# Android init进程之init.rc文件解析

作者:程姚根 联系方式:chengyaogen123@163.com

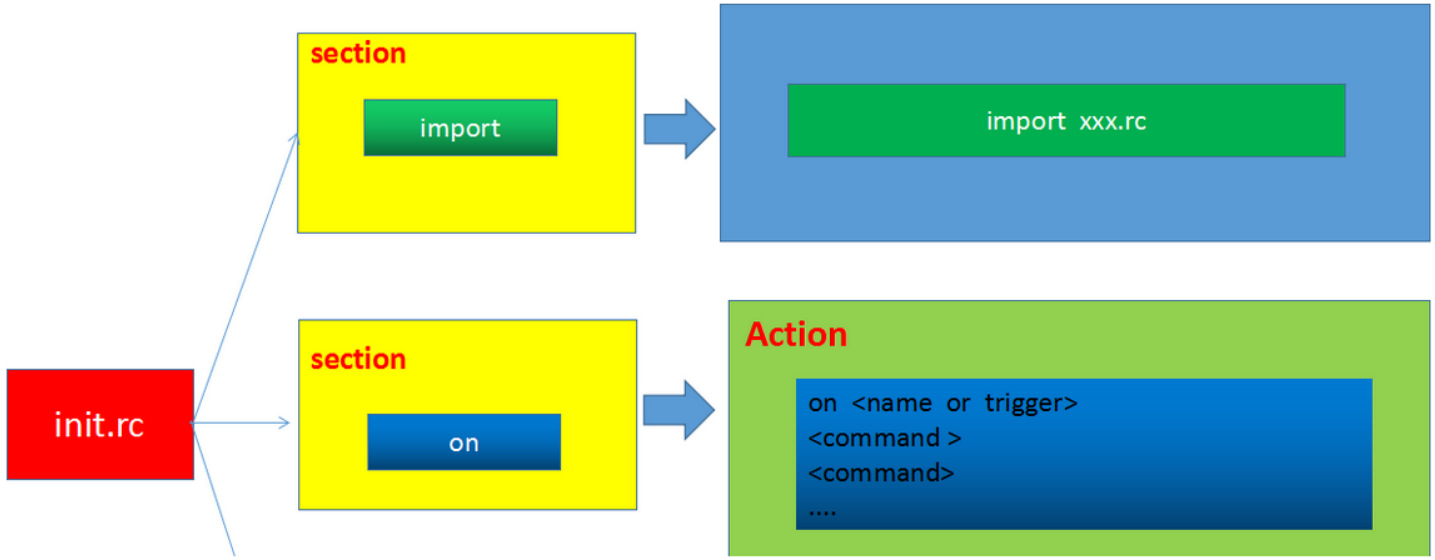
Android 系统本质上并不是名义上的操作系统，它的操作系统是Linux。从这个角度来看，我们可以认为Android 系统是基于Linux 操作系统开发的一些应用程序。这些应用程序在Linux 操作系统的应用层构建出了Android所需要的一切。例如 :Android 系统中用Java语言编写的APP应用程序运行环境。

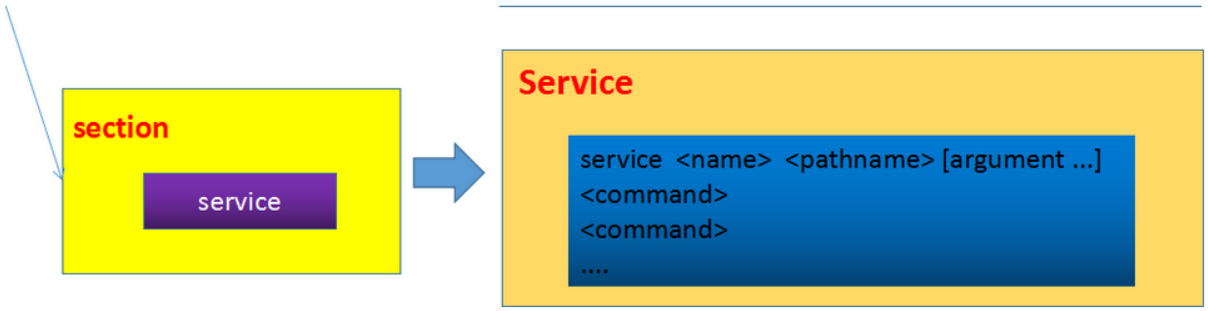
既然是应用程序,在这里我们就不讨论嵌入式Linux的启动流程了，这里我们重点讨论Android 系统中的第一个应用程序init。在Android系统中执行的init进程代码没有直接使用**busybox(制作嵌入式Linux根文件系统的工具，通过它制作出来的根文件系统包含Linux上常用的命令和init应用程序)**中提供的init进程代码，而是根据自己的需求重新实现了init进程相关的代码。

init进程在启动的过程中会去解析一个叫做init.rc文件，下面我们就来分析一下init.rc文件内容以及init进程解析完这个文件后会得到什么东西呢?

## 一、init.rc 文件解析

init.rc 是一个脚本文件，这个脚本文件中的语法是google公司制定的。这个文件的内容会被init进程解析，如果想看懂init进程相关的代码，我们就必须先熟悉init.rc这个文件的编写规则。






init.rc文件中内容我们可以将他们分成一段一段的来看，每一段以import 、on 、service开始。下面我们来分析一下这三个关键字的作用:

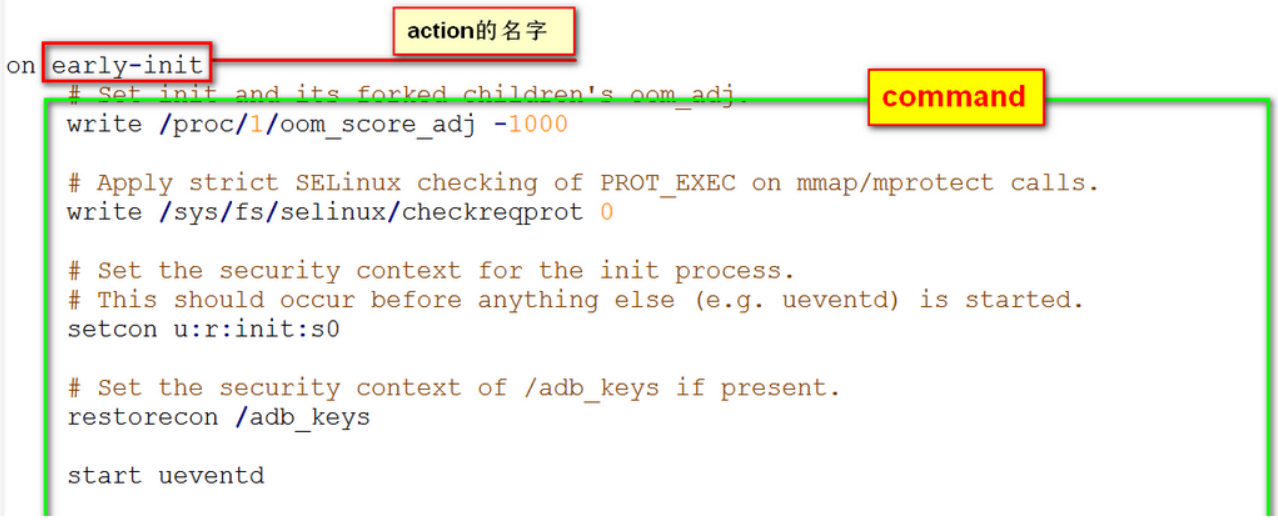
**(1)import**

用来导入一个xxx.rc文件的内容到init.rc文件中，类似于C语言中的include的作用。例如:

```
import /init.environ.rc
import /init.usb.rc
import /init.${ro.hardware}.rc
import /init.${ro.zygote}.rc
import /init.trace.rc
```

**(2)on**

以on开头的段落表示此段落的内容需要创建一个action来存放,和一个action相关联的是一系列的命令，当一个action的触发条件满足的时候和这个action相关联的命令就会被制作。例如:



```
# create mountpoints
mkdir /mnt 0775 root system
```

```
on property:vold.decrypt=trigger_default_encryption
    start defaultcrypto
```

属性action,当属性vold.decrypt的值被设为trigger_default_encryption时，这个action关联的命令就会被执行

我们来看看一个action可以关联哪些命令?

| Action Commands | |
|---|---|
| exec <path> [arg ...]<br><1>path 带路径的程序名<br><2>arg 程序执行的时候，需要的参数 | fork+exec 执行指定程序,init进程会阻塞等待程序执行完成 |
| export <name> <value><br><1>name 环境变量名<br><2>value 环境变量的值 | 设置init进程环境变量，完成后会被所有init启动后的新进程继承 |
| ifup <interface><br><1>interface 网卡设备名称 | 使能网卡 |
| hostname <name><br><1>name 主机名 | 设置主机名 |
| chmod <octal-mode> <path><br><1>octal-mode 8进制的数表示的权限<br><2>path 带路径的文件或目录 | 修改文件或目录的权限 |
| chown <owner> <group> <path><br><1>owner 所有者名<br><2>group 所属组名<br><3>path 带路径的文件或目录 | 修改文件或目录的所有者和组 |
| class_start <service class><br><1>service class服务类名称 | 启动某一类的服务 |
| class_stop <service class><br><1>service class服务类名称 | 停止某一类的服务 |
| domainname <name><br><1>name 域名名称 | 设置域名 |

| | |
|---|---|
| insmod <path><br><1>path 带路径的内核驱动模块 | 加载内核驱动模块 |
| mkdir <path> [mode owner group]<br><1>path 带路径的目录名<br><2>mode 文件权限<br><3>owen 所有者<br><4>group 所属组 | 新建目录,不指定mode owener group为0755 root root |
| mount <type> <device> <dir> [mountoption]<br><1>type 文件系统类型<br><2>device 文件系统所在的设备<br><3>dir 所需要挂载到的目录<br><4>mountoption 挂载时指定的参数 | 挂载文件系统 |
| setprop <name> <value><br><1>name 属性名<br><2>value 属性值 | 设置属性值 |
| setrlimit <resource> <cur> <max><br><1>resource 资源编号<br><2>cur 资源当前限制值<br><3>max 资源最大限制值 | 设置进程占用的资源限制。资源包括用户创建的文件大小、可打开的文<br>件个数等。 |
| start <service><br><1>service 服务名 | 启动服务 |
| stop <service><br><1>service 服务名 | 停止服务 |
| symlink <target> <path><br><1>target 软连接文件<br><2>path 带路径的源文件 | 建立软连接文件 |
| sysclktz <mins_west_of_gmt><br><1>mins_west_of_gmt 设置格林尼治时间差几分钟,一般为0 | 设置系统基准时间 |
| trigger <event><br><1>event 动作名称 | 触发一个动作 |
| write <path> <string> [string ...]<br><1>path 带路径的文件<br><2>string 写入的字符串 | 写字符串到文件中 |
| chdir <path> | 修改init进程的工作目录 |
| rm <path> | 删除文件 |
| rmdir <path> | 删除目录 |
| wait <path> | 等待文件被创建 |

| loglevel <level> | 设置内核的log级别 |
| --- | --- |

在init 进程中，每个action用**struct action**结构体描述，它的定义如下:

```
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;
    struct command *current;
};
```

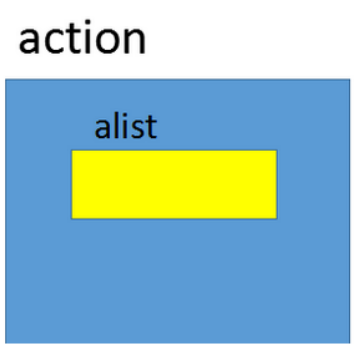
从上面的例子中我们可以知道每个action关联了一系列的命令，每个命令在init进程中，用**struct command**结构体描述， 它的定义如下:

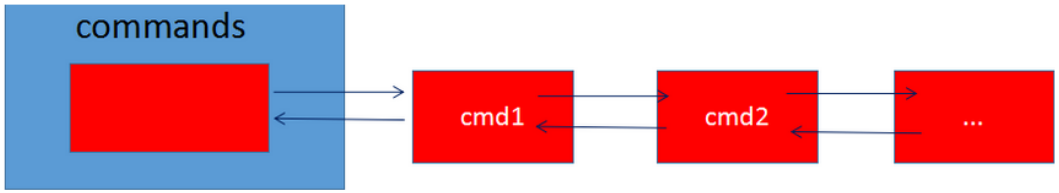
```
struct command
{
        /* list of commands in an action */
    struct listnode clist;

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

    int line;
    const char *filename;

    int nargs;
    char *args[1];
};
```

记录command对应的函数

记录给函数传递的参数个数和参数

和一个action关联的所有的命令会被添加到这个action关联的双向链表中，效果如下图所示:






**(2)service**

以service开头的段落表示此段落的内容需要创建一个service来存放,和一个service关联的是一系列的options,他们用来表示service 的属性信息以及init进程何时、如何运行一个service。例如:

(1)zygote 服务的名称
(2)/sys/bin/app_process 服务对应的应用程序
(3)-Xzygote /system/bin --zygote --start-system-server 应用程序执行的时候给应用程序传递的参数

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

```
service mdnsd /system/bin/mdnsd
    class main
    user mdnsr
    group inet net_raw
    socket mdnsd stream 0660 mdnsr inet
    disabled
    oneshot
```

options

嗯,还是先来看看一个service支持哪些option ?

| service option | |
|---|---|
| disabled | init进程启动的所有进程被包含在名称为"类"的运行组中。进程所属于的类被启动时,若指定进程为distabled,该进程将不会被执行，只能按照名称明确指定后才可以启动 |

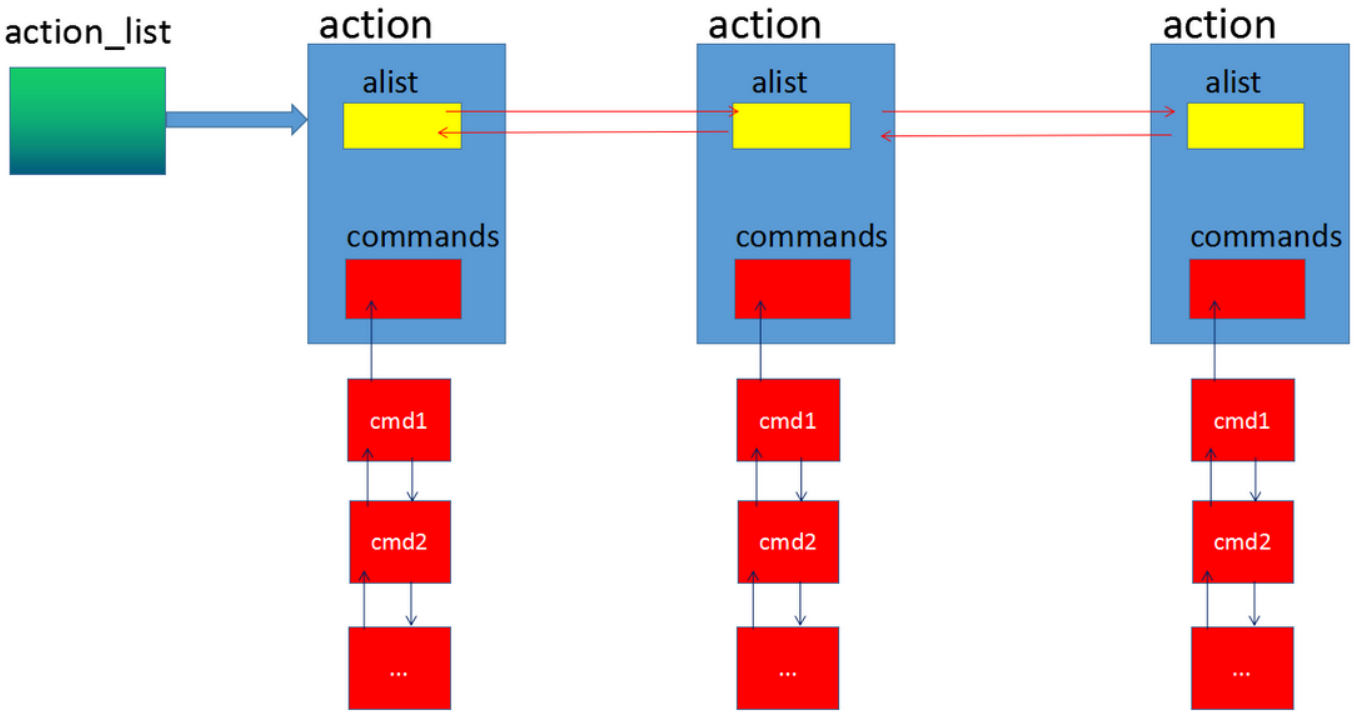
| | |
|---|---|
| socket <name> <type> <perm> [<user> [<group>]]<br>(1)name socket名<br>(2)type socket类型，只能是dgram,stream和 seqpacket<br>[1]dgram 无序报文传送，不保证可靠<br>[2]seqpacket 顺序报文传送，不保证可靠<br>[3]stream 流传送，保证可靠<br>(3)perm 指定访问权限(八进制的数表示)<br>(4)user 指定所属用户，默认root用户<br>(5)group 指定所属组，默认root组 | 创建unix域socket(即创建文件/dev/socket/<name>),并且打开，将文件描述符传递给服务，文件描述符存在ANDROID_SOCKET_<name>的环境变量中 |
| user <username> | 在执行服务前改变用户名。若未指定,则默认为root |
| group <groupname1> <groupname2> ... | 在执行服务前改变组，若未指定，则默认为root组 |
| oneshot | 服务退出后不再重启 |
| class <name> | 在执行服务前为其指定所属于的类。当一个类启动或退出时，其包含类的所有服务可以一同启动或停止。若未指定，服务默认为"default"类。 |
| onrestart | 当服务重启时执行一个命令 |

在init进程中service用struct service结构体描述,如下所示:

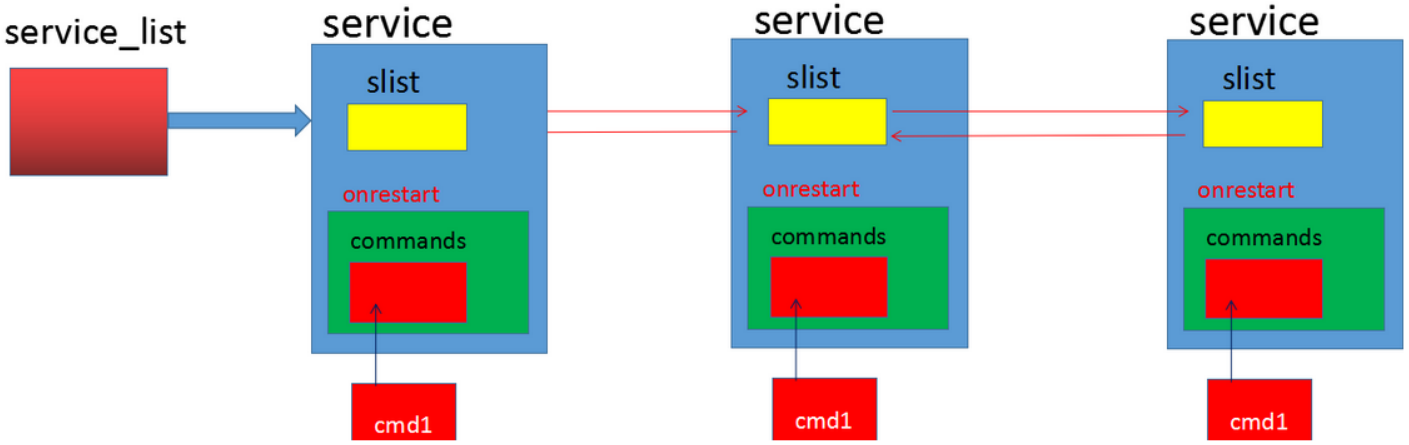
```
struct service {
        /* list of all services */
    struct listnode slist;
    const char *name;
    const char *classname;
    unsigned flags;
    pid_t pid;
    time_t time_started;    /* time of last start */
    time_t time_crashed;    /* first crash within inspection window */
    int nr_crashed;         /* number of times crashed within window */
    uid_t uid;
    gid_t gid;
    struct action onrestart;  /* Actions to execute on restart. */
    int nargs;
    /* "MUST BE AT THE END OF THE STRUCT" */
    char *args[1];
}; /*     ^-------'args' MUST be at the end of this struct! */
```

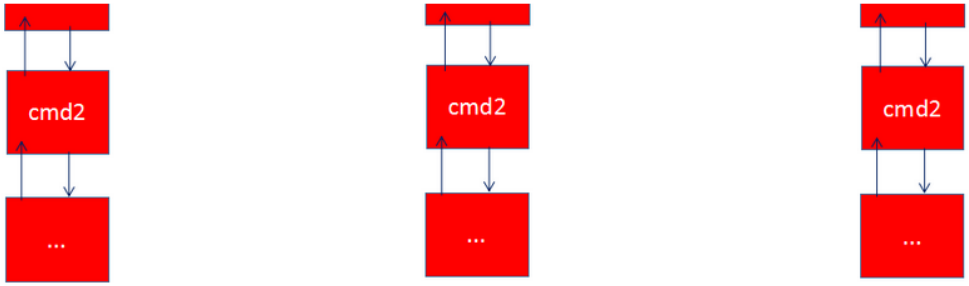
## 二、init进程解析完init.rc文件得到的产物

1、当init进程解析完一个action后就会将它添加到全局的action_list双向链表中，效果如下图所示:




2、init进程每解析一个service就将其插入service_list的双向链表中，效果如下:






## 三、init进程如何解析init.rc文件

关于解析的详细过程网上分析的代码很多，这里就不分析具体的解析代码了，简单的说一下:
init进程读取init.rc的内容后，**主要是通过一些关键单词来判别需要做什么**，这些关键单词我们可以在Android源码的system/core/init/keywords.h中查看,内容如下:

```
#define __MAKE_KEYWORD_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(disabled,    OPTION,  0, 0)
    KEYWORD(domainname,  COMMAND, 1, do_domainname)
    KEYWORD(enable,      COMMAND, 1, do_enable)
    KEYWORD(exec,        COMMAND, 1, do_exec)
    KEYWORD(export,      COMMAND, 2, do_export)
    KEYWORD(format_userdata,      COMMAND, 1, do_format_userdata)
    KEYWORD(group,       OPTION,  0, 0)
    KEYWORD(hostname,    COMMAND, 1, do_hostname)
    KEYWORD(ifup,        COMMAND, 1, do_ifup)
    KEYWORD(insmod,      COMMAND, 1, do_insmod)
    KEYWORD(import,      SECTION, 1, 0)
    KEYWORD(keycodes,    OPTION,  0, 0)
    KEYWORD(mkdir,       COMMAND, 1, do_mkdir)
    KEYWORD(mount_all,   COMMAND, 1, do_mount_all)
    KEYWORD(mount,       COMMAND, 3, do_mount)
    KEYWORD(on,          SECTION, 0, 0)
    KEYWORD(oneshot,     OPTION,  0, 0)
    KEYWORD(onrestart,   OPTION,  0, 0)
```

```
    KEYWORD(powerctl,     COMMAND, 1, do_powerctl)
    KEYWORD(restart,      COMMAND, 1, do_restart)
    KEYWORD(restorecon,   COMMAND, 1, do_restorecon)
    KEYWORD(restorecon_recursive,  COMMAND, 1, do_restorecon_recursive)
    KEYWORD(rm,           COMMAND, 1, do_rm)
    KEYWORD(rmdir,        COMMAND, 1, do_rmdir)
    KEYWORD(seclabel,     OPTION,  0, 0)
    KEYWORD(service,      SECTION, 0, 0)
    KEYWORD(setcon,       COMMAND, 1, do_setcon)
    KEYWORD(setenforce,   COMMAND, 1, do_setenforce)
    KEYWORD(setenv,       OPTION,  2, 0)
    KEYWORD(setkey,       COMMAND, 0, do_setkey)
    KEYWORD(setprop,      COMMAND, 2, do_setprop)
    KEYWORD(setrlimit,    COMMAND, 3, do_setrlimit)
    KEYWORD(setsebool,    COMMAND, 2, do_setsebool)
    KEYWORD(socket,       OPTION,  0, 0)
    KEYWORD(start,        COMMAND, 1, do_start)
    KEYWORD(stop,         COMMAND, 1, do_stop)
    KEYWORD(swapon_all,   COMMAND, 1, do_swapon_all)
    KEYWORD(trigger,      COMMAND, 1, do_trigger)
    KEYWORD(symlink,      COMMAND, 1, do_symlink)
    KEYWORD(sysclktz,     COMMAND, 1, do_sysclktz)
    KEYWORD(user,         OPTION,  0, 0)
    KEYWORD(wait,         COMMAND, 1, do_wait)
    KEYWORD(write,        COMMAND, 2, do_write)
    KEYWORD(copy,         COMMAND, 2, do_copy)
    KEYWORD(chown,        COMMAND, 2, do_chown)
    KEYWORD(chmod,        COMMAND, 2, do_chmod)
    KEYWORD(loglevel,     COMMAND, 1, do_loglevel)
    KEYWORD(load_persist_props,    COMMAND, 0, do_load_persist_props)
    KEYWORD(load_all_props,        COMMAND, 0, do_load_all_props)
    KEYWORD(ioprio,       OPTION,  0, 0)
#ifdef __MAKE_KEYWORD_ENUM__
    KEYWORD_COUNT,
};
#undef __MAKE_KEYWORD_ENUM__
#undef KEYWORD
#endif
```

在来看/system/core/init/init_parser.c中的一段代码，就更明白了:

```
 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:
         state->context = parse_service(state, nargs, args);
         if (state->context) {
             state->parse_line = parse_line_service;
             return;
         }
         break;
     case K_on:
         state->context = parse_action(state, nargs, args);
         if (state->context) {
```

创建service

创建action

```
            state->parse_line = parse_line_action;
            return;
        }
        break;
    case K_import:
        parse_import(state, nargs, args)
        break;
    }
    state->parse_line = parse_line_no_op;
}
```

处理import

参考文档:

(1)书籍:android框架揭秘

(2)博客:http://my.oschina.net/youranhongcha/blog/469028?fromerr=1D3mAWJt