Android tombstone产生机制与分析技巧

Last edited by **caoquanli** 1 month ago

Android 8.1 tombstone产生机制与分析技巧

Table of Contents

1. 概述

1.1. tombstone是什么?

- 在Android系统的调试机制中，tombstone是一种非常重要的文件，记录的是系统Native层某个进程崩溃时的各种堆栈、线程、寄存器等基本信息。简单说，就和其字面意思"墓碑"想要表达的一样，告诉开发者这个进程挂掉时的状态。

1.2. tombstone文件组成

- 当进程发生crash时，会在/data/tombstones目录下产生tombstone_xx文件（其中user版本最多10个，userdebug、eng版本最多50个，文件写满后会覆盖最早产生的那个），文件的基本组成如下：

1.2.1. 设备信息

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'google/angler/angler:8.0.0/OPR5.170623.007/4302479:userdebug/dev-keys'
Revision: '0'
ABI: 'arm64'
```

- 首先是一堆带空格星号，可以通过这种字符串快速查找tombstone文件；然后是通过系统属性参数获取到设备签名（系统指纹）、硬件版本号、ABI（架构类型，如arm、arm64、x86、x86_64）。

1.2.2. 线程信息

```
pid: 28277, tid: 28288, name: Binder:28277_1  >>> system_server <<<
```

- 包括了进程号、线程号、线程名称、进程名称；进程号与线程号相同，表示的是主线程；这一条可以很直观的告诉你是哪个进程挂掉了。

1.2.3. 信号信息

```
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
```

- 信号是应用开始运行时向内核注册的，触发写tomstome的信号有SIGABRT、SIGBUS、SIGFPE、SIGILL、SIGSEGV、SIGTRAP等，具体含义后面再详细介绍；通过分析信号类型也可以简单分析这个进程crash的原因。

1.2.4. 中止消息

```
Abort message: 'indirect_reference_table.cc:243] JNI ERROR (app bug): global reference table overflow (max=51200)'
```

- 中止行信息只有在发生中止时才会有，程序可以通过调用abort函数自行终止。

1.2.5. 寄存器信息

```
    x0   0000000000000000  x1   0000000000006e80  x2   0000000000000006  x3   0000000000000008
    x4   0000000000000000  x5   0000000000000000  x6   0000000000000000  x7   7f7f7f7f7f7f7f7f
    ……
    fpsr 08000013  fpcr 00000000
```

- 收到信号时CPU寄存器里面的内容；因为和CPU相关，不同的ABI,这块内容的格式也有较大差异。

1.2.6. 回溯信息

```
backtrace:
    #00 pc 000000000006a1dc  /system/lib64/libc.so (tgkill+8)
    #01 pc 000000000001d77c  /system/lib64/libc.so (abort+88)
    #02 pc 000000000043d974  /system/lib64/libart.so (_ZN3art7Runtime5AbortEPKc+528)
    #03 pc 000000000043e0c0  /system/lib64/libart.so (_ZN3art7Runtime7AborterEPKc+24)
    #04 pc 000000000052919c  /system/lib64/libart.so (_ZN7android4base10LogMessageD1Ev+912)
    #05 pc 000000000024ded8  /system/lib64/libart.so (_ZN3art22IndirectReferenceTable3AddENS_15IRTSegmentStateENS_6ObjPtr
    ……
    #33 pc 0000000000062250  /system/lib64/libbinder.so (_ZN7android14IPCThreadState14joinThreadPoolEb+60)
    #34 pc 0000000000082c84  /system/lib64/libbinder.so (_ZN7android10PoolThread10threadLoopEv+24)
    #35 pc 0000000000011638  /system/lib64/libutils.so (_ZN7android6Thread11_threadLoopEPv+280)
    #36 pc 00000000000b6544  /system/lib64/libandroid_runtime.so (_ZN7android14AndroidRuntime15javaThreadShellEPv+136)
    #37 pc 0000000000066a4c  /system/lib64/libc.so (_ZL15__pthread_startPv+36)
    #38 pc 000000000001eb94  /system/lib64/libc.so (__start_thread+68)
```

- 回溯信息是非常有用的信息，通过addr2line工具可以定位到代码中发生crash的具体位置。需要注意的是帧的顺序是从下往上，因此在调试的时候，一般从最后一条去定位crash的地方。

1.2.7. 堆栈信息

```
stack:
        0000007b503517e0  0000000000000000
        0000007b503517e8  0000007b4c611670  [anon:libc_malloc]
        0000007b503517f0  0000007b50351850
        0000007b503517f8  00000055846e8408  /system/bin/app_process64 (sigprocmask+284)
        0000007b50351800  0000007b4c611600  [anon:libc_malloc]
        0000007b50351808  0000007b503518c8
        0000007b50351810  ffffffffffffffdf
        0000007b50351818  e1169b2d66798dc6
```

```
        0000007b50351820  ffffffffffffffff
        0000007b50351828  00000000000036e1
        0000007b50351830  0000000000000000
        0000007b50351838  0000007b4c611600  [anon:libc_malloc]
        0000007b50351840  0000000000006e80
        0000007b50351848  0000000000006e75
        0000007b50351850  0000007b503518a0
        0000007b50351858  0000007b6c1e0770  /system/lib64/libc.so (abort+76)
    #00  0000007b50351860  0000007b5fdf6800  [anon:libc_malloc]
        ........  ........
    #01  0000007b50351860  0000007b5fdf6800  [anon:libc_malloc]
        0000007b50351868  00000000000000e1
        0000007b50351870  00000000000036dd
        0000007b50351878  ffffffffffffffff
        ……
```

- 更多的堆栈信息，与第<6>点回溯信息类似，但是可能残留未初始化或者未清空的信息。这里的地址是后面可执行文件或者so动态库的基地址+偏移量，需要转换才能使用工具进行定位。

1.2.8. memory信息

```
memory near x16:
    0000007b6c28b2d8 0000007b6c1df470 00000055846e82ec  p..l{.....n.U...
    0000007b6c28b2e8 0000007b6c1ebed0 0000007b6c1ebe24  ...l{...$..l{...
    0000007b6c28b2f8 0000007b6c22d1d4 0000007b6c1ebe98  .."l{......l{...
    0000007b6c28b308 00000055846e809c 0000007b6c22c4e4  ..n.U....."l{...
    0000007b6c28b318 0000007b6c1e07d8 0000007b6c1e2e14  ...l{......l{...
    0000007b6c28b328 0000007b6c1e08b0 0000007b6c22797c  ...l{...|y"l{...
    0000007b6c28b338 0000007b6c20ca28 0000007b6c1e07e4  (. l{......l{...
    0000007b6c28b348 0000007b6c218214 0000007b6c1ec61c  ..!l{......l{...
    ……
code around pc:
    0000007b6c22d1b8 d28009a8d65f03c0 b140041fd4000001  .._..........@.
    0000007b6c22d1c8 54df57c8da809400 d2801068d65f03c0  .....W.T.._.h...
    0000007b6c22d1d8 b140041fd4000001 54df5708da809400  ......@......W.T
    0000007b6c22d1e8 d2800aa8d65f03c0 b140041fd4000001  .._..........@.
    0000007b6c22d1f8 54df5648da809400 d2800ae8d65f03c0  ....HV.T.._.....
    0000007b6c22d208 b140041fd4000001 54df5588da809400  ......@......U.T
    ……
code around sp:
    0000007b50351840 0000000000006e80 0000000000006e75  .n......un......
    0000007b50351850 0000007b503518a0 0000007b6c1e0770  ..5P{...p..l{...
    0000007b50351860 0000007b5fdf6800 00000000000000e1  .h._{...........
    0000007b50351870 00000000000036dd ffffffffffffffff  .6..............
    0000007b50351880 0000000000000000 ffffffffffffffdf  ................
    ……
memory map:
    00000000'12c00000-00000000'52bfffff rw-         0  40000000  /dev/ashmem/dalvik-main space (region space) (deleted)
    00000000'707e9000-00000000'70a8ffff rw-         0    2a7000  /data/dalvik-cache/arm64/system@framework@boot.art
    00000000'70a90000-00000000'70b95fff rw-         0    106000  /data/dalvik-cache/arm64/system@framework@boot-core-liba
    00000000'70b96000-00000000'70bdffff rw-         0     4a000  /data/dalvik-cache/arm64/system@framework@boot-conscrypt
    00000000'70be0000-00000000'70c15fff rw-         0     36000  /data/dalvik-cache/arm64/system@framework@boot-okhttp.ar
    00000000'70c16000-00000000'70c19fff rw-         0      4000  /data/dalvik-cache/arm64/system@framework@boot-legacy-te
    00000000'70c1a000-00000000'70c5cfff rw-         0     43000  /data/dalvik-cache/arm64/system@framework@boot-bouncycas
  ……
```

- 内存中的一堆信息，其中memory near x16表示靠近寄存器x16的内存信息，memory map表示的内存映射。

1.2.9. 进程关联的文件列表

```
open files:
    fd 0: /dev/null
    fd 1: /dev/null
    fd 2: /dev/null
    fd 3: /dev/cpuset/foreground/tasks
    fd 4: /dev/cpuset/background/tasks
    fd 5: /sys/kernel/debug/tracing/trace_marker
    fd 6: /dev/null
    fd 7: /dev/null
    fd 8: /dev/null
    fd 9: /dev/binder
    fd 10: /system/framework/core-oj.jar
    fd 11: /system/framework/core-libart.jar
    fd 12: /system/framework/conscrypt.jar
    fd 13: /system/framework/okhttp.jar
    fd 14: /system/framework/legacy-test.jar
    fd 15: /system/framework/bouncycastle.jar
    fd 16: /system/framework/ext.jar
    fd 17: /system/framework/framework.jar
    fd 18: /system/framework/telephony-common.jar
    ……
```

- 这里的文件实际是各个包、文件、套接字节点等，由文件描述符表示。

1.2.10. log信息

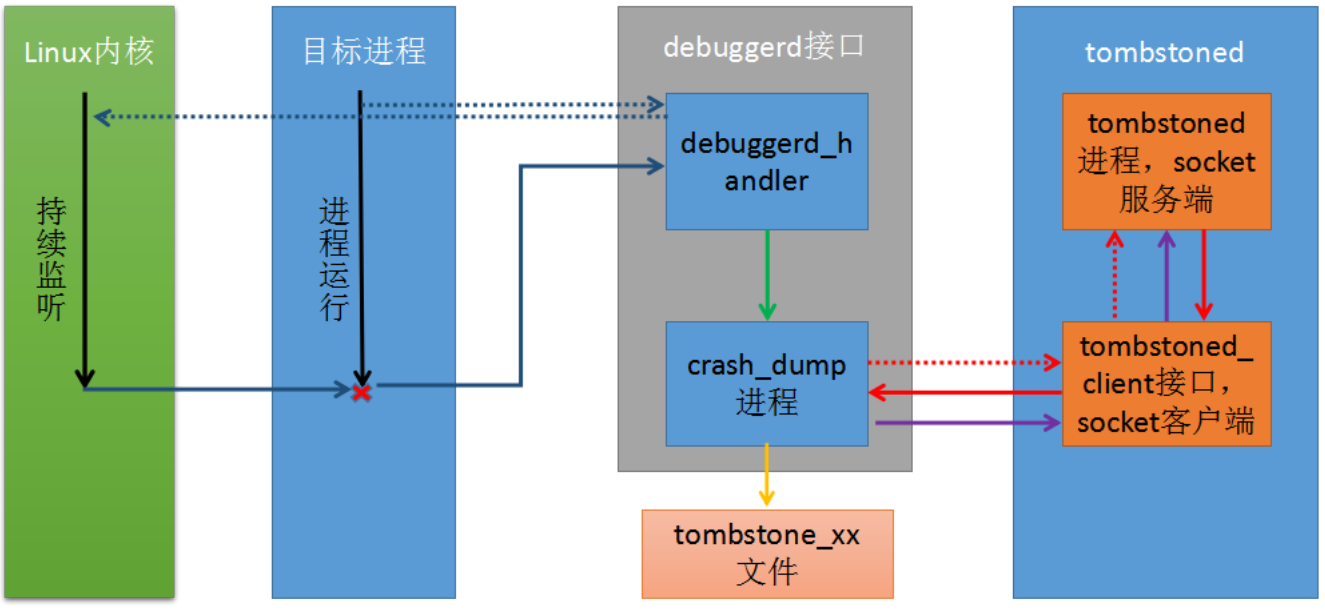
```
 --------- log system
03-12 03:52:03.981 28277 28352 I PowerManagerService: Going to sleep due to power button (uid 1000)...
03-12 03:52:04.483 28277 28303 I DisplayPowerController: Blocking screen off
03-12 03:52:04.498 28277 28303 I DreamManagerService: Entering dreamland.
03-12 03:52:04.499 28277 28303 I PowerManagerService: Dozing...
03-12 03:52:04.499 28277 28299 I DreamController: Starting dream: name=ComponentInfo{com.android.systemui/com.android.sys
03-12 03:52:04.522 28277 28303 I DisplayPowerController: Unblocked screen off after 39 ms
03-12 03:52:04.599 28277 28301 I DisplayManagerService: Display device changed state: "内置屏幕", OFF
……
--------- log main
03-12 03:52:01.291 28277 28584 E LocSvc_ApiV02: I/<--- globalEventCb line 86 QMI_LOC_EVENT_GNSS_SV_INFO_IND_V02
03-12 03:52:01.292 28277 28554 E LocSvc_afw: I/<=== sv_status_cb - line 1071 29
03-12 03:52:01.292 28277 28554 E LocSvc_eng_nmea: I/<=== nmea_cb line 62 0x7b396d8300
03-12 03:52:01.293 28277 28554 I chatty  : uid=1000(system) Thread-5 identical 3 lines
03-12 03:52:01.293 28277 28554 E LocSvc_eng_nmea: I/<=== nmea_cb line 62 0x7b396d8300
03-12 03:52:01.293 28277 28584 E LocSvc_ApiV02: I/<--- globalEventCb line 86 QMI_LOC_EVENT_POSITION_REPORT_IND_V02
03-12 03:52:01.294 28277 28554 E LocSvc_afw: I/<=== location_cb - from line 1058 2
03-12 03:52:01.294 28277 28554 E LocSvc_eng_nmea: I/<=== nmea_cb line 62 0x7b396d82d0
……
```

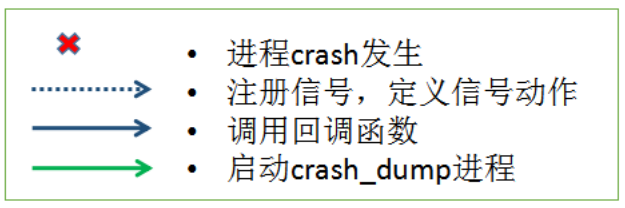
- log信息为从system和main缓存区读出，系统设置了开关来控制是否输出log到tombstone中，如果开关被设置为关闭，则进程崩溃后的产生的tombstone文件将没有这一部分。

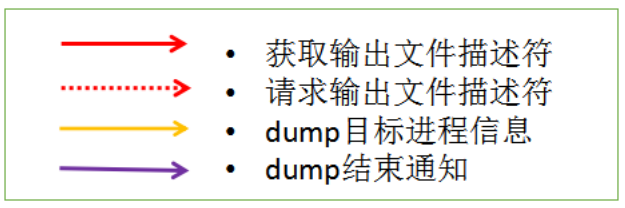
- 此外一个墓碑文件中还包括进程相关的各个子线程，子线程信息包括上述从<2>到<7>几个部分。

1.3. tombstone产生机制

- linux kernel有自己的一套signal机制，在检测到系统一些应用进程出现异常时，可以向这些进程发送一些异常信号，通知其做出相应的处理。
- Android8.1中，应用程序在开始运行时，会通过linker调用debuggerd_init注册一些关键的信号，并且传入信号处理的回调函数为debuggerd_signal_handler()，也就是说当应用程序拦截到注册的那些信号就会调用这个函数进行处理。
- 在实际进程运行过程中，linux内核监测到这个进程造成的某个异常后，便会发送信号给该进程，进程拦截到这个信号后开始调用debuggerd_signal_handler()函数，然后在该函数中调用crash_dump进程 来dump出相关进程、堆栈等信息到tombstone文件，保存在/data/tombstone目录下。基本流程如下图所示：







- 在user版本的系统中，墓碑文件的个数限制在10个以内，以tombstone_xx命名，当文件数量已经饱和后，新来的crash信息会覆盖写入到最早那个tombstone文件中。
- 参考资料：
  - 1、Android debuggerd 源码分析
  - 2、调查崩溃转储
  - 3、Android Native/Tombstone Crash Log 详细分析

2. tombstone文件分析

2.1. 分析技巧

2.1.1. 信号分析

- 应用程序在开始运行时，注册的信号包括以下这些：

信号量	VALUE	备注
SIGABRT	6	调用abort()函数时产生该信号，很多C库发现异常会调用这个函数
SIGBUS	7	非法访问内存地址，不存在的物理地址，与硬件和系统都相关
SIGFPE	8	在发生致命的运算错误时发出，不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误
SIGILL	4	CPU检测到某进程执行了非法指令，如损坏的可执行文件或代码区损坏均可导致
SIGSEGV	11	指示进程进行了无效内存访问，空指针、访问内核区、写只读空间、野指针、数组越界等导致
SIGSTKFLT	16	协处理器栈异常
SIGSYS	31	无效的系统调用
SIGTRAP	5	该信号由断点指令或其他trap指令产生
DEBUGGER_SIGNAL	35	debuggerd里面单独定义的信号（__SIGRTMIN + 3）,用于debuggerd 指令操作

- 这些信号，除了最后一个，都代表了系统内核检测到的异常类型。在信号信息这一栏，还有信号对应的错误码，定义在源码 /prebuilts/gcc/linux-x86/host/x86_64-linux-glibc2.11-4.8/sysroot/usr/include/bits/siginfo.h文件中，可以根据错误码进一步分析信号的产生原因。

2.1.2. 回溯分析

- 在回溯信息中，第一列是帧号；第三列是PC值(共享库中的偏移地址)；下一列是映射区域的名称（通常是共享库或可执行文件）；最后的括号中的内容显示的是与PC值对应的符号（一般提示了函数名称,如"start_thread"），以及与该符号相对应的偏移量。

```
   #38 pc 000000000001eb94  /system/lib64/libc.so (__start_thread+68)
 帧号 |     PC值(偏移地址)         |       映射区          |    （符号+偏移量）
```

- 利用分析工具，可以通过回溯信息准确定位代码中crash的具体位置。

2.1.3. log分析

- 写入到tombstone文件中log信息包括main和system缓存区，开发者可以根据最后打印出来的log，结合代码中打印该log的位置，分析出大致的crash的位置。也可以依据这些crash发生前的log，分析进程运行中的其他调试信息。

- 如果系统属性里设置将写入tombstone文件中的log开关关闭，而默认的tombstone文件又没有打印时间，则无法从tombstone文件中获取系统崩溃时间点的log。这个时候需要使用logcat -b crash指令从crash缓存区里读出tombstone对应的log，以确定tombstone的具体产生时间，然后再打印其他log缓存区找到对应时间的log获取更多信息。

2.2. 分析工具

2.2.1. addr2line

- addr2line 是用来获得指定动态链接库文件或者可执行文件中指定地址对应的源代码信息的工具,该工具的用法可以直接输入“addr2line -h”查询。一般linux系统会自带这个工具，可以直接使用。此外在安卓源码路径 prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.9/bin下也可以找到该工具对应的可执行文件arm-linux-androideabi-addr2line，如果系统不自带的话，将此文件添加到环境变量中也可以使用。
- 使用前，先找到代码编译输出路径下的映射对象，如“out/target/product/XXX/symbols/system/lib64/libc.so"；需要注意的是打开的是XXX/symbols/system目录，而不是信息XXX/system/目录，因为XXX/system/目录的动态库并不包含调试符号信息。
- 基本的调用方法是**“addr2line -e [动态库名称] -a [偏移地址]”**，以上面的#34帧为例，在out/target/product/XXX/symbols/system/lib64/目录下执行addr2line指令：

```
 raokaiyou@raokaiyou:lib$ addr2line -e libc.so -a 000000000001eb94
0x0001eb94
bionic/libc/bionic/mbrtoc16.cpp:63
```

- 可以定位到bionic/libc/bionic/mbrtoc16.cpp文件中的63行。
- 当调用该方法定位代码行号出现“??:?或??:0”时，说明此时的so文件没有附加符号表信息，这个与非安卓环境下代码编译的设置相关。

2.2.2. objdump

- 反汇编工具，先通过动态库的得到汇编代码，然后结合偏移地址来定位错误的原因。反汇编用法如下:

```
objdump -S $(objfile) > $(output_file)
```

2.2.3. stack.py

- stack.py工具相当于addr2line工具的升级版，可以直接将一个tombstone文件中的所有回溯（backtrace）信息一起定位出来。该工具实际就是一个python编写的可执行文件，可以参考下面的参考资料1内容编写。按照参考资料1中的使用方法如下：

```
  python  stack.py  --symbols-dir=out/target/profuct/XXX/sysbols/      tombstone_00(tombstone文件)
```

2.2.4. ndk-stack

- Android NDK自r6版本之后更新的一个自动分析工具，可以将能将崩溃时的调用内存地址和 c++ 代码一行一行对应 起来，使用方法：

```
ndk-stack -sym xxx.so -dump logfile
```

- 这里的so文件即上面的映射对象，logfile指的log文件。当不添加-dump选项时，这个工具可以从终端获取log信息，因此可以在调试android系统源码的时候直接分析log中的crash信息，用法如下：

```
adb shell logcat | ndk-stack -sym out/debug/target/product/XXXX/symbols/system/lib/xxx.so
```

- 参考资料：
  - 1、Android Tombstone分析
  - 2、linux信号分析
  - 3、Android平台Native代码的崩溃捕获机制
  - 4、Android Tombstone/Crash的log分析和定位

3. tombstone文件生成

3.1. crash进程捕获信号并处理

3.1.1. 信号注册

- 应用程序运行开始时会通过linker调用_linker_init做一些连接操作，这里面就有通过debuggerd接口注册信号一项，代码实现在_linker_init_post_relocation函数中路径（android-8.1.0_r41/bionic/linker/linker_main.cpp），如下：

```
 static ElfW(Addr) __linker_init_post_relocation(KernelArgumentBlock& args) {
……
#ifdef __ANDROID__
  debuggerd_callbacks_t callbacks = {
    .get_abort_message = []() {
      return g_abort_message;
    },
    .post_dump = &notify_gdb_of_libraries,
  };
  debuggerd_init(&callbacks);
#endif
```

- debuggerd_init的实现则是在android-8.1.0_r41/system/core/debuggerd/handler/debuggerd_handler.cpp文件中，里面调用debuggerd_register_handlers()进行注册并定义了信号的回调函数为debuggerd_signal_handler(),代码实现如下：

```
 void debuggerd_init(debuggerd_callbacks_t* callbacks) {
  struct sigaction action;
  memset(&action, 0, sizeof(action));
  sigfillset(&action.sa_mask);
  action.sa_sigaction = debuggerd_signal_handler;
  action.sa_flags = SA_RESTART | SA_SIGINFO;

  // Use the alternate signal stack if available so we can catch stack overflows.
  action.sa_flags |= SA_ONSTACK;
  debuggerd_register_handlers(&action);
}
```

- 使用debuggerd_register_handlers注册的信号量可以在对应的头文件hander中，并在这里定义了一个特殊信号量DEBUGGER_SIGNAL，用于debuggerd指令获取回溯信息，代码实现如下：

```
 #define DEBUGGER_SIGNAL (__SIGRTMIN + 3)
static void __attribute__((__unused__)) debuggerd_register_handlers(struct sigaction* action) {
  sigaction(SIGABRT, action, nullptr);
  sigaction(SIGBUS, action, nullptr);
  sigaction(SIGFPE, action, nullptr);
  sigaction(SIGILL, action, nullptr);
  sigaction(SIGSEGV, action, nullptr);
#if defined(SIGSTKFLT)
  sigaction(SIGSTKFLT, action, nullptr);
#endif
```

```
  sigaction(SIGSYS, action, nullptr);
  sigaction(SIGTRAP, action, nullptr);
  sigaction(DEBUGGER_SIGNAL, action, nullptr);
```

3.1.2. 信号捕获处理

- 当目标进程出现crash之后，内核监听到异常给进程发送对应的信号，进程拦截到信号会调用debuggerd_signal_handler()函数进行处理。这个函数会首先分析信号量，然后clone一个crash_dump的子进程，并传递进去目标进程的相关信息，之后再等待子进程去处理dump信息操作；当等待dump操作结束后，恢复信号的默认操作，一般会终止crash掉的目标进程。代码实现如下：

```
 static void debuggerd_signal_handler(int signal_number, siginfo_t* info, void* context) {
……
log_signal_summary(signal_number, info);
……
  pid_t child_pid =
    clone(debuggerd_dispatch_pseudothread, pseudothread_stack,
          CLONE_THREAD | CLONE_SIGHAND | CLONE_VM | CLONE_CHILD_SETTID | CLONE_CHILD_CLEARTID,
          &thread_info, nullptr, nullptr, &thread_info.pseudothread_tid);

  futex_wait(&thread_info.pseudothread_tid, -1);

  // and then wait for it to finish.
  futex_wait(&thread_info.pseudothread_tid, child_pid);
……
  if (signal_number != DEBUGGER_SIGNAL) {
    signal(signal_number, SIG_DFL);
  }
……
```

3.1.3. 启动crash_dump进程

- 由于crash_dump进程并开机启动的，在clone的时候实际调用了debuggerd_dispatch_pseudothread()函数来中启动crash_dump进程，在这个函数里则是调用execl()直接运行“/system/bin/crash_dump32”或者“/system/bin/crash_dump64”启动crash_dump进程。代码实现如下：

```
 static int debuggerd_dispatch_pseudothread(void* arg) {
……
       execl(CRASH_DUMP_PATH, CRASH_DUMP_NAME, main_tid, pseudothread_tid, debuggerd_dump_type,
          nullptr);
……
```

3.2. crash_dump进程dump信息到文件

3.2.1. 连接tombstoned进程

- crash_dump进程启动后，在其main函数里解析一下输入的参数，获取目标进程的相关信息，然后调用/android-8.1.0_r41/system/core/debuggerd/tombstoned/tombstoned_client.cpp文件中的接口tombstoned_connect()函数向tombstoned进程请求输出文件描述符。

3.2.2. attach到目标进程

- carsh_dump进程dump目标进程信息前，需要转换为目标进程的父进程才能有机会获取目标进程的核心image和寄存器的值。这里通过ptrace_seize_thread()函数调用ptrace()函数来attach到目标进程，并在dump结束后解除对子进程的追踪，代码如下：

```
 static bool ptrace_seize_thread(int pid_proc_fd, pid_t tid, std::string* error) {
  if (ptrace(PTRACE_SEIZE, tid, 0, 0) != 0) {
    *error = StringPrintf("failed to attach to thread %d: %s", tid, strerror(errno));
    return false;
  }

  // Make sure that the task we attached to is actually part of the pid we're dumping.
  if (!pid_contains_tid(pid_proc_fd, tid)) {
    if (ptrace(PTRACE_DETACH, tid, 0, 0) != 0) {
      PLOG(FATAL) << "failed to detach from thread " << tid;
    }
    *error = StringPrintf("thread %d is not in process", tid);
    return false;
  }

  // Put the task into ptrace-stop state.
  if (ptrace(PTRACE_INTERRUPT, tid, 0, 0) != 0) {
    PLOG(FATAL) << "failed to interrupt thread " << tid;
  }

  return true;
}
```

3.2.3. dump信息到文件

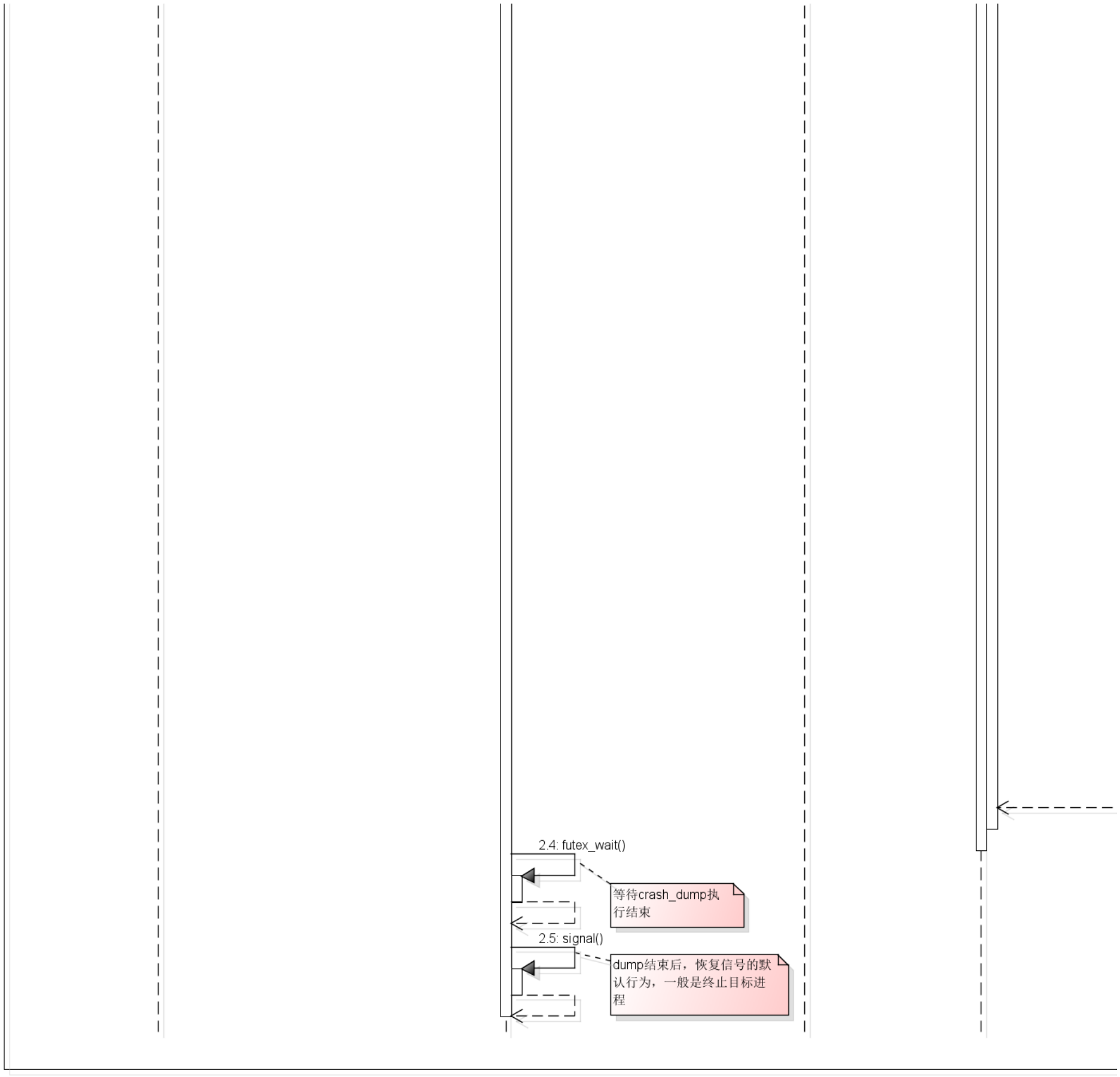
- dump信息的操作在/android-8.1.0_r41/system/core/debuggerd/libdebuggerd/tombstone.cpp文件中实现，接口是engrave_tombstone()函数。engrave_tombstone接口内部会继续调用dump_crash()，并在dump_crash()这个函数里依次调用各个子函数dump出tombstone文件的不同组成部分，代码如下：

```
 static void dump_crash(log_t* log, BacktraceMap* map, const OpenFilesList* open_files, pid_t pid,
                       pid_t tid, const std::string& process_name,
                       const std::map<pid_t, std::string>& threads, uintptr_t abort_msg_address) {
……
  _LOG(log, logtype::HEADER,
       "*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***\n");
  dump_header_info(log);
  dump_thread(log, pid, tid, process_name, threads.find(tid)->second, map, abort_msg_address, true);
  if (want_logs) {
    dump_logs(log, pid, 5);
  }
  for (const auto& it : threads) {
    pid_t thread_tid = it.first;
    const std::string& thread_name = it.second;

    if (thread_tid != tid) {
      dump_thread(log, pid, thread_tid, process_name, thread_name, map, 0, false);
    }
  }
  if (open_files) {
    _LOG(log, logtype::OPEN_FILES, "\nopen files:\n");
    dump_open_files_list_to_log(*open_files, log, "    ");
  }
  if (want_logs) {
    dump_logs(log, pid, 0);
  }
}
```

- 从信号注册到dump目标进程crash信息到tombstone文件的时序图如下：


sd crash_dump_1
linker_main
debuggerd_handler
handler
crash_dump
1: __linker_init()
1.1: __linker_cannot_link()
1.1.1: __linker_init_post_relocation()
1.1.1.1: debuggerd_init()
注册监听信号，传入信号回调函数debuggerd_signal_handler()
1.1.1.1.1: debuggerd_register_handlers()
2: debuggerd_signal_handler()
拦截到相关的信号
2.1: log_signal_summary()
2.2: debuggerd_dispatch_pseudothread()
2.2.1: execl()
2.2.1.1: main()
2.2.1.1.1: tombst
2.2.1.1.2: ptrace()
2.2.1.1.3: wait_for_si
2.2.1.1.3.1:
2.3: clone()



3.3. tombstoned进程管理输出文件

3.3.1. 注册监听对象

- tombstoned进程是开机启动的，负责管理输出文件。在进程关联的.rc文件已经创建了三个套接字，如下：

```
    socket tombstoned_crash seqpacket 0666 system system
    socket tombstoned_intercept seqpacket 0666 system system
    socket tombstoned_java_trace seqpacket 0666 system system
```

- 其中tombstoned_crash用于native层进程请求获取输出文件描述符，tombstoned_intercept用于debuggerd_client传递写管道描述符，tombstoned_java_trace用于signal_catcher线程获取文件描述符（例如，使用“kill -3 pid”则可以拿到/data/anr/trace_xx文件描述符，如果是ANR进程发送SIGQUIT信号则会拿到写管道描述符并最终输出到/data/anr/anr_xx文件）。

- tombstoned进程在启动后，会针对上面三个套接字创建监听对象。其中，对tombstoned_intercept套接字会先创建一个InterceptManager对象去单独管理，并由这个对象调用evconnlistener_new()接口创建监听对象，注册回调函数intercept_accept_cb()；对于另外两个套接字，则直接调用evconnlistener_new()接口创建监听对象，注册回调函数crash_accept_cb()。代码实现如下：

```
int main(int, char* []) {
……
  intercept_manager = new InterceptManager(base, intercept_socket);

  evconnlistener* tombstone_listener = evconnlistener_new(
      base, crash_accept_cb, CrashQueue::for_tombstones(), -1, LEV_OPT_CLOSE_ON_FREE, crash_socket);
……
    evconnlistener* java_trace_listener = evconnlistener_new(
        base, crash_accept_cb, CrashQueue::for_anrs(), -1, LEV_OPT_CLOSE_ON_FREE, java_trace_socket);
……
```

3.3.2. 处理写文件请求

- 当crash_dump进程通过socket的客户端请求获取输出文件描述符时，响应的是上面注册回调函数crash_accept_cb()。在这个函数中会创建一个Crash对象用于管理请求（如果有多个请求，则会创建出多个对象加入到队列中等待处理），然后调用crash_request_cb()处理请求。 crash_request_cb()函数中读取socket获取具体的请求信息，然后调用perform_request()获取输出文件路径。

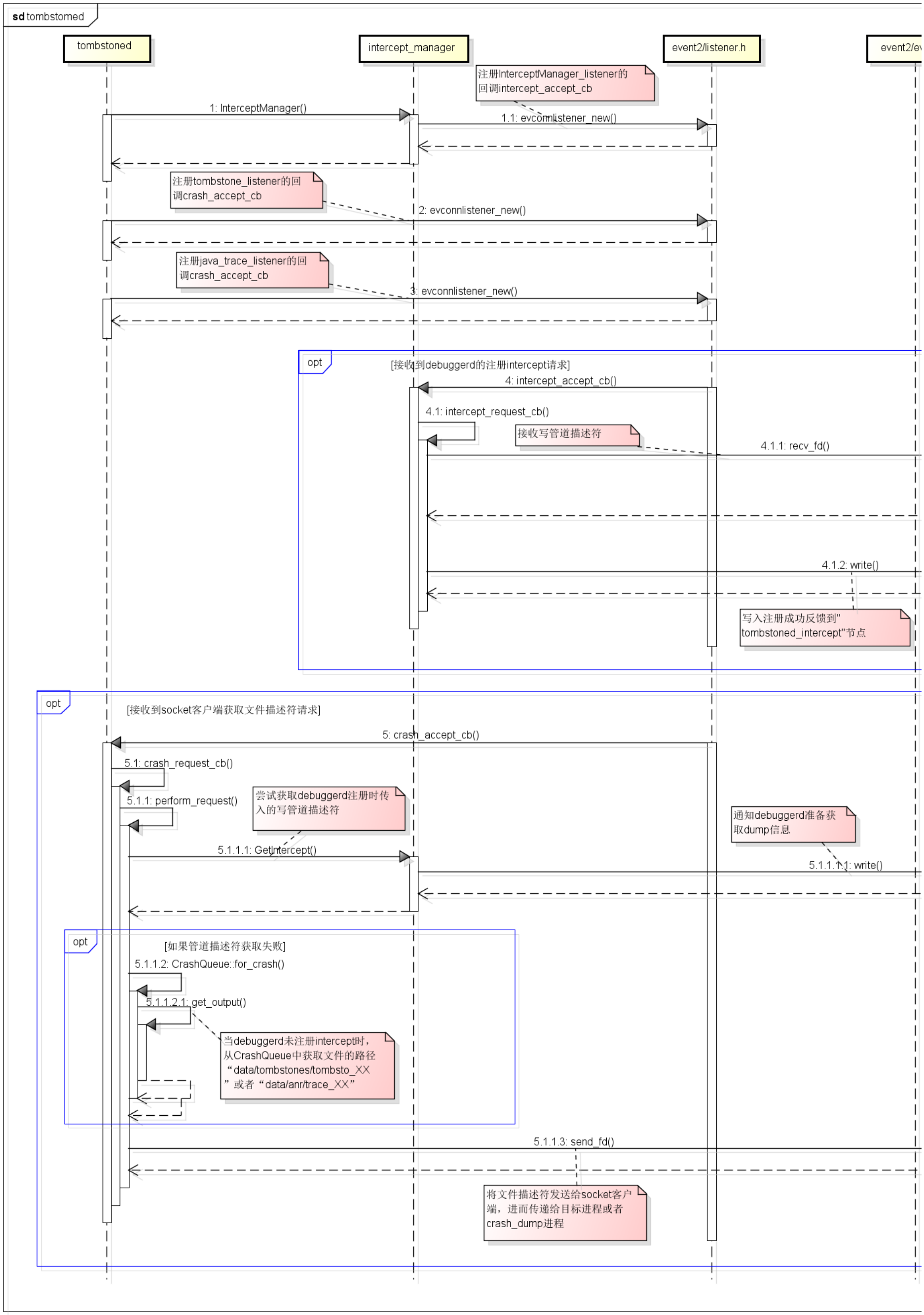
- perform_request函数中先调用GetIntercept()来判断是否在InterceptManager对象中存在了写管道描述符（说明本次请求时debuggerd发起的，目的是将目标进程的backtrace或者完整tombstone信息通过管道打印到终端），如果存在则将这个写管道描述符发送给crash_dump进程，如果不存在则继续调用CrashQueue来获取输出文件描述符。CrashQueue里面会判断请求类型，来决定输出路径是“/data/tombstones/tombstone_XX”或者“/data/anr/trace_XX”。之后再将输出文描述符通过send_fd()发送给crash_dump进程。代码处理如下：

```
static void perform_request(Crash* crash) {
  unique_fd output_fd;
```

```
  if (!intercept_manager->GetIntercept(crash->crash_pid, crash->crash_type, &output_fd)) {
    std::tie(output_fd, crash->crash_path) = CrashQueue::for_crash(crash)->get_output();
  }
……
  ssize_t rc = send_fd(crash->crash_fd, &response, sizeof(response), std::move(output_fd));
 ……
```

3.3.3. 管理请求队列

- 如果有多个进同时请求，则Crash对象会存在多个，CrashQueue会根据设定决定是否同时提供输出文件描述符。目前默认的操作是tombstone一次只能由一个进程处理，trace最多可以同时提供给四个进程。也就是说只有当前一个进程写tombstone_xx文件完毕后，后一个进程才能开始写tombstone_xx+1。
- tombstoned进程管理输出文件时序图如下：



4. debuggerd进程分析

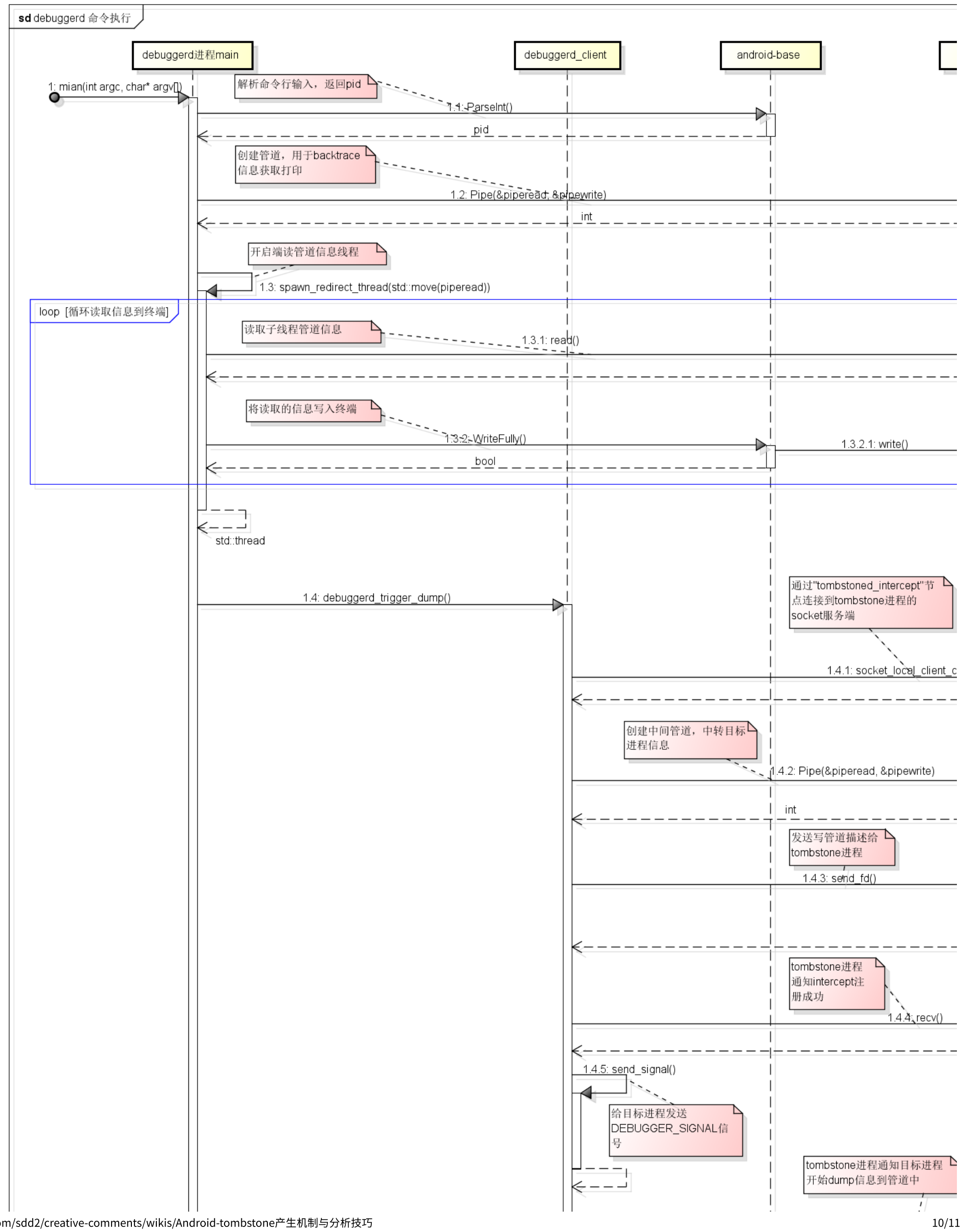
4.1. debuggerd 指令

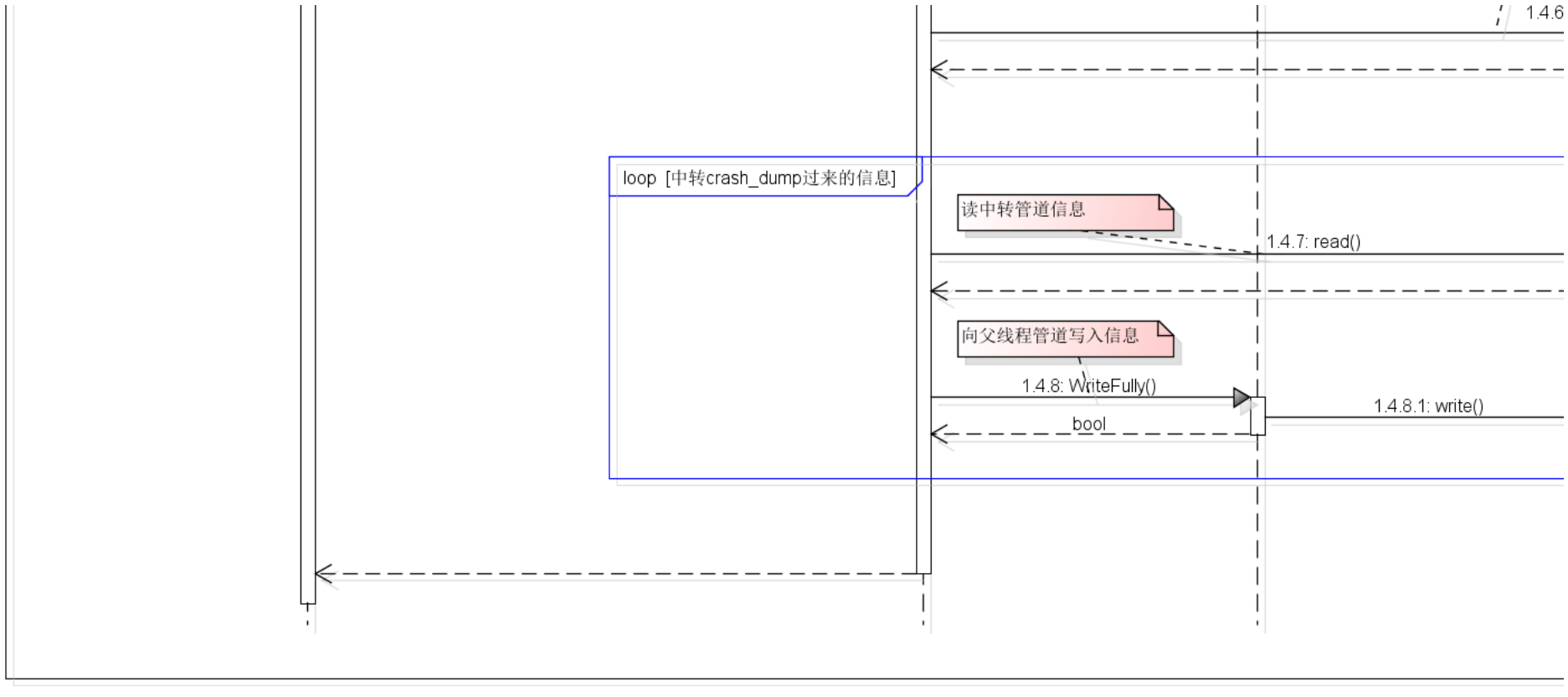
- 自安卓8.0之后，系统不再常驻debuggerd守护进程，目前的debuggerd进程由终端输入“debuggerd -b < tid>” 或者“debuggerd < tid>” 指令启动，前者仅用来输出指定进程的backtrace，后者用来输出完整tombstone信息。
- 当debuggerd指令输入后，系统启动debuggerd进程，在/android-8.1.0_r41/system/core/debuggerd/debuggerd.cpp文件中main()函数里解析指令参数。进程会首先创建一个管道，调用spawn_redirect_thread()函数创建一个线程循环输出backtrace或者完整tombstone信息到终端上。然后调用debuggerd_client的接口debuggerd_trigger_dump()函数去处理目标进程的信息写入操作。实现代码如下：

```
int main(int argc, char* argv[]) {
……
  unique_fd piperead, pipewrite;
  if (!Pipe(&piperead, &pipewrite)) {
    err(1, "failed to create pipe");
  }
  std::thread redirect_thread = spawn_redirect_thread(std::move(piperead));
  if (!debuggerd_trigger_dump(pid, backtrace_only ? kDebuggerdNativeBacktrace : kDebuggerdTombstone,
                              0, std::move(pipewrite))) {
    redirect_thread.join();
    errx(1, "failed to dump process %d", pid);
  }
  redirect_thread.join();
  return 0;
}
```

- debuggerd_trigger_dump函数中，首先通过"tombstoned_intercept"节点连接到tombstoned进程的socket服务端，之后创建中转管道、发送中转管道的写管道描述符给tombstoned进程（InterceptManager对象管理），等待intercept注册成功通知后向目标进程发送DEBUGGER_SIGNAL信号（前面分析可知该信号已经由目标进程注册），目标进程拦截到信号后调用debuggerd_signal_handler处理dump的一系列操作，最后的dump的信会通过中级管道写入到父线程管道（之后spawn_redirect_thread函数里的输出终端的操作开始进行）。

- debuggerd 指令的执行时序图如下：




1.4.6
loop [中转crash_dump过来的信息]
读中转管道信息
1.4.7: read()
向父线程管道写入信息
1.4.8: WriteFully()
1.4.8.1: write()
bool