Android 启动流程（init.c 和 init.rc 分析）

---Written By Kevia Huang

PART 1 :对于 init.c 主要分析 main( )，知道大致流程

```
int main(int argc, char **argv)
{
    int device_fd = -1;
    int property_set_fd = -1;
    int signal_recv_fd = -1;
    int keychord_fd = -1;
    int fd_count;
    int s[2];
    int fd;
    struct sigaction act;
    char tmp[PROP_VALUE_MAX];
    struct pollfd ufds[4];
    char *tmpdev;
    char* debuggable;
    /*
     *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。
     */
    act.sa_handler = sigchld_handler;
    act.sa_flags = SA_NOCLDSTOP;
    act.sa_mask = 0;
    act.sa_restorer = NULL;
    sigaction(SIGCHLD, &act, 0);

    /* clear the umask */
    umask(0);

        /* Get the basic filesystem setup we need put
         * together in the initramdisk on / and then we'll
         * let the rc file figure out the rest.
         */
    //为 rootfs 建立必要的文件夹，并挂载适当的分区
    mkdir("/dev", 0755);
    mkdir("/proc", 0755);
    mkdir("/sys", 0755);

    mount("tmpfs", "/dev", "tmpfs", 0, "mode=0755");
```

```
mkdir("/dev/pts", 0755);
mkdir("/dev/socket", 0755);
mount("devpts", "/dev/pts", "devpts", 0, NULL);
mount("proc", "/proc", "proc", 0, NULL);
mount("sysfs", "/sys", "sysfs", 0, NULL);

        /* We must have some place other than / to create the
          * device nodes for kmsg and null, otherwise we won't
          * be able to remount / read-only later on.
          * Now that tmpfs is mounted on /dev, we can actually
          * talk to the outside world.
          */
  //创建/dev/null 节点,需要在后面的程序中看到打印信息的话，
  //需要屏蔽掉这个函数
open_devnull_stdio();

//初始化 log 系统
log_init();

INFO("reading config file\n");
//解析/init.rc，将所有服务和操作信息加入链表。
parse_config_file("/init.rc");

/* pull the kernel commandline and ramdisk properties file in */
//从/proc/cmdline 中提取信息内核启动参数,并保存到全局变量。
qemu_init();   //---------->仿真器么?
import_kernel_cmdline(0);//--------------->传递参数 0 表示没有工作在仿真器?


//先从上一步获得的全局变量中获取硬件信息和版本号，
//如果没有则从/proc/cpuinfo 中提取,并保存到全局变量。
//static char hardware[32];
get_hardware_name();
snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);
/*
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。
*/
parse_config_file(tmp);
//执行链表中带有“early-init”触发的的命令。
action_for_each_trigger("early-init", action_add_queue_tail);
drain_action_queue();
```

```
INFO("device init\n");
//这段代码是遍历为/sys，添加设备事件响应，创建设备节点。
device_fd = device_init();

/*

这段代码是进行属性初始化。每个属性都有一个名称和值，它们都是字符串格式。
属性被大量使用在 Android 系统中，用来记录系统设置或进程之间的信息交换。属
性是在整个系统中全局可见的。每个进程可以 get/set 属性。在系统初始化时，
Android 将分配一个共享内存区来存储的属性，这里主要是从/default.prop 属性文
件读取属性。这个有点像 Windows 下的注册表的作用。

*/
property_init();
//这段代码是从属性里获取调试标志，如果是可以调试，就打开组合按键输入驱动程序,
//初始化 keychord 监听。
// only listen for keychords if ro.debuggable is true
debuggable = property_get("ro.debuggable");
if (debuggable && !strcmp(debuggable, "1")) {
        keychord_fd = open_keychord();
}

//这段代码是判断是否有控制台，如果没有，就尝试是否是可以打缺省的控制台。

if (console[0]) {
        snprintf(tmp, sizeof(tmp), "/dev/%s", console);
        console_name = strdup(tmp);
}
//打开 console，如果 cmdline 中没有指定 console 则打开默认的/dev/console
fd = open(console_name, O_RDWR);
if (fd >= 0)
        have_console = 1;
close(fd); //--->为何又关闭掉呢?

//读取/initlogo.rle ,如果成功则在/dev/graphics/fb0 显示 logo,如果失败
//则将/dev/tty0 设为 TEXT 模式并打开/dev/tty0,输出文本 ANDROID 字样。
if( load_565rle_image(INIT_IMAGE_FILE) ) {
fd = open("/dev/tty0", O_WRONLY);
if (fd >= 0) {
        const char *msg;
                msg = "\n"
        "\n"
        "\n"
        "\n"
```

```
            "\n"
            "\n"
            "\n"    // console is 40 cols x 30 lines
            "\n"
            "\n"
            "\n"
            "\n"
            "\n"
            "\n"
            "\n"
            "                        A N D R O I D ";
            write(fd, msg, strlen(msg));
            close(fd);
        }
        }

//这段代码是用来判断是否使用模拟器运行，如果是，就加载内核命令行参数。
        if (qemu[0])
            import_kernel_cmdline(1);

//这段代码是根据内核命令行参数来设置工厂模式测试，比如在工厂生产手机过
//程里需要自动化演示功能，就可以根据这个标志来进行特别处理。
        if (!strcmp(bootmode,"factory"))
            property_set("ro.factorytest", "1");
        else if (!strcmp(bootmode,"factory2"))
            property_set("ro.factorytest", "2");
        else
            property_set("ro.factorytest", "0");

//这段代码是设置手机序列号到属性里保存，以便上层应用程序可以识别这台手机。
        property_set("ro.serialno", serialno[0] ? serialno : "");
//这段代码是保存启动模式到属性里。
        property_set("ro.bootmode", bootmode[0] ? bootmode : "unknown");
//这段代码是保存手机基带频率到属性里。
        property_set("ro.baseband", baseband[0] ? baseband : "unknown");
//这段代码是保存手机硬件载波的方式到属性里。
        property_set("ro.carrier", carrier[0] ? carrier : "unknown");
//保存引导程序的版本号到属性里，以便系统知道引导程序有什么特性。
        property_set("ro.bootloader", bootloader[0] ? bootloader : "unknown");
//这里是保存硬件信息到属性里，其实就是获取 CPU 的信息。
        property_set("ro.hardware", hardware);
//这里是保存硬件修订的版本号到属性里，这样可以方便应用程序区分不同的硬件版本。

        snprintf(tmp, PROP_VALUE_MAX, "%d", revision);
```

```
property_set("ro.revision", tmp);

    /* execute all the boot actions to get us started */
    //执行所有触发标志为 init 的 action
action_for_each_trigger("init", action_add_queue_tail);
drain_action_queue();

    /* read any property files on system or data and
     * fire up the property service.    This must happen
     * after the ro.foo properties are set above so
     * that /data/local.prop cannot interfere with them.
     */

/*
开始 property 服务，读取一些 property 文件，这一动作必须在前面那些 ro.foo 设置后做，
以便/data/local.prop 不能干预到他们
        - /system/build.prop
        - /system/default.prop
        - /data/local.prop
        -  在读取认识的 property 后读取 presistent propertie，在/data/property 中

这段代码是加载 system 和 data 目录下的属性，并启动属性监听服务。

*/
property_set_fd = start_property_service();
//为 sighandler 创建信号机制
/*

这段代码是创建一个全双工的通讯机制的两个 SOCKET，信号可以在 signal_fd
和 signal_recv_fd 双向通讯，从而建立起沟通的管道。其实这个信号管理，就
是用来让 init 进程与它的子进程进行沟通的，子进程从 signal_fd 写入信息，init
进程从 signal_recv_fd 收到信息，然后再做处理。

*/
/* create a signalling mechanism for the sigchld handler */
if (socketpair(AF_UNIX, SOCK_STREAM, 0, s) == 0) {
    signal_fd = s[0];
    signal_recv_fd = s[1];
    fcntl(s[0], F_SETFD, FD_CLOEXEC);
    fcntl(s[0], F_SETFL, O_NONBLOCK);
    fcntl(s[1], F_SETFD, FD_CLOEXEC);
    fcntl(s[1], F_SETFL, O_NONBLOCK);
}
```

```
    /* make sure we actually have all the pieces we need */

    /*
    这段代码是判断关键的几个组件是否成功初始化，主要就是设备文件系统是否成
    功初始化，属性服务是否成功初始化，信号通讯机制是否成功初始化。
    */
    if ((device_fd < 0) ||
        (property_set_fd < 0) ||
        (signal_recv_fd < 0)) {
        ERROR("init startup failure\n");
        return 1;
    }

    /* execute all the boot actions to get us started */
    //执行所有触发标志为 early-boot 的 action
    action_for_each_trigger("early-boot", action_add_queue_tail);
    //执行所有触发标志为 boot 的 action
    action_for_each_trigger("boot", action_add_queue_tail);
    drain_action_queue();

        /* run all property triggers based on current state of the properties */
    //基于当前 property 状态，执行所有触发标识为 property 的 action
    //这段代码是根据当前属性，运行属性命令。
    queue_all_property_triggers();
    drain_action_queue();

        /* enable property triggers */
        //    标明属性触发器已经初始化完成。
    property_triggers_enabled = 1;

    /*
        这段代码是保存三个重要的服务 socket，以便后面轮询使用。
    */
    ufds[0].fd = device_fd;
    ufds[0].events = POLLIN;
    ufds[1].fd = property_set_fd;
    ufds[1].events = POLLIN;
    ufds[2].fd = signal_recv_fd;
    ufds[2].events = POLLIN;
    fd_count = 3;

//这段代码是判断是否处理组合键轮询。
    if (keychord_fd > 0) {
        ufds[3].fd = keychord_fd;
```

```
            ufds[3].events = POLLIN;
            fd_count++;
        } else {
            ufds[3].events = 0;
            ufds[3].revents = 0;
        }
//如果支持 BOOTCHART，则初始化 BOOTCHART
#if BOOTCHART
/*

这段代码是初始化 linux 程序启动速度的性能分析工具，这个工具有一个好处，就是图形化显示每个进程启动顺序和占用时间，如果想优化系统的启动速度，记得启用这个工具。

*/
        bootchart_count = bootchart_init();
        if (bootchart_count < 0) {
            ERROR("bootcharting init failure\n");
        } else if (bootchart_count > 0) {
            NOTICE("bootcharting started (period=%d ms)\n",
bootchart_count*BOOTCHART_POLLING_MS);
        } else {
            NOTICE("bootcharting ignored\n");
        }
#endif
/*
进入主进程循环:
        - 重置轮询事件的接受状态，revents 为 0
        - 查询 action 队列，并执行。
        - 重启需要重启的服务
        - 轮询注册的事件
            - 如果 signal_recv_fd 的 revents 为 POLLIN，则得到一个信号，获取并处理
            - 如果 device_fd 的 revents 为 POLLIN,调用 handle_device_fd
            - 如果 property_fd 的 revents 为 POLLIN,调用 handle_property_set_fd
            - 如果 keychord_fd 的 revents 为 POLLIN,调用 handle_keychord
这段代码是进入死循环处理，以便这个 init 进程变成一个服务。
*/
        for(;;) {
            int nr, i, timeout = -1;
            //          清空每个 socket 的事件计数。
            for (i = 0; i < fd_count; i++)
                ufds[i].revents = 0;
            //          这段代码是执行队列里的命令。
            drain_action_queue();
            //  这句代码是用来判断那些服务需要重新启动。
```

```
            restart_processes();
            //          这段代码是用来判断哪些进程启动超时。
            if (process_needs_restart) {
                timeout = (process_needs_restart - gettime()) * 1000;
                if (timeout < 0)
                    timeout = 0;
            }
#if BOOTCHART
//这段代码是用来计算运行性能。
            if (bootchart_count > 0) {
                if (timeout < 0 || timeout > BOOTCHART_POLLING_MS)
                    timeout = BOOTCHART_POLLING_MS;
                if (bootchart_step() < 0 || --bootchart_count == 0) {
                    bootchart_finish();
                    bootchart_count = 0;
                }
            }
#endif
//          这段代码用来轮询几个 socket 是否有事件处理。
            nr = poll(ufds, fd_count, timeout);
            if (nr <= 0)
                continue;
/*
这段代码是用来处理子进程的通讯，并且能删除任何已经退出或者杀
死进程，这样做可以保持系统更加健壮性，增强容错能力。
*/
            if (ufds[2].revents == POLLIN) {
                /* we got a SIGCHLD - reap and restart as needed */
                read(signal_recv_fd, tmp, sizeof(tmp));
                while (!wait_for_one_process(0))
                    ;
                continue;
            }
//          这段代码是处理设备事件。
            if (ufds[0].revents == POLLIN)
                handle_device_fd(device_fd);
//          这段代码是处理属性服务事件。
            if (ufds[1].revents == POLLIN)
                handle_property_set_fd(property_set_fd);
//          这段代码是处理调试模式下的组合按键。
            if (ufds[3].revents == POLLIN)
                handle_keychord(keychord_fd);
        }
```

```
    return 0;
}
```

PART 2:下面是配置脚本解析相关代码的大致分析：

```
---------------->
int parse_config_file(const char *fn)
{
    char *data;
    data = read_file(fn, 0);
    if (!data) return -1;
    //解析配置文件，将所有的 service 和 action 段解析出来，为他们分配数据结构
    //并将他们分别加入相应的服务链表和动作链表。这个过程中会有很多的判断，包括
    //init.rc 配置文件的语法中的:若有命名重复的段，后面重复的段将被忽略
    parse_config(fn, data);
    DUMP();   //什么也没干
    return 0;
}
--------------->
static void parse_config(const char *fn, char *s)
{
    struct parse_state state;
    //#define SVC_MAXARGS 64
    char *args[SVC_MAXARGS];
    int nargs;

    nargs = 0;
    state.filename = fn;
    state.line = 1;
    state.ptr = s;
    state.nexttoken = 0;
    state.parse_line = parse_line_no_op;
    for (;;) {
        //next_token 获取配置脚本中的下一个符号，并做相应的判断，返回判断的结果
        switch (next_token(&state)) {
        case T_EOF:
            state.parse_line(&state, 0, 0);
            return;
        case T_NEWLINE:
            //nargs 何时会被赋予非 0 值呢?---->看下一个 case
            if (nargs) {
                int kw = lookup_keyword(args[0]); //查找关键字，返回关键字
                if (kw_is(kw, SECTION)) {
                    state.parse_line(&state, 0, 0);
```

```
                    //解析这个新的段:
                    parse_new_section(&state, kw, nargs, args);
                } else {
                    state.parse_line(&state, nargs, args);
                }
                nargs = 0;
            }
            break;
        case T_TEXT:
            if (nargs < SVC_MAXARGS) {
                args[nargs++] = state.text;
            }
            break;
        }
    }
}

---------->
//解析一个新的段:
void parse_new_section(struct parse_state *state, int kw,
                       int nargs, char **args)
{
    //这些调试信息都会被打印到/dev/null 中去，若想看到调试信息，
    //可使用 logcat 或者在 init.c  的 main 函数中注释掉 open_devnull_stdio();
    printf("[ %s %s ]\n", args[0],
           nargs > 1 ? args[1] : "");
    switch(kw) {
    case K_service:
        //解析服务段:解析服务段会判断这个是否有已经同名的服务段，若有则忽略
        //当前正解析的段；但是为何解析动作段时没有类似的判断呢?
        state->context = parse_service(state, nargs, args);
        if (state->context) {
                                    //解析行服务?
            state->parse_line = parse_line_service;
            return;
        }
        break;
    case K_on:
                                //解析动作段:
        state->context = parse_action(state, nargs, args);
        if (state->context) {
                                    //解析行动作?-->将命令加入命令列表
            state->parse_line = parse_line_action;
            return;
```

```
            }
            break;
        }
        state->parse_line = parse_line_no_op;
}
---------------->
//解析服务:
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;
        }
        //通过名字找到服务，若找到，返回相应 service 结构体首地址
        svc = service_find_by_name(args[1]);
        if (svc) {//这个到底啥意思?找到了还报错?---->这个意思正好符号 init.rc 语法分析，就是
                  //若一个段名重复声明，后面的重复段将被忽略。
            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";
        //初始化服务的指令链表?
        list_init(&svc->onrestart.commands);
        //添加到 service_list 链表尾部,是个环形链表
        list_add_tail(&service_list, &svc->slist);
        return svc;
}
```

```
---------->
struct service *service_find_by_name(const char *name)
{
    struct listnode *node;
    struct service *svc;
    //service_list 何时创建，添加?是个环形链表?
    list_for_each(node, &service_list) {
        svc = node_to_item(node, struct service, slist);//获取 service 结构体首地址
        if (!strcmp(svc->name, name)) {//采用名字匹配
            return svc;
        }
    }
    return 0;
}
```

PART 3:　init.rc 语法分析

Android 的根目录下有一系列非常重要的配置文件，即：init.rc init.xxxxx.rc

Android 中解析这些配置文件的代码在：system\core\init 目录下。

核心文件是 init.c，相关的文法说明请参见：readme.txt

本节简单介绍一下 init.rc 文法。

Readme.txt 原文及注解如下：

Android Init Language

The Android Init Language consists of four broad classes of statements,
which are Actions, Commands, Services, and Options.

All of these are line-oriented, consisting of tokens separated by
whitespace.　The c-style backslash escapes may be used to insert
whitespace into a token.　Double quotes may also be used to prevent
whitespace from breaking text into multiple tokens.　The backslash,
when it is the last character on a line, may be used for line-folding.

Lines which start with a # (leading whitespace allowed) are comments.

Actions and Services implicitly declare a new section.　All commands
or options belong to the section most recently declared.　Commands
or options before the first section are ignored.

Actions and Services have unique names. If a second Action or Service
is declared with the same name as an existing one, it is ignored as
an error. (??? should we override instead)

该语言的语法包括下列约定：

•所有类型的语句都是基于行（line-oriented）的， 一个语句包含若干个 tokens，token 之间通过空格字符分隔. 如果一个 token 中需要包含空格字符，则需要通过 C 语言风格的反斜线（'\'）来转义，或者使用双引号把整个 token 引起来。反斜线还可以出现在一行的末尾，表示下一行的内容仍然属于当前语句。
•以'#'开始的行是注释行。
•动作（Actions）和服务（Services）语句隐含表示一个新的段落（section）的开始。 所有的指令（commands）和选项（options）归属于上方最近的一个段落。在第一个段落之前的指令（commands）和选项（options）是无效的。
•动作（Actions）和服务（Services）拥有唯一性的名字。如果出现重名，那么后出现的定义将被作为错误忽略掉。

Actions

Actions are named sequences of commands. Actions have a trigger which
is used to determine when the action should occur. When an event
occurs which matches an action's trigger, that action is added to
the tail of a to-be-executed queue (unless it is already on the
queue).

Each action in the queue is dequeued in sequence and each command in
that action is executed in sequence. Init handles other activities
(device creation/destruction, property setting, process restarting)
"between" the execution of the commands in activities.

Actions take the form of:

on <trigger>
    <command>
    <command>
    <command>

Services

Services are programs which init launches and (optionally) restarts
when they exit. Services take the form of:

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

至此可以看出，init.rc 实际上是由两种类型的段落组成：

1。on <trigger>开头的一系列命令

2。Service 开头的服务启动（可以携带参数和选项）

Options

Options are modifiers to services. They affect how and when init
runs the service.

critical
    This is a device-critical service. If it exits more than four times in
    four minutes, the device will reboot into recovery mode.

4分钟内退出4次，重起进入 recovery 模式

disabled
    This service will not automatically start with its class.
    It must be explicitly started by name.

setenv <name> <value>
    Set the environment variable <name> to <value> in the launched process.

socket <name> <type> <perm> [<user> [<group>]]
    Create a unix domain socket named /dev/socket/<name> and pass
    its fd to the launched process. <type> must be "dgram" or "stream".
    User and group default to 0.

创建一个 unix 域的 socket,名为/dev/socket/<name>，并把它的 fd 传给服务进程。

类型必须是'dgram' or 'stream'

user <username>
    Change to username before exec'ing this service.
    Currently defaults to root. (??? probably should default to nobody)
    Currently, if your process requires linux capabilities then you cannot use
    this command. You must instead request the capabilities in-process while
    still root, and then drop to your desired uid.

group <groupname> [<groupname>]*

Change to groupname before exec'ing this service. Additional groupnames beyond the (required) first one are used to set the supplemental groups of the process (via setgroups()). Currently defaults to root. (??? probably should default to nobody)

oneshot

Do not restart the service when it exits.

推出后不重启，默认 service 退出后自动重启

class <name>

Specify a class name for the service. All services in a named class may be started or stopped together. A service is in the class "default" if one is not specified via the class option.

为 service 指定一个 class。同一个 class 下的 service 一起启动或停止。

未被指定过的 service 默认在 class 'default'下

onrestart

Execute a Command (see below) when service restarts.

服务重启的时候，执行命令（重启前？后？待调查。应该是后。）

Triggers

Triggers are strings which can be used to match certain kinds of events and used to cause an action to occur.

boot

This is the first trigger that will occur when init starts (after /init.conf is loaded)

在 init.rc 中除了 boot 事件外常用的事件及其执行顺序如下：

early-init，init，early-fs，fs，post-fs，early-boot，boot

<name>=<value>

Triggers of this form occur when the property <name> is set to the specific value <value>.

device-added-<path>

device-removed-<path>

Triggers of these forms occur when a device node is added

or removed.

某个设备节点 added or removed 时执行

service-exited-<name>
Triggers of this form occur when the specified service exits.

Commands
---

exec <path> [<argument>]*
Fork and execute a program (<path>). This will block until the program completes execution. It is best to avoid exec as unlike the builtin commands, it runs the risk of getting init "stuck". (??? maybe there should be a timeout?)

执行<path>指定的 Program。它会阻塞当前的 init。

export <name> <value>
Set the environment variable <name> equal to <value> in the global environment (which will be inherited by all processes started after this command is executed)

ifup <interface>
Bring the network interface <interface> online.

import <filename>
Parse an init config file, extending the current configuration.

hostname <name>
Set the host name.

chdir <directory>
Change working directory.

chmod <octal-mode> <path>
Change file access permissions.

chown <owner> <group> <path>
Change file owner and group.

chroot <directory>
Change process root directory.

class_start <serviceclass>

Start all services of the specified class if they are
not already running.

启动某个 class 下的服务。（如果还没有启动的话）

class_stop <serviceclass>
Stop all services of the specified class if they are
currently running.

domainname <name>
Set the domain name.

insmod <path>
Install the module at <path>

mkdir <path> [mode] [owner] [group]
Create a directory at <path>, optionally with the given mode, owner, and
group. If not provided, the directory is created with permissions 755 and
owned by the root user and root group.

mount <type> <device> <dir> [<mountoption>]*
Attempt to mount the named device at the directory <dir>
<device> may be of the form mtd@name to specify a mtd block
device by name.
<mountoption>s include "ro", "rw", "remount", "noatime", ...

setkey
TBD

setprop <name> <value>
Set system property <name> to <value>.

setrlimit <resource> <cur> <max>
Set the rlimit for a resource.

start <service>
Start a service running if it is not already running.

启动一个服务

stop <service>
Stop a service from running if it is currently running.

symlink <target> <path>
Create a symbolic link at <path> with the value <target>

sysclktz <mins_west_of_gmt>
    Set the system clock base (0 if system clock ticks in GMT)

设置时区

trigger <event>
    Trigger an event. Used to queue an action from another
    action.

write <path> <string> [<string>]*
    Open the file at <path> and write one or more strings
    to it with write(2)

Properties

Init updates some system properties to provide some insight into
what it's doing:

init.action
    Equal to the name of the action currently being executed or "" if none

init.command
    Equal to the command being executed or "" if none.

init.svc.<name>
    State of a named service ("stopped", "running", "restarting")

Example init.conf
----

not complete -- just providing some examples of usage
#
on boot
    export PATH /sbin:/system/sbin:/system/bin
    export LD_LIBRARY_PATH /system/lib

    mkdir /dev
    mkdir /proc
    mkdir /sys

    mount tmpfs tmpfs /dev
    mkdir /dev/pts
    mkdir /dev/socket

```
    mount devpts devpts /dev/pts
    mount proc proc /proc
    mount sysfs sysfs /sys

    write /proc/cpu/alignment 4

    ifup lo

    hostname localhost
    domainname localhost

    mount yaffs2 mtd@system /system
    mount yaffs2 mtd@userdata /data

    import /system/etc/init.conf

    class_start default

service adbd /sbin/adbd
    user adb
    group adb

service usbd /system/bin/usbd -r
    user usbd
    group usbd
    socket usbd 666

service zygote /system/bin/app_process -Xzygote /system/bin --zygote
    socket zygote 666

service runtime /system/bin/runtime
    user system
    group system

on device-added-/dev/compass
    start akmd

on device-removed-/dev/compass
    stop akmd

service akmd /sbin/akmd
    disabled
    user akmd
    group akmd
```

Debugging notes

By default, programs executed by init will drop stdout and stderr into
/dev/null. To help with debugging, you can execute your program via the
Andoird program logwrapper. This will redirect stdout/stderr into the
Android logging system (accessed via logcat).

For example
service akmd /system/bin/logwrapper /sbin/akmd

调试的时候使用使用上述命令，可以启动一个服务并且看到这个服务的输出。

原因是 android init 将 stdout 和 stderr 转发到了/dev/null

遗留问题：

1。option 中的 socket 创建后怎么传递到 service 中的？

2。如果一个 Service 在 init.rc 中标注了 disable,怎么启动？在 init.rc 中使用 start <service name>?可以在 C 代码中启动么？