

01

Part One

日志分析

- **Shell**: 是用户和Linux内核之间的接口程序, 为用户提供使用操作系统的接口, 它是命令语言、命令解释程序及程序设计语言的统称, 拥有自己内建的shell命令集
- **ADB**: 全称为Android Debug Bridge, 借助这个工具设备上运行Shell命令
- **RAM**: 随机存取存储器 (Random Access Memory), 内存
- **ROM**: Read Only Memory Image (只读内存镜像) 的简称, 常见ROM有刷机镜像/BIOS等
- **ANR**: Application Not Responding, 用程序无响应. 默认情况下, 在Android中Activity的最长执行时间是5s, BroadcastReceiver的最长执行时间则是10s
- **NDK**: Native Development Kit, 通过NDK, 可以用C或C++来开发App

日志分析 – Logcat 日志格式

```
04-01 18:16:07.792 W/GooglePlayServicesUtil(11624): Google Play Store is missing.
04-01 18:16:07.802 E/ALSAStreamOps( 349): setParameters(): keyRouting with device2 0x2
04-01 18:16:07.812 I/OppoExVibrator(11624): OppoExVibrator setSecureController code=10002
04-01 18:16:07.812 I/am_finish_activity(21596): [0,1115134440,597,com.infinitus.idep.bupm.test/com.infinitus.bupm.acti
04-01 18:16:07.822 I/am_pause_activity(21596): [0,1115134440,com.infinitus.idep.bupm.test/com.infinitus.bupm.activity.
04-01 18:16:07.832 I/AndroidRuntime(11624): VM exiting with result code 0, cleanup skipped.
04-01 18:16:07.852 E/ACDB-LOADER( 349): Error: ACDB AudProc vol returned = -19
04-01 18:16:07.892 I/ActivityManager(21596): Process com.infinitus.idep.bupm.test (pid 11624) has died.
04-01 18:16:07.892 I/WindowState(21596): WIN DEATH: Window{43b67068 u0 com.infinitus.idep.bupm.test/com.infinitus.bupm
04-01 18:16:07.902 W/ContextImpl(21596): Calling a method in the system process without a qualified user: android.app.
04-01 18:16:07.902 W/ActivityManager(21596): Scheduling restart of crashed service com.infinitus.idep.bupm.test/com.in
04-01 18:16:07.902 W/ActivityManager(21596): Scheduling restart of crashed service com.infinitus.idep.bupm.test/com.in
04-01 18:16:07.902 I/am_proc_died(21596): [0,11624,com.infinitus.idep.bupm.test]
04-01 18:16:07.902 I/am_proc_died(21596): [0,11624,com.infinitus.idep.bupm.test]
04-01 18:16:07.902 I/am_schedule_service_restart(21596): [0,com.infinitus.idep.bupm.test/com.infinitus.bupm.plugins.so
04-01 18:16:07.902 I/am_schedule_service_restart(21596): [0,com.infinitus.idep.bupm.test/com.infinitus.bupm.plugins.so
04-01 18:16:07.912 I/OppoNarrowFrame(21596): IOppoNarrowFrame setNarrowFrameHome code=10004
04-01 18:16:07.912 W/ContextImpl(21596): Calling a method in the system process without a qualified user: android.app.
```

PID:11624, App(com.infinitus.idep.bupm.test)的进程ID

包名: Package Name (com.infinitus.idep.bupm.test)

优先级按照从低到高顺利排列

- [V] `Verbose` : 详细信息, 输出颜色为黑色
- [D] `Debug` : 调试信息, 输出颜色为蓝色
- [I] `Info` : 通告信息, 输出颜色为绿色
- [W] `Warning` : 警告信息, 输出颜色为橙色
- [E] `Error` : 错误信息, 输出颜色为红色
- [F] `Fatal` : 严重错误, 输出颜色为红色
- [A] `Assert` : 断言信息, 4.0版本新增

日志分析 – TestinExternalLog

名称	Log
开始安装	<<<<<< Begin installing package com.example.android.apis >>>>>>
获取进程名称	<<<<<< Process com.example.android.apis >>>>>>
安装成功	<<<<<< Package com.example.android.apis installed >>>>>>
安装失败	<<<<<< Package com.example.android.apis install failed >>>>>>
开始启动测试流程	<<<<<< StartTest launcher Package com.example.android.apis >>>>>>
启动应用	<<<<<< Starting Package com.example.android.apis >>>>>>
	<<<<<< StartApp >>>>>>
启动耗时	Displayed com.example.android.apis/com.example.android.apis.ApiDemos: +348ms
获取进程id	<<<<<< com.example.android.apis(19201) >>>>>>
启动测试流程结束	<<<<<< StopTest launcher Package com.example.android.apis >>>>>>
开始UI 适配流程	<<<<<< StartTest ui Package com.example.android.apis >>>>>>
UI 适配流程结束	<<<<<< StopTest ui Package com.example.android.apis >>>>>>
开始卸载	<<<<<< Begin uninstalling package com.example.android.apis >>>>>>
卸载完成	<<<<<< Package com.example.android.apis uninstalled >>>>>>
Cpu 占用	CPU>>>> User 15%, System 8%, IOW 0%, IRQ 0%, com.example.android.apis(19201):3%
内存占用	Mem>>>> Available:1137808KB, com.example.android.apis(19326) used:63078KB
网络流量消耗	NetFlow>>>> UpFlow 60, DownFlow 40, TotalFlow 100
截图	<<<<<< Capturing image (1) >>>>>>

日志分析 – TestinExternalLog

04-14 15:35:00.609 I/TestinExternalLog(9299): NetState>>>> cn.bing.com:80,Level 1,Tag 1,Number 268

Level: 网络状态，0表示无网络， 1表示网络很好， 2~3：好 4~6：一般， 7~9：差， 10：很差

Tag: 标识号。1 默认地址（代表终端网络） 2 传入的地址（代表被测APP网络）

Number: 批次号

03-30 04:46:07.181 I/TestinExternalLog(1): Mem>>>> Avalable:670211KB, winretailsaler.net.winchannel.wincrm(13537) used:18019KB

Available: 可用内存，单位KB

userd: 占用内存，单位KB

03-30 04:46:09.546 I/TestinExternalLog(1): CPU>>>> User 21%, System 14%, IOW 2%, IRQ 0%, winretailsaler.net.winchannel.wincrm(13537):20%

User: 用户进程， System: 系统进程， IOW: IO等待， IRQ: 硬中断， App占用

02-13 11:07:28.402 I/TestinExternalLog(6687): GPU>>>> kgsI-3d0:0%

名称(kgsI-2d0\kgsI-2d1\kgsI-3d0) : GPU负载百分比

02-13 11:11:22.412 I/TestinExternalLog(6687): Battery>>>> Temperature 28.7

电池温度，单位℃

02-13 11:11:29.302 I/TestinExternalLog(6687): NetFlow>>>> UpFlow 3651, DownFlow 1842, TotalFlow 5493

上行流量(B), 下行流量(b) , 总流量(B)

02-12 20:33:43.308 I/TestinExternalLog(15221): TestinLog-FPS>>>> Type powervr, Frames 10

日志分析步骤

1. 搜索PackageName或者PID，过滤出主要信息
2. 搜索关键词 TestinExternalLog: <<<<<<< StopTest ui Package ***** >>>>>>>, 确定停止测试的时间点或者文本数据行号
3. 根据错误描述由此向上查找关键词，如：ANR、FATAL、EXCEPTION、am_kill、am_crash、Force stopping
4. 找到错误日志，分析上下文日志，判断问题原因
5. 如果没有明显错误日志（崩溃SDK捕获、系统未输出等） ---➔
6. 分析是否JNI（NDK、 SIGNAL）调用导致
7. 分析App运行、界面（AM、AMS）创建及销毁轨迹，并查看日志中是否有内存(lowmemory、GC)或者CPU(doin too much work)相关日志
8. 综合判断问题原因

ANR 程序无响应

ANR一般有三种类型：

- KeyDispatchTimeout：界面操作(按键/触摸)后等待响应时间超过5s
- BroadcastTimeout：BroadcasterReciver里的onRecive（）方法处理超过10s
- ServiceTimeout：HandleMessage回调方法(Service)执行超过20s，概率较小

为什么超时？超时原因：

- 当前的事件没有机会得到处理（即UI线程正在处理前一个事件，没有及时的完成或者looper被某种原因阻塞住了）
- 当前的事件正在处理，但没有及时完成

ANR与报异常Exception的区别

ANR提示“XXX无响应”，确认“强行关闭”会杀掉进程。

Exception提示“XXX停止”。

ANR的本质是什么？

- 1、JAVA进程都用于一个主线程和主线程消息队列，这里的主线程就是ActivityThread；
- 2、主线程负责监听、接受和处理UI事件，比如绘制界面等；
- 3、主线程会从主线程队列中读取消息，而且需要尽快分发出去；
- 4、主线程只有在完成当前消息处理后，才从消息队列取出下一个消息；
- 5、如果主线程在处理某个消息时卡住了，没能及时分发，就会发生ANR。

ANR诊断的关键点和文件

关键点：ANR诊断的关键是需要知道主线程为什么没能及时的处理消息

- 1、没能获取到CPU时间？
- 2、正在等待某个时间的发生，从而决定下一步动作？
- 3、当前消息的处理事项过于复杂？

ANR 程序无响应

如何避免？

- UI线程尽量只做跟UI相关的工作
- 耗时工作（数据库操作、I/O 操作、连接网络等）放到单独线程处理
- 尽量用Handler来处理UI线程与其他线程间的交互

ANR 程序无响应

ANR 分析定位:

- 查看Log日志，搜索关键词 ANR（后添加一个空格，过滤无效信息）
- 从trace文件查看调用stack

```
04-01 13:12:11.572 I/InputDispatcher( 220): Application is not responding:Window{2b263310com.android.email/com.android.email.activity.SplitScreenActivitypaused=
04-01 13:12:11.572 I/WindowManager( 220): Input event dispatching timedout sending to com.android.email/com.android.email.activity.SplitScreenActivity

04-01 13:12:14.123 I/Process( 220): Sending signal. PID: 21404 SIG: 3
04-01 13:12:14.123 I/dalvikvm(21404):threadid=4: reacting to signal 3

-----
04-01 13:12:15.872 E/ActivityManager( 220): ANR in com.android.email(com.android.email/.activity.SplitScreenActivity)
04-01 13:12:15.872 E/ActivityManager( 220): Reason:keyDispatchingTimedOut
04-01 13:12:15.872 E/ActivityManager( 220): Load: 8.68 / 8.37 / 8.53
04-01 13:12:15.872 E/ActivityManager( 220): CPUUsage from 4361ms to 699ms ago

04-01 13:12:15.872 E/ActivityManager( 220): 5.5%21404/com.android.email: 1.3% user + 4.1% kernel / faults:10 minor
04-01 13:12:15.872 E/ActivityManager( 220): 4.3%220/system_server: 2.7% user + 1.5% kernel / faults: 11 minor 2 major
04-01 13:12:15.872 E/ActivityManager( 220): 0.9%52/spi_qsd.0: 0% user + 0.9% kernel
04-01 13:12:15.872 E/ActivityManager( 220): 0.5%65/irq/170-cyttsp-: 0% user + 0.5% kernel
04-01 13:12:15.872 E/ActivityManager( 220): 0.5%296/com.android.systemui: 0.5% user + 0% kernel
04-01 13:12:15.872 E/ActivityManager( 220): 100%TOTAL: 4.8% user + 7.6% kernel + 87% iowait
04-01 13:12:15.872 E/ActivityManager( 220): CPUUsage from 3697ms to 4223ms later:
04-01 13:12:15.872 E/ActivityManager( 220): 25%21404/com.android.email: 25% user + 0% kernel / faults: 191 minor
04-01 13:12:15.872 E/ActivityManager( 220): 16% 21603/_eas(par.hakan: 16% user + 0% kernel
04-01 13:12:15.872 E/ActivityManager( 220): 7.2% 21406/GC: 7.2% user + 0% kernel
04-01 13:12:15.872 E/ActivityManager( 220): 1.8% 21409/Compiler: 1.8% user + 0% kernel
04-01 13:12:15.872 E/ActivityManager( 220): 5.5%220/system_server: 0% user + 5.5% kernel / faults: 1 minor
04-01 13:12:15.872 E/ActivityManager( 220): 5.5% 263/InputDispatcher: 0% user + 5.5% kernel
04-01 13:12:15.872 E/ActivityManager( 220): 32%TOTAL: 28% user + 3.7% kernel
```

---发生ANR的时间和生成trace.txt的时间

----CPU在ANR发生前的使用情况

-- ANR后CPU的使用量

ANR 程序无响应

从Log分析ANR的类型

- 分析CPU的使用情况，如果CPU使用量接近100%，说明当前设备很忙，有可能是CPU繁忙导致了ANR，如果CPU使用量很少，说明主线程被BLOCK了
- 如果IO wait很高，说明ANR有可能是主线程在进行I/O操作造成的

ANR 程序无响应

ANR 分析定位：

- 查看Log日志，搜索关键词 ANR（后添加一个空格，过滤无效信息）
- 从trace文件查看调用stack（trace.txt文件目录 data/anr/traces.txt）

ANR 程序无响应

从trace.txt文件，看到最多的是如下的信息：

```
-----pid 21404 at 2011-04-01 13:12:14 -----
Cmdline: com.android.email

DALVIK THREADS:
(mutexes: tll=0tsl=0 tscl=0 ghl=0 hwl=0 hwll=0)
"main" prio=5 tid=1 NATIVE
| group="main" sCount=1 dsCount=0obj=0x2aad2248 self=0xcf70
| sysTid=21404 nice=0 sched=0/0cgrp=[fopen-error:2] handle=1876218976
at android.os.MessageQueue.nativePollOnce(Native Method)
at android.os.MessageQueue.next(MessageQueue.java:119)
at android.os.Looper.loop(Looper.java:110)
at android.app.ActivityThread.main(ActivityThread.java:3688)
at java.lang.reflect.Method.invokeNative(Native Method)
at java.lang.reflect.Method.invoke(Method.java:507)
at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:866)
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:624)
at dalvik.system.NativeStart.main(Native Method)
```

说明主线程在等待下条消息进入消息队列

ANR 程序无响应

```
/*
 * Current status; these map to JDWP constants, so don't rearrange them.
 * (If you do alter this, update the strings in dvmDumpThread and the
 * conversion table in VMThread.java.)
 *
 * Note that "suspended" is orthogonal to these values (so says JDWP).
 */
enum ThreadStatus {
    THREAD_UNDEFINED      = -1,          /* makes enum compatible with int32_t */

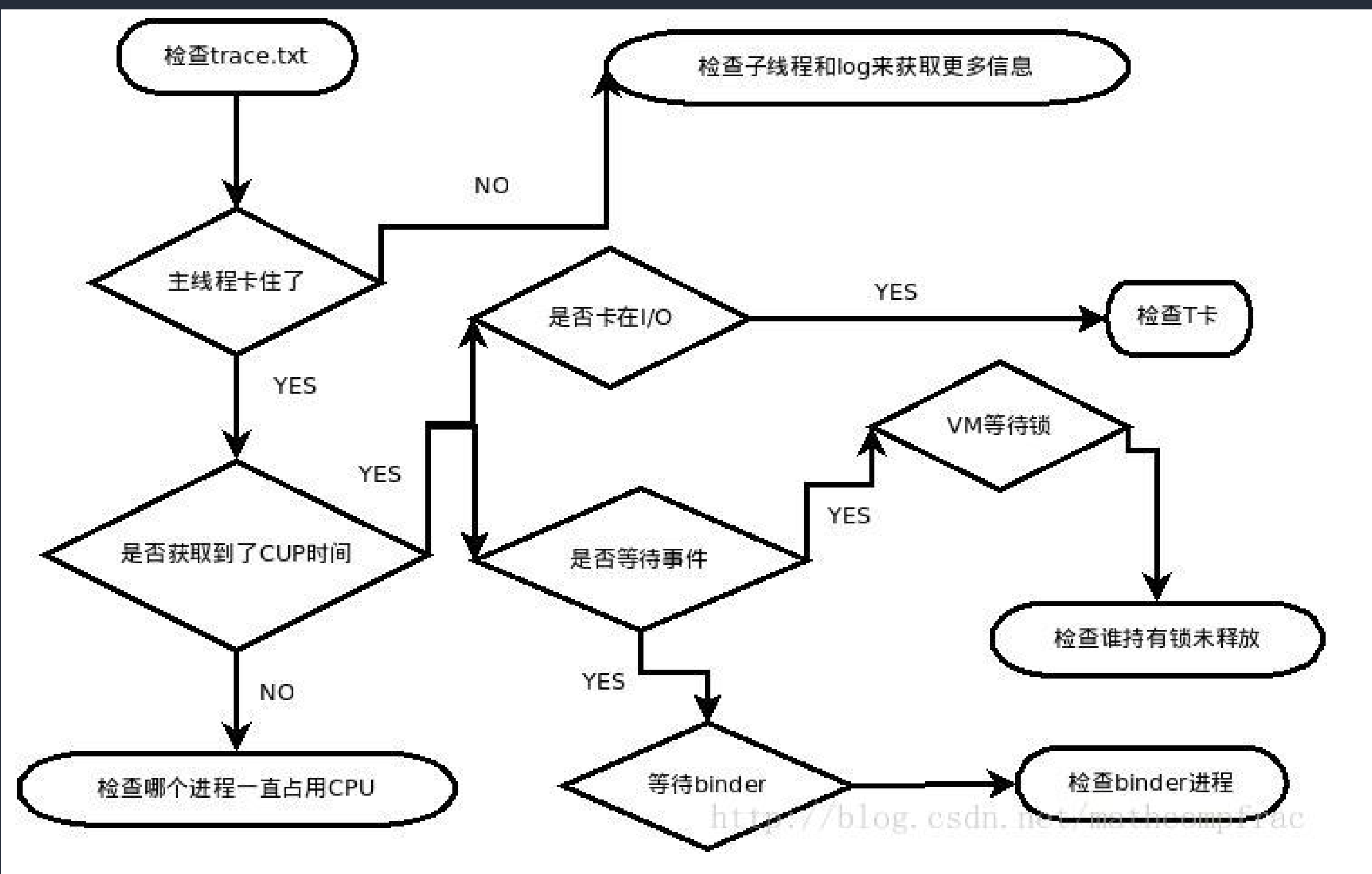
    /* these match up with JDWP values */
    THREAD_ZOMBIE         = 0,           /* TERMINATED */
    THREAD_RUNNING        = 1,           /* RUNNABLE or running now */
    THREAD_TIMED_WAIT     = 2,           /* TIMED_WAITING in Object.wait() */
    THREAD_MONITOR        = 3,           /* BLOCKED on a monitor */
    THREAD_WAIT           = 4,           /* WAITING in Object.wait() */
    /* non-JDWP states */
    THREAD_INITIALIZING   = 5,           /* allocated, not yet running */
    THREAD_STARTING       = 6,           /* started, not yet on thread list */
    THREAD_NATIVE         = 7,           /* off in a JNI native method */
    THREAD_VMWAIT         = 8,           /* waiting on a VM resource */
    THREAD_SUSPENDED      = 9,           /* suspended, usually by GC or debugger */
};
```

网址:

<https://android.googlesource.com/platform/dalvik2/+master/vm/Thread.h>

ANR 程序无响应

DEBUG流程



参考资料：

ANR分析解决方法：

<http://www.cnblogs.com/purediy/p/3225060.html>

死锁ANR：

<http://blog.csdn.net/oujunli/article/details/9102101>

App Crash

Crash就是由于代码异常而导致App非正常退出现象，也就是我们常说的『崩溃』

Android中通常会有以下两种Crash:

- Java Crash
- Native Crash (NDK Crash)

App Crash

Logcat 日志信息:

1. System进程的crash信息:

- 开头 ***** FATAL EXCEPTION IN SYSTEM PROCESS** [线程名];
- 接着输出发生crash时的调用栈信息;

2. App进程crash信息:

开头 **FATAL EXCEPTION:** [线程名];

紧接着 **Process:** [进程名], **PID:** [进程id];

最后输出发生crash时的调用栈信息。

App Crash —— Java Crash

Java Crash 的特点：

1. 这类错误一般由Java层代码触发
2. 一般情况下程序出错时会弹出提示框，JVM会退出
3. 一般Crash工具都能捕获

很抱歉，“AndroidCrashDemo”已停止运行。

确定

通过logcat查看Error级别日志，就可以完整看到打印出来的堆栈信息，我们找到『Caused by』信息：

```
Caused by: java.lang.NullPointerException: Attempt to invoke virtual method 'void
android.widget.Button.setOnClickListener(android.view.View$OnClickListener)' on a null object reference at
com.devilwwj.androidcrashdemo.MainActivity.onCreate(MainActivity.java:18)
```

App Crash -- Java Crash

解决方案：通过 `UncaughtExceptionHandler` 来捕获并记录异常信息

1. 自定义Handler, 继承 `UncaughtExceptionHandler`
2. Application类中进行初始化

App Crash -- Native Crash

Android中最头痛的就是Native层的异常，有时候拿到Log中堆栈信息也不一定能解决问题，例如使用了第三方的.so库

Native Crash 的特点：

1. 出错时界面不会弹出提示框（Android 5.0一下）
2. 出错时弹出提示框提醒程序崩溃（Android 5.0以上）
3. 程序直接闪退，显示系统桌面
4. 一般有C++层代码错误引起
5. 大部分Crash工具无法捕获

App Crash -- Native Crash

Native Crash 常见当然错误类型:

1. 初始化错误
2. 地址访问错误
3. 内存泄漏
4. 参数错误
5. 堆栈溢出
6. 类型转换错误
7. 数字除0错误

App Crash -- Native Crash

```
I/DEBUG: *** **
```

I/DEBUG: Build fingerprint: 'generic/vbox86p/vbox86p:4.4.4/KTU84P/eng.buildbot.20151118.000452:userdebug/test-keys'

I/DEBUG: Revision: '0'

I/DEBUG: pid: 1316, tid: 1316, name: evilwwj.jnidemo >>> com.devilwwj.jnidemo <<<

I/DEBUG: signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 00000000

I/DEBUG: eax 55300019 ebx 00000001 ecx 00000000 edx 00000000

I/DEBUG: esi 9ed92d14 edi bfac317c

I/DEBUG: xcs 00000073 xds 0000007b xes 0000007b xfs 00000000 xss 0000007b

I/DEBUG: eip 955d1730 ebp bfac3198 esp bfac316c flags 00210202

I/DEBUG: backtrace:

#00 pc 00000730 /data/app-lib/com.devilwwj.jnidemo-1/libJNIDemo.so (Java_com_devilwwj_jnidemo_TestJNI_createANativeCrash)

#01 pc 0002a4ab /system/lib/libdvm.so (dvmPlatformInvoke+79)

#02 pc 00006797 [heap]

#03 pc 00086da2 /system/lib/libdvm.so (dvmCallJNIMethod(unsigned int const*, JValue*, Method const*, Thread*)+434)

#04 pc 0008b2b6 /system/lib/libdvm.so (dvmResolveNativeMethod(unsigned int const*, JValue*, Method const*, Thread*)+326)

#05 pc 001775b8 /system/lib/libdvm.so

#06 pc 00005d0b <unknown>

#07 pc 0003b962 /system/lib/libdvm.so (dvmMterpStd(Thread*)+66)

#08 pc 00037029 /system/lib/libdvm.so (dvmInterpret(Thread*, Method const*, JValue*)+217)

#09 pc 000bc1c6 /system/lib/libdvm.so (dvmInvokeMethod(Object*, Method const*, ArrayObject*, ArrayObject*, ClassObject*, bool)

#10 pc 000d1b20 /system/lib/libdvm.so (Dalvik_java_lang_reflect_Method_invokeNative(unsigned int const*, JValue*)+288)

#11 pc 001775b8 /system/lib/libdvm.so

#12 pc 00005d5f <unknown>

#13 pc 0003b962 /system/lib/libdvm.so (dvmMterpStd(Thread*)+66)

#14 pc 00037029 /system/lib/libdvm.so (dvmInterpret(Thread*, Method const*, JValue*)+217)

#15 pc 000bc1c6 /system/lib/libdvm.so (dvmInvokeMethod(Object*, Method const*, ArrayObject*, ArrayObject*, ClassObject*, bool)

#16 pc 000d1b20 /system/lib/libdvm.so (Dalvik_java_lang_reflect_Method_invokeNative(unsigned int const*, JValue*)+288)

#17 pc 001775b8 /system/lib/libdvm.so

#18 pc 00005eff <unknown>

#19 pc 0003b962 /system/lib/libdvm.so (dvmMterpStd(Thread*)+66)

#20 pc 00037029 /system/lib/libdvm.so (dvmInterpret(Thread*, Method const*, JValue*)+217)

#21 pc 000bd027 /system/lib/libdvm.so (dvmCallMethodV(Thread*, Method const*, Object*, bool, JValue*, char*)+759)

#22 pc 0007a70d /system/lib/libdvm.so (CallStaticVoidMethodV(JNIEnv*, jclass*, jmethodID*, char*)+100)

进程信息

错误信号

寄存器快照

堆栈信息

App Crash -- Native Crash

- 进程信息: `pid` 表示进程号, `tid` 表示线程号, `name` 表示进程名
- 错误信号: `signal 11` 表示信号的数字, `SIGSEGV` 表示信号的名字, `code 1` (`SEGV_MAPERR`) 表示出错代码, `fault addr 00000000` 表示出错的地址
- 寄存器快照: 进程收到错误信号时保存下来的寄存器快照, 一共有15个寄存器
- 堆栈信息: `##00`表示栈顶, `##01`调用`#00`, 以此往下都是嵌套的调用关系, 直至到栈顶

问题定位: 第一个(栈顶)出现的动态连接库

App Crash -- Native Crash -- signal

传统 Unix 系统的信号定义和行为

所有的符合Unix规范（如POSIX）的系统都统一定义了SIGNAL的数量、含义和行为。作为Linux系统，Android自然不会更改SIGNAL的定义。在Android代码中，signal的定义一般在 `signum.h` (`prebuilt/linux-x86/toolchain/i686-linux-glibc2.7-4.4.3/sysroot/usr/include/bits/signum.h`) 中：

App Crash -- Native Crash -- signal

```
/* Signals. */
#define SIGHUP      1 /* Hangup (POSIX).  */
#define SIGINT      2 /* Interrupt (ANSI).  */
#define SIGQUIT     3 /* Quit (POSIX).  */
#define SIGILL      4 /* Illegal instruction (ANSI).  */
#define SIGTRAP     5 /* Trace trap (POSIX).  */
#define SIGABRT     6 /* Abort (ANSI).  */
#define SIGIOT      6 /* IOT trap (4.2 BSD).  */
#define SIGBUS      7 /* BUS error (4.2 BSD).  */
#define SIGFPE      8 /* Floating-point exception (ANSI).  */
#define SIGKILL     9 /* Kill, unblockable (POSIX).  */
#define SIGUSR1    10 /* User-defined signal 1 (POSIX).  */
#define SIGSEGV    11 /* Segmentation violation (ANSI).  */
#define SIGUSR2    12 /* User-defined signal 2 (POSIX).  */
#define SIGPIPE    13 /* Broken pipe (POSIX).  */
#define SIGALRM    14 /* Alarm clock (POSIX).  */
#define SIGTERM    15 /* Termination (ANSI).  */
#define SIGSTKFLT  16 /* Stack fault.  */
#define SIGCHLD    SIGCHLD /* Same as SIGCHLD (System V).  */
#define SIGCHLD    17 /* Child status has changed (POSIX).  */
#define SIGCONT    18 /* Continue (POSIX).  */
#define SIGSTOP    19 /* Stop, unblockable (POSIX).  */
#define SIGTSTP    20 /* Keyboard stop (POSIX).  */
#define SIGTTIN    21 /* Background read from tty (POSIX).  */
#define SIGTTOU    22 /* Background write to tty (POSIX).  */
#define SIGURG     23 /* Urgent condition on socket (4.2 BSD).  */
#define SIGXCPU    24 /* CPU limit exceeded (4.2 BSD).  */
#define SIGXFSZ    25 /* File size limit exceeded (4.2 BSD).  */
#define SIGVTALRM  26 /* Virtual alarm clock (4.2 BSD).  */
#define SIGPROF    27 /* Profiling alarm clock (4.2 BSD).  */
#define SIGWINCH   28 /* Window size change (4.3 BSD, Sun).  */
#define SIGPOLL    SIGIO /* Pollable event occurred (System V).  */
#define SIGIO      29 /* I/O now possible (4.2 BSD).  */
#define SIGPWR     30 /* Power failure restart (System V).  */
#define SIGSYS     31 /* Bad system call.  */
#define SIGUNUSED  31
```

App Crash -- Native Crash -- signal

Android 系统 信号处理的行为

我们知道，信号处理的行为是以进程级的。就是说不同的进程可以分别设置不同的信号处理方式而互不干扰。同一进程中的不同线程虽然可以设置不同的信号屏蔽字，但是却共享相同的信号处理方式（也就是说 在一个线程里改变信号处理方式，将作用于该进程中的所有线程）。

Android也是Linux系统。所以其信号处理方式不会有本质的改变。但是为了开发和调试的需要，android对一些信号的处理定义了额外的行为。下面是这些典型的信号在Android系统上的行为：

1. SIGQUIT （整型值为 3）

上面的表10-1显示，传统UNIX系统应用，对SIGQUIT信号的默认行为是 "终止 + CORE"。也就是产生core dump文件后，立即终于运行。

Android Dalvik应用收到该信号后，会打印改应用中所有线程的当前状态，并且并不是强制退出。这些状态通常保存在一个特定的叫做trace的文件中。一般的路径是/data/anr/trace.txt. 下面是一个典型的trace文件的内容：

App Crash -- Native Crash -- signal

```
1.----- pid 503 at 2011-11-21 21:59:12 -----
2.Cmd line: com.android.phone
3.
4.DALVIK THREADS:
5.(mutexes: tll=0 tsl=0 tscl=0 ghl=0 hwl=0 hwll=0)
6."main" prio=5 tid=1 NATIVE
7.  | group="main" sCount=1 dsCount=0 obj=0x400246a0 self=0x12770
8.  | sysTid=503 nice=0 sched=0/0 cgrp=default handle=-1342909272
9.  | schedstat=( 15165039025 12197235258 23068 ) utm=182 stm=1334 core=0
10. at android.os.MessageQueue.nativePollOnce(Native Method)
11. at android.os.MessageQueue.next(MessageQueue.java:119)
12. at android.os.Looper.loop(Looper.java:122)
13. at android.app.ActivityThread.main(ActivityThread.java:4134)
14. at java.lang.reflect.Method.invokeNative(Native Method)
15. at java.lang.reflect.Method.invoke(Method.java:491)
16. at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:841)
17. at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:599)
18. at dalvik.system.NativeStart.main(Native Method)
19.
20."Thread-29" prio=5 tid=24 WAIT
21.  | group="main" sCount=1 dsCount=0 obj=0x406f0d50 self=0x208c18
22.  | sysTid=1095 nice=0 sched=0/0 cgrp=default handle=2133304
23.  | schedstat=( 9521483 7029937750 720 ) utm=0 stm=0 core=0
24. at java.lang.Object.wait(Native Method)
25. - waiting on <0x406f0d50> (a com.motorola.android.telephony.cdma.OemCdmaTelephonyManager$Watchdog)
26. at java.lang.Object.wait(Object.java:361)
27. at com.motorola.android.telephony.cdma.OemCdmaTelephonyManager$Watchdog.run(OemCdmaTelephonyManager.java:229)
28. java.lang.Thread.run(Thread.java:1020)
29....
```

该文件包好很多重要的信息，可以说明在发生异常是，当前进程的状态，参考：

<http://bugly.qq.com/bbs/forum.php?mod=viewthread&tid=35>

App Crash -- Native Crash -- signal

2. 对于很多其他的异常信号（SIGILL, SIGABRT, SIGBUS, SIGFPE, SIGSEGV, SIGSTKFLT），Android进程在退出前，会生成 tombstone文件。记录该进程退出前的轨迹。一个典型的tombstone文件内容如下：

可以看出，它同样包含很多重要的信息（特别是 **stack**）来帮助我们查找异常的原因，参考：

[http://woshi.jp.github.io/2016/06/14/Android-NDK-](http://woshi.jp.github.io/2016/06/14/Android-NDK-Tombstone-Crash-%E5%88%86%E6%9E%90/)

[Tombstone-Crash-%E5%88%86%E6%9E%90/](http://woshi.jp.github.io/2016/06/14/Android-NDK-Tombstone-Crash-%E5%88%86%E6%9E%90/)

<http://www.cnblogs.com/CoderTian/p/5980426.html>

```
*** ** Build fingerprint:
' verizon/pasteur/pasteur:3.2.2/1.6.0_241/eng.drmn68.20111115.094123:eng/test-keys'
pid: 181, tid: 322 >>> /system/bin/mediaserver <<<
signal 8 (SIGFPE), code 0 (?), fault addr 000000b5
r0 00000000 r1 00000008 r2 ffffffff r3 00000020
r4 00000008 r5 00000000 r6 000000a5 r7 00000025
r8 662f9c00 r9 662f9c00 10 00000001 fp 00000000
ip aff17699 sp 4057f9dc lr aff176a7 pc aff0c684 cpsr 00000010
d0 6f762f6f69647502 d1 0000562202000000
d2 0000000400000300 d3 400120dc00000000
d4 0000000000000000 d5 0000000000000000
...
d30 0000000000000000 d31 0000000000000000
scr 20000010

#00 pc 0000c684 /system/lib/libc.so (kill)
#01 pc 000176a4 /system/lib/libc.so (raise)

libc base address: aff00000

stack:
4057f99c a2b6fd15 /system/lib/libstagefright.so
4057f9a0 00000000
4057f9a4 a2b6fe51 /system/lib/libstagefright.so
4057f9a8 000fb02c
...
4057f9d0 00000001
4057f9d4 a801e509 /system/lib/libutils.so
4057f9d8 4057fa14
#01 4057f9dc 00000008
4057f9e0 00000000
```

App Crash -- Native Crash -- signal

Android信号的产生和测试

我们看到，**多数signal的产生是由于某种内部错误**。我们在在开发过程中，当然也可以通过系统调用故意生成signal给某进程。主要的方法如果：

1. 在kernel里 使用 `kill_proc_info()`
2. 在native应用中 使用 `kill()` 或者`raise()`
3. java 应用中使用 `Procees.sendSignal()`等

但是在测试中，最简单的方法莫过于通过 `adb` 工具了，一个典型场景是shell命令：

```
adb shell kill -3 513
```


App Crash – Crash后系统处理流程

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。
1. 通过mUiHandler发送消息SHOW_ERROR_MSG，弹出crash对话框；
 2. 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；
 3. 当crash进程被杀，通过binder死亡通知，告知system_server进程来执行appDiedLocked()；
 4. 最后，执行清理应用相关的activity/service/ContentProvider/receiver组件信息。

App Crash -- Native Crash

Native Crash Log分析工具:

1. Google的[stack工具](#)
2. [Analyze android-ndk stack trace](#)

参考:

<http://code.google.com/p/android-ndk-stacktrace-analyzer/wiki/Usage>

http://source.android.com/porting/debugging_native.html

stack下载:

<http://bootloader.wdfiles.com/local--files/linux%3Aandroid%3Acrashlog/stack>

Android EventLog 含义

```
04-08 03:21:40.180 I/am_create_activity( 3410): [0,758871401,210,com.kunlun.mobdw.testin/.UnityPlayerActivity,android.intent.action.MAIN,NULL,NULL,268435456]
04-08 03:21:40.220 I/am_proc_start( 3410): [0,16607,10187,com.kunlun.mobdw.testin,activity,com.kunlun.mobdw.testin/.UnityPlayerActivity]
04-08 03:21:40.240 I/am_proc_bound( 3410): [0,16607,com.kunlun.mobdw.testin]
04-08 03:21:40.240 I/am_restart_activity( 3410): [0,758871401,210,com.kunlun.mobdw.testin/.UnityPlayerActivity]
04-08 03:21:40.360 I/am_on_resume_called(16607): [0,com.kunlun.mobdw.testin.UnityPlayerActivity]
04-08 03:21:40.470 I/am_activity_launch_time( 3410): [0,758871401,com.kunlun.mobdw.testin/.UnityPlayerActivity,290,9265]
04-08 03:22:25.990 I/am_kill ( 3410): [0,16607,com.kunlun.mobdw.testin,0,stop com.kunlun.mobdw.testin,clear data]
04-08 03:22:26.200 I/am_finish_activity( 3410): [0,758871401,210,com.kunlun.mobdw.testin/.UnityPlayerActivity,proc died without state saved]
04-08 03:22:26.200 I/am_finish_activity( 3410): [0,758871401,210,com.kunlun.mobdw.testin/.UnityPlayerActivity,proc died without state saved]
04-08 03:22:26.250 I/am_resume_activity( 3410): [0,946129220,1,com.huawei.android.launcher/.Launcher]
04-08 03:22:26.310 I/am_proc_start( 3410): [0,16898,10037,com.android.documentsui,broadcast,com.android.documentsui/.PackageReceiver]
04-08 03:22:26.390 I/am_proc_start( 3410): [0,16922,10070,com.huawei.android.totemweather,content provider,com.huawei.android.totemweather/.provider.WeatherPro
04-08 03:22:26.390 I/am_on_resume_called( 4211): [0,com.huawei.android.launcher.Launcher]
04-08 03:22:26.450 I/am_proc_start( 3410): [0,16944,10044,com.huawei.motionservice,content provider,com.huawei.motionservice/.common.HwMotionProvider]
04-08 03:22:26.450 I/am_proc_bound( 3410): [0,16898,com.android.documentsui]
04-08 03:22:26.470 I/am_proc_bound( 3410): [0,16922,com.huawei.android.totemweather]
```

在调试分析Android的过程中，比较常用的地查看EventLog，非常简洁明了地展现当前Activity各种状态，当然不至于此，比如还有window的信息。列举am常用的tags的含义。

Android EventLog 含义

➤ ActivityManager 相关

Android内存之VSS/RSS/PSS/USS

VSS – Virtual Set Size 虚拟耗用内存（包含共享库占用的内存）
RSS – Resident Set Size 实际使用物理内存（包含共享库占用的内存）
PSS – Proportional Set Size 实际使用的物理内存（比例分配共享库占用的内存）
USS – Unique Set Size 进程独自占用的物理内存（不包含共享库占用的内存）
一般来说内存占用大小有如下规律：VSS >= RSS >= PSS >= USS

TagName	格式	功能
am_finish_activity	User,Token,TaskID,ComponentName,Reason	
am_task_to_front	User,Task	
am_new_intent	User,Token,TaskID,ComponentName>Action,MIMEType,URI,Flags	
am_create_task	User,Task ID	
am_create_activity	User,Token,TaskID,ComponentName>Action,MIMEType,URI,Flags	
am_restart_activity	User,Token,TaskID,ComponentName	
am_resume_activity	User,Token,TaskID,ComponentName	
am_anr	User,pid,Package Name,Flags,reason	ANR
am_activity_launch_time	User,Token,ComponentName,time	
am_proc_bound	User,PID,ProcessName	
am_proc_died	User,PID,ProcessName	
am_failed_to_pause	User,Token,Wanting to pause,Currently pausing	
am_pause_activity	User,Token,ComponentName	
am_proc_start	User,PID,UID,ProcessName,Type,Component	
am_proc_bad	User,UID,ProcessName	
am_proc_good	User,UID,ProcessName	
am_low_memory	NumProcesses	Lru
am_destroy_activity	User,Token,TaskID,ComponentName,Reason	
am_relaunch_resume_activity	User,Token,TaskID,ComponentName	
am_relaunch_activity	User,Token,TaskID,ComponentName	
am_on_paused_called	User,ComponentName	
am_on_resume_called	User,ComponentName	
am_kill	User,PID,ProcessName,OomAdj,Reason	杀进程
am_broadcast_discard_filter	User,Broadcast>Action,ReceiverNumber,BroadcastFilter	
am_broadcast_discard_app	User,Broadcast>Action,ReceiverNumber,App	
am_create_service	User,ServiceRecord,Name,UID,PID	
am_destroy_service	User,ServiceRecord,PID	
am_process_crashed_too_much	User,Name,PID	
am_drop_process	PID	
am_service_crashed_too_much	User,Crash Count,ComponentName,PID	
am_schedule_service_restart	User,ComponentName,Time	
am_provider_lost_process	User,Package Name,UID,Name	
am_process_start_timeout	User,PID,UID,ProcessName	timeout
am_crash	User,PID,ProcessName,Flags,Exception,Message,File,Line	Crash
am_wtf	User,PID,ProcessName,Flags,Tag,Message	Wtf
am_switch_user	id	
am_activity_fully_drawn_time	User,Token,ComponentName,time	
am_focused_activity	User,ComponentName	
am_home_stack_moved	User,To Front,Top Stack Id,Focused Stack Id,Reason	
am_pre_boot	User,Package	
am_meminfo	Cached,Free,Zram,Kernel,Native	内存
am_pss	Pid,UID,ProcessName,Pss,Uss	进程

Android EventLog 含义

Tag可能使用的部分场景：

am_low_memory: 位于AMS.killAllBackgroundProcesses或者AMS.appDiedLocked, 记录当前Lru进程队列长度。

am_pss: 位于AMS.recordPssSampleLocked(

am_meminfo: 位于AMS.dumpApplicationMemoryUsage

am_proc_start:位于AMS.startProcessLocked, 启动进程

am_proc_bound:位于AMS.attachApplicationLocked

am_kill: 位于ProcessRecord.kill, 杀掉进程

am_anr: 位于AMS.appNotResponding

am_crash:位于AMS.handleApplicationCrashInner

am_wtf:位于AMS.handleApplicationWtf

am_activity_launch_time: 位于ActivityRecord.reportLaunchTimeLocked(), 后面两个参数分别是thisTime和 totalTime.

am_activity_fully_drawn_time:位于ActivityRecord.reportFullyDrawnLocked, 后面两个参数分别是thisTime和 totalTime

am_broadcast_discard_filter:位于BroadcastQueue.logBroadcastReceiverDiscardLocked

am_broadcast_discard_app:位于BroadcastQueue.logBroadcastReceiverDiscardLocked

Android EventLog 含义

Activity生命周期相关的方法(跟踪App运行轨迹):

am_on_resume_called: 位于AT.performResumeActivity

am_on_paused_called: 位于AT.performPauseActivity, performDestroyActivity

am_resume_activity: 位于AS.resumeTopActivityInnerLocked

am_pause_activity: 位于AS.startPausingLocked

am_finish_activity: 位于AS.finishActivityLocked, removeHistoryRecordsForAppLocked

am_destroy_activity: 位于AS.destroyActivityLocked

am_focused_activity: 位于AMS.setFocusedActivityLocked, clearFocusedActivity

am_restart_activity: 位于ASS.realStartActivityLocked

am_create_activity: 位于ASS.startActivityUncheckedLocked

am_new_intent: 位于ASS.startActivityUncheckedLocked

am_task_to_front: 位于AS.moveTaskToFrontLocked

Android EventLog 含义

➤ Power 相关

TagName	格式	功能
battery_level	level, voltage, temperature	
battery_status	status,health,present,plugged,technology	
battery_discharge	duration, minLevel,maxLevel	
power_sleep_requested	wakeLocksCleared	唤醒锁数量
power_screen_broadcast_send	wakelockCount	
power_screen_broadcast_done	on, broadcastDuration, wakelockCount	
power_screen_broadcast_stop	which,wakelockCount	系统还没进入 ready状态
power_screen_state	offOrOn, becauseOfUser, totalTouchDownTime, touchCycles	
power_partial_wake_state	releasedorAcquired, tag	

崩溃：AMS杀进程

杀进程的 EventLog:

- ◆ 04-08 03:30:17.140 I/am_kill (3410): [0,17032,com.kunlun.mobdw.testin,7,stop com.kunlun.mobdw.testin clear data]
[user pid processname oomadj (进程优先级) reason]
- ◆ 03-27 18:34:11.943 I/ActivityManager(16283): Force stopping package com.bankcomm.maidanba appid=10182 user=0
Fore stopping packageName appid user reason
- ◆ 03-30 13:34:17.230 I/ActivityManager(8545): Killing 8140:com.bankcomm.maidanba/u0a2042 (adj 0): user request after error
- ◆ processes xxx at adjustment 1 //杀adj=1的进程

崩溃：AMS杀进程场景

1. force-stop: 一般出现一个reason来更详细的说明触发force-stop的原因

方法	reason	含义
AMS.forceStopPackage	from pid <code>callingPid</code>	
AMS.finishUserStop	finish user	
AMS.clearApplicationUserData	clear data	
AMS.broadcastIntentLocked	storage unmount	
AMS.finishBooting	query restart	
AMS.finishInstrumentationLocked	finished inst	evenPersistent
AMS.setDebugApp	set debug app	evenPersistent
AMS.startInstrumentation	start inst	evenPersistent
PKMS.deletePackageLI	uninstall pkg	
PKMS.movePackageInternal	move pkg	
PKMS.replaceSystemPackageLI	replace sys pkg	
PKMS.scanPackageDirtyLI	replace pkg	
PKMS.scanPackageDirtyLI	update lib	
PKMS.setApplicationHiddenSettingAsUser	hiding pkg	
MountService.killMediaProvider	vold reset	

崩溃：AMS杀进程(am_kill)场景

2. 异常杀进程

方法	reason	含义
appNotResponding	anr	ANR
appNotResponding	bg anr	ANR
handleAppCrashLocked	crash	CRASH
crashApplication	crash	CRASH
processStartTimedOutLocked	start timeout	
processContentProviderPublishTimedOutLocked	timeout publishing content providers	
removeDyingProviderLocked	depends on provider <code>cpr.name</code> in dying proc <code>processName</code>	

崩溃：AMS杀进程(am_kill)场景

3. 主动杀进程

方法	reason	含义
forceStopPackageLocked	stop user <code>userId</code>	
forceStopPackageLocked	stop <code>packageName</code>	
killBackgroundProcesses	kill background	
killAllBackgroundProcesses	kill all background	
killAppAtUsersRequest	user request after error	FORCE_CLOSE
killUid	kill uid	PERMISSION
killUid	Permission related app op changed	PERMISSION
killProcessesBelowAdj	setPermissionEnforcement	
killApplicationProcess	-	直接杀
killPids	Free memory	
killPids	Unknown	
killPids	自定义	调用者自定义

崩溃：AMS杀进程(am_kill)场景

4. 调度杀进程

方法	reason	含义
cleanUpRemovedTaskLocked	remove task	
attachApplicationLocked	error during init	
systemReady	system update done	
getProcessRecordLocked	lastCachedPss k from cached	
performIdleMaintenance	idle maint (pss lastPss from initialIdlePss)	
checkExcessivePowerUsageLocked	excessive wake held	
checkExcessivePowerUsageLocked	excessive cpu	
scheduleCrash	scheduleCrash for message failed	

崩溃：AMS杀进程(am_kill)场景

4. 其他情况杀进程

方法	reason	含义
cleanUpRemovedTaskLocked	remove task	
attachApplicationLocked	error during init	
systemReady	system update done	
getProcessRecordLocked	lastCachedPss k from cached	
performIdleMaintenance	idle maint (pss lastPss from initialIdlePss)	
checkExcessivePowerUsageLocked	excessive wake held	
checkExcessivePowerUsageLocked	excessive cpu	
scheduleCrash	scheduleCrash for message failed	

崩溃：AMS杀进程(am_kill) 方式

- 以上介绍的所有杀进程都是调用ProcessRecord.kill()方法，必然会输出相应的EventLog
- 那么哪些场景的杀进程不会输出log呢：
 1. Process.killProcess(int pid) //可杀任何指定进程, 或者直接发signal
 2. adb shell kill -9 <pid> //可杀任何指定的进程
 3. 直接lmk杀进程

FATAL EXCEPTION: Multi dex installation failed

```
04-18 17:17:01.231 E/AndroidRuntime(15315): FATAL EXCEPTION: main
04-18 17:17:01.231 E/AndroidRuntime(15315): Process: com.foundersc.app.xf, PID: 15315
04-
18 17:17:01.231 E/AndroidRuntime(15315): java.lang.RuntimeException: Unable to instantiate application com.hundsun.winner.application.base.WinnerApplication: java
.lang.RuntimeException: Multi dex installation failed (write failed: ENOSPC (No space left on device)).
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.app.LoadedApk.makeApplication(LoadedApk.java:507)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.app.ActivityThread.handleBindApplication(ActivityThread.java:4422)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.app.ActivityThread.access$1500(ActivityThread.java:138)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1259)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.os.Handler.dispatchMessage(Handler.java:102)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.os.Looper.loop(Looper.java:136)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.app.ActivityThread.main(ActivityThread.java:5122)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at java.lang.reflect.Method.invokeNative(Native Method)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at java.lang.reflect.Method.invoke(Method.java:515)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:786)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:602)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at dalvik.system.NativeStart.main(Native Method)
04-18 17:17:01.231 E/AndroidRuntime(15315): Caused by: java.lang.RuntimeException: Multi dex installation failed (write failed: ENOSPC (No space left on device)).
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.support.multidex.MultiDex.install(MultiDex.java:178)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at com.hundsun.winner.application.base.WinnerApplication.attachBaseContext(WinnerApplication.java:239)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.app.Application.attach(Application.java:181)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.app.Instrumentation.newApplication(Instrumentation.java:993)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.app.Instrumentation.newApplication(Instrumentation.java:977)
04-18 17:17:01.231 E/AndroidRuntime(15315):   at android.app.LoadedApk.makeApplication(LoadedApk.java:502)
04-18 17:17:01.231 E/AndroidRuntime(15315):   ... 11 more
```

启动失败：启动时加载dex文件，设备空间不足导致

内存泄漏: Window Leaked

```
03-30 15:53:49.398 E/WindowManager(24876): android.view.WindowLeaked: Activity
cn.com.hzb.mobilebank.per.activity.user.LoginActivity has leaked window
com.android.internal.policy.impl.PhoneWindow$DecorView{179bbba7 V.E..... R.....D 0,0-960,1866} that was originally added here
03-30 15:53:49.398 E/WindowManager(24876):      at android.view.ViewRootImpl.<init>(ViewRootImpl.java:390)
03-30 15:53:49.398 E/WindowManager(24876):      at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:271)
03-30 15:53:49.398 E/WindowManager(24876):      at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:85)
03-30 15:53:49.398 E/WindowManager(24876):      at android.app.Dialog.show(Dialog.java:337)
03-30 15:53:49.398 E/WindowManager(24876):      at cn.com.hzb.mobilebank.per.activity.user.a.handleMessage(SourceFile)
03-30 15:53:49.398 E/WindowManager(24876):      at android.os.Handler.dispatchMessage(Handler.java:102)
03-30 15:53:49.398 E/WindowManager(24876):      at android.os.Looper.loop(Looper.java:135)
03-30 15:53:49.398 E/WindowManager(24876):      at android.app.ActivityThread.main(ActivityThread.java:5605)
03-30 15:53:49.398 E/WindowManager(24876):      at java.lang.reflect.Method.invoke(Native Method)
03-30 15:53:49.398 E/WindowManager(24876):      at java.lang.reflect.Method.invoke(Method.java:372)
03-30 15:53:49.398 E/WindowManager(24876):      at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:950)
03-30 15:53:49.398 E/WindowManager(24876):      at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:745)
```

Activity内存泄漏

Log关键字: FAILED BINDER TRANSACTION (TransactionTooLargeException)

03-30 01:14:36.006 E/JavaBinder(1141): !!! FAILED BINDER TRANSACTION !!!

现象: 点击“下一步”到下个页面

原因: Activity之间通过intent传递大量数据, 导致新Activity无法启动。

解决方法: 改变数据传输方式

- 通过Application
- 使用单例
- 静态成员
- 持久化 (sqlite、share preference、file等)

```
Choreographer: Skipped 60 frames! The application may be doing too much work on its main thread.
```

当跳帧数大于设置的SKIPPED_FRAME_WARNING_LIMIT（默认为30） 值时会在当前进程输出这个log

翻译：有很多帧被跳过了，代码耗费了过长的时间进行处理

原因：在UI线程处理了耗时任务

现象：界面卡顿、界面崩溃（如果界面有大量图片操作，内存消耗太多，同时伴有系GC_FOR_ALLOC）

解决方案：

- 把业务放在子线程或者中去完成，通过runOnUiThread方法来实现页面更新
- AsyncTask处理耗时操作



02

Part Two

真机调试

1. **线上预约**: Android目前官网预约机型有商务微信预约和官网在线预约两种方式, 微信预约在企业号-销售管理-机型预约, iOS目前可以支持远程调试的机器都已经上线官网, 限于技术攻克原因, 没有任何其他预约渠道。
2. **线下预约**: 目前支持Android深度兼容客户, 每次测试赠送一次, 五个机器, 两个小时。海外深度兼容客户, 每次测试赠送一次, 三个机器, 两个小时。线下预约均需要提前一天进行预约, 发送邮件给yuting@testin.cn, 有预约模板。线下预约有线下的单独平台, 未与付费平台打通, 时间只做计时, 不做付费。
3. **iOS真机界面卡顿**: iOS远程调试在网络正常的前提下依旧比较卡的原因, 不是网络原因, 也不是卡, iOS的系统比较封闭, 操作指令需要经过三层转发
4. **iOS签名**: iOS远程调试需要使用企业证书签名的包或者加下udid, udid在机器上可以获取

远程真机 - 设备筛选

Testin 企业版

项目 服务地图

3.0项目生产... 三

测试管理

产品开发生阶段

自动化功能测试

自动化兼容测试

自动化拨测

用例测试

远程真机测试

应用安全扫描

产品发版阶段

专家用例测试

Bug探索

兼容测试

产品运营阶段

A/B测试

自动化管理

应用管理

用例管理

Bug管理


3.0项目生产环境 > 测试管理 > 远程真机测试

剩余时长74851916分钟 查看调试记录

谷歌 Android 8.0 专区 所有手机

输入品牌机型查找设备 查询

空闲设备



Google Pixel XL


品牌：谷歌

型号：Pixel XL

系统：Android 8.0.0

分辨率：1440*2560

忙碌 占用中



Google Pixel


品牌：谷歌

型号：Pixel

系统：Android 8.0.0

分辨率：1080*1920

忙碌 占用中



Google Pixel


品牌：谷歌

型号：Pixel

系统：Android 8.0.0

分辨率：1080*1920

忙碌 占用中



Google Pixel


品牌：谷歌

型号：Pixel

系统：Android 8.0.0

分辨率：1080*1920

忙碌 占用中



谷歌 Nexus 5X


品牌：谷歌

型号：Nexus 5X

系统：Android 8.0.0

分辨率：1080*1920

忙碌 占用中



谷歌 Nexus 5X


品牌：谷歌

型号：Nexus 5X

系统：Android 8.0.0

分辨率：1080*1920

忙碌 占用中



谷歌 Nexus 5X


品牌：谷歌

型号：Nexus 5X

系统：Android 8.0.0

分辨率：1080*1920

忙碌 占用中



Google Pixel XL


品牌：谷歌

型号：Pixel XL

系统：Android 8.0.0

分辨率：1440*2560

忙碌 占用中



谷歌 Nexus 6P


品牌：谷歌

型号：Nexus 6P

系统：Android 8.0.0

分辨率：1440*2560

忙碌 占用中



谷歌 Nexus 6P


品牌：谷歌

型号：Nexus 6P

系统：Android 8.0.0

分辨率：1440*2560

忙碌 占用中



谷歌 Nexus 6P


品牌：谷歌

型号：Nexus 6P

系统：Android 8.0.0

分辨率：1440*2560

忙碌 占用中



Google Pixel C

品牌：谷歌

型号：Pixel C

系统：Android 8.0.0

分辨率：1800*2560

忙碌 占用中

在线咨询

Testin

Testin 企业版

冯建峰

三星Ga...

流畅

应用 截图 adb调试 快捷操作

307.15KB/s | 已使用: 0分钟4秒

停止使用

15:59

10月 31日, 星期二

+

+

🔍

🎤

📷

🌸

💾

✉️

照相机

相册

备份与恢复

电子邮件

☎️

👤

✉️

🌐

🗃️

电话

联系人

信息

互联网

应用程序

☰

🏠

⬅️

安装

📄+

尚未安装APP, 赶快上传吧~

上传安装

Logcat

Level Time PID TID Tag Text

获取日志 清空日志

📄

暂无信息

点击“获取”按钮, 查看Logcat日志信息
可以通过滚屏锁定当前日志信息

远程调试

网页调试：Testin企业版 -> 远程真机测试

- ① 设备实时屏幕
- ② 应用安装、启动、卸载
- ③ Logcat在线查看
- ④ 设备截图、保存
- ⑤ 快捷操作：打开网页、复制文本、获取文本、WIFI操作、打开设置页面、打开摄像头

远程调试

本地adb调试

连接方法：

- 复制连接命令：`adb connect debug.testin.cn:4082`
- 本地adb执行该命令
- 网页确认连接
- adb调试

使用说明：

- 本地安装好adb调试工具（需配置环境变量）
- windows环境打开本地cmd窗口，Mac或者Linux环境开启一个新的终端
- 在命令行输入上面的adb connect地址
- 在页面弹出的鉴权窗口中点击"确定"按钮
- 本地命令行中输入adb devices查看设备列表
- 开发工具中查看设备列表并开始debug调试（支持Eclipse和AndroidStudio远程调试）



03

Part Three

常见问题

加载问题

现象:

1. 界面数据加载失败
2. 界面图片加载失败
3. 界面显示网络问题
4. 界面卡顿

一般为客户服务器问题，建议客户请检查:

1. 服务器是否测试服务器，服务器资源配置
2. 在并发测试时服务器状态，是否有网络请求阻塞/服务器响应慢
3. App网络数据请求机制，是否启动后短时间多线程大量请求
4. App是否对图片进行了缓存处理



CPU、GPU、内存占用高

原因:

1. App界面网络请求较多（数据/图片等），导致后台服务（线程）大量网络请求，再获取到数据后进行数据处理, CPU和内存占用会比较大
2. 图片临时下载后绘制显示，GPU占用会比较大
3. 后台服务（线程）有耗时任务

建议:

1. 数据分时顺序下载，限制后台数据处理的线程数量，采用线程池而非临时创建线程
2. 降低网络请求次数，如果多次请求的数据，是否可以合并为一次请求
3. 固定的图片建议在app打包时直接设置
4. 使用缓存（如图片）

Android 内存优化

内存占用高导致的问题:

1. 内存泄漏导致OOM崩溃
2. 界面卡顿, 影响用户体验
3. 高内存耗电, 容易杀

常见内存泄漏和高内存的原因:

1. 慎重使用static变量
2. 长周期内部类、匿名内部类长时间持有外部类引用导致相关资源无法释放
(Handler或者内部线程等)
3. BitMap导致内存溢出
4. 数据库、文件流等没有关闭
5. 监听器、广播注册后没有及时注销
6. Adapter没有使用convertView
7. 字符串拼接尽量使用StringBuilder或者StringBuffer
8. 避免内存抖动, 例如不要在onDraw中创建对象。
9. 界面不可见时, 停止动画和相关线程
10. WebView引起的内存泄漏

Android 内存优化

如何进行内存优化：

1. 使用高性能编程，降低App运行的内存占用
2. 消除内存泄漏：
 - 使用静态变量的时候要小心，尤其要注意Activity/Service等大对象的传值。在单例模式中能用ApplicationContext的都用ApplicationContext，或者把聚合关系改成依赖关系，不在单例对象中持有Context引用；
 - 养成良好的代码习惯。注册/反注册要成对出现，Activity和Service对象中避免使用非静态内部类/匿名内部类，除非十分清楚引用关系；
 - 使用多线程的时候留意线程存活时间。尽量将聚合关系改成依赖关系，减少线程对象持有大对象的时间；
 - 在使用xxxStream, SQLiteDatabase, Cursor类的时候要注意释放资源。使用Timer, TimerTask的时候要记得取消任务。Bitmap在使用结束后要记得recycler()。

Android 内存优化

避免内存泄漏之官方推荐:

In summary, to avoid context-related memory leaks, remember the following:

1. Do not keep long-lived references to a context-activity (a reference to an activity should have the same life cycle as the activity itself)
2. Try using the context-application instead of a context-activity
3. Avoid non-static inner classes in an activity if you don't control their life cycle, use a static inner class and make a weak reference to the activity inside

And remember that a garbage collector is not an insurance against memory leaks.
Last but not least, we try to make such leaks harder to make happen whenever we can.

Android 内存优化

工具:

1. Android Monitor
2. Allocation Tracker
3. MAT
4. LeakCanary 检测内存泄漏