Android IPC数据在内核空间中的发送过程分析

在上一篇文章Android请求注册服务过程源码分析中从Java层面和C++层面分析了服务请求注册的过程，无论Java还是C++最后都是将需要发送的数据写入的Parcel容器中，然后通过Binder线程持有对象IPCThreadState向Binder驱动发送，本文继续在Android请求注册服务过程源码分析的基础上更深入地介绍服务注册的整个过程。

客户进程向ServiceManager进程发送IPC服务注册信息

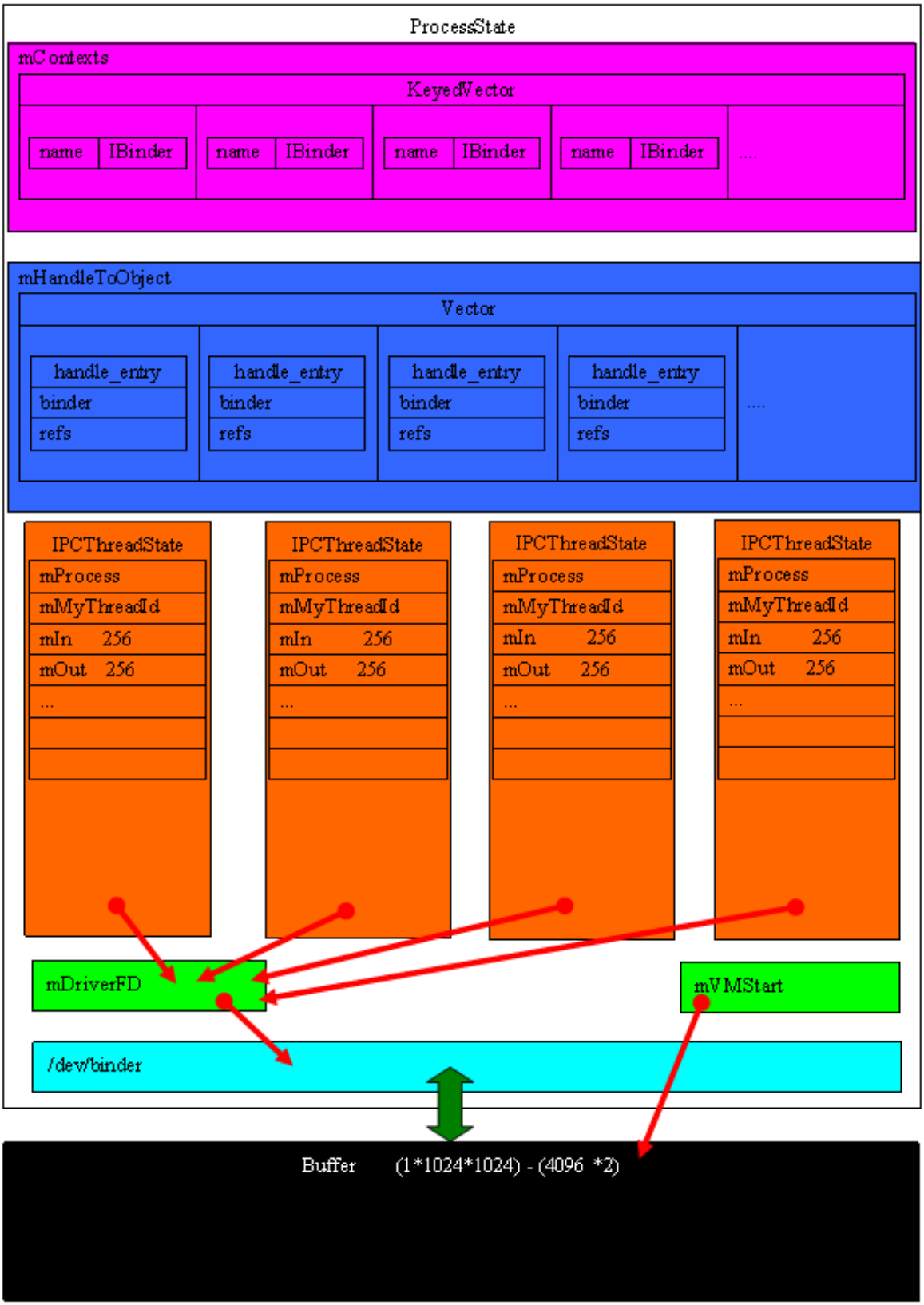
```cpp
status_t BpBinder::transact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    if (mAlive) {
        //mHandle = 0
        //code = ADD_SERVICE_TRANSACTION
        //data.writeInterfaceToken("android.os.IServiceManager");
        //data.writeString16("media.camera");
        //data.writeStrongBinder(new CameraService());
        //flags = 0
        status_t status = IPCThreadState::self()->transact(mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }
    return DEAD_OBJECT;
}
```

由于CameraService服务是向ServiceManager注册，因此将使用ServiceManager的BpBinder来传输Binder数据，由于ServiceManager对应的Handle值为0，因此在这里的mHandle = 0，BpBinder直接调用IPCThreadState来完成数据的传输：

```cpp
status_t IPCThreadState::transact(int32_t handle,uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err = data.errorCheck();
    flags |= TF_ACCEPT_FDS;
    if (err == NO_ERROR) {
        //将要发送的数据写入到IPCThreadState的成员变量mOut中，
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }
    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
```

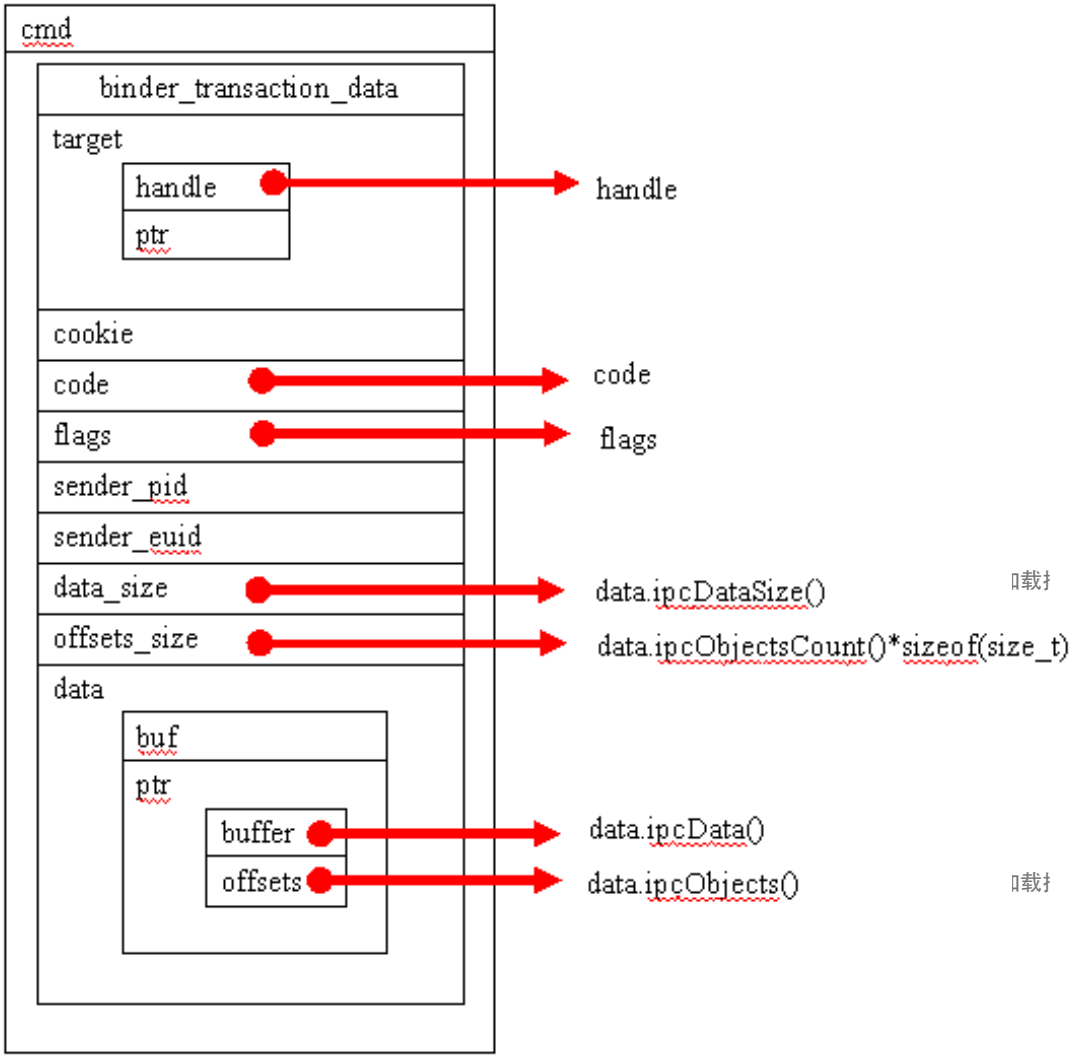
```
    }
    //同步请求
    if ((flags & TF_ONE_WAY) == 0) {
        if (reply) {
            //等待服务端的回复
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }
    return err;
}
```

该函数首先调用writeTransactionData函数将需要发送的数据写入到IPCThreadState的成员变量mOut中，然后调用waitForResponse函数等待服务端返回数据。对于每一个采用binder通信机制的进程有且只有一个ProcessState对象，该对象的成员变量mDriverFD 保存了本进程打开的binder设备文件句柄，成员变量mVMStart保存了设备文件映射到内核空间的起始地址。每个进程拥有一个binder线程池，用于接收客户端的请求，每个线程有且只有一个IPCThreadState对象，当进程需要进程之间通信时，直接通过IPCThreadState对象来访问binder驱动，IPC发送数据保存在IPCThreadState的mOut成员变量中，而接收数据则保存在mIn成员变量中。ProcessState对象与进程一一对应，IPCThreadState对象与线程一一对应，一个进程中存在一个线程池，即一个进程内可以有多个线程，因此一个进程内部可以有多个IPCThreadState对象，如下图所示:



数据打包过程

我们先来分析一下writeTransactionData函数，该函数主要负责将向服务进程发送的数据写入到Parcel对象中。在分析该函数前，先介绍一下Binder传输中使用到的binder_transaction_data数据结构：



对于CameraService服务注册，writeTransactionData函数的参数cmd=BC_TRANSACTION；binderFlags = TF_ACCEPT_FDS；handle = 0；code =ADD_SERVICE_TRANSACTION；statusBuffer = Null；发送的data为：

```cpp
data.writeInterfaceToken("android.os.IServiceManager");
data.writeString16("media.camera");
data.writeStrongBinder(new CameraService());
```

writeTransactionData函数的定义如下：

```cpp
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;
    //将发送的数据信息存放到binder_transaction_data中
    tr.target.handle = handle;
    tr.code = code;
    tr.flags = binderFlags;
    tr.cookie = 0;
    tr.sender_pid = 0;
    tr.sender_euid = 0;
```

```
12.
13.     const status_t err = data.errorCheck();
14.     if (err == NO_ERROR) {
15.         tr.data_size = data.ipcDataSize();
16.         tr.data.ptr.buffer = data.ipcData();
17.         tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
18.         tr.data.ptr.offsets = data.ipcObjects();
19.     } else if (statusBuffer) {
20.         tr.flags |= TF_STATUS_CODE;
21.         *statusBuffer = err;
22.         tr.data_size = sizeof(status_t);
23.         tr.data.ptr.buffer = statusBuffer;
24.         tr.offsets_size = 0;
25.         tr.data.ptr.offsets = NULL;
26.     } else {
27.         return (mLastError = err);
28.     }
29.     //首先写入Binder协议头
30.     mOut.writeInt32(cmd);
31.     //写入binder_transaction_data
32.     mOut.write(&tr, sizeof(tr));
33.
34.     return NO_ERROR;
35. }
```

上图清晰地展示了writeTransactionData函数的赋值过程。

数据发送过程

完成数据打包后，接下来将调用waitForResponse函数来实现数据的真正发送过程，并且等待服务进程的数据返回。

```
[cpp]
01. status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
02. {
03.     int32_t cmd;
04.     int32_t err;
05.
06.     while (1) {
07.         //==================数据发送处理=============================================
08.         //将IPC数据通过Binder驱动发送给服务进程，talkWithDriver是和Binder驱动交互的核心、阻塞型函数
09.         if ((err=talkWithDriver()) < NO_ERROR) break;
10.
11.         //==================数据接收处理=============================================
12.         //检查服务进程返回来的数据
13.         err = mIn.errorCheck();
14.         if (err < NO_ERROR) break;
15.         if (mIn.dataAvail() == 0) continue;
16.         //读取Binder通信协议命令头
17.         cmd = mIn.readInt32();
18.         //针对不同的命令头做不同的处理
19.         switch (cmd) {
20.         case BR_TRANSACTION_COMPLETE:
21.         case BR_DEAD_REPLY:
22.         case BR_FAILED_REPLY:
```

```
        case BR_ACQUIRE_RESULT:
        case BR_REPLY:
        default:
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
    }

finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
    }
    return err;
}
```

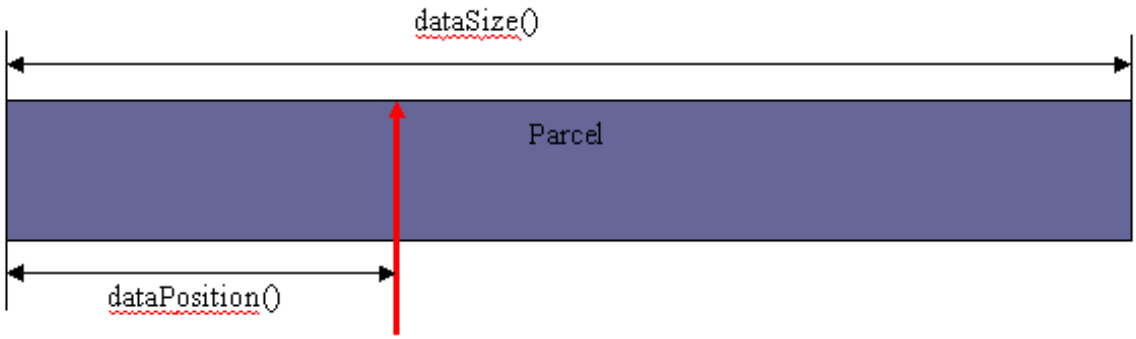
该函数首先通过talkWithDriver向Binder驱动发送数据，talkWithDriver是阻塞类型的函数，只有在发送完数据后才会返回，当talkWithDriver函数返回后，立即从mIn容器中读取服务进程回复的数据，并根据回复的数据中的Binder协议命令头来执行不同的处理，对于BR_TRANSACTION_COMPLETE、BR_DEAD_REPLY、BR_FAILED_REPLY、BR_ACQUIRE_RESULT、BR_REPLY以外的命令则调用executeCommand函数来处理。接下来我们继续分析Binder数据的发送过程，对于接收数据的处理则在接下来的小节中介绍。

[cpp]

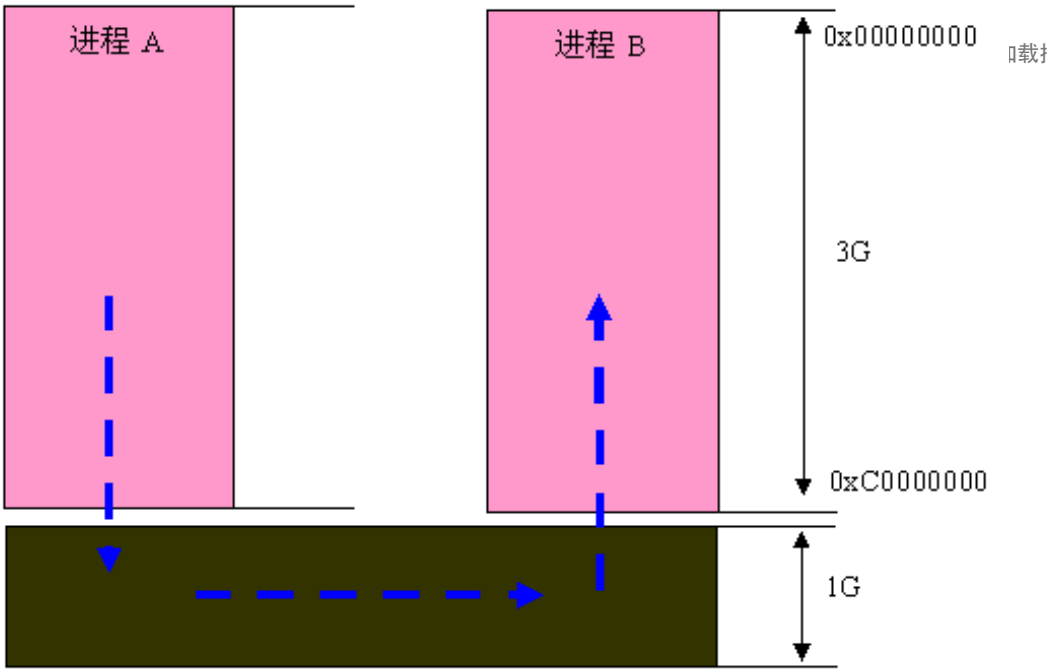
```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    //判断Binder设备驱动是否已打开
    LOG_ASSERT(mProcess->mDriverFD >= 0, "Binder driver is not opened");
    binder_write_read bwr;
    // 判断读buffer是否为空，
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();
    // We don't want to write anything if we are still reading
    // from data left in the input buffer and the caller
    // has requested to read the next data.
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;

    //将打包的Parcel写入到用户空间的binder_write_read数据结构中，binder_write_read存储了发送的数据，同时也存储了接收到的回复数据
    bwr.write_size = outAvail;
    bwr.write_buffer = (long unsigned int)mOut.data();
    // This is what we'll read.
    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (long unsigned int)mIn.data();
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }
    // 如果没有数据需要处理立即返回
    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;

    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
```

```
    status_t err;
    //通过Binder设备驱动发送数据，在数据发送完成前函数阻塞在此
    do {
        //通过ioctl系统调用切换到内核空间
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        else
            err = -errno;

    } while (err == -EINTR);
    //数据发送失败
    if (err >= NO_ERROR) {
        //如果已发送的数据个数大于0
        if (bwr.write_consumed > 0) {
            if (bwr.write_consumed < (ssize_t)mOut.dataSize())
                //从发送数据的Parcel中移除已经发送的数据
                mOut.remove(0, bwr.write_consumed);
            else
                mOut.setDataSize(0);
        }
        if (bwr.read_consumed > 0) {
            mIn.setDataSize(bwr.read_consumed);
            mIn.setDataPosition(0);
        }
        return NO_ERROR;
    }
    return err;
}
```

函数首先从mOut中将数据拷贝到binder_write_read数据结构中，关于Parcel的存储如下图所示：

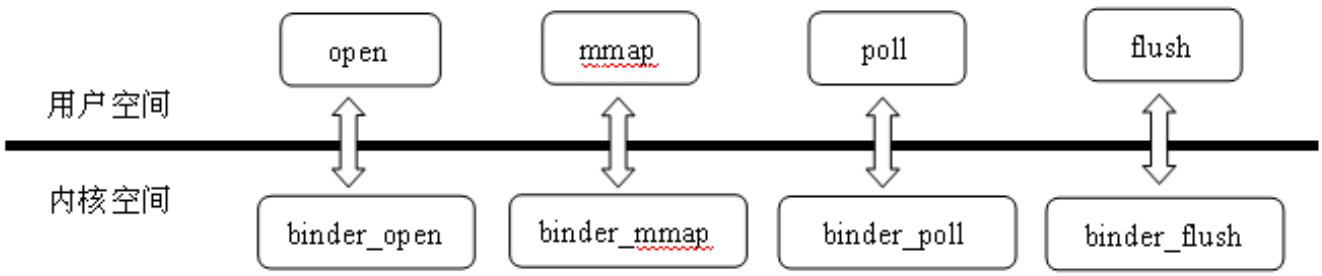


talkWithDriver函数通过ioctl系统调用进入到Binder驱动程序，为什么进程之间的IPC通信一定要通过Binder驱动呢，这里首先介绍一下关于Linux系统的内存空间分配。Android系统是基于LInux内核的操作系统，Android进程与Linux进程一样，只运行在进程固有的虚拟地址空间中。每个进程可寻址的虚拟地址空间大小为4G，其中3G是用户空间，剩余的1G是内核空间。用户代码和相关库分别运行在用户空间的代码区、数据区及堆栈区中，而内核空间中运行的代码则运行在内核空间中的各个区域中，进程具有各自独立的用户空间，所有进程共享内核空间，运行在内核空间中的代码数据是彼此共享的。

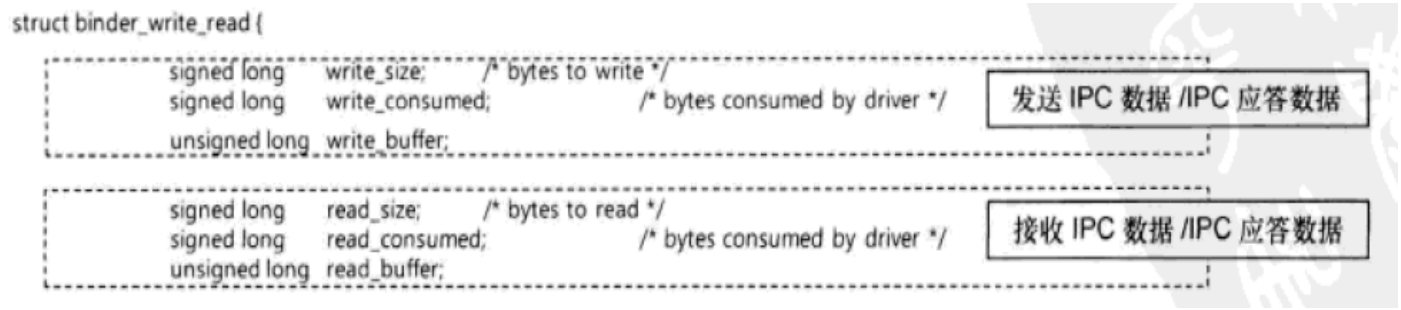


从上图可以清楚地知道，当进程A需要把IPC数据发送给进程B时，必须通过内核空间共享机制来完成，这也印证了为什么Android Binder通信过程中经过层层数据封装后，最终要通过系统调用ioctl进入到Binder驱动中的原因。学过Linux驱动设计的同学应该知道，当调用ioctl函数时，binder_ioctl函数会被调用，这两个函数自己有什么内在关联吗？这里简单介绍一下，Linux设备驱动很好地实现了模块化，在Linux内核启动的时候，会注册所有的内核驱动模块，在Android Init进程源码分析中介绍Init进程启动流程中提到过内核驱动初始化。驱动模块注册完就建立了函数之间的调用关系，在Binder驱动程序中以下代码正是建立ioctl与binder_ioctl之间的映射关系：

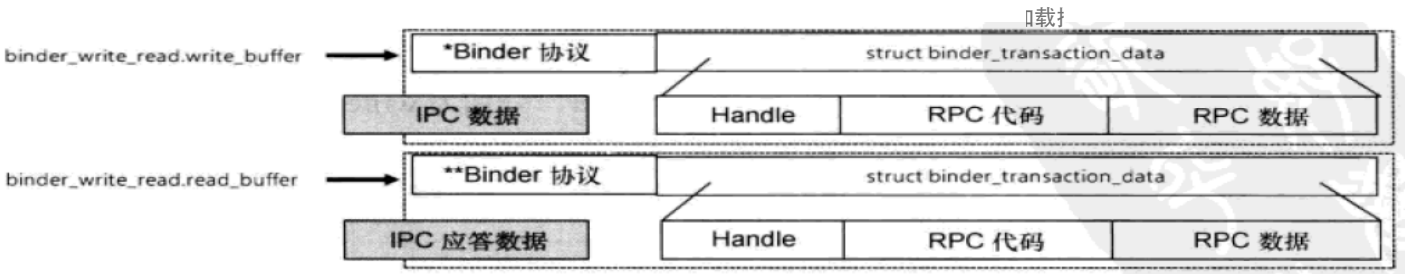
```cpp
static const struct file_operations binder_fops = {
    .owner = THIS_MODULE,
    .poll = binder_poll,
    .unlocked_ioctl = binder_ioctl,
    .mmap = binder_mmap,
    .open = binder_open,
    .flush = binder_flush,
    .release = binder_release,
};
```

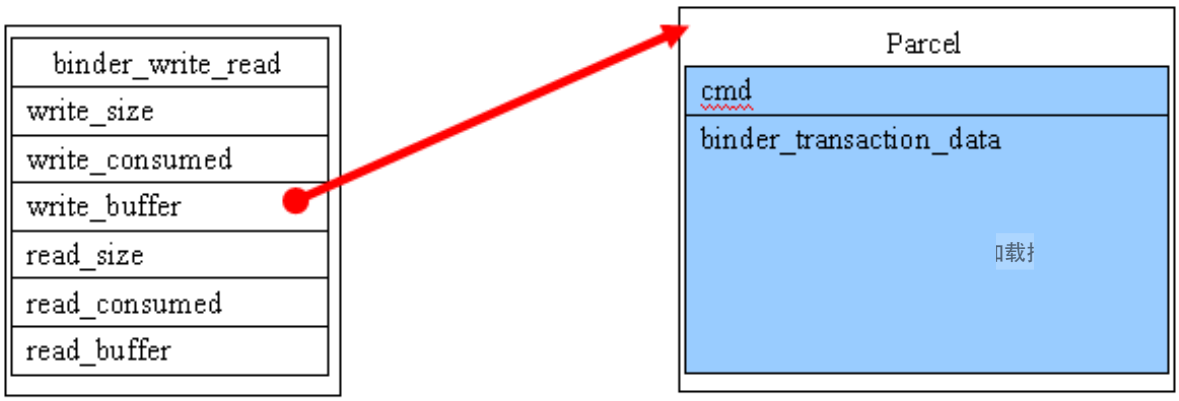


因此当用户程序调用ioctl系统调用时，Binder驱动函数binder_ioctl自动被调用。接下来详细分析该函数，从此我们就进入到了内核空间代码的分析了。ioctl 的命令参数为BINDER_WRITE_READ，数据参数为binder_write_read，该数据结构定义为：



binder_write_read中存储的数据如下：





BINDER_WRITE_READ命令用来请求Binder驱动发送或接收IPC应答数据，在使用BINDER_WRITE_READ命令调用ioctl()函数时，函数的第三个参数是binder_write_read结构体的变量，该结构体中有两个buffer，一个为IPC数据发送buffer，一个为IPC数据接收buffer;write_size与read_size分别用来指定write_buffer与read_buffer的数据大小。write_consumed与read_consumed分别用来设定write_buffer与read_buffer中被处理的数据大小，从上面分析可知，需要发送的数据为：

```cpp
[cpp]
01. bwr.write_size = outAvail;
02. bwr.write_buffer = (long unsigned int)mOut.data();
03. bwr.read_size = 0;
04. bwr.read_buffer = 0;
05. bwr.write_consumed = 0;
06. bwr.read_consumed = 0;
```

把将要发送的IPC数据存放到binder_write_read结构体中后通过ioctl系统调用传入到binder驱动程序中，binder_write_read以参数的形式传入内核空间，关于binder_ioctl函数的介绍在ServiceManager 进程启动源码分析中详细介绍了。由于read_size= 0，write_size 大于0 ，因此binder_ioctl函数对BINDER_WRITE_READ命令的处理如下：

```cpp
[cpp]
```

```
case BINDER_WRITE_READ: {
    //在内核空间中创建一个binder_write_read
    struct binder_write_read bwr;
    if (size != sizeof(struct binder_write_read)) {
        ret = -EINVAL;
        goto err;
    }
    //拷贝用户空间的binder_write_read到内核空间的binder_write_read
    if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
        ret = -EFAULT;
        goto err;
    }
    //根据write_size判断是否需要发送IPC数据
    if (bwr.write_size > 0) {
        //调用binder_thread_write来发送数据
        ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer, bwr.write_size, &
        //如果数据发送失败
        if (ret < 0) {
            bwr.read_consumed = 0;
            if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                ret = -EFAULT;
            goto err;
        }
    }
    //拷贝内核空间的binder_write_read到用户空间的binder_write_read
    if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
        ret = -EFAULT;
        goto err;
    }
    break;
}
```

binder_write_read结构体中包含着用户空间中生成的IPC数据，在Binder驱动中也存在一个相同的binder_write_read结构体，用户空间设置完binder_write_read结构体数据后，调用ioctl()函数传递给Binder驱动，Binder驱动则调用copy_from_user()函数将用户空间中的数据拷贝到Binder内核空间的binder_write_read结构体中。相反，在传递IPC应答数据时，Binder驱动调用copy_to_user()函数将内核空间的binder_write_read结构体中的数据拷贝到用户空间。

由于此时是客户进程向ServiceManager进程发送IPC注册信息，因此binder_write_read结构体的write_size大于0，调用binder_thread_write()函数进一步实现数据的跨进程发送，此时的Binder命令协议头为BC_TRANSACTION

```
[cpp]
int binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
            void __user *buffer, int size, signed long *consumed)
{
    uint32_t cmd;
    //ptr指向要处理的IPC数据，consumed表示IPC数据的个数
    void __user *ptr = buffer + *consumed;
    void __user *end = buffer + size;

```

```
    while (ptr < end && thread->return_error == BR_OK) {
        //将IPC数据的Binder系统头命令拷贝到内核空间的cmd中，此时的命令为BC_TRANSACTION
        if (get_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        //将ptr指针移动到binder_transaction_data所在地址
        ptr += sizeof(uint32_t);
        //统计该Binder命令使用率
        if (_IOC_NR(cmd) < ARRAY_SIZE(binder_stats.bc)) {
            binder_stats.bc[_IOC_NR(cmd)]++;
            proc->stats.bc[_IOC_NR(cmd)]++;
            thread->stats.bc[_IOC_NR(cmd)]++;
        }
        //根据Binder命令做相应的处理
        switch (cmd) {
        case BC_TRANSACTION:
        case BC_REPLY: {
            //在内核空间定义一个binder_transaction_data变量
            struct binder_transaction_data tr;
            //将用户空间的binder_transaction_data拷贝到内核空间，拷贝原因在于参数buffer是用户空间的指针
            if (copy_from_user(&tr, ptr, sizeof(tr)))
                return -EFAULT;
            ptr += sizeof(tr);
            //调用binder_transaction函数来执行Binder寻址、复制Binder IPC数据、生成及检索Binder节点操作
            binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
            break;
        }
        default:
            return -EINVAL;
        }
        *consumed = ptr - buffer;
    }
    return 0;
}
```

binder_transaction()函数相当复杂，主要完成Binder寻址，生成Binder节点等。由于cmd = BC_TRANSACTION，因此binder_transaction函数的第四个参数为false。

```
[cpp]

static void binder_transaction(struct binder_proc *proc,
                   struct binder_thread *thread,
                   struct binder_transaction_data *tr, int reply)
{
    //proc指向的是客户进程的binder_proc
    //thread指向的是客户进程的binder_thread
    //tr指向发送的IPC数据

    //定义数据发送事务
    struct binder_transaction *t;
    struct binder_work *tcomplete;
    size_t *offp, *off_end;

    //指向目标进程的binder_proc结构体
    struct binder_proc *target_proc;
```

```
    //指向目标线程的binder_thread结构体
    struct binder_thread *target_thread = NULL;
    //指向目标Binder实体对象的binder_node结构体
    struct binder_node *target_node = NULL;
    //定义待处理队列
    struct list_head *target_list;
    //定义一个等待队列
    wait_queue_head_t *target_wait;
    //定义应答事务
    struct binder_transaction *in_reply_to = NULL;
    //定义Binder数据传输log
    struct binder_transaction_log_entry *e;
    uint32_t return_error;

    e = binder_transaction_log_add(&binder_transaction_log);
    e->call_type = reply ? 2 : !!(tr->flags & TF_ONE_WAY);
    e->from_proc = proc->pid;
    e->from_thread = thread->pid;
    e->target_handle = tr->target.handle;
    e->data_size = tr->data_size;
    e->offsets_size = tr->offsets_size;
    //reply = false
    if (reply) {
        ...
    } else {
        //查找目标进程对应的Binder节点
        //对于服务注册来说，目标进程为servicemanger，因此它的handle值为0
        if (tr->target.handle) {
            struct binder_ref *ref;
            //通过句柄值在当前进程的binder_proc中查找Binder引用对象
            ref = binder_get_ref(proc, tr->target.handle);
            if (ref == NULL) {
                return_error = BR_FAILED_REPLY;
                goto err_invalid_target_handle;
            }
            //得到Binder引用对象所引用的目标Binder实体对象节点
            target_node = ref->node;
        //对servicemanager的处理
        } else {
            //在ServiceManger进程启动过程中被注册为binder_context_mgr_node，
            target_node = binder_context_mgr_node;
            if (target_node == NULL) {
                return_error = BR_DEAD_REPLY;
                goto err_no_context_mgr_node;
            }
        }

        e->to_node = target_node->debug_id;
        //通过目标Binder实体对象取得目标进程的binder_proc
        target_proc = target_node->proc;
        if (target_proc == NULL) {
            return_error = BR_DEAD_REPLY;
            goto err_dead_binder;
        }
        //查找过程：handle—->binder_ref—->binder_node—->binder_proc

        //如果不是同步请求，并且客户进程的线程binder_thread的事务堆栈transaction_stack不为空
```

```
73.         if (!(tr->flags & TF_ONE_WAY) && thread->transaction_stack) {
74.             //定义临时事务
75.             struct binder_transaction *tmp;
76.             //取出客户线程的事务
77.             tmp = thread->transaction_stack;
78.             //如果负责处理客户线程事务的线程不是客户线程
79.             if (tmp->to_thread != thread) {
80.                 return_error = BR_FAILED_REPLY;
81.                 goto err_bad_call_stack;
82.             }
83.             //遍历客户线程事务堆栈
84.             while (tmp) {
85.                 //
86.                 if (tmp->from && tmp->from->proc == target_proc)
87.                     target_thread = tmp->from;
88.                 tmp = tmp->from_parent;
89.             }
90.         }
91.     }
92.     //如果目标线程不为空
93.     if (target_thread) {
94.         e->to_thread = target_thread->pid;
95.         //取出目标线程的待处理队列作为当前的目标队列
96.         target_list = &target_thread->todo;
97.         //取出目标线程的等待队列作为当前的等待队列
98.         target_wait = &target_thread->wait;
99.     } else {
100.         //取出目标进程的待处理队列作为当前的目标队列
101.         target_list = &target_proc->todo;
102.         //取出目标进程的等待队列作为当前的等待队列
103.         target_wait = &target_proc->wait;
104.     }
105.     e->to_proc = target_proc->pid;
106.
107.     /* 创建数据发送事务 */
108.     t = kzalloc(sizeof(*t), GFP_KERNEL);
109.     if (t == NULL) {
110.         return_error = BR_FAILED_REPLY;
111.         goto err_alloc_t_failed;
112.     }
113.     binder_stats_created(BINDER_STAT_TRANSACTION);
114.
115.     /* 生成binder_work结构体 */
116.     tcomplete = kzalloc(sizeof(*tcomplete), GFP_KERNEL);
117.     if (tcomplete == NULL) {
118.         return_error = BR_FAILED_REPLY;
119.         goto err_alloc_tcomplete_failed;
120.     }
121.     binder_stats_created(BINDER_STAT_TRANSACTION_COMPLETE);
122.
123.     //初始化数据发送事务
124.     t->debug_id = ++binder_last_id;
125.     e->debug_id = t->debug_id;
126.     if (!reply && !(tr->flags & TF_ONE_WAY))
127.         //
128.         t->from = thread;
129.     else
```

```
130.         t->from = NULL;
131.     //设置IPC数据发送进程的euid
132.     t->sender_euid = proc->tsk->cred->euid;
133.     //设置IPC数据接收进程的binder_proc
134.     t->to_proc = target_proc;
135.     //设置IPC数据接收进程的binder_thread
136.     t->to_thread = target_thread;
137.     //设置IPC数据RPC命令码
138.     t->code = tr->code;
139.     //设置IPC数据flags标志位
140.     t->flags = tr->flags;
141.     //设置当前进程的优先级
142.     t->priority = task_nice(current);
143.     //分配一块buffer用来从接收端的IPC数据接收缓冲区复制IPC数据，接收端缓冲区空间在binder_mmap()函数中确
    定，binder_proc中的free_buffers指向该区域，binder_alloc_buf函数在free_buffers中根据RPC数据的大小分配
    binder_buffer
144.     t->buffer = binder_alloc_buf(target_proc, tr->data_size,tr->offsets_size, !reply && (t-
    >flags & TF_ONE_WAY));
145.     if (t->buffer == NULL) {
146.         return_error = BR_FAILED_REPLY;
147.         goto err_binder_alloc_buf_failed;
148.     }
149.     //初始化分配的内核缓冲区binder_buffer结构体
150.     t->buffer->allow_user_free = 0;
151.     t->buffer->debug_id = t->debug_id;
152.     t->buffer->transaction = t;
153.     t->buffer->target_node = target_node;
154.     if (target_node)
155.         binder_inc_node(target_node, 1, 0, NULL);
156.     //计算偏移数组的起始地址
157.     offp = (size_t *)(t->buffer->data + ALIGN(tr->data_size, sizeof(void *)));
158.     //从binder_transaction_data中将RPC数据拷贝到数据发送事务的内核缓冲区binder_buffer的data成员变量
    中
159.     if (copy_from_user(t->buffer->data, tr->data.ptr.buffer, tr->data_size)) {
160.         return_error = BR_FAILED_REPLY;
161.         goto err_copy_data_failed;
162.     }
163.     //从binder_transaction_data中将偏移数组中的数据拷贝到数据发送事务的内核缓冲区binder_buffer的data成
    员变量中
164.     if (copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_size)) {
165.         return_error = BR_FAILED_REPLY;
166.         goto err_copy_data_failed;
167.     }
168.     if (!IS_ALIGNED(tr->offsets_size, sizeof(size_t))) {
169.         return_error = BR_FAILED_REPLY;
170.         goto err_bad_offset;
171.     }
172.     //计算偏移数组的结束地址
173.     off_end = (void *)offp + tr->offsets_size;
174.     //循环遍历所有的Binder实体对象
175.     for (; offp < off_end; offp++) {
176.         //定义一个描述Binder实体对象的flat_binder_object
177.         struct flat_binder_object *fp;
178.         if (*offp > t->buffer->data_size - sizeof(*fp) || t->buffer-
    >data_size < sizeof(*fp) ||!IS_ALIGNED(*offp, sizeof(void *))) {
179.             return_error = BR_FAILED_REPLY;
180.             goto err_bad_offset;
```

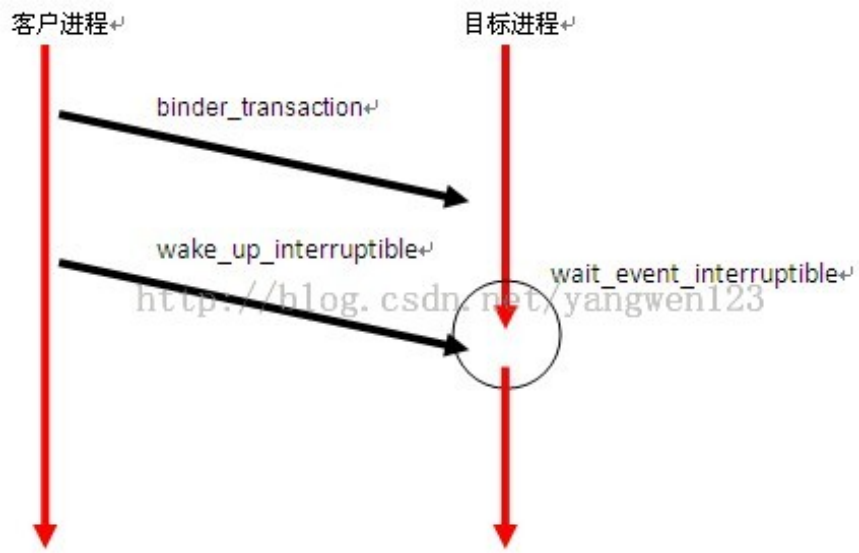
```
        }
        //计算flat_binder_object的地址，起始地址+偏移量
        fp = (struct flat_binder_object *)(t->buffer->data + *offp);
        //判断flat_binder_object对象的类型
        switch (fp->type) {
        //传输的是Binder实体对象
        case BINDER_TYPE_BINDER:
        case BINDER_TYPE_WEAK_BINDER: {
            struct binder_ref *ref;
            //从客户进程的binder_proc中查找传输的Binder实体节点
            struct binder_node *node = binder_get_node(proc, fp->binder);
            //如果不存在
            if (node == NULL) {
                //为传输的Binder对象创建一个新的Binder实体节点
                node = binder_new_node(proc, fp->binder, fp->cookie);
                if (node == NULL) {
                    return_error = BR_FAILED_REPLY;
                    goto err_binder_new_node_failed;
                }
                node->min_priority = fp->flags & FLAT_BINDER_FLAG_PRIORITY_MASK;
                node->accept_fds = !!(fp->flags & FLAT_BINDER_FLAG_ACCEPTS_FDS);
            }
            if (fp->cookie != node->cookie) {
                goto err_binder_get_ref_for_node_failed;
            }
            //从目标进程的binder_proc中查找传输的binder对象的引用对象
            ref = binder_get_ref_for_node(target_proc, node);
            if (ref == NULL) {
                return_error = BR_FAILED_REPLY;
                goto err_binder_get_ref_for_node_failed;
            }
            if (fp->type == BINDER_TYPE_BINDER)
                fp->type = BINDER_TYPE_HANDLE;
            else
                fp->type = BINDER_TYPE_WEAK_HANDLE;
            fp->handle = ref->desc;
            binder_inc_ref(ref, fp->type == BINDER_TYPE_HANDLE,&thread->todo);
        } break;
        //传输的是Binder引用对象
        case BINDER_TYPE_HANDLE:
        case BINDER_TYPE_WEAK_HANDLE: {
            struct binder_ref *ref = binder_get_ref(proc, fp->handle);
            if (ref == NULL) {
                return_error = BR_FAILED_REPLY;
                goto err_binder_get_ref_failed;
            }
            if (ref->node->proc == target_proc) {
                if (fp->type == BINDER_TYPE_HANDLE)
                    fp->type = BINDER_TYPE_BINDER;
                else
                    fp->type = BINDER_TYPE_WEAK_BINDER;
                fp->binder = ref->node->ptr;
                fp->cookie = ref->node->cookie;
                binder_inc_node(ref->node, fp->type == BINDER_TYPE_BINDER, 0, NULL);
            } else {
                struct binder_ref *new_ref;
                new_ref = binder_get_ref_for_node(target_proc, ref->node);
```

```
238.                 if (new_ref == NULL) {
239.                     return_error = BR_FAILED_REPLY;
240.                     goto err_binder_get_ref_for_node_failed;
241.                 }
242.                 fp->handle = new_ref->desc;
243.                 binder_inc_ref(new_ref, fp->type == BINDER_TYPE_HANDLE, NULL);
244.             }
245.         } break;
246.         //传输的是文件描述符
247.         case BINDER_TYPE_FD: {
248.             int target_fd;
249.             struct file *file;
250.             if (reply) {
251.                 if (!(in_reply_to->flags & TF_ACCEPT_FDS)) {
252.                     return_error = BR_FAILED_REPLY;
253.                     goto err_fd_not_allowed;
254.                 }
255.             } else if (!target_node->accept_fds) {
256.                 return_error = BR_FAILED_REPLY;
257.                 goto err_fd_not_allowed;
258.             }
259.
260.             file = fget(fp->handle);
261.             if (file == NULL) {
262.                 return_error = BR_FAILED_REPLY;
263.                 goto err_fget_failed;
264.             }
265.             target_fd = task_get_unused_fd_flags(target_proc, O_CLOEXEC);
266.             if (target_fd < 0) {
267.                 fput(file);
268.                 return_error = BR_FAILED_REPLY;
269.                 goto err_get_unused_fd_failed;
270.             }
271.             task_fd_install(target_proc, target_fd, file);
272.             fp->handle = target_fd;
273.         } break;
274.
275.         default:
276.             return_error = BR_FAILED_REPLY;
277.             goto err_bad_object_type;
278.         }
279.     }
280.     //需要应答
281.     if (reply) {
282.         BUG_ON(t->buffer->async_transaction != 0);
283.         //
284.         binder_pop_transaction(target_thread, in_reply_to);
285.     } else if (!(t->flags & TF_ONE_WAY)) {
286.         BUG_ON(t->buffer->async_transaction != 0);
287.         t->need_reply = 1;
288.         t->from_parent = thread->transaction_stack;
289.         thread->transaction_stack = t;
290.     } else {
291.         BUG_ON(target_node == NULL);
292.         BUG_ON(t->buffer->async_transaction != 1);
293.         if (target_node->has_async_transaction) {
294.             target_list = &target_node->async_todo;
```

```
295.                target_wait = NULL;
296.            } else
297.                target_node->has_async_transaction = 1;
298.        }
299.        //设置事务类型为BINDER_WORK_TRANSACTION
300.        t->work.type = BINDER_WORK_TRANSACTION;
301.        //将事务添加到目标进程或线程的待处理队列中
302.        list_add_tail(&t->work.entry, target_list);
303.
304.        //设置数据传输完成事务类型
305.        tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
306.        //将数据传输完成事务添加到客户进程的待处理队列中
307.        list_add_tail(&tcomplete->entry, &thread->todo);
308.        //唤醒目标进程或线程
309.        if (target_wait)
310.            wake_up_interruptible(target_wait);
311.        return;
312.
313.    err_get_unused_fd_failed:
314.    err_fget_failed:
315.    err_fd_not_allowed:
316.    err_binder_get_ref_for_node_failed:
317.    err_binder_get_ref_failed:
318.    err_binder_new_node_failed:
319.    err_bad_object_type:
320.    err_bad_offset:
321.    err_copy_data_failed:
322.        binder_transaction_buffer_release(target_proc, t->buffer, offp);□载
323.        t->buffer->transaction = NULL;
324.        binder_free_buf(target_proc, t->buffer);
325.    err_binder_alloc_buf_failed:
326.        kfree(tcomplete);
327.        binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
328.    err_alloc_tcomplete_failed:
329.        kfree(t);
330.        binder_stats_deleted(BINDER_STAT_TRANSACTION);
331.    err_alloc_t_failed:
332.    err_bad_call_stack:
333.    err_empty_call_stack:
334.    err_dead_binder:
335.    err_invalid_target_handle:
336.    err_no_context_mgr_node:
337.        {
338.            struct binder_transaction_log_entry *fe;
339.            fe = binder_transaction_log_add(&binder_transaction_log_failed);
340.            *fe = *e;
341.        }
342.
343.        BUG_ON(thread->return_error != BR_OK);
344.        if (in_reply_to) {
345.            thread->return_error = BR_TRANSACTION_COMPLETE;
346.            binder_send_failed_reply(in_reply_to, return_error);
347.        } else
348.            thread->return_error = return_error;
349.    }
```

函数首先根据传进来的参数binder_transaction_data来封装一个工作事务binder_transaction，关于binder_transaction结构的介绍在Android Binder通信数据结构介绍中已经详细介绍过了，然后将封装好的事务挂载到目标进程或线程的待处理队列中，最后唤醒目标进程，这样就将IPC数据发送给了目标进程，同时唤醒目标进程对IPC请求进行处理，以下两句最为重要：

```cpp
//将事务添加到目标进程或线程的待处理队列中
list_add_tail(&t->work.entry, target_list);
//唤醒目标进程或线程
if (target_wait)
    wake_up_interruptible(target_wait);
```



至此就介绍完了IPC数据在内核空间的发送过程。