android ANR机制

Last edited by **caoquanli** 1 month ago

Android ANR机制

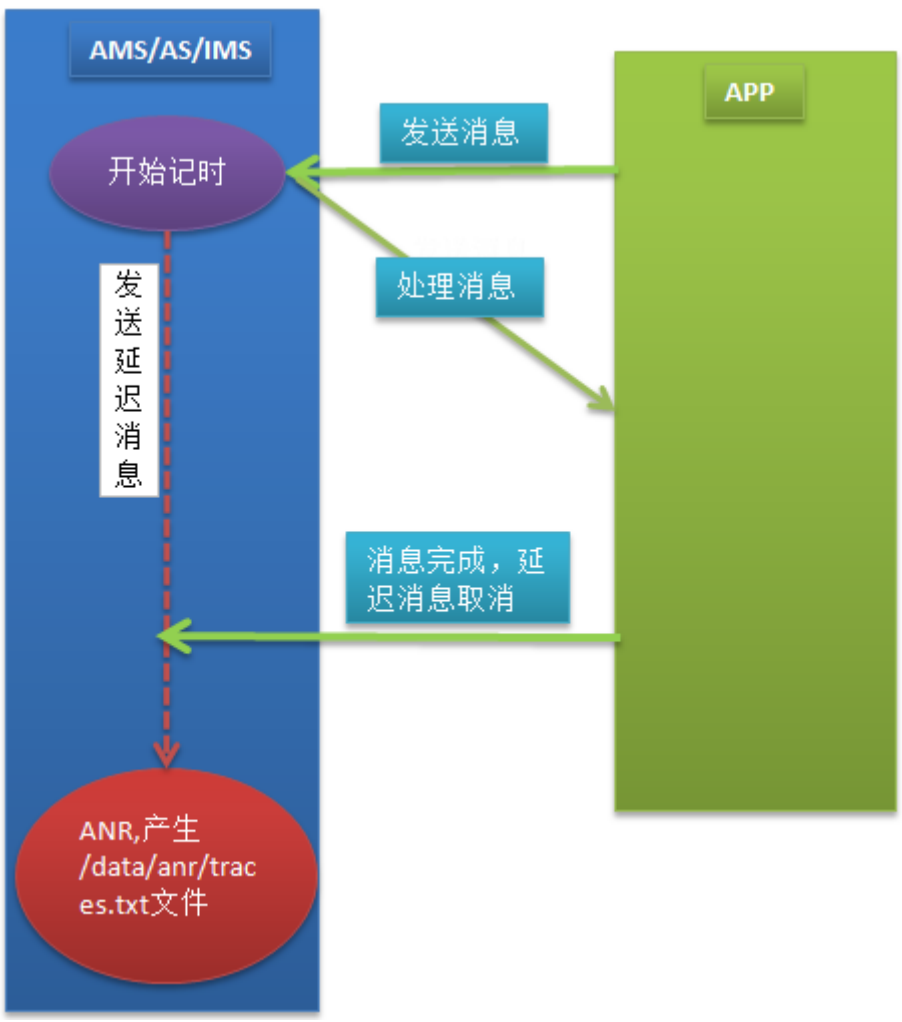
1.概述

ANR全称是Application Not Responding，就是应用程序无响应，Android系统对于主线程处理的事情必须在一定的时间范围内完成，如果超过预定时间能未能得到有效响应或者响应时间过长，都会造成ANR。一般地，这时往往会弹出一个提示框，告知用户当前xxx未响应，用户可选择继续等待或者强制关闭,并在机器的/data/anr的目录下面产生一个**traces.txt文件（anr_xxxx的文件）**.

2.ANR触发原因

- 主线程执行耗时的操作,在规定的时间内没有完成（比如执行请求网络访问，进行耗时的IO操作,数据库的读写, 系统资源已耗尽, 下载东西，死锁，等待其他线程的释放锁等）

3.ANR检测机制



4.ANR的信息处理流程

进程只要发生ANR，最终都会调用**AppErrors.appNotResponding**方法，此方法就是ANR的处理流程，分析此方法你将会更好的理解ANR。在这里不在对代码进行分析，当发生ANR时，此方法会按顺序依次执行：
1.输出ANR原因信息到EventLog,也就是说ANR触发的时间点最接近的就是事件日志中输出的am_anr信息;

2.记录ANR输出到main log中,我们会在main log中搜索ANR in 可以查看到那个package，activity，PID，Reason，Parent ，CPU信息 等。可以根据输出当前各个进程的CPU情况使用以及CPU负载情况，判断进程是否繁忙不繁忙，IO是否阻塞等。

3.会将CPU使用情况和进程trace文件信息，保存到/data/system/dropbox；产生的trace.txt文件，也会保存到/data/anr/中，根据trace文件，可以看线程的状态等 。

4.根据进程类型，决定直接后台杀掉，还是弹出提示框框告知用户。

除了主体逻辑外，发生ANR时还会输出各种类别的日志：

- event log:通过检索"am_anr"关键字，可以找到发生ANR的应用,时间点；
- main log:通过检索"ANR in"关键字，可以找到ANR的信息，日志的上下文会包含CPU的使用情况。
- dropbox:通过检索"anr"关键字，可以找到ANR的信息
- trace：发生ANR时，各进程的函数调用栈信息

5. ANR的种类

- **service timeout:** 比如前台服务在20s内未执行完成.
- **BroadcastQueue Timeout:** 比如前台广播在10s内未执行完成.
- **ContentProvider timeout:** 比如内容提供者，在publish超时10s.
- **InputDispatching Timeout:** 比如输入事件分发超时5s，包括按键和触摸事件.

5.1 Service ANR监测机制

服务超时是位于“ActivityManager”线程中的AMS.MainHandler收到

SERVICE_TIMEOUT_MSG或者 SERVICE_FOREGROUND_TIMEOUT_MSG消息时触发。

对于Service发生ANR条件有三个：

- 对于前台服务，则超时为SERVICE_TIMEOUT = 20s;
- 对于后台服务，则超时为SERVICE_BACKGROUND_TIMEOUT = 200s
- 在调用startForegroundService()创建一个前台服务,并且必须立即（在5秒内）调用该服务的 startForeground 的方法，否则会出现ANR。 SERVICE_START_FOREGROUND_TIMEOUT = 5*1000; (android O的新特性)

Service发生ANR的场景只有一个:

就是在Service启动过程中，主线程阻塞。不管是前台服务启动，还是后台服务启动。

Service发生ANR时出现的一些Log:

```
Slog.w(TAG, "Timeout executing service: " + timeout);
Slog.i(TAG, "During shutdown skipping ANR: " + app + " " + annotation);
Slog.i(TAG, "Skipping duplicate ANR: " + app + " " + annotation);
Slog.i(TAG, "Crashing app skipping ANR: " + app + " " + annotation);
Slog.i(TAG, "App already killed by AM skipping ANR: " + app + " " + annotation);
Slog.i(TAG, "Skipping died app ANR: " + app + " " + annotation);
Log.i(TAG, "Skipped " + skippedFrames + " frames!  "+ "The application may be doing too muc
Slog.i(TAG, "Service foreground-required timeout for " + r);
```

注：上面的log在service ANR的时候不一定都会出现。

5.2 Broadcast ANR监测机制

BroadcastReceiver Timeout是位于”ActivityManager”线程中的BroadcastQueue.BroadcastHandler收到BROADCAST_TIMEOUT_MSG消息时触发，广播的ANR只发生在有序广播中，不会发生在并发广播中. 我们发送的普通广播，静态注册的广播接收器是按有序广播来接收的，动态注册的广播接收器是按并发接收的。我们发送有序广播，动态注册和静态注册都是按有序广播的来接收的。

对于广播队列有两个条件会发生ANR:

- 有序广播的总执行时间超过 2*广播接收者个数 * timeout时长，则会触发anr;
- 有序广播的某一个广播接收者执行过程超过 timeout时长，则会触发anr;
  time out时长：
  对于前台广播，则超时为BROADCAST_FG_TIMEOUT = 10s；
  对于后台广播，则超时为BROADCAST_BG_TIMEOUT = 60s;

BroadcastQueue发生ANR的场景只有一个:

就是在广播发送的过程中，主线程执行了耗时的操作

Broadcast发生ANR时出现的一些Log:

```
Slog.w(TAG, "Timeout of broadcast " + r + " - receiver=" + r. receiver + ", started " + (no
Slog.w(TAG, "Receiver during timeout of " + r + " : " + curReceiver);
Slog.w(TAG, "pending app  [" + mQueueName + "]" + mPendingBroadcast.curApp+ " died before r
Slog.w(TAG, "Hung broadcast ["+ mQueueName + "] discarded after timeout failure:" + " now="
```

注：上面的log不一定都会出现。

5.3 ContentProvider ANR检测机制

ContentProvider超时是由于“ActivityManager”线程中的AMS.MainHandler收到

CONTENT_PROVIDER_PUBLISH_TIMEOUT_MSG消息时触发。

对于ContentProvider有一个条件会发生ANR:

ContentProvider在执行public时超时10S。
CONTENT_PROVIDER_PUBLISH_TIMEOUT = 10s;

ContentProvider发生ANR的场景只有一个:

ContentProvider的publish在10s内没进行完。

ContentProvider发生ANR时出现的一些Log:

```
Slog.d(TAG_PROCESSES,
            "Force removing proc " + app.toShortString() + " (" + name + "/" + uid + ")");
Log.w(TAG, "Detected provider not responding: " + mContentProvider);
```

5.4 Input ANR监测机制

5.4.1 焦点窗口的获取

InputDispatch在派发过程中会使用mFocusedWindowHandle作为目标窗口，这mFocusedWindowHandle变量表示系统处于焦点状态的窗口，焦点窗口的设置是从wms windowstate中获取input窗体的属性，详细获取焦点窗口流程如下：

5.4.2 Input超时处理

在派发事件时，dispatchKeyLocked()和dispatchMotionLocked()需要先找到当前的焦点窗口，焦点窗口是事件最终被派发的地方，在寻找窗口的过程中就会判断是否已经发生了ANR,以Key事件为例阐述，调用过程如下：

在dispatchKeyLocked()函数中调用findFocusedWindowTargetsLocked()寻找聚焦窗口，寻找聚焦窗口失败的情况：

- **无窗口，无应用：** 这种情况不会导致ANR的发生，因为直接将事件丢弃，没有机会调用handleTargetNotReadyLocked()函数
- **无窗口，有应用：** 这种情况是有焦点应用，但是该应用还没有启动完成，焦点窗口还没出现，需要继续等待直到焦点窗口完成初始化

在找到窗口以后会调用checkWindowReadyForMoreInputLocked() ，检查当前窗口是否有能力再接收新的输入事件，检测场景共有6种如下：

- **场景1:** 窗口处于paused状态，不能处理输入事件

  "Waiting because the %s window is paused"

- **场景2:** 窗口还未向InputDispatcher注册，无法将事件派发到窗口

  "Waiting because the %s window's input channel is not registered with the input dispatcher. The window may be in the process of being removed."

- **场景3:** 窗口和InputDispatcher的连接已经中断，即InputChannel不能正常工作

  "Waiting because the %s window's input connection is %s.The window may be in the process of being removed."

- **场景4:** InputChannel已经饱和，不能再处理新的事件

  "Waiting because the %s window's input channel is full. Outbound queue length: %d. Wait queue length: %d."

- **场景5:** 对于按键类型(KeyEvent)的输入事件，需要等待上一个事件处理完毕

  “Waiting to send key event because the [targetType] window has not finished processing all of the input events that were previously delivered to it. Outbound queue length: %d. Wait queue length: %d.”

- **场景6:** 对于触摸类型(TouchEvent)的输入事件，可以立即派发到当前的窗口，因为TouchEvent都是发生在用户当前可见的窗口。但有一种情况， 如果当前应用由于队列有太多的输入事件等待派发，导致发生了ANR，那TouchEvent事件就需要排队等待派发。

  "Waiting to send non-key event because the %s window has not finished processing certain input events that were delivered to it over %0.1fms ago. Wait queue length: %d. Wait queue head age: %0.1fms."

如果上述场景有一个发生了，则说明当前窗口有事件积压，输入事件需要继续等待，然后调用handleTargetsNotReadyLocked()来判断是否需要发送ANR

```
int32_t InputDispatcher::handleTargetsNotReadyLocked(nsecs_t currentTime,
        const EventEntry* entry,
        const sp<InputApplicationHandle>& applicationHandle,
        const sp<InputWindowHandle>& windowHandle,
        nsecs_t* nextWakeupTime, const char* reason) {
    ......
    if (currentTime >= mInputTargetWaitTimeoutTime) {
        onANRLocked(currentTime, applicationHandle, windowHandle,
                entry->eventTime, mInputTargetWaitStartTime, reason);

        *nextWakeupTime = LONG_LONG_MIN;
        return INPUT_EVENT_INJECTION_PENDING;
    } ......
}
```

- 如果当前事件派发已经超时，则说明已经检测到了ANR，调用onANRLocked()方法，将nextWakeupTime设置为最小值，马上开始下一轮的调度；
- 在onANRLocked()方法中，会保存ANR的一些状态信息，调用doNotifyANRLockedInterruptible(),进一步会调用到JNI层的NativeInputManager::NotifyANR()方法，它的主要功能就是衔接Native层和Java层，直接调用Java层的InputManagerService.NotifyANR()方法。

至此，ANR的处理逻辑转交到了Java层。底层(Native)发现一旦有输入事件派发超时，就会通知上层(Java)，上层收到ANR通知后，决定是否终止当前输入事件的派发。

发生ANR时，Java层最开始的入口是InputManagerService.notifyANR()，它是直接被Native层调用的。我们先把ANR的Java层调用关系列出来：

```
InputManagerService.notifyANR()
└── InputMonitor.notifyANR()
    ├── IApplicationToken.keyDispatchingTimedOut()
    │   └── ActivityRecord.keyDispatchingTimedOut()
    │       └── AMS.inputDispatchingTimedOut()
    │           └── AMS.appNotResponding()
    │
    └── AMS.inputDispatchingTimedOut()
        └── AMS.appNotResponding()
```

InputManagerService.notifyANR()只是为Native层定义了一个接口，它直接调用InputMonitor.notifyANR()。 如果该方法的返回值等于0,则放弃本次输入事件;如果大于0,则表示需要继续等待的时间。

```
public long notifyANR(InputApplicationHandle inputApplicationHandle,
      InputWindowHandle inputWindowHandle, String reason) {
    ...
    if (appWindowToken != null && appWindowToken.appToken != null) {
        // appToken实际上就是当前的ActivityRecord。
        // 如果发生ANR的Activity还存在，则直接通过ActivityRecord通知事件派发超时
        boolean abort = appWindowToken.appToken.keyDispatchingTimedOut(reason);
        if (! abort) {
            return appWindowToken.inputDispatchingTimeoutNanos;
        }
    } else if (windowState != null) {
        // 如果发生ANR的Activity已经销毁了，则通过AMS通知事件派发超时
        long timeout = ActivityManagerNative.getDefault().inputDispatchingTimedOut(
                        windowState.mSession.mPid, aboveSystem, reason);
         if (timeout >= 0) {
             return timeout;
         }
    }
    return 0; // abort dispatching
}
```

Input发生ANR时出现的一些Log:

```
Slog.i(TAG_WM, "Input event dispatching timed out "
                        + "sending to " + windowState.mAttrs.getTitle()
                        + ".  Reason: " + reason);
Slog.i(TAG_WM, "Input event dispatching timed out "
                        + "sending to application " + appWindowToken.stringName
                        + ".  Reason: " + reason);
Slog.i(TAG_WM, "Input event dispatching timed out "
                        + ".  Reason: " + reason);
```

6.trace文件解读（anr_xxxx文件）

在发生ANR的时候，系统会把各种线程的堆栈都dump到一个文件中，一般都会在机器的/data/anr/目录下面，详细解读一下trace.txt里面的一些字段，这里主要列出trace文件的重要信息。

```
//显示进程id、ANR发生时间点、ANR发生进程包名
----- pid 23531 at 2018-12-19 10:28:26 -----
//发生ANR的包名
Cmd line: com.demo.sunchao.anrtest
//设备的唯一标识符
Build fingerprint: 'google/taimen/taimen:8.0.0/OPD3.170816.012/4343094:userdebug/dev-keys'
ABI: 'arm64'
//构建类型
Build type: optimized
Zygote loaded classes=4656 post zygote classes=2
Intern table: 42625 strong; 137 weak
JNI: CheckJNI is on; globals=510 (plus 23 weak)
//运行的一些so库 通常可以忽略
Libraries: /system/lib64/libandroid.so /system/lib64/libcompiler_rt.so /system/lib64/libjav
Heap: 58% free, 350KB/846KB; 14143 objects
//一些GC等object信息，通常可以忽略
```

```
Dumping cumulative Gc timings
.....
....
DALVIK THREADS (11):
//main线程堆栈信息
"main" prio=5 tid=1 Sleeping
  | group="main" sCount=1 dsCount=0 flags=1 obj=0x73f85690 self=0x790b0bea00
  | sysTid=23531 nice=0 cgrp=default sched=0/0 handle=0x790fff79b0
  | state=S schedstat=( 97804268 22077708 91 ) utm=5 stm=3 core=0 HZ=100
  | stack=0x7fdab84000-0x7fdab86000 stackSize=8MB
  | held mutexes=
  at java.lang.Thread.sleep(Native method)
  - sleeping on <0x0908d8f7> (a java.lang.Object)
  at java.lang.Thread.sleep(Thread.java:373)
  - locked <0x0908d8f7> (a java.lang.Object)
  at java.lang.Thread.sleep(Thread.java:314)
  at android.os.SystemClock.sleep(SystemClock.java:122)
  at com.demo.sunchao.anrtest.MainService.onCreate(MainService.java:19)
  at android.app.ActivityThread.handleCreateService(ActivityThread.java:3420)
  at android.app.ActivityThread.-wrap4(ActivityThread.java:-1)
  at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1686)
  at android.os.Handler.dispatchMessage(Handler.java:105)
  at android.os.Looper.loop(Looper.java:164)
  at android.app.ActivityThread.main(ActivityThread.java:6600)
  at java.lang.reflect.Method.invoke(Native method)
  at com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:240)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:772)
```

main：main标识的主线程也就是UI线程，如果是其他线程的话，命名就是"Thread-X"格式，x表示线程id，逐步递增。
prio： 线程的优先级，默认是5.
tid：tid不是线程的id，是线程的唯一标识ID
group：是线程组的名字
sCount：该线程被挂起的次数
dsCount：该线程被调试器挂起的此时
obj：对象地址
systid：该线程号（主线程的线程号和进程号相同）
self:线程的native的地址
nice：线程的调用优先级
sched：分别标志了线程的调度策略和优先级
handle：线程处理这个消息(函数)的地址
state：调度状态
schedstat：从机器/pro/[pid]/task/[tid]/schedstat读出，三个值分别表示在CPU上执行的时间，线程的等待时间和线程执行的时间片长度，不支持这项信息的三个值是0
utm：是线程用户态下使用的时间值（单位是jiffes）
stm：是内核态下的调度时间
core：是最后执行这个线程的CPU核的序号

java线程的六种状态：

- **NEW：** 新建状态
- **RUNNABLE：** Java线程中将就绪（ready）和运行中（running）两种状态笼统的称为“运行”。当线程处于 Runnable 状态时，表示线程准备就绪，等待获取 CPU 资源
- **Block：** 阻塞（挂起）状态
- **WAITING：** 进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。
- **TIMED_WAITING：** 定时等待 该状态不同于WAITING，它可以在指定的时间后自行返回。
- **TERMINATED：** 表示该线程已经执行完毕。

java层线程状态图：　　java线程和虚拟机的线程对应关系，根据这个线程关系将有助于我们分析问题

```
//artThread.State               Java Thread.State   JDWP state
Terminated                      // TERMINATED     TS_ZOMBIE    Thread.run has returned,
Runnable,                       // RUNNABLE       TS_RUNNING   runnable
```

```
  TimedWaiting,                   // TIMED_WAITING  TS_WAIT      in Object.wait() with a t
  Sleeping,                       // TIMED_WAITING  TS_SLEEPING  in Thread.sleep()
  Blocked,                        // BLOCKED        TS_MONITOR   blocked on a monitor
  Waiting,                        // WAITING        TS_WAIT      in Object.wait()
  WaitingForGcToComplete,         // WAITING        TS_WAIT      blocked waiting for GC
  WaitingForCheckPointsToRun,     // WAITING        TS_WAIT      GC waiting for checkpoint
  WaitingPerformingGc,            // WAITING        TS_WAIT      performing GC
  WaitingForDebuggerSend,         // WAITING        TS_WAIT      blocked waiting for event
  WaitingForDebuggerToAttach,     // WAITING        TS_WAIT      blocked waiting for debug
  WaitingInMainDebuggerLoop,      // WAITING        TS_WAIT      blocking/reading/processi
  WaitingForDebuggerSuspension,   // WAITING        TS_WAIT      waiting for debugger susp
  WaitingForJniOnLoad,            // WAITING        TS_WAIT      waiting for execution of
  WaitingForSignalCatcherOutput,  // WAITING        TS_WAIT      waiting for signal catche
  WaitingInMainSignalCatcherLoop, // WAITING        TS_WAIT      blocking/reading/processi
  WaitingForDeoptimization,       // WAITING        TS_WAIT      waiting for deoptimizatio
  WaitingForMethodTracingStart,   // WAITING        TS_WAIT      waiting for method tracin
  WaitingForVisitObjects,         // WAITING        TS_WAIT      waiting for visiting obje
  WaitingForGetObjectsAllocated,  // WAITING        TS_WAIT      waiting for getting the n
  WaitingWeakGcRootRead,          // WAITING        TS_WAIT      waiting on the GC to read
  WaitingForGcThreadFlip,         // WAITING        TS_WAIT      waiting on the GC thread
  Starting,                       // NEW            TS_WAIT      native thread started, no
  Native,                         // RUNNABLE       TS_RUNNING   running in a JNI native m
  Suspended,                      // RUNNABLE       TS_RUNNING   suspended by GC or debugg
```

根据这个线程的对应关系我们会发现上面的trace文件的main线程的状态是Sleeping，对应的java线程是TIMED_WAITING。因此我们可以发现这个线程是被定时睡眠了。

7.如何分析ANR

因为目前没有碰到合适的案例，这边举例一个输入事件死锁导致ANR的简单例子 代码如下：

```
public class MainActivity extends AppCompatActivity  implements View.OnClickListener{

    private  Object lock1;
    private  Object lock2;

    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        lock1 = new Object();
        lock2 = new Object();
        Button button = (Button) findViewById(R.id.bt);
        button.setOnClickListener(this);
        textView = (TextView)findViewById(R.id.text);

    }

    @Override
    public void onClick(View view) {

        new Thread(new Runnable() {
            @Override
            public void run() {
                m1();
            }

        }).start();
           m2();
        textView.setText("Successs !!!");
    }

    void m1(){
        synchronized(lock1){
            for (int a = 0; a<100000; a++){
                a++;
                Log.d("sunchao1 lock1","a="+a);
            }
            synchronized(lock2){
                System.out.println("m1 Lock o2 second");
            }
```

```
        }
    }
    void m2(){
        synchronized(lock2){
            for (int a = 0; a<100000; a++){
                a++;
               Log.d("sunchao1 lock2","a="+a);
            }
            synchronized(lock1){
                System.out.println("m2 Lock o1 second");
            }
        }
    }


}
```

第一步首先查看ANR发生的时间点，log中最接近ANR发生时间点是events Log，查看你 events log，搜索am_anr：

```
12-20 16:32:36.169  1159  1198 I am_anr  : [0,3219,com.demo.sunchao.anrtest2,954777414,Inpu
```

以上我们可以发现ANR的时间点16:32:36.169，发生ANR的包com.demo.sunchao.anrtest2，进程号3219， ANR的原因是上面讲的Input 输入事件的场景6，等待发送非键事件，因为触摸的窗口尚未完成处理在500.0ms之前传送给它的某些输入事件。说明，上一个输入事件在5秒内没有完成，导致这一个输入事件处于等待状态，因此我们可以查看在发生ANR的5秒之前，究竟做了什么事。

第二步查看mian log：

```
12-20 16:32:39.780  1159  1198 E ActivityManager: ANR in com.demo.sunchao.anrtest2 (com.dem
12-20 16:32:39.780  1159  1198 E ActivityManager: PID: 3219
12-20 16:32:39.780  1159  1198 E ActivityManager: Reason: Input dispatching timed out (Wait
12-20 16:32:39.780  1159  1198 E ActivityManager: Load: 4.44 / 4.42 / 4.57
//ANR发生之前的十秒，CPU的使用状态
12-20 16:32:39.780  1159  1198 E ActivityManager: CPU usage from 10159ms to 0ms ago (2018-1
12-20 16:32:39.780  1159  1198 E ActivityManager:   20% 589/surfaceflinger: 8.8% user + 11%
12-20 16:32:39.780  1159  1198 E ActivityManager:   11% 9019/com.wandoujia.phoenix2:aid: 9.
12-20 16:32:39.780  1159  1198 E ActivityManager:   9.7% 8939/com.wandoujia.phoenix2: 7.3%
12-20 16:32:39.780  1159  1198 E ActivityManager:   7.8% 774/android.hardware.sensors@1.0-s
12-20 16:32:39.780  1159  1198 E ActivityManager:   5.9% 8223/com.google.android.googlequic
12-20 16:32:39.780  1159  1198 E ActivityManager:   5.4% 1159/system_server: 2.8% user + 2.
12-20 16:32:39.780  1159  1198 E ActivityManager:   2.7% 31613/kworker/u16:8: 0% user + 2.7
12-20 16:32:39.780  1159  1198 E ActivityManager:   2.5% 1504/com.android.systemui: 2% user
12-20 16:32:39.780  1159  1198 E ActivityManager:   2.4% 2101/kworker/u16:0: 0% user + 2.4%
12-20 16:32:39.780  1159  1198 E ActivityManager:   2.3% 31626/kworker/u16:23: 0% user + 2.
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.8% 87/smem_native_rpm: 0% user + 0.8%
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.7% 31475/kworker/3:1: 0% user + 0.7%
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.6% 32437/adbd: 0% user + 0.5% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.4% 619/irq/755-fts_tou: 0% user + 0.4
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.3% 3/ksoftirqd/0: 0% user + 0.3% kern
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.3% 9456/com.wandoujia.phoenix2:channe
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.2% 25/ksoftirqd/2: 0% user + 0.2% ker
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.2% 86/dsps_smem_glink: 0% user + 0.2%
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.2% 786/msm_irqbalance: 0% user + 0.1%
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.2% 31090/kworker/u16:11: 0% user + 0.
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.1% 10/rcuop/0: 0% user + 0.1% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.1% 17/ksoftirqd/1: 0% user + 0.1% ker
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.1% 562/sugov:0: 0% user + 0.1% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.1% 670/netd: 0% user + 0.1% kernel /
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.1% 1142/kworker/u17:1: 0% user + 0.1%
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.1% 2510/com.tencent.mobileqq: 0.1% us
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.1% 2639/kworker/3:0: 0% user + 0.1% k
12-20 16:32:39.780  1159  1198 E ActivityManager:   0.1% 31507/kworker/0:1: 0% user + 0.1%
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 7/rcu_preempt: 0% user + 0% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 11/rcuos/0: 0% user + 0% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 20/rcuop/1: 0% user + 0% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 33/ksoftirqd/3: 0% user + 0% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 57/ksoftirqd/6: 0% user + 0% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 85/smem_native_dsp: 0% user + 0% ker
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 568/logd: 0% user + 0% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 617/jbd2/sda13-8: 0% user + 0% kerne
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 788/sensors.qcom: 0% user + 0% kerne
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 824/thermal-engine: 0% user + 0% ker
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 1241/kworker/2:0: 0% user + 0% kerne
```

```
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 2368/perfd: 0% user + 0% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 2493/com.tencent.mobileqq:MSF: 0% us
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 3248/sh: 0% user + 0% kernel / fault
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 8154/com.google.android.gms.persiste
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 9285/com.mobogenie: 0% user + 0% ker
12-20 16:32:39.780  1159  1198 E ActivityManager:   0% 31813/kworker/1:6: 0% user + 0% kern
12-20 16:32:39.780  1159  1198 E ActivityManager: 7.9% TOTAL: 4% user + 3.2% kernel + 0% io
//发生ANR之后的CPU的使用状态
12-20 16:32:39.780  1159  1198 E ActivityManager: CPU usage from 57ms to 398ms later (2018-
12-20 16:32:39.780  1159  1198 E ActivityManager:   57% 1159/system_server: 32% user + 25%
12-20 16:32:39.780  1159  1198 E ActivityManager:     32% 1198/ActivityManager: 9.6% user +
12-20 16:32:39.780  1159  1198 E ActivityManager:     22% 1169/HeapTaskDaemon: 19% user + 3
12-20 16:32:39.780  1159  1198 E ActivityManager:     3.2% 1167/FinalizerDaemon: 0% user +
12-20 16:32:39.780  1159  1198 E ActivityManager:   15% 589/surfaceflinger: 9.1% user + 6%
12-20 16:32:39.780  1159  1198 E ActivityManager:     6% 659/EventThread: 3% user + 3% kern
12-20 16:32:39.780  1159  1198 E ActivityManager:     3% 615/DispSync: 3% user + 0% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:     3% 1798/Binder:589_4: 3% user + 0% ke
12-20 16:32:39.780  1159  1198 E ActivityManager:   10% 9019/com.wandoujia.phoenix2:aid: 6.
12-20 16:32:39.780  1159  1198 E ActivityManager:     6.8% 9019/ia.phoenix2:aid: 3.4% user
12-20 16:32:39.780  1159  1198 E ActivityManager:   6.1% 774/android.hardware.sensors@1.0-s
12-20 16:32:39.780  1159  1198 E ActivityManager:     3% 1308/HwBinder:774_1: 0% user + 3%
12-20 16:32:39.780  1159  1198 E ActivityManager:   6.7% 8223/com.google.android.googlequic
12-20 16:32:39.780  1159  1198 E ActivityManager:     3.3% 8223/earchbox:search: 3.3% user
12-20 16:32:39.780  1159  1198 E ActivityManager:     3.3% 8239/Binder:8223_1: 0% user + 3.
12-20 16:32:39.780  1159  1198 E ActivityManager:   2.8% 68/rcuop/7: 0% user + 2.8% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   2.9% 149/kswapd0: 0% user + 2.9% kernel
12-20 16:32:39.780  1159  1198 E ActivityManager:   3.3% 2331/irq/34-1008000.: 0% user + 3.
12-20 16:32:39.780  1159  1198 E ActivityManager:   3.4% 8516/com.goog
12-20 16:32:39.809  1708  1708 E QtiImsExtUtils: getConfigForPhoneId phoneId is invalid
12-20 16:32:39.810  1708  1708 E QtiImsExtUtils: isCarrierConfigEnabled bundle is null
```

查看CPU的使用状态，我们可以发现，CPU资源充足，IO正常，总共才使用了7.9%

第三步：查看traces文件

```
----- pid 3219 at 2018-12-20 16:32:36 -----
Cmd line: com.demo.sunchao.anrtest2
Build fingerprint: 'google/taimen/taimen:8.0.0/OPD3.170816.012/4343094:userdebug/dev-keys'
ABI: 'arm64'
Build type: optimized
Zygote loaded classes=4656 post zygote classes=248
Intern table: 43772 strong; 137 weak
......
......
"main" prio=5 tid=1 Blocked
  | group="main" sCount=1 dsCount=0 flags=1 obj=0x73f85690 self=0x72a9cbea00
  | sysTid=3219 nice=-10 cgrp=default sched=0/0 handle=0x72aec6e9b0
  | state=S schedstat=( 1167243465 19438121 434 ) utm=40 stm=75 core=4 HZ=100
  | stack=0x7fbfc60000-0x7fbfc62000 stackSize=8MB
  | held mutexes=
at com.demo.sunchao.anrtest.MainActivity.m2(MainActivity.java:70)
- waiting to lock <0x027badce> (a java.lang.Object) held by thread 15
- locked <0x08f1f4ef> (a java.lang.Object)
at com.demo.sunchao.anrtest.MainActivity.onClick(MainActivity.java:49)
at android.view.View.performClick(View.java:6254)
at android.view.View$PerformClick.run(View.java:24705)
at android.os.Handler.handleCallback(Handler.java:789)
at android.os.Handler.dispatchMessage(Handler.java:98)
at android.os.Looper.loop(Looper.java:164)
at android.app.ActivityThread.main(ActivityThread.java:6600)
at java.lang.reflect.Method.invoke(Native method)
at com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:240)
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:772)
.....
.....
"Thread-2" prio=5 tid=15 Blocked
| group="main" sCount=1 dsCount=0 flags=1 obj=0x12f40000 self=0x729ee80800
| sysTid=3259 nice=0 cgrp=default sched=0/0 handle=0x7292c054f0
| state=S schedstat=( 996840360 13138541 242 ) utm=29 stm=70 core=5 HZ=100
| stack=0x7292b03000-0x7292b05000 stackSize=1037KB
| held mutexes=
at com.demo.sunchao.anrtest.MainActivity.m1(MainActivity.java:59)
- waiting to lock <0x08f1f4ef> (a java.lang.Object) held by thread 1
- locked <0x027badce> (a java.lang.Object)
```

```
    at com.demo.sunchao.anrtest.MainActivity$1.run(MainActivity.java:45)
    at java.lang.Thread.run(Thread.java:764)
```

查看trace文件我们会发现main线程被Blocked，查看堆栈main线程持有了一个0x08f1f4ef（A锁）的锁，还在等待一个0x027badce（B锁）的锁，因此我们需要去查看哪个线程持有了B锁，没有释放出来。搜索B锁我们发现了Thread-2线程持有了一个B锁，并且还在等待A锁，因此我们可以发现，main线程刚好持有A锁，等待B锁，而Thread-2刚好持有B锁，在等待A锁，形成闭环，陷入死锁的无限循环。最后形成ANR。