紅茶一杯話Binder

(傳輸機制篇_下)

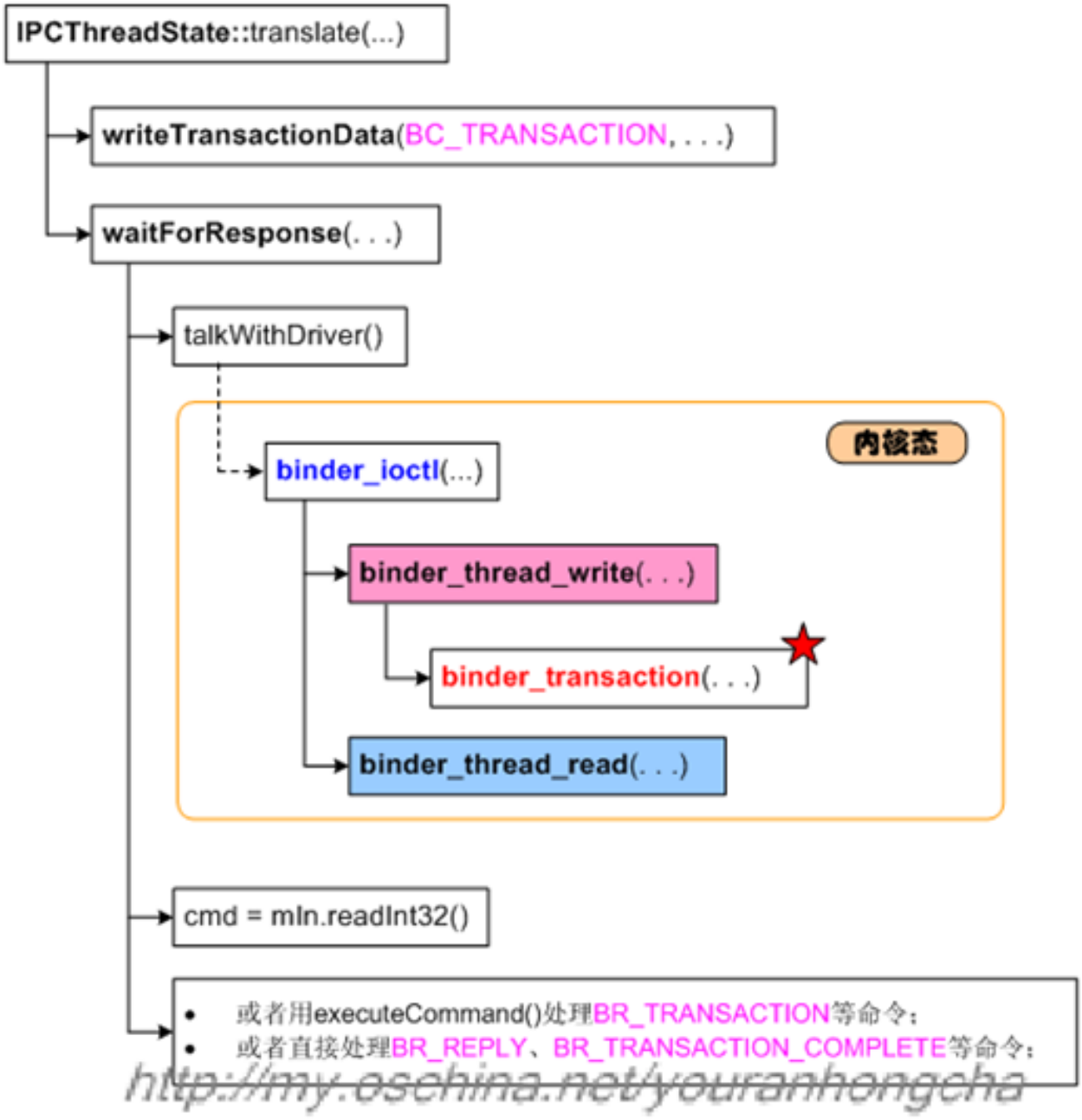
侯亮

1 事務的傳遞和處理

從IPCThreadState的角度看，它的transact()函數是通過向binder驅動發出BC_TRANSACTION語義，來表達其傳輸意圖的，而後如有必要，它會等待從binder發回的回饋，這些回饋語義常常以"BR_ "開頭。另一方面，當IPCThreadState作為處理命令的一方需要向發起方反饋信息的話，它會調用sendReply()函數，向binder驅動發出BC_REPLY語義。當BC_語義經由binder驅動遞送到目標端時，會被binder驅動自動修改為相應的BR_語義，這個我們在後文再細說。

當語義傳遞到binder驅動後，會走到binder_ioctl()函數，該函數又會調用到binder_thread_write()和binder_thread_read()：

在上一篇文章中，我們大體闡述了一下binder_thread_write()和binder_thread_read()的喚醒與被喚醒關係，而且還順帶在"傳輸機制的大體運作"小節中提到了todo隊列的概念。本文將在此基礎上再補充一些知識。需要強調的是，我們必須重視binder_thread_write()和binder_thread_read()，因為事務的傳遞和處理就位於這兩個函數中，它們的調用示意圖如下：



binder_thread_write()的代碼截選如下。因為本文主要關心傳輸方面的問題，所以只摘取了case BC_TRANSACTION、case BC_REPLY部分的代碼：

```
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;
}
```

這部分代碼比較簡單，主要是從用戶態拷貝來binder_transaction_data數據，並傳給binder_transaction()函數進行實際的傳輸。而binder_transaction()可是需要我們費一點兒力氣去分析的，大家深吸一口氣，準備開始。

1.1 BC_TRANSACTION事務（攜帶TF_ONE_WAY標記）的處理

首先我們要認識到，同樣是BC_TRANSACTION事務，帶不帶TF_ONE_WAY標記還是有所不同的。我們先看相對簡單的攜帶TF_ONE_WAY標記的BC_TRANSACTION事務，這種事務是不需要回復的。

1.1.1 binder_transaction()

此時，binder_transaction()所做的工作大概有：

1. 找目標binder_node；
2. 找目標binder_proc；
3. 分析並插入紅黑樹節點；（我們在上一篇文章中已在說過這部分的機理了，只是當時沒有貼出相應的代碼）
4. 創建binder_transaction節點，並將其插入目標進程的todo列表；
5. 嘗試喚醒目標進程。

binder_transaction()代碼截選如下：

```
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;
    . . . . . .
    . . . . . .
    {
        //先從tr->target.handle句柄值，找到對應的binder_ref節點，及binder_node節點
         if (tr->target.handle)
        {
            struct binder_ref * ref ;
             ref = binder_get_ref(proc, tr->target.handle);
            . . . . . .
            target_node = ref ->node;
        }
       else
        {
            //如果句柄值為0，則獲取特殊的binder_context_mgr_node節點，
            //即Service Manager Service對應的節點
            target_node = binder_context_mgr_node;
            . . . . . .
        }
        // 得到目標進程的binder_proc
```

```
        target_proc = target_node->proc;
        . . . . . .
    }

    //對於帶TF_ONE_WAY標記的BC_TRANSACTION來說，此時target_thread為NULL，
    //所以準備向binder_proc的todo中加節點
    . . . . . .
        target_list = &target_proc->todo;
        target_wait = &target_proc->wait;
    . . . . . .

    //創建新的binder_transaction節點。
    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;

    // 將binder_transaction_data的code、flags域記入binder_transaction節點。
    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;
    . . . . . .

    //下面的代碼分析所傳數據中的所有binder對象，如果是binder實體的話，要在紅黑樹中添加相應的節
點。
    //首先，從用戶態獲取所傳輸的數據，以及數據裡的binder對象的偏移信息
    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))
```

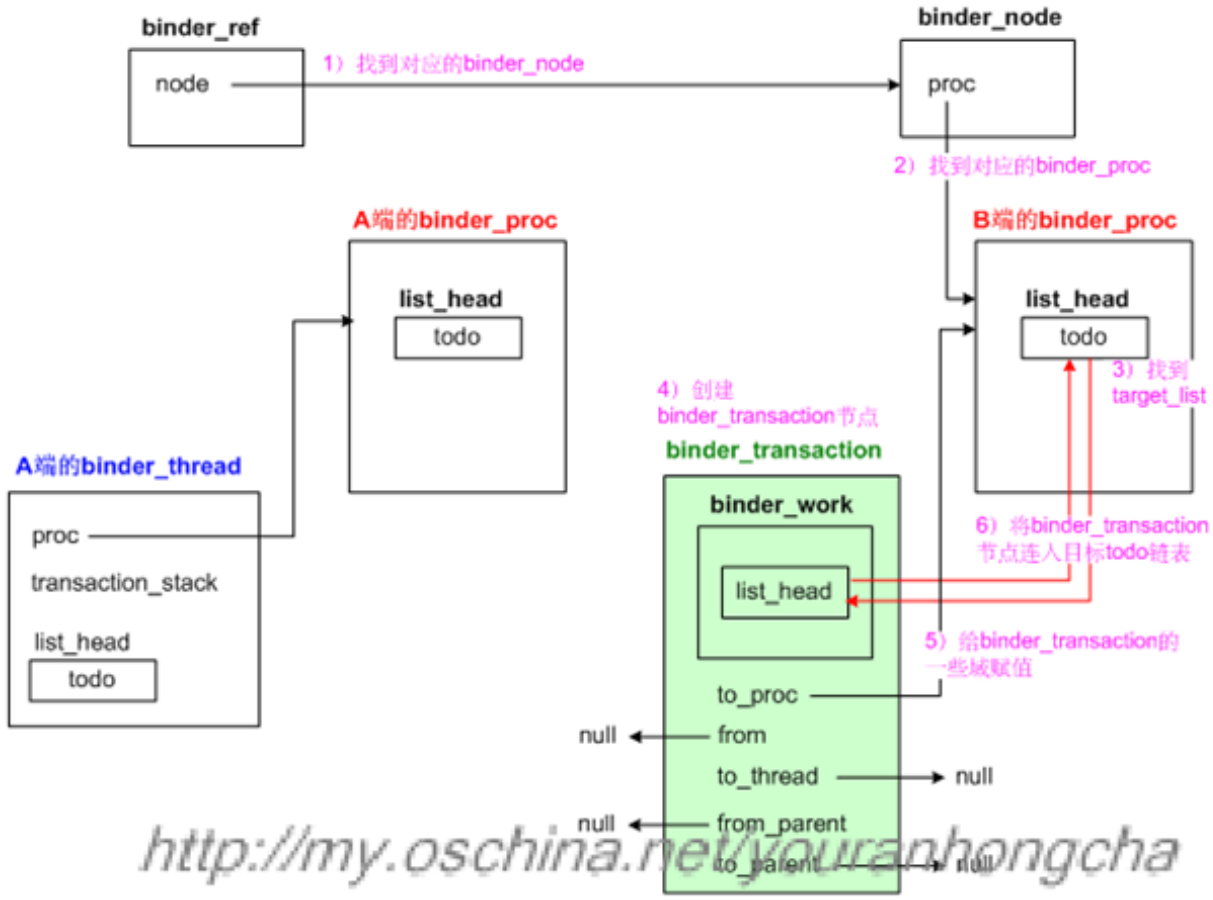
```
        . . . . . .
    . . . . . .
    //遍歷每個flat_binder_object信息，創建必要的紅黑樹節點....
    for (; offp < off_end; offp++)
    {
        struct flat_binder_object *fp;
        . . . . . .
        fp = ( struct flat_binder_object *)(t->buffer->data + *offp);

        switch (fp->type)
        {
        case BINDER_TYPE_BINDER:
         case BINDER_TYPE_WEAK_BINDER:
        {
            //如果是binder實體
            struct binder_ref * ref ;
             struct binder_node *node = binder_get_node(proc, fp->binder);
             if (node == NULL)
            {
                // 又是"沒有則創建"的做法，創建新的binder_node節點
               node = binder_new_node(proc, fp->binder, fp->cookie);
                . . . . . .
            }
            . . . . . .
            //必要時，會在目標進程的binder_proc中創建對應的binder_ref紅黑樹節點
            ref = binder_get_ref_for_node(target_proc, node);
            . . . . . .
            //修改所傳數據中的flat_binder_object信息，因為遠端的binder實體到了目標
            //端，就變為binder代理了，所以要記錄下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);
             //有時候需要對flat_binder_object做必要的修改，比如將BINDER_TYPE_HANDLE
            //改為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;

        // 終於把binder_transaction節點插入target_list（即目標todo隊列）了。
        list_add_tail(&t->work.entry, target_list);
        . . . . . .
        list_add_tail(&tcomplete->entry, &thread->todo);

        //傳輸動作完畢，現在可以喚醒系統中其他相關線程了，wake up!
        if (target_wait)
            wake_up_interruptible(target_wait);
        return ;
        . . . . . .
        . . . . . .
    }
```

雖然已經是截選，代碼卻仍然顯得冗長。這也沒辦法，Android frameworks裡的很多代碼都是這個樣子，又臭又長，大家湊合著看吧。我常常覺得google的工程師多少應該因這樣的代碼而感到臉紅，不過，

哎，這有點兒說遠了。

我們畫一張示意圖，如下：



上圖體現了從binder_ref找到“目標binder_node”以及“目標binder_proc”的意思，其中“A端”表示發起方，“B端”表示目標方。可以看到，攜帶TF_ONE_WAY標記的事務，其實是比較簡單的，驅動甚至不必費心去找目標線程，只需要創建一個binder_transaction節點，並插入目標binder_proc的todo鍊錶即可。

另外，在將binder_transaction節點插入目標todo鍊錶之前，binder_transaction()函數用一個for循環分析了需要傳輸的數據，並為其中包含的binder對像生成了相應的紅黑樹節點。

再後來，binder_transaction節點成功插入目標todo鍊錶，此時說明目標進程有事情可做了，於是binder_transaction()函數會調用wake_up_interruptible()喚醒目標進程。

1.1.2 binder_thread_read()

當目標進程被喚醒時，會接著執行自己的binder_thread_read()，嘗試解析並執行那些剛收來的工

作。無論收來的工作來自於“binder_proc的todo鍊錶”，還是來自於某“binder_thread的todo鍊錶”，現在要開始從todo鍊錶中摘節點了，而且在完成工作之後，會徹底刪除binder_transaction節點。

binder_thread_read()的代碼截選如下：

```
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:
    // 優先考慮thread節點的todo鍊錶中有沒有工作需要完成
    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;

    //如果是非阻塞的情況，ret值非0表示出了問題，所以return。
    //如果是阻塞（non_block）情況，ret值非0表示等到的結果出了問題，所以也return。
    if (ret)
         return ret;

    while (1)
    {
        . . . . . .
        //讀取binder_thread或binder_proc中todo列表的第一個節點
```

```
        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;
            . . . . . .
              // 將binder_transaction節點從todo隊列摘下來
            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;
            //用目標binder_node中記錄的cookie值給binder_transaction_data的cookie域賦值，
              //這個值就是目標binder實體的地址
              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只是記錄了binder緩衝區中的地址信息，並再做copy動
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 *));

         //將cmd命令寫入用戶態，此時應該是BR_TRANSACTION
        if (put_user(cmd, (uint32_t __user *)ptr))
             return -EFAULT;
        ptr += sizeof (uint32_t);
          //當然，binder_transaction_data本身也是要copy到用戶態的
        if (copy_to_user(ptr, &tr, sizeof (tr)))
             return -EFAULT;
        . . . . . .
        . . . . . .
        // 將binder_transaction節點從todo隊列摘下來
        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情況，此時會刪除binder_transaction節點
            kfree(t);
            binder_stats_deleted(BINDER_STAT_TRANSACTION);
        }
        break ;
    }
```

```
    . . . . . .
    . . . . . .
    return 0;
}
```

簡單說來就是，如果沒有工作需要做，binder_thread_read()函數就進入睡眠或返回，否則binder_thread_read()函數會從todo隊列摘下了一個節點，並把節點裡的數據整理成一個binder_transaction_data結構，然後通過copy_to_user()把該結構傳到用戶態。因為這次傳輸帶有TF_ONE_WAY標記，所以copy完後，只是簡單地調用kfree(t)把這個binder_transaction節點幹掉了。

binder_thread_read()嘗試調用wait_event_interruptible()或wait_event_interruptible_exclusive()來等待待處理的工作。wait_event_interruptible()是個宏定義，和wait_event()類似，不同之處在於前者不但會判斷"甦醒條件"，還會判斷當前進程是否帶有掛起的系統信號，當"甦醒條件"滿足時（比如binder_has_thread_work (thread)返回非0值），或者有掛起的系統信號時，表示進程有工作要做了，此時wait_event_interruptible()將跳出內部的for循環。如果的確不滿足跳出條件的話，wait_event_interruptible()會進入掛起狀態。

請注意給binder_transaction_data的cookie賦值的那句：

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

binder_node節點裡儲存的cookie值終於發揮作用了，這個值反饋到用戶態就是目標binder實體的BBinder指針了。

另外，在調用copy_to_user()之前，binder_thread_read()先通過put_user()向上層拷貝了一個命令碼，在當前的情況下，這個命令碼是BR_TRANSACTION。想當初，內核態剛剛從用戶態拷貝來的命令碼是BC_TRANSACTION，現在要發給目標端了，就變成了BR_TRANSACTION。

1.2 BC_TRANSACTION事務（不帶TF_ONE_WAY標記）

1.2.1 再說binder_transaction()

然而，對於不帶TF_ONE_WAY標記的BC_TRANSACTION事務來說，情況就沒那麼簡單了。因為binder驅動不僅要找到目標進程，而且還必須努力找到一個明確的目標線程。正如我們前文所說，binder驅動希望可以充分複用目標進程中的binder工作線程。

那麼，哪些線程（節點）是可以被復用的呢？我們再整理一下binder_transaction()代碼，本次主要截選不帶TF_ONE_WAY標記的代碼部分：

```
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;
            . . . . . .
        }
        . . . . . .
        // 先確定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這條鍊錶中，最後一個可以和target_proc匹配
            //的binder_transaction節點，
            //這個節點的from就是我們要找的"目標線程"
            while (tmp)
            {
                if (tmp->from && tmp->from->proc == target_proc)
                    target_thread = tmp->from;
                tmp = tmp->from_parent;
            }
        }
    . . . . . .
    //要確定target_list和target_wait了，如果能找到"目標線程"，它們就來自目標線程，否則
    //就只能來自目標進程了。
```

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

    t->from = thread;     //新節點的from域記錄事務的發起線程

    t->sender_euid = proc->tsk->cred->euid;
    t->to_proc = target_proc;
    t->to_thread = target_thread;     //新節點的to_thread域記錄事務的目標線程

    t->code = tr->code;
    t->flags = tr->flags;
    t->priority = task_nice(current);
    // 從binder buffer中申請一個區域，用於存儲待傳輸的數據
    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;
    . . . . . .
    //從用戶態拷貝來待傳輸的數據
    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)) {
        . . . . . .
```

```
    }
    //遍歷每個flat_binder_object信息，創建必要的紅黑樹節點....
    for (; offp < off_end; offp++)
    {
        struct flat_binder_object *fp;
        . . . . . .
        . . . . . .
    }

    . . . . . .
        t->need_reply = 1;
        // 新binder_transaction節點成為發起端transaction_stack棧的新棧頂
        t->from_parent = thread->transaction_stack;
        thread->transaction_stack = t;
    . . . . . .
    t->work.type = BINDER_WORK_TRANSACTION;

    // 終於把binder_transaction節點插入target_list（即目標todo隊列）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，獲取目標binder_proc的部分和前一小節沒什麼不同，但是因為本次傳輸不再攜帶TF_ONE_WAY標記了，所以函數中會盡力去查一個合適的"目標binder_thread"，此時會用到binder_thread裡的"事務棧"（transaction_stack）概念。

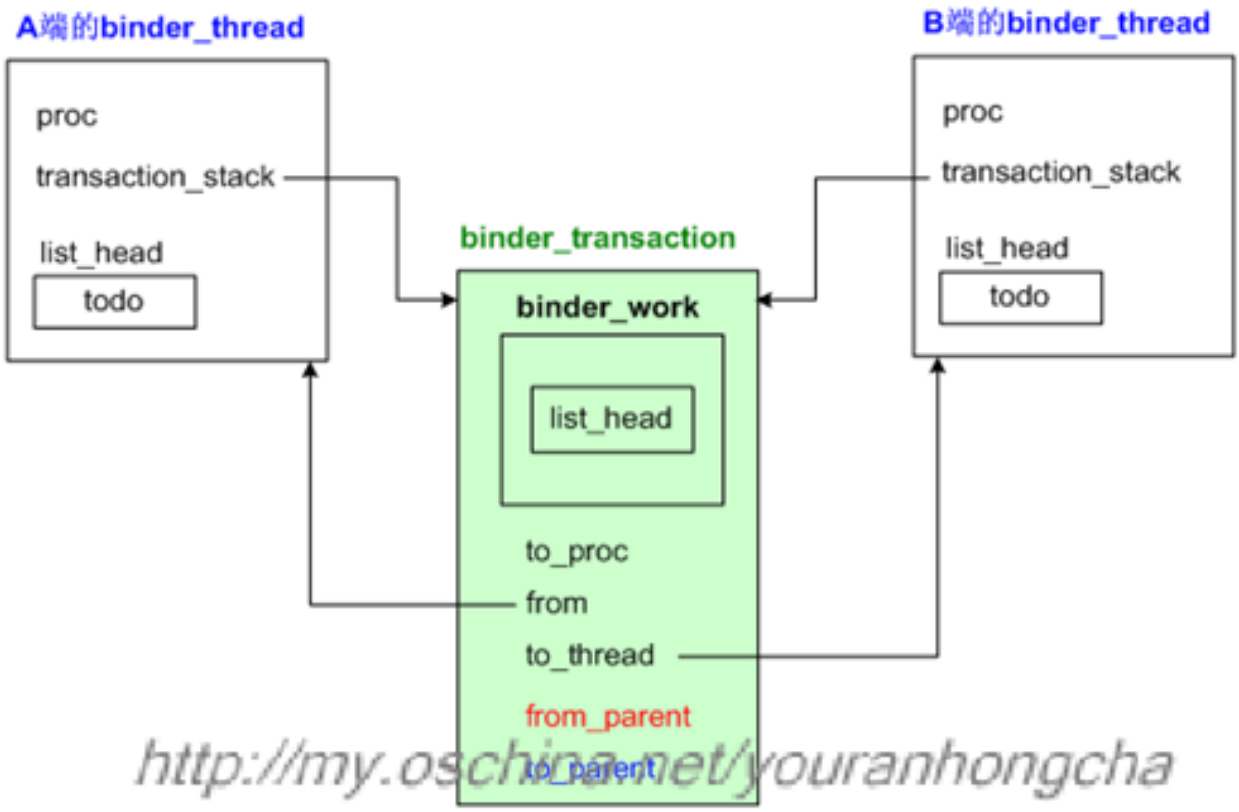
那麼，怎麼找"目標binder_thread"呢？首先，我們很清楚"發起端"的binder_thread節點是哪個，而且也可以找到"目標端"的binder_proc，這就具有了搜索的基礎。在binder_thread節點的transaction_stack域裡，記錄了和它相關的若干binder_transaction，這些binder_transaction事務在邏輯上具有類似堆棧的屬性，也就是說"最後入棧的事務"會最先處理。

從邏輯上說，線程節點的transaction_stack域體現了兩個方面的意義：

1. 這個線程需要別的線程幫牠做某項工作；

2. 別的線程需要這個線程做某項工作；

因此，一個工作節點（即binder_transaction節點）往往會插入兩個transaction_stack堆棧，示意圖如下：



當binder_transaction節點插入"發起端"的transaction_stack棧時，它是用from_parent域來連接堆棧中其他節點的。而當該節點插入"目標端"的transaction_stack棧時，卻是用to_parent域來連接其他節點的。關於插入目標端堆棧的動作，位於binder_thread_read()中，我們在後文會看到。

這麼看來，from_parent域其實將一系列邏輯上有先後關係的若干binder_transaction節點串接起來了，而且這些binder_transaction節點可能是由不同進程、線程發起的。那麼我們只需遍歷一下這個堆棧裡的事務，看哪個事務的"from線程所屬的進程"和"目標端的binder_proc"一致，就說明這個from線程正是我們要找的目標線程。為什麼這麼說呢？這是因為我們的新事務將成為binder_transaction的新棧頂，而這個堆棧裡其他事務一定是在新棧頂事務處理完後才會處理的，因此堆棧裡某個事務的發起端線程可以理解為正處於等待狀態，如果這個發起端線程所從屬的進程恰恰又是我們新事務的目標進程的話，那就算合拍了，這樣就找到"目標binder_thread"了。我把相關的代碼再抄一遍：

```
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;
}
```

代碼用while循環來遍歷thread->transaction_stack，發現tmp->from->proc == target_proc，就算找到了。

如果能夠找到"目標binder_thread"的話，binder_transaction事務就會插到它的todo隊列去。不過有時候找不到"目標binder_thread"，那麼就只好退而求其次，插入binder_proc的todo隊列了。再接下來的動作沒有什麼新花樣，大體上會嘗試喚醒目標進程。

1.2.2 再說binder_thread_read()

目標進程在喚醒後，會接著當初阻塞的地方繼續執行，這個已在前一小節闡述過，我們不再贅述。值得一提的是binder_thread_read()中的以下句子：

```
        // 將binder_transaction節點從todo隊列摘下來
         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情況，此時會刪除binder_transaction節點
             kfree(t);
             binder_stats_deleted(BINDER_STAT_TRANSACTION);
         }
```

因為沒有攜帶TF_ONE_WAY標記，所以此處會有一個入棧操作，binder_transaction節點插入了目標線程的transaction_stack堆棧，而且是以to_thread域來連接堆棧中的其他節點的。

總體說來，binder_thread_read()的動作大體也就是：

1）利用wait_event_xxxx()讓自己掛起，等待下一次被喚醒；

2）喚醒後找到合適的待處理的工作節點，即binder_transaction節點；

3）把binder_transaction中的信息整理到一個binder_transaction_data中；

4）整理一個cmd整數值，具體數值或者為BR_TRANSACTION，或者為BR_REPLY；

5）將cmd數值和binder_transaction_data拷貝到用戶態；

6）如有必要，將得到的binder_transaction節點插入目標端線程的transaction_stack堆棧中。

1.2.3 目標端如何處理傳來的事務

binder_thread_read()本身只負責讀取數據，它並不解析得到的語義。具體解析語義的動作並不在內核態，而是在用戶態。

我們再回到用戶態的IPCThreadState::waitForResponse()函數。

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    while (1)
    {
        // talkWithDriver()內部會完成跨進程事務
        if ((err = talkWithDriver()) < NO_ERROR)
            break ;

        // 事務的回復信息被記錄在mIn中，所以需要進一步分析這個回復
        . . . . . .
        cmd = mIn.readInt32();
        . . . . . .

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

當發起端調用binder_thread_write()喚醒目標端的進程時，目標進程會從其上次調用binder_thread_read()的地方甦醒過來。輾轉跳出上面的talkWithDriver()函數，並走到executeCommand()一句。

因為binder_thread_read()中已經把BR_命令整理好了，所以executeCommand()當然會走到

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,
&reply, tr.flags);
                 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;
    }
```

最關鍵的一句當然是b->transact()啦，此時b的值來自於binder_transaction_data的cookie域，本質上等於驅動層所記錄的binder_node節點的cookie域值，這個值在用戶態就是BBinder指針。

在調用完transact()動作後，executeCommand()會判斷tr.flags有沒有攜帶TF_ONE_WAY標記，如果沒有攜帶，說明這次傳輸是需要回复的，於是調用sendReply()進行回复。

2 小結

至此，《紅茶一杯話Binder（傳輸機制篇）》的上、中、下三篇文章總算寫完了。限於個人水平，文中難免有很多細節交代不清，還請各位看官海涵。作為我個人而言，只是盡力嘗試把一些底層機制說得清楚一點兒，奈何Android內部的代碼細節繁雜，邏輯交疊，往往搞得人頭昏腦漲，所以我也只是針對其中很小的一部分進行闡述而已。因為本人目前的主要興趣已經不在binder了，所以這篇文章耽誤了好久才寫完，呵呵，見諒見諒。