博客分類

- Android Frameworks 4.x (5)
- 日常記錄(0)
- 轉貼的文章(0)

閱讀排行

1. 1. 紅茶一杯話Binder（傳輸機制篇_上）
2. 2. 紅茶一杯話Binder（傳輸機制篇_中）
3. 3. 紅茶一杯話Binder（ServiceManager篇）
4. 4. 紅茶一杯話Binder（初始篇）
5. 5. AlarmManager研究

最新評論

- @Jessie0227：寫的太好了!!請問何時會有下一篇呢(期待中) 查看»
- @公子無憂：請教個問題,BC和BR的命令是什麼關係,分別什麼時候... 查看»
- @xkk609：分析比較深入。 查看»
- @RenKaidi：不明覺厲！ 查看»
- @enull：Mark一下，自學中，感謝。 查看»
- @xway：準備空下來的時候學習下Android開發，這樣認真的... 查看»
- @徐慶-neo：贊一個，非常好的文章 查看»
- @翠屏阿姨：我是沙發，曾經看過沒看懂，今天趁著這篇文章再看... 查看»
- @simonws：fucking source code 查看»
- @悠然紅茶：引用來自“simonws”的評論你是怎麼研究的？無他... 查看»

訪客統計

- **今日訪問： 3**
- **昨日訪問： 7**
- **本周訪問： 19**
- **本月訪問： 10**
- **所有訪問： 2285**

紅茶一杯話Binder（傳輸機制篇_中）

紅茶一杯話Binder

（傳輸機制篇_中）

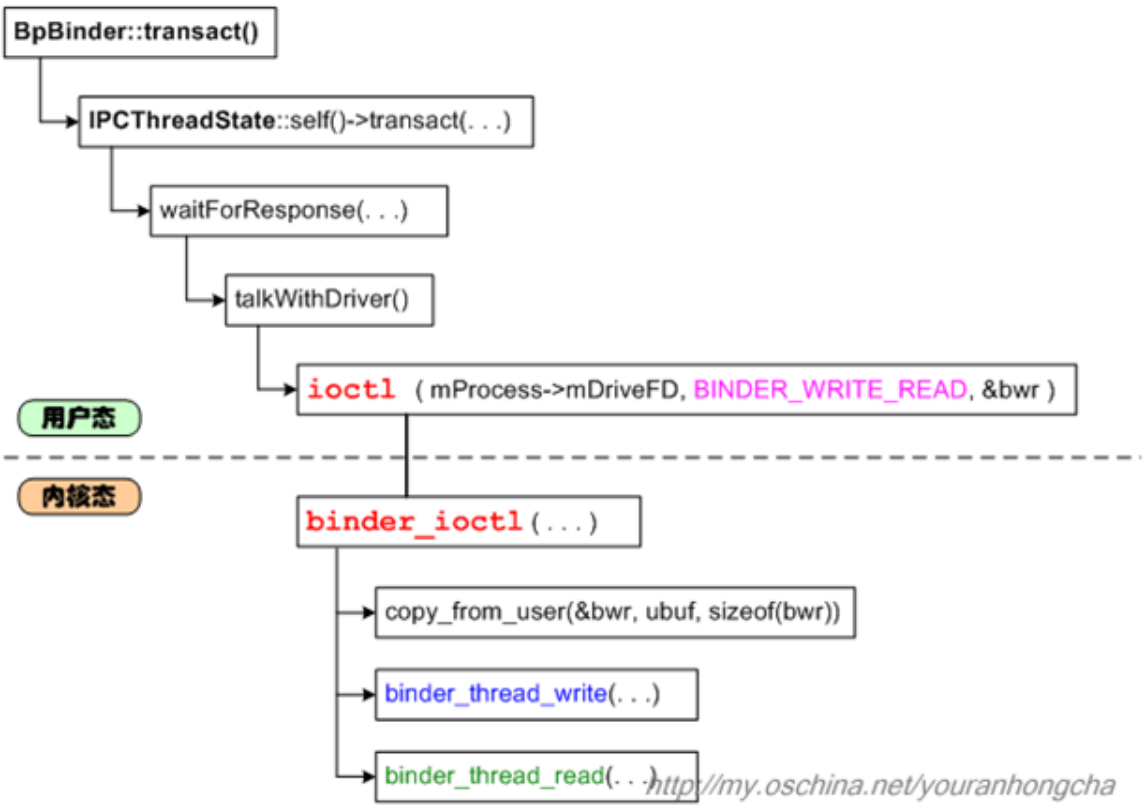
侯亮

1 談談底層IPC機制吧

在上一篇文章的最後，我們說到BpBinder將數據發到了Binder驅動。然而在驅動層，這部分數據又是如何傳遞到BBinder一側的呢？這裡面到底藏著什麼貓膩？另外，上一篇文章雖然闡述了4棵紅黑樹，但是並未說明紅黑樹的節點到底是怎麼產生的。現在，我們試著回答這些問題。

1.1 概述

在Binder驅動層，和ioctl()相對的動作是binder_ioctl()函數。在這個函數里，會先調用類似copy_from_user()這樣的函數，來讀取用戶態的數據。然後，再調用binder_thread_write()和binder_thread_read()進行進一步的處理。我們先畫一張調用關係圖：



binder_ioctl()調用binder_thread_write()的代碼是這樣的：

```
if (bwr.write_size > 0)
{
    ret = binder_thread_write(proc, thread, ( void __user *)bwr.write_buffer,
                              bwr.write_size, &bwr.write_consumed);
    if (ret < 0)
```

```
    {
        bwr.read_consumed = 0;
        if (copy_to_user(ubuf, &bwr, sizeof (bwr)))
            ret = -EFAULT;
        goto err;
    }
}
```

注意binder_thread_write()的前兩個參數，一個是binder_proc指針，另一個是binder_thread指針，表示發起傳輸動作的進程和線程。binder_proc不必多說了，那個binder_thread是怎麼回事？大家應該還記得前文提到的binder_proc裡的4棵樹吧，此處的binder_thread就是從threads樹中查到的節點。

```
thread = binder_get_thread(proc);
```

binder_get_thread()的代碼如下：

```
static  struct binder_thread *binder_get_thread( struct binder_proc *proc)
{
    struct binder_thread *thread = NULL;
     struct rb_node *parent = NULL;
     struct rb_node **p = &proc->threads.rb_node;

    //盡量從threads樹中查找和current線程匹配的binder_thread節點
    while (*p)
    {
        parent = *p;
        thread = rb_entry(parent, struct binder_thread, rb_node);
         if (current->pid < thread->pid)
            p = &(*p)->rb_left;
        else  if (current->pid > thread->pid)
            p = &(*p)->rb_right;
        else
            break ;
    }

    // “找不到就創建”一個binder_thread節點
    if (*p == NULL)
    {
        thread = kzalloc( sizeof (*thread), GFP_KERNEL);
         if (thread == NULL)
             return NULL;
        binder_stats_created(BINDER_STAT_THREAD);
        thread->proc = proc;
        thread->pid = current->pid;
        init_waitqueue_head(&thread->wait);
        INIT_LIST_HEAD(&thread->todo);

         // 新binder_thread節點插入紅黑樹
        rb_link_node(&thread->rb_node, parent, p);
        rb_insert_color(&thread->rb_node, &proc->threads);
        thread->looper |= BINDER_LOOPER_STATE_NEED_RETURN;
        thread->return_error = BR_OK;
        thread->return_error2 = BR_OK;
    }
    return thread;
}
```

binder_get_thread()會盡量從threads樹中查找和current線程匹配的binder_thread節點，如果找不到，就會創建一個新的節點並插入樹中。這種“找不到就創建”的做法，在後文還會看到，我們暫時先不多說。

在調用binder_thread_write()之後，binder_ioctl()接著調用到binder_thread_read()，此時往往需要等待遠端的回復，所以binder_thread_read()會讓線程睡眠，把控制權讓出來。在未來的某個時刻，遠端處理完此處發去的語義，就會著手發回回复。當回復到達後，線程會從以前binder_thread_read()睡眠的地方醒來，並進一步解析收到的回复。

以上所說，都只是概要性的闡述，下面我們要深入一些細節了。

1.2 要進行跨進程調用，需要考慮什麼？

我們可以先考慮一下，要設計跨進程調用機制，大概需要考慮什麼東西呢？我們列一下：

1） 發起端：肯定包括發起端所從屬的進程，以及實際執行傳輸動作的線程。當然，發起端的BpBinder更是重中之重。

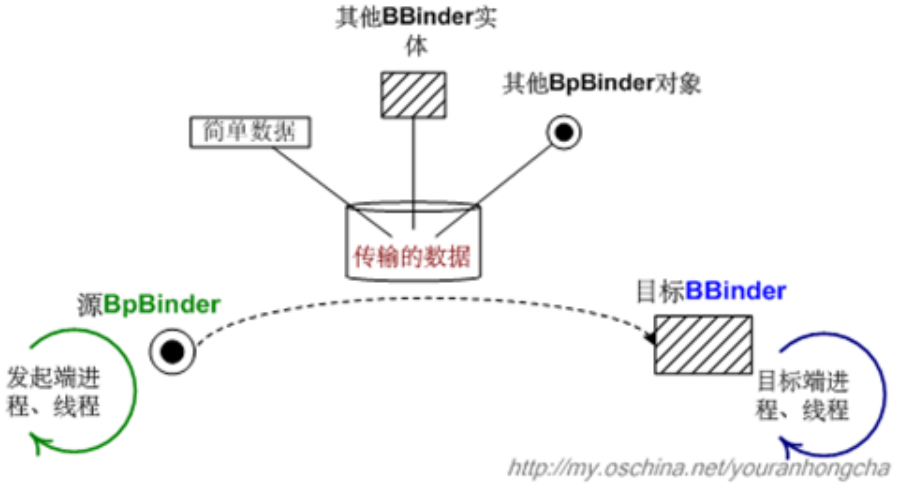
2） 接收端：包括與發起端對應的BBinder，以及目標進程、線程。

3） 待傳輸的數據：其實就是前文IPCThreadState::writeTransactionData()代碼中的binder_transaction_data了，需要注意的是，這份數據中除了包含簡單數據，還可能包含其他binder對象噢，這些對像或許對應binder代理對象，或許對應binder實體對象，視具體情況而定。

4） 如果我們的IPC動作需要接收應答（reply），該如何保證應答能準確無誤地傳回來？

5） 如何讓系統中的多個傳輸動作有條不紊地進行。

我們可以先畫一張示意圖：

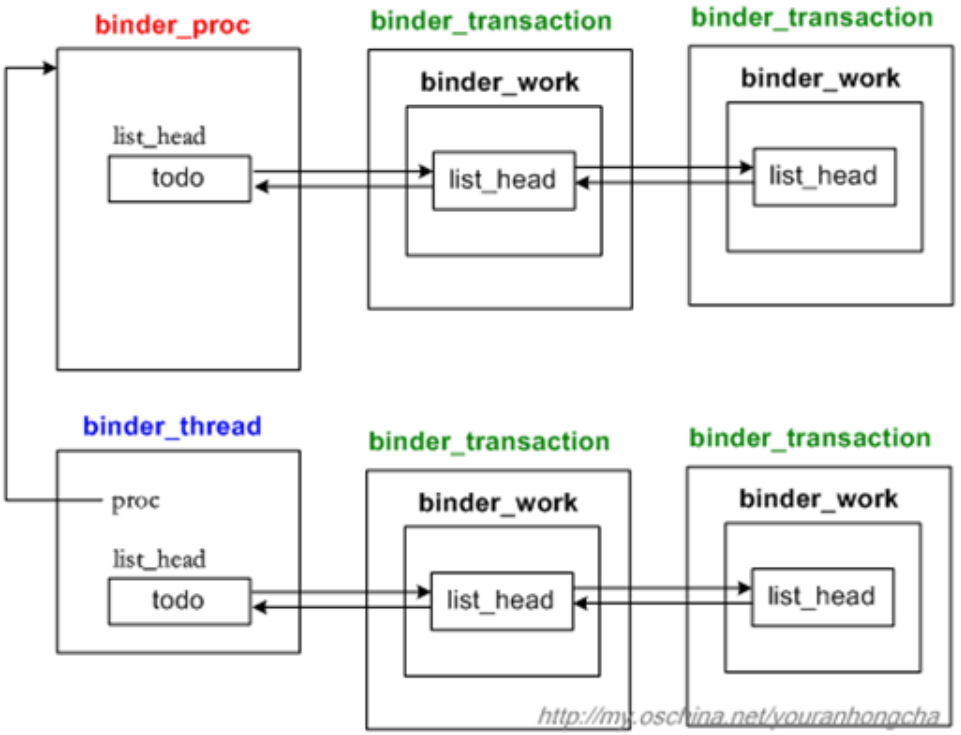


然而這張圖似乎還是串接不起整個傳輸過程，圖中的“傳輸的數據”到底是怎麼發到目標端的呢？要回答這個問題，我們還得繼續研究Binder IPC機制的實現機理。

1.3 傳輸機制的大體運作

Binder IPC機制的大體思路是這樣的，它將每次“傳輸並執行特定語義的”工作理解為一個小事務，既然所傳輸的數據是binder_transaction_data類型的，那麼這種事務的類名可以相應地定為binder_transaction。系統中當然會有很多事務啦，那麼發向同一個進程或線程的若干事務就必須串行化起來，因此binder驅動為進程節點（binder_proc）和線程節點（binder_thread）都設計了個todo隊列。todo隊列的職責就是“串行化地組織待處理的事務”。

下圖繪製了一個進程節點，以及一個從屬於該進程的線程節點，它們各帶了兩個待處理的事務（binder_transaction）：



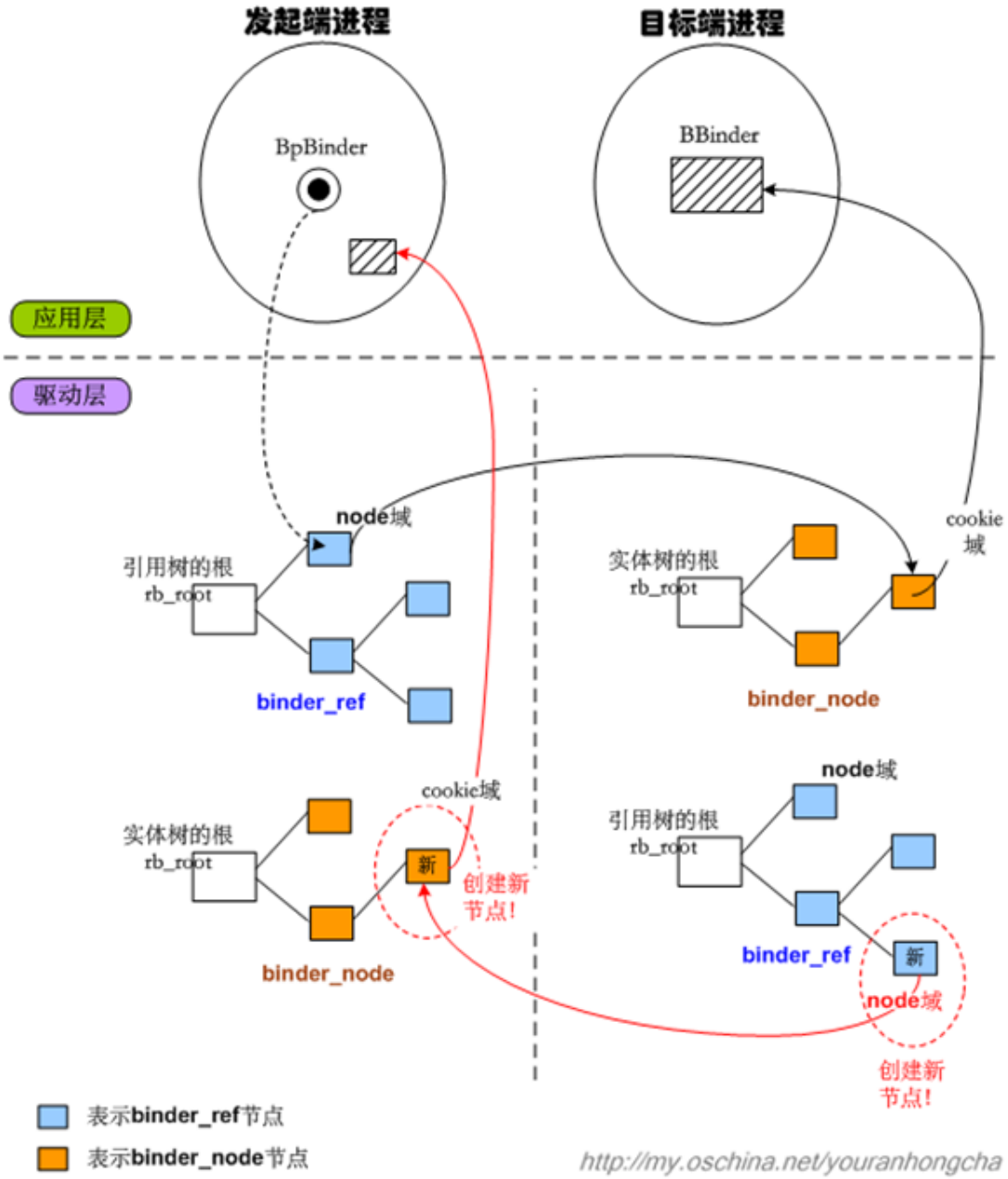
這樣看來，傳輸動作的基本目標就很明確了，就是想辦法把發起端的一個binder_transaction節點，插入到目標端進程或其合適子線程的todo隊列去。

可是，該怎麼找目標進程和目標線程呢？基本做法是先從發起端的BpBinder開始，找到與其對應的binder_node節點，這個在前文闡述binder_proc的4棵紅黑樹時已經說過了，這裡不再贅述。總之拿到目標binder_node之後，我們就可以通過其proc域，拿到目標進程對應的binder_proc了。如果偷懶的話，我們直接把binder_transaction節點插到這個binder_proc的todo鍊錶去，就算完成傳輸動作了。當然，binder驅動做了一些更精細的調整。

binder驅動希望能把binder_transaction節點盡量放到目標進程裡的某個線程去，這樣可以充分利用這個進程中的binder工作線程。比如一個binder線程目前正睡著，它在等待其他某個線程做完某個事情后才會醒來，而那個工作又偏偏需要在當前這個binder_transaction事務處理結束後才能完成，那麼我們就可以讓那個睡著的線程先去做當前的binder_transaction事務，這就達到充分利用線程的目的了。反正不管怎麼說，如果binder驅動可以找到一個合適的線程，它就會把binder_transaction節點插到它的todo隊列去。而如果找不到合適的線程，還可以把節點插入目標binder_proc的todo隊列。

1.4 紅黑樹節點的產生過程

另一個要考慮的東西就是binder_proc裡的那4棵樹啦。前文在闡述binder_get_thread()時，已經看到過向threads樹中添加節點的動作。那麼其他3棵樹的節點該如何添加呢？其實，秘密都在傳輸動作中。要知道，binder驅動在傳輸數據的時候，可不是僅僅簡單地遞送數據噢，它會分析被傳輸的數據，找出其中記錄的binder對象，並生成相應的樹節點。如果傳輸的是個binder實體對象，它不僅會在發起端對應的nodes樹中添加一個binder_node節點，還會在目標端對應的refs_by_desc樹、refs_by_node樹中添加一個binder_ref節點，而且讓binder_ref節點的node域指向binder_node節點。我們把前一篇文章的示意圖加以修改，得到下圖：



圖中用紅色線條來表示傳輸binder實體時在驅動層會添加的紅黑樹節點以及節點之間的關係。

可是，驅動層又是怎麼知道所傳的數據中有多少binder對象，以及這些對象的確切位置呢？答案很簡單，是你告訴它的。大家還記得在向binder驅動傳遞數據之前，都是要把數據打成parcel包的吧。比如：

```
virtual status_t addService( const String16& name, const sp<IBinder>& service)
{
    Parcel data, reply;

    data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
    data.writeString16(name);
    data.writeStrongBinder(service);    //把一個binder實體“打扁”並寫入parcel
    status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
    return err == NO_ERROR ? reply.readExceptionCode() : err;
}
```

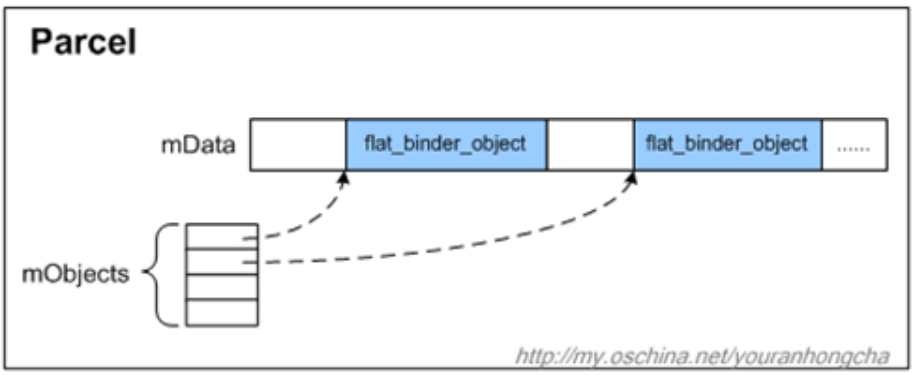
請大家注意上面data.writeStrongBinder()一句，它專門負責把一個binder實體“打扁”並寫入parcel。其代碼如下：

```
status_t Parcel::writeStrongBinder( const sp<IBinder>& val)
{
    return flatten_binder(ProcessState::self(), val, this );
}
```

```
status_t flatten_binder( const sp<ProcessState>& proc, const sp<IBinder>& binder, Parcel* out )
{
    flat_binder_object obj;
    . . . . . .
    if (binder != NULL) {
        IBinder *local = binder->localBinder();
        if (!local) {
            BpBinder *proxy = binder->remoteBinder();
            . . . . . .
            obj.type = BINDER_TYPE_HANDLE;
            obj.handle = handle;
            obj.cookie = NULL;
        } else {
            obj.type = BINDER_TYPE_BINDER;
            obj.binder = local->getWeakRefs();
            obj.cookie = local;
        }
    }
    . . . . . .
    return finish_flatten_binder(binder, obj, out );
}
```

看到了嗎？“打扁”的意思就是把binder對象整理成flat_binder_object變量，如果打扁的是binder實體，那麼flat_binder_object用cookie域記錄binder實體的指針，即BBinder指針，而如果打扁的是binder代理，那麼flat_binder_object用handle域記錄的binder代理的句柄值。

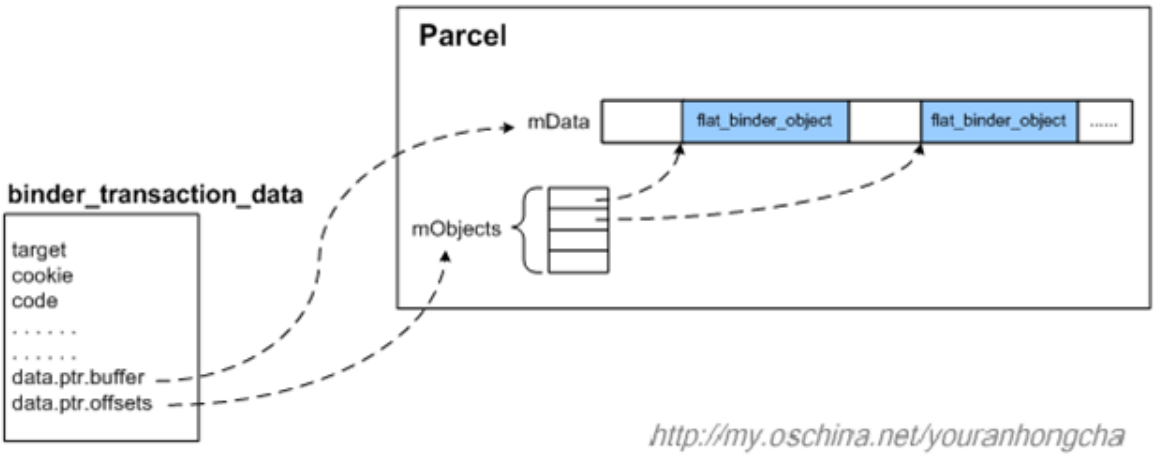
然後flatten_binder()調用了一個關鍵的finish_flatten_binder()函數。這個函數內部會記錄下剛剛被扁平化的flat_binder_object在parcel中的位置。說得更詳細點兒就是，parcel對象內部會有一個buffer，記錄著parcel中所有扁平化的數據，有些扁平數據是普通數據，而另一些扁平數據則記錄著binder對象。所以parcel中會構造另一個mObjects數組，專門記錄那些binder扁平數據所在的位置，示意圖如下：



一旦到了向驅動層傳遞數據的時候，IPCThreadState::writeTransactionData()會先把Parcel數據整理成一個binder_transaction_data數據，這個在上一篇文章已有闡述，但是當時我們並沒有太關心裡面的關鍵句子，現在我們把關鍵句子再列一下：

```
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;
    . . . . . .
        // 這部分是待傳遞數據
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        //這部分是扁平化的binder對像在數據中的具體位置
        tr.offsets_size = data.ipcObjectsCount()* sizeof (size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    . . . . . .
    mOut.write(&tr, sizeof (tr));
    . . . . . .
}
```

其中給tr.data.ptr.offsets賦值的那句，所做的就是記錄下“待傳數據”中所有binder對象的具體位置，示意圖如下：



因此，當binder_transaction_data傳遞到binder驅動層後，驅動層可以準確地分析出數據中到底有多少binder對象，並分別進行處理，從而產生出合適的紅黑樹節點。此時，如果產生的紅黑樹節點是binder_node的話，binder_node的cookie域會被賦值成flat_binder_object所攜帶的cookie值，也就是用戶態的BBinder地址值啦。這個新生成的binder_node節點被插入紅黑樹後，會一直嚴陣以待，以後當它成為另外某次傳輸動作的目標節點時，它的cookie域就派上用場了，此時cookie值會被反映到用戶態，於是用戶態就拿到了BBinder對象。

我們再具體看一下IPCThreadState::waitForResponse()函數，當它輾轉從睡眠態跳出來時，會進一步解析剛收到的命令，此時會調用executeCommand(cmd)一句。

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

    while (1)
    {
        if ((err = talkWithDriver()) < NO_ERROR) break ;
        . . . . . .
        switch (cmd)
        {
            . . . . . .
            . . . . . .
        default :
            err = executeCommand(cmd);
            . . . . . .
            break ;
        }
    }
    . . . . . .

    return err;
}
```

executeCommand()的代碼截選如下：

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    . . . . . .
    switch (cmd)
    {
        . . . . . .
        . . . . . .
    case BR_TRANSACTION:
        {
            binder_transaction_data tr;
            result = mIn.read(&tr, sizeof (tr));
            . . . . . .
            . . . . . .
            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);

            }
            . . . . . .

            if ((tr.flags & TF_ONE_WAY) == 0)
            {
                LOG_ONEWAY( "Sending reply to %d!" , mCallingPid);
                sendReply(reply, 0);
            }
            else
            {
                LOG_ONEWAY( "NOT sending reply to %d!" , mCallingPid);
            }
            . . . . . .
        }
        break ;
    . . . . . .
    . . . . . .
    default :
        printf( "*** BAD COMMAND %d received from Binder driver\n" , cmd);
        result = UNKNOWN_ERROR;
        break ;
    }
    . . . . . .
    return result;
}
```

請注意上面代碼中的sp<BBinder> b((BBinder*)tr.cookie)一句，看到了吧，驅動層的binder_node節點的cookie值終於發揮它的作用了，我們拿到了一個合法的sp<BBinder >。

接下來，程序走到b->transact()一句。transact()函數的代碼截選如下：

```
status_t BBinder::transact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    . . . . . .
    switch (code)
    {
        . . . . . .
        default :
            err = onTransact(code, data, reply, flags);
            break ;
    }
    . . . . . .
}
```

其中最關鍵的一句是調用onTransaction()。因為我們的binder實體在本質上都是繼承於BBinder的，而且我們一般都會重載onTransact()函數，所以上面這句onTransact()實際上調用的是具體binder實體的onTransact()成員函數。

Ok，說了這麼多，我們大概明白了binder驅動層的紅黑樹節點是怎麼產生的，以及binder_node節點的cookie值是怎麼派上用場的。限於篇幅，我們先在這裡打住。下一篇文章我們再來闡述binder事務的傳遞和處理方面的細節。

如需轉載本文內容，請註明出處。

http://my.oschina.net/youranhongcha/blog/152963

謝謝。

- « 紅茶一杯話Binder（傳輸機制篇_上）

共有3 條網友評論

- 1樓：xkk609發表於2013-08-21 17:09 回复此評論
分析比較深入。
- 2樓：公子無憂發表於2013-09-23 18:02 回复此評論
請教個問題,BC和BR的命令是什麼關係,分別什麼時候使用或轉換?
還有,博主的文章真精彩,很期待繼續,什麼時候更新更深入的剖析?
- 3樓：Jessie0227發表於2013-09-24 18:43 回复此評論
寫的太好了!!請問何時會有下一篇呢(期待中)

發表評論
文明上網，理性發言
回到頁首 | 回到評論列表

關閉相關文章閱讀

- 2013/08/02 紅茶一杯話Binder（初始篇）
- 2013/08/02 紅茶一杯話Binder（ServiceManager篇...
- 2013/08/12 紅茶一杯話Binder（傳輸機制篇_上）...
- 2013/08/04 Android Binder的使用和設計[androi...
- 2012/06/02 Android Binder IPC分析...