深入探索Android启动速度优化

前言

成为一名优秀的Android开发，需要一份完备的知识体系，在这里，让我们一起成长为自己所想的那样~。

一、启动优化的意义

二、应用启动流程

1 、应用启动的类型

冷启动

热启动

三、启动耗时检测

1、查看Logcat

2、adb shell

ThisTime

TotalTime

WaitTime

特点：

3、代码打点（函数插桩）

特点

注意

AOP(Aspect Oriented Programming)打点

4、启动速度分析工具 — TraceView

使用方式

Profile CPU

TraceView小结

5、启动速度分析工具 — Systrace

使用方式：代码插桩

Systrace原理

Systrace小结

6、启动监控

1、实验室监控：视频录制

2、线上监控
四、启动优化常规方案
启动过程中的常见问题
优化区域
1、主题切换
优点
缺点
2、第三方库懒加载
3、异步初始化预备知识-线程优化
1、Android线程调度原理剖析
2、Android异步方式
3、Android线程优化实战
4、如何锁定线程创建者
5、线程收敛优雅实践初步
6、线程优化核心问题
4、异步初始化
1、核心思想
2、异步优化注意点
3、异步初始化方案演进
4、异步优化最优解：异步启动器
5、延迟初始化
1、常规方案：利用闪屏页的停留时间进行部分初始化
2、常规初始化痛点
3、延迟优化最优解：延迟启动器
6、Multidex预加载优化
1、优化步骤：
2、dex-opt过程是怎样的?
7、类预加载优化
如何找到耗时较长的类?
8、WebView启动优化
9、页面数据预加载
10、启动阶段不启动子进程

11、闪屏页与主页的绘制优化
五、启动优化黑科技
1、启动阶段抑制GC
前提条件
实现原理
缺点
2、CPU锁频
缺点
CPU工作模式
CPU的工作频率范围
3、IO优化
Linux IO知识补充
4、数据重排
1、类重排
2、资源文件重排
技术视野：
3、Hook框架
5、类加载优化（Dalvik）
1、类预加载原理
2、类加载优化过程
3、延伸：插件化和热修复
6、保活
1、厂商合作
2、微信Hardcoder
3、OPPO Hyper Boost加速引擎
六、总结
1、优化总方针
2、注意事项
1、cpu time和wall time
2、监控的完善
3、常见问题

转载自：深入探索Android启动速度优化

前言

成为一名优秀的Android开发，需要一份完备的知识体系，在这里，让我们一起成长为自己所想的那样~。

一、启动优化的意义

如果我们去一家餐厅吃饭，在点餐的时候等了半天都没有服务人员过来，可能就没有耐心等待直接走了。
对于App来说，也是同样如此，如果用户点击App后，App半天都打不开，用户就可能失去耐心卸载应用。
启动速度是用户对我们App的第一体验，打开应用后才能去使用其中提供的强大功能，就算我们应用的内部界面设计的再精美，功能再强大，如果启动速度过慢，用户第一印象就会很差。
因此，拯救App的启动速度，迫在眉睫。下面，我们来逐步深入探索提升Android App启动速度的奥秘。

二、应用启动流程

1 、应用启动的类型

冷启动

从点击应用图标到UI界面完全显示且用户可操作的全部过程。

特点

耗时最多，衡量标准

启动流程

Click Event -> IPC -> Process.start -> ActivityThread -> bindApplication -> LifeCycle -> ViewRootImpl

热启动

因为会从已有的应用进程启动，所以不会再创建和初始化Application，只会重新创建并初始化Activity。

特点

耗时较少

启动流程

LifeCycle -> ViewRootImpl

ViewRootImpl

ViewRoot是GUI管理系统与GUI呈现系统之间的桥梁。每一个ViewRootImpl关联一个Window，ViewRootImpl最终会通过它的setView方法绑定Window所对应的View，并通过其performTraversals方法对View进行布局、测量和绘制。

三、启动耗时检测

1、查看Logcat

在Android Studio Logcat中过滤关键字“Displayed”，可以看到对应的冷启动耗时日志。

2、adb shell

使用adb shell获取应用的启动时间

```
// 其中的AppstartActivity全路径可以省略前面的packageName
adb shell am start -W [packageName]/[AppstartActivity全路径]
```

执行后会得到三个时间：ThisTime、TotalTime和WaitTime，详情如下：

ThisTime

最后一个Activity启动耗时。

TotalTime

所有Activity启动耗时。

WaitTime

AMS启动Activity的总耗时。
一般查看得到的TotalTime，即应用的启动时间，包括创建进程 + Application初始化 + Activity初始化到界面显示的过程。

特点：

- 1、线下使用方便，不能带到线上。
- 2、非严谨、精确时间。

3、代码打点（函数插桩）

可以写一个统计耗时的工具类来记录整个过程的耗时情况。其中需要注意的有：

- 在上传数据到服务器时建议根据用户ID的尾号来抽样上报。
- 在项目中核心基类的关键回调函数和核心方法中加入打点。

代码如下：

```
/**
* 耗时监视器对象，记录整个过程的耗时情况，可以用在很多需要统计的地方，比如Activity的启动耗时和Fragment的启动耗时。
*/
public class TimeMonitor {
    private final String TAG = TimeMonitor.class.getSimpleName();
    private int mMonitord = -1;
    // 保存一个耗时统计模块的各种耗时，tag对应某一个阶段的时间
    private HashMap<String, Long> mTimeTag = new HashMap<>();
    private long mStartTime = 0;
```

```
10      public TimeMonitor(int mMonitorId) {
11          Log.d(TAG, "init TimeMonitor id: " + mMonitorId);
12          this.mMonitorId = mMonitorId;
13      }
14      public int getMonitorId() {
15          return mMonitorId;
16      }
17      public void startMonitor() {
18          // 每次重新启动都把前面的数据清除，避免统计错误的数据
19          if (mTimeTag.size() > 0) {
20          mTimeTag.clear();
21          }
22          mStartTime = System.currentTimeMillis();
23      }
24      /**
25      * 每打一次点，记录某个tag的耗时
26      */
27      public void recordingTimeTag(String tag) {
28          // 若保存过相同的tag，先清除
29          if (mTimeTag.get(tag) != null) {
30              mTimeTag.remove(tag);
31          }
32          long time = System.currentTimeMillis() - mStartTime;
33          Log.d(TAG, tag + ": " + time);
34          mTimeTag.put(tag, time);
35      }
36      public void end(String tag, boolean writeLog) {
37          recordingTimeTag(tag);
38          end(writeLog);
39      }
40      public void end(boolean writeLog) {
41          if (writeLog) {
42              //写入到本地文件
43          }
44      }
45      public HashMap<String, Long> getTimeTags() {
46          return mTimeTag;
47      }
48  }
```

为了使代码更好管理，定义一个打点配置类：

```
/**
* 打点配置类，用于统计各阶段的耗时，便于代码的维护和管理。
*/
public final class TimeMonitorConfig {
    // 应用启动耗时
    public static final int TIME_MONITOR_ID_APPLICATION_START = 1;
}
```

因为，耗时统计可能会在多个模块和类中需要打点，所以需要一个单例类来管理各个耗时统计的数据：

```
/**
* 采用单例管理各个耗时统计的数据。
*/
public class TimeMonitorManager {
    private static TimeMonitorManager mTimeMonitorManager = null;
private HashMap<Integer, TimeMonitor> mTimeMonitorMap = null;
    public synchronized static TimeMonitorManager getInstance() {
        if (mTimeMonitorManager == null) {
            mTimeMonitorManager = new TimeMonitorManager();
        }
        return mTimeMonitorManager;
    }
    public TimeMonitorManager() {
        this.mTimeMonitorMap = new HashMap<Integer, TimeMonitor>();
    }
    /**
     * 初始化打点模块
    */
    public void resetTimeMonitor(int id) {
        if (mTimeMonitorMap.get(id) != null) {
            mTimeMonitorMap.remove(id);
        }
        getTimeMonitor(id);
    }
    /**
    * 获取打点器
    */
    public TimeMonitor getTimeMonitor(int id) {
```

```
        TimeMonitor monitor = mTimeMonitorMap.get(id);
        if (monitor == null) {
            monitor = new TimeMonitor(id);
            mTimeMonitorMap.put(id, monitor);
        }
        return monitor;
    }
}
```

主要在以下几个方面需要打点：

- 应用程序的生命周期节点。
- 启动时需要初始化的重要方法，如数据库初始化，读取本地的一些数据。
- 其他耗时的一些算法。

例如，启动时在Application和第一个Activity加入打点统计：

Application：

```
@Override
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);
    TimeMonitorManager.getInstance().resetTimeMonitor(TimeMonitorConfig.TIME_MONITOR_ID_APPLICATION_START);
}
@Override
public void onCreate() {
    super.onCreate();
    SoLoader.init(this, /* native exopackage */ false);
    TimeMonitorManager.getInstance().getTimeMonitor(TimeMonitorConfig.TIME_MONITOR_ID_APPLICATION_START).recordingTimeTag("Application-onCreate");
}
```

第一个Activity：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    TimeMonitorManager.getInstance().getTimeMonitor(TimeMonitorConfig.TIME_MONITOR_ID_APPLICATION_START).recordingTimeTag("SplashActivity-onCreate");
    super.onCreate(savedInstanceState);
    initData();
    TimeMonitorManager.getInstance().getTimeMonitor(TimeMonitorConfi
```

```
    g.TIME_MONITOR_ID_APPLICATION_START).recordingTimeTag("SplashActivit
    y-onCreate-Over");
 7  }
 8  @Override
 9  protected void onStart() {
10      super.onStart();
11      TimeMonitorManager.getInstance().getTimeMonitor(TimeMonitorConfi
    g.TIME_MONITOR_ID_APPLICATION_START).end("SplashActivity-onStart", f
    alse);
12  }
```

特点

精确，可带到线上，推荐使用。

注意

- 在上传数据到服务器时建议根据用户ID的尾号来抽样上报。
- onWindowFocusChanged只是首帧时间，App启动完成的结束点应该是真实数据展示出来的时候，如列表第一条数据展示，记得使用getViewTreeObserver().addOnPreDrawListener()，它会把任务延迟到列表显示后再执行。

AOP(Aspect Oriented Programming)打点

面向切面编程，通过预编译和运行期动态代理实现程序功能统一维护的一种技术。

作用

利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合性降低，提高程序的可重用性，同时大大提高了开发效率。

AOP核心概念

1、横切关注点
对哪些方法进行拦截，拦截后怎么处理。
2、切面（Aspect）
类是对物体特征的抽象，切面就是对横切关注点的抽象。
3、连接点（JoinPoint）
被拦截到的点（方法、字段、构造器）。
4、切入点（PointCut）
对JoinPoint进行拦截的定义。
5、通知（Advice）
拦截到JoinPoint后要执行的代码，分为前置、后置、环绕三种类型。

准备

首先，为了在Android使用AOP埋点需要引入AspectJ，在项目根目录的build.gradle下加入：

```
classpath 'com.hujiang.aspectjx:gradle-android-plugin- aspectjx:2.0.0'
```

然后，在app目录下的build.gradle下加入：

```
apply plugin: 'android-aspectjx'
implement 'org.aspectj:aspectjrt:1.8.+'
```

AOP埋点实战

JoinPoint一般定位在如下位置

- 1、函数调用
- 2、获取、设置变量
- 3、类初始化

使用PointCut对我们指定的连接点进行拦截，通过Advice，就可以拦截到JoinPoint后要执行的代码。

Advice通常有以下几种类型：

- 1、Before：PointCut之前执行
- 2、After：PointCut之后执行
- 3、Around：PointCut之前、之后分别执行

首先，我们举一个小栗子：

```
@Before("execution(* android.app.Activity.on**(..))")
public void onActivityCalled(JoinPoint joinPoint) throws Throwable {
...
}
```

在execution中的是一个匹配规则，第一个*代表匹配任意的方法返回值，后面的语法代码匹配所有Activity中on开头的方法。

处理Join Point的类型：

- 1、call：插入在函数体里面
- 2、execution：插入在函数体外面

如何统计Application中的所有方法耗时?

```
@Aspect
public class ApplicationAop {
    @Around("call (* com.json.chao.application.BaseApplication.**(..))")
    public void getTime(ProceedingJoinPoint joinPoint) {
    Signature signature = joinPoint.getSignature();
    String name = signature.toShortString();
    long time = System.currentTimeMillis();
    try {
```

```
 9          joinPoint.proceed();
10      } catch (Throwable throwable) {
11          throwable.printStackTrace();
12      }
13      Log.i(TAG, name + " cost" +     (System.currentTimeMillis() - ti
    me));
14      }
15  }
```

注意

当Action为Before、After时，方法入参为JoinPoint。

当Action为Around时，方法入参为ProceedingPoint。

Around和Before、After的最大区别:

ProceedingPoint不同于JoinPoint，其提供了proceed方法执行目标方法。

总结AOP特性：

- 1、无侵入性
- 2、修改方便

4、启动速度分析工具 — TraceView

使用方式

- 1、代码中添加：Debug.startMethodTracing()、检测方法、Debug.stopMethodTracing()。（需要使用adb pull将生成的**.trace文件导出到电脑，然后使用Android Studio的Profiler加载）
- 2、打开Profiler -> CPU -> 点击 Record -> 点击 Stop -> 查看Profiler下方Top Down/Bottom Up 区域找出耗时的热点方法。

Profile CPU

1、Trace types

Trace Java Methods

会记录每个方法的时间、CPU信息。对运行时性能影响较大。

Sample Java Methods

相比于Trace Java Methods会记录每个方法的时间、CPU信息，它会在应用的Java代码执行期间频繁捕获应用的调用堆栈，对运行时性能的影响比较小，能够记录更大的数据区域。

Sample C/C++ Functions

需部署到Android 8.0及以上设备，内部使用simpleperf跟踪应用的native代码，也可以命令行使用simpleperf。

Trace System Calls

- 检查应用与系统资源的交互情况。
- 查看所有核心的CPU瓶。
- 内部采用systrace，也可以使用systrace命令。

2、Event timeline

显示应用程序中在其生命周期中转换不同状态的活动，如用户交互、屏幕旋转事件等。

3、CPU timeline

显示应用程序实时CPU使用率、其它进程实时CPU使用率、应用程序使用的线程总数。

4、Thread activity timeline

列出应用程序进程中的每个线程，并使用了不同的颜色在其时间轴上指示其活动。

- 绿色：线程处于活动状态或准备好使用CPU。
- 黄色：线程正等待IO操作。（重要）
- 灰色：线程正在睡眠，不消耗CPU时间。

Profile提供的检查跟踪数据窗口有四种

1、Call Chart

提供函数跟踪数据的图形表示形式。

- 水平轴：表示调用的时间段和时间。
- 垂直轴：显示被调用方。
- 橙色：系统API。
- 绿色：应用自有方法
- 蓝色：第三方API（包括Java API）

提示：右键点击Jump to source跳转至指定函数。

2、Flame Chart

将具有相同调用方顺序的完全相同的方法收集起来。

- 水平轴：执行每个方法的相对时间量。
- 垂直轴：显示被调用方。

注意：看顶层的哪个函数占据的宽度最大（平顶），可能存在性能问题。

3、Top Down

- 递归调用列表，提供self、children、total时间和比率来表示被调用的函数信息。
- Flame Chart是Top Down列表数据的图形化。

4、Bottom Up

- 展开函数会显示其调用方。
- 按照消耗CPU时间由多到少的顺序对函数排序。

注意点：

- Wall Clock Time：程序执行时间。
- Thread Time：CPU执行的时间。

TraceView小结

特点

- 1、图形的形式展示执行时间、调用栈等。
- 2、信息全面，包含所有线程。

- 3、运行时开销严重，整体都会变慢，得出的结果并不真实。
- 4、找到最耗费时间的路径：Flame Chart、Top Down。
- 5、找到最耗费时间的节点：Bottom Up。

作用

主要做热点分析，得到两种数据：

- 单次执行最耗时的方法。
- 执行次数最多的方法。

5、启动速度分析工具 — Systrace

使用方式：代码插桩

定义Trace静态工厂类，将Trace.begainSection()，Trace.endSection()封装成i、o方法，然后再在想要分析的方法前后进行插桩即可。
在命令行下执行systrace.py脚本：

```
python /Users/quchao/Library/Android/sdk/platform-tools/systrace/systrace.py -t 20 sched gfx view wm am app webview -a "com.wanandroid.json.chao" -o ~/Documents/open-project/systrace_data/wanandroid_start_1.html
```

具体参数含义如下：

- -t：指定统计时间为20s。
- shced：cpu调度信息。
- gfx：图形信息。
- view：视图。
- wm：窗口管理。
- am：活动管理。
- app：应用信息。
- webview：webview信息。
- -a：指定目标应用程序的包名。
- -o：生成的systrace.html文件。

如何查看数据?

在UIThread一栏可以看到核心的系统方法时间区域和我们自己使用代码插桩捕获的方法时间区域。

Systrace原理

- 在系统的一些关键链路（如SystemServcie、虚拟机、Binder驱动）插入一些信息（Label）；
- 通过Label的开始和结束来确定某个核心过程的执行时间;
  把这些Label信息收集起来得到系统关键路径的运行时间信息，最后得到整个系统的运行性能信息;
- Android Framework里面一些重要的模块都插入了label信息，用户App中可以添加自定义的Lable。

Systrace小结

特性

- 结合Android内核的数据，生成Html报告。
- 系统版本越高，Android Framework中添加的系统可用Label就越多，能够支持和分析的系统模块也就越多。
- 必须手动缩小范围，会帮助你加速收敛问题的分析过程，进而快速地定位和解决问题。

作用

- 主要用于分析绘制性能方面的问题。
- 分析系统关键方法和应用方法耗时。

6、启动监控

1、实验室监控：视频录制

- 80%绘制
- 图像识别

注意

覆盖高中低端机型不同的场景。

2、线上监控

目标

需要准确地统计启动耗时。

1、启动结束的统计时机

是否是使用界面显示且用户真正可以操作的时间作为启动结束时间。

2、启动时间扣除逻辑

闪屏、广告和新手引导这些时间都应该从启动时间里扣除。

3、启动排除逻辑

Broadcast、Server拉起，启动过程进入后台都需要排除统计。

4、使用什么指标来衡量启动速度的快慢?

平均启动时间的问题
一些体验很差的用户很可能被平均了。
建议的指标
1、快开慢开比
如2s快开比，5s慢开比，可以看到有多少比例的用户体验好，多少比例的用户比较糟糕。
2、90%用户的启动时间
如果90%用户的启动时间都小于5s，那么90%区间的启动耗时就是5s。

5、启动的类型有哪几种?

- 首次安装启动
- 覆盖安装启动
- 冷启动（指标）
- 热启动（反映程序的活跃或保活能力）

借鉴Facebook的profilo工具原理，对启动整个流程的耗时监控，在后台对不同的版本做自动化对比，监控新版本是否有新增耗时的函数。

四、启动优化常规方案

启动过程中的常见问题

- 点击图标很久都不响应：预览窗口被禁用或设置为透明。
- 首页显示太慢：初始化任务太多。
- 首页显示后无法进行操作：太多延迟初始化任务占用主线程CPU时间片。

优化区域

Application、Activity创建以及回调等过程。

1、主题切换

使用Activity的windowBackground主题属性预先设置一个启动图片（layer-list），在启动后，在Activity的onCreate()方法中的super.onCreate()前再setTheme(R.style.AppTheme)。

优点

- 使用简单。
- 避免了启动白屏和点击启动图标不响应的情况。

缺点

- 治标不治本，表面上产生一种快的感觉。
- 对于中低端机，总的闪屏时间会更长，建议只在Android6.0/7.0以上才启用“预览闪屏”方案，让手机性能好的用户可以有更好的体验。

2、第三方库懒加载

按需初始化，如图片库的初始化等等。

3、异步初始化预备知识-线程优化

1、Android线程调度原理剖析

线程调度原理

- 任意时刻，只有一个线程占用CPU，处于运行状态。
- 多线程并发，轮流获取CPU使用权。
- JVM负责线程调度，按照特定机制分配CPU使用权。

线程调度模型

1、分时调度模型

轮流获取、均分CPU。

2、抢占式调度模型

优先级高的获取。

如何干预线程调度?

设置线程优先级。

Android线程调度

1、nice值

- Process中定义。
- 值越小，优先级越高。
- 默认是THREAD_PRIORITY_DEFAUT，0。

2、cgroup

- 更严格的群组调度策略。
- 后台group。
- 前台group，保证前台线程可以获取到更多的CPU

注意点

- 线程过多会导致CPU频繁切换，降低线程运行效率。
- 正确认识任务重要性决定哪种优先级。
- 优先级具有继承性。

2、Android异步方式

1、Thread

- 最简单、常见的异步方式。
- 不易复用，频繁创建及销毁开销大。
- 复杂场景不易使用。

2、HandlerThread

- 自带消息循环的线程。
- 串行执行。
- 长时间运行，不断从队列中获取任务。

3、IntentService

- 继承自Service在内部创建HandlerThread。
- 异步，不占用主线程。
- 优先级较高，不易被系统Kill。

4、AsyncTask

- Android提供的工具类。
- 无需自己处理线程切换。

- 需注意版本不一致问题（API 14以上解决）

5、线程池

- Java提供的线程池。
- 易复用，减少频繁创建、销毁的时间。
- 功能强大，如定时、任务队列、并发数控制等。

6、RxJava

由强大的Scheduler集合提供。
不同类型的Scheduler：

- IO
- Computation

异步方式总结

- 推荐度：从后往前排列。
- 正确场景选择正确的方式。

3、Android线程优化实战

线程使用准则

- 1、严禁使用new Thread方式。
- 2、提供基础线程池供各个业务线使用，避免各个业务线各自维护一套线程池，导致线程数过多。
- 3、根据任务类型选择合适的异步方式：优先级低，长时间执行，HandlerThread；定时执行耗时任务，线程池。
- 4、创建线程必须命名，以方便定位线程归属，在运行期Thread.currentThread().setName修改名字。
- 5、关键异步任务监控，注意异步不等于不耗时，建议使用AOP的方式来做监控。
- 6、重视优先级设置（根据任务具体情况），Process.setThreadPriority();可以设置多次。

4、如何锁定线程创建者

锁定线程创建背景

- 项目变大之后收敛线程。
- 项目源码、三方库、aar中都有线程的创建。

锁定线程创建方案

特别适合Hook手段，找Hook点：构造函数或者特定方法，如Thread的构造函数。

实战

在attachBaseContext中调用DexposedBridge.hookAllConstructors方法，如下所示：

```
1 DexposedBridge.hookAllConstructors(Thread.class, new XC_MethodHook()
  {
2     @Override protected void afterHookedMethod (MethodHookParam para
```

```
m) throws Throwable {
        super.afterHookedMethod(param);
        Thread thread = (Thread) param.thisObject;
        LogUtils.i("stack " + Log.getStackTraceString(new Throwable
()));
    }
);
```

从log找到线程创建信息，根据堆栈信息跟相关业务方沟通解决方案。

5、线程收敛优雅实践初步

线程收敛常规方案

- 根据线程创建堆栈考量合理性，使用同一线程库。
- 各业务线下掉自己的线程库。

问题：基础库怎么使用线程?

直接依赖线程库。

缺点：

- 线程库更新可能会导致基础库更新。

基础库优雅使用线程

- 基础库内部暴露API：setExecutor。
- 初始化的时候注入统一的线程库。

统一线程库时区分任务类型

- IO密集型：IO密集型任务不消耗CPU，核心池可以很大。
- CPU密集型：核心池大小和CPU核心数相关。

6、线程优化核心问题

1、线程使用为什么会遇到问题?

项目发展阶段忽视基础设施建设，没有采用统一的线程池，导致线程数量过多。

表现形式：

异步任务执行太耗时，导致主线程卡顿。

问题原因：

- Java线程调度是抢占式的，线程优先级比较重要，需要区分。
- 没有区分IO和CPU密集型任务，导致主线程抢不到CPU。

2、怎么在项目中对线程进行优化?

核心：线程收敛

- 通过Hook方式找到对应线程的堆栈信息，和业务方讨论是否应该单独起一个线程，尽可能使用统一线程池。

- 每个基础库都暴露一个设置线程池的方法，以避免线程库更新导致基础库需要更新的问题。
- 统一线程池应注意IO、CPU密集型任务区分
- 其它细节：重要异步任务统计耗时、注重异步任务优先级和线程名的设置。

4、异步初始化

1、核心思想

子线程分担主线程任务，并行减少时间。

2、异步优化注意点

- 1、不符合异步要求。
- 2、需要在某个阶段完成（采用CountDownLatch确保异步任务完成后才到下一个阶段）。
- 3、如出现主线程要使用时还没初始化则在此次使用前初始化。
- 4、区分CPU密集型和IO密集型任务。

3、异步初始化方案演进

- 1、new Thread
- 2、IntentService
- 3、线程池（合理配置并选择CPU密集型和IO密集型线程池）
- 4、异步启动器

4、异步优化最优解：异步启动器

常规异步优化痛点：

- 1、代码不优雅。
- 2、场景不好处理（依赖关系）。
- 3、维护成本高。

核心思想：

充分利用CPU多核，自动梳理任务顺序。

启动器流程

- 任务Task化，启动逻辑抽象成Task。
- 根据所有任务依赖关系排序生成一个有向无环图。
- 多线程按照排序后的优先级依次执行。

5、延迟初始化

1、常规方案：利用闪屏页的停留时间进行部分初始化

- new Handler().postDelayed()。
- 界面UI展示后调用。

2、常规初始化痛点

- 时机不容易控制。

- 导致界面UI卡顿。

3、延迟优化最优解：延迟启动器

核心思想

利用IdleHandler特性，在CPU空闲时执行，对延迟任务进行分批初始化。

优势

- 执行时机明确。
- 缓解界面UI卡顿。
- 真正提升用户体验。

6、Multidex预加载优化

1、优化步骤：

- 1、启动时单独开一个进程去异步进行Multidex的第一次加载，即Dex提取和Dexopt操作。
- 2、此时，主进程Application进入while循环，不断检测Multidex操作是否完成。
- 3、执行到Multidex时，则已经发现提取并优化好了Dex，直接执行。MultiDex执行完之后主进程Application继续执行ContentProvider初始化和Application的onCreate方法。

注意：

5.0以上默认使用ART，在安装时已将Class.dex转换为oat文件了，无需优化，所以应判断只有在主进程及SDK 5.0以下才进行Multidex的预加载。

2、dex-opt过程是怎样的?

主要包括inline以及quick指令的优化。

inline是什么?

使编译器在函数调用处用函数体代码代替函数调用指令。

inline的作用?

函数调用的转移操作有一定的时间和空间方面的开销，特别是对于一些函数体不大且频繁调用的函数，解决其效率问题更为重要，引入inline函数就是为了解决这一问题。

inline是如何进行优化的?

inline函数至少在三个方面提升了程序的时间性能：

- 1、避免了函数调用必须执行的压栈出栈等操作。
- 2、由于函数体代码被移到函数调用处，编译器可以获得更多的上下文信息，并根据这些信息对函数体代码和被调用者代码进行更进一步的优化。
- 3、若不使用inline函数，程序执行至函数调用处，需要转而去执行函数体所在位置的代码。一般函数调用位置和函数代码所在位置在代码段中并不相近，这样很容易形成操作系统的缺页中断。操作系统需要把缺页地址的代码从硬盘移入内存，所需时间将成数量级增加。而使用inline函数则可以减少缺页中断发生的机会。

inline使用时应注意的问题?

- 由于inline函数在函数调用处插入函数体代码代替函数调用，若该函数在程序的很多位置被调用，有可能造成内存空间的浪费。
- 一般程序的压栈出栈操作也需要一定的代码，这段代码完成栈指针调整、参数传递、现场保护和恢复等操作。
  若函数的函数体代码量小于编译器生成的函数压栈出栈代码，则可以放心地定义为inline，这个时候占用内存空间反而会减小。而当函数体代码大于函数压栈出栈代码时，将函数定义为inline就会增加内存空间的使用。
- C++程序应该根据应用的具体场景、函数体大小、调用位置多少、函数调用的频率、应用场景对时间性能的要求，应用场景对内存性能的要求等各方面因素合理决定是否定义inline函数。
- inline函数内不允许用循环语句和开关语句。

3、预加载SharedPreferences

可以利用MultiDex预加载期间的这段CPU去预加载SharedPreferences。

注意：

需重写getApplicationContext返回this，否则此时可能获取不到context。

7、类预加载优化

在Application中提前异步加载初始化耗时较长的类。

如何找到耗时较长的类?

替换系统的ClassLoader，打印类加载的时间，按需选取需要异步加载的类。

注意：

- Class.forName()只加载类本身及其静态变量的引用类。
- new 类实例 可以额外加载类成员变量的引用类。

8、WebView启动优化

- 1、WebView首次创建比较耗时，需要预先创建WebView提前将其内核初始化。
- 2、使用WebView缓存池，用到WebView的时候都从缓存池中拿，注意内存泄漏问题。
- 3、本地离线包，即预置静态页面资源。

9、页面数据预加载

在主页空闲时，将其它页面的数据加载好保存到内存或数据库，等到打开该页面时，判断已经预加载过，就直接从内存或数据库取数据并显示。

10、启动阶段不启动子进程

子进程会共享CPU资源，导致主进程CPU紧张。此外，在多进程情况下一定要可以在onCreate中去区分进程做一些初始化工作。

注意启动顺序：

App onCreate之前是ContentProvider初始化。

11、闪屏页与主页的绘制优化

- 1、布局优化。

- 2、过渡绘制优化。

关于绘制优化可以参考Android性能优化之绘制优化。

五、启动优化黑科技

1、启动阶段抑制GC

启动时CG抑制，允许堆一直增长，直到手动或OOM停止GC抑制。（空间换时间）

前提条件

- 1、设备厂商没有加密内存中的Dalvik库文件。
- 2、设备厂商没有改动Google的Dalvik源码。

实现原理

- 在源码级别找到抑制GC的修改方法，例如改变跳转分支。
- 在二进制代码里找到 A 分支条件跳转的”指令指纹”，以及用于改变分支的二进制代码，假设为override_A。
- 应用启动后扫描内存中的 libdvm.so，根据”指令指纹”定位到修改位置，然后用 override_A 覆盖。

缺点

白名单覆盖所有设备，但维护成本高。

2、CPU锁频

在Android系统中，CPU相关的信息存储在/sys/devices/system/cpu目录的文件中，通过对该目录下的特定文件进行写值，实现对CPU频率等状态信息的更改。

缺点

暴力拉伸CPU频率，导致耗电量增加。

CPU工作模式

- performance：最高性能模式，即使系统负载非常低，cpu也在最高频率下运行。
- powersave：省电模式，与performance模式相反，cpu始终在最低频率下运行。
- ondemand：CPU频率跟随系统负载进行变化。
- userspace：可以简单理解为自定义模式，在该模式下可以对频率进行设定。

CPU的工作频率范围

对应的文件有：

- cpuinfo_max_freq
- cpuinfo_min_freq
- scaling_max_freq
- scaling_min_freq

3、IO优化

- 1、启动过程不建议出现网络IO。

- 2、为了只解析启动过程中用到的数据，应选择合适的数据结构，如将ArrayMap改造成支持随机读写、延时解析的数据存储结构以替代SharePreference。

注意：考虑重度用户的使用场景。

Linux IO知识补充

1、磁盘高速缓存技术

利用内存中的存储空间来暂存从磁盘中读出的一系列盘块中的信息。因此，磁盘高速缓存在逻辑上属于磁盘，物理上则是驻留在内存中的盘块。

其内存中分为两种形式：

- 在内存中开辟一个单独的存储空间作为磁速缓存，大小固定。
- 把未利用的内存空间作为一个缓沖池，供请求分页系统和磁盘I/O时共享。

2、分页

- 存储器管理的一种技术。
- 可以使电脑的主存使用存储在辅助存储器中的数据。
- 操作系统会将辅助存储器（通常是磁盘）中的数据分区成固定大小的区块，称为“页”（pages）。当不需要时，将分页由主存（通常是内存）移到辅助存储器；当需要时，再将数据取回，加载主存中。
- 相对于分段，分页允许存储器存储于不连续的区块以维持文件系统的整齐。
- 分页是磁盘和内存间传输数据块的最小单位。

3、高速缓存/缓冲器

- 都是介于高速设备和低速设备之间。
- 高速缓存存放的是低速设备中某些数据的复制数据，而缓冲器则可同时存储高低速设备之间的数据。
- 高速缓存存放的是高速设备经常要访问的数据。

4、linux同步IO：sync、fsync、fdatasync

为什么要使用同步IO?

当数据写入文件时，内核通常先将该数据复制到缓冲区高速缓存或页面缓存中，如果该缓冲区尚未写满，则不会将其排入输入队列，而是等待其写满或内核需要重用该缓冲区以便存放其他磁盘块数据时，再将该缓冲排入输出队列，最后等待其到达队首时，才进行实际的IO操作—延迟写。

延迟写减少了磁盘读写次数，但是却降低了文件内容的更新速度，可能会造成文件更新内容的丢失。为了保证数据一致性，则需使用同步IO。

sync

- sync函数只是将所有修改过的块缓冲区排入写队列，然后就返回，它并不等待实际磁盘写操作结束再返回。
- 通常称为update的系统守护进程会周期性地（一般每隔30秒）调用sync函数。这就保证了定期冲洗内核的块缓冲区。

fsync

- fsync函数只对文件描述符filedes指定的单一文件起作用，并且等待磁盘IO写结束后再返回。通常应用于需要确保将修改内容立即写到磁盘的应用如数据库。
- 文件的数据和metadata通常存放在硬盘的不同地方，因此fsync至少需要两次IO操作。

补充：msync

如果当前硬盘的平均寻道时间是3-15ms，7200RPM硬盘的平均旋转延迟大约为4ms，因此一次IO操作的耗时大约为10ms。

如果使用内存映射文件的方式进行文件IO（mmap），将文件的page cache直接映射到进程的地址空间，这时需要使用msync系统调用确保修改的内容完全同步到硬盘之上。

fdatasync

- fdatasync函数类似于fsync，但它只影响文件的数据部分。而fsync还会同步更新文件的属性。
- 仅仅只在必要（如文件尺寸需要立即同步）的情况下才会同步metadata，因此可以减少一次IO操作。

日志文件都是追加性的，文件尺寸一致在增大，如何利用好fdatasync减少日志文件的同步开销?

创建每个log文件时先写文件的最后一个page，将log文件扩展为10MB大小，这样便可以使用fdatasync，每写10MB只有一次同步metadata的开销。

5、网络IO和磁盘IO

磁盘IO（缓存IO）

标准IO，大多数文件系统默认的IO操作。

- 数据先从磁盘复制到内核空间的缓冲区，然后再从内核空间中的缓冲区复制到应用程序的缓冲区。
- 读操作：操作系统检查内核的缓冲区有没有需要的数据，如果已经有缓存了，那么直接从缓存中返回；否则，从磁盘中返回，再缓存在操作系统的磁盘中。
- 写操作：将数据从用户空间复制到内核空间中的缓冲区中，这时对用户来说写操作就已经完成，至于什么时候写到磁盘中，由操作系统决定，除非显示地调用了sync同步命令。

优点

- 在一定程度上分离了内核空间和用户空间，保护系统本身安全。
- 可以减少磁盘IO的读写次数，从而提高性能。

缺点

DMA方式可以将数据直接从磁盘读到页缓存中，或者将数据从页缓存中写回到磁盘，而不能在应用程序地址空间和磁盘之间进行数据传输，这样，数据在传输过程中需要在应用程序地址空间（用户空间）和缓存（内核空间）中进行多次数据拷贝操作，这带来的CPU以及内存开销是非常大的。

磁盘IO主要的延时（15000RPM硬盘为例）

机械转动延时（平均2ms）+ 寻址延时（2~3ms）+ 块传输延时（0.1ms左右）=> 平均5ms

网络IO主要延时

服务器响应延时 + 带宽限制 + 网络延时 + 跳转路由延时 + 本地接收延时（一般为几十毫秒到几千毫秒，受环境影响极大）

6、PIO与DMA

PIO

很早之前，磁盘和内存之间的数据传输是需要CPU控制的，也就是读取磁盘文件到内存中时，数据会经过CPU存储转发，这种方式称为PIO。

DMA（直接内存访问，Direct Memory Access）

- 可以不经过CPU而直接进行磁盘和内存的数据交换。
- CPU只需要向DMA控制器下达指令，让DMA控制器来处理数据的传送即可。

- DMA控制器通过系统总线来传输数据，传送完毕再通知CPU，这样就在很大程度上降低了CPU占用率，大大节省了系统资源，而它的传输速度与PIO的差异并不明显，而这主要取决于慢速设备的速度。

7、直接IO与异步IO

直接IO

应用程序直接访问磁盘数据，而不经过内核缓冲区。以减少从内核缓冲区到用户数据缓存的数据复制。

异步IO

当访问数据的线程发出请求后，线程会接着去处理其它事情，而不是阻塞等待。

8、VFS（虚拟文件系统，Virtual File System）

可以为访问文件系统的系统调用提供一个统一的抽象接口。

4、数据重排

Dex文件用到的类和APK里面各种资源文件都比较小，读取频繁，且磁盘地址分布范围比较广。我们可以利用Linux文件IO流程中的page cache机制将它们按照读取顺序重新排列在一起，以减少真实的磁盘IO次数。

1、类重排

使用Facebook的ReDex的Interdex调整类在Dex中的排列顺序。

2、资源文件重排

- 最佳方案是修改内核源码，实现统计、度量、自动化。
- 其次可以使用Hook框架进行统计得出资源加载顺序列表。
- 最后，调整apk文件列表需要修改7zip源码以支持传入文件列表顺序。

技术视野：

- 所谓的创新，不一定是要创造前所未有的东西，也可以将已有的方案移植到新的平台，并结合该平台的特性落地，就是一个很大的创新。
- 当我们足够熟悉底层的知识时，可以利用系统的特性去做更加深层次的优化。

3、Hook框架

Xposed框架是什么？

一个可以不修改APK就影响程序运行的Hook框架。

原理

用自身实现的app_process替换掉系统/system/bin/app_process，加载一个额外的XposedBridge的jar包，用于将入口osZygoteInit.main()替换成XposedBridge.main()。之后，创建的Zygote进程和其子进程都是Hook过的了。

使用具体细节参见Xposed教程。

5、类加载优化（Dalvik）

1、类预加载原理

对象第一次创建的时候，JVM首先检查对应的Class对象是否已经加载。如果没有加载，JVM会根据类名查找.class文件，将其Class对象载入。同一个类第二次new的时候就不需要加载类对象，而是直接实例化，创建时间就缩短了。

2、类加载优化过程

- 在Dalvik VM加载类的时候会有一个类校验过程，它需要校验方法的每一个指令。
- 通过Hook去掉verify步骤 -> 几十ms的优化
- 最大优化场景在于首次安装和覆盖安装时，在Dalvik平台上，一个2MB的Dex正常需要350ms，将classVerifyMode设为VERIFY_MODE_NONE后，只需150ms，节省超过50%时间。

ART比较复杂，Hook需要兼容几个版本。而且在安装时，大部分Dex已经优化好了，去掉ART平台的verify只会对动态加载的Dex带来一些好处。所以暂时不建议在ART平台使用。

3、延伸：插件化和热修复

在设计上都存在大量的Hook和私有API调用

缺点：

1、稳定性较差

由于厂商的兼容性、安装失败、ART加载时dex2oat失败等原因，还是会有一些代码和资源的异常。Android P推出的non-sdk-interface调用限制，以后适配只会越来越难，成本越来越高。

2、性能问题

用到一些黑科技导致底层Runtime的优化享受不到。如Tinker加载补丁后，启动速度会降低5%~10%。

1、各项热补丁技术的优缺点

缺点：

- 只针对单一客户端版本，随着版本差异变大补丁体积也会变大。
- 不支持所有修改，如AndroidManifest。
- 对代码和资源的更新成功率无法达到100%。

优点：

- 降低开发成本，轻量而快速地升级。发布补丁等同于发布版本，也应该完整地执行测试与上线流程。
- 远端调试，只为特定用户发送补丁。
- 数据统计，对同一批用户更换补丁版本，能够更好地进行ABTest，得到更精确的数据。

2、InstanceRun实现机制

Android官方使用热补丁技术实现InstantRun。

应用构建流程：

构建 -> 部署 -> 安装 -> 重启app -> 重启activity

实现目标：

尽可能多的剔除不必要的步骤，然后提升必要步骤的速度。

InstantRun构建的三种方式：

1、HotSwap

增量构建 -> 改变部署

场景：

适用于多数简单的改变（包括一些方法实现的修改，或者变量值修改）。

2、Warm Swap

增量构建 -> 改变部署 -> activity重启

场景：

一般是修改了resources。

3、Cold Swap

增量构建 -> 改变部署 -> 应用重启 -> activity重启

场景：

涉及结构性变化，如修改了继承规则或方法签名。

补充：apk打包流程

manifest文件合并、打包，和res一起被AAPT合并到APK中，同时项目代码被编译成字节码，然后转换成.dex文件，也被合并到APK中。

Android打包流程回顾，最后对于release签名apk需要进行zipalign优化，它是指什么？

概念补充：内存对齐（DSA，Data Structure Alignment）

各种类型的数据按照一定的规则在内存空间上排列，这就是对齐。

优势：

基于效率考虑，空间换时间，减少数据存取指令周期。

编译器内存字节对齐的原则是什么？

- 1、数据类型的自身对齐值就是其长度（64位 OS）。
- 2、结构体或类的自身对齐值就是成员中自身对齐值最大的那个。需要起始地址必须是其相应有效对齐值的整数，并要求结构体的大小也为该结构体有效对齐值的整数倍。

zipalign优化的最根本目的是**帮助操作系统更高效地根据请求索引资源，使用resource-handling code统一将DSA限定为4byte。**

手动执行Align优化

利用build-tools文件夹下对应Android版本中的zipalign工具：

```
zipalign -v 4 source.apk androidres.apk
```

检查当前APK是否已经执行过Align优化：

```
zipalign -c -v 4 androidres.apk
```

其中：

- -c：检查。
- -v：代表详细输出。
- 4：代表对齐为4个字节。

首次运行Instant Run，Gradle执行的操作

- 在有Instant Run的环境下：一个新的App Server会被注入到App中，与Bytecode instrumentation协同监控代码的变化。
- 同时会有一个新的Application类，它注入了一个自定义类加载器。同时该Application会启动我们所需的新注入的App Server。于是，AndroidManifest会被修改来确保我们能使用这个新的Application。

- 使用的时候，它会通过决策，合理运用冷温热拔插来协助我们大量地缩短构建程序的时间。

HotSwap原理

Android Studio monitors

运行着Gradle任务来生成增量.dex文件（dex对应着开发中的修改类），AS会提取这些.dex文件发送到App Server，然后部署到App。因为原来版本的类都装载在运行中的程序了，Gradle会解释更新好这些.dex文件，发送到App Server的时候，交给自定义的类加载器来加载.dex文件。

App Server会不断地监听是否需要重写类文件，如果需要，任务会被立马执行，新的更改便能立即被响应。

注意：

此时Instant Run是不能回退的，必须重启应用响应修改。

WarmSwap原理

因为资源文件是在Activity创建时加载，所以必须重启Activity加载资源文件。

注意：

AndroidManifest的值是在APK安装的时候被读取的，所以需要触发一个完整的应用构建和部署。

ColdSwap原理

应用部署的时候，会把工程拆分成十个部分，每个部分都拥有自己的.dex文件，然后所有的类会根据包名被分配给相应的.dex文件。当ColdSwap开启时，修改过的类所对应的的.dex文件，会重组生成新的.dex文件，然后再部署到设备上。

注意：

应用多进程会被降级为ColdSwap。

3、AndFix

实现原理：

native hook –> dalvik_repleaceMethod –> 无法支持新增或删除filed的情况 –> 需修复特定问题

优点：

- 立即生效
- 补丁较小

缺点：

- 兼容性不佳
- 开发不透明

4、Qzone

基于Android Dex分包方案。

将多个dex文件放入到app的classloader中，但是android dex拆包方案中的类是没有重复的，如果classes.dex和classes1.dex中有重复的类，**当用到这个重复的类时，系统会选择哪个类进行加载呢？**

一个ClassLoader可以包含多个dex文件，每个dex文件是一个Elements，多个dex文件排列成有序的dexElements，当找类的时候，会按顺序遍历dex文件，然后从当前遍历的dex文件中找类，如果找到则返回，如果找不到从下一个dex文件继续查找。

所以，如果在不同的dex中有相同的类存在，那么会优先选择排在前面的dex文件的类。

Qzone热补丁方案就是把有问题的类打包到一个dex（patch.dex）中去，然后把这个dex插入到Elements的最前面。

实现中遇到的问题

当其它dex文件中的类引用了patch.dex中的类时，会出现校验错误。拆分dex的很多类都不是在同一个dex内的，怎么没有问题?

因为这个校验有个前提，当引用类被打上了CLASS_ISPREVERIFIED标志，那么就会进行dex的校验。

CLASS_ISPREVERIFIED标志是什么时候被打上去的?

- 在dex转换成odex（dexopt过程）时，当apk在安装的时候，apk中的classes.dex会被虚拟机（dexopt）优化成odex文件，然后才会拿去执行。
- 虚拟机在启动的时候，会有许多的启动参数，其中一项就是verify选项，当verify选项被打开时，doVerify变量为true，那么就会执行dvmVerifyClass进行类的校验，如果校验成功，这个类会被打上CLASS_ISPREVERIFIED标志。

具体的校验过程是怎么样的?

有两步验证：

1、验证clazz -> directMethods方法，其包含以下方法：

- static方法
- private方法
- 构造函数

2、clazz -> virtualMethods

- 虚函数 = override方法

如果以上方法中直接引用到的类（第一层级关系，不会进行递归搜索）和clazz**都在同一个dex中**的话，那么这个类就会被打上CLASS_ISPREVERIFED标志。

为了解决补丁方案中遇到的问题，所以必须从这些方法中入手，防止类被打上CLASS_ISPREVERIFIED标志。空间的方案是往所有类的构造函数里面插入一段代码：

```
If (ClassVerifier.PREVENT_VERIFY) {
    System.out.println(AntilazyLoad.class);
}
```

其中AntilazyLoad类会被打包成单独的hack.dex，这样当安装apk的时候，classes.dex中的类都会引用一个在不同dex中的AntilazyLoad类，这样就防止类被打上了CLASS_ISPREVERIFILED标志，只要没被打上这个标志的类都可以进行打补丁操作。

注意：

- 1、在应用启动进行加载时，AntilazyLoad类所在的dex包必须先加载进来，不然AntilazyLoad类会被标记为不存在，即使后续加载了hack.dex包，那么它也是不存在的。
- 2、当在Application的onCreate中加载hack.dex时，Application不能插入上述代码。

为什么要选择构造函数?

因为他不增加方法数，一个类即使没有显示的构造函数，也有一个隐式的默认构造函数。

如何更高效地插入上述代码?

可以使用ASM/javaassist库来进行字节码插入代码。

Art的处理

Art采用了新的方式，插桩对代码的执行效率没有影响。但是补丁中的类出现修改类变量或者方法，可能会导致出现内存地址错乱的情况。

原因：

dex2oat时fast*已经将类能确定的各个地址写死。如果运行时补丁包的地址出现改变，原始类去调用时就会出现地址错乱。

解决方法：

将其父类以及调用类的所有类都加入到补丁包中。

虚拟机在安装期间为类打上CLASS_ISPREVERIFIED标志是为了什么?

为了提高性能。

禁用CLASS_ISPREVERIFIED是否会影响APP的性能?

由于现在很多App都使用了MultiDex分包方案，这导致了很多类都没有被打上这个标志，所以此时禁用所有类打上CLASS_ISPREVERIFIED标志对性能的影响不是很大。

如何有效地生成补丁包?

- 1、在正式版本发布的时候，会生成一份缓存文件，里面记录了所有class文件的MD5值，还有一份mapping混淆文件。
- 2、在后续的版本中使用–applaymapping选项，应用正式版本的mapping文件，然后计算编译完成的class文件的MD5和正式版本进行比较，把不相同的class文件打包成补丁包。

Qzone方案缺点：

在补丁包大小与性能损耗上有一定的局限性。

5、补充：ASM字节码插桩

插桩就是将一段代码插入或者替换原本的代码。

字节码插桩就是在我们的代码编译成字节码（Class）后，在Android下生成dex之前修改Class文件，修改或者增强原有代码逻辑的操作。

原理：

利用注解来标记需要插桩的方法，结合编译后操作字节码来帮助我们自动插入，当不需要时，关闭插桩即可。

除了Javassist框架外，还有一个应用更为广泛的ASM框架同样也是字节码操作框架，Instant Run包括AspectJ就是借助ASM来实现各自的功能。

JSON对于GSON就类似于字节码Class对于Javassist/ASM。

Android ASM自动埋点方案实践

原理：

Android 1.5.0版本以后提供了Transform API，允许第三方Plugin在打包dex文件之前的编译过程中操作.class文件，我们做的就是实现Transform进行.class文件遍历拿到所有方法，修改完成后对文件进行替换。

流程：

1、自动埋点追踪，遍历所有文件更换字节码

```
1 AutoTransform -> transform -> inputs.each {TransformInput input -> in
```

```
put.jarInput.each { JarInput jarInput -> … } input.directoryInputs.ea
ch { DirectoryInput directoryInput -> … }}
```

2、Gradle插件实现

```
1 PluginEntry -> apply -> def android = project.extensions.getByType(Ap
  pExtension)
2 registerTransform(android) -> AutoTransform transform = new AutoTrans
  form
3 android.registerTransform(transform)
```

3、使用ASM进行字节码编写

ASM框架核心类

- ClassReader：读取编译后的.class文件。
- ClassWriter：重新构建编译后的类。
- ClassVisitor：拜访类成员信息。
- AdviceAdapter：实现MethodVisitor接口，拜访方法的信息。

1、visit -> 在ClassVisitor中根据判断是否是实现View$OnClickListener接口的类，只有满足条件的类才会遍历其中的方法进行操作。

2、在MethodVisitor中对该方法进行修改

```
1 visitAnnotation -> onMethodEnter -> onMethodExit
```

3、先在java文件中编写要插入的代码，然后使用ASM插件查看对应的字节码，根据其用ASM提供的Api一一对应地把代码填进来。

6、Tinker

原理：

- 全量替换新的Dex
- 在编译时通过新旧两个Dex生成差异patch.dex。在运行时，将差异patch.dex重新跟原始安装包的旧Dex还原为新的Dex。由于比较耗费时间与内存，放在后台进程:patch中，为了补丁包尽可能小，微信自研了DexDiff算法，它深度利用Dex的格式来减少差异的大小。

DexDiff的粒度是Dex格式的每一项，BsDiff的粒度是文件，AndFix/Qzone的粒度为class。

缺点：

- 1、占用Rom体积，1.5倍所修改Dex大小 = Dex.jar + dexopt文件。
- 2、一个额外的合成过程，合成时间长短和额外的内存消耗也会影响最终的成功率。

热补丁方案对比

若不care性能损耗与补丁包大小，Qzone是最简单且成功率最高的方案。

7、完善的热补丁系统构建

一、网络通道

负责将补丁包交付给用户，包括特定用户和全量用户。

1、pull通道
在登录/24小时等时机，通过pull方式查询后台是否有对应的补丁包更新。
2、指定版本的push通道
在紧急情况下，我们可以在一个小时内向所有用户下发补丁包更新。
3、指定特定用户的push通道
对特定用户或用户组做远程调试。
二、上线与管理平台
快速上线，管理历史记录，以及监控补丁的运行情况。

6、保活

1、厂商合作

2、微信Hardcoder

构建了App与系统（ROM）之间可靠的通信框架，让系统知道App的需求。

原理：

- 让App跨过Framework直接跟厂商ROM通信。
- 分为Client端和Server端，Server端由厂商系统侧自行实现。
- 它们直接采用LocalSocket方式，Hardcoder是Native实现的，使用了Linux的Socket接口实现了一套自己的LocalSocket。

性能提升有多少?

平均10%~30%。

3、OPPO Hyper Boost加速引擎

一种优化资源调度的技术。

原理：

让应用程序与系统资源实现实时”双向对话”。当来自应用和游戏程序的不同场景和用户行为被Hyper Boost识别后，手机会智能地匹配到合理的系统资源，让手机SoC的CPU、GPU、ISP、DSP提供的运算资源更加合理地利用，从而让用户使用手机更加流畅。

六、总结

1、优化总方针

- 异步、延迟、懒加载
- 技术、业务相结合

2、注意事项

1、cpu time和wall time

- wall time（代码执行时间）与cpu time（代码消耗CPU时间），锁冲突会造成两者时间差距过大。

- cpu time才是优化方向，应尽力按照systrace的cpu time和wall time跑满cpu。

2、监控的完善

- 线上监控多阶段时间（App、Activity、生命周期间隔时间）。
- 处理聚合看趋势。
- 收敛启动代码修改权限。
- 结合CI修改启动代码需要Review通知。

3、常见问题

1、启动优化是怎么做的?

- 分析现状、确认问题
- 针对性优化（先概括，引导其深入）
- 长期保持优化效果

2、是怎么异步的，异步遇到问题没有?

- 体现演进过程
- 详细介绍启动器

3、启动优化有哪些容易忽略的注意点?

- cpu time与wall time
- 注意延迟初始化的优化
- 介绍下黑科技

4、版本迭代导致的启动变慢有好的解决方式吗?

- 启动器
- 结合CI
- 监控完善

至此，探索Android启动速度优化的旅途也应该告一段落了，如果你耐心读到最后的话，会发现要想极致地提升App的性能，需要有一定的广度，如我们**引入了始于后端的AOP编程来实现无侵入式的函数插桩**，也需要有一定的深度，从前面的探索之旅来看，**我们先后涉及了Framework层、Native层、Dalvik虚拟机、甚至是Linux IO和文件系统相关的原理**。因此，我想说，Android开发并不简单，即使是App层面的性能优化这一知识体系，也是需要我们不断地加深自身知识的深度和广度。

参考链接：

1、Android开发高手课之启动优化
2、支付宝客户端架构解析：Android
客户端启动速度优化之「垃圾回收」
3、支付宝 App 构建优化解析：通过安装包重排布优化 Android 端启动性能
4、Facebook Redex字节码优化工具
5、微信Android热补丁实践演进之路

6、安卓App热补丁动态修复技术介绍
7、Dalvik Optimization and Verification With dexopt
8、微信在Github开源了Hardcoder，对Android开发者有什么影响?
9、历时三年研发，OPPO 的 Hyper Boost 引擎如何对系统、游戏和应用实现加速?
10、抱歉，Xposed真的可以为所欲为
11、墙上时钟时间 ，用户cpu时间 ，系统cpu时间的理解
12、《Android应用性能优化最佳实践》
13、必知必会 | Android 性能优化的方面方面都在这儿
14、极客时间之Top团队大牛带你玩转Android性能分析与优化
15、启动器源码
16、MultiDex优化源码
17、使用gradle自动化增加Trace Tag