红茶一杯话Binder
（传输机制篇_中）

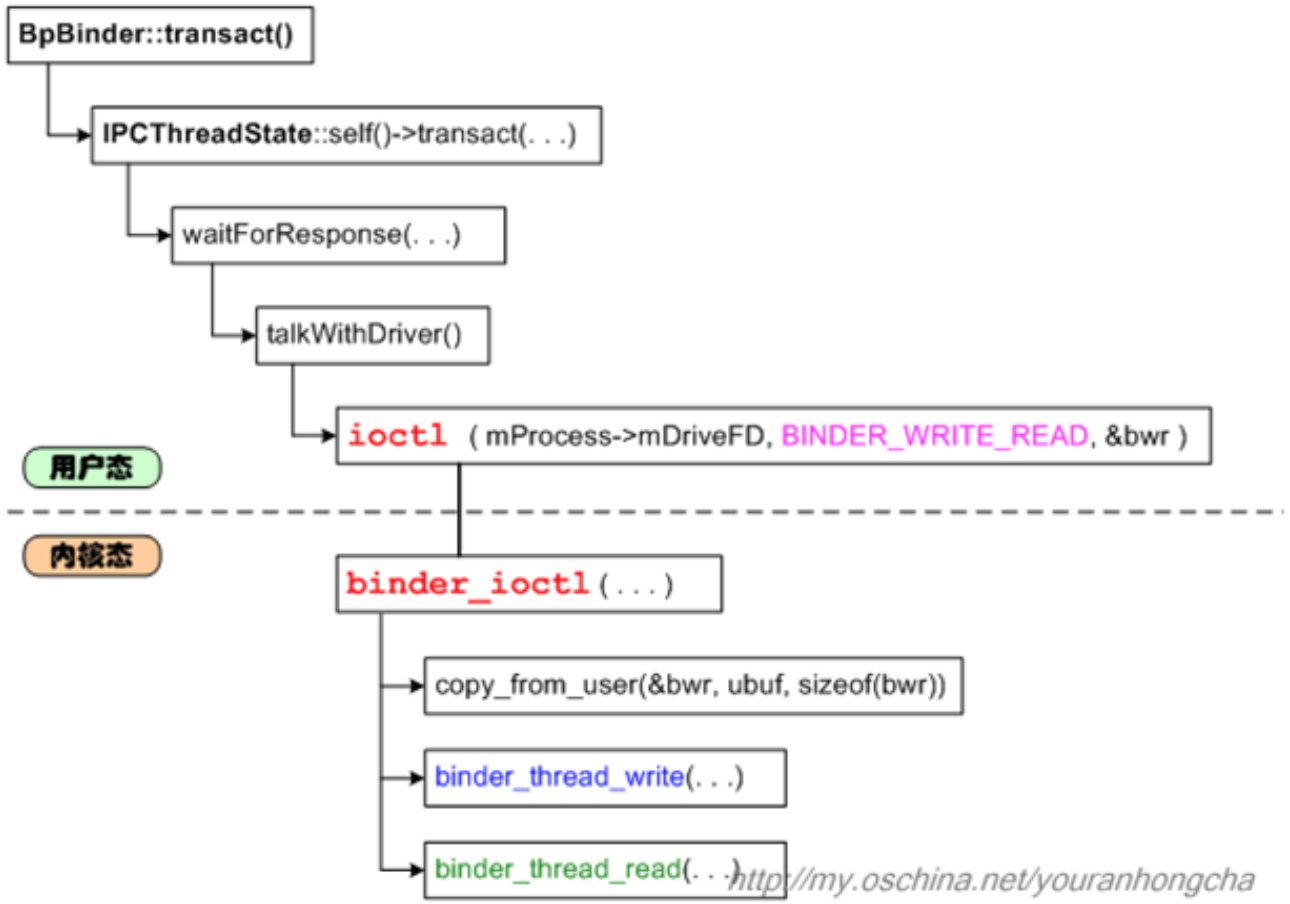
侯 亮

1 谈谈底层IPC机制吧

在上一篇文章的最后，我们说到BpBinder将数据发到了Binder驱动。然而在驱动层，这部分数据又是如何传递到BBinder一侧的呢？这里面到底藏着什么猫腻？另外，上一篇文章虽然阐述了4棵红黑树，但是并未说明红黑树的节点到底是怎么产生的。现在，我们试着回答这些问题。

1.1 概述

在Binder驱动层，和ioctl()相对的动作是binder_ioctl()函数。在这个函数里，会先调用类似copy_from_user()这样的函数，来读取用户态的数据。然后，再调用binder_thread_write()和binder_thread_read()进行进一步的处理。我们先画一张调用关系图：



binder_ioctl()调用binder_thread_write()的代码是这样的：

```
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()的前两个参数，一个是binder_proc指针，另一个是binder_thread指针，表示发起传输动作的进程和线程。binder_proc不必多说了，那个binder_thread是怎么回事？大家应该还记得前文提到的binder_proc里的4棵树吧，此处的binder_thread就是从threads树中查到的节点。

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

binder_get_thread()的代码如下：

```
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;

    // 尽量从threads树中查找和current线程匹配的binder_thread节点
    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;
    }

    // “找不到就创建”一个binder_thread节点
    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节点插入红黑树
        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()会尽量从threads树中查找和current线程匹配的binder_thread节点，如果找不到，就会创建一个新的节点并插入树中。这种"找不到就创建"的做法，在后文还会看到，我们暂时先不多说。

在调用binder_thread_write()之后，binder_ioctl()接着调用到binder_thread_read()，此时往往需要等待远端的回复，所以binder_thread_read()会让线程睡眠，把控制权让出来。在未来的某个时刻，远端处理完此处发去的语义，就会着手发回回复。当回复到达后，线程会从以前binder_thread_read()睡眠的地方醒来，并进一步解析收到的回复。

以上所说，都只是概要性的阐述，下面我们要深入一些细节了。

1.2 要进行跨进程调用，需要考虑什么?

我们可以先考虑一下，要设计跨进程调用机制，大概需要考虑什么东西呢？我们列一下：

1） 发起端：肯定包括发起端所从属的进程，以及实际执行传输动作的线程。当然，发起端的BpBinder更是重中之重。

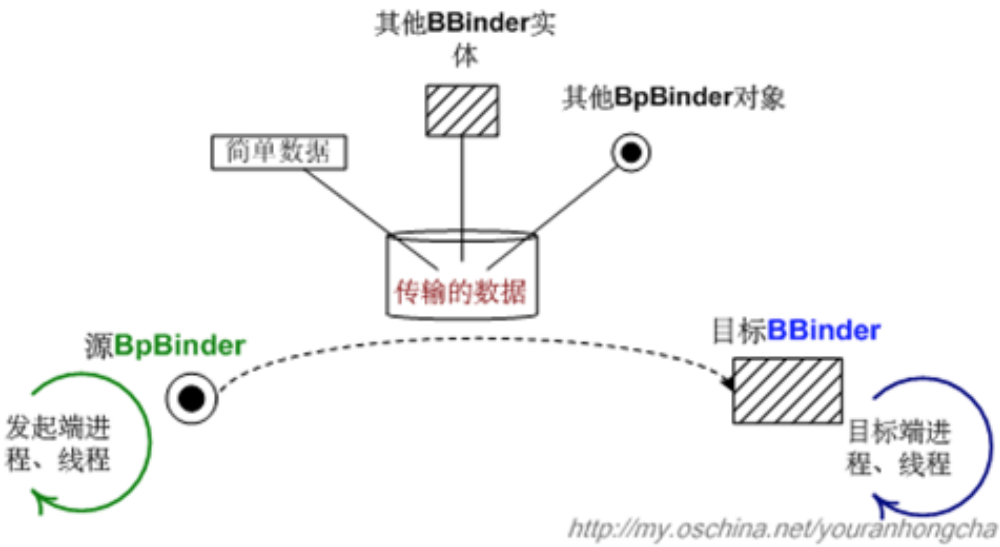
2） 接收端：包括与发起端对应的BBinder，以及目标进程、线程。

3） 待传输的数据：其实就是前文IPCThreadState::writeTransactionData()代码中的binder_transaction_data了，需要注意的是，这份数据中除了包含简单数据，还可能包含其他binder对象噢，这些对象或许对应binder代理对象，或许对应binder实体对象，视具体情况而定。

4） 如果我们的IPC动作需要接收应答（reply），该如何保证应答能准确无误地传回来?

5） 如何让系统中的多个传输动作有条不紊地进行。

我们可以先画一张示意图：

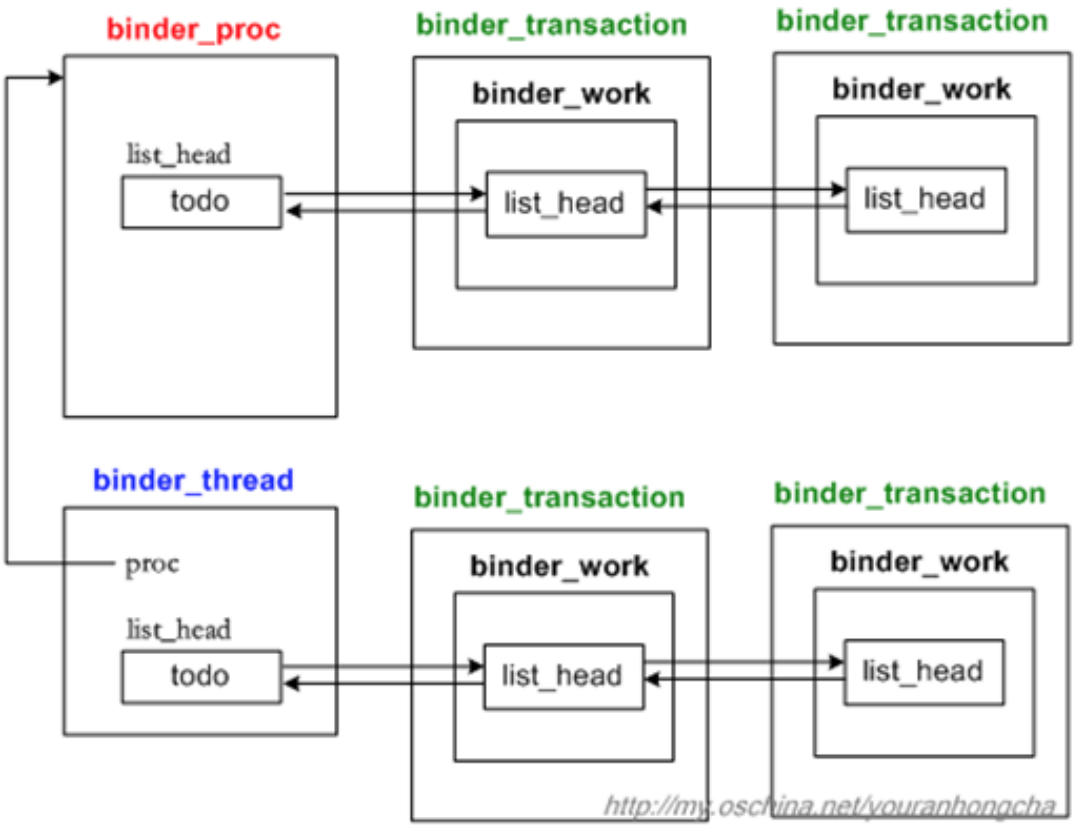


然而这张图似乎还是串接不起整个传输过程，图中的"传输的数据"到底是怎么发到目标端的呢？要回答这个问题，我们还得继续研究Binder IPC机制的实现机理。

1.3 传输机制的大体运作

Binder IPC机制的大体思路是这样的，它将每次"传输并执行特定语义的"工作理解为一个小事务，既然所传输的数据是binder_transaction_data类型的，那么这种事务的类名可以相应地定为binder_transaction。系统中当然会有很多事务啦，那么发向同一个进程或线程的若干事务就必须串行化起来，因此binder驱动为进程节点（binder_proc）和线程节点（binder_thread）都设计了个todo队列。todo队列的职责就是"串行化地组织待处理的事务"。

下图绘制了一个进程节点，以及一个从属于该进程的线程节点，它们各带了两个待处理的事务（binder_transaction）：



这样看来，传输动作的基本目标就很明确了，就是想办法把发起端的一个binder_transaction节点，插入到目标端进程或其合适子线程的todo队列去。

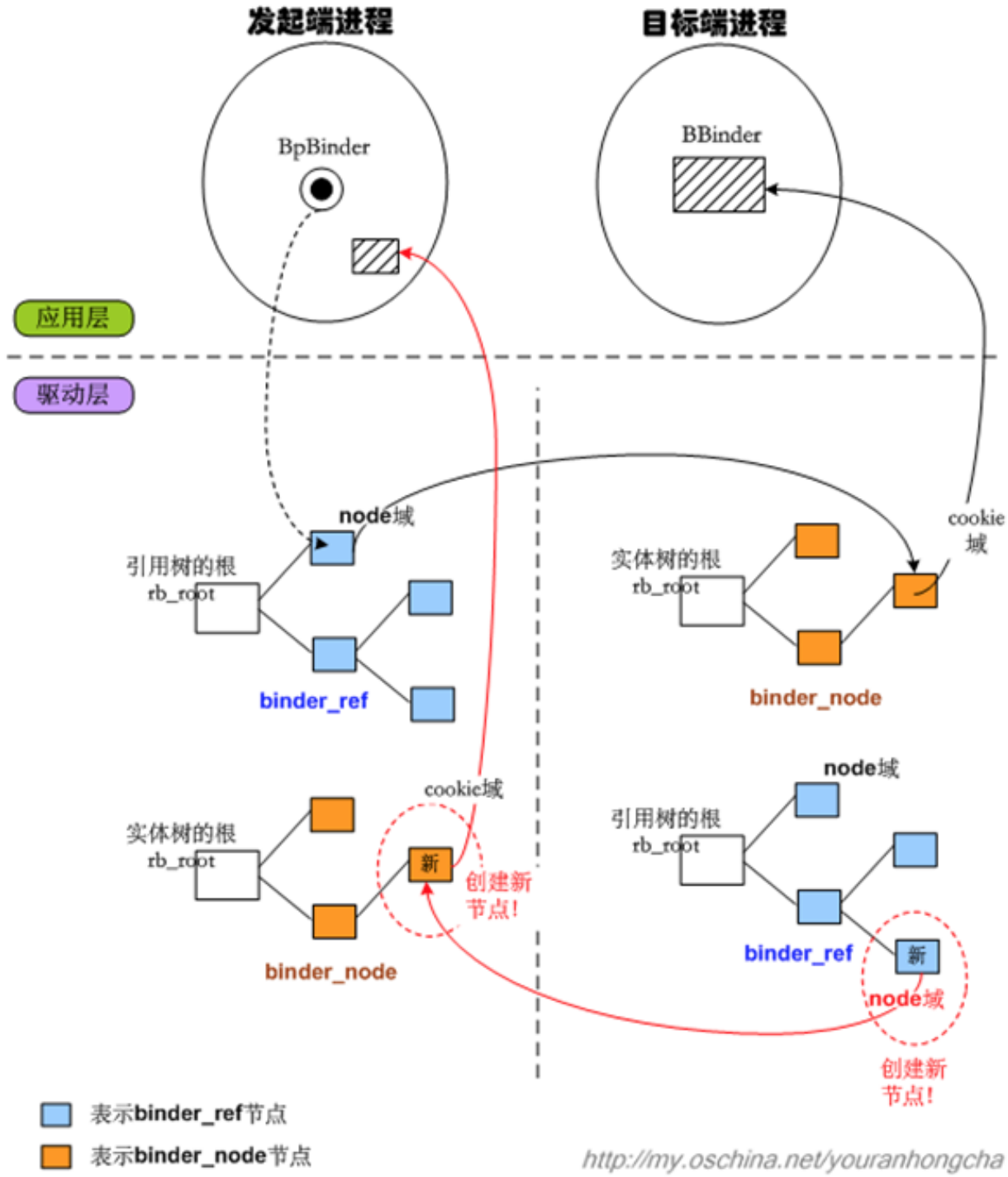
可是，该怎么找目标进程和目标线程呢？基本做法是先从发起端的BpBinder开始，找到与其对应的binder_node节点，这个在前文阐述binder_proc的4棵红黑树时已经说过了，这里不再赘述。总之拿到目标binder_node之后，我们就可以通过其proc域，拿到目标进程对应的binder_proc了。如果偷懒的话，我们直接把binder_transaction节点插到这个binder_proc的todo链表去，就算完成传输动作了。当然，binder驱动做了一些更精细的调整。

binder驱动希望能把binder_transaction节点尽量放到目标进程里的某个线程去，这样可以充分利用这个进程中的binder工作线程。比如一个binder线程目前正睡着，它在等待其他某个线程做完某个事情后才会醒来，而那个工作又偏偏需要在当前这个binder_transaction事务处理结束后才能完成，那么我们就可以让那个睡着的线程先去做当前的binder_transaction事务，这就达到充分利用线程的目的了。反正不管怎么说，如果binder驱动可以找到一个合适的线程，它就会把binder_transaction节点插到它的todo队列去。而如果找不到合适的线程，还可以把节点插入目标binder_proc的todo队列。

1.4 红黑树节点的产生过程

另一个要考虑的东西就是binder_proc里的那4棵树啦。前文在阐述binder_get_thread()时，已经看到过向threads树中添加节点的动作。那么其他3棵树的节点该如何添加呢？其实，秘密都在传输动作中。要知道，bind

er驱动在传输数据的时候，可不是仅仅简单地递送数据噢，它会分析被传输的数据，找出其中记录的binder对象，并生成相应的树节点。如果传输的是个binder实体对象，它不仅会在发起端对应的nodes树中添加一个binder_node节点，还会在目标端对应的refs_by_desc树、refs_by_node树中添加一个binder_ref节点，而且让binder_ref节点的node域指向binder_node节点。我们把前一篇文章的示意图加以修改，得到下图：



图中用红色线条来表示传输binder实体时在驱动层会添加的红黑树节点以及节点之间的关系。

可是，驱动层又是怎么知道所传的数据中有多少binder对象，以及这些对象的确切位置呢？答案很简单，是你告诉它的。大家还记得在向binder驱动传递数据之前，都是要把数据打成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);  // 把一个binder实体“打扁”并写入parcel
    status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
    return err == NO_ERROR ? reply.readExceptionCode() : err;
}
```

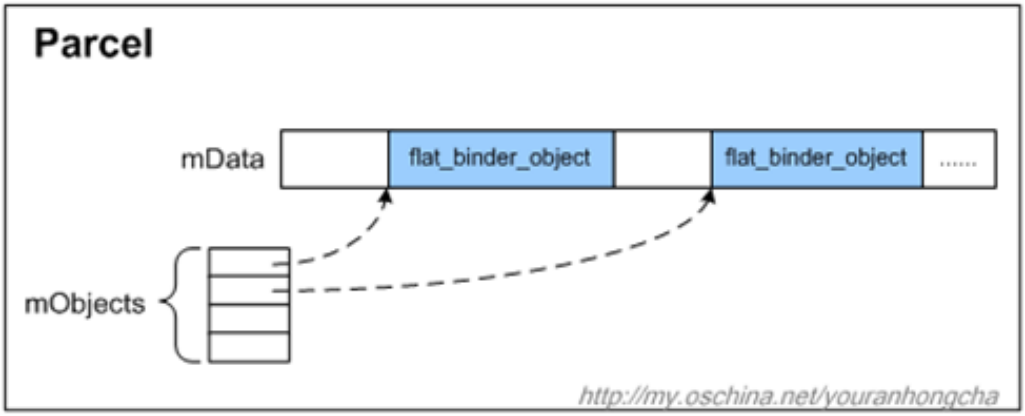
请大家注意上面data.writeStrongBinder()一句，它专门负责把一个binder实体“打扁”并写入parcel。其代码如下：

```
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);
}
```

看到了吗？“打扁”的意思就是把binder对象整理成flat_binder_object变量，如果打扁的是binder实体，那么flat_binder_object用cookie域记录binder实体的指针，即BBinder指针，而如果打扁的是binder代理，那么flat_binder_object用handle域记录的binder代理的句柄值。

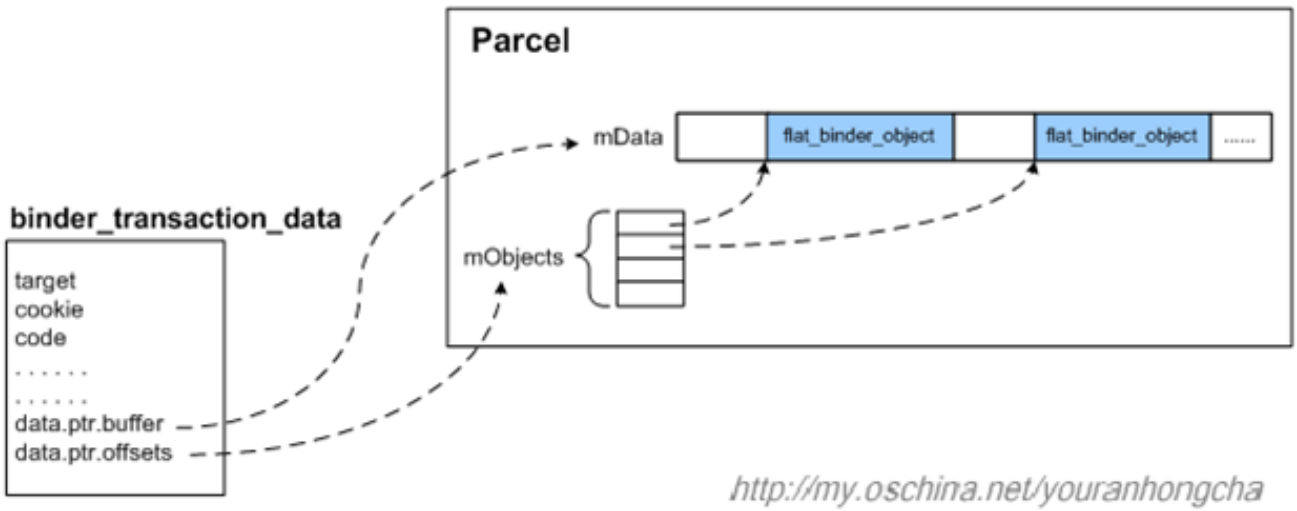
然后flatten_binder()调用了一个关键的finish_flatten_binder()函数。这个函数内部会记录下刚刚被扁平化的flat_binder_object在parcel中的位置。说得更详细点儿就是，parcel对象内部会有一个buffer，记录着parcel中所有扁平化的数据，有些扁平数据是普通数据，而另一些扁平数据则记录着binder对象。所以parcel中会构造另一个mObjects数组，专门记录那些binder扁平数据所在的位置，示意图如下：



一旦到了向驱动层传递数据的时候，IPCThreadState::writeTransactionData()会先把Parcel数据整理成一个binder_transaction_data数据，这个在上一篇文章已有阐述，但是当时我们并没有太关心里面的关键句子，现在我们把关键句子再列一下：

```
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;
    . . . . . .
        // 这部分是待传递数据
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        // 这部分是扁平化的binder对象在数据中的具体位置
        tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    . . . . . .
    mOut.write(&tr, sizeof(tr));
    . . . . . .
}
```

其中给tr.data.ptr.offsets赋值的那句，所做的就是记录下“待传数据”中所有binder对象的具体位置，示意图如下：



因此，当binder_transaction_data传递到binder驱动层后，驱动层可以准确地分析出数据中到底有多少binder对象，并分别进行处理，从而产生出合适的红黑树节点。此时，如果产生的红黑树节点是binder_node的话，binder_node的cookie域会被赋值成flat_binder_object所携带的cookie值，也就是用户态的BBinder地址值啦。这个新生成的binder_node节点被插入红黑树后，会一直严阵以待，以后当它成为另外某次传输动作的目标节点时，它

的cookie域就派上用场了，此时cookie值会被反映到用户态，于是用户态就拿到了BBinder对象。

我们再具体看一下IPCThreadState::waitForResponse()函数，当它辗转从睡眠态跳出来时，会进一步解析刚收到的命令，此时会调用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()的代码截选如下：

```
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;
}
```

请注意上面代码中的sp<BBinder> b((BBinder*)tr.cookie)一句，看到了吧，驱动层的binder_node节点的cookie值终于发挥它的作用了，我们拿到了一个合法的sp<BBinder>。

接下来，程序走到b->transact()一句。transact()函数的代码截选如下：

```
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;
    }
    . . . . . .
}
```

其中最关键的一句是调用onTransaction()。因为我们的binder实体在本质上都是继承于BBinder的，而且我们一般都会重载onTransact()函数，所以上面这句onTransact()实际上调用的是具体binder实体的onTransact()成员函数。

Ok，说了这么多，我们大概明白了binder驱动层的红黑树节点是怎么产生的，以及binder_node节点的cookie值是怎么派上用场的。限于篇幅，我们先在这里打住。下一篇文章我们再来阐述binder事务的传递和处理方面的细节。