深入探索Android卡顿优化（上）

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、卡顿优化分析方法与工具

1、背景介绍

那么卡顿问题到底难在哪里呢?

2、卡顿分析方法之使用shell命令分析CPU耗时

1、了解CPU 性能

2、通过读取/proc/stat与/proc/[PID]/stat文件来计算并评估系统的CPU耗时情况

3、使用top命令查看应用进程的CPU消耗情况

4、PS软件

5、dumpsys cpuinfo

3、卡顿优化工具

1、CPU Profiler回顾

2、Systrace回顾

3、StrictMode

4、Profilo

1、使用profilo的PLTHook来hook libc.so的 write 与 __write_chk 方法

2、使用PLTHook技术来获取线程创建的堆栈

二、自动化卡顿检测方案及优化

1、为什么需要自动化卡顿检测方案?

2、卡顿检测方案原理

卡顿检测方案的具体实现步骤

3、AndroidPerformanceMonitor

BlockCanary的优势如下

那么这种自动检测卡顿的方案有什么问题吗?

那么，我们如何对这种情况进行优化呢?

4、小结

三、总结

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~。

在上篇，笔者详细分析了目前的App绘制与布局优化的相关优化方案，如果对绘制优化与布局优化还不是非常熟悉的可以仔细看看前几篇文章的Android性能优化之绘制优化、深入探索Android布局优化（上）、深入探索Android布局优化（下）。由于卡顿优化这一主题包含的内容太多，因此，笔者将它分为了上、下两篇，本篇，即为深入探索Android卡顿优化的上篇。本篇包含的主要内容如下所示：

- 1、卡顿优化分析方法与工具
- 2、自动化卡顿检测方案及优化

在我们使用各种各样的App的时候，有时会看见有些App运行起来并不流畅，即出现了卡顿现象，那么如何去定义发生了卡顿现象呢？如果App的FPS平均值小于30，最小值小于24，即表明应用发生了卡顿。那我么又如何去分析应用是否出现了卡顿呢？下面，我们就先来了解一下解决卡顿问题时常用到的分析方法与工具。

一、卡顿优化分析方法与工具

1、背景介绍

- 很多性能问题不易被发现，但是卡顿问题很容易被直观感受。
- 卡顿问题难以定位。

那么卡顿问题到底难在哪里呢?

- 1、卡顿产生的原因是错综复杂的，它涉及到代码、内存、绘制、IO、CPU等等。
- 2、线上的卡顿问题在线下是很难复现的，因为它与当时的场景是强相关的，比如说线上用户的磁盘IO空间不足了，它影响了磁盘IO的写入性能，所以导致卡顿。针对这种问题，我们最好在发现卡顿的时候尽量地去记录用户当时发生卡顿时的具体的场景信息。

2、卡顿分析方法之使用shell命令分析CPU耗时

造成卡顿的原因有很多种，不过最终都会反映到CPU时间上。

CPU时间包含用户时间和系统时间。

- 用户时间：执行用户态应用程序代码所消耗的时间。
- 系统时间：执行内核态系统调用所消耗的时间，包括I/O、锁、中断和其它系统调用所消耗的时间。

CPU的问题可以分为以下三类：

1、CPU资源冗余使用

- 算法效率太低：明明可以遍历一次的却需要去遍历两次，主要出现在查找、排序、删除等环节。
- 没有使用cache：明明解码过一次的图片还去重复解码。
- 计算时使用的基本类型不对：明明使用int就足够，却要使用long，这会导致CPU的运算压力多出4倍。

2、CPU资源争抢

- 抢主线程的CPU资源：这是最常见的问题，并且在Android 6.0版本之前没有renderthread的时候，主线程的繁忙程度就决定了是否会引发用户的卡顿问题。
- 抢音视频的CPU资源：音视频编解码本身会消耗大量的CPU资源，并且其对于解码的速度是由硬要求的，达不到就可能产生播放流畅度的问题。我们可以采取两种方式去优化：1、尽量排除非核心业务的消耗。2、优化自身的性能消耗，把CPU负载转化为GPU负载，如使用renderscript来处理视频中的影像信息。
- 大家平等，互相抢：比如在自定义的相册中，我开了20个线程做图片解码，那就是互相抢CPU了，结果就是会导致图片的显示速度非常慢。这简直就是三个和尚没水喝的典型案例。因此，我们需要按照系统核心数去控制线程数。

3、CPU资源利用率低

对于启动、界面切换、音视频编解码这些场景，为了保证其速度，我们需要去好好利用CPU。而导致无法充分利用CPU的因素，不仅有磁盘和网络I/O，还有锁操作、sleep等等。对于锁的优化，通常是尽可能地缩减锁的范围。

1、了解CPU 性能

我们可以通过CPU的主频、核心数、缓存等参数去评估CPU的性能，这些参数的好坏能表现出CPU计算能力和指令执行能力的强弱，也就是CPU每秒执行的浮点计算数和每秒执行的指令数的多少。

此外，现在最新的主流机型都使用了多级能效的CPU架构（即多核分层架构），以确保在平常低负荷工作时能仅使用低频核心来节省电量。

此外，我们还可以通过shell命令直接查看手机的CPU核心数与频率等信息，如下所示：

```
// 先输入adb shell进入手机的shell环境
adb shell
// 获取 CPU 核心数，我的手机是8核
platina:/ $ cat /sys/devices/system/cpu/possible
0-7
// 获取第一个 CPU 的最大频率
platina:/ $ cat
/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq
<
1843200
// 获取第二个CPU的最小频率
platina:/ $ cat
/sys/devices/system/cpu/cpu1/cpufreq/cpuinfo_min_freq
<
633600
```

从 CPU 到 GPU 再到 AI 芯片（如专为神经网络计算打造的 NPU（Neural network Processing Unit）），随着手机 CPU 整体性能的飞跃，医疗诊断、图像超清化等一些 AI 应用场景也可以在移动端更好地落地。我们可以充分利用移动端的计算能力来降低高昂的服务器成本。

此外，CPU的性能越好，应用就能获得更好的支持，如线程池可以根据不同手机的CPU核心数来配备不同的线程数,仅在手机主频比较高或者带有NPU的设备去开启一些高级的AI功能。

2、通过读取/proc/stat与/proc/[PID]/stat文件来计算并评估系统的CPU耗时情况

当应用出现卡顿问题之后，首先我们应该查看系统CPU的使用率。
首先，我们通过读取 /proc/stat 文件获取总的 CPU 时间，并读取 /proc/[PID]/stat 获取应用进程 的 CPU 时间，然后，采样两个足够短的时间间隔的 CPU 快照与进程快照来计算其 CPU 使用率。

计算总的 CPU 使用率

1、采样两个足够短的时间间隔的 CPU 快照，即需要前后两次去读取 /proc/stat 文件，获取两个时间点对应的数据，如下所示：

```
// 第一次采样
platina:/ $ cat /proc/stat
cpu  9931551 1082101 9002534 174463041 340947 1060438 1088978 0 0 0
cpu0 2244962 280573 2667000 22414199 99651 231869 439918 0 0 0
cpu1 2672378 421880 2943791 21540302 121818 236850 438733 0 0 0
cpu2 1648512 76856 1431036 25868789 46970 107094 52025 0 0 0
cpu3 1418757 41280 1397203 25772984 40292 110168 41667 0 0 0
cpu4 573203 79498 178263 19618235 9577 307949 10875 0 0 0
cpu5 522638 67978 155454 19684358 8793 19787 4603 0 0 0
cpu6 458438 64085 132252 19749439 8143 19942 98241 0 0 0
cpu7 392663 49951 97535 19814735 5703 26779 2916 0 0 0
intr...
// 第二次采样
platina:/ $ cat /proc/stat
cpu  9931673 1082113 9002679 174466561 340954 1060446 1088994 0 0 0
cpu0 2244999 280578 2667032 22414604 99653 231869 439918 0 0 0
cpu1 2672434 421881 2943861 21540606 121822 236855 438747 0 0 0
cpu2 1648525 76859 1431054 25869234 46971 107095 52026 0 0 0
cpu3 1418773 41283 1397228 25773412 40292 110170 41668 0 0 0
cpu4 573203 79498 178263 19618720 9577 307949 10875 0 0 0
cpu5 522638 67978 155454 19684842 8793 19787 4603 0 0 0
cpu6 458438 64085 132252 19749923 8143 19942 98241 0 0 0
cpu7 392663 49951 97535 19815220 5703 26779 2916 0 0 0
int...
```

因为我的手机是8核，所以这里的cpu个数是8个，从cpu0到cpu7，第一行的cpu即是8个cpu的指标数据汇总，因为是要计算系统cpu的使用率，那当然应该以cpu为基准了。两次采样的CPU指标数据如下：

```
cpu1  9931551 1082101 9002534 174463041 340947 1060438 1088978 0 0 0
```

```
cpu2  9931673 1082113 9002679 174466561 340954 1060446 1088994 0 0 0
```

其对应的各项指标如下：

```
CPU (user, nice, system, idle, iowait, irq, softirq, stealstolen, gue
st);
```

拿cpu1（9931551 1082101 9002534 174463041 340947 1060438 1088978 0 0 0）的数据来说，下面，我就来详细解释下这些指标的含义。

- user(9931551)： 表示从系统启动开始至今处于用户态的运行时间，注意不包含 nice 值为负的进程。
- nice(1082101)：表示从系统启动开始至今nice 值为负的进程所占用的 CPU 时间。
- system(9002534)： 表示从系统启动开始至今处于内核态的运行时间。
- idle(174463041)：表示从系统启动开始至今除 IO 等待时间以外的其他等待时间。
- iowait(340947)：表示从系统启动开始至今的IO 等待时间。(从Linux V2.5.41开始)
- irq(1060438)：表示从系统启动开始至今的硬中断时间。(从Linux V2.6.0-test4开始)
- softirq(1088978)：表示从系统启动开始至今的软中断时间。(从Linux V2.6.0-test4开始)
- stealstolen(0)：表示当在虚拟化环境中运行时在其他操作系统中所花费的时间。在Android系统下此值为0。(从Linux V2.6.11开始)
- guest(0)：表示当在Linux内核的控制下为其它操作系统运行虚拟CPU所花费的时间。在Android系统下此值为0。(从 V2.6.24开始)

此外，这些数值的单位都是 jiffies，**jiffies 是内核中的一个全局变量，用来记录系统启动以来产生的节拍数，在 Linux 中，一个节拍大致可以理解为操作系统进程调度的最小时间片，不同的 Linux 系统内核这个值可能不同，通常在 1ms 到 10ms 之间。**

了解了/proc/stat命令下各项参数的含义之后，我们就可以由前后两次时间点的CPU数据计算得到cpu1与cpu2的活动时间，如下所示：

```
totalCPUTime = user + nice + system + idle + iowait + irq + softirq +
stealstolen + guest
cpu1 = 9931551 + 1082101 + 9002534 +  174463041 + 340947 + 1060438 +
 1088978 + 0  + 0 + 0 = 196969590jiffies
cpu2 = 9931673 + 1082113 + 9002679 +  174466561 + 340954 + 1060446 +
 1088994 + 0 + 0 + 0 = 196973420jiffies
```

因此可得出总的CPU时间如下所示：

```
totalCPUTime = CPU2 – CPU1 = 3830jiffies
```

最后，我们就可以计算出系统CPU的使用率，如下所示：

```
// 先计算得到CPU的空闲时间
idleCPUTime = idle2 – idle1 = 3520jiffies
// 最后得到系统CPU的使用率
```

```
4 totalCPUUse = (totalCPUTime – idleCPUTime) / totalCPUTime = (3830 – 3
  520) / 3830 = 8%
```

可以看到，前后两次时间点间的CPU使用率大概为8%，说明我们系统的CPU是处于空闲状态的，如果CPU 使用率一直大于 60% ，则表示系统处于繁忙状态，此时就需要进一步分析用户时间和系统时间的比例，看看到底是系统占用了CPU还是应用占用了CPU。

3、使用top命令查看应用进程的CPU消耗情况

此外，由于Android是基于Linux内核改造而成的操作系统，自然而然也能使用Linux的一些常用命令。比如我们可以使用top命令查看哪些进程是 CPU 的主要消耗者。

```
1 1|platina:/ $ top
2 PID USER         PR  NI VIRT  RES  SHR S[%CPU] %MEM     TIME+ ARGS
3 12700 u0_a945      10 -10 4.3G 122M  67M S 15.6   2.1   1:06.41 json.
  chao.com.w+
4 753 system       RT   0  90M 1.1M 1.0M S 13.6   0.0 127:47.73 androi
  d.hardwar+
5 2064 system       18  -2 4.6G 309M 215M S 12.3   5.4 978:15.18 system
  _server
6 22142 u0_a163      20   0 2.0G  97M  41M S 10.3   1.6   2:22.99 com.t
  encent.mob+
7 2293 system       20   0 4.7G 250M  87M S  8.6   4.3 353:15.77 com.an
  droid.sys+
```

从以上可知我们的Awesome-WanAndroid应用进程占用了15%的CPU。此外，这里列举下最常用的top命令，如下所示：

```
1 // 排除0%的进程信息
2 adb shell top | grep -v '0% S'
3 // 只打印1次按CPU排序的TOP 10的进程信息
4 adb shell top -m 10 -s cpu -n 1
5 |platina:/ $ top -d 1|grep json.chao.com.w+
6 5689 u0_a945      10 -10 4.3G 129M  71M S 13.8   2.2   1:04.46 json.
  chao.com.w+
7 5689 u0_a945      10 -10 4.3G 129M  71M S 19.0   2.2   1:04.51 json.
  chao.com.w+
8 5689 u0_a945      10 -10 4.3G 129M  71M S 15.0   2.2   1:04.70 json.
  chao.com.w+
9 5689 u0_a945      10 -10 4.3G 129M  71M S  9.0   2.2   1:04.85 json.
  chao.com.w+
10 5689 u0_a945      10 -10 4.3G 129M  71M S 26.0   2.2   1:04.94 json.
```

```
   chao.com.w+
11 5689 u0_a945      10 -10 4.3G 129M  71M S  9.0   2.2   1:05.20 json.
   chao.com.w+
12 5689 u0_a945      10 -10 4.3G 129M  71M R 17.0   2.2   1:05.29 json.
   chao.com.w+
13 5689 u0_a945      10 -10 4.3G 129M  71M S 20.0   2.2   1:05.46 json.
   chao.com.w+
14 5689 u0_a945      10 -10 4.3G 129M  71M S  9.0   2.2   1:05.66 json.
   chao.com.w+
15 5689 u0_a945      10 -10 4.3G 129M  71M R 21.0   2.2   1:05.75 json.
   chao.com.w+
16 5689 u0_a945      10 -10 4.3G 129M  71M S 14.0   2.2   1:05.96 json.
   chao.com.w+
```

4、PS软件

除了top命令可以比较全面地查看整体的CPU信息之外，如果我们只想查看当前指定进程已经消耗的CPU时间占系统总时间的百分比和其它的状态信息的话，可以使用ps命令，常用的ps命令如下所示：

```
1 // 查看指定进程的状态信息
2 platina:/ $ ps -p 31333
3 USER           PID  PPID     VSZ    RSS WCHAN              ADDR S NAME
4 u0_a945      31333  1277 4521308 127460 0                     0 S json.
  chao.com.w+
5 // 查看指定进程已经消耗的CPU时间占系统总时间的百分比
6 platina:/ $ ps -o PCPU -p 31333
7 %CPU
8 10.8
```

其中输出参数的含义如下所示：

- USER：用户名
- PID：进程ID
- PPID：父进程ID
- VSZ：虚拟内存大小（1k为单位）
- RSS：常驻内存大小（正在使用的页）
- WCHAN：进程在内核态中的运行时间
- Instruction pointer：指令指针
- NAME：进程名字

最后的输出参数S表示的是进程当前的状态，总共有10种可能的状态，如下所示：

```
1 R (running) S (sleeping) D (device I/O) T (stopped)  t (traced)
```

```
2 Z (zombie)  X (deader)   x (dead)       K (wakekill) W (waking)
```

可以看到，我们当前主进程是休眠的状态。

5、dumpsys cpuinfo

使用dumpsys cpuinfo命令获得的信息比起top命令得到的信息要更加精炼，如下所示：

```
1 platina:/ $ dumpsys cpuinfo
2 Load: 1.92 / 1.59 / 0.97
3 CPU usage from 45482ms to 25373ms ago (2020-02-04 17:00:37.666 to 20
  20-02-04 17:00:57.775):
4 33% 2060/system_server: 22% user + 10% kernel / faults: 8152 minor 6
  major
5 17% 2292/com.android.systemui: 12% user + 4.7% kernel / faults: 2163
  6 minor 3 major
6 14% 750/android.hardware.sensors@1.0-service: 4.4% user + 10% kernel
7 6.1% 778/surfaceflinger: 3.3% user + 2.7% kernel / faults: 128 minor
8 3.3% 2598/com.miui.home: 2.8% user + 0.4% kernel / faults: 7655 mino
  r 11 major
9 2.2% 2914/cnss_diag: 1.6% user + 0.6% kernel
10 1.9% 745/android.hardware.graphics.composer@2.1-service: 1.4% user +
  0.5% kernel / faults: 5 minor
11 1.7% 4525/kworker/u16:6: 0% user + 1.7% kernel
12 1.6% 748/android.hardware.memtrack@1.0-service: 0.6% user + 0.9% ker
  nel
13 1.4% 4551/kworker/u16:14: 0% user + 1.4% kernel
14 1.4% 31333/json.chao.com.wanandroid: 0.9% user + 0.4% kernel / fault
  s: 3995 minor 22 major
15 1.1% 6670/kworker/u16:0: 0% user + 1.1% kernel
16 0.9% 448/mmc-cmdqd/0: 0% user + 0.9% kernel
17 0.7% 95/system: 0% user + 0.7% kernel
18 0.6% 4512/mdss_fb0: 0% user + 0.6% kernel
19 0.6% 7393/com.android.incallui: 0.6% user + 0% kernel / faults: 2272
  minor
20 0.6% 594/logd: 0.4% user + 0.1% kernel / faults: 38 minor 3 major
21 0.5% 3108/com.xiaomi.xmsf: 0.2% user + 0.2% kernel / faults: 1812 mi
  nor
22 0.5% 4526/kworker/u16:9: 0% user + 0.5% kernel
23 0.5% 4621/com.gotokeep.keep: 0.3% user + 0.1% kernel / faults: 55 mi
  nor
24 0.5% 354/irq/267-NVT-ts: 0% user + 0.5% kernel
```

```
0.5% 2572/com.android.phone: 0.3% user + 0.1% kernel / faults: 323 minor
0.5% 4554/kworker/u16:15: 0% user + 0.5% kernel
0.4% 290/kgsl_worker_thr: 0% user + 0.4% kernel
0.3% 2933/irq/61-1008000.: 0% user + 0.3% kernel
0.3% 3932/com.tencent.mm: 0.2% user + 0% kernel / faults: 647 minor 1 major
0.3% 4550/kworker/u16:13: 0% user + 0.3% kernel
0.3% 744/android.hardware.graphics.allocator@2.0-service: 0% user + 0.3% kernel / faults: 48 minor
0.3% 8906/com.tencent.mm:appbrand0: 0.2% user + 0% kernel / faults: 45 minor
0.2% 79/smem_native_rpm: 0% user + 0.2% kernel
0.2% 759/vendor.qti.hardware.perf@1.0-service: 0% user + 0.2% kernel / faults: 46 minor
0.2% 3197/com.miui.powerkeeper: 0% user + 0.1% kernel / faults: 141 minor
0.2% 4489/kworker/1:1: 0% user + 0.2% kernel
0.2% 595/servicemanager: 0% user + 0.2% kernel
0.2% 754/android.hardware.wifi@1.0-service: 0.1% user + 0% kernel
0.2% 1258/jbd2/dm-2-8: 0% user + 0.2% kernel
0.2% 5800/com.eg.android.AlipayGphone: 0.1% user + 0% kernel / faults: 48 minor
0.2% 21590/iptables-restore: 0% user + 0.1% kernel / faults: 563 minor
0.2% 21592/ip6tables-restore: 0% user + 0.1% kernel / faults: 647 minor
0.1% 3/ksoftirqd/0: 0% user + 0.1% kernel
0.1% 442/cfinteractive: 0% user + 0.1% kernel
0.1% 568/ueventd: 0% user + 0% kernel
0.1% 1295/netd: 0% user + 0.1% kernel / faults: 250 minor
0.1% 3002/com.miui.securitycenter.remote: 0.1% user + 0% kernel / faults: 818 minor 1 major
0.1% 20555/com.eg.android.AlipayGphone:push: 0% user + 0% kernel / faults: 20 minor
0.1% 7/rcu_preempt: 0% user + 0.1% kernel
0.1% 15/ksoftirqd/1: 0% user + 0.1% kernel
0.1% 76/lpass_smem_glin: 0% user + 0.1% kernel
0.1% 1299/rild: 0.1% user + 0% kernel / faults: 12 minor
0.1% 1448/android.process.acore: 0.1% user + 0% kernel / faults: 171
```

```
9 minor
0% 4419/com.google.android.webview:s: 0% user + 0% kernel / faults:
 602 minor
0% 20465/com.miui.hybrid: 0% user + 0% kernel / faults: 1575 minor
0% 10/rcuop/0: 0% user + 0% kernel
0% 75/smem_native_lpa: 0% user + 0% kernel
0% 90/kcompactd0: 0% user + 0% kernel
0% 1508/msm_irqbalance: 0% user + 0% kernel
0% 1745/cds_mc_thread: 0% user + 0% kernel
0% 2899/charge_logger: 0% user + 0% kernel
0% 3612/com.tencent.mm:tools: 0% user + 0% kernel / faults: 29 minor
0% 4203/kworker/0:0: 0% user + 0% kernel
0% 7377/com.android.server.telecom:ui: 0% user + 0% kernel / faults:
1083 minor
0% 32113/com.tencent.mobileqq: 0% user + 0% kernel / faults: 49 mino
r
0% 8/rcu_sched: 0% user + 0% kernel
0% 22/ksoftirqd/2: 0% user + 0% kernel
0% 25/rcuop/2: 0% user + 0% kernel
0% 29/ksoftirqd/3: 0% user + 0% kernel
0% 39/rcuop/4: 0% user + 0% kernel
0% 53/rcuop/6: 0% user + 0% kernel
0% 487/irq/715-ima-rdy: 0% user + 0% kernel
0% 749/android.hardware.power@1.0-service: 0% user + 0% kernel
0% 764/healthd: 0% user + 0% kernel / faults: 2 minor
0% 845/wlan_logging_th: 0% user + 0% kernel
0% 860/mm-pp-dpps: 0% user + 0% kernel
0% 1297/wificond: 0% user + 0% kernel / faults: 12 minor
 0% 1309/com.miui.weather2: 0% user + 0% kernel / faults: 729 minor
 23 major
0% 1542/rild: 0% user + 0% kernel / faults: 3 minor
0% 2915/tcpdump: 0% user + 0% kernel / faults: 6 minor
0% 2974/com.tencent.mobileqq:MSF: 0% user + 0% kernel / faults: 121
 minor
0% 3044/com.miui.contentcatcher: 0% user + 0% kernel / faults: 315 m
inor
0% 3057/com.miui.dmregservice: 0% user + 0% kernel / faults: 332 min
or
0% 3095/com.xiaomi.mircs: 0% user + 0% kernel
0% 3115/com.xiaomi.finddevice: 0% user + 0% kernel / faults: 270 min
```

```
or 3 major
0% 3513/com.xiaomi.metoknlp: 0% user + 0% kernel / faults: 136 minor
0% 3603/com.tencent.mm:toolsmp: 0% user + 0% kernel / faults: 35 min
or
0% 4527/kworker/u16:11: 0% user + 0% kernel
0% 4841/com.gotokeep.keep:xg_service_v4: 0% user + 0% kernel / fault
s: 275 minor
0% 5064/com.sohu.inputmethod.sogou.xiaomi: 0% user + 0% kernel / fau
lts: 102 minor
0% 5257/kworker/0:1: 0% user + 0% kernel
0% 5839/com.tencent.mm:push: 0% user + 0% kernel / faults: 98 minor
0% 6644/kworker/3:2: 0% user + 0% kernel
0% 6657/com.miui.wmsvc: 0% user + 0% kernel / faults: 52 minor
0% 6945/com.xiaomi.account:accountservice: 0% user + 0% kernel / fau
lts: 1 minor
0% 9387/com.tencent.mm:appbrand1: 0% user + 0% kernel / faults: 27 m
inor
13% TOTAL: 6.8% user + 5.3% kernel + 0.2% iowait + 0.3% irq + 0.4% s
oftirq
```

从上述信息可知，第一行显示的是cpuload （负载平均值）信息：Load: 1.92 / 1.59 / 0.97
这三个数字表示逐渐变长的时间段（平均一分钟，五分钟和十五分钟）的平均值，而较低的数字则更好。数字越大表示问题或机器过载。需要注意的是，这里的Load需要除以核心数，这里的系统核心数为8核，所以最终每一个单核CPU的Load为0.24 / 0.20 / 0.12，如果Load超过1，则表示出现了问题。
此外，占用系统CPU资源最高的是system_server进程，而我们的wanandroid应用进程仅占用了1.4%的CPU资源，其中有0.9%的是用户态所占用的时间，0.4%是内核态所占用的时间。最后，我们可以看到系统总占用的CPU时间是13%，这个值是根据**前面所有值加起来 / 系统CPU数**的处理的，也就是**104% / 8 = 13%**。
除了上述方式来分析系统与应用的CPU使用情况之外，我们还应该关注卡顿率与卡顿树这两个指标。它们能帮助我们有效地去评估、并且更有针对性地去优化应用发生的卡顿。

卡顿率

类似于深入探索Android稳定性优化一文中讲到的UV、PV崩溃率，卡顿也可以有其对应的UV、PV卡顿率，UV就是Unique visitor，指的就是一台手机客户端为一个访客，00:00-24:00内相同的客户端只被计算一次。而PV即Page View，即页面浏览量或点击量。所以UV、PV卡顿率的定义即为如下所示：

```
// UV 卡顿率可以评估卡顿的影响范围
UV 卡顿率 = 发生过卡顿 UV / 开启卡顿采集 UV
// PV 卡顿率评估卡顿的严重程度
PV 卡顿率 = 发生过卡顿 PV / 启动采集 PV
```

因为卡顿问题的采样规则跟内存问题是相似的，一般都是采取抽样上报的方式，并且都应该按照单个用户来抽样。一个用户如果命中采集，那么在一天内都会持续的采集数据。

卡顿树

我们可以实现卡顿的火焰图，即卡顿树，在一张图里就可以看到卡顿的整体信息。由于卡顿的具体耗时跟手机性能与当时的使用场景与环境密切相关，而且卡顿问题在日活大应用上出现的场景非常多，所以对于大于我们指定的卡顿阈值如1s\2s\3s时，我们就可以抛弃具体的耗时，只按照相同堆栈出现的比例来聚合各类卡顿信息。这样我们就能够很直观地从卡顿树上看到到底哪些堆栈出现的卡顿问题最多，以便于我们能够优先去解决 Top 的卡顿问题，达到使用最少的精力获取最大的优化效果的目的。

3、卡顿优化工具

1、CPU Profiler回顾

CPU Profiler的使用笔者已经在深入探索Android启动速度优化中详细分析过了，如果对CPU Profiler还不是很熟悉的话，可以去看看这篇文章。
下面我们来简单来回顾一下CPU Profiler。

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。

劣势：

运行时开销严重，整体都会变慢，可能会带偏我们的优化方向。

使用方式：

```
Debug.startMethodTracing("");
// 需要检测的代码片段
...
Debug.stopMethodTracing();
```

最终生成的生成文件在sd卡：Android/data/packagename/files。

2、Systrace回顾

systrace 利用了 Linux 的ftrace调试工具（ftrace是用于了解Linux内核内部运行情况的调试工具），相当于在系统各个关键位置都添加了一些性能探针，也就是在代码里加了一些性能监控的埋点。Android 在 ftrace 的基础上封装了atrace，并增加了更多特有的探针，比如Graphics、Activity Manager、Dalvik VM、System Server 等等。对于Systrace的使用笔者在深入探索Android启动速度优化这篇文章中已经详细分析过了，如果对Systrace还不是很熟悉的话可以去看看这篇文章。
下面我们来简单回顾一下Systrace。

作用：

监控和跟踪API调用、线程运行情况，生成HTML报告。

建议：

API 18以上使用，推荐使用TraceCompat。

使用方式：

使用python命令执行脚本，后面加上一系列参数，如下所示：

```
python systrace.py -t 10 [other-options] [categories]
```

优势：

- 1、轻量级，开销小。
- 2、 它能够直观地反映CPU的利用率。
- 3、根据右侧的Alerts能够根据我们的问题给出具体的建议，比如说，它会告诉我们App界面的绘制比较慢或者GC比较频繁。

最后，我们还可以通过编译时给每个函数插桩的方式来实现线下自动增加应用程序的耗时分析，但是要注意需过滤大部分的短函数，以减少性能损耗（这一点可以通过黑名单配置的方式去过滤短函数或调用非常频繁的函数）。使用这种方式我们就可以看到整个应用程序的调用流程。包括应用关键线程的函数调用，例如渲染耗时、线程锁，GC 耗时等等。

基于性能的考虑，如果要在线上使用此方案，最好只去监控主线程的耗时。虽然插桩方案对性能的影响并不是很大，但是建议仅在线下或灰度环境中使用。

此外，如果你需要分析Native 函数的调用，请使用Android 5.0 新增的Simpleperf性能分析工具，它利用了 CPU 的性能监控单元（PMU）提供的硬件 perf 事件。使用 Simpleperf 可以看到所有的 Native 代码的耗时，对一些 Android 系统库的调用对分析问题有比较大的帮助，例如加载 dex、verify class 的耗时等等。此外，在 Android Studio 3.2 中的 Profiler 也直接支持了 Simpleper（SampleNative性能分析工具（API Level 26+)），这更加方便了native代码的调试。

3、StrictMode

StrictMode是Android 2.3引入的一个工具类，它被称为严苛模式，是Android提供的一种运行时检测机制，可以用来帮助开发人员用来检测代码中一些不规范的问题。对于我们的项目当中，可能会成千上万行代码，如果我们用肉眼Review，这样不仅效率非常低效，而且比较容易出问题。使用StrictMode之后，系统会自动检测出来在主线程中的一些异常情况，并按照我们的配置给出相应的反应。

StrictMode这个工具是非常强大的，但是我们可能因为对它不熟悉而忽略掉它。StrictMode主要用来检测两大问题：

1、线程策略

线程策略的检测内容，是一些自定义的耗时调用、磁盘读取操作以及网络请求等。

2、虚拟机策略

虚拟机策略的检测内容如下：

- Activity泄漏
- Sqlite对象泄漏
- 检测实例数量

StrictMode实战

如果要在应用中使用StrictMode，只需要在Applicaitoin的onCreate方法中对StrictMode进行统一配置，代码如下所示：

```
private void initStrictMode() {
    // 1、设置Debug标志位，仅仅在线下环境才使用StrictMode
    if (DEV_MODE) {
        // 2、设置线程策略
        StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
                .detectCustomSlowCalls() //API等级11，使用StrictMode.noteSlowCode
                .detectDiskReads()
                .detectDiskWrites()
                .detectNetwork()// or .detectAll() for all detectable problems
                .penaltyLog() //在Logcat 中打印违规异常信息
//              .penaltyDialog()//也可以直接跳出警报dialog
//              .penaltyDeath()//或者直接崩溃
                .build());
        // 3、设置虚拟机策略
        StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
                .detectLeakedSqlLiteObjects()
                // 给NewsItem对象的实例数量限制为1
                .setClassInstanceLimit(NewsItem.class, 1)
                .detectLeakedClosableObjects() //API等级11
                .penaltyLog()
                .build());
    }
}
```

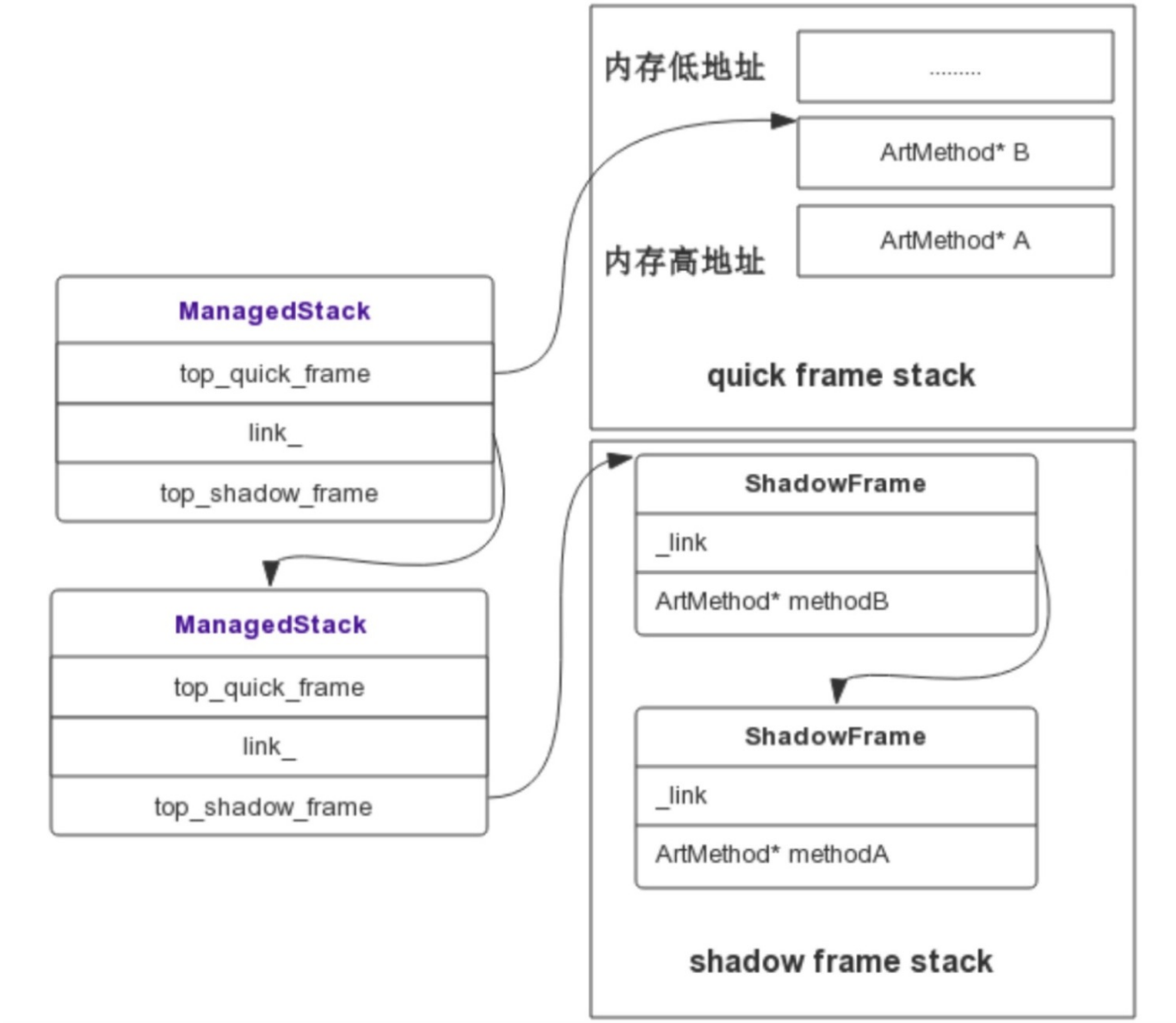
注意使用“StrictMode”关键字过滤出对应的log。

4、Profilo

Profilo是一个用于收集应用程序生产版本的性能跟踪的Android库。

对于Profilo来说，它集成了atrace功能，ftrace 所有的性能埋点数据都会通过 trace_marker 文件写入到内核缓冲区，Profilo 使用了 PLT Hook 拦截了写入操作，以选择部分关心的事件去做特定的分析。这样所有的 systrace 的探针我们都可以拿到，例如四大组件生命周期、锁等待时间、类校验、GC 时间等等。不过大部分的 atrace 事件都比较笼统，从事件“B|pid|activityStart”，我们无法明确知道该事件具体是由哪个 Activity 来创建的。

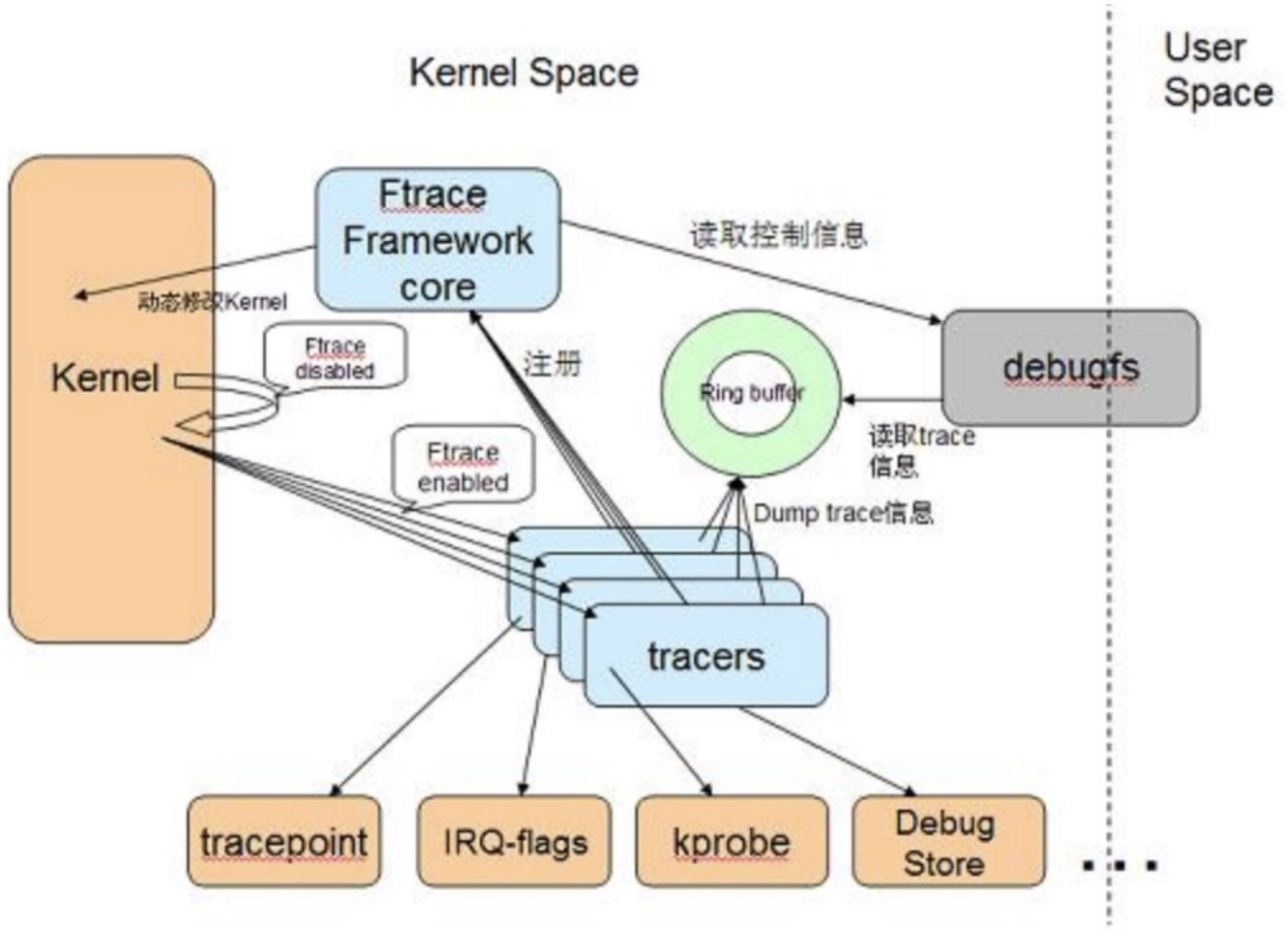
此外，并能够快速获取Java堆栈。由于获取堆栈需要暂停主线程的运行，所以profilo通过间隔发送SIGPROF 信号这样一种类似 Native 崩溃捕捉的方式去快速获取 Java 堆栈。能够低耗时地快速获取Java堆栈的具体实现原理为当Signal Handler 捕获到信号后，它就会获取到当前正在执行的 Thread，通过 Thread 对象就可以拿到当前线程的 ManagedStack，ManagedStack 是一个单链表，它保存了当前的 ShadowFrame 或者 QuickFrame 栈指针，先依次遍历 ManagedStack 链表，然后遍历其内部的 ShadowFrame 或者 QuickFrame 还原一个可读的调用栈，从而 unwind 出当前的 Java 堆栈。关于ManagedStack与ShadowFrame、QuickFrame三者的关系如下图所示:



Profilo通过这种方式，就可以实现线程同步运行的同时，我们还可以去帮它做检查，并且耗时基本可以忽略不计。但是目前 Profilo 快速获取堆栈的功能不支持 Android 8.0 和 Android 9.0，并且它内部使用了Hook等大量的黑科技手段，鉴于稳定性问题，建议采取抽样部分用户的方式来开启该功能。

Profilo项目地址

Systrace主要是根据Linux的ftrace机制来实现的，而ftrace的作用是帮助我们了解 Linux 内核的运行时行为，以便进行故障调试或性能分析。ftrace的整体架构如下所示：



由上图可知，Ftrace 有两大组成部分，一是 framework，另外就是一系列的 tracer 。每个 tracer 用于完成不同的功能，并且它们统一由 framework 管理。 ftrace 的 trace 信息保存在 ring buffer 中，由 framework 负责管理。 Framework 利用 debugfs 系统在 /debugfs 下建立 tracing 目录，并提供了一系列的控制文件。

下面，我们这里给出使用 PLTHook 技术来获取 Atrace 的日志的一个项目。

1、使用profilo的PLTHook来hook libc.so的 write 与 __write_chk 方法

使用 PLTHook 技术来获取 Atrace 的日志-项目地址

运行项目后，我们点击按钮开启Atrace日志，然后就可以在Logcat中看到如下的native层日志信息：

```
2020-02-05 10:58:00.873 13052-13052/com.dodola.atrace I/HOOOOOOOK: ===============install systrace hoook==================
2020-02-05 10:58:00.879 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|inflate
2020-02-05 10:58:00.880 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|LinearLayout
2020-02-05 10:58:00.881 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= E
2020-02-05 10:58:00.882 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|TextView
```

```
2020-02-05 10:58:00.884 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= E
2020-02-05 10:58:00.885 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= E
2020-02-05 10:58:00.888 13052-13075/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|notifyFramePending
2020-02-05 10:58:00.888 13052-13075/com.dodola.atrace I/HOOOOOOOK: ========= E
2020-02-05 10:58:00.889 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|Choreographer#doFrame
2020-02-05 10:58:00.889 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|input
2020-02-05 10:58:00.889 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= E
2020-02-05 10:58:00.889 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|traversal
2020-02-05 10:58:00.889 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|draw
2020-02-05 10:58:00.890 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|Record View#draw()
2020-02-05 10:58:00.891 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= E
2020-02-05 10:58:00.891 13052-13075/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|DrawFrame
2020-02-05 10:58:00.891 13052-13075/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|syncFrameState
2020-02-05 10:58:00.891 13052-13075/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|prepareTree
2020-02-05 10:58:00.891 13052-13075/com.dodola.atrace I/HOOOOOOOK: ========= E
2020-02-05 10:58:00.891 13052-13075/com.dodola.atrace I/HOOOOOOOK: ========= E
2020-02-05 10:58:00.891 13052-13075/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|query
2020-02-05 10:58:00.891 13052-13052/com.dodola.atrace I/HOOOOOOOK: ========= E
2020-02-05 10:58:00.891 13052-13075/com.dodola.atrace I/HOOOOOOOK: ========= E
2020-02-05 10:58:00.891 13052-13075/com.dodola.atrace I/HOOOOOOOK: ========= B|13052|query
```

```
2020-02-05 10:58:00.891 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= E
2020-02-05 10:58:00.892 13052-13052/com.dodola.atrace I/HOOOOOOOK:
 ========= E
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= B|13052|query
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= E
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= B|13052|query
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= E
2020-02-05 10:58:00.892 13052-13052/com.dodola.atrace I/HOOOOOOOK:
 ========= E
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= B|13052|query
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= E
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= B|13052|query
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= E
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= B|13052|setBuffersDimensions
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= E
2020-02-05 10:58:00.892 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= B|13052|dequeueBuffer
2020-02-05 10:58:00.894 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= B|13052|importBuffer
2020-02-05 10:58:00.894 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= B|13052|HIDL::IMapper::importBuffer::passthrough
2020-02-05 10:58:00.894 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= E
2020-02-05 10:58:00.894 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= E
2020-02-05 10:58:00.894 13052-13058/com.dodola.atrace I/HOOOOOOOK:
 ========= B|13052|Compiling
2020-02-05 10:58:00.894 13052-13075/com.dodola.atrace I/HOOOOOOOK:
 ========= E
```

```
46 2020-02-05 10:58:00.894 13052-13075/com.dodola.atrace I/HOOOOOOOK:
    ========= B|13052|query
```

需要注意的是，日志中的B代表begin，也就是对应时间开始的时间，而E代表End，即对应事件结束的时间，并且，B|事件和E|事件是成对出现的，这样我们就可以通过该事件的结束时间减去对应的开始时间来获得每个事件使用的时间。例如，上述log中我们可以看出TextView的draw方法显示使用了3ms。
此外，在下面这个项目里展示了如何使用 PLTHook 技术来获取线程创建的堆栈。

2、使用PLTHook技术来获取线程创建的堆栈

使用 PLTHook 技术来获取线程创建的堆栈-项目地址
运行项目后，我们点击开启 Thread Hook按钮，然后点击新建 Thread按钮。最后可以在Logcat 中看到Thread创建的堆栈信息：

```
1 2020-02-05 13:47:59.006 20159-20159/com.dodola.thread E/HOOOOOOOK:
   stack:com.dodola.thread.ThreadHook.getStack(ThreadHook.java:16)
2 com.dodola.thread.MainActivity$2.onClick(MainActivity.java:40)
3 android.view.View.performClick(View.java:6311)
4 android.view.View$PerformClick.run(View.java:24833)
5 android.os.Handler.handleCallback(Handler.java:794)
6 android.os.Handler.dispatchMessage(Handler.java:99)
7 android.os.Looper.loop(Looper.java:173)
8 android.app.ActivityThread.main(ActivityThread.java:6653)
9 java.lang.reflect.Method.invoke(Native Method)
10 com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeI
   nit.java:547)
11 com.android.internal.os.ZygoteInit.main(ZygoteInit.java:821)
12 2020-02-05 13:47:59.007 20159-20339/com.dodola.thread E/HOOOOOOOK:
    thread name:Thread-2
13 2020-02-05 13:47:59.008 20159-20339/com.dodola.thread E/HOOOOOOOK:
    thread id:1057
14 2020-02-05 13:47:59.009 20159-20339/com.dodola.thread E/HOOOOOOOK:
    stack:com.dodola.thread.ThreadHook.getStack(ThreadHook.java:16)
15 com.dodola.thread.MainActivity$2$1.run(MainActivity.java:38)
16 2020-02-05 13:47:59.011 20159-20340/com.dodola.thread E/HOOOOOOOK:
    inner thread name:Thread-3
17 2020-02-05 13:47:59.012 20159-20340/com.dodola.thread E/HOOOOOOOK:
    inner thread id:1058
```

由于Profilo与PLT Hook涉及了大量的C/C++、NDK开发的知识，所以这部分不做详细讲解，如对NDK开发感兴趣的同学可以期待下我后面的Awesome-Android-NDK系列文章，等性能优化系列文章更新完毕之后，就会开始去系统学习NDK相关的开发知识，敬请期待。

二、自动化卡顿检测方案及优化

1、为什么需要自动化卡顿检测方案?

- 1、Cpu Profiler、Systrace等系统工具仅适合线下针对性分析。
- 2、线上及测试环境需要自动化的卡顿检方案来定位卡顿，同时，更重要的是，它能记录卡顿发生时的场景。

2、卡顿检测方案原理

它的原理源于Android的消息处理机制，一个线程不管有多少Handler，它只会有一个Looper存在，主线程执行的任何代码都会通过Looper.loop()方法执行。而在Looper函数中，它有一个mLogging对象，这个对象在每个message处理前后都会被调用。主线程发生了卡顿，那一定是在dispatchMessage()方法中执行了耗时操作。那么，我们就可以通过这个mLogging对象对dispatchMessage()进行监控。

卡顿检测方案的具体实现步骤

首先，我们看下Looper用于执行消息循环的loop()方法，关键代码如下所示：

```
/**
 * Run the message queue in this thread. Be sure to call
 * {@link #quit()} to end the loop.
 */
public static void loop() {
    ...
    for (;;) {
        Message msg = queue.next(); // might block
        if (msg == null) {
            // No message indicates that the message queue is quitti
ng.
            return;
        }
        // This must be in a local variable, in case a UI event sets
 the logger
        final Printer logging = me.mLogging;
        if (logging != null) {
            // 1
            logging.println(">>>>> Dispatching to " + msg.target + "
 " +
                    msg.callback + ": " + msg.what);
        }
        ...
        try {
```

```
22              // 2
23              msg.target.dispatchMessage(msg);
24             dispatchEnd = needEndTime ? SystemClock.uptimeMillis() :
   0;
25         } finally {
26             if (traceTag != 0) {
27                 Trace.traceEnd(traceTag);
28             }
29         }
30         ...
31         if (logging != null) {
32             // 3
33             logging.println("<<<<< Finished to " + msg.target + " "
    + msg.callback);
34         }
```

在Looper的loop()方法中，在其执行每一个消息（注释2处）的前后都由logging进行了一次打印输出。可以看到，在执行消息前是输出的”>>>>> Dispatching to “，在执行消息后是输出的”<<<<< Finished to “,它们打印的日志是不一样的，我们就可以由此来判断消息执行的前后。

所以，具体的实现可以归纳为如下步骤：

- 1、首先，我们需要使用Looper.getMainLooper().setMessageLogging()去设置我们自己的Printer实现类去打印输出logging。这样，在每个message执行的之前和之后都会调用我们设置的这个Printer实现类。
- 2、如果我们匹配到”>>>>> Dispatching to “之后，我们就可以执行一行代码，也就是在指定的时间阈值之后，我们在子线程去执行一个任务，这个任务就是去获取当前主线程的堆栈信息以及当前的一些场景信息，比如：内存大小、电脑、网络状态等。
- 3、如果匹配到了”<<<<< Finished to “，那么说明在指定的阈值之内，message就被执行完成了，说明此时没有产生我们认为的卡顿效果，那我们就可以将这个子线程任务取消掉。

3、AndroidPerformanceMonitor

它是一个非侵入式的性能监控组件，可以通过通知的形式弹出卡顿信息。它的原理就是我们刚刚讲述到的卡顿监控的实现原理。

接下我们通过一个简单的示例来讲解一下它的使用。

首先，我们需要在moudle的build.gradle下配置它的依赖，如下所示：

```
1 // release：项目中实现了线上监控体系的时候去使用
2 api 'com.github.markzhai:blockcanary-android:1.5.0'
3 // 仅在debug包启用BlockCanary进行卡顿监控和提示的话，可以这么用
4 debugApi 'com.github.markzhai:blockcanary-android:1.5.0'
5 releaseApi 'com.github.markzhai:blockcanary-no-op:1.5.0'
```

其次，在Application的onCreate方法中开启卡顿监控：

```
// 注意在主进程初始化调用
BlockCanary.install(this, new AppBlockCanaryContext()).start();
```

最后，继承BlockCanaryContext类去实现自己的监控配置上下文类：

```
public class AppBlockCanaryContext extends BlockCanaryContext {
    // 实现各种上下文，包括应用标识符，用户uid，网络类型，卡顿判断阈值，Log保存位置等等
    /**
    * 提供应用的标识符
    *
    * @return 标识符能够在安装的时候被指定，建议为 version + flavor.
    */
    public String provideQualifier() {
        return "unknown";
    }
    /**
    * 提供用户uid，以便在上报时能够将对应的
    * 用户信息上报至服务器
    *
    * @return user id
    */
    public String provideUid() {
        return "uid";
    }
    /**
    * 提供当前的网络类型
    *
    * @return {@link String} like 2G, 3G, 4G, wifi, etc.
    */
    public String provideNetworkType() {
        return "unknown";
    }
    /**
    * 配置监控的时间区间，超过这个时间区间    ，BlockCanary将会停止，use
    * with {@code BlockCanary}'s isMonitorDurationEnd
    *
    * @return monitor last duration (in hour)
```

```
33      */
34      public int provideMonitorDuration() {
35          return -1;
36      }
37      /**
38      * 指定判定为卡顿的阈值threshold (in millis),
39      * 你可以根据不同设备的性能去指定不同的阈值
40      *
41      * @return threshold in mills
42      */
43      public int provideBlockThreshold() {
44          return 1000;
45      }
46      /**
47      * 设置线程堆栈dump的间隔, 当阻塞发生的时候使用, BlockCanary 将会根据
48      * 当前的循环周期在主线程去dump堆栈信息
49      * <p>
50      * 由于依赖于Looper的实现机制, 真实的dump周期
51      * 将会比设定的dump间隔要长(尤其是当CPU很繁忙的时候).
52      * </p>
53      *
54      * @return dump interval (in millis)
55      */
56      public int provideDumpInterval() {
57          return provideBlockThreshold();
58      }
59      /**
60      * 保存log的路径, 比如 "/blockcanary/", 如果权限允许的话,
61      * 会保存在本地sd卡中
62      *
63      * @return path of log files
64      */
65      public String providePath() {
66          return "/blockcanary/";
67      }
68      /**
69      * 是否需要通知去通知用户发生阻塞
70      *
71      * @return true if need, else if not need.
72      */
```

```
73      public boolean displayNotification() {
74          return true;
75      }
76      /**
77      * 用于将多个文件压缩为一个.zip文件
78      *
79      * @param src  files before compress
80      * @param dest files compressed
81      * @return true if compression is successful
82      */
83      public boolean zip(File[] src, File dest) {
84          return false;
85      }
86      /**
87      * 用于将已经被压缩好的.zip log文件上传至
88      * APM后台
89      *
90      * @param zippedFile zipped file
91      */
92      public void upload(File zippedFile) {
93          throw new UnsupportedOperationException();
94      }
95      /**
96      * 用于设定包名, 默认使用进程名,
97      *
98      * @return null if simply concern only package with process nam
    e.
99      */
100     public List<String> concernPackages() {
101         return null;
102     }
103     /**
104     * 使用 @{code concernPackages}方法指定过滤的堆栈信息
105     *
106     * @return true if filter, false it not.
107     */
108     public boolean filterNonConcernStack() {
109         return false;
110     }
111     /**
```

```
112     * 指定一个白名单，在白名单的条目将不会出现在展示阻塞信息的UI中
113     *
114     * @return return null if you don't need white-list filter.
115     */
116     public List<String> provideWhiteList() {
117         LinkedList<String> whiteList = new LinkedList<>();
118         whiteList.add("org.chromium");
119         return whiteList;
120     }
121     /**
122     * 使用白名单的时候，是否去删除堆栈在白名单中的文件
123     *
124     * @return true if delete, false it not.
125     */
126     public boolean deleteFilesInWhiteList() {
127         return true;
128     }
129     /**
130     * 阻塞拦截器，我们可以指定发生阻塞时应该做的工作
131     */
132     public void onBlock(Context context, BlockInfo blockInfo) {
133     }
134 }
```

可以看到，在上述配置中，我们指定了卡顿的阈值为1000ms。接下来，我们可以测试一下BlockCanary监测卡顿时的效果，这里我在Activity的onCreate方法中添加如下代码使线程休眠3s：

```
1 try {
2     Thread.sleep(3000);
3 } catch (InterruptedException e) {
4     e.printStackTrace();
5 }
```

然后，我们运行项目，打开App，即可看到类似LeakCanary那样的对应的卡顿信息堆栈。
除了发生卡顿时BlockCanary提供的图形界面可供开发和测试人员直接查看卡顿原因之外。其最大的作用还是在线上环境或者自动化monkey测试的环节进行大范围的log采集与分析，对于分析的纬度，可以从以下两个纬度来进行：

- 卡顿时间。
- 根据同堆栈出现的卡顿次数来进行排序和归类。

BlockCanary的优势如下

- 非侵入式。
- 方便精准，能够定位到代码的某一行代码。

那么这种自动检测卡顿的方案有什么问题吗?

在卡顿的周期之内，应用确实发生了卡顿，但是获取到的卡顿信息可能会不准确，和我们的OOM一样，也就是最后的堆栈信息仅仅只是一个表象，并不是真正发生问题时的一个堆栈。下面，我们先看下如下的一个示意图：

假设主线程在T1到T2的时间段内发生了卡顿，卡顿检测方案获取卡顿时的堆栈信息是T2时刻，但是实际上发生卡顿的时刻可能是在这段时间区域内某个函数的耗时过长，那么可能在我们捕获卡顿的时刻时，真正的卡顿时机已经执行完成了，所以在T2时刻捕获到的一个卡顿信息并不能够反映卡顿的现场，也就是最后呈现出来的堆栈信息仅仅只是一个表象，并不是真正问题的藏身之处。

那么，我们如何对这种情况进行优化呢?

我们可以获取卡顿周期内的多个堆栈，而不仅仅是最后一个，这样的话，如果发生了卡顿，我们就可以根据这些堆栈信息来清晰地还原整个卡顿现场。因为我们有卡顿现场的多个堆栈信息，我们完全知道卡顿时究竟发生了什么，到底哪些函数它的调用时间比较长。接下来，我们看看下面的卡顿检测优化流程图：

根据图中，可以梳理出优化后的具体实现步骤为：

- 1、首先，我们会通过startMonitor方法对这个过程进行监控。
- 2、接着，我们就开始高频采集堆栈信息。如果发生了卡顿，我们就会调用endMonitor方法。
- 3、然后，将之前我们采集的多个堆栈信息记录到文件中。
- 4、最后，在合适的时机上报给我们的服务器。

通过上述的优化，我们就可以知道在整个卡顿周期之内，究竟是哪些方法在执行，哪些方法比较耗时。
但是这种海量卡顿堆栈的处理又存在着另一个问题，那就是高频卡顿上报量太大，服务器压力较大，这里我们来分析下如何减少服务端对堆栈信息的处理量。
在出现卡顿的情况下，我们采集到了多个堆栈，大概率的情况下，可能会存在多个重复的堆栈，而这个重复的堆栈信息才是我们应该关注的地方。我们可以对一个卡顿下的堆栈进行能hash排重，找出重复的堆

栈。这样，服务器需要处理的数据量就会大大减少，同时也过滤出了我们需要重点关注的对象。对于开发人员来说，就能更快地找到卡顿的原因。

4、小结

在本节中，我们学习了自动化卡顿检测的原理，然后，我们使用这种方案进行了实战，最后，我介绍了这种方案的问题和它的优化思路。

三、总结

在本篇文章中，我们主要对卡顿优化分析方法与工具
、自动化卡顿检测方案及优化相关的知识进行了全面的讲解，这里再简单总结一下本篇文章涉及的两大主题：

- 1、卡顿优化分析方法与工具：背景介绍、卡顿分析方法之使用shell命令分析CPU耗时、卡顿优化工具。
- 2、自动化卡顿检测方案及优化：卡顿检测方案原理、AndroidPerformanceMonitor实战及其优化。

下篇，笔者将带领大家更加深入地去学习卡顿优化的相关知识，敬请期待~

参考链接：

1、国内Top团队大牛带你玩转Android性能分析与优化 第6章 卡顿优化
2、极客时间之Android开发高手课 卡顿优化
3、《Android移动性能实战》第四章 CPU
4、《Android移动性能实战》第七章 流畅度
5、Android dumpsys cpuinfo 信息解读
6、如何清楚易懂的解释“UV和PV " 的定义?
7、nanoscope-An extremely accurate Android method tracing tool
8、DroidAssist-A lightweight Android Studio gradle plugin based on Javassist for editing bytecode in Android.
9、lancet-A lightweight and fast AOP framework for Android App and SDK developers
10、MethodTraceMan-用于快速找到高耗时方法，定位解决Android App卡顿问题
11、Linux环境下进程的CPU占用率
12、使用 ftrace
13、profilo-A library for performance traces from production
14、ftrace 简介
15、atrace源码
16、AndroidAdvanceWithGeektime / Chapter06
17、AndroidAdvanceWithGeektime / Chapter06-plus