**Android init.rc文件解析过程详解(一)**

<http://blog.itpub.net/7232789/viewspace-758162/>

一、init.rc文件结构介绍

init.rc文件基本组成单位是section, section分为三种类型，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类型的section表示一系列命令的组合, 例如：

on init

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

    export ANDROID\_ROOT /system

    export ANDROID\_DATA /data

这样一个section包含了三个export命令，命令的执行是以section为单位的，所以这三个命令是一起执行的，不会单独执行, 那什么时候执行呢？ 这是由init.c的main()所决定的，main()里在某个时间会调用

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

这就把on init开始的这样一个section里的所有命令加入到一个执行队列，在未来的某个时候会顺序执行队列里的命令，所以调用action\_for\_each\_trigger的先后决定了命令执行的先后。

service类型的section表示一个可执行程序，例如:

service surfaceflinger /system/bin/surfaceflinger

    class main

    user system

    group graphics drmrpc

    onrestart restart zygote

surfaceflinger作为一个名字标识了这个service,  /system/bin/surfaceflinger表示可执行文件的位置，class、user、group、onrestart这些关键字所对应的行都被称为options, options是用来描述的service一些特点，不同的service有着不同的options。

service类型的section标识了一个service(或者说可执行程序), 那这个service什么时候被执行呢？是在class\_start这个命令被执行的时候，class\_start命令行总是存在于某个on类型的section中，“class\_start core”这样一条命令被执行，就会启动类型为core的所有service。

所以可以看出android的启动过程主要就是on类型的section被执行的过程。

import类型的section表示引入另外一个.rc文件，例如：

import init.test.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。

二、init.rc文件解析过程

       我们已经知道init.rc的结构，应该可以想到解析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(关于双向链表详解见本文第三部分)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。

1、init.c中调用init\_parse\_config\_file(“/init.rc”)， 代码如下：

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

{

    char \*data;

    data = read\_file(fn, 0);        //read\_file()调用open\lseek\read 将init.rc读出来

    if (!data) return -1;

    parse\_config(fn, data);        //调用parse\_config开始解析

    DUMP();

    return 0;

}

2、parse\_config()代码如下：

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;

    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;

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

         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

因此keyword\_info[]中存放的是所有关键字的信息，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。

三种类型的section: service、on、import,  service对应的state.parseline为parse\_line\_service,

on对应的state.parseline为parse\_line\_action, import section中只有一行所以没有对应的state.parseline。

Android init.rc文件解析过程详解(二)

<http://blog.itpub.net/7232789/viewspace-758167/>

3、parse\_new\_section代码如下:

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类型的section

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

        if (state->context) {

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

            return;

        }

        break;

    case K\_on:                                                        \\解析on类型的section

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

        if (state->context) {

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

            return;

        }

        break;

    case K\_import:                                                  \\解析import类型的section

        parse\_import(state, nargs, args);

        break;

    }

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

}

4、parse\_service()和parse\_line\_service()

parse\_service()代码如下：

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;

    }

    svc = service\_find\_by\_name(args[1]);            //在链表中查找当前行对应的service

    if (svc) {

        parse\_error(state, "ignored duplicate definition of service '%s'\n", args[1]);

        return 0;

    }

      //如果当前行对应的service还没有加入service\_list链表，则新建一个

    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);

    list\_add\_tail(&service\_list, &svc->slist);        //将这个service加入到service\_list

//注意此时svc对象基本上是一个空壳，因为相关的options还没有解析

    return svc;

}

parse\_line\_service()解析service对应的options行，主要是填充parse\_service()中创建的service对象。

5、parse\_action()和parse\_line\_action()

   parse\_action()函数主要是根据当前行的信息创建一个action结构体类型的对象，加入到action\_list双向链表中, 代码比较简单，有兴趣可自行研究。

parse\_line\_action()解析对应的命令行, 代码如下:

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

{

    struct command \*cmd;

    struct action \*act = state->context;

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

    int kw, n;

    if (nargs == 0) {

        return;

    }

    kw = lookup\_keyword(args[0]);

    if (!kw\_is(kw, COMMAND)) {

        parse\_error(state, "invalid command '%s'\n", args[0]);

        return;

    }

    n = kw\_nargs(kw);

    if (nargs < n) {

        parse\_error(state, "%s requires %d %s\n", args[0], n - 1,

            n > 2 ? "arguments" : "argument");

        return;

    }

    cmd = malloc(sizeof(\*cmd) + sizeof(char\*) \* nargs);      //生成一个command类型的对象

    cmd->func = kw\_func(kw);

    cmd->nargs = nargs;

    memcpy(cmd->args, args, sizeof(char\*) \* nargs);

    list\_add\_tail(&act->commands, &cmd->clist);      //将这个command对象加入actions->commands

}

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;

    const char \*name;

    struct listnode commands;             //command的双向链表

    struct command \*current;

};

因此，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。

Android init.rc文件解析过程详解(三)

<http://blog.itpub.net/7232789/viewspace-758168/>

三、相关结构体

     1、listnode

listnode结构体用于建立双向链表，这种结构广泛用于kernel代码中, android源代码中定义了listnode结构体以及相关操作双向链表的方法，与kernel中的定义类似。

这个实现的核心思想是：在用户自定义的结构体xx中定义一个listnode类型的成员,

这个listnode类型成员的作用就是能将xx类型的变量组成一个双向链表。下面我们来看一下是listnode是怎么做到的。

//listnode类型里面只有两个指针prev,next

struct listnode

{

    struct listnode \*next;

    struct listnode \*prev;

};

//将链表中的一个node转换成自定义结构体中的一个对象

#define node\_to\_item(node, container, member) \

    (container \*) (((char\*) (node)) - offsetof(container, member))

//初始化一个链表

void list\_init(struct listnode \*node)

{

    node->next = node;

    node->prev = node;

}

//将一个节点到链表

void list\_add\_tail(struct listnode \*head, struct listnode \*item)

{

    item->next = head;

    item->prev = head->prev;

    head->prev->next = item;

    head->prev = item;

}

//删除一个节点

void list\_remove(struct listnode \*item)

{

    item->next->prev = item->prev;

    item->prev->next = item->next;

}

理解node\_to\_item宏是理解listnode用法的关键，这个宏的作用是将一个listnode指针转换成了一个指定类型（自定义）的指针，这个宏先使用offsetof函数获取到指定结构体中指定成员变量的地址偏移量，然后通过指针运算获得listnode指针变量所在结构体变量的指针。

这种实现与我们课堂上所学的链表实现方法不太一样，教科书上的实现是在listnode中存储了自定义的数据，而这个实现是在自定义的数据当中存储listnode指针。

2、action结构体

     前面已经讲过on类型的section解析之后会生成一个双向链表action\_list, 这个action\_list每个node表示就是action结构体的对象，也就是说一个on类型的section都会生成一个action结构体的对象。

action结构体定义如下:

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;

    const char \*name;

    struct listnode commands;        //节点为command结构体的双向链表

    struct command \*current;

};

     action结构体除了用在on类型的section, 也用在service类型的section，下面介绍service结构体时会说明。

3、command结构体

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 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。

4、service结构体

struct service {

        /\* list of all services \*/

    struct listnode slist;                  //将结构体链接成service\_list用

    const 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;

#ifdef HAVE\_SELINUX

    char \*seclabel;

#endif

    struct socketinfo \*sockets;

    struct svcenvinfo \*envvars;

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

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

    int \*keycodes;

    int nkeycodes;

    int keychord\_id;

    int 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, 字段onrestart

就用到了action结构体， 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。

注：本文基于android4.2的源代码分析