

Android 文件系统的移植

作者：汕头大学—黄珠唐

时间：2009 年 10 月 29 日

文章摘要

主要介绍 linux 内核启动过程以及挂载 android 根文件系统的过程，以及介绍 android 源代码中文件系统部分的浅析。

目录

- 1) android 源代码文件系统部分介绍
- 2) Linux 内核启动挂载 android 根文件系统过程分析
- 3) Android 文件系统初始化核心 Init.c 文件分析
- 4) 初始化核心的核心 init.rc 文件分析

1) android 源代码文件系统部分介绍

从 google 获得源代码后，在 platform 目录下 make 编译后我们可以看到生成了 out 目录。

```
[root@monet android platform]# ls
bionic  build  dalvik  external  hardware  lib  prebuilt  system
Makefile

[root@monet android platform]# ll
total 60
drwxrwxrwx  2 root root 4096 2009-10-18 01:42 bionic
drwxrwxrwx  9 root root 4096 2009-10-18 04:03 build
drwxrwxrwx  5 root root 4096 2009-10-18 04:03 bootable
drwxrwxrwx  8 root root 4096 2009-10-18 04:03 dalvik
drwxrwxrwx 18 root root 4096 2009-10-18 04:04 development
drwxrwxrwx 17 root root 4096 2009-10-18 04:05 external
drwxrwxrwx 75 root root 4096 2009-10-18 04:11 framework
drwxrwxrwx  5 root root 4096 2009-10-18 04:12 hardware
drwxrwxrwx  6 root root 4096 2009-10-18 04:12 Makefile
-rwxrwxrwx  1 root root   87 2009-10-18 04:03 bionic
drwxrwxrwx  5 root root 4096 2009-10-20 10:26 build
drwxrwxrwx  5 root root 4096 2009-10-18 04:12 bootable
drwxrwxrwx 11 root root 4096 2009-10-18 04:15 dalvik
drwxrwxrwx  7 root root 4096 2009-10-18 04:15 development
drwxrwxrwx  7 root root 4096 2009-10-18 04:15 system
drwxrwxrwx  7 root root 4096 2009-10-18 04:15 system
[root@monet android platform]#
```

主要源代码目录介绍

Makefile （全局的Makefile）

bionic （Bionic 含义为仿生，这里面是一些基础的库的源代码）

bootable （引导加载器）

build （build 目录中的内容不是目标所用的代码，而是编译和配置所需要的脚本和工具）

dalvik （JAVA 虚拟机）

development （程序开发所需要的模板和工具）

external （目标机器使用的一些库）

frameworks （应用程序的框架层）

hardware （与硬件相关的库）

packages （Android 的各种应用程序）

prebuilt （Android 在各种平台下编译的预置脚本）

recovery （与目标的恢复功能相关）

system （Android 的底层的一些库）

out（编译完成后产生的目录，也就是我们移植文件系统需要的目录）

让我们打开 out 目录看看里面有什么东西，

```
[root@monet android_platform]# cd out/
[root@monet out]# ls
casecheck.txt CaseCheck.txt host target
[root@monet out]# ll
total 20
-rwxrwxrwx 1 root root 2 2009-11-06 20:17 casecheck.txt
-rwxrwxrwx 1 root root 2 2009-11-06 20:17 CaseCheck.txt
drwxrwxrwx 4 root root 4096 2009-10-20 07:21 host
drwxrwxrwx 4 root root 4096 2009-10-20 07:22 target
drwxrwxrwx 3 root root 4096 2009-10-20 10:26 product
[root@monet out]#
```

```
out/
|-- CaseCheck.txt
|-- casecheck.txt
|-- host
|   |-- common
|   |-- linux-x86
|-- target
|   |-- common
|   |-- product
```

主要的两个目录为 host 和 target，前者表示在主机（x86）生成的工具，后者表示目标机（模认为 ARMv5）运行的内容。

host 目录的结构如下所示：

```
out/host/
|-- common
|   |-- obj                (JAVA 库)
|-- linux-x86
|   |-- bin                (二进制程序)
|   |-- framework          (JAVA 库, *.jar 文件)
|   |-- lib                (共享库*.so)
|   |-- obj                (中间生成的目标文件)
```

host 目录是一些在主机上用的工具，有一些是二进制程序，有一些是 JAVA 的程序。

target 目录的结构如下所示：

```
out/target/
|-- common
|   |-- R                  (资源文件)
|   |-- docs
|   |-- obj                (目标文件)
|-- product
|   |-- generic
```

其中 common 目录表示通用的内容，product 中则是针对产品的内容。

在 common 目录的 obj 中，包含两个重要的目录：

APPS 中包含了 JAVA 应用程序生成的目标，每个应用程序对应其中一个子目录，将结合每个应用程序的原始文件生成 Android 应用程序的 APK 包。

JAVA_LIBRARIES 中包含了 JAVA 的库，每个库对应其中一个子目录。

所以，我们提取文件系统主要是在/out/target/product/generic 目录下

```
[root@monet generic]# pwd
/home/android/android_platform/out/target/product/generic
[root@monet generic]# ll
total 49692
-rwxrwxrwx  1 root root      7 2009-10-20 10:36 android-info.txt
-rwxrwxrwx  1 root root    550 2009-11-06 20:17 clean_steps.mk
drwxrwxrwx  2 root root   4096 2009-10-20 07:44 obj
-rw-r--r--  1 root root  13241 2009-11-07 04:51 installed-files.txt
drwxrwxrwx 13 root root   4096 2009-10-20 10:36 obj
-rwxrwxrwx  1 root root    40 2009-11-06 20:17 previous_build_config.mk
-rwxrwxrwx  1 root root 156417 2009-11-07 04:51 ramdisk.img
drwxrwxrwx  8 root root   4096 2009-10-20 08:03 root
drwxrwxrwx  4 root root   4096 2009-10-20 08:03 symbols
drwxrwxrwx 10 root root   4096 2009-10-20 10:13 system
-rwxrwxrwx  1 root root 50607744 2009-11-07 04:51 system.img
-rwxrwxrwx  1 root root   2112 2009-11-07 04:51 userdata.img
[root@monet generic]#
```

我们可以看到里面有 obj 目录，进入 obj 目录看看

```
[root@monet obj]# ll
total 404
drwxrwxrwx  48 root root   4096 2009-10-20 10:28 APPS
drwxrwxrwx   3 root root   4096 2009-10-20 10:28 DATA
drwxrwxrwx  11 root root   4096 2009-10-20 10:28 FTE
drwxrwxrwx 106 root root   4096 2009-10-20 10:30 EXECUTABLES
drwxrwxrwx   7 root root   4096 2009-10-20 07:21 INCLUDE
drwxrwxrwx   5 root root   4096 2009-10-20 10:13 NEWSHARE
drwxrwxrwx   2 root root   4096 2009-10-20 10:36 obj
drwxrwxrwx   4 root root   4096 2009-11-07 04:06 NOTICE_FILES
-rwxrwxrwx   1 root root 293701 2009-11-07 04:07 NOTICE.html
-rwxrwxrwx   1 root root  56410 2009-11-07 04:07 NOTICE.html.gz
drwxrwxrwx   3 root root   4096 2009-10-20 10:36 PACKAGING
drwxrwxrwx 106 root root   4096 2009-10-20 10:36 SHARED_LIBRARIES
drwxrwxrwx 189 root root  12288 2009-11-07 00:02 STATIC_LIBRARIES
[root@monet obj]#
```

里面是 android 文件系统非常重要的内容，

/obj

APPS（文件系统下/system/apps 目录下的各种应用程序）

SHARED_LIBRARIES（存放所有动态库）

STATIC_LIBRARIES（存放所有静态库）

EXECUTABLES（存放各种可执行文件）

还有其他需要的文件都是在/out/target/product/generic 目录下
我们可以从中提取我们需要的内容。

2) Linux 内核启动挂载 android 根文件系统过程分析

要分析 linux 启动过程，一切要从内核/arch/arm/boot/compressed/head.S 说起，不过我们不介绍得那么详细，我们只看和根文件系统有关的部分。

顺便罗列一下内核启动流程：

UBOOT

/arch/arm/boot/compressed/head.S:

Start:

Decompressed_kernel() //在/arch/arm/boot/compressed/misc.c 中

Call_kernel()

Stext:

/init/main.c

Start_kernel()

Setup_arch()

...

Rest_init()

Init()

Do_basic_setup()

Prepare_namespace()

看到了这里，我已激动得说不出话了，因为来到我与挂载根文件系统最重要的接口函数。

```
/* This is a non __init function. Force it to be noinline otherwise gcc
 * makes it inline to init() and it becomes part of init.text section
 */
static int noinline init_post(void)
{

free_initmem();
```

```

unlock_kernel();
mark_rodata_ro();
system_state = SYSTEM_RUNNING;
numa_default_policy();

if (sys_open((const char __user *) "/dev/console", 0_RDWR, 0) < 0)
printk(KERN_WARNING "Warning: unable to open an initial console.\n");
(void) sys_dup(0);
(void) sys_dup(0);

current->signal->flags |= SIGNAL_UNKILLABLE;
if (ramdisk_execute_command) {
run_init_process(ramdisk_execute_command);
printk(KERN_WARNING "Failed to execute %s\n", ramdisk_execute_command);
}

/*
 * We try each of these until one succeeds.
 * The Bourne shell can be used instead of init if we are*
 * trying to recover a really broken machine.*/
if (execute_command) {
run_init_process(execute_command);
printk(KERN_WARNING "Failed to execute %s.  Attempting ""defaults...\n",
execute_command);
}

run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");

panic("No init found.  Try passing init= option to kernel.");
}

```

其中，我们看到行代码 `run_init_process(execute_command);` `execute_command` 是从 UBOOT 传递过来的参数，一般为 `/init`，也就是调用文件系统里的 `init` 初始化进程。如果找不到 `init` 文件就会在 `run_init_process("/sbin/init");` `run_init_process("/etc/init");` `run_init_process("/bin/init");` `run_init_process("/bin/sh");` 中找，否则报错。

在这里由于我们的根文件系统是从/linuxrc 开始的，所以我硬性把它改为

```
if (execute_command) {
    run_init_process("/linuxrc");
    printk(KERN_WARNING "Failed to execute %s. Attempting "
        "defaults...\n", execute_command);
}
```

3) Android 文件系统初始化核心 Init.c 文件分析

上面我们说的 init 这个文件是由 android 源代码编译来的，编译后在 /out/target/product/generic/root/init

```
[root@monet root]# ls
init.goldfish.rc  init.rc
default.prop      init
[root@monet root]# pwd
/home/android/android_platform/out/target/product/generic/root
[root@monet root]#
```

Root 目录下产生了一个文件系统的雏形。

其源码在 m/system/core/init/init.c

```
Android.mk  devices.h  keywords.h  property_service.c
bootchart.c grab-bootchart.sh logo.c  property_service.h
bootchart.h init.c  MODULE_LICENSE_APACHE2  README.BOOTCHART
builtins.c  init.c~  NOTICE  readme.txt
devices.c   init.h   parser.c  util.c
[root@monet init]# pwd
/home/android/android_platform/system/core/init
[root@monet init]#
```

Init.c 主要做了什么呢？我们看看下面这个比较详细的分析，为网上资料整理而来。

1) 安装 SIGCHLD 信号。（如果父进程不等待子进程结束，子进程将成为僵尸进程（zombie）从而占用系统资源。因此需要对 SIGCHLD 信号做出处理，回收僵尸进程的资源，避免造成不必要的资源浪费。）

(2) 对 umask 进行清零。

何为 umask，请看 <http://www.szstudy.cn/showArticle/53978.shtml>

(3) 为 rootfs 建立必要的文件夹，并挂载适当的分区。

```
/dev (tmpfs)
/dev/pts (devpts)
/dev/socket
/proc (proc)
/sys (sysfs)
```


(4) 创建/dev/null 和/dev/kmsg 节点。

(5) 解析/init.rc, 将所有服务和操作信息加入链表。

(6) 从/proc/cmdline 中提取信息内核启动参数, 并保存到全局变量。

(7) 先从上一步获得的全局变量中获取信息硬件信息和版本号, 如果没有则从/proc/cpuinfo 中提取, 并保存到全局变量。

(8) 根据硬件信息选择一个/init.(硬件).rc, 并解析, 将服务和操作信息加入链表。

在 G1 的 ramdisk 根目录下有两个/init.(硬件).rc: init.goldfish.rc 和 init.trout.rc, init 程序会根据上一步获得的硬件信息选择一个解析。

(9) 执行链表中带有“early-init”触发的命令。

(10) 遍历/sys 文件夹, 是内核产生设备添加事件(为了自动产生设备节点)。

(11) 初始化属性系统, 并导入初始化属性文件。

(12) 从属性系统中得到 ro.debuggable, 若为 1, 则初始化 keychord 監聽。

(13) 打開 console, 如果 cmdline 中沒有指定 console 則打開默認的 /dev/console

(14) 讀取/initlogo.rle (一張 565 rle 壓縮的位圖), 如果成功則在 /dev/graphics/fb0 顯示 Logo, 如果失敗則將/dev/tty0 設為 TEXT 模式并打開/dev/tty0, 輸出文本“ANDROID”字樣。

(15) 判斷 cmdline 中的參數, 并设置属性系统中的参数:

1、 如果 bootmode 為

- factory, 設置 ro.factorytest 值為 1

- factory2, 設置 ro.factorytest 值為 2

- 其他的設 ro.factorytest 值為 0

2、 如果有 serialno 参数, 則設置 ro.serialno, 否則為“”

3、 如果有 bootmod 参数, 則設置 ro.bootmod, 否則為“unknown”

4、 如果有 baseband 参数, 則設置 ro.baseband, 否則為“unknown”

5、 如果有 carrier 参数, 則設置 ro.carrier, 否則為“unknown”

6、 如果有 bootloader 参数, 則設置 ro.bootloader, 否則為“unknown”

7、 通过全局变量 (前面从/proc/cpuinfo 中提取的) 設置 ro.hardware 和 ro.version。

(16) 執行所有触发标识为 init 的 action。

(17) 開始 property 服務, 讀取一些 property 文件, 這一動作必須在前面那些 ro.foo 設置后做, 以便/data/local.prop 不能干預到他們。

- /system/build.prop

- /system/default.prop

- /data/local.prop

- 在讀取默認的 property 后讀取 presistent propertie, 在/data/property 中

(18) 為 sigchld handler 創建信號機制

(19) 確認所有初始化工作完成:

device_fd(device init 完成)

property_set_fd(property server start 完成)

signal_recv_fd (信號機制建立)

(20) 執行所有触发标识为 early-boot 的 action

- (21) 執行所有触发标识为 boot 的 action
- (22) 基于當前 property 狀態，執行所有触发标识为 property 的 action
- (23) 注册輪詢事件：
 - device_fd
 - property_set_fd
 - signal_recv_fd
 - 如果有 keychord，則注册 keychord_fd
- (24) 如果支持 BOOTCHART, 則初始化 BOOTCHART
- (25) 進入主進程循環：
 - 重置輪詢事件的接受狀態，revents 為 0
 - 查詢 action 隊列，并执行。
 - 重啟需要重啟的服務
 - 輪詢注册的事件
 - 如果 signal_recv_fd 的 revents 為 POLLIN，則得到一個信號，獲取并處理
 - 如果 device_fd 的 revents 為 POLLIN, 調用 handle_device_fd
 - 如果 property_fd 的 revents 為 POLLIN, 調用 handle_property_set_fd
 - 如果 keychord_fd 的 revents 為 POLLIN, 調用 handle_keychord

到了这里，整个 android 文件系统已经起来了。

4) 初始化核心的核心 init.rc 文件分析

在上面红色那一行 (5) 解析/init.rc，将所有服务和操作信息加入链表。

parse_config_file("/init.rc");//在 init.c 中代码

贴出 init.rc 脚本

on init

sysclktz 0

loglevel 3

setup the global environment

export PATH /bin:/sbin:/system/sbin:/system/bin:/system/xbin

export LD_LIBRARY_PATH /lib:/system/lib

export ANDROID_BOOTLOGO 1

export ANDROID_ROOT /system

export ANDROID_ASSETS /system/app

export ANDROID_DATA /data

export EXTERNAL_STORAGE /sdcard

export

BOOTCLASSPATH

/system/framework/core.jar:/system/framework/ext.jar:/system/framework

```
k/framework.jar:/system/framework/android.policy.jar:/system/framework/services.jar
```

```
symlink /dev/snd/audio /dev/eac
```

```
# Backward compatibility
```

```
symlink /system/etc /etc
```

```
# create mountpoints and mount tmpfs on sqlite_stmt_journals
```

```
mkdir /sdcard 0000 system system
```

```
mkdir /system
```

```
mkdir /data 0771 system system
```

```
mkdir /cache 0770 system cache
```

```
mkdir /sqlite_stmt_journals 01777 root root
```

```
# mount tmpfs tmpfs /sqlite_stmt_journals size=4m
```

```
# mount rootfs rootfs / ro remount
```

```
write /proc/sys/kernel/panic_on_oops 1
```

```
write /proc/sys/kernel/hung_task_timeout_secs 0
```

```
write /proc/cpu/alignment 4
```

```
write /proc/sys/kernel/sched_latency_ns 10000000
```

```
write /proc/sys/kernel/sched_wakeup_granularity_ns 2000000
```

```
# mount mtd partitions
```

```
# Mount /system rw first to give the filesystem a chance to save a checkpoint
```

```
# mount yaffs2 mtd@system /system
```

```
# mount yaffs2 mtd@system /system ro remount
```

```
# We chown/chmod /data again so because mount is run as root + defaults
```

```
#mount ext3 /dev/block/mmcblk0p2 /data nosuid nodev
```

```
chown system system /data
```

```
chmod 0771 /data
```

```
# Same reason as /data above
```

```
# mount yaffs2 mtd@cache /cache nosuid nodev
```

```
chown system cache /cache
```

```
chmod 0770 /cache
```

```
# This may have been created by the recovery system with odd permissions
```

```
chown system system /cache/recovery
```

```
chmod 0770 /cache/recovery
```

```

# create basic filesystem structure
mkdir /data/misc 01771 system misc
mkdir /data/misc/hcid 0770 bluetooth bluetooth
mkdir /data/local 0771 shell shell
mkdir /data/local/tmp 0771 shell shell
mkdir /data/data 0771 system system
mkdir /data/app-private 0771 system system
mkdir /data/app 0771 system system
mkdir /data/property 0700 root root

# create dalvik-cache and double-check the perms
mkdir /data/dalvik-cache 0771 system system
chown system system /data/dalvik-cache
chmod 0771 /data/dalvik-cache

# create the lost+found directories, so as to enforce our permissions
mkdir /data/lost+found 0770
mkdir /cache/lost+found 0770

# double check the perms, in case lost+found already exists, and set
owner
chown root root /data/lost+found
chmod 0770 /data/lost+found
chown root root /cache/lost+found
chmod 0770 /cache/lost+found

on boot
# basic network init
ifup lo
hostname localhost
domainname localdomain

# set RLIMIT_NICE to allow priorities from 19 to -20
setrlimit 13 40 40

# Define the oom_adj values for the classes of processes that can be
# killed by the kernel. These are used in ActivityManagerService.
setprop ro.FOREGROUND_APP_ADJ 0
setprop ro.VISIBLE_APP_ADJ 1
setprop ro.SECONDARY_SERVER_ADJ 2
setprop ro.HIDDEN_APP_MIN_ADJ 7
setprop ro.CONTENT_PROVIDER_ADJ 14
setprop ro.EMPTY_APP_ADJ 15

```

```

# Define the memory thresholds at which the above process classes will
# be killed.  These numbers are in pages (4k).
    setprop ro.FOREGROUND_APP_MEM 1536
    setprop ro.VISIBLE_APP_MEM 2048
    setprop ro.SECONDARY_SERVER_MEM 4096
    setprop ro.HIDDEN_APP_MEM 5120
    setprop ro.CONTENT_PROVIDER_MEM 5632
    setprop ro.EMPTY_APP_MEM 6144

# Write value must be consistent with the above properties.
    write /sys/module/lowmemorykiller/parameters/adj 0,1,2,7,14,15

    write /proc/sys/vm/overcommit_memory 1
    write /sys/module/lowmemorykiller/parameters/minfree
1536,2048,4096,5120,5632,6144

# Set init its forked children's oom_adj.
write /proc/1/oom_adj -16

# Permissions for System Server and daemons.
chown radio system /sys/android_power/state
chown radio system /sys/android_power/request_state
chown radio system /sys/android_power/acquire_full_wake_lock
chown radio system /sys/android_power/acquire_partial_wake_lock
chown radio system /sys/android_power/release_wake_lock
chown radio system /sys/power/state
chown radio system /sys/power/wake_lock
chown radio system /sys/power/wake_unlock
chmod 0660 /sys/power/state
chmod 0660 /sys/power/wake_lock
chmod 0660 /sys/power/wake_unlock
chown system system /sys/class/timed_output/vibrator/enable
chown system system /sys/class/leds/keyboard-backlight/brightness
chown system system /sys/class/leds/lcd-backlight/brightness
chown system system /sys/class/leds/button-backlight/brightness
chown system system /sys/class/leds/red/brightness
chown system system /sys/class/leds/green/brightness
chown system system /sys/class/leds/blue/brightness
chown system system /sys/class/leds/red/device/grpfreq
chown system system /sys/class/leds/red/device/grppwm
chown system system /sys/class/leds/red/device/blink
chown system system /sys/class/leds/red/brightness
chown system system /sys/class/leds/green/brightness

```

```

chown system system /sys/class/leds/blue/brightness
chown system system /sys/class/leds/red/device/grpfreq
chown system system /sys/class/leds/red/device/grppwm
chown system system /sys/class/leds/red/device/blink
chown system system /sys/class/timed_output/vibrator/enable
chown system system /sys/module/sco/parameters/disable_esco
chown system system /sys/kernel/ipv4/tcp_wmem_min
chown system system /sys/kernel/ipv4/tcp_wmem_def
chown system system /sys/kernel/ipv4/tcp_wmem_max
chown system system /sys/kernel/ipv4/tcp_rmem_min
chown system system /sys/kernel/ipv4/tcp_rmem_def
chown system system /sys/kernel/ipv4/tcp_rmem_max
chown root radio /proc/cmdline

# Define TCP buffer sizes for various networks
#   ReadMin, ReadInitial, ReadMax, WriteMin, WriteInitial, WriteMax,
#   setprop                                     net.tcp.buffersize.default
4096, 87380, 110208, 4096, 16384, 110208
#   setprop                                     net.tcp.buffersize.wifi
4095, 87380, 110208, 4096, 16384, 110208
#   setprop                                     net.tcp.buffersize.umts
4094, 87380, 110208, 4096, 16384, 110208
#   setprop                                     net.tcp.buffersize.edge
4093, 26280, 35040, 4096, 16384, 35040
#   setprop                                     net.tcp.buffersize.gprs
4092, 8760, 11680, 4096, 8760, 11680

class_start default

## Daemon processes to be run by init.
##

#service console /system/bin/sh
#   console
service console /bin/busybox sh
#   console

service myInit /bin/busybox sh /system/etc/shine/myInit.rc
#   oneshot

# addb is controlled by the persist.service.adb.enable system property
service addb /sbin/addb
#   disabled

```

```
# addb on at boot in emulator
on property:ro.kernel.qemu=1
    start addb

on property:persist.service.adb.enable=1
    start addb

on property:persist.service.adb.enable=0
    stop addb

service servicemanager /system/bin/servicemanager
    user system
    critical
    onrestart restart zygote
    onrestart restart media

service mounstd /system/bin/mounstd
    socket mounstd stream 0660 root mount

service debuggerd /system/bin/debuggerd

service ril-daemon /system/bin/rild
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    user root
    group radio cache inet misc

service zygote /system/bin/app_process -Xzygote /system/bin --zygote
--start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on

service media /system/bin/mediaserver
    user media
    group system audio camera graphics inet net_bt net_bt_admin

service bootsound /system/bin/playmp3
    user media
    group audio
    oneshot

service dbus /system/bin/dbus-daemon --system --nofork
    socket dbus stream 660 bluetooth bluetooth
```

```

    user bluetooth
    group bluetooth net_bt_admin

#STOPSHIP: disable the verbose logging
service hcid /system/bin/logwrapper /system/bin/hcid -d -s -n -f
/etc/bluetooth/hcid.conf
    socket bluetooth stream 660 bluetooth bluetooth
    socket dbus_bluetooth stream 660 bluetooth bluetooth
    # init.rc does not yet support applying capabilities, so run as root
and
    # let hcid drop uid to bluetooth with the right linux capabilities
    group bluetooth net_bt_admin misc
    disabled

service hfag /system/bin/sdptool add --channel=10 HFAG
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service hsag /system/bin/sdptool add --channel=11 HSAG
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service installd /system/bin/installd
    socket installd stream 600 system system

service      flash_recovery      /system/bin/flash_image      recovery
/system/recovery.img
    oneshot

```

对 init.rc 脚本的解析依赖一定的语法规则, 理解这些语法规则有利于我们很好理解到底 init.rc 执行了哪些命令。

名词解释:

Android 初始化语言由四大类声明组成: 行为类(Actions), 命令类(Commands), 服务类(Services), 选项类(Options)。

初始化语言以行为单位, 由以空格间隔的语言符号组成。C 风格的反斜杠转义符可以用来插入空白到语言符号。双引号也可以用来防止文本被空格分成多个语言符号。当反斜杠在行末时, 作为换行符。

* 以#开始(前面允许空格)的行为注释。

* Actions 和 Services 隐含声明一个新的段落。所有该段落下 Commands 或 Options 的声明属于该段落。第一段落前的 Commands 或 Options 被忽略。

* Actions 和 Services 拥有唯一的命名。在他们之后声明相同命名的类将被当作错误并忽略。

Actions 是一系列命令的命名。Actions 拥有一个触发器(trigger)用来决定 action 何时执行。当一个 action 在符合触发条件被执行时, 如果它还没被加入到待执行队列中的话, 则加入到队列最后。

队列中的 action 依次执行, action 中的命令也依次执行。Init 在执行命令的中间处理其他活动(设备创建/销毁, property 设置, 进程重启)。

Actions 的表现形式:

```
on <trigger>
```

```
    <command>
```

```
    <command>
```

```
    <command>
```

重要的数据结构

两个列表, 一个队列。

```
static list_declare(service_list);
```

```
static list_declare(action_list);
```

```
static list_declare(action_queue);
```

*.rc 脚本中所有 service 关键字定义的服务将会添加到 service_list 列表中。

*.rc 脚本中所有 on 关键开头的项将会被添加到 action_list 列表中。每个 action 列表项都有一个列表, 此列表用来保存该段落下的 Commands

脚本解析过程:

```
parse_config_file("/init.rc")
```

```
int parse_config_file(const char *fn)
```

```
{
```

```
    char *data;
```

```
    data = read_file(fn, 0);
```

```
    if (!data) return -1;
```

```
    parse_config(fn, data);
```

```
    DUMP();
```

```
    return 0;
```

```
}
```

```
static void parse_config(const char *fn, char *s)
```

```
{
```

```
    ...
```

```
    case T_NEWLINE:
```

```
        if (nargs) {
```

```
            int kw = lookup_keyword(args[0]);
```

```
            if (kw_is(kw, SECTION)) {
```

```

        state.parse_line(&state, 0, 0);
        parse_new_section(&state, kw, nargs, args);
    } else {
        state.parse_line(&state, nargs, args);
    }
    nargs = 0;
}
...
}

```

parse_config 会逐行对脚本进行解析，如果关键字类型为 SECTION，那么将会执行 parse_new_section()

类型为 SECTION 的关键字有：on 和 service

关键字类型定义在 Parser.c (system\core\init) 文件中

Parser.c (system\core\init)

```
#define SECTION 0x01
```

```
#define COMMAND 0x02
```

```
#define OPTION 0x04
```

关键字	属性
capability,	OPTION, 0, 0)
class,	OPTION, 0, 0)
class_start,	COMMAND, 1, do_class_start)
class_stop,	COMMAND, 1, do_class_stop)
console,	OPTION, 0, 0)
critical,	OPTION, 0, 0)
disabled,	OPTION, 0, 0)
domainname,	COMMAND, 1, do_domainname)
exec,	COMMAND, 1, do_exec)
export,	COMMAND, 2, do_export)
group,	OPTION, 0, 0)
hostname,	COMMAND, 1, do_hostname)
ifup,	COMMAND, 1, do_ifup)
insmod,	COMMAND, 1, do_insmod)
import,	COMMAND, 1, do_import)
keycodes,	OPTION, 0, 0)
mkdir,	COMMAND, 1, do_mkdir)
mount,	COMMAND, 3, do_mount)
on,	SECTION, 0, 0)
oneshot,	OPTION, 0, 0)
onrestart,	OPTION, 0, 0)
restart,	COMMAND, 1, do_restart)
service,	SECTION, 0, 0)
setenv,	OPTION, 2, 0)
setkey,	COMMAND, 0, do_setkey)

```

setprop,      COMMAND, 2, do_setprop)
setrlimit,    COMMAND, 3, do_setrlimit)
socket,       OPTION, 0, 0)
start,        COMMAND, 1, do_start)
stop,         COMMAND, 1, do_stop)
trigger,      COMMAND, 1, do_trigger)
symlink,      COMMAND, 1, do_symlink)
sysclktz,     COMMAND, 1, do_sysclktz)
user,         OPTION, 0, 0)
write,        COMMAND, 2, do_write)
chown,        COMMAND, 2, do_chown)
chmod,        COMMAND, 2, do_chmod)
loglevel,     COMMAND, 1, do_loglevel)
device,       COMMAND, 4, do_device)

```

parse_new_section()中再分别对 service 或者 on 关键字开头的内容进行解析。

```

...
case K_service:
    state->context = parse_service(state, nargs, args);
    if (state->context) {
        state->parse_line = parse_line_service;
        return;
    }
    break;
case K_on:
    state->context = parse_action(state, nargs, args);
    if (state->context) {
        state->parse_line = parse_line_action;
        return;
    }
    break;
}
...

```

对 on 关键字开头的内容进行解析

```

static void *parse_action(struct parse_state *state, int nargs, char
**args)
{
    ...
    act = calloc(1, sizeof(*act));
    act->name = args[1];
    list_init(&act->commands);
    list_add_tail(&action_list, &act->alist);
}

```

```

    ...
}

```

对 service 关键字开头的内容进行解析

```

static void *parse_service(struct parse_state *state, int nargs, char
**args)

```

```

{
    struct service *svc;
    if (nargs < 3) {
        parse_error(state, "services must have a name and a program\n");
        return 0;
    }
    if (!valid_name(args[1])) {
        parse_error(state, "invalid service name '%s'\n", args[1]);
        return 0;
    }
    //如果服务已经存在 service_list 列表中将会被忽略
    svc = service_find_by_name(args[1]);
    if (svc) {
        parse_error(state, "ignored duplicate definition of service
'%s'\n", args[1]);
        return 0;
    }

```

```

    nargs -= 2;
    svc = calloc(1, sizeof(*svc) + sizeof(char*) * nargs);
    if (!svc) {
        parse_error(state, "out of memory\n");
        return 0;
    }
    svc->name = args[1];
    svc->classname = "default";
    memcpy(svc->args, args + 2, sizeof(char*) * nargs);
    svc->args[nargs] = 0;
    svc->nargs = nargs;
    svc->onrestart.name = "onrestart";
    list_init(&svc->onrestart.commands);
    //添加该服务到 service_list 列表
    list_add_tail(&service_list, &svc->slist);
    return svc;
}

```

服务的表现形式:

```

service <name> <pathname> [ <argument> ]*
<option>

```

<option>

...

申请一个 service 结构体, 然后挂接到 service_list 链表上, name 为服务的名称
pathname 为执行的命令 argument
为命令的参数。之后的 option 用来控制这个 service 结构体的属性, parse_line_service 会对 service 关键字后的内容进行解析并填充到 service 结构中, 当遇到下一个 service 或者 on 关键字的时候此 service 选项解析结束。

例如:

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote
--start-system-server
```

```
    socket zygote stream 666
```

```
    onrestart write /sys/android_power/request_state wake
```

服务名称为: zygote

启动该服务执行的命令: /system/bin/app_process

命令的参数: -Xzygote /system/bin --zygote

```
--start-system-server
```

socket zygote stream 666: 创建一个名为: /dev/socket/zygote 的 socket ,
类型为: stream

至此, 整个流程已经完成了, 以上仅是我个人的理解。分析的结束了, 但是我们的学习不会结束, 学习是一个无止境的过程, 我们对于其深入的东西还不怎么了解, 所以还需继续努力。