**init.rc 语法与解析**

本文基于Android 6.0，涉及源码如下：

/system/core/rootdir/init.rc  
/system/core/init/init\_parser.cpp

**一：init.rc 语法**

一个完整的init.rc 脚本由4中类型声明组成。即：  
Action（动作）  
Commands（命令）  
Services（服务）  
Options（选项）

Action 和Services 表明一个新语句的开始，这两个关键字后面跟着的Commands 或者 Options 都是属于这个语句；  
Action 和Services 都有唯一的名字，如果出现动作或者服务重名，则会被当做错误处理。

**1：Action（动作）**

**语法格式**

on <trigger> ##触发条件

<command1> ##执行命令

<command2> ##可以同时执行多个命令

<command3>

其中trigger 是触发条件，即当触发条件满足时将 依次 执行commands

举例说明：下面这句话的意思是当属性值满足persist.service.adb.enable=1时，启动adbd 进程；

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

start adbd

具体实现：当系统变化的时候，系统会对init.rc 中的每个<trigger>进行匹配，只要发现符合条件的Action，就会把它加入 “命令执行队列”的尾部（除非这个Action 在队列中已经存在）， 然后系统在对这些Commands按顺序执行；

**2：Commands（命令）**

命令将在触发条件发生时被一个个执行。

**init.rc中常见的trigger如下：**

| **trigger** | **Description** |
| --- | --- |
| boot | init程序启动后触发的第一个事件 |
| <name>=<value> | 当属性<name> 满足<value>时触发 |
| device-added/removed-<patch> | 当设备节点添加/删除时触发此事件 |
| sevice-exited-<name> | 当指定服务<name> 存在时触发 |

**init.rc中常见的Commands如下：**

| **Command** | **Description** |
| --- | --- |
| exec <path> [<argument>]\* | Fork 并执行一个程序，其路径为<path>。该命令将阻塞 直到该程序启动完成 |
| export <name> <value> | 设置某个环境变量<name> 的值为<value>。对全局有效，之后的进程都将继承这个变量 |
| ifup <interface> | 使网络接口<interface> 成功连接 |
| import <filename> | 解析另一个配置文件<filename>，以扩展当前配置 |
| chdir <directory> | 更换工作目录为<directory> |
| chmod <octal-mode> <path> | 更改文件访问权限 |
| chown <owner> <group> <path> | 更改文件所有者和群组 |
| mount <type> <device> <dir> [<mountoption>]\* | 尝试在指定路径上挂载一个设备 |
| start <service> | 启动一个服务，如果它没有处于运行状态的话 |
| stop <service> | 停止一个服务，如果它当前处于运行状态的话 |
| setprop <name> <value> | 设置系统属性<name> 的值为 <value> |
| trigger <event> | 触发一个事件 |

很多没有列举完成，但是很多Commands 可以根据名字理解意思，如下举几个例子，例子都是init.rc中的内容；

on init

symlink /sys/kernel/debug /d

mount cgroup none /acct cpuacct

mkdir /acct/uid

chown root system /sys/fs/cgroup/memory/sw/tasks

chmod 0660 /sys/fs/cgroup/memory/sw/tasks

on property:sys.boot\_from\_charger\_mode=1

class\_stop charger

trigger late-init

**3：Services（服务）**

**3.1：命令格式：**

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

<option>

<option>

...

* <name>——表示service 的名字；
* <pathname>——表示service所在路径，此处的service是可执行文件，所以一定有存储路径；
* <argument>——启动service所带的参数；
* <option>——对此service的约束选项，后面将详细讲解；

**Service 样例如下：**

service servicemanager /system/bin/servicemanager

class core

user system

group system

critical

onrestart restart healthd

onrestart restart zygote

onrestart restart media

onrestart restart surfaceflinger

onrestart restart drm

**4：Options（选项）**

**init.rc 中可用的选项如下：**

| **Option** | **Description** |
| --- | --- |
| critical | 表明这是对设备至关重要的服务；如果它在四分钟内退出超过四次，则设备将进入Recovery 模式 |
| disabled | 表示此服务是不会自动启动，而是需要通过显示调用服务名来启动 |
| setenv <name> <value> | 设置环境变量<name> 为值<value> |
| socket <name> <type> <perm> [<user> [<group>] ] | 创建一个名为dev/socket/<name>的 socket，然后将它的fd值传给启动它的进，有效的<type>值包括dgram,stream 和seqpacket。ueser 和group 的默认值为0。 |
| user <username> | 在启动服务前将用户切换至<username>,默认情况下用户都是root。 |
| group <groupname> [<groupname>]\* | 在启动服务前将用户组切换至<groupname> |
| oneshot | 当次服务退出时，不要主动去重启它 |
| class <name> | 为该服务指定一个class 名。同一个class 的所有服务必须同时启动或者停止。默认情况下服务的class名是“default” |
| onrestart | 当次服务重启时，执行某些命令 |

**二：init.rc解析**

/system/core/init/init.cpp  
/system/core/init/init\_parser.cpp

**1：init.rc脚本基本结构**

init.rc 脚本的基本单位为section，即片段，section 分为三类，分别由三个关键字(所谓关键字即每一行的第一列)来区分，这三个关键字是on、service、import。

* on ——即前文的action 的起始关键字；  
  on 类型的section 表示一系列命令的组合；

on init

loglevel 3

symlink /system/etc /etc

symlink /sys/kernel/debug /d

symlink /system/vendor /vendor

mount cgroup none /acct cpuacct

mkdir /acct/uid

这条命令包含上述的命令，命令的执行事以section 为单位执行的，所以这几条命令是*一起执行*的，而不会单独执行，那它们是在那里执行呢？——这是由init.cpp中的main()方法决定的，init.cpp在某个阶段会执行

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

该方法就是将 on init 开始的section 里面的所有命令加入到一个执行队列，在未来的某个时间会**顺序**的执行，所以调用action\_for\_each\_trigger的先后决定了命令执行的先后顺序；

* service ——即前文所说的服务起始关键字；  
  service类型的section代表了一个可执行程序；

service servicemanager /system/bin/servicemanager

class core

user system

group system

critical

onrestart restart healthd

onrestart restart zygote

onrestart restart media

onrestart restart surfaceflinger

onrestart restart drm

服务的内容之前已经介绍过，可以看看前文，那service 类型的section标识的服务在什么时候调用呢？—— 是在class\_start命令被执行的时候，class\_start命令总是存在于on 类型的section中；

**class\_start core** #这样一条命令代表将启动所有类型为core的service  
所以可以看出[android](https://link.jianshu.com/?t=http://lib.csdn.net/base/android" \t "_blank)的启动过程主要就是on类型的section被执行的过程。

* import —— import类型的section代表将引入另外一个.rc文件；

import /init.environ.rc  
import /init.usb.rc

相当包含另外一些section, 在解析完init.rc文件后继续会调用init\_parse\_config\_file来解析引入的.rc文件

**2：解析过程**

init.rc是一个纯文本文件，我们需要程序去解析init.rc，然后交给程序运行，解析init.rc的过程就是识别一个个section的过程，将各个section的信息保存下来，然后在init.c的main()中去执行一个个命令。

android采用双向链表来存储section的信息，解析完成之后，会得到三个双向链表action\_list、service\_list、import\_list来分别存储三种section的信息上。

/system/core/init/init\_parser.cpp  
static list\_declare(service\_list);  
static list\_declare(action\_list);  
static list\_declare(action\_queue);

**1:init.c中调用init\_parse\_config\_file("/init.rc")**

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

{

char \*data;

data = read\_file(fn, 0);//读文件数据

if (!data) return -1;

parse\_config(fn, data);//解析数据

DUMP();

return 0;

}

**2:parse\_config**

static void parse\_config(const char \*fn, const std::string& data)

{

struct listnode import\_list;

struct listnode \*node;

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

int nargs = 0;

parse\_state state;

state.filename = fn;

state.line = 0;

state.ptr = strdup(data.c\_str()); // TODO: fix this code!

state.nexttoken = 0;

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

list\_init(&import\_list);

state.priv = &import\_list;

for (;;) {

switch (next\_token(&state)) { //next\_token()根据从state.ptr开始遍历

case T\_EOF: //遍历到文件结尾，然后goto解析import的.rc文件

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

goto parser\_done;

case T\_NEWLINE: //到了一行结束

state.line++;

if (nargs) {

int kw = lookup\_keyword(args[0]); // 找到这一行的关键字

if (kw\_is(kw, SECTION)) { // 如果这是一个section的第一行

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

parse\_new\_section(&state, kw, nargs, args);

} else { //如果这不是一个section的第一行

state.parse\_line(&state, nargs, args);

}

nargs = 0;

}

break;

case T\_TEXT: // 遇到普通字符

if (nargs < INIT\_PARSER\_MAXARGS) {

args[nargs++] = state.text;

}

break;

}

}

parser\_done:

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

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

int ret;

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

if (ret)

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

import->filename, fn);

}

}

next\_token() 解析完init.rc中一行之后，会返回T\_NEWLINE，这时调用lookup\_keyword函数来找出这一行的关键字, lookup\_keyword返回的是一个整型值，对应keyword\_info[]数组的下标，keyword\_info[]存放的是keyword\_info结构体类型的数据，

struct {

const char \*name; //关键字的名称

int (\*func)(int nargs, char \*\*args); //对应的处理函数

unsigned char nargs; //参数个数

unsigned char flags; //flag标识关键字的类型,包括COMMAND、OPTION、SECTION

} keyword\_info

**keyword\_info[]中存放的是所有关键字信息，每一项对应一个关键字；**  
根据每一项的flags就可以判断出关键字的类型，如新的一行是SECTION，就调用parse\_new\_section()来解析这一行, 如新的一行不是一个SECTION的第一行，那么调用state.parseline()来解析(state.parseline所对应的函数会根据section类型的不同而不同)，在parse\_new\_section()中进行动态设置。

static 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: //解析service

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

if (state->context) {

state->parse\_line = parse\_line\_service; //Service 对应的parse\_line

return;

}

break;

case K\_on://解析 on 开头的action

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

if (state->context) {

state->parse\_line = parse\_line\_action; //action 对应的parse\_line

return;

}

break;

case K\_import: //解析import

parse\_import(state, nargs, args);

break;

}

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

}

*三种类型的section: service、on、import, service对应的state.parse\_line为parse\_line\_service,  
on对应的state.parse\_line为parse\_line\_action, import section中只有一行所以没有对应的state.parseline。*

鉴于篇幅原因，使用文字描述parse\_service过程，parse\_action，parse\_import解析也是类似的：

**parse\_service会去service\_list链表查找有没有该命名的service(svc = service\_find\_by\_name(args[1]);)，如果没有则会构造service 对象并添加到service\_list链表中，返回该service 对象；但是这个service 没有Options，然后通过parse\_line\_service 解析Options选项填充service 对象；**

**3：相关结构体**

一个on类型的section对应一个action， action类型定义如下：

struct action {

/\* node in list of all actions \*/

struct listnode alist;

/\* node in the queue of pending actions \*/

struct listnode qlist;

/\* node in list of actions for a trigger \*/

struct listnode tlist;

unsigned hash;

/\* list of actions which triggers the commands\*/

struct listnode triggers;

struct listnode commands; //command的双向链表

struct command \*current;

};

每个on类型section的第二行开始每一行都解析了一个command, 所有command组成一个双向链表指向该action的commands字段中。

command结构体定义如下：

struct command

{

/\* list of commands in an action \*/

struct listnode clist;

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

int line;

const char \*filename;

int nargs;

char \*args[1];

};

**command结构体比较简单, 用于标识一个命令，包含双向链表指针、对应的执行函数、参数个数以及命令关键字。**

**service结构体定义如下：**

struct service {

void NotifyStateChange(const char\* new\_state);

/\* list of all services \*/

struct listnode slist;

char \*name;

const char \*classname;

unsigned flags;

pid\_t pid;

time\_t time\_started; /\* time of last start \*/

time\_t time\_crashed; /\* first crash within inspection window \*/

int nr\_crashed; /\* number of times crashed within window \*/

uid\_t uid;

gid\_t gid;

gid\_t supp\_gids[NR\_SVC\_SUPP\_GIDS];

size\_t nr\_supp\_gids;

const char\* seclabel;

struct socketinfo \*sockets;

struct svcenvinfo \*envvars;

struct action onrestart; /\* Actions to execute on restart. \*/

std::vector<std::string>\* writepid\_files\_;

/\* keycodes for triggering this service via /dev/keychord \*/

int \*keycodes;

int nkeycodes;

int keychord\_id;

IoSchedClass ioprio\_class;

int ioprio\_pri;

int nargs;

/\* "MUST BE AT THE END OF THE STRUCT" \*/

char \*args[1];

};

service结构体存储了service的相关信息, 包括进程号、启动时间、名字等， onrestart这个option后面通常跟着一个命令，所以也用action结构体来表示。  
例如：

service servicemanager /system/bin/servicemanager

class core

user system

group system

critical

onrestart restart healthd

onrestart restart zygote

onrestart restart media

onrestart restart surfaceflinger

onrestart restart drm

**init.rc中配置的service 启动流程：**

/system/core/rootdir/init.rc

class\_start core

on nonencrypted

class\_start main

class\_start late\_start

/system/core/init/keywords.h

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

/system/core/init/builtins.c

int do\_class\_start(int nargs, char \*\*args)

{

/\* Starting a class does not start services

\* which are explicitly disabled. They must

\* be started individually.

\*/

service\_for\_each\_class(args[1], service\_start\_if\_not\_disabled);

return 0;

}

/system/core/init/init\_parser.cpp

void service\_for\_each\_class(const char \*classname,

void (\*func)(struct service \*svc))

{

struct listnode \*node;

struct service \*svc;

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

svc = node\_to\_item(node, struct service, slist);

if (!strcmp(svc->classname, classname)) {

func(svc);

}

}

}

/system/core/init/builtins.cpp

static void service\_start\_if\_not\_disabled(struct service \*svc)

{

if (!(svc->flags & SVC\_DISABLED)) {

service\_start(svc, NULL);

} else {

svc->flags |= SVC\_DISABLED\_START;

}

}

/system/core/init/init.c

void service\_start(struct service \*svc, const char \*dynamic\_args)

{

struct stat s;

pid\_t pid;

int needs\_console;

int n;

char \*scon = NULL;

int rc;

/\* starting a service removes it from the disabled or reset

\* state and immediately takes it out of the restarting

\* state if it was in there

\*/

......

pid = fork();//fork 子进程

......

for (si = svc->sockets; si; si = si->next) {

int socket\_type = (

!strcmp(si->type, "stream") ? SOCK\_STREAM :

(!strcmp(si->type, "dgram") ? SOCK\_DGRAM : SOCK\_SEQPACKET));//判断socket 类型

int s = create\_socket(si->name, socket\_type,

si->perm, si->uid, si->gid, si->socketcon ?: scon);//创建socket对应的fd

if (s >= 0) {

publish\_socket(si->name, s);//发布socket的fd，即将create\_socket返回的fd添加到

//以"ANDROID\_SOCKET\_"为前缀的环境变量；

//Zygote的socket的key 为ANDROID\_SOCKET\_zygote，也是在这创建的；

}

}

9人点赞