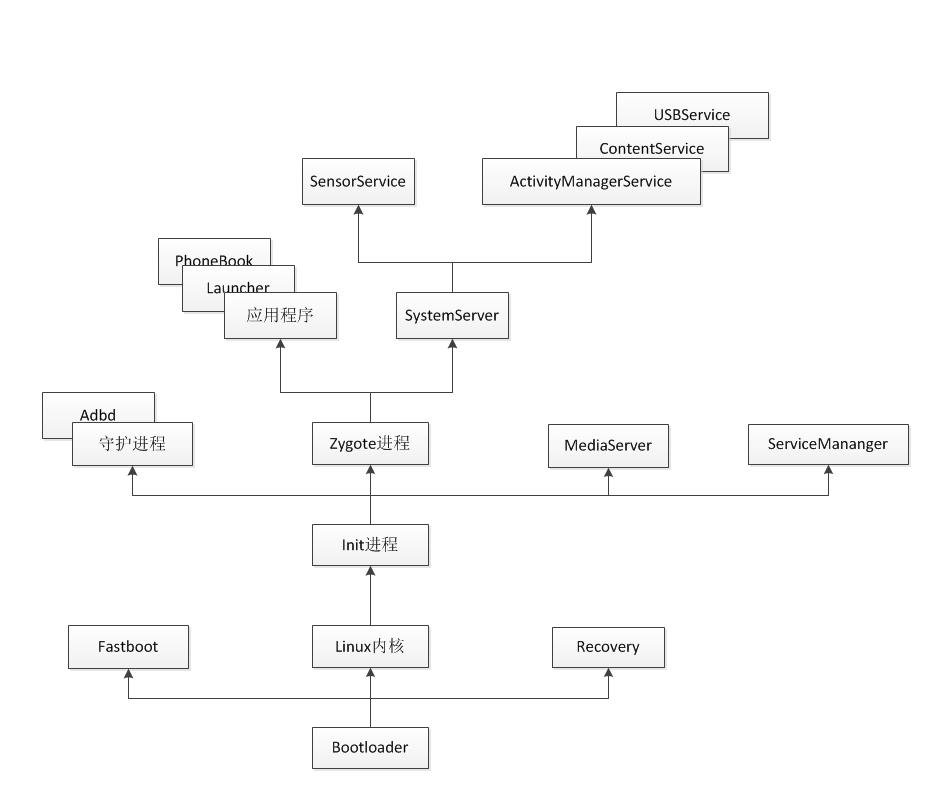
1. 深入理解init

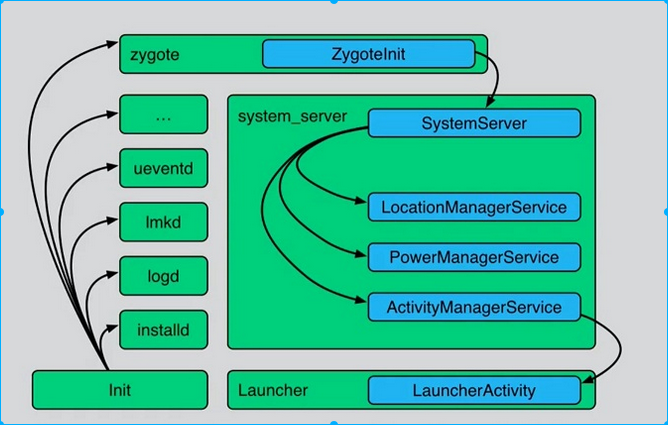
3.1 概述

init是一个进程，确切地说，它是Linux系统中用户空间的第一个进程。由于Android是基于Linux内核的，所以init也是Android系统中用户空间的第一个进程，它的进程号是1。作为天字第一号的进程，init被赋予了很多极其重要的工作职责，本章将关注其中两个比较重要的职责：

· init进程负责创建系统中的几个关键进程，尤其是下一章要介绍的Zygote，它更是Java世界的开创者。那么，init进程是如何创建Zygote的呢？

· Android系统有很多属性，于是init就提供了一个property service（属性服务）来管理它们。那么这个属性服务是怎么工作的呢？





init进程会启动很多守护进程,这些守护进程都是c/c++实现的,后续文章都会对这些守护进程一一介绍.zygote进程前面已经介绍过了.

如上所述，本章将通过下面两方面内容来分析init：

· init如何创建zygote。

· init的属性服务是如何工作的。

3.2 init分析

init进程的入口函数是main，它的代码如下所示：

[-->init.c]

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;

......

//创建一些文件夹，并挂载设备，这些是和Linux相关的，不拟做过多讨论。

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);

//重定向标准输入/输出/错误输出到/dev/\_null\_。

open\_devnull\_stdio();

/\*

设置init的日志输出设备为/dev/\_\_kmsg\_\_，不过该文件打开后，会立即被unlink了,

这样，其他进程就无法打开这个文件读取日志信息了。

\*/

klog\_init();

//上面涉及很多和Linux系统相关的知识，不熟悉的读者可自行研究，它们不影响我们的分析

//初始化和属性相关的资源

property\_init();

//下面这个函数通过读取/proc/cpuinfo得到机器的Hardware名

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

process\_kernel\_cmdline();

INFO("property init\n");

//加载默认属性

property\_load\_boot\_defaults();

#ifndef INIT\_ENG\_BUILD

//调用property\_set函数设置属性项，一个属性项包括属性名和属性值。

property\_set("ro.mtprof.disable", "1");

#endif

//负责初始化和分析init.rc文件

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

/\*

解析完上述配置文件后，会得到一系列的Action（动作），下面两句代码将执行那些处于

early-init阶段的Action。init将动作执行的时间划分为四个阶段：early-init、init、

early-boot、boot。由于有些动作必须在其他动作完成后才能执行，所以就有了先后之分。哪些

动作属于哪个阶段由配置文件决定。后面会介绍配置文件的相关知识。

\*/

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

queue\_builtin\_action(wait\_for\_coldboot\_done\_action, "wait\_for\_coldboot\_done");

queue\_builtin\_action(mix\_hwrng\_into\_linux\_rng\_action, "mix\_hwrng\_into\_linux\_rng");

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

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

/\* execute all the boot actions to get us started \*/

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

/\* Repeat mix\_hwrng\_into\_linux\_rng in case /dev/hw\_random or /dev/random

\* wasn't ready immediately after wait\_for\_coldboot\_done

\*/

queue\_builtin\_action(mix\_hwrng\_into\_linux\_rng\_action, "mix\_hwrng\_into\_linux\_rng");

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

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

/\* Don't mount filesystems or start core system services if in charger mode. \*/

if (is\_charger) {

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

}

else

{

action\_for\_each\_trigger("late-init", 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");

/\*

Boot chart是一个小工具，它能对系统的性能进行分析，并生成系统启动过程的图表，

以提供一些有价值的信息，而这些信息最大的用处就是帮助提升系统的启动速度。

\*/

#if BOOTCHART

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

#endif

for(;;) {

int nr, i, timeout = -1;

execute\_one\_command();

restart\_processes(); // 重启那些已经死去的进程

//处理属性服务的事件

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;

}

//处理keychord事件。

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;

#if BOOTCHART //性能分析小工具

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

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

if (nr <= 0)

continue;

// 处理具体的消息

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;

}

从上面的代码中可知，init的工作任务还是很重的。上面的代码虽已省略了不少行，可结果还是很长，不过从本章要分析的两个知识点来看，可将init的工作流程精简为以下四点：

· 解析两个配置文件，其中，将分析对init.rc文件的解析。

· 执行各个阶段的动作，创建Zygote的工作就是在其中的某个阶段完成的。

· 调用property\_init初始化属性相关的资源，并且通过property\_start\_service启动属性服务。

· init进入一个无限循环，并且等待一些事情的发生。重点关注init如何处理来自socket和来自属性服务器相关的事情。

精简工作流程，是以后分析代码时常用的方法。读者在分析代码的过程中，也可使用这种方法。

3.2.1 解析配置文件

根据上面的代码可知，在init中会解析两个配置文件，其中一个是系统配置文件init.rc，另外一个是和硬件平台相关的配置文件。以HTC G7手机为例，这个配置文件名为init.bravo.rc，其中bravo是硬件平台的名称。对这两个配置文件进行解析，调用的是同一个parse\_config\_file函数。下面就来看这个函数，在分析过程中以init.rc为主。

[-->init\_parser.c]

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

{

char \*data;

data = read\_file(fn, 0); //读取配置文件的内容，这个文件是init.rc。

if (!data) return -1;

parse\_config(fn, data);//调用parse\_config做真正的解析

DUMP();

return 0;

}

读取完文件的内容后，将调用parse\_config进行解析，这个函数的代码如下所示：

[-->init\_parser.c]

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;//设置解析函数，不同的内容用不同的解析函数

list\_init(&import\_list);

state.priv = &import\_list;

for (;;) {

switch (next\_token(&state)) {

case T\_EOF:

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

goto parser\_done;

case T\_NEWLINE:

state.line++;

if (nargs) {

int kw = lookup\_keyword(args[0]); //得到关键字的类型

if (kw\_is(kw, SECTION)) { //判断关键字类型是不是SECTION。

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

parse\_new\_section(&state, kw, nargs, args); //解析这个SECTION。

} else {

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

}

nargs = 0;

}

break;

case T\_TEXT:

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);

}

}

上面就是parse\_config函数，代码虽短，实际却比较复杂。从整体来说，parse\_config首先会找到配置文件的一个section，然后针对不同的 section使用不同的解析函数来解析。那么，什么是section呢？这和init.rc文件的组织结构有关。先不必急着去看init.rc，还是先到代码中去寻找答案。

1. 关键字定义

keywords.h这个文件定义了init中使用的关键字，它的用法很有意思，先来看这个文件，代码如下所示：

[-->keywords.h]

#ifndef KEYWORD //如果没有定义KEYWORD宏，则走下面的分支

......

//声明一些函数，这些函数就是前面所说Action的执行函数。

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

int do\_class\_stop(int nargs, char \*\*args);

......

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

......

#define \_\_MAKE\_KEYWORD\_ENUM\_\_ //定义一个宏

/\*

定义KEYWORD宏，虽然有四个参数，不过这里只用第一个，其中K\_##symbol中的##表示连接

的意思，即最后得到的值为K\_symbol。symbol其实就是init.rc中的关键字

\*/

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

enum { //定义一个枚举，这个枚举定义了各个关键字的枚举值。

K\_UNKNOWN,

#endif

......

//根据上面KEYWORD的定义，这里将得到一个枚举值K\_class,

KEYWORD(class, OPTION, 0, 0)

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

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

KEYWORD(on, SECTION, 0, 0)//K\_on，

KEYWORD(oneshot, OPTION, 0, 0)

KEYWORD(onrestart, OPTION, 0, 0)

KEYWORD(restart, COMMAND, 1,do\_restart)

KEYWORD(service, SECTION, 0,0)

......

KEYWORD(socket, OPTION, 0, 0)

KEYWORD(start, COMMAND, 1,do\_start)

KEYWORD(stop, COMMAND, 1,do\_stop)

......

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

KEYWORD\_COUNT,

};

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

#undef KEYWORD //取消KEYWORD宏定义

#endif

keywords.h好像没什么奇特，不过是个简单的头文件。为什么说它的用法很有意思呢？来看代码中是如何使用它的，如下所示：

[-->init\_parser.c]

#include "keywords.h"

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

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

static 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"

};

#undef KEYWORD

......

//init\_parser.c中将包含keywords.h头文件，而且还不只一次！！

//第一次包含keywords.h，根据keywords.h的代码，我们首先会得到一个枚举定义

#include "keywords.h"

宏替换如下：

......

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

enum {

K\_UNKNOWN,

K\_capability,

K\_chdir,

K\_chroot,

......

}

/\*

重新定义KEYWORD宏，这回四个参数全用上了，看起来好像是一个结构体。其中#symbol表示

一个字符串，其值为“symbol”。

\*/

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

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

//定义一个结构体keyword\_info数组，它用来描述关键字的一些属性，请注意里面的注释内容。

struct {

const char \*name; //关键字的名。

int(\*func)(int nargs, char \*\*args);//对应关键字的处理函数。

unsigned char nargs;//参数个数，每个关键字的参数个数是固定的。

//关键字的属性，有三种属性，COMMAND、OPTION和SECTION。其中COMMAND有对应的处理函数

unsigned char flags;

} keyword\_info[KEYWORD\_COUNT] = {

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

/\*

第二次包含keywords.h，由于已经重新定了KEYWORD宏，所以以前那些作为枚举值的关键字

现在变成keyword\_info数组的索引了。

\*/

#include "keywords.h"

宏替换如下：

[K\_capability]=(capability, 0, 0+1, OPTION)

};

#undef KEYWORD

//一些辅助宏，帮助我们快速操作keyword\_info中的内容。

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

#define kw\_name(kw) (keyword\_info[kw].name)

#define kw\_func(kw) (keyword\_info[kw].func)

#define kw\_nargs(kw) (keyword\_info[kw].nargs)

现在领略了keywords.h的神奇之处了吧？原来它干了两件事情：

· 第一次包含keyworks.h时，它声明了一些诸如do\_classstart这样的函数，另外还定义了一个枚举，枚举值为K\_class，K\_mkdir等关键字。

· 第二次包含keywords.h后，得到了一个keyword\_info结构体数组，这个keyword\_info结构体数组以前面定义的枚举值为索引，存储对应的关键字信息，这些信息包括关键字名、处理函数、处理函数的参数个数，以及属性。

目前，关键字信息中最重要的就是symbol和flags了。什么样的关键字被认为是section呢？根据keywords.h的定义，symbol为下面3个的关键字表示section：

KEYWORD(on, SECTION, 0, 0)

KEYWORD(service, SECTION, 0, 0)

KEYWORD(import, SECTION, 1, 0)

有了上面的知识，再来看配置文件init.rc的内容。

2. init.rc的解析

init.rc的内容如下所示：（我们截取了部分内容，注意，其中的注释符号是#。）

[-->init.rc]

# set up the global environment

on init

#根据上面的分析，on关键字标示一个section，对应的名字是”init”

......

#下面所有的内容都属于这个section，直到下一个section开始时。

export PATH /sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin

export ANDROID\_BOOTLOGO 1 #根据keywords.h的定义，export表示一个COMMAND

export ANDROID\_ROOT /system

export ANDROID\_ASSETS /system/app

export ANDROID\_DATA /data

export ANDROID\_STORAGE /storage

export ASEC\_MOUNTPOINT /mnt/asec

export LOOP\_MOUNTPOINT /mnt/obb

export BOOTCLASSPATH %BOOTCLASSPATH%

export SYSTEMSERVERCLASSPATH %SYSTEMSERVERCLASSPATH%

......

#省略部分内容

on boot #这是一个新的section，名为”boot”

# basic network init

ifup lo #这是一个COMMAND

hostname localhost

domainname localdomain

# set RLIMIT\_NICE to allow priorities from 19 to -20

setrlimit 13 40 40

......

#class\_start也是一个COMMAND，对应函数为do\_class\_start，很重要，切记

class\_start core

......

#下面这个section的意思是：待属性persist.service.adb.enable的值变为1后，

#需要执行对应的COMMAND，这个COMMAND是start adbd

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

start adbd //start是一个COMMAND

on property:persist.service.adb.enable=0

stop adbd

#service也是section的标示，对应section的名为“zygote“

service zygote /system/bin/app\_process -Xzygote /system/bin --zygote --start-system-server

class main

socket zygote stream 660 root system #socket关键字表示OPTION

onrestart write /sys/android\_power/request\_state wake

onrestart write /sys/power/state on #onrestart也是OPTION

onrestart restart media

onrestart restart netd

#一个section，名为”media”

service media /system/bin/mediaserver

class main

user root ####

group audio camera inet net\_bt net\_bt\_admin net\_bw\_acct drmrpc mediadrm media sdcard\_r system net\_bt\_stack ####

从上面对init.rc的分析中可知：

· 一个section的内容从这个标示section的关键字开始，到下一个标示section的地方结束。

· init.rc中出现了名为boot和init的section，这里的boot和init，就是前面介绍的动作执行四个阶段中的boot和init。也就是说，在boot阶段执行的动作都是由boot这个section定义的。

另外还可发现，zygote被放在了一个service section中。下面以zygote这个section为例，介绍service是如何解析的。

3.2.2 解析service

zygote对应的service section内容是：

[-->init.rc::zygote]

service zygote /system/bin/app\_process -Xzygote /system/bin --zygote --start-system-server

class main

socket zygote stream 660 root system #socket关键字表示OPTION

#下面的onrestart是OPTION，而write和restart是COMMAND

onrestart write /sys/android\_power/request\_state wake

onrestart write /sys/power/state on #onrestart也是OPTION

onrestart restart media

onrestart restart netd

解析section的入口函数是parse\_new\_section，它的代码如下所示：

[-->init\_parser.c]

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: //解析on section

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

if (state->context) {

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

return;

}

break;

case K\_import:

parse\_import(state, nargs, args);

break;

}

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

}

其中，service解析时，用到了parse\_service和parse\_line\_service两个函数，在分别介绍它们之前，先看init是如何组织这个service的。

1. service结构体

init中使用了一个叫service的结构体来保存和service section相关的信息，不妨来看这个结构体，代码如下所示：

[-->init.h::service结构体定义]

struct service {

//listnode是一个特殊的结构体，在内核代码中用得非常多，主要用来将结构体链接成一个

//双向链表。init中有一个全局的service\_list，专门用来保存解析配置文件后得到的service。

struct listnode slist;

const char \*name; //service的名字，对应我们这个例子就是”zygote”。

const char \*classname; //service所属class的名字，默认是”defult”

unsigned flags; //service的属性

pid\_t pid; //进程号

time\_t time\_started; //上一次启动的时间

time\_t time\_crashed; //上一次死亡的时间

int nr\_crashed; //死亡次数

uid\_t uid; //uid,gid相关

gid\_t gid;

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

size\_t nr\_supp\_gids;

/\*

有些service需要使用socket，下面这个socketinfo用来描述socket的相关信息。

我们的zygote也使用了socket，配置文件中的内容是socket zygote stream 660。

它表示将创建一个AF\_STREAM类型的socket（其实就是TCP socket），该socket的名为“zygote”，

读写权限是660。

\*/

struct socketinfo \*sockets;

//service一般运行在单独的一个进程中envvars用来描述创建这个进程时所需的环境变量信息。

struct svcenvinfo \*envvars;

/\*

虽然关键字onrestart标示一个OPTION，可是这个OPTION后面一般跟着COMMAND，

下面这个action结构体可用来存储command信息，马上就会分析到它。

\*/

struct action onrestart;

//和keychord相关的内容

int \*keycodes;

int nkeycodes;

int keychord\_id;

//io优先级设置

int ioprio\_class;

int ioprio\_pri;

int nargs;//参数个数

char \*args[1];//用于存储参数

};

我们现在已了解的service的结构体，相对来说还算是清晰易懂的。而zygote中的那三个onrestart该怎么表示呢？请看service中使用的这个action结构体：

[-->init.h::action结构体定义]

struct action {

/\*

一个action结构体可存放在三个双向链表中，其中alist用于存储所有action，

qlist用于链接那些等待执行的action，tlist用于链接那些待某些条件满足后

就需要执行的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;

//这个OPTION对应的COMMAND链表，以zygote为例，它有三个onrestart option，所以

//它对应会创建三个command结构体。

struct listnode commands;

struct command \*current;

};

了解了上面的知识后，你是否能猜到parse\_service和parse\_line\_service的作用了呢？马上就来看它们。

2. parse\_service

parse\_service的代码如下所示：

[-->init\_parser.c]

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

{

struct service \*svc;//声明一个service结构体

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;

}

//init维护了一个全局的service链表，先判断是否已经有同名的service了。

svc = service\_find\_by\_name(args[1]);

if (svc) {

//如果有同名的service，则不能继续后面的操作。

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

return 0;

}

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"; //设置classname为”default”，这个很关键！

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); //把zygote这个service加到全局链表service\_list中。

return svc;

}

parse\_service函数只是搭建了一个service的架子，具体的内容尚需由后面的解析函数来填充。来看service的另外一个解析函数parse\_line\_service。

3. parse\_line\_service

parse\_line\_service的代码如下所示：

[-->init\_parser.c]

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;

......

svc->ioprio\_class = IoSchedClass\_NONE;

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

switch (kw) {

case K\_capability:

break;

case K\_class:

......

break;

case K\_console:

svc->flags |= SVC\_CONSOLE;

break;

......

case K\_oneshot:

/\*

这是service的属性，它一共有五个属性，分别为：

SVC\_DISABLED：不随class自动启动。下面将会看到class的作用。

SVC\_ONESHOT：退出后不需要重启，也就是这个service只启动一次就可以了。

SVC\_RUNNING：正在运行，这是service的状态。

SVC\_RESTARTING：等待重启，这也是service的状态。

SVC\_CONSOLE：该service需要使用控制台 。

SVC\_CRITICAL：如果在规定时间内该service不断重启，则系统会重启并进入恢复模式。

zygote没有使用任何属性，这表明它：会随着class的处理自动启动；

退出后会由init重启；不使用控制台；即使不断重启也不会导致系统进入恢复模式。

\*/

svc->flags |= SVC\_ONESHOT;

break;

case K\_onrestart: //根据onrestart的内容，填充action结构体的内容

nargs--;

args++;

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

......

//创建command结构体

cmd = malloc(sizeof(\*cmd) + sizeof(char\*) \* nargs);

cmd->func = kw\_func(kw);

cmd->nargs = nargs;

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

//把新建的command加入到双向链表中。

list\_add\_tail(&svc->onrestart.commands, &cmd->clist);

break;

......

case K\_socket: {//创建socket相关信息

struct socketinfo \*si;

......

si = calloc(1, sizeof(\*si));

if (!si) {

parse\_error(state, "out of memory\n");

break;

}

si->name = args[1]; //socket的名字

si->type = args[2]; //socket的类型

si->perm = strtoul(args[3], 0, 8); //socket的读写权限

if (nargs > 4)

si->uid = decode\_uid(args[4]);

if (nargs > 5)

si->gid = decode\_uid(args[5]);

if (nargs > 6)

si->socketcon = args[6];

si->next = svc->sockets;

svc->sockets = si;

break;

}

default:

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

}

}

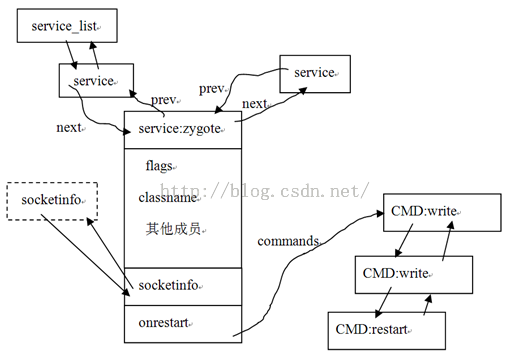


图3-1 zygote解析结果示意图

从上图中可知：

· service\_list链表将解析后的service全部链接到了一起，并且是一个双向链表，前向节点用prev表示，后向节点用next表示。

· socketinfo也是一个双向链表，因为zygote只有一个socket，所以画了一个虚框socket做为链表的示范。

· onrestart通过commands指向一个commands链表，zygote有三个commands。

zygote这个service解析完了，现在就是“万事俱备，只欠东风”了。接下来要了解的是，init是如何控制service的。

3.2.3 init控制service

先看service是如何启动的。

1．启动zygote

init.rc中有这样一句话：

#class\_start是一个COMMAND，对应的函数为do\_class\_start，很重要，切记。

class\_start default

class\_start标示一个COMMAND，对应的处理函数为do\_class\_start，它位于boot section的范围内。为什么说它很重要呢？

还记得init进程中的四个执行阶段吗？当init进程执行到下面几句话时，do\_class\_start就会被执行了。

//将boot section节的command加入到执行队列

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

//执行队列里的命令，class可是一个COMMAND，所以它对应的do\_class\_start会被执行。

[-->builtins.c]

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

{

/\*

args为do\_class\_start的参数，init.rc中只有一个参数，就是default。

下面这个函数将从service\_list中寻找classname为”default”的service，然后

调用service\_start\_if\_not\_disabled函数。现在读者明白了service结构体中

classname的作用了吗？

\*/

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

return 0;

}

我们已经知道，zygote这个service的classname的值就是“default”，所以会针对这个service调用service\_start\_if\_not\_disabled，这个函数的代码是：

[-->init\_parser.c]

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;

}

}

service\_start函数的代码如下所示：

[-->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

\*/

svc->flags &= (~(SVC\_DISABLED|SVC\_RESTARTING|SVC\_RESET|SVC\_RESTART|SVC\_DISABLED\_START));

svc->time\_started = 0;

/\* running processes require no additional work -- if

\* they're in the process of exiting, we've ensured

\* that they will immediately restart on exit, unless

\* they are ONESHOT

\*/

if (svc->flags & SVC\_RUNNING) {

return; //如果这个service已在运行，则不用处理

}

/\*

service一般运行于另外一个进程中，这个进程也是init的子进程，所以启动service前需要判断

对应的可执行文件是否存在，zygote对应的可执行文件是/system/bin/app\_process

\*/

needs\_console = (svc->flags & SVC\_CONSOLE) ? 1 : 0;

if (needs\_console && (!have\_console)) {

ERROR("service '%s' requires console\n", svc->name);

svc->flags |= SVC\_DISABLED;

return;

}

......

pid = fork(); //调用fork创建子进程

if (pid == 0) { //pid为零，我们在子进程中

struct socketinfo \*si;

struct svcenvinfo \*ei;

char tmp[32];

int fd, sz;

umask(077);

if (properties\_inited()) {

//得到属性存储空间的信息并加到环境变量中后面在属性服务节中会碰到使用它的地方

get\_property\_workspace(&fd, &sz);

sprintf(tmp, "%d,%d", dup(fd), sz);

add\_environment("ANDROID\_PROPERTY\_WORKSPACE", tmp);

}

//添加环境变量信息

for (ei = svc->envvars; ei; ei = ei->next)

add\_environment(ei->name, ei->value);

//根据socketinfo创建socket

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));

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

si->perm, si->uid, si->gid, si->socketcon ?: scon);

if (s >= 0) {

//在环境变量中添加socket信息。

publish\_socket(si->name, s);

}

}

freecon(scon);

scon = NULL;

if (svc->ioprio\_class != IoSchedClass\_NONE) {

if (android\_set\_ioprio(getpid(), svc->ioprio\_class, svc->ioprio\_pri)) {

ERROR("Failed to set pid %d ioprio = %d,%d: %s\n",

getpid(), svc->ioprio\_class, svc->ioprio\_pri, strerror(errno));

}

}

if (needs\_console) {

setsid();

open\_console();

} else {

zap\_stdio();

}

......//设置uid，gid等

setpgid(0, getpid());

/\* as requested, set our gid, supplemental gids, and uid \*/

if (svc->gid) {

if (setgid(svc->gid) != 0) {

ERROR("setgid failed: %s service name %s\n", strerror(errno),svc->name);

\_exit(127);

}

}

if (svc->nr\_supp\_gids) {

if (setgroups(svc->nr\_supp\_gids, svc->supp\_gids) != 0) {

ERROR("setgroups failed: %s service name %s\n", strerror(errno),svc->name);

\_exit(127);

}

}

if (svc->uid) {

if (setuid(svc->uid) != 0) {

ERROR("setuid failed: %s service name %s\n", strerror(errno),svc->name);

\_exit(127);

}

}

if (svc->seclabel) {

if (is\_selinux\_enabled() > 0 && setexeccon(svc->seclabel) < 0) {

ERROR("cannot setexeccon('%s'): %s\n", svc->seclabel, strerror(errno));

\_exit(127);

}

}

if (!dynamic\_args) {

/\*

执行/system/bin/app\_process，这样就进入到app\_process的main函数中了。

fork、execve这两个函数都是Linux系统上常用的系统调用。

\*/

if (execve(svc->args[0], (char\*\*) svc->args, (char\*\*) ENV) < 0) {

ERROR("cannot execve('%s'): %s\n", svc->args[0], strerror(errno));

}

} else {

char \*arg\_ptrs[INIT\_PARSER\_MAXARGS+1];

int arg\_idx = svc->nargs;

char \*tmp = strdup(dynamic\_args);

char \*next = tmp;

char \*bword;

/\* Copy the static arguments \*/

memcpy(arg\_ptrs, svc->args, (svc->nargs \* sizeof(char \*)));

while((bword = strsep(&next, " "))) {

arg\_ptrs[arg\_idx++] = bword;

if (arg\_idx == INIT\_PARSER\_MAXARGS)

break;

}

arg\_ptrs[arg\_idx] = '\0';

execve(svc->args[0], (char\*\*) arg\_ptrs, (char\*\*) ENV);

}

\_exit(127);

}

freecon(scon);

if (pid < 0) {

ERROR("failed to start '%s'\n", svc->name);

svc->pid = 0;

return;

}

......//父进程init的处理，设置service的信息，如启动时间、进程号，以及状态等。

svc->time\_started = gettime();

svc->pid = pid;

svc->flags |= SVC\_RUNNING;

if (properties\_inited())

//每一个service都有一个属性，zygote的属性为init.svc.zygote，现在设置它的值为running

notify\_service\_state(svc->name, "running");

}

原来，zygote是通过fork和execv共同创建的！但service结构中的那个onrestart好像没有派上用场，原因何在？

2. 重启zygote

根据名字，就可猜到onrestart应该是在zygote重启时用的。下面先看在zygote死后，它的父进程init会有什么动作：

[-->sigchld\_handler.c]

static void sigchld\_handler(int s)

{ //当子进程退出时，init的这个信号处理函数会被调用

write(signal\_fd, &s, 1); //往signal\_fd write数据

}

signal\_fd，就是在init中通过socketpair创建的两个socket中的一个，既然会往这个signal\_fd中发送数据，那么另外一个socket就一定能接收到，这样就会导致init从poll函数中返回：

[-->init.c::main函数代码片断]

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(); //注意这个信号处理函数

}

}

[-->sigchld\_handler.c]

void handle\_signal(void)

{

char tmp[32];

/\* we got a SIGCHLD - reap and restart as needed \*/

read(signal\_recv\_fd, tmp, sizeof(tmp));

while (!wait\_for\_one\_process(0)) //调用wait\_for\_one\_process函数处理

;

}

//直接看这个wait\_for\_one\_process函数：

static int wait\_for\_one\_process(int block)

{

pid\_t pid;

int status;

struct service \*svc;

struct socketinfo \*si;

time\_t now;

struct listnode \*node;

struct command \*cmd;

while ( (pid = waitpid(-1, &status, block ? 0 : WNOHANG)) == -1 && errno == EINTR );

if (pid <= 0) return -1;

INFO("waitpid returned pid %d, status = %08x\n", pid, status);

//找到死掉的那个service，现在应该找到了代表zygote的那个service。

svc = service\_find\_by\_pid(pid);

......

if (!(svc->flags & SVC\_ONESHOT) || (svc->flags & SVC\_RESTART)) {

//杀掉zygote创建的所有子进程，这就是zygote死后，Java世界崩溃的原因。

kill(-pid, SIGKILL);

NOTICE("process '%s' killing any children in process group\n", svc->name);

}

//清理socket信息，不清楚的读者可以通过命令man 7 AF\_UNIX查询一下相关知识。

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

char tmp[128];

snprintf(tmp, sizeof(tmp), ANDROID\_SOCKET\_DIR"/%s", si->name);

unlink(tmp);

}

svc->pid = 0;

svc->flags &= (~SVC\_RUNNING);

/\* oneshot processes go into the disabled state on exit,

\* except when manually restarted. \*/

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

svc->flags |= SVC\_DISABLED;

}

/\* disabled and reset processes do not get restarted automatically \*/

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

notify\_service\_state(svc->name, "stopped");

return 0;

}

now = gettime();

/\*

如果设置了SVC\_CRITICAL标示，则4分钟内该服务重启次数不能超过4次，否则

机器会重启进入recovery模式。根据init.rc的配置，只有servicemanager进程

享有此种待遇。

\*/

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

if (svc->time\_crashed + CRITICAL\_CRASH\_WINDOW >= now) {

if (++svc->nr\_crashed > CRITICAL\_CRASH\_THRESHOLD) {

ERROR("critical process '%s' exited %d times in %d minutes; "

"rebooting into recovery mode\n", svc->name,

CRITICAL\_CRASH\_THRESHOLD, CRITICAL\_CRASH\_WINDOW / 60);

android\_reboot(ANDROID\_RB\_RESTART2, 0, "recovery");

return 0;

}

} else {

svc->time\_crashed = now;

svc->nr\_crashed = 1;

}

}

//设置标示为SVC\_RESTARTING,然后执行该service onrestart中的COMMAND，这些内容就

//非常简单了，读者可以自行学习。

svc->flags &= (~SVC\_RESTART);

svc->flags |= SVC\_RESTARTING;

/\* Execute all onrestart commands for this service. \*/

list\_for\_each(node, &svc->onrestart.commands) {

cmd = node\_to\_item(node, struct command, clist);

cmd->func(cmd->nargs, cmd->args);

}

//设置init.svc.zygote的值为restarting。

notify\_service\_state(svc->name, "restarting");

return 0;

}

通过上面的代码，可知道onrestart的作用了，但zygote本身又在哪里重启的呢？答案就在下面的代码中：

[-->init.c::main函数代码片断]

for(;;) {

int nr, i, timeout = -1;

execute\_one\_command();

restart\_processes(); //这里会重启所有flag标志为SVC\_RESTARTING的service。

这样，zygote又回来了！

3.2.4 属性服务

我们知道，Windows平台上有一个叫注册表的东西。注册表可以存储一些类似key/value的键值对。一般而言，系统或某些应用程序会把自己的一些属性存储在注册表中，即使下次系统重启或应用程序重启，它还能够根据之前在注册表中设置的属性，进行相应的初始化工作。Android平台也提供了一个类型机制，可称之为属性服务（property service）。应用程序可通过这个属性机制，查询或设置属性。读者可以用adb shell登录到真机或模拟器上，然后用getprop命令查看当前系统中有哪些属性。即如我的HTC G7测试结果，如图3-2所示：（图中只显示了部分属性）

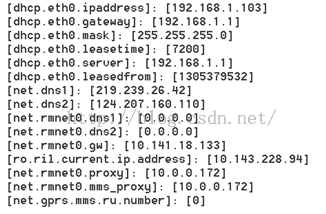


图3-2 HTC G7属性示意图

这个属性服务是怎么实现的呢？下面来看代码，其中与init.c和属性服务有关的代码有下面两行：

property\_init();

property\_set\_fd = start\_property\_service();

分别来看看它们。

1. 属性服务初始化

（1）创建存储空间

先看property\_init函数，代码如下所示：

[-->property\_service.c]

void property\_init(void)

{

init\_property\_area(); //初始化属性存储区域

}

在properyty\_init函数中，先调用init\_property\_area函数，创建一块用于存储属性的存储区域，~~然后加载default.prop文件中的内容~~。再看init\_property\_area是如何工作的，它的代码如下所示：

static int init\_property\_area(void)

{

if (property\_area\_inited)

return -1;

if(\_\_system\_property\_area\_init())

return -1;

/\*

初始化存储空间，PA\_SIZE是这块存储空间的总大小，为32768字节,pa\_workspace

为workspace类型的结构体，下面是它的定义：

typedef struct {

~~void \*data; //存储空间的起始地址~~

size\_t size; //存储空间的大小

int fd; //共享内存的文件描述符

} workspace;

init\_workspace函数调用Android系统提供的ashmem\_create\_region函数创建一块

共享内存。关于共享内存的知识我们在第7章会接触，这里，只需把它当做一块普通的内存就

可以了。

\*/

if(init\_workspace(&pa\_workspace, 0))

return -1;

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

property\_area\_inited = 1;

return 0;

}

上面的内容比较简单，不过最后的赋值语句可是大有来头。\_\_system\_property\_area\_\_是bionic libc库中输出的一个变量，为什么这里要给它赋值呢？

原来，虽然属性区域是由init进程创建，但Android系统希望其他进程也能读取这块内存里的东西。为做到这一点，它便做了以下两项工作：

· 把属性区域创建在共享内存上，而共享内存是可以跨进程的。这一点，已经在上面的代码中见到了，init\_workspace函数内部将创建这个共享内存。

· 如何让其他进程知道这个共享内存呢？Android利用了gcc的constructor属性，这个属性指明了一个\_\_libc\_prenit函数，当bionic libc库被加载时，将自动调用这个\_\_libc\_prenit，这个函数内部就将完成共享内存到本地进程的映射工作。

（2）客户端进程获取存储空间

关于上面的内容，来看相关代码：

[-->libc\_init\_dynamic.c]

//constructor属性指示加载器加载该库后，首先调用\_\_libc\_prenit函数。这一点和Windows上

//动态库的DllMain函数类似

void \_\_attribute\_\_((constructor))\_\_libc\_prenit(void);

void \_\_libc\_prenit(void)

{

......

\_\_libc\_init\_common(elfdata); //调用这个函数

......

}

\_\_libc\_init\_common函数为：

[-->libc\_init\_common.c]

void \_\_libc\_init\_common(uintptr\_t \*elfdata)

{

......

\_\_system\_properties\_init();//初始化客户端的属性存储区域

}

[-->system\_properties.c]

int \_\_system\_properties\_init(void)

{

prop\_area \*pa;

int s,fd;

unsigned sz;

char\*env;

.....

//还记得在启动zygote一节中提到的添加环境变量的地方吗？属性存储区域的相关信息

//就是在那儿添加的，这里需要取出来使用了。

env =getenv("ANDROID\_PROPERTY\_WORKSPACE");

//取出属性存储区域的文件描述符。关于共享内存的知识，第7章中将会进行介绍。

fd =atoi(env);

env =strchr(env, ',');

if(!env) {

return -1;

}

sz =atoi(env + 1);

//映射init创建的那块内存到本地进程空间，这样本地进程就可以使用这块共享内存了。

//注意，映射的时候指定了PROT\_READ属性，所以客户端进程只能读属性，而不能设置属性。

pa =mmap(0, sz, PROT\_READ, MAP\_SHARED, fd, 0);

if(pa== MAP\_FAILED) {

return -1;

}

if((pa->magic != PROP\_AREA\_MAGIC) || (pa->version !=PROP\_AREA\_VERSION)) {

munmap(pa, sz);

return -1;

}

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

return0;

}

上面代码中很多地方和共享内存有关，在第7章中会对与共享内存有关问题进行介绍，读者也可先行学习有关共享内存的知识。

总之，通过这种方式，客户端进程可以直接读取属性空间，但没有权限设置属性。客户端进程又是如何设置属性呢？

2. 启动属性服务器

（1）启动属性服务器

init进程会启动一个属性服务器，而客户端只能通过和属性服务器交互才能设置属性。先来看属性服务器的内容，它由start\_property\_service函数启动，代码如下所示：

[-->Property\_servie.c]

int start\_property\_service(void)

{

int fd;

/\*

加载属性文件，其实就是解析这些文件中的属性，然后把它设置到属性空间中去。Android系统

一共提供了四个存储属性的文件，它们分别是：

#definePROP\_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"

\*/

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

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

load\_properties\_from\_file(PROP\_PATH\_LOCAL\_OVERRIDE);

//有一些属性是需要保存到永久介质上的，这些属性文件则由下面这个函数加载，这些文件

//存储在/data/property目录下，并且这些文件的文件名必须以persist.开头。这个函数

//很简单，读者可自行研究。

load\_persistent\_properties();

//创建一个socket，用于IPC通信。

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

if(fd< 0) return -1;

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

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

listen(fd, 8);

return fd;

}

属性服务创建了一个用来接收请求的socket，可这个请求在哪里被处理呢？事实上，在init中的for循环那里已经进行相关处理了。

（2）处理设置属性请求

接收请求的地方是在init进程中，代码如下所示：

[-->init.c::main函数片断]

if (ufds[1].revents == POLLIN)

handle\_property\_set\_fd(property\_set\_fd);

当属性服务器收到客户端请求时，init会调用handle\_property\_set\_fd进行处理。这个函数的代码如下所示：

[-->property\_service.c]

void handle\_property\_set\_fd()

{

prop\_msg msg;

int s;

int r;

int res;

struct ucred cr;

struct sockaddr\_un addr;

socklen\_t addr\_size = sizeof(addr);

socklen\_t cr\_size = sizeof(cr);

char \* source\_ctx = NULL;

struct pollfd ufds[1];

const int timeout\_ms = 2 \* 1000; /\* Default 2 sec timeout for caller to send property. \*/

int nr;

//先接收TCP连接

if ((s = accept(property\_set\_fd, (struct sockaddr \*) &addr, &addr\_size)) < 0) {

return;

}

/\* //取出客户端进程的权限等属性。Check socket options here \*/

if (getsockopt(s, SOL\_SOCKET, SO\_PEERCRED, &cr, &cr\_size) < 0) {

close(s);

ERROR("Unable to receive socket options\n");

return;

}

ufds[0].fd = s;

ufds[0].events = POLLIN;

ufds[0].revents = 0;

nr = TEMP\_FAILURE\_RETRY(poll(ufds, 1, timeout\_ms));

if (nr == 0) {

close(s);

return;

} else if (nr < 0) {

close(s);

return;

}

//recv接收请求数据

r = TEMP\_FAILURE\_RETRY(recv(s, &msg, sizeof(msg), MSG\_DONTWAIT));

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

close(s);

return;

}

switch(msg.cmd) {

case PROP\_MSG\_SETPROP:

msg.name[PROP\_NAME\_MAX-1] = 0;

msg.value[PROP\_VALUE\_MAX-1] = 0;

if (!is\_legal\_property\_name(msg.name, strlen(msg.name))) {

close(s);

return;

}

getpeercon(s, &source\_ctx);

/\*

如果是ctl开头的消息，则认为是控制消息，控制消息用来执行一些命令，例如用

adb shell登录后，输入setprop ctl.start bootanim就可以查看开机动画了，

关闭的话就输入setprop ctl.stop bootanim，是不是很有意思呢？

\*/

if(memcmp(msg.name,"ctl.",4) == 0) {

// Keep the old close-socket-early behavior when handling

// ctl.\* properties.

close(s);

if (check\_control\_mac\_perms(msg.value, source\_ctx)) {

handle\_control\_message((char\*) msg.name + 4, (char\*) msg.value);

} else {

ERROR("sys\_prop: Unable to %s service ctl [%s] uid:%d gid:%d pid:%d\n",

msg.name + 4, msg.value, cr.uid, cr.gid, cr.pid);

}

} else {

//检查客户端进程是否有足够的权限

if (check\_perms(msg.name, source\_ctx)) {

//然后调用property\_set设置。

property\_set((char\*) msg.name, (char\*) msg.value);

} else {

ERROR("sys\_prop: permission denied uid:%d name:%s\n",

cr.uid, msg.name);

}

// Note: bionic's property client code assumes that the

// property server will not close the socket until \*AFTER\*

// the property is written to memory.

close(s);

}

freecon(source\_ctx);

break;

default:

close(s);

break;

}

}

当客户端的权限满足要求时，init就调用property\_set进行相关处理，这个函数比较简单，代码如下所示：

[-->property\_service.c]

int property\_set(const char \*name, const char \*value)

{

prop\_info \*pi;

int ret;

size\_t namelen = strlen(name);

size\_t valuelen = strlen(value);

if (!is\_legal\_property\_name(name, namelen)) {

ERROR("PropSet Error:[%s:%s] property name is illegal\n", name, value);

return -1;

}

if (valuelen >= PROP\_VALUE\_MAX) {

return -1;

}

//从属性存储空间中寻找是否已经存在该属性

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

if(pi != 0) {

/\* ro.\* properties may NEVER be modified once set \*/

//如果属性名以ro.开头，则表示是只读的，不能设置，所以直接返回。

if(!strncmp(name, "ro.", 3)) {

return -1;

}

//更新该属性的值

\_\_system\_property\_update(pi, value, valuelen);

} else {

//如果没有找到对应的属性，则认为是增加属性，所以需要新创建一项。注意，Android支持

//最多247项属性，如果目前属性的存储空间中已经有247项，则直接返回。

#ifdef INIT\_ENG\_BUILD

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

if ((pa->bytes\_used > PA\_SIZE\_ERR) && (0 == alarmed ))

{

alarmed = 1;

}

else if ((pa->bytes\_used > PA\_SIZE\_WARN) && (0 == warned ))

{

warned = 1;

NOTICE("[Property Warning]: limit would be arrived:%d (Max:%d). "

"Use getprop to review your properties!\n", pa->bytes\_used, PA\_SIZE);

}

#endif

ret = \_\_system\_property\_add(name, namelen, value, valuelen);

if (ret < 0) {

#ifdef INIT\_ENG\_BUILD

ERROR("Failed to set '%s'='%s' bytes\_used=%u\n", name, value, pa->bytes\_used);

#else

ERROR("Failed to set '%s'='%s'\n", name, value);

#endif

return ret;

}

}

/\* If name starts with "net." treat as a DNS property. \*/

//有一些特殊的属性需要特殊处理，这里，主要是以net.change开头的属性。

if (strncmp("net.", name, strlen("net.")) == 0) {

if (strcmp("net.change", name) == 0) {

#ifdef MTK\_INIT

INFO("PropSet [%s:%s] Done\n", name, value);

#endif

return 0;

}

/\*

\* The 'net.change' property is a special property used track when any

\* 'net.\*' property name is updated. It is \_ONLY\_ updated here. Its value

\* contains the last updated 'net.\*' property.

\*/

property\_set("net.change", name);

} else if (persistent\_properties\_loaded &&

strncmp("persist.", name, strlen("persist.")) == 0) {

/\*

\* Don't write properties to disk until after we have read all default properties

\* to prevent them from being overwritten by default values.

\*/

//如果属性名以persist.开头，则需要把这些值写到对应文件中去。

write\_persistent\_property(name, value);

} else if (strcmp("selinux.reload\_policy", name) == 0 &&

strcmp("1", value) == 0) {

selinux\_reload\_policy();

}

/\*

还记得init.rc中的下面这句话吗？

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

startadbd

当persist.service.adb.enable属性置为1后，就会执行start adbd这个command，

这是通过property\_changed函数来完成的，它非常简单，读者可以自己阅读。

\*/

property\_changed(name, value);

#ifdef MTK\_INIT

INFO("PropSet [%s:%s] Done\n", name, value);

#endif

return 0;

}

好，属性服务端的工作已经了解了，下面看客户端是如何设置属性的。

（3）客户端发送请求

客户端通过property\_set发送请求，property\_set由libcutils库提供，代码如下所示：

[-->properties.c]

int property\_set(const char \*key, const char \*value)

{

return \_\_system\_property\_set(key, value);

}

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

{

if (key == 0){

\_\_libc\_format\_log(ANDROID\_LOG\_ERROR, "[PropSet]",

"key==0 return directly\n");

return -1;

}

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

if (strlen(key) >= PROP\_NAME\_MAX){

\_\_libc\_format\_log(ANDROID\_LOG\_ERROR, "[PropSet]",

"key=%s size:%zd >= max:%d\n",

key, strlen(key), PROP\_NAME\_MAX);

return -1;

}

if (strlen(value) >= PROP\_VALUE\_MAX){

\_\_libc\_format\_log(ANDROID\_LOG\_ERROR, "[PropSet]",

"value=%s size:%zd >= max:%d\n",

value, strlen(value), PROP\_VALUE\_MAX);

return -1;

}

prop\_msg msg;

memset(&msg, 0, sizeof msg);

msg.cmd = PROP\_MSG\_SETPROP;//设置消息码为PROP\_MSG\_SETPROP。

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

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

//发送请求

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

if (err < 0) {

\_\_libc\_format\_log(ANDROID\_LOG\_ERROR, "[PropSet]",

"send\_prop\_msg return err %d\n", err);

return err;

}

return 0;

}

[-->system\_properties.cpp]

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

{

const int fd = socket(AF\_LOCAL, SOCK\_STREAM | SOCK\_CLOEXEC, 0);

if (fd == -1) {

\_\_libc\_format\_log(ANDROID\_LOG\_ERROR, "[PropSet]",

"socket fail err:%d\n", fd);

return -1;

}

const size\_t namelen = strlen(property\_service\_socket);

sockaddr\_un addr;

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

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

addr.sun\_family = AF\_LOCAL;

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

//建立和属性服务器的socket连接

if (TEMP\_FAILURE\_RETRY(connect(fd, reinterpret\_cast<sockaddr\*>(&addr), alen)) < 0) {

\_\_libc\_format\_log(ANDROID\_LOG\_ERROR, "[PropSet]",

"connect fail\n");

close(fd);

return -1;

}

//通过socket发送出去

const int num\_bytes = TEMP\_FAILURE\_RETRY(send(fd, msg, sizeof(prop\_msg), 0));

int result = -1;

if (num\_bytes == sizeof(prop\_msg)) {

pollfd pollfds[1];

pollfds[0].fd = fd;

pollfds[0].events = 0;

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

if (poll\_result == 1 && (pollfds[0].revents & POLLHUP) != 0) {

result = 0;

} else {

result = 0;

}

}

close(fd);

return result;

}

至此，属性服务器就介绍完了。总体来说，还算比较简单。

3.3 本章小结

本章讲解了init进程如何解析zygote，以及属性服务器的工作原理，旨在帮助读者认识这个天字号第一进程。从整体来说，init.rc的解析难度相对最大。相信读者通过以上实例分析，已经理解了init.rc的解析原理。另外，inti涉及很多和Linux系统相关的知识，有兴趣的读者可以自行研究。