frameworks\base\packages\Shell\src\com\android\shell

framework/native/cmds/bugreport/bugreport.cpp

framework/native/cmds/dumpstate/dumpstate.cpp

framework/native/cmds/dumpstate/utils.c

<http://gityuan.com/2016/06/10/bugreport/>

<http://blog.csdn.net/feitian_666/article/details/52880213>

# bugreport

## 概述

通过adb命令可获取bugrepport信息，并输出到文件当前路径的bugreport.txt文件：

adb bugreport > bugreport.txt

对于Android系统调试分析，bugreport信息量非常之大，几乎涵盖整个系统各个层面内容，对于分析BUG是一大利器，本文先从从源码角度来分析一下Bugreport的实现原理。

## 原理分析

Android系统源码中framework/native/cmds/bugreport目录通过Android.mk定义了bugreport项目，在系统编译完成后会生成bugreport可执行文件，位于系统/system/bin/bugreport。当执行adb bugreport时，便会调用这个可执行文件，进入bugreport.cpp中的main()方法。

### Mk

LOCAL\_SRC\_FILES:= bugreport.cpp

LOCAL\_MODULE:= bugreport

LOCAL\_CFLAGS := -Wall

LOCAL\_SHARED\_LIBRARIES := libcutils

### bugreport.main

bugreport.cpp

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **int** main() {
2. //启动dumpstate服务
3. property\_set("ctl.start", "dumpstate");
4. //需要多次尝试，直到dumpstate服务启动完成，才能建立socket通信
5. **int** s;
6. **for** (**int** i = 0; i < 20; i++) {
7. s = socket\_local\_client("dumpstate", ANDROID\_SOCKET\_NAMESPACE\_RESERVED,
8. SOCK\_STREAM);
9. **if** (s >= 0)
10. **break**;
11. //休眠1s后再次尝试连接
12. sleep(1);
13. }
14. **if** (s == -1) {
15. printf("Failed to connect to dumpstate service: %s\n", strerror(errno));
16. **return** 1;
17. }
18. //当3分钟没有任何数据可读，则超时停止读取并退出。
19. //dumpstate服务中不存在大于1分钟的timetout，因而不可预见的超时的情况下留有很大的回旋余地。
20. struct timeval tv;
21. tv.tv\_sec = 3 \* 60;
22. tv.tv\_usec = 0;
23. **if** (setsockopt(s, SOL\_SOCKET, SO\_RCVTIMEO, &tv, sizeof(tv)) == -1) {
24. printf("WARNING: Cannot set socket timeout: %s\n", strerror(errno));
25. }
26. **while** (1) {
27. **char** buffer[65536];
28. ssize\_t bytes\_read = TEMP\_FAILURE\_RETRY(read(s, buffer, sizeof(buffer)));
29. **if** (bytes\_read == 0) {
30. **break**;
31. } **else** **if** (bytes\_read == -1) {
32. // EAGAIN意味着timeout，Bugreport读异常终止
33. **if** (errno == EAGAIN) {
34. errno = ETIMEDOUT;
35. }
36. **break**;
37. }
38. ssize\_t bytes\_to\_send = bytes\_read;
39. ssize\_t bytes\_written;
40. //不断循环得将读取数据输出到stdout
41. **do** {
42. bytes\_written = TEMP\_FAILURE\_RETRY(write(STDOUT\_FILENO,
43. buffer + bytes\_read - bytes\_to\_send, bytes\_to\_send));
44. **if** (bytes\_written == -1) {
45. **return** 1; //将数据无法写入stdout
46. }
47. bytes\_to\_send -= bytes\_written;
48. } **while** (bytes\_written != 0 && bytes\_to\_send > 0);
49. }
50. close(s);
51. **return** 0;
52. }

property\_set(“ctl.start”, “dumpstate”)会触发init进程,来fork进程/system/bin/dumpstate, 作为dumpstate服务的进程. Bugreport再通过socket建立于dumpstate的通信，这个过程会尝试20次socket连接建立直到成功连接。 在socket通道中如果持续3分钟没有任何数据可读，则超时停止读取并退出。由于dumpstate服务中不存在大于1分钟的timetout，因而不可预见的超时的情况下留有很大的回旋余地。

当从socket读取到数据后，写入到标准时输出或者重定向到文件。可见bugreport数据的来源都是dumpstate服务，那么接下来去看看dumpstate服务的工作。

## Dumpstate

Sgf

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **int** main(**int** argc, **char** \*argv[]) {
2. struct sigaction sigact;
3. **int** do\_add\_date = 0;
4. **int** do\_vibrate = 1;
5. **char**\* use\_outfile = 0;
6. **int** use\_socket = 0;
7. **int** do\_fb = 0;
8. **int** do\_broadcast = 0;
9. **if** (getuid() != 0) {
10. //兼容性考虑，旧版本支持直接调用dumpstate命令，新版本通过调用/system/bin/bugreport来替代。
11. //当检测到直接调用，则强制执行bugreport命令。
12. **return** execl("/system/bin/bugreport", "/system/bin/bugreport", NULL);
13. }
14. ALOGI("begin\n");
15. //清空句柄SIGPIPE
16. memset(&sigact, 0, sizeof(sigact));
17. sigact.sa\_handler = sigpipe\_handler;
18. sigaction(SIGPIPE, &sigact, NULL);
19. //提高当前进程的优先级，防止被OOM Killer杀死
20. setpriority(PRIO\_PROCESS, 0, -20);
21. FILE \*oom\_adj = fopen("/proc/self/oom\_adj", "we");
22. **if** (oom\_adj) {
23. fputs("-17", oom\_adj);
24. fclose(oom\_adj);
25. }
26. //参数解析
27. **int** c;
28. **while** ((c = getopt(argc, argv, "dho:svqzpB")) != -1) {
29. **switch** (c) {
30. **case** 'd': do\_add\_date = 1;       **break**;
31. **case** 'o': use\_outfile = optarg;  **break**;
32. **case** 's': use\_socket = 1;        **break**;
33. **case** 'v': **break**;  // compatibility no-op
34. **case** 'q': do\_vibrate = 0;        **break**;
35. **case** 'p': do\_fb = 1;             **break**;
36. **case** 'B': do\_broadcast = 1;      **break**;
37. **case** '?': printf("\n");
38. **case** 'h':
39. usage();
40. exit(1);
41. }
42. }
43. //建立socket
44. **if** (use\_socket) {
45. redirect\_to\_socket(stdout, "dumpstate");
46. }
47. //打开vibrator
48. FILE \*vibrator = 0;
49. **if** (do\_vibrate) {
50. vibrator = fopen("/sys/class/timed\_output/vibrator/enable", "we");
51. **if** (vibrator) {
52. vibrate(vibrator, 150);
53. }
54. }
55. //读取/proc/cmdline
56. FILE \*cmdline = fopen("/proc/cmdline", "re");
57. **if** (cmdline != NULL) {
58. fgets(cmdline\_buf, sizeof(cmdline\_buf), cmdline);
59. fclose(cmdline);
60. }
61. //收集虚拟机和native进程的stack traces(需要root权限)
62. dump\_traces\_path = dump\_traces();
63. //获取tombstone文件描述符
64. get\_tombstone\_fds(tombstone\_data);
65. //确保capabilities
66. **if** (prctl(PR\_SET\_KEEPCAPS, 1) < 0) {
67. ALOGE("prctl(PR\_SET\_KEEPCAPS) failed: %s\n", strerror(errno));
68. **return** -1;
69. }
70. //切换到非root用户和组，在切换之前都是处于root权限
71. gid\_t groups[] = { AID\_LOG, AID\_SDCARD\_R, AID\_SDCARD\_RW,
72. AID\_MOUNT, AID\_INET, AID\_NET\_BW\_STATS };
73. **if** (setgroups(sizeof(groups)/sizeof(groups[0]), groups) != 0) {
74. ALOGE("Unable to setgroups, aborting: %s\n", strerror(errno));
75. **return** -1;
76. }
77. **if** (setgid(AID\_SHELL) != 0) {
78. ALOGE("Unable to setgid, aborting: %s\n", strerror(errno));
79. **return** -1;
80. }
81. **if** (setuid(AID\_SHELL) != 0) {
82. ALOGE("Unable to setuid, aborting: %s\n", strerror(errno));
83. **return** -1;
84. }
85. struct \_\_user\_cap\_header\_struct capheader;
86. struct \_\_user\_cap\_data\_struct capdata[2];
87. memset(&capheader, 0, sizeof(capheader));
88. memset(&capdata, 0, sizeof(capdata));
89. capheader.version = \_LINUX\_CAPABILITY\_VERSION\_3;
90. capheader.pid = 0;
91. capdata[CAP\_TO\_INDEX(CAP\_SYSLOG)].permitted = CAP\_TO\_MASK(CAP\_SYSLOG);
92. capdata[CAP\_TO\_INDEX(CAP\_SYSLOG)].effective = CAP\_TO\_MASK(CAP\_SYSLOG);
93. capdata[0].inheritable = 0;
94. capdata[1].inheritable = 0;
95. **if** (capset(&capheader, &capdata[0]) < 0) {
96. ALOGE("capset failed: %s\n", strerror(errno));
97. **return** -1;
98. }
99. //如果需要，则重定向输出
100. **char** path[PATH\_MAX], tmp\_path[PATH\_MAX];
101. pid\_t gzip\_pid = -1;
102. **if** (!use\_socket && use\_outfile) {
103. strlcpy(path, use\_outfile, sizeof(path));
104. **if** (do\_add\_date) {
105. **char** date[80];
106. time\_t now = time(NULL);
107. strftime(date, sizeof(date), "-%Y-%m-%d-%H-%M-%S", localtime(&now));
108. strlcat(path, date, sizeof(path));
109. }
110. **if** (do\_fb) {
111. strlcpy(screenshot\_path, path, sizeof(screenshot\_path));
112. strlcat(screenshot\_path, ".png", sizeof(screenshot\_path));
113. }
114. strlcat(path, ".txt", sizeof(path));
115. strlcpy(tmp\_path, path, sizeof(tmp\_path));
116. strlcat(tmp\_path, ".tmp", sizeof(tmp\_path));
117. redirect\_to\_file(stdout, tmp\_path);
118. }
119. //这里是真正干活的地方 【见小节 2.3】
120. dumpstate();
121. //通过震动提醒已完成所有dump操作
122. **if** (vibrator) {
123. **for** (**int** i = 0; i < 3; i++) {
124. vibrate(vibrator, 75);
125. usleep((75 + 50) \* 1000);
126. }
127. fclose(vibrator);
128. }
129. //等待gzip的完成，等进程退出时则会被杀
130. **if** (gzip\_pid > 0) {
131. fclose(stdout);
132. waitpid(gzip\_pid, NULL, 0);
133. }
134. //重命名.tmp文件到最终位置
135. **if** (use\_outfile && rename(tmp\_path, path)) {
136. fprintf(stderr, "rename(%s, %s): %s\n", tmp\_path, path, strerror(errno));
137. }
138. //通过发送广播告知ActivityManager已完成bugreport操作
139. **if** (do\_broadcast && use\_outfile && do\_fb) {
140. run\_command(NULL, 5, "/system/bin/am", "broadcast", "--user", "0",
141. "-a", "android.intent.action.BUGREPORT\_FINISHED",
142. "--es", "android.intent.extra.BUGREPORT", path,
143. "--es", "android.intent.extra.SCREENSHOT", screenshot\_path,
144. "--receiver-permission", "android.permission.DUMP", NULL);
145. }
146. ALOGI("done\n");
147. **return** 0;
148. }

整个过程的工作流程：

1. 提高执行dumpsate所在进程的优先级，防止被OOM Killer杀死；
2. 参数解析，可通过命令adb shell dumpstate -h查看dumpstate命令所支持的参数；
3. 打开vibrator，用于在执行bugreport时，手机会先震动一下用于提醒开始抓取系统信息；
4. 通过dump\_traces()来完成收集虚拟机和native进程的stack traces；
5. 通过get\_tombstone\_fds来获取tombstone文件描述符；
6. 开始执行切换到非root用户和组，在这之前的执行都处于root权限；
7. **执行dumpstate()，这里是真正干活的地方**；
8. 再次通过震动以提醒dump操作执行完成；
9. 发送广播，告知ActivityManager已完成bugreport操作。

接下来就重点说说dumpstate()功能：

#### dumpstate()

afds

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **static** **void** dumpstate() {
2. ...
3. property\_get("ro.build.display.id", build, "(unknown)");
4. property\_get("ro.build.fingerprint", fingerprint, "(unknown)");
5. property\_get("ro.build.type", build\_type, "(unknown)");
6. property\_get("ro.baseband", radio, "(unknown)");
7. property\_get("ro.bootloader", bootloader, "(unknown)");
8. property\_get("gsm.operator.alpha", network, "(unknown)");
9. strftime(date, sizeof(date), "%Y-%m-%d %H:%M:%S", localtime(&now));
10. //开头信息
11. printf("========================================================\n");
12. printf("== dumpstate: %s\n", date);
13. printf("========================================================\n");
14. printf("\n");
15. printf("Build: %s\n", build);
16. printf("Build fingerprint: '%s'\n", fingerprint);
17. printf("Bootloader: %s\n", bootloader);
18. printf("Radio: %s\n", radio);
19. printf("Network: %s\n", network);
20. printf("Kernel: "); dump\_file(NULL, "/proc/version");
21. printf("Command line: %s\n", strtok(cmdline\_buf, "\n"));
22. printf("\n");
23. //记录系统运行时长和休眠时长
24. run\_command("UPTIME", 10, "uptime", NULL);
26. //输出mmcblk0设备信息
27. dump\_files("UPTIME MMC PERF", mmcblk0, skip\_not\_stat, dump\_stat\_from\_fd);
29. dump\_file("MEMORY INFO", "/proc/meminfo");
30. run\_command("CPU INFO", 10, "top", "-n", "1", "-d", "1", "-m", "30", "-t", NULL);
31. run\_command("PROCRANK", 20, "procrank", NULL);
32. dump\_file("VIRTUAL MEMORY STATS", "/proc/vmstat");
33. dump\_file("VMALLOC INFO", "/proc/vmallocinfo");
34. dump\_file("SLAB INFO", "/proc/slabinfo");
35. dump\_file("ZONEINFO", "/proc/zoneinfo");
36. dump\_file("PAGETYPEINFO", "/proc/pagetypeinfo");
37. dump\_file("BUDDYINFO", "/proc/buddyinfo");
38. dump\_file("FRAGMENTATION INFO", "/d/extfrag/unusable\_index");
39. dump\_file("KERNEL WAKELOCKS", "/proc/wakelocks");
40. dump\_file("KERNEL WAKE SOURCES", "/d/wakeup\_sources");
41. dump\_file("KERNEL CPUFREQ", "/sys/devices/system/cpu/cpu0/cpufreq/stats/time\_in\_state");
42. dump\_file("KERNEL SYNC", "/d/sync");
43. run\_command("PROCESSES", 10, "ps", "-P", NULL);
44. run\_command("PROCESSES AND THREADS", 10, "ps", "-t", "-p", "-P", NULL);
45. run\_command("PROCESSES (SELINUX LABELS)", 10, "ps", "-Z", NULL);
46. run\_command("LIBRANK", 10, "librank", NULL);
48. //输出kernel log
49. do\_dmesg();
51. //所有已打开文件
52. run\_command("LIST OF OPEN FILES", 10, SU\_PATH, "root", "lsof", NULL);
53. //遍历所有进程的show map
54. for\_each\_pid(do\_showmap, "SMAPS OF ALL PROCESSES");
55. //显示所有线程的blocked位置
56. for\_each\_tid(show\_wchan, "BLOCKED PROCESS WAIT-CHANNELS");
58. //SYSTEM LOG
59. timeout = logcat\_timeout("main") + logcat\_timeout("system") + logcat\_timeout("crash");
60. **if** (timeout < 20000) {
61. timeout = 20000;
62. }
63. run\_command("SYSTEM LOG", timeout / 1000, "logcat", "-v", "threadtime", "-d", "\*:v", NULL);
65. //EVENT LOG
66. timeout = logcat\_timeout("events");
67. **if** (timeout < 20000) {
68. timeout = 20000;
69. }
70. run\_command("EVENT LOG", timeout / 1000, "logcat", "-b", "events", "-v", "threadtime", "-d", "\*:v", NULL);
72. //RADIO LOG
73. timeout = logcat\_timeout("radio");
74. **if** (timeout < 20000) {
75. timeout = 20000;
76. }
77. run\_command("RADIO LOG", timeout / 1000, "logcat", "-b", "radio", "-v", "threadtime", "-d", "\*:v", NULL);
79. //Log统计信息
80. run\_command("LOG STATISTICS", 10, "logcat", "-b", "all", "-S", NULL);
82. //输出当前虚拟机和native进程的vm traces
83. **if** (dump\_traces\_path != NULL) {
84. dump\_file("VM TRACES JUST NOW", dump\_traces\_path);
85. }
87. //输出上次发生ANR时vm traces，即路径/data/anr/traces.txt
88. struct stat st;
89. **char** anr\_traces\_path[PATH\_MAX];
90. property\_get("dalvik.vm.stack-trace-file", anr\_traces\_path, "");
91. **if** (!anr\_traces\_path[0]) {
92. printf("\*\*\* NO VM TRACES FILE DEFINED (dalvik.vm.stack-trace-file)\n\n");
93. } **else** {
94. **int** fd = TEMP\_FAILURE\_RETRY(open(anr\_traces\_path,
95. O\_RDONLY | O\_CLOEXEC | O\_NOFOLLOW | O\_NONBLOCK));
96. **if** (fd < 0) {
97. printf("\*\*\* NO ANR VM TRACES FILE (%s): %s\n\n", anr\_traces\_path, strerror(errno));
98. } **else** {
99. dump\_file\_from\_fd("VM TRACES AT LAST ANR", anr\_traces\_path, fd);
100. }
101. }
103. //输出慢操作的vm traces，例如/data/anr/slow1.txt
104. **if** (anr\_traces\_path[0] != 0) {
105. **int** tail = strlen(anr\_traces\_path)-1;
106. **while** (tail > 0 && anr\_traces\_path[tail] != '/') {
107. tail--;
108. }
109. **int** i = 0;
110. **while** (1) {
111. //例如trace文件为/data/anr/slow1.txt
112. sprintf(anr\_traces\_path+tail+1, "slow%02d.txt", i);
113. **if** (stat(anr\_traces\_path, &st)) {
114. **break**;
115. }
116. dump\_file("VM TRACES WHEN SLOW", anr\_traces\_path);
117. i++;
118. }
119. }
121. //输出tombstone信息，NUM\_TOMBSTONES=10，例如/data/tombstones/tombstone\_1
122. **int** dumped = 0;
123. **for** (size\_t i = 0; i < NUM\_TOMBSTONES; i++) {
124. **if** (tombstone\_data[i].fd != -1) {
125. dumped = 1;
126. dump\_file\_from\_fd("TOMBSTONE", tombstone\_data[i].name, tombstone\_data[i].fd);
127. tombstone\_data[i].fd = -1;
128. }
129. }
130. **if** (!dumped) {
131. printf("\*\*\* NO TOMBSTONES to dump in %s\n\n", TOMBSTONE\_DIR);
132. }
134. dump\_file("NETWORK DEV INFO", "/proc/net/dev");
135. dump\_file("QTAGUID NETWORK INTERFACES INFO", "/proc/net/xt\_qtaguid/iface\_stat\_all");
136. dump\_file("QTAGUID NETWORK INTERFACES INFO (xt)", "/proc/net/xt\_qtaguid/iface\_stat\_fmt");
137. dump\_file("QTAGUID CTRL INFO", "/proc/net/xt\_qtaguid/ctrl");
138. dump\_file("QTAGUID STATS INFO", "/proc/net/xt\_qtaguid/stats");
140. //输出上次的kernel log
141. **if** (!stat(PSTORE\_LAST\_KMSG, &st)) {
142. //文件为/sys/fs/pstore/console-ramoops
143. dump\_file("LAST KMSG", PSTORE\_LAST\_KMSG);
144. } **else** {
145. //文件为/proc/last\_kmsg
146. dump\_file("LAST KMSG", "/proc/last\_kmsg");
147. }
149. //输出上次 logcat，内核必须设置CONFIG\_PSTORE\_PMSG
150. run\_command("LAST LOGCAT", 10, "logcat", "-L", "-v", "threadtime",
151. "-b", "all", "-d", "\*:v", NULL);
153. //wifi驱动/固件 以及ip相关信息
154. run\_command("NETWORK INTERFACES", 10, "ip", "link", NULL);
155. run\_command("IPv4 ADDRESSES", 10, "ip", "-4", "addr", "show", NULL);
156. run\_command("IPv6 ADDRESSES", 10, "ip", "-6", "addr", "show", NULL);
157. run\_command("IP RULES", 10, "ip", "rule", "show", NULL);
158. run\_command("IP RULES v6", 10, "ip", "-6", "rule", "show", NULL);
159. dump\_route\_tables();
160. run\_command("ARP CACHE", 10, "ip", "-4", "neigh", "show", NULL);
161. run\_command("IPv6 ND CACHE", 10, "ip", "-6", "neigh", "show", NULL);
162. run\_command("IPTABLES", 10, SU\_PATH, "root", "iptables", "-L", "-nvx", NULL);
163. run\_command("IP6TABLES", 10, SU\_PATH, "root", "ip6tables", "-L", "-nvx", NULL);
164. run\_command("IPTABLE NAT", 10, SU\_PATH, "root", "iptables", "-t", "nat", "-L", "-nvx", NULL);
165. run\_command("IPTABLE RAW", 10, SU\_PATH, "root", "iptables", "-t", "raw", "-L", "-nvx", NULL);
166. run\_command("IP6TABLE RAW", 10, SU\_PATH, "root", "ip6tables", "-t", "raw", "-L", "-nvx", NULL);
167. run\_command("WIFI NETWORKS", 20, SU\_PATH, "root", "wpa\_cli", "IFNAME=wlan0", "list\_networks", NULL);
169. //中断向量表
170. dump\_file("INTERRUPTS (1)", "/proc/interrupts");
171. run\_command("NETWORK DIAGNOSTICS", 10, "dumpsys", "connectivity", "--diag", NULL);
172. //中断向量表(二次输出)
173. dump\_file("INTERRUPTS (2)", "/proc/interrupts");
175. //获取properties属性值
176. print\_properties();
177. run\_command("VOLD DUMP", 10, "vdc", "dump", NULL);
178. run\_command("SECURE CONTAINERS", 10, "vdc", "asec", "list", NULL);
179. //可用空间
180. run\_command("FILESYSTEMS & FREE SPACE", 10, "df", NULL);
181. run\_command("LAST RADIO LOG", 10, "parse\_radio\_log", "/proc/last\_radio\_log", NULL);
183. //背光信息
184. printf("------ BACKLIGHTS ------\n");
185. printf("LCD brightness="); dump\_file(NULL, "/sys/class/leds/lcd-backlight/brightness");
186. printf("Button brightness="); dump\_file(NULL, "/sys/class/leds/button-backlight/brightness");
187. printf("Keyboard brightness="); dump\_file(NULL, "/sys/class/leds/keyboard-backlight/brightness");
188. printf("ALS mode="); dump\_file(NULL, "/sys/class/leds/lcd-backlight/als");
189. printf("LCD driver registers:\n"); dump\_file(NULL, "/sys/class/leds/lcd-backlight/registers");
190. printf("\n");
192. //Binder相关
193. dump\_file("BINDER FAILED TRANSACTION LOG", "/sys/kernel/debug/binder/failed\_transaction\_log");
194. dump\_file("BINDER TRANSACTION LOG", "/sys/kernel/debug/binder/transaction\_log");
195. dump\_file("BINDER TRANSACTIONS", "/sys/kernel/debug/binder/transactions");
196. dump\_file("BINDER STATS", "/sys/kernel/debug/binder/stats");
197. dump\_file("BINDER STATE", "/sys/kernel/debug/binder/state");
199. printf("========================================================\n");
200. printf("== Board\n");
201. printf("========================================================\n");
202. dumpstate\_board(); printf("\n");
204. //输出framework各种服务的dumpsys信息
205. printf("========================================================\n");
206. printf("== Android Framework Services\n");
207. printf("========================================================\n");
208. run\_command("DUMPSYS", 60, "dumpsys", NULL); //很耗时则timeout=60s
210. printf("========================================================\n");
211. printf("== Checkins\n");
212. printf("========================================================\n");
213. run\_command("CHECKIN BATTERYSTATS", 30, "dumpsys", "batterystats", "-c", NULL);
214. run\_command("CHECKIN MEMINFO", 30, "dumpsys", "meminfo", "--checkin", NULL);
215. run\_command("CHECKIN NETSTATS", 30, "dumpsys", "netstats", "--checkin", NULL);
216. run\_command("CHECKIN PROCSTATS", 30, "dumpsys", "procstats", "-c", NULL);
217. run\_command("CHECKIN USAGESTATS", 30, "dumpsys", "usagestats", "-c", NULL);
218. run\_command("CHECKIN PACKAGE", 30, "dumpsys", "package", "--checkin", NULL);
220. //输出当前 运行中activity/service/provider信息
221. printf("========================================================\n");
222. printf("== Running Application Activities\n");
223. printf("========================================================\n");
224. run\_command("APP ACTIVITIES", 30, "dumpsys", "activity", "all", NULL);
225. printf("========================================================\n");
226. printf("== Running Application Services\n");
227. printf("========================================================\n");
228. run\_command("APP SERVICES", 30, "dumpsys", "activity", "service", "all", NULL);
229. printf("========================================================\n");
230. printf("== Running Application Providers\n");
231. printf("========================================================\n");
232. run\_command("APP SERVICES", 30, "dumpsys", "activity", "provider", "all", NULL);
233. printf("========================================================\n");
234. printf("== dumpstate: done\n");
235. printf("========================================================\n");
236. }

该方法涉及run\_command其他几个方法见下方

### run\_command()

功能是fork子进程并等待它执行完成，或者超时退出。当命令title不为空时，每次输出结果，都分别以下面作为开头和结尾:

------ <title> (<command>) ------

[<command>: <执行时长> elapsed]

### dump\_file()

。。。。。。。。。。。。。。。。。。。。。。。。。

## 总结

bugreport通过socket与dumpstate服务建立通信，在dumpstate.cpp中的dumpstate()方法完成核心功能，该功能依次输出内容项， 主要分为5大类：

current log： kernel,system, event, radio;

last log： kernel, system, radio;

vm traces： just now, last ANR, tombstones

dumpsys： all, checkin, app

system info：cpu, memory, io等

从bugreport内容的输出顺序的角度，再详细列举其内容：

系统build以及运行时长等相关信息；

内存/CPU/进程等信息；

kernel log；

lsof、map及Wait-Channels；

system log；

event log；

radio log;

vm traces：

VM TRACES JUST NOW (/data/anr/traces.txt.bugreport) (抓bugreport时主动触发)

VM TRACES AT LAST ANR (/data/anr/traces.txt) (存在则输出)

TOMBSTONE (/data/tombstones/tombstone\_xx) (存在这输出)

network相关信息；

last kernel log;

last system log;

ip相关信息；

中断向量表

property以及fs等信息

last radio log;

Binder相关信息；

dumpsys all：

dumpsys checkin相关:

dumpsys batterystats电池统计；

dumpsys meminfo内存

dumpsys netstats网络统计；

dumpsys procstats进程统计；

dumpsys usagestats使用情况；

dumpsys package.

dumpsys app相关

dumpsys activity;

dumpsys activity service all;

dumpsys activity provider all.

Tips： bugreport几乎涵盖整个系统信息，内容非常长，每一个子项都以------ xxx ------开头。 例如APP ACTIVITIES的开头便是 ------ APP ACTIVITIES (dumpsys activity all) ------，其中括号内的便是输出该信息指令，即dumpsys activity all，还有可能是内容所在节点，各个子项目类似的规律，看完前面的源码分析过程，相信你肯定能明白。下面一篇文章再进一步从bugreport内容的角度来说明其寓意。

## bugreport实战篇

文章Bugreport源码篇(一)从源码角度阐释了Bugreport，那么本文则是从实战角度来说说，如何看懂bugreport，对于通过bugreport获取到内容往往非常长，这里简单划分为19部分，先整体上简单说说每部分log的title信息。

## ChkBugReport

## REF

<http://gityuan.com/2016/06/11/bugreport-2/>

# Bootanimation

## 概述

### 目录结构

frameworks/base/cmds/bootanimation

目录中包含如下文件

文件名 作用

Android.mk mk文件，编译模块使用

AudioPlayer.cpp、AudioPlayer.h 音频播放

BootAnimation.cpp、BootAnimation.h 开机动画的源文件和头文件

bootanimation\_main.cpp 开机动画的入口

### 功能原理

Android系统的开机动画是由应用程序bootanimation来实现的，它位于/system/bin目录下，它的具体实现可以参考Android系统的开机画面显示过程分析一文。为什么要选择Android系统的开机动画来分析Android应用程序与SurfaceFlinger服务的连接过程呢？首先，负责实现开机动画的应用程序bootanimation也是一个Android应用程序，只不过它是使用C++语言来开发的；其次，应用程序bootanimation是与UI相关的，即它与使用Java语言来开发的标准Android应用程序一样，都需要使用SurfaceFlinger服务来创建和渲染自己的Surface，即开机动画；第三，由于应用程序bootanimation不涉及用户输入，即不需要与用户进行交互（触摸屏、键盘等），因此它能够以最简洁的方式来体现Android应用程序与SurfaceFlinger服务的关系。

## 调用入口

调用入口

应用程序bootanimation来实现的位于/system/bin目录下，bootanimation在init.rc中的定义如下：

1. service bootanim /system/bin/bootanimation
2. class core
3. user graphics
4. group graphics audio
5. disabled
6. oneshot

可见，由于设置为"disable"，该应用在init启动过程中是不会启动的，需要其他地方显示的调用才能启动。那是什么时候启动的呢？当SurfaceFlinger服务启动时，会修改系统属性值ctl.start，通知init进程启动bootanimation。

在早期的Android版本中，SurfaceFlinger服务是由SystemServer启动的。但在Android5.1中，该服务是init进程启动过程中就启动了。在init.rc中能看到对该服务的描述：

1. service surfaceflinger /system/bin/surfaceflinger
2. class core
3. user system
4. group graphics drmrpc
5. onrestart restart zygote

SurfaceFlinger服务源码路径为：frameworks\native\services\surfaceflinger

服务的入口在main\_surfaceflinger.cpp中，具体为：

1. int main(int, char\*\*) {
2. // When SF is launched in its own process, limit the number of
3. // binder threads to 4.
4. ProcessState::self()-**>**setThreadPoolMaxThreadCount(4);
6. // start the thread pool
7. sp**<ProcessState>** ps(ProcessState::self());
8. ps-**>**startThreadPool();
10. // instantiate surfaceflinger
11. sp**<SurfaceFlinger>** flinger = new SurfaceFlinger();
13. #if defined(HAVE\_PTHREADS)
14. setpriority(PRIO\_PROCESS, 0, PRIORITY\_URGENT\_DISPLAY);
15. #endif
16. set\_sched\_policy(0, SP\_FOREGROUND);
18. // initialize before clients can connect
19. flinger-**>**init();
21. // publish surface flinger
22. sp**<IServiceManager>** sm(defaultServiceManager());
23. sm-**>**addService(String16(SurfaceFlinger::getServiceName()), flinger, false);
25. // run in this thread
26. flinger-**>**run();
28. return 0;
29. }

主要工作是：新建一个SurfaceFlinger对象，然后调用其中的init()方法，最后调用其中的run()方法。

下面主要看一下SurfaceFlinger::init()方法，具体代码为：

1. void SurfaceFlinger::init() {
2. ALOGI(  "SurfaceFlinger's main thread ready to run. "
3. "Initializing graphics H/W...");
5. ......
7. // start boot animation
8. startBootAnim();
9. }

SurfaceFlinger调用了startBootAnim()启动了开机动画。该函数代码如下：

1. void SurfaceFlinger::startBootAnim() {
2. // start boot animation
3. property\_set("service.bootanim.exit", "0");
4. property\_set("ctl.start", "bootanim");
5. }

剩下部分属于属性系统，不做分析了

开机动画的显示过程

http://blog.csdn.net/u010753159/article/details/51325500

## 源码分析

### 入口类bootanimation\_main.cpp

bootanimation\_main.cpp

文件中定义main函数，也就是C语言中的执行文件的入口函数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | <code>int main(int argc, char\*\* argv)  {  //宏定义判断是否设置进程的优先级  #if defined(HAVE\_PTHREADS)      setpriority(PRIO\_PROCESS, 0, ANDROID\_PRIORITY\_DISPLAY);  #endif        char value[PROPERTY\_VALUE\_MAX];      //这个配置项决定是否存在开机动画      property\_get("debug.sf.nobootanimation", value, "0");      int noBootAnimation = atoi(value);      ALOGI\_IF(noBootAnimation,  "boot animation disabled");      if (!noBootAnimation) {          //创建ProcessSate对象          // 这个过程会打开/dev/binder设备，形成和内核binder机制的交互的通道; 映射fd到内存          sp<processstate> proc(ProcessState::self());          //创建线程并加入到线程池          ProcessState::self()->startThreadPool();            // 创建开机动画对象          sp<bootanimation> boot = new BootAnimation();          //把主线程加入到线程池          IPCThreadState::self()->joinThreadPool();        }      return 0;  } |

### BootAnimation.cpp

Sdf

1. namespace android {
3. class AudioPlayer;
4. class Surface;
5. class SurfaceComposerClient;
6. class SurfaceControl;
8. *// ---------------------------------------------------------------------------*
10. class BootAnimation : public Thread, public IBinder::DeathRecipient
11. {
12. public:
13. BootAnimation();
14. virtual ~BootAnimation();
16. sp<surfacecomposerclient> session() const;
18. private:
19. virtual bool threadLoop();
20. virtual status\_t readyToRun();
21. virtual void onFirstRef();
22. virtual void binderDied(const wp<ibinder>& who);
24. *//Texture类定义*
25. struct Texture {
26. GLint w; *//宽度*
27. GLint h; *//高度*
28. GLuint name; *//名称*
29. };
30. *//动画内容结构体*
31. struct Animation {
32. *//动画帧*
33. struct Frame {
34. String8 name;
35. FileMap\* map;
36. mutable GLuint tid;
37. bool operator < (const Frame& rhs) const {
38. return name < rhs.name;
39. }
40. };
41. *//动画部分，因为动画可能是由几个部分组成*
42. struct Part {
43. int count;
44. int pause;
45. String8 path;
46. SortedVector frames;
47. bool playUntilComplete;
48. float backgroundColor[3];
49. FileMap\* audioFile;
50. };
51. int fps;
52. int width;
53. int height;
54. Vector<part> parts;
55. };
57. status\_t initTexture(Texture\* texture, AssetManager& asset, const char\* name);
58. status\_t initTexture(const Animation::Frame& frame);
59. bool android();
60. bool readFile(const char\* name, String8& outString);
61. bool movie();
63. void checkExit();
65. sp<surfacecomposerclient> mSession;
66. sp mAudioPlayer;
67. AssetManager mAssets;
68. Texture mAndroid[2];
69. int mWidth;
70. int mHeight;
71. EGLDisplay mDisplay;
72. EGLDisplay mContext;
73. EGLDisplay mSurface;
74. sp<surfacecontrol> mFlingerSurfaceControl;
75. sp<surface> mFlingerSurface;
76. ZipFileRO \*mZip;
77. };
79. *// ---------------------------------------------------------------------------*
81. }; *// namespace android*

#### 构造函数

1. BootAnimation::BootAnimation() : Thread(false), mClockEnabled(true), mTimeIsAccurate(false),
2. mTimeCheckThread(NULL) {
3. *////用于界面显示的mSession，与SurfaceFlinger交互的客户端*
4. mSession = new SurfaceComposerClient();
6. *// If the system has already booted, the animation is not being used for a boot.*
7. mSystemBoot = !property\_get\_bool(BOOT\_COMPLETED\_PROP\_NAME, 0);

mSession是BootAnimation类的成员变量，它是一个类型为SurfaceComposerClient的强指针，即sp<SurfaceComposerClient>

然后执行

void BootAnimation::onFirstRef() {

    status\_t err = mSession->linkToComposerDeath(this);

    ALOGE\_IF(err, "linkToComposerDeath failed (%s) ", strerror(-err));

    if (err == NO\_ERROR) {

        run("BootAnimation", PRIORITY\_DISPLAY);

    }

}

由于BootAnimation继承Thread类，首先会调用readyToRun函数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | <code>status\_t BootAnimation::readyToRun() {      mAssets.addDefaultAssets();        sp<ibinder> dtoken(SurfaceComposerClient::getBuiltInDisplay(              ISurfaceComposer::eDisplayIdMain));      DisplayInfo dinfo;      status\_t status = SurfaceComposerClient::getDisplayInfo(dtoken, &dinfo);      if (status)          return -1;        // create the native surface      sp<surfacecontrol> control = session()->createSurface(String8("BootAnimation"),              dinfo.w, dinfo.h, PIXEL\_FORMAT\_RGB\_565);        SurfaceComposerClient::openGlobalTransaction();      control->setLayer(0x40000000);      SurfaceComposerClient::closeGlobalTransaction();        sp<surface> s = control->getSurface();        // initialize opengl and egl      const EGLint attribs[] = {              EGL\_RED\_SIZE,   8,              EGL\_GREEN\_SIZE, 8,              EGL\_BLUE\_SIZE,  8,              EGL\_DEPTH\_SIZE, 0,              EGL\_NONE      };      EGLint w, h, dummy;      EGLint numConfigs;      EGLConfig config;      EGLSurface surface;      EGLContext context;        EGLDisplay display = eglGetDisplay(EGL\_DEFAULT\_DISPLAY);        eglInitialize(display, 0, 0);      eglChooseConfig(display, attribs, &config, 1, &numConfigs);      surface = eglCreateWindowSurface(display, config, s.get(), NULL);      context = eglCreateContext(display, config, NULL, NULL);      eglQuerySurface(display, surface, EGL\_WIDTH, &w);      eglQuerySurface(display, surface, EGL\_HEIGHT, &h);        if (eglMakeCurrent(display, surface, surface, context) == EGL\_FALSE)          return NO\_INIT;        mDisplay = display;      mContext = context;      mSurface = surface;      mWidth = w;      mHeight = h;      mFlingerSurfaceControl = control;      mFlingerSurface = s;        // If the device has encryption turned on or is in process      // of being encrypted we show the encrypted boot animation.      char decrypt[PROPERTY\_VALUE\_MAX];      property\_get("vold.decrypt", decrypt, "");        bool encryptedAnimation = atoi(decrypt) != 0 || !strcmp("trigger\_restart\_min\_framework", decrypt);        ZipFileRO\* zipFile = NULL;      if ((encryptedAnimation &&              (access(SYSTEM\_ENCRYPTED\_BOOTANIMATION\_FILE, R\_OK) == 0) &&              ((zipFile = ZipFileRO::open(SYSTEM\_ENCRYPTED\_BOOTANIMATION\_FILE)) != NULL)) ||                ((access(OEM\_BOOTANIMATION\_FILE, R\_OK) == 0) &&              ((zipFile = ZipFileRO::open(OEM\_BOOTANIMATION\_FILE)) != NULL)) ||                ((access(SYSTEM\_BOOTANIMATION\_FILE, R\_OK) == 0) &&              ((zipFile = ZipFileRO::open(SYSTEM\_BOOTANIMATION\_FILE)) != NULL))) {          mZip = zipFile;      }        return NO\_ERROR;  }</surface></surfacecontrol></ibinder></code> |

上面主要做两个操作：

初始化显示界面用于播放开机动画，egl的一些内容；  
2. 根据手机是否[加密](https://www.2cto.com/article/jiami/)选择不同的开机动画文件，然后拿到入口zipFile  
其中用到了一个文件路径在BootAnimation.cpp的开头有定义

[?](https://www.2cto.com/kf/201608/536186.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | <code>#define OEM\_BOOTANIMATION\_FILE "/oem/media/bootanimation.zip"   //这个应该是OEM厂商自己定制    #define SYSTEM\_BOOTANIMATION\_FILE "/system/media/bootanimation.zip"   //正常情况下的Android原始开机动画    #define SYSTEM\_ENCRYPTED\_BOOTANIMATION\_FILE "/system/media/bootanimation |

readyToRun方法执行完之后，接着看threaLoop函数

### threaLoop

dsf

bool BootAnimation::threadLoop()

{

bool r;

// We have no bootanimation file, so we use the stock android logo

// animation.

if (mZipFileName.isEmpty()) {

r = android();

} else {//存在开机动画文件，则执行对应的开机动画文件解析出来的内容

r = movie();

}

eglMakeCurrent(mDisplay, EGL\_NO\_SURFACE, EGL\_NO\_SURFACE, EGL\_NO\_CONTEXT);

eglDestroyContext(mDisplay, mContext);

eglDestroySurface(mDisplay, mSurface);

mFlingerSurface.clear();

mFlingerSurfaceControl.clear();

eglTerminate(mDisplay);

IPCThreadState::self()->stopProcess();

return r;

#### android

fg

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | <code>bool BootAnimation::android()  {      //初始化两个纹理用于显示logo      initTexture(&mAndroid[0], mAssets, "images/android-logo-mask.png");      initTexture(&mAndroid[1], mAssets, "images/android-logo-shine.png");        // clear screen 清屏      glShadeModel(GL\_FLAT);      glDisable(GL\_DITHER);      glDisable(GL\_SCISSOR\_TEST);      glClearColor(0,0,0,1);      glClear(GL\_COLOR\_BUFFER\_BIT);      eglSwapBuffers(mDisplay, mSurface);        glEnable(GL\_TEXTURE\_2D);      glTexEnvx(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);        const GLint xc = (mWidth  - mAndroid[0].w) / 2;      const GLint yc = (mHeight - mAndroid[0].h) / 2;      const Rect updateRect(xc, yc, xc + mAndroid[0].w, yc + mAndroid[0].h);        glScissor(updateRect.left, mHeight - updateRect.bottom, updateRect.width(),              updateRect.height());        // Blend state      glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);      glTexEnvx(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);        const nsecs\_t startTime = systemTime();      //不停的显示知道exitPending()返回true      do {          ......          EGLBoolean res = eglSwapBuffers(mDisplay, mSurface);          ......            checkExit();      } while (!exitPending());      glDeleteTextures(1, &mAndroid[0].name);      glDeleteTextures(1, &mAndroid[1].name);      return false;  }</code> |

主要工作，初始化显示的logo纹理，不断刷新界面直到exitPending()返回true，exitPenging()是Thread类中定义的函数，在checkExit()函数中通过requestExit()来执行退出

[?](https://www.2cto.com/kf/201608/536186.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <code>void BootAnimation::checkExit() {      // Allow surface flinger to gracefully request shutdown      char value[PROPERTY\_VALUE\_MAX];      //EXIT\_PROP\_NAME "service.bootanim.exit" 这个配置项在SurfaceFlinger的bootFinished函数中设置为1，然后这里才能退出开机动画，这个过程设计到开机启动到launcher的整个过程，这里不赘述      property\_get(EXIT\_PROP\_NAME, value, "0");      int exitnow = atoi(value);      if (exitnow) {          requestExit();          if (mAudioPlayer != NULL) {              mAudioPlayer->requestExit();          }      }  }</code> |

#### movie()

ie()这个函数有点长，我们截断一点一点的看

##### parseAnimationDesc

g if (!readFile("desc.txt", desString)) {

return false;

}

char const\* s = desString.string();

//读取desc.txt文件

// Create and initialize an AudioPlayer if we have an audio\_conf.txt file

String8 audioConf;

//判断是否需要创建AudioPlayer,这部分我们暂时不关注

if (readFile("audio\_conf.txt", audioConf)) {

mAudioPlayer = new AudioPlayer;

if (!mAudioPlayer->init(audioConf.string())) {

ALOGE("mAudioPlayer.init failed");

mAudioPlayer = NULL;

}

}

解析desc.txt文件，就是上面拿到的那个char const \*s

for (;;) {

//一行一行的截取

const char\* endl = strstr(s, "\n");

if (!endl) break;

String8 line(s, endl - s);

const char\* l = line.string();

//几个需要捕获的参数,帧率 宽高，次数

int fps, width, height, count, pause;

char path[ANIM\_ENTRY\_NAME\_MAX];

char color[7] = "000000"; // default to black if unspecified

char pathType;

//读取帧率和宽高

if (sscanf(l, "%d %d %d", &width, &height, &fps) == 3) {

// ALOGD("> w=%d, h=%d, fps=%d", width, height, fps);

animation.width = width;

animation.height = height;

animation.fps = fps;

}

//或者读取part内容

else if (sscanf(l, " %c %d %d %s #%6s", &pathType, &count, &pause, path, color) >= 4) {

// ALOGD("> type=%c, count=%d, pause=%d, path=%s, color=%s", pathType, count, pause, path, color);

Animation::Part part;

part.playUntilComplete = pathType == 'c';

part.count = count;

part.pause = pause;

part.path = path;

part.audioFile = NULL;

if (!parseColor(color, part.backgroundColor)) {

ALOGE("> invalid color '#%s'", color);

part.backgroundColor[0] = 0.0f;

part.backgroundColor[1] = 0.0f;

part.backgroundColor[2] = 0.0f;

}

animation.parts.add(part);

}

s = ++endl;

}

读取所有的数据

// read all the data structures

const size\_t pcount = animation.parts.size();

void \*cookie = NULL;

if (!mZip->startIteration(&cookie)) {

return false;

}

ZipEntryRO entry;

char name[ANIM\_ENTRY\_NAME\_MAX];

while ((entry = mZip->nextEntry(cookie)) != NULL) {

const int foundEntryName = mZip->getEntryFileName(entry, name, ANIM\_ENTRY\_NAME\_MAX);

if (foundEntryName > ANIM\_ENTRY\_NAME\_MAX || foundEntryName == -1) {

ALOGE("Error fetching entry file name");

continue;

}

const String8 entryName(name);

const String8 path(entryName.getPathDir());

const String8 leaf(entryName.getPathLeaf());

if (leaf.size() > 0) {

for (size\_t j=0 ; j<pcount files="" if="" int="" mzip-="" only="" path="=" png="" stored="" supports="">getEntryInfo(entry, &method, NULL, NULL, NULL, NULL, NULL)) {

if (method == ZipFileRO::kCompressStored) {

FileMap\* map = mZip->createEntryFileMap(entry);

if (map) {

Animation::Part& part(animation.parts.editItemAt(j));

if (leaf == "audio.wav") {

// a part may have at most one audio file

part.audioFile = map;

} else {

Animation::Frame frame;

frame.name = leaf;

frame.map = map;

part.frames.add(frame);

}

}

}

}

}

}

}

}

mZip->endIteration(cookie);

##### 显示动画

Rgs

// clear screen

glShadeModel(GL\_FLAT);

glDisable(GL\_DITHER);

glDisable(GL\_SCISSOR\_TEST);

glDisable(GL\_BLEND);

glClearColor(0,0,0,1);

glClear(GL\_COLOR\_BUFFER\_BIT);

eglSwapBuffers(mDisplay, mSurface);

glBindTexture(GL\_TEXTURE\_2D, 0);

glEnable(GL\_TEXTURE\_2D);

glTexEnvx(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

const int xc = (mWidth - animation.width) / 2;

const int yc = ((mHeight - animation.height) / 2);

nsecs\_t lastFrame = systemTime();

nsecs\_t frameDuration = s2ns(1) / animation.fps;

Region clearReg(Rect(mWidth, mHeight));

clearReg.subtractSelf(Rect(xc, yc, xc+animation.width, yc+animation.height));

for (size\_t i=0 ; i<pcount 0="" const="" size\_t="" fcount="part.frames.size();" for="" int="" r="=" part.count="" exit="" any="" non="" playuntil="" complete="" parts="" immediately="" play="" audio="" file="" the="" first="" time="" we="" animate="" part="" if="" maudioplayer="" maudioplayer-="" data-cke-pa-only="">playFile(part.audioFile);

}

glClearColor(

part.backgroundColor[0],

part.backgroundColor[1],

part.backgroundColor[2],

1.0f);

for (size\_t j=0 ; j<fcount const="" nsecs\_t="" lastframe="systemTime();" if="" r=""> 0) {

glBindTexture(GL\_TEXTURE\_2D, frame.tid);

} else {

if (part.count != 1) {

glGenTextures(1, &frame.tid);

glBindTexture(GL\_TEXTURE\_2D, frame.tid);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

}

initTexture(frame);

}

if (!clearReg.isEmpty()) {

Region::const\_iterator head(clearReg.begin());

Region::const\_iterator tail(clearReg.end());

glEnable(GL\_SCISSOR\_TEST);

while (head != tail) {

const Rect& r(\*head++);

glScissor(r.left, mHeight - r.bottom,

r.width(), r.height());

glClear(GL\_COLOR\_BUFFER\_BIT);

}

glDisable(GL\_SCISSOR\_TEST);

}

glDrawTexiOES(xc, yc, 0, animation.width, animation.height);

eglSwapBuffers(mDisplay, mSurface);

nsecs\_t now = systemTime();

nsecs\_t delay = frameDuration - (now - lastFrame);

//ALOGD("%lld, %lld", ns2ms(now - lastFrame), ns2ms(delay));

lastFrame = now;

if (delay > 0) {

struct timespec spec;

spec.tv\_sec = (now + delay) / 1000000000;

spec.tv\_nsec = (now + delay) % 1000000000;

int err;

do {

err = clock\_nanosleep(CLOCK\_MONOTONIC, TIMER\_ABSTIME, &spec, NULL);

} while (err<0 && errno == EINTR);

}

checkExit();

}

usleep(part.pause \* ns2us(frameDuration));

// For infinite parts, we've now played them at least once, so perhaps exit

if(exitPending() && !part.count)

break;

}

// free the textures for this part

if (part.count != 1) {

for (size\_t j=0 ; j<fcount const="" pre="">

## 修改Android系统的开机动画

Android系统中，与开机动画相关的文件都放在一个叫bootanimation.zip的压缩包文件内，bootanimation.zip随system.img一起发布。需要放到system/media目录中.如果我们要查找系统自带的bootanimation.zip文件，可以将system目录还原，在其下的media目录下存在一个bootanimation.zip文件，替换该文件，我们就能完成开机动画的修改。bootanimation.zip文件解压看看里面的具体包含的内容：首先包含一个desc.txt文件，还会包含若干类似part0，part1的目录，其中desc.txt文件是必须的，part0、part1等目录至少要存在一个。part0、part1目录中存放的是图像文件，诸如001.png、002.png类似的命名有规律的图像文件。Android系统中读取这些静态图像，并按一定的显示规律，和频率产生动画效果，desc.txt文件就是用来描述加载规律和频率信息的文件。

为了使Android拥有Root权限，需要在代码中执行su命令。在执行su命令的过程中会创建一个新的拥有root权限的进程，通过该进程进行的任何操作都是在root权限下进行的

//执行su命令，并创建一个新进程（Process对象）

Process process = Runtime.getRuntime().exec("su");

//获取新进程的OutputStream对象，可以通过该对象发出要执行的命令

OutputStream os = process.getOutputStream();

//获取新进程的InputStream对象，可以通过该对象获取命令执行后返回的数据

InputStream is = process.getInputStream();

......

最后还要实现设备的重启，在Android系统中，要实现重启有两种方式：

1. 执行reboot命令.执行reboot命令需要root权限。也就是说，只要拥有了root权限，任何应用程序都可以重启Android设备

2. 调用PowerManager.reboot命令。该种重启方式只有System用户能使用，自由system用户才允许设置android.permission.REBOOT权限

PowerManager pm = (PowerManager)getSystemService(Context.POWER\_SERVICE);

pm.reboot("change boot animation");

最后执行重启设备命令需要在清单配置文件中配置android.permission.REBOOT权限