Android NDK crash 问题定位

NDK是使用C/C++来进行开发，指针和内存管理是最重要也是最容易出问题的地方，稍有不慎就会遇到诸如内存地址访问错误、使用野针对、内存泄露、堆栈溢出、初始化错误、类型转换错误、数字除0等常见的问题，导致最后都是同一个结果：程序崩溃。依靠在各个jni函数中打印日志来跟踪问题，效率实在是太低了，而且还定位不到问题。

一．NDK调试工具定位crash

可用NDK安装包中提供了三个调试工具：addr2line、objdump和ndk-stack来定位crash的地方。其中ndk-stack放在$NDK\_HOME目录下，与ndk-build同级目录。addr2line和objdump在ndk的交叉编译器工具链目录下。

signal.h是C标准函数库中的信号处理部分，定义了程序执行时如何处理不同的信号。信号用作进程间通信，报告异常行为(如除零)、用户的一些按键组合(如同时按下Ctrl与C键，产生信号SIGINT)。常见的crash类型如下表1所示

表1 常见的signal信号

|  |  |  |
| --- | --- | --- |
| SIGSEGV | SEGV\_MAPERR | 地址不在 /proc/self/maps 映射中 |
| SEGV\_ACCERR | 没有访问权限 |
| SEGV\_MTESERR | MTE特有类型 |
| SIGSEGV |  | 程序主动退出，常见调用函数abort()，raise()等 |
| SIGILL | ILL\_ILLOPC | 非法操作码（opcode） |
| ILL\_ILLOPN | 非法操作数（operand） |
| ILL\_ILLADR | 非法寻址 |
| ILL\_ILLTRP | 非法trap，如\_builtintrap()主动崩溃 |
| ILL\_PRVOPC | 非法特权操作码(privileged opcode) |
| ILL\_PRVREG | 非法特权寄存器(privileged register) |
| ILL\_COPROC | 协处理器错误 |
| ILL\_BADSTK | 内部堆栈错误 |
| SIGBUS | BUS\_ADRALN | 访问地址未对齐 |
| BUS\_ADRERR | 访问不存在的物理地址 |
| BUS\_OBJERR | 特定对象的硬件错误 |
| SIGFPE | FPE\_INTDIV | 整数除以0 |
| FPE\_INTOVF | 整数溢出 |
| FPE\_FLTDIV | 浮点数除以0 |
| FPE\_FLTOVF | 浮点数上溢（overflow） |
| FPE\_FLTUND | 浮点数下溢（underflow） |
| FPE\_FLTRES | 浮点数结果不精确 |
| FPE\_FLTINV | 无效的浮点运算 |
| FPE\_FLTSUB | 越界 |

native crash不同于java/kotlin层的crash，在java环境中，如果程序出现了不可预期的crash（即没有捕获），就会往上抛出给最终的线程uncaghtexceptionhandler，在这里我们可以再次处理，比如屏蔽某个exception即可保持app的稳定，然后native层的crash不一样，native 层的crash大多数是“不可恢复”的，比如某个内存方面的错误，这些往往是不可处理的，需要中断当前进程，所以如果发生了native crash。Android 的错误机制会相应的会生成一份 tombstone 文件保存到 /data/tombstones/tombstone\_xx ，对于没有 Root 权限的机器则可以通过 adb bugreport 抓取出 tombstone 文件。日志中有明确的程序类型、时间戳、进程号、线程号、程序名、错误类型、错误原因等，如图1所示。

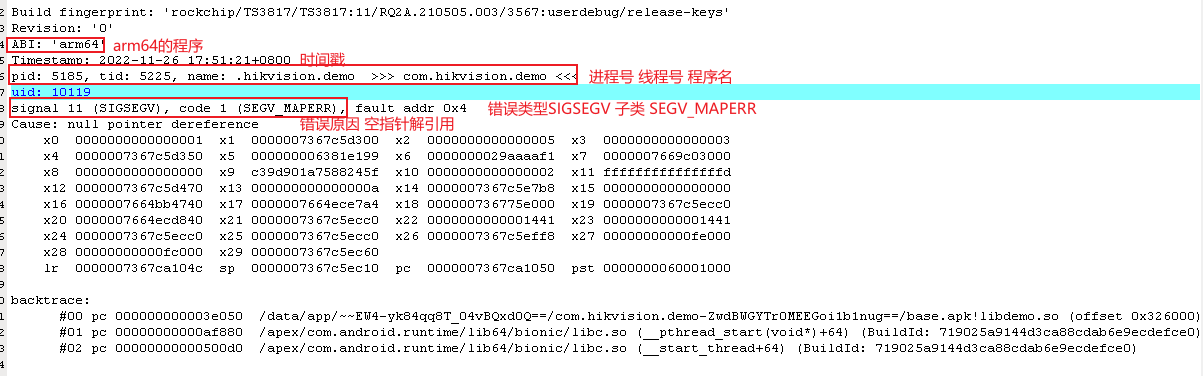


图1 tombstone文件

通过编写NDK一个例子来演示一下如何精确的定位错误，此处崩溃的原因在于空指针异常,如图2所示。



图2 NDK崩溃的示例程序

由backtrack中可以大致定位到开启线程中出现崩溃，后面的 “64” 并不是指原始代码中的第 64 出现问题，实际上编译工具默认在编译过程中会进行优化和对齐。如图3所示。

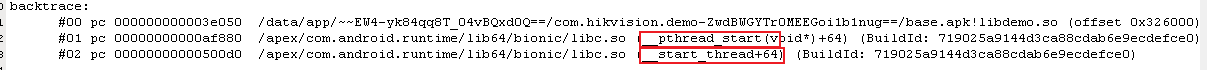


图3 backtrace问题跟踪

addr2line是通过pc（程序计数器）值来定位代码，-e后加 .so 文件名，“-f” 表示输出函数名。首先找出addr2line所在的位置，根据你目标机器的CPU架构来选择，本机的arm64架构选择addr2line的路径为

D:\AndroidSDK\ndk\21.0.6113669\toolchains\aarch64-linux-android-4.9\prebuilt\windows-x86\_64\bin\aarch64-linux-android-addr2line.exe。执行结果如图所示。从结果可得崩溃的函数名\_Z14threadCallbackPv，在native-lib.cpp的第14行位置。如图4所示。

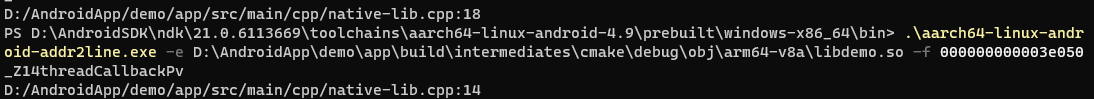


图4 addr2line定位crash问题

上述的\_Z14threadCallbackPv是编译器在编译时对函数名按照一定的规则进行了优化，可利用 <https://demangler.com/>所展示的工具进行转换。转换结果为threadCallback(void\*),与代码中的函数名一致，如图5所示。

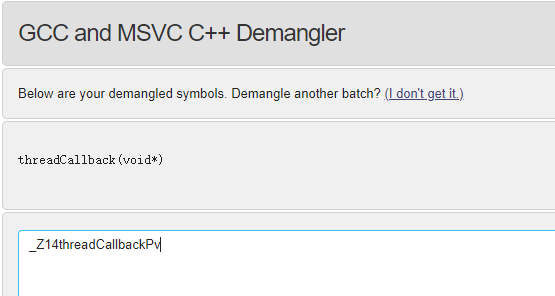


图5 函数名重解析

objdump显示来自对象文件的信息，对目标文件(obj)或可执行文件进行反汇编，它以一种可阅读的格式让你更多的了解二进制文件可能带有的附加信息。-S:将反汇编代码与源代码交叉显示，-D:对所有的sections进行反汇编。

在to\_string()函数中出现崩溃，符合源代码中出错的地方，执行结果如图6所示。

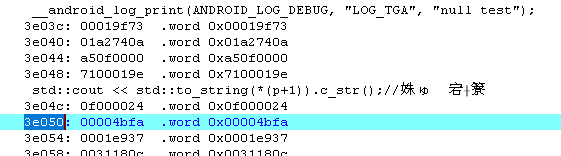


图6 objdump定位到crash位置

要使用 ndk-stack，首先要有一个包含未剥离版应用共享库的目录。如果您使用 ndk-build，则可在 $PROJECT\_PATH/obj/local/<abi> 中找到这些未剥离版共享库，其中 <abi> 是设备的 ABI。将logcat的崩溃信息保存到文件a.txt，崩溃信息一般到backstrace即可，使用ndk-stack -sym so库所在的目录-i崩溃信息即可还原崩溃的堆栈信息，如图7所示。

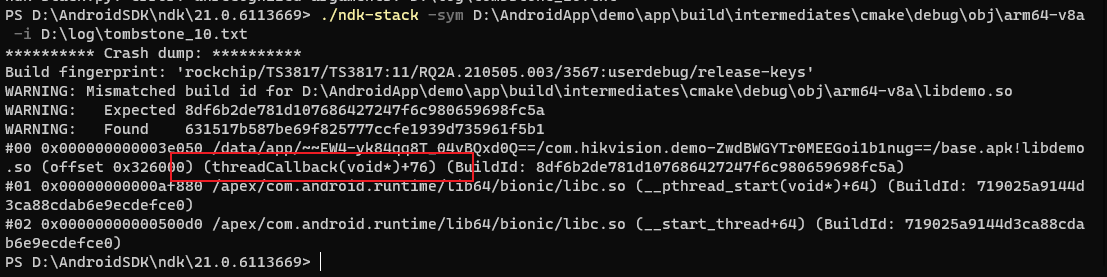


图7 使用ndk-stack定位crash信息

二．Address Sanitizer内存静态检测

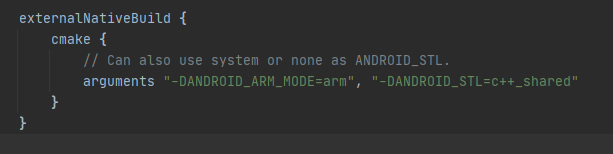
内存访问错误，包括缓冲区溢出和释放堆内存的使用，仍然是C和C++等编程语言的严重问题。这些错误既影响了系统的稳定性，也影响了程序的安全性。另外，内存错误的排查是困难的，很多时候导致问题的地方和发生问题的地方相隔甚远。

Address Sanitizer(ASan)是一个快速的内存错误检测工具，它可以检测堆栈和堆缓冲区上溢/下溢、释放之后的堆使用情况、超出范围的堆栈使用情况、重复释放/错误释放等问题。在Android中，从 API 级别 27 (Android O MR 1) 开始支持Address Sanitizer。

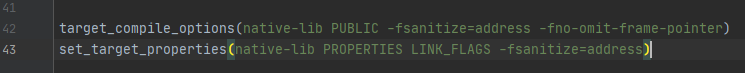
ASan可在32位和64位ARM以及x86和x86-64上运行。ASan的CPU开销约为 2 倍，代码大小开销在50%到2倍之间，并且内存开销很大（具体取决于您的分配模式，但约为 2 倍）。对于64位ARM，强烈建议使用 [HWAddress Sanitizer](https://developer.android.google.cn/ndk/guides/hwasan)。

ASan配置按以下步骤进行：

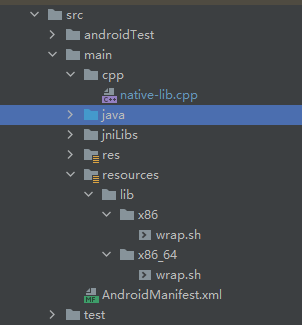
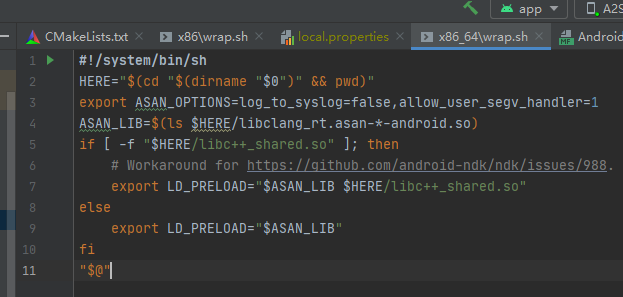
1.在app的 build.gradle的defaultConfig中增加一下配置。



2.在CMakeLists.txt中增加target\_compile\_options和set\_target\_properties配置

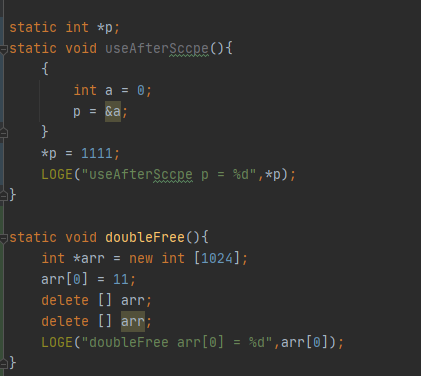
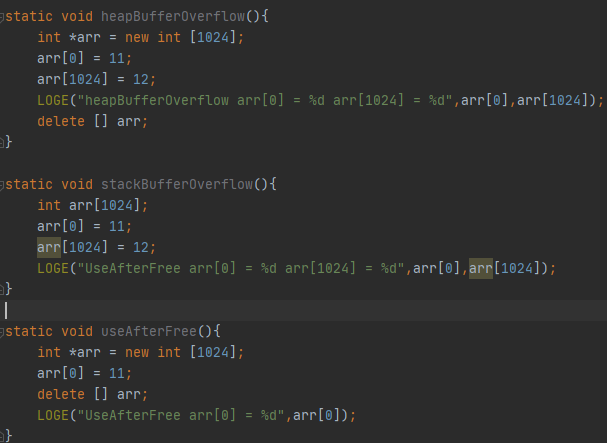


3.输入以下内容至wrap.sh，并将wrap.sh放置到如下目录中

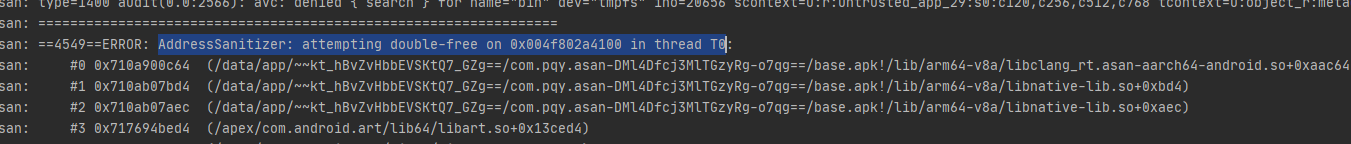
 

4.选用r20版本后的NDK中的libclang\_rt.asan-aarch64-android.so路径如下ndk\21.3.6528147\toolchains\llvm\prebuilt\windows-x86\_64\lib64\clang\9.0.8\lib\linux。放置jniLibs文件夹下。

选用以下示例做测试，分别为堆访问越界、栈访问越界、使用释放后的空间、多次释放等。

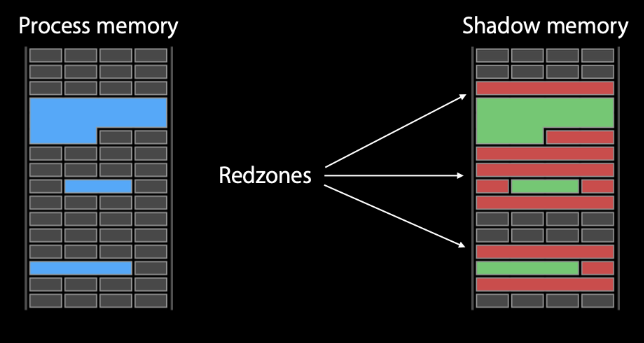


以多此释放为例，会在检测结果中出现attempting double-free的检测结果，运行结果如下图所示。





Address Sanitizer主要是接管了应用程序的 malloc 流程，每次分配内存，一定使用全新的虚拟地址，并且在地址两端加页保护。分配过的地址空间，即使 free 过了，也不会再次分配给用户。用户感受到的就是内存分配器每次分配出来的内存地址总是比上一次的高。已经 free 过的内存区域这里称为 Dead Zone。每次分配一段内存时，总会在两端预留一个页面，并且将这两个页面做读写保护。两端预留的页面称为 Red Zone。内存分配结果如图所示。



在这个机制的基础上，ASAN 只需要跟踪好每次内存分配的堆栈，就能在出问题的时候准确报告出出问题点。能探测到的内存问题：访问已经 free 过的内存，触发 Dead Zone 报警；内存访问越界，触发 Red Zone 报警；部分野指针问题，触发 Dead Zone / Red Zone 报警