深入讲解Android Property机制 原

悠然红茶 发布于 2015/03/21 00:29 字数 7129 阅读 5279 收藏 174 点赞 12 评论 10

Android property

深入讲解Android Property机制

侯亮

1 概述

Android系统(本文以Android 4.4为准)的属性(Property)机制有点儿类似Windows系统的注册表，其中的每个属性被组织成简单的键值对(key/value)供外界使用。

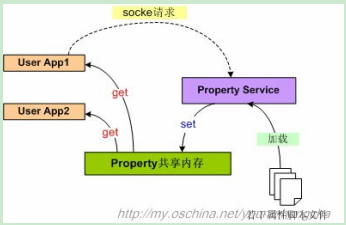
我们可以通过在adb shell里敲入getprop命令来获取当前系统的所有属性内容，而且，我们还可以敲入类似"getprop 属性名"的命令来获取特定属性的值。另外，设置属性值的方法也很简单，只需敲入"setprop 属性名 新值"命令即可。

可是问题在于我们不想只认识到这个层次，我们希望了解更多一些Property机制的运作机理，而这才是本文关心的重点。

说白了，Property机制的运作机理可以汇总成以下几句话：

1) 系统一启动就会从若干属性脚本文件中加载属性内容；
2) 系统中的所有属性(key/value)会存入同一块共享内存中；
3) 系统中的各个进程会将这块共享内存映射到自己的内存空间，这样就可以直接读取属性内容了；
4) 系统中只有一个实体可以设置、修改属性值，它就是属性服务(Property Service)；
5) 不同进程只可以通过socket方式，向属性服务发出修改属性值的请求，而不能直接修改属性值；
6) 共享内存中的键值内容会以一种字典树的形式进行组织。

Property机制的示意图如下：



2 Property Service

2.1 init进程里的Property Service

Property Service实体其实是在init进程里启动的。我们知道，init是Linux系统中用户空间的第一个进程。它负责创建系统中最关键的几个子进程，比如zygote等等。在本节中，我们主要关心init进程是如何启动Property Service的。

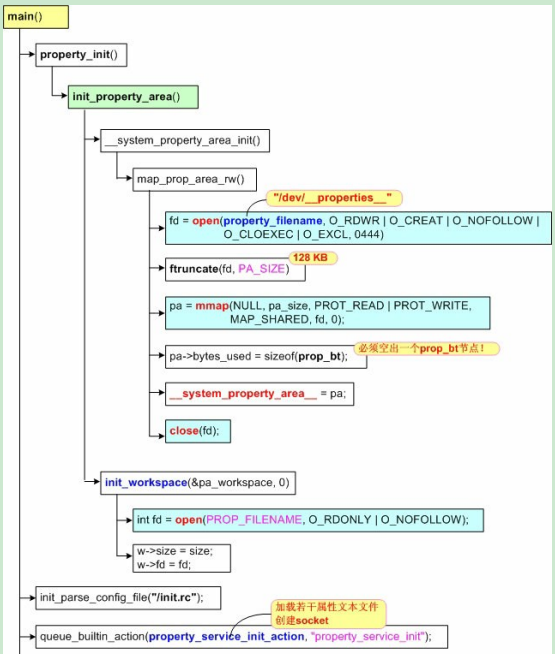
我们查看core/init/Init.c文件，可以看到init进程的main()函数，它里面和property相关的关键动作有：

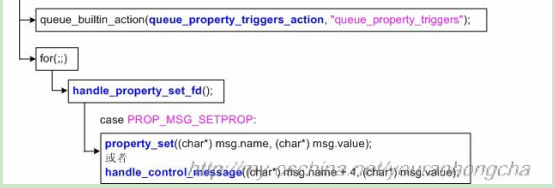
1)间接调用__system_property_area_init()：打开属性共享内存，并记入__system_property_area变量；
2)间接调用init_workspace()：只读打开属性共享内存，并记入环境变量；
3)根据init.rc，异步激发property_service_init_action()，该函数中会：

- 加载若干属性文本文件，将具体属性、属性值记入属性共享内存；
- 创建并监听socket；

4)根据init.rc，异步激发queue_property_triggers_action()，将刚刚加载的属性对应的激发动作，推入action列表。

main()中的调用关系如下：





2.1.1 初始化属性共享内存

我们可以看到，在init进程的main()函数里，辗转打开了一个内存文件"/dev/__properties__"，并把它设定为128KB大小，接着调用mmap()将这块内存映射到init进程空间了。这个内存的首地址被记录在__system_property_area__全局变量里，以后每添加或修改一个属性，都会基于这个__system_property_area__变量来计算位置。

初始化属性内存块时，为什么要两次open那个/dev/__properties__文件呢？我想原因是这样的：第一次open的句柄，最终是给属性服务自己用的，所以需要有读写权限；而第二次open的句柄，会被记入pa_workspace.fd，并在合适时机添加进环境变量，供其他进程使用，因此只能具有读取权限。

第一次open时，执行的代码如下：

```
fd = open(property_filename, O_RDWR | O_CREAT | O_NOFOLLOW | O_CLOEXEC | O_EXCL, 0444);
```

传给open()的参数标识里指明了O_RDWR，表示用"读写方式"打开文件。另外O_NOFOLLOW标识主要是为了防止我们打开"符号链接"，不过我们知道，__properties__文件并不是符号链接，所以当然可以成功open。O_CLOEXEC标识是为了保证一种独占性，也就是说当init进程打开这个文件时，此时就算其他进程也open这个文件，也会在调用exec执行新程序时自动关闭该文件句柄。O_EXCL标识和O_CREATE标识配合起来，表示如果文件不存在，则创建之，而如果文件已经存在，那么open就会失败。第一次open动作后，会给__system_property_area__赋值，然后程序会立即close刚打开的句柄。

第二次open动作发生在接下来的init_workspace()函数里。此时会再一次打开__properties__文件，这次却是以只读模式打开的：

```
int fd = open(PROP_FILENAME, O_RDONLY | O_NOFOLLOW);
```

打开的句柄记录在pa_workspace.fd处，以后每当init进程执行socket命令，并调用service_start()时，会执行类似下面的句子：

```
get_property_workspace(&fd, &sz);   // 读取pa_workspace.fd
sprintf(tmp, "%d,%d", dup(fd), sz);
add_environment("ANDROID_PROPERTY_WORKSPACE", tmp);
```

说白了就是把 pa_workspace.fd 的句柄记入一个名叫"ANDROID_PROPERTY_WORKSPACE "的环境变量去。

【system/core/init/Init.c】

```
/* add_environment - add "key=value" to the current environment */
int add_environment(const char *key, const char *val)
{
    int n;

    for (n = 0; n < 31; n++) {
        if (!ENV[n]) {
            size_t len = strlen(key) + strlen(val) + 2;
            char *entry = malloc(len);
            snprintf(entry, len, "%s=%s", key, val);
            ENV[n] = entry;
            return 0;
        }
    }

    return 1;
}
```

这个环境变量在日后有可能被其他进程拿来用，从而将属性内存区映射到自己的内存空间去，这个后文会细说。

接下来，main()函数在设置好属性内存块之后，会调用queue_builtin_action()函数向内部的action_list列表添加action节点。关于这部分的详情，可参考其他讲述Android启动机制的文档，这里不再赘述。我们只需知道，后续，系统会在合适时机回调"由queue_builtin_action()的参数"所指定的property_service_init_action()函数就可以了。

2.1.2 初始化属性服务

property_service_init_action()函数只是在简单调用start_property_service()而已，后者的代码如下：

【core/init/Property_service.c】

```
void start_property_service(void)
{
    int fd;

    load_properties_from_file(PROP_PATH_SYSTEM_BUILD);
    load_properties_from_file(PROP_PATH_SYSTEM_DEFAULT);

    /* Read vendor-specific property runtime overrides. */
    vendor_load_properties();

    load_override_properties();
    /* Read persistent properties after all default values have been loaded. */
    load_persistent_properties();

    fd = create_socket(PROP_SERVICE_NAME, SOCK_STREAM, 0666, 0, 0);
    if(fd < 0) return;
    fcntl(fd, F_SETFD, FD_CLOEXEC);
    fcntl(fd, F_SETFL, O_NONBLOCK);

    listen(fd, 8);
    property_set_fd = fd;
}
```

其主要动作无非是加载若干属性文件，然后创建并监听一个socket接口。

2.1.2.1 加载属性文本文件

start_property_service()函数首先会调用load_properties_from_file()函数，尝试加载一些属性脚本文件，并将其中的内容写入属性内存块里。从代码里可以看到，主要加载的文件有：

- /system/build.prop
- /system/default.prop(该文件不一定存在)
- /data/local.prop
- /data/property目录里的若干脚本

load_properties_from_file()函数的代码如下：

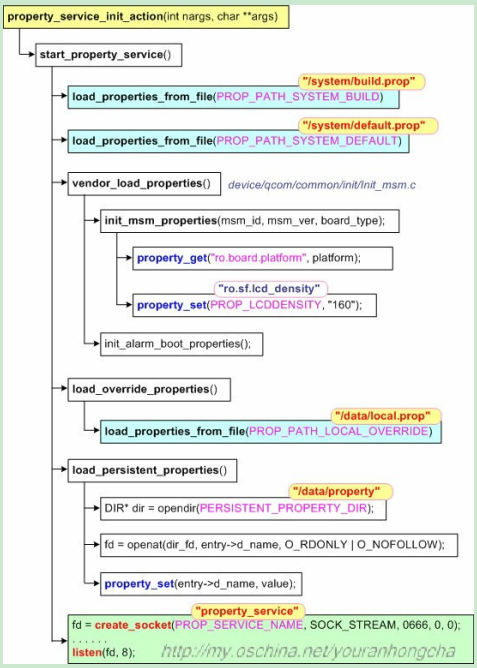
【core/init/Property_service.c】

```
static void load_properties_from_file(const char *fn)
{
    char *data;
    unsigned sz;

    data = read_file(fn, &sz);

    if(data != 0) {
        load_properties(data);
        free(data);
    }
}
```

其中调用的read_file()函数很简单，只是把文件内容的所有字节读入一个buffer，并在内容最后添加两个字节：'\n'和0。

接着调用的load_properties()函数，会逐行分析传来的buffer，解析出行内的key、value部分，并调用property_set()，将key、value设置进系统的属性共享内存去。

我们绘制出property_service_init_action()函数的调用关系图，如下：



2.1.2.2 创建socket接口

在加载动作完成后，start_property_service ()会创建一个socket接口，并监听这个接口。

【core/init/Property_service.c】

```
fd = create_socket(PROP_SERVICE_NAME, SOCK_STREAM, 0666, 0, 0);
if(fd < 0) return;
fcntl(fd, F_SETFD, FD_CLOEXEC);
fcntl(fd, F_SETFL, O_NONBLOCK);
listen(fd, 8);
property_set_fd = fd;
```

这个socket是专门用来监听其他进程发来的“修改”属性值的命令的，它被设置成“非阻塞”(O_NONBLOCK)的socket。

2.1.3 初始化属性后的触发动作

既然在上一小节的property_service_init_action()动作中，系统已经把必要的属性都加载好了，那么现在就可以遍历刚生成的action_list，看看哪个刚加载好的属性可以进一步触发连锁动作。这就是init进程里为什么有两次和属性相关的queue_builtin_action()的原因。

【system/core/init/Init.c】

```
static int queue_property_triggers_action(int nargs, char **args)
{
    queue_all_property_triggers();
    /* enable property triggers */
    property_triggers_enabled = 1;
    return 0;
}
```

【system/core/init/Init_parser.c】

```
void queue_all_property_triggers()
{
    struct listnode *node;
    struct action *act;
    list_for_each(node, &action_list) {
        act = node_to_item(node, struct action, alist);
        if (!strncmp(act->name, "property:", strlen("property:"))) {
            /* parse property name and value
               syntax is property:<name>=<value> */
            const char* name = act->name + strlen("property:");
            const char* equals = strchr(name, '=');
            if (equals) {
                char prop_name[PROP_NAME_MAX + 1];
                char value[PROP_VALUE_MAX];
                int length = equals - name;
                if (length > PROP_NAME_MAX) {
                    ERROR("property name too long in trigger %s", act->name);
                } else {
                    memcpy(prop_name, name, length);
                    prop_name[length] = 0;
```

```
                    /* does the property exist, and match the trigger value? */
                    property_get(prop_name, value);
                    if (!strcmp(equals + 1, value) ||!strcmp(equals + 1, "*")) {
                        action_add_queue_tail(act);
                    }
                }
            }
        }
    }
}
```

这段代码是说，当获取的属性名和属性值，与当初init.rc里记录的某action的激发条件匹配时，就把该action插入执行队列的尾部（action_add_queue_tail(act)）。

2.2 init进程循环监听socket

现在再回过头看init进程，其main()函数的最后，我们可以看到一个for(;;)循环，不断监听外界发来的命令，包括设置属性的命令。

【system/core/init/Init.c】

```
for(;;) {
    . . . . . .
    . . . . . .
    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();
        }
    }
}
```

2.2.1 处理“ctl.”命令

当从socket收到“设置属性”的命令后，会调用上面的handle_property_set_fd()函数，代码截选如下：

【core/init/Property_service.c】

```
void handle_property_set_fd()
{
    prop_msg msg;
    . . . . . .
    if ((s = accept(property_set_fd, (struct sockaddr *) &addr, &addr_size)) < 0) {
        return;
    }
    . . . . . .
    switch(msg.cmd) {
    case PROP_MSG_SETPROP:
        msg.name[PROP_NAME_MAX-1] = 0;
        msg.value[PROP_VALUE_MAX-1] = 0;
        . . . . . .
        if(memcmp(msg.name,"ctl.",4) == 0) {
            . . . . . .
            if (check_control_perms(msg.value, cr.uid, cr.gid, 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 {
            if (check_perms(msg.name, cr.uid, cr.gid, source_ctx)) {
                property_set((char*) msg.name, (char*) msg.value);
            } else {
                ERROR("sys_prop: permission denied uid:%d  name:%s\n",
                      cr.uid, msg.name);
            }
            . . . . . .
            close(s);
        }
        . . . . . .
        break;
    . . . . . .
    }
}
```

看到了吗？设置属性时，一开始就把属性名和属性值的长度都限制了。

```
#define PROP_NAME_MAX   32
#define PROP_VALUE_MAX  92
```

也就是说，有意义的部分的最大字节数分别为31字节和91字节，最后一个字节先被强制设为0了。

2.2.1.1 check_control_perms()

对于普通属性而言，主要是调用property_set()来设置属性值，但是有一类特殊属性是以“ctl.”开头的，它们本质上是一些控制命令，比如启动某个系统服务。这种控制命令需调用handle_control_message()来处理。

当然，并不是随便谁都可以发出这种控制命令的，也就是说，不是谁都可以成功设置以“ctl.”开头的特殊属性。handle_property_set_fd()会先调用check_control_perms()来检查发起方是否具有相应的权限。

【core/init/Property_service.c】

```
static int check_control_perms(const char *name, unsigned int uid, unsigned int gid, char *sctx) {
    int i;
    if (uid == AID_SYSTEM || uid == AID_ROOT)
      return check_control_mac_perms(name, sctx);

    /* Search the ACL */
    for (i = 0; control_perms[i].service; i++) {
        if (strcmp(control_perms[i].service, name) == 0) {
            if ((uid && control_perms[i].uid == uid) ||
                (gid && control_perms[i].gid == gid)) {
                return check_control_mac_perms(name, sctx);
            }
        }
    }
    return 0;
}
```

可以看到，如果设置方的uid是AID_SYSTEM或者AID_ROOT，那么一般都是具有权限的。而如果uid是其他值，那么就得

查control_perms表了，这个表的定义如下：

【core/init/Property_service.c】

```
/*
 * White list of UID that are allowed to start/stop services.
 * Currently there are no user apps that require.
 */
struct {
    const char *service;
    unsigned int uid;
    unsigned int gid;
} control_perms[] = {
    { "dumpstate",AID_SHELL, AID_LOG },
    { "ril-daemon",AID_RADIO, AID_RADIO },
     {NULL, 0, 0 }
};
```

uid为AID_SHELL的进程可以启动、停止dumpstate服务，uid为AID_RADIO的进程可以启动、停止ril-daemon服务。

2.2.1.2 handle_control_message()

在通过权限检查之后，就可以调用handle_control_message()来处理控制命令了：

【system/core/init/Init.c】

```
void handle_control_message(const char *msg, const char *arg)
{
    if (!strcmp(msg,"start")) {
        msg_start(arg);
    } else if (!strcmp(msg,"stop")) {
        msg_stop(arg);
    } else if (!strcmp(msg,"restart")) {
        msg_restart(arg);
    } else {
        ERROR("unknown control msg '%s'\n", msg);
    }
}
```

假设从socket发来的命令是“ctl.start”，那么就会走到msg_start(arg)。

```
static void msg_start(const char *name)
{
    struct service *svc = NULL;
    char *tmp = NULL;
    char *args = NULL;

    if (!strchr(name, ':'))
        svc = service_find_by_name(name);
    else {
        tmp = strdup(name);
        if (tmp) {
            args = strchr(tmp, ':');
            *args = '\0';
            args++;

            svc = service_find_by_name(tmp);
        }
    }

    if (svc) {
        service_start(svc, args);
    } else {
        ERROR("no such service '%s'\n", name);
    }
    if (tmp)
        free(tmp);
}
```

这里启动的service基本上都是在init.rc里说明的系统service。比如netd：

```
service netd /system/bin/netd
    class main
    socket netd stream 0660 root system
    socket dnsproxyd stream 0660 root inet
    socket mdns stream 0660 root system
```

我们知道，init进程在分析init.rc文件时，会形成一个service链表，现在msg_start()就是从这个service链表里去查找相应名称的service节点的。找到节点后，再调用service_start(svc, args)。

service_start()常常会fork一个子进程，然后为它设置环境变量（ANDROID_PROPERTY_WORKSPACE）：

```
void service_start(struct service *svc, const char *dynamic_args)
{
    . . . . . .
    . . . . . .
    pid = fork();

    if (pid == 0) {
        struct socketinfo *si;
        struct svcenvinfo *ei;
        char tmp[32];
        int fd, sz;

        umask(077);
        if (properties_inited()) {
            get_property_workspace(&fd, &sz);
            sprintf(tmp, "%d,%d", dup(fd), sz);
            add_environment("ANDROID_PROPERTY_WORKSPACE", tmp);
        }

        for (ei = svc->envvars; ei; ei = ei->next)
            add_environment(ei->name, ei->value);
    . . . . . .
```

其中 get_property_workspace() 的代码如下：

```
void get_property_workspace(int *fd, int *sz)
{
    *fd = pa_workspace.fd;
    *sz = pa_workspace.size;
}
```

大家还记得前文阐述init_workspace()时，把打开的句柄记入pa_workspace.fd的句子吧，现在就是在用这个句柄。

一切准备好后，service_start()会调用execve()，执行svc->args[0]所指定的可执行文件，然后还要再写个属性值：

```
void service_start(struct service *svc, const char *dynamic_args)
{
    . . . . . .
    . . . . . .
    execve(svc->args[0], (char**) arg_ptrs, (char**) ENV);
    . . . . . .
    . . . . . .
    svc->time_started = gettime();
    svc->pid = pid;
    svc->flags |= SVC_RUNNING;

    if (properties_inited())
        notify_service_state(svc->name, "running");
}
```

其中的notify_service_state()的代码如下：

```
void notify_service_state(const char *name, const char *state)
{
    char pname[PROP_NAME_MAX];
    int len = strlen(name);
    if ((len + 10) > PROP_NAME_MAX)
        return;
    snprintf(pname, sizeof(pname), "init.svc.%s", name);
    property_set(pname, state);
}
```

一般情况下，这种在init.rc里记录的系统service的名字都不会超过22个字节，加上“init.svc.”前缀也不会超过31个字节，所以每次启动service，都会修改相应的属性。比如netd服务，一旦它被启动，就会将init.svc.netd属性的值设为“running”。

以上是handle_control_message()处理“ctl.start”命令时的情况，相应地还有处理“ctl.stop”命令的情况，此时会调用到msg_stop()。

【system/core/init/Init.c】

```
static void msg_stop(const char *name)
{
    struct service *svc = service_find_by_name(name);

    if (svc) {
        service_stop(svc);
    } else {
        ERROR("no such service '%s'\n", name);
    }
}
```

```
void service_stop(struct service *svc)
{
    service_stop_or_reset(svc, SVC_DISABLED);
}
```

```
static void service_stop_or_reset(struct service *svc, int how)
{
    /* The service is still SVC_RUNNING until its process exits, but if it has
     * already exited it shoudn't attempt a restart yet. */
    svc->flags &= (~SVC_RESTARTING);

    if ((how != SVC_DISABLED) && (how != SVC_RESET) && (how != SVC_RESTART)) {
        /* Hrm, an illegal flag.  Default to SVC_DISABLED */
        how = SVC_DISABLED;
    }
        /* if the service has not yet started, prevent
         * it from auto-starting with its class
         */
    if (how == SVC_RESET) {
        svc->flags |= (svc->flags & SVC_RC_DISABLED) ? SVC_DISABLED : SVC_RESET;
    } else {
        svc->flags |= how;
    }

    if (svc->pid) {
        NOTICE("service '%s' is being killed\n", svc->name);
        kill(-svc->pid, SIGKILL);
        notify_service_state(svc->name, "stopping");
    } else {
        notify_service_state(svc->name, "stopped");
    }
}
```

可以看到，停止一个service时，主要是调用kill( )来杀死服务子进程，并将init.svc.xxx属性值设为stopping。

OK，终于把init进程里，处理“ctl.”命令的部分讲完了，下面我们接着看init进程处理普通属性的部分。

2.2.2 处理属性设置命令

我们还是先回到前文init进程处理属性设置动作的地方：

```
void handle_property_set_fd()
{
        . . . . . .
        if(memcmp(msg.name,"ctl.",4) == 0) {
            . . . . . .
        } else {
            if (check_perms(msg.name, cr.uid, cr.gid, source_ctx)) {
                property_set((char*) msg.name, (char*) msg.value);
            } else {
```

```
                ERROR("sys_prop: permission denied uid:%d  name:%s\n",
                      cr.uid, msg.name);
            }
            . . . . . .
            close(s);
        }
        . . . . . .
        break;
    . . . . . .
    }
}
```

2.2.2.1 check_perms()

要设置普通属性，也是要具有一定权限哩。请看上面的 check_perms() 一句。该函数的代码如下：

```
static int check_perms(const char *name, unsigned int uid, unsigned int gid, char *sctx)
{
    int i;
    unsigned int app_id;

    if(!strncmp(name, "ro.", 3))
        name +=3;

    if (uid == 0)
        return check_mac_perms(name, sctx);

    app_id = multiuser_get_app_id(uid);
    if (app_id == AID_BLUETOOTH) {
        uid = app_id;
    }

    for (i = 0; property_perms[i].prefix; i++) {
        if (strncmp(property_perms[i].prefix, name,
                    strlen(property_perms[i].prefix)) == 0) {
            if ((uid && property_perms[i].uid == uid) ||
                (gid && property_perms[i].gid == gid)) {

                return check_mac_perms(name, sctx);
            }
        }
    }

    return 0;
}
```

主要也是在查表，property_perms表的定义如下：

```
struct {
    const char *prefix;
    unsigned int uid;
    unsigned int gid;
} property_perms[] = {
    { "net.rmnet0.",        AID_RADIO,    0 },
    { "net.gprs.",          AID_RADIO,    0 },
    { "net.ppp",            AID_RADIO,    0 },
    { "net.qmi",            AID_RADIO,    0 },
    { "net.lte",            AID_RADIO,    0 },
    { "net.cdma",           AID_RADIO,    0 },
    { "ril.",               AID_RADIO,    0 },
    { "gsm.",               AID_RADIO,    0 },
    { "persist.radio",      AID_RADIO,    0 },
    { "net.dns",            AID_RADIO,    0 },
    { "sys.usb.config",     AID_RADIO,    0 },
    { "net.",               AID_SYSTEM,   0 },
    { "dev.",               AID_SYSTEM,   0 },
    { "runtime.",           AID_SYSTEM,   0 },
    { "hw.",                AID_SYSTEM,   0 },
    { "sys.",               AID_SYSTEM,   0 },
    { "sys.powerctl",       AID_SHELL,    0 },
    { "service.",           AID_SYSTEM,   0 },
    { "wlan.",              AID_SYSTEM,   0 },
    { "bluetooth.",         AID_BLUETOOTH,    0 },
    { "dhcp.",              AID_SYSTEM,   0 },
    { "dhcp.",              AID_DHCP,     0 },
    { "debug.",             AID_SYSTEM,   0 },
    { "debug.",             AID_SHELL,    0 },
    { "log.",               AID_SHELL,    0 },
    { "service.adb.root",   AID_SHELL,    0 },
    { "service.adb.tcp.port", AID_SHELL,      0 },
    { "persist.sys.",       AID_SYSTEM,   0 },
    { "persist.service.",   AID_SYSTEM,   0 },
    { "persist.security.",  AID_SYSTEM,   0 },
    { "persist.service.bdroid.", AID_BLUETOOTH,   0 },
    { "selinux."          , AID_SYSTEM,   0 },
    { "wc_transport.",      AID_BLUETOOTH,    AID_SYSTEM },
#ifdef DOLBY_UDC
    { "dolby.audio",        AID_MEDIA,    0 },
#endif // DOLBY_END
#ifdef DOLBY_DAP
    // used for setting Dolby specific properties
    { "dolby.", AID_SYSTEM,   0 },
#endif // DOLBY_END
    { "usb.uicc.", AID_SYSTEM,  0 },
    { NULL, 0, 0 }
} ; end property_perms ;
```


这其实很容易理解，比如要设置“sys.”打头的系统属性，进程的uid就必须是AID_SYSTEM，否则阿猫阿狗都能设置系统属性，岂不糟糕。

2.2.2.2 property_set()

权限检查通过之后，就可以真正设置属性了。在前文“概述”一节中，我们已经说过，只有Property Service（即init进程）可以写入属性值，而普通进程最多只能通过socket向Property Service发出设置新属性值的请求，最终还得靠Property Service来写。那么我们就来看看Property Service里具体是怎么写的。

总体说来，property_set()会做如下工作：

1） 判断待设置的属性名是否合法；
2） 尽力从“属性共享内存”中找到匹配的prop_info节点，如果能找到，就调用__system_property_update()，当然如果属性是以“ro.”打头的，说明这是个只读属性，此时不会update的；如果找不到，则调用__system_property_add()添加属性节点。
3） 在update或add动作之后，还需要做一些善后处理。比如，如果改动的是“net.”开头的属性，那么需要重新设置一下net.change属性，属性值为刚刚设置的属性名字。
4） 如果要设置persist属性的话，只有在系统将所有的默认persist属性都加载完毕后，才能设置成功。persist属性应该是那种会存入可持久化文件的属性，这样，系统在下次启动后，可以将该属性的初始值设置为系统上次关闭时的值。
5） 如果将“selinux.reload_policy”属性设为“1”了，那么会进一步调用selinux_reload_policy()。这个意味着要重新加载SEAndroid策略。
6） 最后还需调用property_changed()函数，其内部会执行init.rc中指定的那些和property同名的action。

【core/init/Property_service.c】

```
int property_set(const char *name, const char *value)
{
    . . . . . .
    . . . . . .
    pi = (prop_info*) __system_property_find(name);

    if(pi != 0) {
        if(!strncmp(name, "ro.", 3)) return -1;
        __system_property_update(pi, value, valuelen);
    } else {
        ret = __system_property_add(name, namelen, value, valuelen);
        . . . . . .
    }

    if (strncmp("net.", name, strlen("net.")) == 0)  {
        if (strcmp("net.change", name) == 0) {
            return 0;
```

```
        }
        property_set("net.change", name);
    } else if (persistent_properties_loaded &&
            strncmp("persist.", name, strlen("persist.")) == 0) {
        write_persistent_property(name, value);
    } else if (strcmp("selinux.reload_policy", name) == 0 &&
               strcmp("1", value) == 0) {
        selinux_reload_policy();
    }
    property_changed(name, value);
    return 0;
}
```

一开始当然要先找到“希望设置的目标属性”在共享内存里对应的prop_info节点啦，后续关于__system_property_update()和__system_property_add()的操作，主要都是在操作该prop_info节点，代码比较简单。prop_info的详细内容我们会在下文阐述，这里先跳过。

如果可以找到prop_info节点，就尽量将这个属性的值更新一下，除非是遇到“ro.”属性，这种属性是只读的，当然不能set。如果找不到prop_info节点，此时会为这个新属性创建若干字典树节点，包括最终的prop_info叶子。

属性写入完毕后，还要调用property_changed()，做一些善后处理：

【system/core/init/Init.c】

```
void property_changed(const char *name, const char *value)
{
    if (property_triggers_enabled)
        queue_property_triggers(name, value);
}
```

【 system/core/init/Init_parser.c 】

```
void queue_property_triggers(const char *name, const char *value)
{
    struct listnode *node;
    struct action *act;
    list_for_each(node, &action_list) {
        act = node_to_item(node, struct action, alist);
        if (!strncmp(act->name, "property:", strlen("property:"))) {
            const char *test = act->name + strlen("property:");
            int name_length = strlen(name);

            if (!strncmp(name, test, name_length) &&
                    test[name_length] == '=' &&
                    (!strcmp(test + name_length + 1, value) ||
                     !strcmp(test + name_length + 1, "*"))) {
                action_add_queue_tail(act);
            }
        }
    }
}
```

```
void action_add_queue_tail(struct action *act)
{
    if (list_empty(&act->qlist)) {
        list_add_tail(&action_queue, &act->qlist);
    }
}
```

从代码可以看出，当某个属性修改之后，Property Service 会遍历一遍 action_list 列表，找到其中匹配的 action 节点，并将之添加进 action_queue 队列。之所以会有 if (list_empty(&act->qlist)) 判断，是为了防止重复添加。下面是 init.rc 脚本中的一个片段：

【system/core/rootdir/init.rc】

```
on property:vold.decrypt=trigger_reset_main
    class_reset main

on property:vold.decrypt=trigger_load_persist_props
    load_persist_props

on property:vold.decrypt=trigger_post_fs_data
    trigger post-fs-data

on property:vold.decrypt=trigger_restart_min_framework
    class_start main
```

这几个就是和property相关的action，其他相关的action还有不少，我们就不列了。我们以第一个action为例来说明。如果我们修改了vold.decrypt属性的值，那么queue_property_triggers()搜索action_list时，就能找到一个名为“property:vold.decrypt=trigger_reset_main”的action节点，此时的逻辑无非是比较“冒号后的名字”、“赋值号后的值”，是否分别和queue_property_triggers()的name、value参数匹配，如果匹配，就把这个action节点添加进action_queue队列里。

3 客户进程访问属性的机制

3.1 映射“属性共享内存”的时机

现在有一个问题必须先提出来，那就是“属性共享内存”是在什么时刻映射进用户进程空间的？总不会平白无故地就可以成功调用property_get()吧。其实，为了让大家方便地调用property_get()，属性机制的设计者的确是用了一点儿小技巧，下面我们就来看看细节。

3.1.1 静态加载时的初始化

在前文介绍Init进程初始化属性共享内存时，调用了一个叫做__system_property_area_init()的函数：

【bionic/libc/bionic/System_properties.c】

```
int __system_property_area_init()
{
    return map_prop_area_rw();
}
```

它映射时需要的是读写权限。而对普通进程而言，只有读权限，当然不可能调用__system_property_area_init()了。其实在System_properties.c文件中，我们还可以找到另一个长得挺像的初始化函数——__system_properties_init()：

```
int __system_properties_init()
{
    return map_prop_area();
}
```

它调用的map_prop_area()会把属性共享内存，以只读模式映射到用户进程空间：

```
static int map_prop_area()
{
    fd = open(property_filename, O_RDONLY | O_NOFOLLOW | O_CLOEXEC);
    . . . . . .
    if ((fd < 0) && (errno == ENOENT)) {
        fd = get_fd_from_env();
        fromFile = false;
    }

    . . . . . .
    pa_size = fd_stat.st_size;
    pa_data_size = pa_size - sizeof(prop_area);
    prop_area *pa = mmap(NULL, pa_size, PROT_READ, MAP_SHARED, fd, 0);
    . . . . . .
    result = 0;
    __system_property_area__ = pa;
    . . . . . .

    return result;
}
```

其中调用的get_fd_from_env()的代码如下：

```
static int get_fd_from_env(void)
{
    char *env = getenv("ANDROID_PROPERTY_WORKSPACE");
    if (!env) {
        return -1;
    }
    return atoi(env);
}
```

哇，终于看到读取“ANDROID_PROPERTY_WORKSPACE”环境变量的地方啦。不过呢，它的重要性似乎并没有我们一开始想的那么大。在map_prop_area()函数里分明写着，只有在open()属性文件不成功的情况下，才会尝试从环境变量中读取文件句柄，而一般都会open成功的。不管文件句柄fd是怎么得到的吧，反正能映射成空间地址就行。映射后的空间地址，仍然会记录在__system_property_area__全局变量中。

现在我们只需找到调用__system_properties_init()的源头就可以了。经过查找，我们发现__libc_init_common()会调用它，代码如下：

【bionic/libc/bionic/Libc_init_common.cpp】

```
void __libc_init_common(KernelArgumentBlock& args) {
  . . . . . .
  . . . . . .
  _pthread_internal_add(main_thread);
  __system_properties_init(); // Requires 'environ'.
}
```

这个函数可是在bionic目录里的，小技巧已经用到C库里啦。

__libc_init_common()又会被__libc_init()调用：

【bionic/libc/bionic/Libc_init_static.cpp】

```
__noreturn void __libc_init(void* raw_args,
                            void (*onexit)(void),
                            int (*slingshot)(int, char**, char**),
                            structors_array_t const * const structors) {
  KernelArgumentBlock args(raw_args);
  __libc_init_tls(args);
  __libc_init_common(args);
  . . . . . .
  . . . . . .
  call_array(structors->preinit_array);
  call_array(structors->init_array);
  . . . . . .
  exit(slingshot(args.argc, args.argv, args.envp));
}
```

当一个用户进程被调用起来时，内核会先调用到C运行期库（crtbegin）层次来初始化运行期环境，在这个阶段就会调用到__libc_init()，而后才会间接调用到C程序员熟悉的main()函数。可见属性共享内存在执行main()函数之前就已经映射好了。

3.1.2 动态加载时的初始化

除了__libc_init()中会调用__libc_init_common()，还有一处会调用。

【bionic/libc/bionic/Libc_init_dynamic.cpp】

```
__attribute__((constructor)) static void __libc_preinit() {
  . . . . . .
  __libc_init_common(*args);
  . . . . . .
  pthread_debug_init();
  malloc_debug_init();
}
```

请大家注意函数名那一行起始处的__attribute__((constructor))属性，这是GCC的一个特有属性。被这种属性修饰的函数会被放置在特殊的代码段中。这样，当动态链接器一加载libc.so时，会尽早执行__libc_preinit()函数。这样一来，动态库里也可以放心调用property_get()了。

3.2 读取属性值

下面我们来集中精力研究读取属性值的部分。我们在前文留下过一个尾巴，当时对属性共享内存块里的prop_info节点，只做了非常简略的提及，现在我们就来细说它。

说白了，属性共享内存中的内容，其实被组织成一棵字典树。内存块的第一个节点是个特殊的总述节点，类型

为prop_area。紧随其后的就是字典树的“树枝”和“树叶”了，树枝以prop_bt表达，树叶以prop_info表达。我们读取或设置属性值时，最终都只是在操作“叶子”节点而已。

3.2.1 “属性共享内存”里的数据结构

【bionic/libc/bionic/System_properties.c】

```
struct prop_area {
    unsigned bytes_used;
    unsigned volatile serial;
    unsigned magic;
    unsigned version;
    unsigned reserved[28];
    char data[0];
};

typedef struct prop_area prop_area;

struct prop_info {
    unsigned volatile serial;
    char value[PROP_VALUE_MAX];
    char name[0];
};

typedef struct prop_info prop_info;
```

```
typedef volatile uint32_t prop_off_t;
struct prop_bt {
    uint8_t namelen;
    uint8_t reserved[3];

    prop_off_t prop;

    prop_off_t left;
    prop_off_t right;

    prop_off_t children;

    char name[0];
};

typedef struct prop_bt prop_bt;
```

现在的问题是，这棵树是如何组织其枝叶的？System_properties.c文件中，有一段注释，给出了一个不算太清楚的示意图，截取如下：

```
/*
 * Properties are stored in a hybrid trie/binary tree structure.
 * Each property's name is delimited at '.' characters, and the tokens are put
 * into a trie structure.  Siblings at each level of the trie are stored in a
 * binary tree.  For instance, "ro.secure"="1" could be stored as follows:
 *
 * +-----+   children    +----+   children    +--------+
 * |     |-------------->| ro |-------------->| secure |
 * +-----+               +----+               +--------+
 *                       /    \                /   |
 *                 left /      \ right   left /    |  prop   +===========+
 *                     v        v            v     +-------->| ro.secure |
 *                  +-----+   +-----+     +-----+            +-----------+
 *                  | net |   | sys |     | com |            |     1     |
 *                  +-----+   +-----+     +-----+            +===========+
 */
```


看过这张图后，各位同学搞清楚了吗？反正我一开始没有搞清楚，后来只好研究代码，现在算是知道一点儿了，详情如下：

- 一开始的prop_area节点严格地说并不属于字典树，但是它代表着属性共享内存块的起始；
- 紧接着prop_area节点，需要有一个空白的prop_bt节点。这个是必须的噢，在前文说明init进程的main()函数的调用关系图中，我们表达了这个概念：

```
pa->bytes_used = sizeof(prop_bt);  必须空出一个prop_bt节点！
```


这个就是空节点；

- 属性名将以‘.’符号为分割符，被分割开来。比如ro.secure属性名就会被分割成“ro”和“secure”两部分，而且每个部分用一个prop_bt节点表达。
- 属性名中的这种‘.’关系被表示为父子关系，所以“ro”节点的children域，会指向“secure”节点。但是请注意，一个节点只有一个children域，如果它还有其他孩子，那些孩子将会和第一个子节点（比如secure节点）组成一棵二叉树。
- 当一个属性名对应的“字典树枝”都已经形成好后，会另外创建一个prop_info节点，专门表示这个属性，该节点就是“字典树叶”。

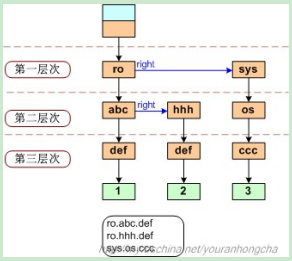
下面我们画几张图来说明问题。比如我们现在手头有3个属性，分别为

ro.abc.def

ro.hhh.def

sys.os.ccc

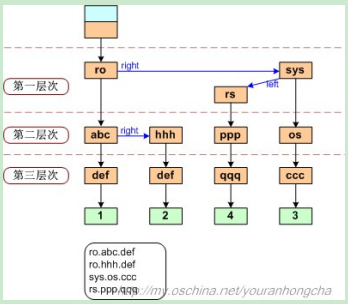
我们依此顺序设置属性，就会形成下面这样的树：



其中天蓝色块表示prop_area节点，桔黄色块表示prop_bt节点，浅绿色块表示prop_info节点。简单地说，父节点的children域，只指代其第一个子节点。后续从属于同一父节点的兄弟子节点，会被组织成一棵二叉子树，该二叉子树的根就是父节点的第一个子节点。我们用蓝色箭头来表示二叉子树的关系，在代码中对应prop_bt的left、right域。这么说来，以不同顺序添加属性，其实会导致最终得到的字典树在形态上发生些许变化。

prop_bt节点的name域只记录“树枝”的名字，比如“ro”、“abc”、“def”等等，而prop_info节点的name域记录的则是属性的全名，比如“ro.abc.def”。

现在我们向上面这棵字典树中再添加一个rs.ppp.qqq属性，会形成如下字典树：



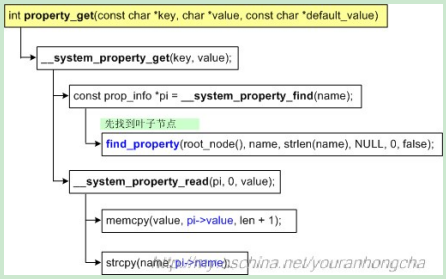
“rs”节点之所以在那个位置，是基于strcmp()的计算结果。“rs”字符串比“ro”字符串大，所以进一步和“ro”的right节点（即“sys”节点）比对，“rs”又比“sys”小，所以在“sys”节点的left枝上建立了新节点。

以上是画成字典树的样子，它表示的是一种逻辑关系。而在实际的“属性共享内存”中，这些节点基本上是紧凑排列的，大体上会形成下面这样的排列关系：

说到这里，大家应该已经比较清楚属性共享内存块是怎么组织的吧。有了这种大致思路，再去看相应的代码，相信大家会轻松一点儿。

3.2.2 property_get()

在读取具体属性值时，最终会调用到property_get()函数，该函数的调用关系如下：



说白了就是先从字典树中找到感兴趣的prop_info叶子，然后把叶子里的值读出来。

4 Java层的封装

接下来我们再说说属性机制里Java层的封装。这部分比较简单，因为它主要只是在简单包装C语言层次的函数。

Java层使用的属性机制被封装在SystemProperties中：

【frameworks/base/core/java/android/os/SystemProperties.java】

```
public class SystemProperties
{
    public static final int PROP_NAME_MAX = 31;
    public static final int PROP_VALUE_MAX = 91;

    private static final ArrayList<Runnable> sChangeCallbacks = new ArrayList<Runnable>();

    private static native String native_get(String key);
    private static native String native_get(String key, String def);
    private static native int native_get_int(String key, int def);
    private static native long native_get_long(String key, long def);
    private static native boolean native_get_boolean(String key, boolean def);
    private static native void native_set(String key, String def);
    private static native void native_add_change_callback();

/**
     * Get the value for the given key.
     * @return an empty string if the key isn't found
     * @throws IllegalArgumentException if the key exceeds 32 characters
     */
    public static String get(String key) {
        if (key.length() > PROP_NAME_MAX) {
            throw new IllegalArgumentException("key.length > " + PROP_NAME_MAX);
        }
        return native_get(key);
    }
    . . . . . .
    . . . . . .
```

我们就以上面的get()成员函数为例来说明，它基本上只是在调用native_get()函数而已，该函数对应的C语言函数可以从下表查到，就是那个SystemProperties_getS()：

【frameworks/base/core/jni/android_os_SystemProperties.cpp】

```
static JNINativeMethod method_table[] = {
    { "native_get", "(Ljava/lang/String;)Ljava/lang/String;",
      (void*) SystemProperties_getS },
    { "native_get", "(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;",
      (void*) SystemProperties_getSS },
    { "native_get_int", "(Ljava/lang/String;I)I",
      (void*) SystemProperties_get_int },
    { "native_get_long", "(Ljava/lang/String;J)J",
      (void*) SystemProperties_get_long },
    { "native_get_boolean", "(Ljava/lang/String;Z)Z",
      (void*) SystemProperties_get_boolean },
    { "native_set", "(Ljava/lang/String;Ljava/lang/String;)V",
      (void*) SystemProperties_set },
    { "native_add_change_callback", "()V",
      (void*) SystemProperties_add_change_callback },
};
```

【frameworks/base/core/jni/android_os_SystemProperties.cpp】

```
static jstring SystemProperties_getS(JNIEnv *env, jobject clazz,
                                     jstring keyJ)
{
    return SystemProperties_getSS(env, clazz, keyJ, NULL);
}

static jstring SystemProperties_getSS(JNIEnv *env, jobject clazz,
                                     jstring keyJ, jstring defJ)
{
    int len;
    const char* key;
    char buf[PROPERTY_VALUE_MAX];
    jstring rvJ = NULL;

    if (keyJ == NULL) {
        jniThrowNullPointerException(env, "key must not be null.");
        goto error;
    }

    key = env->GetStringUTFChars(keyJ, NULL);

    len = property_get(key, buf, "");
    if ((len <= 0) && (defJ != NULL)) {
        rvJ = defJ;
    } else if (len >= 0) {
        rvJ = env->NewStringUTF(buf);
    } else {
        rvJ = env->NewStringUTF("");
    }

    env->ReleaseStringUTFChars(keyJ, key);

error:
    return rvJ;
}
```

最终调用的还是property_get()函数。

5 尾声

至此，有关Android属性机制的大体机理就讲解完毕了，希望对大家有点儿帮助。

‹ 上一篇： Android4.4的init进程

下一篇： 写个简单的飞机游戏玩玩 ›

悠然红茶

粉丝 324 博文 19 码字总数 99747 作品 0

西安 高级程序员

关注 私信 提问

评论(10)

LiDong 2016/03/21 16:04

牛哥!

举报

klzhong 2015/03/23 11:22

> **引用来自“小峰”的评论**
> 写的很好啊，深入学习了。

点击此处输入评论

举报

beason 2015/03/23 07:47

有什么作用

举报

小峰 2015/03/23 00:06

写的很好啊，深入学习了。

举报

悠然红茶 2015/03/22 13:22

> **引用来自“tieyan”的评论**
> 请问楼主， 你用什么工具画这些图。谢谢

我使用visio画图。

举报

tieyan 2015/03/22 13:13

请问楼主， 你用什么工具画这些图。谢谢

举报

gaoxiaoyuan 2015/03/22 11:30

厉害，学习了

举报

maplewang 2015/03/22 11:16

不错哦，收藏

举报

随意、nice 2015/03/22 00:54

android新手，虽然看不懂，但是还是收藏了

举报

信阳农夫 2015/03/21 22:02

不错，收藏了

举报

相关文章 最新文章

Android 性能优化：手把手教你优化Bitmap图片资源的使用

前言 在 开发中，性能优化策略十分重要 本文主要讲解性能优化中的Bitmap 使用优化，希望你们会喜欢 目录 1. 优化原因 即 为什么要优化图片资源，具体如下图：

Carson_Ho 04/24 0 0

SystemProperties源码分析 获取 Android WIFI接口

SystemProperties.java这个类是@hide的，所以不对外公开，一般开发者是访问不到的，但是我们可以通过反射机制来使用。通过反射取得wifi的接口名为例！ 下面通过Android源码讲解SystemPropert...

wangxigui 2013/07/12 0 0

创想汇【大牛面对面】——Android GUI系统高阶培训

这个八月 来自北京opera技术大咖将带你一起"修行"，练就"火眼金睛"破解Android图形的百般变化， 传授"72变"帮你轻松应对各种bug事件， 来吧程序猿们！ 来取属于你的定海神针！！！ 组...

喵一喵 2015/07/31 323 4

转载：最强最全干货分享：Android开发书籍、教程、工具等

最全干货分享，本文收集整理了Android开发所需的书籍、教程、工具、资讯和周刊各种资源，它们能让你在Android开发之旅的各个阶段都受益。 入门 《Learning Android (中文版) 》 本书为Andro...

拉偶有所依 2015/01/09 152 3

最强最全干货分享：Android开发书籍、教程、工具等

拉偶有所依 2015/01/09 0 1

加载更多