Android系统中的应用被Activity Manager及Window Manager两个系统服务监控着，Android系统会在如下情况触发ANR：

1.Input事件超过5s没有被处理完，即5秒内无法对输入事件（按键及触摸）做出响应

2.Service处理超时，前台20s，后台200s

3.BroadcastReceiver（广播接收器）处理超时，前台10S，后台60s

4.ContentProvider执行超时，比较少见

出现ANR之后一个直观现象就是系统会展示出一个ANR弹框。

从发生的原因分：

1.主线程有耗时操作，如有复杂的layout布局，IO操作等。

2.被Binder对端block

3.被子线程同步锁block

4.Binder被占满导致主线程无法和SystemServer通信

5.得不到系统资源（CPU/RAM/IO）

从进程的角度分：

1.问题出在当前进程；

2.主线程本身耗时, 或则主线程的消息队列存在耗时操作;

3.主线程被本进程的其他子线程所blocked;

4.问题出在远端进程(一般是binder call或socket等通信方式)

一般有三种类型:

1.KeyDispatchTimeout(5 seconds) –主要类型：按键或触摸事件在特定时间内无响应

2.BroadcastTimeout(10 seconds) –BroadcastReceiver：在特定时间内无法处理完成

3.ServiceTimeout(20 seconds) –小概率类型：Service在特定的时间内无法处理完成

4.另外还有ProviderTimeout和WatchDog看门狗等导致的ANR。

还有当系统内存或CPU资源不足时容易出现ANR， 一般这种情况会有lowmemorykill的log打印。

XCrash ANR监测：

1.API小于21时，监听/data/anr/文件夹

应用ANR产生的时候，在ActivityManagerService中会调用appNotResponding方法, 然后在/data/anr/traces.txt文件中写入ANR相关信息。

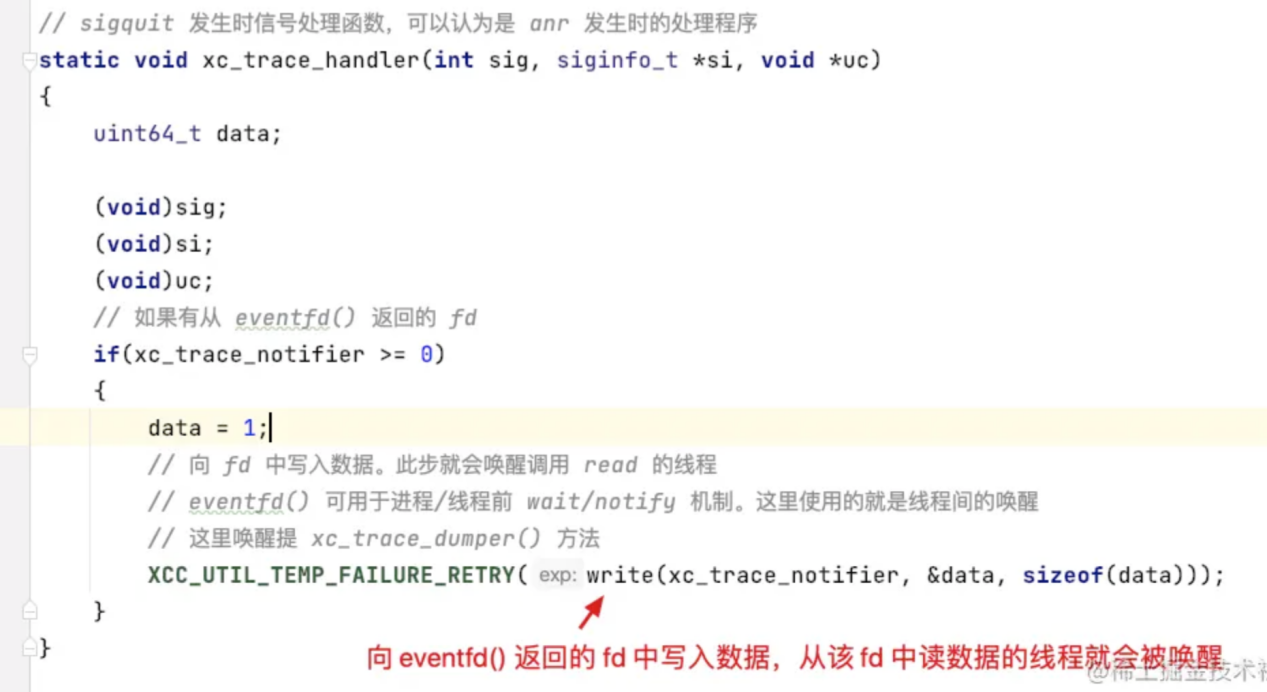
1. API大于21时，native监听信号



eventfd() 在 xcrash 中是用于线程通信。xcrash 会新开一线程用于 dump 日志，所以在发生 anr 时会通过 xc\_trace\_notifier 去唤醒该线程。



通过 sigaction() 注册了对 sigquit 的信号处理。当 sigquit 信号到来时会执行 xc\_trace\_handler。



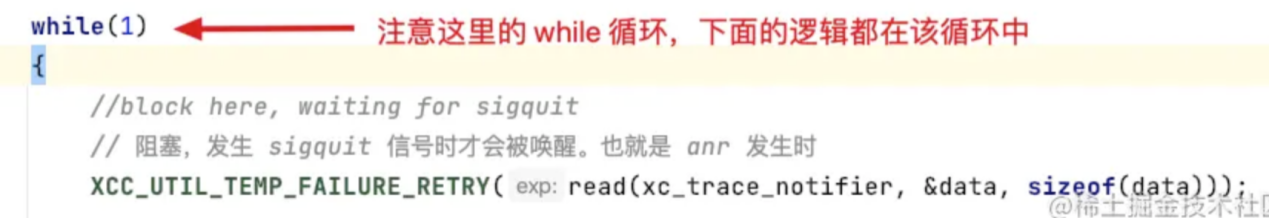
xc\_trace\_init() 中说过，xcrash 会单起一线程用于 dump 日志，这个线程执行的就是 xc\_trace\_dumper() 方法。

xc\_trace\_dumper

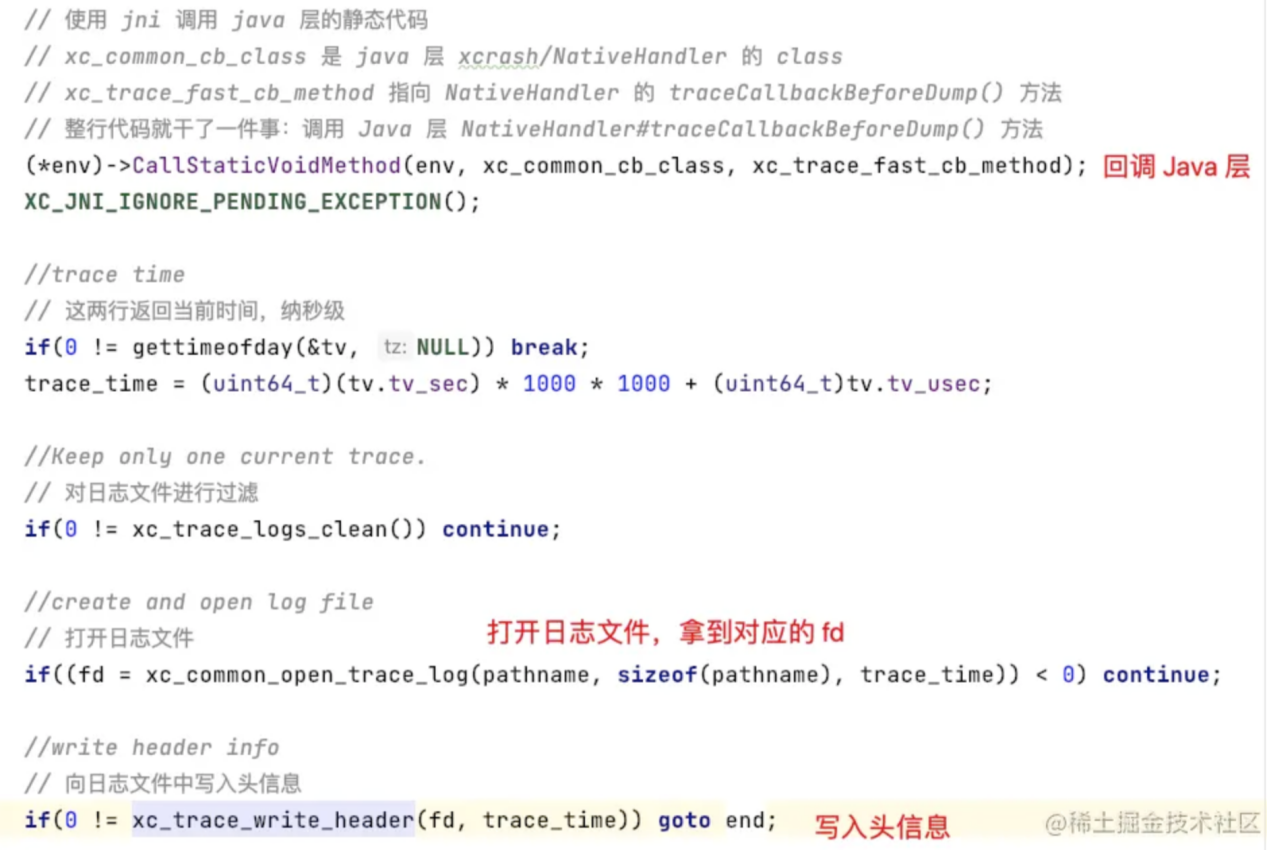
第一部分：关联到 jvm 上，用于回调一些 java 层的代码。由于当前线程是通过 pthread\_create 在 native 层创建的，所以需要关联到 jvm 后才可以执行 java 层代码。



第二部分，从 xc\_trace\_notifier 中读数据。最开始没数据时当前线程会被阻塞，只有 anr 发生时 xc\_trace\_handler 才会执行。



第三部分，回调 java 层，然后向日志文件中写入一些头信息。具体头信息有哪些，不重要，忽略。



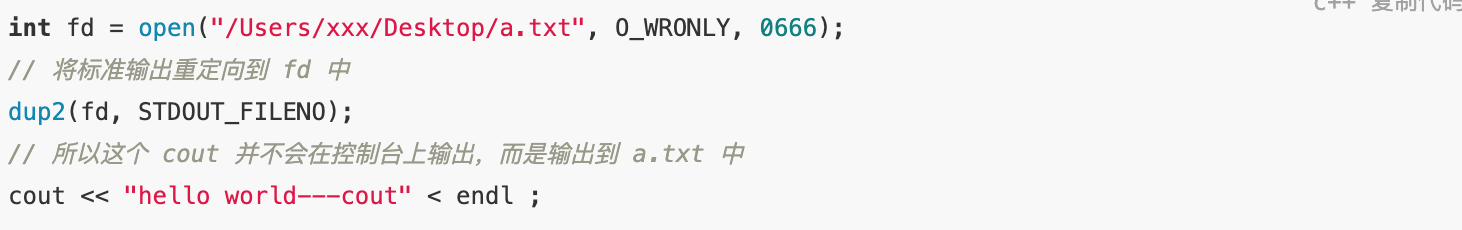
第四部分，解析 libc++/libart 两个 so，拿到某些函数的地址。这里面涉及到 elf 文件的解析。



第五部分，真正的导日志。其实就也调用 Runtime::DumpForSigQuit()，里面涉及到了重定向。



关于重定向 dup2 的使用可参考下面程序。



总结：

自定义 sigquit 信号处理程序，有信号到来时会唤醒一个子线程去 dump 日志

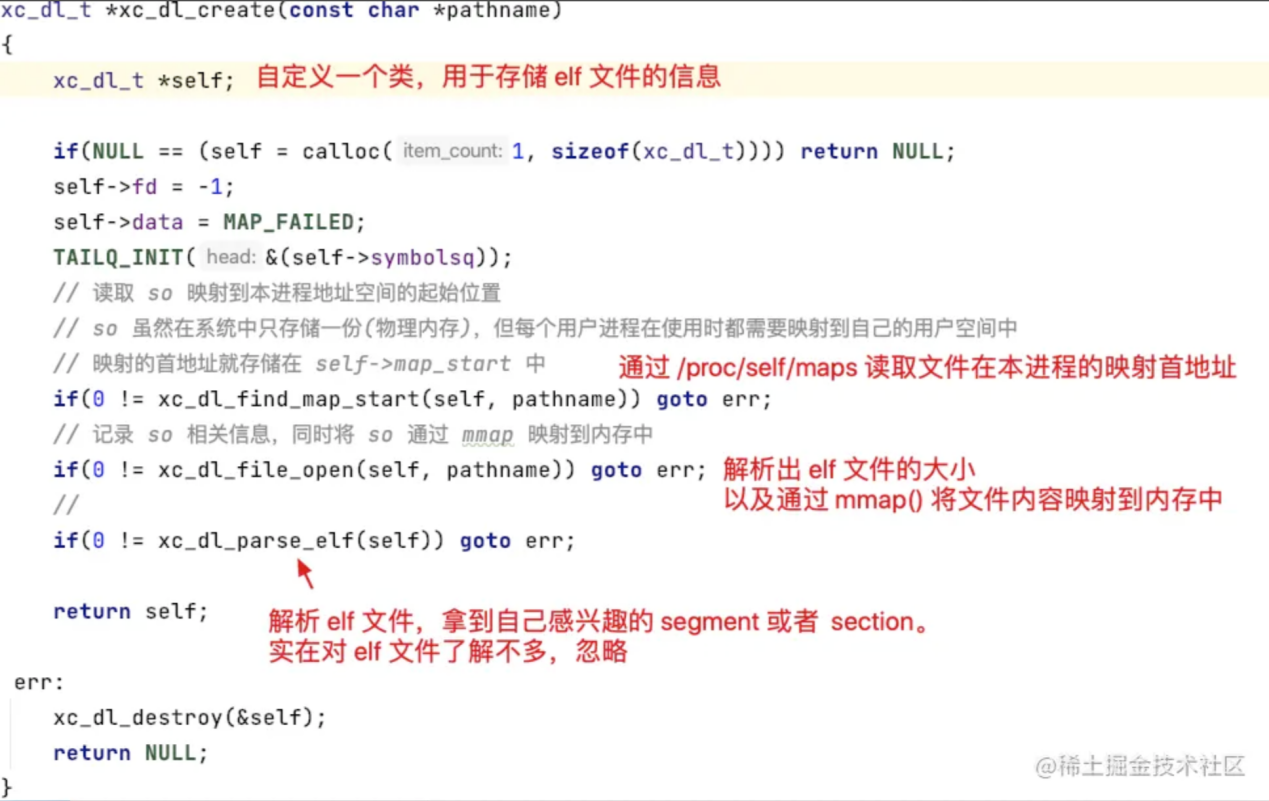
通过调用 Runtime 中方法，并将结果输出到日志文件完成日志收集。这里就涉及到 elf 文件的解析，以及从 elf 文件中拿到自己想要的方法的地址。在 c/c++ 中，拿到函数地址后就可以直接调用，其实有点像 java 的反射。

xc\_trace\_load\_symbols：

通过解析 elf 文件拿到想要的函数的地址。



xc\_dl\_create() 是通过读取 /proc/self/maps 文件找到相应 so 在本进程的地址。



xc\_dl\_parse\_elf() 就是解析出所有 elf 文件中所有感兴趣的 section 和 segment。xc\_dl\_create 解析出指定函数的地址。

xc\_dl\_sym() 是根据映射进来的 so 文件解析出指定的某些函数。

xCrash 信号拦截处理

zygote 进程 Block 了 SIGPIPE, SIGQUIT, SIGUSR1 三个信号。

fork 子进程继承了父进程的信号集。子进程通过 Signal Catcher 线程(sigwait)监听 Block 的 SIGQUIT,SIGUSR1 信号。

在多线程的应用里,信号产生只能有一个线程来处理信号。

信号捕获处理方式:

4.1 sigaction 设置全局信号处理程序,会保存旧的信号集,通过旧的信号集恢复旧的信号处理。

4.2 sigwait 用于子线程监听信号的到来 并且只能监听 Block 的信号。

那么问题来了, 如果自己捕获系统 SIGQUIT 信号该怎么做呢?

方案:

6.1 sigwait : 因为无法确定那一个子线程来处理信号,所以该方案不行。

6.2 sigaction :

6.2.1 拦截 Signal Catcher 线程对信号的捕获?

6.2.2 main 线程 sigaction 如何捕获呢?

6.2.3 non-main 线程 sigaction 如何捕获呢？

接下来看 xCrash 如何捕获 SIGQUIT 信号呢？

// handler 为信号处理函数

int xcc\_signal\_trace\_register(void (\*handler)(int, siginfo\_t \*, void \*))

{

int r;

sigset\_t set;

struct sigaction act;

// 下面的注释先解除信号的阻塞并且希望是主线程。

//un-block the SIGQUIT mask for current thread, hope this is the main thread

sigemptyset(&set);

sigaddset(&set, SIGQUIT);

// 先解除进程 Block 的信号。解决上面提到的问题6.2.1

// Signal Catcher 子线程就无法捕获到 SIGQUIT 信号了。

// 经过测试程序发现下面俩种方式 xCrash 均可以捕获到信号

// 1. pthread\_sigmask & sigaction 均在主线程可运行

// 2. pthread\_sigmask (mian) & sigaction (子线程) 也可以捕获到 SIGQUIT 信号 详见后面实践中的测试程序

// 虽然上面 2 的方式可以处理信号,但是由于信号处理本身单线程就比较复杂,因为信号打断了进程的控制流,所以信号处理函数只能调用异步安全的函数,而异步信号安全更是一个很苛刻的条件。

// 多线程条件下,加剧了这种复杂度,因为信号既可以发送给进程,也可以发送个进程的某一个线程,不同线程还可以设置掩码来实现对信号的屏蔽。

// 在多线程环境下一般建议采用sigwait/ sigwaitinfo等方式同步处理信号,减少异步处理带来的风险和引入 bug 的可能。

// 综上所述

// 1. Android 系统处理信号在子线程使用了 sigwait 的方式来捕获 SIGQUIT 信号

// 2. xCrash 则是采用单线程模式下捕获 SIGQUIT 信号(main 线程初始化信号相关操作)。

if(0 != (r = pthread\_sigmask(SIG\_UNBLOCK, &set, &xcc\_signal\_trace\_oldset))) return r;

// 为 SIGQUIT 注册新的信号处理程序。

//register new signal handler for SIGQUIT

memset(&act, 0, sizeof(act));

sigfillset(&act.sa\_mask);

act.sa\_sigaction = handler;

// 设置 SA\_RESTART属性, 那么当信号处理函数返回后, 被该信号中断的系统调用将自动恢复.

act.sa\_flags = SA\_RESTART | SA\_SIGINFO;

// 使用 sigaction 捕获,解决问题6.2.2

// 并且保存旧的信号集。用于恢复之前的信号处理程序

if(0 != sigaction(SIGQUIT, &act, &xcc\_signal\_trace\_oldact))

{

pthread\_sigmask(SIG\_SETMASK, &xcc\_signal\_trace\_oldset, NULL);

return XCC\_ERRNO\_SYS;

}

return 0;

}

生成 trace 文件

开启子线程

// create thread for dump trace

// If successful, the pthread\_create() function will return zero.

if(0 != (r = pthread\_create(&thd, NULL, xc\_trace\_dumper, NULL))) {

goto err1;

}

dump trace 文件

// dump trace fun

static void \*xc\_trace\_dumper(void \*arg)

{

// JNIEnv 用于回调 Java 函数。

JNIEnv \*env = NULL;

uint64\_t data;

uint64\_t trace\_time;

int fd;

struct timeval tv;

char pathname[1024];

jstring j\_pathname;

(void)arg;

pthread\_detach(pthread\_self());

JavaVMAttachArgs attach\_args = {

.version = XC\_JNI\_VERSION,

.name = "xcrash\_trace\_dp",

.group = NULL

};

// 又见到了熟悉的代码, 关联 C++ Thread 对象

if(JNI\_OK != (\*xc\_common\_vm)->AttachCurrentThread(xc\_common\_vm, &env, &attach\_args)) goto exit;

while(1)

{

// 子线程等待 SIGQUIT 信号的到来

//block here, waiting for sigquit

// 类似 Android Looper 的实现方案 创建 eventfd

// 当 SIGQUIT 信号发生时,write写入信息,这里read读取到信息就往下执行了。

XCC\_UTIL\_TEMP\_FAILURE\_RETRY(read(xc\_trace\_notifier, &data, sizeof(data)));

//check if process already crashed

if(xc\_common\_native\_crashed || xc\_common\_java\_crashed) break;

//trace time

if(0 != gettimeofday(&tv, NULL)) break;

trace\_time = (uint64\_t)(tv.tv\_sec) \* 1000 \* 1000 + (uint64\_t)tv.tv\_usec;

//Keep only one current trace.

if(0 != xc\_trace\_logs\_clean()) continue;

//create and open log file

if((fd = xc\_common\_open\_trace\_log(pathname, sizeof(pathname), trace\_time)) < 0) continue;

//write header info

if(0 != xc\_trace\_write\_header(fd, trace\_time)) goto end;

//write trace info from ART runtime

if(0 != xcc\_util\_write\_format(fd, XCC\_UTIL\_THREAD\_SEP"Cmd line: %s\n", xc\_common\_process\_name)) goto end;

if(0 != xcc\_util\_write\_str(fd, "Mode: ART DumpForSigQuit\n")) goto end;

if(0 != xc\_trace\_load\_symbols())

{

if(0 != xcc\_util\_write\_str(fd, "Failed to load symbols.\n")) goto end;

goto skip;

}

if(0 != xc\_trace\_check\_address\_valid())

{

if(0 != xcc\_util\_write\_str(fd, "Failed to check runtime address.\n")) goto end;

goto skip;

}

if(dup2(fd, STDERR\_FILENO) < 0)

{

if(0 != xcc\_util\_write\_str(fd, "Failed to duplicate FD.\n")) goto end;

goto skip;

}

xc\_trace\_dump\_status = XC\_TRACE\_DUMP\_ON\_GOING;

if(sigsetjmp(jmpenv, 1) == 0)

{

if(xc\_trace\_is\_lollipop)

xc\_trace\_libart\_dbg\_suspend();

// Runtime::DumpForSigQuit(os) 生成 trace 文件。

// 函数符号 \_ZN3art7Runtime14DumpForSigQuitERNSt3\_\_113basic\_ostreamIcNS1\_11char\_traitsIcEEEE

// 可以通过 http://demangler.com/ 反解函数名。

// 如何获取该函数的符号呢?

/\*\*

\* 启动模拟器

\* 1. adb shell

\* 2. su

\* 3. find | grep libart.so

\* 4. adb pull apex/com.android.art/lib64/libart.so ~/

\* 5. 配置电脑环境变量 用 ojbdump 查找函数符号

\* 6. aarch64-linux-android-objdump -Tw /local\_path/libart.so

\* \_ZN3art9JavaVMExt14DumpForSigQuitERNSt3\_\_113basic\_ostreamIcNS1\_11char\_traitsIcEEEE

\* \_ZN3art2gc4Heap14DumpForSigQuitERNSt3\_\_113basic\_ostreamIcNS2\_11char\_traitsIcEEEE

\* \_ZN3art7Runtime14DumpForSigQuitERNSt3\_\_113basic\_ostreamIcNS1\_11char\_traitsIcEEEE

\* 搜索函数名会有好多?

\* 可以结合http://demangler.com/来反解函数

\* 小技巧:一般函数名前面是类名

\*/

// 函数指针可以 sym 解析出函数指针地址。

xc\_trace\_libart\_runtime\_dump(\*xc\_trace\_libart\_runtime\_instance, xc\_trace\_libcpp\_cerr);

if(xc\_trace\_is\_lollipop)

xc\_trace\_libart\_dbg\_resume();

}

else

{

fflush(NULL);

XCD\_LOG\_WARN("longjmp to skip dumping trace\n");

}

dup2(xc\_common\_fd\_null, STDERR\_FILENO);

skip:

if(0 != xcc\_util\_write\_str(fd, "\n"XCC\_UTIL\_THREAD\_END"\n")) goto end;

//write other info

if(0 != xcc\_util\_record\_logcat(fd, xc\_common\_process\_id, xc\_common\_api\_level, xc\_trace\_logcat\_system\_lines, xc\_trace\_logcat\_events\_lines, xc\_trace\_logcat\_main\_lines)) goto end;

if(xc\_trace\_dump\_fds)

if(0 != xcc\_util\_record\_fds(fd, xc\_common\_process\_id)) goto end;

if(xc\_trace\_dump\_network\_info)

if(0 != xcc\_util\_record\_network\_info(fd, xc\_common\_process\_id, xc\_common\_api\_level)) goto end;

if(0 != xcc\_meminfo\_record(fd, xc\_common\_process\_id)) goto end;

end:

//close log file

xc\_common\_close\_trace\_log(fd);

//rethrow SIGQUIT to ART Signal Catcher

// 再发送 SIGQUIT 信号给 Signal Catcher 线程处理

if(xc\_trace\_rethrow && (XC\_TRACE\_DUMP\_ART\_CRASH != xc\_trace\_dump\_status)) xc\_trace\_send\_sigquit();

xc\_trace\_dump\_status = XC\_TRACE\_DUMP\_END;

//JNI callback

//Do we need to implement an emergency buffer for disk exhausted?

if(NULL == xc\_trace\_cb\_method) continue;

if(NULL == (j\_pathname = (\*env)->NewStringUTF(env, pathname))) continue;

// trace 文件生成回调 Java 函数

(\*env)->CallStaticVoidMethod(env, xc\_common\_cb\_class, xc\_trace\_cb\_method, j\_pathname, NULL);

XC\_JNI\_IGNORE\_PENDING\_EXCEPTION();

(\*env)->DeleteLocalRef(env, j\_pathname);

}

// 释放资源

(\*xc\_common\_vm)->DetachCurrentThread(xc\_common\_vm);

exit:

xc\_trace\_notifier = -1;

close(xc\_trace\_notifier);

return NULL;

}

SIGQUIT 信号恢复

sigquit rethrow art

static void xc\_trace\_send\_sigquit()

{

if(xc\_trace\_signal\_catcher\_tid >= 0) {

// 通过系统调用给 signal catcher 线程发送 SIGQUIT 信号

syscall(SYS\_tgkill, xc\_common\_process\_id, xc\_trace\_signal\_catcher\_tid, SIGQUIT);

}

}

