**[Android的init过程详解（一）](http://www.cnblogs.com/nokiaguy/archive/2013/04/14/3020774.html)**

本文使用的软件版本

Android：4.2.2

Linux内核：3.1.10

     本文及后续几篇文章将对Android的初始化（init）过程进行详细地、剥丝抽茧式地分析，并且在其中穿插了大量的知识，希望对读者了解Android的启动过程又所帮助。本章主要介绍了与硬件相关初始化文件名的确定以及属性服务的原理和实现。

     Android本质上就是一个基于Linux内核的操作系统。与Ubuntu Linux、Fedora Linux类似。只是Android在应用层专门为移动设备添加了一些特有的支持。既然Android是Linux内核的系统，那么基本的启动过程也应符合Linux的规则。如果研究过其他Linux系统应该了解，一个完整的Linux系统首先会将一个Linux内核装载到内存，也就是编译Linux内核源代码生成的bzImage文件，对于为Android优化的Linux内核源代码会生成zImage文件。该文件就是Linux内核的二进制版本。由于zImage在内核空间运行，而我们平常使用的软件都是在应用空间运行（关于内核空间和应用空间的详细描述，可以参考[《Android深度探索（卷1）：HAL与驱动开发》](http://product.dangdang.com/main/product.aspx?product_id=23043311&_ddclickunion=P-263982|ad_type=0|sys_id=1#dd_refer=http%3A%2F%2Fblog.csdn.net%2Fnokiaguy)一书的内容，在后续的各卷中将会对Android的整体体系进行全方位的剖析）。内核空间和应用空间是不能直接通过内存地址级别访问的，所以就需要建立某种通讯机制。

     目前Linux有很多通讯机制可以在用户空间和内核空间之间交互，例如设备驱动文件（位于/dev目录中）、内存文件（/proc、/sys目录等）。了解Linux的同学都应该知道Linux的重要特征之一就是一切都是以文件的形式存在的，例如，一个设备通常与一个或多个设备文件对应。这些与内核空间交互的文件都在用户空间，所以在Linux内核装载完，需要首先建立这些文件所在的目录。而完成这些工作的程序就是本文要介绍的init。Init是一个命令行程序。其主要工作之一就是建立这些与内核空间交互的文件所在的目录。当Linux内核加载完后，要做的第一件事就是调用init程序，也就是说，init是用户空间执行的第一个程序。

在分析init的核心代码之前，还需要初步了解init除了建立一些目录外，还做了如下的工作

1. 初始化属性

2. 处理配置文件的命令（主要是init.rc文件），包括处理各种Action。

3. 性能分析（使用bootchart工具）。

4. 无限循环执行command（启动其他的进程）。

     尽管init完成的工作不算很多，不过代码还是非常复杂的。Init程序并不是由一个源代码文件组成的，而是由一组源代码文件的目标文件链接而成的。这些文件位于如下的目录。

<Android源代码本目录>/system/core/init

     其中init.c是init的主文件，现在打开该文件，看看其中的内容。由于init是命令行程序，所以分析init.c首先应从main函数开始，现在好到main函数，代码如下：

[复制代码](javascript:void(0);)

int main(int argc, char \*\*argv)

{

int fd\_count = 0;

struct pollfd ufds[4];

char \*tmpdev;

char\* debuggable;

char tmp[32];

int property\_set\_fd\_init = 0;

int signal\_fd\_init = 0;

int keychord\_fd\_init = 0;

bool is\_charger = false;

if (!strcmp(basename(argv[0]), "ueventd"))

return ueventd\_main(argc, argv);

if (!strcmp(basename(argv[0]), "watchdogd"))

return watchdogd\_main(argc, argv);

/\* clear the umask \*/

umask(0);

// 下面的代码开始建立各种用户空间的目录，如/dev、/proc、/sys等

mkdir("/dev", 0755);

mkdir("/proc", 0755);

mkdir("/sys", 0755);

mount("tmpfs", "/dev", "tmpfs", MS\_NOSUID, "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);

/\* 检测/dev/.booting文件是否可读写和创建\*/

close(open("/dev/.booting", O\_WRONLY | O\_CREAT, 0000));

open\_devnull\_stdio();

klog\_init();

// 初始化属性

property\_init();

get\_hardware\_name(hardware, &revision);

// 处理内核命令行

process\_kernel\_cmdline();

… …

is\_charger = !strcmp(bootmode, "charger");

INFO("property init\n");

if (!is\_charger)

property\_load\_boot\_defaults();

INFO("reading config file\n");

// 分析/init.rc文件的内容

init\_parse\_config\_file("/init.rc");

… …// 执行初始化文件中的动作

action\_for\_each\_trigger("init", action\_add\_queue\_tail);

// 在charger模式下略过mount文件系统的工作

if (!is\_charger) {

action\_for\_each\_trigger("early-fs", action\_add\_queue\_tail);

action\_for\_each\_trigger("fs", action\_add\_queue\_tail);

action\_for\_each\_trigger("post-fs", action\_add\_queue\_tail);

action\_for\_each\_trigger("post-fs-data", action\_add\_queue\_tail);

}

queue\_builtin\_action(property\_service\_init\_action, "property\_service\_init");

queue\_builtin\_action(signal\_init\_action, "signal\_init");

queue\_builtin\_action(check\_startup\_action, "check\_startup");

if (is\_charger) {

action\_for\_each\_trigger("charger", action\_add\_queue\_tail);

} else {

action\_for\_each\_trigger("early-boot", action\_add\_queue\_tail);

action\_for\_each\_trigger("boot", action\_add\_queue\_tail);

}

/\* run all property triggers based on current state of the properties \*/

queue\_builtin\_action(queue\_property\_triggers\_action, "queue\_property\_triggers");

#if BOOTCHART

queue\_builtin\_action(bootchart\_init\_action, "bootchart\_init");

#endif

// 进入无限循环，建立init的子进程（init是所有进程的父进程）

for(;;) {

int nr, i, timeout = -1;

// 执行命令（子进程对应的命令）

execute\_one\_command();

restart\_processes();

if (!property\_set\_fd\_init && get\_property\_set\_fd() > 0) {

ufds[fd\_count].fd = get\_property\_set\_fd();

ufds[fd\_count].events = POLLIN;

ufds[fd\_count].revents = 0;

fd\_count++;

property\_set\_fd\_init = 1;

}

if (!signal\_fd\_init && get\_signal\_fd() > 0) {

ufds[fd\_count].fd = get\_signal\_fd();

ufds[fd\_count].events = POLLIN;

ufds[fd\_count].revents = 0;

fd\_count++;

signal\_fd\_init = 1;

}

if (!keychord\_fd\_init && get\_keychord\_fd() > 0) {

ufds[fd\_count].fd = get\_keychord\_fd();

ufds[fd\_count].events = POLLIN;

ufds[fd\_count].revents = 0;

fd\_count++;

keychord\_fd\_init = 1;

}

if (process\_needs\_restart) {

timeout = (process\_needs\_restart - gettime()) \* 1000;

if (timeout < 0)

timeout = 0;

}

if (!action\_queue\_empty() || cur\_action)

timeout = 0;

// bootchart是一个性能统计工具，用于搜集硬件和系统的信息，并将其写入磁盘，以便其

// 他程序使用

#if BOOTCHART

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

// 等待下一个命令的提交

nr = poll(ufds, fd\_count, timeout);

if (nr <= 0)

continue;

for (i = 0; i < fd\_count; i++) {

if (ufds[i].revents == POLLIN) {

if (ufds[i].fd == get\_property\_set\_fd())

handle\_property\_set\_fd();

else if (ufds[i].fd == get\_keychord\_fd())

handle\_keychord();

else if (ufds[i].fd == get\_signal\_fd())

handle\_signal();

}

}

}

return 0;

}

[复制代码](javascript:void(0);)

     我们可以看到main函数是非常复杂的，不过我们也不需要每条语句都弄得非常清楚（因为这样弄是非常困难的），通常只需要了解init的主线即可。其实从init的main函数可以看出。Init实际上就分为如下两部分。

1.  初始化（包括建立/dev、/proc等目录、初始化属性、执行init.rc等初始化文件中的action等）。

2.  使用for循环无限循环建立子进程。

     第一项工作很好理解。而第二项工作是init中的核心。在Linux系统中init是一切应用空间进程的父进程。所以我们平常在Linux终端执行的命令，并建立进程。实际上都是在这个无限的for循环中完成的。也就是说，在Linux终端执行ps –e 命令后，看到的所有除了init外的其他进程，都是由init负责创建的。而且init也会常驻内容。当然，如果init挂了，Linux系统基本上就崩溃了。

    由于init比较复杂，所以本文只分析其中的一部分，在后续文章中将详细分析init的各个核心组成部分。

      对于main函数最开始完成的建立目录的工作比较简单，这部分也没什么可以分析的。就是调用了一些普通的API（mkdir）建立一些目录。现在说一些题外话，由于Android的底层源代码（包括init）实际上是属于Linux应用编程领域，所以要想充分理解Android源代码，除了Linux的基本结构要了解外，Linux应用层的API需要熟悉。为了满足这些读者的需要，后续我会写一些关于Linux应用编程的文章。Ok，现在言归正传，接下来分析一个比较重要的部分：配置文件的解析。

      这里的配置文件主要指init.rc。读者可以进到Android的shell，会看到根目录有一个init.rc文件。该文件是只读的，即使有了root权限，可以修改该文件也没有。因为我们在根目录看到的文件只是内存文件的镜像。也就是说，android启动后，会将init.rc文件装载到内存。而修改init.rc文件的内容实际上只是修改内存中的init.rc文件的内容。一旦重启android，init.rc文件的内容又会恢复到最初的装载。想彻底修改init.rc文件内容的唯一方式是修改Android的ROM中的内核镜像（boot.img）。其实boot.img名曰内核镜像，不过该文件除了包含完整的Linux内核文件（zImage）外，还包括另外一个镜像文件（ramdisk.img）。ramdisk.img就包含了init.rc文件和init命令。所以只有修改ramdisk.img文件中的init.rc文件，并且重新打包boot.img文件，并刷机，才能彻底修改init.rc文件。如果读者有Android源代码，编译后，就会看到out目录中的相关子目录会生成一个root目录，该目录实际上就是ramdisk.img解压后的内容。会看到有init命令和init.rc文件。在后续的文章中将会讨论具体如何修改init.rc文件，如何刷机。不过这些内容与本文关系不大，所以不做详细的讨论。

现在回到main函数，在创建完目录后，会看到执行了如下3个函数。

    property\_init();  
    get\_hardware\_name(hardware, &revision);  
    process\_kernel\_cmdline();

     其中property\_init主要是为属性分配一些存储空间，该函数并不是核心。不过当我们查看init.rc文件时会发现该文件开始部分用一些import语句导入了其他的配置文件，例如，/init.usb.rc。大多数配置文件都直接使用了确定的文件名，只有如下的代码使用了一个变量（${ro.hardware}）执行了配置文件名的一部分。那么这个变量值是从哪获得的呢？

import /init.${ro.hardware}.rc

     首先要了解init.${ro.hardware}.rc配置文件的内容通常与当前的硬件有关。现在我们先来关注get\_hardware\_name函数，代码如下：

[复制代码](javascript:void(0);)

void get\_hardware\_name(char \*hardware, unsigned int \*revision)

{

char data[1024];

int fd, n;

char \*x, \*hw, \*rev;

/\* 如果hardware已经有值了，说明hardware通过内核命令行提供，直接返回 \*/

if (hardware[0])

return;

// 打开/proc/cpuinfo文件

fd = open("/proc/cpuinfo", O\_RDONLY);

if (fd < 0) return;

// 读取/proc/cpuinfo文件的内容

n = read(fd, data, 1023);

close(fd);

if (n < 0) return;

data[n] = 0;

// 从/proc/cpuinfo文件中获取Hardware字段的值

hw = strstr(data, "\nHardware");

rev = strstr(data, "\nRevision");

// 成功获取Hardware字段的值

if (hw) {

x = strstr(hw, ": ");

if (x) {

x += 2;

n = 0;

while (\*x && \*x != '\n') {

if (!isspace(\*x))

// 将Hardware字段的值都转换为小写，并更新hardware参数的值

// hardware也就是在init.c文件中定义的hardware数组

hardware[n++] = tolower(\*x);

x++;

if (n == 31) break;

}

hardware[n] = 0;

}

}

if (rev) {

x = strstr(rev, ": ");

if (x) {

\*revision = strtoul(x + 2, 0, 16);

}

}

}

[复制代码](javascript:void(0);)

        从get\_hardware\_name方法的代码可以得知，该方法主要用于确定hardware和revision的变量的值。Revision这里先不讨论，只要研究hardware。获取hardware的来源是从Linux内核命令行或/proc/cpuinfo文件中的内容。Linux内核命令行暂且先不讨论（因为很少传递该值），先看看/proc/cpuinfo，该文件是虚拟文件（内存文件），执行cat /proc/cpuinfo命令会看到该文件中的内容，如图1所示。在白框中就是Hardware字段的值。由于该设备是Nexus 7，所以值为grouper。如果程序就到此位置，那么与硬件有关的配置文件名是init.grouper.rc。有Nexus 7的读者会看到在根目录下确实有一个init.grouper.rc文件。说明Nexus 7的原生ROM并没有在其他的地方设置配置文件名，所以配置文件名就是从/proc/cpuinfo文件的Hardware字段中取的值。

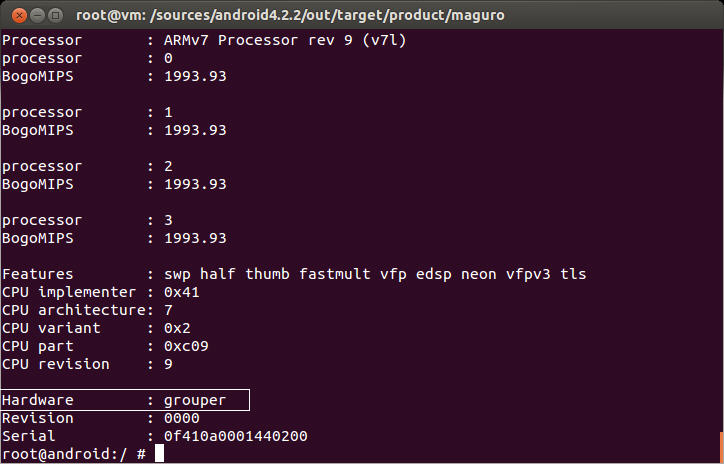


                                                  图1

现在来看在get\_hardware\_name函数后面调用的process\_kernel\_cmdline函数，代码如下：

[复制代码](javascript:void(0);)

static void process\_kernel\_cmdline(void)

{

/\* don't expose the raw commandline to nonpriv processes \*/

chmod("/proc/cmdline", 0440);

// 导入内核命令行参数

import\_kernel\_cmdline(0, import\_kernel\_nv);

if (qemu[0])

import\_kernel\_cmdline(1, import\_kernel\_nv);

// 用属性值设置内核变量

export\_kernel\_boot\_props();

}

[复制代码](javascript:void(0);)

       在process\_kernel\_cmdline函数中除了使用import\_kernel\_cmdline函数导入内核变量外，主要的功能就是调用export\_kernel\_boot\_props函数通过属性设置内核变量，例如，通过ro.boot.hardware属性设置hardware变量，也就是说可以通过ro.boot.hardware属性值可以修改get\_hardware\_name函数中从/proc/cpuinfo文件中得到的hardware字段值。下面看一下export\_kernel\_boot\_props函数的代码。

[复制代码](javascript:void(0);)

static void export\_kernel\_boot\_props(void)

{

char tmp[PROP\_VALUE\_MAX];

const char \*pval;

unsigned i;

struct {

const char \*src\_prop;

const char \*dest\_prop;

const char \*def\_val;

} prop\_map[] = {

{ "ro.boot.serialno", "ro.serialno", "", },

{ "ro.boot.mode", "ro.bootmode", "unknown", },

{ "ro.boot.baseband", "ro.baseband", "unknown", },

{ "ro.boot.bootloader", "ro.bootloader", "unknown", },

};

// 通过内核的属性设置应用层配置文件的属性

for (i = 0; i < ARRAY\_SIZE(prop\_map); i++) {

pval = property\_get(prop\_map[i].src\_prop);

property\_set(prop\_map[i].dest\_prop, pval ?: prop\_map[i].def\_val);

}

// 根据ro.boot.console属性的值设置console变量

pval = property\_get("ro.boot.console");

if (pval)

strlcpy(console, pval, sizeof(console));

/\* save a copy for init's usage during boot \*/

strlcpy(bootmode, property\_get("ro.bootmode"), sizeof(bootmode));

/\* if this was given on kernel command line, override what we read

\* before (e.g. from /proc/cpuinfo), if anything \*/

// 获取ro.boot.hardware属性的值

pval = property\_get("ro.boot.hardware");

if (pval)  
 // 这里通过ro.boot.hardware属性再次改变hardware变量的值

strlcpy(hardware, pval, sizeof(hardware));

// 利用hardware变量的值设置设置ro.hardware属性  
 // 这个属性就是前面提到的设置初始化文件名的属性，实际上是通过hardware变量设置的  
 property\_set("ro.hardware", hardware);

snprintf(tmp, PROP\_VALUE\_MAX, "%d", revision);

property\_set("ro.revision", tmp);

/\* TODO: these are obsolete. We should delete them \*/

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");

}

[复制代码](javascript:void(0);)

      从export\_kernel\_boot\_props函数的代码可以看出，该函数实际上就是来回设置一些属性值，并且利用某些属性值修改console、hardware等变量。其中hardware变量（就是一个长度为32的字符数组）在get\_hardware\_name函数中已经从/proc/cpuinfo文件中获得过一次值了，在export\_kernel\_boot\_props函数中又通过ro.boot.hardware属性设置了一次值，不过在Nexus 7中并没有设置该属性，所以hardware的值仍为grouper。最后用hardware变量设置ro.hardware属性，所以最后的初始化文件名为init.grouper.rc。

      这里还有一个问题，前面多次提到属性或属性文件，那么这些属性文件指的是什么呢？是init.rc？当然不是。实际上这些属性文件是一些列位于不同目录，系统依次读取的配置文件。

**属性服务（Property Service）**

在研究这些配置文件之前应先了解init是如何处理这些属性的。编写过Windows本地应用的读者都应了解，在windows中有一个注册表机制，在注册表中提供了大量的属性。在Linux中也有类似的机制，这就是属性服务。init在启动的过程中会启动属性服务（Socket服务），并且在内存中建立一块存储区域，用来存储这些属性。当读取这些属性时，直接从这一内存区域读取，如果修改属性值，需要通过Socket连接属性服务完成。在init.c文件中的一个action函数中调用了start\_property\_service函数来启动属性服务，action是init.rc及其类似文件中的一种执行机制，由于内容比较多，所以关于init.rc文件中的执行机制将在下一篇文章中详细讨论。

     现在顺藤摸瓜，找到start\_property\_service函数，该函数在Property\_service.c文件中，该文件与init.c文件中同一个目录。

[复制代码](javascript:void(0);)

void start\_property\_service(void)

{

int fd;

// 装载不同的属性文件

load\_properties\_from\_file(PROP\_PATH\_SYSTEM\_BUILD);

load\_properties\_from\_file(PROP\_PATH\_SYSTEM\_DEFAULT);

load\_override\_properties();

/\* Read persistent properties after all default values have been loaded. \*/

load\_persistent\_properties();

// 创建socket服务（属性服务）

fd = create\_socket(PROP\_SERVICE\_NAME, SOCK\_STREAM, 0666, 0, 0);

if(fd < 0) return;

fcntl(fd, F\_SETFD, FD\_CLOEXEC);

fcntl(fd, F\_SETFL, O\_NONBLOCK);

// 开始服务监听

listen(fd, 8);

property\_set\_fd = fd;

}

[复制代码](javascript:void(0);)

      现在我们已经知道属性服务的启动方式了，那么在start\_property\_service函数中还涉及到如下两个宏。

PROP\_PATH\_SYSTEM\_BUILD

PROP\_PATH\_SYSTEM\_DEFAULT

      这两个宏都是系统预定义的属性文件名的路径。为了获取这些宏的定义，我们先进行另外一个函数的分析。

      在前面读取属性值时使用过一个property\_get函数，该函数在Property\_service.c中实现，代码如下：

[复制代码](javascript:void(0);)

const char\* property\_get(const char \*name)

{

prop\_info \*pi;

if(strlen(name) >= PROP\_NAME\_MAX) return 0;

pi = (prop\_info\*) \_\_system\_property\_find(name);

if(pi != 0) {

return pi->value;

} else {

return 0;

}

}

[复制代码](javascript:void(0);)

      可以看到，在property\_get函数中调用了一个核心函数\_\_system\_property\_find，该函数真正实现了获取属性值的功能。该函数属于bionic的一个library，在system\_properties.c文件中实现，读者可以在如下的目录找到该文件。

<Android源代码根目录>/bionic/libc/bionic

\_\_system\_property\_find函数的代码如下：

[复制代码](javascript:void(0);)

const prop\_info \*\_\_system\_property\_find(const char \*name)

{

// 获取属性存储内存区域的首地址

prop\_area \*pa = \_\_system\_property\_area\_\_;

unsigned count = pa->count;

unsigned \*toc = pa->toc;

unsigned len = strlen(name);

prop\_info \*pi;

while(count--) {

unsigned entry = \*toc++;

if(TOC\_NAME\_LEN(entry) != len) continue;

pi = TOC\_TO\_INFO(pa, entry);

if(memcmp(name, pi->name, len)) continue;

return pi;

}

return 0;

}

[复制代码](javascript:void(0);)

      从\_\_system\_property\_find函数的代码很容易看出，第一行使用了一个\_\_system\_property\_area\_\_变量，该变量是全局的。在前面分析main函数时涉及到一个property\_init函数，该函数调用了init\_property\_area函数，该函数用于初始化属性内存区域，也就是\_\_system\_property\_area\_\_变量。

[复制代码](javascript:void(0);)

static int init\_property\_area(void)

{

prop\_area \*pa;

if(pa\_info\_array)

return -1;

if(init\_workspace(&pa\_workspace, PA\_SIZE))

return -1;

fcntl(pa\_workspace.fd, F\_SETFD, FD\_CLOEXEC);

pa\_info\_array = (void\*) (((char\*) pa\_workspace.data) + PA\_INFO\_START);

pa = pa\_workspace.data;

memset(pa, 0, PA\_SIZE);

pa->magic = PROP\_AREA\_MAGIC;

pa->version = PROP\_AREA\_VERSION;

/\* 初始化属性内存区域，属性服务会使用该区域 \*/

\_\_system\_property\_area\_\_ = pa;

property\_area\_inited = 1;

return 0;

}

[复制代码](javascript:void(0);)

      在前面涉及到的system\_properties.c文件对应的头文件system\_properties.h中定义了前面提到的两个表示属性文件路径的宏，其实还有另外两个表示路径的宏，一共4个属性文件。system\_properties.h文件可以在<Android源代码根目录>/bionic/libc/include/sys目录中找到。这4个宏定义如下：

#define PROP\_PATH\_RAMDISK\_DEFAULT "/default.prop"

#define PROP\_PATH\_SYSTEM\_BUILD "/system/build.prop"

#define PROP\_PATH\_SYSTEM\_DEFAULT "/system/default.prop"

#define PROP\_PATH\_LOCAL\_OVERRIDE "/data/local.prop"

       现在读者可以进入Android设备的相应目录，通常可以找到上述4个文件，如一般会在根目录，会发现一个default.prop文件，cat default.prop会看到该文件的内容。而属性服务就是装载所有这4个属性文件中的所有属性以及使用property\_set设置的属性。在Android设备的终端可以直接使用getprop命令从属性服务获取所有的属性值。如图2所示。getprop命令还可以直接根属性名还获取具体的属性值，例如，getprop ro.build.product。

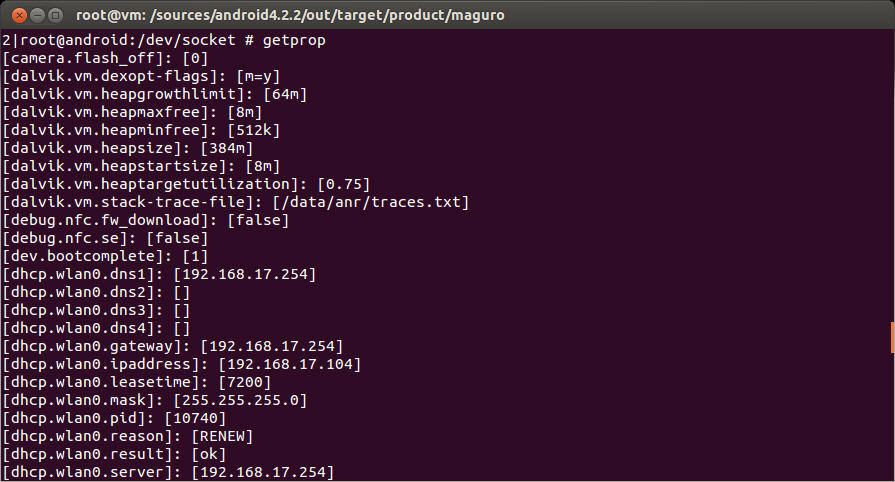


                                                               图2

      如果读者感兴趣，可以看一下getprop是如何通过属性服务读写属性的。getprop命令的源代码文件是getprop.c。读者可以在<Android源代码根目录>/system/core/toolbox目录中找到该文件。实际上，getprop获取属性值也是通过property\_get函数完成的。在前面分析过该函数，实际上调用了\_\_system\_property\_find函数从\_\_system\_property\_area\_\_变量指定的内存区域获取相应的属性值。

      此外在system\_properties.c文件中还有如下两个函数用于通过属性服务修改或添加某个属性的值。

[复制代码](javascript:void(0);)

static int send\_prop\_msg(prop\_msg \*msg)

{

struct pollfd pollfds[1];

struct sockaddr\_un addr;

socklen\_t alen;

size\_t namelen;

int s;

int r;

int result = -1;

// 创建用于连接属性服务的socket

s = socket(AF\_LOCAL, SOCK\_STREAM, 0);

if(s < 0) {

return result;

}

memset(&addr, 0, sizeof(addr));

// property\_service\_socket是Socket设备文件名称

namelen = strlen(property\_service\_socket);

strlcpy(addr.sun\_path, property\_service\_socket, sizeof addr.sun\_path);

addr.sun\_family = AF\_LOCAL;

alen = namelen + offsetof(struct sockaddr\_un, sun\_path) + 1;

if(TEMP\_FAILURE\_RETRY(connect(s, (struct sockaddr \*) &addr, alen)) < 0) {

close(s);

return result;

}

r = TEMP\_FAILURE\_RETRY(send(s, msg, sizeof(prop\_msg), 0));

if(r == sizeof(prop\_msg)) {

pollfds[0].fd = s;

pollfds[0].events = 0;

r = TEMP\_FAILURE\_RETRY(poll(pollfds, 1, 250 /\* ms \*/));

if (r == 1 && (pollfds[0].revents & POLLHUP) != 0) {

result = 0;

} else {

result = 0;

}

}

close(s);

return result;

}

// 用户可以直接调用该函数设置属性值

int \_\_system\_property\_set(const char \*key, const char \*value)

{

int err;

int tries = 0;

int update\_seen = 0;

prop\_msg msg;

if(key == 0) return -1;

if(value == 0) value = "";

if(strlen(key) >= PROP\_NAME\_MAX) return -1;

if(strlen(value) >= PROP\_VALUE\_MAX) return -1;

memset(&msg, 0, sizeof msg);

msg.cmd = PROP\_MSG\_SETPROP;

strlcpy(msg.name, key, sizeof msg.name);

strlcpy(msg.value, value, sizeof msg.value);

// 设置属性值

err = send\_prop\_msg(&msg);

if(err < 0) {

return err;

}

return 0;

}

[复制代码](javascript:void(0);)

    在send\_prop\_msg函数中涉及到一个property\_service\_socket变量，定义如下：

static const char property\_service\_socket[] = "/dev/socket/" PROP\_SERVICE\_NAME;

     实际上，send\_prop\_msg通过这个设备文件与属性服务通讯的。读者可以在Android设备的终端进入/dev/socket目录，通常会看到一个property\_service文件，该文件就是属性服务映射的设备文件。

     现在已经分析完了init如何确定与硬件相关的初始化文件名（init.grouper.rc），并且讨论了4个属性文件及其装载过程，以及属性服务实现的基本原理。在下一篇文章中将讨论更深入的内容，例如，init.rc文件中提供了很多action，那么什么是aciton呢，init有是如何解析init.rc文件呢？这些内容都将在下一篇文章中揭晓。

[**Android的init过程（二）：初始化语言（init.rc）解析**](http://www.cnblogs.com/nokiaguy/p/3164799.html)

    在上一篇文章中介绍了init的初始化第一阶段，也就是处理各种属性。在本文将会详细分析init最重要的一环：解析init.rc文件。

init.rc 文件并不是普通的配置文件，而是由一种被称为“Android初始化语言”（Android Init Language，这里简称为AIL）的脚本写成的文件。在了解init如何解析init.rc文件之前，先了解AIL非常必要，否则机械地分析 init.c及其相关文件的源代码毫无意义。

     为了学习AIL，读者可以到自己Android手机的根目录寻找init.rc文件，最好下载到本地以便查看，如果有编译好的Android源代码， 在<Android源代码根目录>out/target/product/generic/root目录也可找到init.rc文件。

AIL由如下4部分组成。

1.  动作（Actions）

2.  命令（Commands）

3． 服务（Services）

4.  选项（Options）

     这4部分都是面向行的代码，也就是说用回车换行符作为每一条语句的分隔符。而每一行的代码由多个符号（Tokens）表示。可以使用反斜杠转义符在 Token中插入空格。双引号可以将多个由空格分隔的Tokens合成一个Tokens。如果一行写不下，可以在行尾加上反斜杠，来连接下一行。也就是 说，可以用反斜杠将多行代码连接成一行代码。

     AIL的注释与很多Shell脚本一行，以#开头。

     AIL在编写时需要分成多个部分（Section），而每一部分的开头需要指定Actions或Services。也就是说，每一个Actions或 Services确定一个Section。而所有的Commands和Options只能属于最近定义的Section。如果Commands和 Options在第一个Section之前被定义，它们将被忽略。

Actions和Services的名称必须唯一。如果有两个或多个Action或Service拥有同样的名称，那么init在执行它们时将抛出错误，并忽略这些Action和Service。

下面来看看Actions、Services、Commands和Options分别应如何设置。

Actions的语法格式如下：

on <trigger>

<command>

<command>

<command>

      也就是说Actions是以关键字on开头的，然后跟一个触发器，接下来是若干命令。例如，下面就是一个标准的Action。

on boot

ifup lo

hostname localhost

domainname localdomain

其中boot是触发器，下面三行是command

那么init.rc到底支持哪些触发器呢？目前init.rc支持如下5类触发器。

1.  boot

   这是init执行后第一个被触发Trigger，也就是在 /init.rc被装载之后执行该Trigger

2.  <name>=<value>

   当属性<name>被设置成<value>时被触发。例如，

on property:vold.decrypt=trigger\_reset\_main

    class\_reset main

3.  device-added-<path>

    当设备节点被添加时触发

4.  device-removed-<path>

   当设备节点被移除时添加

5. service-exited-<name>

   会在一个特定的服务退出时触发

Actions后需要跟若干个命令，这些命令如下：

1.  exec <path> [<argument> ]\*

  创建和执行一个程序（<path>）。在程序完全执行前，init将会阻塞。由于它不是内置命令，应尽量避免使用exec ，它可能会引起init执行超时。

    2.  export <name> <value>

在全局环境中将 <name>变量的值设为<value>。（这将会被所有在这命令之后运行的进程所继承）

3.  ifup <interface>

   启动网络接口

4.  import <filename>

   指定要解析的其他配置文件。常被用于当前配置文件的扩展

5.  hostname <name>

   设置主机名

6.  chdir <directory>

   改变工作目录

7.  chmod <octal-mode><path>

   改变文件的访问权限

8.  chown <owner><group> <path>

   更改文件的所有者和组

9.  chroot <directory>

  改变处理根目录

10.  class\_start<serviceclass>

   启动所有指定服务类下的未运行服务。

11  class\_stop<serviceclass>

  停止指定服务类下的所有已运行的服务。

12.  domainname <name>

   设置域名

13.  insmod <path>

   加载<path>指定的驱动模块

14.  mkdir <path> [mode][owner] [group]

   创建一个目录<path> ，可以选择性地指定mode、owner以及group。如果没有指定，默认的权限为755，并属于root用户和 root组。

15. mount <type> <device> <dir> [<mountoption> ]\*

   试图在目录<dir>挂载指定的设备。<device> 可以是mtd@name的形式指定一个mtd块设备。<mountoption>包括 "ro"、"rw"、"re

16.  setkey

   保留，暂时未用

17.  setprop <name><value>

   将系统属性<name>的值设为<value>。

18. setrlimit <resource> <cur> <max>

   设置<resource>的rlimit （资源限制）

19.  start <service>

   启动指定服务（如果此服务还未运行）。

20．stop<service>

   停止指定服务（如果此服务在运行中）。

21. symlink <target> <path>

   创建一个指向<path>的软连接<target>。

22. sysclktz <mins\_west\_of\_gmt>

   设置系统时钟基准（0代表时钟滴答以格林威治平均时（GMT）为准）

23.  trigger <event>

  触发一个事件。用于Action排队

24.  wait <path> [<timeout> ]

等待一个文件是否存在，当文件存在时立即返回，或到<timeout>指定的超时时间后返回，如果不指定<timeout>，默认超时时间是5秒。

25. write <path> <string> [ <string> ]\*

向<path>指定的文件写入一个或多个字符串。

Services （服务）是一个程序，他在初始化时启动，并在退出时重启（可选）。Services （服务）的形式如下：

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

<option>

<option>

例如，下面是一个标准的Service用法

[复制代码](javascript:void(0);)

service servicemanager /system/bin/servicemanager

class core

user system

group system

critical

onrestart restart zygote

onrestart restart media

onrestart restart surfaceflinger

onrestart restart drm

[复制代码](javascript:void(0);)

Services的选项是服务的修饰符，可以影响服务如何以及怎样运行。服务支持的选项如下：

1.  critical

表明这是一个非常重要的服务。如果该服务4分钟内退出大于4次，系统将会重启并进入 Recovery （恢复）模式。

2. disabled

 表明这个服务不会同与他同trigger （触发器）下的服务自动启动。该服务必须被明确的按名启动。

3.  setenv <name><value>

在进程启动时将环境变量<name>设置为<value>。

4.  socket <name><type> <perm> [ <user> [ <group> ] ]

   Create a unix domain socketnamed /dev/socket/<name> and pass

   its fd to the launchedprocess.  <type> must be"dgram", "stream" or "seqpacket".

   User and group default to0.

   创建一个unix域的名为/dev/socket/<name> 的套接字，并传递它的文件描述符给已启动的进程。<type> 必须是 "dgram","stream" 或"seqpacket"。用户和组默认是0。

5.  user <username>

在启动这个服务前改变该服务的用户名。此时默认为 root。

6.  group <groupname> [<groupname> ]\*

在启动这个服务前改变该服务的组名。除了（必需的）第一个组名，附加的组名通常被用于设置进程的补充组（通过setgroups函数），档案默认是root。

7.  oneshot

   服务退出时不重启。

8.  class <name>

   指定一个服务类。所有同一类的服务可以同时启动和停止。如果不通过class选项指定一个类，则默认为"default"类服务。

9. onrestart

    当服务重启，执行一个命令（下详）。

     现在接着分析一下init是如何解析init.rc的。现在打开system/core/init/init.c文件，找到main函数。在上一篇文章中 分析了main函数的前一部分（初始化属性、处理内核命令行等），现在找到init\_parse\_config\_file函数，调用代码如下：

init\_parse\_config\_file("/init.rc");

这个方法主要负责初始化和分析init.rc文件。init\_parse\_config\_file函数在init\_parser.c文件中实现，代码如下：

[复制代码](javascript:void(0);)

int init\_parse\_config\_file(const char \*fn)

{

char \*data;

data = read\_file(fn, 0);

if (!data) return -1;

/\* 实际分析init.rc文件的代码 \*/

parse\_config(fn, data);

DUMP();

return 0;

}

[复制代码](javascript:void(0);)

      init\_parse\_config\_file方法开始调用了read\_file函数打开了/init.rc文件，并返回了文件的内容（char\*类 型），然后最核心的函数是parse\_config。该函数也在init\_parser.c文件中实现，代码如下：

[复制代码](javascript:void(0);)

static void parse\_config(const char \*fn, char \*s)

{

struct parse\_state state;

struct listnode import\_list;

struct listnode \*node;

char \*args[INIT\_PARSER\_MAXARGS];

int nargs;

nargs = 0;

state.filename = fn;

state.line = 0;

state.ptr = s;

state.nexttoken = 0;

state.parse\_line = parse\_line\_no\_op;

list\_init(&import\_list);

state.priv = &import\_list;

/\* 开始获取每一个token，然后分析这些token，每一个token就是有空格、字表符和回车符分隔的字符串

\*/

for (;;) {

/\* next\_token函数相当于词法分析器 \*/

switch (next\_token(&state)) {

case T\_EOF: /\* init.rc文件分析完毕 \*/

state.parse\_line(&state, 0, 0);

goto parser\_done;

case T\_NEWLINE: /\* 分析每一行的命令 \*/

/\* 下面的代码相当于语法分析器 \*/

state.line++;

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;

}

break;

case T\_TEXT: /\* 处理每一个token \*/

if (nargs < INIT\_PARSER\_MAXARGS) {

args[nargs++] = state.text;

}

break;

}

}

parser\_done:

/\* 最后处理由import导入的初始化文件 \*/

list\_for\_each(node, &import\_list) {

struct import \*import = node\_to\_item(node, struct import, list);

int ret;

INFO("importing '%s'", import->filename);

/\* 递归调用 \*/

ret = init\_parse\_config\_file(import->filename);

if (ret)

ERROR("could not import file '%s' from '%s'\n",

import->filename, fn);

}

}

[复制代码](javascript:void(0);)

    parse\_config方法的代码就比较复杂了，现在先说说该方法的基本处理流程。首先会调用  list\_init(&import\_list)初始化一个链表，该链表是用于存储通过import语句导入的初始化文件名。然后开始开始在 for循环中分析init.rc文件中的每一行代码。最后将init.rc文件分析完后，就会进入parser\_done部分，并递归调用 init\_parse\_config\_file方法分析通过import导入的初始化文件。

      通过分析parse\_config方法的原理，感觉也并不是很复杂。不过分析parse\_config方法的具体代码，还需要点编译原理的知识（只是概念 上的就可以）。在for循环中调用了一个next\_token方法不断从init.rc文件中获取token。这里的token，就是一种编程语言的最小 单元，也就是不可再分。例如，对于传统的编程语言，if、then等关键字、变量名等标识符都属于一个token。而对于init.rc文件来 说，import、on、以及触发器的参数值，都属于一个token。

     一个完整的编译器（或解析器）最开始需要进行词法和语法分析，词法分析就是在源代码文件中挑出一个个的Token，也就是说，词法分析器的返回值是 Token，而语法分析器的输入就是词法分析器的输出。也就是说，语法分析器需要分析一个个的token，而不是一个个的字符。由于init解析语言很简 单，所以就将词法和语法分析器放到了一起。词法分析器就是next\_token函数，而语法分析器就是T\_NEWLINE分支中的代码。这些就清楚多了。 现在先看看next\_token函数（在parser.c文件中实现）是如何获取每一个token的。

[复制代码](javascript:void(0);)

int next\_token(struct parse\_state \*state)

{

char \*x = state->ptr;

char \*s;

if (state->nexttoken) {

int t = state->nexttoken;

state->nexttoken = 0;

return t;

}

/\* 在这里开始一个字符一个字符地分析 \*/

for (;;) {

switch (\*x) {

case 0:

state->ptr = x;

return T\_EOF;

case '\n':

x++;

state->ptr = x;

return T\_NEWLINE;

case ' ':

case '\t':

case '\r':

x++;

continue;

case '#':

while (\*x && (\*x != '\n')) x++;

if (\*x == '\n') {

state->ptr = x+1;

return T\_NEWLINE;

} else {

state->ptr = x;

return T\_EOF;

}

default:

goto text;

}

}

textdone:

state->ptr = x;

\*s = 0;

return T\_TEXT;

text:

state->text = s = x;

textresume:

for (;;) {

switch (\*x) {

case 0:

goto textdone;

case ' ':

case '\t':

case '\r':

x++;

goto textdone;

case '\n':

state->nexttoken = T\_NEWLINE;

x++;

goto textdone;

case '"':

x++;

for (;;) {

switch (\*x) {

case 0:

/\* unterminated quoted thing \*/

state->ptr = x;

return T\_EOF;

case '"':

x++;

goto textresume;

default:

\*s++ = \*x++;

}

}

break;

case '\\':

x++;

switch (\*x) {

case 0:

goto textdone;

case 'n':

\*s++ = '\n';

break;

case 'r':

\*s++ = '\r';

break;

case 't':

\*s++ = '\t';

break;

case '\\':

\*s++ = '\\';

break;

case '\r':

/\* \ <cr> <lf> -> line continuation \*/

if (x[1] != '\n') {

x++;

continue;

}

case '\n':

/\* \ <lf> -> line continuation \*/

state->line++;

x++;

/\* eat any extra whitespace \*/

while((\*x == ' ') || (\*x == '\t')) x++;

continue;

default:

/\* unknown escape -- just copy \*/

\*s++ = \*x++;

}

continue;

default:

\*s++ = \*x++;

}

}

return T\_EOF;

}

[复制代码](javascript:void(0);)

      next\_token函数的代码还是很多的，不过原理到很简单。就是逐一读取init.rc文件（还有import导入的初始化文件）的字符，并将 由空格、“/t”和“/r”分隔的字符串挑出来，并通过state->text返回。如果返回了正常的token，next\_token函数就返回 T\_TEXT。如果一行结束，就返回T\_NEWLINE，如果init.rc文件的内容已读取完，就返回T\_EOF。当返回T\_NEWLINE时，开始语 法分析（由于init初始化语言是基于行的，所以语言分析实际上就是分析init.rc文件的每一行，只是这些行已经被分解成一个个token了）。感兴 趣的读者可以详细分析一下next\_token函数的代码，尽管代码很多，但并不复杂。而且还很有意思。

      现在回到parse\_config函数，先看一下T\_TEXT分支。该分支将获得的每一行的token都存储在args数组中。现在来看 T\_NEWLINE分支。该分支的代码涉及到一个state.parse\_line函数指针，该函数指针指向的函数负责具体的分析工作。但我们发现，一看 是该函数指针指向了一个空函数parse\_line\_no\_op，实际上，一开始该函数指针什么都不做，只是为了使该函数一开始不至于为null，否则调 用出错。

     现在来回顾一下T\_NEWLINE分支的完整代码。

[复制代码](javascript:void(0);)

case T\_NEWLINE:

state.line++;

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;

}

break;

[复制代码](javascript:void(0);)

      在上面的代码中首先调用了lookup\_keyword方法搜索关键字。该方法的作用是判断当前行是否合法，也就是根据Init初始化语言预定义的关键字 查询，如果未查到，返回K\_UNKNOWN。lookup\_keyword方法在init\_parser.c文件中实现，代码如下：

[复制代码](javascript:void(0);)

int lookup\_keyword(const char \*s)

{

switch (\*s++) {

case 'c':

if (!strcmp(s, "opy")) return K\_copy;

if (!strcmp(s, "apability")) return K\_capability;

if (!strcmp(s, "hdir")) return K\_chdir;

if (!strcmp(s, "hroot")) return K\_chroot;

if (!strcmp(s, "lass")) return K\_class;

if (!strcmp(s, "lass\_start")) return K\_class\_start;

if (!strcmp(s, "lass\_stop")) return K\_class\_stop;

if (!strcmp(s, "lass\_reset")) return K\_class\_reset;

if (!strcmp(s, "onsole")) return K\_console;

if (!strcmp(s, "hown")) return K\_chown;

if (!strcmp(s, "hmod")) return K\_chmod;

if (!strcmp(s, "ritical")) return K\_critical;

break;

case 'd':

if (!strcmp(s, "isabled")) return K\_disabled;

if (!strcmp(s, "omainname")) return K\_domainname;

break;

… …

case 'o':

if (!strcmp(s, "n")) return K\_on;

if (!strcmp(s, "neshot")) return K\_oneshot;

if (!strcmp(s, "nrestart")) return K\_onrestart;

break;

case 'r':

if (!strcmp(s, "estart")) return K\_restart;

if (!strcmp(s, "estorecon")) return K\_restorecon;

if (!strcmp(s, "mdir")) return K\_rmdir;

if (!strcmp(s, "m")) return K\_rm;

break;

case 's':

if (!strcmp(s, "eclabel")) return K\_seclabel;

if (!strcmp(s, "ervice")) return K\_service;

if (!strcmp(s, "etcon")) return K\_setcon;

if (!strcmp(s, "etenforce")) return K\_setenforce;

if (!strcmp(s, "etenv")) return K\_setenv;

if (!strcmp(s, "etkey")) return K\_setkey;

if (!strcmp(s, "etprop")) return K\_setprop;

if (!strcmp(s, "etrlimit")) return K\_setrlimit;

if (!strcmp(s, "etsebool")) return K\_setsebool;

if (!strcmp(s, "ocket")) return K\_socket;

if (!strcmp(s, "tart")) return K\_start;

if (!strcmp(s, "top")) return K\_stop;

if (!strcmp(s, "ymlink")) return K\_symlink;

if (!strcmp(s, "ysclktz")) return K\_sysclktz;

break;

case 't':

if (!strcmp(s, "rigger")) return K\_trigger;

break;

case 'u':

if (!strcmp(s, "ser")) return K\_user;

break;

case 'w':

if (!strcmp(s, "rite")) return K\_write;

if (!strcmp(s, "ait")) return K\_wait;

break;

}

return K\_UNKNOWN;

}

[复制代码](javascript:void(0);)

      lookup\_keyword方法按26个字母顺序（关键字首字母）进行处理。

     现在回到parse\_config方法的T\_NEWLIEN分支，接下来调用了kw\_is宏具体判断当前行是否合法，该宏以及SECTION宏的定义如下。根据这些代码。明显是keyword\_info数组中的某个元素的flags成员变量的值取最后一位。

#define SECTION 0x01

#define kw\_is(kw, type) (keyword\_info[kw].flags & (type))

现在问题又转到keyword\_info数组了。该数组也在init\_parser.c文件中定义，代码如下:

[复制代码](javascript:void(0);)

#include "keywords.h"

#define KEYWORD(symbol, flags, nargs, func) \

[ K\_##symbol ] = { #symbol, func, nargs + 1, flags, },

struct {

const char \*name;

int (\*func)(int nargs, char \*\*args);

unsigned char nargs;

unsigned char flags;

} keyword\_info[KEYWORD\_COUNT] = {

[ K\_UNKNOWN ] = { "unknown", 0, 0, 0 },

#include "keywords.h"

};

[复制代码](javascript:void(0);)

       从表面上看，keyword\_info数组是一个struct数组，但本质上，是一个map。为每一个数组元素设置了一个key，例如，数组元素{ "unknown", 0, 0,0 }的key是K\_UNKNOWN，而#include “keywords.h”大有玄机。上面的代码中引用了两次keywords.h文件，现在可以看一下keywords.h文件的代码。

[复制代码](javascript:void(0);)

#ifndef KEYWORD

int do\_chroot(int nargs, char \*\*args);

… …

int do\_export(int nargs, char \*\*args);

int do\_hostname(int nargs, char \*\*args);

int do\_rmdir(int nargs, char \*\*args);

int do\_loglevel(int nargs, char \*\*args);

int do\_load\_persist\_props(int nargs, char \*\*args);

int do\_wait(int nargs, char \*\*args);

#define \_\_MAKE\_KEYWORD\_ENUM\_\_

/\*

"K\_chdir", ENUM

\*/

#define KEYWORD(symbol, flags, nargs, func) K\_##symbol,

enum {

K\_UNKNOWN,

#endif

KEYWORD(capability, OPTION, 0, 0)

KEYWORD(chdir, COMMAND, 1, do\_chdir)

KEYWORD(chroot, COMMAND, 1, do\_chroot)

KEYWORD(class, OPTION, 0, 0)

KEYWORD(class\_start, COMMAND, 1, do\_class\_start)

KEYWORD(class\_stop, COMMAND, 1, do\_class\_stop)

KEYWORD(class\_reset, COMMAND, 1, do\_class\_reset)

KEYWORD(console, OPTION, 0, 0)

… …

KEYWORD(critical, OPTION, 0, 0)

KEYWORD(load\_persist\_props, COMMAND, 0, do\_load\_persist\_props)

KEYWORD(ioprio, OPTION, 0, 0)

#ifdef \_\_MAKE\_KEYWORD\_ENUM\_\_

KEYWORD\_COUNT,

};

#undef \_\_MAKE\_KEYWORD\_ENUM\_\_

#undef KEYWORD

#endif

[复制代码](javascript:void(0);)

      从keywords.h文件的代码可以看出，如果未定义KEYWORD宏，则在keywords.h文件中定义一个KEYWORD宏，以及一个枚举类型， 其中K\_##symbol的##表示连接的意思。而这个KEYWORD宏只用了第一个参数（symbol）。例 如，KEYWORD(chdir,       COMMAND, 1, do\_chdir)就会生成K\_chdir。

     而在keyword\_info结构体数组中再次导入keywords.h文件，这是KEYWORD宏已经在init\_parser.c文件中重新定义，所以第一次导入keywords.h文件使用的是如下的宏。

#define KEYWORD(symbol, flags, nargs, func) \

[ K\_##symbol ] = { #symbol, func, nargs + 1, flags, },

    这下就明白了，如果不使用keywords.h文件，直接将所有的代码都写到init\_parser.c文件中，就会有下面的代码。

[复制代码](javascript:void(0);)

int do\_chroot(int nargs, char \*\*args);

… …

enum

{

K\_UNKNOWN,

K\_ capability,

K\_ chdir,

… …

}

#define KEYWORD(symbol, flags, nargs, func) \

[ K\_##symbol ] = { #symbol, func, nargs + 1, flags, },

struct {

const char \*name;

int (\*func)(int nargs, char \*\*args);

unsigned char nargs;

unsigned char flags;

} keyword\_info[KEYWORD\_COUNT] = {

[ K\_UNKNOWN ] = { "unknown", 0, 0, 0 },

[K\_ capability] = {" capability ", 0, 1, OPTION },

[K\_ chdir] = {"chdir", do\_chdir ,2, COMMAND},

… …

#include "keywords.h"

};

[复制代码](javascript:void(0);)

  可能我们还记着lookup\_keyword方法，该方法的返回值就是keyword\_info数组的key。

      在keywords.h前面定义的函数指针都是处理init.rc文件中service、action和command的。现在就剩下一个问题了，在哪里 为这些函数指针赋值呢，也就是说，具体处理每个部分的函数在哪里呢。现在回到前面的语法分析部分。如果当前行合法，则会执行 parse\_new\_section函数（在init\_parser.c文件中实现），该函数将为section和action设置处理这两部分的函数。 parse\_new\_section函数的代码如下：

[复制代码](javascript:void(0);)

void parse\_new\_section(struct parse\_state \*state, int kw,

int nargs, char \*\*args)

{

printf("[ %s %s ]\n", args[0],

nargs > 1 ? args[1] : "");

switch(kw) {

case K\_service: // 处理service

state->context = parse\_service(state, nargs, args);

if (state->context) {

state->parse\_line = parse\_line\_service;

return;

}

break;

case K\_on: // 处理action

state->context = parse\_action(state, nargs, args);

if (state->context) {

state->parse\_line = parse\_line\_action;

return;

}

break;

case K\_import: // 单独处理import导入的初始化文件。

parse\_import(state, nargs, args);

break;

}

state->parse\_line = parse\_line\_no\_op;

}

[复制代码](javascript:void(0);)

现在看一下处理service的函数（parse\_line\_service）。

[复制代码](javascript:void(0);)

static void parse\_line\_service(struct parse\_state \*state, int nargs, char \*\*args)

{

struct service \*svc = state->context;

struct command \*cmd;

int i, kw, kw\_nargs;

if (nargs == 0) {

return;

}

svc->ioprio\_class = IoSchedClass\_NONE;

kw = lookup\_keyword(args[0]);

// 下面处理每一个option

switch (kw) {

case K\_capability:

break;

… …

case K\_group:

if (nargs < 2) {

parse\_error(state, "group option requires a group id\n");

} else if (nargs > NR\_SVC\_SUPP\_GIDS + 2) {

parse\_error(state, "group option accepts at most %d supp. groups\n",

NR\_SVC\_SUPP\_GIDS);

} else {

int n;

svc->gid = decode\_uid(args[1]);

for (n = 2; n < nargs; n++) {

svc->supp\_gids[n-2] = decode\_uid(args[n]);

}

svc->nr\_supp\_gids = n - 2;

}

break;

case K\_keycodes:

if (nargs < 2) {

parse\_error(state, "keycodes option requires atleast one keycode\n");

} else {

svc->keycodes = malloc((nargs - 1) \* sizeof(svc->keycodes[0]));

if (!svc->keycodes) {

parse\_error(state, "could not allocate keycodes\n");

} else {

svc->nkeycodes = nargs - 1;

for (i = 1; i < nargs; i++) {

svc->keycodes[i - 1] = atoi(args[i]);

}

}

}

break;

… …

}

……

}

[复制代码](javascript:void(0);)

  Action的处理方式与service类似，读者可以自行查看相应的函数代码。现在一切都清楚了。处理service的函数是 parse\_line\_service，处理action的函数是parse\_line\_action。而前面的state.parse\_line根据当 前是service还是action，指向这两个处理函数中的一个，并执行相应的函数处理actioncommand和serviceoption。

    综合上述，实际上分析init.rc文件的过程就是通过一系列地处理，最终转换为通过parse\_line\_service或parse\_line\_action函数分析Init.rc文件中每一行的行为。