密级状态：绝密(　　)　　秘密(　　)　　内部(　)　　公开(√)

死机问题快速分析指南

（技术部，第二系统产品部）

文件状态： [　] 正在修改 [√] 正式发布	当前版本：	V1.0
	作　　者：	郑建
	完成日期：	2019-03-06
	审　　核：	
	完成日期：	2019-03-12

福州瑞芯微电子股份有限公司

Fuzhou Rockchips Electronics Co., Ltd

版 本 历 史

版本号	作者	修改日期	修改说明	备注
V1.0	郑建	2019-03-06	发布初始版本	

目 录

概述

本文主要介绍针对 Android 系统死机问题的分析策略和思路，以及 Android debug 中可能用到的一些工具、概念和调试手段。

1. 死机问题

1.1 死机现象

1.1.1 死机的定义

当手机长时间无法再被用户控制操作时，我们称为死机。

1.1.2 死机的状态

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。

机器死机后重启。

死机问题的几种情况

- 有死机现场

信息量较为充足，可以使用 adb、串口等分析现场，需要注意的是死机只是最终的结果，即发生 bug 之后手机最后的一个状态，很多时候死机会打印很多服务异常日志，这些异常日志和最终查出来的原因模块可能并没有很强的相关性，只是被连带"误伤"，真正的根本原因可能就埋藏在里面，我们的重要工作之一是要找出最早开始出问题时候的背景、用户操作等，结合他们来为模块定位、查找根本原因提供资讯，最终还原出整个问题的来龙去脉。

- 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。这种情况，死机时的状态重启后可能会被恢复好，以及经常会出现 log 缓冲已经被冲掉的情况，看不到最初发生问题时的实时日志。Debug 时有时候需要把重启问题变成死机问题，方便调试。

- 仅仅一些 LOG 或者其他的资讯

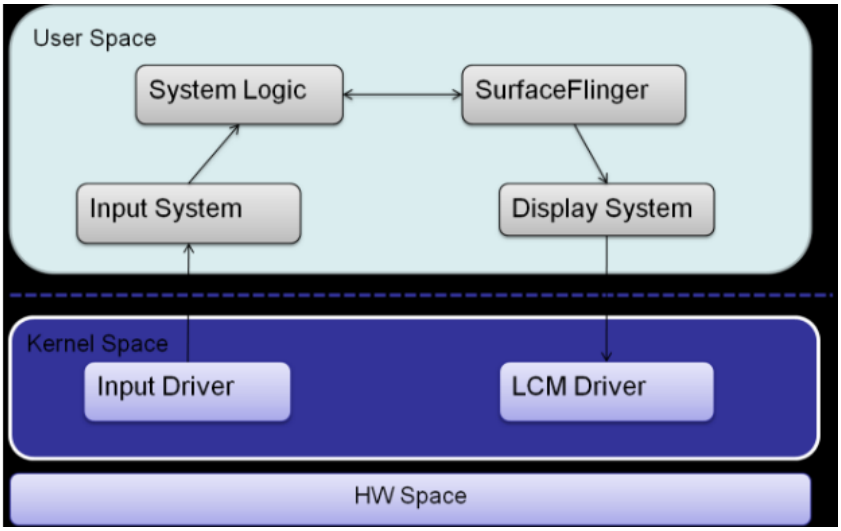
需要从这些 LOG 或者资讯中猜测，当时手机的状态，以及可能死机的原因。

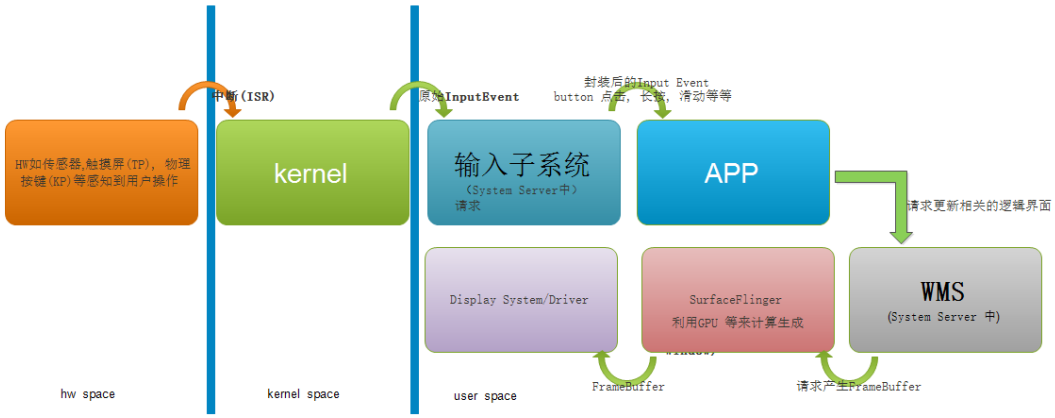
有死机现场的情况下是最容易分析的。而如果仅仅只有一些 LOG 的话，就需要工程师具有非常丰富的经验，从仅有的 LOG 中，提取有价值的资讯，来猜测出当时死机的原因。

强烈建议客户一旦遇到死机，务必保留好现场，以便快速分析解决问题。

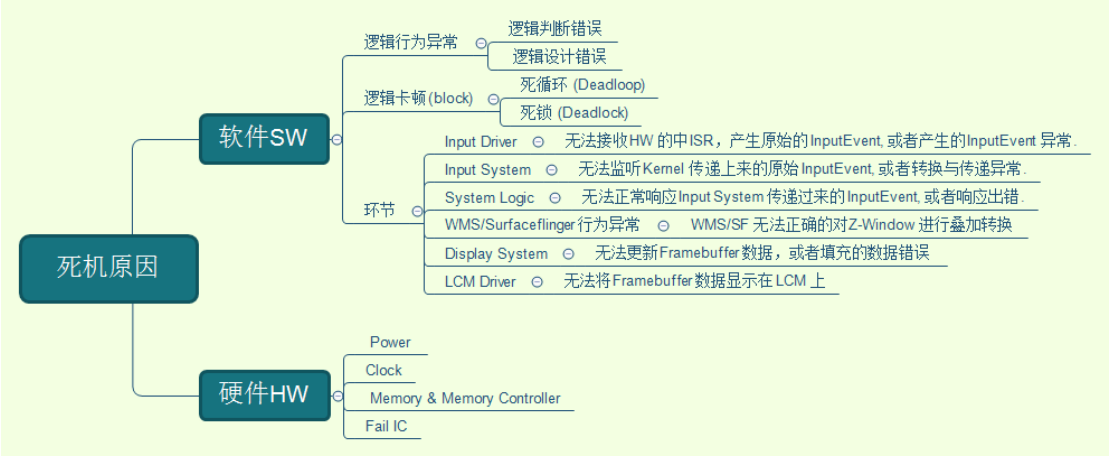
1.1.3 输入子系统、显示子系统概览

死机之后无法控制手机、黑屏或画面卡住，主要与输入子系统、显示子系统有关。





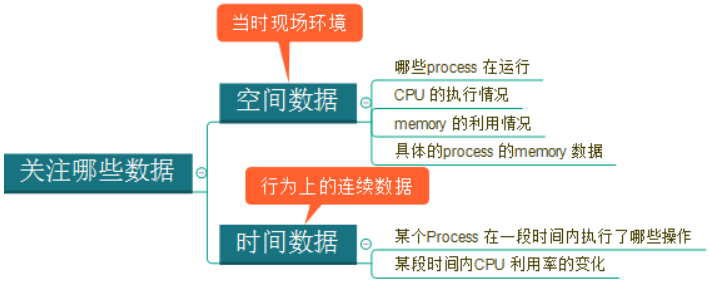
每一步出现问题，都可能引发死机问题。



2. 死机分析资讯与工具

2.1 死机分析数据

- 更宏观一点，具体关注哪些数据：



- 更细化的关注哪些数据：

1) 依靠各种技术方法去诊断当时手机运行的真实状态。

2) 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。

2.2 Backtrace 分析

- 在 ENG Build 中

```
adb remount
adb shell chmod 0777 data/anr
adb shell kill -3 pid
adb pull /data/anr
```

- 在 User Build 中

没有 root 权限的情况下，只能直接 pull 已经存在的 backtrace：

```
adb pull /data/anr
```

- 添加代码直接抓取某个运行时机的 backtrace

(1) 方法一：

```
Thread.dumpStack();
java.lang.Thread.dumpStack();
```

(2) 方法二：

```
Log.d(TAG,"run:",new Throwable());
```

2.2.1 JavaBacktrace 解析

下面是一小段 system server 的 java backtrace 的开始：

```
1. ----- pid 682 at 2014-07-30 18:04:53 -----
2. Cmd line: system_server
3. JNI: CheckJNI is off; workarounds are off; pins=4; globals=1484 (plus 50 weak)
4. DALVIK THREADS:
5. (mutexes: tll=0 tsl=0 tscl=0 ghl=0)
6. "main" prio=5 tid=1 NATIVE
7.   | group="main" sCount=1 dsCount=0 obj=0x4193fde0 self=0x418538f8
8.   | sysTid=682 nice=-2 sched=0/0 cgrp=apps handle=1074835940
```

```
9.    | state=S schedstat=( 47858718206 26265263191 44902 ) utm=4074 stm=711
      core=0
10.   at android.os.MessageQueue.nativePollOnce(Native Method)
11.   at android.os.MessageQueue.next(MessageQueue.java:138)
12.   at android.os.Looper.loop(Looper.java:150)
13.   at com.android.server.ServerThread.initAndLoop(SystemServer.java:1468)
14.   at com.android.server.SystemServer.main(SystemServer.java:1563)
15.   at java.lang.reflect.Method.invokeNative(Native Method)
16.   at java.lang.reflect.Method.invoke(Method.java:515)
17.   at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:
      829)
18.   at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:645)
19.   at dalvik.system.NativeStart.main(Native Method)
```

一行一行来解析：

* 0# 最开始是 -----PID at Time 然后接着 Cmd line: process name

* 1# the backtrace header: dvm thread ：“DALVIK THREADS:”

* 2# Global DVM mutex value: if 0 unlock, else lock

tll: threadListLock,

tsl: threadSuspendLock,

tscl: threadSuspendCountLock

ghl: gcHeapLock

(mutexes: tll=0 tsl=0 tscl=0 ghl=0)

* 3# thread name, java thread Priority, [daemon], DVM thread id, DVM thread status.

"main" -> main thread -> activity thread

prio -> java thread priority default is 5, (nice =0, linux thread priority 120),

domain is [1,10]

DVM thread id, NOT linux thread id

DVM thread Status:

ZOMBIE, RUNNABLE, TIMED_WAIT, MONITOR, WAIT, INITALIZING,STARTING, NATIVE, VMWAIT, SUSPENDED,UNKNOWN

"main" prio=5 tid=1 NATIVE

* 4# DVM thread status

group: default is “main”

Compiler,JDWP,Signal Catcher,GC,FinalizerWatchdogDaemon,FinalizerDaemon,ReferenceQueueDaemon are system group

sCount: thread suspend count

dsCount: thread dbg suspend count

obj: thread obj address

Sef: thread point address

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

* #5 Linux thread status

sysTId: linux thread tid

Nice: linux thread nice value

sched: cgroup policy/gourp id

cgrp: c group

handle: handle address

sysTid=682 nice=-2 sched=0/0 cgrp=apps handle=1074835940

* #6 CPU Sched stat

Schedstat (Run CPU Clock/ns, Wait CPU Clock/ns, Slice times)

utm: utime, user space time(jiffies)

stm: stime, kernel space time(jiffies)

Core now running in cpu.

state=S schedstat=(47858718206 26265263191 44902) utm=4074 stm=711 core=0

* #7-more Call Stack

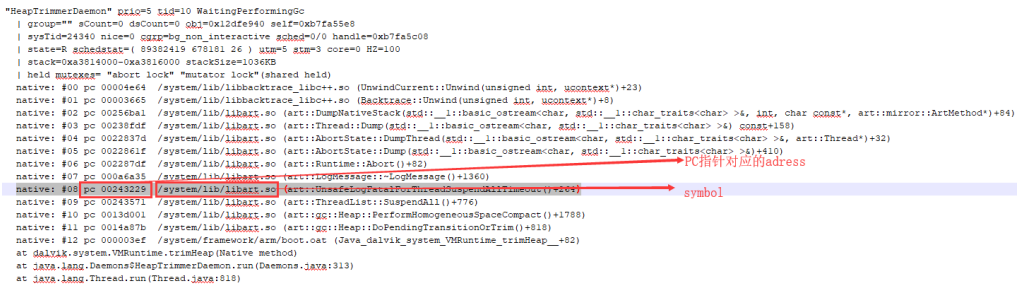
2.2.2 Native Backtrace

- 利用 debuggerd 抓取

  利用 debuggerd 的执行命令是：

```
debuggerd –b [pid]
```

  示例如下：



- 添加代码直接抓取某个运行时机的 backtrace 的方法：

  Google 默认提供了 CallStack API，参考：

```
system/core/include/libutils/CallStack.h
system/core/libutils/CallStack.cpp
```

  可快速打印单个线程的 backtrace。

  在 cpp 文件添加如下信息：

```
#include <utils/CallStack.h>
...
status_t AudioHardware::setIncallPath_l(uint32_t device) {
...
#ifdef _ARM_
    android::CallStack stack;
```

```
    stack.update(1, 100);
    stack.dump("");
    stack.log("test",ANDROID_LOG_VERBOSE,"");
#endif
...
}
```

在 Android.mk 中，加入：

```
LOCAL_CFLAGS += -D_ARM_
LOCAL_SHARED_LIBRARIES += libutils
```

这样将会打印上面所述的调用信息，便于分析代码，debug，定位问题。

- **解析 Native Backtrace**

可以使用 addr2line 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
```

示例如下：

```
zj@RD-DEP1-SERVER-163:~/work/3288/5.1_internal$ arm-linux-androideabi-addr2line -f -C -e out/target/product/rk3288/symbols/system/lib/libart.
so 00243229
UnsafeLogFatalForThreadSuspendAllTimeout
/home2/zj/work/3288/5.1_internal/art/runtime/thread_list.cc:173
```

2.2.3 Kernel Backtrace 抓取方式

- Proc System

```
cat proc/pid/task/tid/stack
```

- Sysrq-trigger

```
adb shell cat proc/kmsg > kmsg.txt
adb shell "echo 8 > proc/sys/kernel/printk" //修改 printk loglevel
adb shell "echo t > /proc/sysrq-trigger" //打印所有的 backtrace
adb shell "echo w > /proc/sysrq-trigger"//打印'-D' status 'D' 的 process
```

- 添加代码直接抓取的方法：

```
#include <linux/sched.h>
当前 thread:  dump_stack();
其他 thread:  show_stack(task, NULL);
```

- Process/Thread 状态

```
"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 (disk sleep)，T (stopped)，Z (zombie) 等就要认真审查。

几种典型的异常情况

- **Deadlock**

下面这个 case 可以看到 PowerManagerService，ActivityManager，WindowManager 相互之间发生 deadlock。

```
1.  "PowerManagerService" prio=5 tid=25 MONITOR
2.    | group="main" sCount=1 dsCount=0 obj=0x42bae270 self=0x6525d5c0
3.    | sysTid=913 nice=0 sched=0/0 cgrp=apps handle=1696964440
4.    | state=S schedstat=( 5088939411 10237027338 34016 ) utm=232 stm=276 co
    re=2
5.    at com.android.server.am.ActivityManagerService.bindService(ActivityManagerSe
    rvice.java:~14079)
6.    - waiting to lock <0x42aa0f78> (a com.android.server.am.ActivityManagerServi
    ce) held by tid=16 (ActivityManager)
7.    at android.app.ContextImpl.bindServiceCommon(ContextImpl.java:1665)
8.    at android.app.ContextImpl.bindService(ContextImpl.java:1648)
9.    at com.android.server.power.PowerManagerService.bindSmartStandByService(Po
    werManagerService.java:4090)
```

```
10.  at com.android.server.power.PowerManagerService.handleSmartStandBySettingC
     hangedLocked(PowerManagerService.java:4144)
11.  at com.android.server.power.PowerManagerService.access$5600(PowerManagerS
     ervice.java:102)
12.  at com.android.server.power.PowerManagerService$SmartStandBySettingObserv
     er.onChange(PowerManagerService.java:4132)
13.  at android.database.ContentObserver$NotificationRunnable.run(ContentObserver.
     java:181)
14.  at android.os.Handler.handleCallback(Handler.java:809)
15.  at android.os.Handler.dispatchMessage(Handler.java:102)
16.  at android.os.Looper.loop(Looper.java:139)
17.  at android.os.HandlerThread.run(HandlerThread.java:58)
18.
19.  "ActivityManager" prio=5 tid=16 MONITOR
20.  | group="main" sCount=1 dsCount=0 obj=0x42aa0d08 self=0x649166b0
21.  | sysTid=902 nice=-2 sched=0/0 cgrp=apps handle=1687251744
22.  | state=S schedstat=( 39360881460 25703061063 69675 ) utm=1544 stm=239
     2 core=2
23.  at com.android.server.wm.WindowManagerService.setAppVisibility(WindowManag
     erService.java:~4783)
24.  - waiting to lock <0x42d17590> (a java.util.HashMap) held by tid=12 (Window
     Manager)
25.  at com.android.server.am.ActivityStack.stopActivityLocked(ActivityStack.java:24
     32)
26.  at com.android.server.am.ActivityStackSupervisor.activityIdleInternalLocked(Acti
     vityStackSupervisor.java:2103)
27.  at com.android.server.am.ActivityStackSupervisor$ActivityStackSupervisorHandle
     r.activityIdleInternal(ActivityStackSupervisor.java:2914)
28.  at com.android.server.am.ActivityStackSupervisor$ActivityStackSupervisorHandle
     r.handleMessage(ActivityStackSupervisor.java:2921)
```

```
29.  at android.os.Handler.dispatchMessage(Handler.java:110)
30.  at android.os.Looper.loop(Looper.java:147)
31.  at com.android.server.am.ActivityManagerService$AThread.run(ActivityManagerS
     ervice.java:2112)
32.
33.  "WindowManager" prio=5 tid=12 MONITOR
34.  | group="main" sCount=1 dsCount=0 obj=0x42a92550 self=0x6491dd48
35.  | sysTid=898 nice=-4 sched=0/0 cgrp=apps handle=1687201104
36.  | state=S schedstat=( 60734070955 41987172579 219755 ) utm=4659 stm=14
     14 core=1
37.  at com.android.server.power.PowerManagerService.setScreenBrightnessOverride
     FromWindowManagerInternal(PowerManagerService.java:~3207)
38.  - waiting to lock <0x42a95140> (a java.lang.Object) held by tid=25 (PowerMan
     agerService)
39.  at com.android.server.power.PowerManagerService.setScreenBrightnessOverride
     FromWindowManager(PowerManagerService.java:3196)
40.  at com.android.server.wm.WindowManagerService.performLayoutAndPlaceSurfac
     esLockedInner(WindowManagerService.java:9686)
41.  at com.android.server.wm.WindowManagerService.performLayoutAndPlaceSurfac
     esLockedLoop(WindowManagerService.java:8923)
42.  at com.android.server.wm.WindowManagerService.performLayoutAndPlaceSurfac
     esLocked(WindowManagerService.java:8879)
43.  at com.android.server.wm.WindowManagerService.access$500(WindowManagerS
     ervice.java:170)
44.  at com.android.server.wm.WindowManagerService$H.handleMessage(WindowMa
     nagerService.java:7795)
45.  at android.os.Handler.dispatchMessage(Handler.java:110)
46.  at android.os.Looper.loop(Looper.java:147)
47.  at android.os.HandlerThread.run(HandlerThread.java:58)
```

- 执行 **JNI native code** 后一去不复返

```
1.  "main" prio=5 tid=1 NATIVE
2.    | group="main" sCount=1 dsCount=0 obj=0x41bb3d98 self=0x41ba2878
3.    | sysTid=814 nice=-2 sched=0/0 cgrp=apps handle=1074389380
4.    | state=D schedstat=( 22048890928 19526803112 32612 ) utm=1670 stm=534
    core=0
5.    (native backtrace unavailable)
6.    at android.hardware.SystemSensorManager$BaseEventQueue.nativeDisableSenso
    r(Native Method)
7.    at android.hardware.SystemSensorManager$BaseEventQueue.disableSensor(Syst
    emSensorManager.java:399)
8.    at android.hardware.SystemSensorManager$BaseEventQueue.removeAllSensors(
    SystemSensorManager.java:325)
9.    at android.hardware.SystemSensorManager.unregisterListenerImpl(SystemSenso
    rManager.java:194)
10.   at android.hardware.SensorManager.unregisterListener(SensorManager.java:560)

11.   at com.android.internal.policy.impl.WindowOrientationListener.disable(WindowOri
    entationListener.java:139)
12.   at com.android.internal.policy.impl.PhoneWindowManager.updateOrientationListe
    nerLp(PhoneWindowManager.java:774)
13.   at com.android.internal.policy.impl.PhoneWindowManager.screenTurnedOff(Phone
    WindowManager.java:4897)
14.   at com.android.server.power.Notifier.sendGoToSleepBroadcast(Notifier.java:518)

15.   at com.android.server.power.Notifier.sendNextBroadcast(Notifier.java:434)
16.   at com.android.server.power.Notifier.access$500(Notifier.java:63)
17.   at com.android.server.power.Notifier$NotifierHandler.handleMessage(Notifier.java
    :584)
18.   at android.os.Handler.dispatchMessage(Handler.java:110)
19.   at android.os.Looper.loop(Looper.java:193)
```

```
20.  at com.android.server.ServerThread.initAndLoop(SystemServer.java:1436)
21.  at com.android.server.SystemServer.main(SystemServer.java:1531)
22.  at java.lang.reflect.Method.invokeNative(Native Method)
23.  at java.lang.reflect.Method.invoke(Method.java:515)
24.  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:
     824)
25.  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:640)
26.  at dalvik.system.NativeStart.main(Native Method)
```

===>

```
1.  KERNEL SPACE BACKTRACE, sysTid=814
2.   [<ffffffff>] 0xffffffff from [<c07e5140>] __schedule+0x3fc/0x950
3.   [<c07e4d50>] __schedule+0xc/0x950 from [<c07e57e4>] schedule+0x40/0x
     80
4.   [<c07e57b0>] schedule+0xc/0x80 from [<c07e5ae4>] schedule_preempt_disa
     bled+0x20/0x2c
5.   [<c07e5ad0>] schedule_preempt_disabled+0xc/0x2c from [<c07e3c3c>] mute
     x_lock_nested+0x1b8/0x560
6.   [<c07e3a90>] mutex_lock_nested+0xc/0x560 from [<c05667d8>] gsensor_o
     perate+0x1bc/0x2c0
7.   [<c0566628>] gsensor_operate+0xc/0x2c0 from [<c0495fa0>] hwmsen_enab
     le+0xa8/0x30c
8.   [<c0495f04>] hwmsen_enable+0xc/0x30c from [<c0496500>] hwmsen_unloc
     ked_ioctl+0x2fc/0x528
9.   [<c0496210>] hwmsen_unlocked_ioctl+0xc/0x528 from [<c018ad98>] do_vfs
     _ioctl+0x94/0x5bc
10.  [<c018ad10>] do_vfs_ioctl+0xc/0x5bc from [<c018b33c>] sys_ioctl+0x7c/0x
     8c
11.  [<c018b2cc>] sys_ioctl+0xc/0x8c from [<c000e480>] ret_fast_syscall+0x0/0
     x40
12.  [<ffffffff>]  from [<ffffffff>]
```

2.3 系统运行环境分析

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 在疯狂地执行，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。

2.3.1 CPU & Memory & IO & Storage Usage

追查 CPU 利用率可大体地知道，当时机器是否有 Process 在疯狂地运行，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.
```

注意的是 top 的 CPU%是按全部 CPU 来计算的，如果以单线程来计算，比如当时有开启 4 个核心，那么最多吃到 25%。

常见的操作方式如： top -m 10

```
1.   User 10%, System 6%, IOW 0%, IRQ 0%
2.  ser 83 + Nice 39 + Sys 82 + Idle 970 + IOW 11 + IRQ 0 + SIRQ 3 = 1188
3.
4.   PID PR CPU% S  #THR    VSS    RSS PCY UID      Name
5.  1631  1   6% S   22 875172K  36904K  fg u0_a54   derhino.baidumapdemo:remote
```

```
450  1   4% S    90 1494368K  78692K  fg system   system_server
1608  2   2% S    45 943132K  69696K  fg u0_a54   derhino.baidumapdemo
153  2   1% S    25 153808K  14740K  fg system   /system/bin/surfaceflinger
779  1   0% S    26 875724K  36388K  fg radio    com.android.phone
2066  1   0% R     1   2496K    968K  fg root     top
147  1   0% S     5   7400K   1832K  fg logd     /system/bin/logd
150  2   0% S     1   1240K    352K  fg system   /system/bin/servicemanager
102  0   0% S     1      0K      0K  fg root     mmcqd/0
41  0   0% S     1      0K      0K  fg root     kworker/u8:1
```

从上面可以看出，系统基本运行正常，没有很吃 CPU 的进程。

I.E 异常情况

```
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 PCY UID      Thread          Proc
16132 32195  3  14% R 997100K  53492K  bg u0_a60   Thread-1401     com.android.mms
16132 32190  1  14% R 997100K  53492K  bg u0_a60   Thread-1400     com.android.mms
16132 32188  2  14% R 997100K  53492K  bg u0_a60   Thread-1399     com.android.mms
16132 32187  0  14% R 997100K  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 PCY UID      Thread          Proc
16132 32195  3  16% R 997100K  53492K  bg u0_a60   Thread-1401     com.android.mms
```

```
16132 32188  2  16% R 997100K  53492K  bg u0_a60   Thread-1399    com.android.mms
16132 32190  0  16% R 997100K  53492K  bg u0_a60   Thread-1400    com.android.mms
16132 32187  1  16% R 997100K  53492K  bg u0_a60   Thread-1398    com.android.mms
18793 18793  4   2% R   2196K   1284K    shell    top            top
```

可以看到，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四个 backtrace，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。

```
"Thread-1401" prio=5 tid=32 SUSPENDED JIT
  | group="main" sCount=1 dsCount=0 obj=0x4264f860 self=0x7b183558
  | sysTid=32195 nice=0 sched=0/0 cgrp=apps/bg_non_interactive handle=2078705952
  | state=S schedstat=( 3284456714198 104216273858 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=31 SUSPENDED JIT
  | group="main" sCount=1 dsCount=0 obj=0x41f5d8f0 self=0x7be2a8e8
  | sysTid=32190 nice=0 sched=0/0 cgrp=apps/bg_non_interactive handle=2078029504
  | state=S schedstat=( 3284905134412 105526230562 382946 ) utm=324805 stm=3685 core=5
  at com.yulong.android.mms.ui.MmsChatActivity$37.run(MmsChatActivity.java:~7586)
```

```
14.  at java.lang.Thread.run(Thread.java:841)
15.
16. "Thread-1399" prio=5 tid=30 SUSPENDED JIT
17.  | group="main" sCount=1 dsCount=0 obj=0x42564d28 self=0x7b0e6838
18.  | sysTid=32188 nice=0 sched=0/0 cgrp=apps/bg_non_interactive handle=20776
     62640
19.  | state=S schedstat=( 3288042313685 103203810616 375959 ) utm=325143 st
     m=3661 core=7
20.  at com.yulong.android.mms.ui.MmsChatActivity$37.run(MmsChatActivity.java:~7
     586)
21.  at java.lang.Thread.run(Thread.java:841)
22.
23. "Thread-1398" prio=5 tid=29 SUSPENDED
24.  | group="main" sCount=1 dsCount=0 obj=0x4248e5a8 self=0x7be0d128
25.  | sysTid=32187 nice=0 sched=0/0 cgrp=apps/bg_non_interactive handle=20792
     51904
26.  | state=S schedstat=( 3287248372432 105116936413 379634 ) utm=325055 st
     m=3669 core=6
27.  at com.yulong.android.mms.ui.MmsChatActivity$37.run(MmsChatActivity.java:~7
     586)
28.  at java.lang.Thread.run(Thread.java:841)
```

mpstat

mpstat 是一个常用的多核 CPU 性能分析工具，用来实时查看每个 CPU 的性能指标，以及所有 CPU 的平均指标，一般地，建议先用 mpstat 来初步快速地统计 CPU 的负载，方便下一步的定位。

```
busybox mpstat
```

当 mpstat 不带参数时，输出为从系统启动以来的平均值，可作为平均负载参考。

```
127|rk3399_all:/system/bin # busybox mpstat
Linux 4.4.126 (localhost)       01/19/13        _aarch64_       (6 CPU)

14:51:25     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
14:51:25     all   13.29    0.02    2.22    0.01    0.00    0.03    0.00    0.00   84.44
```

-P ALL 表示监控所有 CPU，后面数字 2 表示间隔 2 秒后输出一组数据。

```
busybox mpstat -P ALL 2
```

```
rk3399_all:/system/bin # busybox mpstat -P ALL 2
Linux 4.4.126 (localhost)       01/19/13        _aarch64_       (6 CPU)

14:46:31     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
14:46:33     all   11.05    0.00    8.77    0.00    0.00    0.17    0.00    0.00   80.02
14:46:33       0   22.77    0.00   14.85    0.00    0.00    0.99    0.00    0.00   61.39
14:46:33       1   20.00    0.00   11.50    0.00    0.00    0.00    0.00    0.00   68.50
14:46:33       2   17.62    0.00   18.13    0.00    0.00    0.00    0.00    0.00   64.25
14:46:33       3    5.08    0.00    6.09    0.00    0.00    0.00    0.00    0.00   88.83
14:46:33       4    0.51    0.00    2.53    0.00    0.00    0.00    0.00    0.00   96.97
14:46:33       5    0.51    0.00    0.00    0.00    0.00    0.00    0.00    0.00   99.49

14:46:33     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
14:46:35     all   10.07    0.09    9.81    0.00    0.00    0.09    0.00    0.00   79.95
14:46:35       0   16.95    0.56   18.64    0.00    0.00    0.56    0.00    0.00   63.28
14:46:35       1   13.24    0.00   15.20    0.00    0.00    0.00    0.00    0.00   71.57
14:46:35       2   15.43    0.00   12.77    0.00    0.00    0.00    0.00    0.00   71.81
14:46:35       3   12.76    0.00   11.22    0.00    0.00    0.00    0.00    0.00   76.02
14:46:35       4    1.03    0.00    1.03    0.00    0.00    0.00    0.00    0.00   97.95
14:46:35       5    1.50    0.00    0.50    0.00    0.00    0.00    0.00    0.00   98.00

14:46:35     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
14:46:37     all    8.55    0.00    8.11    0.00    0.00    0.09    0.00    0.00   83.25
14:46:37       0   12.92    0.00   14.04    0.00    0.00    0.56    0.00    0.00   72.47
14:46:37       1   18.58    0.00   19.13    0.00    0.00    0.00    0.00    0.00   62.30
14:46:37       2    7.03    0.00    7.03    0.00    0.00    0.00    0.00    0.00   85.95
14:46:37       3   12.82    0.00    8.21    0.00    0.00    0.00    0.00    0.00   78.97
14:46:37       4    1.02    0.00    1.52    0.00    0.00    0.00    0.00    0.00   97.46
14:46:37       5    0.50    0.00    0.50    0.00    0.00    0.00    0.00    0.00   98.99
```

pidstat

pidstat 是一个常用的进程性能分析工具，用来实时查看进程的 CPU、内存、I/O 以及上下文切换等性能指标，用于初步定位系统负载瓶颈在 CPU、内存或者磁盘 IO，缩小排查范围，建议在使用 mpstat 初步定位之后，使用 pidstat 进一步定位负载瓶颈。

pidstat 主要用于监控全部或指定进程占用系统资源的情况，如 CPU、内存、设备 IO、任务切换、线程等。pidstat 首次运行时显示自系统启动开始的各项统计信息，之后运行 pidstat 将显示自上次运行该命令以后的统计信息。用户可以通过指定统计的次数和时间来获得所需的统计信息。

默认参数

执行 pidstat，将输出系统启动后所有活动进程的 CPU 统计信息：

```
pidstat 1
```

命令输出进程的 CPU 占用率，该命令会持续输出，并且不会覆盖之前的数据，可以方便观察系统动态。

```
rk3399_all:/system/bin # pidstat 1
Linux 4.4.126 (localhost)       01/19/13        _armv81_        (6 CPU)

15:01:40      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
15:01:41        0         7    0.00    0.95    0.00    1.90    0.95     2  rcu_preempt
15:01:41        0       165    0.00    0.95    0.00    0.00    0.95     0  irq/122-inv_irq
15:01:41     1000       233   17.14   15.24    0.00    0.00   32.38     3  surfaceflinger
15:01:41     1041      1535    4.76    1.90    0.00    0.00    6.67     0  audioserver
15:01:41     1000      1675    2.86    1.90    0.00    0.00    4.76     3  system_server
15:01:41    10057      3979   36.19   33.33    0.00    0.00   69.52     0  com.tencent.wok
15:01:41        0      5025    0.00    0.95    0.00    0.00    0.95     1  kworker/u13:1
15:01:41        0      5133    0.00    1.90    0.00    0.95    1.90     2  kworker/u13:2
15:01:41        0      5135    3.81    2.86    0.00    0.00    6.67     5  pidstat

15:01:41      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
15:01:42        0         7    0.00    0.99    0.00    0.99    0.99     1  rcu_preempt
15:01:42        0       165    0.00    0.99    0.00    0.00    0.99     0  irq/122-inv_irq
15:01:42     1000       233   18.81   13.86    0.00    0.00   32.67     3  surfaceflinger
15:01:42     1041      1535    2.97    2.97    0.00    0.00    5.94     0  audioserver
15:01:42     1000      1675    1.98    1.98    0.00    0.00    3.96     3  system_server
15:01:42    10057      3979   36.63   31.68    0.00    0.00   68.32     0  com.tencent.wok
15:01:42        0      5025    0.00    1.98    0.00    0.00    1.98     2  kworker/u13:1
15:01:42        0      5105    0.00    0.99    0.00    0.99    0.99     2  kworker/u12:0
15:01:42        0      5133    0.00    2.97    0.00    0.00    2.97     2  kworker/u13:2
15:01:42        0      5135    1.98    3.96    0.00    0.00    5.94     5  pidstat
```

指定采样周期和采样次数

pidstat 命令指定采样周期和采样次数，命令形式为”pidstat [option] interval [count]”，以下 pidstat 输出以 2 秒为采样周期，输出 10 次 CPU 使用统计信息：

```
pidstat 2 10
```

CPU 使用情况统计(-u)

使用-u 选项，pidstat 将显示各活动进程的 CPU 使用统计，执行”pidstat -u”与单独执行”pidstat”的效果一样。

内存使用情况统计(-r)

使用-r 选项，pidstat 将显示各活动进程的内存使用统计，针对特定进程统计(-p)：

```
pidstat -r -p 233 1
```

```
rk3399_all:/system/bin # pidstat -r -p 233 1
Linux 4.4.126 (localhost)       01/20/13        _armv81_        (6 CPU)

06:45:11      UID       PID  minflt/s  majflt/s      VSZ      RSS   %MEM  Command
06:45:12     1000       233      0.00      0.00   713324   131452   3.34  surfaceflinger
06:45:13     1000       233    109.00      0.00   685252   140876   3.58  surfaceflinger
```

以上各列输出的含义如下：

minflt/s：每秒次缺页错误次数(minor page faults)，次缺页错误次数意即虚拟内存地址映射成物理内存地址产生的 page fault 次数。

majflt/s：每秒主缺页错误次数(major page faults)，当虚拟内存地址映射成物理内存地址时，相应的 page 在 swap 中，这样的 page fault 为 major page fault，一般在内存使用紧张时产生。

VSZ：　　该进程使用的虚拟内存（以 kB 为单位）。

RSS：　　该进程使用的物理内存（以 kB 为单位）。

%MEM：　该进程使用内存的百分比。

Command：拉起进程对应的命令。

IO 情况统计(-d)

使用-d 选项，我们可以查看进程 IO 的统计信息：

```
pidstat –d 1
```

```
rk3399_a11:/system/bin # pidstat -d 1
Linux 4.4.126 (localhost)       01/20/13        _armv81_        (6 CPU)

06:49:57      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
06:49:58    10057      8911      0.00     62.96      0.00       0  com.tencent.wok

06:49:58      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
06:49:59        0       320      0.00      4.00      0.00       0  logcatext

06:49:59      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command

06:50:00      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
06:50:01        0      8791      0.00      8.00      0.00       0  kworker/u12:0
^C

Average:      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
Average:        0       320      0.00      0.98      0.00       0  logcatext
Average:        0      8791      0.00      1.96      0.00       0  kworker/u12:0
Average:    10057      8911      0.00     16.67      0.00       0  com.tencent.wok
```

输出信息含义：

kB_rd/s：每秒进程从磁盘读取的数据量（以 kB 为单位）。

kB_wr/s：每秒进程向磁盘写的数据量（以 kB 为单位）。

Command：拉起进程对应的命令。

pidstat 常用命令

使用 pidstat 进行问题定位时，以下命令常被用到：

```
pidstat -u 1
```

```
pidstat -r 1
pidstat -d 1
```

以上命令以 1 秒为信息采集周期，分别获取 CPU、内存和磁盘 IO 的统计信息。

2.3.2 Memory Usage

Memory Usage，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 查看整机内存使用情况
adb shell cat proc/pid/maps 显示进程映射了的内存区域和访问权限
adb shell cat proc/pid/smaps
```

* procrank info: all process memory status

```
adb shell procrank
adb shell procmem pid
adb shell dumpsys meminfo pid
```

2.3.3 Storage Usage

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。

通常使用如 df

```
shell@rk3288:/ # df
Filesystem               Size     Used     Free   Blksize
/dev                  1006.3M    36.0K  1006.3M   4096
/sys/fs/cgroup        1006.3M     0.0K  1006.3M   4096
/mnt/asec             1006.3M     0.0K  1006.3M   4096
/mnt/obb              1006.3M     0.0K  1006.3M   4096
/system                  1.5G   567.0M   942.9M   4096
/cache                 122.0M   132.0K   121.9M   4096
/metadata               11.7M    40.0K    11.7M   4096
/data                  991.9M   356.1M   635.8M   4096
/mnt/shell/emulated    991.9M   356.1M   635.8M   4096
/mnt/internal_sd         4.4G    11.3M     4.4G   8192
/mnt/secure/asec         4.4G    11.3M     4.4G   8192
```

以及 ls，du 等命令，如 du

du -help

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

```
du -LPh -d 1
```

```
shell@rk3288:/ # du -LPh -d 1
8.0K    ./metadata
0B      ./storage
0B      ./config
100K    ./cache
0B      ./acct
11M     ./mnt
566M    ./system
0B      ./sys
1.3M    ./sbin
64K     ./res
du: ./proc/450/task/461/fd/212: No such file or directory
0B      ./proc
355M    ./data
0B      ./root
36K     ./dev
935M    .
```

2.4 死机问题需要知道的资讯

为了能够更好更快的分析死机问题，通常我们都强烈建议客户保存好现场，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地纪录当时现场环境，提供尽量多的信息量：

- 发现死机的时间

(1) 如果是发现时，感觉机器早已经死机，需要说明。

(2) 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

(1) 强调在正常使用到死机过程中的操作，这部分有助于配合 log 还原死机前的实际场景。

(2) 环境状态通常包括温度，湿度，网络信号状况。

(3) 复现流程的视频。

- 复现手机情况

(1) 复现的软件版本：版本号？USER/ENG Build？

(2) 外部设备情况：有插入 SD 卡？耳机？SIM？

(3) 软件开启情况：开启蓝牙？Wi-Fi？GPS？

- 复现的概率

多少台手机做过测试？多少台手机可以复现？

(1) 前后多少个版本可以复现？从哪个版本开始可以复现？

(2) 低概率？高概率？

其中死机的时间点至关重要，需要现场的工程师务必要比较精确地纪录。如果测试工程师提一大堆 LOG，有时可能超过 1G，分析工作量会很大。特别是有一些死机情况，相关 module 的日志不一定已经打开了，log 无法纪录到的话，可能分析了大半天，把 LOG 看完，发现一无所获。

2.5 死机的 Log 分析

- 确认死机时间，和当时是否已经死机至关重要。
- 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 来确认是否挂住。
- 如果当时 kernel 已经异常，无法抓取 log，那么 log 的价值大打折扣。
- 反过来如果当时 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。
- Log 中通常没有明确的字眼说明已经卡住，只能结合多方面的情况来做整体的推理分析。
- 从 LOG 中分析 System Server，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 等。
- 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 system server 没有直接卡住。
- 查看 Surfaceflinger，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 Binder 信息，看看 System server，Surfaceflinger 等的 IPC 情况。

2.6 Bugreport 分析

Android 为了方便开发人员分析整个系统平台和某个 app 在运行一段时间之内的所有信息，专门开发了 bugreport 工具。这个工具使用起来十分简单，只要在终端执行（linux 或者 win）。

```
adb shell bugreport > bugreport.log
```

Bugreport 内容介绍请参考：

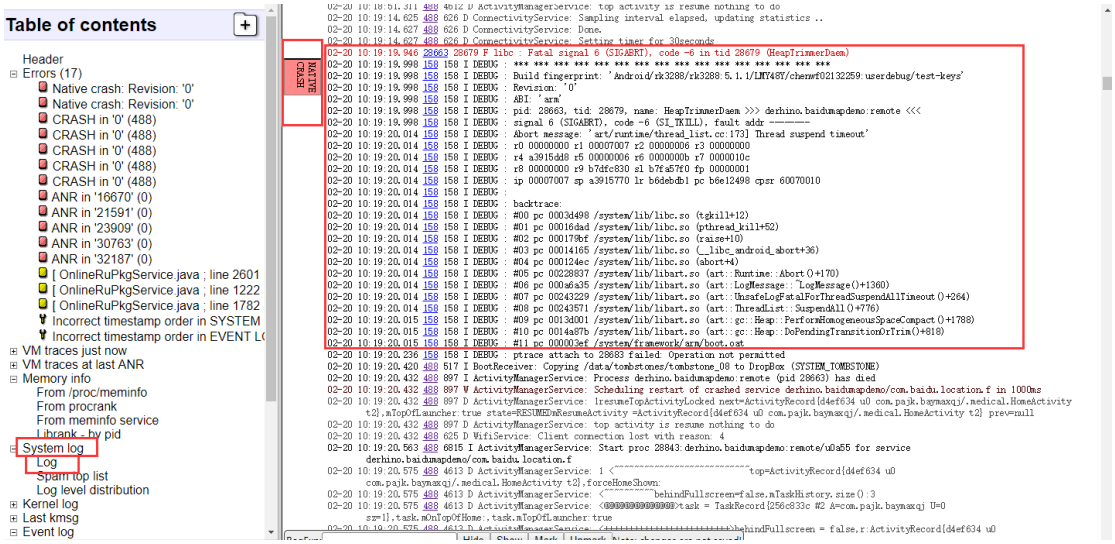
https://source.android.com/source/read-bug-reports

2.7 使用开源工具 ChkBugReport 分析 Bugreport

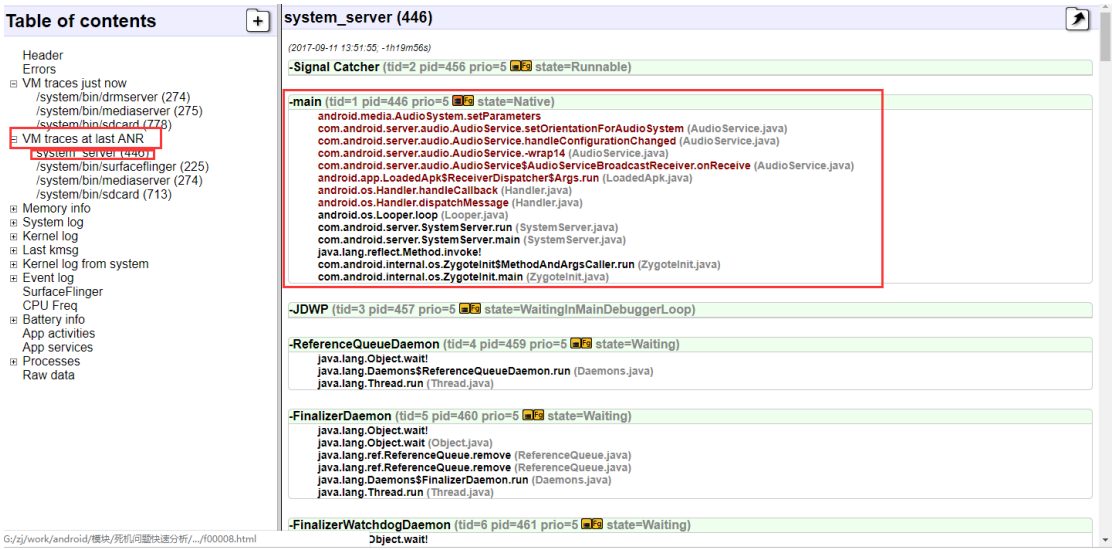
如果足够熟悉 bugreport，可以直接阅读，这里推荐一个解析 bugreport 的神器—ChkBugReport，ChkBugReport 是一个索尼移动的开发人员开发的开源工具，它可以把你得到的 bugreprot 解析成适合阅读的 html 文件。导出的 html 文件包含了根据 bugreport 数据得出的图表和分析结论，也可以用于快速分析 anr、上层死锁问题等。

https://blog.csdn.net/chenzhiqin20/article/details/12506227

示例 1：用于快速解析出疑似的死机 log



示例 2：快速解析出疑似死锁相关的堆栈



- 如果客户有提供/data/anr 下的 trace，或者相关的 db 文件或者 tombstones。在确认死机的时间下，check trace 对应的时间点，log 中的进程&线程号对应于 trace 文件中。如果时间在死机或者死机后，则是一份非常有价值的 backtrace。
- 借机审查 system server，surfaceflinger 的状况。

2.8 基本分析流程

2.8.1 测试手机使用情况

Touch Panel

- 屏幕是否有响应

  如果有响应，可能机器上层已经重启，或者当时把 ANR 认为了死机。

- 确认虚拟按键的情况,

  如果有，那么就认为当时按键事件可以传递到 System Server，此时可能 System Server 逻辑异常。

- Power Key/ Volumn Up/Down Key

  是否可以点亮，关闭屏幕（注意观察背光）。

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

  可进入 launcher 界面，说明是 lockscreen 无法解锁的情况。

- Home Key/ Back Key/Search Key：

  确认当时是否只是 APP hold 住的情况，区别 ANR 和死机的情况。

- 插入 USB 观察充电情况

可确认 Surfaceflinger，System Server 的运行情况。

2.8.2 机器状态基本检测

ps or ps -t or ps -t –p or pidstat 查看进程基本状态

- 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 rss 情况，是否有明显的溢出情况。
- 大体查看当时的 ss pid，sf pid，在 ICS 上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
- 通常正常时，只有一个 debuggerd process，pid < 200。

df 指令

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。

cat proc/meminfo， procrank，pidstat

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

(1) 摸一摸 CPU，感觉有点热，或者发烫的话，说明通常是 CPU 利用率比较高。

(2) 用 top 指令、mpstat 指令，大体上审查当前 CPU 的利用情况。

- 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。

2.8.3 USB/ADB Debug 确认死机的 module

- Input Driver-Input System

确认 Input Driver-Input System 通路是否正常，即 input driver 是否可以传上正常的输入。

最常见的检测方式是 adb shell getevent

- system-server

此分析的关键在于审查 system-server 是否还在正常的运转，这是非常重要的，System server 是整个 android 上层的中枢，容纳了最为重要的居多 service。

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

关注以下关键的系统组成部分：

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 一般是：

Java:

```
"ActivityManager" prio=5 tid=12 Native
  | group="main" sCount=1 dsCount=0 obj=0x12c55430 self=0x55a080dc10
  | sysTid=491 nice=-2 cgrp=default sched=0/0 handle=0x7f7b88c450
  | state=S schedstat=( 360674828 130746999 982 ) utm=23 stm=11 core=3 HZ=100
  | stack=0x7f7b78a000-0x7f7b78c000 stackSize=1037KB
  | held mutexes=
  kernel: __switch_to+0xac/0xb8
  kernel: SyS_epoll_wait+0x2c4/0x374
  kernel: SyS_epoll_pwait+0xb0/0x128
  kernel: el0_svc_naked+0x24/0x28
  native: #00 pc 0000000000069954  /system/lib64/libc.so (__epoll_pwait+8)
  native: #01 pc 00000000001c7b4  /system/lib64/libc.so (epoll_pwait+32)
  native: #02 pc 000000000001ad74  /system/lib64/libutils.so (_ZN7android6Looper9pollInnerEi+144)
  native: #03 pc 000000000001b154  /system/lib64/libutils.so (_ZN7android6Looper8pollOnceEiPiSl_PPv+80)
  native: #04 pc 00000000000d0fb8  /system/lib64/libandroid_runtime.so (_ZN7android18NativeMessageQueue8pollOnceEP7_JNIEnvP8_jobjecti+48)
  native: #05 pc 000000000000083c  /data/dalvik-cache/arm64/system@framework@boot.oat (Java_android_os_MessageQueue_nativePollOnce__JI+144)
  at android.os.MessageQueue.nativePollOnce(Native method)
  at android.os.MessageQueue.next(MessageQueue.java:323)
  at android.os.Looper.loop(Looper.java:135)
  at android.os.HandlerThread.run(HandlerThread.java:61)
  at com.android.server.ServiceThread.run(ServiceThread.java:46)
```

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)
```

Binder Thread

```
"Binder_4" prio=5 tid=55 Native
  | group="main" sCount=1 dsCount=0 obj=0x1317d0a0 self=0x55a09b3660
  | sysTid=763 nice=0 cgrp=default sched=0/0 handle=0x7f765b5450
  | state=S schedstat=( 372093457 110712128 1399 ) utm=23 stm=13 core=3 HZ=100
  | stack=0x7f764b9000-0x7f764bb000 stackSize=1013KB
  | held mutexes=
  kernel: __switch_to+0xac/0xb8
  kernel: binder_thread_read+0xd58/0xf1c
  kernel: binder_ioctl_write_read+0x168/0x2d0
  kernel: binder_ioctl+0x268/0x688
  kernel: do_vfs_ioctl+0x4d4/0x584
  kernel: SyS_ioctl+0x5c/0x8c
  kernel: el0_svc_naked+0x24/0x28
  native: #00 pc 0000000000069a40  /system/lib64/libc.so (__ioctl+4)
  native: #01 pc 0000000000073a64  /system/lib64/libc.so (ioctl+100)
  native: #02 pc 000000000002d584  /system/lib64/libbinder.so (_ZN7android14IPCThreadState14talkWithDriverEb+164)
  native: #03 pc 000000000002ddd8  /system/lib64/libbinder.so (_ZN7android14IPCThreadState20getAndExecuteCommandEv+24)
  native: #04 pc 000000000002def4  /system/lib64/libbinder.so (_ZN7android14IPCThreadState14joinThreadPoolEb+76)
  native: #05 pc 00000000000369e8  /system/lib64/libbinder.so (???)
  native: #06 pc 000000000001579c  /system/lib64/libutils.so (_ZN7android6Thread11_threadLoopEPv+208)
  native: #07 pc 00000000008eaa0  /system/lib64/libandroid_runtime.so (_ZN7android14AndroidRuntime15javaThreadShellEPv+96)
  native: #08 pc 0000000000014fec  /system/lib64/libutils.so (???)
  native: #09 pc 00000000000674c4  /system/lib64/libc.so (_ZL15__pthread_startPv+52)
  native: #10 pc 000000000001c154  /system/lib64/libc.so (__start_thread+16)
  (no managed stack frames)
```

(1) 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。

(2) 当发现 java backtrace 最后钻入到异常 native method 时，抓取其 native backtace，通过 native backtrace 进一步追查。

(3) 如果在 native backtrace 中，发现已经 call 入 binder driver，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 进程进一步排查。

(4) 当确认前面的 key thread 都没问题，而通过 getevent 确认 event 已经 input 到 system server。问题可能出在 input system 中。

(5) 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 如下：

```
"InputReader" prio=10 tid=27 Native
  | group="main" sCount=1 dsCount=0 obj=0x12ef83a0 self=0x55a08dcaa0
  | sysTid=655 nice=-8 cgrp=default sched=0/0 handle=0x7f7a52d450
  | state=S schedstat=( 23123750 2930375 36 ) utm=1 stm=0 core=0 HZ=100
  | stack=0x7f7a431000-0x7f7a433000 stackSize=1013KB
  | held mutexes=
  kernel: __switch_to+0xac/0xb8
  kernel: SyS_epoll_wait+0x2c4/0x374
  kernel: SyS_epoll_pwait+0xb0/0x128
  kernel: el0_svc_naked+0x24/0x28
  native: #00 pc 0000000000069954  /system/lib64/libc.so (__epoll_pwait+8)
  native: #01 pc 000000000001c7b4  /system/lib64/libc.so (epoll_pwait+32)
  native: #02 pc 00000000000281a8  /system/lib64/libinputflinger.so (_ZN7android8EventHub9getEventsEiPNS_8RawEventEm+1512)
  native: #03 pc 000000000003b2d0  /system/lib64/libinputflinger.so (_ZN7android11InputReader8loopOnceEv+168)
  native: #04 pc 0000000000035974  /system/lib64/libinputflinger.so (_ZN7android17InputReaderThread10threadLoopEv+24)
  native: #05 pc 000000000001579c  /system/lib64/libutils.so (_ZN7android6Thread11_threadLoopEPv+208)
  native: #06 pc 000000000008eaa0  /system/lib64/libandroid_runtime.so (_ZN7android14AndroidRuntime15javaThreadShellEPv+96)
  native: #07 pc 0000000000014fec  /system/lib64/libutils.so (???)
  native: #08 pc 00000000000674c4  /system/lib64/libc.so (_ZL15__pthread_startPv+52)
  native: #09 pc 000000000001c154  /system/lib64/libc.so (__start_thread+16)
  (no managed stack frames)
```

Input Dispatcher 正常的 backtrace 如下：

```
"InputDispatcher" prio=10 tid=26 Native
  | group="main" sCount=1 dsCount=0 obj=0x12ef40a0 self=0x55a08dc250
  | sysTid=654 nice=-8 cgrp=default sched=0/0 handle=0x7f7a62c450
  | state=S schedstat=( 20787207 6317208 271 ) utm=1 stm=0 core=3 HZ=100
  | stack=0x7f7a530000-0x7f7a532000 stackSize=1013KB
  | held mutexes=
  kernel: __switch_to+0xac/0xb8
  kernel: SyS_epoll_wait+0x2c4/0x374
  kernel: SyS_epoll_pwait+0xb0/0x128
  kernel: el0_svc_naked+0x24/0x28
  native: #00 pc 0000000000069954  /system/lib64/libc.so (__epoll_pwait+8)
  native: #01 pc 000000000001c7b4  /system/lib64/libc.so (epoll_pwait+32)
  native: #02 pc 000000000001ad74  /system/lib64/libutils.so (_ZN7android6Looper9pollInnerEi+144)
  native: #03 pc 000000000001b154  /system/lib64/libutils.so (_ZN7android6Looper8pollOnceEiPiS1_PPv+80)
  native: #04 pc 0000000000034534  /system/lib64/libinputflinger.so (_ZN7android15InputDispatcher12dispatchOnceEv+128)
  native: #05 pc 000000000002843c  /system/lib64/libinputflinger.so (_ZN7android21InputDispatcherThread10threadLoopEv+24)
  native: #06 pc 000000000001579c  /system/lib64/libutils.so (_ZN7android6Thread11_threadLoopEPv+208)
  native: #07 pc 000000000008eaa0  /system/lib64/libandroid_runtime.so (_ZN7android14AndroidRuntime15javaThreadShellEPv+96)
  native: #08 pc 0000000000014fec  /system/lib64/libutils.so (???)
  native: #09 pc 00000000000674c4  /system/lib64/libc.so (_ZL15__pthread_startPv+52)
  native: #10 pc 000000000001c154  /system/lib64/libc.so (__start_thread+16)
  (no managed stack frames)
```

Backtrace 通常都可以精确地定位问题点，比如卡在了哪一行。

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。

案例分析

案例 1 reboot 压力测试，概率性死机，卡在 android 动画界面

现象

反复开关机，概率性卡在 android 动画界面，重启后可以恢复。

现象分析步骤

- 这里使用 getevent –l 看到按键事件 uevent 已经上报，如果有输出按键信息，大致可以判断 kernel 有没有挂掉。

因为死机，一般人都会习惯性地按 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死机情况。

```
130|shell@rk3288:/ # getevent -l
add device 1: /dev/input/event0
  name:     "gs13673"
add device 2: /dev/input/event1
  name:     "rk29-keypad"
add device 3: /dev/input/event2
  name:     "rockchip_headset.32"


/dev/input/event1: EV_KEY       KEY_POWER            DOWN
/dev/input/event1: EV_SYN       SYN_REPORT           00000000
/dev/input/event0: EV_ABS       ABS_MT_TRACKING_ID   ffffffff
/dev/input/event0: EV_SYN       SYN_REPORT           00000000
/dev/input/event1: EV_KEY       KEY_POWER            UP
/dev/input/event1: EV_SYN       SYN_REPORT           00000000
```

KPD receives Interrupt and generate Power Key

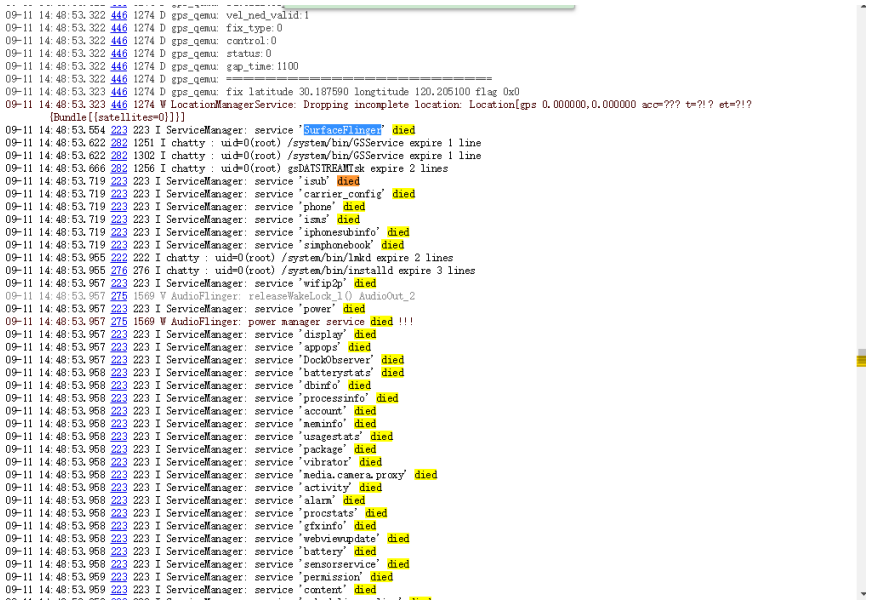
- 抓取 logcat，大致判断按键事件是否被框架服务处理，如果正常会有以下 log：

System server receives Key and call set_screen_state

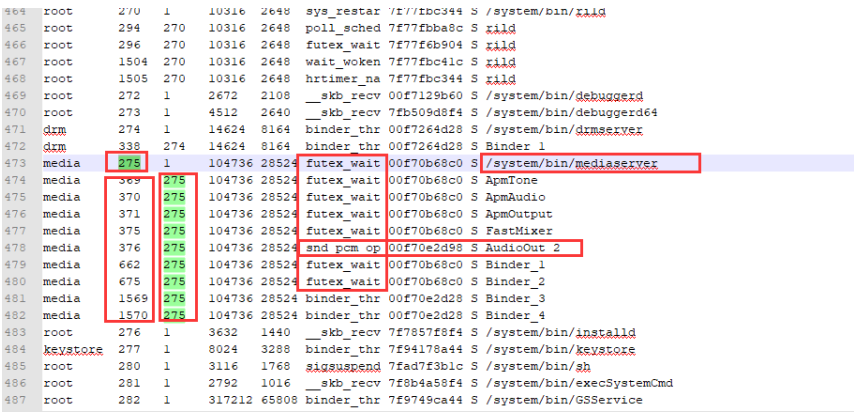
- 然后浏览 logcat，通过审查每一个阶段流程，确认可能的挂机点．注意的是不同的版本可能有所不同，可以先用正常的机器复现一次后比对。
- 在这个案例中，按 power key 不可开启或关闭屏幕，所以不能说明 kernel 是否正常，也无法判断 WindowManagerPolicy-SurfaceFlinger 是否正常。
- 尝试连接 adb，可正常连接，使用 getevent 指令+power 按键有打印，大致方向上判断为上层挂掉。
- 浏览抓到的 logcat，结合现象分析 logcat，注意看 logcat 中是否有"E/" "F/""FATAL""#00""died""error""low mem""""crash""signal""tombstone""ANR""suspend""timeout""EXCEPTION"

等关键字或其他看起来异常的关键字，尝试快速找到有用的内容，找到最早出现异常的日志，对照测试方法看出现时机、现象是否一致。

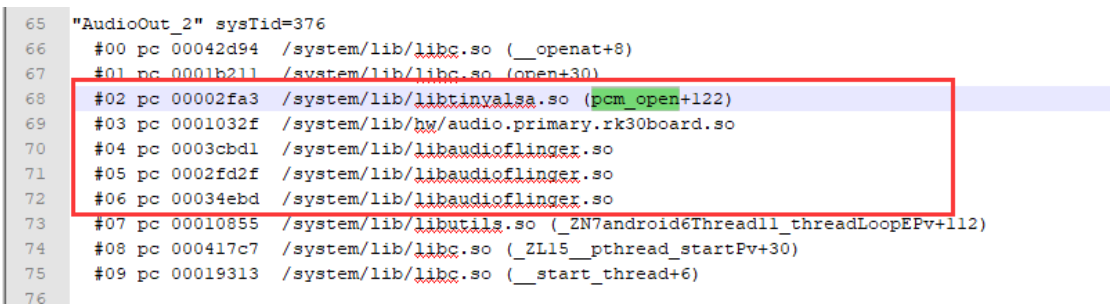
- 这里搜索"died"找到了如下信息，发现大量系统服务 died，确认是上层卡死，此处往上继续查找可能的线索，怀疑代码跑飞、空指针、或者死锁等，继续浏览 log，没有找到代码跑飞、空指针相关内容。



- 使用 ps –t 查看进程&线程状态，发现 mediaserver 进程中疑似死锁。



- **debuggerd -b 275 得到 Trace，看到 hold 在 pcm_open 处。**



- system_server trace 中也发现可疑的地方。

```
"main" prio=5 tid=1 Native
  | group="main" sCount=1 dsCount=0 obj=0x75b57000 self=0x55a07aa9d0
  | sysTid=444 nice=-2 cgrp=default sched=0/0 handle=0x7f8803dfe8
  | state=S schedstat=( 2514995321 229616336 4031 ) utm=200 stm=50 core=1 HZ=100
  | stack=0x7fe9408000-0x7fe940a000 stackSize=8MB
  | held mutexes=
  kernel: __switch_to+0xac/0xb8
  kernel: binder_thread_read+0x3c8/0xf1c
  kernel: binder_ioctl_write_read+0x168/0x2d0
  kernel: binder_ioctl+0x268/0x688
  kernel: do_vfs_ioctl+0x4d4/0x584
  kernel: SyS_ioctl+0x5c/0x8c
  kernel: el0_svc_naked+0x24/0x28
  native: #00 pc 0000000000069a40  /system/lib64/libc.so (__ioctl+4)
  native: #01 pc 0000000000073a64  /system/lib64/libc.so (ioctl+100)
  native: #02 pc 000000000002d584  /system/lib64/libbinder.so (_ZN7android14IPCThreadState14talkWithDriverEb+164)
  native: #03 pc 000000000002e050  /system/lib64/libbinder.so (_ZN7android14IPCThreadState15waitForResponseEPNS_6ParcelEPi+104)
  native: #04 pc 000000000002e2c4  /system/lib64/libbinder.so (_ZN7android14IPCThreadState8transactEijRKNS_6ParcelEPS1_j+176)
  native: #05 pc 0000000000025654  /system/lib64/libbinder.so (_ZN7android8BpBinder8transactEjRKNS_6ParcelEPS1_j+64)
  native: #06 pc 00000000000ad2e4  /system/lib64/libmedia.so (???)
  native: #07 pc 00000000000bf92c  /system/lib64/libmedia.so (_ZN7android11AudioSystem13setParametersERKNS_7String8E+52)
  native: #08 pc 000000000101a9c  /system/lib64/libandroid_runtime.so (???)
  native: #09 pc 00000000034e4e4  /data/dalvik-cache/arm64/system@framework@boot.oat (Java_android_media_AudioSystem_setParameters__Ljava_lang_String_2+152)
  at android.media.AudioSystem.setParameters(Native method)
  at com.android.server.audio.AudioService.setOrientationForAudioSystem(AudioService.java:5339)
  at com.android.server.audio.AudioService.handleConfigurationChanged(AudioService.java:5275)
  at com.android.server.audio.AudioService.-wrap14(AudioService.java:-1)
  at com.android.server.audio.AudioService$AudioServiceBroadcastReceiver.onReceive(AudioService.java:5093)
  at android.app.LoadedApk$ReceiverDispatcher$Args.run(LoadedApk.java:881)
  at android.os.Handler.handleCallback(Handler.java:743)
  at android.os.Handler.dispatchMessage(Handler.java:95)
  at android.os.Looper.loop(Looper.java:148)
  at com.android.server.SystemServer.run(SystemServer.java:294)
  at com.android.server.SystemServer.main(SystemServer.java:173)
  at java.lang.reflect.Method.invoke!(Native method)
  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:772)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:662)
```

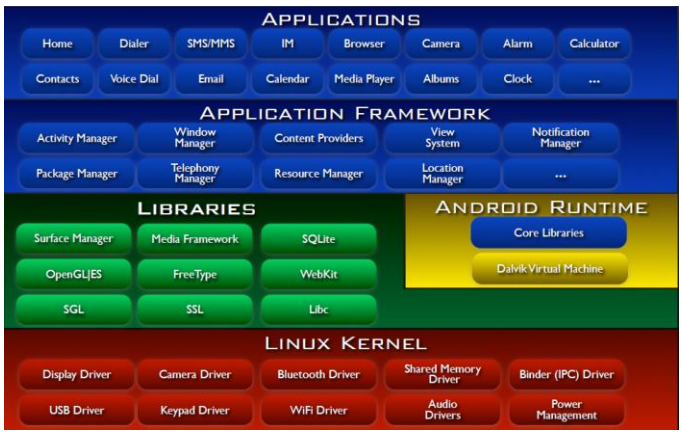
问题解法

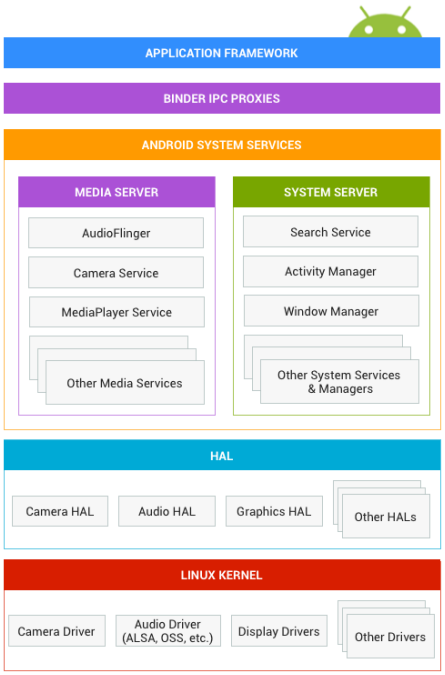
Review 客户代码，发现是客户自行增加的播放音效相关的代码造成的 fd 占用，导致死锁 AudioFlinger 初始化失败，然后系统服务 died，调整播放时序解决。

附录 建立对 Android 系统的大概认知

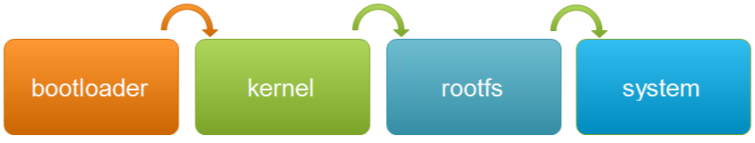
树状的系统：

软件构架图：





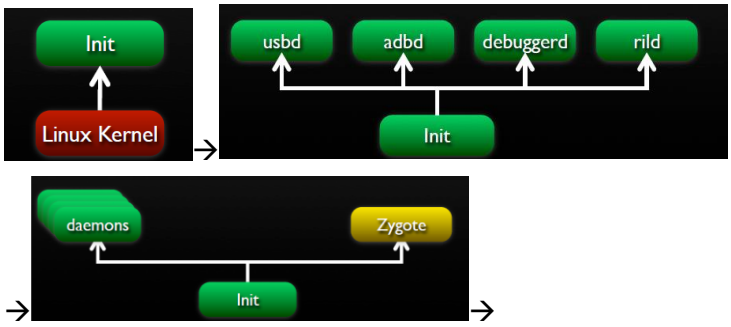
系统整体启动流程概览：

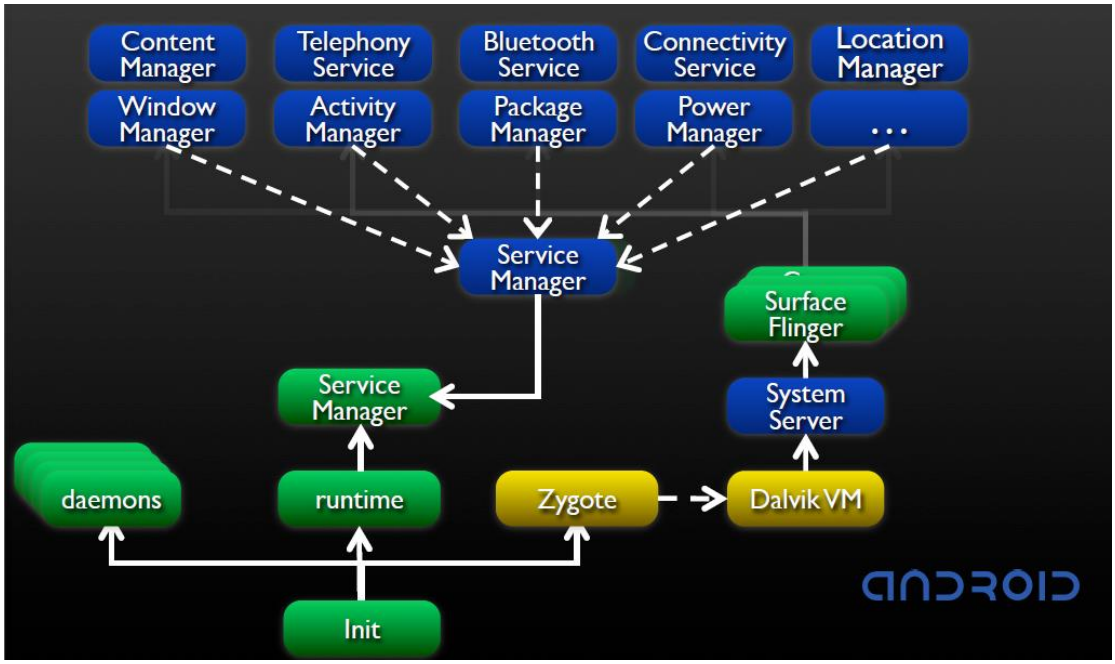


Bootloader 很小，一般在几十 KB 甚至几百 KB，负责做最基本的系统初始化，并把 Kernel 从存储设备（EMMC/NAND）中拷贝到内存（DDR）中，kernel 一般几 MB 到十几 MB、负责控制所有的硬件和系统的调度，根文件系统和 system 属于用户空间的应用，根文件系统一般只有几 MB，负责初始化一个最基本的上层运行环境，为 system 挂载打基

础，system 里面是主要的应用，大小几百 MB 甚至几 GB，主要的应用和库都包含在里面。

ANDROID 上层的启动：



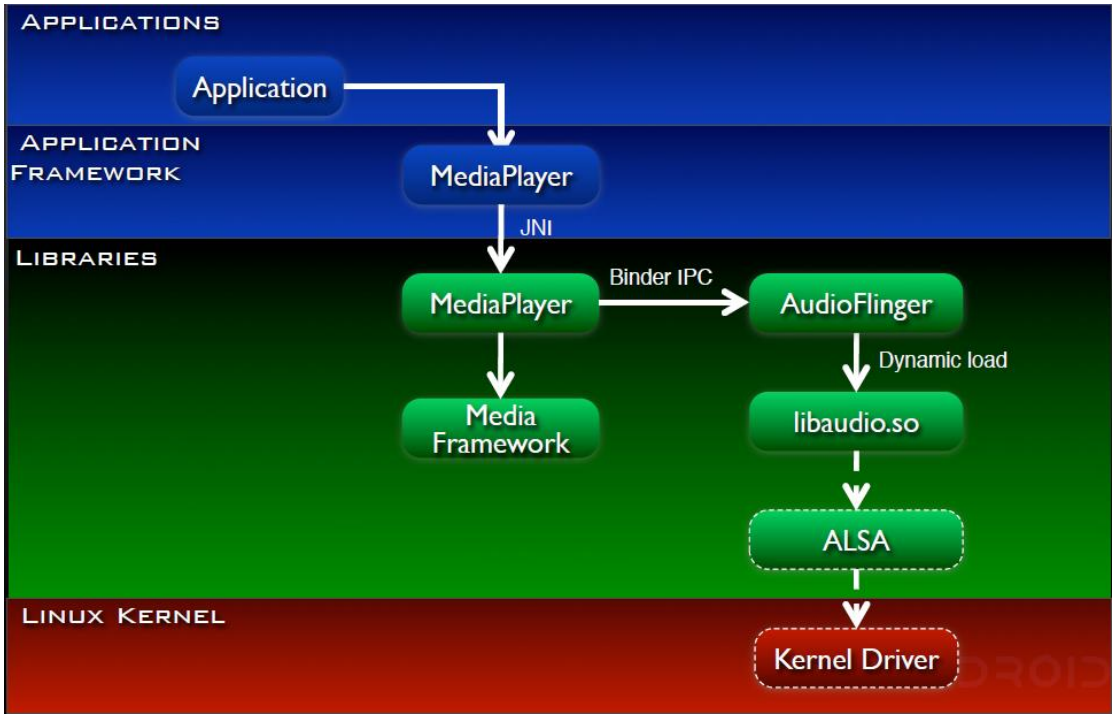


User 空间树状的进程组织：

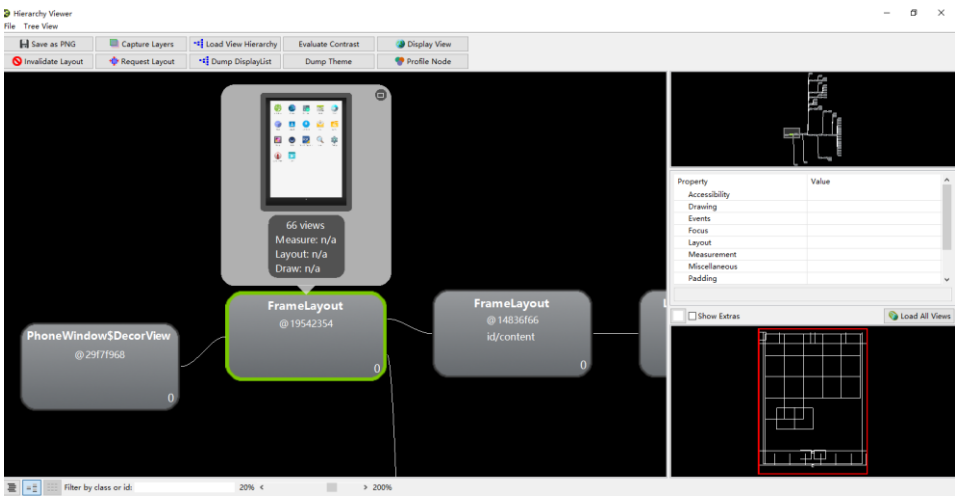
pstree_p.txt

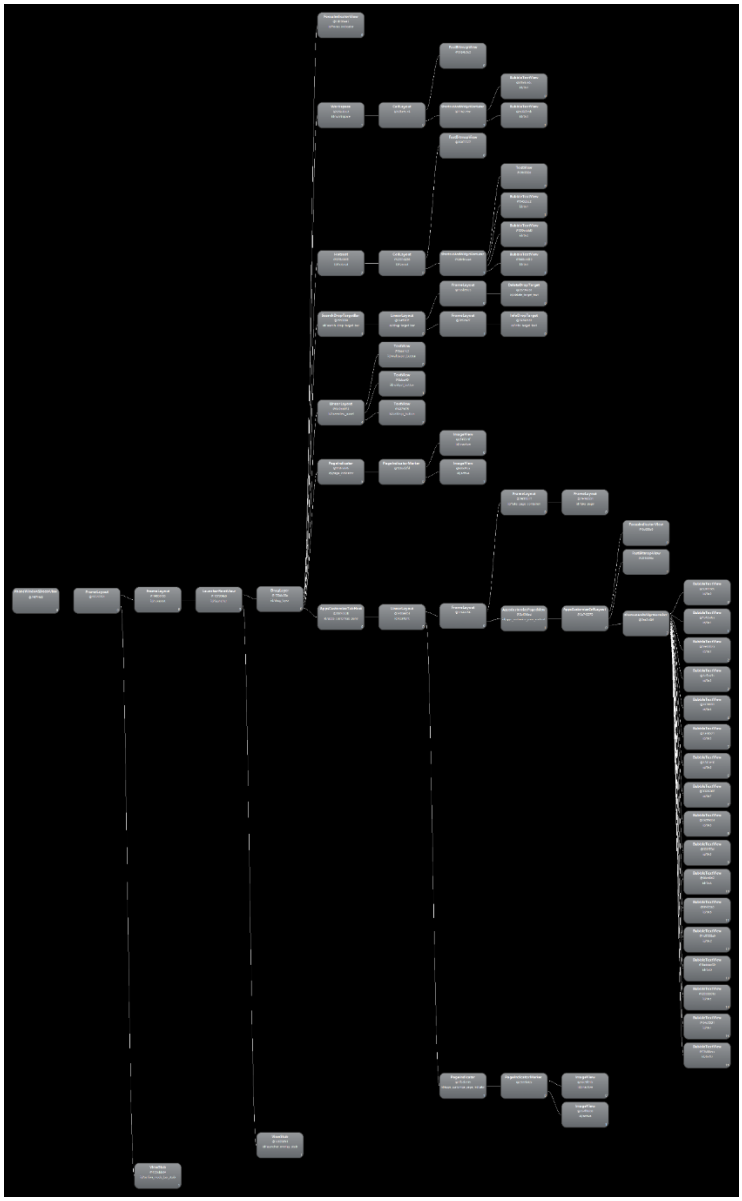

DAEMON PROCESSES
RUNTIME
ZYGOTE
SYSTEM SERVER
HOME
CONTACTS
Activity Manager
Package Manager
Window Manager
...
Dalvik VM
Surface Flinger
Audio Flinger
libc
Home
Dalvik VM
libc
Contacts
Dalvik VM
libc
daemons
libc
runtime
libc
Zygote
libc

模块的软件层级示例：


APPLICATIONS
Application
APPLICATION FRAMEWORK
MediaPlayer
JNI
LIBRARIES
MediaPlayer
Binder IPC
AudioFlinger
Dynamic load
Media Framework
libaudio.so
ALSA
LINUX KERNEL
Kernel Driver

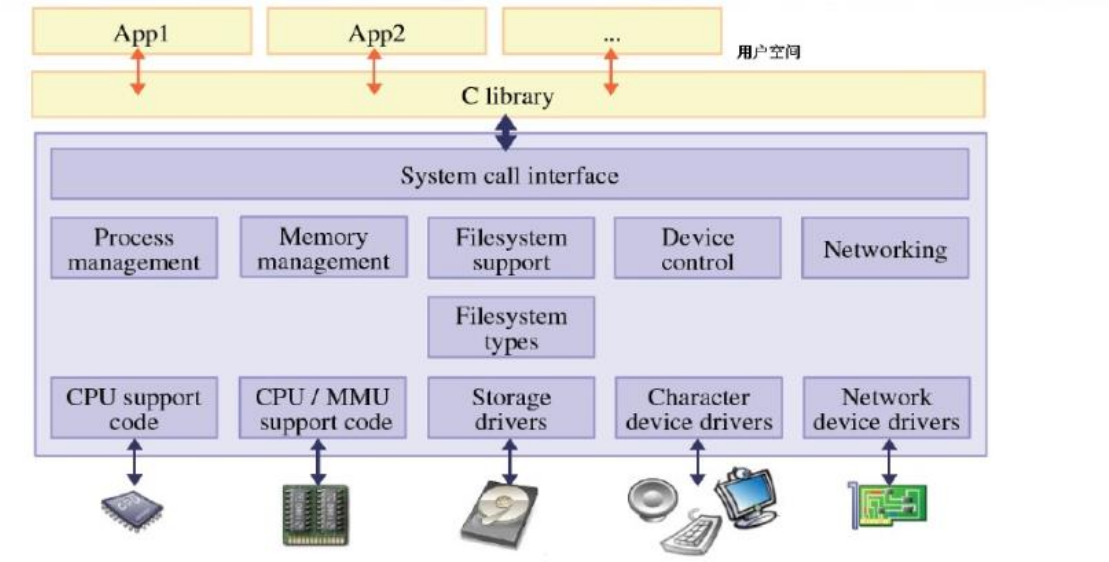
界面：树状的 UI 层级结构

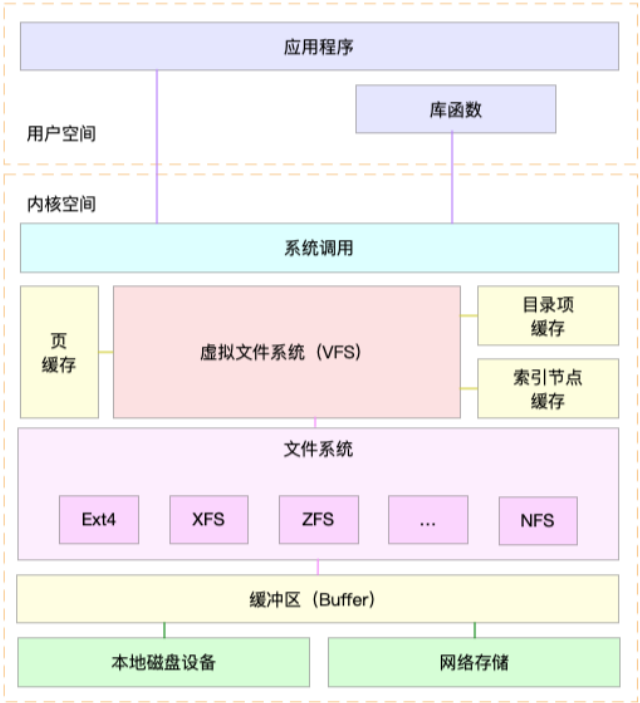




万物皆文件，Linux 树状的文件系统：

du -a-h.log





在 Linux 系统中，一切设备都被抽象为文件，并在/dev/目录下生成对应的设备节点，用户空间的应用要想操作硬件设备，必须通过 open 系统调用打开对应的设备节点，然后通过 read/write/ioctl 这些系统调用再经过该驱动对应的子系统所实现的 file_operations 接口最终和驱动交互。