干货：ANR日志分析全面解析

vivo互联网技术 发布于 2021-06-08

English

一、概述

解决ANR一直是Android 开发者需要掌握的重要技巧，一般从三个方面着手。

> 开发阶段：通过工具检查各个方法的耗时，卡顿情况，发现一处修改一处。
>
> 线上阶段：这个阶段主要依靠监控工具发现ANR并上报，比如matrix。
>
> 分析阶段：如果线上用户发生ANR，并且你获取了一份日志，这就涉及了本文要分享的内容——ANR日志分析技巧。

二、ANR产生机制

网上通俗的一段面试答题

> ANR——应用无响应，Activity是5秒，BroadCastReceiver是10秒，Service是20秒。

这句话说的很笼统，要想深入分析定位ANR，需要知道更多知识点，一般来说，ANR按产生机制，分为4类：

2.1 输入事件超时(5s)

InputEvent Timeout

a.InputDispatcher发送key事件给 对应的进程的 Focused Window ，对应的window不存在、处于暂

b.InputDispatcher发送MotionEvent事件有个例外之处：当对应Touched Window的 input waitQu

c.下一个事件到达，发现有一个超时事件才会触发ANR

2.2 广播类型超时（前台15s，后台60s）

BroadcastReceiver Timeout

a.静态注册的广播和有序广播会ANR，动态注册的非有序广播并不会ANR

b.广播发送时，会判断该进程是否存在，不存在则创建，创建进程的耗时也算在超时时间里

c.只有当进程存在前台显示的Activity才会弹出ANR对话框，否则会直接杀掉当前进程

d.当onReceive执行超过阈值（前台15s，后台60s），将产生ANR

e.如何发送前台广播：Intent.addFlags(Intent.FLAG_RECEIVER_FOREGROUND)

2.3 服务超时（前台20s，后台200s）

Service Timeout

a.Service的以下方法都会触发ANR：onCreate(),onStartCommand(), onStart(), onBind(), o
onDestroy().

b.前台Service超时时间为20s，后台Service超时时间为200s

c.如何区分前台、后台执行———当前APP处于用户态，此时执行的Service则为前台执行。

d.用户态：有前台activity、有前台广播在执行、有foreground service执行

2.4 ContentProvider 类型

a.ContentProvider创建发布超时并不会ANR

b.使用ContentProviderclient来访问ContentProverder可以自主选择触发ANR，超时时间自己定
client.setDetectNotResponding(PROVIDER_ANR_TIMEOUT);

ps：Activity生命周期超时会不会ANR？——经测试并不会。

```
override fun onCreate(savedInstanceState: Bundle?) {
        Thread.sleep(60000)
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
```

三、导致ANR的原因

很多开发者认为，那就是耗时操作导致ANR，全部是app应用层的问题。实际上，线上环境大部分ANR由系统原因导致。

3.1 应用层导致ANR（耗时操作）

a. 函数阻塞：如死循环、主线程IO、处理大数据

b. 锁出错：主线程等待子线程的锁

c. 内存紧张：系统分配给一个应用的内存是有上限的，长期处于内存紧张，会导致频繁内存交换，进而导致

3.2 系统导致ANR

a. CPU被抢占：一般来说，前台在玩游戏，可能会导致你的后台广播被抢占CPU

b. 系统服务无法及时响应：比如获取系统联系人等，系统的服务都是Binder机制，服务能力也是有限的，有

c. 其他应用占用的大量内存

四、分析日志

发生ANR的时候，系统会产生一份anr日志文件（手机的/data/anr 目录下，文件名称可能各厂商不一样，业内大多称呼为trace文件），内含如下几项重要信息。

4.1 CPU 负载

```
Load: 2.62 / 2.55 / 2.25
CPU usage from 0ms to 1987ms later (2020-03-10 08:31:55.169 to 2020-03-10 08:32:
  41% 2080/system_server: 28% user + 12% kernel / faults: 76445 minor 180 major
  26% 9378/com.xiaomi.store: 20% user + 6.8% kernel / faults: 68408 minor 68 maj
.........省略N行.....
66% TOTAL: 20% user + 15% kernel + 28% iowait + 0.7% irq + 0.7% softirq
```

如上所示：

- 第一行：1、5、15 分钟内正在使用和等待使用CPU 的活动进程的平均数
- 第二行：表明负载信息抓取在ANR发生之后的0~1987ms。同时也指明了ANR的时间点：2020-03-10 08:31:55.169
- 中间部分：各个进程占用的CPU的详细情况

- 最后一行：各个进程合计占用的CPU信息。

名词解释：

a. user:用户态,kernel:内核态

b. faults:内存缺页，minor—轻微的，major—重度，需要从磁盘拿数据

c. iowait:IO使用（等待）占比

d. irq:硬中断，softirq:软中断

注意：

- iowait占比很高，意味着有很大可能，是io耗时导致ANR，具体进一步查看有没有进程faults major比较多。
- 单进程CPU的负载并不是以100%为上限，而是有几个核，就有百分之几百，如4核上限为400%。

4.2 内存信息

```
Total number of allocations 476778    进程创建到现在一共创建了多少对象

Total bytes allocated 52MB  进程创建到现在一共申请了多少内存

Total bytes freed 52MB      进程创建到现在一共释放了多少内存

Free memory 777KB       不扩展堆的情况下可用的内存

Free memory until GC 777KB    GC前的可用内存

Free memory until OOME 383MB    OOM之前的可用内存

Total memory 当前总内存（已用+可用）

Max memory 384MB  进程最多能申请的内存
```

从含义可以得出结论：**Free memory until OOME** 的值很小的时候，已经处于内存紧张状态。应用可能是占用了过多内存。

另外，除了trace文件中有内存信息，普通的eventlog日志中，也有内存信息（不一定打印）

```
04-02 22:00:08.195  1531  1544 I am_meminfo: [350937088,41086976,492830720,42793
```

以上四个值分别指的是：

- Cached
- Free,
- Zram,
- Kernel,Native

Cached+Free的内存代表着当前整个手机的可用内存，如果值很小，意味着处于内存紧张状态。一般低内存的判定阈值为：4G 内存手机以下阀值：350MB，以上阀值则为：450MB

ps:如果ANR时间点前后，日志里有打印onTrimMemory，也可以作为内存紧张的一个参考判断

4.3 堆栈消息

堆栈信息是最重要的一个信息，展示了ANR发生的进程当前所有线程的状态。

```
suspend all histogram:  Sum: 2.834s 99% C.I. 5.738us-7145.919us Avg: 607.155us M
DALVIK THREADS (248):
"main" prio=5 tid=1 Native
  | group="main" sCount=1 dsCount=0 flags=1 obj=0x74b17080 self=0x7bb7a14c00
  | sysTid=2080 nice=-2 cgrp=default sched=0/0 handle=0x7c3e82b548
  | state=S schedstat=( 757205342094 583547320723 2145008 ) utm=52002 stm=23718
  | stack=0x7fdc995000-0x7fdc997000 stackSize=8MB
  | held mutexes=
  kernel: __switch_to+0xb0/0xbc
  kernel: SyS_epoll_wait+0x288/0x364
  kernel: SyS_epoll_pwait+0xb0/0x124
  kernel: cpu_switch_to+0x38c/0x2258
  native: #00 pc 000000000007cd8c  /system/lib64/libc.so (__epoll_pwait+8)
  native: #01 pc 0000000000014d48  /system/lib64/libutils.so (android::Looper::p
  native: #02 pc 0000000000014c18  /system/lib64/libutils.so (android::Looper::p
  native: #03 pc 0000000000127474  /system/lib64/libandroid_runtime.so (android:
  at android.os.MessageQueue.nativePollOnce(Native method)
  at android.os.MessageQueue.next(MessageQueue.java:330)
  at android.os.Looper.loop(Looper.java:169)
  at com.android.server.SystemServer.run(SystemServer.java:508)
  at com.android.server.SystemServer.main(SystemServer.java:340)
  at java.lang.reflect.Method.invoke(Native method)
  at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.jav
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:856)

  .........省略N行.....
```

如上日志所示，本文截图了两个线程信息，一个是主线程main，它的状态是native。

另一个是OkHttp ConnectionPool，它的状态是TimeWaiting。众所周知，教科书上说线程状态有5种：**新建、就绪、执行、阻塞、死亡**。而Java中的线程状态有6种，6种状态都定义在：java.lang.Thread.State中

状态	解释
NEW	线程刚被创建，但是并未启动。还没调用start方法。
RUNNABLE	线程可以在java虚拟机中运行的状态，可能正在运行自己代码，也可能没有，这取决于操作系统处理器。
BLOCKED	当一个线程试图获取一个对象锁，而该对象锁被其他的线程持有，则该线程进入Blocked状态；当该线程持有锁时，该线程将变成Runnable状态。
WATING	一个线程在等待另一个线程执行一个（唤醒）动作时，该线程进入Waiting状态。进入这个状态后是不能自动唤醒的，必须等待另一个线程调用notify或者notifyAll方法才能够
TIMED_WAITING	同waiting状态，有超时参数，超时后自动唤醒，比如Thread.Sleep(1000)
TERMINATED	线程已经执行完毕

问题来了，上述main线程的native是什么状态，哪来的？其实trace文件中的状态是是CPP代码中定义的状态，下面是一张对应关系表。

java thread 状态	cpp thread状态	说明
TERMINATED	ZOMBIE	线程死亡，终止运行
RUNNABLE	RUNNING/RUNNABLE	线程可运行或正在运行
TIMED_WAITING	TIMED_WAIT	执行了带有超时参数的wait、sleep或join函数
BLOCKED	MONITOR	线程阻塞，等待获取对象锁
WAITING	WAIT	执行了无超时参数的wait函数
NEW	INITIALIZING	新建，正在初始化，为其分配资源
NEW	STARTING	新建，正在启动
RUNNABLE	NATIVE	正在执行JNI本地函数
WAITING	VMWAIT	正在等待VM资源
RUNNABLE	SUSPENDED	线程暂停，通常是由于GC或debug被暂停
	UNKNOWN	未知状态

由此可知，main函数的native状态是正在执行JNI函数。堆栈信息是我们分析ANR的第一个重要的信息，一般来说：

> main线程处于 BLOCK、WAITING、TIMEWAITING状态，那基本上是函数阻塞导致ANR；
>
> 如果main线程无异常，则应该排查CPU负载和内存环境。

五、典型案例分析

5.1 主线程无卡顿，处于正常状态堆栈

```
"main" prio=5 tid=1 Native
  | group="main" sCount=1 dsCount=0 flags=1 obj=0x74b38080 self=0x7ad9014c00
  | sysTid=23081 nice=0 cgrp=default sched=0/0 handle=0x7b5fdc5548
  | state=S schedstat=( 284838633 166738594 505 ) utm=21 stm=7 core=1 HZ=100
  | stack=0x7fc95da000-0x7fc95dc000 stackSize=8MB
  | held mutexes=
  kernel: __switch_to+0xb0/0xbc
  kernel: SyS_epoll_wait+0x288/0x364
  kernel: SyS_epoll_pwait+0xb0/0x124
  kernel: cpu_switch_to+0x38c/0x2258
  native: #00 pc 000000000007cd8c  /system/lib64/libc.so (__epoll_pwait+8)
  native: #01 pc 0000000000014d48  /system/lib64/libutils.so (android::Looper::p
  native: #02 pc 0000000000014c18  /system/lib64/libutils.so (android::Looper::p
  native: #03 pc 00000000001275f4  /system/lib64/libandroid_runtime.so (android:
  at android.os.MessageQueue.nativePollOnce(Native method)
  at android.os.MessageQueue.next(MessageQueue.java:330)
  at android.os.Looper.loop(Looper.java:169)
  at android.app.ActivityThread.main(ActivityThread.java:7073)
  at java.lang.reflect.Method.invoke(Native method)
  at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.jav
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:876)
```

上述主线程堆栈就是一个很正常的空闲堆栈，表明主线程正在等待新的消息。

如果ANR日志里主线程是这样一个状态，那可能有两个原因：

> 该ANR是CPU抢占或内存紧张等其他因素引起
>
> 这份ANR日志抓取的时候，主线程已经恢复正常

遇到这种空闲堆栈，可以按照第3节的方法去分析CPU、内存的情况。其次可以关注抓取日志的时间和ANR发生的时间是否相隔过久，时间过久这个堆栈就没有分析意义了。

5.2 主线程执行耗时操作

```
"main" prio=5 tid=1 Runnable
  | group="main" sCount=0 dsCount=0 flags=0 obj=0x72deb848 self=0x7748c10800
  | sysTid=8968 nice=-10 cgrp=default sched=0/0 handle=0x77cfa75ed0
  | state=R schedstat=( 24783612979 48520902 756 ) utm=2473 stm=5 core=5 HZ=100
  | stack=0x7fce68b000-0x7fce68d000 stackSize=8192KB
  | held mutexes= "mutator lock"(shared held)
  at com.example.test.MainActivity$onCreate$2.onClick(MainActivity.kt:20)——关键行
  at android.view.View.performClick(View.java:7187)
  at android.view.View.performClickInternal(View.java:7164)
  at android.view.View.access$3500(View.java:813)
  at android.view.View$PerformClick.run(View.java:27640)
  at android.os.Handler.handleCallback(Handler.java:883)
  at android.os.Handler.dispatchMessage(Handler.java:100)
  at android.os.Looper.loop(Looper.java:230)
  at android.app.ActivityThread.main(ActivityThread.java:7725)
  at java.lang.reflect.Method.invoke(Native method)
  at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.jav
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1034)
```

上述日志表明，主线程正处于执行状态，看堆栈信息可知不是处于空闲状态，发生ANR是因为一处click监听函数里执行了耗时操作。

5.3 主线程被锁阻塞

```
  | sysTid=22858 nice=-10 cgrp=default sched=0/0 handle=0x77cfa75ed0
  | state=S schedstat=( 390366023 28399376 279 ) utm=34 stm=5 core=1 HZ=100
  | stack=0x7fce68b000-0x7fce68d000 stackSize=8192KB
  | held mutexes=
  at com.example.test.MainActivity$onCreate$1.onClick(MainActivity.kt:15)
  - waiting to lock <0x01aed1da> (a java.lang.Object) held by thread 3 ————关键
  at android.view.View.performClick(View.java:7187)
  at android.view.View.performClickInternal(View.java:7164)
  at android.view.View.access$3500(View.java:813)
  at android.view.View$PerformClick.run(View.java:27640)
  at android.os.Handler.handleCallback(Handler.java:883)
  at android.os.Handler.dispatchMessage(Handler.java:100)
  at android.os.Looper.loop(Looper.java:230)
  at android.app.ActivityThread.main(ActivityThread.java:7725)
  at java.lang.reflect.Method.invoke(Native method)
  at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.jav
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1034)

  .........省略N行.....

  "WQW TEST" prio=5 tid=3 TimeWating
```

```
    | group="main" sCount=1 dsCount=0 flags=1 obj=0x12c44230 self=0x772f0ec000
    | sysTid=22938 nice=0 cgrp=default sched=0/0 handle=0x77391fbd50
    | state=S schedstat=( 274896 0 1 ) utm=0 stm=0 core=1 HZ=100
    | stack=0x77390f9000-0x77390fb000 stackSize=1039KB
    | held mutexes=
    at java.lang.Thread.sleep(Native method)
```

这是一个典型的主线程被锁阻塞的例子；

```
  waiting to lock <0x01aed1da> (a java.lang.Object) held by thread 3
```

其中等待的锁是<0x01aed1da>，这个锁的持有者是线程 3。进一步搜索 “tid=3” 找到线程3， 发现它正在TimeWating。

那么ANR的原因找到了：线程3持有了一把锁，并且自身长时间不释放，主线程等待这把锁发生超时。在线上环境中，常见因锁而ANR的场景是SharePreference写入。

5.4 CPU被抢占

```
  CPU usage from 0ms to 10625ms later (2020-03-09 14:38:31.633 to 2020-03-09 14:38
    543% 2045/com.alibaba.android.rimet: 54% user + 89% kernel / faults: 4608 minc
    99% 674/android.hardware.camera.provider@2.4-service: 81% user + 18% kernel /
    24% 32589/com.wang.test: 22% user + 1.4% kernel / faults: 7432 minor 1 major
    .........省略N行.....
```

如上日志，第二行是钉钉的进程，占据CPU高达543%，抢占了大部分CPU资源，因而导致发生ANR。

5.5 内存紧张导致ANR

如果有一份日志，CPU和堆栈都很正常（不贴出来了），仍旧发生ANR，考虑是内存紧张。

从CPU第一行信息可以发现，ANR的时间点是2020-10-31 22:38:58.468—CPU usage from 0ms to 21752ms later (2020-10-31 22:38:58.468 to 2020-10-31 22:39:20.220)

接着去系统日志里搜索am_meminfo， 这个没有搜索到。再次搜索onTrimMemory，果然发现了很多条记录；

```
  10-31 22:37:19.749 20733 20733 E Runtime : onTrimMemory level:80,pid:com.xxx.xx
  10-31 22:37:33.458 20733 20733 E Runtime : onTrimMemory level:80,pid:com.xxx.xx
  10-31 22:38:00.153 20733 20733 E Runtime : onTrimMemory level:80,pid:com.xxx.xx
```

```
10-31 22:38:58.731 20733 20733 E Runtime : onTrimMemory level:80,pid:com.xxx.xx
10-31 22:39:02.816 20733 20733 E Runtime : onTrimMemory level:80,pid:com.xxx.xx
```

可以看出，在发生ANR的时间点前后，内存都处于紧张状态，level等级是80，查看Android API文档；

```
/**
 * Level for {@link #onTrimMemory(int)}: the process is nearing the end
 * of the background LRU list, and if more memory isn't found soon it will
 * be killed.
 */
static final int TRIM_MEMORY_COMPLETE = 80;
```

可知80这个等级是很严重的，应用马上就要被杀死，被杀死的这个应用从名字可以看出来是桌面，连桌面都快要被杀死，那普通应用能好到哪里去呢?

一般来说，发生内存紧张，会导致多个应用发生ANR，所以在日志中如果发现有多个应用一起ANR了，可以初步判定，此ANR与你的应用无关。

5.6 系统服务超时导致ANR

系统服务超时一般会包含BinderProxy.transactNative关键字，请看如下日志：

```
"main" prio=5 tid=1 Native
  | group="main" sCount=1 dsCount=0 flags=1 obj=0x727851e8 self=0x78d7060e00
  | sysTid=4894 nice=0 cgrp=default sched=0/0 handle=0x795cc1e9a8
  | state=S schedstat=( 8292806752 1621087524 7167 ) utm=707 stm=122 core=5 HZ=1
  | stack=0x7febb64000-0x7febb66000 stackSize=8MB
  | held mutexes=
  kernel: __switch_to+0x90/0xc4
  kernel: binder_thread_read+0xbd8/0x144c
  kernel: binder_ioctl_write_read.constprop.58+0x20c/0x348
  kernel: binder_ioctl+0x5d4/0x88c
  kernel: do_vfs_ioctl+0xb8/0xb1c
  kernel: SyS_ioctl+0x84/0x98
  kernel: cpu_switch_to+0x34c/0x22c0
  native: #00 pc 000000000007a2ac  /system/lib64/libc.so (__ioctl+4)
  native: #01 pc 00000000000276ec  /system/lib64/libc.so (ioctl+132)
  native: #02 pc 00000000000557d4  /system/lib64/libbinder.so (android::IPCThrea
  native: #03 pc 0000000000056494  /system/lib64/libbinder.so (android::IPCThrea
  native: #04 pc 00000000000562d0  /system/lib64/libbinder.so (android::IPCThrea
  native: #05 pc 000000000004ce1c  /system/lib64/libbinder.so (android::BpBinder
  native: #06 pc 00000000001281c8  /system/lib64/libandroid_runtime.so (???)
  native: #07 pc 0000000000947ed4  /system/framework/arm64/boot-framework.oat (J
```

```
    at android.os.BinderProxy.transactNative(Native method) ————————————————关键行!
    at android.os.BinderProxy.transact(Binder.java:804)
    at android.net.IConnectivityManager$Stub$Proxy.getActiveNetworkInfo(IConnectiv
    at android.net.ConnectivityManager.getActiveNetworkInfo(ConnectivityManager.ja
```

从堆栈可以看出获取网络信息发生了ANR：getActiveNetworkInfo。

前文有讲过：系统的服务都是Binder机制（16个线程），服务能力也是有限的，有可能系统服务长时间不响应导致ANR。如果其他应用占用了所有Binder线程，那么当前应用只能等待。

可进一步搜索：blockUntilThreadAvailable关键字：

```
at android.os.Binder.blockUntilThreadAvailable(Native method)
```

如果有发现某个线程的堆栈，包含此字样，可进一步看其堆栈，确定是调用了什么系统服务。此类ANR也是属于系统环境的问题，如果某类型机器上频繁发生此问题，应用层可以考虑规避策略。

六、结语

本文总结的技巧来自笔者工作中的大量ANR日志分析经验，如有错漏请留言指出，交流促使进步！

> 作者：vivo互联网客户端团队—Wang Qinwei