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

# 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？