Android init进程之启动流程分析

作者:程姚根　　联系方式:chengyaogen123@163.com

上一节我们了解了一下init.rc文件的内容，以及init进程解析完init.rc文件后得到的效果。这一节我们来看看init进程启动过程。

一 、init 进程启动流程

```
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;

    if (!strcmp(basename(argv[0]), "ueventd"))
        return ueventd_main(argc, argv);

    if (!strcmp(basename(argv[0]), "watchdogd"))
        return watchdogd_main(argc, argv);

    /* clear the umask */
    umask(0);

    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);

    /*
     *(1)在/dev目录下生成__null__设备节点
     *(2)将标准输入、标准输出、标准出错重定向到__null__设备
     *
```

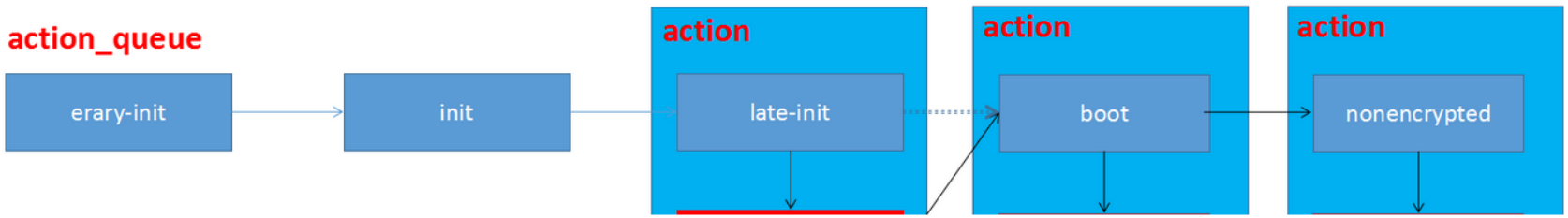
```
 */
open_devnull_stdio();

/*
 *在/dev目录下生成__kmsg__设备节点,__kmsg__设备调用内核输出函数printk,init进程
 *即是通过该函数来输出log信息的
 *
 */
klog_init();

/*初始化属性域:分配一块内存区域用来存放属性内容
 */
property_init();

/*获得硬件参数信息*/
get_hardware_name(hardware, &revision);

/*处理bootloader传递给内核参数*/
process_kernel_cmdline();

is_charger = !strcmp(bootmode, "charger");

/*加载/defulat.prop到属性内存区域*/
property_load_boot_defaults();

/*
 *解析init.rc文件，生成action_list 和 services_list
 */
init_parse_config_file("/init.rc");

/*
 *从action_list中寻找名字为early-init的action,并且将它添加到action_queue
 *
 */
action_for_each_trigger("early-init", action_add_queue_tail);

/*分别创建以下action,并且将它添加到action_queue
 *
 *(1)wait_for_cold_boot_done
 *(2)mix_hwrng_into_linux_rng
 *(3)keychord_init
 *(4)console_init
 */
queue_builtin_action(wait_for_coldboot_done_action, "wait_for_coldboot_done");
queue_builtin_action(keychord_init_action, "keychord_init");
queue_builtin_action(console_init_action, "console_init");

/*
 *从action list中寻找名字为init的action,并且将它添加到action_queue
 *
 */
action_for_each_trigger("init", action_add_queue_tail);

/*
 *分别创建以下action,并且将它添加到action_queue
 *(1)property_service_init
 *(2)signal_init
 */
queue_builtin_action(property_service_init_action, "property_service_init");
queue_builtin_action(signal_init_action, "signal_init");
```

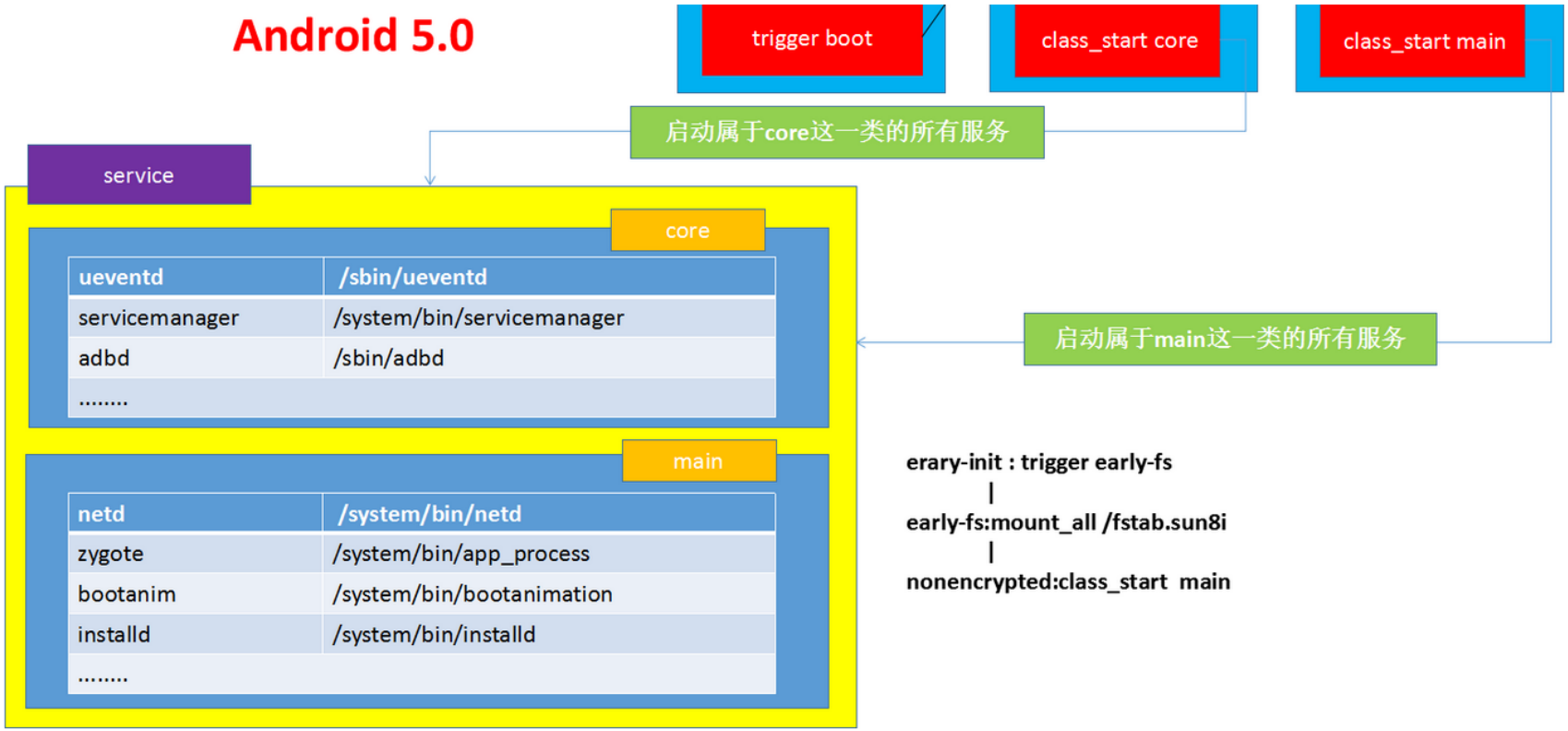
```
    /* Don't mount filesystems or start core system services if in charger mode. */
    if (is_charger) {
        action_for_each_trigger("charger", action_add_queue_tail);
    } else {
        action_for_each_trigger("late-init", action_add_queue_tail);
    }

    /*
     *创建名字为"queue_property_triggers",然后添加到action_queue
     *
     */
    queue_builtin_action(queue_property_triggers_action, "queue_property_triggers");

    for(;;) {
        int nr, i, timeout = -1;

        /*
         *从action_queue中提取一个action,然后执行这个action关联的第一个命令
         *,下一次调用这个函数的时候，执行action关联的下一个命令，直到将这个action
         *的所有命令都执行完毕，接着下一次调用这个函数的时候，重新从action_queue中
         *提取一个新的action,然后依次执行这个action相关联的命令
         */
        execute_one_command();

        /*
         *查询server_list中查询每个service的flag,如果flag标示为SVC_RESTARTING
         *则重新启动这个service
         */
        restart_processes();

        /*
         *判断一下文件描述符是否已经初始化过，如果初始化过则添加到poll函数关联的文件描述符
         *表中，让poll函数监视他们是否有相应的事件触发，如果有则调用相关的函数进行处理
         *
         *(1)property_set_fd
         *   init 进程创建的一个unix socket,用来和其他进程通信的，其他进程可以通过这个套接字
         *   向init进程发起修改属性请求，然后init进程修改共享内存中的值
         *
         *(2)signal_recv_fd
         *   init进程创建的一个unix socket,通过这个套接字，可以知道子进程已经结束，然后init进
         *   会对已经结束的子进程做一些处理
         *
         *(3)keychord_fd
         *   是/dev/keychord设备的文件描述符，通过这个文件描述init进程可以知道用户按下了一个
         *   然后执行这个组合键对应的代码，完成一些事情
         */
        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;
```

```
        }
        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;

        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();
            }
        }
    }

    return 0;
}
```

二、init.rc 中action和service运行

在前面的代码分析过程中我们发现，init进程在启动的过程中会把需要执行的action添加到全局的**action_queue**中，然后通过**execute_one_command()**执行action相关的命令。但是我们并没有看到init进程是如何启动init.rc文件中service的，下面我们就带这个问题出发，寻找最终的答案。





从图中可以看到Android 中service的启动实际上是通过action关联的相关命令来启动的，action中和service相关的命令有:

action中和service相关的命令	
class_start <service class>	启动所有指定属于class类的服务
class_stop <service class>	停止所有指定属于class类的服务，后续没法通过class_start启动
class_reset <service class>	停止服务，后续可以通过class_start启动
restart <service name>	重启指定名称的服务，先stop,在start
start <service name>	启动指定名称的服务
stop <service name>	停止指定名称的服务

在来总结一下，init.rc中启动的相关服务吧！

core 类相关的服务		
ueventd	/sbin/ueventd	处理内核抛出的uevent消息
console	/system/bin/sh	控制台服务
adbd	/sbin/adbd	adb调试的服务端
servicemanager	/system/bin/servicemanager	管理服务的服务，被管理的服务通常是供应用程序使用的
vold	/system/bin/vold	管理存储设备

注意:

可以看到，core服务都是系统基本的服务，只要core服务全部启动，手机此时是可以运行的，但是却看不到东西，原因framework没有启动。此时启动的都是C/C ++的进程。 此时是不能打电话的，因为ril-deamon没有启动。

main 类相关的服务		
netd	/system/bin/netd	网络管理器
deguggerd	/system/bin/deguggerd	可以在logcat中输出调试信息
ril-deamon	/system/bin/rild	打电话的服务
surfaceflinger	/system/bin/surfaceflinger	合成framebuffer的服务
zygote	/system/bin/app_process	孵化java应用进程的服务
drm	/system/bin/drmserver	DRM服务,frameworks/base/drm
media	/system/bin/mediaserver	多媒体服务
bootanim	/system/bin/bootanimation	开机动画服务
dbus	/system/bin/dbus-daemon	用于进程间通讯的服务
bluetoothd	/system/bin/bluetoothd	蓝牙
installd	/system/bin/installd	apk安装的服务
flash_recovery	/system/etc/install-recovery.sh	recover recovery分区

racoon	/system/bin/racoon	key management daemon
mtpd	/system/bin/mtpd	MTP(Media Transfer Protocol)daemon
keystore	/system/bin/keysotre	应用签名
dumpstate	/system/bin/dumpstate	性能测试工具

注意:
可以看到main的服务相对多一些，看到zygote了吧，由此可见main服务大部分是建立在java层或者与java层息息相关的系统服务。

late_start 类相关的服务		
sdcard(服务名称)	init.sun8i.rc(所属的文件)	/system/bin/sdcard(命令所在的路径)

在来看一下相关的代码，加深理解（下面的代码是从原来的Android源码中摘出来做了精简，目的是为了让大家能更容易看懂）

```
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;
}
```

通过class_start命令启动的服务的时候，调用的是do_class_start函数。

```
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);
        }
    }
}
```

从service_list中寻找指定类的服务,然后启动它。

```
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;
    }
}
```

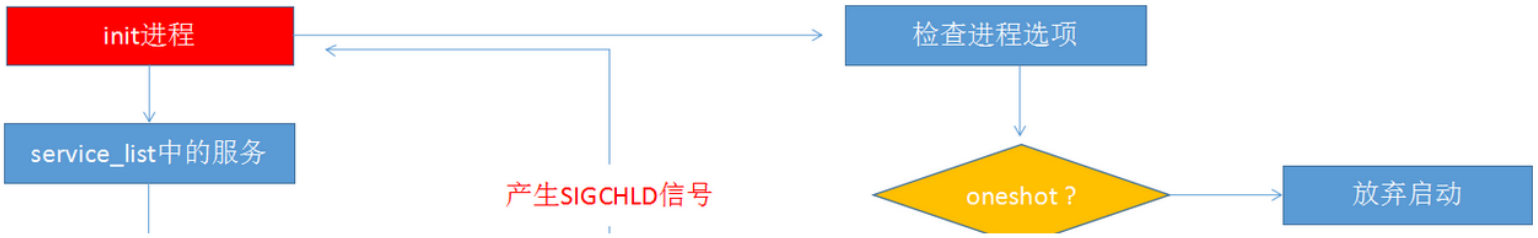
如果这个服务的选项中没有指定disabled则启动这个服务。

```
void service_start(struct service *svc, const char *dynamic_args)
{

    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;
    }

    NOTICE("starting '%s'\n", svc->name);

    pid = fork();

    if (pid == 0) {

        execve(svc->args[0], (char**) arg_ptrs, (char**) ENV);

        _exit();
    }

    svc->pid = pid;
    svc->flags |= SVC_RUNNING;

    return;
}
```

通过fork()函数创建子进程，然后调用execve函数执行这个服务对应的应用程序。

三、init进程如何守护service运行

在Android系统运行的过程中，可能由于用户非法操作或者其他原因导致一些核心服务进程死掉，而这些核心服务进程死掉后，必定会导致Android系统瘫痪，为了避免这种情况，一些核心的服务进程死掉后,init进程就会重新启动这些服务。下面我们来看看init进程是如何重新启动死掉的进程的。



我们知道init进程会启动服务列表中的服务，创建相应的子进程。 如上图所示，当init的子进程意外终止时，会向父进程init进程传递SIGCHLD信号，init进程接收该信号，检查进程选项是否设置为oneshot,若设置为oneshot,init进程将放弃重启进程，否则重启进程。

下面我们来看看相关代码的实现:

我们知道，当子进程的状态发生改变时，Linux 操作会向父进程发送SIGCHLD信号，父亲默认对SIGCHLD信号是忽略的，如果想处理SIGCHLD信号，父进程必须先对SIGCHLD信号进行设置。我们先看看init进程是如何设置对SIGCHLD信号处理的。

(1)**queue_builtin_action(signal_init_action, "signal_init");**
建立一个新的action,并且将它添加到action_queue,当这个action从队列中取出来执行的时候，会执行signal_init_action这个函数，下面我们来看看这个函数是如何实现的。

(2)**static int signal_init_action(int nargs, char **args)**

```
static int signal_init_action(int nargs, char **args)
{
    signal_init();
    if (get_signal_fd() < 0) {
        ERROR("signal_init() failed\n");
        exit(1);
    }
    return 0;
}
```

(3)**signal_init()**

```
void signal_init(void)
{
    int s[2];

    struct sigaction act;
    memset(&act, 0, sizeof(act));
    act.sa_handler = sigchld_handler;
    act.sa_flags = SA_NOCLDSTOP;
    sigaction(SIGCHLD, &act, 0);

    /* create a signalling mechanism for the sigchld handler */
```

```
    if (socketpair(AF_UNIX, SOCK_STREAM, 0, s) == 0) {
        signal_fd = s[0];
        signal_recv_fd = s[1];
        fcntl(s[0], F_SETFD, FD_CLOEXEC);
        fcntl(s[0], F_SETFL, O_NONBLOCK);
        fcntl(s[1], F_SETFD, FD_CLOEXEC);
        fcntl(s[1], F_SETFL, O_NONBLOCK);
    }

    handle_signal();
}
```

<1>从代码中我们可以知道init进程将SIGCHLD信号设置为捕捉方式，当收到SIGCHLD信号会调用**sigchld_handler**函数
<2>创建了一个匿名的unix域socket,它是一个全双工的通信方式。每个文件描述符都可以读写，从第一个文件描述写入就可以从第二个文件描述读出，从第二个文件描述符写入就可以从第一个文件描述符读出。这里用**signal_fd**记录第一个文件描述符，**signal_recv_fd**记录第二个文件描述符号。

看我init进程初始化对SIGCHLD信号处理方式后，我们思考一个问题，此时如果有一个子进程结束，会发送什么事情呢?

(1)**init进程收到SIGCHLD信号，调用sigchld_handler函数**

```
static void sigchld_handler(int s)
{
    write(signal_fd, &s, 1);
}
```

很简单，就是向前面初始化的SIGCHLD信号时创建的套接字中写入了信号的编号数字。

(2)**init进程的poll函数探测到signal_recv_fd文件描述符就绪，调用handler_signal函数**

```
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))
        ;
}
```

从socket中读取数据，然后调用wait_for_one_process(0)函数。

(3)**static int wait_for_one_process(int block)**

```
static int wait_for_one_process(int block)
{
    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);

    svc = service_find_by_pid(pid);
    NOTICE("process '%s', pid %d exited\n", svc->name, pid);
```

```
    if (!(svc->flags & SVC_ONESHOT) || (svc->flags & SVC_RESTART)) {
        kill(-pid, SIGKILL);
        NOTICE("process '%s' killing any children in process group\n", svc->name);
    }
        /* 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;
    }
    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);
    }
    notify_service_state(svc->name, "restarting");
    return 0;
}
```

<1>通过waitpid函数回收僵尸态子进程未释放的资源

<2>通过service_find_by_pid函数根据子进程的PID寻找到这个子进程对应的服务。

<3>如果这个service的flags包含有SVC_ONESHOT,就将和这个service在同一个进程组的所有进程全部杀掉，并且将这个service的flags加上SVC_DISABLED，接着代码就返回啦。

<4>如果这个service的flags不包含有SVC_ONESHOT,就将这个服务的 flags 添加上SVC_RESTARTING标志

<4>如果这个服务存在onrestart选项，将其关联的命令都执行一遍

貌似，没有看到重新启动服务的代码呀?不急接着看，马上就看到了....

(4)init进程死循环中restart_processes()函数调用

```
static void restart_processes()
{
    process_needs_restart = 0;
    service_for_each_flags(SVC_RESTARTING,
                           restart_service_if_needed);
}
```

(5)void service_for_each_flags(unsigned matchflags, void (*func)(struct service *svc))

```
void service_for_each_flags(unsigned matchflags,
                            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 (svc->flags & matchflags) {
            func(svc);
        }
```

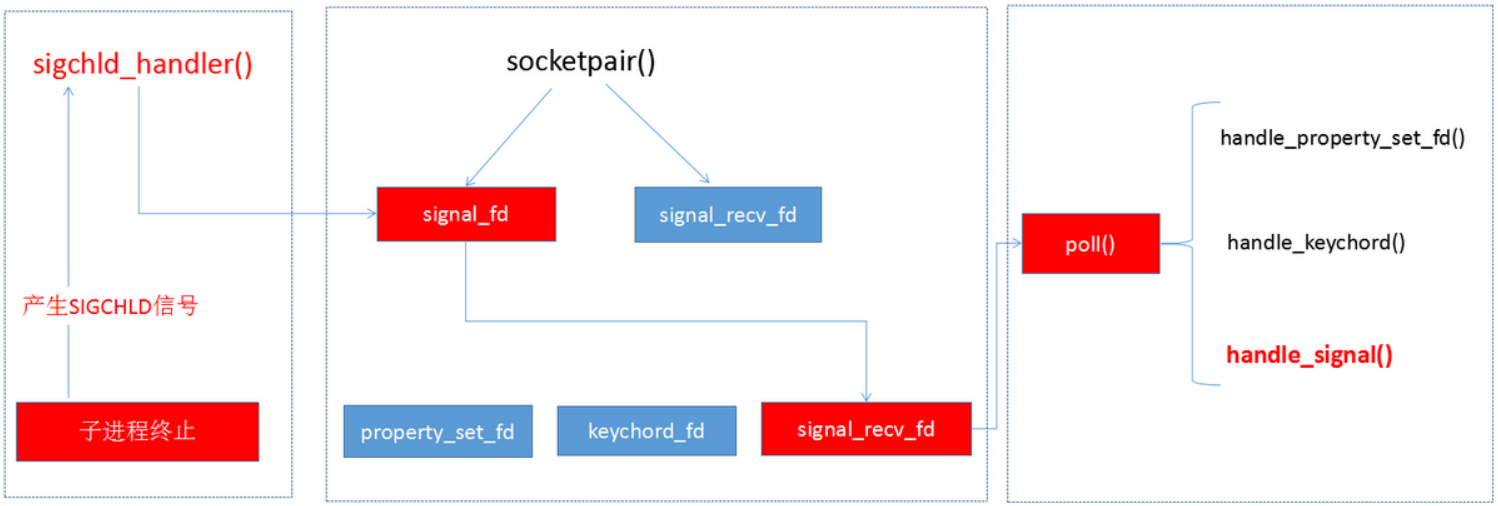
```
    }
}
```

从service_list中寻找满足标志(SVC_RESTARTING)的service,然后调用func函数(restart_service_if_needed)

(6)static void restart_service_if_needed(struct service *svc)

```
static void restart_service_if_needed(struct service *svc)
{
    time_t next_start_time = svc->time_started + 5;

    if (next_start_time <= gettime()) {
        svc->flags &= (~SVC_RESTARTING);
        service_start(svc, NULL);

        return;
    }

    if ((next_start_time < process_needs_restart) ||
        (process_needs_restart == 0)) {
        process_needs_restart = next_start_time;
    }
}
```

呵呵，终于看到service_start启动我们的服务了。好了最后我们在画一幅图，加深我们的理解。

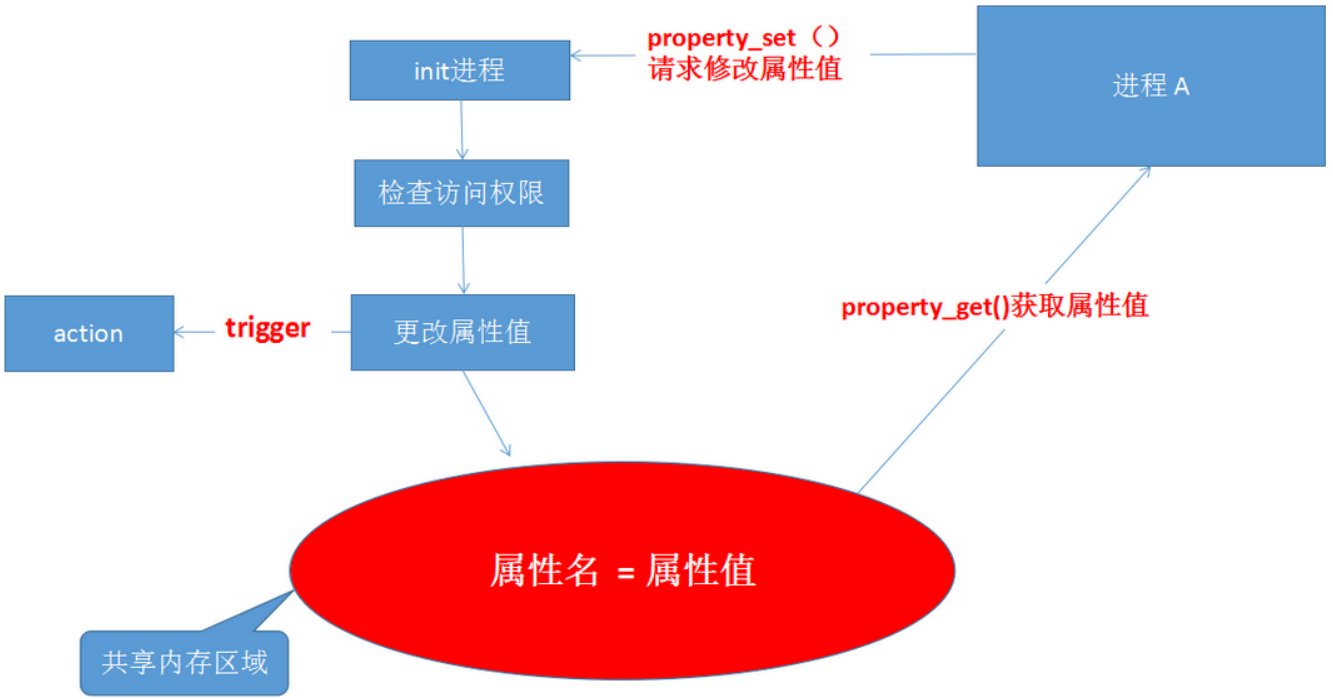


三、init进程如何修改属性值

属性变更请求是init事件处理循环处理的另一个事件。在Android平台中，为了让运行中的所有进程共享系统运行时所需要的各种设置值，系统开辟了属性存储区域，并提供了访问该区域的API。属性由键(key)与值(value)构成，其表现形式为"键 = 值"。**在Linux系统中，属性服务主要用来设置环境变量，提供各进程访问设定的环境变量值。**在Android平台中，属性服务得到更系统地应用，在访问属性值时，添加了访问权限控制，增强了访问的安全性。**系统中所有运行中**

的进程都可以访问属性值，但仅有init进程才能属性值。其他进程修改属性值时，必须向init进程提出请求，最终由init进程负责修改属性值。在此过程中，init进程会先检查各属性的访问权限，然后再修改属性值。**当属性值更改后，若定义在init.rc文件中的某个特定条件得到满足，则与此条件相匹配的动作就会发生。**每个动作都是有一个"触发器"(trigger),它决定动作的执行时间，记录在"on property"关键字后的命令立即被执行。

下面我们简单描述init进程与其他进程在访问并修改属性值得大致情形。



相关的代码就不分析了，想了解的读者可以自己去网上了解，这里推荐一篇博客，写的比较详细。

深入讲解Android Property机制：http://my.oschina.net/youranhongcha/blog/389640