

1 常用工具介绍

我们将介绍 Systrace 工具，traceview, adb, DeviceAnalyzer，tsensor等常用的工具。

Tools	Precondition	Install	Where to find
Systrace	SDK tool	http://developer.android.com/tools/sdk/tools-notes.html	Android SDK toolkit
Hierarchy, trace, pixel view	SDK tool	http://developer.android.com/tools/sdk/tools-notes.html	Android SDK toolkit
Adb	SDK tool	http://developer.android.com/tools/sdk/tools-notes.html	Android SDK toolkit
DeviceAnalyzer	QCOM	Windows	QCOM provided
tsensor	QCOM	Binary	QCOM provided

1.1 Systrace 工具 及其使用

在android中有许多调试UX的工具Systrace就是其中最重要的工具之一.请参考下面的文档怎样使用systracet工具来debug 性能问题。在给高通提性能相关的case 时，Systrace log 是必须的。

<http://developer.android.com/tools/debugging/systrace.html>

<http://developer.android.com/tools/help/systrace.html>

1.1.1 工具下载和安装

请通过下面的连接下载和安装Android SDK. 需要注意的是我们需要及时更新我们的Android SDK , 以便systrace log 能匹配 Android 的版本和其分析工具 (Chrome)

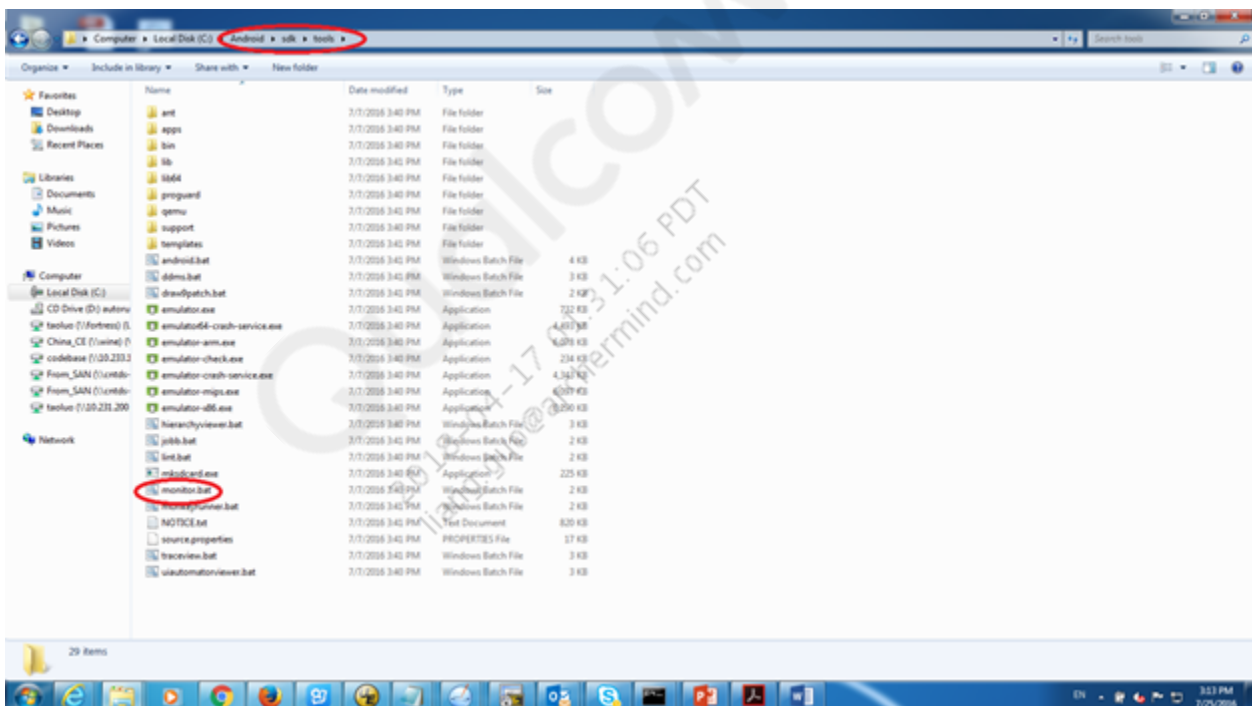
<https://developer.android.com/studio/releases/sdk-tools.html>

1.1.2 Monitor 使用

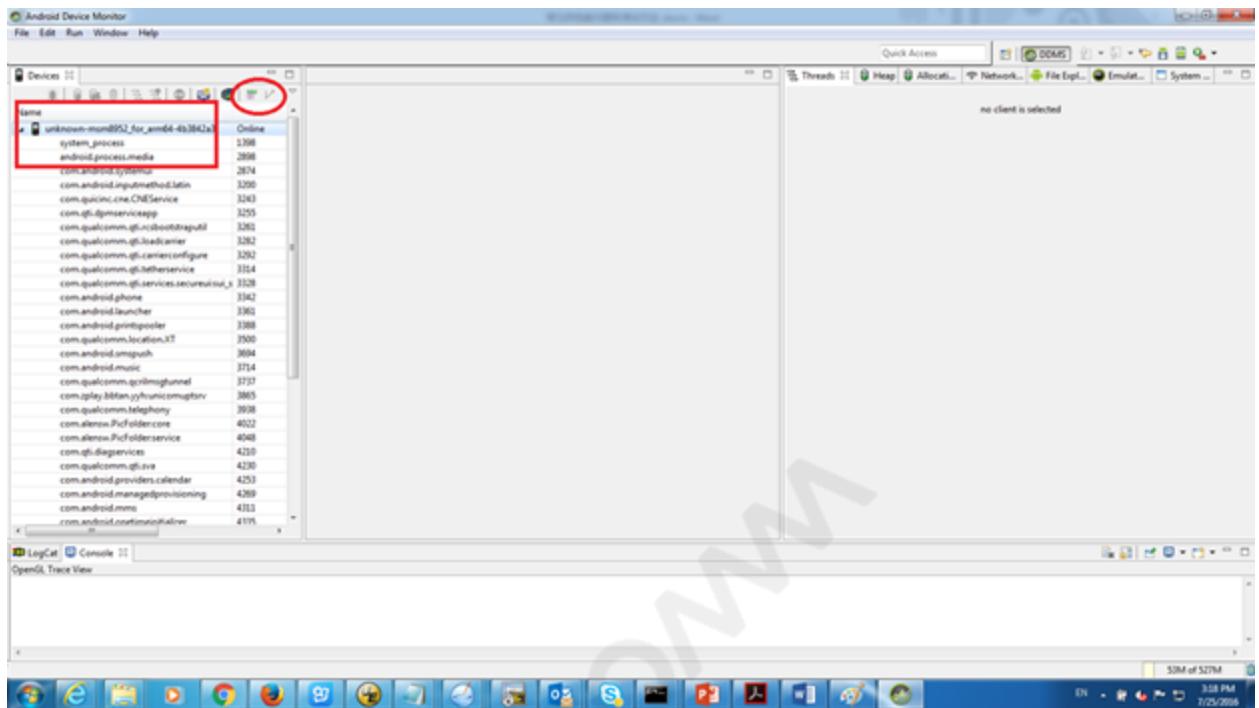
Android 中的很多debug工具都已经整合到Monitor中 , Systrace 工具也不例外。这里介绍monitor的使用。

○ 启动Monitor

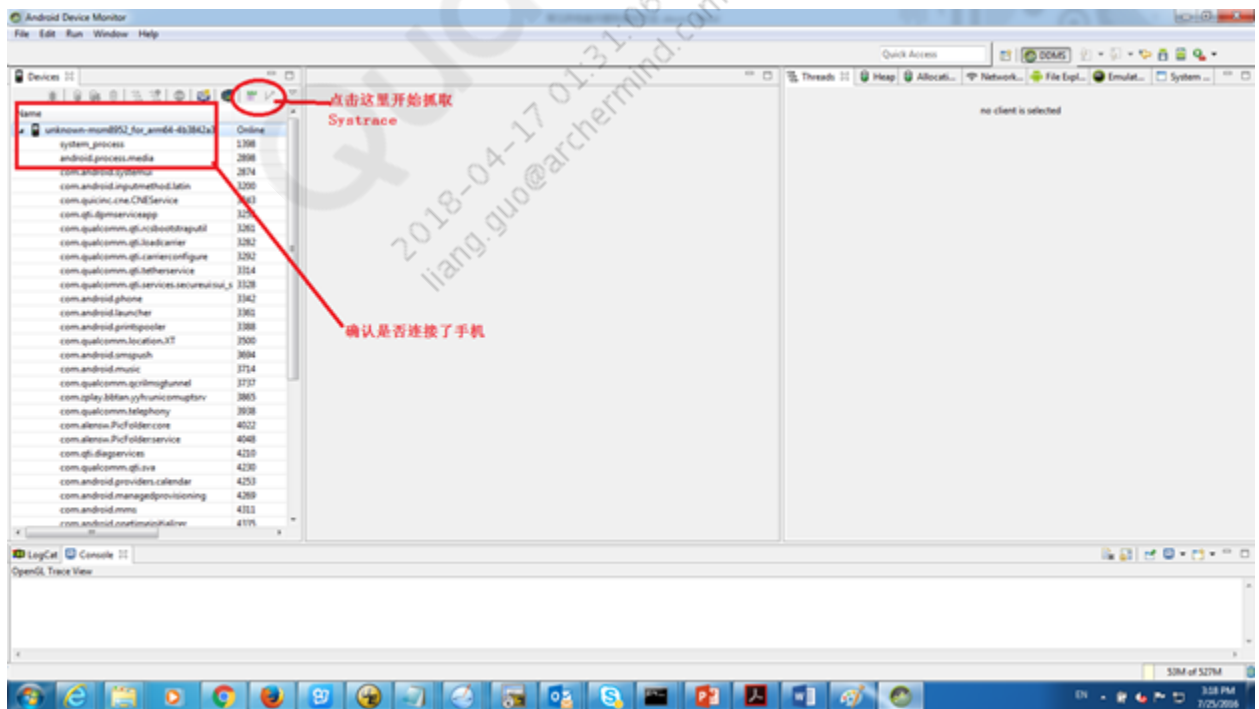
安装完Android SDK并下载platform tools后 , 在SDK\tools的目录下即可看到Monitor工具 :



双击monitor.bat 就可以启动monitor。启动后的Monitor 如下所示

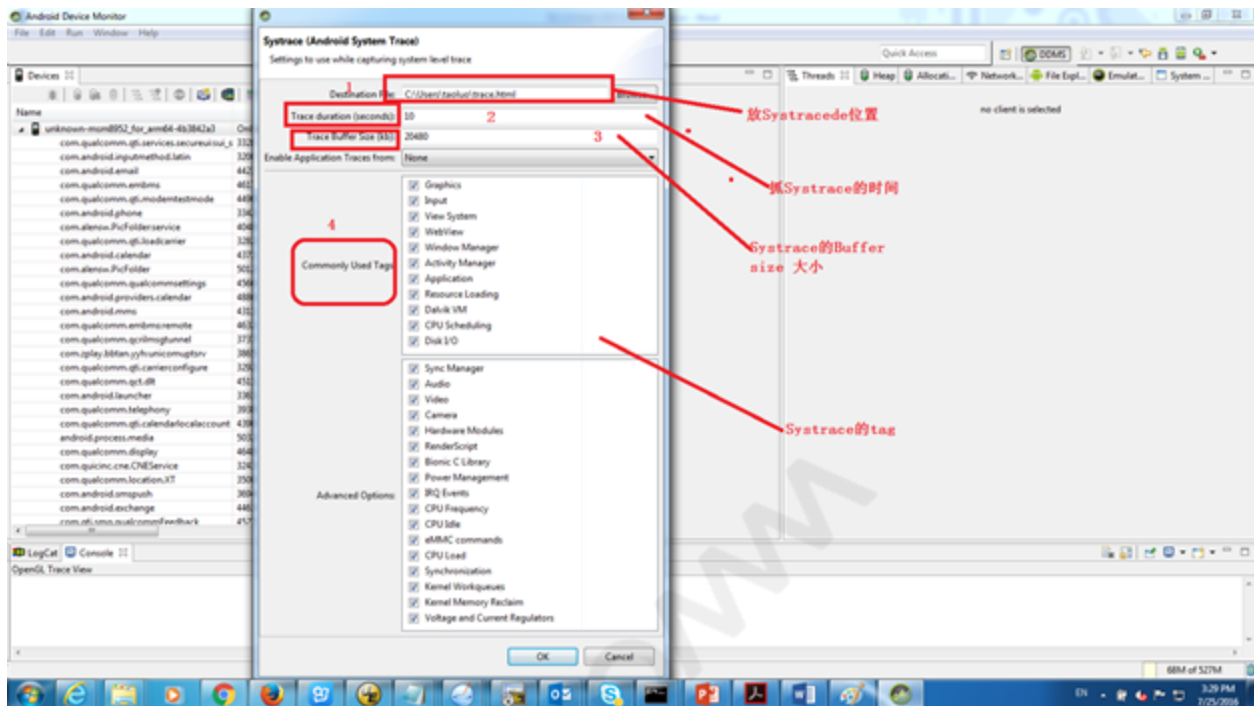


1.1.3 Window Systrace配置和抓取



在抓取systrace之前，我们需要确认已经连接上手机，如上图所示。

- Systrace的配置



点击抓取systrace的图标后，我们就可以看到systrace的配置，如果上图所示。
其配置的含义如下：

1：Systrace输出的文件路径

2：配置抓取systrace的时间，其默认值是5秒。我们可以改变其值来满足我们抓log的要求。
其原则是，需要在设定的时间内，能复现问题并能抓取log。时间太短会导致问题重现时没有被抓到，时间太长会导致内存不够而无法保存，因此在能抓到问题点的情况下，时间越小越好。

3：Buffer Size是存储systrace的buffer size。太小会导致信息丢失，太大会导致手机内存不够而无法抓取，一般情况下，建议20480（20M）。

4：需要抓取的tags,一般情况下，我们可以全选。注意不同的Android SDK其tags是有所不同的，为了获取足够多的log，我们需要选择足够多的tags,简单来说我们可以选择全部的tags。

○ Systrace的抓取

在设置完成后点击"OK"开始抓取systrace，然后请重现问题，当抓取log的时间到了设置的时间，trace.html文件会被自动保存下来。

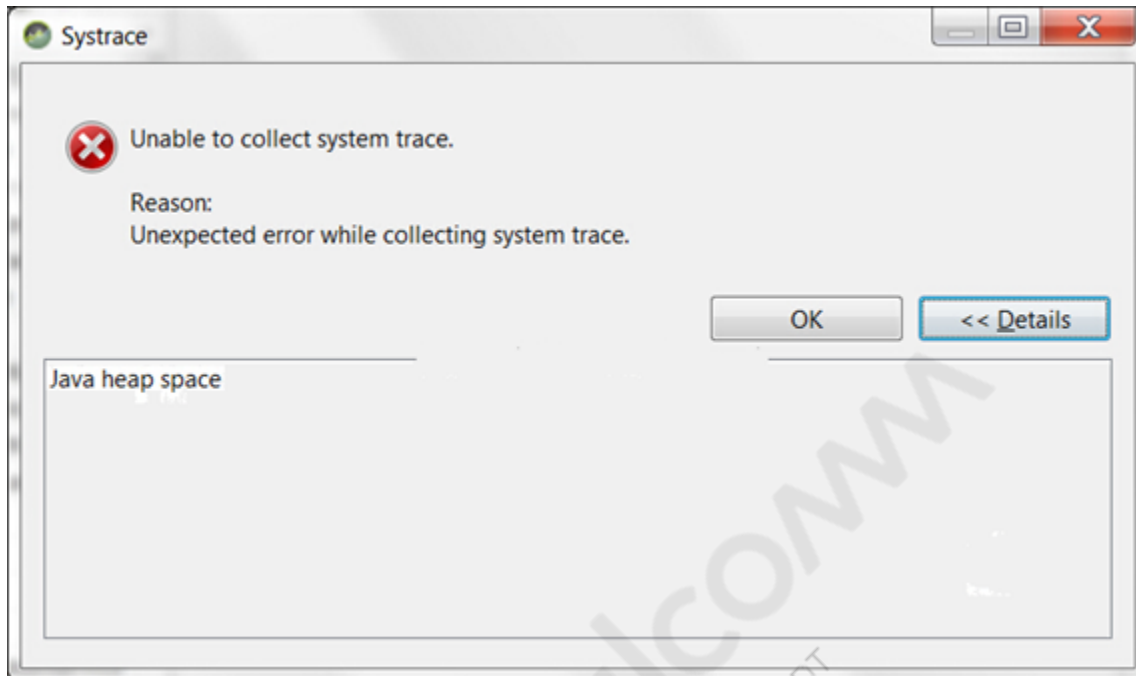
注意：需要先开始抓systrace,然后再重现问题，而不是当问题出现时，再抓Systrace。

○ Systrace文件的有效性

按照前面的步骤，我们应该可以抓到systrace，但是我们需要确认systrace log 的有效性。一般来说"Did Not Finish"的systrace log 是无效的，需要重新抓取。

1.1.4 常见错误

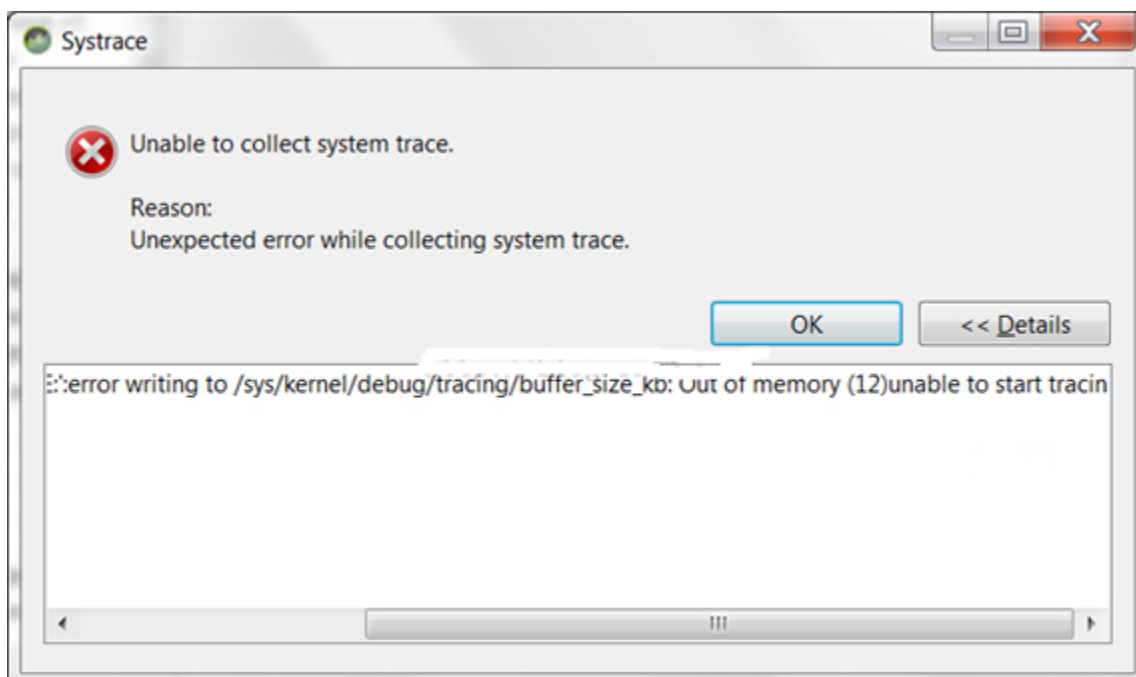
- Monitor Java heap space



此问题是由于Monitor的 jvm的heap 太小导致，可以做如下修改即可。

修改sdk\tools\lib\monitor-x86_64\monitor.ini 或者sdk\tools\lib\monitor-x86\ monitor.ini 的-Xmx1024m 为-Xmx4096m即可。（JVM 从1G 到 4G ）。

- 手机内存不够



如果出现上图的错误，则大多数情况是手机当前内存不够导致，可以减小systrace buffer size 或清除后台不用的app，或者重启手机以便获得比较多的空闲内存。

1.1.5 Python 脚本获取systrace

如果通过python 获取systrace，我们建议通过下面的脚本进行抓取。

```
systrace.py gfx rs input view sched am wm camera dalvik freq idle load sync workq power mmc  
disk sm audio hal video app res binder_driver binder_lock -b 20480 -t 10 -o trace.html
```

-b trace buffer 大小，上面设置为 20M。可以修改如果buffer size 不够

-t 持续时间，上面设置为10s。

注意：不同的Android SDK 版本其tags是有所不同的。我们可以用下面的命令来获取相关的tags

```
systrace --list-categories
```

1.1.6 用atrace获取systrace

有些性能问题只有在不连接USB的时候才能重现问题，我们就需要利用atrace 来获取离线的systrace。请参考下面的文档来进行离线离线systrace的获取。

<https://createpoint.qti.qualcomm.com/search/contentdocument/stream/dcn/KBA-160803171155>

1.2 TraceView工具 及其使用

Traceview是android平台性能debug的又一个常用的工具。

它可以通过图形化的方式让我们了解我们要跟踪的程序的性能，并且能具体到每个方法执行的时间。与Systrace 工具相同，TraceView 工具已经集成到Android的Monitor中。

1.2.1 抓取TraceView log

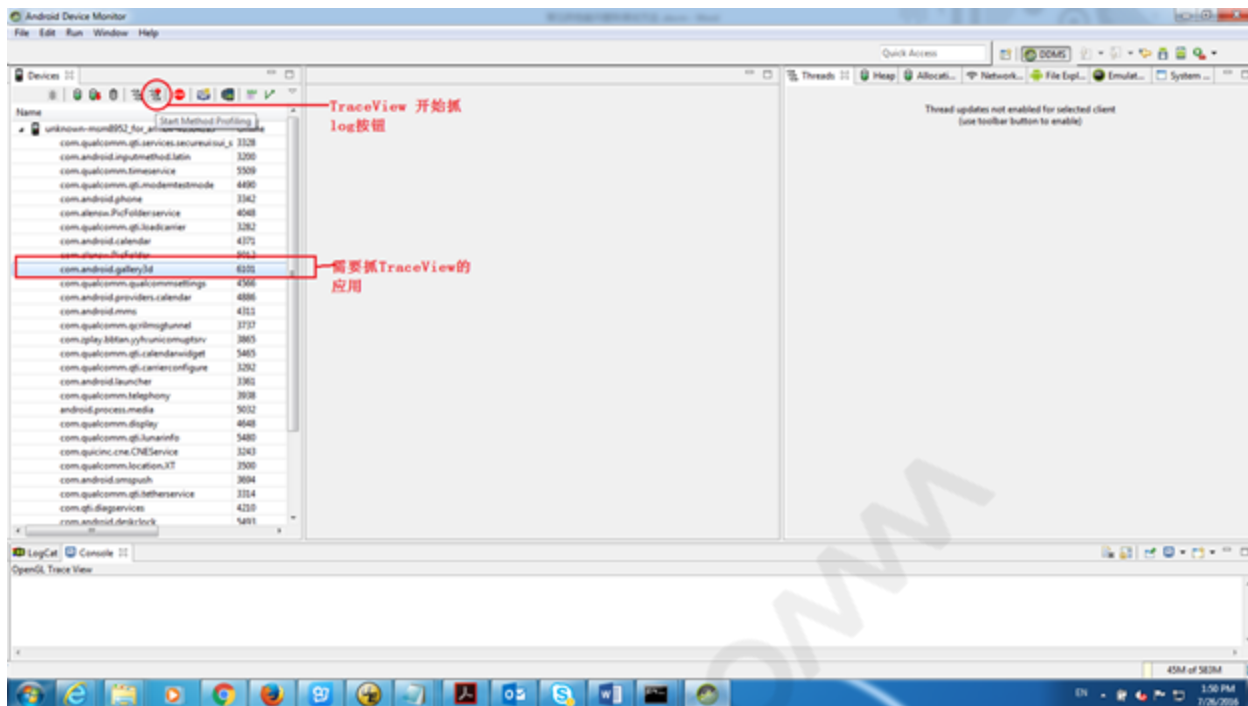
可以参考 Android 的开发文档 怎样抓取和使用TraceView。

<https://developer.android.com/studio/profile/traceview-walkthru.html>

<https://developer.android.com/studio/profile/traceview.html>

这里我们介绍通过Monitor来抓取TraceView log

- 启动Monitor

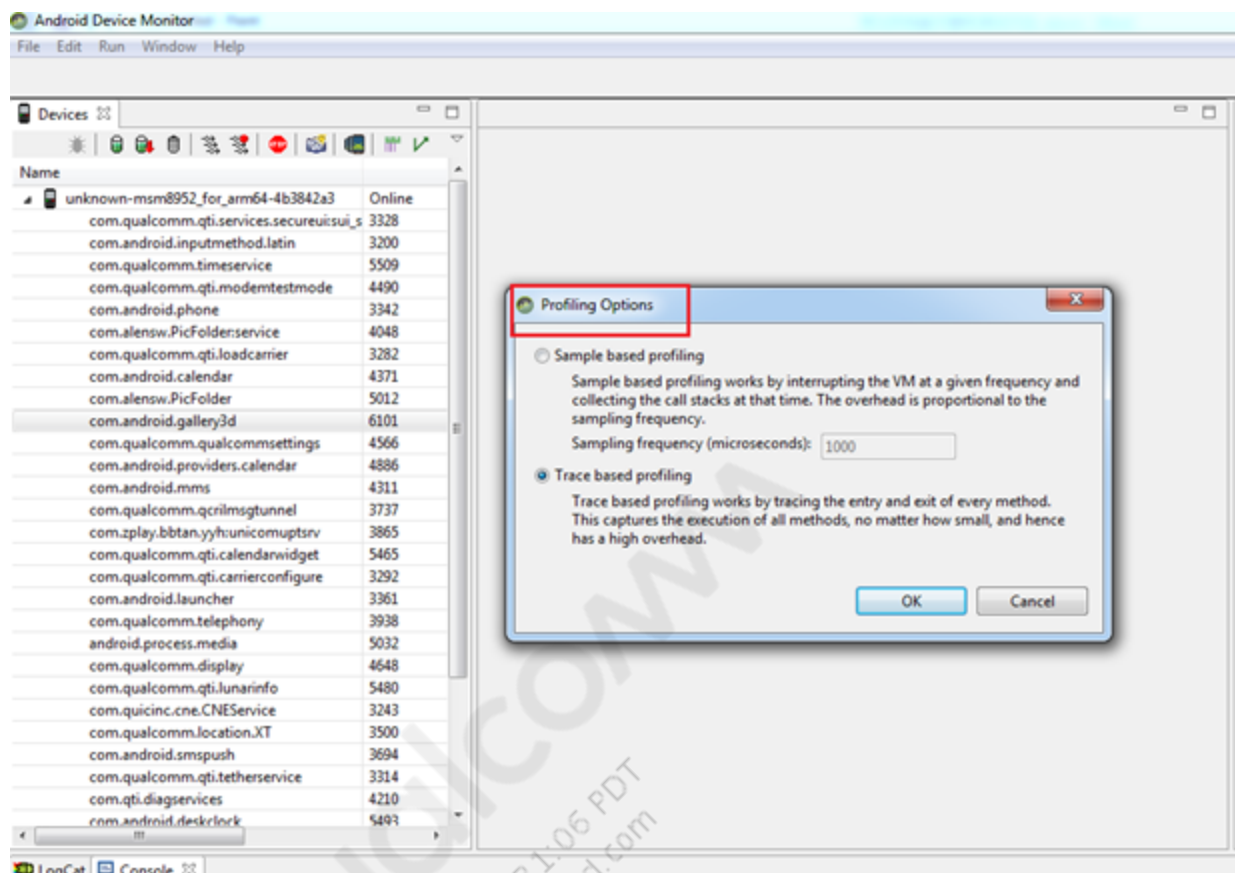


在抓取Ttraceview之前我们需要在Monitor中选择需要获取Traceview log的进程。例如上图所选中的是com.android.gallery3d,然后点击 TraceView开始抓取log 按钮。

○ TraceView 配置

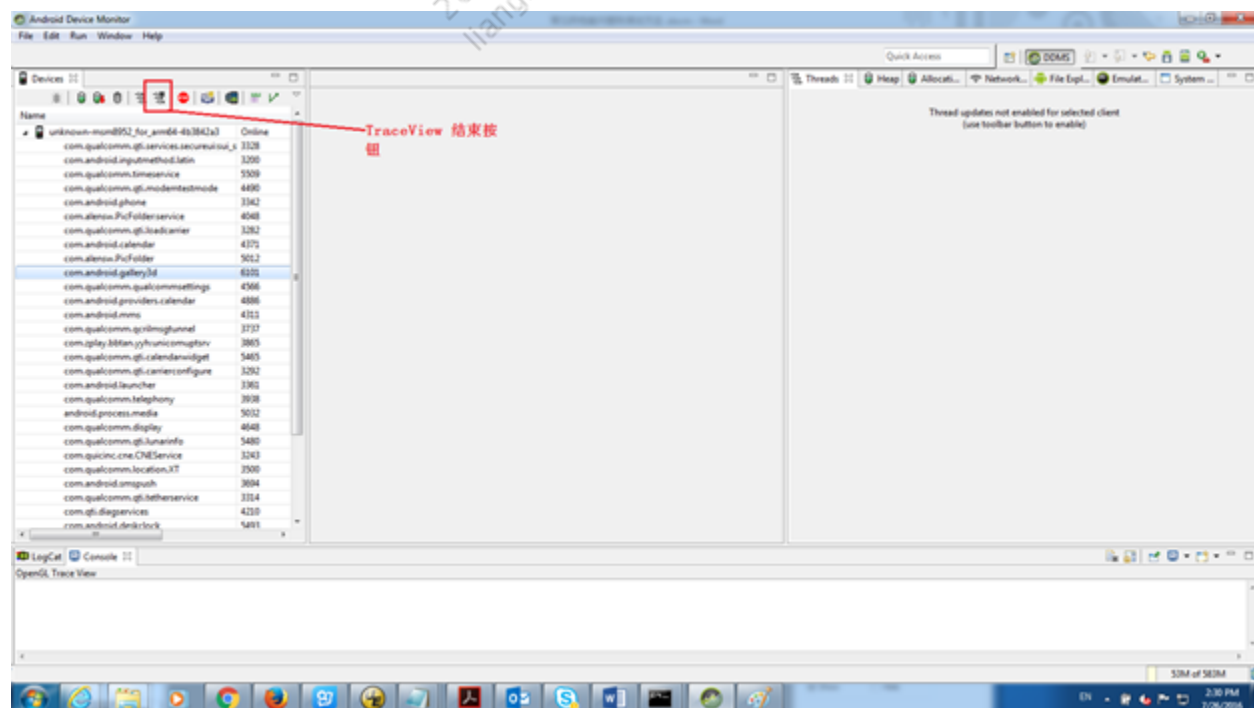


当我们点击开始抓TraceView log 的按钮后，Monitor便会显示 TraceView 的配置选择界面，如下图所示。这里有两种配置选择，一是采样式，每隔一个采样周期进行一次方法的profiling.另一个是Trace base的 profiling，其是对每个方法都进行profiling。在抓取TraceView的log时，我们推荐选择"Trace based profiling"如下图所示。



○ 抓取Traceview LOG

在选择好"Profiling Options"后，点击OK 按钮开始抓取log，如下图所示。

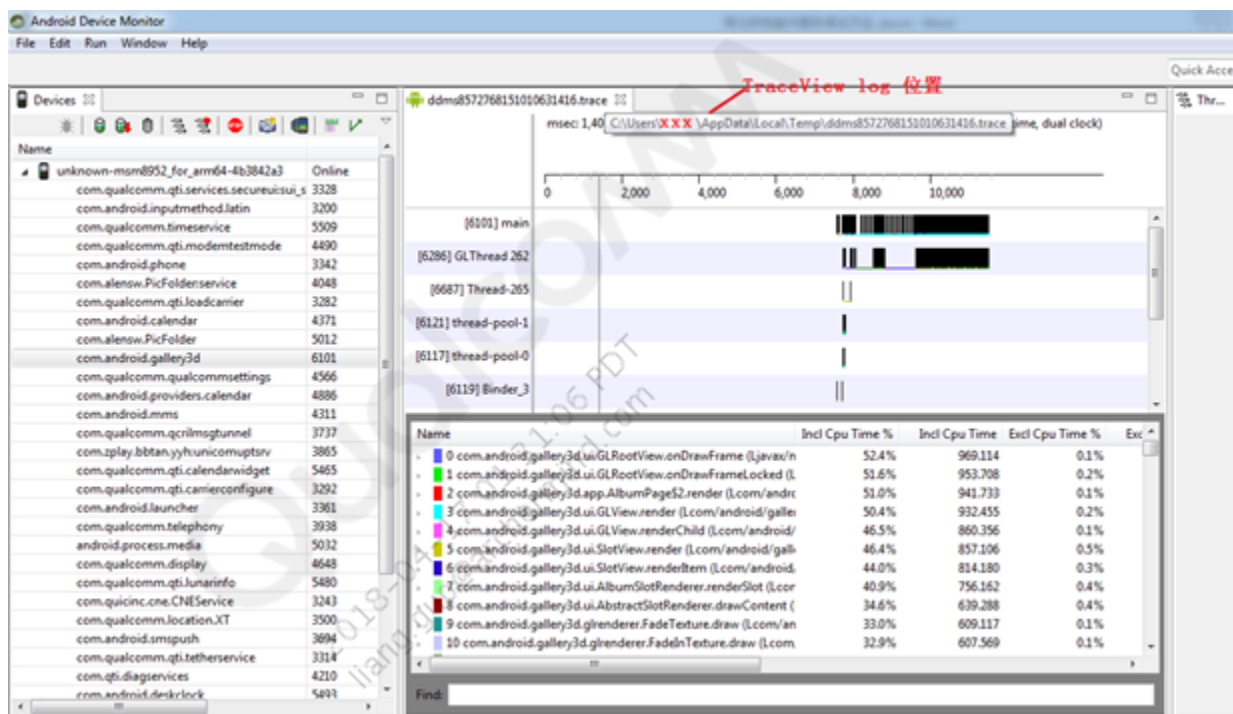


在抓TraceView的过程中，我们可以做相关的操作。当我们做完所有的操作后，我们就可以结束

TraceView log的抓取。点击  按钮，结束 log的抓取。

○ 获取TraceView log

当我们点击结束按钮后，我们会看到如下的画面。如果我们把鼠标移动到TraceView log名称附近如下图"ddms857227xxxxx.trace"便会显示其所在的位置，我们可以把获取其log。



○ Adb command 抓取traceview log

我们有时也可以通过adb command来抓取traceview log

启动：adb shell am profile <PROCESS> start <FILE>

关闭：adb shell am profile <PROCESS> stop

<PROCESS> 填写进程名,例如AndroidManifest.xml中声明的包名，通常都是主进程名。

如：

adb shell am profile com.android.gallery3d start /data/local/tmp/galler3d.trace

做相关的操作，然后stop

adb shell am profile com.android.galler3d.stop

我们就可以直接pull出traceview log.这种方式需要<PROCESS>的app已经运行。

```
adb shell am start -n <Package Name>/<Package Name>.<Activity Name> --start-profiler <FILE>
```

例如：

```
adb shell am start -n com.example/com.example.MainActivity --start-profiler /data/local/tmp/example.trace
```

我们可以用这个command来profiling某些app 启动时的性能问题。但是需要在其activity的方法中添加 profiling的代码否则将得到一个空的文件。如：

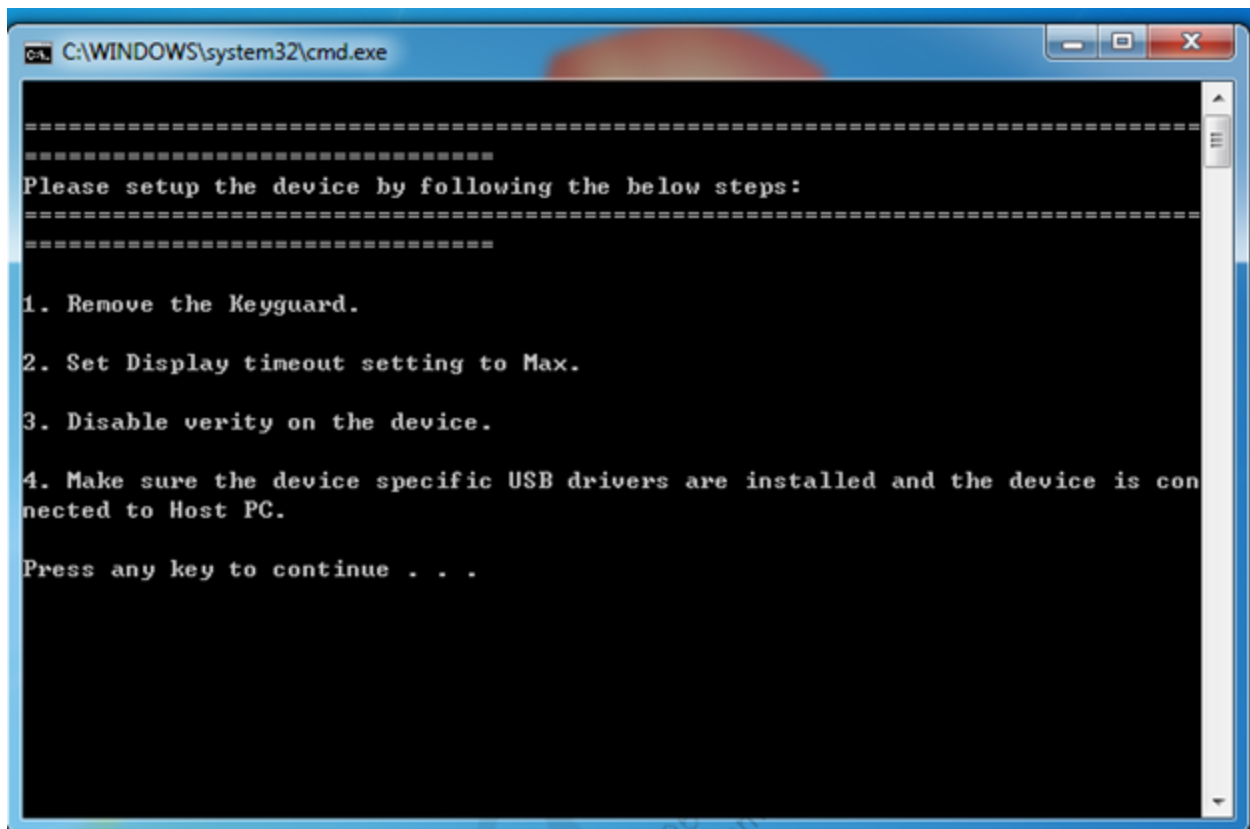
```
1. public class MainActivity extends Activity {  
3. @Override  
4. protected void onCreate(Bundle savedInstanceState) {  
5. super.onCreate(savedInstanceState);  
6. setContentView(R.layout.activity_main);  
8. Debug.startMethodTracing("MainActivity Trace" ); //trace begin  
9. }  
11. @Override  
12. protected void onStart() {  
13. super.onStart();  
15. Debug.stopMethodTracing(); //trace end  
16. }  
18. }
```

1.3 DeviceAnalyzer工具 及其使用

DeviceAnalyzer是QCOM开发的可用于手机基本配置及Sanity test，其运行在windows OS。运行该工具时，要求有root权限。

1.3.1 运行DeviceAnalyzer

运行该tool后显示如下的界面。



```
C:\WINDOWS\system32\cmd.exe

=====
=====
Please setup the device by following the below steps:
=====
=====

1. Remove the Keyguard.

2. Set Display timeout setting to Max.

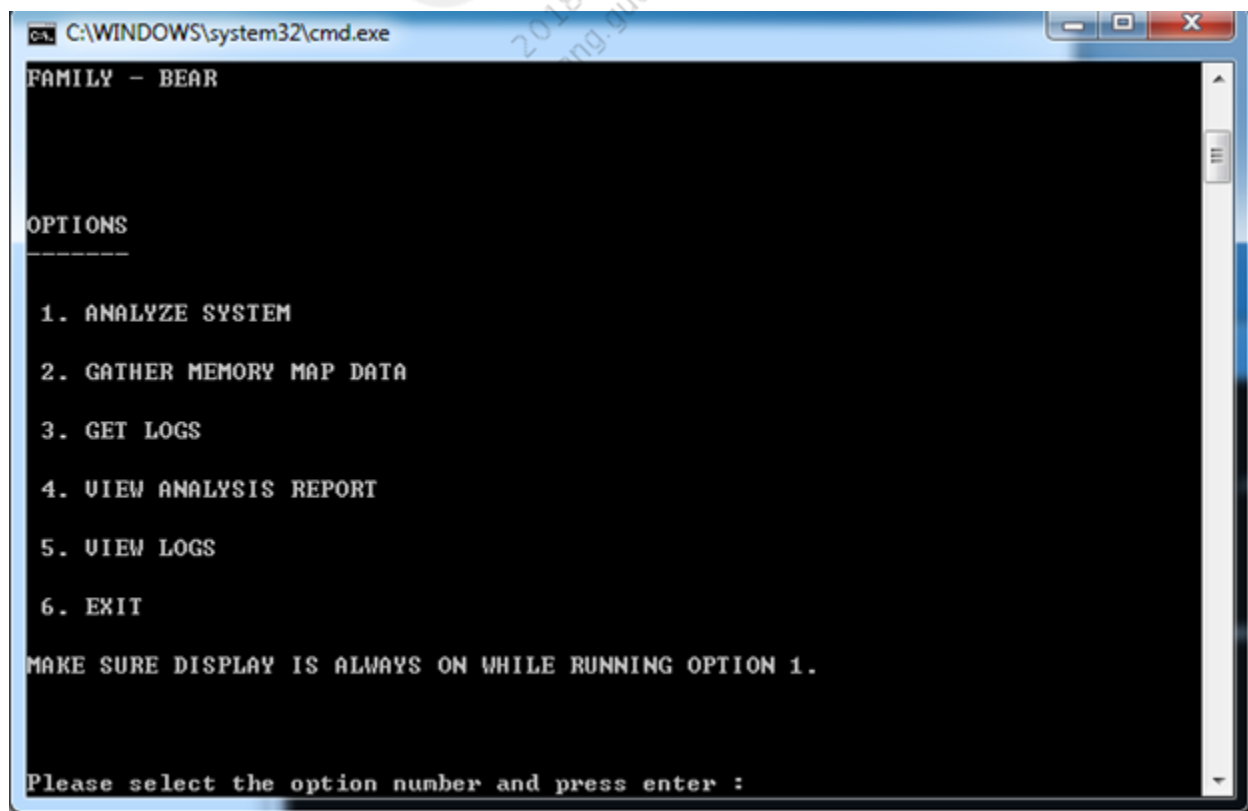
3. Disable verity on the device.

4. Make sure the device specific USB drivers are installed and the device is connected to Host PC.

Press any key to continue . . .
```

- 选择测试选项

如下图所示，请依次选择 1，2 和3 三个测试项。



```
C:\WINDOWS\system32\cmd.exe

FAMILY - BEAR

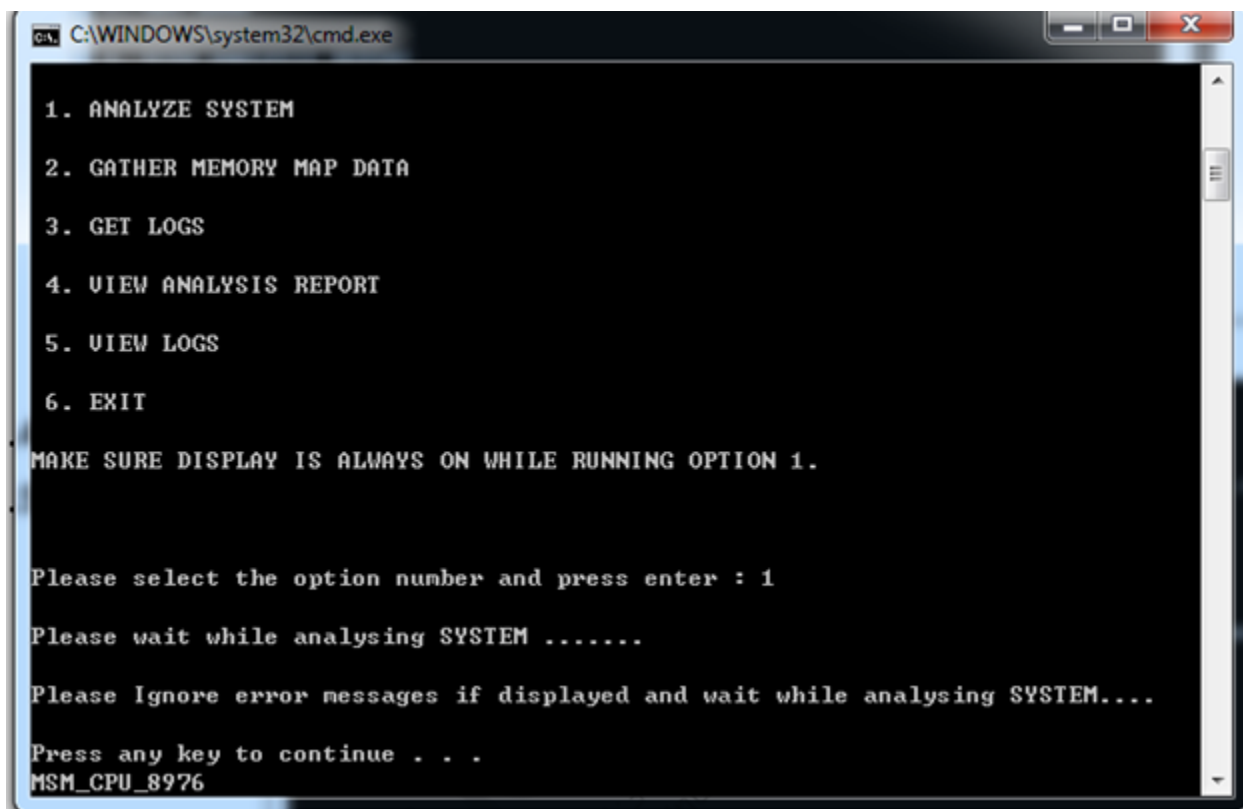
OPTIONS
-----

1. ANALYZE SYSTEM
2. GATHER MEMORY MAP DATA
3. GET LOGS
4. VIEW ANALYSIS REPORT
5. VIEW LOGS
6. EXIT

MAKE SURE DISPLAY IS ALWAYS ON WHILE RUNNING OPTION 1.

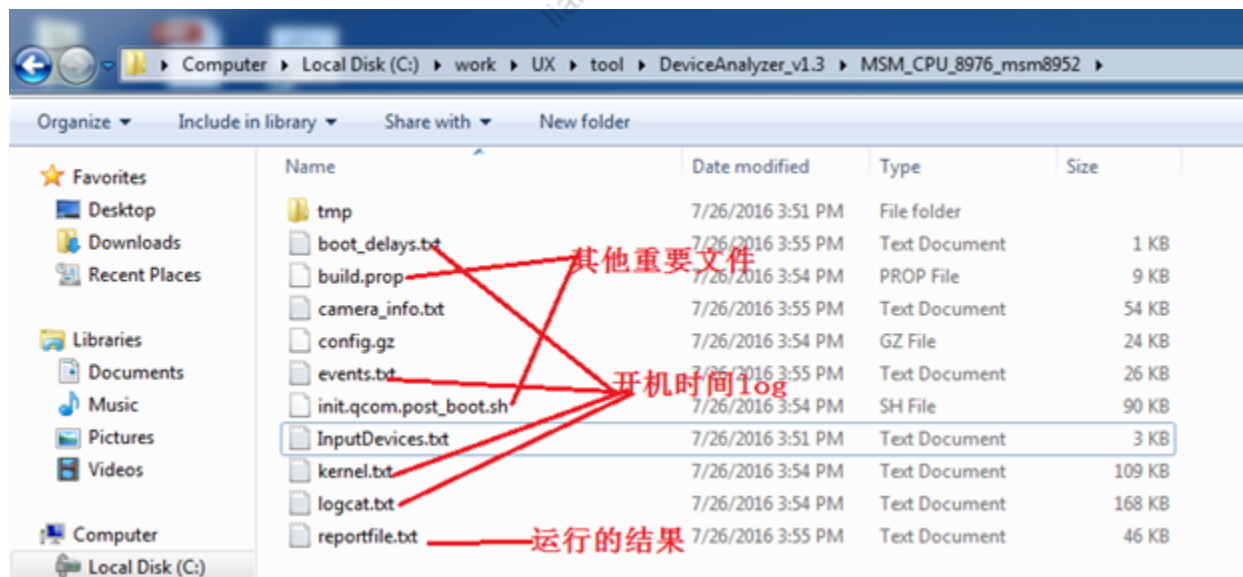
Please select the option number and press enter :
```

○ 测试中



○ 测试后

测试完所有的选项后，我们可以在运行的目录下发现运行后的log。其文件如下图所示。



1.3.2 测试结果分析

通常我们可以通过此工具的测试结果（reportfile.txt）快速判断是否是正确的config 或是否missing了相关的重要patch。下面我们列出几个重要的检查点。

- Kernel 版本

在进行性能方面的测试时，需要用kernel perf config。 我们可以通过检查kernel version 来初步确定是否用了perf config

KERNEL VERSION

Linux version 3.10.84-perf-gb737a83-00002-g8294845 (lnxbuild@abait183-sd-lnx) (gcc version 4.9.x-google 20140827 (prerelease) (GCC)) #1 SMP PREEMPT Thu May 5 07:35:55 PDT 2016

如果出现 perf 如上门的log所示则可以初步确定是perf config，要想进一步确定还需要检查 kernel config，请参考 [Kernel config](#)。

- Kernel Boot Arguments

Kernel Boot Arguments

boot_cpus=0,1,2,3,4,5 sched_enable_hmp=1 console=ttyHSL0,115200,n8
androidboot.console=ttyHSL0 androidboot.hardware=qcom msm_rtb.filter=0x237
ehci-hcd.park=3 androidboot.bootdevice=7824900.sdhci lpm_levels.sleep_disabled=1
earlyprintk androidboot.emmc=true androidboot.verifiedbootstate=orange
androidboot.veritymode=enforcing androidboot.keymaster=1 androidboot.serialno=4b3842a3
androidboot.baseband=msm mdss_mdp.panel=1:dsi:0:qcom,mdss_dsi_nt35597_
wqxda_video:1:qcom,mdss_dsi_nt35597_wqxda_video:cfg:split_dsi

我们可以检查开机启动时，其设置的参数是否是我们所要求的参数，如CPU的数量等。

- ALMK feature

ADAPTIVE LMK

Adaptvie LMK parameter patch is present.

vmpressure_file_min

81250

ALMK是QCOM针对QCOM平台对LMK做的系列优化，强烈建议使能该feature。
目前高通所有平台都默认使能了该feature。 如果vmpressure_file_min 的值不为0
则说明已经使能了。如果为0 则没有使能。如果没有使能则需要检查没有使能的原因。

○ TRIM PARAMETERS

TRIM PARAMETERS

[ro.sys.fw.empty_app_percent]: [50]

[ro.sys.fw.trim_cache_percent]: [100]

[ro.sys.fw.trim_empty_percent]: [100]

[ro.sys.fw.use_trim_settings]: [true]

如果和上面的参数值有差异，请检查是否missing了patch 或则是否调整了相关参数。

○ SWAPPINESS

SWAPPINESS

adb shell cat /proc/sys/vm/swappiness

100

如果其值不是100，请检查是否修改过该值。

○ SCALING GOVERNOR PARAMETERS

SCALING GOVERNOR PARAMETERS

"adb wait-for-device shell cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor"

interactive

```
"adb wait-for-device shell cat /sys/devices/system/cpu/cpu4/cpufreq/scaling_governor"
```

interactive

这里只列出了重要的governor 参数。当前QCOM所有的平台都使采用interactive governor，如果不是interactive governor请检查。

1.4 tsensor工具 及其使用

tsensor 工具是QCOM 开发的来获取 手机内部tsensor 的温度，以及CPU的fmax 和当前CPU的频率。通过该log 我们可以判断是否由于thermal 原因 限制CPU的fmax，从而影响性能。

1.4.1 tsensor使用

- Push tool到手机里

```
adb push <source_to_tsens_logger> /data
```

```
adb shell chmod 777 /data/msm_tsens_logging
```

- 运行

```
adb shell./data/msm_tsens_logging 250 3600000 &
```

这里250是采样频率，3600000是获取log的时间，两者的单位都使ms。

上面的command 其含义是每隔250ms读一次tsensor log，持续1个小时。

- 获取log

其log 存放在/data/tsens_logger.csv。当测试完毕，pull 其log 即可。

```
adb pull /data/tsens_logger.csv <path_to_pull_to>
```

1.5 adb 常用command

1.5.1 adb 通过wifi连接

有些性能问题只有在不连接USB的时候才能重现，这样我们就不能通过USB连接 adb 抓取log。我们可以通过WIFI将 adb 和手机连接起来。其连接方法如下：

- PC和手机都连上同一个WIFI热点
- 进入手机获取手机的地址 (通过usb连接)

adb shell ifconfig 假设其IP地址是 : 192.168.1.1

- 运行下面的命令

```
adb tcpip 5555
```

```
adb connect 192.168.1.1
```

- 断开USB连接。此时应该可以通过adb (WIFI) 连接上手机。

1.5.2 常用的adb性能debug命令

- CPU performance mode

我们知道引起性能问题的因素很多，通常为了初步确认该问题是否是系统处理能力不足而导致的，我们可以让系统运行在 performance mode 下测试该问题是否可以重现，从而进行初步的诊断。

4 CPUs

```
adb shell root
```

```
adb shell setenforce 0
```

```
adb shell stop thermal-engine
```

```
adb shell rmmod core_ctl
```

```
adb shell "echo 1 > /sys/devices/system/cpu/cpu1/online"
```

```
adb shell "echo 1 > /sys/devices/system/cpu/cpu2/online"
```

```
adb shell "echo 1 > /sys/devices/system/cpu/cpu3/online"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu1/cpufreq/scaling_governor"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu2/cpufreq/scaling_governor"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu3/cpufreq/scaling_governor"
```

8 CPUs

```
adb shell root
```

```
adb shell setenforce 0
```

```
adb shell stop thermal-engine
```

```
adb shell rmmod core_ctl
```

```
adb shell "echo 1 > /sys/devices/system/cpu/cpu1/online"
```

```
adb shell "echo 1 > /sys/devices/system/cpu/cpu2/online"
```

```
adb shell "echo 1 > /sys/devices/system/cpu/cpu3/online"
```

```
adb shell "echo 1 > /sys/devices/system/cpu/cpu4/online"
```

```
adb shell "echo 1 > /sys/devices/system/cpu/cpu5/online"
```

```
adb shell "echo 1 > /sys/devices/system/cpu/cpu6/online"
```

```
adb shell "echo 1 > /sys/devices/system/cpu/cpu7/online"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu1/cpufreq/scaling_governor"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu2/cpufreq/scaling_governor"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu3/cpufreq/scaling_governor"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu4/cpufreq/scaling_governor"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu5/cpufreq/scaling_governor"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu6/cpufreq/scaling_governor"
```

```
adb shell "echo performance > /sys/devices/system/cpu/cpu7/cpufreq/scaling_governor"
```

○ GPU performance mode

如果我们发现GPU draw得太慢并且GPU的clock不是运行在比较高的频率的情况下，我们可以让GPU运行在performance mode 下，测试该问题是否可以重现。

```
adb shell root
```

```
adb shell setenforce 0
```

```
adb shell stop thermal-engine
```

```
adb shell echo 0 > /sys/class/kgsl/kgsl-3d0/bus_split
```

```
adb shell echo performance > /sys/class/kgsl/kgsl-3d0/devfreq/governor
adb shell echo 1 > /sys/class/kgsl/kgsl-3d0/force_bus_on
adb shell echo 1 > /sys/class/kgsl/kgsl-3d0/force_rail_on
adb shell echo 1 > /sys/class/kgsl/kgsl-3d0/force_clk_on
adb shell echo 1000000 > /sys/class/kgsl/kgsl-3d0/idle_timer
```

- DDR at max

有时DDR频率过低也会引起性能问题，所以我们可以让DDR工作在比较高的频率下测试是否还可以重现该问题。

```
adb shell "echo 1 > /sys/kernel/debug/msm-bus-dbg/shell-client/mas"
```

```
adb shell "echo 512 > /sys/kernel/debug/msm-bus-dbg/shell-client/slv"
```

```
adb shell "echo 0 > /sys/kernel/debug/msm-bus-dbg/shell-client/ab"
```

```
adb shell "echo 16 * DDR max frequency > /sys/kernel/debug/msm-bus-dbg/shell-client/ib"
```

```
adb shell "echo 1 > /sys/kernel/debug/msm-bus-dbg/shell-client/update_request"
```

- 获取thermal-engine debug log

```
adb shell stop thermal-engine
```

```
adb shell thermal-engine --debug &
```

```
adb shell logcat -v time -s ThermalEngine >your path
```

- 打开perfd debug log

```
adb pull /system/build.prop
```

在build.prop 中增加debug.trace.perf=1

```
adb push build.prop /system/
```

```
adb shell chmod 0644 /system/build.prop
```

```
adb shell sync
```

```
adb shell reboot
```

或则

```
adb shell root
```

adb shell setenforce 0

adb shell setprop debug.trace.perf 1

adb shell stop perfd

adb shell start perfd

perfd的log 就会显示在logcat和systrace 中。

2 常见问题分析和调式

2.1 报case 给QCOM

当出现performance问题时，如果需要高通帮助请通过case系统报case给QCOM，当报case时请选择下面的PA，并附上足够的log。

Case Type Bug/Issue or Tuning/Optimization

Problem Area 1 BSP/HLOS

Problem Area 2 Performance

Problem Area 3 System Performance

2.2 性能优化常见文档

- 通用的debug 文档

80-P0584-1 Common Performance Issues Debugging Guide

80-NJ221-1 Android Memory Leak Analysis Guide

80-NV303-1 MSM8916 MSM8909 Memory Optimization Guidelines

80-NM449-1 Android User Experience Performance Overview Mem Analysis

- 系统性能debug文档

80-NF341-1 Enable Bus Profiling A-Family Customer Devices Using Android

80-NJ799-1 Enable Bus Profiling MSM8974 Newer Chipset Customer Devices Using Android

80-NM328-709 Battery Current Limit and Tuning Overview

80-P0907-1 Profile Manager Architecture

- ART

80-P1017-1 Configuring ART for Pre-Optimization

- Scheduler

80-NV396-18 MSM8996 LA APSS CPU Power MgmtOverview

80-NM328-13 MSM8994 LA Heterogeneous Cluster Architecture Programming Overview

80-NM846-31 MSM8939LA HeterogenCluster Arch & Program Overview

- PerfLock

80-NT384-1 PerflockAPI overview

80-NR256-2 MPCTL feature

- FPS

80-NP885-1 Graphics Power Performance Overview

80-P0397-1 A Fence Sync Object Overview

- Android Boot Time measurement

80-N9266-1 Android Boot Time Measurement

- Android Performance Patch list

80-P3936-2 Performance_Improvement_Patches_Android_Nougat_Builds

80-P3936-1 Performance Improvement Patches Android Marshmallow Builds

80-NT978-1 Performance Improvement Patches AndoridLollipop Builds

2.3 Sanity test

我们在做相关性能测试的时候需要让测试机器工作在"量产"状态，具体来说需要用user 或user debug build,并且关闭相关log。

2.3.1 Kernel config

Kernel有两套config文件，一套用于研发阶段，会使能一些debug功能。例如msm8937_defconfig。另一套用于商业化量产阶段，会关闭debug功能，以确保系统的性能以及功耗。文件名包含perf，例如msm8937-perf_defconfig。在debug 性能相关的问题时，请使用kernel perf config。通常perf config 在kernel\arch\arm64\configs\下。

一般来说我们的kernel config文件需要移出下面的flag.需要注意的是这些flag可能会随着平台或OS的不同而不同，请以 kernel perf config为准。

CONFIG_SCHED_DEBUG

CONFIG_DEBUG_KMEMLEAK

CONFIG_DEBUG_KMEMLEAK_EARLY_LOG_SIZE=400

CONFIG_DEBUG_KMEMLEAK_DEFAULT_OFF

CONFIG_DEBUG_SPINLOCK

CONFIG_DEBUG_MUTEXES

CONFIG_DEBUG_ATOMIC_SLEEP

CONFIG_DEBUG_STACK_USAGE

CONFIG_DEBUG_LIST

CONFIG_FAULT_INJECTION_DEBUG_FS

CONFIG_LOCKUP_DETECTOR

CONFIG_DEBUG_PAGEALLOC

CONFIG_PAGE_POISONING

CONFIG_RMNET_DATA_DEBUG_PKT

CONFIG_MMC_PERF_PROFILING

CONFIG_DEBUG_BUS_VOTER

CONFIG_SLUB_DEBUG

CONFIG_ALLOC_BUFFERS_IN_4K_CHUNK

CONFIG_SERIAL_CORE

CONFIG_SERIAL_CORE_CONSOLE

CONFIG_SERIAL_MSM_HSL

CONFIG_SERIAL_MSM_HSL_CONSOLE

2.3.2 运行DeviceAnalyzer工具

运用DeviceAnalyzer工具进行分析。详见1.3 [DeviceAnalyzer工具 及其使用](#)

2.3.3 Perfd 进程

Perfd是QCOM开发的一个和性能相关的后台程序。在默认情况下，该程序开机自启动，该进程对机器性能至关重要，所以我们要确保该进程已经在后台运行。可以使用下面的命令来检查perfd是否运行。

```
adb shell ps |grep perfd
```

```
root 4326 1 8704 844 futex_wait 7fa7af1984 S /system/vendor/bin/perfd
```

如果有类似上面的输出，则perfd 已经启动，反之，则没有启动需要检查其原因。

2.4 开机启动

2.4.1 获得正确的log

对于开机启动慢的问题的debug，我们需要kernel，event，logcat log，请用如下命令获取log。

```
adb wait-for-device root
```

```
adb wait-for-device
```

```
adb shell dmesg > dmesg.txt
```

```
adb logcat -b events -d > logcat_events.txt
```

```
adb logcat -v thread time -d *:V > logcat.txt
```

2.4.2 Log分析

- kernel log

我们知道kernel可以分为两部分，一是boot loader 部分，一是加载driver 部分。

Boot loader 部分

我们可以用Bootloader的KPI 来计算bootloader的所用的时间。Bootloader KPI的时间会输出到dmsg如：

[0.524325] KPI: Bootloader start count = 20820 //A 为LK 开始时

[0.524334] KPI: Bootloader end count = 231148//B 为LK 结束时间

[0.524341] KPI: Bootloader display count = 36470

[0.524348] KPI: Bootloader load kernel count = 2232

[0.524356] KPI: Kernel MPM timestamp = 254555 // C bootloader 完成时间

[0.524363] KPI: Kernel MPM Clock frequency = 32768 //D clock.

NHLOS 时间： $A/D = 20820/32768=0.63$

LK 时间： $(B-A)/D = (231148 - 20820) = 6.41s$

Bootloader 时间： $C/D - kmsg(C)=254555/32768-0.52=7.24s$

如果boot loader的时间太长，我们需要检查其是否正常。

Kernel driver 部分

如果从kernel初始化到 Zygote启动时间太长，我们可以打开每个module的加载时间，然后找到其耗时比较多的module并优化之。

下面这个patch 可以打开module ini的时间。

```
diff --git a/init/main.c b/init/main.c
```

```
index 7af2174..2d11927 100644
```

```
--- a/init/main.c
```

```
+++ b/init/main.c
```

```
@@ -785,7 +785,7 @@ int __init_or_module do_one_initcall(initcall_t fn)
```

```
if (initcall_blacklisted(fn))
```

```
return -EPERM;
```

```
- if (initcall_debug)
```

```
+ if (1)
```

```
ret = do_one_initcall_debug(fn);
```

```
else
```

```
ret = fn();
```

输出log 如：

```
initcall msm_serial_hsl_init+0x0/0xac returned 0 after 262555 usecs
```

```
initcall fts_driver_init+0x0/0x20 returned 0 after 171317 usecs
```

```
initcall ufs_qcom_phy_qmp_20nm_driver_init+0x0/0x20 returned 0 after 2572 usecs
```

```
initcall ufs_qcom_phy_qmp_14nm_driver_init+0x0/0x24 returned 0 after 1727 usecs
```

```
initcall ufs_qcom_phy_qmp_v3_driver_init+0x0/0x24 returned 0 after 1010 usecs
```

```
initcall ufs_qcom_phy_qrbtc_v2_driver_init+0x0/0x24 returned 0 after 838 usecs
```

○ User space log

Event log

系统启动过程中，我们可以获取event log得到boot event，其含义如下： boot_progress_start / / user space 开始时间

boot_progress_preload_start // Zygote 进程preload 开始时间boot_progress_preload_end // Zygote进程preload 结束时间boot_progress_system_run //System server 开始运行时间
boot_progress_pms_start // Package Scan 开始boot_progress_pms_system_scan_start // System 目录开始scan

boot_progress_pms_data_scan_start //data 目录开始scan

boot_progress_pms_scan_end // package scan 结束时boot_progress_pms_ready// package manager ready

boot_progress_ams_ready // Activity manager ready，这个事件之后便会启动home Activity。

boot_progress_enable_screen// HomeActivity 启动完毕。

当HomeActivity 启动完毕后，系统将检查当前所有可见的window是否画完，如果所有的window（包括wallpaper，Keyguard等）都已经画好，系统会设置属性service.bootanim.exit值为1.而bootanimation 在检查到service.bootanim.exit 属性值为1时，便会结束bootanimation，从而显示home界面。所以我们需要辅助logcat log 来检查bootanimation结束是否正常。

如下面的log

```
14:21:36.716 1455 1607 I boot_progress_enable_screen: 21242
```

```
07-21 14:21:43.230 1014 1772 D BootAnimation: media player is completed.
```

这里从Homeactivity启动完毕到bootanimation退出用了大约6.5s的时间。

我们需要检查这个时间是否正常是否还有优化的空间。

2.4.3 常见Issue

- Kernel config

没有使用kernel perf config，没有关闭串口。

- 开机动画

开机动画没有及时退出。开机动画在检测到service.bootanim.exit 为1时，应该及时退出。

如果出现开机动画没有及时退出的情况，则应该检查开机动画的实现并进行优化。

- App 没有做oat优化

特别是在第一次开机过程中，如果没有做app的oat优化，系统会在第一次开机过程中进行优化，会占用大量时间。如下面的log，当出现此问题时，请在编译时进行oat的优化，可以参考80-P1017-1 文档怎样进行oat优化，避免开机过长。

```
01-07 00:59:50.046 I/dex2oat ( 1154): dex2oat took 14.271s (threads: 4) arena alloc=4KB  
java alloc=7MB native alloc=10MB free=4MB
```

```
01-07 00:59:50.046 I/dex2oat ( 1398): dex2oat took 13.502s (threads: 4) arena alloc=2928B  
java alloc=4MB native alloc=8MB free=4MB
```

```
01-07 00:59:50.212 I/dex2oat ( 3237): dex2oat took 63.748ms (threads: 4) arena alloc=144B  
java alloc=73KB native alloc=332KB free=111KB
```

```
01-07 00:59:56.394 I/dex2oat ( 3588): dex2oat took 277.397ms (threads: 4) arena alloc=0B  
java alloc=27KB native alloc=200KB free=99KB
```

```
01-07 00:59:56.686 I/dex2oat ( 3593): dex2oat took 89.380ms (threads: 4) arena alloc=0B  
java alloc=26KB native alloc=200KB free=99KB
```

```
01-07 01:00:01.711 I/dex2oat ( 3637): dex2oat took 1.252s (threads: 4) arena alloc=456B  
java alloc=4MB native alloc=2MB free=374KB
```

2.5 关机速度

2.5.1 获得正确的log

提供出现问题时的logcat log。

2.5.2 Log分析

检查logcat 中TAG为ShutdownThread的log，例如

```
01-22 01:52:35.289 1319 1584 D ShutdownThread: Notifying thread to start shutdown  
longPressBehavior=1
```

```
01-22 01:52:35.320 1319 3802 I ShutdownThread: Sending shutdown broadcast...
```

```
01-22 01:52:35.472 1319 3802 I ShutdownThread: Shutting down activity manager...
```

```
01-22 01:52:35.564 1319 3802 I ShutdownThread: Shutting down package manager...
```

```
01-22 01:52:35.572 1319 3835 W ShutdownThread: Turning off NFC...
```

```
01-22 01:52:35.576 1319 3835 I ShutdownThread: Waiting for NFC, Bluetooth and Radio...
```

```
01-22 01:52:41.085 1319 3835 I ShutdownThread: NFC turned off.
```

```
01-22 01:52:41.085 1319 3835 I ShutdownThread: NFC, Radio and Bluetooth shutdown  
complete.
```

```
01-22 01:52:41.086 1319 3802 I ShutdownThread: Shutting down MountService
```

```
01-22 01:52:46.875 1319 1815 W ShutdownThread: Result code 0 from  
MountService.shutdown
```

```
01-22 01:52:46.876 1319 3802 E ShutdownThread: Unable to find class  
com.qti.server.power.ShutdownOem
```

```
01-22 01:52:47.377 1319 3802 I ShutdownThread: Performing low-level shutdown...
```

如log所示,ShutdownThread这个线程依次关闭:ActivityManager,PackageManager,NFC,Bluetooth,Radio等模块。

通过时间戳可以看出具体模块关闭所花的时间，则需要针对相关模块进一步分析。

2.5.3 常见Issue

- MountService关闭时间太长

07-06 11:15:12.384 I/ShutdownThread(1402): Shutting down MountService

07-06 11:15:19.111 W/ShutdownThread(1402): Result code 0 from MountService.shutdown

可以修改 system/vold/Utils.cpp 文件中的函数:status_t ForceUnmount(const std::string& path) 3处 sleep(5); 改成 sleep(2)来规避该问题。

- radio turn off 耗时太长

07-01 08:03:07.989 W/ShutdownThread(1032): Turning off radio on Subscription :0

07-01 08:03:07.999 W/ShutdownThread(1032): Turning off radio on Subscription :1

07-01 08:03:07.999 I/ShutdownThread(1032): Waiting for NFC, Bluetooth and Radio...

07-01 08:03:19.969 W/ShutdownThread(1032): Timed out waiting for NFC, Radio and Bluetooth shutdown.

需要检查 radio module是否可以优化。

2.6 Power key 唤醒慢

2.6.1 获得正确的log

该问题可以简单的分为两大部分kernel 唤醒部分和user space 的唤醒部分。我们需要相关的log来进行debug。

- Kernel log

kernel config: CONFIG_PM_SLEEP_DEBUG=y

device\qcom\msmxxx\BoardConfig.mk:

BOARD_KERNEL_CMDLINE += initcall_debug log_buf_len=16M

- User Space

需要抓取Systrace log。在进行systrace 抓取之前，需要运行下面的命令。

adb shell "echo mdss:* >> /d/tracing/set_event"

adb shell "echo 1 > /d/tracing/events/mdss/enable"

adb shell "echo 1 > /d/tracing/events/mdss/tracing_mark_write/enable"

- 抓取Log

- 连接usb线，按下power键，关闭屏幕，等1分钟
- 开始抓kernel 和 adb log

- 开始抓systrace，稍等2s后，按下power键，点亮屏幕
- 如果连接USB后，不能重现该问题，我们就不能用连接USB的方法抓取Systrace log，可以采用不用连接USB来catch systrace log. 请参考 [用atrace获取systrace](#) 怎样用离线的方式获取systrace log.

2.6.2 Log分析

○ Kernel Log

在kernel log 中可以检查下面的关键节点。

dpm_resume_noirq完成

[141.782368] PM: noirq resume of devices complete after 37.011 msecs

dpm_resume_early完成

[141.794448] PM: early resume of devices complete after 7.862 msecs

dpm_resume_end

[141.906757] PM: resume of devices complete after 112.295 msecs

○ Systrace

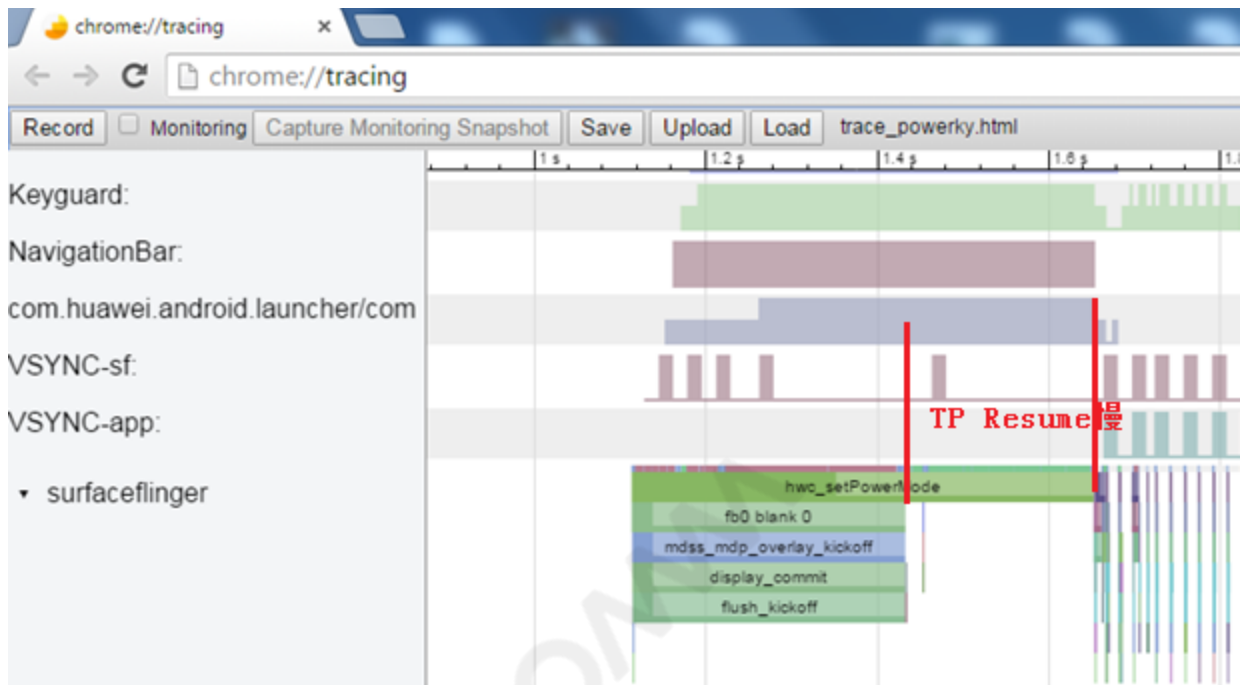
我们可以大致从



2.6.3 常见Issue

○ TP resume 慢

典型的systrace log 如下



请优化TP driver。可以参考下面的code。

<https://codeaurora.org/cgit/quic/la/kernel/msm-3.10/commit/?h=LA.BR.1.3.2&id=dd6e2392f921e3b3368a3e5d4f2885258a48d323>

<https://codeaurora.org/cgit/quic/la/kernel/msm-3.10/commit/?h=LA.BR.1.3.2&id=e291ac12f2e1ce2c0cb269b0b441d74c65a936c6>

2.7 APP 冷启动

2.7.1 获得正确的log

对于启动慢的问题，我们需要logcat，kernel和systrace log.在获取systrace之前,需要打开perfd log

- Enable perfd log，请参考 打开[perfd debug log](#)

2.7.2 Log分析

对于冷启动，主要分析点如下：

- 检查是否正确enable了launch boost功能

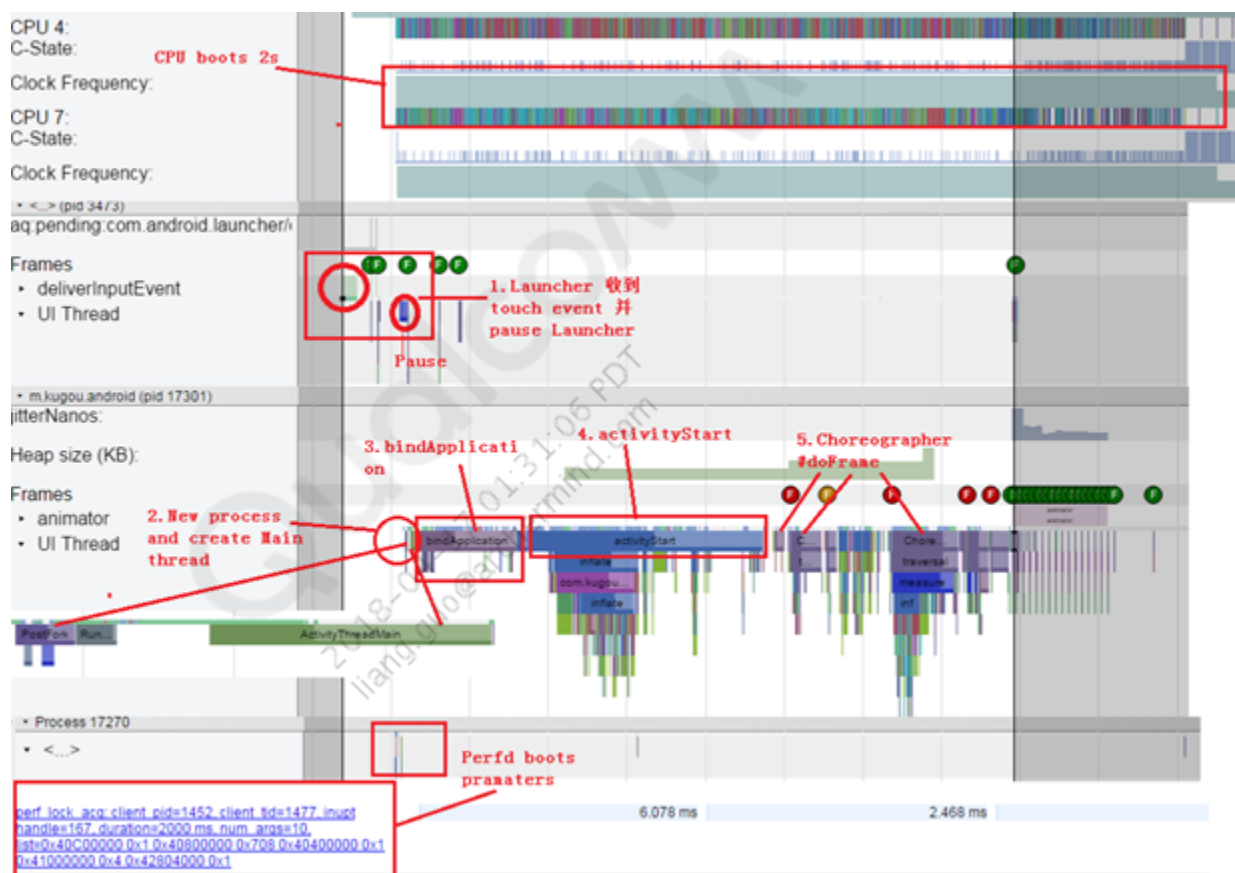
在QCOM所有平台中，对于冷启动，都默认enable了launch boost功能。即在冷启动时，CPU将运行在最大频率上，并且保持2s。

- 启动时间分解

我们知道，在APP在冷启动时，一般的操作是，点击launcher上 APP的图标-> APP启动。
这一过程在systrace 中可以分解为。

Launcher 收到 touch event -> Launcher pause-> new process(APP 进程) -> bindApplication-> activityStart-> Choreographer#doFrame (2~3)

如在下面的 systrace 中，可以看到这些操作。我们可以看看每部分的时间是否合理，
如果不合理则检查相关部分的代码，看看是否有可以优化的空间。



2.7.3 常见Issue

- 没有enable launch boost 功能

这是常见的系统config错误，需要检查是否正确使能了launch boost功能以及perfd是否正常运行。

- APP design 问题

APP 设计太复杂，在UI thread(main thread) 做了太多与UI无关的操作，例如数据库的访问，本地文件访问大量的I/O操作，网络访问等。这需要从APP的角度去解决这些问题。

2.8 机器反应慢 (长时间运行)

2.8.1 获得正确的log

有多种原因可能导致此问题，例如程序本身的健壮性，有时候是程序本身crash导致机器反应慢，网络问题，thermal 等问题。所以当发生此问题时，提供充足的log是必要的。需要提供kernel,adb 和 systrace log.由于可能存在thermal 限制CPU的最大频率，从而影响系统性能，所以在获取log时，最好能提供thermal相关的log 如 tsensor log，请参考 [tsensor工具的使用](#)，怎样获取tsensor log；请参考获取[thermal log](#)，怎样获取thermal log。当发生该问题时，请尽可能提供adb，kernel，systrace,meminfo，thermal log和tsensor log（如有可能）。可以如下command 获取 meminfo log。

```
adb shell dumsys meminfo
```

```
adb shell cat /proc/meminfo
```

```
adb shell dumsys procstats --hours 3
```

```
adb shell dumsys usagestats
```

2.8.2 Log分析

○ Thermal kick-in

如果在我们在logcat里发现了类似下面的log，说明thermal 限制了CPU运行的最大频率，需要检查thermal 是否可以优化。

```
03-23 19:32:45.455 I/ThermalEngine( 366): Mitigation:CPU[0]:800000 Khz
03-23 19:32:45.455 I/ThermalEngine( 366): Mitigation:CPU[1]:800000 Khz
03-23 19:32:45.455 I/ThermalEngine( 366): Mitigation:CPU[2]:800000 Khz
03-23 19:32:45.456 I/ThermalEngine( 366): Mitigation:CPU[3]:800000 Khz
03-23 19:32:48.461 I/ThermalEngine( 366): Mitigation:CPU[0]:533333 Khz
03-23 19:32:48.462 I/ThermalEngine( 366): Mitigation:CPU[1]:533333 Khz
03-23 19:32:48.462 I/ThermalEngine( 366): Mitigation:CPU[2]:533333 Khz
03-23 19:32:48.462 I/ThermalEngine( 366): Mitigation:CPU[3]:533333 Khz
```

○ 网络问题

如果相关应用和网络有关，则可以测试该问题在网络好的情况下能否发生，从而排除网络问题的干扰。

○ LMK

如果机器上运行了大量的APP，在机器可用内存比较少的环境下便会触发LMK。

如下面的log

```
<6>[ 180.396882] lowmemorykiller: Killing 'ualcomm.qct.dlt' (4704), adj 1000,
```

<6>[180.396882] to free 25296kB on behalf of 'kswapd0' (96) because

<6>[180.396882] cache 291536kB is below limit 322560kB for oom_score_adj 1000

<6>[180.396882] Free memory is 12960kB above reserved.

<6>[180.396882] Free CMA is 87864kB

<6>[180.396882] Total reserve is 14216kB

<6>[180.396882] Total free pages is 100824kB

<6>[180.396882] Total file cache is 294420kB

<6>[180.396882] Slab Reclaimable is 17240kB

<6>[180.396882] Slab UnReclaimable is 116920kB

<6>[180.396882] Total Slab is 134160kB

<6>[180.396882] GFP mask is 0xd0

<6>[181.196126] <3>[judege_suspend] lis3dh_acc_enable_set LINE 2470: judege_suspend = 0

<6>[181.529640] lowmemorykiller: Killing '.xxxx(4647)', adj 1000,

<6>[181.529640] to free 24232kB on behalf of 'kswapd0' (96) because

<6>[181.529640] cache 291592kB is below limit 322560kB for oom_score_adj 1000

<6>[181.529640] Free memory is 12088kB above reserved.

<6>[181.529640] Free CMA is 86964kB

<6>[181.529640] Total reserve is 14216kB

<6>[181.529640] Total free pages is 99100kB

<6>[181.529640] Total file cache is 294476kB

<6>[181.529640] Slab Reclaimable is 17240kB

<6>[181.529640] Slab UnReclaimable is 116616kB

<6>[181.529640] Total Slab is 133856kB

<6>[181.529640] GFP mask is 0xd0

2.8.3 常见Issue

- Thermal kick-in

如上面分析如果出现了thermal 限制fmax，请优化thermal。

- System Config

当我们进行性能测试时，首先请检查build是否是user或user debug。不能用 debug 或eng build 来进行性能方面的测试。然后请检查系统的config 是否正确，请参考[Sanity test](#) 怎样进行系统方面的基本设置。

- 没有enable ALMK feature

请参考[ALMK检查](#) 是否enable了 ALMK feature.如果没有请enable。

2.9 卡顿

2.9.1 获得正确的log

卡顿是一种常见性能问题，其涉及到比较多的模块。例如 CPU scheduler,CPU governor ,LPM，HWUI，display 合成，surfaceFinger, openGL ,GPU 性能，UI framework等。我们需要systrace log 来进行初步的分析。

当出现此问题时，请enable下面的event并开始抓取systrace log

```
adb shell "echo sched:sched_migrate_task >> /sys/kernel/debug/tracing/set_event"
```

```
adb shell "echo sched:sched_switch >> /sys/kernel/debug/tracing/set_event"
```

```
adb shell "echo sched:sched_wakeup >> /sys/kernel/debug/tracing/set_event"
```

```
adb shell "echo sched:sched_cpu_load >> /sys/kernel/debug/tracing/set_event"
```

```
adb shell "echo kgs!:* >> /d/tracing/set_event"
```

```
adb shell "echo mdss:* >> /d/tracing/set_event"
```

```
adb shell "echo 1 > /d/tracing/events/mdss/enable"
```

```
adb shell "echo 1 > /d/tracing/events/mdss/tracing_mark_write/enable"
```

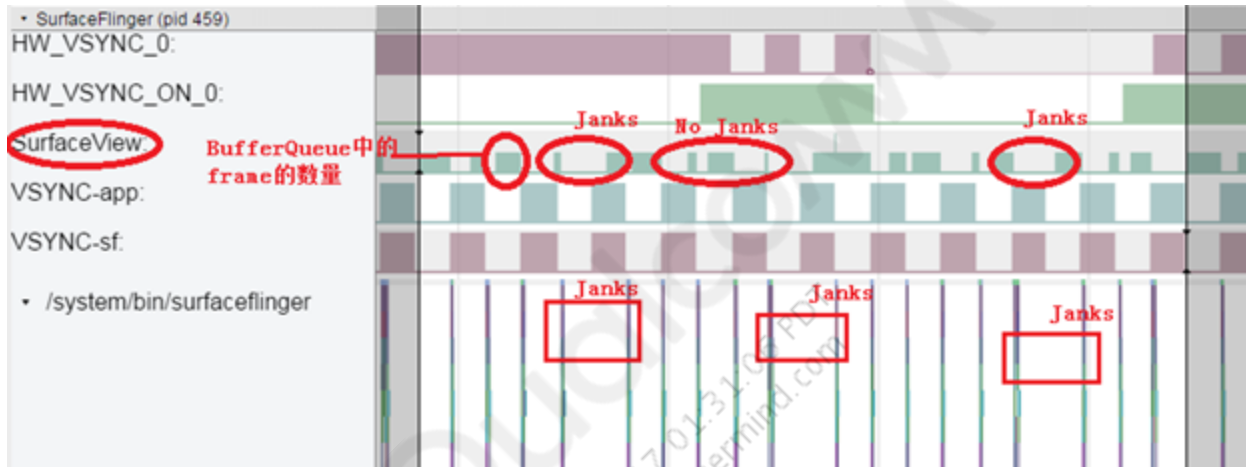
```
adb shell "echo 1 > /d/tracing/events/binder/enable"
```

2.9.2 Log分析

○ Janks

简单来说，如果应用的fps小于60便认为有Janks。可能终端用户就会感觉到卡顿。

我们可以通过systrace 来判断某一应用是否有Janks。如下图 在VSYNC-APP的上方有当前正在合成的app的 view的新的帧数（bufferqueue中的帧数）。有几帧数据，这个值便是几。如 0，表示当前没有新的帧需要更新到LCD上，如果是1表示当前有一帧数据需要更新到LCD上。在通常情况下,0,1是交替出现的。如果连续两个1之间的时间超过16.67ms，则会发生Janks，用户可能就会感觉到卡顿。



总之，该问题比较复杂涉及到的模块比较多，需要systrace log 和相关的log来进行详细的case by case的分析。

2.9.3 常见Issue

○ Build issue

没有用kernel perf config +user build 进行测试。

○ 硬件限制

我们知道手机游戏越来越复杂，比较大型的游戏需要处理能力比较强的CPU和GPU才能胜任。如果CPU或GPU的运算能力不够，则可能会发生此问题。我们可以让CPU和GPU都跑在performance 模式下，看看是否还有此问题，从而简单的判断是否和硬件的处理能力相关。可以参考[CPU performance](#) 和[GPU Performance](#) 如何让机器运行在performance 模式下。

○ App issue

除了分析systrace外，一种简单的方法，我们可以让App 运行在相同或相似的硬件和软件的其他机器上，看看能否发生该问题，从而来确定问题的根源。

2.10 Benchmarks 跑分低

2.10.1 获得正确的log

如果benchmark运行的时间比较长，并且在运行的时候，机器比较烫，我们就需要考虑可能thermal会影响其分数。如果属于这种情况，就需要抓tsensor log. 请参考 [tsensor工具的使用](#) 如何获得tsensor log。

2.10.2 Log分析

由于benchmark比较多且不同的benchmark测试项不同，需要case by case的分析。这里我们看看tsensor log的分析。

- tsensor log 分析

当我们获取了tsensor log 后就可以分析tsensor log，从而确认是否是由于thermal限制了CPU的fmax，进而影响系统性能。

- 获取thermal config

分析tsensor log我们应该首先获取CPU的tsensor配置。可以用下面的cmd来获得。

```
adb shell thermal-engine -o > thermal-engine.conf
```

下面是其CPU tsensor的配置情况，我们略去了thermal-engine的其他部分，只留下和CPU相关的tsensor分布信息。

```
# SENSOR : ALIAS
```

```
# tsens_tz_sensor4 : cpu4 // tsens_tz_sensor4 监控CPU4的温度。
```

```
# tsens_tz_sensor2 : pop_mem
```

```
# tsens_tz_sensor15 : gpu // tsens_tz_sensor15 监控GPU的温度
```

```
# tsens_tz_sensor13 : L2_cache_0
```

```
# tsens_tz_sensor12 : cpu3 // tsens_tz_sensor12监控CPU3的温度。
```

```
# tsens_tz_sensor11 : cpu2 // tsens_tz_sensor11监控CPU2的温度。
```

```
# tsens_tz_sensor10 : cpu1 // tsens_tz_sensor10监控CPU1的温度。
```

```
# tsens_tz_sensor9 : cpu0 // tsens_tz_sensor9 监控CPU0的温度。
```

```
# tsens_tz_sensor8 : L2_cache_1
```


70	960
70	960
70	960
50	1401
42	1401
42	1401
85	0
85	0
80	0
75	960

2.10.3 常见Issue

- Build issue

没有用kernel perf config +user build 进行测试。

- 硬件限制

有些Benchmarks与硬件性能有比较大的关系如测试eMMC的benchmarks。这需要参考硬件的spec来确认是否是issue。

- Thermal Issue

需要优化thermal来规避该问题。

2.11 ART 相关参数和更改

2.11.1 JIT系统属性

`dalvik.vm.usejit - true|false`

是否启用 JIT。

`dalvik.vm.jitinitialsize - 默认 64K`

代码缓存的初始容量。代码缓存将定期进行垃圾回收 (GC) 并且可以视情况增加缓存。

`dalvik.vm.jitmaxsiz -默认 64M`

代码缓存的最大容量。

`dalvik.vm.jitthreshold - 默认 10000`

此项属性是 JIT 编译方法时需要的"热度"计数器传递的阈值。"热度"计数器是运行时的内部度量标准。它包括调用次数、后向分支及其他因素。

`dalvik.vm.usejitprofiles true|false`

是否启用 JIT 配置文件；当 `usejit` 为 `False` 时也可以使用此属性。

`dalvik.vm.jitprithreadweight - 默认 $\text{dalvik.vm.jitthreshold} / 20$`

应用界面线程的 JIT"示例" (参见 `jitthreshold`) 的权重。用于加速方法的编译。在与应用交互时，这些方法会直接影响用户体验。

`dalvik.vm.jittransitionweight - 默认 $\text{dalvik.vm.jitthreshold} / 10$`

在编译代码和解释器转换之间选择调用的方法的权重。

这有助于确保所涉及的方法在编译时尽可能减少转换的量 (转换需要很大开销)。

2.11.2 pm.dexopt系统属性

在 Android 7.0 中以及以后的版本中，有基于不同的用例来指定编译/验证级别的通用方式。例如，安装时间的默认选项是仅进行验证 (并将编译推迟到后期)。编译级别可通过系统属性进行配置。请注意，我们强烈建议您使用 `pm.dexopt` 默认设置，因为它是经过Google以及QCOM测试后得到的参数，这些值可能随Android的版本变化而变化。

`pm.dexopt.install=interpret-only`

为了加快安装速度，我们建议 `interpret-only`

`pm.dexopt.bg-dexopt=speed-profile`

这是当设备处于空闲状态和在充电以及充满电时所使用的编译过滤器。我们建议您使用 `speed-profile`，以利用配置文件引导的编译并节省存储空间。

pm.dexopt.ab-ota=speed-profile

这是在进行 A/B OTA 更新时使用的编译过滤器。如果设备支持 A/B OTA，我们建议使用 speed-profile，以利用配置文件引导的编译并节省存储空间。

pm.dexopt.nsys-library=speed

pm.dexopt.shared-apk=speed

pm.dexopt.forced-dexopt=speed

pm.dexopt.core-app=speed

您可以使用这些不同的选项来控制如何从根本上编译被其他应用使用的应用。对于此类应用，我们建议使用 speed 过滤器。

pm.dexopt.first-boot=interpret-only

在设备初次启动时使用的编译过滤器。此时使用的过滤器只会影响出厂后的启动时间。对于这种情况，我们建议使用 interpret-only 过滤器，以免用户在首次使用时需要很长时间才能开始使用手机。

pm.dexopt.boot=verify-profile

OTA更新后使用的编译过滤器。对于此选项，我们强烈建议使用 verify-profile，以免更新时间过久。

2.11.3 dalvik.vm 属性

dalvik.vm.heapstartsize=8m

相当于虚拟机的 -Xms配置，该项用来设置堆内存的初始大小。

dalvik.vm.heapgrowthlimit=192m

相当于虚拟机的 -XX:HeapGrowthLimit配置，该项用来设置一个标准的应用的最大堆内存大小。一个标准的应用就是没有使用android:largeHeap的应用。

dalvik.vm.heapsize=512m

相当于虚拟机的 -Xmx配置，该项设置了使用android:largeHeap的应用的最大堆内存大小。

dalvik.vm.heaptargetutilization=0.75

相当于虚拟机的 -XX:HeapTargetUtilization,该项用来设置当前理想的堆内存利用率。其取值位于0与1之间。当GC进行完垃圾回收之后，Dalvik的堆内存会进行相应的调整，通常结果是当前存活的对象的大小与堆内存大小做除法，得到的值为这个选项的设置，即这里的0.75。

dalvik.vm.heapminfree=4m与 dalvik.vm.heapmaxfree=8m

dalvik.vm.heapminfree对应的是-XX:HeapMinFree配置，用来设置单次堆内存调整的最小值。

dalvik.vm.heapmaxfree 对应的是-XX:HeapMaxFree配置，用来设置单次堆内存调整的最大值。通常情况下，还需要结合上面的 -XX:HeapTargetUtilization的值，才能确定内存调整时，需要调整的大小。

在 dex2oat 编译启动映像时(boot.art和boot.oat)的属性：

dalvik.vm.image-dex2oat-Xms=64m：初始堆大小

dalvik.vm.image-dex2oat-Xmx=64m：最大堆大小

dalvik.vm.image-dex2oat-filter (verify-none|verify-at-runtime|verify-profile|interpret-only|time|space-profile|space|balanced|speed-profile|speed|everything-profile|everything) 编译器过滤器选项。默认值是speed。

dalvik.vm.image-dex2oat-threads：要使用的线程数。默认值是online的cpu数量。

需要注意的在QCOM的平台中，都进行了预优化，所以这些属性不会用到。

在 dex2oat 编译除启动映像之外的属性：

dalvik.vm.dex2oat-Xms=64m：初始堆大小

dalvik.vm.dex2oat-Xmx=512m：最大堆大小

dalvik.vm.dex2oat-filter：(verify-none|verify-at-runtime|verify-profile|interpret-only|time|space-profile|space|balanced|speed-profile|speed|everything-profile|everything) 编译器过滤器选项，默认值是speed。

dalvik.vm.boot-dex2oat-threads：启动时要使用的线程数，默认值是online的cpu数量。

dalvik.vm.dex2oat-threads：启动后要使用的线程数，默认值是online的cpu数量。

Android 7.1 及之后的版本提供了两个选项来控制编译除启动映像之外的所有内容时的内存使用方式：

dalvik.vm.dex2oat-very-large：停用 AOT 编译的最小总 dex 文件大小（以字节为单位）

dalvik.vm.dex2oat-swap：使用 dex2oat 交换文件（用于低内存设备）

2.11.4 更改属性

注意我们不建议OEM更改这些值，如果需要更改，请做充分的测试。可以在产品的 device.mk 中来更改。这里以8953 平台64bit build 为例。

更改文件：device/qcom/msm8953_64/msm8953_64.mk

更改内容：

```
PRODUCT_PROPERTY_OVERRIDES += \
```

```
dalvik.vm.dex2oat-filter=interpret-only \
```

```
dalvik.vm.image-dex2oat-filter=speed \
```

```
dalvik.vm.heapminfree=4m \
```

```
dalvik.vm.heapstartsize=16m
```

这样我们就更改了这些属性的值，然后重新编译，就可生效。