# TASK

ublic static final int DEBUG\_ENABLE\_DEBUGGER

开机日志

Dgb调试C

# 行业标准

LOG的抓取方式：

拨号盘 里面输入 \*#\*#284#\*#\*，等待约1-2分钟；进入到手机的文件管理器，路径的正确打开方式为：MIUI→debug\_log 文件夹下，有一个包含 bugreport 名字的压缩包，这个就是需要的log。为了保证有效Log不被冲掉，请在遇到Bug时立即抓取Log

<http://www.miui.com/thread-14130379-1-1.html>

# 截屏

Adsf

获取当前activity的名字

adb shell screencap -p /sdcard/findui/c2.png

adb pull /sdcard/findui

# 抓到pc端

adb shell logcat -b main -b system -b radio -b events -v time > logcat0630.log

adb bugreport > bugreport0622.txt

adb pull /data/anr

# lint

代码质量检查

# Debug

## 调试方法

### 串口调试

咖啡色接gnd口就好

打开CRT，选择波特率为115200，

串口只能链接一个！不支持多个会话

后台服务的话，加上&符号

logcat –v time &

dmesg &

logcat -v time&

######################################USB-USB-IN

logcat -b main -b system -b radio -b events -v time

### ADB

搜索关键字：\* FOR %%i IN (\*.apk)

git，googel，

目标文件：apk

日志加入

#扫描当前apk文件夹

@REM 循环安装本目录下的APK文件

FOR %%i IN (\*.apk) DO (

ECHO 正在安装：%%i

adb install -r %%i

)

#### APK安装

ADB APK安装

adb install

APK 命令可以用adb install [-lrtsd] <file>或者adb install-multiple [-lrtsdp] <file...>，adb install-multiple表示批量安装。  
参数介绍：

* -l：锁定该程序
* -r：可以覆盖已有的应用，保留数据和缓存文件
* -t：允许测试该应用
* -s：安装在SD卡中
* -d：允许降低版本安装
* -p：部分应用程序安装

1. 无需数据线

且可解决部分机器在方法1时出现的“unable to connect to 192.168.1.199:5555”错误

在android设备上安装 “终端模拟器”等类似shell命令工具，使用下面命令（**需root权限**）：

TCP/IP方式：

setprop service.adb.tcp.port 5555

stop adbd

start adbd

usb方式：

setprop service.adb.tcp.port -1

stop adbd

start adbd

<https://twiceyuan.com/2014/11/21/Android%E7%BD%91%E7%BB%9C%E8%B0%83%E8%AF%95%E5%BC%80%E5%90%AF%E6%96%B9%E6%B3%95/>

#### 常见问题

##### 三星手机5.1调试

1. ~~左上角插入手机卡2才有用（待验证）~~

　　1、点开设置—更多—关于设备-》软件信息-》版本号。点内部版本号7到8次左右。

　　2、点过之后返回，就会看到开发者选项，这个勾选USB调试就行啦！

　　3、如果不能安装驱动，在三星官网下载PC套件Samsung kies(包含了链接所需要的驱动)。点击打开。【[PC套件下载地址](http://www.samsung.com/cn/support/usefulsoftware/KIES/)】http://www.samsung.com/cn/support/usefulsoftware/KIES/

4、那个KIES软件安装到电脑上，通过数据线连接电脑（链接模式为媒体设备MTP）即可。

实在连不上可以尝试重启一下

提示：若要解除阻止并通过USB数据线连接至+其他设备，请插入USIM&oq=若要解除阻止并通过USB数据线连接至+其他设备，请插入USIM

##### adb无法启动

现象1：端口被占用，Unable to create Debug Bridge: Unable to start adb server: error: could not install \*smartsocket\* listener: cannot bind to 127.0.0.1:5037

现象2：unknown host service

解决办法：kill 端口占用的进程，常见的是手机助手占用了。

Netstat –ano | findstr 5037



[Android Debugger port : Connection refused:](http://www.jianshu.com/p/96aaed625ad0) adb kill-server; adb start-server

Adb启动ok，无法发现设备

Adb是32位的，设备是64位的，请用android stdio的最新tool的adb来使用就好了。

##### MIX开启 USB安装/USB调试（安全设置）

需要登录小米账号，[more](http://www.miui.com/thread-5711795-1-1.html)

然后才可以执行adb shell。

参考：

[ANDROID开发常用的ADB命令整理kg](http://www.keyguan.com/blog/2017/02/11/android%E5%BC%80%E5%8F%91%E5%B8%B8%E7%94%A8%E7%9A%84adb%E5%91%BD%E4%BB%A4%E6%95%B4%E7%90%86/)

串口adb

Wifi-adb调试

Win-cmd:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| cls | Clear screen |  |  |  |
| Findstr | 正则匹配 |  |  |  |
|  |  |  |  |  |

#### 源码分析

ADB是Android debug bridge的缩写，它使用PC机可以通过USB或网络与android设备通讯。

源码：system/core/adb

先来看下编译脚本

##### Android.mk

可以看到，最终会有3个执行文件被生成，adbd和两个adb程序。adbd是手机终的守护进程；adb一个是windows、linux、darwin或freebsd运行的程序，另一个是目标机上运行的程序。

宏ADB\_HOST用于区分是PC端程序还是目标机端的程序。宏ADB\_HOST\_ON\_TARGET用于区分adb程序是否是在目标机上运行。这3个程序使用的是同一份源码，在内部，使用这些宏来区别不同的程序。

程序的入口在adb.c的main()函数：

##### 入口adb. main()函数

1. **int** main(**int** argc, **char** \*\*argv)
2. {
3. #if ADB\_HOST
4. adb\_sysdeps\_init();
5. adb\_trace\_init();
6. D("Handling commandline()\n");
7. **return** adb\_commandline(argc - 1, argv + 1);
8. #else
9. /\* If adbd runs inside the emulator this will enable adb tracing via
10. \* adb-debug qemud service in the emulator. \*/
11. adb\_qemu\_trace\_init();
12. **if**((argc > 1) && (!strcmp(argv[1],"recovery"))) {
13. adb\_device\_banner = "recovery";
14. recovery\_mode = 1;
15. }
17. start\_device\_log();
18. D("Handling main()\n");
19. **return** **adb\_main(0, DEFAULT\_ADB\_PORT)**;
20. #endif
21. }

先来看adbd程序，此时宏的设置是ADB\_HOST=0。上面代码中start\_device\_log()是log的初始化操作，可以重定向输出的log信息，接着进入adb\_main()函数。先来看下它的参数DEFAULT\_ADB\_PORT：

1. #if ADB\_HOST\_ON\_TARGET
2. /\* adb and adbd are coexisting on the target, so use 5038 for adb
3. \* to avoid conflicting with adbd's usage of 5037
4. \*/
5. #  define DEFAULT\_ADB\_PORT 5038
6. #else
7. #  define DEFAULT\_ADB\_PORT 5037
8. #endif

如果是目标机程序，它的值是5038，否则它的值是5037。这里没有定义ADB\_HOST\_ON\_TARGET， 所以它是5037。

##### adb\_main()的源代码如下：

1. **int** adb\_main(**int** is\_daemon, **int** server\_port)
2. {
3. #if !ADB\_HOST
4. **int** port;
5. **char** value[PROPERTY\_VALUE\_MAX];
7. umask(000);
8. #endif
10. atexit(adb\_cleanup);
11. #ifdef HAVE\_WIN32\_PROC
12. SetConsoleCtrlHandler( ctrlc\_handler, TRUE );
13. #elif defined(HAVE\_FORKEXEC)
14. // No SIGCHLD. Let the service subproc handle its children.
15. signal(SIGPIPE, SIG\_IGN);
16. #endif
18. init\_transport\_registration();
20. #if ADB\_HOST
21. HOST = 1;
22. usb\_vendors\_init();
23. usb\_init();
24. local\_init(DEFAULT\_ADB\_LOCAL\_TRANSPORT\_PORT);
25. adb\_auth\_init();
27. **char** local\_name[30];
28. build\_local\_name(local\_name, **sizeof**(local\_name), server\_port);
29. **if**(install\_listener(local\_name, "\*smartsocket\*", NULL)) {
30. exit(1);
31. }
32. #else
33. property\_get("ro.adb.secure", value, "0");
34. auth\_enabled = !strcmp(value, "1");
35. **if** (auth\_enabled)
36. adb\_auth\_init();
38. // Our external storage path may be different than apps, since
39. // we aren't able to bind mount after dropping root.
40. **const** **char**\* adb\_external\_storage = getenv("ADB\_EXTERNAL\_STORAGE");
41. **if** (NULL != adb\_external\_storage) {
42. setenv("EXTERNAL\_STORAGE", adb\_external\_storage, 1);
43. } **else** {
44. D("Warning: ADB\_EXTERNAL\_STORAGE is not set.  Leaving EXTERNAL\_STORAGE"
45. " unchanged.\n");
46. }
48. /\* don't listen on a port (default 5037) if running in secure mode \*/
49. /\* don't run as root if we are running in secure mode \*/
50. **if** (should\_drop\_privileges()) {
51. **struct** \_\_user\_cap\_header\_struct header;
52. **struct** \_\_user\_cap\_data\_struct cap;
54. **if** (prctl(PR\_SET\_KEEPCAPS, 1, 0, 0, 0) != 0) {
55. exit(1);
56. }
58. /\* add extra groups:
59. \*\* AID\_ADB to access the USB driver
60. \*\* AID\_LOG to read system logs (adb logcat)
61. \*\* AID\_INPUT to diagnose input issues (getevent)
62. \*\* AID\_INET to diagnose network issues (netcfg, ping)
63. \*\* AID\_GRAPHICS to access the frame buffer
64. \*\* AID\_NET\_BT and AID\_NET\_BT\_ADMIN to diagnose bluetooth (hcidump)
65. \*\* AID\_SDCARD\_R to allow reading from the SD card
66. \*\* AID\_SDCARD\_RW to allow writing to the SD card
67. \*\* AID\_MOUNT to allow unmounting the SD card before rebooting
68. \*\* AID\_NET\_BW\_STATS to read out qtaguid statistics
69. \*/
70. gid\_t groups[] = { AID\_ADB, AID\_LOG, AID\_INPUT, AID\_INET, AID\_GRAPHICS,
71. AID\_NET\_BT, AID\_NET\_BT\_ADMIN, AID\_SDCARD\_R, AID\_SDCARD\_RW,
72. AID\_MOUNT, AID\_NET\_BW\_STATS };
73. **if** (setgroups(**sizeof**(groups)/**sizeof**(groups[0]), groups) != 0) {
74. exit(1);
75. }
77. /\* then switch user and group to "shell" \*/
78. **if** (setgid(AID\_SHELL) != 0) {
79. exit(1);
80. }
81. **if** (setuid(AID\_SHELL) != 0) {
82. exit(1);
83. }
85. /\* set CAP\_SYS\_BOOT capability, so "adb reboot" will succeed \*/
86. header.version = \_LINUX\_CAPABILITY\_VERSION;
87. header.pid = 0;
88. cap.effective = cap.permitted = (1 << CAP\_SYS\_BOOT);
89. cap.inheritable = 0;
90. capset(&header, &cap);
92. D("Local port disabled\n");
93. } **else** {
94. **char** local\_name[30];
95. build\_local\_name(local\_name, **sizeof**(local\_name), server\_port);
96. **if**(install\_listener(local\_name, "\*smartsocket\*", NULL)) {
97. exit(1);
98. }
99. }
101. **int** usb = 0;
102. **if** (access(USB\_ADB\_PATH, F\_OK) == 0 || access(USB\_FFS\_ADB\_EP0, F\_OK) == 0) {
103. // listen on USB
104. usb\_init();
105. usb = 1;
106. }
108. // If one of these properties is set, also listen on that port
109. // If one of the properties isn't set and we couldn't listen on usb,
110. // listen on the default port.
111. property\_get("service.adb.tcp.port", value, "");
112. **if** (!value[0]) {
113. property\_get("persist.adb.tcp.port", value, "");
114. }
115. **if** (sscanf(value, "%d", &port) == 1 && port > 0) {
116. printf("using port=%d\n", port);
117. // listen on TCP port specified by service.adb.tcp.port property
118. local\_init(port);
119. } **else** **if** (!usb) {
120. // listen on default port
121. local\_init(DEFAULT\_ADB\_LOCAL\_TRANSPORT\_PORT);
122. }
124. D("adb\_main(): pre init\_jdwp()\n");
125. init\_jdwp();
126. D("adb\_main(): post init\_jdwp()\n");
127. #endif
129. **if** (is\_daemon)
130. {
131. // inform our parent that we are up and running.
132. #ifdef HAVE\_WIN32\_PROC
133. **DWORD**  count;
134. WriteFile( GetStdHandle( STD\_OUTPUT\_HANDLE ), "OK\n", 3, &count, NULL );
135. #elif defined(HAVE\_FORKEXEC)
136. fprintf(stderr, "OK\n");
137. #endif
138. start\_logging();
139. }
140. D("Event loop starting\n");
142. fdevent\_loop();
144. usb\_cleanup();
146. **return** 0;
147. }

(1) init\_transport\_registration()初始化fevent transport\_registration\_fde；

(2) 判断系统属性ro.adb.secure，目标板设置这个宏；

sagit:/ $ getprop | grep ro.adb.secure

[ro.adb.secure]: [1]

(3) 没有定义环境变量adb\_external\_storage；只有$EXTERNAL\_STORAGE

sagit:/ $ echo $EXTERNAL\_STORAGE

/sdcard

(4) should\_drop\_privileges()根据android编译环境should\_drop\_privileges返回不同的值，如果它的值是userdebug或eng，宏ALLOW\_ADBD\_ROOT的值被定义为1，执行install\_listener()，否则不会定义，这种情况下，由于adbd运行在root下，为保证它的安全性，它需要降级运行；

ifneq (,$(filter userdebug eng,$(TARGET\_BUILD\_VARIANT)))

LOCAL\_CFLAGS += -DALLOW\_ADBD\_ROOT=1

Endif

http://blog.csdn.net/xgbing/article/details/52058390

## Native奔溃调试

## [android native 内存泄露检查（libc.debug.malloc ）](http://blog.csdn.net/haima1998/article/details/51508947)

\*c c++代码，由于其特殊性质，没有虚拟机概念，内存则直接是由用户管理，比如申请，释放，都是需要用户主动去触发，如果用户出现使用了申请，但是用完之后，没有调用释放，则会引起内存泄露。这种叫真正意义的内存泄露，只有重启机子，才能恢复。

相对而已java端的内存泄露，指的是一个应用长期运行，导致相互引用，无法释放，GC没法回收，引起的有效内存越来越小，我们将此现象叫做，内存泄露，通过关闭此应用，重新打开即可恢复内存。因此看来，java内存泄露和c c ++ 的 还是有本质区别的。

java本身的虚拟机里面会关注对象的申请，释放，这些不需要用户直接注，java虚拟机通过管理机制，将调用c c++里面真正的malloc free 方法，封装起来，将java对象的生命周期和malloc free 进行关联，则可以保证在对象不使用的时候，内存紧张时，释放掉不再被引用的对象，GC回收就是在做这件事请。回到我们这节的主要内容，如何定位我们的c c++的内存泄露。

Android对内存的使用包括内存泄漏和内存越界，内存泄漏会导致系统内存减少，最终分配不到内存，这样大的程序就不能运行，甚至系统没有内存而崩溃。Android中kernel和应用程序都可能会有内存泄漏和越界。对于Java代码，在越界的时候虚拟机会加以检查并抛出异常。而对于C/C++代码，越界的时候就悄无声息地让程序出错或crash

### 调试手段

在adb shell之下:确保/system/lib/ libc\_malloc\_debug\_leak.so存在

root@zs600b:/ # stop

root@zs600b:/ # setprop libc.debug.malloc 1

root@zs600b:/ # start

执行成功之后，浏览器和cs-app都无法打开了。。。

指针溢出，野指针，堆内存指针释放多遍等问题如何调试？

Android这边打开bionic/libc/bionic/malloc\_debug\_common.c里的配置

如adb shell setprop persist.libc.debug.malloc 10 再重启

之后可以复现问题，抓bugreport来分析问题，当然要匹配symbols(out/target/product/project/symbols)

此时overhead会比较重，可能会有一些不预期的anr(出现ANR时请点击等待)，但不影响测试。

start & stop are tools under /system/bin (system.img)

### 常见问题

/system/bin/sh: Missing module libc\_malloc\_debug\_leak.so required for malloc debug level 1: dlopen failed: library "libc\_malloc\_debug\_leak.so" not found

userdebug才是自带so，否则自己手动push进去吧。。，。

可以看看eng或这user-debug版中的/system/lib/下是否有这两个文件，其中libc\_malloc\_debug\_qemu.so是模拟器用的。

### 复现场景

堆引起的异常(调用malloc申请的内存后使用不当引起的异常)：

1. 申请后多次释放 (double free)

2. 释放后又去使用 (used after free)

3. 使用越界 (比如申请了50节内存，结果在使用时多用了8字节的内存，这样就把后面的内存的内容踩坏，引起堆结构异常)

4. 释放时传给free()的地址不是malloc()申请的地址，比如：p = malloc(10); free(p + 5);

5. 内存泄露：申请内存后，忘记释放或某些代码路径没有释放

### 源码分析

此malloc的调试原理是：当系统发现我们有libc.debug.malloc的一些列配置成立时，此时系统会将malloc free 等方法，重新指向到 lib\_malloc\_debug.so里面的对应实现方法，lib\_malloc\_debug.so里面的方法，像比较而言，多了一些记录信息，将每次的申请时的地址，堆栈，so等信息记录下来，然后我们需要的时候，则通过工具ddms dump出来，进行分析每个申请的内存，是否正常的释放了，是否出现了内存泄露。

|  |
| --- |
| // The value of libc.debug.malloc.  #if !defined(LIBC\_STATIC)  static int g\_malloc\_debug\_level = 0;  #endif  // Initializes memory allocation framework once per process.  static void malloc\_init\_impl() {  const char\* so\_name = NULL;  // If debug level has not been set by memcheck option in the emulator,  // lets grab it from libc.debug.malloc system property.  if (g\_malloc\_debug\_level == 0 && \_\_system\_property\_get("libc.debug.malloc", env)) {  g\_malloc\_debug\_level = atoi(env);  }  // Choose the appropriate .so for the requested debug level.  switch (g\_malloc\_debug\_level) {  case 1:  case 5:  case 10:  so\_name = "libc\_malloc\_debug\_leak.so";  break;  }  // Load .so that implements the required malloc debugging functionality.  void\* malloc\_impl\_handle = dlopen(so\_name, RTLD\_LAZY);  // No need to init the dispatch table because we can only get  // here if debug level is 1, 5, 10, or 20.  static MallocDebug malloc\_dispatch\_table \_\_attribute\_\_((aligned(32)));  switch (g\_malloc\_debug\_level) {  case 1:  InitMalloc(malloc\_impl\_handle, &malloc\_dispatch\_table, "leak");  break;  }  } |

android系统启动流程

## framework debug 技巧

### 安装

拷贝文件到system/framework/;  
stop;  
start重启system\_server进程

### Java调试

其实整个调试过程非常简单：

* 打断点
* 跟踪代码（Step in/out/over等等）

### 打断点

在正确的进程的合适位置打断点:区别于普通app的进程，framwork的所在进程需要进行源码分析，比如ams是运行在system\_server中的，并且这些进程只能在root的机子调试（模拟器或者root真机）

### 跟踪代码

### 打印日志

-系统一般都会有debug标志，我们修改源码中的标志，打印就好，比如

修改后

-- Log.e("tag",logStr);

### REF

[android native 代码内存泄露 定位方案](http://www.jianshu.com/p/2b43abdd6647)

[[MTK软件原创] 如何调试malloc(堆越界)问题](http://bbs.16rd.com/thread-54815-1-1.html)

[more detail about adb shell start, stop and reboot](http://blog.csdn.net/johnnylq/article/details/6401531)

[android native 内存泄露检查（libc.debug.malloc ）](http://blog.csdn.net/haima1998/article/details/51508947)

# 日志分析logcat

Logcat是存到缓存到内存中的，英文

打印的都是 /dev/log

[Android6.0 Log的工作机制，源码分析](https://zhuanlan.zhihu.com/p/24372024)

adb logcat -c  
清除log缓存

adb bugreport  
查看bug报告

查看event节点，调试activity生命周期

logcat -b events -v time

logcat -b events -v time | grep am\_

查看slog

adb logcat -b system -v time | grep WindowManagerService

logcat -b system -v time | grep WindowManagerService > /sdcard/dlg.txt

cat /proc/kmsg 实时打印内核日志

内核的 dmesg 静态显示内核日志

## 日志抓取方案

Android日志主要分为kernel、radio、event、main这四种log。

dmesg

kernel log属于Linux内核的log ，可以通过读取/proc/kmsg或者通过串口来抓取。

adb 抓取kernel log的命令如下（需要有root权限）：

kernel.log抓取

adb shell rm -rf /sdcard/logs

adb shell mkdir /sdcard/logs

adb reboot

adb root

adb remount

adb shell

#dmesg > /sdcard/logs/kernel.log

cat /proc/kmsg > /sdcard/logs/kernel0324.log

adb shell

logcat -b main -b system -b radio -b events -v time > /sdcard/logs/logcat0324.log

adb shell

. hardware\_monitor.sh > /sdcard/logs/hardware0324.log

cp -rf /data/tombstones /sdcard/logs/

cp -rf /data/anr /sdcard/logs/

exit

adb pull /sdcard/logs

# 日志系统

打印5级堆栈

Debug.getCallers(5));

## TAG中心

ActivityManagerDebugConfig 分类法

类名/别名

子业务名

static final boolean TAG\_WITH\_CLASS\_NAME = false;

static final boolean APPEND\_CATEGORY\_NAME = false;

// Default log tag for the activity manager package.

static final String TAG\_AM = "ActivityManager";

static final String POSTFIX\_BROADCAST = (APPEND\_CATEGORY\_NAME) ? "\_Broadcast" : "";

ActivityManagerService：

private static final String TAG = TAG\_WITH\_CLASS\_NAME ? "ActivityManagerService" : TAG\_AM;

private static final String TAG\_BROADCAST = TAG + POSTFIX\_BROADCAST;

# C++debug

打印调用堆栈

#include <utils/CallStack.h>

void \_\_egl\_get\_backtrace(void)

{

ALOGD("%s start", \_\_func\_\_);

android::CallStack stack(\_\_func\_\_);

ALOGD("%s end", \_\_func\_\_);

}

在需要的地方，添加这个函数就可以了。记得include这个头文件。

# DEBUG ZGOTE

Zygote

/\*\* enable debugging over JDWP \*/

public static final int DEBUG\_ENABLE\_DEBUGGER = 1;

/\*\* enable JNI checks \*/

public static final int DEBUG\_ENABLE\_CHECKJNI = 1 << 1;

/\*\* enable Java programming language "assert" statements \*/

public static final int DEBUG\_ENABLE\_ASSERT = 1 << 2;

/\*\* disable the AOT compiler and JIT \*/

public static final int DEBUG\_ENABLE\_SAFEMODE = 1 << 3;

/\*\* Enable logging of third-party JNI activity. \*/

public static final int DEBUG\_ENABLE\_JNI\_LOGGING = 1 << 4;

/\*\* Force generation of native debugging information. \*/

public static final int DEBUG\_GENERATE\_DEBUG\_INFO = 1 << 5;

/\*\* Always use JIT-ed code. \*/

public static final int DEBUG\_ALWAYS\_JIT = 1 << 6;

/\*\* Make the code debuggable with turning off some optimizations. \*/

public static final int DEBUG\_NATIVE\_DEBUGGABLE = 1 << 7;

## debug.checkjn

if ("1".equals(SystemProperties.get("debug.checkjni"))) {

debugFlags |= Zygote.DEBUG\_ENABLE\_CHECKJNI;

}

String genDebugInfoProperty = SystemProperties.get("debug.generate-debug-info");

if ("true".equals(genDebugInfoProperty)) {

debugFlags |= Zygote.DEBUG\_GENERATE\_DEBUG\_INFO;

}

if ("1".equals(SystemProperties.get("debug.jni.logging"))) {

debugFlags |= Zygote.DEBUG\_ENABLE\_JNI\_LOGGING;

}

if ("1".equals(SystemProperties.get("debug.assert"))) {

debugFlags |= Zygote.DEBUG\_ENABLE\_ASSERT;

}

Trace和EventLog.writeEvent等系统监控原理

checkTime(startTime, "startProcess: returned from zygote!");

# 卡机日志

01-18 01:45:27.305 W/SystemServer( 577): System clock is before 1970; setting to 1970.

01-18 01:45:27.308 I/SystemServer( 577): Entered the Android system server!