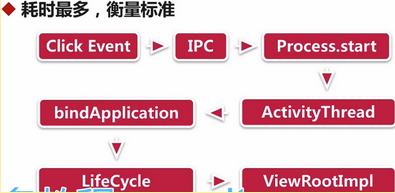
# **启动优化**

## **1.启动的三种方式**

### （1）冷启动



优化方向: Application和Activity生命周期

### （2）热启动

最快，后台切到前台

### （3）温启动

较快LifeCycle

## **2.启动时间测量方式**

### 1.adb命令

**adb shell am start -W packagename/packagename.MainActivity  
然后运行后出现三个时间  
ThisTime:最后一个Activity启动耗时  
TotalTime:所有Activity启动耗时**

**说明：如果有启动页的话就是启动页和主页面启动的总耗时  
WaitTime:AMS启动Activity的总耗时**

**总结：**

* **时间耗时排序**

**ThisTime< TotalTime< WaitTime**

* **优缺点**
* **线下使用方便，不能带到线上**
* **非严谨、精确时间**

### 手动打点的方式-->（启动时埋点，启动结束埋点，二者差值） 1.开始时间为 application中执行attachBaseContext()方法为第一步 2.结束时间为 Feed的第一条展示的时间也就是Activity显示第一条数据的时间 （误区：认为 onWindowFocusChange这个方法是activity第一帧）

总结：

1. 精确可带到线上，推荐使用
2. 避开误区采用Feed第一条展示

## **3.工具类使用**

### 1.traceView

（1）traceView优缺点

优势

* 图形的形式展示执行时间，调用栈等
* 信息全面包含所有线程

缺点

* 运行时开销严重，整体会变慢
* 可能会带偏优化方向

（2）使用方式

Debug.startMethodTracing("")  
Debug.startMethodTracing()

生成文件在SD卡：Android /data/packagename/files



重点关注：call Chart 和 TopDown

### 2.systrace

（1）特点

* 结合Android内核的数据，生成html报告
* API 18以上使用，推荐TraceCompat

（2）优点

轻量级开销小，直观反映cpu利用率

（3）使用

方式一

1. **进入Android/Sdk/platform-tools/systrace目录下**
2. **可以配置一些参数，类似于通过Android Device Monitor抓取时步骤2配置的显示信息，若不选择则默认全部抓取**

**-o： 指定文件输出位置和文件名**

**-t： 抓取systrace的时间长度**

**-a： 指定特殊进程包名（自己加Label时必须加上）**

**eg: python systrace.py -t 10 -o /Users/Downloads/boot.html -a gaosi.com.learn**

1. **Chrome浏览器（必须）。在地址栏输入**[**chrome://tracing**](chrome://tracing/)**命令，然后将生成的trace.html文件拖进来，或者通过load按钮导入**

方式二、

TraceCompat.beginSection("beginSection")

TraceCompat.endSection()

（4）cputime和walltime区别

cputime：代码消耗cpu的时间（重要指标）举例：锁冲突

walltime：代码执行时间

## **AOP优雅获取方法执行耗时**

### (1) AOP特点

* 针对同一类问题进行处理
* 无侵入添加代码

### (2)Aspectj使用

1.添加依赖

classpath 'com.hujiang.aspectjx:gradle-android-plugin-aspectjx:2.0.10'

apply plugin: 'android-aspectjx'

implementation 'org.aspectj:aspectjrt:1.9.5'

2.JoinPoint🡪可以作为切面的地方

* 函数调用、执行
* 获取设置变量
* 类初始化

3.PointCut->带条件的JoinPoint

4.Advice->要插入代码的位置

* Before：PointCut之前执行
* After：PointCut之后执行
* Around：PointCut之前、之后都执行

语法：详解网址：https://www.jianshu.com/p/d32a2453786e

**@Before("execution(\* android.app.Activity+.on\*\*(..))")**

**public void** **onActivityCalled (JoinPoint joinPoint) throws Throwable {**

**...**

**}**

* Before：Advice具体插入位置
* execution:处理JoinPoint的类型

JoinPoint类型分为两种 1：call(插入在函数体里面) 2：excution(插入在函数体外面)

* (\* android.app.Activity+.on\*\*(..))" 匹配规则
* onActivityCalled:要插入的代码

## **异步优化**

核心思想：子线程分担主线程任务，并行减少时间

问题：

1. 不符合异步要求(比如：weex初始化必须在onCreate完成)
2. 需要在某阶段完成

Eg:放在子线程初始化，不知道什么时候初始化完成，主线程调用该sdk的方法可能子线程还没有初始化完成

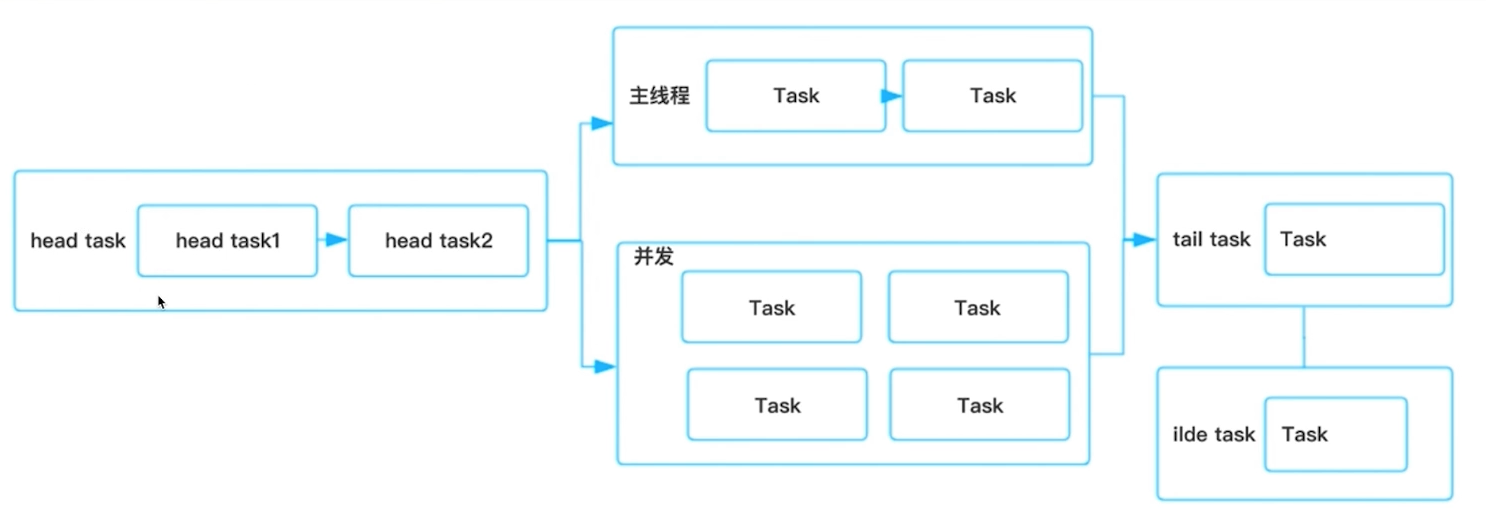
### 1.异步优化的最优解—》启动器

核心思想：充分利用cpu多核，自动梳理任务顺序

1、启动器流程：

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

2、流程图：



### 2.更优秀的延时初始化方案

#### **1.常规延迟初始化的痛点**

* 时机控制不准;比如：postDelay()
* 导致Feed卡顿 比如界面展示后延时初始化操作

#### **2.更优方案—》核心思想：对延时任务进行分批初始化**

* 利用IdleHandler特性，空闲执行

### 3.其他优化方案

1.提前加载SharedPerferences

* Multidex之前加载，利用此阶段cpu
* 覆写getApplicationContext()返回this

2.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
* 注意启动顺序：App onCreate之前是ContentProvider

3.类加载优化：提前异步类加载

* 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

### 总结

#### **1.注意事项：**

1.WallTime与cpu time

* cpuTime才是优化方向
* 按照systrace与cpu 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

#### **2.面试：**

#### **(1)你做启动优化是怎么做的**

1.分析现状，确认问题 （比如主线程初始化任务多）

2.针对性优化---》由异步线程过渡到启动器、优先级不高延时初始化

3.怎么异步的，异步遇到问题没有

（1）体现演进过程—》比如使用Thread、或者线程池 缺点：不够优雅和一些场景不好处理比如依赖关系、某个task在特定生命周期结束

（2）详细介绍启动器 –》代码抽象程task 然后对task依赖关系进行排序—》有向无环图—》使用异步队列异步执行---》异步队列和cpu核心数强烈相关的，最大程度保证主进程和其他线程都能够执行我们的task任务，也就是大家同时完成

(2)你做启动优化觉得哪些容易忽略的注意点

1.cpu time 与 wall time区别

2.注意延迟初始化的优化

（3）版本迭代导致的启动变慢有好的解决方式吗？

1.添加sdk添加到启动器

2.结合CI

3.监控完善，和之前版本进行耗时对比

# **2.内存优化**

## **1.内存问题**

* 内存抖动：锯齿状、GC导致卡顿
* 内存泄漏：可用内存减少、频繁GC
* 内存溢出：OOM、程序异常

## **2.工具选择**

* 1.Memory Profiler

（1）实时图表展示应用内存使用量

（2）识别内存泄漏、抖动等

（3）提供捕获堆转储、强制GC以及跟踪内存分配的能力

* 2.Memory Analyzer

1. 强大的java Heap分析工具，查找内存泄漏及内存占用
2. 生成整体报告、分析问题等
3. 线下深入使用

* 3.LeackCanary

1. 自动内存泄漏检测
2. 线下集成

## **3.内存管理机制**

### **1.java内存分配**

方法区、虚拟机栈、本地方法栈、堆、程序计数器

### 2.Android内存分配

（1）内存弹性分配，分配值与最大值受具体设备影响

（2）OOM场景：内存真正不足、可用内存不足

### 3.Dalvik与Art区别

* Dalvik仅固定一种回收算法
* Art回收算法可运行期选择
* Art具备内存整理能力，减少内存空间

## **4.内存抖动解决实战**

* 使用Memory profiler初步排除呈现锯齿状条
* 使用Memory profiler或CPU profiler结合代码排查

技巧：找循环或者频繁调用的地方

## **5****.内存泄漏解决实战**

定义：内存中存在已经没有用的对象

表现：内存抖动、可用内存逐渐减少

危害：内存不足、GC频繁、OOM

查找定位：通过正则找到对应的类

1. Objects（实例的个数）---》2.右击菜单选中List Objects的with incoming references(强引用引向的自己)->3.右击菜单选中Path To GC Roots—》4. 看到的就是内存泄漏的地方

总结：

1. 使用Memory Profiler初步观察可用内存逐步减少
2. 通过Memory Analyzer结合代码确认

## **6.图片优化**

图片所占内存：宽\*高\*每像素所占字节大小

背景：图片对内存的优化至关重要、图片宽高大于控件宽高

### （1）常规方式：

继承ImageView，覆写实现计算大小

缺点：不通用、侵入性不强

### （2）ARTHook

挂钩，将额外的方法勾住原有方法，修改执行逻辑

运行时插装

性能分析

#### **1.Epic简介**

Epic是一个虚拟机层面、以java Method为粒度的运行时Hook框架，支持API 4.0-9.0 链接地址：https://github.com/chago/epic

2.Epic使用

（1）compile 'me.weishu:epic:0.2.3'

（2）继承XC\_MethodHook，实现相应逻辑

（3）注入Hook：DexposedBridge.findAndHookMethod()

## 7.**线上内存监控方案**

1. 常规方式

* 设定场景线上Dump：Debug.dumpHprofData()将线上内存情况转换为本地文件



总结：Dump文件太大、上传失败率高，分析困难

1. LeakCanary定制

原理：

* 监控生命周期，onDestory添加RefWatcher检测
* 二次确认断定发生内存泄漏
* 分析泄漏，找到引用链

## **8.优化细节**

* LargeHeap属性申请更大内存
* OnTripMemory
* 使用优化后的SparseArray
* 谨慎使用SharedPreference存储大的数据会加载到内存中
* 谨慎使用没有验证过的外部库