# Todo

Debuggerd

Kill

调试技巧

Xposed研究下

# 调试技巧

http://gityuan.com/2017/07/11/android\_debug/

## 一. 获取Trace

调用栈信息(Trace)是分析异常经常使用的，这里简单划分两类情况：

* 当前线程Trace: 当前执行流所在线程的调用栈信息；
* 目标进程Trace：可获取目标进程的调用栈，用于动态调试；

#### 1.1 当前线程Trace

**1) Java层**

Thread.currentThread().dumpStack(); //方法1

Log.d(TAG,"Gityuan", new RuntimeException("Gityuan")); //方法2

new RuntimeException("Gityuan").printStackTrace(); //方法3

**2） Native层**

#include <utils/CallStack.h>

android::CallStack stack(("Gityuan"));

#### 1.2 目标进程Trace

**1) Java层**

adb shell kill -3 [pid] //方法1

Process.sendSignal(pid, Process.SIGNAL\_QUIT) //方法2

生成trace文件保存在文件data/anr/traces.txt

adb shell top | findstr system\_server

577 system 18 -2 0% S 105 2662836K 223376K fg system\_server

adb shell kill -3 526

adb pull data/anr/traces.txt

**2） Native层**

adb shell debuggerd -b [tid] //方法1

Debug.dumpNativeBacktraceToFile(pid, tracesPath) //方法2

前两条命令输出内容相同:

* 命令1输出到控制台
* 命令2输出到目标文件

对于debuggerd命令，若不带参数则输出tombstones文件，保存到目录/data/tombstones

**3） Kernel层**

adb shell cat /proc/[tid]/stack //方法1

WatchDog.dumpKernelStackTraces() //方法2

其中dumpKernelStackTraces()只能用于打印当前进程的kernel线程

#### 1.3 小节

以下分别列举输出Java, Native, Kernel的调用栈方式：

| **类别** | **函数式** | **命令式** |
| --- | --- | --- |
| Java | Process.sendSignal(pid, Process.SIGNAL\_QUIT) | kill -3 [pid] |
| Native | Debug.dumpNativeBacktraceToFile(pid, tracesPath) | debuggerd -b [pid] |
| Kernel | WD.dumpKernelStackTraces() | cat /proc/[tid]/stack |

分析异常时往往需要关注的重要目录：

/data/anr/traces.txt

/data/tombstones/tombstone\_X

/data/system/dropbox/

## 二. 时间调试

为了定位耗时过程，有时需要在关注点添加相应的systrace，而systrace可跟踪系统cpu,io以及各个子系统运行状态等信息，对于kernel是利用Linux的ftrace功能。当然也可以直接在方法前后加时间戳，输出log的方式来分析。

#### 2.1 新增systrace

**1) App**

import android.os.Trace;

void foo() {

Trace.beginSection("app:foo");

...

Trace.endSection();

}

**2) Java Framework**

import android.os.Trace;

void foo() {

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "fw:foo");

...

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

}

**3） Native Framework**

#define ATRACE\_TAG ATRACE\_TAG\_GITYUAN

#include <utils/Trace.h> // used for C++

#include <cutils/trace.h> // used for C

void foo() {

ATRACE\_CALL();

...

}

或者

#define ATRACE\_TAG ATRACE\_TAG\_GITYUAN

#include <utils/Trace.h> // used for C++

#include <cutils/trace.h> // used for C

void foo() {

ATRACE\_BEGIN();

...

ATRACE\_END();

}

#### 2.2 打印时间戳

**1) Java**

import android.util.Log;

void foo(){

long startTime = System.currentTimeMillis();

...

long spendTime = System.currentTimeMillis() - startTime;

Log.i(TAG,"took " + spendTime + “ ms.”);

}

**2） C/C++**

#include <stdio.h>

#include <sys/time.h>

void foo() {

struct timeval time;

gettimeofday(&time, NULL); //精度us

printf("took %lld ms.\n", time.tv\_sec \* 1000 + time.tv\_usec /1000);

}

#### 2.3 kernel log

有时候Kernel log的输出是由级别限制，可通过如下命令查看：

adb shell cat /proc/sys/kernel/printk

4 4 1 7

参数解读：

* 控制台日志级别：优先级高于该值的消息将被打印至控制台。
* 缺省的消息日志级别：将用该值来打印没有优先级的消息。
* 最低的控制台日志级别：控制台日志级别可能被设置的最小值。
* 缺省的控制台日志级别：控制台日志级别的缺省值

日志级别：

| **级别** | **值** | **说明** |
| --- | --- | --- |
| KERN\_EMERG | 0 | 致命错误 |
| KERN\_ALERT | 1 | 报告消息 |
| KERN\_CRIT | 2 | 严重异常 |
| KERN\_ERR | 3 | 出错 |
| KERN\_WARNING | 4 | 警告 |
| KERN\_NOTICE | 5 | 通知 |
| KERN\_INFO | 6 | 常规 |
| KERN\_DEBUG | 7 | 调试 |

Log相关命令

* dmesg 或 cat /proc/kmsg
* logcat -L 或 cat /proc/last\_kmsg
* logcat -b events -b system

## 三. addr2line

addr2line功能是将函数地址解析为函数名。分析过Native Crash，那么对addr2line一定不会陌生。 addr2line命令参数：

Usage: addr2line [option(s)] [addr(s)]

The options are:

@<file> Read options from <file>

-a --addresses Show addresses

-b --target=<bfdname> Set the binary file format

-e --exe=<executable> Set the input file name (default is a.out)

-i --inlines Unwind inlined functions

-j --section=<name> Read section-relative offsets instead of addresses

-p --pretty-print Make the output easier to read for humans

-s --basenames Strip directory names

-f --functions Show function names

-C --demangle[=style] Demangle function names

-h --help Display this information

-v --version Display the program's version

#### 3.1 Native地址转换

**Step 1: 获取symbols表**

先获取对应版本的symbols，即可找到对应的so库。（最好是对应版本addr2line，可确保完全匹配）

一般在out/taeget/product/XXX/symbols/system/lib64

@190703测试成功

**Step 2: 执行addr2line命令**

// 64位

cd prebuilts/gcc/linux-x86/aarch64/aarch64-linux-android-4.9/bin

./aarch64-linux-android-addr2line -f -C -e libxxx.so <addr1>

//32位

cd /prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.9/bin

./arm-linux-androideabi-addr2line -f -C -e libxxx.so <addr1>

另外，有兴趣可以研究下development/scripts/stack，地址批量转换工具。

#### 3.2 kernel地址转换

addr2line也适用于调试分析Linux Kernel的问题。例如，查询如下命令所对应的代码行号

[<0000000000000000>] binder\_thread\_read+0x2a0/0x324

**Step 1: 获取符号地址**

通过命令arm-eabi-nm从vmlinux找到目标方法的符号地址，其中nm和vmlinux所在目录：

* arm-eabi-nm位于目录prebuilts/gcc/linux-x86/arm/arm-eabi-4.8/bin/
* vmlinux位于目录out/target/product/xxx/obj/KERNEL\_OBJ/

执行如下命令：(需要带上绝对目录)

arm-eabi-nm vmlinux |grep binder\_thread\_read

则输出结果： c02b2f28 T binder\_thread\_read，可知binder\_thread\_read的符号地址为c02b2f28， 其偏移量为0x2a0，则计算后的目标符号地址= c02b2f28 + 2a0，然后再采用addr2line转换得到方法所对应的行数

**Step 2: 执行addr2line命令**

./aarch64-linux-android-addr2line -f -C -e vmlinux [目标地址]

注意:对于kernel调用栈翻译过程都是通过vmlinux来获取的

# 行业标准

LOG的抓取方式：

拨号盘 里面输入 \*#\*#284#\*#\*，等待约1-2分钟；进入到手机的文件管理器，路径的正确打开方式为：MIUI→debug\_log 文件夹下，有一个包含 bugreport 名字的压缩包，这个就是需要的log。为了保证有效Log不被冲掉，请在遇到Bug时立即抓取Log

<http://www.miui.com/thread-14130379-1-1.html>



# debug 三部曲

系统加大亮屏时间：settings put system screen\_off\_timeout 90000000000

Setprop ro.debuggble 1

## dumpsys activity

adb shell dumpsys activity | findstr mFoc

android看layout tool

## 快速抓取日志

抓起trace

Kill –3 pid

adb pull /data/anr

产生trance文件，打出状态

adb logcat -v time -b main -b system > 1023\_boot.logcat

adb pull /sdcard/rk\_logs

adb pull /data/logs/

adb bugreport > bugreport0926.txt

adb pull /data/anr

adb shell logcat -b main -b system -b radio -b events -v time > logcat1023.log

"main" prio=5 tid=1 Native

at android.view.ViewRootImpl.draw(ViewRootImpl.java:2802)

at android.view.ViewRootImpl.performDraw(ViewRootImpl.java:2610)

debuggerd -b $SurfaceFlinger >> /sdcard/sf.txt

surfaceflinger 的 pid

**4. logcat小技巧**

通过adb bugreport抓取log信息.先看zygote是否起来, 再看system\_server主线程的运行情况,再看ActivityManager情况

adb logcat -s Zygote

adb logcat -s SystemServer

adb logcat -s SystemServiceManager

adb logcat | grep "1359 1359" //system\_server情况

adb logcat -s ActivityManager

现场调试命令

1. cat proc/[pid]/stack ==> 查看kernel调用栈
2. debuggerd -b [pid] ==> 也不可以不带参数-b, 则直接输出到/data/tombstones/目录
3. kill -3 [pid] ==> 生成/data/anr/traces.txt文件
4. lsof [pid] ==> 查看进程所打开的文件

kill

## 截屏

获取当前activity的名字

adb shell mkdir /sdcard/screencap/

adb shell screencap -p /sdcard/screencap/settings\_bt\_pair.png

adb pull /sdcard/screencap ./Desktop

adb shell screencap -p /sdcard/ck.png

adb pull /sdcard/ck.png

## 服务器联调

查询本机ip https://httpbin.org/ip

# 恢复出厂设置

adb shell am broadcast -a android.intent.action.MASTER\_CLEAR

echo --wipe\_all > /cache/recovery/command

reboot recovery

# lint

代码质量检查

# Debug

## 调试方法

### 串口调试

咖啡色接gnd口就好

打开CRT，选择波特率为115200，

串口只能链接一个！不支持多个会话

后台服务的话，加上&符号

logcat –v time &

dmesg &

logcat -v time&

######################################USB-USB-IN

logcat -b main -b system -b radio -b events -v time

## wifi

在客户端设备敲打

adb tcpip 5555

查询到IP

在服务端adb connect 10.129.49.143

<https://blog.csdn.net/wlly1/article/details/54912079>

### ADB

搜索关键字：\* FOR %%i IN (\*.apk)

git，googel，

目标文件：apk

日志加入

#扫描当前apk文件夹

@REM 循环安装本目录下的APK文件

FOR %%i IN (\*.apk) DO (

ECHO 正在安装：%%i

adb install -r %%i

)

#### APK安装

ADB APK安装

adb install

APK 命令可以用adb install [-lrtsd] <file>或者adb install-multiple [-lrtsdp] <file...>，adb install-multiple表示批量安装。  
参数介绍：

* -l：锁定该程序
* -r：可以覆盖已有的应用，保留数据和缓存文件
* -t：允许测试该应用
* -s：安装在SD卡中
* -d：允许降低版本安装
* -p：部分应用程序安装

1. 无需数据线

且可解决部分机器在方法1时出现的“unable to connect to 192.168.1.199:5555”错误

在android设备上安装 “终端模拟器”等类似shell命令工具，使用下面命令（**需root权限**）：

TCP/IP方式：

setprop service.adb.tcp.port 5555

stop adbd

start adbd

usb方式：

setprop service.adb.tcp.port -1

stop adbd

start adbd

<https://twiceyuan.com/2014/11/21/Android%E7%BD%91%E7%BB%9C%E8%B0%83%E8%AF%95%E5%BC%80%E5%90%AF%E6%96%B9%E6%B3%95/>

#### 常见问题

##### 三星手机5.1调试

1. ~~左上角插入手机卡2才有用（待验证）~~

　　1、点开设置—更多—关于设备-》软件信息-》版本号。点内部版本号7到8次左右。

　　2、点过之后返回，就会看到开发者选项，这个勾选USB调试就行啦！

　　3、如果不能安装驱动，在三星官网下载PC套件Samsung kies(包含了链接所需要的驱动)。点击打开。【[PC套件下载地址](http://www.samsung.com/cn/support/usefulsoftware/KIES/)】http://www.samsung.com/cn/support/usefulsoftware/KIES/

4、那个KIES软件安装到电脑上，通过数据线连接电脑（链接模式为媒体设备MTP）即可。

实在连不上可以尝试重启一下

提示：若要解除阻止并通过USB数据线连接至+其他设备，请插入USIM&oq=若要解除阻止并通过USB数据线连接至+其他设备，请插入USIM

##### adb无法启动

现象1：端口被占用，Unable to create Debug Bridge: Unable to start adb server: error: could not install \*smartsocket\* listener: cannot bind to 127.0.0.1:5037

现象2：unknown host service

解决办法：kill 端口占用的进程，常见的是手机助手占用了。

Netstat –ano | findstr 5037



[Android Debugger port : Connection refused:](http://www.jianshu.com/p/96aaed625ad0) adb kill-server; adb start-server

Adb启动ok，无法发现设备

Adb是32位的，设备是64位的，请用android stdio的最新tool的adb来使用就好了。

##### MIX开启 USB安装/USB调试（安全设置）

需要登录小米账号，[more](http://www.miui.com/thread-5711795-1-1.html)

然后才可以执行adb shell。

参考：

[ANDROID开发常用的ADB命令整理kg](http://www.keyguan.com/blog/2017/02/11/android%E5%BC%80%E5%8F%91%E5%B8%B8%E7%94%A8%E7%9A%84adb%E5%91%BD%E4%BB%A4%E6%95%B4%E7%90%86/)

串口adb

Wifi-adb调试

Win-cmd:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| cls | Clear screen |  |  |  |
| Findstr | 正则匹配 |  |  |  |
|  |  |  |  |  |

## Native奔溃调试

## [android native 内存泄露检查（libc.debug.malloc ）](http://blog.csdn.net/haima1998/article/details/51508947)

\*c c++代码，由于其特殊性质，没有虚拟机概念，内存则直接是由用户管理，比如申请，释放，都是需要用户主动去触发，如果用户出现使用了申请，但是用完之后，没有调用释放，则会引起内存泄露。这种叫真正意义的内存泄露，只有重启机子，才能恢复。

相对而已java端的内存泄露，指的是一个应用长期运行，导致相互引用，无法释放，GC没法回收，引起的有效内存越来越小，我们将此现象叫做，内存泄露，通过关闭此应用，重新打开即可恢复内存。因此看来，java内存泄露和c c ++ 的 还是有本质区别的。

java本身的虚拟机里面会关注对象的申请，释放，这些不需要用户直接注，java虚拟机通过管理机制，将调用c c++里面真正的malloc free 方法，封装起来，将java对象的生命周期和malloc free 进行关联，则可以保证在对象不使用的时候，内存紧张时，释放掉不再被引用的对象，GC回收就是在做这件事请。回到我们这节的主要内容，如何定位我们的c c++的内存泄露。

Android对内存的使用包括内存泄漏和内存越界，内存泄漏会导致系统内存减少，最终分配不到内存，这样大的程序就不能运行，甚至系统没有内存而崩溃。Android中kernel和应用程序都可能会有内存泄漏和越界。对于Java代码，在越界的时候虚拟机会加以检查并抛出异常。而对于C/C++代码，越界的时候就悄无声息地让程序出错或crash

### 调试手段

在adb shell之下:确保/system/lib/ libc\_malloc\_debug\_leak.so存在

root@zs600b:/ # stop

root@zs600b:/ # setprop libc.debug.malloc 1

root@zs600b:/ # start

执行成功之后，浏览器和cs-app都无法打开了。。。

指针溢出，野指针，堆内存指针释放多遍等问题如何调试？

Android这边打开bionic/libc/bionic/malloc\_debug\_common.c里的配置

如adb shell setprop persist.libc.debug.malloc 10 再重启

之后可以复现问题，抓bugreport来分析问题，当然要匹配symbols(out/target/product/project/symbols)

此时overhead会比较重，可能会有一些不预期的anr(出现ANR时请点击等待)，但不影响测试。

start & stop are tools under /system/bin (system.img)

### 常见问题

/system/bin/sh: Missing module libc\_malloc\_debug\_leak.so required for malloc debug level 1: dlopen failed: library "libc\_malloc\_debug\_leak.so" not found

userdebug才是自带so，否则自己手动push进去吧。。，。

可以看看eng或这user-debug版中的/system/lib/下是否有这两个文件，其中libc\_malloc\_debug\_qemu.so是模拟器用的。

### 复现场景

堆引起的异常(调用malloc申请的内存后使用不当引起的异常)：

1. 申请后多次释放 (double free)

2. 释放后又去使用 (used after free)

3. 使用越界 (比如申请了50节内存，结果在使用时多用了8字节的内存，这样就把后面的内存的内容踩坏，引起堆结构异常)

4. 释放时传给free()的地址不是malloc()申请的地址，比如：p = malloc(10); free(p + 5);

5. 内存泄露：申请内存后，忘记释放或某些代码路径没有释放

### 源码分析

此malloc的调试原理是：当系统发现我们有libc.debug.malloc的一些列配置成立时，此时系统会将malloc free 等方法，重新指向到 lib\_malloc\_debug.so里面的对应实现方法，lib\_malloc\_debug.so里面的方法，像比较而言，多了一些记录信息，将每次的申请时的地址，堆栈，so等信息记录下来，然后我们需要的时候，则通过工具ddms dump出来，进行分析每个申请的内存，是否正常的释放了，是否出现了内存泄露。

|  |
| --- |
| // The value of libc.debug.malloc.  #if !defined(LIBC\_STATIC)  static int g\_malloc\_debug\_level = 0;  #endif  // Initializes memory allocation framework once per process.  static void malloc\_init\_impl() {  const char\* so\_name = NULL;  // If debug level has not been set by memcheck option in the emulator,  // lets grab it from libc.debug.malloc system property.  if (g\_malloc\_debug\_level == 0 && \_\_system\_property\_get("libc.debug.malloc", env)) {  g\_malloc\_debug\_level = atoi(env);  }  // Choose the appropriate .so for the requested debug level.  switch (g\_malloc\_debug\_level) {  case 1:  case 5:  case 10:  so\_name = "libc\_malloc\_debug\_leak.so";  break;  }  // Load .so that implements the required malloc debugging functionality.  void\* malloc\_impl\_handle = dlopen(so\_name, RTLD\_LAZY);  // No need to init the dispatch table because we can only get  // here if debug level is 1, 5, 10, or 20.  static MallocDebug malloc\_dispatch\_table \_\_attribute\_\_((aligned(32)));  switch (g\_malloc\_debug\_level) {  case 1:  InitMalloc(malloc\_impl\_handle, &malloc\_dispatch\_table, "leak");  break;  }  } |

android系统启动流程

## framework debug 技巧

### 安装

拷贝文件到system/framework/;  
stop;  
start重启system\_server进程

### Java调试

其实整个调试过程非常简单：

* 打断点
* 跟踪代码（Step in/out/over等等）

### 打断点

在正确的进程的合适位置打断点:区别于普通app的进程，framwork的所在进程需要进行源码分析，比如ams是运行在system\_server中的，并且这些进程只能在root的机子调试（模拟器或者root真机）

### 跟踪代码

### 打印日志

-系统一般都会有debug标志，我们修改源码中的标志，打印就好，比如

修改后

-- Log.e("tag",logStr);

### REF

[android native 代码内存泄露 定位方案](http://www.jianshu.com/p/2b43abdd6647)

[[MTK软件原创] 如何调试malloc(堆越界)问题](http://bbs.16rd.com/thread-54815-1-1.html)

[more detail about adb shell start, stop and reboot](http://blog.csdn.net/johnnylq/article/details/6401531)

[android native 内存泄露检查（libc.debug.malloc ）](http://blog.csdn.net/haima1998/article/details/51508947)

# 日志分析logcat

Logcat是存到缓存到内存中的，英文

打印的都是 /dev/log

[Android6.0 Log的工作机制，源码分析](https://zhuanlan.zhihu.com/p/24372024)

adb logcat -c  
清除log缓存

adb bugreport  
查看bug报告

查看event节点，调试activity生命周期

logcat -b events -v time

logcat -b events -v time | grep am\_

查看slog

adb logcat -b system -v time | grep WindowManagerService

logcat -b system -v time | grep WindowManagerService > /sdcard/dlg.txt

cat /proc/kmsg 实时打印内核日志

内核的 dmesg 静态显示内核日志

## 日志抓取方案

Android日志主要分为kernel、radio、event、main这四种log。

dmesg

kernel log属于Linux内核的log ，可以通过读取/proc/kmsg或者通过串口来抓取。

adb 抓取kernel log的命令如下（需要有root权限）：

kernel.log抓取

adb shell rm -rf /sdcard/logs

adb shell mkdir /sdcard/logs

adb reboot

adb root

adb remount

adb shell

#dmesg > /sdcard/logs/kernel.log

cat /proc/kmsg > /sdcard/logs/kernel0324.log

adb shell

logcat -b main -b system -b radio -b events -v time > /sdcard/logs/logcat0324.log

adb shell

. hardware\_monitor.sh > /sdcard/logs/hardware0324.log

cp -rf /data/tombstones /sdcard/logs/

cp -rf /data/anr /sdcard/logs/

exit

adb pull /sdcard/logs

# 日志系统

打印5级堆栈

Debug.getCallers(5));

## TAG中心

ActivityManagerDebugConfig 分类法

类名/别名

子业务名

static final boolean TAG\_WITH\_CLASS\_NAME = false;

static final boolean APPEND\_CATEGORY\_NAME = false;

// Default log tag for the activity manager package.

static final String TAG\_AM = "ActivityManager";

static final String POSTFIX\_BROADCAST = (APPEND\_CATEGORY\_NAME) ? "\_Broadcast" : "";

ActivityManagerService：

private static final String TAG = TAG\_WITH\_CLASS\_NAME ? "ActivityManagerService" : TAG\_AM;

private static final String TAG\_BROADCAST = TAG + POSTFIX\_BROADCAST;

# C++debug

coredump 的基本调试方法



### 案例分析



crash 时 反汇编是： 将寄存器 x21 的数据存储到 x23+48 的一个地址空间，往上推， x21 内容是 0x2,是没有问题的，地址都正常





injectionState 看起来是没有问题的，问题看起来在 lastInjectedEntry 这里了



lastInjectedEntry 是有问题的地址

看了下逻辑，lastInjectedEntry 没有初始化，这里有可能是一个野地址

## 符号表

C++如何debug?

dump tombstones

何时生成？

D:\K\tools\kpad>md kpad

D:\K\tools\kpad>adb pull /data/tombstones

backtrace:

#00 pc 00017216 /system/bin/dji\_sys

#01 pc 00016da5 /system/bin/dji\_sys

#02 pc 00016df3 /system/bin/dji\_sys

#03 pc 00016577 /system/bin/dji\_sys

#04 pc 0001659b /system/lib/libc.so (\_\_pthread\_start(void\*)+30)

#05 pc 000144c3 /system/lib/libc.so (\_\_start\_thread+6)

把找出工具链

gj@linuxe:~/work/android4$ cp prebuilts/gcc/linux-x86/arm/arm-eabi-4.8/bin/arm-eabi-\* ~/bin/

使用命令分析arm-eabi-addr2line

gj@linuxe:~/work/android4$ arm-eabi-addr2line -f -e out/target/product/kpad/symbols/system/bin/dji\_sys 00016577

consistency\_checking\_thread

/home/key/work/android4/opt/dji\_sys/./src/update.cpp:869

gj@linuxe:~/work/android4$ arm-eabi-addr2line -f -e out/target/product/kpad/symbols/system/bin/dji\_sys 00016df3

\_ZN11DJIFwParser20DJIFwParserInitByDirEPc

/home/key/work/android4/opt/dji\_sys/./src/fwupgrade.cpp:261

gj@linuxe:~/work/android4$ arm-eabi-addr2line -f -e out/target/product/gl300k/symbols/system/bin/dji\_sys 00016da5

\_ZN11DJIFwParser19FwUpgrade\_Check\_DirEPcP16T\_MyXmlDjiStruct

/home/key/work/android4/opt/dji\_sys/./src/fwupgrade.cpp:317

gj@linuxe:~/work/android4$ arm-eabi-addr2line -f -e out/target/product/kpad/symbols/system/bin/dji\_sys 00017216

MyXml\_Parse\_Firmware

/home/key/work/android4/opt/dji\_sys/./src/myxml.cpp:298

---------------------------------

aarch64-linux-android-addr2line -f -e out/target/product/gj500/symbols/system/lib64/libinputflinger.so 000000000003356c

## 打印调用堆栈

#include <utils/CallStack.h>

void \_\_egl\_get\_backtrace(void)

{

ALOGD("%s start", \_\_func\_\_);

android::CallStack stack(\_\_func\_\_);

ALOGD("%s end", \_\_func\_\_);

}

在需要的地方，添加这个函数就可以了。记得include这个头文件。

# monkey Test

adb shell monkey --ignore-crashes --ignore-timeouts --ignore-security-exceptions --ignore-native-crashes --kill-process-after-error --bugreport --throttle 100 -v -v -v 100000

adb shell monkey -p com.android.settings --ignore-crashes --ignore-timeouts --ignore-security-exceptions --ignore-native-crashes --kill-process-after-error --bugreport --throttle 100 -v -v -v 100000

<https://www.jianshu.com/p/12842de165cc>

# DEBUG ZGOTE

Zygote

/\*\* enable debugging over JDWP \*/

public static final int DEBUG\_ENABLE\_DEBUGGER = 1;

/\*\* enable JNI checks \*/

public static final int DEBUG\_ENABLE\_CHECKJNI = 1 << 1;

/\*\* enable Java programming language "assert" statements \*/

public static final int DEBUG\_ENABLE\_ASSERT = 1 << 2;

/\*\* disable the AOT compiler and JIT \*/

public static final int DEBUG\_ENABLE\_SAFEMODE = 1 << 3;

/\*\* Enable logging of third-party JNI activity. \*/

public static final int DEBUG\_ENABLE\_JNI\_LOGGING = 1 << 4;

/\*\* Force generation of native debugging information. \*/

public static final int DEBUG\_GENERATE\_DEBUG\_INFO = 1 << 5;

/\*\* Always use JIT-ed code. \*/

public static final int DEBUG\_ALWAYS\_JIT = 1 << 6;

/\*\* Make the code debuggable with turning off some optimizations. \*/

public static final int DEBUG\_NATIVE\_DEBUGGABLE = 1 << 7;

## debug.checkjn

if ("1".equals(SystemProperties.get("debug.checkjni"))) {

debugFlags |= Zygote.DEBUG\_ENABLE\_CHECKJNI;

}

String genDebugInfoProperty = SystemProperties.get("debug.generate-debug-info");

if ("true".equals(genDebugInfoProperty)) {

debugFlags |= Zygote.DEBUG\_GENERATE\_DEBUG\_INFO;

}

if ("1".equals(SystemProperties.get("debug.jni.logging"))) {

debugFlags |= Zygote.DEBUG\_ENABLE\_JNI\_LOGGING;

}

if ("1".equals(SystemProperties.get("debug.assert"))) {

debugFlags |= Zygote.DEBUG\_ENABLE\_ASSERT;

}

Trace和EventLog.writeEvent等系统监控原理

checkTime(startTime, "startProcess: returned from zygote!");

# 卡机日志

01-18 01:45:27.305 W/SystemServer( 577): System clock is before 1970; setting to 1970.

01-18 01:45:27.308 I/SystemServer( 577): Entered the Android system server!