1. **init.c**

main函数调用init\_parse\_config\_file(“/init.rc”)；

**Init\_praser.c**

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。

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

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。