01-29 11:34:10.444 2984-2994/dji.pilot.pad W/art: Suspending all threads took: 38.273ms

01-29 11:34:10.472 2984-3041/dji.pilot.pad E/UsbManager: RemoteException in getAccessoryList

android.os.DeadObjectException

at android.os.BinderProxy.transactNative(Native Method)

at android.os.BinderProxy.transact(Binder.java:496)

at android.hardware.usb.IUsbManager$Stub$Proxy.getCurrentAccessory(IUsbManager.java:395)

at android.hardware.usb.UsbManager.getAccessoryList(UsbManager.java:296)

at dji.midware.usb.P3.DJIUsbAccessoryReceiver.b(Unknown Source)

at dji.midware.usb.P3.DJIUsbAccessoryReceiver.a(Unknown Source)

at dji.midware.usb.P3.DJIUsbAccessoryReceiver$1.handleMessage(Unknown Source)

at android.os.Handler.dispatchMessage(Handler.java:98)

at android.os.Looper.loop(Looper.java:135)

at android.os.HandlerThread.run(HandlerThread.java:61)

连上飞机卡死

25s

正常

自动重启，数据恢复，还是卡

1. 不进入FPV呢？

手动重启，然后不连接飞机&不进入fpv：1min后，正常。进入fpv，黑屏。多次按快捷键，系统自动变为fpv界面（上次的快照？）此时点击界面没有任何反应。dji.pilot.pad死了，adb没死。

adb reboot之后出现问题

# QA

## 已有规律

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | FPV | 版本 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

复现所需要的时间：连飞机<进入图传界面<插SD卡< 普通操作

：重新同一个系统版本也恢复不了

点击“enter device”进入DJI界面，放置半分钟左右，再次点击屏幕界面无反应

1. ：提供pad@v0.2.1.0固件包（bug）：在切分支install之后无法复现？

连上飞机呢？卡死

2.重新刷机还是不正常，不联飞机也会卡死，不进入图传界面也会卡死

## 和FPV有关

0341&&FPV

## 版本无关？

### P4P

|  |  |  |
| --- | --- | --- |
| **P4P** | 不良品（卡屏） | 良品 |
| 版本范围 | ( 0.2.1.0, 0.3.4.1,) | 0.0.7.2 |
| 打包时间 | (170804,180101) |  |
| 开发日志 | 156的服务器编译的分支gl300e\_0.2.1.0 |  |
| **CS5** | **不良品（卡屏）0.25%** | **良品** |
| 版本范围 | (0.2.2.0, ,) | 0.1.0.0 |
| 开发打包时间 | (181106，) | (170920，) |
| 工厂导入时间 | (170927，) |  |
| **CS7** | **不良品（卡屏）1.857%** | **良品** |
| 版本范围 | 0.2.7.0 |  |
| 开发打包时间 | 181215 |  |
| 工厂导入时间 |  |  |

## 跟sd卡有关？

## 工厂问题

P4P带屏幕我也有一些数据，其它数据郭淋稍后补充一下   
1.直接点击遥控器的界面会出现卡屏幕的现象---概率极低  
2.与飞机对频 进入图传界面 马上出现卡死 ----概率非常高  
3.产线上周每天2%左右的不良 （目前产线堆积100+不良）

不良的安兔兔测试可以通过

Guolin

1.CS系统（CrystalSky）出现该问题的最大版本是:5.5寸显示屏0.2.2.0，7.85寸显示屏0.2.7.0，  
2.重新刷机还是不正常，不联飞机也会卡死，不进入图传界面也会卡死  
3.不出现该问题的版本号是2.3.0.0（2017.9.27导入）  
4.5.5寸显示屏卡屏不良率是0.25%；7.85寸高亮屏显示屏卡屏不良率是1.857%；7.85寸高亮屏显示屏卡屏不良率是1.5%；  
5.已在两周前提供一台7.85寸高亮屏卡屏不良品给天空了

## 其他同事分析

app\_process，所以go里面去获取binder获取service卡主了



# ****死机定义****

**死机:** 当手机长时间无法再被用户控制操作时，我们称为死机。在这里我们强调长时间，如果是短时间，归结为性能问题。

死机表现:

* 用户操作手机无任何响应，如触摸屏幕，按键操作等。
* 手机屏幕黑屏，无法点亮屏幕。
* 手机界面显示内容和用户输入不相干。

## 系统简图

当用户对手机进行操作时，对应的数据流将是下面一个概括的流程图

\* HW 如传感器，触摸屏(TP)，物理按键(KP)等感知到用户操作后，触发相关的中断(ISR)传递给Kernel，Kernel相关的driver 对这些中断进行处理后，转化成标准的InputEvent。

\* UserSpace 的System Server中的Input System则持续监听Kernel传递上来的原始InputEvent，对其进行进一步的处理后，变成上层APP可直接处理的Input Event，如button点击，长按，滑动等等。

\* APP 对相关的事件进行处理后，请求更新相关的逻辑界面，而这个则由System Server中的WMS 等来负责。

\* 相关的逻辑界面更新后，则会请求SurfaceFlinger来产生FrameBuffer数据，SurfaceFlinger则会利用GPU 等来计算生成。

\* DisplaySystem/Driver 则会将FrameBuffer中的数据更新显示出来，这样用户才能感知到他的操作行为。

## 可能死机的原因

原则上上面流程中，每一步出现问题，都可能引发死机问题。大的方面讲，可以分成硬件HW和软件SW两个层次，硬件HW不在我们的范围之内。

软件SW上，死机的原因可以分成两种:

(1). 逻辑行为异常

\*\* 逻辑判断错误

\*\* 逻辑设计错误

(2). 逻辑卡顿(block)

\* 死循环 (Deadloop)

\* 死锁 (Deadlock)

从具体的原因上将，可以进一步分成:

(1). InputDriver

\* 无法接收HW的中ISR，产生原始的InputEvent或者产生的InputEvent异常。

分析死机、触屏无响应的问题的时候，第一步要先看看有没有inputEvent上来，即有没有报点，各个平台inputEvent的设备节点都不一样，可以通过下面方法获取：

adb shell getevent

这个命令下下去以后会在屏幕上输出当前设备的节点，可从name一行判断哪一个是TW，当然也可以在此时直接在屏幕上触摸看有没有时间上报，正常情况如下：

可以看出event2是触摸事件。

异常情况下触摸是没有事件上报的。

当然也可以直接下adb shell getevent /dev/input/event2检查。

(2). InputSystem

无法监听Kernel传递上来的原始InputEvent，或者转换与传递异常。

(3). SystemLogic

无法正常响应InputSystem传递过来的InputEvent，或者响应出错。

(4).WMS/Surfaceflinger 行为异常

WMS/ Surfaceflinger 无法正确的对Window进行叠加转换

(5).Display System

无法更新Framebuffer数据，或者填充的数据错误

(6). LCMDriver

无法将Framebuffer数据显示在LCM上

对应硬件HW异常，经常见得的情况有:

\*Power

\* Clock

\* Memory& Memory Controller

\* Fail IC

# 数据分析

死机分析，同样需要获取第一手的资料，方可分析问题.那么哪些数据可以用来分析死机呢?

大概的讲，可以分成空间数据和时间数据。空间数据，即当时现场环境，如有哪些process在运行，CPU 的执行情况，memory 的利用情况，以及具体的process的memory 数据等。时间数据，即行为上的连续数据，比如某个Process在一段时间内执行了哪些操作，某段时间内CPU利用率的变化等。通常时空都是交融的，对应我们抓取log时往往也是。

## Backtrace

Backtrace 又分成Java backtrace，Native Backtrace，Kernel Backtrace。它是分析死机的非常重要的手段，借助Backtrace，我们可以快速的知道，对应的process/thread在当时正在执行哪些动作，卡住哪里等。可以非常直观的分析死机现场。

### Java Backtrace

从Java Backtrace，我们可以知道当时Process的虚拟机执行状态。 JavaBacktrace依靠SignalCatcher来抓取。

Googledefault: SignalCatcher catchs SIGQUIT(3)， and thenprint the java backtrace to /data/anr/trace.txt

MTKEnhance:  SignalCatcher catchs SIGSTKFLT(16)， and thenprint the java backtrace to /data/anr/mtktrace.txt( After 6577.SP/ICS2.MP)

可以通过修改系统属性dalvik.vm.stack-trace-file改变trace文件路径， 默认路径为/data/anr/traces.txt

#### **几种常见的java backtrace**

##### ActivityThread 正常状态的Backtrace

发给

essageQueue is empty， and thread wait fornext message.

 "main" prio=5 tid=1 NATIVE

   | group="main"sCount=1 dsCount=0 obj=0x4193fde0 self=0x418538f8

   | sysTid=11559 nice=0sched=0/0 cgrp=apps/bg\_non\_interactive handle=1074835940

   | state=S schedstat=(2397315020 9177261498 7975 ) utm=100 stm=139 core=1

   at android.os.MessageQueue.nativePollOnce(NativeMethod)

   at android.os.MessageQueue.next(MessageQueue.java:138)

   at android.os.Looper.loop(Looper.java:150)

   at android.app.ActivityThread.main(ActivityThread.java:5299)

   at java.lang.reflect.Method.invokeNative(NativeMethod)

   at java.lang.reflect.Method.invoke(Method.java:515)

   at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:829)

   at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:645)

   at dalvik.system.NativeStart.main(NativeMethod)

##### Monitor case

SynchronizedLock：等待同步锁时的backtrace.

 "AnrMonitorThread" prio=5tid=24 MONITOR

   | group="main"sCount=1 dsCount=0 obj=0x41fd80c8 self=0x551ac808

   | sysTid=711 nice=0 sched=0/0cgrp=apps handle=1356369328

   | state=S schedstat=(8265377638 4744771625 6892 ) utm=160 stm=666 core=0

   at com.android.server.am.ANRManager$AnrDumpMgr.dumpAnrDebugInfoLocked(SourceFile:~832)

   - waiting to lock<0x42838968> (a com.android.server.am.ANRManager$AnrDumpRecord) held bytid=20 (ActivityManager)

   at com.android.server.am.ANRManager$AnrDumpMgr.dumpAnrDebugInfo(SourceFile:824)

   at com.android.server.am.ANRManager$AnrMonitorHandler.handleMessage(SourceFile:220)

   at android.os.Handler.dispatchMessage(Handler.java:110)

   at android.os.Looper.loop(Looper.java:193)

   at android.os.HandlerThread.run(HandlerThread.java:61)

##### 执行JNI code未返回，状态是native的情况

"WifiP2pService" prio=5tid=37 NATIVE

   | group="main"sCount=1 dsCount=0 obj=0x427a9910 self=0x55f088d8

   | sysTid=734 nice=0 sched=0/0cgrp=apps handle=1443230288

   | state=S schedstat=( 91121772135245305 170 ) utm=7 stm=2 core=1

   #00  pc 00032700 /system/lib/libc.so (epoll\_wait+12)

   #01  pc 000105e3 /system/lib/libutils.so (android::Looper::pollInner(int)+94)

   #02  pc 00010811 /system/lib/libutils.so (android::Looper::pollOnce(int， int\*， int\*， void\*\*)+92)

   #03  pc 0006c96d /system/lib/libandroid\_runtime.so(android::NativeMessageQueue::pollOnce(\_JNIEnv\*， int)+22)

   #04  pc 0001eacc /system/lib/libdvm.so (dvmPlatformInvoke+112)

   #05  pc 0004fed9 /system/lib/libdvm.so (dvmCallJNIMethod(unsigned int const\*， JValue\*， Method const\*， Thread\*)+484)

   #06  pc 00027ea8 /system/lib/libdvm.so

   #07  pc 0002f4b0 /system/lib/libdvm.so (dvmMterpStd(Thread\*)+76)

   #08  pc 0002c994 /system/lib/libdvm.so (dvmInterpret(Thread\*， Methodconst\*， JValue\*)+188)

   #09  pc 000632a5 /system/lib/libdvm.so (dvmCallMethodV(Thread\*， Methodconst\*， Object\*， bool， JValue\*，std::\_\_va\_list)+340)

   #10  pc 000632c9 /system/lib/libdvm.so (dvmCallMethod(Thread\*， Methodconst\*， Object\*， JValue\*， ...)+20)

   #11  pc 00057961 /system/lib/libdvm.so

   #12  pc 0000dd40 /system/lib/libc.so (\_\_thread\_entry+72)

   at android.os.MessageQueue.nativePollOnce(NativeMethod)

   at android.os.MessageQueue.next(MessageQueue.java:138)

   at android.os.Looper.loop(Looper.java:150)

   at android.os.HandlerThread.run(HandlerThread.java:61)

##### 执行object.wait等待状态

"AsyncTask #1" prio=5tid=33 WAIT

   | group="main"sCount=1 dsCount=0 obj=0x427a8480 self=0x56036b40

   | sysTid=733 nice=10sched=0/0 cgrp=apps/bg\_non\_interactive handle=1443076000

   | state=S schedstat=(1941480839 10140523154 4229 ) utm=119 stm=75 core=0

   at java.lang.Object.wait(NativeMethod)

   - waiting on<0x427a8618> (a java.lang.VMThread) held by tid=33 (AsyncTask #1)

   at java.lang.Thread.parkFor(Thread.java:1212)

   at sun.misc.Unsafe.park(Unsafe.java:325)

   at java.util.concurrent.locks.LockSupport.park(LockSupport.java:157)

   at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2017)

   at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:410)

   at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1035)

   at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1097)

   at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:587)

   at java.lang.Thread.run(Thread.java:848)

##### Suspend 状态，通常表明是抓取backtrace时，当时还正在执行java代码，被强制suspend的情况

"FileObserver" prio=5tid=23 SUSPENDED

   | group="main"sCount=1 dsCount=0 obj=0x41fd1dc8 self=0x551abda0

   | sysTid=710 nice=0 sched=0/0cgrp=apps handle=1427817920

   | state=S schedstat=(130152222 399783851 383 ) utm=9 stm=4 core=0

   #00  pc 000329f8 /system/lib/libc.so (\_\_futex\_syscall3+8)

   #01  pc 000108cc /system/lib/libc.so (\_\_pthread\_cond\_timedwait\_relative+48)

   #02  pc 0001092c /system/lib/libc.so (\_\_pthread\_cond\_timedwait+64)

   #03  pc 00055a93 /system/lib/libdvm.so

   #04  pc 0005614d  /system/lib/libdvm.so(dvmChangeStatus(Thread\*， ThreadStatus)+34)

   #05  pc 0004ae7f /system/lib/libdvm.so

   #06  pc 0004e353 /system/lib/libdvm.so

   #07  pc 000518d5 /system/lib/libandroid\_runtime.so

   #08  pc 0008af9f /system/lib/libandroid\_runtime.so

   #09  pc 0001eacc /system/lib/libdvm.so (dvmPlatformInvoke+112)

   #10  pc 0004fed9 /system/lib/libdvm.so (dvmCallJNIMethod(unsigned int const\*， JValue\*， Method const\*， Thread\*)+484)

   #11  pc 00027ea8 /system/lib/libdvm.so

   #12  pc 0002f4b0 /system/lib/libdvm.so (dvmMterpStd(Thread\*)+76)

   #13  pc 0002c994 /system/lib/libdvm.so (dvmInterpret(Thread\*， Methodconst\*， JValue\*)+188)

   #14  pc 000632a5 /system/lib/libdvm.so (dvmCallMethodV(Thread\*， Methodconst\*， Object\*， bool， JValue\*， std::\_\_va\_list)+340)

   #15  pc 000632c9 /system/lib/libdvm.so (dvmCallMethod(Thread\*， Methodconst\*， Object\*， JValue\*， ...)+20)

   #16  pc 00057961 /system/lib/libdvm.so

   #17  pc 0000dd40 /system/lib/libc.so (\_\_thread\_entry+72)

   at android.os.FileObserver$ObserverThread.observe(NativeMethod)

   at android.os.FileObserver$ObserverThread.run(FileObserver.java:88)

### Native Backtrace

#### **Native Backtrace 抓取方式**

##### 添加代码直接抓取

Google 默认提供了CallStack API，请参考

system/core/include/libutils/CallStack.h

system/core/libutils/CallStack.cpp

可快速打印单个线程的backtrace.

##### 自动抓取

Natice层的进程发生异常后一般都在/data/tombstones目录下生成文件（墓碑文件），该文件描述了该进程当前的Backtrace，不过是链接地址，需要进行解析。

##### 2.1.3利用debuggerd抓取

MTK 已经制作了一个利用debuggerd抓取Native backtrace的tool RTT(Runtime Trace)，对应的执行命令是:

rtt builttimestamp (Apr 18 2014 15:36:21)

USAGE : rtt[-h] -f function -p pid [-t tid]

  -ffuncion : current support functions:

                bt  (Backtrace function)

  -ppid     : pid to trace

  -ttid     : tid to trace

  -nname    : process name to trace

 -h         : help menu

#### **解析Native Backtrace**

你可以使用GDB，或者addr2line等 tool 来解析抓回的Native Backtrace，从而知道当时正在执行的native代码，

arm-linux-androideabi-addr2line-f -C -e symbols address

公司将addr2line封装后实现了一套解析NativeBacktrace(show Stack)。参考下文

### Kernel Backtrace

#### **Kernel Backtrace 抓取方式**

##### 运行时抓取

Sadf

\* AEE/RTT 工具

\* ProcSystem

  Cat proc/pid/task/tid/stack

\*Sysrq-trigger

 Adb shell cat proc/kmsg > kmsg.txt

 adbshell "echo 8 > proc/sys/kernel/printk“ //修改printk loglevel

 adbshell "echo t > /proc/sysrq-trigger“ //打印所有的backtrace

 adbshell "echo w > /proc/sysrq-trigger“//打印'-D' status'D'的 process

\* KDB

 Longpress volume UP and DOWN more then 10s

 btp            <pid>

 Displaystack for process <pid>

 bta            [DRSTCZEUIMA]

 Displaystack all processes

 btc

 Backtracecurrent process on each cpu

 btt            <vaddr>

 Backtraceprocess given its struct task add

##### 添加代码直接抓取

Sg

\* #include<linux/sched.h>

 当前thread:  dump\_stack();

 其他thread:  show\_stack(task， NULL);

## **Process/Thread 状态**

大风格

 "R(running)"，  /\*  0 \*/

 "S(sleeping)"，  /\*  1 \*/

 "D(disk sleep)"， /\*   2\*/

 "T(stopped)"，  /\*  4 \*/

 "t(tracing stop)"， /\*   8\*/

 "Z(zombie)"，  /\* 16 \*/

 "X(dead)"，  /\* 32 \*/

 "x(dead)"，  /\* 64 \*/

 "K(wakekill)"，  /\* 128 \*/

 "W(waking)"，  /\* 256 \*/

通常一般的Process处于的状态都是S(sleeping)，而如果一旦发现处于如D (disksleep)， T (stopped)， Z (zombie)等就要认真审查。

## 系统运行环境

客观的反应系统的执行环境，通常包括如CPU利用率，Memory 使用情况， Storage 剩余情况等。这些资料也非常重要，比如可以快速的知道，当时是否有Process在疯狂的执行，当时是不是处于严重的low memory情况， Storage是否有耗尽的情况发生等。

## 程序执行环境

客观的反应当时某个程序(Kernel也可以看成一个程序)的执行现场，此类资讯通常包括如process的coredump， java heap prof， kernel的memory dump 等。完整的执行环境，我们可以快速的知道当时具体的变量的值，寄存器值等，可以精细的分析问题。

## 其他的一些资讯

这些资讯相对来说，比较零散了，如通常的LOG，一些debug命令的结果数据等。

# 几种典型的异常情况

# 死机运行环境分析

## 系统运行环境

客观的反应系统的执行环境，通常包括如CPU利用率，Memory 使用情况， Storage 剩余情况等。这些资料也非常重要，比如可以快速的知道，当时是否有Process在疯狂的执行，当时是不是处于严重的low memory情况， Storage是否有耗尽的情况发生等

## CPU Usage

追查CPU利用率可大体的知道，当时机器是否有Process在疯狂的运行，当时系统运行是否繁忙。通常死机分析，只需要抓取基本的使用情况即可。下面说一下一般的抓取方式

### top

top 可以简单的查询Cpu的基本使用情况

Usage: top[ -m max\_procs ] [ -n iterations ] [ -d delay ] [ -s sort\_column ] [ -t ] [ -h]

   -m num  Maximum number of processes to display.

   -n num  Updates to show before exiting.

   -d num  Seconds to wait between updates.

   -s col  Column to sort by (cpu，vss，rss，thr).

   -t      Show threads instead of processes.

   -h      Display this help screen.

#### 正常情况

Ghbsfg

User 2%， System 12%， IOW 0%， IRQ 0%

User 14 + Nice 0 + Sys 67 + Idle 471 + IOW0 + IRQ 0 + SIRQ 2 = 554

  PID   TID PR CPU%S     VSS     RSS PCYUID     Thread          Proc

 2423  2423  1  8% R   2316K   1128K    root    top             top

  270   270 0   1% S   2160K   924K     root    aee\_resmon      /system/bin/aee\_resmon

  159   159 0   0% D     0K      0K    root     bat\_thread\_kthr

   3     3  0   0%S      0K     0K     root     ksoftirqd/0

   57    57 0   0% D     0K      0K    root     hps\_main

User 1%， System 7%， IOW 0%， IRQ 0%

User 10 + Nice 0 + Sys 41 + Idle 494 + IOW0 + IRQ 0 + SIRQ 0 = 545

  PID   TID PR CPU%S     VSS     RSS PCYUID     Thread          Proc

 2423  2423  1  8% R   2324K   1152K    root    top             top

   57    57 0   0% D     0K      0K    root     hps\_main

  242   419 0   0% S   8600K   4540K    shell    mobile\_log\_d    /system/bin/mobile\_log\_d

  982   991 0   0% S   4364K   1156K    media\_rw sdcard         /system/bin/sdcard

  272   272 0   0% S  30680K   9048K    root    em\_svr          /system/bin/em\_svr

  从上面可以看出，系统基本运行正常，没有很吃CPU的进程。

#### 异常情况

Dfhg

User 59%， System 4%， IOW 2%， IRQ 0%

User 1428 + Nice 0 + Sys 110 + Idle 811 +IOW 67 + IRQ 0 + SIRQ 1 = 2417

  PID   TID PR CPU%S     VSS     RSS PCYUID     Thread          Proc

16132 32195  3  14% R997100K  53492K  bg u0\_a60  Thread-1401     com.android.mms

16132 32190  1  14% R997100K  53492K  bg u0\_a60  Thread-1400     com.android.mms

16132 32188  2  14% R997100K  53492K  bg u0\_a60  Thread-1399     com.android.mms

16132 32187  0  14% R997100K  53492K  bg u0\_a60  Thread-1398     com.android.mms

18793 18793  4   1%R   2068K   1020K    shell   top             top

User 67%， System 3%， IOW 7%， IRQ 0%

User 1391 + Nice 0 + Sys 75 + Idle 435 +IOW 146 + IRQ 0 + SIRQ 1 = 2048

  PID   TID PR CPU%S     VSS     RSS PCYUID     Thread          Proc

16132 32195  3  16% R997100K  53492K  bg u0\_a60  Thread-1401     com.android.mms

16132 32188  2  16% R997100K  53492K  bg u0\_a60  Thread-1399     com.android.mms

16132 32190  0  16% R997100K  53492K  bg u0\_a60  Thread-1400     com.android.mms

16132 32187  1  16% R997100K  53492K  bg u0\_a60  Thread-1398     com.android.mms

18793 18793  4   2%R   2196K   1284K    shell   top             top

可以明显的看到，贵司的mms的4个thread都有进入了deadloop，分别占用了一个cpu core. 同时可以快速抓取他们的java trace， 进一步可以看到当时MMS的四个backtrace，以便快速分析。

"Thread-1401" prio=5 tid=32SUSPENDED JIT

  | group="main" sCount=1dsCount=0 obj=0x4264f860 self=0x7b183558

  | sysTid=32195 nice=0 sched=0/0cgrp=apps/bg\_non\_interactive handle=2078705952

  | state=S schedstat=( 3284456714198104216273858 383002 ) utm=324720 stm=3725 core=5

  at com.yulong.android.mms.c.f.d(MmsChatDataServer.java:~1095)

  at com.yulong.android.mms.ui.MmsChatActivity$37.run(MmsChatActivity.java:7582)

  at java.lang.Thread.run(Thread.java:841)

"Thread-1400" prio=5 tid=31SUSPENDED JIT

  | group="main" sCount=1dsCount=0 obj=0x41f5d8f0 self=0x7be2a8e8

  | sysTid=32190 nice=0 sched=0/0cgrp=apps/bg\_non\_interactive handle=2078029504

  | state=S schedstat=( 3284905134412105526230562 382946 ) utm=324805 stm=3685 core=5

  at com.yulong.android.mms.ui.MmsChatActivity$37.run(MmsChatActivity.java:~7586)

  at java.lang.Thread.run(Thread.java:841)

"Thread-1399" prio=5 tid=30SUSPENDED JIT

  | group="main" sCount=1dsCount=0 obj=0x42564d28 self=0x7b0e6838

  | sysTid=32188 nice=0 sched=0/0cgrp=apps/bg\_non\_interactive handle=2077662640

  | state=S schedstat=( 3288042313685103203810616 375959 ) utm=325143 stm=3661 core=7

  at com.yulong.android.mms.ui.MmsChatActivity$37.run(MmsChatActivity.java:~7586)

  at java.lang.Thread.run(Thread.java:841)

"Thread-1398" prio=5 tid=29SUSPENDED

  | group="main" sCount=1dsCount=0 obj=0x4248e5a8 self=0x7be0d128

  | sysTid=32187 nice=0 sched=0/0cgrp=apps/bg\_non\_interactive handle=2079251904

  | state=S schedstat=( 3287248372432105116936413 379634 ) utm=325055 stm=3669 core=6

  at com.yulong.android.mms.ui.MmsChatActivity$37.run(MmsChatActivity.java:~7586)

  at java.lang.Thread.run(Thread.java:841)

当时处于suspend，即意味着当时这四个thread正在执行java code，而抓取backtrace时强制将thread suspend。看起来客户改动所致，并且客户有Proguard，麻烦客户自己review代码。

### systrace

ftrace 可以纪录CPU最为详细的执行情况，即linux scheduler的执行情况。通常默认只开启 sched\_switch。

如何抓取ftrace可以查询相关的FAQ。

### Kernelcore status

有的时候我们需要追查一下，当时Kernel的基本调度情况，以及接收中断的情况，以判断当前CPU执行的基本情况是否异常。比如有时候如果某个中断上来太过频繁，就容易导致系统运行缓慢，甚至死机。

\* CPU Sched status

     adb shell cat proc/sched\_debug

     Use sysrq-trigger

\* CPU interrupts

     adb shell cat proc/interrupts

     adb shell cat proc/irq/irq\_id/spurious

## Memory Usage

Gh

MemoryUsage，我们通常会审查，系统当时memory是否足够，是否处于low memory状态，是否可能出现因无法申请到memory而卡死的情况。

常见的一些基本信息如下:

\* meminfo: basic memory status

adb shell cat proc/meminfo

adb shell cat proc/pid/maps

adb shell cat proc/pid/smaps

\* procrank info: all process memory status

adb shell procrank

adb shell procmem pid

adb shell dumpsys meminfo pid

\* zoneinfo:

adb shell cat proc/zoneinfo

\* buddyinfo:

adb shell cat /proc/buddyinfo

## Storage Usage

Sdfg查看Storage的情况，通常主要是查询data分区是否已经刷满， sdcard是否已经刷满，剩余的空间是否足够。以及是否有产生超大文件等。

通常使用的命令如 df

df

Filesystem              Size     Used     Free  Blksize

/dev                  446.0M   128.0K   445.8M   4096

/sys/fs/cgroup        446.0M    12.0K   445.9M   4096

/mnt/secure           446.0M     0.0K   446.0M   4096

/mnt/asec             446.0M     0.0K   446.0M   4096

/mnt/obb              446.0M     0.0K   446.0M   4096

/system                 1.2G   915.3M   355.5M   4096

/data                   1.1G   136.7M  1010.1M   4096

/cache                106.2M    48.0K   106.2M   4096

/protect\_f              4.8M    52.0K     4.8M   4096

/protect\_s              4.8M    48.0K     4.8M   4096

/mnt/cd-rom             1.2M     1.2M     0.0K   2048

/mnt/media\_rw/sdcard0    4.6G     1.1G     3.4G  32768

/mnt/secure/asec        4.6G     1.1G     3.4G  32768

/storage/sdcard0        4.6G     1.1G     3.4G  32768

以及ls， du 等命令，如du

du -help

usage: du [-H | -L | -P] [-a | -d depth |-s] [-cgkmrx] [file …]

du -LP -d 1

8       ./lost+found

88      ./local

384     ./misc

48      ./nativebenchmark

912     ./nativetest

8       ./dontpanic

13376  ./data

8       ./app-private

8       ./app-asec

129424  ./app-lib

8       ./app

136     ./property

16      ./ssh

116312  ./dalvik-cache

8       ./resource-cache

48      ./drm

8       ./mediadrm

8       ./security

3888    ./nvram

8       ./amit

8       ./acdapi

88      [./@btmtk](mailto:./@btmtk)

32      ./sec

8       ./user

16      ./media

16      ./agps\_supl

8       ./anr

8       ./gps\_mnl

8       ./nfc\_socket

16      ./ccci\_cfg

32      ./mdlog

1312    ./system

176     ./recovery

32      ./backup

274688  .

# 进程运行环境分析

## 系统运行环境

当我们怀疑死机问题可能是某个进程出现问题而引发时，通常我们需要对这个进程进行深入的分析，即进程运行环境分析。通常包括分析如，线程状态，各种变量值，寄存器状态等。在Android系统中，我们将其划分成三个层次。

即 Java运行环境分析， Native运行环境分析， Kernel运行环境分析。下面分别说明。

## Java 运行环境分析

Fg

我们对于Zygote fork出来的process，如APP以及system\_server，都会进行Java运行环境分析。其关键是分析Java Heap，以便快速知道某个Java变量的值，以及Java对象的分布和引用情况。

通常Java Heap的分析方式则是抓取Java Hprof，然后使用MAT等工具进行分析。

\* 抓取Hprof的手法，如:

### 第一种方式:使用am 命令

   adb shell am dumpheap {Process} file

   如：

 adbshell chmod 777 /data/anr

 adbshell am dumpheap com.android.phone /data/anr/phone.hprof

 adbpull /data/anr/phone.hprof

### 第二种方式:使用DDMS 命令

 在DDMS中选择对应的process，然后在Devices按钮栏中选择Dump Hprof file，保存即可

### 第三种方式:通过代码的方式

   在android.os.Debug这个class 中有定义相关的抓取hprof 的method。

 如: public static void dumpHprofData(String fileName) throwsIOException;

 这样即可在代码中直接将这个process的hprof 保存到相对应的文件中，注意这个只能抓取当时的process。

 如果想抓其他的process的hprof，那么就必须通过AMS帮忙了。

 可以先获取IActivityManager接口，然后调用它的dumpheap方法。具体的代码，大家可以参考

 frameworks/base/cmds/am/src/com/android/commands/am/am.java中的调用代码

### 第四种方式:发送SIGUSER1

   在部分机器中，如果具有root权限，可以直接发送SIG 10来抓取，此时对应的Hprof保存在/data/misc下面，文件名如：

 heap-dump-tm1357153307-pid1882.hprof

\* 快速分析

首先， DVM的Hprof 和标准的Java Hprof 有一些差别，需要使用hprof-conv进行一次转换，将DVM格式的hprof 转换成标准的java 命令的hprof

   hprof-conv in.hprof out.hprof

其次，使用如MAT Tool，打开转换后的hprof文件，通常我们会

 analysis java thread information

 analysis java var value

 analysis Object reference

 analysis GC path

具体如何使用MAT分析可以参考MAT 的官方网站

## Native 运行环境分析

Sd

Native 运行环境分析，我们通常会采用Core dump分析手法。 Core dump纪录了当时进程的各类关键资讯，比如变量参数，线程stack，heap， register等。通常可以认为是这个Process当时最为完整的资讯了。但Core dump往往比较大，有时甚至会超过1G，属于比较重量型的分析手法了。

\* 如何抓取Core Dump。

  目前MTK的机器会将Core Dump转换成AEE DB。否则对应的Core dump文件即存放在/data/core目录下

  手工抓取时，可以：

  adb shell aee -d coreon

  adb shell kill -31 PID

  此时core dump就可能存放在两个目录下:/data/core，以及/sdcard/mtklog/aee\_exp下面新的DB 文件。

\* 如何分析Core Dump。

  因为通常已经将Core Dump转换成了AEE DB。所以首先将AEE DB解开，即可以看到PROCESS\_COREDUMP的文件，有的时候此文件很大，比如超过1G。

  而分析Core Dump的Tools 很多，比如traces32， GDB等，这里就不详加说明，可以参考网络上的相关文档。

## Kernel 运行环境分析

Fgh

从82平台上多了ramdump功能，可以发生KE后将82/92的物理内存压缩保存到EMMC的内置卡(默认保存到EMMC内置卡上)(92可以选择外置t卡)上，拿到该文件后就可以转换为kernel space，查看kernel各种变量，比查看kernel log更加方便快捷。

只有在eng版本下支持该功能，并且是EMMC的，存在内置T卡才行，

在projectConfig.mk里的MTK\_SHARED\_SDCARD必须为no即MTK\_SHARED\_SDCARD=no

连上adb后：

adb shell

#echo Y> /sys/module/mrdump/parameters/enable

#echo emmc> /sys/module/mrdump/parameters/device  (注意82只能在EMMC内置t卡上，不能下这条命令，92可以下这条命令修改到sdcard：#echo sdcard > /sys/module/mrdump/parameters/device)

这样就开启了ramdump功能，注意重启后无效，必须重新设置才行

之后重新开机，此时会在

内置T卡：/storage/sdcard0/

或外置T卡：/storage/sdcard1/

看到CEDump.kdmp文件，结合kernel/out/vmlinux或out/target/product/$proj/obj/KERNEL\_OBJ/vmlinux一起提供给Mediatek即可做进一步kernel异常重启的分析。

# 死机问题场景

Fgh

当你遇到死机问题时，你可能面临的是下面三种情况。

\* 有死机现场

  资讯最为充足，你可以快速的利用死机现场来分析。

\* 已经重启过的手机

  死机现场已经不复存在，但手机还没有刷机，可以从手机中抓取已经存在的资讯来分析。

\* 仅仅一些LOG或者其他的资讯

  需要从这些LOG或者资讯中猜测当时手机的状态，以及可能死机的原因。

 从这三种情况，不难得知，有死机现场的情况下是最容易分析的。而如果仅仅只有一些LOG的话，就需要工程师具有非常丰富的经验，从仅有的LOG中，提取有价值的资讯，来猜测出当时死机的原因。

# 死机问题现场分析

## 死机现场分析手法

Sdf

死机分析如同医生给病人诊断病情，所有的手法不在乎分为两种。

\* 高科技的诊断设备，分析病人的病情。死机分析就依靠各种技术方法去诊断当时手机的运行的真实状态。

\* 通过病人的各种活动，分析病人的潜在病因。就依靠各种对手机的操作，以及命令，让手机跑相应的流程，从而进一步分析。

## Test Phone Usage

Gf  
\* 通过做一些基本的测试，可大体上确认可能引发死机的模块，为后续Debug确定好方向。

\* 通常只能正向推理，可以做什么 =》推断什么模块正常；逆向推理可能因为各种原因而失常。

### Touch Panel

  - 屏幕是否有响应(一般情况下没响应)。

  - 如果有响应，可能机器已经活过来了，或者当时把ANR认为了hang 机；需要进一步确认情况。

  - 确认按键的情况，通常都设定振动反馈，如果有，那么就认为当时按键事件可以传递到SystemServer，此时可能System Server 逻辑异常。

### Power Key/ Volumn Up/Down Key

H

- 是否可以亮、灭屏

  - 可点亮关闭屏幕，说明KPD ->input manager->PowerManagerService->Power->LCDdriver正常；通常可以怀疑TPD，以及

SurfaceFlinger。

### SurfaceFlinger。

\* 是否可以显示音量调整情况

- 可显示音量调整情况，进一步说明SurfaceFlinger也正常，进一步怀疑TPD，或者直接的APP无响应的情况。

### SystemUI & Status Bar

- Status Bar 是否可以拉下，以便防止只是活动区卡住的情况，可下拉，说明只是APP卡住，或者lockscreen无法解锁的情况

### LockScreen

- 测试LockScreen是否可以解锁，

### Home Key/ Back Key/Menu Key

- 确认当时是否只是APP hang住的情况，避免将ANR误判为死机

### 插入USB观察充电情况

- 可确认Surfaceflinger， System Server的运行情况

## USB/ADB Debug Hang

### USB/ADB Debug 准备

L

\* 插入USB，确认ADB是否可以使用

 - 首先查看windows的设备管理器里面是否出现对应的设备

 - 在命令行中输入adb devices，看是否可以打印设备信息，在输入之前您最好先输入adb kill-server 保证pc上的adb client没有卡住

 - 请确保您使用的PC上已经安装ADB，USB端口本身正常

### USB/ADB 连接分析

Hj

 \*Case1: adb 能正常连接，adb shell #

- adb 目前可以正常接入，可以进行USB/ADB Debug

 \*Case2: windows 设备管理器有反应，adb shell提示device not found。

- 确认adb驱动是否安装好，USB的uid 和 vid 是否和driver匹配。

- 可能机器的adbd已经无法对接，尝试adbkill-server来确认

 \*Case3:  windows 设备管理器有反应，adb shell提示offline。

- 机器的adbd拒绝连接，或者adbd无法拉起。

- PC adb 版本过低

- 很可能adbd已经卡住， adbdprocess status为 ‘D’

### 机器状态基本检测

#### ps or ps-t or ps -t -p查看进程基本状态

gh

-t  以线程为单位打印

 -p  附加打印线程优先级信息

 \*\* 追查如system server， surfaceflinger， service manager， media server(ss， sf， sm、ms)，zygote，等关键进程的状态，进行初步确认。正常情况下，都应处于’S’，异常情况有如’D’， ‘T’， ‘Z’ ， ‘R’等

\*\* 大体追查ss， sf， sm， ms， zygote等的memory情况，是否有明显的溢出情况

\*\* 大体查看当时的ss pid， sf pid，如果机器上层没有重启过，通常sf pid < 200， ss pid < 600，如果pid比较大就说明上层重启过。

\*\* 是否还存在特别进程，如ipod，表示在ipo关机中，如aee\_core\_forworder，表示在抓core dump，aee\_dumpstate表示aee 在抓取db 信息等。

\*\* 追查debuggerd的process 数量，通常正常时，只有一个debuggerd process，pid < 200，如果有达到4个debuggerd，这个时候将无法进行rtt之类的操作

#### df

- 审查storage的使用情况，查看SD 卡和 data 分区使用情况，特别是如SD卡已满，或者data 分区写满等

#### catproc/meminfo， procrank

- 审查当前的memory使用情况，追查各个进程的memory情况

#### getprop

- 审查当前system properties情况

#### top-t -m 5 -n 2

dsf

top -t -m 5 -n 2

- 摸一摸手机，感觉有点热，或者发烫的话，说明通常是CPU利用率比较高

- 大体上审查当前CPU的利用情况，不求精湛

总之，机器状态基本检测，目标就是通过简单几个命令直接侦测当前手机最为可能的异常情况。包括关键进程基本状态， CPU利用率，memory 状况，storage 状况等。做出基本的预先分析，从而为下一步的debug打好基础。

## UART Debug

当usb/adb无法确认问题时或者已经无法连接时，我们需要利用UART/KDB来进一步厘清问题。

确认UART以及 UART Console已经开启。

ENG 版本默认开启，User版本默认关闭。通常在User版本上如果需要抓取UART日志的话需要用散件工具单烧ENG版本的LK分区重启就可以抓取UART日志了。

目前我司一般的UART端口都不支持输入，所以只能通过输出的UART输出的log来分析问题。

## 确认死机的模块

我们又回到了最开始的可能会导致死机的模块，下面我们将针对每一个模块做详细的说明。

### Input Driver-Input System

F

\* 确认Input Driver-Input System通路是否正常，即inputdriver是否可以传上正常的输入

\* 最常见的检测方式是 adb shell getevent

如直接输入adb shell getevent可以看到:

D:\log>adbshell getevent

add device 1: /dev/input/event1

name: "fts\_ts"

add device 2: /dev/input/event6

name: "temperature"

add device 3: /dev/input/event5

name: "gyro"

add device 4: /dev/input/event4

name: "gsensor"

add device 5: /dev/input/event2

name: "compass"

add device 6: /dev/input/event3

name: "lightsensor-level"

add device 7: /dev/input/event0

name: "rk29-keypad"

/dev/input/event3: 0003 0028 00000106

对应的操作命令是:

Usage:getevent [-t] [-n] [-s switchmask] [-S] [-v [mask]] [-d] [-p] [-i] [-l] [-q][-c count] [-r] [device]

-t: show time stamps

-n: don't print newlines

-s: print switch states for given bits

-S: print all switch states

-v: verbosity mask (errs=1， dev=2， name=4， info=8， vers=16， pos. events=32， props=64)

-d: show HID descriptor， if available

-p: show possible events (errs， dev， name， pos. events)

-i: show all device info and possible events

-l: label event types and names in plain text

-q: quiet (clear verbosity mask)

-c: print given number of events then exit

-r: print rate events are received

通常我们会一项一项的确认native层是否可以收到Kernel传递来的input event事件。

比如我们追查KPD是否有效果，按power key可以看到相关的行为。

#### /dev/input/event0

D:\log >adbshell getevent -t -l /dev/input/event0

130|shell@gl300e:/ $ getevent -t -l /dev/input/event0

[ 1639.330086] EV\_KEY KEY\_POWER DOWN

[ 1639.330086] EV\_SYN SYN\_REPORT 00000000

[ 1639.510170] EV\_KEY KEY\_POWER UP

[ 1639.510170] EV\_SYN SYN\_REPORT 00000000

当然如果屏幕是点亮时，肯定就要快速追查tpd以及模拟按键是否正常

#### getevent -t -l /dev/input/event1

shell@gl300e:/ $ getevent -t -l /dev/input/event1

[ 1933.322190] EV\_ABS ABS\_MT\_TRACKING\_ID 000000a7

[ 1933.322190] EV\_ABS ABS\_MT\_POSITION\_X 00000315

[ 1933.322190] EV\_ABS ABS\_MT\_POSITION\_Y 000003c1

[ 1933.322190] EV\_KEY BTN\_TOUCH DOWN

[ 1933.322190] EV\_SYN SYN\_REPORT 00000000

[ 1933.421282] EV\_ABS ABS\_MT\_TRACKING\_ID ffffffff

[ 1933.421282] EV\_KEY BTN\_TOUCH UP

[ 1933.421282] EV\_SYN SYN\_REPORT 00000000

如果发现无法收到此类资讯，那么就可以确认对应的devices可能有故障，需要请对应的driver& vendor工程师来分析。

### system-server logic

g

此分析的关键在于分析 system-server是否还在正常的运转，这是非常重要的，据统计，40%的usb可以debug 的hang 机问题，都可以通过对system-server逻辑的审查找出原因。

Systemserver 是整个android上层的中枢，容纳了最为重要的service。

对System server的分析主要是通过java native的backtrace来追查Systemserver的关键thread 有没有被lock/dead lock，有没有进入dead loop，状态是否正常。

从死机机的角度来看system server关键的thread 如:

#### 关键的thread

Serverthread: System-server的main looper建立在serverthread上，所有service如果不单独创立thread-looper那么都将运行在这个Server thread上，其关键性不言而喻，用watchdog来监测该thread。

ActivityManager:

处理所有的Activity状态切换，broadcast，以及ANR 监测等等，非常重要。

WindowManager:

处理各种window叠加与切换，Input相关处理。

WindowManagerPolicy/UI:

主要处理LockScreen相关流程，它卡住，lockscreen无法解锁

PowerManagerService:

处理Power相关事宜，它卡住，屏幕无法熄灭，点亮

InputDispatcher/InputReader

处理Input System相关事宜，一旦他们卡住，那么所有的Input Event都无法反馈到其他的module，机器死机就成必然。

在KK以及KK 以后的版本中Google 取消了serverthread这样的main looper thread，直接使用system server 的第一个thread作为main looper。 Google对system server的threads 做了进一步的整合，以减少繁杂的线程数，新增了如android.io，android.ui，android.fg，android.bg等threads。将大量的普通型的Handler按时效分类分散到上面的4个threads中。

所以KK及更高的版本还需要分析ndroid.io， android.ui， android.fg， android.bg等thread。

这些thread都通过执行MQ-Looper-Handler的模式运行，所以正常的时候的java/native backtrace都是：

Java:

at android.os.MessageQueue.nativePollOnce(NativeMethod)

at android.os.MessageQueue.next(MessageQueue.java:138)

at android.os.Looper.loop(Looper.java:150)

Native:

#00 pc 0002599c /system/lib/libc.so (epoll\_wait+12)

#01 pc 000105e3 /system/lib/libutils.so (android::Looper::pollInner(int)+94)

#02 pc 00010811 /system/lib/libutils.so (android::Looper::pollOnce(int， int\*， int\*， void\*\*)+92)

#03 pc 0006ca5d /system/lib/libandroid\_runtime.so(android::NativeMessageQueue::pollOnce(\_JNIEnv\*， int)+22)

一旦不是就可能已经出现异常了。

BinderThread

Systemserver 为外界提供服务，绝大多数都通过binder与其他process 对接。BinderThread 的数量动态调整，默认不会超过16个，如果数量达到16个，即说明当前system server可能非常繁忙，有process非常频繁的和Process通信。

在JB以及以前版本，System-server的第一个thread，在调起serverthread后，自己加到IPCthread pool中，成为Binder Thread的一员。

对于Binder Thread，正常时对应的Backtrace:

#00 pc 000247e8 /system/lib/libc.so(\_\_ioctl+8)

#01 pc 00038248 /system/lib/libc.so (ioctl+28)

#02 pc 0001d3a9 /system/lib/libbinder.so(android::IPCThreadState::talkWithDriver(bool)+140)

#03 pc 0001dae3 /system/lib/libbinder.so(android::IPCThreadState::getAndExecuteCommand()+6)

#04 pc 0001db79 /system/lib/libbinder.so(android::IPCThreadState::joinThreadPool(bool)+48)

#05 pc 00021a79 /system/lib/libbinder.so

#06 pc 0000ea01 /system/lib/libutils.so(android::Thread::\_threadLoop(void\*)+216)

#07 pc 0004f1c9 /system/lib/libandroid\_runtime.so (android::AndroidRuntime::javaThreadShell(void\*)+68)

#08 pc 0000e533 /system/lib/libutils.so

#09 pc 0000d600 /system/lib/libc.so(\_\_thread\_entry+72)

at dalvik.system.NativeStart.run(Native Method)

抓取system server的java backtrace，依次check serverthread(JB)，ActivityManager， WindowManager， WindowManagerPolicy，PowerManagerService以及android.io， android.bg， android.fg， android.ui的状态，如状态异常，则依次推导。

当发现java backtrace最后调用到异常nativemethod时，抓取其native backtace，通过native backtrace 进一步追查.

如果在native backtrace中，发现已经调入binderdriver，那就是通过binder进行IPC call，这个时候就要知道binder的对端process，然后查阅它的binder thread进程进一步分析问题。

Case1: Abnormal Java Backtrace “Deadlock”

图片deadlock-java-backtrace.png

Case2: Abnormal Java Backtrace “Lock in native thread"

图片lock-in-native-thread.png

Case3: Abnormal native Backtrace “Lock in Binder”

图片lock-in-binder.png

Case4: Abnormal native Backtrace "Lock in Binder thread"

图片lock-in-binder-thread.png

当确认前面的key thread都没用问题，而通过getevent确认event 已经传递到system server。问题可能出在inputsystem中。

WindowManagerService通过InputManager提供的接口开启一个线程驱动InputReader不断地从/dev/input /目录下面的设备文件读取事件，然后通过InputDispatcher分发给连接到WindowManagerService服务的客户端。

Input Reader 正常的backtrace:

"InputReader" sysTid=611

#00 pc 00027754 /system/lib/libc.so(epoll\_wait+12)

#01 pc 0001f345 /system/lib/libinput.so(android::EventHub::getEvents(int， android::RawEvent\*， unsigned int)+1092)

#02 pc 0002aaf7 /system/lib/libinput.so(android::InputReader::loopOnce()+142)

#03 pc 00027613 /system/lib/libinput.so(android::InputReaderThread::threadLoop()+8)

#04 pc 00015318 /system/lib/libutils.so(android::Thread::\_threadLoop(void\*)+492)

#05 pc 0004d12b/system/lib/libandroid\_runtime.so(android::AndroidRuntime::javaThreadShell(void\*)+150)

#06 pc 00014948 /system/lib/libutils.so

#07 pc 0000f66c /system/lib/libc.so (\_\_thread\_entry+80)

Input Dispatcher 正常的backtrace:

"InputDispatcher" sysTid=610

#00 pc 00027754 /system/lib/libc.so(epoll\_wait+12)

#01 pc 0001c05c /system/lib/libutils.so(android::Looper::pollInner(int)+140)

#02 pc 0001c3d4 /system/lib/libutils.so (android::Looper::pollOnce(int， int\*， int\*， void\*\*)+76)

#03 pc 00026317 /system/lib/libinput.so(android::InputDispatcher::dispatchOnce()+94)

#04 pc 0001f5bd /system/lib/libinput.so(android::InputDispatcherThread::threadLoop()+8)

#05 pc 00015318 /system/lib/libutils.so(android::Thread::\_threadLoop(void\*)+492)

#06 pc 0004d12b/system/lib/libandroid\_runtime.so(android::AndroidRuntime::javaThreadShell(void\*)+150)

#07 pc 00014948 /system/lib/libutils.so

#08 pc 0000f66c /system/lib/libc.so(\_\_thread\_entry+80)

Backtrace通常都可以精确的定位问题点，比如卡在了哪一行。

那有没有可能您抓backtrace时，恰好运行到，造成乌龙的情况呢？这个通常需要具体情况具体分析

--- NativeBacktrace 处于R状态

--- JavaBacktrace 处于 Suspend状态， Running状态

---Backtrace 处于明确的非block状态

--- DoubleBacktrace 确认是否CPU有运转

随着android版本的推进，system-server越来越显得庞大，为此Google对system-server做了分拆动作.

After 4.0 SurfaceFlinger removed from system-server and created by init， single process SurfaceFlinger。

除SurfaceFlinger外，对system-server影响最大的是MediaServer。

注意对于surfaceflinger， mediaserver等非DVM Process，切忌不要对他们send signalSIGQUIT(3)，将导致对应的process直接退出

### Display logic

F

DisplayControl Flow

图片: display-control-flow.png

 \*Surface Manager(Surfaceflinger)

\*\*Surfaceflinger 是用户空间中Framework下libraries中负责显示相关的一个模块

\*\* 当系统同时执行多个应用程序时，Surfaceflinger负责管理显示，主要包括显示的叠加和显示的渲染

\*\* 其中显示的叠加，MTK架构按照一定的原则和算法，主要用到GPU和 OVL 模块的overlay 功能来做2D/3D 绘图的显示合成

 \* HWC

\*\* 基于DDP OVL硬件做HW Overlay功能

 \* GPU(Graphic Processing Unit )

\*\*aspecialized electronic circuit designed to accelerate the image processing， frame buffer rendering for output to a display。

 \*FrameBuffer

\*\* 基于Linux Frame buffer模型的MTKFramebuffer实作，主要来管理frame buffer。

 \*DDP(DisplayData Path)

\*\* MTKDisplay Hardware Controller，包括OVL， DMA， Color Processor，BLS，  DSI/DBI/DPIController

 DisplayData Flow

图片: display-data-flow.jpg

 \* Display大致的流程说明

\*\* APP(包括WMS)会把数据透过Surface送到SF(surfaceflinger)

\*\* SF 按照一定规则，通过GPU/HWC把多个Surface合成，并设定到DDP OVL 的对应layer

\*\*\* MTK OVL一般有4 layer(可能不同平台会有差异，目前基本都是4 layer)，每个layer都可以设定一个对应的图层，然后OVL把这写图层合成后送给DDP的下一级Module

\*\*\* 某些情况下(比如总的图层超过4个，OVL无法处理等各种case)，会使用GPU 去做2D/3D 的图层合成到frame buffer

\*\* SF/HWC 最后把GPU合成图层以及需要OVL合成的图层，设定到OVL的HW Register，并trigger HW Engine

\*\* MTK DDP 就会把合成的数据像管道一样流向LCM module，最后Panel显示相关的画面

  对Display的快速分析，我们主要是两个手段，第一个确认FrameBuffer数据是否和理想数据一致，第二个确认SurfaceFlinger状态是否正确。

\* 确认FrameBuffer数据和理想数据一致，那么说明android上层的处理行为都是正常的，而问题就很可能出在LCM的driver 或者 LCM 本身问题，此时需要联系LCM的driver 工程师以及 LCM vendor 来确认分析，这里不再详细说明。

 \* 确认SurfaceFlinger的状态，用于审查SurfaceFlinger的行为是否正常，对应的Thread是否能够正常工作，我司已经在SurfaceFlinger里面导入了Watchdog机制，审查SurfaceFlinger是否有卡住的情况，对应在main log里面会打印如:

 [SF-WD]detect SF maybe hang!!!

这样的LOG，并且会纪录卡顿的时机，如果持续卡顿，毫无疑问， SurfaceFlinger已经卡住，此时就要分析对应的backtrace以及 LOG。

\*SurfaceFlinger 的线程情况在不同的版本上变化比较大，最直接的方式是审查它binder thread情况，以及Event Loop的执行情况。 binderthread和 system server情况一致，不再详细说明。 Event Loop在JB 以及，JB 以及KK上都不相同，这里分别说明。

 在JB以前的版本，SurfaceFlinger的Event Loop使用的是普通的pthread控制，并且SurfaceFlinger的第一个Thread 在执行SurfaceFlinger的初始化后，转换为了binder thread。而Event Loop的thread 是新起的 "SurfaceFlinger" thread。对应的正常backtrace是:

图片: surfaceflinger-event-loop-beforce-jb.png

 在JB版本，SurfaceFlinger的Event Loop换成了Message Queue-Looper，对应的backtrace是:

图片: surfaceflinger-event-loop-jb.png

 在KK版本，SurfaceFlinger的第一个thread 直接作为了Event Loop thread，对应的backtrace是:

图片: surfaceflinger-event-loop-kk.png

 然后再根据SurfaceFlinger的代码逻辑，以及backtrace的lock 情况进行审查分析。

# 无现场猜测分析

D

前面已经提到，无场景分成两类

\* 已经重启过的手机

  死机现场已经不复存在，但手机还没有刷机，可以从手机中抓取已经存在的资讯来分析。

  \*仅仅一些LOG或者其他的资讯

  需要从这些LOG或者资讯中猜测，当时手机的状态，以及可能死机的原因。

  对于前者，如同法医解剖逝者遗体，分析可能的死亡原因。而对于后者，就相当于警察叔叔对一些曾年旧案的考证，挖掘与分析，难度可想而知。

## 死机资讯的完整纪录

D

为了能够更好更快的分析死机问题，通常我们都强烈建议保存好现场,如果一些特殊环境下无法保存现场，那么就要求测试工程师详细的纪录当时现场环境，包括但不限于下面的资讯:

\* 发现死机的时间

--- 如果是发现时，感觉机器早已经死机，也请说明

--- 如有截图，拍照，可以从图像上先获取

 \* 复现手法，操作的流程，当时环境

--- 强调在正常使用到死机过程中的操作。

--- 环境状态通常包括温度，湿度，网络信号状况。

--- 复现流程的视频

 \* 复现手机情况

--- 复现的软件版本:版本号？

--- 外部设备情况：有插入SD卡？耳机？SIM ？

--- 软件开启情况:  开启蓝牙？ WIFI？数据服务？GPS?

 \*复现的概率

--- 多少台手机做过测试，多少台手机可以复现。

--- 前后多少个版本可以复现，从哪个版本开始可以复现。

 其中死机的时间点至关重要，需要现场的工程师一定要比较精确的纪录。

## 死机Log 分析

g \* 确认hang机时间，和当时是否已经死机至关重要

\* 如果当时kernel已经异常，mobile log无法抓取log，那么log的价值大打折扣。

\* 反过来如果当时mobile log还在继续，说明至少kernel整体功能正常，先分析android上层的情况。

\* 从LOG中分析SystemServer， Surfaceflinger行为是否正常，是否有Lock住。

\* 查看System Server中关键Service 执行情况，如AMS， WMS， PowerManagerService， WindowManagerPolicy。

\* 观测AMS是比较快速的方法，因为AMS工作时，会使用到很多其他Service的Lock，比如broadcast， start activity， start service。通常一旦system server有卡住，则AMS 也会卡住，反过来如果AMS 没有卡住，那么通常就意味着systemserver没有直接卡住。

\* 查看Surfaceflinger，先查sf-wd，看看是否surfaceflinger是否已经卡住，然后再追查fps情况，HWComposer等情况。

\* 查看Power Key行为来确认上下通路是否正常，一般人看到死机，通常会去按一下Power Key来确认是否死机。

\* 查看Binder信息，看看Systemserver， Surfaceflinger等的IPC 情况。

\* 按Power Key 分析

因为死机，一般人都会习惯性的按Power key来查看是否可以恢复，而按Power Key的处理流程，涉及从Kernel => Input System => System Server => SurfaceFlinger等的整个流程，我们可以观察这个流程来查看hang机情况。

\* KPDreceives Interrupt and generate Power Key

图片: power-key-1。png

\* Systemserver receives Key and call set\_screen\_state

图片: power-key-2。png

\* PowerState Update:  request\_suspend\_state

图片: power-key-3.png

\* Setscreen on:  mt65xx\_leds\_set\_cust

图片: power-key-4.png

 通过审查每一个阶段流程，确认可能的hang机点。注意的是不同的版本可能有所不同，可以先用正常的机器复现一次后比对。

## 死机Log 分析， trace 辅助

A

\* 如果有提供/data/anr下的trace，或者相关的db文件。在确认死机的时间下，check trace对应的时间点，如果时间在死机或者hang机后，则是一份非常有价值的trace。通常查“-----”or “pid”

\* 借机审查system server， surfaceflinger的状况。

\* 如果/data/anr下有标准full\_traces\_xx即标柱这个backtrace是 SWT 产生的宝贵backtrace，里面一定有system server等的backtrace。

# 死机日志抓取

通常情况下，依靠Log可以找到对应的异常模块，但无法找出根本原因。需要依赖更多的信息来debug确认，所以针对这种情况，我们需要让测试人员抓出更多的信息来分析。

## 当机器死机时，建议使用下面脚本抓取资讯:

F

@echo "Pull important informationfrom hang phone， written by Yanghui Li"

adb devices

@echo "attach sdcard/mtklog"

adb pull /sdcard/mtklog mtklog/

 @echo “attach sdcard2/mtklog”

adb pull /sdcard2/mtklog mtklog/sdcard2

 @echo "attach trace"

adb pull /data/anr mtklog/anr

@echo "attach rtt dump forsurfaceflinger"

adb pull /data/rtt\_dump\* mtklog/sf\_dump

adb pull /data/anr/sf\_rtt mtklog/sf\_rtt\_1

@echo "attach data aee db"

adb pull /data/aee\_exp mtklog/data\_aee\_exp

@echo "attach data mobilelog"

adb pull /data/mobilelogmtklog/data\_mobilelog

 @echo "attach NE core"

adb pull /data/core mtklog/data\_core

 @echo "attach tombstones"

adb pull /data/tombstonesmtklog/tombstones

 @echo “attach phone state”

adb shell ps -t> mtklog/ps.txt

adb shell top -t -m 5 -n 3 > mtklog/top.txt

adb shell service list  > mtklog/serviceList.txt

adb shell cat /proc/meminfo >mtklog/meminfo

adb shell cat /proc/buddyinfo >mtklog/buddyinfo

adb shell procrank > mtklog/procrank.txt

adb shell cat proc/sched\_debug >mtklog/sched\_debug.txt

adb shell cat proc/interrupts >mtklog/interrupts.txt

adb shell dumpstate > mtklog/dumpstate.txt

@echo "finish."

pause

## 当机器重启后，可以使用下面脚本抓取：

H adb devices

@echo"抓出sdcard/mtklog"

adb pull/sdcard/mtklog mtklog/

 @echo"抓出trace"

adb pull/data/anr mtklog/anr

 @echo"抓出data aee db"

adb pull/data/aee\_exp mtklog/data\_aee\_exp

 @echo"抓出data mobilelog"

adb pull/data/mobilelog mtklog/data\_mobilelog

 @echo"抓出NE core"

adb pull/data/core mtklog/data\_core

 @echo"抓出tombstones"

adb pull/data/tombstones mtklog/tombstones

 @echo"抓sf rtt"

adb pull/data/rtt\_dump\* mtklog/sf\_rtt

adb pull/data/anr/sf\_rtt mtklog/sf\_rtt\_1

 @echo"完成"

pause

应当说针对不同的场景，死机和重启意义不一样。

对于普通用户来说，死机的危害性和重要性远大于重启。死机意味着机器在很长一段时间内机器都运行不正常，不能使用。危害性仅次于不能开机。重启很多时候用户都无法感知到，除非当时正在操作手机，并且通常很快就可以恢复。所以在USER版本中，我们要尽可能的把死机转换成重启。

对于开发者来说，重要的是解决问题，通常把死机转换成重启，为了规避一些未知因素，通常抓取的资讯比较少，难以直接分析问题。而如果有死机的现场，分析起来就相对容易得多。

所以针对ENG版本，我们建议不要把死机转换成重启。针对USER版本，尽可能的把死机转换成重启。

## 死机如何转换成重启

G

死机转换成重启的通用手段就是Watchdog机制，通过watchdog监测系统执行情况，一旦发生watchdog timeout，则做相应的异常处理。通常对于严重的情况，就是重启系统。

 在android中有两个最为重要的Watchdog机制，一个是System Server内部的Watchdog，用于监测System Server重要Thread 是否正常，关键Lock是否可以快速获取。一旦出现故障，则快速重启android上层。另外一个就是整个系统的Watchdog，这个依赖于不同平台厂商的实现，对于MTK平台来说，大体的做法是每一个cpu core对应一个Watchdog Thread [wdtk-X]，周期性(20s)的去写RGUregister，而RGUregister的timeout 时间是30s，即如果出现一个core的watchdog thread没有按时踢狗，那么就会触发一个timeout的FIQ，产生一个KE引发系统完整重启。

 MTK 在KK以后的平台，又做了hangmonitor机制， system server的watchdog thread周期性的踢kernel中的一个计时器，并设定下一次踢狗的时间，如果在设定的时间内没有踢狗，即意味着system server的watchdog 本身可能卡住，那么hang monitor机制就会自己触发一个HWT，引发系统完整重启。

当然并非所有的死机问题都能转换成重启。

## 死机VS重启

F

HWHang [maybe]

IRQ Disable & run longtime [After ICS Reboot， GB Hang]

SoftIRQ run longtime [Reboot]

Preempt Disable longtime [Reboot]

RT Thread busy[RT <99] [Hang or Hanglong time then reboot]

Kernel ModuleHang [maybe]

---- Not block watchdog thread reboot.

Native ModuleHang [maybe]

---- Not block watchdog thread reboot.

SurfaceFlingerHang [GB reboot， ICS/JB Hang， KK Hang long time then reboot]

Display subsystemHang [Hang]

LCMHang [Hang]

Input DriverFail [Hang]

Input SystemHang [GB， ICS Hang and JB， KK Reboot]

WindowManagerPolicyHang [Maybe]

---- Need check the serverthread state

System serverdeadlock [Reboot]

System server logicabnormal [Maybe]

Surfaceflinger logic abnormal [Maybe]

# REF

Keywords: android 系统卡死 分析

[Android的死机、重启问题分析方法](http://blog.csdn.net/jinlu7611/article/details/50592385)

# TODO

kernel.log抓取

全面的日志

timsk怎么生成的

我们自己的调试信息也写到Developr options里面吧？

[Android的死机、重启问题分析方法](http://blog.csdn.net/jinlu7611/article/details/50592385)的其他友链不错