# Android 调试初步

修改记录

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 说明 | 日期 | 作者 |
| V0.01 | 初始版本。 | 10年8月17日 | 徐申龙 |
|  |  |  |  |

内容目录

ADB 1

ADB shell简介 2

adb shell 无法启动 （insufficient permissions for device） 2

logcat 3

控制台 4

调试消息 4

Java 的 打印信息 4

打印堆栈 5

Trace 5

调试器 6

Java 调试器 6

C/C++ 调试器 6

debuggerd 服务 6

模拟器 7

在 Eclipse 浏览 Android 的源码 8

# 前言

为了方便，把Android SDK 中的tools目录或者源码编译后 out/host/linux-x86/bin的路径添加到环境变量 PATH 中，如：

$ tail ~/.bashrc

export PATH=${PATH}:~/android/platform/out/host/linux-x86/bin

# 

# ADB

***ADB*** 是 ***Andrdoi Debug Bridge*** （***Android*** 调试桥 ） 的简称，是***Android*** 调试的利器 。

***$adb shell #***进入***ADB shell***

***$adb devices #***列出挂接的设备，取得当前运行的模拟器***/***设备的实例状态

***$adb connect <host>:<port> - connect to a device via TCP/IP***

***$adb start-server #*启动*adb* 服务，一般如果在使用*adb*时，没有*adb* 服务，它会自动开启。 *$adb kill-server #*杀死*adb* 服务。**

***$adb bugreport #*获取设备的全部状态信息，在 *bug report* 中也写入此命令的输出。**

***$adb install [-r] [-s] [-l] xxx.apk # -r***参数会强制卸载老版本，安装新版本。 ***-s*** 表示安装到***SD Card***中。

***#'-l' means forward-lock the app***

***$adb uninstall [-k] xxx.apk #***卸载软件， ***-k*** 表示保留数据和缓存目录。

***$adb remount # /system***重新挂载为可读写。

***$adb push test.txt /data/test.txt #*** 把主机当前目录的***test.txt***复制到设备或模拟器上的***/tmp/test.txt***

***$adb pull /system/lib/libwebcore.so ./ #***把设备或模拟器上的文件***/system/lib/libwebcore.so***拷贝到主机 ***#***当前目录中。

***$adb get-serialno #***打印设备的端口名称。

***$ adb forward <local> <remote> # adb forward tcp:5555 tcp:8000*** 设置主机***5555***端口号转发到模 ***#***拟器或设备***8000***端口。

***$adb help #***打印 ***adb*** 帮组信息

***-d*** 选项 把子命令发送到***USB device***

***-e*** 选项 把自命令发送到 ***emulator***

***system/core/adb*** 代码， ***system/core/adb/SERVICES.txt***

## ADB shell简介

***ADB shell*** 提示符为“***#”***，下面是一些有用的命令

***#iftop #***查看网络流量

***#drpd #***查看丢包数

***#tcpdump #***查看***TCP/IP***传输数据

***#netstat #***网络状态

***#ifconfig #***网络接口配置

***#dumpstat #***转储系统运行状态到文件，***LauncherModel.java dumpstat*** 方法

***#dumpsys #***在串口打印系统数据

***#dmesg #***打印内核调试信息到串口

***#procmem -p 1***

***#start #***执行***emulator/device***实例 ***stop zygote***

***#stop #***停止***emulator/device***实例 ***start zygote***

***#logcat -b radio #***打印***log***

***#sqlite3 #***启动***sqlite***命令行。

***#monkey -v -p com.android.launcher 500 #monkey*** 是用于自动化测试，通过发送事件模拟用户交互。 ***#***代码在***development/cmds/monkey/***

在***Android*** 目录***/system/xbin*** 及 ***/system/bin***下的脚本和可执行程序都在***ADB shell***下执行， ***/system/bin/toolbox,*** 如果觉得***toolbox*** 太简略了，不能满足开发需要， 可以考虑安装 ***busybox***。

如果不想进入 ***adb shell*** 执行，可以在 ***adb shell*** 后面加这些命令***,*** 形如：

***adb shell [command]***

例如：

***$ adb shell iftop***

***Android命令行启动程序的方法为***

***# am start -n ｛包(package)名｝/｛包名｝.{活动(activity)名称}***

***启动的方法可以从每个应用的AndroidManifest.xml的文件中得到***

***Camera（照相机）的Android命令行启动程序方法为：***

***# am start -n com.android.camera/com.android.camera.Camera***

***Browser（浏览器）的Android命令行启动程序方法为：***

***# am start -n com.android.browser/com.android.browser.BrowserActivity***

***mmm frameworks/base/tests/AndroidTests  
adb install -r -f out/target/product/passion/data/app/AndroidTests.apk  
adb shell am instrument -w -e class  
com.android.unit\_tests.PackageManagerTests  
com.android.unit\_tests/android.test.InstrumentationTestRunner***

***# am start -a android.intent.action.CALL -d tel:88888888***

***#am start -a android.intent.action.VIEW -d http://3g.163.com***

***#am start -n com.android.browser/.BrowserActivity***

***usage: am [subcommand] [options]***

***start an Activity: am start [-D] [-W] <INTENT>***

***-D: enable debugging***

***-W: wait for launch to complete***

***start a Service: am startservice <INTENT>***

***send a broadcast Intent: am broadcast <INTENT>***

***start an Instrumentation: am instrument [flags] <COMPONENT>***

***-r: print raw results (otherwise decode REPORT\_KEY\_STREAMRESULT)***

***-e <NAME> <VALUE>: set argument <NAME> to <VALUE>***

***-p <FILE>: write profiling data to <FILE>***

***-w: wait for instrumentation to finish before returning***

***start profiling: am profile <PROCESS> start <FILE>***

***stop profiling: am profile <PROCESS> stop***

***<INTENT> specifications include these flags:***

***[-a <ACTION>] [-d <DATA\_URI>] [-t <MIME\_TYPE>]***

***[-c <CATEGORY> [-c <CATEGORY>] ...]***

***[-e|--es <EXTRA\_KEY> <EXTRA\_STRING\_VALUE> ...]***

***[--esn <EXTRA\_KEY> ...]***

***[--ez <EXTRA\_KEY> <EXTRA\_BOOLEAN\_VALUE> ...]***

***[-e|--ei <EXTRA\_KEY> <EXTRA\_INT\_VALUE> ...]***

***[-n <COMPONENT>] [-f <FLAGS>]***

***[--grant-read-uri-permission] [--grant-write-uri-permission]***

***[--debug-log-resolution]***

***[--activity-brought-to-front] [--activity-clear-top]***

***[--activity-clear-when-task-reset] [--activity-exclude-from-recents]***

***[--activity-launched-from-history] [--activity-multiple-task]***

***[--activity-no-animation] [--activity-no-history]***

***[--activity-no-user-action] [--activity-previous-is-top]***

***[--activity-reorder-to-front] [--activity-reset-task-if-needed]***

***[--activity-single-top]***

***[--receiver-registered-only] [--receiver-replace-pending]***

***[<URI>]***

***am*** 其实是一个脚本，代码放在 ***frameworks/base/cmds/am/am***。类似的，还有***input***， ***pm***， ***ime***，都在 ***cmds***下的相应目录下。

***Pm***是应用软件包管理器， 命令***# pm list packages*** 可以列出系统中安装的包。

***pm install [-l] [-r] [-t] [-i INSTALLER\_PACKAGE\_NAME] [-s] [-f] PATH***

***pm uninstall [-k] PACKAGE***

***adb install /adb uninstall*** 就是通过这个命令来实现的吧！

## adb shell ***无法启动 （***insufficient permissions for device***）***

在连接到开发机器出现如下错误信息：  
shily@hh-desktop:~$adb shell  
error: insufficient permissions for device

用 adb devices查看，得到如下信息：

shily@hh-desktop:~$ adb devices  
List of devices attached  
???????????? no permissions

SDK 中的文档 ***docs/guide/developing/device.html*** 给出了解决方法：

***If you're developing on Ubuntu Linux, you need to add a rules file that contains a USB configuration for each type of device you want to use for development. Each device manufacturer uses a different vendor ID. The example rules files below show how to add an entry for a single vendor ID (the HTC vendor ID). In order to support more devices, you will need additional lines of the same format that provide a different value for the SYSFS{idVendor} property. For other IDs, see the table of*** [***USB Vendor IDs***](#VendorIds)***, below.***

***1. Log in as root and create this file: /etc/udev/rules.d/51-android.rules.***

***For Gusty/Hardy, edit the file to read:  
SUBSYSTEM=="usb", SYSFS{idVendor}=="0bb4", MODE="0666"***

***For Dapper, edit the file to read:  
SUBSYSTEM=="usb\_device", SYSFS{idVendor}=="0bb4", MODE="0666"***

***2.Now execute:  
chmod a+r /etc/udev/rules.d/51-android.rules***

***USB Vendor IDs*** 可以用***lsusb***查看，***Samsung*** 的设备是 ***04e8***，***Google Vendor ID 18d1 http://www.linux-usb.org/usb.ids,***

shily@hh-desktop:~$ lsusb  
Bus 002 Device 003: ID 413c:2003 Dell Computer Corp. Keyboard  
Bus 002 Device 002: ID 0461:4d22 Primax Electronics, Ltd  
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub  
Bus 001 Device 010: ID 18a1:0002  
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub  
上面的id是18a1 没有销售商名，它就是设备的 Vendor ID。

最新版本***Ubuntu***中***SYSFS被ATTRS替换了。我的Lucid Lynx上可以使用的配置：***

***$ cat /etc/udev/rules.d/51-android.rules***

***SUBSYSTEM=="usb", ATTRS{idVendor}=="18d1", MODE="0666"***

修改/etc/udev/rules.d/51-android.rules不需要重启Linux机器，重新插拔一下设备就可以了。再次运行adb devices就可以看到你的设备已经连接  
shily@hh-desktop:~$ adb devices  
List of devices attached  
0403502001011000 device

模拟器链接adb shell不会出现这个问题。

## logcat

***Logcat*** 查看***log*** ，一般格式

***logcat [options] [filterspecs]***

***logcat***

***$adb logcat #***打印***main***缓冲区里的信息

***$adb logcat -b radio #***打印***radio***缓冲区里的信息

***$adb logcat -b radio -b main #*** 打印***radio***、***main***里的信息

或者

***$adb logcat -b radio &; adb logcat -b main& #***串口下

### logcat 选项

***-b [main|radio|system|events]*** 指定消息的***Buffer***，其中***main***、***radio***、***system***保存在***/dev/log/***下，  ***event***比较特殊，它是打开***os-events***相关的***log***打印，默认被写入到手机的***/data/anr/traces.txt***文件中。

***-c*** 清理（***Clear***）***buffer***。

***-d*** 把***buffer***中的日志转储（***Dump***）到屏幕并退出。

***-v <format>*** 输出日志消息的格式。默认是***brief***。

***process tag thread raw time threadtime long***

***-f <filename>*** 把日志消息写到指定的文件***<filename>***，默认是写到标准输出***(stdout)***

***-r [<kbytes>]*** 当日志文件达到***<kilobytes>***时，清空文件，循环记录。默认是***16***。需要 ***-f***。

***-n <count>*** 循环记录的次数，默认是***4***。

如： ***adb logcat -f /data/data/droid.log -r 2 -n 10***

默认格式，即 ***-v brief***

***D/View ( 51): android.widget.TextView@43f0fc10 View.setFrame(125,0,163,25)***

***-v long***

***[ 01-01 08:47:00.210 51:0x3d D/View ]***

***com.android.server.status.DateView@43dacf00 View.setFrame(0,0,31,25)***

***-v thread***

***D( 51:0x3d) com.android.server.status.DateView@43dacf00 View.setFrame(0,0,31,25)***

***-v raw***

***android.widget.FrameLayout@43f3a688 View.setFrame(100,0,125,25)***

### logcat 过滤器

先看看默认的输出

***D/View ( 51): android.widget.TextView@43f0fc10 View.setFrame(125,0,163,25)***

过滤器就是根据上面红色部分过滤的。

其中 ***D*** 表示 ***Debug***，是优先级，也就是***Log.d*** 打印的***; View***表示的***Tag***，即***Log.d***的第一个参数。

要只打印上面部分，可以输入：

***$adb logcat View:d***

更多详细帮组请键入：

***$ adb logcat --help #***在***adb shell***里，***logcat --help***

# 控制台

在模拟器远行后，查询模拟器使用的端口：

$ adb devices

List of devices attached

emulator-5554 device

$telnet localhost 5554

登入Android Console。开发板需要连上网线。而adb走的是串口。

一些命令：

window scale 0.8 # 模拟器窗口缩为0.8

sms send 13451851888 “Hello” #向号码13451851888 发送短信，内容是“Hello ”

power status full #电池状态

power ac off #关闭 AC充电

Android Console 的代码及相关指令可以参考代码，external/qemu/android/console.c

# 调试消息

## Java 的 打印信息

***import android.util.Log;***

方法原型：***Log.v(String tag, String msg);***

使用举例：***Log.v("Foo","Hello world");***

除了***v***，还有其他的方法：

***v :Verbose*** 冗余消息，优先级最低。

***d :Debug*** 调试消息，优先级比 ***Verbose*** 高一级。

***i :Info*** 普通消息，优先级比 ***Debug*** 高一级。

***w :Warn*** 警告消息，优先级比 ***Info*** 高一级。

***e :Error*** 错误消息，优先级最高。

***android.util.Slog;***

与 ***Log*** 类似，区别在于， ***SLog*** 把消息送往 ***system*** 缓冲区里，而 ***Log*** 把消息送往了 ***main*** 缓冲区。其中的 ***v, d, i, w, e*** 的含义与 ***Log*** 中的相同。 但是，看了源码之后，发现把***Log***日志的文件句柄赋给了***SLog***。所以，实际上***SLog***也是往***main Buffer***打印。

***W/PackageManager( 59): Not granting permission android.permission.SEND\_DOWNLOAD\_COMPLETED\_INTENTS to package com.android.browser (protectionLevel=2 flags=0x1be45)***

就是***frameworks/base/services/java/com/android/server/PackageManagerService.java***用***Slog.w***打印出来的。

此类的调用栈如下：

***frameworks/base/core/java/android/util/Log.java -> Log.d("HellWorld","detail");***

***frameworks/base/core/jni/android\_util\_Log.cpp -> android\_util\_Log\_println\_native***

***system/core/liblog/logd\_write.c -> \_\_android\_log\_buf\_write***

最后调用系统函数***writev***

***main***、***radio***、***events***、***system*** ，见***system/core/include/cutils/logger.h***。

## JNI 和 C/C++库的 打印信息

***frameworks/base/core/jni/AndroidRuntime.cpp***

***AndroidRuntime::start*** 成员函数调用

***LOGD("\n>>>>>>>>>>>>>> AndroidRuntime START <<<<<<<<<<<<<<\n");***

在系统启动时打印了这条消息。

除了***LOGD***，还有***LOGV***，***LOGI***，***LOGW***，***LOGE***，***SLOGD***，***SLOGV***，***SLOGI***，***SLOGW***，***SLOGW*** ，等等，详细请见头文件 ***system/core/include/cutils/log.h***。这些都是对函数 ***android\_printLog***包装的宏。

***Printf*** 函数也可以使用。具体细节待研究。

***LOGV*** 默认情况下是不打印的。如果要查看请在***#include <utils/Log.h>***前，加上”***#define LOG\_NDEBUG 0”***

***(***这是***system/core/include/cutils/log.h*** 说明的。***)***

如果需要在***C***程序中加打印语句，有两种方法

1) 直接在代码中使用printf，此方法只能应用于从命令行启动程序的情况，运行时可以adb shell中看到打印信息

2) 使用程序中提供的重定项后的打印语句，并在logcat中看到它  
例如在libdvm.so中使用dvmFprintf(stderr, “xieyan log\n”);

## 内核的打印信息

***decompress\_kernel*** （***arch/arm/boot/compressed/misc.c***） 调用 ***putstr("Uncompressing Linux...");*** 向控制台打印消息。

函数***printk***应该也可以的。这方面没经验，希望大家补充。

## 打印堆栈

打印出的当前函数调用关系

***new Exception().printStackTrace();***

这个是 *Dalvik VM* 级别的打印信息：

*W/System.err( 296): java.lang.Exception: print stack trace*

*W/System.err( 296): at com.android.launcher2.DragController.startDrag(DragController.java:170)*

*W/System.err( 296): at com.android.launcher2.Workspace.startDrag(Workspace.java:1048)*

*W/System.err( 296): at com.android.launcher2.Launcher.onLongClick(Launcher.java:1622)*

*W/System.err( 296): at android.view.View.performLongClick(View.java:2427)*

*W/System.err( 296): at android.widget.TextView.performLongClick(TextView.java:7295)*

*W/System.err( 296): at android.view.View$CheckForLongPress.run(View.java:8795)*

*W/System.err( 296): at android.os.Handler.handleCallback(Handler.java:587)*

*W/System.err( 296): at android.os.Handler.dispatchMessage(Handler.java:92)*

*W/System.err( 296): at android.os.Looper.loop(Looper.java:123)*

*W/System.err( 296): at android.app.ActivityThread.main(ActivityThread.java:4627)*

*W/System.err( 296): at java.lang.reflect.Method.invokeNative(Native Method)*

*W/System.err( 296): at java.lang.reflect.Method.invoke(Method.java:521)*

*W/System.err( 296): at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:868)*

*W/System.err( 296): at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:626)*

*W/System.err( 296): at dalvik.system.NativeStart.main(Native Method)*

***Throwable***类（***libcore/luni-kernel/src/main/java/java/lang/Throwable.java*** ）还有其他的***API*** 可以供调试之用。

# 调试器

## Java 调试器

***JDWP***用于在***java***程序层面的调试***,***当然也可以在***eclipse***中使用***,***或利用***ddms***调试。

* 在某一终端运行虚拟机  
  *$ adb forward tcp:8000 tcp:8000  
  $ adb shell  
  $ dalvikvm -agentlib:jdwp=transport=dt\_socket,address=8000,server=y,suspend=y -cp /sdcard/foo.jar Foo*  
  此时挂起等待调试
* 在另一终端开启调试  
  *$ jdb -attach localhost:8000  
  [jdb提示符] run*此时程序继续运行
* jdb常用命令  
  [jdb 提示符] threads 看当前所有线程  
  [jdb 提示符] trace methods 0x12aac5a00 跟踪线程号为0x12aac5a00的线程(线程号从threads得到)  
  [jdb 提示符] next/step 下一步  
  [jdb 提示符] 40 next 执行40次next

***jdwp(java debug wire protocol)***及原理

1) 虚拟机（设备端）在启动时加载了Agent JDWP 从而具备了调试功能。在调试器端（PC端）通过JDWP协议与设备连接，通过发送命令来获取的状态和控制Java程序的执行。JDWP 是通过命令（command）和回复（reply）进行通信的。

2) JDK 中调试工具 jdb 就是一个调试器，DDMS也提供调试器与设备相连。

3) dalvik为JDWP提供了两种连接方式：tcp方式和adb方式，tcp方式可以手工指定端口，adb方式自动设定为8700端口，通常使用DDMS调试就是通过adb方式

***Android*** 有个工具 ***jdwpspy*** ，以后研究如何使用。

## C/C++ 调试器

* 监听端口

***$adb shell***

***#cd /data/example***

***# gdbserver 10.0.2.2:1234 ./hello***

***Process ./hello created; pid = 274***

***Listening on port 1234***

***10.0.2.2***是模拟器的默认***IP***地址，源码 ***system/core/rootdir/etc/init.goldfish.sh*** 和模拟器中脚本***/system/etc/init.goldfish.sh***中设置 的。

* 数据转发

打开另一个***terminal***，***telnet Android Console***

***$telnet localhost 5554***

在 ***Android*** 控制台，输入：

***redir add tcp:1234:1234***

这条命令表示将所有到***localhost:1234***的数据转发到模拟器的***1234***端口。

* 主机***(host)***远程调试

***$arm-none-linux-gnueabi-gdb hello***

***(gdb) set solib-absolute-prefix /out/target/product/xxxxxx/symbols/system/lib/  
(gdb) set solib-search-path out/target/product/xxxxxx/symbols/system/lib/***  
***(gdb) target remote localhost:1234***

## 调优工具

***MethodTracing***

用于热点分析和性能优化，分析每个函数占用的CPU时间，调用次数，函数调用关系等

a) 在程序代码中加入追踪开关  
*import android.os.Debug;  
……  
android.os.Debug.startMethodTracing(“/data/tmp/test”); // 先建/data/tmp目录  
…… // 被追踪的程序段  
android.os.Debug.stopMethodTracing();*

b) 编译，运行后，设备端生成/data/tmp/test.trace文件

c) 把trace文件复制到PC端  
*$ adb pull /data/tmp/test.trace ./*

d) 使用android自带工具分析trace文件  
*$ traceview test.trace*  
此时可看到各个函数被调用的次数CPU占用率等信息

e) 使用android自带工具分析生成调用关系类图（*安装graphviz*） *$dmtracedump -g test.png test.trace*  
此时目录下生成类图test.png。

trace文件生成与libdvm模块DEBUG版本相冲突，所以此方法只适用于对非DEBUG版本模拟器的调试，否则在分析trace文件时会报错

***HProf (Heap Profile)***

用于java层面的内存分析，显示详细的内存占用信息，指出可疑的内存泄漏对象

a) 在代码中加入dump动作  
*import android.os.Debug;  
import java.io.IOException;  
……  
try {  
 android.os.Debug.dumpHprofData(“/data/tmp/input.hprof”); // 先建/data/tmp目录  
} catch (IOException ioe) {  
}*

b) 把hprof文件复制到PC端  
*$ adb pull /data/tmp/input.hprof ./*

c) 使用命令hprof-conv把hprof转成MAT识别的标准的hprof  
*$ hprof-conv input.hprof output.hprof*

d) 使用ＭＡＴ工具看hprof信息  
下载MAT工具：<http://www.eclipse.org/mat/downloads.php>  
用工具打开output.hprof

***SamplingProfile (android 2.0*上版本使用*)***

每隔Ｎ毫秒对当前正在运行的函数取样，并输出到log中。

*import dalvik.system.SamplingProfiler  
……  
SamplingProfile sp = SamplingProfiler.getInstance();  
sp.start(n); // n为设定每秒采样次数  
sp.logSnapshot(sp.snapshot());  
……  
sp.shutDown();*它会启一个线程监测，在*logcat*中打印信息

***android.os.Debug*中提供的其他工具**

*1)* 取毫微秒级的时间，用于计算时间 *threadCpuTimeNanos()*

2)统计两点间的内存分配情况  
*startAllocCounting()  
stopAllocCounting()  
getGlobalAllocCount()*

*3)*打印当前已*load*的*class  
getLoadedClassCount()  
printLoadedClasses() 它需要打开NDEBUG功能才能打开system/core/中Log功能*

## debuggerd 服务

这个 ***log*** 形式如下：

*I/DEBUG ( 371): \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\**

*I/DEBUG ( 371): Build fingerprint: 'generic/sdk/generic/:2.2/FRF42/36942:eng/test-keys'*

*I/DEBUG ( 371): pid: 377, tid: 388 >>> com.android.browser <<<*

*I/DEBUG ( 371): signal 11 (SIGSEGV), fault addr fbf5d5b3*

*I/DEBUG ( 371): r0 817be6e0 r1 810e5378 r2 810e5373 r3 006f6cfc*

*I/DEBUG ( 371): r4 817be6e4 r5 0000000b r6 00000004 r7 00000000*

*I/DEBUG ( 371): r8 45464d80 r9 428c3ee0 10 428c3ec8 fp 0025bf30*

*I/DEBUG ( 371): ip 810000dc sp 45464180 lr 810e5379 pc b0001330 cpsr 80000010*

*I/DEBUG ( 371): #00 pc b0001330 /system/bin/linker*

*I/DEBUG ( 371): #01 pc 000e5376 /data/data/com.adobe.flashplayer/lib/libflashplayer.so*

*I/DEBUG ( 371): #02 pc 000e5376 /data/data/com.adobe.flashplayer/lib/libflashplayer.so*

*... ...*

*I/DEBUG ( 371): code around pc:*

*I/DEBUG ( 371): b0001310 2b60688b 4a23d112 4e214b24 18e218a0*

*I/DEBUG ( 371): b0001320 4c2319a5 33201c2b 68099400 21c09101*

*I/DEBUG ( 371): b0001330 97020089 f0012500 e02afe5f 0072268c*

*I/DEBUG ( 371): b0001340 1c0d435a 18ae350c 3301604e 608b36a4*

*I/DEBUG ( 371): b0001350 4d126030 0052228c 68451960 36a41c2e*

*I/DEBUG ( 371):*

*I/DEBUG ( 371): code around lr:*

*I/DEBUG ( 371): 810e5358 f85ee00c 68513003 47986892 bd00b003*

*I/DEBUG ( 371): 810e5368 eb006810 e7f60061 bf004b56 f8dfa100*

*I/DEBUG ( 371): 810e5378 b09cc158 a91c185b f8534a54 f8df000c*

*I/DEBUG ( 371): 810e5388 189ac150 0d70f841 e882c903 a91c0003*

*I/DEBUG ( 371): 810e5398 000cf853 0c10f102 0d68f841 e88cc903*

*I/DEBUG ( 371):*

*I/DEBUG ( 371): stack:*

*I/DEBUG ( 371): 45464140 81002180 /data/data/com.adobe.flashplayer/lib/libflashplayer.so*

*I/DEBUG ( 371): 45464144 b0001599 /system/bin/linker*

***... ...***

这个是***C/C++***库出问题时输出的消息，由服务***debuggerd*** 打印的，好像与***JIT*** 相关。 ***Debuggerd***的实现在 ***system/core/debuggerd/debuggerd.c*** ，***system/core/rootdir/init.rc***中定义了此服务，所以是由***init***进程启动的。 ***dalvik/tools/gdbjithelper/README.txt*** 给出了在遇到这类问题如何调试，代码可以阅读文件

***dalvik/tools/gdbjithelper/gdbjithelper.c***等。

# 模拟器

***MMI*** 开发是我们开发的很重要的内容，模拟器能帮助加速开发，调试。

***$android #android SDK*** 下载管理工具

***$android list #***列出***Android SDK*** 安装的组件

***Android*** 模拟器启动

***$emulator @sd -shell-serial***

如果不使用 ***AVD*** 设备里指定的镜像文件，可以使用如下命令：

***$emulator -system system.img -data userdata.img -ramdisk ramdisk.img***

***-kernel kernel/arch/arm/boot/zImage @sd***

其他选项

***-help*** 帮助信息

***-verbose*** 显示***emulator***调用***log***， 等价于 ***-debug-init***

***-debug <tags> tags*** 是***all***，或者是***init***、***console***、***modem***、***radio***、***keys***的任意组合***(***以”***,”***分割***)***

***-logcat v*** 启动***logcat***，跟 ***adb logcat*** 相当。

***-show-kernel*** 打印内核的启动信息

***-avd <name>*** 相当于 ***@ <name>*** 指定 虚拟设备（***Android Virtual Device***）

***-trace <filename>*** 文件保存在***~/.android/avd/<avdname>.avd/traces/<filename>***，记录***trace***消息，

按***9***开始、停止记录。

***-tcpdump <file>*** 把抓到的网络包保存到本地文件

***-partition-size*** 指定***/system***、***/data***两个分区的大小

***-sdcard***

***-memory***

***-cpu-delay 0***

***-prop dalvik.vm.heapsize = 32m Dalvik*** 虚拟机的堆设置 ***32M Byte***。

***# df***

***/dev: 47048K total, 0K used, 47048K available (block size 4096)***

***/mnt/asec: 47048K total, 0K used, 47048K available (block size 4096)***

***/system: 76544K total, 76544K used, 0K available (block size 4096)***

***/data: 65536K total, 31784K used, 33752K available (block size 4096)***

***/cache: 65536K total, 1156K used, 64380K available (block size 4096)***

***/mnt/sdcard: 258064K total, 170870K used, 87193K available (block size 512)***

***/mnt/secure/asec: 258064K total, 170870K used, 87193K available (block size 512)***

***/system*** 分区已没有空间，可以使用***-partition-size*** 选项。

***$ emulator @hasd -logcat v -shell -partition-size 128***

# 在 Eclipse 浏览 Android 的源码

传建 ***Android*** 中，会包含 ***Android***包 ***android.jar***，但是这个包里只是编译过的字节码和资源，没有源码。在***ddms***调试代码时，打开相应的 ***Android Framework*** 的类时，在类名、常变量名按***F3***查看实现时，呈现给你的是毫无意义的***.class***文件。

有一个解决方法：

如果***Android***包 是***~/android/sdk/platforms/android-8/android.jar***，那么在 ***android.jar***所在的目录下建一个目录***sources***。把***Android*** 工程中***frameworks/base/core/java***目录下的文件全都拷贝到这个目录中，这是***framework*** 的源码***;*** 再把***libcore/luni/src/main/java***、***libcore/luni-kernel/src/main/java***目录中的文件也拷贝进来，这些是***Java*** 的基本类的实现代码。***android.jar*** 其余类的源码基本都在***frameworks***及***libcore***两个目录中，按需要添加吧。

原理是：

***android.jar*** 的属性中不允许修改源码链接。***Google***这么做是考虑到程序安全问题吧！我们开发中这个累赘，或许我们修改一下 ***android.jar***的编译选项可以去掉这个限制。

******

好在sdk 对此另有准备，

sdk/eclipse/plugins/com.android.ide.eclipse.adt/src/com/android/ide/eclipse/adt/internal/project/AndroidClasspathContainerInitializer.java:636: paths.add(target.getPath(IAndroidTarget.SOURCES));

sdk/sdkmanager/libs/sdklib/src/com/android/sdklib/SdkConstants.java:212: public static final String FD\_ANDROID\_SOURCES = "sources";

sdk/sdkmanager/libs/sdklib/src/com/android/sdklib/PlatformTarget.java

方法PlatformTarget

mPaths.put(SOURCES, mRootFolderOsPath + SdkConstants.FD\_ANDROID\_SOURCES);

据网上资料，以前（2.2）是放在com.android.ide.eclipse.adt.project.internal包中的，

plugins/com.android.ide.eclipse.adt/src/com/android/ide/eclipse/adt/AndroidConstants.java:

public static final String FD\_ANDROID\_SOURCES = "sources";

# 参考资料

1. [android](http://xy0811.spaces.live.com/Blog/cns!F8AECD2A067A6B17!1484.entry)的调试工具集

# ***Debugging Native Code***，[***http://source.android.com/porting/debugging\_native.html***](http://source.android.com/porting/debugging_native.html)

# ***Debugging with GDB***，[***http://source.android.com/porting/debugging\_gdb.html***](http://source.android.com/porting/debugging_gdb.html)

1. ***- if you have jtag debuggers, nothing like that.,  
     pls refer [http://omappedia.org/wiki/Android\_Debugging#Lauterbach\_TRACE32](http://omappedia.org/wiki/Android_Debugging" \l "Lauterbach_TRACE32)  
     you can debug starting right from bootloader, kernel, android init,  
   and any library.  
     
   - if you can get trace via debug port, you  can start logcat very  
   early from your init.rc, way before you get shell or adb.  
      service logcat /system/bin/logcat -f /dev/kmsg  
          oneshot***