原 荐 红茶一杯话Binder（传输机制篇_下）

发表于2年前(2013-10-08 22:44) 阅读（4496） | 评论（9） 56 人收藏此文章, 我要收藏

红茶一杯话Binder
（传输机制篇_下）

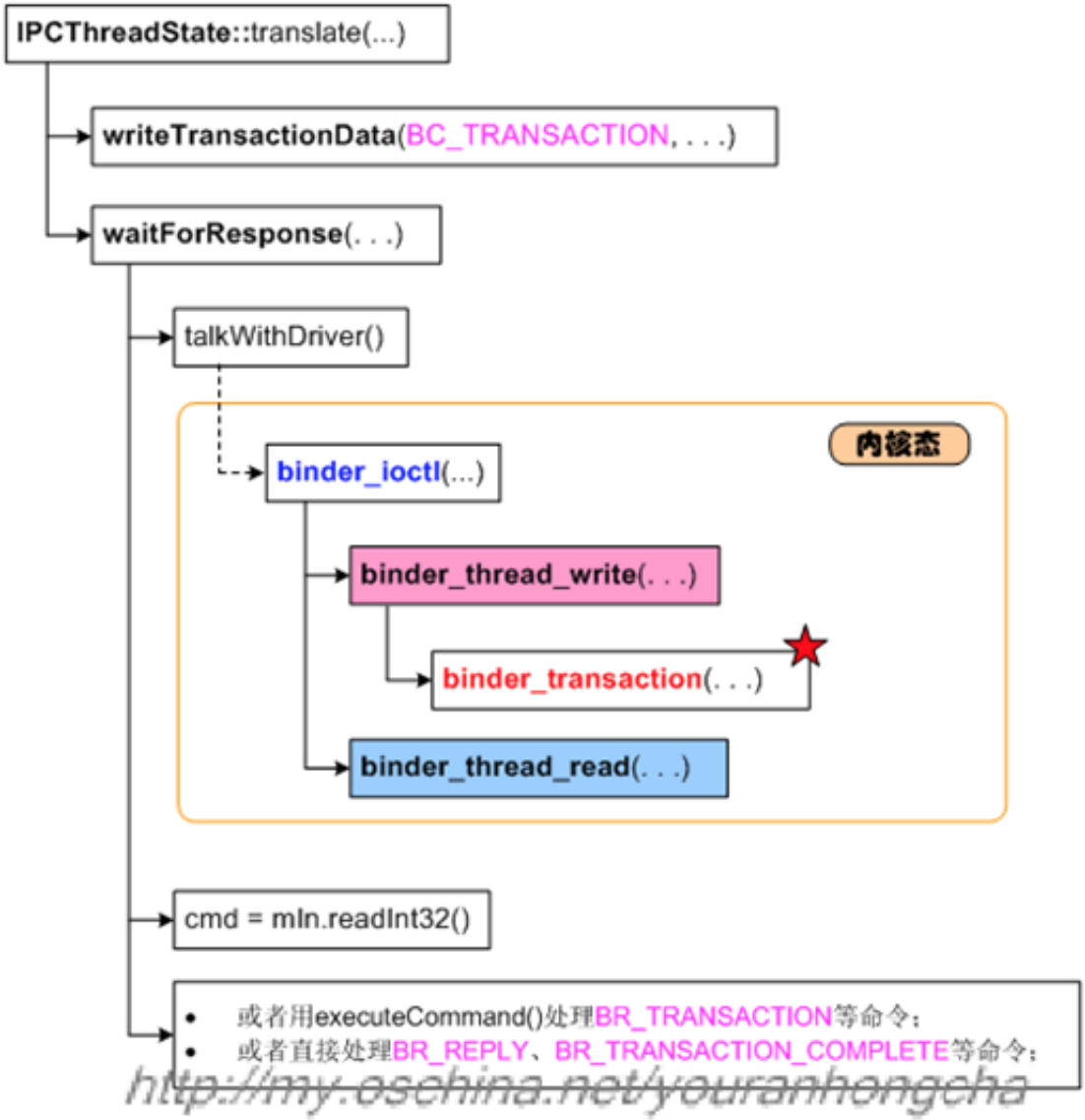
侯 亮

1 事务的传递和处理

从IPCThreadState的角度看，它的transact()函数是通过向binder驱动发出BC_TRANSACTION语义，来表达其传输意图的，而后如有必要，它会等待从binder发回的回馈，这些回馈语义常常以“BR_”开头。另一方面，当IPCThreadState作为处理命令的一方需要向发起方反馈信息的话，它会调用sendReply()函数，向binder驱动发出BC_REPLY语义。当BC_语义经由binder驱动递送到目标端时，会被binder驱动自动修改为相应的BR_语义，这个我们在后文再细说。

当语义传递到binder驱动后，会走到binder_ioctl()函数，该函数又会调用到binder_thread_write()和binder_thread_read()：

在上一篇文章中，我们大体阐述了一下binder_thread_write()和binder_thread_read()的唤醒与被唤醒关系，而且还顺带在“传输机制的大体运作”小节中提到了todo队列的概念。本文将在此基础上再补充一些知识。需要强调的是，我们必须重视binder_thread_write()和binder_thread_read()，因为事务的传递和处理就位于这两个函数中，它们的调用示意图如下：



binder_thread_write()的代码截选如下。因为本文主要关心传输方面的问题，所以只摘取了case BC_TRANSACTION、case BC_REPLY部分的代码：

```
int binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
                        void __user *buffer, int size, signed long *consumed)
{
    . . . . . .
    while (ptr < end && thread->return_error == BR_OK)
    {
        . . . . . .
        switch (cmd)
        {
        . . . . . .
        . . . . . .
        case BC_TRANSACTION:
        case BC_REPLY: {
            struct binder_transaction_data tr;

            if (copy_from_user(&tr, ptr, sizeof(tr)))
                return -EFAULT;
            ptr += sizeof(tr);
            binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
            break;
```

```
        }
        . . . . . .
        . . . . . .
        }
        *consumed = ptr - buffer;
    }
    return 0;
}
```

这部分代码比较简单，主要是从用户态拷贝来binder_transaction_data数据，并传给binder_transaction()函数进行实际的传输。而binder_transaction()可是需要我们费一点儿力气去分析的，大家深吸一口气，准备开始。

1.1 BC_TRANSACTION事务（携带TF_ONE_WAY标记）的处理

首先我们要认识到，同样是BC_TRANSACTION事务，带不带TF_ONE_WAY标记还是有所不同的。我们先看相对简单的携带TF_ONE_WAY标记的BC_TRANSACTION事务，这种事务是不需要回复的。

1.1.1 binder_transaction()

此时，binder_transaction()所做的工作大概有：

1. 找目标binder_node；
2. 找目标binder_proc；
3. 分析并插入红黑树节点；（我们在上一篇文章中已在说过这部分的机理了，只是当时没有贴出相应的代码）
4. 创建binder_transaction节点，并将其插入目标进程的todo列表；
5. 尝试唤醒目标进程。

binder_transaction()代码截选如下：

```
static void binder_transaction(struct binder_proc *proc,
                   struct binder_thread *thread,
                   struct binder_transaction_data *tr, int reply)
{
    struct binder_transaction *t;
    . . . . . .
    struct binder_proc *target_proc;
    struct binder_thread *target_thread = NULL;
    struct binder_node *target_node = NULL;
    struct list_head *target_list;
    wait_queue_head_t *target_wait;
    . . . . . .
    . . . . . .
    {
        // 先从tr->target.handle句柄值，找到对应的binder_ref节点，及binder_node节点
         if (tr->target.handle)
        {
            struct binder_ref *ref;
            ref = binder_get_ref(proc, tr->target.handle);
            . . . . . .
```

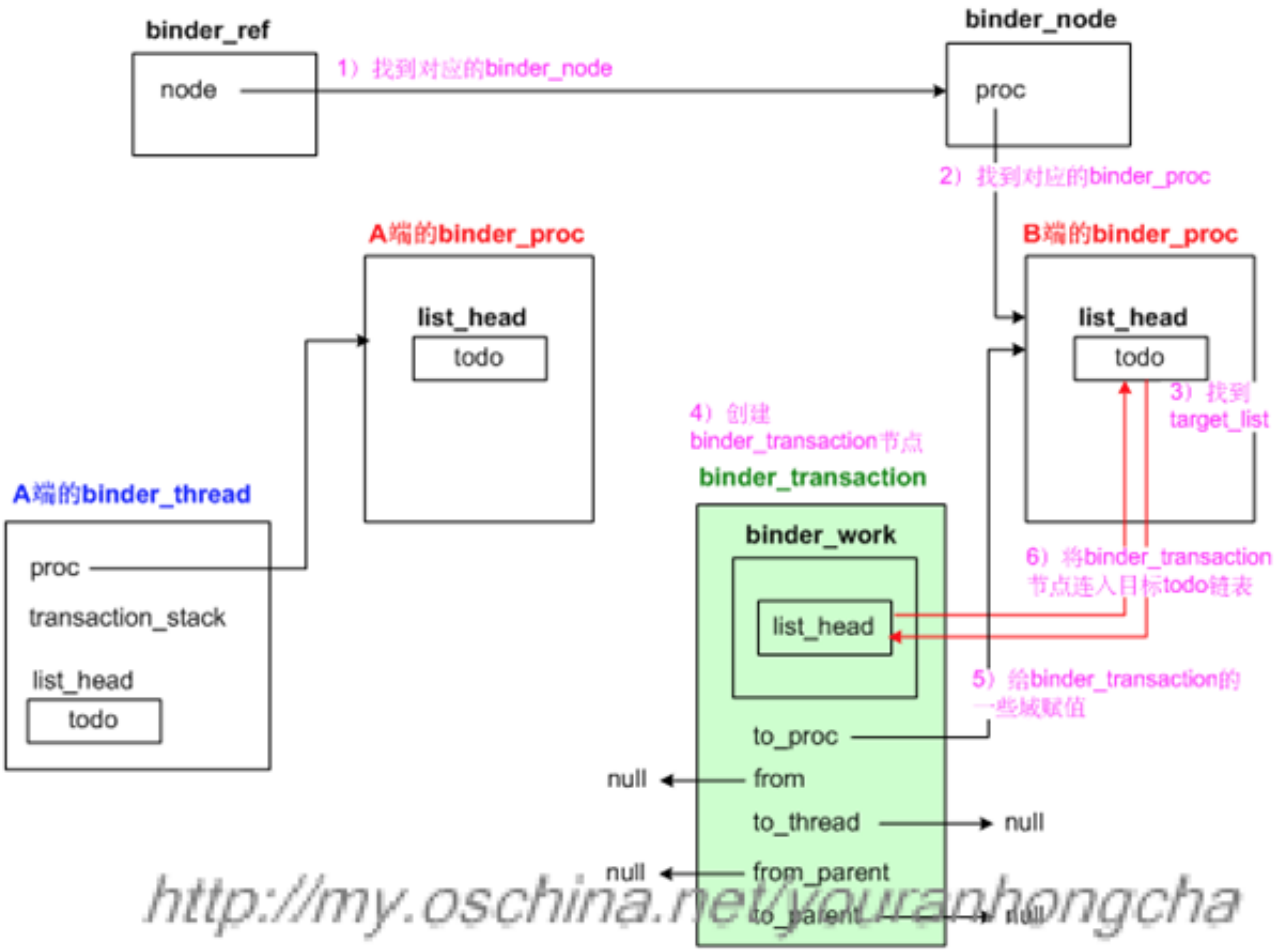
```
            target_node = ref->node;
        }
        else
        {
            // 如果句柄值为0，则获取特殊的binder_context_mgr_node节点，
            // 即Service Manager Service对应的节点
            target_node = binder_context_mgr_node;
            . . . . . .
        }
        // 得到目标进程的binder_proc
        target_proc = target_node->proc;
        . . . . . .
    }

    // 对于带TF_ONE_WAY标记的BC_TRANSACTION来说，此时target_thread为NULL，
    // 所以准备向binder_proc的todo中加节点
    . . . . . .
        target_list = &target_proc->todo;
        target_wait = &target_proc->wait;
    . . . . . .

    // 创建新的binder_transaction节点。
    t = kzalloc(sizeof(*t), GFP_KERNEL);
    . . . . . .
    t->from = NULL;

    t->sender_euid  = proc->tsk->cred->euid;
    t->to_proc      = target_proc;
    t->to_thread    = target_thread;

    // 将binder_transaction_data的code、flags域记入binder_transaction节点。
    t->code     = tr->code;
    t->flags    = tr->flags;
    t->priority = task_nice(current);

    t->buffer = binder_alloc_buf(target_proc, tr->data_size, tr->offsets_size,
                                 !reply && (t->flags & TF_ONE_WAY));
    . . . . . .
    t->buffer->transaction = t;
    t->buffer->target_node = target_node;
    . . . . . .

    // 下面的代码分析所传数据中的所有binder对象，如果是binder实体的话，要在红黑树中添加相应的节点。
    // 首先，从用户态获取所传输的数据，以及数据里的binder对象的偏移信息
    offp = (size_t *)(t->buffer->data + ALIGN(tr->data_size, sizeof(void *)));
    if (copy_from_user(t->buffer->data, tr->data.ptr.buffer, tr->data_size))
    . . . . . .
    if (copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_size))
        . . . . . .
    . . . . . .
    // 遍历每个flat_binder_object信息，创建必要的红黑树节点 ....
    for (; offp < off_end; offp++)
    {
        struct flat_binder_object *fp;
```

```
    . . . . . .
    fp = (struct flat_binder_object *)(t->buffer->data + *offp);

    switch (fp->type)
    {
    case BINDER_TYPE_BINDER:
    case BINDER_TYPE_WEAK_BINDER:
    {
        // 如果是binder实体
        struct binder_ref *ref;
        struct binder_node *node = binder_get_node(proc, fp->binder);
        if (node == NULL)
        {
            // 又是"没有则创建"的做法，创建新的binder_node节点
           node = binder_new_node(proc, fp->binder, fp->cookie);
            . . . . . .
        }
        . . . . . .
        // 必要时，会在目标进程的binder_proc中创建对应的binder_ref红黑树节点
        ref = binder_get_ref_for_node(target_proc, node);
        . . . . . .
        // 修改所传数据中的flat_binder_object信息，因为远端的binder实体到了目标
        // 端，就变为binder代理了，所以要记录下binder句柄了。
        fp->handle = ref->desc;
        . . . . . .
    } break;

    case BINDER_TYPE_HANDLE:
    case BINDER_TYPE_WEAK_HANDLE: {
        struct binder_ref *ref = binder_get_ref(proc, fp->handle);
        // 有时候需要对flat_binder_object做必要的修改，比如将BINDER_TYPE_HANDLE
        // 改为BINDER_TYPE_BINDER
        . . . . . .
    } break;

    case BINDER_TYPE_FD: {
        . . . . . .
    } break;
    . . . . . .
}

. . . . . .
{
    . . . . . .
    if (target_node->has_async_transaction)
    {
        target_list = &target_node->async_todo;
        target_wait = NULL;
    }
    else
        target_node->has_async_transaction = 1;
}

t->work.type = BINDER_WORK_TRANSACTION;
```

```
    // 终于把binder_transaction节点插入target_list（即目标todo队列）了。
    list_add_tail(&t->work.entry, target_list);
    . . . . . .
    list_add_tail(&tcomplete->entry, &thread->todo);

    // 传输动作完毕，现在可以唤醒系统中其他相关线程了，wake up!
    if (target_wait)
        wake_up_interruptible(target_wait);
    return;
    . . . . . .
    . . . . . .
}
```

虽然已经是截选，代码却仍然显得冗长。这也没办法，Android frameworks里的很多代码都是这个样子，又臭又长，大家凑合着看吧。我常常觉得google的工程师多少应该因这样的代码而感到脸红，不过，哎，这有点儿说远了。

我们画一张示意图，如下：



上图体现了从binder_ref找到“目标binder_node”以及“目标binder_proc”的意思，其中“A端”表示发起方，“B端”表示目标方。可以看到，携带TF_ONE_WAY标记的事务，其实是比较简单的，驱动甚至不必费心去找目标线程，只需要创建一个binder_transaction节点，并插入目标binder_proc的todo链表即可。

另外，在将binder_transaction节点插入目标todo链表之前，binder_transaction()函数用一个for循环分析了需要传输的数据，并为其中包含的binder对象生成了相应的红黑树节点。

再后来，binder_transaction节点成功插入目标todo链表，此时说明目标进程有事情可做了，于是binder_transaction()函数会调用wake_up_interruptible()唤醒目标进程。

1.1.2 binder_thread_read()

当目标进程被唤醒时，会接着执行自己的binder_thread_read()，尝试解析并执行那些刚收来的工作。无论收来的工作来自于"binder_proc的todo链表"，还是来自于某"binder_thread的todo链表"，现在要开始从todo链表中摘节点了，而且在完成工作之后，会彻底删除binder_transaction节点。

binder_thread_read()的代码截选如下：

```
static int binder_thread_read(struct binder_proc *proc,
                  struct binder_thread *thread,
                  void  __user *buffer, int size,
                  signed long *consumed, int non_block)
{
    . . . . . .
retry:
    // 优先考虑thread节点的todo链表中有没有工作需要完成
    wait_for_proc_work = thread->transaction_stack == NULL
                        && list_empty(&thread->todo);
    . . . . . .
    . . . . . .
    if (wait_for_proc_work)
    {
        . . . . . .
        ret = wait_event_interruptible_exclusive(proc->wait,
        binder_has_proc_work(proc, thread));
    }
    else
    {
        . . . . . .
        ret = wait_event_interruptible(thread->wait, binder_has_thread_work(thread));
    }
    . . . . . .
    thread->looper &= ~BINDER_LOOPER_STATE_WAITING;

    // 如果是非阻塞的情况，ret值非0表示出了问题，所以return。
    // 如果是阻塞（non_block）情况，ret值非0表示等到的结果出了问题，所以也return。
    if (ret)
        return ret;

    while (1)
    {
        . . . . . .
        // 读取binder_thread或binder_proc中todo列表的第一个节点
        if (!list_empty(&thread->todo))
            w = list_first_entry(&thread->todo, struct binder_work, entry);
        else if (!list_empty(&proc->todo) && wait_for_proc_work)
            w = list_first_entry(&proc->todo, struct binder_work, entry);
        . . . . . .
```

```
        switch (w->type)
        {
        case BINDER_WORK_TRANSACTION: {
            t = container_of(w, struct binder_transaction, work);
        } break;

        case BINDER_WORK_TRANSACTION_COMPLETE: {
            cmd = BR_TRANSACTION_COMPLETE;
            . . . . . .
              // 将binder_transaction节点从todo队列摘下来
            list_del(&w->entry);
            kfree(w);
            binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
        } break;
          . . . . . .
          . . . . . .
        }

        if (!t)
            continue;

        . . . . . .
        if (t->buffer->target_node)
{
            struct binder_node *target_node = t->buffer->target_node;
            tr.target.ptr = target_node->ptr;
            // 用目标binder_node中记录的cookie值给binder_transaction_data的cookie域赋值，
              // 这个值就是目标binder实体的地址
              tr.cookie =  target_node->cookie;
            t->saved_priority = task_nice(current);
            . . . . . .
            cmd = BR_TRANSACTION;
        }
        . . . . . .
        tr.code = t->code;
        tr.flags = t->flags;
        tr.sender_euid = t->sender_euid;
        . . . . . .
        tr.data_size = t->buffer->data_size;
        tr.offsets_size = t->buffer->offsets_size;
         // binder_transaction_data中的data只是记录了binder缓冲区中的地址信息，并再做copy动作
        tr.data.ptr.buffer = (void *)t->buffer->data +
                    proc->user_buffer_offset;
        tr.data.ptr.offsets = tr.data.ptr.buffer +
                    ALIGN(t->buffer->data_size,
                        sizeof(void *));

         // 将cmd命令写入用户态，此时应该是BR_TRANSACTION
        if (put_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
         // 当然，binder_transaction_data本身也是要copy到用户态的
        if (copy_to_user(ptr, &tr, sizeof(tr)))
            return -EFAULT;
```

```
        . . . . . .
        . . . . . .
        // 将binder_transaction节点从todo队列摘下来
        list_del(&t->work.entry);
        t->buffer->allow_user_free = 1;
        if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
            t->to_parent = thread->transaction_stack;
            t->to_thread = thread;
            thread->transaction_stack = t;
        } else {
            t->buffer->transaction = NULL;
            // TF_ONE_WAY情况，此时会删除binder_transaction节点
            kfree(t);
            binder_stats_deleted(BINDER_STAT_TRANSACTION);
        }
        break;
    }
    . . . . . .
    . . . . . .
    return 0;
}
```

简单说来就是，如果没有工作需要做，binder_thread_read()函数就进入睡眠或返回，否则binder_thread_read()函数会从todo队列摘下了一个节点，并把节点里的数据整理成一个binder_transaction_data结构，然后通过copy_to_user()把该结构传到用户态。因为这次传输带有TF_ONE_WAY标记，所以copy完后，只是简单地调用kfree(t)把这个binder_transaction节点干掉了。

binder_thread_read()尝试调用wait_event_interruptible()或wait_event_interruptible_exclusive()来等待待处理的工作。wait_event_interruptible()是个宏定义，和wait_event()类似，不同之处在于前者不但会判断“苏醒条件”，还会判断当前进程是否带有挂起的系统信号，当“苏醒条件”满足时（比如binder_has_thread_work(thread)返回非0值），或者有挂起的系统信号时，表示进程有工作要做了，此时wait_event_interruptible()将跳出内部的for循环。如果的确不满足跳出条件的话，wait_event_interruptible()会进入挂起状态。

请注意给binder_transaction_data的cookie赋值的那句：

```
tr.cookie =  target_node->cookie;
```

binder_node节点里储存的cookie值终于发挥作用了，这个值反馈到用户态就是目标binder实体的BBinder指针了。

另外，在调用copy_to_user()之前，binder_thread_read()先通过put_user()向上层拷贝了一个命令码，在当前的情况下，这个命令码是BR_TRANSACTION。想当初，内核态刚刚从用户态拷贝来的命令码是BC_TRANSACTION，现在要发给目标端了，就变成了BR_TRANSACTION。

1.2 BC_TRANSACTION事务（不带TF_ONE_WAY标记）

1.2.1 再说binder_transaction()

然而，对于不带TF_ONE_WAY标记的BC_TRANSACTION事务来说，情况就没那么简单了。因为bind

er驱动不仅要找到目标进程，而且还必须努力找到一个明确的目标线程。正如我们前文所说，binder驱动希望可以充分复用目标进程中的binder工作线程。

那么，哪些线程（节点）是可以被复用的呢？我们再整理一下binder_transaction()代码，本次主要截选不带TF_ONE_WAY标记的代码部分：

```
static void binder_transaction(struct binder_proc *proc,
                   struct binder_thread *thread,
                   struct binder_transaction_data *tr, int reply)
{
    struct binder_transaction *t;
    . . . . . .
    . . . . . .
        if (tr->target.handle)
        {
            . . . . . .
            target_node = ref->node;
        }
        else
        {
            target_node = binder_context_mgr_node;
            . . . . . .
        }
        . . . . . .
        // 先确定target_proc
        target_proc = target_node->proc;
        . . . . . .
        if (!(tr->flags & TF_ONE_WAY) && thread->transaction_stack)
        {
            struct binder_transaction *tmp;
            tmp = thread->transaction_stack;
            . . . . . .
            // 找到from_parent这条链表中，最后一个可以和target_proc匹配
            // 的binder_transaction节点，
            // 这个节点的from就是我们要找的“目标线程”
            while (tmp)
            {
                if (tmp->from && tmp->from->proc == target_proc)
                    target_thread = tmp->from;
                tmp = tmp->from_parent;
            }
        }
    . . . . . .
    // 要确定target_list和target_wait了，如果能找到“目标线程”，它们就来自目标线程，否则
    // 就只能来自目标进程了。
    if (target_thread)
    {
        e->to_thread = target_thread->pid;
        target_list = &target_thread->todo;
        target_wait = &target_thread->wait;
    }
    else {
        target_list = &target_proc->todo;
        target_wait = &target_proc->wait;
```

```
    }
    . . . . . .
    // 创建新的binder_transaction节点。
    t = kzalloc(sizeof(*t), GFP_KERNEL);
    . . . . . .
    . . . . . .

    t->from = thread;    // 新节点的from域记录事务的发起线程

    t->sender_euid = proc->tsk->cred->euid;
    t->to_proc = target_proc;
    t->to_thread = target_thread;    // 新节点的to_thread域记录事务的目标线程

    t->code = tr->code;
    t->flags = tr->flags;
    t->priority = task_nice(current);
    // 从binder buffer中申请一个区域，用于存储待传输的数据
    t->buffer = binder_alloc_buf(target_proc, tr->data_size,
                                 tr->offsets_size,
                                 !reply && (t->flags & TF_ONE_WAY));
    . . . . . .
    t->buffer->transaction = t;
    t->buffer->target_node = target_node;
    . . . . . .
    // 从用户态拷贝来待传输的数据
    if (copy_from_user(t->buffer->data, tr->data.ptr.buffer, tr->data_size)) {
        . . . . . .
    }
    if (copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_size)) {
        . . . . . .
    }
    // 遍历每个flat_binder_object信息，创建必要的红黑树节点 ....
    for (; offp < off_end; offp++)
    {
        struct flat_binder_object *fp;
        . . . . . .
        . . . . . .
    }

    . . . . . .
        t->need_reply = 1;
        // 新binder_transaction节点成为发起端transaction_stack栈的新栈顶
        t->from_parent = thread->transaction_stack;
        thread->transaction_stack = t;
    . . . . . .
    t->work.type = BINDER_WORK_TRANSACTION;

    // 终于把binder_transaction节点插入target_list（即目标todo队列）了。
    list_add_tail(&t->work.entry, target_list);
    tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
    list_add_tail(&tcomplete->entry, &thread->todo);
    if (target_wait)
        wake_up_interruptible(target_wait);
    return;
```

```
    . . . . . .
    . . . . . .
}
```

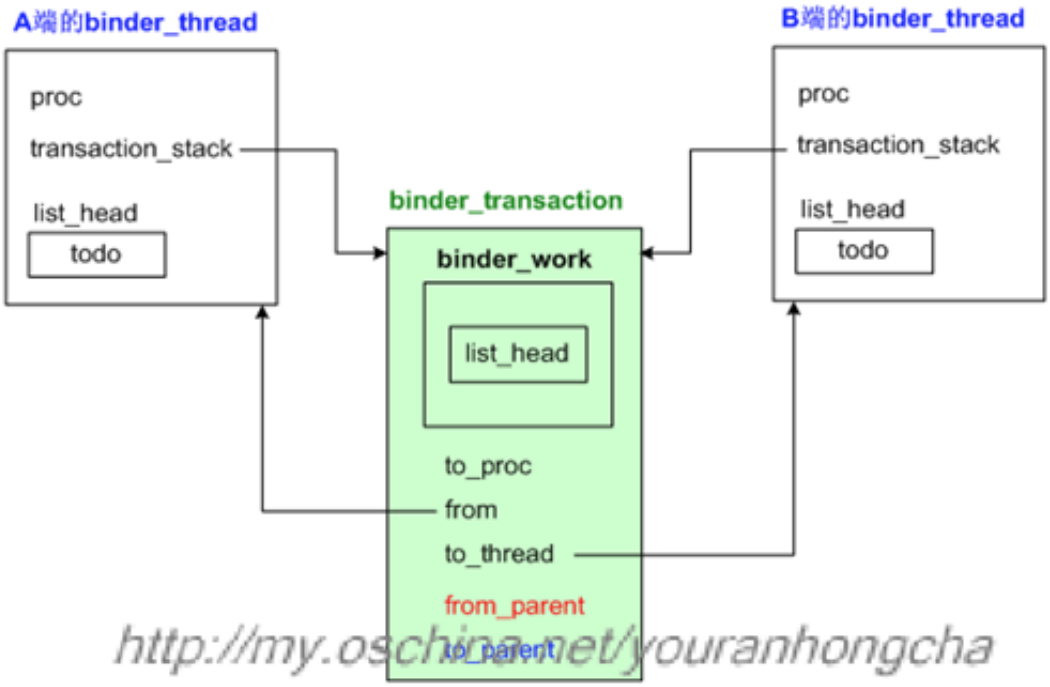
其中，获取目标binder_proc的部分和前一小节没什么不同，但是因为本次传输不再携带TF_ONE_WAY标记了，所以函数中会尽力去查一个合适的“目标binder_thread”，此时会用到binder_thread里的“事务栈”（transaction_stack）概念。

那么，怎么找“目标binder_thread”呢？首先，我们很清楚“发起端”的binder_thread节点是哪个，而且也可以找到“目标端”的binder_proc，这就具有了搜索的基础。在binder_thread节点的transaction_stack域里，记录了和它相关的若干binder_transaction，这些binder_transaction事务在逻辑上具有类似堆栈的属性，也就是说“最后入栈的事务”会最先处理。

从逻辑上说，线程节点的transaction_stack域体现了两个方面的意义：

1. 这个线程需要别的线程帮它做某项工作；
2. 别的线程需要这个线程做某项工作；

因此，一个工作节点（即binder_transaction节点）往往会插入两个transaction_stack堆栈，示意图如下：



当binder_transaction节点插入“发起端”的transaction_stack栈时，它是用from_parent域来连接堆栈中其他节点的。而当该节点插入“目标端”的transaction_stack栈时，却是用to_parent域来连接其他节点的。关于插入目标端堆栈的动作，位于binder_thread_read()中，我们在后文会看到。

这么看来，from_parent域其实将一系列逻辑上有先后关系的若干binder_transaction节点串接起来了，而且这些binder_transaction节点可能是由不同进程、线程发起的。那么我们只需遍历一下这个堆栈里的事务，看哪个事务的“from线程所属的进程”和“目标端的binder_proc”一致，就说明这个from线程正是我们要找的目标线程。为什么这么说呢？这是因为我们的新事务将成为binder_transaction的新栈顶，而这个堆栈里其他事务一定是在新栈顶事务处理完后才会处理的，因此堆栈里某个事务的发起端线程可以理解为正处于等待状态，如果这个发起端线程所从属的进程恰恰又是我们新事务的目标进程的话，那就算合拍了，这样就找到“目标binder_thread”了。我把相关的代码再抄一遍：

```
struct binder_transaction *tmp;
tmp = thread->transaction_stack;
```

```
while (tmp) {
    if (tmp->from && tmp->from->proc == target_proc)
        target_thread = tmp->from;
    tmp = tmp->from_parent;
}
```

代码用while循环来遍历thread->transaction_stack，发现tmp->from->proc == target_proc，就算找到了。

如果能够找到"目标binder_thread"的话，binder_transaction事务就会插到它的todo队列去。不过有时候找不到"目标binder_thread"，那么就只好退而求其次，插入binder_proc的todo队列了。再接下来的动作没有什么新花样，大体上会尝试唤醒目标进程。

1.2.2 再说binder_thread_read()

目标进程在唤醒后，会接着当初阻塞的地方继续执行，这个已在前一小节阐述过，我们不再赘述。值得一提的是binder_thread_read()中的以下句子：

```
        // 将binder_transaction节点从todo队列摘下来
        list_del(&t->work.entry);
        t->buffer->allow_user_free = 1;
        if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
            t->to_parent = thread->transaction_stack;
            t->to_thread = thread;
            thread->transaction_stack = t;
        } else {
            t->buffer->transaction = NULL;
              // TF_ONE_WAY情况，此时会删除binder_transaction节点
            kfree(t);
            binder_stats_deleted(BINDER_STAT_TRANSACTION);
        }
```

因为没有携带TF_ONE_WAY标记，所以此处会有一个入栈操作，binder_transaction节点插入了目标线程的transaction_stack堆栈，而且是以to_thread域来连接堆栈中的其他节点的。

总体说来，binder_thread_read()的动作大体也就是：

1） 利用wait_event_xxxx()让自己挂起，等待下一次被唤醒；
2） 唤醒后找到合适的待处理的工作节点，即binder_transaction节点；
3） 把binder_transaction中的信息整理到一个binder_transaction_data中；
4） 整理一个cmd整数值，具体数值或者为BR_TRANSACTION，或者为BR_REPLY；
5） 将cmd数值和binder_transaction_data拷贝到用户态；
6） 如有必要，将得到的binder_transaction节点插入目标端线程的transaction_stack堆栈中。

1.2.3 目标端如何处理传来的事务

binder_thread_read()本身只负责读取数据，它并不解析得到的语义。具体解析语义的动作并不在内

核态，而是在用户态。

我们再回到用户态的IPCThreadState::waitForResponse()函数。

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    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();
        . . . . . .

            err = executeCommand(cmd);
        . . . . . .
    }
    . . . . . .
}
```

当发起端调用binder_thread_write()唤醒目标端的进程时，目标进程会从其上次调用binder_thread_read()的地方苏醒过来。辗转跳出上面的talkWithDriver()函数，并走到executeCommand()一句。

因为binder_thread_read()中已经把BR_命令整理好了，所以executeCommand()当然会走到case BR_TRANSACTION分支：

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    RefBase::weakref_type* refs;
    . . . . . .
    . . . . . .

    case BR_TRANSACTION:
        {
            binder_transaction_data tr;
            result = mIn.read(&tr, sizeof(tr));
            . . . . . .
            mCallingPid = tr.sender_pid;
            mCallingUid = tr.sender_euid;
            mOrigCallingUid = tr.sender_euid;
            . . . . . .
            Parcel reply;
            . . . . . .
            if (tr.target.ptr) {
                sp<BBinder> b((BBinder*)tr.cookie);
                const status_t error = b->transact(tr.code, buffer, &reply, tr.flags);
                if (error < NO_ERROR) reply.setError(error);

            } else {
                const status_t error = the_context_object->transact(tr.code, buffer, &rep
```

```
                if (error < NO_ERROR) reply.setError(error);
            }
            . . . . . .
            if ((tr.flags & TF_ONE_WAY) == 0)
            {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
                sendReply(reply, 0);
            }
            . . . . . .
            . . . . . .
        }
        break;

    . . . . . .
    . . . . . .
    return result;
}
```

最关键的一句当然是b->transact()啦，此时b的值来自于binder_transaction_data的cookie域，本质上等于驱动层所记录的binder_node节点的cookie域值，这个值在用户态就是BBinder指针。

在调用完transact()动作后，executeCommand()会判断tr.flags有没有携带TF_ONE_WAY标记，如果没有携带，说明这次传输是需要回复的，于是调用sendReply()进行回复。

2 小结

至此，《红茶一杯话Binder（传输机制篇）》的上、中、下三篇文章总算写完了。限于个人水平，文中难免有很多细节交代不清，还请各位看官海涵。作为我个人而言，只是尽力尝试把一些底层机制说得清楚一点儿，奈何Android内部的代码细节繁杂，逻辑交叠，往往搞得人头昏脑涨，所以我也只是针对其中很小的一部分进行阐述而已。因为本人目前的主要兴趣已经不在binder了，所以这篇文章耽误了好久才写完，呵呵，见谅见谅。