# TODO

system/core/adb/remount\_service.cpp root的原理嗷

Connected to the target VM, address: 'localhost:8676', transport: 'socket'

吸附调试原理

vendor-id 和 product-id

PC端

C:\Users\key.guan\Desktop>adb shell

\* daemon not running; starting now at tcp:5037

\* daemon started successfully

# 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。路径：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，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。

host端==PC端==电脑端

device端==手机端==目标端

Adb（server）：host端，ADB\_HOST，一个是windows、linux、darwin或freebsd运行的程序

Adb（client）：device端，手机进程，

Adbd： device端，ADB\_HOST，手机终的守护进程

宏ADB\_HOST用于区分是PC端程序还是目标机端的程序。用于device端还是host端

宏ADB\_HOST\_ON\_TARGET用于区分adb程序是否是在目标机上运行。

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>

# device

## disable-verity

本质就是修改文件分区表的只读属性set\_verity\_enable\_state\_service.cpp

### set\_verity\_enabled\_state\_service

**void** set\_verity\_enabled\_state\_service(**int** fd, **void**\* cookie) {  
 unique\_fd closer(fd);  
  
 **bool** enable = (cookie != NULL);  
 **if** (!kAllowDisableVerity) {//  
 WriteFdFmt(fd, **"%s-verity only works for userdebug builds\n"**,  
 enable ? **"enable"** : **"disable"**);  
 }  
  
 **if** (!android::base::GetBoolProperty(**"ro.secure"**, **false**)) {  
 WriteFdFmt(fd, **"verity not enabled - ENG build\n"**);  
 **return**;  
 }  
 **if** (!\_\_android\_log\_is\_debuggable()) {  
 WriteFdFmt(fd, **"verity cannot be disabled/enabled - USER build\n"**);  
 **return**;  
 }  
  
 *// read all fstab entries at once from all sources* fstab = fs\_mgr\_read\_fstab\_default();  
 **if** (!fstab) {  
 WriteFdFmt(fd, **"Failed to read fstab\nMaybe run adb root?\n"**);  
 **return**;  
 }  
  
 *// Loop through entries looking for ones that vold manages.* **bool** any\_changed = **false**;  
 **for** (**int** i = 0; i < fstab->num\_entries; i++) {  
 **if** (fs\_mgr\_is\_verified(&fstab->recs[i])) {  
 **if** (!set\_verity\_enabled\_state(fd, fstab->recs[i].blk\_device,  
 fstab->recs[i].mount\_point,  
 enable)) {  
 any\_changed = **true**;  
 }  
 }  
 }

### kAllowDisableVerity

ifneq (,$(filter userdebug eng,$(TARGET\_BUILD\_VARIANT)))  
LOCAL\_CFLAGS += -DALLOW\_ADBD\_DISABLE\_VERITY=1

#ifdef ALLOW\_ADBD\_DISABLE\_VERITY  
**static const bool** kAllowDisableVerity = **true**;  
#else  
**static const bool** kAllowDisableVerity = **false**;  
#endif

### ro.secure

**ro.secure**

bullhead:/ # getprop ro.secure

1

### \_\_android\_log\_is\_debuggable

**"ro.debuggable"**;

127|bullhead:/ # getprop ro.debuggable

1

### fstab = fs\_mgr\_read\_fstab\_default();

### set\_verity\_enabled\_state

fh.set\_verity\_status(enable)

方法二：修改系统源码

device/google/marlin/charger.fstab.qcom

修改源码中的fstab文件，把system分区的verify标志去掉   
如把   
/dev/block/platform/mtk-msdc.0/11230000.msdc0/by-name/system /system ext4 ro wait,verify   
修改为   
/dev/block/platform/mtk-msdc.0/11230000.msdc0/by-name/system /system ext4 ro wait

# Adb（client）？？

system/core/adb/client/main.cpp

system/core/adb/commandline.cpp

## shell

**else if** (!strcmp(argv[0], **"shell"**)) {  
 **return** adb\_shell(argc, argv);  
}

## Adb tcpip 5555

**else if** (!strcmp(argv[0], **"tcpip"**) && argc > 1) {  
 **return** adb\_connect\_command(android::base::StringPrintf(**"tcpip:%s"**, argv[1]));  
}

# adb install 流程分析

system/core/adb/commandline.cpp

* 首先定义两个拷贝位置：分别对应拷贝到内部或者外部存储。这里需要根据传入参数确定拷贝位置。默认是拷贝到内部存储中。
* 执行拷贝：将制定的apk文件从电脑端拷贝到制定的目录处。
* 执行安装：调用pm\_command()函数开始去安装。
* 删除文件：当安装结束时删除该apk文件。

int adb\_commandline(int argc, const char \*\*argv) {

**else if** (!strcmp(argv[0], **"install"**)) {  
 **if** (argc < 2) **return** syntax\_error(**"install requires an argument"**);  
 **if** (\_use\_legacy\_install()) {  
 **return install\_app\_legacy**(argc, argv);  
 }  
 **return** install\_app(argc, argv);  
}

接下来会调用到 install\_app函数. 在其内，会执行一个pm脚本，该脚本主要功能就是通过app\_process来执行pm.jar 的main函数(有关于app\_procee相关信息大家可以去查阅一下), pm.jar的main函数就相当于java进程的入口函数，下面对它进行分析（PM.java）

## install\_app\_legacy

**static bool** \_use\_legacy\_install() {  
 FeatureSet features;  
 std::string error;  
 **if** (!adb\_get\_feature\_set(&features, &error)) {  
 fprintf(stderr, **"error: %s\n"**, error.c\_str());  
 **return true**;  
 }  
 **return** !CanUseFeature(features, kFeatureCmd);  
}

n. 遗产; 遗赠财物; 遗留：legacy

\_use\_legacy\_install

## install\_app\_legacy

确定apk拷贝的目标地址

找出apk

拷贝到目标地址

**static int** install\_app\_legacy(**int** argc, **const char**\*\* argv) {  
 **static const char** \***const** DATA\_DEST = **"/data/local/tmp/%s"**;  
 **static const char** \***const** SD\_DEST = **"/sdcard/tmp/%s"**;  
 **const char**\* where = DATA\_DEST;  
  
 **for** (**int** i = 1; i < argc; i++) {  
 **if** (!strcmp(argv[i], **"-s"**)) {  
 where = SD\_DEST;  
 }  
 }  
  
 *// Find last APK argument.  
 // All other arguments passed through verbatim.* **int** last\_apk = -1;  
 **for** (**int** i = argc - 1; i >= 0; i--) {  
 **if** (android::base::EndsWithIgnoreCase(argv[i], **".apk"**)) {  
 last\_apk = i;  
 **break**;  
 }  
 }  
  
 **if** (last\_apk == -1) **return** syntax\_error(**"need APK file on command line"**);  
  
 **int** result = -1;  
 std::vector<**const char**\*> apk\_file = {argv[last\_apk]};  
 std::string apk\_dest = android::base::StringPrintf(  
 where, android::base::Basename(argv[last\_apk]).c\_str());  
 **if** (!do\_sync\_push(apk\_file, apk\_dest.c\_str(), **false**)) **goto** cleanup\_apk;  
 argv[last\_apk] = apk\_dest.c\_str(); */\* destination name, not source location \*/* result = pm\_command(argc, argv);  
  
cleanup\_apk:  
 delete\_file(apk\_dest);  
 **return** result;  
}

### do\_sync\_push

### pm\_command

修改了目标源目录后

封装成一个shell命令 pm install

argv[last\_apk] = apk\_dest.c\_str(); */\* destination name, not source location \*/* result = pm\_command(argc, argv);

**static int** pm\_command(**int** argc, **const char**\*\* argv) {  
 std::string cmd = **"pm"**;  
  
 **while** (argc-- > 0) {  
 cmd += **" "** + escape\_arg(\*argv++);  
 }  
  
 **return** send\_shell\_command(cmd, **false**);  
}

std::string error;  
std::string service\_string = ShellServiceString(use\_shell\_protocol, **""**, command);  
  
fd = adb\_connect(service\_string, &error);  
**if** (fd >= 0) {  
 **break**;  
}

fprintf(stderr, **"- waiting for device -\n"**);  
**if** (!wait\_for\_device(**"wait-for-device"**)) {  
 **return** 1;  
}

解释了手机没有连接会等待的问题

# 案例

## ADB\_SERVER\_VERSION

system/core/adb/adb.h

*// Increment this when we want to force users to start a new adb server.*#define ADB\_SERVER\_VERSION 39

版本不匹配

C:\Users\10288>adb shell

adb server version (31) doesn't match this client (40); killing...

\* daemon started successfully

查看版本

C:\Users\10288>adb version

Android Debug Bridge version 1.0.40

Version 4986621

Installed as C:\Users\10288\AppData\Local\Android\Sdk\platform-tools\adb.exe

adb version

解决办法

在cmd中输入adb nodaemon server -a后解决了，

<https://www.jianshu.com/p/e3342b5e5581>