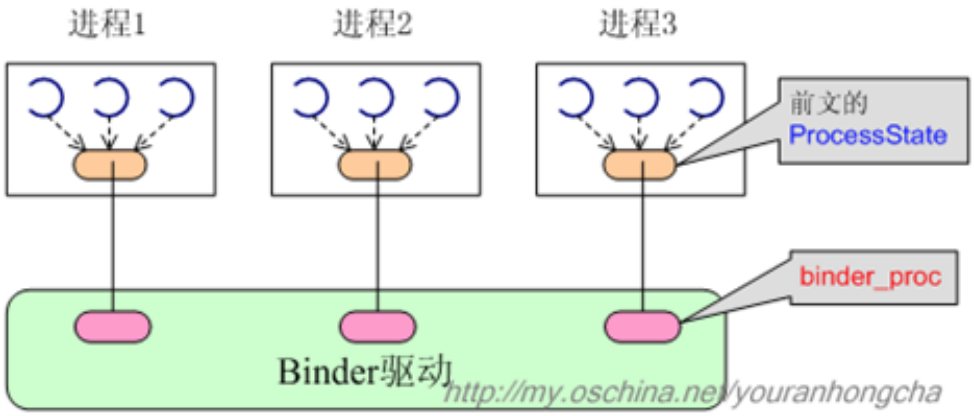
红茶一杯话Binder
（传输机制篇_上）

侯 亮

1 Binder是如何做到精确打击的?

我们先问一个问题，binder机制到底是如何从代理对象找到其对应的binder实体呢？难道它有某种制导装置吗？要回答这个问题，我们只能静下心来研究binder驱动的代码。在本系列文档的初始篇中，我们曾经介绍过ProcessState，这个结构是属于应用层次的东西，仅靠它当然无法完成精确打击。其实，在binder驱动层，还有个与之相对的结构，叫做binder_proc。为了说明问题，我修改了初始篇中的示意图，得到下图：



1.1 创建binder_proc

当构造ProcessState并打开binder驱动之时，会调用到驱动层的binder_open()函数，而binder_proc就是在binder_open()函数中创建的。新创建的binder_proc会作为一个节点，插入一个总链表（binder_procs）中。具体代码可参考kernel/drivers/staging/android/Binder.c。

驱动层的binder_open()的代码如下：

```
static int binder_open(struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc;

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

    get_task_struct(current);
    proc->tsk = current;
    . . . . . .
    hlist_add_head(&proc->proc_node, &binder_procs);
    proc->pid = current->group_leader->pid;
    . . . . . .
    filp->private_data = proc;
    . . . . . .
}
```

注意，新创建的binder_proc会被记录在参数filp的private_data域中，以后每次执行binder_ioctl()，都会从filp->private_data域重新读取binder_proc的。

binder_procs总表的定义如下：

```
static HLIST_HEAD(binder_procs);
```

我们可以在List.h中看到HLIST_HEAD的定义：

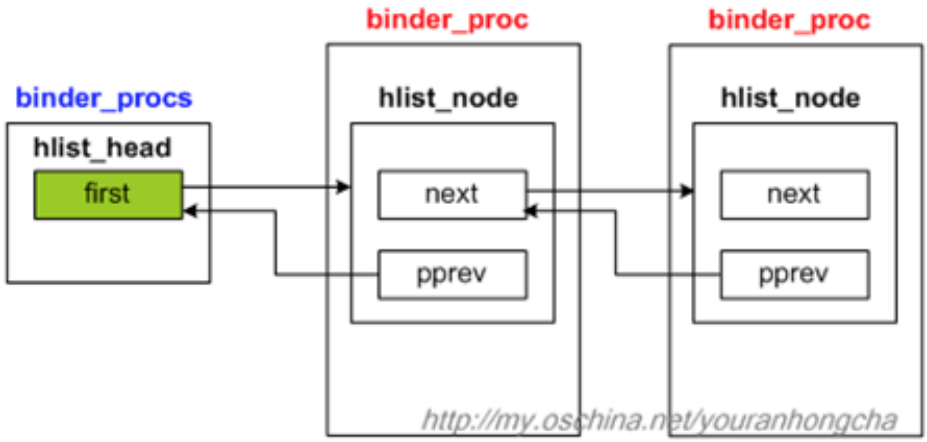
【kernel/include/linux/List.h】

```
#define HLIST_HEAD(name) struct hlist_head name = {  .first = NULL }
```

于是binder_procs的定义相当于：

```
struct hlist_head binder_procs  = { .first = NULL };
```

随着后续不断向binder_procs表中添加节点，这个表会不断加长，示意图如下：

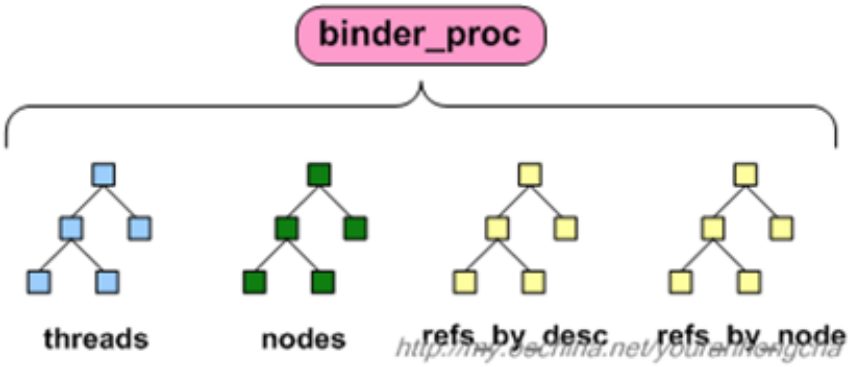


1.2 binder_proc中的4棵红黑树

binder_proc里含有很多重要内容，不过目前我们只需关心其中的几个域：

```
struct binder_proc
{
    struct hlist_node proc_node;
    struct rb_root threads;
    struct rb_root nodes;
    struct rb_root refs_by_desc;
    struct rb_root refs_by_node;
    int pid;
    . . . . . .
    . . . . . .
};
```

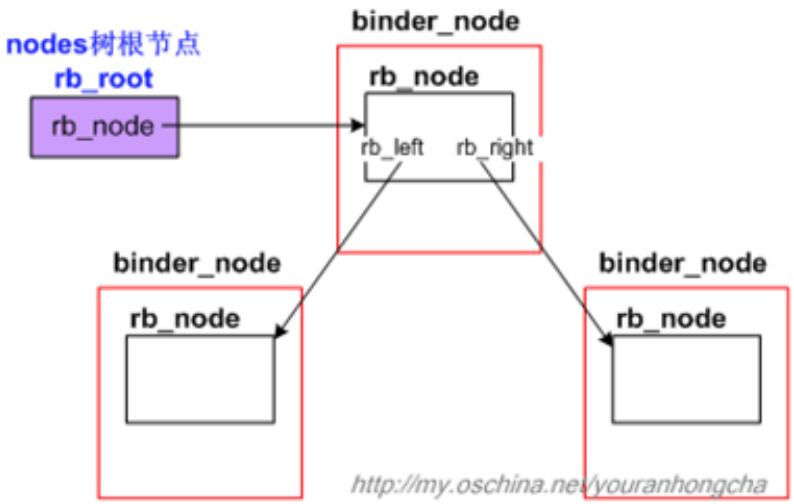
注意其中的那4个rb_root域，“rb”的意思是“red black”，可见binder_proc里搞出了4个红黑树。



其中，nodes树用于记录binder实体，refs_by_desc树和refs_by_node树则用于记录binder代理。之所以会有两个代理树，是为了便于快速查找，我们暂时只关心其中之一就可以了。threads树用于记录执行传输动作的线程信息。

在一个进程中，有多少“被其他进程进行跨进程调用的”binder实体，就会在该进程对应的nodes树中生成多少个红黑树节点。另一方面，一个进程要访问多少其他进程的binder实体，则必须在其refs_by_desc树中拥有对应的引用节点。

这4棵树的节点类型是不同的，threads树的节点类型为binder_thread，nodes树的节点类型为binder_node，refs_by_desc树和refs_by_node树的节点类型相同，为binder_ref。这些节点内部都会包含rb_node子结构，该结构专门负责连接节点的工作，和前文的hlist_node有点儿异曲同工，这也是linux上一个常用的小技巧。我们以nodes树为例，其示意图如下：



rb_node和rb_root的定义如下：

```
struct rb_node
{
    unsigned long  rb_parent_color;
#define RB_RED      0
#define RB_BLACK    1
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
    /* The alignment might seem pointless, but allegedly CRIS needs it */

struct rb_root
{
    struct rb_node *rb_node;
};
```

binder_node的定义如下：

```
struct binder_node
{
    int debug_id;
    struct binder_work work;
    union {
        struct rb_node rb_node;
        struct hlist_node dead_node;
    };
    struct binder_proc *proc;
    struct hlist_head refs;
    int internal_strong_refs;
    int local_weak_refs;
    int local_strong_refs;
    void __user *ptr;       // 注意这个域!
    void __user *cookie;    // 注意这个域!
    unsigned has_strong_ref:1;
    unsigned pending_strong_ref:1;
    unsigned has_weak_ref:1;
    unsigned pending_weak_ref:1;
    unsigned has_async_transaction:1;
    unsigned accept_fds:1;
    unsigned min_priority:8;
    struct list_head async_todo;
};
```

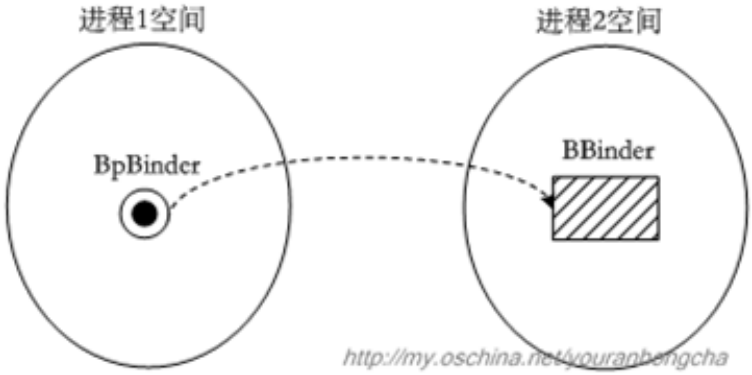
我们前文已经说过，nodes树是用于记录binder实体的，所以nodes树中的每个binder_node节点，必须能够记录下相应binder实体的信息。因此请大家注意binder_node的ptr域和cookie域。

另一方面，refs_by_desc树和refs_by_node树的每个binder_ref节点则和上层的一个BpBinder对应，而且更重要的是，它必须具有和"目标binder实体的binder_node"进行关联的信息。binder_ref的定义如下：

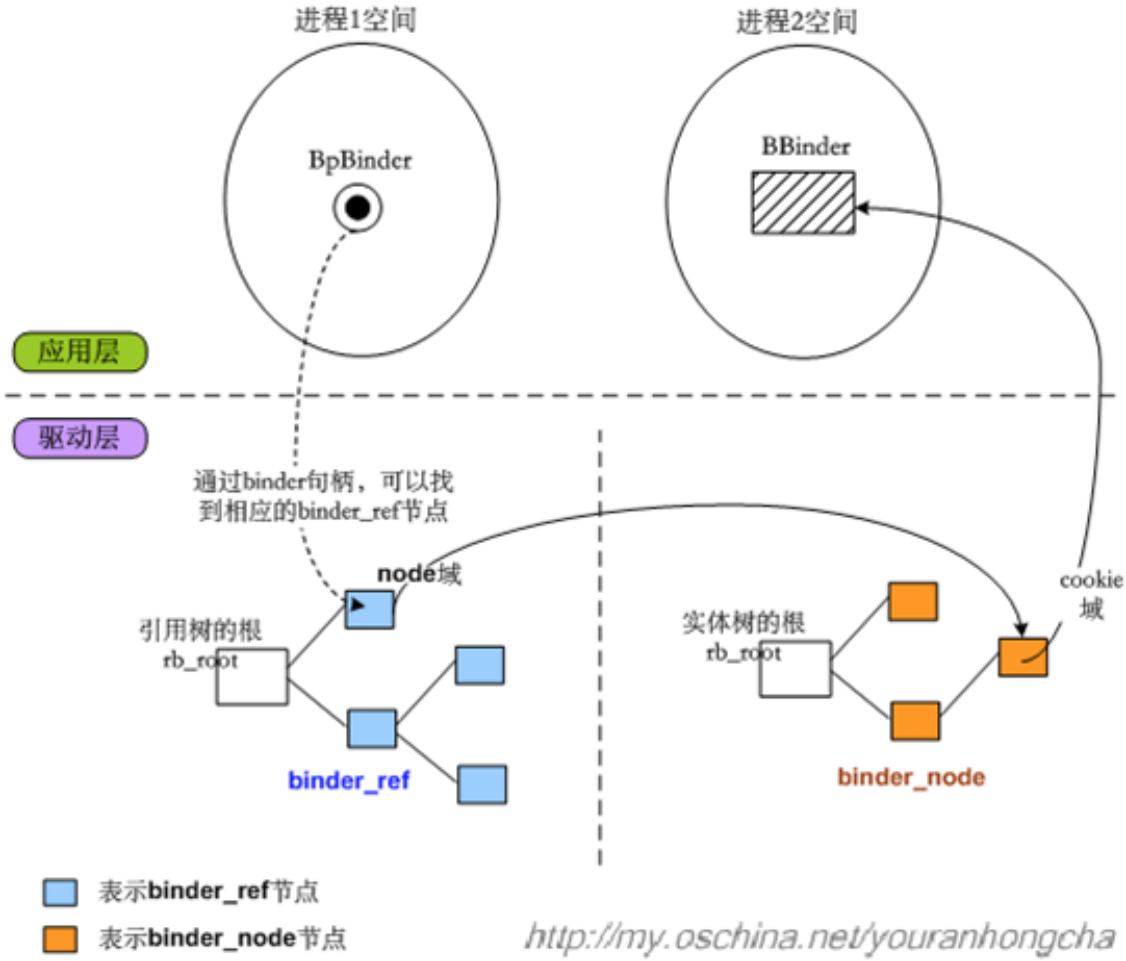
```
struct binder_ref
{
    int debug_id;
    struct rb_node rb_node_desc;
    struct rb_node rb_node_node;
    struct hlist_node node_entry;
    struct binder_proc *proc;
    struct binder_node *node;   // 注意这个node域
    uint32_t desc;
    int strong;
    int weak;
    struct binder_ref_death *death;
};
```

请注意那个node域，它负责和binder_node关联。另外，binder_ref中有两个类型为rb_node的域：rb_node_desc域和rb_node_node域，它们分别用于连接refs_by_desc树和refs_by_node。也就是说虽然binder_proc中有两棵引用树，但这两棵树用到的具体binder_ref节点其实是复用的。

大家应该还记得，在《初始篇》中我是这样表达BpBinder和BBinder关系的：



现在，我们有了binder_ref和binder_node知识，可以再画一张图，来解释BpBinder到底是如何和BBinder联系上的：



上图只表示了从进程1向进程2发起跨进程传输的意思，其实反过来也是可以的，即进程2也可以通过自己的"引用树"节点找到进程1的"实体树"节点，并进行跨进程传输。大家可以自己补充上图。

OK，现在我们可以更深入地说明binder句柄的作用了，比如进程1的BpBinder在发起跨进程调用时，向binder驱动传入了自己记录的句柄值，binder驱动就会在"进程1对应的binder_proc结构"的引用树中查找和句柄值相符的binder_ref节点，一旦找到binder_ref节点，就可以通过该节点的node域找到对应的binder_node节点，这个目标binder_node当然是从属于进程2的binder_proc啦，不过不要紧，因为binder_ref和binder_node都处于binder驱动的地址空间中，所以是可以用指针直接指向的。目标binder_node节点的cookie域，记录的其实是进程2中BBinder的地址，binder驱动只需把这个值反映给应用层，应用层就可以直接拿到BBinder了。这就是Binder完成精确打击的大体过程。

2 BpBinder和IPCThreadState

接下来我们来谈谈Binder传输机制。

在《初始篇》中，我们已经提到了BpBinder和ProcessState。当时只是说BpBinder是代理端的核心，主要负责跨进程传输，并且不关心所传输的内容。而ProcessState则是进程状态的记录器，它里面记录着打开binder驱动后得到的句柄值。因为我们并没有进一步展开来讨论BpBinder和ProcessState，所以也就没有进一步打通BpBinder和ProcessState之间的关系。现在，我们试着补充一些内容。

作为代理端的核心，BpBinder总要通过某种方式和binder驱动打交道，才可能完成跨进程传递语义的工作。既然binder驱动对应的句柄在ProcessState中记着，那么现在就要看BpBinder如何和ProcessState联系了。此时，我们需要提到IPCThreadState。

从名字上看，IPCThreadState是“和跨进程通信（IPC）相关的线程状态”。那么很显然，一个具有多个线程的进程里应该会有多个IPCThreadState对象了，只不过每个线程只需一个IPCThreadState对象而已。这有点儿“局部单例”的意思。所以，在实际的代码中，IPCThreadState对象是存放在线程的局部存储区（TLS）里的。

2.1 BpBinder的transact()动作

每当我们利用BpBinder的transact()函数发起一次跨进程事务时，其内部其实是调用IPCThreadState对象的transact()。BpBinder的transact()代码如下：

```
status_t BpBinder::transact(uint32_t code, const Parcel& data,
Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    if (mAlive)
    {
        status_t status = IPCThreadState::self()->transact(mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

当然，进程中的一个BpBinder有可能被多个线程使用，所以发起传输的IPCThreadState对象可能并不是同一个对象，但这没有关系，因为这些IPCThreadState对象最终使用的是同一个ProcessState对象。

2.1.1 调用IPCThreadState的transact()

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    . . . . . .
        // 把data数据整理进内部的mOut包中
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    . . . . . .

    if ((flags & TF_ONE_WAY) == 0)
```

```
    {
        . . . . . .
        if (reply)
        {
            err = waitForResponse(reply);
        }
        else
        {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        . . . . . .
    }
    else
    {
        err = waitForResponse(NULL, NULL);
    }

    return err;
}
```

IPCThreadState::transact()会先调用writeTransactionData()函数将data数据整理进内部的mOut包中，这个函数的代码如下：

```
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
                                              int32_t handle, uint32_t code,
                                              const Parcel& data, status_t* statusBuffer)
{
    binder_transaction_data tr;

    tr.target.handle = handle;
    tr.code = code;
    tr.flags = binderFlags;
    tr.cookie = 0;
    tr.sender_pid = 0;
    tr.sender_euid = 0;

    . . . . . .
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    . . . . . .

    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));

    return NO_ERROR;
}
```

接着IPCThreadState::transact()会考虑本次发起的事务是否需要回复。“不需要等待回复的”事务，在其flag标志中会含有TF_ONE_WAY，表示一去不回头。而“需要等待回复的”，则需要在传递时提供记录回复信息的Parcel对象，一般发起transact()的用户会提供这个Parcel对象，如果不提供，transact()函数内部会临时构造一个假的Parcel对象。

上面代码中，实际完成跨进程事务的是waitForResponse()函数，这个函数的命名不太好，但我们也不必太在意，反正Android中写得不好的代码多了去了，又不只多这一处。waitForResponse()的代码截选如下：

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    int32_t cmd;
    int32_t err;

    while (1)
    {
        // talkWithDriver()内部会完成跨进程事务
        if ((err = talkWithDriver()) < NO_ERROR)
            break;

        // 事务的回复信息被记录在mIn中，所以需要进一步分析这个回复
        . . . . . .
        cmd = mIn.readInt32();
        . . . . . .
        switch (cmd)
        {
        case BR_TRANSACTION_COMPLETE:
            if (!reply && !acquireResult) goto finish;
            break;

        case BR_DEAD_REPLY:
            err = DEAD_OBJECT;
            goto finish;

        case BR_FAILED_REPLY:
            err = FAILED_TRANSACTION;
            goto finish;
        . . . . . .
        . . . . . .
        default:
            // 注意这个executeCommand()噢，它会处理BR_TRANSACTION的。
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
    }

finish:
    . . . . . .
    return err;
}
```

2.1.2 talkWithDriver()

waitForResponse()中是通过调用talkWithDriver()来和binder驱动打交道的，说到底会调用ioctl()函数。因为ioctl()函数在传递BINDER_WRITE_READ语义时，既会使用“输入buffer”，也会使用“输出buffer”，所以IPCThreadState专门搞了两个Parcel类型的成员变量：mIn和mOut。总之就是，mOut中的内容发出去，发送后的回复写进mIn。

talkWithDriver()的代码截选如下：

```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    . . . . . .
    binder_write_read bwr;

    . . . . . .
    bwr.write_size = outAvail;
    bwr.write_buffer = (long unsigned int)mOut.data();
    . . . . . .
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (long unsigned int)mIn.data();
    . . . . . .
    . . . . . .
    do
    {
        . . . . . .
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        . . . . . .
    } while (err == -EINTR);

    . . . . . .
    . . . . . .
    return err;
}
```

看到了吗？mIn和mOut的data会先整理进一个binder_write_read结构，然后再传给ioctl()函数。而最关键的一句，当然就是那句ioctl()了。此时使用的文件描述符就是前文我们说的ProcessState中记录的mDriverFD，说明是向binder驱动传递语义。BINDER_WRITE_READ表示我们希望读写一些数据。

至此，应用程序通过BpBinder向远端发起传输的过程就交代完了，数据传到了binder驱动，一切就看binder驱动怎么做了。至于驱动层又做了哪些动作，我们留在下一篇文章再介绍。