Android 通信机制

Android 的 IPC 机制之 Binder 详细介绍

http://www.xxlinux.com/linux/article/development/embed/2009-01-14/14894.html

第一部分 Binder 的组成

一篇 android 的 IPC 机制 binder 实例 AudioFlinger 国外文档

http://blog.chinaunix.net/u1/38994/showart_1676822.html

Android JAVA Binder IPC System

http://blog.chinaunix.net/u1/38994/showart_1680617.html

binder 官网

http://blog.chinaunix.net/u1/38994/showart_1222450.html

Android IPC 通讯机制源码分析

http://hi.baidu.com/albertchen521/blog/item/30c32d3f4bee993a71cf6ca0.html

http://hi.baidu.com/albertchen521/blog/item/822058d0f63ea2d4562c84a1.html

Reference/android/app/Service

Android 底层库 libutils 介绍

http://www.androidin.com/learn/cn/200901/22-437.html

Android 初学

http://www.fulema.com/forumdisplay.php?fid=4

服务管理器：

Service_manager.c (frameworks\base\cmds\servicemanager):

```
int main(int argc, char **argv)
{
    struct binder_state *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;
    //打开 binder 驱动,映射一个 128*1024 字节的内存
    bs = binder_open(128*1024);
    //设置上下文为 mgr
    if (binder_become_context_manager(bs)) {
        LOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }
    svcmgr_handle = svcmgr;
    //进入主循环
    binder_loop(bs, svcmgr_handler);
```

```
        return 0;
}
```

binder_open()函数
binder_open()->open()->binder_open()
框架层的binder_open()函数调用vfs的open ，最终调用的是内核的 binder_open()函数。
框架层和内核层的binder_open()函数定义所在文件：
Binder.c (frameworks\base\cmds\servicemanager):struct binder_state *binder_open(unsigned mapsize)

Binder.c (kernel\drivers\misc):static int binder_open(struct inode *nodp, struct file *filp)
框架层binder_open()函数分析

```
struct binder_state *binder_open(unsigned mapsize)
{
        struct binder_state *bs;
        bs = malloc(sizeof(*bs));
        ...
        //打开binder驱动
        bs->fd = open("/dev/binder", O_RDWR);
        ...
        bs->mapsize = mapsize;
        //映射一个128*1024字节的内存
        bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
        ...
}
```

binder驱动程序是一个miscdevice，主设备号为10，此设备号使用动态获得(MISC_DYNAMIC_MINOR),其设备的节点为：/dev/binder

设置上下文为mgr

```
int binder_become_context_manager(struct binder_state *bs)
{
        return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

binder_loop()主循环

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
        int res;
        struct binder_write_read bwr;
        unsigned readbuf[32];
        ...
        readbuf[0] = BC_ENTER_LOOPER;
        binder_write(bs, readbuf, sizeof(unsigned));

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

                res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
        ...
        //binder 循环处理过程
        res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
    ...
    }
}
```

调用 ioctl 读取设备文件，

此处的 ioctl 最终调用的是内核 Binder.c (kernel\drivers\misc)中的驱动函数：

static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)

调用层次如下：

服务管理器->ioctl()->sys_ioctl()->do_vfs_ioctl()->file_ioctl()->vfs_ioctl()->binder_ioctl()

还应该注意以下 binder_write()函数，它最终调用的也是内核中的 binder_ioctl()。

对于 ioctl 的读取结果在 binder_parse()函数中进行处理。

binder_parse()函数分析：

当处理 BR_TRANSACTION 的时候，调用 svcmgr_handler()处理增加服务、检查服务等工作。各种服务存放在一个链表(svclist)中。其中调用 binder_等开头的函数，又会调用 ioctl 的各种命令。

处理 BR_REPLY 的时候，填充 binder_io 类型的数据结

```
int binder_parse(struct binder_state *bs, struct binder_io *bio,
                 uint32_t *ptr, uint32_t size, binder_handler func)
{
    ...
    while (ptr < end) {
    ...
        switch(cmd) {
    ...
        case BR_TRANSACTION: {
            struct binder_txn *txn = (void *) ptr;
            if ((end - ptr) * sizeof(uint32_t) < sizeof(struct binder_txn)) {
                LOGE("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);
        //此处调用的是函数    svcmgr_handler()
                res = func(bs, txn, &msg, &reply);
                binder_send_reply(bs, &reply, txn->data, res);
            }
            ptr += sizeof(*txn) / sizeof(uint32_t);
            break;
        }
        case BR_REPLY: {
            struct binder_txn *txn = (void*) ptr;
        ...
```

```
            if (bio) {
                bio_init_from_txn(bio, txn);
                bio = 0;
            } else {
                    /* todo FREE BUFFER */
            }
        ...
        }
    ...
        }
    }

    return r;
}
```

增加服务、检查服务

各种服务存放在一个链表(svclist)中。

```
int svcmgr_handler(struct binder_state *bs,
                   struct binder_txn *txn,
                   struct binder_io *msg,
                   struct binder_io *reply)
{
    struct svcinfo *si;
    uint16_t *s;
    unsigned len;
    void *ptr;
    ...
    switch(txn->code) {
    case SVC_MGR_GET_SERVICE:
    case SVC_MGR_CHECK_SERVICE:
        s = bio_get_string16(msg, &len);
        ptr = do_find_service(bs, s, len);
        if (!ptr)
            break;
        bio_put_ref(reply, ptr);
        return 0;

    case SVC_MGR_ADD_SERVICE:
        s = bio_get_string16(msg, &len);
        ptr = bio_get_ref(msg);
        if (do_add_service(bs, s, len, ptr, txn->sender_euid))
            return -1;
        break;

    case SVC_MGR_LIST_SERVICES: {
        unsigned n = bio_get_uint32(msg);

        si = svclist;
        while ((n-- > 0) && si)
            si = si->next;
        if (si) {
            bio_put_string16(reply, si->name);
```

```
                return 0;
            }
            return -1;
        }
        default:
            LOGE("unknown code %d\n", txn->code);
            return -1;
        }

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

talkWithDriver

Binder.h (bionic\libc\kernel\common\linux):#define BINDER_WRITE_READ _IOWR('b', 1, struct binder_write_read)

Binder.c (frameworks\base\cmds\servicemanager):

binder_write(), binder_call(),binder_loop() 调用了 ioctl(bs->fd, BINDER_WRITE_READ, &bwr);

IPCThreadState.cpp (frameworks\base\libs\utils):

talkWithDriver() 函数调用了：if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)

最终调用的是 Binder.c (kernel\drivers\misc)中的驱动函数：binder_ioctl()

```
Binder.c (kernel\drivers\misc):
        case BINDER_WRITE_READ: {
```

Service_manager.c (frameworks\base\cmds\servicemanager):   bs = binder_open(128*1024);

binder_open(128*1024) \\

binder_loop(bs, svcmgr_handler)

sys_ioctl
do_vfs_ioctl
do_vfs_ioctl
file_ioctl
vfs_ioctl
binder_ioctl()
binder_thread_write()
binder_thread_read()

binder_ioctl()分析

struct binder_proc *proc = filp->private_data;

此处的 proc 是在 binder_open() 从申请赋值的。

proc = kzalloc(sizeof(*proc), GFP_KERNEL);

```
filp->private_data = proc;
```

```
        struct binder_write_read bwr;
        int res;
        bwr.write_size = len;
        bwr.write_consumed = 0;
        bwr.write_buffer = (unsigned) data;
        bwr.read_size = 0;
        bwr.read_consumed = 0;
        bwr.read_buffer = 0;
        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
```

binder_poll()解析

binder_open() 开发一段共享的内存

```
bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
```