**一、Native Crash是怎么来的？**  
所有的Native Crash都是由于应用进程收到信号引起的。  
当应用收到这个一个信号时，  
1、它的信号处理函数会被触发，  
2、信号处理函数会通过socket将信号的一些信息发送给debuggerd进程。  
3、debuggerd进程通过ptrace attach到应用后，读取应用的寄存器、内存等信息，并用log和tombstone的方式将其输出。  
4、debuggerd进程dettach应用后，应用在kernel中执行信号的默认处理程序，也就是进程coredump、退出和回收。  
  
  
**二、信号处理函数**  
绝大部分的应用程序入口都是linker，linker在启动过程中会调用debuggerd\_init()来注册信号处理函数。  
@bionic/linker/debugger.cpp  
void debuggerd\_init() {  
    struct sigaction action;  
    memset(&action, 0, sizeof(action));  
    sigemptyset(&action.sa\_mask);  
    action.sa\_sigaction = debuggerd\_signal\_handler;  
    action.sa\_flags = SA\_RESTART | SA\_SIGINFO;  
  
    sigaction(SIGABRT, &action, NULL);  
    sigaction(SIGBUS, &action, NULL);  
    sigaction(SIGFPE, &action, NULL);  
    sigaction(SIGILL, &action, NULL);  
    sigaction(SIGPIPE, &action, NULL);  
    sigaction(SIGSEGV, &action, NULL);  
    sigaction(SIGSTKFLT, &action, NULL);  
    sigaction(SIGTRAP, &action, NULL);  
}  
  
可以看出，应用会用注册debuggerd\_signal\_handler()函数来处理SIGABRT、SIGBUS、SIGFPE、SIGILL、SIGPIPE、SIGSEGV、SIGSTKFLT、SIGTRAP等信号。  
  
**三、debuggerd的启动**  
@system/core/debuggerd/debuggerd.c  
int main(int argc, char\*\* argv) {  
    if (argc == 1) {  
        // debuggerd程序不带任何参数时，会以server的形式存在，init.rc中启动debuggerd的方式就是无参启动。  
        return do\_server();  
    }  
    ...  
}  
  
static int do\_server() {  
    int s;  
  
    // 创建socket server  
    s = socket\_local\_server(DEBUGGER\_SOCKET\_NAME,  
            ANDROID\_SOCKET\_NAMESPACE\_ABSTRACT, SOCK\_STREAM);  
    for(;;) {  
        struct sockaddr addr;  
        socklen\_t alen;  
        int fd;  
  
        alen = sizeof(addr);  
        // 监听  
        fd = accept(s, &addr, &alen);  
        fcntl(fd, F\_SETFD, FD\_CLOEXEC);  
  
        // 处理client请求  
        handle\_request(fd);  
    }  
    return 0;  
}  
  
**四、信号的处理流程**  
1、应用收到信号，触发信号处理函数  
@bionic/linker/debugger.cpp  
void debuggerd\_signal\_handler(int n, siginfo\_t\* info, void\*) {  
    ...  
    // 获取当前线程号发送给debuggerd  
    pid\_t tid = gettid();  
  
    // 用socket链接debuggerd  
    int s = socket\_abstract\_client(DEBUGGER\_SOCKET\_NAME, SOCK\_STREAM);          
    if (s >= 0) {  
        debugger\_msg\_t msg;  
        msg.action = DEBUGGER\_ACTION\_CRASH;  
        msg.tid = tid;  
        msg.abort\_msg\_address = reinterpret\_cast<uintptr\_t>(gAbortMessage);  
        // 将信号相关信息发送给debuggerd  
        int ret = TEMP\_FAILURE\_RETRY(write(s, &msg, sizeof(msg)));  
        if (ret == sizeof(msg)) {  
            // 执行read()系统调用，block再kernel态中，等待debuggerd的通知  
            ret = TEMP\_FAILURE\_RETRY(read(s, &tid, 1));  
            ...

2、debuggerd接受请求并attach应用  
@system/core/debuggerd/debuggerd.c  
static void handle\_request(int fd) {  
    debugger\_request\_t request;  
    memset(&request, 0, sizeof(request));  
    // 读取应用发过来的信息  
    int status = read\_request(fd, &request);  
    if (!status) {  
       // 用ptrace attach收到信号的应用的线程，attach的时目标进程会收到SIGSTOP信号  
       if (ptrace(PTRACE\_ATTACH, request.tid, 0, 0)) {  
            LOG("ptrace attach failed: %s\n", strerror(errno));  
        } else {  
            ...  
            // 这里就是回复应用的地方，这时block在read（kernel）里的应用会被唤醒，继续执行完kernel中的read操作。  
            if (TEMP\_FAILURE\_RETRY(write(fd, "\0", 1)) != 1) {              
                ...  
            } else {  
                ...  
                for (;;) {  
                    // 当应用执行完read操作返回到用户空间的时候，由于ptrace的attach给这个应用发了SIGSTOP，所以走到信号处理流程。  
                    // 此时由于应用是被debuggerd attach的状态，所以这个信号会被debuggerd的wait\_for\_signal()截获。  
                    // 原先block在wait\_for\_signal(）的debuggerd得到这个信号后继续往下执行。  
                    int signal = wait\_for\_signal(request.tid, &total\_sleep\_time\_usec);  
                    ...  
                    switch (signal) {  
                    // 这里就是ptrace attach产生的SIGSTOP信号  
                    case SIGSTOP:  
                        if (request.action == DEBUGGER\_ACTION\_DUMP\_TOMBSTONE) {        // 这个是debuggerd客户端发来的请求，如adb shell debuggerd {pid}  
                            ...  
                        } else if (request.action == DEBUGGER\_ACTION\_DUMP\_BACKTRACE) {    // 这个也是debuggerd客户端发来的请求，如adb shell debuggerd -b {pid}  
                            ...  
                        } else {  
                            // 这里是应用crash的case，debuggerd用PTRACE\_CONT命令让应用继续执行，  
                            // 这样应用的read系统调用就可以返回到用户态，继续执行debuggerd\_signal\_handler()  
                            // 此时，debuggerd进入下一次循环，block在wait\_for\_signal，继续等待应用的下一个信号。  
                            status = ptrace(PTRACE\_CONT, request.tid, 0, 0);          
                            continue;   
                        }  
                        break;  
                        ...  
  
3、应用的信号处理函数触发第二次的信号  
@system/core/debuggerd/debuggerd.c  
void debuggerd\_signal\_handler(int n, siginfo\_t\* info, void\*) {  
    ...  
  
    if (s >= 0) {  
        ...  
        // 将信号相关信息发送给debuggerd  
        int ret = TEMP\_FAILURE\_RETRY(write(s, &msg, sizeof(msg)));  
        if (ret == sizeof(msg)) {  
            // 现在read()系统调用处理完毕，继续往下执行  
            ret = TEMP\_FAILURE\_RETRY(read(s, &tid, 1));  
        }  
        ...  
    }  
  
    // 将信号设置成默认处理函数，这样信号就可以在kernel执行信号的默认处理流程  
    signal(n, SIG\_DFL);  
  
    // 因为debuggerd流程还没走完，这里又一次把相同信号发送给自己。因为目前还是被debuggerd attach的状态，所以这个信号还是会被debuggerd截获。  
    // 其中SIGILL、SIGBUS、SIGSEGV情况不需要这里发，因为这三个信号都是CPU发现指令异常后主动发过来的，当信号处理函数执行完后继续执行原先的代码，还会触发相同的信号。  
    // SIGABRT、SIGFPE、SIGPIPE、SIGSTKFLT是非CPU异常，所以这里要再给自己发一个信号  
    switch (n) {  
        case SIGABRT:  
        case SIGFPE:  
        case SIGPIPE:  
        case SIGSTKFLT:  
            (void) tgkill(getpid(), gettid(), n);  
            break;  
        default:    // SIGILL, SIGBUS, SIGSEGV  
            break;  
    }  
}  
  
4、debuggerd处理第二次的信号  
@system/core/debuggerd/debuggerd.c  
static void handle\_request(int fd) {  
                ...  
                for (;;) {  
                    // 这里收到第3步发过来的SIGNAL  
                    int signal = wait\_for\_signal(request.tid, &total\_sleep\_time\_usec);  
                    ...  
                    switch (signal) {  
                    case SIGILL:  
                    case SIGABRT:  
                    case SIGBUS:  
                    case SIGFPE:  
                    case SIGSEGV:  
                    case SIGPIPE:  
                    case SIGSTKFLT:  
                        // 先给进程中的所有线程发送SIGSTOP命令，保证打印调用栈时其他线程都是等待状态  
                        kill(request.pid, SIGSTOP);  
                        // 打印tombstone，log等信息  
                        tombstone\_path = engrave\_tombstone(request.pid, request.tid,  
                                signal, request.abort\_msg\_address, !attach\_gdb, false,  
                                &detach\_failed, &total\_sleep\_time\_usec);  
                        break;  
                    }  
                    ...  
            }  
            // detach应用线程  
            ptrace(PTRACE\_DETACH, request.tid, 0, 0);  
  
            // 让应用继续执行，SIGILL、SIGBUS、SIGSEGV三种情况，如果程序继续执行立马会触发第3次的信号  
            // 这里需要注意，第3步中已经将信号处理函数设置成默认的了，所以下一步会执行kernel中的信号处理流程  
            // SIGABRT、SIGFPE、SIGPIPE、SIGSTKFLT这四种情况，只有代码中再次给这个进程发送相同SIGNAL才会进入kernel的信号处理流程，否则程序继续执行。  
            // 所以一般发送SIGABRT、SIGFPE、SIGPIPE、SIGSTKFLT信号的地方要连着发送两次，比如abort()、dvmNukeThread()等  
            kill(request.pid, SIGCONT);  
        }  
    }  
    ...  
}  
  
@dalvik/vm/Thread.cpp  
void dvmNukeThread(Thread\* thread)  
{  
    // 第一次发送SIGSTKFLT  
    killResult = pthread\_kill(thread->handle, SIGSTKFLT);  
    ...  
    usleep(2 \* 1000 \* 1000);    // TODO: timed-wait until debuggerd attaches  
    // 第二次发送SIGSTKFLT  
    killResult = pthread\_kill(thread->handle, SIGSTKFLT);  
    ...  
    usleep(8 \* 1000 \* 1000);    // TODO: timed-wait until debuggerd finishes  
  
    ...  
}  
  
@bionic/libc/bionic/abort.cpp  
void abort()  
{  
  ...  
  // 第一次发送SIGABRT  
  raise(SIGABRT);  
  
  struct sigaction sa;  
  sa.sa\_handler = SIG\_DFL;  
  sa.sa\_flags   = SA\_RESTART;  
  sigemptyset(&sa.sa\_mask);  
  sigaction(SIGABRT, &sa, &sa);  
  sigprocmask(SIG\_SETMASK, &mask, NULL);  
  // 第二次发送SIGABRT  
  raise(SIGABRT);  
  \_exit(1);  
}  
  
5、信号的默认处理流程  
@kernel/kernel/signal.c  
int get\_signal\_to\_deliver(siginfo\_t \*info, struct k\_sigaction \*return\_ka,  
                          struct pt\_regs \*regs, void \*cookie)  
{  
        struct sighand\_struct \*sighand = current->sighand;  
        struct signal\_struct \*signal = current->signal;  
        int signr;  
  
        for (;;) {  
            struct k\_sigaction \*ka;  
            ...  
            // 获取信号类型  
            signr = dequeue\_signal(current, &current->blocked, info);  
  
            if (!signr)  
                break; /\* will return 0 \*/  
  
            // 获取sigaction  
            ka = &sighand->action[signr-1];  
  
            // 如果是信号被设置成忽略的，则不做任何事情  
            if (ka->sa.sa\_handler == SIG\_IGN) /\* Do nothing.  \*/  
                    continue;  
            if (ka->sa.sa\_handler != SIG\_DFL) {  
                // 如果应用注册了信号处理函数，则执行该信号处理函数  
                \*return\_ka = \*ka;  
                if (ka->sa.sa\_flags & SA\_ONESHOT)  
                    ka->sa.sa\_handler = SIG\_DFL;  
                break; /\* will return non-zero "signr" value \*/  
            }  
  
            // 这里开始是信号的默认处理流程了  
  
            // 如果信号的default action本身是ignore的，则啥都不做  
            if (sig\_kernel\_ignore(signr)) /\* Default is nothing. \*/  
                 continue;  
  
            // 如果是stop类型的，则stop当前线程  
            if (sig\_kernel\_stop(signr)) {  
                ...  
                if (likely(do\_signal\_stop(info->si\_signo))) {  
                    goto relock;  
                }  
                continue;  
            }  
  
            // 如果是coredump类型的，则进行coredump  
            if (sig\_kernel\_coredump(signr)) {  
               if (print\_fatal\_signals)  
                    print\_fatal\_signal(info->si\_signo);  
                proc\_coredump\_connector(current);  
                do\_coredump(info);  
            }  
  
            // 退出进程  
            do\_group\_exit(info->si\_signo);  
        }  
        spin\_unlock\_irq(&sighand->siglock);  
        return signr;  
}  
  
6、信号的default action列表：  
 \*      +--------------------+------------------+  
 \*      |  POSIX signal      |  default action  |  
 \*      +--------------------+------------------+  
 \*      |  SIGHUP            |  terminate       |  
 \*      |  SIGINT            |  terminate       |  
 \*      |  SIGQUIT           |  coredump        |  
 \*      |  SIGILL            |  coredump        |  
 \*      |  SIGTRAP           |  coredump        |  
 \*      |  SIGABRT/SIGIOT    |  coredump        |  
 \*      |  SIGBUS            |  coredump        |  
 \*      |  SIGFPE            |  coredump        |  
 \*      |  SIGKILL           |  terminate(+)    |  
 \*      |  SIGUSR1           |  terminate       |  
 \*      |  SIGSEGV           |  coredump        |  
 \*      |  SIGUSR2           |  terminate       |  
 \*      |  SIGPIPE           |  terminate       |  
 \*      |  SIGALRM           |  terminate       |  
 \*      |  SIGTERM           |  terminate       |  
 \*      |  SIGCHLD           |  ignore          |  
 \*      |  SIGCONT           |  ignore(\*)       |  
 \*      |  SIGSTOP           |  stop(\*)(+)      |  
 \*      |  SIGTSTP           |  stop(\*)         |  
 \*      |  SIGTTIN           |  stop(\*)         |  
 \*      |  SIGTTOU           |  stop(\*)         |  
 \*      |  SIGURG            |  ignore          |  
 \*      |  SIGXCPU           |  coredump        |  
 \*      |  SIGXFSZ           |  coredump        |  
 \*      |  SIGVTALRM         |  terminate       |  
 \*      |  SIGPROF           |  terminate       |  
 \*      |  SIGPOLL/SIGIO     |  terminate       |  
 \*      |  SIGSYS/SIGUNUSED  |  coredump        |  
 \*      |  SIGSTKFLT         |  terminate       |  
 \*      |  SIGWINCH          |  ignore          |  
 \*      |  SIGPWR            |  terminate       |  
 \*      |  SIGRTMIN-SIGRTMAX |  terminate       |  
 \*      +--------------------+------------------+  
 \*      |  non-POSIX signal  |  default action  |  
 \*      +--------------------+------------------+  
 \*      |  SIGEMT            |  coredump        |  
 \*      +--------------------+------------------+  
  
可以看到除了SIGPIPE和SIGSTKFLTNativeCrash相关信号的默认default action都是coredump，  
因此只要配置好coredump开关，所有的native crash都会引起coredump。  
  
  
**五、信号的来源**  
正如上面内容所述，信号的来源有两类：一类是CPU异常处理程序触发的、另一类是代码中显示地发出来的。  
1、SIGABRT  
这个信号一般用于代码中某个判断条件在正常流程始终为真，但作者又担心实际运行中会出现条件为假的情况。  
为了方便调试，作者会在这种情况下，加入一个abort语句，给自己发一个SIGABRT后触发coredump。  
  
例如：  
pid: 2226， tid: 2297， name: launcher-loader >>> com.miui.home <<<  
signal 6 (SIGABRT)， code -6 (SI\_TKILL)， fault addr --------  
backtrace:  
#00 pc 00036efc /system/lib/libc.so (tgkill+12)  
#01 pc 00013fb9 /system/lib/libc.so (pthread\_kill+52)  
#02 pc 00014bd7 /system/lib/libc.so (raise+10)  
#03 pc 00011519 /system/lib/libc.so (\_\_libc\_android\_abort+36)  
#04 pc 0000fc9c /system/lib/libc.so (abort+4)  
#05 pc 00214ded /system/lib/libart.so (art::Runtime::Abort()+160)  
#06 pc 000a63d3 /system/lib/libart.so (art::LogMessage::LogMessage()+1322)  
#07 pc 002223ab /system/lib/libart.so (art::Thread::AssertNoPendingExceptionForNewException(char const\*) const+386)  
  
signal 6 (SIGABRT), code -6 (SI\_TKILL), fault addr --------  
backtrace:  
#00 pc 0002218c /system/lib/libc.so (tgkill+12)  
#01 pc 000131e1 /system/lib/libc.so (pthread\_kill+48)  
#02 pc 000133f5 /system/lib/libc.so (raise+10)  
#03 pc 0001212b /system/lib/libc.so  
#04 pc 00021a40 /system/lib/libc.so (abort+4)  
#05 pc 00048c9f /system/lib/libdvm.so (dvmAbort+78)  
#06 pc 0002a7c8 /system/lib/libdvm.so (IndirectRefTable::get(void\*) const+116)  
  
signal 6 (SIGABRT)? code -6 (SI\_TKILL)? fault addr --------  
backtrace:  
#00 pc 000220fc /system/lib/libc.so (tgkill+12)  
#01 pc 00013151 /system/lib/libc.so (pthread\_kill+48)  
#02 pc 00013365 /system/lib/libc.so (raise+10)  
#03 pc 0001209b /system/lib/libc.so  
#04 pc 000219b0 /system/lib/libc.so (abort+4)  
#05 pc 0020d755 /system/lib/libmiui\_chromium.so  
#06 pc 0021f223 /system/lib/libmiui\_chromium.so  
#07 pc 00208cdd /system/lib/libmiui\_chromium.so  
  
signal 6 (SIGABRT), code -6 (SI\_TKILL), fault addr --------  
Abort message: 'heap corruption detected by dlfree'  
backtrace:  
#00 pc 0002210c /system/lib/libc.so (tgkill+12)  
#01 pc 00013165 /system/lib/libc.so (pthread\_kill+48)  
#02 pc 00013379 /system/lib/libc.so (raise+10)  
#03 pc 000120a3 /system/lib/libc.so  
#04 pc 000219c0 /system/lib/libc.so (abort+4)  
#05 pc 00012b89 /system/lib/libc.so  
#06 pc 0000f295 /system/lib/libc.so  
#07 pc 00011863 /system/lib/libc.so (dlfree+1222)  
#08 pc 0000dce3 /system/lib/libc.so (free+10)  
#09 pc 0006955f /system/lib/libicuuc.so (icu\_51::UnicodeString::~UnicodeString()+38)  
#10 pc 000b9879 /system/lib/libicui18n.so (icu\_51::RegexPattern::zap()+128)  
  
这种情况先要根据调用栈找到调用abort函数的语句及判断条件，也就是出问题的内存地址。  
然后再用gdb做内存分析，看看是模块本身的问题还是模块的地址被别人踩坏导致的。  
  
2、SIGSTKFLT  
这个信号是从dvm模块中发出来的。  
dvm触发gc或则signalCatcher收到signal 3打印调用栈的时候，需要suspend当前进程里的所有线程。  
而线程在某些情况下是无法suspend的，如果dvm发现某个线程在一段时间（7~8秒）内无法被suspend，  
它就会打印当前进程里的所有线程的调用栈，并且调用dvmNukeThread()，  
dvmNukeThread()又会调用pthread\_kill()给自己发送SIGSTKFLT信号来完成自杀。  
这种问题，一般跟native没关系，可以通过log中打印出的java调用栈来排查问题。  
如：   
10-27 15:45:30.658 I/DEBUG   (30514): pid: 30528, tid: 30528, name: com.mi.vtalk  >>> com.mi.vtalk <<<   
10-27 15:45:30.658 I/DEBUG   (30514): signal 16 (SIGSTKFLT), code -6 (SI\_TKILL), fault addr --------   
  
此时的native调用栈基本上没啥用，得看之前的dvm log。  
比如：  
10-27 15:45:22.498 I/Process ( 1188): Sending signal. PID: 30528 SIG: 3   
10-27 15:45:22.498 I/dalvikvm(30528): threadid=3: reacting to signal 3  
10-27 15:45:23.498 W/dalvikvm(30528): threadid=3: spin on suspend #1 threadid=1 (pcf=0)  
10-27 15:45:24.248 W/dalvikvm(30528): threadid=3: spin on suspend #2 threadid=1 (pcf=0)   
10-27 15:45:24.248 I/dalvikvm(30528): "Signal Catcher" daemon prio=5 tid=3 RUNNABLE   
10-27 15:45:24.248 I/dalvikvm(30528):   | group="system" sCount=0 dsCount=0 obj=0x43196180 self=0x41b8ca78   
10-27 15:45:24.248 I/dalvikvm(30528):   | sysTid=30533 nice=0 sched=0/0 cgrp=apps handle=1102628912   
10-27 15:45:24.248 I/dalvikvm(30528):   | state=R schedstat=( 2986246 191254114 53 ) utm=0 stm=0 core=3   
10-27 15:45:24.248 I/dalvikvm(30528):   at dalvik.system.NativeStart.run(Native Method)   
10-27 15:45:24.248 I/dalvikvm(30528):    
10-27 15:45:24.248 I/dalvikvm(30528): "main" prio=5 tid=1 RUNNABLE   
10-27 15:45:24.248 I/dalvikvm(30528):   | group="main" sCount=1 dsCount=0 obj=0x41ba5e58 self=0x41b94890   
10-27 15:45:24.248 I/dalvikvm(30528):   | sysTid=30528 nice=0 sched=0/0 cgrp=apps handle=1073942868   
10-27 15:45:24.248 I/dalvikvm(30528):   | state=S schedstat=( 15003200422 7197818685 42204 ) utm=1322 stm=178 core=0   
10-27 15:45:24.248 I/dalvikvm(30528):   at android.os.MessageQueue.nativePollOnce(Native Method)   
10-27 15:45:24.248 I/dalvikvm(30528):   at android.os.MessageQueue.next(MessageQueue.[java:138](http://java:138/))   
10-27 15:45:24.248 I/dalvikvm(30528):   at android.os.Looper.loop(Looper.[java:123](http://java:123/))   
10-27 15:45:24.248 I/dalvikvm(30528):   at android.app.ActivityThread.main(ActivityThread.[java:5056](http://java:5056/))   
10-27 15:45:24.248 I/dalvikvm(30528):   at java.lang.reflect.Method.invokeNative(Native Method)   
10-27 15:45:24.248 I/dalvikvm(30528):   at java.lang.reflect.Method.invoke(Method.[java:515](http://java:515/))   
10-27 15:45:24.248 I/dalvikvm(30528):   at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.[java:909](http://java:909/))   
10-27 15:45:24.248 I/dalvikvm(30528):   at com.android.internal.os.ZygoteInit.main(ZygoteInit.[java:725](http://java:725/))   
10-27 15:45:24.248 I/dalvikvm(30528):   at dalvik.system.NativeStart.main(Native Method)   
10-27 15:45:24.248 I/dalvikvm(30528):   
10-27 15:45:24.998 W/dalvikvm(30528): threadid=3: spin on suspend #3 threadid=1 (pcf=0)   
10-27 15:45:24.998 I/dalvikvm(30528): "Signal Catcher" daemon prio=5 tid=3 RUNNABLE   
10-27 15:45:24.998 I/dalvikvm(30528):   | group="system" sCount=0 dsCount=0 obj=0x43196180 self=0x41b8ca78   
10-27 15:45:25.008 I/dalvikvm(30528):   | sysTid=30533 nice=0 sched=0/0 cgrp=apps handle=1102628912   
10-27 15:45:25.008 I/dalvikvm(30528):   | state=R schedstat=( 5843847 196806200 85 ) utm=0 stm=0 core=3   
10-27 15:45:25.008 I/dalvikvm(30528):   at dalvik.system.NativeStart.run(Native Method)   
10-27 15:45:25.008 I/dalvikvm(30528):    
10-27 15:45:25.008 I/dalvikvm(30528): "main" prio=5 tid=1 RUNNABLE   
10-27 15:45:25.008 I/dalvikvm(30528):   | group="main" sCount=1 dsCount=0 obj=0x41ba5e58 self=0x41b94890   
10-27 15:45:25.008 I/dalvikvm(30528):   | sysTid=30528 nice=0 sched=0/0 cgrp=apps handle=1073942868   
10-27 15:45:25.008 I/dalvikvm(30528):   | state=S schedstat=( 15003200422 7197818685 42204 ) utm=1322 stm=178 core=0   
10-27 15:45:25.008 I/dalvikvm(30528):   at android.os.MessageQueue.nativePollOnce(Native Method)   
10-27 15:45:25.008 I/dalvikvm(30528):   at android.os.MessageQueue.next(MessageQueue.[java:138](http://java:138/))   
10-27 15:45:25.008 I/dalvikvm(30528):   at android.os.Looper.loop(Looper.[java:123](http://java:123/))   
10-27 15:45:25.008 I/dalvikvm(30528):   at android.app.ActivityThread.main(ActivityThread.[java:5056](http://java:5056/))   
10-27 15:45:25.008 I/dalvikvm(30528):   at java.lang.reflect.Method.invokeNative(Native Method)   
10-27 15:45:25.008 I/dalvikvm(30528):   at java.lang.reflect.Method.invoke(Method.[java:515](http://java:515/))   
10-27 15:45:25.008 I/dalvikvm(30528):   at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.[java:909](http://java:909/))   
10-27 15:45:25.008 I/dalvikvm(30528):   at com.android.internal.os.ZygoteInit.main(ZygoteInit.[java:725](http://java:725/))   
10-27 15:45:25.008 I/dalvikvm(30528):   at dalvik.system.NativeStart.main(Native Method)  
...  
10-27 15:45:30.318 W/dalvikvm(30528): threadid=3: spin on suspend #10 threadid=1 (pcf=0)   
10-27 15:45:30.318 I/dalvikvm(30528): "Signal Catcher" daemon prio=5 tid=3 RUNNABLE   
10-27 15:45:30.318 I/dalvikvm(30528):   | group="system" sCount=0 dsCount=0 obj=0x43196180 self=0x41b8ca78   
10-27 15:45:30.318 I/dalvikvm(30528):   | sysTid=30533 nice=0 sched=0/0 cgrp=apps handle=1102628912   
10-27 15:45:30.318 I/dalvikvm(30528):   | state=R schedstat=( 29816814 248098492 282 ) utm=1 stm=2 core=3   
10-27 15:45:30.318 I/dalvikvm(30528):   at dalvik.system.NativeStart.run(Native Method)   
10-27 15:45:30.318 I/dalvikvm(30528):    
10-27 15:45:30.318 I/dalvikvm(30528): "main" prio=5 tid=1 RUNNABLE   
10-27 15:45:30.318 I/dalvikvm(30528):   | group="main" sCount=1 dsCount=0 obj=0x41ba5e58 self=0x41b94890   
10-27 15:45:30.318 I/dalvikvm(30528):   | sysTid=30528 nice=0 sched=0/0 cgrp=apps handle=1073942868   
10-27 15:45:30.318 I/dalvikvm(30528):   | state=S schedstat=( 15003200422 7197818685 42204 ) utm=1322 stm=178 core=0   
10-27 15:45:30.318 I/dalvikvm(30528):   at android.os.MessageQueue.nativePollOnce(Native Method)   
10-27 15:45:30.318 I/dalvikvm(30528):   at android.os.MessageQueue.next(MessageQueue.[java:138](http://java:138/))   
10-27 15:45:30.328 I/dalvikvm(30528):   at android.os.Looper.loop(Looper.[java:123](http://java:123/))   
10-27 15:45:30.328 I/dalvikvm(30528):   at android.app.ActivityThread.main(ActivityThread.[java:5056](http://java:5056/))   
10-27 15:45:30.328 I/dalvikvm(30528):   at java.lang.reflect.Method.invokeNative(Native Method)   
10-27 15:45:30.328 I/dalvikvm(30528):   at java.lang.reflect.Method.invoke(Method.[java:515](http://java:515/))   
10-27 15:45:30.328 I/dalvikvm(30528):   at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.[java:909](http://java:909/))   
10-27 15:45:30.328 I/dalvikvm(30528):   at com.android.internal.os.ZygoteInit.main(ZygoteInit.[java:725](http://java:725/))   
10-27 15:45:30.328 I/dalvikvm(30528):   at dalvik.system.NativeStart.main(Native Method)   
10-27 15:45:30.328 I/dalvikvm(30528):    
10-27 15:45:30.328 E/dalvikvm(30528): Fatal spin-on-suspend, dumping threads   
10-27 15:45:30.328 I/dalvikvm(30528): DALVIK THREADS:  
...  
  
这里的关键log是：  
threadid=3: spin on suspend #1 threadid=1 (pcf=0)  
其中  
threadid=3表示当前发去suspend操作的线程的id是3，也就是Signal Catcher线程  
spin on suspend #1中数字1表示尝试suspend的次数，一般会尝试10次，第一次间隔0.25秒，之后每次间隔0.75秒。尝试10次后都失败了，就直接dump所有线程的栈，然后发出SIGSTKFLT信号给自己。  
threadid=1表示要挂起的目标线程，这里是主线程。  
pcf=0表示优先级的状态，当尝试两次suspend后会尝试提高目标线程的优先级。  
相关代码为：waitForThreadSuspend()@dalvik/vm/Thread.cpp  
  
就这个问题来说，主线程是RUNNABLE状态，但一直得不到调度无法设置suspend flag，所以Signal Catcher一直在等。  
所以这个错误可能跟这个应用本身没多大关系，可能是系统非常繁忙导致的。  
从log来看，相同时间点确实存在由于性能引起的ANR，而ANR本身又会收集很多调试信息，会加剧系统的繁忙程度。  
  
  
3、SIGFPE  
由于arm没有除法指令，所以gcc会提供内置的函数来实现除法功能，定义了除法函数的gcc.a已静态链接的方式链接到每一个so和bin中。  
当除法函数检测到除数为0时，会调用\_\_aeabi\_idiv0()函数，\_\_aeabi\_idiv0()函数又会调用raise(SIGFPE)，将SIGFPE信号发送给自己。  
典型的调用栈如下：  
pid: 8406, tid: 8480, name: Thread-479  >>> com.qiyi.video <<<  
signal 8 (SIGFPE), code -6 (SI\_TKILL), fault addr 000020d6  
    r0 00000000  r1 00002120  r2 00000008  r3 00000000  
    r4 00000008  r5 00000000  r6 00002120  r7 0000010c  
    r8 60bdde21  r9 61bab000  sl 5f9915fc  fp 400eb2fc  
    ip 00000001  sp 61ca87b0  lr 400b10bd  pc 400bfff8  cpsr 000f0010  
backtrace:  
    #00  pc 00021ff8  /system/lib/libc.so (tgkill+12)  
    #01  pc 000130b9  /system/lib/libc.so (pthread\_kill+48)  
    #02  pc 000132cd  /system/lib/libc.so (raise+10)  
    #03  pc 001e706c  /data/app-lib/com.qiyi.video-1/libkernel.so (\_\_aeabi\_ldiv0+8)  
    #04  pc 0014cf03  /data/app-lib/com.qiyi.video-1/libkernel.so (ServerManager2::GetPeer(framework::ID const&, unsigned int)+126)  
  
这种问题，只需要看调用栈中\_\_aeabi\_ldiv0下一个行，可以用addr2line等工具定位源代码。  
  
  
4、SIGPIPE  
这种信号一般是读端已经关闭，写端还在写的时候会触发。通常是pipe或则socket链接会有这种问题。  
典型的调用栈如下：  
pid: 25534, tid: 25534, name: ping >>> ping <<<  
signal 13 (SIGPIPE), code -6 (SI\_TKILL), fault addr --------  
backtrace:  
#00 pc 00020464 /system/lib/libc.so (write+12)  
#01 pc 00025745 /system/lib/libc.so (\_\_sflush+54)  
#02 pc 000257d5 /system/lib/libc.so (fflush+60)  
#03 pc 000041ef /system/bin/ping  
#04 pc 000048e5 /system/bin/ping  
#05 pc 00002725 /system/bin/ping  
#06 pc 0000e4c3 /system/lib/libc.so (\_\_libc\_init+50)  
#07 pc 00001348 /system/bin/ping  
  
此时的调用栈顶端一般都是系统调用write()函数。系统调用的kernel实现流程，检测到读端被关闭的状态后发送SIGPIPE给自己。  
@kernel/fs/pipe.c  
static ssize\_t  
pipe\_write(struct kiocb \*iocb, const struct iovec \*\_iov,  
            unsigned long nr\_segs, loff\_t ppos)  
{  
        ...  
        if (!pipe->readers) {  
                send\_sig(SIGPIPE, current, 0);  
                ret = -EPIPE;  
                goto out;  
        }  
  
我们可以通过调用栈找到写端，再从模块逻辑的角度找到它的读端，找到读写端后再做相应的保护即可。  
如果这种读段状态不可控，就可以用下面语句忽略这个信号，这样可避免程序crash。  
signal (SIGPIPE， SIG\_IGN)；  
  
小结：  
前面讲的都是代码中显示地触发信号的情况，  
其中SIGABRT、SIGSTKFLT、SIGFPE是用户态发下去的，调用栈的顶端应该是tgkill()，  
如果不是tgkill()就有可能这个信号是别处的模块发送给当前模块的，属于特殊情况，很少会遇到。  
SIGPIPE是系统调用write()在kernel中的执行流程发出来的。与上面三种不一样，一般调用栈顶端是write()函数。  
  
剩下三种信号SIGILL、SIGBUS、SIGSEGV都是CPU异常处理程序触发的，分析它的时候我们需要格外地关注寄存器及内存的值，分析这些问题需要一定的汇编基础。  
  
5、SIGSEGV  
段错误是我们最常见的信号，分SEGV\_MAPERR和SEGV\_ACCERR两种。  
SEGV\_MAPERR：地址不在进程地址空间内时触发：  
pid: 1219, tid: 1219, name: ndroid.systemui >>> com.android.systemui <<<  
signal 11 (SIGSEGV), code 1 (SEGV\_MAPERR), fault addr 00000000  
r0 00000000 r1 00000000 r2 000010a0 r3 00000175  
r4 be9bc150 r5 00000000 r6 be9bbfac r7 4053763d  
r8 00000174 r9 00001200 sl 00001200 fp 000010e0  
ip 40664b9c sp be9bbf8c lr 40538701 pc 40128310 cpsr 200f0010  
backtrace:  
#00 pc 00022310 /system/lib/libc.so (memset+24)  
#01 pc 000b66ff /system/lib/libskia.so (SkDraw::drawPaint(SkPaint const&) const+286)  
#02 pc 000b1023 /system/lib/libskia.so (SkCanvas::internalDrawPaint(SkPaint const&)+66)  
#03 pc 000aff65 /system/lib/libskia.so (SkCanvas::drawColor(unsigned int, SkXfermode::Mode)+44)  
#04 pc 0002034c /system/lib/libdvm.so (dvmPlatformInvoke+112)  
  
从调用栈中可以看出，程序执行到memset+24的位置时，cpu发现异常。  
  
我们可以通过gdb或者objdump工具查看这个汇编：  
(gdb) disassemble 0x401282f8  
Dump of assembler code for function memset:  
   0x401282f8 <+0>:    stmfd    sp!, {r0}  
   0x401282fc <+4>:    vdup.8    q0, r1  
   0x40128300 <+8>:    subs    r2, r2, #32  
   0x40128304 <+12>:    bcc    0x4012e318 <memset+32>  
   0x40128308 <+16>:    vorr    q1, q0, q0  
   0x4012830c <+20>:    subs    r2, r2, #32  
   0x40128310 <+24>:    vst1.8    {d0-d3}, [r0]!            <<<<  
  
可以看到是把d0-d3寄存器的值写到r0寄存器指向的地址时发生的异常。  
我们可以从r0寄存器的值可以知道，这个地址是0x00000000，而0x00000000不在进程地址空间范围内，所以会引起SEGV\_MAPERR错误。  
"fault addr 00000000" 这个信息也能说明问题，但我们不看汇编不能确定是哪个寄存器（r0和r1都有可能）。  
看了汇编后能确定是r0，也就是memset的第一个参数为空导致了这个问题。  
  
  
SEGV\_ACCERR：地址在进程地址空间内，但访问权限不够时触发。  
pid: 246, tid: 13956, name: Binder\_1  >>> /system/bin/mediaserver <<<  
signal 11 (SIGSEGV), code 2 (SEGV\_ACCERR), fault addr b74eeee0  
    r0 b771c1a0  r1 b771c0a0  r2 0f2ac7ff  r3 b74eeeb8  
    r4 b771c1a0  r5 b74eeee0  r6 b0162878  r7 b6f99394  
    r8 b6f99394  r9 b011f255  sl b011eabd  fp b011f255  
    ip b6f99394  sp affcf010  lr b00d61c9  pc b74eeee0  cpsr 600b0010  
backtrace:  
    #00  pc 00000ee0  [heap]  
    #01  pc 001061c7  /system/lib/libchromium\_net.so  
    #02  pc 0010495b  /system/lib/libchromium\_net.so  
    #03  pc 00104a1f  /system/lib/libchromium\_net.so  
    #04  pc 00104d63  /system/lib/libchromium\_net.so  
    #05  pc 00104de5  /system/lib/libchromium\_net.so  
  
memory map around fault addr b74eeee0:  
    b6fc5000-b6fc6000 rw-  
    b74ee000-b7728000 rw- [heap]  
    bed7c000-bed9d000 rw- [stack]  
  
上面出错的地址是b74eeee0，这个地址是有rw权限的，少的是x权限也就是可执行权限。  
又pc地址是b74eeeb8，因此不看汇编也能确定这里是pc指针跑飞导致的，一般是函数指针被覆盖导致的。  
  
6、SIGILL  
有ILL\_ILLOPC和ILL\_ILLTRP两类。  
ILL\_ILLOPC类表示PC指向的地址在进程地址空间范围内，且也有可执行访问权限，  
但arm解析指令时，发现不是合法指令，这个时候会发ILL\_ILLOPC类型的SIGILL信号。  
  
pid: 29499, tid: 29499, name: m.miui.weather2 >>> com.miui.weather2 <<<  
signal 4 (SIGILL), code 1 (ILL\_ILLOPC), fault addr 750f6064  
r0 8d2b7c00 r1 8d2b7c00 r2 bef112e8 r3 00000009  
r4 8d2b7c00 r5 bef112e8 r6 71a56f80 r7 71a56fe0  
r8 00000001 r9 00007205 sl 00007206 fp 00007211  
ip 40188f38 sp bef112c0 lr 750fb324 pc 750f6064 cpsr a0070030  
backtrace:  
#00 pc 00000064 /data/app-lib/com.lvtech.ydserver-2/libjohn.so  
#01 pc 00005320 /data/app-lib/com.lvtech.ydserver-2/libjohn.so  
#02 pc 0001d3ed /system/lib/libbinder.so (android::IPCThreadState::talkWithDriver(bool)+140)  
#03 pc 0001d8d7 /system/lib/libbinder.so (android::IPCThreadState::waitForResponse(android::Parcel\*, int\*)+42)  
  
一般静态编译的代码不可能有非法指令，所以这种情况经常是函数执行到动态生成的代码时会遇到。  
比如dvm中的git，又比如带有root权限的第三方APP修改code段（如libjohn）时会出现这种错误。  
  
ILL\_ILLTRP类会在系统调用出错时触发，一般极少能遇到。  
  
7、SIGBUS  
有BUS\_ADRERR和BUS\_ADRALN两类。  
一般文件映射到进程后映射的内存大小比实际的文件大小大，当进程访问这个映射后的内存时，如果访问到超出实际文件大小的地址时，会触发BUS\_ADRERR类SIGBUS信号。  
如果是随机的，可能是野指针引起的，如下：  
pid: 10214, tid: 10960, name: Visualizer >>> com.android.systemui <<<  
signal 7 (SIGBUS), code 2 (BUS\_ADRERR), fault addr 67481000  
r0 67480ff8 r1 00000080 r2 0071ece4 r3 00770065  
r4 00000000 r5 6747fcb8 r6 67b9fd24 r7 67b9fd20  
r8 67b9fd18 r9 67aa2000 sl bee8d294 fp 400e22ec  
ip 40ccfbd8 sp 6747fc94 lr 40ca2bd3 pc 400b8310 cpsr 20020010  
backtrace:  
#00 pc 00022310 /system/lib/libc.so (memset+24)  
#01 pc 00067bcf /system/lib/libmedia.so (android::Visualizer::getWaveForm(unsigned char\*)+58)  
#02 pc 00067d29 /system/lib/libmedia.so (android::Visualizer::periodicCapture()+68)  
#03 pc 00067da7 /system/lib/libmedia.so (android::Visualizer::CaptureThread::threadLoop()+8)  
因为代码中的memset是操作堆地址，并不是文件映射。正常情况下是不会出现BUS\_ADRERR错误的，除非这个堆已经被释放，后面又被映射了文件。  
  
如果是高概率的，可能是模块本身的问题了。  
pid: 3413, tid: 31109, name: Thread-198 >>> com.miui.securitycenter <<<  
signal 7 (SIGBUS), code 2 (BUS\_ADRERR), fault addr 758ab000  
r0 00000002 r1 0000000b r2 0000000a r3 00000097  
r4 00000012 r5 758aaffe r6 00000000 r7 755cbfc0  
r8 755cb548 r9 00000001 sl 00000001 fp 758c3b85  
ip 758aaffe sp 766d78e8 lr 402aca58 pc 402ae674 cpsr 800e0010  
backtrace:  
#00 pc 0000d674 /system/lib/libz.so (inflate\_fast+1544)  
#01 pc 0000ba54 /system/lib/libz.so (inflate+5644)  
#02 pc 000172f9 /system/lib/libandroidfw.so (android::ZipFileRO::inflateBuffer(void\*, void const\*, unsigned int, unsigned int)+96)  
#03 pc 0000c64d /system/lib/libandroidfw.so (android::\_CompressedAsset::getBuffer(bool)+48)  
#04 pc 0000daa3 /system/lib/libandroidfw.so (android::AssetManager::SharedZip::setResourceTableAsset(android::Asset\*)+34)  
#05 pc 0000e3db /system/lib/libandroidfw.so (android::AssetManager::ZipSet::setZipResourceTableAsset(android::String8 const&, android::Asset\*)+26)  
这里正在操作ZipFile，通过maps可以看到，这个地址确实对应文件。  
  
所以这类问题的关键是得通过调用栈和概率先区分是否是文件映射大了还是野指针引起的。  
  
BUS\_ADRERR类的SIGBUS一般是地址不对齐导致的  
pid: 1252, tid: 1278, name: Binder\_2  >>> com.android.systemui <<<  
signal 7 (SIGBUS), code 1 (BUS\_ADRALN), fault addr 75326ec9  
    r0 00000002  r1 00002801  r2 00000005  r3 75321a84  
    r4 6efefb95  r5 753219f4  r6 74a56b40  r7 0000f5f3  
    r8 416fb4c0  r9 00000103  sl 753219e0  fp 7531ba84  
    ip 00005445  sp 7531b948  lr 00000000  pc 416ff198  cpsr 20070010  
#00  pc 00024198  /system/lib/libdvm.so  
#01  pc 00030c6c  /system/lib/libdvm.so (dvmMterpStd(Thread\*)+76)  
#02  pc 0002e304  /system/lib/libdvm.so (dvmInterpret(Thread\*, Method const\*, JValue\*)+184)  
#03  pc 00063431  /system/lib/libdvm.so (dvmCallMethodV(Thread\*, Method const\*, Object\*, bool, JValue\*, std::\_\_va\_list)+336)  
#04  pc 00063455  /system/lib/libdvm.so (dvmCallMethod(Thread\*, Method const\*, Object\*, JValue\*, ...)+20)  
  
通过gdb可以看出  
   0x416ff190 <+15556>:    cmp    r3, #0  
   0x416ff194 <+15560>:    beq    0x41704a6c <common\_errNullObject>  
=> 0x416ff198 <+15564>:    ldrd    r0, [r3, r12]  
  
读取r3(0x75321a84)偏移r12(0x00005445)的地址(0x75326ec9)的时候因为地址不对齐导致BUS\_ADRERR类异常。  
CPU异常相关的NativeCrash基本上（除了BUS\_ADRERR类SIGBUS）都需要分析内存，也就是需要coredump来分析。  
  
  
**六、关于野指针**  
因为堆是随机离散地分布在进程空间内，它的危害是及其严重的。一个野指针可能会引起同进程的任何一个模块发生上述7种异常中的任意一种异常。  
  
如果野指针指向的地址中存在指针并且相关模块会读写这个指针，那野指针模块出错的概率非常高，且调用栈一般都是一样的，这种情况比较好定位问题，比如Virsualizer模块野指针问题。  
  
如果野指针指向的地址没有指针，也就是纯数据  
1、如果有明显pattern，比如大量相同的字节、字符串、特殊地址等，可以通过pattern来猜测出问题的模块。  
2、如果没有明显pattern，那只能通过log查看出问题前后的操作，尽量提高复现概率。  
3、如果没有明显pattern，概率也低，那这种问题就解不了了。

如果概率低要么是被野指针破坏的，要么就是上述的第三种情况，我们不需要再这种低概率问题上浪费时间。  
所以野指针问题最关键的就是看概率，有概率有样本才有可能解决问题。