深入探索Android稳定性优化

转载自：**深入探索Android稳定性优化**

前言

成为一名优秀的Android开发，需要一份完备的知识体系，在这里，让我们一起成长为自己所想的那样~。

一、正确认识

- 稳定性很重要，Crash是P0优先级
- 稳定性可优化的面很广

1.1 稳定性纬度

- Crash纬度
- 性能纬度：启动速度、内存、绘制等等优化方向，相对于Crash来说是次要的
- 业务高可用纬度

1.2 稳定性优化概述

- 重在预防、监控必不可少
- 思考更深一层、重视隐含信息：如解决Crash问题时思考是否会引发同一类问题
- 长效保持需要科学流程

1.3 Crash相关指标

1.3.1 UV、PV

- PV（Page View）：访问量
- UV（Unique Visitor）：独立访客，0 – 24小时内的同一终端只计算一次

1.3.2 UV、PV、启动Crash率

- UV Crash率：针对用户使用量的统计，统计一段时间内所有用户发生崩溃的占比
- Crash UV / DAU：评估Crash率的影响范围，结合PV

注意：沿用同一指标

- PV Crash率：评估相关Crash影响的严重程度
- 启动Crash率：影响最严重的Crash，对用户伤害最大，无法通过热修复拯救，需结合客户端容灾
- 增量、存量Crash率：增量Crash是新版本重点，存量Crash是需要持续啃的硬骨头，优先解决增量、持续跟进存量

1.4 Crash率评价

- 必须在千分之二以下
- 万分位为优秀

1.5 Crash关键问题

尽可能还原Crash现场:

- 堆栈、设备、OS版本、进程、线程名、Logcat
  前后台、使用时长、App版本、小版本、渠道
  CPU架构、内存信息、线程数、资源包信息、用户行为日志
- Crash现场信息、Crash Top机型、OS版本、分布版本、区域
  Crash起始版本、上报趋势、是否新增、持续、量级
  根据以上信息决定Crash是否需要立马解决以及在哪个版本进行解决
- 参考Bugly平台的APM后台聚合展示

1.6 APM Crash部分整体架构

采集层

- 错误堆栈
- 设备信息
- 行为日志
- 其它信息

处理层

- 数据清洗
- 数据聚合
- 纬度分类
- 趋势对比

展示层

- 数据还原
- 纬度信息
- 起始版本
- 其它信息

报警层

- 环比：期与上一期进行对比
- 同比：如本月10号与上月10号
- 邮件
- IM
- 电话

责任归属

- 设立专项小组轮值
- 自动匹配责任人
- 处理流程全纪录

二、Crash优化

2.1 单个Crash处理方案

1、根据堆栈及现场信息找答案

- 解决90%问题
- 解决完后需考虑产生Crash深层次的原因

2、找共性：机型、OS、实验开关、资源包，考虑影响范围

3、线下复现、远程调试

2.2 Crash率治理方案

- 1、解决线上常规Crash
- 2、系统级Crash尝试Hook绕过
- 3、疑难Crash重点突破或更换方案

2.3 Java Crash

出现未捕获异常，导致出现异常退出

```
1 Thread.setDefaultUncaughtExceptionHandler();
```

我们通过设置自定义的UncaughtExceptionHandler，就可以在崩溃发生的时候获取到现场信息。注意，这个钩子是针对单个进程而言的，在多进程的APP中，监控哪个进程，就需要在哪个进程中设置一遍ExceptionHandler。

获取主线程的堆栈信息：

```
1 Looper.getMainLooper().getThread().getStackTrace();
```

获取当前线程的堆栈信息：

```
1 Thread.currentThread().getStackTrace();
```

获取全部线程的堆栈信息：

```
1 Thread.getAllStackTraces();
```

第三方Crash监控工具如Fabric、腾讯Bugly，都是以字符串拼接的方式将数组StackTraceElement[]转换成字符串形式，进行保存、上报或者展示。

2.3.1 如何反混淆上传的堆栈信息?

- 每次打包生成混淆APK的时候，需要把Mapping文件保存并上传到监控后台；
- Android原生的反混淆的工具包是retrace.jar，在监控后台用来实时解析每个上报的崩溃时。它会将Mapping文件进行文本解析和对象实例化，这个过程比较耗时。因此可以将Mapping对象实例进行内存缓存，但为了防止内存泄露和内存过多占用，需要增加定期自动回收的逻辑。

2.3.2 获取logcat方法

logcat日志流程是这样的，应用层 -> liblog.so -> logd，底层使用ring buffer来存储数据。获取的方式有以下三种：

1. 通过logcat命令获取。

- 优点：非常简单，兼容性好。
- 缺点：整个链路比较长，可控性差，失败率高，特别是堆破坏或者堆内存不足时，基本会失败。

2. hook liblog.so实现。通过hook liblog.so 中__android_log_buf_write 方法，将内容重定向到自己的buffer中。

- 优点：简单，兼容性相对还好。
- 缺点：要一直打开。

3. 自定义获取代码。通过移植底层获取logcat的实现，通过socket直接跟logd交互。

- 优点：比较灵活，预先分配好资源，成功率也比较高。
- 缺点：实现非常复杂

2.3.3 获取Java 堆栈

native崩溃时，通过unwind只能拿到Native堆栈。我们希望可以拿到当时各个线程的Java堆栈。

1. Thread.getAllStackTraces()。

优点：简单，兼容性好。
缺点：

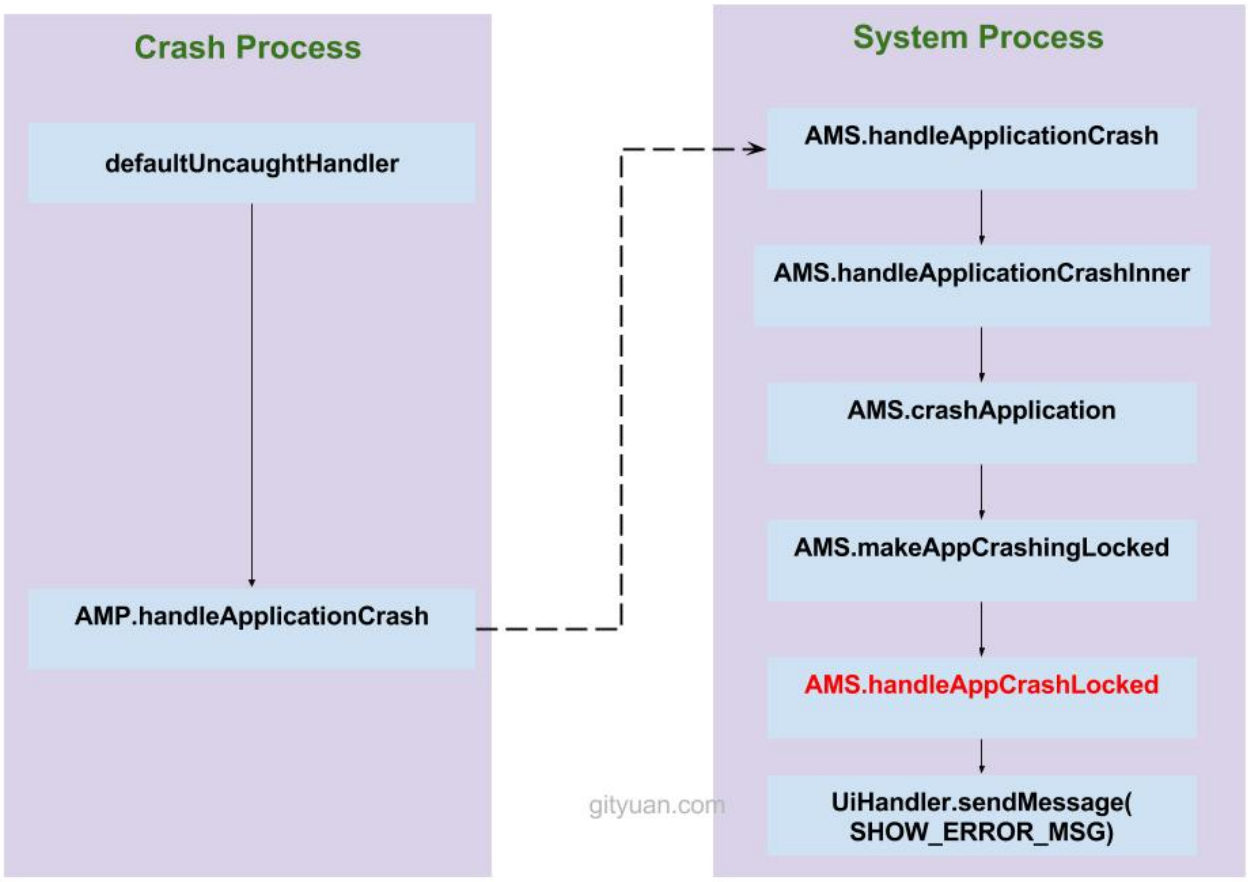
- 成功率不高，依靠系统接口在极端情况也会失败。
- 7.0之后这个接口是没有主线程堆栈。
- 使用Java层的接口需要暂停线程。

2. hook libart.so。通过hook ThreadList和Thread的函数，获得跟ANR一样的堆栈。为了稳定性，需要在fork的子进程中执行。

- 优点：信息很全，基本跟ANR的日志一样，有native线程状态，锁信息等等。
- 缺点：黑科技的兼容性问题，失败时可以用Thread.getAllStackTraces()兜底

2.3.4 Java Crash处理流程

借用Gityuan流程图如下所示：



1、首先发生crash所在进程，在创建之初便准备好了defaultUncaughtHandler，用来来处理Uncaught Exception，并输出当前crash基本信息；

2、调用当前进程中的AMP.handleApplicationCrash；经过binder ipc机制，传递到system_server进程；

3、接下来，进入system_server进程，调用binder服务端执行AMS.handleApplicationCrash；

4、从mProcessNames查找到目标进程的ProcessRecord对象；并将进程crash信息输出到目录/data/system/dropbox；

5、执行makeAppCrashingLocked：

- 创建当前用户下的crash应用的error receiver，并忽略当前应用的广播；
- 停止当前进程中所有activity中的WMS的冻结屏幕消息，并执行相关一些屏幕相关操作；

6、再执行handleAppCrashLocked方法：

- 当1分钟内同一进程连续crash两次时，且非persistent进程，则直接结束该应用所有activity，并杀死该进程以及同一个进程组下的所有进程。然后再恢复栈顶第一个非finishing状态的activity;
- 当1分钟内同一进程连续crash两次时，且persistent进程，，则只执行恢复栈顶第一个非finishing状态的activity;
- 当1分钟内同一进程未发生连续crash两次时，则执行结束栈顶正在运行activity的流程。

7、通过mUiHandler发送消息SHOW_ERROR_MSG，弹出crash对话框；

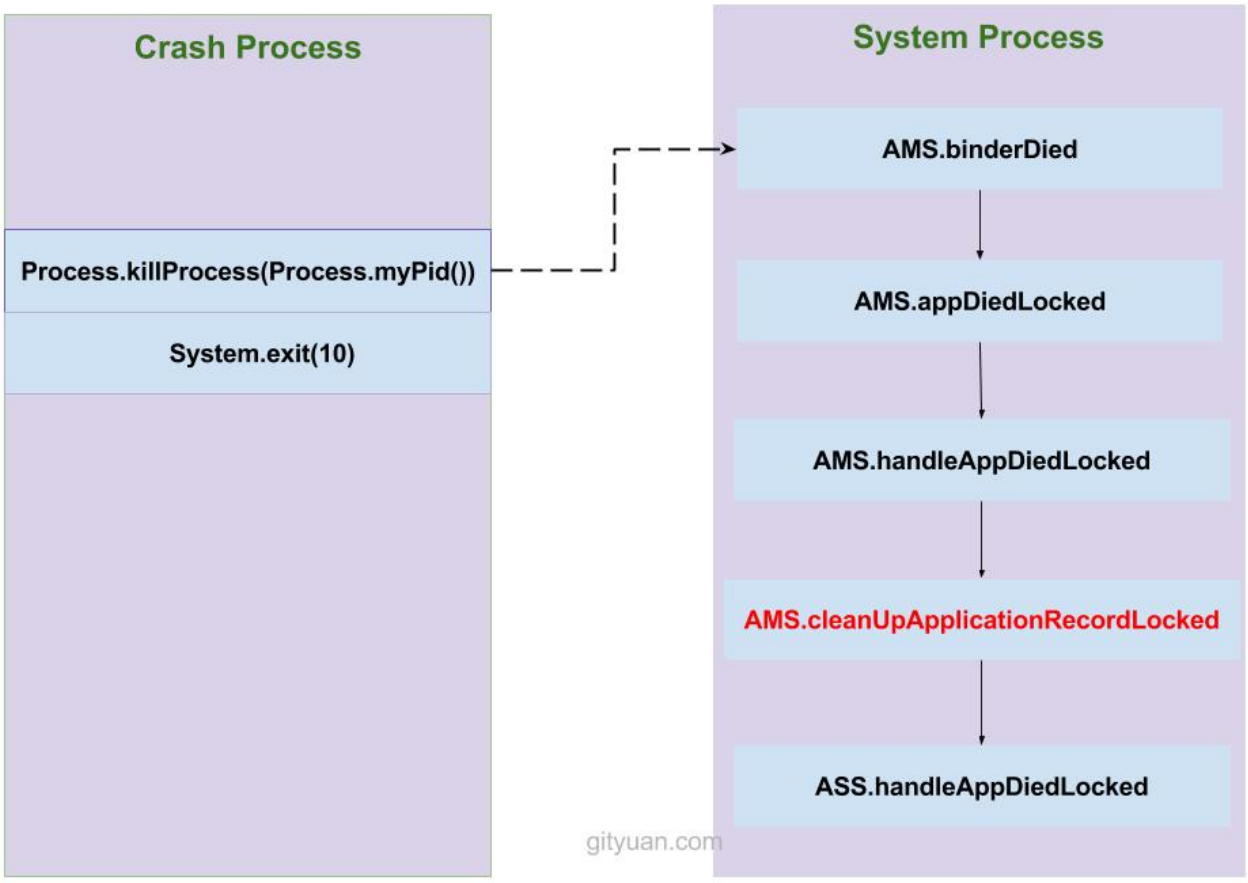
8、到此，system_server进程执行完成。回到crash进程开始执行杀掉当前进程的操作；

9、当crash进程被杀，通过binder死亡通知，告知system_server进程来执行appDiedLocked()；

10、最后，执行清理应用相关的activity/service/ContentProvider/receiver组件信息。

补充：binder 死亡通知原理

流程图如下：



由于Crash进程中拥有一个Binder服务端ApplicationThread，而应用进程在创建过程调用attachApplicationLocked()，从而attach到system_server进程，在system_server进程内有一个ApplicationThreadProxy，这是相对应的Binder客户端。当Binder服务端ApplicationThread所在进程(即Crash进程)挂掉后，则Binder客户端能收到相应的死亡通知，从而进入binderDied流程。

2.4 Native Crash

特点:

- 访问非法地址
- 地址对齐出错
- 发送程序主动abort

上述都会产生相应的signal信号，导致程序异常退出

2.4.1 合格的异常捕获组件

- 支持在crash时进行更多扩张操作
- 打印logcat和日志
- 上报crash次数

- 对不同crash做不同恢复措施
- 可以针对业务不断改进的适应

2.4.2 现有方案

Google Breakpad

- 优点：权威、跨平台
- 缺点：代码体量较大

Logcat

- 优点：利用安卓系统实现
- 缺点：需要在crash时启动新进程过滤logcat日志，不可靠

coffeecatch

- 优点：实现简洁、改动容易
- 缺点：有兼容性问题

2.4.3 Native崩溃捕获流程

1、编译端

编译C/C++需将带符号信息的文件保留下来。

2、客户端

捕获到崩溃时，将收集到尽可能多的有用信息写入日志文件，然后选择合适的时机上传到服务器。

3、服务端

读取客户端上报的日志文件，寻找合适的符号文件，生成可读的C/C++调用栈。

2.4.4 Native崩溃捕获的难点

核心：如何确保客户端在各种极端情况下依然可以生成崩溃日志。

1、文件句柄泄漏，导致创建日志文件失败?

提前申请文件句柄fd预留。

2、栈溢出导致日志生成失败?

- 使用额外的栈空间signalstack，避免栈溢出导致进程没有空间创建调用栈执行处理函数。（signalstack：系统会在危险情况下把栈指针指向这个地方，使得可以在一个新的栈上运行信号处理函数）
- 特殊请求需直接替换当前栈，所以应在堆中预留部分空间。

3、堆内存耗尽导致日志生产失败?

参考Breakpad重新封装Linux Syscall Support的做法以避免直接调用libc去分配堆内存。

4、堆破坏或二次崩溃导致日志生成失败?

Breakpad使用了fork子进程甚至孙进程的方式去收集崩溃现场，即便出现二次崩溃，也只是这部分信息丢失。

这里说下Breakpad缺点：

- 生成的minidump文件时二进制的，包含过多不重要的信息，导致文件数MB。但minidump可以使用gdb调试、看到传入参数。

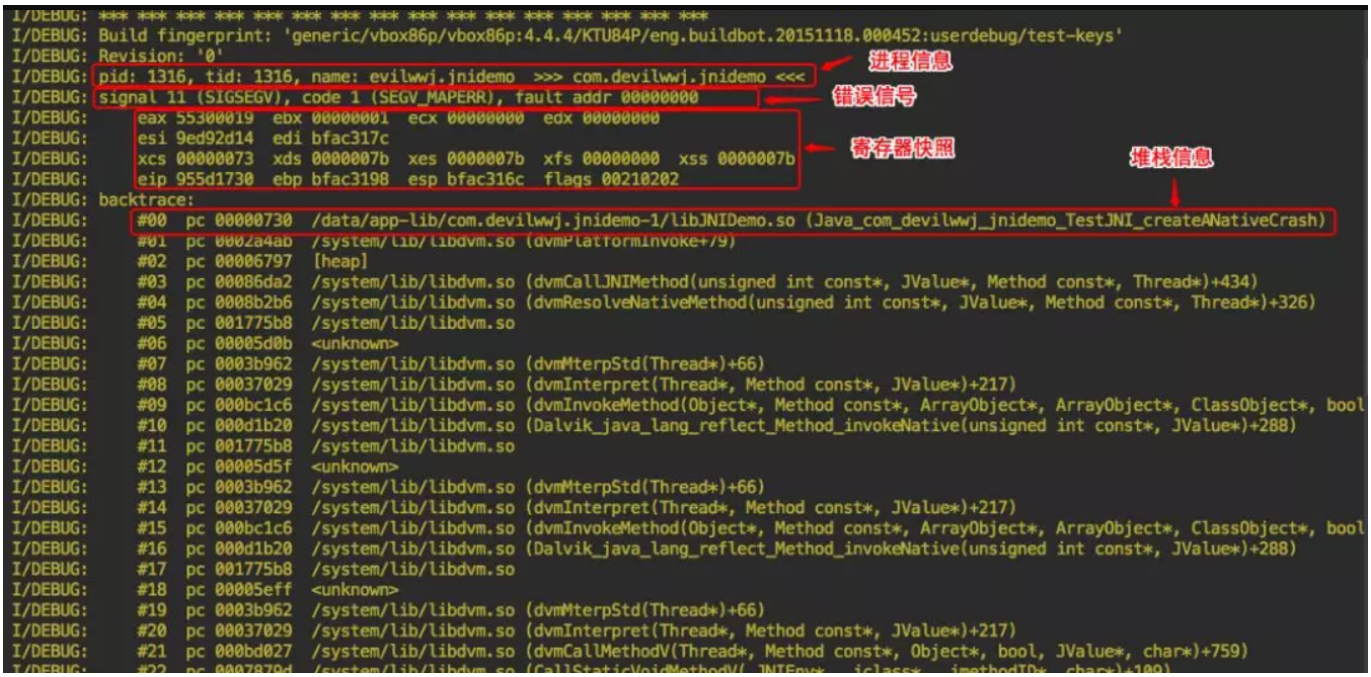
未来Chromium会使用Crashpad替代Breakpad。

5、想要遵循Android的文本格式并添加更多重要的信息？

改造Breakpad，增加Logcat信息，Java调用栈信息、其它有用信息。

2.4.5 Native崩溃捕获注册

一个Native Crash log信息如下：



堆栈信息中 pc 后面跟的内存地址，就是当前函数的栈地址，我们可以通过下面的命令行得出出错的代码行数

```
arm-linux-androideabi-addr2line -e 内存地址
```

下面列出全部的信号量以及所代表的含义：

```
#define SIGHUP 1  // 终端连接结束时发出(不管正常或非正常)
#define SIGINT 2  // 程序终止(例如Ctrl-C)
#define SIGQUIT 3 // 程序退出(Ctrl-\)
#define SIGILL 4 // 执行了非法指令，或者试图执行数据段，堆栈溢出
#define SIGTRAP 5 // 断点时产生，由debugger使用
#define SIGABRT 6 // 调用abort函数生成的信号，表示程序异常
#define SIGIOT 6 // 同上，更全，IO异常也会发出
#define SIGBUS 7 // 非法地址，包括内存地址对齐出错，比如访问一个4字节的整数，但
```

```
   其地址不是4的倍数
 9 #define SIGFPE 8 // 计算错误，比如除0、溢出
10 #define SIGKILL 9 // 强制结束程序，具有最高优先级，本信号不能被阻塞、处理和忽略
11 #define SIGUSR1 10 // 未使用，保留
12 #define SIGSEGV 11 // 非法内存操作，与 SIGBUS不同，他是对合法地址的非法访问，
   比如访问没有读权限的内存，向没有写权限的地址写数据
13 #define SIGUSR2 12 // 未使用，保留
14 #define SIGPIPE 13 // 管道破裂，通常在进程间通信产生
15 #define SIGALRM 14 // 定时信号，
16 #define SIGTERM 15 // 结束程序，类似温和的 SIGKILL，可被阻塞和处理。通常程序如
   果终止不了，才会尝试SIGKILL
17 #define SIGSTKFLT 16  // 协处理器堆栈错误
18 #define SIGCHLD 17 // 子进程结束时，父进程会收到这个信号。
19 #define SIGCONT 18 // 让一个停止的进程继续执行
20 #define SIGSTOP 19 // 停止进程,本信号不能被阻塞,处理或忽略
21 #define SIGTSTP 20 // 停止进程,但该信号可以被处理和忽略
22 #define SIGTTIN 21 // 当后台作业要从用户终端读数据时，该作业中的所有进程会收到S
   IGTTIN信号
23 #define SIGTTOU 22 // 类似于SIGTTIN，但在写终端时收到
24 #define SIGURG 23 // 有紧急数据或out-of-band数据到达socket时产生
25 #define SIGXCPU 24 // 超过CPU时间资源限制时发出
26 #define SIGXFSZ 25 // 当进程企图扩大文件以至于超过文件大小资源限制
27 #define SIGVTALRM 26 // 虚拟时钟信号. 类似于SIGALRM,      但是计算的是该进程
   占用的CPU时间.
28 #define SIGPROF 27 // 类似于SIGALRM/SIGVTALRM，但包括该进程用的CPU时间以及
   系统调用的时间
29 #define SIGWINCH 28 // 窗口大小改变时发出
30 #define SIGIO 29 // 文件描述符准备就绪，可以开始进行输入/输出操作
31 #define SIGPOLL SIGIO // 同上，别称
32 #define SIGPWR 30 // 电源异常
33 #define SIGSYS 31 // 非法的系统调用
```

一般关注SIGILL, SIGABRT, SIGBUS, SIGFPE, SIGSEGV, SIGSTKFLT, SIGSYS即可。
要订阅异常发生的信号，最简单的做法就是直接用一个循环遍历所有要订阅的信号，对每个信号调用sigaction()。
注意：

- JNI_OnLoad是最适合安装信号初识函数的地方。
- 建议在上报时调用Java层的方法统一上报。Native崩溃捕获注册。

2.4.6 崩溃分析流程

首先，应收集崩溃现场的一些信息，如下：

1、崩溃信息

- 进程名、线程名
- 崩溃堆栈和类型
- 有时候也需要知道主线程的调用栈

2、系统信息

- 系统运行日志

```
1 /system/etc/event-log-tags
```

- 机型、系统、厂商、CPU、ABI、Linux版本等

注意寻找共性问题

- 设备状态
- 是否root
- 是否是模拟器

3、内存信息

系统剩余内存

```
1 /proc/meminfo
```

当系统可用内存小于MemTotal的10%时，OOM、大量GC、系统频繁自杀拉起等问题非常容易出现

应用使用内存

包括Java内存、RSS、PSS

PSS和RSS通过/proc/self/smap计算，可以得到apk、dex、so等更详细的分类统计。

虚拟内存

大小：

```
1 /proc/self/status
```

具体分布情况：

```
1 /proc/self/maps
```

注意：

对于32位进程，32位CPU，虚拟内存达到3GB就可能会引起内存失败的问题。如果是64位的CPU，虚拟内存一般在3~4GB。如果支持64位进程，虚拟内存就不会成为问题。

4、资源信息

如果应用堆内存和设备内存比较充足，但还出现内存分配失败，则可能跟资源泄漏有关。

文件句柄fd

限制数：

```
/proc/self/limits
```

一般单个进程允许打开的最大句柄个数为1024，如果超过800需将所有fd和文件名输出日志进行排查。

线程数

大小：

```
/proc/self/status
```

一个线程一般占2MB的虚拟内存，线程数超过400个比较危险，需要将所有tid和线程名输出到日志进行排查。

JNI

容易出现引用失效、引用爆表等崩溃。

通过DumpReferenceTables统计JNI的引用表，进一步分析是否出现JNI泄漏等问题。

补充：dumpReferenceTables的出处

在dalvik.system.VMDebug类中，是一个native方法，亦是static方法；在JNI中可以这么调用

```
jclass vm_class = env->FindClass("dalvik/system/VMDebug");
jmethodID dump_mid = env->GetStaticMethodID( vm_class, "dumpReference
Tables", "()V" );
env->CallStaticVoidMethod( vm_class, dump_mid );
```

5、应用信息

- 崩溃场景
- 关键操作路径
- 其它跟自身应用相关的自定义信息：运行时间、是否加载补丁、是否全新安装或升级。

接下来进行崩溃分析：

1、确定重点

- 确认严重程度
- 优先解决Top崩溃或对业务有重大影响的崩溃：如启动、支付过程的崩溃
- Java崩溃：如果是OOM，需进一步查看日志中的内存信息和资源信息
- Native崩溃：查看signal、code、fault addr以及崩溃时的Java堆栈

常见的崩溃类型有

SIGSEGV：空指针、非法指针等

SIGABRT：ANR、调用abort推出等

如果是ANR，先看主线程堆栈、是否因为锁等待导致，然后看ANR日志中的iowait、CPU、GC、systemserver等信息，确定是I/O问题或CPU竞争问题还是大量GC导致的ANR。

注意：

当从一条崩溃日志中无法看出问题原因时，需要查看相同崩溃点下的更多崩溃日志，或者也可以查看内存信息、资源信息等进行异常排查。

2、查找共性

机型、系统、ROM、厂商、ABI这些信息都可以作为共性参考，对于下一步复现问题有明确指引。

3、尝试复现

复现之后再增加日志或使用Debugger、GDB进行调试。如不能复现，可以采用一些高级手段，如xlog日志、远程诊断、动态分析等等。

补充：系统崩溃解决方式

- 1、通过共性信息查找可能的原因
- 2、尝试使用其它使用方式规避
- 3、Hook解决

2.4.7 实战：使用Breakpad捕获native崩溃

首先，这里给出《Android开发高手课》张绍文老师写的crash捕获示例工程，工程里面已经集成了Breakpad 来获取发生 native crash 时候的系统信息和线程堆栈信息。下面来详细介绍下使用Breakpad来分析native崩溃的流程：

1、示例工程是采用cmake的构建方式，所以需要先到Android Studio中SDK Manager中的SDK Tools下下载NDK和cmake。

2、安装实例工程后，点击CRASH按钮产生一个native崩溃。生成的 crash信息，如果授予Sdcard权限会优先存放在/sdcard/crashDump下，便于我们做进一步的分析。反之会放到目录/data/data/com.dodola.breakpad/files/crashDump中。

3、使用adb pull命令将抓取到的crash日志文件放到电脑本地目录中：

```
adb pull /sdcard/crashDump/***.dmp > ~/Documents/crash_log.dmp
```

4、下载并编译Breakpad源码，在src/processor目录下找到minidump_stackwalk，使用这个工具将dmp文件转换为txt文件：

```
// 在项目目录下clone Breakpad仓库
git clone https://github.com/google/breakpad.git
// 切换到Breakpad根目录进行配置、编译
cd breakpad
./configure && make
// 使用src/processor目录下的minidump_stackwalk工具将dmp文件转换为txt文件
./src/processor/minidump_stackwalk ~/Documents/crashDump/crash_log.dmp >crash_log.txt
```

5、打开crash_log.txt，可以得到如下内容：

```
Operating system: Android
                  0.0.0 Linux 4.4.78-perf-g539ee70 #1 SMP PREEMPT Mon Jan 14 17:08:14 CST 2019 aarch64
```

```
CPU: arm64
     8 CPUs
GPU: UNKNOWN
Crash reason:  SIGSEGV /SEGV_MAPERR
Crash address: 0x0
Process uptime: not available
Thread 0 (crashed)
 0  libcrash-lib.so + 0x650
```

其中我们需要的关键信息为CPU是arm64的，并且crash的地址为0x650。接下来我们需要将这个地址转换为代码中对应的行。

6、ndk 中提供的addr2line来根据地址进行一个符号反解的过程。

如果是arm64的so使用 $NDKHOME/toolchains/aarch64-linux-android-4.9/prebuilt/darwin-x86_64/bin/aarch64-linux-android-addr2line。

如果是arm的so使用 $NDKHOME/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-addr2line。

由crash_log.txt的信息可知，我们机器的cpu架构是arm64的，因此需要使用aarch64-linux-android-addr2line这个命令行工具。该命令的一般使用格式如下：

// 注意：在mac下 ./ 代表执行文件 ./aarch64-linux-android-addr2line -e 对应的.so 需要解析的地址

上述中对应的.so文件在项目编译之后，会出现在Chapter01-master/sample/build/intermediates/merged_native_libs/debug/out/lib/arm64-v8a/libcrash-lib.so这个位置，由于我的手机CPU架构是arm64的，所以这里选择的是arm64-v8a中的libcrash-lib.so。接下来我们使用aarch64-linux-android-addr2line这个命令：

```
./aarch64-linux-android-addr2line -f -C -e ~/Documents/open-project/Chapter01-master/sample/build/intermediates/merged_native_libs/debug/out/lib/arm64-v8a/libcrash-lib.so 0x650
参数含义：
-e --exe=<executable>：指定需要转换地址的可执行文件名。
-f --functions：在显示文件名、行号输出信息的同时显示函数名信息。
-C --demangle[=style]：将低级别的符号名解码为用户级别的名字。
```

结果输出为：

```
Crash()
/Users/quchao/Documents/open-project/Chapter01-master/sample/src/main/cpp/crash.cpp:10
```

由此，我们得出crash的代码行为crahs.cpp文件中的第10行，接下来根据项目具体情况进行相应的修改即可。

Tips：这是从事NDK开发（音视频、图像处理、OpenCv、热修复框架开发）同学调试native层错误时经常要使用的技巧，强烈建议熟练掌握。

2.4.8 疑难Crash解决方案

问题1：如何解决Android 7.0 Toast BadTokenException?

参考Android 8.0 try catch的做法，代理Toast里的mTN（handler）就可以实现捕获异常。

问题2：如果解决 SharedPreference apply 引起的 ANR 问题

apply为什么会引起ANR?
SP 调用 apply 方法，会创建一个等待锁放到 QueuedWork 中，并将真正数据持久化封装成一个任务放到异步队列中执行，任务执行结束会释放锁。Activity onStop 以及 Service 处理 onStop，onStartCommand 时，执行 QueuedWork.waitToFinish() 等待所有的等待锁释放。
如何解决?
所有此类 ANR 都是经由 QueuedWork.waitToFinish() 触发的，只要在调用此函数之前，将其中保存的队列手动清空即可。
具体是Hook ActivityThrad的Handler变量，拿到此变量后给其设置一个Callback，Handler 的 dispatchMessage 中会先处理 callback。最后在 Callback
中调用队列的清理工作，注意队列清理需要反射调用 QueuedWork。
注意：
apply 机制本身的失败率就比较高（1.8%左右），清理等待锁队列对持久化造成的影响不大。

问题3：如何解决TimeoutExceptin异常?

它是由系统的FinalizerWatchdogDaemon抛出来的。
这里首先介绍下看门狗 WatchDog，它 的作用是监控重要服务的运行状态，当重要服务停止时，发生 Timeout 异常崩溃，WatchDog 负责将应用重启。而当关闭 WatchDog（执行stop（）方法）后，当重要服务停止时，也不会发生 Timeout 异常，是一种通过非正常手段防止异常发生的方法。
规避方案：
stop方法，在Android 6.0之前会有线程同步问题。
因为6.0之前调用threadToStop的interrupt方法是没有加锁的，所以可能会有线程同步的问题。
注意：Stop的时候有一定概率导致即使没有超时也会报timeoutexception。
缺点：
只是为了避免上报异常采取的一种hack方案，并没有真正解决引起finialize超市的问题。

问题4：如何解决输入法的内存泄漏?

通过反射将输入法的两个View置空。

2.4.9 进程保活

请参考深入探索Android启动速度优化一文。
这里补充一个方案，利用SyncAdapter提高进程优先级，它是Android系统提供一个账号同步机制，它属于核心进程级别，而使用了SyncAdapter的进程优先级本身也会提高，使用方式请Google，关联

SyncAdapter后，进程的优先级变为1，仅低于前台正在运行的进程，因此可以降低应用被系统杀掉的概率。

2.5 总结

- 重在预防：重视应用的整个流程、包括开发人员的培训、编译检查、静态扫描、规范的测试、灰度、发布流程等
- 不应该随意使用try catch去隐藏问题，而应该从源头入手，了解崩溃的本质原因，保证后面的运行流程。
- 解决崩溃的过程应该由点到面，考虑一类崩溃怎么解决。
- 崩溃与内存、卡顿、I/O内存紧密相关

三、ANR优化

3.1 ANR监控实现方式

1、使用FileObserver监听 /data/anr/traces.txt的变化

缺点：高版本ROM需要root权限
解决方案：海外Google Play服务、国内Hardcoder

2、监控消息队列的运行时间（WatchDog）

卡顿监控原理：

利用主线程的消息队列处理机制，应用发生卡顿，一定是在dispatchMessage中执行了耗时操作。我们通过给主线程的Looper设置一个Printer，打点统计dispatchMessage方法执行的时间，如果超出阀值，表示发生卡顿，则dump出各种信息，提供开发者分析性能瓶颈。
为卡顿监控代码增加ANR的线程监控，在发送消息时，在ANR线程中保存一个状态，主线程消息执行完后再Reset标志位。如果在ANR线程中收到发送消息后，超过一定时间没有复位，就可以任务发生了ANR。
缺点：

- 无法准确判断是否真正出现ANR，只能说明APP发生了UI阻塞，需要进行二次校验。校验的方式就是等待手机系统出现发生了Error的进程，并且Error类型是NOT_RESPONDING（值为2）。
  在每次出现ANR弹框前，Native层都会发出signal为SIGNAL_QUIT（值为3）的信号事件，也可以监听此信号。
- 无法得到完整ANR日志
- 隶属于卡顿优化的方式

需要考虑应用退出场景

- 主动自杀
- Process.killProcess()、exit()等。
- 崩溃
- 系统重启
- 系统异常、断电、用户重启等：通过比较应用开机运行时间是否比之前记录的值更小。
- 被系统杀死

- 被LMK杀死、从系统的任务管理器中划掉等。

注意：由于traces.txt上传比较耗时，所以一般线下采用，线上建议综合ProcessErrorStateInfo和出现ANR时的堆栈信息来实现ANR的实时上传。

3.2 ANR优化

ANR发生原因：没有在规定的时间内完成要完成的事情。

3.2.1 ANR分类

发生场景

- Activity onCreate方法或Input事件超过5s没有完成
- BroadcastReceiver前台10s，后台60s
- ContentProvider 在publish过超时10s;
- Service前台20s，后台200s

发生原因

- 主线程有耗时操作
- 复杂布局
- IO操作
- 被子线程同步锁block
- 被Binder对端block
- Binder被占满导致主线程无法和SystemServer通信
- 得不到系统资源（CPU/RAM/IO）

从进程角度看发生原因有：

- 当前进程：主线程本身耗时或者主线程的消息队列存在耗时操作、主线程被本进程的其它子线程所blocked
- 远端进程：binder call、socket通信

Andorid系统监测ANR的核心原理是消息调度和超时处理。

3.2.2 ANR排查流程

1、Log获取

1、抓取bugreport

```
adb shell bugreport > bugreport.txt
```

2、直接导出/data/anr/traces.txt文件

```
adb pull /data/anr/traces.txt trace.txt
```

2、搜索“ANR in”处log关键点解读

- 发生时间（可能会延时10-20s）

- pid：当pid=0，说明在ANR之前，进程就被LMK杀死或出现了Crash，所以无法接受到系统的广播或者按键消息，因此会出现ANR
- cpu负载Load: 7.58 / 6.21 / 4.83
  代表此时一分钟有平均有7.58个进程在等待
  1、5、15分钟内系统的平均负荷
  当系统负荷持续大于1.0，必须将值降下来
  当系统负荷达到5.0，表面系统有很严重的问题
- cpu使用率

```
CPU usage from 18101ms to 0ms ago
28% 2085/system_server: 18% user + 10% kernel / faults: 8689 minor 24 major
11% 752/android.hardware.sensors@1.0-service: 4% user + 6.9% kernel / faults: 2 minor
9.8% 780/surfaceflinger: 6.2% user + 3.5% kernel / faults: 143 minor 4 major
```

上述表示Top进程的cpu占用情况。
注意：如果CPU使用量很少，说明主线程可能阻塞。

3、在bugreport.txt中根据pid和发生时间搜索到阻塞的log处

```
----- pid 10494 at 2019-11-18 15:28:29 -----
```

4、往下翻找到“main”线程则可看到对应的阻塞log

```
"main" prio=5 tid=1 Sleeping
| group="main" sCount=1 dsCount=0 flags=1 obj=0x746bf7f0 self=0xe7c8f000
| sysTid=10494 nice=-4 cgrp=default sched=0/0 handle=0xeb6784a4
| state=S schedstat=( 5119636327 325064933 4204 ) utm=460 stm=51 core=4 HZ=100
| stack=0xff575000-0xff577000 stackSize=8MB
| held mutexes=
```

关键字段含义：

- tid：线程号
- sysTid：主进程线程号和进程号相同
- Waiting/Sleeping：各种线程状态
- nice：nice值越小，则优先级越高，-17~16
- schedstat：Running、Runable时间(ns)与Switch次数

- utm：该线程在用户态的执行时间(jiffies)
- stm：该线程在内核态的执行时间(jiffies)
- sCount：该线程被挂起的次数
- dsCount：该线程被调试器挂起的次数
- self：线程本身的地址

其它分析方法：Java线程调用分析方法

- 先使用jps命令列出当前系统中运行的所有Java虚拟机进程，拿到应用进程的pid。
- 然后再使用jstack命令查看该进程中所有线程的状态以及调用关系，以及一些简单的分析结果。

问题：

1、sp调用apply导致anr问题?
虽然apply并不会阻塞主线程，但是会将等待时间转嫁到主线程。
2、检测运行期间是否发生过异常退出?
在应用启动时设定一个标志，在主动自杀或崩溃后更新标志 ，下次启动时检测此标志即可判断。

3.2.4 理解ANR的触发流程

broadcast跟service超时机制大抵相同，但有一个非常隐蔽的技能点，那就是通过静态注册的广播超时会受SharedPreferences(简称SP)的影响。
当SP有未同步到磁盘的工作，则需等待其完成，才告知系统已完成该广播。并且只有XML静态注册的广播超时检测过程会考虑是否有SP尚未完成，动态广播并不受其影响。

- 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.
- 对于输入事件发生ANR，首先会调用InputMonitor.notifyANR，最终也会调用AMS.appNotResponding。

3.2.4.1 AMS.appNotResponding流程

- 输出ANR Reason信息到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;
- 收集并输出重要进程列表中的各个线程的traces信息，该方法较耗时;
- 输出当前各个进程的CPU使用情况以及CPU负载情况;
- 将traces文件和 CPU使用情况信息保存到dropbox，即data/system/dropbox目录（ANR信息最为重要的信息）
- 根据进程类型,来决定直接后台杀掉,还是弹框告知用户.

3.2.4.2 AMS.dumpStackTraces流程

1、收集firstPids进程的stacks：

- 第一个是发生ANR进程；
- 第二个是system_server；
- 其余的是mLruProcesses中所有的persistent进程。

2、收集Native进程的stacks。(dumpNativeBacktraceToFile)

- 依次是mediaserver,sdcard,surfaceflinger进程。

3、收集lastPids进程的stacks：

- 依次输出CPU使用率top 5的进程；

注意：

上述导出每个进程trace时，进程之间会休眠200ms。

四、移动端业务高可用方案建设

2.3.1 业务高可用重要性

- 高可用
- 性能
- 业务
- 侧重于用户功能完整可用
- 真实影响收入

2.3.2 业务高可用方案建设

- 数据采集
- 梳理项目主流程、核心路径、关键节点
- Aop自动采集、统一上报
- 报警策略：阈值报警、趋势报警、特定指标报警、直接上报（或底阈值）
- 异常监控
- 单点追查：需要针对性分析的特定问题，全量日志回捞，专项分析
- 兜底策略
- 配置中心、功能开关
- 跳转分发中心（组件化路由）

2.3.3 移动端容灾方案

灾包括：

- 性能异常
- 业务异常

传统流程：

用户反馈、重新打包、渠道更新、不可接受。

2.3.3.1 容灾方案建设：

功能开关

配置中心，服务端下发配置控制

针对场景：

- 功能新增
- 代码改动

统跳中心

- 界面切换通过路由，路由决定是否重定向
- Native Bug不能热修复则跳转到临时H5页面

动态化修复

- 热修复能力，可监控、灰度、回滚、清除

推拉结合、多场景调用保证到达率

Weex、RN增量更新

安全模式

微信读书、蘑菇街、淘宝、天猫等“重运营”的APP都使用了安全模式保障客户端启动流程，启动失败后给用户自救机会。先介绍一下它的核心特点：

- 根据Crash信息自动恢复，多次启动失败重置应用为安装初始状态
- 严重Bug可阻塞性热修复

安全模式设计

配置后台：统一的配置后台，具备灰度发布机制

1、客户端能力：

- 在APP连续Crash的情况下具备分级、无感自修复能力
- 具备同步热修复能力
- 具备指定触发某项特定功能的能力
- 具体功能注册能力，方便后期扩展安全模式

2、数据统计及告警

- 统一的数据平台
- 监控告警功能，及时发现问题
- 查看热修复成功率等数据

3、快速测试

- 优化预发布环境下测试
- 优化回归验证安全模式难点等

天猫安全模式原理

1、如何判断异常退出?

APP启动时记录一个flag值，满足以下条件时，将flag值清空

- APP正常启动10秒
- 用户正常退出应用
- 用户主动从前台切换到后台

如果在启动阶段发生异常，则flag值不会清空，通过flag值就可以判断客户端是否异常退出，每次异常退出，flag值都+1。

2、安全模式的分级执行策略

分为两级安全模式，连续Crash 2次为一级安全模式，连续Crash 2次及以上为二级安全模式。

业务线可以在一级安全模式中注册行为，比如清空缓存数据，再进入该模式时，会使用注册行为尝试修复客户端

如果一级安全模式无法修复APP，则进入二级安全模式将APP恢复到初次安装状态，并将Document、Library、Cache三个根目录清空。

3、热修复执行策略

只要发现配置中需要热修复，APP就会同步阻塞进行热修复,保证修复的及时性

4、灰度方案

灰度时，配置中会包含灰度、正式两份配置及其灰度概率

APP根据特定算法算出自己是否满足灰度条件，则使用灰度配置

易用性考量

1、接入成本

完善文档、接口简洁

2、统一配置后台

可按照APP、版本配置

3、定制性

支持定制功能，让接入方来决定具体行为

4、灰度机制

5、数据分析

采用统一数据平台，为安全模式改进提供依据

6、快速测试

创建更多的针对性测试案例，如模拟连续Crash

异常熔断

- 多次请求失败则可让网络库主动拒绝请求

容灾方案集合路径

功能开关 -> 统跳中心 -> 动态修复 -> 安全模式

五、稳定性长效治理

开发阶段

- 统一编码规范、增强编码功底、技术评审、CodeReview机制
- 架构优化
- 能力收敛
- 统一容错：如在网络库utils中统一对返回信息进行预校验，如不合法就直接不走接下来的流程。

测试阶段

- 功能测试、自动化测试、回归测试、覆盖安装
- 特殊场景、机型等边界测试：如服务端返回异常数据、服务端宕机
- 云测平台：提供更全面的机型进行测试

合码阶段

- 编译检测、静态扫描
- 预编译流程、主流程自动回归

发布阶段

- 多轮灰度
- 分场景、纬度全面覆盖

运维阶段

- 灵敏监控
- 回滚、降级策略
- 热修复、本地容灾方案

六、稳定性优化问题

1、你们做了哪些稳定性方面的优化?

- Crash专项优化
- 性能稳定性优化
- 业务稳定性优化

根据以上三方面的优化我们搭建了移动端的高可用平台。

2、性能稳定性是怎么做的?

- 全面的性能优化：启动速度、内存优化、绘制优化
- 线下发现问题、优化为主
- 线上监控为主
- Crash专项优化

3、业务稳定性如何保障?

- 数据采集 + 报警
- 需要对项目的主流程与核心路径进行埋点监控，
- 同时还需知道每一步发生了多少异常，这样，我们就知道了所有业务流程的转换率以及相应界面的转换率
- 结合大盘，如果转换率低于某个值，进行报警
- 异常监控 + 单点追查
- 兜底策略

4、如果发送了异常情况，怎么快速止损?

- 功能开关
- 统跳中心
- 动态修复：热修复、资源包更新
- 自主修复：安全模式

七、总结

Android稳定性优化是一个需要长期投入，持续运营和维护的一个过程，上文中我们不仅深入探讨了Java Crash、Native Crash和ANR的解决流程及方案，还分析了其内部实现原理和监控流程。到这里，可以看到，要想做好稳定性优化，我们必须对虚拟机运行、Linux信号处理和内存分配有一定程度的了解，只有深入了解这些底层知识，我们才能比别人设计出更好的稳定性优化方案。

参考链接：

1、《Android性能优化最佳实践》第五章 稳定性优化
2、慕课网之国内Top团队大牛带你玩转Android性能分析与优化 第十一章 App稳定性优化
3、极客时间之Android开发高手课 崩溃优化
4、Android 平台 Native 代码的崩溃捕获机制及实现
5、安全模式：天猫App启动保护实践
6、美团外卖Android Crash治理之路 （进阶）
7、海神平台Crash监控SDK（Android）开发经验总结
8、Android Native Crash 收集
9、理解Android Crash处理流程
10、Android应用ANR分析
11、理解Android ANR的触发原理
12、Input系统—ANR原理分析
13、ANR监测机制
14、理解Android ANR的触发原理
15、理解Android ANR的信息收集过程
16、应用与系统稳定性第一篇—ANR问题分析的一般套路
17、巧妙定位ANR问题
18、剖析 SharedPreference apply 引起的 ANR 问题
19、Linux错误信号