuild构建时，将同时用32、64位的MSBuild.exe去编译。



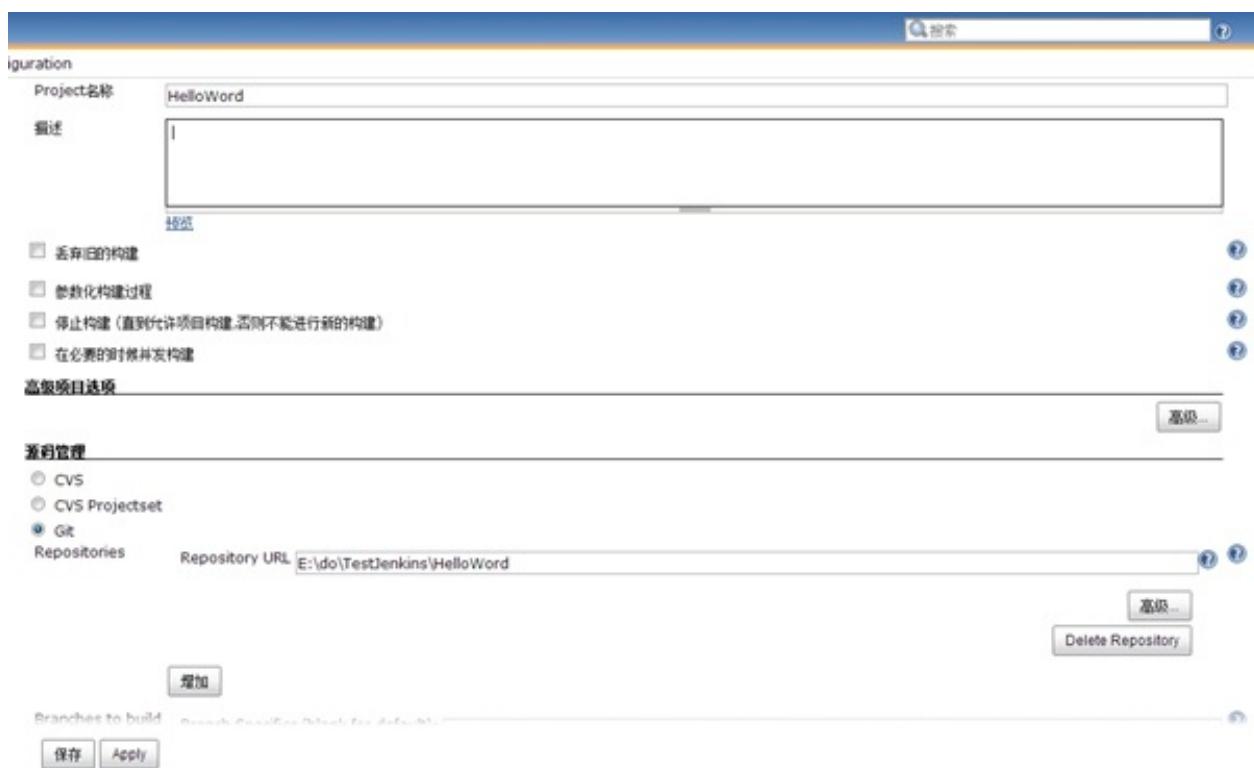
Ant

红色框中是MSBuild.exe的路径。

## 2. 创建新任务

点击新建Job，输入任务名称，如下图，选择构建一个自由风格的软件项目，点击确定按钮，进入Job详细配置界面。





为了能完成一次基本的构建，至少需要做如下设置，在源码管理选项，选择源代码管理工具。



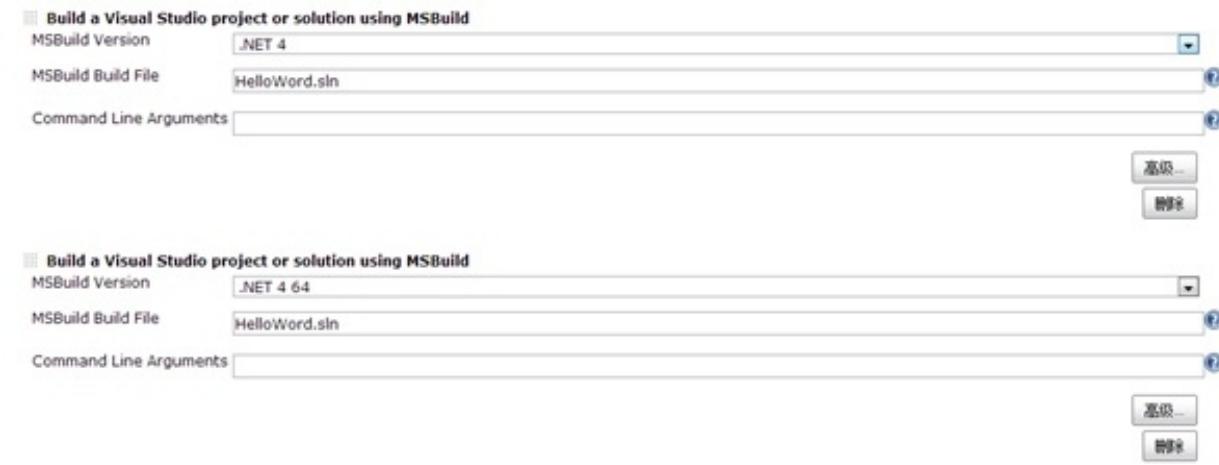
例子中选择Git作为源代码管理工具，还需安装 Git Plugin和Git Client Plugin作为插件。系统自带SVN插件，若用SVN做源代码管理，则无需再去安装插件。

选择了代码管理工具，还关键的一步，添加构建。



点击增加构建，选择前面安装的MSBuild，将构建文件设置为项目文件，如下图：

拾遺



完成这些基本配置后，点击保存按钮，保存配置，返回任务界面，点击立即构建，Jenkins将会自动重源代码服务获取代码，并编译，并显示构建状态。



红色图标表示构建失败，绿色图标表示构建成功。可以点击控制台输出查看详细的构建信息。

The screenshot shows the Jenkins control console output for a build step. The log starts with the command 'git clone' failing to determine its version. It then attempts to clone the 'origin' repository, which fails due to a class loading exception for 'org.jenkins.plugins.gitclient.CliGitAPIImpl\$1\$1'. This is traced back to a 'ResourceController\$execute' method. The log ends with a Java exception message about a missing 'git.exe' file.

## 三、自动测试

### 1. 相关插件安装

对于C#语言，在自动化测试中，可以用到如下插件：

- Jenkins MSTestRunner plugin：系统使用MSTest来测试；
- Jenkins NUnit plugin：系统使用NUnit来测试；
- JENKINS MSTest plugin：来发布MSTest的测试结果。

本例以MSTestRunner和MSTest plugin来作为例子来说明。首先到系统的插件管理界面安装这两个插件，如下图



安装完这两个插件后，重启Jenkins。打开系统配置界面，还需对MSTest进行配置。

红色框是MSTest.exe的路径位置，该工具为VS自带工具。



- 配置任务

安装完前面的插件后，通过配置，就可以让Jenkins在构建的时候自动运行测试用例。

打开任务的配置界面，在构建项，选择添加构建，运行单元测试。

The screenshot shows the Jenkins job configuration interface under the "增加构建步骤" (Add build step) dropdown. It lists several options:
 

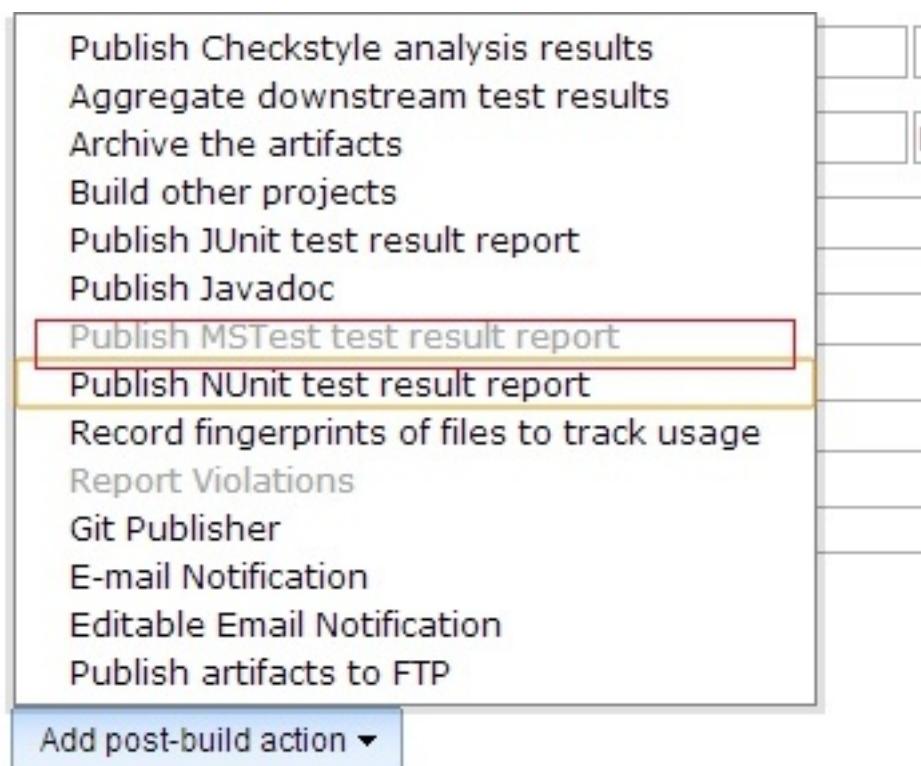
- Build a Visual Studio project or solution using MSBuild
- Execute Windows batch command
- Execute shell
- FxCop exec.
- Invoke Ant
- Invoke top-level Maven targets
- Run unit tests with MSTest** (highlighted with a blue bar)

添加相关配置，测试文件为我们测试项目所生成的dll或者exe文件。测试结果文件为MSTest.exe所产生的测试结果文件，文件后缀为trx。注意：这里文件是相对路径，相对于项目的路径。



完成了这一步的配置，系统在构建的时候，就能自动运行测试用例，但系统还不能将测试结果展示出来，要展示结果结果，还需做如下配置。

在任务配置的构建后的操作里，选择添加后构建后的操作，如下图。



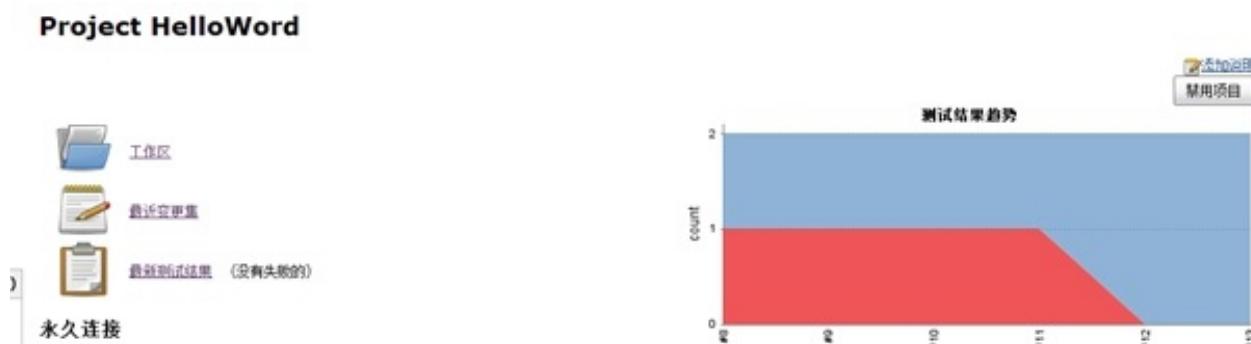
选择发布MSTest 测试结果报告。



在测试报告文件中添加前面配置MSTest.exe生的结果文件。

通过这些基本的配置，系统就能在构建的时候运行测试用例，并将测试结果展示出来。

在每个任务的主页，有测试结果的统计图。



对于每次的测试，都有很详细的信息展示。要查看某一次构建的测试结果，可以点击构建历史中的测试结果。



The screenshot shows the Jenkins Test Results page for a build named 'HelloWord #11'. The main header indicates 'Test Result' with '3 failures (e0)'. A large red bar at the top right shows '2 tests (e0)' and 'Took 43毫秒'. Below this, under 'All Failed Tests', there is a table with one row: 'Test Name' (UnitTestHelloWord.UnitTest1.TestMethod2), 'Duration' (37毫秒), and 'Age' (1天). Under 'All Tests', another table shows 'Package' (UnitTestHelloWord), 'Duration' (43毫秒), 'Fail' (1), 'Skip' (0), 'Total' (2), and '(diff)' (0). At the bottom right, it says '页面生成于: 2013-5-22 23:01:30 REST API Jenkins ver. 1.515'.

在测试结果页面中，展示的测试结果的总体详细，点击某个测试名称，还能显示该测试的详细信息。

The screenshot shows the Jenkins Failed test details page for 'UnitTestHelloWord.UnitTest1.TestMethod2 (from MSTestSuite)'. The title is 'Failed' with a red background. It states 'Failing for the past 4 builds (Since 1天 ago) Took 37毫秒'. Below this, there is a 'Stacktrace' section with a message about an Assert.AreEqual failure and a stack trace pointing to 'UnitTestHelloWord.UnitTest1.TestMethod2()' in 'C:\Windows\Temp\jenkins\home\jobs\HelloWorld\workspace\HelloWorld\bin\TestMethod2.cs' at line 20.

## 2. 代码质量管理

### 相关插件安装

对于C#语言，代码质量管理可用到如下插件：

- Jenkins FxCop Runner plugin：该插件就是在构建的时候调用FxCop来分析代码。
- Static Analysis Utilities
- Jenkins Violations plugin：该插件主要用来展示各种分析工具所产生的结果。

在安装完Jenkins FxCop Runner plugin后，还需对其进行配置。点击系统管理，系统设置，如下图。



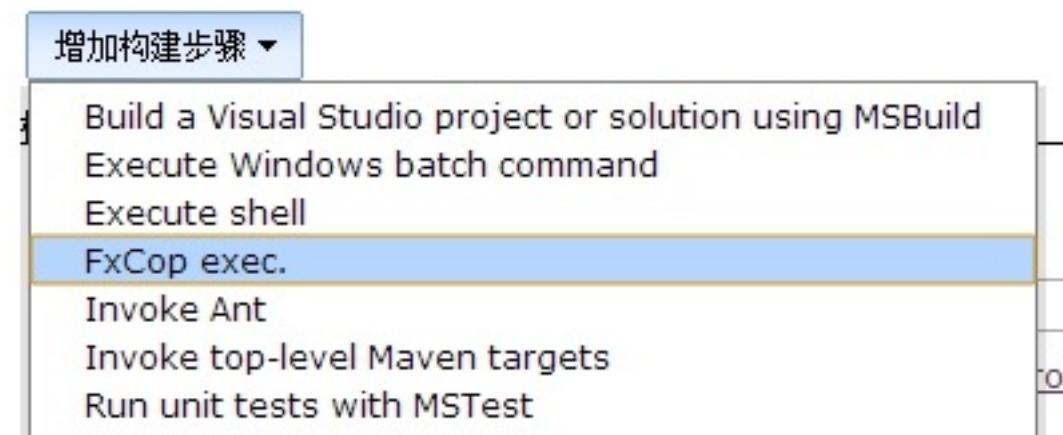
点击FxCop安装，设置FxCop程序的路径。



## 使用FxCop分析程序集

FxCop是一个代码分析工具，它依照微软.NET框架的设计规范对托管代码assembly进行检查。它使用基于规则的引擎，来检查出你代码中不合规范的部分；也可以定制自己的规则加入到这个引擎。

为了能让FxCop在构建系统时运行，还需在任务中添加配置。在构建项目，选择增加构建步骤，选择FxCop exec。



添加配置信息。程序集名称为我们需要分析的项目所生成的程序集。输入xml文件为分析所产生的结果文件，为xml格式文件。

|                            |                                                       |
|----------------------------|-------------------------------------------------------|
| FxCop Name                 | FxCop2012                                             |
| Assembly Files             | HelloWord\bin\Debug\HelloWord.exe                     |
| Output XML                 | HelloWord\bin\Debug\HelloWord.exe.CodeAnalysisLog.xml |
| RuleSet File               |                                                       |
| IgnoreGeneratedCode        | <input checked="" type="checkbox"/>                   |
| ForceOutput                | <input checked="" type="checkbox"/>                   |
| Command Line Arguments     |                                                       |
| Fail build on test failure | <input checked="" type="checkbox"/>                   |

完成好这些配置后，系统在构建的时候将会调用配置的工具进行代码分析。

## 使用StyleCop审查代码

StyleCop的终极目标是让所有人都能写出优雅和一致的代码，因此这些代码具有很高的可读性。

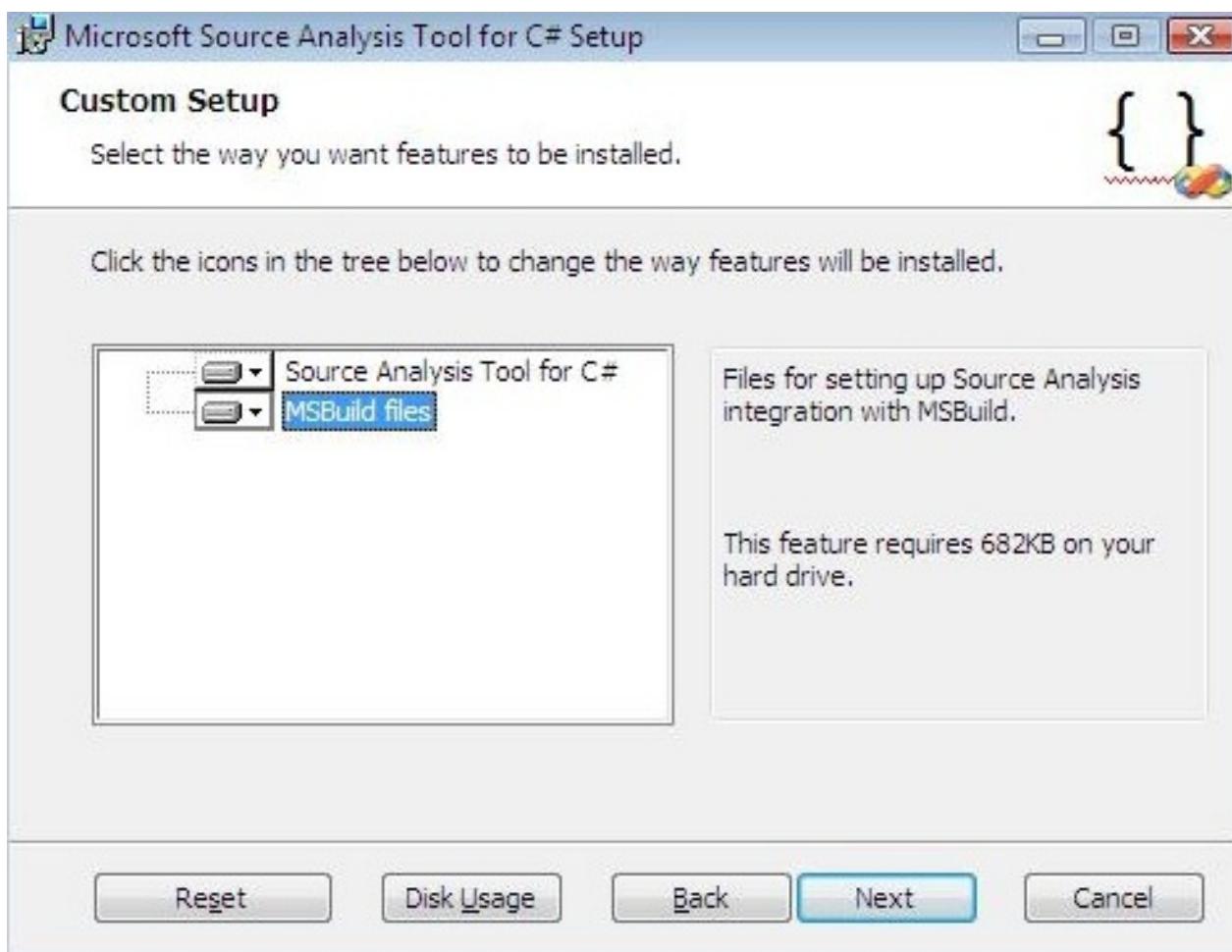
StyleCop不是代码格式化（代码美化）工具，而是代码规范检查工具（Code Review 工具），它不仅仅检查代码格式，而是编码规范，包括命名和注释等。

StyleCop目的是帮助项目团队执行一系列常用的源代码格式规范，这些规范是关于如何开发布局规整，易读，易维护并且文档良好的优雅代码的。

StyleCop现在包含了 200 个左右的最佳实践规则（best practice rules），这些规则与 Visual Studio 2005 和 Visual Studio 2008 中默认的代码格式化规则是一致的。

- StyleCop安装

StyleCop是微软的开源项目，可到<http://stylecop.codeplex.com/>下载最新的安装包。在安装该工具的时候，尽量默认安装，并且MSBuild files一定要选上，不然有可能导致不能正常使用。



为了能让系统在构建时自动运行该工具，需编辑一下项目文件，添加如下配置：

```
<Project DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

...Contents Removed...

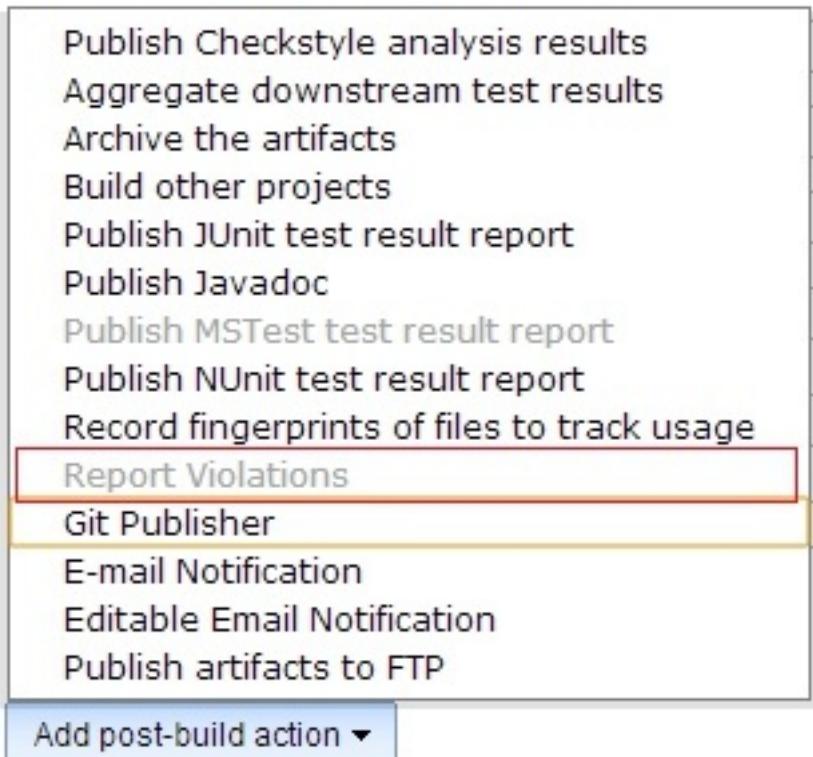
<Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
<Import Project="$(ProgramFiles)\MSBuild\StyleCop\v4.x\StyleCop.targets" />

...Contents Removed...

</Project>
```

- 展示分析结果

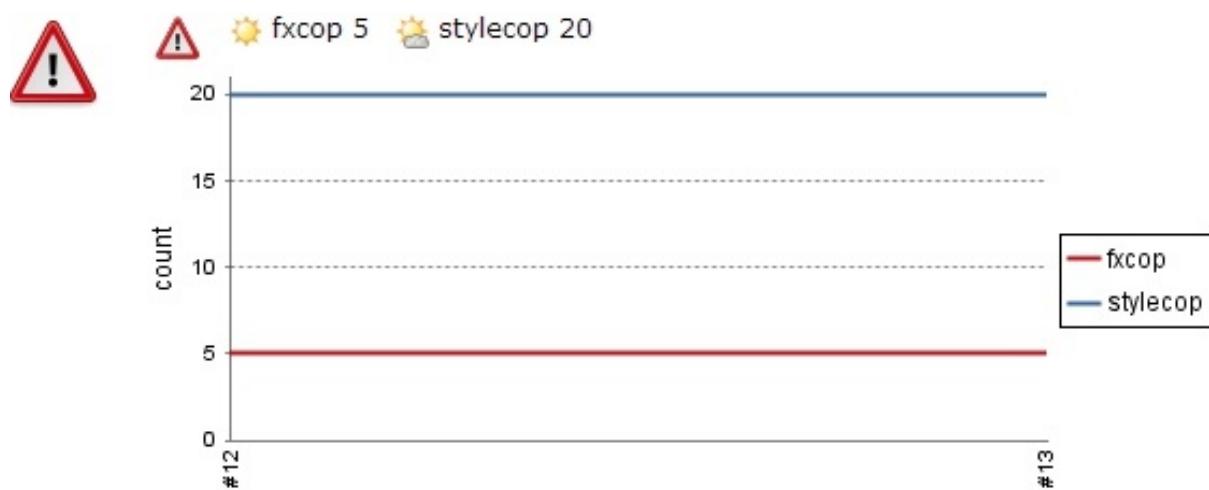
在任务配置页面，构建后操作，增加构建后操作，选择Report Violations。



在Report Violations配置页面，设置要展示的文件。我这里设置FxCop、StyleCop分析所产生的文件。值得注意的是，StyleCop所产生的文件是在obj\Debug目录下。

| Report Violations |     |     |                      |
|-------------------|-----|-----|----------------------|
|                   | 违例数 | 严重性 | XML filename pattern |
| checkstyle        | 10  | 999 | 999                  |
| codenarc          | 10  | 999 | 999                  |
| cpd               | 10  | 999 | 999                  |
| cpplint           | 10  | 999 | 999                  |
| csslint           | 10  | 999 | 999                  |
| findbugs          | 10  | 999 | 999                  |
| fxcop             | 10  | 999 | 999                  |
| gendarme          | 10  | 999 | 999                  |
| jcreport          | 10  | 999 | 999                  |
| jslint            | 10  | 999 | 999                  |
| pep8              | 10  | 999 | 999                  |
| perlcritic        | 10  | 999 | 999                  |
| pmd               | 10  | 999 | 999                  |
| pylint            | 10  | 999 | 999                  |
| simian            | 10  | 999 | 999                  |
| stylecop          | 10  | 999 | 999                  |

完成这些配置后，重新构建任务。任务构建的主页会多出配置的分析报告。



点击某一项，还可以进入该项的详细分析报表。

**Violations Report /jenkins/job/HelloWord/13 for build 13**

| Type     | Violations | Files in violation |
|----------|------------|--------------------|
| fxcop    | 5          | 3                  |
| stylecop | 20         | 3                  |

**fxcop**

filename: HelloWord\Program.cs, helloworld.exe, HelloWord\Class1.cs

| level  | l | m | h | number |
|--------|---|---|---|--------|
| Low    | 0 | 2 | 0 | 2      |
| Medium | 0 | 1 | 1 | 2      |
| High   | 0 | 1 | 0 | 1      |

**stylecop**

## 四、用户权限管理

### 1. 使用系统自带的授权机制

默认地Jenkins不包含任何的安全检查，任何人可以修改Jenkins设置、任务和启动构建等。显然地在大规模的公司需要多个部门一起协调工作的时候，没有任何安全检查会带来很多的问题。我们可以通过以下2方面来增强Jenkins的安全：

- Security Realm, 用来决定用户名和密码，且指定用户属于哪个组；
- Authorization Strategy, 用来决定用户对那些资源有访问权限；

在系统管理，Configure Global Security页面中启用权限管理。

**Configure Global Security**

全局权限

安全域

授权策略

全局权限矩阵

首先要启用安全，在安全域中选择Jenkins专有用户数据库，并选择允许用户注册。在授权策略选项中选择项目矩阵授权策略。设置好这些基本属性后，就是添加用户。



输入用户名，点击添加系统就添加了用户，但需注意，在这里只添加了用户名，添加的用户还不能登录，还需返回系统注册页面，注册一个与此设置的相同用户名的用户。如下图：



## 2. Role-based Authorization Strategy授权机制

打开系统管理，插件管理页面，安装Role-based Authorization Strategy插件。



进入系统管理，系统设置，如下图所示：“安全域”选择使用Jenkins专有用户数据库，可以在初始化的时候勾选“允许用户注册”；“授权策略”选择使用“Role-Based Strategy”。



配置完成后在“系统管理”下新增选项“Manage and Assign Roles”。点击“管理用户”新建账户后即可进行账户、群组的安全策略配置。



点击Manage and Assign Roles，进入如下界面，页面有添加角色，给角色授权功能。

Jenkins > Manage and Assign Roles

**Manage and Assign Roles**

**Manage Roles** Manage Roles

**Assign Roles** Assign Roles

点击添加角色，进入角色管理页面。在这里有两种角色，一种是全局角色，一种是项目角色，一般用户需授这两种角色。

| Role           | Overall                  |                          |                          |                          |                          |                          |                          |                          |                          |                          | Slave                    |                          | Job                      |                          | Run                      |                          | View                     |                          |                          |                          |                          |                          |           |
|----------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|-----------|
|                | Administrator            | Read                     | RunScripts               | UploadPlugins            | ConfigureUpdateCenter    | Configure                | Delete                   | Create                   | Disconnect               | Connect                  | Create                   | Delete                   | Configure                | Read                     | Discover                 | Build                    | Workspace                | Cancel                   | Delete                   | Update                   | Create                   | Delete                   | Configure |
| ProjectManager | <input type="checkbox"/> |           |
| admin          | <input type="checkbox"/> |           |

点击Assing Roles，界面用户授权界面。注意：这里添加用户的方式与前面介绍的相同。

## 五、构建通知

### 1. 通过邮件通知

进行系统管理，系统设置页面，设置邮件服务器。

|                                              |                |
|----------------------------------------------|----------------|
| SMTP服务器                                      | smtp.gmail.com |
| 用户默认邮件地址                                     | @gmail.com     |
| <input checked="" type="checkbox"/> 使用SMTP认证 |                |
| 用户名                                          | xiaodao0007    |
| 密码                                           | *****          |
| <input type="checkbox"/> 使用SSL协议             |                |
| SMTP端口                                       | 465            |
| Reply-To Address                             |                |
| 字符集                                          | UTF-8          |
| <input type="checkbox"/> 通过发送测试邮件测试配置        |                |

配置完邮件服务器后，进行任务配置页面，配置邮件通知。

|                                                                                                                                                                               |                      |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------|
| <input type="checkbox"/> E-mail Notification                                                                                                                                  |                      |
| Recipient                                                                                                                                                                     | mottolog@foxmail.com |
| Whitespace-separated list of recipient addresses. May reference build parameters like \$EMAIL. E-mail will be sent when a build fails, becomes unstable or returns to stable. |                      |
| <input type="checkbox"/>                                                                                                                                                      | 每次不稳定的构建都发送邮件通知      |
| <input type="checkbox"/>                                                                                                                                                      | 单独发送邮件给对构建或不影响的负责人   |
| <input type="button" value="取消"/>                                                                                                                                             |                      |

当完成一次构建后，系统将发送详细的构建信息，如下图。

**Build failed in Jenkins: HelloWord1 #22**

附件人: [xiaodao gmail](#) <xiaodao0007@gmail.com>

时间: 2013年5月23日(星期四)中午12:52

收件人: [mottolog](#) <mottolog@foxmail.com>

这不是腾讯公司的官方邮件。请勿轻信虚假、汇款、中奖信息，与轻易拨打陌生电话。如何识别腾讯公司邮件 | [举报垃圾邮件](#)

See <<http://localhost:8080/jenkins/job/HelloWord1/22/>>

---

```
Started by user admin
Building in workspace <http://localhost:8080/jenkins/job/HelloWord1/ws/>
Checkout:workspace / <http://localhost:8080/jenkins/job/HelloWord1/ws/> - hudson.remoting.LocalChannel@1bd1ce2
Using strategy: Default
Last Built Revision: Revision 457413cef3136f4c6446a52d79b542842ff309 (origin/master, origin/HEAD)
Fetching changes from 1 remote Git repository
Fetching upstream changes from origin
Seen branch in repository origin/HEAD
Seen branch in repository origin/master
Seen 2 remote branches
Commencing build of Revision 457413cef3136f4c6446a52d79b542842ff309 (origin/master, origin/HEAD)
Checking out Revision 457413cef3136f4c6446a52d79b542842ff309 (origin/master, origin/HEAD)
Warning : There are multiple branch changesets here
Path To MSBuild.exe: C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe
Executing the command cmd.exe /C C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe HelloWord1.sln && exit %%ERRORLEVEL%% from <http://localhost:8080/jenkins/job/HelloWord1/ws/>
[workspace] $ cmd.exe /C C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe HelloWord1.sln && exit %%ERRORLEVEL%%
Microsoft(R) 生成引擎版本 4.0.30319.17929
[Minvndfr] NFT Framework 版本 4.0.30319.17929
```

## 2. 通过实时消息通知

实时消息通知有两种方式，一种是通过Jabber，另一种是通过IRC的方式通知。例子通过Jabber的方式来说明。

Jabber 是著名的Linux即时通讯服务服务器，它是一个自由开源软件，能让用户自己架即时通讯服务器，可以在Internet上应用，也可以在局域网中应用。Jabber最有优势的就是其通信协议，可以和多种即时通讯对接。

- 安装Jabber实时通讯服务器

到<http://www.igniterealtime.org>下载Openfire和Spark。其中Openfire为服务器，Spark为客户端。

安装好Openfire服务器后，启动相关服务，在其管理界面添加用户。这里新建用户jenkins作为Jenkins系统的账户，如下图。

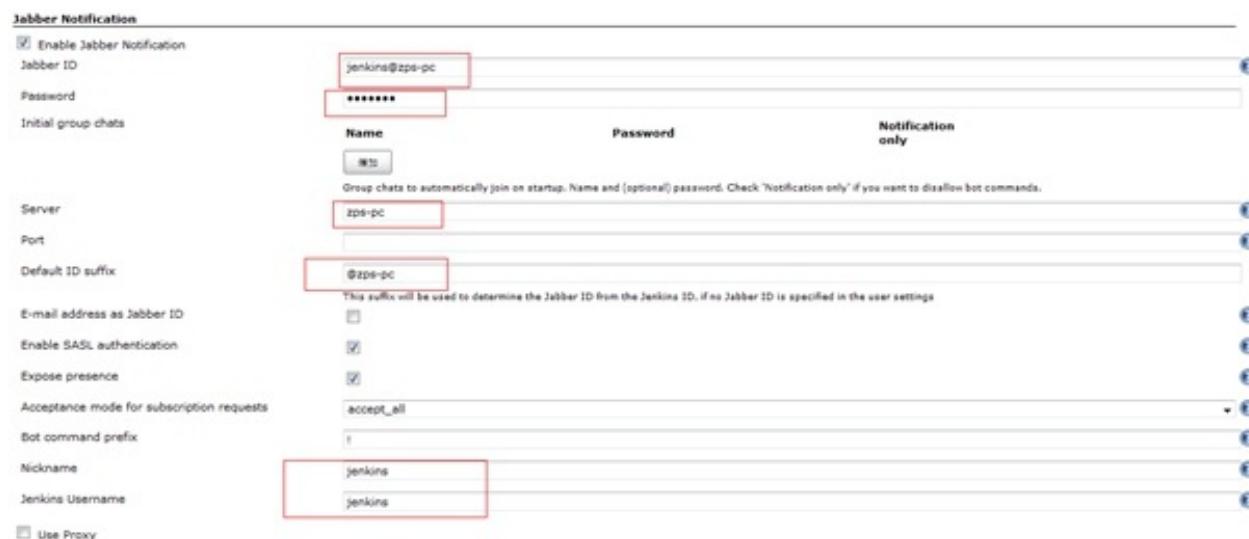
用户摘要

| 用户总数: 4 – 按用户名排序 – 用户/页: 100 |                         |               |           |        |    |
|------------------------------|-------------------------|---------------|-----------|--------|----|
| 在线                           | 用户名                     | 名称            | 已创建       | 最近一次退出 | 编辑 |
| 1                            | <a href="#">admin</a> ★ | Administrator | 2013-5-21 |        |    |
| 2                            | <a href="#">hehe</a>    |               | 2013-5-21 |        |    |
| 3                            | <a href="#">jenkins</a> |               | 2013-5-21 |        |    |
| 4                            | <a href="#">xiaodao</a> |               | 2013-5-21 |        |    |

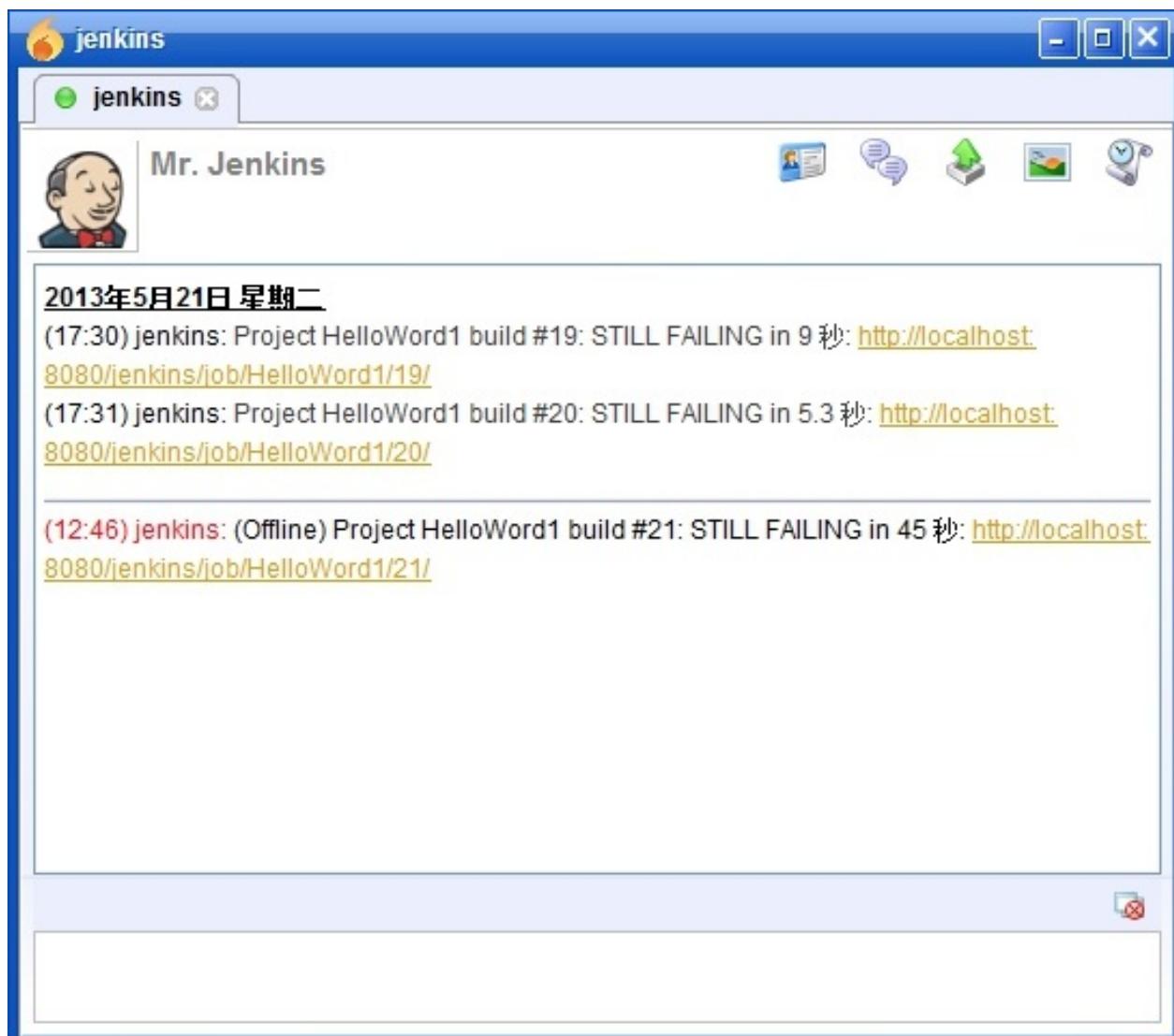
为了能上Jenkins能使用Jabber服务，还需安装下图所显示的两个插件。

|                                                                                                                                                                                                                                                                                                                                     |      |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <input checked="" type="checkbox"/> <a href="#">Jenkins instant-messaging plugin</a><br>This plugin provides abstract support for build notification via instant-messaging.                                                                                                                                                         | 1.25 |
| <input checked="" type="checkbox"/> <a href="#">Jenkins Jabber notifier plugin</a><br>Sends build notifications to jabber contacts and/or chatrooms. Also allows control of builds via a jabber 'bot'.<br>Note that the instant-messaging plugin 1.24 is a requirement for this plugin. Please make sure that it is installed, too! | 1.25 |

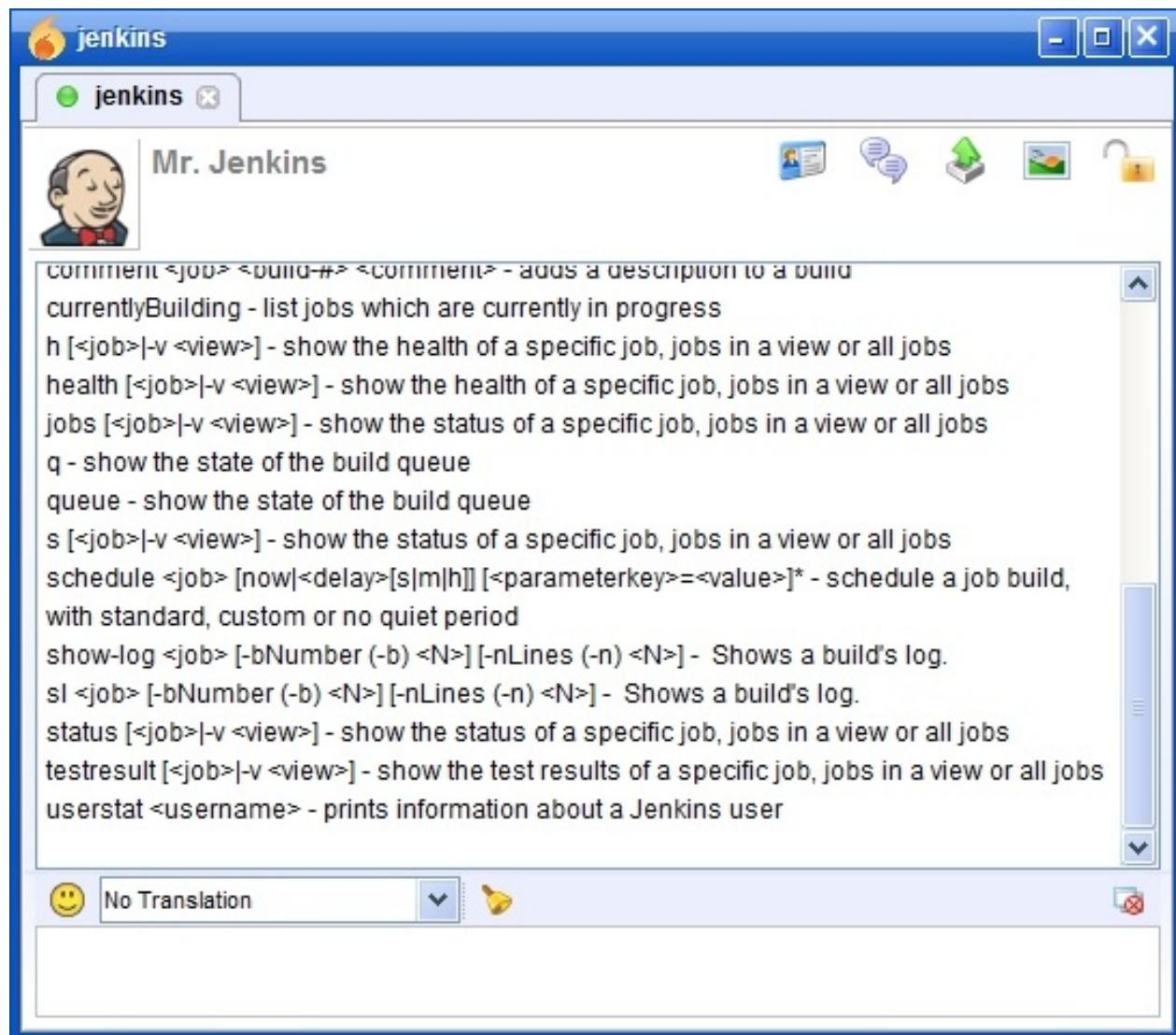
安装完这两个插件后，进行系统管理，系统设置页面，配置Jabber插件。



完成配置后，开发人员可通过Jabber客户端与Jenkins系统交互。Jenkins每次构建后，都将会给用户发送消息，如下图。



开发人员，还可以通过这个工具与Jenkins服务器交互，如下图。



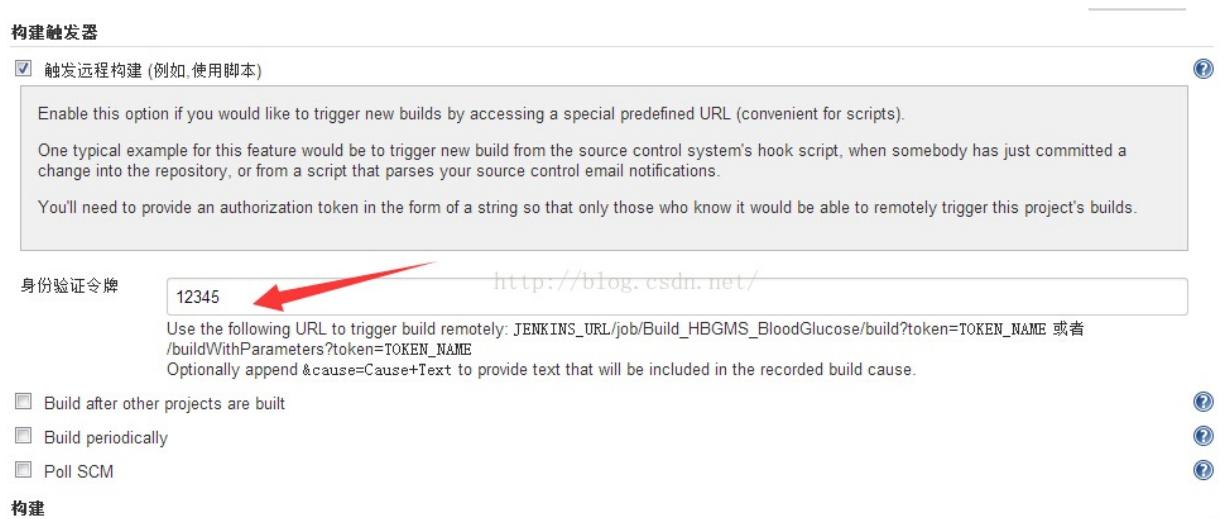
# jenkins 如何做到触发远程构建

来源:测试蜗牛, 一步一个脚印

搭建好了jenkins环境, 并且创建好了Job后, 如何远程触发构建而不需要登录Jenkins管理系统。

很简单的几步就能搞定。步骤如下

1、选择“触发远程构建”->输入口令牌“123456”, 如下图



2、组合url地址

jenkins地址/job/Job名称/build??token=口令&cause=书写构建原因 , 如下是我的地址:

[http://192.168.100.\\*\\*:8080/job/Bulid\\_HBGMS\\_framework/build?token=123456&cause=书写构建原因](http://192.168.100.**:8080/job/Bulid_HBGMS_framework/build?token=123456&cause=书写构建原因)

3、发送构建请求, 在浏览器中访问url

[http://192.168.100.\\*\\*:8080/job/Bulid\\_HBGMS\\_framework/build?token=123456&cause=书写构建原因](http://192.168.100.**:8080/job/Bulid_HBGMS_framework/build?token=123456&cause=书写构建原因) 如果不需要构建原因, 则不需要cause, 如下:

[http://192.168.100.\\*\\*:8080/job/Bulid\\_HBGMS\\_framework/build?token=123456](http://192.168.100.**:8080/job/Bulid_HBGMS_framework/build?token=123456)

4、请求后浏览器不会有反应, 后台收到请求开始构建。

# Jenkins学习四：Jenkins 邮件配置

来源:<http://www.cnblogs.com/yangxia-test/p/4366172.html>

## 本文主要对Windows环境 jenkins 的邮件通知进行介绍

jenkins 内置的邮件功能 使用email-ext插件扩展的邮件功能

邮件通知功能主要包含两个部分：全局配置和项目配置。

### 一. 先介绍下内置的Jenkins 邮件服务器 配置

1、系统管理－系统设置，先设置发件人的邮件，切记：一定要设置，且在系统管理员那个地方设置的email地址要和email配置的相同

Jenkins Location

|             |                                                         |                   |
|-------------|---------------------------------------------------------|-------------------|
| Jenkins URL | <input type="text" value="http://192.168.9.188:8080/"/> | <a href="#">?</a> |
| 系统管理员邮件地址   | <input type="text" value="[REDACTED]"/>                 | <a href="#">?</a> |

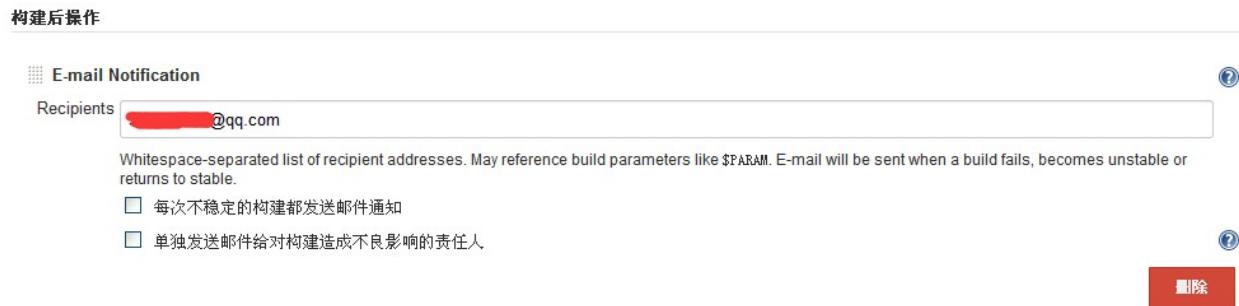
2、系统管理－系统设置，再设置全局设置：

邮件通知

|                                       |                                                |                   |
|---------------------------------------|------------------------------------------------|-------------------|
| SMTP服务器                               | <input type="text" value="smtp.qq.com"/>       | <a href="#">?</a> |
| 用户名                                   | <input type="text" value="[REDACTED]@qq.com"/> | <a href="#">?</a> |
| 密码                                    | <input type="password" value="....."/>         | <a href="#">?</a> |
| 使用SSL协议                               | <input checked="" type="checkbox"/>            | <a href="#">?</a> |
| SMTP端口                                | <input type="text" value=""/>                  | <a href="#">?</a> |
| Reply-To Address                      | <input type="text" value=""/>                  | <a href="#">?</a> |
| 字符集                                   | <input type="text" value="UTF-8"/>             | <a href="#">?</a> |
| <input type="checkbox"/> 通过发送测试邮件测试配置 |                                                |                   |

可以勾选“通过发送测试邮件配置” 测试此配置能否连通，如果收到以下邮件，恭喜 This is test email #1 sent from Jenkins

3、对构建的job 添加邮件发送的步骤，针对具体job名称-配置，如下图：



这样每次build后都会发送邮件给这个接收者，到这里你会发现，只能发给固定的对象，且格式单一（txt）

## 二、介绍email-ext插件配置

Jenkins默认提供了一个邮件通知，能在构建失败、构建不稳定等状态后发送邮件。但是它本身有很多局限性，比如它的邮件通知无法提供详细的邮件内容、无法定义发送邮件的格式、无法定义灵活的邮件接收配置等等。在这样的情况下，我们找到了Jenkins Email Extension Plugin。该插件能允许你自定义邮件通知的方方面面，比如在发送邮件时你可以自定义发送给谁，发送具体什么内容等等。

email-ext插件可根据构建的结果，发送构建报告，给当前的committer（用git做代码管理）

- 1) 该插件支持jenkins 1.5以上的版本，插件的安装此处略，若您可选插件的页卡的列表是空的，先去高级页面检查更新下。
- 2) 插件用于job配置页面，添加构建后步骤“Editable Email Notification”

### 1、系统管理 – 系统设置，先设置全局：

**Extended E-mail Notification**

Override Global Settings ?

SMTP server mail.iflashbuy.com → 设置邮件服务器，如qq、126等 ?

Default user E-mail suffix  ?

System Admin E-mail Address [REDACTED] → 设置发件人的邮箱 ?

**高级...**

Default Content Type Plain Text (text/plain) → 可选html或text ?

Use List-ID Email Header ?

Add 'Precedence: bulk' Email Header ?

Default Recipients [REDACTED]@qq.com → 设置默认接收人邮箱 ?

Reply To List [REDACTED]@qq.com → 接收人列表 ?

Emergency reroute  ?

Excluded Recipients  ?

Default Subject \$PROJECT\_NAME - Build # \$BUILD\_NUMBER - \$BUILD\_STATUS! → 邮件主题 ?

Maximum Attachment Size 5 → 最大附件数量 ?

Default Content \$PROJECT\_NAME - Build # \$BUILD\_NUMBER - \$BUILD\_STATUS:  
Check console output at \$BUILD\_URL to view the results.  
p 这是一个测试邮件 ?

Default Pre-send Script  ?

Additional groovy classpath 增加 ?

Enable Debug Mode ?

Enable Security ?

Require Administrator for Template Testing ?

Content Token Reference ?

详细参数说明如下：

- 1、**Override Global Settings**: 如果不选，该插件将使用默认的E-mail Notification通知选项。反之，您可以通过指定不同于(默认选项)的设置来进行覆盖。
- 2、**Default Content Type**: 指定构建后发送邮件内容的类型，有Text和HTML两种。
- 3、**Use List-ID Email Header**: 为所有的邮件设置一个List-ID的邮件信头，这样你就可以在邮件客户端使用过滤。它也能阻止邮件发件人大部分的自动回复(诸如离开办公室、休假等等)。你可以使用你习惯的任何名称或者ID号，但是他们必须符合如下其中一种格式(真实的ID必须要包含在<和>标记里)：

### Build Notifications

“Build Notifications”

4、**Add 'Precedence: bulk' Email Header:** 设置优先级,

5、**Default Recipients:** 自定义默认电子邮件收件人列表。如果没有被项目配置覆盖,该插件会使用这个列表。您可以在项目配置使用\$ DEFAULT\_RECIPIENTS参数包括此默认列表, 以及添加新的地址在项目级别。添加抄送: cc:电子邮件地址例如,CC:someone@somewhere.com

6、**Reply To List:** 回复列表, A comma separated list of e-mail addresses to use in the Reply-To header of the email. This value will be available as \$DEFAULT\_REPLYTO in the project configuration.

7、**Emergency reroute:** 如果这个字段不为空, 所有的电子邮件将被单独发送到该地址 (或地址列表) 。

8、**Excluded Committers:** 防止邮件被邮件系统认为是垃圾邮件,邮件列表应该没有扩展的账户名(如:@domain.com),并且使用逗号分隔

9、**Default Subject:** 自定义邮件通知的默认主题名称。该选项能在邮件的主题字段中替换一些参数, 这样你就可以在构建中包含指定的输出信息。

10、**Maximum Attachment Size:** 邮件最大附件大小。

11、**Default Content:** 自定义邮件通知的默认内容主体。该选项能在邮件的内容中替换一些参数, 这样你就可以在构建中包含指定的输出信息。

12、**Default Pre-send Script:** 默认发送前执行的脚本 (注: grooy脚本, 这是我在某篇文章上看到的, 不一定准确) 。

13、**Enable Debug Mode:** 启用插件的调试模式。这将增加额外的日志输出, 构建日志以及Jenkins的日志。在调试时是有用的, 但不能用于生产。

14、**Enable Security:** 启用时, 会禁用发送脚本的能力, 直接进入Jenkins实例。如果用户试图访问Jenkins管理对象实例, 将抛出一个安全异常。

15、**Content Token Reference:** 邮件中可以使用的变量, 所有的变量都是可选的。

## 2、项目配置

1) 要想在一个项目中使用email-ext插件, 你首先必须在项目配置页激活它。在构建后操作——“Add Post-build Actions”选项中勾选“Editable Email Notification”标签。如下图:

|                                                                                                                                                                             |                                                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| <b>Editable Email Notification</b>                                                                                                                                          |                                                                          |
| Disable Extended Email Publisher <input type="checkbox"/>                                                                                                                   |                                                                          |
| Allows the user to disable the publisher, while maintaining the settings                                                                                                    |                                                                          |
| Project Recipient List                                                                                                                                                      | <input type="text"/> @qq.com <b>接收人地址列表</b>                              |
| Comma-separated list of email address that should receive notifications for this project.                                                                                   |                                                                          |
| Project Reply-To List                                                                                                                                                       | <input type="text"/> @qq.com <b>接收人地址</b>                                |
| Comma-separated list of email address that should be in the Reply-To header for this project.                                                                               |                                                                          |
| Content Type                                                                                                                                                                | <input type="text"/> Plain Text (text/plain) <b>内容格式</b>                 |
| Default Subject                                                                                                                                                             | <input type="text"/> \$DEFAULT_SUBJECT <b>标题</b>                         |
| Default Content                                                                                                                                                             | <input type="text"/> \${JELLY_SCRIPT,template="text"} <b>内容</b>          |
| Attachments                                                                                                                                                                 | <input type="text"/> result.zip <b>附件</b>                                |
| Can use wildcards like 'module/dist/**/*.zip'. See the <a href="#">@includes of Ant fileset</a> for the exact format. The base directory is <a href="#">the workspace</a> . |                                                                          |
| Attach Build Log                                                                                                                                                            | <input type="button" value="Do Not Attach Build Log"/> <b>附件是否包括生成日志</b> |

## 项目基本配置参数说明：

当插件激活后你就能编辑如下字段（只列出常用的字段）：

**Project Recipient List:** 这是一个以逗号(或者空格)分隔的收件人邮件的邮箱地址列表。允许您为每封邮件指定单独的列表。Ps：如果你想在默认收件人的基础上添加收件人：\$DEFAULT\_RECIPIENTS,<新的收件人>

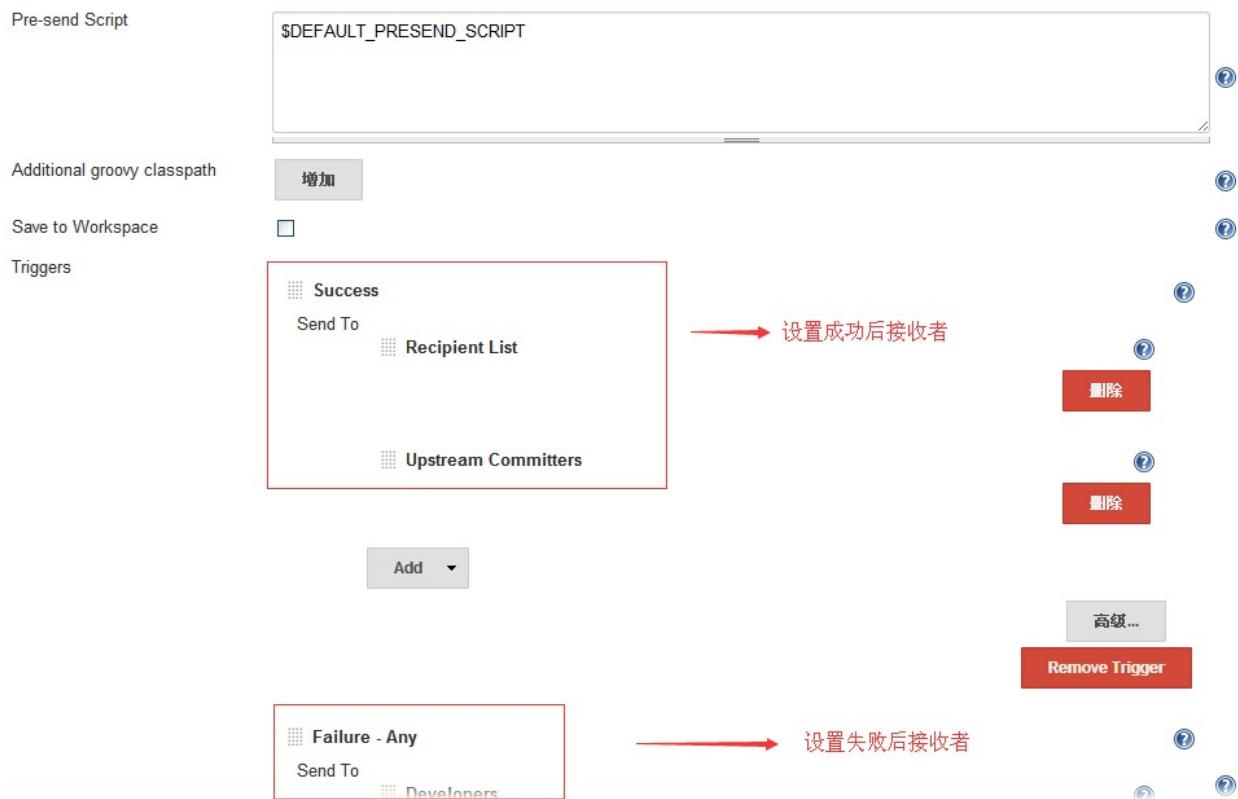
**Default Subject:** 允许你配置此项目邮件的主题。

**Default Content:** 跟Default Subject的作用一样，但是是替换邮件内容。

**Attach Build Log:** 附件构建日志。

**Compress Build Log before sending:** 发送前压缩生成日志（zip格式）。

2) 点击高级，设置触发器：（注意：所有的触发器都只能配置一次）



### 触发器参数说明：

**Failure:** 即时发送构建失败的邮件。如果“Still Failing”触发器已配置，而上一次构建的状态是“Failure”，那么“Still Failing”触发器将发送一封邮件来替代(它)。

**Unstable:** 即时发送构建不稳固的邮件。如果“Still Unstable”触发器已配置，而上一次构建的状态是“Unstable”，那么“Still Unstable”触发器将发送一封邮件来替代(它)。

**Still Failing:** 如果两次或两次以上连续构建的状态为“Failure”，发送该邮件。

**Success:** 如果构建的状态为“Successful”发送邮件。如果“Fixed”已配置，而上次构建的状态为“Failure”或“Unstable”，那么“Fixed”触发器将发送一封邮件来替代(它)。

**Fixed:** 当构建状态从“Failure”或“Unstable”变为“Successful”时发送邮件。

**Still Unstable:** 如果两次或两次以上连续构建的状态为“Unstable”，发送该邮件。

**Before Build:** 当构建开始时发送邮件。

对于内容，你也许注意到了这里调用了个 '`html.jelly`' 的模板，这是插件内置的，直接用即可。

当然也可以自己写 `jelly`文件，确保放置 `jenkins/home/email-template` 下以供jenkins调用。

### 三、附email-ext邮件通知模板

发现一个很好的邮件通知模板，如下：

Default Subject:

```
构建通知: ${BUILD_STATUS} - ${PROJECT_NAME} - Build #
${BUILD_NUMBER} !
```

Default Content:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>${ENV, var="JOB_NAME"}-第${BUILD_NUMBER}次构建日志</title>
</head>

<body leftmargin="8" marginwidth="0" topmargin="8" marginheight="4"
 offset="0">
 <table width="95%" cellpadding="0" cellspacing="0"
 style="font-size: 11pt; font-family: Tahoma, Arial, Helvetica, sans-serif">
 <tr>
 <td>(本邮件是程序自动下发的, 请勿回复!)</td>
 </tr>
 <tr>
 <td><h2>
 构建结果 - ${BUILD_STATUS}
 </h2></td>
 </tr>
 <tr>
 <td>

 构建信息
 <hr size="2" width="100%" align="center" /></td>
 </tr>
 <tr>
 <td>

 项目名称: ${PROJECT_NAME}
 构建编号: 第${BUILD_NUMBER}次构建
 SVN版本: ${SVN_REVISION}
 触发原因: ${CAUSE}
 构建日志: ${BUILD_URL}console
 构建: ${BUILD_URL}
 工作目录: ${PROJECT_URL}ws
 项目: ${PROJECT_URL}

 </td>
 </tr>
 <tr>
 <td>Changes Since Last
 Successful Build:
 <hr size="2" width="100%" align="center" /></td>
 </tr>
 <tr>
 <td>

 历史变更记录 : ${PROJECT_URL}changes

 ${CHANGES_SINCE_LAST_SUCCESS, reverse=true, format="Changes for Build #%n:
%c
", showPaths=true, changesFormat=""}
<pre>[%a]
%m</pre>,pathFormat="&nbsp&nbsp&nbsp&p"
 </td>
 </tr>
 <tr>
 <td>Failed Test Results
 <hr size="2" width="100%" align="center" /></td>
 </tr>
 <tr>
 <td><pre
 style="font-size: 11pt; font-family: Tahoma, Arial, Helvetica, sans-serif">$FAILED_TESTS</pre>

</td>
 </tr>
 <tr>
 <td>构建日志 (最后 100行):

```

```
<hr size="2" width="100%" align="center" /></td>
</tr>
<!-- <tr>
 <td>Test Logs (if test has ran): ${PROJECT_URL}/ws/TestResult/archive_logs/Log-
Build-${BUILD_NUMBER}.zip

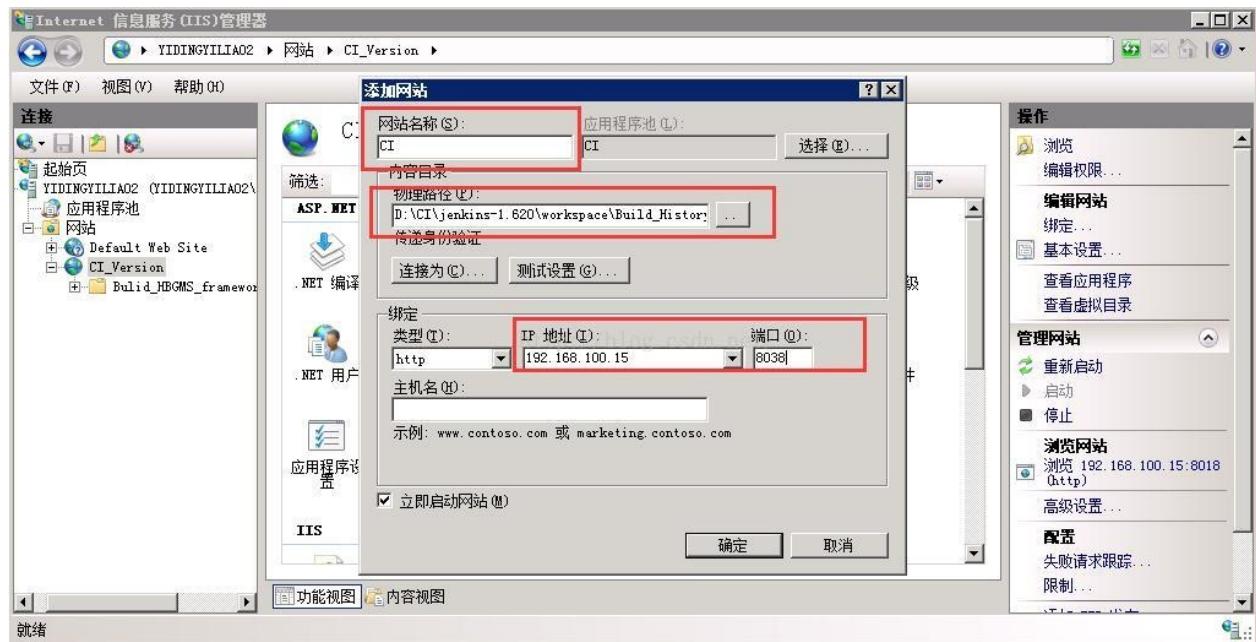
 </td>
</tr> -->
<tr>
 <td><textarea cols="80" rows="30" readonly="readonly"
 style="font-family: Courier New">${BUILD_LOG, maxLines=100}</textarea>
 </td>
</tr>
</table>
</body>
</html>
```

# Jenkins构建的版本包如何发布成可下载的资源—IIS发布方式

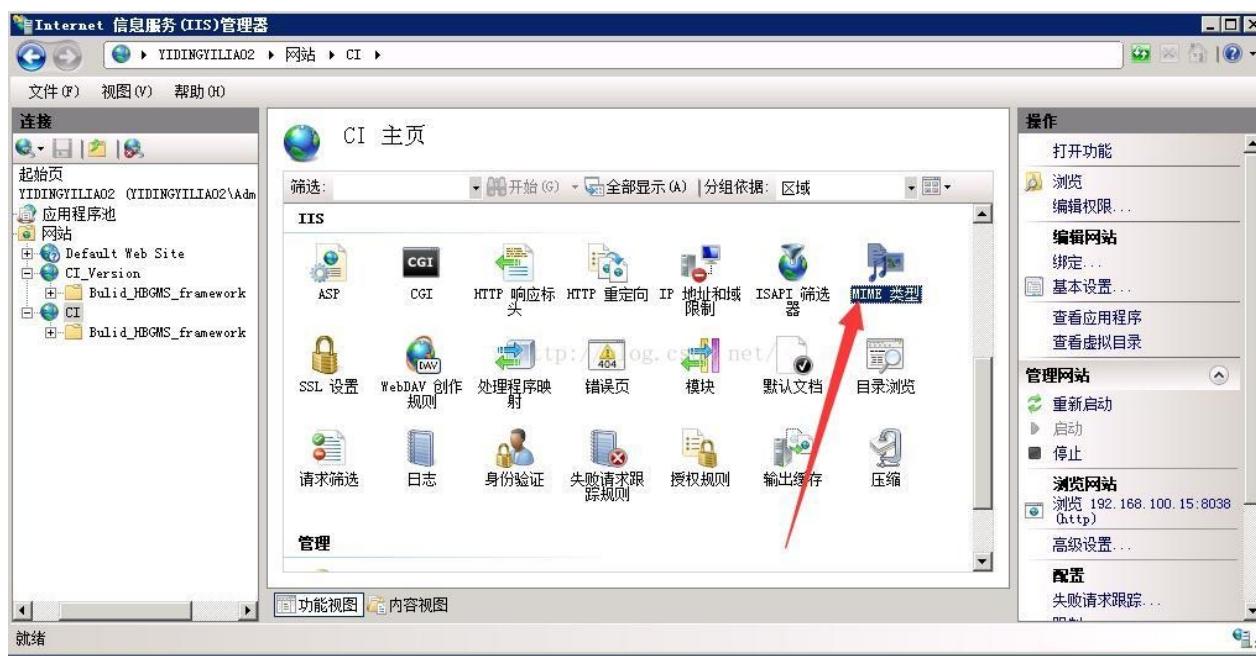
来源:测试蜗牛，一步一个脚印

备注: 另外一种是ftp站点方式，可以百度搜下就出来了。

1、右键->添加站点，物理路径为版本包的跟目录

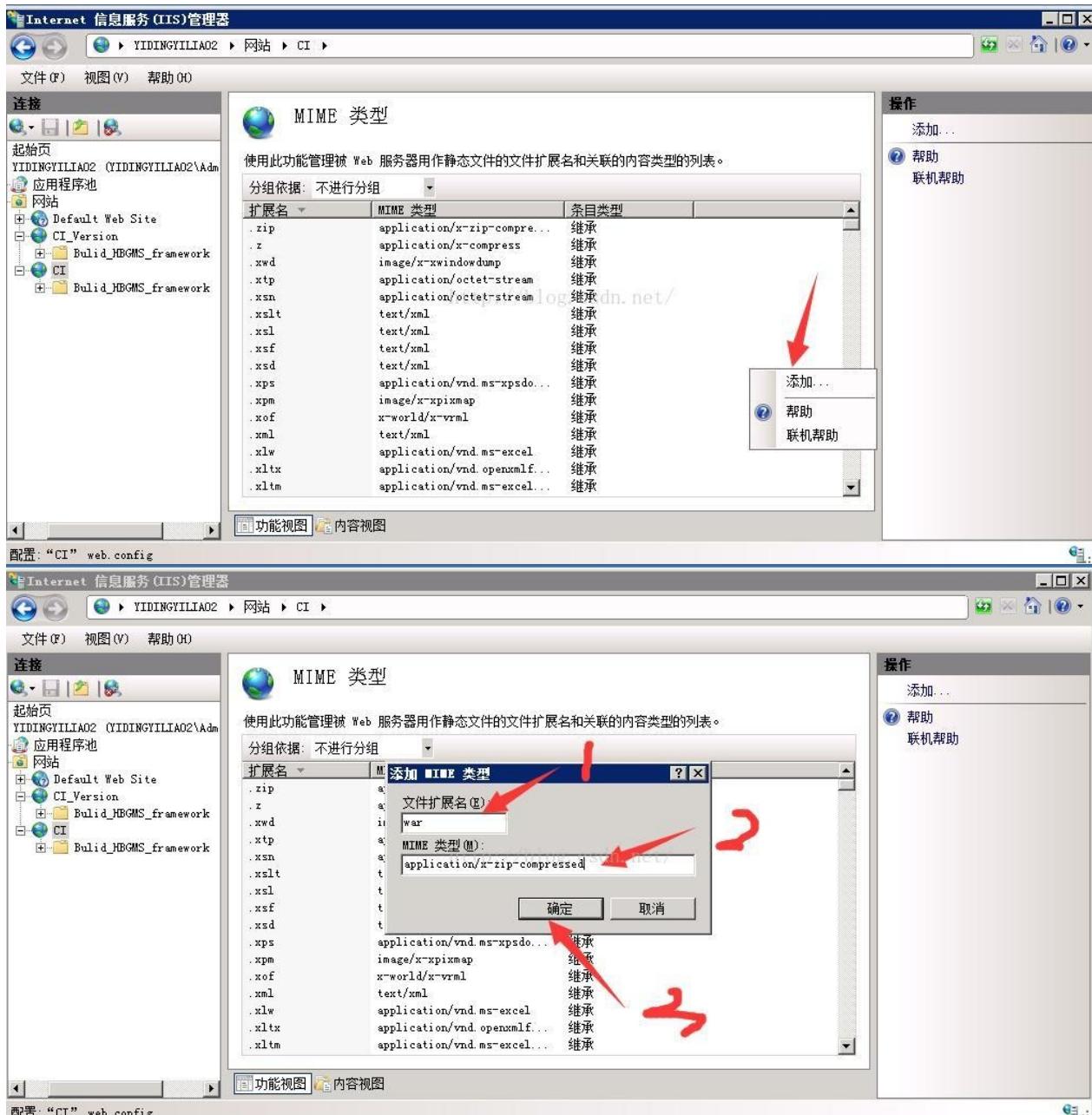


2、添加mime类型



如果是zip压缩包，则添加类型名“application/x-zip-compressed”，扩展名为“zip”

如果是apk包，则增加类型名为“application/vnd.android”，文件扩展名为“apk”



### 3、通过连接访问并下载

[http://192.168.\\*\\*.\\*\\*:8018/Bulid\\_HBGMS\\_framework/](http://192.168.**.**:8018/Bulid_HBGMS_framework/)

The screenshot shows a browser window with the URL [http://192.168.100.15:8018/Bulid\\_HBGMS\\_framework/](http://192.168.100.15:8018/Bulid_HBGMS_framework/). The title bar indicates the page is for 'Bulid\_HBG'. The browser interface includes a toolbar with icons for back, forward, search, and refresh, and a tab bar with multiple open tabs. The main content area displays a table of build logs:

| Date      | Time  | Action                                   |
|-----------|-------|------------------------------------------|
| 2015/8/14 | 16:34 | <dir> <a href="#">Build 63 framework</a> |
| 2015/8/14 | 16:40 | <dir> <a href="#">Build 64 framework</a> |
| 2015/8/14 | 16:41 | <dir> <a href="#">Build 65 framework</a> |
| 2015/8/14 | 16:46 | <dir> <a href="#">Build 66 framework</a> |
| 2015/8/14 | 16:46 | <dir> <a href="#">Build 67 framework</a> |
| 2015/8/14 | 16:50 | <dir> <a href="#">Build 68 framework</a> |
| 2015/8/14 | 17:33 | <dir> <a href="#">Build 69 framework</a> |
| 2015/8/17 | 9:26  | <dir> <a href="#">Build 70 framework</a> |
| 2015/8/17 | 9:27  | <dir> <a href="#">Build 71 framework</a> |
| 2015/8/17 | 9:32  | <dir> <a href="#">Build 72 framework</a> |
| 2015/8/17 | 9:51  | <dir> <a href="#">Build 73 framework</a> |
| 2015/8/17 | 10:10 | <dir> <a href="#">Build 74 framework</a> |
| 2015/8/17 | 16:11 | <dir> <a href="#">Build 75 framework</a> |
| 2015/8/17 | 16:23 | <dir> <a href="#">Build 76 framework</a> |
| 2015/8/17 | 16:34 | <dir> <a href="#">Build 77 framework</a> |
| 2015/8/17 | 16:37 | <dir> <a href="#">Build 78 framework</a> |
| 2015/8/17 | 17:37 | <dir> <a href="#">Build 79 framework</a> |
| 2015/8/18 | 9:41  | <dir> <a href="#">Build 80 framework</a> |

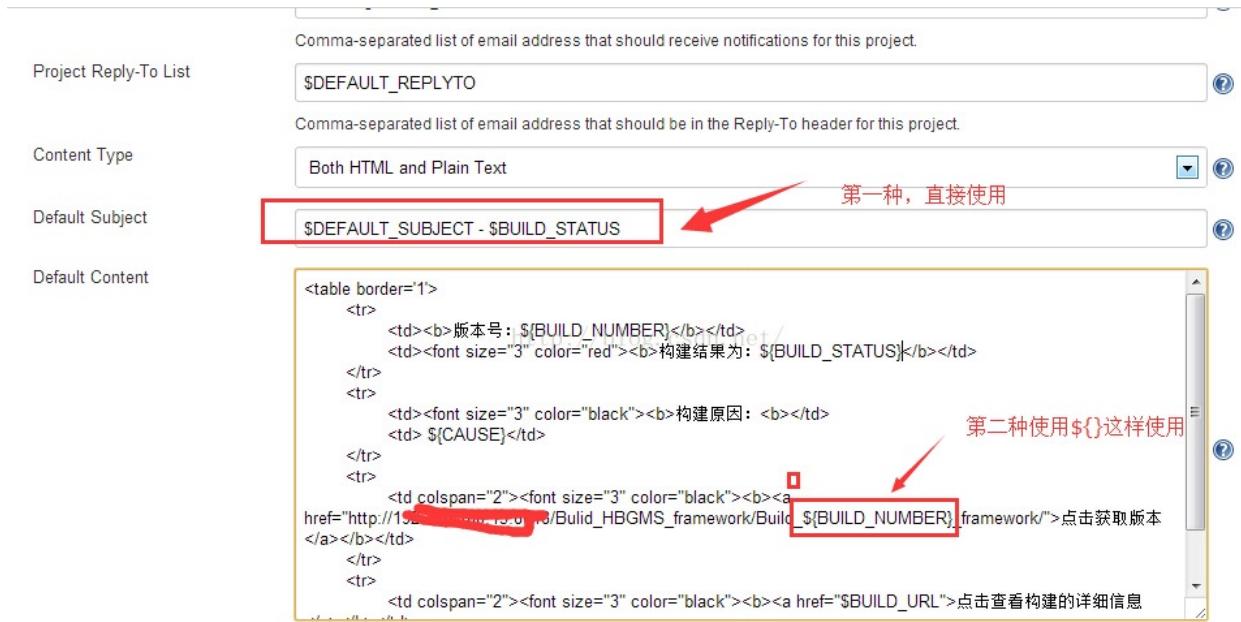
# Jenkins的环境变量的使用

来源：[测试蜗牛，一步一个脚印](#)

两种使用方式

1、直接使用\$标志：如\${BUILD\_STATUS}

2、使用\${}标志：如\${BUILD\_STATUS}



备注：

如果是使用批处理命令来使用环境变量，则是通过%%来标志，如% BUILD\_STATUS %

环境变量列表：

**BUILD\_NUMBER**, 唯一标识一次build, 例如23;

**BUILD\_ID**, 基本上等同于BUILD\_NUMBER, 但是是字符串, 例如2011-11-15\_16-06-21;

**JOB\_NAME**, job的名字, 例如JavaHelloWorld;

**BUILD\_TAG**, 作用同BUILD\_ID,BUILD\_NUMBER,用来全局地唯一标识此次build, 例如jenkins-JavaHelloWorld-23;

**EXECUTOR\_NUMBER**, 例如0;

**NODE\_NAME**, slave的名字, 例如MyServer01;

**NODE\_LABELS**, slave的label, 标识slave的用处, 例如JavaHelloWorldMyServer01;

**JAVA\_HOME**, java的home目录, 例如C:\Program Files (x86)\Java\jdk1.7.0\_01;

**WORKSPACE**, job的当前工作目录, 例如c:\jenkins\workspace\JavaHelloWorld;

**HUDSON\_URL = JENKINS\_URL**, jenkins的url, 例如<http://localhost:8000/>;

**BUILD\_URL**, build的url 例如<http://localhost:8000/job/JavaHelloWorld/23/>;

**JOB\_URL**, job的url, 例如<http://localhost:8000/job/JavaHelloWorld/>;

**SVN\_REVISION**, svn 的revision, 例如4;

# Jenkins自编邮件模板

来源:测试蜗牛，一步一个脚印

```
\<table border='1'>
<tr>
 <td>版本号: ${BUILD_NUMBER}</td>
 <td>构建结果为: ${BUILD_STATUS}</td>
</tr>
<tr>
 <td>构建原因: </td>
 <td> ${CAUSE}</td>
</tr>
<tr>
 <td colspan="2">点击查看构</td>
</tr>
<tr>
 <td colspan="2">点击查看构建详细信息</td>
</tr>
</table>
```

上面的邮件模板，拷贝到content里面即可，效果如下：



# Jenkins 邮件配置 (使用 Jenkins Email Extension Plugin)

来源:H@H@

本文主要对 jenkins 的邮件通知进行介绍，

- jenkins 内置的邮件功能
  - 使用插件扩展的邮件功能

## 1. 先介绍下 基本的Jenkins 邮件服务器 配置

1) system config 页面 (以公用的163邮件服务器为例) :

可以勾选“通过发送测试邮件配置”测试此配置能否连通。如果收到以下邮件，恭喜

This is test email #1 sent from Jenkins

2) 下面接着对构建的job 添加邮件发送的步骤。

**E-mail Notification**

Recipients `XXXX@163.com`

Whitespace-separated list of recipient addresses. May reference build parameters like `$PARAM`. E-mail will be sent when a build fails, becomes unstable or returns stable.

每次不稳定的构建都发送邮件通知

单独发送邮件给对构建造成不良影响的责任人

这样每次build后都会发送邮件给这个接收者，到这里你会发现，只能发给固定的对象，且格式单一（txt）

====好了，现在进入主题=====

## 2. 使用插件“Email Extension Plugin”进行扩展

它可根据构建的结果，发送构建报告，给当前的committer（用git做代码管理）

**1) 该插件支持jenkins 1.5以上的版本，至少我的 1.486是不支持的啦。所以果断升级吧。。**

插件的安装此处略，若您可选插件的页卡的列表是空的，先去高级页面检查更新下。

**2) 插件用于job配置页面，添加构建后步骤“Editable Email Notification”**

**Editable Email Notification**

Project Recipient List `XXXX@163.com`

Comma-separated list of email address that should receive notifications for this project.

Project Reply-To List `$DEFAULT_REPLYTO`

Command-separated list of email address that should be in the Reply-To header for this project.

Content Type `HTML (text/html)`

Default Subject `$DEFAULT_SUBJECT`

Default Content  `${JELLY_SCRIPT,template='html'}`

Attachments

Can use wildcards like 'module/dist/\*\*/\*.\*zip'. See the [@includes of Ant fileset](#) for the exact format. The base di

上面的配置给出了该工程的默认接收列表，当然抄送的话直接可以这么写  
cc:xxxx@163.com

对于内容，你也许注意到了 这里调用了个 ‘html.jelly’ 的模板，这是插件内置的，直接用即可。 （支持git每次变更的记录，mvn 及junit 等编译的结果报告）

当然也可以自己写 jelly文件， 确保放置 jenkins/home/email-template下 以供jenkins调用。

### 3) 至此你也许会问 这不还是用的固定的接收列表嘛 (⊙\_⊙) , 别着急 看到右下角的高级选项没， 继续配置，



我设置了 build成功和失败都发给 默认的接收者和当前提交代码的家伙，而send to requester 是指手动触发构建时当前登陆jenkins的用户。

ps:

- 1.如果有人 git commit时候没有进行global的name和email设置， 将不会发送到正确的邮箱 (jenkins将按各自的机器名作为域名地址发送到错误的邮箱)
- 2.当然还可在jenkins 管理用户中 个别设置 邮箱。不过对于团队较多的话，你就苦了。 所以还是有必要请大家提交前进行实名设置。

# Jenkins里邮件触发器配置Send to Developers

来源:[测试蜗牛，一步一个脚印](#)

## 邮件触发类型介绍（Triggers）

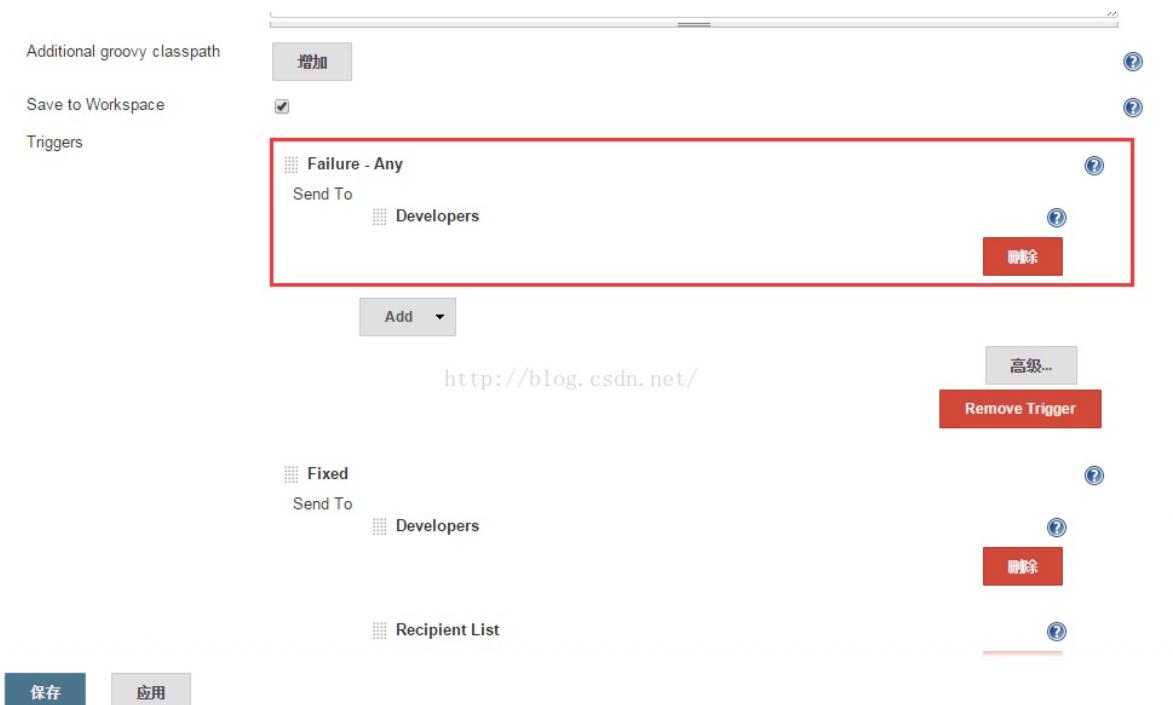
By default, the only trigger configured is the "Failure" trigger. To add more triggers, select one from the dropdown, and it will be added to the list. Once you have added a trigger, you have several options. If you click "?"(question mark) next to a trigger, it will tell you what conditions must be met for it to send an email.

- **Send to Recipient List** (邮件发送给全局邮件列表配置的所有邮件地址) - Check this checkbox if you would like to have the email sent to the "Global Recipient List" configured above.
- **Send to Developers** (发送给开发，谁check in就发送给谁) - Check this checkbox to send the email to anyone who checked in code for the last build. The plugin will generate an email address based on the committer's id and an appended "default email suffix" from Jenkins's global configuration page. For instance, if a change was committed by someone with an id "first.last", and the default email suffix is "@somewhere.com", then an email will be sent to first.last@somewhere.com
- **Include Culprits** (发送给所有提交代码的人，直到最后build成功) - If this is checked AND Send To Developers is checked, emails will include everyone who committed since the last successful build.
- **More Configuration** (更多设置) - Configure properties at a per-trigger level.
- **Recipient List** (邮件接收者) - A comma (逗号) (and whitespace) separated list of email address that should receive this email if it is triggered. This list is appended to the "Global Recipient List" above.
- **Subject** (指定项目名称) - Specify the subject line of the selected email.
- **Content** (指定邮箱内容) - Specify the body of the selected email.
- **Remove** - Click the delete button next to an email trigger to remove it from the configured triggers list.

Send to Developers (发送给开发，谁check in就发送给谁) 配置介绍：

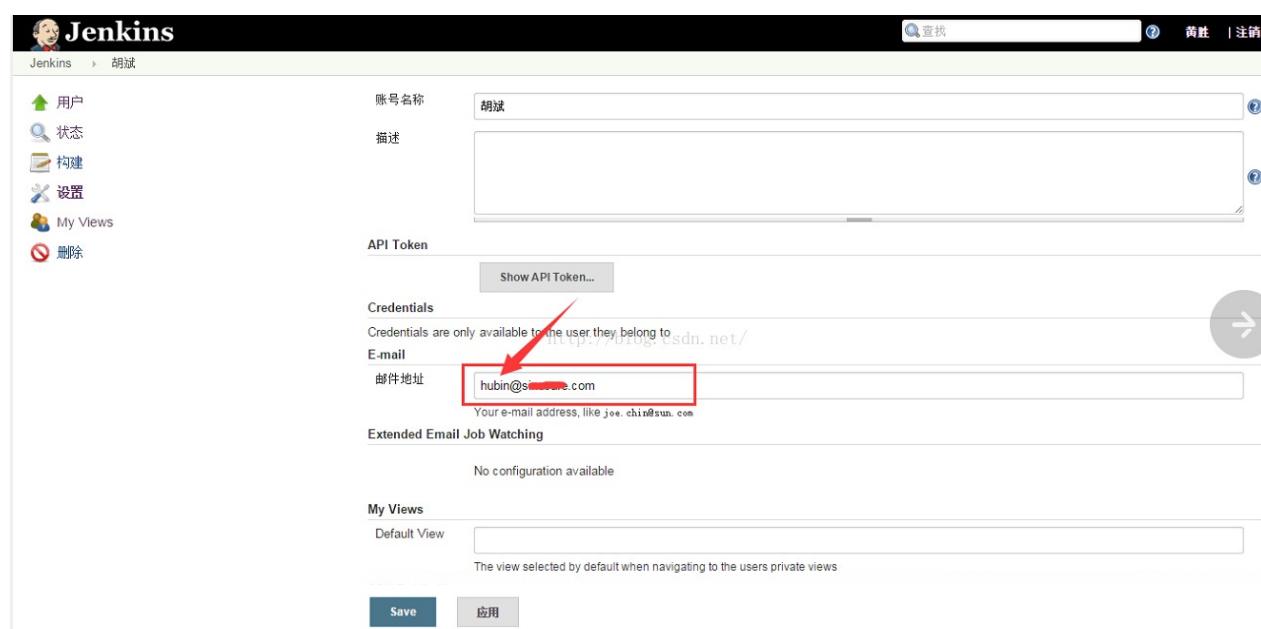
备注：

## 1、构建job中添加配置触发器



2、用户邮箱的配置，该开发者必须有自己的jenkins账户以及配置了邮箱

在Jenkins里面增加开发者账号



并且svn的账号和jenkins的ID保持一致

The screenshot shows the Jenkins interface with a user named '胡斌' (Hubin) and a Jenkins user ID of 'hubin'. A red box highlights the Jenkins user ID. Below it, a TortoiseSVN log messages window shows a commit from 'hubin' on 2015/8/7 at 10:14:12. Another red box highlights the author 'hubin' in the log table. A red arrow points from the Jenkins user ID to the author in the log table, with the text '两者一致' (Consistent) above the arrow.

邮箱的合成：

- 从系统管理里面查看邮箱SMTP Server的后缀
- 从svn里面查看账号
- 最后合成的邮箱就是为： hubin@\*\*.com

The screenshot shows the Jenkins 'System Management' page under 'Email' settings. It includes fields for 'Shell executable', 'Extended E-mail Notification' (with 'SMTP server' set to 'pop.sinocare.com' and 'Default user E-mail suffix' set to '@sinocare.com'), and 'Default Content Type' (set to 'HTML (text/html)'). A red arrow points from the 'Default user E-mail suffix' field to a Jenkins user profile window below, which displays a list of SVN commits. Another red arrow points from the Jenkins user profile window to the 'Default user E-mail suffix' field. A red box highlights the Jenkins user 'hubin' in both the Jenkins profile and the SVN log table. A red arrow points from the Jenkins user 'hubin' to the 'Default user E-mail suffix' field, with the text 'ID' above the arrow. To the right, the composed email address 'hubin@sinocare.com' is shown.

3、开发者上传了代码后，本机器第一次checkout，第二次就无法获取到最新代码更新者。

Jenkins  
Jenkins > Build\_HBGMs\_framework > #24

控制台输出

```
Started by user 董胜
Building in workspace D:\CI\jenkins-1.620\workspace\Build_HBGMs_framework
Updating https://hgxxxxxx:18443/svn/HBGMs/trunk/devl.0/code/easy.framework at revision '2015-08-10T15:06:54.625 +0800'
U src\com\easy\modules\dataworkbench\P601201.java
At revision 912
No emails were triggered.
[INFO] Scanning for projects...
[INFO] BUILD FAILURE
[INFO] http://blog.csdn.net/
[INFO] Total time: 0.173 s
[INFO] Finished at: 2015-08-10T15:07:04+08:00
[INFO] Final Memory: 5M/111M
[INFO]
[ERROR] The goal you specified requires a project to execute but there is no POM in this directory (D:\CI\jenkins-1.620\workspace\Build_HBGMs_framework\easy.framework). Please verify you invoked Maven from the correct directory. --> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MissingProjectException
Build step 'Invoke top-level Maven targets' marked build as failure
Email was triggered for: Failure - Any
Sending email for trigger: Failure - Any
Sending email to: hubin@xxxxxxxx.com
```

上传代码后，第一次checkout时，  
build失败后发送给相应的开发者

# jenkins里面使用批处理命令进行自动部署

来源：[测试蜗牛，一步一个脚印](#)

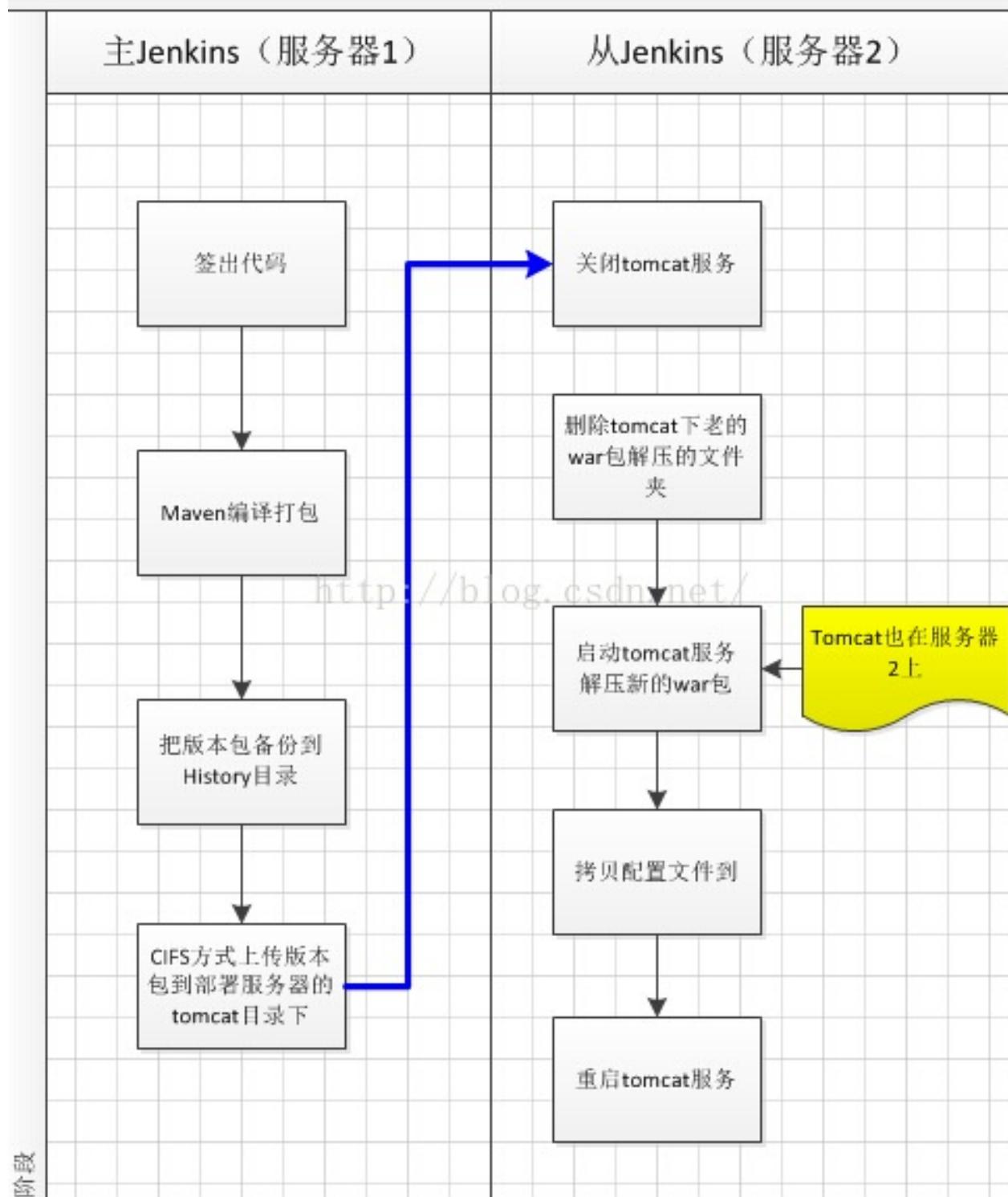
说明：

自动构建Job在服务器1上

自动部署Job在服务器2上

版本部署服务器在服务器2上

## Jenkins构建流程



1、自动构建Job（服务器1）使用批处理命令，拷贝构建包到发布目录，用以从邮件的超级链接中获取版本包

```
@echo off

echo 包重命名

d:

cd D:\CI\jenkins-1.620\workspace\Old_Bulid_HBGMS_framework\framework\target\

rename framework-1.0-SNAPSHOT.war framework.war

echo 拷贝版本包到版本下载目录

echo 拷贝

xcopy D:\CI\jenkins-1.620\workspace\Old_Bulid_HBGMS_framework\framework\target\framewo
 "D:\CI\jenkins-1.620\workspace\Build_History\Old_Bulid_HBGMS_framework\Build_%BUI

echo 拷贝结束

echo 将构建包拷贝到deploy的目录下

echo 先清空deploy下面的版本包

d:

cd D:\CI\jenkins-1.620\workspace\Old_Depoy_HBGMS_framework

if exist framework rd framework /s /q
echo 拷贝到deploy目录下
xcopy D:\CI\jenkins-1.620\workspace\Old_Bulid_HBGMS_framework\framework
 \target\framework.war D:\CI\jenkins-1.620\workspace
 \Old_Depoy_HBGMS_framework\framework\ /y

echo 拷贝结束
```

2、自动构建Job（服务器1）使用Send files to a windows share（需要安装插件CIFS Publishers）把版本包拷贝到版本部署服务器（服务器2）

Send files to a windows share

CIFS Publishers

Name Deploy-100-16

Transfers

Transfer Set

Source files framework\target\framework.war

Remove prefix http://blog.csdn.net/

Remote directory tomcat\_8028\webapps\

All of the transfer fields support substitution of Jenkins environment variables

Add Transfer Set

3、自动部署job（服务器2）里面使用批处理命令进行老版本包的备份、配置文件替换、tomcat服务重启

项目名称 Deploy\_HBGMs\_framework

描述

[Plain text] 预览

丢弃旧的构建

参数化构建过程

关闭构建 (重新开启构建前不允许进行新的构建)

在必要的时候并发构建

Restrict where this project can be run

Label Expression Deploy\_16

配置该job执行的Jenkins节点

Slaves in label: 1

构建

Execute Windows batch command

命令

```

echo 备份旧版本文件到bak
echo 先关闭tomcat再操作文件
taskkill /f /im Tomcat_8028.exe
echo 延时10s让tomcat服务有足够时间关闭
ping 127.1 -n 10 >nul
echo 移动文件夹framework
d:
cd D:\huangweihua\tomcat_8028\bak
if exist framework rd framework /s /q
move D:\huangweihua\tomcat_8028\webapps\framework D:\huangweihua\tomcat_8028\bak
echo 重命名，后缀增加时间戳
set today=%date:~0,4%-%date:~5,2%-%date:~8,2%
echo 加里左左立性而不命定

```

部署处理步骤

http://blog.csdn.net/

参阅 可用环境变量列表

删除

```
@echo off
echo 包重命名
d:
cd D:\CI\jenkins-1.620\workspace\Old_Bulid_HBGMS_framework\framework\target\
rename framework-1.0-SNAPSHOT.war framework.war

echo 拷贝版本包到版本下载目录
ehco 拷贝
xcopy D:\CI\jenkins-1.620\workspace\Old_Bulid_HBGMS_framework\framework\target\framework\
"D:\CI\jenkins-1.620\workspace\Build_History
 \Old_Bulid_HBGMS_framework\Build_%BUILD_NUMBER%_framework\" /y
echo 拷贝结束

echo 将构建包拷贝到deploy的目录下
echo 先清空deploy下面的版本包
d:
cd D:\CI\jenkins-1.620\workspace\Old_Depoy_HBGMS_framework
if exist framework rd framework /s /q
echo 拷贝到deploy目录下
xcopy D:\CI\jenkins-1.620\workspace\Old_Bulid_HBGMS_framework\framework
 \target\framework.war D:\CI\jenkins-1.620\workspace
 \Old_Depoy_HBGMS_framework\framework\ /y
echo 拷贝结束
```

# Jenkins 集成 APK size 与 dexcount 趋势图

来源:<http://www.jianshu.com/p/c5c8528841eb#rd>

[TOC]

声明：本文也在我的微信公众号 **Android程序员(AndroidTrending)** 发布。原文

链接：[Android APK size + dexcount charts on Jenkins](#)

原文作者：[Marc Reichelt](#)

译文出自：[汤涛的简书](#)

译者：汤涛

状态：完成

最近在 **Android Weekly** 上看到的一篇文章，感觉有些帮助，文章不长，就顺手翻译了一下。**Jenkins** 这个持续集成工具，我们一直在用，感觉不错，用好它能帮助我们解决很多问题，极大提高团队开发效率。

我们在 [flinc](#) 为持续集成投入了大量的精力。为了改善 APK size 与 dex 中的方法/字段数，首先必须测量它。当我在 [Twitter](#) 上提出这个问题时，有人告诉了我 [dexcount-gradle-plugin](#)，它使用起来非常容易：

top-level build.gradle

```
buildscript {
 repositories {
 jcenter()
 }

 dependencies {
 // other dependencies go here...
 classpath 'com.getkeepsafe.dexcount:dexcount-gradle-plugin:0.5.2'
 }
}
```

app/build.gradle

```
android {
 // your android block goes here
}
apply plugin: 'com.getkeepsafe.dexcount'
```

现在，如果编译你的工程，这个插件将会在每次 build 时，输出 dex 方法数与字段数。

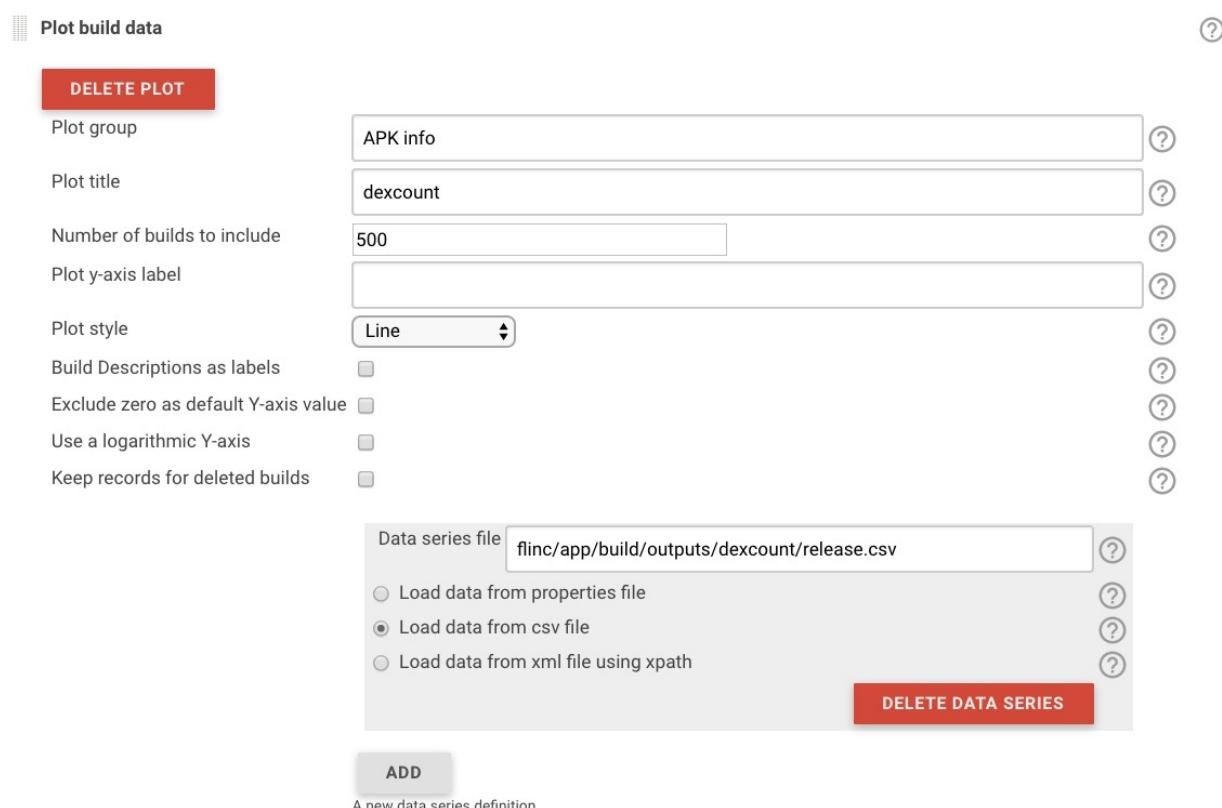
```
:app:assembleDebug
:app:countDebugDexMethods
Total methods in app-debug.apk: 61678 (94,11% used)
Total fields in app-debug.apk: 37417 (57,09% used)
Methods remaining in app-debug.apk: 3857
Fields remaining in app-debug.apk: 28118

BUILD SUCCESSFUL

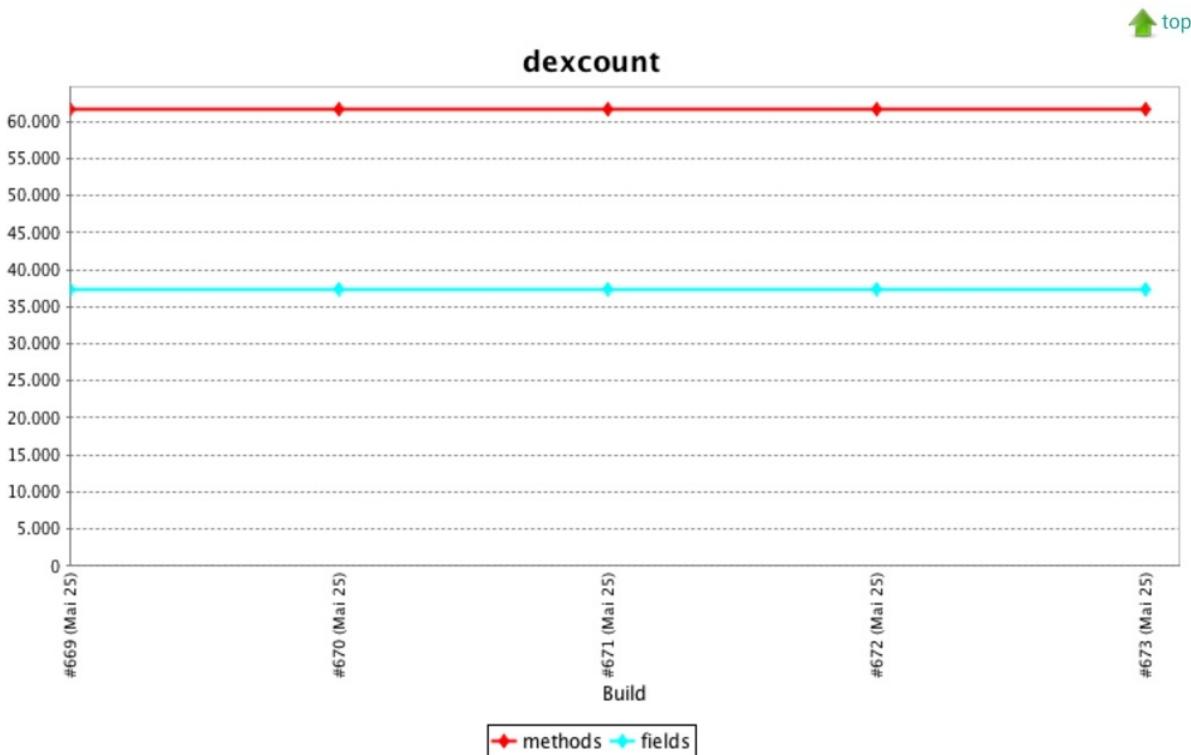
Total time: 41.057 secs
→ flinc git:(develop) ✘ cat app/build/outputs/dexcount/debug.csv
methods,fields
61678,37417
→ flinc git:(develop) ✘
```

以上已经非常有帮助了:-) 如果你不用 Jenkins 的话，可以不用继续往下看了。

接下来，让我们看看 Plot 这个 Jenkins 插件。给 Jenkins 安装 [Plot 插件](#)，并为你的 build 配置它。



注意根据你的工程调整 CSV 文件路径，debug 与 release build 各有不同的文件名。如果准备就绪，在你的 build 任务里点击 Plots（注意应该先跑 1~2 次 build 任务），应该可以看到一个漂亮的图表。来看看我们现在有什么了？



现在让我们来看看 APK size，打开 Jenkins build 配置，在 build 的最后，加一小段脚本。

```
OUTFILE=flinc/app/build/outputs/apksize.csv
echo filesize > $OUTFILE
yep, that's for Mac. Use "stat -c %s" instead on Linux
stat -f%z flinc/app/build/outputs/apk/app-release.apk >> $OUTFILE
```

同时添加新的 plot 变量：'Add Plot' 按钮（不是 'Add' 按钮，这个也会被添加到同样的图表中）。注意我在 group 字段里，填写了与之前一样的 'APK info'。

**DELETE PLOT**

Plot group  [?](#)

Plot title  [?](#)

Number of builds to include  [?](#)

Plot y-axis label  [?](#)

Plot style  [?](#)

Build Descriptions as labels  [?](#)

Exclude zero as default Y-axis value  [?](#)

Use a logarithmic Y-axis  [?](#)

Keep records for deleted builds  [?](#)

Data series file  [?](#)

Load data from properties file [?](#)

Load data from csv file [?](#)

Include all columns by name  [?](#)

Include columns by name  [?](#)

Exclude columns by name  [?](#)

Include columns by index  [?](#)

Exclude columns by index  [?](#)

CSV Exclusion values  [?](#)

URL  [?](#)

Display original csv above plot  [?](#)

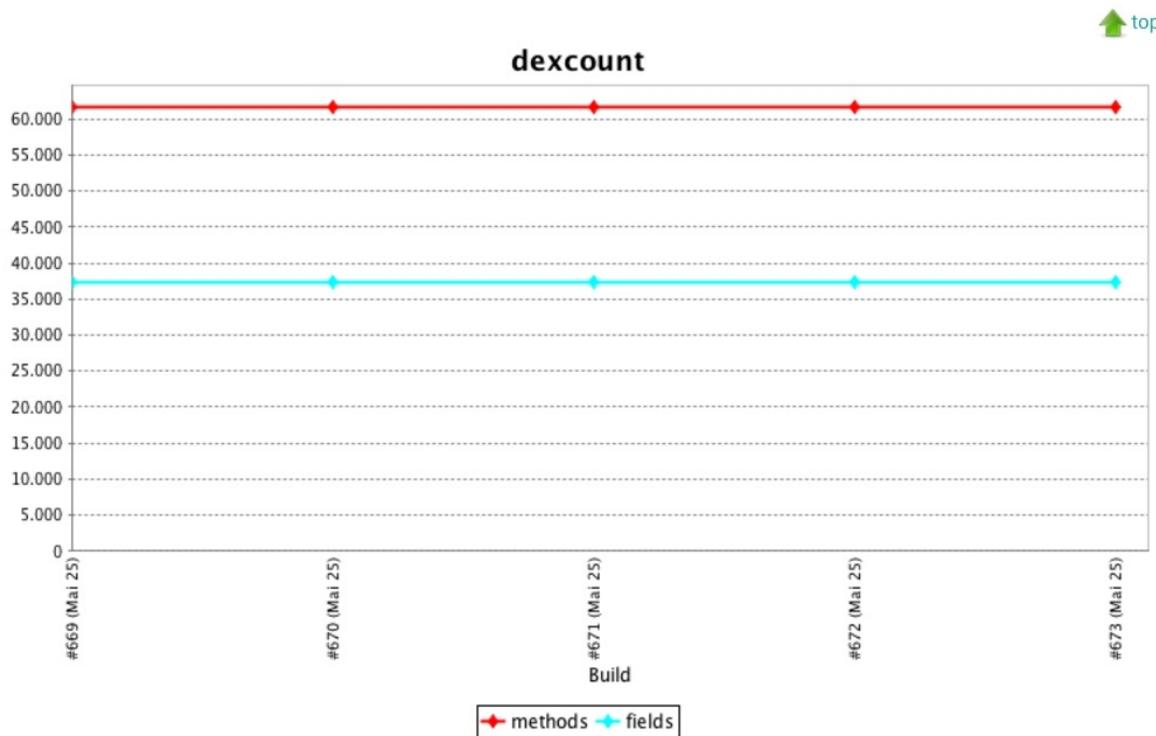
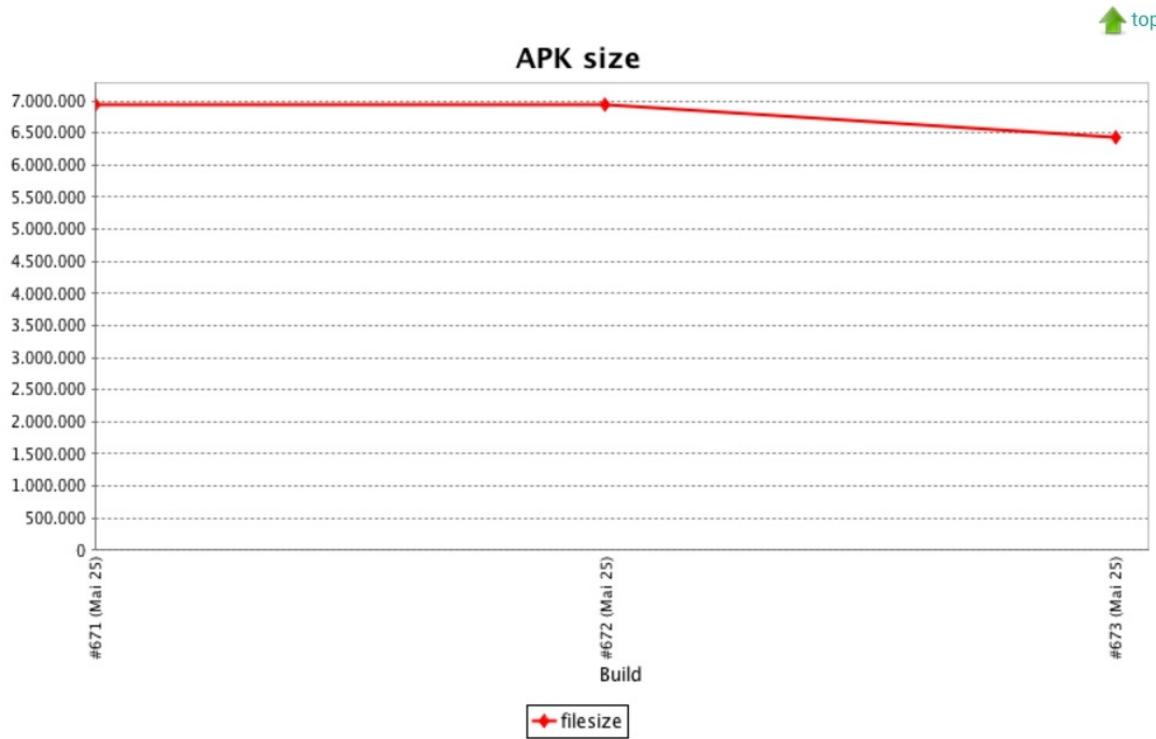
Load data from xml file using xpath [?](#)

**DELETE DATA SERIES**

**ADD**

A new data series definition

让我们来看看最终效果吧：

 **APK info**
Jump to [Plot 1: APK size](#) ↑

超赞！

特别提示：为了更轻松地创建 plots, 你可以简单创建一个 free-style 类型的 Jenkins 任务, 将指定的 .csv 文件复制到其 workspace, 并开始测试, 分分钟就可以搞定。

希望这篇文章会帮助你更容易地跟踪各种优化的效果，包括：[APK 瘦身](#)，[开启混淆](#)，[压缩图片](#)或者使用[矢量图片](#)等。祝你玩得愉快！

# Mac配置Jenkins

来源:<http://www.cnblogs.com/tangbinblog/p/3949078.html>

[TOC]

## 关于 Jenkins

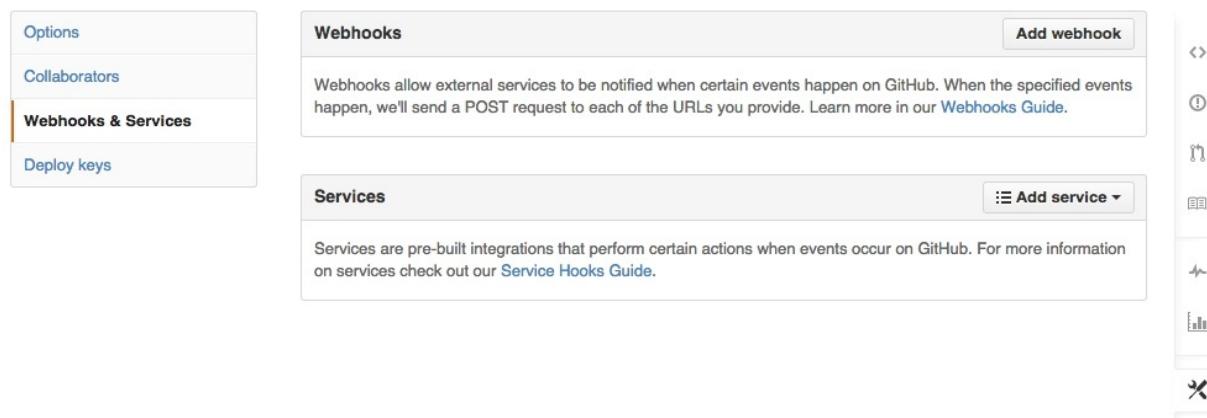
Jenkins 是一个开源软件项目，旨在提供一个开放易用的软件平台，使持续集成变成可能。

安装 Jenkins 并配置， <http://linjunpop.logdown.com/posts/162202-set-up-jenkins-server-on-the-mac-mini-to-run-ios-tests>

大致步骤也就是 安装Jenkins 创建job 根据配置的github 仓库地址，

然后jenkins 会在网站根目录下 【jenkins 就是java 编写的网站应用程序，跨平台】 创建工作区，拉一份代码。调用你配置的build 脚本 构建项目。

里面提到了 **github pull request build** 功能，其中涉及到 hooks 概念。hooks 简单理解就是 回调或切面编程。如果你在一个有域名的机器上部署 jenkins 。那么你可以通过设置在 github 对应仓库的



webhooks 对应地址 大致是这样的 `http://你的域名:端口/github-webhook/`

github 在某个事件发生时 【push pull delete】 发送post 请求到你的jenkins 。jenkins 解析，然后调用你配置好的处理脚本或其他的什么操作。

这里有篇文章可以参考：<http://www.fancycoding.com/automatic-deploy-task-using-github-hooks/>

它仅仅是个平台 开源功能丰富。像这种构建也可以在服务器创建代码仓库，定期执行脚本，上传ipa包，或其他操作。

详细jenkins 入门参考：[http://www.cnblogs.com/ppinfo/p/3224643.html#\\_Toc357079428](http://www.cnblogs.com/ppinfo/p/3224643.html#_Toc357079428)

# 使用jenkins自动化构建android和ios应用

来源:[www.jayfeng.com](http://www.jayfeng.com)

## 背景

随着业务需求的演进，工程的复杂度会逐渐增加，自动化的践行日益强烈。事实上，工程的自动化一直是我们努力的目标，能有效提高我们的生产效率，最大化减少人为出错的概率，实现一些复杂的业务需求应变。场景如下，公司现在的测试人员每次需要测试新版本，都需要开发人员打包，放到ftp，测试人员然后从ftp上拷贝到本地（或者用手机的ES文件管理器），再安装。尤其临近发版的一周，几乎每天都要新版本。这样的话，有两方面的影响：第一，打断了开发人员的开发进度；第二，开发人员打包效率低下，尤其是ios，不顺的话，总是打的不对（可能是证书的问题）。要解决这个问题，必须实现移动端应用的自动化构建。具体说来就是，使用持续集成（CI）系统jenkins，自动检测并拉取最新代码，自动打包android的apk和ios的ipa，自动上传到内测分发平台蒲公英上。（接下来，测试人员只要打开一个（或多个）固定的网址，扫描一下二维码，就能下载最新的版本了...）

## 环境

因为要编译ios，所以选择Mac OS X 10.11.1。

无论是哪个操作系统，jenkins的配置是一样的。

## 安装Jenkins

官网地址：<http://jenkins-ci.org/>

```
// 使用brew安装
brew install jenkins

// 启动，直接运行jenkins即可启动服务
jenkins
```

默认访问<http://localhost:8080/>，可进入jenkins配置页面。

## 安装Jenkins相关插件

点击系统管理>管理插件>可选插件， 可搜索以下插件安装

- git插件(GIT plugin)
- ssh插件(SSH Credentials Plugin)
- Gradle插件(Gradle plugin) - android专用
- Xcode插件(Xcode integration) - ios专用

## 新建Job

主页面， 新建 -> 构建一个自由风格的软件项目即可。

对于类似的项目， 可以选择 -> 复制已有的Item， 要复制的任务名称里输入其他job的首字符会有智能提示。

## 配置git仓库

如果安装了git插件， 在源码管理会出现Git， 选中之后：

Repositories -> <https://github.com/openproject/ganchai>, 如果是ssh还要配置 Credentials。

Branch -> \*/master, 选定一个要编译的分支代码。

如下：

● Git

Repositories 仓库地址 → Repository URL  
git@192.168.3.170:/Users/.../ANDROID\_CODE.git

Credentials - none - Add

Branches to build Branch Specifier (blank for 'any') \*/master

源码库浏览器 (自动) ↓ 代码分支

Additional Behaviours Add

如果是私有的仓库（比如git://xxxxx.git）,点击Credentials - Add, 弹出对话框, 配置sshkey最简单了:

Add Credentials

Kind SSH Username with private key

Scope Global (Jenkins, nodes, items, all child items, etc)

Username sshkey → 随便写

Description

Private Key  Enter directly  
Key -----BEGIN RSA PRIVATE KEY-----  
MII...  
-----END RSA PRIVATE KEY-----  
From a file on Jenkins master  
From the Jenkins master ~/.ssh

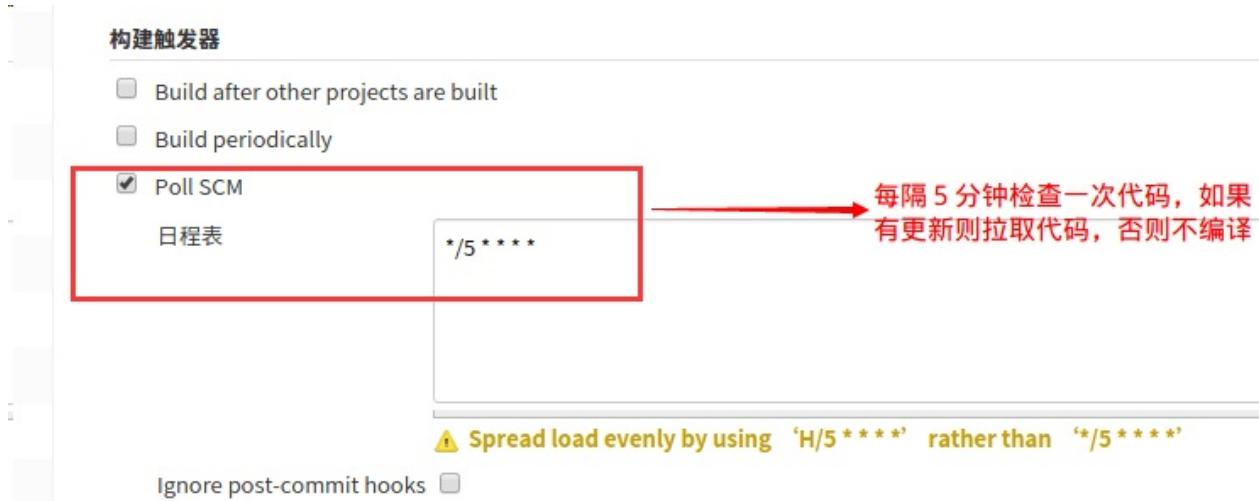
private ssh key, 一般写id\_rsa

Add Cancel

# 配置自动拉取最新代码

在构建触发器中，有两种自动拉取代码并编译的策略：

- 设置Poll SCM，设置定时器，定时检查代码更新，有更新则编译，否则不编译（我暂时用的是这个）。



- 也可以设置Build periodically，周期性的执行编译任务。



关于定时器的格式，我只能从网上摘抄一段稍微靠谱一点的说明：

This field follows the syntax of cron (with minor differences). Specifically, each line is defined by:

MINUTE HOUR DOM MONTH DOW

MINUTE Minutes within the hour (0-59)

HOUR The hour of the day (0-23)

DOM The day of the month (1-31)

MONTH The month (1-12)

DOW The day of the week (0-7) where 0 and 7 are Sunday.

To specify multiple values for one field, the following operators are available. In the following examples, the value '0,30' means 0 and 30.

\* '\*' can be used to specify all valid values.

\* 'M-N' can be used to specify a range, such as "1-5"

\* 'M-N/X' or '\*/X' can be used to specify skips of X's value through the range, such as "1-5/2".

\* 'A,B,...,Z' can be used to specify multiple values, such as "0,30" or "1,3,5"

Empty lines and lines that start with '#' will be ignored as comments.

In addition, @yearly, @annually, @monthly, @weekly, @daily, @midnight, @hourly are supported.

举两个例子：

```
// every minute
* * * * *

// every 5 mins past the hour
5 * * * *
```

## 配置gradle - android专用

请ios的朋友们请飘过.

如果安装gradle插件成功的话，应该会出现下图的Invoke Gradle script，配置一下：

## 构建

**Invoke Gradle script**

(Default)

Gradle Version

Build step description

Switches

Tasks

build

编译所有版本

Root Build script

`${WORKSPACE}/...`  整个工程目录

Build File

`${WORKSPACE}/.../app/build.gradle`  app工程的build.gradle

Specify Gradle build file to run. Also, some environment variables are available to the build script

Force GRADLE\_USER\_HOME to use workspace

`${WORKSPACE}`  表示当前job下的workspace目录，主要是存放代码。更多的环境变量请参考文末附录。这样，就能自动在project下的app的build/outputs/apk下生成相应的apk。编译失败？可能要解决以下2个问题：

- gradle没配置环境变量。

比如我在/etc/profile中配置一下GRADLE\_HOME:

```
export GRADLE_HOME='/home/jay/.gradle/wrapper/dists/gradle-2.2.1-all/c64ydeuardnfqctv
export PATH=$GRADLE_HOME/bin:$PATH
```

- 找不到local.properties中sdk定义。

因为一般来说local.properties不会添加到版本库。

所以需要手动copy到 `${WORKSPACE}` 下的Project目录下（可参考自己Android Studio工程结构）。

关于local.properties的定义，这里记录一下，做个备份：

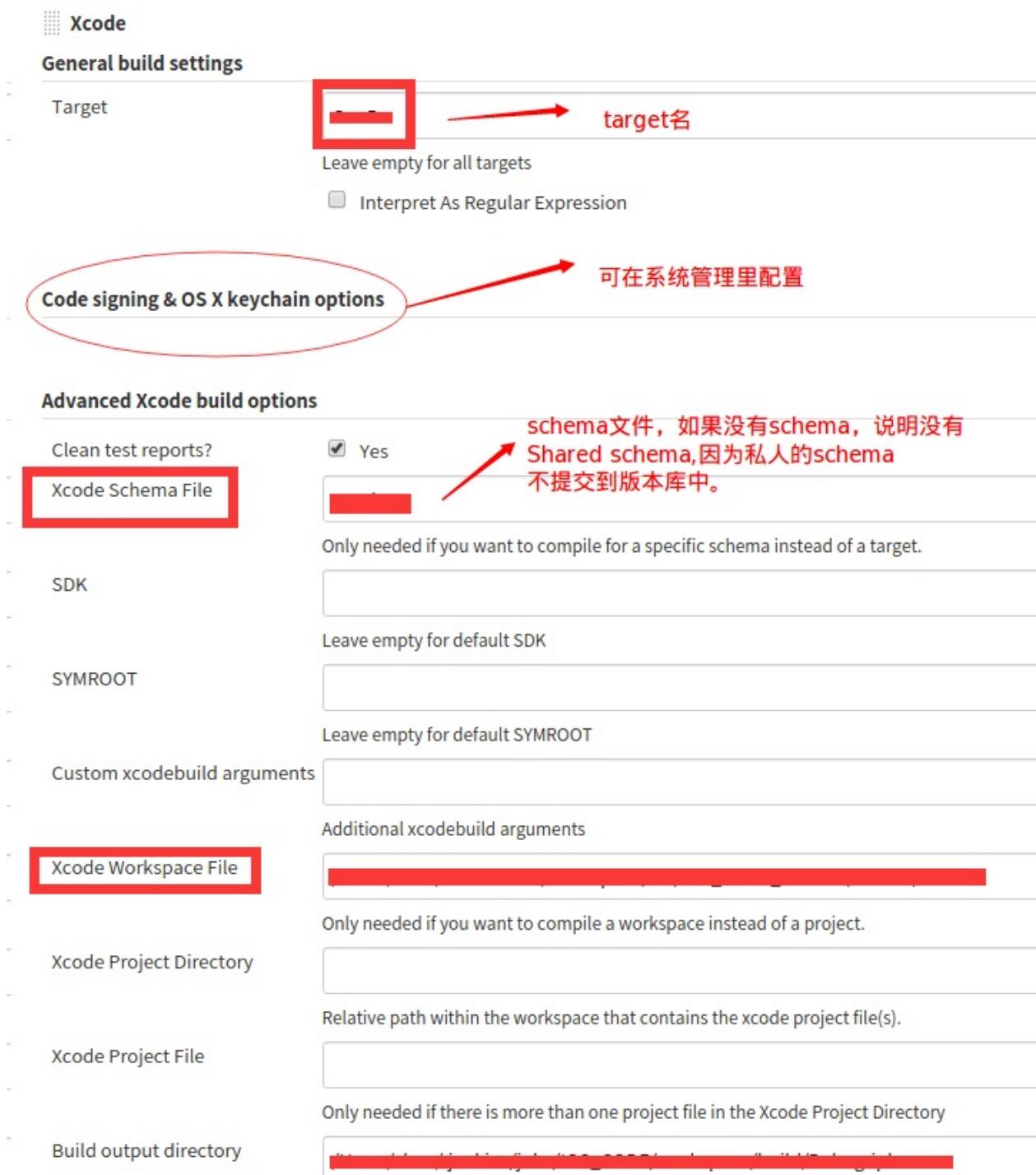
```
sdk.dir=xx/xx/android-sdk
```

再编译一般就会编译成功，当然当那些第三方库需要重新下载的话，编译可能会很慢。

## 配置xcode - ios专用

请android的同学们飘过。

安装Xcode插件后，可看到如下图界面，并配置：



这里有两个地方需要注意。

- 签名
- 需要Shared Schema文件.

# 上传到蒲公英平台

在官网文档里有说明，通过linux平台上传app的关键代码

```
curl -F "file=@/tmp/example.ipa" -F "uKey="
 -F "_api_key=" http://www.pgyer.com/api/v1/app/upload
```

具体来说，

```
先把${version}看成v1.0吧
curl -F "file=@/home/xxx/release/ganchai-release-${version}-0101-dev.apk"
 -F "uKey=231xxxxe6" -F "_api_key=0xxxx499"
 -F "publishRange=2" http://www.pgyer.com/api/v1/app/upload
```

这样就完成一个app上传到蒲公英了。

实际上，我们可能会面对更复杂的场景，比如上面的\${version}，而version定义于build.gradle如下：

```
ext {
 compileSdkVersion = 22
 buildToolsVersion = "23.0.1"
 minSdkVersion = 10
 targetSdkVersion = 22
 versionCode = 1111
 versionName = "v1.2.0.0"
}
```

得想办法读到versionName，然后拼出最终的文件名，这样下次版本升级了之后也能动态的上传app到蒲公英了。

```
使用sed命令读取，使用cut切割，最终动态读取到versionName
version=`sed -n '21,1p' ${WORKSPACE}/xxx/build.gradle | cut -c20-27`
```

这是android的apk上传过程，相应的，ios是上传ipa，方法是一样的，不再赘述。

## 小结

把开发人员发布版本的工作自动化之后，如此一来，方便了测试人员随时拉取并构建最新版本，更解放了开发人员自己的发版本的工作，一个字，善！

# 附录

jenkins中定义的那些环境变量：

```
The following variables are available to shell scripts

BUILD_NUMBER
The current build number, such as "153"
BUILD_ID
The current build id, such as "2005-08-22_23-59-59" (YYYY-MM-DD_hh-mm-ss)
BUILD_DISPLAY_NAME
The display name of the current build, which is something like "#153" by default.
JOB_NAME
Name of the project of this build, such as "foo" or "foo/bar". (To strip off folder p
BUILD_TAG
String of "jenkins-${JOB_NAME}-${BUILD_NUMBER}". Convenient to put into a resource fil
EXECUTOR_NUMBER
The unique number that identifies the current executor (among executors of the same ma
NODE_NAME
Name of the slave if the build is on a slave, or "master" if run on master
NODE_LABELS
Whitespace-separated list of labels that the node is assigned.
WORKSPACE
The absolute path of the directory assigned to the build as a workspace.
JENKINS_HOME
The absolute path of the directory assigned on the master node for Jenkins to store da
JENKINS_URL
Full URL of Jenkins, like http://server:port/jenkins/ (note: only available if Jenkins
BUILD_URL
Full URL of this build, like http://server:port/jenkins/job/foo/15/ (Jenkins URL must b
JOB_URL
Full URL of this job, like http://server:port/jenkins/job/foo/ (Jenkins URL must be set
SVN_REVISION
Subversion revision number that's currently checked out to the workspace, such as "12345"
SVN_URL
Subversion URL that's currently checked out to the workspace.
```

# 如何在Jenkins发送的构建邮件中提供版本包的下载

来源:测试蜗牛,一步一个脚印

1、首先每次构建都将版本包放到一个目录下存储下来,如使用批处理命令将Maven打包的war包拷贝到History文件夹下存储。

在Job中增加批处理命令:

```

@echo off
echo 包重命名
cd D:\CI\jenkins-1.620\workspace\Build_HBGMS_framework\framework\target\
rename framework-1.0-SNAPSHOT.war framework.war
 ↗ 重命名

echo 拷贝版本包到版本下载目录
echo 拷贝
xcopy D:\CI\jenkins-1.620\workspace\Build_HBGMS_framework\framework\target\framework.war "D:\CI\jenkins-1.620\workspace\Build_History\Build_HBGMS_framework\Build_%BUILD_NUMBER%\framework" /y
 ↗ 拷贝到版本 History 目录下

echo 将构建包拷贝到 deploy 的目录下
echo 先清空 deploy 下面的版本包
rd D:\CI\jenkins-1.620\workspace\Deploy_HBGMS_framework\framework*/s/q /
echo 拷贝到 deploy 目录下
xcopy D:\CI\jenkins-1.620\workspace\Build_HBGMS_framework\framework\target\framework.war D:\CI\jenkins-1.620\workspace\Deploy_HBGMS_framework\framework\ /y
echo 拷贝结束
 ↗ 拷贝到 deploy 目录下,给 deploy 的 job 使用

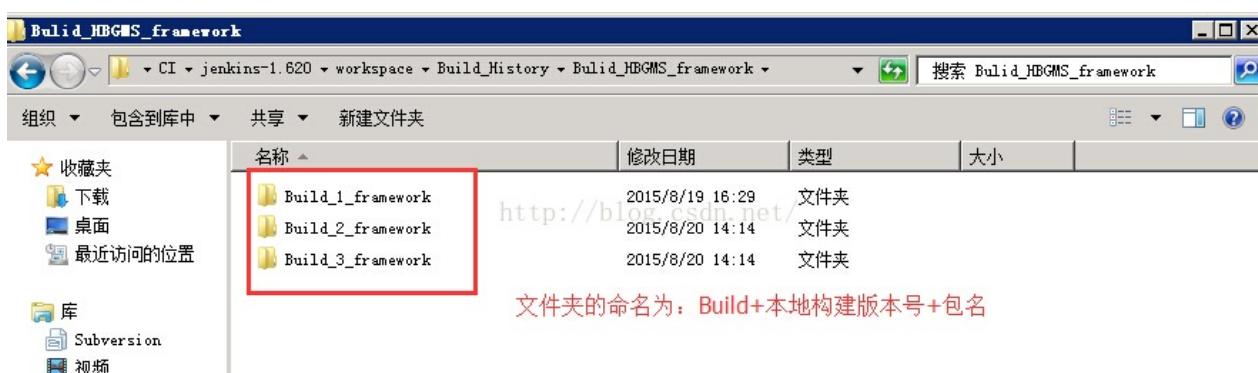
```

参阅 可用环境变量列表

删除

保存 应用

History目录结构如下:



2、将文件夹Build\_History文件夹发布

参考文章《如何发布版本包远程下载》。我使用的是IIS站点发布功能。

3、在邮件中增加版本地址超级链接

**Editable Email Notification**

Disable Extended Email Publisher

Allows the user to disable the publisher, while maintaining the settings

Project Recipient List

Project Reply-To List

Content Type

Both HTML and Plain Text

Default Subject

http://blog.esdn.net/  
\$DEFAULT SUBJECT - \$BUILD\_STATUS

Default Content

```
<tr><td>版本号: ${BUILD_NUMBER}</td>
<td>构建结果为: ${BUILD_STATUS}</td>
</tr>
<tr>
<td>构建原因: </td>
<td> ${CAUSE}</td>
</tr>
<tr>
<td colspan="2">点击获取版本包名</td>
</tr>
```

目标为：Build\_版本号\_包名

邮件的代码为如下：

|                                                                                                                                                                                     |                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|
| <b>版本号:</b>                                                                                                                                                                         | \${BUILD_NUMBER} |
| <b>构建结果为:</b>                                                                                                                                                                       | <b>构建原因:</b>     |
| <b>构建原因:</b> \${CAUSE}                                                                                                                                                              |                  |
| <b>点击获取版本包名</b>                                                                                                                                                                     |                  |
| <a href="http://192.168.100.15:8018/Bulid_HBGMS_framework/Build_\${BUILD_NUMBER}_framework/">http://192.168.100.15:8018/Bulid_HBGMS_framework/Build_\${BUILD_NUMBER}_framework/</a> |                  |
| <b>点击查看构建的详细信息</b>                                                                                                                                                                  |                  |

# LeakCanary

# 用 LeakCanary 检测内存泄漏

来源:[realm.io](#)

我们的 App 曾经遇到很多的内存泄漏导致 OutOfMemoryError 的崩溃，一些甚至是在生产环境。Square 的 Pierre-Yvews Ricau 开发了 LeakCanary 最终解决了这些问题。LeakCanary 是一个帮助你检测和修复内存泄漏的工具。在这个分享中，Pierre 教授大家如何修复内存泄漏的错误，让你的 App 更稳定和可靠。

[视频链接](#)

About the Author: Pierre-Yves Ricau

Pierre-Yves Ricau 是 Square 的一枚员工，享受编程和优质的代码

[@piwai Website](#)

Save the date for [Droidcon SF](#) in March — a conference with best-in-class presentations from leaders in all parts of the Android ecosystem.

## 介绍 (0:00)

大家好，我是 Pierre-Yvews Ricau (叫我 PY 就行)，现在在 Square 工作。

Square 出了一款名为：[Square Register](#) 的 App，帮助你用移动设备完成支付。在用这个 App 的时候，用户先要登陆他的个人账号。

不幸的是，在签名页面有的时候会因为内存溢出而出现崩溃。老实说，这个崩溃来的太不是时候了 — 用户和商家都无法确认交易是否完成了，更何况是在和钱打交道的时候。我们也强烈的意识到，我们需要处理下内存溢出或者内存泄露这种事情了。

## 内存泄漏：非技术讲解 (1:40)

我想要聊的内存泄露解决方案是：[LeakCanary](#)。LeakCanary 是一个可以帮助你发现和解决内存泄露的开源工具。但是到底什么是内存泄露呢？我们从一个非技术角度来开始，先来举个例子。

假设我的手代表着我们 App 能用的所有内存。我的手里能放很多东西。比如：钥匙，Android 玩偶等等。设想我的 Android 玩偶需要扬声器才能工作，而我的扬声器也需要依赖 Android 玩偶才能工作，因此他们持有彼此的引用。

我的手里可以持有如此多的东西。扬声器依附到 Android 玩偶上会增加总重量，就像引用会占用内存一样。一旦我放弃了我的玩偶，把他扔到地上，会有垃圾回收器来回收掉它。一旦所有的东西都进了垃圾桶，我的手又轻便了。

不幸的是，有的时候，一些不好的情况会发生。比如：我的钥匙没准和我的 Android 玩偶黏在了一起，阻止我把 Android 玩偶扔到地上。最终的结果就是 Android 玩偶无论如何都不会被回收掉。这就是内存泄露。

有外部的引用（钥匙，扬声器）指向了本不应该再指向的对象（Andorid 玩偶）。类似这样的小规模的内存泄露堆积以后就会造成大麻烦。

## LeakCanary to the Rescue (3:47)

这就是我们为什么要开发 LeakCanary。

我现在可能已经清楚了 可被回收的 Android 对象应该及时被销毁。

但是我还是没法清楚饿看到这些对象是否已经被回收掉。有了 LeakCanary 以后，我们给可被回收的 Android 对象上打了智能标记。智能标记能知道他们所指向的对象是否被成功释放掉。如果过一小段时间对象依然没有被释放，他就会给内存做个快照。

LeakCanary 随后会把结果发布出来，帮助我们看到内存到底怎么泄露了，清晰的展示无法被释放的对象的引用链。

举个具体的例子：在我们的 Square App 里的签名页面。用户准备签名的时候，App 因为内存溢出出错崩溃了。我们不能确认内存错误到底出在哪儿了。

签名页面持有了一个很大的有用户签名的 Bitmap 图片对象。图片的大小和用户手机屏幕大小一致 — 我们猜测这个有可能会造成内存泄露。首先，我们可以配置 Bitmap 为 alpha 8-bit 来节省内存。这是很常见的一种修复方案，而且效果也不错。但是并没有彻底解决问题，只是减少了泄露的内存总量。但是内存泄露依然在哪儿。

最主要的问题是我们 App 的堆满了，应该要留有足够的空间给我们的签名图片，但是由于很多处的内存泄露叠加在一起占用了很多内存。

## 技术讲解内存泄漏 (8:06)

假设，我有一个 App，这个 App 点一下就能买一个法棍面包（哈哈，可能只有法国人需要这样的App，没错，我就是法国人）。

```
private static Button buyNowButton;
```

由于某种原因，我把这个 button 设置成了 static 的。问题随之而来，这个按钮除非你设置成了null，不然就内存泄露了。

你也许会说：“只是一个按钮而已，没啥大不了”。问题是这个按钮还有一个成员变量：叫“mContext”，这个东西指向了一个 Acitvity，Acitvty 又指向了一个 Window，Window 有拥有整个 View 继承树。算下来，那可是一大段的内存空间。

静态的变量是 [GC root](#) 类型的一种。垃圾回收器会尝试回收所有非 GC root 的对象，或者某些被 GC root 持有的对象。所以如果你创建一个对象，并且移除了这个对象的所有指向，他就会被回收掉。但是一旦你将一个对象设置成 GC root，那他就不会被回收掉。

当你看到类似“法棍按钮”的时候，很显然这个按钮持有了一个 Activity 的引用，所以我们必须清理掉它。当你沉浸在你的代码的时候，你肯定很难发现这个问题。你可能只看到了引出的引用。你可以知道 Activity 引用了一个 Window，但是谁引用了 Activity？

你可以用像 IntelliJ 这样的工具做些分析，但是它并不会告诉你所有的东西。通常，你可以把这些 Object 的引用关系组织成图，但是是个单向图。

## 分析堆 (10:16)

我们能做些什么呢？我们来做个快照。我们拿出所有的内存然后导出到文件里，这个文件会被用来分析和解析堆结构。其中一个工具叫做 Memory Analyzer，也叫 MAT。它会通过 dump 的内存，然后分析所有存活在内存中的对象和类。你可以用 SQL 对他做些查询，类似如下：

```
SELECT * FROM INSTANCEOF android.app.Activity a WHERE a.mDestroyed = true
```

这条语句会返回所有的状态为 destroyed 的实例。一旦你发现了泄露的 Activity，你可以执行 `merge_shortest_paths` 的操作来计算出最短的 GC root 路径。从而找出阻止你 Acitvty 释放的那个对象。

之所以说要“最短路径”，是因为通常从一个 GC root 到 Acitvty，有很多条路径可以到达。比如说：我的按钮的 parent view，同样也持有一个 mContext 对象。

当我们看到内存泄露的时候，我们通常不需要去查看所有的这些路径。我们只需要最短的一条。那样的话，我们就排除了噪音，很快的找到问题所在。

## LeakCanary 救你于水火之中 (12:04)

有 MAT 这样一个帮我们发现内存泄露的工具是个很棒的事情。但是在一个正在运行的 App 的上下文中，我们很难像我们的用户发现泄露那样发现问题所在。我们不能要求他们在做一遍相同操作，然后留言描述，再把 70 多 MB 的文件发回给我们。我们可以在后台做这个，但是并不 Cool。我们期望的是，我们能够尽早的发现泄露。比如在我们开发的时候就发现这些问题。这也是 LeakCanary 诞生的意义。

一个 Activity 有自己生命周期。你了解它是如何被创建的，如何被销毁的，你期望他会在 `onDestroy()` 函数调用后，回收掉你所有的空闲内存。如果你有一个能够检测一个对象是否被正常的回收掉了的工具，那么你就会很惊讶的喊出：“这个可能造成内存泄露！仍然没有被垃圾回收掉，它本该被回收掉的！”

Activity 无处不在。很多人都把 Activity 当做神级 Object 一般的存在，因为它可以操作 Services，文件系统等等。经常会发生对象泄漏的情况，如果泄漏对象还持有 context 对象，那 context 也就跟着泄漏了。

```
public class MyActivity extends Activity {

 @Override protected void onDestroy() {
 super.onDestroy();
 // instance should be GCed soon.
 }
}
```

```
Resources resources = context.getResources();
LayoutInflater inflater = LayoutInflater.from(context);
File filesDir = context.getFilesDir();
InputMethodManager inputMethodManager =
(InputMethodManager) context.getSystemService(Context.INPUT_METHOD_SERVICE);
```

## LeakCanary API Walkthrough (13:32)

我们回过头来再看看智能标记（smart pin），我们希望知道的是当生命后期结束后，发生了什么。幸运的时，LeakCanary 有一个很简单的 API。

第一步：创建 RefWatcher。给 RefWatcher 传入一个对象的实例，它会检测这个对象是否被成功释放掉。

```
public class ExampleApplication extends Application {

 public static RefWatcher getRefWatcher(Context context) {
 ExampleApplication application = (exampleApplication) context.getApplication();
 return application.refWatcher;
 }

 private RefWatcher refWatcher;

 @Override public void onCreate () {
 super.onCreate();
 // Using LeakCanary
 refWatcher = LeakCanary.install(this);
 }
}
```

第二步：监听 Activity 生命周期。然后，当 onDestroy 被调用的时候，我们传入 Activity。

```
public ActivityRefWatcher(Application application, final RefWatcher refWatcher) {
 this.application = checkNotNull(application, "application");
 checkNotNull(refWatcher, "androidLeakWatcher");
 lifecycleCallbacks = new ActivityLifecycleCallbacksAdapter() {
 @Override public void onActivityDestroyed(Activity activity) {
 refWatcher.watch(activity);
 }
 };
}

public void watchActivities() {
 // Make sure you don't get installed twice.
 stopWatchingActivities();
 application.registerActivityLifecycleCallbacks(lifecycleCallbacks);
}
```

## What are Weak References (14:17)

想要了解这个是怎么工作的，我得先跟大家聊聊弱引用（weak reference）。我刚才提到过静态域的变量会持有Activity 的引用。所以刚才说的“下单”按钮就会持有 mContext 对象，导致 Activity 无法被释放掉。这个被称作强引用（strong reference）。在垃圾回收过

程中，你可以对一个对象有很多的强引用。当这些强引用的个数总和为零的时候，垃圾回收器就会释放掉它。

弱引用，就是一种不增加引用总数的持有引用方式。垃圾回收期是否决定要回收一个对象，只取决于它是否还存在强引用。所以说，如果我们将我们的 Activity 持有为弱引用，一旦我们发现弱引用持有的对象已经被销毁了，那么这个 Activity 就已经被垃圾回收器回收了。否则，那可以大概确定这个 Activity 已经被泄露了。

```
private static Button buyNowButton;
Context mContext;
```

```
WeakReference<T>
/** Treated specially by GC. */
T referent;
```

```
public class Baguette Activity
extends Activity {

@Override protected void onCreate(Bundle state) {
 super.onCreate(state);
 setContentView(R.layout.activity_main);
}
}
```

弱引用的主要目的是为了做 Cache，而且非常有用。主要就是告诉 GC，尽管我持有了这个对象，但是如果一旦没有对象在用这个对象的时候，GC 就可以在需要的时候销毁掉。

在下面的例子中，我们继承了 WeakReference：

```
final class KeyedWeakReference extends WeakReference<Object> {
 public final String key; // (1) Unique identifier
 public final String name;

 KeyedWeakReference(Object referent, String key, String name, ReferenceQueue<Object>
 super(checkNotNull(referent, "referent"), checkNotNull(referenceQueue, "referenceQueue"),
 this.key = checkNotNull(key, "key");
 this.name = checkNotNull(name, "name");
 }
}
```

你可以看到，我们给弱引用添加了一个 Key，这个 Key 是一个唯一字符串。想法是这样的：当我们解析一个 heap dump 文件的时候，我们可以询问所有的 KeyedWeakReference 实例，然后找到对应的 Key。

首先，我们创建一个 weakReference，然后我们写入『一会儿，我需要检查弱引用』。（尽管一会儿可能就是几秒后）。当我们调用 watch 函数的时候，其实就是发生了这些事情。

```
public void watch(Object watchedReference, String referenceName) {
 checkNotNull(watchedReference, "watchedReference");
 checkNotNull(referenceName, "referenceName");
 if (debuggerControl.isDebuggerAttached()) {
 return;
 }
 final long watchStartNanoTime = System.nanoTime();
 String key = UUID.randomUUID().toString();
 retainedKeys.add(key);
 final KeyedWeakReference reference =
 new KeyedWeakReference(watchedReference, key, referenceName, queue);

 watchExecutor.execute(() -> { ensureGone(reference, watchStartNanoTime); });
}
```

在这一切的背后，我们调用了 System.GC — 免责声明— 我们本不应该去做这件事情。然而，这是一种告诉垃圾回收器：『Hey，垃圾回收器，现在是一个不错的清理垃圾的时机。』，然后我们再检查一遍，如果发现有些对象依然存活者，那么可能就有问题了。我们就要触发 heap dump 操作了。

## HAHA! (16:55)

亲手做 heap dump 是件超酷的事情。当我亲手做这些的时候，花了很多时间和功夫。我每次都是做相同的操作：下载 heap dump 文件，在内存分析工具里打开它，找到实例，然后计算最短路径。但是我很懒，我根本不想一次次的做这个。（我们都懒对吧，因为我们是开发者啊！）

我本可以为内存分析器写一个 Eclipse 插件，但是 Eclipse 插件机制太糟糕了。后来我灵机一动，我其实可以把某个 Eclipse 的插件，移除 UI，利用它的代码。

HAHA 是一个无UI Android 内存分析器。基本上就是把另一个人写的代码重新打包。开始的时候，我就是 fork 了一份别的代码然后移除了UI部分。两年前，有人重新fork了我的代码，然后添加了 Android 支持。又过了两年，我发现这个人的仓储，然后我又重新打包上传到了 maven center。

我最近根据 Android Studio 修改了代码实现。代码还说的过去，还会继续维护。

## LeakCanary 的实现 (19:19)

我们有自己的库去解析 heap dump 文件，而且实现的很容易。我们打开 heap dump，加载进来，然后解析。然后我们根据 key 找到我们的引用。然后我们根据已有的 Key 去查看拥有的引用。我们拿到实例，然后得到对象图，再反向推导发现泄漏的引用。

所有的工作实际上都发生在 Android 设备上。当 LeakCanary 探测到一个 Activity 已经被销毁掉，而没有被垃圾回收器回收掉的时候，它就会强制导出一份 heap dump 文件存在磁盘上。然后开启另外一个进程去分析这个文件得到内存泄漏的结果。如果在同一进程做这件事的话，可能会在尝试分析堆内存结构的时候而发生内存不足的问题。

最后，你会得到一个通知，点击一下就会展示出详细的内存泄漏链。而且还会展示出内存泄漏的大小，你也会很明确自己解决掉这个内存泄漏后到底能够解救多少内存出来。

LeakCanary 也是支持 API 的，这样你就可以挂载内存泄漏的回调，比方说可以把内存泄漏问题传到服务器上。在 Square 我们用了 Slack 的 API，在测试阶段出现内存泄漏的时候，它就会通知我们。

```
@Override protected void onLeakDetected(HeapDump heapDump, AnalysisResult result) {
 String name = classSimpleName(result.className);
 String title = name + " has leaked";
 slackUploader.uploadHeapDumpBlocking(heapDump.headDumpFile, title, result.leakTrace
 MEMORY_LEAK_CHANNEL);
}
```

用上 API 以后，我们的程序崩溃率降低了 94%！简直棒呆！

## Debug 一个真实的例子 (22:12)

这里有个例子，是来自 AOSB 的一个内存泄漏的代码。假设我们一个 App，包含了一个 undobar，你在点击某些按钮的时候 undobar 会消失掉。

```

public class MyActivity extends Activity {

 @Override protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);

 find ViewbyID(R.id.button).setOnClickListener(new View.OnClickListener() {
 @Override public void onClick(View v) {
 removeUndoBar();
 }
 });
 }

 private void removeUndoBar() {...}

 private void checkUndoBarGCed(ViewGroup undoBar) {...}
}

```

我们把所有的 View 都存起来，然后设置了一个 Layout 的 Transition 动画（1）。我们还给 undobar 增加了一个 View 进去（2）。然后我们从 parent layout 移除了 undobar（3）。

```

public class MyActivity extends Activity {

 @Override protected void onCreate(Bundle savedInstanceState) {...}

 private void removeUndoBar() {
 ViewGroup rootLayout = (ViewGroup) findViewById(R.id.root);
 ViewGroup undoBar = (ViewGroup) findViewById(R.id.undo_bar);

 undoBar.setLayoutTransition(new LayoutTransition()); // (1)

 // (2)
 View someView = new View(this);
 undoBar.addView(someView);

 rootLayout.removeView(undoBar); // (3)

 checkUndoBarGCed(undoBar);
 }

 private void checkUndoBarGCed(ViewGroup undoBar) {...}
}

```

现在看来，没有任何对象指向 undobar 了，所以说它应该被回收掉。否则的话...我们可能就遇到麻烦了。

我们用 LeakCanary 的 API 告诉系统：『Hey，现在该回收掉 undobar 了，几秒后检查下 undobar 是否被回收掉了』：

```
public class MyActivity extends Activity {

 @Override protected void onCreate(Bundle savedInstanceState) {...}

 private void removeUndoBar() {...}

 private void checkUndoBarGCed(ViewGroup undoBar) {
 RefWatcher watcher = MyApplication.from(this).getWatcher();
 watcher.watch(undoBar); // (1)
 }
}
```

我们删除了 undobar，但是... LeakCanary 好像不高兴了，它发现了内存泄露，返回的报告如下：

```
static InputMethodManager.sInstance

references
InputMethodManager.mCurRootView

references
PhoneWindow$DecorView.mAttachInfo

references View$AttachInfo.mTreeObserver

references
ViewTreeObserver.mOnPreDrawListeners

references
ViewTreeObserver$CopyOnWriteArray mData

references LayoutTransition$1.val$parent

leaks FrameLayout instance
```

不难发现，静态的InputMethodManager有一个引用指向了当前的 root view：mCurRootView。mCurRootView 是当前窗口所有 View 的父容器。Root view 有一个叫做 TreeObserver 的对象。TreeObserver 是用在布局改变时候的做回调，通知监听布局改变的 listener 的，一个 View 关系树上只有一个 TreeObserver，所以你会看到它有很多的 PreDrawListener。这就意味着你每添加一个 PreDrawLisener，当布局改变的时候就会发起一次回调。

到现在为止看起来没有什么异常。但是不难发现某个 PreDrawListener 持有一个 LayoutTransition\$1，这种写法是 Java 表示匿名类的一种写法，意味着这是第一个在 LayoutTransition 里定义的匿名类。然后你还会看到有一个叫做 val\$parent 的变量指向了我们泄露的 undoBar。val\$parent 意思就是这是在匿名类外声明的 final 类型的临时变量，变量名为 parent。

我们继续来看：

```
android.animation.LayoutTransition#runChangeTransition

// This is the cleanup step. When we get this rendering event, we know that all of
// the appropriate animations have been set up and run. Now we can clear out the
// layout listeners.
observer.addOnPreDrawListener(new ViewTreeObserver.OnPreDrawListener() {
 public boolean onPreDraw () {
 parent.getViewTreeObserver().removeOnPreDrawListener(this);
 // ... More code
 return true;
 }
});
```

这是 Android 源码的一部分 — LayoutTransition 是一个 Android 的类。这里的 Observer 是 ViewTreeObserver。一切都看起来没啥问题，注册了一个 ViewTreeObserver，然后马上在第一个回调里解除掉注册。这个按理说不应该出问题，但是到底发生了什么？我们来看看 getViewTreeObserver。

```
public ViewTreeObserver getViewTreeObserver() {
 if (mAttachInfo != null) {
 return mAttachInfo.mTreeObserver; // (1)
 }
 if (mFloatingTreeObserver == null) {
 mFloatingTreeObserver = new ViewTreeObserver();
 }
 return mFloatingTreeObserver; // (2)
}
undoBar.setLayoutTransition(new LayoutTransition()); // (3)

View someView = new View(this);
undoBar.addView(someView); // (4)

rootLayout.removeView(undoBar); // (5)
```

getViewTreeObserver 函数首先判断 attached 状态，如果是，则返回一个 view 树，即 ViewTreeObserver。否则，返回一个临时的 ViewTreeObserver。

问题是如果我的 View 被 detach 掉了，我将会得到一个假的 ViewTreeObserver，而非一个真实的 ViewTreeObserver。你会发现，我们设置了 LayoutTransition (3)，然后增加了一个 view (4)。这个触发了 addOnPreDrawListener 监听器。然后我们移除掉了 undo-bar，这就意味着 undobar 无法再访问 ViewTreeObserver 了。

```
final ViewTreeObserver observer = parent.getViewTreeObserver(); // used for later cleanup
if (!observer.isAlive()) {
 // If the observer's not in a good state, skip the transition
 return;
}

public void onAnimationEnd(Animator animator) {
 ...
 // layout listeners.

 observer.addOnPreDrawListener(new ViewTreeObserver.OnPreDrawListener() {
 public boolean onPreDraw() {
 observer.removeOnPreDrawListener(this);
 parent.getViewTreeObserver().removeOnPreDrawListener(this);
 }
 });
}
```

它得到了一个假的 ViewTreeObserver，并且无法移除自己，因为他并不在这个假的 ViewTreeObserver 里。

这是 Android 4 年前的一次代码修改留下的问题，当时是为了修复另一个 bug，然而带来了无法避免的内存泄漏。我们也不知道何时能被修复。

## 忽略 Android SDK Crashes (28:10)

通常来说，总是有些内存泄漏是你无法修复的。我们某些时候需要忽略掉这些无法修复的内存泄漏提醒。在 LeakCanary 里，有内置的方法去忽略无法修复的问题。

```
LAYOUT_TRANSITION(SDK_INT >= ICE_CREAM_SANDWICH && SDK_INT <= LOLLIPOP_MR1) {
 @Override void add (ExcludedRefs.Builder excluded) {
 // LayoutTransition leaks parent ViewGroup through ViewTreeObserver.OnPreDrawListener.
 // When triggered, this leak stays until the window is destroyed.
 // Tracked here: https://code.google.com/p/android/issues/detail?id=171830
 excluded.instanceField("android.animation.LayoutTransition$1", "val$parent");
 }
}
```

我想要重申一下，LeakCanary 只是一个开发工具。不要将它用到生产环境中。一旦有内存泄漏，就会展示一个通知给用户，这一定不是用户想看到的。

我们即便用上了 LeakCanary 依然有内存溢出的错误出现。我们的内存泄露依然有多个。有没有办法改变这些呢？

## LeakCanary 的未来 (29:14)

```
public class OomExceptionHandler implements Thread.UncaughtExceptionHandler {
 private final Thread.UncaughtExceptionHandler defaultHandler;
 private final Context context;

 public OomExceptionHandler(Thread.UncaughtExceptionHandler defaultHandler, Context context) {
 this.defaultHandler = defaultHandler;
 this.context = context;
 }

 @Override public void uncaughtException(Thread thread, Throwable ex) {
 if (containsOom(ex)) {
 File heapDumpFile = new File(context.getFilesDir(), "out-of-memory.hprof");
 try {
 Debug.dumpHprofData(heapDumpFile.getAbsolutePath());
 } catch (Throwable ignored) {}
 }
 defaultHandler.uncaughtException(thread, ex);
 }

 private boolean containsOom(Throwable ex) {...}
}
```

这是一个 Thread.UncaughtExceptionHandler，你可以将线程崩溃委托给它，它会导出 heap dump 文件，并且在另一个进程里分析内存泄漏情况。

有了这个以后，我们就能做一些好玩儿的事情了，比如：列出所有的应该被销毁却依然在内存里存活的 Activity，然后列出所有的 Detached View。我们可以依此来为泄漏的内存按重要性排序。

我实际上已经有一个很简单的 Demo 了，是我在飞机上写的。还没有发布，因为还有些问题，最严重的问题是没有足够的内存去解析 heap dump 文件。想要修复这个问题，得想想别的办法。比如采用 stream 的方法去加载文件等等。

## Q&A (31:50)

**Q:** LeakCanary 能用于 Kotlin 开发的 App?

**PY:** 我不知道，但是应该是可以的，毕竟到最后他们都是字节码，而且 Kotlin 也有引用。

**Q:**你们是在 Debug 版本一直开启 LeakCanary 么？还是只在最后的某些版本开启做做测试

**PY:** 不同的人有不同的方法，我们通常是一直都开着的。

## 引用

### GC Root

The so-called GC (Garbage Collector) roots are objects special for garbage collector. Garbage collector collects those objects that are not GC roots and are not accessible by references from GC roots.

There are several kinds of GC roots. One object can belong to more than one kind of root. The root kinds are:

- **Class** - class loaded by system class loader. Such classes can never be unloaded. They can hold objects via static fields. Please note that classes loaded by custom class loaders are not roots, unless corresponding instances of `java.lang.Class` happen to be roots of other kind(s).
- **Thread** - live thread
- **Stack Local** - local variable or parameter of Java method
- **JNI Local** - local variable or parameter of JNI method
- **JNI Global** - global JNI reference
- **Monitor Used** - objects used as a monitor for synchronization
- **Held by JVM** - objects held from garbage collection by JVM for its purposes.  
Actually the list of such objects depends on JVM implementation. Possible known cases are: the system class loader, a few important exception classes which the JVM knows about, a few pre-allocated objects for exception handling, and custom class loaders when they are in the process of loading classes. **Unfortunately, JVM provides absolutely no additional detail for such objects. Thus it is up to the analyst to decide to which case a certain "Held by JVM" belongs.**

# Lint

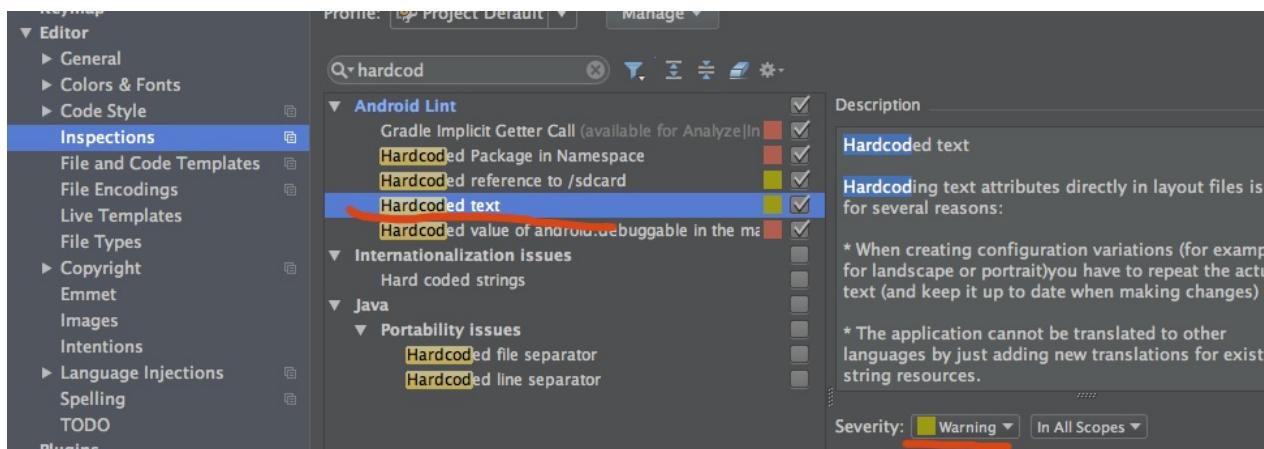
# AndroidStudio & lint 代码检查设置

来源:简书:楚云之南

AndroidStudio作为google官方推荐的Android开发IDE, 提供了一整套强大的静态代码分析工具, 使用它们可以很好地帮助我们进行更加规范的开发。从一个常见的场景入手吧。

几乎所有的开发团队的代码规范里面都有这么一条: 不允许在布局文件中进行 `hardcode`, 原因参加见: [stackoverflow](#)

为了达到上面的目的, 我们可以通过设置AS的 `code inspections` 来设置静态代码检查的规则, 找到`hardcode`的配置:



可以看到默认的 `hardcode` 配置的 `severity` 是警告, 所以我们在xml中直接写字符串时, 将光标放到去可以看到警告提示:

```
<TextView
 android:id="@+id/tv_error_details"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_marginTop="6dp"
 android:text="error" />
</LinearLayout>
```

[I18N] Hardcoded string "error", should use @string resource more... (⌘F1)

但是这个提示也太弱了吧, 我们将'severity'提升到error试试:

```
<TextView
 android:id="@+id/tv_error_details"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_marginTop="6dp"
 android:text="error" />
```

注意，提升这个地方的等级不会对代码和其他静态分析工具如lint产生影响，也不会对运行产生影响，它的作用域仅仅是IDE展示

同样的，我们还可以设置很多其他的IDE静态代码检查，通过改变其severity达到更直观提示的作用，可以让开发者有一个直观的认识，哪些代码是合法但不合规的。如果有人不按照这个约束进行开发，那么代码中到处都是红色的错误(额，希望他不是个处女座..)。如果说通过IDE的code inspections是进行自律的话(实际上这个配置也是个人的行为)，那么Android提供的另外一个静态代码工具lint就是一种对别人的约束了。

lint是Android提供的一个静态代码检查的工具，我们可以在gradle的构建task中加入link检查。具体的使用请移步到goole文档。link能够检查的东西很多，参考 [所有check issue](#)。

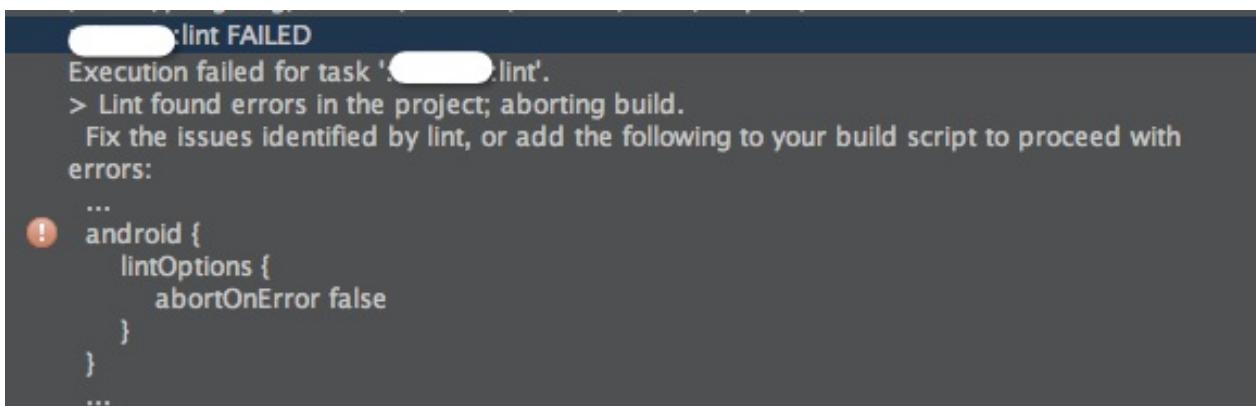
还是上面的场景，如果我的需求是代码中存在hardcode,那么所有人的代码都编译不通过(现实中不可能这么变态)。

lint工具可以通过一个xml文件来配置，它可以用来修改某些check issue是否忽略(典型的例子是第三方库里面存在问题)，同时可以修改某些issue的默认等级。

**HardcodedText**的默认等级是警告，我们升级成error，并在配置文件中增加

```
lintOptions {
 lintConfig file("lint.xml")
 abortOnError true
}
```

，这样在我们构建项目的时候如果发生错误，将直接中断构建。



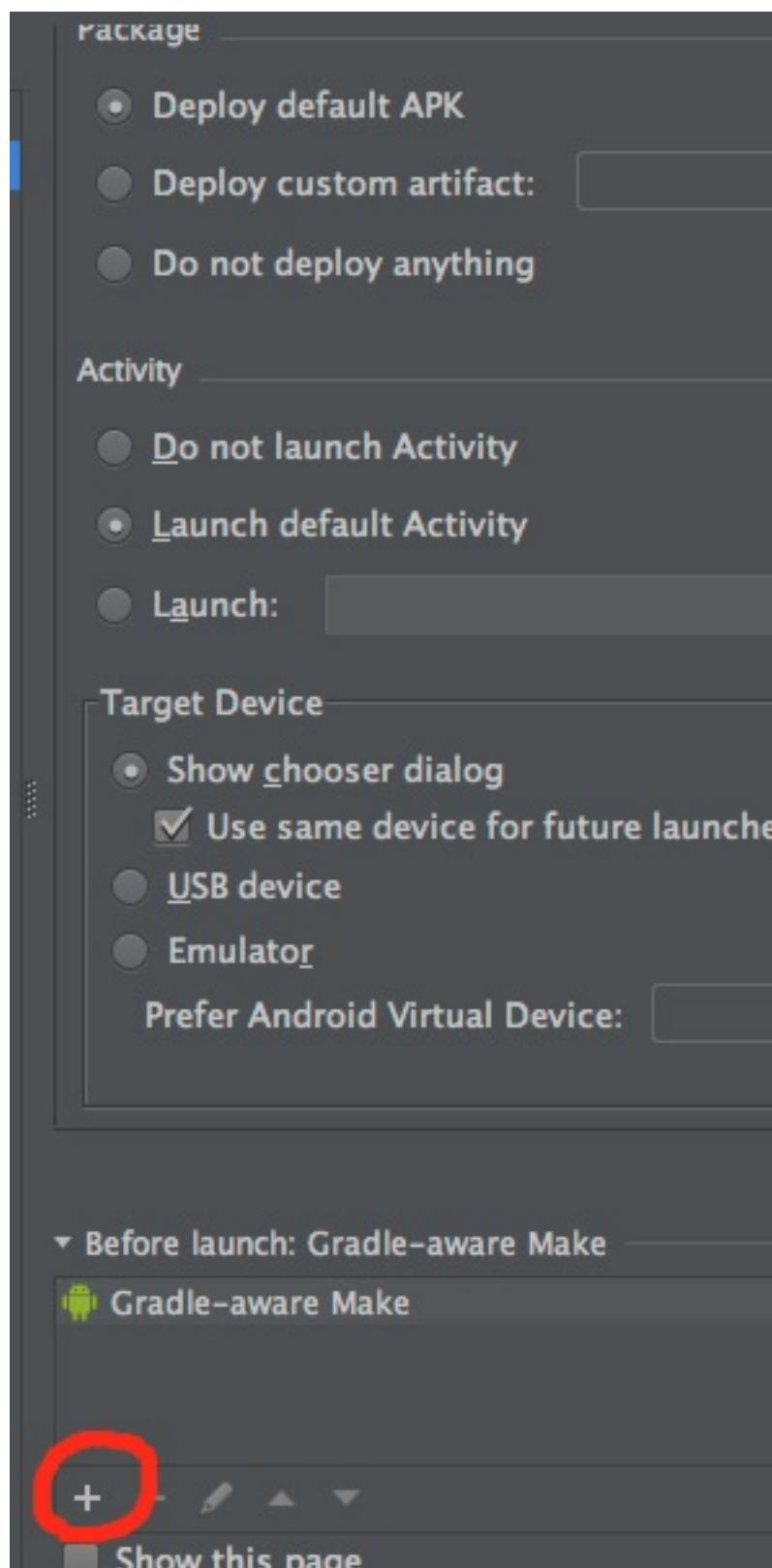
The screenshot shows a terminal window with the following output:

```
:lint FAILED
Execution failed for task ':app:lint'.
> Lint found errors in the project; aborting build.
Fix the issues identified by lint, or add the following to your build script to proceed with errors:
...
! android {
 lintOptions {
 abortOnError false
 }
}
...
```

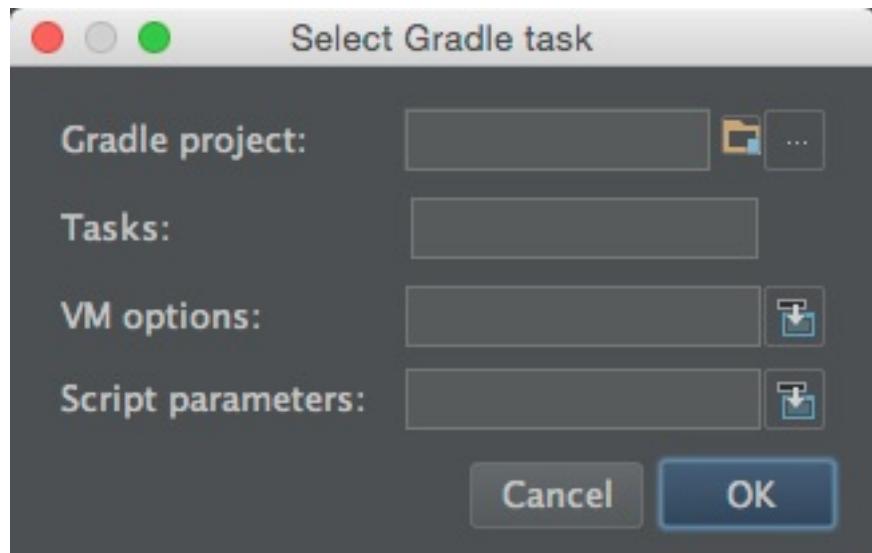
The terminal title bar indicates the build failed with the message "lint FAILED". The error message states that execution failed because Lint found errors in the project and is aborting the build. It suggests fixing the issues or adding code to the build script to proceed despite errors. A portion of the build.gradle file is shown at the bottom, specifically the android block with lintOptions set to abortOnError false.

需要注意的是，点击AS上面的绿色的运行按钮是不会触发lint检查的，如果你想每次点击运行都进行lint检查，可以如下设置：





添加 run gradle task



task填lint即可，这样在点击AS上的运行时，会自动执行lint，不过这样会导致每次运行时都会变慢。

# Android自定义Lint实践

来源:[tech.meituan.com](http://tech.meituan.com)

## 概述

Android Lint是Google提供给Android开发者的静态代码检查工具。使用Lint对Android工程代码进行扫描和检查，可以发现代码潜在的问题，提醒程序员及早修正。

为保证代码质量，美团在开发流程中加入了代码检查，如果代码检测到问题，则无法合并到正式分支中，这些检查中就包括Lint。

## 为什么需要自定义

我们在实际使用Lint中遇到了以下问题：

- 原生Lint无法满足我们团队特有的需求，例如：编码规范。
- 原生Lint存在一些检测缺陷或者缺少一些我们认为有必要的检测。

基于上面的考虑，我们开始调研并开发自定义Lint。

## 自定义Lint入门

在介绍美团的实践之前，先用一个小例子，来看看如何进行自定义Lint。

## 示例介绍

开发中我们希望开发者使用RoboGuice的Ln替代`Log/System.out.println`。

Ln相比于Log有以下优势：

- 对于正式发布包来说，debug和verbose的日志会自动不显示。
- 拥有更多的有用信息，包括应用程序名字、日志的文件和行信息、时间戳、线程等。
- 由于使用了可变参数，禁用后日志的性能比Log高。因为最冗长的日志往往都是debug或verbose日志，这可以稍微提高一些性能。
- 可以覆盖日志的写入位置和格式。

这里我们以此为例，让Lint检查代码中Log/System.out.println的调用，提醒开发者使用Ln。

## 创建Java工程，配置Gradle

```
apply plugin: 'java'

dependencies {
 compile fileTree(dir: 'libs', include: ['*.jar'])
 compile 'com.android.tools.lint:lint-api:24.5.0'
 compile 'com.android.tools.lint:lint-checks:24.5.0'
}
```

注：

- **lint-api**: 官方给出的API，API并不是最终版，官方提醒随时有可能会更改API接口。
- **lint-checks**: 已有的检查。

## 创建Detector

Detector负责扫描代码，发现问题并报告。

```
/**
 * 避免使用Log / System.out.println ,提醒使用Ln
 *
 * RoboGuice's Ln logger is similar to Log, but has the following advantages:
 * - Debug and verbose logging are automatically disabled for release builds.
 * - Your app name, file and line of the log message, time stamp, thread, and other
 * - Performance of disabled logging is faster than Log due to the use of the vararg
 * - You can override where the logs are written to and the format of the logging
 *
 * https://github.com/roboguice/roboguice/wiki/Logging-via-Ln
 *
 * Created by chentong on 18/9/15.
 */
public class LogDetector extends Detector implements Detector.JavaScanner{

 public static final Issue ISSUE = Issue.create(
 "LogUse",
 "避免使用Log/System.out.println",
 "使用Ln, 防止在正式包打印log",
 Category.SECURITY, 5, Severity.ERROR,
 new Implementation(LogDetector.class, Scope.JAVA_FILE_SCOPE));

 @Override
 public List<Class<? extends Node>> getApplicableNodeTypes() {
```

```

 return Collections.<Class<? extends Node>>singletonList(MethodInvocation.class);
 }

 @Override
 public AstVisitor createJavaVisitor(final JavaContext context) {
 return new ForwardingAstVisitor() {
 @Override
 public boolean visitMethodInvocation(MethodInvocation node) {

 if (node.toString().startsWith("System.out.println")) {
 context.report(ISSUE, node, context.getLocation(node),
 "请使用Ln，避免使用System.out.println");
 return true;
 }

 JavaParser.ResolvedNode resolve = context.resolve(node);
 if (resolve instanceof JavaParser.ResolvedMethod) {
 JavaParser.ResolvedMethod method = (JavaParser.ResolvedMethod) resolve;
 // 方法所在的类校验
 JavaParser.ResolvedClass containingClass = method.getContainingClass();
 if (containingClass.matches("android.util.Log")) {
 context.report(ISSUE, node, context.getLocation(node),
 "请使用Ln，避免使用Log");
 return true;
 }
 }
 return super.visitMethodInvocation(node);
 }
 };
 }
}

```

可以看到这个Detector继承Detector类，然后实现Scanner接口。

自定义Detector可以实现一个或多个Scanner接口,选择实现哪种接口取决于你想要的扫描范围

- Detector.XmlScanner
- Detector.JavaScanner
- Detector.ClassScanner
- Detector.BinaryResourceScanner
- Detector.ResourceFolderScanner
- Detector.GradleScanner
- Detector.OtherFileScanner

这里因为我们是要针对Java代码扫描，所以选择使用JavaScanner。

代码中getApplicableNodeTypes方法决定了什么样的类型能够被检测到。这里我们想看Log以及println的方法调用，选取MethodInvocation。对应的，我们在createJavaVisitor创建一个ForwardingAstVisitor通过visitMethodInvocation方法来接收被检测到的Node。

可以看到getApplicableNodeTypes返回值是一个List，也就是说可以同时检测多种类型的节点来帮助精确定位到代码，对应的ForwardingAstVisitor接受返回值进行逻辑判断就可以了。

可以看到JavaScanner中还有其他很多方法，getApplicableMethodNames（指定方法名）、visitMethod（接收检测到的方法），这种对于直接找寻方法名的场景会更方便。当然这种场景我们用最基础的方式也可以完成，只是比较繁琐。

## 那么其他Scanner如何去写呢？

可以去查看各接口中的方法去实现，一般都是有这两种对应：什么样的类型需要返回、接收发现的类型。

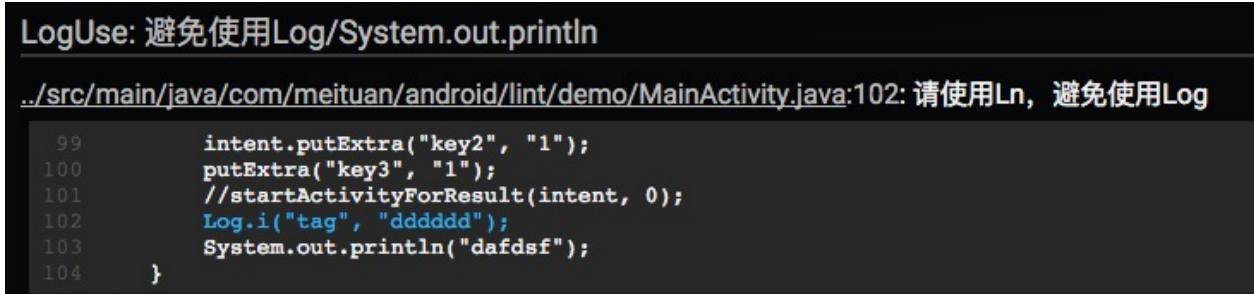
这里插一句，Lint是如何实现Java扫描分析的呢？Lint使用了[Lombok](#)做抽象语法树的分析。所以在我们告诉它需要什么类型后，它就会把相应的Node返回给我们。

回到示例，当接收到返回的Node之后需要进行判断，如果调用方法是System.out.println或者属于android.util.Log类，则调用context.report上报。

```
context.report(ISSUE, node, context.getLocation(node), "请使用Ln, 避免使用Log");
```

- 第一个参数是Issue，这个之后会讲到；
- 第二个参数是当前节点；
- 第三个参数location会返回当前的位置信息，便于在报告中显示定位；

最后的字符串用来为警告添加解释。对应报告中的位置如下图：



The screenshot shows a Lint report window. At the top, it says "LogUse: 避免使用Log/System.out.println". Below that, it shows the file path: ".../src/main/java/com/meituan/android/lint/demo/MainActivity.java:102: 请使用Ln, 避免使用Log". The code snippet below the path is as follows:

```
99 intent.putExtra("key2", "1");
100 putExtra("key3", "1");
101 //startActivityForResult(intent, 0);
102 Log.i("tag", "dddddd");
103 System.out.println("dafdsf");
104 }
```

这里还需要说明report会自动处理被suppress(suppressLint)/ignore(tools:ignore)的警告。所以发现问题直接调用report就可以，不用担心其他问题。

## Issue

Issue由Detector发现并报告，是Android程序代码可能存在的bug。

```
public static final Issue ISSUE = Issue.create(
 "LogUse",
 "避免使用Log/System.out.println",
 "使用Ln, 防止在正式包打印log",
 Category.SECURITY, 5, Severity.ERROR,
 new Implementation(LogDetector.class, Scope.JAVA_FILE_SCOPE));
```

声明为final class，由静态工厂方法创建。对应参数解释如下：

- id：唯一值，应该能简短描述当前问题。利用Java注解或者XML属性进行屏蔽时，使用的就是这个id。
- summary：简短的总结，通常5-6个字符，描述问题而不是修复措施。
- explanation：完整的问题解释和修复建议。
- category：问题类别。详见下文详述部分。
- priority：优先级。1-10的数字，10为最重要/最严重。
- severity：严重级别：Fatal, Error, Warning, Informational, Ignore。
- Implementation：为Issue和Detector提供映射关系，Detector就是当前Detector。声明扫描检测的范围Scope，Scope用来描述Detector需要分析时需要考虑的文件集，包括：Resource文件或目录、Java文件、Class文件。

## 与Lint HTML报告对应关系

| Security | severity | id     | summary                    |
|----------|----------|--------|----------------------------|
| 2        | !        | LogUse | 避免使用Log/System.out.println |

**LogUse: 避免使用Log/System.out.println**

./src/main/java/com/meituan/android/lint/demo/MainActivity.java:102: 请使用Ln, 避免使用Log

```

99 intent.putExtra("key2", "1");
100 putExtra("key3", "1");
101 //startActivityForResult(intent, 0);
102 Log.i("tag", "dddddd");
103 System.out.println("dafdsf");
104 }

```

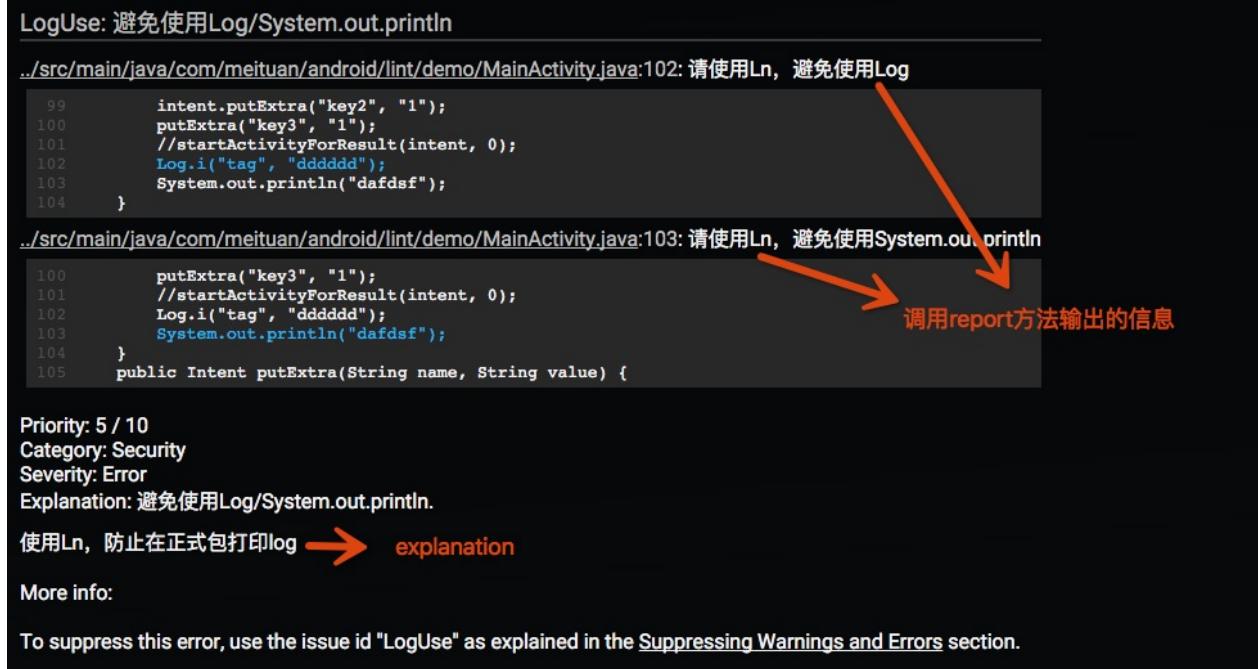
./src/main/java/com/meituan/android/lint/demo/MainActivity.java:103: 请使用Ln, 避免使用System.out.println

```

100 putExtra("key3", "1");
101 //startActivityForResult(intent, 0);
102 Log.i("tag", "dddddd");
103 System.out.println("dafdsf");
104 }
105 public Intent putExtra(String name, String value) {

```

Priority: 5 / 10  
Category: Security  
Severity: Error  
Explanation: 避免使用Log/System.out.println.  
使用Ln, 防止在正式包打印log → explanation  
More info:  
To suppress this error, use the issue id "LogUse" as explained in the Suppressing Warnings and Errors section.



## Category详述

系统现在已有的类别如下：

- Lint
- Correctness (incl. Messages)
- Security
- Performance
- Usability (incl. Icons, Typography)
- Accessibility
- Internationalization
- Bi-directional text

## 自定义Category

```

public class MTCategory {
 public static final Category NAMING_CONVENTION = Category.create("命名规范", 101);
}

```

## 使用

```
public static final Issue ISSUE = Issue.create(
 "IntentExtraKey",
 "intent extra key 命名不规范",
 "请在接受此参数中的Activity中定义一个按照EXTRA_<name>格式命名的常量",
 MTCCategory.NAMING_CONVENTION, 5, Severity.ERROR,
 new Implementation(IntentExtraKeyDetector.class, Scope.JAVA_FILE_SCOPE));
```

## IssueRegistry

提供需要被检测的Issue列表

```
public class MTIssueRegistry extends IssueRegistry {
 @Override
 public synchronized List<Issue> getIssues() {
 System.out.println("==== MT lint start =====");
 return Arrays.asList(
 DuplicatedActivityIntentFilterDetector.ISSUE,
 //IntentExtraKeyDetector.ISSUE,
 //FragmentArgumentsKeyDetector.ISSUE,
 LogDetector.ISSUE,
 PrivateModeDetector.ISSUE,
 WebViewSafeDetector.ON RECEIVED SSL ERROR,
 WebViewSafeDetector.SET_SAVE_PASSWORD,
 WebViewSafeDetector.SET_ALLOW_FILE_ACCESS,
 WebViewSafeDetector.WEB_VIEW_USE,
 HashMapForJDK7Detector.ISSUE
);
 }
}
```

在getIssues()方法中返回需要被检测的Issue List。

在build.grade中声明Lint-Registry属性

```
jar {
 manifest {
 attributes("Lint-Registry": "com.meituan.android.lint.core.MTIssueRegistry")
 }
}
```

至此，自定义Lint的编码部分就完成了。

之前提到自定义Lint是一个Java工程，那么打出的jar包如何使用呢？

# jar包使用

## Google方案

将jar拷贝到`~/.android/lint`中

```
$ mkdir ~/.android/lint/
$ cp customrule.jar ~/.android/lint/
```

缺点：针对所有工程，会影响同一台机器其他工程的Lint检查。即便触发工程时拷贝过去，执行完删除，但其他进程或线程使用`./gradlew lint`仍可能会受到影响。

## LinkedIn方案

LinkedIn提供了另一种思路：将jar放到一个aar中。这样我们就可以针对工程进行自定义Lint，lint.jar只对当前工程有效。

详细介绍请看LinkedIn博客：[Writing Custom Lint Checks with Gradle](#)。

我们对此方案进行调研，得出以下结论：

## 可行性

[AAR Format](#) 中写明可以有lint.jar。

从Google Groups adt-dev论坛讨论来看是官方目前的推荐方案，详见：[Specify custom lint JAR outside of lint tools settings directory](#)

测试后发现aar中有lint.jar，最终APK中并不会引起包体积变化。

## 缺点

官方plugin偶尔出bug，给人一种不太重视的感觉。

目前plugin的支持情况是：1.1.x正常，1.2.x不支持，1.3.x修复问题，1.5.x正常。

1.2.x Gradle plugin遇到的两个问题：

- [Issue 174808:custom lint in AAR doesn't work](#)
- [Issue 178699:lint.jar in AAR doesn't work sometimes](#)

经过对比，我们最终选择了LinkedIn的方案。

# 美团的实践

在确定方案后，我们为Lint增加了很多功能，包括编码规范和原生Lint增强。这里以HashMap检测为例，介绍一下美团Lint。

## 增强HashMap检测

Lint检测中有一项是Java性能检测，常见的就是：HashMap can be replaced with SparseArray。

```
public static void testHashMap() {
 HashMap<Integer, String> map1 = new HashMap<Integer, String>();
 map1.put(1, "name");
 HashMap<Integer, String> map2 = new HashMap<>();
 map2.put(1, "name");
 Map<Integer, String> map3 = new HashMap<>();
 map3.put(1, "name");
}
```

对于上述代码，原生Lint只能检测第一种情况，JDK 7泛型新写法还检测不到。

了解到这点之后，我们决定为HashMap提供增强检测。

分析源码后发现，HashMap检测是根据new HashMap处的泛型来判断是否符合条件。

于是我们想到，在发现new HashMap后去找前面的泛型，因为本身Java就是靠类型推断的，我们可以直接根据前面的泛型来确定是否使用SparseArray。当然，是不是HashMap还需要通过后面的new HashMap来判断，否则容易出现问题。

代码如下：

```
@Override
public List<Class<? extends Node>> getApplicableNodeTypes() {
 return Collections.<Class<? extends Node>>singletonList(ConstructorInvocation.class)
}

private static final String INTEGER = "Integer"; // $NON-NLS-1$
private static final String BOOLEAN = "Boolean"; // $NON-NLS-1$
private static final String BYTE = "Byte"; // $NON-NLS-1$
private static final String LONG = "Long"; // $NON-NLS-1$
private static final String HASH_MAP = "HashMap"; // $NON-NLS-1$

@Override
public AstVisitor createJavaVisitor(@NonNull JavaContext context) {
 return new ForwardingAstVisitor() {
```

```

@Override
public boolean visitConstructorInvocation(ConstructorInvocation node) {
 TypeReference reference = node.astTypeReference();
 String typeName = reference.astParts().last().astIdentifier().astValue();
 // TODO: Should we handle factory method constructions of HashMaps as well
 // e.g. via Guava? This is a bit trickier since we need to infer the type
 // arguments from the calling context.
 if (typeName.equals(HASH_MAP)) {
 checkHashMap(context, node, reference);
 }
 return super.visitConstructorInvocation(node);
}
};

}

/**
 * Checks whether the given constructor call and type reference refers
 * to a HashMap constructor call that is eligible for replacement by a
 * SparseArray call instead
 */
private void checkHashMap(JavaContext context, ConstructorInvocation node, TypeReference reference) {
 StrictListAccessor<TypeReference, TypeReference> types = reference.getTypeArguments();
 if (types == null || types.size() != 2) {
 /*
 * JDK 7 新写法
 */
 HashMap<Integer, String> map2 = new HashMap<>();
 map2.put(1, "name");
 Map<Integer, String> map3 = new HashMap<>();
 map3.put(1, "name");
 /*
 */
 Node variableDefinition = node.getParent().getParent();
 if (variableDefinition instanceof VariableDefinition) {
 TypeReference typeReference = ((VariableDefinition) variableDefinition).getTypeReference();
 checkCore(context, variableDefinition, typeReference); // 此方法即原HashMap检测
 }
 }
 // else --> lint本身已经检测
}
}

```

代码很简单，总体就是获取变量定义的地方，将泛型值传入原先的检测逻辑。

当然这里的增强也是有局限的，比如这个变量是成员变量，向前的推断就会有问题，这点我们还在持续的优化中。

总结一下实践过程中的技巧：

- 因为没有好的文档，我们更多地是要从源码的检测中学习，多看lint-checks。
- 需要的时候使用SdkConstants，充分利用LintUtils，Lint给我们提供了很多方便的工具。

## 为自定义Lint开发plugin

aar虽然很方便，但是在团队内部推广中我们遇到了以下问题：

- 配置繁琐，不易推广。每个库都需要自行配置lint.xml、lintOptions，并且compile aar。
- 不易统一。各库之间需要使用相同的配置，保证代码质量。但现在手动来回拷贝规则，且配置文件可以自己修改。

于是我们想到开发一个plugin，统一管理lint.xml和lintOptions，自动添加aar。

## 统一lint.xml

我们在plugin中内置lint.xml，执行前拷贝过去，执行完成后删除。

```
lintTask.doFirst {

 if (lintFile.exists()) {
 lintOldFile = project.file("lintOld.xml")
 lintFile.renameTo(lintOldFile)
 }
 def isLintXmlReady = copyLintXml(project, lintFile)

 if (!isLintXmlReady) {
 if (lintOldFile != null) {
 lintOldFile.renameTo(lintFile)
 }
 throw new GradleException("lint.xml不存在")
 }
}

project.gradle.taskGraph.afterTask { task, TaskState state ->
 if (task == lintTask) {
 lintFile.delete()
 if (lintOldFile != null) {
 lintOldFile.renameTo(lintFile)
 }
 }
}
```

## 统一lintOptions

Android plugin在1.3以后允许我们替换Lint Task的lintOptions:

```
def newOptions = new LintOptions()
newOptions.lintConfig = lintFile
newOptions.warningsAsErrors = true
newOptions.abortOnError = true
newOptions.htmlReport = true
//不放在build下，防止被clean掉
newOptions.htmlOutput = project.file("${project.projectDir}/lint-report/lint-report.h
newOptions.xmlReport = false

lintTask.lintOptions = newOptions
```

## 自动添加最新aar

这里还涉及一个问题：当我们plugin开发完成提供给团队使用的时候，假设我们需要修改lint aar，那么团队的plugin就要统一升级。这点就比较繁琐。

考虑到plugin只是一个检查代码插件，它最需要的应该是实时更新。我们引入了Gradle Dynamic Versions：

```
project.dependencies {
 compile 'com.meituan.android.lint:lint:latest.integration'
}

project.configurations.all {
 resolutionStrategy.cacheDynamicVersionsFor 0, 'seconds'
}
```

plugin开发完成，就可以提供给团队部署了。

当然为了团队更方便地接入检查，我们在检查流程中内置了脚本来自动添加plugin，这样团队就可以在不添加任何代码的情况下，实现自定义Lint检查。

## 参考文献

- Google. [Writing Custom Lint Rules](#). Android Tools Project Site.
- Google. [Writing a Lint Check](#). Android Tools Project Site.
- Prengemann M. [The Power of Custom Lint Checks](#). SpeakerDeck.

- Diermann A. [Custom Lint Rules](#). GitHub.
- Diermann A. [Android Lint API Reference Guide](#). GitHub.
- Google. [Android Custom Lint Rules Sample Code](#). GitHub.
- Cheng Yang. [Writing Custom Lint Checks with Gradle](#). LinkedIn.

# Tools

# Android 性能优化之使用MAT分析内存泄露问题

来源:[CSDN](#)

我们平常在开发Android应用程序的时候，稍有不慎就有可能产生OOM，虽然JAVA有垃圾回收机，但也不能杜绝内存泄露，内存溢出等问题，随着科技的进步，移动设备的内存也越来越大了，但由于Android设备的参差不齐，可能运行在这台设备好好的，运行在那台设备就报OOM，这些适配问题也是比较蛋疼的，比如我们平常运行着一个应用程序，运行的好好的，突然到某个Activity就给你爆出一个OOM的错误，你可能会以为是这个Activity导致的内存泄露，你会想到也有可能是内存有泄露吗？内存泄露就像一个定时炸弹，随时都有可能使我们的应用程序崩溃掉，所以作为一名Android开发人员，还是需要有分析内存泄露的能力，说道这里我们还是要说下什么是内存泄露，内存泄露是指有个引用指向一个不再被使用的对象，导致该对象不会被垃圾回收器回收。因此，垃圾回收器是无法回收内存泄露的对象。本文就使用DDMS(Dalvik Debug Monitor Server)和MAT(Memory Analyzer Tool)工具带大家来分析内存泄露问题。

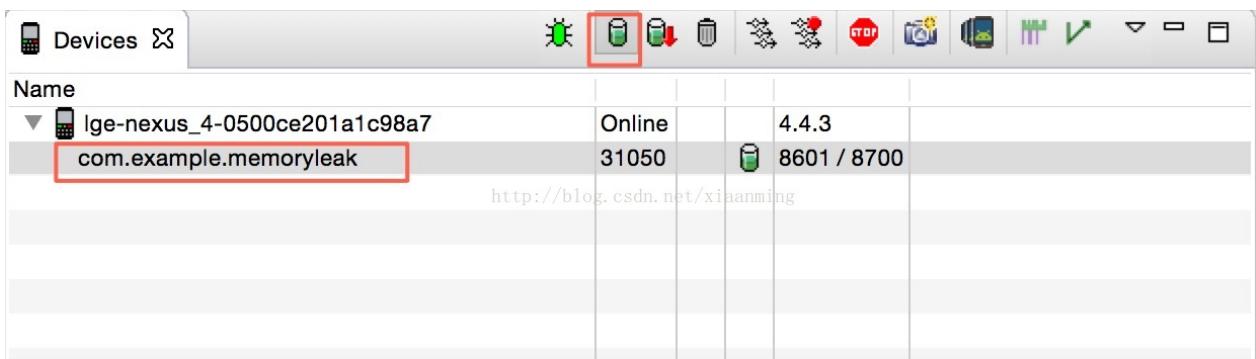
## 工具的准备

DDMS是ADT自带的调试工具,有关DDMS的使用请参考<http://developer.android.com/tools/debugging/ddms.html>，而MAT的就需要我们自行安装Eclipse插件，安装方法我就不多说了，下面给出一个在线安装的地址：<http://download.eclipse.org/mat/1.3/update-site/>，MAT可以检测到内存泄露，降低内存消耗，它有着非常强大的解析堆内存空间dump能力。

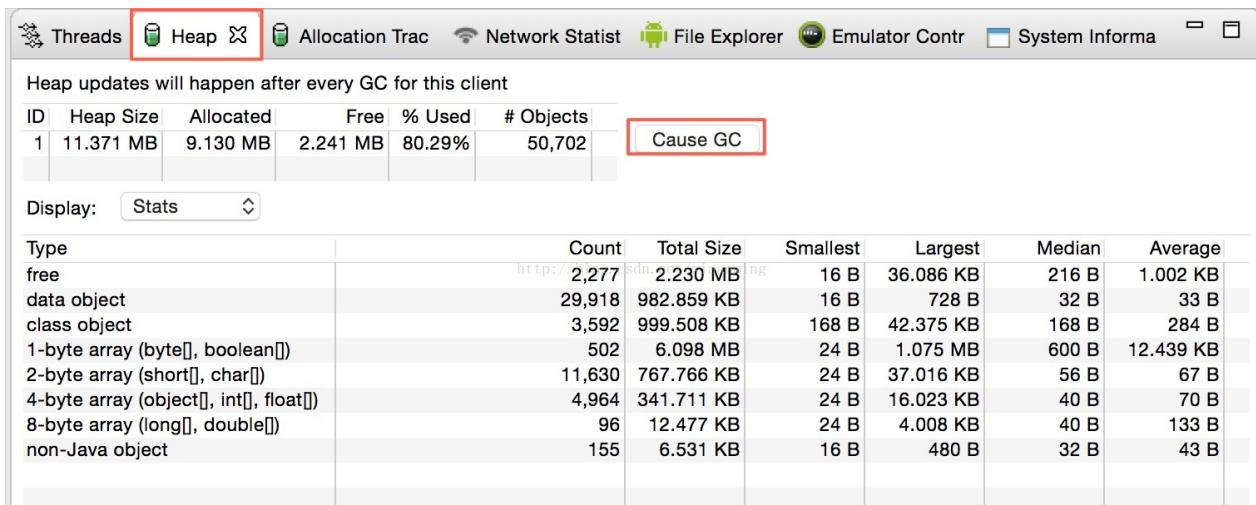
## 如何检测内存泄露

### 1. 使用DDMS检测内存泄露

打开Devices视图,选择我们需要分析的应用程序进程，点击 `Update Heap` 按钮



然后在打开DDMS, 选择Heap标签, 然后点击 Cause GC 按钮, 点击 Cause GC 是手动触发 JAVA垃圾回收器, 如下图



如果我们要测试某个Activity是否发生内存泄露, 我们可以反复进入和退出这个Activity, 再手动触发几次垃圾回收, 观察上图中 data object 这一栏中的 Total Size 的大小是保持稳定还是有明显的变大趋势, 如果有明显的变大趋势就说明这个Activity存在内存泄露的问题, 我们就需要在具体分析。

## 2. 使用Logcat检测内存泄露

当垃圾回收机在进行垃圾回收之后, 会在Logcat中作相对于的输出, 所以我们也可以通过这些信息来判断是否存在内存泄露问题

```
. dalvikvm GC_EXPLICIT freed 1434K, 14% free 10065K/11644K, paused 5ms+25ms, total 108ms
. dalvikvm GC_EXPLICIT freed 716K, 20% free 9349K/11644K, paused 8ms+3ms, total 71ms
```

- 一, 上面消息的第一个部分产生GC的原因, 一共有四种类型
  - **GC\_CONCURRENT** 当你的堆内存快被用完的时候, 就会触发这个GC回收
  - **GC\_FOR\_MALLOC** 堆内存已经满了, 同时又要试图分配新的内存, 所以系统要回收内存

- **GC\_EXTERNAL\_ALLOC** 在Android3.0 (Honeycomb)以前，释放通过外部内存（比如在2.3以前，产生的Bitmap对象存储在Native Memory中）时产生。Android3.0和更高版本中不再有这种类型的内存分配了。
- **GC\_EXPLICIT** 调用System.gc时产生，上图中就是点击Cause GC按钮手动触发垃圾回收器产生的log信息
- 二， freed 1413K表示GC释放了1434K的内存
- 三， 20% free 9349K/11644K，20%表示目前可分配内存占的比例，9349K表示当前活动对象所占内存，11644K表示Heap的大小
- 四， paused 8ms + 3ms，total 71ms，则表示触发GC应用暂停的时间和GC总共消耗的时间

有了这些log信息，我们就可以知道GC运行几次以后有没有成功释放出一些内存，如果分配出去的内存持续增加，那么很明显存在内存泄露，如下存在内存泄露的Log信息

```
GC_FOR_ALLOC freed 206K, 13% free 10132K/11644K, paused 18ms, total ↴
18ms
GC_FOR_ALLOC freed 250K, 11% free 10394K/11644K, paused 43ms, total ↴
43ms
GC_FOR_ALLOC freed 243K, 9% free 10662K/11644K, paused 22ms, total ↴
22ms
GC_FOR_ALLOC freed 220K, 6% free 10984K/11644K, paused 23ms, total ↴
23ms
GC_FOR_ALLOC freed 257K, 3% free 11377K/11668K, paused 24ms, total ↴
25ms
GC_FOR_ALLOC freed 318K, 3% free 11840K/12192K, paused 29ms, total ↴
29ms
```

很明显Heap中空闲内存占总Heap的比例在缩小，Heap中活动对象所占的内存增加。

## 内存泄露分析实战

下面是一个存在内存泄露的例子代码，这也是比较常见的一种内存泄露的方式，就是在Activity中写一些内部类，并且这些内部类具有生命周期过长的现象

```
package com.example.memoryleak;

import java.util.ArrayList;
import java.util.List;

import android.app.Activity;
import android.os.Bundle;

public class LeakActivity extends Activity {
 private List<String> list = new ArrayList<String>();

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);

 //模拟Activity一些其他的对象
 for(int i=0; i<10000;i++){
 list.add("Memory Leak!");
 }

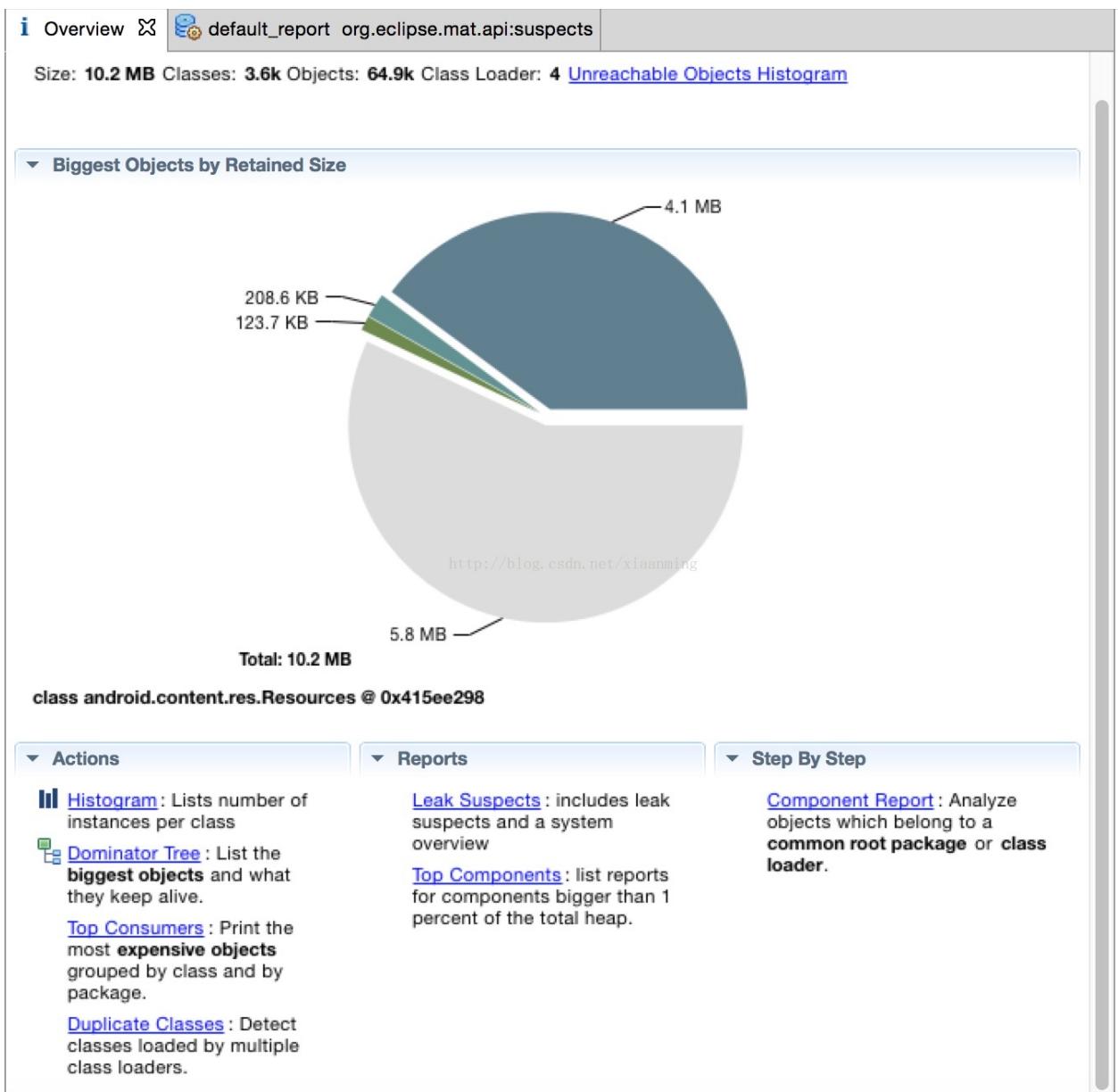
 //开启线程
 new MyThread().start();
 }

 public class MyThread extends Thread{

 @Override
 public void run() {
 super.run();

 //模拟耗时操作
 try {
 Thread.sleep(10 * 60 * 1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
}
```

运行例子代码，选择 Devices 视图，点击上面 Update Heap 标签，然后再旋转屏幕，多重复几次，然后点击 Dump HPROF file，之后Eclipse的MAT插件会自动帮我们打开，如下图



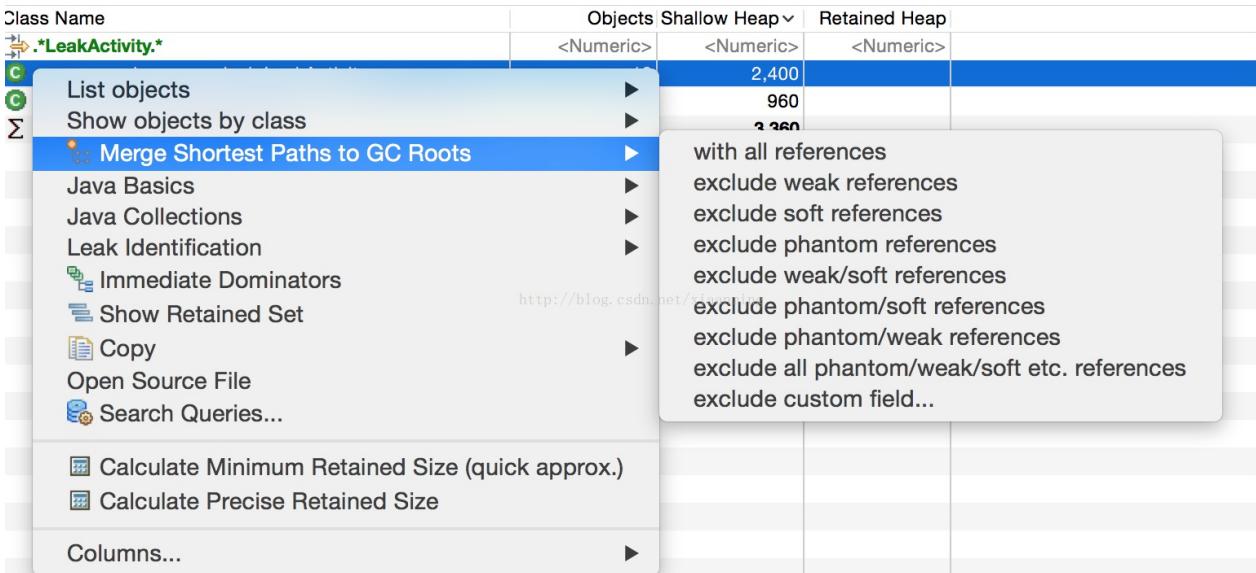
我们看到下面有Histogram（直方图）他列举了每个对象的统计，Dominator Tree(支配树) 提供了程序中最占内存的对象的排列，这两个是我在排查内存泄露的时候用的最多的

## Histogram（直方图）

我们先来看Histogram, MAT最有用的工具之一，它可以列出任意一个类的实例数。它支持使用正则表达式来查找某个特定的类，还可以计算出该类所有 对象的保留堆最小值或者精确值, 我们可以通过正则表达式输入LeakActivity, 看到Histogram列出了与LeakActivity相关的类

| Class Name                                    | Objects   | Shallow Heap | Retained Heap |
|-----------------------------------------------|-----------|--------------|---------------|
| <Numeric>                                     | <Numeric> | <Numeric>    |               |
| .*LeakActivity.*                              | 16        | 3,200        | 1,084,848     |
| com.example.memoryleak.LeakActivity           | 16        | 1,280        | >= 945,000    |
| com.example.memoryleak.LeakActivity\$MyThread | 32        | 4,480        |               |
| <b>Σ Total: 2 entries (3,590 filtered)</b>    |           |              |               |

我们可以看到LeakActivity,和MyThread内部类都存在16个对象，虽然LeakActivity和MyThread存在那么多对象，但是到这里并不能让我们准确的判断这两个对象是否存在内存泄露问题，选中com.example.memoryleak.LeakActivity，点击右键，如下图



Merge Shortest Paths to GC Roots 可以查看一个对象到 GC Roots 是否存在引用链相连接，在JAVA中是通过可达性 (Reachability Analysis)来判断对象是否存活，这个算法的基本思想是通过一系列的称谓 GC Roots 的对象作为起始点，从这些节点开始向下搜索，搜索所走得路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连则该对象被判定为可以被回收的对象，反之不能被回收，我们可以选择 exclude all phantom/weak/soft etc.references (排查虚引用/弱引用/软引用等) 因为被虚引用/弱引用/软引用的对象可以直接被GC给回收.

| Class Name                                                                    | Ref. Objects | Shallow Heap | Ref. Shallow Hea |
|-------------------------------------------------------------------------------|--------------|--------------|------------------|
|                                                                               | <Numeric>    | <Numeric>    | <Numeric>        |
| <Regex>                                                                       |              |              |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x41e8f958 Thread-3952 Thread | 1            | 80           |                  |
| this\$0 com.example.memoryleak.LeakActivity @ 0x41e9a3f0                      | 1            | 200          |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x420e6ac8 Thread-3955 Thread | 1            | 80           |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x4205e8f0 Thread-3954 Thread | 1            | 80           |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x41fb43a0 Thread-3950 Thread | 1            | 80           |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x41e99de8 Thread-3959 Thread | 1            | 80           |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x42124420 Thread-3951 Thread | 1            | 80           |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x420e7538 Thread-3956 Thread | 1            | 80           |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x4211ef50 Thread-3958 Thread | 1            | 80           |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x41ffb970 Thread-3960 Thread | 1            | 80           |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x4205c540 Thread-3953 Thread | 1            | 80           |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x41e8db38 Thread-3957 Thread | 1            | 80           |                  |
| com.example.memoryleak.LeakActivity\$MyThread @ 0x42072488 Thread-3961 Thread | 1            | 80           |                  |
| <b>Total: 12 entries</b>                                                      | 12           | 960          | 2                |

可以看到LeakActivity存在GC Roots链，即存在内存泄露问题，可以看到LeakActivity被MyThread的this\$0持有。

除了使用 Merge Shortest Paths to GC Roots 我们还可以使用

- List object - With outgoing References 显示选中对象持有那些对象

- `List object - With incoming References` 显示选中对象被那些外部对象所持有
- `Show object by class - With outgoing References` 显示选中对象持有哪些对象, 这些对象按类合并在一起排序
- `Show object by class - With incoming References` 显示选中对象被哪些外部对象持有, 这些对象按类合并在一起排序

## Dominator Tree(支配树)

它可以将所有对象按照Heap大小排序显示, 使用方法跟Histogram (直方图)差不多, 在这里我就不做过多的介绍了

我们知道上面的例子代码中我们知道内部类会持有外部类的引用, 如果内部类的生命周期过长, 会导致外部类内存泄露, 那么你会问, 我们应该怎么写那不会出现内存泄露的问题呢? 既然内部类不行, 我们就外部类或者static的内部类, 如果我们需要用到外部类里面的一些东西, 我们可以将外部类Weak Reference传递进去

```
package com.example.memoryleak;

import java.lang.ref.WeakReference;
import java.util.ArrayList;
import java.util.List;

import android.app.Activity;
import android.os.Bundle;

public class LeakActivity extends Activity {
 private List<String> list = new ArrayList<String>();

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);

 //模拟Activity一些其他的对象
 for(int i=0; i<10000;i++){
 list.add("Memory Leak!");
 }

 //开启线程
 new MyThread(this).start();
 }

 public static class MyThread extends Thread{
 private WeakReference<LeakActivity> mLeakActivityRef;
```

```
public MyThread(LeakActivity activity){
 mLeakActivityRef = new WeakReference<LeakActivity>(activity);
}

@Override
public void run() {
 super.run();

 //模拟耗时操作
 try {
 Thread.sleep(10 * 60 * 1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }

 //如果需要使用LeakActivity, 我们需要添加一个判断
 LeakActivity activity = mLeakActivityRef.get();
 if(activity != null){
 //do something
 }
}

}
```

同理，Handler也存在同样的问题，比如下面的代码

```
package com.example.memoryleak;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;

public class LeakActivity extends Activity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);

 MyHandler handler = new MyHandler();
 handler.sendMessageDelayed(Message.obtain(), 10 * 60 * 1000);
 }

 public class MyHandler extends Handler{

 @Override
 public void handleMessage(Message msg) {
 super.handleMessage(msg);
 }
 }
}
```

我们知道使用MyHandler发送消息的时候，Message会被加入到主线程的MessageQueue里面，而每条Message的target会持有MyHandler对象，而MyHandler的this\$0又会持有LeakActivity对象，所以我们在旋转屏幕的时候，由于每条Message被延迟了10分钟，所以必然会导致LeakActivity泄露，所以我们需要将代码进行修改下

```
package com.example.memoryleak;

import java.lang.ref.WeakReference;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;

public class LeakActivity extends Activity {
 MyHandler handler;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);

 handler = new MyHandler(this);
 handler.sendMessageDelayed(Message.obtain(), 10 * 60 * 1000);
 }

 public static class MyHandler extends Handler{
 public WeakReference<LeakActivity> mLeakActivityRef;

 public MyHandler(LeakActivity leakActivity) {
 mLeakActivityRef = new WeakReference<LeakActivity>(leakActivity);
 }

 @Override
 public void handleMessage(Message msg) {
 super.handleMessage(msg);

 if(mLeakActivityRef.get() != null){
 //do something
 }
 }
 }

 @Override
 protected void onDestroy() {
 handler.removeCallbacksAndMessages(null);
 super.onDestroy();
 }
}
```

上面的代码就能保证LeakActivity不会被泄露，注意我们在Activity的onDestory方法中使用了 `handler.removeCallbacksAndMessages(null)`，这样子能保证LeakActivity退出的时候，每条Message的target MyHandler也会被释放，所以我们在使用非static的内部类的时候，

要注意该内部类的生命周期是否比外部类要长，如果是的话我们可以使用上面的解决方法。

## 常见的内存泄露问题

- 1.上面两种情形
- 2.资源对象没有关闭，比如数据库操作中得Cursor,IO操作的对象
- 3.调用了registerReceiver注册广播后未调用unregisterReceiver()来取消
- 4.调用了 `View.getViewTreeObserver().addOnXXXListener`,而没有调用 `View.getViewTreeObserver().removeXXXListener`
- 5.Android 3.0以前，没有对不在使用的Bitmap调用 `recycle()`,当然在Android 3.0以后就不需要了，更详细的请查看<http://blog.csdn.net/xiaanming/article/details/41084843>
- 6.Context的泄露，比如我们在单例类中使用Context对象，如下

```
import android.content.Context;

public class Singleton {
 private Context context;
 private static Singleton mSingleton;

 private Singleton(Context context){
 this.context = context;
 }

 public static Singleton getInstance(Context context){
 if(mSingleton == null){
 synchronized (Singleton.class) {
 if(mSingleton == null){
 mSingleton = new Singleton(context);
 }
 }
 }
 return mSingleton;
 }
}
```

假如我们在某个Activity中使用 `Singleton.getInstance(this)` 或者该实例，那么会造成该Activity一直被Singleton对象引用着，所以这时候我们应该使用 `getApplicationContext()` 来代替Activity的Context，`getApplicationContext()` 获取的Context是一个全局的对象，所以这样就避免了内存泄露。相同的还有将Context成员设置为static也会导致内存泄露问题。

- 7.不要重写 `finalize()` 方法，我们有时候可能会在某个对象被回收前去释放一些资源，可能会在 `finalize()` 方法中去做，但是实现了 `finalize` 的对象，创建和回收的过程都更耗时。创建时，会新建一个额外的 `Finalizer` 对象指向新创建的对象。而回收时，至少需要经过两次GC，第一次GC检测到对象只有被Finalizer引用，将这个对象放入 `Finalizer.ReferenceQueue` 此时，因为 `Finalizer` 的引用，对象还无法被GC，`Finalizer$FinalizerThread` 会不停的清理Queue的对象，remove掉当前元素，并执行对象的`finalize`方法，清理后对象没有任何引用，在下一次GC被回收，所以说该对象存活时间更久，导致内存泄露。

# Scalpel

# Scalpel: Jake大神的第三把刀

来源:[微信公众号](#)

Jake Wharton, Android开发领域传奇一般的存在，熟悉Android开发的同学应该都听说过，即便没有，也应该会经常用到他主导或参与贡献的开源项目，他在GitHub上开源了多个Android兼容性、依赖注入相关的知名项目，目前就职Square, 也参与贡献了Square公司开源的诸如Retrofit, okhttp等热门项目。

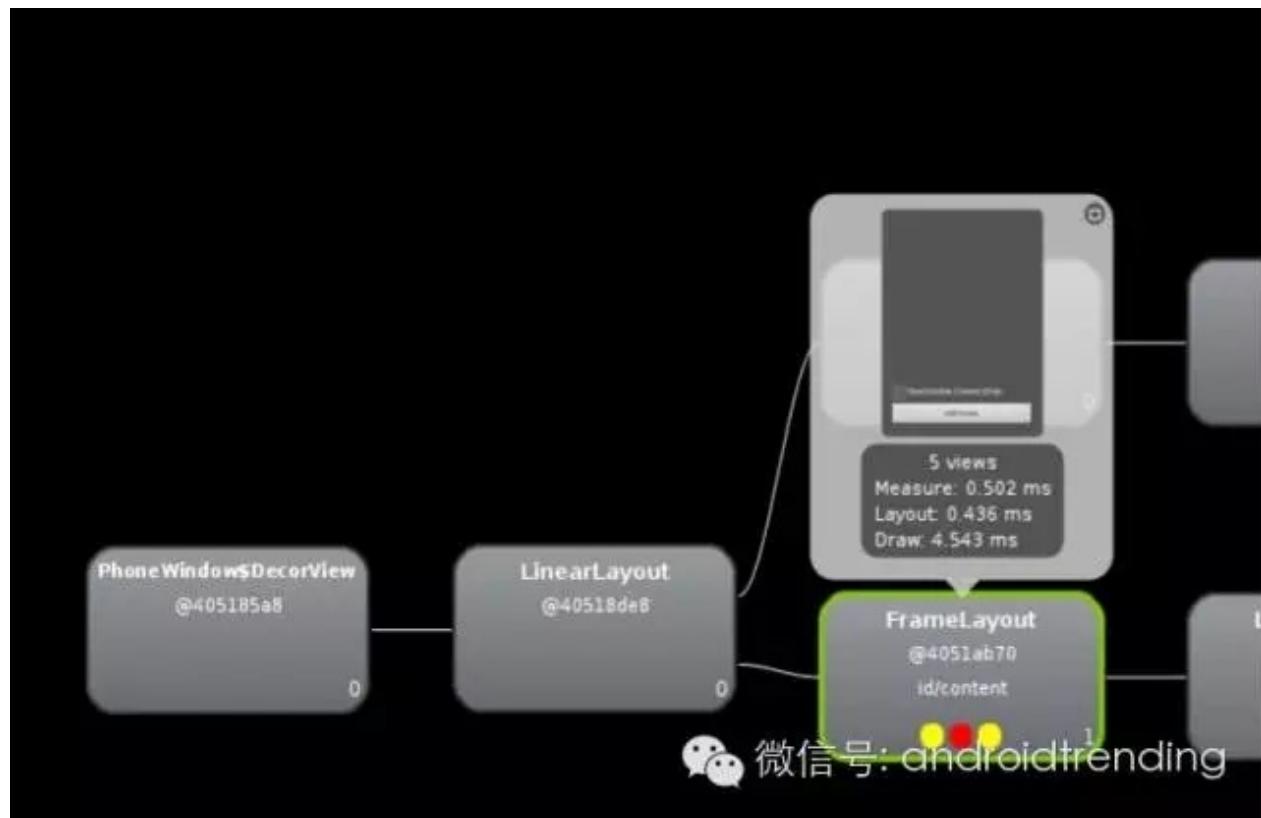
Jake大神喜欢以刀命名自己的项目，可能他觉得这能很好的表达它们作为开发利器的犀利程度，不过没准他也很喜欢中国武侠小说呢。

他最有名的两把刀，一把是Dagger, 匕首，一个依赖注入框架，用来解耦开发中各模块依赖的，最早由他开发与维护，后来转给Google维护；另一把是同样大名鼎鼎的ButterKnife，黄油刀，有了它，你再也不用写findViewById了，以后有机会给大家详细介绍它们。

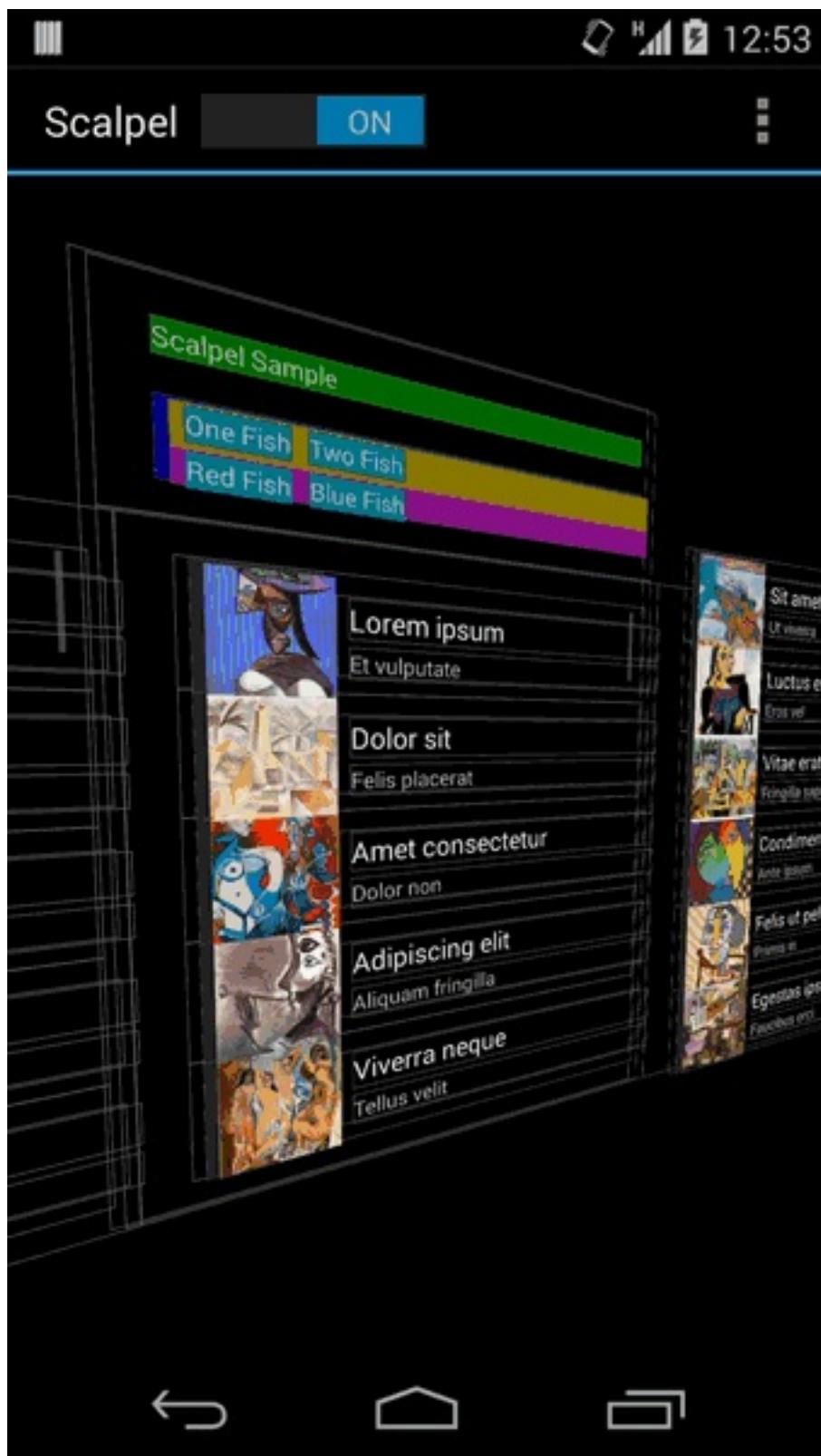
今天的主角是它们不那么知名的小兄弟，Scalpel，这把非常酷炫实用的解剖刀。

- 大家在Android界面开发中可能会经常碰到这样的困扰：
- 新接手某个项目，它的界面布局复杂且乱，怎么快速了解清楚呢？
- 感觉界面布局层次太深，对性能有影响，应该是有不合理的地方，怎么看是哪里不对呢？
- 我写的这个控件怎么就是不出来？它跑哪去了？
- 这里的局部关系是怎样的？要怎么修改才可以实现设计效果呀？

是的，这时候就轮到我们的主角出场了。Scal... 啊不对，是Hierarchy Viewer，Hierarchy Viewer是Android给我们提供的一款查看View Tree的小工具，通过它可以清晰的看到界面布局与层次细节，相当强大，大家应该都很熟悉它了，截张图大家感受下就好。



我们今天的主角Scalpel(终于轮到我出场了吗？！)与Hierarchy Viewer有关，一句话介绍它：A surgical debugging tool to uncover the layers under your app，大家看张图就明白它能干嘛了。



是不是很酷？他其实就是一个三维效果的界面布局层次展示，不需要手机连接开发设备，只需要简单几行代码将其集成到你的应用中，即可开启酷炫之旅，它提供的功能包括：

- 通过setLayerInteractionEnabled(boolean)可开启与关闭此功能。
- 通过setDrawViews(boolean)可控制是否绘制View，也就是说它可以仅仅是个简单的

布局线框图，也可以是色彩丰富的真实效果图。

- 通过setDrawIds(boolean)可控制是否显示各控件的Id, 你就能知道你自己正在调试的那个控件到底在哪了?
- 通过setChromeColor, setChromeShadowColor可自定义线框图的颜色，随你喜欢。

太小了看不清怎么办？布局太深了里面的怎么看？不用担心，它提供了强大的手势功能。

- 单点触摸：可以控制其旋转角度。
- 两指垂直滑动：可以放大或缩小。
- 两指水平滑动：可以调整布局层次之间的间距，清晰看到深层次的布局结构。

值得注意的是，它要求修改根布局来实现功能，如果担心对线上用户的影响，可以仅在Debug模式下将其嵌入，类似下面这样几行代码即可在你的复杂项目中将其引入：

```
if (BuildConfig.DEBUG) {
 View mainView = getLayoutInflater()
 .inflate(R.layout.activity_main, null);
 mScalpelView = new ScalpelFrameLayout(this);
 mScalpelView.addView(mainView);
 setContentView(mScalpelView);
} else {
 setContentView(R.layout.activity_main);
}
```

微信公众号: androidtrending

其实从实用角度上来说，还是Hierarchy Viewer更强大，可为什么我们还需要Scalpel呢？除了不用找USB线连电脑外，最主要是因为它真的很酷啊，这就够了，不是吗？

如果喜欢这篇文章，记得点赞跟分享给你的好友哟。如果你还想了解更多Android开发最佳实践、技术前沿、最好用的工具与服务，请长按下方二维码关注我们，我会继续保持精品。

[项目地址](#)

# Systrace

# 使用Systrace分析UI性能

来源:[www.devtf.cn](http://www.devtf.cn)

原文链接 : [Analyzing UI Performance with Systrace](#)

原文作者 : [Android Developers](#)

译者 : [desmond1121](#)

开发应用的时候，应该检查它是否有流畅的用户体验，即60fps的帧率。如果由于某种原因丢帧，我们首先要做就是知道系统在做什么（造成丢帧的原因）。

Systrace允许你监视和跟踪Android系统的行为(trace)。它会告诉你系统都在哪些工作上花费时间、CPU周期都用在哪里，甚至你可以看到每个线程、进程在指定时间内都在干嘛。它同时还会突出观测到的问题，从垃圾回收到渲染内容都可能是问题对象，甚至提供给你建议的解决方案。本文章将介绍如何导出trace以及使用它来优化UI的办法。

## 总览

Systrace可以帮助你分析应用在不同Android系统上的运行情况。它将系统和应用的线程运行情况放置在同一条时间线上分析。你首先需要收集系统和应用的trace（后面会告诉你怎么做），之后Systrace会帮你生成一份细致、直观的报告，它展示了设备在你监测的这段时间内所发生的事情。

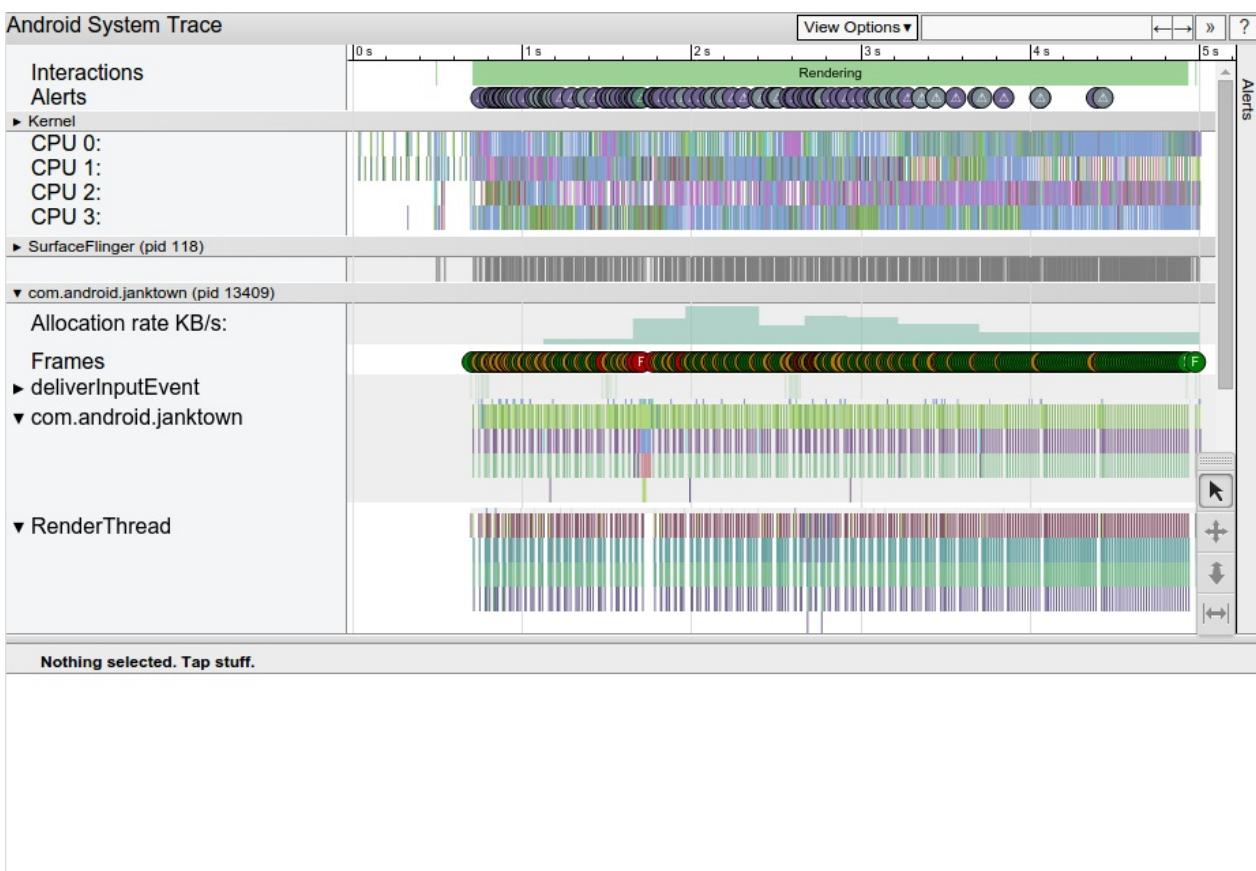


图1. 连续滑动应用5秒的Trace，它并没有表现得很完美。

图1展示了应用在滑动不流畅的时候生成的trace。默认缩放成全局显示，你可以放大到自己所关注的地方。横轴代表着时间线，事件记录按照进程分组，同一个进程中按线程进行纵向拆分，每个线程记录自己的工作。

在本例中，一共有三个组：Kernel, SurfaceFlinger, App，他们分别以包名为标识。每个应用进程都会包含其中所有线程的记录信号，你可以看到从InputEvent到RenderThread都有。

## 生成Trace

在获取trace之前需要做一些启动工作。首先，设备要求API $\geq$ 16(Android 4.1)，之后通过正常的Debug流程（开启调试、连接工作环境、安装App）连接设备。由于需要记录磁盘活动和内核工作，你可能需要root权限。不过大部分时候你只要能够正常Debug即可。

Systrace 可以通过[命令行](#)或者[图形界面](#)启动，本篇文章重点介绍通过命令行使用Systrace。

## 在Android 4.3及以上的系统中获取trace

在4.3以上的系统获取Trace步骤：

- 保证设备USB连接正常，并可以debug；
- 在命令行中设置选项，开启trace，比如：

```
$ cd android-sdk/platform-tools/systrace
$ python systrace.py --time=10 -o mynewtrace.html sched gfx view wm
```

在设备上做任何你想让trace记录的操作。

你可以通过[Systrace选项](#)来了解更多命令行选项。

## 在Android 4.2及以下的系统中获取trace

在4.2及以下的系统中高效地使用Systrace的话，你需要在配置的时候显式指定要trace的进程种类。一共有这两类种类：

- 普通系统进程，比如图形、声音、输入等。（通过tags设置，具体在[Systrace命令行](#)中有介绍）
- 底层系统进程，比如CPU、内核、文件系统活动。（通过options设置，具体在[Systrace命令行](#)中有介绍）

你可以通过以下命令行操作来设置tags：

- 使用--set-tags选项：

```
$ cd android-sdk/platform-tools/systrace
$ python systrace.py --set-tags=gfx,view,wm
```

- 重启adb shell来trace这些进程：

```
$ adb shell stop
$ adb shell start
```

你也可以通过手机上的图形界面设置tags：

- 1、在设备上进入设置>开发者选项>监控>启用跟踪（部分手机上没有这个选项）；
- 2、选择追踪进程类型，点击确认。

**注意：**在图形界面中设置tag时adb shell不用重新启动。

在配置完tags后，你可以开始收集操作信息了。

如何在当前设置下启动trace：

- 保证设备的usb连接正常，并且可以正常debug；
- 使用低系统等级的命令行选项开启trace，比如：

    \$ python systrace.py --cpu-freq --cpu-load --time=10 -o mytracefile.html

- 在设备上做任何你想让trace记录的操作。

你可以通过[Systrace选项](#)来了解更多命令行选项。

## 分析trace报告

在你获取trace之后你可以在网页浏览器中打开它。这部分内容告诉你怎么通过trace去分析和解决UI性能。

### 监视帧数

每个应用都有一行专门显示frame，每一帧就显示为一个绿色的圆圈。不过也有例外，当显示为黄色或者红色的时候，它的渲染时间超过了16.6ms（即达不到60fps的水准）。'w'键可以放大，看看这一帧的渲染过程中系统到底做了什么。

    提示：你可以按右上角的'?'按钮来查看界面使用帮助。

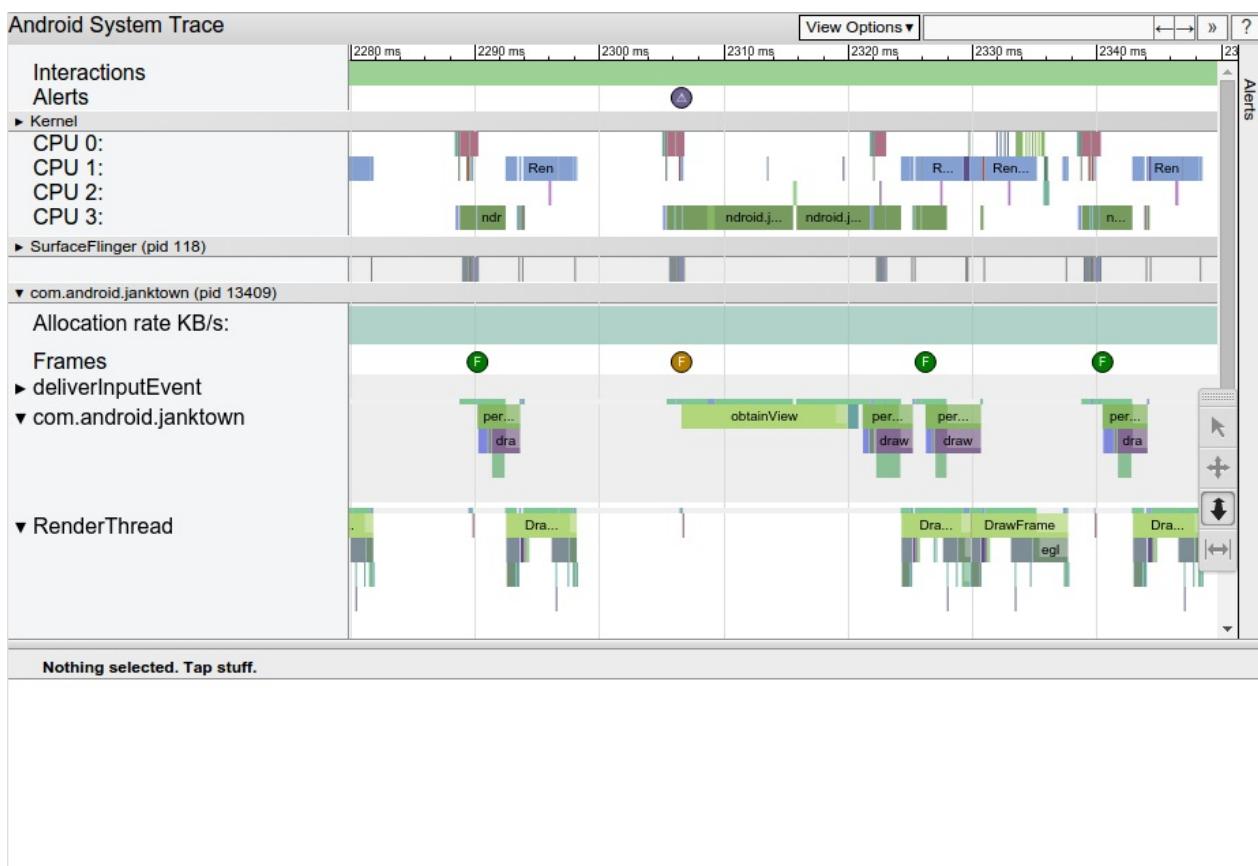


图2. Systrace显示长渲染时间的帧

单击该帧可以高亮它，这时候跟该帧有关的内容会被突出显示。在5.0及以上的系统中，显示工作被拆分成UI线程和Render线程两部分；在5.0以下的系统中，所有的显示工作在UI线程中执行。

点击单个Frame下面的组件可以看他们所花费的时间。每个事件（比如 `performTraversals`）都会在你选中的时候显示出它们调用了哪些方法及所用的时间。

## 调查警告事件

Systrace会自动分析事件，它会将任何它认为性能有问题的东西都高亮警告，并提示你要怎么去优化。

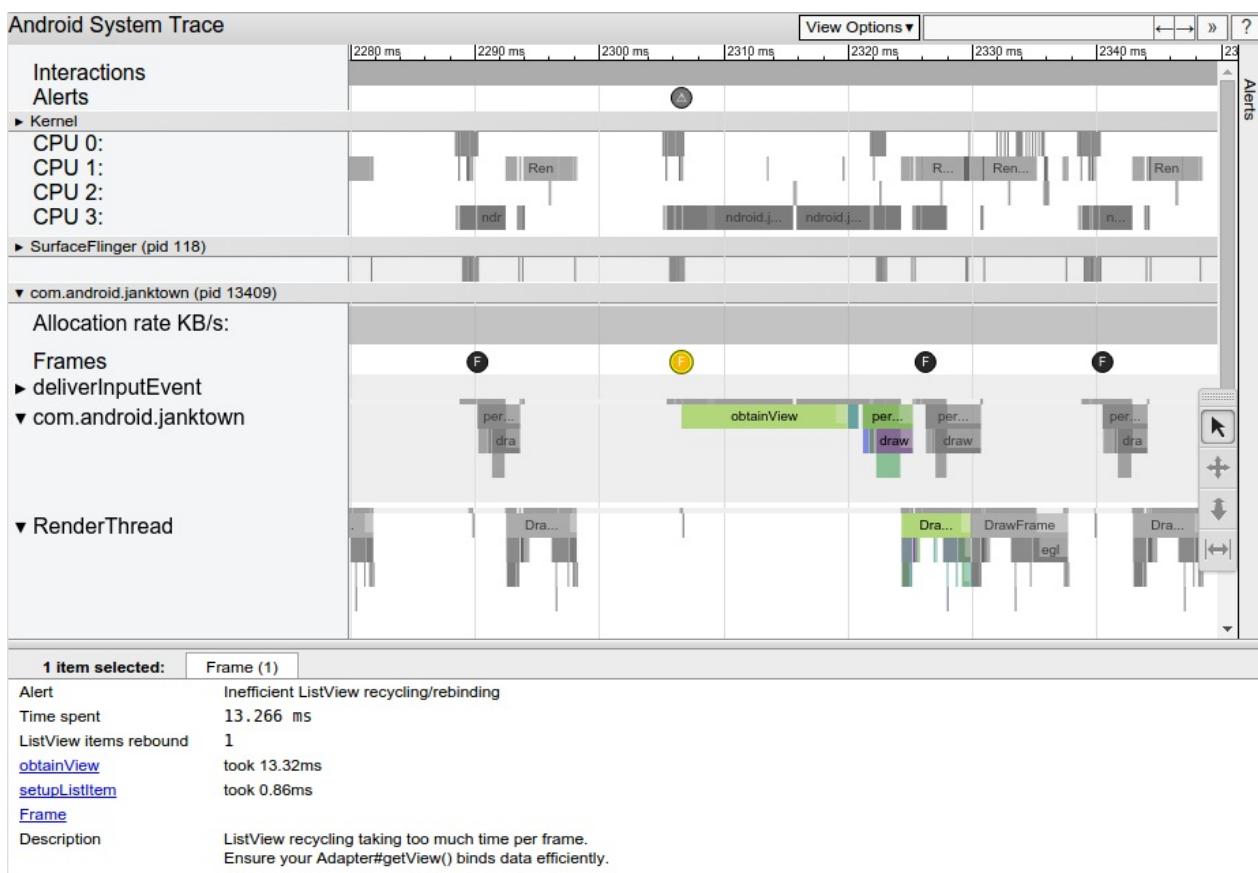


图3. 选择一个被高亮帧，它会显示出检测到的问题（回收*ListView*消耗时间太长）。在你选择类似图三中的问题帧之后，它就会提示你检测出的问题。在这个例子中，它被警告的主要原因是*ListView*的回收和重新绑定花费太多时间。在Systrace中也会提供一些对应链接，它们会提供更多解释。

如果你想知道UI线程怎么会花费这么多时间的话，你可以使用[TraceView](#)，它会告诉你都是哪些函数在消耗时间。

你可以通过右侧的'Alert'选项卡来查看整个trace过程中发生的所有问题，并进行快速定位。

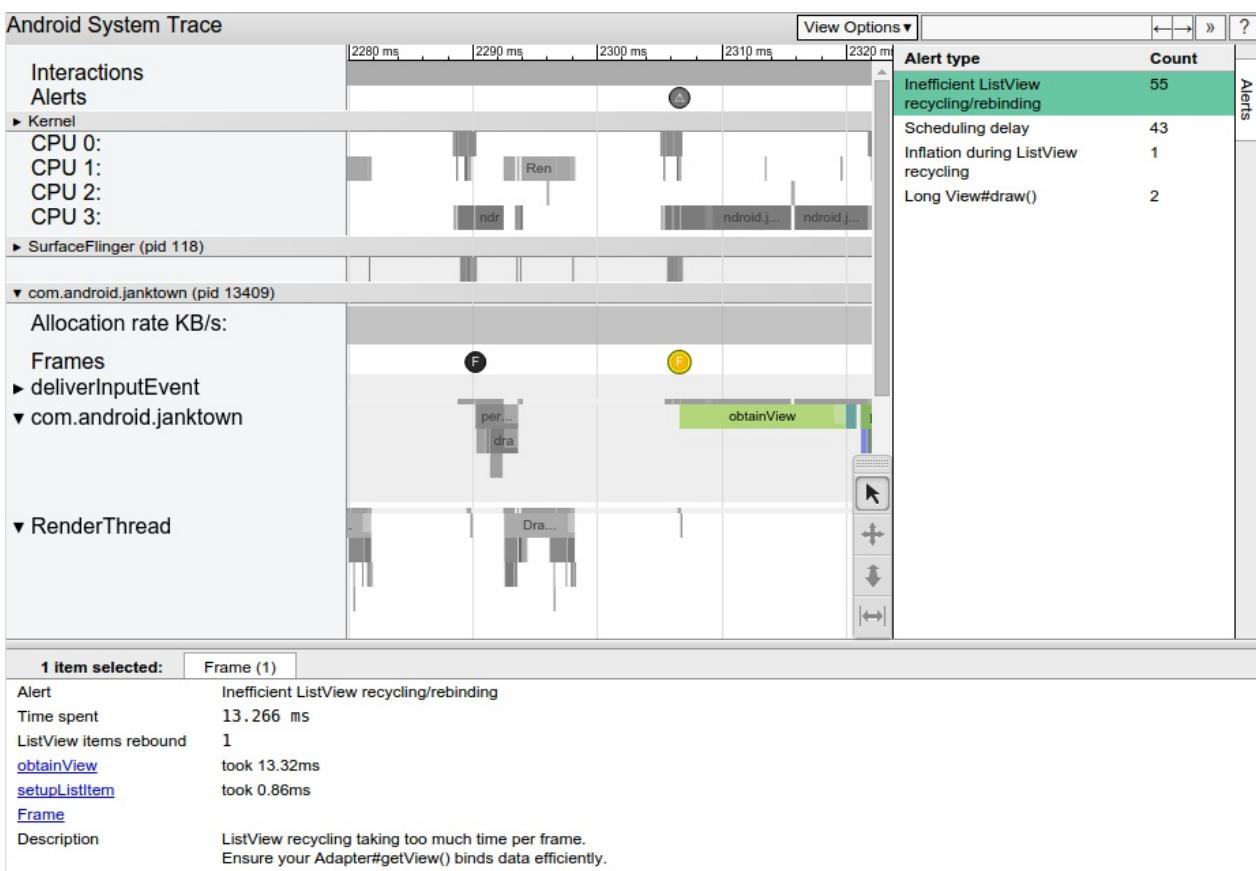


图4. 点击Alert选项卡。

你可以将Alert面板中的问题视为需要处理的bug，很有可能每一次微小的优化能够去除整个应用中的警告！

## 应用级别调试

Systrace并不会追踪应用的所有工作，所以你可以在有需求的情况下自己添加要追踪的代码部分。在Android 4.3及以上的代码中，你可以通过 `Trace` 类来实现这个功能。它能够让你在任何时候跟踪应用的一举一动。在你获取trace的过程  
中，`Trace.beginSection()` 与 `Trace.endSection()` 之间代码工作会一直被追踪。

下面这部分代码展示了使用Trace的例子，在整个方法中含有两个Trace块。

```
public void ProcessPeople() {
 Trace.beginSection("ProcessPeople");
 try {
 Trace.beginSection("Processing Jane");
 try {
 // 待追踪的代码
 } finally {
 Trace.endSection(); // 结束 "Processing Jane"
 }

 Trace.beginSection("Processing John");
 try {
 // 待追踪的代码
 } finally {
 Trace.endSection(); // 结束 "Processing John"
 }
 } finally {
 Trace.endSection(); // 结束 "ProcessPeople"
 }
}
```

注意：在Trace是被嵌套在另一个Trace中的时候，`endSection()` 方法只会结束理它最近的一个 `beginSection(String)`。即在一个Trace的过程中是无法中断其他Trace的。所以你要保证 `endSection()` 与 `beginSection(String)` 调用次数匹配。

注意：Trace的begin与end必须在同一线程之中执行！

当你使用应用级别追踪的时候，你必须通过 `-a` 或者 `-app=` 来显式地指定应用包名。可以通过[Systrace指南](#)查看更多关于它的信息。

你在评估应用的时候应该开启应用级别跟踪，即使当你没有手动添加 `Trace` 信号。因为很多库函数里面是有添加 `Trace` 信号的（比如 `RecyclerView`），它们往往能够提供很多信息。

# TraceView

# TraceView性能优化工具使用

来源:[wangxinghe.me](http://wangxinghe.me)

## 一、面板组成

### 1. Timeline Panel

- (1) 每一行表示一个线程
- (2) 每一种不同颜色代表一个不同的方法
- (3) 每一个bar对应一个方法执行过程， bar的宽度表示方法执行的时长
- (4) 对于缩小图， bar表示方法开始执行时间；对于放大图，每个方法对应的bar会扩大成一个有色的U型， U型的左边 表示方法开始时间， 右边表示方法结束时间
- (5) 小技巧：双击上面的时间轴可以缩小图，鼠标选中线程的颜色部分轻微水平拉动可以放大图

补充（参考4）：

- (1) 颜色脉冲bar的高度表示cpu的利用率，高度越高表示cpu利用率越高
- (2) 白色gap空白块表示该线程目前没有占CPU，被其他线程占用
- (3) 黑块表示系统空闲(system idle)

### 2. Profile Panel

- (1) 表示执行的方法列表
- (2) 任意点击一个方法展开， Parent表示调用该方法的方法， Children表示该方法调用的方法
- (3) 颜色和Timeline中的颜色含义相同，且选中一个方法会在Timeline中同步高亮
- (4) 参数解释：

- **Incl Cpu Time(%)** – 方法自身及该方法调用的所有子方法所占时间和（百分比）
- **Excl Cpu Time(%)** – 方法自身所占时间（百分比）
- **Incl Real Time(%)** – 类似于Incl Cpu Time(%)
- **Excl Real Time(%)** – 类似于Excl Cpu Time(%)
- **Calls + Recur Calls/Total** – 调用次数 + 递归调用次数。或某子方法被该父方法调用的次数 / 某子方法被所有方法调用的总次数。参考5。
- **Cpu Time/Call** – 方法调用一次所占Cpu Time。 $[Calls + Recur Calls/Total] * [Cpu$

Time/Call] = [Incl Cpu Time]

- **Real Time/Call** – 方法调用一次所占Real Time。[Calls + Recur Calls/Total] \* [Real Time/Call] = [Incl Real Time]

关于Cpu Time和Real Time的区别（参考6）：

- **Cpu Time** 指的是方法执行所占用的cpu时间，不包括中间的等待时间
- **Real Time** 指的是方法从开始执行到结束执行总的时间，包括中间的等待时间

## 二、产生trace logs的两种方式

- (1) 更精确方式：Debug.startMethodTracing(), Debug.stopMethodTracing(), 生成.trace文件，导出到PC目录adb pull /sdcard/\*.trace /tmp, 然后DDMS -> Open file

\* (2) 次精确方式：打开Tools->Android->Android Device Monitor。在Devices下选中进程，然后选中Start Method Profiling图标，然后在设备上进行可能存在性能问题的操作，然后点击Stop Method Profiling图标

## 三、关于分析方法

性能问题通常分为两类：

- (1) 调用次数不多，但每次耗时长的方法
- (2) 自身耗时不长，但频繁调用的方法

关于第一种，通常做法是先按Cpu Time/Call降序排序，然后看Incl Cpu Time的大小，综合起来越大的性能问题越严重

关于第二种，通常做法是按Calls + Recur Calls/Total降序排序，然后看Incl Cpu Time的大小，综合起来越大的性能问题越严重

总的来说，还是要自己多试验多体会，具体问题具体分析，我不便于给出绝对或误导人的结论，这里给出我认为很好的两篇博文，见参考文献2和3。

## 参考文献：

- (0) <http://developer.android.com/intl/zh-cn/tools/performance/traceview/index.html>
- (1) <http://developer.android.com/intl/zh-cn/tools/debugging/debugging-tracing.html>
- (2)

<http://jwzhangjie.cn/2015/07/14/android%E7%B3%BB%E7%BB%9F%E6%80%A7%E8%83%BD%E8%B0%83%E4%BC%98%E5%B7%A5%E5%85%B7%E4%BB%8B%E7%BB%8D/>

- (3) <http://myeyeofjava.iteye.com/blog/2250801>
- (4) <http://stackoverflow.com/questions/6476991/android-eclipse-traceview-i-just-dont-get-it>
- (5) <http://stackoverflow.com/questions/28125441/how-to-understanding-callsrecursions-total-in-android-traceview-tool/28149711#28149711>
- (6) <http://stackoverflow.com/questions/15760447/what-is-the-meaning-of-incl-cpu-time-excl-cpu-time-incl-real-cpu-time-excl-re>

# 静态检查工具

# 如何更好地利用Pmd、Findbugs和CheckStyle分析结果

来源:<http://www.importnew.com/11119.html>

本文由 ImportNew - Jmiracle1919 翻译自 [javadepend](#)。欢迎加入[翻译小组](#)。转载请见文末要求。

这里列出了很多Java静态分析工具，每一种工具关注一个特定的能发挥自己特长的领域，我们可以列举一下：

**Pmd**它是一个基于静态规则集的Java源码分析器，它可以识别出潜在的如下问题：

- 可能的bug——空的try/catch/finally/switch块。
- 无用代码(Dead code): 无用的本地变量，方法参数和私有方法。
- 空的if/while语句。
- 过度复杂的表达式——不必要的if语句，本来可以用while循环但是却用了for循环。
- 可优化的代码：浪费性能的String/StringBuffer的使用。

**FindBugs**它用来查找Java代码中存在的bug。它使用静态分析方法标识出Java程序中上百种潜在的不同类型的错误。

**Checkstyle**它定义了一系列可用的模块，每一个模块提供了严格程度(强制的，可选的...)可配置的检查规则。规则可以触发通知(notification)，警告(warning)和错误(error)。

现在有很多查看这些工具的处理结果的方式：

- **XML格式**: 这些工具都可以产生XML文件，这些XML文件能用来产生HTML报表或者是被别的工具用来浏览分析的结果。
- **HTML格式**: HTML格式是最受欢迎的产生报表和团队间分享的方式，你也可以用xsl表格创建你自己的报表。
- **IDE插件**: 几乎所有叫得上名字的IDE都给这些工具提供了插件，这使得发现源码中存在的所有问题几乎变成可能。

代码质量工具的一个问题是，它们有时候会给开发者提示很多不是错误的错误-也叫做假阳性(false positives)。当这种情况发生的时候，开发者可以学着忽略工具的输出信息，或者是把这些输出全部抛弃掉。

为了更好的利用这些工具的输出结果，给开发者一个更有用的视图，最好是有一种只关注我们想要的东西的方式。本文中，我们将找出其他有趣的方式来更好的利用所有这些有名的Java静态分析工具的输出结果，然后可以像查询数据库那样查询这些结果。

## JArchitect和CQLinq

JArchitect是另一个静态分析工具，它弥补了其他工具(的不足)，它是使用一种基于Linq(CQLinq)的代码查询语言像查询数据库那样来查询代码。

JArchitect3的以前版本，只能查询从JArchitect提取出来的分析数据，但是从JArchitect4开始，可以把许多其他静态分析工具的输出结果包含进来，然后使用CQLinq做查询。

让我们以PDT核心(Eclipse的Php插件)的源码为例来说明如何在JArchitect中利用好这些静态工具的分析结果。

在查询分析结果以前，要遵守以下几个步骤：

- 第一步：

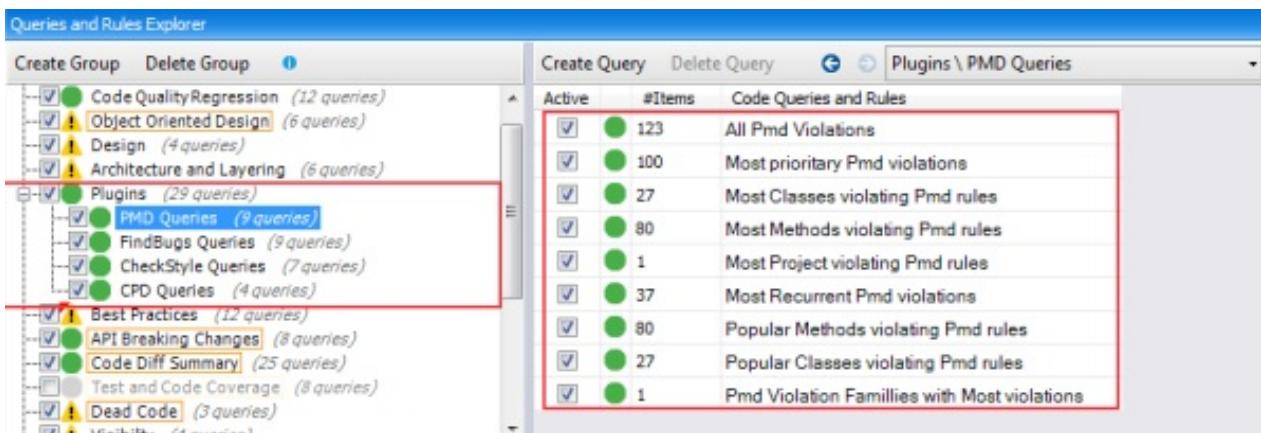
用PMD，CPD，FindBugs和CheckStyle分析项目工程，生成包含分析结果的XML文件。

- 第二步：

用JArchitect分析项目工程。

- 第三步：

在JArchitect点击菜单“插件(Plugins)”->“导入插件结果文件(Import Plugins Result Files)”把所有的XML文件导入到JArchitect中。



JArchitect默认给这些工具提供了许多有用的查询，并且这些查询都是可以很简单的进行定制的。

让我们来看一些CQLinq的查询：

## 获取的所有的问题(issue):

获取所有问题的请求很简单，但是没什么用处，因为如何利用23272个问题的分析结果确实是一个很大的挑战。

Queries and Rules Edit - 232 725 fields matched

from issue in Issues ¶s...\* from issue in Issues ¶s...\* X + □ ▾

**Critical** Report:

The query is not persisted! 336 ms

```
from issue in Issues
select new { issue,issue.FilePath,issue.BeginLine }
```

Group by: Export to Excel ▾

| fields                                    | FilePath | BeginLine  |
|-------------------------------------------|----------|------------|
| <b>232 725 fields matched</b>             |          |            |
| ▶  CheckStyle.javadoc (14 006 fields)     |          |            |
| ▶  CheckStyle.regexp (13 986 fields)      |          |            |
| ▶  CheckStyle.sizes (39 159 fields)       |          |            |
| ▶  CheckStyle.imports (265 fields)        |          |            |
| ▶  CheckStyle.whitespace (108 890 fields) |          |            |
| ▶  CheckStyle.modifier (2 163 fields)     |          |            |
| ▶  CheckStyle.checks (9 843 fields)       |          |            |
| ▶  CheckStyle.design (4 823 fields)       |          |            |
| ▶  CheckStyle.naming (3 944 fields)       |          |            |
| ▶  CheckStyle.coding (15 436 fields)      |          |            |
| ▶  FindBugs.STYLE (130 fields)            |          |            |
| ▶  CheckStyle.blocks (5 841 fields)       |          |            |
| ▶  FindBugs.PERFORMANCE (82 fields)       |          |            |
| ▶  FindBugs.MALICIOUS_CODE (132 fields)   |          |            |
| ▶  CPD.duplication (13 796 fields)        |          |            |
| ▶  Pmd.Import Statements (123 fields)     |          |            |
| ▶  FindBugs.CORRECTNESS (20 fields)       |          |            |
| ▶  FindBugs.BAD_PRACTICE (63 fields)      |          |            |
| ▶  FindBugs.MT_CORRECTNESS (11 fields)    |          |            |
| ▶  FindBugs.I18N (12 fields)              |          |            |
| Sum                                       | 0        | 920 993 .. |

Class Browser Queries and Rules Edit Search Results

为了更好的利用这些工具的分析结果，我们可以用CQLinq来做过滤，然后只关注那些我们想要关注的东西。

## 根据所使用的检查工具发请求

我们可以修改第一个请求，然后添加一个查询工具的criteria。

The screenshot shows the 'Queries and Rules Edit' interface with the title 'Queries and Rules Edit - 450 fields matched'. At the top, there are two search bars: 'from issue in ...\*' and 'from issue in ...\*', followed by a dropdown menu 'All FindBugs Vi...'. Below the search bar is a toolbar with icons for file operations (New, Open, Save, Print, Copy, Paste, Find, Replace) and a gear icon for settings.

The main area displays a CQLinq query:

```
// <Name>All FindBugs Violations</Name>
from issue in Issues where issue.ToolName=="FindBugs"
select new { issue,issue.FilePath,issue.BeginLine }
```

Below the query, a status bar indicates '12 ms'.

At the bottom of the interface, there are buttons for 'Group by' (with icons for Date, File, Class, Method, Line), 'Export to Excel', and tabs for 'Class Browser', 'Queries and Rules Edit', and 'Search Results'.

The central part of the interface is a table titled '450 fields matched' with columns 'fields', 'FilePath', and 'BeginLine'. The table lists various FindBugs categories and their counts:

| fields                                 | FilePath      | BeginLine |
|----------------------------------------|---------------|-----------|
| <b>450 fields matched</b>              |               |           |
| ▶ FindBugs.STYLE (130 fields)          |               |           |
| ▶ FindBugs.PERFORMANCE (82 fields)     |               |           |
| ▶ FindBugs.MALICIOUS_CODE (132 fields) |               |           |
| ▶ FindBugs.CORRECTNESS (20 fields)     |               |           |
| ▶ FindBugs.BAD_PRACTICE (63 fields)    |               |           |
| ▶ FindBugs.MT_CORRECTNESS (11 fields)  |               |           |
| ◀ FindBugs.I18N (12 fields)            |               |           |
| ◀ I18N (12 fields)                     |               |           |
| DM_DEFAULT_ENCODING                    | C:\work\pd... | 86        |
| DM_DEFAULT_ENCODING                    | C:\work\pd... | 289       |
| DM_DEFAULT_ENCODING                    | C:\work\pd... | 310       |
| DM_DEFAULT_ENCODING                    | C:\work\pd... | 2 280     |
| DM_DEFAULT_ENCODING                    | C:\work\pd... | 1 266     |
| DM_DEFAULT_ENCODING                    | C:\work\pd... | 1 262     |
| DM_DEFAULT_ENCODING                    | C:\work\pd... | 205       |
| DM_DEFAULT_ENCODING                    | C:\work\pd... | 61        |
| DM_DEFAULT_ENCODING                    | C:\work\pd... | 124       |
| Sum                                    | 0             | 256 389   |

## 据规则集发请求

我们也可以根据问题的规则集做过滤：

The screenshot shows the 'Queries and Rules Edit' interface with the following details:

- Top Bar:** 'Queries and Rules Edit - 3 944 fields matched'. Buttons for 'Most Classes viola...\*', 'Most Classes viola...\*', 'TODO short descri...', and 'Report'.
- Message:** 'The query is not persisted!' and '42 ms'.
- Query:**

```
// <Name>TODO short description</Name>
from issue in Issues where issue.IssueFamily.Contains("naming")
select new { issue,issue.FilePath,issue.BeginLine }
```
- Toolbar:** Buttons for 'New', 'Open', 'Save', 'Group by', 'Export to Excel'.
- Table:** A grid showing the results of the search. The columns are 'fields', 'FilePath', and 'BeginLine'.
 

| fields                                                                                     | FilePath    | BeginLine  |
|--------------------------------------------------------------------------------------------|-------------|------------|
| <b>3 944 fields matched</b>                                                                |             |            |
| <b>CheckStyle.naming (3 944 fields)</b>                                                    |             |            |
| Name 'NonPhP' must match pattern "[A-Z][A-Z0-9]"                                           | C:\work\... | 25         |
| Name 'Internal' must match pattern "[A-Z][A-Z0-9]"                                         | C:\work\... | 32         |
| Name 'Constructor' must match pattern "[A-Z][A-Z0-9]"                                      | C:\work\... | 37         |
| Name 'AccTrait' must match pattern "[A-Z][A-Z0-9]"                                         | C:\work\... | 39         |
| Name 'AccMagicProperty' must match pattern "[A-Z][A-Z0-9]"                                 | C:\work\... | 40         |
| Name 'ID' must match pattern "[a-zA-Z][a-zA-Z0-9]*\$".                                     | C:\work\... | 14         |
| Name 'PHPCorePlugin_initializingPHPToolkit' must match pattern "[a-zA-Z][a-zA-Z0-9]*\$".   | C:\work\... | 48         |
| Name 'PHPCorePlugin_initializingSearchEngine' must match pattern "[a-zA-Z][a-zA-Z0-9]*\$". | C:\work\... | 49         |
| Name 'PHPTodoTaskAstParser_0' must match pattern "[a-zA-Z][a-zA-Z0-9]*\$".                 | C:\work\... | 50         |
| Name '_log' must match pattern "[a-zA-Z][a-zA-Z0-9]*\$".                                   | C:\work\... | 53         |
| Name '_trace' must match pattern "[a-zA-Z][a-zA-Z0-9]*\$".                                 | C:\work\... | 93         |
| Name 'AccClassField' must match pattern "[A-Z][A-Z0-9]*\$".                                | C:\work\... | 210        |
| Name 'isDebugEnabled' must match pattern "[A-Z][A-Z0-9]*\$".                               | C:\work\... | 312        |
| Name 'PHPFacets_SettingVersionFailed' must match pattern "[A-Z][A-Z0-9]*\$".               | C:\work\... | 19         |
| Name 'instance' must match pattern "[A-Z][A-Z0-9]*\$".                                     | C:\work\... | 31         |
| <b>Sum</b>                                                                                 | 0           | 22 995 836 |
- Bottom Navigation:** Buttons for 'Class Browser', 'Queries and Rules Edit', and 'Search Results'.

## 根据优先级发请求

也可以根据优先级做过滤：

Queries and Rules Edit - 100 fields matched

Most priority Pmd violations

Critical Report:

Plugins \ PMD Queries 2 ms

```
// <Name>Most priority Pmd violations</Name>
(from issue in Issues where issue.ToolName=="Pmd"
orderby issue.Priority ascending
select new { issue,issue.Priority,issue.FilePath,issue.BeginLine }).
```

Group by: Export to Excel

| fields                                    | Priority        | FilePath | BeginLine |
|-------------------------------------------|-----------------|----------|-----------|
| <b>100 fields matched</b>                 |                 |          |           |
| org.eclipse.php.core_3.2.0.3 (100 fields) |                 |          |           |
| Pmd.Import Statements (100 fields)        |                 |          |           |
| Import Statements (100 fields)            |                 |          |           |
| Avoid unused imports such as 'jav 4       | C:\work\pdt\... | 14       |           |
| Avoid duplicate imports such as 'c 4      | C:\work\pdt\... | 19       |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 275      |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 273      |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 271      |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 269      |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 267      |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 265      |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 263      |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 261      |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 259      |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 257      |           |
| Unnecessary use of fully qualifie 4       | C:\work\pdt\... | 255      |           |
| Sum                                       | 400             | 0        | 224 811   |

Class Browser Queries and Rules Edit Search Results

## 出现次数最多的问题

知道哪些问题是被这些工具报告次数最多的是很有用的。

Queries and Rules Edit - 4 590 fields matched

Most Recurrent violations\* ✎ X All CheckStyle Violations ✎ + -

Critical Report:

Plugins \ PMD Queries 102 ms

```
// <Name>Most Recurrent violations</Name>
let groups = Application.Issues.GroupBy(t => t.Type)
from g in groups
orderby g.Count() descending
select new { issue=g.FirstOrDefault(),count=g.Count() }
```

Group by: Export to Excel ▾

| fields                                                 | count   |
|--------------------------------------------------------|---------|
| <b>4 590 fields matched</b>                            |         |
| org.eclipse.php.core_3.2.0.3 (4 590 fields)            | 27 506  |
| '+ ' should be on a new line.                          | 23 823  |
| 'cast' is not followed by whitespace.                  | 23 754  |
| '-' is not followed by whitespace.                     | 23 752  |
| '-' is not preceded with whitespace.                   | 13 986  |
| Line has trailing spaces.                              | 10 186  |
| Missing a Javadoc comment.                             | 7 428   |
| '+' is not preceded with whitespace.                   | 4 593   |
| Avoid nested blocks.                                   | 3 093   |
| Name 'RESULT' must match pattern "[a-zA-Z][a-zA-Z0-9]" | 1 823   |
| Line is longer than 80 characters (found 81).          | 1 752   |
| Redundant 'public' modifier.                           | 1 682   |
| Line is longer than 80 characters (found 131).         | 1 578   |
| '3' is a magic number.                                 | 1 339   |
| Line is longer than 80 characters (found 82).          | 1 218   |
| Line is lonner than 80 characters (found 121)          | 1 201   |
| Sum                                                    | 218 475 |

Class Browser Queries and Rules Edit Search Results

## 出现问题最多的类

知道哪些类包含了最多的问题是很有用的。

Queries and Rules Edit - 100 types matched

Most Classes violating Pmd rules\* ✎ X

Critical Report:

Plugins \ PMD Queries 54 ms

```
// <Name>Most Classes violating Pmd rules</Name>
(from t in Types where t.Issues.Count()>0
orderby t.Issues.Count() descending
select new { t,nbIssues=t.Issues.Count(),
CheckStyle=t.Issues.Count(i=>i.ToolName=="CheckStyle"),
Pmd=t.Issues.Count(i=>i.ToolName=="Pmd"),
FindBugs=t.Issues.Count(i=>i.ToolName=="FindBugs")
}).Take(100)
```

Group by: Export to Excel ▾

| types                    | nbIssues       | CheckStyle     | Pmd       | FindBugs   |
|--------------------------|----------------|----------------|-----------|------------|
| <b>100 types matched</b> |                |                |           |            |
| CompilerAstParser        | 21 848         | 21 841         | 1         | 6          |
| PhpAstParser             | 20 440         | 20 433         | 1         | 6          |
| CompilerAstParser        | 19 674         | 19 667         | 1         | 6          |
| PhpAstParser             | 18 227         | 18 220         | 1         | 6          |
| CompilerAstParser        | 16 988         | 16 981         | 1         | 6          |
| PhpAstParser             | 15 839         | 15 832         | 1         | 6          |
| CompilerAstParser        | 14 562         | 14 555         | 1         | 6          |
| PhpAstParser             | 13 639         | 13 632         | 1         | 6          |
| PhpAstLexer              | 2 916          | 2 900          | 6         | 10         |
| PhpLexer                 | 2 910          | 2 899          | 0         | 11         |
| PhpAstLexer              | 2 834          | 2 818          | 6         | 10         |
| PhpLexer                 | 2 830          | 2 819          | 0         | 11         |
| PhpAstLexer              | 2 786          | 2 770          | 6         | 10         |
| PhpLexer                 | 2 681          | 2 671          | 0         | 10         |
| PhnAstLexer              | 7 452          | 7 447          | 1         | 10         |
| <b>Sum</b>               | <b>192 164</b> | <b>191 847</b> | <b>64</b> | <b>253</b> |

Class Browser Queries and Rules Edit Search Results

上图可以看出来，CheckStyle报告的上千个问题中有很多是可以忽略的。

前面的查询很有用，但是，它并没有给我们一个精确的类质量的信息，因为要考虑的另一个有用的维度就是代码行数(NBLinesOfCode)。一般来说代码行数多的类会包含更多的问题，基于这个考虑，我们可以修改之前的请求来计算出问题数目和代码行数(NBLinesOfCode)的比率。

Queries and Rules Edit - 100 types matched

Most Classes violating Pmd rules\*      Most Classes violating Pmd rules\*

Critical Report:

The query is not persisted! 9 ms

```
// <Name>Most Classes violating Pmd rules</Name>
(from t in Types where t.Issues.Count()>0 && t.NbLinesOfCode>0
let ratio=t.Issues.Count()/t.NbLinesOfCode
orderby ratio descending
select new { t,nbIssues=t.Issues.Count(),Ratio=ratio
}).Take(100)
```

Group by: Export to Excel

| types                                   | nbIssues | Ratio |
|-----------------------------------------|----------|-------|
| 100 types matched                       |          |       |
| CompilerAstParser                       | 21 848   | 642   |
| CompilerAstParser                       | 19 674   | 578   |
| PhpAstParser                            | 20 440   | 511   |
| CompilerAstParser                       | 16 988   | 499   |
| PhpAstParser                            | 18 227   | 455   |
| PhpAstParser                            | 15 839   | 395   |
| CompilerAstParser                       | 14 562   | 264   |
| PhpAstParser                            | 13 639   | 239   |
| PHPFacetsConstants                      | 15       | 15    |
| InstallActionDelegate                   | 12       | 12    |
| VersionChangeActionDelegate             | 12       | 12    |
| RewriteEvent                            | 12       | 12    |
| PHPTypeInference                        | 43       | 10    |
| PhplIndexingVisitorExtension            | 25       | 8     |
| GlobalMethodStatementContextForTemplate | 8        | 8     |
| PHPSourceElementRequestorExtension      | 36       | 7     |
| Message                                 | 14       | 7     |
| Sum                                     | 144 465  | 4 000 |

Class Browser Queries and Rules Edit Search Results

上面的查询结果看上去很奇怪，前8个类的问题数和代码行数比率超过了200，也就是说一行代码有超过200个问题。

为了解释这种行为，我们看下CompilerAstParser的一些代码：



```

public CompilerAstParser() {super();}

/** Constructor which sets the default scanner. */
public CompilerAstParser(java_cup.runtime.Scanner s) {super(s);}

/** Production table. */
protected static final short _production_table[][] =
 unpackFromStrings(new String[] {
 "\u000\u01dc\000\002\003\003\000\002\002\004\000\002\004" +
 "\003\000\002\005\003\000\002\005\005\000\002\010\004" +
 "\000\002\010\002\000\002\011\004\000\002\011\002\000" +
 "\002\012\003\000\002\012\003\000\002\012\003\000\002" +
 "\012\006\000\002\220\002\000\002\012\006\000\002\221" +
 "\002\000\002\012\005\000\002\012\005\000\002\012\004" +
 "\000\002\126\003\000\002\126\003\000\002\127\005\000" +
 "\002\127\003\000\002\007\005\000\002\007\003\000\002" +
 "\006\003\000\002\006\005\000\002\006\004\000\002\006" +
 "\006\000\002\014\004\000\002\014\002\000\002\015\003" +
 "\000\002\015\003\000\002\015\003\000\002\015\006\000" +
 "\002\013\003\000\002\013\004\000\002\013\003\000\002" +
 "\013\003\000\002\222\002\000\002\016\006\000\002\016" +
 "\011\000\002\016\014\000\002\016\007\000\002\016\011" +
 "\000\002\016\013\000\002\016\007\000\002\016\004\000" +
 "\002\016\005\000\002\016\004\000\002\016\005\000\002" +
 "\016\004\000\002\016\005\000\002\016\005\000\002\016" +
 "\005\000\002\016\005\000\002\016\005\000\002\016\003" +
 "\000\002\016\004\000\002\016\005\000\002\016\007\000" +
 "\002\016\012\000\002\016\012\000\002\016\007\000\002" +
 "\016\003\000\002\016\017\000\002\016\005\000\002\016" +
 "\005\000\002\137\003\000\002\137\002\000\002\140\003" +
 "\000\002\140\004\000\002\141\012\000\002\017\003\000" +
 "\000\002\140\004\000\002\141\012\000\002\017\003\000" +
 ...
 });

```

代码行数(NBLinesOfCode)指的是语句的数目而不是代码的物理行数，CompilerAstParser这个类声明了很多数组，每一个都包含了几千个物理行，但是，每一个数组都被认为是一个语句。

就像前面展示的出现次数最多的问题那样，每一个数组都把”+应该在一个新行上”这个规则违反了上千次。或许最好是应该把这样的规则从CheckStyle的配置文件中删掉。

## 出问题最多的方法

当静态警察工具报告了问题以后，定位解决问题的优先级是很有用的，尤其是当包含bug的时候。

bug可能存在于某一个特定的方法中，但是，知道还有多少方法也受这个bug的影响是非常有用的。知道了出问题最多的这个方法做好事尽快把它解决掉。

Queries and Rules Edit - 100 methods matched

Most Recurrent vi...\* Popular Methods...\* Popular Methods v... X +

Critical Report:

Plugins \ FindBugs Queries 60 ms

```
// <Name>Popular Methods violating FindBugs rules</Name>
(from m in Methods where m.Issues.Where(i=>i.ToolName=="FindBugs").Count() > 0)
orderby m.MethodsCallingMe.Count() descending
select new { m, NbCallingMethods= m.MethodsCallingMe }.Take(100)
```

| methods                                    | NbCallingMethods |
|--------------------------------------------|------------------|
| <b>100 methods matched</b>                 |                  |
| org.eclipse.php.core_3.2.0.3 (100 methods) |                  |
| getReferences()                            | 13 methods       |
| getLineFormPosition(int)                   | 9 methods        |
| addDeclarationStatement(Statement)         | 8 methods        |
| extractNamespaceName(String,ISourceM       | 8 methods        |
| getIdFromPos(int,int)                      | 7 methods        |
| isPHPQuotesState(String)                   | 6 methods        |
| getTags(int)                               | 5 methods        |
| _log(int,String,Throwable)                 | 4 methods        |
| getStructuralProperty(StructuralPropertyD  | 4 methods        |
| getLhsTypes()                              | 4 methods        |
| getPartitionType(IStructuredDocument,int,  | 4 methods        |
| getRegionType(IStructuredDocument,int)     | 4 methods        |
| Sum                                        | 0                |

Class Browser Queries and Rules Edit Search Results

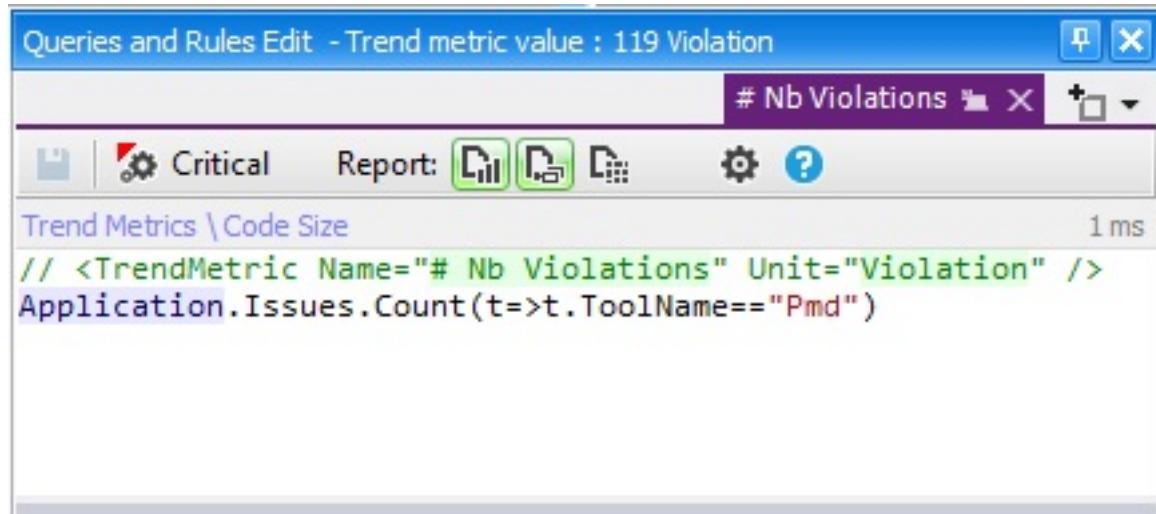
使用CQLinq，我们可以把这些工具的结果和JArchitect的结果结合起来创建出更复杂的查询，然后把这些检查规则添加到构建过程中去。

## 问题的趋势

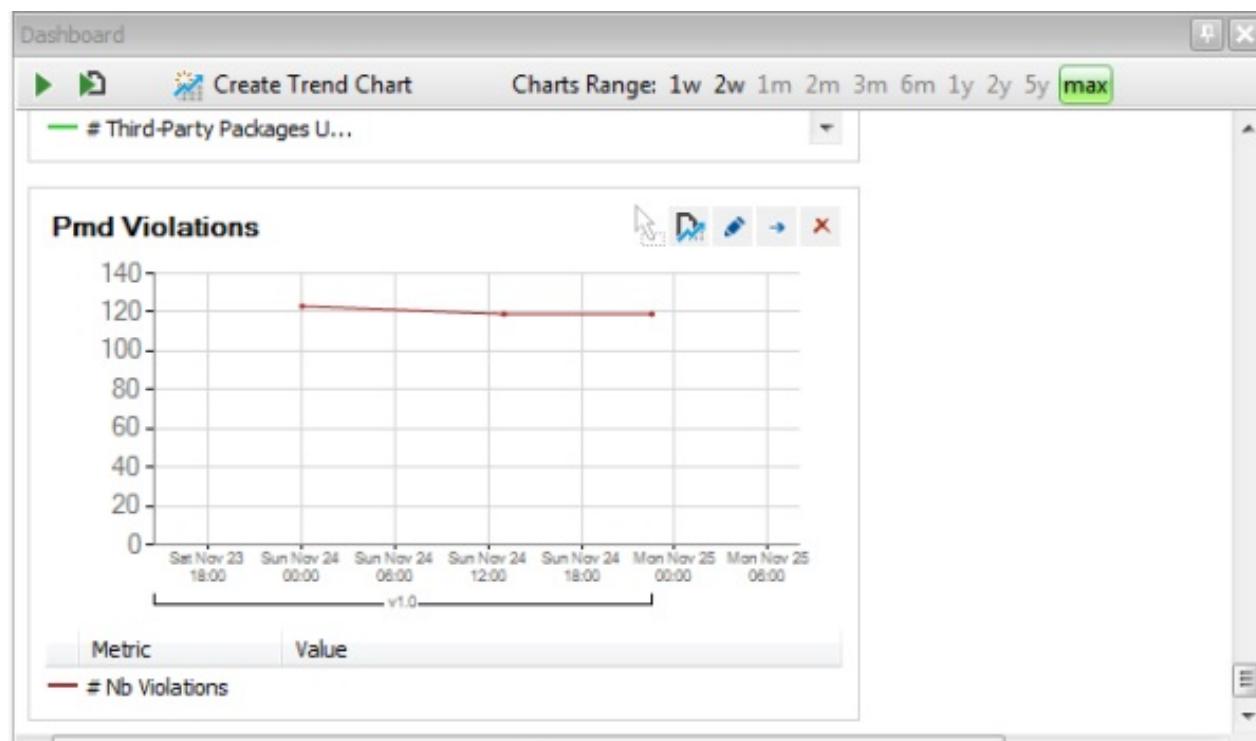
工程中有问题并不是异常情况，我们甚至可以说是正常的，但是，我们要检查工程的质量趋势。如果随着工程的更新和演化问题数目增加了，将会是一个很坏的指标。

JArchitect提供了趋势监控特性来创建趋势图。趋势图是根据分析时间记录的特定维度上的值创建出来的。默认有50多个趋势维度，也可以很简单定制趋势维度。

下面给Pmd问题创建一个趋势维度：



然后，你就可以很简单的创建趋势图在趋势维度上做监控，然后把它添加到JArchitect的操作面板中。

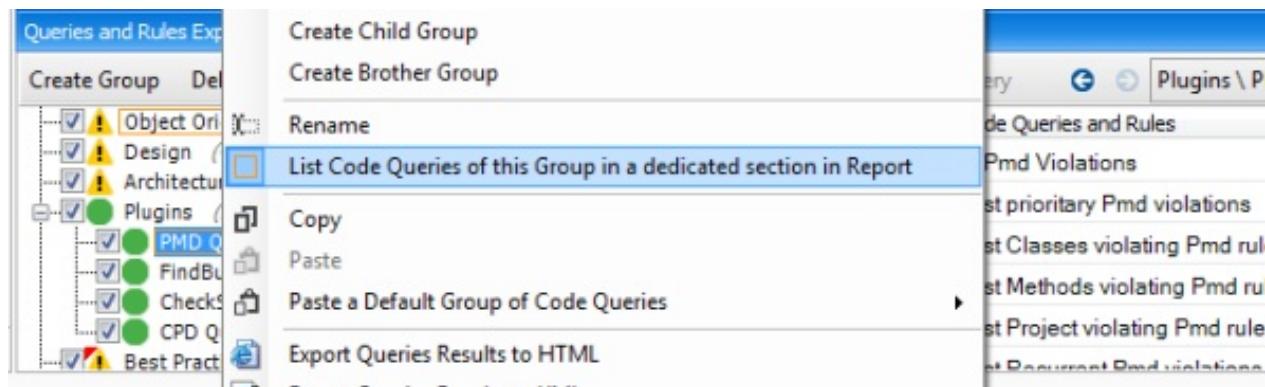


有了这个趋势图，我们就可以监视Pmd问题的进化，然后发现这个维度的问题随版本进化的原因。

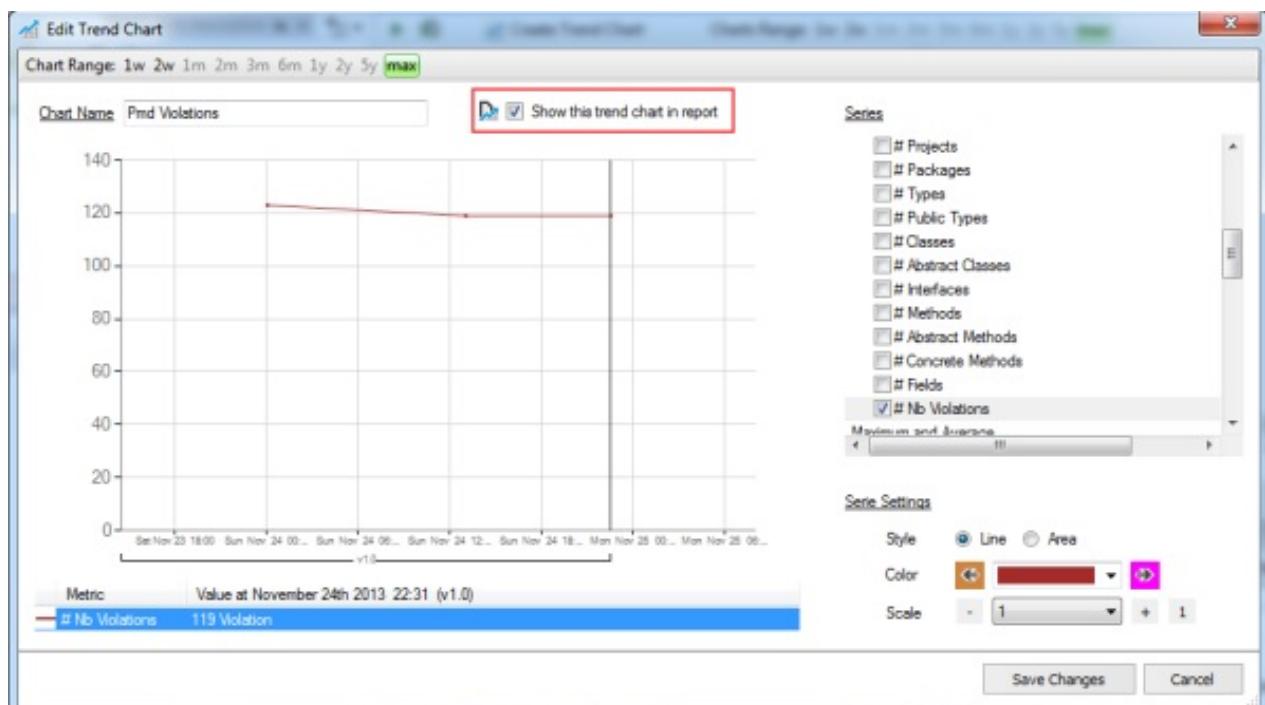
## 定制JArchitect报表

JArchitect可以在列出了CQLinq查询的HTML报表中追加额外的报表区。

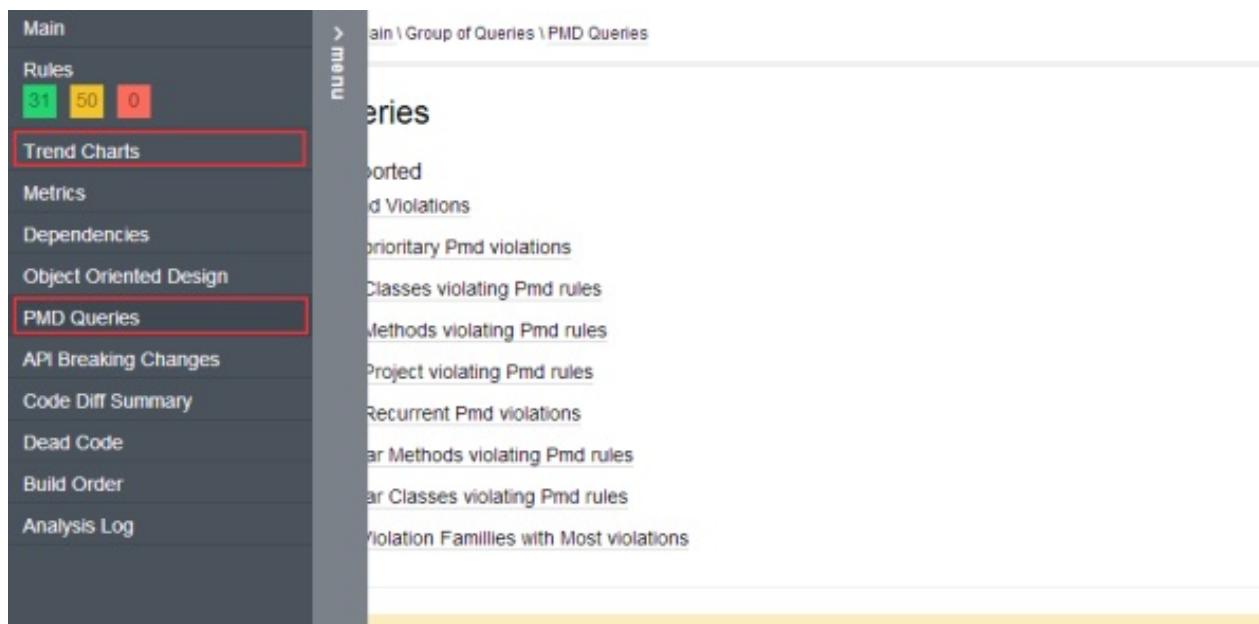
在CQLinq查询浏览面板中，一个特定的CQLinq组是用橙色的边框包围的。



也可以把Pmd趋势图添加到报表中：



在HTML报表中，这些被添加进来的区域可以通过菜单访问：



这是被添加进Pmd查询报表中的页面：

This screenshot shows a detailed view of the 'All Pmd Violations' query from the previous report. It includes a legend for the nine categories listed above, followed by the SQL query code, and a note indicating 119 fields matched. The URL in the browser bar is 'Main \ Group of Queries \ PMD Queries \ All Pmd Violations'.

```
// <Name>All Pmd Violations</Name>
from issue in Issues where issue.ToolName=="Pmd"
select new { issue,issue.FilePath,issue.BeginLine }
```

119 fields matched

## 结论

JArchitect 4 对其他的静态分析工具是开放的，你也可以很简单的像本文说的那样把自己的工具做成它的插件。这样你就可以使用JArchitect的所有功能来更好的利用那些有名的java静态分析工具的分析结果。



# 常用Java静态代码分析工具的分析与比较

来源:[稀土掘金](#)、[oschina](#)



**简介：**本文首先介绍了静态代码分析的基本概念及主要技术，随后分别介绍了现有 4 种主流 Java 静态代码分析工具 (Checkstyle, FindBugs, PMD, Jtest)，最后从功能、特性等方面对它们进行分析和比较，希望能够帮助 Java 软件开发人员了解静态代码分析工具，并选择合适的工具应用到软件开发中。

在 Java 软件开发过程中，开发团队往往要花费大量的时间和精力发现并修改代码缺陷。Java 静态代码分析 (static code analysis) 工具能够在代码构建过程中帮助开发人员快速、有效的定位代码缺陷并及时纠正这些问题，从而极大地提高软件可靠性并节省软件开发和测试成本。目前市场上的 Java 静态代码分析工具种类繁多且各有千秋，因此本文将分别介绍现有 4 种主流 Java 静态代码分析工具 (Checkstyle, FindBugs, PMD, Jtest)，并从功能、特性等方面对它们进行分析和比较，希望能够帮助 Java 软件开发人员了解静态代码分析工具，并选择合适的工具应用到软件开发中。

## 静态代码分析工具简介

### 什么是静态代码分析

静态代码分析是指无需运行被测代码，仅通过分析或检查源程序的语法、结构、过程、接口等来检查程序的正确性，找出代码隐藏的错误和缺陷，如参数不匹配，有歧义的嵌套语句，错误的递归，非法计算，可能出现的空指针引用等等。

在软件开发过程中，静态代码分析往往先于动态测试之前进行，同时也可作为制定动态测试用例的参考。统计证明，在整个软件开发生命周期中，30% 至 70% 的代码逻辑设计和编码缺陷是可以通过静态代码分析来发现和修复的。

但是，由于静态代码分析往往要求大量的时间消耗和相关知识的积累，因此对于软件开发团队来说，使用静态代码分析工具自动化执行代码检查和分析，能够极大地提高软件可靠性并节省软件开发和测试成本。

## 静态代码分析工具的优势

1. 帮助程序开发人员自动执行静态代码分析，快速定位代码隐藏错误和缺陷。
2. 帮助代码设计人员更专注于分析和解决代码设计缺陷。
3. 显著减少在代码逐行检查上花费的时间，提高软件可靠性并节省软件开发和测试成本。

## Java 静态代码分析理论基础和主要技术

- 缺陷模式匹配：缺陷模式匹配事先从代码分析经验中收集足够多的共性缺陷模式，将待分析代码与已有的共性缺陷模式进行模式匹配，从而完成软件的安全分析。这种方式的优点是简单方便，但是要求内置足够多缺陷模式，且容易产生误报。
- 类型推断：类型推断技术是指通过对代码中运算对象类型进行推理，从而保证代码中每条语句都针对正确的类型执行。这种技术首先将预定义一套类型机制，包括类型等价、类型包含等推理规则，而后基于这一规则进行推理计算。类型推断可以检查代码中的类型错误，简单，高效，适合代码缺陷的快速检测。
- 模型检验：模型检验建立于有限状态自动机的概念基础之上，这一理论将被分析代码抽象为一个自动机系统，并且假设该系统是有限状态的、或者是可以通过抽象归结为有限状态。模型检验过程中，首先将被分析代码中的每条语句产生的影响抽象为一个有限状态自动机的一个状态，而后通过分析有限状态机从而达到代码分析的目的。模型检验主要适合检验程序并发等时序特性，但是对于数据值域数据类型等方面作用较弱。
- 数据流分析：数据流分析也是一种软件验证技术，这种技术通过收集代码中引用到的变量信息，从而分析变量在程序中的赋值、引用以及传递等情况。对数据流进行分析可以确定变量的定义以及在代码中被引用的情况，同时还能够检查代码数据流异常，如引用在前赋值在后、只赋值无引用等。数据流分析主要适合检验程序中的数据域特性。

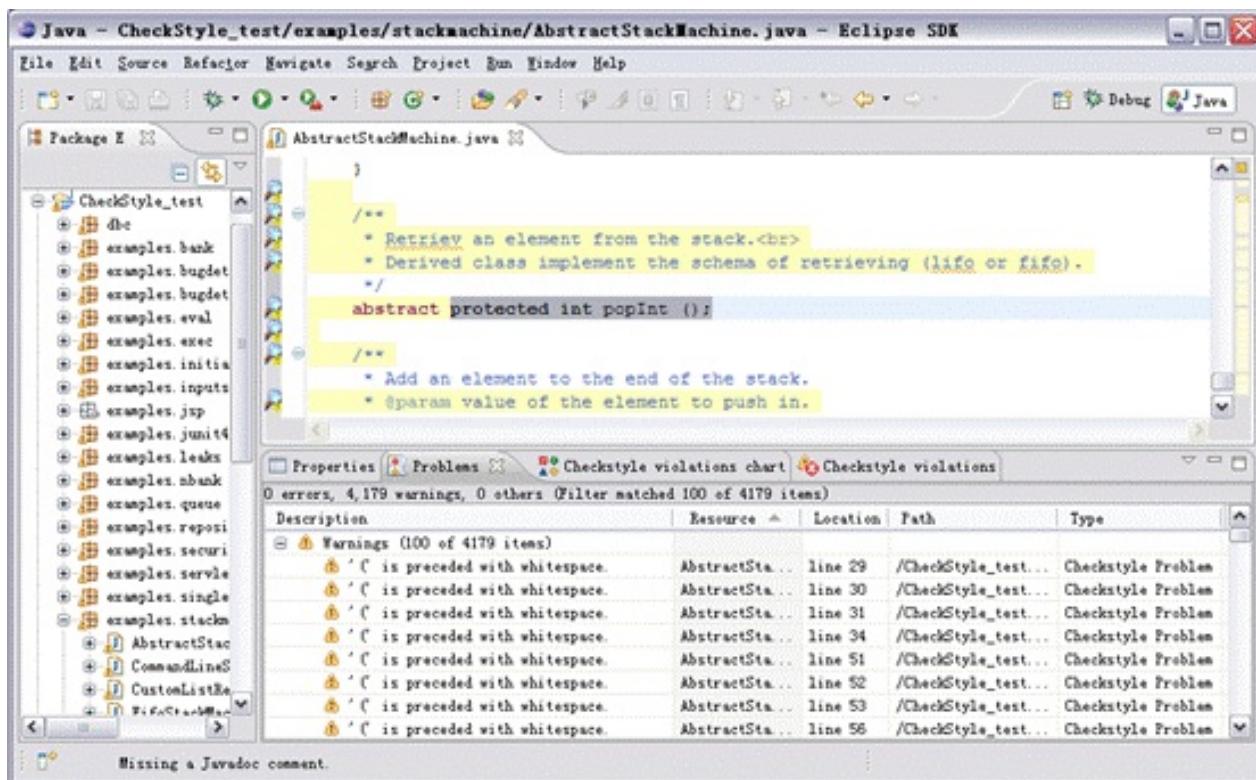
## 现有主流 Java 静态分析工具

## Checkstyle

Checkstyle 是 SourceForge 的开源项目，通过检查对代码编码格式，命名约定，Javadoc，类设计等方面进行代码规范和风格的检查，从而有效约束开发人员更好地遵循代码编写规范。

Checkstyle 提供了支持大多数常见 IDE 的插件，文本主要使用 Eclipse 中的 Checkstyle 插件。如下图 1 所示，Checkstyle 对代码进行编码风格检查，并将检查结果显示在 Problems 视图中。图中，代码编辑器中每个放大镜图标表示一个 Checkstyle 找到的代码缺陷。开发人员可通过在 Problems 视图中查看错误或警告详细信息。

图 1. 使用 Checkstyle 进行编码风格检查



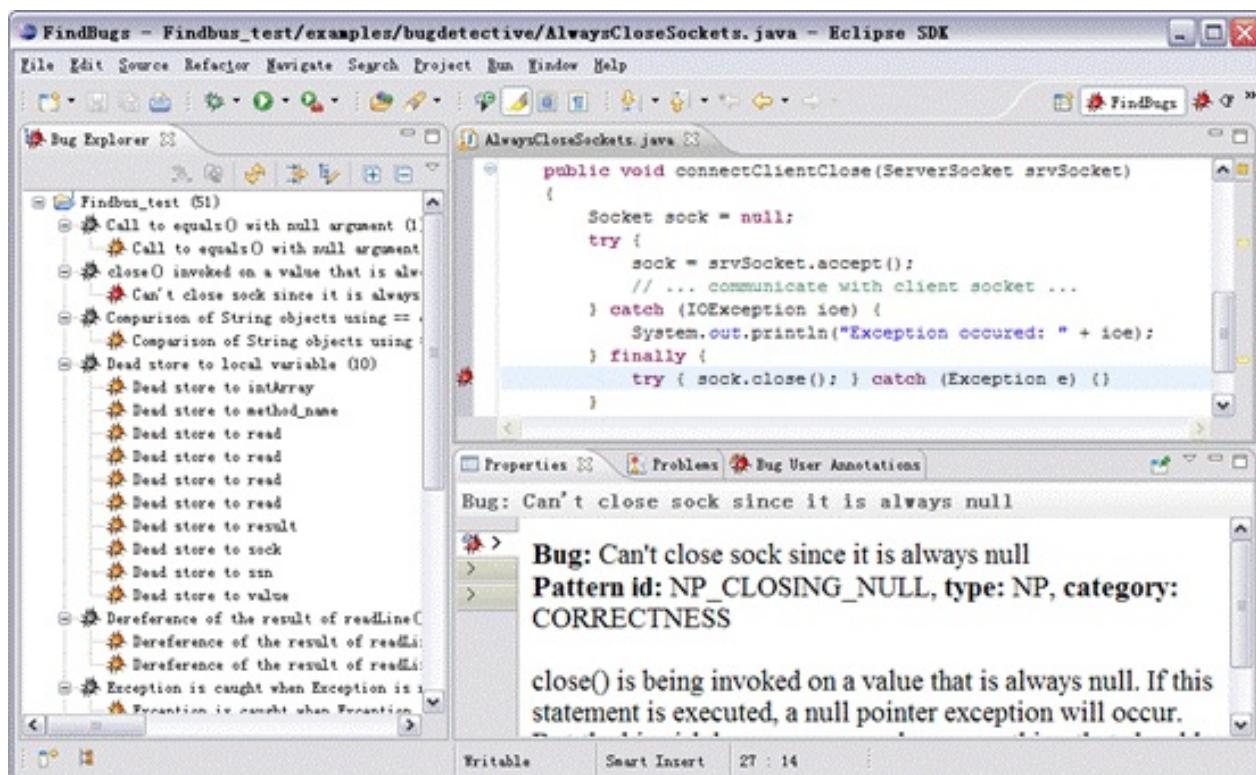
此外，Checkstyle 支持用户根据需求自定义代码检查规范，在下图 2 中的配置面板中，用户可以在已有检查规范如命名约定，Javadoc，块，类设计等方面的基础上添加或删除自定义检查规范。



# FindBugs

FindBugs 是由马里兰大学提供的一款开源 Java 静态代码分析工具。FindBugs 通过检查类文件或 JAR 文件，将字节码与一组缺陷模式进行对比从而发现代码缺陷，完成静态代码分析。FindBugs 既提供可视化 UI 界面，同时也可以作为 Eclipse 插件使用。文本将主要使用将 FindBugs 作为 Eclipse 插件。在安装成功后会在 eclipse 中增加 FindBugs perspective，用户可以对指定 Java 类或 JAR 文件运行 FindBugs，此时 FindBugs 会遍历指定文件，进行静态代码分析，并将代码分析结果显示在 FindBugs perspective 的 bugs explorer 中，如下图 3 所示：

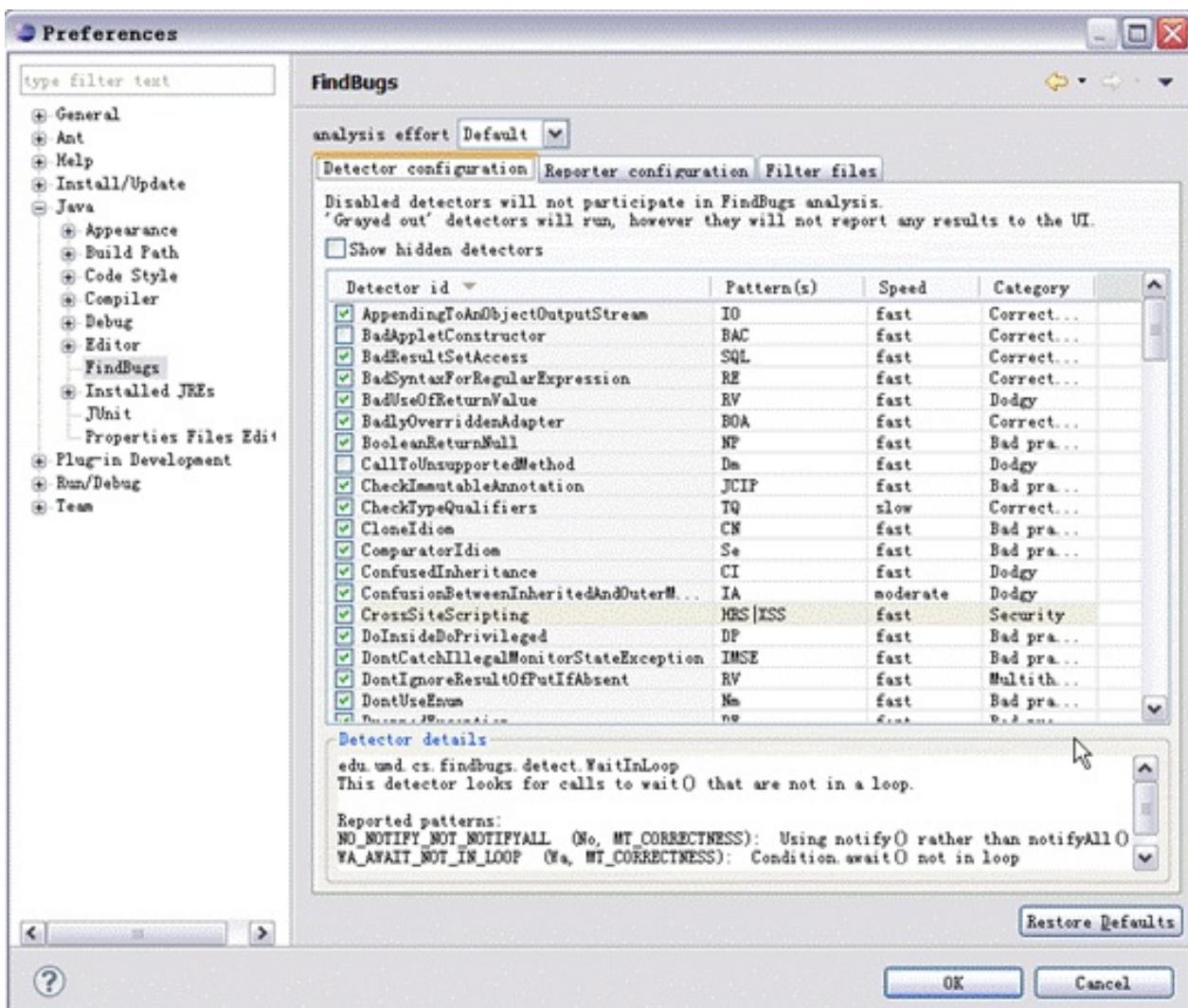
图 3. 使用 FindBugs 进行静态代码分析



图中 Bug Explorer 中的灰色图标处为 Bug 类型，每种分类下红色图标表示 bug 较为严重，黄色的图标表示 bug 为警告程度。Properties 列出了 bug 的描述信息及修改方案。

此外，FindBugs 还为用户提供定制 Bug Pattern 的功能。用户可以根据需求自定义 FindBugs 的代码检查条件，如下图 4 所示：

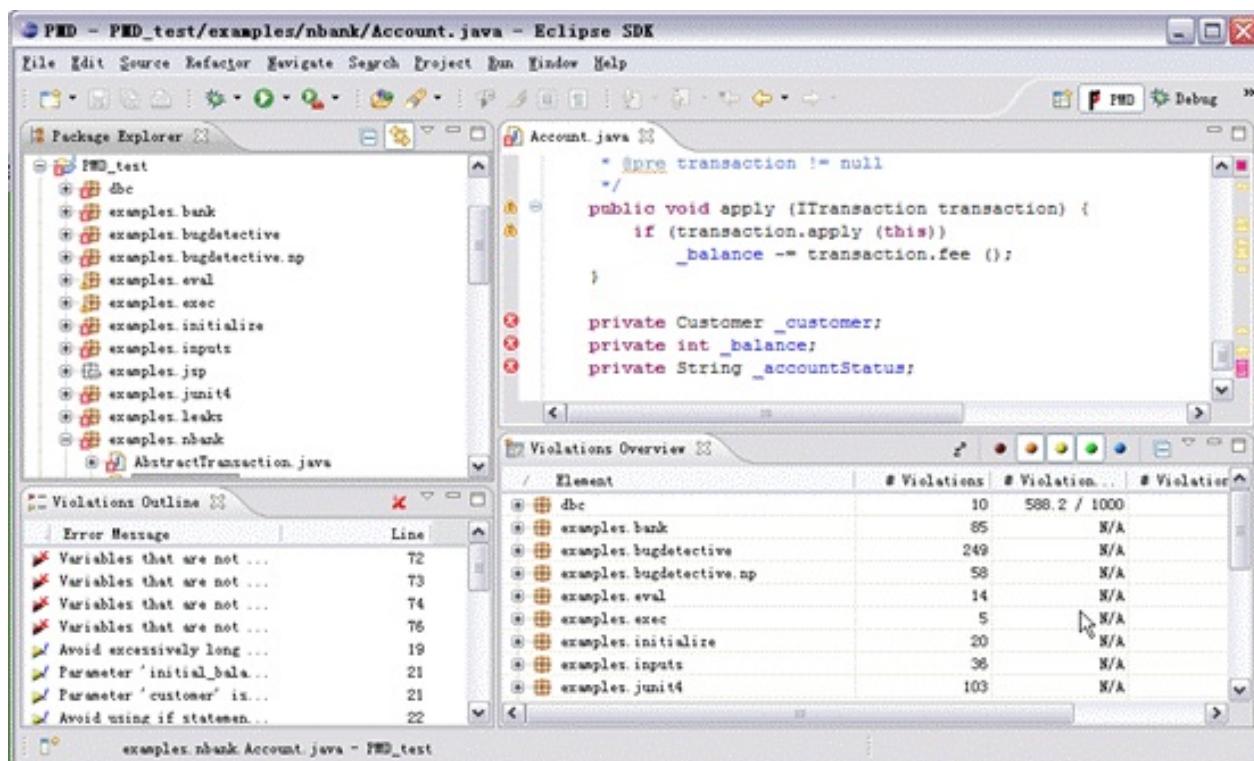
**图 4. 使用 FindBugs 添加自定义代码检查规范**



## PMD

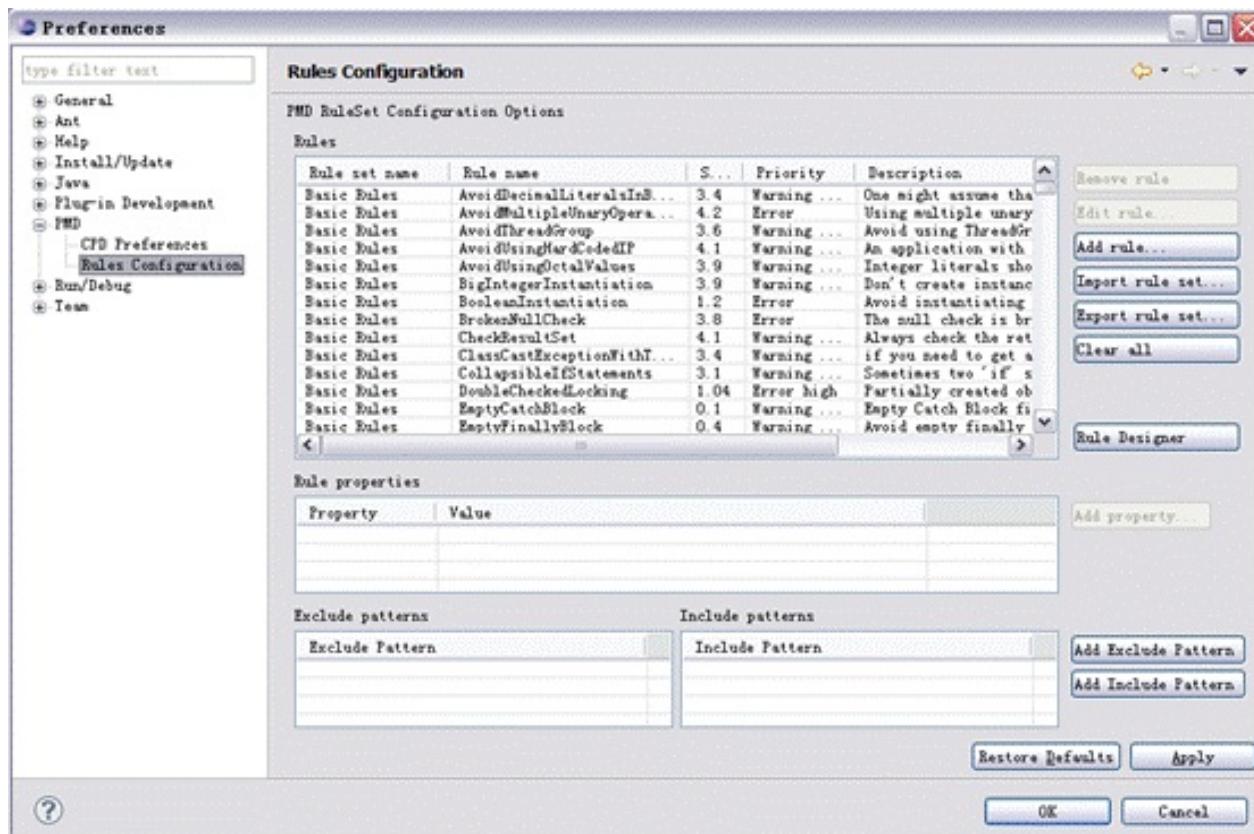
PMD 是由 DARPA 在 SourceForge 上发布的开源 Java 代码静态分析工具。PMD 通过其内置的编码规则对 Java 代码进行静态检查，主要包括对潜在的 bug，未使用的代码，重复的代码，循环体创建新对象等问题的检验。PMD 提供了和多种 Java IDE 的集成，例如 Eclipse，IDEA，NetBean 等。本文主要使用 PMD 以插件方式与 Eclipse 集成。如下图 5 所示：在 Violations Overview 视图中，按照代码缺陷严重性集中显示了 PMD 静态代码分析的结果。

图 5. 使用 PMD 进行静态代码分析



PMD 同样也支持开发人员对代码检查规范进行自定义配置。开发人员可以在下图 6 中的面板中添加、删除、导入、导出代码检查规范。

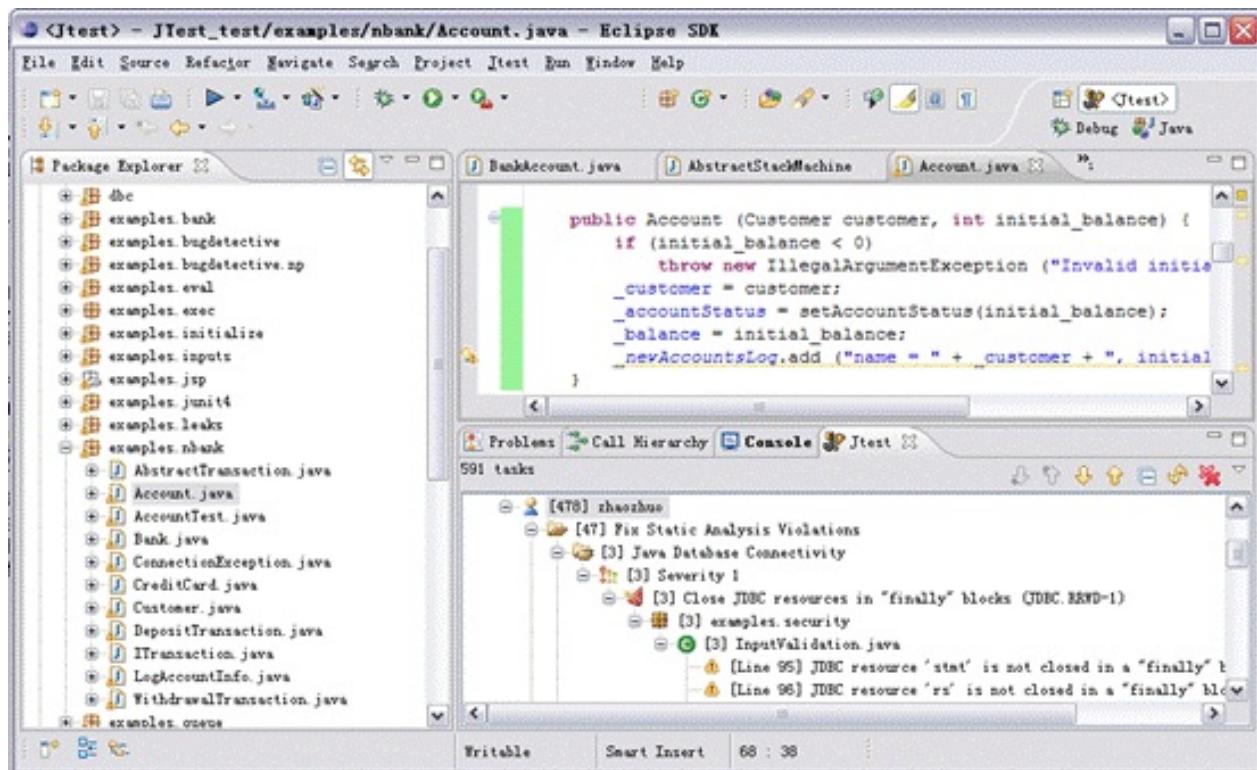
图 6. 使用 PMD 添加自定义代码检查规范



## Jtest

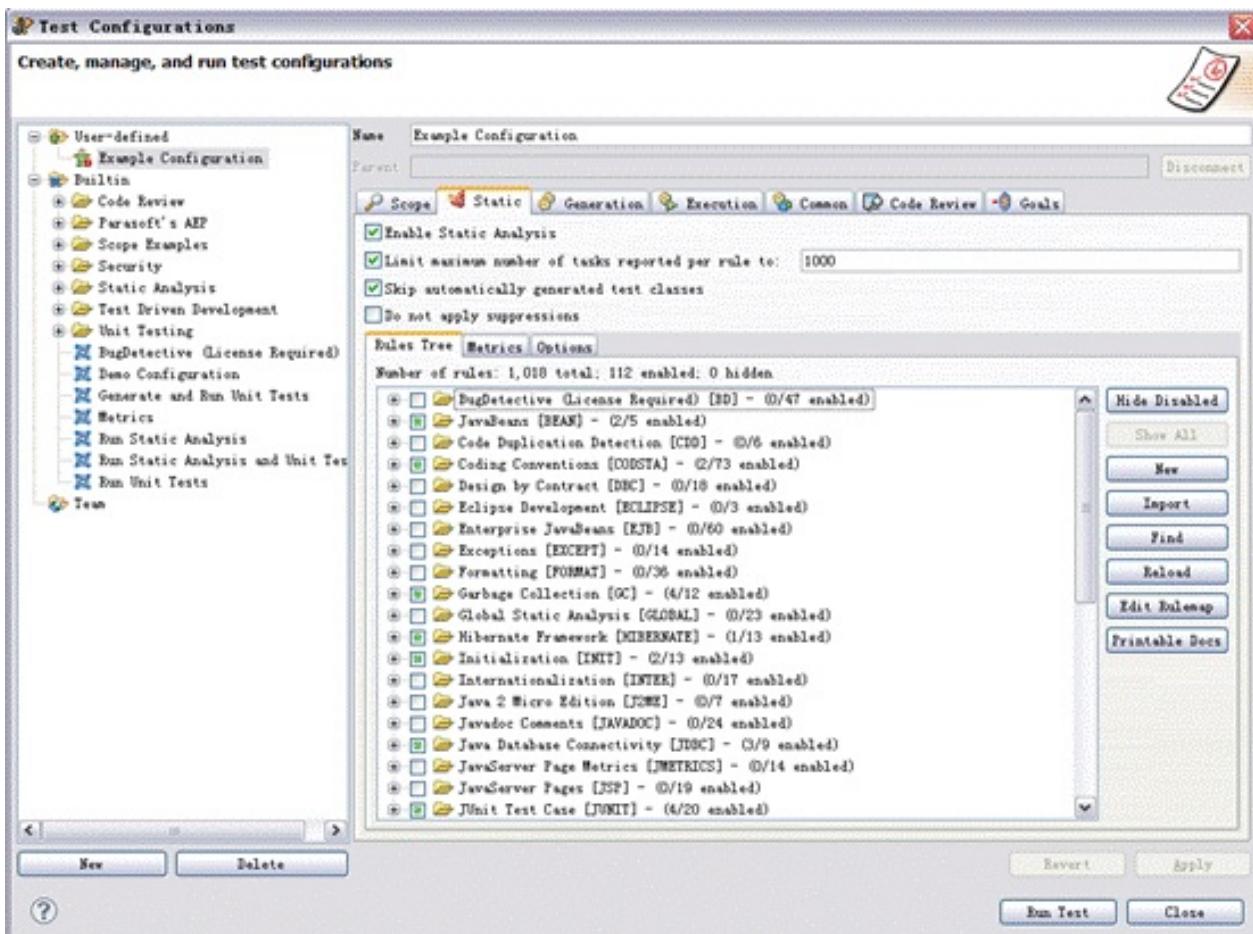
Jtest 是 Parasoft 公司推出的一款针对 Java 语言的自动化代码优化和测试工具，Jtest 的静态代码分析功能能够按照其内置的超过 800 条的 Java 编码规范自动检查并纠正这些隐蔽且难以修复的编码错误。同时，还支持用户自定义编码规则，帮助用户预防一些特殊用法的错误。Jtest 提供了基于 Eclipse 的插件安装。Jtest 支持开发人员对 Java 代码进行编码规范检查，并在 Jtask 窗口中集中显示检查结果，如下图 7 所示：

图 7. 使用 Jtest 进行静态代码分析



同时，Jtest 还提供了对用户定制代码检查配置甚至自定义编码规则的支持，这一功能使得开发人员可以基于不同场景定制所需要的编码规范，如图 8 所示：

图 8. 使用 Jtest 添加自定义代码检查规范



## Java 静态分析工具对比

本章节将从以下几个方面对上述 Java 静态分析工具进行比较：

### 应用技术及分析对象

下表 1 列出了不同工具的分析对象及应用技术对比：

**表 1. 不同工具的分析对象及应用技术对比**

| Java静态分析工具 | 分析对象     | 应用技术         |
|------------|----------|--------------|
| Checkstyle | Java 源文件 | 缺陷模式匹配       |
| FindBugs   | 字节码      | 缺陷模式匹配；数据流分析 |
| PMD        | Java 源代码 | 缺陷模式匹配       |
| Jtest      | Java源代码  | 缺陷模式匹配；数据流分析 |

### 内置编程规范

### **Checkstyle:**

- Javadoc 注释：检查类及方法的 Javadoc 注释
- 命名约定：检查命名是否符合命名规范
- 标题：检查文件是否以某些行开头
- Import 语句：检查 Import 语句是否符合定义规范
- 代码块大小，即检查类、方法等代码块的行数
- 空白：检查空白符，如 tab, 回车符等
- 修饰符：修饰符号的检查，如修饰符的定义顺序
- 块：检查是否有空块或无效块
- 代码问题：检查重复代码，条件判断，魔数等问题
- 类设计：检查类的定义是否符合规范，如构造函数的定义等问题

### **FindBugs:**

- Bad practice 坏的实践：常见代码错误，用于静态代码检查时进行缺陷模式匹配
- Correctness 可能导致错误的代码，如空指针引用等
- 国际化相关问题：如错误的字符串转换
- 可能受到的恶意攻击，如访问权限修饰符的定义等
- 多线程的正确性：如多线程编程时常见的同步，线程调度问题。
- 运行时性能问题：如由变量定义，方法调用导致的代码低效问题。

### **PMD:**

- 可能的 Bugs：检查潜在代码错误，如空 try/catch/finally/switch 语句
- 未使用代码（Dead code）：检查未使用的变量，参数，方法
- 复杂的表达式：检查不必要的 if 语句，可被 while 替代的 for 循环
- 重复的代码：检查重复的代码
- 循环体创建新对象：检查在循环体内实例化新对象
- 资源关闭：检查 Connect, Result, Statement 等资源使用之后是否被关闭掉

### **Jtest:**

- 可能的错误：如内存破坏、内存泄露、指针错误、库错误、逻辑错误和算法错误等
- 未使用代码：检查未使用的变量，参数，方法
- 初始化错误：内存分配错误、变量初始化错误、变量定义冲突
- 命名约定：检查命名是否符合命名规范
- Javadoc 注释：检查类及方法的 Javadoc 注释
- 线程和同步：检验多线程编程时常见的同步，线程调度问题
- 国际化问题：
- 垃圾回收：检查变量及 JDBC 资源是否存在内存泄露隐患

## 错误检査能力

为比较上述 Java 静态分析工具的代码缺陷检测能力，本文将使用一段示例代码进行试验，示例代码中将涵盖我们开发中的几类常见错误，如引用操作、对象操作、表达式复杂化、数组使用、未使用变量或代码段、资源回收、方法调用及代码设计几个方面。最后本文将分别记录在默认检查规范设置下，不同工具对该示例代码的分析结果。以下为示例代码 Test.java。其中，代码的注释部分列举了代码中可能存在的缺陷。

### 清单 1. Test.java 示例代码

```
package Test;
import java.io.*;
public class Test {
 /**
 * Write the bytes from input stream to output stream.
 * The input stream and output stream are not closed.
 * @param is
 * @param os
 * @throws IOException
 */
 public boolean copy(InputStream is, OutputStream os) throws IOException {
 int count = 0;
 //缺少空指针判断
 byte[] buffer = new byte[1024];
 while ((count = is.read(buffer)) >= 0) {
 os.write(buffer, 0, count);
 }
 //未关闭I/O流
 return true;
 }
 /**
 *
 * @param a
 * @param b
 * @param ending
 * @return copy the elements from a to b, and stop when meet element ending
 */
 public void copy(String[] a, String[] b, String ending)
 {
 int index;
 String temp = null;
 //空指针错误
 System.out.println(temp.length());
 //未使用变量
 int length=a.length;
 for(index=0; index<a.length; index++)
 {
 //多余的if语句
 if(true)

```

```
{
 //对象比较 应使用equals
 if(temp==ending)
 {
 break;
 }
 //缺少 数组下标越界检查
 b[index]=temp;
}
}
}
/**
 * @param file
 * @return file contents as string; null if file does not exist
 */
public void readFile(File file) {
 InputStream is = null;
 OutputStream os = null;
 try {
 is = new BufferedInputStream(new FileInputStream(file));
 os = new ByteArrayOutputStream();
 //未使用方法返回值
 copy(is,os);
 is.close();
 os.close();
 } catch (IOException e) {
 //可能造成I/O流未关闭
 e.printStackTrace();
 }
 finally
 {
 //空的try/catch/finally块
 }
}
```

通过以上测试代码，我们对已有 Java 静态代码分析工具的检验结果做了如下比较，如下表 2 所示。

表 2. Java 静态代码分析工具对比

| 代码缺陷分类    | 示例                     | Checkstyle | FindBugs | PMD | Jtest |
|-----------|------------------------|------------|----------|-----|-------|
| 引用操作      | 空指针引用                  | √          | √        | √   | √     |
| 对象操作      | 对象比较（使用 == 而不是 equals） |            | √        | √   | √     |
| 表达式复杂化    | 多余的 if 语句              |            |          | √   |       |
| 数组使用      | 数组下标越界                 |            |          |     | √     |
| 未使用变量或代码段 | 未使用变量                  |            | √        | √   | √     |
| 资源回收      | I/O 未关闭                |            | √        |     | √     |
| 方法调用      | 未使用方法返回值               |            | √        |     |       |
| 代码设计      | 空的 try/catch/finally 块 |            |          |     | √     |

由表中可以看出几种工具对于代码检查各有侧重。其中，Checkstyle 更偏重于代码编写格式，及是否符合编码规范的检验，对代码 bug 的发现功能较弱；而 FindBugs，PMD，Jtest 着重于发现代码缺陷。在对代码缺陷检查中，这三种工具在针对的代码缺陷类别也各有不同，且类别之间有重叠。

本文分别从功能、特性和内置编程规范等方面详细介绍了包括 Checkstyle，FindBugs，PMD，Jtest 在内的四种主流 Java 静态代码分析工具，并通过一段 Java 代码示例对这四种工具的代码分析能力进行比较。由于这四种工具内置编程规范各有不同，因此它们对不同种类的代码问题的发现能力也有所不同。其中 Checkstyle 更加偏重于代码编写格式检查，而 FindBugs，PMD，Jtest 着重于发现代码缺陷。最后，希望本文能够帮助 Java 软件开发和测试人员进一步了解以上四种主流 Java 静态分析工具，并帮助他们根据需求选择合适的工具。

看完了直接我就给我的Android Studio装上了，此乃神器

补充评论内容：

这个文档写得不错，最近正在使用Jtest这个工具，正好可以参考一下，呵呵。不过我用这个实例代码在我的Jtest中跑了一下，找到了您说不能找到的两个bug。

- 多余的 if 语句. 在 Jtest 中的规则是 Avoid unnecessary 'if' statements [UC.UCIF-3]

你可以看我的测试截图。

The screenshot shows the Jtest interface with the following details:

- Code Snippet:**

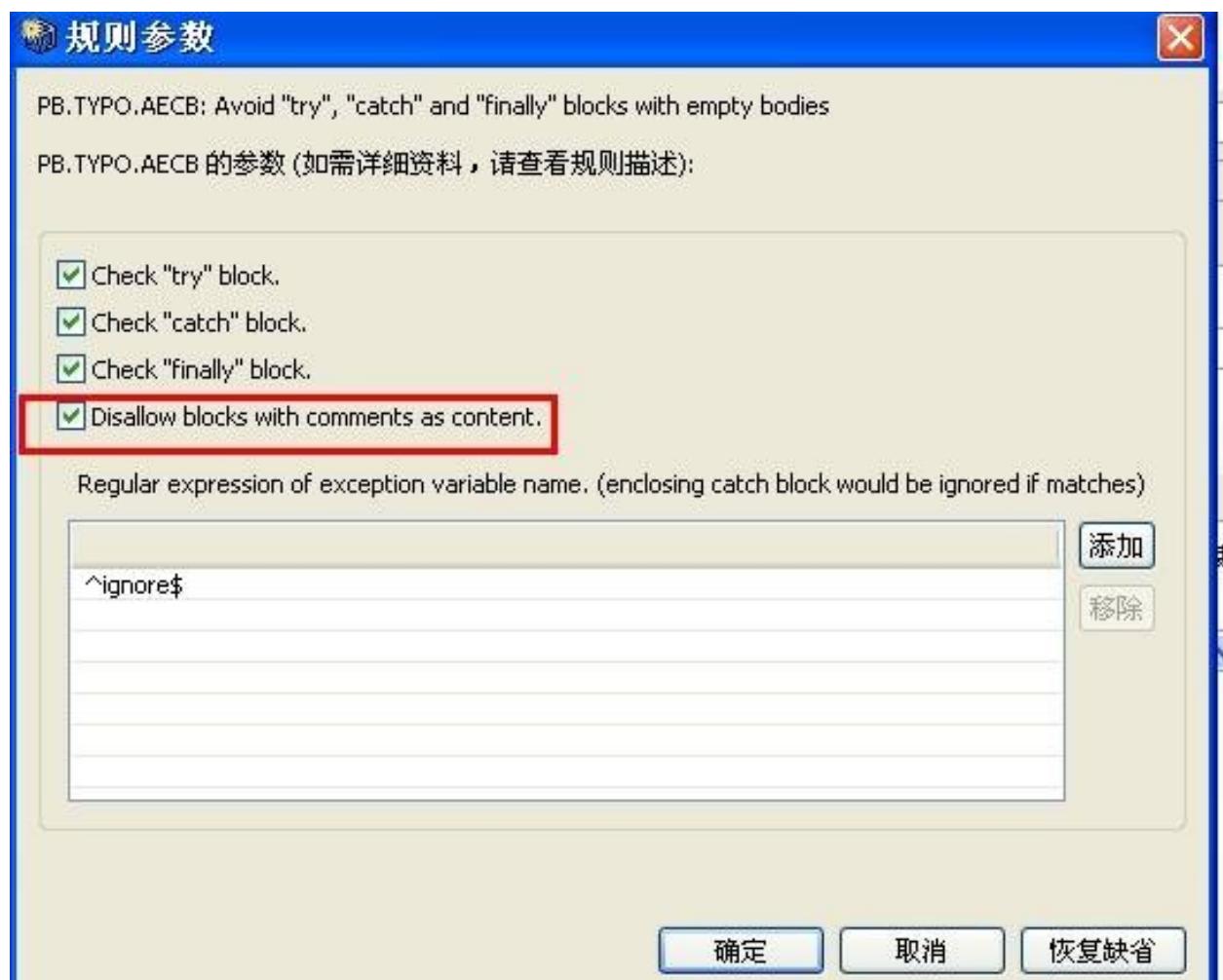
```

37
38
39 //多余的if语句
40
41 //对象比较 应使用 equals
42

```
- Analysis Results:**
  - Quality Tasks: 2 tasks, 0 static analysis.
  - Test Results: 2 test results, 0 code review.
  - Tree View:
    - [2] > 修复静态分析的违规
      - [2] > 未使用的代码
        - [1] > Avoid local variables that are never read (UC.AURV-1)
        - [1] > Avoid unnecessary 'if' statements. (UC.UCIF-3) **(Selected)**
          - [1] > examples.eval
          - [1] > Test.java
            - [行 39] Unconditionally true "if" statement

- 空的 try/catch/finally 块.

在 Jtest 中的规则是: Avoid unnecessary 'if' statements [UC.UCIF-3] 不过这个部分在刚开始没有找到，后来发现Jtest有条规则是可以参数化的，就是默认Jtest会把注释认为是内容。当我把下图的选项勾上后，就能找到这个bug。看来Jtest还是很人性化的嘛，很多都考虑的比我们要仔细。呵呵。



测试结果参见下图



未使用方法返回值这个好像暂时没有在Jtest中找到规则，不过我把所有的规则都启用了后，这个实例代码报告了几百个问题。除了你埋入的bug还有一些有用的结果，不过还需要时间去筛选哪部分更重要。

总的来说，我使用Jtest感受还是认为它很专业的，以前我也用过FindBugs，这个工具就比较简陋啦。呵呵。我们的项目现在正在用Jtest的单元测试功能，以前还没做过，现在为了过认证要做这部分。不知道你有没有这方面的经验。

- 个人觉得Jtest软件不错。
  - Jtest软件介绍：<http://www.innovatedigital.com/JavaTuning/Jtest.shtml>
  - Jtest软件详细中文资料参考：<http://www.innovatedigital.com/taxonomy/term/74>

# 开发Tips

# Android开发的那些坑和小技巧

来源:[cnblogs](#)

## 1、 android:clipToPadding

意思是控件的绘制区域是否在padding里面。默认为true。如果你设置了此属性值为false，就能实现一个在布局上事半功倍的效果。先看一个效果图。



上图中的ListView顶部默认有一个间距，向上滑动后，间距消失，如下图所示。

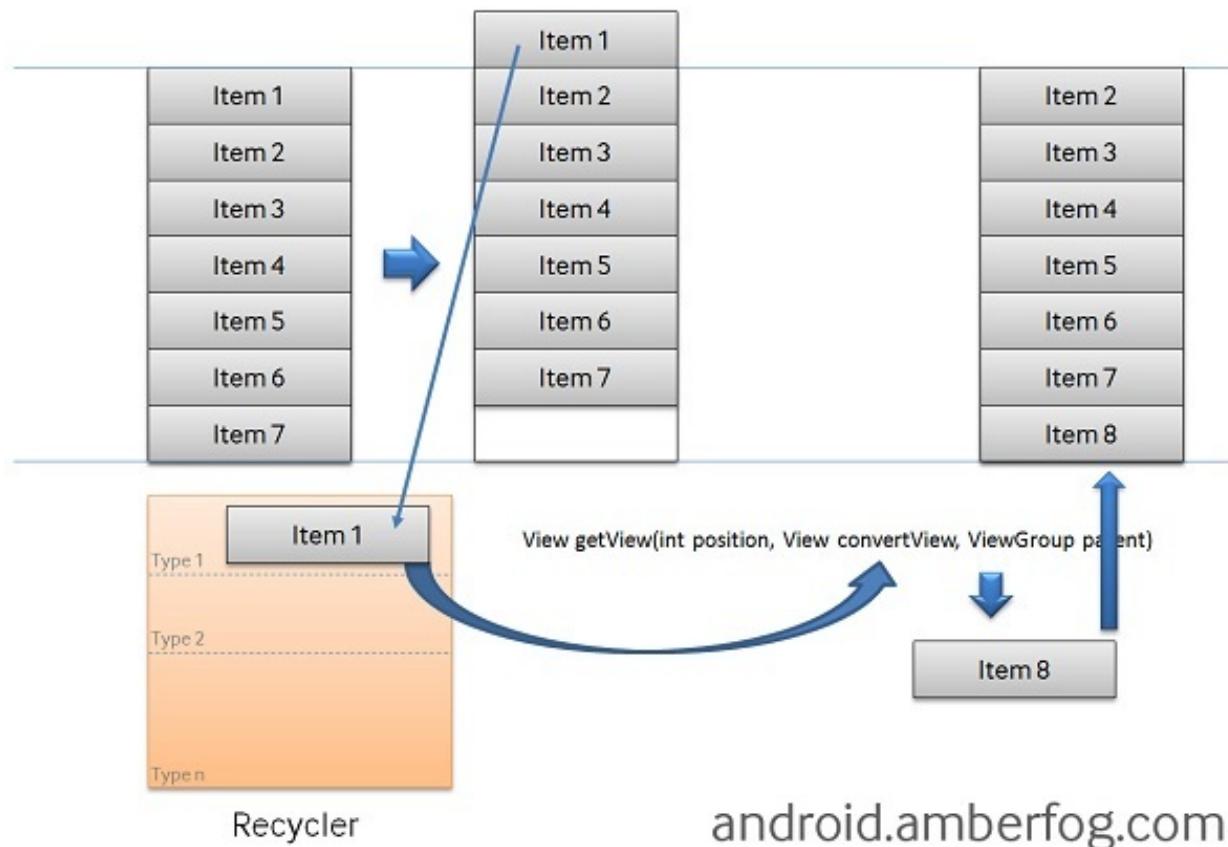


如果使用margin或padding，都不能实现这个效果。加一个headerView又显得大材小用，而且过于麻烦。此处的clipToPadding配合paddingTop效果就刚刚好。

同样，还有另外一个属性也很神奇：`android:clipChildren`，具体请参考：[【Android】神奇的android:clipChildren属性](#)

## 2、match\_parent和wrap\_content

按理说这两个属性一目了然，一个是填充布局空间适应父控件，一个是适应自身内容大小。但如果在列表如ListView中，用错了问题就大了。ListView中的getView方法需要计算列表条目，那就必然需要确定ListView的高度，onMeasure才能做测量。如果指定了`wrap_content`，就等于告诉系统，如果我有一万个条目，你都帮我计算显示出来，然后系统按照你的要求就new了一万个对象出来。那你不悲剧了？先看一个图。



假设现在ListView有8条数据，match\_parent需要new出7个对象，而wrap\_content则需要8个。这里涉及到View的重用，就不多探讨了。所以这两个属性的设置将决定getView的调用次数。

由此再延伸出另外一个问题：**getView被多次调用**。

什么叫多次调用？比如position=0它可能调用了几次。看似很诡异吧。GridView和ListView都有可能出现，说不定这个祸首就是wrap\_content。说到底是View的布局出现了问题。如果嵌套的View过于复杂，解决方案可以是通过代码测量列表所需要的宽度，或者在getView中使用一个小技巧： `parent.getChildCount == position`

```

@Override
public View getView(int position, View convertView, ViewGroup parent) {
 if (parent.getChildCount() == position) {
 // does things here
 }

 return convertView;
}

```

### 3、IllegalArgumentException: pointerIndex out of range

出现这个Bug的场景还是很无语的。一开始我用ViewPager + PhotoView(一个开源控件)显示图片，在多点触控放大缩小时就出现了这个问题。一开始我怀疑是PhotoView的bug，找了半天无果。要命的是不知如何try，老是crash。后来才知道是android遗留下来的bug，源码里没对pointer index做检查。改源码重新编译不太可能吧。明知有exception，又不能从根本上解决，如果不让它crash，那就只能try-catch了。解决办法是：自定义一个ViewPager并继承ViewPager。请看以下代码：

```
/*
 * 自定义封装android.support.v4.view.ViewPager，重写onInterceptTouchEvent事件，捕获系统级别
 */
public class CustomViewPager extends ViewPager {

 public CustomViewPager(Context context) {
 super(context, null);
 }

 public CustomViewPager(Context context, AttributeSet attrs) {
 super(context, attrs);
 }

 @Override
 public boolean onInterceptTouchEvent(MotionEvent ev) {
 try {
 return super.onInterceptTouchEvent(ev);
 } catch (IllegalArgumentException e) {
 LogUtil.e(e);
 } catch (ArrayIndexOutOfBoundsException e) {
 LogUtil.e(e);
 }
 return false;
 }
}
```

把用到ViewPager的布局文件，替换成CustomViewPager就OK了。

## 4、ListView中item点击事件无响应

listView的Item点击事件突然无响应，问题一般是在listView中加入了button、checkbox等控件后出现的。这个问题是聚焦冲突造成的。在android里面，点击屏幕之后，点击事件会根据你的布局来进行分配的，当你的listView里面增加了button之后，点击事件第一优先分配给你listView里面的button。所以你的点击Item就失效了，这个时候你就要根据你的需求，是给你的item的最外层layout设置点击事件，还是给你的某个布局元素添加点击事件了。

解决办法：在ListView的根控件中设置（若根控件是LinearLayout，则在LinearLayout中加入以下属性设置）`descendantFocusability` 属性。

```
android:descendantFocusability="blocksDescendants"
```

官方文档也是这样说明。

| Constant                       | Value | Description                                                           |
|--------------------------------|-------|-----------------------------------------------------------------------|
| <code>beforeDescendants</code> | 0     | The ViewGroup will get focus before any of its descendants.           |
| <code>afterDescendants</code>  | 1     | The ViewGroup will get focus only if none of its descendants want it. |
| <code>blocksDescendants</code> | 2     | The ViewGroup will block its descendants from receiving focus.        |

## 5、`getSupportFragmentManager()`和`getChildFragmentManager()`

有一个需求，Fragment需要嵌套3个Fragment。基本上可以想到用ViewPager实现。开始代码是这样写的：

```
mViewPager.setAdapter(new CustomizePagerAdapter(
 getActivity().getSupportFragmentManager()
 , subFragmentList));
```

导致的问题是嵌套的Fragment有时会莫名其妙不显示。开始根本不知道问题出现在哪，当你不知道问题的原因时，去解决这个问题显然比较麻烦。经过一次又一次的寻寻觅觅，终于在stackoverflow上看到了同样的提问。说是用`getChildFragmentManager()`就可以了。真是这么神奇！

```
mViewPager.setAdapter(new CustomizePagerAdapter(
 ``getChildFragmentManager(), subFragmentList));
```

让我们看一下这两个有什么区别。首先是`getSupportFragmentManager`（或者`getFragmentManager`）的说明：

Return the FragmentManager for interacting with fragments associated with this fragment's activity.

然后是`getChildFragmentManager`：

Return a private FragmentManager for placing and managing Fragments inside of this Fragment. Basically, the difference is that Fragment's now have their own internal FragmentManager that can handle Fragments. The child FragmentManager is the one that handles Fragments contained within only the Fragment that it was added to. The other FragmentManager is contained within the entire Activity.

已经说得比较明白了。

## 6、ScrollView嵌套ListView

这样的设计是不是很奇怪？两个同样会滚动的View居然放到了一起，而且还是嵌套的关系。曾经有一个这样的需求：界面一共有4个区域部分，分别是公司基本信息（logo、名称、法人、地址）、公司简介、公司荣誉、公司口碑列表。每部分内容都需要根据内容自适应高度，不能写死。鄙人首先想到的也是外部用一个ScrollView包围起来。然后把这4部分分别用4个自定义控件封装起来。基本信息和公司简介比较简单，荣誉需要用到RecyclerView和TextView的组合，RecyclerView（当然，用GridView也可以，3列多行的显示）存放荣誉图片，TextView显示荣誉名称。最后一部分口碑列表当然是ListView了。这时候，问题就出来了。需要解决ListView放到ScrollView中的滑动问题和RecyclerView的显示问题（如果RecyclerView的高度没法计算，你是看不到内容的）。

当然，网上已经有类似的提问和解决方案了。

给一个网址：

### 四种方案解决ScrollView嵌套ListView问题

ListView的情况还比较好解决，优雅的做法无非写一个类继承ListView，然后重写onMeasure方法。

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
 int expandSpec = MeasureSpec.makeMeasureSpec(Integer.MAX_VALUE >> 2, MeasureSpec.AT_MOST);
 super.onMeasure(widthMeasureSpec, expandSpec);
}
```

ListView可以重写onMeasure解决，RecyclerView重写这个方法是行不通的。

说到底其实计算高度嘛。有两种方式，一种是动态计算RecyclerView，然后设置setLayoutParams；另外一种跟ListView的解决方式类似，定义一个类继承LinearLayoutManager或GridLayoutManager（注意：可不是继承RecyclerView），重写onMeasure方法（此方法比较麻烦，此处不表，下次写一篇文章再作介绍）。

动态计算高度如下：

```
int heightPx = DensityUtil.dip2px(getActivity(), (imageHeight + imageRowHeight) * lines);
MarginLayoutParams mParams = new MarginLayoutParams(LayoutParams.MATCH_PARENT, heightPx);
mParams.setMargins(0, 0, 0, 0);
LinearLayout.LayoutParams lParams = new LinearLayout.LayoutParams(mParams);
honorImageRecycler_view.setLayoutParams(lParams);
```

思路是这样的：服务端返回荣誉图片后，由于是3列显示的方式，只需要计算需要显示几行，然后给定行间距和图片的高度，再设置setLayoutParams就行了。

```
int lines = (int) Math.ceil(totalImages / 3d);
```

至此，这个奇怪的需求得到了解决。

可是在滑动的时候，感觉出现卡顿的现象。聪明的你肯定想到是滑动冲突了。应该是ScrollView的滑动干扰到了ListView的滑动。怎么办呢？能不能禁掉ScrollView的滑动？

百度一下，你肯定能搜索到答案的。先上代码：

```
/*
 * @author Leo
 *
 * Created in 2015-9-12
 * 拦截ScrollView滑动事件
 */
public class CustomScrollView extends ScrollView {

 private int downY;
 private int touchSlop;

 public CustomScrollView(Context context) {
 this(context, null);
 }

 public CustomScrollView(Context context, AttributeSet attrs) {
 this(context, attrs, 0);
 }

 public CustomScrollView(Context context, AttributeSet attrs, int defStyleAttr) {
 super(context, attrs, defStyleAttr);
 touchSlop = ViewConfiguration.get(context).getScaledTouchSlop();
 }

 @Override
 public boolean onInterceptTouchEvent(MotionEvent e) {
 int action = e.getAction();
 switch (action) {
 case MotionEvent.ACTION_DOWN:
 downY = (int) e.getRawY();
 break;
 case MotionEvent.ACTION_MOVE:
 int moveY = (int) e.getRawY();
 if (Math.abs(moveY - downY) > touchSlop) {
 return true;
 }
 }
 return super.onInterceptTouchEvent(e);
 }
}
```

只要理解了getScaledTouchSlop()这个方法就好办了。这个方法的注释是：Distance in pixels a touch can wander before we think the user is scrolling。说这是一个距离，表示滑动的时候，手的移动要大于这个距离才开始移动控件，如果小于此距离就不触发移动。

看似很完美了。

但是还有另外一个问题：我每次加载这个界面花的时间太长了，每次由其它界面启动这个界面时，都要卡上1~2秒，而且因手机性能时间不等。并不是由于网络请求，取数据由子线程做，跟UI线程毫无关系。这样的体验自己看了都很不爽。

几天过去了，还是那样。马上要给老板演示了。这样的体验要被骂十次呀。

难道跟ScrollView的嵌套有关？

好吧，那我重构代码。不用ScrollView了。直接用一个ListView，然后add一个headerView存放其它内容。因为控件封装得还算好，没改多少布局就OK了，一运行，流畅顺滑，一切迎刃而解！

本来就是这么简单的问题，为什么非得用ScrollView嵌套呢？

stackoverflow早就告诉你了，不要这样嵌套！不要这样嵌套！不要这样嵌套！重要的事情说三遍。

[ListView inside ScrollView is not scrolling on Android](#)

当然，从android 5.0 Lollipop开始提供了一种新的API支持嵌入滑动，此时，让像这样的需求也能很好实现。

此处给一个网址，大家有兴趣自行了解，此处不再讨论。

[Android NestedScrolling 实战](#)

## 7、EmojiconTextView的setText(null)

这是开源表情库com.rockerhieu.emojicon中的TextView加强版。相信很多人用到过这个开源工具包。TextView用setText(null)完全没问题。但EmojiconTextView setText(null)后就悲剧了，直接crash，显示的是null pointer。开始我怀疑时这个view没初始化，但并不是。那就调试一下呗。

```
@Override
public void setText(CharSequence text, BufferType type) {
 SpannableStringBuilder builder = new SpannableStringBuilder(text);
 EmojiconHandler.addEmojis(getApplicationContext(), builder, mEmojiconSize);
 super.setText(builder, type);
}
```

EmojiconTextView中的setText看来没什么问题。点SpannableStringBuilder进去看看，源码原来是这样的：

```
/**
 * Create a new SpannableStringBuilder containing a copy of the
 * specified text, including its spans if any.
 */
public SpannableStringBuilder(CharSequence text) {
 this(text, 0, text.length());
}
`
```

好吧。问题已经找到了，`text.length()`，不空指针才怪。

```
text = text == null ? "" : text;
SpannableStringBuilder builder = new SpannableStringBuilder(text);
```

加一行判断就行了。

8、`cursor.close()` 一般来说，`database`的开和关不太会忘记，但游标的使用可能并不会引起太多重视，尤其是游标的随意使用。比如用`ContentResolver`结合`Cursor`查询SD卡中图片，很容易写出以下的代码：

```
Cursor cursor = contentResolver.query(uri, null, MediaStore.Images.Media.MIME_TYPE +
 + MediaStore.Images.Media.MIME_TYPE + "=?", new String[] { "image/*" });
while (cursor.moveToNext()) {
 // TODO
}
```

`cursor`都不做非空判断，而且往往在关闭游标的时候不注意有可能异常抛出。

以前在项目中，经常出现由于游标没及时关闭或关闭出异常没处理好导致其它的问题产生，而且问题看起来非常的诡异，不好解决。后来，我把整个项目中有关游标的使用重构一遍，后来就再没发生过类似的问题。

原则很简单，所有`Cursor`的声明为：

```
Cursor cursor = null;
```

且放在try-catch外面；需要用到`cursor`，先做非空判断。然后在方法的最后用一个工具类处理游标的关闭。

```
/*
 * @author Leo
 *
 * Created in 2015-9-15
 */
public class IOUtil {

 private IOUtil() {
 }

 public static void closeQuietly(Closeable closeable) {
 if (closeable != null) {
 try {
 closeable.close();
 } catch (Throwable e) {
 }
 }
 }

 public static void closeQuietly(Cursor cursor) {
 if (cursor != null) {
 try {
 cursor.close();
 } catch (Throwable e) {
 }
 }
 }
}
```

我想，这样就没必要在每个地方都try-catch-finally了。

## 9、java.lang.String cannot be converted to JSONObject

解析服务端返回的JSON字符串时，居然抛出了这个异常。调试没发现任何问题，看起来是正常的JSON格式。后来发现居然是JSON串多了BOM（Byte Order Mark）。服务端的代码由PHP实现，有时开发为了修改方便，直接用windows记事本打开保存，引入了人眼看不到的问题。其实就是多了"\ufeff"这个玩意，客户端代码过滤一下就行了。

```
// in case: Value of type java.lang.String cannot be converted to JSONObject
// Remove the BOM header
if (jsonStr != null) {
 jsonStr = jsonStr.trim();
 if (jsonStr.startsWith("\ufeff")) {
 jsonStr = jsonStr.substring(1);
 }
}
```

## 10、Shape round rect too large to be rendered into a texture

圆形矩形太大？

一开始我发现一个activity中的scrollView滑动一顿一顿的，而实际上没有嵌套任何的列表控件如ListView、GridView，包含的无非是一些TextView、ImageView等。看了下Eclipse中log输出，发现出现了这个warn级别的提示。难道是我在外层嵌套了这个圆形矩形？我在很多地方都用了呀，为何就这个界面出现问题了？

后来才发现，这个圆形矩形包含的内容太多了，已经超出了手机的高度，而且可以滑好几页。

StackOverFlow上有人说：The easiest solution is to get rid of the rounded corners. If you remove the rounded corners and use a simple rectangle, the hardware renderer will no longer create a single large texture for the background layer, and won't run into the texture size limit any more.

也有建议：`to draw onto the canvas`。

具体链接：[How Do Solve Shape round rect too large to be rendered into a texture](#)

我试了下自定义控件LinearLayout，通过canvas进行draw，没能解决。去掉radius属性确实可行，可我想保留怎么办？

还有一个解决办法，通过在androidManifest.xml中禁用硬件加速，为了控制粒度，我只在此activity中禁用此功能。

```
<activity android:hardwareAccelerated="false" />
```

先想到这么多，以后再补充。

## 参考：

- android:clipToPadding和android:clipChildren
- HowTo: ListView, Adapter, getView and different list items' layouts in one ListView
- android ListView 在初始化时多次调用getView()原因分析
- java.lang.IllegalArgumentException: pointerIndex out of range Exception - dispatchTouchEvent
- What is difference between getSupportFragmentManager() and getChildFragmentManager()?

# ListView和SwipeRefreshLayout冲突解决

来源：

[关于SwipeRefreshLayout和ListView滑动冲突问题解决办法](#)

[解决listview与SwipeRefreshLayout滑动冲突问题](#)

[滑动ViewPager引起swiperefreshlayout刷新的冲突](#)

## 1. 关于SwipeRefreshLayout和ListView滑动冲突问题解决办法

自从Google推出 SwipeRefreshLayout 后相信很多人都开始使用它来实现listView的下拉刷新了，但是在使用的时候，有一个很有趣的现象，当 SwipeRefreshLayout 只有listview一个子view的时候是没有任何问题的，但如果不是得话就会出现问题了，向上滑动listview一切正常，向下滑动的时候就会出现还没有滑倒listview顶部就触发下拉刷新的动作了，看 SwipeRefreshLayout 源码可以看到在 `onInterceptTouchEvent` 里面有这样的一段代码

```
if (!isEnabled() || mReturningToStart || canChildScrollUp() || mRefreshing) {
 // Fail fast if we're not in a state where a swipe is possible
 return false;
}
```

其中有个 `canChildScrollUp` 方法，在往下看

```
public boolean canChildScrollUp() {
 if (android.os.Build.VERSION.SDK_INT < 14) {
 if (mTarget instanceof AbsListView) {
 final AbsListView absListView = (AbsListView) mTarget;
 return absListView.getChildCount() > 0
 && (absListView.getFirstVisiblePosition() > 0
 || absListView.getChildAt(0)
 .getTop() < absListView.getPaddingTop());
 } else {
 return ViewCompat.canScrollVertically(mTarget, -1) || mTarget.getScrollY();
 }
 } else {
 return ViewCompat.canScrollVertically(mTarget, -1);
 }
}
```

决定子view 能否滑动就是在这里了，所以我们只有写一个类继承 SwipeRefreshLayout，然后重写该方法即可

```
public class SimpleSwipeRefreshLayout extends SwipeRefreshLayout {

 private View view;
 public SimpleSwipeRefreshLayout(Context context) {
 super(context);
 }

 public SimpleSwipeRefreshLayout(Context context, AttributeSet attrs) {
 super(context, attrs);
 }

 public void setViewGroup(View view) {
 this.view = view;
 }

 @Override
 public boolean canChildScrollUp() {
 if (view != null && view instanceof AbsListView) {
 final AbsListView absListView = (AbsListView) view;
 return absListView.getChildCount() > 0
 && (absListView.getFirstVisiblePosition() > 0
 || absListView.getChildAt(0)
 .getTop() < absListView.getPaddingTop());
 }
 return super.canChildScrollUp();
 }
}
```

## 2. 解决listview与SwipeRefreshLayout滑动冲突问题

```

listView.setOnScrollListener(new OnScrollListener() {

 @Override
 public void onScrollStateChanged(AbsListView view, int scrollState) {
 }

 @Override
 public void onScroll(AbsListView view, int firstVisibleItem,
 int visibleItemCount, int totalItemCount) {
 boolean enable = false;
 if(listView != null && listView.getChildCount() > 0){
 // check if the first item of the list is visible
 boolean firstItemVisible = listView.getFirstVisiblePosition() == 0;
 // check if the top of the first item is visible
 boolean topOfFirstItemVisible = listView.getChildAt(0).getTop() == 0;
 // enabling or disabling the refresh layout
 enable = firstItemVisible && topOfFirstItemVisible;
 }
 swipeRefreshLayout.setEnabled(enable);
 });
}

```

以上方案在ListView没有数据的时候有问题，所以改正如下：

```

mLvAttentions.setOnScrollListener(new AbsListView.OnScrollListener() {

 @Override
 public void onScrollStateChanged(AbsListView view, int scrollState) {
 }

 @Override
 public void onScroll(AbsListView view, int firstVisibleItem,
 int visibleItemCount, int totalItemCount) {

 boolean enable = false;
 if(mLvAttentions != null && mLvAttentions.getChildCount() > 0){
 // check if the first item of the list is visible
 boolean firstItemVisible = mLvAttentions.getFirstVisiblePosition() == 0;
 // check if the top of the first item is visible
 boolean topOfFirstItemVisible = mLvAttentions.getChildAt(0).getTop() == 0;
 // enabling or disabling the refresh layout
 enable = firstItemVisible && topOfFirstItemVisible;
 }else if(mLvAttentions != null){
 enable = true;
 }
 mRefresh.setEnabled(enable);
 });
}

```

### 3. 滑动ViewPager引起SwipeRefreshLayout刷新的冲突

ViewPager 是Android中提供的页面切换的控件， SwipeRefreshLayout 是Android提供的下拉刷新控件，通过 SwipeRefreshLayout 可以很简单的实现下拉刷新的功能，但是如果 SwipeRefreshLayout 的子view中如果包含了ViewPager，会发现滑动ViewPager的时候，很容易引起 SwipeRefreshLayout 的下拉刷新操作。为了解决这个冲突可以这样实现

```
viewPager.setOnTouchListener(new View.OnTouchListener() {
 @Override
 public boolean onTouch(View v, MotionEvent event) {
 switch (event.getAction()) {
 case MotionEvent.ACTION_MOVE:
 swipeRefreshLayout.setEnabled(false);
 break;
 case MotionEvent.ACTION_UP:
 case MotionEvent.ACTION_CANCEL:
 swipeRefreshLayout.setEnabled(true);
 break;
 }
 return false;
 }
});
```

ViewPager，设置 OnTouchListener，里面当ACTION\_MOVE的时候设置 SwipeRefreshLayout 不可用，当 ACTION\_UP 或者 ACTION\_CANCEL 的时候设置 SwipeRefreshLayout 可以，就可以解决这个冲突了

# 神奇的android:clipChildren属性

来源:[cnblogs](#)

## 前言

前几天有在微博上推荐过一个博客，看他文章时发现了这个属性。有些属性不常用，但需要的时候非常有用，于是做了个例子，正好项目用到，与大家分享一下。

## 声明

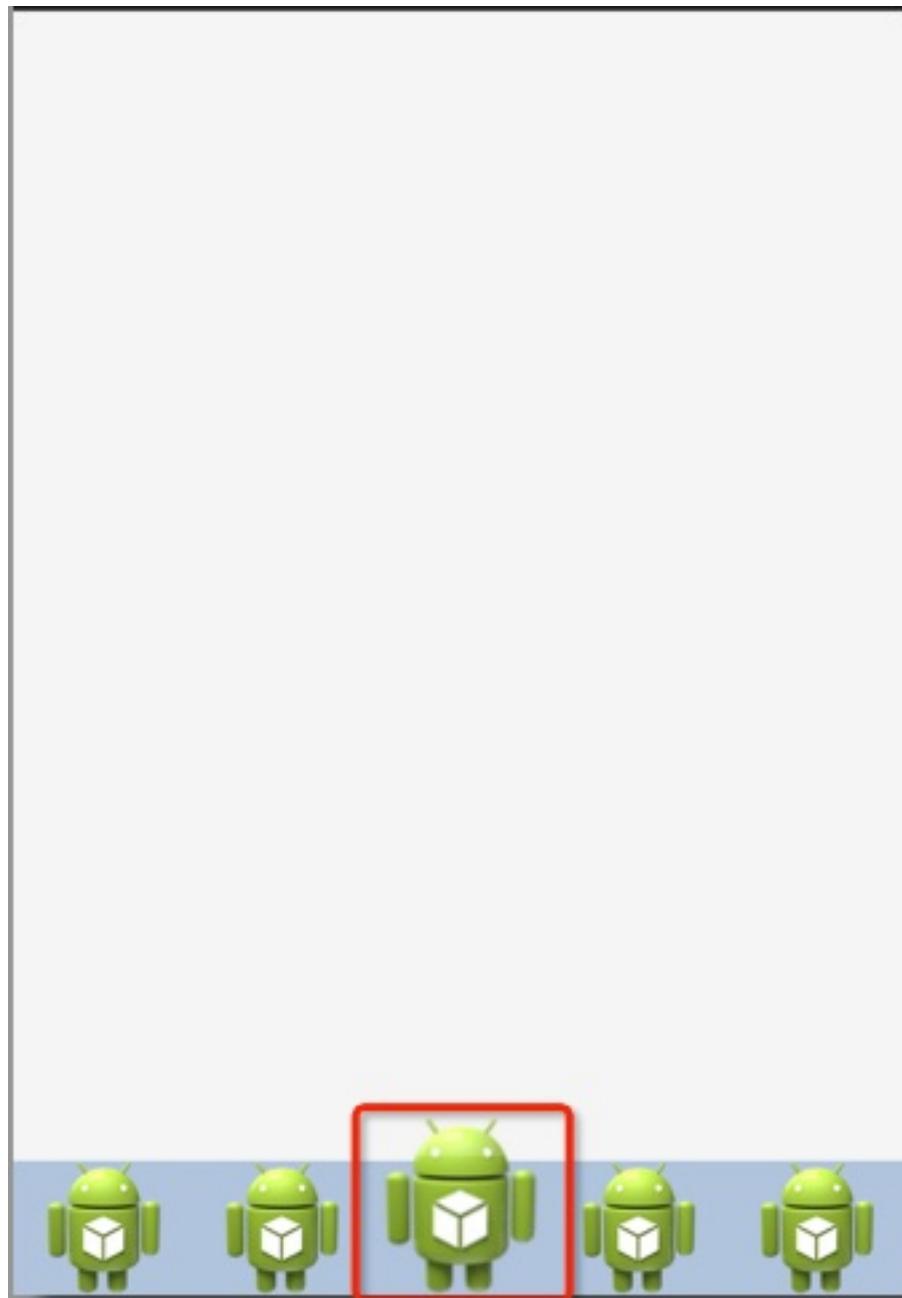
欢迎转载，请注明出处！

博客园：<http://www.cnblogs.com/>

农民伯伯：<http://www.cnblogs.com/over140/>

## 正文

### 一、效果图



看到这个图时你可以先想想如果是你，你怎么实现这个效果。马上想到用  
RelativeLayout? NO,NO,NO,,,

## 二、实现代码

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:clipChildren="false"
 android:orientation="vertical" >

 <android.support.v4.view.ViewPager
 android:id="@+id/view_page" />
```

```
 android:layout_width="match_parent"
 android:layout_height="0dip"
 android:layout_weight="1.0" />

 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="48dip"
 android:background="#B0C4DE"
 android:orientation="horizontal" >

 <ImageView
 android:layout_width="0dip"
 android:layout_height="fill_parent"
 android:layout_weight="1.0"
 android:scaleType="fitCenter"
 android:src="@drawable/ic_launcher" />

 <ImageView
 android:layout_width="0dip"
 android:layout_height="fill_parent"
 android:layout_weight="1.0"
 android:scaleType="fitCenter"
 android:src="@drawable/ic_launcher" />

 <ImageView
 android:layout_width="0dip"
 android:layout_height="64dip"
 android:layout_gravity="bottom"
 android:layout_weight="1.0"
 android:scaleType="fitCenter"
 android:src="@drawable/ic_launcher" />

 <ImageView
 android:layout_width="0dip"
 android:layout_height="fill_parent"
 android:layout_weight="1.0"
 android:scaleType="fitCenter"
 android:src="@drawable/ic_launcher" />

 <ImageView
 android:layout_width="0dip"
 android:layout_height="fill_parent"
 android:layout_weight="1.0"
 android:scaleType="fitCenter"
 android:src="@drawable/ic_launcher" />
 </LinearLayout>

</LinearLayout>
```

代码说明：

- 1、只需在根节点设置android:clipChildren为false即可， 默认为true
- 2、可以通过android:layout\_gravity控制超出的部分如何显示。
- 3、android:clipChildren的意思：是否限制子View在其范围内

### 三、 其他

在做动画的时候非常有用（听说的，，，回头写例子试试看）



# 怎样阅读源代码

来源：[怎样阅读源代码](#)

原题目：[How To Read Source Code](#), 原作者：Aria Stewart

中文翻译：

这篇文章基于我在Oneshot Nodeconf Christchurch的一个演讲。

我本来没有想要写这篇文章。程序员不读源代码听起来似乎是很荒谬的。然后我真遇到了一群不读源代码的程序员。接着我又跟更多的人进行了交谈，发现他们除了读代码示例或测试脚本之外什么也不看。最重要的是，我遇到过很多新手程序员，对他们来说，找到从哪个地方开始阅读是非常困难的。

## 当我们说读源代码的时候，我们要表达什么意思？

我们是为了什么去读源代码？为了理解它。为了找bug，为了知道这些代码和系统中的其他软件是怎样交互的。我们还会为了回顾、品评而去读。为了找出其中的接口信息，为了理解和找到不同模块之间的界线，为了学习，我们都会去读源代码。

## 读代码不是一个线性过程

读代码的过程不是线性的。人们往往认为读源代码就像读一本书一样：先搞定简介或者 README，然后从第一章开始一章一章的读，直到结束。其实并不是这样的。我们甚至都不能确定一个程序有没有结束，很多程序是不会终止的。我们应该在章节、模块之间跳转，反复阅读。我们也可以选择通读单个模块，但是这样我们就无法理解这个模块所引用的其他模块的代码。我们也可以根据程序的执行顺序去阅读，但是我们最后并不会清楚程序会向哪里执行。

## 读的顺序

要从一个库的入口开始读吗？对于Node库来说，即从index.js或者主脚本开始？

在浏览器中的情况呢？即使是找到程序入口，弄清楚载入了哪些文件，是怎样载入的都是一个很关键的事情。去研究文件之间是怎样关联起来的是一个很好的着手点。

除此之外，还可以从最大的一个源文件开始读。或者尝试在debugger中设置一个“浅”断点，通过函数调用去追溯源代码。亦或在某些复杂晦涩的地方设置一个深断点，然后去读函数调用栈里面的每一个函数。

## 代码的分类

我通常习惯于以语言去区分源代码，如Javascript, C++, ES6, Befunge, Forth或者LISP。熟悉的语言读起来相对更容易，对于不太熟悉的语言，我们往往就不会去看了。

这里还有另外一种看待源代码的方法，就是基于每一的模块的功能去分类：

- 连接类代码 (Glue)
- 接口定义类代码
- 实现类
- 算法类
- 配置类
- 任务类

关于算法类的代码已经有很多研究了，因为学术界就是干这个的。算法就是用数学模型去处理一件事情的方法。其他种类的代码就要模糊很多了，但是我认为它们更加有趣。它们才是绝大多数人在编程的时候写的代码。

当然，很多时候一个模块可以同时做以上的很多事情。而读代码首先要做的事情之一恰恰就是弄清楚每一个部分是在做什么。

## 什么是连接类代码

并不是所有的接口都能很好的工作在一起。连接类代码是将各模块连在一起的管道。中间件、Promise和绑定回调函数的代码、将参数导入object中或者解析object的代码，这些都属于连接类代码。

## 怎样阅读连接类代码

关键要弄清楚两个接口是怎样以不同方式构建起来的，以及它们共通的地方。

下面的例子来自于Ben Drucker的 `stream-to-promise`

```
internals.writable = function (stream) {
 return new Promise(function (resolve, reject) {
 stream.once('finish', resolve);
 stream.once('error', reject);
 });
};
```

可以看到，上面两个接口分别是Node的stream和Promise接口。

它们的共同点在于两者完成的时候都会执行一个操作，在Node中通过事件（event）来实现，而在Promise中通过调用resolve函数实现。

可以看到，Promise只能执行一次，但是stream可以发出同样的事件很多次。

更多连接类代码的例子：

```
var record = {
 name: (input.name || '').trim(),
 age: isNaN(Number(input.age)) ? null : Number(input.age),
 email: validateEmail(input.email.trim())
}
```

在读连接类代码的时候，还可以思考的是报错情况下的处理。

一段程序会抛出错误吗？能删除坏值吗？或者可以将其强制转换成可以接受的值？对于它们执行的环境来说，这些是正确的选择吗？类型转换是否是有损的？这些问题都是很值得思考的。

## 什么是接口定义类代码

可能你写过一些只在一两个地方用得到的模块，它们基本是在内部使用的，你不希望有人费很大劲去读这些模块。

而接口定义类代码却恰恰和上述情况相反。它们是模块之间泾渭分明的边界线。

下面的例子来自于Node的events.js：

```
exports.usingDomains = false;

function EventEmitter() { }
exports.EventEmitter = EventEmitter;

EventEmitter.prototype.setMaxListeners = function setMaxListeners(n) { };
EventEmitter.prototype.emit = function emit(type) { };
EventEmitter.prototype.addListener = function addListener(type, listener) { };
EventEmitter.prototype.on = EventEmitter.prototype.addListener;
EventEmitter.prototype.once = function once(type, listener) { };
EventEmitter.prototype.removeListener = function removeListener(type, listener) { };
EventEmitter.prototype.removeAllListeners = function removeAllListeners(type) {};
EventEmitter.prototype.listeners = function listeners(type) { };
EventEmitter.listenerCount = function(emitter, type) { };
```

上面的代码在定义EventEmitter的接口。

关于这类代码可以思考的问题包括：它们是否完全？能提供哪些保证？内部的细节信息会暴露给用户吗？

在一个有严格接口定义的架构中，上面就是这类代码应该出现的地方。

就像连接类代码一样，我们可以思考它是怎样处理错误和报错的。处理方法前后一致吗？能区分出因为内部不一致而出现的错误和因为用户使用不当而出现的错误吗？

## 实现类代码

```
startRouting: function() {
 this.router = this.router || this.constructor.map(K);

 var router = this.router;
 var location = get(this, 'location');
 var container = this.container;
 var self = this;
 var initialURL = get(this, 'initialURL');
 var initialTransition;

 // Allow the Location class to cancel the router setup while it refreshes
 // the page
 if (get(location, 'cancelRouterSetup')) {
 return;
 }

 this._setupRouter(router, location);

 container.register('view:default', _MetamorphView);
 container.register('view:toplevel', EmberView.extend());

 location.onUpdateURL(function(url) {
 self.handleURL(url);
 });

 if (typeof initialURL === "undefined") {
 initialURL = location.getURL();
 }
 initialTransition = this.handleURL(initialURL);
 if (initialTransition && initialTransition.error) {
 throw initialTransition.error;
 }
},
```

上面的示例取自Ember.Router。

这里往往是需要在“为什么”上作更多说明的地方。

读这类代码的时候应着重思考它是怎样跟更大的整体相融合的。

从公共接口中传入的值是怎样的？哪些需要验证(validation)？包含我们认为该有的值吗？会影响到其他哪些部分？当需要改写以加入新功能的时候会有哪些潜在危险？可能会导致程序崩溃的地方有哪些？有测试代码来对其进行测试吗？变量的生命周期是什么？

## 算法

算法类代码是实现类代码的一种，通常是封装起来不对外暴露的。它可以说是程序的血肉。也是一款软件的业务逻辑和主进程所在。

```
function Grammar(rules) {
 // Processing The Grammar
 //
 // Here we begin defining a grammar given the raw rules, terminal
 // symbols, and symbolic references to rules
 //
 // The input is a list of rules.
 //
 // The input grammar is amended with a final rule, the 'accept' rule,
 // which if it spans the parse chart, means the entire grammar was
 // accepted. This is needed in the case of a nulling start symbol.
 rules.push(Rule('_accept', [Ref('start')]));
 rules.acceptRule = rules.length - 1;
```

上面的代码出自一个叫做lotsawa的解析引擎。

人们常说好的注释应该告诉读者为什么这件事情要这样做，而不是一段代码在做什么。算法类代码通常需要更多的解释，因为如果是一个很简单的算法的话，那通常它就已经被吸纳进标准库中了。

下面这段代码是计算机系学生喜欢的（或者有糟糕记忆的）：

```
// Build a list of all the symbols used in the grammar so they can be numbered instead
// by name, and therefore their presence can be represented by a single bit in a set.
function censusSymbols() {
 var out = [];
 rules.forEach(function(r) {
 if (!~out.indexOf(r.name)) out.push(r.name);

 r.symbols.forEach(function(s, i) {
 var symNo = out.indexOf(s.name);
 if (!~out.indexOf(s.name)) {
 symNo = out.length;
 out.push(s.name);
 }

 r.symbols[i] = symNo;
 });
 });

 r.sym = out.indexOf(r.name);
});

return out;
}

rules.symbols = censusSymbols();
```

读起来像是数学论文，对吗？

在读算法类代码的时候需要关注的事情之一就是其运用的数据结构。上面的程序建了一个符号列表，并确保其中没有重复元素。

读的时候同时也要注意有关程序时间复杂度的线索。上面的代码中，有两个循环。用大O来记的话，时间复杂度就是 $O(n * m)$ 。但已经有人注意到，循环之中还有`indexOf`的调用。这在Javascript中也是循环操作。因此这又在时间复杂度上加了一个因子。还好这段代码并不是该算法的主要部分。

## 配置

源代码和配置文件之间的界线非常的窄。对配置文件来说，表达力强、可读性强和直接之间永远是冲突的。

```

app.configure('production', 'staging', function() {
 app.enable('emails');
});

app.configure('test', function() {
 app.disable('emails');
});

```

上面是一个用Javascript进行配置的例子。

在这里可能会遇到的问题是不同选项的组合爆炸性增长。应该配置多少个环境？在每一个环境实例中又配置哪些东西？我们很容易就会过度配置，去考虑所有的情况，但是bug可能只出现于其中一种情况中。时刻注意配置文件给我们多少自由度是非常有用的。

```

"express": {
 "env": "", // NOTE: `env` is managed by the framework. This value will be overwri
 "x-powered-by": false,
 "views": "path:./views",
 "mountpath": "/"
},

"middleware": {

 "compress": {
 "enabled": false,
 "priority": 10,
 "module": "compression"
 },

 "favicon": {
 "enabled": false,
 "priority": 30,
 "module": {
 "name": "serve-favicon",
 "arguments": ["resolve:kraken-js/public/favicon.ico"]
 }
 },
}

```

上面是一小段kraken的配置文件。

Kraken采取了“低功耗(low power)语言”的路，其配置文件采用JSON。更多一点“配置”，而相对少一点“源代码”。其目的之一就是让可选择项的数目可控。很多工具都采用简单的key-value对或者ini类的文件来做配置是有道理的，即使这样会使配置文件的表达力受限。

配置类的代码有如下一些值得思考的有趣而独特的限制：

- 生命周期
- 机器可写性
- 对一段代码负有责任的人会多很多
- 怎样适用在一些奇怪的地方，比如环境变量
- 往往有涉及安全的敏感信息
- 任务类

对50张信用卡计费，用编译器和其他构建工具开发一个复杂的软件，发出100封电子邮件。这些事情有什么共同点？

事务性。通常一个系统的某一部分只需严格执行一次，而遇到错误的话则完全不执行。编译器遗留下的错误构建文件是bug的一大来源。对客户重复收费是很糟糕的。因为重试而向用户邮箱滥发邮件也很可怕。

重启性。能根据系统状态，在之前退出的地方继续运行。

序列性。对于不是严格线性的程序，通常都有一个方向明晰的执行流程。循环是其中一大块。

- 阅读杂乱代码

一般人会怎样入手下面的一段代码：

```
DuplexCombination.prototype.on = function(ev, fn) {
 switch (ev) {
 case 'data':
 case 'end':
 case 'readable':
 this.reader.on(ev, fn);
 return this
 case 'drain':
 case 'finish':
 this.writer.on(ev, fn);
 return this
 default:
 return Duplex.prototype.on.call(this, ev, fn);
 }
};
```

对，这就是反向缩进，要怪就怪Issac吧。

## 美化一下！

## standard -F dc.js

```
DuplexCombination.prototype.on = function (ev, fn) {
 switch (ev) {
 case 'data':
 case 'end':
 case 'readable':
 this.reader.on(ev, fn)
 return this
 case 'drain':
 case 'finish':
 this.writer.on(ev, fn)
 return this
 default:
 return Duplex.prototype.on.call(this, ev, fn)
 }
}
```

# 读代码的时候使用工具是很好的。

又比如下面这段代码：

```
(function(t,e){if(typeof define==="function"&&define.amd){define(["underscore",
 "jquery","exports"],function(i,r,s){t.Backbone=e(t,s,i,r)}))else if(typeof export
 s!=="undefined"){var i=require("underscore");e(t,exports,i)}else{t.Backbone=e(t,
 {},t._,t.jQuery||t.Zepto||t.ender||t.$)})(this,function(t,e,i,r){var s=t.Backbo
 ne;var n=[];var a=n.push;var o=n.slice;var h=n.splice;e.VERSION="1.1.2";e.$=r;e.
 noConflict=function(){t.Backbone=s;return this};e.emulateHTTP=false;e.emulateJSO
 N=false;var u=e.Events={on:function(t,e,i){if(!c(this,"on",t,[e,i])||!e)retur
 n this;this._events||(this._events={});var r=this._events[t]||(this._events[t]=[]);
 r.push({callback:e,context:i,ctx:i||this});return this},once:function(t,e,r){if(
 !c(this,"once",t,[e,r])||!e)return this;var s=this;var n=i.once(function(){s.off
 (t,n);e.apply(this,arguments)});n._callback=e;return this.on(t,n,r)},off:functio
 n(t,e,r){var s,n,a,o,h,u,l,f;if(!this._events||!c(this,"off",t,[e,r]))return thi
 s;if(!t&&!e&&!r){this._events=void 0;return this}o=t?[t]:i.keys(this._events);fo
 r(h=0,u=o.length;h<u;h++){t=o[h];if(a=this._events[t]){this._events[t]=s=[];if(e
 ||r){for(l=0,f=a.length;l<f;l++){n=a[l];if(e&&e!==n.callback&&e!==n.callback._ca
 llback||r&&r!==n.context){s.push(n)}}}if(!s.length)delete this._events[t]}return thi
 s},trigger:function(t){if(!this._events)return this;var e=o.call(arguments,
 1);if(!c(this,"trigger",t,e))return this;var i=this._events[t];var r=this._event
 s.all;if(i)f(i,e);if(r)f(r,arguments);return this},stopListening:function(t,e,r)
 {var s=this._listeningTo;if(!s)return this;var n=!e&&!r;if(!r&&typeof e==="objec
 "||e===null||e===undefined){s=null}else{s[e]=r}this._listeningTo=s}}})
```

uglifyjs -b < backbone-min.js

```

(function(t, e) {
 if (typeof define === "function" && define.amd) {
 define(["underscore", "jquery", "exports"], function(i, r, s) {
 t.Backbone = e(t, s, i, r);
 });
 } else if (typeof exports !== "undefined") {
 var i = require("underscore");
 e(t, exports, i);
 } else {
 t.Backbone = e(t, {}, t._, t.jQuery || t.Zepto || t.ender || t.$);
 }
})(this, function(t, e, i, r) {
 var s = t.Backbone;
 var n = [];
 var a = n.push;
 var o = n.slice;
 var h = n.splice;
 e.VERSION = "1.1.2";
 e.$ = r;
 e.noConflict = function() {

```

## 人的部分

猜测代码作者的意图有很多的方法。

## 找guards和coercions

```
if (typeof arg != 'number') throw new TypeError("arg must be a number");
```

看上去函数的定义域是数值型。

```
arg = Number(arg)
```

强制转换为数值类型。和前面的一样，只是不再抛出错误。可能会有NaN出现。

## 找默认值

```
arg = arg || {}
```

默认为空。

```
arg = (arg == null ? true : arg)
```

如果没有相关参数传进来的话，则默认为true。

## 找层 (layers)

Express中的req和res是跟web相连的。它们会作用到哪一层呢？能够找到接口的边界吗？

## 找轨迹 (tracing)

有检查点吗？有debug日志吗？它们是一个完整的功能，还是之前的bug残留下来的呢？

## 找自反性 (reflexivity)

识别符是动态生成的吗？有eval、元编程和新函数定义吗？func.toString()是一个很好的切入点。

## 找生命周期

- 被谁初始化的
- 什么时候会改变
- 被谁改变
- 上述信息会在其他地方出现吗
- 会出现不一致性吗

某个时间，某个人输入了一个值。在另外的某个地方，某个时候这个值会对其他人产生影响，他们是谁？是什么或者谁来决定的？这个值会改变吗？由谁来改变？

## 找隐藏状态机 (hidden state machines)

有时候布尔变量会被当作解构了的状态机使用。

例如，变量isReadied和isFinished可能隐含如下状态：

```
var isReadied = false;
var isFinished = false;
```

或： START -> READY -> FINISHED

| isReadied | isFinished | state    |
|-----------|------------|----------|
| false     | false      | START    |
| false     | true       | invalid  |
| true      | false      | READY    |
| true      | true       | FINISHED |

## 找构造 (composition) 和继承

这段代码有我认识的部分吗？它们有名字吗？

## 找相同的操作

map, reduce, cross-join。

是时候开始读源代码了，Enjoy！

## Github:

<https://github.com/freedombird9/how-to-read-source-code>

# 面试时，问哪些问题能试出一个Android应用开发者真正的水平？

来源:[稀土掘金](#)

这是我在知乎上关于《[面试时，问哪些问题能试出一个Android应用开发者真正的水平？](#)》的回答，大家感觉有些帮助，就收录在这里，以下是我的回答：

一般面试时间短则30分钟，多则1个小时，这么点时间要全面考察一个人难度很大，需要一些技巧，这里我不局限于回答题主的问题，而是分享一下我个人关于如何做好Android技术面试的一些经验：

## 面试前的准备

### 1. 简历调查

简历到你手上的时候，你要做好充分的调查分析，不仅仅是对公司负责，也是对自己与候选人时间的尊重，明显不match的简历，就不要抱着“要不喊过来试试看”的想法了，候选人也许很不错，但如果跟你的岗位不match，也不要浪费大家时间，你要想清楚现在需要的人是有潜力可以培养的，还是亟需帮忙干活的。另外如果简历里附带了博客链接，GitHub地址，相关作品的，可以提前去看看，直接看人家多年积累的文章与代码，比这短短一小时的面试来得靠谱的多。

### 2. 准备问题

了解清楚候选人背景后，要根据简历，有针对性的准备问题，可以是他作品或做过项目里的某个技术细节的实现方式，也可以是他声称精通的某些领域的相关问题。总之不要等到面试过程中现想问题，特别是刚开始面试别人的同学，往往经验不足稍带紧张导致大脑短路，其实也是很尴尬的，把要问的问题提前写下来，准备充分。

## 考察哪些点？

### 1. 简历是否真实

这其实是面试第一要务，面试的过程其实就是看简历是否属实的过程，因为能到面试环节，说明这个人是符合要求的，不满足要求的早就被剔除了，如果他真的如简历描述的那样，100%会招过来，如果人人都如此，那就不需要有面试这种过程了。

需要注意的是这里的真实有三层含义：

- 一是他如实描述了自身经历，很多人只在一些大项目里做一个很小的螺丝钉，但简历里往往夸张这段经历。
- 二是不知道自己不知道，常见于简历里各种“精通”开头的描述，因为知识体系与视野的局限，明明只是了解很浅却夸口精通，很多时候他并不认为自己说的有问题，而是真的以为自己已然精通，有点井底之蛙的感觉。
- 三是简历里的真实要与你的期望相匹配，一门技术了解到怎样的程度才算精通，很难有定论，所以这里的“真实”只能是候选人与面试官标准之间的契合，这种有主观运气成分，也许面试官水平不够错误判断了你，也不用感到不爽，面试何尝不是种双向选择呢。

## 2. 技术的深度

技术的深度一向是我最看重的部分，当今任何一个技术领域都非常宽广，一个人要同时掌握那么多知识并且都深入几乎不可能，那都需要拼学习效率与工作年限了。而你曾经做过的东西，正在做的东西，是绝对可以了解得更深入的，一个对技术有好奇心，有技术热情的人，都不会仅仅停留在这个东西挺好用，而是会忍不住去探究它背后的技术原理，即便不是亲自去看源码，也会花点时间了解别人整理过的经验，所以单凭考察技术上的深度，就可以考察一个人是否对技术有热情，是否有技术好奇心等等这些很多大牛认为的所谓“优秀程序员的特征”。

之前曾看到过一句话：“一个人对他所做的事情了解得越深，他就能做的越好”。放在这里再合适不过了。

## 3. 技术的广度

深度是有了，还需要广度吗？我个人的理解是：深度是必要条件，广度是加分项。同样的有技术好奇心的优秀程序员，也不会满足于仅仅局限于自己的一亩三分地，工作之余，也会想要尝试一些其它的领域和方向，因为投入问题也许不够深入，但很多领域知识你知道与不知道，对你个人知识体系的形成关系很大。比如你要实现一个功能，在你当前熟悉的技术领域上很困难或者效果不佳，在你就要放弃时你的同事告诉你，这用一个简单sql语句就可以实现啦，为什么要搞得那么麻烦？这个例子虽然举得很蹩脚，但是我想意思大家应该已经明白了。知识越有广度，头脑里的技术体系就越完备，同样的问题，你就

可以想到N个解，思考一下就得出最优解了，如果你听都没听过一些东西，就会经常说出“这个好难搞啊”，“这根本就不可能”，其实有的时候真是知识的局限问题，所谓的从0到1难，也是这个意思。

## 4. 逻辑思维能力

这也是我比较看重的一点，这里并不是指那些臭名昭彰的脑经急转弯问题，而是通过交流观察，判断一个人表达观点逻辑是否清晰，回答问题是否有章法，这个很难描述，但如果你细心观察，你会发现很容易通过一些简单的交流，就可以看出一个人是否逻辑清晰。有时候你会觉得某个人表达沟通很不错，其实不是沟通的问题，是他说出去的话，经过了他大脑的条理清晰的整理，让你很容易就能明白。这种习惯不是一朝一夕就能养成的，所以面试过程中这点装不出来。

另外一个人如果逻辑清晰，而且反应又敏捷，语速很快，那是大大的加分项，恭喜你，碰到一个聪明人了。

# 具体问哪些问题？

前面提到的是要重点考察的点，那么具体的Android开发，有没有一些通用的问题可以问的呢？我个人一般会从这几个角度考察候选人：

## 1. Android经验

如果不是校招，Android经验是必须的，我比较喜欢问一些基础概念与技术原理，比如Activity、View、Window的理解，各LaunchMode的使用场景，View的绘制流程，Touch事件机制，Android动画的原理，Handler, Looper的理解，Android跨进程通讯的方式，Binder的理解，Android Mashup设计的理解等等。

## 2. Java水平

基本上就是Effective Java那本书里提到的东西，如果你背完那本书里的问题，并且对答如流，没问题，就要你这样的。其实也会考察关于final用法，反射原理，注解原理，java编译过程，GC等一些常见问题。

## 3. IT基础知识

其实就是计算机科班学生学校里学到的一些东西，在校招时这块是重点，社招会放宽，但一些基本的常识是要有的，比如不少人都不知道http的get post有啥区别，https的那个s是什么意思，讲不清进程与线程的概念，不知道二分算法是个啥东西。这些简单问题的筛选，可以过滤一些所谓野路子的程序员，是不是科班出身不重要，搞这行就得对一些基本常识有概念，不然以后怎么愉快的交流呢？

## 4. 代码质量的认识

我们需要的是一个对代码味道有感觉的人，关于这点，看下《Clean Code》就够了，面试中这点其实不好考察，可以让他聊一聊对代码质量的认识，虽然不能排除对方夸夸其谈，至少想法不多，只能提到命名风格这一点的人是不符合要求的，也可以在写Code的环节中观察。

## 5. 技术视野

比如对Android开发新技术的了解与学习，对其它流行技术领域的了解，这其实与我刚才提到的技术广度的考察有关，就我面试过程中，发现很多非互联网行业的从业人员，因为公司各种操蛋规定与公司技术氛围的原因，技术视野相当狭窄。

我个人对这点深有体会，2011年我还在传统行业从事软件研发，当时的公司因为担心技术信息泄露，不让上网，相当封闭，我个人虽然自认为已在那个行业内做到业内专家的级别，但总感觉哪里不对，有一天我很兴奋的打算跟身边同事聊一聊Android的时候，发现他们居然都不知Android为何物？2011年啊同志们，当时的震惊无法言表，深切感觉到需要作出改变了，毅然放弃多年行业积累，转战移动互联网，直到现在。时至今日，多年前的小伙伴也有很多混出了名堂，开始走向人生巅峰，我也从来没有后悔当初做出的选择。

## 6. 技术想象力

一个优秀的技术人，如果知识的深度与广度足够，知识已成体系，那么他对于一些从未接触过的领域，也是可以做出足够合理的想象与判断，面试过程中如果问到一些领域候选人没有涉猎，这时候一般不用过多纠缠，但如果想借这个问题考察下他的技术想象力，可以深入下去，比如问他：“你觉得这个东西应该是什么原理呢？”，“这个酷炫的控件，如果要你来做，你会怎么实现？”。在这方面表现出色的同学无疑是有深厚基础与足够广度的人。

## 7. 技术习惯

好的程序员都会有好的习惯，比如各种快捷键的熟练应用，各种命令行的掌握，一些提高开发效率的工具与习惯，碰到问题是baidu还是google，有没有做一些小工具帮助减少重复工作，工作之余有没有继续学习？有没有看什么不错的书等等，这些小细节很大程度上决定了程序员的开发效率，这也是为什么很多人说一个优秀程序员抵得上100个普通程序员，这也是重要原因之一。

## 面试后的反馈：

面试一般不止一轮，你需要给出你的反馈，多轮面试结果一起考量，减少误判的风险，反馈一般怎么写呢？以下是我的建议：

### 1. 面试纪录

面试过程中的完整纪录，尽量客观评价，让其它面试官知道你问了哪些问题，回答的怎么样，也避免了重复问题的尴尬。

### 2. 优点与缺点

你的主观评价，亮点有哪些，你觉得哪些地方不够好？

### 3. 综合评价

你对候选人的综合评价，hire或者no hire的根本原因，如果有些地方感觉没考察清楚，期望其它面试官继续加强考察，也可以写上。

### 4. 怎样才给通过？

通过标准因人而异，每个人都有自己心中的bar，但还是有些可直观考量的因素的：

- 一是岗位的要求，不同的岗位标准当然不一样，校招与设招肯定也不一样。
- 二是岗位的紧急程度，兄弟们天天加班忙死了，赶紧找人过来帮忙吧哈哈。
- 三是候选人的年龄，大龄程序员莫怪，一把年纪了还跟刚毕业一两年的同事一个水平，说明成长太慢，做技术的潜力有限，这个大家应该能理解。
- 四是前面提到的做技术的深度，这个是必须的，广度也要有一些，视野不能太窄。
- 五是要有亮点，大家在面试的过程中要注意发掘亮点，有时候他问题很多但有一个足够的亮点就够了，用心观察也发现不了什么亮点的，就要注意了。





# Android打包的那些事

来源:[www.jayfeng.com](http://www.jayfeng.com)

使用gradle打包apk已经成为当前主流趋势，我也在这个过程中经历了各种需求，并不断结合gradle新的支持，一一改进。在此，把这些相关的东西记录，做一总结。

## 1. 替换AndroidManifest中的占位符

我想把其中的\${app\_label}替换为@string/app\_name

```
android{
 defaultConfig{
 manifestPlaceholders = [app_label:@"string/app_name"]
 }
}
```

如果只想替换debug版本：

```
android{
 buildTypes {
 debug {
 manifestPlaceholders = [app_label:@"string/app_name_debug"]
 }
 release {
 }
 }
}
```

更多的需求是替换渠道编号：

```
android{
 productFlavors {
 // 把dev产品型号的apk的AndroidManifest中的channel替换dev
 "dev"{
 manifestPlaceholders = [channel:"dev"]
 }
 }
}
```

## 2. 独立配置签名信息

对于签名相关的信息,直接写在gradle当然不好,特别是一些开源项目, 可以添加到gradle.properties:

```
RELEASE_KEY_PASSWORD=xxxx
RELEASE_KEY_ALIAS=xxx
RELEASE_STORE_PASSWORD=xxx
RELEASE_STORE_FILE=../../keystore/xxx.jks
```

然后在build.gradle中引用即可:

```
android {
 signingConfigs {
 release {
 storeFile file(RELEASE_STORE_FILE)
 storePassword RELEASE_STORE_PASSWORD
 keyAlias RELEASE_KEY_ALIAS
 keyPassword RELEASE_KEY_PASSWORD
 }
 }
}
```

如果不提交到版本库, 可以添加到local.properties中, 然后在build.gradle中读取。

## 3. 多渠道打包

多渠道打包的关键之处在于, 定义不同的product flavor, 并把AndroiManifest中的channel渠道编号替换为对应的flavor标识:

```
android {
 productFlavors {
 dev{
 manifestPlaceholders = [channel:"dev"]
 }
 official{
 manifestPlaceholders = [channel:"official"]
 }
 //
 wandoujia{
 manifestPlaceholders = [channel:"wandoujia"]
 }
 xiaomi{
 manifestPlaceholders = [channel:"xiaomi"]
 }
 "360"{
 manifestPlaceholders = [channel:"360"]
 }
 }
}
```

注意一点，这里的flavor名如果是数字开头，必须用引号想起来。

构建一下，就能生成一系列的Build Variant了：

```
devDebug
devRelease
officialDebug
officialRelease
wandoujiaDebug
wandoujiaRelease
xiaomiDebug
xiaomiRelease
360Debug
360Release
```

其中debug, release是gradle默认自带的两个build type, 下一节还会继续说明。

选择一个，就能编译出对应渠道的apk了。

## 4. 自定义Build Type

前面说到默认的build type有两种debug和release，区别如下：

## 4. 自定义Build Type

前面说到默认的build type有两种debug和release，区别如下：

```
// release版本生成的BuildConfig特性信息
public final class BuildConfig {
 public static final boolean DEBUG = false;
 public static final String BUILD_TYPE = "release";
}

// debug版本生成的BuildConfig特性信息
public final class BuildConfig {
 public static final boolean DEBUG = true;
 public static final String BUILD_TYPE = "debug";
}
```

现在有一种需求，增加一种build type，介于debug和release之间，就是和release版本一样，但是要保留debug状态（如果做过rom开发的话，类似于user debug版本），我们称为preview版本吧。其实很简单：

```
android {
 signingConfigs {
 debug {
 storeFile file(RELEASE_STORE_FILE)
 storePassword RELEASE_STORE_PASSWORD
 keyAlias RELEASE_KEY_ALIAS
 keyPassword RELEASE_KEY_PASSWORD
 }
 preview {
 storeFile file(RELEASE_STORE_FILE)
 storePassword RELEASE_STORE_PASSWORD
 keyAlias RELEASE_KEY_ALIAS
 keyPassword RELEASE_KEY_PASSWORD
 }
 release {
 storeFile file(RELEASE_STORE_FILE)
 storePassword RELEASE_STORE_PASSWORD
 keyAlias RELEASE_KEY_ALIAS
 keyPassword RELEASE_KEY_PASSWORD
 }
 }

 buildTypes {
 debug {
 manifestPlaceholders = [app_label:@"string/app_name_debug"]
 }
 release {
 manifestPlaceholders = [app_label:@"string/app_name"]
 }
 preview{
 manifestPlaceholders = [app_label:@"string/app_name_preview"]
 }
 }
}
```

另外，build type还有一个好处，如果想要一次性生成所有的preview版本，执行assemblePreview即可，debug和release版本同理。

## 5. build type中的定制参数

上面我们在不同的build type替换\${app\_label}为不同的字符串，这样安装到手机上就能明显的区分出不同build type的版本。除此之外，可能还可以配置一些参数，我这里列几个我在工作中用到的：

```

 android {
 debug {
 manifestPlaceholders = [app_label:@"string/app_name_debug"]
 applicationIdSuffix ".debug"
 minifyEnabled false
 signingConfig signingConfigs.debug
 proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-r
 }
 release {
 manifestPlaceholders = [app_label:@"string/app_name"]
 minifyEnabled true
 shrinkResources true
 signingConfig signingConfigs.release
 proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-r
 }
 preview{
 manifestPlaceholders = [app_label:@"string/app_name_preview"]
 applicationIdSuffix ".preview"
 debuggable true // 保留debug信息
 minifyEnabled true
 shrinkResources true
 signingConfig signingConfigs.preview
 proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-r
 }
 }
}

```

这些都用的太多了，稍微解释一下：

```

// minifyEnabled 混淆处理
// shrinkResources 去除无用资源
// signingConfig 签名
// proguardFiles 混淆配置
// applicationIdSuffix 增加APP ID的后缀
// debuggable 是否保留调试信息
// ...

```

## 6. 多工程全局配置

随着产品渠道的铺开，往往一套代码需要支持多个产品形态，这就需要抽象出主要代码到一个Library，然后基于Library扩展几个App Module。

相信每个module的build.gradle都会有这个代码：

```
android {
 compileSdkVersion 22
 buildToolsVersion "23.0.1"

 defaultConfig {
 minSdkVersion 10
 targetSdkVersion 22
 versionCode 34
 versionName "v2.6.1"
 }
}
```

当升级sdk、build tool、target sdk等，几个module都要更改，非常的麻烦。最重要的是，很容易忘记，最终导致app module之间的差异不统一，也不可控。

强大的gradle插件在1.1.0支持全局变量设定，一举解决了这个问题。

先在project的根目录下的build.gradle定义ext全局变量：

```
ext {
 compileSdkVersion = 22
 buildToolsVersion = "23.0.1"
 minSdkVersion = 10
 targetSdkVersion = 22
 versionCode = 34
 versionName = "v2.6.1"
}
```

然后在各module的build.gradle中引用如下：

```
android {
 compileSdkVersion rootProject.ext.compileSdkVersion
 buildToolsVersion rootProject.ext.buildToolsVersion

 defaultConfig {
 applicationId "com.xxx.xxx"
 minSdkVersion rootProject.ext.minSdkVersion
 targetSdkVersion rootProject.ext.targetSdkVersion
 versionCode rootProject.ext.versionCode
 versionName rootProject.ext.versionName
 }
}
```

然后每次修改project级别的build.gradle即可实现全局统一配置。

## 7. 自定义导出的APK名称

默认android studio生成的apk名称为app-debug.apk或者app-release.apk，当有多个渠道的时候，需要同时编出50个渠道包的时候，就麻烦了，不知道谁是谁了。

这个时候，就需要自定义导出的APK名称了，不同的渠道编出的APK的文件名应该是不一样的。

```
android {
 // rename the apk with the version name
 applicationVariants.all { variant ->
 variant.outputs.each { output ->
 output.outputFile = new File(
 output.outputFile.parent,
 "ganchai-${variant.buildType.name}-${variant.versionName}-${variant.name}.apk"
)
 }
 }
}
```

当apk太多时，如果能把apk按debug, release, preview分一下类就更好了（事实上，对于我这样经常发版的人，一编往往就要编四五十个版本的人，debug和release版本全混在一起没法看，必须分类），简单：

```
android {
 // rename the apk with the version name
 // add output file sub folder by build type
 applicationVariants.all { variant ->
 variant.outputs.each { output ->
 output.outputFile = new File(
 output.outputFile.parent + "/${variant.buildType.name}",
 "ganchai-${variant.buildType.name}-${variant.versionName}-${variant.name}.apk"
)
 }
 }
}
```

现在生成了类似于ganchai-dev-preview-v2.4.0.0.apk这样格式的包了，preview的包自然就放在preview的文件夹下，清晰明了。

## 8. 混淆技巧

混淆能让反编译的代码可读性变的很差，而且还能显著的减少APK包的大小。

## 1). 第一个技巧

相信很多朋友对混淆都觉得麻烦，甚至说，非常乱。因为添加混淆规则需要查询官方说明文档，甚至有的官方文档还没说明。当你引用了太多库后，添加混淆规则将使一场噩梦。这里介绍一个技巧，不用查官方文档，不用逐个库考虑添加规则。首先，除了默认的混淆配置(android-sdk/tools/proguard/proguard-android.txt)，自己的代码肯定是要自己配置的：

```
位于module下的proguard-rules.pro
#####
主程序不能混淆的代码
#####

-dontwarn xxx.model.**
-keep class xxx.model.** { *; }

等等，自己的代码自己清楚

#####
不优化泛型和反射
#####

-keepattributes Signature
```

接下来是麻烦的第三方库，一般来说，如果是极光推的话，它的包名是cn.jpush，添加如下代码即可：

```
-dontwarn cn.jpush.**
-keep class cn.jpush.** { *; }
```

其他的第三库也是如此，一个一个添加，太累！其实可以用第三方反编译工具（比如 jadx：<https://github.com/skylot/jadx>），打开apk后，一眼就能看到引用的所有第三方库的包名，把所有不想混淆或者不确定能不能混淆的，直接都添加又有何不可：

```
#####
第三方库或者jar包
#####

-dontwarn cn.jpush.**
-keep class cn.jpush.** { *; }

-dontwarn com.squareup.**
-keep class com.squareup.** { *; }

-dontwarn com.octo.**
-keep class com.octo.** { *; }

-dontwarn de.**
-keep class de.** { *; }

-dontwarn javax.**
-keep class javax.** { *; }

-dontwarn org.**
-keep class org.** { *; }

-dontwarn u.aly.**
-keep class u.aly.** { *; }

-dontwarn uk.**
-keep class uk.** { *; }

-dontwarn com.baidu.**
-keep class com.baidu.** { *; }

-dontwarn com.facebook.**
-keep class com.facebook.** { *; }

-dontwarn com.google.**
-keep class com.google.** { *; }

... ...

```

## 2). 第二个技巧

一般release版本混淆之后，像友盟这样的统计系统如果有崩溃异常，会记录如下：

```
java.lang.NullPointerException: java.lang.NullPointerException
 at com.xxx.TabMessageFragment$7.run(Unknown Source)
```

这个Unknown Source是很要命的，排除错误无法定位到具体行了，大大降低调试效率。

当然，友盟支持上传Mapping文件，可帮助定位，mapping文件的位置在：

```
project > module
 > build > outputs > {flavor name} > {build type} > mapping.txt
```

如果版本一多，mapping.txt每次都要重新生成，还要上传，终归还是麻烦。

其实，在proguard-rules.pro中添加如下代码即可：

```
-keepattributes SourceFile,LineNumberTable
```

当然apk会大那么一点点（我这里6M的包，大个200k吧），但是再也不用mapping.txt也能定位到行了，为了这种解脱，这个代价我个人觉得是值的，而且超值！

## 9. 动态设置一些额外信息

假如想把当前的编译时间、编译的机器、最新的commit版本添加到apk，而这些信息又不好写在代码里，强大的gradle给了我们创造可能的自信：

```
android {
 defaultConfig {
 resValue "string", "build_time", buildTime()
 resValue "string", "build_host", hostName()
 resValue "string", "build_revision", revision()
 }
}
def buildTime() {
 return new Date().format("yyyy-MM-dd HH:mm:ss")
}
def hostName() {
 return System.getProperty("user.name") + "@" + InetAddress.localHost.hostName
}
def revision() {
 def code = new ByteArrayOutputStream()
 exec {
 commandLine 'git', 'rev-parse', '--short', 'HEAD'
 standardOutput = code
 }
 return code.toString()
}
```

上述代码实现了动态的添加了3个字符串资源：build\_time、build\_host、build\_revision，然后在其他地方可像如引用字符串一样使用如下：

```
// 在Activity里调用
getString(R.string.build_time) // 输出2015-11-07 17:01
getString(R.string.build_host) // 输出jay@deepin, 这是我的电脑的用户名和PC名
getString(R.string.build_revision) // 输出3dd5823, 这是最后一次commit的sha值
```

这个地方，如何从命令行读取返回结果，很有意思。

其实这段代码来自我学习VLC源码时偶然看到，深受启发，不敢独享，特摘抄在此。

VLC源码及编译地址：<https://wiki.videolan.org/AndroidCompile>，有兴趣可以过去一观。

## 10. 给自己留个”后门”：点七下

为了调试方便，我们往往会在debug版本留一个显示我们想看的界面（记得之前微博的一个iOS版本就泄露了一个调试界面），如何进入到一个界面，我们可以仿照android开发者选项的方式，点七下才显示，我们来实现一个：

```
private int clickCount = 0;

private long clickTime = 0;

sevenClickView.setOnClickListener(new View.OnClickListener() {

 @Override
 public void onClick(View view) {
 if (clickTime == 0) {
 clickTime = System.currentTimeMillis();
 }
 if (System.currentTimeMillis() - clickTime > 500) {
 clickCount = 0;
 } else {
 clickCount++;
 }
 clickTime = System.currentTimeMillis();
 if (clickCount > 6) {
 // 点七下条件达到，跳到debug界面
 }
 }
});
```

release版本肯定是不能暴露这个界面的，也不能让人用am在命令行调起，如何防止呢，可以在release版本把这个debug界面的exported设为false。

## 11. 自动化构建

如何使用jenkins打包android和ios，并上传到蒲公英平台，这个可以参考我的另外一篇文章专门介绍：《[使用jenkins自动化构建android和ios应用](#)》，不过，这篇文章还没写完，实际上在公司里已经一直在用了，哪天心情好了总会写完的，这里不再赘述。

## 12. 小结

android打包因为groovy语言的强大，变的强大的同时必然也变的复杂，今天把我经历的这些门道拿出来说道一下，做一个小小的总结，后续有更新我还会添加。

# 一种为 Apk 动态写入信息的方案

来源:[微信公众号](#)

这篇文章来自天猫无线-基础业务团队童鞋的投稿。

他们之前投稿过一篇《Android App 安全的 HTTPS 通信》，建议大家到历史记录看看。

## 1. 背景

我们在日常使用应用可能会遇到以下场景。

- 场景1： 用户浏览 H5 页面时，通过该页面下载链接下载安装 App，启动会来到首页而不是用户之前浏览的 H5 页面，造成使用场景的割裂。
- 场景2： 用户通过二维码把一个页面分享出去，没有装猫客的用户如果直接安装启动后，无法回到分享的页面。

如果用户在当前页面下载了应用，安装之后直接跳转到刚才浏览的界面，不仅可以将这一部分流量引回客户端，还可以让用户获得完整的用户体验。下面提出一种方案来满足这个业务需求。

## 2. 原理

Android 使用的 Apk 包的压缩方式是 zip，与 zip 有相同的文件结构，在 zip 文件的末尾有一个 Central Directory Record 区域，其末尾包含一个 File comment 区域，可以存放一些数据，所以 File comment 是 zip 文件一部分，如果可以正确的修改这个部分，就可以在不破坏压缩包、不用重新打包的前提下快速的给 Apk 文件写入自己想要的数据。

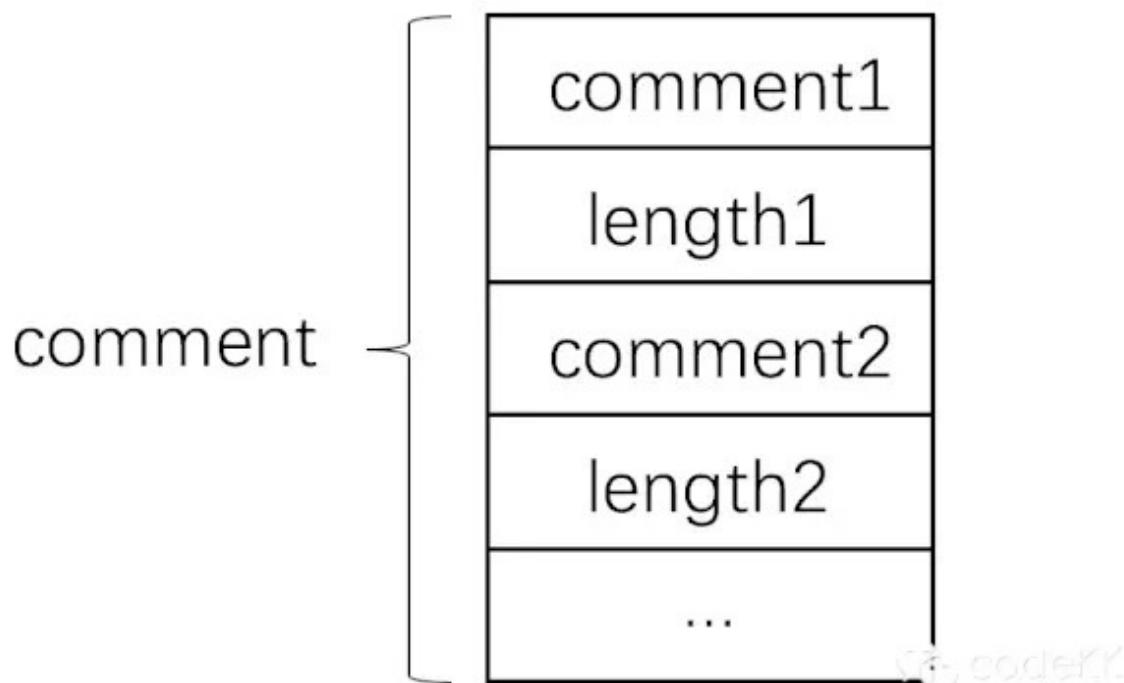
comment 是在 Central Directory Record 末尾储存的，可以将数据直接写在这里，下表是 header 末尾的结构。

End of central directory record(EOCD)

| Offset | Bytes | Description                                                       |
|--------|-------|-------------------------------------------------------------------|
| 0      | 4     | End of central directory signature=0x06054b50                     |
| 4      | 2     | Number of this disk                                               |
| 6      | 2     | Disk where central directory starts                               |
| 8      | 2     | Number of central directory records on this disk                  |
| 10     | 2     | Total number of central directory records                         |
| 12     | 4     | Size of central directory(bytes)                                  |
| 16     | 4     | Offset of start of central directory,relative to start of archive |
| 20     | 2     | Comment length(n)                                                 |
| 22     | n     | Comment                                                           |

从表中可以看到定义 comment 长度的字段位于 comment 之前，当我们从文件最后开始读取时，由于 comment 数据是不确定的，我们无法知道它的长度，从而也无法从 zip 中直接获取 Comment length。

这里我们需要自定义 comment，在自定义 comment 内容的后面添加一个区域储存 comment 的长度，结构如下图。



## 3. 实现

### 1 服务端将数据写入 Apk 的 comment

这一部分可以在本地或服务端进行，需要定义一个长度为 2 的 byte[] 来储存 comment 的长度，直接使用 Java 的 api 就可以把 comment 和 comment 的长度写到 Apk 的末尾，代码如下：

```
public static void writeApk(File file, String comment) {
 ZipFile zipFile = null;
 ByteArrayOutputStream outputStream = null;
 RandomAccessFile accessFile = null;
 try {
 zipFile = new ZipFile(file);
 String zipComment = zipFile.getComment();
 if (zipComment != null) {
 return;
 }

 byte[] byteComment = comment.getBytes();
 outputStream = new ByteArrayOutputStream();

 outputStream.write(byteComment);
 outputStream.write(short2Stream((short) byteComment.length));

 byte[] data = outputStream.toByteArray();

 accessFile = new RandomAccessFile(file, "rw");
 accessFile.seek(file.length() - 2);
 accessFile.write(short2Stream((short) data.length));
 accessFile.write(data);
 } catch (IOException e) {
 e.printStackTrace();
 } finally {
 try {
 if (zipFile != null) {
 zipFile.close();
 }
 if (outputStream != null) {
 outputStream.close();
 }
 if (accessFile != null) {
 accessFile.close();
 }
 } catch (Exception e) {
 }
 }
}
```

## 2 Apk 中读取 comment 数据

首先获取 Apk 的路径，通过 context 中的 `getPackageCodePath()` 方法就可以获取，代码如下：

```

public static String getPackagePath(Context context) {
 if (context != null) {
 return context.getPackageCodePath();
 }
 return null;
}

```

获取 Apk 路径之后就可以读取 comment 的内容了，这里不能直接使用 ZipFile 中的 getComment() 方法直接获取 comment，因为这个方法是 Java7 中的方法，在 android4.4 之前是不支持 Java7 的，所以我们需要自己去读取 Apk 文件中的 comment。

首先根据之前自定义的结构，先读取写在最后的 comment 的长度，根据这个长度，才可以获取真正 comment 的内容，代码如下：

```

public static String readApk(File file) {
 byte[] bytes = null;
 try {
 RandomAccessFile accessFile = new RandomAccessFile(file, "r");
 long index = accessFile.length();

 bytes = new byte[2];
 index = index - bytes.length;
 accessFile.seek(index);
 accessFile.readFully(bytes);

 int contentLength = stream2Short(bytes, 0);

 bytes = new byte[contentLength];
 index = index - bytes.length;
 accessFile.seek(index);
 accessFile.readFully(bytes);

 return new String(bytes, "utf-8");
 } catch (FileNotFoundException e) {
 e.printStackTrace();
 } catch (IOException e) {
 e.printStackTrace();
 }
 return null;
}

```

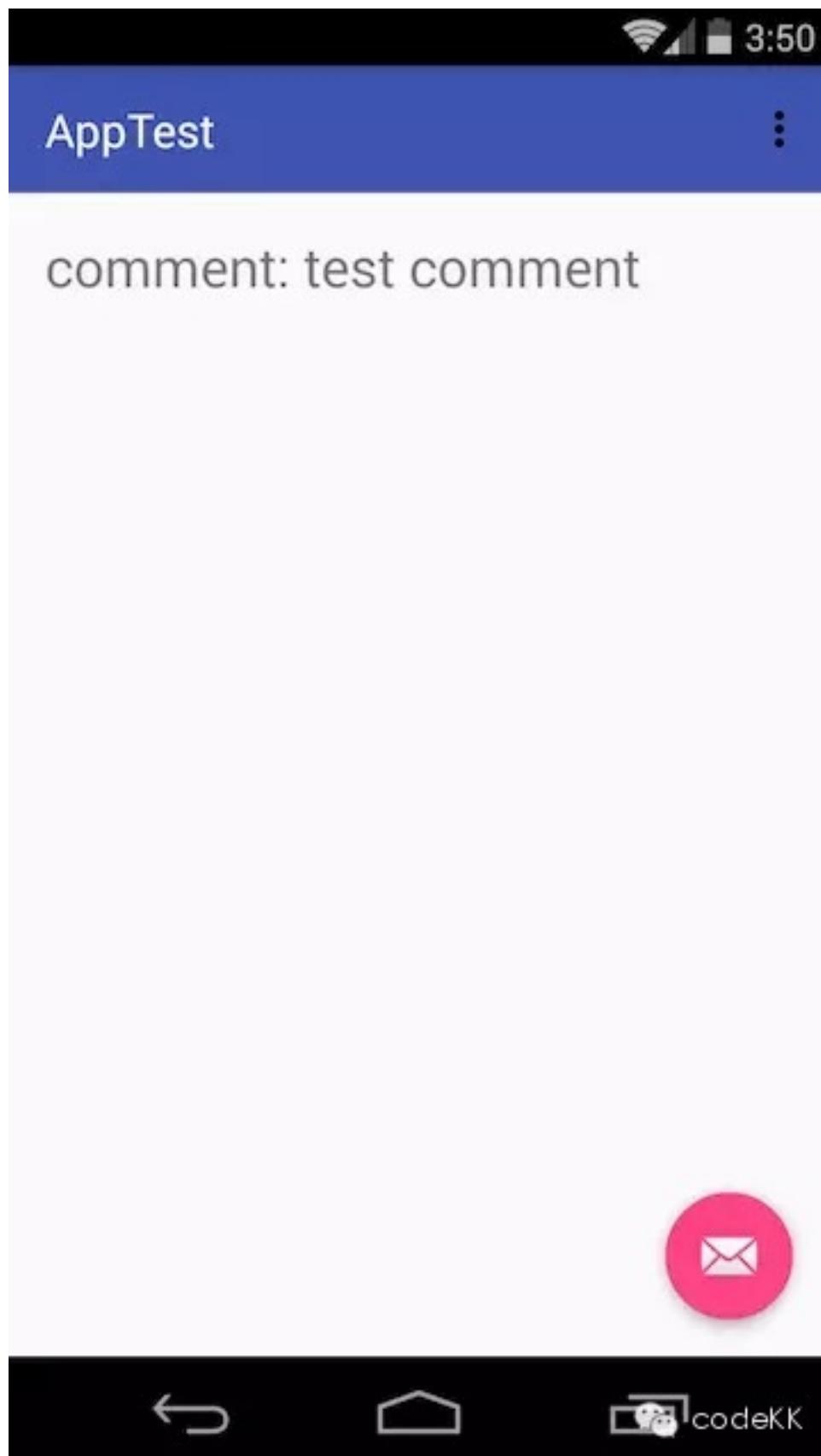
这里的 stream2Short() 和 short2Stream() 参考了 MultiChannelPackageTool 中的方法。

## 4. 测试

在生成 Apk 后，调用下面的代码写入我们想要的数据，

```
File file = new File("/Users/zhaolin/app-debug.apk");writeApk(file, "test comment");
```

安装这个 Apk 之后运行，让 comment 显示在屏幕上，运行结果如下。



运行结果符合预期，安装包也没有被破坏，可以正常安装。

## 5. 结论

通过修改 comment 将数据传递给 App 的方案是可行的，由于是修改 Apk 自有的数据，  
并不会对 Apk 造成破坏，修改后可以正常安装。

这种方案不用重新打包 Apk，并且在服务端只是写文件的操作，效率很高，可以适用于动  
态生成 Apk 的场景。

可以通过这个方案进行 H5 到 App 的引流，用户操作不会产生割裂感，保证用户体验的统  
一。

## 6. 参考

<https://github.com/mcxiaoke/packer-ng-plugin>

<https://github.com/seven456/MultiChannelPackageTool>

以上两个开源项目都是利用同样的原理实现 Android 快速打渠道包

关于渠道包美团有个类似的 hack 方式，通过添加固定规则文件名的空文件，避免重新签  
名，从而实现一分钟打近千个渠道包。可回复 30 查看《国内四个不错的技术团体博客》  
中介绍

# 批量打包-gradle-配置Manifest

节选来源:[stormzhang.com](http://stormzhang.com)

由于国内Android市场众多渠道，为了统计每个渠道的下载及其它数据统计，就需要我们针对每个渠道单独打包，如果让你打几十个市场的包岂不烦死了，不过有了Gradle，这再也不是事了。

## 友盟多渠道打包

废话不多说，以友盟统计为例，在AndroidManifest.xml里面会有这么一段：

```
<meta-data
 android:name="UMENG_CHANNEL"
 android:value="Channel_ID" />
```

里面的Channel\_ID就是渠道标示。我们的目标就是在编译的时候这个值能够自动变化。

- 第一步 在AndroidManifest.xml里配置PlaceHolder

```
<meta-data
 android:name="UMENG_CHANNEL"
 android:value="${UMENG_CHANNEL_VALUE}" />
```

- 第二步 在build.gradle设置productFlavors

```
android {
 productFlavors {
 xiaomi {
 manifestPlaceholders = [UMENG_CHANNEL_VALUE: "xiaomi"]
 }
 _360 {
 manifestPlaceholders = [UMENG_CHANNEL_VALUE: "_360"]
 }
 baidu {
 manifestPlaceholders = [UMENG_CHANNEL_VALUE: "baidu"]
 }
 wandoujia {
 manifestPlaceholders = [UMENG_CHANNEL_VALUE: "wandoujia"]
 }
 }
}
```

或者批量修改为渠道名

```
android {
 productFlavors {
 xiaomi {}
 _360 {}
 baidu {}
 wandoujia {}
 }

 productFlavors.all {
 flavor -> flavor.manifestPlaceholders = [UMENG_CHANNEL_VALUE: name]
 }
}
```

很简单清晰有没有？直接执行 `./gradlew assembleRelease`，然后就可以静静的喝杯咖啡等待打包完成吧。

# 使用gradle批量打包

## 1.signingConfigs(签名配置)

在 build.gradle 文件的 android 块中添加 signingConfigs

例子：

```
signingConfigs {
 debugConfigs{
 storeFile file('/Users/laowang/keystore/debug/debug.keystore')
 keyAlias 'androiddebugkey'
 keyPassword 'android'
 storePassword 'android'

 }
 releaseConfigs {
 keyAlias 'laowang'
 keyPassword 'laowang'
 storeFile file('/Users/laowang/keystore/release/release.keystore')
 storePassword 'laowang'

 }
}
```

- 这一步也可以在图形界面中完成，具体步骤如下：

```
File ->
Project Structure ->
选择具体的Module ->
选择signing选项卡 ->
底部可以选择添加或者删除配置, 右侧可以配置具体的配置内容
```

## 2、buildTypes(构建类型配置)

在 build.gradle 文件的 android 块中添加 buildTypes

```

buildTypes {
 release {
 minifyEnabled false
 proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules
 }
 qihu360{
 signingConfig signingConfigs.releaseConfigs
 minifyEnabled true
 zipAlignEnabled true
 }
 bd{
 signingConfig signingConfigs.releaseConfigs
 minifyEnabled true
 zipAlignEnabled true
 }
 anzhi{
 signingConfig signingConfigs.releaseConfigs
 minifyEnabled true
 zipAlignEnabled true
 }
 wandoujia{
 signingConfig signingConfigs.releaseConfigs
 minifyEnabled true
 zipAlignEnabled true
 }
}

```

- 这一步也可以在图形界面中完成，具体步骤如下：

```

File ->
Project Structure ->
选择具体的Module ->
选择Build Type选项卡 ->
底部可以选择添加或者删除配置，右侧可以配置具体的配置内容

```

### 3、在项目根文件夹下生成保存要替换文件的目录

在项目根文件夹下生成保存要替换文件的目录，并添加具体的渠道子文件夹，把 Manifest.xml 文件拷贝到各个目录下，并修改相应的渠道号(注意某些版本需要对meta-data添加 tools:replace="android:value")

例子：

```
/
.... /app/
..... /channels/
..... /bd/
..... /qihu360/
..... /anzhi/
..... /wandoujia/
```

## Manifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 package="com.xinye.test">

 <application android:allowBackup="true" android:label="@string/app_name"
 android:icon="@mipmap/ic_launcher" android:theme="@style/AppTheme">
 <!-- 在各个不同的文件夹下, value替换成相应的渠道号 -->
 <meta-data android:name="CHANNEL" android:value="anzhi"
 tools:replace="android:value"/>

 <activity android:name=".TestActivity"
 android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
 android:screenOrientation="portrait">

 <intent-filter>
 <action android:name="android.intent.action.MAIN"/>
 <category android:name="android.intent.category.LAUNCHER"/>
 </intent-filter>

 </activity>
 </application>

</manifest>
```

## 4、sourceSets(设置替换文件夹)

在 build.gradle 文件的 andorid 块中添加 sourceSet

```
sourceSets{
 bd.setRoot('channels/bd')
 qihu360.setRoot('channels/qh360')
 anzhi.setRoot('channels/anzhi')
 wandoujia.setRoot('channels/wandoujia')
}
```

## 5、构建

点击 `graile -> 具体的moudle -> Task -> build -> build` 进行构建

可以在 `/具体的模块/build/outputs/apk` 文件夹下看到生成的apk文件

## 6、附加内容

读取Manifest文件中的渠道号的代码

```
public static String getApplicationMetadata(Context context, String metaDataKey) {
 ApplicationInfo info = null;
 try {
 PackageManager pm = context.getPackageManager();

 info = pm.getApplicationInfo(context.getPackageName(),
 PackageManager.GET_META_DATA);

 return String.valueOf(info.metaData.get(metaDataKey));
 } catch (Exception e) {
 e.printStackTrace();
 }
 return null;
}
```

# 批量打包-使用Python

## 1. 使用方式

### 1.1 按照正常流程打包APK

### 1.2 修改渠道文件channels.txt

文件内的每一行一个渠道号

例子：

```
360
xiaomi
baidu
91
guanwang
offline
tencent
wandoujia
```

### 1.3 批量打包Python脚本

机器上一定要安装python,建议安装2.7.\*版本

### 1.4 执行python脚本

```
python package.py APK文件名 输出文件夹名
```

例子：

```
python package.py test-2015-05-19.apk out
```

## 2 原理

- 如果能直接修改apk的渠道号，而不需要再重新签名能节省不少打包的时间。幸运的是我们找到了这种方法。直接解压apk，解压后的根目录会有一个META-INF目录
- 如果在META-INF目录内添加空文件，可以不用重新签名应用。因此，通过为不同渠道的应用添加不同的空文件，可以唯一标识一个渠道。
- 下面的python代码用来给apk添加空的渠道文件，渠道名的前缀为laowang\_：

```
import zipfile
import shutil
import sys
import os

apk_path = sys.argv[1]
out_path = sys.argv[2]

if not os.path.exists(out_path):
 os.makedirs(out_path)

name = os.path.basename(apk_path)

channels_file = open('channels.txt')

origin_apk_name = os.path.splitext(name)[0]

for channel in channels_file:
 channel_apk_name = "{}_{}.apk".format(origin_apk_name, channel.strip())
 channel_apk_path = os.path.join(out_path, channel_apk_name)
 shutil.copy2(apk_path, channel_apk_path)
 zipped = zipfile.ZipFile(channel_apk_path, 'a', zipfile.ZIP_DEFLATED)
 empty_channel_file = "META-INF/laowang_{}".format(channel.strip())
 zipped.writestr(empty_channel_file, '')
 zipped.close()
```

- 执行Python命令,将会输出所有指定渠道号的APK文件

```
python package.py test-2015-05-19-2.apk out
```

- 在Android中得到渠道号

```
public static String getMetaInfChannel(Context context) {
 ApplicationInfo appinfo = context.getApplicationInfo();
 String sourceDir = appinfo.sourceDir;
 String ret = "";
 ZipFile zipfile = null;
 try {
 zipfile = new ZipFile(sourceDir);
 Enumeration<?> entries = zipfile.entries();
 while (entries.hasMoreElements()) {
 ZipEntry entry = ((ZipEntry) entries.nextElement());
 String entryName = entry.getName();
 //如果想修改此标示，直接编辑pack.py即可
 if (entryName.startsWith("META-INF/laowang")) {
 ret = entryName;
 break;
 }
 }
 } catch (IOException e) {
 e.printStackTrace();
 } finally {
 if (zipfile != null) {
 try {
 zipfile.close();
 } catch (IOException e) {
 e.printStackTrace();
 }
 }
 }
 String[] split = ret.split("_");
 if (split != null && split.length >= 2) {
 return ret.substring(split[0].length() + 1);
 } else {
 return "";
 }
}
```

# 批量打包-美团gradle

来源:[美团](#)

## 概述

前一篇文章([美团Android自动化之旅—生成渠道包](#))介绍了Android中几种生成渠道包的方式，基本解决了打包慢的问题。

但是，随着渠道越来越多，不同渠道对应用的要求也不尽相同。例如，有的渠道要求美团客户端的应用名为美团，有的渠道要求应用名为美团团购。又比如，有些渠道要求应用不能使用第三方统计工具（如flurry）。总之，每次打包都需要对这些渠道进行适配。

之前的做法是为每个需要适配的渠道创建一个Git分支，发版时再切换到相应的分支，并合并主分支的代码。适配的渠道比较少的话这种方式还可以接受，如果分支比较多，对开发人员来说简直就是噩梦。还好，自从有了Gradle flavor，一切都变得简单了。本文假定读者使用过Gradle，如果还不了解建议先阅读相关文档。

## Flavor

先来看build.gradle文件中的一段代码：

```
android {
 ...

 productFlavors {
 flavor1 {
 minSdkVersion 14
 }
 }
}
```

上例定义了一个flavor: flavor1，并指定了应用的minSdkVersion为14（当然还可以配置更多的属性，具体可参考相关文档）。与此同时，Gradle还会为该flavor关联对应的sourceSet，默认位置为 `src/<flavorName>` 目录，对应到本例就是 `src/flavor1`。

接下来，要做的就是根据具体的需求在build.gradle文件中配置flavor，并添加必要的代码和资源文件。以flavor1为例，运行 gradle assembleFlavor1 命令既可生成所需的适配包。下面主要介绍美团团购Android客户端的一些适配案例。

## 案例

### 使用不同的包名

美团团购Android客户端之前有两个版本：手机版(**com.meituan.group**)和hd版(**com.meituan.group.hd**)，两个版本使用了不同的代码。目前hd版对应的代码已不再维护，希望能直接使用手机版的代码。解决该问题可以有多种方法，不过使用flavor相对比较简单，示例如下：

```
productFlavors {
 hd {
 applicationId "com.meituan.group.hd"
 }
}
```

上面的代码添加了一个名为hd的flavor，并指定了应用的包名为**com.meituan.group.hd**，运行 gradle assembleHd 命令即可生成hd适配包。

### 控制是否自动更新

美团团购Android客户端在启动时会默认检查客户端是否有更新，如果有更新就会提示用户下载。但是有些渠道和应用市场不允许这种默认行为，所以在适配这些渠道时需要禁止自动更新功能。

解决的思路是提供一个配置字段，应用启动的时候检查该字段的值以决定是否开启自动更新功能。使用flavor可以完美的解决这类问题。

Gradle会在generateSources阶段为flavor生成一个BuildConfig.java文件。BuildConfig类默认提供了一些常量字段，比如应用的版本名(VERSION\_NAME)，应用的包名(PACKAGE\_NAME)等。更强大的是，开发者还可以添加自定义的一些字段。下面的示例假设wandoujia市场默认禁止自动更新功能：

```

 android {
 defaultConfig {
 buildConfigField "boolean", "AUTO_UPDATES", "true"
 }

 productFlavors {
 wandoujia {
 buildConfigField "boolean", "AUTO_UPDATES", "false"
 }
 }
 }
}

```

上面的代码会在BuildConfig类中生成AUTO\_UPDATES布尔常量，默认值为true，在使用wandoujia flavor时，该值会被设置成false。接下来就可以在代码中使用AUTO\_UPDATES常量来判断是否开启自动更新功能了。最后，运行gradle assembleWandoujia命令即可生成默认不开启自动升级功能的渠道包，是不是很简单。

## 使用不同的应用名

最常见的一类适配是修改应用的资源。例如，美团团购Android客户端的应用名是美团，但有的渠道需要把应用名修改为美团团购；还有，客户端经常会和一些应用分发市场合作，需要在应用的启动界面中加上第三方市场的Logo，类似这类适配形式还有很多。Gradle在构建应用时，会优先使用flavor所属dataSet中的同名资源。所以，解决思路就是在flavor的数据集中添加同名的字符串资源，以覆盖默认的资源。下面以适配wandoujia渠道的应用名为美团团购为例进行介绍。

首先，在build.gradle配置文件中添加如下flavor：

```

 android {
 productFlavors {
 wandoujia {
 }
 }
 }
}

```

上面的配置会默认src/wandoujia目录为wandoujia flavor的数据集。

接下来，在src目录内创建wandoujia目录，并添加如下应用名字符串资源

( src/wandoujia/res/values/appname.xml ) :

```
<resources>
 <string name="app_name">美团团购</string>
</resources>
```

默认的应用名字符串资源如下（`src/main/res/values/strings.xml`）：

```
<resources>
 <string name="app_name">美团</string>
</resources>
```

最后，运行 `gradle assembleWandoujia` 命令即可生成应用名为美团团购的应用了。

## 使用第三方SDK

某些渠道会要求客户端嵌入第三方SDK来满足特定的适配需求。比如360应用市场要求美团团购Android客户端的精品应用模块使用他们提供的SDK。问题的难点在于如何只为特定的渠道添加SDK，其他渠道不引入该SDK。使用flavor可以很好的解决这个问题，下面以为qihu360 flavor引入 `com.qihoo360.union.sdk:union:1.0` SDK为例进行说明：

```
android {
 productFlavors {
 qihu360 {
 }
 }
}
...
dependencies {
 provided 'com.qihoo360.union.sdk:union:1.0'
 qihu360Compile 'com.qihoo360.union.sdk:union:1.0'
}
```

上例添加了名为qihu360的flavor，并且指定编译和运行时都依赖 `com.qihoo360.union.sdk:union:1.0`。而其他渠道只是在构建的时候依赖该SDK，打包的时候并不会添加它。

接下来，需要在代码中使用反射技术判断应用程序是否添加了该SDK，从而决定是否要显示360 SDK提供的精品应用。部分代码如下：

```
class MyActivity extends Activity {
 private boolean useQihuSdk;

 @override
 public void onCreate(Bundle savedInstanceState) {
 try {
 Class.forName("com.qihoo360.union.sdk.UnionManager");
 useQihuSdk = true;
 } catch (ClassNotFoundException ignored) {

 }
 }
}
```

最后，运行 `gradle assembleQihu360` 命令即可生成包含360精品应用模块的渠道包了。

## 总结

适配是一项dirty工作，尤其是适配的渠道比较多的时候。上面介绍了几种使用Gradle flavor进行适配的例子，基本解决了繁杂的适配工作。

## 参考资料

- <http://www.gradle.org/>
- Product flavors



# Android博客周刊专题之 # 插件化开发 #

来源:[Android博客周刊专题之 # 插件化开发 #](#)

本期专栏讨论插件化开发。插件化涉及的东西很多，所以我们需要多个维度去学习。大概分为5个部分：预备知识、入门、进阶、系列、类库。一步一步深入了解插件的原理。本专栏会不定时更新相关内容，请留意更新的消息。请加入QQ群：149581646.会统一通知最新的文章。

Posted 2016-03-16 by Jomeslu.

## 基础

### 1. Java 类加载器

类加载器（class loader）是 Java™中的一个很重要的概念。类加载器负责加载 Java 类的字节代码到 Java 虚拟机中。本文首先详细介绍了 Java 类加载器的基本概念，包括代理模式、加载类的具体过程和线程上下文类加载器等，接着介绍如何开发自己的类加载器，最后介绍了类加载器在 Web 容器和 OSGi™中的应用。

### 2. 反射原理

Java 提供的反射机制允许您于执行时期动态载入类别、检视类别资讯、生成物件或操作生成的物件，要举反射机制的一个应用实例，就是在整合式开发环境中所提供的方法提示或是类别检视工具，另外像 JSP 中的 JavaBean 自动收集请求资讯也使用到反射，而一些軟體开发框架（Framework）也常见到反射机制的使用，以达到动态载入使用者自订类别的目的。

### 3. 代理模式及Java实现动态代理

定义：给某个对象提供一个代理对象，并由代理对象控制对于原对象的访问，即客户不直接操控原对象，而是通过代理对象间接地操控原对象。

## 入门

### 1. Android动态加载dex技术初探

Android使用Dalvik虚拟机加载可执行程序，所以不能直接加载基于class的jar，而是需要将class转化为dex字节码，从而执行代码。优化后的字节码文件可以存在一个.jar中，只要其内部存放的是.dex即可使用。

## 2.Android插件化入门

开发者将插件代码封装成Jar或者APK。宿主下载或者从本地加载Jar或者APK到宿主中。将宿主调用插件中的算法或者Android特定的Class（如Activity）

## 3.插件化开发—动态加载技术加载已安装和未安装的apk

动态加载技术就是使用类加载器加载相应的apk、dex、jar（必须含有dex文件），再通过反射获得该apk、dex、jar内部的资源（class、图片、color等等）进而供宿主app使用。

## 4.Android动态加载技术三个关键问题详解

动态加载技术（也叫插件化技术）在技术驱动型的公司中扮演着相当重要的角色，当项目越来越庞大的时候，需要通过插件化来减轻应用的内存和CPU占用，还可以实现热插拔，即在不发布新版本的情况下更新某些模块。

# 进阶

## 1.携程Android App插件化和动态加载实践

携程Android App的插件化和动态加载框架已上线半年，经历了初期的探索和持续的打磨优化，新框架和工程配置经受住了生产实践的考验。本文将详细介绍Android平台插件式开发和动态加载技术的原理和实现细节，回顾携程Android App的架构演化过程，期望我们的经验能帮助到更多的Android工程师。

## 2.动态加载APK原理分享

被加载的apk称之为插件，因为机制类似于生物学的“寄生”，加载了插件的应用也被称为宿主。往往不是所有的apk都可作为插件被加载，往往需要遵循一定的“开发规范”，还需要插件项目引入某种api类库，业界通常都是这么做的。

## 3.Android插件化的一种实现

Android的插件化已经是老生常谈的话题了，插件化的好处有很多：解除代码耦合，插件支持热插拔，静默升级，从根本上解决65K属性和方法的bug等等。下面给大家介绍一下我们正在用的差价化框架。本片主要以类图的方式向大家介绍插件话框架的实现。

## 4.蘑菇街 App 的组件化之路

随着我街业务的蓬勃发展，产品和运营随时上新功能新活动的需求越来越强烈，经常可以听到“有个功能我想周x上，行不行”。行么？当然是不行啦，上新功能得发新版本啊，到时候费时费力打乱开发节奏不说，覆盖率也是个问题。

- [那些年蘑菇街Android组件与插件化背后的故事 -- 插件篇\(一\)](#)
- [那些年蘑菇街Android组件与插件化背后的故事 -- 插件篇\(二\)](#)

## 5.DynamicLoadApk 源码解析

DynamicLoadApk 是一个开源的 Android 插件化框架。插件化的优点包括：(1) 模块解耦，(2) 动态升级，(3) 高效并行开发(编译速度更快)(4) 按需加载，内存占用更低等等 DynamicLoadApk 提供了 3 种开发方式，让开发者在无需理解其工作原理的情况下快速的集成插件化功能。

## 6.Android apk动态加载机制的研究

问题是这样的：我们知道，apk必须安装才能运行，如果不安装要是也能运行该多好啊，事实上，这不是完全不可能的，尽管它比较难实现。在理论层面上，我们可以通过一个宿主程序来运行一些未安装的apk，当然，实践层面上也能实现，不过这对未安装的apk有要求。我们的想法是这样的，首先要明白apk未安装是不能被直接调起来.

## 7.美团Android DEX自动拆包及动态加载简介

作为一个android开发者，在开发应用时，随着业务规模发展到一定程度，不断地加入新功能、添加新的类库，代码在急剧的膨胀，相应的apk包的大小也急剧增加，那么终有一天，你会不幸遇到这个错误.

## 8.途牛原创|途牛Android App的插件实现

途牛的插件化是基于dynamic-load-apk (github) 实现的。定义了宿主和插件的通信方式，使得两者能够互起对方的页面，调用彼此的功能。同时对activity的启动方式 singletask等进行了模式实现，并增加了对Service的支持等。总之使得插件开发最大限度的保持着原有的Android开发习惯。

## 9. Android apk资源加载和activity生命周期管理

博主分析了Android中apk的动态加载机制，并在文章的最后指出需要解决的两个复杂问题：资源的访问和activity生命周期的管理，而本文将会分析这两个复杂问题的解决方法。

(续6)

## 10.APK动态加载框架（DL）解析

首先要说的是动态加载技术（或者说插件化）在技术驱动型的公司中扮演这相当重要的角色，当项目越来越庞大的时候，需要通过插件化来减轻应用的内存和cpu占用，还可以实现热插拔，即在不发布新版本的情况下更新某些模块。

# 系列

## 0.Kaedea---Android动态加载技术 索引

### 1.Kaedea---Android动态加载技术 简单易懂的介绍

我们很早开始就在Android项目中采用了动态加载技术，主要目的是为了达到让用户不用重新安装APK就能升级应用的功能（特别是SDK项目），这样一来不但可以大大提高应用新版本的覆盖率，也减少了服务器对旧版本接口兼容的压力，同时如果也可以快速修复一些线上的BUG。

### 2.Kaedea---Android动态加载基础 ClassLoader的工作机制

早期使用过Eclipse等Java编写的软件的同学可能比较熟悉，Eclipse可以加载许多第三方的插件（或者叫扩展），这就是动态加载。这些插件大多是一些Jar包，而使用插件其实就是动态加载Jar包里的Class进行工作。

### 3.Kaedea---Android动态加载补充 加载SD卡的SO库

Android中JNI的使用其实就包含了动态加载，APP运行时动态加载.so库并通过JNI调用其封装好的方法。后者一般是使用NDK工具从C/C++代码编译而成，运行在Native层，效率会比执行在虚拟机的Java代码高很多，所以Android中经常通过动态加载.so库来完成一些对性能比较有需求的工作（比如T9搜索、或者Bitmap的解码、图片高斯模糊处理等）。

### 4.Kaedea---Android动态加载入门 简单加载模式

Java程序中，JVM虚拟机是通过类加载器ClassLoader加载.jar文件里面的类的。Android也类似，不过Android用的是Dalvik/ART虚拟机，不是JVM，也不能直接加载.jar文件，而是加载dex文件。

## 5.Kaedea---Android动态加载进阶 代理Activity模式

简单模式中，使用ClassLoader加载外部的Dex或Apk文件，可以加载一些本地APP不存在的类，从而执行一些新的代码逻辑。但是使用这种方法却不能直接启动插件里的Activity。

## 6.Kaedea---Android动态加载黑科技 动态创建Activity模式

还记得我们在代理Activity模式里谈到启动插件APK里的Activity的两个难题吗，由于插件里的Activity没在主项目的Manifest里面注册，所以无法经历系统Framework层级的一系列初始化过程，最终导致获得的Activity实例并没有生命周期和无法使用res资源。

## 7.尼古拉斯---插件开发基础篇：动态加载技术解读

在目前的软硬件环境下，Native App与Web App在用户体验上有着明显的优势，但在实际项目中有些会因为业务的频繁变更而频繁的升级客户端，造成较差的用户体验，而这也恰恰是Web App的优势。本文对网上Android动态加载jar的资料进行梳理和实践在这里与大家一起分享，试图改善频繁升级这一弊病。

## 8.尼古拉斯---插件开发开篇：类加载器分析

这篇文章主要介绍了Android中主要的两个类加载器：PathClassLoader和DexClassLoader,他们的区别，联系，用法等问题，以及我们在制作插件的过程中会遇到哪些常见的问题。这篇文章也是后续两篇文章的基础，因为如果不了解这两个类的话，我们将无法进行后续的操作。

## 9.尼古拉斯---插件开发中篇：资源加载问题(换肤原理解析)

这篇文章主要通过现在一些应用自带的换肤技术的解读来看看，在开发插件的过程中如何解决一些资源加载上的问题，这个问题为何要单独拿出来解释，就是因为他涉及的知识很多，也是后面一篇文章的基础，我们在需要加载插件中的资源文件的时候。

## 10.尼古拉斯---插件开发终极篇：动态加载Activity(免安装运行程序)

这篇文章主要是讲解了如何加载插件中的Activity。从而实现免安装运行程序，同时这篇文章也是对前三篇文章知识的综合使用。下载很多应用都会使用到插件技术，因为包的大小和一些功能的优先级来决定哪些模块可以制作成插件。

## 11.Weishu---Android插件化原理解析——概要

类的加载可以使用Java的ClassLoader机制，但是对于Android来说，并不是说类加载进来就可以用了，很多组件都是有“生命”的；因此对于这些有血有肉的类，必须给它们注入活力，也就是所谓的组件生命周期管理.

## 12.Weishu---Android插件化原理解析——Hook机制之动态代理

使用代理机制进行API Hook进而达到方法增强是框架的常用手段，比如J2EE框架Spring通过动态代理优雅地实现了AOP编程，极大地提升了Web开发效率；同样，插件框架也广泛使用了代理机制来增强系统API从而达到插件化的目的.

## 13.Weishu---Android插件化原理解析——Hook机制之Binder Hook

Android系统通过Binder机制给应用程序提供了一系列的系统服务，诸如ActivityManagerService, ClipboardManager, AudioManager等；这些广泛存在系统服务给应用程序提供了诸如任务管理，音频，视频等异常强大的功能。

## 14.Weishu---Android 插件化原理解析——Hook机制之AMS&PMS

在前面的文章中我们介绍了DroidPlugin的Hook机制，也就是代理方式和Binder Hook；插件框架通过AOP实现了插件使用和开发的透明性。在讲述DroidPlugin如何实现四大组件的插件化之前，有必要说明一下它对AMS以及PMS的Hook方式。

## 15.Weishu---Android 插件化原理解析——Activity生命周期管理

之前的 Android插件化原理解析 系列文章揭开了Hook机制的神秘面纱，现在我们手握倚天屠龙，那么如何通过这种技术完成插件化方案呢？具体来说，插件中的Activity, Service等组件如何在Android系统上运行起来？

## 16.Weishu---Android 插件化原理解析——插件加载机制

上文 Activity 生命周期管理 中我们地完成了『启动没有在AndroidManifest.xml中显式声明的Activity』的任务；通过Hook AMS和拦截ActivityThread中H类对于组件调度我们成功地绕过了AndroidManifest.xml的限制。

## 17.Weishu---Android插件化原理解析——广播的管理

在Activity生命周期管理 以及 插件加载机制 中我们详细讲述了插件化过程中对于Activity组件的处理方式，为了实现Activity的插件化我们付出了相当多的努力；那么Android系统的其他组件，比如BroadcastReceiver，Service还有ContentProvider，它们又该如何处理呢？

## 18.Weishu---Android 插件化原理解析——Service的插件化

在 Activity 生命周期管理 以及 广播的管理 中我们详细探讨了Android系统中的Activity、BroadcastReceiver组件的工作原理以及它们的插件化方案，相信读者已经对Android Framework

## 19.Weishu---Android插件化原理解析——ContentProvider的插件化

目前为止我们已经完成了Android四大组件中Activity，Service以及BroadcastReceiver的插件化，这几个组件各不相同，我们根据它们的特点定制了不同的插件化方案；那么对于ContentProvider，它又有什么特点？应该如何实现它的插件化？

## 20.Weishu---Android插件化原理解析——DroidPlugin插件通信机制

## 21.Weishu---Android插件化原理解析——插件机制之资源管理

## 22.Weishu---Android插件化原理解析——不同插件框架方案对比

## 23.Weishu---Android插件化原理解析——插件化的未来

## 类库

### 1.DroidPlugin

是360手机助手在Android系统上实现了一种新的插件机制

### 2.Android-Plugin-Framework

此项目是Android插件开发框架完整源码及示例。用来通过动态加载的方式在宿主程序中运行插件APK。

### 3.Small

世界那么大，组件那么小。Small，做最轻巧的跨平台插件化框架。里面有很详细的文档

### 4.dynamic-load-apk

Android 使用动态加载框架DL进行插件化开发

### 5.AndroidDynamicLoader

Android 动态加载框架，他不是用代理 Activity 的方式实现而是用 Fragment 以及 Schema 的方式实现

### 6.DynamicAPK

实现Android App多apk插件化和动态加载，支持资源分包和热修复.携程App的插件化和动态加载框架.

### 7.ACDD

非代理Android动态部署框架

### 8.android-pluginmgr

不需要插件规范的apk动态加载框架。

## 参考视频

## 1.android插件化及动态部署

阿里技术沙龙第十六期《android插件化及动态部署》视频

## 2.android插件化及动态部署

阿里技术沙龙第十六期《android插件化及动态部署》视频

微信公众号：Android博客周刊



微博：[陆镇生\\_Jomeslu](#)

邮箱：[luzhensheng72@gmail.com](mailto:luzhensheng72@gmail.com)



# Android Dynamic Loading



## 索引帖

来源:<http://segmentfault.com>

## 基本信息

**Author:** [kaedea](#)

**GitHub:** [android-dynamical-loading](#)

## 动态加载

在Android开发中采用基于ClassLoader的动态加载技术，可以达到不安装新APK就升级APP的目的，也可以用来修复一些紧急BUG。

[《Android动态加载技术 简单易懂的介绍方式》](#)

1. 动态加载技术在Android中的使用背景；
2. Android的动态的加载大致可以分为“加载SO库”和“加载DEX/JAR/APK”两种；
3. 动态加载的基础是类加载器ClassLoader；
4. “加载DEX/JAR/APK”的三种模式；
5. 采用动态加载的作用与代价；
6. 除了ClassLoader之外的动态修改代码的技术(HotFix)；
7. 一些动态加载的开源项目；

## 《Android动态加载基础 ClassLoader的工作机制》

1. 类加载器ClassLoader的创建过程和加载类的过程；
2. ClassLoader的双亲代理模式；
3. DexClassLoader和PathClassLoader之间的区别；
4. 使用ClassLoader加载外部类需要注意的一些问题；
5. 自定义ClassLoader (Hack开发)

## 《Android动态加载补充 加载SD卡的SO库》

1. 如何编译和使用SO库；
2. 如何加载SD卡中的SO库（也是动态加载APK需要解决的问题）；

## 《Android动态加载入门 简单加载模式》

1. 如何创建我们需要的dex文件；
2. 如何加载dex文件里面的类；
3. 动态加载dex文件在ART虚拟机的兼容性问题；

## 《Android动态加载进阶 代理Activity模式》

1. 如何启动插件APK中没有注册的Activity
2. 代理Activity模式开源项目“dynamic-load-apk”

## 《Android动态加载黑科技 动态创建Activity模式》

1. 如何在运行时动态创建一个Activity；
2. 自定义ClassLoader并偷梁换柱替换想要加载的类；
3. 动态创建Activity模式开源项目“android-pluginmgr”
4. 代理模式与动态创建类模式的区别；

# Android动态加载技术 简单易懂的介绍方式

来源：<http://segmentfault.com>

## 基本信息

我们很早开始就在Android项目中采用了动态加载技术，主要目的是为了达到让用户不用重新安装APK就能升级应用的功能，这样一来不但可以大大提高应用新版本的覆盖率，也减少了服务器对旧版本接口兼容的压力，同时如果也可以快速修复一些线上的BUG。

这种技术并不是常规的Android开发方式，早期并没有完善的解决方案。从“不明觉厉”到稳定投入生产，一直以来我总想对此编写一些文档，这也是这篇日志的由来，没想到前前后后竟然拖沓着编辑了一年多，所以日志里有的地方思路可能有点衔接得不是很好，日后我会慢慢修正。

## 技术背景

通过服务器配置一些参数，Android APP获取这些参数再做出相应的逻辑，这是常有的事情。

比如现在大部分APP都有一个启动页面，如果到了一些重要的节日，APP的服务器会配置一些与时节相关的图片，APP启动时候再把原有的启动图换成这些新的图片，这样就能提高用户的体验了。

再则，早期个人开发者在安卓市场上发布应用的时候，如果应用里包含有广告，那么有可能会审核不通过。那么就通过在服务器配置一个开关，审核应用的时候先把开关关闭，这样应用就不会显示广告了；安卓市场审核通过后，再把服务器的广告开关给打开，以这样的手段规避市场的审核。

**道高一尺魔高一丈。**安卓市场开始扫描APK里面的Manifest甚至dex文件，查看开发者的APK包里是否有广告的代码，如果有就有可能审核不通过。

通过服务器怕配置开关参数的方法行不通了，开发者们开始想，“既然这样，能不能先不要在APK写广告的代码，在用户运行APP的时候，再从服务器下载广告的代码，运行，再现实广告呢？”。答案是肯定的，这就是动态加载：

在程序运行的时候，加载一些程序自身原本不存在的可执行文件并运行这些文件里的代码逻辑。

看起来就像是应用从服务器下载了一些代码，然后再执行这些代码！

## 传统PC软件中的动态加载技术

动态加载技术在PC软件领域广泛使用，比如输入法的截图功能。刚刚安装好的输入法软件可能没有截图功能，当你第一次使用的时候，输入法会先从服务器下载并安装截图软件，然后再执行截图功能。

此外，许多的PC软件的安装目录里面都有大量的DLL文件（Dynamic Link Library），PC软件则是通过调用这些DLL里面的代码执行特定的功能的，这就是一种动态加载技术。

熟悉Java的同学应该比较清楚，Java的可执行文件是Jar，运行在虚拟机上JVM上，虚拟机通过ClassLoader加载Jar文件并执行里面的代码。所以Java程序也可以通过动态调用Jar文件达到动态加载的目的。

## Android应用的动态加载技术

Android应用类似于Java程序，只不过虚拟机换成了Dalvik/ART，而Jar换成了Dex。在Android APP运行的时候，我们是不是也可以通过下载新的应用，或者通过调用外部的Dex文件来实现动态加载呢？

然而在Android上实现起来可没那么容易，如果下载一个新的APK下来，不安装这个APK的话可不能运行。如果让用户手动安装完这个APK再启动，那可不像是动态加载，纯粹就是用户安装了一个新的应用，然后再启动这个新的应用。

动态调用外部的Dex文件则是完全没有问题的。在APK文件中往往有一个或者多个Dex文件，我们写的每一句代码都会被编译到这些文件里面，Android应用运行的时候就是通过执行这些Dex文件完成应用的功能的。虽然一个APK一旦构建出来，我们是无法更换里面的Dex文件的，但是我们可以通过加载外部的Dex文件来实现动态加载，这个外部文件可以放在外部存储，或者从网络下载。

## 动态加载的定义

开始正题之前，在这里可以先给动态加载技术做一个简单的定义。真正的动态加载应该是

1. 应用在运行的时候通过加载一些本地不存在的可执行文件实现一些特定的功能
2. 这些可执行文件是可以替换的
3. 更换静态资源（比如换启动图、换主题、或者用服务器参数开关控制广告的隐藏现实）

等) 不属于动态加载

4. Android中动态加载的核心思想是动态调用外部的**Dex文件**, 极端的情况下, Android APK自身带有的Dex文件只是一个程序的入口 (或者说空壳), 所有的功能都通过从服务器下载最新的Dex文件完成

## Android动态加载的类型

Android项目中, 动态加载按技术实现上的区别大致可以分为两种:

1. 动态加载 .so 库
2. 动态加载 .dex/jar/apk (现在动态加载普遍说的是这种)

其一, Android中NDK中其实就使用了动态加载, 动态加载 .so 库并通过JNI调用其封装好的方法。后者一般是由C++编译而成, 运行在Native层, 效率会比执行在虚拟机的Java代码高很多, 所以Android中经常通过动态加载 .so 库来完成一些对性能比较有需求的工作 (比如T9搜索、或者Bitmap的解码、图片高斯模糊处理等)。此外, 由于 .so 库是由C++编译而来的, 只能被反编译成汇编代码, 相比Smali更难被破解, 因此 .so 库也可以被用于安全领域。需要特别说明的是, 一般情况下我们是把 .so 库一并打包在APK内部的, 但是 .so 库其实也是可以从外部存储文件加载的。

其二, “基于ClassLoader的动态加载 .dex/jar/apk ”就是我们上面提到的“在Android中动态加载由Java代码编译而来的Dex包并执行其中的代码逻辑”, 这是常规Android开发比较少用到的一种技术, 目前网络上大多文章说到的动态加载指的就是这种 (后面我们谈到“动态加载”如果没有特别指定, 均默认是这种)。

Android项目中, 所有Java代码都会被编译成 Dex 包, Android应用运行时, 就是通过执行 Dex 包里的业务代码逻辑来工作的。使用动态加载技术可以在Android应用运行时加载外部的Dex包, 而通过网络下载新的 Dex 包并替换原有的 Dex 包就可以达到不安装新 APK文件就升级应用 (改变代码逻辑) 的目的。同时, 使用动态加载技术, 一般来说会使得Android开发工作变得更加复杂, 这种开发方式不是官方推荐的, 不是目前主流的Android开发方式, **Github**和**StackOverflow**上面外国的开发者也对此不是很感兴趣, 外国相关的教程更是少得可怜, 目前只有在大天朝才有比较深入的研究和应用, 特别是一些SDK组件项目和BAT家族的项目上, Github上的相关开源项目基本是国人在维护, 偶尔有几个外国人请求更新英文文档。

## Android动态加载的大致过程

无论上面的哪种动态加载，其实基本原理都是在程序运行时加载一些外部的可执行的文件，然后调用这些文件的某个方法执行业务逻辑。需要说明的是，因为文件是可执行的（`.so` 库或者 `dex` 包，也就是一种动态链接库），出于安全问题，Android并不允许直接加载手机外部存储这类 `noexec`（不可执行）存储路径上的可执行文件。

对于这些外部的可执行文件，在Android应用中调用它们前，都要先把他们拷贝到 `data/packagename/` 内部储存文件路径，确保库不会被第三方应用恶意修改或拦截，然后再将他们加载到当前的运行环境并调用需要的方法执行相应的逻辑，从而实现动态调用。

动态加载的大致过程就是：

1. 把可执行文件（`.so/dex/jar/apk`）拷贝到应用APP内部存储
2. 加载可执行文件
3. 调用具体的方法执行业务逻辑

以下分别对这两种动态加载的实现方式做比较深入的介绍。

## 动态加载.so库

动态加载 `.so` 库应该就是Android最早期的动态加载了，不过 `.so` 库不仅可以存放在APK文件内部，还可以存放在外部存储。Android开发中，更换 `.so` 库的情形并不多，但是可以通过把 `.so` 库挪动到APK外部，减少APK的体积，毕竟许多 `.so` 文件的体积可是非常大的。

详细的应用方式请参考后续日志[Android动态加载补充-加载SD卡的SO库](#)

## 动态加载.dex/jar/apk文件

我们经常讲到的那种Android动态加载技术就是这种，后面我们谈到“动态加载”如果没有特别指定，均默认是这。为了方便区分概念，讨论之前先要阐述一些术语。

## 主APK和插件APK

- 主APK：主项目APK、宿主APK（Host APK），也就是我们希望采用动态加载技术的主项目；
- 插件APK：Plugin，从主项目分离开来，我们能通过动态加载加载到主项目里面来的模块，一个主APK可以同时加载多个插件APK；

## 基础知识：`ClassLoader`和Dex

动态加载 .dex/jar/apk 文件的基础是类加载器ClassLoader，它的包路径是 `java.lang`，由此可见其重要性，虚拟机就是通过类加载器加载期需要用的Class，这是Java程序运行的基础。关于类加载器ClassLoader的工作机制，请参考[Android动态加载基础 ClassLoader的工作机制](#)

现在网上有多种基于ClassLoader的Android动态加载的开源项目，大部分核心思想都殊途同归，按照复杂程度以及具体实现的框架，大致可以分为以下三种模式。

## 简单的动态加载模式

Android在运行时使用ClassLoader动态加载外部的Dex文件非常简单，不用覆盖安装新的APK，就可以更改APP的代码逻辑。但是Android却很难使用插件APK里的res资源，这意味着无法使用新的XML布局等资源，同时由于无法更改本地的Manifest清单文件，所以无法启动新的Activity等组件。

不过可以先把要用到的全部res资源都放到主APK里面，同时把所有需要的Activity先全部写进Manifest里，只通过动态加载更新代码，不更新res资源，如果需要改动UI界面，可以通过使用纯Java代码创建布局的方式绕开XML布局，也可以使用Fragment代替Activity。

某种程度上，简单的动态加载功能已经能满足部分业务需求了，特别是一些早期的Android项目，那时候Android的技术还不是很成熟，而且早期的Android设备更是有大量的兼容性问题（做过Android1.6兼容的同学可能深有体会），只有这种简单的加载方式才能稳定运行。这种模式的框架比较适用一些UI变化比较少的项目，比如游戏SDK，基本就只有登陆、注册界面，而且基本不会变动，更新的往往只有代码逻辑。

详细的应用方式请参考后续日志[Andorid动态加载入门-简单加载模式](#)

## 使用代理Activity模式

从这个阶段开始就稍微有点“黑科技”的味道了，比如我们可以通过动态加载，让现在的Android应用启动一些“新”的Activity，甚至不用安装就启动一个“新”的APK（原来的APK叫“主APK”，新的APK称为“插件APK”）。主APK需要先注册一个空壳的Activity用于代理执行插件APK的Activity的生命周期。

主要有以下特点：

1. 主APK可以启动未安装的插件APK；
2. 插件APK也可以作为一个普通APK安装并且启动；
3. 插件APK可以调用主APK里的一些功能；
4. 主APK和插件APK都要接入一套指定的接口才能实现以上功能；

详细的应用方式请参考后续日志[Android动态加载进阶-代理Activity模式](#)

## 动态创建Activity模式

天了噜，到了这个阶段就真的是“黑科技”的领域了，可以试想“从网络下载一个Flappy Bird的APK，不用安装就直接运行游戏”，或者“同时运行两个甚至多个微信”。

这个阶段有以下特点

主APK可以启动一个未安装的插件APK； 插件APK可以是任意第三方APK，无需接入指定的接口； 详细的应用方式请参考后续日志[Android动态加载黑科技-动态创建Activity模式](#)

## 为什么我们要使用动态加载技术

说实话，我也不知道，产品要求的！（警察蜀黍就是他，他只问我能不能实现，并没有问我实现起来难不难...）

Android开发中，最先使用动态加载技术的应该是SDK项目吧。现在网上有一大堆Android SDK项目，比如Google的Goole Play Service，向开发者提供支付、地图等功能，又比如一些Android游戏市场的SDK，用于向游戏开发者提供账号和支付功能。和普通Android应用一样，这些SDK项目也是要升级的，比如现在别人的Android应用里使用了我们的SDK1.0版本，然后发布到安卓市场上去。现在我们发现SDK1.0有一些紧急的BUG，所以升级了一个SDK1.1版本，没办法，只能让人家重新接入1.1版本再发布到市场。万一我们有SDK1.2、1.3等版本呢，本来让人家每个版本都重新接入也无可厚非，不过产品可关心体验啊，他就会问咯，“虽然我不懂技术，但是我想知道有没有办法，能让人家只接入一次我们的SDK，以后我们发布新的SDK版本的时候他们的项目也能跟着自动升级？”，答曰，“有，使用动态加载的技术能办到，只不过（开发工作量会剧增...）”，“那就用吧，我们要把产品的体验做到极致”。

好吧，我并没有黑产品的意思，现在团队的产品也不错，不过与上面类似的对话确实发生在13年我的项目里。这里提出来只是为了强调一下Android项目中采用动态加载技术的作用以及由此带来的代价。

## 作用与代价

凡事都有两面性，特别是这种非官方支持的非常规开发方式，在采用前一定要权衡清楚其作用与代价。如果决定了要采用动态加载技术，个人推荐可以现在实际项目的一些比较独立的模块使用这种框架，把遇到的一些问题解决之后，再慢慢引进到项目的核心模块；如果遇到了一些无法跨越的问题，要有能够迅速投入生产的替代方案。

## 作用

1. 规避APK覆盖安装的升级过程，提高用户体验，顺便能规避一些安卓市场的限制；
2. 动态修复应用的一些紧急BUG，做好最后一道保障；
3. 当应用体积太庞大的时候，可以把一些模块通过动态加载以插件的形式分割出去，这样可以减少主项目的体积，提高安装速度；
4. 插件模块可以用懒加载的方式在需要的时候才初始化，从而提高应用的启动速度；
5. 从项目管理上来看，分割插件模块的方式从项目级别做到了代码分离，大大降低模块之间的耦合度，如果出现BUG，只需要修改对应的插件即可；
6. 在Android应用上推广其他应用的时候，可以使用动态加载技术让用户优先体验新应用的功能，而不用下载并安装新的应用；
7. 减少主项目DEX的方法数，65535问题彻底成为历史（虽然现在在Android Studio中很容易开启MultiDex，但仍然存在一些兼容性问题）；

## 代价

1. 开发方式可能变得比较诡异、繁琐，与常规开发方式不同；
2. 随着动态加载框架复杂程度的加深，项目的构建过程也变得复杂，有可能要主项目和插件项目分别构建，再整合；
3. 由于插件项目是独立开发的，当主项目加载插件运行时，插件的运行环境已经完全不同，代码逻辑容易出现BUG，且不易定位；
4. 非常规的开发方式，有些框架使用反射强行调用了部分Android系统Framework层的代码，部分Android ROM可能无法正常运行；
5. 采用动态加载的插件在使用系统资源（特别是主题）时经常有一些兼容性问题，特别是部分三星的手机；

## 其他动态修改代码的技术

上面说到的都是基于ClassLoader的动态加载技术（除了加载SO库外），使用ClassLoader的一个特点就是，如果程序不重新启动，加载过一次的类就无法重新加载。因此，如果使用ClassLoader来动态升级APP或者动态修复BUG，都需要重新启动APP才能生效。

除了使用ClassLoader外，还可以使用jni hook的方式修改程序的执行代码。前者是在虚拟机上操作的，而后者做的已经是Native层级的工作了，直接修改应用运行时的内存地址，所以使用jni hook的方式时，不用重新应用就能生效。

目前采用jni hook方案的项目中比较热门的有阿里的dexposed和AndFix，有兴趣的同学可以参考[各大热补丁方案分析和比较](#)。

## 动态加载开源项目

[dynamic-load-apk](#)

[android-pluginmgr](#)

[Direct-Load-apk](#)

[360 DroidPlugin](#)

[携程网 DynamicAPK](#)

[Nuwa](#)

# Android动态加载基础 ClassLoader工作机制

来源:<http://segmentfault.com/>

## 类加载器ClassLoader

早期使用过Eclipse等Java编写的软件的同学可能比较熟悉，Eclipse可以加载许多第三方的插件，这其实就是动态加载。这些插件大多是一些Jar包，使用插件其实就是动态加载Jar包里的Class进行工作。

Android的 Dalvik/ART 虚拟机如同标准JAVA的JVM虚拟机一样，在运行程序时首先需要将对应的类加载到内存中。因此，我们常常利用这一点，在程序运行时手动加载Class，从而达到代码动态加载执行的目的。

这其实非常好理解，Java代码都是写在Class里面的，程序运行在虚拟机上时，虚拟机需要把需要的Class加载进来才能创建实例对象并工作，而完成这一个加载工作的角色就是 classLoader 。Android的 Dalvik/ART 虚拟机虽然与标准Java的JVM虚拟机不一样，ClassLoader 具体的加载细节不一样，但是工作机制是类似的，也就是说在Android中同样可以采用类似的动态加载插件的功能，只是在Android应用中动态加载一个插件的工作要比Eclipse加载一个插件复杂许多（毕竟不是常规开发方式）。

## 有几个ClassLoader实例？

动态加载的基础是ClassLoader，从名字也可以看出，ClassLoader就是专门用来处理类加载工作的，所以这货也叫类加载器，而且一个运行中的APP不仅只有一个类加载器。

其实，在Android系统启动的时候会创建一个Boot类型的ClassLoader实例，用于加载一些系统Framework层级需要的类，我们的Android应用里也需要用到一些系统的类，所以APP启动的时候也会把这个Boot类型的ClassLoader传进来。

此外，APP也有自己的类，这些类保存在APK的dex文件里面，所以APP启动的时候，也会创建一个自己的ClassLoader实例，用于加载自己dex文件中的类。下面我们在项目里验证看看

```

@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 ClassLoader classLoader = getClassLoader();
 if (classLoader != null){
 Log.i(TAG, "[onCreate] classLoader " + i + " : " + classLoader.toString());
 while (classLoader.getParent()!=null){
 classLoader = classLoader.getParent();
 Log.i(TAG,"[onCreate] classLoader " + i + " : " + classLoader.toString());
 }
 }
}

```

输出结果为

```

[onCreate] classLoader 1 : dalvik.system.PathClassLoader[DexPathList[[zip file "/data
[onCreate] classLoader 2 : java.lang.BootClassLoader@14af4e32

```

可以看见有2个Classloader实例，一个是BootClassLoader（系统启动的时候创建的），另一个是PathClassLoader（应用启动时创建的，用于加载“/data/app/me.kaede.androidclassloadersample-1/base.apk”里面的类）。由此也可以看出，一个运行的Android应用至少有2个ClassLoader(系统的和应用的)。

## 创建自己ClassLoader实例

动态加载外部的dex文件的时候，我们也可以使用自己创建的ClassLoader实例来加载dex里面的Class，不过ClassLoader的创建方式有点特殊，我们先看看它的构造方法

```

/*
 * constructor for the BootClassLoader which needs parent to be null.
 */
ClassLoader(ClassLoader parentLoader, boolean nullAllowed) {
 if (parentLoader == null && !nullAllowed) {
 throw new NullPointerException("parentLoader == null && !nullAllowed");
 }
 parent = parentLoader;
}

```

创建一个ClassLoader实例的时候，需要使用一个现有的ClassLoader实例作为新创建的实例的Parent。这样一来，一个Android应用，甚至整个Android系统里所有的ClassLoader实例都会被一棵树关联起来，这也是ClassLoader的双亲代理模型（Parent-Delegation Model）的特点。

## ClassLoader双亲代理模型加载类的特点和作用

JVM中ClassLoader通过defineClass方法加载jar里面的Class，而Android中这个方法被弃用了。

```
@Deprecated
protected final Class<?> defineClass(byte[] classRep, int offset, int length)
 throws ClassFormatError {
 throw new UnsupportedOperationException("can't load this type of class file");
}
```

取而代之的是loadClass方法

```
public Class<?> loadClass(String className) throws ClassNotFoundException {
 return loadClass(className, false);
}

/**
 * Loads the class with the specified name, optionally linking it after
 * loading. The following steps are performed:
 *
 * Call {@link #findLoadedClass(String)} to determine if the requested
 * class has already been loaded.
 * If the class has not yet been loaded: Invoke this method on the
 * parent class loader.
 * If the class has still not been loaded: Call
 * {@link #findClass(String)} to find the class.
 *
 * <p>
 * Note: In the Android reference implementation, the
 * {@code resolve} parameter is ignored; classes are never linked.
 * </p>
 *
 * @return the {@code Class} object.
 * @param className
 * the name of the class to look for.
 * @param resolve
 * Indicates if the class should be resolved after loading. This
 * parameter is ignored on the Android reference implementation;
 * classes are not resolved.
 * @throws ClassNotFoundException
```

```

 * if the class can not be found.
 */
protected Class<?> loadClass(String className, boolean resolve) throws ClassNotFoundException {
 Class<?> clazz = findLoadedClass(className);

 if (clazz == null) {
 ClassNotFoundException suppressed = null;
 try {
 clazz = parent.loadClass(className, false);
 } catch (ClassNotFoundException e) {
 suppressed = e;
 }

 if (clazz == null) {
 try {
 clazz = findClass(className);
 } catch (ClassNotFoundException e) {
 e.addSuppressed(suppressed);
 throw e;
 }
 }
 }

 return clazz;
}

```

## 特点

从源码中我们也可以看出，`loadClass`方法在加载一个类的实例的时候，

1. 会先查询当前ClassLoader实例是否加载过此类，有就返回；
2. 如果没有。查询Parent是否已经加载过此类，如果已经加载过，就直接返回Parent加载的类；
3. 如果继承路线上ClassLoader都没有加载，才由Child执行类的加载工作；

这样做有个明显的特点，如果一个类被位于树根的ClassLoader加载过，那么在以后整个系统的生命周期内，这个类永远会被重新加载。

## 作用

首先是**共享功能**，一些Framework层级的类一旦被顶层的ClassLoader加载过就缓存在内存里面，以后任何地方用到都不需要重新加载。

除此之外还有隔离功能，不同继承路线上的ClassLoader加载的类肯定不是同一个类，这样的限制避免了用户自己的代码冒充核心类库的类访问核心类库包可见成员的情况。这也好理解，一些系统层级的类会在系统初始化的时候被加载，比如java.lang.String，如果在一个应用里面能够简单地用自定义的String类把这个系统的String类给替换掉，那将会有严重安全问题。

## 使用ClassLoader一些需要注意的问题

我们都知道，我们可以通过动态加载获得新的类，从而升级一些代码逻辑，这里有几个问题要注意一下。

- 如果你希望通过动态加载的方式，加载一个新版本的dex文件，使用里面的新类替换原有的旧类，从而修复原有类的BUG，那么你必须保证在加载新类的时候，旧类还没有被加载，因为如果已经加载过旧类，那么ClassLoader会一直优先使用旧类。
- 如果旧类总是优先于新类被加载，我们也可以使用一个与加载旧类的ClassLoader没有树的继承关系的另一个ClassLoader来加载新类，因为ClassLoader只会检查其Parent有没有加载过当前要加载的类，如果两个ClassLoader没有继承关系，那么旧类和新类都能被加载。

不过这样一来又有另一个问题了，在Java中，只有当两个实例的类名、包名以及加载其的ClassLoader都相同，才会被认为是同一种类型。上面分别加载的新类和旧类，虽然包名和类名都完全一样，但是由于加载的ClassLoader不同，所以并不是同一种类型，在实际使用中可能会出现类型不符异常。

同一个Class = 相同的 ClassName + PackageName + ClassLoader

以上问题在采用动态加载功能的开发中容易出现，请注意。

## DexClassLoader 和 PathClassLoader

ClassLoader是一个抽象类，实际开发过程中，我们一般是使用其具体的子类DexClassLoader、PathClassLoader这些类加载器来加载类的，它们的不同之处是：

- DexClassLoader 可以加载jar/apk/dex中的类，可以从SD卡中加载未安装的apk中的类；
- PathClassLoader 只能加载系统中已经安装过的apk中的类；

## 类加载器的初始化

平时开发的时候，使用DexClassLoader就够用了，但是我们不妨挖一下这两者具体细节上的区别。

```
// DexClassLoader.java
public class DexClassLoader extends BaseDexClassLoader {
 public DexClassLoader(String dexPath, String optimizedDirectory,
 String libraryPath, ClassLoader parent) {
 super(dexPath, new File(optimizedDirectory), libraryPath, parent);
 }
}

// PathClassLoader.java
public class PathClassLoader extends BaseDexClassLoader {
 public PathClassLoader(String dexPath, ClassLoader parent) {
 super(dexPath, null, null, parent);
 }

 public PathClassLoader(String dexPath, String libraryPath,
 ClassLoader parent) {
 super(dexPath, null, libraryPath, parent);
 }
}
```

这两者只是简单的对BaseDexClassLoader做了一下封装，具体的实现还是在父类里。不过这里也可以看出，PathClassLoader的 optimizedDirectory 只能是null，进去BaseDexClassLoader看看这个参数是干什么的

```
public BaseDexClassLoader(String dexPath, File optimizedDirectory,
 String libraryPath, ClassLoader parent) {
 super(parent);
 this.originalPath = dexPath;
 this.pathList = new DexPathList(this, dexPath, libraryPath, optimizedDirectory);
}
```

这里创建了一个DexPathList实例，进去看看

```
public DexPathList(ClassLoader definingContext, String dexPath,
 String libraryPath, File optimizedDirectory) {

 this.dexElements = makeDexElements(splitDexPath(dexPath), optimizedDirectory);
}

private static Element[] makeDexElements(ArrayList<File> files,
 File optimizedDirectory) {
 ArrayList<Element> elements = new ArrayList<Element>();
 for (File file : files) {
```

```

 ZipFile zip = null;
 DexFile dex = null;
 String name = file.getName();
 if (name.endsWith(DEX_SUFFIX)) {
 dex = loadDexFile(file, optimizedDirectory);
 } else if (name.endsWith(APK_SUFFIX) || name.endsWith(JAR_SUFFIX)
 || name.endsWith(ZIP_SUFFIX)) {
 zip = new ZipFile(file);
 }

 if ((zip != null) || (dex != null)) {
 elements.add(new Element(file, zip, dex));
 }
 }
 return elements.toArray(new Element[elements.size()]);
}

private static DexFile loadDexFile(File file, File optimizedDirectory) throws IOException {
 if(optimizedDirectory == null) {
 return new DexFile(file);
 } else {
 String optimizedPath = optimizedPathFor(file, optimizedDirectory);
 return DexFile.loadDex(file.getPath(), optimizedPath, 0);
 }
}

/**
 * Converts a dex/jar file path and an output directory to an output file path for an
 */
private static String optimizedPathFor(File path, File optimizedDirectory) {
 String fileName = path.getName();
 if(! fileName.endsWith(DEX_SUFFIX)) {
 int lastDot = fileName.lastIndexOf(".");
 if(lastDot < 0) {
 fileName += DEX_SUFFIX;
 } else {
 StringBuilder sb = new StringBuilder(lastDot + 4);
 sb.append(fileName, 0, lastDot);
 sb.append(DEX_SUFFIX);
 fileName = sb.toString();
 }
 }
 File result = new File(optimizedDirectory, fileName);
 return result.getPath();
}

```

看到这里我们明白了，optimizedDirectory是用来缓存我们需要加载的dex文件的，并创建一个DexFile对象，如果它为null，那么会直接使用dex文件原有的路径来创建DexFile对象。

`optimizedDirectory` 必须是一个内部存储路径，还记得我们之前说过的，无论哪种动态加载，加载的可执行文件一定要存放在内部存储。`DexClassLoader`可以指定自己的 `optimizedDirectory`，所以它可以加载外部的dex，因为这个dex会被复制到内部路径的 `optimizedDirectory`；而`PathClassLoader`没有`optimizedDirectory`，所以它只能加载内部的dex，这些大都是存在系统中已经安装过的apk里面的。

## 加载类的过程

上面还只是创建了类加载器的实例，其中创建了一个`DexFile`实例，用来保存dex文件，我们猜想这个实例就是用来加载类的。

Android中，`ClassLoader`用`loadClass`方法来加载我们需要的类

```
public Class<?> loadClass(String className) throws ClassNotFoundException {
 return loadClass(className, false);
}

protected Class<?> loadClass(String className, boolean resolve) throws ClassNotFoundException {
 Class<?> clazz = findLoadedClass(className);

 if (clazz == null) {
 ClassNotFoundException suppressed = null;
 try {
 clazz = parent.loadClass(className, false);
 } catch (ClassNotFoundException e) {
 suppressed = e;
 }

 if (clazz == null) {
 try {
 clazz = findClass(className);
 } catch (ClassNotFoundException e) {
 e.addSuppressed(suppressed);
 throw e;
 }
 }
 }

 return clazz;
}
```

`loadClass`方法调用了`findClass`方法，而`BaseDexClassLoader`重载了这个方法，得到`BaseDexClassLoader`看看

```

@Override
protected Class<?> findClass(String name) throws ClassNotFoundException {
 Class clazz = pathList.findClass(name);
 if (clazz == null) {
 throw new ClassNotFoundException(name);
 }
 return clazz;
}

```

结果还是调用了DexPathList的findClass

```

public Class findClass(String name) {
 for (Element element : dexElements) {
 DexFile dex = element.dexFile;
 if (dex != null) {
 Class clazz = dex.loadClassBinaryName(name, definingContext);
 if (clazz != null) {
 return clazz;
 }
 }
 }
 return null;
}

```

这里遍历了之前所有的DexFile实例，其实也就是遍历了所有加载过的dex文件，再调用loadClassBinaryName方法一个个尝试能不能加载想要的类，真是简单粗暴

```

public Class loadClassBinaryName(String name, ClassLoader loader) {
 return defineClass(name, loader, mCookie);
}
private native static Class defineClass(String name, ClassLoader loader, int cookie);

```

看到这里想必大家都明白了，loadClassBinaryName中调用了Native方法defineClass加载类。

至此，ClassLoader的创建和加载类的过程的完成了。有趣的是，标准JVM中，ClassLoader是用defineClass加载类的，而Android中defineClass被弃用了，改用了loadClass方法，而且加载类的过程也挪到了DexFile中，在DexFile中加载类的具体方法也叫defineClass，不知道是Google故意写成这样的还是巧合。

## 自定义ClassLoader

平时进行动态加载开发的时候，使用DexClassLoader就够了。但我们也就可以创建自己的类去继承ClassLoader，需要注意的是loadClass方法并不是final类型的，所以我们可以重载loadClass方法并改写类的加载逻辑。

通过前面我们分析知道，ClassLoader双亲代理的实现很大一部分就是在loadClass方法里，我们可以通过重写loadClass方法避开双亲代理的框架，这样一来就可以在重新加载已经加载过的类，也可以在加载类的时候注入一些代码。这是一种Hack的开发方式，采用这种开发方式的程序稳定性可能比较差，但是却可以实现一些“黑科技”的功能。

# Android动态加载补充 加载SD卡中的SO库

来源:[segmentfault.com](https://segmentfault.com)

## JNI与NDK

Android中JNI的使用其实就包含了动态加载，APP运行时动态加载 .so 库并通过JNI调用其封装好的方法。后者一般是使用NDK工具从C/C++代码编译而成，运行在Native层，效率会比执行在虚拟机的Java代码高很多，所以Android中经常通过动态加载 .so 库来完成一些对性能比较有需求的工作（比如T9搜索、或者Bitmap的解码、图片高斯模糊处理等）。此外，由于 .so 库是由C++编译而来的，只能被反编译成汇编代码，相比Smali更难被破解，因此 .so 库也可以被用于安全领域。

与我们常说的基于ClassLoader的动态加载不同，SO库的加载是使用System类的（由此可见对SO库的支持也是Android的基础功能），所以这里这是作为补充说明。不过，如果使用ClassLoader加载SD卡里插件APK，而插件APK里面包含有SO库，这就涉及到了对插件APK里的SO库的加载，所以我们也要知道如何加载SD卡里面的SO库。

## 一般的SO文件的使用姿势

以一个“图片高斯模糊”的功能为例，如果使用Java代码对图像Bitmap的每一个像素点进行计算，那整体耗时将会非常大，所以可以考虑使用JNI。（详细的JNI使用教程网络上有许多，这里不赘述）

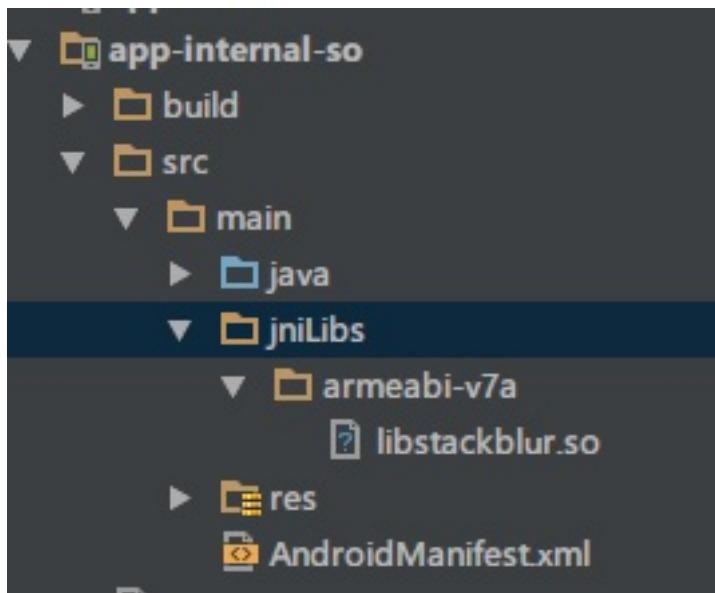
这里推荐一个开源的高斯模糊项目 [Android StackBlur](#)

在命令行定位到Android.mk文件所在目录，运行NDK工具的 `ndk-build` 命令就能编译出我们需要SO库

```
F:\Kaede\GitHub\android-dynamical-loading\dynamic-load-so\AndroidJNISample\jni>ndk-build
[armeabi-v7a] Compile thumb : stackblur <= stackblur.c
[armeabi-v7a] SharedLibrary : libstackblur.so
[armeabi-v7a] Install : libstackblur.so => libs/armeabi-v7a/libstackblur.so

F:\Kaede\GitHub\android-dynamical-loading\dynamic-load-so\AndroidJNISample\jni>
```

再把SO库复制到Android Studio项目的jniLibs目录中



(Android Studio现在也支持直接编译SO库，但是有许多坑，这里我选择手动编译)

接着在Java中把SO库对应的模块加载进来

```
// load so file from internal directory
try {
 System.loadLibrary("stackblur");
 NativeBlurProcess.isLoadLibraryOk.set(true);
 Log.i("MainActivity", "loadLibrary success!");
} catch (Throwable throwable) {
 Log.i("MainActivity", "loadLibrary error!" + throwable);
}
```

加载成功后就可以直接使用Native方法了

```
public class NativeBlurProcess {
 public static AtomicBoolean isLoadLibraryOk = new AtomicBoolean(false);
 //native method
 private static native void functionToBlur(Bitmap bitmapOut, int radius, int threadCount);
}
```

由此可见，在Android项目中，SO库的使用也是一种动态加载，在运行时把可执行文件加载进来。一般情况下，SO库都是打包在APK内部的，不允许修改。这种“动态加载”看起来不是我们熟悉的那种啊，貌似没什么卵用。不过，其实SO库也是可以存放在外部存储路径的。

## 如何把SO文件存放在外部存储

注意到上面加载SO库的时候我们用到了System类的“loadLibrary”方法，同时我们也发现System类还有一个“load”方法，看起来差不多啊，看看他们有什么区别吧！

```
/**
 * See {@link Runtime#load}.
 */
public static void load(String pathName) {
 Runtime.getRuntime().load(pathName, VMStack.getCallingClassLoader());
}

/**
 * See {@link Runtime#loadLibrary}.
 */
public static void loadLibrary(String libName) {
 Runtime.getRuntime().loadLibrary(libName, VMStack.getCallingClassLoader());
}
```

先看看loadLibrary，这里调用了Runtime的loadLibrary，进去一看，又是动态加载熟悉的ClassLoader了（这里也佐证了SO库的使用就是一种动态加载的说法）

```
public void loadLibrary(String nickname) {
 loadLibrary(nickname, VMStack.getCallingClassLoader());
}

/*
 * Searches for and loads the given shared library using the given ClassLoader.
 */
void loadLibrary(String libraryName, ClassLoader loader) {
 if (loader != null) {
 String filename = loader.findLibrary(libraryName);
 if (filename == null) {
 // It's not necessarily true that the ClassLoader used
 // System.mapLibraryName, but the default setup does, and it's
 // misleading to say we didn't find "libMyLibrary.so" when we
 // actually searched for "liblibMyLibrary.so.so".
 throw new UnsatisfiedLinkError(loader + " couldn't find '" +
 System.mapLibraryName(libraryName) + "'");
 }
 String error = doLoad(filename, loader);
 if (error != null) {
 throw new UnsatisfiedLinkError(error);
 }
 return;
 }
 ...
}
```

看样子就像是通过库名获取一个文件路径，再调用“doLoad”方法加载这个文件，先看看 `loader.findLibrary(libraryName)`

```
protected String findLibrary(String libName) {
 return null;
}
```

ClassLoader只是一个抽象类，它的大部分工作都在BaseDexClassLoader类中实现，进去看看

```
public class BaseDexClassLoader extends ClassLoader {
 public String findLibrary(String name) {
 throw new RuntimeException("Stub!");
 }
}
```

不对啊，这里只是抛了一个 `RuntimeException` 异常，什么都没做啊！

其实这里有一个误区，也是刚开始开Android SDK源码的同学容易搞混的。Android SDK自带的源码其实只是给我们开发者参考的，基本只是一些常用的类，Google不会把整个Android系统的源码都放到这里来，因为整个项目非常大，ClassLoader类平时我们接触得少，所以它的具体实现的源码并没有打包进SDK里，如果需要，我们要到官方AOSP项目里面去看（顺便一提，整个AOSP5.1项目大小超过150GB，真的有需要的话推荐用一个移动硬盘存储）。

这里为了方便，我们可以直接看在线的代码 [BaseDexClassLoader.java](#)

```
@Override
public String findLibrary(String name) {
 return pathList.findLibrary(name);
}
```

再看进去DexPathList类

```

/**
 * Finds the named native code library on any of the library
 * directories pointed at by this instance. This will find the
 * one in the earliest listed directory, ignoring any that are not
 * readable regular files.
 *
 * @return the complete path to the library or {@code null} if no
 * library was found
 */
public String findLibrary(String libName) {
 String fileName = System.mapLibraryName(libName);
 for (File directory : nativeLibraryDirectories) {
 File file = new File(directory, fileName);
 if (file.exists() && file.isFile() && file.canRead()) {
 return file.getPath();
 }
 }
 return null;
}

```

到这里已经明朗了，根据传进来的libName，扫描APK内部的nativeLibrary目录，获取并返回内部SO库文件的完整路径filename。再回到Runtime类，获取filename后调用了“doLoad”方法，看看

```

private String doLoad(String name, ClassLoader loader) {
 String ldLibraryPath = null;
 String dexPath = null;
 if (loader == null) {
 ldLibraryPath = System.getProperty("java.library.path");
 } else if (loader instanceof BaseDexClassLoader) {
 BaseDexClassLoader dexClassLoader = (BaseDexClassLoader) loader;
 ldLibraryPath = dexClassLoader.getLdLibraryPath();
 }
 synchronized (this) {
 return nativeLoad(name, loader, ldLibraryPath);
 }
}

```

到这里就彻底清楚了，调用Native方法“nativeLoad”，通过完整的SO库路径filename，把目标SO库加载进来。

说了半天还没有进入正题呢，不过我们可以想到，如果使用loadLibrary方法，到最后还是要找到目标SO库的完整路径，再把SO库加载进来，那我们能不能一开始就给出SO库的完整路径，然后直接加载进来？我们猜想load方法就是干这个的，看看。

```
void load(String absolutePath, ClassLoader loader) {
 if (absolutePath == null) {
 throw new NullPointerException("absolutePath == null");
 }
 String error = doLoad(absolutePath, loader);
 if (error != null) {
 throw new UnsatisfiedLinkError(error);
 }
}
```

我勒个去，一上来就直接来到doLoad方法了，这证明我们的猜想可能是正确的，那么在实际项目中测试看看吧！

我们先把SO放在Asset里，然后再复制到内部存储，再使用load方法把其加载进来。

```

public class MainActivity extends AppCompatActivity {
 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);

 File dir = this.getDir("jniLibs", Activity.MODE_PRIVATE);
 File distFile = new File(dir.getAbsolutePath() + File.separator + "libstackblu

 if (copyFileFromAssets(this, "libstackblur.so", distFile.getAbsolutePath())){
 //使用load方法加载内部储存的SO库
 System.load(distFile.getAbsolutePath());
 NativeBlurProcess.isLoadLibraryOk.set(true);
 }
 }

 public void onDoBlur(View view){
 ImageView imageView = (ImageView) findViewById(R.id.iv_app);
 Bitmap bitmap = BitmapFactory.decodeResource(getResources(), android.R.drawable.ic_menu_gallery);
 Bitmap blur = NativeBlurProcess.blur(bitmap,20,false);
 imageView.setImageBitmap(blur);
 }

 public static boolean copyFileFromAssets(Context context, String fileName, String path) {
 boolean copyIsFinish = false;
 try {
 InputStream is = context.getAssets().open(fileName);
 File file = new File(path);
 file.createNewFile();
 FileOutputStream fos = new FileOutputStream(file);
 byte[] temp = new byte[1024];
 int i = 0;
 while ((i = is.read(temp)) > 0) {
 fos.write(temp, 0, i);
 }
 fos.close();
 is.close();
 copyIsFinish = true;
 } catch (IOException e) {
 e.printStackTrace();
 Log.e("MainActivity", "[copyFileFromAssets] IOException "+e.toString());
 }
 return copyIsFinish;
 }
}

```

点击onDoBlur按钮，果然加载成功了！



DO BLUR

那能不能直接加载外部存储上面的SO库呢，把SO库拷贝到SD卡上面试试。

```
e: FATAL EXCEPTION: main
Process: me.kaede.androidjnisan, PID: 3307
java.lang.UnsatisfiedLinkError: dlopen failed: couldn't map "/storage/emulated/0/libstackblur.so" segment 1: Permission denied
 at java.lang.Runtime.load(Runtime.java:331)
 at java.lang.System.load(System.java:982)
 at me.kaede.androidjnisan.MainActivity.onCreate(MainActivity.java:27)
 at android.app.Activity.performCreate(Activity.java:5985)
 at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1108)
```

看起来是不可以的样子，Permission denied！

```
java.lang.UnsatisfiedLinkError: dlopen failed: couldn't map
"/storage/emulated/0/libstackblur.so" segment 1: Permission denied
```

看起来像是没有权限的样子，看看源码哪里抛出的异常吧

```
/*
 * Loads the given shared library using the given ClassLoader.
 */
void load(String absolutePath, ClassLoader loader) {
 if (absolutePath == null) {
 throw new NullPointerException("absolutePath == null");
 }
 String error = doLoad(absolutePath, loader);
 if (error != null) {
 // 这里抛出的异常
 throw new UnsatisfiedLinkError(error);
 }
}
```

应该是执行doLoad方法时出现了错误，但是上面也看过了，doLoad方法里调用了Native方法 nativeLoad，那应该就是Native代码里出现的错误。平时我很少看到Native里面，上一次看的时候，是因为需要看看点九图NinePathDrawable的缩放控制信息chunk数组的具体作用是怎么样，费了好久才找到我想要的一小段代码。所以这里就暂时不跟进去了，有兴趣的同学可以告诉我关键代码的位置。

我在一个Google的开发者论坛上找到了一些答案

The SD Card is mounted noexec, so I'm not sure this will work.  
Moreover, using the SD Card as a storage location is a really bad idea, since any other application can modify/delete/corrupt it easily. Try downloading the library to your application's data directory instead, and load it from here.

这也容易理解，SD卡等外部存储路径是一种可拆卸的（mounted）不可执行（noexec）的储存媒介，不能直接用来作为可执行文件的运行目录，使用前应该把可执行文件复制到APP内部存储再运行。

最后，我们也可以看看官方的API文档

**public void load (String absolutePath)**

Added in API level 1

Loads the shared library found at the given absolute path. This should be of the form `/path/to/library/libMyLibrary.so`. Most callers should use `loadLibrary(String)` instead, and let the system find the correct file to load.

**Throws**

`UnsatisfiedLinkError` if the library can not be loaded, either because it's not found or because there is something wrong with it.

**public void loadLibrary (String nickname)**

Added in API level 1

Loads a shared library. Class loaders have some influence over this process, but for a typical Android app, it works as follows:

Given the name `"MyLibrary"`, that string will be passed to `mapLibraryName(String)`. That means it would be a mistake for the caller to include the usual `"lib"` prefix and `".so"` suffix.

That file will then be searched for on the application's native library search path. This consists of the application's own native library directory followed by the system's native library directories.

**Throws**

`UnsatisfiedLinkError` if the library can not be loaded, either because it's not found or because there is something wrong with it.

看来load方法的用途和我们理解的一致，文档里说的shared library就是指SO库（shared object），至此，我们就可以把SO文件移动到外部存储了，或者从网络下载都行。

# Android动态加载入门 简单加载模式

来源:[segmentfault.com](https://segmentfault.com)

Java程序中，JVM虚拟机是通过类加载器ClassLoader加载.jar文件里面的类的。Android也类似，不过Android用的是Dalvik/ART虚拟机，不是JVM，也不能直接加载.jar文件，而是加载dex文件。

先要通过Android SDK提供的DX工具把.jar文件优化成.dex文件，然后Android的虚拟机才能加载。注意，有的Android应用能直接加载.jar文件，那是因为这个.jar文件已经经过优化，只不过后缀名没改（其实已经是.dex文件）。

如果对ClassLoader的工作机制有兴趣，具体过程请参考[ClassLoader工作机制](#)，这里不再赘述。

## 如何获取能够加载的.dex文件

首先我们可以通过JDK的编译命令javac把Java代码编译成.class文件，再使用jar命令把.class文件封装成.jar文件，这与编译普通Java程序的时候完全一样。

之后再用Android SDK的DX工具把.jar文件优化成.dex文件（在 `android-sdk\build-tools\具体版本\` 路径下）

```
dx --dex --output=target.dex origin.jar // target.dex就是我们要的了
```

此外，我们可以先把代码编译成APK文件，再把APK里面的.dex文件解压出来，或者直接把APK文件当成.dex使用（只是APK里面的静态资源文件我们暂时还用不到）。至此我们发现，无论加载.jar，还是.apk，其实都和加载.dex是等价的，Android能加载.jar和.apk，是因为它们都包含有.dex，直接加载.apk文件时，ClassLoader也会自动把.apk里的.dex解压出来。

## 加载并调用.dex里面的方法

与JVM不同，Android的虚拟机不能用ClassCload直接加载.dex，而是要用DexClassLoader或者PathClassLoader，他们都是ClassLoader的子类，这两者的区别是

- DexClassLoader：可以加载 `jar/apk/dex`，可以从SD卡中加载未安装的apk；
- PathClassLoader：要传入系统中apk的存放Path，所以只能加载已经安装的apk文

件； 使用前，先看看DexClassLoader的构造方法

```
public DexClassLoader(String dexPath, String optimizedDirectory, String libraryPath,
 super((String)null, (File)null, (String)null, (ClassLoader)null);
 throw new RuntimeException("Stub!");
}
```

注意，我们之前提到的，DexClassLoader并不能直接加载外部存储的.dex文件，而是要先拷贝到内部存储里。这里的dexPath就是.dex的外部存储路径，而optimizedDirectory则是内部路径，libraryPath用null即可，parent则是要传入当前应用的ClassLoader，这与ClassLoader的“双亲代理模式”有关。

实例使用DexClassLoader的代码

```
File optimizedDexOutputPath = new File(
 Environment.getExternalStorageDirectory().getAbsolutePath()
 + File.separator + "test_dexloader.jar");// 外部路径
// 无法直接从外部路径加载.dex文件，需要指定APP内部路径作为缓存目录（.dex文件会被解压到此目录）
File dexOutputDir = this.getDir("dex", 0);
DexClassLoader dexClassLoader = new DexClassLoader(
 optimizedDexOutputPath.getAbsolutePath(),
 dexOutputDir.getAbsolutePath(),
 null,
 getClassLoader());
```

到这里，我们已经成功把.dex文件给加载进来了，接下来就是如何调用.dex里面的代码，主要有两种方式。

## 使用反射的方式

使用DexClassLoader加载进来的类，我们本地并没有这些类的源码，所以无法直接调用，不过可以通过反射的方法调用，简单粗暴。

```

DexClassLoader dexClassLoader = new DexClassLoader(
 optimizedDexOutputPath.getAbsolutePath(),
 dexOutputDir.getAbsolutePath(),
 null,
 getClassLoader());
Class libProviderClazz = null;
try {
 libProviderClazz = dexClassLoader.loadClass("me.kaede.dexclassloader.MyLoader");
 // 遍历类里所有方法
 Method[] methods = libProviderClazz.getDeclaredMethods();
 for (int i = 0; i < methods.length; i++) {
 Log.e(TAG, methods[i].toString());
 }
 Method start = libProviderClazz.getDeclaredMethod("func");// 获取方法
 start.setAccessible(true);// 把方法设为public, 让外部可以调用
 String string = (String) start.invoke(libProviderClazz.newInstance());// 调用方法并
 Toast.makeText(this, string, Toast.LENGTH_LONG).show();
} catch (Exception exception) {
 // Handle exception gracefully here.
 exception.printStackTrace();
}

```

## 使用接口的方式

毕竟.dex文件也是我们自己维护的，所以可以把方法抽象成公共接口，把这些接口也复制到主项目里面去，就可以通过这些接口调用动态加载得到的实例的方法了。

```

public interface IFunc{
 public String func();
}

// 调用
IFunc ifunc = (IFunc)libProviderClazz;
String string = ifunc.func();
Toast.makeText(this, string, Toast.LENGTH_LONG).show();

```

到这里，我们已经成功从外部路径动态加载一个.dex文件，并执行里面的代码逻辑了。通过从服务器下载最新的.dex文件并替换本地的旧文件，就能初步实现“APP的动态升级了”。

## 如何动态更改XML布局

虽然已经能动态更改代码逻辑了，但是UI界面要怎么更改啊？Android开发中大部分的情况下，UI界面都是通过XML布局实现的，放在res目录下，可是.dex库里面并没有这些静态资源啊，所以无法改变XML布局。（这里即使直接动态加载APK文件，但是通过DexClassLoader只能加载新的APK其中的.dex文件，并无法加载其中的res资源文件，所以如果在动态加载的.dex中直接使用新的APK的res资源的话会抛出异常。）

大家都知道，所有的XML布局在运行的时候都要通过LayoutInflater渲染成View的实例，这个实例与我们使用纯Java代码创建的View实例几乎是等价的，而且后者可能效率还更高，所有的XML布局实现的UI界面都有等价的纯代码的创建方案。由此伸展开来，res目录下所有XML资源都有等价的纯代码的实现方式，比如XML动画、XML Drawable等。

所以，如果想要动态更改应用的UI界面的话，可以通过用纯代码创建布局的形式来解决。此外，还可以模仿LayoutInflater的工作方式，自己写一套布局解析器来解析XML文件，这样就能在完全不依赖res资源的情况下创建UI界面了，当然这样的工作量不少，而且，完全避开res资源的话，所有的分辨率、国际化等自适应问题都要自己在应用层写代码维护了，显然脱离res资源框架不是一个很明智的做法，但是这种做法确实可行，在我们之前的实际生产中的项目中也稳定使用着，这里出于责任问题就不方便公开细节了。

（说实在，这种方案非常繁琐，不好维护，一方面，这是产品一句“技术可行就做呗”而产生的解决方案；另一方面，但是动态加载技术还很不成熟，也没有什么实际投入到生产的项目，所以采取了非常保守的开发方式）。

## 使用Fragment代替Activity

Activity需要在Manifest里注册，然后一标准的Intent启动才会具有生命周期，很明显，如果想要动态加载的.dex里的Activity没有注册的话，是无法启动的。

有一种简单粗暴的做法就是可以把.dex里所有需要用到的Activity都事先注册到原项目里，不过这样一来如果.dex里的Activity有变化，原项目就必须跟着升级。

另外一种方案是使用Fragment，Fragment自带生命周期，不需要在Manifest里注册，所以可以在.dex里使用Fragment来代替Activity，代价就是Fragment之间的切换会繁琐许多。

## ART模式的兼容性问题

当初我们开始设计动态加载方案的时候，还没有ART模式。随着Kitkat的发布以及ART模式的出现，我们开始担心“用DexClassLoader加载.dex文件”的方案会不会在ART模式上面存在兼容性问题。

其实，ART模式相比原来的Dalvik，会在安装APK的时候，使用Android系统自带的dex2oat工具把APK里面的.dex文件转化成OAT文件，OAT文件是一种Android私有ELF文件格式，它不仅包含有从DEX文件翻译而来的本地机器指令，还包含有原来的DEX文件内容。这使得我们无需重新编译原有的APK就可以让它正常地在ART里面运行，也就是我们不需要改变原来的APK编程接口。ART模式的系统里，同样存在DexClassLoader类，包名路径也没变，只不过它的具体实现与原来的有所不同，但是接口是一致的。

```
package dalvik.system;

import dalvik.system.BaseDexClassLoader;
import java.io.File;

public class DexClassLoader extends BaseDexClassLoader {
 public DexClassLoader(String dexPath, String optimizedDirectory, String libraryPath,
 super((String)null, (File)null, (String)null, (ClassLoader)null);
 throw new RuntimeException("Stub!");
 }
}
```

也就是说，ART模式在加载.dex文件的方法上，对Dalvik做了向下兼容，所以使用DexClassLoader加载进来的.dex文件同样也会被转化成OAT文件再被执行，“以DexClassLoader为核心的动态加载方案”在ART模式上可以稳定运行。

关于ART模式以及OAT文件的详细分析，请参考官方的[ART and Dalvik](#)，以及老罗的[Android ART运行时无缝替换Dalvik虚拟机的过程分析](#)。

## 存在的问题与改进方案

以上大致就是“Android动态性加载初级阶段”的解决方案，虽然现在已经能投入到具体的生产中去，但是还有一些问题无法忽略。

- 无法使用res目录下的资源，特别是使用XML布局，以及无法通过res资源到达自适应
- 无法动态加载新的Activity等组件，因为这些组件需要在Manifest中注册，动态加载无法更改当前APK的Manifest

以上问题可以通过反射调用Framework层代码以及代理Activity的方式解决，可以把这种的动态加载框架成为“代理模式”。

## 参考日志

- <http://44289533.iteye.com/blog/1954453>
- <http://blog.csdn.net/bboyfeiyu/article/details/11710497>
- <http://www.cnblogs.com/over140/archive/2011/11/23/2259367.html>

# Android动态加载进阶 代理Activity模式

来源:[segmentfault.com](https://segmentfault.com)

## 技术背景

简单模式中，使用ClassLoader加载外部的Dex或Apk文件，可以加载一些本地APP不存在的类，从而执行一些新的代码逻辑。但是使用这种方法却不能直接启动插件里的Activity。

## 启动没有注册的Activity的两个主要问题

Activity等组件是需要在Manifest中注册后才能以标准Intent的方式启动的（如果有兴趣强烈推荐你了解下Activity生命周期实现的机制及源码），通过ClassLoader加载并实例化的Activity实例只是一个普通的Java对象，能调用对象的方法，但是它没有生命周期，而且Activity等系统组件是需要Android的上下文环境的（Context等资源），没有这些东西Activity根本无法工作。

使用插件APK里的Activity需要解决两个问题：

- 如何使插件APK里的Activity具有生命周期；
- 如何使插件APK里的Activity具有上下文环境（使用R资源）；

代理Activity模式为解决这两个问题提供了一种思路。

## 代理Activity模式

这种模式也是我们项目中，继“简单动态加载模式”之后，第二种投入实际生产项目的开发方式。

其主要特点是：主项目APK注册一个代理Activity（命名为ProxyActivity），ProxyActivity是一个普通的Activity，但只是一个空壳，自身并没有什么业务逻辑。每次打开插件APK里的某一个Activity的时候，都是在主项目里使用标准的方式启动ProxyActivity，再在ProxyActivity的生命周期里同步调用插件中的Activity实例的生命周期方法，从而执行插件APK的业务逻辑。

ProxyActivity + 没注册的Activity = 标准的Activity

下面谈谈代理模式是怎么处理上面提到的两个问题的。

## 处理插件Activity的生命周期

目前还真的没什么办法能够处理这个问题，一个Activity的启动，如果不采用标准的Intent方式，没有经历过Android系统Framework层级的一系列初始化和注册过程，它的生命周期方法是不会被系统调用的（除非你能够修改Android系统的一些代码，而这已经是另一个领域的话题了，这里不展开）。

那把插件APK里所有Activity都注册到主项目的Manifest里，再以标准Intent方式启动。但是事先主项目并不知道插件Activity里会新增哪些Activity，如果每次有新加的Activity都需要升级主项目的版本，那不是本末倒置了，不如把插件的逻辑直接写到主项目里来得方便。

那就绕绕弯吧，生命周期不就是系统对Activity一些特定方法的调用嘛，那我们可以在主项目里创建一个ProxyActivity，再由它去代理调用插件Activity的生命周期方法（这也是代理模式叫法的由来）。用ProxyActivity（一个标准的Activity实例）的生命周期同步控制插件Activity（普通类的实例）的生命周期，同步的方式可以有下面两种：

- 在ProxyActivity生命周期里用反射调用插件Activity相应生命周期的方法，简单粗暴。
- 把插件Activity的生命周期抽象成接口，在ProxyActivity的生命周期里调用。另外，多了这一层接口，也方便主项目控制插件Activity。

这里补充说明下，Fragment自带生命周期，用Fragment来代替Activity开发可以省去大部分生命周期的控制工作，但是会使得界面跳转比较麻烦，而且Honeycomb以前没有Fragment，无法在API11以前的系统使用。

## 在插件Activity里使用R资源

使用代理的方式同步调用生命周期的做法容易理解，也没什么问题，但是要使用插件里面的res资源就有点麻烦了。简单的说，res里的每一个资源都会在R.java里生成一个对应的Integer类型的id，APP启动时会先把R.java注册到当前的上下文环境，我们在代码里以R文件的方式使用资源时正是通过使用这些id访问res资源，然而插件的R.java并没有注册到当前的上下文环境，所以插件的res资源也就无法通过id使用了。

这个问题困扰了我们很久，一开始的项目急于投入生产，所以我们索性抛开res资源，插件里需要用到的新资源都通过纯Java代码的方式创建（包括XML布局、动画、点九图等），笨重但有效。知道网上出现了解决这一个问题的有效方法（一开始貌似是在手机QQ项目中出现的，但是没有开源所以不清楚，在这里真的佩服这些对技术这么有追求的开发者）。

记得我们平时怎么使用res资源的吗，就是 `getResources().getXXX(resid)`，看看 `getResources()`

```
@Override
public Resources getResources() {
 if (mResources != null) {
 return mResources;
 }
 if (mOverrideConfiguration == null) {
 mResources = super.getResources();
 return mResources;
 } else {
 Context resc = createConfigurationContext(mOverrideConfiguration);
 mResources = resc.getResources();
 return mResources;
 }
}
```

看起来像是通过mResources实例获取res资源的，在找找mResources实例是怎么初始化的，看看上面的代码发现是使用了super类ContextThemeWrapper里的“`getResources()`”方法，看进去

```
Context mBase;
public ContextWrapper(Context base) {
 mBase = base;
}

@Override
public Resources getResources(){
 return mBase.getResources();
}
```

看样子又调用了Context的“`getResources()`”方法，看到这里，我们知道Context只是个抽象类，其实际工作都是在ContextImpl完成的，赶紧去ContextImpl里看看“`getResources()`”方法吧

```
@Override
public Resources getResources() {
 return mResources;
}
```

你TM在逗我么，还是没有mResources的创建过程啊！啊，不对，mResources是ContextImpl的成员变量，可能是在构造方法中创建的，赶紧去看看构造方法（这里只给出关键代码）。

```

resources = mResourcesManager.getTopLevelResources(packageInfo.getResDir(),
 packageInfo.getSplitResDirs(), packageInfo.getOverlayDirs(),
 packageInfo.getApplicationInfo().sharedLibraryFiles, displayId,
 overrideConfiguration, compatInfo);
mResources = resources;

```

看样子是在ResourcesManager的“getTopLevelResources”方法中创建的，看进去

```

Resources getTopLevelResources(String resDir, String[] splitResDirs,
 String[] overlayDirs, String[] libDirs, int displayId,
 Configuration overrideConfiguration, CompatibilityInfo compatInfo) {
 Resources r;
 AssetManager assets = new AssetManager();
 if (libDirs != null) {
 for (String libDir : libDirs) {
 if (libDir.endsWith(".apk")) {
 if (assets.addAssetPath(libDir) == 0) {
 Log.w(TAG, "Asset path '" + libDir +
 "' does not exist or contains no resources.");
 }
 }
 }
 }
 DisplayMetrics dm = getDisplayMetricsLocked(displayId);
 Configuration config;
 r = new Resources(assets, dm, config, compatInfo);
 return r;
}

```

看来这里是关键了，看样子就是通过这些代码从一个APK文件加载res资源并创建Resources实例，经过这些逻辑后就可以使用R文件访问资源了。具体过程是，获取一个AssetManager实例，使用其“addAssetPath”方法加载APK（里的资源），再使用DisplayMetrics、Configuration、CompatibilityInfo实例一起创建我们想要的Resources实例。

最终访问插件APK里res资源的关键代码如下

```

try {
 AssetManager assetManager = AssetManager.class.newInstance();
 Method addAssetPath = assetManager.getClass().getMethod("addAssetPath", String.class);
 addAssetPath.invoke(assetManager, mDexPath);
 mAssetManager = assetManager;
} catch (Exception e) {
 e.printStackTrace();
}
Resources superRes = super.getResources();
mResources = new Resources(mAssetManager, superRes.getDisplayMetrics(),
 superRes.getConfiguration());

```

注意，有的人担心从插件APK加载进来的res资源的ID可能与主项目里现有的资源ID冲突，其实这种方式加载进来的res资源并不是融入到主项目里面来，主项目里的res资源是保存在ContextImpl里面的Resources实例，整个项目共有，而新加进来的res资源是保存在新创建的Resources实例的，也就是说ProxyActivity其实有两套res资源，并不是把新的res资源和原有的res资源合并了（所以不怕R.id重复），对两个res资源的访问都需要用对应的Resources实例，这也是开发时要处理的问题。（其实应该有3套，Android系统会加载一套framework-res.apk资源，里面存放系统默认Theme等资源）

额外补充下，这里你可能注意到了我们采用了反射的方法调用AssetManager的addAssetPath方法，而在上面ResourcesManager中调用AssetManager的addAssetPath方法是直接调用的，不用反射啊，而且看看SDK里AssetManager的addAssetPath方法的源码（这里也能看到具体APK资源的提取过程是在Native里完成的），发现它也是public类型的，外部可以直接调用，为什么还要用反射呢？

```

/**
 * Add an additional set of assets to the asset manager. This can be
 * either a directory or ZIP file. Not for use by applications. Returns
 * the cookie of the added asset, or 0 on failure.
 * {@hide}
 */
public final int addAssetPath(String path) {
 synchronized (this) {
 int res = addAssetPathNative(path);
 makeStringBlocks(mStringBlocks);
 return res;
 }
}

```

这里有个误区，SDK的源码只是给我们参考用的，APP实际上运行的代码逻辑在android.jar里面（位于 android-sdk\platforms\android-xx），反编译android.jar并找到ResourcesManager类就可以发现这些接口都是对应用层隐藏的。

```

public final class AssetManager{
 AssetManager(){throw new RuntimeException("Stub!"); }
 public void close() { throw new RuntimeException("Stub!"); }
 public final InputStream open(String fileName) throws IOException { throw new RuntimeException("Stub!");
 public final InputStream open(String fileName, int accessMode) throws IOException { throw new RuntimeException("Stub!");
 public final AssetFileDescriptor openFd(String fileName) throws IOException { throw new RuntimeException("Stub!");
 public final native String[] list(String paramString) throws IOException;

 public final AssetFileDescriptor openNonAssetFd(String fileName) throws IOException { throw new RuntimeException("Stub!");
 public final AssetFileDescriptor openNonAssetFd(int cookie, String fileName) throws IOException { throw new RuntimeException("Stub!");
 public final XmlResourceParser openXmlResourceParser(String fileName) throws IOException { throw new RuntimeException("Stub!");
 public final XmlResourceParser openXmlResourceParser(int cookie, String fileName) throws IOException { throw new RuntimeException("Stub!");
 protected void finalize() throws Throwable { throw new RuntimeException("Stub!");
 }
 public final native String[] getLocales();
}

```

到此，启动插件里的Activity的两大问题都有解决的方案了。

## 代理模式的具体项目

- [dynamic-load-apk](#)

上面只是分析了代理模式的关键技术点，如果运用到具体项目中去的话，除了两个关键的问题外，还有许多繁琐的细节需要处理，我们需要设计一个框架，规范插件APK项目的开发，也方便以后功能的扩展。这里，dynamic-load-apk向我们展示了许多优秀的处理方法，比如：

- 把Activity关键的生命周期方法抽象成DLPlugin接口，ProxyActivity通过DLPlugin代理调用插件Activity的生命周期；
- 设计一个基础的BasePluginActivity类，插件项目里使用这些基类进行开发，可以以接近常规Android开发的方式开发插件项目；
- 以类似的方式处理Service的问题；
- 处理了大量常见的兼容性问题（比如使用Theme资源时出现的问题）；
- 处理了插件项目里的so库的加载问题；
- 使用PluginPackage管理插件APK，从而可以方便地管理多个插件项目；

## 处理插件项目里的so库的加载

这里需要把插件APK里面的SO库文件解压释放出来，在根据当前设备CPU的型号选择对应的SO库，并使用System.load方法加载到当前内存中来，具体分析请参考[加载SD卡的SO库](#)。

## 多插件APK的管理

动态加载一个插件APK需要三个对应的 DexClassLoader、AssetManager、Resources 实例，可以用组合的方式创建一个 PluginPackage 类存放这三个变量，再创建一个管理类 PluginManager，用成员变量 `HashMap<dexPath, pluginPackage>` 的方式保存 `PluginPackage` 实例。

具体的代码请参考原项目的文档、源码以及Sample里面的示例代码，在这里感谢[singwhatiwanna](#)的开源精神。

## 实际应用中可能要处理的问题

### 插件APK的管理后台

使用动态加载的目的，就是希望可以绕过APK的安装过程升级应用的功能，如果插件APK是打包在主项目内部的那动态加载纯粹是多次一举。更多的时候我们希望可以在线下载插件APK，并且在插件APK有新版本的时候，主项目要从服务器下载最新的插件替换本地已经存在的旧插件。为此，我们应该有一个管理后台，它大概有以下功能：

- 上传不同版本的插件APK，并向APP主项目提供插件APK信息查询功能和下载功能；
- 管理在线的插件APK，并能向不同版本号的APP主项目提供最合适的插件APK；
- 万一最新的插件APK出现紧急BUG，要提供旧版本回滚功能；
- 出于安全考虑应该对APP项目的请求信息做一些安全性校验；

### 插件APK合法性校验

加载外部的可执行代码，一个逃不开的问题就是要确保外部代码的安全性，我们可不希望加载一些来历不明的插件APK，因为这些插件有的时候能访问主项目的关键数据。

最简单可靠的做法就是校验插件APK的MD5值，如果插件APK的MD5与我们服务器预置的数值不同，就认为插件被改动过，弃用。

## 是热部署，还是插件化？

这一部分作为补充说明，如果不太熟悉动态加载的使用姿势，可能不是那么容易理解。

谈到动态加载的时候我们经常说到“热部署”和“插件化”这些名词，它们虽然都和动态加载有关，但是还是有一点区别，这个问题涉及到主项目与插件项目的交互方式。前面我们说到，动态加载方式，可以在“项目层级”做到代码分离，按道理我们希望是主项目和插件项目不要有任何交互行为，实际上也应该如此！这样做不仅能确保项目的安全性，也能简化开发工作，所以一般的做法是

## 只有在用户使用到的时候才加载插件

主项目还是像常规Android项目那样开发，只有用户使用插件APK的功能时才动态加载插件并运行，插件一旦运行后，与主项目没有任何交互逻辑，只有在主项目启动插件的时候才触发一次调用插件的行为。比如，我们的主项目里有几款推广的游戏，平时在用户使用主项目功能时，可以先静默把游戏（其实就是一个插件APK）下载好，当用户点击游戏入口时，以动态加载的方式启动游戏，游戏只运行插件APK里的代码逻辑，结束后返回主项目界面。

## 一启动主项目就加载插件

另外一种完全相反的情形是，主项目只提供一个启动的入口，以及从服务器下载最新插件的更新逻辑，这两部分的代码都是长期保持不变的，应用一启动就动态加载插件，所有业务逻辑的代码都在插件里实现。比如现在一些游戏市场都要求开发者接入其SDK项目，如果SDK项目采用这种开发方式，先提供一个空壳的SDK给开发者，空壳SDK能从服务器下载最新的插件再运行插件里的逻辑，就能保证开发者开发的游戏每次启动的时候都能运行最新的代码逻辑，而不用让开发者在SDK有新版本的时候重新更换SDK并构建新的游戏APK。

## 让插件使用主项目的功能

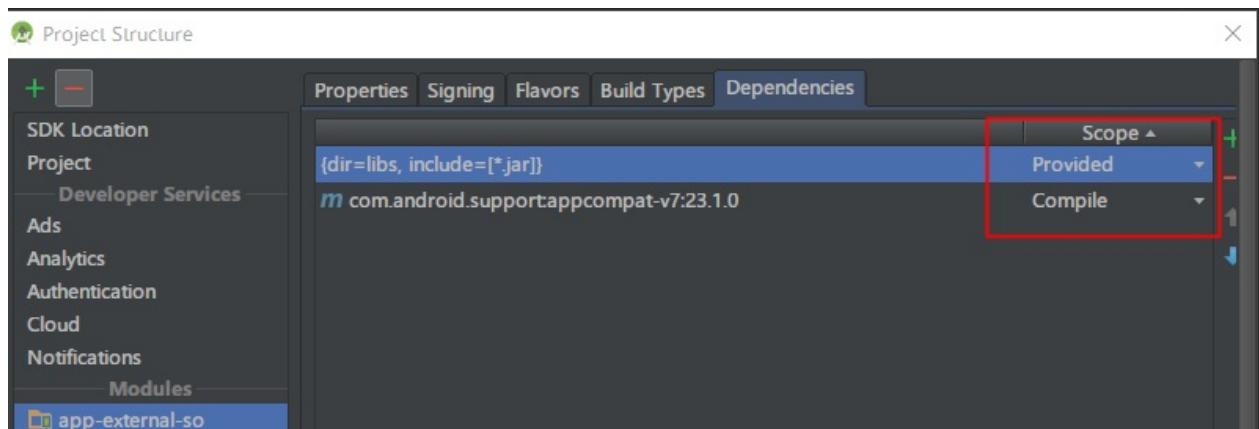
明明，说了不要交互的，偏偏，Android开发者就是这么执着于技术。

有些时候，比如，主项目里有一个成熟的图片加载框架ImageLoader，而插件里也有一个ImageLoader。如果一个应用同时运行两套ImageLoader，那会有许多额外的性能开销，如果能让插件也用主项目的ImageLoader就好了。另外，如果在插件里需要用到用户登录功能，我们总不希望用户使用主项目时进行一次登录，进入插件时再来一次登录，如果能在插件里使用主项目的登录状态就好了。

因此，有些时候我们希望插件项目能调用主项目的功能。怎么处理好呢，由于插件项目与主项目是分开的，我们在开发插件的时候，怎么调用主项目的代码啊？这里需要稍微了解一下Android项目间的依赖方式。

想想一个普通的APK是怎么构建和运行的，Android SDK提供了许多系统类（如Activity、Fragment等，一般我们也喜欢在这里查看源码），我们的Android项目依赖Android SDK项目并使用这些类进行开发，那构建APK的时候会把这些类打包进来吗？不会，要是每个APK都打包一份，那得有多少冗余啊。所以Android项目至少有两种依赖的方式，一种构建时会把被依赖的项目（Library）的类打包进来，一种不会。

在Android Studio打开项目的Project Structure，找到具体Module的Dependencies选项卡



可以看到Library项目有个Scope属性，这里的Compile模式就是会把Library的类打包进来，而Provided模式就不会。

注意，使用Provided模式的Library只能是jar文件，而不能是一个Android Library项目，因为后者可能自带了一些res资源，这些资源无法一并塞进标准的jar文件里面。到这里我们明白，Android SDK的代码其实是打包进系统ROM（俗称Framework层级）里面的，我们开发Android项目的时候，只是以Provided模式引用android.jar，从这个角度也佐证了上面谈到的“为什么APP实际运行时AssetManager类的逻辑会与Android SDK里的源码不一样”。

现在好办了，如果要在插件里使用主项目的ImageLoader，我们可以把ImageLoader的相关代码抽离成一个Android Library项目，主项目以Compile模式引用这个Library，而插件项目以Provided模式引用这个Library（编译出来的jar），这样能实现两者之间的交互了，当然代价也是明显的。

- 我们应该只给插件开放一些必要的接口，不然会有安全性问题；
- 作为通用模块的Library应该保持不变（起码接口不变），不然主项目与插件项目的版本同步会复杂许多；
- 因为插件项目已经严重依赖主项目了，所以插件项目不能独立运行，因为缺少必要的环境；

最后我们再说说“热部署”和“插件化”的区别，一般我们把独立运行的插件APK叫热部署，而需要依赖主项目的环境运行的插件APK叫做插件化。



# Android动态加载黑科技 动态创建Activity模式

来源:[segmentfault.com](https://segmentfault.com)

## 代理Activity模式的限制

还记得我们在代理Activity模式里谈到启动插件APK里的Activity的两个难题吗，由于插件里的Activity没在主项目的Manifest里面注册，所以无法经历系统Framework层级的一系列初始化过程，最终导致获得的Activity实例并没有生命周期和无法使用res资源。

使用代理Activity能够解决这两个问题，但是有一些限制

- 实际运行的Activity实例其实都是ProxyActivity，并不是真正想要启动的Activity；
- ProxyActivity只能指定一种LaunchMode，所以插件里的Activity无法自定义LaunchMode；
- 不支持静态注册的BroadcastReceiver；
- 往往不是所有的apk都可作为插件被加载，插件项目需要依赖特定的框架，还有需要遵循一定的"开发规范"；

特别是最后一个，无法直接把一个普通的APK作为插件使用。怎么避开这些限制呢？插件的Activity不是标准的Activity对象才会有这些限制，使其成为标准的Activity是解决问题的关键，而要使其成为标准的Activity，则需要在主项目里注册这些Activity。

总不能把插件APK所有的Activity都事先注册到主项目里面吧，想到代理模式需要注册一个代理的ProxyActivity，那么能不能在主项目里注册一个通用的Activity（比如TargetActivity）给插件里所有的Activity用呢？解决对策就是，在需要启动插件的某一个Activity（比如PlugActivity）的时候，动态创建一个TargetActivity，新创建的TargetActivity会继承PlugActivity的所有共有行为，而这个TargetActivity的包名与类名刚好与我们事先注册的TargetActivity一致，我们就能以标准的方式启动这个Activity。

## 动态创建Activity模式

运行时动态创建并编译一个Activity类，这种想法不是天方夜谭，动态创建类的工具有[dexmaker](#)和[asmdex](#)，二者均能实现动态字节码操作，最大的区别是前者是创建dex文件，而后者是创建class文件。

## 使用dexmaker动态创建一个类

运行时创建一个编译好并能运行的类叫做“动态字节码操作（runtime bytecode manipulation）”，使用dexmaker工具能创建一个dex文件，之后我们再反编译这个dex看看创建出来的类是什么样子。

```
public class MainActivity extends AppCompatActivity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 }

 public void onMakeDex(View view){
 try {
 DexMaker dexMaker = new DexMaker();
 // Generate a HelloWorld class.
 TypeId<?> helloWorld = TypeId.get("LHelloWorld;");
 dexMaker.declare(helloWorld, "HelloWorld.generated", Modifier.PUBLIC, Type
 generateHelloMethod(dexMaker, helloWorld);
 // Create the dex file and load it.
 File outputDir = new File(Environment.getExternalStorageDirectory() + File
 if (!outputDir.exists())outputDir.mkdir();
 ClassLoader loader = dexMaker.generateAndLoad(this.getClassLoader(), output
 Class<?> helloWorldClass = loader.loadClass("HelloWorld");
 // Execute our newly-generated code in-process.
 helloWorldClass.getMethod("hello").invoke(null);
 } catch (Exception e) {
 Log.e("MainActivity", "[onMakeDex]", e);
 }
 }

 /**
 * Generates Dalvik bytecode equivalent to the following method.
 * public static void hello() {
 * int a = 0abcd;
 * int b = 0aaaa;
 * int c = a - b;
 * String s = Integer.toHexString(c);
 * System.out.println(s);
 * return;
 * }
 */
 private static void generateHelloMethod(DexMaker dexMaker, TypeId<?> declaringType
 // Lookup some types we'll need along the way.
 TypeId<System> systemType = TypeId.get(System.class);
 TypeId<PrintStream> printStreamType = TypeId.get(PrintStream.class);

 // Identify the 'hello()' method on declaringType.
```

```

MethodId hello = declaringType.getMethod(TypeId.VOID, "hello");

// Declare that method on the dexMaker. Use the returned Code instance
// as a builder that we can append instructions to.
Code code = dexMaker.declare(hello, Modifier.STATIC | Modifier.PUBLIC);

// Declare all the locals we'll need up front. The API requires this.
Local<Integer> a = code.newLocal(TypeId.INT);
Local<Integer> b = code.newLocal(TypeId.INT);
Local<Integer> c = code.newLocal(TypeId.INT);
Local<String> s = code.newLocal(TypeId.STRING);
Local<PrintStream> localSystemOut = code.newLocal(printStreamType);

// int a = 0xabcd;
code.loadConstant(a, 0xabcd);

// int b = 0xaaaa;
code.loadConstant(b, 0xaaaa);

// int c = a - b;
code.op(BinaryOp.SUBTRACT, c, a, b);

// String s = Integer.toHexString(c);
MethodId<Integer, String> toHexString
 = TypeId.get(Integer.class).getMethod(TypeId.STRING, "toHexString", T
code.invokeStatic(toHexString, s, c);

// System.out.println(s);
FieldId<System, PrintStream> systemOutField = systemType.getField(printStream
code.sget(systemOutField, localSystemOut);
MethodId<PrintStream, Void> printlnMethod = printStreamType.getMethod(
 TypeId.VOID, "println", TypeId.STRING);
code.invokeVirtual(printlnMethod, null, localSystemOut, s);

// return;
code.returnVoid();
}

}

```

运行后在SD卡的dexmaker目录下找到刚创建的文件“Generated1532509318.jar”，把里面的“classes.dex”解压出来，然后再用“dex2jar”工具转化成jar文件，最后再用“jd-gui”工具反编译jar的源码。



## 修改需要启动的目标Activity

接下来的问题是如何把需要启动的、在Manifest里面没有注册的PlugActivity换成有注册的TargetActivity。

在Android，虚拟机加载类的时候，是通过ClassLoader的loadClass方法，而loadClass方法并不是final类型的，这意味着我们可以创建自己的类去继承ClassLoader，以重载loadClass方法并改写类的加载逻辑，在需要加载PlugActivity的时候，偷偷把其换成TargetActivity。

大致思路如下

```

public class CJClassLoader extends ClassLoader{

@Override
public Class loadClass(String className){
 if(当前上下文插件不为空) {
 if(className 是 TargetActivity){
 找到当前实际要加载的原始PlugActivity，动态创建类 (TargetActivity extends PlugActivity)
 return 从dex文件中加载的TargetActivity
 }else{
 return 使用对应的PluginClassLoader加载普通类
 }
 }else{
 return super.loadClass() //使用原来的类加载方法
 }
}
}

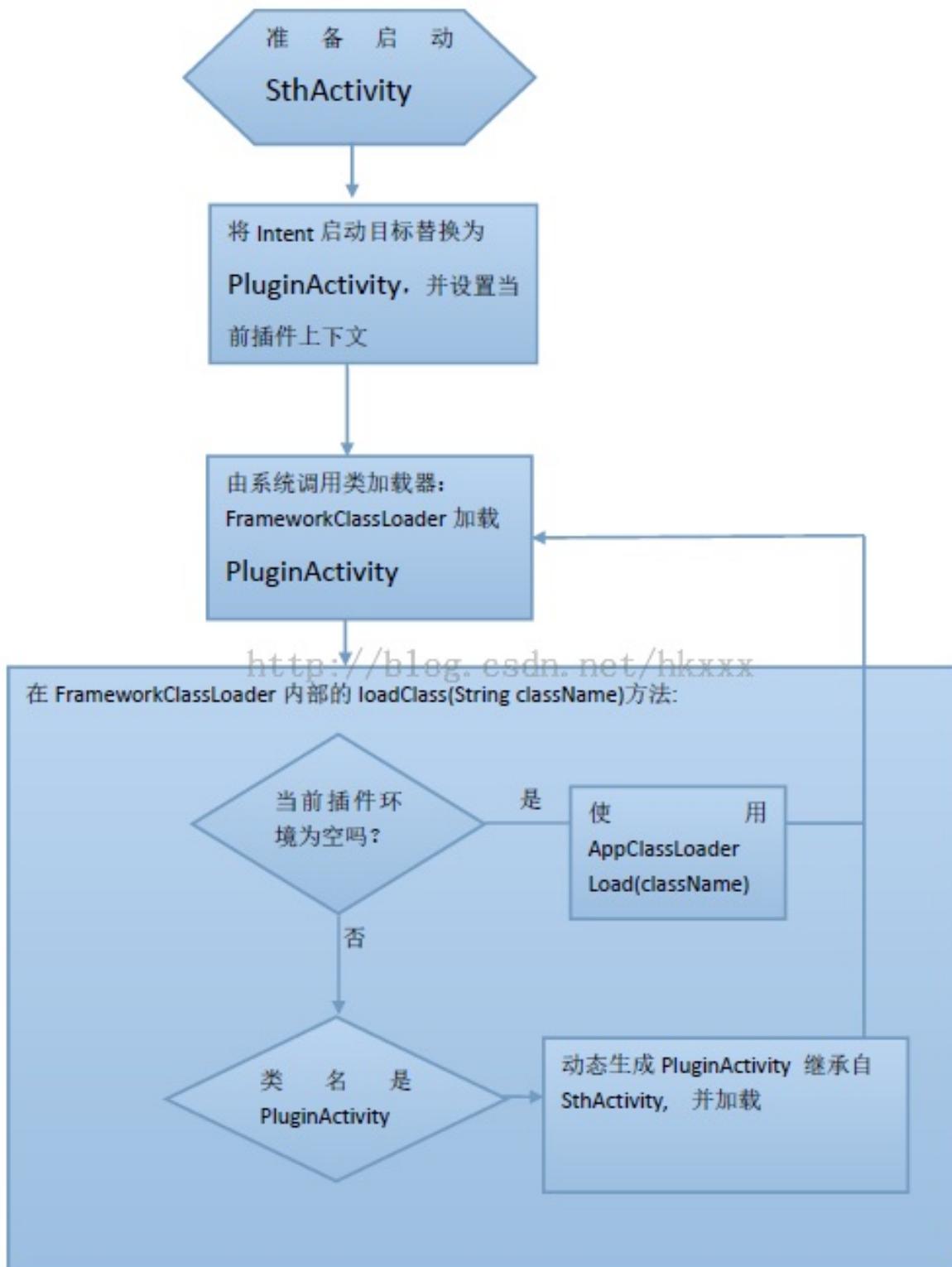
```

这样就能把启动插件里的PlugActivity变成启动动态创建的TargetActivity。

不过还有一个问题，主项目启动插件Activity的时候，我们可以替换Activity，但是如果在插件Activity（比如MainActivity）启动另一个Activity（SubActivity）的时候怎么办？插件时普通的第三方APK，我们无法更改里面跳转Activity的逻辑。其实，从主项目启动插件MainActivity的时候，其实启动的是我们动态创建的TargetActivity（extends MainActivity），而我们知道Activity启动另一个Activity的时候都是使用其“startActivityForResult”方法，所以我们可以创建TargetActivity时，重写其“startActivityForResult”方法，让它在启动其他Activity的时候，也采用动态创建Activity的方式，这样就能解决问题。

## 动态创建Activity开源项目

这种脑洞大开的动态加载思路来自于houkx的开源项目[android-pluginmgr](#)



`android-pluginmgr` 项目中有三种 `ClassLoader`，一是用于替换宿主APK的 `Application` 的 `CJClassLoader`，二是用于加载插件APK的 `PluginClassLoader`，再来是用于加载启动插件Activity时动态生成的 `PluginActivity` 的dex包的 `DexClassLoader`（放在 `Map`集合 `proxyActivityLoaderMap` 里面）。其中 `CJClassLoader` 是 `PluginClassLoader` 的 Parent，而 `PluginClassLoader` 又是第三种 `DexClassLoader` 的Parent。

ClassLoader 类加载Class的时候，会先使用Parent的 classLoader，但Parent不能完成加载工作时，才会调用Child的 classLoader 去完成工作。

### java.lang.ClassLoader

Loads classes and resources from a repository. One or more class loaders are installed at runtime. These are consulted whenever the runtime system needs a specific class that is not yet available in-memory. Typically, class loaders are grouped into a tree where child class loaders delegate all requests to parent class loaders. Only if the parent class loader cannot satisfy the request, the child class loader itself tries to handle it.

具体分析请参考[ClassLoader的工作机制](#)

所以每加载一个Activity的时候都会调用到最上级的CJClassLoader的loadClass方法，从而保证启动插件Activity的时候能顺利替换成PlugActivity。当然如何控制着三种ClassLoader的加载工作，也是pluginmgr项目的设计难度之一。

## 存在的问题

动态类创建的方式，使得注册一个通用的Activity就能给多给Activity使用，对这种做法存在的问题也是明显的

- 使用同一个注册的Activity，所以一些需要在Manifest注册的属性无法做到每个Activity都自定义配置；
- 插件中的权限，无法动态注册，插件需要的权限都得在宿主中注册，无法动态添加权限；
- 插件的Activity无法开启独立进程，因为这需要在Manifest里面注册；
- 动态字节码操作涉及到Hack开发，所以相比代理模式起来不稳定；

其中不稳定的问题出现在对Service的支持上，使用动态创建类的方式可以搞定Activity和Broadcast Receiver，但是使用类似的方式处理Service却不行，因为“ContextImpl.getApplicationContext”期待得到一个非ContextWrapper的context，如果不是则继续下次循环，目前的Context实例都是wrapper，所以会进入死循环。

据houkx称他现在有另外的思路实现“启动为安装的普通第三方APK”的目的，而且不是基于动态类创建的原理，期待他的开源项目的更新。

## 代理Activity模式与动态创建Activity模式的区别

简单地说，最大的不同是代理模式使用了一个代理的Activity，而动态创建Activity模式使用了一个通用的Activity。

代理模式中，使用一个代理Activity去完成本应该由插件Activity完成的工作，这个代理Activity是一个标准的Android Activity组件，具有生命周期和上下文环境

(ContextWrapper和ContextCompl)，但是它自身只是一个空壳，并没有承担什么业务逻辑；而插件Activity其实只是一个普通的Java对象，它没有上下文环境，但是却能正常执行业务逻辑的代码。代理Activity和不同的插件Activity配合起来，就能完成不同的业务逻辑了。所以代理模式其实还是使用常规的Android开发技术，只是在处理插件资源的时候强制调用了系统的隐藏API，因此这种模式还是可以稳定工作和升级的。

动态创建Activity模式，被动态创建出来的Activity类是有在主项目里面注册的，它是一个标准的Activity，它有自己的Context和生命周期，不需要代理的Activity。

# Android插件化原理解析

# Android插件化原理解析——概要

来源:Weishu's Notes

2015年是Android插件化技术突飞猛进的一年，随着业务的发展各大厂商都碰到了Android Native平台的瓶颈：

- 1、从技术上讲，业务逻辑的复杂导致代码量急剧膨胀，各大厂商陆续出到65535方法数的天花板；同时，运营为王的时代对于模块热更新提出了更高的要求。
- 2、在业务层面上，功能模块的解耦以及维护团队的分离也是大势所趋；各个团队维护着同一个App的不同模块，如果每个模块升级新功能都需要对整个app进行升级，那么发布流程不仅复杂而且效率低下；在讲究小步快跑和持续迭代的移动互联网必将遭到淘汰。

H5和Hybird可以解决这些问题，但是始终比不上native的用户体验；于是，国外的FaceBook推出了 react-native；而国内各大厂商几乎都选择纯native的插件化技术。可以说，Android的未来必将是 react-native 和插件化的天下。

react-native 资料很多，但是讲述插件化的却凤毛菱角；插件化技术听起来高深莫测，实际上要解决的就是两个问题：

- 代码加载
- 资源加载

## 代码加载

类的加载可以使用Java的 ClassLoader 机制，但是对于Android来说，并不是说类加载进来就可以用了，很多组件都是有“生命”的；因此对于这些有血有肉的类，必须给它们注入活力，也就是所谓的组件生命周期管理；

另外，如何管理加载进来的类也是一个问题。假设多个插件依赖了相同的类，是抽取公共依赖进行管理还是插件单独依赖？这就是ClassLoader的管理问题；

## 资源加载

资源加载方案大家使用的原理都差不多，都是用 AssetManager 的隐藏方法 addAssetPath；但是，不同插件的资源如何管理？是公用一套资源还是插件独立资源？共用资源如何避免资源冲突？对于资源加载，有的方案共用一套资源并采用资源分段

机制解决冲突（要么修改aapt要么添加编译插件）；有的方案选择独立资源，不同插件管理自己的资源。

目前国内开源的较成熟的插件方案有[DL](#)和[DroidPlugin](#)；但是DL方案仅仅对Framework的表层做了处理，严重依赖 that 语法，编写插件代码和主程序代码需单独区分；而DroidPlugin通过Hook增强了Framework层的很多系统服务，开发插件就跟开发独立app差不多；就拿Activity生命周期的管理来说，DL的代理方式就像是牵线木偶，插件只不过是操纵傀儡而已；而DroidPlugin则是借尸还魂，插件是有血有肉的系统管理的真正组件；DroidPlugin Hook了系统几乎所有的Service，欺骗了大部分的系统API；掌握这个Hook过程需要掌握很多系统原理，因此学习DroidPlugin对于整个Android FrameWork层大有裨益。

接下来的一系列文章将以DroidPlugin为例讲解插件框架的原理，揭开插件化的神秘面纱；同时还能帮助深入理解Android Framework；主要内容如下：

- [Hook机制之动态代理](#)
- [Hook机制之Binder Hook](#)
- [Hook机制之AMS&PMS](#)
- [Activity生命周期管理](#)
- [插件加载机制](#)
- [广播的管理方式](#)
- [Service的插件化](#)
- [ContentProvider的插件化](#)
- [DroidPlugin插件通信机制](#)
- [插件机制之资源管理](#)
- [不同插件框架方案对比](#)
- [插件化的未来](#)

另外，对于每一章内容都会有详细的demo，具体见[understand-plugin-framework](#)；喜欢就点个关注吧～定期更新，敬请期待！









# Android插件化原理解析——Hook机制之动态代理

来源:[Weishu's Notes](#)

使用代理机制进行API Hook进而达到方法增强是框架的常用手段，比如J2EE框架Spring通过动态代理优雅地实现了AOP编程，极大地提升了Web开发效率；同样，插件框架也广泛使用了代理机制来增强系统API从而达到插件化的目的。本文将带你了解基于动态代理的Hook机制。

阅读本文之前，可以先clone一份[understand-plugin-framework](#)，参考此项目的 `dynamic-proxy-hook` 模块。另外，插件框架原理解析系列文章见索引。

## 代理是什么

为什么需要代理呢？其实这个代理与日常生活中的“代理”，“中介”差不多；比如你想海淘买东西，总不可能亲自飞到国外去购物吧，这时候我们使用第三方海淘服务比如惠惠购物助手等；同样拿购物为例，有时候第三方购物会有折扣比如当初的米折网，这时候我们可以少花点钱；当然有时候这个“代理”比较坑，坑我们的钱，坑我们的货。

从这个例子可以看出来，**代理可以实现方法增强**，比如常用的日志,缓存等；也可以实现方法拦截，通过代理方法修改原方法的参数和返回值，从而实现某种不可告人的目的～接下来我们用代码解释一下。

## 静态代理

静态代理，是最原始的代理方式；假设我们有一个购物的接口，如下：

```
public interface Shopping {
 Object[] doShopping(long money);
}
```

它有一个原始的实现，我们可以理解为亲自，直接去商店购物：

```

public class ShoppingImpl implements Shopping {
 @Override
 public Object[] doShopping(long money) {
 System.out.println("逛淘宝，逛商场，买买买!!!");
 System.out.println(String.format("花了%s块钱", money));
 return new Object[] { "鞋子", "衣服", "零食" };
 }
}

```

好了，现在我们自己没时间但是需要买东西，于是我们就找了个代理帮我们买：

```

public class ProxyShopping implements Shopping {

 Shopping base;

 ProxyShopping(Shopping base) {
 this.base = base;
 }

 @Override
 public Object[] doShopping(long money) {

 // 先黑点钱(修改输入参数)
 long readCost = (long) (money * 0.5);

 System.out.println(String.format("花了%s块钱", readCost));

 // 帮忙买东西
 Object[] things = base.doShopping(readCost);

 // 偷梁换柱(修改返回值)
 if (things != null && things.length > 1) {
 things[0] = "被掉包的东西!!";
 }

 return things;
 }
}

```

很不幸，我们找的这个代理有点坑，坑了我们的钱还坑了我们的货；先忍忍。

## 动态代理

传统的静态代理模式需要为每一个需要代理的类写一个代理类，如果需要代理的类有几百个那不是要累死？为了更优雅地实现代理模式，JDK提供了动态代理方式，可以简单理解为在JVM可以在运行时帮我们动态生成一系列的代理类，这样我们就不需要手写每一个静

态的代理类了。依然以购物为例，用动态代理实现如下：

```
public static void main(String[] args) {
 Shopping women = new ShoppingImpl
 // 正常购物
 System.out.println(Arrays.toString(women.doShopping(100))
 // 招代理
 women = (Shopping) Proxy.newProxyInstance(Shopping.class.getClassLoader(),
 women.getClass().getInterfaces(), new ShoppingHandler(women)
 System.out.println(Arrays.toString(women.doShopping(100)));
}
```

动态代理主要处理 `InvocationHandler` 和 `Proxy` 类；完整代码可以见[github](#)

## 代理Hook

我们知道代理有比原始对象更强大的能力，比如飞到国外买东西，比如坑钱坑货；那么很自然，如果我们自己创建代理对象，然后把原始对象替换为我们的代理对象，那么就可以在这个代理对象为所欲为了；修改参数，替换返回值，我们称之为Hook。

下面我们Hook掉 `startActivity` 这个方法，使得每次调用这个方法之前输出一条日志；（当然，这个输入日志有点点弱，只是为了展示原理；只要你想，你想可以替换参数，拦截这个 `startActivity` 过程，使得调用它导致启动某个别的Activity，指鹿为马！）

首先我们得找到被Hook的对象，我称之为Hook点；什么样的对象比较好Hook呢？自然是容易找到的对象。什么样的对象容易找到？静态变量和单例；在一个进程之内，静态变量和单例变量是相对不容易发生变化的，因此非常容易定位，而普通的对象则要么无法标志，要么容易改变。我们根据这个原则找到所谓的Hook点。

然后我们分析一下 `startActivity` 的调用链，找出合适的Hook点。我们知道对于 `Context.startActivity`（`Activity.startActivity` 的调用链与之不同），由于Context的实现实际上是 `ContextImpl`；我们看 `ContextImpl` 类的 `startActivity` 方法：

```

@Override
public void startActivityForResult(Intent intent, Bundle options) {
 warnIfCallingFromSystemProcess();
 if ((intent.getFlags() & Intent.FLAG_ACTIVITY_NEW_TASK) == 0) {
 throw new AndroidRuntimeException(
 "Calling startActivityForResult() from outside of an Activity "
 + " context requires the FLAG_ACTIVITY_NEW_TASK flag."
 + " Is this really what you want?");
 }
 mMainThread.getInstrumentation().execStartActivity(
 getOuterContext(), mMainThread.getApplicationThread(), null,
 (Activity)null, intent, -1, options);
}

```

这里，实际上使用了 `ActivityThread` 类的 `mInstrumentation` 成员的 `execStartActivity` 方法；注意到，`ActivityThread` 实际上是主线程，而主线程一个进程只有一个，因此这里是一个良好的Hook点。

接下来就是想要Hook掉我们的主线程对象，也就是把这个主线程对象里面的 `mInstrumentation` 给替换成我们修改过的代理对象；要替换主线程对象里面的字段，首先我们得拿到主线程对象的引用，如何获取呢？`ActivityThread` 类里面有一个静态方法 `currentActivityThread` 可以帮助我们拿到这个对象类；但是 `ActivityThread` 是一个隐藏类，我们需要用反射去获取，代码如下：

```

// 先获取到当前的ActivityThread对象
Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
Method currentActivityThreadMethod = activityThreadClass.getDeclaredMethod("currentActivityThread");
currentActivityThreadMethod.setAccessible(true);
Object currentActivityThread = currentActivityThreadMethod.invoke(null);

```

拿到这个 `currentActivityThread` 之后，我们需要修改它的 `mInstrumentation` 这个字段为我们的代理对象，我们先实现这个代理对象，由于JDK动态代理只支持接口，而这个 `Instrumentation` 是一个类，没办法，我们只有手动写静态代理类，覆盖掉原始的方法即可。（`cglib` 可以做到基于类的动态代理，这里先不介绍）

```

public class EvilInstrumentation extends Instrumentation {
 private static final String TAG = "EvilInstrumentation";
 // ActivityThread中原始的对象，保存起来
 Instrumentation mBase;
 public EvilInstrumentation(Instrumentation base) {
 mBase = base;
 }
 public ActivityResult execStartActivity(
 Context who, IBinder contextThread, IBinder token, Activity target,
 Intent intent, int requestCode, Bundle options){
 // Hook之前，XXX到此一游！
 Log.d(TAG, "\n执行了startActivity，参数如下：\n" + "who = [" + who + "], " +
 "\ncontextThread = [" + contextThread + "], \ntoken = [" + token + "]"
 "\ntarget = [" + target + "], \nintent = [" + intent +
 "], \nrequestCode = [" + requestCode + "], \noptions = [" + options +
 "\n"
 // 开始调用原始的方法，调不调用随你，但是不调用的话，所有的startActivity都失效了。
 // 由于这个方法是隐藏的，因此需要使用反射调用；首先找到这个方法
 try {
 Method execStartActivity = Instrumentation.class.getDeclaredMethod(
 "execStartActivity",
 Context.class, IBinder.class, IBinder.class, Activity.class,
 Intent.class, int.class, Bundle.class);
 execStartActivity.setAccessible(true);
 return (ActivityResult) execStartActivity.invoke(mBase, who,
 contextThread, token, target, intent, requestCode, options);
 } catch (Exception e) {
 // 某该死的rom修改了 需要手动适配
 throw new RuntimeException("do not support!!! pls adapt it");
 }
 }
}

```

Ok，有了代理对象，我们要做的就是偷梁换柱！代码比较简单，采用反射直接修改：

```

public static void attachContext() throws Exception{
 // 先获取到当前的ActivityThread对象
 Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
 Field currentActivityThreadField = activityThreadClass.getDeclaredField("sCurrentActivityThread");
 currentActivityThreadField.setAccessible(true);
 Object currentActivityThread = currentActivityThreadField.get(null);

 // 拿到原始的 mInstrumentation字段
 Field mInstrumentationField = activityThreadClass.getField("mInstrumentation");
 mInstrumentationField.setAccessible(true);
 Instrumentation mInstrumentation = (Instrumentation) mInstrumentationField.get(currentActivityThread);

 // 创建代理对象
 Instrumentation evilInstrumentation = new EvilInstrumentation(mInstrumentation);

 // 偷梁换柱
 mInstrumentationField.set(currentActivityThread, evilInstrumentation);
}

```

好了，我们启动一个Activity测试一下，结果如下：

```

02-22 00:19:39.515 9207-9207/com.weishu.upf.dynamic_proxy_hook.app2 D/EvilInstrumentation: 执行了startActivity, 参数如下:
who = [android.app.Application@7654b0b],
contextThread = [android.app.ActivityThread$ApplicationThread@8807de8],
token = [null],
target = [null],
intent = [Intent { act=android.intent.action.VIEW dat=http://www.baidu.com/... flg=0x10000000 }],
requestCode = [-1],
options = [null]

```

可见，Hook确实成功了！这就是使用代理进行Hook的原理——偷梁换柱。整个Hook过程简要总结如下：

- 1、寻找Hook点，原则是静态变量或者单例对象，尽量Hook public的对象和方法，非public不保证每个版本都一样，需要适配。
- 2、选择合适的代理方式，如果是接口可以用动态代理；如果是类可以手动写代理也可以使用cglib。
- 3、偷梁换柱——用代理对象替换原始对象

完整代码参照：[understand-plugin-framework](#)；里面留有一个作业：我们目前仅Hook了 context 类的 startActivity 方法，但是 Activity 类却使用了自己 的 mInstrumentation；你可以尝试Hook掉 Activity 类的 startActivity 方法。

喜欢就点个赞吧～持续更新，请关注github项目 [understand-plugin-framework](#)和[我的博客](#)！

# Android插件化原理解析——Hook机制之 Binder Hook

来源:[Weishu's Notes](#)

Android系统通过Binder机制给应用程序提供了一系列的系统服务，诸如 `ActivityManagerService`，`ClipboardManager`，`AudioManager` 等；这些广泛存在系统服务给应用程序提供了诸如任务管理，音频，视频等异常强大的功能。

插件框架作为各个插件的管理者，为了使得插件能够无缝地使用这些系统服务，自然会对这些系统服务做出一定的改造(Hook)，使得插件的开发和使用更加方便，从而大大降低插件的开发和维护成本。比如，Hook住 `ActivityManagerService` 可以让插件无缝地使用 `startActivity` 方法而不是使用特定的方式(比如that语法)来启动插件或者主程序的任意界面。

我们把这种Hook系统服务的机制称之为 `Binder Hook`，因为本质上这些服务提供者都是存在于系统各个进程的 `Binder` 对象。因此，要理解接下来的内容必须了解Android的 `binder` 机制，可以参考我之前的文章[Binder学习指南](#)

阅读本文之前，可以先clone一份 [understand-plugin-framework](#)，参考此项目的 `binder-hook` 模块。另外，插件框架原理解析系列文章见[索引](#)。

## 系统服务的获取过程

我们知道系统的各个远程 `service` 对象都是以 `Binder` 的形式存在的，而这些 `Binder` 有一个管理者，那就是 `ServiceManager`；我们要Hook掉这些 `service`，自然要从这个 `ServiceManager` 下手，不然星罗棋布的 `Binder` 广泛存在于系统的各个角落，要一个个找出来还真是大海捞针。

回想一下我们使用系统服务的时候是怎么干的，想必这个大家一定再熟悉不过了：通过 `Context` 对象的 `getSystemService` 方法；比如要使用 `ActivityManager`：

```
ActivityManager am = (ActivityManager)
 context.getSystemService(Context.ACTIVITY_SERVICE);
```

可是这个貌似跟 `ServiceManager` 没有什么关系啊？我们再查看 `getSystemService` 方法；(`Context`的实现在`ContextImpl`里面)：

```

public Object getSystemService(String name) {
 ServiceFetcher fetcher = SYSTEM_SERVICE_MAP.get(name);
 return fetcher == null ? null : fetcher.getService(this);
}

```

很简单，所有的service对象都保存在一张map里面，我们再看这个map是怎么初始化的：

```

registerService(ACCOUNT_SERVICE, new ServiceFetcher() {
 public Object createService(ContextImpl ctx) {
 IBinder b = ServiceManager.getService(ACCOUNT_SERVICE);
 IAccountManager service = IAccountManager.Stub.asInterface(b);
 return new AccountManager(ctx, service);
 }
});

```

在 ContextImpl 的静态初始化块里面，有的 service 是像上面这样初始化的；可以看到，确实使用了 ServiceManager；当然还有一些service并没有直接使用 ServiceManager，而是做了一层包装并返回了这个包装对象，比如我们的 ActivityManager，它返回的是 ActivityManager 这个包装对象：

```

registerService(ACTIVITY_SERVICE, new ServiceFetcher() {
 public Object createService(ContextImpl ctx) {
 return new ActivityManager(ctx.getOuterContext(), ctx.mMainThread
 }
});

```

但是在 ActivityManager 这个类内部，也使用了 ServiceManager；具体来说，因为 ActivityManager 里面所有的核心操作都是使用 ActivityManagerNative.getDefault() 完成的。那么这个语句干了什么呢？

```

private static final Singleton<IActivityManager> gDefault = new Singleton<IActivityManager>() {
 protected IActivityManager create() {
 IBinder b = ServiceManager.getService("activity");
 IActivityManager am = asInterface(b);
 return am;
 }
};

```

因此，通过分析我们得知，系统Service的使用其实就分为两步：

```

IBinder b = ServiceManager.getService("service_name"); // 获取原始的IBinder对象
IXXInterface in = IXXInterface.Stub.asInterface(b); // 转换为Service接口

```

# 寻找Hook点

在[插件框架原理解析——Hook机制之动态代理](#)里面我们说过，Hook分为三步，最关键的一步就是寻找Hook点。我们现在已经搞清楚了系统服务的使用过程，那么就需要找出在这个过程中，在哪个环节是最合适hook的。

由于系统服务的使用者都是对第二步获取到的IXXInterface进行操作，因此如果我们要hook掉某个系统服务，**只需要把第二步的asInterface方法返回的对象修改为我们Hook过的对象就可以了。**

## asInterface过程

接下来我们分析 asInterface 方法，然后想办法把这个方法的返回值修改为我们Hook过的系统服务对象。这里我们以系统剪切板服务为例，源码位置

为 `android.content.IClipboard, IClipboard.Stub.asInterface` 方法代码如下：

```
public static android.content.IClipboard asInterface(android.os.IBinder obj) {
 if ((obj == null)) {
 return null;
 }
 android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR); // Hook点
 if (((iin != null) && (iin instanceof android.content.IClipboard))) {
 return ((android.content.IClipboard) iin);
 }
 return new android.content.IClipboard.Stub.Proxy(obj);
}
```

这个方法的意思就是：先查看本进程是否存在这个Binder对象，如果有那么直接就是本进程调用了；如果不存在那么创建一个代理对象，让代理对象委托驱动完成跨进程调用。

观察这个方法，前面的那个if语句判空返回肯定动不了手脚；最后一句调用构造函数然后直接返回我们也是无从下手，要修改 `asInterface` 方法的返回值，我们唯一能做的就是从这一句下手：

```
android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR); // Hook点
```

我们可以尝试修改这个obj对象的queryLocalInterface 方法的返回值，并保证这个返回值符合接下来的if条件检测，那么就达到了修改 `asInterface` 方法返回值的目的。

而这个obj对象刚好是我们第一步返回的 `IBinder` 对象，接下来我们尝试对这个 `IBinder` 对象的 `queryLocalInterface` 方法进行hook。

## getService过程

上文分析得知，我们想要修改IBinder对象的 queryLocalInterface 方法；获取IBinder对象的过程如下：

```
IBinder b = ServiceManager.getService("service_name");
```

因此，我们希望能修改这个 getService 方法的返回值，让这个方法返回一个我们伪造过的 IBinder 对象；这样，我们可以在自己伪造的 IBinder 对象的 queryLocalInterface 方法作处理，进而使得 asInterface 方法返回在 queryLocalInterface 方法里面处理过的值，最终实现hook系统服务的目的。

在跟踪这个 getService 方法之前我们思考一下，由于系统服务是一系列的远程 service，它们的本体，也就是 Binder 本地对象一般都存在于某个单独的进程，在这个进程之外的其他进程存在的都是这些 Binder 本地对象的代理。因此在我们的进程里面，存在的也只是这个 Binder 代理对象，我们也只能对这些 Binder 代理对象下手。(如果这一段看不懂，建议不要往下看了，先看[Binder学习指南](#))

然后，这个 getService 是一个静态方法，如果此方法什么都不做，拿到 Binder 代理对象之后直接返回；那么我们就无能为力了：我们没有办法拦截一个静态方法，也没有办法获取到这个静态方法里面的局部变量(即我们希望修改的那个Binder代理对象)。

接下来就可以看这个getService的代码了：

```
ublic static IBinder getService(String name) {
 try {
 IBinder service = sCache.get(name);
 if (service != null) {
 return service;
 } else {
 return getIServiceManager().getService(name);
 }
 } catch (RemoteException e) {
 Log.e(TAG, "error in getService", e);
 }
 return null;
}
```

天无绝人之路！ ServiceManager 为了避免每次都进行跨进程通信，把这些 Binder 代理对象缓存在一张map里面。

我们可以替换这个 `map` 里面的内容为 Hook 过的 `IBinder` 对象，由于系统在 `getService` 的时候每次都会优先查找缓存，因此返回给使用者的都是被我们修改过的对象，从而达到瞒天过海的目的。

总结一下，要达到修改系统服务的目的，我们需要如下两步：

- 首先肯定需要伪造一个系统服务对象，接下来就要想办法让 `asInterface` 能够返回我们的这个伪造对象而不是原始的系统服务对象。
- 通过上文分析我们知道，只要让 `getService` 返回 `IBinder` 对象的 `queryLocalInterface` 方法直接返回我们伪造过的系统服务对象就能达到目的。所以，我们需要伪造一个 `IBinder` 对象，主要是修改它的 `queryLocalInterface` 方法，让它返回我们伪造的系统服务对象；然后把这个伪造对象放置在 `ServiceManager` 的缓存 `map` 里面即可。

我们通过 Binder 机制的优先查找本地 Binder 对象的这个特性达到了 Hook 掉系统服务对象的目的。因此 `queryLocalInterface` 也失去了它原本的意义(只查找本地 Binder 对象，没有本地对象返回 null)，这个方法只是一个傀儡，是我们实现 hook 系统对象的桥梁：我们通过这个“漏洞”让 `asInterface` 永远都返回我们伪造过的对象。由于我们接管了 `asInterface` 这个方法的全部，我们伪造过的这个系统服务对象不能是只拥有本地 `Binder` 对象(原始 `queryLocalInterface` 方法返回的对象)的能力，还要有 `Binder` 代理对象操纵驱动的能力。

接下来我们就以 Hook 系统的剪切版服务为例，用实际代码来说明，如何 Hook 掉系统服务。

## Hook 系统剪切版服务

### 伪造剪切版服务对象

首先我们用代理的方式伪造一个剪切版服务对象，关于如何使用代理的方式进行 hook 以及其中的原理，可以查看[插件框架原理解析——Hook 机制之动态代理](#)。

具体代码如下，我们用动态代理的方式 Hook 掉了 `hasPrimaryClip()`，`getPrimaryClip()` 这两个方法：

```

public class BinderHookHandler implements InvocationHandler {

 private static final String TAG = "BinderHookHandler";

 // 原始的Service对象 (IInterface)
 Object base;

 public BinderHookHandler(IBinder base, Class<?> stubClass) {
 try {
 Method asInterfaceMethod = stubClass.getDeclaredMethod("asInterface", IBi
 // IClipboard.Stub.asInterface(base);
 this.base = asInterfaceMethod.invoke(null, base);
 } catch (Exception e) {
 throw new RuntimeException("hooked failed!");
 }
 }

 @TargetApi(Build.VERSION_CODES.HONEYCOMB)
 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable

 // 把剪切版的内容替换为 "you are hooked"
 if ("getPrimaryClip".equals(method.getName())) {
 Log.d(TAG, "hook getPrimaryClip");
 return ClipData.newPlainText(null, "you are hooked");
 }

 // 欺骗系统,使之认为剪切版上一直有内容
 if ("hasPrimaryClip".equals(method.getName())) {
 return true;
 }

 return method.invoke(base, args);
 }
}

```

注意，我们拿到原始的 `IBinder` 对象之后，如果我们希望使用被Hook之前的系统服务，并不能直接使用这个 `IBinder` 对象，而是需要使用 `asInterface` 方法将它转换为 `IClipboard` 接口；因为 `getService` 方法返回的 `IBinder` 实际上是一个裸 `Binder` 代理对象，它只有与驱动打交道的能力，但是它并不能独立工作，需要人指挥它； `asInterface` 方法返回的 `IClipboard.Stub.Proxy` 类的对象通过操纵这个裸 `BinderProxy` 对象从而实现了具体的 `IClipboard` 接口定义的操作。

## 伪造`IBinder`对象

在上一步中，我们已经伪造好了系统服务对象，现在要做的就是想办法让 `asInterface` 方法返回我们伪造的对象了；我们伪造一个 `IBinder` 对象：

```

public class BinderProxyHookHandler implements InvocationHandler {

 private static final String TAG = "BinderProxyHookHandler";

 // 绝大部分情况下, 这是一个BinderProxy对象
 // 只有当Service和我们在同一个进程的时候才是Binder本地对象
 // 这个基本不可能
 IBinder base;

 Class<?> stub;

 Class<?> iinterface;

 public BinderProxyHookHandler(IBinder base) {
 this.base = base;
 try {
 this.stub = Class.forName("android.content.IClipboard$Stub");
 this.iinterface = Class.forName("android.content.IClipboard");
 } catch (ClassNotFoundException e) {
 e.printStackTrace();
 }
 }

 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable

 if ("queryLocalInterface".equals(method.getName())) {

 Log.d(TAG, "hook queryLocalInterface");

 // 这里直接返回真正被Hook掉的Service接口
 // 这里的 queryLocalInterface 就不是原本的意思了
 // 我们肯定不会真的返回一个本地接口, 因为我们接管了 asInterface方法的作用
 // 因此必须是一个完整的 asInterface 过的 IInterface对象, 既要处理本地对象,也要处理
 // 这只是一个Hook点而已, 它原始的含义已经被我们重定义了; 因为我们会永远确保这个方法不
 // 让 IClipboard.Stub.asInterface 永远走到if语句的else分支里面
 return Proxy.newProxyInstance(proxy.getClass().getClassLoader(),

 // asInterface 的时候会检测是否是特定类型的接口然后进行强制转换
 // 因此这里的动态代理生成的类型信息的类型必须是正确的
 new Class[] { IBinder.class, IInterface.class, this.iinterface },
 new BinderHookHandler(base, stub));
 }

 Log.d(TAG, "method:" + method.getName());
 return method.invoke(base, args);
 }
}

```

我们使用动态代理的方式伪造了一个跟原始IBinder一模一样的对象，然后在这个伪造的IBinder对象的queryLocalInterface方法里面返回了我们第一步创建的伪造过的系统服务对象；注意看注释，详细解释可以看代码

## 替换ServiceManager的IBinder对象

现在就是万事具备，只欠东风了；我们使用反射的方式修改ServiceManager类里面缓存的Binder对象，使得getService方法返回我们伪造的IBinder对象，进而asInterface方法使用伪造IBinder对象的queryLocalInterface方法返回了我们伪造的系统服务对象。代码较简单，如下：

```
final String CLIPBOARD_SERVICE = "clipboard";

// 下面这一段的意思实际就是：ServiceManager.getService("clipboard");
// 只不过ServiceManager这个类是@hide的
Class<?> serviceManager = Class.forName("android.os.ServiceManager");
Method getService = serviceManager.getDeclaredMethod("getService", String.class);
// ServiceManager里面管理的原始的Clipboard Binder对象
// 一般来说这是一个Binder代理对象
IBinder rawBinder = (IBinder) getService.invoke(null, CLIPBOARD_SERVICE);

// Hook 掉这个Binder代理对象的queryLocalInterface方法
// 然后在queryLocalInterface返回一个IInterface对象，hook掉我们感兴趣的方法即可。
IBinder hookedBinder = (IBinder) Proxy.newProxyInstance(serviceManager.getClassLoader(),
 new Class<?>[] { IBinder.class },
 new BinderProxyHookHandler(rawBinder));

// 把这个hook过的Binder代理对象放进ServiceManager的cache里面
// 以后查询的时候会优先查询缓存里面的Binder，这样就会使用被我们修改过的Binder了
Field cacheField = serviceManager.getDeclaredField("sCache");
cacheField.setAccessible(true);
Map<String, IBinder> cache = (Map) cacheField.get(null);
cache.put(CLIPBOARD_SERVICE, hookedBinder);
```

接下来，在app里面使用剪切版，比如长按进行粘贴之后，剪切版的内容永远都是you are hooked了；这样，我们Hook系统服务的目的宣告完成！详细的代码参见[github](#)。

也许你会问，插件框架会这么hook吗？如果不是那么插件框架hook这些干什么？插件框架当然不会做替换文本这么无聊的事情，DroidPlugin插件框架管理插件使得插件就像是主程序一样，因此插件需要使用主程序的剪切版，插件之间也会共用剪切版；其他的一些系统服务也类似，这样就可以达到插件和宿主程序之间的天衣无缝，水乳交融！另

外，ActivityManager 以及 PackageManager 这两个系统服务虽然也可以通过这种方式 hook，但是由于它们的重要性和特殊性，DroidPlugin 使用了另外一种方式，我们会单独讲解。

喜欢就点个赞吧～持续更新，请关注github项目 [understand-plugin-framework](#) 和[我的博客](#)！

# Android 插件化原理解析——Hook机制之AMS&PMS

来源:[Weishu's Notes](#)

在前面的文章中我们介绍了DroidPlugin的Hook机制，也就是**代理方式**和**Binder Hook**；插件框架通过AOP实现了插件使用和开发的透明性。在讲述DroidPlugin如何实现四大组件的插件化之前，有必要说明一下它对ActivityManagerService以及PackageManagerService的Hook方式（以下简称AMS， PMS）。

ActivityManagerService对于FrameWork层的重要性不言而喻，Android的四大组件无一不与它打交道：

- `startActivity` 最终调用了AMS的 `startActivity` 系列方法，实现了Activity的启动；Activity的生命周期回调，也在AMS中完成；
- `startService` , `bindService` 最终调用到AMS的 `startService` 和 `bindService` 方法；
- 动态广播的注册和接收在AMS中完成（静态广播在**PMS**中完成）
- `getContentResolver` 最终从AMS的 `getContentProvider` 获取到 `ContentProvider`

而PMS则完成了诸如权限校检(`checkPermission` , `checkUidPermission`)，Apk meta信息获取(`getApplicationInfo` 等)，四大组件信息获取(query系列方法)等重要功能。

在上文[Android插件化原理解析——Hook机制之Binder Hook](#)中讲述了DroidPlugin的Binder Hook机制；我们知道AMS和PMS就是以Binder方式提供给应用程序使用的系统服务，理论上我们也可以采用这种方式Hook掉它们。但是由于这两者使用得如此频繁，Framework给他们了一些“特别优待”，这也给了我们相对于Binder Hook更加稳定可靠的hook方式。

阅读本文之前，可以先clone一份[understand-plugin-framework](#)，参考此项目的`ams-pms-hook`模块。另外，插件框架原理解析系列文章见[索引](#)。

## AMS获取过程

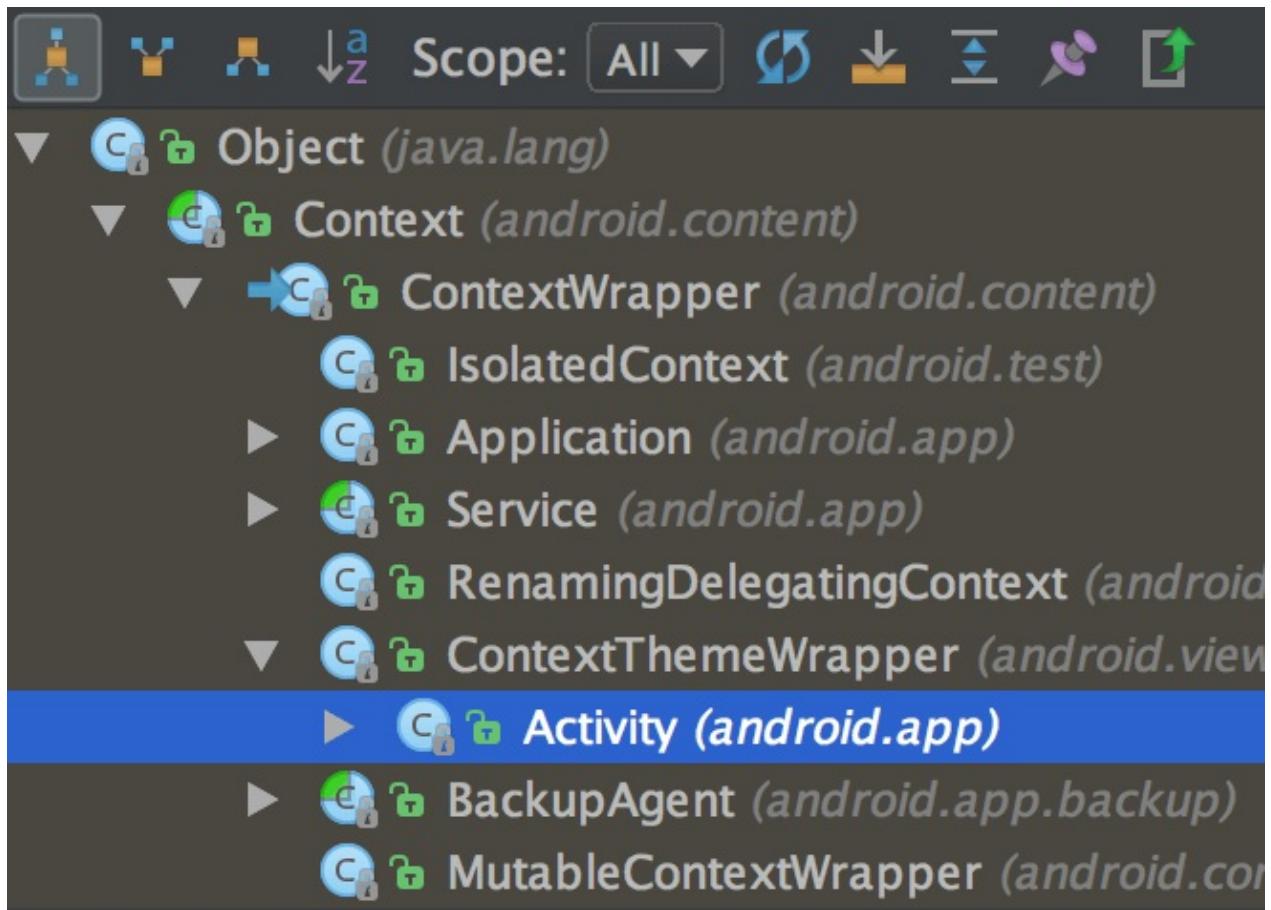
前文提到Android的四大组件无一不与AMS相关，也许读者还有些疑惑；这里我就挑一个例子，依据Android源码来说明，一个简单的 `startActivity` 是如何调用AMS最终通过IPC到 `system_server` 的。

不论读者是否知道，我们使用`startActivity`有两种形式：

- 直接调用Context类的startActivity方法；这种方式启动的Activity没有Activity栈，因此不能以standard方式启动，必须加上**FLAG\_ACTIVITY\_NEW\_TASK**这个Flag。
- 调用被Activity类重载过的startActivity方法，通常在我们的Activity中直接调用这个方法就是这种形式；

## Context.startActivity

我们查看Context类的 `startActivity` 方法，发现这竟然是一个抽象类；查看Context的类继承关系图如下：



我们看到诸如 `Activity`，`Service` 等并没有直接继承 `Context`，而是继承了 `ContextWrapper`；继续查看 `ContextWrapper` 的实现：

```

@Override
public void startActivity(Intent intent) {
 mBase.startActivity(intent);
}

```

WTF!! 果然人如其名，只是一个wrapper而已；这个mBase是什么呢？这里我先直接告诉你，它的真正实现是 `ContextImpl` 类；至于为什么，有一条思路：`mBase` 是在 `contextWrapper` 构造的时候传递进来的，那么在 `contextWrapper` 构造的时候可以找到

## 答案

什么时候会构造 ContextWrapper 呢？它的子类 Application , Service 等被创建的时候。

可以在App的主线程 ActivityThread 的 performLaunchActivity 方法里面找到答案；更详细的解析可以参考老罗的 [Android应用程序启动过程源代码分析](#)

好了，我们姑且当作已经知道 Context.startActivity 最终使用了 ContextImpl 里面的方法，代码如下：

```
public void startActivity(Intent intent, Bundle options) {
 warnIfCallingFromSystemProcess();
 if ((intent.getFlags() & Intent.FLAG_ACTIVITY_NEW_TASK) == 0) {
 throw new AndroidRuntimeException(
 "Calling startActivity() from outside of an Activity "
 + " context requires the FLAG_ACTIVITY_NEW_TASK flag."
 + " Is this really what you want?");
 }
 mMainThread.getInstrumentation().execStartActivity(
 getOuterContext(), mMainThread.getApplicationThread(), null,
 (Activity)null, intent, -1, options);
}
```

代码相当简单；我们知道了两件事：

- 其一，我们知道了在Service等非Activity的Context里面启动Activity为什么需要添加 FLAG\_ACTIVITY\_NEW\_TASK ；
- 其二，真正的 startActivity 使用了 Instrumentation 类的 execStartActivity 方法；继续跟踪：

```

public ActivityResult execStartActivity(
 Context who, IBinder contextThread, IBinder token, Activity target,
 Intent intent, int requestCode, Bundle options) {
 // ... 省略无关代码
 try {
 intent.migrateExtraStreamToClipData();
 intent.prepareToLeaveProcess();
 // -----look here!!!!!!!!!!!!!!
 int result = ActivityManagerNative.getDefault()
 .startActivity(whoThread, who.getPackageName(), intent,
 intent.resolveTypeIfNeeded(who.getContentResolver()),
 token, target != null ? target.mEmbeddedID : null,
 requestCode, 0, null, null, options);
 checkStartActivityResult(result, intent);
 } catch (RemoteException e) {
 }
 return null;
}

```

到这里我们发现真正调用的是 `ActivityManagerNative` 的 `startActivity` 方法；如果你不清楚 `ActivityManager`，`ActivityManagerService` 以及 `ActivityManagerNative` 之间的关系；建议先仔细阅读我之前关于 Binder 的文章 [Binder 学习指南](#)。

## Activity.startActivity

`Activity` 类的 `startActivity` 方法相比 `Context` 而言直观了很多；这个 `startActivity` 通过若干次调用辗转到达 `startActivityForResult` 这个方法，在这个方法内部有如下代码：

```

Instrumentation.ActivityResult ar =
 mInstrumentation.execStartActivity(
 this, mMainThread.getApplicationThread(), mToken, this,
 intent, requestCode, options);

```

可以看到，其实通过 `Activity` 和 `ContextImpl` 类启动 `Activity` 并无本质不同，他们都通过 `Instrumentation` 这个辅助类调用到了 `ActivityManagerNative` 的方法。

## Hook AMS

OK，我们现在知道；其实 `startActivity` 最终通过 `ActivityManagerNative` 这个方法远程调用了 AMS 的 `startActivity` 方法。那么这个 `ActivityManagerNative` 是什么呢？

`ActivityManagerNative` 实际上就是 `ActivityManagerService` 这个远程对象的 Binder 代理对象；每次需要与 AMS 打交道的时候，需要借助这个代理对象通过驱动进而完成 IPC 调用。

我们继续看 `ActivityManagerNative` 的 `getDefault()` 方法做了什么：

```
static public IActivityManager getDefault() {
 return gDefault.get();
}
```

`gDefault` 这个静态变量的定义如下：

```
private static final Singleton<IActivityManager> gDefault = new
 Singleton<IActivityManager>() {
 protected IActivityManager create() {
 IBinder b = ServiceManager.getService("activity"
 IActivityManager am = asInterface(
 return am;
 }
};
```

由于整个Framework与AMS打交道是如此频繁，framework使用了一个单例把这个AMS的代理对象保存了起来；这样只要需要与AMS进行IPC调用，获取这个单例即可。这是AMS这个系统服务与其他普通服务的不同之处，也是我们不通过Binder Hook的原因——我们只需要简单地Hook掉这个单例即可。

这里还有一点小麻烦：Android不同版本之间对于如何保存这个单例的代理对象是不同的；Android 2.x系统直接使用了一个简单的静态变量存储，Android 4.x以上抽象出了一个Singleton类；具体的差异可以使用grepcode进行比较：[差异](#)

我们以4.x以上的代码为例说明如何Hook掉AMS；方法使用的动态代理，如果有不理解的，可以参考之前的系列文章[Android插件化原理解析——Hook机制之动态代理](#)

```

Class<?> activityManagerNativeClass = Class.forName("android.app.ActivityManagerNative");

// 获取 gDefault 这个字段，想办法替换它
Field gDefaultField = activityManagerNativeClass.getDeclaredField("gDefault");
gDefaultField.setAccessible(true);
Object gDefault = gDefaultField.get(null);

// 4.x以上的gDefault是一个 android.util.Singleton对象；我们取出这个单例里面的字段
Class<?> singleton = Class.forName("android.util.Singleton");
Field mInstanceField = singleton.getDeclaredField("mInstance");
mInstanceField.setAccessible(true);

// ActivityManagerNative 的gDefault对象里面原始的 IActivityManager对象
Object rawIActivityManager = mInstanceField.get(gDefault);

// 创建一个这个对象的代理对象，然后替换这个字段，让我们的代理对象帮忙干活
Class<?> iActivityManagerInterface = Class.forName("android.app.IActivityManager");

Object proxy = Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
 new Class<?>[] { iActivityManagerInterface },
 new IActivityManagerHandler(rawIActivityManager));

mInstanceField.set(gDefault, proxy);

```

好了，我们hook成功之后启动Activity看看会发生什么：

```

D/HookHelper : hey, baby; you are hook!!
D/HookHelper : method:activityResumed called with args:[android.os.BinderProxy@9bc71b2]
D/HookHelper : hey, baby; you are hook!!
D/HookHelper : method:activityIdle called with args:[android.os.BinderProxy@9bc71b2, r
D/HookHelper : hey, baby; you are hook!!
D/HookHelper : method:startActivity called with args:[
 android.app.ActivityThread$ApplicationThread@17e750c,
 com.weishu.upf.ams_pms_hook.app, Intent {
 act=android.intent.action.VIEW dat=http://www.baidu.com/... }, null,
 android.os.BinderProxy@9bc71b2, null, -1, 0, null, null]
D/HookHelper : hey, baby; you are hook!!
D/HookHelper : method:activityPaused called with args:[android.os.BinderProxy@9bc71b2]

```

可以看到，简单的几行代码，AMS已经被我们完全劫持了!! 至于劫持了能干什么，自己发挥想象吧~

DroidPlugin关于AMS的Hook，可以查看IActivityManagerHook这个类，它处理了我上述所说的兼容性问题，其他原理相同。另外，也许有童鞋有疑问了，你用startActivity为例怎么能确保Hook掉这个静态变量之后就能保证所有使用AMS的入口都被Hook了呢？

答曰：无他，唯手熟尔。

Android Framework层对于四大组件的处理，调用AMS服务的时候，全部都是通过使用这种方式；若有疑问可以自行查看源码。你可以从Context类的startActivity, startService, bindService, registerBroadcastReceiver, getContentResolver 等等入口进行跟踪，最终都会发现它们都会使用ActivityManagerNative的这个AMS代理对象来完成对远程AMS的访问。

## PMS获取过程

PMS的获取也是通过Context完成的，具体就是getPackageManager这个方法；我们姑且当作已经知道了**Context的实现在ContextImpl类里面，直奔ContextImpl类的getPackageManager方法：**

```
public PackageManager getPackageManager() {
 if (mPackageManager != null) {
 return mPackageManager;
 }

 I PackageManager pm = ActivityThread.getPackageManager();
 if (pm != null) {
 // Doesn't matter if we make more than one instance.
 return (mPackageManager = new ApplicationPackageManager(this, pm));
 }
 return null;
}
```

可以看到，这里干了两件事：

- 真正的PMS的代理对象在ActivityThread类里面
- ContextImpl通过ApplicationPackageManager对它还进行了一层包装

我们继续查看ActivityThread类的getPackageManager方法，源码如下：

```
public static I PackageManager getPackageManager() {
 if (sPackageManager != null) {
 return sPackageManager;
 }
 IBinder b = ServiceManager.getService("package");
 sPackageManager = I PackageManager.Stub.asInterface(b);
 return sPackageManager;
}
```

可以看到，和AMS一样，PMS的Binder代理对象也是一个全局变量存放在一个静态字段中；我们可以如法炮制，Hook掉PMS。

现在我们的目的很明切，如果需要Hook PMS有两个地方需要Hook掉：

- ActivityThread的静态字段sPackageManager
- 通过Context类的getPackageManager方法获取到的ApplicationPackageManager对象里面的mPM字段。

## Hook PMS

现在使用代理Hook应该是轻车熟路了吧，通过上面的分析，我们Hook两个地方；代码信手拈来：

```
// 获取全局的ActivityThread对象
Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
Method currentActivityThreadMethod = activityThreadClass
 .getDeclaredMethod("currentActivityThread");
Object currentActivityThread = currentActivityThreadMethod.invoke(null);

// 获取ActivityThread里面原始的 sPackageManager
Field sPackageManagerField = activityThreadClass
 .getDeclaredField("sPackageManager");
sPackageManagerField.setAccessible(true);
Object sPackageManager = sPackageManagerField.get(currentActivityThread);

// 准备好代理对象，用来替换原始的对象
Class<?> iPackageManagerInterface = Class.forName("android.content.pm.IPackageManager");
Object proxy = Proxy.newProxyInstance(iPackageManagerInterface.getClassLoader(),
 new Class<?>[] { iPackageManagerInterface },
 new HookHandler(sPackageManager));

// 1. 替换掉ActivityThread里面的 sPackageManager 字段
sPackageManagerField.set(currentActivityThread, proxy);

// 2. 替换 ApplicationPackageManager里面的 mPM对象
PackageManager pm = context.getPackageManager();
Field mPmField = pm.getClass().getDeclaredField("mPM");
mPmField.setAccessible(true);
mPmField.set(pm, proxy);
```

好了，Hook完毕我们验证以下结论；调用一下PMS的 `getInstalledApplications` 方法，打印日志如下：

```
03-07 15:07:27.187 8306-8306/com.weishu.upf.ams_pms_hook.app
D/IActivityManagerHandler: hey, baby; you are hook!!
03-07 15:07:27.187 8306-8306/com.weishu.upf.ams_pms_hook.app
D/IActivityManagerHandler: method:getInstalledApplications called with args:[0, 0]
```

OK，我们又成功劫持了PackageManager！！DroidPlugin 处理PMS的代码可以在 IPackageManagerHook查看。

在结束讲解PackageManager的Hook之前，我们需要说明一点；那就是Context的实现类里面没有使用静态全局变量来保存PMS的代理对象，而是每拥有一个Context的实例就持有了一个PMS代理对象的引用；所以这里有个很蛋疼的事情，那就是我们如果想要完全 Hook住PMS，需要精确控制整个进程内部创建的Context对象；所幸，插件框架中，插件的Activity，Service，ContentProvider，Broadcast等所有使用到Context的地方，都是由框架控制创建的；因此我们要小心翼翼地替换掉所有这些对象持有的PMS代理对象。

我前面也提到过，静态变量和单例都是良好的Hook点，这里很好地反证了这句话：**想要 Hook掉一个实例变量该是多么麻烦！**

## 小结

写到这里，关于DroidPlugin的Hook技术的讲解已经完结了；我相信读者或多或少地认识到，其实Hook并不是一项神秘的技术；一个干净，透明的框架少不了AOP，而AOP也少不了Hook。

我所讲解的Hook仅仅使用反射和动态代理技术，更加强大的Hook机制可以进行字节码编织，比如J2EE广泛使用了cglib和asm进行AOP编程；而Android上现有的插件框架还是加载编译时代码，采用动态生成类的技术理论上也是可行的；之前有一篇文章[Android动态加载黑科技 动态创建Activity模式](#)，就讲述了这种方式；现在全球的互联网公司不排除有用这种技术实现插件框架的可能；我相信不远的未来，这种技术也会在Android上大放异彩。

了解完Hook技术之后，接下来的系列文章会讲述DroidPlugin对Android四大组件在插件系统上的处理，插件框架对于这一部分的实现是DroidPlugin的精髓，Hook只不过是工具而已。学习这部分内容需要对于Activity，Service，Broadcast以及ContentProvider的工作机制有一定的了解，因此我也会在必要的时候穿插讲解一些Android Framework的知识；我相信这一定会对读者大有裨益。

喜欢就点个赞吧～持续更新，请关注github项目 [understand-plugin-framework](#)和我的博客！



# Android 插件化原理解析——Activity生命周期管理

来源:[Weishu's Notes](#)

之前的 [Android插件化原理解析](#) 系列文章揭开了Hook机制的神秘面纱，现在我们手握倚天屠龙，那么如何通过这种技术完成插件化方案呢？具体来说，插件中的Activity，Service等组件如何在Android系统上运行起来？

在Java平台要做到动态运行模块、热插拔可以使用 `ClassLoader` 技术进行动态类加载，比如广泛使用的 `OSGi` 技术。在Android上当然也可以使用动态加载技术，但是仅仅把类加载进来就足够了吗？Activity，Service等组件是有生命周期的，它们统一由系统服务AMS管理；使用ClassLoader可以从插件中创建Activity对象，但是，一个没有生命周期的Activity对象有什么用？所以在Android系统上，仅仅完成动态类加载是不够的；我们需要想办法把我们加载进来的Activity等组件交给系统管理，让AMS赋予组件生命周期；这样才算是一个有血有肉的完善的插件化方案。

接下来的系列文章会讲述 DroidPlugin对于Android四大组件的处理方式，我们且看它如何采用Hook技术坑蒙拐骗把系统玩弄于股掌之中，最终赋予Activity，Service等组件生命周期，完成借尸还魂的。

首先，我们来看看DroidPlugin对于Activity组件的处理方式。

阅读本文之前，可以先clone一份 [understand-plugin-framework](#)，参考此项目的intercept-activity模块。另外，如果对于Hook技术不甚了解，请先查阅我之前的文章：

- [Android插件化原理解析——Hook机制之动态代理](#)
- [Android插件化原理解析——Hook机制之Binder Hook](#)
- [Android 插件化原理解析——Hook机制之AMS&PMS](#)

## AndroidManifest.xml的限制

读到这里，或许有部分读者觉得疑惑了，启动Activity不就是一个`startActivity`的事吗，有这么神秘兮兮的？

启动Activity确实非常简单，但是Android却有一个限制：**必须在AndroidManifest.xml中显式声明使用的Activity**；我相信读者肯定会遇到下面这种异常：

```
03-18 15:29:56.074 20709-20709/com.weishu.intercept_activity.app E/AndroidRuntime: F
Process: com.weishu.intercept_activity.app, PID: 20709
 android.content.ActivityNotFoundException: Unable to find explicit activity class
 {com.weishu.intercept_activity.app/com.weishu.intercept_activity.app.TargetActivi
 have you declared this activity in your AndroidManifest.xml?
```

『必须在AndroidManifest.xml中显示声明使用的Activity』这个硬性要求很大程度上限制了插件系统的发挥：假设我们需要启动一个插件的Activity，插件使用的Activity是无法预知的，这样肯定也不会在Manifest文件中声明；如果插件新添加一个Activity，主程序的AndroidManifest.xml就需要更新；既然双方都需要修改升级，何必要使用插件呢？这已经违背了动态加载的初衷：不修改插件框架而动态扩展功能。

能不能想办法绕过这个限制呢？

束手无策啊，怎么办？借刀杀人偷梁换柱无中生有以逸待劳乘火打劫瞒天过海...等等！偷梁换柱瞒天过海？貌似可以一试。

我们可以耍个障眼法：既然AndroidManifest文件中必须声明，那么我就声明一个（或者有限个）替身Activity好了，当需要启动插件的某个Activity的时候，先让系统以为启动的是AndroidManifest中声明的那个替身，暂时骗过系统；然后到合适的时候又替换回我们需要启动的真正的Activity；所谓瞒天过海，莫过如此！

现在有了方案了，但是该如何做呢？兵书又说，知己知彼百战不殆！如果连Activity的启动过程都不熟悉，怎么完成这个瞒天过海的过程？

## Activity启动过程

启动Activity非常简单，一个startActivity就完事了；那么在这个简单调用的背后发生了什么呢？Look the fucking source code！

关于Activity的启动过程，也不是三言两语能解释清楚的，如果按照源码一步一步走下来，插件化系列文章就不用写了；所以这里我就给出一个大致流程，只列出关键的调用点（以Android 6.0源码为例）；如果读者希望更详细的讲解，可以参考老罗的 [Android应用程序的Activity启动过程简要介绍和学习计划](#)

首先是Activity类的startActivity方法：

```
public void startActivityForResult(Intent intent) {
 startActivityForResult(intent, null);
}
```

跟着这个方法一步一步跟踪，会发现它最后在 `startActivityForResult` 里面调用了 `Instrument` 对象的 `execStartActivity` 方法；接着在这个函数里面调用了 `ActivityManagerNative` 类的 `startActivity` 方法；这个过程在前文已经反复举例讲解了，我们知道接下来会通过 Binder IPC 到 AMS 所在进程调用 **AMS** 的 `startActivity` 方法；Android 系统的组件生命周期管理就是在 AMS 里面完成的，那么在 AMS 里面到底做了什么呢？

`ActivityManagerService` 的 `startActivity` 方法如下：

```
public final int startActivity(IAplicationThread caller, String callingPackage,
 Intent intent, String resolvedType, IBinder resultTo,
 String resultWho, int requestCode, int startFlags,
 String profileFile, ParcelFileDescriptor profileFd, Bundle options) {
 return startActivityAsUser(caller, callingPackage, intent, resolvedType, resultTo,
 resultWho, requestCode,
 startFlags, profileFile, profileFd, options, UserHandle.getCallingUserId()
 }
}
```

很简单，直接调用了 `startActivityAsUser` 这个方法；接着是 `ActivityStackSupervisor` 类的 `startActivityMayWait` 方法。这个 `ActivityStackSupervisor` 类到底是个啥？如果仔细查阅，低版本的 Android 源码上是没有这个类的；后来 AMS 的代码进行了部分重构，关于 Activity 栈管理的部分单独提取出来成为了 `ActivityStackSupervisor` 类；好了，继续看代码。

`startActivityMayWait` 这个方法前面对参数进行了一系列处理，我们需要知道的是，在这个方法内部对传进来的 Intent 进行了解析，并尝试从中取出关于启动 Activity 的信息。

然后这个方法调用了 `startActivityLocked` 方法；在 `startActivityLocked` 方法内部进行了一系列重要的检查：比如权限检查，`Activity` 的 `exported` 属性检查等等；我们上文所述的，启动没有在 Manifestfest 中显示声明的 Activity 抛异常也是这里发生的：

```
if (err == ActivityManager.START_SUCCESS && aInfo == null) {
 // We couldn't find the specific class specified in the Intent.
 // Also the end of the line.
 err = ActivityManager.START_CLASS_NOT_FOUND;
}
```

这里返回 **ActivityManager.START\_CLASS\_NOT\_FOUND** 之后，在 `Instrument` 的 `execStartActivity` 返回之后会检查这个值，然后跑出异常：

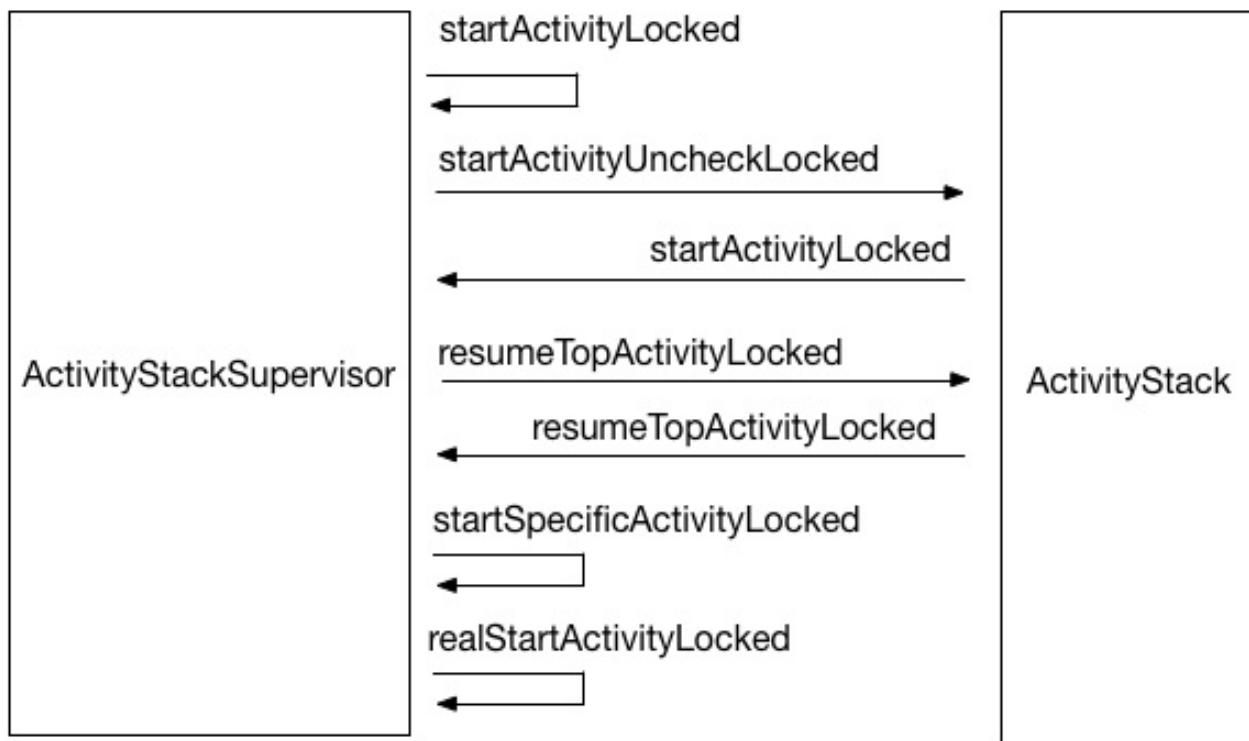
```

case ActivityManager.START_CLASS_NOT_FOUND:
 if (intent instanceof Intent && ((Intent)intent).getComponent() != null)
 throw new ActivityNotFoundException(
 "Unable to find explicit activity class "
 + ((Intent)intent).getComponent().toShortString()
 + "; have you declared this activity in your AndroidManifest.xml?");

```

源码看到这里，我们已经确认了『必须在AndroidManifest.xml中显示声明使用的Activity』的原因；然而这个校检过程发生在AMS所在的进程 `system_server`，我们没有办法篡改，只能另寻他路。

OK，我们继续跟踪源码；在 `startActivityLocked` 之后处理的都是Activity任务栈相关内容，这一系列 `ActivityStack` 和 `ActivityStackSupervisor` 纠缠不清的调用看下图就明白了；不明白也没关系：D 目前用处不大。



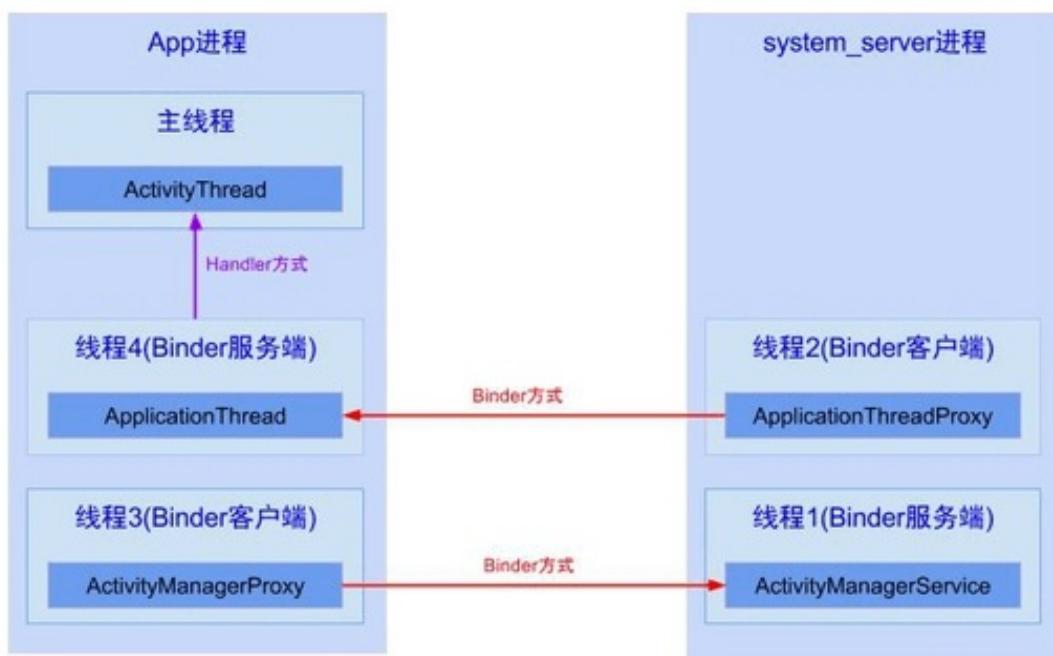
这一系列调用最终到达了 `ActivityStackSupervisor` 的 `realStartActivityLocked` 方法；人如其名，这个方法开始了真正的“启动Activity”：它调用了 `ApplicationThread` 的 `scheduleLaunchActivity` 方法，开始了真正的Activity对象创建以及启动过程。

这个 `ApplicationThread` 是什么，是一个线程吗？与 `ActivityThread` 有什么区别和联系？

不要被名字迷惑了，这个 `ApplicationThread` 实际上是一个 `Binder` 对象，是 App 所在的进程与 AMS 所在进程 `system_server` 通信的桥梁；在 Activity 启动的过程中，App 进程会频繁地与 AMS 进程进行通信：

- App进程会委托AMS进程完成Activity生命周期的管理以及任务栈的管理；这个通信过程AMS是Server端，App进程通过持有AMS的client代理ActivityManagerNative完成通信过程；
- AMS进程完成生命周期管理以及任务栈管理后，会把控制权交给App进程，让App进程完成Activity类对象的创建，以及生命周期回调；这个通信过程也是通过Binder完成的，App所在server端的Binder对象存在于ActivityThread的内部类ApplicationThread；AMS所在client通过持有IApplicationThread的代理对象完成对于App进程的通信。

App进程与AMS进程的通信过程如图所示：



App进程内部的ApplicationThread server端内部有自己的Binder线程池，它与App主线程的通信通过Handler完成，这个Handler存在于ActivityThread类，它的名字很简单就叫H，这一点我们接下来就会讲到。

现在我们明白了这个ApplicationThread到底是个什么东西，接上文继续跟踪Activity的启动过程；我们查看ApplicationThread的 `scheduleLaunchActivity` 方法，这个方法很简单，就是包装了参数最终使用Handler发了一个消息。

正如刚刚所说，ApplicationThread所在的Binder服务端使用Handler与主线程进行通信，这里的`scheduleLaunchActivity`方法直接把启动Activity的任务通过一个消息转发给了主线程；我们查看Handler类对于这个消息的处理：

```

race.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityStart");
ActivityClientRecord r = (ActivityClientRecord)msg.obj;

r.packageInfo = getPackageInfoNoCheck(
 r.activityInfo.applicationInfo, r.compatInfo);
handleLaunchActivity(r, null);
Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);

```

可以看到，这里直接调用了ActivityThread的 handleLaunchActivity 方法，在这个方法内部有一句非常重要：

```
Activity a = performLaunchActivity(r, customIntent);
```

绕了这么多弯，我们的Activity终于被创建出来了！继续跟踪这个performLaunchActivity方法看看发生了什么；由于这个方法较长，我就不贴代码了，读者可以自行查阅；要指出的是，这个方法做了两件很重要的事情：

- 使用ClassLoader加载并通过反射创建Activity对象

```

java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
activity = mInstrumentation.newActivity(
 cl, component.getClassName(), r.intent);
StrictMode.incrementExpectedActivityCount(activity.getClass());
r.intent.setExtrasClassLoader(cl);

```

- 如果Application还没有创建，那么创建Application对象并回调相应的生命周期方法；

```

Application app = r.packageInfo.makeApplication(false, mInstrumentation);
// ... 省略

if (r.isPersistable()) {
 mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);
} else {
 mInstrumentation.callActivityOnCreate(activity, r.state);
}

```

Activity的启动过程到这里就结束了，可能读者还是觉得迷惑：不就是调用了一系列方法吗？具体做了什么还是不太清楚，而且为什么Android要这么设计？

方法调用链再长也木有关系，有两点需要明白：

- 平时我们所说的Application被创建了，onCreate方法被调用了，我们或许并没有意识到我们所说的Application，Activity除了代表Android应用层通常所代表的“组件”之外，

它们其实都是普通的Java对象，也是需要被构造函数构造出来的对象的；在这个过程中，我们明白了这些对象到底是如何被创建的。

- 为什么需要一直与AMS进行通信？哪些操作是在AMS中进行的？其实AMS正如名字所说，管理所有的“活动”，整个系统的Activity堆栈，Activity生命周期回调都是由AMS所在的系统进程system\_server帮开发者完成的；Android的Framework层帮忙完成了诸如生命周期管理等繁琐复杂的过程，简化了应用层的开发。

## 瞒天过海——启动不在AndroidManifest.xml中声明的Activity

### 简要分析

通过上文的分析，我们已经对Activity的启动过程了如指掌了；就让我们干点坏事吧 :D

对与『必须在AndroidManifest.xml中显示声明使用的Activity』这个问题，上文给出了思路——瞒天过海；我们可以在AndroidManifest.xml里面声明一个替身Activity，然后在合适的时候把这个假的替换成我们真正需要启动的Activity就OK了。

那么问题来了，『合适的时候』到底是什么时候？在前文[Hook机制之动态代理](#)中我们提到过Hook过程最重要的一步是寻找Hook点；如果是在同一个进程，startActivity到Activity真正启动起来这么长的调用链，我们随便找个地方Hook掉就完事儿了；但是问题木有这么简单。

Activity启动过程中很多重要的操作（正如上文分析的『必须在AndroidManifest.xml中显式声明要启动的Activity』）都不是在App进程里面执行的，而是在AMS所在的系统进程system\_server完成，由于进程隔离的存在，我们对别的进程无能为力；所以这个Hook点就需要花点心思了。

这时候Activity启动过程的知识就派上用场了；虽然整个启动过程非常复杂，但其实一张图就能总结：



先从App进程调用startActivity；然后通过IPC调用进入系统进程system\_server，完成Activity管理以及一些校检工作，最后又回到了APP进程完成真正的Activiity对象创建。

由于这个检验过程是在AMS进程完成的，我们对system\_server进程里面的操作无能为力，只有在我们APP进程里面执行的过程才是有可能被Hook掉的，也就是第一步和第三步；具体应该怎么办呢？

既然需要一个显式声明的Activity，那就声明一个！可以在第一步假装启动一个已经在**AndroidManifest.xml**里面声明过的替身Activity，让这个Activity进入AMS进程接受检验；最后在第三步的时候换成我们真正需要启动的Activity；这样就成功欺骗了AMS进程，瞒天过海！

说到这里，是不是有点小激动呢？我们写个demo验证一下：『启动一个并没有在AndroidManifest.xml中显示声明的Activity』

## 实战过程

具体来说，我们打算实现如下功能：在MainActivity中启动一个并没有在AndroidManifest.xml中声明的TargetActivity；按照上文分析，我们需要声明一个替身Activity，我们叫它StubActivity；

那么，我们的AndroidManifest.xml如下：

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 package="com.weishu.intercept_activity.app">

 <application
 android:allowBackup="true"
 android:label="@string/app_name"
 android:icon="@mipmap/ic_launcher"
 >

 <activity android:name=".MainActivity">
 <intent-filter>
 <action android:name="android.intent.action.MAIN"/>
 <category android:name="android.intent.category.LAUNCHER"/>
 </intent-filter>
 </activity>

 <!-- 替身Activity, 用来欺骗AMS -->
 <activity android:name=".StubActivity"/>

 </application>

</manifest>

```

OK，那么我们启动TargetActivity很简单，就是个startActivity调用的事：

```
startActivity(new Intent(MainActivity.this, TargetActivity.class));
```

如果你直接这么运行，肯定会直接抛出ActivityNotFoundException然后直接退出；我们接下来要做的就是让这个调用成功启动TargetActivity。

## 狸猫换太子——使用替身Activity绕过AMS

由于AMS进程会对Activity做显式声明验证，因此在启动Activity的控制权转移到AMS进程之前，我们需要想办法临时把TargetActivity替换成替身StubActivity；在这之间有很长的一段调用链，我们可以轻松Hook掉；选择什么地方Hook是一个很自由的事情，但是Hook的步骤越后越可靠——Hook得越早，后面的调用就越复杂，越容易出错。

我们可以选择在进入AMS进程的入口进行Hook，具体来说也就是Hook AMS在本进程的代理对象ActivityManagerNative。如果你不知道如何Hook掉这个AMS的代理对象，请查阅我之前的文章 [Hook机制之AMS&PMS](#)

我们Hook掉ActivityManagerNative对于startActivity方法的调用，替换掉交给AMS的intent对象，将里面的TargetActivity的暂时替换成已经声明好的替身StubActivity；这种Hook方式前文讲述的很详细，不赘述；替换的关键代码如下：

```
if ("startActivity".equals(method.getName())) {
 // 只拦截这个方法
 // 替换参数，任你所为；甚至替换原始Activity启动别的Activity偷梁换柱
 // API 23:
 // public final Activity startActivityNow(Activity parent, String id,
 // Intent intent, ActivityInfo activityInfo, IBinder token, Bundle state,
 // Activity.NonConfigurationInstances lastNonConfigurationInstances) {

 // 找到参数里面的第一个Intent 对象

 Intent raw;
 int index = 0;

 for (int i = 0; i < args.length; i++) {
 if (args[i] instanceof Intent) {
 index = i;
 break;
 }
 }
 raw = (Intent) args[index];

 Intent newIntent = new Intent();

 // 这里包名直接写死，如果再插件里，不同的插件有不同的包 传递插件的包名即可
 String targetPackage = "com.weishu.intercept_activity.app";

 // 这里我们把启动的Activity临时替换为 StubActivity
 ComponentName componentName = new ComponentName(targetPackage, StubActivity.class
newIntent.setComponent(componentName);

 // 把我们原始要启动的TargetActivity先存起来
 newIntent.putExtra(HookHelper.EXTRA_TARGET_INTENT, raw);

 // 替换掉Intent，达到欺骗AMS的目的
 args[index] = newIntent;

 Log.d(TAG, "hook success");
 return method.invoke(mBase, args);
 }

 return method.invoke(mBase, args);
```

Binder驱动的消息，开始执行ActivityManagerService里面真正的startActivity方法；这时候AMS看到的intent参数里面的组件已经是StubActivity了，因此可以成功绕过检查，这时候如果不做后面的Hook，直接调用

```
startActivity(new Intent(MainActivity.this, TargetActivity.class));
```

也不会出现上文的ActivityNotFoundException

## 借尸还魂——拦截Callback从恢复真身

行百里者半九十。现在我们的startActivity启动一个没有显式声明的Activity已经不会抛异常了，但是要真正正确地把TargetActivity启动起来，还有一些事情要做。其中最重要的一点是，我们用替身StubActivity临时换了TargetActivity，肯定需要在『合适的』时候替换回来；接下来我们就完成这个过程。

在AMS进程里面我们是没有办法换回来的，因此我们要等AMS把控制权交给App所在进程，也就是上面那个『Activity启动过程简图』的第三步。AMS进程转移到App进程也是通过Binder调用完成的，承载这个功能的Binder对象是IApplicationThread；在App进程它是Server端，在Server端接受Binder远程调用的是Binder线程池，Binder线程池通过Handler将消息转发给App的主线程；（我这里不厌其烦地叙述Binder调用过程，希望读者不要反感，其一加深印象，其二懂Binder真的很重要）我们可以在这个**Handler**里面将替身恢复成真身。

这里不打算讲述Handler 的原理，我们简单看一下Handler是如何处理接收到的Message的，如果我们能拦截这个Message的接收过程，就有可能完成替身恢复工作；Handler类的dispatchMessage如下：

```
public void dispatchMessage(Message msg) {
 if (msg.callback != null) {
 handleCallback(msg);
 } else {
 if (mCallback != null) {
 if (mCallback.handleMessage(msg)) {
 return;
 }
 }
 handleMessage(msg);
 }
}
```

从这个方法可以看出来，Handler类消息分发的过程如下：

- 如果传递的Message本身就有callback，那么直接使用Message对象的callback方法；
- 如果Handler类的成员变量mCallback存在，那么首先执行这个mCallback回调；
- 如果mCallback的回调返回true，那么表示消息已经成功处理；直接结束。
- 如果mCallback的回调返回false，那么表示消息没有处理完毕，会继续使用Handler类的handleMessage方法处理消息。

那么，ActivityThread中的Handler类H是如何实现的呢？H的部分源码如下：

```
public void handleMessage(Message msg) {
 if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
 switch (msg.what) {
 case LAUNCH_ACTIVITY: {
 Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityStart");
 ActivityClientRecord r = (ActivityClientRecord)msg.obj;

 r.packageInfo = getPackageInfoNoCheck(
 r.activityInfo.applicationInfo, r.compatInfo);
 handleLaunchActivity(r, null);
 Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
 } break;
 case RELAUNCH_ACTIVITY: {
 Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityRestart");
 ActivityClientRecord r = (ActivityClientRecord)msg.obj;
 handleRelaunchActivity(r);
 Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);

 // 以下略
 }
 }
}
```

可以看到H类仅仅重载了handleMessage方法；通过dispathMessage的消息分发过程得知，我们可以拦截这一过程：把这个H类的mCallback替换为我们的自定义实现，这样dispathMessage就会首先使用这个自定义的mCallback，然后看情况使用H重载的handleMessage。

这个Handler.Callback是一个接口，我们可以使用动态代理或者普通代理完成Hook，这里我们使用普通的静态代理方式；创建一个自定义的Callback类：

```
/* package */ class ActivityThreadHandlerCallback implements Handler.Callback {

 Handler mBase;

 public ActivityThreadHandlerCallback(Handler base) {
 mBase = base;
 }
}
```

```

@Override
public boolean handleMessage(Message msg) {

 switch (msg.what) {
 // ActivityThread里面 "LAUNCH_ACTIVITY" 这个字段的值是100
 // 本来使用反射的方式获取最好，这里为了简便直接使用硬编码
 case 100:
 handleLaunchActivity(msg);
 break;
 }

 mBase.handleMessage(msg);
 return true;
}

private void handleLaunchActivity(Message msg) {
 // 这里简单起见，直接取出TargetActivity;

 Object obj = msg.obj;
 // 根据源码：
 // 这个对象是 ActivityClientRecord 类型
 // 我们修改它的intent字段为我们原来保存的即可。
/*
 switch (msg.what) {
 / case LAUNCH_ACTIVITY: {
 / Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityStart");
 / final ActivityClientRecord r = (ActivityClientRecord) msg.obj;
 /
 / r.packageInfo = getPackageInfoNoCheck(
 / r.activityInfo.applicationInfo, r.compatInfo);
 / handleLaunchActivity(r, null);
 */
}

try {
 // 把替身恢复成真身
 Field intent = obj.getClass().getDeclaredField("intent");
 intent.setAccessible(true);
 Intent raw = (Intent) intent.get(obj);

 Intent target = raw.getParcelableExtra(HookHelper.EXTRA_TARGET_INTENT);
 raw.setComponent(target.getComponent());

} catch (NoSuchFieldException e) {
 e.printStackTrace();
} catch (IllegalAccessException e) {
 e.printStackTrace();
}
}
}

```

这个Callback类的使命很简单：把替身**StubActivity**恢复成真身**TargetActivity**；有了这个自定义的Callback之后我们需要把ActivityThread里面处理消息的Handler类H的的mCallback修改为自定义callback类的对象：

```
// 先获取到当前的ActivityThread对象
Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
Field currentActivityThreadField = activityThreadClass.getDeclaredField("sCurrentActivityThread");
currentActivityThreadField.setAccessible(true);
Object currentActivityThread = currentActivityThreadField.get(null);

// 由于ActivityThread一个进程只有一个，我们获取这个对象的mH
Field mHField = activityThreadClass.getDeclaredField("mH");
mHField.setAccessible(true);
Handler mH = (Handler) mHField.get(currentActivityThread);

// 设置它的回调，根据源码：
// 我们自己给他设置一个回调，就会替代之前的回调；

// public void dispatchMessage(Message msg) {
// if (msg.callback != null) {
// handleCallback(msg);
// } else {
// if (mCallback != null) {
// if (mCallback.handleMessage(msg)) {
// return;
// }
// }
// handleMessage(msg);
// }
// }

Field mCallBackField = Handler.class.getDeclaredField("mCallback");
mCallBackField.setAccessible(true);

mCallBackField.set(mH, new ActivityThreadHandlerCallback(mH));
```

到这里，我们已经成功地绕过AMS，完成了『启动没有在AndroidManifest.xml中显式声明的Activity』的过程；瞒天过海，这种玩弄系统与股掌之中的快感你们能体会到吗？

## 僵尸or活人？——能正确收到生命周期回调吗

虽然我们完成了『启动没有在AndroidManifest.xml中显式声明的Activity』，但是启动的TargetActivity是否有自己的生命周期呢，我们还需要额外的处理过程吗？

实际上TargetActivity已经是一个有血有肉的Activity了：它具有自己正常的生命周期；可以运行[Demo代码](#)验证一下。

这个过程是如何完成的呢？我们以`onDestroy`为例简要分析一下：

从Activity的`finish`方法开始跟踪，最终会通过`ActivityManagerNative`到AMS然后接着通过`ApplicationThread`到`ActivityThread`，然后通过H转发消息到`ActivityThread`的`handleDestroyActivity`，接着这个方法把任务交给`performDestroyActivity`完成。

在真正分析这个方法之前，需要说明一点的是：不知读者是否感受得到，App进程与AMS交互几乎都是这么一种模式，几个角色`ActivityManagerNative`, `ApplicationThread`, `ActivityThread`以及Handler类H分工明确，读者可以按照这几个角色的功能分析AMS的任何调用过程，屡试不爽；这也是我的初衷——希望分析插件框架的过程中能帮助深入理解Android Framework。

好了继续分析`performDestroyActivity`，关键代码如下：

```
ActivityClientRecord r = mActivities.get(token);
// ...略
mInstrumentation.callActivityOnDestroy(r.activity);
```

这里通过`mActivities`拿到了一个`ActivityClientRecord`，然后直接把这个record里面的Activity交给Instrument类完成了`onDestroy`的调用。

在我们这个demo的场景下，`r.activity`是`TargetActivity`还是`StubActivity`？按理说，由于我们欺骗了AMS，AMS应该只知道`StubActivity`的存在，它压根儿就不知道`TargetActivity`是什么，为什么它能正确完成对`TargetActivity`生命周期的回调呢？

一切的秘密在`token`里面。AMS与`ActivityThread`之间对于Activity的生命周期的交互，并没有直接使用Activity对象进行交互，而是使用一个`token`来标识，这个`token`是binder对象，因此可以方便地跨进程传递。Activity里面有一个成员变量`mToken`代表的就是它，`token`可以唯一地标识一个Activity对象，它在Activity的`attach`方法里面初始化；

在AMS处理Activity的任务栈的时候，使用这个`token`标记Activity，因此在我们的demo里面，AMS进程里面的`token`对应的是`StubActivity`，也就是AMS还在傻乎乎地操作`StubActivity`（关于这一点，你可以dump出任务栈的信息，可以观察到dump出的确实是`StubActivity`）。但是在我们App进程里面，`token`对应的却是`TargetActivity`！因此，在`ActivityThread`执行回调的时候，能正确地回调到`TargetActivity`相应的方法。

为什么App进程里面，`token`对应的是`TargetActivity`呢？

回到代码，`ActivityClientRecord`是在`mActivities`里面取出来的，确实是根据`token`取；那么这个`token`是什么时候添加进去的呢？我们看`performLaunchActivity`就明白了：它通过`classloader`加载了`TargetActivity`，然后完成一切操作之后把这个activity添加进了

mActivities！另外，在这个方法里面我们还能看到对Activityattach方法的调用，它传递给了新创建的Activity一个token对象，而这个token是在ActivityClientRecord构造函数里面初始化的。

至此我们已经可以确认，通过这种方式启动的Activity有它自己完整而独立的生命周期！

## 小节

本文讲述了『启动一个并没有在AndroidManifest.xml中显示声明的Activity』的解决办法，我们成功地绕过了Android的这个限制，这个是插件Activity管理技术的基础；但是要做到启动一个插件Activity问题远没有这么简单。

首先，在Android中，Activity有不同的启动模式；我们声明了一个替身StubActivity，肯定没有满足所有的要求；因此，我们需要在AndroidManifest.xml中声明一系列的有不同launchMode的Activity，还需要完成替身与真正Activity launchMode的匹配过程；这样才能完成启动各种类型Activity的需求，关于这一点，在DroidPlugin的com.morgoo.droidplugin.stub包下面可以找到。

另外，每启动一个插件的Activity都需要一个StubActivity，但是AndroidManifest.xml中肯定只能声明有限个，如果一直startActivity而不finish的话，那么理论上就需要无限个StubActivity；这个问题该如何解决呢？事实上，这个问题在技术上没有好的解决办法。但是，如果你的App startActivity了十几次，而没有finish任何一个Activity，这样在Activity的回退栈里面有十几个Activity，用户难道按back十几次回到主页吗？有这种需求说明你的产品设计有问题；一个App一级页面，二级页面..到五六级的页面已经影响体验了，所以，每种LaunchMode声明十个StubActivity绝对能满足需求了。

最后，在本文所述例子中，TargetActivity与StubActivity存在于同一个Apk，因此系统的ClassLoader能够成功加载并创建TargetActivity的实例。但是在实际的插件系统中，要启动的目标Activity肯定存在于一个单独的文件中，系统默认的ClassLoader无法加载插件中的Activity类——系统压根儿就不知道要加载的插件在哪，谈何加载？因此还有一个很重要的问题需要处理：

**我们要完成插件系统中类的加载，这可以通过自定义ClassLoader实现。**解决了『启动没有在AndroidManifest.xml中显式声明的，并且存在于外部文件中的Activity』的问题，插件系统对于Activity的管理才算得上是一个完全体。篇幅所限，欲知后事如何，请听下回分解！

喜欢就点个赞吧～持续更新，请关注github项目[understand-plugin-framework](#)和我的[博客](#)！



# Android 插件化原理解析——插件加载机制

来源:[Weishu's Notes](#)

上文 [Activity生命周期管理](#) 中我们地完成了『启动没有在AndroidManifest.xml中显式声明的Activity』的任务；通过Hook AMS和拦截ActivityThread中H类对于组件调度我们成功地绕过了AndroidManifest.xml的限制。

但是我们启动的『没有在AndroidManifest.xml中显式声明』的Activity和宿主程序存在于同一个Apk中；通常情况下，插件均以独立的文件存在甚至通过网络获取，这时候插件中的Activity能否成功启动呢？

要启动Activity组件肯定先要创建对应的Activity类的对象，从上文 [Activity生命周期管理](#) 知道，创建Activity类对象的过程如下：

```
java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
activity = mInstrumentation.newActivity(
 cl, component.getClassName(), r.intent);
StrictMode.incrementExpectedActivityCount(activity.getClass());
r.intent.setExtrasClassLoader(cl);
```

也就是说，系统通过ClassLoader加载了需要的Activity类并通过反射调用构造函数创建出了Activity对象。如果Activity组件存在于独立于宿主程序的文件之中，系统的ClassLoader怎么知道去哪里加载呢？因此，如果不做额外的处理，插件中的Activity对象甚至都没有办法创建出来，谈何启动？

因此，要使存在于独立文件或者网络中的插件被成功启动，首先就需要解决这个插件类加载的问题。

下文将围绕此问题展开，完成『启动没有在AndroidManifest.xml中显示声明，并且存在于外部插件中的Activity』的任务。

阅读本文之前，可以先clone一份 [understand-plugin-framework](#)，参考此项目的 classloader-hook 模块。另外，插件框架原理解析系列文章见索引。

## ClassLoader机制

或许有的童鞋还不太了解Java的ClassLoader机制，我这里简要介绍一下。

Java虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校检、转换解析和初始化的，最终形成可以被虚拟机直接使用的Java类型，这就是虚拟机的类加载机制。

与那些在编译时进行链接工作的语言不同，在Java语言里面，类型的加载、连接和初始化都是在程序运行期间完成的，这种策略虽然会令类加载时稍微增加一些性能开销，但是会为Java应用程序提供高度的灵活性，Java里天生可以动态拓展的语言特性就是依赖运行期动态加载和动态链接这个特点实现的。例如，如果编写一个面向接口的应用程序，可以等到运行时在制定实际的实现类；用户可以通过Java与定义的和自定义的类加载器，让一个本地的应用程序可以在运行时从网络或其他地方加载一个二进制流作为代码的一部分，这种组装应用程序的方式目前已经广泛应用于Java程序之中。从最基础的Applet，JSP到复杂的OSGi技术，都使用了Java语言运行期类加载的特性。

Java的类加载是一个相对复杂的过程；它包括加载、验证、准备、解析和初始化五个阶段；对于开发者来说，可控性最强的是**加载阶段**；加载阶段主要完成三件事：

- 根据一个类的全限定名来获取定义此类的二进制字节流
- 将这个字节流所代表的静态存储结构转化为JVM方法区中的运行时数据结构
- 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。『通过一个类的全限定名获取描述此类的二进制字节流』这个过程被抽象出来，就是Java的类加载器模块，也即JDK中ClassLoader API。

Android Framework提供了DexClassLoader这个类，简化了『通过一个类的全限定名获取描述次类的二进制字节流』这个过程；我们只需要告诉DexClassLoader一个dex文件或者apk文件的路径就能完成类的加载。因此本文的内容用一句话就可以概括：

**将插件的dex或者apk文件告诉『合适的』 DexClassLoader，借助它完成插件类的加载**

关于ClassLoader机制更多的内容，请参阅『深入理解Java虚拟机』这本书。

## 思路分析

Android系统使用了ClassLoader机制来进行Activity等组件的加载；apk被安装之后，APK文件的代码以及资源会被系统存放在固定的目录（比如/data/app/package\_name/base-1.apk）系统在进行类加载的时候，会自动去这一个或者几个特定的路径来寻找这个类；但是系统并不知道存在于插件中的Activity组件的信息（插件可以是任意位置，甚至是网络，系统无法提前预知），因此正常情况下系统无法加载我们插件中的类；因此也没有办法创建Activity的对象，更不用谈启动组件了。

解决这个问题有两个思路，要么全盘接管这个类加载的过程；要么告知系统我们使用的插件存在于哪里，让系统帮忙加载；这两种方式或多或少都需要干预这个类加载的过程。老规矩，知己知彼，百战不殆。我们首先分析一下，系统是如果完成这个类加载过程的。

我们再次搬出Activity的创建过程的代码：

```
java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
activity = mInstrumentation.newActivity(cl, component.getClassName(), r.intent);
StrictMode.incrementExpectedActivityCount(activity.getClass());
r.intent.setExtrasClassLoader(cl);
```

这里可以很明显地看到，系统通过待启动的Activity的类名className，然后使用ClassLoader对象cl把这个类加载进虚拟机，最后使用反射创建了这个Activity类的实例对象。要想干预这个ClassLoader（告知它我们的路径或者替换他），我们首先得看看这玩意到底是个什么来头。（从哪里创建的）

cl这个ClassssLoader对象通过r.packageInfo对象的getClassLoader()方法得到，r.packageInfo是一个LoadedApk类的对象；那么，LoadedApk到底是个什么东西？？

我们查阅LoadedApk类的文档，只有一句话，不过说的很明白：

Local state maintained about a currently loaded .apk.

**LoadedApk**对象是APK文件在内存中的表示。Apk文件的相关信息，诸如Apk文件的代码和资源，甚至代码里面的Activity，Service等组件的信息我们都可以通过此对象获取。

OK，我们知道这个LoadedApk是何方神圣了；接下来我们要搞清楚的是：这个r.packageInfo到底是从哪里获取的？

我们顺着performLaunchActivity上溯，辗转handleLaunchActivity回到了**H**类的**LAUNCH\_ACTIVITY**消息，找到了r.packageInfo的来源：

```
final ActivityClientRecord r = (ActivityClientRecord) msg.obj;
r.packageInfo = getPackageInfoNoCheck(
 r.activityInfo.applicationInfo, r.compatInfo);
handleLaunchActivity(r, null);
```

getPackageInfoNoCheck方法很简单，直接调用了getPackageInfo方法：

```
public final LoadedApk getPackageInfoNoCheck(ApplicationInfo ai,
 CompatibilityInfo compatInfo) {
 return getPackageInfo(ai, compatInfo, null, false, true, false);
}
```

在这个 `getPackageInfo` 方法里面我们发现了端倪：

```

private LoadedApk getPackageInfo(ApplicationInfo aInfo, CompatibilityInfo compatInfo,
 ClassLoader baseLoader, boolean securityViolation, boolean includeCode,
 boolean registerPackage) {
 // 获取userid信息
 final boolean differentUser = (UserHandle.myUserId() != UserHandle.getUserId(aInfo));
 synchronized (mResourcesManager) {
 // 尝试获取缓存信息
 WeakReference<LoadedApk> ref;
 if (differentUser) {
 // Caching not supported across users
 ref = null;
 } else if (includeCode) {
 ref = mPackages.get(aInfo.packageName);
 } else {
 ref = mResourcePackages.get(aInfo.packageName);
 }

 LoadedApk packageInfo = ref != null ? ref.get() : null;
 if (packageInfo == null || (packageInfo.mResources != null
 && !packageInfo.mResources.getAssets().isUpToDate())) {
 // 缓存没有命中，直接new
 packageInfo =
 new LoadedApk(this, aInfo, compatInfo, baseLoader,
 securityViolation, includeCode &&
 (aInfo.flags&ApplicationInfo.FLAG_HAS_CODE) != 0, registerPac
 // 省略。。更新缓存
 return packageInfo;
 }
 }
}

```

这个方法很重要，我们必须弄清楚每一步；

首先，它判断了调用方和或许App信息的一方是不是同一个userId；如果是同一个user，那么可以共享缓存数据（要么缓存的代码数据，要么缓存的资源数据）

接下来尝试获取缓存数据；如果没有命中缓存数据，才通过`LoadedApk`的构造函数创建了`LoadedApk`对象；创建成功之后，如果是同一个uid还放入了缓存。

提到缓存数据，看过[Hook机制之Binder Hook](#)的童鞋可能就知道了，我们之前成功借助ServiceManager的本地代理使用缓存的机制Hook了各种Binder；因此这里完全可以如法炮制——我们拿到这一份缓存数据，修改里面的ClassLoader；自己控制类加载的过程，这样加载插件中的Activity类的问题就解决了。这就引出了我们加载插件类的第一种方案：

# 激进方案：Hook掉ClassLoader，自己操刀

从上述分析中我们得知，在获取LoadedApk的过程中使用了一份缓存数据；这个缓存数据是一个Map，从包名到LoadedApk的一个映射。正常情况下，我们的插件肯定不会存在于这个对象里面；但是如果我们手动把我们插件的信息添加到里面呢？系统在查找缓存的过程中，会直接命中缓存！进而使用我们添加进去的LoadedApk的ClassLoader来加载这个特定的Activity类！这样我们就能接管我们自己插件类的加载过程了！

这个缓存对象 `mPackages` 存在于 `ActivityThread` 类中；老方法，我们首先获取这个对象：

```
// 先获取到当前的ActivityThread对象
Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
Method currentActivityThreadMethod = activityThreadClass.getDeclaredMethod("currentActivityThread");
currentActivityThreadMethod.setAccessible(true);
Object currentActivityThread = currentActivityThreadMethod.invoke(null);

// 获取到 mPackages 这个静态成员变量，这里缓存了dex包的信息
Field mPackagesField = activityThreadClass.getDeclaredField("mPackages");
mPackagesField.setAccessible(true);
Map mPackages = (Map) mPackagesField.get(currentActivityThread);
```

拿到这个Map之后接下来怎么办呢？**我们需要填充这个map，把插件的信息塞进这个map里面**，以便系统在查找的时候能命中缓存。但是这个填充这个Map我们出了需要包名之外，还需要一个LoadedApk对象；如何创建一个LoadedApk对象呢？

我们当然可以直接反射调用它的构造函数直接创建出需要的对象，但是万一哪里有疏漏，构造参数填错了怎么办？又或者Android的不同版本使用了不同的参数，导致我们创建出来的对象与系统创建出的对象不一致，无法work怎么办？

因此我们需要使用与系统完全相同的方式创建LoadedApk对象；从上文分析得知，系统创建LoadedApk对象是通过getPackageInfo来完成的，因此我们可以调用这个函数来创建LoadedApk对象；但是这个函数是private的，我们无法使用。

有的童鞋可能会有疑问了，private不是也能反射到吗？我们确实能够调用这个函数，但是private表明这个函数是内部实现，或许那一天Google高兴，把这个函数改个名字我们就直接GG了；但是public函数不同，public被导出的函数你无法保证是否有别人调用它，因此大部分情况下不会修改；我们最好调用public函数来保证尽可能少的遇到兼容性问题。

（当然，如果实在木有路可以考虑调用私有方法，自己处理兼容性问题，这个我们以后也会遇到）

间接调用getPackageInfo这个私有函数的public函数有同名的getPackageInfo系列和getPackageInfoNoCheck；简单查看源代码发现，getPackageInfo除了获取包的信息，还检查了包的一些组件；为了绕过这些验证，我们选择使用getPackageInfoNoCheck获取LoadedApk信息。

## 构建插件LoadedApk对象

我们这一步的目的很明确，通过getPackageInfoNoCheck函数创建出我们需要的LoadedApk对象，以供接下来使用。

这个函数的签名如下：

```
public final LoadedApk getPackageInfoNoCheck(ApplicationInfo ai,
 CompatibilityInfo compatInfo) {
```

因此，为了调用这个函数，我们需要构造两个参数。其一是ApplicationInfo，其二是CompatibilityInfo；第二个参数顾名思义，代表这个App的兼容性信息，比如targetSDK版本等等，这里我们只需要提取出app的信息，因此直接使用默认的兼容性即可；在CompatibilityInfo类里面有一个公有字段DEFAULT\_COMPATIBILITY\_INFO代表默认兼容性信息；因此，我们的首要目标是获取这个ApplicationInfo信息。

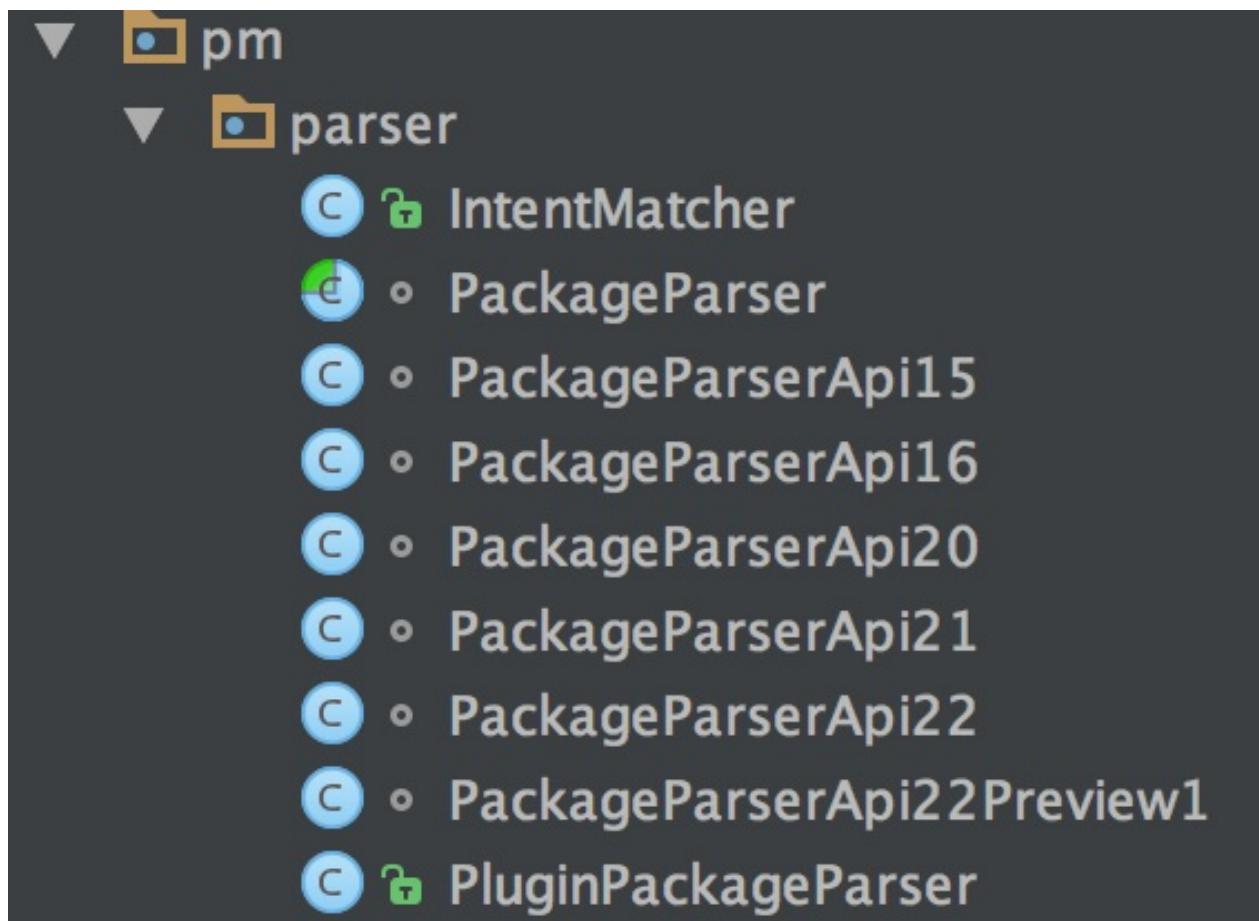
### 构建插件ApplicationInfo信息

我们首先看看ApplicationInfo代表什么，这个类的文档说得很清楚：

Information you can retrieve about a particular application. This corresponds to information collected from the AndroidManifest.xml's tag.

也就是说，这个类就是AndroidManifest.xml里面的这个标签下面的信息；这个AndroidManifest.xml无疑是一个标准的xml文件，因此我们完全可以自己使用parse来解析这个信息。

那么，系统是如何获取这个信息的呢？其实Framework就有一个这样的parser，也即PackageParser；理论上，我们也可以借用系统的parser来解析AndroidManifest.xml从而得到ApplicationInfo的信息。但遗憾的是，这个类的兼容性很差；Google几乎在每一个Android版本都对这个类动刀子，如果坚持使用系统的解析方式，必须写一系列兼容行代码！！DroidPlugin就选择了这种方式，相关类如下：



看到这里我就问你怕不怕！！！这也是我们之前提到的私有或者隐藏的API可以使用，但必须处理好兼容性问题；如果Android 7.0发布，这里估计得添加一个新的类 PackageParseApi24。

我这里使用API 23作为演示，版本不同的可能无法运行请自行查阅 DroidPlugin 不同版本如何处理。

OK回到正题，我们决定使用PackageParser类来提取ApplicationInfo信息。下图是API 23上，PackageParser的部分类结构图：

```

m Ȑ PackageParser(String)
m Ȑ collectCertificates(Package, int): boolean
m Ȑ collectManifestDigest(Package): boolean
d m Ȑ generateActivityInfo(Activity, int, PackageUserState, int): ActivityInfo
d m Ȑ generateApplicationInfo(Package, int, PackageUserState): ApplicationInfo
d m Ȑ generateApplicationInfo(Package, int, PackageUserState, int): ApplicationInfo
d m Ȑ generateInstrumentationInfo(Instrumentation, int): InstrumentationInfo
d m Ȑ generatePackageInfo(Package, int[], int, long, long, HashSet<String>, Package)
d m Ȑ generatePackageInfo(Package, int[], int, long, long, HashSet<String>, Package)
d m Ȑ generatePermissionGroupInfo(PermissionGroup, int): PermissionGroupInfo
d m Ȑ generatePermissionInfo(Permission, int): PermissionInfo
d m Ȑ generateProviderInfo(Provider, int, PackageUserState, int): ProviderInfo

```

看起来有我们需要的方法 `generateApplication`；确实如此，依靠这个方法我们可以成功地拿到 `ApplicationInfo`。

由于 `PackageParser` 是 `@hide` 的，因此我们需要通过反射进行调用。我们根据这个 `generateApplicationInfo` 方法的签名：

```

public static ApplicationInfo generateApplicationInfo(Package p, int flags,
 PackageUserState state)

```

可以写出调用 `generateApplicationInfo` 的反射代码：

```

Class<?> packageParserClass = Class.forName("android.content.pm.PackageParser");
// 首先拿到我们得终极目标：generateApplicationInfo方法
// API 23 !!!
// public static ApplicationInfo generateApplicationInfo(Package p, int flags,
// PackageUserState state) {
// 其他Android版本不保证也是如此。
Class<?> packageParser$PackageClass = Class.forName("android.content.pm.PackageParser$Package");
Class<?> packageUserStateClass = Class.forName("android.content.pm.PackageUserState");
Method generateApplicationInfoMethod = packageParserClass.getDeclaredMethod("generateApplicationInfo",
 packageParser$PackageClass,
 int.class,
 packageUserStateClass);

```

要成功调用这个方法，还需要三个参数；因此接下来我们需要一步一步构建调用此函数的参数信息。

## 构建 `PackageParser.Package`

`generateApplicationInfo` 方法需要的第一个参数是 `PackageParser.Package`；从名字上看这个类代表某个apk包的信息，我们看看文档怎么解释：

Representation of a full package parsed from APK files on disk. A package consists of a single base APK, and zero or more split APKs.

果然，这个类代表从 `PackageParser` 中解析得到的某个 apk 包的信息，是磁盘上 apk 文件在内存中的数据结构表示；因此，要获取这个类，肯定需要解析整个 apk 文件。

`PackageParser` 中解析 apk 的核心方法是 `parsePackage`，这个方法返回的就是一个 `Package` 类型的实例，因此我们调用这个方法即可；使用反射代码如下：

```
// 首先，我们得创建出一个Package对象出来供这个方法调用
// 而这个需要得对象可以通过 android.content.pm.PackageParser#parsePackage 这个方法返回得 Pac
// 创建出一个PackageParser对象供使用
Object packageParser = packageParserClass.newInstance();
// 调用 PackageParser.parsePackage 解析apk的信息
Method parsePackageMethod = packageParserClass.getDeclaredMethod("parsePackage", File

// 实际上是一个 android.content.pm.PackageParser.Package 对象
Object packageObj = parsePackageMethod.invoke(packageParser, apkFile, 0);
```

这样，我们就得到了 `generateApplicationInfo` 的第一个参数；第二个参数是解析包使用的 flag，我们直接选择解析全部信息，也就是 0；

## 构建 `PackageUserState`

第三个参数是 `PackageUserState`，代表不同用户中包的信息。由于 Android 是一个多任务多用户系统，因此不同的用户同一个包可能有不同的状态；这里我们只需要获取包的信息，因此直接使用默认的即可；

至此，`generateApplicaionInfo` 的参数我们已经全部构造完成，直接调用此方法即可得到我们需要的 `applicationInfo` 对象；在返回之前我们需要做一点小小的修改：使用系统系统的这个方法解析得到的 `ApplicationInfo` 对象中并没有 apk 文件本身的信息，所以我们把解析的 apk 文件的路径设置一下（`ClassLoader` 依赖 dex 文件以及 apk 的路径）：

```
// 第三个参数 mDefaultPackageUserState 我们直接使用默认构造函数构造一个出来即可
Object defaultPackageUserState = packageUserStateClass.newInstance();

// 万事具备!!!!!!!!!!!!!!
ApplicationInfo applicationInfo = (ApplicationInfo) generateApplicationInfoMethod.invoke(
 packageObj, 0, defaultPackageUserState);
String apkPath = apkFile.getPath();
applicationInfo.sourceDir = apkPath;
applicationInfo.publicSourceDir = apkPath;
```

## 替换ClassLoader

### 获取LoadedApk信息

方才为了获取ApplicationInfo我们费了好大一番精力；回顾一下我们的初衷：

我们最终的目的是调用getPackageInfoNoCheck得到LoadedApk的信息，并替换其中的mClassLoader然后把添加到ActivityThread的mPackages缓存中；从而达到我们使用自己的ClassLoader加载插件中的类的目的。

现在我们已经拿到了getPackageInfoNoCheck这个方法中至关重要的第一个参数applicationInfo；上文提到第二个参数CompatibilityInfo代表设备兼容性信息，直接使用默认的值即可；因此，两个参数都已经构造出来，我们可以调用getPackageInfoNoCheck获取LoadedApk：

```
// android.content.res.CompatibilityInfo
Class<?> compatibilityInfoClass = Class.forName("android.content.res.CompatibilityInfo");
Method getPackageInfoNoCheckMethod = activityThreadClass.getDeclaredMethod("getPackageInfoNoCheck",
 new Class[]{ActivityThread.class, ApplicationInfo.class});

Field defaultCompatibilityInfoField = compatibilityInfoClass.getDeclaredField("DEFAULT_COMPATIBILITY_INFO");
defaultCompatibilityInfoField.setAccessible(true);

Object defaultCompatibilityInfo = defaultCompatibilityInfoField.get(null);
ApplicationInfo applicationInfo = generateApplicationInfo(apkFile);

Object loadedApk = getPackageInfoNoCheckMethod.invoke(currentActivityThread, applicationInfo,
```

我们成功地构造出了LoadedAPK，接下来我们需要替换其中的ClassLoader，然后把它添加进ActivityThread的mPackages中：

```

String odexPath = Utils.getPluginOptDexDir(applicationInfo.packageName).getPath();
String libDir = Utils.getPluginLibDir(applicationInfo.packageName).getPath();
ClassLoader classLoader = new CustomClassLoader(apkFile.getPath(), odexPath, libDir,
Field mClassLoaderField = loadedApk.getClass().getDeclaredField("mClassLoader");
mClassLoaderField.setAccessible(true);
mClassLoaderField.set(loadedApk, classLoader);

// 由于是弱引用，因此我们必须在某个地方存一份，不然容易被GC；那么就前功尽弃了。
sLoadedApk.put(applicationInfo.packageName, loadedApk);

WeakReference weakReference = new WeakReference(loadedApk);
mPackages.put(applicationInfo.packageName, weakReference);

```

我们的这个CustomClassLoader非常简单，直接继承了DexClassLoader，什么都没有做；当然这里可以直接使用DexClassLoader，这里重新创建一个类是为了更有区分度；以后也可以通过修改这个类实现对于类加载的控制：

```

public class CustomClassLoader extends DexClassLoader {

 public CustomClassLoader(String dexPath, String optimizedDirectory, String libraryPath, ClassLoader parent) {
 super(dexPath, optimizedDirectory, libraryPath, parent);
 }
}

```

到这里，我们已经成功地把插件的信息放入ActivityThread中，这样我们插件中的类能够成功地被加载；因此插件中的Activity实例能被成功第创建；由于整个流程较为复杂，我们简单梳理一下：

在ActivityThread接收到IApplication的scheduleLaunchActivity远程调用之后，将消息转发给H类在handleMessage的时候，调用了getPackageInfoNoCheck方法来获取待启动的组件信息。在这个方法中会优先查找mPackages中的缓存信息，而我们已经手动把插件信息添加进去；因此能够成功命中缓存，获取到独立存在的插件信息。H类然后调用handleLaunchActivity最终转发到performLaunchActivity方法；这个方法使用从getPackageInfoNoCheck中拿到LoadedApk中的mClassLoader来加载Activity类，进而使用反射创建Activity实例；接着创建Application、Context等完成Activity组件的启动。看起来好像已经天衣无缝万事大吉了；但是运行一下会出现一个异常，如下：

```
04-05 02:49:53.742 11759-11759/com.weishu.upf.hook_classloader
E/AndroidRuntime : FATAL EXCEPTION: main Process:
com.weishu.upf.hook_classloader, PID: 11759 java.lang.RuntimeException:
Unable to start activity
ComponentInfo{com.weishu.upf.ams_pms_hook.app/com.weishu.upf.ams_pms_h
ook.app.MainActivity}: java.lang.RuntimeException: Unable to instantiate
application android.app.Application: java.lang.IllegalStateException: Unable to get
package info for com.weishu.upf.ams_pms_hook.app; is package not installed?
```

错误提示说是无法实例化 Application，而Application的创建也是在performLaunchActivity 中进行的，这里有些蹊跷，我们仔细查看一下。

## 绕过系统检查

通过ActivityThread的performLaunchActivity方法可以得知，Application通过LoadedApk的makeApplication方法创建，我们查看这个方法，在源码中发现了上文异常抛出的位置：

```
try {
 java.lang.ClassLoader cl = getClassLoader();
 if (!mPackageName.equals("android")) {
 initializeJavaContextClassLoader();
 }
 ContextImpl appContext = ContextImpl.createAppContext(mActivityThread, this);
 app = mActivityThread.mInstrumentation.newApplication(
 cl, appClass, appContext);
 appContext.setOuterContext(app);
} catch (Exception e) {
 if (!mActivityThread.mInstrumentation.onException(app, e)) {
 throw new RuntimeException(
 "Unable to instantiate application " + appClass
 + ": " + e.toString(), e);
 }
}
```

木有办法，我们只有一行一行地查看到底是谁抛出这个异常的了；所幸代码不多。（所以说，缩小异常范围是一件多么重要的事情！！！）

第一句 getClassLoader() 没什么可疑的，虽然方法很长，但是它木有抛出任何异常（当然，它调用的代码可能抛出异常，万一找不到只能进一步深搜了；所以我觉得这里应该使用受检异常）。

然后我们看第二句，如果包名不是android开头，那么调用了一个叫做 initializeJavaContextClassLoader 的方法；我们查阅这个方法：

```
private void initializeJavaContextClassLoader() {
 I PackageManager pm = ActivityThread.getPackageManager();
 android.content.pm.PackageInfo pi;
 try {
 pi = pm.getPackageInfo(mPackageName, 0, UserHandle.myUserId());
 } catch (RemoteException e) {
 throw new IllegalStateException("Unable to get package info for "
 + mPackageName + "; is system dying?", e);
 }
 if (pi == null) {
 throw new IllegalStateException("Unable to get package info for "
 + mPackageName + "; is package not installed?");
 }

 boolean sharedUserIdSet = (pi.sharedUserId != null);
 boolean processNameNotDefault =
 (pi.applicationInfo != null &&
 !mPackageName.equals(pi.applicationInfo.processName));
 boolean sharable = (sharedUserIdSet || processNameNotDefault);
 ClassLoader contextClassLoader =
 (sharable)
 ? new WarningContextClassLoader()
 : mClassLoader;
 Thread.currentThread().setContextClassLoader(contextClassLoader);
}
```

这里，我们找出了这个异常的来源：原来这里调用了getPackageInfo方法获取包的信息；而我们的插件并没有安装在系统上，因此系统肯定认为插件没有安装，这个方法肯定返回null。所以，我们还要欺骗一下PMS，让系统觉得插件已经安装在系统上了；至于如何欺骗 PMS，[Hook机制之AMS&PMS](#) 有详细解释，这里直接给出代码，不赘述了：

```

private static void hookPackageManager() throws Exception {

 // 这一步是因为 initializeJavaContextClassLoader 这个方法内部无意中检查了这个包是否在系统安装目录
 // 如果没有安装，直接抛出异常，这里需要临时Hook掉 PMS，绕过这个检查。

 Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
 Method currentActivityThreadMethod = activityThreadClass.getDeclaredMethod("currentActivityThread");
 currentActivityThreadMethod.setAccessible(true);
 Object currentActivityThread = currentActivityThreadMethod.invoke(null);

 // 获取ActivityThread里面原始的 sPackageManager
 Field sPackageManagerField = activityThreadClass.getDeclaredField("sPackageManager");
 sPackageManagerField.setAccessible(true);
 Object sPackageManager = sPackageManagerField.get(currentActivityThread);

 // 准备好代理对象，用来替换原始的对象
 Class<?> iPackageManagerInterface = Class.forName("android.content.pm.IPackageManager");
 Object proxy = Proxy.newProxyInstance(iPackageManagerInterface.getClassLoader(),
 new Class<?>[] { iPackageManagerInterface },
 new IPackageManagerHookHandler(sPackageManager));

 // 1. 替换掉ActivityThread里面的 sPackageManager 字段
 sPackageManagerField.set(currentActivityThread, proxy);
}

```

OK到这里，我们已经能够成功地加载简单的独立的存在于外部文件系统中的apk了。至此关于 DroidPlugin 对于Activity生命周期的管理已经完全讲解完毕了；这是一种极其复杂的Activity管理方案，我们仅仅写一个用来理解的demo就Hook了相当多的东西，在Framework层来回牵扯；这其中的来龙去脉要完全把握清楚还请读者亲自翻阅源码。另外，我在此对DroidPlugin 作者献上我的膝盖～这其中的玄妙让人叹为观止！

上文给出的方案中，我们全盘接管了插件中类的加载过程，这是一种相对暴力的解决方案；能不能更温柔一点呢？通俗来说，我们可以选择改革，而不是革命——告诉系统ClassLoader一些必要信息，让它帮忙完成插件类的加载。

## 保守方案：委托系统，让系统帮忙加载

我们再次搬出ActivityThread中加载Activity类的代码：

```

java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
activity = mInstrumentation.newActivity(
 cl, component.getClassName(), r.intent);
StrictMode.incrementExpectedActivityCount(activity.getClass());
r.intent.setExtrasClassLoader(cl);

```

我们知道这个r.packageInfo中的r是通过getPackageInfoNoCheck获取到的；在『激进方案』中我们把插件apk手动添加进缓存，采用自己加载办法解决；如果我们不干预这个过程，导致无法命中mPackages中的缓存，会发生什么？

查阅getPackageInfo方法如下：

```

private LoadedApk getPackageInfo(ApplicationInfo aInfo, CompatibilityInfo compatInfo,
 ClassLoader baseLoader, boolean securityViolation, boolean includeCode,
 boolean registerPackage) {
 final boolean differentUser = (UserHandle.myUserId() != UserHandle.getUserId(aInfo));
 synchronized (mResourcesManager) {
 WeakReference<LoadedApk> ref;
 if (differentUser) {
 // Caching not supported across users
 ref = null;
 } else if (includeCode) {
 ref = mPackages.get(aInfo.packageName);
 } else {
 ref = mResourcePackages.get(aInfo.packageName);
 }

 LoadedApk packageInfo = ref != null ? ref.get() : null;
 if (packageInfo == null || (packageInfo.mResources != null
 && !packageInfo.mResources.getAssets().isUpToDate())) {
 packageInfo =
 new LoadedApk(this, aInfo, compatInfo, baseLoader,
 securityViolation, includeCode &&
 (aInfo.flags&ApplicationInfo.FLAG_HAS_CODE) != 0, registerPac
 // 略
 }
 }
}

```

可以看到，没有命中缓存的情况下，系统直接new了一个LoadedApk；注意这个构造函数的第二个参数aInfo，这是一个ApplicationInfo类型的对象。在『激进方案』中我们为了获取独立插件的ApplicationInfo花了不少心思；那么如果不做任何处理这里传入的这个aInfo参数是什么？

追本溯源不难发现，这个aInfo是从我们的替身StubActivity中获取的！而StubActivity存在于宿主程序中，所以，这个aInfo对象代表的实际上就是宿主程序的Application信息！

我们知道，接下来会使用new出来的这个LoadedApk的getClassLoader()方法获取到ClassLoader来对插件的类进行加载；而获取到的这个ClassLoader是宿主程序使用的ClassLoader，因此现在还无法加载插件的类；那么，**我们能不能让宿主的ClassLoader获得加载插件类的能力呢？**；如果我们告诉宿主使用的ClassLoader插件使用的类在哪里，就能帮助他完成加载！

宿主的ClassLoader在哪里，是唯一的吗？

上面说到，我们可以通过告诉宿主程序的ClassLoader插件使用的类，让宿主的ClassLoader完成对于插件类的加载；那么问题来了，我们如何获取到宿主的ClassLoader？宿主程序使用的ClassLoader默认情况下是全局唯一的吗？

答案是肯定的。

因为在FrameWork中宿主程序也是使用LoadedApk表示的，如同Activity启动是加载Activity类一样，宿主中的类也都是通过LoadedApk的getClassLoader()方法得到的ClassLoader加载的；由类加载机制的『双亲委派』特性，只要有一个应用程序类由某一个ClassLoader加载，那么它引用到的别的类除非父加载器能加载，否则都是由这同一个加载器加载的（不遵循双亲委派模型的除外）。

表示宿主的LoadedApk在Application类中有一个成员变量mLoadedApk，而这个变量是从ContextImpl中获取的；ContextImpl重写了getClassLoader方法，因此我们在**Context**环境中直接getClassLoader()获取到的就是宿主程序唯一的ClassLoader。

## LoadedApk的ClassLoader到底是什么？

现在我们确保了『使用宿主ClassLoader帮助加载插件类』可行性；那么我们应该如何完成这个过程呢？

知己知彼，百战不殆。

不论是宿主程序还是插件程序都是通过LoadedApk的getClassLoader()方法返回的ClassLoader进行类加载的，返回的这个ClassLoader到底是个什么东西？？这个方法源码如下：

```
public ClassLoader getClassLoader() {
 synchronized (this) {
 if (mClassLoader != null) {
 return mClassLoader;
 }

 if (mIncludeCode && !mPackageName.equals("android")) {
 // 略...
 mClassLoader = ApplicationLoaders.getDefault().getClassLoader(zip, lib,
 mBaseClassLoader);

 StrictMode.setThreadPolicy(oldPolicy);
 } else {
 if (mBaseClassLoader == null) {
 mClassLoader = ClassLoader.getSystemClassLoader();
 } else {
 mClassLoader = mBaseClassLoader;
 }
 }
 return mClassLoader;
 }
}
```

可以看到，非android开头的包和android开头的包分别使用了两种不同的ClassLoader，我们只关心第一种；因此继续跟踪ApplicationLoaders类：

```

public ClassLoader getClassLoader(String zip, String libPath, ClassLoader parent)
{
 ClassLoader baseParent = ClassLoader.getSystemClassLoader().getParent();

 synchronized (mLoaders) {
 if (parent == null) {
 parent = baseParent;
 }

 if (parent == baseParent) {
 ClassLoader loader = mLoaders.get(zip);
 if (loader != null) {
 return loader;
 }
 }

 Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, zip);
 PathClassLoader pathClassloader =
 new PathClassLoader(zip, libPath, parent);
 Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);

 mLoaders.put(zip, pathClassloader);
 return pathClassloader;
 }

 Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, zip);
 PathClassLoader pathClassloader = new PathClassLoader(zip, parent);
 Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
 return pathClassloader;
}
}

```

可以看到，应用程序使用的ClassLoader都是PathClassLoader类的实例。那么，这个PathClassLoader是什么呢？从Android SDK给出的源码只能看出这么多：

```

public class PathClassLoader extends BaseDexClassLoader {
 public PathClassLoader(String dexPath, ClassLoader parent) {
 super((String)null, (File)null, (String)null, (ClassLoader)null);
 throw new RuntimeException("Stub!");
 }

 public PathClassLoader(String dexPath, String libraryPath, ClassLoader parent) {
 super((String)null, (File)null, (String)null, (ClassLoader)null);
 throw new RuntimeException("Stub!");
 }
}

```

SDK没有导出这个类的源码，我们去[androidxref](#)上面看；发现其实这个类真的就这么多内容；我们继续查看它的父类[BaseDexClassLoader](#)；ClassLoader嘛，我们查看findClass或者defineClass方法，BaseDexClassLoader的findClass方法如下：

```
protected Class<?> findClass(String name) throws ClassNotFoundException {
 List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
 Class c = pathList.findClass(name, suppressedExceptions);
 if (c == null) {
 ClassNotFoundException cnfe = new ClassNotFoundException("Didn't find class \
for (Throwable t : suppressedExceptions) {
 cnfe.addSuppressed(t);
 }
 throw cnfe;
}
return c;
}
```

可以看到，查找Class的任务通过pathList完成；这个pathList是一个DexPathList类的对象，它的findClass方法如下：

```
public Class findClass(String name, List<Throwable> suppressed) {
 for (Element element : dexElements) {
 DexFile dex = element.dexFile;

 if (dex != null) {
 Class clazz = dex.loadClassBinaryName(name, definingContext, suppressed);
 if (clazz != null) {
 return clazz;
 }
 }
 }
 if (dexElementsSuppressedExceptions != null) {
 suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));
 }
 return null;
}
```

这个DexPathList内部有一个叫做dexElements的数组，然后findClass的时候会遍历这个数组来查找Class；如果我们把插件的信息塞进这个数组里面，那么不就能够完成类的加载过程吗？！！

## 给默认ClassLoader打补丁

通过上述分析，我们知道，可以把插件的相关信息放入BaseDexClassLoader的表示dex文件的数组里面，这样宿主程序的ClassLoader在进行类加载，遍历这个数组的时候，会自动遍历到我们添加进去的插件信息，从而完成插件类的加载！

接下来，我们实现这个过程；我们会用到一些较为复杂的反射技术哦～不过代码非常短：

```

public static void patchClassLoader(ClassLoader cl, File apkFile, File optDexFile)
 throws IllegalAccessException, NoSuchMethodException, IOException, InvocationTargetException {
 // 获取 BaseDexClassLoader : pathList
 Field pathListField = DexClassLoader.class.getSuperclass().getDeclaredField("pathList");
 pathListField.setAccessible(true);
 Object pathListObj = pathListField.get(cl);

 // 获取 PathList: Element[] dexElements
 Field dexElementArray = pathListObj.getClass().getDeclaredField("dexElements");
 dexElementArray.setAccessible(true);
 Object[] dexElements = (Object[]) dexElementArray.get(pathListObj);

 // Element 类型
 Class<?> elementClass = dexElements.getClass().getComponentType();

 // 创建一个数组，用来替换原始的数组
 Object[] newElements = (Object[]) Array.newInstance(elementClass, dexElements.length);

 // 构造插件Element(File file, boolean isDirectory, File zip, DexFile dexFile) 这个构造器
 Constructor<?> constructor = elementClass.getConstructor(File.class, boolean.class, File.class, DexFile.class);
 Object o = constructor.newInstance(apkFile, false, apkFile, DexFile.loadDex(apkFile));

 Object[] toAddElementArray = new Object[] { o };
 // 把原始的elements复制进去
 System.arraycopy(dexElements, 0, newElements, 0, dexElements.length);
 // 插件的那个element复制进去
 System.arraycopy(toAddElementArray, 0, newElements, dexElements.length, toAddElementArray.length);

 // 替换
 dexElementArray.set(pathListObj, newElements);

}

```

短短的二十几行代码，我们就完成了『委托宿主ClassLoader加载插件类』的任务；因此第二种方案也宣告完成！我们简要总结一下这种方式的原理：

- 默认情况下performLaunchActivity会使用替身StubActivity的ApplicationInfo也就是宿主程序的ClassLoader加载所有的类；我们的思路是告诉宿主ClassLoader我们在哪，让其帮助完成类加载的过程。
- 宿主程序的ClassLoader最终继承自BaseDexClassLoader，BaseDexClassLoader通

过DexPathList进行类的查找过程；而这个查找通过遍历一个dexElements的数组完成；我们通过把插件dex添加进这个数组就让宿主ClasLoader获取了加载插件类的能力。

## 小结

本文中我们采用两种方案成功完成了『启动没有在AndroidManifest.xml中显示声明，并且存在于外部插件中的Activity』的任务。

『激进方案』中我们自定义了插件的ClassLoader，并且绕开了Framework的检测；利用ActivityThread对于LoadedApk的缓存机制，我们把携带这个自定义的ClassLoader的插件信息添加进mPackages中，进而完成了类的加载过程。

『保守方案』中我们深入探究了系统使用ClassLoader findClass的过程，发现应用程序使用的非系统类都是通过同一个PathClassLoader加载的；而这个类的最终父类BaseDexClassLoader通过DexPathList完成类的查找过程；我们hack了这个查找过程，从而完成了插件类的加载。

这两种方案孰优孰劣呢？

很显然，『激进方案』比较麻烦，从代码量和分析过程就可以看出来，这种机制异常复杂；而且在解析apk的时候我们使用的PackageParser的兼容性非常差，我们不得不手动处理每一个版本的apk解析api；另外，它Hook的地方也有多：不仅需要Hook AMS和H，还需要Hook ActivityThread的mPackages和PackageManager！

『保守方案』则简单得多（虽然原理也不简单），不仅代码很少，而且Hook的地方也不多；有一点正本清源的意思，从最最上层Hook住了整个类的加载过程。

但是，我们不能简单地说『保守方案』比『激进方案』好。从根本上说，这两种方案的差异在哪呢？

『激进方案』是多ClassLoader构架，每一个插件都有一个自己的ClassLoader，因此类的隔离性非常好——如果不同的插件使用了同一个库的不同版本，它们相安无事！『保守方案』是单ClassLoader方案，插件和宿主程序的类全部都通过宿主的ClasLoader加载，虽然代码简单，但是鲁棒性很差；一旦插件之间甚至插件与宿主之间使用的类库有冲突，那么直接GG。

多ClassLoader还有一个优点：可以真正完成代码的热加载！如果插件需要升级，直接重新创建一个自定的ClassLoader加载新的插件，然后替换掉原来的版本即可（Java中，不同ClassLoader加载的同一个类被认为是不同的类）；单ClassLoader的话实现非常麻烦，有可能需要重启进程。

在J2EE领域中广泛使用ClassLoader的地方均采用多ClassLoader架构，比如Tomcat服务器，Java模块化事实标准的OSGi技术；所以，我们有足够的理由认为**选择多ClassLoader架构在大多数情况下是明智之举。**

目前开源的插件方案中，DroidPlugin采用的『激进方案』，Small采用的『保守方案』那么，有没有两种优点兼顾的方案呢？？

答案自然是有的。

DroidPlugin和Small的共同点是**两者都是非侵入式的插件框架**；什么是『非侵入式』呢？打个比方，你启动一个插件Activity，直接使用startActivity即可，就跟开发普通的apk一样，开发插件和普通的程序对于开发者来说没有什么区别。

如果我们一定程度上放弃这种『侵入性』，那么我们就能实现一个两者优点兼而有之的插件框架！这里我先卖个关子～

OK，本文的内容就到这里了；关于『插件机制对于Activity的处理方式』也就此完结。要说明的是，在本文的『保守方案』其实只处理了代码的加载过程，它并不能加载有资源的apk！所以目前我这个实现基本没什么用；当然我这里只是就『代码加载』进行举例；至于资源，那牵扯到另外一个问题——**插件系统的资源管理机制**这个在后续文章的合适机会我会单独讲解。

接下来的文章，会讲述Android四大组件的另外三个Service，BroadCastReceiver，ContentProvider的处理方式。喜欢就点个赞吧～持续更新，请关注github项目[understand-plugin-framework](#)和我的[博客](#)！这文章我前前后后准备了快两个星期，如果你看到了这里，还请支持一下：)

# Android插件化原理解析——广播的管理

来源：[Weishu's Notes](#)

在[Activity生命周期管理](#) 以及 [插件加载机制](#) 中我们详细讲述了插件化过程中对于Activity组件的处理方式，为了实现Activity的插件化我们付出了相当多的努力；那么Android系统的其他组件，比如BroadcastReceiver，Service还有ContentProvider，它们又该如何处理呢？

相比Activity，BroadcastReceiver要简单很多——广播的生命周期相当简单；如果希望插件能够支持广播，这意味着什么？

回想一下我们日常开发的时候是如何使用BroadcastReceiver的：注册，发送和接收；因此，要实现BroadcastReceiver的插件化就这三种操作提供支持；接下来我们将一步步完成这个过程。

阅读本文之前，可以先clone一份 [understand-plugin-framework](#)，参考此项目的receiver-management 模块。另外，插件框架原理解析系列文章见索引。

如果连BroadcastReceiver的工作原理都不清楚，又怎么能让插件支持它？老规矩，知己知彼。

## 源码分析

我们可以注册一个BroadcastReceiver然后接收我们感兴趣的广播，也可以给某有缘人发出某个广播；因此，我们对源码的分析按照两条路线展开：

### 注册过程

不论是静态广播还是动态广播，在使用之前都是需要注册的；动态广播的注册需要借助 Context类的 `registerReceiver` 方法，而静态广播的注册直接在 `AndroidManifest.xml` 中声明即可；我们首先分析一下动态广播的注册过程。

Context类的 `registerReceiver` 的真正实现在 `ContextImpl` 里面，而这个方法间接调用了 `registerReceiverInternal`，源码如下：

```

private Intent registerReceiverInternal(BroadcastReceiver receiver, int userId,
 IntentFilter filter, String broadcastPermission,
 Handler scheduler, Context context) {
 IIntentReceiver rd = null; // Important !!!!!
 if (receiver != null) {
 if (mPackageInfo != null && context != null) {
 if (scheduler == null) {
 scheduler = mMainThread.getHandler();
 }
 rd = mPackageInfo.getReceiverDispatcher(
 receiver, context, scheduler,
 mMainThread.getInstrumentation(), true);
 } else {
 if (scheduler == null) {
 scheduler = mMainThread.getHandler();
 }
 rd = new LoadedApk.ReceiverDispatcher(
 receiver, context, scheduler, null, true).getIIntentReceiver();
 }
 }
 try {
 return ActivityManagerNative.getDefault().registerReceiver(
 mMainThread.getApplicationThread(), mBasePackageName,
 rd, filter, broadcastPermission, userId);
 } catch (RemoteException e) {
 return null;
 }
}

```

可以看到，BroadcastReceiver的注册也是通过AMS完成的；在进入AMS跟踪它的registerReceiver方法之前，我们先弄清楚这个IIntentReceiver类型的变量rd是什么。首先查阅API文档，很遗憾SDK里面没有导出这个类，我们直接去[grepcode](#)上看，文档如下：

System private API for dispatching intent broadcasts. This is given to the activity manager as part of registering for an intent broadcasts, and is called when it receives intents.

这个类是通过AIDL工具生成的，它是一个Binder对象，因此可以用来跨进程传输；文档说的很清楚，它是用来进行广播分发的。什么意思呢？

由于广播的分发过程是在AMS中进行的，而AMS所在的进程和BroadcastReceiver所在的进程不一样，因此要把广播分发到BroadcastReceiver具体的进程需要进行跨进程通信，这个通信的载体就是IIntentReceiver类。其实这个类的作用跟[Activity生命周期管理](#)中提

到的 `IApplicationThread` 相同，都是 App 进程给 AMS 进程用来进行通信的对象。另外，`IIntentReceiver` 是一个接口，从上述代码中可以看出，它的实现类为 `LoadedApk.ReceiverDispatcher`。

OK，我们继续跟踪源码，AMS 类的 `registerReceiver` 方法代码有点多，这里不一一解释了，感兴趣的话可以自行查阅；这个方法主要做了以下两件事：

对发送者的身份和权限做出一定的校检 把这个 `BroadcastReceiver` 以 `BroadcastFilter` 的形式存储在 AMS 的 `mReceiverResolver` 变量中，供后续使用。就这样，被传递过来的 `BroadcastReceiver` 已经成功地注册在系统之中，能够接收特定类型的广播了；那么注册在 `AndroidManifest.xml` 中的静态广播是如何被系统感知的呢？

在 插件加载机制 中我们知道系统会通过 `PackageParser` 解析 Apk 中的 `AndroidManifest.xml` 文件，因此我们有理由认为，系统会在解析 `AndroidManifest.xml` 的标签（也即静态注册的广播）的时候保存相应的信息；而 Apk 的解析过程是在 PMS 中进行的，因此 **静态注册广播的信息存储在 PMS 中**。接下来的分析会证实这一结论。

## 发送和接收过程

### 发送过程

发送广播很简单，就是一句 `context.sendBroadcast()`，我们顺藤摸瓜，跟踪这个方法。前文也提到过，`Context` 中方法的调用都会委托到 `ContextImpl` 这个类，我们直接看 `ContextImpl` 对这个方法的实现：

```
public void sendBroadcast(Intent intent) {
 warnIfCallingFromSystemProcess();
 String resolvedType = intent.resolveTypeIfNeeded(getApplicationContext());
 try {
 intent.prepareToLeaveProcess();
 ActivityManagerNative.getDefault().broadcastIntent(
 mMainThread.getApplicationThread(), intent, resolvedType, null,
 Activity.RESULT_OK, null, null, null, AppOpsManager.OP_NONE, null, false,
 getUserId());
 } catch (RemoteException e) {
 throw new RuntimeException("Failure from system", e);
 }
}
```

嗯，发送广播也是通过AMS进行的，我们直接查看ActivityManagerService类的broadcastIntent方法，这个方法仅仅是调用了broadcastIntentLocked方法，我们继续跟踪；broadcastIntentLocked这个方法相当长，处理了诸如粘性广播，顺序广播，各种Flag以及动态广播静态广播的接收过程，这些我们暂时不关心；值得注意的是，在这个方法中我们发现，**其实广播的发送和接收是融为一体的**。某个广播被发送之后，AMS会找出所有注册过的BroadcastReceiver中与这个广播匹配的接收者，然后将这个广播分发给相应的接收者处理。

## 匹配过程

某一条广播被发出之后，并不是阿猫阿狗都能接收它并处理的；BroadcastReceiver可能只对某些类型的广播感兴趣，因此它也只能接收和处理这种特定类型的广播；在broadcastIntentLocked方法内部有如下代码：

```
// Figure out who all will receive this broadcast.
List<BroadcastFilter> receivers = null;
List<BroadcastFilter> registeredReceivers = null;
// Need to resolve the intent to interested receivers...
if ((intent.getFlags() & Intent.FLAG_RECEIVER_REGISTERED_ONLY)
 == 0) {
 receivers = collectReceiverComponents(intent, resolvedType, callingUid, users);
}
if (intent.getComponent() == null) {
 if (userId == UserHandle.USER_ALL && callingUid == Process.SHELL_UID) {
 // Query one target user at a time, excluding shell-restricted users
 // 略
 } else {
 registeredReceivers = mReceiverResolver.queryIntent(intent,
 resolvedType, false, userId);
 }
}
```

这里有两个列表receivers和registeredReceivers，看名字好像是广播接收者的列表；下面是它们的赋值过程：

```
receivers = collectReceiverComponents(intent, resolvedType, callingUid, users);
registeredReceivers = mReceiverResolver.queryIntent(intent, resolvedType, false, user
```

读者可以自行跟踪这两个方法的代码，过程比较简单，我这里直接给出结论：

- receivers是对这个广播感兴趣的静态BroadcastReceiver列表；

`collectReceiverComponents` 通过 `PackageManager` 获取了与这个广播匹配的静态 `BroadcastReceiver` 信息；这里也证实了我们在分析 `BroadcastReceiver` 注册过程中的推论——静态 `BroadcastReceiver` 的注册过程的确在 PMS 中进行的。

- `mReceiverResolver` 存储了动态注册的 `BroadcastReceiver` 的信息；还记得这个 `mReceiverResolver` 吗？我们在分析动态广播的注册过程中发现，动态注册的 `BroadcastReceiver` 的相关信息最终存储在此对象之中；在这里，通过 `mReceiverResolver` 对象匹配出了对应的 `BroadcastReceiver` 供进一步使用。

现在系统通过 PMS 拿到了所有符合要求的静态 `BroadcastReceiver`，然后从 AMS 中获取了符合要求的动态 `BroadcastReceiver`；因此接下来的工作非常简单：唤醒这些广播接受者。简单来说就是回调它们的 `onReceive` 方法。

## 接收过程

通过上文的分析过程我们知道，在 AMS 的 `broadcastIntentLocked` 方法中找出了符合要求的所有 `BroadcastReceiver`；接下来就需要把这个广播分发到这些接收者之中。在 `broadcastIntentLocked` 方法的后半部分有如下代码：

```

BroadcastQueue queue = broadcastQueueForIntent(intent);
BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,
 callerPackage, callingPid, callingUid, resolvedType,
 requiredPermissions, appOp, brOptions, receivers, resultTo, resultCode,
 resultData, resultExtras, ordered, sticky, false, userId);

boolean replaced = replacePending && queue.replaceOrderedBroadcastLocked(r);
if (!replaced) {
 queue.enqueueOrderedBroadcastLocked(r);
 queue.scheduleBroadcastsLocked();
}

```

首先创建了一个 `BroadcastRecord` 代表此次发送的这条广播，然后把它丢进一个队列，最后通过 `scheduleBroadcastsLocked` 通知队列对广播进行处理。

在 `BroadcastQueue` 中通过 `Handle` 调度了对于广播处理的消息，调度过程由 `processNextBroadcast` 方法完成，而这个方法通过 `performReceiveLocked` 最终调用了 `IIntentReceiver` 的 `performReceive` 方法。

这个 `IIntentReceiver` 正是在广播注册过程中由 App 进程提供给 AMS 进程的 `Binder` 对象，现在 AMS 通过这个 `Binder` 对象进行 IPC 调用通知广播接受者所在进程完成余下操作。在上文我们分析广播的注册过程中提到过，这个 `IIntentReceiver` 的实现是 `LoadedApk.ReceiverDispatcher`；我们查看这个对象的 `performReceive` 方法，源码如下：

```
public void performReceive(Intent intent, int resultCode, String data,
 Bundle extras, boolean ordered, boolean sticky, int sendingUser) {
 Args args = new Args(intent, resultCode, data, extras, ordered,
 sticky, sendingUser);
 if (!mActivityThread.post(args)) {
 if (mRegistered && ordered) {
 IActivityManager mgr = ActivityManagerNative.getDefault();
 args.sendFinished(mgr);
 }
 }
}
```

这个方法创建了一个Args对象，然后把它post到了mActivityThread这个Handler中；我们查看Args类的run方法：(坚持一下，马上就分析完了 ^ ^)

```

public void run() {
 final BroadcastReceiver receiver = mReceiver;
 final boolean ordered = mOrdered;
 final IActivityManager mgr = ActivityManagerNative.getDefault();
 final Intent intent = mCurIntent;
 mCurIntent = null;

 if (receiver == null || mForgotten) {
 if (mRegistered && ordered) {
 sendFinished(mgr);
 }
 return;
 }

 try {
 ClassLoader cl = mReceiver.getClass().getClassLoader(); // Important!! load
 intent.setExtrasClassLoader(cl);
 setExtrasClassLoader(cl);
 receiver.setPendingResult(this);
 receiver.onReceive(mContext, intent); // callback
 } catch (Exception e) {
 if (mRegistered && ordered) {
 sendFinished(mgr);
 }
 if (mInstrumentation == null ||
 !mInstrumentation.onException(mReceiver, e)) {
 throw new RuntimeException(
 "Error receiving broadcast " + intent
 + " in " + mReceiver, e);
 }
 }

 if (receiver.getPendingResult() != null) {
 finish();
 }
}

```

这里，我们看到了相应BroadcastReceiver的onReceive回调；因此，广播的工作原理到这里就水落石出了；我们接下来将探讨如何实现对于广播的插件化。

## 思路分析

上文中我们分析了BroadcastReceiver的工作原理，那么怎么才能实现对BroadcastReceiver的插件化呢？

从分析过程中我们发现，Framework对于静态广播和动态广播的处理是不同的；不过，这个不同之处仅仅体现在注册过程——静态广播需要在AndroidManifest.xml中注册，并且注册的信息存储在PMS中；动态广播不需要预注册，注册的信息存储在AMS中。

从实现Activity的插件化过程中我们知道，需要在AndroidManifest.xml中预先注册是一个相当麻烦的事情——我们需要使用『替身』并在合适的时候进行『偷梁换柱』；因此看起来动态广播的处理要容易那么一点，我们先讨论一下如何实现动态注册BroadcastReceiver的插件化。

首先，广播并没有复杂的生命周期，它的整个存活过程其实就是一个onReceive回调；而动态广播又不需要在AndroidManifest.xml中预先注册，所以动态注册的BroadcastReceiver其实可以当作一个普通的Java对象；我们完全可以用纯ClassLoader技术实现它——不就是把插件中的Receiver加载进来，然后想办法让它能接受onReceive回调嘛。

静态BroadcastReceiver看起来要复杂一些，但是我们连Activity都搞定了，还有什么难得倒我们呢？对于实现静态BroadcastReceiver插件化的问题，有的童鞋或许会想，我们可以借鉴Activity的工作方式——用替身和Hook解决。但是很遗憾，这样是行不通的。为什么呢？

BroadcastReceiver有一个IntentFilter的概念，也就是说，每一个BroadcastReceiver只对特定的Broadcast感兴趣；而且，AMS在进行广播分发的时候，也会对这些BroadcastReceiver与发出的广播进行匹配，只有Intent匹配的Receiver才能收到广播；在分析源码的时候也提到了这个匹配过程。如果我们尝试用替身Receiver解决静态注册的问题，那么它的IntentFilter该写什么？我们无法预料插件中静态注册的Receiver会使用什么类型的IntentFilter，就算我们在AndroidManifest.xml中声明替身也没有用——我们压根儿收不到与我们的IntentFilter不匹配的广播。其实，我们对于Activity的处理方式也有这个问题；如果你尝试用IntentFilter的方式启动Activity，这并不能成功；这算得上是DroidPlugin的缺陷之一。

那么，我们就真的对静态BroadcastReceiver无能为力吗？想一想这里的难点是什么？

没错，主要是在静态BroadcastReceiver里面这个IntentFilter我们事先无法确定，它是动态变化的；但是，动态BroadcastReceiver不是可以动态添加IntentFilter吗！！！

## 可以把静态广播当作动态广播处理

既然都是广播，它们的功能都是订阅一个特定的消息然后执行某个特定的操作，我们完全可以把插件中的静态广播全部注册为动态广播，这样就解决了静态广播的问题。当然，这样也是有缺陷的，静态BroadcastReceiver与动态BroadcastReceiver一个非常大的不同之

处在于：动态BroadcastReceiver在进程死亡之后是无法接收广播的，而静态BroadcastReceiver则可以——系统会唤醒Receiver所在进程；这算得上缺陷之二，当然，瑕不掩瑜。

## 静态广播非静态的实现

上文我们提到，可以把静态BroadcastReceiver当作动态BroadcastReceiver处理；我们接下来实现这个过程。

### 解析

要把插件中的静态BroadcastReceiver当作动态BroadcastReceiver处理，我们首先得知道插件中到底注册了哪些广播；这个过程归根结底就是获取AndroidManifest.xml中的`<receiver>`标签下面的内容，我们可以选择手动解析xml文件；这里我们选择使用系统的 PackageParser 帮助解析，这种方式在之前的 [插件加载过程] 中也用到过，如果忘记了可以温习一下。

PackageParser中有一系列方法用来提取Apk中的信息，可是翻遍了这个类也没有找到与「Receiver」名字相关的方法；最终我们发现BroadcastReceiver信息是用与Activity相同的类存储的！这一点可以在PackageParser的内部类Package中发现端倪——成员变量`receivers`和`activities`的范型类型相同。所以，我们要解析apk的信息，可以使用 PackageParser的`generateActivityInfo`方法。

知道这一点之后，代码就比较简单了；使用反射调用相应的隐藏接口，并且在必要的时候构造相应参数的方式我们在插件化系列文章中已经讲述过很多，相信读者已经熟练，这里就不赘述，直接贴代码：

```

private static void parserReceivers(File apkFile) throws Exception {
 Class<?> packageParserClass = Class.forName("android.content.pm.PackageParser");
 Method parsePackageMethod = packageParserClass.getDeclaredMethod("parsePackage",
 File.class);

 Object packageParser = packageParserClass.newInstance();

 // 首先调用parsePackage获取到apk对象对应的Package对象
 Object packageObj = parsePackageMethod.invoke(packageParser, apkFile, PackageManager.class);

 // 读取Package对象里面的receivers字段,注意这是一个 List<Activity> (没错,底层把<receiver>
 // 接下来要做的就是根据这个List<Activity> 获取到Receiver对应的 ActivityInfo (依然是把receiv
 Field receiversField = packageObj.getClass().getDeclaredField("receivers");
 List receivers = (List) receiversField.get(packageObj);

 // 调用generateActivityInfo 方法, 把PackageParser.Activity 转换成
 Class<?> packageParser$ActivityClass = Class.forName("android.content.pm.PackageParser$Activity");
 Class<?> packageUserStateClass = Class.forName("android.content.pm.PackageUserState");
 Class<?> userHandler = Class.forName("android.os.UserHandle");
 Method getCallingUserIdMethod = userHandler.getDeclaredMethod("getCallingUserId");
 int userId = (Integer) getCallingUserIdMethod.invoke(null);
 Object defaultUserState = packageUserStateClass.newInstance();

 Class<?> componentClass = Class.forName("android.content.pm.PackageParser$ComponentInfo");
 Field intentsField = componentClass.getDeclaredField("intents");

 // 需要调用 android.content.pm.PackageParser#generateActivityInfo(android.content.pm.Activity)
 Method generateReceiverInfo = packageParserClass.getDeclaredMethod("generateActivityInfo",
 packageParser$ActivityClass, int.class, packageUserStateClass, int.class);

 // 解析出 receiver以及对应的 intentFilter
 for (Object receiver : receivers) {
 ActivityInfo info = (ActivityInfo) generateReceiverInfo.invoke(packageParser,
 receiver);
 List<? extends IntentFilter> filters = (List<? extends IntentFilter>) intentsField.get(info);
 info.setIntentFilters(filters);
 intentFiltersCache.put(info, filters);
 }
}

```

## 注册

我们已经解析得到了插件中静态注册的BroadcastReceiver的信息，现在我们只需要把这些静态广播动态注册一遍就可以了；但是，由于BroadcastReceiver的实现类存在于插件之后，我们需要手动用ClassLoader来加载它；这一点在[插件加载机制](#)已有讲述，不啰嗦了。

```
ClassLoader cl = null;
for (ActivityInfo activityInfo : ReceiverHelper.sCache.keySet()) {
 Log.i(TAG, "preload receiver:" + activityInfo.name);
 List<? extends IntentFilter> intentFilters = ReceiverHelper.sCache.get(activityIn
 if (cl == null) {
 cl = CustomClassLoader.getPluginClassLoader(apk, activityInfo.packageName);
 }

 // 把解析出来的每一个静态Receiver都注册为动态的
 for (IntentFilter intentFilter : intentFilters) {
 BroadcastReceiver receiver = (BroadcastReceiver) cl.loadClass(activityInfo.na
 context.registerReceiver(receiver, intentFilter);
 }
}
```

就这样，我们对插件静态BroadcastReceiver的支持已经完成了，是不是相当简单？至于插件中的动态广播如何实现插件化，这一点交给读者自行完成，希望你在解决这个问题的过程中能够加深对于插件方案的理解 ^ ^

## 小节

本文我们介绍了BroadcastReceiver组件的插件化方式，可以看到，插件方案对于BroadcastReceiver的处理相对简单；同时「静态广播非静态」的特性以及BroadcastReceiver先天的一些特点导致插件方案没有办法做到尽善尽美，不过这都是大醇小疵——在绝大多数情况下，这样的处理方式是可以满足需求的。

虽然对于BroadcastReceiver的处理方式相对简单，但是文章的内容却并不短——我们花了大量的篇幅讲述BroadcastReceiver的原理，这也是我的初衷：借助DroidPlugin更深入地了解Android Framework。

接下来为文章会讲述四大组件中的另外两个——Service和ContentProvider的插件化方案；喜欢就点个赞吧～持续更新，请关注github项目 [understand-plugin-framework](#) 和我的 [博客](#)！如果你觉得能从文中学到皮毛，还请支持一下 :)

# Android 插件化原理解析——Service的插件化

来源:[Weishu's Notes](#)

在 [Activity生命周期管理](#) 以及 [广播的管理](#) 中我们详细探讨了Android系统中的Activity、BroadcastReceiver组件的工作原理以及它们的插件化方案，相信读者已经对Android Framework和插件化技术有了一定的了解；本文将探讨Android四大组件之一——Service组件的插件化方式。

与Activity, BroadcastReceiver相比，Service组件的不同点在哪里呢？我们能否用与之相同的方式实现Service的插件化？如果不行，它们的差别在哪里，应该如何实现Service的插件化？

我们接下来将围绕这几个问题展开，最终给出Service组件的插件化方式；阅读本文之前，可以先clone一份 [understand-plugin-framework](#)，参考此项目的 service-management 模块。另外，插件框架原理解析系列[文章见索引](#)。

## Service工作原理

连Service的工作原理都不了解，谈何插件化？知己知彼。

Service分为两种形式：以startService启动的服务和用bindService绑定的服务；由于这两个过程大体相似，这里以稍复杂的bindService为例分析Service组件的工作原理。

绑定Service的过程是通过Context类的bindService完成的，这个方法需要三个参数：第一个参数代表想要绑定的Service的Intent，第二个参数是一个ServiceConnection，我们可以通过这个对象接收到Service绑定成功或者失败的回调；第三个参数则是绑定时候的一些FLAG；关于服务的基本概念，可以参阅 [官方文档](#)。（现在汉化了哦，E文不好童鞋的福音）

Context的具体实现在ContextImpl类，ContextImpl中的bindService方法直接调用了 bindServiceCommon方法，此方法源码如下：

```

private boolean bindServiceCommon(Intent service, ServiceConnection conn, int flags,
 UserHandle user) {
 IServiceConnection sd;
 if (conn == null) {
 throw new IllegalArgumentException("connection is null");
 }
 if (mPackageInfo != null) {
 // important
 sd = mPackageInfo.getServiceDispatcher(conn, getOuterContext(),
 mMainThread.getHandler(), flags);
 } else {
 throw new RuntimeException("Not supported in system context");
 }
 validateServiceIntent(service);
 try {
 IBinder token = getActivityToken();
 if (token == null && (flags&BIND_AUTO_CREATE) == 0 && mPackageInfo != null
 && mPackageInfo.getApplicationInfo().targetSdkVersion
 < android.os.Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
 flags |= BIND_WAIVE_PRIORITY;
 }
 service.prepareToLeaveProcess();
 int res = ActivityManagerNative.getDefault().bindService(
 mMainThread.getApplicationThread(), getActivityToken(), service,
 service.resolveTypeIfNeeded(getContentResolver()),
 sd, flags, getOpPackageName(), user.getIdentifier());
 if (res < 0) {
 throw new SecurityException(
 "Not allowed to bind to service " + service);
 }
 return res != 0;
 } catch (RemoteException e) {
 throw new RuntimeException("Failure from system", e);
 }
}

```

大致观察就能发现这个方法最终通过ActivityManagerNative借助AMS进而完成Service的绑定过程，在跟踪AMS的bindService源码之前，我们关注一下这个方法开始处创建的sd变量。这个变量的类型是IServiceConnection，如果读者还有印象，我们在[广播的管理](#)一文中也遇到过类似的处理方式——IIntentReceiver；所以，这个IServiceConnection与IApplicationThread以及IIntentReceiver相同，都是ActivityThread给AMS提供的用来与之进行通信的Binder对象；这个接口的实现类为LoadedApk.ServiceDispatcher。

这个方法最终调用了ActivityManagerNative的bindService，而这个方法的真正实现在AMS里面，源码如下：

```
public int bindService(IApplicationThread caller, IBinder token, Intent service,
 String resolvedType, IServiceConnection connection, int flags, String callingP
 int userId) throws TransactionTooLargeException {
 enforceNotIsolatedCaller("bindService");
 // 略去参数校检
 synchronized(this) {
 return mServices.bindServiceLocked(caller, token, service,
 resolvedType, connection, flags, callingPackage, userId);
 }
}
```

bindService这个方法相当简单，只是做了一些参数校检之后直接调用了ActivityServices类的bindServiceLocked方法：

```

int bindServiceLocked(IApplicationThread caller, IBinder token, Intent service,
 String resolvedType, IServiceConnection connection, int flags,
 String callingPackage, int userId) throws TransactionTooLargeException {
 final ProcessRecord callerApp = mAm.getRecordForAppLocked(caller);
 // 参数校检, 略

 ServiceLookupResult res =
 retrieveServiceLocked(service, resolvedType, callingPackage,
 Binder.getCallingPid(), Binder.getCallingUid(), userId, true, callerFg);
 // 结果校检, 略
 ServiceRecord s = res.record;

 final long origId = Binder.clearCallingIdentity();

 try {
 // ... 不关心, 略

 mAm.startAssociationLocked(callerApp.uid, callerApp.processName,
 s.appInfo.uid, s.name, s.processName);

 AppBindRecord b = s.retrieveAppBindingLocked(service, callerApp);
 ConnectionRecord c = new ConnectionRecord(b, activity,
 connection, flags, clientLabel, clientIntent);
 IBinder binder = connection.asBinder();
 ArrayList<ConnectionRecord> clist = s.connections.get(binder);

 // 对connection进行处理, 方便存取, 略
 clist.add(c);

 if ((flags&Context.BIND_AUTO_CREATE) != 0) {
 s.lastActivity = SystemClock.uptimeMillis();
 if (bringUpServiceLocked(s, service.getFlags(), callerFg, false) != null)
 return 0;
 }
 }

 // 与BIND_AUTO_CREATE不同的启动FLAG, 原理与后续相同, 略

} finally {
 Binder.restoreCallingIdentity(origId);
}

return 1;
}

```

这个方法比较长，我这里省去了很多无关代码，只列出关键逻辑；首先它通过retrieveServiceLocked方法获取到了intent匹配到的需要bind到的Service组件res；然后把ActivityThread传递过来的IServiceConnection使用ConnectionRecord进行了包装，方便

接下来使用；最后如果启动的FLAG为BIND\_AUTO\_CREATE，那么调用bringUpServiceLocked开始创建Service，我们跟踪这个方法：（非这种FLAG的代码已经省略，可以自行跟踪）

```
private final String bringUpServiceLocked(ServiceRecord r, int intentFlags, boolean e
 boolean whileRestarting) throws TransactionTooLargeException {

 // 略。。

 final boolean isolated = (r.serviceInfo.flags&ServiceInfo.FLAG_ISOLATED_PROCESS)
 final String procName = r.processName;
 ProcessRecord app;

 if (!isolated) {
 app = mAm.getProcessRecordLocked(procName, r.appInfo.uid, false);
 if (app != null && app.thread != null) {
 try {
 app.addPackage(r.appInfo.packageName, r.appInfo.versionCode, mAm.mPro
 // 1. important !!!
 realStartServiceLocked(r, app, execInFg);
 return null;
 } catch (TransactionTooLargeException e) {
 throw e;
 } catch (RemoteException e) {
 Slog.w(TAG, "Exception when starting service " + r.shortName, e);
 }

 }
 } else {
 app = r.isolatedProc;
 }

 // Not running -- get it started, and enqueue this service record
 // to be executed when the app comes up.
 if (app == null) {
 // 2. important !!!
 if ((app=mAm.startProcessLocked(procName, r.appInfo, true, intentFlags,
 "service", r.name, false, isolated, false)) == null) {
 bringDownServiceLocked(r);
 return msg;
 }
 if (isolated) {
 r.isolatedProc = app;
 }
 }
 // 略。。
 return null;
}
```

这个方案同样也很长，但是实际上非常简单：注意我注释的两个important的地方，如果Service所在的进程已经启动，那么直接调用realStartServiceLocked方法来真正启动Service组件；如果Service所在的进程还没有启动，那么先在AMS中记下这个要启动的Service组件，然后通过startProcessLocked启动新的进程。

我们先看Service进程已经启动的情况，也即realStartServiceLocked分支：

```

private final void realStartServiceLocked(ServiceRecord r,
 ProcessRecord app, boolean execInFg) throws RemoteException {

 // 略。。

 boolean created = false;
 try {
 synchronized (r.stats.getBatteryStats()) {
 r.stats.startLaunchedLocked();
 }
 mAm.ensurePackageDexOpt(r.serviceInfo.packageName);
 app.forceProcessStateUpTo(ActivityManager.PROCESS_STATE_SERVICE);
 app.thread.scheduleCreateService(r, r.serviceInfo,
 mAm.compatibilityInfoForPackageLocked(r.serviceInfo.applicationInfo),
 app.repProcState);
 r.postNotification();
 created = true;
 } catch (DeadObjectException e) {
 mAm.appDiedLocked(app);
 throw e;
 } finally {
 // 略。。
 }

 requestServiceBindingsLocked(r, execInFg);

 // 不关心，略。。
}

```

这个方法首先调用了app.thread的scheduleCreateService方法，我们知道，这是一个IApplicationThread对象，它是App所在进程提供给AMS的用来与App进程进行通信的Binder对象，这个Binder的Server端在ActivityThread的ApplicationThread类，因此，我们跟踪ActivityThread类，这个方法的实现如下：

```

public final void scheduleCreateService(IBinder token,
 ServiceInfo info, CompatibilityInfo compatInfo, int processState) {
 updateProcessState(processState, false);
 CreateServiceData s = new CreateServiceData();
 s.token = token;
 s.info = info;
 s.compatInfo = compatInfo;

 sendMessage(H.CREATE_SERVICE, s);
}

```

它不过是转发了一个消息给ActivityThread的H这个Handler，H类收到这个消息之后，直接调用了ActivityThread类的handleCreateService方法，如下：

```

private void handleCreateService(CreateServiceData data) {
 unscheduleGcIdler();

 LoadedApk packageInfo = getPackageInfoNoCheck(
 data.info.applicationInfo, data.compatInfo);
 Service service = null;
 try {
 java.lang.ClassLoader cl = packageInfo.getClassLoader();
 service = (Service) cl.loadClass(data.info.name).newInstance();
 } catch (Exception e) {
 }

 try {
 ContextImpl context = ContextImpl.createAppContext(this, packageInfo);
 context.setOuterContext(service);

 Application app = packageInfo.makeApplication(false, mInstrumentation);
 service.attach(context, this, data.info.name, data.token, app,
 ActivityManagerNative.getDefault());
 service.onCreate();
 mServices.put(data.token, service);
 try {
 ActivityManagerNative.getDefault().serviceDoneExecuting(
 data.token, SERVICE_DONE_EXECUTING_ANON, 0, 0);
 } catch (RemoteException e) {
 // nothing to do.
 }
 } catch (Exception e) {
 }
}

```

看到这段代码，是不是似曾相识？！没错，这里与Activity组件的创建过程如出一辙！所以这里就不赘述了，可以参阅[Activity生命周期管理](#)。

需要注意的是，这里Service类的创建过程与Activity是略微有点不同的，虽然都是通过ClassLoader通过反射创建，但是Activity却把创建过程委托给了Instrumentation类，而Service则是直接进行。

OK，现在ActivityThread里面的handleCreateService方法成功创建出了Service对象，并且调用了它的onCreate方法；到这里我们的Service已经启动成功。

scheduleCreateService这个Binder调用过程结束，代码又回到了AMS进程的realStartServiceLocked方法。这里我们不得不感叹Binder机制的精妙，如此简洁方便高效的跨进程调用，在进程之间来回穿梭，游刃有余。

realStartServiceLocked方法的代码如下：

```

private final void realStartServiceLocked(ServiceRecord r,
 ProcessRecord app, boolean execInFg) throws RemoteException {

 // 略。。

 boolean created = false;
 try {
 synchronized (r.stats.getBatteryStats()) {
 r.stats.startLaunchedLocked();
 }
 mAm.ensurePackageDexOpt(r.serviceInfo.packageName);
 app.forceProcessStateUpTo(ActivityManager.PROCESS_STATE_SERVICE);
 app.thread.scheduleCreateService(r, r.serviceInfo,
 mAm.compatibilityInfoForPackageLocked(r.serviceInfo.applicationInfo),
 app.repProcState);
 r.postNotification();
 created = true;
 } catch (DeadObjectException e) {
 mAm.appDiedLocked(app);
 throw e;
 } finally {
 // 略。。
 }

 requestServiceBindingsLocked(r, execInFg);

 // 不关心，略。。
}

```

这个方法在完成scheduleCreateService这个binder调用之后，执行了一个requestServiceBindingsLocked方法；看方法名好像于「绑定服务」有关，它简单地执行了一个遍历然后调用了另外一个方法：

```

private final boolean requestServiceBindingLocked(ServiceRecord r, IntentBindRecord i
 boolean execInFg, boolean rebind) throws TransactionTooLargeException {
 if (r.app == null || r.app.thread == null) {
 return false;
 }
 if ((!i.requested || rebind) && i.apps.size() > 0) {
 try {
 bumpServiceExecutingLocked(r, execInFg, "bind");
 r.app.forceProcessStateUpTo(ActivityManager.PROCESS_STATE_SERVICE);
 r.app.thread.scheduleBindService(r, i.intent.getIntent(), rebind,
 r.app.repProcState);
 // 不关心，略。。
 }
 return true;
 }
}

```

可以看到，这里又通过IApplicationThread这个Binder进行了一次IPC调用，我们跟踪ActivityThread类里面的ApplicationThread的scheduleBindService方法，发现这个方法不过通过Handler转发了一次消息，真正的处理代码在handleBindService里面：

```

private void handleBindService(BindServiceData data) {
 Service s = mServices.get(data.token);
 if (s != null) {
 try {
 data.intent.setExtrasClassLoader(s.getClassLoader());
 data.intent.prepareToEnterProcess();
 try {
 if (!data.rebind) {
 IBinder binder = s.onBind(data.intent);
 ActivityManagerNative.getDefault().publishService(
 data.token, data.intent, binder);
 } else {
 s.onRebind(data.intent);
 ActivityManagerNative.getDefault().serviceDoneExecuting(
 data.token, SERVICE_DONE_EXECUTING_ANON, 0, 0);
 }
 ensureJitEnabled();
 } catch (RemoteException ex) {
 }
 } catch (Exception e) {
 }
 }
}

```

我们要Bind的Service终于在这里完成了绑定！绑定之后又通过ActivityManagerNative这个Binder进行一次IPC调用，我们查看AMS的publishService方法，这个方法简单第调用了publishServiceLocked方法，源码如下：

```

void publishServiceLocked(ServiceRecord r, Intent intent, IBinder service) {
 final long origId = Binder.clearCallingIdentity();
 try {
 if (r != null) {
 Intent.FilterComparison filter
 = new Intent.FilterComparison(intent);
 IntentBindRecord b = r.bindings.get(filter);
 if (b != null && !b.received) {
 b.binder = service;
 b.requested = true;
 b.received = true;
 for (int conni=r.connections.size()-1; conni>=0; conni--) {
 ArrayList<ConnectionRecord> clist = r.connections.valueAt(conni);
 for (int i=0; i<clist.size(); i++) {
 ConnectionRecord c = clist.get(i);
 if (!filter.equals(c.binding.intent.intent)) {
 continue;
 }
 try {
 c.conn.connected(r.name, service);
 } catch (Exception e) {
 }
 }
 }
 }
 }
 }

 serviceDoneExecutingLocked(r, mDestroyingServices.contains(r), false);
}
} finally {
 Binder.restoreCallingIdentity(origId);
}
}

```

还记得我们之前提到的那个IServiceConnection吗？在bindServiceLocked方法里面，我们把这个IServiceConnection放到了一个ConnectionRecord的List中存放在ServiceRecord里面，这里所做的就是取出已经被Bind的这个Service对应的IServiceConnection对象，然后调用它的connected方法；我们说过，这个IServiceConnection也是一个Binder对象，它的Server端在LoadedApk.ServiceDispatcher里面。代码到这里已经很明确了，由于分析过程过长，再继续估计大家要瞌睡了；接下来的过程非常简单，感兴趣的读者自行查阅LoadedApk.ServiceDispatcher的connected方法，一路跟踪弄清楚ServiceConnection回调过程，完成最后的拼图！

最后提一点，以上我们分析了Service所在进程已经存在的情况，如果Service所在进程不存在，那么会调用startProcessLocked方法创建一个新的进程，并把需要启动的Service放在一个队列里面；创建进程的过程通过Zygote fork出来，进程创建成功之后会调用ActivityThread的main方法，在这个main方法里面间接调用到了AMS的attachApplication方法，在AMS的attachApplication里面会检查刚刚那个待启动Service队列里面的内容，并执行Service的启动操作；之后的启动过程与进程已经存在的情况下相同；可以自行分析。

## Service的插件化思路

现在我们已经明白了Service组件的工作原理，可对如何实现Service的插件化依然是一头雾水。

从上文的源码分析来看，Service组件与Activity有着非常多的相似之处：它们都是通过Context类完成启动，接着通过ActivityManagerNative进入AMS，最后又通过IApplicationThread这个Binder IPC到App进程的Binder线程池，然后通过H转发消息到App进程的主线程，最终完成组件生命周期的回调；对于Service组件，看起来好像可以沿用Activity组件的插件化方式：Hook掉ActivityManagerNative以及H类，但事实真的如此吗？

## Service与Activity的异同

Service组件和Activity组件有什么不同？这些不同使得我们对于插件化方案的选择又有什么影响？

## 用户交互对于生命周期的影响

首先，Activity与Service组件最大的不同点在于，Activity组件可以与用户进行交互；这一点意味着用户的行为会对Activity组件产生影响，对我们来说最重要的影响就是Activity组件的生命周期；用户点击按钮从界面A跳转到界面B，会引起A和B这两个Activity一系列生命周期的变化。而Service组件则代表后台任务，除了内存不足系统回收之外，它的生命周期完全由我们的代码控制，与用户的交互无关。

这意味着什么？

Activity组件的生命周期受用户交互影响，而这种变化只有Android系统才能感知，因此我们必须把插件的Activity交给系统管理，才能拥有完整的生命周期；但Service组件的生命周期不受外界因素影响，那么自然而然，我们可以手动控制它的生命周期，就像我们对于BroadcastReceiver的插件化方式一样！Activity组件的插件化无疑是复杂的，为了把

插件Activity交给系统管理进而拥有完整生命周期，我们设计了一个天衣无缝的方案骗过了AMS；既然Service的生命周期可以由我们自己控制，那么我们可以有更简单的方案实现它的插件化。

## Activity的任务栈

上文指出了Activity和Service组件在处理用户交互方面的不同，这使得我们对于Service组建的插件化可以选择一种较为简单的方式；也许你会问，那采用Activity插件化的那一套技术能够实现Service组件的插件化吗？

很遗憾，答案是不行的。虽然Activity的插件化技术更复杂，但是这种方案并不能完成Service组件的插件化——复杂的方案并不意味着它能处理更多的问题。

原因在于Activity拥有任务栈的概念。或许你觉得任务栈并不是什么了不起的东西，但是，这确实是Service组件与Activity组件插件化方式分道扬镳的根本原因。

任务栈的概念使得Activtiy的创建就代表着入栈，销毁则代表出栈；又由于Activity代表着与用户交互的界面，所以这个栈的深度不可能太深——Activity栈太深意味着用户需要狂点back键才能回到初始界面，这种体验显然有问题；因此，插件框架要处理的Activity数量其实是有限的，所以我们在AndroidManifest.xml中声明有限个StubActivity就能满足插件启动近乎无限个插件Activity的需求。

但是Service组件不一样，理论情况下，可以启动的Service组件是无限的——除了硬件以及内存资源，没有什么限制它的数目；如果采用Activity的插件化方式，就算我们在AndroidManifest.xml中声明再多的StubService，总有不能满足插件中要启动的Service数目的情况出现。也许有童鞋会说，可以用一个StubService对应多个插件Service，这确实能解决部分问题；但是，下面的这个区别让这种设想彻底泡汤。

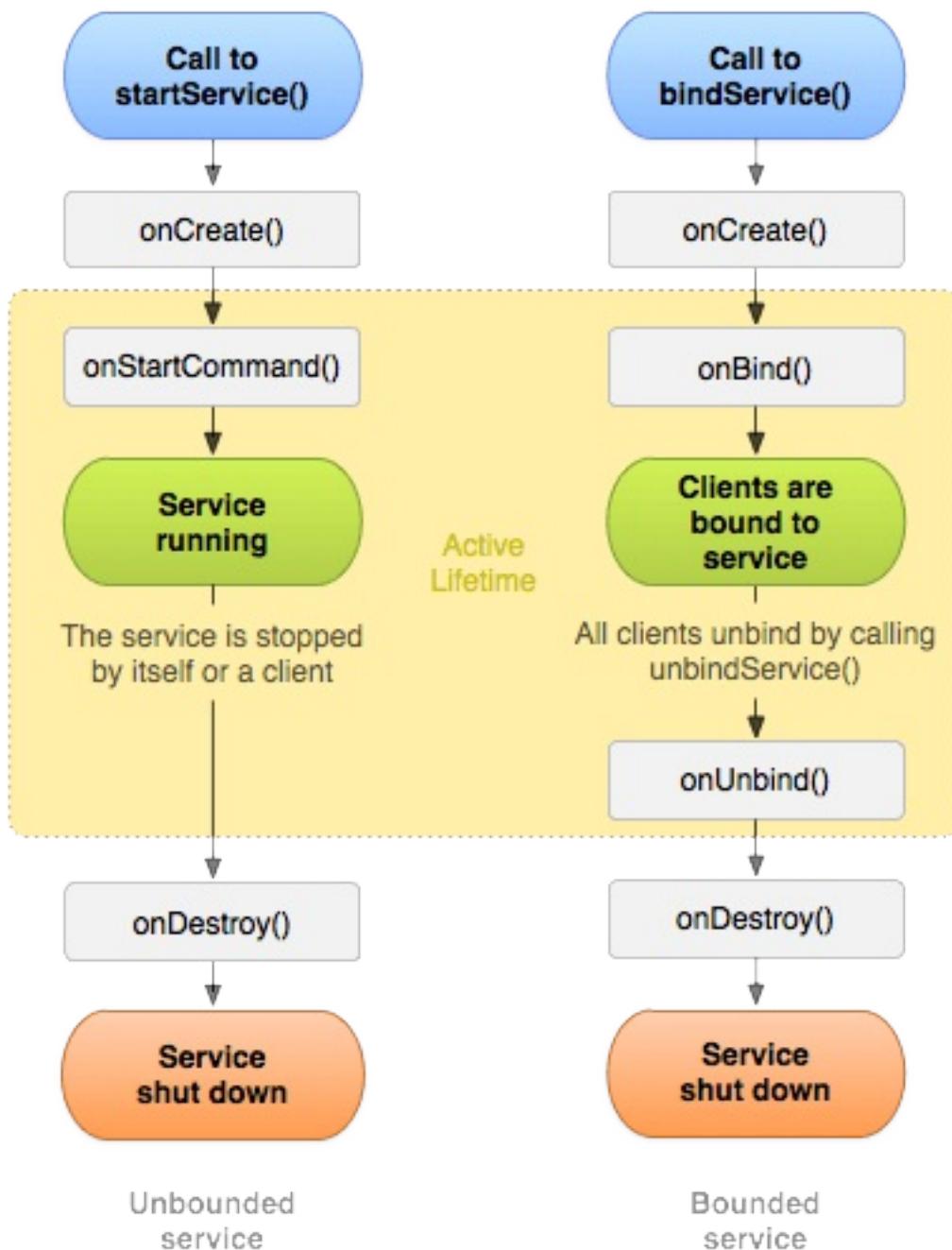
## Service无法拥有多实例

Service组件与Activity组件另外一个不同点在于，对同一个Service调用多次startService并不会启动多个Service实例，而非特定Flag的Activity是可以允许这种情况存在的，因此如果用StubService的方式，为了实现Service的这种特性，必须建立一个StubService到插件Service的一个Map，Map的这种一一对应关系使得我们使用一个StubService对应多个插件Service的计划成为天方夜谭。

至此，结论已经非常清晰——对于Service组件的插件化，我们不能简单地套用Activity的方案。

# 如何实现Service的插件化？

上文指出，我们不能套用Activity的方案实现Service组件的插件化，可以通过手动控制Service组件的生命周期实现；我们先来看一下Service的生命周期：



从图中可以看出，Service的生命周期相当简单：整个生命周期从调用 onCreate() 开始起，到 onDestroy() 返回时结束。对于非绑定服务，就是从startService调用到 stopService或者stopSelf调用。对于绑定服务，就是bindService调用到unbindService调用；

如果要手动控制Service组件的生命周期，我们只需要模拟出这个过程即可；而实现这一点并不复杂：

- 如果以startService方式启动插件Service，直接回调要启动的Service对象的

onStartCommand方法即可；如果用stopService或者stopSelf的方式停止Service，只需要回调对应的Service组件的onDestroy方法。

- 如果用bindService方式绑定插件Service，可以调用对应Service对应的onBind方法，获取onBind方法返回的Binder对象，然后通过ServiceConnection对象进行回调统计；unBindService的实现同理。

## 完全手动控制

现在我们已经有了实现思路，那么具体如何实现呢？

我们必须在startService,stopService等方法被调用的时候拿到控制权，才能手动去控制Service的生命周期；要达到这一目的非常简单——Hook ActivityManagerNative即可。在Activity的插件化方案中我们就通过这种方式接管了startActivity调用，相信读者并不陌生。

我们Hook掉ActivityManagerNative之后，可以拦截对于startService以及stopService等方法的调用；拦截之后，我们可以直接对插件Service进行操作：

- 拦截到startService之后，如果Service还没有创建就直接创建Service对象（可能需要加载插件），然后调用这个Service的onCreate,onStartCommand方法；如果Service已经创建，获取到原来创建的Service对象并执行其onStartCommand方法。
- 拦截到stopService之后，获取到对应的Service对象，直接调用这个Service的onDestroy方法。这种方案简直简单得让人不敢相信！很可惜，这么干是不行的。

首先，Service存在的意义在于它作为一个后台任务，拥有相对较高运行时优先级；除非在内存及其不足威胁到前台Activity的时候，这个组件才会被系统杀死。上述这种实现完全把Service当作一个普通的Java对象使用了，因此并没有完全实现Service所具备的能力。

其次，Activity以及Service等组件是可以指定进程的，而让Service运行在某个特定进程的情况非常常见——所谓的远程Service；用上述这种办法压根儿没有办法让某个Service对象运行在一个别的进程。Android系统给开发者控制进程的机会太少了，要么在AndroidManifest.xml中通过process属性指定，要么借助Java的Runtime类或者native的fork；这几种方式都无法让我们以一种简单的方式配合上述方案达到目的。

## 代理分发技术

既然我们希望插件的Service具有一定的运行时优先级，那么一个货真价实的Service组件是必不可少的——只有这种被系统认可的真正的Service组件才具有所谓的运行时优先级。

因此，我们可以注册一个真正的Service组件ProxyService，让这个Service承载一个真正的Service组件所具备的能力（进程优先级等）；当启动插件的服务比如PluginService的时候，我们统一启动这个ProxyService，当这个ProxyService运行起来之后，再在它的onStartCommand等方法里面进行分发，执行PluginService的onStartCommand等对应的方法；我们把这种方案形象地称为「代理分发技术」

代理分发技术也可以完美解决插件Service可以运行在不同的进程的问题——我们可以在AndroidManifest.xml中注册多个ProxyService，指定它们的process属性，让它们运行在不同的进程；当启动的插件Service希望运行在一个新的进程时，我们可以选择某一个合适的ProxyService进行分发。也许有童鞋会说，那得注册多少个ProxyService才能满足需求啊？理论上确实存在这问题，但事实上，一个App使用超过10个进程的几乎没有；因此这种方案是可行的。

## Service插件化的实现

现在我们已经设计出了Service组件的插件化方案，接下来我们以startService以及stopService为例实现这个过程。

### 注册代理Service

我们需要一个货真价实的Service组件来承载进程优先级等功能，因此需要在AndroidManifest.xml中声明一个或者多个（用以支持多进程）这样的Service：

```
<service
 android:name="com.weishu.upf.service_management.app.ProxyService"
 android:process="plugin01"/>
```

### 拦截startService等调用过程

要手动控制Service组件的生命周期，需要拦截startService,stopService等调用，并且把启动插件Service全部重定向为启动ProxyService（保留原始插件Service信息）；这个拦截过程需要Hook ActivityManagerNative，我们对这种技术应该是轻车熟路了；不了解的童鞋可以参考之前的文章[Hook机制之AMS&PMS](#)。

```
public static void hookActivityManagerNative() throws ClassNotFoundException,
 NoSuchMethodException, InvocationTargetException,
 IllegalAccessException, NoSuchFieldException {

 Class<?> activityManagerNativeClass = Class.forName("android.app.ActivityManagerNative");

 Field gDefaultField = activityManagerNativeClass.getDeclaredField("gDefault");
 gDefaultField.setAccessible(true);

 Object gDefault = gDefaultField.get(null);

 // gDefault是一个 android.util.Singleton对象; 我们取出这个单例里面的字段
 Class<?> singleton = Class.forName("android.util.Singleton");
 Field mInstanceField = singleton.getDeclaredField("mInstance");
 mInstanceField.setAccessible(true);

 // ActivityManagerNative 的gDefault对象里面原始的 IActivityManager对象
 Object rawIActivityManager = mInstanceField.get(gDefault);

 // 创建一个这个对象的代理对象, 然后替换这个字段, 让我们的代理对象帮忙干活
 Class<?> iActivityManagerInterface = Class.forName("android.app.IActivityManager");
 Object proxy = Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
 new Class<?>[] { iActivityManagerInterface }, new IActivityManagerHandler());
 mInstanceField.set(gDefault, proxy);
}
```

我们在收到startService,stopService之后可以进行具体的操作，对于startService来说，就是直接替换启动的插件Service为ProxyService等待后续处理，代码如下：

```

if ("startService".equals(method.getName())) {
 // API 23:
 // public ComponentName startService(IApplicationThread caller, Intent service,
 // String resolvedType, int userId) throws RemoteException
 // 找到参数里面的第一个Intent 对象
 Pair<Integer, Intent> integerIntentPair = foundFirstIntentOfArgs(args);
 Intent newIntent = new Intent();
 // 代理Service的包名，也就是我们自己的包名
 String stubPackage = UPFApplication.getContext().getPackageName();
 // 这里我们把启动的Service替换为ProxyService，让ProxyService接收生命周期回调
 ComponentName componentName = new ComponentName(stubPackage, ProxyService.class.get
 newIntent.setComponent(componentName);
 // 把我们原始要启动的TargetService先存起来
 newIntent.putExtra(AMSHookHelper.EXTRA_TARGET_INTENT, integerIntentPair.second);

 // 替换掉Intent，达到欺骗AMS的目的
 args[integerIntentPair.first] = newIntent;
 Log.v(TAG, "hook method startService success");
 return method.invoke(mBase, args);
}

```

对stopService的处理略有不同但是大同小异，读者可以上github查阅源码。

## 分发Service

Hook ActivityManagerNative之后，所有的插件Service的启动都被重定向到了我们注册的ProxyService，这样可以保证我们的插件Service有一个真正的Service组件作为宿主；但是要执行特定插件Service的任务，我们必须把这个任务分发到真正要启动的Service上去；以onStart为例，在启动ProxyService之后，会收到ProxyService的onStart回调，我们可以在这个方法里面把具体的任务交给原始要启动的插件Service组件：

```

public void onStart(Intent intent, int startId) {
 Log.d(TAG, "onStart() called with " + "intent = [" + intent + "], startId = [" + startId + "]");

 // 分发Service
 ServiceManager.getInstance().onStart(intent, startId);
 super.onStart(intent, startId);
}

```

## 加载Service

我们可以在ProxyService里面把任务转发给真正要启动的插件Service组件，要完成这个过程肯定需要创建一个对应的插件Service对象，比如PluginService；但是通常情况下插件存在与单独的文件之中，正常的方式是无法创建这个PluginService对象的，宿主程序默认的ClassLoader无法加载插件中对应的这个类；所以，要创建这个对应的PluginService对象，必须先完成插件的加载过程，让这个插件中的所有类都可以被正常访问；这种技术我们在之前专门讨论过，并给出了「激进方案」和「保守方案」，不了解的童鞋可以参考文章 插件加载机制；这里我选择代码较少的「保守方案」为例（Droid Plugin中采用的激进方案）：

```

public static void patchClassLoader(ClassLoader cl, File apkFile, File optDexFile)
 throws IllegalAccessException, NoSuchMethodException, IOException, InvocationTargetException, InstantiationException, IllegalAccessException {
 // 获取 BaseDexClassLoader : pathList
 Field pathListField = DexClassLoader.class.getSuperclass().getDeclaredField("pathList");
 pathListField.setAccessible(true);
 Object pathListObj = pathListField.get(cl);

 // 获取 PathList: Element[] dexElements
 Field dexElementArray = pathListObj.getClass().getDeclaredField("dexElements");
 dexElementArray.setAccessible(true);
 Object[] dexElements = (Object[]) dexElementArray.get(pathListObj);

 // Element 类型
 Class<?> elementClass = dexElements.getClass().getComponentType();

 // 创建一个数组，用来替换原始的数组
 Object[] newElements = (Object[]) Array.newInstance(elementClass, dexElements.length);

 // 构造插件Element(File file, boolean isDirectory, File zip, DexFile dexFile) 这个构造器
 Constructor<?> constructor = elementClass.getConstructor(File.class, boolean.class, File.class, DexFile.class);
 Object o = constructor.newInstance(apkFile, false, apkFile, DexFile.loadDex(apkFile));

 Object[] toAddElementArray = new Object[] { o };
 // 把原始的elements复制进去
 System.arraycopy(dexElements, 0, newElements, 0, dexElements.length);
 // 插件的那个element复制进去
 System.arraycopy(toAddElementArray, 0, newElements, dexElements.length, toAddElementArray.length);

 // 替换
 dexElementArray.set(pathListObj, newElements);

}

```

## 匹配过程

上文中我们把启动插件Service重定向为启动ProxyService，现在ProxyService已经启动，因此必须把控制权交回原始的PluginService；在加载插件的时候，我们存储了插件中所有的Service组件的信息，因此，只需要根据Intent里面的Component信息就可以取出对应的PluginService。

```
private ServiceInfo selectPluginService(Intent pluginIntent) {
 for (ComponentName componentName : mServiceInfoMap.keySet()) {
 if (componentName.equals(pluginIntent.getComponent())) {
 return mServiceInfoMap.get(componentName);
 }
 }
 return null;
}
```

## 创建以及分发

插件被加载之后，我们就需要创建插件Service对应的Java对象了；由于这些类是在运行时动态加载进来的，肯定不能直接使用new关键字——我们需要使用反射机制。但是下面的代码创建出插件Service对象能满足要求吗？

```
ClassLoader cl = getClassLoader();
Service service = cl.loadClass("com.plugin.www.PluginService1");
```

Service作为Android系统的组件，最重要的特点是它具有Context；所以，直接通过反射创建出来的这个PluginService就是一个壳子——没有Context的Service能干什么？因此我们需要给将要创建的Service类创建出Conetxt；但是Context应该如何创建呢？我们平时压根儿没有这么干过，Context都是系统给我们创建好的。既然这样，我们可以参照一下系统是如何创建Service对象的；在上文的Service源码分析中，在ActivityThread类的handleCreateService完成了这个步骤，摘要如下：

```

try {
 java.lang.ClassLoader cl = packageInfo.getClassLoader();
 service = (Service) cl.loadClass(data.info.name).newInstance();
} catch (Exception e) {
}

try {
 ContextImpl context = ContextImpl.createAppContext(this, packageInfo);
 context.setOuterContext(service);

 Application app = packageInfo.makeApplication(false, mInstrumentation);
 service.attach(context, this, data.info.name, data.token, app,
 ActivityManagerNative.getDefault());
 service.onCreate();
}

```

可以看到，系统也是通过反射创建出了对应的Service对象，然后也创建了对应的Context，并给Service注入了活力。如果我们模拟系统创建Context这个过程，势必需要进行一系列反射调用，那么我们何不直接反射handleCreateService方法呢？

当然，handleCreateService这个方法并没有把创建出来的Service对象作为返回值返回，而是存放在ActivityThread的成员变量mService之中，这个是小case，我们反射取出来就行；所以，创建Service对象的代码如下：

```

/**
 * 通过ActivityThread的handleCreateService方法创建出Service对象
 * @param serviceInfo 插件的ServiceInfo
 * @throws Exception
 */
private void proxyCreateService(ServiceInfo serviceInfo) throws Exception {
 IBinder token = new Binder();

 // 创建CreateServiceData对象，用来传递给ActivityThread的handleCreateService 当作参数
 Class<?> createServiceDataClass = Class.forName("android.app.ActivityThread$CreateServiceData");
 Constructor<?> constructor = createServiceDataClass.getDeclaredConstructor();
 constructor.setAccessible(true);
 Object createServiceData = constructor.newInstance();

 // 写入我们创建的createServiceData的token字段，ActivityThread的handleCreateService用这
 Field tokenField = createServiceDataClass.getDeclaredField("token");
 tokenField.setAccessible(true);
 tokenField.set(createServiceData, token);

 // 写入info对象
 // 这个修改是为了loadClass的时候，LoadedApk会是主程序的ClassLoader，我们选择Hook BaseDexClassLoader
 serviceInfo.applicationInfo.packageName = UPFApplication.getContext().getPackageName();
 Field infoField = createServiceDataClass.getDeclaredField("info");
 infoField.setAccessible(true);
 infoField.set(createServiceData, serviceInfo);
}

```

```
// 写入compatInfo字段
// 获取默认的compatibility配置
Class<?> compatibilityClass = Class.forName("android.content.res.CompatibilityInfo");
Field defaultCompatibilityField = compatibilityClass.getDeclaredField("DEFAULT_COMPATIBILITY");
Object defaultCompatibility = defaultCompatibilityField.get(null);
Field compatInfoField = createServiceDataClass.getDeclaredField("compatInfo");
compatInfoField.setAccessible(true);
compatInfoField.set(createServiceData, defaultCompatibility);

Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
Method currentActivityThreadMethod = activityThreadClass.getDeclaredMethod("getCurrentActivityThread");
Object currentActivityThread = currentActivityThreadMethod.invoke(null);

// private void handleCreateService(CreateServiceData data) {
Method handleCreateServiceMethod = activityThreadClass.getDeclaredMethod("handleCreateService");
handleCreateServiceMethod.setAccessible(true);

handleCreateServiceMethod.invoke(currentActivityThread, createServiceData);

// handleCreateService创建出来的Service对象并没有返回，而是存储在ActivityThread的mServices中
Field mServicesField = activityThreadClass.getDeclaredField("mServices");
mServicesField.setAccessible(true);
Map mServices = (Map) mServicesField.get(currentActivityThread);
Service service = (Service) mServices.get(token);

// 获取到之后，移除这个service，我们只是借花献佛
mServices.remove(token);

// 将此Service存储起来
mServiceMap.put(serviceInfo.name, service);
}
```

现在我们已经创建出了对应的PluginService，并且拥有至关重要的Context对象；接下来就可以把消息分发给原始的PluginService组件了，这个分发的过程很简单，直接执行消息对应的回调(onStart, onDestroy等) 即可；因此，完整的startService分发过程如下：

```

public void onStart(Intent proxyIntent, int startId) {

 Intent targetIntent = proxyIntent.getParcelableExtra(AMSHookHelper.EXTRA_TARGET_INTENT);
 ServiceInfo serviceInfo = selectPluginService(targetIntent);

 if (serviceInfo == null) {
 Log.w(TAG, "can not found service : " + targetIntent.getComponent());
 return;
 }
 try {

 if (!mServiceMap.containsKey(serviceInfo.name)) {
 // service还不存在, 先创建
 proxyCreateService(serviceInfo);
 }

 Service service = mServiceMap.get(serviceInfo.name);
 service.onStart(targetIntent, startId);
 } catch (Exception e) {
 e.printStackTrace();
 }
}

```

至此，我们已经实现了Service组件的插件化；完整的代码见 [github](#)，代码以startService, stopService为例进行了说明，bindService以及unbindService的原理是一样的，感兴趣的读者可以自行实现；欢迎PR。

## 小节

本文中我们以绑定服务为例分析了Service组件的工作原理，并指出用户交导致组件生命周期的变化是Activity与Service的根本差别，这种差别使得插件方案对于它们必须采取不同的处理方式；最后我们通过手动控制Service组件的生命周期结合「代理分发技术」成功地实现了Service组件的插件化；这种插件化方案堪称「完美」，如果非要吹毛求疵，那只能说由于同一个进程的所有Service都挂载在同一个ProxyService上面，如果系统可用内存不够必须回收Service，杀死一个ProxyService会导致一大票的插件Service歇菜。

实际使用过程中，Service组件的更新频度并不高，因此直接把插件Service注册到主程序也是可以接受的；而且如果需要绑定远程Service，完全可以使用一个Service组件根据不同的Intent返回不同的IBinder，所以不实现Service组件的插件化也能满足工程需要。值得一提的是，我们对于Service组件的插件化方案实际上是一种「代理」的方式，用这种方式也能实现Activity组件的插件化，有一些开源的插件方案比如 DL 就是这么做的。

迄今为止，我们讲述了Activity、BroadcastReceiver以及Service的插件化方式，不知读者思索过没有，实现插件化的关键点在哪里？

Service、Activity等不过就是一些普通的Java类，它们之所以称为四大组件，是因为他们有生命周期；这也是简单地采用Java的动态加载技术无法实现插件化的原因——动态加载进来的Service等类如果没有它的生命周期，无异于一个没有灵魂的傀儡。对于Activity组件，由于他的生命周期受用户交互影响，只有系统本身才能对这种交互有全局掌控力，因此它的插件化方式是Hook AMS，但是生命周期依然交由系统管理；而Service以及BroadcastReceiver的生命周期没有额外的因素影响，因此我们选择了手动控制其生命周期的方式。不论是借尸还魂还是女娲造人，对这些组件的插件化终归结底是要赋予组件“生命”。

插件化系列的文章有整整一个月没有更新了，非常抱歉！这段时间发生了很多事情，我实在抽不出时间照顾博客；而写这种文章又需要足够的时间准备，要跟踪源码分析过程，要找联系DroidPlugin作者确认设计思路，还要亲自写demo验证。

喜欢就点个赞吧，兜里有一块钱的童鞋可以点击下面的打赏然后扫一下二维码哦～持续更新，请关注github项目 [understand-plugin-framework](#) 和我的 [博客](#)！

# Android插件化原理解析——ContentProvider的插件化

来源:[Weishu's Notes](#)

目前为止我们已经完成了Android四大组件中Activity, Service以及BroadcastReceiver的插件化，这几个组件各不相同，我们根据它们的特点定制了不同的插件化方案；那么对于ContentProvider，它又有什么特点？应该如何实现它的插件化？

与Activity, Broadcast Receiver等频繁被使用的组件不同，我们接触和使用ContentProvider的机会要少得多；但是，ContentProvider这个组件对于Android系统有着特别重要的作用——作为一种极其方便的数据共享的手段，ContentProvider使得广大第三方App能够在壁垒森严的系统中自由呼吸。

在Android系统中，每一个应用程序都有自己的用户ID，而每一个应用程序所创建的文件的读写权限都是只赋予给自己所属的用户，这就限制了应用程序之间相互读写数据的操作。应用程序之间如果希望能够进行交互，只能采取跨进程通信的方式；Binder机制能够满足一般的IPC需求，但是如果应用程序之间需要共享大量数据，单纯使用Binder是很难办到的——我相信大家对于Binder 1M缓冲区以及TransactionTooLargeException一定不陌生；ContentProvider使用了匿名共享内存(Ashmem)机制完成数据共享，因此它可以很方便地完成大量数据的传输。Android系统的短信，联系人，相册，媒体库等等一系列的基础功能都依赖与ContentProvider，它的重要性可见一斑。

既然ContentProvider的核心特性是数据共享，那么要实现它的插件化，必须能让插件能够把它的ContentProvider共享给系统——如果不能「provide content」那还叫什么ContentProvider？

但是，如果回想一下Activity等组件的插件化方式，在涉及到「共享」这个问题上，一直没有较好的解决方案：

- 系统中的第三方App无法启动插件中带有特定IntentFilter的Activity，因为系统压根儿感受不到插件中这个真正的Activity的存在。
- 插件中的静态注册的广播并不真正是静态的，而是使用动态注册广播模拟实现的；这就导致如果宿主程序进程死亡，这个静态广播不会起作用；这个问题的根本原因在于BroadcastReceiver的IntentFilter的不可预知性，使得我们没有办法把静态广播真正“共享”给系统。
- 我们没有办法在第三方App中启动或者绑定插件中的Service组件；因为插件的Service并不是真正的Service组件，系统能感知到的只是那个代理Service；因此如果

插件如果带有远程Service组件，它根本不能给第三方App提供远程服务。

虽然在插件系统中一派生机勃勃的景象，Activity，Service等插件组件百花齐放，插件与宿主、插件与插件争奇斗艳；但是一旦脱离了插件系统的温室，这一片和谐景象不复存在：插件组件不过是傀儡而已；活着的，只有宿主——整个插件系统就是一座死寂的鬼城，各个插件组件借尸还魂般地依附在宿主身上，了无生机。

既然希望把插件的ContentProvider共享给整个系统，让第三方的App都能获取到我们插件共享的数据，我们必须解决这个问题；下文将会围绕这个目标展开，完成ContentProvider的插件化，并且顺带给出上述问题的解决方案。阅读本文之前，可以先clone一份[understand-plugin-framework](#)，参考此项目的 contentprovider-management 模块。另外，插件框架原理解析系列文章见[索引](#)。

## ContentProvider工作原理

首先我们还是得分析一下ContentProvider的工作原理，很多插件化的思路，以及一些Hook点的发现都严重依赖于对于系统工作原理的理解；对于ContentProvider的插件化，这一点特别重要。

### 铺垫工作

如同我们通过startActivity来启动Activity一样，与ContentProvider打交道的过程也是从Context类的一个方法开始的，这个方法叫做getContentResolver，使用ContentProvider的典型代码如下：

```
ContentResolver resolver = content.getContentResolver();
resolver.query(Uri.parse("content://authority/test"), null, null, null, null);
```

直接去ContextImpl类里面查找的 `getContentResolver` 实现，发现这个方法返回的类型是 `android.app.ContextImpl.ApplicationContentResolver`，这个类是抽象类 `android.content.ContentResolver` 的子类，`resolver.query` 实际上是调用父类 `ContentResolver` 的query实现：

```

public final @Nullable Cursor query(final @NonNull Uri uri, @Nullable String[] projection,
 @Nullable String selection, @Nullable String[] selectionArgs,
 @Nullable String sortOrder, @Nullable CancellationSignal cancellationSignal)
{
 Preconditions.checkNotNull(uri, "uri");
 IContentProvider unstableProvider = acquireUnstableProvider(uri);
 if (unstableProvider == null) {
 return null;
 }
 IContentProvider stableProvider = null;
 Cursor qCursor = null;
 try {
 long startTime = SystemClock.uptimeMillis();

 ICancellationSignal remoteCancellationSignal = null;
 if (cancellationSignal != null) {
 cancellationSignal.throwIfCanceled();
 remoteCancellationSignal = unstableProvider.createCancellationSignal();
 cancellationSignal.setRemote(remoteCancellationSignal);
 }
 try {
 qCursor = unstableProvider.query(mPackageName, uri, projection,
 selection, selectionArgs, sortOrder, remoteCancellationSignal);
 } catch (DeadObjectException e) {
 // The remote process has died... but we only hold an unstable
 // reference though, so we might recover!!! Let's try!!!!
 // This is exciting!!1!!1!!!!1
 unstableProviderDied(unstableProvider);
 stableProvider = acquireProvider(uri);
 if (stableProvider == null) {
 return null;
 }
 qCursor = stableProvider.query(mPackageName, uri, projection,
 selection, selectionArgs, sortOrder, remoteCancellationSignal);
 }
 // 略...
 }
}

```

注意这里面的那个try..catch语句，query方法首先尝试调用抽象方法acquireUnstableProvider拿到一个IContentProvider对象，并尝试调用这个“unstable”对象的query方法，万一调用失败（抛出DeadObjectException，熟悉Binder的应该了解这个异常）说明ContentProvider所在的进程已经死亡，这时候会尝试调用acquireProvider这个抽象方法来获取一个可用的IContentProvider（代码里面那个萌萌的注释说明了一切^\_^）；由于这两个acquire\*都是抽象方法，我们可以直接看子类ApplicationContentResolver的实现：

```

@Override
protected IContentProvider acquireUnstableProvider(Context c, String auth) {
 return mMainThread.acquireProvider(c,
 ContentProvider.getAuthorityWithoutUserId(auth),
 resolveUserIdFromAuthority(auth), false);
}
@Override
protected IContentProvider acquireProvider(Context context, String auth) {
 return mMainThread.acquireProvider(context,
 ContentProvider.getAuthorityWithoutUserId(auth),
 resolveUserIdFromAuthority(auth), true);
}

```

可以看到这两个抽象方法最终都通过调用ActivityThread类的acquireProvider获取到IContentProvider，接下来我们看看到底是如何获取到ContentProvider的。

## ContentProvider获取过程

ActivityThread类的acquireProvider方法如下，我们需要知道的是，方法的最后一个参数stable代表着ContentProvider所在的进程是否存活，如果进程已死，可能需要在必要的时候唤起这个进程；

```

public final IContentProvider acquireProvider(
 Context c, String auth, int userId, boolean stable) {
 final IContentProvider provider = acquireExistingProvider(c, auth, userId, stable);
 if (provider != null) {
 return provider;
 }

 IActivityManager.ContentProviderHolder holder = null;
 try {
 holder = ActivityManagerNative.getDefault().getContentProvider(
 getApplicationThread(), auth, userId, stable);
 } catch (RemoteException ex) {
 }
 if (holder == null) {
 Slog.e(TAG, "Failed to find provider info for " + auth);
 return null;
 }

 holder = installProvider(c, holder, holder.info,
 true /*noisy*/, holder.noReleaseNeeded, stable);
 return holder.provider;
}

```

这个方法首先通过acquireExistingProvider尝试从本进程中获取ContentProvider，如果获取不到，那么再请求AMS获取对应ContentProvider；想象一下，如果你查询的是自己App内部的ContentProvider组件，干嘛要劳烦AMS呢？不论是从哪里获取到的ContentProvider，获取完毕之后会调用installProvider来安装ContentProvider。

OK打住，我们思考一下，如果要实现ContentProvider的插件化，我们需要完成些什么工作？开篇的时候我提到了数据共享，那么具体来说，实现插件的数据共享，需要完成什么？ContentProvider是一个数据共享组件，也就是说它不过是一个**携带数据的载体而已**。为了支持跨进程共享，这个载体是**Binder调用**，为了共享大量数据，使用了匿名共享内存；这么说还是有点抽象，那么想一下，给出一个ContentProvider，你能对它做一些什么操作？如果能让插件支持这些操作，不就支持了插件化么？这就是典型的duck type思想——如果一个东西看起来像ContentProvider，用起来也像ContentProvider，那么它就是ContentProvider。

ContentProvider主要支持query, insert, update, delete操作，由于这个组件一般工作在别的进程，因此这些调用都是Binder调用。从上面的代码可以看到，这些调用最终都是委托给一个IContentProvider的Binder对象完成的，如果我们Hook掉这个对象，那么对于ContentProvider的所有操作都会被我们拦截掉，这时候我们可以做进一步的操作来完成对于插件ContentProvider组件的支持。要拦截这个过程，我们可以**假装插件的ContentProvider是自己App的ContentProvider**，也就是说，让acquireExistingProvider方法可以直接获取到插件的ContentProvider，这样我们就不需要欺骗AMS就能完成插件化了。当然，你也可以选择Hook掉AMS，让AMS的getContentProvider方法返回被我们处理过的对象，这也是可行的；但是，为什么要舍近求远呢？

从上文的分析暂时得出结论：我们可以把插件的ContentProvider信息预先放在App进程内部，使得对于ContentProvider执行CURD操作的时候，可以获取到插件的组件，这样或许就可以实现插件化了。具体来说，我们要做的事情就是让ActivityThread的acquireExistingProvider方法能够返回插件的ContentProvider信息，我们看看这个方法的实现：

```
public final IContentProvider acquireExistingProvider(
 Context c, String auth, int userId, boolean stable) {
 synchronized (mProviderMap) {
 final ProviderKey key = new ProviderKey(auth, userId);
 final ProviderClientRecord pr = mProviderMap.get(key);
 if (pr == null) {
 return null;
 }

 // 略。。
 }
}
```

可以看出，App内部自己的ContentProvider信息保存在ActivityThread类的mProviderMap中，这个map的类型是ArrayMap；我们当然可以通过反射修改这个成员变量，直接把插件的ContentProvider信息填进去，但是这个ProviderClientRecord对象如何构造？我们姑且看看系统自己是如何填充这个字段的。在ActivityThread类中搜索一遍，发现调用mProviderMap对象的put方法之后installProviderAuthoritiesLocked，而这个方法最终被installProvider方法调用。在分析ContentProvider的获取过程中我们已经知道，不论是通过本进程的acquireExistingProvider还是借助AMS的getContentProvider得到ContentProvider，最终都会对这个对象执行installProvider操作，也就是「安装」在本进程内部。那么，我们接着看这个installProvider做了什么，它是如何「安装」ContentProvider的。

## 进程内部ContentProvider安装过程

首先，如果之前没有“安装”过，那么holder为null，下面的代码会被执行，

```
final java.lang.ClassLoader cl = c.getClassLoader();
localProvider = (ContentProvider)cl.
 loadClass(info.name).newInstance();
provider = localProvider.getIContentProvider();
if (provider == null) {
 Slog.e(TAG, "Failed to instantiate class " +
 info.name + " from sourceDir " +
 info.applicationInfo.sourceDir);
 return null;
}
if (DEBUG_PROVIDER) Slog.v(
 TAG, "Instantiating local provider " + info.name);
// XXX Need to create the correct context for this provider.
localProvider.attachInfo(c, info);
```

比较直观，直接load这个ContentProvider所在的类，然后用反射创建出这个ContentProvider对象；但是由于查询是需要进行跨进程通信的，在本进程创建出这个对象意义不大，所以我们需要取出ContentProvider承载跨进程通信的Binder对象IContentProvider；创建出对象之后，接下来就是构建合适的信息，保存在ActivityThread内部，也就是mProviderMap：

```

if (localProvider != null) {
 ComponentName cname = new ComponentName(info.packageName, info.name);
 ProviderClientRecord pr = mLocalProvidersByName.get(cname);
 if (pr != null) {
 if (DEBUG_PROVIDER) {
 Slog.v(TAG, "installProvider: lost the race, "
 + "using existing local provider");
 }
 provider = pr.mProvider;
 } else {
 holder = new IActivityManager.ContentProviderHolder(info);
 holder.provider = provider;
 holder.noReleaseNeeded = true;
 pr = installProviderAuthoritiesLocked(provider, localProvider, holder);
 mLocalProviders.put(jBinder, pr);
 mLocalProvidersByName.put(cname, pr);
 }
 retHolder = pr.mHolder;
} else {
}

```

以上就是安装代码，不难理解。

## 思路尝试——本地安装

那么，了解了「安装」过程再结合上文的分析，我们似乎可以完成ContentProvider的插件化了——直接把插件的ContentProvider安装在进程内部就行了。如果插件系统有多个进程，那么必须在每个进程都「安装」一遍，如果你熟悉Android进程的启动流程那么就会知道，这个安装ContentProvider的过程适合放在Application类中，因为每个Android进程启动的时候，App的Application类是会被启动的。

看起来实现ContentProvider的思路有了，但是这里实际上有一个严重的缺陷！

我们依然没有解决「共享」的问题。我们只是在插件系统启动的进程里面的ActivityThread的mProviderMap给修改了，这使得只有通过插件系统启动的进程，才能感知到插件中的ContentProvider(因为我们手动把插件中的信息install到这个进程中去了)；如果第三方的App想要使用插件的ContentProvider，那系统只会告诉它查无此人。

那么，我们应该如何解决共享这个问题呢？看来还是逃不过AMS的魔掌，我们继续跟踪源码，看看如果在本进程查询不到ContentProvider，AMS是如何完成这个过程的。在ActivityThread的acquireProvider方法中我们提到，如果acquireExistingProvider方法返回null，会调用ActivityManagerNative的getContentProvider方法通过AMS查询整个系统中是否存在需要的这个ContentProvider。如果第三方App查询插件系统的ContentProvider必然走的是这个流程，我们仔细分析一下这个过程；

## AMS中的ContentProvider

首先我们查阅ActivityManagerService的getContentTypeProvider方法，这个方法间接调用了getContentTypeProviderImpl方法；getContentTypeProviderImpl方法体相当的长，但是实际上只做了两件事情（我这就不贴代码了，读者可以对着源码看一遍）：

- 使用PackageManagerService的resolveContentProvider根据Uri中提供的auth信息查阅对应的ContentProvider的信息ProviderInfo。
- 根据查询到的ContentProvider信息，尝试将这个ContentProvider组件安装到系统上。

## 查询ContentProvider组件的过程

查询ContentProvider组件的过程看起来很简单，直接调用PackageManager的resolveContentProvider就能从URI中获取到对应的ProviderInfo信息：

```

@Override
public ProviderInfo resolveContentProvider(String name, int flags, int userId) {
 if (!sUserManager.exists(userId)) return null;
 // reader
 synchronized (mPackages) {
 final PackageParser.Provider provider = mProvidersByAuthority.get(name);
 PackageSetting ps = provider != null
 ? mSettings.mPackages.get(provider.owner.packageName)
 : null;
 return ps != null
 && mSettings.isEnabledLPr(provider.info, flags, userId)
 && (!mSafeMode || (provider.info.applicationInfo.flags
 & ApplicationInfo.FLAG_SYSTEM) != 0)
 ? PackageParser.generateProviderInfo(provider, flags,
 ps.readUserState(userId), userId)
 : null;
 }
}

```

但是实际上我们关心的是，这个mProvidersByAuthority里面的信息是如何添加进PackageManagerService的，会在什么时候更新？在PackageManagerService这个类中搜索mProvidersByAuthority.put这个调用，会发现在scanPackageDirtyLI会更新mProvidersByAuthority这个map的信息，接着往前追踪会发现：这些信息是在Android系统启动的时候收集的。也就是说，Android系统在启动的时候会扫描一些App的安装目录，典型的比如/data/app/\*，获取这个目录里面的apk文件，读取其AndroidManifest.xml中的

信息，然后把这些信息保存在PackageManagerService中。合理猜测，在系统启动之后，安装新的App也会触发对新App中AndroidManifest.xml的操作，感兴趣的读者可以自行翻阅源码。

现在我们知道，查询ContentProvider的信息来源在Android系统启动的时候已经初始化好了，这个过程对于我们第三方app来说是鞭长莫及，想要使用类似在进程内部Hack ContentProvider的查找过程是不可能的。

## 安装ContentProvider组件的过程

获取到URI对应的ContentProvider的信息之后，接下来就是把它安装到系统上了，这样以后有别的查询操作就可以直接拿来使用；但是这个安装过程AMS是没有办法以一己之力完成的。想象一下App DemoA 查询App DemoB 的某个ContentProviderAppB，那么这个ContentProviderAppB必然存在于DemoB这个App中，AMS所在的进程(system\_server)连这个ContentProviderAppB的类都没有，因此，AMS必须委托DemoB完成它的ContentProviderAppB的安装；这里就分两种情况：其一， DemoB这个App已经在运行了，那么AMS直接通知DemoB安装ContentProviderAppB（如果B已经安装了那就更好了）；其二， DemoB这个app没在运行，那么必须把B进程唤醒，让它干活；这个过程也就是ActivityManagerService的getContentProviderImpl方法所做的，如下代码：

```

if (proc != null && proc.thread != null) {
 if (!proc.pubProviders.containsKey(cpi.name)) {
 proc.pubProviders.put(cpi.name, cpr);
 try {
 proc.thread.scheduleInstallProvider(cpi);
 } catch (RemoteException e) {
 }
 }
} else {
 proc = startProcessLocked(cpi.processName,
 cpr.appInfo, false, 0, "content provider",
 new ComponentName(cpi.applicationInfo.packageName,
 cpi.name), false, false, false);
 if (proc == null) {
 return null;
 }
}

```

如果查询的ContentProvider所在进程处于运行状态，那么AMS会通过这个进程给AMS的ApplicationThread这个Binder对象完成scheduleInstallProvider调用，这个过程比较简单，最终会调用到目标进程的installProvider方法，而这个方法我们在上文已经分析过了。我们看一下如果目标进程没有启动，会发生什么情况。

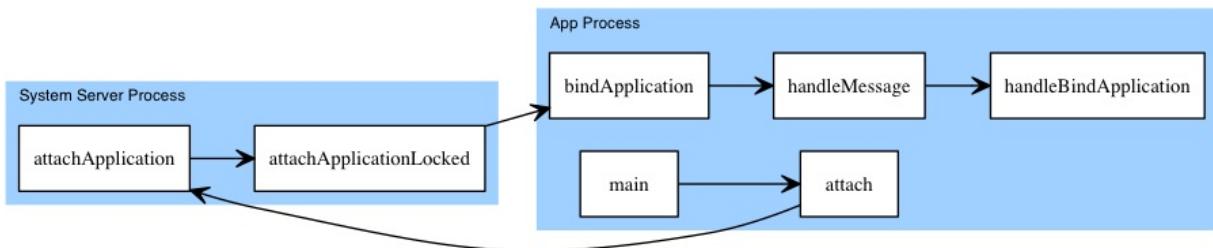
如果ContentProvider所在的进程已经死亡，那么会调用startProcessLocked来启动新的进程，startProcessLocked有一系列重载函数，我们一路跟踪，发现最终启动进程的操作交给了Process类的start方法完成，这个方法通过socket与Zygote进程进行通信，通知Zygote进程fork出一个子进程，然后通过反射调用了之前传递过来的一个入口类的main函数，一般来说这个入口类就是ActivityThread，因此子进程fork出来之后会执行ActivityThread类的main函数。

在我们继续观察子进程ActivityThread的main函数执行之前，我们看看AMS进程这时候会干什么——startProcessLocked之后AMS进程和fork出来的DemoB进程分道扬镳；AMS会继续往下面执行。我们暂时回到AMS的getContentProviderImpl方法：

```
// Wait for the provider to be published...
synchronized (cpr) {
 while (cpr.provider == null) {
 if (cpr.launchingApp == null) {
 return null;
 }
 try {
 if (conn != null) {
 conn.waiting = true;
 }
 cpr.wait();
 } catch (InterruptedException ex) {
 } finally {
 if (conn != null) {
 conn.waiting = false;
 }
 }
 }
}
```

你没看错，一个死循环就在上面：AMS进程会通过一个死循环等到进程B完成ContentProvider的安装，等待完成之后会把ContentProvider的信息返回给进程A。那么，我们现在的疑惑是，**进程B在启动之后，在哪个时间点会完成ContentProvider的安装呢？**

我们接着看ActivityThread的main函数，顺便寻找我们上面那个问题的答案；这个分析实际上就是Android App的启动过程，更详细的过程可以参阅老罗的文章 [Android应用程序启动过程源代码分析](#)，这里只给出简要调用流程：



最终，DemoB进程启动之后会执行ActivityThread类的handleBindApplication方法，这个方法相当之长，基本完成了App进程启动之后所有必要的操作；这里我们只关心ContentProvider相关的初始化操作，代码如下：

```

// If the app is being launched for full backup or restore, bring it up in
// a restricted environment with the base application class.
Application app = data.info.makeApplication(data.restrictedBackupMode, null);
mInitialApplication = app;

// don't bring up providers in restricted mode; they may depend on the
// app's custom Application class
if (!data.restrictedBackupMode) {
 List<ProviderInfo> providers = data.providers;
 if (providers != null) {
 installContentProviders(app, providers);
 // For process that contains content providers, we want to
 // ensure that the JIT is enabled "at some point".
 mH.sendEmptyMessageDelayed(H.ENABLE_JIT, 10*1000);
 }
}

// Do this after providers, since instrumentation tests generally start their
// test thread at this point, and we don't want that racing.
try {
 mInstrumentation.onCreate(data.instrumentationArgs);
}
catch (Exception e) {

}

try {
 mInstrumentation.callApplicationOnCreate(app);
} catch (Exception e) {
}

```

仔细观察以上代码，你会发现：ContentProvider的安装比Application的onCreate回调还要早！！因此，分析到这里我们已经明白了前面提出的那个问题，进程启动之后会在Aplication类的onCreate回调之前，在Application对象创建之后完成ContentProvider的安装。

然后不要忘了，我们的AMS进程还在那傻傻等待DemoB进程完成ContentProviderAppB的安装呢！在DemoB的Application的onCreate回调之前，DemoB的ContentProviderAppB已经安装好了，因此AMS停止等待，把DemoB安装的结果返回给请求这个ContentProvider的DemoA。我们必须对这个时序保持敏感，有时候就是失之毫厘，差之千里！！

到这里，有关ContentProvider的调用过程以及简要的工作原理我们已经分析完毕，关于它如何共享数据，如何使用匿名共享内存这部分不是插件化的重点，感兴趣的可以参考[Android应用程序组件Content Provider在应用程序之间共享数据的原理分析](#)。

## 不同之处

在实现ContentProvider的插件化之前，通过分析这个组件的工作原理，我们可以得出它的一些与众不同的特性：

- ContentProvider本身是用来共享数据的，因此它提供一般的CURD服务；它类似HTTP这种无状态的服务，没有Activity, Service所谓的生命周期的概念，服务要么可用，要么不可用；对应着ContentProvider要么启动，要么随着进程死亡；而通常情况下，死亡之后还会被系统启动。所以，ContentProvider，只要有人需要这个服务，系统可以保证是永生的；这是与其他组件的最大不同；完全不用考虑生命周期的概念。
- ContentProvider被设计为共享数据，这种数据量一般来说是相当大的；熟悉Binder的人应该知道，Binder进行数据传输有1M限制，因此如果要使用Binder传输大数据，必须使用类似socket的方式一段一段的读，也就是说需要自己在上层架设一层协议；ContentProvider并没有采取这种方式，而是采用了Android系统的匿名共享内存机制，利用Binder来传输这个文件描述符，进而实现文件的共享；这是第二个不同，因为其他的三个组建通信都是基于Binder的，只有ContentProvider使用了Ashmem。
- 一个App启动过程中，ContentProvider组件的启动是非常早的，甚至比Application的onCreate还要早；我们可以利用这个特性结合它不死的特点，完成一些有意义的事情。
- ContentProvider存在优先查询本进程的特点，使得它的插件化甚至不需要Hook AMS就能完成。

## 思路分析

在分析ContentProvider的工作原理的过程中我们提出了一种插件化方案：在进程启动之初，手动把ContentProvider安装到本进程，使得后续对于插件ContentProvider的请求能够顺利完成。我们也指出它的一个严重缺陷，那就是它只能在插件系统内部掩耳盗铃，在插件系统之外，第三方App依然无法感知到插件中的ContentProvider的存在。

如果插件的ContentProvider组件仅仅是为了共享给其他插件或者宿主程序使用，那么这种方案可以解决问题；不需要Hook AMS，非常简单。

但是，如果希望把插件ContentProvider共享给整个系统呢？在分析AMS中获取ContentProvider的过程中我们了解到，ContentProvider信息的注册是在Android系统启动或者新安装App的时候完成的，而AMS把ContentProvider返回给第三方App也是在system\_server进程完成；我们无法对其暗箱操作。

在完成Activity、Service组件的插件化之后，这种限制对我们来说已经是小case了：我们在宿主程序里面注册一个货真价实、被系统认可的StubContentProvider组件，把这个组件共享给第三方App；然后通过代理分发技术把第三方App对于插件ContentProvider的请求通过这个StubContentProvider分发给对应的插件。

但是这还存在一个问题，由于第三方App查阅的其实是StubContentProvider，因此他们查阅的URI也必然是StubContentProvider的authority，要查询到插件的ContentProvider，必须把要查询的真正的插件ContentProvider信息传递进来。这个问题的解决方案也很容易，我们可以制定一个「插件查询协议」来实现。

举个例子，假设插件系统的宿主程序在AndroidManifest.xml中注册了一个StubContentProvider，它的Authority为com.test.host\_authority；由于这个组件被注册在AndroidManifest.xml中，是系统认可的ContentProvider组件，整个系统都是可以使用这个共享组件的，使用它的URI一般为content://com.test.host\_authority；那么，如果插件系统中存在一个插件，这个插件提供了一个PluginContentProvider，它的Authority为com.test.plugin\_authorith，因为这个插件的PluginContentProvider没有在宿主程序的AndroidManifest.xml中注册（预先注册就失去插件的意义了），整个系统是无法感知到它的存在的；前面提到代理分发技术，也就是，我们让第三方App请求宿主程序的StubContentProvider，这个StubContentProvider把请求转发给合适的插件的ContentProvider就能完成了（插件内部通过预先installProvider可以查询所有的ContentProvider组件）；这个协议可以有很多，比如说：如果第三方App需要请求插件的StubContentProvider，可以以content://com.test.host\_authority/com.test.plugin\_authorith去查询系统；也就是说，我们假装请求StubContentProvider，把真正的需要请求的PluginContentProvider的Authority放在路径参数里面，StubContentProvider收到这个请求之后，拿到这个真正的Authority去请求插件的PluginContentProvider，拿到结果之后再返回给第三方App。

这样，我们通过「代理分发技术」以及「插件查询协议」可以完美解决「共享」的问题，开篇提到了我们之前对于Activity、Service组件插件化方案中对于「共享」功能的缺失，按照这个思路，基本可以解决这一系列问题。比如，对于第三方App无法绑定插件服务的

问题，我们可以注册一个StubService，把真正需要bind的插件服务信息放在intent的某个字段中，然后在StubService的onBind中解析出这个插件服务信息，然后去拿到插件Service组件的Binder对象返回给第三方。

## 实现

上文详细分析了如何实现ContentProvider的插件化，接下来我们就实现这个过程。

### 预先installProvider

要实现预先installProvider，我们首先需要知道，所谓的「预先」到底是在什么时候？

前文我们提到过App进程安装ContentProvider的时机非常之早，在Application类的onCreate回调执行之前已经完成了；这意味着什么？

现在我们对于ContentProvider插件化的实现方式是通过「代理分发技术」，也就是说在请求插件ContentProvider的时候会先请求宿主程序的StubContentProvider；如果一个第三方App查询插件的ContentProvider，而宿主程序没有启动的话，AMS会启动宿主程序并等待宿主程序的StubContentProvider完成安装，**一旦安装完成就会把得到的**

**IContentProvider**返回给这个第三方App；第三方App拿到IContentProvider这个Binder对象之后就可能发起CURD操作，如果这个时候插件ContentProvider还没有启动，那么肯定就会出异常；要记住，“这个时候”可能宿主程序的onCreate还没有执行完毕呢！！

所以，我们基本可以得出结论，预先安装这个所谓的「预先」必须早于Application的onCreate方法，在Android SDK给我们的回调里面，attachBaseContext这个方法是可以满足要求的，它在Application这个对象被创建之后就会立即调用。

解决了时机问题，那么我们接下来就可以安装ContentProvider了。

安装ContentProvider也就是要调用ActivityThread类的installProvider方法，这个方法需要的参数有点多，而且它的第二个参数IActivityManager.ContentProviderHolder是一个隐藏类，我们不知道如何构造，就算通过反射构造由于SDK没有暴露稳定性不易保证，我们看看有什么方法调用了这个installProvider。

installContentProviders这个方法直接调用installProvder看起来可以使用，但是它是一个private的方法，还有public的方法吗？继续往上寻找调用链，发现了installSystemProviders这个方法：

```
public final void installSystemProviders(List<ProviderInfo> providers) {
 if (providers != null) {
 installContentProviders(mInitialApplication, providers);
 }
}
```

但是，我们说过ContentProvider的安装必须相当早，必须在Application类的attachBaseContext方法内，而这个mInitialApplication字段是在onCreate方法调用之后初始化的，所以，如果直接使用这个installSystemProviders势必抛出空指针异常；因此，我们只有退而求其次，选择通过**installContentProviders**这个方法完成ContentProvider的安装

要调用这个方法必须拿到ContentProvider对应的ProviderInfo，这个我们在之前也介绍过，可以通过PackageParser类完成，当然这个类有一些兼容性问题，我们需要手动处理：

```

/**
 * 解析Apk文件中的 <provider>, 并存储起来
 * 主要是调用PackageParser类的generateProviderInfo方法
 *
 * @param apkFile 插件对应的apk文件
 * @throws Exception 解析出错或者反射调用出错, 均会抛出异常
 */
public static List<ProviderInfo> parseProviders(File apkFile) throws Exception {
 Class<?> packageParserClass = Class.forName("android.content.pm.PackageParser");
 Method parsePackageMethod = packageParserClass.getDeclaredMethod("parsePackage", |

 Object packageParser = packageParserClass.newInstance();

 // 首先调用parsePackage获取到apk对象对应的Package对象
 Object packageObj = parsePackageMethod.invoke(packageParser, apkFile, PackageManager |

 // 读取Package对象里面的services字段
 // 接下来要做的就是根据这个List<Provider> 获取到Provider对应的ProviderInfo
 Field providersField = packageObj.getClass().getDeclaredField("providers");
 List providers = (List) providersField.get(packageObj);

 // 调用generateProviderInfo 方法, 把PackageParser.Provider转换成ProviderInfo
 Class<?> packageParser$ProviderClass = Class.forName("android.content.pm.PackageP |
 Class<?> packageUserStateClass = Class.forName("android.content.pm.PackageUserSta |
 Class<?> userHandler = Class.forName("android.os.UserHandle");
 Method getCallingUserIdMethod = userHandler.getDeclaredMethod("getCallingUserId");
 int userId = (Integer) getCallingUserIdMethod.invoke(null);
 Object defaultUserState = packageUserStateClass.newInstance();

 // 需要调用 android.content.pm.PackageParser#generateProviderInfo
 Method generateProviderInfo = packageParserClass.getDeclaredMethod("generateProvide |
 packageParser$ProviderClass, int.class, packageUserStateClass, int.class)

 List<ProviderInfo> ret = new ArrayList<>();
 // 解析出intent对应的Provider组件
 for (Object service : providers) {
 ProviderInfo info = (ProviderInfo) generateProviderInfo.invoke(packageParser, |
 ret.add(info);
 }

 return ret;
}

```

解析出ProviderInfo之后，就可以直接调用installContentProvider了：

```
/**
 * 在进程内部安装provider，也就是调用 ActivityThread.installContentProviders方法
 *
 * @param context you know
 * @param apkFile
 * @throws Exception
 */
public static void installProviders(Context context, File apkFile) throws Exception {
 List<ProviderInfo> providerInfos = parseProviders(apkFile);

 for (ProviderInfo providerInfo : providerInfos) {
 providerInfo.applicationInfo.packageName = context.getPackageName();
 }

 Log.d("test", providerInfos.toString());
 Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
 Method currentActivityThreadMethod = activityThreadClass.getDeclaredMethod("currentActivityThread");
 Object currentActivityThread = currentActivityThreadMethod.invoke(null);
 Method installProvidersMethod = activityThreadClass.getDeclaredMethod("installContentProviders");
 installProvidersMethod.setAccessible(true);
 installProvidersMethod.invoke(currentActivityThread, context, providerInfos);
}
```

整个安装过程必须在Application类的**attachBaseContext**里面完成：

```

/**
 * 一定需要Application，并且在attachBaseContext里面Hook
 * 因为provider的初始化非常早，比Application的onCreate还要早
 * 在别的地方hook都晚了。
 *
 * @author weishu
 * @date 16/3/29
 */
public class UPFApplication extends Application {

 @Override
 protected void attachBaseContext(Context base) {
 super.attachBaseContext(base);

 try {
 File apkFile = getFileStreamPath("testcontentprovider-debug.apk");
 if (!apkFile.exists()) {
 Utils.extractAssets(base, "testcontentprovider-debug.apk");
 }

 File odexFile = getFileStreamPath("test.odex");

 // Hook ClassLoader，让插件中的类能够被成功加载
 BaseDexClassLoaderHookHelper.patchClassLoader(getClassLoader(), apkFile,
 ProviderHelper.installProviders(base, getFileStreamPath("testcontentprovider-debug.apk")));
 } catch (Exception e) {
 throw new RuntimeException("hook failed", e);
 }
 }
}

```

## 代理分发以及协议解析

把插件中的ContentProvider安装到插件系统中之后，在插件内部就可以自由使用这些ContentProvider了；要把这些插件共享给整个系统，我们还需要一个货真价实的ContentProvider组件来执行分发：

```

<provider
 android:name="com.example.weishu.contentprovider_management.StubContentProvider"
 android:authorities="com.example.weishu.contentprovider_management.StubContentProvider"
 android:process=":p"
 android:exported="true" />

```

第三方App如果要查询到插件的ContentProvider，必须遵循一个「插件查询协议」，这样StubContentProvider才能把对于插件的请求分发到正确的插件组件：

```
/*
 * 为了使得插件的ContentProvder提供给外部使用，我们需要一个StubProvider做中转；
 * 如果外部程序需要使用插件系统中插件的ContentProvider，不能直接查询原来的那个uri
 * 我们对uri做一些手脚，使得插件系统能识别这个uri；
 *
 * 这里的处理方式如下：
 *
 * 原始查询插件的URI应该为：
 * content://plugin_auth/path/query
 *
 * 如果需要查询插件，需要修改为：
 *
 * content://stub_auth/plugin_auth/path/query
 *
 * 也就是，我们把插件ContentProvider的信息放在URI的path中保存起来；
 * 然后在StubProvider中做分发。
 *
 * 当然，也可以使用QueryParamerter，比如：
 * content://plugin_auth/path/query/ -> content://stub_auth/path/query?plugin=
 * @param raw 外部查询我们使用的URI
 * @return 插件真正的URI
 */
private Uri getRealUri(Uri raw) {
 String rawAuth = raw.getAuthority();
 if (!AUTHORITY.equals(rawAuth)) {
 Log.w(TAG, "rawAuth:" + rawAuth);
 }

 String uriString = raw.toString();
 uriString = uriString.replaceAll(rawAuth + '/', "");
 Uri newUri = Uri.parse(uriString);
 Log.i(TAG, "realUri:" + newUri);
 return newUri;
}
```

通过以上过程我们就实现了ContentProvider的插件化。需要说明的是，DroidPlugind的插件化与上述介绍的方案有一些不同之处：

- 首先DroidPlugin并没有选择预先安装的方案，而是选择Hook ActivityManagerNative，拦截它的getContentProvider以及publishContentProvider方法实现对于插件组件的控制；从这里可以看出它对ContentProvider与Service的插件化几乎是相同的，Hook才是DroidPlugin Style ^\_^.
- 然后，关于携带插件信息，或者说「插件查询协议」方面；DroidPlugin把插件信息放

在查询参数里面，本文呢则是路径参数；这一点完全看个人喜好。

## 小结

本文我们通过「代理分发技术」以及「插件查询协议」完成了ContentProvider组件的插件化，并且给出了对「插件共享组件」的问题的一般解决方案。值得一提的是，系统的ContentProvider其实是lazy load的，也就是说只有在需要使用的时候才会启动对应的ContentProvider，而我们对于插件的实现则是预先加载，这里还有改进的空间，读者可以思考一下解决方案。

由于ContentProvider的使用频度非常低，而很多它使用的场景（比如系统）并不太需要「插件化」，因此在实际的插件方案中，提供ContentProvider插件化的方案非常之少；就算需要实现ContentProvider的插件化，也只是解决插件内部之间共享组件的问题，并没有把插件组件暴露给整个系统。我个人觉得，如果只是希望插件化，那么是否支持ContentProvider无伤大雅，但是，如果希望实现虚拟化或者说容器技术，所有组件是必须支持插件化的。

至此，对于Android系统的四大组件的插件化已经全部介绍完毕；由于是最后一个要介绍的组件，我并没有像之前一样先给出组件的运行原理，然后一通分析最后给出插件方案，而是一边分析代码一边给出自己的思路，把思考——推翻——改进的整个过程完全展现了出来，Android的插件化已经到达了百花齐放的阶段，插件化之路也不只有一条，但是万变不离其宗，希望我的分析和思考对各位读者理解甚至创造插件化方案带来帮助。接下来我会介绍「插件通信机制」，它与本文的ContentProvider以及我反复强调过的一些特性密切相关，敬请期待！

喜欢就点个赞吧，兜里有一块钱的童鞋可以点击下面的打赏然后扫一下二维码哦～持续更新，请关注github项目 [understand-plugin-framework](#) 和我的 [博客](#)！另外很抱歉一个多月没有更新博客了，每天看到各位的来访记录深感惭愧，实在是业务繁忙，身不由己！不出意外接下来会以正常速度更新内容，谢谢支持 ^\_^

# 入门

# Android动态加载dex技术初探

来源:[blog.csdn.net](http://blog.csdn.net)

今天不忙，研究了下Android动态加载dex的技术，主要参考：

- 1、<http://www.cnblogs.com/over140/archive/2011/11/23/2259367.html>
- 2、<http://www.fengyoutian.com/web/single/13>

好歹算是跑通了。下面把实现过程与遇到的问题归纳下，方便后续查找使用。

## 一、综述

Android使用Dalvik虚拟机加载可执行程序，所以不能直接加载基于class的jar，而是需要将class转化为dex字节码，从而执行代码。优化后的字节码文件可以存在一个.jar中，只要其内部存放的是.dex即可使用。

将class的jar包转化为dex需要用到命令dx（在 \*\\android-sdk\\build-tools\\version[23.0.1] 或 \*\\android-sdk\\platform-tools 下能找到）；命令使用方式为： dx --dex --output=output.jar origin.jar，该命令将包含class的origin.jar转化为包含dex的output.jar文件。

Android支持动态加载的两种方式是： DexClassLoader和PathClassLoader， DexClassLoader可加载jar/apk/dex，且支持从SD卡加载； PathClassLoader据说只能加载已经安装在Android系统内APK文件，此说法我尚未验证  
(<http://blog.csdn.net/quaful/article/details/6096951>)；以下这一段是摘录：

/\*\*/

PathClassLoader 的限制要更多一些，它只能加载已经安装到 Android 系统中的 apk 文件，也就是 /data/app 目录下的 apk 文件。其它位置的文件加载的时候都会出现 ClassNotFoundException. 例如：

```
PathClassLoader cl = new PathClassLoader(jarFile.toString(),
 "/data/app/", ClassLoader.getSystemClassLoader());
```

为什么有这个限制呢？我认为这其实是当前 Android 的一个 bug，因为 PathClassLoader 会去读取 /data/dalvik-cache 目录下的经过 Dalvik 优化过的 dex 文件，这个目录的 dex 文件是在安装 apk 包的时候由 Dalvik 生成的。例如，如果包的名字是 com.qihoo360.test，Android 应用安装之后都保存在 /data/app 目录下，即 /data/app/com.qihoo360.test-1.apk，那么 /data/dalvik-cache 目录下就会生成 data@app@com.qihoo360.test-1.apk@classes.dex 文件。在调用 PathClassLoader 时，它就会按照这个规则去找 dex 文件，如果你指定的 apk 文件是 /sdcard/test.apk，它按照这个规则就会去读 /data/dalvik-cache/sdcard@test.apk@classes.dex 文件，显然这个文件不会存在，所以 PathClassLoader 会报错。

```
/***/
```

## 二、实验步骤

新建一个Android工程，并进行如下操作：

- 2.1 定义一个接口IShowToast.java

```
package com.example.testdextoast;

import android.content.Context;

public interface IShowToast {
 public int showToast(Context context);
}
```

- 2.2 再定义一个简单实现ShowToastImpl.java

```
package com.example.testdextoast;

import android.content.Context;
import android.widget.Toast;

public class ShowToastImpl implements IShowToast {

 @Override
 public int showToast(Context context) {
 Toast.makeText(context, "我来自另一个dex文件", Toast.LENGTH_LONG).show();
 return 100;
 }
}
```

- 2.3 将且仅将ShowToastImpl.java导出为jar（Eclipse工程右键->Export->Java->Jar file），这时候我们得到一个包含class文件的jar，命名为origin.jar。（此处按文章1说法同时导出接口IShowToast.java会报错：java.lang.IllegalAccessError: Class ref in pre-verified class resolved to unexpected implementation，但是我测试了导出接口IShowToast.java的情况，可以正常使用，没有崩溃问题。）
- 2.4 在命令行下，进入dx所在目录，将jar文件拷到该目录下，执行dx --dex --output=output.jar origin.jar，得到内含class.dex的output.jar
- 2.5 将output.jar文件放到SD卡下adb push output.jar sdcard/output.jar（或者将jar内的dex抽出来放到SD卡亦可）
- 2.6 开始编写调用代码：新建调用的Android工程，将IShowToast.java放入该工程，注意：**此处一定不能改变此文件的包路径，否则dex中的实现类找不到接口，会报错：**

```
[plain] view plain copy print?java.lang.ClassNotFoundException: Didn't find class "com.example.testdextoast.ShowToastImpl" while trying to load class "com.example.testdextoast.ShowToastImpl"
...
Suppressed: java.lang.NoClassDefFoundError: Failed resolution of: Lcom/example/testdextoast/ShowToastImpl;
 ... 16 more
Caused by: java.lang.ClassNotFoundException: Didn't find class "com.example.testdextoast.ShowToastImpl" while trying to load class "com.example.testdextoast.ShowToastImpl"
 ... 21 more
Suppressed: java.lang.ClassNotFoundException: com.example.testdextoast.ShowToastImpl;
 ... 22 more
Suppressed: java.lang.ClassNotFoundException: com.example.testdextoast.ShowToastImpl;
 ... 23 more
Caused by: java.lang.NoClassDefFoundError: Class not found using the boot class loader.
Suppressed: java.lang.ClassNotFoundException: Didn't find class "com.example.testdextoast.ShowToastImpl" while trying to load class "com.example.testdextoast.ShowToastImpl"
 ... 15 more
Suppressed: java.lang.ClassNotFoundException: com.example.testdextoast.ShowToastImpl;
 ... 16 more
Caused by: java.lang.NoClassDefFoundError: Class not found using the boot class loader.
```

测试工程代码如下：

```

package com.example.testshowtoastdex;

import java.io.File;

import com.example.testdextoast.IShowToast;

import dalvik.system.DexClassLoader;
import android.app.Activity;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;

public class MainActivity extends Activity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);

 File dexOutputDir = getDir("dex1", 0);
 String dexPath = Environment.getExternalStorageDirectory().toString() + File.separator;
 DexClassLoader loader = new DexClassLoader(dexPath,
 dexOutputDir.getAbsolutePath(),
 null, getClassLoader());

 try {
 Class cls = loader.loadClass("com.example.testdextoast.ShowToastImpl");
 IShowToast impl = (IShowToast) cls.newInstance();
 impl.showToast(this);
 } catch (Exception e) {
 Log.d("TEST111", "error happened", e);
 }
 }
}

```

此处需要注意DexClassLoader的四个参数：

**参数1 dexPath:** 待加载的dex文件路径，如果是外存路径，一定要加上读外存文件的权限（<uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE"/>），否则会报与上面一样的错误，这点参考文章2中说这个权限可有可无是错误的。(更正下：

**Android4.4 KitKat及以后的版本需要此权限，之前的版本不需要权限)**

**参数2 optimizedDirectory:** 解压后的dex存放位置，此位置一定要是可读写且仅该应用可读写（安全性考虑），所以只能放在data/data下。本文 getDir("dex1", 0) 会在 /data/data/\*\*package/ 下创建一个名叫"app\_dex1"的文件夹，其内存放的文件是自动生成output.dex；如果不满足条件，Android会报的错误为：

```
java.lang.IllegalArgumentException:
 optimizedDirectory not readable/writable: /storage/sdcard0

java.lang.IllegalArgumentException: Optimized data directory /storage/sdcard0
 is not owned by the current user.
 Shared storage cannot protect your application from code injection attacks.
```

**参数3 libraryPath:** 指向包含本地库(so)的文件夹路径，可以设为null

**参数4 parent:** 父级类加载器，一般可以通过 Context.getClassLoader 获取到，也可以通过 classLoader.getSystemClassLoader() 取到。

## 三、执行结果

上述步骤完成后就能调用到ShowToastImpl中方法并展示一个Toast，动态加载dex的皮毛已经了解了。另外文章1谈到”在实践中发现，自己新建一个Java工程然后导出jar是无法使用的“，这点经测试是不正确了，在Java工程中导出的class也可以被转化为dex并加载。导出jar的操作步骤见上文。

```
package com.example.testdextoast;

public interface IShowToast {
 public String getToast();
}

public class ShowToastImpl implements IShowToast {

 @Override
 public String getToast() {
 return "我来自另一个dex文件";
 }
}
```

```
package com.example.testshowtoastdex;

[java] view plain copy print?import java.io.File;
import com.example.testdextoast.IShowToast;
import dalvik.system.DexClassLoader;
import android.app.Activity;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.widget.Toast;

public class MainActivity extends Activity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);

 File dexOutputDir = getDir("dex1", 0);
 String dexPath = Environment.getExternalStorageDirectory().toString() + File.separator;
 DexClassLoader loader = new DexClassLoader(dexPath,
 dexOutputDir.getAbsolutePath(),
 null, getClassLoader());
 try {
 Class cls = loader.loadClass("com.example.testdextoast.ShowToastImpl");
 IShowToast impl = (IShowToast) cls.newInstance();
 Toast.makeText(this, "来自Java包1: " + impl.getToast(), Toast.LENGTH_SHORT)
 } catch (Exception e) {
 Log.d("TEST111", "error happened", e);
 }
 }
}
```

这段代码弹出的Toast是：“来自Java包1：我来自另一个dex文件”，说明java工程也是可以导出class转为dex的。我估计原文意思是java工程生成的可执行jar文件不能使用。

## 四、补充

### 1、如何在不安装APK的情况下调用其内部的Activity？

<http://blog.zhourunsheng.com/2011/09/%E6%8E%A2%E7%A7%98%E8%85%BE%E8%AE%AFandroid%E6%89%8B%E6%9C%BA%E6%B8%B8%E6%88%8F%E5%B9%B3%E5%8F%B0%E4%B9%8B%E4%B8%8D%E5%AE%89%E8%A3%85%E6%B8%B8%E6%88%8Fapk%E7%9B%B4%E6%8E%A5%E5%90%AF%E5%8A%A8%E6%B3%95/>

核心思想是反射目标APK中的Activity，构造完成后将当前应用的Context传入到目标APK中使其具有上下文环境。

# Android动态加载技术三个关键问题详解

来源:[infoQ](#)

动态加载技术（也叫插件化技术）在技术驱动型的公司中扮演着相当重要的角色，当项目越来越庞大的时候，需要通过插件化来减轻应用的内存和CPU占用，还可以实现热插拔，即在不发布新版本的情况下更新某些模块。动态加载是一项很复杂的技术，这里主要介绍动态加载技术中的三个基础性问题，至于完整的动态加载技术的实现请参考笔者发起的[开源插件化框架DL](#)。项目期间有多位开发人员一起贡献代码。

不同的插件化方案各有各的特色，但是它们都必须要解决三个基础性问题：资源访问、Activity生命周期的管理和ClassLoader的管理。在介绍它们之前，首先要明白宿主和插件的概念，宿主是指普通的apk，而插件一般是指经过处理的dex或者apk，在主流的插件化框架中多采用经过特殊处理的apk来作为插件，处理方式往往和编译以及打包环节有关，另外很多插件化框架都需要用到代理Activity的概念，插件Activity的启动大多数是借助一个代理Activity来实现的。

## 1. 资源访问

我们知道，宿主程序调起未安装的插件apk，一个很大的问题就是资源如何访问，具体来说就是插件中凡是以R开头的资源都不能访问了。这是因为宿主程序中并没有插件的资源，所以通过R来加载插件的资源是行不通的，程序会抛出异常：无法找到某某id所对应的资源。

针对这个问题，有人提出了将插件中的资源在宿主程序中也预置一份，这虽然能解决问题，但是这样就会产生一些弊端。首先，这样就需要宿主和插件同时持有一份相同的资源，增加了宿主apk的大小；其次，在这种模式下，每次发布一个插件都需要将资源复制到宿主程序中，这意味着每发布一个插件都要更新一下宿主程序，这就和插件化的思想相违背了。

因为插件化的目的就是要减小宿主程序apk包的大小，同时降低宿主程序的更新频率并做到自由装载模块，所以这种方法不可取，它限制了插件的线上更新这一重要特性。还有人提供了另一种方式，首先将插件中的资源解压出来，然后通过文件流去读取资源，这样做理论上是可行的，但是实际操作起来还是有很大难度的。首先不同资源有不同的文件流格式，比如图片、XML等，其次针对不同设备加载的资源可能是不一样的，如何选择合适的资源也是一个需要解决的问题，基于这两点，这种方法也不建议使用，因为它实现起来有较大难度。为了方便地对插件进行资源管理，下面给出一种合理的方式。

我们知道，Activity的工作主要是通过ContextImpl来完成的，Activity中有一个叫mBase的成员变量，它的类型就是ContextImpl。注意到Context中有如下两个抽象方法，看起来是和资源有关的，实际上Context就是通过它们来获取资源的。这两个抽象方法的真正实现是在ContextImpl中，也就是说，只要实现这两个方法，就可以解决资源问题了。

```
/** Return an AssetManager instance for your application's package. */

public abstract AssetManager getAssets();

/** Return a Resources instance for your application's package. */

public abstract Resources getResources();
```

下面给出具体的实现方式，首先要加载apk中的资源，如下所示。

```
protected void loadResources() {
 try {
 AssetManager assetManager = AssetManager.class.newInstance();
 Method addAssetPath = assetManager.getClass().getMethod("addAssetPath", String.class);
 addAssetPath.invoke(assetManager, mDexPath);
 mAssetManager = assetManager;
 } catch (Exception e) {
 e.printStackTrace();
 }
 Resources superRes = super.getResources();
 mResources = new Resources(mAssetManager, superRes.getDisplayMetrics(),
 superRes.getConfiguration());
 mTheme = mResources.newTheme();
 mTheme.setTo(super.getTheme());
}
```

从loadResources()的实现可以看出，加载资源的方法是通过反射，通过调用AssetManager中的addAssetPath方法，我们可以将一个apk中的资源加载到Resources对象中，由于addAssetPath是隐藏API我们无法直接调用，所以只能通过反射。下面是它的声明，通过注释我们可以看出，传递的路径可以是zip文件也可以是一个资源目录，而apk就是一个zip，所以直接将apk的路径传给它，资源就加载到AssetManager中了。然后再通过AssetManager来创建一个新的Resources对象，通过这个对象我们就可以访问插件apk中的资源了，这样一来问题就解决了。

```

/**
 * Add an additional set of assets to the asset manager. This can be
 * either a directory or ZIP file. Not for use by applications. Returns
 * the cookie of the added asset, or 0 on failure.
 * {@hide}
 */
public final int addAssetPath(String path) {
 synchronized (this) {
 int res = addAssetPathNative(path);
 makeStringBlocks(mStringBlocks);
 return res;
 }
}

```

接着在代理Activity中实现getAssets()和getResources(), 如下所示。关于代理Activity的含义请参看DL开源插件化框架的实现细节，这里不再详细描述了。

```

@Override
public AssetManager getAssets() {
 return mAssetManager == null ? super.getAssets() : mAssetManager;
}

@Override
public Resources getResources() {
 return mResources == null ? super.getResources() : mResources;
}

```

通过上述这两个步骤，就可以通过R来访问插件中的资源了。

## 2. Activity生命周期的管理

管理Activity生命周期的方式各种各样，这里只介绍两种：反射方式和接口方式。反射的方式很好理解，首先通过Java的反射去获取Activity的各种生命周期方法，比如onCreate、onStart、onResume等，然后在代理Activity中去调用插件Activity对应的生命周期方法即可，如下所示：

```
@Override

protected void onResume() {

 super.onResume();

 Method onResume = mActivityLifecycleMethods.get("onResume");

 if (onResume != null) {

 try {

 onResume.invoke(mRemoteActivity, new Object[] { });

 } catch (Exception e) {

 e.printStackTrace();

 }

 }

}

@Override

protected void onPause() {

 Method onPause = mActivityLifecycleMethods.get("onPause");

 if (onPause != null) {

 try {

 onPause.invoke(mRemoteActivity, new Object[] { });

 } catch (Exception e) {

 e.printStackTrace();

 }

 }

 super.onPause();

}
```

使用反射来管理插件Activity的生命周期是有缺点的，一方面是反射代码写起来比较复杂，另一方面是过多使用反射会有一定的性能开销。下面介绍接口方式，接口方式很好地解决了反射方式的不足之处，这种方式将Activity的生命周期方法提取出来作为一个接口（比如叫DLPlugin），然后通过代理Activity去调用插件Activity的生命周期方法，这样就完成了插件Activity的生命周期管理，并且没有采用反射，这就解决了性能问题。同时接口的声明也比较简单，下面是DLPlugin的声明：

```
public interface DLPlugin {

 public void onStart();

 public void onRestart();

 public void onActivityResult(int requestCode, int resultCode, Intent
data);

 public void onResume();

 public void onPause();

 public void onStop();

 public void onDestroy();

 public void onCreate(Bundle savedInstanceState);

 public void setProxy(Activity proxyActivity, String dexPath);

 public void onSaveInstanceState(Bundle outState);

 public void onNewIntent(Intent intent);

 public void onRestoreInstanceState(Bundle savedInstanceState);

 public boolean onTouchEvent(MotionEvent event);

 public boolean onKeyDown(int keyCode, KeyEvent event);

 public void onWindowAttributesChanged(LayoutParams params);

 public void onWindowFocusChanged(boolean hasFocus);

 public void onBackPressed();

 ...
}
```

在代理Activity中只需要按如下方式即可调用插件Activity的生命周期方法，这就完成了插件Activity的生命周期的管理。

```
@Override

protected void onStart() {

 mRemoteActivity.onStart();

 super.onStart();

}

@Override

protected void onRestart() {

 mRemoteActivity.onRestart();

 super.onRestart();

}

@Override

protected void onResume() {

 mRemoteActivity.onResume();

 super.onResume();

}
```

通过上述代码应该不难理解接口方式对插件Activity生命周期的管理思想，其中mRemoteActivity就是DLPlugin的实现。

### 3. 插件ClassLoader的管理

为了更好地对多插件进行支持，需要合理地去管理各个插件的DexClassLoader，这样同一个插件就可以采用同一个ClassLoader去加载类，从而避免了多个ClassLoader加载同一个类时所引发的类型转换错误。在下面的代码中，通过将不同插件的ClassLoader存储在一个HashMap中，这样就可以保证不同插件中的类彼此互不干扰。

```
public class DLClassLoader extends DexClassLoader {

 private static final String TAG = "DLClassLoader";

 private static final HashMap<String, DLClassLoader> mPluginClassLoaders
 = new HashMap<String, DLClassLoader>();

 protected DLClassLoader(String dexPath, String optimizedDirectory, String libraryPath,
 ClassLoader parent) {
 super(dexPath, optimizedDirectory, libraryPath, parent);
 }

 /**
 * return a available classloader which belongs to different apk
 */
 public static DLClassLoader getClassLoader(String dexPath, Context context,
 ClassLoader parentLoader) {
 DLClassLoader dLClassLoader = mPluginClassLoaders.get(dexPath);
 if (dLClassLoader != null)
 return dLClassLoader;

 File dexOutputDir = context.getDir("dex", Context.MODE_PRIVATE);
 final String dexOutputPath = dexOutputDir.getAbsolutePath();
 dLClassLoader = new DLClassLoader(dexPath, dexOutputPath, null,
 parentLoader);
 mPluginClassLoaders.put(dexPath, dLClassLoader);

 return dLClassLoader;
 }
}
```

事实上插件化的技术细节非常多，这绝非一个章节的内容所能描述清楚的，另外插件化作为一种核心技术，需要开发者有较深的开发功底才能够很好地理解，因此本节的内容更多是让读者对插件化开发有一个感性的了解，细节上还需要读者自己去钻研，也可以通过DL插件化框架去深入地学习。

# Android插件化基础

来源:[Android插件化基础](#)

说到Android插件化，简单来说就是如下的操作：

- 开发者将插件代码封装成Jar或者APK。
- 宿主下载或者从本地加载Jar或者APK到宿主中。
- 将宿主调用插件中的算法或者Android特定的Class（如Activity）

第一步的封装Jar和APK。就不用多说了。下面说说第二步，将Jar或者Apk加载到宿主当中来，使用ClassLoader，可以加载文件系统上的jar、dex、apk。从而将Class加载到宿主之中。再通过反射进行调用。下面是个简单的例子：

[Demo地址](#)

首先定义一个MyClassLoader继承ClassLoader：

```
public class MyClassLoader extends ClassLoader{
 private String mDirPath;
 public MyClassLoader(String dirPath) {
 //这里获取文件路径。后面getClassData的时候和class名字一起组成class的文件路径。
 mDirPath = dirPath;
 }
 @Override
 protected Class<?> findClass(String arg0) throws ClassNotFoundException {
 //读取*.class文件内容，获得该class文件的字节码
 byte[] classData = getClassData(arg0);
 if (classData == null) {
 throw new ClassNotFoundException();
 }
 //将class的字节码转换成Class类的实例
 return defineClass(arg0, classData, 0, classData.length);
 }
}
```

上面通过重载findClass方法，我们读取class文件的字节码然后通过调用ClassLoader的defineClass方法来定义Class。具体使用MyClassLoader的时候我们将文件地址以及类名传给MyClassLoader：

```
MyClassLoader myClassLoader = new MyClassLoader(dirPath);
Class loadedClass = myClassLoader.loadClass("MainTest");
```

这样就得到了Class。后面我们通过反射调用他的相应方法就实现了最简单的插件化。

```
Method method = loadedClass.getMethod("sayHello", null);
method.invoke(object, null);
```

上面使用了自定义的ClassLoader来实现的加载，功能比较简单，只支持class的加载。

另外也可以通过使用系统的DexClassLoader来直接加载Jar,Apk。

```
File dexOutputDir = this.getDir("dex", 0);
final String dexOutputPath = dexOutputDir.getAbsolutePath();
ClassLoader localClassLoader = ClassLoader.getSystemClassLoader();
DexClassLoader dexClassLoader = new DexClassLoader([Jar或者Apk的路径], dexOutputPath, null, localClassLoader);
try {
 Class<?> localClass = dexClassLoader.loadClass(className);
 Constructor<?> localConstructor = localClass.getConstructor(new Class[] {});
 Object instance = localConstructor.newInstance();
} catch (Exception e) {
 e.printStackTrace();
}
```

怎么样？通过这样的操作，就可以做一个简单的代码逻辑的插件化了。将你的逻辑处理代码打包成Jar或者APK，放到服务器上。客户端开启后检查是否有新的版本，如果有就后台下载，覆盖源文件（这样逻辑上应该先把旧的逻辑加载到内存中，这里可以有很多不同的策略）。

下次打开的时候，直接加载新下载的Jar或者APK就动态更新了内部逻辑。

上面只是简单的逻辑代码的插件化，Android资源以及其他具有特性的Class的插件化就复杂很多，处理方法也多种多样。

后面通过对DynamicLoadApk,DirectLoadApk以及ACDD进行分析，由简入深的体验下Android插件化的各种实现方式。



# 插件化开发—动态加载技术加载已安装和未安装的apk

来源:[http://blog.csdn.net/u010687392/article/details/47121729?  
hmsr=toutiao.io&utm\\_medium=toutiao.io&utm\\_source=toutiao.io](http://blog.csdn.net/u010687392/article/details/47121729?hmsr=toutiao.io&utm_medium=toutiao.io&utm_source=toutiao.io)

## [TOC]

首先引入一个概念，动态加载技术是什么？为什么要引入动态加载？它有什么好处呢？首先要明白这几个问题，我们先从应用程序入手，大家都知道在Android App中，一个应用程序dex文件的方法数最大不能超过65536个，否则，你的app将出异常了，那么如果越大的项目那肯定超过了，像美团、支付宝等都是使用动态加载技术，支付宝在去年的一个技术分享类会议上就推崇让应用程序插件化，而美团也公布了他们的解决方案：Dex自动拆包和动态加载技术。所以使用动态加载技术解决此类问题。而它的优点可以让应用程序实现插件化、插拔式结构，对后期维护作用那不用说了。

## 1、什么是动态加载技术？

动态加载技术就是使用类加载器加载相应的apk、dex、jar（必须含有dex文件），再通过反射获得该apk、dex、jar内部的资源（class、图片、color等等）进而供宿主app使用。

## 2、关于动态加载使用的类加载器

使用动态加载技术时，一般需要用到这两个类加载器：

- PathClassLoader - 只能加载已经安装的apk，即/data/app目录下的apk。
- DexClassLoader - 能加载手机中未安装的apk、jar、dex，只要能在找到对应的路径。

这两个加载器分别对应使用的场景各不同，所以接下来，分别讲解它们各自加载相同的插件apk的使用。

### 3、使用PathClassLoader加载已安装的apk插件，获取相应的资源供宿主app使用

下面通过一个demo来介绍PathClassLoader的使用：

- 1、首先我们需要知道一个manifest中的属性：SharedUserId。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 package="com.example.apkplugindemo"
 android:sharedUserId="com.sunzxyong.myapp">
```

该属性是用来干嘛的呢？简单的说，应用从一开始安装在Android系统上时，系统都会给它分配一个*linux user id*，之后该应用在今后都将运行在独立的一个进程中，其它应用程序不能访问它的资源，那么如果两个应用的sharedUserId相同，那么它们将共同运行在相同的linux进程中，从而便可以数据共享、资源访问了。所以我们在宿主app和插件app的manifest上都定义一个相同的sharedUserId。

- 2、那么我们将插件apk安装在手机上后，宿主app怎么知道手机内该插件是否是我们应用程序的插件呢？

我们之前是不是定义过插件apk也是使用相同的sharedUserId，那么，我就可以这样思考了，是不是可以得到手机内所有已安装apk的sharedUserId呢，然后通过判断sharedUserId是否和宿主app的相同，如果是，那么该app就是我们的插件app了。确实是这样的思路的，那么有了思路最大的问题就是怎么获取一个应用程序内的sharedUserId了，我们可以通过PackageManager.sharedUserId来获取，请看代码：

```

/**
 * 查找手机内所有的插件
 * @return 返回一个插件List
 */
private List<PluginBean> findAllPlugin() {
 List<PluginBean> plugins = new ArrayList<>();
 PackageManager pm = getPackageManager();
 //通过包管理器查找所有已安装的apk文件
 List<PackageInfo> packageInfos = pm.getInstalledPackages(
 PackageManager.GET_UNINSTALLED_PACKAGES);
 for (PackageInfo info : packageInfos) {
 //得到当前apk的包名
 String pkgName = info.packageName;
 //得到当前apk的sharedUserId
 String shareUesrId = info.sharedUserId;
 //判断这个apk是否是我们应用程序的插件
 if (shareUesrId != null && shareUesrId.equals("com.sunzxyong.myapp")
 && !pkgName.equals(this.getPackageName())) {
 //得到插件apk的名称
 String label = pm.getApplicationLabel(info.applicationInfo).toString();
 PluginBean bean = new PluginBean(label, pkgName);
 plugins.add(bean);
 }
 }
 return plugins;
}

```

通过这段代码，我们就可以轻松的获取手机内存在的所有插件，其中PluginBean是定义的一个实体类而已，就不贴它的代码了。

- 3、如果找到了插件，就把可用的插件显示出来了，如果没有找到，那么就可提示用户先去下载插件什么的。

```

List<HashMap<String, String>> datas = new ArrayList<>();
List<PluginBean> plugins = findAllPlugin();
if (plugins != null && !plugins.isEmpty()) {
 for (PluginBean bean : plugins) {
 HashMap<String, String> map = new HashMap<>();
 map.put("label", bean.getLabel());
 datas.add(map);
 }
} else {
 Toast.makeText(this, "没有找到插件，请先下载！", Toast.LENGTH_SHORT).show();
}
showEnableAllPluginPopup(datas);

```

- 4、如果找到后，那么我们选择对应的插件时，在宿主app中就加载插件内对应的资源，这个才是PathClassLoader的重点。我们首先看看怎么实现的吧：

```
/**
 * 加载已安装的apk
 * @param packageName 应用的包名
 * @param pluginContext 插件app的上下文
 * @return 对应资源的id
 */
private int dynamicLoadApk(String packageName, Context pluginContext) throws Exception
 //第一个参数为包含dex的apk或者jar的路径，第二个参数为父加载器
 PathClassLoader pathClassLoader = new PathClassLoader(
 pluginContext.getPackageResourcePath(),
 ClassLoader.getSystemClassLoader());
 //通过使用自身的加载器反射出mipmap类进而使用该类的功能
 Class<?> clazz = pathClassLoader.loadClass(packageName + ".R$mipmap");
 //参数：1、类的全名，2、是否初始化类，3、加载时使用的类加载器
 Class<?> clazz = Class.forName(packageName + ".R$mipmap", true, pathClassLoader);
 //使用上述两种方式都可以，这里我们得到R类中的内部类mipmap，通过它得到对应的图片id，进而给我们使用
 Field field = clazz.getDeclaredField("one");
 int resourceId = field.getInt(R.mipmap.class);
 return resourceId;
}
```

这个方法就是加载包名为packageName的插件，然后获得插件内名为one.png的图片的资源id，进而供宿主app使用该图片。现在我们一步一步来讲解一下：

- 首先就是new出一个PathClassLoader对象，它的构造方法为：

```
public PathClassLoader(String dexPath, ClassLoader parent)
```

其中第一个参数是通过插件的上下文来获取插件apk的路径，其实获取到的就是 `data/app/apkthemepugin.apk`，那么插件的上下文怎么获取呢？在宿主app中我们只有本app的上下文啊，答案就是为插件app创建一个上下文：

```
//获取对应插件中的上下文，通过它可得到插件的Resource
Context pluginContext = this.createPackageContext(packageName,
 CONTEXT_IGNORE_SECURITY | CONTEXT_INCLUDE_CODE);
```

通过插件的包名来创建上下文，不过这种方法只适合获取已安装的app上下文。或者不需要通过反射直接通过插件上下文`getResource().getxxx(R..)`也行，而这里用的是反射方法。

第二个参数是父加载器，都是**ClassLoader.getSystemClassLoader()**。

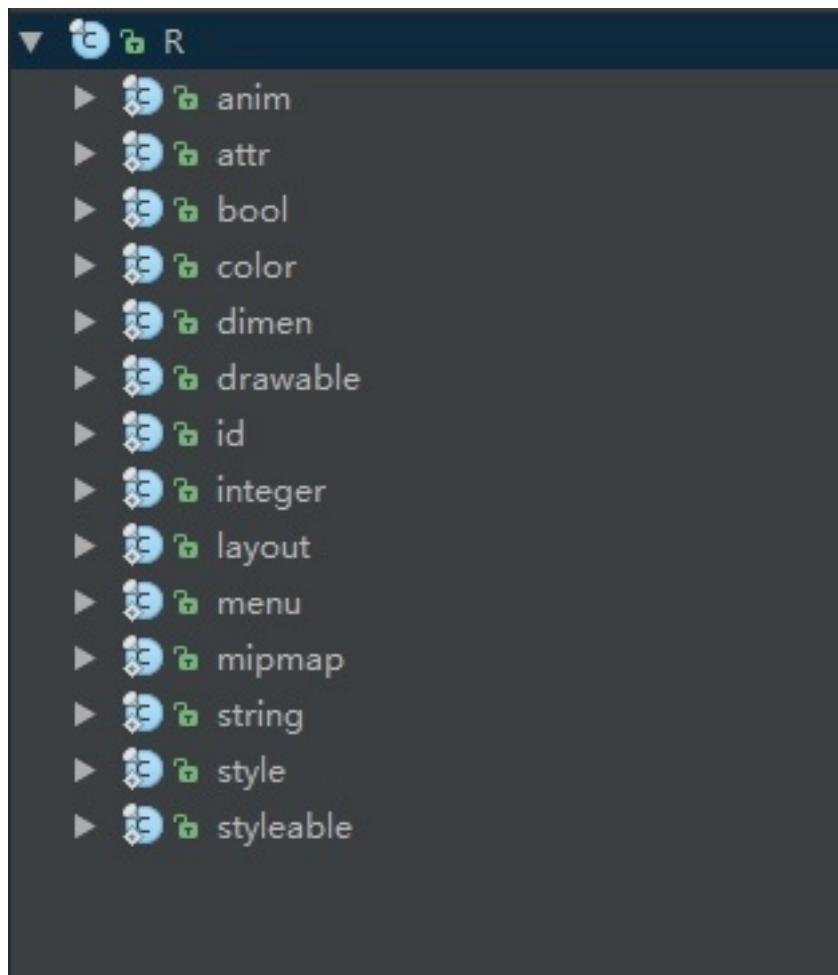
- 好了，插件app的类加载器我们创建出来了，接下来就是通过反射获取对应类的资源了，这里我是获取R类中的内部类mipmap类，然后通过反射得到mipmap类中名为one的字段的值，

```
public static final class mipmap {
 public static final int ic_launcher=0x7f030000;
 public static final int one=0x7f030001;
}
```

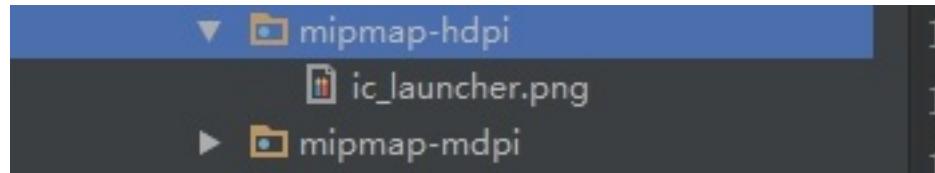
```
plugnContext.getResources().getDrawable(resouceId)
```

就可以获取对应id的Drawable得到该图片资源进而宿主app的可用它设置背景等。当然也可以获取到其它的资源或者获取Activity类等，这里只是做一个示例。

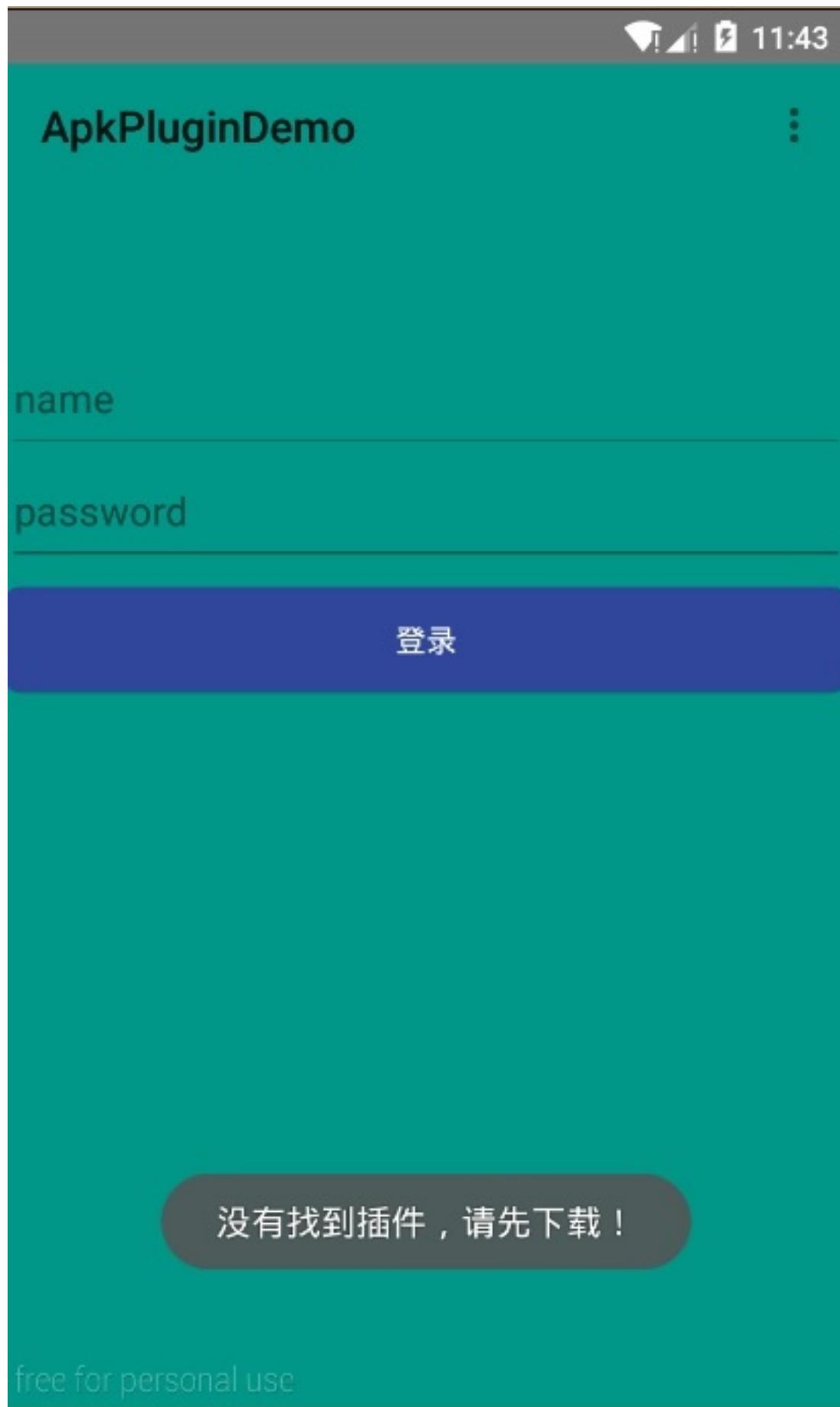
- 备：关于R类，在AS中的目录为：/build/generated/source/r/debug/<- packageName ->。它的内部类有：



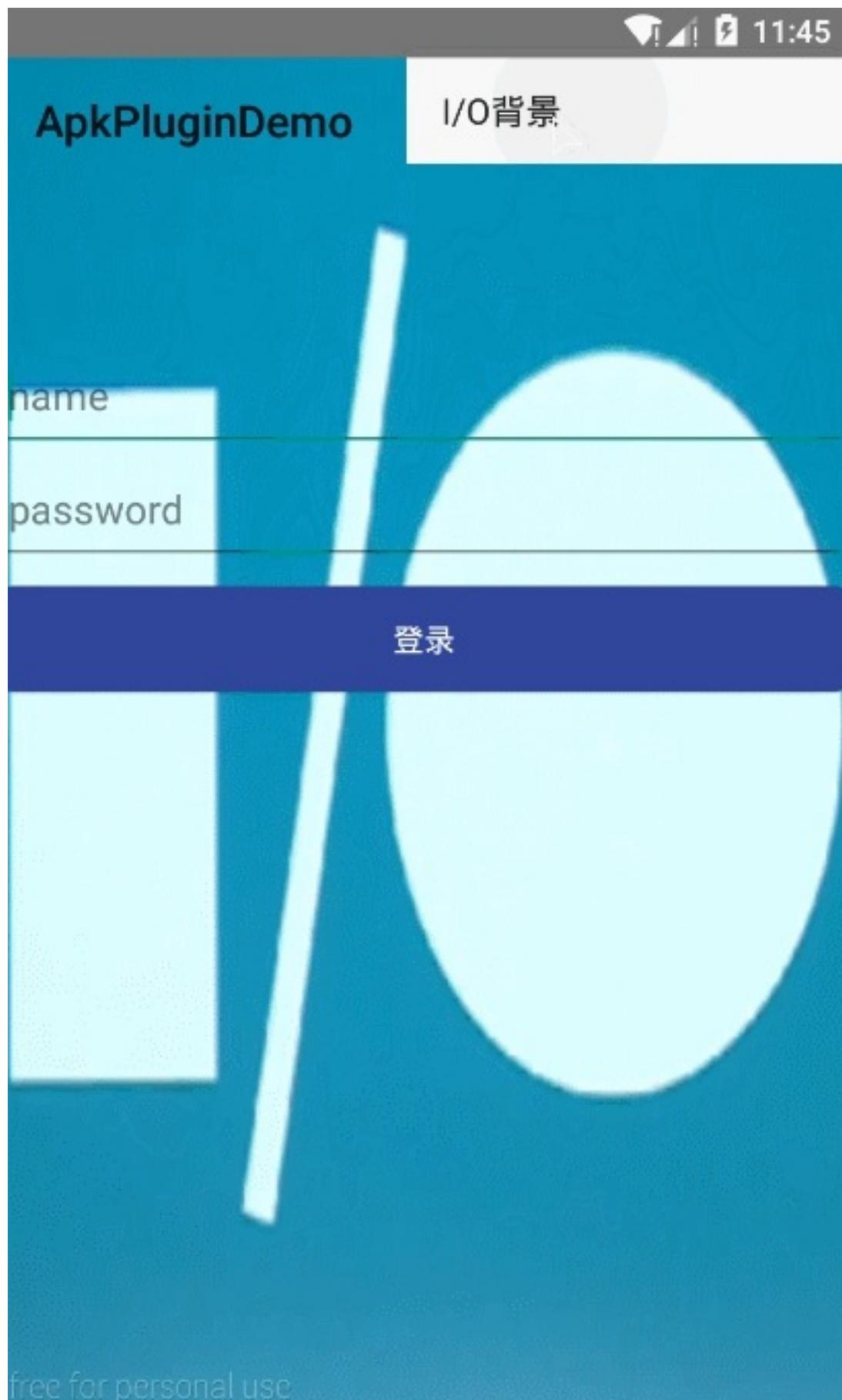
脑洞大的可以尽可能的利用这些资源吧！！！ 下面演示下该demo效果，在没有插件情况下会提示请先下载插件，有插件时候就选择对应的插件而供宿主app使用，本demo是换背景的功能演示，我来看宿主app中mipmap文件夹下并没有one.png这张图片，截图为证：



在没有安装插件情况下：



安装插件后：



可以看到，宿主app使用了插件中的图片资源。

这时，有的人就会想，这个插件需要下载下来还需要安装到手机中去，这不就是又安装了一个apk啊，只是没显示出来而已，这种方式不太友好，那么，可不可以只下载下来，不用安装，也能供宿主app使用呢？像微信上可以运行没有安装的飞机大战这样的，这当然可以的。这就需要用到另外一个加载器DexClassLoader。

## 4、DexClassLoader加载未安装的apk，提供资源供宿主app使用

关于动态加载未安装的apk，我先描述下思路：首先我们得到事先知道我们的插件apk存放 在哪个目录下，然后分别得到插件apk的信息（名称、包名等），然后显示可用的插件，最后动态加载apk获得资源。

按照上面这个思路，我们需要解决几个问题：

- 1、怎么得到未安装的apk的信息
- 2、怎么得到插件的context或者Resource，因为它是未安装的不可能通过 createPackageContext(...);方法来构建出一个context，所以这时只有在Resource上 下功夫。

现在我们就一一来解答这些问题吧：

- 1、得到未安装的apk信息可以通过mPackageManager.getPackageArchiveInfo()方法获得，

```
public PackageInfo getPackageArchiveInfo(String archiveFilePath, int flags)
```

它的参数刚好是传入一个FilePath，然后返回apk文件的PackageInfo信息：

```
/**
 * 获取未安装apk的信息
 * @param context
 * @param archiveFilePath apk文件的path
 * @return
 */
private String[] getUninstallApkInfo(Context context, String archiveFilePath) {
 String[] info = new String[2];
 PackageManager pm = context.getPackageManager();
 PackageInfo pkgInfo = pm.getPackageArchiveInfo(archiveFilePath,
 PackageManager.GET_ACTIVITIES);
 if (pkgInfo != null) {
 ApplicationInfo appInfo = pkgInfo.applicationInfo;
 String versionName = pkgInfo.versionName;//版本号
 Drawable icon = pm.getApplicationIcon(appInfo);//图标
 String appName = pm.getApplicationLabel(appInfo).toString();//app名称
 String pkgName = appInfo.packageName;//包名
 info[0] = appName;
 info[1] = pkgName;
 }
 return info;
}
```

- 2、得到对应未安装apk的Resource对象，我们需要通过反射来获得：

```
/**
 * @param apkName
 * @return 得到对应插件的Resource对象
 */
private Resources getPluginResources(String apkName) {
 try {
 AssetManager assetManager = AssetManager.class.newInstance();
 //反射调用方法addAssetPath(String path)
 Method addAssetPath = assetManager.getClass().getMethod("addAssetPath", String.class);
 //第二个参数是apk的路径:
 // Environment.getExternalStorageDirectory().getPath()
 // +File.separator+"plugin"+File.separator+"apkplugin.apk"
 //将未安装的Apk文件的添加进AssetManager中, 第二个参数为apk文件的路径带apk名
 addAssetPath.invoke(assetManager, apkDir+ File.separator+apkName);
 Resources superRes = this.getResources();
 Resources mResources = new Resources(assetManager,
 superRes.getDisplayMetrics(),
 superRes.getConfiguration());
 return mResources;
 } catch (Exception e) {
 e.printStackTrace();
 }
 return null;
}
```

通过得到AssetManager中的内部的方法addAssetPath, 将未安装的apk路径传入从而添加进assetManager中, 然后通过new Resource把assetManager传入构造方法中, 进而得到未安装apk对应的Resource对象。

好了! 上面两个问题解决了, 那么接下来就是加载未安装的apk获得它的内部资源。

```

/**
 * 加载apk获得内部资源
 * @param apkDir apk目录
 * @param apkName apk名字,带.apk
 * @throws Exception
 */
private void dynamicLoadApk(String apkDir, String apkName, String apkPackageName) throws Exception {
 //在应用安装目录下创建一个名为app_dex文件夹目录,如果已经存在则不创建
 File optimizedDirectoryFile = getDir("dex", Context.MODE_PRIVATE);
 // /data/data/com.example.dynamicloadapk/app_dex
 Log.v("zxy", optimizedDirectoryFile.getPath().toString());
 //参数: 1、包含dex的apk文件或jar文件的路径,
 // 2、apk、jar解压缩生成dex存储的目录,
 // 3、本地library库目录,一般为null, 4、父ClassLoader
 DexClassLoader dexClassLoader = new DexClassLoader(apkDir+File.separator+apkName,
 optimizedDirectoryFile.getPath(), null, ClassLoader.getSystemClassLoader());
 //通过使用apk自己的类加载器,反射出R类中相应的内部类进而获取我们需要的资源id
 Class<?> clazz = dexClassLoader.loadClass(apkPackageName + ".R$ipmap");
 Field field = clazz.getDeclaredField("one");//得到名为one的这张图片字段
 int resId = field.getInt(R.id.class); //得到图片id
 Resources mResources = getPluginResources(apkName); //得到插件apk中的Resource
 if (mResources != null) {
 //通过插件apk中的Resource得到resId对应的资源
 findViewById(R.id.background).setBackgroundDrawable(mResources.getDrawable(resId));
 }
}

```

其中通过new DexClassLoader()来创建未安装apk的类加载器，我们来看看它的参数：

```

public DexClassLoader(String dexPath,
 String optimizedDirectory,
 String libraryPath,
 ClassLoader parent)

```

- dexPath - 就是apk文件的路径
- optimizedDirectory - apk解压缩后的存放dex的目录，值得注意的是，在4.1以后该目录不允许在sd卡上，看官方文档：

A class loader that loads classes from .jar and .apk files containing a classes.dex entry. This can be used to execute code not installed as part of an application. This class loader requires an application-private, writable directory to cache optimized classes. Use Context.getDir(String, int) to create such a directory:

```
File dexOutputDir = context.getDir("dex", 0);
```

Do not cache optimized classes on external storage. External storage does not provide access controls necessary to protect your application from code injection attacks.

，所以我们用getDir()方法在应用内部创建一个dexOutputDir。

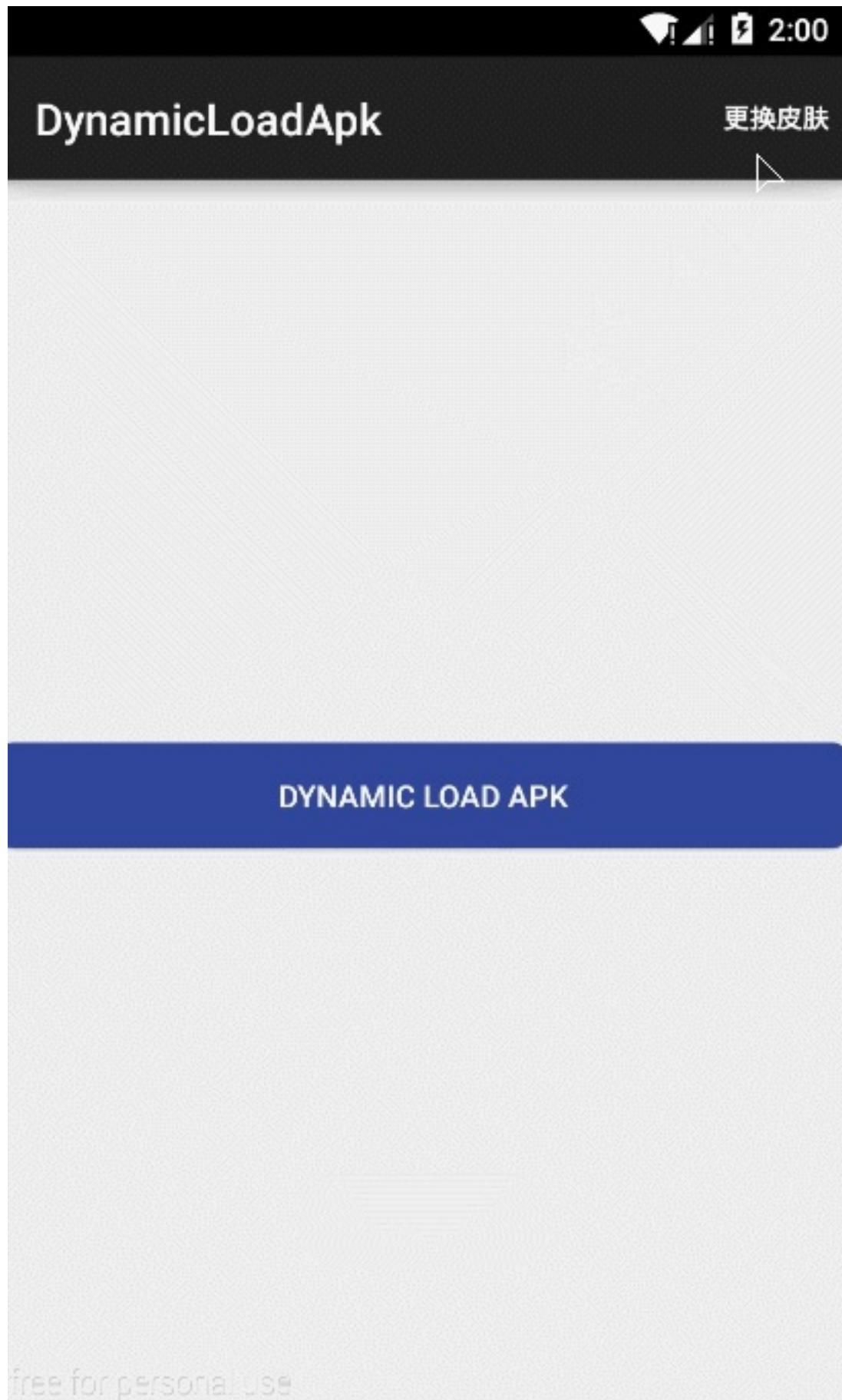
- libraryPath - 本地的library，一般为null
- parent - 父加载器

接下来，就是通过反射的方法，获取出需要的资源。

下面我们来看看demo演示的效果，我是把三个apk插件先放在assets目录下，然后copy到sd上来模仿下载过程，然后加载出相应插件的资源：

先只拷贝一个插件：

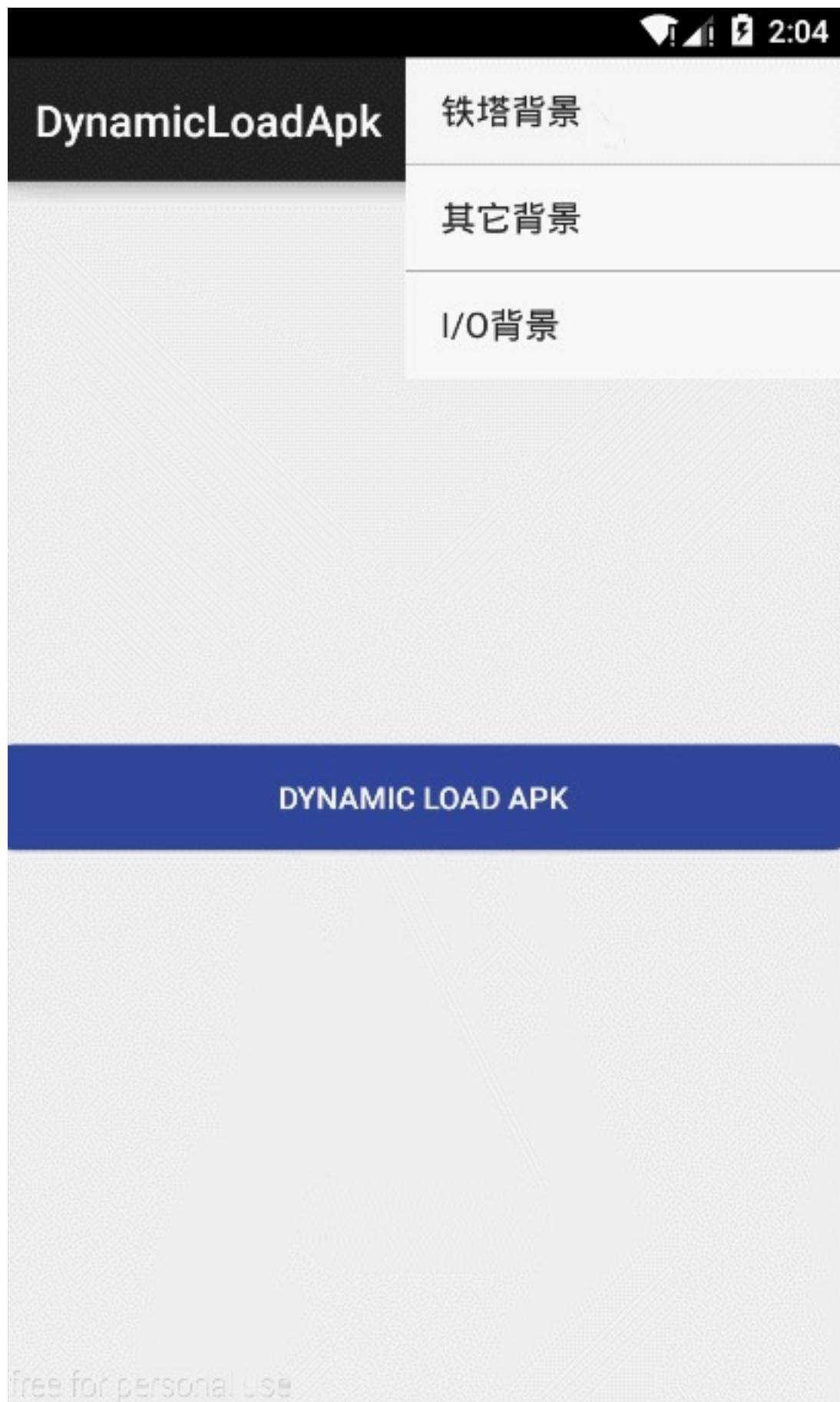
```
copyApkFile("apkthemeplugin-1.apk");
```



可以看到正常的获取到了未安装apk的资源。

再看看拷贝了三个插件：

```
copyApkFile("apkthemeplugin-1.apk");
copyApkFile("apkthemeplugin-2.apk");
copyApkFile("apkthemeplugin-3.apk");
```



可以看到只要一有插件下载，就能显示出来并使用它。

当然插件化开发不只是像只有这种换肤那么简单的用途，这只是个demo，学习这种插件化开发思想的。由此可以联想，这种插件化的开发，是不是像QQ里的表情包啊、背景皮肤啊，通过线上下载线下维护的方式，可以在线下载使用相应的皮肤，不使用时候就可以删了，所以插件化开发是插件与宿主app进行解耦了，即使在没有插件情况下，也不会对宿主app有任何影响，而有的话就供用户选择性使用了。

[戳这下载Demo](#)

# 其他

# Android关于Dex拆分(MultiDex)技术详解

来源:[Android关于Dex拆分\(MultiDex\)技术详解](#)

## 一、前言

关于Android中的分包技术，已经不是什么新的技术了，网上也有很多解析了，但是他们都是给了理论上的知道和原理解析，并没有详细的案例说明，所以这里我们就来详细讲解一下Android中dex拆分技术的解析。在讲解之前，我们还是先来看一下为什么有这个技术的出现？google为什么提供这样的技术。

## 二、背景

在开发应用时，随着业务规模发展到一定程度，不断地加入新功能、添加新的类库，代码在急剧的膨胀，相应的apk包的大小也急剧增加，那么终有一天，你会不幸遇到这个错误：

生成的apk在android 2.3或之前的机器上无法安装，提示：**INSTALL\_FAILED\_DEXOPT**  
方法数量过多，编译时出错，提示：

```
Conversion to Dalvik format failed:Unable to execute dex: method ID not in [0,
0xffff]: 65536
```

无法安装（Android 2.3 **INSTALL\_FAILED\_DEXOPT**）问题，是由dexopt的LinearAlloc限制引起的，在Android版本不同分别经历了4M/5M/8M/16M限制，目前主流4.2.x系统上可能都已到16M，在Gingerbread或者以下系统LinearAllocHdr分配空间只有5M大小的，高于Gingerbread的系统提升到了8M。Dalvik linearAlloc是一个固定大小的缓冲区。在应用的安装过程中，系统会运行一个名为dexopt的程序为该应用在当前机型中运行做准备。dexopt使用LinearAlloc来存储应用的方法信息。Android 2.2和2.3的缓冲区只有5MB，Android 4.x提高到了8MB或16MB。当方法数量过多导致超出缓冲区大小时，会造成dexopt崩溃。

超过最大方法数限制的问题，是由于DEX文件格式限制，一个DEX文件中method个数采用使用原生类型short来索引文件中的方法，也就是4个字节共计最多表达65536个method，field/class的个数也均有此限制。对于DEX文件，则是将工程所需全部class文件

合并且压缩到一个DEX文件期间，也就是Android打包的DEX过程中，单个DEX文件可被引用的方法总数（自己开发的代码以及所引用的Android框架、类库的代码）被限制为65536。

我们知道原因了，但是我们可以看到，google提供了一个方案：Multidex技术来解决这样的问题，为了兼容老版本SDK,但是我们想一下，是不是所有的项目都会用到这个技术呢？答案肯定不是的，这种问题不是所有的项目都会遇到的，只有当你的项目足够庞大，导致方法个数超出限制了，才会使用到这种技术。那么既然要用到，这里就还是要介绍一下，在介绍这篇文章之前，我们先要准备哪些知识点呢？

- 1、了解如何使用Ant脚本编译出一个apk包
- 2、了解编译一个apk包出来的整个流程和步骤
- 3、了解Android中的dx命令和aapt命令的用法
- 4、了解Android中动态加载机制

关于这些资料，我在之前的文章中有说道，不了解的同学可以转战：

Ant脚本编译一个apk包以及apk包打包的整个流程和步骤可以查看这篇文章：

<http://blog.csdn.net/jiangwei0910410003/article/details/50740026>

关于Android中的动态加载机制不了解的同学可以查看这篇文章：

<http://blog.csdn.net/jiangwei0910410003/article/details/48104581>

只有了解了这四个知识点，下面介绍的内容才能看的明白。所以这两篇文章希望能够仔细看一下。

## 三、技术原理

既然准备知识已经做完了，下面我们就来详细介绍一下拆包的技术原理，其实就是google提供个方案：

### 第一步：使用dx进行拆包

这里还有两个两类：一个是自动拆包，一个手动拆包

#### 1、自动拆包

使用google在5.0+的SDK开始，dx命令就已经支持：--multi-dex 参数来直接自动分包

```
C:\Users\i>dx
error: no command specified
usage:
dx --dex [--debug] [--verbose] [--positions=<style>] [--no-locals]
[--no-optimize] [--statistics] [--[no-]optimize-list=<file>] [--no-strict]
[--keep-classes] [--output=<file>] [--dump-to=<file>] [--dump-width=<n>]
[--dump-method=<name>[*]] [--verbose-dump] [--no-files] [--core-library]
[--num-threads=<n>] [--incremental] [--force-iumbol]
[--multi-dex [--main-dex-list=<file>] [--minimal-main-dex]]
[--input-list=<file>]
<file>.class ! <file>.{zip,jar,apk} ! <directory>] ...
 Convert a set of classfiles into a dex file, optionally embedded in a
 jar/zip. Output name must end with one of: .dex .jar .zip .apk or be a directory.
 Positions options: none, important, lines.
 --multi-dex: allows to generate several dex files if needed. This option is
 exclusive with --incremental, causes --num-threads to be ignored and only
 supports folder or archive output.
 --main-dex-list=<file>: <file> is a list of class file names, classes defined by
 those class files are put in classes.dex.
 --minimal-main-dex: only classes selected by --main-dex-list are to be put in
 the main dex. http://blog.csdn.net/
 --input-list: <file> is a list of inputs.
 Each line in <file> must end with one of: .class .jar .zip .apk or be a directory.
dx --annotool --annotation=<class> [--element=<element types>]
[--print=<print types>]
dx --dump [--debug] [--strict] [--bytes] [--optimize]
[--basic-blocks] [--rop-blocks] [--ssa-blocks] [--dot] [--ssa-step=<step>]
[--width=<n>] [<file>.class ! <file>.txt] ...
 Dump classfiles, or transformations thereof, in a human-oriented format.
dx --find-usages <file.dex> <declaring type> <member>
 Find references and declarations to a field or method.
 declaring type: a class name in internal form, like Ljava/lang/Object;
 member: a field or method name, like hashCode
dx -J<option> ... <arguments, in one of the above forms>
 Pass VM-specific options to the virtual machine that runs dx.
dx --version
 Print the version of this tool (1.10).
dx --help
 Print this message.
```

## 2、手动拆包

手动拆包其实就是为了解决低版本的SDK，不支持--multidex参数这种情况，所以我们得想个办法，我们知道，android中有一个步骤就是使用dx命令将class文件变成dex，那么这里我们就可以这么做了，在dx命令执行前，先将javac编译之后的所有class文件进行分类，然后在对具体的分类使用dx来转化成dex文件，这样结果也是有多个dex了。这时候我们可能需要写一个额外的脚本去进行class分类了，具体分多少种dex，那就自己决定了。

上面就说完了拆包的两种方式，其实这两种方式各有优势和缺点，当然我们提倡的还是使用第一种方式，因为现在SDK已经是5.0+了，而且这种方式也是google所倡导的，而且也方便。

**关于上面的两种分包技术，有一个共同需要注意的地方：**

就是Android中在分包之后会有多个dex，但是系统默认会先找到classes.dex文件然后自动加载运行，所以这里就有一个问题，我们需要将一些初始化的重要类放到classes.dex中，不然运行就会报错或者闪退。

那么这里就可以看到这两种分包技术的区别了：

数的限制来进行分类成从dex,比如classes2.dex...classes3.dex...这里没有classes1.dex。需要注意这点。同时，subdex是不支持class的归类的，完全依靠dx自动分类，这个也是为什么叫做自动拆包的原因吧。

```
usage:
 dx --dex [--debug] [--verbose] [--positions=<style>] [--no-locals]
 [--no-optimize] [--statistics] [--no-optimize-list=<file>] [--no-strict]
 [--keep-classes] [--output=<file>] [--dump-to=<file>] [--dump-width=<n>]
 [--dump-method=<name>[*]] [http://blog.csdn.net/]
 [--num-threads=<n>] [--incremental] [-force-jumbo]
 [--multi-dex [--main-dex-list=<file>] [--minimal-main-dex]]
 [--input-list=<file>]
```

但是如果我们要是使用第二种拆包技术的话，就是很随意的操作了，因为本身分类就是我们自己操作的，所以想怎么分就怎么分，而且这里支持分多少个dex，哪些class归到哪个dex，都是可以做到的。灵活性比较好，但是这种方式有一个不好就是需要自己写脚本去归类，这时候就要非常小心，因为可能会遗漏一个class没有归入到具体的dex的话，运行就会报错，找不到这个类。

## 第二步：Dex的加载

在第一步中我们讲解完了拆包技术，如何将class拆分成多个dex。那么问题也就来了，上面我们也说到了，Android中在运行的时候默认只会加载classes.dex这个dex文件，我们也叫作主dex.那么拆分之后还有其他的dex怎么加载到系统运行呢？这时候google就提供了一个方案，使用DexClassLoader来进行动态加载，关于动态加载dex的话，这里不想讲解的太多了，因为我之前的很多文章都介绍了，具体可以参考这篇文章：<http://blog.csdn.net/jiangwei0910410003/article/details/48104581>

所以我们只要获取到所有的dex，除了classes.dex之外。我们然后一一的将各个dex加载到系统中即可。那么这里有两个问题需要解决的：

1、如何获取所有的subdex呢？

这里我们可以这么做，在Application的attachContext方法中(因为这个方法的时机最早)，我们可以获取到程序的apk路径，然后使用ZipFile来解压apk文件，取到所有的subdex文件即可，具体的操作看后面的案例详解。

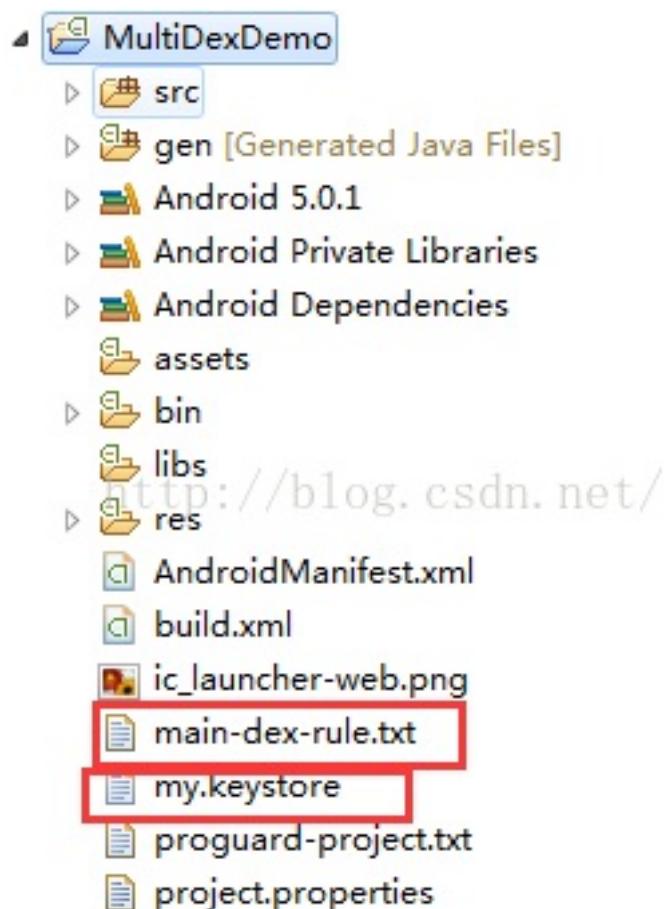
2、我们使用DexClassLoader来加载subdex之后，然后让系统知道？

这里我们使用反射的机制，来合并系统的PathClassLoader和DexClassLoader中的DexList变量值。这个技术也是google提供的。具体技术，也是到后面的案例中我们详细讲解。

## 四、案例分析

到这里我们就介绍完了，我们拆包的两个步骤：拆分dex和加载dex；下面来使用具体的案例来实践一下吧，这个也是网上现在很多介绍了dex拆分技术，但是就是没有具体案例的不好的地方，理论知识谁都知道，但是没有案例讲解的话，就不知道在实际的过程中会遇到什么问题，以及详细的操作步骤，所以这里必须用一个案例来详细介绍一下。

我们的案例采用ant脚本来编译，其实现在google推荐的是gradle来编译，因为这个已经集成到AndroidStudio中了，说句实话，我是因为gradle的语法不太熟，所以就没用gradle来讲解了，当然gradle语法和ant语法很想，如果有同学会gradle的话，可以使用gradle来进行操作一次，这里我不会太多的介绍ant脚本语法，而是介绍详细的命令使用。其实所有的编译脚本理论上就是对编译命令的优化已经加上一些功能罢了。好了，不多说，看案例：



这里的 `main-dex-rule.txt` 是我们后面需要用到的主dex包含的class文件清单，`my.keystore`是签名apk文件

下面我们先来看一下编译脚本`build.xml`

这里再次说明一下，这里不会全部解读，这个脚本是在我之前的一篇文章：

<http://blog.csdn.net/jiangwei0910410003/article/details/50740026>

中的用到的脚本基础上修改的，所以一定要先看这篇文章呀~~

首先来看一下如何使用dx命令来自动拆分dex的：

```
<target
 name="multi-dex"
 depends="compile">
 <echo>
 Generate multi-dex...
 </echo>
 <exec
 executable="${tools.dx}"
 failonerror="true">
 <arg value="--dex" />
 <arg value="--multi-dex" />
 <arg value="--set-max-idx-number=2000" />
 <arg value="--main-dex-list" />
 <arg value="${main-dex-rule}" />
 <arg value="--minimal-main-dex" />
 <arg value="--output=${bin}" />
 <arg value="${bin}" />
 </exec>
</target>
```

这里的参数很简单：

参数说明：

--multi-dex: 多 dex 打包的开关

--main-dex-list=: 参数是一个类列表的文件，在该文件中的类会被打包在第一个 dex 中  
--minimal-main-dex: 只有在--main-dex-list 文件中指定的类被打包在第一个 dex，其余的都在第二个 dex 文件中

因为后两个参数是 optional 参数，所以理论上只需给 dx 加上“--multi-dex”参数即可生成出 classes.dex、classes2.dex、classes3.dex 等。

这里我们指定了--main-dex-list 参数，将指定的 class 合并到 classes.dex 中，看看 main-dex-rule.txt 的内容：

```
1 com/example/multidexdemo/MainActivity.class
2 com/example/multidexdemo/MainActivity$1.class
3 com/example/multidexdemo/MyApplication.class
4 multidex/loader/SecondaryDexEx.class
5 multidex/loader/SecondaryDexEx$LoadedDex.class
```

这里我们将程序的MyApplication类和入口Activity以及需要加载dex的这三个类合并到主dex中，因为这三个是初始化时机最早的，而且也是加载后面subdex类的重要类，所以必须放到主dex中。不然程序都运行不起来的。当然也要注意内部类的情况，也要进行添加的。

好了，我们运行build之后，发现有一个问题，就是我们只看到了一个dex，那就是classes.dex。为什么会这样呢？再看dx的参数，main-dex-list和minimal-main-dex只会影响到主dex中包含的文件，不会影响到从dex是否生成，所以应该是其他原因造成的。

查不到资料，分析源代码就是解决问题的不二法门。于是我把dx.jar反编译了一下，通过分析，找到了下面的几行关键代码：

```
/* 1233 */ this.multiDex = false;
/*
 */
/*
 */
/*
 */
/* 1237 */ this.mainDexListFile = null;
/*
 */
/*
 */
http://blog.csdn.net/
/*
*/
/*
 */
/* 1241 */ this.minimalMainDex = false;
/*
 */
/*
 */
/* 1243 */ this.maxNumberOfIdxPerDex = 65536;
```

显然，dx进行多dex打包时，默认每个dex中方法数最大为65536。而查看当前应用dex的方法数，一共只有51392（方法数没超标，主要是LinearAlloc超标），没有达到65536，所以打包的时候只有一个dex。

再继续分析代码，发现下面一段关键代码：

```
private static boolean processClass(String paramString, byte[] paramArrayOfByte)
{
 if (!args.coreLibrary) {
 checkClassName(paramString);
 }
 DirectClassFile localDirectClassFile = new DirectClassFile(paramArrayOfByte, paramString, args.cfOptions.strictNameCheck);

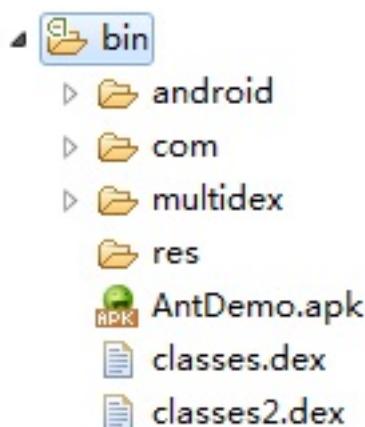
 localDirectClassFile.setAttributeFactory(StdAttributeFactory.THE_ONE);
 localDirectClassFile.getMagic();
 http://blog.csdn.net/

 int i = outputDex.getMethodIds().items().size();
 int j = outputDex.getFieldIds().items().size();
 int k = outputDex.getTypeIds().items().size();
 int m = localDirectClassFile.getConstantPool().size();
 if ((args.multiDex) && ((i + m > args.maxNumberOfIdxPerDex) || (j + m > args.maxNumberOfIdxPerDex) || (k + m + 7 > args.maxNumberOfIdxPerDex))) {
 createDexFile();
 }
}
```

这说明dx有一个隐藏的参数：--set-max-idx-number，这个参数可以用于设置dx打包时每个dex最大的方法数，但是此参数并未在dx的Usage中列出（坑爹啊！）。

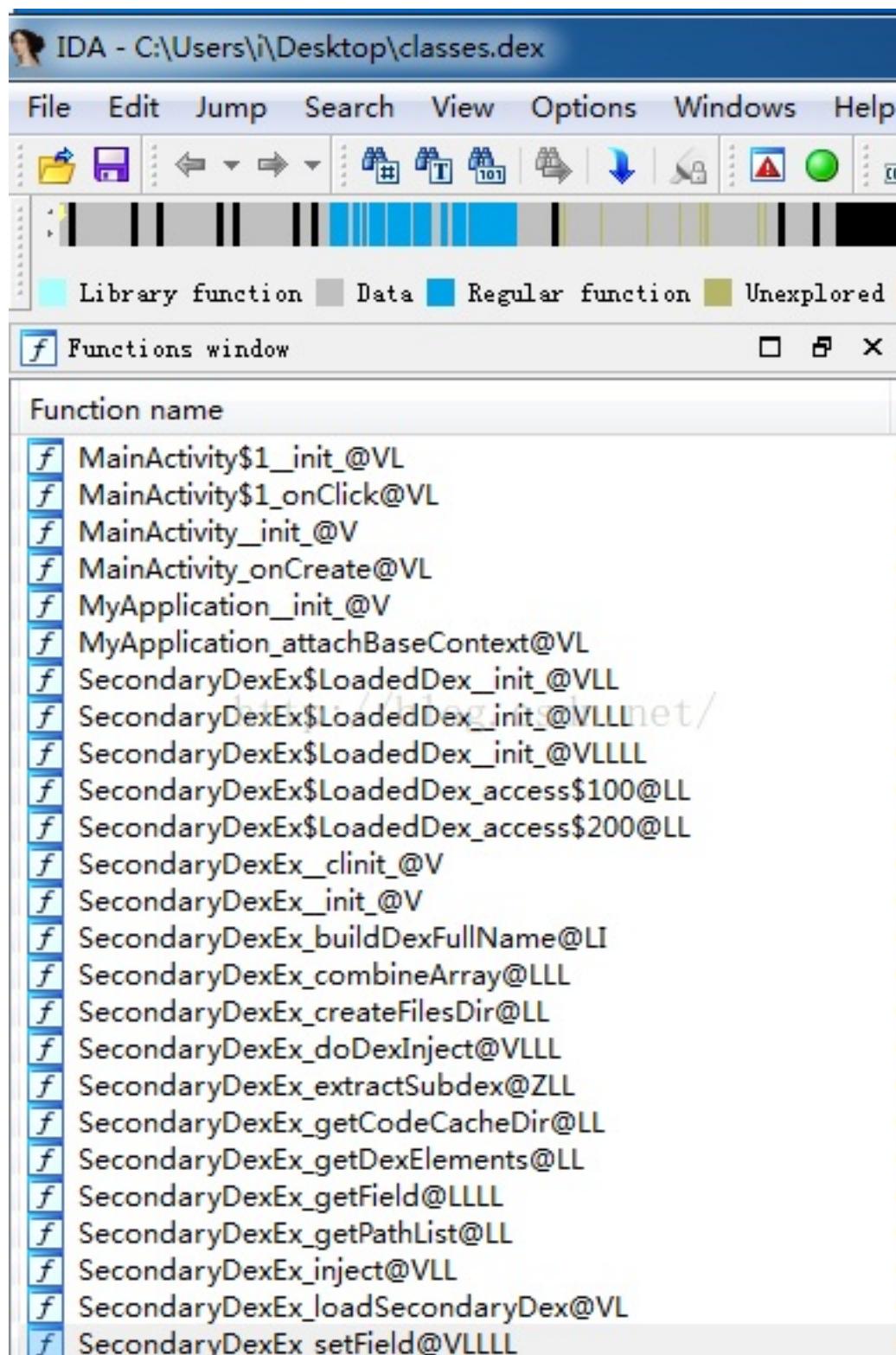
```
else if (localArgumentsParser.isArg("--multi-dex"))
{
 this.multiDex = true;
}
else if (localArgumentsParser.isArg("--main-dex-list="))
{
 this.mainDexListFile = localArgumentsParser.getLastValue();
}
else if (localArgumentsParser.isArg("-minimal-main-dex"))
{
 this.minimalMainDex = true;
}
else if (localArgumentsParser.isArg("--set-max-idx-number="))
{
 this.maxNumber0fIdxPerDex = Integer.parseInt(localArgumentsParser.getLastValue());
}
```

我们在 ant 脚本中把这个参数设置上，暂时设置每个 dex 的方法数最大为：2000，然后在编译：

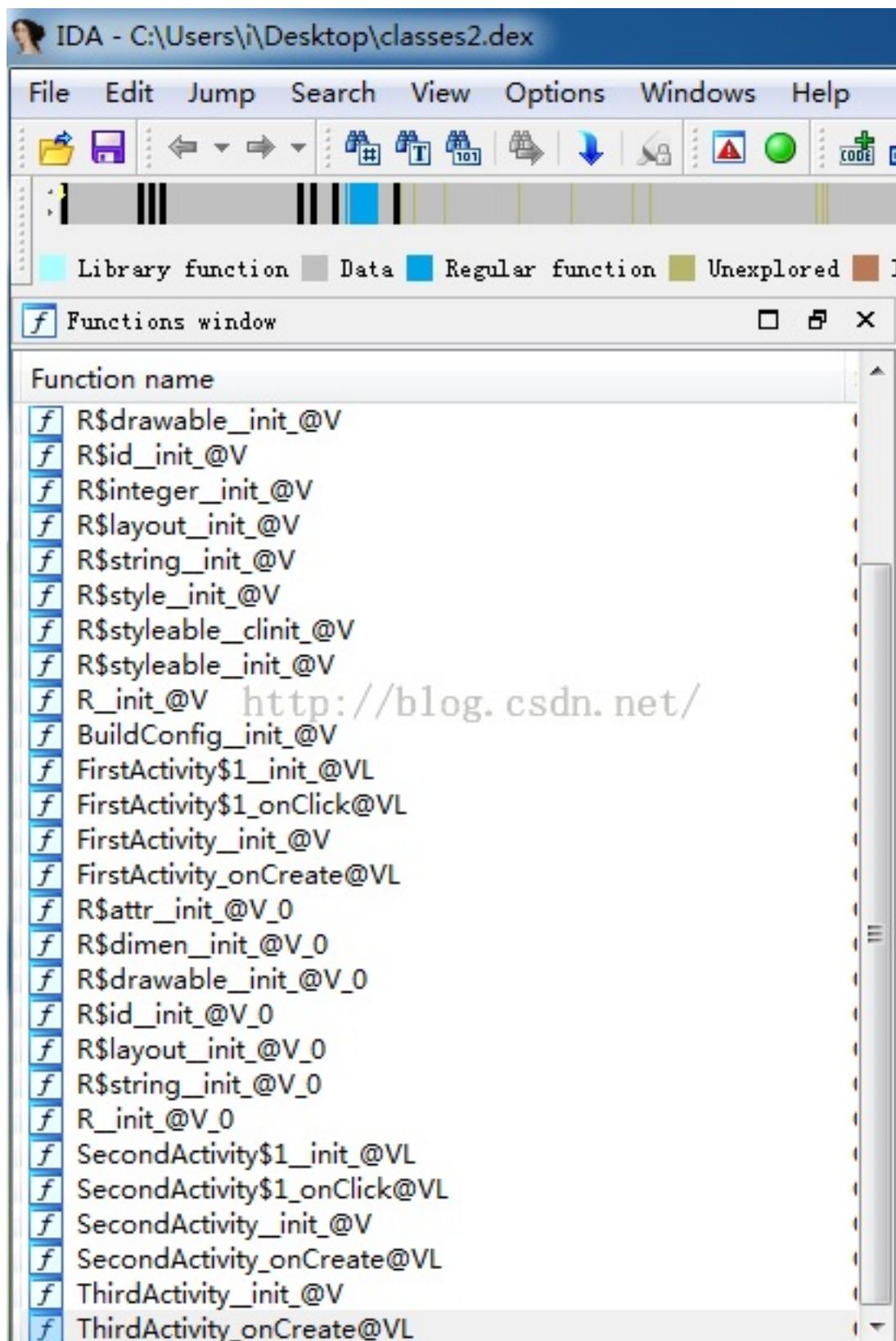


好吧，这下出来了两个dex了，这里我可以发现，没有classes1.dex的，下标是从2开始的，这个需要注意的。这里我们可以用IDA查看dex文件：

classes.dex：



classes2.dex:



看来是成功了，那么下面我们在继续看，现在有了两个dex文件，下面如何编译成apk文件呢？

我们之前知道这时候可以在使用apkbuilder命令将我们编译完的：resource.arsc, asset, dex文件打包成apk文件。

但是这里遇到一个蛋疼问题来了：

apkbuilder命令支持的参数中，只能包含一个dex文件，说白了就是只能打入一个dex文件。

```
-v Verbose.
-d Debug Mode: Includes debug files in the APK file.
-u Creates an unsigned package.
-storetype Forces the KeyStore type. If omitted the default is used.

-z Followed by the path to a zip archive.
 Adds the content of the application package.

-f Followed by the path to a file.
 Adds the file to the application package.

-rf Followed by the path to a source foldersdn.net/
 Adds the java resources found in that folder to the application
 package, while keeping their path relative to the source folder.

-rj Followed by the path to a jar file or a folder containing
 jar files.
 Adds the java resources found in the jar file(s) to the application
 package.

-nf Followed by the root folder containing native libraries to
 include in the application package.
```

这下蛋疼了，该怎么办呢？这里当时急需验证结果，先简单弄了一下，也算是一个小技巧，直接使用WinRAR软件直接把subdex导进去：



然后在单独签名apk，运行，可以了，心情也是很好的，但是感觉这个不能自动化呀。太黑了，所以得想个办法，想到了ZipFile这个类，我们可以解压apk,然后将subdex添加进去，然后把这部分功能打包成一个可执行的jar,然后在build.xml中运行即可。这个也是个方法，但是还是感觉不好，有点繁琐。思前想后，最后找到了一个好办法，也是一个很重要的知识点，那就是使用aapt命令来进行操作，我们在之前可能知道aapt命令用来编译资源文件的，其实他还有一个重要的用途就是可以编辑apk文件：

```
aapt r[emove] [-v] file.{zip,jar,apk} file1 [file2 ...]
Delete specified files from Zip-compatible archive.

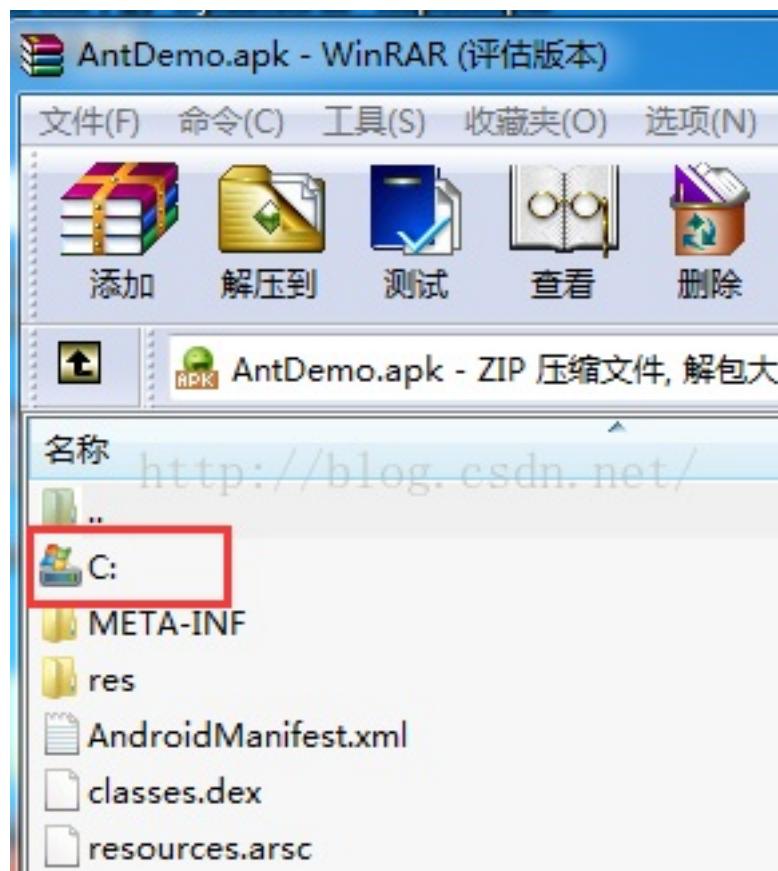
aapt a[dd] [-v] file.{zip,jar,apk} file1 [file2 ...]
Add specified files to Zip-compatible archive.
http://blog.csdn.net/

aapt c[runch] [-v] -S resource-sources ... -C output-folder ...
Do PNG preprocessing on one or several resource folders
and store the results in the output folder.
```

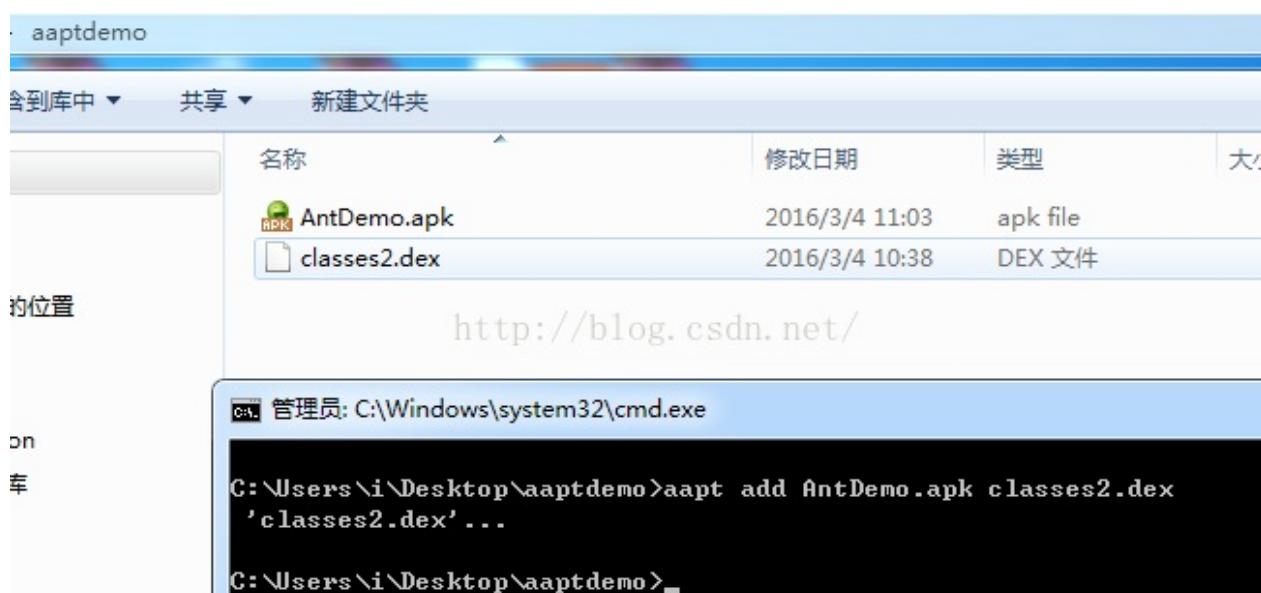
可以看到，我们可以删除apk中的一个文件，或者是添加一个文件，好的，方法终于找到了，下面就直接操作吧，首先我们单独用命令来操作一下看看效果：

```
C:\Users\i\workspace\MultiDexDemo>aapt add C:\Users\i\Desktop\AntDemo.apk C:\Users\i\Desktop\classes2.dex
'C:\Users\i\Desktop\classes2.dex'...
C:\Users\i\workspace\MultiDexDemo>
```

成功了，接下来我们在来看看apk文件：



尼玛，发现还不对，这里有一个坑，就是我们没看到classes2.dex，而看到了一个目录结构，哦，想想应该明白了，aapt命令在添加或者删除的时候，文件的路径必须明确，不能是绝对路径，而是相对路径，这个相对就要相对apk的跟目录，我们现在想把classes2.dex放到apk的根目录下面，那么就应该直接是classes2.dex，我们可以将dex和apk放到一个目录下面来进行操作：



这时候我们在来看一下apk文件：



好吧，成功了，所以这里我们需要注意的是subdex的路径。

下面我们继续来看build.xml脚本内容，看看在拆包成功之后，如何打包成apk中：

```
<!-- 拷贝文件到项目的根目录下面，因为我们的脚本是在根目录下面，这样在运行aapt的时候，可以直接操作dex
<target name="copy_dex"
 depends="build-unsigned-apk">
 <echo message="copy dex..." />
 <copy todir="${project-dir}">
 <fileset dir="${bin}" >
 <include name="classes*.dex" />
 </fileset>
 </copy>
</target>

<!-- 使用aapt命令添加dex文件-->
<target name="aapt-add-dex">
 <echo message="${dir.name}" />
 <!-- 使用正则表达式获取classes的文件名-->
 <propertyregex
 property="dexfile"
 input="${dir.name}"
 regexp="classes(.*).dex"
 select="\0"
 casesensitive="false" />
 <!-- 这里不需要添加classes.dex文件 -->
 <if>
 <equals arg1="${dexfile}" arg2="classes.dex"/>
 <then>
```

```

<echo >${dexfile} is not handle</echo>
</then>
<else>
<echo>${dexfile} is handle</echo>
<exec
 executable="${tools.aapt}"
 failonerror="true" >
 <arg value="add" />
 <arg value="${unsigned-apk-path}" />
 <arg value="${dexfile}" />
</exec>
</else>
</if>
<delete file="${project-dir}\${dexfile}" />
</target>

<!-- 循环遍历bin目录下的所有dex文件 -->
<target name="add-subdex-toapk" depends="copy_dex">
<echo message="Add Subdex to Apk ..." />
<foreach target="aapt-add-dex" param="dir.name">
 <path>
 <fileset dir="${bin}" includes="classes*.dex"/>
 </path>
</foreach>
</target>

```

这里的大体流程是这样的：

- 1、遍历bin目录下所有的dex文件
- 2、将dex文件先拷贝到项目的根目录下面，因为我们的脚本文件是在根目录下面，运行脚本的时候也是在根目录下面，所以需要把dex文件拷贝到这里，不然会发生上面说到的问题，不能使用绝对路径，而是相对路径来添加dex文件。
- 3、因为apk默认的话是包含了classes.dex文件，所以这里需要过滤classes.dex文件，对其他的dex文件进行aapt命令进行添加。

但是这里当时还遇到了一个问题，就是在ant脚本中如何使用循环，正则表达式，条件判断等标签：

具体标签不说了，主要说明一下如何使用这些标签？因为默认情况下，这些标签是不能用的，所以需要导入jar包？

```
<taskdef resource="net/sf/antcontrib/antlib.xml" />
```

网上光说导入这个定义，但是还是报错的：

```
BUILD FAILED
C:\Users\i\workspace\MultiDexDemo\build.xml:259: Problem: failed to create task or type foreach
Cause: The name is undefined.
Action: Check the spelling. http://blog.csdn.net/
Action: Check that any custom tasks/types have been declared.
Action: Check that any <presetdef>/<macrodef> declarations have taken place.
```

这时候我们还需要把：ant-contrib-1.0b3.jar 放到ant的lib目录下面， 默认是没有这个jar的。关于这个jar， 网上很多，自行下载即可

(D:) ▶ Android\_Tools ▶ apache-ant-1.9.6-bin ▶ apache-ant-1.9.6 ▶ lib

文件夹

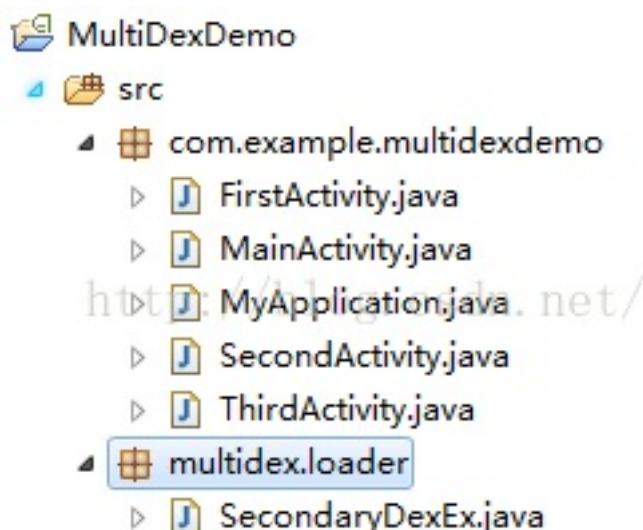
| 名称                      | 修改日期           | 类型                  | 大小     |
|-------------------------|----------------|---------------------|--------|
| ant-apache-bsf.jar      | 2015/6/29 6:45 | Executable Jar File | 20 KB  |
| ant-apache-bsf.pom      | 2015/6/29 6:45 | POM 文件              | 3 KB   |
| ant-apache-log4j.jar    | 2015/6/29 6:45 | Executable Jar File | 9 KB   |
| ant-apache-log4j.pom    | 2015/6/29 6:45 | POM 文件              | 3 KB   |
| ant-apache-oro.jar      | 2015/6/29 6:45 | Executable Jar File | 10 KB  |
| ant-apache-oro.pom      | 2015/6/29 6:45 | POM 文件              | 3 KB   |
| ant-apache-regexp.jar   | 2015/6/29 6:45 | Executable Jar File | 10 KB  |
| ant-apache-regexp.pom   | 2015/6/29 6:45 | POM 文件              | 3 KB   |
| ant-apache-resolver.jar | 2015/6/29 6:45 | Executable Jar File | 10 KB  |
| ant-apache-resolver.pom | 2015/6/29 6:45 | POM 文件              | 3 KB   |
| ant-apache-xalan2.jar   | 2015/6/29 6:45 | Executable Jar File | 8 KB   |
| ant-apache-xalan2.pom   | 2015/6/29 6:45 | POM 文件              | 4 KB   |
| ant-commons-logging.jar | 2015/6/29 6:45 | Executable Jar File | 10 KB  |
| ant-commons-logging.pom | 2015/6/29 6:45 | POM 文件              | 3 KB   |
| ant-commons-net.jar     | 2015/6/29 6:45 | Executable Jar File | 90 KB  |
| ant-commons-net.pom     | 2015/6/29 6:45 | POM 文件              | 4 KB   |
| ant-contrib-1.0b3.jar   | 2016/3/3 16:25 | Executable Jar File | 220 KB |

这时候我们就可以运行build.xml脚本了。

上面就介绍完了使用脚本拆包，以及如何将多个dex打包成apk.

下面来看一下代码如何实现：

代码中我们定义了四个Activity：



MainActivity是我们的入口Activity，SecondaryDexEx是动态加载的核心类

首先如何从apk中获取dex文件：

```

ZipFile apkFile = null;
try {
 apkFile = new ZipFile(getApplicationContext().getApplicationInfo().sourceDir);
} catch (Exception e) {
 Log.i("multidex", "create zipfile error:"+Log.getStackTraceString(e));
 return;
}

Log.i("multidex", "zipfile:"+apkFile.getName());

File filesDir = applicationContext.getDir("odex", Context.MODE_PRIVATE);
Log.i("multidex", "filedir:"+filesDir.getAbsolutePath());
for(int i = 0 ; i < SUB_DEX_NUM; i ++){
 String possibleDexName = buildDexFullName(i);
 Log.i("multidex", "pronyme:"+possibleDexName);
 ZipEntry zipEntry = apkFile.getEntry(possibleDexName);
 Log.i("multidex", "entry:"+zipEntry);
 if(zipEntry == null) {
 break;
 }
 msLoadedDexList.add(new LoadedDex(filesDir,possibleDexName,zipEntry));
}

```

先得到应用apk的路径，使用ZipFile来解压apk,获取所有的dex文件，这里我们不需要处理classes.dex文件了，默认就会加载，所以这里进行过滤一下，还有就是subdex的下标都是从2开始的，没有classes1.dex。需要注意的。同时这里获取应用的apk文件是不需要任何权限限制的，类似于QQ中可以分享本地app给好友的功能一样。

接下来我们获取到了所有的subdex之后，然后就可以动态加载了，需要把我们的dex加载到系统中，这里主要就是使用反射技术合并dexList即可。

```
/**
 * 这里需要注入DexClassLoader
 * 因为GDT内部点击广告是去下载App, 是启动一个DownloadService去下载App
 * 所以这里需要这么做，不然会报异常
 * @param loader
 */
private static void inject(DexClassLoader loader, Context ctx){
 PathClassLoader pathLoader = (PathClassLoader) ctx.getClassLoader();
 try {
 Object dexElements = combineArray(
 getDexElements(getPathList(pathLoader)),
 getDexElements(getPathList(loader)));
 Object pathList = getPathList(pathLoader);
 setField(pathList, pathList.getClass(), "dexElements", dexElements);
 } catch (Exception e) {
 Log.i("multidex", "inject dexclassloader error:" + Log.getStackTraceString(e));
 }
}

private static Object getPathList(Object baseDexClassLoader)
 throws IllegalArgumentException, NoSuchFieldException, IllegalAccessException
return getField(baseDexClassLoader, Class.forName("dalvik.system.BaseDexClassLoader"));

private static Object getField(Object obj, Class<?> cl, String field)
 throws NoSuchFieldException, IllegalArgumentException, IllegalAccessException
Field localField = cl.getDeclaredField(field);
localField.setAccessible(true);
return localField.get(obj);
}

private static Object getDexElements(Object paramObject)
 throws IllegalArgumentException, NoSuchFieldException, IllegalAccessException
return getField(paramObject, paramObject.getClass(), "dexElements");
}

private static void setField(Object obj, Class<?> cl, String field,
 Object value) throws NoSuchFieldException,
 IllegalArgumentException, IllegalAccessException {

 Field localField = cl.getDeclaredField(field);
 localField.setAccessible(true);
 localField.set(obj, value);
}

private static Object combineArray(Object arrayLhs, Object arrayRhs) {
 Class<?> localClass = arrayLhs.getClass().getComponentType();
 int i = Array.getLength(arrayLhs);
 int j = i + Array.getLength(arrayRhs);
```

```
Object result = Array.newInstance(localClass, j);
for (int k = 0; k < j; ++k) {
 if (k < i) {
 Array.set(result, k, Array.get(arrayLhs, k));
 } else {
 Array.set(result, k, Array.get(arrayRhs, k - i));
 }
}
return result;
}
```

在MyApplication中的attachBaseContext方法中调用loadSecondaryDex方法即可：

```
public class MyApplication extends Application{
 @Override
 protected void attachBaseContext(Context base) {
 super.attachBaseContext(base);
 SecondaryDexEx.LoadSecondaryDex(base);
 }
}
```

到这里我们介绍完了脚本和代码，下面就开跑一边脚本吧：ant release

```
C:\Users\i\workspace\MultiDexDemo>ant release
Buildfile: C:\Users\i\workspace\MultiDexDemo\build.xml

init:
[echo]
[echo] Initialize...
[echo]
[delete] Deleting directory C:\Users\i\workspace\MultiDexDemo\bin
[mkdir] Created dir: C:\Users\i\workspace\MultiDexDemo\bin

gen-R:
[echo]
[echo] Generating R.java from the resources...
[echo]

compile: http://blog.csdn.net/
[echo]
[echo] Compile...
[echo]
[javac] Compiling 9 source files to C:\Users\i\workspace\MultiDexDemo\bin
[javac] C:\Users\i\workspace\MultiDexDemo\gen\android\support\w7\appcompat\R.java
[javac] C:\Users\i\workspace\MultiDexDemo\gen\com\example\multidexdemo\BuildConfig.java
[javac] C:\Users\i\workspace\MultiDexDemo\gen\com\example\multidexdemo\R.java
[javac] C:\Users\i\workspace\MultiDexDemo\src\com\example\multidexdemo\FirstActivity.java
[javac] C:\Users\i\workspace\MultiDexDemo\src\com\example\multidexdemo>MainActivity.java
[javac] C:\Users\i\workspace\MultiDexDemo\src\com\example\multidexdemo\MyApplication.java
[javac] C:\Users\i\workspace\MultiDexDemo\src\com\example\multidexdemo\SecondActivity.java
[javac] C:\Users\i\workspace\MultiDexDemo\src\com\example\multidexdemo\ThirdActivity.java
[javac] C:\Users\i\workspace\MultiDexDemo\src\multidex\loader\SecondaryDexEx.java

multi-dex:
[echo]
[echo] Generate multi-dex...
[echo]

package:
[echo]
[echo] Package resource and assets...
[echo]

build-unsigned-apk:
[echo]
[echo] Build unsigned apk
[echo]
[java]
[java] THIS TOOL IS DEPRECATED. See --help for more information.
[java]

copy_dex:
[echo] copy dex...
[copy] Copying 2 files to C:\Users\i\workspace\MultiDexDemo

add-subdex-toapk:
[echo] Add Subdex to Apk ...

aapt-add-dex:
[echo] C:\Users\i\workspace\MultiDexDemo\bin\classes.dex
[echo] classes.dex is not handle
[delete] Deleting: C:\Users\i\workspace\MultiDexDemo\classes.dex
http://blog.csdn.net/

aapt-add-dex:
[echo] C:\Users\i\workspace\MultiDexDemo\bin\classes2.dex
[echo] classes2.dex is handle
[exec] 'classes2.dex'...
[delete] Deleting: C:\Users\i\workspace\MultiDexDemo\classes2.dex

sign-apk:
[echo]
[echo] Sign apk
[echo]
[exec] jar 已签名。
[exec]
[exec] 警告:
[exec] 未提供 -tsa 或 -tsacert, 此 jar 没有时间戳。如果没有时间戳, 则在签名者证书的到期日期 <2015-06-25>
[exec]

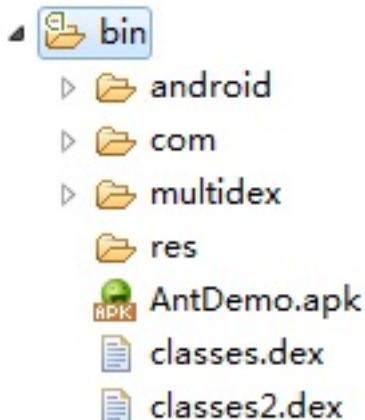
release:
```

```
[Delete] Deleting: C:\Users\i\workspace\MultiDexDemo\bin\AntDemo.arsc
[Delete] Deleting: C:\Users\i\workspace\MultiDexDemo\bin\AntDemo_unsigned.apk
[echo]
[echo] APK is released. path:..\bin\AntDemo.apk
[echo]

BUILD SUCCESSFUL
Total time: 6 seconds

C:\Users\i\workspace\MultiDexDemo>
```

跑完之后，我们看一下bin目录下多了一个成功的apk文件：



我们安装运行，结果如下：



看到了，我们首先启动的是MainActivity入口，然后我们依次点击btn进入不同的Activity。结果运行正常，同时我们也可以查看一下日志，获取dex和加载dex是正常的。

```
C:\Users\i>adb logcat -s multidex
----- beginning of /dev/log/system
----- beginning of /dev/log/main
I/multidex< 3133>: context:android.app.ContextImpl@42f24d40
I/multidex< 3133>: zipfile:/data/app/com.example.multidexdemo-1.apk
I/multidex< 3133>: filedir:/data/data/com.example.multidexdemo/app_ode
I/multidex< 3133>: pronyme:classes2.dex/csdn.net/
I/multidex< 3133>: entry:classes2.dex
I/multidex< 3133>: pronyme:classes3.dex
I/multidex< 3133>: entry:null
I/multidex< 3133>: dex size:1
I/multidex< 3133>: result:true
```

到这里我们的心情很激动，终于搞定了拆包，遇到一些问题，但是我们都解决了。哈哈~~

项目下载：<http://download.csdn.net/detail/jiangwei0910410003/9452599>

## 五、梳理流程

下面我们来总结一下我们做的哪些工作：

- 1、首先我们使用脚本进行自动拆包，得到两个dex文件。
- 2、因为apkbuilder命令在打apk包的时候，只能包含一个dex文件，所以我们需要在使用aapt命令添加其他的subdex文件
- 3、得到应用程序的apk文件，然后得到所有的dex文件，在使用DexClassLoader进行subdex的加载。

但是关于我们之前说到的拆包的第二个技术，手动拆包这里就没有介绍了，其实很简单，就是在ant脚本中进行分类copy文件就好了，得到了多个dex之后，同样使用aapt命令添加到apk中就可以了。

## 六、知识点概要

学习到了哪些知识点：

- 1、更加深入的了解了ant脚本的语法
- 2、学习到了使用dx命令来进行自动拆包
- 3、复习了DexClassLoader的用法

## 七、遇到的错误

我们在这个过程中可能会遇到一些错误，最多的就是：ClassNotFound和NotDefClass这两个错误，这个处理很简单的，我们查看是哪个类，然后用IDA查看每个dex文件是否包含了这个类，如果没有，那说明没有将这个类归类进去

## 八、性能分析

在冷启动时因为需要加载多个DEX文件，如果DEX文件过大时，处理时间过长，很容易引发ANR (Application Not Responding)；

采用MultiDex方案的应用可能不能在低于Android 4.0 (API level 14) 机器上启动，这个主要是因为Dalvik linearAlloc的一个bug (Issue 22586);采用MultiDex方案的应用因为需要申请一个很大的内存，在运行时可能导致程序的崩溃，这个主要是因为Dalvik linearAlloc 的一个限制(Issue 78035). 这个限制在 Android 4.0 (API level 14)已经增加了，应用也有可能在低于 Android 5.0 (API level 21)版本的机器上触发这个限制。

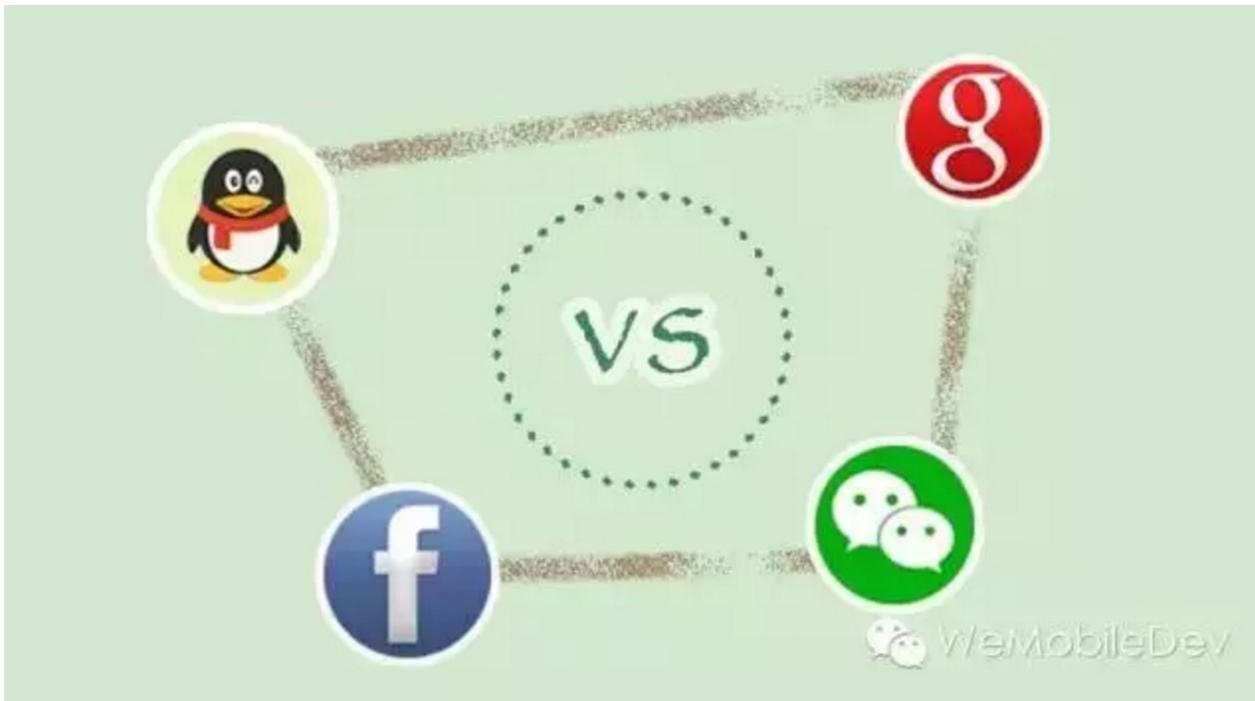
Dex分包后，如果是启动时同步加载，对应用的启动速度会有一定的影响，但是主要影响的是安装后首次启动。这是因为安装后首次启动时，Android系统会对加载的从dex做Dexopt并生成ODEX，而 Dexopt 是比较耗时的操作，所以对安装后首次启动速度影响较大。在非安装后首次启动时，应用只需加载 ODEX，这个过程速度很快，对启动速度影响不大。同时，从dex 的大小也直接影响启动速度，即从dex 越小则启动越快。

## 九、总结

关于Android中的拆包技术网上已经有很多案例和解释了，但是大部分都是理论知识，没有说到细节的，更没有什么demo。所以这篇文章就主要详细介绍了分包的技术点以及用一个demo来进行案例分析，更加深入的了解了拆包技术，本文中虽然用到了是ant脚本进行编译的，但是同样可以使用gradle来进行编译，只要理解了原理，其实都是命令在操作的，和脚本没有任何关系的。从此我们也不会在感觉拆包技术多不觉明历了。其实就是那么回事。

# Android拆分与加载Dex的多种方案对比

来源:[微信团队](#)



对于Android大型程序来说，64k方法数与线性内存的限制都是必须要考虑的问题。对于它们的原理与分析，可参考下面这篇文章：[預防 Android Dex 64k Method Size Limit](#)。同时Android官方也推出了自己的解决方案，但却不能满足所有应用的需求。

事实上，解决64K方法限制的唯一方法是拆分多dex，不同方案的差异在于需指定哪些类必须在主dex，这与我们期待的效果以及加载方式相关。解决安装过程线性内存的方法是限制主dex的linearalloc大小，这里经验值为3355444(2.2以上，如何计算？)。对于运行过程的线性内存限制可参考Facebook。下面分别对我了解的几种分dex与加载dex的方案作简单的对比分析。

## Android官方方案

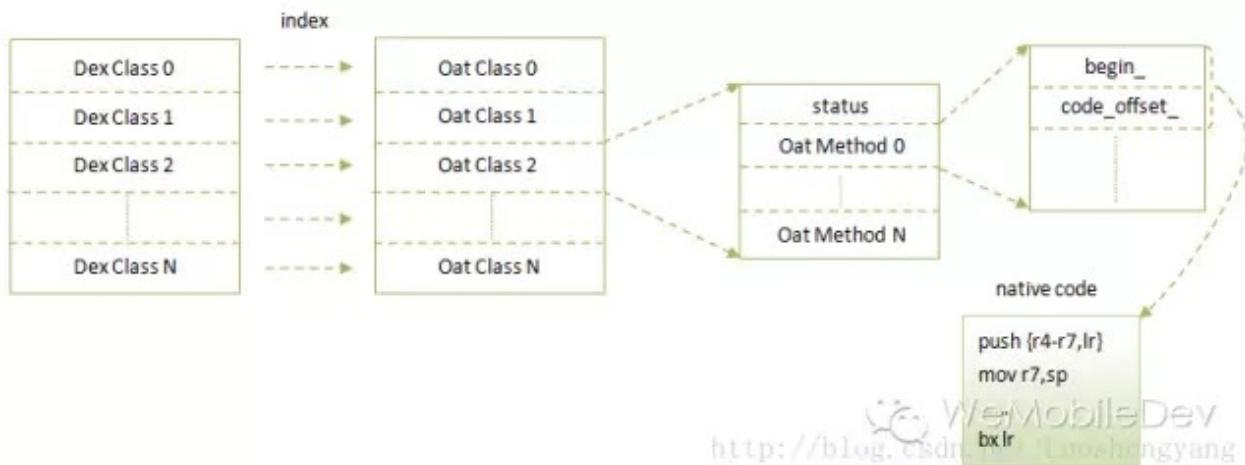
Android推出了官方的[multidex support library](#)。集成与使用的方法非常简单，下面从几方面简述：

### 1. Dex形式

我们只需指定`multiDexEnabled`, 即可编译自动拆分多dex, 拆分出来的dex以`classes(..N).dex`形式命名, 与主dex同样放于安装包的根目录。

```
defaultConfig {
 ...
 // Enabling multidex support.
 multiDexEnabled true
}
```

为什么要以`classes(..N).dex`, 而不是我们常见的放于`assets`?这是为了5.0以上系统在安装过程中的art阶段就将所有的`classes(..N).dex`合并到一个单独的oat文件(5.0以下只能苦逼的启动时加载)。对于Art相关知识, 可以参考老罗的系列文章。



上图即为一个oat文件的格式图。Android采取这种方式, 明显也是为了擦之前的屁股。另一方面, 最新报告5.0以上已经占了超过9%, 这说明是非常有必要采用这种方式以减少首次启动的耗时。

## 2. Dex类分包的规则

我们只是指定了`multiDexEnabled`, 那系统会将那些类放在主dex?其实它利用的是Android sdk build tool中的`mainDexClasses`脚本, 这在版本21以上才会有。使用方法非常很简单:

```
mainDexClasses [--output <output file>] <application path>
```

该脚本要求输入一个文件组 (包含编译后的目录或jar包), 然后分析文件组中的类并写入到`--output`所指定的文件中。实现原理也不复杂, 主要分为三步:

- a. 环境检查, 包括传入参数合法性检查, 路径检查以及proguard环境检测等。
- b. 使用`mainDexClasses.rules`规则, 通过Proguard的shrink功能, 裁剪无关类, 生成一个`tmp.jar`包。
- c. 通过生成的`tmp jar`包, 调用`MainDexListBuilder`类生成主dex的文件列

表。

这里只是简单的得到所有入口类(即rules中的Instrumentation、application、Activity、Annotation等等)的直接引用类。何为直接引用类? 在init过程, 会在校验阶段去resolve它各个方法、变量引用到的类, 这些类统称为某个类的直接引用类。举个栗子:

```
public class MainActivity extends Activity {
 protected void onCreate(Bundle savedInstanceState) {
 DirectReferenceClass test = new DirectReferenceClass();
 }
}

public class DirectReferenceClass {
 public DirectReferenceClass() {
 InDirectReferenceClass test = new InDirectReferenceClass();
 }
}

public class InDirectReferenceClass {
 public InDirectReferenceClass() {

 }
}
```

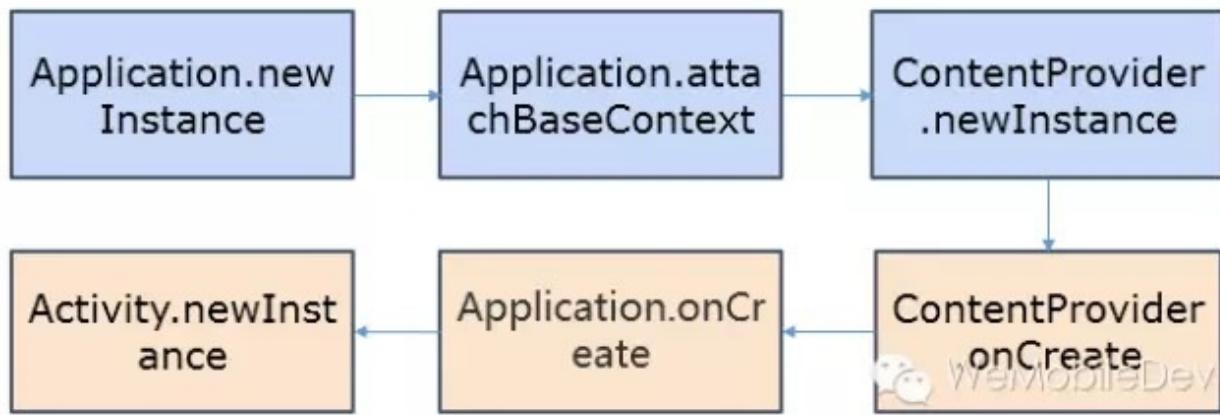
上面有 `MainActivity` 、 `DirectReferenceClass` 、 `InDirectReferenceClass` 三个类, 其中 `DirectReferenceClass` 是 `MainActivity` 的直接引用类, `InDirectReferenceClass` 是 `DirectReferenceClass` 的直接引用类。而 `InDirectReferenceClass` 是 `MainActivity` 的间接引用类(即直接引用类的所有直接引用类)。

### 3. 加载Dex的方式

对于5.0以下的系统, 我们需要在启动时手动加载其他的dex。而我们并没有要求得到所有的间接引用类, 这是因为我们在 `attachBaseContext` 的时候, 已将其他dex加载。例如:

```
public class HelloMultiDexApplication extends Application {
 @Override
 protected void attachBaseContext(Context base) {
 super.attachBaseContext(base);
 MultiDex.install(this);
 }
}
```

`attachBaseContext` 究竟处于生命周期的哪一步? 可看下图:



事实上，若我们在 `attachBaseContext` 中调用 `Multidex.install`，我们只需引入 `Application` 的直接引用类即可，`mainDexClasses` 将 `Activity`、`ContentProvider`、`Service` 等的直接引用类也引入，主要是满足需要在非 `attachBaseContext` 加载多dex的需求。另一方面，若存在以下代码，将出现 `NoClassDefFoundError` 错误。

```

public class HelloMultiDexApplication extends Application {
 @Override
 protected void attachBaseContext(Context base) {
 super.attachBaseContext(base);
 DirectReferenceClass test = new DirectReferenceClass();
 MultiDex.install(this);
 }
}

```

这是因为在实际运行过程中，`DirectReferenceClass` 需要的 `InDirectReferenceClass` 并不一定在主dex。解决方法是手动将该类放于dx的 `-main-dex-list` 参数中：

```

afterEvaluate {
 tasks.matching {
 it.name.startsWith('dex')
 }.each { dx ->
 if (dx.additionalParameters == null) {
 dx.additionalParameters = []
 }
 dx.additionalParameters += '--multi-dex'
 dx.additionalParameters += "--main-dex-list=$projectDir/<filename>.toString()
 }
}

```

Android提供的方案，或者延伸为在 `attachBaseContext` 中同步加载dex的方案，它的好处是非常简单，所需的依赖集也非常少。但是它的缺点也非常明显，即若其他dex比较大，首次加载时会出现明显的黑屏，甚至会出现ANR。

## 微信/手Q加载方案

对于微信来说，我们一共有 111052 个方法。以线性内存 3355444 (限制5m,给系统预留部分)、方法数 64K 为限制，即当满足任意一个条件时，将拆分dex。由此微信将得到一个主dex,两个子dex，若微信采用Android方案，在首次启动时将长期无响应(没有出现黑屏时因为默认皮肤的原因)，这对处女座的我来说是无法接受的。应该如何去做？微信与手Q的方案是类似的，将首次加载放于地球中，并用线程去加载(但是5.0之前加载dex时还是会挂起主线程)。

### 1. Dex形式

暂时我们还是放于 `assets` 下，以 `assets/secondary-program-dex-jars/secondary-N.dex.jar` 命名。为什么不以 `classes(..N).dex` ? 这是因为一来觉得以Android的推广速度，5.0用户增长应该是遥遥无期的，二来加载Dex的代码，传进去的是zip，在加载前我们需要验证MD5，确保所加载的Dex没有被篡改(Android官方没有验证，主要是只有root才能更改吧)。

```
/**
 * Makes an array of dex/resource path elements, one per element of
 * the given array.
 */
private static Element[] makeDexElements(ArrayList<File> files, File optimizedDirectory,
 ArrayList<IOException> suppressedExceptions)
```

事实上，应该传进去的是dex也是应该可以的，这块在下一个版本将采用 `classes(..N).dex` 。但是如果我们将使用了线程加载，并且弹出提示界面，对用户来说并不是无法接受。

### 2. Dex类分包的规则

分包规则即将所有 `Application` 、`ContentProvider` 以及所有 `export` 的 `Activity` 、`Service` 、`Receiver` 的间接依赖集都必须放在主dex。对于微信现在来说，这部分大约有 41306 个方法，每次通过扫描 `AndroidManifest` 计算耗时大约为 20s不到。怎么计算？可以参考 `buck` 或者 `mainDexClasses` 的做法。

```
public MainDexListBuilder(String rootJar, String pathString) throws IOException {
 path = new Path(pathString);
 ClassReferenceListBuilder mainListBuilder=new ClassReferenceListBuilder(path);
```

### 3. 加载Dex的方式

加载逻辑这边主要判断是否已经 dexopt，若已经 dexopt，即放在 attachBaseContext 加载，反之放于地球中用线程加载。怎么判断？其实很低级，因为在微信中，若判断 revision 改变，即将 dex 以及 dexopt 目录清空。只需简单判断两个目录dex名称、数量是否与配置文件的一致。

```
(name md5 校验是否加载成功)
secondary-1.dex.jar 63e5240eac9bdb5101fc35bd40a98679 secondary.dex01.Canary
secondary-2.dex.jar e7d2a4a181f579784a4286193feaf457 secondary.dex02.Canary
```

总的来说，这种方案用户体验较好，缺点在于太过复杂，每次需重新扫描依赖集，而且使用的是比较大的间接依赖集(要真正运行到，直接依赖集是不行的)。当前微信必要的依赖集已经41306个方法，说不定哪一天就爆了。

## FaceBook加载方案

那是否存在一种加载方式它的依赖集很小，但却不会像官方方案一样造成明显的卡顿？逆过不少app,发现 facebook 的思路还是挺不错的，下面作一个简单的说明：

### 1. Dex形式

微信与 facebook 的 dex 形式是完全一致的，这是因为我们也是使用 facebook 开源工具 buck 编译的。但是我们做了一个自动生成 buck 脚本的工作，即开发人员无须关心 buck 脚本如何编写。

### 2. Dex类分包的规则

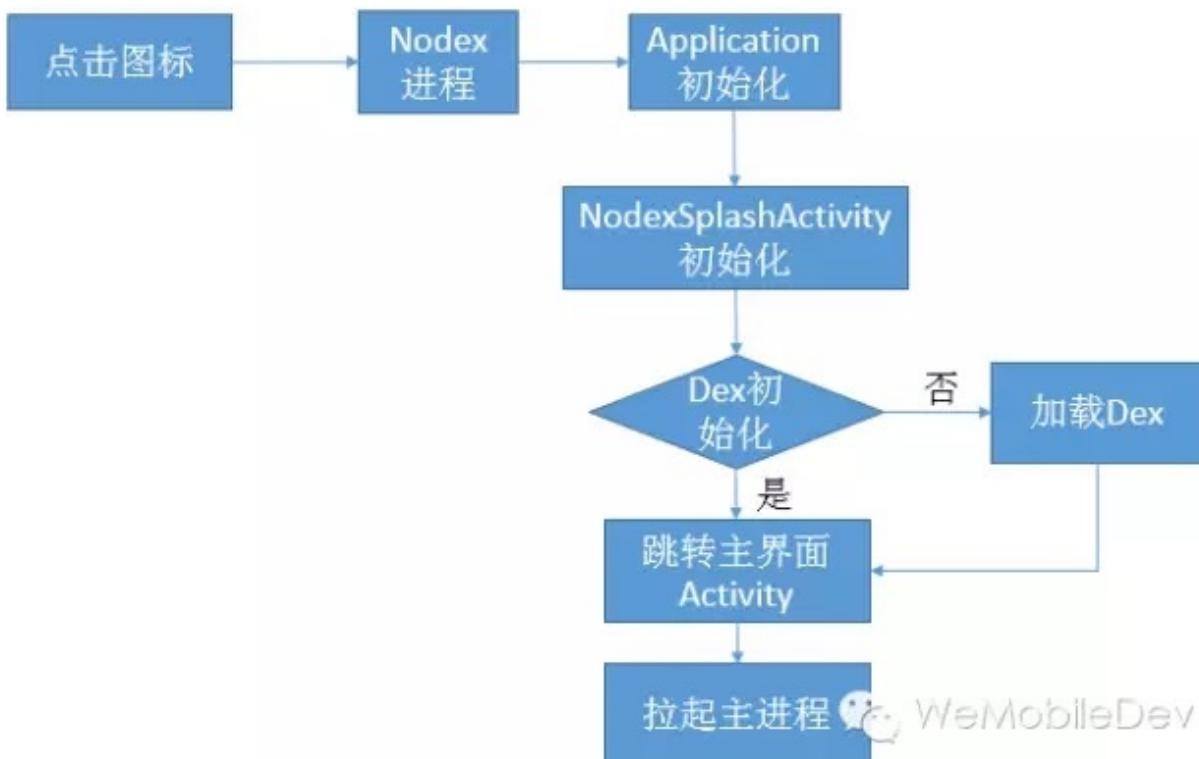
facebook 将加载 Dex 的逻辑放于单独的 nodex 进程，这是一个非常简单、轻量级的进程。它没有任何的 contentProvider，只有有限的几个 Activity 、 Service 。

```
<activity android:exported="false" android:process=":nodex"
 android:name="com.facebook.nodex.startup.splashscreen.NodexSplashActivity">
```

所以依赖集为 `Application`、`NodexSplashActivity` 的间接依赖集即可，而且这部分逻辑应该相对稳定，我们无须做动态扫描。这就实现了一个非常轻量级的依赖集方案。

### 3. 加载Dex的方式

加载dex逻辑也非常简单，由于 `NodexSplashActivity` 的 `intent-filter` 指定为 `Main` 与 `LAUNCHER`。首先拉起 `nodex` 进程，然后初始化 `NodexSplashActivityActivity`，若此时 Dex 已经初始化过，即直接跳转到主页面。



这种方式好处在于依赖集非常简单，同时首次加载 Dex 时也不会卡死。但是它的缺点也很明显，即每次启动主进程时，都需先启动 `nodex` 进程。尽管 `nodex` 进程逻辑非常简单，这也需 100ms 以上。若微信对启动时间非常敏感，很难会去采用这个方案。

## 测试加载方案

`Facebook` 的缺陷在于多起一个 `nodex` 进程，那是否可以直接在主进程做这个操作？想到一种方案，通过简单测试应该可行，下面做简单说明：

### 1. Dex形式

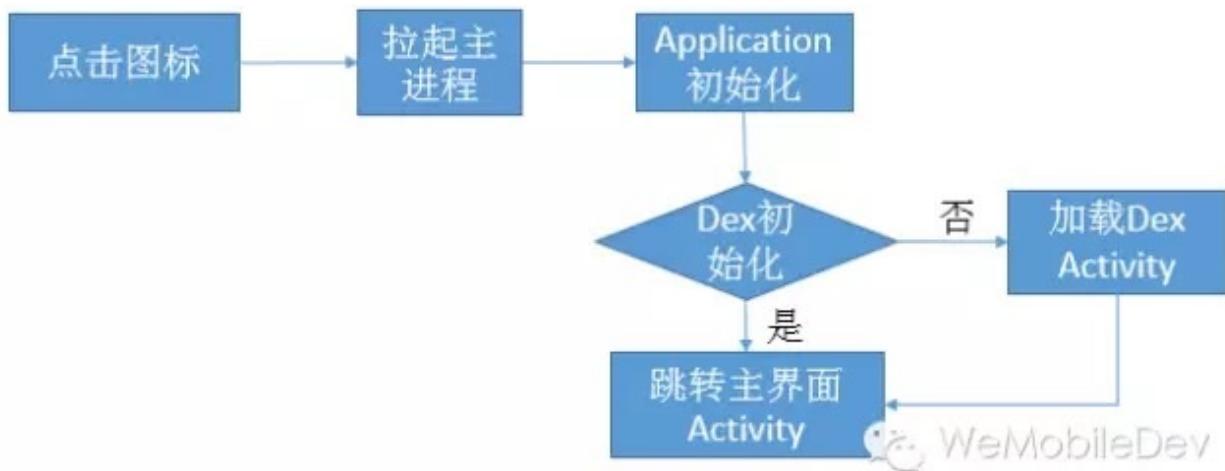
Dex 形式并不是重点，假定我们使用当前微信的Dex形式，即 `assets/secondary-program-dex-jars/secondary-N.dex.jar`。

## 2. Dex类分包的规则

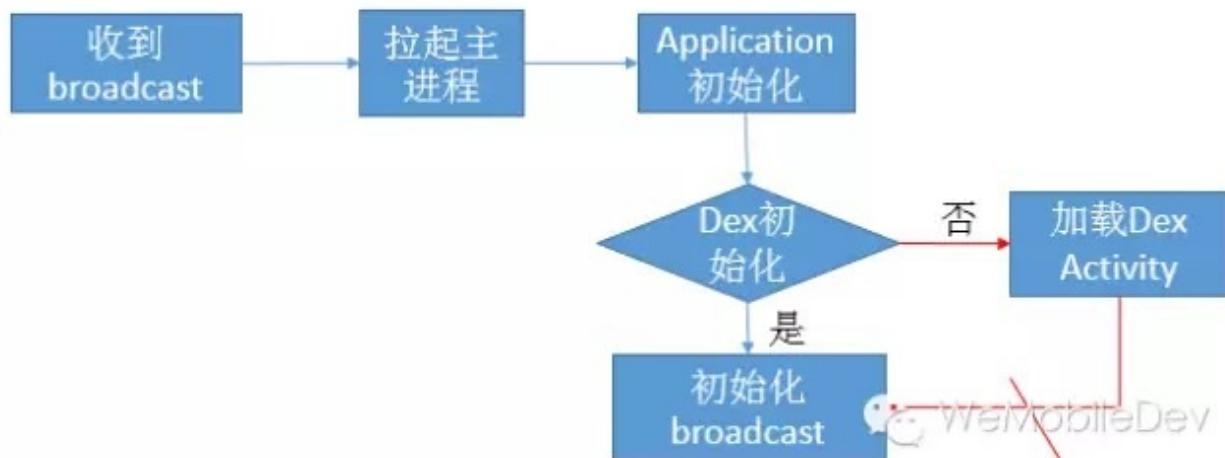
主 Dex 应该保证简单，即类似 Facebook，只需要少量与Dex加载相关的类即可，并且这部分代码是相对稳定。我也无须去更改任何非加载相关的代码，即不会像微信/QQ方案，我们需要修改 `BaseExportActivity`、`BaseExportServer`、`BaseExportBroadcast` 等代码。

## 3. 加载Dex的方式

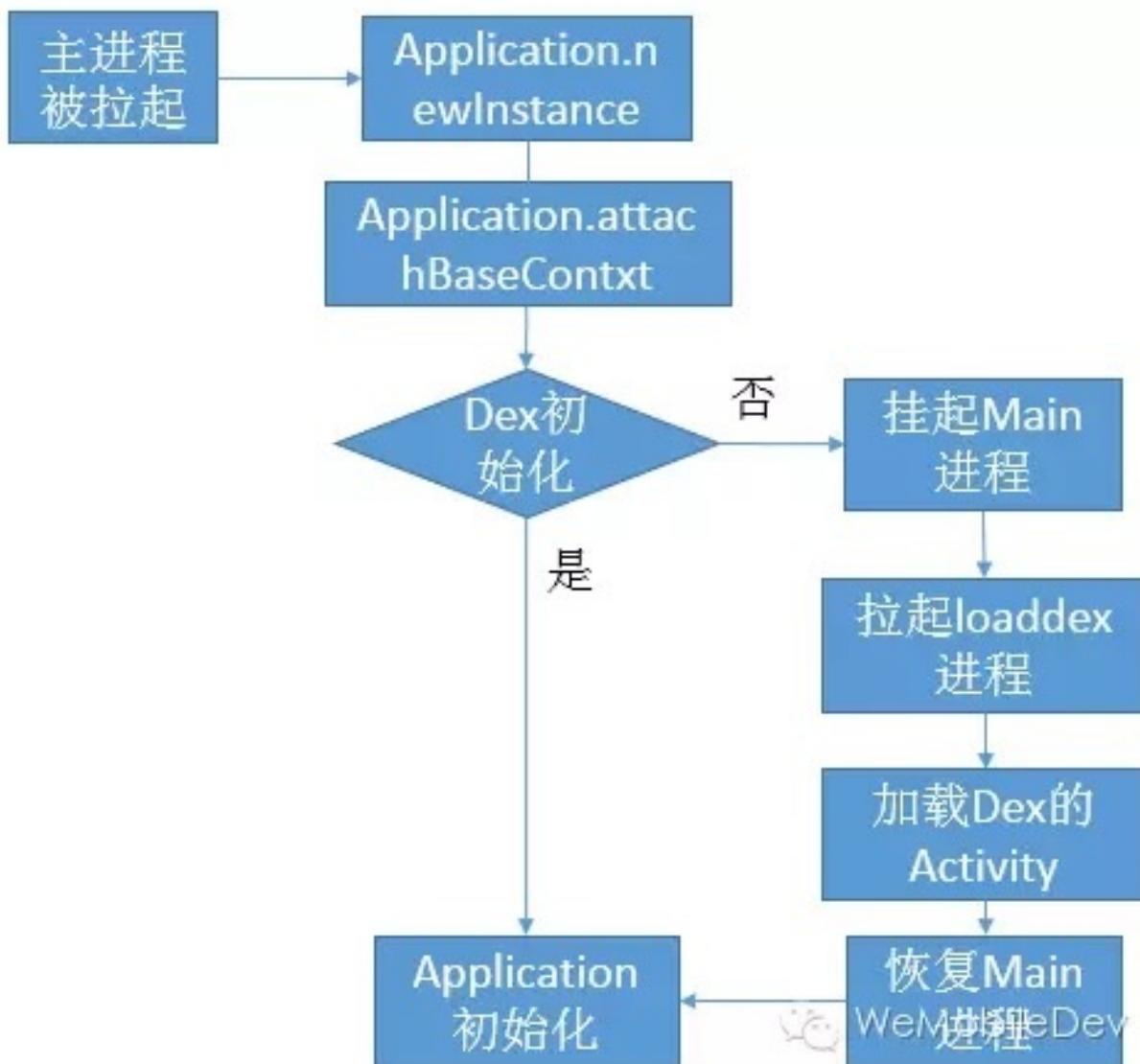
回到重点，我们应该通过什么加载方案去实现这样的分包规则。首先若是点击图标，我们的确无须再起一个进程，即下面是可行的：



但是问题就在于在 Application 初始化时，或是在 attachBaseContext 时，我们无法确保即将进入的是主界面 Activity。可能系统要起的是某一个 service 或 Receiver，这种跳转方式是不行的。例如下图中的红色部分，我们无法知道将跳转到哪里：



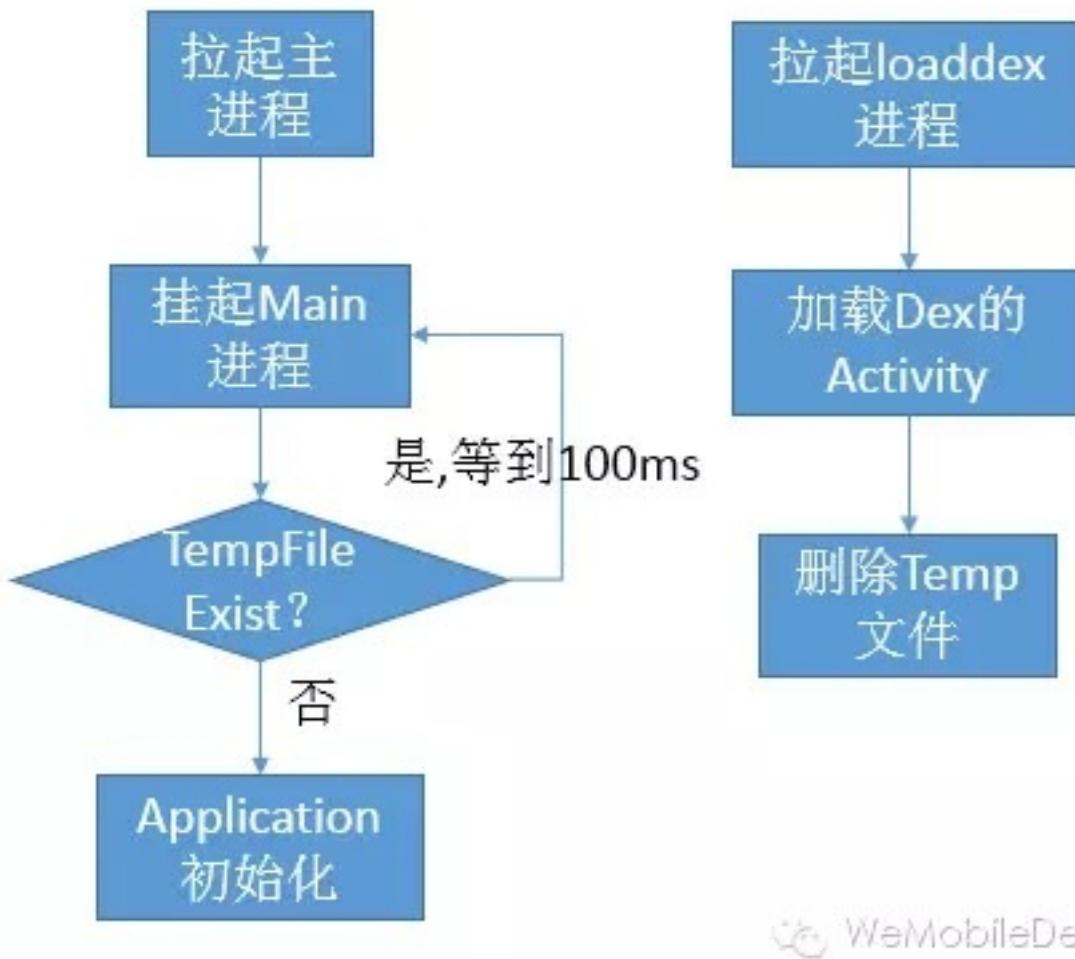
能不能换种思路，即假设发现Dex没有初始化，在 attachBaseContext 的时候挂起主进程，然后起另外一个 loaddex 进程去加载 Dex。等 loaddex 进程加载完后，我们再通知主进程继续往下走。



现在转化为两个问题：

- a. 通过何种方式挂起主进程？
- b. 挂住主进程中，是否会产生ANR？

进程同步可以使用 `pthread_mutex_xxx`、`pthread_cond_xxx`，但是 `mutex` 或 `cond` 要放于共享内存中，过于复杂。或者由于在主进程访问远端 service，也是同步的，这应该也是一种不错的方法。但是我最后测试时采用的是一个最简单的方法，即检测到没有加载 dex 时，在 `com.tencent.mm` 下新建一个临时文件，每隔100ms去询问文件是否存在。而在 `loaddex` 结束后，即主动的删除该文件。



WeMobileDev

那会不会出现ANR呢？事实上是不会的，因为主进程已经不是前台进程了，经过测试，在 `attachBaseContext`，无论将要启动的 `Activity`、`Broadcast` 还是 `Service`，尽管卡住 100s，也不会出现ANR(回想ANR的几个原因，按键消息、`Broadcast onReceiver` 或者 `Service`)。

总的来说，这种方式好处在于依赖集非常简单，同时它的集成方式也是非常简单，我们无须去修改与加载无关的代码。但是没有经过广泛的测试，欢迎大家交流或指正方案中的缺陷。

## 总结

对 `dex` 以及 `art` 研究过一段时间，不少细节还是要注意的。列一下微信现在的 `ToDo List`：

1. 使用 `classes(..N).dex` 形式；
2. 论证第四种方案是否可行，想到一个问题就是如果起得是一个 `Service` 弹出界面，似乎有点唐突，但是这个可以通过判断当前 `TopActivity` 方式解决。 ...

以上都是个人的理解，可能有错误或纰漏的地方，欢迎大家指正与技术交流。



# CodeBoy微信抢红包外挂

来源:[www.happycodeboy.com](http://www.happycodeboy.com)



源码下载地址:<https://github.com/lendylongli/qianghongbao>

apk下载地址：[百度云下载](#) [本地下载](#)

## 前言

Codeboy微信抢红包是我在2015年春节过年期间编写的一个开源的兴趣项目，只要是将整个核心抢红包的流程编写出来，至于再复杂的一些操作就没深入研究。当这个项目发布后，也是反应挺大的，很多网友也找到我了与交流，也有做淘宝的人给钱让我去增加一些功能，当然我是拒绝的。

## 作者声明

在这里，我声明一下，我所做的是自己有兴趣的事情，只是通过开源的方式让大家去学习相关技术，并不是为了营利，而我也知道淘宝上有人直接拿我的应用去售卖，这些都是没经过我的允许，我也没有半点收益，我留下联系方式是为了方便开发者之间的讨论与学习，所以请商业合作的与小白不要加我QQ，谢谢。

## 技术详述

一开始大家都会觉得做一个Android外挂会汲取很多东西或者底层的东西，但当发现Android里有一个叫 `AccessibilityService` 的服务时，一切都变得很简单。

### 关于`AccessibilityService`

- 先看看官网的介绍`Accessibility`

Many Android users have different abilities that require them to interact with their Android devices in different ways. These include users who have visual, physical or age-related limitations that prevent them from fully seeing or using a touchscreen, and users with hearing loss who may not be able to perceive audible information and alerts...

#### [Android官网详解accessibility](#)

上面大概的意思就是`Accessibility`是一个辅助服务，主要是面向一些使用Android手机的用户有相关障碍(年龄、视觉、听力、身体等)，这个功能可以更容易使用手机，可以帮用户在点击屏幕或者显示方面得到帮助等等。接下来就是查找相关API，看能做到哪个地步。

#### [Accessibility相关API描述](#)

当然`accessibility`除了可以辅助点击界面的事件外，还可以用作自动化测试，或者一键返回，是一个非常强大与实用的功能，具体实例可以看我另一个 [App虚拟按键助手](#) 请往下载 [GooglePlay市场](#) 或 [应用宝](#)。

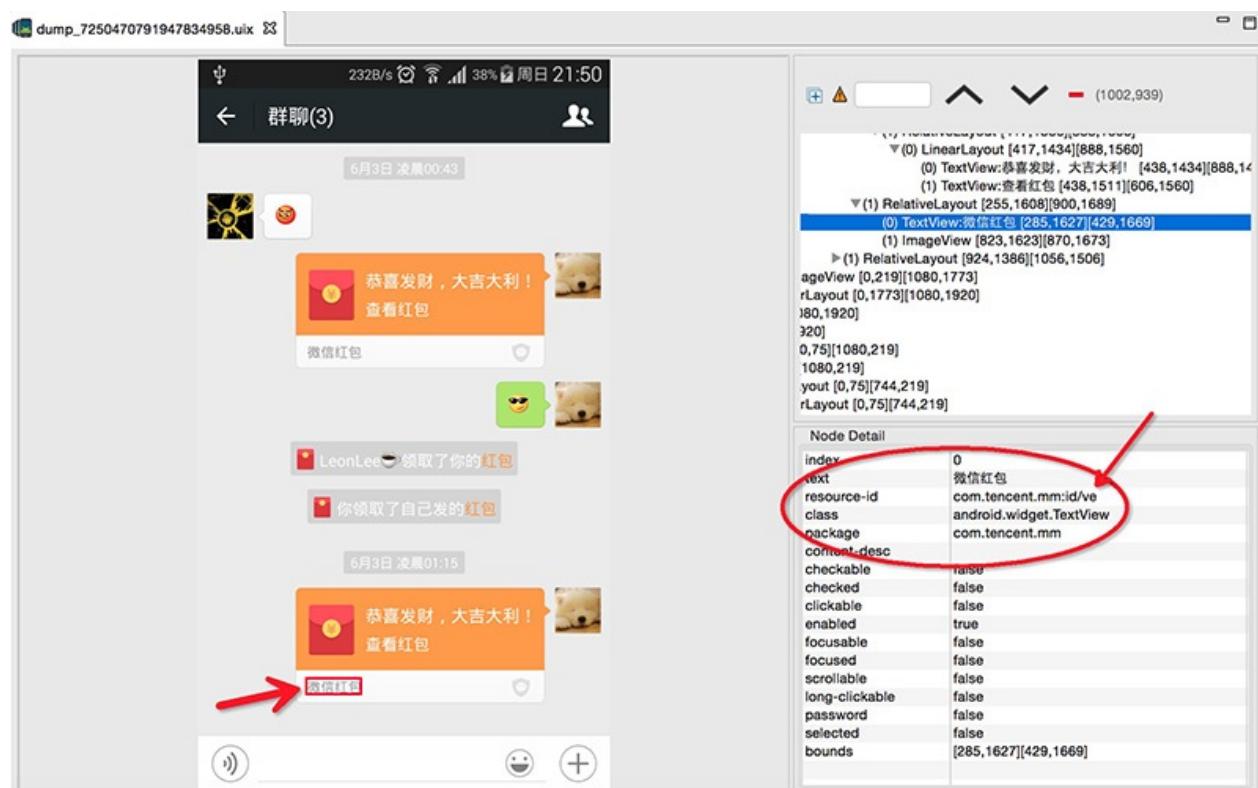
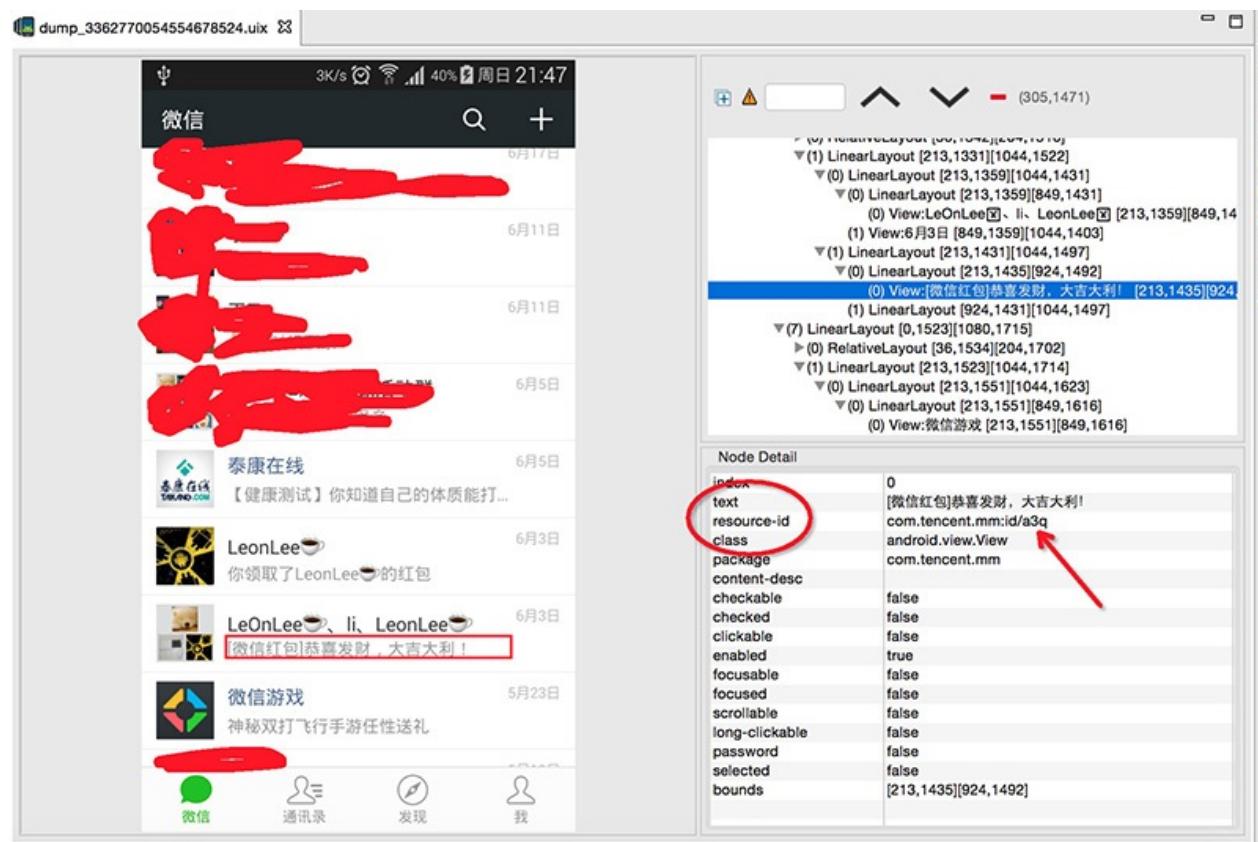
# 关于抢红包的流程

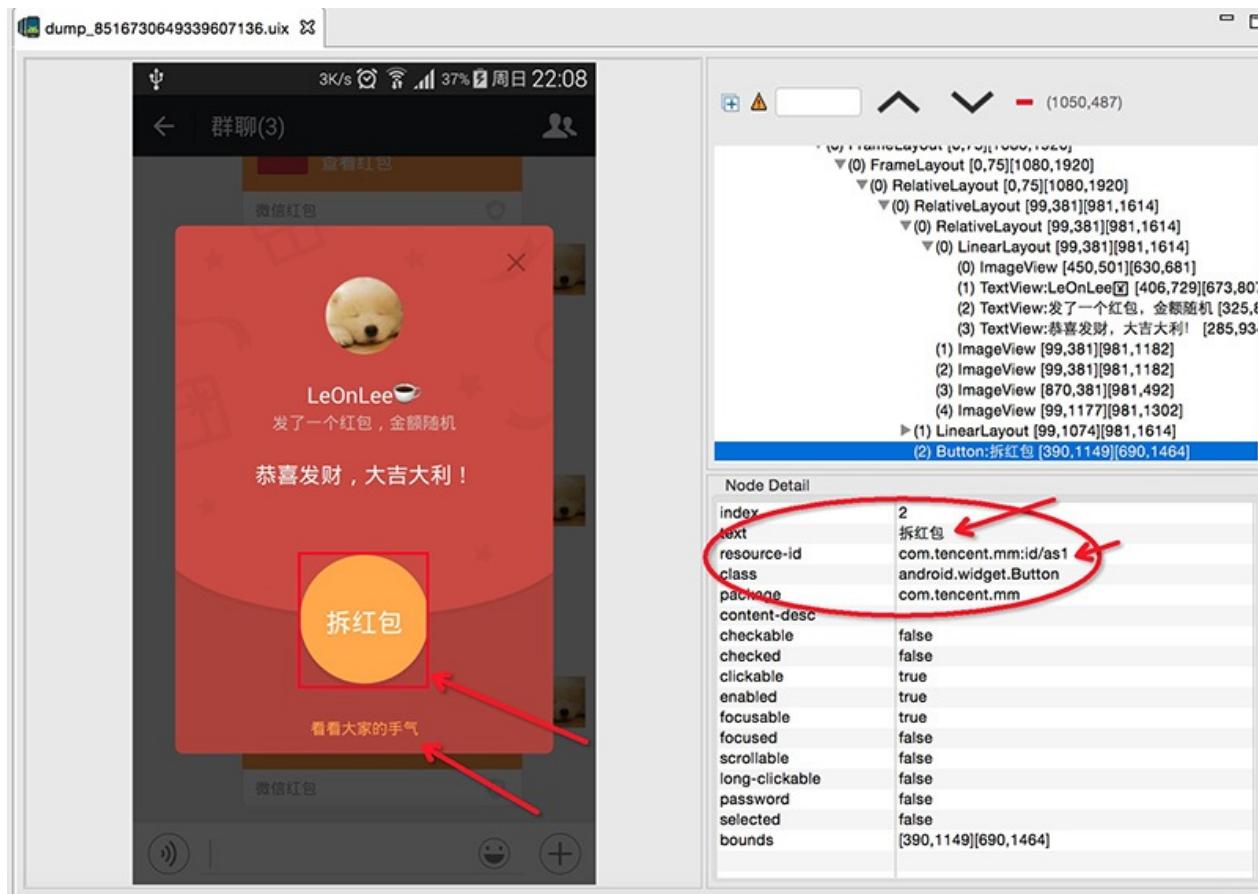
在有以上的一些关于辅助服务的基础知识后，我们就可以分析怎样自动化抢红包。大家使用过微信都知道，如果不是在微信的可见界面范围（在桌面或者在使用其它应用时），在收到新的消息，就会在通知栏提醒用户。而在微信的消息列表界面，就不会弹出通知栏，所以可以区分这两种情况。然后抓取相关关键字作进一步处理。

- 1、在非微信消息列表界面，收到通知消息的事件，判断通知栏里的文本是否有[微信红包]的关键字，有则可以判断为用户收到红包的消息(当然，你可以故意发一条包括这个关键字的文本消息去整蛊你的朋友)。然后，我们就自动化触发这个消息的意图事件(Intent);
- 2、在通知栏跳进微信界面后，是去到 `com.tencent.mm.ui.LauncherUI` 这个Activity 界面。我们知道，红包的消息上，包括了关键字领取红包或者View的id，那我们就根据这个关键字找到相应的View，然后再触发ACTION\_CLICK(点击事件);
- 3、在点击红包后，会跳到  
`com.tencent.mm.plugin.luckymoney.ui.LuckyMoneyReceiveUI` 这个拆红包的Activity,当然老方法，找关键字拆红包或id,然后触发自动化点击事件。

这样就可以完成整个自动化完成抢红包的流程了,所以核心就是找关键字，然后模拟用户点击事件，就这么简单。以下详细说一下代码的实现。

以下是通过DDMS工具里的 `Dump View Hierarchy For UI Automator` 去分析微信UI结构。





## 使用AccessibilityService去一步步监听微信的动作

- 1、新建一个继承 AccessibilityService 的类,如 QiangHongBaoService , 然后在 `AndroidManifest.xml` 里声明组件, 如下

```
<service
 android:label="@string/app_name"
 android:name=".QiangHongBaoService"
 android:permission="android.permission.BIND_ACCESSIBILITY_SERVICE">
 <intent-filter>
 <action android:name="android.accessibilityservice.AccessibilityService"/>
 </intent-filter>
 <meta-data
 android:name="android.accessibilityservice"
 android:resource="@xml/qianghongbao_service_config"/>
</service>
```

在 `meta-data` 里声明的是辅助配置, 这个是 `Android4.0` 之后才支持的写法, 在 `4.0` 之前的系统要在代码里声明。

- 2、在 `res/xml` 目录下生成辅助服务的配置文件 `qianghongbao_service_config.xml`

```
<accessibility-service
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:description="@string/accessibility_description"
 android:accessibilityEventTypes="typeNotificationStateChanged|typeWindowStateChanged"
 android:packageNames="com.tencent.mm"
 android:accessibilityFeedbackType="feedbackGeneric"
 android:notificationTimeout="100"
 android:accessibilityFlags=""
 android:canRetrieveWindowContent="true"/>
```

`android:description` 这个是设置服务的描述，在用户授权的界面可以看到。

`android:accessibilityEventTypes` 这个是配置要监听的辅助事件，我们只需要用到 `typeNotificationStateChanged` (通知变化事件)、`typeWindowStateChanged` (界面变化事件)

`android:packageNames` 这个是要监听应用的包名，如果要监听多个应用，则用,去分隔，这里我们只需要监听微信的就可以了

`android:accessibilityFeedbackType` 这个是设置反馈方式

| FeedbackType    | 描述           |
|-----------------|--------------|
| feedbackSpoken  | 语音反馈         |
| feedbackHaptic  | 触感反馈         |
| feedbackAudible | 表示声音(不是语音)反馈 |
| feedbackVisual  | 视觉反馈         |
| feedbackGeneric | 通用反馈         |
| feedbackAllMask | 所有以上的反馈      |

[详细看AccessibilityServiceInfo类文档描述](#)

- 3、在以上都配置好后，我们就可以在 `QiangHongBaoService` 这个服务里进行编码了，要做的就是将整个UI跳转流程与逻辑串联起来。

```
@Override
public void onAccessibilityEvent(AccessibilityEvent event) {
 //接收事件,如触发了通知栏变化、界面变化等
}

@Override
protected boolean onKeyDown(KeyEvent event) {
 //接收按键事件
 return super.onKeyDown(event);
}

@Override
public void onInterrupt() {
 //服务中断,如授权关闭或者将服务杀死
}

@Override
protected void onServiceConnected() {
 super.onServiceConnected();
 //连接服务后,一般是在授权成功后会接收到
}
```

具体内容请看源码

## 其它

### 如何防止外挂

在了解整个核心后,获取事件不外乎就是通过文本与id判断,那么就可以将文本改为图标方式,将id改为动态id(每次显示都是随机生成),这样一来就可以提高外挂的门槛。

如何发红包会安全点

现在抢红包就看谁的外挂工具反应够快,如何去干扰这些外挂,其实也有点小技巧,就是在发红包前,发送文本[微信红包],可以导致部分外挂工具失效。

# dexopt的源码跟踪

来源:[zongwu233.github.io](https://github.com/zongwu233/dexopt)

关于dexopt的预备知识在[这里](#)。

一般都是从 `AndroidStudio run` 出来的apk安装到手机，这里从 `adb shell pm install -r xxx.apk` 开始看起：

- 找到 `android/frameworks/base/cmds/pm/src/com/android/commands/pm/pm.java` `pm` 命令的源码
- `main` 函数调用了 `new Pm().run(args);`
- 找到 `runInstall()` 方法  
    |    哦刚才的常用参数 `-r` 表示覆盖安装模式 `INSTALL_REPLACE_EXISTING`
- `runInstall()` 方法最终会调用

```
mPm.installPackageWithVerification(apkURI, obs, installFlags,
 installerPackageName, verificationURI, null, encryptionParams);
```

而 `mPm = IPackageManager.Stub.asInterface(ServiceManager.getService("package"))`; 就是 `PackageManagerService`

`PackageManagerService` 的 `main` 方法：

```
public static final IPackageManager main(Context context, Installer installer,
 boolean factoryTest, boolean onlyCore) {
 PackageManagerService m = new PackageManagerService(context, installer,
 factoryTest, onlyCore);
 ServiceManager.addService("package", m);
 return m;
}
```

去看 `installPackageWithVerification()`

```

@Override
public void installPackageWithVerification(Uri packageURI,
 IPackageInstallObserver observer,
 int flags,
 String installerPackageName,
 Uri verificationURI,
 ManifestDigest manifestDigest,
 ContainerEncryptionParams encryptionParams) {

 VerificationParams verificationParams = new VerificationParams(
 verificationURI,
 null,
 null,
 VerificationParams.NO_UID,
 manifestDigest);

 installPackageWithVerificationAndEncryption(
 packageURI,
 observer,
 flags,
 installerPackageName,
 verificationParams,
 encryptionParams);
}

```

## 接着看 `installPackageWithVerificationAndEncryption()` 方法

```

public void installPackageWithVerificationAndEncryption(
 Uri packageURI,
 IPackageInstallObserver observer,
 int flags,
 String installerPackageName,
 VerificationParams verificationParams,
 ContainerEncryptionParams encryptionParams)

```

在最后的时候：

```

final Message msg = mHandler.obtainMessage(INIT_COPY);
msg.obj = new InstallParams(packageURI, observer,
 filteredFlags, installerPackageName,
 verificationParams, encryptionParams, user);
mHandler.sendMessage(msg);

```

发messge 给handler，找到这个handler的定义

```
mHandlerThread.start();
mHandler = new PackageHandler(mHandlerThread.getLooper());
final HandlerThread mHandlerThread = new HandlerThread("PackageManager",
 Process.THREAD_PRIORITY_BACKGROUND);
final PackageHandler mHandler;
```

看来是个工作线程。

跟进 `handler` 的 `handlemsg()` 消息类型会从 `INIT_COPY` 绕成 `MCS_BOUND`。总算调用了 `params.startCopy()` 先看 `handleStartCopy()`。`InstallParams` 对该方法的实现。略过各种检查和校验 找到了 `ret = args.copyApk(mContainerService, true);` `copy apk文件到 /data/app 目录下 而且是以一种临时文件的格式。`

```
final boolean startCopy() {
 boolean res;
 try {
 if (DEBUG_INSTALL) Slog.i(TAG, "startCopy");
 if (++mRetries > MAX_RETRIES) {
 Slog.w(TAG, "Failed to invoke remote methods on default container service");
 mHandler.sendEmptyMessage(MCS_GIVE_UP);
 handleServiceError();
 return false;
 } else {
 handleStartCopy();
 res = true;
 }
 } catch (RemoteException e) {
 if (DEBUG_INSTALL) Slog.i(TAG, "Posting install MCS_RECONNECT");
 mHandler.sendEmptyMessage(MCS_RECONNECT);
 res = false;
 }
 handleReturnCode();
 return res;
}
```

然后是 `handleReturnCode()` 主要是:

```
private void processPendingInstall(final InstallArgs args, final int currentStatus)
```

该方法调用

```
private void installNewPackageLI(PackageParser.Package pkg,
 int parseFlags, int scanMode, UserHandle user,
 String installerPackageName, PackageInstalledInfo res)
```

终于到了 `installNewPackage` 了

```
private void installNewPackageLI(PackageParser.Package pkg,
 int parseFlags, int scanMode, UserHandle user,
 String installerPackageName, PackageInstalledInfo res)

private PackageParser.Package scanPackageLI(File scanFile,
 int parseFlags, int scanMode, long currentTime, UserHandle user)
```

然后调用了

```
// Note that we invoke the following method only if we are about to unpack an application
PackageParser.Package scannedPkg = scanPackageLI(pkg, parseFlags, scanMode
 | SCAN_UPDATE_SIGNATURE, currentTime, user);
```

其中有

```
//invoke installer to do the actual installation
int ret = createDataDirsLI(pkgName, pkg.applicationInfo.uid);

if((scanMode & SCAN_NO_DEX) == 0) {
 if(performDexOptLI(pkg, forceDex, (scanMode & SCAN_DEFER_DEX) != 0) == DEX_OPT_FAILED) {
 mLastScanError = PackageManager.INSTALL_FAILED_DEXOPT;
 return null;
 }
}
```

先看 `scanPackageLI`

```
private int createDataDirsLI(String packageName, int uid) {
 int[] users = sUserManager.getUserIds();
 int res = mInstaller.install(packageName, uid, uid);
 if(res < 0) {
 return res;
 }
 for(int user: users) {
 if(user != 0) {
 res = mInstaller.createUserData(packageName,
 UserHandle.getUid(user, uid), user);
 if(res < 0) {
 return res;
 }
 }
 }
 return res;
}
```

mInstaller.install() 出现了,再看 performDexOptLI()

```

private int performDexOptLI(PackageParser.Package pkg, boolean forceDex, boolean defer)
 boolean performed = false;
 if((pkg.applicationInfo.flags & ApplicationInfo.FLAG_HAS_CODE) != 0) {
 String path = pkg.mScanPath;
 int ret = 0;
 try {
 if(forceDex || dalvik.system.DexFile.isDexOptNeeded(path)) {
 if(! forceDex && defer) {
 mDeferredDexOpt.add(pkg);
 return DEX_OPT_DEFERRED;
 } else {
 Log.i(TAG, "Running dexopt on: " + pkg.applicationInfo.packageName);
 final int sharedGid = UserHandle.getSharedAppGid(pkg.applicationInfo);
 ret = mInstaller.dexopt(path, sharedGid, ! isForwardLocked(pkg));
 pkg.mDidDexOpt = true;
 performed = true;
 }
 }
 } catch(FileNotFoundException e) {
 Slog.w(TAG, "Apk not found for dexopt: " + path);
 ret = - 1;
 } catch(IOException e) {
 Slog.w(TAG, "IOException reading apk: " + path, e);
 ret = - 1;
 } catch(dalvik.system.StaleDexCacheError e) {
 Slog.w(TAG, "StaleDexCacheError when reading apk: " + path, e);
 ret = - 1;
 } catch(Exception e) {
 Slog.w(TAG, "Exception when doing dexopt : ", e);
 ret = - 1;
 }
 if(ret < 0) {
 // error from installer
 return DEX_OPT_FAILED;
 }
 }
 return performed ? DEX_OPT_PERFORMED : DEX_OPT_SKIPPED;
}

```

`mInstaller.dexopt()` 终于也找出来了

PackageManagerService 通过 socket 访问 installd 的服务进程，而 installd 的服务是在 init 进程里被启动的。继续看 frameworks/base/cmds/installd/installd.c

```

int main(const int argc, const char *argv[]) {
 char buf[BUFFER_MAX];
 struct sockaddr addr;
 socklen_t alen;

```

```
int lsocket, s, count;
if (initialize_globals() < 0) {
 ALOGE("Could not initialize globals; exiting.\n");
 exit(1);
}
if (initialize_directories() < 0) {
 ALOGE("Could not create directories; exiting.\n");
 exit(1);
}
lsocket = android_get_control_socket(SOCKET_PATH);
if (lsocket < 0) {
 ALOGE("Failed to get socket from environment: %s\n", strerror(errno));
 exit(1);
}
if (listen(lsocket, 5)) {
 ALOGE("Listen on socket failed: %s\n", strerror(errno));
 exit(1);
}
fcntl(lsocket, F_SETFD, FD_CLOEXEC);
for (;;) {
 alen = sizeof(addr);
 s = accept(lsocket, &addr, &alen);
 if (s < 0) {
 ALOGE("Accept failed: %s\n", strerror(errno));
 continue;
 }
 fcntl(s, F_SETFD, FD_CLOEXEC);
 ALOGI("new connection\n");
 for (;;) {
 unsigned short count;
 if (readx(s, &count, sizeof(count))) {
 ALOGE("failed to read size\n");
 break;
 }
 if ((count < 1) || (count >= BUFFER_MAX)) {
 ALOGE("invalid size %d\n", count);
 break;
 }
 if (readx(s, buf, count)) {
 ALOGE("failed to read command\n");
 break;
 }
 buf[count] = 0;
 if (execute(s, buf))
 break;
 }
 ALOGI("closing connection\n");
 close(s);
}
return 0;
}
```

接收的命令：

```
struct cmdinfo {
 const char *name;
 unsigned numargs;
 int (*func)(char **arg, char reply[REPLY_MAX]);
};

struct cmdinfo cmds[] = { { "ping", 0, do_ping },
 { "install", 3, do_install },
 { "dexopt", 3, do_dexopt },
 { "movedex", 2, do_move_dex },
 { "rmdex", 1, do_rm_dex },
 { "remove", 2, do_remove },
 { "rename", 2, do_rename },
 { "freecache", 1, do_free_cache },
 { "rmcache", 1, do_rm_cache },
 { "protect", 2, do_protect },
 { "getsize", 4, do_get_size },
 { "rmuserdata", 2, do_rm_user_data },
 {"movefiles", 0, do_movefiles },
 { "linklib", 2, do_linklib },
 {"unlinklib", 1, do_unlinklib },
 { "mkuserdata", 3, do_mk_user_data },
 { "rmuser", 1, do_rm_user },
 {"cloneuserdata", 3, do_clone_user_data }, };
```

比如

```
static int do_install(char **arg, char reply[REPLY_MAX]) {
 /* pkgname, uid, gid */
 return install(arg[0], atoi(arg[1]), atoi(arg[2]));
}
static int do_dexopt(char **arg, char reply[REPLY_MAX]) {
 /* apk_path, uid, is_public */
 return dexopt(arg[0], atoi(arg[1]), atoi(arg[2]));
}
```

frameworks/base/cmds/installd/commands.c

```

int install(const char *pkgname, uid_t uid, gid_t gid) {
 char pkgdir[PKG_PATH_MAX];
 char libdir[PKG_PATH_MAX];
 if ((uid < AID_SYSTEM) || (gid < AID_SYSTEM)) {
 ALOGE("invalid uid/gid: %d %d\n", uid, gid);
 return -1;
 }
 if (create_pkg_path(pkgdir, pkgname, PKG_DIR_POSTFIX, 0)) {
 ALOGE("cannot create package path\n");
 return -1;
 }
 if (create_pkg_path(libdir, pkgname, PKG_LIB_POSTFIX, 0)) {
 ALOGE("cannot create package lib path\n");
 return -1;
 }
 if (mkdir(pkgdir, 0751) < 0) {
 ALOGE("cannot create dir '%s': %s\n", pkgdir, strerror(errno));
 return -errno;
 }
 if (chmod(pkgdir, 0751) < 0) {
 ALOGE("cannot chmod dir '%s': %s\n", pkgdir, strerror(errno));
 unlink(pkgdir);
 return -errno;
 }
 if (chown(pkgdir, uid, gid) < 0) {
 ALOGE("cannot chown dir '%s': %s\n", pkgdir, strerror(errno));
 unlink(pkgdir);
 return -errno;
 }
 if (mkdir(libdir, 0755) < 0) {
 ALOGE("cannot create dir '%s': %s\n", libdir, strerror(errno));
 unlink(pkgdir);
 return -errno;
 }
 if (chmod(libdir, 0755) < 0) {
 ALOGE("cannot chmod dir '%s': %s\n", libdir, strerror(errno));
 unlink(libdir);
 unlink(pkgdir);
 return -errno;
 }
 if (chown(libdir, AID_SYSTEM, AID_SYSTEM) < 0) {
 ALOGE("cannot chown dir '%s': %s\n", libdir, strerror(errno));
 unlink(libdir);
 unlink(pkgdir);
 return -errno;
 }
 return 0;
}

```

·adb pm -r install xxx.apk`的流程跟踪结束。

再来看看 `MultiDex.install()` 的流程：

```
android.support.multidex.MultiDex.V14#install()
 android.support.multidex.MultiDex.V14#makeDexElements();
```

通过反射调用了 `DexPathList` 的 `makeDexElements()`，该方法就是调用 `dex = loadDexFile(file, optimizedDirectory);`

看看 `loadDexFile`

```
/**
 * Constructs a {@code DexFile} instance, as appropriate depending
 * on whether {@code optimizedDirectory} is {@code null}.
 */
private static DexFile loadDexFile(File file,
 File optimizedDirectory) throws IOException {
 if(optimizedDirectory == null) {
 return new DexFile(file);
 } else {
 String optimizedPath = optimizedPathFor(file, optimizedDirectory);
 return DexFile.loadDex(file.getPath(), optimizedPath, 0);
 }
}
```

第一次 `load dex` 文件的时候是 `optimizedDirectory` 空的。进入 `return new DexFile(file);` 的分支。

```
/*
 * Open a DEX file. The value returned is a magic VM cookie. On
 * failure, an IOException is thrown.
 */
native private static int openDexFile(String sourceName, String outputName,
 int flags) throws IOException;
```

进入 native 层代码 `openDexFile()`

```
static void Dalvik_dalvik_system_DexFile_openDexFile(const u4* args,
 JValue* pResult)
```

该方法比较长主要是调用了 `dvmJarFileOpen()` 方法，在 `android/dalvik/vm/JarFile.cpp`

```
/*
 * Open a Jar file. It's okay if it's just a Zip archive without all of
 * the Jar trimmings, but we do insist on finding "classes.dex" inside
 * or an appropriately-named ".odex" file alongside.
 */
```

```

 * If "isBootstrap" is not set, the optimizer/verifier regards this DEX as
 * being part of a different class loader.
 */
int dvmJarFileOpen(const char* fileName, const char* odexOutputName,
 JarFile** ppJarFile, bool isBootstrap) {
/*
 * TODO: This function has been duplicated and modified to become
 * dvmRawDexFileOpen() in RawDexFile.c. This should be refactored.
 */
ZipArchive archive;
DvmDex* pDvmDex = NULL;
char* cachedName = NULL;
bool archiveOpen = false;
bool locked = false;
int fd = -1;
int result = -1;
/* Even if we're not going to look at the archive, we need to
 * open it so we can stuff it into ppJarFile.
 */
if (dexZipOpenArchive(fileName, &archive) != 0)
 goto bail;
archiveOpen = true;
/* If we fork/exec into dexopt, don't let it inherit the archive's fd.
 */
dvmSetCloseOnExec(dexZipGetArchiveFd(&archive));
/* First, look for a ".odex" alongside the jar file. It will
 * have the same name/path except for the extension.
 */
fd = openAlternateSuffix(fileName, "odex", O_RDONLY, &cachedName);
if (fd >= 0) {
 ALOGV("Using alternate file (odex) for %s ...", fileName);
 if (!dvmCheckOptHeaderAndDependencies(fd, false, 0, 0, true, true)) {
 ALOGE("%s odex has stale dependencies", fileName);
 free(cachedName);
 cachedName = NULL;
 close(fd);
 fd = -1;
 goto tryArchive;
 } else {
 ALOGV("%s odex has good dependencies", fileName);
 //TODO: make sure that the .odex actually corresponds
 // to the classes.dex inside the archive (if present).
 // For typical use there will be no classes.dex.
 }
} else {
 ZipEntry entry;
 tryArchive:
 /*
 * Pre-created .odex absent or stale. Look inside the jar for a
 * "classes.dex".
 */
 entry = dexZipFindEntry(&archive, kDexInJarName);
}
}

```

```

if (entry != NULL) {
 bool newFile = false;
 /*
 * We've found the one we want. See if there's an up-to-date copy
 * in the cache.
 *
 * On return, "fd" will be seeked just past the "opt" header.
 *
 * If a stale .odex file is present and classes.dex exists in
 * the archive, this will *not* return an fd pointing to the
 * .odex file; the fd will point into dalvik-cache like any
 * other jar.
 */
 if (odexOutputName == NULL) {
 cachedName = dexOptGenerateCacheFileName(fileName,
 kDexInJarName);
 if (cachedName == NULL)
 goto bail;
 } else {
 cachedName = strdup(odexOutputName);
 }
 ALOGV("dvmJarFileOpen: Checking cache for %s (%s)", fileName,
 cachedName);
 fd = dvmOpenCachedDexFile(fileName, cachedName,
 dexGetZipEntryModTime(&archive, entry),
 dexGetZipEntryCrc32(&archive, entry), isBootstrap, &newFile, /*cr
 true);
 if (fd < 0) {
 ALOGI("Unable to open or create cache for %s (%s)", fileName,
 cachedName);
 goto bail;
 }
 locked = true;
 /*
 * If fd points to a new file (because there was no cached version,
 * or the cached version was stale), generate the optimized DEX.
 * The file descriptor returned is still locked, and is positioned
 * just past the optimization header.
 */
 if (newFile) {
 u8 startWhen, extractWhen, endWhen;
 bool result;
 off_t dexOffset;
 dexOffset = lseek(fd, 0, SEEK_CUR);
 result = (dexOffset > 0);
 if (result) {
 startWhen = dvmGetRelativeTimeUsec();
 result = dexZipExtractEntryToFile(&archive, entry, fd) == 0;
 extractWhen = dvmGetRelativeTimeUsec();
 }
 if (result) {
 result = dvmOptimizeDexFile(fd, dexOffset,

```

```

 dexGetZipEntryUncomplLen(&archive, entry), fileName,
 dexGetZipEntryModTime(&archive, entry),
 dexGetZipEntryCrc32(&archive, entry), isBootstrap);
 }
 if (!result) {
 ALOGE("Unable to extract+optimize DEX from '%s'", fileName);
 goto bail;
 }
 endWhen = dvmGetRelativeTimeUsec();
 ALOGD("DEX prep '%s': unzip in %dms, rewrite %dms", fileName,
 (int) (extractWhen - startWhen) / 1000,
 (int) (endWhen - extractWhen) / 1000);
}
} else {
 ALOGI("Zip is good, but no %s inside, and no valid .odex "
 "file in the same directory", kDexInJarName);
 goto bail;
}
/*
 * Map the cached version. This immediately rewinds the fd, so it
 * doesn't have to be seeked anywhere in particular.
 */
if (dvmDexFileOpenFromFd(fd, &pDvmDex) != 0) {
 ALOGI("Unable to map %s in %s", kDexInJarName, fileName);
 goto bail;
}
if (locked) {
 /* unlock the fd */
 if (!dvmUnlockCachedDexFile(fd)) {
 /* uh oh -- this process needs to exit or we'll wedge the system */
 ALOGE("Unable to unlock DEX file");
 goto bail;
 }
 locked = false;
}
ALOGV("Successfully opened '%s' in '%s'", kDexInJarName, fileName);
ppJarFile = (JarFile) calloc(1, sizeof(JarFile));
(*ppJarFile)->archive = archive;
(*ppJarFile)->cacheFileName = cachedName;
(*ppJarFile)->pDvmDex = pDvmDex;
cachedName = NULL; // don't free it below
result = 0;
bail:
/* clean up, closing the open file */
if (archiveOpen && result != 0)
 dexZipCloseArchive(&archive);
free(cachedName);
if (fd >= 0) {
 if (locked)
 (void) dvmUnlockCachedDexFile(fd);
 close(fd);
}

```

```

 }
 return result;
}

```

其中，执行了 `dvmOpenCachedDexFile()` 后 `newFile` 为 `true`。会执行

```

result = dvmOptimizeDexFile(fd, dexOffset,
 dexGetZipEntryUncompLen(&archive, entry),
 fileName,
 dexGetZipEntryModTime(&archive, entry),
 dexGetZipEntryCrc32(&archive, entry),
 isBootstrap);

```

这里调用了 `/dalvik/vm/RawDexFile.cpp` 中的 `dvmOptimizeDexFile()` 方法

```

/*
 * Given a descriptor for a file with DEX data in it, produce an
 * optimized version.
 *
 * The file pointed to by "fd" is expected to be a locked shared resource
 * (or private); we make no efforts to enforce multi-process correctness
 * here.
 *
 * "fileName" is only used for debug output. "modWhen" and "crc" are stored
 * in the dependency set.
 *
 * The "isBootstrap" flag determines how the optimizer and verifier handle
 * package-scope access checks. When optimizing, we only load the bootstrap
 * class DEX files and the target DEX, so the flag determines whether the
 * target DEX classes are given a (synthetic) non-NULL classLoader pointer.
 * This only really matters if the target DEX contains classes that claim to
 * be in the same package as bootstrap classes.
 *
 * The optimizer will need to load every class in the target DEX file.
 * This is generally undesirable, so we start a subprocess to do the
 * work and wait for it to complete.
 *
 * Returns "true" on success. All data will have been written to "fd".
 */
bool dvmOptimizeDexFile(int fd, off_t dexOffset, long dexLength,
 const char* fileName, u4 modWhen, u4 crc, bool isBootstrap) {
 const char* lastPart = strrchr(fileName, '/');
 if (lastPart != NULL)
 lastPart++;
 else
 lastPart = fileName;
 ALOGD("DexOpt: --- BEGIN '%s' (bootstrap=%d) ---", lastPart, isBootstrap);
}

```

```

pid_t pid;
/*
 * This could happen if something in our bootclasspath, which we thought
 * was all optimized, got rejected.
 */
if (gDvm.optimizing) {
 ALOGW("Rejecting recursive optimization attempt on '%s'", fileName);
 return false;
}
pid = fork();
if (pid == 0) {
 static const int kUseValgrind = 0;
 static const char* kDexOptBin = "/bin/dexopt";
 static const char* kValgrinder = "/usr/bin/valgrind";
 static const int kFixedArgCount = 10;
 static const int kValgrindArgCount = 5;
 static const int kMaxIntLen = 12; // '-'+10dig+'\0' -OR- 0x+8dig
 int bcpSize = dvmGetBootPathSize();
 int argc = kFixedArgCount + bcpSize
 + (kValgrindArgCount * kUseValgrind);
 const char* argv[argc + 1]; // last entry is NULL
 char values[argc][kMaxIntLen];
 char* execFile;
 const char* androidRoot;
 int flags;
 /* change process groups, so we don't clash with ProcessManager */
 setpgid(0, 0);
 /* full path to optimizer */
 androidRoot = getenv("ANDROID_ROOT");
 if (androidRoot == NULL) {
 ALOGW("ANDROID_ROOT not set, defaulting to /system");
 androidRoot = "/system";
 }
 execFile = (char*) alloca(strlen(androidRoot) + strlen(kDexOptBin) + 1);
 strcpy(execFile, androidRoot);
 strcat(execFile, kDexOptBin);
 /*
 * Create arg vector.
 */
 int curArg = 0;
 if (kUseValgrind) {
 /* probably shouldn't ship the hard-coded path */
 argv[curArg++] = (char*) kValgrinder;
 argv[curArg++] = "--tool=memcheck";
 argv[curArg++] = "--leak-check=yes"; // check for leaks too
 argv[curArg++] = "--leak-resolution=med"; // increase from 2 to 4
 argv[curArg++] = "--num-callers=16"; // default is 12
 assert(curArg == kValgrindArgCount);
 }
 argv[curArg++] = execFile;
 argv[curArg++] = "--dex";
 sprintf(values[2], "%d", DALVIK_VM_BUILD);
}

```

```

 argv[curArg++] = values[2];
 sprintf(values[3], "%d", fd);
 argv[curArg++] = values[3];
 sprintf(values[4], "%d", (int) dexOffset);
 argv[curArg++] = values[4];
 sprintf(values[5], "%d", (int) dexLength);
 argv[curArg++] = values[5];
 argv[curArg++] = (char*) fileName;
 sprintf(values[7], "%d", (int) modWhen);
 argv[curArg++] = values[7];
 sprintf(values[8], "%d", (int) crc);
 argv[curArg++] = values[8];
 flags = 0;
 if (gDvm.dexOptMode != OPTIMIZE_MODE_NONE) {
 flags |= DEXOPT_OPT_ENABLED;
 if (gDvm.dexOptMode == OPTIMIZE_MODE_ALL)
 flags |= DEXOPT_OPT_ALL;
 }
 if (gDvm.classVerifyMode != VERIFY_MODE_NONE) {
 flags |= DEXOPT_VERIFY_ENABLED;
 if (gDvm.classVerifyMode == VERIFY_MODE_ALL)
 flags |= DEXOPT_VERIFY_ALL;
 }
 if (isBootstrap)
 flags |= DEXOPT_IS_BOOTSTRAP;
 if (gDvm.generateRegisterMaps)
 flags |= DEXOPT_GEN_REGISTER_MAPS;
 sprintf(values[9], "%d", flags);
 argv[curArg++] = values[9];
 assert(
 ((!kUseValgrind && curArg == kFixedArgCount)
 || ((kUseValgrind
 && curArg == kFixedArgCount + kValgrindArgCount))));

 ClassPathEntry* cpe;
 for (cpe = gDvm.bootClassPath; cpe->ptr != NULL; cpe++) {
 argv[curArg++] = cpe->fileName;
 }
 assert(curArg == argc);
 argv[curArg] = NULL;
 if (kUseValgrind)
 execv(kValgrinder, const_cast<char**>(argv));
 else
 execv(execFile, const_cast<char**>(argv));
 ALOGE("execv '%s'%s failed: %s", execFile,
 kUseValgrind ? " [valgrind]" : "", strerror(errno));
 exit(1);
} else {
 ALOGV("DexOpt: waiting for verify+opt, pid=%d", (int) pid);
 int status;
 pid_t gotPid;
 /*
 * Wait for the optimization process to finish. We go into VMWAIT

```

```

 * mode here so GC suspension won't have to wait for us.
 */
ThreadStatus oldStatus = dvmChangeStatus(NULL, THREAD_VMWAIT);
while (true) {
 gotPid = waitpid(pid, &status, 0);
 if (gotPid == -1 && errno == EINTR) {
 ALOGD("waitpid interrupted, retrying");
 } else {
 break;
 }
}
dvmChangeStatus(NULL, oldStatus);
if (gotPid != pid) {
 ALOGE("waitpid failed: wanted %d, got %d: %s", (int) pid,
 (int) gotPid, strerror(errno));
 return false;
}
if (WIFEXITED(status) && WEXITSTATUS(status) == 0) {
 ALOGD("DexOpt: --- END '%s' (success) ---", lastPart);
 return true;
} else {
 ALOGW("DexOpt: --- END '%s' --- status=0x%04x, process failed",
 lastPart, status);
 return false;
}
}
}

```

终于找到 `ALOGD("DexOpt: --- BEGIN '%s' (bootstrap=%d) ---", lastPart, isBootstrap);` 这句日志的出处了！

该方法的注释写的也比较详细了，其中 `pid=fork();` 表明 `dexopt` 确实开启了新的进程来完成，而且当前进程是使用 `while` 循环等待这个子进程返回结果的。

那么之前关于 `dexopt` 的所有疑惑都解开了。

# 其实你不知道MultiDex到底有多坑

来源:[zongwu233.github.io](https://zongwu233.github.io)

## 遭遇MultiDex

愉快地写着Android代码的总悟君往工程里引入了一个默默无闻的jar然后Run了一下， 经过漫长的等待AndroidStudio构建失败了。

于是总悟君带着疑惑查看错误信息。

```
UNEXPECTED TOP-LEVEL EXCEPTION: java.lang.IllegalArgumentException: method ID not in
 at com.android.dx.merge.DexMerger$6.updateIndex(DexMerger.java:501)
 at com.android.dx.merge.DexMerger$IdMerger.mergeSorted(DexMerger.java:276)
 at com.android.dx.merge.DexMerger.mergeMethodIds(DexMerger.java:490)
 at com.android.dx.merge.DexMerger.mergeDexes(DexMerger.java:167)
 at com.android.dx.merge.DexMerger.merge(DexMerger.java:188)
 at com.android.dx.command.dexer.Main.mergeLibraryDexBuffers(Main.java:439)
 at com.android.dx.command.dexer.Main.runMonoDex(Main.java:287)
 at com.android.dx.command.dexer.Main.run(Main.java:230)
 at com.android.dx.command.dexer.Main.main(Main.java:199)
 at com.android.dx.command.Main.main(Main.java:103):Derp:dexDerpDebug FAILED
```

看起来是：在试图将 classes和jar塞进一个Dex文件的过程中产生了错误。

早期的Dex文件保存所有classes的方法个数的范围在0~65535之间。业务一直在增长，总悟君写(copy)的代码越来越长引入的库越来越多，超过这个范围只是时间问题。

怎么解？？太阳底下木有新鲜事，淡定先google一发，找找已经踩过坑的小伙伴。

StackOverflow 的网友们对该问题表示情绪稳定，谈笑间抛出multiDex。

这是Android官网对当初的短视行为给出的[补丁方案](#)。文档说，Dalvik Executable (DEX) 文件的总方法数限制在65536以内，其中包括Android framwork method, lib method (后来总悟君发现仅仅是Android自己的框架的方法就已经占用了1w多)，还有你的 code method，所以请使用MultiDex。对于5.0以下版本，请使用multidex support library (这个是我们的补丁包！build tools 请升级到21)。而5.0及以上版本，由于ART模式的存在，

app第一次安装之后会进行一次预编译(pre-compilation)，如果这时候发现了 classes(..N).dex文件的存在就会将他们最终合成为一个.oat的文件，嗯看起来很厉害的样子。

同时Google建议review代码的直接或者间接依赖，尽可能减少依赖库，设置proguard参数进一步优化去除无用的代码。嗯，这两个实施起来倒是很简单，但是治标不治本，躲得过初一躲不过十五。在Google给出这个解决方案之前，他们的开发人员先给了一个简陋简易版本的multiDex具体参看这里。（怀疑后来的官方解决方案就有这家伙参与）。简单地说就是：1.先把你的app 的class 拆分成主次两个dex。2.你的程序运行起来后，自己把第二个dex给load进来。看就这么简单！而且这就是个动态加载模块的框架！然而总悟君早已看穿Dalvik VM 这种动态加载dex 的能力归根结底还是因为java 的classloader类加载机制。沿着这条道走，Android模块动态化加载，包括dex级别和apk级别的动态化加载，各种玩法层出不穷。参见[这里](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#)。

## 第一回合 天真的官方补丁方案

还是先解决打包问题，回头再研究那些高深的动态化加载技术。偷懒一下咯考虑到投入产出比，决定使用Google官方的multiDex解决。(Google的补丁方案啊，不会再有坑了吧？后面才发现还是太天真) 该方案有两步：

- 1.修改gradle脚本来产生多dex。
- 2.修改manifest 使用MulitDexApplication。

- 步骤1.在gradle脚本里写上：

```

 android {
 compileSdkVersion 21
 buildToolsVersion "21.1.0"

 defaultConfig {
 ...
 minSdkVersion 14
 targetSdkVersion 21
 ...

 // Enabling multidex support.
 multiDexEnabled true
 }
 ...
 }

 dependencies {
 compile 'com.android.support:multidex:1.0.0'
 }
}

```

- 步骤2. manifest声明修改

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 package="com.example.android.multidex.myapplication">
 <application
 ...
 android:name="android.support.multidex.MultiDexApplication">
 ...
 </application>
</manifest>

```

如果有自己的 Application，继承 MultiDexApplication。如果当前代码已经继承自其它 Application 没办法修改那也行，就重写 Application 的 attachBaseContext() 这个方法。

```

@Override
protected void attachBaseContext(Context base) {
 super.attachBaseContext(base);
 MultiDex.install(this);

}

```

run一下，可以了！但是dex过程好像变慢了。。。

文档还写明了multiDex support lib 的局限。瞄一下是什么：

- 1.在应用安装到手机上的时候dex文件的安装是复杂的(complex)有可能会因为第二个dex文件太大导致ANR。请用proguard优化你的代码。呵呵
- 2.使用了mulitDex的App有可能在4.0(api level 14)以前的机器上无法启动，因为Dalvik linearAlloc bug([Issue 22586](#))。请多多测试自祈多福。用proguard优化你的代码将减少该bug几率。呵呵
- 3.使用了mulitDex的App在runtime期间有可能因为Dalvik linearAlloc limit ([Issue 78035](#)) Crash。该内存分配限制在 4.0版本被增大，但是5.0以下的机器上的Apps依然会存在这个限制。
- 4.主dex被dalvik虚拟机执行时候，哪些类必须在主dex文件里面这个问题比较复杂。build tools 可以搞定这个问题。但是如果你代码存在反射和native的调用也不保证100%正确。呵呵

感觉这就是个坑啊。补丁方案又引入一些问题。但是插件化方案要求对现有代码有比较大的改动，代价太大，而且动态化加载框架意味着维护成本更高，会有更多潜在bug。所以先测试，遇到有问题的版本再解决。

## 第二回合 啥？ dexopt failed?

呵呵，部分低端2.3机型(话说2.3版本的android机有高端机型么)安装失败！

`INSTALL_FAILED_DEXOPT`。这个就是前面说的[Issue 22586](#)问题。

apk是一个zip压缩包，dalvik每次加载apk都要从中解压出class.dex文件，加载过程还涉及到dex的classes需要的杂七杂八的依赖库的加载，真耗时间。于是Android决定优化一下这个问题，在app安装到手机之后，系统运行dexopt程序对dex进行优化，将dex的依赖库文件和一些辅助数据打包成odex文件。存放在cache/dalvik\_cache目录下。保存格式为apk路径 @ apk名 @ classes.dex。这样以空间换时间大大缩短读取/加载dex文件的过程。

那刚才那个bug是啥问题呢，原来dexopt程序的dalvik分配一块内存来统计你的app的dex里面的classes的信息，由于classes太多方法太多超过这个linearAlloc 的限制。那减小dex的大小就可以咯。

gradle脚本如下：

```

 android.applicationVariants.all {
 variant ->
 dex.doFirst{
 dex->
 if (dex.additionalParameters == null) {
 dex.additionalParameters = []
 }
 dex.additionalParameters += '--set-max-idx-number=48000'

 }
 }
}

```

--set-max-idx-number= 用于控制每一个dex的最大方法个数，写小一点可以产生好几个dex。踩过更多坑的FB的工程师表示这个linearAlloc的限制不仅仅在安装时候的dexopt程序里<sup>7</sup>，还在你的app的dalvik runtime里。（很显然啊d藻 vm的宿主进程fork自于同一个母体啊）。为了表示对这个坑的不满以及对Google的产品表示遗憾，FB工程师Read The Fucking Source Code找到了一个hack方案。这个linearAlloc的size定义在c层而且是一个全局变量，他们通过对结构体的size的计算成功覆盖了该值的内容，这里要特别感谢C语言的指针和内存的设计。C的世界里，You Are The King of This World。当然实际情况是大部分用户用这把利刃割伤了自己。。。别问总悟君谁是世界上最好的语言。。。

为FB的工程师的机智和务实精神点赞！然而总悟君不愿意花那么多精力实现FB的hack方法。（d藻虚拟机c层代码在2.x 4.x 版本里有变更，找到那个内存地址太难，未必搞得定啊）我们有偷懒的解决方案，为了避免2.3机型runtime 的linearAlloclimit ,最好保持每一个dex体积<4M ,刚才的的value<=48000

好了 现在2.3的机器可以安装run起来了！

## 第三回合 ANR的意思就是Application Not Responding

问题又来了！这次不仅仅是2.3 的机型！还有一些中档配置的4.x系统的机型。问题现象是：第一次安装后，点击图标，1s, 2s, 3s... 程序没有任何反应就好像你没点图标一样。

5s过去。。。程序ANR！

其实不仅仅总悟君的App存在这个问题，其他很多App也存在首次安装运行后几秒都无任何响应的现象或者最后ANR了。唯一的例外是美团App，点击图标立马就出现界面。唉要不就算啦？反正就一次。。。不行，这可是产品给用户的第一印象啊太重要了，而且美团搞得定就说明这问题有解决方案。

ANR了是不是局限1描述的现象？？不过也不重要...因为Google只是告诉你说第二个dex太大了导致的。并没有进一步解释根本原因。怎么办？Google一发？搜索点击图标然后ANR？怎么可能有解决方案嘛。ANR就意味着UI线程被阻塞了，老老实实查看log吧。

```
adb logcat -v time > log.txt
```

于是发现是 `install dex + dexopt` 时间太长！

梳理一下流程：

- 安装完app点击图标之后，系统木有发现对应的process，于是从该apk抽取 `classes.dex` (主dex) 加载，触发一次dexopt。
- App 的laucherActivity准备启动，触发Application启动，
- Application的 `onattach ()` 方法调用，这时候`MultiDex.install ()` 调用，`classes2.dex` 被install，再次触发dexopt。
- 然后Applicaiton `onCreate ()` 执行。
- 然后 launcher Activity真的起来了。

这些必须在5s内完成不然就ANR给你看！

有点棘手。首先主dex是无论如何都绕不过加载和dexopt的。如果主dex比较小的话可以节省时间。主dex小就意味着后面的dex大啊，`MultiDex.install()` 是在主线程里做的，总时间又没有实质性改变。`install()` 能不能放到线程里做啊？貌似不行。。。如果异步化，什么时候install完成都不知道。这时候如果进程需要 `secondary.dex` 里的 `classes` 信息不就悲剧？主dex越小这个错误几率就越大。要悲剧啊总悟君。

淡定，这次Google搜索 `MultiDex.install` 。于是总悟君发现了[美团多dex拆包方案](#)。读完之后感觉看到胜利曙光。美团的主要思路是：**精简主dex+异步加载secondary.dex**。对异步化执行速度的不确定性，他们的解决方案是重写 `Instrumentation execStartActivity` 方法，hook跳转Activity的总入口做判断，如果当前 `secondary.dex` 还没有加载完成，就弹一个 `loading Activity` 等待加载完成，如果已经加载完成那最好不过了。不错，`RTFSC` 果然是王道。可以试一试。

但是有几个问题需要解决：

- 1.分析主dex需要的 `classes` 这个脚本比较难写。。。Google文档说过这个问题比较复杂，而且 `buildTools` 不是已经帮我们搞定了吗？去瞄一下主dex的大小：8M 以及 `secondary.dex` 3M 。它是如何工作的？文档说dx的时候，先依据manifest里注册的组件生成一个 `main-list` ，然后把这list里的 `classes` 所依赖的 `classes` 找出来，把

他们打成 `classes.dex` 就是主dex。剩下的`classes`都放 `clsses2.dex` (如果使用参数限制dex大小的话可能会有 `classe3.dex` 等等) 。主dex至少含有 `main-list` 的 `classes` + 直接依赖`classes`，使用 `mini-main-list` 参数可以仅仅包含刚才说的 `classes`。

- 关于写分析脚本的思路是：直接使用 `mini-main-list` 参数获取build目录下的 `main-list` 文件，这样manifest声明的类和他们的直接依赖类搞定的了，那后者的直接依赖类怎么解？这些在dtk runtime也是必须的`classes`。一个思路是解析class文件获得该class的依赖类。还有一个思路是自己使用 `DexClassLoader` 加载dex，然后 `hook getClass()` 方法，调用一次就记录一个。都挺折腾的。
- 2.由于历史原因，总悟君在维护的App的manifest注册的组件的那些类，承载业务太多，依赖很多三方jar，导致直接依赖类非常多，而且短时间内无法梳理精简，没办法mini化主dex。
- 3.Application的启动入口太多。Application初始化未必是由 `launcher Activity` 的启动触发，还有可能是因为Service，Receiver，ContentProvider的启动。靠拦截重写 `Instrumentation execStartActivity` 解决不了问题。要为 Service，Receiver，ContentProvider 分别写基类，然后在 `onCreate()` 里判断是否要异步加载 `secondary.dex`。如果需要，弹出 Loading Activity？用户看到这个会感觉比较怪异。

结合自身App的实际情况来看美团的拆包方案虽然很美好然但是不能照搬啊。果然不能愉快地回家看动漫了。

## 第四回合 换一种思路

考虑到刚才说的2, 3原因，先不要急着动手写分析脚本。总悟君期望找到更好的方案。问题到现在变成了：既希望在Application的 `attachContext()` 方法里同步加载 `secondary.dex`，又不希望卡住UI线程。如果思路限制在线程异步化上，确实不可能实现。于是发现了[微信开发团队的这篇文章][wechat-multidex]。该文章介绍了关于这一问题 FB/QQ/微信的解决方案。FB的解决思路特别赞，让 `Launcher Activity` 在另外一个进程启动！当然这个 `Launcher Activity` 就是用来 `load dex` 的，`load`完成就启动 `Main Activity`。

微信这篇文章给出了一个非常重要的观点：安装完成之后第一次启动时，是 `secondary.dex` 的dexopt花费了更多的时间。认识到这点非常重要，使得问题又转化为：在不阻塞UI线程的前提下，完成dexopt，以后都不需要再次dexopt，所以可以在UI线程install dex 了！文章最后给了一个对FB方案的改进版。仔细读完感觉完全可行。

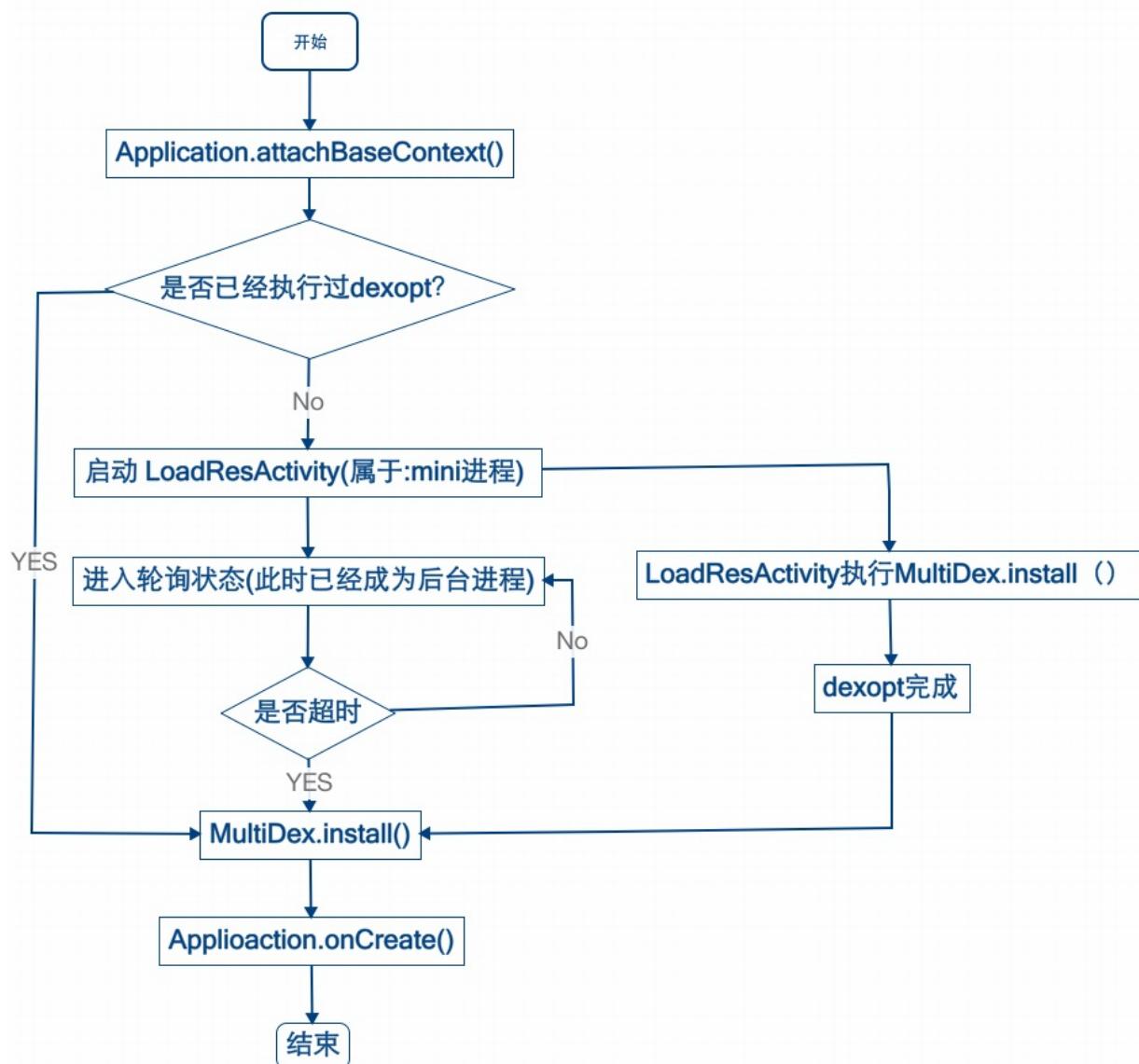
- 1.对现有代码改动量最小。
- 2.该方案不关注Application被哪个组件启动。Activity，Service，Receiver，

ContentProvider 都满足。(有个问题要说明：如细心网友指出的那样，新安装还未启动但是收到Receiver的场景下，会导致Load界面出现。这个场景实际出现几率比较少，且仅出现一次。可以接受。)

- 3.该方案不限制 Application , Activity , Service , Receiver , ContentProvider 继续新增业务。

于是总悟君实现了这篇文章最后介绍的改进版的方法，稍微有一点点扩充。

流程图如下：



上最终解决问题版的代码！在Application里面(这里不要再继承自MultiApplication了，我们要手动加载Dex)：

```

import java.util.Map;
import java.util.jar.Attributes;
import java.util.jar.JarFile;

```

```

import java.util.jar.Manifest;

public class App extends Application {
 public static final String KEY_DEX2_SHA1 = "dex2-SHA1-Digest";
 @Override
 protected void attachBaseContext(Context base) {
 super.attachBaseContext(base);
 LogUtils.d("loadDex", "App attachBaseContext ");
 if (!quickStart() && Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {/
 if (needWait(base)){
 waitForDexopt(base);
 }
 MultiDex.install (this);
 } else {
 return;
 }
 }

 @Override
 public void onCreate() {
 super.onCreate();
 if (quickStart()) {
 return;
 }
 ...
 }

 public boolean quickStart() {
 if (StringUtils.contains(getCurProcessName(this), ":mini")) {
 LogUtils.d("loadDex", ":mini start!");
 return true;
 }
 return false ;
 }
 //neead wait for dexopt ?
 private boolean needWait(Context context){
 String flag = get2thDexSHA1(context);
 LogUtils.d("loadDex", "dex2-sha1 "+flag);
 SharedPreferences sp = context.getSharedPreferences(
 PackageUtil.getPackageName(context).versionName, MODE_MULTI_PROCESS)
 String saveValue = sp.getString(KEY_DEX2_SHA1, "");
 return !StringUtils.equals(flag,saveValue);
 }
 /**
 * Get classes.dex file signature
 * @param context
 * @return
 */
 private String get2thDexSHA1(Context context) {
 ApplicationInfo ai = context.getApplicationInfo();
 String source = ai.sourceDir;
 try {

```

```

 JarFile jar = new JarFile(source);
 Manifest mf = jar.getManifest();
 Map<String, Attributes> map = mf.getEntries();
 Attributes a = map.get("classes2.dex");
 return a.getValue("SHA1-Digest");
 } catch (Exception e) {
 e.printStackTrace();
 }
 return null ;
}
// optDex finish
public void installFinish(Context context){
 Sharedpreferences sp = context.getSharedPreferences(
 PackageUtil.getPackageInfo(context).versionName, MODE_MULTI_PROCESS);
 sp.edit().putString(KEY_DEX2_SHA1,get2thDexSHA1(context)).commit();
}

public static String getCurProcessName(Context context) {
 try {
 int pid = android.os.Process.myPid();
 ActivityManager mActivityManager = (ActivityManager) context
 .getSystemService(Context.ACTIVITY_SERVICE);
 for (ActivityManager.RunningAppProcessInfo appProcess : mActivityManager
 .getRunningAppProcesses()) {
 if (appProcess.pid == pid) {
 return appProcess.processName;
 }
 }
 } catch (Exception e) {
 // ignore
 }
 return null ;
}
public void waitForDexopt(Context base) {
 Intent intent = new Intent();
 ComponentName componentName = new
 ComponentName("com.zongwu", LoadResActivity.class.getName());
 intent.setComponent(componentName);
 intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
 base.startActivity(intent);
 long startWait = System.currentTimeMillis();
 long waitTime = 10 * 1000 ;
 if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB_MR1) {
 waitTime = 20 * 1000 ;//实测发现某些场景下有些2.3版本有可能10s都不能完成optdex
 }
 while (needWait(base)) {
 try {
 long nowWait = System.currentTimeMillis() - startWait;
 LogUtils.d("loadDex" , "wait ms :" + nowWait);
 if (nowWait >= waitTime) {
 return;
 }
 }
 }
}

```

```
 }
 Thread.sleep(200);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
}
```

## PackageUtil的方法

```
public static PackageInfo getPackageInfo(Context context){
 PackageManager pm = context.getPackageManager();
 try {
 return pm.getPackageInfo(context.getPackageName(), 0);
 } catch (PackageManager.NameNotFoundException e) {
 LogUtils.e(e.getLocalizedMessage());
 }
 return new PackageInfo();
}
```

这里使用了classes(N).dex的方式保存了后面的dex而不是像微信目前的做法放到asset文件夹。前面有说到ART模式会将多个dex优化合并成oat文件。如果放置在asset里面就没有这个好处了。

Launcher Activity 依然是原来的代码里的WelcomeActivity。

在Application启动的时候会检测dexopt是否已经完成过，（检测方式是查看sp文件是否有dex文件的SHA1-Digest记录，这里要两个进程读取该sp,读取模式是MODE\_MULTI\_PROCESS）。如果没有就启动LoadDexActivity(属于：mini进程)。否则就直接install dex！对，直接install。通过日志发现，已经dexopt的dex文件再次install的时候只耗费几十毫秒。

LoadDexActivity 的逻辑比较简单，启动AsyncTask 来install dex 这时候会触发dexopt。

```
public class LoadResActivity extends Activity {
 @Override
 public void onCreate(Bundle savedInstanceState) {
 requestWindowFeature(Window.FEATURE_NO_TITLE);
 super.onCreate(savedInstanceState);
 getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN, WindowManager.LayoutParams.FLAG_FULLSCREEN);
 overridePendingTransition(R.anim.null_anim, R.anim.null_anim);
 setContentView(R.layout.layout_load);
 new LoadDexTask().execute();
 }
 class LoadDexTask extends AsyncTask {
 @Override
 protected Object doInBackground(Object[] params) {
 try {
 MultiDex.install(getApplicationContext());
 LogUtils.d("loadDex", "install finish");
 ((App) getApplication()).installFinish(getApplicationContext());
 } catch (Exception e) {
 LogUtils.e("loadDex", e.getLocalizedMessage());
 }
 return null;
 }
 @Override
 protected void onPostExecute(Object o) {
 LogUtils.d("loadDex", "get install finish");
 finish();
 System.exit(0);
 }
 }
 @Override
 public void onBackPressed() {
 //cannot backpress
 }
}
```

Manifest.xml 里面

```
<activity
 android:name= "com.zongwu.LoadResActivity"
 android:launchMode= "singleTask"
 android:process= ":mini"
 android:alwaysRetainTaskState= "false"
 android:excludeFromRecents= "true"
 android:screenOrientation= "portrait" />

<activity
 android:name= "com.zongwu.WelcomeActivity"
 android:launchMode= "singleTop"
 android:screenOrientation= "portrait">
 <intent-filter >
 <action android:name="android.intent.action.MAIN"/>
 <category android:name="android.intent.category.LAUNCHER"/>
 </intent-filter >
</activity>
```

替换Activity默认的出现动画 R.anim.null\_anim 文件的定义：

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
 <alpha
 android:fromAlpha="1.0"
 android:toAlpha="1.0"
 android:duration="550"/>
</set>
```

如[微信开发团队的这篇文章](#)所说， application 启动了 LoadDexActivity 之后，自身不再是前台进程所以怎么hold 线程都不会ANR。

系统何时会对apk进行dexopt总悟君其实并没有十分明白。通过查看安装运行的日志发现，安装的时候 packageManagerService 会对classes.dex 进行dexopt 。在调用 MultiDex.install() 加载 secondary.dex 的时候，也会进行一次dexopt 。这背后的流程到底是怎样的？dexopt是如何在另外一个进程执行的？如果是另外一个进程执行为何会阻塞主app的UI进程？官方文档并没有详细介绍这个，那就RTFSC一探究竟吧。

源代码跟踪比较长，移步到[这里](#)看吧。

其实你不知道MultiDex到底有多坑 2015-09-26 by 总悟君

遭遇MultiDex

愉快地写着Android代码的总悟君往工程里引入了一个默默无闻的jar然后Run了一下， 经过漫长的等待AndroidStudio构建失败了。于是总悟君带着疑惑查看错误信息。

UNEXPECTED TOP-LEVEL EXCEPTION: java.lang.IllegalArgumentException: method ID not in [0, 0xffff]: 65536 at  
com.android.dx.merge.DexMerger\$6.updateIndex(DexMerger.java:501) at  
com.android.dx.merge.DexMerger\$IdMerger.mergeSorted(DexMerger.java:276) at  
com.android.dx.merge.DexMerger.mergeMethodIds(DexMerger.java:490) at  
com.android.dx.merge.DexMerger.mergeDexes(DexMerger.java:167) at  
com.android.dx.merge.DexMerger.merge(DexMerger.java:188) at  
com.android.dx.command.dexer.Main.mergeLibraryDexBuffers(Main.java:439) at  
com.android.dx.command.dexer.Main.runMonoDex(Main.java:287) at  
com.android.dx.command.dexer.Main.run(Main.java:230) at  
com.android.dx.command.dexer.Main.main(Main.java:199) at  
com.android.dx.command.Main.main(Main.java:103):Derp:dexDerpDebug FAILED 看起来是：在试图将 classes 和 jar 塞进一个 Dex 文件的过程中产生了错误。早期的 Dex 文件保存所有 classes 的方法个数的范围在 0~65535 之间。业务一直在增长，总悟君写(copy)的代码越来越长引入的库越来越多，超过这个范围只是时间问题。怎么解？？太阳底下木有新鲜事，淡定先 google 一发，找找已经踩过坑的小伙伴。StackOverflow 的网友们对该问题表示情绪稳定，谈笑间抛出 multiDex。这是 Android 官网对当初的短视行为给出的补丁方案。文档说，Dalvik Executable (DEX) 文件的总方法数限制在 65536 以内，其中包括 Android framework method, lib method (后来总悟君发现仅仅是 Android 自己的框架的方法就已经占用了 1w 多)，还有你的 code method，所以请使用 MultiDex。对于 5.0 以下版本，请使用 multidex support library (这个是我们的补丁包！build tools 请升级到 21)。而 5.0 及以上版本，由于 ART 模式的存在，app 第一次安装之后会进行一次预编译(pre-compilation)，如果这时候发现了 classes(..N).dex 文件的存在就会将它们最终合成为一个.oat 的文件，嗯看起来很厉害的样子。同时 Google 建议 review 代码的直接或者间接依赖，尽可能减少依赖库，设置 proguard 参数进一步优化去除无用的代码。嗯，这两个实施起来倒是很简单，但是治标不治本，躲得过初一躲不过十五。在 Google 给出这个解决方案之前，他们的开发人员先给了一个简陋简易版本的 multiDex 具体参看这里。（怀疑后来的官方解决方案就有这家伙参与）。简单地说就是：1. 先把你的 app 的 class 拆分成主次两个 dex。2. 你的程序运行起来后，自己把第二个 dex 给 load 进来。看就这么简单！而且这就是个动态加载模块的框架！然而总悟君早已看穿 Dalvik VM 这种动态加载 dex 的能力归根结底还是因为 Java 的 classloader 类加载机制。沿着这条道走，Android 模块动态化加载，包括 dex 级别和 apk 级别的动态化加载，各种玩法层出不穷。参见这里 123456。

## 第一回合 天真的官方补丁方案

还是先解决打包问题，回头再研究那些高深的动态化加载技术。偷懒一下咯考虑到投入产出比，决定使用 Google 官方的 multiDex 解决。（Google 的补丁方案啊，不会再有坑了吧？后面才发现还是太天真）该方案有两步：1. 修改 gradle 脚本来产生多 dex。2. 修改 manifest 使用 MultiDexApplication。步骤 1. 在 gradle 脚本里写上：

```
android { compileSdkVersion 21 buildToolsVersion "21.1.0" defaultConfig { ...
minSdkVersion 14 targetSdkVersion 21 ... // Enabling multidex support.
multiDexEnabled true } ... } dependencies { compile 'com.android.support:multidex:1.0.0'
} 步骤2. manifest声明修改
```

<?xml version="1.0" encoding="utf-8"?>

... </manifest> 如果有自己的Application，继承MultiDexApplication。如果当前代码已经继承自其它Application没办法修改那也行，就重写 Application的attachBaseContext()这个方法。

```
@Override protected void attachBaseContext(Context base) {
super.attachBaseContext(base); MultiDex.install(this);
```

} run一下，可以了！但是dex过程好像变慢了。。。文档还写明了multiDex support lib 的局限。瞄一下是什么： 1.在应用安装到手机上的时候dex文件的安装是复杂的(complex)有可能会因为第二个dex文件太大导致ANR。请用proguard优化你的代码。呵呵 2.使用了multiDex的App有可能在4.0(api level 14)以前的机器上无法启动，因为Dalvik linearAlloc bug(Issue 22586)。请多多测试自祈多福。用proguard优化你的代码将减少该bug几率。呵呵 3.使用了multiDex的App在runtime期间有可能因为Dalvik linearAlloc limit (Issue 78035) Crash。该内存分配限制在 4.0版本被增大，但是5.0以下的机器上的Apps依然会存在这个限制。 4.主dex被dalvik虚拟机执行时候，哪些类必须在主dex文件里面这个问题比较复杂。build tools 可以搞定这个问题。但是如果你代码存在反射和native的调用也不保证100%正确。呵呵

感觉这就是个坑啊。补丁方案又引入一些问题。但是插件化方案要求对现有代码有比较大的改动，代价太大，而且动态化加载框架意味着维护成本更高，会有更多潜在bug。所以先测试，遇到有问题的版本再解决。

第二回合 啥？ dexopt failed?

呵呵，部分低端2.3机型(话说2.3版本的android机有高端机型么)安装失败！

INSTALL\_FAILED\_DEXOPT。这个就是前面说的Issue 22586问题。apk是一个zip压缩包，dalvik每次加载apk都要从中解压出class.dex文件，加载过程还涉及到dex的classes需要的杂七杂八的依赖库的加载，真耗时间。于是Android决定优化一下这个问题，在app安装到手机之后，系统运行dexopt程序对dex进行优化，将dex的依赖库文件和一些辅助数据打包成odex文件。存放在cache/dalvik\_cache目录下。保存格式为apk路径 @ apk名 @ classes.dex。这样以空间换时间大大缩短读取/加载dex文件的过程。那刚才那个bug是啥问题呢，原来dexopt程序的dalvik分配一块内存来统计你的app的dex里面的classes的信息，由于classes太多方法太多超过这个linearAlloc 的限制。那减小dex的大小就可以咯。gradle脚本如下：

```
android.applicationVariants.all { variant -> dex.doFirst{ dex-> if
(dex.additionalParameters == null) { dex.additionalParameters = [] }
dex.additionalParameters += '--set-max-idx-number=48000' } } --set-max-idx-number= 用
于控制每一个dex的最大方法个数，写小一点可以产生好几个dex。踩过更多坑的FB的工
程师表示这个linearAlloc的限制不仅仅在安装时候的dexopt程序里7，还在你的app的
dalvik rumtime里。（很显然啊dtk vm的宿主进程fork自于同一个母体啊）。为了表示对
这个坑的不满以及对Google的产品表示遗憾，FB工程师Read The Fucking Source Code
找到了一个hack方案。这个linearAlloc的size定义在c层而且是一个全局变量，他们通过对
结构体的size的计算成功覆盖了该值的内容，这里要特别感谢C语言的指针和内存的设
计。C的世界里，You Are The King of This World。当然实际情况是大部分用户用这把利
刃割伤了自己。。。别问总悟君谁是世界上最好的语言。。。为FB的工程师的机智和务实
精神点赞！然而总悟君不愿意花那么多精力实现FB的hack方法。（dtk虚拟机c层代码
在2.x 4.x 版本里有变更，找到那个内存地址太难，未必搞得定啊）我们有偷懒的解决方
案，为了避免2.3机型runtime 的linearAlloclimit ,最好保持每一个dex体积<4M ,刚才的的
value<=48000 好了 现在2.3的机器可以安装run起来了！
```

### 第三回合 ANR的意思就是Application Not Responding

问题又来了！这次不仅仅是2.3 的机型！还有一些中档配置的4.x系统的机型。问题现象是：第一次安装后，点击图标，1s, 2s, 3s... 程序没有任何反应就好像你没点图标一样。5s过去。。。程序ANR！其实不仅仅总悟君的App存在这个问题，其他很多App也存在首次安装运行后几秒都无任何响应的现象或者最后ANR了。唯一的例外是美团App，点击图标立马就出现界面。唉要不就算啦？反正就一次。。。不行，这可是产品给用户的第一印象啊太重要了，而且美团搞得定就说明这问题有解决方案。ANR了是不是局限1描述的现象？？不过也不重要...因为Google只是告诉你说第二个dex太大了导致的。并没有进一步解释根本原因。怎么办？Google一发？搜索点击图标然后ANR？怎么可能有解决方案嘛。ANR就意味着UI线程被阻塞了，老老实实查看log吧。adb logcat -v time > log.txt 于是发现 是 install dex + dexopt 时间太长！梳理一下流程：安装完app点击图标之后，系统木有发现对应的process，于是从该apk抽取classes.dex(主dex) 加载，触发一次dexopt。App的laucherActivity准备启动，触发Application启动，Application的onattach () 方法调用，这时候MultiDex.install () 调用，classes2.dex 被install，再次触发dexopt。然后Applicaiton onCreate () 执行。然后 launcher Activity真的起来了。这些必须在5s内完成不然就ANR给你看！有点棘手。首先主dex是无论如何都绕不过加载和dexopt的。如果主dex比较小的话可以节省时间。主dex小就意味着后面的dex大啊，MultiDex.install () 是在主线程里做的，总时间又没有实质性改变。install () 能不能放到线程里做啊？貌似不行。。。如果异步化，什么时候install完成都不知道。这时候如果进程需要seconday.dex里的classes信息不就悲剧？主dex越小这个错误几率就越大。要悲剧啊总悟君。淡定，这次Google搜索MultiDex.install 。于是总悟君发现了美团多dex拆包方案。读完之后感觉看到胜利曙光。美团的主要思路是：精简主dex+异步加载

secondary.dex。对异步化执行速度的不确定性，他们的解决方案是重写Instrumentation execStartActivity 方法，hook跳转Activity的总入口做判断，如果当前secondary.dex还没有加载完成，就弹一个loading Activity等待加载完成，如果已经加载完成那最好不过了。不错，RTFSC果然是王道。可以试一试。但是有几个问题需要解决：1.分析主dex需要的classes这个脚本比较难写。。。Google文档说过这个问题比较复杂，而且buildTools不是已经帮我们搞定了吗？去瞄一下主dex的大小：8M 以及secondary.dex 3M。它是如何工作的？文档说dx的时候，先依据manifest里注册的组件生成一个 main-list，然后把这list里的classes所依赖的classes找出来，把他们打成classes.dex就是主dex。剩下的classes都放clsses2.dex(如果使用参数限制dex大小的话可能会有classe3.ex 等等)。主dex至少含有main-list 的classes + 直接依赖classes，使用mini-main-list参数可以仅仅包含刚才说的classes。关于写分析脚本的思路是：直接使用mini-main-list参数获取build目录下的main-list文件，这样manifest声明的类和他们的直接依赖类搞定的了，那后者的直接依赖类怎么解？这些在dtk runtime也是必须的classes。一个思路是解析class文件获得该class的依赖类。还有一个思路是自己使用Dexclassloader 加载dex，然后hook getClass()方法，调用一次就记录一个。都挺折腾的。2.由于历史原因，总悟君在维护的App的manifest注册的组件的那些类，承载业务太多，依赖很多三方jar，导致直接依赖类非常多，而且短时间内无法梳理精简，没办法mini化主dex。3.Application的启动入口太多。Appication初始化未必是由launcher Activity的启动触发，还有可能是因为Service，Receiver，ContentProvider 的启动。靠拦截重写Instrumentation execStartActivity 解决不了问题。要为 Service， Receiver， ContentProvider 分别写基类，然后在oncreate()里判断是否要异步加载secondary.dex。如果需要，弹出Loading Acitvity？用户看到这个会感觉比较怪异。结合自身App的实际情况来看美团的拆包方案虽然很美好然但是不能照搬啊。果然不能愉快地回家看动漫了。

#### 第四回合 换一种思路

考虑到刚才说的2，3原因，先不要急着动手写分析脚本。总悟君期望找到更好的方案。问题到现在变成了：既希望在Application的attachContext () 方法里同步加载secondary.dex，又不希望卡住UI线程。如果思路限制在线程异步化上，确实不可能实现。于是发现了微信开发团队的这篇文章。该文章介绍了关于这一问题 FB/QQ/微信的解决方案。FB的解决思路特别赞，让Launcher Activity在另外一个进程启动！当然这个Launcher Activity就是用来load dex 的，load完成就启动Main Activity。微信这篇文章给出了一个非常重要的观点：安装完成之后第一次启动时，是secondary.dex的dexopt花费了更多的时间。认识到这点非常重要，使得问题又转化为：在不阻塞UI线程的前提下，完成dexopt，以后都不需要再次dexopt，所以可以在UI线程install dex 了！文章最后给了一个对FB方案的改进版。仔细读完感觉完全可行。1.对现有代码改动量最小。2.该方案不关注Application被哪个组件启动。Activity，Service，Receiver，ContentProvider 都满足。(有个问题要说明：如细心网友指出的那样，新安装还未启动但是收到Receiver的场景下，会导致Load界面出现。这个场景实际出现几率比较少，且仅出现一次。可以接

受。) 3.该方案不限制 Application , Activity , Service , Receiver , ContentProvider 继续新增业务。于是总悟君实现了这篇文章最后介绍的改进版的方法，稍微有一点点扩充。

流程图如下

方案的流程图 上最终解决问题版的代码！ 在Application里面(这里不要再继承自 MultiApplication了， 我们要手动加载Dex):

```
import java.util.Map; import java.util.jar.Attributes; import java.util.jar.JarFile; import
java.util.jar.Manifest; public class App extends Application { public static final String
KEY_DEX2_SHA1 = "dex2-SHA1-Digest"; @Override protected void
attachBaseContext(Context base) { super .attachBaseContext(base); LogUtils.d(
"loadDex", "App attachBaseContext "); if (!quickStart() && Build.VERSION.SDK_INT <
Build.VERSION_CODES.LOLLIPOP) {//>=5.0的系统默认对dex进行oat优化 if
(needWait(base)){ waitForDexopt(base); } MultiDex.install (this); } else { return; } }
@Override public void onCreate() { super .onCreate(); if (quickStart()) { return; } ... }
public boolean quickStart() { if (StringUtils.contains(getCurProcessName(this), ":mini")) {
LogUtils.d("loadDex", ":mini start!"); return true; } return false ; } //neead wait for dexopt
? private boolean needWait(Context context){ String flag = get2thDexSHA1(context);
LogUtils.d("loadDex", "dex2-sha1 "+flag); SharedPreferences sp =
context.getSharedPreferences(PackageUtil.getPackageInfo(context). versionName,
MODE_MULTI_PROCESS); String saveValue = sp.getString(KEY_DEX2_SHA1, ""); return !StringUtils.equals(flag,saveValue); } /**
```

```
* Get classes.dex file signature
* @param context
* @return
*/
private String get2thDexSHA1(Context context) {
 ApplicationInfo ai = context.getApplicationInfo();
 String source = ai.sourceDir;
 try {
 JarFile jar = new JarFile(source);
 Manifest mf = jar.getManifest();
 Map<String, Attributes> map = mf.getEntries();
 Attributes a = map.get("classes2.dex");
 return a.getValue("SHA1-Digest");
 } catch (Exception e) {
 e.printStackTrace();
 }
 return null ;
}
// optDex finish
public void installFinish(Context context){
```

```

 SharedPreferences sp = context.getSharedPreferences(
 PackageUtil.getPackageInfo(context).versionName, MODE_MULTI_PROCESS);
 sp.edit().putString(KEY_DEX2_SHA1, get2thDexSHA1(context)).commit();
 }

 public static String getCurProcessName(Context context) {
 try {
 int pid = android.os.Process.myPid();
 ActivityManager mActivityManager = (ActivityManager) context
 .getSystemService(Context.ACTIVITY_SERVICE);
 for (ActivityManager.RunningAppProcessInfo appProcess : mActivityManager
 .getRunningAppProcesses()) {
 if (appProcess.pid == pid) {
 return appProcess.processName;
 }
 }
 } catch (Exception e) {
 // ignore
 }
 return null;
 }

 public void waitForDexopt(Context base) {
 Intent intent = new Intent();
 ComponentName componentName = new
 ComponentName("com.zongwu", LoadResActivity.class.getName());
 intent.setComponent(componentName);
 intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
 base.startActivity(intent);
 long startWait = System.currentTimeMillis();
 long waitTime = 10 * 1000;
 if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB_MR1) {
 waitTime = 20 * 1000; // 实测发现某些场景下有些2.3版本有可能10s都不能完成optdex
 }
 while (needWait(base)) {
 try {
 long nowWait = System.currentTimeMillis() - startWait;
 LogUtils.d("loadDex", "wait ms :" + nowWait);
 if (nowWait >= waitTime) {
 return;
 }
 Thread.sleep(200);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
}

```

} PackageUtil的方法

public static PackageInfo getPackageInfo(Context context){ PackageManager pm = context.getPackageManager(); try { return pm.getPackageInfo(context.getPackageName(), 0); } catch (PackageManager.NameNotFoundException e) { LogUtils.e(e.getLocalizedMessage()); } return new PackageInfo(); } 这里使用了classes(N).dex的方式保存了后面的dex而不是像微信目前的做法放到assest文件夹。前面有说到ART模式会将多个dex优化合并成oat文件。如果放置在asset里面就没有这个好处了。Launcher Activity 依然是原来的代码里的WelcomeActivity。在Application启动的时候会检测dexopt是否已经完成过，（检测方式是查看sp文件是否有dex文件的SHA1-Digest记录，这里要两个进程读取该sp,读取模式是MODE\_MULTI\_PROCESS）。如果没有就启动LoadDexActivity(属于：mini进程)。否则就直接install dex！对，直接install。通过日志发现，已经dexopt的dex文件再次install的时候只耗费几十毫秒。LoadDexActivity 的逻辑比较简单，启动AsyncTask 来install dex这时候会触发dexopt。

```
public class LoadResActivity extends Activity { @Override public void onCreate(Bundle savedInstanceState) { requestWindowFeature(Window.FEATURE_NO_TITLE); super.onCreate(savedInstanceState); getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN , WindowManager.LayoutParams.FLAG_FULLSCREEN); overridePendingTransition(R.anim.null_anim, R.anim.null_anim); setContentView(R.layout.layout_load); new LoadDexTask().execute(); } class LoadDexTask extends AsyncTask { @Override protected Object doInBackground(Object[] params) { try { MultiDex.install(getApplicationContext()); LogUtils.d("loadDex" , "install finish"); ((App) getApplication()).installFinish(getApplicationContext()); } catch (Exception e) { LogUtils.e("loadDex" , e.getLocalizedMessage()); } return null; } @Override protected void onPostExecute(Object o) { LogUtils.d("loadDex" , "get install finish"); finish(); System.exit(0); } } @Override public void onBackPressed() { //cannot backpress } Manifest.xml 里面
```

</intent-filter> 替换Activity默认的出现动画 R.anim.null\_anim 文件的定义：

如微信开发团队的这篇文章所说，application启动了LoadDexActivity之后，自身不再是前台进程所以怎么hold 线程都不会ANR。

系统何时会对apk进行dexopt总悟君其实并没有十分明白。通过查看安装运行的日志发现，安装的时候packageManagerService会对classes.dex 进行dexopt。在调用MultiDex.install () 加载 secondary.dex的时候，也会进行一次dexopt。这背后的流程到底是怎样的？dexopt是如何在另外一个进程执行的？如果是另外一个进程执行为何会阻塞主app的UI进程？官方文档并没有详细介绍这个，那就RTFSC一探究竟吧.

源代码跟踪比较长，移步到[这里](#)看吧。

## 最终章碎碎念

- MultiDex的问题难点在：要持续解决好几个bug才能最终解决问题。进一步的，想要仔细分辨且解决这些bug，就必须持续探索一些关联性的概念和原理
- 耗费了这么多时间来解决了Android系统的缺陷是不是有点略伤心。这不应该是Google给出一个比较彻底的解决方案吗？
- FB的工程师们脑洞好大。思考问题的方式很值得借鉴。
- 微信团队的文章提到逆向了不少App。哈！总悟君感觉增长知识拓宽视野的新技能加强。
- RTFSC是王道。
- 在查看log的过程中发现一个比较有趣的现象。在App的secondary.dex加载之前居然先加载了某数字公司的dex！（手机没有root但是安装了xx手机助手）再加上之前看到的错误堆栈里Android framework的调用堆栈之间也赫然有他们的代码。总悟君恶意猜测该app利用了某种手段进行了提权，hook了系统框架代码，将自己的代码注入到了每一个应用app的进程里。嗯。。。有趣。。。
- �恩今晚已经没有时间看动漫了。。。

# 各大热补丁方案分析和比较

来源:[blog.zhaiyifan.cn](http://blog.zhaiyifan.cn)

最近开源界涌现了很多热补丁项目，但从方案上来说，主要包括

[Dexposed](#)、[AndFix](#)、[ClassLoader](#)（来源是原QZone，现淘宝的工程师陈钟，在15年年初就已经开始实现）三种。前两个都是阿里巴巴内部的不同团队做的（淘宝和支付宝），后者则来自腾讯的QQ空间团队。

开源界往往一个方案会有好几种实现（比如ClassLoader方案已经有不下三种实现了），但这三种方案的原理却迥然不同，那么让我们来看看它们三者的原理和各自的优缺点吧。

## Dexposed

基于[Xposed](#)的AOP框架，方法级粒度，可以进行AOP编程、插桩、热补丁、SDK hook等功能。

Xposed需要Root权限，是因为它要修改其他应用、系统的行为，而对单个应用来说，其实不需要root。Xposed通过修改Android Dalvik运行时的Zygote进程，并使用Xposed Bridge来hook方法并注入自己的代码，实现非侵入式的runtime修改。比如蜻蜓fm和喜马拉雅做的事情，其实就很适合这种场景，别人反编译市场下载的代码是看不到patch的行为的。小米(onVmCreated里面还未小米做了资源的处理)也重用了dexposed，去做了很多自定义主题的功能，还有沉浸式状态栏等。

我们知道，应用启动的时候，都会fork zygote进程，装载class和invoke各种初始化方法，Xposed就是在这个过程中，替换了app\_process，hook了各种入口级方法（比如handleBindApplication、ServerThread、ActivityThread、ApplicationPackageManager的getResourcesForApplication等），加载XposedBridge.jar提供动态hook基础。

具体到方法，可参见[XposedBridge](#):

```
/**
 * Intercept every call to the specified method and call a handler function instead.
 * @param method The method to intercept
 */
private native synchronized static void hookMethodNative(Member method, Class<?> decl,
```

其具体native实现则在[Xposed的libxposed\\_common.cpp](#)里面有注册，根据系统版本分发到libxposed\_dalvik和libxposed\_art里面，以dalvik为例大致来说就是记录下原来的方法信息，并把方法指针指向我们的hookedMethodCallback，从而实现拦截的目的。

方法级的替换是指，可以在方法前、方法后插入代码，或者直接替换方法。只能针对java方法做拦截，不支持C的方法。

来说说硬伤吧，不支持art，不支持art，不支持art。重要的事情要说三遍。尽管在6月，项目网站的roadmap就写了7、8月会支持art，但事实是现在还无法解决art的兼容。

另外，如果线上release版本进行了混淆，那写patch也是一件很痛苦的事情，反射+内部类，可能还有包名和内部类的名字冲突，总而言之就是写得很痛苦。

了7、8月会支持art，但事实是现在还无法解决art的兼容。

另外，如果线上release版本进行了混淆，那写patch也是一件很痛苦的事情，反射+内部类，可能还有包名和内部类的名字冲突，总而言之就是写得很痛苦。

## AndFix

同样是方法的hook，AndFix不像Dexposed从Method入手，而是以Field为切入点。

先看Java入口，`AndFixManager.fix`：

```

/**
 * fix
 *
 * @param file patch file
 * @param classLoader classloader of class that will be fixed
 * @param classes classes will be fixed
 */
public synchronized void fix(File file, ClassLoader classLoader, List<String> classes
 // 省略...判断是否支持, 安全检查, 读取补丁的dex文件

 ClassLoader patchClassLoader = new ClassLoader(classLoader) {
 @Override
 protected Class<?> findClass(String className) throws ClassNotFoundException {
 Class<?> clazz = dexFile.loadClass(className, this);
 if (clazz == null && className.startsWith("com.alipay.euler.andfix"))
 return Class.forName(className); // annotation's class not found
 }
 if (clazz == null) {
 throw new ClassNotFoundException(className);
 }
 return clazz;
 }
 };
 Enumeration<String> entrys = dexFile.entries();
 Class<?> clazz = null;
 while (entrys.hasMoreElements()) {
 String entry = entrys.nextElement();
 if (classes != null && !classes.contains(entry)) {
 continue; // skip, not need fix
 }
 // 找到了, 加载补丁class
 clazz = dexFile.loadClass(entry, patchClassLoader);
 if (clazz != null) {
 fixClass(clazz, classLoader);
 }
 }
} catch (IOException e) {
 Log.e(TAG, "patch", e);
}
}

```

看来最终fix是在 fixClass 方法:

```

private void fixClass(Class<?> clazz, ClassLoader classLoader) {
 Method[] methods = clazz.getDeclaredMethods();
 MethodReplace methodReplace;
 String clz;
 String meth;
 // 遍历补丁class里的方法，进行一一替换，annotation则是补丁工具自动加上的
 for (Method method : methods) {
 methodReplace = method.getAnnotation(MethodReplace.class);
 if (methodReplace == null)
 continue;
 clz = methodReplace.clazz();
 meth = methodReplace.method();
 if (!isEmpty(clz) && !isEmpty(meth)) {
 replaceMethod(classLoader, clz, meth, method);
 }
 }
}

private void replaceMethod(ClassLoader classLoader, String clz, String meth, Method m
try {
 String key = clz + "@" + classLoader.toString();
 Class<?> clazz = mFixedClass.get(key);
 if (clazz == null) {// class not load
 // 要被替换的class
 Class<?> clzz = classLoader.loadClass(clz);
 // 这里也很黑科技，通过C层，改写accessFlags，把需要替换的类的所有方法（Field）改成了public
 clazz = AndFix.initTargetClass(clzz);
 }
 if (clazz != null) {// initialize class OK
 mFixedClass.put(key, clazz);
 // 需要被替换的函数
 Method src = clazz.getDeclaredMethod(meth, method.getParameterTypes());
 // 这里是调用了jni，art和dalvik分别执行不同的替换逻辑，在cpp进行实现
 AndFix.addReplaceMethod(src, method);
 }
} catch (Exception e) {
 Log.e(TAG, "replaceMethod", e);
}
}

```

在dalvik和art上，系统的调用不同，但是原理类似，这里我们尝个鲜，以6.0为例 `art_method_replace_6_0`：

```

// 进行方法的替换
void replace_6_0(JNIEnv* env, jobject src, jobject dest) {
 art::mirror::ArtMethod* smeth = (art::mirror::ArtMethod*) env->FromReflectedMethod(
 art::mirror::ArtMethod* dmeth = (art::mirror::ArtMethod*) env->FromReflectedMethod(
 dmeth->declaring_class_->class_loader_ =
 smeth->declaring_class_->class_loader_; //for plugin classloader
 dmeth->declaring_class_->clinit_thread_id_ =
 smeth->declaring_class_->clinit_thread_id_;
 dmeth->declaring_class_->status_ = smeth->declaring_class_->status_-1;

// 把原方法的各种属性都改成补丁方法的
smeth->declaring_class_ = dmeth->declaring_class_;
smeth->dex_cache_resolved_types_ = dmeth->dex_cache_resolved_types_;
smeth->access_flags_ = dmeth->access_flags_;
smeth->dex_cache_resolved_methods_ = dmeth->dex_cache_resolved_methods_;
smeth->dex_code_item_offset_ = dmeth->dex_code_item_offset_;
smeth->method_index_ = dmeth->method_index_;
smeth->dex_method_index_ = dmeth->dex_method_index_;

// 实现的指针也替换为新的
smeth->ptr_sized_fields_.entry_point_from_interpreter_ =
 dmeth->ptr_sized_fields_.entry_point_from_interpreter_;
smeth->ptr_sized_fields_.entry_point_from_jni_ =
 dmeth->ptr_sized_fields_.entry_point_from_jni_;
smeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_ =
 dmeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_;

LOGD("replace_6_0: %d , %d",
 smeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_,
 dmeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_);
}

// 这就是上面提到的，把方法都改成public的，所以说了解一下jni还是很有必要的，java世界在c世界是有映射的
void setFieldFlag_6_0(JNIEnv* env, jobject field) {
 art::mirror::ArtField* artField =
 (art::mirror::ArtField*) env->FromReflectedField(field);
 artField->access_flags_ = artField->access_flags_ & (~0x0002) | 0x0001;
 LOGD("setFieldFlag_6_0: %d ", artField->access_flags_);
}

```

在dalvik上的实现略有不同，是通过jni bridge来指向补丁的方法。

使用上，直接写一个新的类，会由补丁工具会生成注解，描述其与要打补丁的类和方法的对应关系。

## ClassLoader

原腾讯空间Android工程师，也是我的启蒙老师的陈钟发明的热补丁方案，是他在看源码的时候偶然发现的切入点。

我们知道，multidex方案的实现，其实就是把多个dex放进app的classloader之中，从而使得所有dex的类都能被找到。而实际上findClass的过程中，如果出现了重复的类，参照下面的类加载的实现，是会使用第一个找到的类的。

```
public Class findClass(String name, List<Throwable> suppressed) {

 for (Element element : dexElements) { //每个Element就是一个dex文件
 DexFile dex = element.dexFile;
 if (dex != null) {
 Class clazz = dex.loadClassBinaryName(name, definingContext, suppressed);
 if (clazz != null) {
 return clazz;
 }
 }
 }
 if (dexElementsSuppressedExceptions != null) {
 suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));
 }
 return null;
}
```

该热补丁方案就是从这一点出发，只要把有问题的类修复后，放到一个单独的dex，通过反射插入到dexElements数组的最前面，不就可以让虚拟机加载到打完补丁的class了吗。

说到此处，似乎已经是一个完整的方案了，但在实践中，会发现运行加载类的时候报 preverified 错误，原来在 `DexPrepare.cpp`，将dex转化成odex的过程中，会在 `DexVerify.cpp` 进行校验，验证如果直接引用到的类和clazz是否在同一个dex，如果是，则会打上 `CLASS_ISPREVERIFIED` 标志。通过在所有类（Application除外，当时还没加载自定义类的代码）的构造函数插入一个对在单独的dex的类的引用，就可以解决这个问题。空间使用了 `javaassist` 进行编译时字节码插入。

开源实现有 [Nuwa](#), [HotFix](#), [DroidFix](#)。

## 比较

Dexposed不支持Art模式（5.0+），且写补丁有点困难，需要反射写混淆后的代码，粒度太细，要替换的方法多的话，工作量会比较大。

AndFix支持2.3-6.0，但是不清楚是否有一些机型的坑在里面，毕竟jni层不像java曾一样标准，从实现来说，方法类似Dexposed，都是通过jni来替换方法，但是实现上更简洁直接，应用patch不需要重启。但由于从实现上直接跳过了类初始化，设置为初始化完毕，所以像是静态函数、静态成员、构造函数都会出现问题，复杂点的类Class.forName很可能直接就会挂掉。

ClassLoader方案支持2.3-6.0，会对启动速度略微有影响，只能在下一次应用启动时生效，在空间中已经有了较长时间的线上应用，如果可以接受在下次启动才应用补丁，是很好的选择。

总的来说，在兼容性稳定性上，ClassLoader方案很可靠，如果需要应用不重启就能修复，而且方法足够简单，可以使用 AndFix，而 Dexposed由于还不能支持art，所以只能暂时放弃，希望开发者们可以改进使它能支持art模式，毕竟xposed的种种能力还是很吸引人的（比如hook别人app的方法拿到解密后的数据，嘿嘿），还有比如无痕埋点啊线上追踪问题之类的，随时可以下掉。

# 当Field邂逅65535

来源:[Jason's Blog](#)

很多Android开发者都知道，同一个dex中方法数不能超过65535。为什么呢？来看一下[这篇文章](#)，有3万的阅读量啊，当年一定风靡一时，作者说65535是dexopt引起的，我特马真是呵呵了，写文章不动脑子，dexopt都到了什么阶段了？再来看[这篇文章](#)，解释和上面如出一辙，可以看到前面那篇文章流毒不浅啊，关键这哥们貌似最近还出了本书，这种狗屁态度还好意思出书？

『天下熙熙，皆为利来；天下攘攘，皆为利往。』大家写文章，为名为利无可厚非，有错误都是有情可原，但关键是要有点认真的态度啊。当年写 构建神器Gradle 花了好几天的时间，投稿后又修改到深夜，依然还是有些错误，后来也在自己的博客上做了勘误更新。我看扔物线那篇RxJava的文章都有内测读者群，想来也是极为认真，非常尊敬这种态度，非常喜欢这样的开发者。

## Field 65K

言归正传，来聊聊为什么方法数不能超过65535？搬上Dalvik工程师在[SF上的回答](#)，因为在[Dalvik指令集](#)里，调用方法的invoke-kind指令中，method reference index只给了16bits，最多能调用65535个方法，所以在生成dex文件的过程中，当方法数超过65535就会报错。细看指令集，除了method，field和class的index也是16bits，所以也存在65535的问题。一般来说，method的数目会比field和class多，所以method数会首先遇到65535问题，你可能都没机会见到field过65535的情况。

幸运的我见到了，呵呵。

```
UNEXPECTED TOP-LEVEL EXCEPTION:
com.android.dex.DexException: Too many classes in --main-dex-list, main dex capacity exceeded
at com.android.dx.command.dexer.Main.processAllFiles(Main.java:494)
at com.android.dx.command.dexer.Main.runMultiDex(Main.java:334)
at com.android.dx.command.dexer.Main.run(Main.java:244)
at com.android.dx.command.dexer.Main.main(Main.java:215)
at com.android.dx.command.Main.main(Main.java:106)
Execution failed for task ':Nova:dexCommonDebug'.
!> com.android.ide.common.process.ProcessException: org.gradle.process.internal.ExecException: Process
'command 'C:\Program Files\Java\jdk1.8.0_31\bin\java.exe'' finished with non-zero exit value 2
```

明明显示的是 DexException: Too many classes in --main-dex-list，和field有毛关系？你可能想问我脑子是不是进水了，唉，明明是Google工程师脑子有问题，你看看Google自家兄弟提的[issue](#)就知道了。在使用multidex的情况下，主dex不管method还是field数目超

了65535都会报 `Too many classes` 的错误，不使用multidex，就报正常的错误了，呵呵。

你可能要纳闷了，我为啥会遇到field超65535的问题？看看我“新大美”同事写的，哈哈：

实际应用中我们还遇到另外一个比较棘手的问题，就是Field的过多的问题，Field过多是我们目前采用的代码组织结构引入的，我们为了方便多业务线、多团队并发协作的情况下开发，我们采用的aar的方式进行开发，并同时在aar依赖链的最底层引入了一个通用业务aar，而这个通用业务aar中包含了很多资源，而ADT14以及更高的版本中对Library资源处理时，Library的R资源不再是static final的了，详情请查看google官方说明，这样在最终打包时Library中的R没法做到内联，这样带来了R field过多的情况。

差不多的情况，我们上层十几个业务线为独立module，都依赖base，而base的资源id有个三四千，上层R文件会把下层的R文件合并过来，使用multidex后，会把manifest里的activity、service等和其直接引用类加到main dex中，所以很多R文件涌入，field超个65535那都不叫事。

## 修改R

field这么多怎么办呢？我们大胆假设只保留最顶层的R文件，因为这个R文件会把下层R文件合并过来，所有的R引用都可以指向这个文件。下层的类要引用最上层的R文件，下层不可能依赖上层，所以修改源代码肯定是走不通的，那就改class文件字节码吧。遍历class文件，把R的引用都指向最上层，把其他没用的R文件删掉。

## 问题

思路有了，接下来的操作有以下问题：

- 什么时候修改？

dex过程是把全部class文件转换成dex文件，所以class字节码的修改要在dex之前，我们决定介入构建流程，在dex之前添加一个gradle任务，用来修改字节码。

- 用什么修改？

可以使用asm这个库，由于android gradle间接依赖asm，所以我们可以在build.gradle中直接import相关类。

- 修改什么？

当然是修改class文件，那么class文件的路径在哪里？主工程的 build/intermediates/exploded-aar 中包含了库工程aar解压后的内容，有很多jar文件，这些jar文件太过分散不好操作，由于我们使用了multidex，看一下dex任务的输入，发现是主工程的 build/intermediates/multi-dex/common/debug/allclasses.jar 文件，顾名思义，这个文件包含了所有的class文件，我们直接修改这个jar包里的class文件就可以了。

## 代码

将下面这段代码放在主工程的build.gradle就不报错了。在我们的代码中，unifyRImport任务能跑个10秒钟左右，说长不长，说短不短。

```

import org.apache.commons.compress.utils.IOUtils
import org.objectweb.asm.*

import java.util.jar.JarEntry
import java.util.jar.JarFile
import java.util.jar.JarOutputStream
import java.util.zip.ZipEntry

ext {
 dpPackagePrefix = 'com/dianping/'
 libDrawableClass = 'com/dianping/nova/R\$drawable.class'
}

byte[] unifyR(InputStream inputStream, String rootPackagePrefix) {
 ClassReader cr = new ClassReader(inputStream);
 ClassWriter cw = new ClassWriter(cr, 0);
 ClassVisitor cv = new ClassVisitor(Opcode.ASM4, cw) {
 @Override
 public MethodVisitor visitMethod(int access, String name, String desc,
 String signature, String[] exceptions) {
 MethodVisitor mv = cv.visitMethod(access, name, desc, signature, exceptions);
 if (owner.contains(dpPackagePrefix) && owner.contains("R\$") && !owner.contains("R\$drawable")) {
 super.visitFieldInsn(opcode, owner, fName, fDesc);
 } else {
 super.visitFieldInsn(opcode, owner, fName, fDesc);
 }
 }
 return mv;
 };
 cr.accept(cv, 0);
 return cw.toByteArray();
}

```

```

}

afterEvaluate {
 def manifestFile = android.sourceSets.main.manifest.srcFile
 def packageName = new XmlParser().parse(manifestFile).attribute('package')
 def rootPackagePrefix = packageName.replace('.', '/') + '/'
 println packageName
 android.applicationVariants.each { variant ->
 def dx = tasks.findByName("dex${variant.name.capitalize()}")
 def unifyRImport = "unifyRImport${variant.name.capitalize()}"
 task(unifyRImport) << {
 Set<File> inputFiles = dx.inputs.files.files
 inputFiles.each {
 if (it.name.endsWith(".jar")) {
 println it
 JarFile jarFile = new JarFile(it);
 Enumeration enumeration = jarFile.entries();
 File tmpFile = new File(it.getParent() + File.separator + "classes");
 JarOutputStream jarOutputStream = new JarOutputStream(new FileOutputStream(tmpFile));
 while (enumeration.hasMoreElements()) {
 JarEntry jarEntry = (JarEntry) enumeration.nextElement();
 String entryName = jarEntry.getName();
 ZipEntry zipEntry = new ZipEntry(entryName);

 InputStream inputStream = jarFile.getInputStream(jarEntry);
 if (entryName.startsWith(dpPackagePrefix) && entryName.endsWith("$")) {
 if (!entryName.contains("R\\$")) {
 jarOutputStream.putNextEntry(zipEntry);
 jarOutputStream.write(unifyR(inputStream, rootPackagePrefix));
 } else {
 //NovaLib中R$drawable有被反射使用，不删除
 if (entryName.startsWith(rootPackagePrefix) || entryName.endsWith("$")) {
 jarOutputStream.putNextEntry(zipEntry);
 jarOutputStream.write(IOUtils.toByteArray(inputStream));
 }
 }
 } else {
 jarOutputStream.putNextEntry(zipEntry);
 jarOutputStream.write(IOUtils.toByteArray(inputStream));
 }
 jarOutputStream.closeEntry();
 }
 jarOutputStream.close();
 jarFile.close();
 tmpFile.renameTo(it);
 }
 }
 }
 }

 tasks.findByName(unifyRImport).dependsOn dx.taskDependencies.getDependencies()
}

```

```
 dx.dependsOn tasks.findByName(unifyRImport)
 }
}
```

## 坑

我一般不喜欢用坑这个词，感觉遇到坑更多是因为无知，但特马这的确是个坑，在mac上跑的好好的，windows跑不通了，还是报class太多的错误。我一想两个系统文件分隔符不同，不会是路径上出了问题吧，最后竟然发现是 `tmpFile.renameTo(it);` 这行命令没有重命名成功，Google一搜发现遇到这坑的不在少数，a重命名成b，如果b已经存在，mac的做法直接覆盖，windows就会重命名失败。所以最后在rename前面加了一句，`it.delete();`。

## dex任务增量

代码欢乐的跑了几天，有哥们提意见了，什么代码就不改，为什么点击run还要跑一两分钟？在这段时间的背后AS背地里做了什么？数百头母驴为何半夜惨叫？小卖部安全套为何屡遭黑手？女生宿舍内裤为何频频失窃？连环强奸母猪案，究竟是何人所为？老尼姑的门夜夜被敲，究竟是人是鬼？数百只小母狗意外身亡的背后又隐藏着什么？这一切的背后，是人性的扭曲还是道德的沦丧？是性的爆发还是饥渴的无奈？唉，真是崇拜爱哥。

我们可以在AS中看到dex任务又重新跑了一遍，主要时间就花在这上面了。之前的博客讲过，任务增量构建要求输入和输出较上次没有区别，dex重新跑说明输入或者输出有变化，输出是多个dex文件我们没有改动，输入allclasses.jar虽然有更改，但因为源码不变，第二次运行allclasses.jar应该和上次一样的，不应该重新跑啊。比较了两次运行的allclasses.jar的md值，发现还真是不一样啊，看来问题就出在这里了。

## zip哈希

关键是什么前后两次运行allclasses.jar的哈希值不同呢？

话说之前向maven上打包上传aar的时候，发现代码资源都不改动，上传上去的aar哈希值竟然不同，为什么呢？一个简单的a.txt前后zip压缩两次，得到的zip文件哈希值也不同，用beyond compare看了下二进制，还真的不一样。那就去看看[zip算法](#)吧，可以看到header中有时间戳相关的东西，应该就是这导致同样的文件zip压缩后哈希值不同。

jar打包也是用的zip算法，因为第一次运行我们修改了allclasses.jar，导致第二次运行时，某个任务的输出发生了变化，所以会重新运行生成allclasses.jar，前后两次的allclasses.jar哈希值就发生了变化，dex任务就要重新跑了。

## 增量思路

之前的问题，主要还是没有把allclasses.jar及时还原。因为allclasses.jar是dex的输入，所以我们需要在dex之后把allclasses.jar还原，既然需要还原，那就需要在修改allclasses.jar的时候有个备份（classes.bak）。还有个问题，每次unifyRImport任务运行时，都要重新去生成精简后的allclasses.jar，这一步可以加上缓存，根据allclasses.jar的md5值命名缓存文件（.jar.opt），如果有缓存直接复制成allclasses.jar就可以了。

## 代码

Talk is cheap. Show me the code.

```
afterEvaluate {
 def manifestFile = android.sourceSets.main.manifest.srcFile
 def packageName = new XmlParser().parse(manifestFile).attribute('package')
 def rootPackagePrefix = packageName.replace('.', '/') + '/'

 android.applicationVariants.each { variant ->
 def dx = tasks.findByName("dex${variant.name.capitalize()}")
 Set<File> inputFiles = dx.inputs.files.files
 def allClassesJar;
 inputFiles.each {
 if (it.name.endsWith(".jar")) {
 allClassesJar = it;
 }
 }

 if (allClassesJar != null) {
 def unifyRImport = "unifyRImport${variant.name.capitalize()}"
 def bakJar = new File(allClassesJar.getParent(), allClassesJar.name + ".bak")
 task(unifyRImport) << {
 File unifyRJar = new File(allClassesJar.getParent(), "${md5(allClassesJar)}")
 if (!unifyRJar.exists()) {
 allClassesJar.getParentFile().eachFile { file ->
 if (file.name.endsWith(".jar.opt")) {
 file.delete()
 }
 }
 }
 JarFile jarFile = new JarFile(allClassesJar);
 Enumeration enumeration = jarFile.entries();
 JarOutputStream jarOutputStream = new JarOutputStream(new FileOutputStream(bakJar))
 while (enumeration.hasMoreElements()) {
 JarEntry entry = enumeration.nextElement();
 if (entry.getName().endsWith(".jar.opt")) {
 continue;
 }
 jarOutputStream.putNextEntry(entry);
 byte[] buffer = new byte[1024];
 int len;
 while ((len = jarFile.read(buffer)) > 0) {
 jarOutputStream.write(buffer, 0, len);
 }
 jarOutputStream.closeEntry();
 }
 jarOutputStream.close();
 }
 }
 }
}
```

```

 while (enumeration.hasMoreElements()) {
 JarEntry jarEntry = (JarEntry) enumeration.nextElement();
 String entryName = jarEntry.getName();
 ZipEntry zipEntry = new ZipEntry(entryName);

 InputStream inputStream = jarFile.getInputStream(jarEntry);
 if (entryName.startsWith(dpPackagePrefix) && entryName.endsWith(dpPackageSuffix)) {
 if (!entryName.contains("R\\$")) {
 jarOutputStream.putNextEntry(zipEntry);
 jarOutputStream.write(unifyR(inputStream, rootPackageName));
 } else {
 //NovaLib中R$drawable有被反射使用，不删除
 if (entryName.startsWith(rootPackagePrefix) || entryName.endsWith(rootPackageSuffix)) {
 jarOutputStream.putNextEntry(zipEntry);
 jarOutputStream.write(IOUtils.toByteArray(inputStream));
 }
 }
 } else {
 jarOutputStream.putNextEntry(zipEntry);
 jarOutputStream.write(IOUtils.toByteArray(inputStream));
 }
 jarOutputStream.closeEntry();
 }
 jarOutputStream.close();
 jarFile.close();
 }

 if (bakJar.exists()) {
 bakJar.delete();
 }
 allClassesJar.renameTo(bakJar)
 copyFileUsingStream(unifyRJar, allClassesJar)
}

tasks[unifyRImport].dependsOn dx.taskDependencies.getDependencies(dx)
dx.dependsOn tasks[unifyRImport]

//还原allclasses.jar
def assemble = tasks.findByName("assemble${variant.name.capitalize()}")
def restoreClassesJar = "restore${variant.name.capitalize()}"
task(restoreClassesJar) << {
 if (bakJar.exists()) {
 allClassesJar.delete()
 bakJar.renameTo(allClassesJar)
 }
}
tasks[restoreClassesJar].dependsOn dx
assemble.dependsOn tasks[restoreClassesJar]
}
}

```

这次主要修改了afterEvaluate里面的东西，然后新加了自定义的md5和copyFileUsingStream方法，groovy都有些脚本的特性了，获取md5和复制文件还要自己撸，我也是醉了。

## dex增量

至此，Field 65535的问题基本上算是完美解决了。但是你会发现改了一行代码，build的时间还是很久，主要耗时的任务就是dex，这个怎么搞？

两种方案，[Buck](#)和[LayoutCast](#)。Buck是facebook出品的，微信很早就用上了，但有很多规则，侵入性较强，代码改动大。LayoutCast是我司屠大师研发的，对项目改动非常小，应该也有借鉴buck的一些思路。

稍微看了一下buck的思路，buck的dex粒度非常小，每个module都会打成一个dex，最后合并成一个大的dex，修改代码后，只需要重新生成代码所在的dex，然后通过adb传递到手机，动态替换该dex即可，都不需要重新生成apk，也节省了安装的时间。

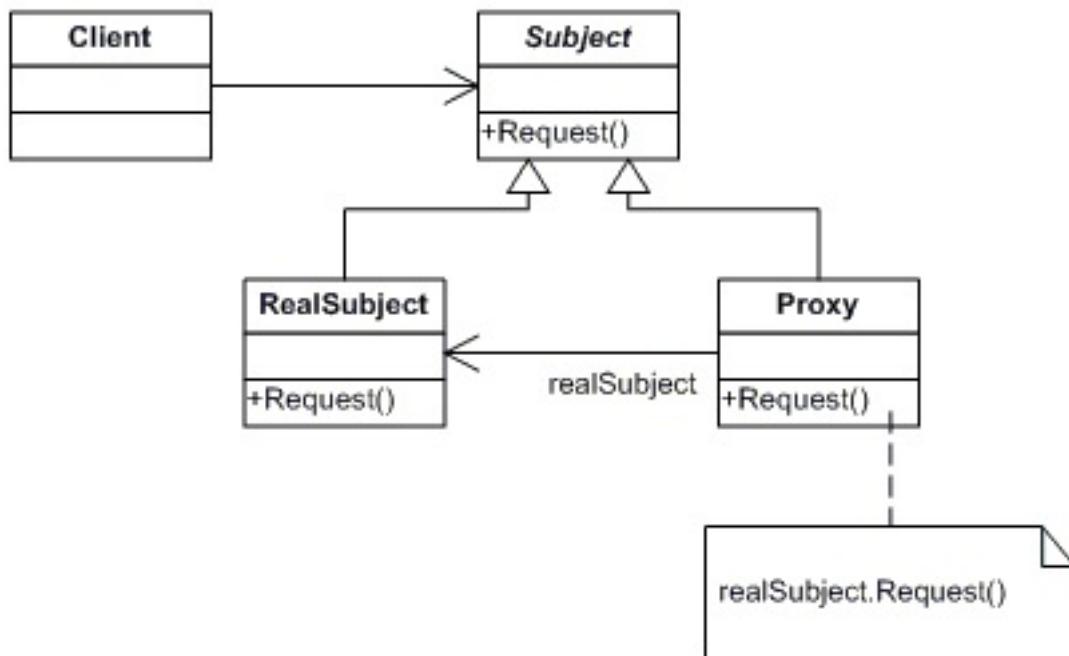
# 基础

# 代理模式及Java实现动态代理

来源:<http://www.jianshu.com/p/6f6bb2f0ece9>

## 代理模式

定义：给某个对象提供一个代理对象，并由代理对象控制对于原对象的访问，即客户不直接操控原对象，而是通过代理对象间接地操控原对象。



在上图中：

- RealSubject 是原对象（本文把原对象称为"委托对象"），Proxy 是代理对象。
- Subject 是委托对象和代理对象都共同实现的接口。
- Request() 是委托对象和代理对象共同拥有的方法。

要理解代理模式很简单，其实生活当中就存在代理模式：

我们购买火车票可以去火车站买，但是也可以去火车票代售处买，此处的火车票代售处就是火车站购票的代理，即我们在代售点发出买票请求，代售点会把请求发给火车站，火车站把购买成功响应发给代售点，代售点再告诉你。但是代售点只能买票，不能退票，而火车站能买票也能退票，因此代理对象支持的操作可能和委托对象的操作有所不同。

再举一个写程序会碰到的一个例子：

如果现在有一个已有项目（你没有源代码，只能调用它）能够调用 int compute(String exp1) 实现对于后缀表达式的计算，你想使用这个项目实现对于中缀表达式的计算，那么你可以写一个代理类，并且其中也定义一个compute(String exp2)，这个exp2参数是中缀表达式，因此你需要在调用已有项目的compute() 之前将中缀表达式转换成后缀表达式(Preprocess)，再调用已有项目的compute()，当然你还可以接收到返回值之后再做些其他操作比如存入文件(Postprocess)，这个过程就是使用了代理模式。

在平时用电脑也会碰到代理模式的应用：

远程代理：我们在国内因为GFW，所以不能访问 facebook，我们可以用翻墙（设置代理）的方法访问。访问过程是：

- (1)用户把HTTP请求发给代理
- (2)代理把HTTP请求发给web服务器
- (3)web服务器把HTTP响应发给代理
- (4)代理把HTTP响应发回给用户

Java 实现上面的UML图的代码（即实现静态代理）为：

```

public class ProxyDemo {
 public static void main(String args[]){
 RealSubject subject = new RealSubject();
 Proxy p = new Proxy(subject);
 p.request();
 }
}

interface Subject{
 void request();
}

class RealSubject implements Subject{
 public void request(){
 System.out.println("request");
 }
}

class Proxy implements Subject{
 private Subject subject;
 public Proxy(Subject subject){
 this.subject = subject;
 }
 public void request(){
 System.out.println("PreProcess");
 subject.request();
 System.out.println("PostProcess");
 }
}

```

代理的实现分为：

- **静态代理**：代理类是在编译时就实现好的。也就是说 Java 编译完成后代理类是一个实际的 class 文件。
- **动态代理**：代理类是在运行时生成的。也就是说 Java 编译完之后并没有实际的 class 文件，而是在运行时动态生成的类字节码，并加载到JVM中。

## Java 实现动态代理

在前一节我们已经介绍了静态代理（第一节已经实现）和动态代理，那么在 Java 中是如何实现动态代理，即如何做到在运行时动态的生成代理类呢？

首先先说明几个词：

- 委托类和委托对象：委托类是一个类，委托对象是委托类的实例。
- 代理类和代理对象：代理类是一个类，代理对象是代理类的实例。

Java实现动态代理的大致步骤如下：

- 1、定义一个委托类和公共接口。
- 2、自己定义一个类（调用处理器类，即实现 InvocationHandler 接口），这个类的目的是指定运行时将生成的代理类需要完成的具体任务（包括Preprocess和Postprocess），即代理类调用任何方法都会经过这个调用处理器类（在本文最后一节对此进行解释）。
- 3、生成代理对象（当然也会生成代理类），需要为他指定(1)委托对象(2)实现的一系列接口(3)调用处理器类的实例。因此可以看出一个代理对象对应一个委托对象，对应一个调用处理器实例。

Java 实现动态代理主要涉及以下几个类：

- `java.lang.reflect.Proxy`: 这是生成代理类的主类，通过 `Proxy` 类生成的代理类都继承了 `Proxy` 类，即 `DynamicProxyClass extends Proxy`。
- `java.lang.reflect.InvocationHandler`: 这里称他为"调用处理器"，他是一个接口，我们动态生成的代理类需要完成的具体内容需要自己定义一个类，而这个类必须实现 `InvocationHandler` 接口。`Proxy` 类主要方法为：

```
//创建代理对象
static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHan
```

这个静态函数的第一个参数是类加载器对象（即哪个类加载器来加载这个代理类到 JVM 的方法区），第二个参数是接口（表明你这个代理类需要实现哪些接口），第三个参数是调用处理器类实例（指定代理类中具体要干什么）。这个函数是 JDK 为了程序员方便创建代理对象而封装的一个函数，因此你调用`newProxyInstance()`时直接创建了代理对象（略去了创建代理类的代码）。其实他主要完成了以下几个工作：

```
static Object newProxyInstance(ClassLoader loader,
 Class<?>[] interfaces, InvocationHandler handler){
 //1. 根据类加载器和接口创建代理类
 Class clazz = Proxy.getProxyClass(loader, interfaces);
 //2. 获得代理类的带参数的构造函数
 Constructor constructor = clazz.getConstructor(new Class[] { InvocationHandler.class });
 //3. 创建代理对象，并制定调用处理器实例为参数传入
 Interface Proxy = (Interface)constructor.newInstance(new Object[] { handler });
}
```

`Proxy` 类还有一些静态方法，比如：

- `InvocationHandler getInvocationHandler(Object proxy)`: 获得代理对象对应的调用处理器对象。
- `Class getProxyClass(ClassLoader loader, Class[] interfaces)`: 根据类加载器和实现的接口获得代理类。

Proxy 类中有一个映射表，映射关系为：`(,(,))`，可以看出一级key为类加载器，根据这个一级key获得二级映射表，二级key为接口数组，因此可以看出：一个类加载器对象和一个接口数组确定了一个代理类。

我们写一个简单的例子来阐述 Java 实现动态代理的整个过程：

```

public class DynamicProxyDemo01 {
 public static void main(String[] args) {
 RealSubject realSubject = new RealSubject(); //1.创建委托对象
 ProxyHandler handler = new ProxyHandler(realSubject); //2.创建调用处理器对象
 Subject proxySubject = (Subject)Proxy.newProxyInstance(RealSubject.class.get
 RealSubject.class.getInterface
 proxySubject.request(); //4.通过代理对象调用方法
 }
}

/**
 * 接口
 */
interface Subject{
 void request();
}

/**
 * 委托类
 */
class RealSubject implements Subject{
 public void request(){
 System.out.println("====RealSubject Request====");
 }
}
/**
 * 代理类的调用处理器
 */
class ProxyHandler implements InvocationHandler{
 private Subject subject;
 public ProxyHandler(Subject subject){
 this.subject = subject;
 }
 @Override
 public Object invoke(Object proxy, Method method, Object[] args)
 throws Throwable {
 System.out.println("====before===="); //定义预处理的工作，当然你也可以根据 method 的
 Object result = method.invoke(subject, args);
 System.out.println("====after====");
 return result;
 }
}

```

InvocationHandler 接口中方法：

```
invoke(Object proxy, Method method, Object[] args)
```

这个函数是在代理对象调用任何一个方法时都会调用的，方法不同会导致第二个参数 method 不同，第一个参数是代理对象（表示哪个代理对象调用了 method 方法），第二个参数是 Method 对象（表示哪个方法被调用了），第三个参数是指定调用方法的参数。

动态生成的代理类具有几个特点：

- 继承 Proxy 类，并实现了在 Proxy.newProxyInstance() 中提供的接口数组。
- public final。
- 命名方式为 \$ProxyN，其中 N 会慢慢增加，一开始是 \$Proxy1，接下来是 \$Proxy2...
- 有一个参数为 InvocationHandler 的构造函数。这个从 Proxy.newProxyInstance() 函数内部的 clazz.getConstructor(new Class[] { InvocationHandler.class }) 可以看出。

Java 实现动态代理的缺点：因为 Java 的单继承特性（每个代理类都继承了 Proxy 类），只能针对接口创建代理类，不能针对类创建代理类。

不难发现，代理类的实现是有很多共性的（重复代码），动态代理的好处在于避免了这些重复代码，只需要关注操作。

## Java 动态代理的内部实现

现在我们就会有一个问题：Java 是怎么保证代理对象调用的任何方法都会调用 InvocationHandler 的 invoke() 方法的？

这就涉及到动态代理的内部实现。假设有一个接口 Subject，且里面有 int request(int i) 方法，则生成的代理类大致如下：

```
public final class $Proxy1 extends Proxy implements Subject{
 private InvocationHandler h;
 private $Proxy1(){}
 public $Proxy1(InvocationHandler h){
 this.h = h;
 }
 public int request(int i){
 Method method = Subject.class.getMethod("request", new Class[]{int.class});
 return (Integer)h.invoke(this, method, new Object[]{new Integer(i)}); //调用了
 }
}
```

通过上面的方法就成功调用了 invoke() 方法。

## 元编程

到最后，我还想介绍一下元编程，我是从《MacTalk·人生元编程》中了解到元编程这个词的：

元编程就是能够操作代码的代码

Java 支持元编程因为反射、动态代理、内省等特性。

## 参考文献

[Java 动态代理机制分析及扩展，第 1 部分](#)

[动态代理的作用是什么？](#)

[JDK动态代理实现原理](#)

[彻底理解JAVA动态代理](#)

[慕课网：模式的秘密---代理模式](#)

# 第 16 章 反射 (Reflection)

来源:<https://github.com/JustinSDK/JavaSE6Tutorial/blob/master/docs/CH16.md>

Java 提供的反射機制允許您於執行時期動態載入类、检查类信息、生成对象或操作生成的对象，要舉反射機制的一個應用實例，就是在整合式開發環境中所提供的方法提示或是类检查工具，另外像 JSP 中的 JavaBean 自動收集請求信息也使用到反射，而一些软件開發框架 (Framework) 也常見到反射機制的使用，以達到動態載入使用者自訂类的目的。

即使您暫時用不到反射機制，也建議您花時間看看這個章節，藉由對反射機制的認識，您可以瞭解 Java 中是如何載入类的，而且瞭解到每個被載入的类在 JVM 中，都以 Class 类的一個實例存在的事實。

## 16.1 类載入與檢查

即使您拿到一個类並對它一無所知，但其實它本身就包括了許多信息，Java 在需要使用到某個类時才會將类載入，並在 JVM 中以一個 `java.lang.Class` 的實例存在，從 `Class` 實例開始，您可以獲得类的許多訊息。

### 16.1.1 簡介 Class 與类載入

Java 在真正需要使用一個类時才會加以載入，而不是在程序啟動時就載入所有的类，因為大多數的使用者都只使用到應用程序的部份資源，在需要某些功能時才載入某些資源，可以讓系統的資源運用更有效率 (Java 本來就是為了資源有限的小型設備而設計的，這樣的考量是必然的) 。

一個 `java.lang.Class` 对象代表了 Java 應用程序在運行時所載入的类或接口實例，也用來表達 `enum` (屬於类的一種) 、`annotation` (屬於接口的一種) 、数组、原生型態 (Primitive type) 、`void`；`Class` 类沒有公開的 (public) 构造方法，`Class` 对象是由 JVM 自動產生，每當一個类被載入時，JVM 就自動為其生成一個 `Class` 对象。

您可以透過 `Object` 的 `getClass()` 方法來取得每一個对象對應的 `Class` 对象，或者是透過 "class" 常量 (Class literal)，在取得 `Class` 对象之後，您就可以操作 `Class` 对象上的一些公開方法來取得类的基本信息，範例 16.1 簡單的使用 `getClass()` 方法來取得 `String` 类的 `Class` 實例，並從中得到 `String` 的一些基本信息。

## 範例 16.1 ClassDemo.java

```
package onlyfun.caterpillar;

public class ClassDemo {
 public static void main(String[] args) {
 String name = "caterpillar";
 Class stringClass = name.getClass();
 System.out.println("类名稱: " +
 stringClass.getName());
 System.out.println("是否為接口: " +
 stringClass.isInterface());
 System.out.println("是否為基本型態: " +
 stringClass.isPrimitive());
 System.out.println("是否為数组对象: " +
 stringClass.isArray());
 System.out.println("父类名稱: " +
 stringClass.getSuperclass().getName());
 }
}
```

執行結果：

```
类名稱: java.lang.String
是否為接口: false
是否為基本型態: false
是否為数组对象: false
父类名稱: java.lang.Object
```

您也可以直接使用以下的方式來取得 String 类的 Class 对象：

```
Class stringClass = String.class;
```

Java 在真正需要类時才會載入类，所謂「真正需要」通常指的是要使用指定的类生成对象時（或是使用者指定要載入类時，例如使用 `Class.forName()` 載入类，或是使用 `ClassLoader` 的 `loadClass()` 載入类，稍後都會說明）。使用类名稱來宣告參考名稱並不會導致类的載入，可以設計一個測試类的印證這個說法。

## 範例 16.2 TestClass.java

```
package onlyfun.caterpillar;

public class TestClass {
 static {
 System.out.println("类被载入");
 }
}
```

在範例中定義了一個靜態區塊，「預設」在類第一次被載入時會執行靜態區塊（說預設的原因，是因為可以設定載入類時不執行靜態區塊，使用 Class 生成對象時才執行靜態區塊，稍後會介紹），藉由在文字模式下顯示訊息，您可以瞭解類何時被載入，可以使用範例 16.3 來測試類載入時機。

### 範例 16.3 LoadClassTest.java

```
package onlyfun.caterpillar;

public class LoadClassTest {
 public static void main(String[] args) {
 TestClass test = null;
 System.out.println("宣告TestClass參考名稱");
 test = new TestClass();
 System.out.println("生成TestClass實例");
 }
}
```

執行結果：

```
宣告TestClass參考名稱
类被载入
生成TestClass實例
```

從執行結果中可以看出，宣告參考名稱並不導致 TestClass 類被載入，而是在使用 "new" 生成對象時才會載入類。

Class 的訊息是在編譯時期就被加入至 .class 檔案中，這是 Java 支援執行時期型別辨識 (RTTI, Run-Time Type Information 或 Run-Time Type Identification) 的一種方式，在編譯時期編譯器會先檢查對應的 .class 檔案，而執行時期 JVM 在使用某類時，會先檢查對應的 Class 對象是否已經載入，如果沒有載入，則會尋找對應的 .class 檔案並載入，一個類在 JVM 中只會有一個 Class 實例，每個類的實例都會記得自己是由哪個 Class 實例所生成，您可以使用 getClass() 或 .class 來取得 Class 實例。

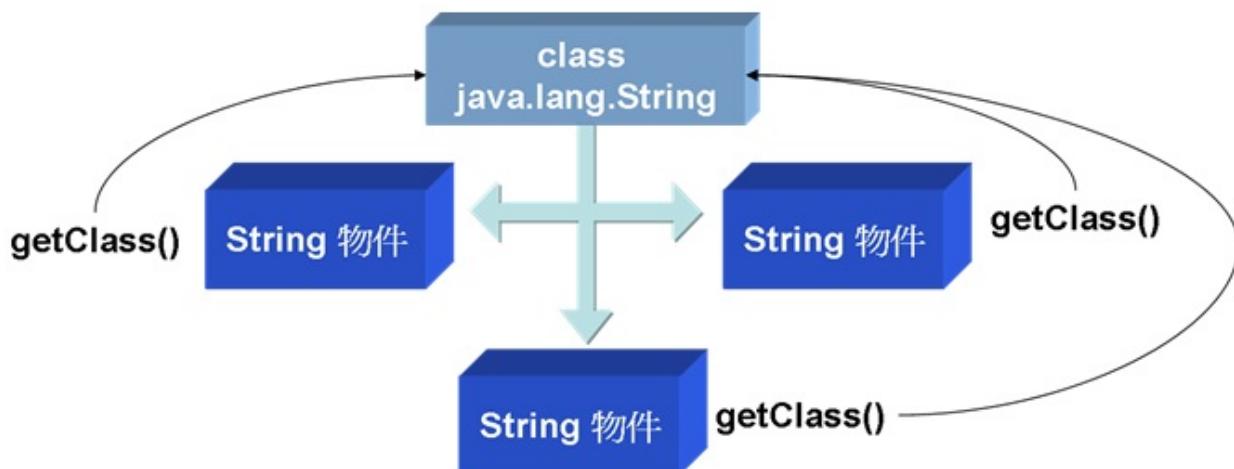


圖 16.1 每個对象會記得生成它的 Class 實例

在 Java 中，数组也是一个对象，也有其對應的 Class 實例，這個对象是由具相同元素與維度的数组所共用，而基本型態像是 boolean, byte, char, short, int, long, float, double 以及關鍵字 void，也都有對應的 Class 对象，您可以用类常量（Class literal）來取得這些对象。

#### 範例 16.4 ClassDemo2.java

```
package onlyfun.caterpillar;

public class ClassDemo2 {
 public static void main(String[] args) {
 System.out.println(boolean.class);
 System.out.println(void.class);

 int[] iarr = new int[10];
 System.out.println(iarr.getClass().toString());

 double[] darr = new double[10];
 System.out.println(darr.getClass().toString());
 }
}
```

執行結果：

```
boolean
void
class [I
class [D
```

在 Java 中数组確實是以对象的形式存在，其對應的类是由 JVM 自動生成，當您使用 `toString()` 來顯示数组对象的描述時，[表示為数组型態，並加上一個型態代表字，範例中表示是一個 int 数组，而 D 表示是一個 double 数组，16.2.4 還會對数组对象加以討論。

## 16.1.2 使用 `Class.forName()` 載入类

在一些應用中，您無法事先知道使用者將載入什麼类，而必須讓使用者指定类名稱以載入类，您可以使用 `Class` 的靜態 `forName()` 方法實現動態加載类，範例 16.5 是個簡單示範，可以讓您可以指定类名稱來獲得类的相關信息。

### 範例 16.5 ForNameDemo.java

```
package onlyfun.caterpillar;

public class ForNameDemo {
 public static void main(String[] args) {
 try {
 Class c = Class.forName(args[0]);
 System.out.println("类名稱: " +
 c.getName());
 System.out.println("是否為接口: " +
 c.isInterface());
 System.out.println("是否為基本型態: " +
 c.isPrimitive());
 System.out.println("是否為数组: " + c.isArray());
 System.out.println("父类: " +
 c.getSuperclass().getName());
 }
 catch(ArrayIndexOutOfBoundsException e) {
 System.out.println("沒有指定类名稱");
 }
 catch(ClassNotFoundException e) {
 System.out.println("找不到指定的类");
 }
 }
}
```

在指定类給 `forName()` 方法後，如果找不到指定的类，會丟出 `ClassNotFoundException` 例外，一個的執行結果如下：

```
java onlyfun.caterpillar.ForNameDemo java.util.Scanner
类名稱: java.util.Scanner
是否為接口: false
是否為基本型態: false
是否為数组: false
父类: java.lang.Object
```

Class 的靜態 forName() 方法有兩個版本，範例16.5所示範的是只指定類名稱的版本，而另一個版本可以讓您指定類名稱、載入類時是否執行靜態區塊、指定類載入器（Class loader）：

```
static Class forName(String name, boolean initialize, ClassLoader loader)
```

之前曾經說過，預設上在載入類的時候，如果類中有定義靜態區塊則會執行它，您可以使用 forName() 的第二個版本，將 initialize 設定為 false，如此在載入類時並不會馬上執行靜態區塊，而會在使用類建立對象時才執行靜態區塊，為了印證，您可以先設計一個測試類。

## 範例 16.6 TestClass2.java

```
package onlyfun.caterpillar;

public class TestClass2 {
 static {
 System.out.println("[執行靜態區塊]");
 }
}
```

範例 16.6 中只定義了靜態區塊顯示一段訊息，以觀察靜態區塊何時被執行，您可以設計範例 16.7 使用第一個版本的 forName() 方法。

## 範例 16.7 ForNameDemoV1.java

```
package onlyfun.caterpillar;

public class ForNameDemoV1 {
 public static void main(String[] args) {
 try {
 System.out.println("載入TestClass2");
 Class c = Class.forName("onlyfun.caterpillar.TestClass2");

 System.out.println("使用TestClass2宣告參考名稱");
 TestClass2 test = null;

 System.out.println("使用TestClass2建立对象");
 test = new TestClass2();
 }
 catch(ClassNotFoundException e) {
 System.out.println("找不到指定的类");
 }
 }
}
```

執行結果如下：

```
載入TestClass2
[執行靜態區塊]
使用TestClass2宣告參考名稱
使用TestClass2建立对象
```

從執行結果中可以看到，第一個版本的 `forName()` 方法在載入類之後，預設會馬上執行靜態區塊，來看看範例 16.8 中使用第二個版本的 `forName()` 方法會是如何。

## 範例 16.8 ForNameDemoV2.java

```

package onlyfun.caterpillar;

public class ForNameDemoV2 {
 public static void main(String[] args) {
 try {
 System.out.println("載入TestClass2");
 Class c = Class.forName(
 "onlyfun.caterpillar.TestClass2",
 false, // 載入类時不執行靜態方法
 Thread.currentThread().getContextClassLoader());
 }

 System.out.println("使用TestClass2宣告參考名稱");
 TestClass2 test = null;

 System.out.println("使用TestClass2建立对象");
 test = new TestClass2();
 }
 catch(ClassNotFoundException e) {
 System.out.println("找不到指定的类");
 }
}
}

```

執行結果如下：

```

載入TestClass2
使用TestClass2宣告參考名稱
使用TestClass2建立对象
[執行靜態區塊]

```

由於使用第二個版本的 `forName()` 方法時，設定 `initialize` 為 `false`，所以載入类時並不會馬上執行靜態區塊，而會在使用类建立对象時才去執行靜態區塊，第二個版本的 `forName()` 方法會需要一個类載入器（Class loader），範例中所使用的是主執行緒的类載入器，16.1.4 還會詳細介紹 Java 中的类載入器機制。

### 16.1.3 從 Class 中獲取信息

Class 对象表示所載入的类，取得 Class 对象之後，您就可以取得與类相關聯的信息，像是包（package）（別忘了 package 也是类名稱的一部份）、构造方法、方法成員、資料成員等的訊息，而每一個訊息，也會有相應的类型態，例如包的對應型態是 `java.lang.Package`，构造方法的對應型態是 `java.lang.reflect.Constructor`，方法成員的對應型態是 `java.lang.reflect.Method`，資料成員的對應型態是 `java.lang.reflect.Field` 等。

來看個簡單的示範，範例 16.9 可以讓您取得所指定类上的包名稱。

## 範例 16.9 ClassInfoDemo.java

```
package onlyfun.caterpillar;

public class ClassInfoDemo {
 public static void main(String[] args) {
 try {
 Class c = Class.forName(args[0]);
 Package p = c.getPackage();
 System.out.println(p.getName());
 }
 catch(ArrayIndexOutOfBoundsException e) {
 System.out.println("沒有指定类");
 }
 catch(ClassNotFoundException e) {
 System.out.println("找不到指定类");
 }
 }
}
```

執行結果：

```
java onlyfun.caterpillar.ClassInfoDemo java.util.ArrayList
java.util
```

您可以分別取回 Field、Constructor、Method等对象，分別代表資料成員、构造方法與方法成員，範例16.10 簡單的實作了取得类基本信息的程序。

## 範例 16.10 SimpleClassViewer.java

```
package onlyfun.caterpillar;

import java.lang.reflect.*;

public class SimpleClassViewer {
 public static void main(String[] args) {
 try {
 Class c = Class.forName(args[0]);
 // 取得包代表对象
 Package p = c.getPackage();

 System.out.printf("package %s;%n", p.getName());

 // 取得型態修飾, 像是class、interface
 int m = c.getModifiers();
 }
 }
}
```

```
System.out.print(Modifier.toString(m) + " ");
// 如果是接口
if(Modifier.isInterface(m)) {
 System.out.print("interface ");
}
else {
 System.out.print("class ");
}

System.out.println(c.getName() + " {");

// 取得宣告的資料成員代表对象
Field[] fields = c.getDeclaredFields();
for(Field field : fields) {
 // 顯示權限修飾, 像是public、protected、private
 System.out.print("\t" +
 Modifier.toString(field.getModifiers()));
 // 顯示型態名稱
 System.out.print(" " +
 field.getType().getName() + " ");
 // 顯示資料成員名稱
 System.out.println(field.getName() + ";");
}

// 取得宣告的构造方法代表对象
Constructor[] constructors =
 c.getDeclaredConstructors();
for(Constructor constructor : constructors) {
 // 顯示權限修飾, 像是public、protected、private
 System.out.print("\t" +
 Modifier.toString(
 constructor.getModifiers()));
 // 顯示构造方法名稱
 System.out.println(" " +
 constructor.getName() + "();");
}

// 取得宣告的方法成員代表对象
Method[] methods = c.getDeclaredMethods();
for(Method method : methods) {
 // 顯示權限修飾, 像是public、protected、private
 System.out.print("\t" +
 Modifier.toString(
 method.getModifiers()));
 // 顯示返回值型態名稱
 System.out.print(" " +
 method.getReturnType().getName() + " ");
 // 顯示方法名稱
 System.out.println(method.getName() + "();");
}
System.out.println("}");

}

catch(ArrayIndexOutOfBoundsException e) {
```

```
 System.out.println("沒有指定类");
 }
 catch(ClassNotFoundException e) {
 System.out.println("找不到指定类");
 }
}
```

## 執行結果：

```
package java.util;
public class java.util.ArrayList {
 private static final long serialVersionUID;
 private transient [Ljava.lang.Object; elementData;
 private int size;
 public java.util.ArrayList();
 public java.util.ArrayList();
 public java.util.ArrayList();
 public boolean add();
 public void add();
 public java.lang.Object clone();
 public void clear();
 public boolean contains();
 public int indexOf();
 略...
}
```

一些类检查器的實作原理基本上就是範例 16.10 所示範的，當然還可以取得更多的信息，您可以參考 Class 的線上 API 文件得到更多的訊息。

#### 16.1.4 簡介类載入器

Java 在需要使用類的時候，才會將類載入，Java 的類載入是由類載入器（Class loader）來達到的。

當您在文字模式下執行 `java XXX` 指令後，`java` 執行程序會嘗試找到 JRE 安裝的所在目錄，然後尋找 `jvm.dll`（預設是在JRE目錄下`bin\client`目錄中），接著啟動 JVM 並進行初始化動作，接著產生 Bootstrap Loader，Bootstrap Loader 會載入 Extended Loader，並設定 Extended Loader 的 parent 為 Bootstrap Loader，接著 Bootstrap Loader 會載入 System Loader，並將 System Loader 的 parent 設定為 Extended Loader。

Bootstrap Loader 通常由 C 撰寫而成； Extended Loader 是由 Java 所撰寫而成，實際是對應於 sun.misc.Launcher\$ExtClassLoader (Launcher 中的內部類)； System Loader 是由 Java 撰寫而成，實際對應於 sun.misc.Launcher\$AppClassLoader (Launcher 中的

內部类)。

圖 16.2 是 java 程序啟動與載入類的順序圖，也就是所謂的「類載入器階層架構」。

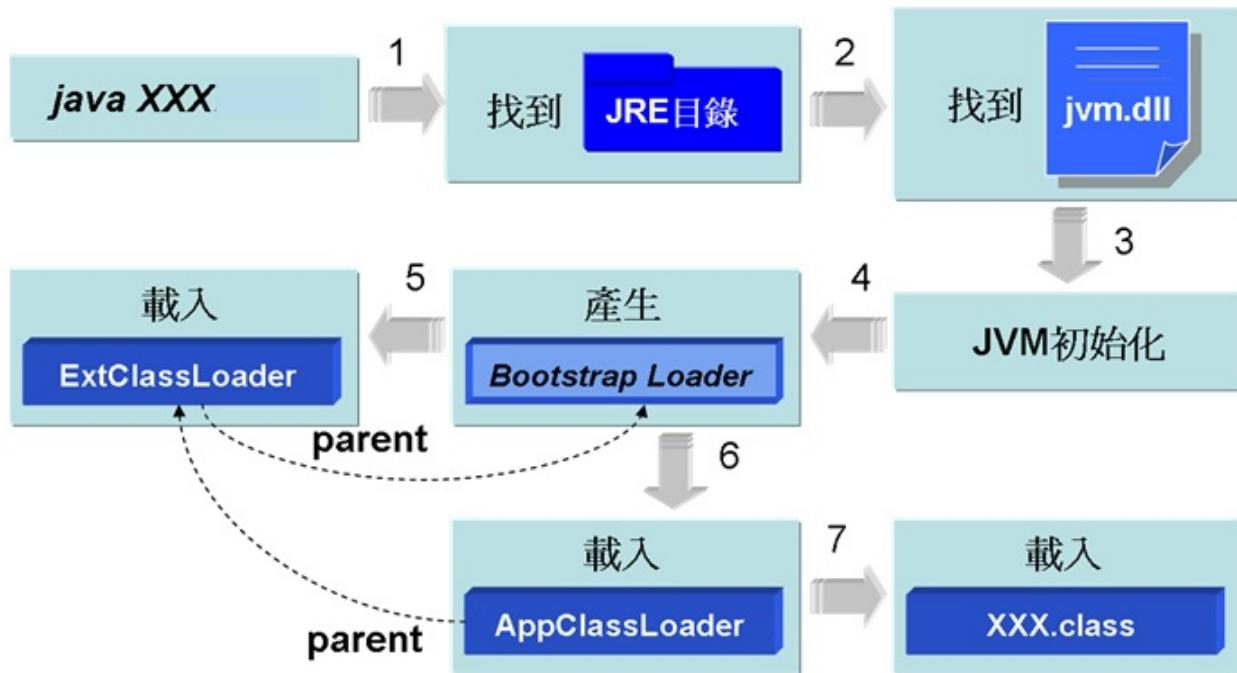


圖 16.2 Java 類載入器階層架構

Bootstrap Loader 會搜尋系統參數 `sun.boot.class.path` 中指定位置的類，預設是 JRE classes 下之 檔案，或 lib 目錄下 .jar 檔案中（例如 rt.jar）的類並載入，您可以使用 `System.getProperty("sun.boot.class.path")` 陳述來顯示 `sun.boot.class.path` 中指定的路徑，例如在我的電腦中顯示的是以下的路徑：

```
C:\Program Files\Java\jre1.5.0_03\lib\rt.jar;
C:\Program Files\Java\jre1.5.0_03\lib\i18n.jar;
C:\Program Files\Java\jre1.5.0_03\lib\sunrsasign.jar;
C:\Program Files\Java\jre1.5.0_03\lib\jsse.jar;
C:\Program Files\Java\jre1.5.0_03\lib\jce.jar;
C:\Program Files\Java\jre1.5.0_03\lib\charsets.jar;
C:\Program Files\Java\jre1.5.0_03\classes
```

Extended Loader (`sun.misc.Launcher$ExtClassLoader`) 是由 Java 撰寫而成，會搜尋系統參數 `java.ext.dirs` 中指定位置的類，預設是 JRE 目錄下的 `lib\ext\classes` 目錄下的 .class 檔案，或 `lib\ext` 目錄下的 .jar 檔案中（例如 rt.jar）的類並載入，您可以使用 `System.getProperty("java.ext.dirs")` 陳述來顯示 中指定的路徑，例如在我的電腦中顯示的是以下的路徑：

```
C:\Program Files\Java\jre1.5.0_03\lib\ext
```

System Loader (sun.misc.Launcher\$AppClassLoader) 是由 Java 撰寫而成，會搜尋系統參數 `java.class.path` 中指定位置的類，也就是 Classpath 所指定的路徑，預設是目前工作路徑下的 `.class` 檔案，您可以使用 `System.getProperty("java.class.path")` 陳述來顯示 `java.class.path` 中指定的路徑，在使用 java 執行程序時，您也可以加上 `-cp` 來覆蓋原有的 Classpath 設定，例如：

```
java -cp ./classes SomeClass
```

Bootstrap Loader 會在 JVM 啟動之後產生，之後它會載入 Extended Loader 並將其 parent 設為 Bootstrap Loader，然後 Bootstrap Loader 再載入 System Loader 並將其 parent 設定為 ExtClassLoader，接著 System Loader 開始載入您指定的類，在載入類時，每個類載入器會先將載入類的任務交由其 parent，如果 parent 找不到，才由自己負責載入，所以在載入類時，會以 Bootstrap Loader→Extended Loader→System Loader 的順序來尋找類，如果都找不到，就會丟出 `NoClassDefFoundError`。

類載入器在 Java 中是以 `java.lang.ClassLoader` 型態存在，每一個類被載入後，都會有一個 `Class` 的實例來代表，而每個 `Class` 的實例都會記得自己是由哪個 `ClassLoader` 載入的，可以由 `Class` 的 `getClassLoader()` 取得載入該類的 `ClassLoader`，而從 `ClassLoader` 的 `getParent()` 方法可以取得自己的 parent，圖 16.3 顯示了一個自訂的 `SomeClass` 實例與 `Class`、`ClassLoader` 及各 parent 的關係。

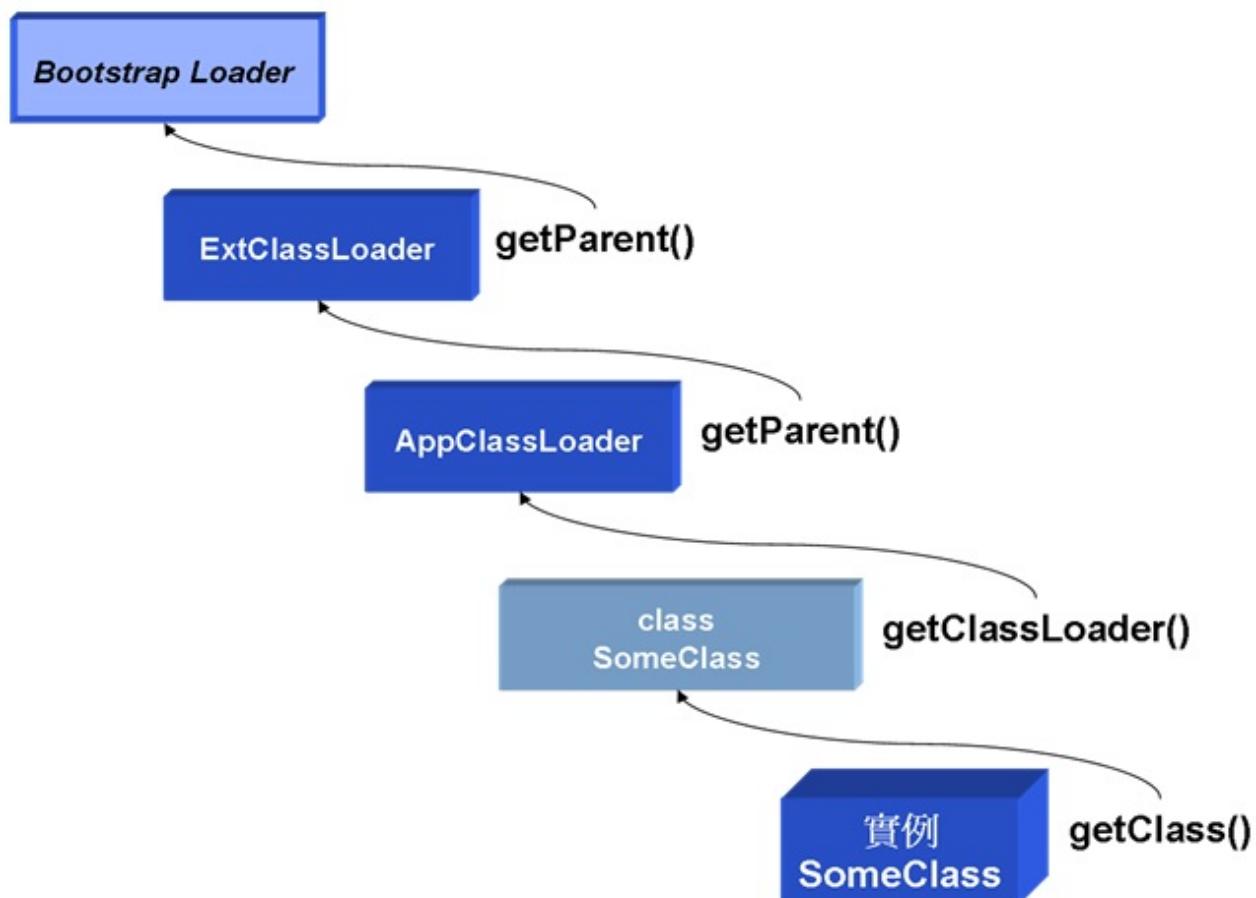


圖 16.3 對象、Class、ClassLoader 與 parent 的關係

範例 16.11 示範了圖 16.3 的一個實際例子。

## 範例 16.11 SomeClass.java

```
package onlyfun.caterpillar;

public class SomeClass {
 public static void main(String[] args) {
 // 建立SomeClass實例
 SomeClass some = new SomeClass();
 // 取得SomeClass的Class實例
 Class c = some.getClass();
 // 取得ClassLoader
 ClassLoader loader = c.getClassLoader();
 System.out.println(loader);
 // 取得父ClassLoader
 System.out.println(loader.getParent());
 // 再取得父ClassLoader
 System.out.println(loader.getParent().getParent());
 }
}
```

執行結果：

```
sun.misc.Launcher$AppClassLoader@82ba41
sun.misc.Launcher$ExtClassLoader@923e30
null
```

onlyfun.caterpillar.SomeClass 是個自訂類，您在目前的工作目錄下執行程序，首先 AppClassLoader 會將載入類的任務交給 ExtClassLoader，而 ExtClassLoader 會將載入類的任務交給 Bootstrap Loader，由於 Bootstrap Loader 在它的路徑設定 (sun.boot.class.path) 下找不到類，所以由 ExtClassLoader 來試著尋找，而 ExtClassLoader 在它的路徑設定 (java.ext.dirs) 下也找不到類，所以由 AppClassLoader 來試著尋找，AppClassLoader 最後在 Classpath (java.class.path) 設定下找到指定的類並載入。

在執行結果中可以看到，載入 SomeClass 的 ClassLoader 是 AppClassLoader，而 AppClassLoader 的 parent 是 ExtClassLoader，而 ExtClassLoader 的 parent 是 null，null 並不是表示 ExtClassLoader 沒有設定 parent，而是因為 Bootstrap Loader 通常由 C 所撰寫而成，在 Java 中並沒有一個實際的類來表示它，所以才會顯示為 null。

如果把 SomeClass 的 .class 檔案移至 JRE 目錄下的 lib\ext\classes 下（由於設定了包，所以實際上 SomeClass.class 要放置在 JRE 目錄下的 lib\ext\classes\onlyfun\caterpillar 下），並重新（於任何目錄下）執行程序，您會看到以下的訊息：

```
sun.misc.Launcher$ExtClassLoader@923e30
null
Exception in thread "main" java.lang.NullPointerException
 at onlyfun.caterpillar.SomeClass.main(SomeClass.java:15)
```

由於 SomeClass 這次可以在 ExtClassLoader 的設定路徑下找到，所以會由 ExtClassLoader 來載入 SomeClass 類，而 ExtClassLoader 的 parent 顯示為 null，指的是它的 parent 是由 C 撰寫而成的 Bootstrap Loader，因為沒有實際的 Java 類而表示為 null，所以再由 null 上嘗試調用 getParent() 方法就會丟出 NullPointerException 例外。

如果再把 SomeClass 的 .class 檔案移至 JRE 目錄下的 classes 目錄下（由於設定了包，所以實際上 SomeClass.class 要放置在 JRE 目錄下的 classes\onlyfun\caterpillar 下），並重新（於任何目錄下）執行程序，您會看到以下的訊息：

```
null
Exception in thread "main" java.lang.NullPointerException
 at onlyfun.caterpillar.SomeClass.main(SomeClass.java:13)
```

由於 SomeClass 這次可以在 Bootstrap Loader 的設定路徑下找到，所以會由 Bootstrap Loader 來載入 SomeClass 類，Bootstrap Loader 通常由 C 撰寫而成，在 Java 中沒有一個實際的類來表示，所以顯示為 null，因為表示為 null，所以再由 null 上嘗試調用 getParent() 方法就會丟出 NullPointerException 例外。

取得 ClassLoader 的實例之後，您可以使用它的 loadClass() 方法來載入類，使用 loadClass() 方法載入別時，不會執行靜態區塊，靜態區塊的執行會等到真正使用類來建立實例時，例如您可以改寫範例 16.7 為範例 16.12。

## 範例 16.12 ForNameDemoV3.java

```

package onlyfun.caterpillar;

public class ForNameDemoV3 {
 public static void main(String[] args) {
 try {
 System.out.println("載入TestClass2");
 ClassLoader loader = ForNameDemoV3.class.getClassLoader();
 Class c = loader.loadClass("onlyfun.caterpillar.TestClass2");

 System.out.println("使用TestClass2宣告參考名稱");
 TestClass2 test = null;

 System.out.println("使用TestClass2建立对象");
 test = new TestClass2();
 }
 catch(ClassNotFoundException e) {
 System.out.println("找不到指定的类");
 }
 }
}

```

從執行結果中可以看到，`loadClass()` 不會在載入類時執行靜態區塊，而會在使用類新建對象時才執行靜態區塊，結果如下所示：

```

載入TestClass2
使用TestClass2宣告參考名稱
使用TestClass2建立对象
[執行靜態區塊]

```

## 16.1.5 使用自己的 ClassLoader

`ExtClassLoader` 與 `AppClassLoader` 都是 `java.net.URLClassLoader` 的子類，您可以在使用 `java` 啟動程序時，使用以下的指令來指定 `ExtClassLoader` 的搜尋路徑：

```
java -Djava.ext.dirs=c:\workspace\ YourClass
```

可以在使用 `java` 啟動程序時，使用 `-classpath` 或 `-cp` 來指定 `AppClassLoader` 的搜尋路徑，也就是設定 Classpath：

```
java -classpath c:\workspace\ YourClass
```

ExtClassLoader 與 AppClassLoader 在程序啟動後會在虛擬機器中存在一份，您在程序運行過程中就無法再改變它的搜尋路徑，如果在程序運行過程中，打算動態決定從其它的路徑載入類，就要產生新的類載入器。

您可以使用 URLClassLoader 來產生新的類載入器，它需要 java.net.URL 作為其參數來指定類載入的搜尋路徑，例如：

```
URL url = new URL("file:/d:/workspace/");
ClassLoader urlClassLoader =
 new URLClassLoader(new URL[] {url});
Class c = urlClassLoader.loadClass("SomeClass");
```

由於 ClassLoader 是 Java SE 的標準API之一，可以在 rt.jar 中找到，因而會由 Bootstrap Loader 來載入 ClassLoader 類，在新增了 ClassLoader 實例後，您可以使用它的 loadClass() 方法來指定要載入的類名稱，在新增 ClassLoader 時，會自動將新建的 ClassLoader 的 parent 設定為 AppClassLoader，並在每次載入類時，先委託 parent 代為搜尋，所以上例中搜尋 SomeClass 類時，會一路往上委託至 Bootstrap Loader 先開始搜尋，接著是 ExtClassLoader、AppClassLoader，如果都找不到，才使用新建的 ClassLoader 搜尋。

Java 的類載入器階層架構除了可以達到動態載入類目的之外，還有著安全上的考量，首先，因為每次尋找類時都是委託 parent 開始尋找，所以除非有人可以侵入您的電腦，置換掉標準 Java SE API 與您自己安裝的延伸包，否則是不可能藉由撰寫自己的類載入器來載入惡意類，以置換掉標準 Java SE API 與您自己安裝的延伸包。

由於每次的類載入是由子 ClassLoader 委託父 ClassLoader 先嘗試載入，但父 ClassLoader 看不到子 ClassLoader，所以同一階層的子 ClassLoader 不會被誤用，從而避兔了載入錯誤類的可能性，例如在圖 16.4 中，您想從 YourClassLoader 來載入類的話，類載入器階層不會看到 MaliciousClassLoader。

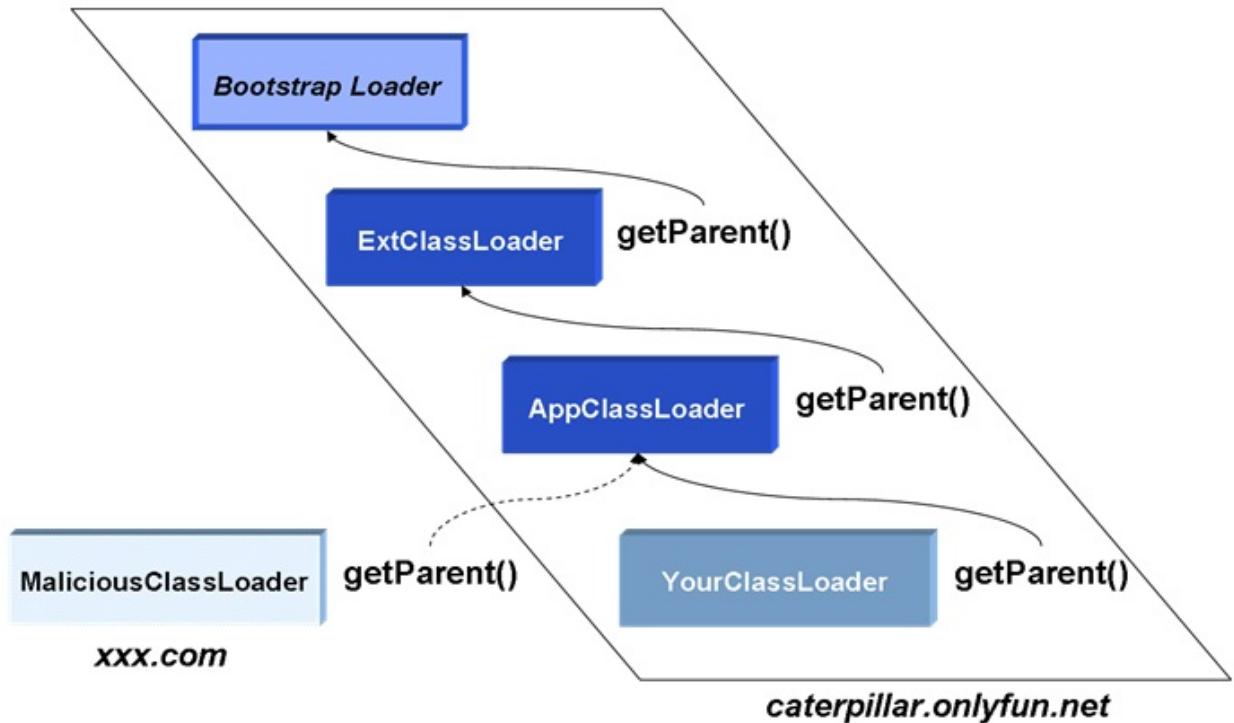


圖 16.4 类載入器階層的安全設計

由同一個 ClassLoader 載入的类檔案，會只有一份Class實例，如果同一個类檔案是由兩個不同的ClassLoader 載入，則會有兩份不同的 Class 實例。注意這個說法，如果有兩個不同的 ClassLoader 搜尋同一個类，而在 parent 的 AppClassLoader 搜尋路徑中就可以找到指定类的話，則 Class 實例就只會有一個，因為兩個不同的 ClassLoader 都是在委託父 ClassLoader 時找到該类的，如果父 ClassLoader 找不到，而是由各自的 ClassLoader 搜尋到，則 Class 的實例會有兩份。

範例 16.13 是個簡單的示範，可用來測試載入路徑與Class實例是否為同一对象。

### 範例 16.13 ClassLoaderDemo.java

```

package onlyfun.caterpillar;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class ClassLoaderDemo {
 public static void main(String[] args) {
 try {
 // 測試路徑
 String classPath = args[0];
 // 測試類
 String className = args[1];

 URL url1 = new URL(classPath);
 // 建立ClassLoader
 ClassLoader loader1 =
 new URLClassLoader(new URL[] {url1});
 // 載入指定類
 Class c1 = loader1.loadClass(className);
 // 顯示類描述
 System.out.println(c1);

 URL url2 = new URL(classPath);
 ClassLoader loader2 =
 new URLClassLoader(new URL[] {url2});
 Class c2 = loader2.loadClass(className);

 System.out.println(c2);

 System.out.println("c1 與 c1 為同一實例? "
 + (c1 == c2));
 }
 catch(ArrayIndexOutOfBoundsException e) {
 System.out.println("沒有指定類載入路徑與名稱");
 }
 catch(MalformedURLException e) {
 System.out.println("載入路徑錯誤");
 }
 catch(ClassNotFoundException e) {
 System.out.println("找不到指定的類");
 }
 }
}

```

您可以任意設計一個類，例如 TestClass，其中 classPath 可以輸入不為 ExtClassLoader 或 AppClassLoader 的搜尋路徑，例如 file:/d:/workspace/，這樣同一個類會分由兩個 ClassLoader 載入，結果會有兩份 Class 實例，則測試 c1 與 c2 是否為同一實例時，則結果會顯示 false，一個執行結果如下：

```
java onlyfun.caterpillar.ClassLoaderDemo file:/d:/workspace/ TestClass
class TestClass
class TestClass
c1 與 c1 為同一實例? false
```

如果您在執行程序時，以 -cp 將 file:/d:/workspace/ 加入為 Classpath 的一部份，由於兩個 ClassLoader 的 parent 都是 AppClassLoader，而 AppClassLoader 會在 Classpath 中找到指定的類，所以最後會只有一個指定的類之 Class 實例，則測試 c1 與 c2 是否為同一實例時，結果會顯示 true，一個執行結果如下：

```
java -cp .;d:\workspace onlyfun.caterpillar.ClassLoaderDemo file:/d:/workspace/ TestC
class TestClass
class TestClass
c1 與 c1 為同一實例? true
```

使用 -cp 指定 Classpath 時，會覆蓋原有的 Classpath 定義，也就是連現行工作目錄的路徑也覆蓋了，由於我的 ClassLoaderDemo 類是在現行工作目錄下，所以使用 -cp 時，也包括了現行工作目錄，記得組合多個 Classpath 路徑時，可以使用「;」。

## 16.2 使用反射生成與操作对象

使用反射機制，您可以於執行時期動態載入類並生成对象，操作对象上的方法、改變類成員的值，甚至連私用（private）成員的值也可以改變。

### 16.2.1 生成对象

您可以使用 Class 的 newInstance() 方法來實例化一個对象，實例化的对象是以 Object 型態傳回，例如：

```
Class c = Class.forName(className);
Object obj = c.newInstance();
```

範例 16.14 是個簡單的示範，您可以動態載入實現了 List 接口的類。

### 範例 16.14 NewInstanceDemo.java

```

package onlyfun.caterpillar;

import java.util.*;

public class NewInstanceDemo {
 public static void main(String[] args) {
 try {
 Class c = Class.forName(args[0]);
 List list = (List) c.newInstance();

 for(int i = 0; i < 5; i++) {
 list.add("element " + i);
 }

 for(Object o: list.toArray()) {
 System.out.println(o);
 }
 }
 catch(ClassNotFoundException e) {
 System.out.println("找不到指定的类");
 } catch (InstantiationException e) {
 e.printStackTrace();
 } catch (IllegalAccessException e) {
 e.printStackTrace();
 }
 }
}

```

執行結果：

```

java onlyfun.caterpillar.NewInstanceDemo java.util.ArrayList
element 0
element 1
element 2
element 3
element 4

```

實際上如果想要使用反射來動態載入類，通常是對對象的接口或類型都一無所知，也就無法像範例 16.14 中對 `newInstance()` 傳回的對象進行接口轉換動作，稍後會介紹如何以反射來調用方法以操作 `newInstance()` 所傳回的對象。

如果載入的類中具備無參數的構造方法，則可以無參數的 `newInstance()` 來建構一個不指定初始引數的對象，如果您要在動態載入及生成對象時指定對象的初始化引數，則要先指定參數型態、取得 `Constructor` 對象、使用 `Constructor` 的 `newInstance()` 並指定參數的接受值。

以一個例子來說明，首先定義一個 Student 類。

## 範例 16.15 Student.java

```
package onlyfun.caterpillar;

public class Student {
 private String name;
 private int score;

 public Student() {
 name = "N/A";
 }

 public Student(String name, int score) {
 this.name = name;
 this.score = score;
 }

 public void setName(String name) {
 this.name = name;
 }

 public void setScore(int score) {
 this.score = score;
 }

 public String getName() {
 return name;
 }

 public int getScore() {
 return score;
 }

 public String toString() {
 return name + ":" + score;
 }
}
```

您可以用 Class.forName() 來載入 Student 類，並使用第二個有參數的構造方法來建構 Student 實例，如範例 16.16 所示。

## 範例 16.16 NewInstanceDemo2.java

```
package onlyfun.caterpillar;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class NewInstanceDemo2 {
 public static void main(String[] args) {
 try {
 Class c = Class.forName(args[0]);

 // 指定參數型態
 Class[] params = new Class[2];
 // 第一個參數是String
 params[0] = String.class;
 // 第二個參數是int
 params[1] = Integer.TYPE;

 // 取得對應參數列的构造方法
 Constructor constructor =
 c.getConstructor(params);

 // 指定引數內容
 Object[] argObjs = new Object[2];
 argObjs[0] = "caterpillar";
 argObjs[1] = new Integer(90);

 // 給定引數並實例化
 Object obj = constructor.newInstance(argObjs);
 // 調用toString()來觀看描述
 System.out.println(obj);
 } catch (ClassNotFoundException e) {
 System.out.println("找不到类");
 } catch (SecurityException e) {
 e.printStackTrace();
 } catch (NoSuchMethodException e) {
 System.out.println("沒有所指定的方法");
 } catch (IllegalArgumentException e) {
 e.printStackTrace();
 } catch (InstantiationException e) {
 e.printStackTrace();
 } catch (IllegalAccessException e) {
 e.printStackTrace();
 } catch (InvocationTargetException e) {
 e.printStackTrace();
 }
 }
}
```

注意在指定基本型態時，要使用對應的包裹類（Wrapper）並使用 .TYPE，例如指定 int 型態時，則使用 Integer.TYPE，如果要指定 Integer 型態的參數的話，才是使用 Integer.class，範例 16.16 會根據指定的引數調用對應的構造方法，載入 onlyfun.caterpillar.Student 的執行結果如下：

```
java onlyfun.caterpillar.newInstanceDemo2 onlyfun.caterpillar.Student
caterpillar:90
```

## 16.2.2 調用方法

使用反射可以取回類上方法的對象代表，方法的對象代表是 java.lang.reflect.Method 的實例，您可以使用它的 invoke() 方法來動態調用指定的方法，例如調用範例 16.15 的 Student 類上之 setName() 等方法，這邊直接以範例 16.17 作為示範。

### 範例 16.17 InvokeMethodDemo.java

```
package onlyfun.caterpillar;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class InvokeMethodDemo {
 public static void main(String[] args) {
 try {
 Class c = Class.forName(args[0]);
 // 使用無參數构造方法建立对象
 Object targetObj = c.newInstance();
 // 設定參數型態
 Class[] param1 = {String.class};
 // 根據參數型態取回方法对象
 Method setNameMethod = c.getMethod("setName", param1);
 // 設定引數值
 Object[] argObjs1 = {"caterpillar"};
 // 給定引數調用指定对象之方法
 setNameMethod.invoke(targetObj, argObjs1);

 Class[] param2 = {Integer.TYPE};
 Method setScoreMethod =
 c.getMethod("setScore", param2);

 Object[] argObjs2 = {new Integer(90)};
 setScoreMethod.invoke(targetObj, argObjs2);
 // 顯示对象描述
 System.out.println(targetObj);

 } catch (ClassNotFoundException e) {
 System.out.println("找不到类");
 } catch (SecurityException e) {
 e.printStackTrace();
 } catch (NoSuchMethodException e) {
 System.out.println("沒有這個方法");
 } catch (IllegalArgumentException e) {
 e.printStackTrace();
 } catch (IllegalAccessException e) {
 e.printStackTrace();
 } catch (InvocationTargetException e) {
 e.printStackTrace();
 } catch (InstantiationException e) {
 e.printStackTrace();
 }
 }
}
```

範例 16.17 可以指定載入 Student 类並生成實例，接著可以動態调用 `setName()` 與 `setScore()` 方法，範例中參數型態與引數值的設定與範例 16.16 是類似的，由於调用 `setName()` 與 `setScore()` 所給定的引數是 "caterpillar" 與 90，所以執行的結果與範例 16.16 是相同的。

在很少的情況下，您會需要突破 Java 的存取限制來调用受保護的（protected）或私有（private）的方法（例如您拿到一個組件（Component），但您沒法修改它的原始碼來改變某個私有方法的權限，而您又一定要调用某個私有方法），這時候您可以使用反射機制來達到您的目的，一個存取私有方法的例子如下：

```
Method privateMethod =
 c.getDeclaredMethod("somePrivateMethod", new Class[0]);
privateMethod.setAccessible(true);
privateMethod.invoke(targetObj, argObjs);
```

使用反射來動態调用方法的實際例子之一是在 JavaBean 的設定，例如在 JSP/Servlet 中，可以根據使用者的請求名稱與 JavaBean 的屬性名稱自動比對，將字串請求值設定至指定的 JavaBean 上，並自動根據參數型態作型態轉換（詳細說明請見本章後網路索引）。範例 16.18 是個簡單的示範，您可以給 CommandUtil 工具类一個 Map 对象與类名稱，然後取得一個更新了值的實例，其中參數 Map 对象的「鍵」（Key）為要调用的 setter 方法名稱（不包括set開頭的名稱，例如 `setName()` 方法的話，只要給定鍵為 name 即可），而「值」（Value）為要設定給 setter 的引數。

## 範例 16.18 CommandUtil.java

```
package onlyfun.caterpillar;

import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.Map;

public class CommandUtil {
 // 給定Map对象及要產生的Bean类名稱
 // 可以取回已經設定完成的对象
 public static Object getCommand(Map requestMap,
 String commandClass)
 throws Exception {
 Class c = Class.forName(commandClass);
 Object o = c.newInstance();

 return updateCommand(requestMap, o);
 }

 // 使用reflection自動找出要更新的屬性
}
```

```

public static Object updateCommand(
 Map requestMap,
 Object command)
 throws Exception {
 Method[] methods =
 command.getClass().getDeclaredMethods();

 for(int i = 0; i < methods.length; i++) {
 // 略過private、protected成員
 // 且找出必須是set開頭的方法名稱
 if(!Modifier.isPrivate(methods[i].getModifiers()) &&
 !Modifier.isProtected(methods[i].getModifiers()) &&
 methods[i].getName().startsWith("set")) {
 // 取得不包括set的名稱
 String name = methods[i].getName()
 .substring(3)
 .toLowerCase();
 // 如果setter名稱與鍵值相同
 // 調用對應的setter並設定值
 if(requestMap.containsKey(name)) {
 String param = (String) requestMap.get(name);
 Object[] values = findOutParamValues(
 param, methods[i]);
 methods[i].invoke(command, values);
 }
 }
 }
 return command;
}

// 轉換為對應型態的值
private static Object[] findOutParamValues(
 String param, Method method) {
 Class[] params = method.getParameterTypes();
 Object[] objs = new Object[params.length];

 for(int i = 0; i < params.length; i++) {
 if(params[i] == String.class) {
 objs[i] = param;
 }
 else if(params[i] == Short.TYPE) {
 short number = Short.parseShort(param);
 objs[i] = new Short(number);
 }
 else if(params[i] == Integer.TYPE) {
 int number = Integer.parseInt(param);
 objs[i] = new Integer(number);
 }
 else if(params[i] == Long.TYPE) {
 long number = Long.parseLong(param);
 objs[i] = new Long(number);
 }
 }
}

```

```

 else if(params[i] == Float.TYPE) {
 float number = Float.parseFloat(param);
 objs[i] = new Float(number);
 }
 else if(params[i] == Double.TYPE) {
 double number = Double.parseDouble(param);
 objs[i] = new Double(number);
 }
 else if(params[i] == Boolean.TYPE) {
 boolean bool = Boolean.parseBoolean(param);
 objs[i] = new Boolean(bool);
 }
 }
 return objs;
}
}

```

CommandUtil 可以自動根據方法上的參數型態，將 Map 對象中的「值」對象轉換為屬性上的對應型態，目前它可以轉換基本型態與 String 型態的屬性，一個使用 CommandUtil 類的例子如範例 16.19 所示。

## 範例 16.19 CommandUtilDemo.java

```

package onlyfun.caterpillar;

import java.util.*;

public class CommandUtilDemo {
 public static void main(String[] args) throws Exception {
 Map<String, String> request =
 new HashMap<String, String>();
 request.put("name", "caterpillar");
 request.put("score", "90");
 Object obj = CommandUtil.getCommand(request, args[0]);
 System.out.println(obj);
 }
}

```

您可以使用範例 16.19 來載入 Student 類，使用 CommandUtil.getCommand() 方法可以返回一個設定好值的 Student 實例，雖然您設定給 request 的「值」是字串型態，但 CommandUtil 會使用反射機制來自動轉換為屬性上的對應型態，一個執行的範例如下所示：

```

java onlyfun.caterpillar.CommandUtilDemo onlyfun.caterpillar.Student
caterpillar:90

```

良葛格的話匣子 不知道方法的名稱如何調用？其實範例 16.17 就給出了答案，透過規範方法的命名方式，之後就可以透過反射機制加上方法名稱的比對，以正確調用對應的方法。

### 16.2.3 修改成員值

儘管直接存取類的資料成員（Field）是不被鼓勵的，但您仍是可以直接存取公開的（public）資料成員，而您甚至也可以透過反射機制來存取私用資料成員，以一個實例來說明，首先撰寫個 TestField 類。

### 範例 16.20 TestField.java

```
package onlyfun.caterpillar;

public class TestField {
 public int testInt;
 public String testString;

 public String toString() {
 return testInt + ":" + testString;
 }
}
```

範例 16.21 則利用反射機制動態載入類來存取資料成員。

### 範例 16.21 AssignFieldDemo.java

```

package onlyfun.caterpillar;

import java.lang.reflect.Field;

public class AssignFieldDemo {
 public static void main(String[] args) {
 try {
 Class c = Class.forName(args[0]);
 Object targetObj = c.newInstance();

 Field testInt = c.getField("testInt");
 testInt.setInt(targetObj, 99);

 Field testString = c.getField("testString");
 testString.set(targetObj, "caterpillar");

 System.out.println(targetObj);
 } catch(ArrayIndexOutOfBoundsException e) {
 System.out.println("沒有指定類");
 } catch (ClassNotFoundException e) {
 System.out.println("找不到指定的類");
 } catch (SecurityException e) {
 e.printStackTrace();
 } catch (NoSuchFieldException e) {
 System.out.println("找不到指定的資料成員");
 } catch (InstantiationException e) {
 e.printStackTrace();
 } catch (IllegalAccessException e) {
 e.printStackTrace();
 }
 }
}

```

您可以載入 TestField 类來看看執行的結果，如下所示：

```

java onlyfun.caterpillar.AssignFieldDemo onlyfun.caterpillar.TestField
99:caterpillar

```

如果有必要的話，您也可以透過反射機制來存取私有的資料成員，例如：

```

Field privateField = c.getDeclaredField("privateField");
privateField.setAccessible(true);
privateField.setInt(targetObj, 99);

```

## 16.2.4 再看数组对象

在 Java 中数组也是一个对象，也會有一個 Class 實例來表示它，範例 16.22 顯示幾個基本型態以及 String 数组的类名稱描述。

## 範例 16.22 ArrayDemo.java

```
package onlyfun.caterpillar;

public class ArrayDemo {
 public static void main(String[] args) {
 short[] sArr = new short[5];
 int[] iArr = new int[5];
 long[] lArr = new long[5];
 float[] fArr = new float[5];
 double[] dArr = new double[5];
 byte[] bArr = new byte[5];
 boolean[] zArr = new boolean[5];
 String[] strArr = new String[5];

 System.out.println("short 数组类: " + sArr.getClass());
 System.out.println("int 数组类: " + iArr.getClass());
 System.out.println("long 数组类: " + lArr.getClass());
 System.out.println("float 数组类: " + fArr.getClass());
 System.out.println("double 数组类: " + dArr.getClass());
 System.out.println("byte 数组类: " + bArr.getClass());
 System.out.println("boolean 数组类: " + zArr.getClass());
 System.out.println("String 数组类: " + strArr.getClass());
 }
}
```

使用 `toString()` 來顯示数组对象的类名稱描述時，會以 "class [" 作為開始，之後跟隨著一個型態表示字元，執行結果如下所示：

```
short 数组类: class [S
int 数组类: class [I
long 数组类: class [J
float 数组类: class [F
double 数组类: class [D
byte 数组类: class [B
boolean 数组类: class [Z
String 数组类: class [Ljava.lang.String;
```

要使用反射機制動態生成数组的話，可以使用 `java.lang.reflect.Array` 來協助，範例 16.23 簡單的示範了如何生成 String 数组。

## 範例 16.23 NewArrayDemo.java

```
package onlyfun.caterpillar;

import java.lang.reflect.Array;

public class NewArrayDemo {
 public static void main(String[] args) {
 Class c = String.class;
 Object objArr = Array.newInstance(c, 5);

 for(int i = 0; i < 5; i++) {
 Array.set(objArr, i, i+"");
 }

 for(int i = 0; i < 5; i++) {
 System.out.print(Array.get(objArr, i) + " ");
 }
 System.out.println();

 String[] strs = (String[]) objArr;
 for(String s : strs) {
 System.out.print(s + " ");
 }
 }
}
```

Array.newInstance() 的第一個參數是指定元素型態，而第二個參數是用來指定数组長度，執行結果如下：

```
0 1 2 3 4
0 1 2 3 4
```

Array.newInstance() 還有另一個版本，可用於建立二維数组，只要用一個表示二維数组的兩個維度長度的 int 数组，傳遞給第二個參數，範例 16.24 是個簡單示範。

## 範例 16.24 NewArrayDemo2.java

```

package onlyfun.caterpillar;

import java.lang.reflect.Array;

public class NewArrayDemo2 {
 public static void main(String[] args) {
 Class c = String.class;

 // 打算建立一個3*4数组
 int[] dim = new int[]{3, 4};
 Object objArr = Array.newInstance(c, dim);

 for(int i = 0; i < 3; i++) {
 Object row = Array.get(objArr, i);
 for(int j = 0; j < 4; j++) {
 Array.set(row, j, "" + (i+1)*(j+1));
 }
 }

 for(int i = 0; i < 3; i++) {
 Object row = Array.get(objArr, i);
 for(int j = 0; j < 4; j++) {
 System.out.print(Array.get(row, j) + " ");
 }
 System.out.println();
 }
 }
}

```

執行結果如下：

```

1 2 3 4
2 4 6 8
3 6 9 12

```

如果想要得知数组元素的型態，可以在取得数组的 Class 實例之後，使用 Class 實例的 `getComponentType()` 方法，所取回的是元素的 Class 實例，例如：

```

int[] iArr = new int[5];
System.out.println(iArr.getClass().getComponentType());

```

## 16.2.5 Proxy 类

Java 在 J2SE 1.3 之後加入 `java.lang.reflect.Proxy` 类，可協助您實現動態代理功能，舉個實際應用的例子，假設今天您打算開發一個 `HelloSpeaker` 类，當中有一個 `hello()` 方法，而您想要在這個 `hello()` 調用前後加上記錄（log）的功能，但又不想將記錄的功能寫到 `HelloSpeaker` 类中，這時您可以使用 `Proxy` 类來實現動態代理。要實現動態代理，首先要定義所要代理的接口，範例 16.25 為定義了有 `hello()` 方法的 `IHello` 接口。

## 範例 16.25 IHello.java

```
package onlyfun.caterpillar;

public interface IHello {
 public void hello(String name);
}
```

您的 `HelloSpeaker` 类實現了 `IHello` 接口，如範例 16.26 所示。

## 範例 16.26 HelloSpeaker.java

```
package onlyfun.caterpillar;

public class HelloSpeaker implements IHello {
 public void hello(String name) {
 System.out.println("Hello, " + name);
 }
}
```

您可以實作一個處理記錄（log）的處理者（Handler），讓處理者在調用 `hello()` 方法的前後進行記錄的動作，一個處理者必須實現 `java.lang.reflect.InvocationHandler` 接口，`InvocationHandler` 有一個 `invoke()` 方法必須實現，範例 16.27 是個簡單的實現。

## 範例 16.27 LogHandler.java

```

package onlyfun.caterpillar;

import java.util.logging.*;
import java.lang.reflect.*;

public class LogHandler implements InvocationHandler {
 private Logger logger =
 Logger.getLogger(this.getClass().getName());
 private Object delegate;

 // 綁定要代理的对象
 public Object bind(Object delegate) {
 this.delegate = delegate;
 // 建立並傳回代理对象
 return Proxy.newProxyInstance(
 delegate.getClass().getClassLoader(),
 // 要被代理的接口
 delegate.getClass().getInterfaces(),
 this);
 }

 // 代理要调用的方法，並在其前後增加行為
 public Object invoke(Object proxy,
 Method method,
 Object[] args) throws Throwable {
 Object result = null;
 try {
 logger.log(Level.INFO,
 "method starts..." + method.getName());
 result = method.invoke(delegate, args);
 logger.log(Level.INFO,
 "method ends..." + method.getName());
 } catch (Exception e){
 logger.log(Level.INFO, e.toString());
 }
 return result;
 }
}

```

主要的概念是使用 `Proxy.newProxyInstance()` 方法建立一個代理对象，建立代理对象時必須告知所要代理的操作接口，之後您可以操作所建立的代理对象，在每次操作時會調用 `InvocationHandler` 的 `invoke()` 方法，`invoke()` 方法會傳入被代理对象的方法名稱與執行引數，實際上要執行的方法交由 `method.invoke()`，您在 `method.invoke()` 前後加上記錄動作，`method.invoke()` 傳回的对象是實際方法執行過後的回傳結果，先從範例 16.28 來看看一個執行的例子。

## 範例 16.28 ProxyDemo.java

```

package onlyfun.caterpillar;

public class ProxyDemo {
 public static void main(String[] args) {
 LogHandler handler = new LogHandler();
 IHello speaker = new HelloSpeaker();

 // 代理speaker的对象
 IHello speakerProxy =
 (IHello) handler.bind(speaker);

 speakerProxy.hello("Justin");
 }
}

```

執行結果如下：

```

2005/6/4 上午 09:39:11 onlyfun.caterpillar.LogHandler invoke
信息: method starts...hello
Hello, Justin
2005/6/4 上午 09:39:11 onlyfun.caterpillar.LogHandler invoke
信息: method ends...hello

```

透過代理機制，在不將記錄動作寫入為 HelloSpeaker 類程序碼的情況下，您可以為其加入記錄的功能，這並不是什麼魔法，只不過是在 hello() 方法前後由代理對象 speakerProxy 先執行記錄功能而已，而真正執行 hello() 方法時才使用 speaker 對象。

**良葛格的話匣子** 這邊的例子是「Proxy 模式」的實現，您可以進一步參考：

- [Proxy 模式（一）](#)
- [Proxy 模式（二）](#)

## 16.3 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 瞭解 Class 實例與類的關係
- 會使用 Class.forName() 載入類並獲得類信息
- 會使用 Class 建立實例
- 會使用反射機制調用對象上的方法

下一個章節要來介紹 J2SE 5.0 中新增的 Annotation 功能，Annotation 是 J2SE 5.0 對 metadata 的支援，metadata 簡單的說就是「資料的資料」（Data about data），您可以使用 Annotation 對程序碼作出一些說明，以利一些程序碼分析工具來使用這些信息。

# 深入探讨 Java 类加载器

来源:[www.ibm.com](http://www.ibm.com)

类加载器 (class loader) 是 Java™中的一个很重要的概念。类加载器负责加载 Java 类的字节代码到 Java 虚拟机中。本文首先详细介绍了 Java 类加载器的基本概念，包括代理模式、加载类的具体过程和线程上下文类加载器等，接着介绍如何开发自己的类加载器，最后介绍了类加载器在 Web 容器和 OSGi™中的应用。

类加载器是 Java 语言的一个创新，也是 Java 语言流行的重要原因之一。它使得 Java 类可以被动态加载到 Java 虚拟机中并执行。类加载器从 JDK 1.0 就出现了，最初是为了满足 Java Applet 的需要而开发出来的。Java Applet 需要从远程下载 Java 类文件到浏览器中并执行。现在类加载器在 Web 容器和 OSGi 中得到了广泛的使用。一般来说，Java 应用的开发人员不需要直接同类加载器进行交互。Java 虚拟机默认的行为就已经足够满足大多数情况的需求了。不过如果遇到了需要与类加载器进行交互的情况，而对类加载器的机制又不是很了解的话，就很容易花大量的时间去调试 ClassNotFoundException 和 NoClassDefFoundError 等异常。本文将详细介绍 Java 的类加载器，帮助读者深刻理解 Java 语言中的这个重要概念。下面首先介绍一些相关的基本概念。

## 类加载器基本概念

顾名思义，类加载器 (class loader) 用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式如下：Java 源程序 (.java 文件) 在经过 Java 编译器编译之后就被转换成 Java 字节代码 (.class 文件)。类加载器负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例。每个这样的实例用来表示一个 Java 类。通过此实例的 newInstance() 方法就可以创建出该类的一个对象。实际的情况可能更加复杂，比如 Java 字节代码可能是通过工具动态生成的，也可能是通过网络下载的。

基本上所有的类加载器都是 java.lang.ClassLoader 类的一个实例。下面详细介绍这个 Java 类。

## java.lang.ClassLoader类介绍

java.lang.ClassLoader 类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个 Java 类，即 java.lang.Class 类的一个实例。除此之外，ClassLoader 还负责加载 Java 应用所需的资源，如图像文件和配置文件。

等。不过本文只讨论其加载类的功能。为了完成加载类的这个职责，ClassLoader提供了一系列的方法，比较重要的方法如表 1 所示。关于这些方法的细节会在下面进行介绍。

**表 1. ClassLoader 中与加载类相关的方法**

| 方法                                                   | 说明                                                                   |
|------------------------------------------------------|----------------------------------------------------------------------|
| getParent()                                          | 返回该类加载器的父类加载器。                                                       |
| loadClass(String name)                               | 加载名称为 name 的类，返回的结果是 java.lang.Class 类的实例。                           |
| findClass(String name)                               | 查找名称为 name 的类，返回的结果是 java.lang.Class 类的实例。                           |
| findLoadedClass(String name)                         | 查找名称为 name 的已经被加载过的类，返回的结果是 java.lang.Class 类的实例。                    |
| defineClass(String name, byte[] b, int off, int len) | 把字节数组 b 中的内容转换成 Java 类，返回的结果是 java.lang.Class 类的实例。这个方法被声明为 final 的。 |
| resolveClass(Class<?> c)                             | 链接指定的 Java 类。                                                        |

对于表 1 中给出的方法，表示类名称的 name 参数的值是类的二进制名称。需要注意的是内部类的表示，如 com.example.Sample\$1 和 com.example.Sample\$Inner 等表示方式。这些方法会在下面介绍类加载器的工作机制时，做进一步的说明。下面介绍类加载器的树状组织结构。

## 类加载器的树状组织结构

Java 中的类加载器大致可以分成两类，一类是系统提供的，另外一类则是由 Java 应用开发人员编写的。系统提供的类加载器主要有下面三个：

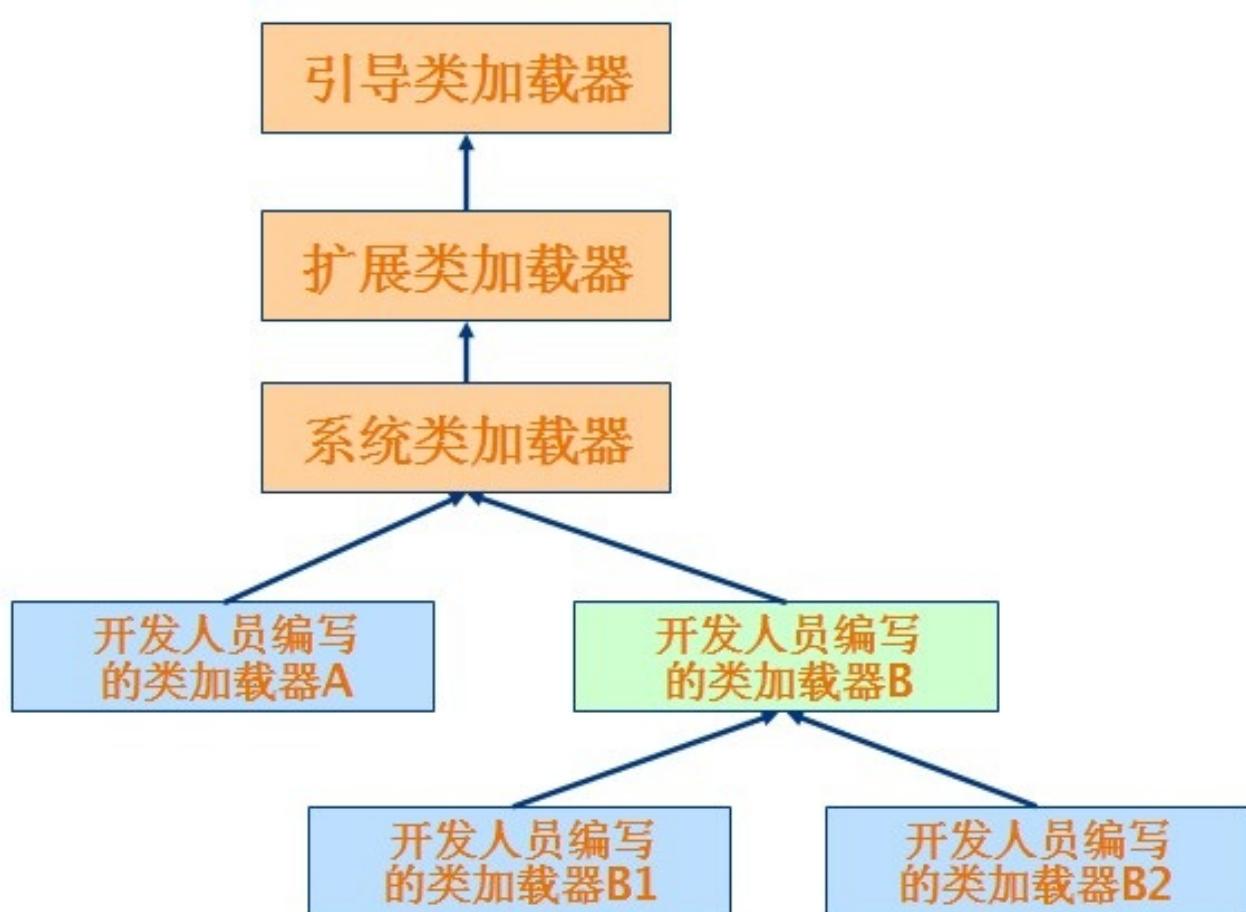
- 引导类加载器（bootstrap class loader）：它用来加载 Java 的核心库，是用原生代码来实现的，并不继承自 java.lang.ClassLoader。
- 扩展类加载器（extensions class loader）：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
- 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader() 来获取它。

除了系统提供的类加载器以外，开发人员可以通过继承 java.lang.ClassLoader 类的方式实现自己的类加载器，以满足一些特殊的需求。

除了引导类加载器之外，所有的类加载器都有一个父类加载器。通过表 1 中给出的 getParent() 方法可以得到。对于系统提供的类加载器来说，系统类加载器的父类加载器是扩展类加载器，而扩展类加载器的父类加载器是引导类加载器；对于开发人员编写的类加载器来说，其父类加载器是加载此类加载器 Java 类的类加载器。因为类加载器 Java 类如同其它的 Java 类一样，也是要由类加载器来加载的。一般来说，开发人员编写的类加

载器的父类加载器是系统类加载器。类加载器通过这种方式组织起来，形成树状结构。树的根节点就是引导类加载器。图 1 中给出了一个典型的类加载器树状组织结构示意图，其中的箭头指向的是父类加载器。

图 1. 类加载器树状组织结构示意图



代码清单 1 演示了类加载器的树状组织结构。

清单 1. 演示类加载器的树状组织结构

```
public class ClassLoaderTree {
 public static void main(String[] args) {
 ClassLoader loader = ClassLoaderTree.class.getClassLoader();
 while (loader != null) {
 System.out.println(loader.toString());
 loader = loader.getParent();
 }
 }
}
```

每个 Java 类都维护着一个指向定义它的类加载器的引用，通过 `getClassLoader()`方法就可以获取到此引用。代码清单 1 中通过递归调用 `getParent()`方法来输出全部的父类加载器。代码清单 1 的运行结果如 代码清单 2 所示。

### 清单 2. 演示类加载器的树状组织结构的运行结果

```
sun.misc.Launcher$AppClassLoader@9304b1
sun.misc.Launcher$ExtClassLoader@190d11
```

如代码清单 2 所示，第一个输出的是 `ClassLoaderTree` 类的类加载器，即系统类加载器。它是 `sun.misc.Launcher$AppClassLoader` 类的实例；第二个输出的是扩展类加载器，是 `sun.misc.Launcher$ExtClassLoader` 类的实例。需要注意的是这里并没有输出引导类加载器，这是由于有些 JDK 的实现对于父类加载器是引导类加载器的情况，`getParent()`方法返回 `null`。

在了解了类加载器的树状组织结构之后，下面介绍类加载器的代理模式。

## 类加载器的代理模式

类加载器在尝试自己去查找某个类的字节代码并定义它时，会先代理给其父类加载器，由父类加载器先去尝试加载这个类，依次类推。在介绍代理模式背后的动机之前，首先需要说明一下 Java 虚拟机是如何判定两个 Java 类是相同的。Java 虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。比如一个 Java 类 `com.example.Sample`，编译之后生成了字节代码文件 `Sample.class`。两个不同的类加载器 `ClassLoaderA` 和 `ClassLoaderB` 分别读取了这个 `Sample.class` 文件，并定义出两个 `java.lang.Class` 类的实例来表示这个类。这两个实例是不相同的。对于 Java 虚拟机来说，它们是不同的类。试图对这两个类的对象进行相互赋值，会抛出运行时异常 `ClassCastException`。下面通过示例来具体说明。代码清单 3 中给出了 Java 类 `com.example.Sample`。

### 清单 3. `com.example.Sample` 类

```
package com.example;

public class Sample {
 private Sample instance;

 public void setSample(Object instance) {
 this.instance = (Sample) instance;
 }
}
```

如代码清单 3所示， com.example.Sample类的方法 setSample接受一个 java.lang.Object类型的参数，并且会把该参数强制转换成 com.example.Sample类型。测试 Java 类是否相同的代码如代码清单 4所示。

#### 清单 4. 测试 Java 类是否相同

```
public void testClassIdentity() {
 String classDataRootPath = "C:\\workspace\\Classloader\\classData";
 FileSystemClassLoader fscl1 = new FileSystemClassLoader(classDataRootPath);
 FileSystemClassLoader fscl2 = new FileSystemClassLoader(classDataRootPath);
 String className = "com.example.Sample";
 try {
 Class<?> class1 = fscl1.loadClass(className);
 Object obj1 = class1.newInstance();
 Class<?> class2 = fscl2.loadClass(className);
 Object obj2 = class2.newInstance();
 Method setSampleMethod = class1.getMethod("setSample", java.lang.Object.class);
 setSampleMethod.invoke(obj1, obj2);
 } catch (Exception e) {
 e.printStackTrace();
 }
}
```

代码清单 4中使用了类 FileSystemClassLoader的两个不同实例来分别加载类 com.example.Sample，得到了两个不同的 java.lang.Class的实例，接着通过 newInstance()方法分别生成了两个类的对象 obj1和 obj2，最后通过 Java 的反射 API 在对象 obj1上调用方法 setSample，试图把对象 obj2赋值给 obj1内部的 instance对象。代码清单 4的运行结果如 代码清单 5所示。

#### 清单 5. 测试 Java 类是否相同的运行结果

```
java.lang.reflect.InvocationTargetException
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:597)
at classloader.ClassIdentity.testClassIdentity(ClassIdentity.java:26)
at classloader.ClassIdentity.main(ClassIdentity.java:9)
Caused by: java.lang.ClassCastException: com.example.Sample
cannot be cast to com.example.Sample
at com.example.Sample.setSample(Sample.java:7)
... 6 more
```

从代码清单 5给出的运行结果可以看到，运行时抛出了 `java.lang.ClassCastException` 异常。虽然两个对象 `obj1` 和 `obj2` 的类的名字相同，但是这两个类是由不同的类加载器实例来加载的，因此不被 Java 虚拟机认为是相同的。

了解了这一点之后，就可以理解代理模式的设计动机了。代理模式是为了保证 Java 核心库的类型安全。所有 Java 应用都至少需要引用 `java.lang.Object` 类，也就是说在运行的时候，`java.lang.Object` 这个类需要被加载到 Java 虚拟机中。如果这个加载过程由 Java 应用自己的类加载器来完成的话，很可能就存在多个版本的 `java.lang.Object` 类，而且这些类之间是不兼容的。通过代理模式，对于 Java 核心库的类的加载工作由引导类加载器来统一完成，保证了 Java 应用所使用的都是同一个版本的 Java 核心库的类，是互相兼容的。

不同的类加载器为相同名称的类创建了额外的名称空间。相同名称的类可以并存在 Java 虚拟机中，只需要用不同的类加载器来加载它们即可。不同类加载器加载的类之间是不兼容的，这就相当于在 Java 虚拟机内部创建了一个个相互隔离的 Java 类空间。这种技术在许多框架中都被用到，后面会详细介绍。

下面具体介绍类加载器加载类的详细过程。

## 加载类的过程

在前面介绍类加载器的代理模式的时候，提到过类加载器会首先代理给其它类加载器来尝试加载某个类。这就意味着真正完成类的加载工作的类加载器和启动这个加载过程的类加载器，有可能不是同一个。真正完成类的加载工作是通过调用 `defineClass` 来实现的；而启动类的加载过程是通过调用 `loadClass` 来实现的。前者称为一个类的定义加载器

(defining loader)，后者称为初始加载器 (initiating loader)。在 Java 虚拟机判断两个类是否相同的时候，使用的是类的定义加载器。也就是说，哪个类加载器启动类的加载过程并不重要，重要的是最终定义这个类的加载器。两种类加载器的关联之处在于：一个类的定义加载器是它引用的其它类的初始加载器。如类 `com.example.Outer` 引用了类 `com.example.Inner`，则由类 `com.example.Outer` 的定义加载器负责启动类 `com.example.Inner` 的加载过程。

方法 `loadClass()` 抛出的是 `java.lang.ClassNotFoundException` 异常；方法 `defineClass()` 抛出的是 `java.lang.NoClassDefFoundError` 异常。

类加载器在成功加载某个类之后，会把得到的 `java.lang.Class` 类的实例缓存起来。下次再请求加载该类的时候，类加载器会直接使用缓存的类的实例，而不会尝试再次加载。也就是说，对于一个类加载器实例来说，相同全名的类只加载一次，即 `loadClass` 方法不会被重复调用。

下面讨论另外一种类加载器：线程上下文类加载器。

## 线程上下文类加载器

线程上下文类加载器 (context class loader) 是从 JDK 1.2 开始引入的。类 `java.lang.Thread` 中的方法 `getContextClassLoader()` 和 `setContextClassLoader(ClassLoader cl)` 用来获取和设置线程的上下文类加载器。如果没有通过 `setContextClassLoader(ClassLoader cl)` 方法进行设置的话，线程将继承其父线程的上下文类加载器。Java 应用运行的初始线程的上下文类加载器是系统类加载器。在线程中运行的代码可以通过此类加载器来加载类和资源。

前面提到的类加载器的代理模式并不能解决 Java 应用开发中会遇到的类加载器的全部问题。Java 提供了很多服务提供者接口 (Service Provider Interface, SPI)，允许第三方为这些接口提供实现。常见的 SPI 有 JDBC、JCE、JNDI、JAXP 和 JBI 等。这些 SPI 的接口由 Java 核心库来提供，如 JAXP 的 SPI 接口定义包含在 `javax.xml.parsers` 包中。这些 SPI 的实现代码很可能是作为 Java 应用所依赖的 jar 包被包含进来，可以通过类路径 (CLASSPATH) 来找到，如实现了 JAXP SPI 的 Apache Xerces 所包含的 jar 包。SPI 接口中的代码经常需要加载具体的实现类。如 JAXP 中的 `javax.xml.parsers.DocumentBuilderFactory` 类中的 `newInstance()` 方法用来生成一个新的 `DocumentBuilderFactory` 的实例。这里的实例的真正的类是继承自 `javax.xml.parsers.DocumentBuilderFactory`，由 SPI 的实现所提供的。如在 [Apache Xerces](#) 中，实现的类是 `org.apache.xerces.jaxp.DocumentBuilderFactoryImpl`。而问题在于，SPI 的接口是 Java 核心库的一部分，是由引导类加载器来加载的；SPI 实现的 Java 类一般是由系统类加载器来加载的。引导类加载器是无法找到 SPI 的实现类的，因为它只加载 Java 的核心库。它也不能代理给系统类加载器，因为它是系统类加载器的祖先类加载器。也就是说，类加载器的代理模式无法解决这个问题。

线程上下文类加载器正好解决了这个问题。如果不做任何的设置，Java 应用的线程的上下文类加载器默认就是系统上下文类加载器。在 SPI 接口的代码中使用线程上下文类加载器，就可以成功的加载到 SPI 实现的类。线程上下文类加载器在很多 SPI 的实现中都会用到。

下面介绍另外一种加载类的方法：`Class.forName`。

## Class.forName

`Class.forName` 是一个静态方法，同样可以用来加载类。该方法有两种形式：

`Class.forName(String name, boolean initialize, ClassLoader loader)` 和 `Class.forName(String className)`。

第一种形式的参数 name 表示的是类的全名； initialize 表示是否初始化类； loader 表示加载时使用的类加载器。第二种形式则相当于设置了参数 initialize 的值为 true， loader 的值为当前类的类加载器。 Class.forName 的一个很常见的用法是在加载数据库驱动的时候。如 Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance() 用来加载 Apache Derby 数据库的驱动。

在介绍完类加载器相关的基本概念之后，下面介绍如何开发自己的类加载器。

## 开发自己的类加载器

虽然在绝大多数情况下，系统默认提供的类加载器实现已经可以满足需求。但是在某些情况下，您还是需要为应用开发出自己的类加载器。比如您的应用通过网络来传输 Java 类的字节代码，为了保证安全性，这些字节代码经过了加密处理。这个时候您就需要自己的类加载器来从某个网络地址上读取加密后的字节代码，接着进行解密和验证，最后定义出要在 Java 虚拟机中运行的类来。下面将通过两个具体的实例来说明类加载器的开发。

### 文件系统类加载器

第一个类加载器用来加载存储在文件系统上的 Java 字节代码。完整的实现如代码清单 6 所示。

#### 清单 6. 文件系统类加载器

```

public class FileSystemClassLoader extends ClassLoader {

 private String rootDir;

 public FileSystemClassLoader(String rootDir) {
 this.rootDir = rootDir;
 }

 protected Class<?> findClass(String name) throws ClassNotFoundException {
 byte[] classData = getClassData(name);
 if (classData == null) {
 throw new ClassNotFoundException();
 }
 else {
 return defineClass(name, classData, 0, classData.length);
 }
 }

 private byte[] getClassData(String className) {
 String path = classNameToPath(className);
 try {
 InputStream ins = new FileInputStream(path);
 ByteArrayOutputStream baos = new ByteArrayOutputStream();
 int bufferSize = 4096;
 byte[] buffer = new byte[bufferSize];
 int bytesNumRead = 0;
 while ((bytesNumRead = ins.read(buffer)) != -1) {
 baos.write(buffer, 0, bytesNumRead);
 }
 return baos.toByteArray();
 } catch (IOException e) {
 e.printStackTrace();
 }
 return null;
 }

 private String classNameToPath(String className) {
 return rootDir + File.separatorChar
 + className.replace('.', File.separatorChar) + ".class";
 }
}

```

如代码清单 6 所示，类 FileSystemClassLoader 继承自类 java.lang.ClassLoader。在表 1 中列出的 java.lang.ClassLoader 类的常用方法中，一般来说，自己开发的类加载器只需要覆写 findClass(String name) 方法即可。java.lang.ClassLoader 类的方法 loadClass() 封装了前面提到的代理模式的实现。该方法会首先调用 findLoadedClass() 方法来检查该类是否已经被加载过；如果没有加载过的话，会调用父类加载器的 loadClass() 方法来尝试加

载该类；如果父类加载器无法加载该类的话，就调用 `findClass()` 方法来查找该类。因此，为了保证类加载器都正确实现代理模式，在开发自己的类加载器时，最好不要覆写 `loadClass()` 方法，而是覆写 `findClass()` 方法。

类 `FileSystemClassLoader` 的 `findClass()` 方法首先根据类的全名在硬盘上查找类的字节代码文件（.class 文件），然后读取该文件内容，最后通过 `defineClass()` 方法来把这些字节代码转换成 `java.lang.Class` 类的实例。

## 网络类加载器

下面将通过一个网络类加载器来说明如何通过类加载器来实现组件的动态更新。即基本的场景是：Java 字节代码（.class）文件存放在服务器上，客户端通过网络的方式获取字节代码并执行。当有版本更新的时候，只需要替换掉服务器上保存的文件即可。通过类加载器可以比较简单的实现这种需求。

类 `NetworkClassLoader` 负责通过网络下载 Java 类字节代码并定义出 Java 类。它的实现与 `FileSystemClassLoader` 类似。在通过 `NetworkClassLoader` 加载了某个版本的类之后，一般有两种做法来使用它。第一种做法是使用 Java 反射 API。另外一种做法是使用接口。需要注意的是，并不能直接在客户端代码中引用从服务器上下载的类，因为客户端代码的类加载器找不到这些类。使用 Java 反射 API 可以直接调用 Java 类的方法。而使用接口的做法则是把接口的类放在客户端中，从服务器上加载实现此接口的不同版本的类。在客户端通过相同的接口来使用这些实现类。网络类加载器的具体代码见[下载](#)。

在介绍完如何开发自己的类加载器之后，下面说明类加载器和 Web 容器的关系。

## 类加载器与 Web 容器

对于运行在 Java EE™ 容器中的 Web 应用来说，类加载器的实现方式与一般的 Java 应用有所不同。不同的 Web 容器的实现方式也会有所不同。以 Apache Tomcat 来说，每个 Web 应用都有一个对应的类加载器实例。该类加载器也使用代理模式，所不同的是它是首先尝试去加载某个类，如果找不到再代理给父类加载器。这与一般类加载器的顺序是相反的。这是 Java Servlet 规范中的推荐做法，其目的是使得 Web 应用自己的类的优先级高于 Web 容器提供的类。这种代理模式的一个例外是：Java 核心库的类是不在查找范围之内的。这也是为了保证 Java 核心库的类型安全。

绝大多数情况下，Web 应用的开发人员不需要考虑与类加载器相关的细节。下面给出几条简单的原则：

- 每个 Web 应用自己的 Java 类文件和使用的库的 jar 包，分别放在 WEB-INF/classes 和 WEB-INF/lib 目录下面。

- 多个应用共享的 Java 类文件和 jar 包，分别放在 Web 容器指定的由所有 Web 应用共享的目录下面。
- 当出现找不到类的错误时，检查当前类的类加载器和当前线程的上下文类加载器是否正确。

在介绍完类加载器与 Web 容器的关系之后，下面介绍它与 OSGi 的关系。

## 类加载器与 OSGi

OSGi™是 Java 上的动态模块系统。它为开发人员提供了面向服务和基于组件的运行环境，并提供标准的方式来管理软件的生命周期。OSGi 已经被实现和部署在很多产品上，在开源社区也得到了广泛的支持。Eclipse 就是基于 OSGi 技术来构建的。OSGi 中的每个模块（bundle）都包含 Java 包和类。模块可以声明它所依赖的需要导入（import）的其它模块的 Java 包和类（通过 Import-Package），也可以声明导出（export）自己的包和类，供其它模块使用（通过 Export-Package）。也就是说需要能够隐藏和共享一个模块中的某些 Java 包和类。这是通过 OSGi 特有的类加载器机制来实现的。OSGi 中的每个模块都有对应的一个类加载器。它负责加载模块自己包含的 Java 包和类。当它需要加载 Java 核心库的类时（以 java 开头的包和类），它会代理给父类加载器（通常是启动类加载器）来完成。当它需要加载所导入的 Java 类时，它会代理给导出此 Java 类的模块来完成加载。模块也可以显式的声明某些 Java 包和类，必须由父类加载器来加载。只需要设置系统属性 org.osgi.framework.bootdelegation 的值即可。

假设有两个模块 bundleA 和 bundleB，它们都有自己对应的类加载器 classLoaderA 和 classLoaderB。在 bundleA 中包含类 com.bundleA.Sample，并且该类被声明为导出的，也就是说可以被其它模块所使用的。bundleB 声明了导入 bundleA 提供的类 com.bundleA.Sample，并包含一个类 com.bundleB.NewSample 继承自 com.bundleA.Sample。在 bundleB 启动的时候，其类加载器 classLoaderB 需要加载类 com.bundleB.NewSample，进而需要加载类 com.bundleA.Sample。由于 bundleB 声明了类 com.bundleA.Sample 是导入的，classLoaderB 把加载类 com.bundleA.Sample 的工作代理给导出该类的 bundleA 的类加载器 classLoaderA。classLoaderA 在其模块内部查找类 com.bundleA.Sample 并定义它，所得到的类 com.bundleA.Sample 实例就可以被所有声明导入了此类的模块使用。对于以 java 开头的类，都是由父类加载器来加载的。如果声明了系统属性 org.osgi.framework.bootdelegation=com.example.core.\*，那么对于包 com.example.core 中的类，都是由父类加载器来完成的。

OSGi 模块的这种类加载器结构，使得一个类的不同版本可以共存在 Java 虚拟机中，带来了很大的灵活性。不过它的这种不同，也会给开发人员带来一些麻烦，尤其当模块需要使用第三方提供的库的时候。下面提供几条比较好的建议：

- 如果一个类库只有一个模块使用，把该类库的 jar 包放在模块中，在 Bundle-ClassPath 中指明即可。
- 如果一个类库被多个模块共用，可以为这个类库单独的创建一个模块，把其它模块需要用到的 Java 包声明为导出的。其它模块声明导入这些类。
- 如果类库提供了 SPI 接口，并且利用线程上下文类加载器来加载 SPI 实现的 Java 类，有可能会找不到 Java 类。如果出现了 NoClassDefFoundError 异常，首先检查当前线程的上下文类加载器是否正确。通过 Thread.currentThread().getContextClassLoader() 就可以得到该类加载器。该类加载器应该是该模块对应的类加载器。如果不是的话，可以首先通过 class.getClassLoader() 来得到模块对应的类加载器，再通过 Thread.currentThread().setContextClassLoader() 来设置当前线程的上下文类加载器。

## 总结

类加载器是 Java 语言的一个创新。它使得动态安装和更新软件组件成为可能。本文详细介绍了类加载器的相关话题，包括基本概念、代理模式、线程上下文类加载器、与 Web 容器和 OSGi 的关系等。开发人员在遇到 ClassNotFoundException 和 NoClassDefFoundError 等异常的时候，应该检查抛出异常的类的类加载器和当前线程的上下文类加载器，从中可以发现问题的所在。在开发自己的类加载器的时候，需要注意与已有的类加载器组织结构的协调。

## 下载

| 描述       | 名字                              | 大小    |
|----------|---------------------------------|-------|
| 类加载器示例代码 | <a href="#">classloader.zip</a> | 13 KB |

## 参考资料

## 学习

- “The Java Language Specification”的第 12 章“Execution”和“The Java Virtual Machine Specification”的第 5 章“Loading, Linking, and Initializing” 详细介绍了 Java 类的加载、链接和初始化。
- “developerWorks 教程：了解 Java ClassLoader”：概述了 Java ClassLoader，并指

导您构造在装入代码之前自动编译代码的示例 ClassLoader。

- “[OSGi Service Platform Core Specification](#)”: OSGi 规范文档
- “[The Apache Tomcat 5.5 Servlet/JSP Container - Class Loader HOW-TO](#)”: 详细介绍了 Tomcat 5.5 中的类加载器机制。
- [技术书店](#): 浏览关于这些和其他技术主题的图书。
- [developerWorks Java 技术专区](#): 数百篇关于 Java 编程各个方面文章。

## 获得产品和技术

下载 [IBM 软件试用版](#), 体验强大的 DB2®, Lotus®, Rational®, Tivoli®和 WebSphere® 软件。

## 讨论

加入 [developerWorks 社区](#)。

查看 [developerWorks 博客](#) 的最新信息。

# 尼古拉斯

# Android中插件开发篇之----动态加载Activity(免安装运行程序)

来源:[尼古拉斯](#)

## 一、前言

又到周末了，时间过的很快，今天我们来看一下Android中插件开发篇的最后一篇文章的内容：动态加载Activity(免安装运行程序)，在上一篇文章中说道了，如何动态加载资源(应用换肤原理解析)，没看过的同学，可以转战：

<http://blog.csdn.net/jiangwei0910410003/article/details/47679843>

当然，今天说道的内容还这这篇文章有关系。关于动态加载Activity的内容，网上也是有很多文章介绍了。但是他们可能大部分都是介绍通过代理的方式去实现的，所以今天我要说的加载会有两种方式：

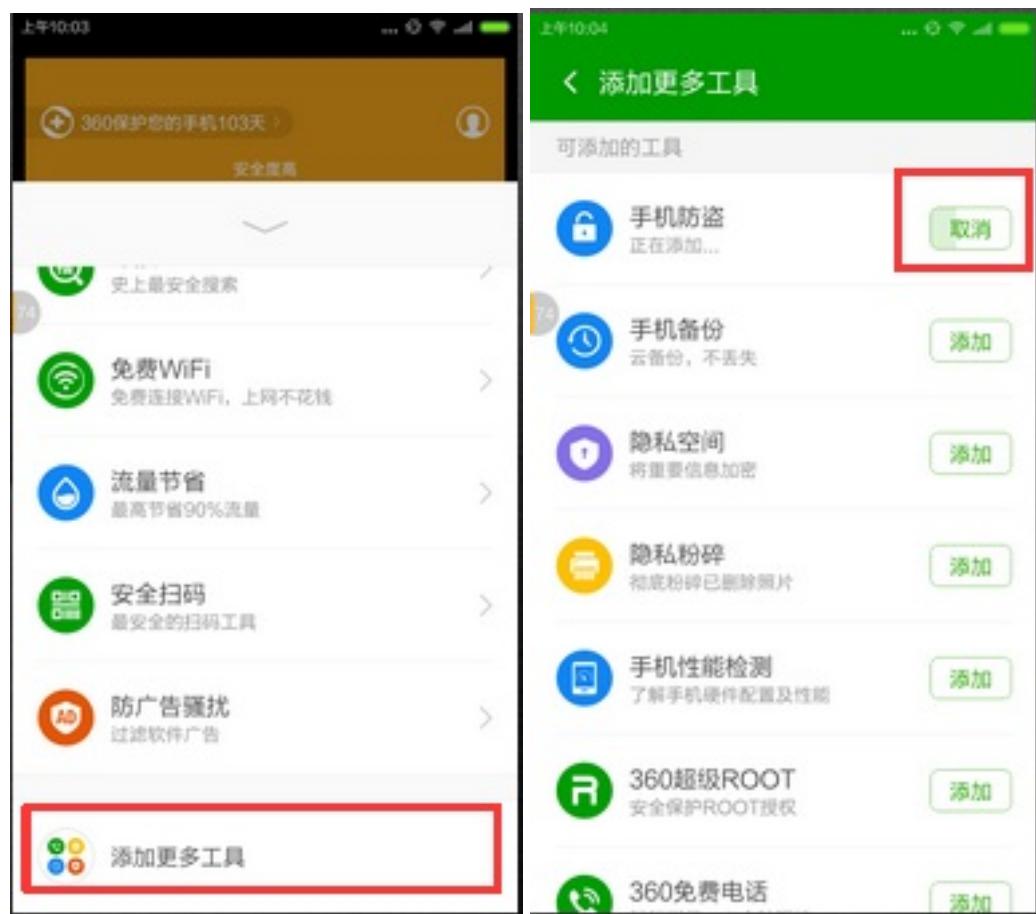
- 1、使用反射机制修改类加载器
- 2、使用代理的方式

这两种方式都有各自的优缺点，我会在后面的文章详细解说。

## 二、技术介绍

1、第一种方式：使用反射机制修改类加载器来实现动态加载Activity

首先来看一个例子：360安全卫士



在主界面有一个添加更多工具的菜单，点进去之后，可以看到有很多功能选项。我们添加一个手机防盗的功能：有一个进度条开始添加。那么我们如何知道他是使用动态加载的呢？我们可以去查看他的数据文件目录：

```
C:\Users\i>adb shell
shell@cancro:/ $ su
su
root@cancro:/ # cd data/data/com.qihoo360.mobilesafe
cd data/data/com.qihoo360.mobilesafe
cd data/data/com.qihoo360.mobilesafe
root@cancro:/data/data/com.qihoo360.mobilesafe # ls
ls
ls
app_gabage
app_plugins_v3
app_plugins_v3_odex ←这个目录应该是存放释放后的Dex
app_webview
auth.jar
cache
databases
files
lib
shared_prefs
root@cancro:/data/data/com.qihoo360.mobilesafe # cd app_plugins_v3
cd app_plugins_v3
cd app_plugins_v3
root@cancro:/data/data/com.qihoo360.mobilesafe/app_plugins_v3 # ls
ls
ls
adblockui-10-10-107.jar
antitheft-10-10-101.jar
antivirus-10-10-505.jar
apull-10-10-110.jar
barcode-10-10-103.jar
blockui-10-10-101.jar
callshowmgr-10-10-102.jar
clean-10-10-209.jar
mainp-10-10-101.jar
nt-10-10-140.jar
ntsvc-10-10-140.jar
persistp-10-10-101.jar
push-10-10-126.jar
root-10-10-101.jar
seccstore-10-10-104.jar
shield-10-10-110.jar
shieldui-10-10-122.jar
wifiexam-10-10-110.jar
root@cancro:/data/data/com.qihoo360.mobilesafe/app_plugins_v3 # _
```

我们可以看到有两个目录，比较见名知意：

app\_plugins\_v3  
app\_plugins\_v3\_odex

第一个目录是存放需要动态加载的功能插件，第二个目录是存放加载之后释放的dex目录。

上面分析了360的动态加载Activity功能，下面我们就来实现以下这个功能吧：

不过我们还得了解一下Android中的类加载器的相关知识，这里就不做介绍了：我在这篇文章中详细介绍了类加载器：

<http://blog.csdn.net/jiangwei0910410003/article/details/41384667>

我们知道PathClassLoader是一个应用的默认加载器(而且他只能加载data/app/xxx.apk的文件)，但是我们加载插件一般使用DexClassLoader加载器，所以这里就有问题了，其实如果对于开始的时候，每个人都会认为很简单，很容易想到使用DexClassLoader来加载Activity获取到class对象，在使用Intent启动，这个很简单呀？但是实际上并不是想象的这么简单。原因很简单，因为Android中的四大组件都有一个特点就是他们有自己的启动流程和生命周期，我们使用DexClassLoader加载进来的Activity是不会涉及到任何启动流程和生命周期的概念，说白了，他就是一个普普通通的类。所以启动肯定会出现错误。

所以我们知道了问题所在，解决起来也就简单了，但是我们这里还有两种思路去解决这个问题：

1) 第一个思路：替换LoadedApk中的mClassLoader 我们只要让加载进来的Activity有启动流程和生命周期就OK了，所以这里我们需要看一下一个Activity的启动过程。当然这里不会详细介绍一个Activity启动过程的，因为那个太复杂了，而且我也说不清楚，我们知道我们可以将我们使用的DexClassLoader加载器绑定到系统加载Activity的类加载器上就可以了，这个是我们的思路。也是最重要的突破点。下面我们就来通过源码看看如何找到加载Activity的类加载器。

加载Activity的时候，有一个很重要的类：LoadedApk.java，这个类是负责加载一个Apk程序的，我们可以看一下他的源码：

```

/**
 * Local state maintained about a currently loaded .apk.
 * @hide
 */
public final class LoadedApk {

 private static final String TAG = "LoadedApk";

 private final ActivityThread mActivityThread;
 private final ApplicationInfo mApplicationInfo;
 final String mPackageName;
 private final String mAppDir;
 private final String mResDir;
 private final String[] mOverlayDirs;
 private final String[] mSharedLibraries;
 private final String mDataDir;
 private final String mLibDir;
 private final File mDataDirFile;
 private final ClassLoader mBaseClassLoader;
 private final boolean mSecurityViolation; 一个Apk加载的类加载器
 private final boolean mIncludeCode;
 private final DisplayAdjustments mDisplayAdjustments = new DisplayAdjustments();
 Resources mResources;
 private ClassLoader mClassLoader; ←
 private Application mApplication;

 private final ArrayMap<Context, ArrayMap<BroadcastReceiver, ReceiverDispatcher>> mReceivers
 = new ArrayMap<Context, ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>>();
 private final ArrayMap<Context, ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>> mUnregisteredReceivers
 = new ArrayMap<Context, ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>>();
 private final ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>> mServices
 = new ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>>();
 private final ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>> mUnboundServices
 = new ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>>();

 int mClientCount = 0;

 Application getApplication() {
 return mApplication;
 }
}

```

我们可以看到他内部有一个mClassLoader变量，他就是负责加载一个Apk程序的，那么我们只要获取到这个类加载器就可以了。他不是static的，所以我们还得获取一个LoadedApk对象。我们在去看一下另外一个类：ActivityThread.java的源码

```

// set of instantiated backup agents, keyed by package name
final ArrayMap<String, BackupAgent> mBackupAgents = new ArrayMap<String, BackupAgent>();
/** Reference to singleton {@link ActivityThread} */
private static ActivityThread sCurrentActivityThread;
Instrumentation mInstrumentation,
String mInstrumentationAppDir = null;
String mInstrumentationAppLibraryDir = null;
String mInstrumentationAppPackage = null;
String mInstrumentedAppDir = null;
String mInstrumentedAppLibraryDir = null;
boolean mSystemThread = false;
boolean mJitEnabled = false;

// These can be accessed by multiple threads; mPackages is the lock.
// XXX For now we keep around information about all packages we have
// seen, not removing entries from this map.
// NOTE: The activity and window managers need to call in to
// ActivityThread to do things like update resource configurations,
// which means this lock gets held while the activity and window manager
// holds their own lock. Thus you MUST NEVER call back into the activity manager
// or window manager or anything that depends on them while holding this lock.
final ArrayMap<String, WeakReference<LoadedApk>> mPackages
 = new ArrayMap<String, WeakReference<LoadedApk>>();
final ArrayMap<String, WeakReference<LoadedApk>> mResourcePackages
 = new ArrayMap<String, WeakReference<LoadedApk>>();
final ArrayList<ActivityClientRecord> mRelaunchingActivities
 = new ArrayList<ActivityClientRecord>();
Configuration mPendingConfiguration = null;

```

这里我们可以看到ActivityThread类中有一个自己的static对象，然后还有一个ArrayMap存放Apk包名和LoadedApk映射关系的数据结构，那么我们分析清楚了，下面就来通过反射来获取mClassLoader对象吧。

**友情提示：**这里可能有些同学会困惑，怎么能够找到这个mClassLoader呢。我在这里因为是为了讲解内容，所以反过来找这个东西了，其实正常情况下，我们在找关于一个Apk或者是Activity的相关信息的时候，特别是启动流程的时候，我们肯定会去找：

ActivityThread.java这个类，这个类是很重要很重要的，也是关键的突破口，它内部其实有很多信息的，所以，我们应该先去找这个ActivityThread,然后从这个类中发现信息，然后会找到了LoadedApk这个类。关于ActivityThread这个类，为何如此重要，我们可以在看看他的源码：

```

public static void main(String[] args) {
 SamplingProfilerIntegration.start();

 // CloseGuard defaults to true and can be quite spammy. We
 // disable it here, but selectively enable it later (via
 // StrictMode) on debug builds, but using DropBox, not logs.
 CloseGuard.setEnabled(false);

 Environment.initForCurrentUser();

 // Set the reporter for event logging in libcore
 EventLogger.setReporter(new EventLoggingReporter());

 Security.addProvider(new AndroidKeyStoreProvider());

 Process.setArgV0("<pre-initialized>");

 Looper.prepareMainLooper();

 ActivityThread thread = new ActivityThread();
 thread.attach(false);

 if (sMainThreadHandler == null) {
 sMainThreadHandler = thread.getHandler();
 }

 AsyncTask.init();

 if (false) {
 Looper.myLooper().setMessageLogging(new
 LogPrinter(Log.DEBUG, "ActivityThread"));
 }
}

```

他有这个方法？这个方法看到很熟悉呀？他不是Java程序运行的入口main方法吗？是的，没错，所有的app程序的执行入口就是这里，所以以后有人问你Android中程序运行的入口是哪里？不要再说Application的onCreate方法了，其实是ActivityThread中的main方法。

好的，回到主干上来，我们现在开始编写代码来实现反射获取mClassLoader类，然后将其替换成我们的DexClassLoader类，不多说了，看一下Demo工程结构：

- PluginActivity1 ==》 插件工程
- DynamicActivityForClassLoader ==》 宿主工程

其中PluginActivity1工程很简单啦，就一个Activity：

```
package com.example.dynamicactivityapk;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Toast;

public class MainActivity extends Activity {

 private static View parentView;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 if(parentView == null){
 setContentView(R.layout.activity_main);
 }else{
 setContentView(parentView);
 }

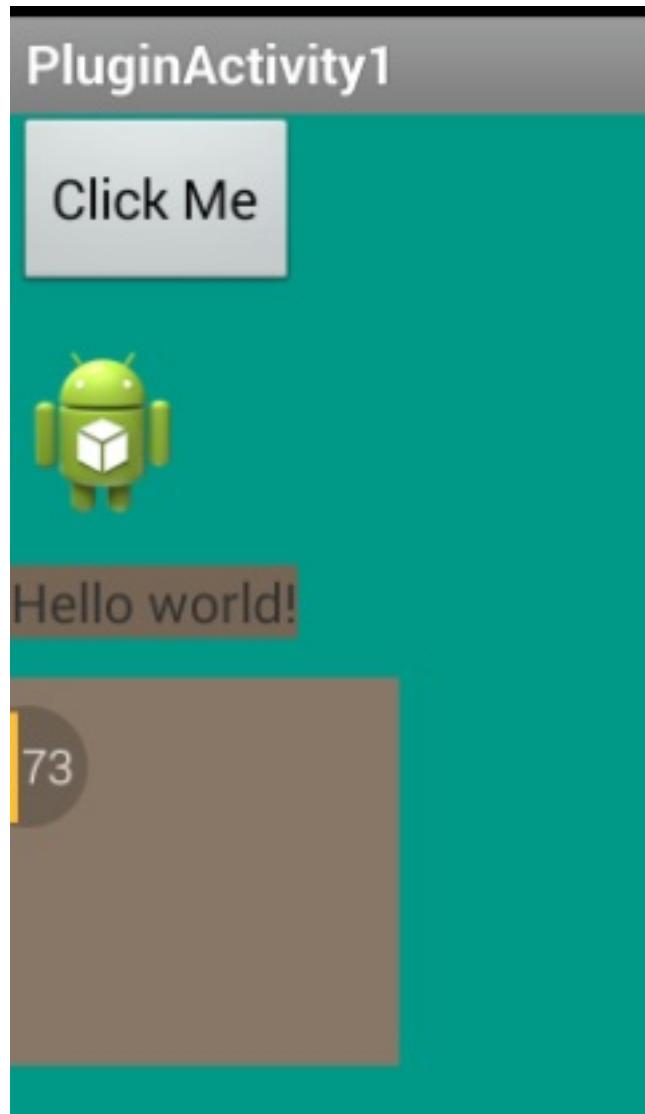
 findViewById(R.id.btn).setOnClickListener(new OnClickListener(){

 @Override
 public void onClick(View arg0) {
 Toast.makeText(getApplicationContext(), "I came from 插件~~", Toast.LENGTH_SHORT);
 }
 }

 public static void setLayoutView(View view){
 parentView = view;
 }
 }
}
```

我们看到其实这里有一个问题，为何要定义一个setLayoutView的方法，这个我们后面会说道。

我们编译这个工程，得到PluginActivity1.apk程序：



下面来看一下宿主工程

宿主工程其实最大的功能就是加载上面的PluginActivity1.apk，然后启动内部的MainActivity就可以了，这里的代码就是如何通过反射替换系统的mClassLoader类：

```

@SuppressLint("NewApi")
private void loadApkClassLoader(DexClassLoader dLoader){
 try{

 String filesDir = this.getCacheDir().getAbsolutePath();
 String libPath = filesDir + File.separator +"PluginActivity1.apk";

 // 配置动态加载环境
 Object currentActivityThread = RefInvoke.invokeStaticMethod(
 "android.app.ActivityThread", "currentActivityThread",
 new Class[] {}, new Object[] {});//获取主线程对象
 String packageName = this.getPackageName();//当前apk的包名
 ArrayMap mPackages = (ArrayMap) RefInvoke.getFieldObject(
 "android.app.ActivityThread", currentActivityThread,
 "mPackages");
 WeakReference wr = (WeakReference) mPackages.get(packageName);
 RefInvoke.setFieldObject("android.app.LoadedApk", "mClassLoader",
 wr.get(), dLoader);

 Log.i("demo", "classloader:"+dLoader);

 }catch(Exception e){
 Log.i("demo", "load apk classloader error:"+Log.getStackTraceString(e));
 }
}

```

这里有一个参数就是需要替换的DexClassLoader的，从外部传递过来，然后进行替换。我们看看外部定义的DexClassLoader类：

```

String filesDir = this.getCacheDir().getAbsolutePath();
String libPath = filesDir + File.separator +"PluginActivity1.apk";
Log.i("inject", "fileexist:"+new File(libPath).exists());

//loadResources(libPath);

DexClassLoader loader = new DexClassLoader(libPath, filesDir, filesDir, getClassLoader)

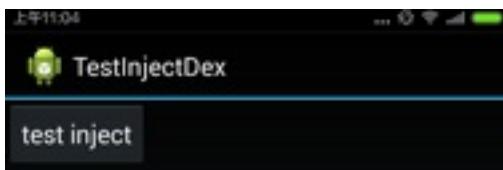
```

这里，需要注意的是，DexClassLoader的最后一个参数，是DexClassLoader的parent，这里需要设置成PathClassLoader类，因为我们上面虽然说是替换PathClassLoader为DexClassLoader。但是PathClassLoader是系统本身默认的类加载器(也就是mClassLoader变量的值，我们如果单独的将DexClassLoader设置为mClassLoader的值的话，就会出错的)，所以一定要讲DexClassLoader的父加载器设置成PathClassLoader，因为类加载器是符合双亲委派机制的。

下面我们来运行一下这个程序，首先我们将PluginActivity1.apk放到宿主工程的data/data/cache目录下：

```
C:\Users\i>adb push C:\Users\i\Desktop\PluginActivity1.apk /data/data/com.example.testinjectdex/cache
1364 KB/s <44720 bytes in 0.032s>
```

运行程序，点击加载：



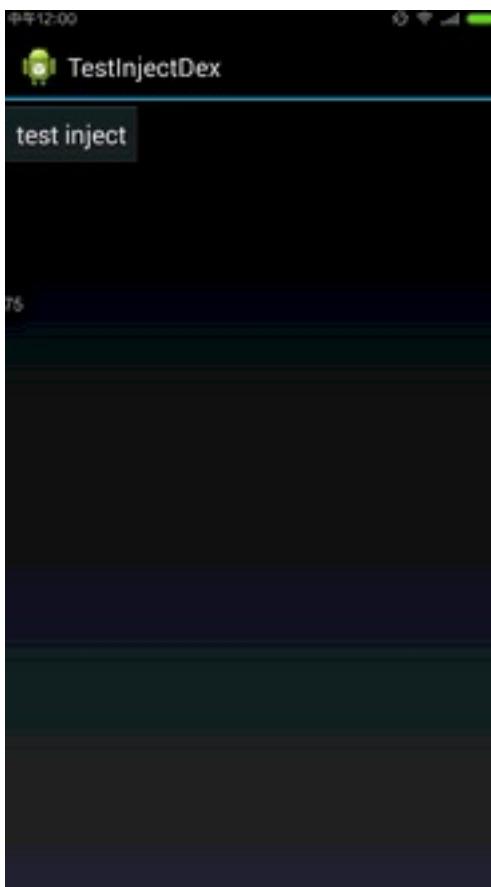
运行发现失败，说是插件中的那个**MainActivity**没有在**AndroidManifest.xml**中声明？不对呀，我们在插件工程中明明声明了呀，为何他还是提示没有声明呢？



哎，其实仔细想想原因很简单的，因为DexClassLoader加载插件Apk，不会将其xml中的内容加载进来，所以在插件中声明是没有任何用途的，必须在宿主工程中声明：

```
<activity
 android:name="com.example.dynamicactivityapk.MainActivity">
</activity>
```

我们在运行程序：



我们点击按钮，看效果啦。果然可以加载成功了啦啦啦。很开心了。

不过这里还是需要注意两个问题：

- 1>、因为要加载插件中的资源，所以需要调用loadResources方法
  - 2>、在测试的过程中，发现插件工程中setContentView方法没有效果了。所以就在插件工程中定义一个static的方法，用来提前设置视图的。
- 2) 第二思路：合并PathClassLoader和DexClassLoader中的dexElements数组**

好了，这里就介绍了一个如何使用反射机制来动态加载一个Activity了，但是到这里还没有结束呢？因为还要介绍另外一种方式来设置类加载器。

我们首先来看一下PathClassLoader和DexClassLoader类加载器的父类  
BaseDexClassLoader的源码：

(这里需要注意的是PathClassLoader和DexClassLoader类的父加载器是  
BootClassLoader,他们的父类是BaseDexClassLoader)

```
/**
 * Base class for common functionality between various dex-based
 * {@link ClassLoader} implementations.
 */
public class BaseDexClassLoader extends ClassLoader {
 /** originally specified path (just used for {@code toString()}) */
 private final String originalPath;

 /** originally specified library path (just used for {@code toString()}) */
 private final String originalLibraryPath;

 /** structured lists of path elements */
 private final DexPathList pathList;

 /**
 * Constructs an instance.
 *
 * @param dexPath the list of jar/apk files containing classes and
 * resources, delimited by {@code File.pathSeparator}, which
 * defaults to {@code ":"} on Android
 * @param optimizedDirectory directory where optimized dex files
 * should be written; may be {@code null}
 * @param libraryPath the list of directories containing native
 * libraries, delimited by {@code File.pathSeparator}; may be
 * {@code null}
 * @param parent the parent class loader
 */
 public BaseDexClassLoader(String dexPath, File optimizedDirectory,
 String libraryPath, ClassLoader parent) {
 super(parent);

 this.originalPath = dexPath;
 this.originalLibraryPath = libraryPath;
 this.pathList =
 new DexPathList(this, dexPath, libraryPath, optimizedDirectory);
 }
}
```

这里有一个DexPathList对象，在来看一下DexPathList.java源码：

```

/**
 * A pair of lists of entries, associated with a {@code ClassLoader}.
 * One of the lists is a dex/resource path — typically referred
 * to as a "class path" — list, and the other names directories
 * containing native code libraries. Class path entries may be any of:
 * a {@code .jar} or {@code .zip} file containing an optional
 * top-level {@code classes.dex} file as well as arbitrary resources,
 * or a plain {@code .dex} file (with no possibility of associated
 * resources).
 *
 * <p>This class also contains methods to use these lists to look up
 * classes and resources.</p>
 */
/*package*/ final class DexPathList {
 private static final String DEX_SUFFIX = ".dex";
 private static final String JAR_SUFFIX = ".jar";
 private static final String ZIP_SUFFIX = ".zip";
 private static final String APK_SUFFIX = ".apk";

 /** class definition context */
 private final ClassLoader definingContext;

 /** list of dex/resource (class path) elements */
 private final Element[] dexElements;

 /** list of native library directory elements */
 private final File[] nativeLibraryDirectories;
}

```

首先看一下这个类的描述，还有一个Elements数组，我们看到这个变量他是专门存放加载的dex文件的路径的，系统默认的类加载器是PathClassLoader，本身一个程序加载之后会释放一个dex出来，这时候会将dex路径放到里面，当然DexClassLoader也是一样的，那么我们会想到，我们是否可以将DexClassLoader中的dexElements和PathClassLoader中的dexElements进行合并，然后在设置给PathClassLoader中呢？这也是一个思路。我们来看代码：

```

/**
 * 以下是一种方式实现的
 * @param loader
 */
private void inject(DexClassLoader loader){
 PathClassLoader pathLoader = (PathClassLoader) getClassLoader();

 try {
 Object dexElements = combineArray(
 getDexElements(getPathList(pathLoader)),
 getDexElements(getPathList(loader)));
 Object pathList = getPathList(pathLoader);
 setField(pathList, pathList.getClass(), "dexElements", dexElements);
 }
}

```

```

 } catch (IllegalArgumentException e) {
 e.printStackTrace();
 } catch (NoSuchFieldException e) {
 e.printStackTrace();
 } catch (IllegalAccessException e) {
 e.printStackTrace();
 } catch (ClassNotFoundException e) {
 e.printStackTrace();
 }
 }

private static Object getPathList(Object baseDexClassLoader)
 throws IllegalArgumentException, NoSuchFieldException, IllegalAccessException
ClassLoader bc = (ClassLoader)baseDexClassLoader;
return getField(baseDexClassLoader, Class.forName("dalvik.system.BaseDexClassLoader"));
}

private static Object getField(Object obj, Class<?> cl, String field)
 throws NoSuchFieldException, IllegalArgumentException, IllegalAccessException
Field localField = cl.getDeclaredField(field);
localField.setAccessible(true);
return localField.get(obj);
}

private static Object getDexElements(Object paramObject)
 throws IllegalArgumentException, NoSuchFieldException, IllegalAccessException
return getField(paramObject, paramObject.getClass(), "dexElements");
}

private static void setField(Object obj, Class<?> cl, String field,
 Object value) throws NoSuchFieldException,
 IllegalArgumentException, IllegalAccessException {

 Field localField = cl.getDeclaredField(field);
 localField.setAccessible(true);
 localField.set(obj, value);
}

private static Object combineArray(Object arrayLhs, Object arrayRhs) {
 Class<?> localClass = arrayLhs.getClass().getComponentType();
 int i = Array.getLength(arrayLhs);
 int j = i + Array.getLength(arrayRhs);
 Object result = Array.newInstance(localClass, j);
 for (int k = 0; k < j; ++k) {
 if (k < i) {
 Array.set(result, k, Array.get(arrayLhs, k));
 } else {
 Array.set(result, k, Array.get(arrayRhs, k - i));
 }
 }
 return result;
}

```

我们在运行宿主程序，发现发现也是可以的，这里就不演示了，效果都是一样的。这里总结一下：

我们在使用反射机制来动态加载Activity的时候，有两个思路：

1>、替换LoadApk类中的mClassLoader变量的值，将我们动态加载类DexClassLoader设置为mClassLoader的值

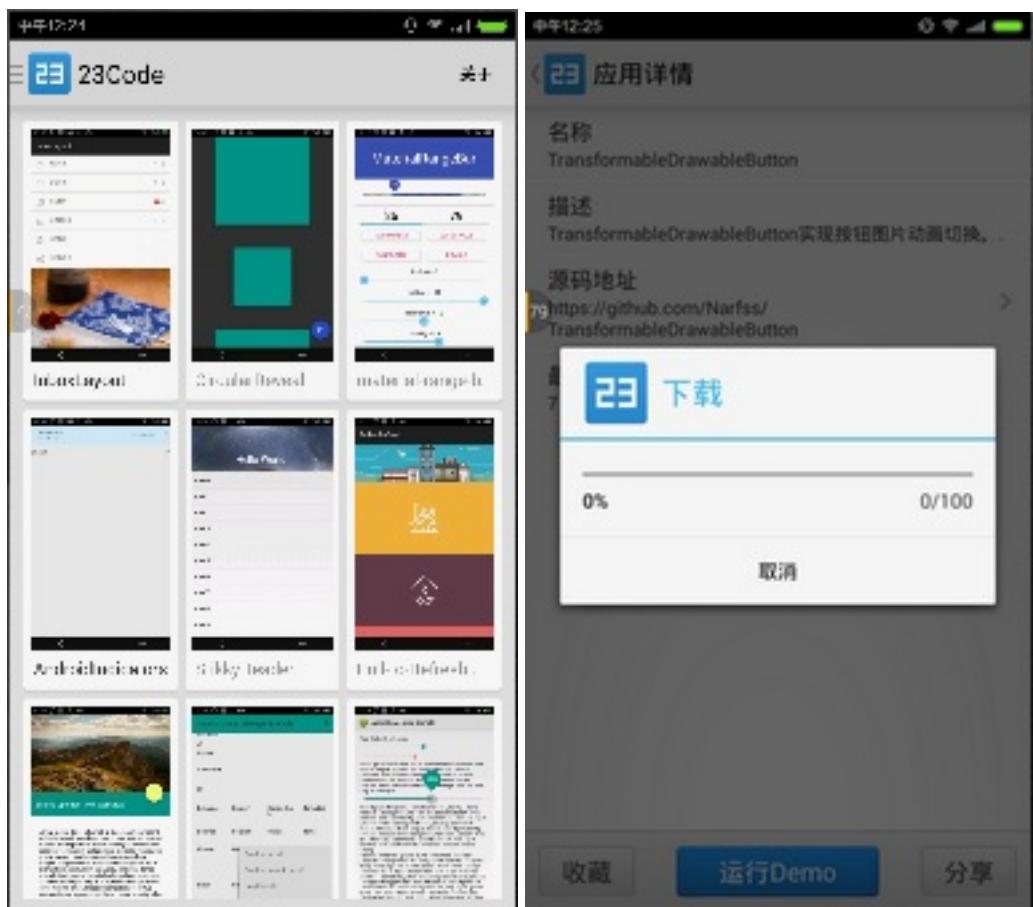
2>、合并系统默认加载器PathClassLoader和动态加载器DexClassLoader中的dexElements数组 这两个的思路原理都是一样的：就是让我们动态加载进来的Activity能够具备正常的启动流程和生命周期。

项目下载地址：<http://download.csdn.net/detail/jiangwei0910410003/9063377>

## 2、第二种方式来动态加载Activity：静态代理的方式

首先我们也是先来看一个例子：23Code

这个应用的功能就是实时的展示一些开源的UI控件。他是在线下载，然后动态加载进行展示的：



我们看到，点击运行Demo的时候，他会去下载apk,我们看看他的数据目录结构：

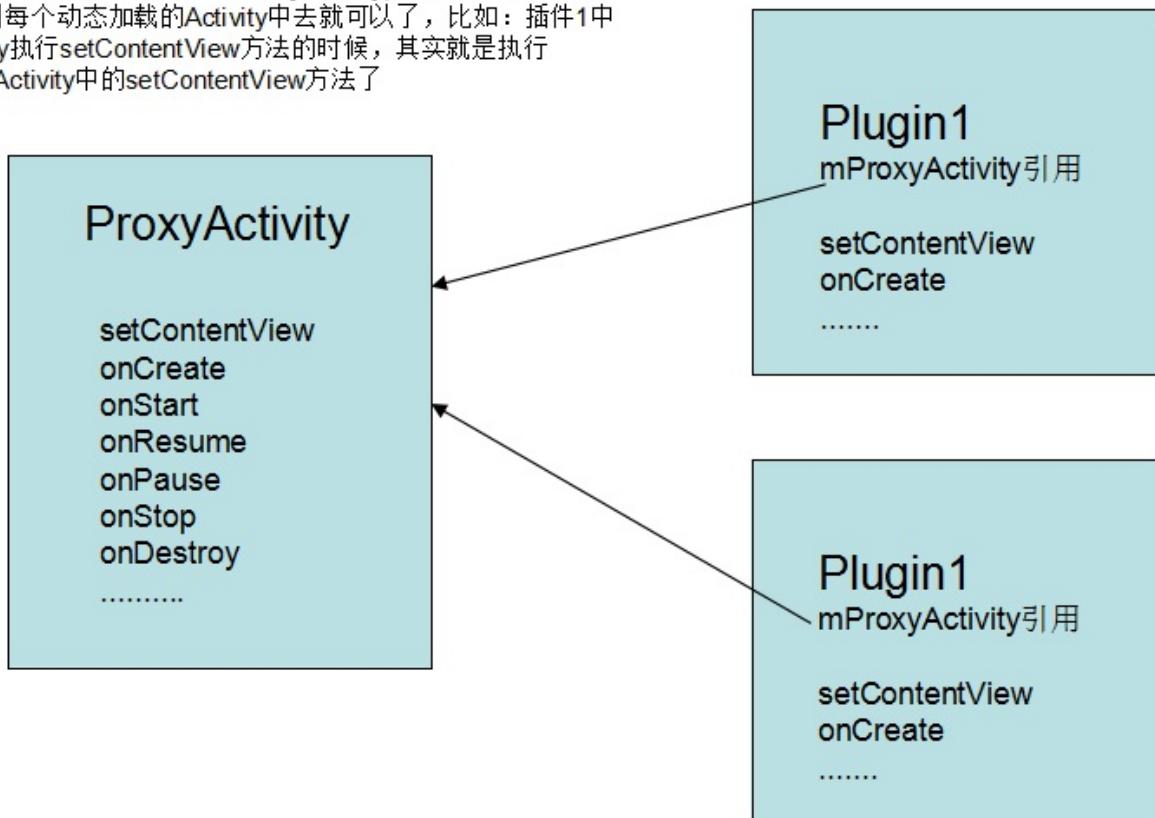
```
C:\Users\i>adb shell
shell@cancro:/ $ su
su
root@cancro:/ # cd /data/data/com.ttcode.appdirect
cd /data/data/com.ttcode.appdirect
cd /data/data/com.ttcode.appdirect
root@cancro:/data/data/com.ttcode.appdirect # ls
ls
ls
app_dex
app_outdex
app_webview
cache
com.ttcode.appdirect-1.dex
com.ttcode.appdirect-2.dex
databases
files
lib
shared_prefs
root@cancro:/data/data/com.ttcode.appdirect # cd files
cd files
cd files
root@cancro:/data/data/com.ttcode.appdirect/files # ls
ls
ls
CacheTime.dat
com.andtinder.demo
com.appyvet.rangebarsample
com.astuetz.viewpager.extensions.sample
com.atermenji.android.iconicroid.sample
com.balysv.material.drawable.menu.demo
com.chiemy.cardview
com.ecloud.pulltozoomview.demo
com.elliott.chenger.sizeadjustingtextview
com.example.android.wizardpager
com.example.croppersample
com.example.htmltextview
com.flavienlaurent.notboringactionbar
com.fmsirvent.transformablebuttonsample
com.github.ksoichiro.android.observablescrollview.samples
com.gnod.parallaxlistview
com.indris.sample
com.kmshack.newsstand
com.larswerkman.quickreturnlistview
com.manuelpeinado.fadingactionbar.demo
com.mattkula.secrettextview
com.moshx.androidindicators
com.mrbug.pulltoidmisspager.example
com.ortiz.touch
com.soundcloud.android.crop.example
com.storm.swiperefreshlayoutdemo
com.tonicartos.superslimexample
com.xxmassdeveloper.mpchartexample
com.yalantis.pulltorefresh.sample
de.androidpit.androidcolorthief.sample
de.ankri
io.codetail.circularrevealsample
it.carlom.stickyheader.example
it.gmariotti.cardslib.demo
me.yugy.github.residelayout
root@cancro:/data/data/com.ttcode.appdirect/files # cd de.ankri
cd de.ankri
```

这里我们看到了，他把下载之后的apk都用每个插件的功能包名存起来的。

好了，上面分析了23Code的加载机制，我们来看看如何使用代理的方式来动态加载Activity

先来看看原理：

这里的ProxyActivity就是代理对象，而每个插件Activity都是被代理对象，每个插件Activity对象中都会有一个代理Activity的对象引用，这个就是静态代理，原理就很简单了，就是插件Activity实际上不是真正意义上的Activity了，而是他们把所有Activity中的任务用代理对象Activity来执行了，所以我们只需要在宿主工程中声明一个ProxyActivity，然后用反射的方法设置到每个动态加载的Activity中去就可以了，比如：插件1中Activity执行setContentView方法的时候，其实就是执行ProxyActivity中的setContentView方法了



所以说，这种方式来加载Activity的话，其实真正意义上每个插件的Activity都不再是像方式一中的那样，没有生命周期，没有启动流程了，他们就是一个普通的Activity类，然后将其生命周期的所有任务都交给代理Activity去执行就可以了。

下面我们来看一下项目工程：

1、DynamicActivityForProxy ==》宿主工程

2、PluginActivity2 ==》插件工程

看一下插件工程

BaseActivity.java

```
package com.example.dynamicactivity;

import com.example.dynamicactivity.R;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Toast;

public class BaseActivity extends Activity {
 protected Activity mProxyActivity;

 public void setProxy(Activity proxyActivity) {
 mProxyActivity = proxyActivity;
 }

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 }

 @Override
 public void setContentView(int layoutResID) {
 if (mProxyActivity != null && mProxyActivity instanceof Activity) {
 mProxyActivity.setContentView(layoutResID);
 mProxyActivity.findViewById(R.id.btn).setOnClickListener(
 new OnClickListener() {
 @Override
 public void onClick(View v) {
 Toast.makeText(mProxyActivity, "我是插件，你是谁!", Toast.LENGTH_SHORT);
 }
 });
 }
 }
}
```

这里会重写setContentView的方法，同时有一个setProxy方法。

在来看一下MainActivity.java

```
package com.example.dynamicactivity;

import com.example.dynamicactivity.R;

import android.os.Bundle;
import android.util.Log;

public class MainActivity extends BaseActivity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 }

 @Override
 protected void onDestroy() {
 //这里需要注意的是，不能调用super.onDestroy的方法了，不然报错，原因也很简单，这个Activit
 Log.i("demo", "onDestory");
 }

 @Override
 protected void onPause() {
 Log.i("demo", "onPause");
 }

 @Override
 protected void onResume() {
 Log.i("demo", "onResume");
 }

 @Override
 protected void onStart() {
 Log.i("demo", "onStart");
 }

 @Override
 protected void onStop() {
 Log.i("demo", "onStop");
 }
}
```

这里打印一下生命周期中的每个方法，待会需要验证。

注意：这里的生命周期方法不能再调用super.XXX方法了，否则会报错的，原因很简单啦。插件Activity不在是真正意义上的Activity了，就是一个空壳的Activity。所以调用的话，肯定会出错

运行插件工程，得到一个PluginActivity2.apk

下面来看一下宿主工程：

首先我们来看一下重要的代理对象ProxyActivity

```
package com.example.dynamic.activity;

import java.io.File;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.util.HashMap;

import android.annotation.SuppressLint;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import dalvik.system.DexClassLoader;

public class ProxyActivity extends BaseActivity {

 private Object pluginActivity;
 private Class<?> pluginClass;

 private HashMap<String, Method> methodMap = new HashMap<String, Method>();

 @SuppressLint("NewApi")
 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 try {
 DexClassLoader loader = initClassLoader();

 //动态加载插件Activity
 pluginClass = loader.loadClass("com.example.dynamicactivity.MainActivity");
 Constructor<?> localConstructor = pluginClass.getConstructor(new Class[] {});
 pluginActivity = localConstructor.newInstance(new Object[] {});

 //将代理对象设置给插件Activity
 Method setProxy = pluginClass.getMethod("setProxy", new Class[] { Activity.class });
 setProxy.setAccessible(true);
 setProxy.invoke(pluginActivity, new Object[] { this });

 initMethodMap();

 //调用它的onCreate方法
 Method onCreate = pluginClass.getDeclaredMethod("onCreate",
 new Class[] { Bundle.class });
 onCreate.setAccessible(true);
 onCreate.invoke(pluginActivity, new Object[] { new Bundle() });
 }
 }
}
```

```

 } catch (Exception e) {
 Log.i("demo", "load activity error:"+Log.getStackTraceString(e));
 }
 }

/**
 * 存储每个生命周期的方法
 */
private void initMethodMap(){
 methodMap.put("onPause", null);
 methodMap.put("onResume", null);
 methodMap.put("onStart", null);
 methodMap.put("onStop", null);
 methodMap.put("onDestroy", null);

 for(String key : methodMap.keySet()){
 try{
 Method method = pluginClass.getDeclaredMethod(key);
 method.setAccessible(true);
 methodMap.put(key, method);
 }catch(Exception e){
 Log.i("demo", "get method error:"+Log.getStackTraceString(e));
 }
 }
}

@SuppressWarnings("NewApi")
private DexClassLoader initClassLoader(){
 String filesDir = this.getCacheDir().getAbsolutePath();
 String libPath = filesDir + File.separator +"PluginActivity2.apk";
 Log.i("inject", "fileexist:"+new File(libPath).exists());
 loadResources(libPath);
 DexClassLoader loader = new DexClassLoader(libPath, filesDir, null , getClass());
 return loader;
}

@Override
protected void onDestroy() {
 super.onDestroy();
 Log.i("demo", "proxy onDestroy");
 try{
 methodMap.get("onDestroy").invoke(pluginActivity, new Object[]{});
 }catch(Exception e){
 Log.i("demo", "run destroy error:"+Log.getStackTraceString(e));
 }
}

@Override
protected void onPause() {
 super.onPause();
 Log.i("demo", "proxy onPause");
}

```

```
try{
 methodMap.get("onPause").invoke(pluginActivity, new Object[]{});
}catch(Exception e){
 Log.i("demo", "run pause error:"+Log.getStackTraceString(e));
}

@Override
protected void onResume() {
 super.onResume();
 Log.i("demo", "proxy onResume");
 try{
 methodMap.get("onResume").invoke(pluginActivity, new Object[]{});
 }catch(Exception e){
 Log.i("demo", "run resume error:"+Log.getStackTraceString(e));
 }
}

@Override
protected void onStart() {
 super.onStart();
 Log.i("demo", "proxy onStart");
 try{
 methodMap.get("onStart").invoke(pluginActivity, new Object[]{});
 }catch(Exception e){
 Log.i("demo", "run start error:"+Log.getStackTraceString(e));
 }
}

@Override
protected void onStop() {
 super.onStop();
 Log.i("demo", "proxy onStop");
 try{
 methodMap.get("onStop").invoke(pluginActivity, new Object[]{});
 }catch(Exception e){
 Log.i("demo", "run stop error:"+Log.getStackTraceString(e));
 }
}

}
```

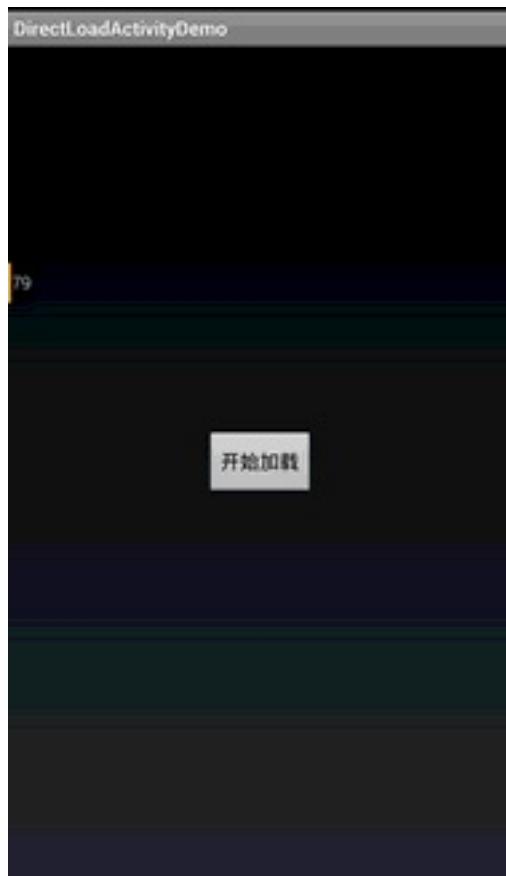
这里主要就是：

- 1、加载插件Activity
- 2、使用反射将代理对象设置给插件Activity
- 3、测试插件Activity中的生命周期方法

运行程序，我们将上面的PluginActivity2.apk放到宿主程序的cache目录下

```
C:\Users\i>adb push C:\Users\i\Desktop\PluginActivity2.apk /data/data/com.example.dynamic.activity/cache
5491 KB/s (843584 bytes in 0.150s)
C:\Users\i>
```

运行：



也是成功了，看到效果啦啦，同时我们打印一下Log日志：

```
C:\Users\i>adb logcat -s demo
----- beginning of /dev/log/main
----- beginning of /dev/log/system
I/demo < 8597>: proxy onPause
I/demo < 8597>: onPause
I/demo < 8597>: proxy onStop
I/demo < 8597>: onStop
I/demo < 8597>: proxy onDestroy
I/demo < 8597>: onDestroy
I/demo < 8597>: proxy onStart
I/demo < 8597>: onStart
I/demo < 8597>: proxy onResume
I/demo < 8597>: onResume
I/demo < 8597>: proxy onPause
I/demo < 8597>: onPause
I/demo < 8597>: proxy onStop
I/demo < 8597>: onStop
I/demo < 8597>: proxy onDestroy
I/demo < 8597>: onDestroy
```

插件Activity的每个生命周期的方法也是都执行了。

到这里我们就讲解完了使用代理的方式来实现动态加载Activity。这种方式其实还是很简单的。

项目下载地址：<http://download.csdn.net/detail/jiangwei0910410003/9063483>

总算是讲完了，累死了，两种方式各有千秋，各有各的好处。

### 三、案例分析

上面讲解的两种方式的时候，介绍了两个例子：一个是360卫士，一个是23Code为什么要用这两个例子呢？原因下载说明一下啦：

1、首先来看一下360卫士，我们打开它的一个辅助功能：

使用：`adb shell dumpsys activity top`

```
C:\Users\i>adb shell dumpsys activity top
TASK com.qihoo360.mobilesafe id=188
 ACTIVITY com.qihoo360.mobilesafe/.loader.a.ActivityNiNR0 439261f0 pid=30827
 Local FragmentActivity 43d93c58 state:
 mCreated=true mResumed=true mStopped=false mReallyStopped=false
 mLoadersStarted=true
 Active Fragments in 43681160:
 #0: dt<437dec40 #0 id=0x7f090023>
 mFragmentId=#7f090023 mContainerId#=7f090023 mTag=null
 mState=5 mIndex=0 mWho=android:fragment:0 mBackStackNesting=0
 mAdded=true mRemoving=false mResumed=true mFromLayout=false mInLayout=false
 mHidden=false mDetached=false mMenuVisible=true mHasMenu=false
 mRetainInstance=false mRetaining=false mUserVisibleHint=true
 mFragmentManager=FragmentManager<43681160 in AdBlockMainActivity@43d93c58>
 mActivity=com.qihoo.antivirus.adblock.ui.AdBlockMainActivity@43d93c58
 mContainer=android.widget.FrameLayout<43785480 V.E..... D 0,0-1080,1920 #7f090023 app:id/>
 mView=adblockui.y@438652f0 V.E..... D 0,0-1080,1920>
 mInnerView=adblockui.y@438652f0 V.E..... D 0,0-1080,1920>
 Added Fragments:
 #0: dt<437dec40 #0 id=0x7f090023>
 FragmentManager misc state:
 mCurState=5 mStateSaved=false mDestroyed=false

C:\Users\i>
```

我们再来看一下他的AndroidManifest.xml内容：

```
<activity
 android:theme="@android:0103000B"
 android:name="com.qihoo360.mobilesafe.loader.a.ActivityN1SID1"
 android:exported="false"
 android:launchMode="3"
 android:screenOrientation="1"
 android:configChanges="0x000000B0"
 >
</activity>
<activity
 android:theme="@android:0103000B"
 android:name="com.qihoo360.mobilesafe.loader.a.ActivityN1SID2"
 android:exported="false"
 android:launchMode="3"
 android:screenOrientation="1"
 android:configChanges="0x000000B0"
 >
</activity>
<activity
 android:theme="@android:01030006"
 android:name="com.qihoo360.mobilesafe.loader.a.ActivityN1NR0"
 android:exported="false"
 android:screenOrientation="1"
 android:configChanges="0x000000B0"
 >
</activity>
<activity
 android:theme="@android:01030006"
 android:name="com.qihoo360.mobilesafe.loader.a.ActivityN1NR1"
 android:exported="false"
 android:screenOrientation="1"
 android:configChanges="0x000000B0"
 >
</activity>
<activity
 android:theme="@android:01030006"
 android:name="com.qihoo360.mobilesafe.loader.a.ActivityN1NR2"
 android:exported="false"
 android:screenOrientation="1"
 android:configChanges="0x000000B0"
 >
</activity>
<activity
 android:theme="@android:01030006"
 android:name="com.qihoo360.mobilesafe.loader.a.ActivityN1NR3"
 android:exported="false"
 android:screenOrientation="1"
 android:configChanges="0x000000B0"
```

我们可以看到，他在宿主工程中声明了，而且我们会看到有很多个辅助功能都声明了。所以我们断定他使用的是第一种方式去实现动态加载的，所以我们可以看到这种方式有一个不好就是：需要在宿主工程中声明很多个插件Activity。

## 2、在来看一下23Code应用

运行一个例子之后，我们看到他的Activity是TestActivity

```
C:\Users\i>adb shell dumpsys activity top
TASK com.ttcode.appdirect id=189
ACTIVITY com.ttcode.appdirect/.TestActivity 43ef8788 pid=1066
 Local FragmentActivity 43cb82d8 State:
 mCreated=true mResumed=true mStopped=false mReallyStopped=false
 mLoadersStarted=true
 Active Fragments in 43cb8538:
 #0: TestFragment<43d405c0 #0 id=0x7f090043 android:switcher:2131296323:0>
 mFragmentId=#7f090043 mContainerId=#7f090043 mTag=android:switcher:2131296323:0
 mState=5 mIndex=0 mWho=android:fragment:0 mBackStackNesting=0
 mAdded=true mRemoving=false mResumed=true mFromLayout=false mInLayout=false
 mHidden=false mDetached=false mMenuVisible=true mHasMenu=false
 mRetainInstance=false mRetaining=false mUserVisibleHint=true
 mFragmentManager=FragmentManager<43cb8538 in MainActivity<43cb82d8>>
 mActivity=com.moshx.androidindicators.MainActivity@43cb82d8
 mArguments=Bundle[<pos=0>]
 mContainer=android.support.v4.view.ViewPager<437a4258 UFED....I. 0,498-1080,1812 #7f090043 app:id/viewPager>
 mView=android.support.v4.app.NoSaveStateFrameLayout<43d42cf8 U.E....I. 0,0-1080,1314>
 mInnerView=android.support.v4.app.NoSaveStateFrameLayout<43d42cf8 U.E....I. 0,0-1080,1314>
 ChildFragmentManager<43d41348 in TestFragment<43d405c0>>:
 FragmentManager misc state:
 mActivity=com.moshx.androidindicators.MainActivity@43cb82d8
```

我们再去切换另外一个例子运行之后，也是发现还是这个TestActivity。看看他的AndroidManifest.xml

```
<activity
 android:name="com.ttcode.appdirect.TestActivity"
 >
 <intent-filter
 >
 <action
 android:name="com.youba.barcode"
 >
 </action>
 <category
 android:name="android.intent.category.DEFAULT"
 >
 </category>
 </intent-filter>
 </activity>
```

那么我们断定，这个TestActivity就是一个代理的Activity,他使用的是第二种方式来实现动态加载的。现在知道为何我开始的时候为什么用这两个例子来说明了吧。

## 四、两种方式的比较

### 第一种方式：使用反射机制来实现

优点：可以不用太多的关心插件中的Activity的生命周期方法，因为他加载进来之后就是一个真正意义上的Activity了

缺点：需要在宿主工程中进行声明，如果插件中的Activity多的话，那么就不灵活了。

### 第二种方式：使用代理机制来实现

优点：不需要在宿主工程中进行声明太多的Activity了，只需要有一个代理Activity的声明就可以了，很灵活

缺点：需要管理手动的去管理插件中Activity的生命周期方法，难度复杂。

## 五、存在的问题

我们看到上面讲到的两种方式去动态加载Activity,其实两种方式都还存在很多问题：

- 1、其他组件的动态加载问题(服务，广播，ContentProvier)
- 2、跨进程访问的问题

## 六、总结

这篇文章讲完之后，那么插件开发篇的三部曲就算结束了，Android中的插件开发也算是有一个好的总结了，也是讲解了现在主流市场中加载的原理和机制。当然我们在真正的使用过程中还会存在很多问题，当然这个就需要我们自己去探索和解决了。

(PS：两种方式的项目下载地址都在上面给出了，如果在运行的过程中有什么问题的话，请留言。能帮就尽量帮助解决一下~~)

# Android中插件开发篇之----应用换肤原理解析

来源:[尼古拉斯](#)

## 一、前言

今天又到周末了，感觉时间过的很快呀.又要写blog了。那么今天就来看看应用的换肤原理解析。在之前的一篇博客中我说道了Android中的插件开发篇的基础：类加载器的相关知识。没看过的同学可以转战：

<http://blog.csdn.net/jiangwei0910410003/article/details/41384667>

## 二、原理介绍

现在市场上有很多应用都有换肤的功能，就是能够提供给用户一些皮肤包，然后下载，替换。而且有些皮肤是要收费的。对于这个功能的话，其实没有什么技术难度的，但是他包含了一个现阶段很火的一个技术：动态加载

好了，既然说到了动态加载，那么如果有不熟悉的同学，可以转战看另外的一篇blog了：

<http://blog.csdn.net/jiangwei0910410003/article/details/17679823>

我们先来看一个市场上的一个app具有的换肤功能的例子：QQ空间

点击我的空间=>个性化=>原创主题=>选择下载主题



下载主题，然后可以替换了。接下来我们看看这个主题包放到哪了？因为既然下载肯定是要存放起来了。

两个地方可以放：一个是SD卡，一个是应用的数据目录

我们先来看看应用的目录(配置好了adb命令)：

## 第一步：得到QQ空间的包名：

打开QQ空间app,不要退出。然后执行命令： adb shell dumpsys activity top

```
C:\Users\viadb shell dumpsys activity top
TASK com.qzone
ACTIVITY com.qzonex.app.tab.QZoneTabActivity 4358d888 pid=8179
 Local Activity 438b3390 State:
 mResumed=true mStopped=false mFinished=false
 mLoadersStarted=true
 mChangingConfigurations=false
 mCurrentConfig={1.0 460mcc7mnc zh_CN ldltr sw360dp w360dp h620dp 480dpi nrml long port finger -keyb/u/h -nav/h s.5 themeChanged=0 themeChangedFlags=0}
 FragmentManager misc state:
 mActivity=com.qzonex.app.tab.QZoneTabActivity@438b3390
 mContainer=android.app.Activity$1@459dc4b0
 mCurState=5 mStateSaved=false mDestroyed=false
 ViewRoot:
 mAdded=true mRemoved=false
```

这个命令还是很有用的吧，能够快速的得到一个应用的包名

我们看到QQ空间的包名：com.qzone

## 第二步：进入QQ应用的目录，查看对应的资源

```
root@cancro:/data/data/com.qzone/cache # cd ..
cd ..
cd ..
root@cancro:/data/data/com.qzone # cd shared_prefs
cd shared_prefs
cd shared_prefs
root@cancro:/data/data/com.qzone/shared_prefs # ls
ls
ls
HSPK_com.qzone.xml
HSPK_com.qzone:service.xml
MLOGIN_DEVICE_INFO.xml
com.qzone_default_pref_0.xml
com.qzone_preferences.xml
com.qzone_preferences0.xml
com.qzone_public_settings.xml
downloa_stragegy.xml
guarder.xml
options.for.com.qzone.xml
safe_mode_com.qzone.xml
safe_mode_com.qzone:service.xml
tbs_download_config.xml
tbs_download_stat.xml
the_pweadx5_check_cfg_file.xml
theme.xml
wns.config.start.xml
wns_share_data.xml
root@cancro:/data/data/com.qzone/shared_prefs # cat theme.xml
cat theme.xml
cat theme.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
 <boolean name="compiled" value="true" />
 <string name="pending_theme_root">/data/user/0/com.qzone/files/cache/qz_external_resource/theme_res/38/res/</string>
 <boolean name="pending_theme_resources_compiled" value="true" />
 <string name="theme_root">/data/user/0/com.qzone/files/cache/qz_external_resource/theme_res/38/res/</string>
</map>
root@cancro:/data/data/com.qzone/shared_prefs #
```

我们在他的shared\_prefs中找到了theme.xml文件，查看该文件，就可以找到了对应皮肤的位置：

/data/user/0/com.qzone/files/cache/qz\_external\_resource/theme\_res/38

我们进入到这个目录：

```
root@cancro:/ # cd /data/user/0/com.qzone/files/cache/qz_external_resource/theme_res/38
cd /data/user/0/com.qzone/files/cache/qz_external_resource/theme_res/38
/cache/qz_external_resource/theme_res/38 <
root@cancro:/data/user/0/com.qzone/files/cache/qz_external_resource/theme_res/38 # ls
ls
ls
AndroidManifest.xml
classes.dex
res
resources.arsc
root@cancro:/data/user/0/com.qzone/files/cache/qz_external_resource/theme_res/38 # cd res
cd res
cd res
root@cancro:/data/user/0/com.qzone/files/cache/qz_external_resource/theme_res/38/res # ls
ls
ls
color
drawable
drawable-xhdpi
root@cancro:/data/user/0/com.qzone/files/cache/qz_external_resource/theme_res/38/res # cd drawable
cd drawable
cd drawable
root@cancro:/data/user/0/com.qzone/files/cache/qz_external_resource/theme_res/38/res/drawable # ls
ls
ls
b1.png
b1_click.png
b1_disable.png
b2.png
b2_click.png
b2_disable.png
b3.png
b3_click.png
```

看到上面红色圈起来的地方的目录结构和文件名是不是很眼熟.对，这个就是我们把一个正常的apk解压之后得到的东西。那么我们可以断定，QQ空间的皮肤包其实就是一个apk，然后动态加载apk，取到对应的资源然后替换。

### 三、如何设计一个换肤插件

好了，既然上面我们解读了QQ空间的换肤功能，也知道了它的大体的原理了，下面我们来自动手制作我们自己的主题包。

关于动态加载的相关技术这里就不详细介绍了，看我的前面提到的两个相关文章的介绍。

我们这里需要建立三个工程：

```
宿主程序(主程序): ResourceLoader
主题包1的工程: ResourceLoaderApk1
主题包2的工程: ResourceLoaderApk2
```

在宿主程序中我们需要编写动态加载的代码：

下面来看一下具体代码：

### MainActivity.java

```
package com.example.resourceloader;

import java.io.File;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

import android.annotation.SuppressLint;
import android.content.Context;
import android.graphics.drawable.Drawable;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ImageView;
import android.widget.LinearLayout;
import android.widget.TextView;

public class MainActivity extends BaseActivity {

 /**
 * 需要替换主题的控件
 * 这里就列举三个： TextView, ImageView, LinearLayout
 */
 private TextView textV;
 private ImageView imgV;
 private LinearLayout layout;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);

 textV = (TextView) findViewById(R.id.text);
 imgV = (ImageView) findViewById(R.id.imageview);
 layout = (LinearLayout) findViewById(R.id.layout);

 findViewById(R.id.btn1).setOnClickListener(new OnClickListener(){

 @Override
 public void onClick(View arg0) {
 String filesDir = getCacheDir().getAbsolutePath();
 String filePath = filesDir + File.separator + "apk1.apk";
 Log.i("Loader", "filePath:" + filePath);
 Log.i("Loader", "isExist:" + new File(filePath).exists());
 //loadResources(filePath);
 //setContent();
 //printResourceId();
 }
 });
 }
}
```

```

 setContent1();
 //printRField();
 });

findViewById(R.id.btn2).setOnClickListener(new OnClickListener(){

 @Override
 public void onClick(View v) {
 String filesDir = getCacheDir().getAbsolutePath();
 String filePath = filesDir + File.separator +"apk2.apk";
 //loadResources(filePath);
 setContent();
 });
}

/**
 * 动态加载主题包中的资源，然后替换每个控件
 */
@SuppressLint("NewApi")
private void setContent(){
 try{
 Class clazz = classLoader.loadClass("com.example.resourceloaderapk.UIUtil");
 Method method = clazz.getMethod("getTextString", Context.class);
 String str = (String)method.invoke(null, this);
 textV.setText(str);
 method = clazz.getMethod("getImageDrawable", Context.class);
 Drawable drawable = (Drawable)method.invoke(null, this);
 imgV.setBackground(drawable);
 method = clazz.getMethod("getLayout", Context.class);
 View view = (View)method.invoke(null, this);
 layout.addView(view);
 }catch(Exception e){
 Log.i("Loader", "error:"+Log.getStackTraceString(e));
 }
}

/**
 * 另外的一种方式获取
 */
private void setContent1(){
 int stringId = getTextStringId();
 int drawableId = getImgDrawableId();
 int layoutId = getLayoutId();
 Log.i("Loader", "stringId:"+stringId+",drawableId:"+drawableId+",layoutId:"+layoutId);
}

@SuppressLint("NewApi")
private int getTextStringId(){
 try{
 Class clazz = classLoader.loadClass("com.example.resourceloaderapk1.R$string");
 Field field = clazz.getField("app_name");
 int resId = (int)field.get(null);
 }
}

```

```
 return resId;
 }catch(Exception e){
 Log.i("Loader", "error:"+Log.getStackTraceString(e));
 }
 return 0;
}

@SuppressLint("NewApi")
private int getImgDrawableId(){
 try{
 Class clazz = classLoader.loadClass("com.example.resourceloaderapk1.R$drawa
 Field field = clazz.getField("ic_launcher");
 int resId = (int)field.get(null);
 return resId;
 }catch(Exception e){
 Log.i("Loader", "error:"+Log.getStackTraceString(e));
 }
 return 0;
}

@SuppressLint("NewApi")
private int getLayoutId(){
 try{
 Class clazz = classLoader.loadClass("com.example.resourceloaderapk1.R$layo
 Field field = clazz.getField("activity_main");
 int resId = (int)field.get(null);
 return resId;
 }catch(Exception e){
 Log.i("Loader", "error:"+Log.getStackTraceString(e));
 }
 return 0;
}

@SuppressLint("NewApi")
private void printResourceId(){
 try{
 Class clazz = classLoader.loadClass("com.example.resourceloaderapk.UIUtil
 Method method = clazz.getMethod("getTextStringId", null);
 Object obj = method.invoke(null, null);
 Log.i("Loader", "stringId:"+obj);
 Log.i("Loader", "newId:"+R.string.app_name);
 method = clazz.getMethod("getImageDrawableId", null);
 obj = method.invoke(null, null);
 Log.i("Loader", "drawableId:"+obj);
 Log.i("Loader", "newId:"+R.drawable.ic_launcher);
 method = clazz.getMethod("getLayoutId", null);
 obj = method.invoke(null, null);
 Log.i("Loader", "layoutId:"+obj);
 Log.i("Loader", "newId:"+R.layout.activity_main);
 }catch(Exception e){
 Log.i("Loader", "error:"+Log.getStackTraceString(e));
 }
}
```

```
}

private void printRField(){
 Class clazz = R.id.class;
 Field[] fields = clazz.getFields();
 for(Field field : fields){
 Log.i("Loader", "fields:"+field);
 }
 Class clazzs = R.layout.class;
 Field[] fieldss = clazzs.getFields();
 for(Field field : fieldss){
 Log.i("Loader", "fieldss:"+field);
 }
}

}
```

这里的代码没有大的难度，就是我们使用**DexClassLoader**类加载每个主题的apk包，然后用反射的方法调用apk包中的方法来获取资源。

下面来看一下主题包工程代码：

UIUtil.java

```
package com.example.resourceloaderapk;

import android.content.Context;
import android.graphics.drawable.Drawable;
import android.view.LayoutInflater;
import android.view.View;

import com.example.resourceloaderapk1.R;

public class UIUtil {

 public static String getTextString(Context ctx){
 return ctx.getResources().getString(R.string.app_name);
 }

 public static Drawable getImageDrawable(Context ctx){
 return ctx.getResources().getDrawable(R.drawable.ic_launcher);
 }

 public static View getLayout(Context ctx){
 return LayoutInflater.from(ctx).inflate(R.layout.activity_main, null);
 }

 public static int getTextStringId(){
 return R.string.app_name;
 }

 public static int getImageDrawableId(){
 return R.drawable.ic_launcher;
 }

 public static int getLayoutId(){
 return R.layout.activity_main;
 }

}
```

这个类就是提供给外部的获取资源的方法，我们在宿主程序中也就是反射这个方法来获取资源的，这个方法中我们提供了两种方式获取资源：一种是直接返回资源的内容，还有一种是返回一个资源的Id。

关于主题包2的工程这里就不介绍了，代码是一样的，只是资源不一样。

我们运行两个主题包，得到两个apk

```
ResourceLoaderApk1.apk
ResourceLoaderApk2.apk
```

这时候我们使用adb push命令，将这两个apk放到宿主程序的cache目录下。

```
C:\Users\i>adb push C:\Users\i\Desktop\ResourceLoaderApk1.apk /data/data/com.example.resourceloader/cache
5083 KB/s (843376 bytes in 0.162s)

C:\Users\i>adb push C:\Users\i\Desktop\ResourceLoaderApk2.apk /data/data/com.example.resourceloader/cache
4760 KB/s (843413 bytes in 0.173s)

C:\Users\i>
```

温馨提示：

这里不能将需要加载的apk放到非宿主程序的沙盒目录外，不然会加载失败，抛出异常。关于程序的沙盒目录概念其实很好理解：就是/data/data/xxx.xxx/目录，就是这个目录是这个程序所独有的，其他没有共享权限的app是不能访问的(当然除了获取root权限外)，这个其实也很好理解为何要这么做，Google也是为了安全，自己需要加载的apk/dex/jar就应当被保护起来。

```
C:\Users\i>adb shell
shell@cancro:/ $ su
su
root@cancro:/ # cd data/data/com.example.resourceloader
cd data/data/com.example.resourceloader
cd data/data/com.example.resourceloader
root@cancro:/data/data/com.example.resourceloader # cd cache
cd cache
cd cache
root@cancro:/data/data/com.example.resourceloader/cache # ls
ls
ls
root@cancro:/data/data/com.example.resourceloader/cache # ls
ls
ls
ResourceLoaderApk1.apk
ResourceLoaderApk2.apk
com.android.opengl.shaders_cache
root@cancro:/data/data/com.example.resourceloader/cache #
```

当然这里不一定要放到cache目录下，只要是沙盒目录下都可以，新建一个目录也是可以的。不过一般都是使用cache目录。

项目地址：<http://download.csdn.net/detail/jiangwei0910410003/9008423>

这时候我们运行宿主程序：



两个btn，可以加载不同的主题内容，但是问题来啦。。点击之后发现没有效果，捕获异常，我们打印log看看：adb logcat -s Loader

```
C:\Users\i>adb logcat -s Loader
----- beginning of /dev/log/main
----- beginning of /dev/log/system
I/Loader <18983>: filePath:/data/data/com.example.resourceloader/cache/ResourceLoaderApk1.apk
I/Loader <18983>: isExist:true
I/Loader <18983>: error:java.lang.reflect.InvocationTargetException
I/Loader <18983>: at java.lang.reflect.Method.invokeNative(Native Method)
I/Loader <18983>: at java.lang.reflect.Method.invoke(Method.java:515)
I/Loader <18983>: at com.example.resourceloader.MainActivity.setContent(MainActivity.java:82)
I/Loader <18983>: at com.example.resourceloader.MainActivity.access$0(MainActivity.java:78)
I/Loader <18983>: at com.example.resourceloader.MainActivity$1.onClick(MainActivity.java:56)
I/Loader <18983>: at android.view.View.performClick(View.java:4444)
I/Loader <18983>: at android.view.View$PerformClick.run(View.java:18440)
I/Loader <18983>: at android.os.Handler.handleCallback(Handler.java:733)
I/Loader <18983>: at android.os.Handler.dispatchMessage(Handler.java:95)
I/Loader <18983>: at android.os.Looper.loop(Looper.java:136)
I/Loader <18983>: at android.app.ActivityThread.main(ActivityThread.java:5047)
I/Loader <18983>: at java.lang.reflect.Method.invokeNative(Native Method)
I/Loader <18983>: at java.lang.reflect.Method.invoke(Method.java:515)
I/Loader <18983>: at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:806)
I/Loader <18983>: at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:622)
I/Loader <18983>: at dalvik.system.NativeStart.main(Native Method)
I/Loader <18983>: Caused by: android.content.res.Resources$NotFoundException: String resource ID #0x7f0a000d
I/Loader <18983>: at android.content.res.Resources.getText(Resources.java:252)
I/Loader <18983>: at android.content.res.MiuiResources.getText(MiuiResources.java:107)
I/Loader <18983>: at android.content.res.Resources.getString(Resources.java:338)
I/Loader <18983>: at com.example.resourceloaderapk.UIUtil.getTextString(UIUtil.java:13)
I/Loader <18983>: ... 16 more
```

他说找不到资源异常，我们来分析一下。

我们在主题1工程中，调用的是主题1apk中的资源R.string.xxx

但是我们知道获取资源的时候是用Resource类来得到的，对于一个程序来说一个Context只会持有一个Resource对象，但是我们加载apk的时候，主题apk没有得到对应的Context。因为动态加载不想正常的运行一个程序，每个程序都有一个全局的Context变量，但是加载出来的话是没有的。那有人就说了：在代码里面我们用反射的方式去获取的时候不是将宿主的Context变量传递过去了吗？

```

/*
@SuppressLint("NewApi")
private void setContent(){
 try{
 Class clazz = classLoader.loadClass("com.example.resourceloaderapk.UIUtil");
 Method method = clazz.getMethod("getTextString", Context.class);
 String str = (String)method.invoke(null, this);
 textView.setText(str);
 method = clazz.getMethod("getImageDrawable", Context.class);
 Drawable drawable = (Drawable)method.invoke(null, this);
 imgView.setBackground(drawable);
 method = clazz.getMethod("getLayout", Context.class);
 View view = (View)method.invoke(null, this);
 layout.addView(view);
 }catch(Exception e){
 Log.i("Loader", "error:"+Log.getStackTraceString(e));
 }
}

```

对，看上去是没有任何问题，但是这里其实还是那个问题：就是宿主的Context如何能加载插件apk中的资源，我们知道一个app的工程的资源文件都会映射到R文件中，而这个R文件的包名则是这个应用的包名，一个包名一般对应一个Context。那么我们现在即使将宿主的Context传递过去，也是对应宿主的包名，也就是找到宿主工程的R文件，所以还是找不到对应的资源。其实我们要解决的问题就是将插件apk中资源添加到宿主apk中。这时候就需要用一种方式了，采用反射的机制：

通过调用AssetManager中的addAssetPath方法，我们可以将一个apk中的资源加载到Resources中，由于addAssetPath是隐藏api我们无法直接调用，所以只能通过反射，下面是它的声明，通过注释我们可以看出，传递的路径可以是zip文件也可以是一个资源目录，而apk就是一个zip，所以直接将apk的路径传给它，资源就加载到AssetManager中了，然后再通过AssetManager来创建一个新的Resources对象，这个对象就是我们可以使用的apk中的资源了。

我们看一下代码：

```

protected void loadResources(String dexPath) {
 try {
 AssetManager assetManager = AssetManager.class.newInstance();
 Method addAssetPath = assetManager.getClass().getMethod("addAssetPath", S
 addAssetPath.invoke(assetManager, dexPath);
 mAssetManager = assetManager;
 } catch (Exception e) {
 e.printStackTrace();
 }
 Resources superRes = super.getResources();
 superRes.getDisplayMetrics();
 superRes.getConfiguration();
 mResources = new Resources(mAssetManager, superRes.getDisplayMetrics(), superR
 mTheme = mResources.newTheme();
 mTheme.setTo(super.getTheme());
}

```

参数就是需要加载资源的包的路径。

当然我们还需要重写Context的三个方法：

```

@Override
public AssetManager getAssets() {
 return mAssetManager == null ? super.getAssets() : mAssetManager;
}

@Override
public Resources getResources() {
 return mResources == null ? super.getResources() : mResources;
}

@Override
public Theme getTheme() {
 return mTheme == null ? super.getTheme() : mTheme;
}

```

重写的这三个方法就是让系统获取我们加载apk包之后的变量即可

这里，我们把代码在修改一下：在宿主工程中添加一个 BaseActivity类：

**BaseActivity.java**

```

package com.example.resourceloader;

import java.io.File;
import java.lang.reflect.Method;

```

```
import android.app.Activity;
import android.content.res.AssetManager;
import android.content.res.Resources;
import android.content.res.Resources.Theme;
import android.os.Bundle;
import dalvik.system.DexClassLoader;

public class BaseActivity extends Activity{

 protected AssetManager mAssetManager;//资源管理器
 protected Resources mResources;//资源
 protected Theme mTheme;//主题

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 }

 protected void loadResources(String dexPath) {
 try {
 AssetManager assetManager = AssetManager.class.newInstance();
 Method addAssetPath = assetManager.getClass().getMethod("addAssetPath", S
 addAssetPath.invoke(assetManager, dexPath);
 mAssetManager = assetManager;
 } catch (Exception e) {
 e.printStackTrace();
 }
 Resources superRes = super.getResources();
 superRes.getDisplayMetrics();
 superRes.getConfiguration();
 mResources = new Resources(mAssetManager, superRes.getDisplayMetrics(),superR
 mTheme = mResources.newTheme();
 mTheme.setTo(super.getTheme());
 }

 @Override
 public AssetManager getAssets() {
 return mAssetManager == null ? super.getAssets() : mAssetManager;
 }

 @Override
 public Resources getResources() {
 return mResources == null ? super.getResources() : mResources;
 }

 @Override
 public Theme getTheme() {
 return mTheme == null ? super.getTheme() : mTheme;
 }

}
```

在MainActivity中调用loadResource方法：

```
findViewById(R.id.btn1).setOnClickListener(new OnClickListener(){
 @Override
 public void onClick(View arg0) {
 String filesDir = getCacheDir().getAbsolutePath();
 String filePath = filesDir + File.separator + "ResourceLoaderApk1.apk";
 Log.i("Loader", "filePath:" + filePath);
 Log.i("Loader", "isExist:" + new File(filePath).exists());
 classLoader = new DexClassLoader(filePath, fileRelease.getAbsolutePath(), null, getClassLoader());
 loadResources(filePath);
 setContentView();
 //printResourceId();
 //setContent1();
 //printRFField();
 }
});

findViewById(R.id.btn2).setOnClickListener(new OnClickListener(){
 @Override
 public void onClick(View v) {
 String filesDir = getCacheDir().getAbsolutePath();
 String filePath = filesDir + File.separator + "ResourceLoaderApk2.apk";
 classLoader = new DexClassLoader(filePath, fileRelease.getAbsolutePath(), null, getClassLoader());
 loadResources(filePath);
 setContentView();
 }
});
```

这时候我们在运行宿主程序：



点击主题1：我们发现文字变成了：ResourceLoaderApk；因为这里的图片都是用的机器人所以看上去没变化，看到底下的LinearLayout加载了主题包中的布局xml内容。

点击主题2：效果同上，只是内容是主题包2apk中的。

好了。到这里我们就完美的开发了我们自己的换肤功能。但是有的同学可能认为，这哪是换肤的功能，没看到效果呢？我这个例子不是完完全全的开发一个换肤的工程。只是介绍原理呢。不过真的换肤也没有难度的，我们需要解决一些问题：

- 1、对于需要替换主题的控件需要统一定义一下。
- 2、对于每个主题包的工程中的对外接口要统一(或者是要符合一定规范), 比如这个例子中主题包中必须有一个: `com.example.resourceloaderapk.UIUtil`类, 而且这个类中必须有三个方法: `getTextString, getImageDrawable, getLayout`所以这就是一个规范, 当然我这里的规范设计的不是很好, 正确的做法是在提供一个接口, 然后每个主题包工程必须实现这个接口, 然后主题包工程和宿主工程都包含这个接口, 这样就能够很灵活了。
- 3、一般主题包apk是从网上下载下来的, 所以我们需要事前设置要几个默认的主题包在本地, 如果从网上下载下来的主题包出现问题了, 我们去加载默认的主题。这样就不会出现任何异常情况。

## 四、问题总结

其实这篇文章我们看到了上面我们其实就是解决一个问题, 就是如何加载主题包apk中的资源。其实这个问题有人还有一种想法, 就是我们将需要的资源全部打包(可以是任何压缩包的格式), 从网上下载下来之后, 解压文件, 通过流的方式读取每个资源文件到工程中, 其实这种方式是可行的, 但是效率上有很大的问题(反正我是没有尝试过)。所以这里的这种方式很方便而且高效。

## 五、实际用途

本文说到的这个技术现在市面上主要的作用就是:

- 1、在线替换主题(皮肤), 语言包等
- 2、减小主apk的包大小, 将不是很重要的资源打包成apk放到服务端。

## 六、总结

这篇文章主要介绍了应用换肤的原理, 核心技术就是: 如何加载插件Apk中的资源。后续还会技术讲解Android中插件的用途: 免安装运行程序, 制作中。。。

# Android中插件开发篇之----类加载器

来源:[尼古拉斯](#)

## 前言

关于插件，已经在各大平台上出现过很多，eclipse插件、chrome插件、3dmax插件，所有这些插件大概都为了在一个主程序中实现比较通用的功能，把业务相关或者可以让用户自定义扩展的功能不附加在主程序中，主程序可在运行时安装和卸载。在Android如何实现插件也已经被广泛传播，实现的原理都是实现一套插件接口，把插件实现编成apk或者dex，然后在运行时使用DexClassLoader动态加载进来，不过在这个开发过程中会遇到很多的问题，所以这一片就先不介绍如何开发插件，而是先解决一下开发过程中会遇到的问题，这里主要就是介绍DexClassLoader这个类使用的过程中出现的错误

## 导读

Java中的类加载器：<http://blog.csdn.net/jiangwei0910410003/article/details/17733153>

Android中的动态加载机

制：<http://blog.csdn.net/jiangwei0910410003/article/details/17679823>

System.loadLibrary的执行过

程：<http://blog.csdn.net/jiangwei0910410003/article/details/41490133>

## 一、预备知识

### Android中的各种加载器介绍

插件开发的过程中DexClassLoader和PathClassLoader这两个类加载器了是很重要的，但是他们也是有区别的，而且我们也知道**PathClassLoader是Android应用中的默认加载器**。他们的区别是：

DexClassLoader可以加载任何路径的apk/dex/jar PathClassLoader只能加载/data/app中的apk，也就是已经安装到手机中的apk。这个也是PathClassLoader作为默认的类加载器的原因，因为一般程序都是安装了，在打开，这时候PathClassLoader就去加载指定的apk(解压成dex，然后在优化成odex)就可以了。

我们可以看一下他们的源码：

```
/*
 * Copyright (C) 2008 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package dalvik.system;

import java.io.File;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.zip.ZipFile;

/**
 * Provides a simple {@link ClassLoader} implementation that operates on a
 * list of jar/apk files with classes.dex entries. The directory that
 * holds the optimized form of the files is specified explicitly. This
 * can be used to execute code not installed as part of an application.
 *
 * The best place to put the optimized DEX files is in app-specific
 * storage, so that removal of the app will automatically remove the
 * optimized DEX files. If other storage is used (e.g. /sdcard), the
 * app may not have an opportunity to remove them.
 */
public class DexClassLoader extends ClassLoader {

 private static final boolean VERBOSE_DEBUG = false;

 /* constructor args, held for init */
 private final String mRawDexPath;
 private final String mRawLibPath;
 private final String mDexOutputPath;

 /*
 * Parallel arrays for jar/apk files.
 *
 * (could stuff these into an object and have a single array;
 * improves clarity but adds overhead)
}
```

```

/*
private final File[] mFiles; // source file Files, for rsrc URLs
private final ZipFile[] mZips; // source zip files, with resources
private final DexFile[] mDexs; // opened, prepped DEX files

/**
 * Native library path.
 */
private final String[] mLibPaths;

/**
 * Creates a {@code DexClassLoader} that finds interpreted and native
 * code. Interpreted classes are found in a set of DEX files contained
 * in Jar or APK files.
 *
 * The path lists are separated using the character specified by
 * the "path.separator" system property, which defaults to ":".
 *
 * @param dexPath
 * the list of jar/apk files containing classes and resources
 * @param dexOutputDir
 * directory where optimized DEX files should be written
 * @param libPath
 * the list of directories containing native libraries; may be null
 * @param parent
 * the parent class loader
 */
public DexClassLoader(String dexPath, String dexOutputDir, String libPath,
 ClassLoader parent) {

 super(parent);
.....

```

我们看到，他是继承了ClassLoader类的，ClassLoader是类加载器的鼻祖类。同时我们也会发现DexClassLoader只有一个构造函数，而且这个构造函数是：dexPath、dexOutDir、libPath、parent

**dexPath：**是加载apk/dex/jar的路径

**dexOutDir：**是dex的输出路径(因为加载apk/jar的时候会解压除dex文件，这个路径就是保存dex文件的)

**libPath：**是加载的时候需要用到的lib库，这个一般不用

**parent：**给DexClassLoader指定父加载器

我们在来看一下PathClassLoader的源码

PathClassLoader.java

```

/*
 * Copyright (C) 2007 The Android Open Source Project

```

```
/*
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package dalvik.system;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.RandomAccessFile;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.zip.ZipEntry;
import java.util.zip.ZipFile;

/**
 * Provides a simple {@link ClassLoader} implementation that operates on a list
 * of files and directories in the local file system, but does not attempt to
 * load classes from the network. Android uses this class for its system class
 * loader and for its application class loader(s).
 */
public class PathClassLoader extends ClassLoader {

 private final String path;
 private final String libPath;

 /*
 * Parallel arrays for jar/apk files.
 *
 * (could stuff these into an object and have a single array;
 * improves clarity but adds overhead)
 */
 private final String[] mPaths;
 private final File[] mFiles;
 private final ZipFile[] mZips;
 private final DexFile[] mDexs;
```

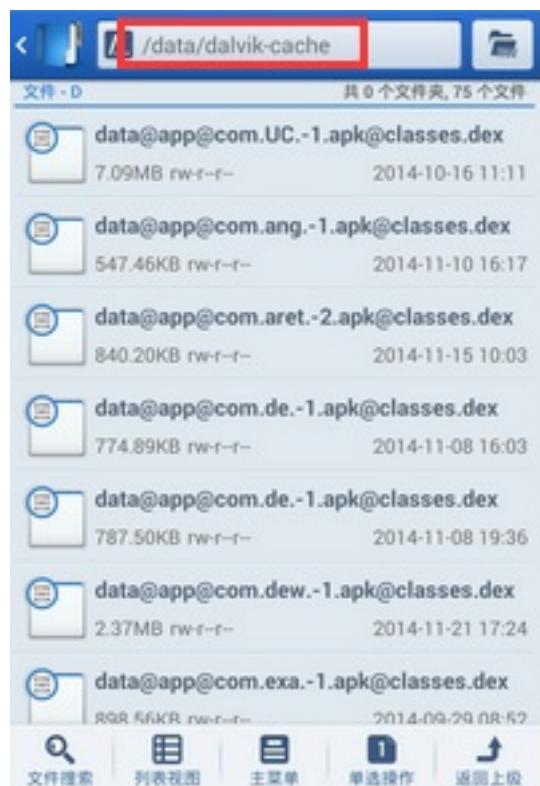
```
/**
 * Native library path.
 */
private final List<String> libraryPathElements;

/**
 * Creates a {@code PathClassLoader} that operates on a given list of files
 * and directories. This method is equivalent to calling
 * {@link #PathClassLoader(String, String, ClassLoader)} with a
 * {@code null} value for the second argument (see description there).
 *
 * @param path
 * the list of files and directories
 *
 * @param parent
 * the parent class loader
 */
public PathClassLoader(String path, ClassLoader parent) {
 this(path, null, parent);
}

/**
 * Creates a {@code PathClassLoader} that operates on two given
 * lists of files and directories. The entries of the first list
 * should be one of the following:
 *
 *
 * Directories containing classes or resources.
 * JAR/ZIP/APK files, possibly containing a "classes.dex" file.
 * "classes.dex" files.
 *
 *
 * The entries of the second list should be directories containing
 * native library files. Both lists are separated using the
 * character specified by the "path.separator" system property,
 * which, on Android, defaults to ":".
 *
 * @param path
 * the list of files and directories containing classes and
 * resources
 *
 * @param libPath
 * the list of directories containing native libraries
 *
 * @param parent
 * the parent class loader
 */
public PathClassLoader(String path, String libPath, ClassLoader parent) {
 super(parent);
 ...
}
```

看到了PathClassLoader类也是继承了ClassLoader的，但是他的构造函数和DexClassLoader有点区别就是，少了一个dexOutDir，这个原因也是很简单的，因为PathClassLoader是加载/data/app中的apk，而这部分的apk都会解压释放dex到指定的目录：

/data/dalvik-cache



这个释放解压操作是系统做的。所以PathClassLoader可以不需要这个参数的。

上面看了他们两的区别，下面在来看一下Android中的各种类加载器分别加载哪些类：

```
package com.example.androidddemo;

import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.util.Log;
import android.widget.ListView;

public class MainActivity extends Activity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);

 Log.i("DEMO", "Context的类加载器:"+Context.class.getClassLoader());
 Log.i("DEMO", "ListView的类加载器:"+ListView.class.getClassLoader());
 Log.i("DEMO", "应用程序默认加载器:"+getClassLoader());
 Log.i("DEMO", "系统类加载器:"+ClassLoader.getSystemClassLoader());
 Log.i("DEMO", "系统类加载器和Context的类加载器是否相等:"+((Context.class.getClassLoader() == ClassLoa
 Log.i("DEMO", "系统类加载器和应用程序默认加载器是否相等:"+getClassLoader() == ClassLo

 Log.i("DEMO", "打印应用程序默认加载器的委派机制:");
 ClassLoader classLoader = getClassLoader();
 while(classLoader != null){
 Log.i("DEMO", "类加载器:"+classLoader);
 classLoader = classLoader.getParent();
 }

 Log.i("DEMO", "打印系统加载器的委派机制:");
 classLoader = ClassLoader.getSystemClassLoader();
 while(classLoader != null){
 Log.i("DEMO", "类加载器:"+classLoader);
 classLoader = classLoader.getParent();
 }

 }
}
```

运行结果：

```

com.example.andro... DEMO Context的类加载器:java.lang.BootClassLoader@415e7fe0
com.example.andro... DEMO ListView的类加载器:java.lang.BootClassLoader@415e7fe0
com.example.andro... DEMO 应用程序默认加载器:dalvik.system.PathClassLoader[DexPathList[[zip file "/data/app/...
example.androidddemo-2.apk"],nativeLibraryDirectories=[/data/app-lib/com.example.an...
le.androidddemo-2, /vendor/lib, /system/lib]]]
com.example.andro... DEMO 系统类加载器:dalvik.system.PathClassLoader[DexPathList[[directory "."],nativeLibrar...
yDirectories=[/vendor/lib, /system/lib]]]
com.example.andro... DEMO 系统类加载器和Context的类加载器是否相等:false
com.example.andro... DEMO 系统类加载器和应用程序默认加载器是否相等:false
ht 打印应用程序默认加载器的委派机制:gwei0910410003
com.example.andro... DEMO 类加载器:dalvik.system.PathClassLoader[DexPathList[[zip file "/data/app/com.example.an...
le.androidddemo-2.apk"],nativeLibraryDirectories=[/data/app-lib/com.example.an...
droiddemo-2, /vendor/lib, /system/lib]]]
com.example.andro... DEMO 类加载器:java.lang.BootClassLoader@415e7fe0
com.example.andro... DEMO 打印系统加载器的委派机制:
com.example.andro... DEMO 类加载器:dalvik.system.PathClassLoader[DexPathList[[directory "."],nativeLibraryD...
irectories=[/vendor/lib, /system/lib]]]
com.example.andro... DEMO 类加载器:java.lang.BootClassLoader@415e7fe0

```

依次来看一下

## 1) 系统类的加载器

```

Log.i("DEMO", "Context的类加载器:"+Context.class.getClassLoader());
Log.i("DEMO", "ListView的类加载器:"+ListView.class.getClassLoader());

```

```

package java.lang;

import java.io.IOException;

class BootClassLoader extends ClassLoader
{
 static BootClassLoader instance;

 public static synchronized BootClassLoader getInstance()
 {
 if (instance == null) {
 instance = new BootClassLoader();
 }
 return instance;
 }

 public BootClassLoader() {
 super(null, true);
 }
}

```

## 2) 应用程序的默认加载器

```
Log.i("DEMO", "应用程序默认加载器:"+getClassLoader());
```

运行结果：

```
应用程序默认加载器:dalvik.system.PathClassLoader[DexPathList[[zip file "/data/app/...example.androidddemo-2.apk"],nativeLibraryDirectories=[/data/app-lib/com.example.androidddemo-2, /vendor/lib, /system/lib]]]
```

默认类加载器是PathClassLoader，同时可以看到加载的apk路径，libPath(一般包括/vendor/lib和/system/lib)

### 3) 系统类加载器

```
Log.i("DEMO", "系统类加载器:"+ClassLoader.getSystemClassLoader());
```

运行结果：

```
系统类加载器:dalvik.system.PathClassLoader[DexPathList[[directory "."],nativeLibraryDirectories=[/vendor/lib, /system/lib]]]
```

系统类加载器其实还是PathClassLoader，只是加载的apk路径不是/data/app/xxx.apk了，而是系统apk的路径：/system/app/xxx.apk

### 4) 默认加载器的委派机制关系

```
Log.i("DEMO","打印应用程序默认加载器的委派机制:");
ClassLoader classLoader = getClassLoader();
while(classLoader != null){
 Log.i("DEMO", "类加载器:"+classLoader);
 classLoader = classLoader.getParent();
}
```

打印结果：

打印应用程序默认加载器的委派机制：

```
类加载器:dalvik.system.PathClassLoader[DexPathList[[zip file "/data/app/com.example.androidddemo-2.apk"],nativeLibraryDirectories=[/data/app-lib/com.example.androidddemo-2, /vendor/lib, /system/lib]]]
类加载器:java.lang.BootClassLoader@415e7fe0
```

默认加载器PathClassLoader的父亲是BootClassLoader

## 5) 系统加载器的委派机制关系

```
Log.i("DEMO", "打印系统加载器的委派机制:");
ClassLoader = ClassLoader.getSystemClassLoader();
while(classLoader != null){
 Log.i("DEMO", "类加载器:"+classLoader);
 classLoader = classLoader.getParent();
}
```

运行结果：

```
打印系统加载器的委派机制:
类加载器:dalvik.system.PathClassLoader[DexPathList[[directory "."],nativeI
irectories=[/vendor/lib, /system/lib]]]
类加载器:java.lang.BootClassLoader@415e7fe0
```

可以看到系统加载器的父亲也是BootClassLoader

## 二、分析遇到的问题的原因和解决办法

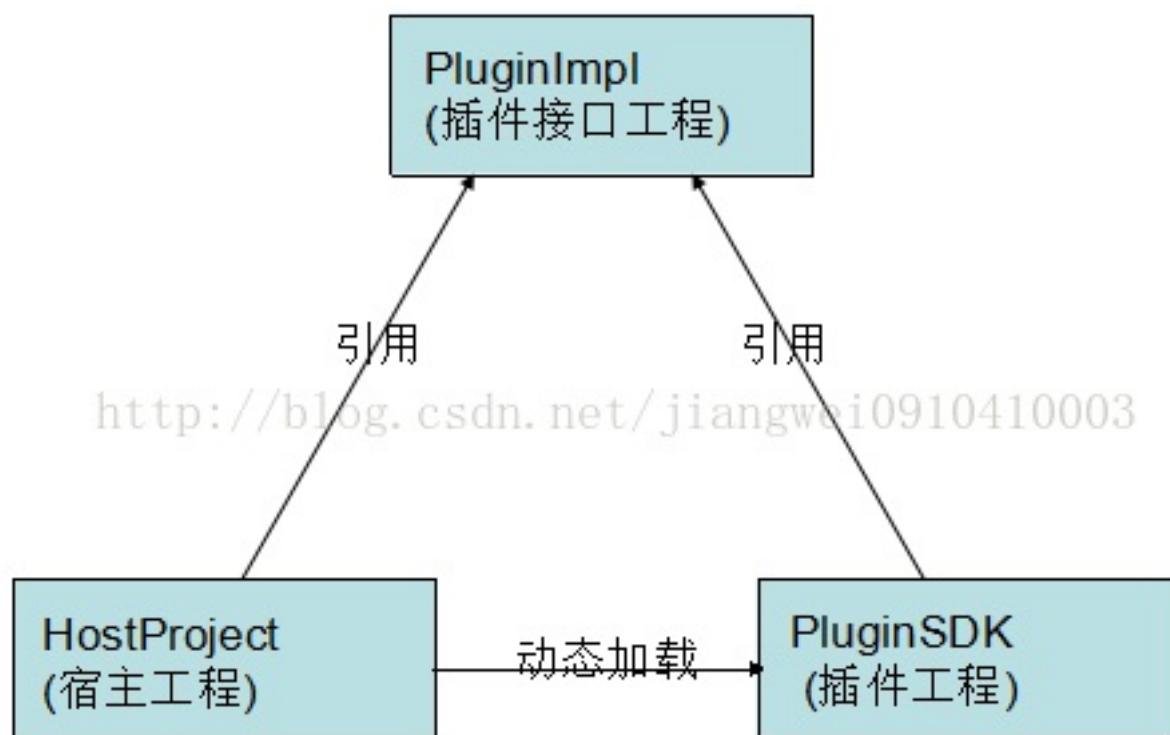
DexClassLoader加载原理和分析在实现插件时不同操作造成错误的原因分析

这里主要用了三个工程：

PluginImpl：插件接口工程(只是接口的定义)

PluginSDK：插件工程(实现插件接口，定义具体的功能)

HostProject：宿主工程(需要引用插件接口工程，然后动态的加载插件工程)(例子项目中名字是PluginDemos)



## 第一、项目介绍

下面来看一下源代码：

1、PluginImpl工程：

1) IBean.java

```
package com.pluginsdk.interfaces;

public abstract interface IBean{
 public abstract String getName();
 public abstract void setName(String paramString);
}
```

2) IDynamic.java

```

package com.pluginsdk.interfaces;

import android.content.Context;

public abstract interface IDynamic{
 public abstract void methodWithCallBack(YKCallBack paramYKCallBack);
 public abstract void showPluginWindow(Context paramContext);
 public abstract void startPluginActivity(Context context,Class<?> cls);
 public abstract String getStringForResId(Context context);
}

```

其他的就不列举了。

## 2、PluginSDK工程：

### 1) Dynamic.java

```

/**
 * Dynamic1.java
 * com.youku.pluginsdk.imp
 *
 * Function: TODO
 *
 * ver date author
 * _____
 * 2014-10-20 Administrator
 *
 * Copyright (c) 2014, TNT All Rights Reserved.
 */

package com.pluginsdk.imp;

import android.app.AlertDialog;
import android.app.AlertDialog.Builder;
import android.app.Dialog;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;

import com.pluginsdk.bean.Bean;
import com.pluginsdk.interfaces.IDynamic;
import com.pluginsdk.interfaces.YKCallBack;
import com.youku.pluginsdk.R;

/**
 * ClassName:Dynamic1
 *
 * @author jiangwei
 * @version

```

```

* @since Ver 1.1
* @Date 2014-10-20 下午5:57:10
*/
public class Dynamic implements IDynamic{
 /**
 */
 public void methodWithCallBack(YKCallBack callback) {
 Bean bean = new Bean();
 bean.setName("PLUGIN_SDK_USER");
 callback.callback(bean);
 }

 public void showPluginWindow(Context context) {
 AlertDialog.Builder builder = new Builder(context);
 builder.setMessage("对话框");
 builder.setTitle(R.string.hello_world);
 builder.setNegativeButton("取消", new Dialog.OnClickListener() {
 @Override
 public void onClick(DialogInterface dialog, int which) {
 dialog.dismiss();
 }
 });
 Dialog dialog = builder.create(); // .show();
 dialog.show();
 }

 public void startPluginActivity(Context context, Class<?> cls){
 /**
 * 这里要注意几点：
 * 1、如果单纯的写一个MainActivity的话，在主工程中也有一个MainActivity，开启的Activity还
 * 2、如果这里将MainActivity写成全名的话，还是有问题，会报找不到这个Activity的错误
 */
 Intent intent = new Intent(context,cls);
 context.startActivity(intent);
 }

 public String getStringForResId(Context context){
 return context.getResources().getString(R.string.hello_world);
 }
}

```

## 2) Bean.java

```

/**
 * User.java
 * com.youku.pluginsdk.bean
 *
 * Function: TODO
 *
 * ver date author
 * _____
 * 2014-10-20 Administrator
 *
 * Copyright (c) 2014, TNT All Rights Reserved.
 */

package com.pluginsdk.bean;

/**
 * ClassName:User
 *
 * @author jiangwei
 * @version
 * @since Ver 1.1
 * @Date 2014-10-20 下午1:35:16
 */
public class Bean implements com.pluginsdk.interfaces.IBean{

 /**
 *
 */
 private String name = "这是来自于插件工程中设置的初始化的名字";

 public String getName() {
 return name;
 }

 public void setName(String name) {
 this.name = name;
 }

}

```

### 3、宿主工程HostProject

#### 1) MainActivity.java

```

package com.plugindemo;
import java.io.File;
import java.lang.reflect.Method;

```

```
import android.annotation.SuppressLint;
import android.app.Activity;
import android.content.Context;
import android.content.res.AssetManager;
import android.content.res.Resources;
import android.content.res.Resources.Theme;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.ListView;
import android.widget.Toast;

import com.pluginsdk.interfaces.IBean;
import com.pluginsdk.interfaces.IDynamic;
import com.pluginsdk.interfaces.YKCallBack;
import com.youku.plugindemo.R;

import dalvik.system.DexClassLoader;

public class MainActivity extends Activity {
 private AssetManager mAssetManager;//资源管理器
 private Resources mResources;//资源
 private Theme mTheme;//主题
 private String apkFileName = "PluginSDKs.apk";
 private String dexpath = null;//apk文件地址
 private File fileRelease = null; //释放目录
 private DexClassLoader classLoader = null;
 @SuppressLint("NewApi")
 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 dexpath = Environment.getExternalStorageDirectory() + File.separator+apkFile
 fileRelease = getDir("dex", 0);

 /*初始化classloader
 * dexpath dex文件地址
 * fileRelease 文件释放地址
 * 父classLoader
 */
 Log.d("DEMO", (getClassLoader() == ListView.class.getClassLoader())+"");
 Log.d("DEMO", ListView.class.getClassLoader()+"");
 Log.d("DEMO", Context.class.getClassLoader()+"");
 Log.d("DEMO", Context.class.getClassLoader().getSystemClassLoader()+"");
 Log.d("DEMO", Activity.class.getClassLoader()+"");
 Log.d("DEMO", (Context.class.getClassLoader().getSystemClassLoader() == Class
 Log.d("DEMO", ClassLoader.getSystemClassLoader()+""));

 classLoader = new DexClassLoader(dexpath, fileRelease.getAbsolutePath(),null,
```

```

 Button btn_1 = (Button) findViewById(R.id.btn_1);
 Button btn_2 = (Button) findViewById(R.id.btn_2);
 Button btn_3 = (Button) findViewById(R.id.btn_3);
 Button btn_4 = (Button) findViewById(R.id.btn_4);
 Button btn_5 = (Button) findViewById(R.id.btn_5);
 Button btn_6 = (Button) findViewById(R.id.btn_6);

 btn_1.setOnClickListener(new View.OnClickListener() {//普通调用 反射的方式
 @Override
 public void onClick(View arg0) {
 Class mLoadClassBean;
 try {
 mLoadClassBean = classLoader.loadClass("com.pluginsdk.bean.Bean");
 Object beanObject = mLoadClassBean.newInstance();
 Log.d("DEMO", "ClassLoader:"+mLoadClassBean.getClassLoader());
 Log.d("DEMO", "ClassLoader:"+mLoadClassBean.getClassLoader().getPa
 Method getNameMethod = mLoadClassBean.getMethod("getName");
 getNameMethod.setAccessible(true);
 String name = (String) getNameMethod.invoke(beanObject);
 Toast.makeText(MainActivity.this, name, Toast.LENGTH_SHORT).show();
 } catch (Exception e) {
 Log.e("DEMO", "msg:"+e.getMessage());
 }
 }
 });
 btn_2.setOnClickListener(new View.OnClickListener() {//带参数调用
 @Override
 public void onClick(View arg0) {
 Class mLoadClassBean;
 try {
 mLoadClassBean = classLoader.loadClass("com.pluginsdk.bean.Bean");
 Object beanObject = mLoadClassBean.newInstance();
 //接口形式调用
 Log.d("DEMO", beanObject.getClass().getClassLoader()+"");
 Log.d("DEMO", IBean.class.getClassLoader()+"");
 Log.d("DEMO", ClassLoader.getSystemClassLoader()+"");
 IBean bean = (IBean)beanObject;
 bean.setName("宿主程序设置的新名字");
 Toast.makeText(MainActivity.this, bean.getName(), Toast.LENGTH_SH
 }catch (Exception e) {
 Log.e("DEMO", "msg:"+e.getMessage());
 }
 }
 });
 btn_3.setOnClickListener(new View.OnClickListener() {//带回调函数的调用
 @Override
 public void onClick(View arg0) {
 Class mLoadClassDynamic;
 try {
 mLoadClassDynamic = classLoader.loadClass("com.pluginsdk.imp.Dyna

```

```

 Object dynamicObject = mLoadClassDynamic.newInstance();
 //接口形式调用
 IDynamic dynamic = (IDynamic)dynamicObject;
 //回调函数调用
 YKCallBack callback = new YKCallBack() {//回调接口的定义
 public void callback(IBean arg0) {
 Toast.makeText(MainActivity.this, arg0.getName(), Toast.LENGTH_SHORT).show();
 }
 };
 dynamic.methodWithCallBack(callback);
 } catch (Exception e) {
 Log.e("DEMO", "msg:"+e.getMessage());
 }

}

});

btn_4.setOnClickListener(new View.OnClickListener() {//带资源文件的调用
 @Override
 public void onClick(View arg0) {
 loadResources();
 Class mLoadClassDynamic;
 try {
 mLoadClassDynamic = classLoader.loadClass("com.pluginsdk.imp.Dynamic");
 Object dynamicObject = mLoadClassDynamic.newInstance();
 //接口形式调用
 IDynamic dynamic = (IDynamic)dynamicObject;
 dynamic.showPluginWindow(MainActivity.this);
 } catch (Exception e) {
 Log.e("DEMO", "msg:"+e.getMessage());
 }
 }
});

btn_5.setOnClickListener(new View.OnClickListener() {//带资源文件的调用
 @Override
 public void onClick(View arg0) {
 loadResources();
 Class mLoadClassDynamic;
 try {
 mLoadClassDynamic = classLoader.loadClass("com.pluginsdk.imp.Dynamic");
 Object dynamicObject = mLoadClassDynamic.newInstance();
 //接口形式调用
 IDynamic dynamic = (IDynamic)dynamicObject;
 dynamic.startPluginActivity(MainActivity.this,
 classLoader.loadClass("com.plugindemo.MainActivity"));
 } catch (Exception e) {
 Log.e("DEMO", "msg:"+e.getMessage());
 }
 }
});

btn_6.setOnClickListener(new View.OnClickListener() {//带资源文件的调用
 @Override
 public void onClick(View arg0) {

```

```
 loadResources();
 Class mLoadClassDynamic;
 try {
 mLoadClassDynamic = classLoader.loadClass("com.pluginsdk.imp.Dynamic");
 Object dynamicObject = mLoadClassDynamic.newInstance();
 //接口形式调用
 IDynamic dynamic = (IDynamic)dynamicObject;
 String content = dynamic.getStringForResId(MainActivity.this);
 Toast.makeText(getApplicationContext(), content+"", Toast.LENGTH_SHORT);
 } catch (Exception e) {
 Log.e("DEMO", "msg:"+e.getMessage());
 }
 }
});

protected void loadResources() {
 try {
 AssetManager assetManager = AssetManager.class.newInstance();
 Method addAssetPath = assetManager.getClass().getMethod("addAssetPath");
 addAssetPath.invoke(assetManager, dexpath);
 mAssetManager = assetManager;
 } catch (Exception e) {
 e.printStackTrace();
 }
 Resources superRes = super.getResources();
 superRes.getDisplayMetrics();
 superRes.getConfiguration();
 mResources = new Resources(mAssetManager, superRes.getDisplayMetrics(), superRes.getConfiguration());
 mTheme = mResources.newTheme();
 mTheme.setTo(super.getTheme());
}
}

@Override
public AssetManager getAssets() {
 return mAssetManager == null ? super.getAssets() : mAssetManager;
}

@Override
public Resources getResources() {
 return mResources == null ? super.getResources() : mResources;
}

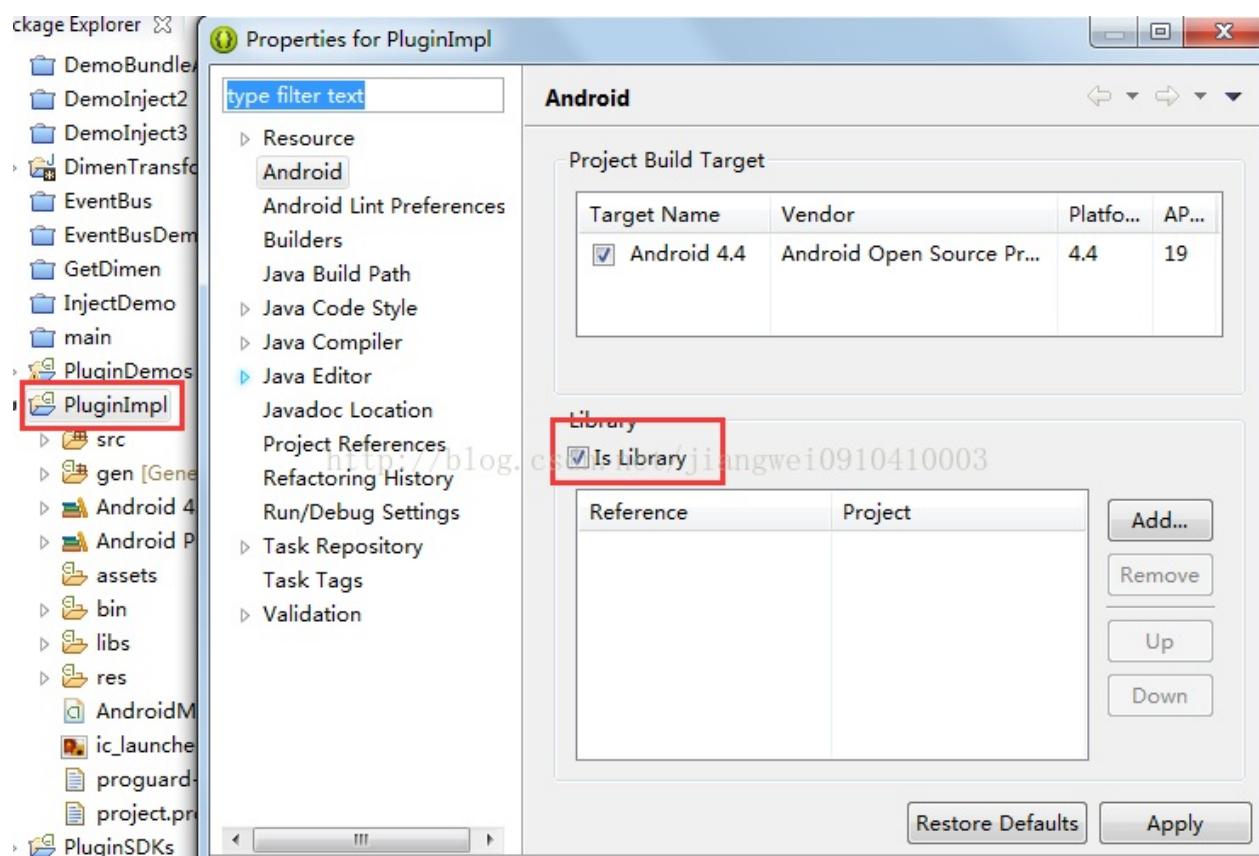
@Override
public Theme getTheme() {
 return mTheme == null ? super.getTheme() : mTheme;
}
}
```

三个工程的下载地址：<http://download.csdn.net/detail/jiangwei0910410003/8188011>

## 第二、项目引用关系

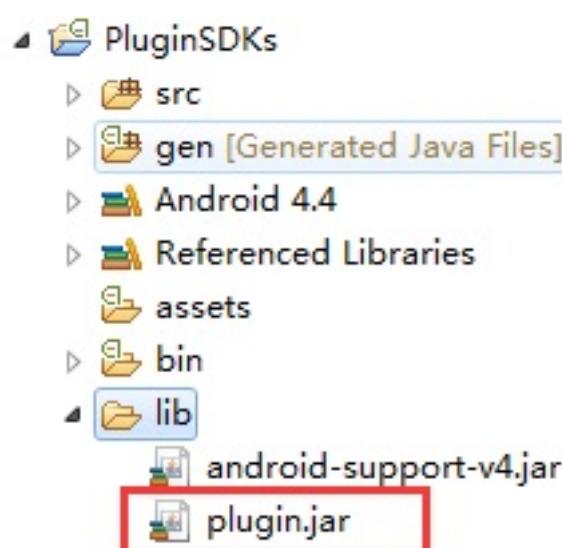
工程文件现在大致看完了，我们看一下他们的引用关系吧：

### 1、将接口工程PluginImpl设置成一个Library

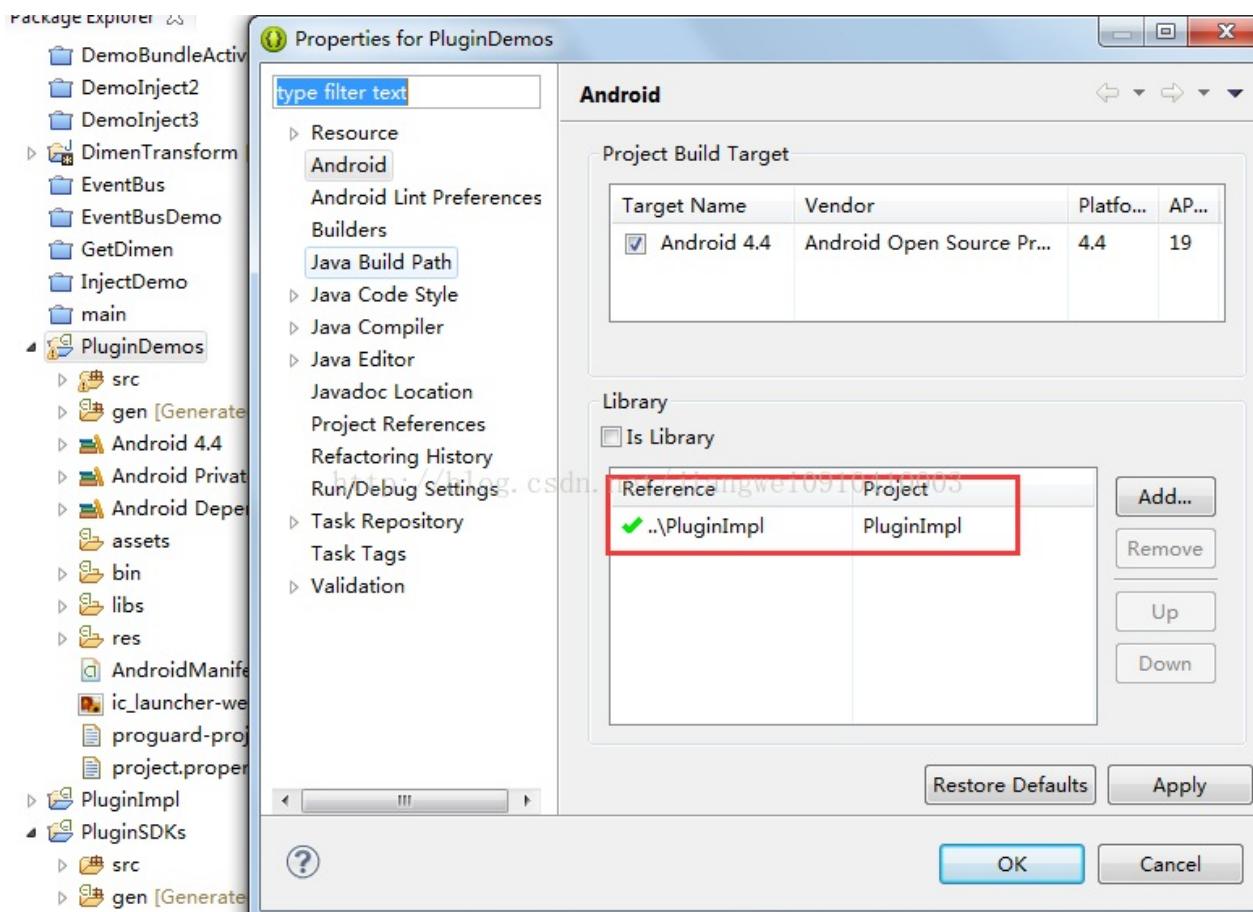


### 2、插件工程PluginSDKs引用插件的jar

注意是lib文件夹，不是libs，这个是有区别的，后面会说道



### 3、HostProject项目引用PluginImpl这个library



项目引用完成之后，我们编译PluginSDKs项目，生成PluginSDKs.apk放到手机的sdcard的根目录(因为我代码中是从这个目录进行加载apk的，当然这个目录是可以修改的)，然后运行HostProject



看到效果了吧。运行成功，其实这个对话框是在插件中定义的，但是我们知道定义对话框是需要context变量的，所以这个变量就是通过参数从宿主工程中传递到插件工程即可，成功了就不能这么了事，因为我还没有说道我遇到的问题，下面就来看一下遇到的几个问题

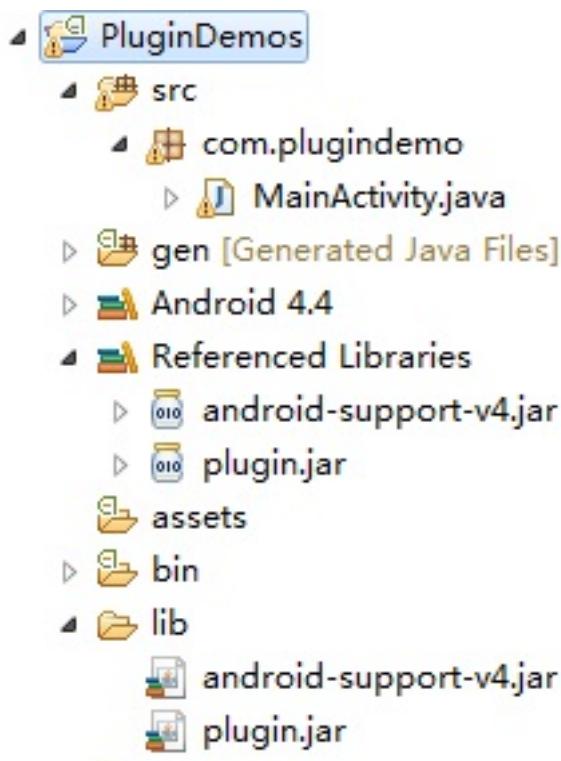
### 三、问题分析

问题一：Could not find class...(找不到指定的类)

```
com.youku.plugindemo dalvikvm Could not find class 'com.pluginsdk.interfaces.IBean', referenced from method http://com.youku.plugindemo.MainActivity\$2.onClick@10410003
```

这个问题产生的操作：

插件工程PluginSDKs的引用方式不变，宿主工程PluginDemos的引用方式改变



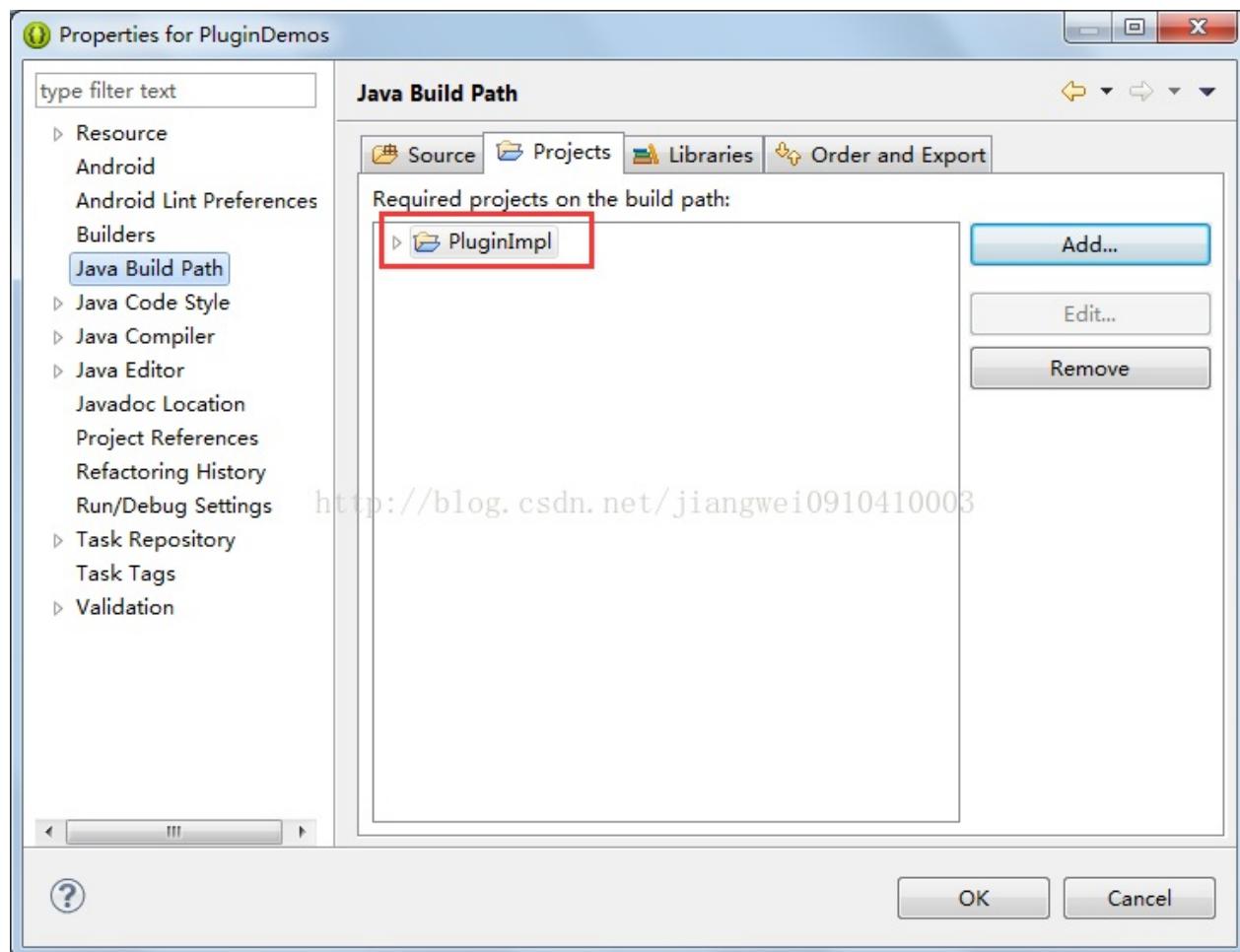
在说这个原因之前先来了解一下Eclipse中引用工程的不同方式和区别：

**第一种：**最常用的将引用工程打成jar放到需要引用工程的libs下面(这里是将PluginImpl打成jar,放到HostProject工程的libs中)

这种方式是Eclipse推荐使用的，当我们在建立一个项目的时候也会自动产生这个文件夹，当我们把我们需要引用的工程打成jar，然后放到这个文件夹之后，Eclipse就自动导入了(这个功能是Eclipse3.7之后有的)。

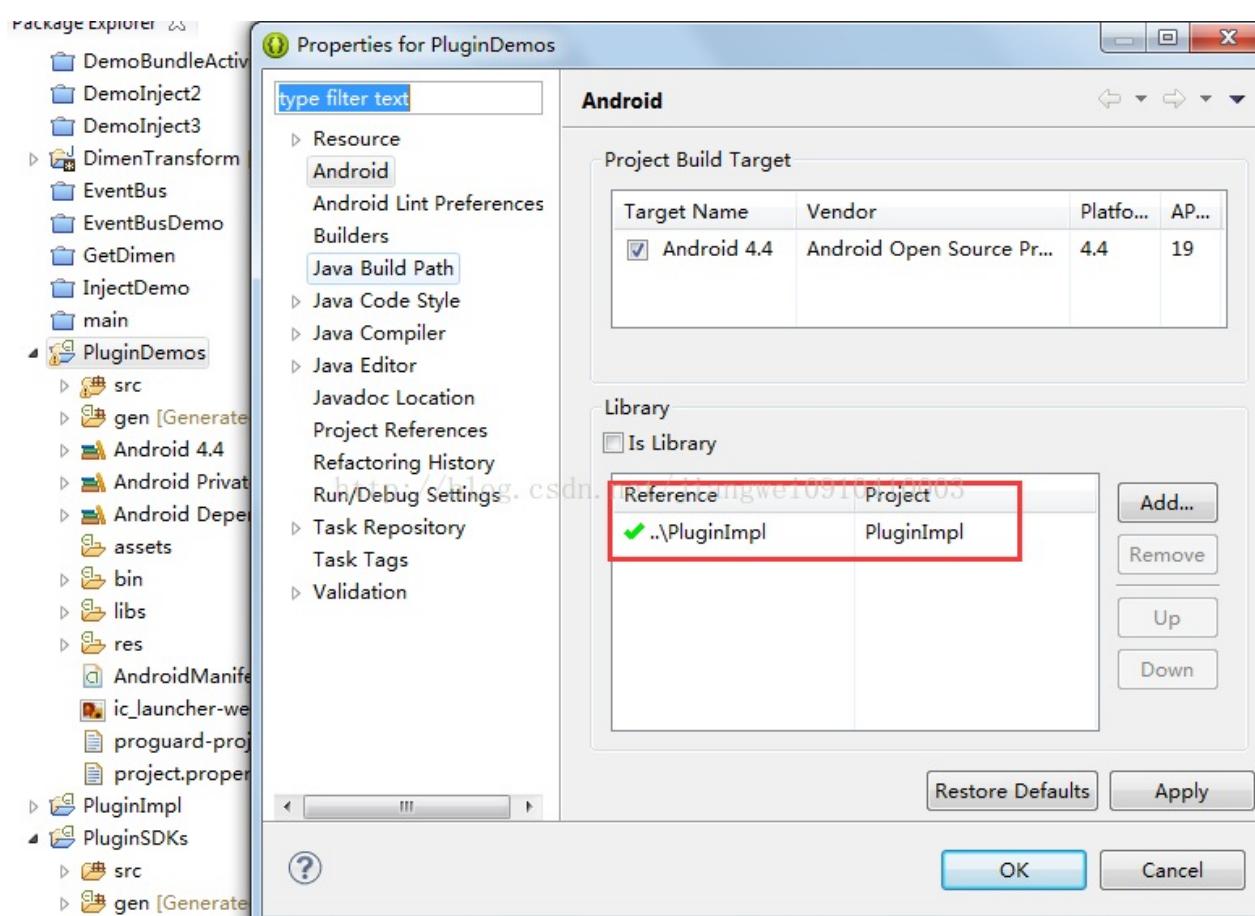
**第二种：**和第一种的区别是，我们可以从新新建一个文件夹比如是lib,然后将引用的jar放到这个文件夹中，但是此时Eclipse是不会自动导入的，需要我们手动的导入(add build path...),但是这个是一个区别，还有一个区别，也是到这个这个报错原因的区别，就是libs文件夹中的jar，在运行的时候是会将这个jar集成到程序中的，而我们新建的文件夹(名字非libs即可)，即使我们手动的导入，编译是没有问题的，但是运行的时候，是不会将jar集成到程序中。

**第三种：**和前两种的区别是不需要将引用工程打成jar，直接引用这个工程



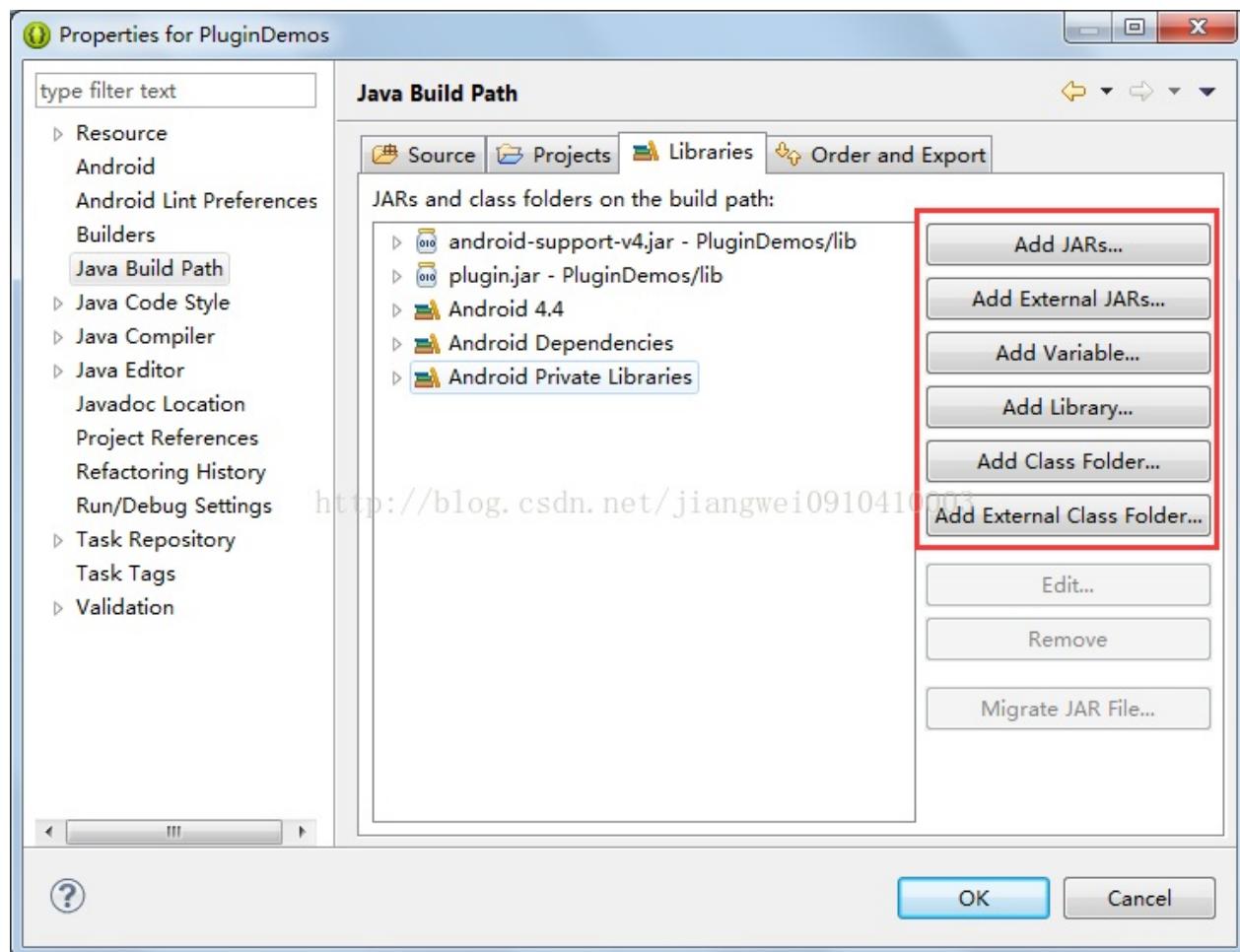
这种方式其实效果和第一种差不多，唯一的区别就是不需要打成jar,但是运行的时候是不会将引用工程集成到程序中的。

**第四种：**和第三种的方式是一样的，也是不需要将引用工程打成jar,直接引用工程：



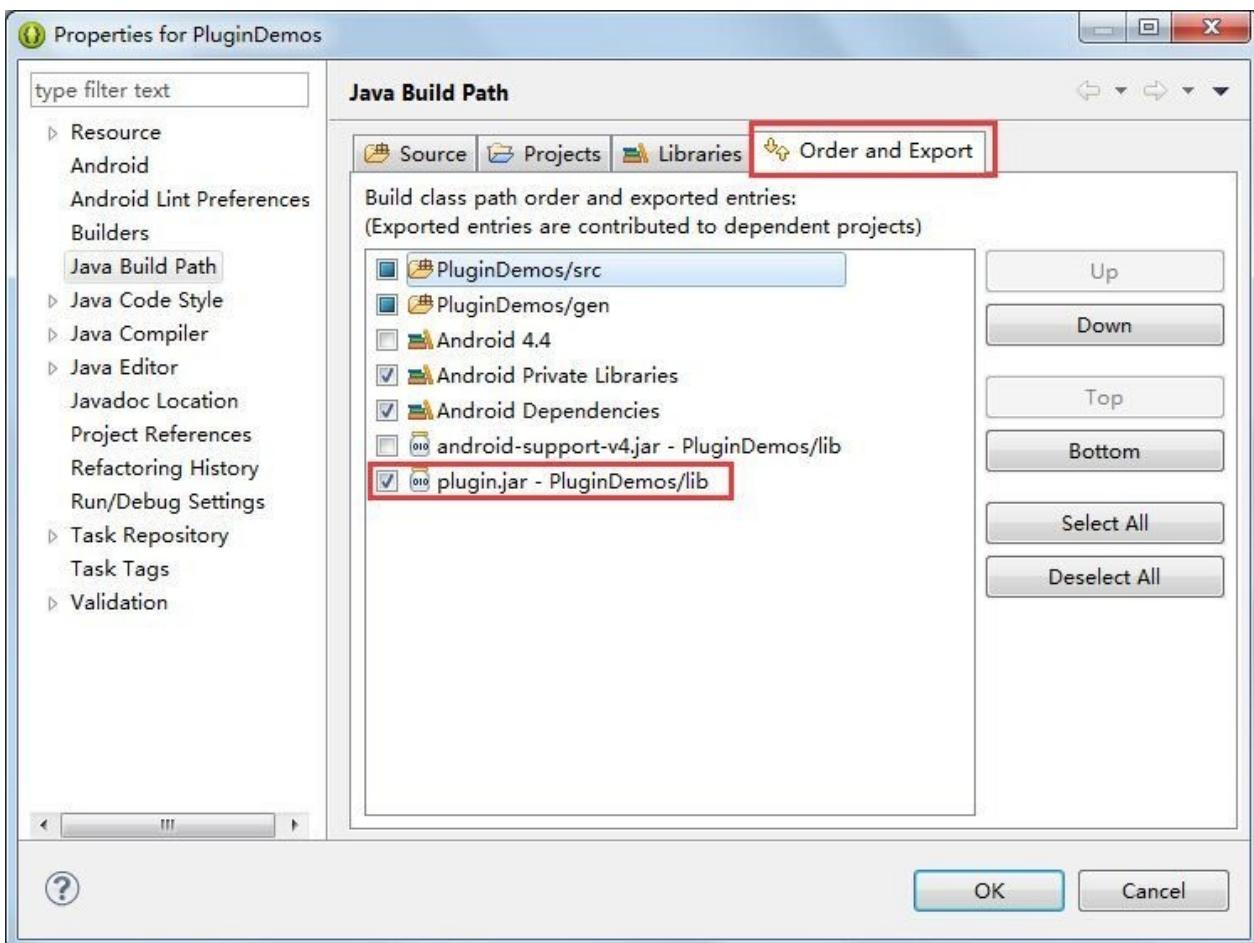
这个前提是需要设置PluginImpl项目为Library,同时引用的项目和被引用的项目必须在一个工作空间中，不然会报错，这种的效果和第二种是一样的，在运行的时候是会将引用工程集成到程序中的。

**第五种：**和第一种、第二种差不多，导入jar:



这里有很多种方式选择jar的位置，但是这些操作的效果和第一种是一样的，运行的时候是不会将引用的jar集成到程序中的。

总结上面的五种方式，我们可以看到，第二种和第四种的效果是一样的，也是最普遍的导入引用工程的方式，因为其他三种方式的话，其实在编译的时候是不会有问题是的，但是在运行的时候会报错(找不到指定的类，可以依次尝试一下)，**不过这三种方式只要一步就可以和那两种方式实现的效果一样了**



只要设置导出的时候勾选上这个jar就可以了。那么其实这五种方式都是可以的，性质和效果是一样的。

说完了Eclipse中引用工程的各种方式以及区别之后，我们在回过头来看一下，上面遇到的问题：Could not find class...

其实这个问题就简单了，原因是：插件工程PluginSDKs使用的是lib文件夹导入的jar(这个jar是不会集成到程序中的)，而宿主工程PluginDemos的引用工程的方式也变成了lib文件夹(jar也是不会集成到程序中的)。那么程序运行的时候就会出现错误：

Could not find class 'com.pluginsdk.interfaces.IBean'

**问题二：Class ref in pre-verified class resolved to unexpected implementation(相同的类加载了两次)**

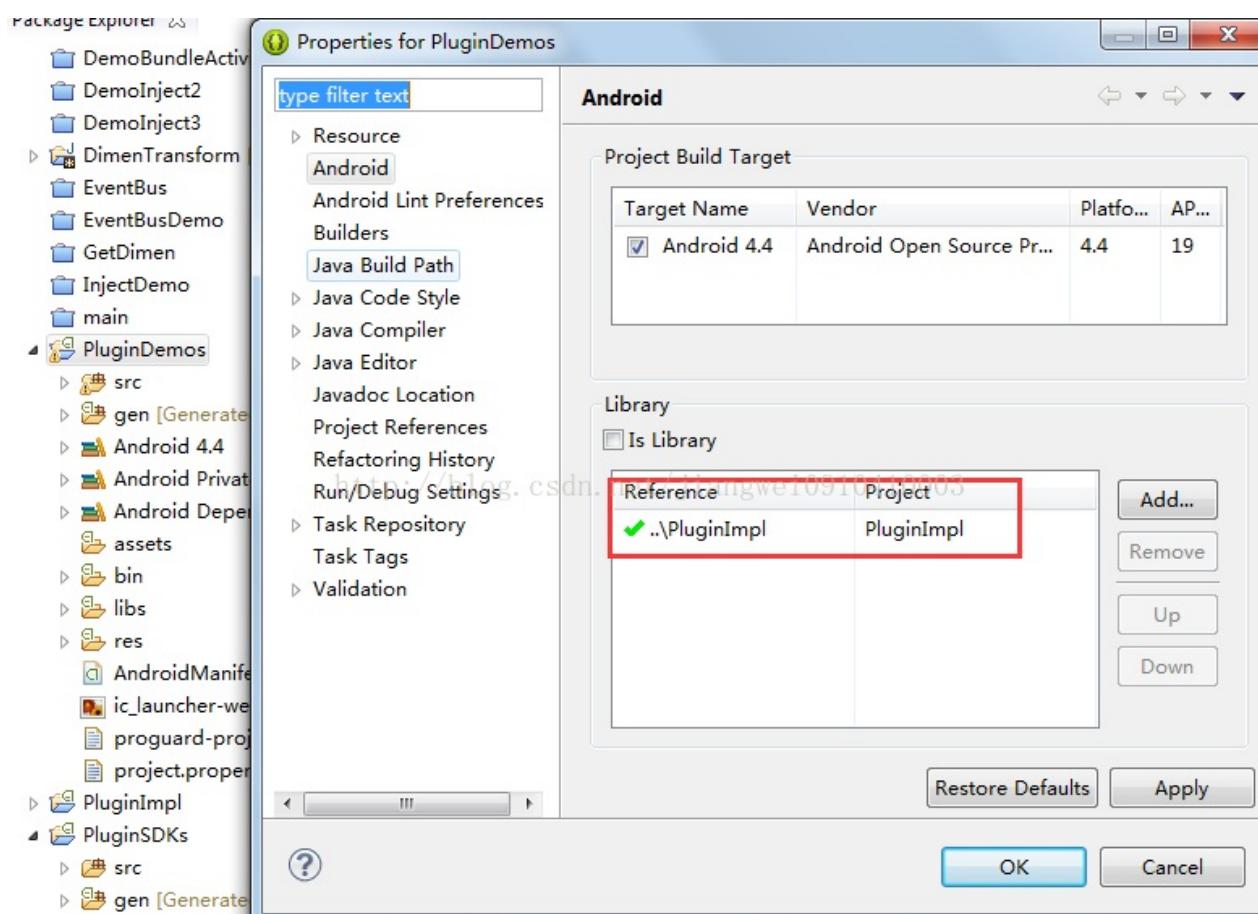
```

6557 6557 com.youku.plugindemo AndroidRuntime FATAL EXCEPTION: main
6557 6557 com.youku.plugindemo AndroidRuntime Process: com.youku.plugindemo, PID: 6557
6557 6557 com.youku.plugindemo AndroidRuntime java.lang.IllegalAccessError: Class ref in pre-verified class resolved to unexpected implementation
6557 6557 com.youku.plugindemo AndroidRuntime at dalvik.system.DexFile.defineClassNative(Native Method)
6557 6557 com.youku.plugindemo AndroidRuntime at dalvik.system.DexFile.defineClass(DexFile.java:222)
6557 6557 com.youku.plugindemo AndroidRuntime at dalvik.system.DexFile.loadClassBinaryName(DexFile.java:215)
6557 6557 com.youku.plugindemo AndroidRuntime at dalvik.system.DexPathList.findClass(DexPathList.java:322)
6557 6557 com.youku.plugindemo AndroidRuntime at dalvik.system.BaseDexClassLoader.findClass(BaseDexClassLoader.java:54)
6557 6557 com.youku.plugindemo AndroidRuntime at java.lang.ClassLoader.loadClass(ClassLoader.java:497)
6557 6557 com.youku.plugindemo AndroidRuntime at java.lang.ClassLoader.loadClass(ClassLoader.java:457)

```

这个问题产生的操作：

插件工程PluginSDKs和宿主工程PluginDemos引用工程的方式都变成library(或者是都用libs文件夹导入jar)



这个错误的原因也是很多做插件的开发者第一次都会遇到的问题，其实这个问题的本质是 **PluginImpl** 中的接口被加载了两次，因为插件工程和宿主工程在运行的时候都会把 **PluginImpl** 集成到程序中。对于这个问题，我们来分析一下，首先对于宿主apk，他的类加载器是 **PathClassLoader**（这个对于每个应用来说是默认的加载器，原因很简单，**PathClassLoader** 只能加载 **/data/app** 目录下的 apk，就是已经安装的 apk，一般我们的 apk 都是安装之后在运行，所以用这个加载器也是理所当然的）。这个加载器开始加载插件接口工程（宿主工程中引入的 **PluginImpl**）中的 **IBean**。当使用 **DexClassLoader** 加载 **PluginSDKs.apk** 的时候，首先会让宿主 apk 的 **PathClassLoader** 加载器去加载，这个好多人有点迷糊了，为什么会先让 **PathClassLoader** 加载器去加载呢？

这个就是 Java 中的类加载机制的双亲委派机

制：<http://blog.csdn.net/jiangwei0910410003/article/details/17733153>

Android 中的加载机制也是类似的，我们这里的代码设置了 **DexClassLoader** 的父加载器为当前类加载器（宿主 apk 的 **PathClassLoader**），不行的话，可以打印一下 **getClassLoader()** 方法的返回结果看一下。

```
classLoader = new DexClassLoader(dexpath, fileRelease.getAbsolutePath(), null, getClass
```

那么加载器就是一样的了(宿主apk的PathClassLoader), 那么就奇怪了, 都是一个为什么还有错误呢? 查看系统源码可以了解: Resolve.c源码(这个是在虚拟机dalvik中的): 源码下载地址为: <http://blog.csdn.net/jiangwei0910410003/article/details/37988637>

我们来看一下他的一个主要函数:

```
/*
 * Find the class corresponding to "classIdx", which maps to a class name
 * string. It might be in the same DEX file as "referrer", in a different
 * DEX file, generated by a class loader, or generated by the VM (e.g.
 * array classes).
 *
 * Because the DexTypeId is associated with the referring class' DEX file,
 * we may have to resolve the same class more than once if it's referred
 * to from classes in multiple DEX files. This is a necessary property for
 * DEX files associated with different class loaders.
 *
 * We cache a copy of the lookup in the DexFile's "resolved class" table,
 * so future references to "classIdx" are faster.
 *
 * Note that "referrer" may be in the process of being linked.
 *
 * Traditional VMs might do access checks here, but in Dalvik the class
 * "constant pool" is shared between all classes in the DEX file. We rely
 * on the verifier to do the checks for us.
 *
 * Does not initialize the class.
 *
 * "fromUnverifiedConstant" should only be set if this call is the direct
 * result of executing a "const-class" or "instance-of" instruction, which
 * use class constants not resolved by the bytecode verifier.
 *
 * Returns NULL with an exception raised on failure.
 */
ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx,
 bool fromUnverifiedConstant)
{
 DvmDex* pDvmDex = referrer->pDvmDex;
 ClassObject* resClass;
 const char* className;

 /*
 * Check the table first -- this gets called from the other "resolve"
 * methods.
 */
```

```

resClass = dvmDexGetResolvedClass(pDvmDex, classIdx);
if (resClass != NULL)
 return resClass;

LOGVV("--- resolving class %u (referrer=%s cl=%p)\n",
 classIdx, referrer->descriptor, referrer->classLoader);

/*
 * Class hasn't been loaded yet, or is in the process of being loaded
 * and initialized now. Try to get a copy. If we find one, put the
 * pointer in the DexTypeId. There isn't a race condition here --
 * 32-bit writes are guaranteed atomic on all target platforms. Worst
 * case we have two threads storing the same value.
 *
 * If this is an array class, we'll generate it here.
 */
className = dexStringByTypeIdx(pDvmDex->pDexFile, classIdx);
if (className[0] != '\0' && className[1] == '\0') {
 /* primitive type */
 resClass = dvmFindPrimitiveClass(className[0]);
} else {
 resClass = dvmFindClassNoInit(className, referrer->classLoader);
}

if (resClass != NULL) {
 /*
 * If the referrer was pre-verified, the resolved class must come
 * from the same DEX or from a bootstrap class. The pre-verifier
 * makes assumptions that could be invalidated by a wacky class
 * loader. (See the notes at the top of oo/Class.c.)
 *
 * The verifier does *not* fail a class for using a const-class
 * or instance-of instruction referring to an unresolvable class,
 * because the result of the instruction is simply a Class object
 * or boolean -- there's no need to resolve the class object during
 * verification. Instance field and virtual method accesses can
 * break dangerously if we get the wrong class, but const-class and
 * instance-of are only interesting at execution time. So, if we
 * we got here as part of executing one of the "unverified class"
 * instructions, we skip the additional check.
 *
 * Ditto for class references from annotations and exception
 * handler lists.
 */
 if (!fromUnverifiedConstant &&
 IS_CLASS_FLAG_SET(referrer, CLASS_ISPREVERIFIED))
 {
 ClassObject* resClassCheck = resClass;
 if (dvmIsArrayClass(resClassCheck))
 resClassCheck = resClassCheck->elementClass;

 if (referrer->pDvmDex != resClassCheck->pDvmDex &&

```

```

 resClassCheck->classLoader != NULL)
{
 LOGW("Class resolved by unexpected DEX:
 "%s(%p):%p ref [%s] %s(%p):%p\n",
 referrer->descriptor, referrer->classLoader,
 referrer->pDvmDex,
 resClass->descriptor, resClassCheck->descriptor,
 resClassCheck->classLoader, resClassCheck->pDvmDex);
 LOGW("(%s had used a different %s during pre-verification)\n",
 referrer->descriptor, resClass->descriptor);
 dvmThrowException("Ljava/lang/IllegalAccessError;",
 "Class ref in pre-verified class resolved to unexpected "
 "implementation");
 return NULL;
}
}

LOGVV("##### +ResolveClass(%s): referrer=%s dex=%p ldr=%p ref=%d\n",
 resClass->descriptor, referrer->descriptor, referrer->pDvmDex,
 referrer->classLoader, classIdx);

/*
 * Add what we found to the list so we can skip the class search
 * next time through.
 *
 * TODO: should we be doing this when fromUnverifiedConstant==true?
 * (see comments at top of oo/Class.c)
 */
dvmDexSetResolvedClass(pDvmDex, classIdx, resClass);
} else {
 /* not found, exception should be raised */
 LOGVV("Class not found: %s\n",
 dexStringByTypeIdx(pDvmDex->pDexFile, classIdx));
 assert(dvmCheckException(dvmThreadSelf()));
}

return resClass;
}

```

我们看下面的判断可以得到，就是在这里抛出的异常，代码逻辑我们就不看了，因为太多的头文件相互引用，看起来很费劲，直接看一下函数的说明：

```

/*
 * Find the class corresponding to "classIdx", which maps to a class name
 * string. It might be in the same DEX file as "referrer", in a different
 * DEX file, generated by a class loader, or generated by the VM (e.g.
 * array classes).
 *
 * Because the DexTypeId is associated with the referring class' DEX file,
 * we may have to resolve the same class more than once if it's referred
 * to from classes in multiple DEX files. This is a necessary property for
 * DEX files associated with different class loaders.
 *
 * We cache a copy of the lookup in the DexFile's "resolved class" table,
 * so future references to "classIdx" are faster.
 *
 * Note that "referrer" may be in the process of being linked.
 *
 * Traditional VMs might do access checks here, but in Dalvik the class
 * "constant pool" is shared between all classes in the DEX file. We rely
 * on the verifier to do the checks for us.
 *
 * Does not initialize the class.
 *
 * "fromUnverifiedConstant" should only be set if this call is the direct
 */

```

红色部分内容，他的意思是我们需要解决从不同的dex文件中加载相同的class,需要使用不同的类加载器。

说白了就是，同一个类加载器从不同的dex文件中加载相同的class。所以上面是同一个类加载器PathClassLoader去加载(宿主apk和插件apk)来自不同的dex中的相同的类IBean。所以我们在做动态加载的时候都说过：不要把接口的jar一起打包成jar/dex/apk

### 问题三：Connot be cast to....(类型转化异常)

7559 7559 com.youku.plugindemo DEMO msg:com.pluginsdk.bean.Bean cannot be cast to com.pluginsdk.interfaces.IBean

这个问题产生的操作：

插件工程PluginSDKs和宿主工程都是用Library方式引用工程(或者是libs)，同时将上面的一行代码

```
classLoader = new DexClassLoader(dexpath, fileRelease.getAbsolutePath(), null, getClassLoader());
```

修改成：

```
classLoader = new DexClassLoader(dexpath, fileRelease.getAbsolutePath(), null, getClassLoader());
```

就是将DexClassLoader的父加载器修改了一下：我们知道getClassLoader()获取到的是应用的默认加载器PathClassLoader，而ClassLoader.getSystemClassLoader()是获取系统类加载器，这样修改之后会出现这样的错误的原因是：插件工程和宿主工程都集成了PluginImpl，所以DexClassLoader在加载Bean的时候，首先会让ClassLoader.getSystemClassLoader()类加载器(DexClassLoader的父加载器)去查找，因为Bean是实现了IBean接口，这时候ClassLoader.getSystemClassLoader就会从插件工程的apk中查找这个接口，结果没找到，没找到的话就让DexClassLoader去找，结果在PluginSDKs.apk中找到了，就加载进来，同时宿主工程中也集成了插件接口PluginImpl，他使用PathClassLoader去宿主工程中去查找，结果也是查找到了，也加载进来了，但是在进行类型转化的时候出现了错误：

```
IBean bean = (IBean)beanObject;
```

原因说白了就是：同一个类，用不同的类加载器进行加载产生出来的对象是不同的，不能进行相互赋值，负责就会出现转化异常。

## 总结

上面就说到了一些开发插件的过程中会遇到的一些问题，当我们知道这些问题之后，解决方法自然就会有了，

1) 为了避免Could not find class...，我们必须集成PluginImpl，方式是使用Library或者是libs文件夹导入jar(这里要注意，因为我们运行的其实是宿主工程apk，所以宿主工程一定要集成PluginImpl，如果他不集成的话，即使插件工程apk集成了也还是没有效果的)

2) 为了避免Class ref in pre-verified class resolved to unexpected implementation，我们在宿主工程和插件工程中只能集成一份PluginImpl，在结合上面的错误避免方式，可以得到正确的方式：一定是宿主工程集成PluginImpl，插件工程一定不能集成PluginImpl。

(以后再制作插件的时候记住一句话就可以了，插件工程打包不能集成接口jar，宿主工程打包一定要集成接口jar)

关于第三个问题，其实在开发的过程中一般不会碰到，这里说一下主要是为了马上介绍Android中的类加载器的相关只是来做铺垫的

(PS:问题都解决了，后续就要介绍插件的制作了~~)

# Android中的动态加载机制

来源:[尼古拉斯](#)

在目前的软硬件环境下，Native App与Web App在用户体验上有着明显的优势，但在实际项目中有些会因为业务的频繁变更而频繁的升级客户端，造成较差的用户体验，而这也恰恰是Web App的优势。本文对网上Android动态加载jar的资料进行梳理和实践在这里与大家一起分享，试图改善频繁升级这一弊病。

Android应用开发在一般情况下，常规的开发方式和代码架构就能满足我们的普通需求。但是有些特殊问题，常常引发我们进一步的沉思。我们从沉思中产生顿悟，从而产生新的技术形式。

如何开发一个可以自定义控件的Android应用？就像eclipse一样，可以动态加载插件；如何让Android应用执行服务器上的不可预知的代码？如何对Android应用加密，而只在执行时自解密，从而防止被破解？……

熟悉Java技术的朋友，可能意识到，我们需要使用类加载器灵活的加载执行的类。这在Java里已经算是一项比较成熟的技术了，但是在Android中，我们大多数人都还非常陌生。

## 类加载机制

Dalvik虚拟机如同其他Java虚拟机一样，在运行程序时首先需要将对应的类加载到内存中。而在Java标准的虚拟机中，类加载可以从class文件中读取，也可以是其他形式的二进制流，因此，我们常常利用这一点，在程序运行时手动加载Class，从而达到代码动态加载执行的目的

然而Dalvik虚拟机毕竟不算是标准的Java虚拟机，因此在类加载机制上，它们有相同的地方，也有不同之处。我们必须区别对待

例如，在使用标准Java虚拟机时，我们经常自定义继承自ClassLoader的类加载器。然后通过defineClass方法来从一个二进制流中加载Class。然而，这在Android里是行不通的，大家就没必要走弯路了。参看源码我们知道，Android中ClassLoader的defineClass方法具体是调用VMClassLoader的defineClass本地静态方法。而这个本地方法除了抛出一个“UnsupportedOperationException”之外，什么都没做，甚至连返回值都为空

```

static void Dalvik_java_lang_VMClassLoader_defineClass(const u4* args, JValue* pResult
Object* loader = (Object*) args[0];
StringObject* nameObj = (StringObject*) args[1];
const u1* data = (const u1*) args[2];
int offset = args[3];
int len = args[4];
Object* pd = (Object*) args[5];
char* name = NULL;
name = dvmCreateCstrFromString(nameObj);
LOGE("ERROR: defineClass(%p, %s, %p, %d, %d, %p)\n", loader, name, data, offset, len, pd);
dvmThrowException("Ljava/lang/UnsupportedOperationException;", "can't load this type");
free(name);
RETURN_VOID();
}

```

## Dalvik虚拟机类加载机制

那如果在Dalvik虚拟机里，ClassLoader不好使，我们如何实现动态加载类呢？Android为我们从ClassLoader派生出了两个类：DexClassLoader和PathClassLoader。其中需要特别说明的是PathClassLoader中一段被注释掉的代码：

```

/* --this doesn't work in current version of Dalvik--
if (data != null) {
 System.out.println("--- Found class " + name
 + " in zip[" + i + "] '" + mZips[i].getName() + "'");
 int dotIndex = name.lastIndexOf('.');
 if (dotIndex != -1) {
 String packageName = name.substring(0, dotIndex);
 synchronized (this) {
 Package packageObj = getPackage(packageName);
 if (packageObj == null) {
 definePackage(packageName, null, null,
 null, null, null, null, null);
 }
 }
 }
 return defineClass(name, data, 0, data.length);
}
*/

```

这从另一方面佐证了defineClass函数在Dalvik虚拟机里确实是被阉割了。而在这两个继承自ClassLoader的类加载器，本质上是重载了ClassLoader的findClass方法。在执行loadClass时，我们可以参看ClassLoader部分源码：

```

protected Class<?> loadClass(String className, boolean resolve) throws ClassNotFoundException {
 Class<?> clazz = findLoadedClass(className);
 if (clazz == null) {
 try {
 clazz = parent.loadClass(className, false);
 } catch (ClassNotFoundException e) {
 // Don't want to see this.
 }
 if (clazz == null) {
 clazz = findClass(className);
 }
 }
 return clazz;
}

```

因此DexClassLoader和PathClassLoader都属于符合双亲委派模型的类加载器（因为它们没有重载loadClass方法）。也就是说，它们在加载一个类之前，回去检查自己以及自己以上的类加载器是否已经加载了这个类。如果已经加载过了，就会直接将之返回，而不会重复加载。

DexClassLoader和PathClassLoader其实都是通过DexFile这个类来实现类加载的。这里需要顺便提一下的是，Dalvik虚拟机识别的是dex文件，而不是class文件。因此，我们供类加载的文件也只能是dex文件，或者包含有dex文件的.apk或.jar文件。

也许有人想到，既然DexFile可以直接加载类，那么我们为什么还要使用ClassLoader的子类呢？DexFile在加载类时，具体是调用成员方法loadClass或者loadClassBinaryName。其中loadClassBinaryName需要将包含包名的类名中的“.”转换为“/”我们看一下loadClass代码就清楚了：

```

public Class loadClass(String name, ClassLoader loader) {
 String slashName = name.replace('.', '/');
 return loadClassBinaryName(slashName, loader);
}

```

在这段代码前有一段注释，截取关键一部分就是说：If you are not calling this from a class loader, this is most likely not going to do what you want. Use {@link Class#forName(String)} instead. 这就是我们需要使用ClassLoader子类的原因。至于它是如何验证是否是在ClassLoader中调用此方法的，我没有研究，大家如果有兴趣可以继续深入下去。

有一个细节，可能大家不容易注意到。PathClassLoader是通过构造函数new DexFile(path)来产生DexFile对象的；而DexClassLoader则是通过其静态方法loadDex (path, outpath, 0) 得到DexFile对象。这两者的区别在于DexClassLoader需要提供一个可写的outpath路径，用来释放.apk包或者.jar包中的dex文件。换个说法来说，就是PathClassLoader不能主动从zip包中释放出dex，因此只支持直接操作dex格式文件，或者已经安装的apk（因为已经安装的apk在cache中存在缓存的dex文件）。而DexClassLoader可以支持.apk、.jar和.dex文件，并且会在指定的outpath路径释放出dex文件。

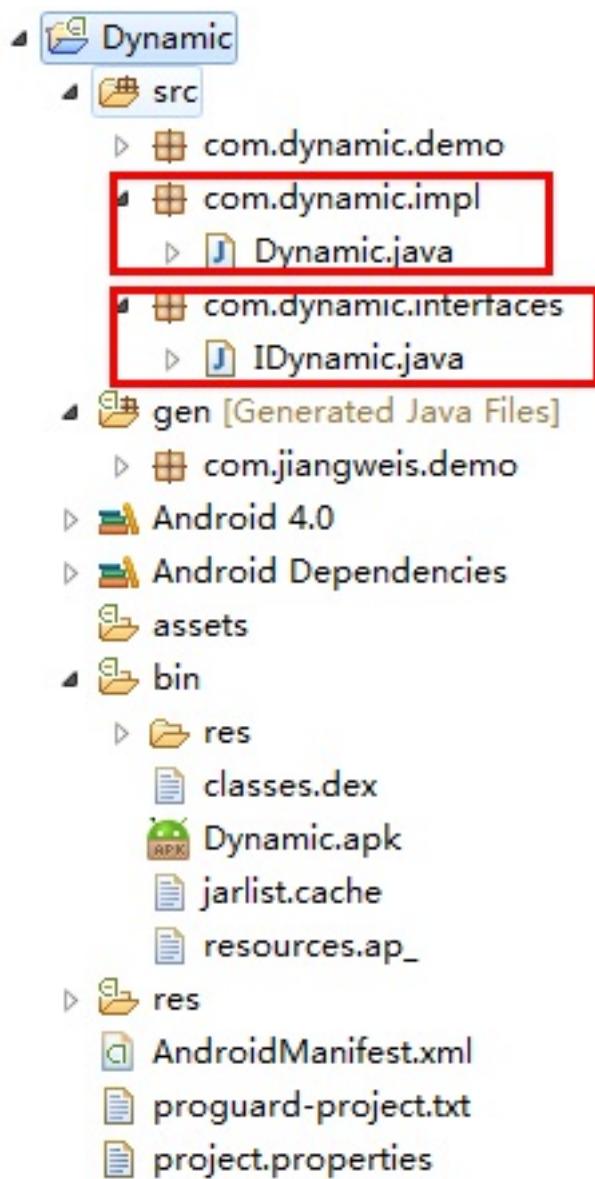
另外，PathClassLoader在加载类时调用的是DexFile的loadClassBinaryName，而DexClassLoader调用的是loadClass。因此，在使用PathClassLoader时类全名需要用"/"替换"."

## 实际操作

使用到的工具都比较常规：javac、dx、eclipse等其中dx工具最好是指明--no-strict，因为class文件的路径可能不匹配 加载好类后，通常我们可以通过Java反射机制来使用这个类但是这样效率相对不高，而且老用反射代码也比较复杂凌乱。更好的做法是定义一个interface，并将这个interface写进容器端。待加载的类，继承自这个interface，并且有一个参数为空的构造函数，以使我们能够通过Class的newInstance方法产生对象然后将对象强制转换为interface对象，于是就可以直接调用成员方法了，下面是具体的实现步骤了：

### 第一步：

编写好动态代码类：



```

package com.dynamic.interfaces;
import android.app.Activity;
/**
 * 动态加载类的接口
 */
public interface IDynamic {
 /**初始化方法*/
 public void init(Activity activity);
 /**自定义方法*/
 public void showBanner();
 public void showDialog();
 public void showFullScreen();
 public void showAppWall();
 /**销毁方法*/
 public void destory();
}

```

实现类代码如下：

```
package com.dynamic.impl;

import android.app.Activity;
import android.widget.Toast;

import com.dynamic.interfaces.IDynamic;

/**
 * 动态类的实现
 *
 */
public class Dynamic implements IDynamic{

 private Activity mActivity;

 @Override
 public void init(Activity activity) {
 mActivity = activity;
 }

 @Override
 public void showBanner() {
 Toast.makeText(mActivity, "我是ShowBanner方法", 1500).show();
 }

 @Override
 public void showDialog() {
 Toast.makeText(mActivity, "我是ShowDialog方法", 1500).show();
 }

 @Override
 public void showFullScreen() {
 Toast.makeText(mActivity, "我是ShowFullScreen方法", 1500).show();
 }

 @Override
 public void showAppWall() {
 Toast.makeText(mActivity, "我是ShowAppWall方法", 1500).show();
 }

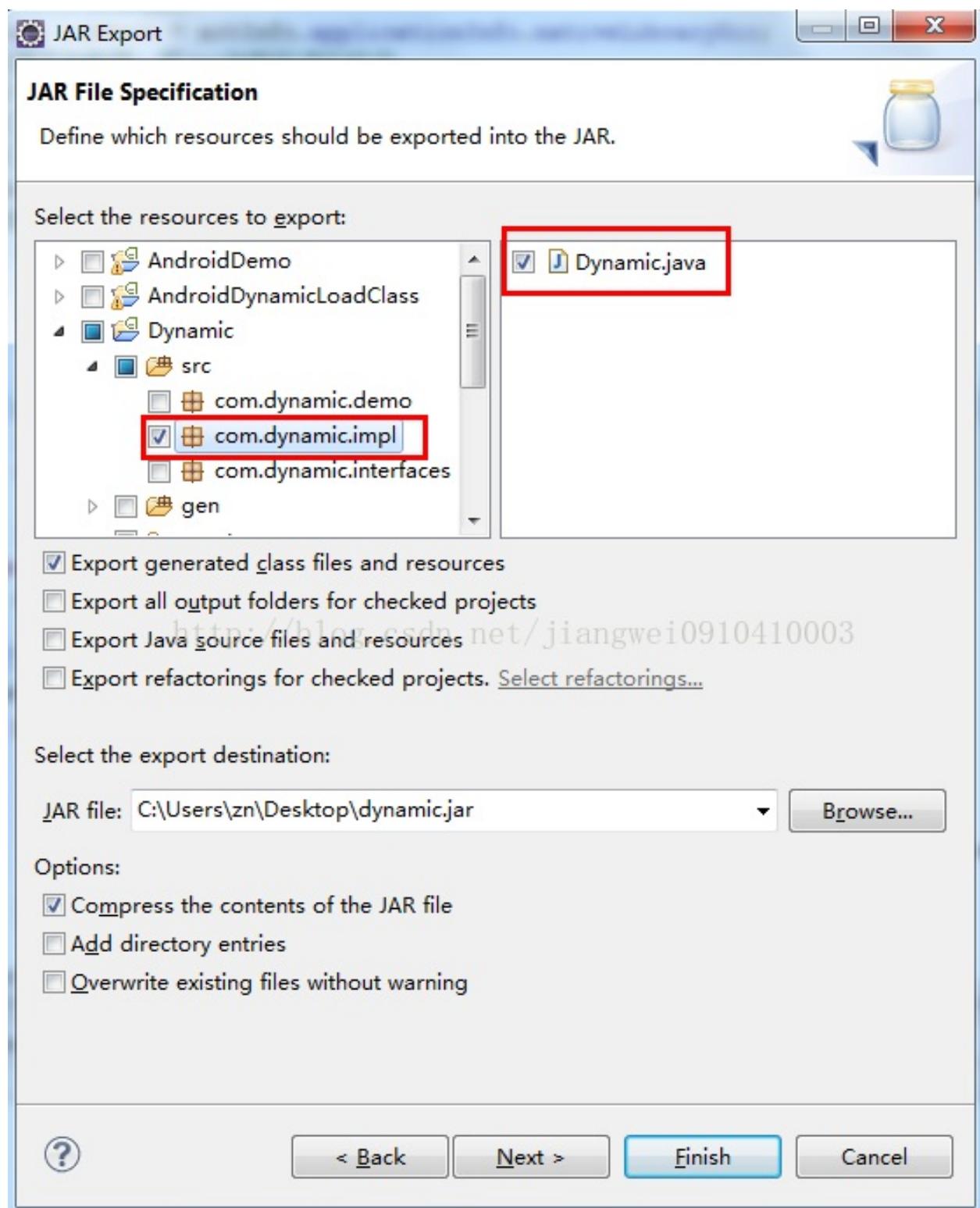
 @Override
 public void destory() {
 }

}
```

这样动态类就开发好了

## 第二步：

将上面开发好的动态类打包成.jar，这里要注意的是只打包实现类Dynamic.java，不打包接口类IDynamic.java，



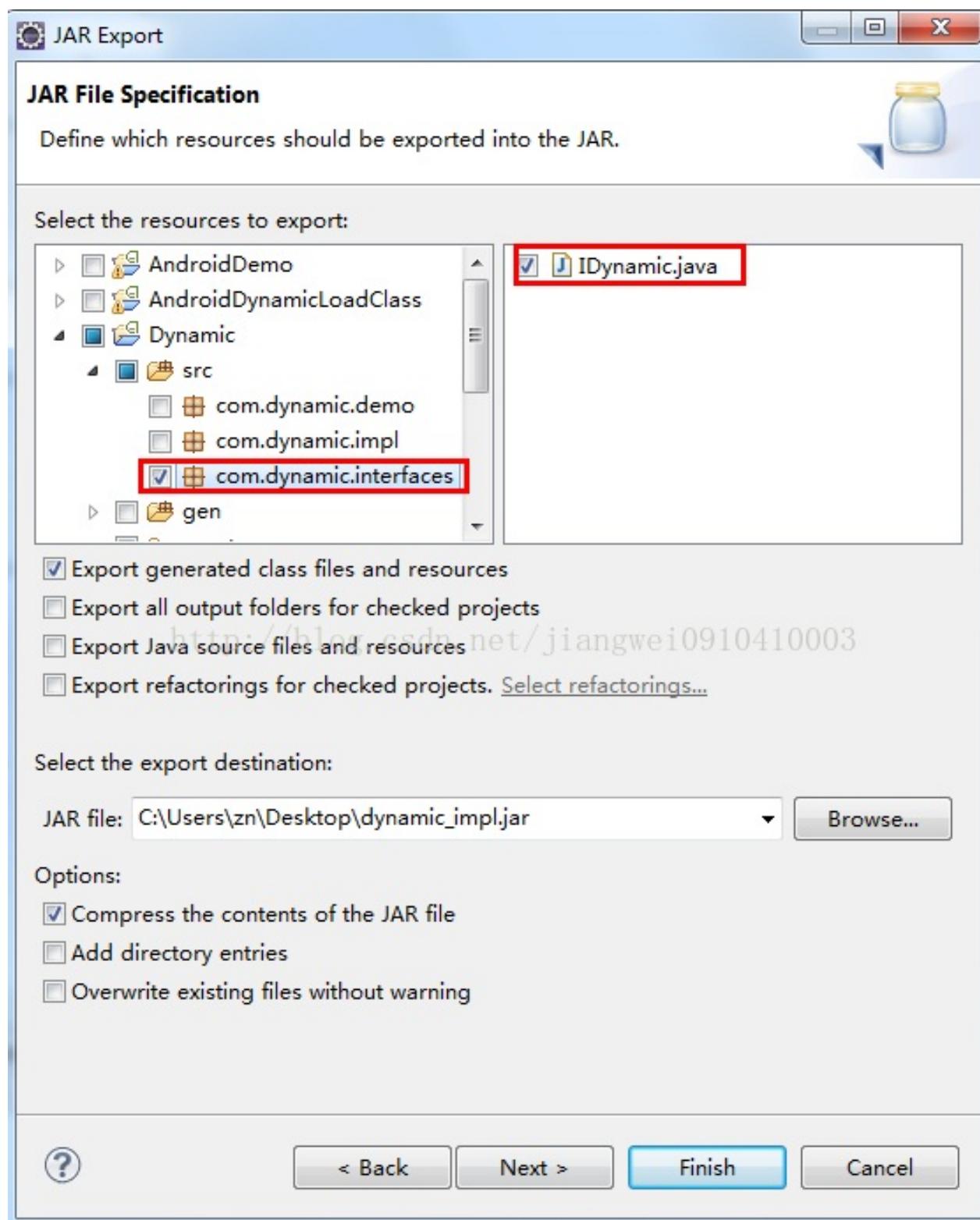
然后将打包好的jar文件拷贝到android的安装目录中的platform-tools目录下，使用dx命令：  
(我的jar文件是dynamic.jar)

```
dx --dex --output=dynamic_temp.jar dynamic.jar
```

这样就生成了dynamic\_temp.jar，这个jar和dynamic.jar有什么区别呢？

其实这条命令主要做的工作是：首先将dynamic.jar编译成dynamic.dex文件（Android虚拟机认识的字节码文件），然后再将dynamic.dex文件压缩成dynamic\_temp.jar，当然你也可以压缩成.zip格式的，或者直接编译成.apk文件都可以的，这个后面会说到。

到这里还不算完事，因为你想用什么来连接动态类和目标类呢？那就是动态类的接口了，所以这时候还要打个.jar包，这时候只需要打接口类IDynamic.java了



然后将这个.jar文件引用到目标类中，下面来看一下目标类的实现：

```
package com.jiangwei.demo;

import java.io.File;
import java.util.List;

import android.app.Activity;
import android.content.Intent;
import android.content.pm.ActivityInfo;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

import com.dynamic.interfaces.IDynamic;

import dalvik.system.DexClassLoader;
import dalvik.system.PathClassLoader;

public class AndroidDynamicLoadClassActivity extends Activity {

 //动态类加载接口
 private IDynamic lib;

 @Override
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.main);
 //初始化组件
 Button showBannerBtn = (Button) findViewById(R.id.show_banner_btn);
 Button showDialogBtn = (Button) findViewById(R.id.show_dialog_btn);
 Button showFullScreenBtn = (Button) findViewById(R.id.show_fullscreen_btn);
 Button showAppWallBtn = (Button) findViewById(R.id.show_appwall_btn);
 /**使用DexClassLoader方式加载类*/
 //dex压缩文件的路径(可以是apk,jar,zip格式)
 String dexPath = Environment.getExternalStorageDirectory().toString() + File.separator;
 //dex解压释放后的目录
 //String dexOutputDir = getApplicationInfo().dataDir;
 String dexOutputDirs = Environment.getExternalStorageDirectory().toString();
 //定义DexClassLoader
 //第一个参数：是dex压缩文件的路径
 //第二个参数：是dex解压缩后存放的目录
 //第三个参数：是C/C++依赖的本地库文件目录，可以为null
 //第四个参数：是上一级的类加载器
 DexClassLoader cl = new DexClassLoader(dexPath,dexOutputDirs,null,getClassLoa

 /**使用PathClassLoader方法加载类*/
 }
}
```

```
//创建一个意图，用来找到指定的apk：这里的"com.dynamic.impl"是指定apk在AndroidManifest.xml中的包名
Intent intent = new Intent("com.dynamic.impl", null);
//获得包管理器
PackageManager pm = getPackageManager();
List<ResolveInfo> resolveinfoes = pm.queryIntentActivities(intent, 0);
//获得指定的activity的信息
ActivityInfo actInfo = resolveinfoes.get(0).activityInfo;
//获得apk的目录或者jar的目录
String apkPath = actInfo.applicationInfo.sourceDir;
//native代码的目录
String libPath = actInfo.applicationInfo.nativeLibraryDir;
//创建类加载器，把dex加载到虚拟机中
//第一个参数：是指定apk安装的路径，这个路径要注意只能是通过actInfo.applicationInfo.sourceDir
//第二个参数：是C/C++依赖的本地库文件目录，可以为null
//第三个参数：是上一级的类加载器
PathClassLoader pcl = new PathClassLoader(apkPath, libPath, this.getClassLoader());
//加载类
try {
 //com.dynamic.impl.Dynamic是动态类名
 //使用DexClassLoader加载类
 //Class libProviderClazz = cl.loadClass("com.dynamic.impl.Dynamic");
 //使用PathClassLoader加载类
 Class libProviderClazz = pcl.loadClass("com.dynamic.impl.Dynamic");
 lib = (IDynamic)libProviderClazz.newInstance();
 if(lib != null){
 lib.init(AndroidDynamicLoadClassActivity.this);
 }
} catch (Exception exception) {
 exception.printStackTrace();
}
/**下面分别调用动态类中的方法*/
showBannerBtn.setOnClickListener(new View.OnClickListener() {
 public void onClick(View view) {
 if(lib != null){
 lib.showBanner();
 }else{
 Toast.makeText(getApplicationContext(), "类加载失败", 1500).show();
 }
 }
});
showDialogBtn.setOnClickListener(new View.OnClickListener() {
 public void onClick(View view) {
 if(lib != null){
 lib.showDialog();
 }else{
 Toast.makeText(getApplicationContext(), "类加载失败", 1500).show();
 }
 }
});
showFullScreenBtn.setOnClickListener(new View.OnClickListener() {
 public void onClick(View view) {
 if(lib != null){
```

```

 lib.showFullScreen();
 }else{
 Toast.makeText(getApplicationContext(), "类加载失败", 1500).show();
 }
}
);
showAppWallBtn.setOnClickListener(new View.OnClickListener() {
 public void onClick(View view) {
 if(lib != null){
 lib.showAppWall();
 }else{
 Toast.makeText(getApplicationContext(), "类加载失败", 1500).show();
 }
 }
});
}
}

```

这里面定义了一个IDynamic接口变量，同时使用了DexClassLoader和PathClassLoader来加载类，这里面先来说一说DexClassLoader方式加载：

```

//定义DexClassLoader
//第一个参数：是dex压缩文件的路径
//第二个参数：是dex解压缩后存放的目录
//第三个参数：是C/C++依赖的本地库文件目录，可以为null
//第四个参数：是上一级的类加载器
DexClassLoader cl = new DexClassLoader(dexPath,dexOutputDirs,null,getClassLoader());

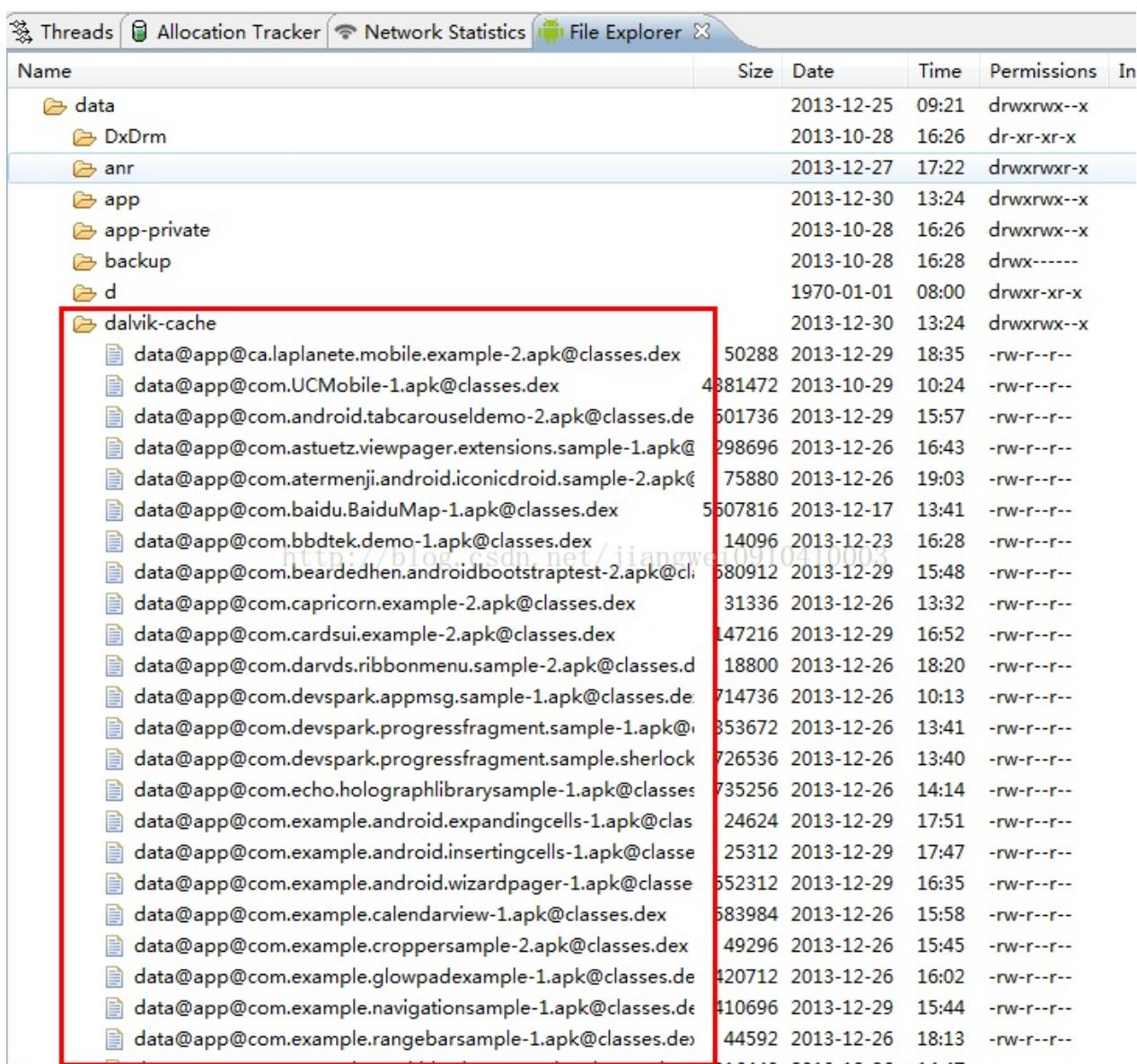
```

上面已经说了， DexClassLoader是继承ClassLoader类的，这里面的参数说明：

第一个参数是:dex压缩文件的路径:这个就是我们将上面编译后的dynamic\_temp.jar存放的目录，当然也可以是.zip和.apk格式的

第二个参数是:dex解压后存放的目录:这个就是将.jar,.zip,.apk文件解压出的dex文件存放的目录，这个就和PathClassLoader方法有区别了，同时你也可以看到PathClassLoader方法中没有这个参数，这个也真是这两个类的区别：

PathClassLoader不能主动从zip包中释放出dex，因此只支持直接操作dex格式文件，或者已经安装的apk（因为已经安装的apk在手机的data/dalvik目录中存在缓存的dex文件）。而DexClassLoader可以支持.apk、.jar和.dex文件，并且会在指定的outpath路径释放出dex文件。



| Name                                                           | Size    | Date       | Time  | Permissions | In |
|----------------------------------------------------------------|---------|------------|-------|-------------|----|
| data                                                           |         | 2013-12-25 | 09:21 | drwxrwx--x  |    |
| DxDrm                                                          |         | 2013-10-28 | 16:26 | dr-xr-xr-x  |    |
| anr                                                            |         | 2013-12-27 | 17:22 | drwxrwxr-x  |    |
| app                                                            |         | 2013-12-30 | 13:24 | drwxrwx--x  |    |
| app-private                                                    |         | 2013-10-28 | 16:26 | drwxrwx--x  |    |
| backup                                                         |         | 2013-10-28 | 16:28 | drwx-----   |    |
| d                                                              |         | 1970-01-01 | 08:00 | drwxr-xr-x  |    |
| dalvik-cache                                                   |         | 2013-12-30 | 13:24 | drwxrwx--x  |    |
| data@app@ca.laplanete.mobile.example-2.apk@classes.dex         | 50288   | 2013-12-29 | 18:35 | -rw-r---    |    |
| data@app@com.UCMobile-1.apk@classes.dex                        | 4381472 | 2013-10-29 | 10:24 | -rw-r---    |    |
| data@app@com.android.tabcarouseldemo-2.apk@classes.dex         | 501736  | 2013-12-29 | 15:57 | -rw-r---    |    |
| data@app@com.astuetz.viewpager.extensions.sample-1.apk@classe  | 298696  | 2013-12-26 | 16:43 | -rw-r---    |    |
| data@app@com.atermenji.android.iconicdroid.sample-2.apk@classe | 75880   | 2013-12-26 | 19:03 | -rw-r---    |    |
| data@app@com.baidu.BaiduMap-1.apk@classes.dex                  | 5607816 | 2013-12-17 | 13:41 | -rw-r---    |    |
| data@app@com.bbdtek.demo-1.apk@classes.dex                     | 14096   | 2013-12-23 | 16:28 | -rw-r---    |    |
| data@app@com.beardedhen.androidbootstrapTest-2.apk@classe      | 580912  | 2013-12-29 | 15:48 | -rw-r---    |    |
| data@app@com.capricorn.example-2.apk@classes.dex               | 31336   | 2013-12-26 | 13:32 | -rw-r---    |    |
| data@app@com.cardsui.example-2.apk@classes.dex                 | 147216  | 2013-12-29 | 16:52 | -rw-r---    |    |
| data@app@com.darvds.ribbonmenu.sample-2.apk@classes.dex        | 18800   | 2013-12-26 | 18:20 | -rw-r---    |    |
| data@app@com.devspark.appmsg.sample-1.apk@classes.dex          | 714736  | 2013-12-26 | 10:13 | -rw-r---    |    |
| data@app@com.devspark.progressfragment.sample-1.apk@classe     | 353672  | 2013-12-26 | 13:41 | -rw-r---    |    |
| data@app@com.devspark.progressfragment.sample.sherlock         | 726536  | 2013-12-26 | 13:40 | -rw-r---    |    |
| data@app@com.echo.holographlibrarysample-1.apk@classes.dex     | 735256  | 2013-12-26 | 14:14 | -rw-r---    |    |
| data@app@com.example.android.expandingcells-1.apk@classe       | 24624   | 2013-12-29 | 17:51 | -rw-r---    |    |
| data@app@com.example.android.insertingcells-1.apk@classe       | 25312   | 2013-12-29 | 17:47 | -rw-r---    |    |
| data@app@com.example.android.wizardpager-1.apk@classe          | 552312  | 2013-12-29 | 16:35 | -rw-r---    |    |
| data@app@com.example.calendarview-1.apk@classes.dex            | 583984  | 2013-12-26 | 15:58 | -rw-r---    |    |
| data@app@com.example.croppersample-2.apk@classes.dex           | 49296   | 2013-12-26 | 15:45 | -rw-r---    |    |
| data@app@com.example.glowpadexample-1.apk@classes.dex          | 420712  | 2013-12-26 | 16:02 | -rw-r---    |    |
| data@app@com.example.navigationsample-1.apk@classes.dex        | 410696  | 2013-12-29 | 15:44 | -rw-r---    |    |
| data@app@com.example.rangebarsample-1.apk@classes.dex          | 44592   | 2013-12-26 | 18:13 | -rw-r---    |    |

然而我们可以通过DexClassLoader方法指定解压后的dex文件的存放目录，但是我们一般不这么做，因为这样做无疑的暴露了dex文件，所以我们一般不会将.jar/.zip/.apk压缩文件存放到用户可以察觉到的位置，同时解压dex的目录也是不能让用户看到的。

第三个参数和第四个参数用到的不是很多，所以这里就不做太多的解释了。

这里还要注意一点就是PathClassLoader方法的时候，第一个参数是dex存放的路径，这里传递的是：

```
//获得apk的目录或者jar的目录
String apkPath = actInfo.applicationInfo.sourceDir;
```

指定的apk安装路径,这个值只能这样获取，不然会加载类失败的

### 第三步：

运行目标类:

要做的工作是:

如果用的是DexClassLoader方式加载类:这时候需要将.jar或者.zip或者.apk文件放到指定的目录中, 我这里为了方便就放到sd卡的根目录中

如果用的是PathClassLoader方法加载类:这时候需要先将Dynamic.apk安装到手机中, 不然找不到这个activity, 同时要注意的是:

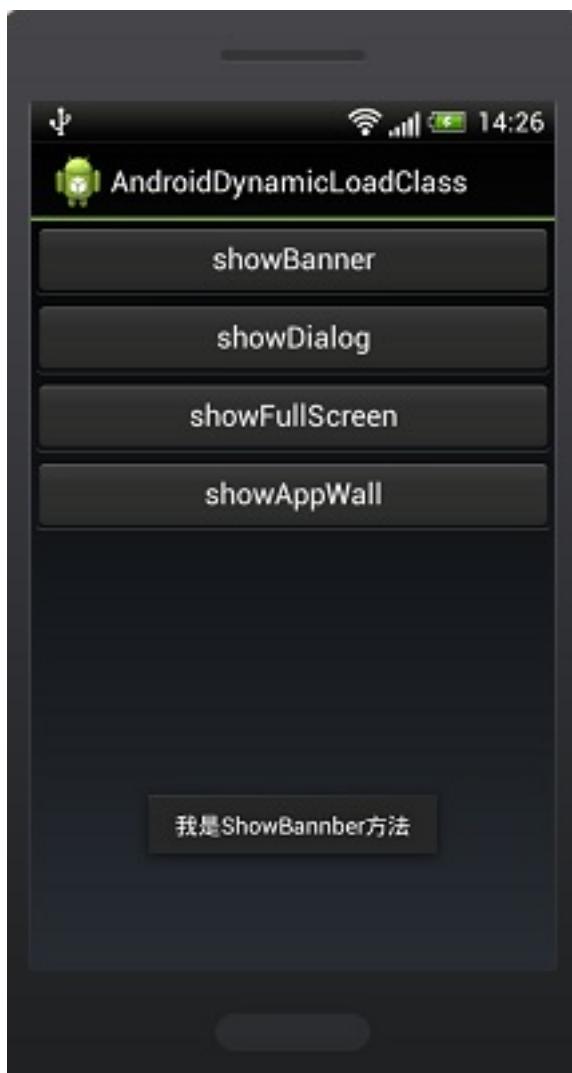
在CODE上查看代码片派生到我的代码片

```
//创建一个意图, 用来找到指定的apk: 这里的"com.dynamic.impl"是指定apk在AndroidManifest.xml文件
Intent intent = new Intent("com.dynamic.impl", null);
```

这里的com.dynamic.impl是一个action需要在指定的apk中定义, 这个名称是动态apk和目标apk之间约定好的

```
<application
 android:icon="@drawable/ic_launcher"
 android:label="@string/app_name" >
 <activity
 android:name="com.dynamic.demo.DynamicActivity"
 android:label="@string/app_name" >
 <intent-filter>
 <action android:name="android.intent.action.MAIN" />
 <action android:name="com.dynamic.impl" />
 <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
 </activity>
</application>
```

运行结果



点击showBanner显示一个Toast，成功的运行了动态类中的代码！其实更好的办法就是将动态的.jar.zip.apk文件从网络上获取，安全可靠，同时本地的目标项目不需要改动代码就可以执行不同的逻辑了

## 关于代码加密的一些设想

最初设想将dex文件加密，然后通过JNI将解密代码写在Native层。解密之后直接传上二进制流，再通过defineClass将类加载到内存中。

现在也可以这样做，但是由于不能直接使用defineClass，而必须传文件路径给dalvik虚拟机内核，因此解密后的文件需要写到磁盘上，增加了被破解的风险。

Dalvik虚拟机内核仅支持从dex文件加载类的方式是不灵活的，由于没有非常深入的研究内核，我不能确定是Dalvik虚拟机本身不支持还是Android在移植时将其阉割了。不过相信Dalvik或者是Android开源项目都正在向能够支持raw数据定义类方向努力。

我们可以在文档中看到Google说： Jar or APK file with "classes.dex". (May expand this to include "raw DEX" in the future.); 在Android的Dalvik源码中我们也能看到RawDexFile的身影（不过没有具体实现）

在RawDexFile出来之前，我们都只能使用这种存在一定风险的加密方式。需要注意释放的dex文件路径及权限管理，另外，在加载完毕类之后，除非出于其他目的否则应该马上删除临时的解密文件

# images

# 进阶

# 5

# Android apk动态加载机制的研究

来源:singwhatiwanna的csdn博客

转载请注明出处：<http://blog.csdn.net/singwhatiwanna/article/details/22597587>（来自singwhatiwanna的csdn博客）

## 背景

问题是这样的：我们知道，apk必须安装才能运行，如果不安装要是也能运行该多好啊，事实上，这不是完全不可能的，尽管它比较难实现。在理论层面上，我们可以通过一个宿主程序来运行一些未安装的apk，当然，实践层面上也能实现，不过这对未安装的apk有要求。我们的想法是这样的，首先要明白apk未安装是不能被直接调起来的，但是我们可以采用一个程序（称之为宿主程序）去动态加载apk文件并将其放在自己的进程中执行，本文要介绍的就是这么一种方法，同时这种方法还有很多问题，尤其是资源的访问。因为将apk加载到宿主程序中去执行，就无法通过宿主程序的Context去取到apk中的资源，比如图片、文本等，这是很好理解的，因为apk已经不存在上下文了，它执行时所采用的上下文是宿主程序的上下文，用别人的Context是无法得到自己的资源的，不过这个问题貌似可以这么解决：将apk中的资源解压到某个目录，然后通过文件去操作资源，这只是理论上可行，实际上还是会有很多的难点的。除了资源存取的问题，还有一个问题是activity的生命周期，因为apk被宿主程序加载执行后，它的activity其实就是一个普通的类，正常情况下，activity的生命周期是由系统来管理的，现在被宿主程序接管了以后，如何替代系统对apk中的activity的生命周期进行管理是有难度的，不过这个问题比资源的访问好解决一些，比如我们可以在宿主程序中模拟activity的生命周期并合适地调用apk中activity的生命周期方法。本文暂时不对这两个问题进行解决，因为很难，本文仅仅对apk的动态执行机制进行介绍，尽管如此，听起来还是有点小激动，不是吗？

## 工作原理

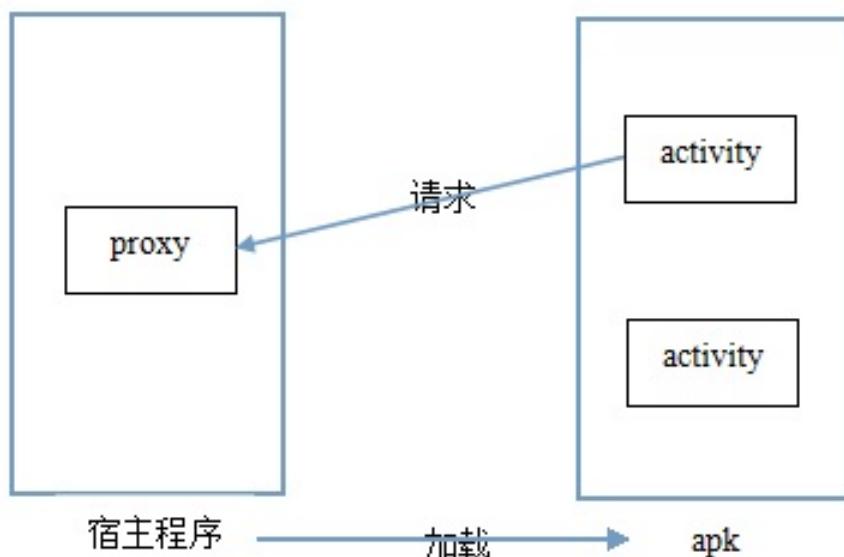
如下图所示，首先宿主程序会到文件系统比如sd卡去加载apk，然后通过一个叫做proxy的activity去执行apk中的activity。关于动态加载apk，理论上可以用到的有DexClassLoader、PathClassLoader和URLClassLoader。

- DexClassLoader：可以加载文件系统上的jar、dex、apk
- PathClassLoader：可以加载/data/app目录下的apk，这也意味着，它只能加载已经安装的apk

- URLClassLoader：可以加载java中的jar，但是由于dalvik不能直接识别jar，所以此方法在Android中无法使用，尽管还有这个类

关于jar、dex和apk，dex和apk是可以直接加载的，因为它们都是或者内部有dex文件，而原始的jar是不行的，必须转换成dalvik所能识别的字节码文件，转换工具可以使用android sdk中platform-tools目录下的dx

转换命令：`dx --dex --output=dest.jar src.jar`



<http://blog.csdn.net/singwhatiwanna>

## 示例

### 宿主程序的实现

1. 主界面很简单，放了一个button，点击就会调起apk，我把apk直接放在了sd卡中，至于先把apk从网上下载到本地再加载其实是一个道理。

```

@Override
public void onClick(View v) {
 if (v == mOpenClient) {
 Intent intent = new Intent(this, ProxyActivity.class);
 intent.putExtra(ProxyActivity.EXTRA_DEX_PATH,
 "/mnt/sdcard/DynamicLoadHost/plugin.apk");
 startActivity(intent);
 }
}

```

点击button以后， proxy会被调起， 然后加载apk并调起的任务就交给它了

## 2.代理activity的实现（proxy）

```

package com.ryg.dynamicloadhost;

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

import dalvik.system.DexClassLoader;
import android.annotation.SuppressLint;
import android.app.Activity;
import android.content.pm.PackageInfo;
import android.os.Bundle;
import android.util.Log;

public class ProxyActivity extends Activity {

 private static final String TAG = "ProxyActivity";

 public static final String FROM = "extra.from";
 public static final int FROM_EXTERNAL = 0;
 public static final int FROM_INTERNAL = 1;

 public static final String EXTRA_DEX_PATH = "extra.dex.path";
 public static final String EXTRA_CLASS = "extra.class";

 private String mClass;
 private String mDexPath;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 mDexPath = getIntent().getStringExtra(EXTRA_DEX_PATH);
 mClass = getIntent().getStringExtra(EXTRA_CLASS);

 Log.d(TAG, "mClass=" + mClass + " mDexPath=" + mDexPath);
 if (mClass == null) {

```

```

 launchTargetActivity();
 } else {
 launchTargetActivity(mClass);
 }
}

@SuppressWarnings("NewApi")
protected void launchTargetActivity() {
 PackageInfo packageInfo = getPackageManager().getPackageArchiveInfo(
 mDexPath, 1);
 if ((packageInfo.activities != null)
 && (packageInfo.activities.length > 0)) {
 String activityName = packageInfo.activities[0].name;
 mClass = activityName;
 launchTargetActivity(mClass);
 }
}

@SuppressWarnings("NewApi")
protected void launchTargetActivity(final String className) {
 Log.d(TAG, "start launchTargetActivity, className=" + className);
 File dexOutputDir = this.getDir("dex", 0);
 final String dexOutputPath = dexOutputDir.getAbsolutePath();
 ClassLoader localClassLoader = ClassLoader.getSystemClassLoader();
 DexClassLoader dexClassLoader = new DexClassLoader(mDexPath,
 dexOutputPath, null, localClassLoader);
 try {
 Class<?> localClass = dexClassLoader.loadClass(className);
 Constructor<?> localConstructor = localClass
 .getConstructor(new Class[] {});
 Object instance = localConstructor.newInstance(new Object[] {});
 Log.d(TAG, "instance = " + instance);

 Method setProxy = localClass.getMethod("setProxy",
 new Class[] { Activity.class });
 setProxy.setAccessible(true);
 setProxy.invoke(instance, new Object[] { this });

 Method onCreate = localClass.getDeclaredMethod("onCreate",
 new Class[] { Bundle.class });
 onCreate.setAccessible(true);
 Bundle bundle = new Bundle();
 bundle.putInt(FROM, FROM_EXTERNAL);
 onCreate.invoke(instance, new Object[] { bundle });
 } catch (Exception e) {
 e.printStackTrace();
 }
}
}

```

说明：程序不难理解，思路是这样的：采用DexClassLoader去加载apk，然后如果没有指定class，就调起主activity，否则调起指定的class。activity被调起的过程是这样的：首先通过类加载器去加载apk中activity的类并创建一个新对象，然后通过反射去调用这个对象的setProxy方法和onCreate方法，setProxy方法的作用是将activity内部的执行全部交由宿主程序中的proxy（也是一个activity），onCreate方法是activity的入口，setProxy以后就调用onCreate方法，这个时候activity就被调起来了。

## 待执行apk的实现

1.为了让proxy全面接管apk中所有activity的执行，需要为activity定义一个基类 BaseActivity，在基类中处理代理相关的事情，同时BaseActivity还对是否使用代理进行了判断，如果不使用代理，那么activity的逻辑仍然按照正常的方式执行，也就是说，这个apk既可以按照执行，也可以由宿主程序来执行。

```
package com.ryg.dynamicloadclient;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.ViewGroup.LayoutParams;

public class BaseActivity extends Activity {

 private static final String TAG = "Client-BaseActivity";

 public static final String FROM = "extra.from";
 public static final int FROM_EXTERNAL = 0;
 public static final int FROM_INTERNAL = 1;
 public static final String EXTRA_DEX_PATH = "extra.dex.path";
 public static final String EXTRA_CLASS = "extra.class";

 public static final String PROXY_VIEW_ACTION = "com.ryg.dynamicloadhost.VIEW";
 public static final String DEX_PATH = "/mnt/sdcard/DynamicLoadHost/plugin.apk";

 protected Activity mProxyActivity;
 protected int mFrom = FROM_INTERNAL;

 public void setProxy(Activity proxyActivity) {
 Log.d(TAG, "setProxy: proxyActivity= " + proxyActivity);
 mProxyActivity = proxyActivity;
 }

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 if (savedInstanceState != null) {

```

```
 mFrom = savedInstanceState.getInt(FROM, FROM_INTERNAL);
 }
 if (mFrom == FROM_INTERNAL) {
 super.onCreate(savedInstanceState);
 mProxyActivity = this;
 }
 Log.d(TAG, "onCreate: from= " + mFrom);
}

protected void startActivityByProxy(String className) {
 if (mProxyActivity == this) {
 Intent intent = new Intent();
 intent.setClassName(this, className);
 this.startActivity(intent);
 } else {
 Intent intent = new Intent(PROXY_VIEW_ACTION);
 intent.putExtra(EXTRA_DEX_PATH, DEX_PATH);
 intent.putExtra(EXTRA_CLASS, className);
 mProxyActivity.startActivity(intent);
 }
}

@Override
public void setContentView(View view) {
 if (mProxyActivity == this) {
 super.setContentView(view);
 } else {
 mProxyActivity.setContentView(view);
 }
}

@Override
public void setContentView(View view, LayoutParams params) {
 if (mProxyActivity == this) {
 super.setContentView(view, params);
 } else {
 mProxyActivity.setContentView(view, params);
 }
}

@Deprecated
@Override
public void setContentView(int layoutResID) {
 if (mProxyActivity == this) {
 super.setContentView(layoutResID);
 } else {
 mProxyActivity.setContentView(layoutResID);
 }
}

@Override
public void addContentView(View view, LayoutParams params) {
```

```
 if (mProxyActivity == this) {
 super.addContentView(view, params);
 } else {
 mProxyActivity.addView(view, params);
 }
}
```

说明：相信大家一看代码就明白了，其中setProxy方法的作用就是为了让宿主程序能够接管自己的执行，一旦被接管以后，其所有的执行均通过proxy，且Context也变成了宿主程序的Context，也许这么说比较形象：宿主程序其实就是一个空壳，它只是把其它apk加载到自己的内部去执行，这也就更能理解为什么资源访问变得很困难，你会发现好像访问不到apk中的资源了，的确是这样的，但是目前我还没有很好的方法去解决。

## 2. 入口activity的实现

```

public class MainActivity extends BaseActivity {

 private static final String TAG = "Client-MainActivity";

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 initView(savedInstanceState);
 }

 private void initView(Bundle savedInstanceState) {
 mProxyActivity.setContentView(generateContentView(mProxyActivity));
 }

 private View generateContentView(final Context context) {
 LinearLayout layout = new LinearLayout(context);
 layout.setLayoutParams(new LayoutParams(LayoutParams.MATCH_PARENT,
 LayoutParams.MATCH_PARENT));
 layout.setBackgroundColor(Color.parseColor("#F79AB5"));
 Button button = new Button(context);
 button.setText("button");
 layout.addView(button, LayoutParams.MATCH_PARENT,
 LayoutParams.WRAP_CONTENT);
 button.setOnClickListener(new OnClickListener() {
 @Override
 public void onClick(View v) {
 Toast.makeText(context, "you clicked button",
 Toast.LENGTH_SHORT).show();
 startActivityByProxy("com.ryg.dynamicloadclient.TestActivity");
 }
 });
 return layout;
 }

}

```

说明：由于访问不到apk中的资源了，所以界面是代码写的，而不是写在xml中，因为xml读不到了，这也是个大问题。注意到主界面中有一个button，点击后跳到了另一个activity，这个时候是不能直接调用系统的startActivity方法的，而是必须通过宿主程序中的proxy来执行，原因很简单，首先apk本书没有Context，所以它无法调起activity，另外由于这个子activity是apk中的，通过宿主程序直接调用它也是不行的，因为它对宿主程序来说是不可见的，所以只能通过proxy来调用，是不是感觉很麻烦？但是，你还有更好的办法吗？

### 3.子activity的实现

```
package com.ryg.dynamicloadclient;

import android.graphics.Color;
import android.os.Bundle;
import android.view.ViewGroup.LayoutParams;
import android.widget.Button;

public class TestActivity extends BaseActivity{

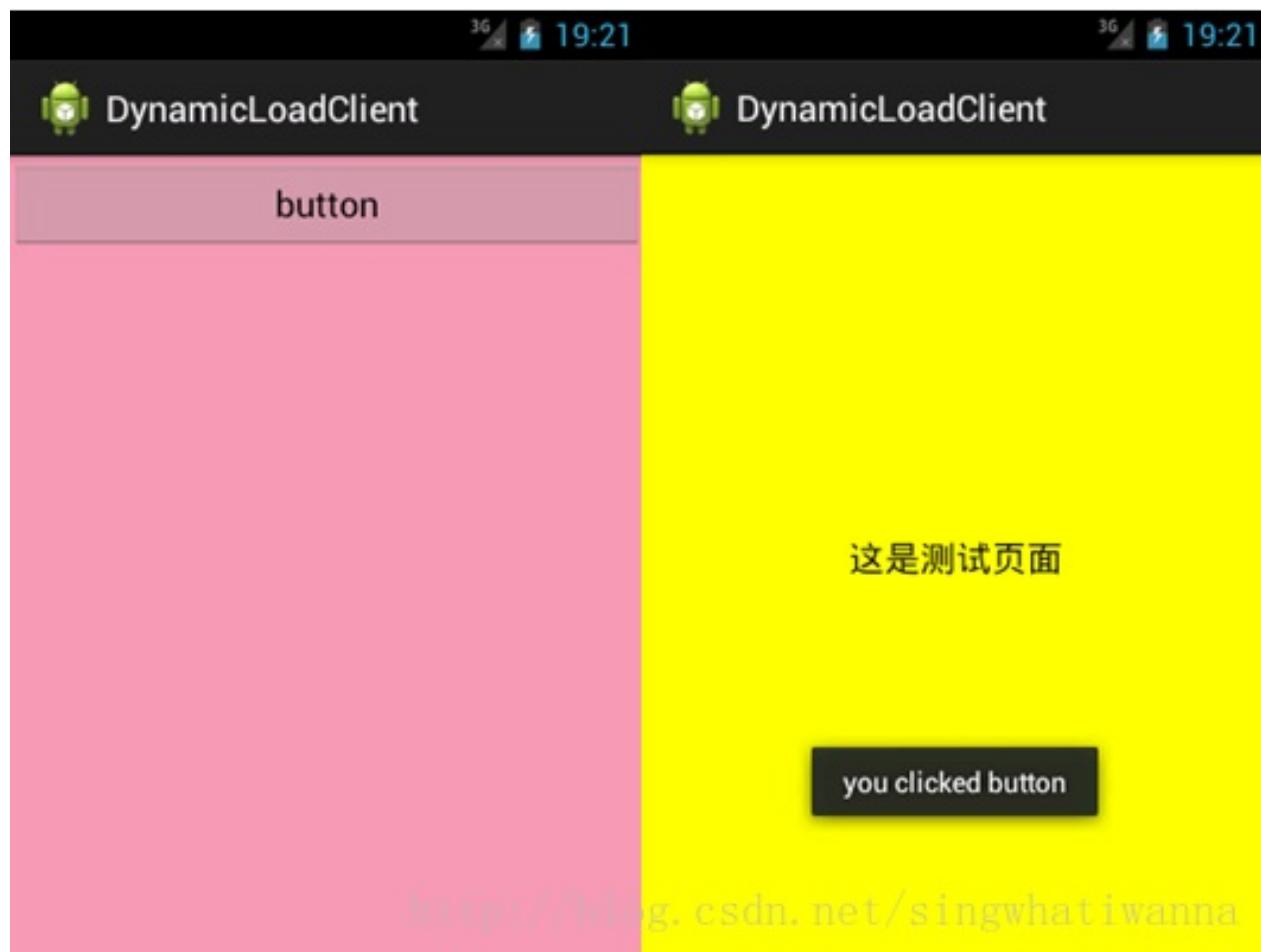
 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 Button button = new Button(mProxyActivity);
 button.setLayoutParams(new LayoutParams(LayoutParams.MATCH_PARENT,
 LayoutParams.MATCH_PARENT));
 button.setBackgroundColor(Color.YELLOW);
 button.setText("这是测试页面");
 setContentView(button);
 }

}
```

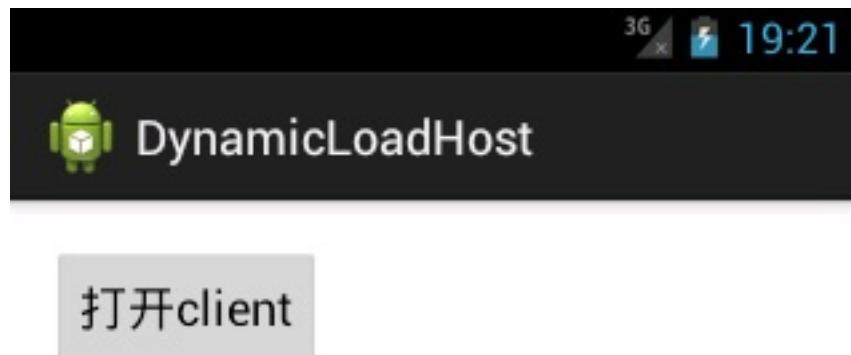
说明：代码很简单，不用介绍了，同理，界面还是用代码来写的。

## 运行效果

1.首先看apk安装时的运行效果



2.再看看未安装时被宿主程序执行的效果





说明：可以发现，安装和未安装，执行效果是一样的，差别在于：首先未安装的时候由于采用了反射，所以执行效率会略微降低，其次，应用的标题发生了改变，也就是说，尽管apk被执行了，但是它毕竟是在宿主程序里面执行的，所以它还是属于宿主程序的，因此apk未安装被执行时其标题不是自己的，不过这也可以间接证明，apk的确被宿主程序执行了，不信看标题。最后，我想说一下这么做的意义，这样做有利于实现模块化，同时还可以实现插件机制，但是问题还是很多的，最复杂的两个问题：资源的访问和activity生命周期的管理，期待大家有好的解决办法，欢迎交流。

代码下载：

<https://github.com/singwhatiwanna/dynamic-load-apk>

<http://download.csdn.net/detail/singwhatiwanna/7121505>

# Android apk动态加载机制的研究（二）：资源加载和activity生命周期管理

来源:singwhatiwanna的csdn博客

转载请注明出处：<http://blog.csdn.net/singwhatiwanna/article/details/23387079> （来自singwhatiwanna的csdn博客）

## 前言

为了更好地阅读本文，你需要先阅读[Android apk动态加载机制的研究](#)，在此文中，博主分析了Android中apk的动态加载机制，并在文章的最后指出需要解决的两个复杂问题：资源的访问和activity生命周期的管理，而本文将会分析这两个复杂问题的解决方法。需要说明的一点是，我们不可能调起任何一个未安装的apk，这在技术上是无法实现的，我们调起的apk必须受某种规范的约束，只有在这种约束下开发的apk，我们才能将其调起。另外，本文给出的解决方案也不是完美的，但是逻辑已经可以正常地跑通了，剩下的极个别细节问题是可优化的。

## 资源管理

我们知道，宿主程序调起未安装的apk，一个很大的问题就是资源如何访问，具体来说就是，凡是以R开头的资源都不能访问了，因为宿主程序中并没有apk中的资源，所以通过R来加载资源是行不通的，程序会报错：无法找到某某id所对应的资源。针对这个问题，有人提出了将apk中的资源在宿主程序中也copy一份，这虽然能解决问题，可以一听起来就很奇怪，首先这样会持有两份资源，会增加宿主程序包的大小，其次，没发布一个插件都需要将资源copy到宿主程序中，这样就意味着每发布一个插件都要更新一下宿主程序，这和插件化的思想是相悖的，插件化的目的就是要减小宿主程序apk包的大小同时降低宿主程序的更新频率并做到自由装载模块。所以这种方法并不可行。还有人提供了一种方式：将apk中的资源解压出来，然后通过文件流去读取资源，这样做理论上是可行的，但是实际操作起来还是有很大难度的，首先不同资源有不同的文件流格式，比如图片、xml等，还有就是针对不同设备加载的资源可能是不一样的，如果选择合适的资源也是一个需要解决的问题，基于这两点，这种方法不建议使用，因为它实现起来有难度。下面说说本文所采用的方法。

我们知道，activity的工作主要是由ContextImpl来完成的，它在activity中是一个叫做mBase的成员变量。注意到Context中有如下两个抽象方法，看起来是和资源有关的，实际上context就是通过它们来获取资源的，这两个抽象方法的真正实现是在ContextImpl中。也即是说，只要我们自己实现这两个方法，就可以解决资源问题了。

```
/** Return an AssetManager instance for your application's package. */
public abstract AssetManager getAssets();
/** Return a Resources instance for your application's package. */
public abstract Resources getResources();
```

下面看一下如何实现这两个方法

首先要加载apk中的资源：

```
protected void loadResources() {
 try {
 AssetManager assetManager = AssetManager.class.newInstance();
 Method addAssetPath = assetManager.getClass().getMethod("addAssetPath", String.class);
 addAssetPath.invoke(assetManager, mDexPath);
 mAssetManager = assetManager;
 } catch (Exception e) {
 e.printStackTrace();
 }
 Resources superRes = super.getResources();
 mResources = new Resources(mAssetManager, superRes.getDisplayMetrics(),
 superRes.getConfiguration());
 mTheme = mResources.newTheme();
 mTheme.setTo(super.getTheme());
}
```

说明：加载的方法是通过反射，通过调用AssetManager中的addAssetPath方法，我们可以将一个apk中的资源加载到Resources中，由于addAssetPath是隐藏api我们无法直接调用，所以只能通过反射，下面是它的声明，通过注释我们可以看出，传递的路径可以是zip文件也可以是一个资源目录，而apk就是一个zip，所以直接将apk的路径传给它，资源就加载到AssetManager中了，然后再通过AssetManager来创建一个新的Resources对象，这个对象就是我们可以使用的apk中的资源了，这样我们的问题就解决了。

```
/**
 * Add an additional set of assets to the asset manager. This can be
 * either a directory or ZIP file. Not for use by applications. Returns
 * the cookie of the added asset, or 0 on failure.
 * {@hide}
 */
public final int addAssetPath(String path) {
 int res = addAssetPathNative(path);
 return res;
}
```

其次是要实现那两个抽象方法

```
@Override
public AssetManager getAssets() {
 return mAssetManager == null ? super.getAssets() : mAssetManager;
}

@Override
public Resources getResources() {
 return mResources == null ? super.getResources() : mResources;
}
```

okay，问题搞定。这样一来，在apk中就可以通过R来访问资源了。

## activity生命周期的管理

这是本文开头提到的另一个需要解决的难题。为什么会有这个问题，其实很好理解，apk被宿主程序调起以后，apk中的activity其实就是一个普通的对象，不具有activity的性质，因为系统启动activity是要做很多初始化工作的，而我们在应用层通过反射去启动activity是很难完成系统所做的初始化工作的，所以activity的大部分特性都无法使用包括activity的生命周期管理，这就需要我们自己去管理。谈到activity生命周期，其实就是那几个常见的方法：onCreate、onStart、onResume、onPause等，由于apk中的activity不是真正意义上的activity（没有在宿主程序中注册且没有完全初始化），所以这几个生命周期的方法系统就不会去自动调用了。针对此类问题，采用Fragment是一个不错的方法，Fragment从3.0引入，通过support-v4包，可以兼容3.0以下的android版本。Fragment既有类似于Activity的生命周期，又有类似于View的界面，将Fragment加入到Activity中，activity会自动管理Fragment的生命周期，通过第一篇文章我们知道，apk中的activity是通过宿主程序中的代理activity启动的，将Fragment加入到代理activity内部，其生命周期将完全由代理activity

来管理，但是采用这种方法，就要求apk尽量采用Fragment来实现，还有就是在做页面跳转的时候有点麻烦，当然关于Fragment相关的内容我将在后面再做研究，本文不采用Fragment而是通过反射去手动管理activity的生命周期。

我们要在代理activity中去反射apk中activity的所有生命周期的方法，然后将activity的生命周期和代理activity的生命周期进行同步。首先，反射activity生命周期的所有方法，还反射了onActivityResult这个方法，尽管它不是典型的生命周期方法，但是它很有用。

```
protected void instantiateLifecycleMethods(Class<?> localClass) {
 String[] methodNames = new String[] {
 "onRestart",
 "onStart",
 "onResume",
 "onPause",
 "onStop",
 "onDestory"
 };
 for (String methodName : methodNames) {
 Method method = null;
 try {
 method = localClass.getDeclaredMethod(methodName, new Class[] { });
 method.setAccessible(true);
 } catch (NoSuchMethodException e) {
 e.printStackTrace();
 }
 mActivityLifecycleMethods.put(methodName, method);
 }

 Method onCreate = null;
 try {
 onCreate = localClass.getDeclaredMethod("onCreate", new Class[] { Bundle.class });
 onCreate.setAccessible(true);
 } catch (NoSuchMethodException e) {
 e.printStackTrace();
 }
 mActivityLifecycleMethods.put("onCreate", onCreate);

 Method onActivityResult = null;
 try {
 onActivityResult = localClass.getDeclaredMethod("onActivityResult",
 new Class[] { int.class, int.class, Intent.class });
 onActivityResult.setAccessible(true);
 } catch (NoSuchMethodException e) {
 e.printStackTrace();
 }
 mActivityLifecycleMethods.put("onActivityResult", onActivityResult);
}
```

其次，同步生命周期，主要看一下onResume和onPause，其他方法是类似的。看如下代码，很好理解，就是当系统调用代理activity生命周期方法的时候，就通过反射去显式调用apk中activity的对应方法。

```

@Override
protected void onResume() {
 super.onResume();
 Method onResume = mActivityLifeCircleMethods.get("onResume");
 if (onResume != null) {
 try {
 onResume.invoke(mRemoteActivity, new Object[] { });
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
}

@Override
protected void onPause() {
 Method onPause = mActivityLifeCircleMethods.get("onPause");
 if (onPause != null) {
 try {
 onPause.invoke(mRemoteActivity, new Object[] { });
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
 super.onPause();
}

```

## 插件apk的开发规范

文章开头提到，要想成为一个插件apk，是要满足一定条件的，如下是采用本文机制开发插件apk所需要遵循的规范：

- 1.不能用this：因为this指向的是当前对象，即apk中的activity，但是由于activity已经不是常规意义上的activity，所以this是没有意义的。
- 2.使用that：既然this不能用，那就用that，that是apk中activity的基类 BaseActivity 中的一个成员，它在apk安装运行的时候指向this，而在未安装的时候指向宿主程序中的代理activity，anyway，that is better than this。
- 3.不能直接调用activity的成员方法：而必须通过that去调用，由于that的动态分配特性，通过that去调用activity的成员方法，在apk安装以后仍然可以正常运行。
- 4.启动新activity的约束：启动外部activity不受限制，启动apk内部的activity有限制，首先由于apk中的activity没注册，所以不支持隐式调用，其次必须通过BaseActivity中

定义的新方法startActivityByProxy和startActivityForResultByProxy，还有就是不支持LaunchMode。

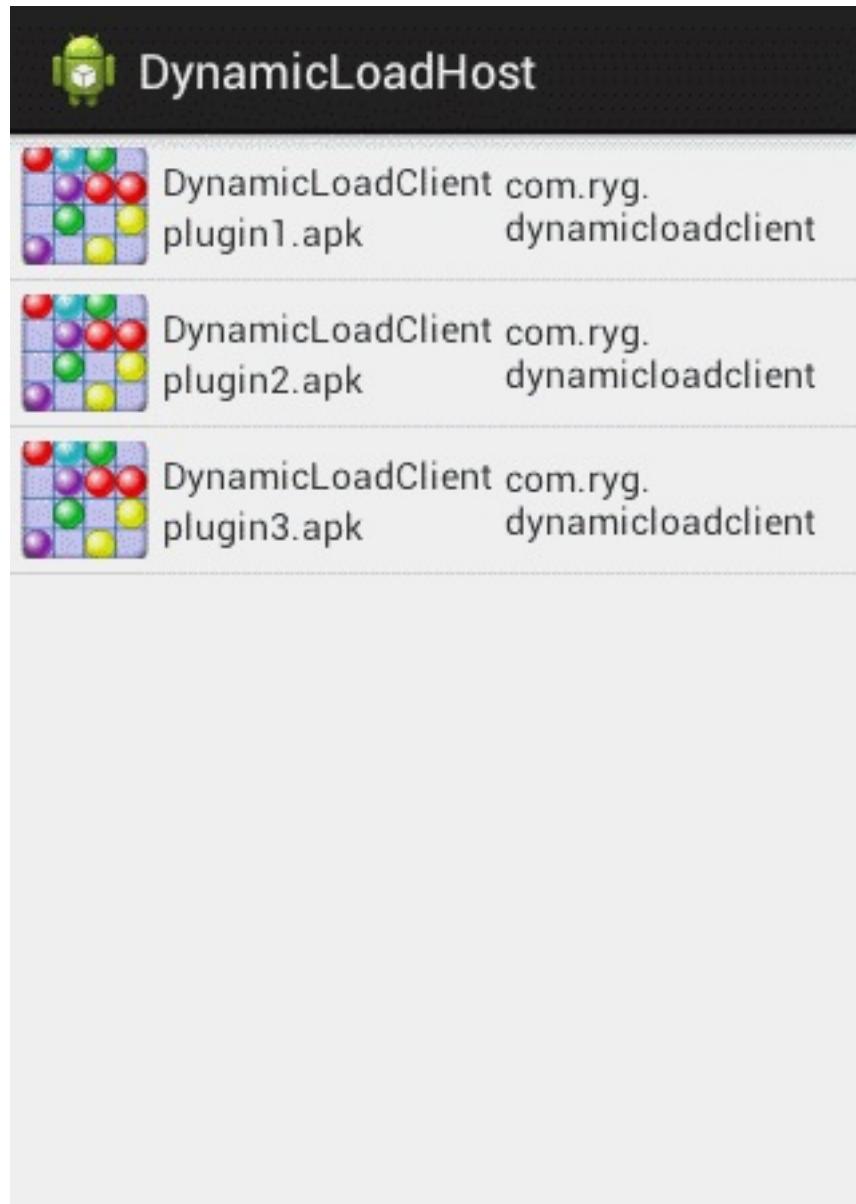
- 5. 目前暂不支持Service、BroadcastReceiver等需要注册才能使用的组件。

## 后续工作

- 1.DLIntent的定义，通过自定义的intent，来完成activity的无约束调起
- 2.采用Fragment的生命周期管理
- 3.Service、BroadcastReceiver等组件的调起
- 4.性能优化

## 效果

首先宿主程序运行后，会把位于 /mnt/sdcard/DynamicLoadHost 目录下的所有apk都加载进来，然后点击列表就可以调起对应的apk，本文中的demo和第一篇文章中的demo看起来差不多，实际是有区别的，区别有两点：activity具有生命周期、加载资源可以用R，具体的代码实现请大家参见源码。



源码下载: <https://github.com/singwhatiwanna/dynamic-load-apk>

# Android插件化的一种实现

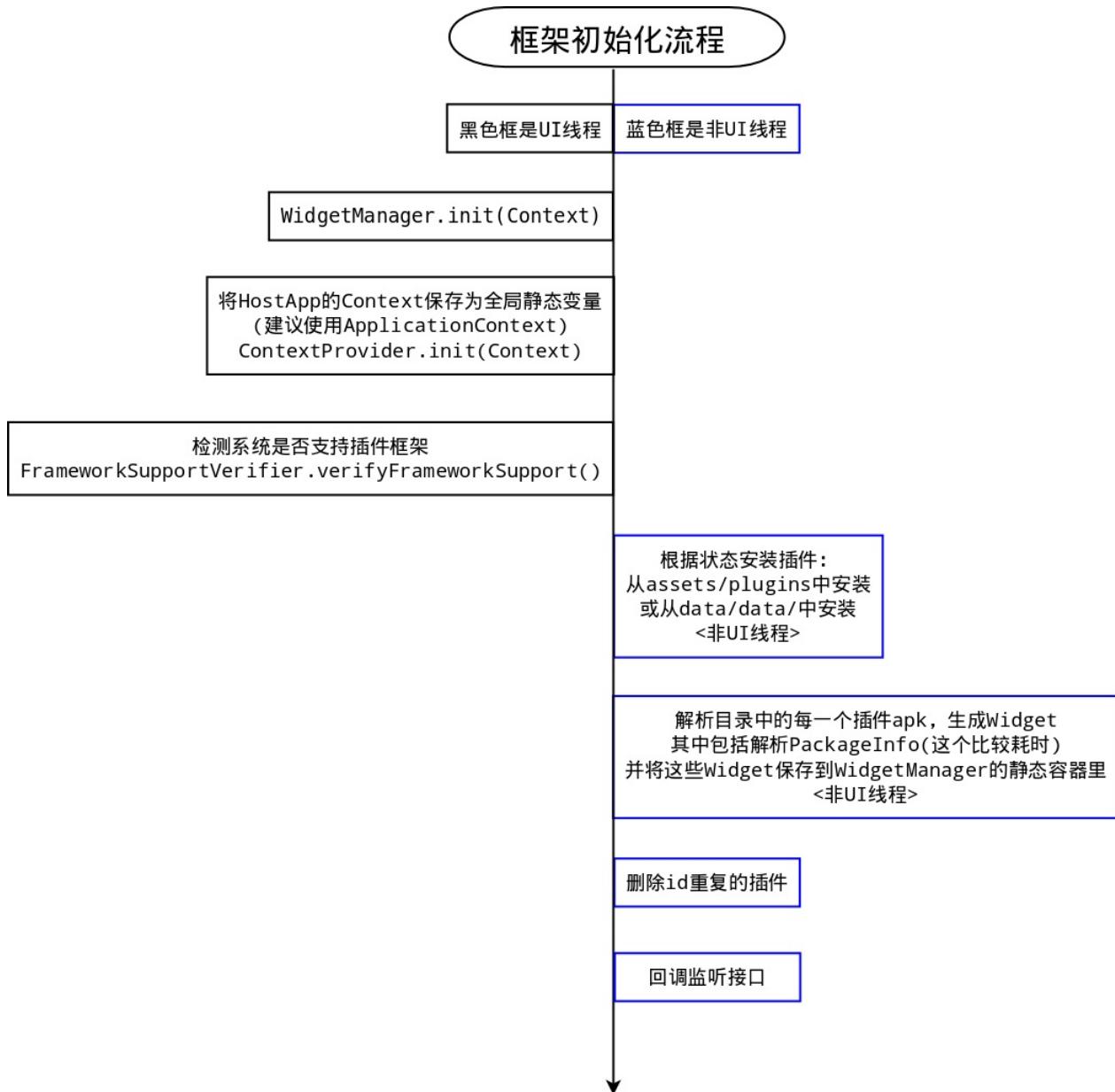
来源:<http://www.cnblogs.com/coding-way/p/4669591.html>

Android的插件化已经是老生常谈的话题了，插件化的好处有很多：解除代码耦合，插件支持热插拔，静默升级，从根本上解决65K属性和方法的bug等等。

下面给大家介绍一下我们正在用的差价化框架。

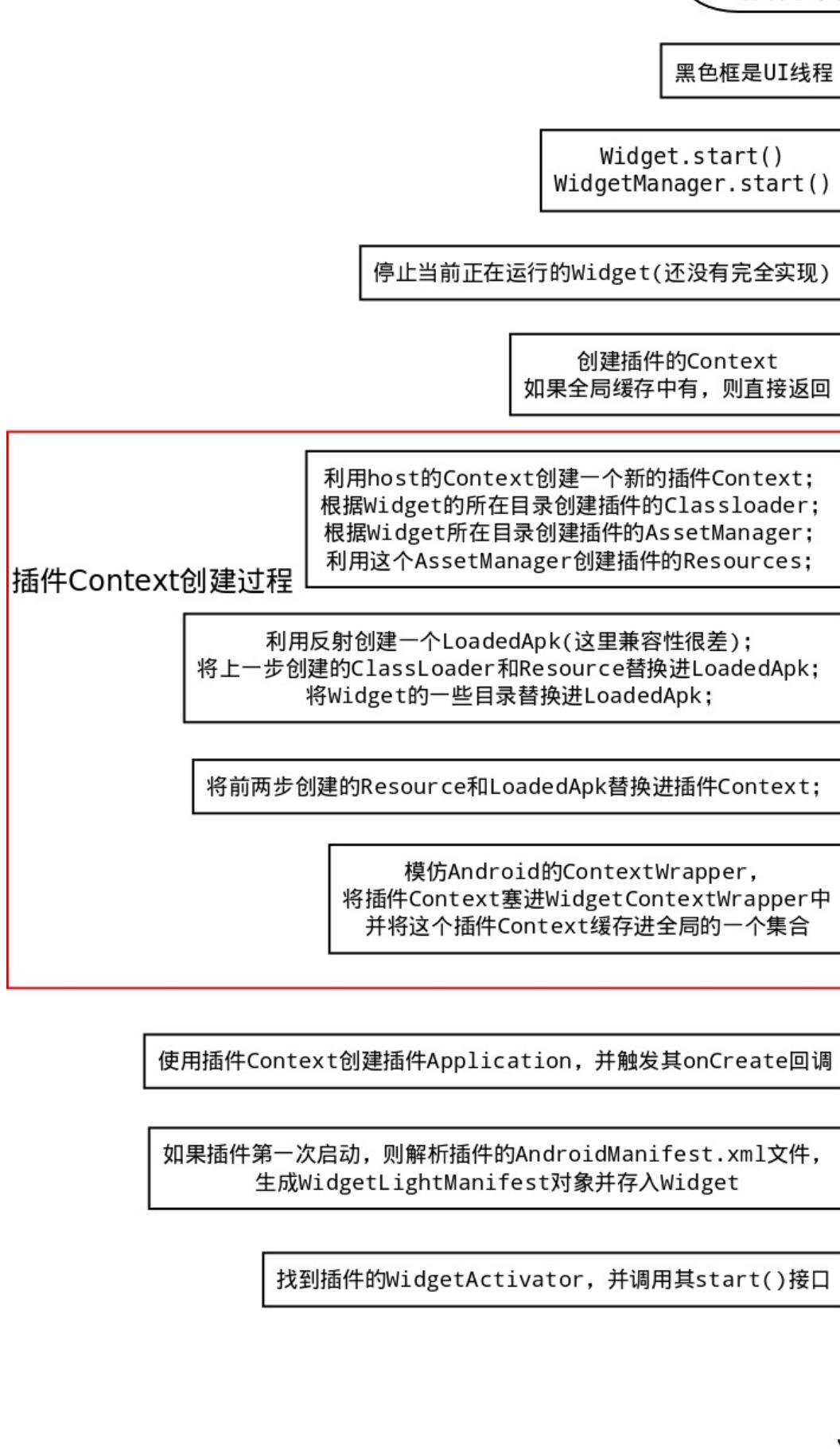
本片主要以类图的方式向大家介绍插件话框架的实现。

下图是框架的初始化流程：



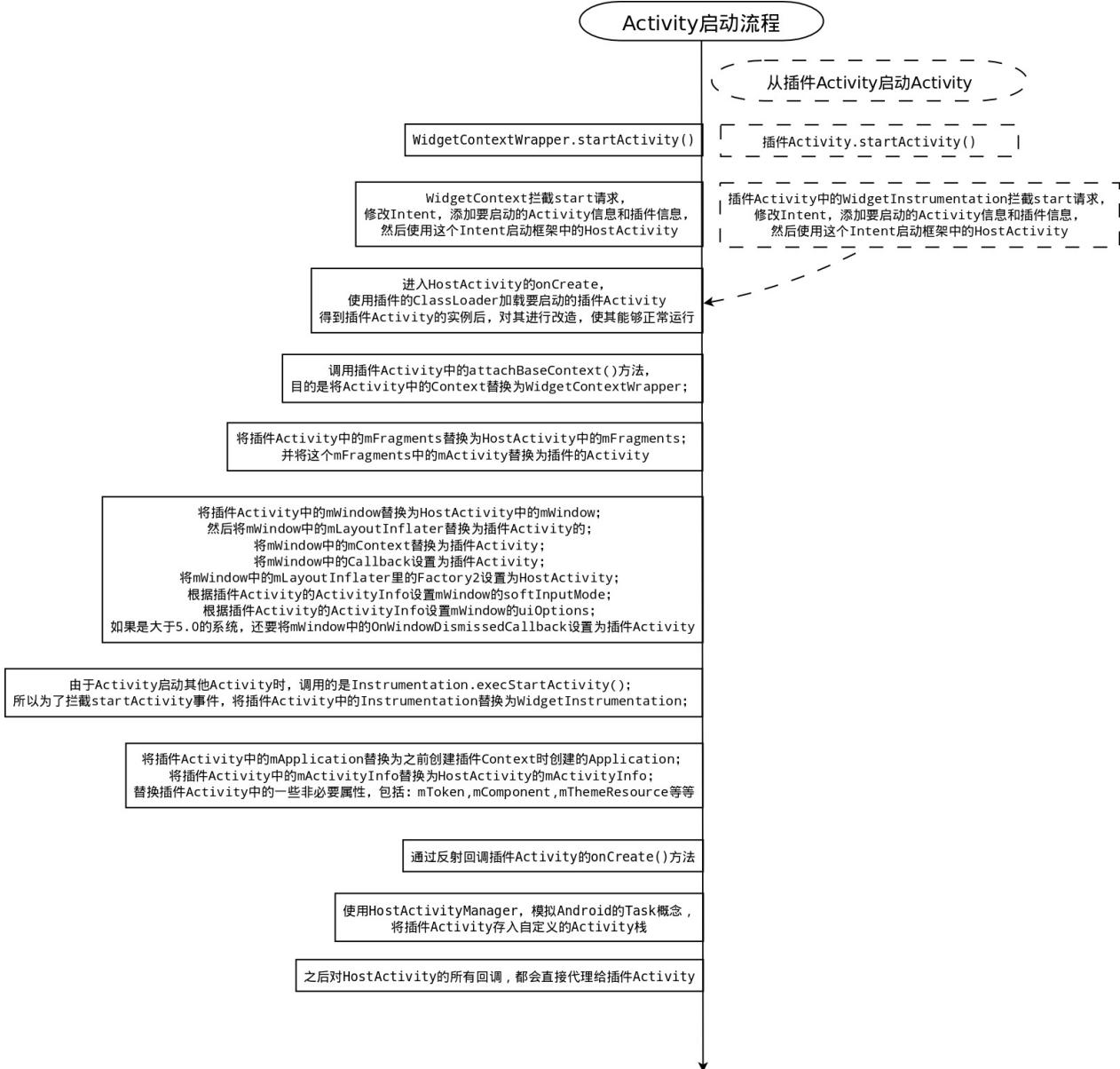
框架初始化后，就该启动插件了，下图是插件的启动流程：

## 插件启动流程



这个步骤主要是初始化插件的运行环境，利用宿主的Context改造成插件的Context。

接下来是插件启动Activity的流程：



该步骤主要是用宿主的Activity包装插件的Activity。

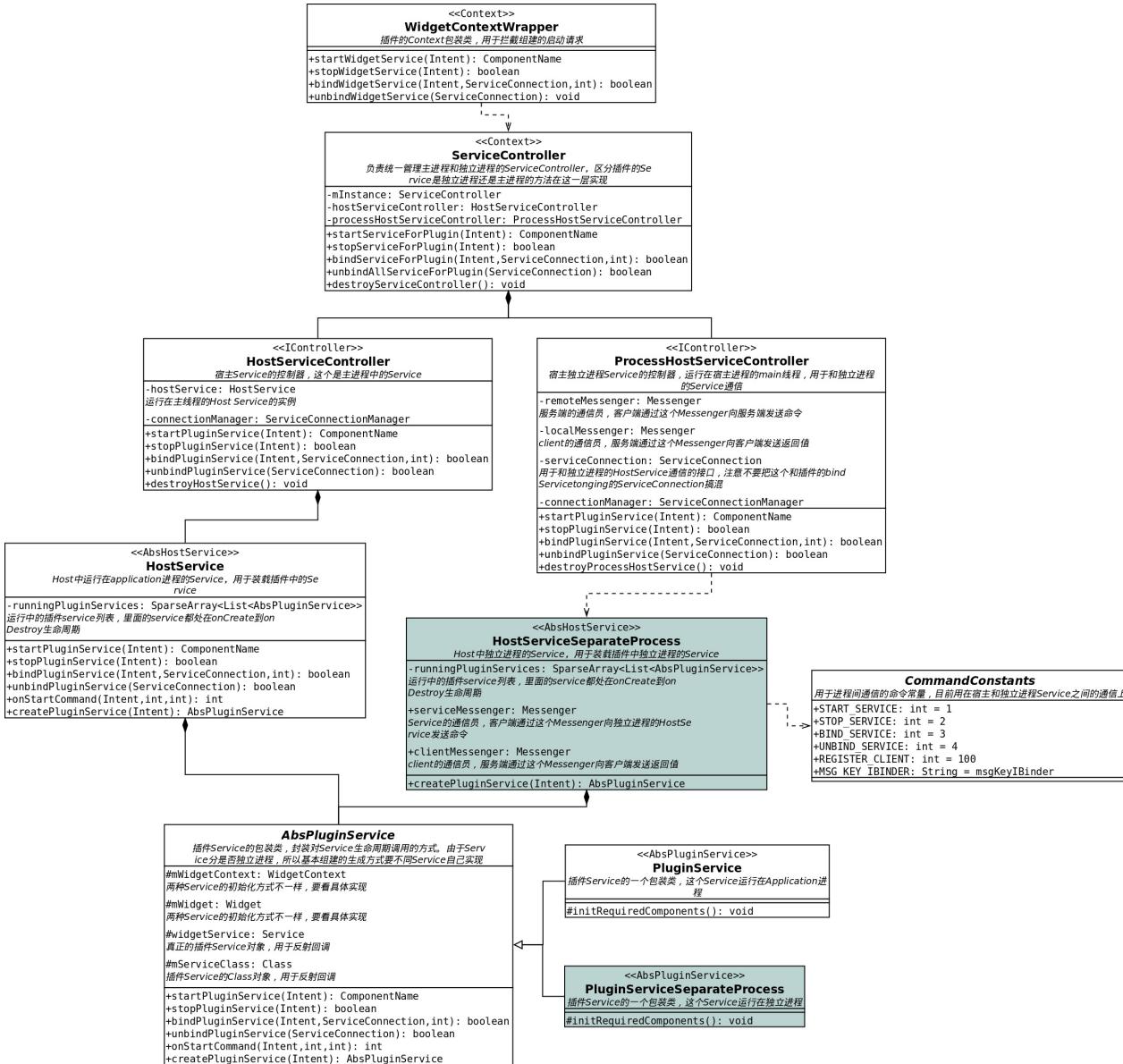
通过上面的几个流程图，我们得知插件框架的基本原理如下：

- 利用DexClassLoader来实现动态加载插件中的class。
- 通过反射替换ContextImpl中的mResources, mPackageManager，并替换插件Activity中的相关属性，来实现加载插件中的资源文件。
- 通过WidgetContext和WidgetInstrumentation来拦截startActivity的操作。
- 通过启动HostActivity来代替插件Activity，也就是说一个HostActivity对应一个插件Activity。

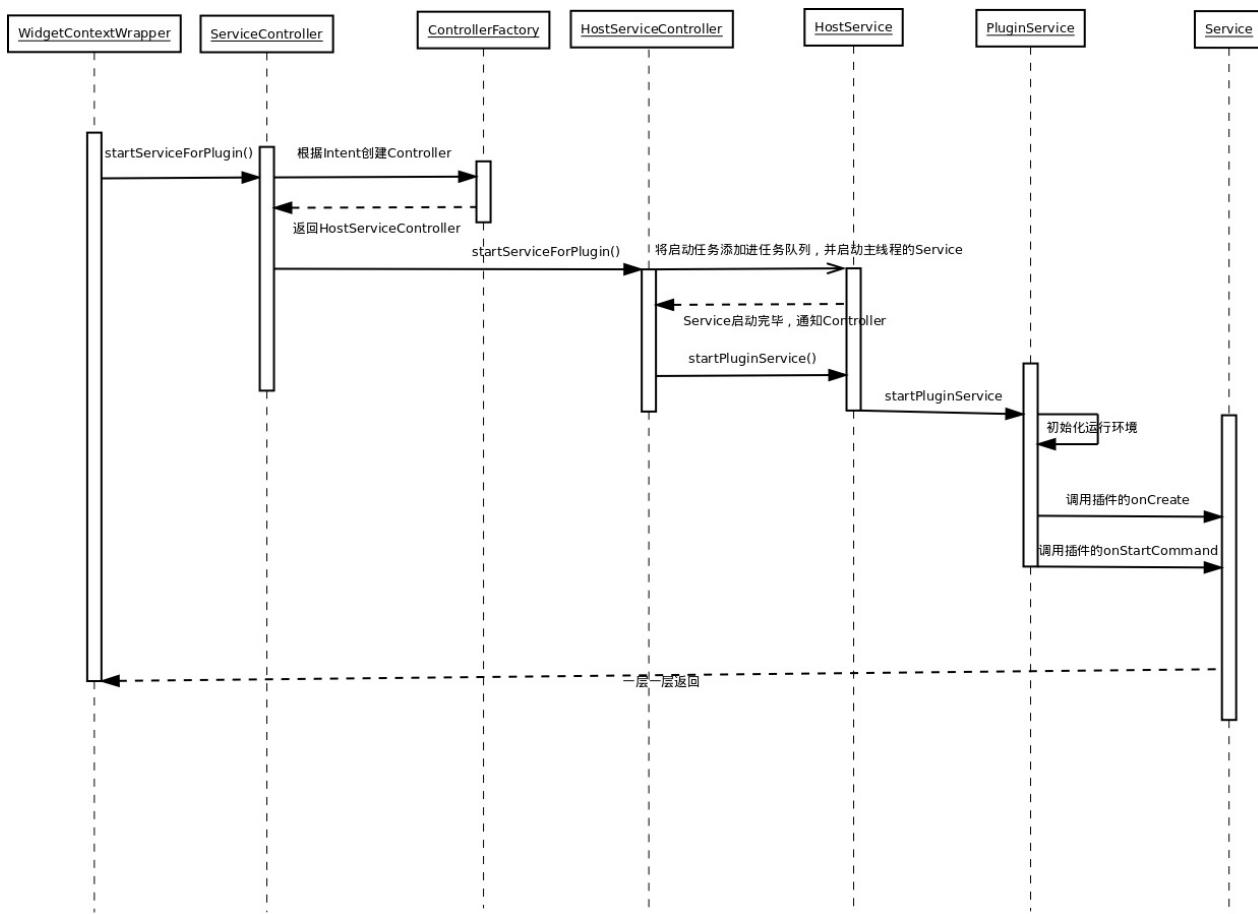
知道了启动插件Activity的原理后，我们思考下如何支持启动插件Service：

- 由于Android系统中同一个Service只会存在一个实例，这点与Activity不同，所以考虑用框架模拟Android系统来维护插件的Service。Service本质上分为两种，运行在独立进程和非独立进程的，所以宿主应用至少需要启动两个Service，用来装载插件的不同Service。

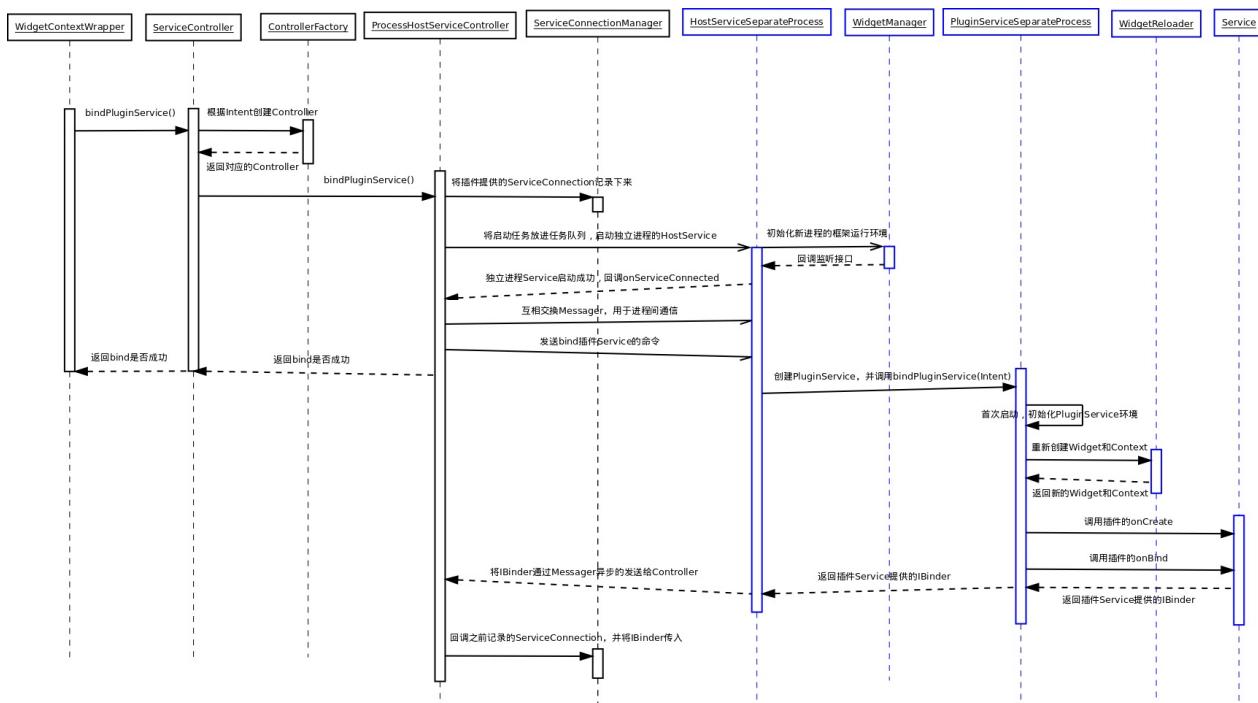
启动Service的实现方式和Activity有些不同，下面是框架管理Service的类图：



了解了框架支持Service的基本结构，我们看下启动插件非独立进程Service的流程：



启动独立进程插件Service要麻烦一些，请看流程图：



这个插件框架还有很多不足，我们还在继续完善。



# APK动态加载框架（DL）解析

来源：<http://blog.csdn.net/singwhatiwanna/article/details/39937639>

转载请注明出处：<http://blog.csdn.net/singwhatiwanna/article/details/39937639> （来自singwhatiwanna的csdn博客）

## 前言

好久没有发布新的文章，这次打算发表一下我这几个月的一个核心研究成果：APK动态加载框架（DL）。这段时间我致力于github的开源贡献，开源了2个比较有用且有意义的项目，一个是PinnedHeaderExpandableListView，另一个是APK动态加载框架。具体可以参见我的github：<https://github.com/singwhatiwanna>

本次要介绍的是APK动态加载框架（DL），这个项目除了我以外，还有两个共同开发者：田啸（时之沙），宋思宇。

为了更好地理解本文，你需要首先阅读Android apk动态加载机制的研究这一系列文章，分别为：

[Android apk动态加载机制的研究](#)

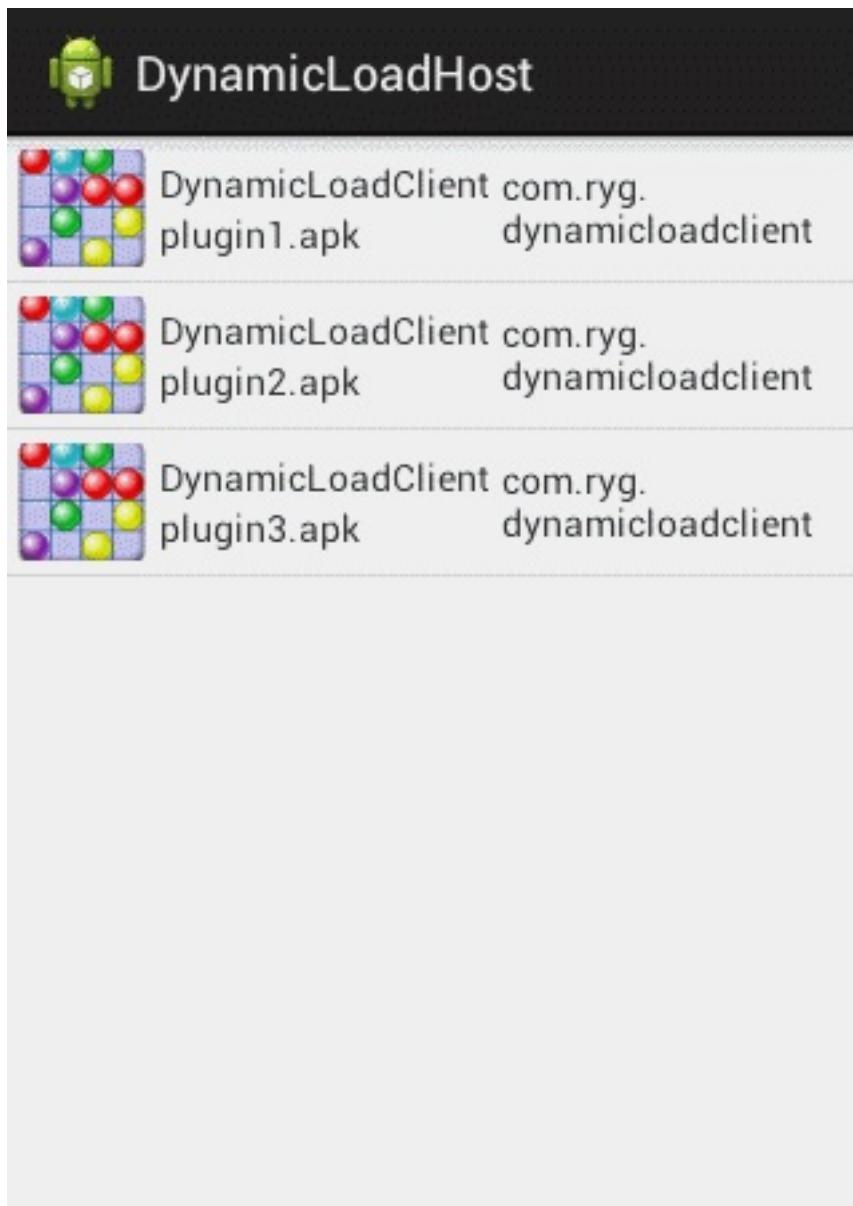
[Android apk动态加载机制的研究（二）：资源加载和activity生命周期管理](#)

另外，这个开源项目我起了个名字，叫做DL。本文中的DL均指APK动态加载框架。

## 项目地址

<https://github.com/singwhatiwanna/dynamic-load-apk>，欢迎star和fork。

运行效果图：



## 意义

这里说说这个开源项目的意义。首先要说的是动态加载技术（或者说插件化）在技术驱动型的公司中扮演这相当重要的角色，当项目越来越庞大的时候，需要通过插件化来减轻应用的内存和cpu占用，还可以实现热插拔，即在不发布新版本的情况下更新某些模块。

我几个月前开始进行这项技术的研究，当时查询了很多资料，没有找到很好的开源。目前淘宝、微信等都有成熟的动态加载框架，包括apkplug，但是它们都是不开源的。还有github上有一个开源项目AndroidDynamicLoader，其思想是通过Fragment以及schema的方式实现的，这是一种可行的技术方案，但是还有限制太多，这意味着你的activity必须通过Fragment去实现，这在activity跳转和灵活性上有一定的不便，在实际的使用中会有一些很奇怪的bug不好解决，总之，这还是一种不是特别完备的动态加载技术。然后，我

发现，目前针对动态加载这一块成熟的开源基本还是空白的，不管是国内还是国外。而在公司内部，动态加载作为一项核心技术，也不可能是在初级开发人员所能够接触到的，于是，我决定做一个成熟点的开源，期待能填补这一块空白。

## DL功能介绍

DL支持很多特性，而这些特性使得插件的开发过程变得透明、高效。

- 1.plugin无需安装即可由宿主调起。
- 2.支持用R访问plugin资源
- 3.plugin支持Activity和FragmentActivity（未来还将支持其他组件）
- 4.基本无反射调用
- 5.插件安装后仍可独立运行从而便于调试
- 6.支持3种plugin对host的调用模式：
  - (1) 无调用（但仍然可以用反射调用）。
  - (2) 部分调用，host可公开部分接口供plugin调用。这前两种模式适用于plugin开发者无法获得host代码的情况。
    - (3) 完全调用，plugin可以完全调用host内容。这种模式适用于plugin开发者能获得host代码的情况。
- 7.只需引入DL的一个jar包即可高效开发插件，DL的工作过程对开发者完全透明
- 8.支持android2.x版本

## 架构解析

如果大家阅读过本文头部提到的两篇文章，那么对DL的架构应该有大致的了解，本文就不再从头开始介绍了，而是从如下变更的几方面进行解析，这些优化使得DL的功能和之前比起来更加强大更加易用使用易于扩展。

- 1.DL对activity生命周期管理的改进
- 2.DL对类加载器的支持（DLClassLoader）
- 3.DL对宿主（host）和插件（plugin）通信的支持
- 4.DL对插件独立运行的支持
- 5.DL对activity随意跳转的支持（DLIntent）
- 6.DL对插件管理的支持（DLPluginManager）

其中5和6属于加强功能，目前正在dev分支上进行开发（本文暂不介绍），其他功能均在稳定版分支master上。

## DL对activity生命周期管理的改进

大家知道，DL最开始的时候采用反射去管理activity的生命周期，这样存在一些不便，比如反射代码写起来复杂，并且过多使用反射有一定的性能开销。针对这个问题，我们采用了接口机制，将activity的大部分生命周期方法提取出来作为一个接口（DLPlugin），然后通过代理activity（DLProxyActivity）去调用插件activity实现的生命周期方法，这样就完成了插件activity的生命周期管理，并且没有采用反射，当我们想增加一个新的生命周期方法的时候，只需要在接口中声明一下同时在代理activity中实现一下即可，下面看一下代码：

### 接口DLPlugin

```
public interface DLPlugin {

 public void onStart();
 public void onRestart();
 public void onActivityResult(int requestCode, int resultCode, Intent data);
 public void onResume();
 public void onPause();
 public void onStop();
 public void onDestroy();
 public void onCreate(Bundle savedInstanceState);
 public void setProxy(Activity proxyActivity, String dexPath);
 public void onSaveInstanceState(Bundle outState);
 public void onNewIntent(Intent intent);
 public void onRestoreInstanceState(Bundle savedInstanceState);
 public boolean onTouchEvent(MotionEvent event);
 public boolean onKeyDown(int keyCode, KeyEvent event);
 public void onWindowAttributesChanged(LayoutParams params);
 public void onWindowFocusChanged(boolean hasFocus);
 public void onBackPressed();
}
```

在代理类DLProxyActivity中的实现

```
...
@Override
protected void onStart() {
 mRemoteActivity.onStart();
 super.onStart();
}

@Override
protected void onRestart() {
 mRemoteActivity.onRestart();
 super.onRestart();
}

@Override
protected void onResume() {
 mRemoteActivity.onResume();
 super.onResume();
}

@Override
protected void onPause() {
 mRemoteActivity.onPause();
 super.onPause();
}

@Override
protected void onStop() {
 mRemoteActivity.onStop();
 super.onStop();
}
...
```

说明：通过上述代码应该不难理解DL对activity生命周期的管理，其中mRemoteActivity就是DLPlugin的实现。

## DL对类加载器的支持

为了更好地对多插件进行支持，我们提供了一个DLClassLoader类，专门去管理各个插件的DexClassLoader，这样，同一个插件就可以采用同一个ClassLoader去加载类从而避免多个classloader加载同一个类时所引发的类型转换错误。

```
public class DLClassLoader extends DexClassLoader {
 private static final String TAG = "DLClassLoader";

 private static final HashMap<String, DLClassLoader> mPluginClassLoaders = new HashMap<String, DLClassLoader>();

 protected DLClassLoader(String dexPath, String optimizedDirectory, String libraryPath, ClassLoader parent) {
 super(dexPath, optimizedDirectory, libraryPath, parent);
 }

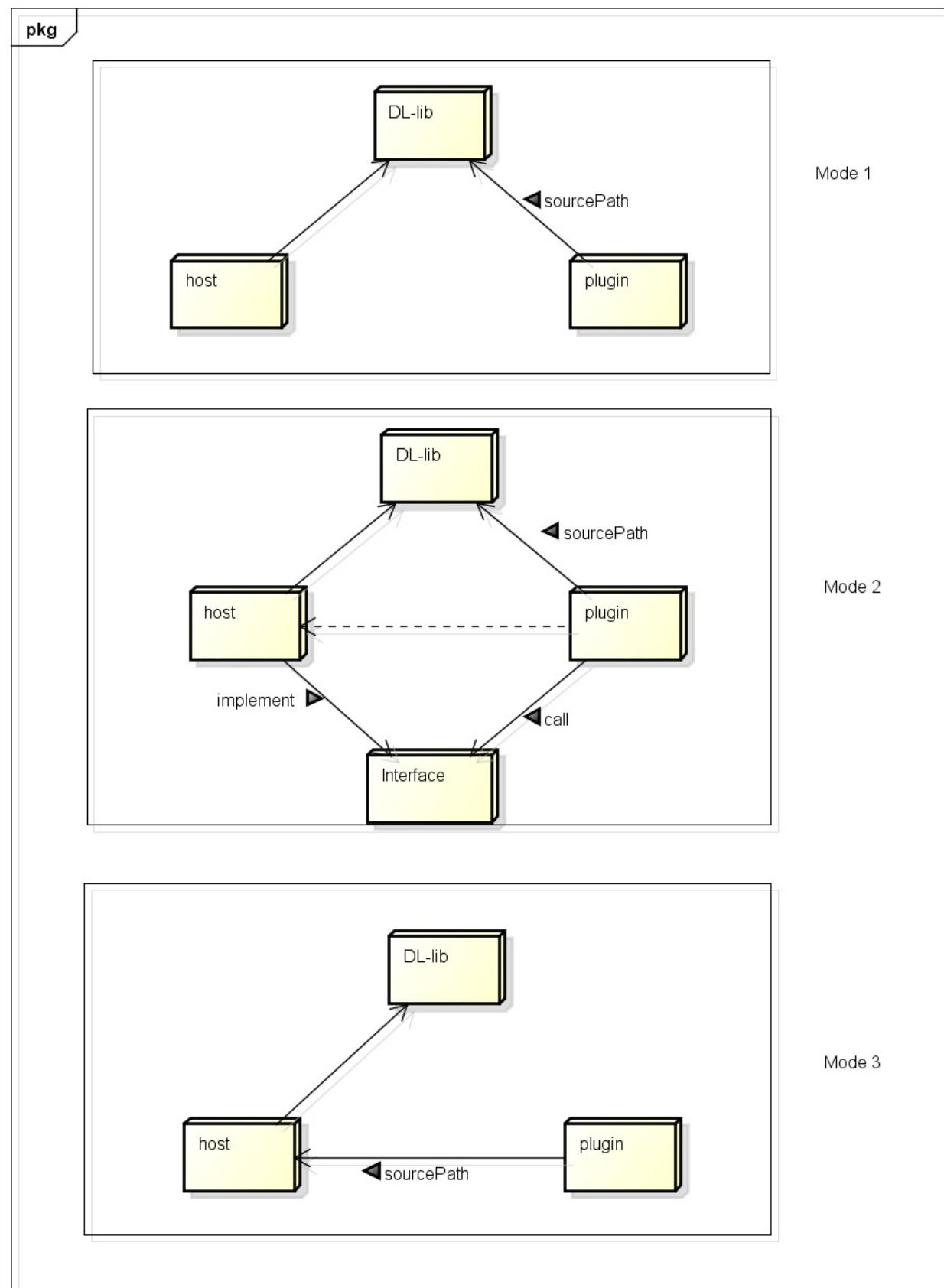
 /**
 * return a available classloader which belongs to different apk
 */
 public static DLClassLoader getClassLoader(String dexPath, Context context, ClassLoader parentLoader) {
 DLClassLoader dLClassLoader = mPluginClassLoaders.get(dexPath);
 if (dLClassLoader != null)
 return dLClassLoader;

 File dexOutputDir = context.getDir("dex", Context.MODE_PRIVATE);
 final String dexOutputPath = dexOutputDir.getAbsolutePath();
 dLClassLoader = new DLClassLoader(dexPath, dexOutputPath, null, parentLoader);
 mPluginClassLoaders.put(dexPath, dLClassLoader);

 return dLClassLoader;
 }
}
```

## DL对宿主（host）和插件（plugin）通信的支持

这一点很重要，因为往往宿主需要和插件进行各种通信，因此DL对宿主和插件的通信做了很好的支持，目前总共有3中模式，如下图所示：



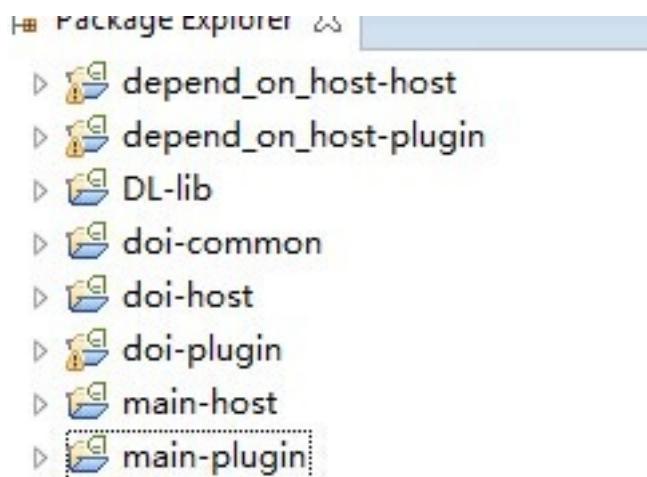
<http://blog.csdn.net/singwhatiwear>  
powered by Astah

下面分别介绍上述三种模式，针对上述三种模式，我们分别提供了3组例子，其中：

**1. depend\_on\_host:** 插件完全依赖宿主的模式，适合于能够到宿主的源代码的情况 其中host指宿主工程， plugin指插件工程

**2. depend\_on\_interface:** 插件部分依赖宿主的模式，或者说插件依赖宿主提供的接口，适合能够拿到宿主的接口的情况,其中host指宿主工程， plugin指插件工程， common指接口工程

**3.main:** 插件不依赖宿主的模式，这是DL推荐的模式 其中host指宿主工程， plugin指插件工程



**模式1：**这是DL推荐的模式，对应的工程目录为main。在这种模式下，宿主和插件不需要通信，两者是独立开发的，宿主引用DL的jar包（dl-lib.jar），插件也需要引用DL的jar包，但是不能放入到插件工程的libs目录下面，换句话说，就是插件编译的时候依赖jar包但是打包成apk的时候不要把jar包打进去，这是因为，dl-lib.jar已经在宿主工程中存在了，如果插件中也有这个jar包，就会发生类链接错误，原因很简单，内存中有两份一样的类，重复了。至于support-v4也是同样的道理。对于eclipse很简单，只需要在插件工程中创建一个目录，比如external-jars，然后把dl-lib.jar和support-v4.jar放进去，同时在.classpath中追加如下两句即可：

```
<classpathentry kind="lib" path="external-jars/dl-lib.jar"/>
<classpathentry kind="lib" path="external-jars/android-support-v4.jar"/>
```

这样，编译的时候就能够正常进行，但是打包的时候，就不会把上面两个jar包打入到插件apk中。

至于ant环境和gradle，解决办法不一样，具体方法后面再补上，但是思想都是一样的，即：插件apk中不要打入上述2个jar包。

**模式2：**插件部分依赖宿主的模式，或者说插件依赖宿主提供的接口，适合能够拿到宿主的接口的情况。在这种模式下，宿主放出一些接口并实现一些接口，然后给插件调用，这样插件就可以访问宿主的一些服务等

**模式3：**插件完全依赖宿主的模式，适合于能够到宿主的源代码的情况。这种模式一般多用在公司内部，插件可以访问宿主的所有代码，但是，这样插件和宿主的耦合比较高，宿主一动，插件就必须动，比较麻烦

具体采用哪种方式，需要结合实际情况来选择，一般来说，如果是宿主和插件不是同一个公司开发，建议选择模式1和模式2；如果宿主和插件都在同一个公司开发，那么选择哪个都可以。从DL的实现出发，我们推荐采用模式1，真的需要通信的话采用模式2，尽量不要采用模式3.

## DL对插件独立运行的支持

为了便于调试，采用DL所开发的插件都可以独立运行，当然，这要分情况来说：

对于模式1，如果插件想独立运行，只需要把external-jars下的jar包拷贝一份到插件的libs目录下即可

对于模式2，只需要提供一个宿主接口的默认实现即可

对于模式3，只需要apk打包时把所引用的宿主代码打包进去即可，具体方式可以参看sample/depend\_on\_host目录。

在开发过程中，应该先开启插件的独立运行功能以便于调试，等功能开发完毕后再将其插件化。

## DLIntent和DLPluginManager

这两项都属于加强功能，目前正在dev分支进行code review，大家感兴趣可以去dev分支上查看，等验证通过即merge到稳定版master分支。

DLIntent：通过DLIntent来完成activity的无约束调起

DLPluginManager：对宿主的所有插件提供综合管理功能。

## 开发规范

目前DL已经达到了第一个稳定版，经过大量机型的验证，目前得出的结论是DL是可靠的（兼容到android2.x），可以用在实际的应用开发中。但是，我们知道，动态加载是一个技术壁垒，其很难达到一种完美的状态，毕竟，让一个apk不安装跑起来，这是多么不可思议的事情。因此，希望大家辩证地去看这个问题，下面列出我们目前还不支持的功能，或者说是一种开发规范吧，希望大家在开发过程中去遵守这个规范，这样才能让插件稳定地跑起来。

## DL 1.0开发规范：

- 1.目前不支持service
- 2.目前只支持动态注册广播
- 3.目前支持Activity和FragmentActivity，这也是常用的activity
- 4.目前不支持插件中的assets
- 5.调用Context的时候，请适当使用that，大部分常用api是不需要用that的，但是一些不常用api还是需要用that来访问。that是apk中activity的基类 BaseActivity 系列中的一个成员，它在apk安装运行的时候指向this，而在未安装的时候指向宿主程序中的代理 activity，由于that的动态分配特性，通过that去调用activity的成员方法，在apk安装以后仍然可以正常运行。
- 6.慎重使用this，因为this指向的是当前对象，即apk中的activity，但是由于activity已经不是常规意义上的activity，所以this是没有意义的，但是，当this表示的不是 Context对象的时候除外，比如this表示一个由activity实现的接口。

希望能够给大家带来一些帮助，希望大家多多支持！

本开源项目地址：<https://github.com/singwhatiwanna/dynamic-load-apk>，欢迎大家star和fork。

# DynamicLoadApk 源码解析

---

=====

本文为 [Android 开源项目源码解析](#) 中 DynamicLoadApk 部分

项目地址: [DynamicLoadApk](#), 分析的版本: [144571b](#), Demo 地址: [DynamicLoadApk Demo](#)

分析者: [FFish](#), 分析状态: 完成, 校对者: [Trinea](#), 校对状态: 初审完成

## 1. 功能介绍

### 1.1 简介

DynamicLoadApk 是一个开源的 Android 插件化框架。

插件化的优点包括: (1) 模块解耦, (2) 动态升级, (3) 高效并行开发(编译速度更快) (4) 按需加载, 内存占用更低等等。

DynamicLoadApk 提供了 3 种开发方式, 让开发者在无需理解其工作原理的情况下快速的集成插件化功能。

1. 宿主程序与插件完全独立
2. 宿主程序开放部分接口供插件与之通信
3. 宿主程序耦合插件的部分业务逻辑

三种开发模式都可以在 demo 中看到。

### 1.2 核心概念

**(1) 宿主:** 主 App, 可以加载插件, 也称 Host。

**(2) 插件:** 插件 App, 被宿主加载的 App, 也称 Plugin, 可以是跟普通 App 一样的 Apk 文件。

**(3) 组件:** 指 Android 中

的 Activity 、 Service 、 BroadcastReceiver 、 ContentProvider , 目前 DL 支持 Activity 、 Service 以及动态的 BroadcastReceiver 。

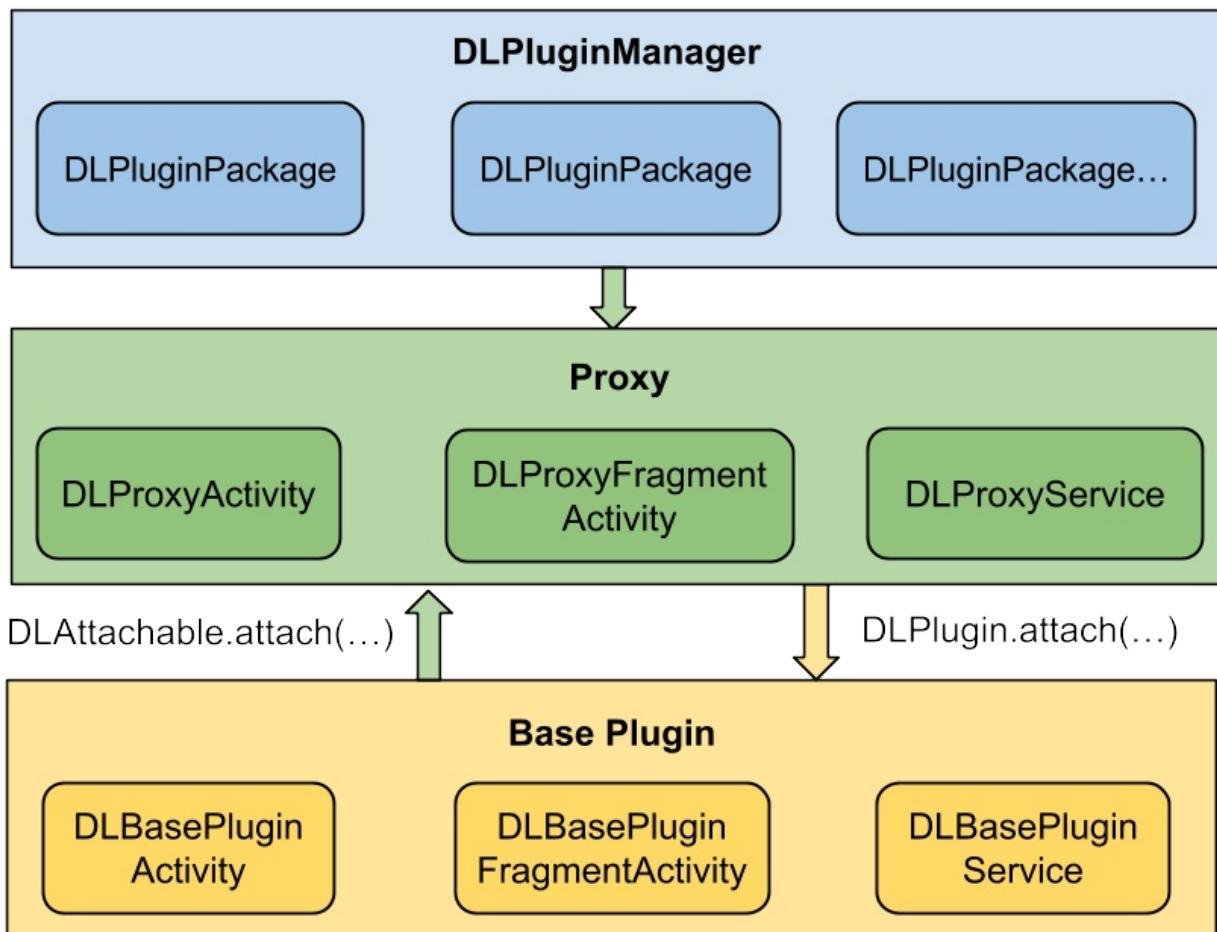
**(4) 插件组件:** 插件中的组件。

**(5) 代理组件：**在宿主的 Manifest 中注册，启动插件组件时首先被启动的组件。目前包括 DLProxyActivity(代理 Activity)、DLProxyFragmentActivity(代理 FragmentActivity)、DLProxyService(代理 Service)。

**(6) Base 组件：**插件组件的基类，目前包括 DLBasePluginActivity(插件 Activity 的基类)、DLBasePluginFragmentActivity(插件 FragmentActivity 的基类)、DLBasePluginService(插件 Service 的基类)。

DynamicLoadApk 原理的核心思想可以总结为两个字：代理。通过在 Manifest 中注册代理组件，当启动插件组件时首先启动一个代理组件，然后通过这个代理组件来构建、启动插件组件。

## 2. 总体设计



上面是 DynamicLoadApk 的总体设计图，DynamicLoadApk 主要分为四大模块：

### (1) DLPluginManager

插件管理模块，负责插件的加载、管理以及启动插件组件。

### (2) Proxy

代理组件模块，目前包括 DLProxyActivity(代理 Activity)、DLProxyFragmentActivity(代理 FragmentActivity)、DLProxyService(代理 Service)。

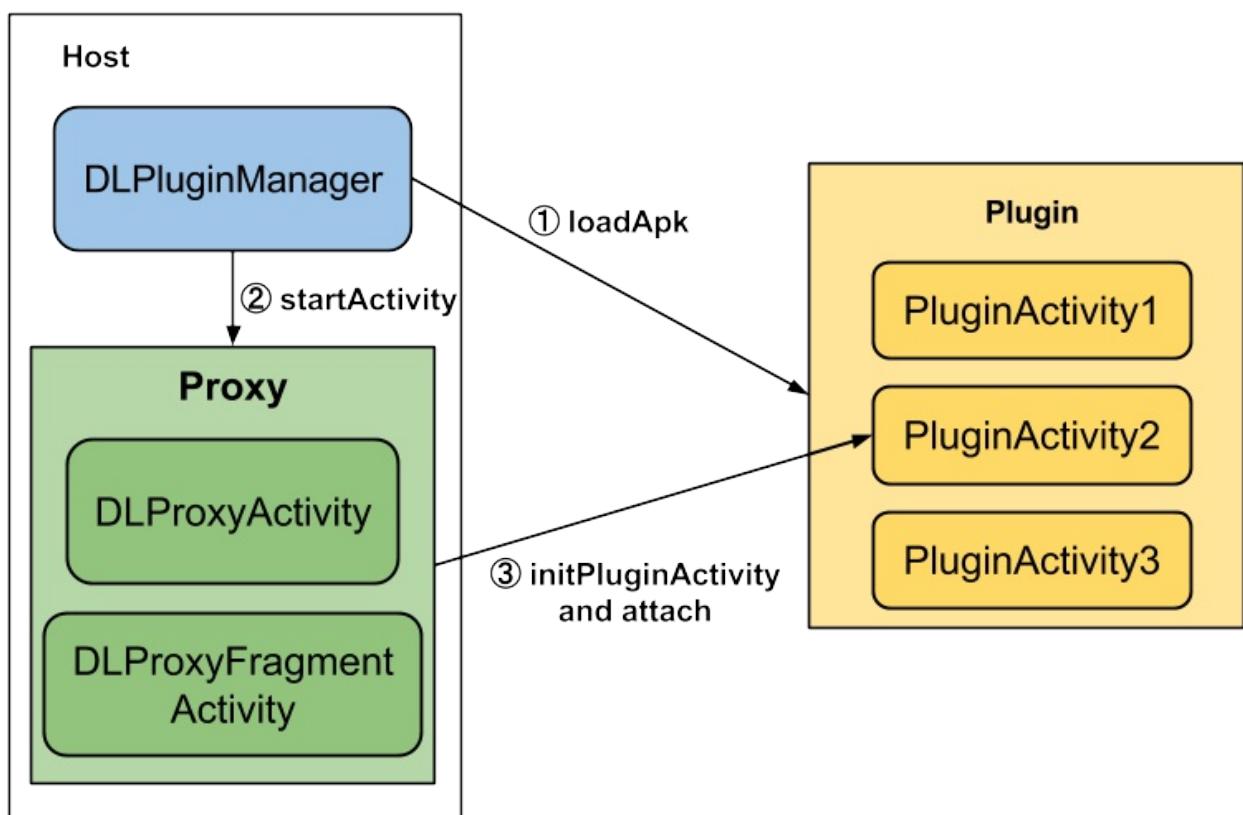
### (3) Proxy Impl

代理组件公用逻辑模块，与(2)中的 Proxy 不同的是，这部分并不是一个组件，而是负责构建、加载插件组件的管理器。这些 Proxy Impl 通过反射得到插件组件，然后将插件与 Proxy 组件建立关联，最后调用插件组件的 onCreate 函数进行启动。

### (4) Base Plugin

插件组件的基类模块，目前包括 DLBasePluginActivity(插件 Activity 的基类)、DLBasePluginFragmentActivity(插件 FragmentActivity 的基类)、DLBasePluginService(插件 Service 的基类)。

## 3. 流程图

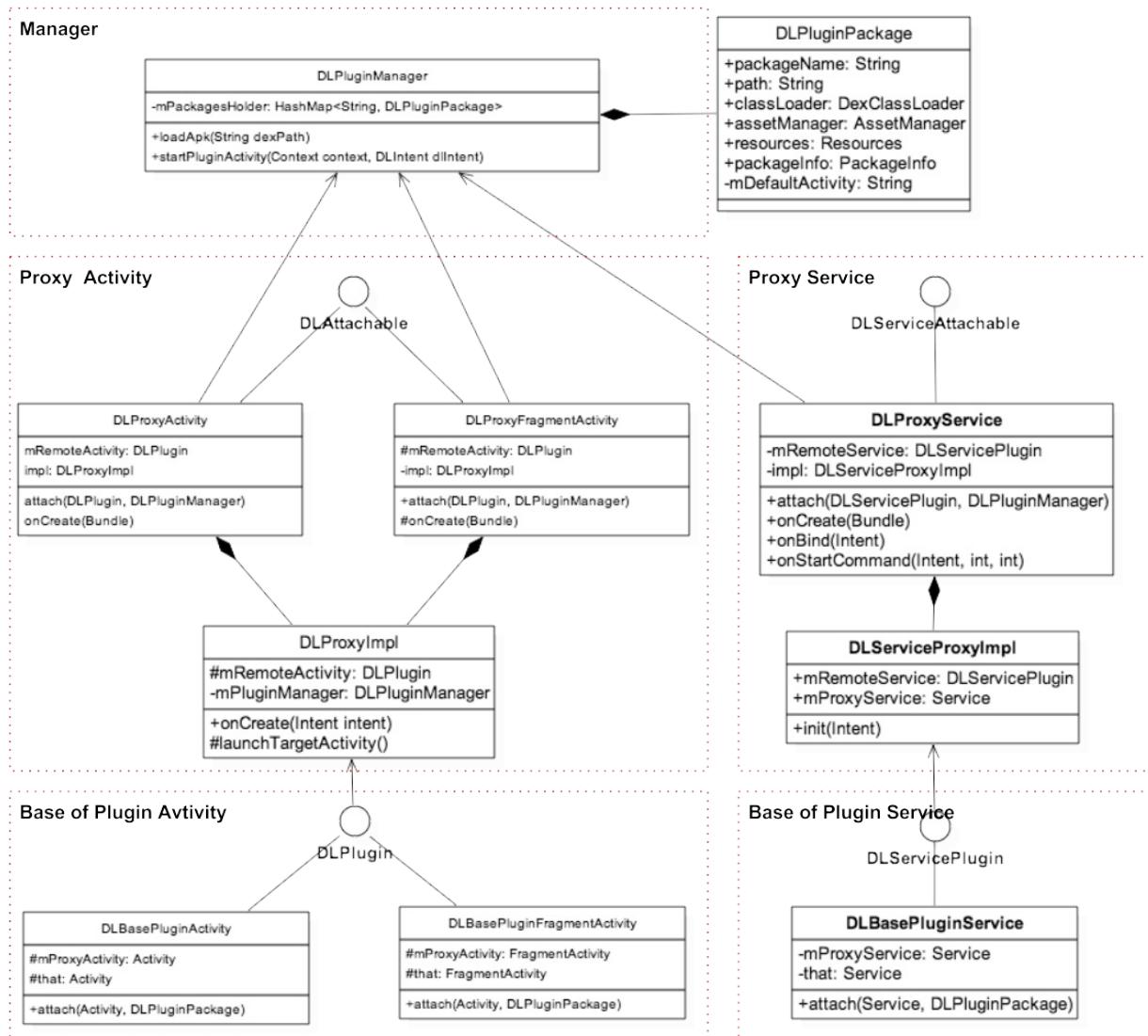


上面是调用插件 Activity 的流程图，其他组件调用流程类似。

- (1) 首先通过 DLPluginManager 的 loadApk 函数加载插件，这步每个插件只需调用一次。
- (2) 通过 DLPluginManager 的 startPluginActivity 函数启动代理 Activity。
- (3) 代理 Activity 启动过程中构建、启动插件 Activity。

## 4. 详细设计

### 4.1 类关系图



以上是 DynamicLoadApk 主要类的关系图，跟总体设计中介绍的一样大致分为三部分。

- (1) 对于 Proxy 部分，每个组件都存在 `DLAttachable` 接口，方便统一该组件不同类，如 `Activity`、`FragmentActivity`。每个组件的公共实现部分都统一放到了对应的 `DLProxyImpl` 中。
- (2) 对于 Base Plugin 部分，每个组件都存在 `DLPlugin` 接口，同样是方便统一该组件不同类。

## 4.2 类功能介绍

### 4.2.1 DLPluginManager.java

DynamicLoadApk 框架的核心类，主要功能包括：

- (1) 插件的加载和管理；
- (2) 启动插件的组件，目前包括 `Activity`、`Service`。

#### 主要属性：

`mNativeLibDir` 为插件 Native Library 拷贝到宿主中后的存放目录路径。

`mPackagesHolder` `HashMap`, `key` 为包名, `value` 为表示插件信息的 `DLPluginPackage` , 存储已经加载过的插件信息。

**主要函数:**

**(1) getInstance(Context context)**

获取 `DLPluginManager` 对象的单例。

在私有构造函数中将 `mNativeLibDir` 变量赋值为宿主 App 应用程序数据目录下名为 `pluginlib` 子目录的全路径。

**(2) loadApk(String dexPath)**

加载插件。参数 `dexPath` 为插件的文件路径。

这个函数直接调用 `loadApk(final String dexPath, boolean hasSoLib)`。

**(3) loadApk(final String dexPath, boolean hasSoLib)**

加载插件 Apk。参数 `dexPath` 为插件的文件路径, `hasSoLib` 表示插件是否含有 so 库。

**注意:** 在启动插件的组件前, 必须先调用上面两个函数之一加载插件, 并且只能在宿主中调用。

流程图如下:

![(5/image/load-apk-flow-chart.png)]

`loadApk` 函数调用 `preparePluginEnv` 函数加载插件, 图中虚线框为 `preparePluginEnv` 的流程图。

**(4) preparePluginEnv(PackageInfo packageInfo, String dexPath)**

加载插件及其资源。流程图如上图。

调用 `createDexClassLoader(...)`、`createAssetManager(...)`、`createResources(...)` 函数完成相应初始化部分。

**(5) createDexClassLoader(String dexPath)**

利用 `DexClassLoader` 加载插件, `DexClassLoader` 初始化函数如下:

```
public DexClassLoader (String dexPath, String optimizedDirectory, String libraryPath,
```

其中 `dexPath` 为插件的路径。

`optimizedDirectory` 优化后的 `dex` 存放路径。这里将路径设置为当前 App 应用程序数据目录下名为 `dex` 的子目录中。

`libraryPath` 为 Native Library 存放的路径。这里将路径设置为 `mNativeLibDir` 属性, 其

在 `getInstance(Context)` 函数中已经初始化。

`parent` 父 `ClassLoader`, `ClassLoader` 采用双亲委托模式查找类, 具体加载方式可见 [ClassLoader 基础](#)。

### (6) `createAssetManager(String dexPath)`

创建 `AssetManager`, 加载插件资源。

在 Android 中, 资源是通过 `R.java` 中的 `id` 来调用访问的。但是实现插件化之后, 宿主是无法通过 `R` 文件访问插件的资源, 所以这里使用反射来生成属于插件的 `AssetManager`, 并利用 `addAssetPath` 函数加载插件资源。

```
private AssetManager createAssetManager(String dexPath) {
 try {
 AssetManager assetManager = AssetManager.class.newInstance();
 Method addAssetPath = assetManager.getClass().getMethod("addAssetPath", S
 addAssetPath.invoke(assetManager, dexPath);
 return assetManager;
 } catch (Exception e) {
 e.printStackTrace();
 return null;
 }
}
```

`AssetManager` 的无参构造函数以及 `addAssetPath` 函数都被 `hide` 了, 通过反射调用。

### (7) `createResources(AssetManager assetManager)`

利用 `AssetManager` 中已经加载的资源创建 `Resources`, 代理组件中会从这个 `Resources` 中读取资源。

关于 `AssetManager`、`Resources` 深入的信息可参考: [Android 应用程序资源的查找过程分析](#)。

### (8) `copySoLib(String dexPath)`

调用 `SoLibManager` 拷贝 so 库到 Native Library 目录。

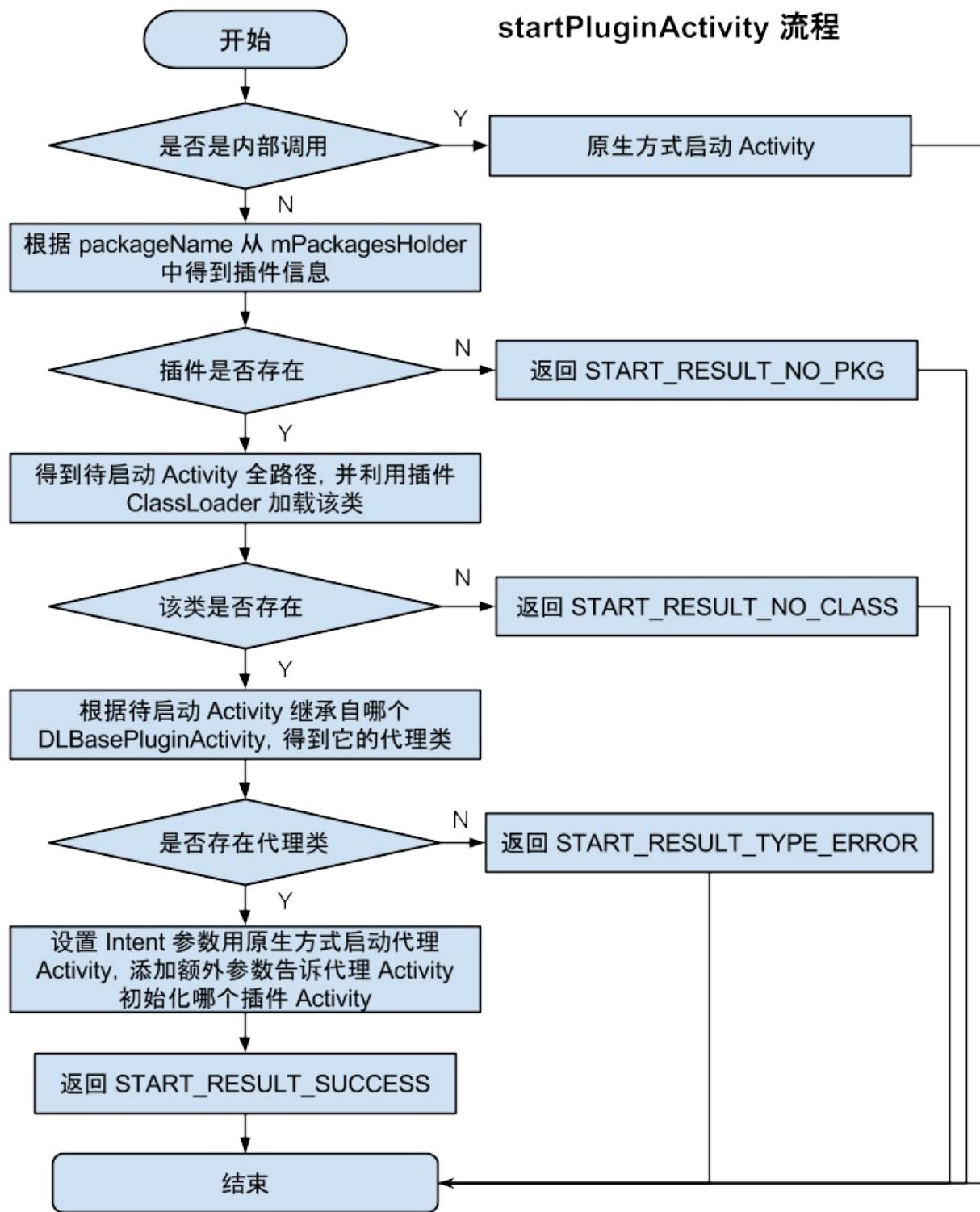
### (9) `startPluginActivity(Context context, DLIntent dlIntent)`

启动插件 Activity, 会直接调用 `startPluginActivityForResult(...)` 函数。

插件自己内部 Activity 启动依然是调用 `Context#startActivity(...)` 方法。

### (10) `startPluginActivityForResult(Context context, DLIntent dlIntent, int requestCode)`

启动插件 Activity, 流程图如下:



### (11) startPluginService(final Context context, final DLIntent dlIntent)

启动插件 Service。

主要逻辑在函数 `fetchProxyServiceClass(...)` 中，流程与 `startPluginActivity(...)` 类似，只是换成了回调的方式，在各种条件成立后调用原生方式启动代理 Service，不再赘述。

### (12) bindPluginService(...) unBindPluginService(...)

`bind` 或是 `unBind` 插件 Service。逻辑与 `startPluginService(...)` 类似，不再赘述。

#### 4.2.2 DLPluginPackage

插件信息对应的实体类，主要属性如下：

```
public String packageName;
public String defaultActivity;
public DexClassLoader classLoader;
public AssetManager assetManager;
public Resources resources;
public PackageInfo packageInfo;
```

`packageName` 为插件的包名；

`defaultActivity` 为插件的 Launcher Main Activity；

`classLoader` 为加载插件的 ClassLoader；

`assetManager` 为加载插件资源的 AssetManager；

`resources` 利用 `assetManager` 中已经加载的资源创建的 `Resources`，代理组件中会从这个 `Resources` 中读取资源。

`packageInfo` 被 `PackageManager` 解析后的插件信息。

这些信息都会在 `DLPluginManager#loadApk(...)` 时初始化。

#### 4.2.3 DLAttachable.java/DLServiceAttachable.java

`DLServiceAttachable` 与 `DLAttachable` 类似，下面先分析 `DLAttachable.java`。

`DLAttachable` 是一个接口，主要作用是以统一所有不同类型的代理 Activity，如 `DLProxyActivity`、`DLProxyFragmentActivity`，方便作为同一接口统一处理。

`DLProxyActivity` 和 `DLProxyFragmentActivity` 都实现了这个类。

`DLAttachable` 目前只有一个

```
attach(DLPlugin pluginActivity, DLPluginManager pluginManager)
```

抽象函数，表示将插件 `Activity` 和代理 `Activity` 绑定在一起，其中的 `pluginActivity` 参数就是指插件 `Activity`。

同样 `DLServiceAttachable` 类似，作用是统一所有不同类型的代理 Service，实现插件 `Service` 和代理 `Service` 的绑定。虽然目前只有 `DLProxyService`。

#### 4.2.4 DLPlugin.java/DLServicePlugin.java

DLPlugin 与 DLServicePlugin 类似，下面先分析 DLPlugin.java。

DLPlugin 是一个接口，包含 Activity 生命周期、触摸、菜单等抽象函数。

DLBase\*Activity 都实现了这个类，这样插件的 Activity 间接实现了此类。

主要作用是统一所有不同类型的插件 Activity，如 Activity 、 FragmentActivity ，方便作为同一接口统一处理，所以这个类叫 DLPluginActivity 更合适。

同样 DLServicePlugin 主要作用是统一所有不同类型的插件 Service，方便作为统一接口统一处理，目前包含 Service 生命周期等抽象函数。

#### 4.2.5 DLProxyActivity.java/DLProxyFragmentActivity.java

代理 Activity，他们是在宿主 Manifest 中注册的组件，也是启动插件 Activity 时，真正被启动的 Activity，他们的内部会完成插件 Activity 的初始化和启动。

这两个类大同小异，所以这里只分析 DLProxyActivity 。

首先来看下它的成员变量。

##### (1). DLPlugin mRemoteActivity

表示真正需要启动的插件 Activity 。这个属性名应该叫做 pluginActivity 更合适。

上面我们已经介绍了， DLPlugin 是所有插件 Activity 都间接实现了的接口。

接下来在代理 Activity 的生命周期、触摸、菜单等函数中我们都会同时调用 mRemoteActivity 的相关函数，模拟插件 Activity 的相关功能。

##### (2). DLProxyImpl impl

主要封装了插件 Activity 的公用逻辑，如初始化插件 Activity 并和代理 Activity 绑定、获取资源等。

#### 4.2.6 DLProxyImpl.java/DLServiceProxyImpl.java

DLProxyImpl 与 DLServiceProxyImpl 类似，下面先分析 DLProxyImpl.java。

DLProxyImpl 主要封装了插件 Activity 的公用逻辑，如初始化插件 Activity 并和代理 Activity 绑定、获取资源等，相当于把 DLProxyActivity 和 DLProxyFragmentActivity 的公共实现部分提出出来，核心逻辑位于下面介绍的 onCreate() 函数。

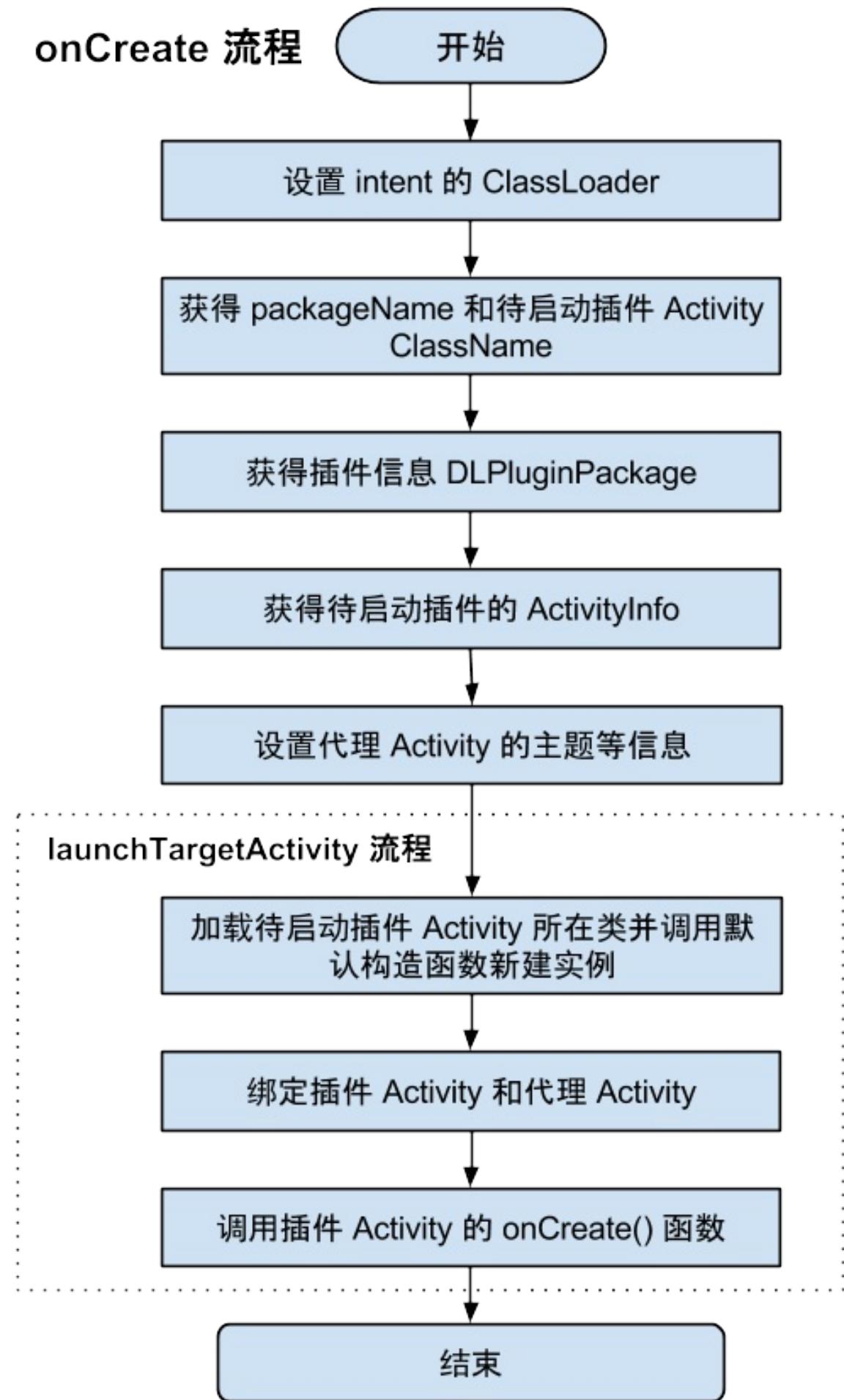
主要函数：

##### (1) DLProxyImpl(Activity activity)

构造函数，参数为代理 Activity。

## (2) **public void onCreate(Intent intent)**

onCreate 函数，会在代理 Activity onCreate 函数中被调用，流程图如下：



其中第一步 设置 `intent` 的 `ClassLoader` 是用于 unparcel `Parcelable` 数据的，可见介绍：[android.os.BadParcelableException](#)。

### (3) `protected void launchTargetActivity()`

加载待启动插件 `Activity` 完成初始化流程，并通过 `DLPlugin` 和 `DLAAttachable` 接口的 `attach` 函数实现和代理 `Activity` 的双向绑定。流程图见上图虚线框部分。

### (4) `private void initializeActivityInfo()`

获得待启动插件的 `ActivityInfo`。

### (5) `private void handleActivityInfo()`

设置代理 `Activity` 的主题等信息。

其他的 `get*` 函数都是获取一些插件相关信息，会被代理 `Activity` 调用。

同样 `DLServiceProxyImpl` 主要封装了插件 `service` 的公用逻辑，如初始化插件 `Service` 并和代理 `Activity` 绑定。

## 4.2.7 `DLBasePluginActivity.java/DLBasePluginFragmentActivity.java`

插件 `Activity` 基类，插件中的 `Activity` 都要继承

`DLBasePluginActivity/DLBasePluginFragmentActivity` 之一(目前尚不支持 `ActionBarActivity`)。

主要作用是根据是否被代理，确定一些函数直接走父类逻辑还是代理 `Activity` 或是空逻辑。

`DLBasePluginActivity` 继承自 `Activity`，同时实现了 `DLPlugin` 接口。这两个类大同小异，所以这里只分析 `DLBasePluginActivity`。

主要变量：

```
protected Activity mProxyActivity;
protected Activity that;
protected DLPluginManager mPluginManager;
protected DLPluginPackage mPluginPackage;
```

`mProxyActivity` 为代理 `Activity`，通过 `attach(...)` 函数绑定。

`that` 与 `mProxyActivity` 等同，只是为了和 `this` 指针区分，表示真实的 `Context`，这里真实指的是被代理情况下为代理 `Activity`，未被代理情况下等同于 `this`。

#### 4.2.8 DLBasePluginService.java

插件 Service 基类，插件中的 Service 要继承这个基类，主要作用是根据是否被代理，确定一些函数直接走父类逻辑还是代理 Service 或是空逻辑。

主要变量含义与 `DLBasePluginActivity` 类似，不重复介绍。

PS：截止目前这个类还是不完善的，至少和 `DLBasePluginActivity` 对比，还不支持非代理的情况

#### 4.2.9 DLIntent.java

继承自 Intent，封装了待启动组件的 PackageName 和 ClassName。

#### 4.2.10 SoLibManager.java

调用 `SoLibManager` 拷贝 so 库到 Native Library 目录。

主要函数：

(1) `copyPluginSoLib(Context context, String dexPath, String nativeLibDir)`

函数中以 `zipFile` 形式加载插件，循环读取其中的文件，如果为 `.so` 结尾文件、符合当前平台 CPU 类型且尚未拷贝过最新版，则新建 `Runnable` 拷贝 so 文件。

#### 4.2.11 DLUtils.java

这个类中大都是无用或是不该放在这里的函数，也许是大版本升级及维护人过多后对工具函数的维护不够所致。

### 5. 杂谈

#### 5.1 插件不能打包 dl-lib.jar

原因是插件和宿主属于不同的 ClassLoader，如果同时打包 `dl-lib.jar`，会因为 ClassLoader 隔离导致类型转换错误，具体可见：[ClassLoader 隔离](#)

Eclipse 打包解决方式见项目主页；

Android Studio 打包解决方式见 5.2；

Ant 打包需要修改 `build.xml` 中 `dex target` 引用到的 `compileclasspath` 属性。

#### 5.2 在 Android Studio 下使用 DynamicLoadApk

在使用 DynamicLoadApk 时有个地方要注意，就是插件 Apk 在打包的时候不能把 dl-lib.jar 文件打包进去，不然会报错(java.lang.IllegalAccessError: Class ref in pre-verified class resolved to unexpected implementation)。换句话说，dl-lib.jar 要参与编译，但不参与打包。该框架作者已经给出了 Eclipse 下的解决方案。我这里再说下怎么在 Android Studio 里使用。

```
dependencies {
 provided fileTree(dir: 'dl-lib', include: ['*.jar'])
}
```

## 5.3 DynamicLoadApk 待完善的问题

- (1) 还未支持广播；
- (2) Base Plugin 中的 that 还未去掉，需要覆写 Activity 的相关方法；
- (3) 插件和宿主资源 id 可能重复的问题没有解决，需要修改 aapt 中资源 id 的生成规则；
- (4) 不支持自定义主题，不支持系统透明主题；
- (5) 插件中的 so 处理有异常；
- (6) 不支持静态 Receiver；
- (7) 不支持 Provider；
- (8) 插件不能直接用 this；

## 5.4 其他插件化方案

除了 DynamicLoadApk 用代理的方式实现外，目前还有两种插件化方案：

- (1) 用 Fragment 以及 schema 的方式实现。
- (2) 利用字节码库动态生成一个插件类 A 继承自待启动插件 Activity，启动插件 A。这个插件 A 名称固定且已经在 Manifest 中注册。

具体可见：[Android 插件化](#)

最后 H5 框架越来越多，也能解决插件化解决的自动升级这部分功能，硬件、网络也在改善，未来何如？

# 动态加载APK原理分享

来源:<http://blog.csdn.net/hkxxx/article/details/42194387>

项目地址: <https://github.com/houkx/android-pluginmgr/> 欢迎star and fork

## (一) 综述

随着智能手机硬件性能的逐步提升，移动应用也做的越来越复杂，android平台上应用的apk包体积也越来越大，然后同类产品开始比拼谁的体积小，实现方案呢，然后很容易想到“插件化”，也就是说可以发布内核很小的产品，随着添加功能的需求而动态下载功能模块，促使插件化的另一个动机是App应用固有的问题，那就是很多组件需要注册，更新功能的话不能像Web应用那样可在用户无察觉的情况下通过升级服务器而方便升级，只能弹出个框让用户重新下载整个程序包，然后调取系统安装流程。

被加载的apk称之为插件，因为机制类似于生物学的“寄生”，加载了插件的应用也被称为宿主。

往往不是所有的apk都可作为插件被加载，往往需要遵循一定的“开发规范”，还需要插件项目引入某种api类库，业界通常都是这么做的。

这里介绍一种无须规范限制的动态加载解决方案，插件不需要依赖任何API，这也是本人突发异想，灵感所致。

## 二) 功能介绍

特点：

- 插件为普通apk，无须依赖任何jar
- Activity生命周期由系统自己管理
- 使用简单，只需要了解一个类PluginManager的两个方法
- 启动Activity的效率高
- 不修改插件，被加载的插件仍然可以独立安装。

功能点：

- 可加载任意apk中的Activity(包括子类ActionBarActivity、FragmentActivity)的派生类(不包括违反限制条件的Activity)

- 支持插件自定义Application
- 支持插件Apk中的Activity跳转到别的Activity(插件内部的或系统的,外部已安装apk的,甚至是别的插件中的),也没有任何限制
- 支持Activity设置主题(与系统的主题应用规则一样,如果Activity没指定Theme,但所在Application指定了Theme,则使用Application的Theme)
- 初步支持.so
- 支持插件使用 SharedPreferences 或 SQLite数据库(尚未完善)



ps:第一个插件代码来自<https://github.com/viacheslavtitov/NDKBeginning>,作者是个老外,不过也比较粗心,要正常运行你需要在sd卡下创建目录:FFMPEG

第二个插件代码来自这篇博文:<http://blog.csdn.net/caihanyuan/article/details/7367351>

第三个插件代码来自大名鼎鼎的 PullToRefresh : <https://github.com/johannilsson/android-pulltorefresh>

第6第7个插件代码是自测项目, 分别测试ActionBarActivity和Activity基本加载和跳转

其他apk还没能正常加载, 框架还在不断完善中, 不过腾讯的开心消消乐可以帮助记录crashlog, 这点真不错~

将要支持的特性:PackageManager service 等等, 详情都列在开源项目[android-pluginmgr](#)下的TODO文件中

ps:修正项目主页README描述的限制条件1, 目前已经没有这个限制了, 在插件中调用 `context.getPackageName()` 返回插件的清单文件中声明的包名

限制条件(永不支持的):

- 插件apk中不能假定自己已经安装, 以及由此造成的影响, 比如认为 `applicationInfo.dataDir==/data/data/packageName`
- 不能依赖清单文件中的进程声明, 被加载的apk以及里面的任何组件目前都在同一个进程管理。
- 插件中的权限, 无法动态注册, 插件中的权限都得在宿主中注册 (暂无解决方案)

## (三) 实现

动态加载需要处理很多问题, 虽然有很多问题, 但是核心问题就是加载Activity, 因为Activity是可见的, 人们对可以看到的东西总是那么重视, 视觉信息占人所处理信息的90%以上。

Activity如何调起来? 资源的加载等等已经有大牛的文章介绍了, 我本菜鸟, 不再赘述。

目前Activity的加载或许有很多处理方式, 但是可以分为两种: 一是自己new 二是系统new。很多动态加载框架基于第一种方式。我这个方案基于第二种, 既然要系统new, 就要系统自己可以找到相应的Activity. 由于Activity需要在清单文件注册了才能使用, 所以要注册Activity, 但是如何注册呢?

我在网上看到有人用极端的方式: 插件里的所有Activiy都在宿主里注册, 既然宿主总要修改升级, 何必要插件呢, 这已经违背了动态加载的初衷:不修改框架而动态扩展功能更多的是这么做, 注册一个Activity基类, 供插件中的Activity继承, 在这个基类里做动态加载的核心逻辑, 这就要求插件必须依赖某种API类库。

我的方案通俗的说是这样, 依赖倒转, 不让插件依赖框架API,而是反过来, 自动生成一个Activity类依赖(继承)插件中的Activity, 这个自动生成的类就叫PluginActivity

并且声明在框架的清单文件中, 如下:

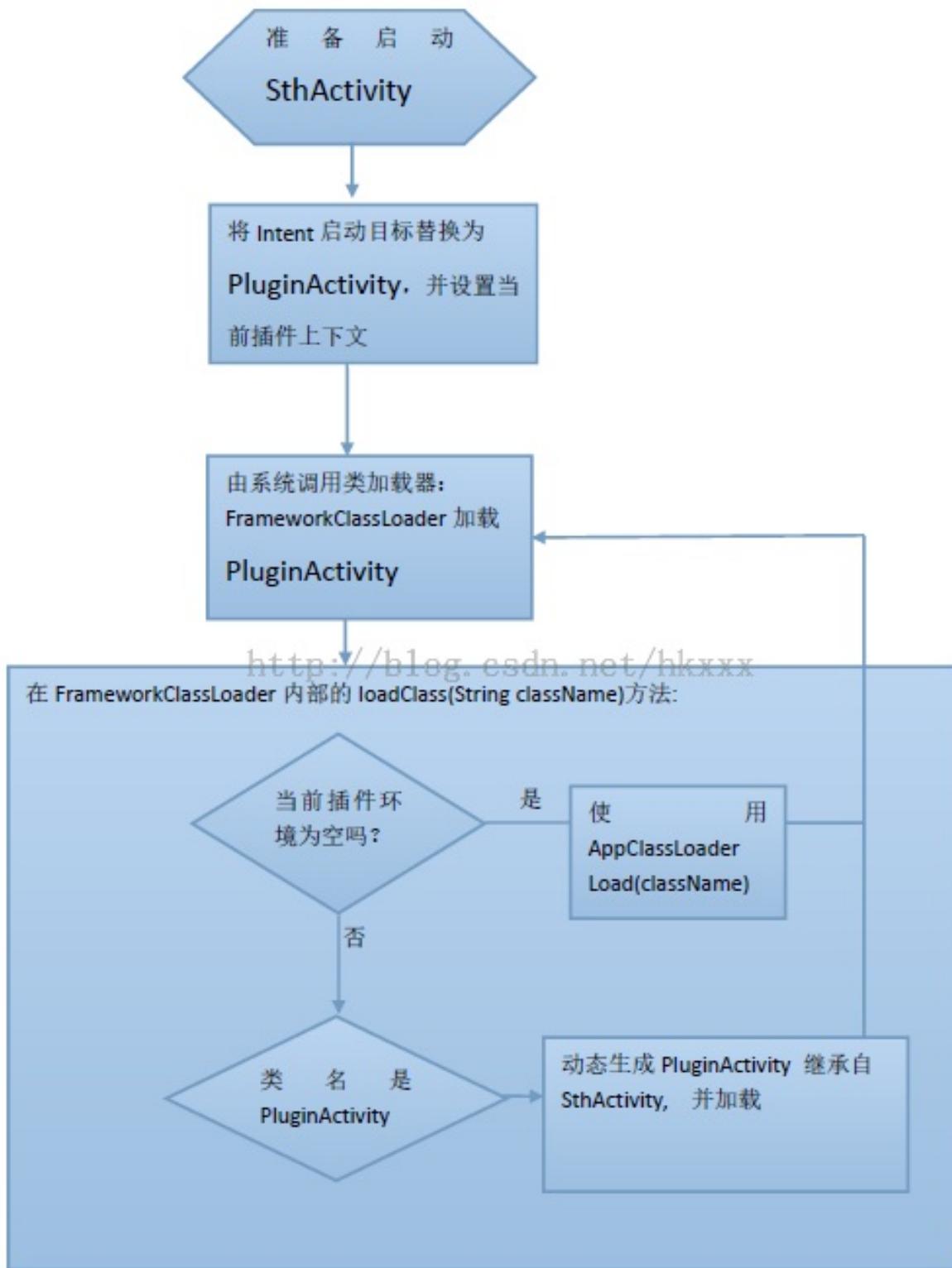
```
<activity name="androidx.pluginmgr.PluginActivity" />
```

聪明的读者会想,等一下, 插件里面Activity可不止一个, 你就注册一个?

是的，就一个，自动生成的Activity类名都是 `androidx.pluginmgr.PluginActivity`，不过放在不同的文件中，最简单的映射，原始Activity类名.dex文件中存储对应的子类：`PluginActivity`

其实也是偷梁换柱了，如果你想启动插件里的Activity，如 `com.test.MyPlugActivity`，我就把启动目标修改为 `androidx.pluginmgr.PluginActivity` 类，然后从 `com.test.MyPlugActivity.dex` 文件中找到 `public class androidx.pluginmgr.PluginActivity extends com.test.MyPlugActivity{....}`

以启动SthActivity为例：



好了，核心思想已经表达清楚了，下面介绍如何让系统按你说的路径去找类文件，这涉及到类加载器。自定义类加载器比较简单，继承 `java.lang.ClassLoader` 即可。

在我的开源项目源码中对应的类是 `FrameworkClassLoader`，`PluginManager` 初始化时就去修改 `Application` 的类加载器，替换为 `FrameworkClassLoader`。

`FrameworkClassLoader` 其实不干什么实际加载工作，只是分发任务：

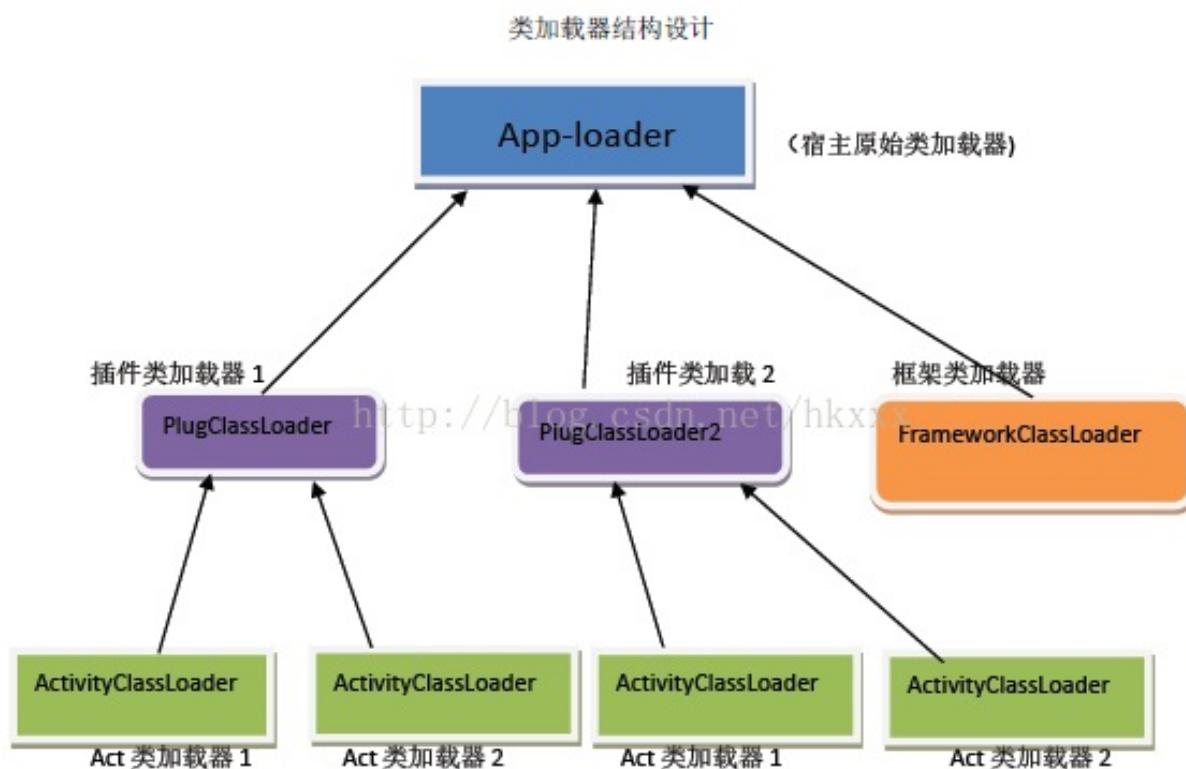
```

public Class loadClass(String className){
 if(当前上下文插件不为空) {
 if(className 是 PluginActivity){
 // 找到当前实际要加载的原始 Activity
 return // 使用插件对应的ActivityClassLoader从 (自动生成的)
 // 原始Activity类名.dex 文件 加载PluginActivity
 }else{
 return 使用对应的 PluginClassLoader 加载普通类
 }
 }else{
 return super.loadClass();//即委派给宿主Application的原始类加载器加载
 }
}

```

其中，`PluginClassLoader` 是一个 `DexClassLoader`，`parent` 指向 `FrameworkClassLoader`，`ActivityClassLoader` 也是一个 `DexClassLoader`，`parent` 指向 `PluginClassLoader`

插图2（类加载器结构图）：



```
package androidx.pluginmgr;
```

```
import android.util.Log;
```

```

/**
 * 框架类加载器 (Application 的 classLoder被替换成此类的实例)
 *
 */
class FrameworkClassLoader extends ClassLoader {
 private String[] plugIdAndActname;//代表插件上下文

 public FrameworkClassLoader(ClassLoader parent) {
 super(parent);
 }
 //在外部或插件内部的 startActivity 时调用这个方法设置插件上下文
 String newActivityClassName(String plugId, String actName) {
 plugIdAndActname = new String[] { plugId, actName };
 // targetClassName即宿主manifest配置的androidx.pluginmgr.PluginActivity
 return ActivityOverider.targetClassName;
 }

 protected Class<?> loadClass(String className, boolean resolv)
 throws ClassNotFoundException {
 Log.i("cl", "loadClass: " + className);
 String[] plugIdAndActname = this.plugIdAndActname;
 Log.i("cl", "plugIdAndActname = " + Arrays.toString(plugIdAndActname));
 if (plugIdAndActname != null) {
 String pluginId = plugIdAndActname[0];

 PlugInfo plugin = PluginManager.getInstance().getPluginById(
 pluginId);
 Log.i("cl", "plugin = " + plugin);
 if (plugin != null) {
 try {
 if (className.equals(ActivityOverider.targetClassName)) {
 String actClassName = plugIdAndActname[1];//被 (继承) 代理的Acti
 return plugin.getClassLoader().loadActivityClass(
 actClassName);// 加载动态生成的Activity类
 }else{
 return plugin.getClassLoader().loadClass(className);//加载普通类
 }
 } catch (ClassNotFoundException e) {
 e.printStackTrace();
 }
 }
 }
 return super.loadClass(className, resolv);
 }
}

```

到此为止，一个动态加载框架的核心雏形已经有了，但是还有许多细节待完善。

附加：另外，关于动态生成类，对于davikvm环境，我所知道的工具有asmdex和dexmaker，我在项目中选用的是dexmaker

用什么不是重点，看生成的代码吧：(ActivityOverider 类负责与自动生成的Activity类交互)

```
package androidx.pluginmgr;

import android.content.ComponentName;
import android.content.Intent;
import android.content.ServiceConnection;
import android.content.res.AssetManager;
import android.content.res.Resources;
import android.os.Bundle;
import androidplugdemo.SthActivity;

public final class PluginActivity extends SthActivity {
 private static final String _pluginId = "activityTest_v1";
 private AssetManager mAssertManager;
 private Resources mResources;

 public boolean bindService(Intent paramIntent,
 ServiceConnection paramServiceConnection, int paramInt) {
 return ActivityOverider.overrideBindService(this, _pluginId,
 paramInt, paramServiceConnection, paramInt);
 }

 public AssetManager getAssets() {
 AssetManager localAssetManager = this.mAssertManager;
 if (localAssetManager == null) {
 localAssetManager = super.getAssets();
 }
 return localAssetManager;
 }

 public Resources getResources() {
 Resources localResources = this.mResources;
 if (localResources == null) {
 localResources = super.getResources();
 }
 return localResources;
 }

 public void onBackPressed() {
 if (ActivityOverider.overrideOnBackPressed(this, _pluginId)) {
 super.onBackPressed();
 }
 }
}
```

```
}

protected void onCreate(Bundle paramBundle) {

 String str = _pluginId;
 AssetManager localAssetManager = ActivityOverider.getAssetManager(str, this);
 this.mAssertManager = localAssetManager;
 Resources localResources = super.getResources();
 this.mResources = new Resources(localAssetManager,
 localResources.getDisplayMetrics(), localResources.getConfiguration());
 ActivityOverider.callback_onCreate(str, this);
 super.onCreate(paramBundle);
}

protected void onDestroy() {
 String str = _pluginId;
 super.onDestroy();
 ActivityOverider.callback_onDestroy(str, this);
}

protected void onPause() {
 String str = _pluginId;
 super.onPause();
 ActivityOverider.callback_onPause(str, this);
}

protected void onRestart() {
 String str = _pluginId;
 super.onRestart();
 ActivityOverider.callback_onRestart(str, this);
}

protected void onResume() {
 String str = _pluginId;
 super.onResume();
 ActivityOverider.callback_onResume(str, this);
}

protected void onStart() {
 String str = _pluginId;
 super.onStart();
 ActivityOverider.callback.onStart(str, this);
}

protected void onStop() {
 String str = _pluginId;
 super.onStop();
 ActivityOverider.callback_onStop(str, this);
}

public void startActivityForResult(Intent paramInt,
 int paramInt, Bundle paramBundle) {
```

```
super.startActivityForResult(ActivityOverider
 .overrideStartActivityResult(
 this, _pluginId, paramInt,
 paramInt, paramBundle), paramInt, paramBundle);
}

public ComponentName startService(Intent paramInt) {
 return ActivityOverider.overrideStartService(
 this, _pluginId, paramInt);
}

public boolean stopService(Intent paramInt) {
 return ActivityOverider.overrideStopService(
 this, _pluginId, paramInt);
}

public void unbindService(ServiceConnection paramServiceConnection) {
 ActivityOverider.overrideUnbindService(
 this, _pluginId, paramServiceConnection);
}
}
```

项目地址: <https://github.com/houkx/android-pluginmgr/> 欢迎star and fork

# 携程Android App插件化和动态加载实践

来源:[infoQ](#)

编者按：本文为携程无线基础团队投稿，介绍它们已经开源的Android动态加载解决方案[DynamicAPK](#)，本文作者之一，携程无线研发总监陈浩然将会在[ArchSummit北京2015架构师大会](#)上分享架构优化相关内容，欢迎关注。

携程Android App的插件化和动态加载框架已上线半年，经历了初期的探索和持续的打磨优化，新框架和工程配置经受住了生产实践的考验。本文将详细介绍Android平台插件式开发和动态加载技术的原理和实现细节，回顾携程Android App的架构演化过程，期望我们的经验能帮助到更多的Android工程师。

## 需求驱动

2014年，随着业务发展需要和携程无线部门的拆分，各业务产品模块归属到各业务BU，原有携程无线App开发团队被分为基础框架、酒店、机票、火车票等多个开发团队，从此携程App的开发和发布进入了一个全新模式。在这种模式下，开发沟通成本大大提高，之前的协作模式难以为继，需要新的开发模式和技术解决需求问题。

另一方面，从技术上来说，携程早在2012年就触到Android平台史上最坑天花板（没有之一）：65535方法数问题。旧方案是把所有第三方库放到第二个dex中，并且利用Facebook当年发现的[hack方法](#)扩大点 `LinearAllocHdr` 分配空间（5M提升到8M），但随着代码的膨胀，旧方案也逐渐捉襟见肘。拆or不拆，根本不是可考虑问题，继续拆分dex是我们的唯一出路。问题在于：怎么拆才比较聪明？

其次，随着组织架构调整的影响，给我们的App质量控制带来极高的挑战，这种紧张和压力让我们的开发团队心力憔悴。此时除了流着口水羡慕前端同事们的在线更新持续发布能力之外，难道就没有办法解决Native架构这一根本性缺陷了吗？NO！插件化动态加载带来的额外好处就是客户端的热部署能力。

从以上几点根本性需求可以看出，插件化动态加载架构方案会为我们带来多么巨大的收益，除此之外还有诸多好处：

- 编译速度提升

工程被拆分为十来个子工程之后，Android Studio编译流程繁冗的缺点被迅速放大，在Win7机械硬盘开发机上编译时间曾突破1小时，令人发指的龟速编译让开发人员叫苦不迭（当然现在换成Mac+SSD快太多）。

- 启动速度提升

Google提供的MultiDex方案，会在主线程中执行所有dex的解压、dexopt、加载操作，这是一个非常漫长的过程，用户会明显的看到长久的黑屏，更容易造成主线程的ANR，导致首次启动初始化失败。

- A/B Testing

可以独立开发AB版本的模块，而不是将AB版本代码写在同一个模块中。

- 可选模块按需下载

例如用于调试功能的模块可以在需要时进行下载后进行加载，减少App Size

列举了这么多痛点，童鞋们早就心潮澎湃按捺不住了吧？言归正传，开始插件化动态加载架构探索之旅。

## 原理

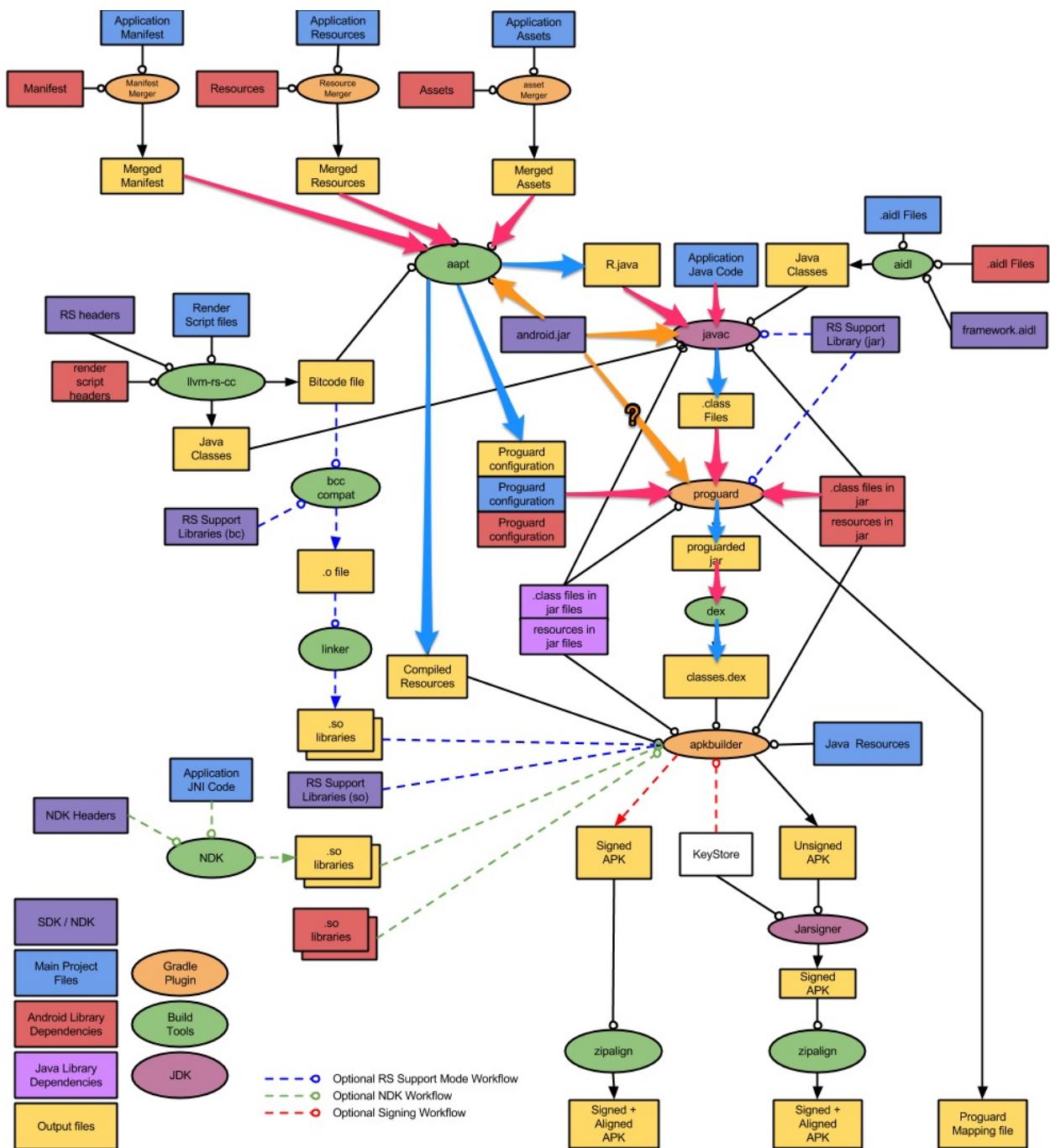
关于插件化思想，软件业已经有足够多的用户教育。无论是日常使用的浏览器，还是陪伴程序员无数日夜的Eclipse，甚至连QQ背后，都有插件化技术的支持。我们要在Android上实现插件化，主要需要考虑2个问题：

- 编译期：资源和代码的编译
- 运行时：资源和代码的加载

解决了以上2个关键问题，之后如何实现插件化的具体接口，就变成个人技术喜好或者具体需求场景差异而已。现在我们就针对以上关键问题逐一破解，其中最麻烦的还是资源的编译和加载问题。

## Android是如何编译的？

首先来回顾下Android是如何进行编译的。请看下图：



整个流程庞大而复杂，我们主要关注几个重点环节： `aapt`、`javac`、`proguard`、`dex`。相关环节涉及到的输入输出都在图上重点标粗。

## 资源的编译

Android的资源编译依赖一个强大的命令行工具：`aapt`，它位于 `<SDK>/build-tools/<buildToolsVersion>/aapt`，有着众多的[命令行参数](#)，其中有几个值得我们特别关注：

- `-I add an existing package to base include set`

这个参数可以在依赖路径中追加一个已经存在的package。在Android中，资源的编译也需要依赖，最常用的依赖就是SDK自带的android.jar本身。打开android.jar可以看到，其实不是一个普通的jar包，其中不但包含了已有SDK类库class，还包含了SDK自带的已编译资源以及资源索引表resources.arsc文件。在日常的开发中，我们也经常通过@android:color/opaque\_red形式来引用SDK自带资源。这一切都来自于编译过程中aapt对android.jar的依赖引用。同理，我们也可以使用这个参数引用一个已存在的apk包作为依赖资源参与编译。

- **-G A file to output proguard options into.**

资源编译中，对组件的类名、方法引用会导致运行期反射调用，所以这一类符号量是不能在代码混淆阶段被混淆或者被裁减掉的，否则等到运行时会找不到布局文件中引用到的类和方法。-G方法会导出在资源编译过程中发现的必须keep的类和接口，它将作为追加配置文件参与到后期的混淆阶段中。

- **-J specify where to output R.java resource constant definitions**

在Android中，所有资源会在Java源码层面生成对应的常量ID，这些ID会记录到R.java文件中，参与到之后的代码编译阶段中。在R.java文件中，Android资源在编译过程中会生成所有资源的ID，作为常量统一存放在R类中供其他代码引用。在R类中生成的每一个int型四字节资源ID，实际上都由三个字段组成。第一字节代表了Package，第二字节为分类，三四字节为类内ID。例如：

```
//android.jar中的资源，其PackageID为0x01
public static final int cancel = 0x01040000;

//用户app中的资源，PackageID总是0x7F
public static final int zip_code = 0x7f090f2e;
```

我们修改aapt后，是可以给每个子apk中的资源分配不同头字节PackageID，这样就不会再互相冲突。

## 代码的编译

大家对Java代码的编译应该相当熟悉，只需要注意以下几个问题即可：

- classpath:Java源码编译中需要找齐所有依赖项，classpath就是用来指定去哪些目录、文件、jar包中寻找依赖。
- 混淆:为了安全需要，绝大部分Android工程都会被混淆。混淆的原理和配置可参考[Proguard手册](#)。

有了以上背景知识，我们就可以思考并设计插件化动态加载框架的基本原理和主要流程了。

## 实现

实现分为两类：

1. 针对插件子工程做的编译流程改造
2. 运行时动态加载改造（宿主程序动态加载插件，有两个壁垒需要突破：资源如何访问，代码如何访问）。

## 插件资源编译

针对插件的资源编译，我们需要考虑到以下几点：

- 使用-I参数对宿主的apk进行引用。

据此，插件的资源、xml布局中就可以使用宿主的资源和控件、布局类了。

- 为aapt增加--apk-module参数。

如前所述，资源ID其实有一个PackageID的内部字段。我们为每个插件工程指定独特的PackageID字段，这样根据资源ID就很容易判明，此资源需要从哪个插件apk中去查找并加载了。在后文的资源加载部分会有进一步阐述。

- 为aapt增加--public-R-path参数。

按照对android.jar包中资源使用的常规手段，引用系统资源可使用它的R类的全限定名 android.R 来引用具体ID，以便和当前项目中的R类区分。插件对于宿主的资源引用，当然也可以使用 base.package.name.R 来完成。但由于历史原因，各子BU的“插件”代码是从主app中解耦独立出去的，资源引用还是直接使用当前工程的R。如果改为标准模式，则当前大量遗留代码中R都需要酌情改为base.R，工程量大并且容易出错，未来对bu开发人员的使用也有点不够“透明”。因此我们在设计上做了让步，额外增加 --public-R-path 参数，为aapt指明了 base.R 的位置，让它在编译期间把base的资源ID定义在插件的R类中完整复制一份，这样插件工程即可和之前一样，完全不用在乎资源来自于宿主或者自身，直接使用即可。当然这样做带来的副作用就是宿主和插件的资源不应有重名，这点我们通过开发规范来约束，相对比较容易理解一些。

## 插件代码编译

针对插件的代码编译，需要考虑以下几点：

- **classpath:**对于插件的编译来说，除了对android.jar以及自己需要的第三方库进行依赖之外，还需要依赖宿主导出的base.jar类库。同时对宿主的混淆也提出了要求：宿主的所有public/protected都可能被插件依赖，所以这些接口都不允许被混淆。
- **混淆:**插件工程在混淆的时候，当然也要把宿主的混淆后jar包作为参考库导入。

自此，编译期所有重要步骤的技术方案都已经确定，剩下的工作就只是把插件apk导入到先一步生成好的base.apk中并重新进行签名对齐而已。

万事俱备，只欠表演。接下来我们看看在运行时插件们是如何登台亮相的。

## 运行时资源的加载

平常我们使用资源，都是通过AssetManager类和Resources类来访问的。获取它们的方法位于Context类中。

Context.java

```
/** Return an AssetManager instance for your application's package. */
public abstract AssetManager getAssets();

/** Return a Resources instance for your application's package. */
public abstract Resources getResources();
```

它们是两个抽象方法，具体的实现在ContextImpl类中。ContextImpl类中初始化Resources对象后，后续Context各子类包括Activity、Service等组件就都可以通过这两个方法读取资源了。

ContextImpl.java

```
private final Resources mResources;

@Override
public AssetManager getAssets() {
 return getResources().getAssets();
}

@Override
public Resources getResources() {
 return mResources;
}
```

既然我们已经知道一个资源ID应该从哪个apk去读取（前面在编译期我们已经在资源ID第一个字节标记了资源所属的package），那么只要我们重写这两个抽象方法，即可指导应用程序去正确的地方读取资源。

至于读取资源，AssetManager有一个隐藏方法 `addAssetPath`，可以为AssetManager添加资源路径。

```
/*
 * Add an additional set of assets to the asset manager. This can be
 * either a directory or ZIP file. Not for use by applications. Returns
 * the cookie of the added asset, or 0 on failure.
 * {@hide}
 */
public final int addAssetPath(String path) {
 synchronized (this) {
 int res = addAssetPathNative(path);
 makeStringBlocks(mStringBlocks);
 return res;
 }
}
```

我们只需反射调用这个方法，然后把插件apk的位置告诉AssetManager类，它就会根据apk内的resources.arsc和已编译资源完成资源加载的任务了。

以上我们已经可以做到加载插件资源了，但使用了一大堆定制类实现。要做到“无缝”体验，还需要一步：使用Instrumentation来接管所有Activity、Service等组件的创建（当然也就包含了它们使用到的Resources类）。

话说Activity、Service等系统组件，都会经由`android.app.ActivityThread`类在主线程中执行。`ActivityThread`类有一个成员叫`mInstrumentation`，它会负责创建Activity等操作，这正是注入我们的修改资源类的最佳时机。通过篡改`mInstrumentation`为我们自己的`InstrumentationHook`，每次创建Activity的时候顺手把它的`mResources`类偷天换日为我们的`DelegateResources`，以后创建的每个Activity都拥有一个懂得插件、懂得委托的资源加载类啦！

当然，上述替换都会针对Application的Context来操作。

## 运行时类的加载

类的加载相对比较简单。与Java程序的运行时classpath概念类似，Android的系统默认类加载器`PathClassLoader`也有一个成员`pathList`，顾名思义它从本质来说是一个List，运行时会从其间的每一个dex路径中查找需要加载的类。既然是个List，一定就会想到，给它追

加一堆dex路径不就得了？实际上，Google官方推出的MultiDex库就是用以上原理实现的。下面代码片段展示了修改pathList路径的细节：

### MuliDex.java

```
private static void install(ClassLoader loader, List<File> additionalClassPathEntries
 File optimizedDirectory)
 throws IllegalArgumentException, IllegalAccessException,
 NoSuchFieldException, InvocationTargetException, NoSuchMethodException {
 /* The patched class loader is expected to be a descendant of
 * dalvik.system.BaseDexClassLoader. We modify its
 * dalvik.system.DexPathList pathList field to append additional DEX
 * file entries.
 */
 Field pathListField = findField(loader, "pathList");
 Object dexPathList = pathListField.get(loader);
 expandFieldArray(dexPathList, "dexElements", makeDexElements(dexPathList,
 new ArrayList<File>(additionalClassPathEntries), optimizedDirectory));
}
```

当然，针对不同Android版本，类加载方式略有不同，可以参考[MultiDex源码](#)做具体的区别处理。

至此，之前提出的四个根本性问题，都已经有了具体的解决方案。剩下的就是编码！

编码主要分为三部分：

- 对aapt工具的修改。
- gradle打包脚本的实现。
- 运行时加载代码的实现。

具体实现可以参考我们在GitHub上的开源项目[DynamicAPK](#)。

## 收益与代价

任何事物都有其两面性，尤其像动态加载这种使用了非官方Hack技术的方案，更需要在规划阶段把收益和代价考虑清楚，方便完成后进行复盘。

### 收益

- 插件化架构适应现有组织架构和开发节奏需求，各BU不但从代码层面，更从项目控制层面做到了高内聚低耦合，极大降低了沟通成本，提高了工作效率。

- 拆分成多个小的插件后，dex从此告别方法数天花板。
- HotFix为app质量做好最后一层保障方案，再也没有无法挽回的损失了，而且现在HotFix的级别粒度可控，即可以是传统class级别（直接使用pathClassLoader实现），也可以是带资源的apk级别。
- ABTesting脱离古老丑陋的if/else实现，多套方案随心挑选按需加载。
- 编译速度大大提高，各BU只需使用宿主的编译成果更新编译自己子工程部分，分分钟搞定。
- App宿主apk大大减小，各业务模块按需后台加载或者延迟懒加载，启动速度优化，告别黑屏和启动ANR。
- 各BU插件apk独立，谁胖谁瘦一目了然，app size控制有的放矢。

以上收益，基本达到甚至超出了项目的预期目标：D

## 代价

- 资源别名

Android提供了强大的资源别名规则，参考可以获取更多细节描述。但不幸的是，在三星S6等部分机型上使用资源别名会出现宿主资源和插件资源ID错乱导致资源找不到的问题。无奈只能禁止使用这一技术，所幸放弃这个高级特性不会引起根本性损失。

- 重名资源

如前文所述的原因，宿主的资源ID会在插件中完整复制一份。失去了包名这一命名空间的保护，重名资源会直接造成冲突。暂时通过命名规范的方式规避，好在良好的命名习惯也是各开发应该做到的，因此解决代价较小。

- 枚举

很多控件都会使用枚举来约束属性的取值范围。不幸的是Android的枚举居然是用命名来唯一确定R中生成的id常量，毫无命名空间或者所属控件等顾忌。因为上一点同样的原因，宿主和插件内的同名枚举会造成id冲突。暂时同样通过命名规范的方式规避。

- 外部访问资源能力。

对于极少数需要从外部访问apk资源的场合（例如发送延时通知），此时App尚未启动，资源的获取由系统代劳，理所当然无法洞悉内部插件的资源位置和获取方式。对于这种情况实在无能为力，只好特别准许此类资源直接放在宿主apk内。

以上代价，或者无伤大雅，或者替代方案成本非常低，都在可接受范围内。

## 未来优化

还有一些高级特性，因为优先级关系暂未实现，但随着各业务线的开发需求也被提到优化日程上来，如：

- 插件工程支持so库。
- 插件工程支持lib工程依赖、aar依赖、maven远程依赖等各种高级依赖特性。
- IDE友好，让开发人员可以更方便的生成插件apk。

## 开源

经过以上介绍，相信各位对携程Android插件化开发和动态加载方案有了初步了解。细节请移步GitHub开源项目[DynamicAPK](#)。携程无线基础研发团队未来会继续努力，为大家分享更多项目实践经验。

# 美团Android DEX自动拆包及动态加载简介

来源:[美团Android DEX自动拆包及动态加载简介](#)

## 概述

作为一个android开发者，在开发应用时，随着业务规模发展到一定程度，不断地加入新功能、添加新的类库，代码在急剧的膨胀，相应的apk包的大小也急剧增加，那么终有一天，你会不幸遇到这个错误：

- 生成的apk在android 2.3或之前的机器上无法安装，提示  
INSTALL\_FAILED\_DEXOPT
- 方法数量过多，编译时出错，提示：  
Conversion to Dalvik format failed:Unable to execute dex: method ID not in  
[0, 0xffff]: 65536

而问题产生的具体原因如下：

- 无法安装（Android 2.3 INSTALL\_FAILED\_DEXOPT）问题，是由dexopt的LinearAlloc限制引起的，在Android版本不同分别经历了4M/5M/8M/16M限制，目前主流4.2.x系统上可能都已到16M，在Gingerbread或者以下系统LinearAllocHdr分配空间只有5M大小的，高于Gingerbread的系统提升到了8M。Dalvik linearAlloc是一个固定大小的缓冲区。在应用的安装过程中，系统会运行一个名为dexopt的程序为该应用在当前机型中运行做准备。dexopt使用LinearAlloc来存储应用的方法信息。  
Android 2.2和2.3的缓冲区只有5MB，Android 4.x提高到了8MB或16MB。当方法数量过多导致超出缓冲区大小时，会造成dexopt崩溃。
- 超过最大方法数限制的问题，是由于DEX文件格式限制，一个DEX文件中method个数采用使用原生类型short来索引文件中的方法，也就是4个字节共计最多表达65536个method，field/class的个数也均有此限制。对于DEX文件，则是将工程所需全部class文件合并且压缩到一个DEX文件期间，也就是Android打包的DEX过程中，单个DEX文件可被引用的方法总数（自己开发的代码以及所引用的Android框架、类库的代码）被限制为65536；

## 插件化？ MultiDex？

解决这个问题，一般有下面几种方案，一种方案是加大Proguard的力度来减小DEX的大小和方法数，但这是治标不治本的方案，随着业务代码的添加，方法数终究会到达这个限制，一种比较流行的方案是插件化方案，另外一种是采用google提供的MultiDex方案，以及google在推出MultiDex之前Android Developers博客介绍的通过[自定义类加载过程](#)，再就是Facebook推出的为Android应用开发的[Dalvik补丁](#)，但facebook博客里写的不是很详细；我们在插件化方案上也做了探索和尝试，发现部署插件化方案，首先需要梳理和修改各个业务线的代码，使之解耦，改动的面和量比较大，通过一定的探讨和分析，我们认为对我们目前来说采用MultiDex方案更靠谱一些，这样我们可以快速和简洁的对代码进行拆分，同时代码改动也在可以接受的范围内；这样我们采用了google提供的MultiDex方式进行了开发。

插件化方案在业内有不同的实现原理，这里不再一一列举，这里只列举下Google为构建超过65K方法数的应用提供官方支持的方案：[MultiDex](#)。

首先使用Android SDK Manager升级到最新的Android SDK Build Tools和Android Support Library。然后进行以下两步操作：

## 1.修改Gradle配置文件，启用MultiDex并包含MultiDex支持

```
android {
 compileSdkVersion 21 buildToolsVersion "21.1.0"

 defaultConfig {
 ...
 minSdkVersion 14
 targetSdkVersion 21
 ...

 // Enabling MultiDex support.
 multiDexEnabled true
 }
 ...
}

dependencies {
 compile 'com.android.support:MultiDex:1.0.0'
}
```

## 2.让应用支持多DEX文件。在官方文档中描述了三种可选方法

- 在AndroidManifest.xml的application中声明  
    `android.support.MultiDex.MultiDexApplication;`
- 如果你已经有自己的Application类，让其继承MultiDexApplication；
- 如果你的Application类已经继承自其它类，你不想/能修改它，那么可以重写

attachBaseContext()方法：

```
@Override
protected void attachBaseContext(Context base) {
 super.attachBaseContext(base);
 MultiDex.install(this);
}
```

并在Manifest中添加以下声明：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.example">
 <application
 ...
 android:name="android.support.MultiDex.MultiDexApplication"
 ...
 </application>
</manifest>
```

如果已经有自己的Application，则让其继承MultiDexApplication即可。

## Dex自动拆包及动态加载

### MultiDex带来的问题

在第一版本采用MultiDex方案上线后，在Dalvik下MultiDex带来了下列几个问题：

1. 在冷启动时因为需要安装DEX文件，如果DEX文件过大时，处理时间过长，很容易引发ANR (Application Not Responding)；
2. 采用MultiDex方案的应用可能不能在低于Android 4.0 (API level 14) 机器上启动，这个主要是因为Dalvik linearAlloc的一个bug ([Issue 22586](#))；
3. 采用MultiDex方案的应用因为需要申请一个很大的内存，在运行时可能导致程序的崩溃，这个主要是因为Dalvik linearAlloc 的一个限制([Issue 78035](#)). 这个限制在Android 4.0 (API level 14)已经增加了，应用也有可能在低于 Android 5.0 (API level 21)版本的机器上触发这个限制；

而在ART下MultiDex是不存在这个问题的，这主要是因为ART下采用Ahead-of-time (AOT) compilation技术，系统在APK的安装过程中会使用自带的dex2oat工具对APK中可用的DEX文件进行编译并生成一个可在本地机器上运行的文件，这样能提高应用的启动速度，

因为是在安装过程中进行了处理这样会影响应用的安装速度，对ART感兴趣的可以参考一下[ART和Dalvik的区别](#).

MultiDex的基本原理是把通过DexFile来加载Secondary DEX，并存放  
在[BaseDexClassLoader](#)的DexPathList中。

下面代码片段是BaseDexClassLoader findClass的过程：

```
protected Class<?> findClass(String name) throws ClassNotFoundException {
 List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
 Class c = pathList.findClass(name, suppressedExceptions);
 if (c == null) {
 ClassNotFoundException cnfe = new ClassNotFoundException("Didn't find class \
 for (Throwable t : suppressedExceptions) {
 cnfe.addSuppressed(t);
 }
 throw cnfe;
 }
 return c;
}
```

下面代码片段为怎么通过DexFile来加载Secondary DEX并放到BaseDexClassLoader的  
DexPathList中：

```

private static void install(ClassLoader loader, List<File> additionalClassPathEntries
 File optimizedDirectory)
 throws IllegalArgumentException, IllegalAccessException,
 NoSuchFieldException, InvocationTargetException, NoSuchMethodException {
 /* The patched class loader is expected to be a descendant of
 * dalvik.system.BaseDexClassLoader. We modify its
 * dalvik.system.DexPathList pathList field to append additional DEX
 * file entries.
 */
 Field pathListField = findField(loader, "pathList");
 Object dexPathList = pathListField.get(loader);
 ArrayList<IOException> suppressedExceptions = new ArrayList<IOException>();
 expandFieldArray(dexPathList, "dexElements", makeDexElements(dexPathList,
 new ArrayList<File>(additionalClassPathEntries), optimizedDirectory,
 suppressedExceptions));
 try {
 if (suppressedExceptions.size() > 0) {
 for (IOException e : suppressedExceptions) {
 //Log.w(TAG, "Exception in makeDexElement", e);
 }
 Field suppressedExceptionsField =
 findField(loader, "dexElementsSuppressedExceptions");
 IOException[] dexElementsSuppressedExceptions =
 (IOException[]) suppressedExceptionsField.get(loader);

 if (dexElementsSuppressedExceptions == null) {
 dexElementsSuppressedExceptions =
 suppressedExceptions.toArray(
 new IOException[suppressedExceptions.size()]);
 } else {
 IOException[] combined =
 new IOException[suppressedExceptions.size() +
 dexElementsSuppressedExceptions.length];
 suppressedExceptions.toArray(combined);
 System.arraycopy(dexElementsSuppressedExceptions, 0, combined,
 suppressedExceptions.size(), dexElementsSuppressedExceptions..
 dexElementsSuppressedExceptions = combined;
 }
 suppressedExceptionsField.set(loader, dexElementsSuppressedExceptions);
 }
 } catch(Exception e) {
 }
}

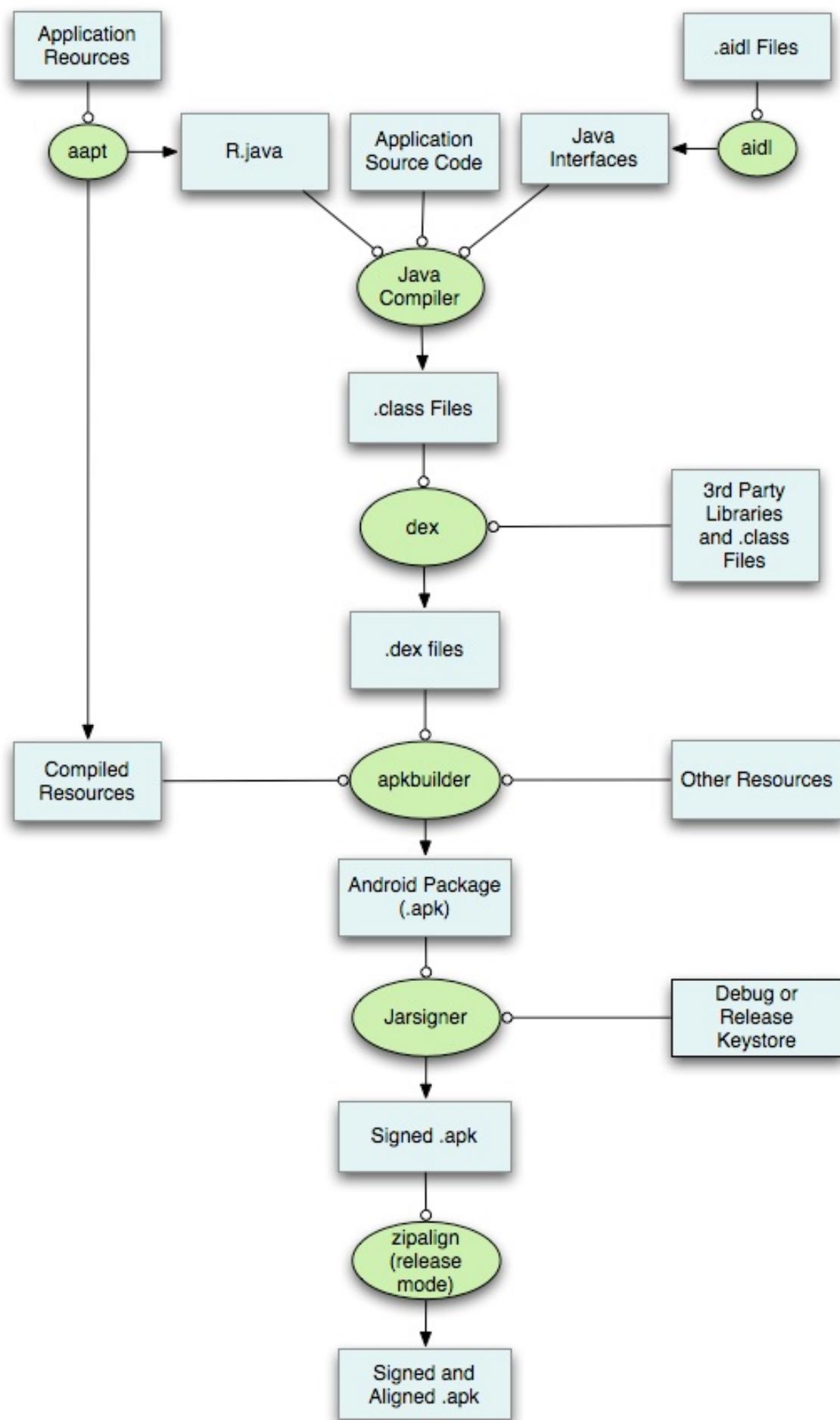
```

## Dex自动拆包及动态加载方案简介

通过查看MultiDex的源码，我们发现MultiDex在冷启动时容易导致ANR的瓶颈，在2.1版本之前的Dalvik的VM版本中，MultiDex的安装大概分为几步，第一步打开apk这个zip包，第二步把MultiDex的dex解压出来（除去Classes.dex之外的其他DEX，例如：classes2.dex, classes3.dex等等），因为android系统在启动app时只加载了第一个Classes.dex，其他的DEX需要我们人工进行安装，第三步通过反射进行安装，这三步其实都比较耗时，为了解决这个问题我们考虑是否可以把DEX的加载放到一个异步线程中，这样冷启动速度能提高不少，同时能够减少冷启动过程中的ANR，对于Dalvik linearAlloc的一个缺陷([Issue 22586](#))和限制([Issue 78035](#))，我们考虑是否可以人工对DEX的拆分进行干预，使每个DEX的大小在一定的合理范围内，这样就减少触发Dalvik linearAlloc的缺陷和限制；为了实现这几个目的，我们需要解决下面三个问题：

- 在打包过程中如何产生多个的DEX包？
- 如果做到动态加载，怎么决定哪些DEX动态加载呢？
- 如果启动后在工作线程中做动态加载，如果没有加载完而用户进行页面操作需要使用到动态加载DEX中的class怎么办？

我们首先来分析如何解决第一个问题，在使用MultiDex方案时，我们知道BuildTool会自动把代码进行拆成多个DEX包，并且可以通过配置文件来控制哪些代码放到第一个DEX包中，下图是Android的打包流程示意图：



为了实现产生多个DEX包，我们可以在生成DEX文件的这一步中，在Ant或gradle中自定义一个Task来干预DEX产生的过程，从而产生多个DEX，下图是在ant和gradle中干预产生DEX的自定task的截图：

```

tasks.whenTaskAdded { task ->
 if (task.name.startsWith('proguard') && (task.name.endsWith('Debug') || task.name
 task.doLast {
 makeDexFileAfterProguardJar();
 }
 task.doFirst {
 delete "${project.buildDir}/intermediates/classes-proguard";

 String flavor = task.name.substring('proguard'.length(), task.name.lastIn
 generateMainIndexKeepList(flavor.toLowerCase());
 }
 } else if (task.name.startsWith('zipalign') && (task.name.endsWith('Debug') || ta
 task.doFirst {
 ensureMultiDexInApk();
 }
 }
}
}

```

上一步解决了如何打包出多个DEX的问题了，那我们该怎么根据什么来决定哪些class放到Main DEX，哪些放到Secondary DEX呢（这里的Main DEX是指在2.1版本的Dalvik VM之前由android系统在启动apk时自己主动加载的Classes.dex，而Secondary DEX是指需要我们自己安装进去的DEX，例如：Classes2.dex, Classes3.dex等），这个需要分析出放到Main DEX中的class依赖，需要确保把Main DEX中class所有的依赖都要放进来，否则在启动时会发生ClassNotFoundException，这里我们的方案是把Service、Receiver、Provider涉及到的代码都放到Main DEX中，而把Activity涉及到的代码进行了一定的拆分，把首页Activity、Laucher Activity、欢迎页的Activity、城市列表页Activity等所依赖的class放到了Main DEX中，把二级、三级页面的Activity以及业务频道的代码放到了Secondary DEX中，为了减少人工分析class的依赖所带了的不可维护性和高风险性，我们编写了一个能够自动分析Class依赖的脚本，从而能够保证Main DEX包含class以及他们所依赖的所有class都在其内，这样这个脚本就会在打包之前自动分析出启动到Main DEX所涉及的所有代码，保证Main DEX运行正常。

随着第二个问题的迎刃而解，我们来到了比较棘手的第三问题，如果我们在后台加载Secondary DEX过程中，用户点击界面将要跳转到使用了在Secondary DEX中class的界面，那此时必然发生ClassNotFoundException，那怎么解决这个问题呢，在所有的Activity跳转代码处添加判断Secondary DEX是否加载完成？这个方法可行，但工作量非常大；那有没有更好的解决方案呢？我们通过分析Activity的启动过程，发现Activity是由

ActivityThread 通过Instrumentation来启动的，我们是否可以在Instrumentation中做一定的手脚呢？通过分析代码ActivityThread和Instrumentation发现，Instrumentation有关Activity启动相关的方法大概有：execStartActivity、newActivity等等，这样我们就可以在这些方法中添加代码逻辑进行判断这个Class是否加载了，如果加载则直接启动这个Activity，如果没有加载完成则启动一个等待的Activity显示给用户，然后在这个Activity中等待后台Secondary DEX加载完成，完成后自动跳转到用户实际要跳转的Activity；这样在代码充分解耦合，以及每个业务代码能够做到颗粒化的前提下，我们就做到Secondary DEX的按需加载了，下面是Instrumentation添加的部分关键代码：

```

public ActivityResult execStartActivity(Context who, IBinder contextThread, IBinder token,
 Intent intent, int requestCode) {
 ActivityResult activityResult = null;
 String className;
 if (intent.getComponent() != null) {
 className = intent.getComponent().getClassName();
 } else {
 ResolveInfo resolveActivity = who.getPackageManager().resolveActivity(intent,
 PackageManager.MATCH_DEFAULT_ONLY);
 if (resolveActivity != null && resolveActivity.activityInfo != null) {
 className = resolveActivity.activityInfo.name;
 } else {
 className = null;
 }
 }

 if (!TextUtils.isEmpty(className)) {
 boolean shouldInterrupted = !MeituanApplication.isDexAvailable();
 if (MeituanApplication.sIsDexAvailable.get() || mByPassActivityClassNameList.contains(className))
 shouldInterrupted = false;
 }
 if (shouldInterrupted) {
 Intent interruptedIntent = new Intent(mContext, WaitingActivity.class);
 activityResult = execStartActivity(who, contextThread, token, target, interruptedIntent);
 } else {
 activityResult = execStartActivity(who, contextThread, token, target, intent);
 }

 return activityResult;
}

public Activity newActivity(Class<?> clazz, Context context, IBinder token,
 Application application, Intent intent, ActivityInfo info,
 CharSequence title, Activity parent, String id, Object
 throws InstantiationException, IllegalAccessException {
}

```

```
String className = "";
Activity newActivity = null;
if (intent.getComponent() != null) {
 className = intent.getComponent().getClassName();
}

boolean shouldInterrupted = !MeituanApplication.isDexAvailable();
if (MeituanApplication.sIsDexAvailable.get() || mByPassActivityClassNameList.contains(className)) {
 shouldInterrupted = false;
}
if (shouldInterrupted) {
 intent = new Intent(mContext, WaitingActivity.class);
 newActivity = mBase.newActivity(clazz, context, token,
 application, intent, info, title, parent, id,
 lastNonConfigurationInstance);
} else {
 newActivity = mBase.newActivity(clazz, context, token,
 application, intent, info, title, parent, id,
 lastNonConfigurationInstance);
}
return newActivity;
}
```

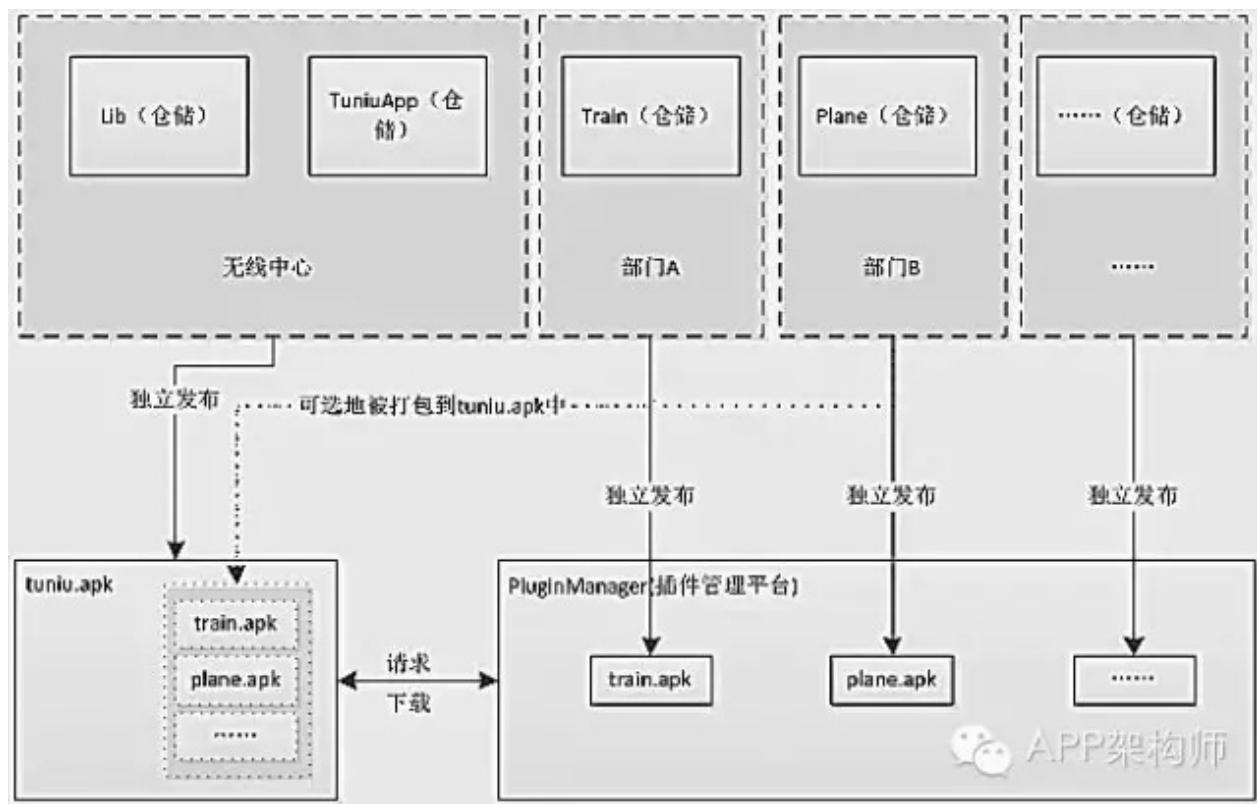
实际应用中我们还遇到另外一个比较棘手的问题，就是Field的过多的问题，Field过多是我们目前采用的代码组织结构引入的，我们为了方便多业务线、多团队并发协作的情况下开发，我们采用的aar的方式进行开发，并同时在aar依赖链的最底层引入了一个通用业务aar，而这个通用业务aar中包含了很多资源，而ADT14以及更高的版本中对Library资源处理时，Library的R资源不再是static final的了，详情请查看[google官方说明](#)，这样在最终打包时Library中的R没法做到内联，这样带来了R field过多的情况，导致需要拆分多个Secondary DEX，为了解决这个问题我们采用的是在打包过程中利用脚本把Libray中R field（例如ID、Layout、Drawable等）的引用替换成常量，然后删去Library中R.class中的相应Field。

## 总结

上面就是我们在使用MultiDex过程中进化而来的DEX自动化拆包的方案，这样我们就可以通过脚本控制来进行自动化的拆分DEX，然后在运行时自由的加载Secondary DEX，既能保证冷启动速度，又能减少运行时的内存占用。

# 途牛原创|途牛Android App的插件实现

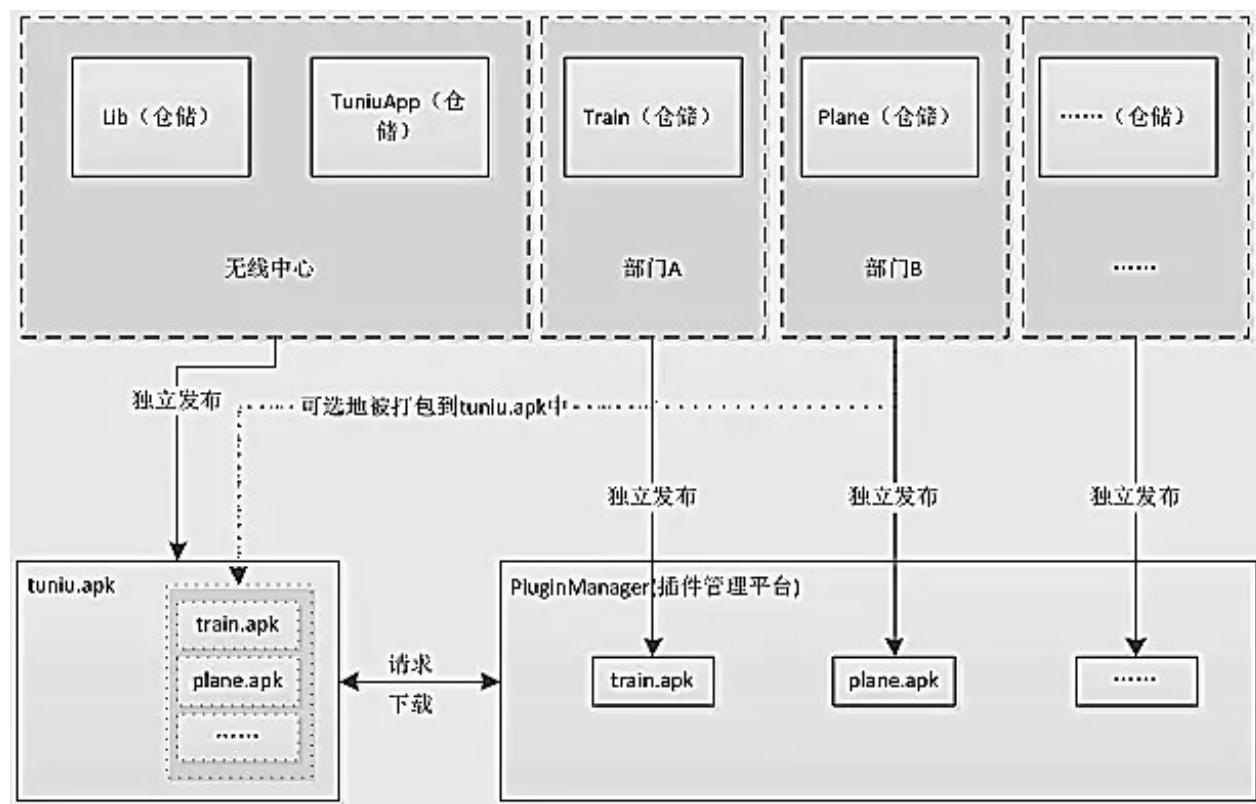
来源：[途牛原创|途牛Android App的插件实现](#)



## 途牛的Android App插件化

途牛的插件化是基于dynamic-load-apk (github) 实现的。定义了宿主和插件的通信方式，使得两者能够互起对方的页面，调用彼此的功能。同时对activity的启动方式 singletask等进行了模式实现，并增加了对Service的支持等。总之使得插件开发最大限度的保持着原有的Android开发习惯。

然后，我们看下引入插件化后，途牛的组织架构，代码管理及版本发布方式：

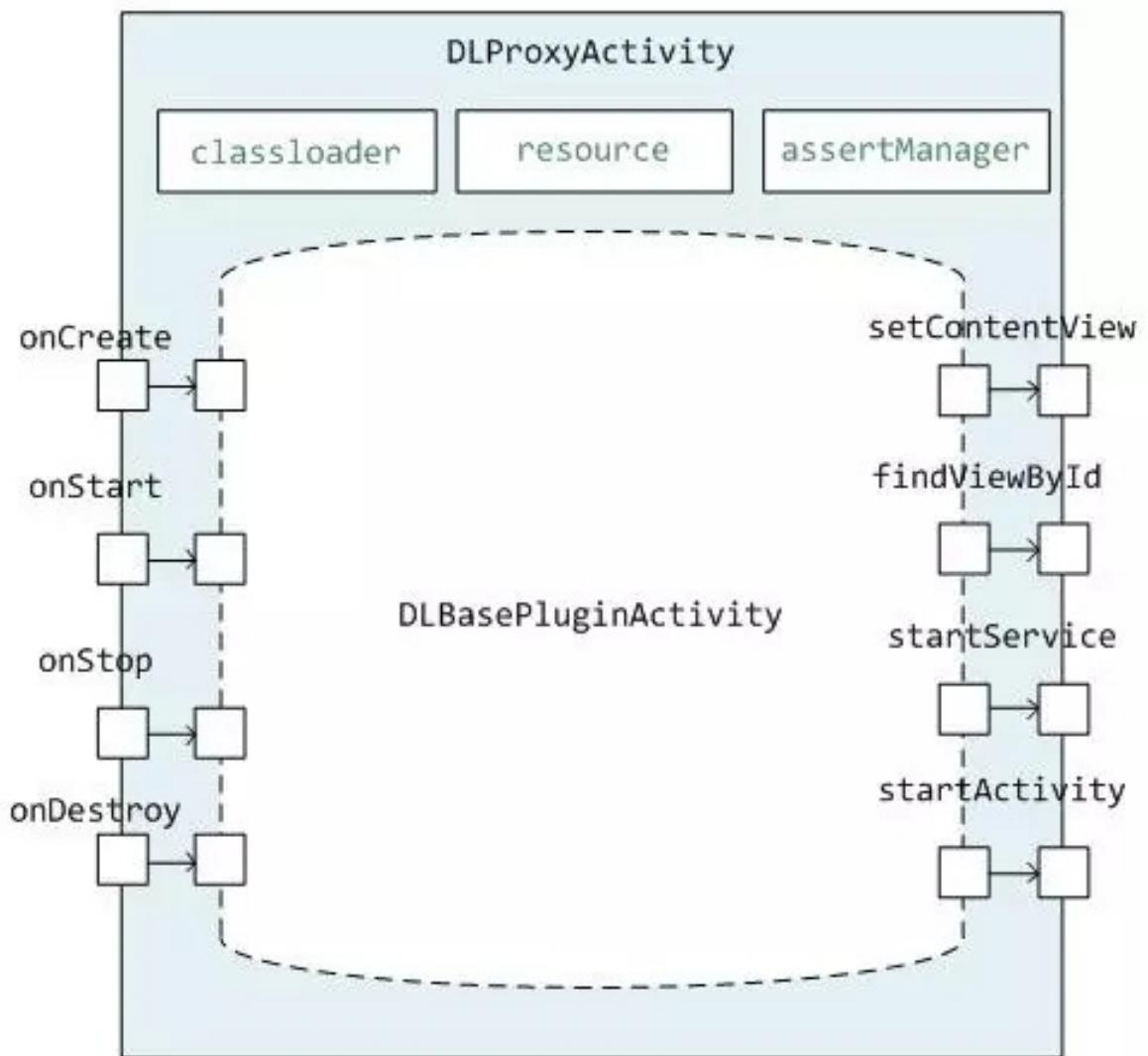


从图中可以看出，部门独立出后，代码也随之独立出，再无强依赖，部门可以根据自己的需求快速响应，独立发版，再也不用依赖于宿主app的版本周期。这样带来的好处，我想大家都是懂的。

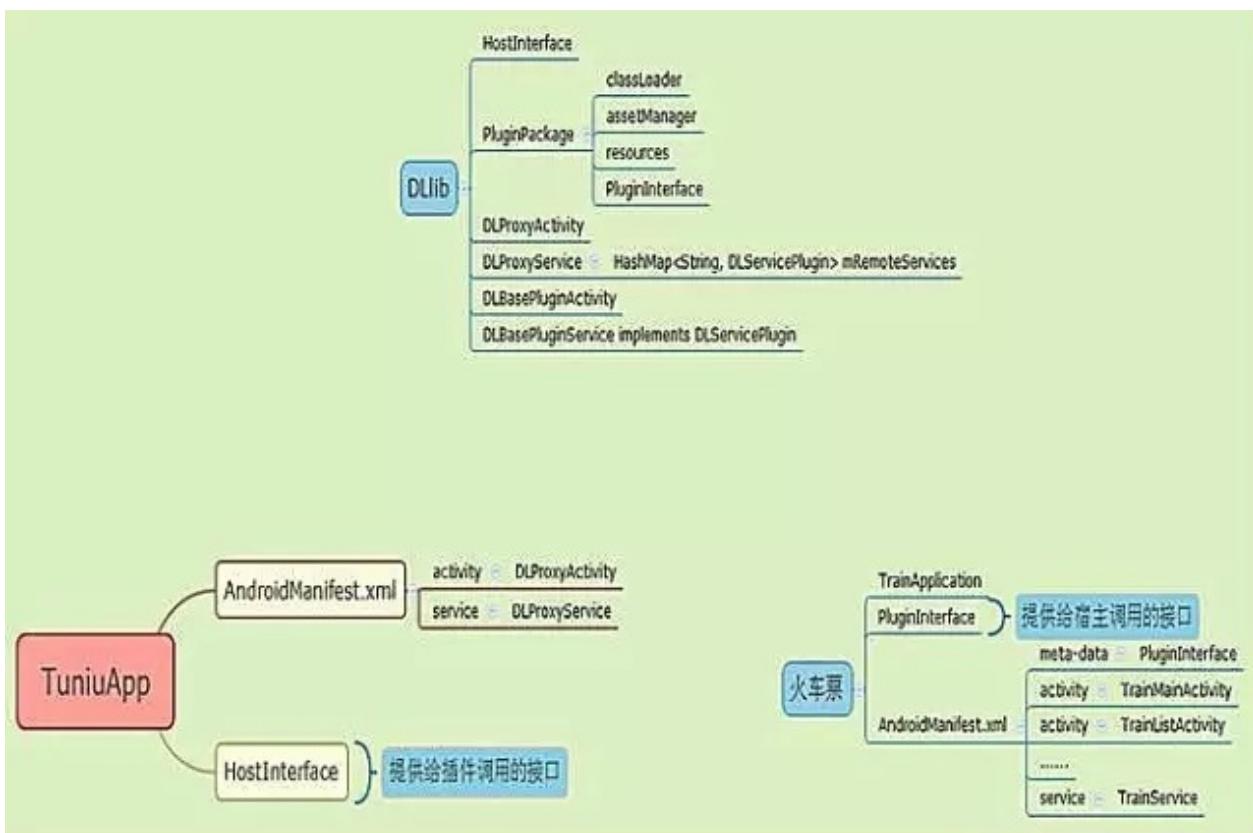
## 技术实现

我们来看下技术实现：

关于代理Activity的概念这里就不再赘述，详细请参阅[dynamic-load-apk \(github\)](#) [参见本文最后]。这里给一个代理Activity和插件activity的关系图。



下面主要描述下对其进行的扩展支持。先上图，如下：



## 1. 插件框架对于原生功能支持的实现

### a) 宿主App和插件的通讯方式定义

宿主App和插件的通讯方式定义使用桥接模式，分别为宿主和插件定义了接口（HostInterface, PluginInterface），各自对接口进行实现。宿主app在启动时将其接口实现set到DLlib中。插件将接口实现发布在其AndroidMainfest.xml中，并在宿主app初始化插件包时，通过反射获取该实现set到DLlib中。这样以来，两者可实现相互的界面跳转，及功能函数的调用。

### b) Activity启动方式的模拟支持

Activity启动方式的模拟支持通过对代理Activity原理的了解，我们都会发现启动的插件页面实际上都是DLProxyActivity，插件activity只是这个代理Activity的一个成员对象，基于这个一对一的映射关系，我们为插件activity建立独立的task管理，通过监控DLProxyActivity的生命周期，来模拟实现插件activity的singletask, singleinstance的实现。

### c) Service支持

Service支持参照代理Activity思想，同样设置代理Service。这个代理Service中维护着一个插件service的列表。每次向插件Service发送请求，实际上仍然是通过反射方式向代理Service发送请求，代理Service再进行分发。

### d) 通过对代理Activity的监控，增加插件页面在被回收后，能够由系统自动恢复的支持。

e)其它细节不再赘述。

## 2. 宿主App和插件app的打包和发布方式

- a) 宿主App打包时， compile依赖Lib库并一起打包成apk。
- b) 插件打包时， provide依赖Lib库， 打包出的apk中不包含Lib库的内容。

## 3. 插件app的开发调测

- a) 插件app可更改Lib库的依赖为compile， 打包生成可安装运行apk， 进行开发调测。
- b) 插件app依赖宿主app进行开发调测。更改宿主app加载插件路径为自己方便拷贝的路径， 然后插件app打包进行替换。需要debug时， 可直接添加断点， 进入debug模式。

## 4. 关于插件管理平台

- a) 插件管理以宿主app的版本为key， 进行管理。每个宿主app有其对应支持的插件列表。
- b) 插件的更新采用增量更新模式。
- c) 对于下载的插件包或增量包进行签名校验

## 对比

最后来做个对比，说下不足：

### 1. DynamicAPK（携程）

- a) 看到插件的Activity需要注册在宿主app的AndroidManifest.xml里， 我就不喜欢了。需求上不希望插件增加一个界面， 主app要跟着发版。
- b) 打包要改aapt， 插件编译要依赖宿主app， 主app发版要合并R文件， 这还是要捆绑发版， 不喜欢。
- c) Hook系统Instrumentation， 这个很厉害， 值得借鉴， 看能否在此层面完成插件activity启动模式的支持。
- d) 迁移成本极其小， 值得赞扬。

### 2. DroidPlugin（奇虎360）

这个不满足需求，途牛app需要宿主app和插件件进行频繁复杂的交互。直接pass。

# 优势与不足

## 不足：

- a) that的存在，导致迁移时要进行适当培训。
- b) 不支持插件自定义scheme应用到插件activity上。
- c) 不支持宿主FragmentActivity + 宿主Fragement + 插件Fragement这样的混合使用。
- d) 不支持插件ContentProvider。
- e) 不支持插件静态广播。
- f)不支持对Lib和宿主app中的资源访问.

## 优势：

再次说下优势，做个宣传。

- 最大的优势：插件增加或减少页面，宿主app不需要做任何修改，而且插件可独立发布。

综上，算是选择了一个最适合需求的框架。

备注：dynamic-load-apk (github) 的链接：

<https://github.com/singwhatiwanna/dynamic-load-apk>

# 那些年蘑菇街Android组件与插件化背后的故事 -- 插件篇(一)

来源:[那些年蘑菇街Android组件与插件化背后的故事 -- 插件篇\(一\)](#)

## 插件化的基石 -- apk动态加载

随着我街业务的蓬勃发展，产品和运营随时上新功能新活动的需求越来越强烈，经常可以听到“有个功能我想周x上，行不行”。行么？当然是不行啦，上新功能得发新版本啊，到时候费时费力打乱开发节奏不说，覆盖率也是个问题。苏格拉底曾经说过：“现在移动端的主要矛盾是产品日益增长的功能需求与平台落后的发布流程之间的矛盾”。

当然，作为一个靠谱的程序猿，我们就是为了满足产品的需求而存在的(正义脸)。于是在一个阳光明媚的早晨，吃完公司的免费早餐后，我和小强、叶开，决定做一个完善的Android动态加载框架。

Android动态加载技术在蘑菇街的第一次实践，还是在14年的时候，使用的就是之前网上广(tu)为(du)流(si)传(fang)的方式，这种方式有一个重大缺陷，就是插件内部对资源的访问只能通过自己定义的方式，包括对layout文件的inflate等，使用getResources的方式，分分钟crash给你看，而且内部实现有些复杂，容易出现莫名其妙的ResourcesNotFound错误。在一段时间的使用之后，始终无法大面积推广，原因就是对开发人员来说，写一个“正常”的模块和写一个动态加载模块，写法是不一样的。这件事一直如鲠在喉，如果这个框架无法做到对开发业务的同学们透明，那么就很难推广开来。如何做到对业务开发者透明呢，最重要的是对于各类系统api的使用，尤其是Android四大组件的使用和资源访问，都要遵循系统提供的方式。

抛开上面的东西，从头开始讲述一下动态加载的原理：

**Android**应用程序的.java文件在编译期会通过javac命令编译成.class文件，最后再把所有的.class文件编译成.dex文件放在.apk包里面。那么动态加载就是在运行时把插件apk直接加载到classloader里面的技术。

看完上面的原理，不知道你有没有什么疑问，反正我是有的。

- 如何加载插件里面的.dex文件。
- apk里面的资源怎么办。

上面两个问题是动态加载框架最重要的两点，无法动态安装dex或资源文件的动态加载框架都是要流氓。我们在实现这个框架的时候同样也遇到了这两个问题。

## 如何动态加载插件代码：

关于代码加载，系统提供了[DexClassLoader](#)来加载插件代码。开发者可以对每一个插件分配一个[DexClassLoader](#)(这是目前最常见的一种方式)，也可以动态得把插件加载到当前运行环境的classloader中。蘑菇街采用的是后者，这种方式可以有效的防止各种莫名其妙的[ClassCastException](#)，当你在crash后台看到各种 A cast A错误而欲哭无泪的时候，我想你会喜欢上这种方式。

事情当然不会这么简单，系统提供的DexClassLoader对外api中，只有一种方式可以向类加载器指定加载路径。就是在构造函数中传入apk/zip/dex路径。这完全不符合我们“动态”的原则，难道每次加载一个插件，都必须重新实例化一个类加载器出来吗？这个时候我们想到了google提供的multidex插件，这个插件旨在帮助函数超过65536上限的应用在编译期切割class到多个dex文件中。经过观察发现，5.0以下的Android系统，在应用安装的时候只认classes.dex文件，并在安装期对这个dex文件进行opt操作，生成的odex文件放在/data/dalvik-cache里面。那么剩下classes(N).dex怎么办呢，答案就是如果在编译期使用multidex插件的话，开发者还需要让自己的Application继承[MultiDexApplication](#)，这样想起来，这个[MultiDexApplication](#)应该就有加载剩下的classes(N).dex的能力了。查看[MultiDexApplication](#)代码，果然找到了线索：

```
public class MultiDexApplication extends Application {
 @Override
 protected void attachBaseContext(Context base) {
 super.attachBaseContext(base);
 MultiDex.install(this);
 }
}
```

可以看到，它在attachBaseContext函数调用了support包中[MultiDex](#)类的install函数来安装classes(N).dex，于是都是应用层代码，它能动态安装那表示我们也可以。有了以上的分析，剩下要做的就只是去扒一扒install这个函数了。

## 如何动态加载插件资源：

我们在开发的时候，当有需要用到资源的地方，可以直接调用[Context](#)的getResources()函数返回[Resources](#)的来访问打包在apk中的资源文件。在研究如何动态添加资源到系统的[Resources](#)对象的时候，有必要先了解一下[Resources](#)本身是如何访问到资源的。

查看系统的Resources源码，我们发现这个类主要做了两件事，首当其冲的当然是访问资源，另外一件就是管理资源配置信息。对于资源的动态加载来说，我们关心的是它如何做第一件事的。实际上，**Resources**对资源的访问，全部代理给了另一个重要的对象**AssetManager**。那么问题转化成了，**AssetManager**是如何做到对资源的访**Resources**类在它的构造函数里对**AssetManager**做了一些重要的初始化：

```
public Resources(AssetManager assets, DisplayMetrics metrics, Configuration config,
 CompatibilityInfo compatInfo, IBinder token) {
 mAssets = assets;
 mMetrics.setToDefaults();
 if (compatInfo != null) {
 mCompatibilityInfo = compatInfo;
 }
 mToken = new WeakReference<IBinder>(token);
 updateConfiguration(config, metrics);
 assets.ensureStringBlocks();
}
```

其中的重点就是调用了**AssetManager**对象的 `ensureStringBlocks()` 函数，这个函数的实现如下：

```
/*package*/ final void ensureStringBlocks() {
 if (mStringBlocks == null) {
 synchronized (this) {
 if (mStringBlocks == null) {
 makeStringBlocks(sSystem.mStringBlocks);
 }
 }
 }
}
```

函数先判断 `mStringBlocks` 变量是否为空，如果不为空的话，表示需要被初始化，于是调用 `makeStringBlocks` 函数初始化 `mStringBlocks`：

```

/*package*/ final void makeStringBlocks(StringBlock[] seed) {
 final int seedNum = (seed != null) ? seed.length : 0;
 final int num = getStringBlockCount();
 mStringBlocks = new StringBlock[num];
 if (localLOGV) Log.v(TAG, "Making string blocks for " + this
 + ": " + num);
 for (int i=0; i<num; i++) {
 if (i < seedNum) {
 mStringBlocks[i] = seed[i];
 } else {
 mStringBlocks[i] = new StringBlock(getNativeStringBlock(i), true);
 }
 }
}

```

这里的mStringBlocks对象是一个StringBlock数组，这个类被标记为@hide，表示应用层根本不需要关心它的存在。那么它是做什么用的呢，它就是AssetManager能够访问资源的奥秘所在，AssetManager所有访问资源的函数，例如getResourceTextArray()，都最终通过StringBlock再代理到native进行访问的。看到这里，依然没有任何看到能够指示为什么开发者可以访问自己应用的资源，那么我们再看得前面一点，看看传入Resources的构造函数之前，asset参数是不是被“做过手脚”。函数调用辗转到ResourceManager的getTopLevelResources 函数：

```

public Resources getTopLevelResources(String resDir, String[] splitResDirs,
 String[] overlayDirs, String[] libDirs, int displayId,
 Configuration overrideConfiguration, CompatibilityInfo compatInfo, IB...
...
AssetManager assets = new AssetManager();
if (resDir != null) {
 if (assets.addAssetPath(resDir) == 0) {
 return null;
 }
}
...
}

```

函数代码有点多，截取最重要的部分，那就是系统通过调用AssetManager的addAssetPath 函数，将需要加载的资源路径加了进去。addAssetPath函数返回一个int类型，它指示了每个被添加的资源路径在native层一个数组中的位置，这个数组保存了系统资源路径(framework-res.apk)，和应用自己添加的所有资源路径。再回过头看makeStringBlocks函数，就豁然开朗了：

- makeStringBlocks函数的参数也是一个StringBlock数组，它表示系统资源，首先它调

用getStringBlockCount函数得到当前应用所有要加载的资源路径数量。

- 然后进入循环，如果属于系统资源，就直接用传入参数seed中的对象来赋值。
- 如果是应用自己的资源，就实例化一个新的StringBlock对象来赋值。并在StringBlock的构造函数中调用getNativeStringBlock函数来获取一个native层的对象指针，这个指针被java层StringBlock对象用来调用native函数，最终达到访问资源的目的。

有兴趣的同学可以继续深入native层的源码，可以看到不管是addAssetPath函数还是makeStringBlocks函数，使用的都是native层同一个数组，这样，这两个函数就被关联了起来。

到这里，我们已经知道了如何动态添加资源路径的“秘密”。

解决了以上两个问题，一个基本满足要求的动态加载框架就被搭了起来。

关于如何延迟加载组件的问题，请期待下一期的那些年蘑菇街Android组件与插件化背后的故事。

ps:查看native层Resources.cpp的代码，我们发现，Android5.0及以上版本是真正的支持动态添加资源路径到系统Resources对象，直接反射调用getAsset.addAssetPath即可。5.0以下版本只是“伪动态”，需要自己重新实例化一个Resources对象和AssetManager对象，添加完所有需要的资源路径后，替换运行环境的Resources对象才可以做到“动态”。这个跟5.0以下的Resources.cpp在初始化完成之后，无法动态扩展resTable有关。

# 那些年蘑菇街Android组件与插件化背后的故事 -- 插件篇(二)

来源:[那些年蘑菇街Android组件与插件化背后的故事 -- 插件篇\(二\)](#)

这阵子比较忙，组里的赤(xing)木(xing)大(dui)神(zhang)一直催我写第二篇。今天抽出空来，把这篇补上。

上一篇已经交代了如何把一个插件apk动态加载到内存中，并做到可用。这篇介绍的是，我们如何在“合适”的时间加载某个插件。试想一个理想化的场景，一个app中，所有的模块都是一个个的插件，我们不可能在应用初始化的时候把所有的插件全部加载，dex optimize时长都能把人弄哭。

那么就得找一个时间点，使我们可以知道，某个插件第一次被用到是什么时候，只要在这个时间加载这个插件就好。分析一下插件主要有哪些东西构成：

- 首先是逻辑代码，这些函数有些是自己插件内部用的，有些是外部其他插件也要用的。
- 页面，各种Activity。

Ok，我们一个个得解决上面的问题

## 对外函数

在这种情况下，如果一个插件A需要调用插件B的函数，我们可以去检查插件B是否被加载过，如果还未加载，就加载它。这里不知道大家会不会有个疑问，我们是如何感知到插件A调用了插件B中的函数了呢？

首先有个天然性的隔离就是，插件与插件之间在开发的时候，都是独立工程的，它们之间不应该存在直接的函数调用。

然后，我们需要一个中间管理框架，用来统一处理插件间的函数(服务)调用，这也是蘑菇街组件管理框架所做的很重要的一件事。所有的插件/组件之间的函数(服务)调用，必须过管理框架中转，最后由框架进行实际的调用。

由于管理框架是一个通用框架，并不针对某个插件，所以插件所能提供的对外函数，要使用配置的方式告知外部和管理框架，未在配置定义的函数是不允许被调用的。

于是我们可以拦截到插件间的函数(服务)调用了，这是第一个可以检测到某个插件是否开始被使用的地方。

## 页面，Activity

另外一种常见情况就是插件A需要启动插件B中的一个Activity。为了让开发插件的同学最小化得感知自己在开发一个插件，业务开发的同学依然可以使用任何系统提供的任何启动Activity的方式。但是作为需要知道启动了哪个Activity以便去安装某个插件的管理框架来说，需要对这个操作进行拦截。

又到了read the fuck source code的时间啦~

首先我们从一个常见的startActivity函数调用开始，不论调用了哪个Context的startActivity函数，最后都会调用到ContextImpl的startActivity函数：

```
@Override
public void startActivityForResult(Intent intent, Bundle options) {
 ...
 mMainThread.getInstrumentation().execStartActivity(
 getOuterContext(), mMainThread.getApplicationThread(), null,
 (Activity)null, intent, -1, options);
}
```

看到它最后调用了mMainThread.getInstrumentation().execStartActivity来启动一个Activity，mMainThread是一个ActivityThread类型的对象，getInstrumentation函数返回一个Instrumentation类型的对象。顺理成章得，接下去是Instrumentation类的execStartActivity函数：

```

public ActivityResult execStartActivity(
 Context who, IBinder contextThread, IBinder token, Activity target,
 Intent intent, int requestCode, Bundle options) {
 ...
 try {
 ...
 int result = ActivityManagerNative.getDefault()
 .startActivity(whoThread, who.getPackageName(), intent,
 intent.resolveTypeIfNeeded(who.getContentResolver()),
 token, target != null ? target.mEmbeddedID : null,
 requestCode, 0, null, options);
 checkStartActivityResult(result, intent);
 } catch (RemoteException e) {
 }
 return null;
}

```

调用到了ActivityManagerNative的startActivity函数，这是一个IPC调用，用于向远端的ActivityManagerService发起一个start activity的请求。当然大家可以拦截Instrumentation的execStartActivity函数，因为通过函数的入参，已经知道了应用层请求启动哪个Activity了。但拦截这个函数有几个弊端，

- 第一就是execStartActivity函数只是启动Activity的某一个函数，类似的函数还有一些，如果我们都把它们拦截了，显得有点冗余了。
- 第二个弊端就是这里只是向ActivityManagerService发起一个IPC请求，具体这个Activity能否被启动，都还是未知数，如果在这里拦截而这个Activity，也会造成一些浪费和不确定性。

另外我们可以非常肯定得认为，在整个Activity启动的过程中，实例化某个具体Activity的操作一定是在当前应用程序进程进行的，不可能由远端的ActivityManagerService来执行，因为只有当前应用程序进行的内存中，才会有某个具体Activity的类存在。

于是我们在ActivityThread类中找到了最后实例化某个Activity类的函数

```
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
 ...

 Activity activity = null;
 try {
 java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
 activity = mInstrumentation.newActivity(
 cl, component.getClassName(), r.intent);
 StrictMode.incrementExpectedActivityCount(activity.getClass());
 r.intent.setExtrasClassLoader(cl);
 r.intent.prepareToEnterProcess();
 if (r.state != null) {
 r.state.setClassLoader(cl);
 }
 } catch (Exception e) {
 if (!mInstrumentation.onException(activity, e)) {
 throw new RuntimeException(
 "Unable to instantiate activity " + component
 + ": " + e.toString(), e);
 }
 }
 ...

 return activity;
}
```

最关键的一句，就是调用mInstrumentation.newActivity来返回一个Activity对象，所以这里是最后实例化的地方，Instrumentation类的newActivity也就是我们需要拦截下来的操作。简单明了，只需要复写这个函数然后把自己的Instrumentation对象替换成运行上下文的就可以了~

到这里为止，Activity启动也被管理框架感知到了，自然可以去检测安装Activity所属的插件了。

以上就是蘑菇街组件与插件管理框架中插件安装的逻辑。



# 设计原则

# 面向对象编程的六大原则

来源:[CSDN](#)

## 单一职责原则(SRP,Single Responsibility Principle)

就一个类而言，应该只专注于做一件事和仅有一个引起它变化的原因。所谓职责，我们可以理解他为功能，就是设计的这个类功能应该只有一个，而不是两个或更多。也可以理解为引用变化的原因，当你发现有两个变化会要求我们修改这个类，那么你就要考虑拆分这个类了。因为职责是变化的一个轴线，当需求变化时，该变化会反映类的职责的变化。

注意点：

- 1、一个合理的类，应该仅有一个引起它变化的原因，即单一职责；
- 2、在没有变化征兆的情况下应用SRP或其他原则是不明智的；
- 3、在需求实际发生变化时就应该应用SRP等原则来重构代码；
- 4、使用测试驱动开发会迫使我们在设计出现臭味之前分离不合理代码；
- 5、如果测试不能迫使职责分离，僵化性和脆弱性的臭味会变得很强烈，那就应该用Facade或Proxy模式对代码重构

优点：

消除耦合，减小因需求变化引起代码僵化。

## 里氏代换原则(LSP,Liskov Substitution Principle)

子类型必须能够替换它们的基类型。一个软件实体如果使用的是一个基类，那么当把这个基类替换成继承该基类的子类，程序的行为不会发生任何变化。软件实体察觉不出基类对象和子类对象的区别。

优点：

可以很容易的实现同一父类下各个子类的互换，而客户端可以毫不察觉。

# 依赖倒置原则(DIP,Dependence Inversion Principle)

要依赖于抽象，不要依赖于具体。客户端依赖于抽象耦合。抽象不应当依赖于细节；细节应当依赖于抽象；要针对接口编程，不针对实现编程。

优点：

使用传统过程化程序设计所创建的依赖关系，策略依赖于细节，这是糟糕的，因为策略受到细节改变的影响。依赖倒

## 怎样做到依赖倒置？

以抽象方式耦合是依赖倒转原则的关键。抽象耦合关系总要涉及具体类从抽象类继承，并且需要保证在任何引用到基类的地方都可以改换成其子类，因此，里氏代换原则是依赖倒转原则的基础。

在抽象层次上的耦合虽然有灵活性，但也带来了额外的复杂性，如果一个具体类发生变化的可能性非常小，那么抽象耦合能发挥的好处便十分有限，这时可以用具体耦合反而会更好。

**层次化：**所有结构良好的面向对象构架都具有清晰的层次定义，每个层次通过一个定义良好的、受控的接口向外提供一组内聚的服务。

**依赖于抽象：**建议不依赖于具体类，即程序中所有的依赖关系都应该终止于抽象类或者接口。尽量做到：

- 1、任何变量都不应该持有一个指向具体类的指针或者引用。
- 2、任何类都不应该从具体类派生。
- 3、任何方法都不应该覆写它的任何基类中的已经实现的方法。

# 接口隔离原则(ISP,Interface Segregation Principle)

使用多个专一功能的接口比使用一个的总接口总要好。从一个客户类的角度来讲：一个类对另外一个类的依赖性应当是建立在最小接口上的。过于臃肿的接口是对接口的污染，不应该强迫客户依赖于它们不用的方法。

优点：

会使一个软件系统功能扩展时，修改的压力不会传到别的对象那里。

### 如何实现接口隔离原则？

- 1、不应该强迫用户依赖于他们不用的方法。
- 2、利用委托分离接口。
- 3、利用多继承分离接口。

## 迪米特法则(Law of Demeter,Least Knowledge Principle 最少知识原则 )

对象与对象之间应该使用尽可能少的方法来关联，避免千丝万缕的关系

**如何实现迪米特法则？** 迪米特法则的主要用意是控制信息的过载，在将其运用到系统设计中应注意以下几点：

- 1) 在类的划分上，应当创建有弱耦合的类。类之间的耦合越弱，就越有利于复用。
- 2) 在类的结构设计上，每一个类都应当尽量降低成员的访问权限。一个类不应当public自己的属性，而应当提供取
- 3) 在类的设计上，只要有可能，一个类应当设计成final类。
- 4) 在对其它对象的引用上，一个类对其它对象的引用应该降到最低。

## 开闭原则(OCS,Open Close Principle)

对扩展开放，对修改关闭

优点：

按照OCP原则设计出来的系统，降低了程序各部分之间的耦合性，其适应性、灵活性、稳定性都比较好。当已有软件

**如何实现“开-闭”原则？** 在面向对象设计中，不允许更改的是系统的抽象层，而允许扩展的是系统的实现层。换言之，定义一个一劳永逸的抽象设计层，允许尽可能多的行为在实现层被实现。

解决问题关键在于抽象化，抽象化是面向对象设计的第一个核心本质。

对一个事物抽象化，实质上是在概括归纳总结它的本质。抽象让我们抓住最最重要的东西，从更高一层去思考。这降在面向对象编程中，通过抽象类及接口，规定了具体类的特征作为抽象层，相对稳定，不需更改，从而满足“对修改对实体进行扩展时，不必改动软件的源代码或者二进制代码。关键在于抽象。

## 总结

你不必严格遵守这些原则，违背它们也不会被处以宗教刑罚。但你应当把这些原则看成警铃，若违背了其中的一条，那么警铃就会响起。 -----Arthur J.Riel

(1)所有数据都应该隐藏在所在的类的内部。

(2)类的使用者必须依赖类的共有接口，但类不能依赖它的使用者。p15

(3)尽量减少类的协议中的消息。

(4)实现所有类都理解的最基本公有接口[例如，拷贝操作(深拷贝和浅拷贝)、相等性判断、正确输出内容、从ASCII

(5)不要把实现细节(例如放置共用代码的私有函数)放到类的公有接口中。

如果类的两个方法有一段公共代码，那么就可以创建一个防止这些公共代码的私有函数。

(6)不要以用户无法使用或不感兴趣的东西扰乱类的公有接口。p17

(7)类之间应该零耦合，或者只有导出耦合关系。也即，一个类要同另一个类毫无关系，要么只使用另一个类的公

(8)类应该只表示一个关键抽象。

包中的所有类对于同一类性质的变化应该是共同封闭的。一个变化若对一个包影响，则将对包中的所有类产生影响，

(9)把相关的数据和行为集中放置。

设计者应当留意那些通过get之类操作从别的对象中获取数据的对象。这种类型的行为暗示着这条经验原则被违反了，

(10)把不相关的信息放在另一个类中(也即：互不沟通的行为)。p19

朝着稳定的方向进行依赖。

(11)确保你为之建模的抽象概念是类，而不只是对象扮演的角色。p23

(12)在水平方向上尽可能统一地分布系统功能，也即：按照设计，顶层类应当统一地共享工作。

(13)在你的系统中不要创建全能类/对象。对名字包含Driver、Manager、System、Subsystem的类要特别多加小心。规划一个接口而不是实现一个接口。

(14)对公共接口中定义了大量访问方法的类多加小心。大量访问方法意味着相关数据和行为没有集中存放。

(15)对包含太多互不沟通的行为的类多加小心。

这个问题的另一表现是在你的应用程序中的类的公有接口中创建了很多的get和set函数。

(16)在由同用户界面交互的面向对象模型构成的应用程序中，模型不应该依赖于界面，界面则应当依赖于模型。

(17)尽可能地按照现实世界建模(我们常常为了遵守系统功能分布原则、避免全能类原则以及集中放置相关数据和行

(18)从你的设计中去除不需要的类。

一般来说，我们会把这个类降级成一个属性。

(19)去除系统外的类。

系统外的类的特点是，抽象地看它们只往系统领域发送消息但并不接受系统领域内其他类发出的消息。

(20)不要把操作变成类。质疑任何名字是动词或者派生自动词的类，特别是只有一个有意义行为的类。考虑一下那个

(21)我们在创建应用程序的分析模型时常常引入代理类。在设计阶段，我们常会发现很多代理没有用的，应当去除。

(22)尽量减少类的协作者的数量。

一个类用到的其他类的数目应当尽量少。

(23)尽量减少类和协作者之间传递的消息的数量。

(24)尽量减少类和协作者之间的协作量，也即：减少类和协作者之间传递的不同消息的数量。

(25)尽量减少类的扇出，也即：减少类定义的消息数和发送的消息数的乘积

(26)如果类包含另一个类的对象，那么包含类应当给被包含的对象发送消息。也即：包含关系总是意味着使用关系。

(27)类中定义的大多数方法都应当在大多数时间里使用大多数数据成员。

(28)类包含的对象数目不应当超过开发者短期记忆的容量。这个数目常常是6。

当类包含多于6个数据成员时，可以把逻辑相关的数据成员划分为一组，然后用一个新的包含类去包含这一组成员。

(29)让系统功能在窄而深的继承体系中垂直分布。

(30)在实现语义约束时，最好根据类定义来实现。这常常会导致类泛滥成灾，在这种情况下，约束应当在类的行为中实现。

(31)在类的构造函数中实现语义约束时，把约束测试放在构造函数领域所允许的尽量深的包含层次中。

(32)约束所依赖的语义信息如果经常改变，那么最好放在一个集中式的第3方对象中。

(33)约束所依赖的语义信息如果很少改变，那么最好分布在约束所涉及的各个类中。

(34)类必须知道它包含什么，但是不能知道谁包含它。

(35)共享字面范围(也就是被同一个类所包含)的对象相互之间不应当有使用关系。

(36)继承只应被用来为特化层次结构建模。

(37)派生类必须知道基类，基类不应该知道关于它们的派生类的任何信息。

(38)基类中的所有数据都应当是私有的，不要使用保护数据。类的设计者永远都不应该把类的使用者不需要的东西放在基类中。

(39)在理论上，继承层次体系应当深一点，越深越好。

(40)在实践中，继承层次体系的深度不应当超出一个普通人的短期记忆能力。一个广为接受的深度值是6。

(41)所有的抽象类都应当是基类。

(42)所有的基类都应当是抽象类。

(43)把数据、行为和/或接口的共性尽可能地放到继承层次体系的高端。

(44)如果两个或更多个类共享公共数据(但没有公共行为)，那么应当把公共数据放在一个类中，每个共享这个数据的类都应当从一个表示了这个公共数据的基类中继承。

(45)如果两个或更多个类有共同的数据和行为(就是方法)，那么这些类的每一个都应当从一个表示了这些数据和方法的基类中继承。

(46)如果两个或更多个类共享公共接口(指的是消息，而不是方法)，那么只有他们需要被多态地使用时，他们才应当从一个表示了这个公共接口的基类中继承。

(47)对对象类型的显示的分情况分析一般是错误的。在大多数这样的情况下，设计者应当使用多态。

(48)对属性值的显示的分情况分析常常是错误的。类应当解耦合成为一个继承层次结构，每个属性值都被转换成一个消息，然后由继承层次体系的最底层类处理。

(49)不要通过继承关系来为类的动态语义建模。试图用静态语义关系来为动态语义建模会导致在运行时切换类型。

(50)不要把类的对象变成派生类。对任何只有一个实例的派生类都要多加小心。

(51)如果你觉得需要在运行时刻创建新的类，那么退后一步以认清你要创建的是对象。现在，把这些对象概括成一个类，然后从这个类派生出新的类。

(52)在派生类中用空方法(也就是什么也不做的方法)来覆写基类中的方法应当是非法的。

(53)不要把可选包含同对继承的需要相混淆。把可选包含建模成继承会带来泛滥成灾的类。

(54)在创建继承层次时，试着创建可复用的框架，而不是可复用的组件。

(55)如果你在设计中使用了多重继承，先假设你犯了错误。如果没犯错误，你需要设法证明。

(56)只要在面向对象设计中用到了继承，问自己两个问题：(1)派生类是否是它继承的那个东西的一个特殊类型？(2)派生类是否是它继承的那个东西的一个子集？

(57)如果你在一个面向对象设计中发现了多重继承关系，确保没有哪个基类实际上是另一个基类的派生类。

(58)在面向对象设计中如果你需要在包含关系和关联关系间作出选择，请选择包含关系。

(59)不要把全局数据或全局函数用于类的对象的薄记工作。应当使用类变量或类方法。

(60)面向对象设计者不应当让物理设计准则来破坏他们的逻辑设计。但是，在对逻辑设计作出决策的过程中我们经常需要考虑物理设计的准则。

(61)不要绕开公共接口去修改对象的状态。

# S 代表着单一职责原则

来源:[Realm](#)

这是五个系列文章的第一部分，关于 [SOLID 原则](#))。

SOLID 是面向对象设计五原则的缩写：

- 单一职责原则（本文）
- 开-闭原则
- 里氏替换原则
- 接口隔离原则
- 依赖倒置原则
- 迪米特法则(*Law of Demeter,Least Knowledge Principle* 最少知识原则 )

过去几周，我深入谈及到了每个原则，解释它们的含义，它们和安卓开发的关系。在这个系列的结尾，你会牢固地掌握这些核心原则的含义，理解为什么它们对安卓开发如此重要和你该如何在每天的安卓开发中使用它们。

## SOLID 背景

SOLID 最初始于 2000 年左右，由 [Robert Martin \(AKA: Uncle Bob\)](#) 和 Michael Feathers 一起提出。当这五个基础的面向对象的设计原则一起提出的时候，它帮助开发者来开发可维护和可扩展的软件。

如果你对 Uncle Bob 或 Michael Feathers 不熟悉，我强烈推荐你看看他们的一些著作。Uncle Bob's 的著作 “[Agile Software Development, Principles, Patterns and Practices](#)” 和 “[Clean Code](#)” 是软件社区的名著。Michael Feathers 的书 “[Working Effectively with Legacy Code](#)” 是我带团队的时候要求所有开发者必读的著作。它帮助你思考如何改造你的旧代码，并且使它的可维护性强起来。更重要的是，它帮你重构你对遗留代码的理解。提示 - 你的代码有测试吗？没有！？！好吧，你的代码可能已经……你认为是……遗留代码了。

阅读这些书在我的职业生涯里起了非常关键的作用，所以我建议每个开发者把这些书放在他们的读书列表里，然后把它们放在书架上以备不时地查看。

我记得我个人是在2003年的多个 .NET 项目里使用了这些 SOLID 原则。那个时候，SOLID 原则好似一阵清风，因为我的 .NET 代码在没有什么架构和指导的情况下，已经变成一团糟了。这不仅仅是 .NET 的问题，在新技术出现的时候，它常常出现（比如，安卓

移动开发，还有其他）。新技术最终会达到一个成熟的水平来适用于这些 SOLID 讨论，讨论讨论它如何并且为什么这么重要。

最近，Uncle Bob 的 [Clean Architecture talk](#) 在安卓开发社区里重新热起来，所以我觉得现在是时候解释一下 Uncle Bob 在他的书里面提出的一些基本原则。这个系列的文章会谈及 [SOLID](#) 原则 和它们与安卓开发是如何联系起来的。

## 第一部分：单一职责原则

单一原则十分好理解。它如下描述：

**一个类应该有且只有一个发生改变的原因。**

让我们看看这个例子 [RecyclerView](#) 和它的 [adapter](#)。正如你所知道的那样，一个 RecyclerView 是一个灵活的视图，它可以显示一组数据到屏幕上。为了让这些数据显示到屏幕上，我们需要一个 RecyclerView adapter。

一个 adapter 从它的数据组里获取数据，然后和视图做匹配。一个适配器最有用的部分按理说就是 `onBindViewHolder` 方法（有时候是 ViewHolder 本身，但是为了简洁起见，我们仅仅关注 `onBindViewHolder`）。RecyclerView 的适配器有一个原则：把一个对象和它相关的需要显示在屏幕上的视图对应起来。

假设有这些对象，`RecyclerView.Adapter` 按如下实现：

```
public class LineItem {
 private String description;
 private int quantity;
 private long price;
 // ... getters/setters

}

public class Order {
 private int orderNumber;
 private List<LineItem> lineItems = new ArrayList<LineItem>();
 // ... getters/setters

}

public class OrderRecyclerAdapter extends RecyclerView.Adapter<OrderRecyclerAdapter.V.

 private List<Order> items;
 private int itemLayout;

 public OrderRecyclerAdapter(List<Order> items, int itemLayout) {
 this.items = items;
 this.itemLayout = itemLayout;
 }

 @Override
 public OrderRecyclerAdapter.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
 View view = LayoutInflater.from(parent.getContext()).inflate(itemLayout, parent, false);
 return new ViewHolder(view);
 }

 @Override
 public void onBindViewHolder(OrderRecyclerAdapter.ViewHolder holder, int position) {
 Order order = items.get(position);
 LineItem lineItem = order.getLineItems().get(0);
 holder.bind(lineItem);
 }

 @Override
 public int getItemCount() {
 return items.size();
 }

 static class ViewHolder extends RecyclerView.ViewHolder {
 // ...
 }
}
```

```

 this.items = items;
 this.itemLayout = itemLayout;
 }

 @Override public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
 View v = LayoutInflater.from(parent.getContext()).inflate(itemLayout, parent,
 return new ViewHolder(v);
 }

 @Override public void onBindViewHolder(ViewHolder holder, int position) {
 // TODO: bind the view here
 }

 @Override public int getItemCount() {
 return items.size();
 }

 public static class ViewHolder extends RecyclerView.ViewHolder {
 public TextView orderNumber;
 public TextView orderTotal;

 public ViewHolder(View itemView) {
 super(itemView);
 orderNumber = (TextView) itemView.findViewById(R.id.order_number);
 orderTotal = (ImageView) itemView.findViewById(R.id.order_total);
 }
 }
}

```

在上面的例子中，`onBindViewHolder` 是空的。我看到过的一个实现可能看起来如下：

```

@Override
public void onBindViewHolder(ViewHolder holder, int position) {
 Order order = items.get(position);
 holder.orderNumber.setText(order.getOrderNumber().toString());
 long total = 0;
 for (LineItem item : order.getItems()) {
 total += item.getPrice();
 }
 NumberFormat formatter = NumberFormat.getCurrencyInstance(Locale.US);
 String totalValue = formatter.format(cents / 100.0); // Must divide by a double or
 holder.orderTotal.setText(totalValue)
 holder.itemView.setTag(order);
}

```

这段代码违反了单一职责的原则。

为什么？

适配器的 `onBindViewHolder` 方法不仅仅是实现 `Order` 对象到视图的对应关系，它还完成了价格计算和格式化的功能。这违反了单一职责原则。适配器应该仅仅负责适配一个 `order` 对象到它的视图表示。`onBindViewHolder` 执行了本不属于它的两个额外功能。

为什么这是个问题？

一个类中包含多个原则会导致许多问题。

- 第一，计算逻辑现在和适配器耦合了。如果你需要在别处显示一个 `order` 的总数（你很有可能需要这样做），你就会有重复的逻辑了。一旦如此，你的应用就会有一个我们都熟悉的传统软件的冗余逻辑的问题。比如，你在一个地方更新了逻辑而忘记了在另一个地方做同样的事情。你明白了。
- 第二个问题和第一个问题类似 - 你耦合了格式化逻辑和适配器。如果你需要删除或者更新呢？最终，我们会让类完成比它应该有的责任更多的工作，而且现在的应用会更容易引起缺陷，因为在一个地方有了太多的责任。

谢天谢地，这个简单的例子可以很容易的改进，只需要把 `order` 总数计算的逻辑抽取出来放到 `Order` 对象里，然后把货币格式的功能移到一个货币格式化的类中就可以了。这个格式化类也可以被 `Order` 使用。

一个更新后的 `onBindViewHolder` 方法看起来像这样：

```
@Override
public void onBindViewHolder(ViewHolder holder, int position) {
 Order order = items.get(position);
 holder.orderNumber.setText(order.getOrderNumber().toString());
 holder.orderTotal.setText(order.getOrderTotal()); // A String, the calculation and
 holder.itemView.setTag(order);
}
```

我肯定你可能在想“好吧，这很简单。这太简单了不是吗？”它总是这么简单吗？在软件实际中大多数的答案是 - “好吧，这取决于实际情况……”。

让我们再深入一点……

## “责任”到底意味着什么？

没有谁比 Uncle Bob 说的更好了，让我直接引用如下：

**在单一职责原则（SRP）中我们这样定义职责，“改变的原因”。如果你能想到多于一个改变类的动机，那么那个类就有多于一个的责任。**

实际情况是，有些时候这很难看出来 - 特别是你在代码里沉寂了很久以后。这个时候，一句名言能提醒你：

**只见树木，不见森林。**

在软件上下文中，这句话意思是对你太关注你的代码的细节而忘记了整体。例如 - 你工作的类看起来很棒，但是那是因为你已经很熟悉这些类了，以至于你很难看到它已经有了多个责任了。

挑战是：你什么时候运用 SRP，什么时候不用。拿刚才的适配器举例，如果你再审视一边代码，你看到各种不同的改进点，在不同的地方因为不同的原因都有必要发生改变。

```
public class OrderRecyclerAdapter extends RecyclerView.Adapter<OrderRecyclerAdapter.V...{

 private List<Order> items;
 private int itemLayout;

 public OrderRecyclerAdapter(List<Order> items, int itemLayout) {
 this.items = items;
 this.itemLayout = itemLayout;
 }

 @Override public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
 View v = LayoutInflater.from(parent.getContext()).inflate(itemLayout, parent,
 return new ViewHolder(v);
 }

 @Override public void onBindViewHolder(ViewHolder holder, int position) {
 Order order = items.get(position);
 holder.orderNumber.setText(order.getOrderNumber().toString());
 holder.orderTotal.setText(order.getOrderTotal()); // Move the calculation and
 holder.itemView.setTag(order);
 }

 @Override public int getItemCount() {
 return items.size();
 }

 public static class ViewHolder extends RecyclerView.ViewHolder {
 public TextView orderNumber;
 public TextView orderTotal;

 public ViewHolder(View itemView) {
 super(itemView);
 orderNumber = (TextView) itemView.findViewById(R.id.order_number);
 orderTotal = (ImageView) itemView.findViewById(R.id.order_total);
 }
 }
}
```

适配器渲染一个视图，它把一个 order 和一个视图绑定起来，构建了一个视图持有者，等等。这个类有着多个责任。

这些责任应该被拆分开来吗？

这最终取决于你的应用是如何改变的。如果你的应用一变，你的视图组成也会发生改变，而且它的连接函数（视图的逻辑）也会发生改变。那么按照 Uncle Bob 说的，这个设计看起来比较死板，因为一个改变会导致另外一个改变。视图结构的改变也要求适配器本身发

生变化，这就是设计上的死板。然而，也有人说如果你的应用不需要这种会使得多个功能在不同时刻都发生变化的改变，那么就没有必要拆分它们。在这种情况下，拆分会带来不必要的复杂性。

所以，我们该如何做？

## 一个解释僵化的例子

让我们假设一个新产品需求，当 `order` 的总数是零的时候要有声明，视图需要在屏幕上显示一个黄色的“免费”图像而不是数字的总数。这部分逻辑应该加到哪里？一个方法是，你需要一个 `TextView`，另一个方法是，你需要一个 `ImageView`。有两个地方你需要修改：

- 视图里
- 表示层逻辑中

我看到的一些应用中这样处理，把这些逻辑加到适配层中。不幸的是，这要求 `Adapter` 在视图发生变化的时候也需要修改。如果这部分逻辑在适配器里面，那这就要求适配器的逻辑如果发生改变，视图的代码也要跟着变。这就给适配器增加了另外一个责任。

这也是 [数据-视图-主导器](#) 模式所强调的必要的解耦，这样做，类就不会变得特别死板，软件才有扩展、组合和测试的灵活性。例如，视图会实现一个接口，这个接口定义了它是如何和别人交互的，主导器会执行必要的逻辑。数据-视图-主导器模式中的主导器只负责视图、显示的逻辑，别的都不负责。

把这部分逻辑从适配器移到主导器会使得适配器更聚合，更符合单一职责原则。

这不是全部……

如果你深入看看任何 `RecyclerView` 适配器，你可能已经意识到适配器做了许多的事情。适配器还做了的事情是：

- 渲染视图
- 创建视图持有者
- 回收视图持有者
- 提供条目计数
- 其他。

由于 SRP 是单一职责，你可能想知道这些行为中的一部分到底该不该坚持运用 SRP 原则给提取出来。重申一遍，我将引用 Uncle Bob Martin 的话，因为他的解释实在是太准确了：

**变化的指针只在变化真正发生时起作用。如果没有任何征兆，应用 SRP 或者其它原则都是不明智的。**

适配器仍然包含着各种功能，事实上，就是这样设计的。毕竟，RecyclerView 适配器是一个简单的对于 [适配器模式](#) 的实现。在这个例子里，保持视图渲染和视图持有者的机制有意义；这是这个类的职责，而且是最好的实现。然而，引入额外的行为（比如视图逻辑）破坏了 SRP 原则，通过使用数据-视图-主导器模式或其他重构方式可以避免这个问题。

## 结论

单一职责原则可能是 SOLID 原则中最简单的原则了，因为它不言自明（再说一次） - **一个类应该有且只有一个发生改变的原因。**

当然，它也是个非常难以运用的原则。坚持认为需要使用 SRP 会很容易导致过度分析代码，这样你只会发现，如果你使用了 SRP，只是增加了应用的复杂性。我的建议是试着退一步，然后客观地审视代码。移除对代码的情感依赖，你将会发现你有一双清澈的眼睛。如果你这样做，你可能会从代码中发现你之前从来不曾了解的东西。你会意识到你需要使用单一职责原则，或者你可能意识到你已经做得很好了。无论如何，花点时间，多想想。

最后，如果你的应用改变了，你会发现你需要在你以前认为不需要采用 SRP 的地方使用 SRP。这完全没有问题，而且推荐这样做。

编码快乐。：）

看看这个系列的 [第二部分 英文版！](#)





# Android应用架构

来源:简书

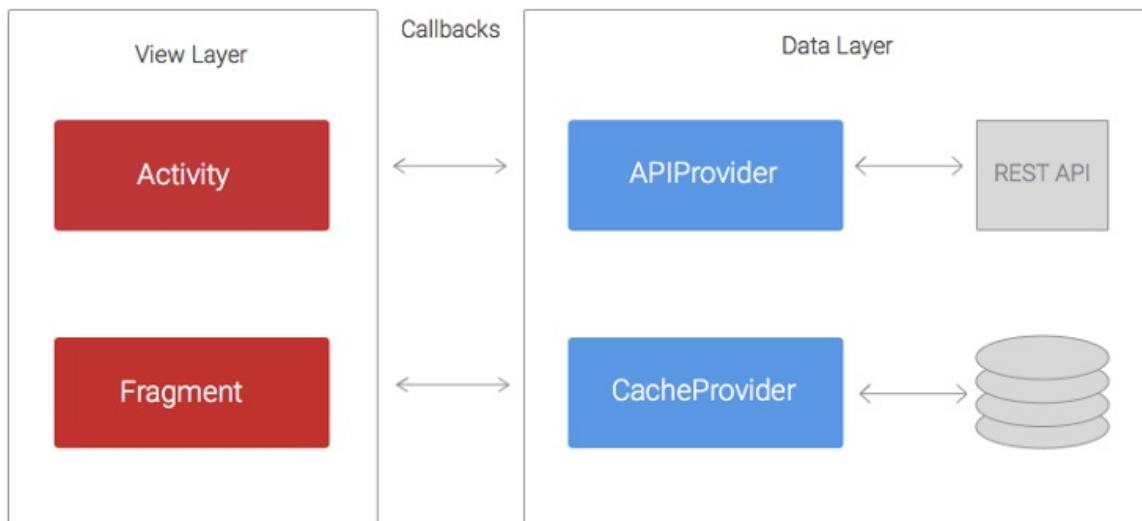
- 原文链接: [Android Application Architecture](#)
- 原文作者: Iván Carballo
- 译文出自: 小鄧子的简书
- 译者: 小鄧子
- 状态: 完成

Android开发生态圈的节奏非常之快。每周都会有新的工具诞生，类库的更新，博客的发表以及技术探讨。如果你外出度假一个月，当你回来的时候可能已经发布了新版本的 **Support Library** 或者 **Play Services**

我与 **Ribot Team** 一起做 Android 应用已经超过三年了。这段时间，我们所构建的 Android 应用架构和技术也在不断地演变。本文将向您阐述我们的经验，错误以及架构变化背后的原因。

## 曾经的架构

追溯到2012年我们的代码库使用的是基本结构，那个时候我们没有使用任何第三方网络类库，而且 `AsyncTask` 也是我们的好朋友。当时的架构可以大致表示为下图。



代码被划分为两层结构：**Data Layer**（数据层）负责从**REST API**或者持久数据存储区检索和存储数据；**View Layer**（视图层）的职责是处理并将数据展示在UI上。

**APIProvider**提供了一些方法，使 `Activity` 和 `Fragment` 能够很容易的实现与**REST API**的数据交互。这些方法使用 `URLConnection` 和 `AsyncTask` 在一个单独的线程内执行网络请求，然后通过回调将结果返回给 `Activity`。

按照同样的方式，**CacheProvider** 所包含的方法负责从 `SharedPreferences` 和 `SQLite` 数据库检索和存储数据。同样使用回调的方式，将结果传回 `Activity`。

## 存在的问题：

使用这种结构，最主要的问题在于**View Layer**持有太多的职责。想象一个简单且常见的场景，应用需要加载一个博客文章列表，然后缓存这些条目到 `SQLite` 数据库，最后将他们展示到 `ListView` 等列表视图上。`Activity` 要做到以下几个步骤：

1. 通过**APIProvider**调用 `loadPosts` 方法（回调）
2. 等待**APIProvider**的回调结果，然后调用**CacheProvider**中的 `savePosts` 方法（回调）
3. 等待**CacheProvider**的回调结果，然后将这些文章展示到 `ListView` 等列表视图上
4. 分别处理**APIProvider**和**CacheProvider**回调中潜在的异常。

这是一个非常简单的例子，在实际开发环境中**REST API**返回的数据可能并不是View直接需要的。因此，`Activity`在进行展示之前不得不通过某种方式将数据进行转换或过滤。另一个常见的情况是，调用 `loadPosts()` 所需要的参数，需要事先从其他地方获取到，比如，需要**Play Services SDK**提供一个Email地址参数。就像SDK通过异步回调的方式返回Email地址，这就意味着现在我们至少有三层嵌套的回调。如果继续添加复杂的业务逻辑，这种架构就会陷入众所周知的**Callback Hell**（回调地狱）。

## 总结：

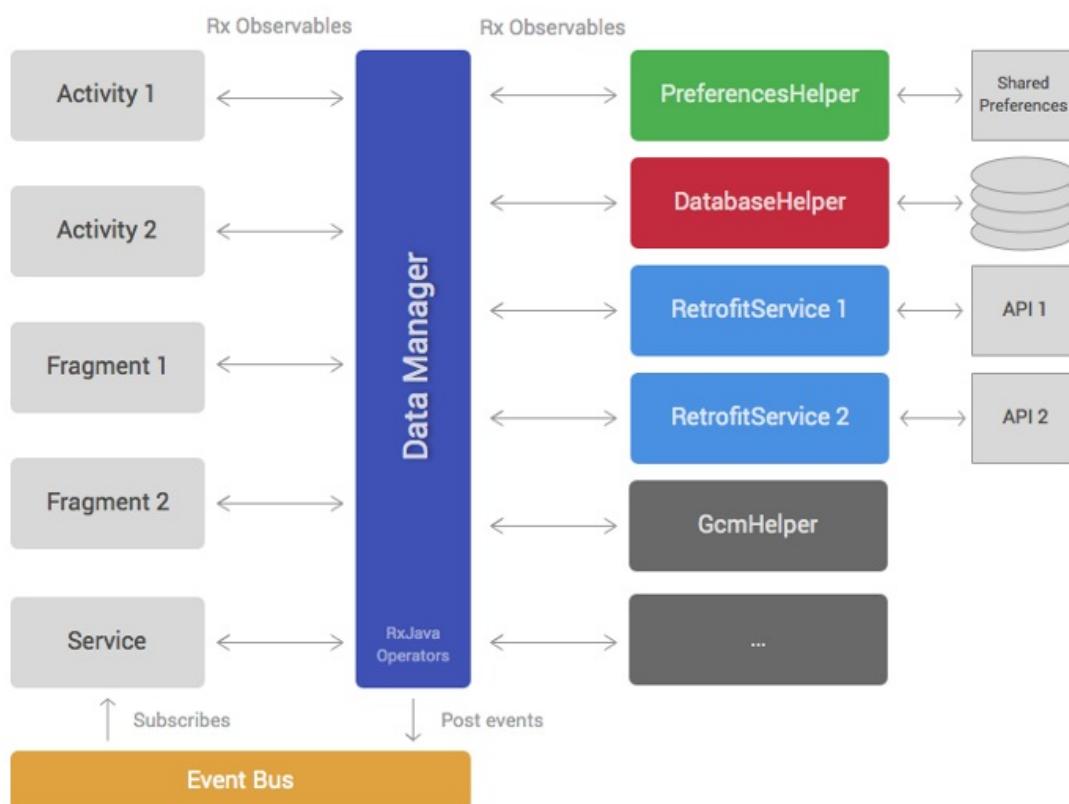
- `Activity`和`Fragment`变得非常庞大并且难以维护。
- 太多的回调嵌套意味着丑陋的代码结构而且不易读懂和理解。如果在这个基础上做更改或者添加新特性会感到很痛苦。
- 单元测试变得非常有挑战性，如果有可能的话，因为很多逻辑都留在了`Activity`或者`Fragment`中，这样进行单元测试是很艰难的。

# RxJava驱动的新型架构

我们使用上文提到的组织架构差不多两年的时间。在那段时间内，我们做了一些改进，稍微缓解了上述问题。例如，我们添加了一些**Helper Class**（帮助类）用来减少 `Activity` 和 `Fragment` 中的代码，在**APIProvider**中使用了**Volley**。尽管做出了这些改变，我们应用程序的代码还是不能进行友好的测试，并且**Callback Hell**（回调地狱）的问题还是经常发生。

直到2014年我们开始了解**RxJava**。在尝试了几个示例项目之后，我们意识到她可能最终帮助我们解决掉嵌套回调的问题。如果你还不熟悉响应式编程，可以阅读[本文](#)（译者注：译文点这里[那些年我们错过的响应式编程](#)）。简而言之，**RxJava**允许通过异步流的方式处理数据，并且提供了很多操作符，你可以将这些操作符作用于流上从而实现转换，过滤或者合并数据等操作。

考虑到经历了前几年的痛苦，我们开始考虑，一个新的应用程序体系架构看起来会是怎样的。因此，我们想出了这个。



类似于第一种架构，这种体系架构同样被划分为**Data Layer**和**View Layer**。**Data Layer**持有**DataManager**和一系列的**Helper classe**。**View Layer**由Android的**Framework**组件组成，例如，`Fragment`，`Activity`，`ViewGroup`等。

**Helper classes** (图标中的第三列) 有着非常特殊的职责以及简洁的实现方式。例如，很多项目需要一些帮助类对**REST API**进行访问，从数据库读取数据，或者与三方SDK进行交互等。不同的应用拥有不同数量的帮助类，但也存在着一些共性：

- **PreferencesHelper**: 从 `SharedPreferences` 读取和存储数据。
- **DatabaseHelper**: 处理操作SQLite数据库。
- **Retrofit services**: 执行访问**REST API**，我们现在使用Retrofit来代替Volley，因为它天生支持RxJava。而且也更好用。

帮助类里面的大多数public方法都会返回 RxJava 的 `Observable`。

**DataManager**是整个架构中的大脑。它广泛的使用了 RxJava 的操作符用来合并，过滤和转换从帮助类中返回的数据。**DataManager**旨在减少 `Activity` 和 `Fragment` 的工作量，它们（译者注：指 `Activity` 和 `Fragment`）要做的就是展示已经准备好的数据而不需要再进行转换了。

下面这段代码展示了一个**DataManager**方法可能的样子。这个简单的示例方法如下：

- 调用**Retrofit service**从**REST API**加载一个博客文章列表
- 使用**DatabaseHelper**保存文章到本地数据库，达到缓存的目的
- 筛选出今天发表的博客，因为那才是**View Layer**想要展示的。

```
public Observable<Post> loadTodayPosts() {
 return mRetrofitService.loadPosts()
 .concatMap(new Func1<List<Post>, Observable<Post>>() {
 @Override
 public Observable<Post> call(List<Post> apiPosts) {
 return mDatabaseHelper.savePosts(apiPosts);
 }
 })
 .filter(new Func1<Post, Boolean>() {
 @Override
 public Boolean call(Post post) {
 return isToday(post.date);
 }
 });
}
```

在**View Layer**中诸如 `Activity` 或者 `Fragment` 等组件只需调用这个方法，然后订阅返回的 `observable` 即可。一旦订阅完成，通过 `observable` 发送的不同博客，就能够立即被添加进 `Adapter` 从而展示到 `RecyclerView` 或其他类似控件上。

这个架构的最后元素就是**Event Bus**（事件总线）。它允许我们在**Data Layer**中发送事件，以便**View Layer**中的多个组件都能够订阅到这些事件。比如**DataManager**中的退出登录方法可以发送一个事件，订阅这个事件的多个 `Activity` 在接收到该事件后就能够更改它们的UI视图，从而显示一个登出状态。

为什么这种架构更好？

- RxJava的 `Observable` 和操作符避免了嵌套回调的出现。



- **DataManager**接管了以前**View Layer**的部分职责。因此，它使 `Activity` 和 `Fragment` 变得更轻量了。
- 将代码从 `Activity` 和 `Fragment` 转移到了**DataManager**和帮助类中，就意味着使写单元测试变得更简单。
- 明确的职责分离和**DataManager**作为唯一与**Data Layer**进行交互的点，使这个架构变得**Test-Friendly**。帮助类和**DataManager**能够很容易的被模拟出来。

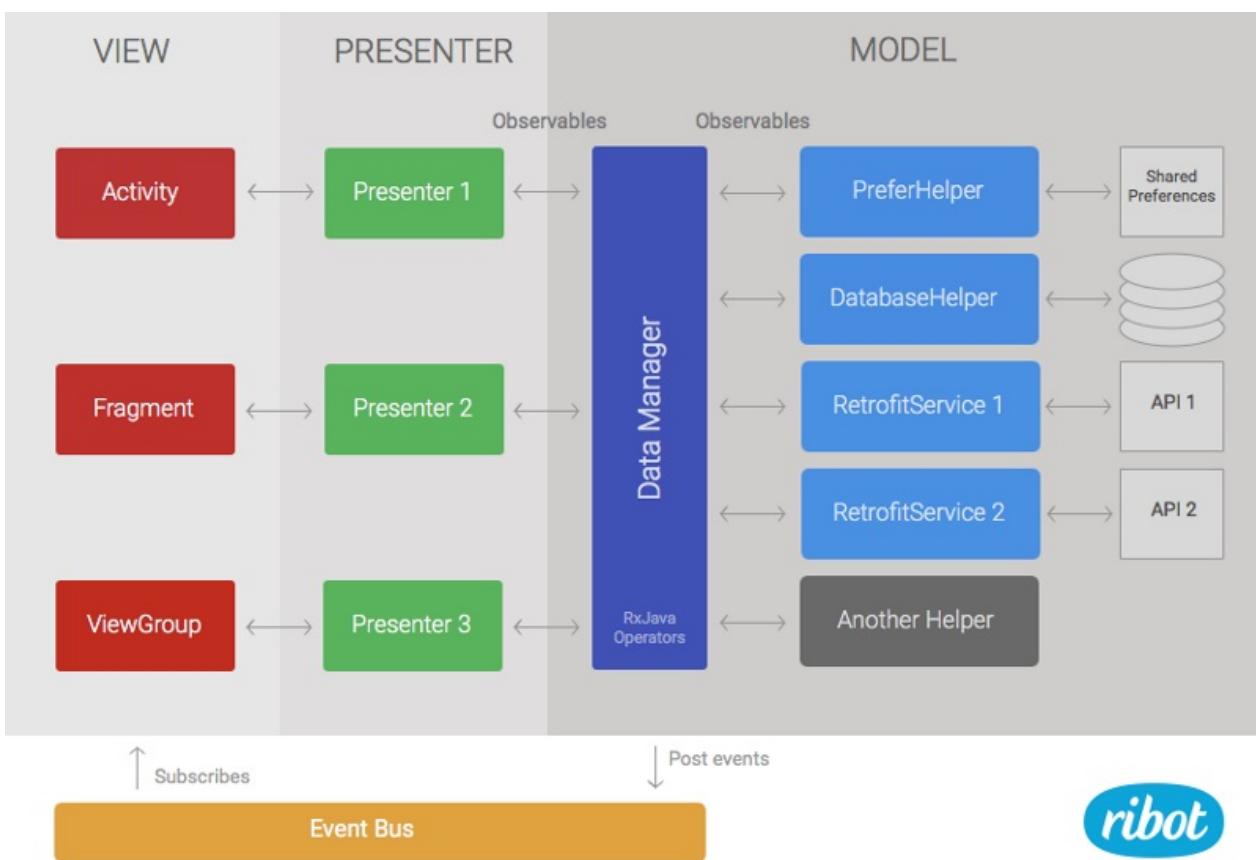
我们还存在什么问题？

- 对于庞大和复杂的项目来讲，**DataManager**会变得非常的臃肿和难以维护。

- 尽管**View Layer**诸如 Activity 和 Fragment 等组件变得更轻量，它们仍然要处理大量的逻辑，如管理 RxJava 的订阅，解析错误等方面。

## 集成MVP

在过去的一年中，几个架构设计模式，如**MVP**或者**MVVM**在Android社区内已经越来越受欢迎了。通过在[示例工程](#)和[文章](#)中进行探索后，我们发现MVP，可能给我们现有的架构带来非常价值的改进。因为当前我们的架构已经被划分为两个层（视图层和数据层），添加MVP会更自然些。我们只需要添加一个新的presenter层，然后将View中的部分代码转移到presenter就行了。



留下的**Data Layer**保持不变，只不过为了与这种模式保持一致性，它现在被叫做**Model**。

Presenter 负责从 Model 中加载数据，然后当数据准备好之后调用 view 中相对应的方法。还负责订阅DataManager返回的 observable。所以，他们还需要处理 schedulers 和 subscriptions。此外，它们还能分析错误代码或者在需要的情况下为数据流提供额外的操作。例如，如果我们需要过滤一些数据而且这个相同的过滤器是不可能被重用在其他地方的，这样的话在 Presenter 中实现比在DataManager中或许更有意义。

下面你将看到在 Presenter 中一个 public 方法将是什么样子。这段代码订阅我们在前一节中定义的 `dataManager.loadTodayPosts()` 所返回的 observable。

```

public void loadTodayPosts() {
 mMvpView.showProgressIndicator(true);
 mSubscription = mDataManager.loadTodayPosts().toList()
 .observeOn(AndroidSchedulers.mainThread())
 .subscribeOn(Schedulers.io())
 .subscribe(new Subscriber<List<Post>>() {
 @Override
 public void onCompleted() {
 mMvpView.showProgressIndicator(false);
 }

 @Override
 public void onError(Throwable e) {
 mMvpView.showProgressIndicator(false);
 mMvpView.showError();
 }

 @Override
 public void onNext(List<Post> postsList) {
 mMvpView.showPosts(postsList);
 }
 });
}

```

`mMvpView` 是与 `Presenter` 一起协助的 `View` 组件。通常情况下是一个 `Activity` , `Fragment` 或者 `ViewGroup` 的实例。

像之前的架构, **View Layer** 持有标准的 `Framework` 组件, 如 `ViewGroup` , `Fragment` 或者 `Activity` 。最主要的不同在于这些组件不再直接订阅 `Observable` 。取而代之的是通过实现 `MvpView` 接口, 然后提供一些更简洁的方法函数, 比如 `showError()` 或者 `showProgressIndicator()` 。这个 `view` 组件也负责处理用户交互, 如点击事件和调用相应 `Presenter` 中的正确方法。例如, 我有一个按钮用来加载博客列表, `Activity` 将会在点击事件的监听中调用 `presenter.loadTodayPosts()`

如果你想看到一个完整的运用MVP基本架构的工作示例, 可以从 [Github](#) 检出我们的[Android Boilerplate project](#)。也可以从这里阅读关于它的更多信息[Ribot的架构指导](#)

## 为什么这种架构更好?

- `Activity` 和 `Fragment` 变得非常轻量。他们唯一的职责就是建立/更新UI和处理用户事件。因此, 他们变得更容易维护。
- 现在我们通过模拟**View Layer**可以很容易的编写出单元测试。之前这些代码是**View Layer**的一部分, 所以我们很难对它进行单元测试。整个架构变得测试友好。

- 如果**DataManager**变得臃肿，我们可以通过转移一些代码到 `Presenter` 来缓解这个问题。

## 我们依然存在哪些问题？

- 当代码库变得非常庞大和复杂时，单一的**DataManager**依然是一个问题。虽然我们还没有走到这一步，但这是一个真正值得注意的问题，我们已经意识到了这一点，它可能发生。

值得一提的是它并不是一个完美的架构。事实上，不要天真的认为这是一个独特且完美的方案，能够解决你所有的问题。

Android生态系统将保持快速发展的步伐，我们必须继续探索。不断地阅读和尝试，这样我们才能找到更好的方法来继续构建优秀的Android应用程序。



# Android MVVM到底是啥？看完就明白了

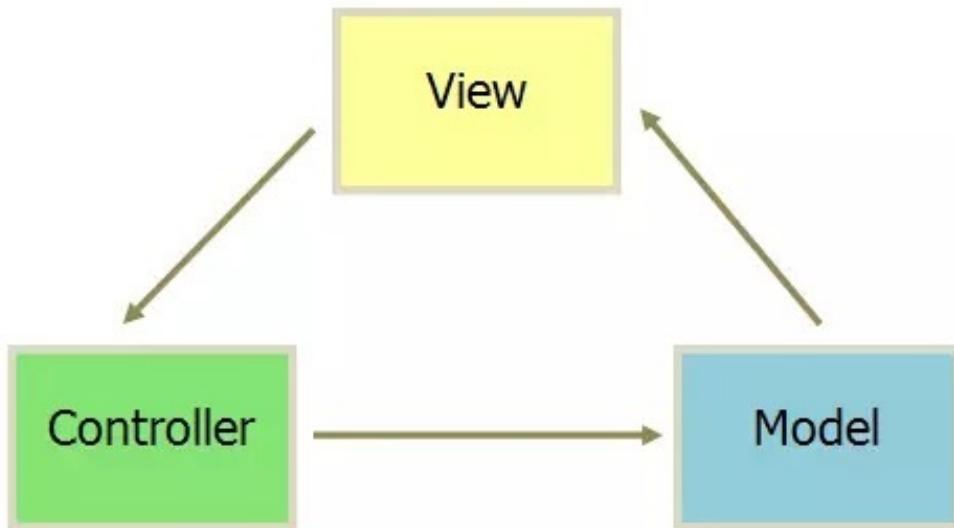
来源:[微信公众号](#)

今天投稿的胡笛同学是我共事多年的同事，他是我见过最优秀的程序员之一，他运营的技术公众号极客联盟干货很多，值得大家关注。今天他将为我们介绍Android MVVM框架Data Binding Library的使用。

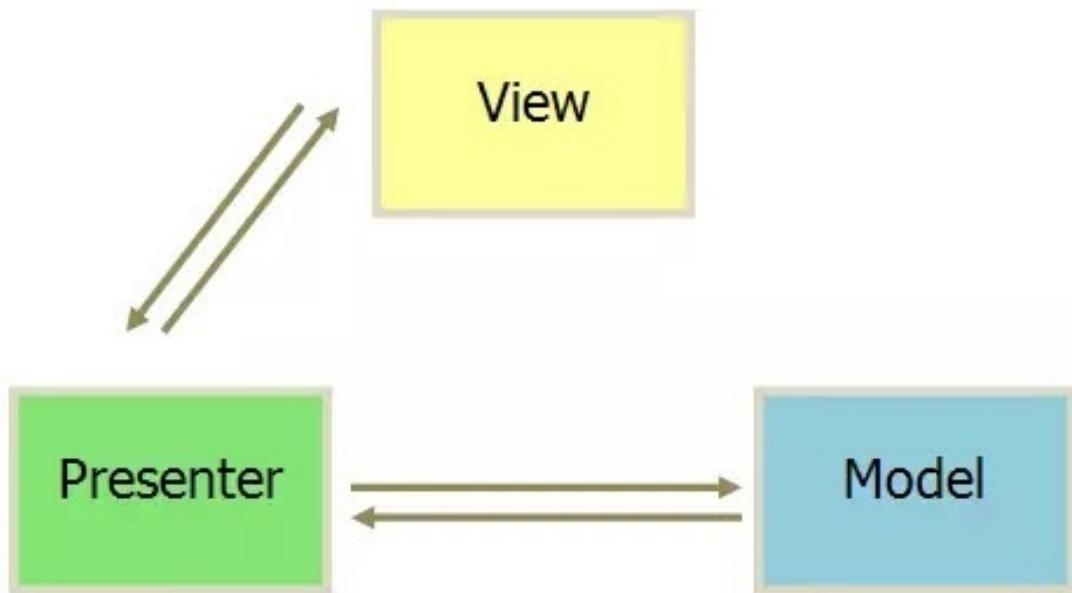
## 什么是MVVM

我们一步步来，从MVC开始。

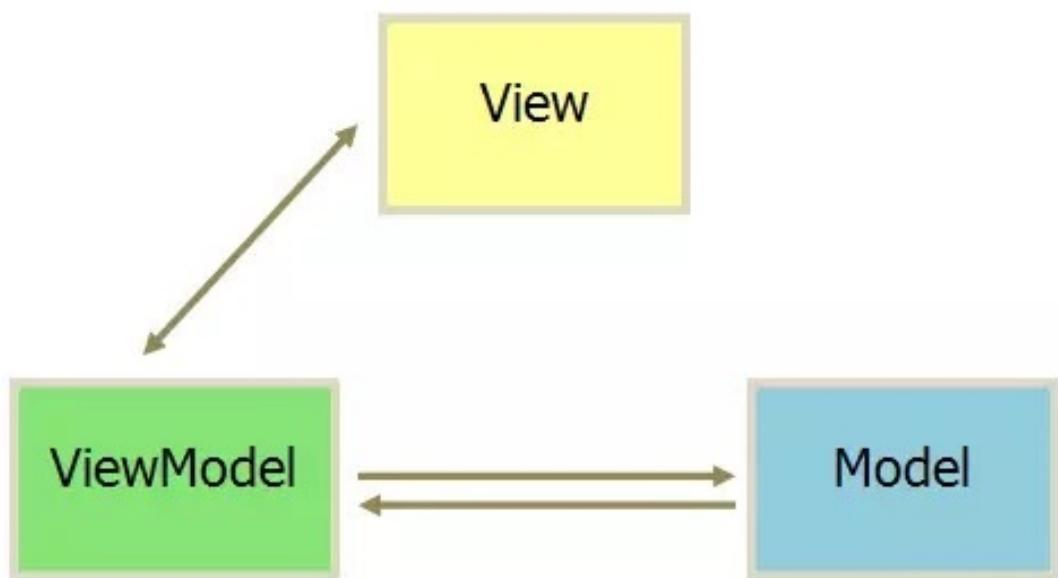
MVC 我们都知道，模型——视图——控制器。为了使得程序的各个部分分离降低耦合性，我们对代码的结构进行了划分。



他们的通信方式也如上图所示，即View层触发操作通知到业务层完成逻辑处理，业务层完成业务逻辑之后通知Model层更新数据，数据更新完之后通知View层展现。在实际运用中人们发现View和Model之间的依赖还是太强，希望他们可以绝对独立的存在，慢慢的就演化出了MVP。



Presenter 替换掉了Controller，不仅仅处理逻辑部分。而且还控制着View的刷新，监听 Model层的数据变化。这样隔离掉View和Model的关系后使得View层变的非常的薄，没有任何的逻辑部分又不用主动监听数据，被称之为“被动视图”。



至于MVVM基本上和MVP一模一样，感觉只是名字替换了一下。他的关键技术就是今天的主题(Data Binding)。View的变化可以自动的反应在ViewModel，ViewModel的数据变化也会自动反应到View上。这样开发者就不用处理接收事件和View更新的工作，框架已经帮你做好了。

## Data Binding Library

今年的Google IO 大会上，Android 团队发布了一个数据绑定框架（Data Binding Library）。以后可以直接在 layout 布局 xml 文件中绑定数据了，无需再 findViewById 然后手工设置数据了。其语法和使用方式和 JSP 中的 EL 表达式非常类似。

下面就来介绍怎么使用Data Binding Library。

### 配置环境

目前，最新版的Android Studio已经内置了该框架的支持，配置起来也很简单，只需要编辑app目录下的build.gradle文件，添加下面的内容就好了

```
android {
 ...
 dataBinding {
 enabled = true
 }
}
```

### Data Binding Layout文件

Data Binding layout文件有点不同的是：起始根标签是 layout，接下来一个 data 元素以及一个 view 的根元素。这个 view 元素就是你没有使用Data Binding的layout文件的根元素。举例说明如下：

```
<?xml version="1.0" encoding="utf-8"?><layout xmlns:android="http://schemas.android.com/apk/res/android">
<data>
 <variable name="user" type="com.example.User"/>
</data>
<LinearLayout
 android:orientation="vertical"
 android:layout_width="match_parent"
 android:layout_height="match_parent">
 <TextView android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="@{user.firstName}"/>
 <TextView android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="@{user.lastName}"/>
</LinearLayout></layout>
```

上面定义了一个 `com.example.User` 类型的变量 `user`，然后接着 `android:text="@{user.firstName}"` 把变量 `user` 的 `firstName` 属性的值和 `TextView` 的 `text` 属性绑定起来。

## Data Object

我们来看下上面用到的 `com.example.User` 对象。

```
public class User {
 public final String firstName;
 public final String lastName;

 public User(String firstName, String lastName) {
 this.firstName = firstName;
 this.lastName = lastName;
 }
}
```

他有两个 `public` 的属性 `firstName`，`lastName`，这和上面 `layout` 文件里面的 `@{user.firstName}` 和 `@{user.lastName}` 对应，或者下面这种形式的对象也是支持的。

```

public class User {
 private final String firstName;
 private final String lastName;

 public User(String firstName, String lastName) {
 this.firstName = firstName;
 this.lastName = lastName;
 }

 // getXXX形式
 public String getFirstName() {
 return this.firstName;
 }

 // 或者属性名和方法名相同
 public String lastName() {
 return this.lastName;
 }
}

```

## 绑定数据

添加完 `<data>` 标签后，Android Studio 就会根据 xml 的文件名自动生成一个继承 `viewDataBinding` 的类。例如：`activity_main.xml` 就会生成 `ActivityMainBinding`，然后我们在 `Activity` 里面添加如下代码：

```

@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 MainActivitiyBinding binding = DataBindingUtil.setContentView(this, R.layout.main_
 User user = new User("Test", "User");
 binding.setUser(user);
}

```

## 绑定事件

就像你可以在 xml 文件里面使用属性 `android:onClick` 绑定 `Activity` 里面的一个方法一样，Data Binding Library 扩展了更多的事件可以用来绑定方法，比如 `view.OnLongClickListener` 有个方法 `onLongClick()`，你就可以使 `android:onLongClick` 属性来绑定一个方法，需要注意的是绑定的方法的签名必须和该属性原本对应的方法的签名完全一样，否则编译阶段会报错。

下面举例来说明具体怎么使用，先看用来绑定事件的类：

```
public class MyHandlers {
 public void onClickButton(View view) { ... }

 public void afterFirstNameChanged(Editable s) { ... }
}
```

然后就是layout文件：

```
<?xml version="1.0" encoding="utf-8"?><layout xmlns:android="http://schemas.android.com/apk/res/android">
 <data>
 <variable name="handlers" type="com.example.Handlers"/>
 <variable name="user" type="com.example.User"/>
 </data>
 <LinearLayout
 android:orientation="vertical"
 android:layout_width="match_parent"
 android:layout_height="match_parent">
 <EditText android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="@{user.firstName}"
 android:afterTextChanged="@{handlers.afterFirstNameChanged}"/>
 <Button android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:onClick="@{handlers.onClickButton}"/>
 </LinearLayout>
</layout>
```

## 表达式语言 (Expression Language)

你可以直接在layout文件里面使用常见的表达式：

- 数学表达式 + - \* %
- 字符串链接 +
- 逻辑操作符 && ||
- 二元操作符 & | ^
- 一元操作符 + - ! ~
- Shift >> >>> <<
- 比较 == > < >= <=
- instanceof
- Grouping ()
- Literals – character, String, numeric, null
- Cast

- 函数调用
- 值域引用 (Field access)
- 通过[]访问数组里面的对象
- 三元操作符 ?:

示例：

```
android:text="@{String.valueOf(index + 1)}"
android:visibility="@{age < 13 ? View.GONE : View.VISIBLE}"
android:transitionName='@{"image_" + id}'
```

更多语法可以参考官网文档：[http://developer.android.com/tools/data-binding/guide.html#expression\\_language](http://developer.android.com/tools/data-binding/guide.html#expression_language)

## 更新界面

有些时候，代码会修改我们绑定的对象的某些属性，那么怎么通知界面刷新呢？下面就给出两种方案。

### 方案一

让你的绑定数据类继承 `BaseObservable`，然后通过调用 `notifyPropertyChanged` 方法来通知界面属性改变，如下：

```

private static class User extends BaseObservable {
 private String firstName;
 private String lastName;

 @Bindable
 public String getFirstName() {
 return this.firstName;
 }

 @Bindable
 public String getLastName() {
 return this.lastName;
 }

 public void setFirstName(String firstName) {
 this.firstName = firstName;
 notifyPropertyChanged(BR.firstName);
 }

 public void setLastName(String lastName) {
 this.lastName = lastName;
 notifyPropertyChanged(BR.lastName);
 }
}

```

在需要通知的属性的get方法上加上 `@Bindable`，这样编译阶段会生成 `BR.[property name]`，然后使用这个调用方法 `notifyPropertyChanged` 就可以通知界面刷新了。如果你的数据绑定类不能继承 `BaseObservable`，那你就只能自己实现 `Observable` 接口，可以参考 `BaseObservable` 的实现。

## 方案二

Data Binding Library提供了很便利的类 `ObservableField`，还有 `ObservableBoolean`，`ObservableByte`，`ObservableChar`，`ObservableShort`，`ObservableInt`，`ObservableLong`，`ObservableFloat`，`ObservableDouble`，和 `ObservableParcelabel`，基本上涵盖了各种我们需要的类型。用法很简单，如下：

```

private static class User {
 public final ObservableField<String> firstName = new ObservableField<>();

 public final ObservableField<String> lastName = new ObservableField<>();

 public final ObservableInt age = new ObservableInt();
}

```

然后使用下面的代码来访问：

```
user.firstName.set("Google");
int age = user.age.get();
```

调用set方法时， Data Binding Library就会自动的帮我们通知界面刷新了。

## 绑定AdapterView

在一个实际的项目中，相信AdapterView是使用得很多的，使用官方提供给的API来进行 AdapterView的绑定需要写很多代码，使用起来不方便，但是由于Data Binding Library提供丰富的扩展功能，所以出现了很多第三方的库来扩展它，下面就来介绍一个比较好用的库[binding-collection-adapter](#),使用的时候在你的 build.gradle 文件里面添加 compile 'me.tatarka:bindingcollectionadapter:0.16' ,如果你要是用 RecyclerView ，还需要添加 compile 'me.tatarka:bindingcollectionadapter-recyclerview:0.16'

下面就是ViewModel的写法：

```
public class ViewModel {
 public final ObservableList<String> items = new ObservableArrayList<>();

 public final ItemView itemView = ItemView.of(BR.item, R.layout.item);
}
```

这里用到了 ObservableList ,他会在items变化的时候自动刷新界面,然后下面是layout.xml文件:

```

<layout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">

 <data>
 <variable name="viewModel" type="com.example.ViewModel"/>
 <import type="me.tatarka.bindingcollectionadapter.LayoutManagers" />
 </data>
 <ListView
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 app:items="@{viewModel.items}"
 app:itemView="@{viewModel.itemView}"/>
 <android.support.v7.widget.RecyclerView
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 app:layoutManager="@{LayoutManagers.linear()}"
 app:items="@{viewModel.items}"
 app:itemView="@{viewModel.itemView}"/>
 <android.support.v4.view.ViewPager
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 app:items="@{viewModel.items}"
 app:itemView="@{viewModel.itemView}"/>
 <Spinner
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 app:items="@{viewModel.items}"
 app:itemView="@{viewModel.itemView}"
 app:dropDownItemView="@{viewModel.dropDownItemView}"/>
</layout>

```

然后是item.xml：

```

<layout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">
 <data>
 <variable name="item" type="String"/>
 </data>
 <TextView
 android:id="@+id/text"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:text="@{item}"/>
</layout>

```

如果有多种样式的布局，那么就需要把ItemView换成 ItemViewSelector，如下：

```
public final ItemViewSelector<String> itemView = new BaseItemViewSelector<String>() {

 @Override
 public void select(ItemView itemView, int position, String item) {
 itemView.set(BR.item, position == 0 ? R.layout.item_header : R.layout.item);
 }

 // This is only needed if you are using a BindingListViewAdapter
 @Override
 public int viewTypeCount() {
 return 2;
 }
};
```

## 自定义绑定

正常情况下，Data Binding Library会根据属性名去找对应的set方法，但是我们有时候需要自定义一些属性，Data Binding Library也提供了很便利的方法让我们来实现。

比如我们想在layout文件里面设置ListView的emptyView，以前这个是无法做到的，只能在代码里面通过调用setEmptyView来做；

但是现在借助Data Binding Library，我们可以很容易的实现这个功能了。先看layout文件：

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">
 <data>
 <variable
 name="viewModel"
 type="com.example.databinding.viewmodel.ViewAlbumsViewModel"/>
 </data>
 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:paddingLeft="10dp"
 android:paddingRight="10dp"
 android:orientation="vertical">
 <ListView
 android:layout_width="fill_parent"
 android:layout_height="0px"
 android:layout_weight="1.0"
 app:items="@{viewModel.albums}"
 app:itemView="@{viewModel.itemView}"
 app:emptyView="@{@id/empty_view}"
 android:onItemClick="@{viewModel.viewAlbum}"
 android:id="@+id/albumListView"/>
 <TextView
 android:id="@+id/empty_view"
 android:layout_width="fill_parent"
 android:layout_height="0px"
 android:layout_weight="1.0"
 android:gravity="center"
 android:text="@string/albums_list_empty" />
 <Button
 android:layout_width="fill_parent"
 android:layout_height="wrap_content"
 android:text="@string/create"
 android:onClick="@{viewModel.createAlbum}"/>
 </LinearLayout>
</layout>
```

`app:emptyView="@{@id/empty_view}"` 这个代码就用来指定 `emptyView`，下面来看下实现的代码：

```
@BindingAdapter("emptyView")
public static <T> void setEmptyView(AdapterView adapterView, int viewId) {

 View rootView = adapterView.getRootView();
 View emptyView = rootView.findViewById(viewId);
 if (emptyView != null) {
 adapterView.setEmptyView(emptyView);
 }
}
```

下面我们来分析上面的代码，`@{@id/empty_view}` 表示引用了 `@id/empty_view` 这个 id，所以它的值就是 int，再看上面的 `setEmptyView` 方法，第一个参数 `AdapterView adapterView` 表示使用 `emptyView` 这个属性的控件，而第二个参数 `int viewId` 则是 `emptyView` 属性传进来的值，上面的 `layout` 可以看出来它就是 `R.id.empty_view`，然后通过 id 找到控件，然后调用原始的 `setEmptyView` 来设置。

上面的代码来自我写的一个 Data Binding Library 的示例项目 [DataBinding-album-sample](#) 它基本上包含了开发一个 app 常用到的东西，大家有兴趣可以通过阅读原文去看看。

扫描或长按下方二维码可以关注胡笛同学的微信技术公众号



“崇尚自由，推崇技术，拥抱开源” - 极客联盟：传播新技术理念，分享技术经验。打造华中区最有影响力的技术公众号。



# 使用

# 选择恐惧症的福音！教你认清MVC， MVP和MVVM

来源:[zjutkz.net](http://zjutkz.net)

相信大家对MVC， MVP和MVVM都不陌生，作为三个最耳熟能详的Android框架，它们的应用可以是非常广泛的，但是对于一些新手来说，可能对于区分它们三个都有困难，更别说在实际的项目中应用了，有些时候想用MVP的，代码写着写着就变成了MVC，久而久之就对它们三个的选择产生了恐惧感，如果你也是这样的人群，那么这篇文章可能会对你有很大的帮助，希望大家看完都会有收获吧！

文章重点：

- (1)了解并区分MVC， MVP， MVVM。
- (2)知道这三种模式在Android中如何使用。
- (3)走出data binding的误区。
- (4)了解MVP+data binding的开发模式。

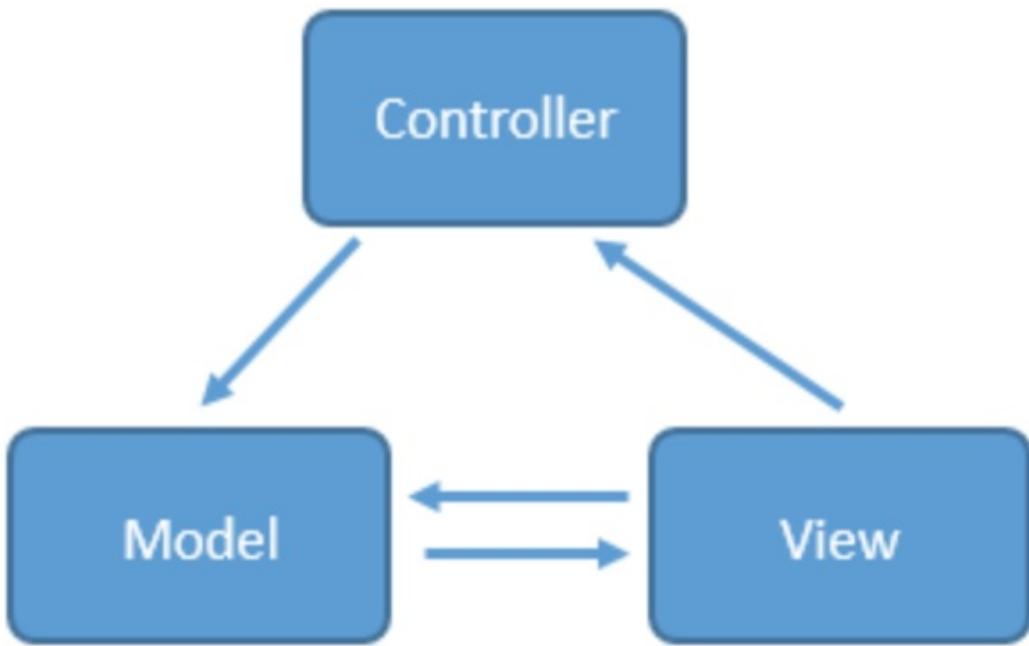
本篇文章的demo我将会上传到[我的github](#)上。

## 水之积也不厚，则其负大舟也无力

正如庄子在逍遥游中说的，如果水不够深，那就没有能够担负大船的力量。所以在真正开始涉及具体的代码之前，我们要先对MVC， MVP和MVVM做一个初步的了解。如果各位同学对此已经有所了解了，可以选择性跳过这一节。

## MVC

MVC， Model View Controller，是软件架构中最常见的一种框架，简单来说就是通过controller的控制去操作model层的数据，并且返回给view层展示，具体见下图



当用户出发事件的时候，view层会发送指令到controller层，接着controller去通知model层更新数据，model层更新完数据以后直接显示在view层上，这就是MVC的工作原理。

那具体到Android上是怎么样一个情况呢？

大家都知道一个Android工程有什么对吧，有java的class文件，有res文件夹，里面是各种资源，还有类似manifest文件等等。对于原生的Android项目来说，layout.xml里面的xml文件就对应于MVC的view层，里面都是一些view的布局代码，而各种java bean，还有一些类似repository类就对应于model层，至于controller层嘛，当然就是各种activity咯。大家可以试着套用我上面说的MVC的工作原理是理解。比如你的界面有一个按钮，按下这个按钮去网络上下载一个文件，这个按钮是view层的，是使用xml来写的，而那些和网络连接相关的代码写在其他类里，比如你可以写一个专门的networkHelper类，这个就是model层，那怎么连接这两层呢？是通过button.setOnClickListener()这个函数，这个函数就写在了activity中，对应于controller层。是不是很清晰。

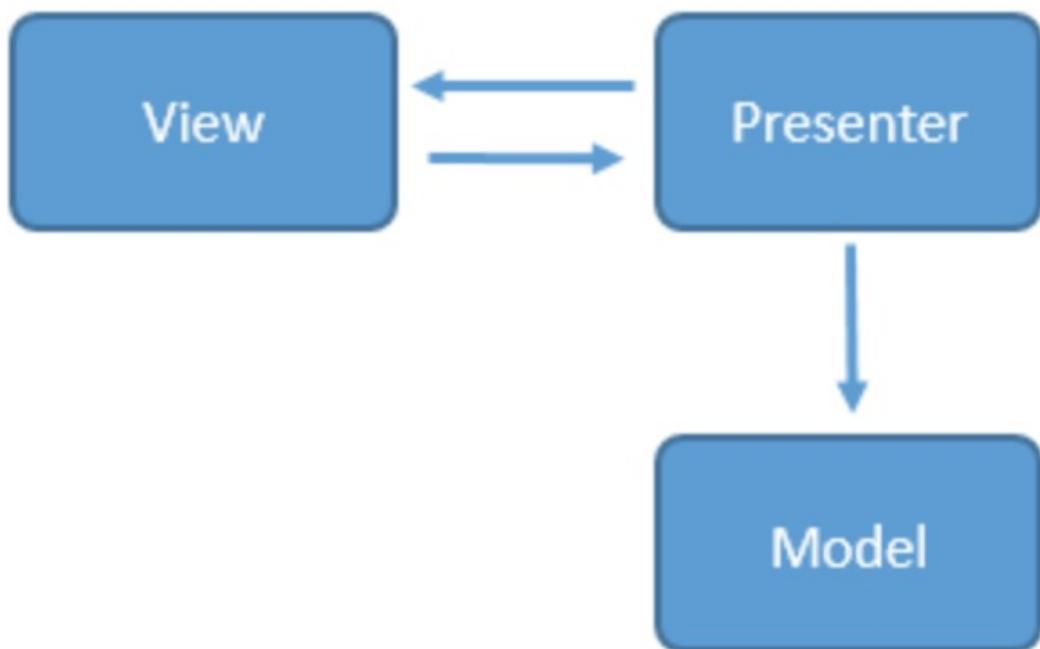
大家想过这样会有什么问题吗？显然是有的，不然为什么会有MVP和MVVM的诞生呢，是吧。问题就在于xml作为view层，控制能力实在太弱了，你想去动态的改变一个页面的背景，或者动态的隐藏/显示一个按钮，这些都没办法在xml中做，只能把代码写在activity中，造成了activity既是controller层，又是view层的这样一个窘境。大家回想一下自己写的代码，如果是一个逻辑很复杂的页面，activity或者fragment是不是动辄上千行呢？这样不仅写起来麻烦，维护起来更是噩梦。（当然看过Android源码的同学其实会发现上千行的代码不算啥，一个RecyclerView.class的代码都快上万行了。。）

MVC还有一个重要的缺陷，大家看上面那幅图，view层和model层是相互可知的，这意味着两层之间存在耦合，耦合对于一个大型程序来说是非常致命的，因为这表示开发，测试，维护都需要花大量的精力。

正因为MVC有这样那样的缺点，所以才演化出了MVP和MVVM这两种框架。

## MVP

MVP作为MVC的演化，解决了MVC不少的缺点，对于Android来说，MVP的model层相对于MVC是一样的，而activity和fragment不再是controller层，而是纯粹的view层，所有关于用户事件的转发全部交由presenter层处理。下面还是让我们看图

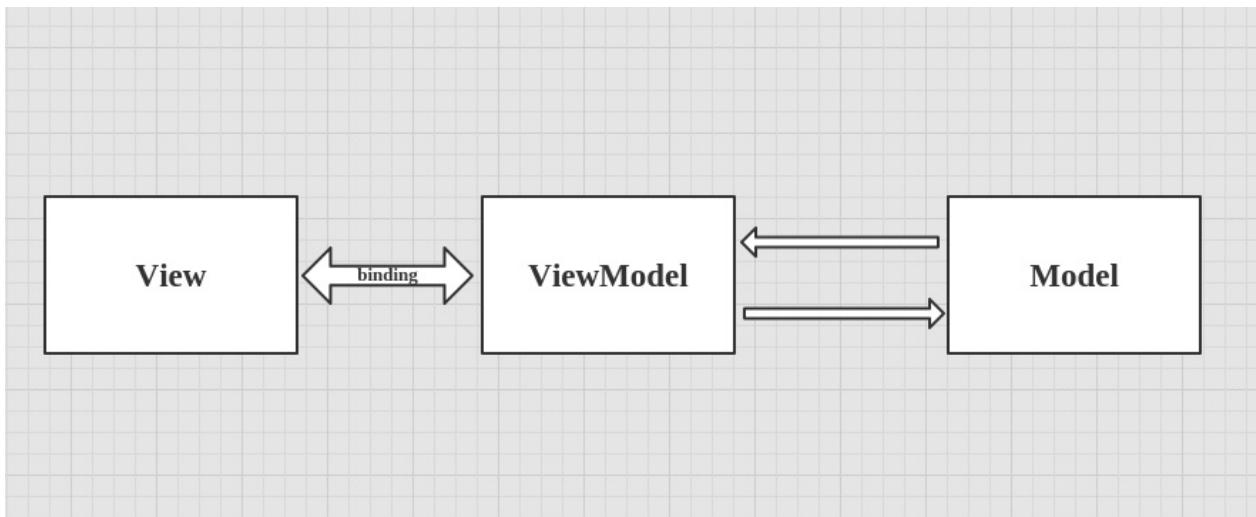


从图中就可以看出，最明显的差别就是view层和model层不再相互可知，完全的解耦，取而代之的presenter层充当了桥梁的作用，用于操作view层发出的事件传递到presenter层中，presenter层去操作model层，并且将数据返回给view层，整个过程中view层和model层完全没有联系。看到这里大家可能会问，虽然view层和model层解耦了，但是view层和presenter层不是耦合在一起了吗？其实不是的，对于view层和presenter层的通信，我们是可以通过接口实现的，具体的意思就是说我们的activity，fragment可以去实现实现定义好的接口，而在对应的presenter中通过接口调用方法。不仅如此，我们还可以编写测试用的View，模拟用户的各种操作，从而实现对Presenter的测试。这就解决了MVC模式中测试，维护难的问题。

当然，其实最好的方式是使用fragment作为view层，而activity则是用于创建view层(fragment)和presenter层(presenter)的一个控制器。

## MVVM

MVVM最早是由微软提出的



这里要感谢[泡在网上的日子](#)，因为前面看到的三张图我都是从它的博客中摘取的，如果有人知道不允许这样做的话请告诉我，我会从我的博客中删除的，谢谢。

从图中看出，它和MVP的区别貌似不大，只不过是presenter层换成了viewmodel层，还有一点就是view层和viewmodel层是相互绑定的关系，这意味着当你更新viewmodel层的数据的时候，view层会相应的变动ui。

我们很难去说MVP和MVVM这两个MVC的变种孰优孰劣，还是要具体情况具体分析。

## 纸上得来终觉浅，绝知此事要躬行

对于程序员来说，空谈是最没效率的一种方式，相信大家看了我上面对于三种模式的分析，或多或少都会有点云里雾里，下面让我们结合代码来看看。

让我们试想一下下面这个情景，用户点击一个按钮A，获取github上对公司对应仓库中贡献排行第一的任的名字，然后我们还会有一个按钮B，用户点击按钮B，界面上排行第一的那个人的名字就会换成自己的。

## MVC

MVC实现是最简单的。

## 首先看对应view层的xml文件

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"

 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:id="@+id/container"
 android:orientation="vertical"
 tools:context=".ui.view.MainActivity"
 android:fitsSystemWindows="true">

 <Button
 android:text="get"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:onClick="get"/>

 <Button
 android:text="change"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:onClick="change"/>

 <TextView
 android:id="@+id/top_contributor"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:gravity="center"
 android:textSize="30sp"/>

```

很简单，两个Button一个TextView

接着看对应controller层的activity

```
public class MainActivity extends AppCompatActivity {

 private ProgressDialog dialog;

 private Contributor contributor = new Contributor();

 private TextView topContributor;

 private Subscriber<Contributor> contributorSub = new Subscriber<Contributor>() {

 @Override
 public void onStart() {
 showProgress();
 }

 @Override
 public void onCompleted() {

 }

 @Override
 public void onError(Throwable e) {

 }

 @Override
 public void onNext(Contributor contributor) {
 MainActivity.this.contributor = contributor;
 topContributor.setText(contributor.login);
 dismissProgress();
 }
 };

 @Override
 protected void onCreate(Bundle savedInstanceState) {

 super.onCreate(savedInstanceState);

 setContentView(R.layout.activity_main);

 topContributor = (TextView) findViewById(R.id.top_contributor);
 }

 public void get(View view) {
```

```
 getTopContributor("square", "retrofit");
 }

 public void change(View view) {
 contributor.login = "zjutkz";
 topContributor.setText(contributor.login);
 }

 public void getTopContributor(String owner, String repo) {

 GitHubApi.getContributors(owner, repo)
 .take(1)
 .observeOn(AndroidSchedulers.mainThread())
 .subscribeOn(Schedulers.newThread())
 .map(new Func1<List<Contributor>, Contributor>() {

 @Override

 public Contributor call(List<Contributor> contributors) {
 return contributors.get(0);
 }
 })
 .subscribe(contributorSub);
 }

 public void showProgress() {
 if (dialog == null) {
 dialog = new ProcessDialog(this);
 }
 dialog.showMessage("正在加载...");
 }

 public void dismissProgress() {
 if (dialog == null) {
 dialog = new ProcessDialog(this);
 }
 dialog.dismiss();
 }

}
```

我们看一下get()方法中调用的getTopContributor方法

```
public void getTopContributor(String owner, String repo){
 GitHubApi.getContributors(owner, repo)
 .take(1)
 .observeOn(AndroidSchedulers.mainThread())
 .subscribeOn(Schedulers.newThread())
 .map(new Func1<List<Contributor>, Contributor>() {

 @Override
 public Contributor call(List<Contributor> contributors) {
 return contributors.get(0);
 }
 })
 .subscribe(contributorSub);
}
```

熟悉rxjava和retrofit的同学应该都明白这是啥意思，如果对这两个开源库不熟悉也没事，可以参考[给 Android 开发者的 RxJava 详解](#)和[用 Retrofit 2 简化 HTTP 请求](#)这两篇文章。

对于这里大家只要知道这段代码的意思就是去获取github上owner公司中的repo仓库里贡献排名第一的那个人。贡献者是通过Contributor这个java bean存储的。

```
public class Contributor {

 public String login;
 public int contributions;

 @Override
 public String toString() {
 return login + ", " + contributions;
 }
}
```

很简单，login表示贡献者的名字，contributor表示贡献的次数。

然后通过rxjava的subscriber中的onNext()函数得到这个数据。

```
private Subscriber<Contributor> contributorSub = new Subscriber<Contributor>() {

 @Override
 public void onStart() {
 showProgress();
 }

 @Override
 public void onCompleted() {

 }

 @Override
 public void onError(Throwable e) {

 }

 @Override
 public void onNext(Contributor contributor) {
 MainActivity.this.contributor = contributor;

 topContributor.setText(contributor.login);

 dismissProgress();
 }
};
```

至于另外那个change按钮的工作大家应该都看得懂，这里不重复了。

好了，我们来回顾一遍整个流程。

首先在xml中写好布局代码。

其次，activity作为一个controller，里面的逻辑是监听用户点击按钮并作出相应的操作。比如针对get按钮，做的工作就是调用GithubApi的方法去获取数据。

GithubApi、Contributor等类则表示MVC中的model层，里面是数据和一些具体的逻辑操作。

说完了流程再来看看问题，还记得我们前面说的吗，MVC在Android上的应用，一个具体的问题就是activity的责任过重，既是controller又是view。这里是怎体现的呢？看了代码大家发现其中有一个progressDialog，在加载数据的时候显示，加载完了以后取消，逻辑其实是view层的逻辑，但是这个我们没办法写到xml里面啊，包括TextView.setText()，这个也一样。我们只能把这些逻辑写到activity中，这就造成了activity的臃肿，这个例子可能还好，如果是一个复杂的页面呢？大家自己想象一下。

# MVP

通过具体的代码大家知道了MVC在Android上是如何工作的，也知道了它的缺点，那MVP是如何修正的呢？

这里先向大家推荐github上的一个第三方库，通过这个库大家可以很轻松的实现MVP。好了，还是看代码吧。

首先还是xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:id="@+id/container"
 android:orientation="vertical"
 tools:context=".ui.view.MainActivity"
 android:fitsSystemWindows="true">

 <Button
 android:text="get"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:onClick="get"/>

 <Button
 android:text="change"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:onClick="change"/>

 <TextView
 android:id="@+id/top_contributor"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:gravity="center"
 android:textSize="30sp"/>

</LinearLayout>
```

这个和MVC是一样的，毕竟界面的形式是一样的嘛。

接下去，我们看一个接口。

```
public interface ContributorView extends MvpView {

 void onLoadContributorStart();

 void onLoadContributorComplete(Contributor topContributor);

 void onChangeContributorName(String name);
}
```

这个接口起什么作用呢？还记得我之前说的吗？MVP模式中，view层和presenter层靠的就是接口进行连接，而具体的就是上面的这个了，里面定义的三个方法，第一个是开始获取数据，第二个是获取数据成功，第三个是改名。我们的view层（activity）只要实现这个接口就可以了。

下面看activity的代码

```
public class MainActivity extends MvpActivity<ContributorView, ContributorPresenter> implements
 ContributorView {

 private ProgressDialog dialog;

 private TextView topContributor;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 topContributor = (TextView) findViewById(R.id.top_contributor);
 }

 @NotNull
 @Override
 public ContributorPresenter createPresenter() {
 return new ContributorPresenter();
 }

 public void get(View view){
 getPresenter().get("square", "retrofit");
 }

 public void change(View view){
 getPresenter().change();
 }

 @Override
 public void onLoadContributorStart() {
 showProgress();
 }
}
```

```

@Override
public void onLoadContributorComplete(Contributor contributor) {
 topContributor.setText(contributor.toString());
 dismissProgress();
}

@Override
public void onChangeContributorName(String name) {
 topContributor.setText(name);
}

public void showProgress(){
 if(dialog == null){
 dialog = new ProcessDialog(this);
 }

 dialog.showMessage("正在加载...");
}

public void dismissProgress(){
 if(dialog == null){
 dialog = new ProcessDialog(this);
 }

 dialog.dismiss();
}
}

```

它继承自MvpActivity，实现了刚才的ContributorView接口。继承的那个MvpActivity大家这里不用太关心主要是做了一些初始化和生命周期的封装。我们只要关心这个activity作为view层，到底是怎么工作的。

```

public void get(View view){
 getPresenter().get("square", "retrofit");
}

public void change(View view){
 getPresenter().change();
}

```

get()和change()这两个方法是我们点击按钮以后执行的，可以看到，里面完完全全没有任何和model层逻辑相关的东西，只是简单的委托给了presenter，那我们再看看presenter层做了什么

```
public class ContributorPresenter extends MvpBasePresenter<ContributorView> {

 private Subscriber<Contributor> contributorSub = new Subscriber<Contributor>() {

 @Override
 public void onStart() {
 ContributorView view = getView();
 if(view != null){
 view.onLoadContributorStart();
 }
 }

 @Override
 public void onCompleted() {

 }

 @Override
 public void onError(Throwable e) {

 }

 @Override
 public void onNext(Contributor topContributor) {
 ContributorView view = getView();
 if(view != null){
 view.onLoadContributorComplete(topContributor);
 }
 }
 };

 public void get(String owner, String repo){
 GitHubApi.getContributors(owner, repo)
 .take(1)
 .observeOn(AndroidSchedulers.mainThread())
 .subscribeOn(Schedulers.newThread())
 .map(new Func1<List<Contributor>, Contributor>() {

 @Override
 public Contributor call(List<Contributor> contributors) {
 return contributors.get(0);
 }
 })
 .subscribe(contributorSub);
 }

 public void change(){
 ContributorView view = getView();
 if(view != null){
 view.onChangeContributorName("zjutkz");
 }
 }
}
```

```
 }
}
```

其实就是在刚才MVC中activity的那部分和model层相关的逻辑抽取了出来，并且在相应的时机调用ContributorView接口对应的方法，而我们的activity是实现了这个接口的，自然会走到对应的方法中。

好了，我们来捋一捋。

首先，和MVC最大的不同，MVP把activity作为view层，通过代码也可以看到，整个activity没有任何和model层相关的逻辑代码，取而代之的是把代码放到了presenter层中，presenter获取了model层的数据之后，通过接口的形式将view层需要的数据返回给它就OK了。

这样的好处是什么呢？首先，activity的代码逻辑减少了，其次，view层和model层完全解耦，具体来说，如果你需要测试一个http请求是否顺利，你不需要写一个activity，只需要写一个java类，实现对应的接口，presenter获取了数据自然会调用相应的方法，相应的，你也可以自己在presenter中mock数据，分发给view层，用来测试布局是否正确。

## MVVM

首先在看这段内容之前，你需要保证你对data binding框架有基础的了解。不了解的同学可以去看下[这篇文章](#)。在接下去让我们开始探索MVVM，MVVM最近在Android上可谓十分之火，最主要的原因就是谷歌推出了data binding这个框架，可以轻松的实现MVVM。但是，我在网上查阅关于Android的data binding资料的时候，发现国内有很多人都误解了，首先，我们从一篇[错误的文章](#)开始。当然我在这里引用这篇文章也是对事不对人，如果对文章的作者产生了不好的影响我这里说一声抱歉。

上面那篇文章是一个关于data binding的使用，看起来很美好，但是，其中有一个错误可以说是非常，非常，非常严重的。

定义好了Model和View之后，需要把两者连起来，当Model的数据变化后，自动更新View。

Android Data Binding中的ViewModel是根据layout自动生成的Binding类，如果layout的名称是movie\_item.xml，生成的Binding类名称就是MovieItemBinding。

它竟然说data binding的viewmodel层是binding类，其实不止是这篇文章，其他有一些开发者写的关于data binding的文章里都犯了一样的错误。大家如果也有这样的概念，请务必纠正过来！！

说完了错误的概念，那data binding中真正的viewmodel是什么呢？我们还是以之前MVC， MVP的那个例子做引导。

首先是view层，这没啥好说的，和MVP一样，只不过多了数据绑定。view层就是xml和activity。

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
 <data>
 <variable name="contributor" type="zjutkz.com.mvvm.viewmodel.Contributor"/>
 </data>
 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:id="@+id/container"
 android:orientation="vertical"
 android:fitsSystemWindows="true">

 <Button
 android:id="@+id/get"
 android:text="get"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:onClick="get"/>

 <Button
 android:id="@+id/change"
 android:text="change"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:onClick="change"/>

 <TextView
 android:id="@+id/top_contributor"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:gravity="center"
 android:textSize="30sp"
 android:text="@{contributor.login}"/>
 </LinearLayout>

</layout>
```

```
public class MainActivity extends AppCompatActivity {

 private Subscriber<Contributor> contributorSub = new Subscriber<Contributor>() {

 @Override
```

```
public void onStart() {
 showProgress();
}

@Override
public void onCompleted() {

}

@Override
public void onError(Throwable e) {

}

@Override
public void onNext(Contributor contributor) {
 binding.setContributor(contributor);

 dismissProgress();
}
};

private ProcessDialog dialog;

private MvvmActivityMainBinding binding;

@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 binding = DataBindingUtil.setContentView(this, R.layout.mvvm_activity_main);
}

public void get(View view){
 getContributors("square", "retrofit");
}

public void change(View view){
 if(binding.getContributor() != null){
 binding.getContributor().setLogin("zjutkz");
 }
}

public void showProgress(){
 if(dialog == null){
 dialog = new ProcessDialog(this);
 }

 dialog.showMessage("正在加载...");
}

public void dismissProgress(){
 if(dialog == null){


```

```
 dialog = new ProcessDialog(this);
 }

 dialog.dismiss();
}

public void getContributors(String owner, String repo){
 GitHubApi.getContributors(owner, repo)
 .take(1)
 .observeOn(AndroidSchedulers.mainThread())
 .subscribeOn(Schedulers.newThread())
 .map(new Func1<List<Contributor>, Contributor>() {

 @Override
 public Contributor call(List<Contributor> contributors) {
 return contributors.get(0);
 }
 })
 .subscribe(contributorSub);
}

}
```

如果你对data binding框架是有了解的，上面的代码你能轻松的看懂。

那model层又是什么呢？当然就是那些和数据相关的类，GithubApi等等。

重点来了，viewmodel层呢？好吧，viewmodel层就是是Contributor类！大家不要惊讶，我慢慢的来说。

```

public class Contributor extends BaseObservable{

 private String login;
 private int contributions;

 @Bindable
 public String getLogin(){
 return login;
 }

 @Bindable
 public int getContributions(){
 return contributions;
 }

 public void setLogin(String login){
 this.login = login;
 notifyPropertyChanged(BR.login);
 }

 public void setContributions(int contributions){
 this.contributions = contributions;
 notifyPropertyChanged(BR.contributions);
 }

 @Override
 public String toString() {
 return login + ", " + contributions;
 }
}

```

我们可以看到，Contributor和MVP相比，继承自了BaseObservable，有基础的同学都知道这是为了当Contributor内部的variable改变的时候ui可以同步的作出响应。

我为什么说Contributor是一个viewmodel呢。大家还记得viewmodel的概念吗？view和viewmodel相互绑定在一起，viewmodel的改变会同步到view层，从而view层作出响应。这不就是Contributor和xml中那些组件元素的关系吗？所以，大家不要被binding类迷惑了，data binding框架中的viewmodel是自己定义的那些看似是model类的东西！比如这里的Contributor！

话说到这里，那binding类又是什么呢？其实具体对应到之前MVVM的那张图就很好理解了，我们想一下，binding类的工作是什么？

```

binding = DataBindingUtil.setContentView(this, R.layout.mvvm_activity_main);

binding.setContributor(contributor);

```

首先， binding要通过DataBindingUtil.setContentView()方法将xml， 也就是view层设定。

接着， 通过setXXX()方法将viewmodel层注入进去。

由于这两个工作， view层(xml的各个组件)和viewmodel层(contributor)绑定在了一起。

好了， 大家知道了吗， binding类， 其实就是上图中view和viewmodel中间的那根线啊！ !

## 真理在荒谬被证实以前， 都只是暗室里的装饰

前面讨论了MVC， MVP和MVVM具体的实现方案， 大家肯定都了解了它们三者的关系和使用方式。但是， 这里我想说， 不要把一个框架看作万能的， 其实MVP和MVVM都是有自己的缺陷的！下面我一一来说。

### MVP

MVP的问题在于， 由于我们使用了接口的方式去连接view层和presenter层， 这样就导致了一个问题， 如果你有一个逻辑很复杂的页面， 你的接口会有很多， 十几二十个都不足为奇。想象一个app中有多个这样复杂的页面， 维护接口的成本就会非常的大。

这个问题的解决方案就是你得根据自己的业务逻辑去斟酌着写接口。你可以定义一些基类接口， 把一些公共的逻辑， 比如网络请求成功失败， toast等等放在里面， 之后你再定义新的接口的时候可以继承自那些基类， 这样会好不少。

### MVVM

MVVM的问题呢， 其实和MVC有一点像。data binding框架解决了数据绑定的问题， 但是view层还是会过重， 大家可以看我上面那个MVVM模式下的activity

```
public class MainActivity extends AppCompatActivity {

 private Subscriber<Contributor> contributorSub = new Subscriber<Contributor>() {

 @Override
 public void onStart() {
 showProgress();
 }

 @Override
 public void onCompleted() {

 }
 }
}
```

```
 @Override
 public void onError(Throwable e) {
 }

 @Override
 public void onNext(Contributor contributor) {
 binding.setContributor(contributor);

 dismissProgress();
 }
};

private ProcessDialog dialog;

private MvvmActivityMainBinding binding;

@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 binding = DataBindingUtil.setContentView(this, R.layout.mvvm_activity_main);
}

public void get(View view){
 getContributors("square", "retrofit");
}

public void change(View view){
 if(binding.getContributor() != null){
 binding.getContributor().setLogin("zjutkz");
 }
}

public void showProgress(){
 if(dialog == null){
 dialog = new ProcessDialog(this);
 }

 dialog.showMessage("正在加载...");
}

public void dismissProgress(){
 if(dialog == null){
 dialog = new ProcessDialog(this);
 }

 dialog.dismiss();
}

public void getContributors(String owner, String repo){
 GitHubApi.getContributors(owner, repo)
 .take(1)
```

```
.observeOn(AndroidSchedulers.mainThread())
.subscribeOn(Schedulers.newThread())
.map(new Func1<List<Contributor>, Contributor>() {

 @Override
 public Contributor call(List<Contributor> contributors) {
 return contributors.get(0);
 }
})
.subscribe(contributorSub);
}
```

大家有没有发现，activity在MVVM中应该是view层的，但是里面却和MVC一样写了对model的处理。有人会说你可以把对model的处理放到viewmodel层中，这样不是更符合MVVM的设计理念吗？这样确实可以，但是progressDialog的show和dismiss呢？你怎么在viewmodel层中控制？这是view层的东西啊，而且在xml中也没有，我相信会有解决的方案，但是我们有没有一种更加便捷的方式呢？

## 路漫漫其修远兮，吾将上下而求索

其实，真正的最佳实践都是人想出来的，我们为何不结合一下MVP和MVVM的特点呢？其实谷歌已经做了这样的事，大家可以看下这个。没错，就是MVP+data binding，我们可以使用presenter去做和model层的通信，并且使用data binding去轻松的bind data。还是让我们看代码吧。

首先还是view层。

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
 <data>
 <variable name="contributor" type="zjutkz.com.mvpdatabinding.viewmodel.ContributorView" />
 </data>
 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:id="@+id/container"
 android:orientation="vertical"
 android:fitsSystemWindows="true">

 <Button
 android:id="@+id/get"
 android:text="get"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:onClick="get"/>

 <Button
 android:id="@+id/change"
 android:text="change"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:onClick="change"/>

 <TextView
 android:id="@+id/top_contributor"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:gravity="center"
 android:textSize="30sp"
 android:text="@{contributor.login}"/>
 </LinearLayout>

</layout>
```

```
public class MainActivity extends MvpActivity<ContributorView, ContributorPresenter> implements ContributorView {

 private ProcessDialog dialog;

 private ActivityMainBinding binding;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 binding = DataBindingUtil.setContentView(this, R.layout.activity_main);
 }
}
```

```
@NonNull
@Override
public ContributorPresenter createPresenter() {
 return new ContributorPresenter();
}

public void get(View view){
 getPresenter().get("square", "retrofit");
}

public void change(View view){
 if(binding.getContributor() != null){
 binding.getContributor().setLogin("zjutkz");
 }
}

@Override
public void onLoadContributorStart() {
 showProgress();
}

@Override
public void onLoadContributorComplete(Contributor contributor) {
 binding.setContributor(contributor);
 dismissProgress();
}

public void showProgress(){
 if(dialog == null){
 dialog = new ProcessDialog(this);
 }

 dialog.showMessage("正在加载...");
}

public void dismissProgress(){
 if(dialog == null){
 dialog = new ProcessDialog(this);
 }

 dialog.dismiss();
}
}
```

然后是presenter层

```
public class ContributorPresenter extends MvpBasePresenter<ContributorView> {

 private Subscriber<Contributor> contributorSub = new Subscriber<Contributor>() {

 @Override
 public void onStart() {
 ContributorView view = getView();
 if(view != null){
 view.onLoadContributorStart();
 }
 }

 @Override
 public void onCompleted() {

 }

 @Override
 public void onError(Throwable e) {

 }

 @Override
 public void onNext(Contributor topContributor) {
 ContributorView view = getView();
 if(view != null){
 view.onLoadContributorComplete(topContributor);
 }
 }
 };

 public void get(String owner, String repo){
 GitHubApi.getContributors(owner, repo)
 .take(1)
 .observeOn(AndroidSchedulers.mainThread())
 .subscribeOn(Schedulers.newThread())
 .map(new Func1<List<Contributor>, Contributor>() {

 @Override
 public Contributor call(List<Contributor> contributors) {
 return contributors.get(0);
 }
 })
 .subscribe(contributorSub);
 }
}
```

model层就是GithubApi等等。

我们使用了data binding框架去节省了类似findViewById和数据绑定的时间，又使用了presenter去将业务逻辑和view层分离。

当然这也不是固定的，你大可以在viewmodel中实现相应的接口，presenter层的数据直接发送到viewmodel中，在viewmodel里更新，因为view和viewmodel是绑定的，这样view也会相应的作出反应。

说到这里，我还是想重复刚才的那句话，最佳实践都是人想出来的，用这些框架根本的原因也是为了尽量低的耦合性和尽量高的可复用性。

## 下期预告

还是想好下期写点啥，如果我能想到的话，see you next week.

# 架构设计

# Android简洁架构设计

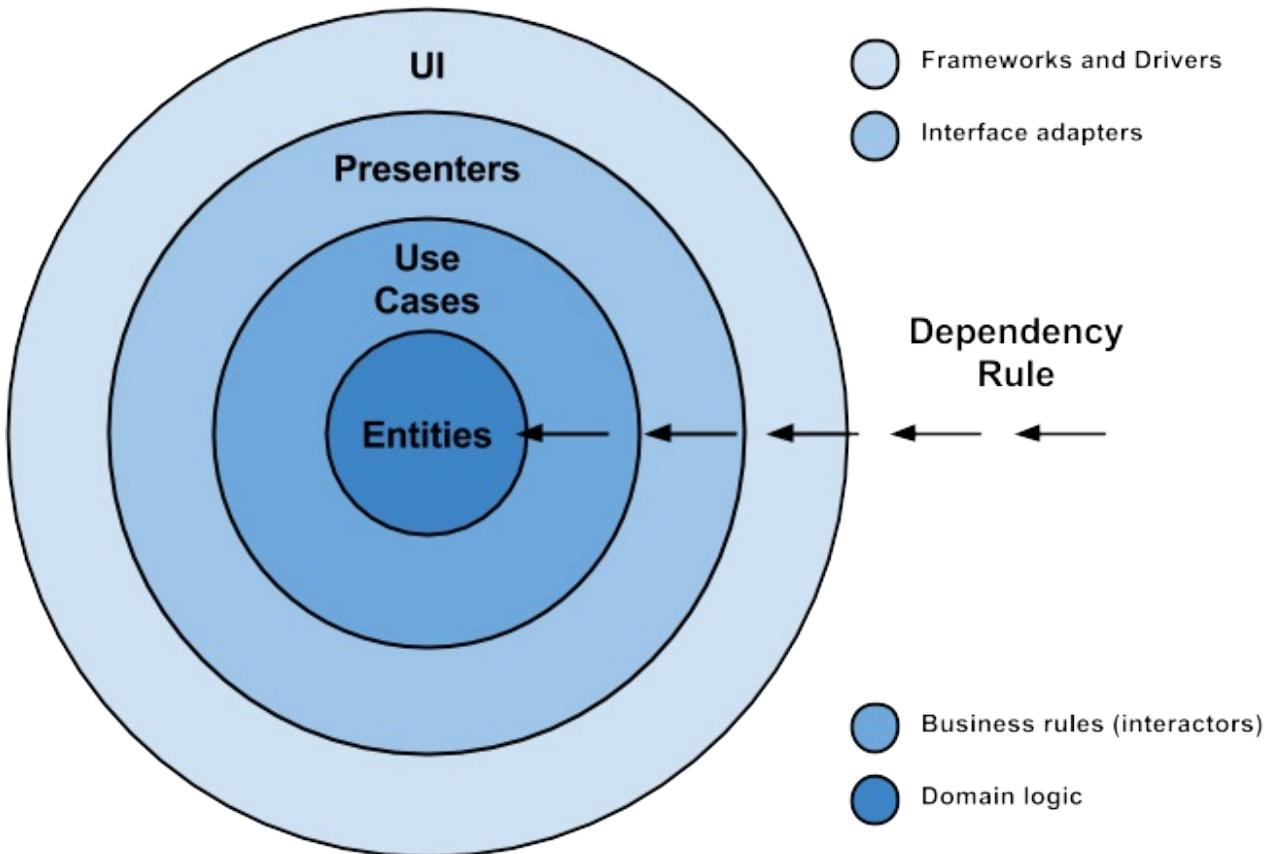
来源:[伯乐在线](#)

过去几个月，在与我Tuenti的几位同事进行友好交流之后，其中有[@pedro\\_g\\_s](#)和[@flipper83](#)（他们都是Android开发中很难对付的人），我认为是时候写篇关于架构Android应用的文章了。这篇文章的目的是介绍近几个月来我所想到的一些方法，以及从调查和实践中学到的东西。

## 开始

我们知道编写高质量软件是既困难又复杂的：不仅是满足需求方面，还要健壮、可维护、可测试，并且足够灵活以适应增长和变化。这就是“代码整洁之道”的来源，并可以成为开发任何软件应用程序的良好方法。思想很简单：代码整洁之道代表构建系统的一组实践：

- 独立于框架。
- 可测试性。
- 独立于UI。
- 独立于数据库。
- 独立于任何外部代理。



只限于使用图示中的4个圆圈并不是必须的，因为这只是语义描述，你还要考虑依赖规则（the Dependency Rule）：源码依赖只应该指向内圈，内圈不应该知道外圈的任何东西。

下面的相关词汇可以帮助熟悉理解这个方法：

- 实体（Entities）：应用的逻辑对象。
- 用例（Use Cases）：用例编排数据流从实体的流入和流出。也叫交互器（Interactor）。
- 接口适配器（Interface Adapters）：这些适配器将数据转换为用例和实体最合适的格式。表示器（Presenter）和控制器（Controller）就位于这里。
- 框架和驱动器（Frameworks and Drivers）：这是所有细节集中的地方，UI、工具、框架等。

为了更好更深入地理解，看下[这篇文章](#)或者[这个视频](#)。

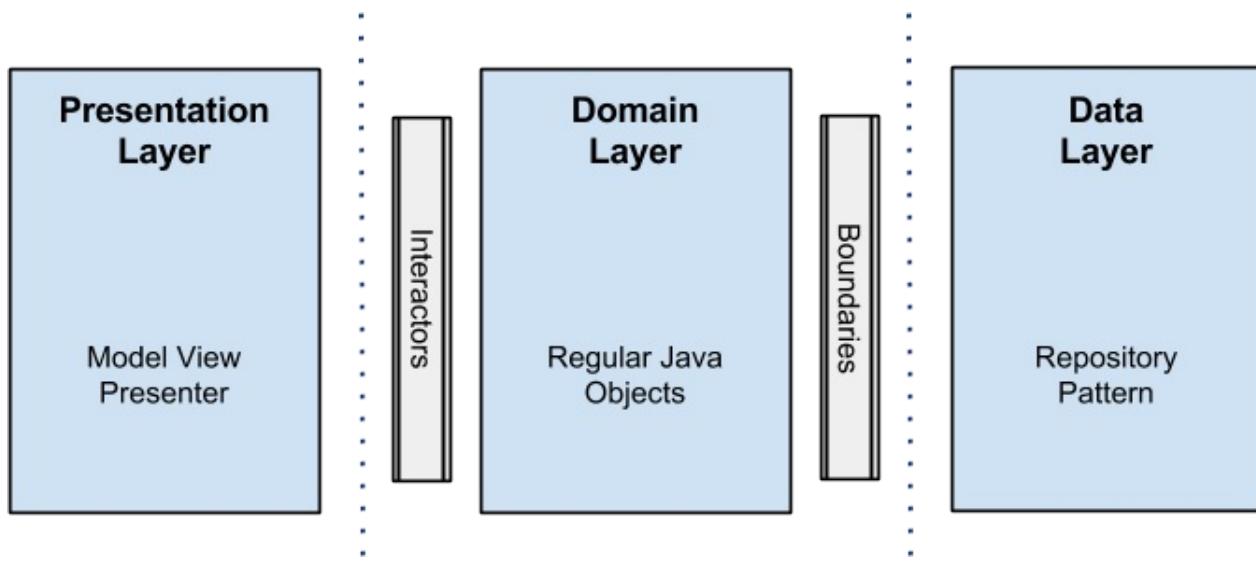
## 场景

我用一个简单的场景来描述事情过程：创建一个小应用，显示从云端获取的朋友或用户列表，点击其中任意一个，将会打开新窗口来显示用户的更多信息。我提供了一个视频来帮助理解我所说的大致过程：

嵌入视频，需自备梯子

## Android架构

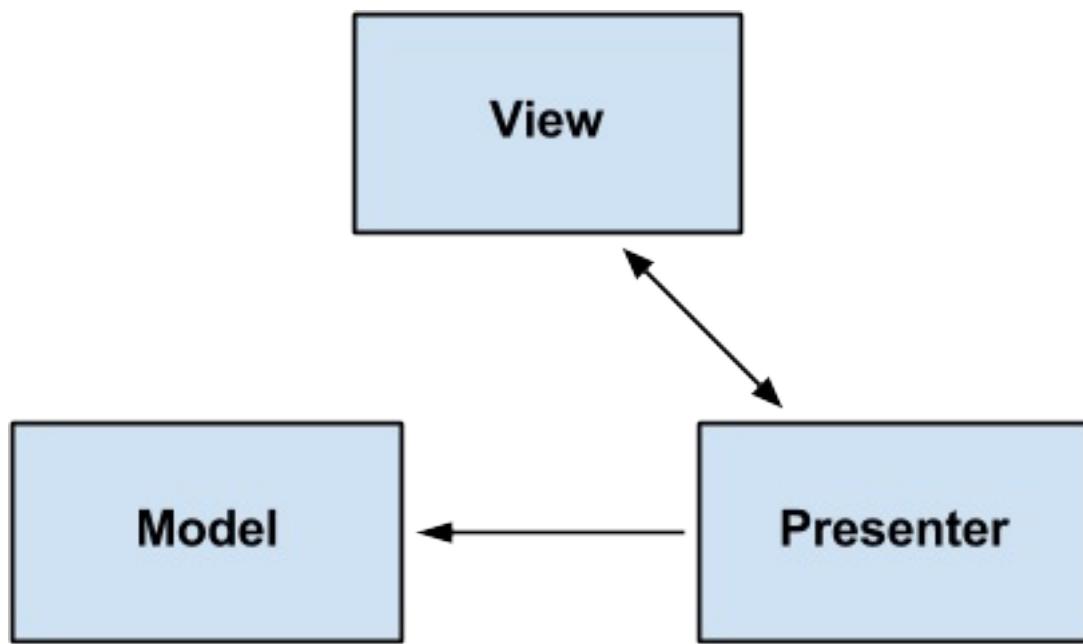
架构的目标是通过将业务规则与外部世界隔离以分离关注点，这样，才可以在不依赖外部元素的情况下测试业务规则。为了达到这个目标，我的建议是将项目拆分为3个不同的层次，每层都有自己的目标，独立地工作。需要提及的是每层都使用自己的数据模型，这样才可以取得依赖（你可以看到，在代码中需要数据映射器（data mapper）来完成数据转换，不想在整个应用之上交叉使用自己的模型总要付出点代价）。下面是一个帮助你理解的模式：



注意：我没有使用任何外部库（除了解析json数据的gson，还有测试用的junit、mockito、robolectri和espresso）。原因是这么做会让例子会更清楚。无论如何，一定不要犹豫使用那些让生活更美好的东西，比如添加ORM来存储磁盘数据，或者任何依赖注入框架，或者任何你熟悉的工具或库。（记住，重新发明轮子不是一个好的实践）。

## 表示层

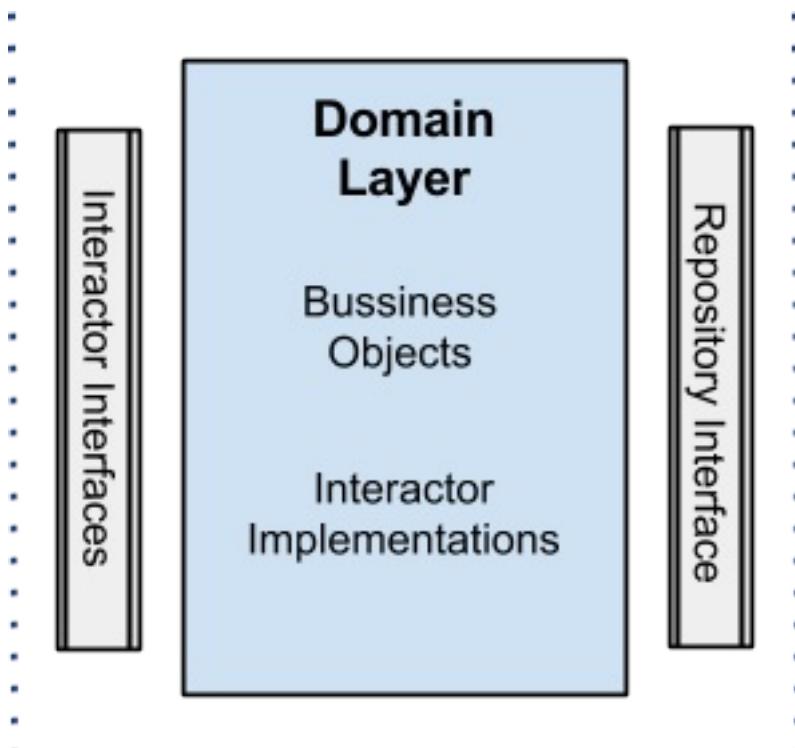
这里实现了与视图和动画有关的逻辑。它只使用一个模型-视图-表示器（Model View Presenter，下文称MVP），但是你可以使用其他任何模式，如MVC或MVVM。在这里我不会深入介绍，这里的片段和活动只有视图，除了UI逻辑之外没有任何其他逻辑，这也是所有渲染发生的地方。这层中的表示器与交互器（用例）共同在Android UI线程之外开启新线程执行这些工作，并通过回调来处理数据，数据将在视图中渲染。



如果你想要一个使用MVP和MVVM更酷的[高效Android UI](#)例子，去看看我朋友Pedro Gómez所做的工作。

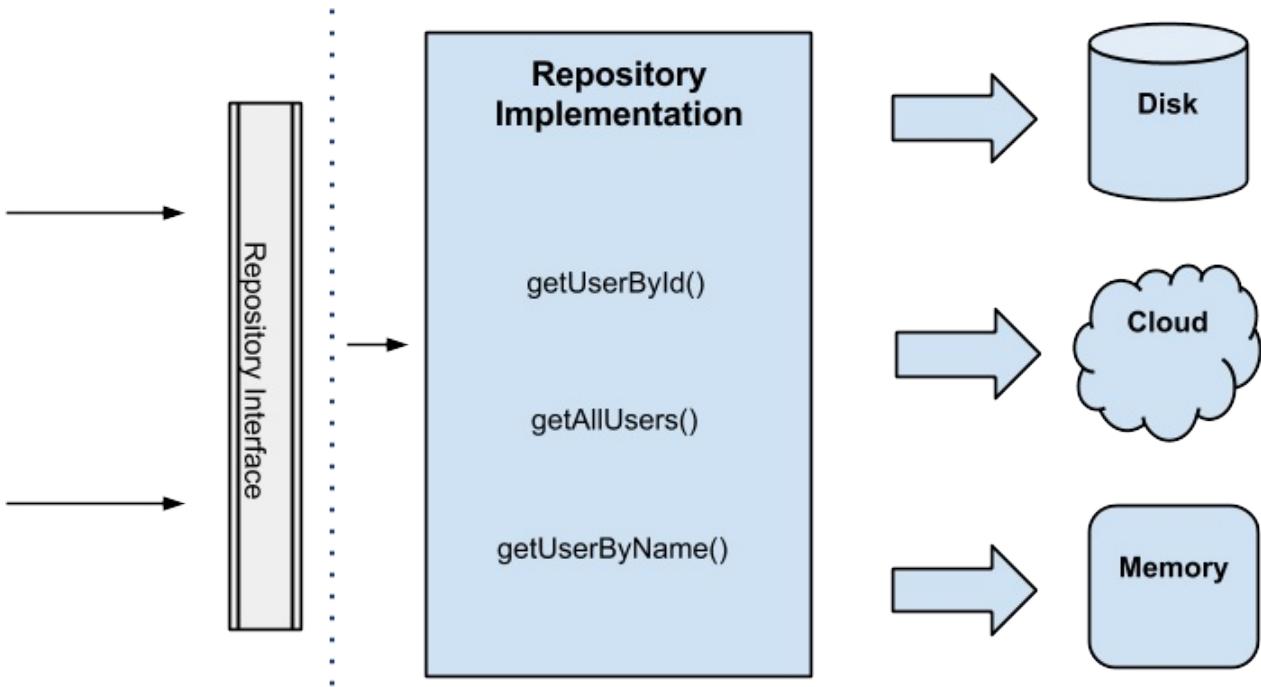
## 领域层

这里的业务规则：所有逻辑都在这层。至于Android项目，你也将会看到所有的交互器（用例）实现。这层是不依赖Android的纯Java模块。所有的外部组件使用接口连接到业务对象。



# 数据层

应用需要的全部数据都来自这层，数据通过一个使用[仓储模式（Repository Pattern）](#)实现的 UserRepository 存取，其策略是通过一个工厂，根据特定的条件来选择不同的数据源。比如，当通过 id 获取用户时，如果用户已存在于缓存中，就会选取磁盘缓存数据源，不然就会查询云端获取数据，之后保存到磁盘缓存。这里的前提是，数据的来源对客户端是透明的，它不关心数据到底来自内存、磁盘还是云端，唯一的事是数据将会到达，然后被获取。



注意：我使用文件系统和Android首选项实现了一个简单的磁盘缓存，仅用于学习目的。再次提醒，如果已经存在某些库可以更好地实现这些功能，你不应该重新发明轮子。

## 错误处理

这个话题很容易引发讨论，如果你能分享自己的解决方案会更棒。我的策略是采用回调，因此，一旦数据仓库有事件发生，回调包含 2 个方法 `onResponse()` 和 `onError()`。最后一个将异常封装到称作“`ErrorBundle`”的包装类中：这种方法会带来很多难处，因为只有一个回调链，错误会一直到达表示层被渲染。代码可读性略显不佳。另外，我也可以实现一个事件总线系统，一旦出错就抛出事件，但是这个方案就像使用 `GOTO` 一样；我认为，有时候如果你订阅了多个事件，不仔细控制的话很容易迷失。

## 测试

关于测试，我倾向于根据不同层选择多重方案：

- 表示层：使用Android instrumentation或espresso进行集成和功能测试。
- 领域层：JUnit和mockito进行单元测试。
- 数据层：Robolectric（由于这层有Android依赖）和junit、mockito进行集成和单元测试。

## 代码展示

我知道你可能想知道代码在哪里，对吧？这里是[github链接](#)，你可以找到。关于目录结构要说明的是，不同层使用模块来表示：

- presentation：表示层的Android模块。
- domain：无Android依赖的java模块。
- data：所有数据存放的Android模块。
- data-test：数据层的测试。由于使用Robolectric有些限制，我不得不在新的Java模块中使用。

## 结论

就像Bob大叔说过的，“架构在于目的而非框架”，我完全赞同。当然，有很多不同的做事方式（不同的实现），我确定你（像我一样）每天都面对很多挑战，但是使用这些技巧，你可以保证应用将会：

- 易于维护。
- 易于测试。
- 非常内聚。
- 解耦。

作为总结，我非常推荐你试试，分享你的结果和体验，当然你也会发现更好的方法：我们都知道持续的提升总是很好充满正能量的事情。希望这篇文章对你有用，非常欢迎你的反馈。

## 链接和资源

- 源代码：<https://github.com/android10/Android-CleanArchitecture>
- Bob大叔的整洁架构
- 架构在于目的而非框架

- 模型 视图 表示器
- Martin Fowler仓储模式
- Android设计模式演示稿

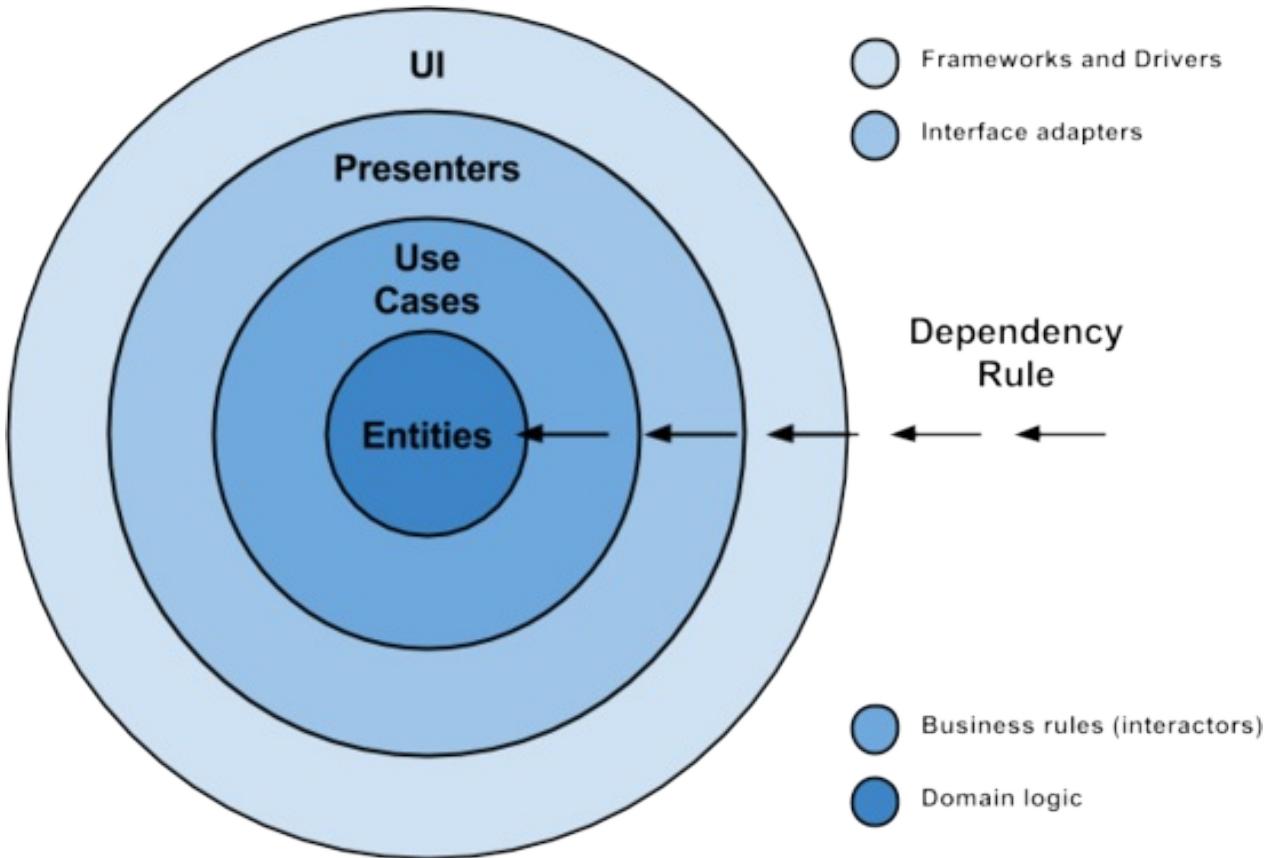
# 解析Android架构设计原则

来源：[伯乐在线](#)

原文出处：[Fernando Cejas](#) 译文出处：[张挥戈](#)

嘿！经过一段时间收集了大量反馈意见后，我认为应该来说说这个话题了。我会在这里给出我认为构建现代移动应用（Android）的好方法，这会是另一番体味。

开始之前，假设你已经阅读过我之前撰写的文章[“Architecting Android...The clean way?”](#)。如果还没有阅读过，为了更好地理解这篇文章，应借此机会读一读：



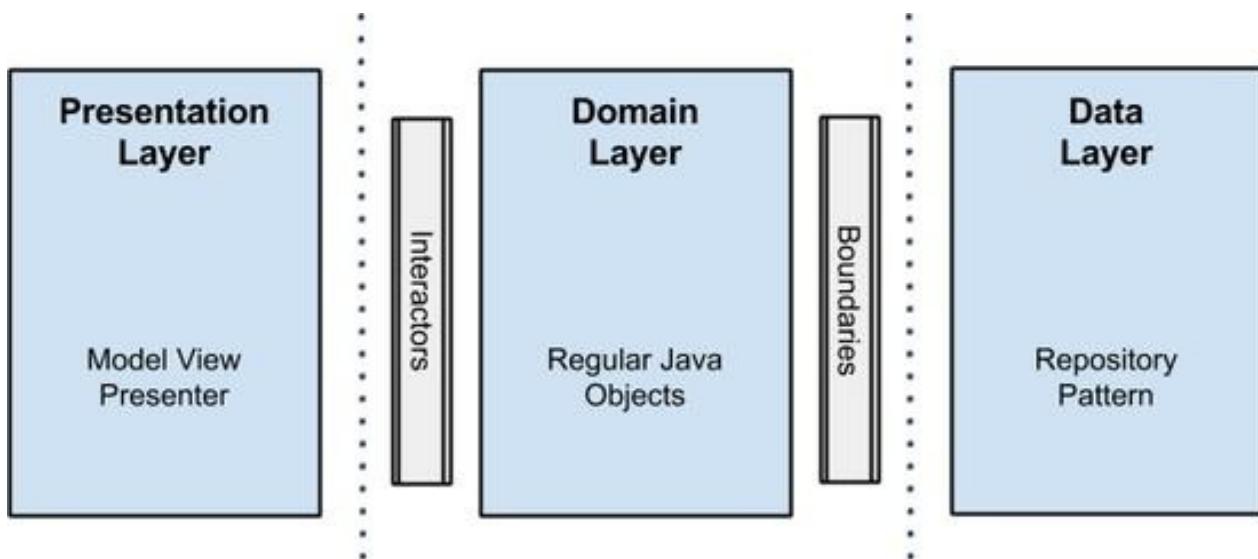
## 架构演变

演变意味着一个循序渐进的过程，由某些状态改变到另一种不同的状态，且新状态通常更好或更复杂。

照这么一说，软件是随着时间发展和改变的，是架构上的发展和改变。实际上，好的软件设计必须能够帮助我们发展和扩充解决方案，保持其健壮性，而不必每件事都重写代码（虽然在某些情况下重写的方法更好，但是那是另一篇文章的话题，所以相信我，让我们

聚焦在前面所讨论的话题上）。

在这篇文章中，我将讲解我认为是必需的和重要的要点，为了保持基本代码条理清晰，要记住下面这张图片，我们开始吧！



## 响应式方法： RxJava

因为已经有[很多这方面的文章](#)，还有这方面[做得很好、令人景仰的人](#)，所以我不打算在这里讨论RxJava的好处（[我假设您已经对它有所体验了](#)）。但是，我将指出在Android应用程序开发方面的有趣之处，以及如何帮助我形成第一个清晰的架构的方法。

首先，我选择了一种响应式的模式通过转换usecase（在这个清晰的架构命名规则中，其被称为interactor）返回Observables，表示所有底层都遵循这一链条，也返回Observables。

```

public abstract class UseCase {

 private final ThreadExecutor threadExecutor;
 private final PostExecutionThread postExecutionThread;

 private Subscription subscription = Subscriptions.empty();

 protected UseCase(ThreadExecutor threadExecutor,
 PostExecutionThread postExecutionThread) {
 this.threadExecutor = threadExecutor;
 this.postExecutionThread = postExecutionThread;
 }

 protected abstract Observable buildUseCaseObservable();

 public void execute(Subscriber UseCaseSubscriber) {
 this.subscription = this.buildUseCaseObservable()
 .subscribeOn(Schedulers.from(threadExecutor))
 .observeOn(postExecutionThread.getScheduler())
 .subscribe(UseCaseSubscriber);
 }

 public void unsubscribe() {
 if (!subscription.isUnsubscribed()) {
 subscription.unsubscribe();
 }
 }
}

```

正如你所看到的，所有用例继承这个抽象类，并实现抽象方法

`buildUseCaseObservable()`。该方法将建立一个Observables，它承担了繁重的工作，还要返回所需的数据。

需要强调是，在`execute()`方法中，要确保Observables是在独立线程执行，因此，要尽可能减轻阻止android主线程的程度。其结果就是会通过android主线程调度程序将主线程压入线程队列的尾部（push back）。

到目前为止，我们的Observables启动并且运行了。但是，正如你所知，必须要观察它所发出的数据序列。要做到这一点，我改进了presenters（MVP模式表现层的一部分），把它变成了观察者（Subscribers），它通过用例对发出的项目做出“react”，以便更新用户界面。

观察者是这样的：

```

private final class UserListSubscriber extends DefaultSubscriber<List<User>> {

 @Override public void onCompleted() {
 userListPresenter.this.hideViewLoading();
 }

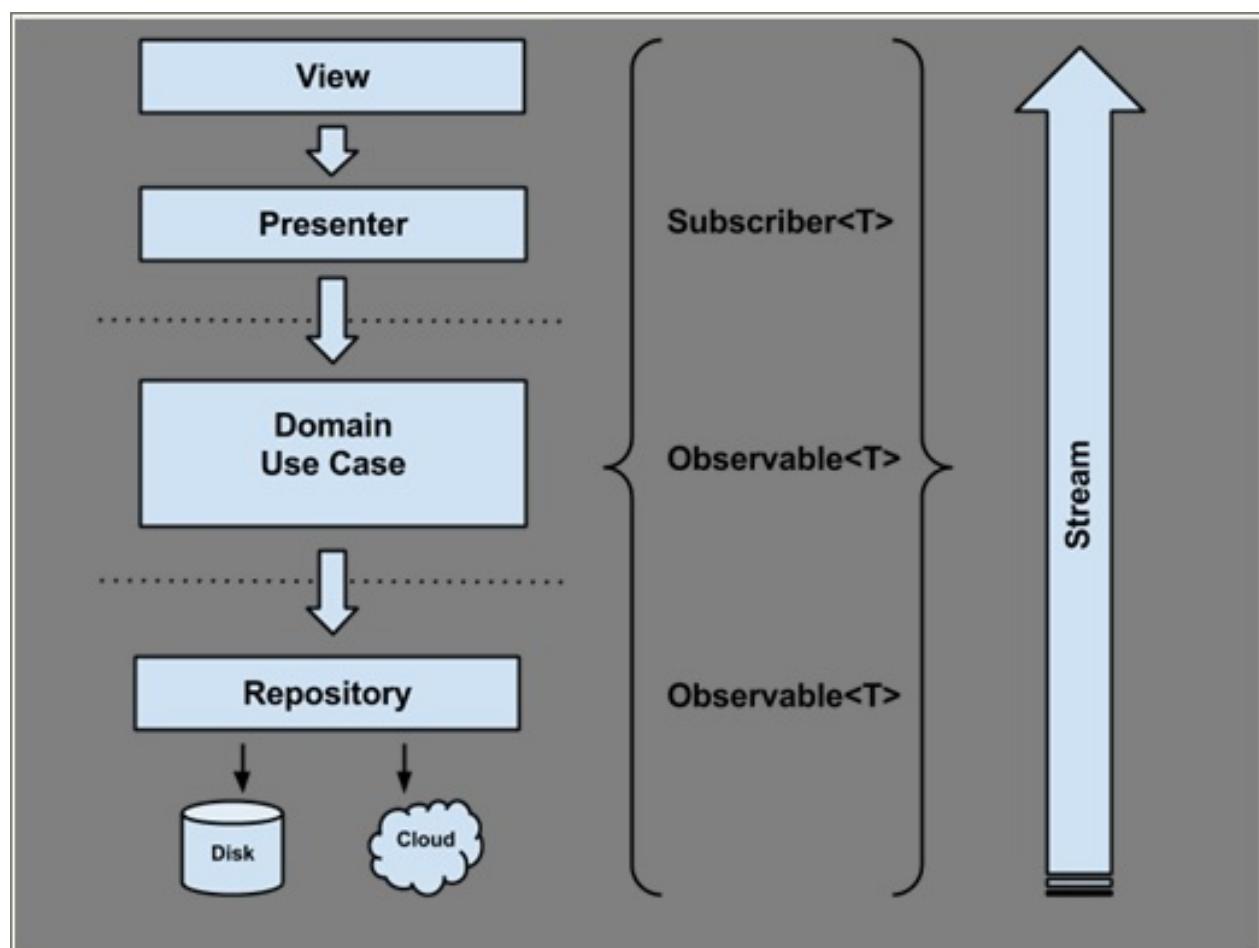
 @Override public void onError(Throwable e) {
 userListPresenter.this.hideViewLoading();
 userListPresenter.this.showErrorMessage(new DefaultErrorBundle((Exception) e));
 userListPresenter.this.showViewRetry();
 }

 @Override public void onNext(List<User> users) {
 userListPresenter.this.showUsersCollectionInView(users);
 }
}

```

每个观察者都是每个presenter的内部类，并实现了一个Defaultsubscriber接口，创建了基本的默认错误处理。

将所有的片段放在一起后，通过下面的图，你可以获得完整的概念：



让我们列举一些摆脱基于RxJava方法的好处：

在观察者（Subscribers）与被观察者（Observables）之间去耦合：更加易于维护和测试。

- 简化异步任务：如果要求多个异步执行时，如果需要一个以上异步执行的级别，Java的thread和future的操作和同步比较复杂，因此通过使用调度程序，我们可以很方便地（不需要额外工作）在后台与主线程之间跳转，特别是当我们需要更新UI时。还可以避免“回调的坑”——它使我们代码可读性差，且难以跟进。
- 数据转换/组成：在不影响客户端情况下，我们能够整合多个Observables，使解决方案更灵活。
- 错误处理：在任何Observables内发生错误时，就向消费者发出信号。

从我的角度看有一点不足，甚至要为此需要付出代价，那些还不熟悉概念的开发人员还是要遵循学习曲线。但你从中得到了极有价值的东西。为了成功而reactive起来吧！

## 依赖注入：Dagger 2

关于依赖注入，因为我[已经写了一篇完整的文章](#)，我不想说太多。强烈建议你阅读它，这样我们就可以接着说下面的内容了。

值得一提的是，通过实现一个像Dagger 2那样的依赖注入框架我们能够获得：

- 组件重用，因为依赖的对象可以在外部注入和配置。
- 当注入对象作为协作者（collaborators）时，由于对象的实例存在于在一个隔离和解耦地方，这样在我们的代码库中，就不需要做很多的改变，就可以改变任何对象的实现。
- 依赖可以注入到一个组件：这些将这些模拟实现的依赖对象注入成为可能，这使得测试更容易。

## Lambda表达式：Retrolambda

没有人会抱怨在代码中使用Java 8的lambda表达式，甚至在简化并摆脱了很多样板代码以后，使用得更多，如你看到这段代码：

```

private final Action1<UserEntity> saveToCacheAction =
 userEntity -> {
 if (userEntity != null) {
 CloudUserDataStore.this.userCache.put(userEntity);
 }
 };

```

然而，我百感交集，为什么呢？我们曾在[@SoundCloud](#)讨论[Retrolambda](#)，主要是是否使用它，结果是：

- 1. 赞成的理由：
    - Lambda表达式和方法引用
    - “try-with-resources”语句
    - 使用karma做开发
- 1. 反对的理由：
    - Java 8 API的意外使用
    - 十分令人反感的第三方库
    - 要与Android一起使用的第三方插件Gradle

最后，我们认定它不能为我们解决任何问题：你的代码看起来很好且具有可读性，但这不是我们与之共存的东西，由于现在所有功能最强大的IDE都包含代码折叠式选项，这就涵盖这一需求了，至少是一个可接受的方式。

说实话，尽管我可能会在业余时间的项目中使用它，但在这里使用它的主要原因是尝试和体验Android中Lambda表达式。是否使用它由你自己决定。在这里我只是展示我的视野。当然，对于这样一项了不起的工作，这个库的作者值得称赞。

## 测试方法

在测试方面，与示例的第一个版本相关的部分变化不大：

- 表现层：用Espresso 2和Android Instrumentation测试框架测试UI。
- 领域层：JUnit + Mockito —— 它是Java的标准模块。
- 数据层：将测试组合换成了Robolectric 3 + JUnit + Mockito。这一层的测试曾经存在于单独的Android模块。由于当时（当前示例程序的第一个版本）没有内置单元测试的支持，也没有建立像robolectric那样的框架，该框架比较复杂，需要一群黑客的帮助才能让其正常工作。

幸运的是，这都是过去的一部分，而现在所有都是即刻可用，这样我可以把它们重新放到数据模块内，专门为默认的测试路径：src/test/java。

## 包的组织

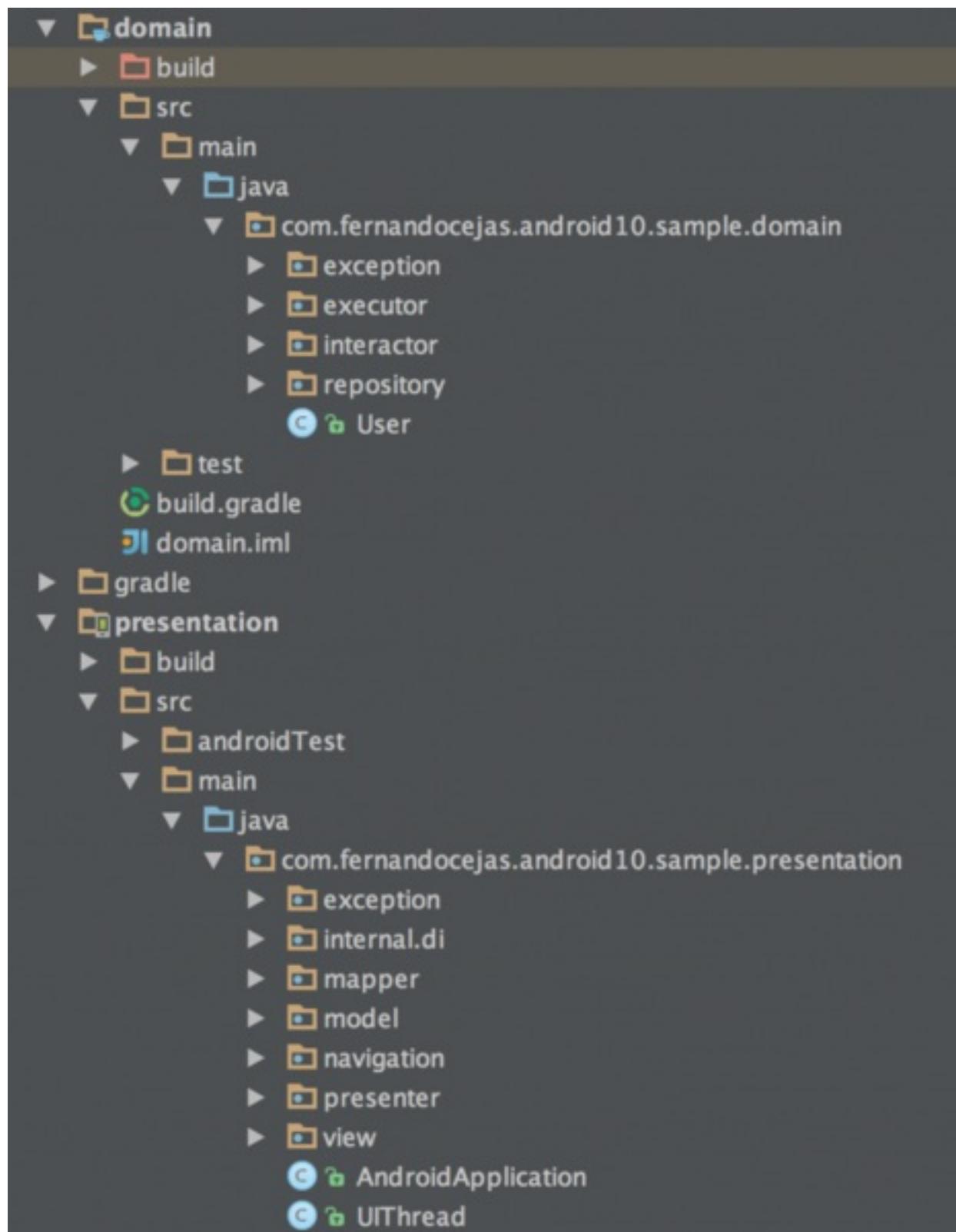
我认为一个好的架构关键因素之一是代码/包的组织：程序员浏览源代码遇到的第一件事情就是包结构。一切从它流出，一切依赖于它。

我们能够辨别出将应用程序封装进入包（package）的2个路径：

- 按层分包：每一个包（package）中包含的项通常不是彼此密切相关的。这样包的内聚性低、模块化程度低，包之间偶合度高。因此，编辑某个特性要编辑来自不同包的文件。另外，单次操作几乎不可能删除掉某个功能特性。
- 按特性分包：用包来体现特性集。把所有相关某一特性（且仅特性相关）的项放入一个包中。这样包的内聚性高，模块化程度高，包之间偶合度低。紧密相关的项放在一起。它们没有分散到整个应用程序中。我的建议是去掉按特性分包，会带来的好处有以下主要几点：

模块化程度更高

- 代码导航更容易
- 功能特性的作用域范围最小化了
- 如果与功能特性团队一起工作（就像我们在@SoundCloud的所作所为），也会是非常有趣的事情。代码的所有权会更容易组织，也更容易被模块化。在许多开发人员共用一个代码库的成长型组织当中，这是一种成功。



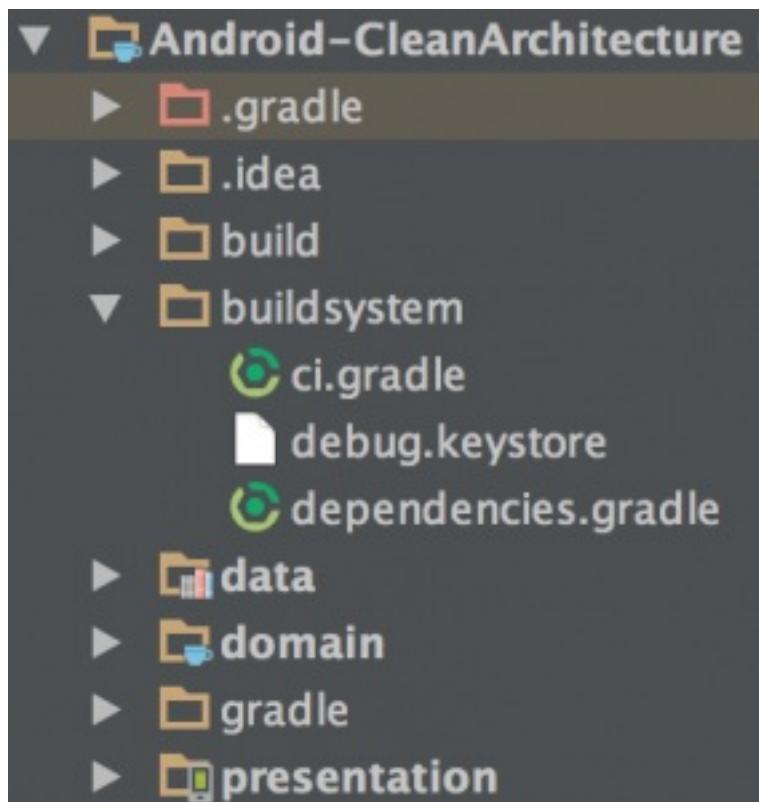
如你所见，我的方法看起来就像按层分包：这里我可能会犯错（例如，在“users”下组织一切），但在这种情况下我会原谅自己，因为这是个以学习为目的的例子，而且我想显示的是清晰架构方法的主要概念。领会其意，切勿盲目模仿:-)。

## 还需要做的事：组织构建逻辑

我们都知道，房子是从地基上建立起来的。软件开发也是这样，我想说的是，从我的角度来看，构建系统（及其组织）是软件架构的重要部分。

在Android平台上，我们采用Gradle，它事实上是一种与平台无关的构建系统，功能非常强大。这里的想法是通过一些提示和技巧，让你组织构建应用程序时能够得到简化。

- 在单独的gradle构建文件中按功能对内容进行分组



```

def ciServer = 'TRAVIS'
def executingOnCI = "true".equals(System.getenv(ciServer))

// Since for CI we always do full clean builds, we don't want to pre-dex
// See http://tools.android.com/tech-docs/new-build-system/tips
subprojects {
 project.plugins.whenPluginAdded { plugin ->
 if ('com.android.build.gradle.AppPlugin'.equals(plugin.class.name) ||
 'com.android.build.gradle.LibraryPlugin'.equals(plugin.class.name)) {
 project.android.dexOptions.preDexLibraries = !executingOnCI
 }
 }
}

```

```
apply from: 'buildsystem/ci.gradle'
apply from: 'buildsystem/dependencies.gradle'

buildscript {
 repositories {
 jcenter()
 }
 dependencies {
 classpath 'com.android.tools.build:gradle:1.2.3'
 classpath 'com.neenbedankt.gradle.plugins:android-apt:1.4'
 }
}

allprojects {
 ext {
 ...
 }
}
...
```

因此，你可以用“`apply from: 'buildsystem/ci.gradle'`”插入到任何Gradle建立的文件中进行配置。不要把所有都放置在一个`build.gradle`文件中，否则就是去创建一个怪物，这是教训。

- 创建依赖关系图

```
...

ext {
 //Libraries
 daggerVersion = '2.0'
 butterKnifeVersion = '7.0.1'
 recyclerViewVersion = '21.0.3'
 rxJavaVersion = '1.0.12'

 //Testing
 robolectricVersion = '3.0'
 jUnitVersion = '4.12'
 assertJVersion = '1.7.1'
 mockitoVersion = '1.9.5'
 dexmakerVersion = '1.0'
 espressoVersion = '2.0'
 testingSupportLibVersion = '0.1'

 ...

 domainDependencies = [
 daggerCompiler: "com.google.dagger:dagger-compiler:${daggerVersion}",
 dagger: "com.google.dagger:dagger:${daggerVersion}",
 javaxAnnotation: "org.glassfish:javax.annotation:${javaxAnnotationVersion}",
 rxJava: "io.reactivex:rxjava:${rxJavaVersion}",
]

 domainTestDependencies = [
 junit: "junit:junit:${jUnitVersion}",
 mockito: "org.mockito:mockito-core:${mockitoVersion}",
]

 ...

 dataTestDependencies = [
 junit: "junit:junit:${jUnitVersion}",
 assertj: "org.assertj:assertj-core:${assertJVersion}",
 mockito: "org.mockito:mockito-core:${mockitoVersion}",
 robolectric: "org.robolectric:robolectric:${robolectricVersion}",
]
}
```

```
apply plugin: 'java'

sourceCompatibility = 1.7
targetCompatibility = 1.7

...

dependencies {
 def domainDependencies = rootProject.ext.domainDependencies
 def domainTestDependencies = rootProject.ext.domainTestDependencies

 provided domainDependencies.daggerCompiler
 provided domainDependencies.javaxAnnotation

 compile domainDependencies.dagger
 compile domainDependencies.rxJava

 testCompile domainTestDependencies.junit
 testCompile domainTestDependencies.mockito
}
```

如果想在项目的不同模块间重用相同的组件版本，这很好；否则就要在不同的模块间使用不同的版本的组件依赖。另外一点，你是在同一个地方控制依赖关系，像组件版本发生冲突这样的事情一眼就能看出来。

## 结语

到目前为止讲了那么多，一句话，要记住没有灵丹妙药。但好的软件架构会帮助代码保持清晰和健壮，还可以保持代码的可扩展性，易于维护。

我想指出一些事情。面对软件存在的问题，要报以本应当解决的态度：

- 遵守SOLID原则
- 不要过度思考（不过度工程化）
- 务实
- 尽可能降低（Android）框架中模块的依赖性

## 源代码

- [Clean architecture github repository – master branch](#)
- [Clean architecture github repository – releases](#)

## 延伸阅读

- [Architecting Android..the clean way](#)
- [Tasting Dagger 2 on Android](#)
- [The Mayans Lost Guide to RxJava on Android](#)
- [It is about philosophy: Culture of a good programmer](#)

## 参考资料

- [RxJava wiki by Netflix](#)
- [Framework bound by Uncle Bob](#)
- [Gradle user guide](#)
- [Package by feature, not layer](#)

# 案例

# Repository设计模式

来源:[Repository设计模式](#)

原文链接: <https://medium.com/@krzychukosobudzki/repository-design-pattern-bc490b256006#.r0my8xrj6>

在Android中我们需要将数据存储起来以持久化，大部分情况下我们使用SQLite存储数据。但当你想通过一种简单而只读的方式存储，那该怎么做呢？没错，你要找一个好的ORM。你可以开始写制表的类并设计所有需要的方法。

```
public interface NewsesDao {
 void add(News news);

 void update(News news);

 void remove(News news);

 News getById();

 List<News> getNewest();

 List<News> getAll();

 List<News> getByCategory(Category category);

 List<News> getNewerThan(long date);
}
```

看起来熟悉吗？这个有错吗？当然没有，但很有提高的空间。

与其写一个（或两个）超级类做所有的事情，还不如遵循单一职责原则并应用[Repository模式](#)。

Repository是domain层与数据层的中介，如同一个内存中的domain模型集合。使用端会以声明的方式构造查询条件并将其发送给Repository以获取数据。Repository可以增删对象，就如一个对象集合，而其内部封装的数据映射代码会在后台进行逻辑操作。

我们根据Repository的基础定义开始写代码：

```

public interface Repository<T> {
 void add(T item);

 void update(T item);

 void remove(T item);

 List<T> query(Specification specification);
}

```

在这里我用Specification替代了Criteria，二者区别除了名字以外，还有Specification不关注数据会存在哪里。在后续使用过程中，我发现有两个方法特别有用，极建议添加。

```

public interface Repository<T> {
 void add(T item);

 void add(Iterable<T> items);

 void update(T item);

 void remove(T item);

 void remove(Specification specification);

 List<T> query(Specification specification);
}

```

当然还是要根据具体需求来写。在几乎所有的应用中你都需要同时存储于删除多个条目。Repository是不是简化了你的工作？

回到Specification，这只是一个普通的名字接口，一旦定义了Repository是什么，就可以开始实现Specification了：

```

public interface SqlSpecification extends Specification {
 String toSqlQuery();
}

```

基于数据库的基本Repository实现如下：

```

public class NewsSqlRepository implements Repository<News> {
 private final SQLiteOpenHelper openHelper;

 private final Mapper<News, ContentValues> toContentValuesMapper;
 private final Mapper<Cursor, News> toNewsMapper;
}

```

```
public NewsSqlRepository(SQLiteOpenHelper openHelper) {
 this.openHelper = openHelper;

 this.toContentValuesMapper = new NewsToContentValuesMapper();
 this.toNewsMapper = new CursorToNewsMapper();
}

@Override
public void add(News item) {
 add(Collections.singletonList(item));
}

@Override
public void add(Iterable<News> items) {
 final SQLiteDatabase database = openHelper.getWritableDatabase();
 database.beginTransaction();

 try {
 for (News item : items) {
 final ContentValues contentValues = toContentValuesMapper.map(item);

 database.insert(NewsTable.TABLE_NAME, null, contentValues);
 }

 database.setTransactionSuccessful();
 } finally {
 database.endTransaction();
 database.close();
 }
}

@Override
public void update(News item) {
 // TODO to be implemented
}

@Override
public void remove(News item) {
 // TODO to be implemented
}

@Override
public void remove(Specification specification) {
 // TODO to be implemented
}

@Override
public List<News> query(Specification specification) {
 final SqlSpecification sqlSpecification = (SqlSpecification) specification;

 final SQLiteDatabase database = openHelper.getReadableDatabase();
 final List<News> newses = new ArrayList<>();
```

```

 try {
 final Cursor cursor = database.rawQuery(sqlSpecification.toSqlQuery(), new String[] {});

 for (int i = 0, size = cursor.getCount(); i < size; i++) {
 cursor.moveToPosition(i);

 newses.add(toNewsMapper.map(cursor));
 }

 cursor.close();
 }

 return newses;
 } finally {
 database.close();
 }
}
}

```

当使用Lambda和Kotlin时代码会显得简短一些。在这个例子中我故意只实现了add和query方法因为其他的方法非常类似。相信你已经注意到了Mapper接口，其职责是将一个对象映射到另一个。在这个例子中Mapper起到了很大的作用。在使用Repository模式时，Mapper类让你可以只关注重要的事情。

```

public interface Mapper<From, To> {
 To map(From from);
}

```

为了进行数据库或其他存储的查询，我们需要实现Specification：

### NewestNewsesSpecification.class

```

public class NewestNewsesSpecification implements SqlSpecification {

 @Override
 public String toSqlQuery() {
 return String.format(
 "SELECT * FROM %1$s ORDER BY `%2$s` DESC;",
 NewsTable.TABLE_NAME,
 NewsTable.Fields.DATE
);
 }
}

```

### NewsByIdSpecification.class

```

public class NewsByIdSpecification implements SqlSpecification {
 private final int id;

 public NewsByIdSpecification(final int id) {
 this.id = id;
 }

 @Override
 public String toSqlQuery() {
 return String.format(
 "SELECT * FROM %1$s WHERE `%2$s` = %3$d;",
 NewsTable.TABLE_NAME,
 NewsTable.Fields.ID,
 id
);
 }
}

```

### NewsesByCategorySpecification.class

```

public class NewsesByCategorySpecification implements SqlSpecification {
 private final Category category;

 public NewsesByCategorySpecification(final Category category) {
 this.category = category;
 }

 @Override
 public String toSqlQuery() {
 return String.format(
 "SELECT * FROM %1$s WHERE `%2$s` = '%3$d',
 NewsTable.TABLE_NAME,
 NewsTable.Fields.CATEGORY_ID,
 category.getId()
);
 }
}

```

Specification很简单，一般只有一个方法。你想写多少Specification都行，而不会影响DAO类。而且这样测试就变得更加简单，通过接口，Specification可以被随意模拟、替换，甚至可以用假数据测试。你能想象与又臭又长的DAO相比Specification类有多么简单易懂吗？

Repository可以与MVP很好地融合，也能在第三方开源框架中使用。如果你用Dagger2你需要指定在某处要注入哪种Repository实现。

```

public class LatestNewsesPresenter implements Presenter<LatestNewsesView> {
 private final LatestNewsesView view;
 private final Repository<News> repository;

 public LatestNewsesPresenter(LatestNewsesView view, Repository<News> repository) {
 this.view = view;
 this.repository = repository;
 }

 @Override
 public void onCreate(Bundle savedInstanceState) {
 final List<News> newses = repository.query(new NewestNewsesSpecification());
 view.setNewses(newses);
 }

 @Override
 public void onDestroy() {
 }
}

```

下面开始复杂一些

现在需求变了，你不能用SQLite了，现在要用Realm。

//那边的程序员我听到了你们的声音！！！

如果你已经用DAO实现了应用，难道需要重写一堆一堆的类，包括那些非数据存储层的吗？因为你在使用Repository模式，这都是不必的，你只需要实现Repository和Specification来适应新需求即可（下面就以Realm为例）。你不需要修改其他的类，现在让我们看一下新的实现：

```

public class NewsRealmRepository implements Repository<News> {
 private final RealmConfiguration realmConfiguration;

 private final Mapper<NewsRealm, News> toNewsMapper;

 public NewsRealmRepository(final RealmConfiguration realmConfiguration) {
 this.realmConfiguration = realmConfiguration;
 this.toNewsMapper = new NewsRealmToNewsMapper();
 }

 @Override
 public void add(final News item) {
 final Realm realm = Realm.getInstance(realmConfiguration);

```

```
realm.executeTransaction(new Realm.Transaction() {
 @Override
 public void execute(Realm realm) {
 final NewsRealm newsRealm = realm.createObject(NewsRealm.class);
 newsRealm.setCategory(item.getCategory());
 newsRealm.setDate(item.getDate());
 newsRealm.setTitle(item.getTitle());
 newsRealm.setText(item.getText());
 }
});

realm.close();
}

@Override
public void add(Iterable<News> items) {
 // TODO to be implemented
}

@Override
public void update(News item) {
 // TODO to be implemented
}

@Override
public void remove(News item) {
 // TODO to be implemented
}

@Override
public void remove(Specification specification) {
 // TODO to be implemented
}

@Override
public List<News> query(Specification specification) {
 final RealmSpecification realmSpecification = (RealmSpecification) specification;

 final Realm realm = Realm.getInstance(realmConfiguration);
 final RealmResults<NewsRealm> realmResults = realmSpecification.toRealmResults();

 final List<News> newses = new ArrayList<>();

 for (NewsRealm news : realmResults) {
 newses.add(toNewsMapper.map(news));
 }

 realm.close();

 return newses;
}
}
```

```
}
```

Specification接口也要重写一下：

```
public interface RealmSpecification extends Specification {
 RealmResults<NewsRealm> toRealmResults(Realm realm);
}
```

以及一个RealmSpecification的实现：

```
public class NewsByIdSpecification implements RealmSpecification {
 private final int id;

 public NewsByIdSpecification(final int id) {
 this.id = id;
 }

 @Override
 public RealmResults<NewsRealm> toRealmResults(Realm realm) {
 return realm.where(NewsRealm.class)
 .equalTo(NewsRealm.Fields.ID, id)
 .findAll();
 }
}
```

现在你就可以拿Realm替代SQLite了。（还记得Presenter中依赖是如何被定义的吗？）

## 奖

这里还有一个用例，你可以用Repository模式提供缓存。就像前面的例子一样，写一个CacheSpecification。

```
public interface CacheSpecification<T> extends Specification {
 boolean accept(T item);
}
```

与其他接口一起使用：

```
public class NewsByIdSpecification implements RealmSpecification, CacheSpecification<News> {
 private final int id;

 public NewsByIdSpecification(final int id) {
 this.id = id;
 }

 @Override
 public RealmResults<NewsRealm> toRealmResults(Realm realm) {
 return realm.where(NewsRealm.class)
 .equalTo(NewsRealm.Fields.ID, id)
 .findAll();
 }

 @Override
 public boolean accept(News item) {
 return id == item.getId();
 }
}
```

最重要的是，不是所有的Specification都需要实现所有的接口，如果在缓存中只用到NewsByIdSpecification，那就不需要让NewsByCategorySpecification也实现CacheSpecification。

现在你的代码更加整洁了，你的队友更加爱你了，产品经理不用死了，就算你跳槽了，接替你的人也不会砍你了。

欢迎关注我的公众号，将零碎时间都用在刷干货上！



# 笔记 - Android应用架构 (Android Dev Summit 2015)

来源:[markzhai's home](#)

视频见：[https://www.youtube.com/watch?v=BlkJzgjzL0c&feature=em-subs\\_digest](https://www.youtube.com/watch?v=BlkJzgjzL0c&feature=em-subs_digest)

印度哥们的发音每次都能让我一阵沉醉。

## 尽快行动

- 早期的设计抉择对app的影响很大
- 基本架构会让你思考需要解决的问题变得更容易或困难。

## 哪种模式？

- MVC
- MVP
- Reactive
- Cairngorm
- Flux
- fdsafdsa
- MVVM
- CLEAN

## 这不是一个库的调查

- 很多很棒的libraries展示了思考app的不同方式
- 趋势迅速改变，但一些挑战是永恒的
- 久而久之“熔岩流”发生了

Lava Flow 熔岩流：解释一下，大致就是说一些因为种种状况写得不太好的代码，恩，比如产品/PM/某些开发老大等呵呵呵催进度的时候，被加入到实际生产环境，而其实际仍处于开发中的阶段，导致后期需要维护原先未完成状态的设计的兼容性，比如各种API/命名

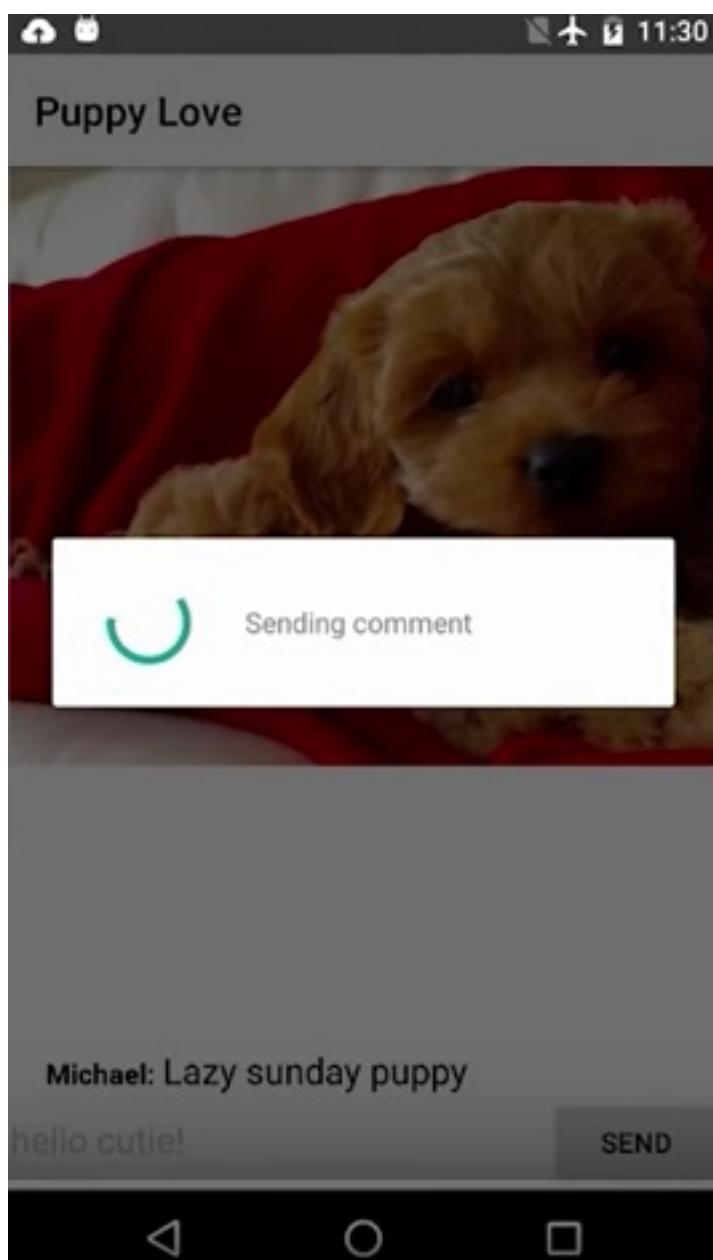
等。而在这种team里面，随着熔岩流的不断发生，人员的进进出出变动，系统内部各种设计的目的丢失，导致之后的工程师不仅不敢清理这些代码，反而只能继续增加它们的复杂性，把系统弄得越来越混乱。

如果你有一定工作经验，相信对这种状况一定不会陌生。:-D

## 为了用户体验（UX）的架构

用户不会关心你系统的架构，他们只在意用户体验。

接着我们看到了一个IM app的例子，用户发送了一个消息，然后就看到菊花转啊转，用户并不知道发生了什么。

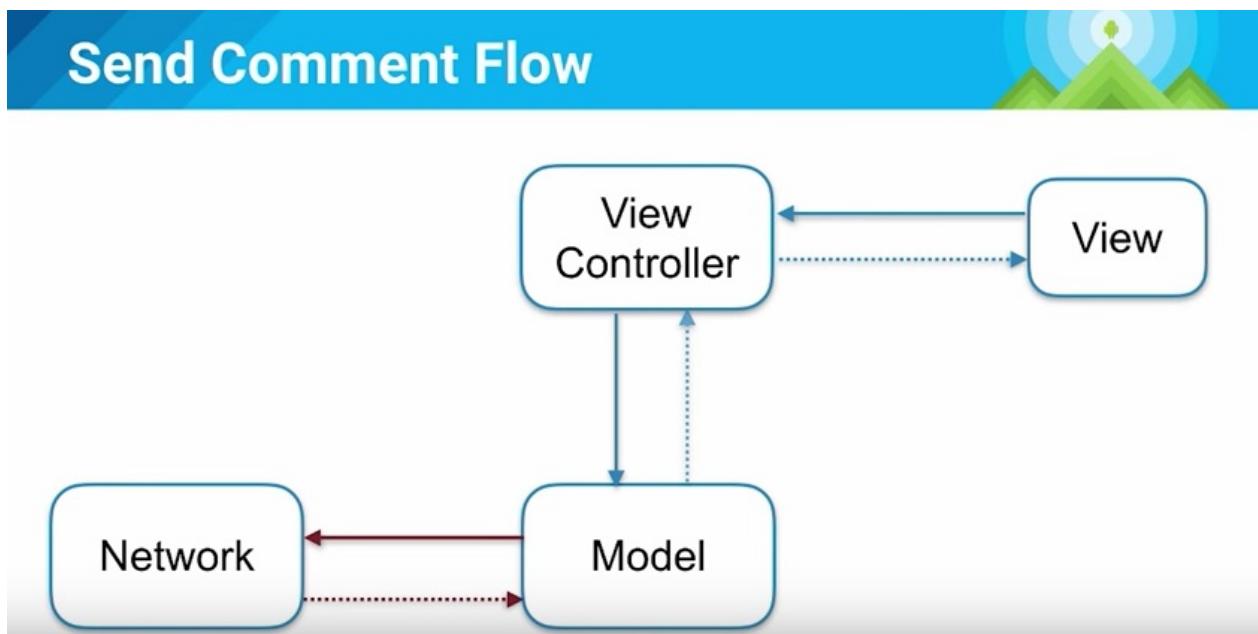


# 非持久化的提示

见下图，view发送命令给view controller，然后controller请求网络，并等待网络response再更新view。



这样就会让用户觉得体验很差劲，如果我们试着再增加一个Model层（存储），如下图，当用户点击后，view controller并不会直接发送请求，而是先更改Model层状态，Model会先把更新后的状态回调给View Controller，更新View的UI，然后才去发送网络请求，并等待response，然后再次更新UI（其实这里的Model就是Repository层了，对上面屏蔽了数据来源，UGC/FEED流的app中这种体验很重要）。



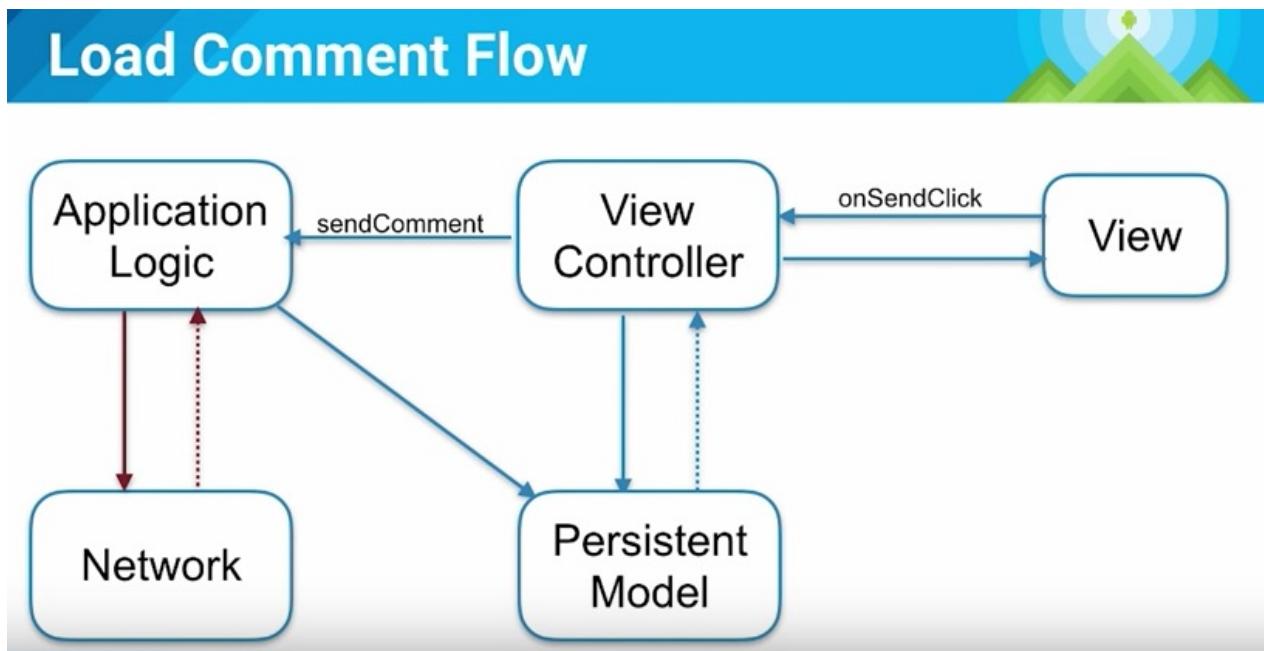
这种设计下，我们就很容易可以造一种假feed（比如真实的是黑色，假的是灰色），等到网络response回来了，再变成真feed（黑色）。

这就是非持久化的提示。

## 缓存

第二个例子，当我们从详情页退出回到列表页，然后再次点击列表项进入刚才的详情页，数据竟然不见了。2秒前我还在这里，为什么当我再次回来的时候数据就没了呢。

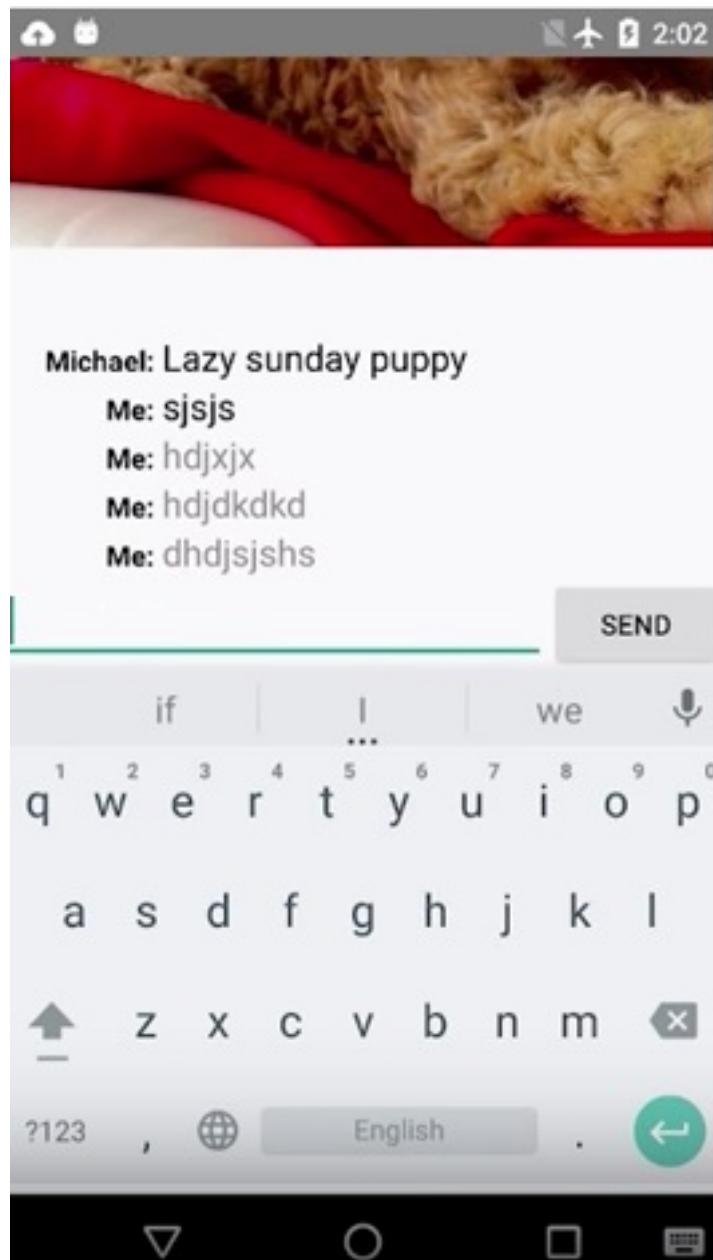
因为数据不是持久化的！它只是一个短暂的存储，数据来源是网络。



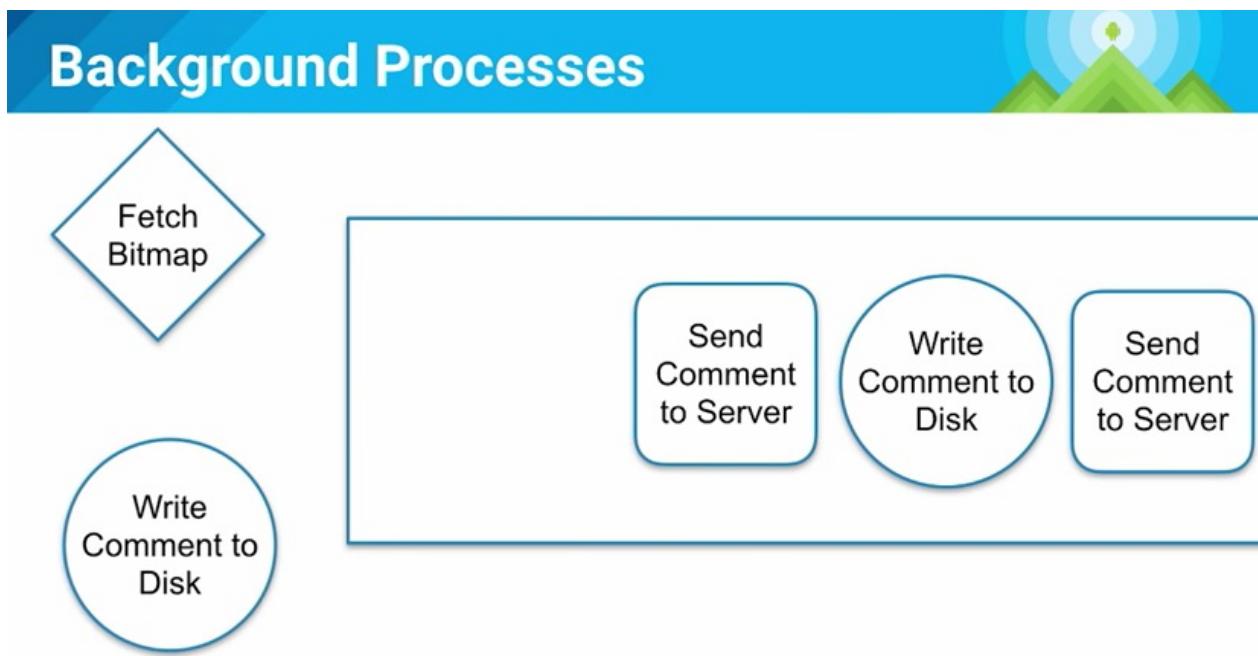
于是我们就得到了上图这样的设计，这样一来我们一进页面就能看到数据，因为数据来自本地，Application Logic会在收到View Controller请求的时候同时请求Persistent Model和Network。

## 非持久化提示队列

再来个神奇的场景，见下图，当用户输入一条信息后，UI展示了假数据，但当他继续输入后，却没有更新新的假数据到界面，这其中发生了什么呢？

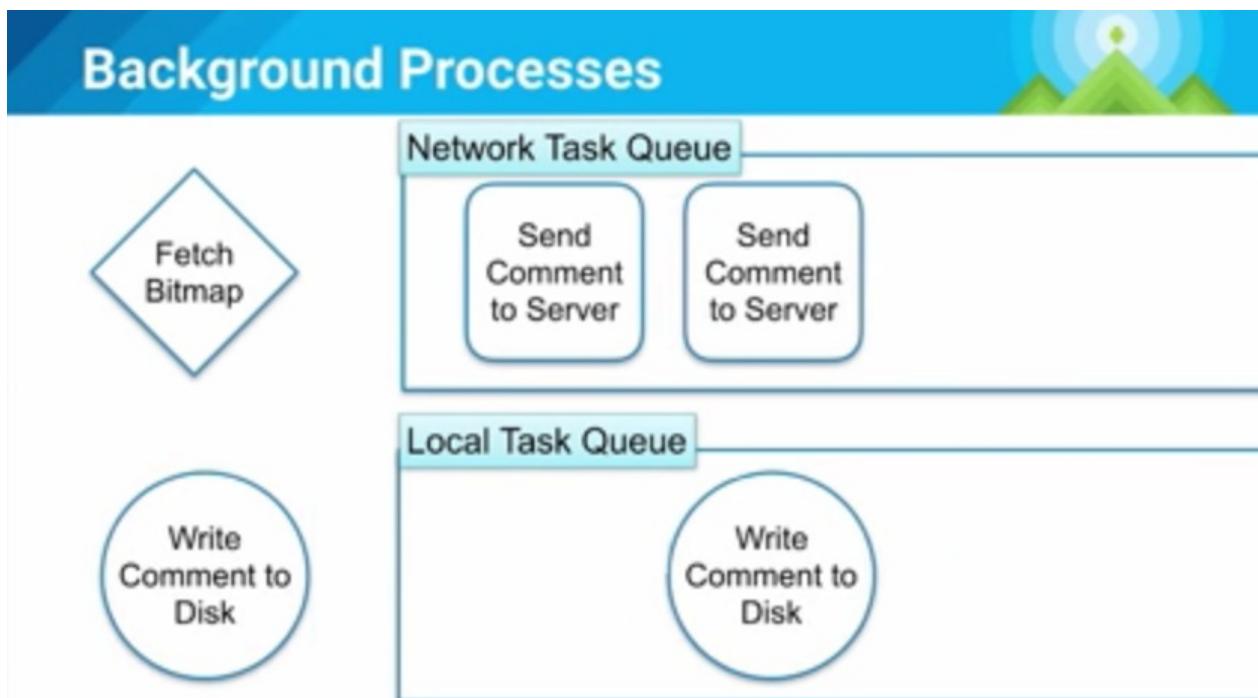


因为我们的架构是基于命令的！见下面的后台处理示意图：



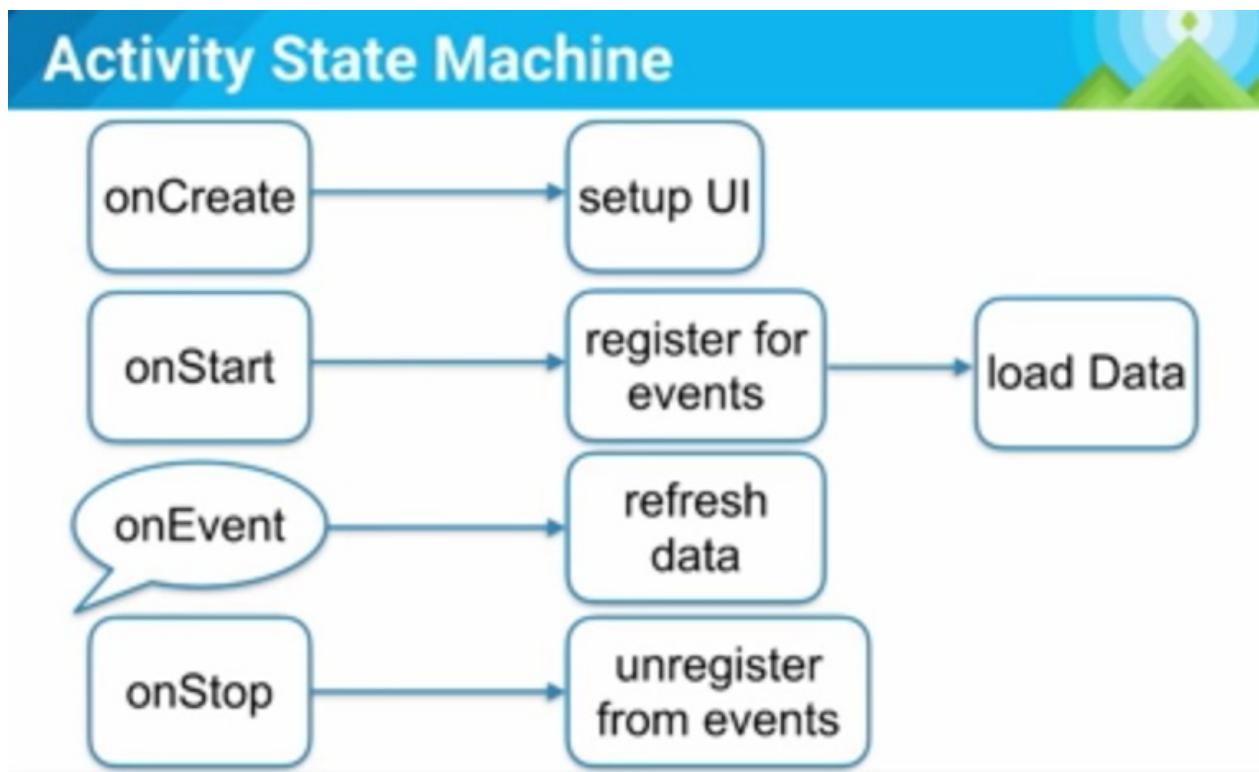
当我们处理请求的时候，我们同时只能做2件事（想象一个一个max为2的线程池），当我们在一个线程抓取bitmap，另一个线程发送数据到服务器的时候，我们无法处理第三个命令（比如从磁盘读取评论）。这是一种很糟糕的情况，调度的优先级有问题，但设计者很难料到这种情况的发生，因为并不知道每一个命令要花多久，你并不能总是估算出来。

但我们可以分开这些东西。



见上图，我们把队列分为网络和本地队列，这样就不会出现刚才的情形了。就算网络出了问题，用户还是能看到本地的假反馈。

# Activity状态机



## 小结

为了离线设计（为了更好的UX不能责怪网络）

- UI是基于model的
- App Logic负责同步model和服务器
- 这两者不互相依赖（在必要的时候使用events和callbacks通知状态改变）

# Application Logic - 应用逻辑

## 解耦

- 如果有用的话，使用依赖注入（use it if it helps）
- 了解副作用（这不是那么容易的事，尤其是当你在只在自己桌上测试的时候，当你坐在那儿的时候一切都是在最佳状态）
- 为了更好的性能，避免反射（笔者看过一些项目在命令中心大肆使用反射，这真的很让我无语）

很多依赖注入框架有一个很重的运行期组件或者编译期组件。

比如Dagger2在编译期做这些事，从而达到更好的性能。所以要仔细看看这些库，权衡它们的优劣。

## 网络

### API设计

- 为了你的用户设计后台（想到了一些后台API为了自己内部的解耦让app一个页面请求十几个接口）
- 在服务器尽量处理更多
- 传递metadata给客户端

比如有一张很大很大的图片：

```
{
 "user": {
 "name": "MarkZhai",
 "photoUrl": "https://blog.zhaiyifan.cn/..."
 }
}
```

如果只是传这些数据给客户端，那就意味着客户端需要自己处理很多逻辑，比如图片宽/高，processor等等：

```
{
 "user": {
 "name": "MarkZhai",
 "photoUrl": {
 "width" : 300,
 "height" : 500,
 "url" : "https://blog.zhaiyifan.cn/...",
 "palette" : {}
 }
 }
}
```

这样就灵活很多。

### 电池和数据

- 尽量做批量请求
- 如果有用的话，使用JobScheduler

批量请求会更省电。不想让你的app臭名昭著吧。

# Act locally, sync globally!

为了更好的用户体验。全局只做同步，更多的用户回馈性操作在本地进行。

## Activities 和 Fragments

(又想到了知乎3.0是怎么回事)

- Q: 我应该使用哪一个?
- A: Fragments是activity的封装部件

So...Fragment的英译是碎片，所以更多的不要为了Fragment而Fragment，当Activity太大，承载太多不同UI的时候，再去尝试拆分为独立的一个个Fragment。笔者在这里想到了单一职责原则。

## Fragments 和 Views

Q: 我应该使用哪一个?

- Views只是螺母和螺钉，即最基本的组成部分
- 而Activities和Fragments是有生命周期的结构
- 使用两者，保持职责清晰

## 内存

- 避免在代码热路径上分配对象（如：可以在外面申明的对象去放到循环里重复申明）
- 对象池 和 复用

一切都是为了减少GC。

来看一个例子

```
Rect getBounds() VS. void getBounds(Rect in)
```

显然第一个设计更干净更符合语言习惯，但它会额外创建一个Rect；而第二个方法则会从外面带Rect进来，从而不会做额外的内存分配。

难道我们都得用第二个么？不。我们只需要在一些很热的路径上去考虑这些问题，比如那些会运行很多很多次的布局，比如每秒需要进行60次的绘制。而像是你的那些事件处理，像是点击等，你真的不需要去为此而那样在内部扭曲系统。

# 性能

- 如果丑陋的代码可以帮助你的用户，没问题
- 用户不会去看你的代码，只会看你的UI，如果你的UI很难看，那才是真正影响他们体验的。而且编译后的代码不管怎么都会很难看（笑）
- 大部分的代码都不是性能关键

## Demo

Talk is cheap, show me the code.

## Q and A

- 36min: 自定义view的使用

笔者在这想到了facebook发过的那篇feed view优化的文章，QQ空间Android版也有类似的实践。

许多人在应该使用Custom View的时候使用了Fragment，应该更多从内聚UI块来看待这个问题。比如有一些事件需要这块UI的某一个部分来响应，那它就更像viewgroup。而如果其中的一部分需要响应应用的其他元素，像是生命周期，做了一些事像是注册事件，那它可能更适合作为一个fragment。但人们在考虑特定字段的时候总会更偏重于fragment。

- 37:33min: 怎么在app死掉的时候处理任务到磁盘的串行化和停止

在Demo里直接使用了演讲者自己写的Job Queue，开发者可以对它做简单的串行化，比如使用 Tape。

- 在这个应用里，你会使用sync adapter还是异步REST请求来和服务器沟通？

在这个demo里，我们分开了本地和网络任务。对本地任务，我们总是使用同步请求。而对网络任务，我们使用Job Queue。

- 39min的时候有一个问题，提到了服务器数据问题，这里也有一个很重要的原则：不信任服务器，保持本地model的一致性。

在Model保存前，总是去检查是否正确（Null Pattern也是一个很好的实践哦）。

# 重构



# 关于烂代码的那些事（上）

来源:[Axb的自我修养](#)

## 1.摘要

最近写了不少代码，review了不少代码，也做了不少重构，总之是对着烂代码工作了几周。为了抒发一下这几周里好几次到达崩溃边缘的情绪，我决定写一篇文章谈一谈烂代码的那些事。这里是上篇，谈一谈烂代码产生的原因和现象。

## 2.写烂代码很容易

刚入程序员这行的时候经常听到一个观点：你要把精力放在ABCD（需求文档/功能设计/架构设计/理解原理）上，写代码只是把想法翻译成编程语言而已，是一个没什么技术含量的事情。

当时的我在听到这种观点时会有一种近似于高冷的不屑：你们就是一群傻X，根本不懂代码质量的重要性，这么下去迟早有一天会踩坑，呸。

可是几个月之后，他们似乎也没怎么踩坑。而随着编程技术一直在不断发展，带来了更多的我以前认为是傻X的人加入到程序员这个行业中来。

语言越来越高级、封装越来越完善，各种技术都在帮助程序员提高生产代码的效率，依靠层层封装，程序员真的不需要了解一丁点技术细节，只要把需求里的内容逐行翻译出来就可以了。

很多程序员不知道要怎么组织代码、怎么提升运行效率、底层是基于什么原理，他们写出来的是在我心目中烂成一坨翔一样的代码。

但是那一坨翔一样代码竟然他妈的能正常工作。

即使我认为他们写的代码是坨翔，但是从不接触代码的人的视角来看（比如说你的boss），代码编译过了，测试过了，上线运行了一个月都没出问题，你还想要奢求什么？

所以，即使不情愿，也必须承认，时至今日，写代码这件事本身没有那么难了。

## 3.烂代码终究是烂代码

但是偶尔有那么几次，写烂代码的人离职了之后，事情似乎又变得不一样了。

想要修改功能时却发现程序里充斥着各种无法理解的逻辑、改完之后莫名其妙的bug一个接一个，接手这个项目的人开始漫无目的的加班，并且原本一个挺乐观开朗的人渐渐的开始喜欢问候别人祖宗了。



### 3.1. 意义不明

能力差的程序员容易写出意义不明的代码，他们不知道自己究竟在做什么。

就像这样：

```
public void save() {
 for(int i=0;i<100;i++) {
 //防止保存失败，重试100次
 document.save();
 }
}
```

```
public void save() {
 for(int i=0;i<100;i++) {
 //防止保存失败，重试100次
 document.save();
 }
}
```

对于这类程序员，我一般建议他们转行。

### 3.2. 不说人话

不说人话是新手最经常出现的问题，直接的表现就是写了一段很简单的代码，其他人却看不懂。

比如下面这段：

```
public boolean getUrl(Long id) {
 UserProfile up = us.getUser(ms.get(id).getMessage().aid);
 if (up == null) {
 return false;
 }
 if (up.type == 4 || ((up.id >> 2) & 1) == 1) {
 return false;
 }
 if(Util.getUrl(up.description)) {
 return true;
 } else {
 return false;
 }
}
```

```
public boolean getUrl(Long id) {
 UserProfile up = us.getUser(ms.get(id).getMessage().aid);
 if (up == null) {
 return false;
 }
 if (up.type == 4 || ((up.id >> 2) & 1) == 1) {
 return false;
 }
 if(Util.getUrl(up.description)) {
 return true;
 } else {
 return false;
 }
}
```

很多程序员喜欢简单的东西：简单的函数名、简单的变量名、代码里翻来覆去只用那么几个单词命名；能缩写就缩写、能省略就省略、能合并就合并。这类人写出来的代码里充斥着各种 g/s/gos/of/mss 之类的全世界没人懂的缩写，或者一长串不知道在做什么的连续调用。

还有很多程序员喜欢复杂，各种宏定义、位运算之类写的天花乱坠，生怕代码让别人一下子看懂了会显得自己水平不够。

简单的说，他们的代码是写给机器的，不是给人看的。

### 3.3.不恰当的组织

不恰当的组织是高级一些的烂代码，程序员在写过一些代码之后，有了基本的代码风格，但是对于规模大一些的工程的掌控能力不够，不知道代码应该如何解耦、分层和组织。

这种反模式的现象是经常会看到一段代码在工程里拷来拷去；某个文件里放了一大堆砌起来的代码；一个函数堆了几百上千行；或者一个简单的功能七拐八绕的调了几十个函数，在某个难以发现的猥琐的小角落里默默的调用了某些关键逻辑。



这类代码大多复杂度高，难以修改，经常一改就崩；而另一方面，创造了这些代码的人倾向于修改代码，畏惧创造代码，他们宁愿让原本复杂的代码一步步变得更复杂，也不愿意重新组织代码。当你面对一个几千行的类，问为什么不把某某逻辑提取出来的时候，他们会说：

“但是，那样就多了一个类了呀。”

### 3.4.假设和缺少抽象

相对于前面的例子，假设这种反模式出现的场景更频繁，花样更多，始作俑者也更难以自己意识到问题。比如：

```
public String loadString() {
 File file = new File("c:/config.txt");
 // read something
}
```

```
public String loadString() {
 File file = new File("c:/config.txt");
 // read something
}
```

文件路径变更的时候，会把代码改成这样：

```
public String loadString(String name) {
 File file = new File(name);
 // read something
}
```

```
public String loadString(String name) {
 File file = new File(name);
 // read something
}
```

需要加载的内容更丰富的时候，会再变成这样：

```
public String loadString(String name) {
 File file = new File(name);
 // read something
}
public Integer loadInt(String name) {
 File file = new File(name);
 // read something
}
```

```
public String loadString(String name) {
 File file = new File(name);
 // read something
}
public Integer loadInt(String name) {
 File file = new File(name);
 // read something
}
```

之后可能会再变成这样：

```
public String loadString(String name) {
 File file = new File(name);
 // read something
}
public String loadStringUtf8(String name) {
 File file = new File(name);
 // read something
}
public Integer loadInt(String name) {
 File file = new File(name);
 // read something
}
public String loadStringFromNet(String url) {
 HttpClient ...
}
public Integer loadIntFromNet(String url) {
 HttpClient ...
}
```

```
public String loadString(String name) {
 File file = new File(name);
 // read something
}
public String loadStringUtf8(String name) {
 File file = new File(name);
 // read something
}
public Integer loadInt(String name) {
 File file = new File(name);
 // read something
}
public String loadStringFromNet(String url) {
 HttpClient ...
}
public Integer loadIntFromNet(String url) {
 HttpClient ...
}
```

这类程序员往往是项目组里开发效率比较高的人，但是大量的业务开发工作导致他们不会做多余的思考，他们的口头禅是：“我每天要做XX个需求”或者“先做完需求再考虑其他的吧”。

这种反模式表现出来的后果往往是代码很难复用，面对deadline的时候，程序员迫切的想要把需求落实成代码，而这往往也会是个循环：写代码的时候来不及考虑复用，代码难复用导致之后的需求还要继续写大量的代码。

一点点积累起来的大量的代码又带来了组织和风格一致性等问题，最后形成了一个新功能基本靠拷的遗留系统。

### 3.5.还有吗

烂代码还有很多种类型，沿着功能-性能-可读-可测试-可扩展这条路线走下去，还能看到很多匪夷所思的例子。

那么什么是烂代码？个人认为，烂代码包含了几个层次：

- 如果只是一个人维护的代码，满足功能和性能要求倒也足够了。
- 如果在一个团队里工作，那就必须易于理解和测试，让其它人员有能力修改各自的代码。
- 同时，越是处于系统底层的代码，扩展性也越重要。

所以，当一个团队里的底层代码难以阅读、耦合了上层的逻辑导致难以测试、或者对使用场景做了过多的假设导致难以复用时，虽然完成了功能，它依然是坨翔一样的代码。

### 3.6.够用的代码

而相对的，如果一个工程的代码难以阅读，能不能说这个是烂代码？很难下定义，可能算不上好，但是能说它烂吗？如果这个工程自始至终只有一个人维护，那个人也维护的很好，那它似乎就成了“够用的代码”。

很多工程刚开始可能只是一个人负责的小项目，大家关心的重点只是代码能不能顺利的实现功能、按时完工。

过上一段时间，其他人参与时才发现代码写的有问题，看不懂，不敢动。需求方又开始催着上线了，怎么办？只好小心翼翼的只改逻辑而不动结构，然后在注释里写上这么实现很 ugly，以后明白内部逻辑了再重构。

再过上一段时间，有个相似的需求，想要复用里面的逻辑，这时才意识到代码里做了各种特定场景的专用逻辑，复用非常麻烦。为了赶进度只好拷代码然后改一改。问题解决了，问题也加倍了。

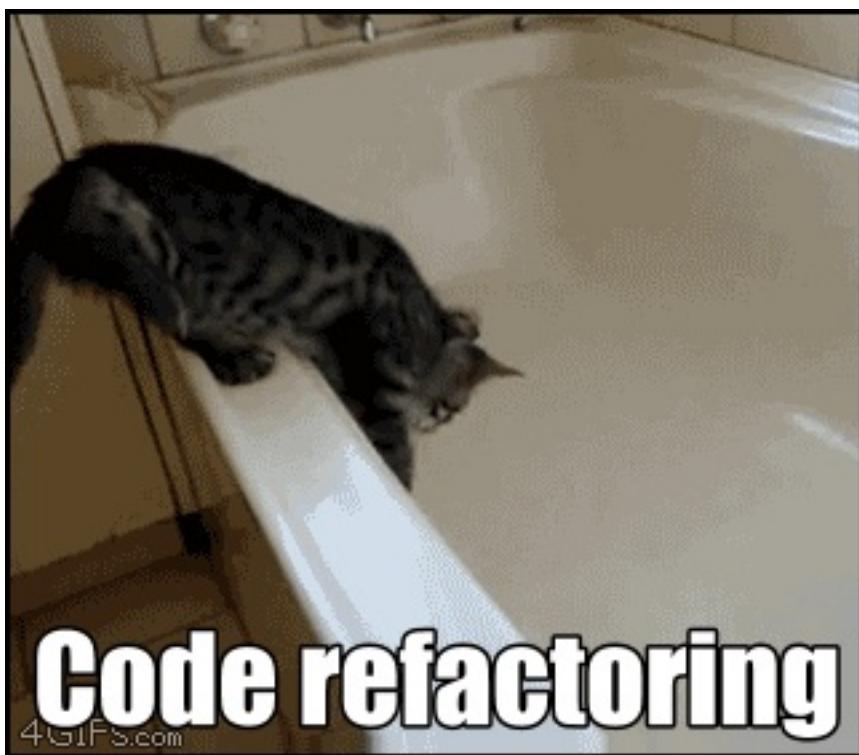
几乎所有的烂代码都是从“够用的代码”演化来的，代码没变，使用代码的场景发生了变化，原本够用的代码不符合新的场景，那么它就成了烂代码。

## 4. 重构不是万能药

程序员最喜欢跟程序员说的谎话之一就是：现在进度比较紧，等X个月之后项目进度宽松一些再去做重构。

不能否认在某些（极其有限的）场景下重构是解决问题的手段之一，但是写了不少代码之后发现，重构往往是程序开发过程中最复杂的工作。花一个月写的烂代码，要花更长的时间、更高的风险去重构。

曾经经历过几次忍无可忍的大规模重构，每一次重构之前都是找齐了组里的高手，开了无数次分析会，把组内需求全部暂停之后才敢开工，而重构过程中往往哀嚎遍野，几乎每天都会出上很多意料之外的问题，上线时也几乎必然会出现几个问题。



从技术上来说，重构复杂代码时，要做三件事：理解旧代码、分解旧代码、构建新代码。而待重构的旧代码往往难以理解；模块之间过度耦合导致牵一发而动全身，不易控制影响范围；旧代码不易测试导致无法保证新代码的正确性。

这里还有一个核心问题，重构的复杂度跟代码的复杂度不是线性相关的。比如有1000行烂代码，重构要花1个小时，那么5000行烂代码的重构可能要花2、3天。要对一个失去控制的工程做重构，往往还不如重写更有效率。

而抛开具体的重构方式，从受益上来说，重构也是一件很麻烦的事情：它很难带来直接受益，也很难量化。这里有个很有意思的现象，基本关于重构的书籍无一例外的都会有独立的章节介绍“如何向boss说明重构的必要性”。

重构之后能提升多少效率？能降低多少风险？很难答上来，烂代码本身不是一个可以简单的标准化的东西。

举个例子，一个工程的代码可读性很差，那么它会影响多少开发效率？

你可以说：之前改一个模块要3天，重构之后1天就可以了。但是怎么应对“不就是做个数据库操作吗为什么要3天”这类问题？烂代码“烂”的因素有不确定性、开发效率也因人而异，想要证明这个东西“确实”会增加两天开发时间，往往反而会变成“我看了3天才看懂这个函数是做什么的”或者“我做这么简单的修改要花3天”这种神经病才会去证明的命题。

而另一面，许多技术负责人也意识到了代码质量和重构的必要性，“那就重构嘛”，或者“如果看到问题了，那就重构”。上一个问题解决了，但实际上关于重构的代价和收益仍然是一笔糊涂账，在没有分配给你更多资源、没有明确的目标、没有具体方法的情况下，很难想象除了有代码洁癖的人还有谁会去执行这种莫名其妙的任务。

于是往往就会形成这种局面：

- 不写代码的人认为应该重构，重构很简单，无论新人还是老人都有责任做重构。
- 写代码老手认为应该迟早应该重构，重构很难，现在凑合用，这事别落在我头上。
- 写代码的新手认为不出bug就谢天谢地了，我也不知道怎么重构。

## 5.写好代码很难

与写出烂代码不同的是，想写出好代码有很多前提：

- 理解要开发的功能需求。
- 了解程序的运行原理。
- 做出合理的抽象。
- 组织复杂的逻辑。
- 对自己开发效率的正确估算。
- 持续不断的练习。

写出好代码的方法论很多，但我认为写出好代码的核心反而是听起来非常low的“持续不断的练习”。这里就不展开了，留到下篇再说。

很多程序员在写了几年代码之后并没有什么长进，代码仍然烂的让人不忍直视，原因有两个主要方面：

- 环境是很重要的因素之一，在烂代码的熏陶下很难理解什么是好代码，知道的人大部分也会选择随波逐流。
- 还有个人性格之类的说不清道不明的主观因素，写出烂代码的程序员反而都是一些很好相处的人，他们往往热爱公司团结同事平易近人工作任劳任怨—只是代码很烂而已。

而工作几年之后的人很难再说服他们去提高代码质量，你只会反复不断的听到：“那又有什么用呢？”或者“以前就是这么做的啊？”之类的说法。

那么从源头入手，提高招人时对代码的质量的要求怎么样？

前一阵面试的时候增加了白板编程、最近又增加了上机编程的题目。发现了一个现象：一个人工作了几年、做过很多项目、带过团队、发了一些文章，不一定能代表他代码写的好；反之，一个人代码写的好，其它方面的能力一般不会太差。

举个例子，最近喜欢用“写一个代码行数统计工具”作为面试的上机编程题目。很多人看到题目之后第一反映是，这道题太简单了，这不就是写写代码嘛。

从实际效果来看，这道题识别度却还不错。

首先，题目足够简单，即使没有看过《面试宝典》之类书的人也不会吃亏。而题目的扩展性很好，即使提前知道题目，配合不同的条件，可以变成不同的题目。比如要求按文件类型统计行数、或者要求提高统计效率、或者统计的同时输出某些单词出现的次数，等等。

从考察点来看，首先是基本的树的遍历算法；其次有一定代码量，可以看出程序员对代码的组织能力、对问题的抽象能力；上机编码可以很简单的看出应聘者是不是很久没写程序了；还包括对于程序易用性和性能的理解。

最重要的是，最后的结果是一个完整的程序，我可以按照日常工作的标准去评价程序员的能力，而不是从十几行的函数里意淫这个人在日常工作中大概会有什么表现。

但即使这样，也很难拍着胸脯说，这个人写的代码质量没问题。毕竟面试只是代表他有写出好代码的能力，而不是他将来会写出好代码。

## 6. 悲观的结语

说了那么多，结论其实只有两条，作为程序员：

- 不要奢望其他人会写出高质量的代码
- 不要以为自己写出来的是高质量的代码

如果你看到了这里还没有丧失希望，那么可以期待一下这篇文章的第二部分，关于如何提高代码质量的一些建议和方法。



# 关于烂代码的那些事（下）

来源:[Axb的自我修养](#)

假设你已经读过烂代码系列的前两篇：了解了什么是烂代码，什么是好代码，但是还是不可避免的接触到了烂代码（就像之前说的，几乎没有程序员可以完全避免写出烂代码！）接下来的问题便是：如何应对这些身边的烂代码。

## 1. 改善可维护性

改善代码质量是项大工程，要开始这项工程，从可维护性入手往往是一个好的开始，但也仅仅只是开始而已。

### 1.1. 重构的悖论

很多人把重构当做一种一次性运动，代码实在是烂的没法改了，或者没什么新的需求了，就召集一帮人专门拿出来一段时间做重构。这在传统企业开发中多少能生效，但是对于互联网开发来说却很难适应，原因有两个：

- 互联网开发讲究快速迭代，如果要做大型重构，往往需要暂停需求开发，这个基本上很难实现。
- 对于没有什么新需求的项目，往往意味着项目本身已经过了发展期，即使做了重构也带来不了什么收益。

这就形成了一个悖论：一方面那些变更频繁的系统更需要重构；另一方面重构又会耽误开发进度，影响变更效率。

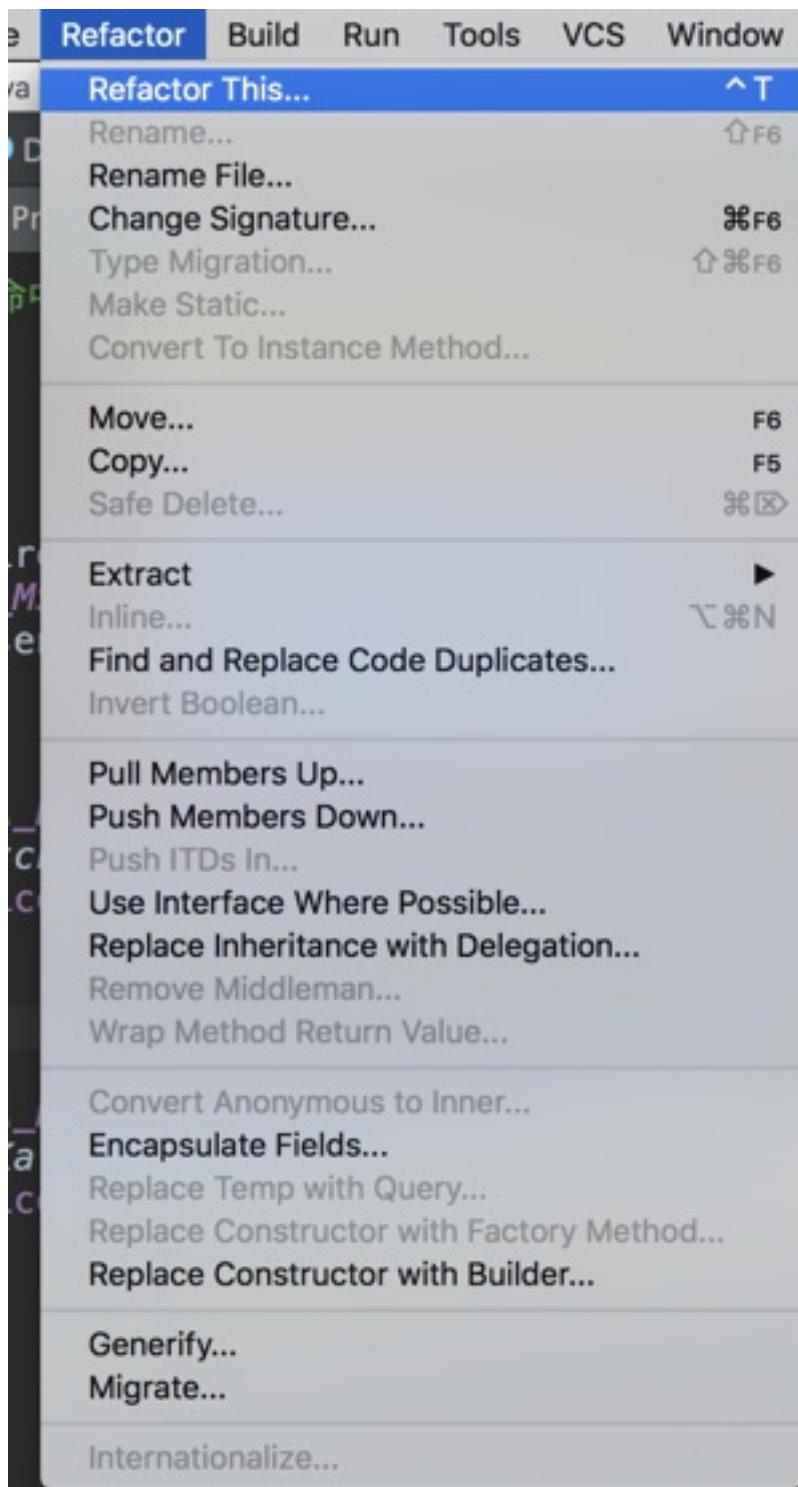
面对这种矛盾，一种方式是放弃重构，让代码质量自然下降，直到工程的生命周期结束，选择放弃或者重来。在某些场景下这种方式确实是有效的，但是我并不喜欢：比起让工程师不得不把每天的精力都浪费在毫无意义的事情上，为什么不做些更有意义的事呢？

### 1.2. 重构step by step

#### 1.2.1. 开始之前

开始改善代码的第一步是把IDE的重构快捷键设到一个顺手的键位上，这一步非常重要：决定重构成败的往往不是你的新设计有多么牛逼，而是重构本身会占用多少时间。

比如对于IDEA来说，我会把重构菜单设为快捷键：



这样在我想去重构的时候就可以随手打开菜单，而不是用鼠标慢慢去点，快捷键每次只能为重构节省几秒钟时间，但是却能明显减少工程师重构时的心理负担，后面会提到，小规模的重构应该跟敲代码一样属于日常开发的一部分。

我把重构分为三类：模块内部的重构、模块级别的重构、工程级别的重构。分为这三类并不是因为我是分类强迫症，后面会看到对重构的分类对于重构的意义。

## 1.2.2.随时进行模块内部的重构

模块内部重构的目的是把模块内部的逻辑梳理清楚，并且把一个巨大无比的函数拆分成可维护的小块代码。大部分IDE都提供了对这类重构的支持，类似于：

- 重命名变量
- 重命名函数
- 提取内部函数
- 提取内部常量
- 提取变量

这类重构的特点是修改基本集中在一个地方，对代码逻辑的修改很少并且基本可控，IDE的重构工具比较健壮，因而基本没有什么风险。

以下例子演示了如何通过IDE把一个冗长的函数做重构：

```
public class Demo {
 private UserDAO userDAO;

 void update(User u, String s) {
 if (u.getType() == 1) {
 if (s.length() > 0 && s.length() < 10) {
 List<Group> list = u.getGroups();
 for (Group g : list) {
 if ((g.getStatus() == 5 || g.getStatus() == 7)) {
 u.setName(s);
 userDAO.save(u);
 break;
 }
 }
 }
 } else {
 if (s.length() > 0 && s.length() < 10) {
 String url = "http://xxxxx/check.json?uid=" + u.getId();
 HttpClient client = new HttpClient();
 GetMethod method = new GetMethod(url);
 try {
 client.executeMethod(method);
 String s1 = method.getResponseBodyAsString();
 if ("1".equals(s1)) {
 u.setName(s);
 userDAO.save(u);
 }
 } catch (IOException e) {
 e.printStackTrace();
 }
 }
 }
 }
}
```

上图的例子中，我们基本依靠IDE就把一个冗长的函数分成了两个子函数，接下来就可以针对子函数中的一些烂代码做进一步的小规模重构，而两个函数内部的重构也可以用同样的方法。每一次小规模重构的时间都不应该超过60s，否则将会严重影响开发的效率，进而导致重构被无尽的开发需求淹没。

在这个阶段需要对现有的模块补充一些单元测试，以保证重构的正确。不过以我的经验来看，一些简单的重构，例如修改局部变量名称，或者提取变量之类的重构，即使没有测试也是基本可靠的，如果要在快速完成模块内部重构和100%的单元测试覆盖率中选一个，我可能会选择快速完成重构。

而这类重构的收益主要是提高函数级别的可读性，以及消除超大函数，为未来进一步做模块级别的拆分打好基础。

### 1.2.3.一次只做一个较模块级别的的重构

之后的重构开始牵扯到多个模块，例如：

- 删除无用代码
- 移动函数到其它类
- 提取函数到新类
- 修改函数逻辑

IDE往往对这类重构的支持有限，并且偶尔会出一些莫名其妙的问题，（例如修改类名时一不小心把配置文件里的常量字符串也给修改了）。

这类重构主要在于优化代码的设计，剥离不相关的耦合代码，在这类重构期间你需要创建大量新的类和新的单元测试，而此时的单元测试则是必须的了。

为什么要创建单元测试？

- 一方面，这类重构因为涉及到具体代码逻辑的修改，靠集成测试很难覆盖所有情况，而单元测试可以验证修改的正确性。
- 更重要的意义在于，写不出单元测试的代码往往意味着糟糕的设计：模块依赖太多或者一个函数的职责太重，想象一下，想要执行一个函数却要模拟十几个输入对象，每个对象还要模拟自己依赖的对象……如果一个模块无法被单独测试，那么从设计的角度来考虑，无疑是不合格的。

还需要啰嗦一下，这里说的单元测试只对一个模块进行测试，依赖多个模块共同完成的测试并不包含在内-例如在内存里模拟了一个数据库，并在上层代码中测试业务逻辑-这类测试并不能改善你的设计。

在这个期间还会写一些过渡用的临时逻辑，比如各种adapter、proxy或者wrapper，这些临时逻辑的生存期可能会有几个月到几年，这些看起来没什么必要的工作是为了控制重构范围，例如：

```
class Foo {
 String foo() {
 ...
 }
}
```

```
class Foo {
 String foo() {
 ...
 }
}
```

如果要把函数声明改成

```
class Foo {
 boolean foo() {
 ...
 }
}
```

```
class Foo {
 boolean foo() {
 ...
 }
}
```

那么最好通过加一个过渡模块来实现：

```
class FooAdaptor {
 private Foo foo;
 boolean foo() {
 return foo.foo().isEmpty();
 }
}
```

```
class FooAdaptor {
 private Foo foo;
 boolean foo() {
 return foo.foo().isEmpty();
 }
}
```

这样做的好处是修改函数时不需要改动所有调用方，烂代码的特征之一就是模块间的耦合比较高，往往一个函数有几十处调用，牵一发而动全身。而一旦开始全面改造，往往就会把一次看起来很简单的重构演变成几周的大工程，这种大规模重构往往是不可靠的。

每次模块级别的重构都需要精心设计，提前划分好哪些是需要修改的，哪些是需要用兼容逻辑做过渡的。但实际动手修改的时间都不应该超过一天，如果超过一天就意味着这次重构改动太多，需要控制一下修改节奏了。

### 1.2.4. 工程级别的重构不能和任何其他任务并行

不安全的重构相对而言影响范围比较大，比如：

- 修改工程结构
- 修改多个模块

我更建议这类操作不要用IDE，如果使用IDE，也只使用最简单的“移动”操作。这类重构单元测试已经完全没有作用，需要集成测试的覆盖。不过也不必紧张，如果只做“移动”的话，大部分情况下基本的冒烟测试就可以保证重构的正确性。

这类重构的目的是根据代码的层次或者类型进行拆分，切断循环依赖和结构上不合理的地方。如果不知道如何拆分，可以依照如下思路：

- 优先按部署场景进行拆分，比如一部分代码是公用的，一部分代码是自己用的，可以考虑拆成两个部分。换句话说，A服务的修改能不能影响B服务。
- 其次按照业务类型拆分，两个无关的功能可以拆分成两个部分。换句话说，A功能的修改能不能影响B功能。
- 除此之外，尽量控制自己的代码洁癖，不要把代码切成一大堆豆腐块，会给日后的维护工作带来很多不必要的成本。
- 方案可以提前review几次，多参考一线工程师的意见，避免实际动手时才冒出新的问题。

而这类重构绝对不能跟正常的需求开发并行执行：代码冲突几乎无法避免，并且会让所有人崩溃。我的做法一般是在这类重构前先演练一次：把模块按大致的想法拖来拖去，通过编译器找到依赖问题，在日常上线中把容易处理的依赖问题解决掉；然后集中团队里的精英，通知所有人暂停开发，花最多2、3天时间把所有问题集中突击掉，新的需求都在新代码的基础上进行开发。

如果历史包袱实在太重，可以把这类重构也拆成几次做：先大体拆分成几块，再分别拆分。无论如何，这类重构务必控制好变更范围，一次严重的合并冲突有可能让团队中的所有人几个周缓不过劲来。

## 1.3. 重构的周期

典型的重构周期类似下面的过程：

- 1、在正常需求开发的同时进行模块内部的重构，同时理解工程原有代码。

- 2、在需求间隙进行模块级别的重构，把大模块拆分为多个小模块，增加脚手架类，补充单元测试，等等。
- 3、（如果有必要，比如工程过于巨大导致经常出现相互影响问题）进行一次工程级别的拆分，期间需要暂停所有开发工作，并且这次重构除了移动模块和移动模块带来的修改之外不做任何其他变更。
- 4、重复1、2步骤

### 1.3.1.一些重构的tips

- 只重构经常修改的部分，如果代码一两年都没有修改过，那么说明改动的收益很小，重构能改善的只是可维护性，重构不维护的代码不会带来收益。
- 抑制住自己想要多改一点的冲动，一次失败的重构对代码质量改进的影响可能是毁灭性的。
- 重构需要不断的练习，相比于写代码来说，重构或许更难一些。
- 重构可能需要很长时间，有可能甚至会达到几年的程度（我之前用断断续续两年多的时间重构了一个项目），主要取决于团队对于风险的容忍程度。
- 删除无用代码是提高代码可维护性最有效的方式，切记，切记。
- 单元测试是重构的基础，如果对单元测试的概念还不是很清晰，可以参考使用[Spock框架进行单元测试](#)

## 2.改善性能与健壮性

### 2.1.改善性能的80%

性能这个话题越来越多的被人提起，随便收到一份简历不写上点什么熟悉高并发、做过性能优化之类的似乎都不好意思跟人打招呼。

说个真事，几年前在我做某公司的ERP项目，里面有个功能是生成一个报表。而使用我们系统的公司里有一个人，他每天要在下班前点一下报表，导出到excel，再发一封邮件出去。

问题是，那个报表每次都要2，3分钟才能生成。

我当时正年轻气盛，看到有个两分钟才能生成的报表一下就来了兴趣，翻出了那段不知道谁写的代码，发现里面用了3层循环，每次都会去数据库查一次数据，再把一堆数据拼起来，一股脑塞进一个tableview里。

面对这种代码，我还能做什么呢？

- 我立刻把那个三层循环干掉了，通过一个存储过程直接输出数据。

- sql数据计算的逻辑也被我精简了，一些没必要做的外联操作被我干掉了。
- 我还发现很多ctrl+v生成的无用的控件（那时还是用的delphi），那些控件密密麻麻的贴在显示界面上，只是被前面的大table挡住了，我当然也把这些玩意都删掉了；
- 打开界面的时候还做了一些杂七杂八的工作（比如去数据库里更新点击数之类的），我把这些放到了异步任务里。
- 后面我又觉得没必要每次打开界面都要加载所有数据（那个tableview有几千行，几百列！），于是我hack了默认的tableview，每次打开的时候先计算当前实际显示了多少内容，把参数发给存储过程，初始化只加载这些数据，剩下的再通过线程异步加载。

做了这些之后，界面只需要不到1s就能展示出来了，不过我要说的不是这个。

后来我去客户公司给那个操作员演示新的模块的时候，点一下，刷，数据出来了。那个人很惊恐的看着我，然后问我，是不是数据不准了。

再后来，我又加了一个功能，那个模块每次打开之后都会显示一个进度条，上面的标题是“正在校验数据……”，进度条走完大概要1分钟左右，我跟那人说校验数据计算量很大，会比较慢。当然，实际上那60秒里程序毛事都没做，只是在一点点的更新那个进度条（我还做了个彩蛋，在读进度的时候按上上下下左右左右BABA的话就可以加速10倍读条…）。客户很开心，说感觉数据准确多了，当然，他没发现彩蛋。

我写了这么多，是想让你明白一个事实：大部分程序对性能并不敏感。而少数对性能敏感的程序里，一大半可以靠调节参数解决性能问题；最后那一小撮需要修改代码优化性能的程序里，性价比高的工作又是少数。

什么是性价比？回到刚才的例子里，我做了那么多事，每件事的收益是多少？

- 把三层循环sql改成了存储过程，大概让我花了一天时间，让加载时间从3分钟变成了2秒，模块加载变成了“唰”的一下。后面的一堆事情大概花了我一周多时间，尤其是hack那个tableview，让我连周末都搭进去了。而所有的优化加起来，大概优化了1秒左右，这个数据是通过日志查到的：即使是我自己，打开模块也没感觉出有什么明显区别。
- 我现在遇到的很多面试者说程序优化时总是喜欢说一些玄乎的东西：调用栈、尾递归、内联函数、GC调优……但是当我问他们：把一个普通函数改成内联函数是把原来运行速度是多少的程序优化成多少了，却很少有人答出来；或者是扭扭捏捏的说，应该很多，因为这个函数会被调用很多遍。我再问会被调用多少遍，每遍是多长时间，就答不上来了。

所以关于性能优化，我有两个观点：

- 优化主要部分，把一次网络IO改为内存计算带来的收益远大于捯饬编译器优化之类的东西。这部分内容可以参考Numbers you should know；或者自己写一个for循环，做

一个无限`i++`的程序，看看一秒钟`i`能累加多少次，感受一下cpu和内存的性能。

- 性能优化之后要有量化数据，明确的说出优化后哪个指标提升了多少。如果有人因为“提升性能”之类的原因写了一堆让人无法理解的代码，请务必让他给出性能数据：这很有可能是一坨没有什么收益的烂代码。

至于具体的优化措施，无外乎几类：

- 让计算靠近存储
- 优化算法的时间复杂度
- 减少无用的操作
- 并行计算

关于性能优化的话题还可以讲很多内容，不过对于这篇文章来说有点跑题，这里就不再详细展开了。

## 2.2.决定健壮性的20%

前一阵听一个技术分享，说是他们在编程的时候要考虑太阳黑子对cpu计算的影响，或者是农民伯伯的猪把基站拱塌了之类的特殊场景。如果要优化程序的健壮性，那么有时候就不得不去考虑这些极端情况对程序的影响。

大部分的人应该不用考虑太阳黑子之类的高深的问题，但是我们需要考虑一些常见的特殊场景，大部分程序员的代码对于一些特殊场景都会有或多或少考虑不周全的地方，例如：

- 用户输入
- 并发
- 网络IO

常规的方法确实能够发现代码中的一些bug，但是到了复杂的生产环境中时，总会出现一些完全没有想到的问题。虽然我也想了很久，遗憾的是，对于健壮性来说，我并没有找到什么立竿见影的解决方案，因此，我只能谨慎的提出一点点建议：

- 更多的测试测试的目的是保证代码质量，但测试并不等于质量，你做覆盖80%场景的测试，在20%测试不到的地方还是有可能出问题。关于测试又是一个巨大的话题，这里就先不展开了。
- 谨慎发明轮子。例如UI库、并发库、IO client等等，在能满足要求的情况下尽量采用成熟的解决方案，所谓的“成熟”也就意味着经历了更多实际使用环境下的测试，大部分情况下这种测试的效果是更好的。

## 3.改善生存环境

看了上面的那么多东西之后，你可以想一下这么个场景：

在你做了很多事情之后，代码质量似乎有了质的飞跃。正当你以为终于可以摆脱天天踩屎的日子了的时候，某次不小心瞥见某个类又长到几千行了。

你愤怒的翻看提交日志，想找出罪魁祸首是谁，结果却发现每天都会有人往文件里提交那么十几二十行代码，每次的改动看起来都没什么问题，但是日积月累，一年年过去，当初花了九牛二虎之力重构的工程又成了一坨烂代码……

任何一个对代码有追求的程序员都有可能遇到这种问题，技术在更新，需求在变化，公司人员会流动，而代码质量总会在不经意间偷偷的变差……

想要改善代码质量，最后往往就会变成改善生存环境。

### 3.1.统一环境

团队需要一套统一的编码规范、统一的语言版本、统一的编辑器配置、统一的文件编码，如果有条件最好能使用统一的操作系统，这能避免很多无意义的工作。

就好像最近渣浪给开发全部换成了统一的macbook，一夜之间以前的很多问题都变得不是问题了：字符集、换行符、IDE之类的问题只要一个配置文件就解决了，不再有各种稀奇古怪的代码冲突或者不兼容的问题，也不会有人突然提交上来一些编码格式稀奇古怪的文件了。

### 3.2.代码仓库

代码仓库基本上已经是每个公司的标配，而现在的代码仓库除了储存代码，还可以承担一些团队沟通、代码review甚至工作流程方面的任务，如今这类开源的系统很多，像gitlab(github)、Phabricator这类优秀的工具都能让代码管理变得简单很多。我这里无意讨论svn、git、hg还是什么其它的代码管理工具更好，就算最近火热的git在复杂性和集中化管理上也有一些问题，其实我是比较期待能有替代git的工具产生的，扯远了。

代码仓库的意义在于让更多的人能够获得和修改代码，从而提高代码的生命周期，而代码本身的生命周期足够持久，对代码质量做的优化才有意义。

### 3.3.持续反馈

大多数烂代码就像癌症一样，当烂代码已经产生了可以感觉到的影响时，基本已经是晚期，很难治好了。

因此提前发现代码变烂的趋势很重要，这类工作可以依赖类似于checkstyle、findbug之类的静态检查工具，及时发现代码质量下滑的趋势，例如：

- 每天都在产生大量的新代码
- 测试覆盖率下降
- 静态检查的问题增多

有了代码仓库之后，就可以把这种工具与仓库的触发机制结合起来，每次提交的时候做覆盖率、静态代码检查等工作，jenkins+sonarqube或者类似的工具就可以完成基本的流程：伴随着代码提交进行各种静态检查、运行各种测试、生成报告并供人参考。

在实践中会发现，关于持续反馈的五花八门的工具很多，但是真正有用的往往只有那么一两个，大部分人并不会去在每次提交代码之后再打开一个网页点击“生成报告”，或者去登陆什么系统看一下测试的覆盖率是不是变低了，因此一个一站式的系统大多数情况下会表现的更好。与其追求更多的功能，不如把有限的几个功能整合起来，例如我们把代码管理、回归测试、代码检查、和code review集成起来，就是这个样子：

The screenshot shows a Jenkins interface for a merge request. At the top, it says "im / attachment · Merge Requests". A blue button indicates the merge was "Merged". Below that, the merge request details are shown: "Merge Request #34 · created by 秦迪 · 27 minutes ago · 22 minutes ago". The title of the merge request is "add sonar comment". The description states "Request to merge add\_sonar\_comment into master". A green checkmark indicates "CI build success for c554d2d2. View build details". The merge message says "Merged by 秦迪 22 minutes ago" and "The changes were merged into master. The source branch has been removed." Below this, there are tabs for "Discussion 1", "Commits 1", and "Changes 1". The "Discussion" tab is active, showing 1 participant. The participant, "Bad Smell", started a discussion on commit c554d2d2. The message from "Bad Smell" (@badsmell) says: "SonarQube analysis reported 5 issues: • 5 major". It also notes: "Watch the comments in this conversation to review them. Note: the following issues could not be reported as comments because they are located on lines that are not displayed in this commit:". A list of 5 major issues is provided, each with a "REPLY" link. To the right of the discussion, there are fields for "Assignee: none", "Select assignee", "Milestone: none", "Select milestone", and a "Subscription" section with an "UNSUBSCRIBE" button. A note at the bottom right says: "You're receiving notifications because you're subscribed to this thread." At the bottom right of the interface, there is a watermark: "@往深里Jaxb weibo.com/2baxb".

### 3.4.质量文化

不同的团队文化会对技术产生微妙的影响，关于代码质量没有什么共同的文化，每个公司都有自己的一套观点，并且似乎都能说得通。

对于我自己来说，关于代码质量是这样的观点：

- 烂代码无法避免
- 烂代码无法接受
- 烂代码可以改进
- 好的代码能让工作更开心一些

如何让大多数人认同关于代码质量的观点实际上是一些难度的，大部分技术人员对代码质量的观点是既不赞成、也不反对的中立态度，而代码质量就像是熵值一样，放着不管总是会像更加混乱的方向演进，并且写烂代码的成本实在是太低了，以至于一个实习生花上一个礼拜就可以毁了你花了半年精心设计的工程。

所以在提高代码质量时，务必想办法拉上团队里的其他人一起。虽然“引导团队提高代码质量”这件事情一开始会很辛苦，但是一旦有了一些支持者，并且有了可以参考的模板之后，剩下的工作就简单多了。

这里推荐《布道之道：引领团队拥抱技术创新》这本书，里面大部分的观点对于代码质量也是可以借鉴的。仅靠喊口号很难让其他人写出高质量的代码，让团队中的其他体会到高质量代码的收益，比喊口号更有说服力。

## 4.最后再说两句

优化代码质量是一件很有意思，也很有挑战性的事情，而挑战不光来自于代码原本有多烂，要改进的也并不只是代码本身，还有工具、习惯、练习、开发流程、甚至团队文化这些方方面面的事情。

写这一系列文章前前后后花了半年多时间，一直处在写一点删一点的状态：我自身关于代码质量的想法和实践也在经历着不断变化。我更希望能写出一些能够实践落地的东西，而不是喊口号，忽悠忽悠“敏捷开发”、“测试驱动”之类的几个名词就结束了。

但是在写文章的过程中就会慢慢发现，很多问题的改进方法确实不是一两篇文章可以说明白的，问题之间往往又相互关联，全都展开说甚至超出了一本书的信息量，所以这篇文章也只能删去了很多内容。

我参与过很多代码质量很好的项目，也参与过一些质量很烂的项目，改进了很多项目，也放弃了一些项目，从最初的单打独斗自己改代码，到后来带领团队优化工作流程，经历了很多。无论如何，关于烂代码，我决定引用一下《布道之道》这本书里的一句话：

“‘更好’，其实不是一个目的地，而是一个方向...在当前的位置和将来的目标之间，可能有很多相当不错的地方。你只需关注离开现在的位置，而不要关心去向何方。”

# 关于烂代码的那些事（中）

来源:[Axb的自我修养](#)

## 1.摘要

这是烂代码系列的第二篇，在文章中我会跟大家讨论一下如何尽可能高效和客观的评价代码的优劣。

在发布了关于烂代码的那些事（上）之后，发现这篇文章竟然意外的很受欢迎，很多人也描(tu)述(cao)了各自代码中这样或者那样的问题。

最近部门在组织bootcamp，正好我负责培训代码质量部分，在培训课程中让大家花了不少时间去讨论、改进、完善自己的代码。虽然刚毕业的同学对于代码质量都很用心，但最终呈现出来的质量仍然没能达到“十分优秀”的程度。究其原因，主要是不了解好的代码“应该”是什么样的。

## 2.什么是好代码

写代码的第一步是理解什么是好代码。在准备bootcamp的课程的时候，我就为这个问题犯了难，我尝试着用一些精确的定义区分出“优等品”、“良品”、“不良品”；但是在总结的过程中，关于“什么是好代码”的描述却大多没有可操作性

### 2.1.好代码的定义

随便从网上搜索了一下“优雅的代码”，找到了下面这样的定义：

**Bjarne Stroustrup, C++之父:**

- 逻辑应该是清晰的，bug难以隐藏；
- 依赖最少，易于维护；
- 错误处理完全根据一个明确的策略；
- 性能接近最佳化，避免代码混乱和无原则的优化；
- 整洁的代码只做一件事。

**Grady Booch, 《面向对象分析与设计》作者:**

- 整洁的代码是简单、直接的；

- 整洁的代码，读起来像是一篇写得很好的散文；
- 整洁的代码永远不会掩盖设计者的意图，而是具有少量的抽象和清晰的控制行。

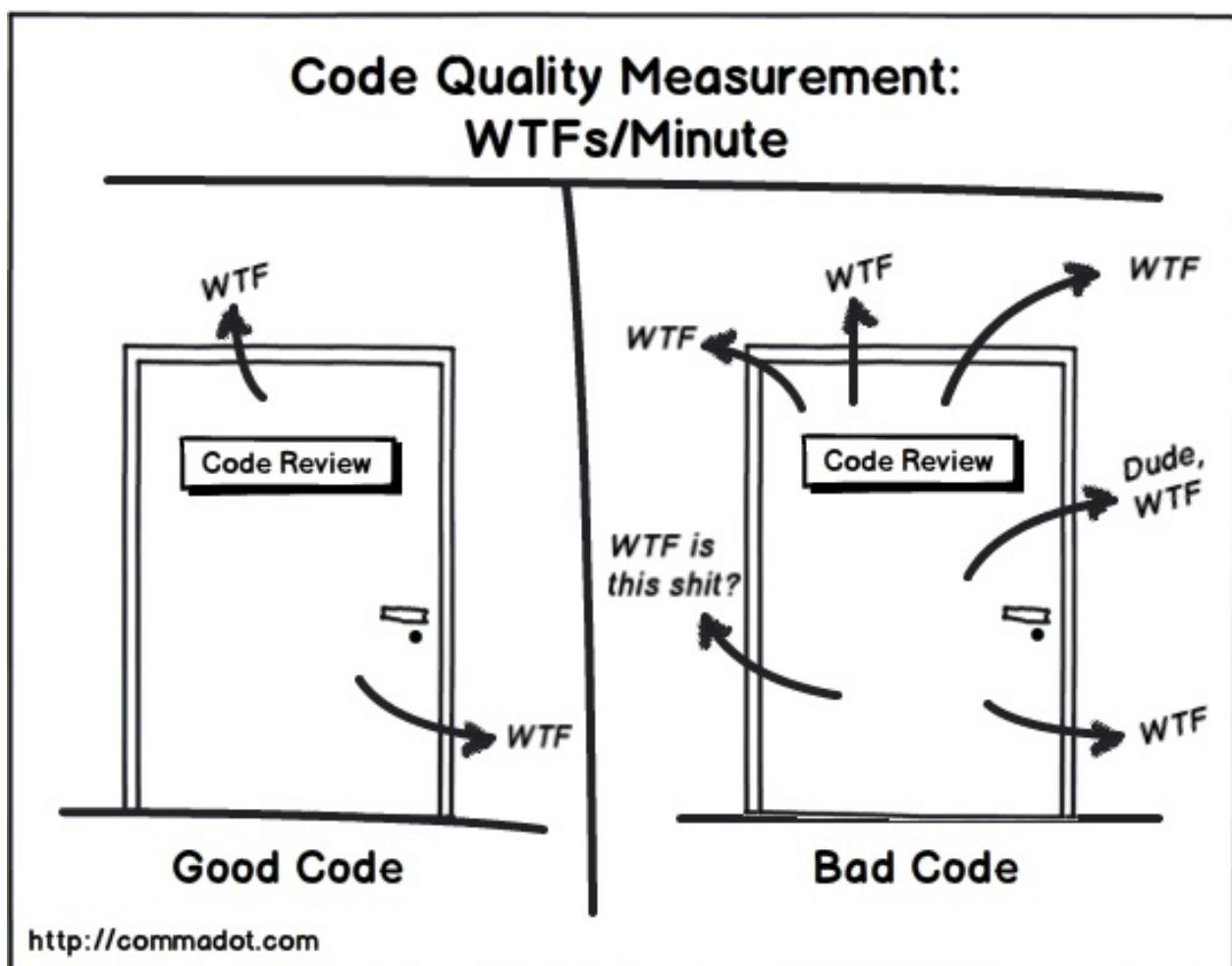
Michael Feathers，《修改代码的艺术》作者：

- 整洁的代码看起来总是像很在乎代码质量的人写的；
- 没有明显的需要改善的地方；
- 代码的作者似乎考虑到了所有的事情。

看起来似乎说的都很有道理，可是实际评判的时候却难以参考，尤其是对于新人来说，如何理解“简单的、直接的代码”或者“没有明显的需要改善的地方”？

而实践过程中，很多同学也确实面对这种问题：对自己的代码总是处在一种心里不踏实的状态，或者是自己觉得很好了，但是却被其他人认为很烂，甚至有几次我和新同学因为代码质量的标准一连讨论好几天，却谁也说服不了谁：我们都坚持自己对于好代码的标准才是正确的。

在经历了无数次code review之后，我觉得这张图似乎总结的更好一些：



代码质量的评价标准某种意义上有点类似于文学作品，比如对小说的质量的评价主要来自于它的读者，由个体主观评价形成一个相对客观的评价。并不是依靠字数，或者作者使用了哪些修辞手法之类的看似完全客观但实际没有什么意义的评价手段。

但代码和小说还有些不一样，它实际存在两个读者：计算机和程序员。就像上篇文章里说的，即使所有程序员都看不懂这段代码，它也是可以被计算机理解并运行的。

所以对于代码质量的定义我需要于从两个维度分析：主观的，被人类理解的部分；还有客观的，在计算机里运行的状况。

既然存在主观部分，那么就会存在个体差异，对于同一段代码评价会因为看代码的人的水平不同而得出不一样的结论，这也是大多数新人面对的问题：他们没有一个可以执行的评价标准，所以写出来的代码质量也很难提高。

有些介绍代码质量的文章讲述的都是倾向或者原则，虽然说的很对，但是实际指导作用不大。所以在这篇文章里我希望尽可能把评价代码的标准用（我自认为）与实际水平无关的评价方式表示出来。

## 2.2. 可读的代码

在权衡很久之后，我决定把可读性的优先级排在前面：一个程序员更希望接手一个有bug但是看的懂的工程，还是一个没bug但是看不懂的工程？如果是后者，可以直接关掉这个网页，去做些对你来说更有意义的事情。

### 2.2.1. 逐字翻译

在很多跟代码质量有关的书里都强调了一个观点：程序首先是给人看的，其次才是能被机器执行，我也比较认同这个观点。在评价一段代码能不能让人看懂的时候，我习惯让作者把这段代码逐字翻译成中文，试着组成句子，之后把中文句子读给另一个人没有看过这段代码的人听，如果另一个人能听懂，那么这段代码的可读性基本就合格了。

用这种判断方式的原因很简单：其他人在理解一段代码的时候就是这么做的。阅读代码的人会一个词一个词的阅读，推断这句话的意思，如果仅靠句子无法理解，那么就需要联系上下文理解这句代码，如果简单的联系上下文也理解不了，可能还要掌握更多其它部分的细节来帮助推断。大部分情况下，理解一句代码**在做什么**需要联系的上下文越多，意味着代码的质量越差。

逐字翻译的好处是能让作者能轻易的发现那些只有自己知道的、没有体现在代码里的假设和可读性陷阱。无法从字面意义上翻译出原本意思的代码大多都是烂代码，比如“ms代表messageService”，或者“ms.proc()是发消息”，或者“tmp代表当前的文件”。

## 2.2.2. 遵循约定

约定包括代码和文档如何组织，注释如何编写，编码风格的约定等等，这对于代码未来的维护很重要。对于遵循何种约定没有一个强制的标准，不过我更倾向于遵守更多人的约定。

与开源项目保持风格一致一般来说比较靠谱，其次也可以遵守公司内部的编码风格。但是如果公司内部的编码风格和当前开源项目的风格冲突比较严重，往往代表着这个公司的技术倾向于封闭，或者已经有些跟不上节奏了。

但是无论如何，遵守一个约定总比自己创造出一些规则要好很多，这降低了理解、沟通和维护的成本。如果一个项目自己创造出了一些奇怪的规则，可能意味着作者看过的代码不够多。

一个工程是否遵循了约定往往需要代码阅读者有一定经验，或者需要借助checkstyle这样的静态检查工具。如果感觉无处下手，那么大部分情况下跟着google做应该不会有什么大问题：可以参考[google code style](#)，其中一部分有对应的[中文版](#)。

另外，没有必要纠结于遵循了约定到底有什么收益，就好像走路是靠左好还是靠右好一样，即使得出了结论也没有什么意义，大部分约定只要遵守就可以了。

## 2.2.3. 文档和注释

文档和注释是程序很重要的部分，他们是理解一个工程或项目的途径之一。两者在某些场景下定位会有些重合或者交叉（比如javadoc实际可以算是文档）。

对于文档的标准很简单，能找到、能读懂就可以了，一般来说我比较关心这几类文档：

- 1、对于项目的介绍，包括项目功能、作者、目录结构等，读者应该能在3分钟内大致理解这个工程是做什么的。
- 2、针对新人的QuickStart，读者按照文档说明应该能在1小时内完成代码构建和简单使用。
- 3、针对使用者的详细说明文档，比如接口定义、参数含义、设计等，读者能通过文档了解这些功能（或接口）的使用方法。

有一部分注释实际是文档，比如之前提到的javadoc。这样能把源码和注释放在一起，对于读者更清晰，也能简化不少文档的维护的工作。

还有一类注释并不作为文档的一部分，比如函数内部的注释，这类注释的职责是说明一些代码本身无法表达的作者在编码时的思考，比如“为什么这里没有做XXX”，或者“这里要注意XXX问题”。

一般来说我首先会关心注释的数量：函数内部注释的数量应该不会有很多，也不会完全没有，个人经验值是滚动几屏幕看到一两处左右比较正常。过多的话可能意味着代码本身的可读性有问题，而如果一点都没有可能意味着有些隐藏的逻辑没有说明，需要考虑适当的增加一点注释了。

其次也需要考虑注释的质量：在代码可读性合格的基础上，注释应该提供比代码更多的信息。文档和注释并不是越多越好，它们可能会导致维护成本增加。关于这部分的讨论可以参考简洁部分的内容。

## 2.2.4. 推荐阅读

《代码整洁之道》

## 2.3. 可发布的代码

新人的代码有一个比较典型的特征，由于缺少维护项目的经验，写的代码总会有很多考虑不到的地方。比如说测试的时候似乎没什么异常，项目发布之后才发现有很多意料之外的状况；而出了问题之后不知道从哪下手排查，或者仅能让系统处于一个并不稳定的状态，依靠一些巧合勉强运行。

### 2.3.1. 处理异常

新手程序员普遍没有处理异常的意识，但代码的实际运行环境中充满了异常：服务器会死机，网络会超时，用户会胡乱操作，不怀好意的人会恶意攻击你的系统。

我对一段代码异常处理能力的第一印象来自于单元测试的覆盖率。大部分异常难以在开发或者测试环境里复现，即使有专业的测试团队也很难在集成测试环境中模拟所有的异常情况。

而单元测试可以比较简单的模拟各种异常情况，如果一个模块的单元测试覆盖率连50%都不到，很难想象这些代码考虑了异常情况下的处理，即使考虑了，这些异常处理的分支都没有被验证过，怎么指望实际运行环境中出现问题时表现良好呢？

### 2.3.2. 处理并发

我收到的很多简历里都写着：精通并发编程/熟悉多线程机制，诸如此类，跟他们聊的时候也说的头头是道，什么锁啊互斥啊线程池啊同步啊信号量啊一堆一堆的名词滔滔不绝。而给应聘者一个实际场景，让应聘者写一段很简单的并发编程的小程序，能写好的却不多。

实际上并发编程也确实很难，如果说写好同步代码的难度为5，那么并发编程的难度可以达到100。这并不是危言耸听，很多看似稳定的程序，在面对并发场景的时候仍然可能出现问题：比如最近我们就碰到了一个linux kernel在调用某个系统函数时由于同步问题而出现crash的情况。

而是否高质量的实现并发编程的关键并不是是否应用了某种同步策略，而是看代码中是否保护了共享资源：

- 局部变量之外的内存访问都有并发风险（比如访问对象的属性，访问静态变量等）
- 访问共享资源也会有并发风险（比如缓存、数据库等）。
- 被调用方如果不是声明为线程安全的，那么很有可能存在并发问题（比如java的 hashmap）。
- 所有依赖时序的操作，即使每一步操作都是线程安全的，还是存在并发问题（比如先删除一条记录，然后把记录数减一）。

前三种情况能够比较简单的通过代码本身分辨出来，只要简单培养一下自己对于共享资源调用的敏感度就可以了。

但是对于最后一种情况，往往很难简单的通过看代码的方式看出来，甚至出现并发问题的两处调用并不是在同一个程序里（比如两个系统同时读写一个数据库，或者并发的调用了一个程序的不同模块等）。但是，只要是代码里出现了不加锁的，访问共享资源的“先做A，再做B”之类的逻辑，可能就需要提高警惕了。

### 2.3.3. 优化性能

性能是评价程序员能力的一个重要指标，很多程序员也对程序的性能津津乐道。但程序的性能很难直接通过代码看出来，往往要借助于一些性能测试工具，或者在实际环境中执行才能有结果。

如果仅从代码的角度考虑，有两个评价执行效率的办法：

- 算法的时间复杂度，时间复杂度高的程序运行效率必然会低。
- 单步操作耗时，单步耗时高的操作尽量少做，比如访问数据库，访问io等。

而实际工作中，也会见到一些程序员过于热衷优化效率，相对的会带来程序易读性的降低、复杂度提高、或者增加工期等等。对于这类情况，简单的办法是让作者说出这段程序的瓶颈在哪里，为什么会有这个瓶颈，以及优化带来的收益。

当然，无论是优化不足还是优化过度，判断性能指标最好的办法是用数据说话，而不是单纯看代码，性能测试这部分内容有些超出这篇文章的范围，就不详细展开了。

### 2.3.4. 日志

日志代表了程序在出现问题时排查的难易程度，经(jing)验(chang)丰(cai)富(keng)的程序员大概都会遇到过这个场景：排查问题时就少一句日志，查不到某个变量的值不知道是什么，导致死活分析不出来问题到底出在哪。

对于日志的评价标准有三个：

- 日志是否足够，所有异常、外部调用都需要有日志，而一条调用链路上的入口、出口和路径关键点上也需要有日志。
- 日志的表达是否清晰，包括是否能读懂，风格是否统一等。这个的评价标准跟代码的可读性一样，不重复了。
- 日志是否包含了足够的信息，这里包括了调用的上下文、外部的返回值，用于查询的关键字等，便于分析信息。

对于线上系统来说，一般可以通过调整日志级别来控制日志的数量，所以打印日志的代码只要不对阅读造成障碍，基本上都是可以接受的。

### 2.3.5. 扩展阅读

- 《Release It!: Design and Deploy Production-Ready Software》（不要看中文版，翻译的实在是太烂了）
- [Numbers Everyone Should Know](#)

## 2.4. 可维护的代码

相对于前两类代码来说，可维护的代码评价标准更模糊一些，因为它要对应的是未来的情况，一般新人很难想象现在的一些做法会对未来造成什么影响。不过根据我的经验，一般来说，只要反复的提问两个问题就可以了：

- 他离职了怎么办？
- 他没这么做怎么办？

### 2.4.1. 避免重复

几乎所有程序员都知道要避免拷代码，但是拷代码这个现象还是不可避免的成为了程序可维护性的杀手。

代码重复分为两种：模块内重复和模块间重复。无论何种重复，都在一定程度上说明了程序员的水平有问题，模块内重复的问题更大一些，如果在同一个文件里都能出现大片重复的代码，那表示他什么不可思议的代码都有可能写出来。

对于重复的判断并不需要反复阅读代码，一般来说现代的IDE都提供了检查重复代码的工具，只需点几下鼠标就可以了。

除了代码重复之外，很多热衷于维护代码质量的程序员新人很容易出现另一类重复：信息重复。

我见过一些新人喜欢在每行代码前面写一句注释，比如：

```
// 成员列表的长度>0并且<200
if(memberList.size() > 0 && memberList.size() < 200) {
 // 返回当前成员列表
 return memberList;
}
```

```
// 成员列表的长度>0并且<200
if(memberList.size() > 0 && memberList.size() < 200) {
 // 返回当前成员列表
 return memberList;
}
```

看起来似乎很好懂，但是几年之后，这段代码就变成了：

```
// 成员列表的长度>0并且<200
if(memberList.size() > 0 && memberList.size() < 200 || (tmp.isOpen() && flag)) {
 // 返回当前成员列表
 return memberList;
}
```

```
// 成员列表的长度>0并且<200
if(memberList.size() > 0 && memberList.size() < 200 || (tmp.isOpen() && flag)) {
 // 返回当前成员列表
 return memberList;
}
```

再之后可能会改成这样：

```
// edit by axb 2015.07.30
// 成员列表的长度>0并且<200
//if(memberList.size() > 0 && memberList.size() < 200 || (tmp.isOpen() && flag)) {
// // 返回当前成员列表
// return memberList;
//}
if(tmp.isOpen() && flag) {
 return memberList;
}
```

```
// edit by axb 2015.07.30
// 成员列表的长度>0并且<200
//if(memberList.size() > 0 && memberList.size() < 200 || (tmp.isOpen() && flag)) {
// 返回当前成员列表
// return memberList;
//}
if(tmp.isOpen() && flag) {
 return memberList;
}
```

随着项目的演进，无用的信息会越积越多，最终甚至让人无法分辨哪些信息是有效的，哪些是无效的。

如果在项目中发现好几个东西都在做同一件事情，比如通过注释描述代码在做什么，或者依靠注释替代版本管理的功能，那么这些代码也不能称为好代码。

## 2.4.2.模块划分

模块内高内聚与模块间低耦合是大部分设计遵循的标准，通过合理的模块划分能够把复杂的功能拆分为更易于维护的更小的功能点。

一般来说可以从代码长度上初步评价一个模块划分的是否合理，一个类的长度大于2000行，或者一个函数的长度大于两屏幕都是比较危险的信号。

另一个能够体现模块划分水平的地方是依赖。如果一个模块依赖特别多，甚至出现了循环依赖，那么也可以反映出作者对模块的规划比较差，今后在维护这个工程的时候很有可能出现牵一发而动全身的情况。

一般来说有不少工具能提供依赖分析，比如IDEA中提供的[Dependencies Analysis](#)功能，学会这些工具的使用对于评价代码质量会有很大的帮助。

值得一提的是，绝大部分情况下，不恰当的模块划分也会伴随着极低的单元测试覆盖率：复杂模块的单元测试非常难写的，甚至是不可能完成的任务。所以直接查看单元测试覆盖率也是一个比较靠谱的评价方式。

## 2.4.3.简洁与抽象

只要提到代码质量，必然会提到简洁、优雅之类的形容词。简洁这个词实际涵盖了很多东西，代码避免重复是简洁、设计足够抽象是简洁，一切对于提高可维护性的尝试实际都是在试图做减法。

编程经验不足的程序员往往不能意识到简洁的重要性，乐于捣鼓一些复杂的玩意并乐此不疲。但复杂是代码可维护性的天敌，也是程序员能力的一道门槛。

跨过门槛的程序员应该有能力控制逐渐增长的复杂度，总结和抽象出事物的本质，并体现到自己设计和编码中。一个程序的生命周期也是在由简入繁到化繁为简中不断迭代的过程。

对于这部分我难以总结出简单易行的评价标准，它更像是一种思维方式，除了要理解、还需要练习。多看、多想、多交流，很多时候可以简化的东西会大大超出原先的预计。

## 2.4.4.推荐阅读

- 《重构-改善既有代码的设计》
- 《设计模式-可复用面向对象软件的基础》
- 《Software Architecture Patterns-Understanding Common Architecture Patterns and When to Use Them》

## 3.结语

这篇文章主要介绍了一些评价代码质量优劣的手段，这些手段中，有些比较客观，有些主观性更强。之前也说过，对代码质量的评价是一件主观的事情，这篇文章里虽然列举了很多评价手段。但是实际上，很多我认为没有问题的代码也会被其他人吐槽，所以这篇文章只能算是初稿，更多内容还需要今后继续补充和完善。

虽然每个人对于代码质量评价的倾向都不一样，但是总体来说评价代码质量的能力可以被比作程序员的“品味”，评价的准确度会随着自身经验的增加而增长。在这个过程中，需要随时保持思考、学习和批判的精神。

下篇文章里，会谈一谈具体如何提高自己的代码质量。

# 进程保活

# android persistent属性研究

来源:<http://blog.csdn.net/windskier/article/details/6560925>

[TOC]

为什么写这篇文章呢？前段时间在研究telephony时，一直没有在framework下发现对telephony的初始化(*PhoneFactory.java*中的`makeDefaultPhones`函数)的调用。结果全局搜索之后发现在 `application PhoneApp(packages/apps/Phone)`中调用了。但是`application PhoneApp`既没有被Broadcast唤醒，也没有被其他service调用，那么是Android是通过什么方式来启动PhoneApp，所以就发现了属性**android:persistent**。

在AndroidManifest.xml定义中，`application`有这么一个属性**android:persistent**，根据字面意思来理解就是说该应用是可持久的，也即是常驻的应用。其实就这么个理解，被**android:persistent**修饰的应用会在系统启动之后被AM启动。

AM首先去PM(**PackageManagerService**)中去查找设置了**android:persistent**的应用。

```
public void systemReady(final Runnable goingCallback) {
 if (mFactoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL) {
 try {
 List<App> apps = AppGlobals.getPackageManager().
 getPersistentApplications(STOCK_PM_FLAGS);
 if (apps != null) {
 int N = apps.size();
 int i;
 for (i=0; i<N; i++) {
 ApplicationInfo info
 = (ApplicationInfo)apps.get(i);
 if (info != null &&
 !info.packageName.equals("android")) {
 addAppLocked(info);
 }
 }
 }
 } catch (RemoteException ex) {
 // pm is in same process, this will never happen.
 }
 }
}
```

假如该被`android:persistent`修饰的应用此时并未运行的话，那么AM将调用**`startProcessLocked`**启动该app，关于`startProcessLocked`不再描述，另外一篇文章《How to start a new process for Android?》中做了详细的介绍。

app的启动过程就是启动app所在的package对应的进程。

```
final ProcessRecord addAppLocked(ApplicationInfo info) {
 ProcessRecord app = getProcessRecordLocked(info.processName, info.uid);
 if (app == null) {
 app = newProcessRecordLocked(null, info, null);
 mProcessNames.put(info.processName, info.uid, app);
 updateLruProcessLocked(app, true, true);
 }
 if ((info.flags & (ApplicationInfo.FLAG_SYSTEM | ApplicationInfo.FLAG_PERSISTENT))
 == (ApplicationInfo.FLAG_SYSTEM | ApplicationInfo.FLAG_PERSISTENT)) {
 app.persistent = true;
 app.maxAdj = CORE_SERVER_ADJ;
 }
 if (app.thread == null && mPersistentStartingProcesses.indexOf(app) < 0) {
 mPersistentStartingProcesses.add(app);
 startProcessLocked(app, "added application", app.processName);
 }
 return app;
}
```

下面介绍app所在的package对应的进程启动完成之后，app是如何被create的。

从文章《How to start a new process for Android?》中可知，zygote在创建新的进程均会启动它的mainThread `android.app.ActivityThread`，因此我们从ActivityThread的main函数中接着分析app的create过程。

在main中有下面这个操作

```
thread.attach(false);
```

在attach过程中，ActivityThread会将对应的**application attach**到AM中去，交与AM去管理。这里需要注意一个变量

```
final ApplicationThread mAppThread = new ApplicationThread();
```

`mAppThread`是一个ApplicationThread对象，`mAppThread`可以看作是当前进程主线程的核心，它负责处理本进程与其他进程(主要是AM)之间的通信，同时通过`attachApplication`将`mAppThread`的代理Binder传递给AM。

```

private final void attach(boolean system) {
 sThreadLocal.set(this);
 mSystemThread = system;
 if (!system) {
 ViewRoot.addFirstDrawHandler(new Runnable() {
 public void run() {
 ensureJitEnabled();
 }
 });
 android.ddm.DdmHandleAppName.setAppName("<pre-initialized>");
 RuntimeInit.setApplicationObject(mAppThread.asBinder());
 IActivityManager mgr = ActivityManagerNative.getDefault();
 try {
 mgr.attachApplication(mAppThread);
 } catch (RemoteException ex) {
 }
 }
}

```

上面的attach代码中，我们顺着IPC调用AM的attachApplication过程再往下看。

在该过程中，AM调用到了IPC通信调用mAppThread的bindApplication；

```

private final boolean attachApplicationLocked(IApplicationThread thread,
 int pid) {

 thread.bindApplication(processName, app.instrumentationInfo != null
 ? app.instrumentationInfo : app.info, providers,
 app.instrumentationClass, app.instrumentationProfileFile,
 app.instrumentationArguments, app.instrumentationWatcher, testMode,
 isRestrictedBackupMode || !normalMode,
 mConfiguration, getCommonServicesLocked());
 updateLruProcessLocked(app, false, true);
 app.lastRequestedGc = app.lastLowMemory = SystemClock.uptimeMillis();
}

```

mAppThread的bindApplication再通过消息机制向ActivityThread自身维护的handler发送**BIND\_APPLICATION**消息。下面看看ActivityThread自身维护的handler对消息**BIND\_APPLICATION**的处理，最终会调用到handleBindApplication函数

你会发现在handleBindApplication函数中有这么一句

```
mInstrumentation.callApplicationOnCreate(app);
```

我们最终在绕了好大一圈之后，调用了app的onCreate函数来启动这个application。



# 关于 Android 进程保活，你所需要知道的一切

来源:<http://www.jianshu.com/p/63aafe3c12af>

[TOC]

早前，我在知乎上回答了这样一个问题：[怎么让 Android 程序一直后台运行，像 QQ 一样不被杀死？](#)。关于 Android 平台的进程保活这一块，想必是所有 Android 开发者瞩目的内容之一。你到网上搜 Android 进程保活，可以搜出各种各样神乎其技的做法，绝大多数都是极其不靠谱。前段时间，Github还出现了一个很火的“黑科技”进程保活库，声称可以做到进程永生不死。



怀着学习和膜拜的心情进去Github围观，结果发现很多人提了 Issue 说各种各样的机子无法成功保活。

# 剩下我一脸楞逼



看到这里，我瞬间就放心了。坦白的讲，我是真心不希望有这种黑科技存在的，它只会滋生更多的流氓应用，拖垮我大 Android 平台的流畅性。扯了这么多，接下来就直接进入本文的正题，谈谈关于进程保活的知识。提前声明以下四点

- 本文是本人开发 Android 至今综合各方资料所得
- 不以节能来维持进程保活的手段，都是耍流氓
- 本文不是教你做永生不死的进程，如果指望实现进程永生不死，请忽略本文
- 本文有错误的地方，欢迎留下评论互相探讨（拍砖请轻拍）

## 保活手段

当前业界的Android进程保活手段主要分为 黑、白、灰 三种，其大致的实现思路如下：

- **黑色保活：**不同的app进程，用广播相互唤醒（包括利用系统提供的广播进行唤醒）

- **白色保活：**启动前台Service
- **灰色保活：**利用系统的漏洞启动前台Service

## 黑色保活

所谓黑色保活，就是利用不同的app进程使用广播来进行相互唤醒。举个3个比较常见的场景：

**场景1：**开机、网络切换、拍照、拍视频时候，利用系统产生的广播唤醒app **场景2：**接入第三方SDK也会唤醒相应的app进程，如微信sdk会唤醒微信，支付宝sdk会唤醒支付宝。由此发散开去，就会直接触发了下面的 **场景3** **场景3：**假如你手机里装了支付宝、淘宝、天猫、UC等阿里系的app，那么你打开任意一个阿里系的app后，有可能就顺便把其他阿里系的app给唤醒了。（只是拿阿里打个比方，其实BAT系都差不多）

没错，我们的Android手机就是一步一步的被上面这些场景给拖卡机的。

针对场景1，估计Google已经开始意识到这些问题，所以在最新的Android N取消了**ACTION\_NEW\_PICTURE**（拍照），**ACTION\_NEW\_VIDEO**（拍视频），**CONNECTIVITY\_ACTION**（网络切换）等三种广播，无疑给了很多app沉重的打击。我猜他们的心情是下面这样的



而开机广播的话，记得有一些定制ROM的厂商早已经将其去掉。

针对场景2和场景3，因为调用SDK唤醒app进程属于正常行为，此处不讨论。但是在借助LBE分析app之间的唤醒路径的时候，发现了两个问题：

很多推送SDK也存在唤醒app的功能

- app之间的唤醒路径真是多，且错综复杂
- 我把自己使用的手机测试结果给大家围观一下（我的手机是小米4C，刷了原生的Android5.1系统，且已经获得Root权限才能查看这些唤醒路径）



## 15组相互唤醒路径

0b/s 0b/s 22:38

## ← 自启管家

开机自启 | 相互唤醒 | **全部路径**

运行中 (20)

-  简书  
Jianshu.com  
4个被唤醒路径，1个禁止 部分允许 >
-  脉脉  
6个被唤醒路径，1个禁止 部分允许 >
-  美团  
5个被唤醒路径，1个禁止 部分允许 >
-  Android Keyboard (AO...  
1个被唤醒路径 全部允许 >
-  Baidu IME  
7个被唤醒路径 全部允许 >
-  Clock  
1个被唤醒路径 全部允许 >
-  com.android.smspush  
5个被唤醒路径 全部允许 >
-  Equalizer 全部允许 >

全部唤醒路径 我们直接点开 简书 的唤醒路径进行查看



## 简书唤醒路径

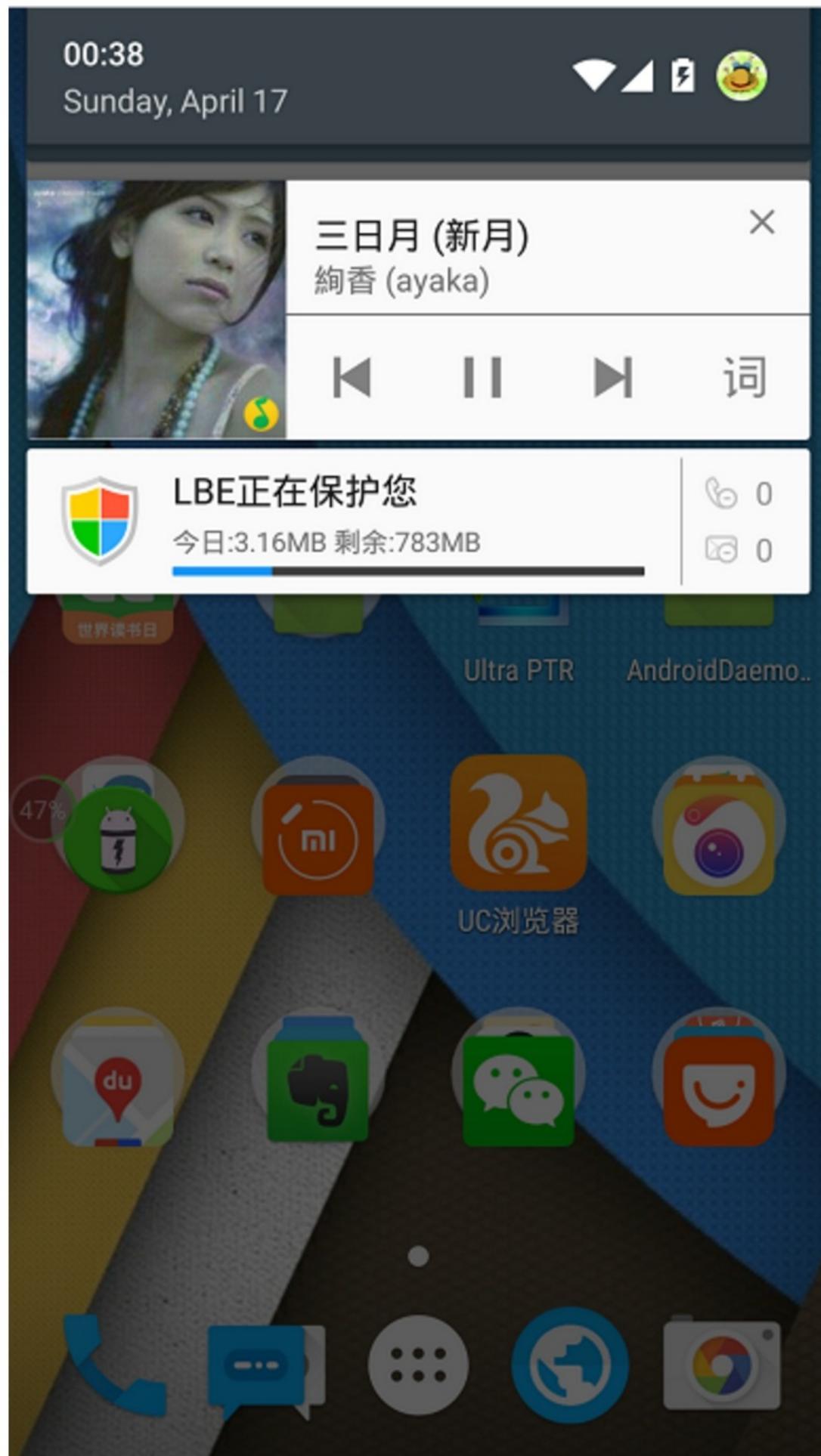
可以看到以上3条唤醒路径，但是涵盖的唤醒应用总数却达到了 $23+43+28$ 款，数目真心惊人。请注意，这只是我手机上一款app的唤醒路径而已，到了这里是不是有点细思极恐。

当然，这里依然存在一个疑问，就是LBE分析这些唤醒路径和互相唤醒的应用是基于什么思路，我们不得而知。所以我们也无法确定其分析结果是否准确，如果有LBE的童鞋看到此文章，不知可否告知一下思路呢？但是，手机打开一个app就唤醒一大批，我自己可是亲身体验到这种酸爽的……



## 白色保活

白色保活手段非常简单，就是调用系统api启动一个前台的Service进程，这样会在系统的通知栏生成一个Notification，用来让用户知道有这样一个app在运行着，哪怕当前的app退到了后台。如下方的LBE和QQ音乐这样：



# 灰色保活

灰色保活，这种保活手段是应用范围最广泛。它是利用系统的漏洞来启动一个前台的Service进程，与普通的启动方式区别在于，它不会在系统通知栏处出现一个Notification，看起来就如同运行着一个后台Service进程一样。这样做带来的好处就是，用户无法察觉到你运行着一个前台进程（因为看不到Notification），但你的进程优先级又是高于普通后台进程的。那么如何利用系统的漏洞呢，大致的实现思路和代码如下：

- 思路一：API < 18，启动前台Service时直接传入new Notification();
- 思路二：API >= 18，同时启动两个id相同的前台Service，然后再将后启动的Service做stop处理；

```

public class GrayService extends Service {
 private final static int GRAY_SERVICE_ID = 1001;
 @Override
 public int onStartCommand(Intent intent, int flags, int startId) {
 if (Build.VERSION.SDK_INT < 18) {
 //API < 18，此方法能有效隐藏Notification上的图标
 startForeground(GRAY_SERVICE_ID, new Notification());
 } else {
 Intent innerIntent = new Intent(this, GrayInnerService.class);
 startService(innerIntent);
 startForeground(GRAY_SERVICE_ID, new Notification());
 }
 return super.onStartCommand(intent, flags, startId);
 }

 ...
 ...
 /**
 * 给 API >= 18 的平台上用的灰色保活手段
 */
 public static class GrayInnerService extends Service {
 @Override
 public int onStartCommand(Intent intent, int flags, int startId) {
 startForeground(GRAY_SERVICE_ID, new Notification());
 stopForeground(true);
 stopSelf();
 return super.onStartCommand(intent, flags, startId);
 }
 }
}

```

代码大致就是这样，能让你神不知鬼不觉的启动着一个前台Service。其实市面上很多app都用着这种灰色保活的手段，什么？你不信？好吧，我们来验证一下。流程很简单，打开一个app，看下系统通知栏有没有一个 Notification，如果没有，我们就进入手机的adb shell模式，然后输入下面的shell命令

```
dumpsys activity services PackageName
```

打印出指定包名的所有进程中的Service信息，看下有没有 **isForeground=true** 的关键信息。如果通知栏没有看到属于app的 Notification 且又看到 **isForeground=true** 则说明了，此app利用了这种灰色保活的手段。

下面分别是我手机上微信、qq、支付宝、陌陌的测试结果，大家有兴趣也可以自己验证一下。

```
* ServiceRecord{372af996 u0 com.tencent.mm/.booter.CoreService}
intent={cmp=com.tencent.mm/.booter.CoreService}
packageName=com.tencent.mm
processName=com.tencent.mm:push
baseDir=/data/app/com.tencent.mm-1/base.apk
dataDir=/data/data/com.tencent.mm
app=ProcessRecord{589886f 11800:com.tencent.mm:push/u0a64}
isForeground=true foregroundId=-1213 foregroundNoti=Notification(pri=0 contentView=com.tencent.mm/0x1090077 vibrate=null sound=null defaults=0x0 flags=0x62 color=0xff607d8b vis=PRIVATE)
createTime=-49m8s963ms startingBgTimeout=-42m14s862ms
lastActivity=-5m11s338ms restartTime=-42m29s812ms createdFromFg=false
startRequested=true delayedStop=false stopIfKilled=false callStart=true lastStartId=11
Bindings:
* IntentBindRecord{e362ccc CREATE}:
intent={cmp=com.tencent.mm/.booter.CoreService}
binder=android.os.BinderProxy@7f6b15
requested=true received=true hasBound=true doRebind=false
* Client AppBindRecord{301f2e2a ProcessRecord{106a7714 11638:com.tencent.mm/u0a64}}
 Per-process Connections:
 ConnectionRecord{425f9d4 u0 CR com.tencent.mm/.booter.CoreService:@1dfcd327}
 ConnectionRecord{425f9d4 u0 CR com.tencent.mm/.booter.CoreService:@1dfcd327}
```

微信

```
* ServiceRecord{207e66f7 u0 com.tencent.mobileqq/.app.CoreService$KernelService}
intent={cmp=com.tencent.mobileqq/.app.CoreService$KernelService}
packageName=com.tencent.mobileqq
processName=com.tencent.mobileqq
baseDir=/data/app/com.tencent.mobileqq-2/base.apk
dataDir=/data/data/com.tencent.mobileqq
app=ProcessRecord{33e262bf 26530:com.tencent.mobileqq/u0a65}
isForeground=true foregroundId=537045978 foregroundNoti=Notification(pri=0 contentView=com.tencent.mobileqq/0x1090077 vibrate=null sound=null defaults=0x0 flags=0x62 color=0xff607d8b vis=PRIVATE)
createTime=-40s775ms startingBgTimeout=-40s775ms
lastActivity=-40s775ms restartTime=-40s775ms createdFromFg=true
startRequested=true delayedStop=false stopIfKilled=true callStart=true lastStartId=1
```

手Q

```
* ServiceRecord{28193c6f u0 com.eg.android.AlipayGphone/com.alipay.android.launcher.service.LauncherService$InnerService}
intent={cmp=com.eg.android.AlipayGphone/com.alipay.android.launcher.service.LauncherService$InnerService}
packageName=com.eg.android.AlipayGphone
processName=com.eg.android.AlipayGphone
baseDir=/data/app/com.eg.android.AlipayGphone-2/base.apk
dataDir=/data/data/com.eg.android.AlipayGphone
app=ProcessRecord{1ad3039b 15556:com.eg.android.AlipayGphone/u0a73}
isForeground=true foregroundId=168816881 foregroundNoti=Notification(pri=0 contentView=com.eg.android.AlipayGphone/0x1090077 vibrate=null sound=null defaults=0x0 flags=0x62 color=0xffff07d8b vis=PRIVATE)
createTime=-2m56s270ms startingBgTimeout=-
lastActivity=-2m56s270ms restartTime=-2m56s270ms createdFromFg=true
startRequested=true delayedStop=false stopIfKilled=true callStart=true lastStartId=1
```

## 支付宝

```
* ServiceRecord{36569b0b u0 com.immomo.momo/.android.service.XService}
intent={cmp=com.immomo.momo/.android.service.XService}
packageName=com.immomo.momo
processName=com.immomo.momo:im
baseDir=/data/app/com.immomo.momo-2/base.apk
dataDir=/data/data/com.immomo.momo
app=ProcessRecord{369bba5c 13252:com.immomo.momo:im/u0a106}
isForeground=true foregroundId=9998 foregroundNoti=Notification(pri=0 contentView=com.immomo.momo/0x1090077 vibrate=null sound=null defaults=0x0 flags=0x72 color=0x00000000 vis=PRIVATE)
createTime=-39m12s718ms startingBgTimeout=-38m57s663ms
lastActivity=-34s2ms restartTime=-39m12s570ms createdFromFg=false
startRequested=true delayedStop=false stopIfKilled=false callStart=true lastStartId=11
Bindings:
* IntentBindRecord{10647021}:
 intent={cmp=com.immomo.momo/.android.service.XService}
```

## 陌陌

其实Google察觉到了此漏洞的存在，并逐步进行封堵。这就是为什么这种保活方式分 **API >= 18 和 API < 18** 两种情况，从Android5.0的ServiceRecord类的postNotification函数源代码中可以看到这样的一行注释

```

public void postNotification() {
 final int appUid = appInfo.uid;
 final int appId = app.pid;
 if (foregroundId != 0 && foregroundNoti != null) {
 // Do asynchronous communication with notification manager to
 // avoid deadlocks.
 final String localPackageName = packageName;
 final int localForegroundId = foregroundId;
 final Notification localForegroundNoti = foregroundNoti;
 ams.mHandler.post(new Runnable() {
 public void run() {
 NotificationManagerInternal nm = LocalServices.getService(
 NotificationManagerInternal.class);
 if (nm == null) {
 return;
 }
 try {
 if (localForegroundNoti.icon == 0) {
 // It is not correct for the caller to supply a notification
 // icon, but this used to be able to slip through, so for
 // those dirty apps give it the app's icon.
 localForegroundNoti.icon = appInfo.icon;

 // Do not allow apps to present a sneaky invisible content view either.
 localForegroundNoti.contentView = null;
 localForegroundNoti.bigContentView = null;
 CharSequence appName = appInfo.loadLabel(
 ams.mContext.getPackageManager());
 if (appName == null) {
 appName = appInfo.packageName;
 }
 }
 }
 });
 }
}

```

当某一天 API  $\geq 18$  的方案也失效的时候，我们就又要另谋出路了。需要注意的是，使用灰色保活并不代表着你的Service就永生不死了，只能说是提高了进程的优先级。如果你的app进程占用了大量的内存，按照回收进程的策略，同样会干掉你的app。感兴趣于灰色保活是如何利用系统漏洞不显示 Notification 的童鞋，可以研究一下系统的 ServiceRecord、NotificationManagerService 等相关源代码，因为不是本文的重点，所以不做详述。

## 唠叨的分割线

到这里基本就介绍完了 黑、白、灰 三种实现方式，仅仅从代码层面去讲保活是不够的，我希望能够通过系统的进程回收机制来理解保活，这样能够让我们更好的避免踩到进程被杀的坑。

## 进程回收机制

熟悉Android系统的童鞋都知道，系统出于体验和性能上的考虑，app在退到后台时系统并不会真正的kill掉这个进程，而是将其缓存起来。打开的应用越多，后台缓存的进程也越多。在系统内存不足的情况下，系统开始依据自身的一套进程回收机制来判断要kill掉哪些进程，以腾出内存来供给需要的app。这套杀进程回收内存的机制就叫 **Low Memory Killer**，它是基于Linux内核的 **OOM Killer (Out-Of-Memory killer)** 机制诞生。

了解完 **Low Memory Killer**，再科普一下 **oom\_adj**。什么是 **oom\_adj**？它是linux内核分配给每个系统进程的一个值，代表进程的优先级，进程回收机制就是根据这个优先级来决定是否进行回收。对于 **oom\_adj** 的作用，你只需要记住以下几点即可：

- 进程的 **oom\_adj** 越大，表示此进程优先级越低，越容易被杀回收；越小，表示进程优先级越高，越不容易被杀回收
- 普通app进程的 **oom\_adj >= 0**，系统进程的 **oom\_adj** 才可能 < 0

那么我们如何查看进程的 **oom\_adj** 呢，需要用到下面的两个shell命令

```
ps | grep PackageName //获取你指定的进程信息
```

这里是以我写的demo代码为例子，红色圈中部分别为下面三个进程的ID

```
shell@cancro:/ $ ps | grep com.clock.daemon
u0_a263 18937 271 1491564 49004 ffffffff 00000000 S com.clock.daemon:bg
u0_a263 19016 271 1549640 66664 ffffffff 00000000 S com.clock.daemon
u0_a263 27785 271 1490508 50936 ffffffff 00000000 S com.clock.daemon:gray
```

UI进程：**com.clock.daemon** 普通后台进程：**com.clock.daemon:bg** 灰色保活进程：**com.clock.daemon:gray**

当然，这些进程的id也可以通过AndroidStudio获得



接着我们来再来获取三个进程的 **oom\_adj**

```
cat /proc/进程ID/oom_adj
```

```
shell@cancro:/ $ ps | grep com.clock.daemon
u0_a263 18937 271 1491564 49004 ffffffff 00000000 S com.clock.daemon:bg
u0_a263 19016 271 1549640 66664 ffffffff 00000000 S com.clock.daemon
u0_a263 27785 271 1490508 50936 ffffffff 00000000 S com.clock.daemon:gray
shell@cancro:/ $ cat /proc/27785/oom_adj
0
shell@cancro:/ $ cat /proc/18937/oom_adj
15
shell@cancro:/ $ cat /proc/19016/oom_adj
0
```

从上图可以看到UI进程和灰色保活Service进程的`oom_adj=0`, 而普通后台进程`oom_adj=15`。到这里估计你也能明白, 为什么普通的后台进程容易被回收, 而前台进程则不容易被回收了吧。但明白这个还不够, 接着看下图

```
shell@cancro:/ $ ps | grep com.clock.daemon
u0_a263 18937 271 1491564 49004 ffffffff 00000000 S com.clock.daemon:bg
u0_a263 19016 271 1520460 63188 ffffffff 00000000 S com.clock.daemon
u0_a263 27785 271 1490508 50936 ffffffff 00000000 S com.clock.daemon:gray
shell@cancro:/ $ cat /proc/19016/oom_adj
6
shell@cancro:/ $ cat /proc/27785/oom_adj
1
shell@cancro:/ $
```

上面是我把app切换到后台, 再进行一次`oom_adj`的检验, 你会发现UI进程的值从0变成了6, 而灰色保活的Service进程则从0变成了1。这里可以观察到, app退到后台时, 其所有的进程优先级都会降低。但是UI进程是降低最为明显的, 因为它占用的内存资源最多, 系统内存不足的时候肯定优先杀这些占用内存高的进程来腾出资源。所以, 为了尽量避免后台UI进程被杀, 需要尽可能的释放一些不用的资源, 尤其是图片、音视频之类的。

从Android官方文档中, 我们也能看到优先级从高到低列出了这些不同类型的进程:  
**Foreground process**、**Visible process**、**Service process**、**Background process**、**Empty process**。而这些进程的`oom_adj`分别是多少, 又是如何挂钩起来的呢? 推荐大家阅读下面这篇文章:

<http://www.cnblogs.com/angeldevil/archive/2013/05/21/3090872.html>

## 总结 (文末有福利)

絮絮叨叨写完了这么多, 最后来做个小小的总结。回归到开篇提到QQ进程不死的问题, 我也曾认为存在这样一种技术。可惜我把手机root后, 杀掉QQ进程之后就再也起不来了。有些手机厂商把这些知名的app放入了自己的白名单中, 保证了进程不死来提高用户

体验（如微信、QQ、陌陌都在小米的白名单中）。如果从白名单中移除，他们终究还是和普通app一样躲避不了被杀的命运，为了尽量避免被杀，还是老老实实去做好优化工作吧。

所以，进程保活的根本方案终究还是回到了性能优化上，进程永生不死终究是个彻头彻尾的伪命题！

## 补充更新（2016-04-20）

有童鞋问，在华为的机子上发现微信和手Q的UI进程退到后台，**oom\_adj**的值一点都没有变，是不是有什么黑科技在其中。为此，我稍稍验证了一下，验证方式就是把demo工程的包名改成手机QQ的，编译运行在华为的机子上，发现我的进程怎么杀也都是不死的，退到后台**oom\_adj**的值同样不发生变化，而恢复原来的包名就不行了。所以，你懂的，手Q就在华为机子的白名单中。

文章到此结束，相关简单的实践代码请看

<https://github.com/D-clock/AndroidDaemonService>

为了感谢看完本文的童鞋，特地献上福利图片一张……请注意：

如果你屏幕旁有人在，请谨慎往下观看！！！！！！！！！

如果你屏幕旁有人在，请谨慎往下观看！！！！！！！！！

如果你屏幕旁有人在，请谨慎往下观看！！！！！！！！！

## 福利



文 / D\_clock (简书作者)

原文链接: <http://www.jianshu.com/p/63aafe3c12af>

# 微信Android客户端后台保活经验分享

来源:[infoQ](#)

[TOC]

本文为『移动前线』群在3月31日的分享总结整理而成，转载请注明来自『移动开发前线』公众号。

## 嘉宾介绍

杨干荣，微信Android客户端基础平台、性能优化负责人

保活，按照我们的理解包含两部分：

- **网络连接保活**: 如何保证消息接收实时性。
- **进程保活**: 尽量保证应用的进程不被Android系统回收。

## 1 网络连接保活

网络保活，业界主要手段有：

- a. GCM
- b. 公共的第三方push通道(信鸽等)
- c. 自身跟服务器通过轮询，或者长连接

国产机器大多缺乏GMS，在国内GCM也不稳定(心跳原因)，第三方通道需要考虑安全问题和承载能力，最后微信选择使用自己的长连接。而国外，GCM作为辅助，微信无法建立长连接时，才使用GCM。

之前看到大家在聊各种Java网络框架，而微信实际上都是没用上的。早年的微信，直接通过Java socket实现。微信v5.0后，考虑各系统平台的统一，开始使用自研c++组件。

长连接实现包括几个要素：

- a. 网络切换或者初始化时 server ip 的获取。
- b. 连接前的 ip筛选，出错后ip 的抛弃。
- c. 维护长连接的心跳。
- d. 服务器通过长连notify。
- e. 选择使用长连通道的业务。
- f. 断开后重连的策略。

今天主题在保活， 我们重点讨论心跳和 notify 机制。

## 1.1 心跳机制

心跳的目的很简单：通过定期的数据包，对抗NAT超时。以下是部分地区网络NAT 超时统计：

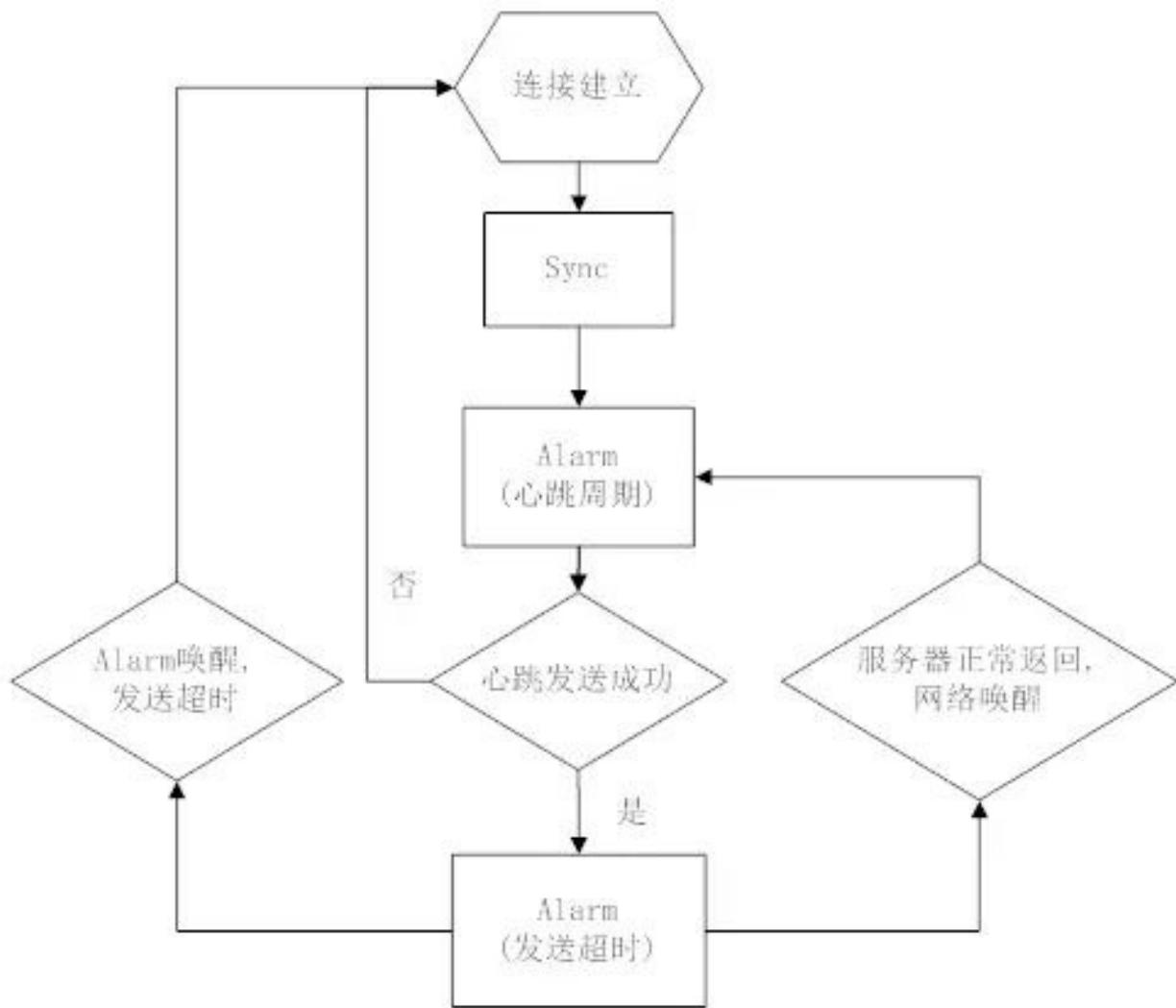
| 地区/网络        | NAT 超时时间 |
|--------------|----------|
| 中国移动 3G 和 2G | 5 分钟     |
| 中国联通 2G      | 5 分钟     |
| 中国电信 3G      | 大于 28 分钟 |
| 美国 3G        | 大于 28 分钟 |
| 台湾 3G        | 大于 28 分钟 |

上表说明：

- a. GCM无法适应国内2G环境(GCM 28分钟心跳)。
- b. 为了兼容国内网络要求，我们至少5分钟心跳一次。

老版本的微信是4.5分钟发送一次心跳，运行良好。

心跳的实现：



- a. 连接后主动到服务器Sync拉取一次数据，确保连接过程的新消息。
- b. 心跳周期的Alarm 唤醒后，一般有几秒的cpu 时间，无需wakelock。
- c. 心跳后的Alarm防止发送超时，如服务器正常回包，该Alarm 取消。
- d. 如果服务器回包，系统通过网络唤醒，无需wakelock。

流程基于两个系统特性：

- a. Alarm唤醒后，足够cpu时间发包。
- b. 网络回包可唤醒机器。

特别是b项，假如Android封堵该特性，那就只能用GCM了。API level >= 23的doze就关闭所有的网络，alarm等。但进入doze条件苛刻，现在6.0普及低，至今微信没收到相关投诉。另Google也最终加入REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS权限。

## 1.2 动态心跳

4.5min心跳周期是稳定可靠的，但无法确定是最大值。通过终端的尝试，可以获取到特定用户网络下，心跳的最大值。

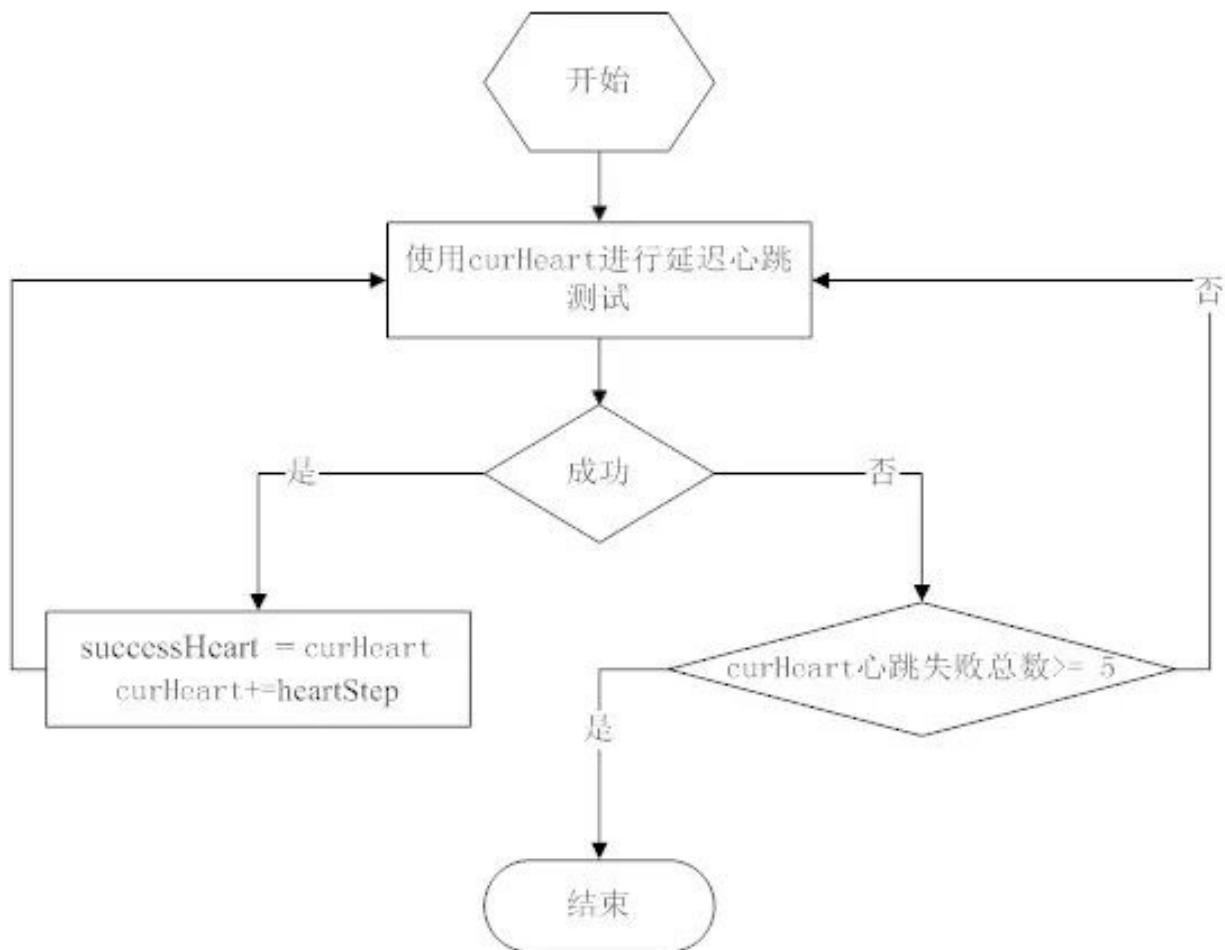
引入该特性的背景：

- a. 运营商的信令风暴
- b. 运营商网络换代，NAT超时趋于增大
- c. Alarm耗电，心跳耗流量。

动态心跳引入下列状态：

- a. 前台活跃态：亮屏，微信在前台，周期minHeart (4.5min)，保证体验。
- b. 后台活跃态：微信在后台10分钟内，周期minHeart，保证体验。
- c. 自适应计算态：步增心跳，尝试获取最大心跳周期(sucHeart)。
- d. 后台稳定态：通过最大周期，保持稳定心跳。

自适应计算态流程：



在自适应态：

- a. curHeart初始值为minHeart，步增(heartStep)为1分钟。
- b. curHeart 失败5次，意味着整个自适应态最多只有5分钟无法接收消息。
- c. 结束后，如果sucHeart > minHeart，会减去10s(避开临界)，为该网络下的稳定周期。

- d. 进入稳定态时，要求连接连续三次成功minHeart心跳周期，再使用sucHeart。

稳定态的退出：

sucHeart 会对应网络存储下来，重启后正常使用。考虑到网络的不稳定，如NAT超时变小，用户地理位置变换。当发现sucHeart 连续5次失败，sucHeart 置为minHeart，重新进入自适应态。

## 1.3 notify机制

网络保活的意义在于消息实时。通过长连接，微信有下列机制保证消息的实时。

### Sync：

通过Sync CGI直接请求后台数据。Sync 通过后台和终端的seq值对比，判断该下发哪些消息。终端正常处理消息后，seq更新为最新值。

Sync 的主要场景：

- a. 长连无法建立时，通过Sync 定期轮询
- b. 微信切到前台时，触发Sync(保命机制)
- c. 长连建立完成，立即触发Sync，防止连接过程漏消息
- d. 接收到Notify 或者 gcm 后，终端触发Sync 接收消息.

### Notify：

类似于GCM。通过长连接，后台发出仅带seq的小包，终端根据seq决定是否触发Sync拉取消息。

### NotifyData：

在长连稳定，Notify机制正常的情况下(保证seq的同步)。后台直接推送消息内容，节省1个RTT (Sync) 消息接收时间。终端收到内容后，带上seq回应NotifyAck，确认成功。这里会出现Notify和NotifyData状态互相切换的情况：

如NotifyData 后，服务器在没收到NotifyAck，而有新消息的情况下，会切换回到Notify，Sync可能需要冗余之前NotifyData的消息。终端要保证串行处理NotifyData和Sync，否则seq可能回退。

### GCM：

只要机器上有GMS，启动时就尝试注册GCM，并通知后台。服务器会根据终端是否保持长连，决定是否由GCM通知。GCM主要针对国外比较复杂的网络环境。

## 2 进程保活

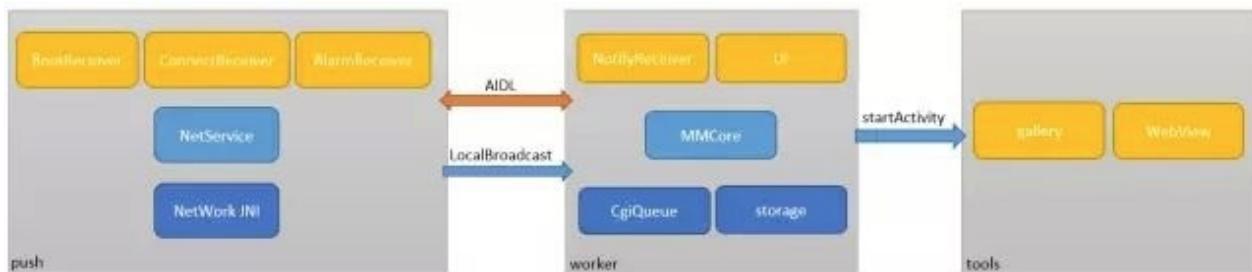
在Android系统里，进程被杀的原因通常为以下几个方面：

- a. 应用Crash
- b. 系统回收内存
- c. 用户触发
- d. 第三方root权限app.

原因a可以单独作为一个课题研究。原因c、d目前在微信上没有特殊处理。这里讨论的就是如何应对**Android Low Memory Killer**。

下面分享几个微信保活的方法：

### 2.1 进程拆分



上图表述的是微信主要的几个进程：

- a. push主要用于网络交互，没有UI
- b. worker就是用户看到的主要UI
- c. tools主要包含gallery和webview

拆分网络进程，确实就是为了减少进程回收带来的网络断开。



可以看到push的内存要远远小于worker。而且push的工作性质稳定，内存增长会非常少。这样就可以保证，尽量的减少push 被杀的可能。

这里有个思路，但限制比较多，也抛砖引玉。启动一个纯C/C++ 的进程，没有Java run time，内存使用极低。

这种做法限制很明显，如：没有Java run time，所以无法使用Android系统接口。缺乏权限，也无法使用各种shell命令操作(如am)。但可以考虑一下用途：高强度运算，网络连接，心跳维持等。比如Shadowsocks-android就如此，通过纯c命令行进程，维护着socks5代理 (Android M运行正常)。

tools进程的拆分也同样是内存的原因：

- a. 老版本的webview 是有内存泄漏的
- b. Gallery大量缩略图导致内存使用大

微信在进入后台后，会主动把tools进程Kill掉。

## 2.2 及时拉起

系统回收不可避免，及时重新拉起的手段主要依赖系统特性。从上图看到，push有 AlarmReceiver, ConnectReceiver, BootReceiver。这些receiver 都可以在push被杀后，重新拉起。特别AlarmReceiver，结合心跳逻辑，微信被杀后，重新拉起最多一个心跳周期。

而对于worker，除了用户UI操作启动。在接收消息，或者网络切换等事件，push也会通过LocalBroadcast，重新拉起worker。这种拉起的worker，大部分初始化已经完成，也能大大提高用户点击微信的启动速度。

历史原因，我们在push和worker通信使用Broadcast和AIDL。实际上，我一直不喜欢这里的实现，AIDL代码冗余多，broadcast效率低。欢迎大家分享更好的思路或者方法。

## 2.3 进程优先级

**Low Memory Killer** 决定是否杀进程除了内存大小，还有进程优先级：

| 名称               | oom_adj |
|------------------|---------|
| FOREGROUD_APP    | 0       |
| VISIBLE_APP      | 1       |
| SECONDARY_SERVER | 2       |
| HOME_APP         | 4       |
| HIDDEN_APP       | 7       |
| CONTENT_PROVIDER | 14      |
| EMPTY_APP        | 15      |

上表的数字可能在不同系统会有一定的出入，但明确的是，数值越小，优先级越高。对于优先级相同的进程，总是会把内存占用多的先kill。提高进程优先级是保活的最好手段。

正常情况下微信的oom\_adj：

```
shell@mako:/ $ ps | grep tencent.mm
u0_a74 25517 174 1172380 95448 ffffffff 00000000 S com.tencent.mm
u0_a74 25617 174 877856 38200 ffffffff 00000000 S com.tencent.mm:push
shell@mako:/ $ cat /proc/25517/oom_adj
7
shell@mako:/ $ cat /proc/25617/oom_adj
5
```

而被提高优先级后：

```
root@hammerhead:/ # ps | grep com.tencent.mm
u0_a119 8187 225 2063864 199140 sys_epoll_ b6d167e8 S com.tencent.mm
u0_a119 8657 225 1654556 83704 sys_epoll_ b6d167e8 S com.tencent.mm:push
root@hammerhead:/ # cat /proc/8187/oom_adj
1
root@hammerhead:/ # cat /proc/8657/oom_adj
1
```

从统计上报看，提高后的效果极佳。

原理：Android 的前台service机制。但该机制的缺陷是通知栏保留了图标。

对于 API level < 18：调用startForeground(ID, new Notification())，发送空的Notification，图标则不会显示。

对于 API level >= 18：在需要提优先级的service A启动一个InnerService，两个服务同时startForeground，且绑定同样的 ID。Stop 掉InnerService，这样通知栏图标即被移除。

这方案实际利用了Android前台service的漏洞。微信在评估了国内不少app已经使用后，才进行了部署。其实目标是让大家站同一起跑线上，哪天google 把漏洞堵了，效果也是一样的。

## QA环节

- Q：在智能心跳自适应阶段，如果5次心跳失败是否会促发重连？因为5次心跳都失败的话连接是不是已经断开了？

A：这里可能刚才描述不够清晰，任何一次心跳失败后，必然就已经断开重连了，所以每次心跳失败，对应一次重连操作。

- Q：在某些网络下，经常出现网络闪断的情况，这种情况下势必会引起频繁的socket重连，微信有没有遇到类似的情况？有没有什么优化的方法，求指教。

A：这种情况是有的，微信在前台时，我们会比较积极的更换ip重试，或者换短连ip。在后台时，如果出现频繁，会加上比较长的间隔。

- Q：之前看微信的架构分享，貌似是通过单一Activity，用多个Fragment切换来实现的多窗口。如果分进程的话，看起来Gallery和 WebView是单独的一个Activity，我的理解是否正确呢？以及进入后台之后，为何只kill tools而不一起释放work呢？

A：Fragment 的改造只是用在有限的几个UI上，大部分的UI，对于切换时间要求不高，还是保留成activity， Gallery和WebView都是单独的 activity，所以才可能另外一个进程的。对于我们来说worker的保活仅次于网络的push，worker如果频繁被杀，用户每次启动微信都需要等待，这个就不好了。所以，我们在后台，只会kill tools，不会主动kill worker。

- Q：除了提高进程的优先级，微信在内存方面有什么处理或优化的技术吗？

A：不可否认，其实微信是内存大户了，现阶段我们主要关注内存泄漏，没有专门去减少内存的使用，毕竟内存意味着cache，意味着用户体验更快，后续对于内存优化我们有一些规划，比如说，在cache这块照顾一些低端机。

- Q：我记得很久以前听说过微信使用一个像素的浮动窗口来保活，不知道现在还有没有呢？

A：我们有想过，也听说过有其他app是这样做的，但从来没实现过这个方案。

- Q：多端同时登录情况(手机，电脑同时登录)，假如有一端网络情况不好，怎么保证收到消息一致性？

A: 多终端登录消息一致的问题，是由后台保证的，实际原理也就是上面提到的seq。

- Q: 你们push进程与worker进程采用过socket通信方案么？采用的话效果怎么样？

A: 有考虑过用socket，后续也可能会有这种尝试，但因为push和worker依赖代码太多，伤筋动骨了，但估计也要比AIDL好，AIDL对于应用出问题后能做的事情太少了。

- Q: 有没有遇到过有一些端口被运营商封了的情况？我们之前有一些用户就是死活连不上某个端口。

A: 服务器给我们开的端口有好几个，比如80/8080/443等，而且允许服务器下发，所以实际上现在服务器会用哪些端口，终端这边都无需关注了。

- Q: 再问一个问题，服务端主动notify的话，时间间隔是如何选择的？因为这个关系到用户的流量消耗。

A: notifydata是实时的，只要你的状态允许，你的好友给你发消息的时候就会立即在服务器转换成notifydata给到你，所以这里的频率并不在于时间间隔，而在于你接收消息然后返回ack的间隔。流量消耗上，实际要比触发sync更少。

- Q: 这种保活机制会极大的增加app的耗电量，在可以通过GCM稳定唤醒app的场景下，是否可以停用后台保活，从而省电？

A: 其实心跳机制真的不会带来多少耗电，一个心跳包发出和接收，实际的消耗远远低于您收发一条消息。心跳间隔时间微信实际不会使用任何cpu的。唤醒机制靠的是网络回包。

GCM唤醒这种模式国外有app是这样做的，但还是因为国内GCM不靠谱，另外这种模式要比notify, notifydata都慢。

# 知乎问题：怎么让 Android 程序一直后台运行，像 QQ 一样不被杀死？

来源:[知乎](#)

[TOC]

## 问题

怎么让 Android 程序一直后台运行，像 QQ 一样不被杀死？

我开发了一个应用，由于需求，需要开启许多service许多线程一直运行（刷一些网站数据，挂得越久刷得越多），用户想要的当然是开启后一直挂在后台不管，但是这个应用显然很耗费资源，长时间在后台很容易被Android杀死，就是请问各位大手，有没有什么办法像QQ一样一直在后台跑，或者说怎么提升app权限，要他不那么容易被杀死

## 回答1

作者：[闭关写代码](#)

链接：<https://www.zhihu.com/question/29826231/answer/71207109>

强烈建议不要这么做，不仅仅从用户角度考虑，作为Android开发者也有责任去维护Android的生态环境。现在很多Android开发工程师，主力机居然是iPhone而不是Android设备，感到相当悲哀。

从技术角度概括一下现在普遍的防杀方法

- 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
- 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
- 双进程Service：让2个进程互相保护，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
- QQ黑科技：在应用退到后台后，另起一个只有 1 像素的页面停留在桌面上，让自己保持前台状态，保护自己不被后台清理工具杀死
- 在已经root的设备下，修改相应的权限文件，将App伪装成系统级的应用（Android4.0系列的一个漏洞，已经确认可行）

- 联系厂商，加入白名单
- 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。鉴于目前提到的在Android-Service层做双守护都会失败，我们可以fork出c进程，多进程守护。死循环在那检查是否还存在，具体的思路如下（Android5.0以下可行）
  - 用C编写守护进程(即子进程)，守护进程做的事情就是循环检查目标进程是否存在，不存在则启动它。
  - 在NDK环境中将1中编写的C代码编译打包成可执行文件(BUILD\_EXECUTABLE)。
  - 主进程启动时将守护进程放入私有目录下，赋予可执行权限，启动它即可。

---

**TIP:** 面对各种流氓软件后台常驻问题，建议使用“绿色守护”来解决，可是杀掉那些第三方清理工具难以清除的后台程序

## 回答2

c/c++混合编程，让service常驻后台，不过特别讨厌这样的做法，高一点的安卓版本已经不可以了，还有祝楼主公司早点倒闭，做出这么恶心的应用！

## 回答3

用Foreground Service.

另外，android:persistent只对System App管用

QQ之所以不会被杀死是因为各个手机厂商不想让它被杀死

## 回答4

作者：clock 链接：<https://www.zhihu.com/question/29826231/answer/79475911>

作为一个Android程序狗，来回答一下这问题，首先从性能上讲，告诉题主千万别这么做。。

别做死，别做死，别做死，

好，进入正题。

从我目前的研究角度来讲，保活方式可以分黑白灰三种。

白～直接按照系统那样，生成前台service，在notification栏可见到一天bar横在那里，这种是系统提供的合法保活方式。

灰～在白的方式上，利用系统漏洞开启前台service，但是不会在N栏上出现一条bar，这个bug在Android4.3后已经被Google修复。。

黑，最无耻的方式，拉帮结派。例如百度全家桶那样，一人得道，全家开启。。。呵呵哒，我不会告诉你像，微信这样的应用在我的手机里面有二三十条唤醒路径。。其他脉脉，小米，陌陌之类的APP也都不是神马善类，同样几十条唤醒路径。所以，你就可以知道为什么Android机子会慢慢卡成一坨翔，这他妈没root过的手机开个bat系的APP，能把你一大堆APP在后台给你搞活了。。。。。

最后还有一种属于底层一些的，就是利用C Cpp jni fork一个C进程。。但是这种方式也在Android L上被Google封杀。

所以，醒醒吧，QQ 微信没你想的那么神，我把手机root后，直接干掉他们，完全重启不了。

## 回答5

Larry Howell

这问题让我想起了 android:persistent 这个属性，据说可以保证app始终运行，但本人没有测试过，而且一直运行感觉挺流氓的。。

## 回答6

作者：nekocode

链接：<https://www.zhihu.com/question/29826231/answer/70255956>

题主试试这个：[droidwolf/NativeSubprocess · GitHub](https://github.com/droidwolf/NativeSubprocess)

创建 linux 子进程的 so 库，当初用在 service 免杀上，经测试，在大部分机子上有用。

安全软件卸载后调出浏览器苦苦哀求"主人，为什么要抛弃我..."页面是怎么做到的？service 经常莫名挂了肿么办？用 NativeSubprocess 一切都很简单。

NativeSubprocess 是一个可以让你在 android 程序中创建 linux 子进程并执行你的 java 代码的 so 库。由于市面上典型的内存清理工具值清理 apk 包关联的进程，而不会处理 linux 原生进程，所以 NativeSubprocess 可以做什么您懂滴！

## 回答7

作者：[子墨](<https://www.zhihu.com/people/zzimoo>)

链接：<https://www.zhihu.com/question/29826231/answer/55997438>

这里感谢各位的回答，我最后是这样解决的，基本的service用START\_STICKY以及轮询唤醒电池就不多说了，重点就是我在基类service的oncreate中定义了一个notification，重要属性：**notification.flags =**

**Notification.FLAG\_NO\_CLEAR|Notification.FLAG\_ONGOING\_EVENT;**然后**startForeground(setClass().hashCode(), notification);**使得服务能挂在通知栏，感谢1、3楼提供的思路。接着我监听了一个系统广播Intent.ACTION\_TIME\_TICK，这个广播每分钟发送一次，我们可以每分钟检查一次Service的运行状态，如果已经被结束了，就重新启动Service。关键代码：

```
boolean isServiceRunning = false;
ActivityManager manager = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE)
for (RunningServiceInfo service : manager.getRunningServices(Integer.MAX_VALUE)) {
 if("com.XXX.XXX.XXXService".equals(service.service.getClassName())){
 isServiceRunning = true;
 }
}
if (!isServiceRunning) {
 Intent i = new Intent(context, com.XXX.XXX.XXXService.class);
 i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
 context.startService(i);
}
```

## 回答8

作者：安地

链接：<https://www.zhihu.com/question/29826231/answer/71652980>

进程常驻你的对手不是Android系统，而是各大rom。

先说结论：没有root不是系统应用的情况下在国内主流系统下绕过系统实现进程常驻是基本不可能实现的。（抓住系统漏洞或许可以，在系统补丁又会让你失效）

道高一尺魔高一丈。国产rom就是魔，也是Android开发者的梦魇。

再讲过程：

我先试一些正常方法，@红楼 介绍的前面两种方法

- 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
- 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

主动杀掉之后服务不能再起来。

再试了双进程守护，可以做到在设置里面杀掉一个服务另一个马上起来，也可以防止一些清理工具，但直接在任务管理器杀死还是不行。

传说中的1dp并不能防止主动被杀死，只是把应用当成在前台，可以防止清理工具清理。

都不行后我就想知道微信和QQ是怎么做到杀不死的呢？

现在很多国产rom（小米，魅族，华为等）都有一个自启权限。android系统本身是没有这个权限管理的，本来是所有应用都可以申请此权限，然后借此权限实现开机自启动。但过程rom额外加了这个权限，没有这个权限无法自启动，有了这个权限系统可以开机自启动，还可以帮你在服务被杀死后自动重启，就是杀不死的应用了。一般应用安装时自启都是默认关的，而微信和QQ是默认开的。这就是厂商白名单。我写了个应用包名改成微信，就可以常驻后台杀不死了。你们也可以自己试试，把微应用的自启动关掉，然后再把它杀掉，它就再也起不来了。

我们的应用有常驻需求，所以现在的方案是推荐用户自己去打开自启动权限。

## 回答9

作者：梁秋实 链接：<https://www.zhihu.com/question/29826231/answer/96125226>

刑法第286条第一款规定：违反国家规定，对计算机信息系统功能进行删除、依赖性、增加、干扰，造成计算机信息系统不能正常运行，后果严重的，处5年以下有期徒刑或者拘役；后果特别严重的，处5年以上有期徒刑。

第三款规定：故意制作、传播计算机病毒等破坏性程序，影响计算机系统正常运行，后果严重的，依照第一款的规定处罚。

## 回答10

android:persistent

前台service

监听各种系统广播，每收到一个广播，都去启动service。

多进程相互保活。

然后多个应用相互保活.....

越到后面越流氓！！！

# 问题解决

# Ubuntu64位系统无法使用Android命令问题解决

来源：

- 解决64位Ubuntu无法使用adb、aapt的32位兼容问题
- Cannot run program "/android-sdk-linux/aapt.exe": error=2, 没有那个文件或目录
- Why can't I run the android emulator?

## 解决64位Ubuntu无法使用adb、aapt的32位兼容问题

Ubuntu从13.10就已经去除了对ia32-lib这一32位库的支持，使得很多基于32位库的应用无法正常使用，比如BCompare、adb、aapt等。新版本的BCompare(4.0+)已经支持新架构的Ubuntu，这里不再赘述，我们只简单的给出adb和aapt兼容解决方案。

### 1、兼容adb

```
sudo apt-get install lib32stdc++6
```

### 2、兼容aapt

```
sudo apt-get install zlib1g:i386
```

adb和aapt是Android SDK的核心组件，解决这两个组件的兼容问题，就可以顺利的在64bit版本的Linux系统中进行相关开发了。

ps:测试基于ubuntu 15.04 64bit版本,可用。

相关文章：

- 在你的Ubuntu上安装ADB (Android Debugging Bridge) : <http://www.linuxdiyf.com/linux/8478.html>
- Ubuntu下安装ADB: <http://www.linuxdiyf.com/linux/6114.html>
- 64位Ubuntu用不了adb: <http://www.linuxdiyf.com/linux/3697.html>

## Cannot run program "/android-sdk-linux/aapt.exe": error=2, 没有那个文件或目录

在用ant编译打包android的apk文件时报错： Execute failed: java.io.IOException: Cannot run program "/android-sdk-linux/aapt.exe": error=2, 没有那个文件或目录

首先，确定环境变量没有问题，谷歌之

解决：由于系统为Ubuntu 64位系统，而aapt工具需要32位库的支持才能运行因此执行：`sudo apt-get install ia32-libs` 安装32位库

安装好后仍不行，依然是这个报错，细想了下，linux系统没有exe这样的后缀，而 build.xml 是Windows上复制的，需要修改

```
<condition property="exe" value=".exe" else=""><os family="windows" /></condition>
<condition property="bat" value=".bat" else=""><os family="windows" /></condition>
<property name="aapt" value="${android_platform-tools}/aapt${{exe}}" />
<property name="aidl" value="${android_platform-tools}/aidl${{exe}}" />
<property name="dx" value="${android_platform-tools}/dx${{bat}}" />
<property name="apk-builder" value="${android-tools}/apkbuilder${{bat}}" />
<property name="proguard-home" value="${android-tools}/proguard/lib"/>
```

将build.xml做如上修改，根据不同平台做个判断，当在Windows系统中时，tools下的工具均带有exe、bat后缀，否则则为空，不带后缀。

在linux中终于不报错了，但在jenkins中构建时仍然报这个错，原因在于构建时，使用SVN上传到服务器中的代码中Linux中并不是使用的root用户权限，而是另一个用户的权限，当ant打包时会产生一些新的文件，而这些文件是root权限的，导致在编译过程中出现跨用户。

解决：在配置环境变量时确保不同用户均可以找到aapt，尽量让jenkins下的工作空间处于同一用户下，注意不同文件的文件权限。

```
#echo $ANDROID_HOME
#echo $JAVA_HOME
#echo $PATH //查看当前用户环境变量
```

## Why can't I run the android emulator?

问题：

```
I have installed everything like I was told to by the android website
and all I keep getting after I create my avd is
```

```
"Failed to start emulator: Cannot run program
"/home/christopher/Desktop/android-sdk-linux_86//tools/emulator":
java.io.IOException: error=2, No such file or directory".
Anybody got any ideas??? I'm running linux if that helps.
```

## 解决方案：

If you're running a 64-bit system, you need to install ia32-libs:

```
sudo apt-get install ia32-libs
```

---

If you are running Ubuntu 13.10 x64 or the latest Linux Mint x64 then the ia32-libs package is not available anymore. The solution which worked for me without any problems is to:

```
sudo apt-get install libc6-i386 lib32stdc++6 lib32gcc1 lib32ncurses5 lib32z1
```

Hope this will help!

---

## Try this, for me work fine

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libncurses5:i386 libstdc++6:i386 zlib1g:i386
```

# 项目经验

# 聚划算客户端

# 聚划算android客户端1期教训总结

来源:[Liter's Blog](#)

## 技术方面

- 1.一般性控件需要设置onclick事件才会有点击效果（selector）。
- 2.要写在selector的最后才会有点击效果。
- 3.制作.9格式图片选最小图，否则默认大小撑大控件。
- 4.如果将一个对象的属性设置为static，那么就算对象实例被回收了，该属性也存在内存，生命周期为app的生命周期。
- 5.OOM：普通视图和listview等大数据量展示视图的图片控制分开来。
- 6.OOM：listview等列表1秒真释放，把大数据量加载后不用的图片释放。
- 7.OOM：大图片使用前压缩。
- 8.OOM：减少大图。美工将全部规则图最小化，制作.9格式，以最小程序度占用内存。
- 9.OOM：背景图、大图谨慎使用，不规则大图显式释放。10.提前得知大小可使用，view.measure(-1, -1);但是view必须得有父view。11.listview列表呈现多种样式，getViewTypeCount()方法返回全部样式的总数，getItemViewType(int pos)返回的值必须小于getViewTypeCount()，否则报错。12.公共类、接口、基类设计要职责清晰易理解，最大量减少别人使用时的难度。
- 13.OOM：webview 内存溢出（OOM），重启一个新进程。

并且设置：需要在onPause时停止Timer，解决由于Timer在，导致WebCoreThread一直在，WebviewCache.db被锁定，图文详情无法进入的问题webview.pauseTimers();

当Activity返回onResume时WebView.enablePlatformNotifications();webview.resumeTimers();

14.触控范围要作为一个规范来控制到开发的每一步中，有src属性的可以设置padding，没有的为了不失真，套一个layout，最小宽度48dp（9mm）。

15.请求带有时间戳请注意，yy-MM-dd hh:mm:ss是12小时制格式。yy-MM-dd HH:mm:ss是24小时制格式。差别巨大。

16.基础数据类型的封装类型是有预装缓存的，JVM给Byte缓存了-128~127的对象，Integer缓存了-128~127。所以Integer i =k, Integer j =k, , k = 127, i == j为true, k=128则为false。

17.逻辑条件加紧要慎重，放宽松更要慎重；放宽后考虑是否更引发副作用问题，聚划算将id=()、itemId=(), i()都抓下拉起详情，结果频繁无辜拉起。收紧后考虑是否会引起扩展问题

18.最后一刻加上的代码要严格的测试，很多时候就是最后‘以为’加上了‘无关紧要’的东西而导致崩溃掉。

19.Math.abs()取到的不一定是正数，Integer.minValue就是负值。

20.多线程请使用并发容器放置变量，不轻易认为机会少不会冲突，并发量一大什么都有可能。ThreadPool.shutdownNow()之后只是清除等待队列，然后等待活动线程执行完。

21.强转类型之前先先确定对象不为空。

22.android2.3以下版本listView.setDividerHeight()函数调用后，notifyDataSetChanged()便不能记住位置。可使用setSelection记住位置。

23.finish和startActivity位置很重要. 由A跳转向SingleTask的Activity B, A.finish的位置在startActivityB之前，退出B按home回到(home键退出或back键finishB)应用界面仍然是B，无论B是否是action.MAIN，overridePendingTransition需要在finish或者startActivity之后才有效。

24.区域事件拦截：比如只要ViewA获取点击事件而组织其父控件和其他子控件触发事件，可重写activity的dispatchTouchEvent()函数，调用ViewA.getHitRect(rect),初始化一个Rect，判断event的getX和getY如果在rec之内，拦截ACTION\_DOWN返回true，其余ACTION调用ViewA.dispatchTouchEvent()即可拦截事件。

25.一次有效触摸，当ACTION\_DOWN返回ture时，其他事件也不会在得到响应。当event在rect之外时，可以通过  
event.setAction(MotionEvent.ACTION\_DOWN);activity.onTouchEvent(event);来重新触发事件。

26.WebView：缓存与不缓存，很关键。尤其在活动、计时、含session界面。

27.WebView：当webview占用大量内存时，可以将WebView全部启动在另一个进程中。

28.WebView：当多个重定向干扰或不能后退到上一页时，不使用webview.goBack()，自己用栈Stack维护Url，其关键在于区分是否是重定向，目前采用java调用js获取、分析网页内容判断是否重定向，如果不是再将url放入stack，反之不入栈。

29.无线电波状态机：应用运行在前台考虑避免延迟阻塞，运行在后台关注电量浪费。优化网络连接：预取数据，批量传输与连接（包含携带、顺带其他数据），减少连接次数（规避高频心跳）。

30.当listview含有Header时，在onItemClick事件中请这样获取ItemObject： Object obj = parent.getAdapter().getItem(position); 先判空，再强转为需要的对象。

31.WebView：注意对下载文件的支持；shouldOverrideUrlLoading返回false，会自动加载该页，返回true不会加载网页，需要自己处理（之前返回true，调用WebView.load(url)结果造成重定向网页不能回退的问题，自己花了很大代价才解决，直接返回false会自动加载）。

32.使用一个函数，尤其别人写的函数，不管怎么诚恳的承诺参数不会为null，请尽量做非空判断。除以一个变量之前，先确定其不为0.

33.如果程序自启动，或者后台耗流量，首先检测manifest中静态注册的广播，它会拉起程序。

34.findbugs结合使用ADT（16以后）自带的lint检测程序中的问题，lint可以检测出未使用的图片和更具android特性的问题。

35.View onMeasure之后，width不一定有值，如果设置了LayoutParams那么view.getLayoutParams().width将有设定值。

36.Gallery特性改善：一次触摸只切换一张图片：复写onFling直接返回true；使触摸更加灵敏：复写onScroll 调用super.onScroll(e1, e2, distanceX \* 1.5f, distanceY)，使distanceX变大就更加灵敏。

37.Gallery视觉优化：setStaticTransformationsEnabled(true)之后，getChildStaticTransformation方法生效，默认方法会使图片alpha值改变而视觉不清，复写可以利用Camera产生xyz和角度的改变，从而优化视觉体验，比如打造3D画廊。

38.可共用的对象属性用static来保持一份节省资源，每个实例或者对象单独享用的属性切记不要static。

39.改变一个类的私有属性：

```
Field field = ViewGroup.class.getDeclaredField("hsl"); field.setAccessible(true);
field.set(listView, 0);
```

40.Which client is best?

Apache HTTP client has fewer bugs on Eclair and Froyo. It is the best choice for these releases.

For Gingerbread and better, HttpURLConnection is the best choice. Its simple API and small size makes it great fit for Android. Transparent compression and response caching reduce network use, improve speed and save battery. New applications should use HttpURLConnection; it is where we will be spending our energy going forward.

## 产品层面

- 1.特效的运用要考虑用户的心理趋向和感受，炫的效果很酷，可能会让人产生某种心理导向。
- 2.特效的运用要考虑将来功能的扩展，一味的渲染也不一定很好，可能会为后续功能带来难度。
- 3.一个产品能帮助用户解决至少一个问题，那解决问题会有主流程，引导要强化，比如按钮大，颜色显眼，字体稍大，放置位置移动少。
- 4.产品不能为照顾运营而偏离主题，或者严重影响美观和主要功能，聚划算妹纸儿们说广告banner曝光度不够，设计变为由轮转变为平铺，但是广告良莠不齐，视觉效果不好，并且主页list仅能显示1、2个商品，影响了主题功能。

## 项目方面

- 1.计划进度，前紧后慢，提前实施。
- 2.不拘泥与细节，不干扰整体进度。
- 3.个人有担当，能者多劳，最大程度保证进度。
- 4.有问题不能包着，有些项目问题不是一个人的问题，不是一人能担得住的。
- 5.宝不能压在责任无关人身上，别人的资源、承诺要充分利用，但是最终还是靠自己突破。
- 6.干一样活的人放在一起，沟通迅速，快速解决问题。
- 7.早上碰一下，清晰任务；晚上碰一下一下，检查进度。
- 8.bug先把基础性、功能性问题解决，适配、显示、兼容问题靠后。
- 9.不同网络机型下的跑测，不同网络环境下的测试提前入测。
- 10.写完一天代码，尽量坚持用findbugs查找潜在危险。

11.同一版本多次发布，或多个版本发布后，记录下的crash问题如果没有版本属性就很难定位，混淆map也对应不上。发板后立即封代码，保证bug日志中的bug发生行数和源码一致。

12.开发新功能、打包、发布过程规范化，规范减少问题发生概率。

13.日志规范：

- 1). 冗长、惯例日志打到Verbose级别，例如网络返回信息，经常携带大量的信息，会严重干扰视觉；
- 2). 细粒度信息打到Debug级别，对出问题时调试有帮助作用，比如某个参数的值，我并不想知道是多少，但某些情况我就得看一下；
- 3). 粗粒度级别信息打到Info级别，比如应用业务主逻辑，运行中重要的过程或关键点；
- 4). 会出现潜在错误的信息或者可以接受的错误打到Warn级别，比如某个地方我想警告一下说这里是有问题的，或者可能引出其他什么问题；
- 5). 错误或者致命信息，理应放到Error级别，血红的颜色很显眼~

## 管理方面

1.会议总结问题，就一定要着手解决，不解决会议时间就白浪费了。

2.对于人的期望，快速的给予回馈，帮助其向希望的方向发展。

更多经验分享请看我的另一篇分享：<http://www.vmatianyu.cn/summarization-of-technical-experience.html>

个人开源站点：<http://vmatianyu.cn>

# 聚划算android客户端2期总结

来源:[Liter's Blog](#)

1.掌握整体，关心细节：和测试、开发人员沟通确定每个职位的人了解各自的流程和细节；

实例：布置好一切之后，本以为完整的掌握了项目，但是在云飞（开发）和小改（测试）的疑问下其实很多问题还没有清楚，这也是必然的，所以要关心细节才能更好的掌握整体！

2.提前疏导流程和关节，排除项目中存在的风险。

实例：测试人员很担心2天的时间不能搞定测试任务，我们做了提前的流程疏导和关键节点打通，在真实PID没有的情况下，使用伪PID（测试必须的）走下流程，与合作测试部门电话沟通保证顺利进展。

3.确定问题，‘想当然，甚至是猜测’是绝对不靠谱的，需要找专门、专业的人确定是怎样。

实例：接手无线联盟初期，我什么都不了解，找到PD、TL、对方开发、运营、通过专业的人确认了专业的问题。

4.了解问题，即追根溯源才能治病除根。刨根问底

实例：Bug也不可以说基本上解决了，要确定真的是这里问题，治病除根。聚划算消息功能上线后引起了重复的问题，我们失误没有真正确认在服务器、代码上的问题，并给与彻底解决就上线了。可用 可靠 可扩展

5.终结问题，先不做定义，为引出话题，在入题之初，看我杜撰的一个问题：

A询问了你一个问题，但你杜绝确实是不知道，不过同组的B知道，你将如何答复A？

你要先靠直觉，你真遇到了同样情况，你会怎样处理？再参考下我个人理解。

这个问题有N种答案，每种都代表着不同的做事风格。

- 1 坦诚不知。你很诚实，但是在于事无补，基本是推卸了问题，同时不相信同伴。
- 2 告诉A：B知道。很自然的答案，不过这并没有终结问题而是将问题传递了；
- 3 靠自己弄清楚。能力不错，不过没有充分利用已有条件，是缓慢的解决了问题。
- 4 和B直接沟通，快速找到答案，答复A。很小的问题，我将其阐释为一个道理：避免做问题制造者和问题传递者，而是要尽量快速终结它。

- 5 时间VS质量：时间换质量，质量第一，时间有时可以适当让步。有很多问题，甚至会惹恼用户，就算按时发了版本又能如何呢？
- 6 发布下载点统一调用一个逻辑页面完成下载，统一、动态返回包地址。
- 7 测试一个功能一次通过了远远不够，一次通过、数量稍多、大量测试的结果可能远不同。
  - 原则1：质量第一。追根溯源，杜绝P3以上级别Bug出现！质量没了，一切无从谈起。
  - 原则2：联调先自检。联调问题，首先自检，杜绝己方出现问题而导致进入混沌状态。

猜测各种环节上的可能，而忽略了确认自己完全没有问题，是不理智的做法。

更多经验分享请看我的另一篇分享：<http://www.vmatianyu.cn/summarization-of-technical-experience.html>