ANR

ANR

概念

ANR(Application Not responding)，是指应用程序未响应，Android系统对于一些事件需要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

在 Android 里，应用程序的响应性是由 Activity Manager 和 WindowManager 系统服务监视的。当它监测到以下情况中的一个时，Android 就会针对特定的应用程序显示 ANR：

场景

- Service Timeout
- BroadcastQueue Timeout
- ContentProvider Timeout
- InputDispatching Timeout

Timeout时长

- 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
- 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;
- ContentProvider超时为CONTENT_PROVIDER_PUBLISH_TIMEOUT = 10s;
- InputDispatching Timeout: 输入事件分发超时5s，包括按键和触摸事件。

> 注意事项: Input的超时机制与其他的不同，对于input来说即便某次事件执行时间超过timeout时长，只要用户后续在没有再生成输入事件，则不会触发ANR

超时检测机制

1. Service超时检测机制：
    - 超过一定时间没有执行完相应操作来触发移除延时消息，则会触发anr;
2. BroadcastReceiver超时检测机制：
    - 有序广播的总执行时间超过 2* receiver个数 * timeout时长，则会触发anr;
      有序广播的某一个receiver执行过程超过 timeout时长，则会触发anr;
3. 另外:
    - 对于Service, Broadcast, Input发生ANR之后,最终都会调用AMS.appNotResponding;
    - 对于provider,在其进程启动时publish过程可能会出现ANR, 则会直接杀进程以及清理相应信息,而不会弹出ANR的对话框

如何避免 ANR?

考虑上面的 ANR 定义，让我们来研究一下为什么它会在 Android 应用程序里发生和如何最佳 构建应用程序来避免 ANR。

- Android 应用程序通常是运行在一个单独的线程（例如，main）里。这意味着你的应用程序所做的事情如果在主线程里占用了太长的时间的话，就会引发 ANR 对话框，因为你的应用程序并没有给自己机会来处理输入事件或者 Intent 广播。
- 因此，运行在主线程里的任何方法都尽可能少做事情。特别是，Activity 应该在它的关键生命周期方法（如 onCreate()和 onResume()）里尽可能少的去做创建操作。潜在的耗时操作，例如网络或数据库操作，或者高耗时的计算如改变位图尺寸，应该在子线程里（或者以数据 库操作为例，通过异步请求的方式）来完成。然而，不是说你的主线程阻塞在那里等待子线程的完成——也不是调用 Thread.wait()或是 Thread.sleep()。替代的方法是，主线程应该为子线程提供一个 Handler，以便完成时能够提交给主线程。以这种方式设计你的应用程序，将 能保证你的主线程保持对输入的响应性并能避免由于 5 秒输入事件的超时引发的 ANR 对话框。这种做法应该在其它显示 UI 的线程里效仿，因为它们都受相同的超 时影响。
- IntentReceiver 执行时间的特殊限制意味着它应该做：在后台里做小的、琐碎的工作如保存设定或者注册一个 Notification。和在主线 程里调用的其它方法一样，应用程序应该避免在 BroadcastReceiver 里做耗时的操作或计算。但不再是在子线程里做这些任务（因为 BroadcastReceiver 的生命周期短），替代的是，如果响应 Intent 广播需要执行一个耗时的动作的话，应用程序应该启动一个 Service。顺便提及一句，你也应该避免在IntentReceiver 里启动一个 Activity，因为它会创建一个新的画面，并从当前用户正在运行的程序上抢夺焦点。如果你的应用程序在响应 Intent 广播时需要向用户展示什么，你应该使用 Notification Manager 来实现。
- 增强响应灵敏性
  一般来说，在应用程序里，100 到 200ms 是用户能感知阻滞的时间阈值。因此，这里有一些额外的技巧来避免 ANR，并有助于让你的应用程序看起来有响应性。
  如果你的应用程序为响应用户输入正在后台工作的话，可以显示工作的进度（ProgressBar 和 ProgressDialog 对这种情况来说很有用）。
  特别是游戏，在子线程里做移动的计算。

如果你的应用程序有一个耗时的初始化过程的话，考虑可以显示一个 SplashScreen 或者快 速显示主画面并异步来填充这些信息。在这两种情况下，你都应该显示正在进行的进度，以 免用户认为应用程序被冻结了。

前台与后台ANR【了解下就行】

- 前台ANR：用户能感知，比如拥有前台可见的activity的进程，或者拥有前台通知的fg-service的进程，此时发生ANR对用户体验影响比较大，需要弹框让用户决定是否退出还是等待
- 后台ANR：，只抓取发生无响应进程的trace，也不会收集CPU信息，并且会在后台直接杀掉该无响应的进程，不会弹框提示用户

ANR分析

1. 前台ANR发生后，系统会马上去抓取现场的信息，用于调试分析，收集的信息如下:
   - 将am_anr信息输出到EventLog，也就是说ANR触发的时间点最接近的就是EventLog中输出的am_anr信息
   - 收集以下重要进程的各个线程调用栈trace信息，保存在data/anr/traces.txt文件
     - 当前发生ANR的进程，system_server进程以及所有persistent进程
     - audioserver, cameraserver, mediaserver, surfaceflinger等重要的native进程
     - CPU使用率排名前5的进程
   - 将发生ANR的reason以及CPU使用情况信息输出到main log

- 将traces文件和CPU使用情况信息保存到dropbox，即data/system/dropbox目录
- 对用户可感知的进程则弹出ANR对话框告知用户，对用户不可感知的进程发生ANR则直接杀掉

2. 分析步骤
    1. 定位发生ANR时间点
    2. 查看trace信息
    3. 分析是否有耗时的message,binder调用，锁的竞争，CPU资源的抢占
    4. 结合具体的业务场景的上下文来分析

如何避免ANR发生

1. 主线程尽量只做UI相关的操作,避免耗时操作，比如过度复杂的UI绘制，网络操作，文件IO操作；
2. 避免主线程跟工作线程发生锁的竞争，减少系统耗时binder的调用，谨慎使用sharePreference，注意主线程执行provider query操作

> 总之,尽可能减少主线程的负载，让其空闲待命，以期可随时响应用户的操作

trace.txt文件解读 【重点要看的】

1. 人为的收集trace.txt的命令
   `adb shell kill -3 888 //可指定进程pid`
   执行完该命令后traces信息的结果保存到文件/data/anr/traces.txt
2. trace文件解读

```
----- pid 888 at 2016-11-11 22:22:22 -----
Cmd line: system_server
ABI: arm
Build type: optimized
Zygote loaded classes=4113 post zygote classes=3239
Intern table: 57550 strong; 9315 weak
JNI: CheckJNI is off; globals=2418 (plus 115 weak)
Libraries: /system/lib/libandroid.so /system/lib/libandroid_servers.so
/system/lib/libaudioeffect_jni.so /system/lib/libcompiler_rt.so /system/lib/libjavacrypto.so
/system/lib/libjnigraphics.so /system/lib/libmedia_jni.so /system/lib/librs_jni.so
/system/lib/libsechook.so /system/lib/libshell_jni.so /system/lib/libsoundpool.so
/system/lib/libwebviewchromium_loader.so /system/lib/libwifi-service.so
/vendor/lib/libalarmservice_jni.so /vendor/lib/liblocationservice.so libjavacore.so (16)
//已分配堆内存大小40MB，其中29M已用，总分配207772个对象
Heap: 27% free, 29MB/40MB; 307772 objects
... //省略GC相关信息

//当前进程总99个线程
DALVIK THREADS (99):
//主线程调用栈
"main" prio=5 tid=1 Native
  | group="main" sCount=1 dsCount=0 obj=0x75bd9fb0 self=0x5573d4f770
  | sysTid=12078 nice=-2 cgrp=default sched=0/0 handle=0x7fa75fafe8
  | state=S schedstat=( 5907843636 827600677 5112 ) utm=453 stm=137 core=0 HZ=100
  | stack=0x7fd64ef000-0x7fd64f1000 stackSize=8MB
  | held mutexes=
  //内核栈
  kernel: __switch_to+0x70/0x7c
```

```
  kernel: SyS_epoll_wait+0x2a0/0x324
  kernel: SyS_epoll_pwait+0xa4/0x120
  kernel: cpu_switch_to+0x48/0x4c
  native: #00 pc 0000000000069be4 /system/lib64/libc.so (__epoll_pwait+8)
  native: #01 pc 000000000001cca4 /system/lib64/libc.so (epoll_pwait+32)
  native: #02 pc 000000000001ad74 /system/lib64/libutils.so (_ZN7android6Looper9pollInnerEi+144)
  native: #03 pc 000000000001b154 /system/lib64/libutils.so
(_ZN7android6Looper8pollOnceEiPiS1_PPv+80)
  native: #04 pc 00000000000d4bc0 /system/lib64/libandroid_runtime.so
(_ZN7android18NativeMessageQueue8pollOnceEP7_JNIEnvP8_jobjecti+48)
  native: #05 pc 000000000000082c /data/dalvik-cache/arm64/system@framework@boot.oat
(Java_android_os_MessageQueue_nativePollOnce__JI+144)
  at android.os.MessageQueue.nativePollOnce(Native method)
  at android.os.MessageQueue.next(MessageQueue.java:323)
  at android.os.Looper.loop(Looper.java:135)
  at com.android.server.SystemServer.run(SystemServer.java:290)
  at com.android.server.SystemServer.main(SystemServer.java:175)
  at java.lang.reflect.Method.invoke!(Native method)
  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628)

"Binder_1" prio=5 tid=8 Native
  | group="main" sCount=1 dsCount=0 obj=0x12c610a0 self=0x5573e5c750
  | sysTid=12092 nice=0 cgrp=default sched=0/0 handle=0x7fa2743450
  | state=S schedstat=( 796240075 863170759 3586 ) utm=50 stm=29 core=1 HZ=100
  | stack=0x7fa2647000-0x7fa2649000 stackSize=1013KB
  | held mutexes=
  kernel: __switch_to+0x70/0x7c
  kernel: binder_thread_read+0xd78/0xeb0
  kernel: binder_ioctl_write_read+0x178/0x24c
  kernel: binder_ioctl+0x2b0/0x5e0
  kernel: do_vfs_ioctl+0x4a4/0x578
  kernel: SyS_ioctl+0x5c/0x88
  kernel: cpu_switch_to+0x48/0x4c
  native: #00 pc 0000000000069cd0 /system/lib64/libc.so (__ioctl+4)
  native: #01 pc 0000000000073cf4 /system/lib64/libc.so (ioctl+100)
  native: #02 pc 000000000002d6e8 /system/lib64/libbinder.so
(_ZN7android14IPCThreadState14talkWithDriverEb+164)
  native: #03 pc 000000000002df3c /system/lib64/libbinder.so
(_ZN7android14IPCThreadState20getAndExecuteCommandEv+24)
  native: #04 pc 000000000002e114 /system/lib64/libbinder.so
(_ZN7android14IPCThreadState14joinThreadPoolEb+124)
  native: #05 pc 0000000000036c38 /system/lib64/libbinder.so (???)
  native: #06 pc 000000000001579c /system/lib64/libutils.so
(_ZN7android6Thread11_threadLoopEPv+208)
  native: #07 pc 0000000000090598 /system/lib64/libandroid_runtime.so
(_ZN7android14AndroidRuntime15javaThreadShellEPv+96)
  native: #08 pc 0000000000014fec /system/lib64/libutils.so (???)
  native: #09 pc 0000000000067754 /system/lib64/libc.so (_ZL15__pthread_startPv+52)
  native: #10 pc 000000000001c644 /system/lib64/libc.so (__start_thread+16)
  (no managed stack frames)
... //此处省略剩余的N个线程.
```

3. trace参数解读

```
"Binder_1" prio=5 tid=8 Native
  | group="main" sCount=1 dsCount=0 obj=0x12c610a0 self=0x5573e5c750
  | sysTid=12092 nice=0 cgrp=default sched=0/0 handle=0x7fa2743450
  | state=S schedstat=( 796240075 863170759 3586 ) utm=50 stm=29 core=1 HZ=100
  | stack=0x7fa2647000-0x7fa2649000 stackSize=1013KB
  | held mutexes=
```

- 第0行:
  - 线程名: Binder_1（如有daemon则代表守护线程)
  - prio: 线程优先级
  - tid: 线程内部id
  - 线程状态: NATIVE
- 第1行:
  - group: 线程所属的线程组
  - sCount: 线程挂起次数
  - dsCount: 用于调试的线程挂起次数
  - obj: 当前线程关联的java线程对象
  - self: 当前线程地址
- 第2行:
  - sysTid：线程真正意义上的tid
  - nice: 调度有优先级
  - cgrp: 进程所属的进程调度组
  - sched: 调度策略
  - handle: 函数处理地址
- 第3行:
  - state: 线程状态
  - schedstat: CPU调度时间统计
  - utm/stm: 用户态/内核态的CPU时间(单位是jiffies)
  - core: 该线程的最后运行所在核
  - HZ: 时钟频率
- 第4行:
  - stack：线程栈的地址区间
  - stackSize：栈的大小
- 第5行:
  - mutex: 所持有mutex类型，有独占锁exclusive和共享锁shared两类
- schedstat含义说明:
  - nice值越小则优先级越高。此处nice=-2, 可见优先级还是比较高的;
  - schedstat括号中的3个数字依次是Running、Runable、Switch，紧接着的是utm和stm
    - Running时间：CPU运行的时间，单位ns
    - Runable时间：RQ队列的等待时间，单位ns
    - Switch次数：CPU调度切换次数
    - utm: 该线程在用户态所执行的时间，单位是jiffies，jiffies定义为sysconf(_SC_CLK_TCK)，默认等于10ms

    - stm: 该线程在内核态所执行的时间，单位是jiffies，默认等于10ms
- 可见，该线程Running=186667489018ns,也约等于186667ms。在CPU运行时间包括用户态(utm)和内核态(stm)。 utm + stm = (12112 + 6554) ×10 ms = 186666ms。
- 结论：utm + stm = schedstat第一个参数值。