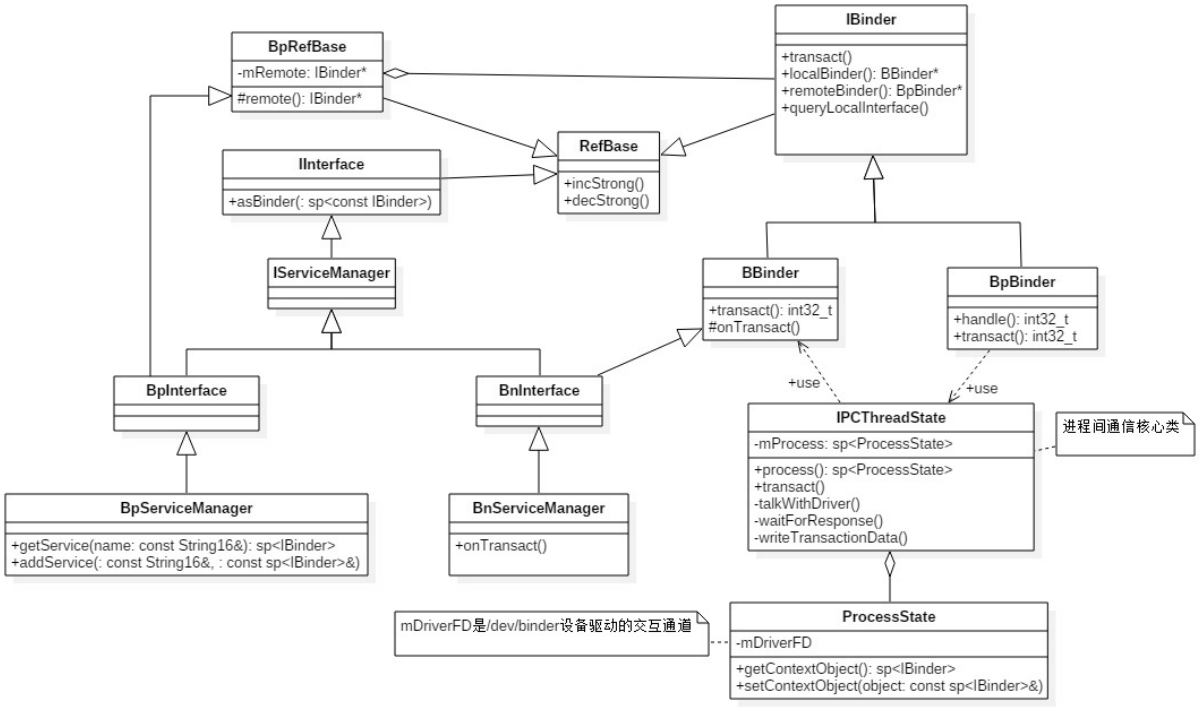
Binder系列2—启动Service Manager

Nov 7, 2015

> 基于Android 6.0的源码剖析， 本文详细地讲解了Service Manager如何产生

类关系图

先来一张整个native层中所涉及类的关系图，接下来几篇文章会详细介绍下图中相关类。

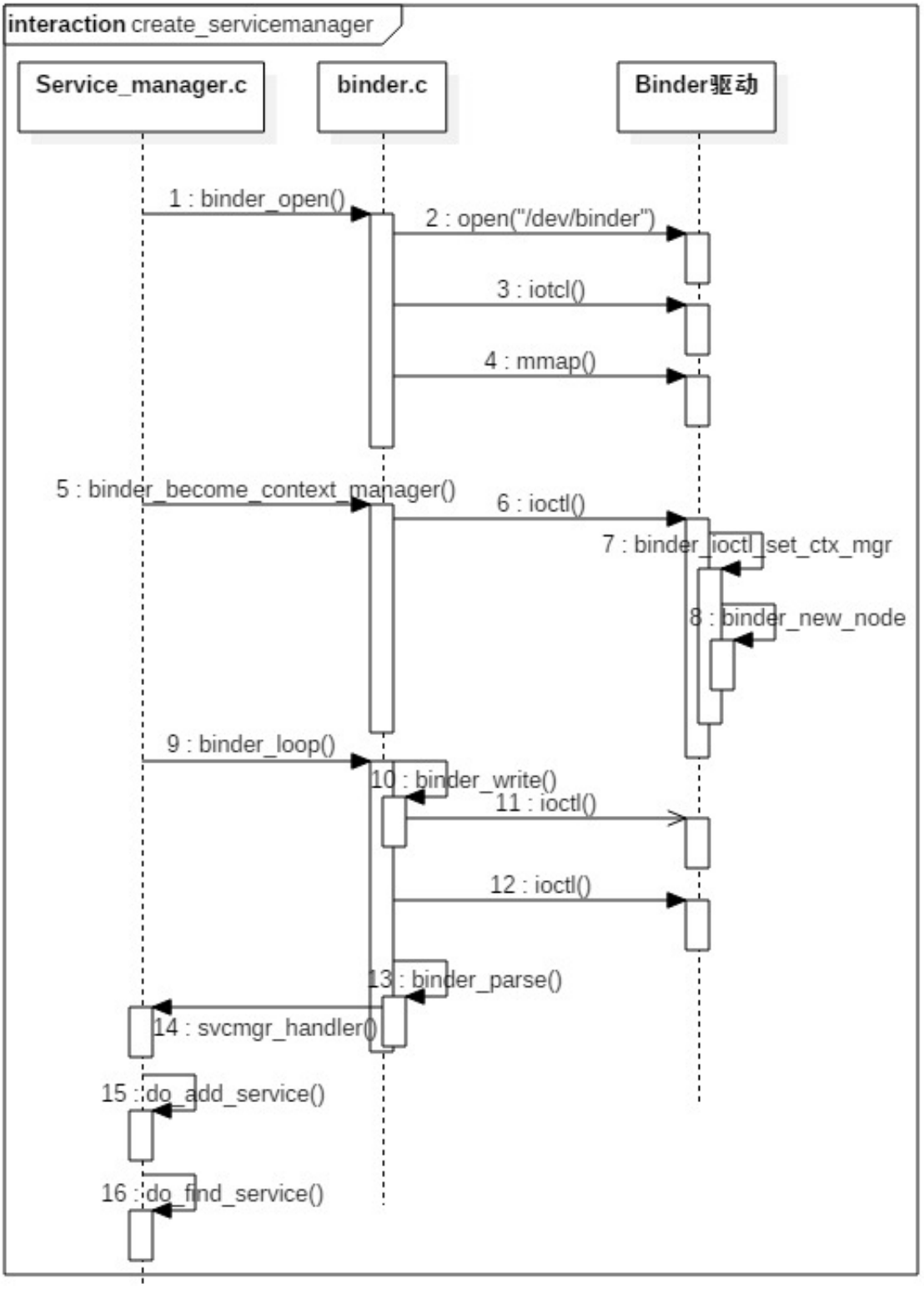


源码分析

相关源码

```
/framework/native/cmds/servicemanager/service_manager.c
/framework/native/cmds/servicemanager/binder.c
/kernel/drivers/android/binder.c
```

时序图



先展示时序图，让大家对整个流程有一个大致的了解，下面将开始正式介绍整个时序图中每个流程的主要工作。结合时序图，来看下面的介绍，理解起来会比较方便。

入口

```
==> /framework/native/cmds/servicemanager/service_manager.c
```

service manager的主方法入口

```
int main(int argc, char **argv)
{
    struct binder_state *bs;

    bs = binder_open(128*1024);  //打开binder驱动，申请128k大小的内存空间【见流程1】
    if (!bs) {
        return -1;
    }

    if (binder_become_context_manager(bs)) {  //成为上下文管理者  【见流程5】
        return -1;
    }

    selinux_enabled = is_selinux_enabled(); //判断selinux权限问题
    sehandle = selinux_android_service_context_handle();
    selinux_status_open(true);

    if (selinux_enabled > 0) {
        if (sehandle == NULL) {  //无法获取sehandle
            abort();
        }

        if (getcon(&service_manager_context) != 0) { //无法获取service_manager上下文
            abort();
        }
    }

    union selinux_callback cb;
    cb.func_audit = audit_callback;
    selinux_set_callback(SELINUX_CB_AUDIT, cb);
    cb.func_log = selinux_log_callback;
    selinux_set_callback(SELINUX_CB_LOG, cb);

    binder_loop(bs, svcmgr_handler); //进入无限循环，处理client端发来的请求 【见流程9】

    return 0;
}
```

主要分为4个步骤：

- 打开binder设备 binder_open()；
- binder成为守护进程 binder_become_context_manager()；
- 验证selinux权限；
- 进入无限循环，等待Client的连接。

[1] binder_open

==> /framework/native/cmds/servicemanager/binder.c

打开binder驱动相关操作

```
struct binder_state *binder_open(size_t mapsize)
{
    struct binder_state *bs;
    struct binder_version vers;

    bs = malloc(sizeof(*bs));
    if (!bs) {
        errno = ENOMEM;
        return NULL;
    }

    bs->fd = open("/dev/binder", O_RDWR);  //通过系统调用，陷入内核，打开Binder设备驱动 【流程2】
    if (bs->fd < 0) {
        goto fail_open; // 无法打开binder设备
    }

     //通过系统调用，ioctl获取binder版本信息【流程3】
    if ((ioctl(bs->fd, BINDER_VERSION, &vers) == -1) ||
        (vers.protocol_version != BINDER_CURRENT_PROTOCOL_VERSION)) {
        goto fail_open; //内核空间与用户空间的binder不是同一版本
    }

    bs->mapsize = mapsize;
        //通过系统调用，mmap内存映射 【流程4】
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
    if (bs->mapped == MAP_FAILED) {
        goto fail_map; // binder设备内存无法映射
    }

    return bs;

fail_map:
    close(bs->fd);
fail_open:
    free(bs);
    return NULL;
}
```

binder_open功能是打开binder设备，并把binder内存映射，fd记录binder设备描述符。对于流程图中的2、3、4步骤，都是通过系统调用，最后是调用Binder驱动方法，关于binder驱动，Binder系列1 —— Binder驱动(http://www.yuanhh.com/2015/11/01/binder-driver/)中有详细说明。

[5] binder_become_context_manager

==> /framework/native/cmds/servicemanager/binder.c

成为上下文的管理者，整个系统中只有一个这样的管理者。

```
int binder_become_context_manager(struct binder_state *bs)
{
        //通过ioctl，传递BINDER_SET_CONTEXT_MGR指令。再调用【流程7】
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

[7] binder_ioctl_set_ctx_mgr

==> kernel/drivers/android/binder.c

binder驱动操作

```
static int binder_ioctl_set_ctx_mgr(struct file *filp)
{
        int ret = 0;
        struct binder_proc *proc = filp->private_data;
        kuid_t curr_euid = current_euid();

        if (binder_context_mgr_node != NULL) {
                ret = -EBUSY;
                goto out;
        }

        if (uid_valid(binder_context_mgr_uid)) {
                if (!uid_eq(binder_context_mgr_uid, curr_euid)) {
                        ret = -EPERM;
                        goto out;
                }
        } else {
                binder_context_mgr_uid = curr_euid; //设置当前线程euid作为Service Manager的uid
        }
        binder_context_mgr_node = binder_new_node(proc, 0, 0); //创建Service Manager实体【流程8】
        if (binder_context_mgr_node == NULL) {
                ret = -ENOMEM;
                goto out;
        }
        binder_context_mgr_node->local_weak_refs++;
        binder_context_mgr_node->local_strong_refs++;
        binder_context_mgr_node->has_strong_ref = 1;
        binder_context_mgr_node->has_weak_ref = 1;
out:
        return ret;
}
```

[8] binder_new_node

==> kernel/drivers/android/binder.c

创建一个binder_node，binder_node结构体见Binder系列1 —— Binder驱动(http://www.yuanhh.com/2015/11/01/binder-driver/)中3.2小节。

```
static struct binder_node *binder_new_node(struct binder_proc *proc,
                                           binder_uintptr_t ptr,
                                           binder_uintptr_t cookie)
{
        struct rb_node **p = &proc->nodes.rb_node;
        struct rb_node *parent = NULL;
        struct binder_node *node;
        //首次进来为空
        while (*p) {
                parent = *p;
                node = rb_entry(parent, struct binder_node, rb_node);

                if (ptr < node->ptr)
                        p = &(*p)->rb_left;
                else if (ptr > node->ptr)
                        p = &(*p)->rb_right;
                else
                        return NULL;
        }

        node = kzalloc(sizeof(*node), GFP_KERNEL); //分配内核空间
        if (node == NULL)
                return NULL;
        binder_stats_created(BINDER_STAT_NODE);
        rb_link_node(&node->rb_node, parent, p);
        rb_insert_color(&node->rb_node, &proc->nodes);
        node->debug_id = ++binder_last_id;
        node->proc = proc;
        node->ptr = ptr;
        node->cookie = cookie;
        node->work.type = BINDER_WORK_NODE;
        INIT_LIST_HEAD(&node->work.entry);
        INIT_LIST_HEAD(&node->async_todo);
        return node;
}
```

[9] binder_loop

==> /framework/native/cmds/servicemanager/binder.c

进入循环读写操作

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
    int res;
    struct binder_write_read bwr;
    uint32_t readbuf[32];

    bwr.write_size = 0;
    bwr.write_consumed = 0;
    bwr.write_buffer = 0;

    readbuf[0] = BC_ENTER_LOOPER;
    //将BC_ENTER_LOOPER命令发送给binder驱动，让Service Manager进入循环【流程10】
    binder_write(bs, readbuf, sizeof(uint32_t));

    for (;;) {
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (uintptr_t) readbuf;

        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr); //进入循环，不断地binder读写过程
        if (res < 0) {
            break;
        }

        // 解析binder信息 【流程13】
        res = binder_parse(bs, 0, (uintptr_t) readbuf, bwr.read_consumed, func);
        if (res == 0) {
            break;
        }
        if (res < 0) {
            break;
        }
    }
}
```

[10] binder_write

==> /framework/native/cmds/servicemanager/binder.c

```
int binder_write(struct binder_state *bs, void *data, size_t len)
{
    struct binder_write_read bwr;
    int res;

    bwr.write_size = len;
    bwr.write_consumed = 0;
    bwr.write_buffer = (uintptr_t) data; //此处data为BC_ENTER_LOOPER
    bwr.read_size = 0;
    bwr.read_consumed = 0;
    bwr.read_buffer = 0;
    res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);

    return res;
}
```

初始化bwr，将BC_ENTER_LOOPER命令，bwr地址，发送给binder驱动，让Service Manager进入循环。

[13] binder_parse

==> /framework/native/cmds/servicemanager/binder.c

解析binder信息，此处参数ptr指向BC_ENTER_LOOPER。

```
int binder_parse(struct binder_state *bs, struct binder_io *bio,
                 uintptr_t ptr, size_t size, binder_handler func)
{
    int r = 1;
    uintptr_t end = ptr + (uintptr_t) size;

    while (ptr < end) {
        uint32_t cmd = *(uint32_t *) ptr;
        ptr += sizeof(uint32_t);
        switch(cmd) {
        case BR_NOOP:  //无操作，退出循环
            break;
        case BR_TRANSACTION_COMPLETE:
            break;
        case BR_INCREFS:
        case BR_ACQUIRE:
        case BR_RELEASE:
        case BR_DECREFS:
            ptr += sizeof(struct binder_ptr_cookie);
            break;
        case BR_TRANSACTION: {
            struct binder_transaction_data *txn = (struct binder_trans
action_data *) ptr;
            if ((end - ptr) < sizeof(*txn)) {
                ALOGE("parse: txn too small!\n");
                return -1;
            }
            binder_dump_txn(txn);
            if (func) {
                unsigned rdata[256/4];
                struct binder_io msg;
                struct binder_io reply;
                int res;

                bio_init(&reply, rdata, sizeof(rdata), 4);
                bio_init_from_txn(&msg, txn);
                res = func(bs, txn, &msg, &reply); // 收到Binder事务【见
流程14】
                binder_send_reply(bs, &reply, txn->data.ptr.buffer, re
s);
            }
            ptr += sizeof(*txn);
            break;
        }
        case BR_REPLY: {
            struct binder_transaction_data *txn = (struct binder_trans
action_data *) ptr;
            if ((end - ptr) < sizeof(*txn)) {
```

```
                ALOGE("parse: reply too small!\n");
                return -1;
            }
            binder_dump_txn(txn);
            if (bio) {
                bio_init_from_txn(bio, txn);
                bio = 0;
            } else {
                /* todo FREE BUFFER */
            }
            ptr += sizeof(*txn);
            r = 0;
            break;
        }
        case BR_DEAD_BINDER: {
            struct binder_death *death = (struct binder_death *)(uintp
tr_t) *(binder_uintptr_t *)ptr;
            ptr += sizeof(binder_uintptr_t);
            death->func(bs, death->ptr); // binder死亡消息【见流程14】
            break;
        }
        case BR_FAILED_REPLY:
            r = -1;
            break;
        case BR_DEAD_REPLY:
            r = -1;
            break;
        default:
            return -1;
        }
    }

    return r;
}
```

此处func函数指针 指向 svcmgr_handler，将接受到的请求，最终调用svcmgr_handler。

[14] svcmgr_handler

==> /framework/native/cmds/servicemanager/service_manager.c

service manager操作的真正处理函数

```
int svcmgr_handler(struct binder_state *bs,
                   struct binder_transaction_data *txn,
                   struct binder_io *msg,
                   struct binder_io *reply)
{
    struct svcinfo *si;
    uint16_t *s;
    size_t len;
    uint32_t handle;
    uint32_t strict_policy;
    int allow_isolated;

    if (txn->target.ptr != BINDER_SERVICE_MANAGER) //判断target是否是Ser
vice Manager
        return -1;

    if (txn->code == PING_TRANSACTION)
        return 0;

    strict_policy = bio_get_uint32(msg);
    s = bio_get_string16(msg, &len);
    if (s == NULL) {
        return -1;
    }
    //svcmgr_id是由“android.os.IServiceManager”字符组成的。svcmgr_id与s的
内存块的内容是否一致。
    if ((len != (sizeof(svcmgr_id) / 2)) ||
        memcmp(svcmgr_id, s, sizeof(svcmgr_id))) {
        return -1;
    }

    if (sehandle && selinux_status_updated() > 0) {
        struct selabel_handle *tmp_sehandle = selinux_android_servic
e_context_handle();
        if (tmp_sehandle) {
            selabel_close(sehandle);
            sehandle = tmp_sehandle;
        }
    }

    switch(txn->code) {
    case SVC_MGR_GET_SERVICE:  //对应于getService
    case SVC_MGR_CHECK_SERVICE:  //对应于checkService
        s = bio_get_string16(msg, &len);
        if (s == NULL) {
            return -1;
        }
        handle = do_find_service(bs, s, len, txn->sender_euid, txn->se
```

```
nder_pid); //根据名称查找service 【见流程15】
        if (!handle)
            break;
        bio_put_ref(reply, handle);
        return 0;

    case SVC_MGR_ADD_SERVICE:  //对应于addService
        s = bio_get_string16(msg, &len);
        if (s == NULL) {
            return -1;
        }
        handle = bio_get_ref(msg);
        allow_isolated = bio_get_uint32(msg) ? 1 : 0;
        if (do_add_service(bs, s, len, handle, txn->sender_euid,
            allow_isolated, txn->sender_pid)) 【见流程16】
            return -1;
        break;

    case SVC_MGR_LIST_SERVICES: {   // 对应于listService
        uint32_t n = bio_get_uint32(msg);

        if (!svc_can_list(txn->sender_pid)) {
            return -1;
        }
        si = svclist;
        while ((n-- > 0) && si)
            si = si->next;
        if (si) {
            bio_put_string16(reply, si->name);
            return 0;
        }
        return -1;
    }
    default:
        return -1;
    }

    bio_put_uint32(reply, 0);
    return 0;
}
```

[15] do_add_service

==> /framework/native/cmds/servicemanager/service_manager.c

注册服务

```
int do_add_service(struct binder_state *bs,
                   const uint16_t *s, size_t len,
                   uint32_t handle, uid_t uid, int allow_isolated,
                   pid_t spid)
{
    struct svcinfo *si;

    if (!handle || (len == 0) || (len > 127))
        return -1;

    if (!svc_can_register(s, len, spid)) { //权限检查
        return -1;
    }

    si = find_svc(s, len); //服务检索
    if (si) {
        if (si->handle) {
            svcinfo_death(bs, si); //服务已注册时，释放相应的服务
        }
        si->handle = handle;
    } else {
        si = malloc(sizeof(*si) + (len + 1) * sizeof(uint16_t));
        if (!si) {  //内存不足，无法分配足够内存
            return -1;
        }
        si->handle = handle;
        si->len = len;
        memcpy(si->name, s, (len + 1) * sizeof(uint16_t)); //内存拷贝服
务信息
        si->name[len] = '\0';
        si->death.func = (void*) svcinfo_death;
        si->death.ptr = si;
        si->allow_isolated = allow_isolated;
        si->next = svclist; // svclist保存所有已注册的服务
        svclist = si;
    }

    //以BC_ACQUIRE命令，handle为目标的信息，通过ioctl发送给binder驱动
    binder_acquire(bs, handle);
    //以BC_REQUEST_DEATH_NOTIFICATION命令的信息，通过ioctl发送给binder驱
动，主要用于清理内存等收尾工作。
    binder_link_to_death(bs, handle, &si->death);
    return 0;
}
```

(1)检查权限

检查selinux权限是否满足，

```
static int svc_can_register(const uint16_t *name, size_t name_len, pi
d_t spid)
{
    const char *perm = "add";
    return check_mac_perms_from_lookup(spid, perm, str8(name, name_le
n)) ? 1 : 0;
}
```

(2)查询服务

从svclist服务列表中，根据服务名遍历查找是否已经注册。当服务已存在svclist，则返回相应的服务名，否则返回NULL。

```
struct svcinfo *find_svc(const uint16_t *s16, size_t len)
{
    struct svcinfo *si;

    for (si = svclist; si; si = si->next) {
        if ((len == si->len) &&
            !memcmp(s16, si->name, len * sizeof(uint16_t))) {
            return si;
        }
    }
    return NULL;
}
```

(3)释放服务

```
void svcinfo_death(struct binder_state *bs, void *ptr)
{
    struct svcinfo *si = (struct svcinfo* ) ptr;

    if (si->handle) {
        binder_release(bs, si->handle);
        si->handle = 0;
    }
}
```

[16] do_find_service

==> /framework/native/cmds/servicemanager/service_manager.c

查询服务

```
uint32_t do_find_service(struct binder_state *bs, const uint16_t *s, s
ize_t len, uid_t uid, pid_t spid)
{
    struct svcinfo *si = find_svc(s, len); //查询相应的服务

    if (!si || !si->handle) {
        return 0;
    }

    if (!si->allow_isolated) {
        uid_t appid = uid % AID_USER;
        if (appid >= AID_ISOLATED_START && appid <= AID_ISOLATED_END)
{ //检查该服务是否允许孤立于进程而单独存在。
            return 0;
        }
    }

    if (!svc_can_find(s, len, spid)) {  //服务是否满足 查询条件
        return 0;
    }

    return si->handle;
}
```

在前面注册服务的过程中，其实已经涉及了查询服务的具体方法 `find_svc` ，该方法比较简单。

小结

Service Manager成为IPC守护进程流程：

1. 打开binder驱动，并建立128k的内存映射空间：binder_open();
2. 通知binder驱动它是守护进程：binder_become_context_manager()；
3. 进入循环状态，等待请求：binder_loop()。

Service Manger意义：

1. ServiceManger能集中管理系统内的所有服务，它能施加权限控制，并不是任何进程都能注册服务；
2. ServiceManager支持通过字符串名称来查找对应的Service。这个功能很像DNS；
3. Server进程随时可能会挂了，如果让每个Client都去检测会导致负载过重。但有了ServiceManager，Client只需要查询ServiceManager，就能把握动向，得到最新信息。

喜欢

0条评论 最新 最早 最热

还没有评论，沙发等你来抢

嘿嘿参北斗哇 (http://www.baidu.com/p/嘿嘿参北斗哇) 帐号管理


(http://duoshuo.com/settings/avatar/)

说点什么吧…

分享到: 发布

多说 (http://duoshuo.com)

gityuan@gmail.com (mailto:gityuan@gmail.com) · Github

(https://github.com/yuanhuihui) · 天道酬勤 · © 2015 Yuanhh · Jekyll

(https://github.com/jekyll/jekyll) theme by HyG (https://github.com/Gaohaoyang)