Android服务查询完整过程源码分析

标签： service android systemserver servicemanager binder 2013-07-04 14:52 1263人阅读 评论(0) 收藏 举报

分类： 【Android Binder通信】（8）

Android服务注册完整过程源码分析中从上到下详细分析了Android系统的服务注册过程，本文同样针对AudioService服务来介绍Android服务的查询过程。

客户端进程数据发送过程

```java
[java]
01. private static IAudioService getService()
02. {
03.     if (sService != null) {
04.         return sService;
05.     }
06.     IBinder b = ServiceManager.getService(Context.AUDIO_SERVICE);
07.     sService = IAudioService.Stub.asInterface(b);
08.     return sService;
09. }
```

函数调用ServiceManager.getService(Context.AUDIO_SERVICE)函数向ServiceManager进程查询AudioService服务对应的binder代理对象。

```java
[java]
01. public static IBinder getService(String name) {
02.     try {
03.         IBinder service = sCache.get(name);
04.         if (service != null) {
05.             return service;
06.         } else {
07.             return getIServiceManager().getService(name);
08.         }
09.     } catch (RemoteException e) {
10.         Log.e(TAG, "error in getService", e);
11.     }
12.     return null;
13. }
```

为了加快查询速度，对每次查找的服务都存储在缓存HashMap<String, IBinder> sCache = new HashMap<String, IBinder>()中，如果是第一次查找AudioService，则缓存中不存在，于是通过getIServiceManager().getService(name)来向ServiceManager进程查询服务。getIServiceManager()函数已经在Android请求注册服务过程源码分析文中进行了详细分析，该函数返回一个ServiceManagerProxy对象，因此调用ServiceManagerProxy对象的getService函数来完成服务查询

frameworks\base\core\java\android\os\ServiceManagerNative.java

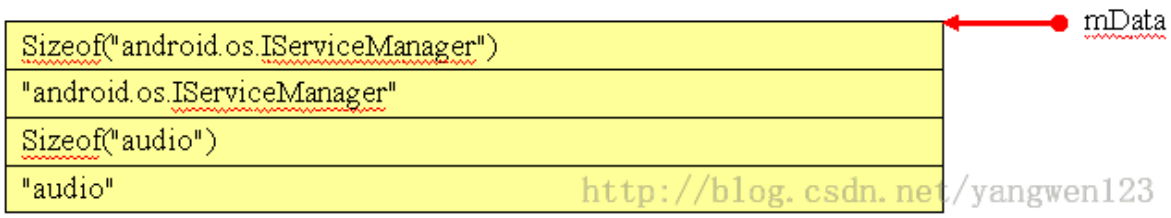
```java
[java]
01. public IBinder getService(String name) throws RemoteException {
```

```
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IServiceManager.descriptor);
    data.writeString(name);
    mRemote.transact(GET_SERVICE_TRANSACTION, data, reply, 0);
    IBinder binder = reply.readStrongBinder();
    reply.recycle();
    data.recycle();
    return binder;
}
```

发送的数据为：

```
[cpp]
data.writeInterfaceToken("android.os.IServiceManager");
data.writeString("audio");
```

此时Parcel对象中存储的数据如下：



关于数据在Binder驱动中的读写过程在Android服务注册完整过程源码分析中已经详细介绍了，这里不在分析。客户进程通过binder_thread_write函数将以上数据发送给ServiceManager进程，同时唤醒ServiceManager进程，并进入binder_thread_read函数，睡眠等待ServiceManager进程的服务查询结果。

ServiceManager服务查询过程

在binder_parse函数中，通过bio_init和bio_init_from_txn函数分别初始化了reply和msg变量，初始化值为：

```
[cpp]
reply->data = (char *) rdata + n;
reply->offs = rdata;
reply->data0 = (char *) rdata + n;
reply->offs0 = rdata;
reply->data_avail = sizeof(rdata) - n;
reply->offs_avail = 4;
reply->flags = 0;

msg->data = txn->data;
msg->offs = txn->offs;
msg->data0 = txn->data;
msg->offs0 = txn->offs;
msg->data_avail = txn->data_size;
msg->offs_avail = txn->offs_size / 4;
msg->flags = BIO_F_SHARED;
```

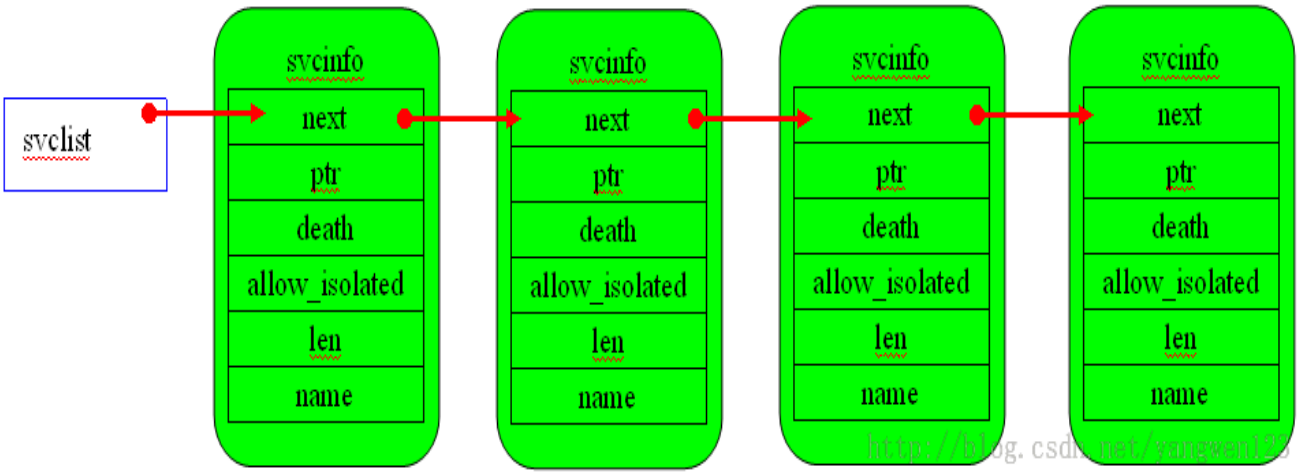
ServiceManager进程对服务的查询或注册都是通过回调函数svcmgr_handler来完成的

```
[cpp]
int svcmgr_handler(struct binder_state *bs,
                   struct binder_txn *txn,
```

```
                    struct binder_io *msg,
                    struct binder_io *reply)
{
    struct svcinfo *si;
    uint16_t *s;
    unsigned len;
    void *ptr;
    uint32_t strict_policy;
    int allow_isolated;
    if (txn->target != svcmgr_handle)
        return -1;
    //读取RPC头
    strict_policy = bio_get_uint32(msg);
    //读取字符串，在AudioService服务查询时，发送了以下数据:
    //data.writeInterfaceToken("android.os.IServiceManager");
    //data.writeString("audio");
    //这里取出来的字符串s = "android.os.IServiceManager"
    s = bio_get_string16(msg, &len);
    if ((len != (sizeof(svcmgr_id) / 2)) ||
    /* 将字符串s和数值svcmgr_id进行比较，uint16_t svcmgr_id[] = {
    'a','n','d','r','o','i','d','.','o','s','.',
    'I','S','e','r','v','i','c','e','M','a','n','a','g','e','r'
    }*/
        memcmp(svcmgr_id, s, sizeof(svcmgr_id))) {
        fprintf(stderr,"invalid id %s\n", str8(s));
        return -1;
    }
    //txn->code = GET_SERVICE_TRANSACTION
    switch(txn->code) {
    //服务查询
    case SVC_MGR_GET_SERVICE:
    case SVC_MGR_CHECK_SERVICE:
        //读取服务名称，data.writeString("audio");
        s = bio_get_string16(msg, &len);
        //查询服务
        ptr = do_find_service(bs, s, len, txn->sender_euid);
        if (!ptr)
            break;
        bio_put_ref(reply, ptr);
        return 0;
    //服务注册
    case SVC_MGR_ADD_SERVICE:
        //读取服务名称，data.writeString("audio");
        s = bio_get_string16(msg, &len);
        ptr = bio_get_ref(msg);
        allow_isolated = bio_get_uint32(msg) ? 1 : 0;
        if (do_add_service(bs, s, len, ptr, txn->sender_euid, allow_isolated))
            return -1;
        break;
    //列出服务
    case SVC_MGR_LIST_SERVICES: {
        unsigned n = bio_get_uint32(msg);
        si = svclist;
        while ((n-- > 0) && si)
            si = si->next;
        if (si) {
            bio_put_string16(reply, si->name);
            return 0;
        }
        return -1;
    }
    default:
```

```
65.         ALOGE("unknown code %d\n", txn->code);
66.         return -1;
67.     }
68.     bio_put_uint32(reply, 0);
69.     return 0;
70. }
```

客户端进程可以向ServiceManager请求查询服务、注册服务、列出服务。所有服务在ServiceManager进程中的组织方式如下图：



ServiceManager进程查询服务过程

```
[cpp]

01. case SVC_MGR_GET_SERVICE:
02. case SVC_MGR_CHECK_SERVICE:
03.     //读取服务名称，data.writeString("audio");
04.     s = bio_get_string16(msg, &len);
05.     //查询服务
06.     ptr = do_find_service(bs, s, len, txn->sender_euid);
07.     if (!ptr)
08.         break;
09.     bio_put_ref(reply, ptr);
10.     return 0;
```

首先从发送过来的data中读取服务的名称和长度，然后根据服务名称查询对应的服务。

```
[cpp]

01. void *do_find_service(struct binder_state *bs, uint16_t *s, unsigned len, unsigned uid)
02. {
03.     struct svcinfo *si;
04.     //根据传进来的服务名称在全局链表中查找对应的服务
05.     si = find_svc(s, len);
06.     if (si && si->ptr) {
07.         if (!si->allow_isolated) {
08.             // If this service doesn't allow access from isolated processes,
09.             // then check the uid to see if it is isolated.
10.             unsigned appid = uid % AID_USER;
11.             if (appid >= AID_ISOLATED_START && appid <= AID_ISOLATED_END) {
12.                 return 0;
13.             }
14.         }
15.         return si->ptr;
16.     } else {
17.         return 0;
```

```
    }
}
```

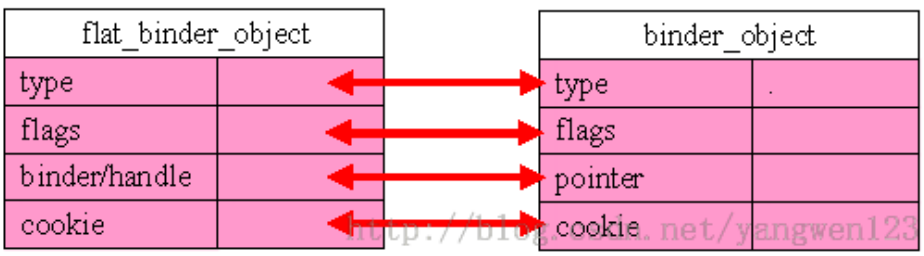
该函数调用find_svc函数从全局链表中根据服务名称查找对应的服务，然后经过权限检查，最后返回该服务在ServiceManager进程中的Binder引用对象的描述符，在服务注册中介绍过，服务首先在用户空间创建JavaBBinder本地对象，同时在内核空间创建对应的Binder节点，并且为ServiceManager进程创建Binder引用对象，将该引用对象的描述符保存在ServiceManager进程用户空间的全局链表中，就是svcinfor结构体的ptr成员变量中，服务查找过程就是根据服务名称从ServiceManager进程用户空间的服务链表中找到对应的服务，并得到该服务在ServiceManager进程中的Binder引用对象的描述符。

```cpp
struct svcinfo *find_svc(uint16_t *s16, unsigned len)
{
    struct svcinfo *si;
    //循环遍历全局链表svclist
    for (si = svclist; si; si = si->next) {
        //首先判断服务名称的长度是否相等，然后判断服务名称是否相同
        if ((len == si->len) && !memcmp(s16, si->name, len * sizeof(uint16_t))) {
            //返回指定的服务
            return si;
        }
    }
    return 0;
}
```

查找到对应的服务后，得到该服务在ServiceManager进程的Binder引用对象的描述符，然后调用函数bio_put_ref生成binder_object结构体

```cpp
void bio_put_ref(struct binder_io *bio, void *ptr)
{
    struct binder_object *obj;

    if (ptr)
        obj = bio_alloc_obj(bio);
    else
        obj = bio_alloc(bio, sizeof(*obj));

    if (!obj)
        return;

    obj->flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
    obj->type = BINDER_TYPE_HANDLE;
    obj->pointer = ptr;
    obj->cookie = 0;
}
```

flat_binder_object结构与binder_object结构之间的关系



因此bio_put_ref()函数其实就是根据查找到服务在ServiceManager进程中的Binder引用对象的描述符构造flat_binder_object结构体，并将对Binder引用对象的描述结构写入到Parcel对象reply中

执行完服务查询后，并将binder_object保存到reply变量中，然后通过binder_send_reply(bs, &reply, txn->data, res)将查询的服务发送给客户进程。

ServiceManager进程发送服务查询结果

```cpp
void binder_send_reply(struct binder_state *bs,
                       struct binder_io *reply,
                       void *buffer_to_free,
                       int status)
{
    struct {
        uint32_t cmd_free;
        void *buffer;
        uint32_t cmd_reply;
        struct binder_txn txn;
    } __attribute__((packed)) data;

    data.cmd_free = BC_FREE_BUFFER;
    data.buffer = buffer_to_free;
    data.cmd_reply = BC_REPLY;
    data.txn.target = 0;
    data.txn.cookie = 0;
    data.txn.code = 0;
    if (status) {
        data.txn.flags = TF_STATUS_CODE;
        data.txn.data_size = sizeof(int);
        data.txn.offs_size = 0;
        data.txn.data = &status;
        data.txn.offs = 0;
    } else {
        data.txn.flags = 0;
        data.txn.data_size = reply->data - reply->data0;
        data.txn.offs_size = ((char*) reply->offs) - ((char*) reply->offs0);
        data.txn.data = reply->data0;
        data.txn.offs = reply->offs0;
    }
    binder_write(bs, &data, sizeof(data));
}
```

该函数在Android服务注册完整过程源码分析中详细分析过了，向data中写入BC_FREE_BUFFER、BC_REPLY及这两个命令的数据

data	
cmd_free	BC_FREE_BUFFER
buffer	txn->data
cmd_reply	BC_REPLY

binder_txn	
target	0
cookie	0
code	0
flags	0
sender_pid	
sender_euid	
data_size	reply->data - reply->data0
offsets_size	((char*) reply->offs) - ((char*) reply->offs0)
data	reply->data0
offs	reply->offs0

BC_FREE_BUFFER命令是用来通知Binder驱动释放内核缓冲区的，而BC_REPLY是向服务查询进程返回服务查询结果的。通过binder_write(bs, &data, sizeof(data))函数向Binder驱动中写入数据data

```cpp
int binder_write(struct binder_state *bs, void *data, unsigned len)
{
    struct binder_write_read bwr;
    int res;
    bwr.write_size = len;
    bwr.write_consumed = 0;
    bwr.write_buffer = (unsigned) data;
    bwr.read_size = 0;
    bwr.read_consumed = 0;
    bwr.read_buffer = 0;
    res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
    if (res < 0) {
        fprintf(stderr,"binder_write: ioctl failed (%s)\n",
                strerror(errno));
    }
    return res;
}
```

binder_write函数将需要发送的数据data封装到binder_write_read结构体中，然后使用ioctl系统调用进入内核空间的Binder驱动中，此时的binder_write_read内容为：

bwr.write_consumed = 0;

bwr.write_buffer = (unsigned) data;

bwr.read_size = 0;

bwr.read_consumed = 0;

bwr.read_buffer = 0;

bwr.write_size = len;

由于bwr.read_size = 0，而bwr.write_size >0 因此binder_ioctl函数对BINDER_WRITE_READ命令处理中只会调用binder_thread_write函数进行Binder写操作

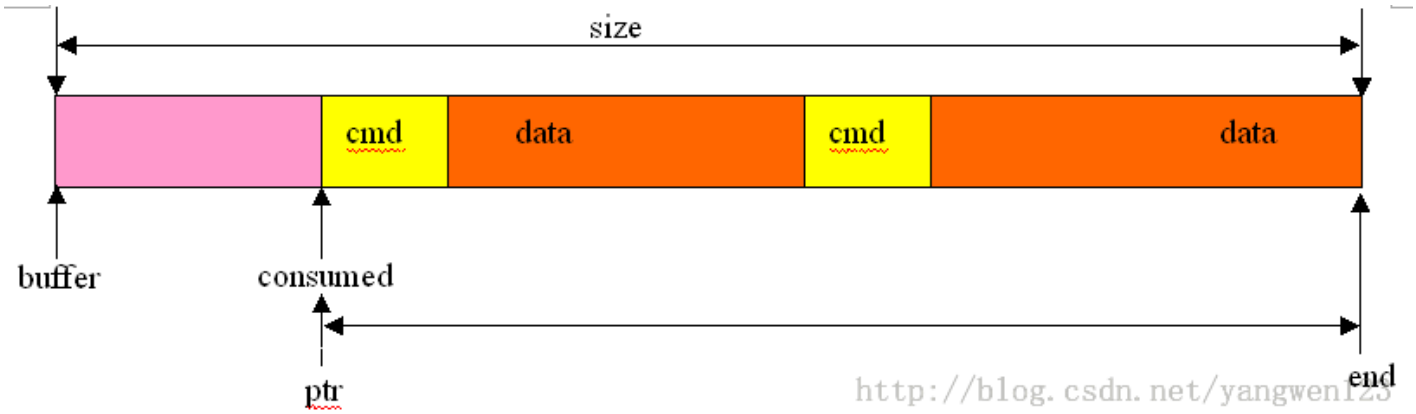
```cpp
case BINDER_WRITE_READ: {
    struct binder_write_read bwr;
    if (size != sizeof(struct binder_write_read)) {
        ret = -EINVAL;
        goto err;
    }
    if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
        ret = -EFAULT;
        goto err;
    }
    if (bwr.write_size > 0) {
        ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer, bwr.write_size, &
        if (ret < 0) {
            bwr.read_consumed = 0;
            if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                ret = -EFAULT;
            goto err;
        }
    }
    if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
        ret = -EFAULT;
        goto err;
    }
    break;
}
```

首先调用copy_from_user函数将用户空间的binder_write_read结构体内容拷贝到内核空间的binder_write_read结构体中，然后调用binder_thread_write函数完成数据发送任务，最后调用函数copy_to_user将内核空间的binder_write_read结构体内容拷贝的用户空间的binder_write_read结构体中，数据的发送结果及读取的数据都存放在binder_write_read结构体中

```cpp
int binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
            void __user *buffer, int size, signed long *consumed)
{
    uint32_t cmd;
    void __user *ptr = buffer + *consumed;
    void __user *end = buffer + size;

    while (ptr < end && thread->return_error == BR_OK) {
        if (get_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
        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)]++;
        }
        switch (cmd) {
        case BC_FREE_BUFFER: {
            void __user *data_ptr;
            struct binder_buffer *buffer;

            if (get_user(data_ptr, (void * __user *)ptr))
                return -EFAULT;
            ptr += sizeof(void *);

            buffer = binder_buffer_lookup(proc, data_ptr);
            if (buffer == NULL) {
```

```
                break;
            }
            if (!buffer->allow_user_free) {
                break;
            }
            if (buffer->transaction) {
                buffer->transaction->buffer = NULL;
                buffer->transaction = NULL;
            }
            if (buffer->async_transaction && buffer->target_node) {
                BUG_ON(!buffer->target_node->has_async_transaction);
                if (list_empty(&buffer->target_node->async_todo))
                    buffer->target_node->has_async_transaction = 0;
                else
                    list_move_tail(buffer->target_node->async_todo.next, &thread->todo);
            }
            binder_transaction_buffer_release(proc, buffer, NULL);
            binder_free_buf(proc, buffer);
            break;
        }
        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;
        }
        default:
            printk(KERN_ERR "binder: %d:%d unknown command %d\n",
                   proc->pid, thread->pid, cmd);
            return -EINVAL;
        }
        *consumed = ptr - buffer;
    }
    return 0;
}
```

首先介绍一下各个变量的含义，如下图所示：



buffer指向要传输数据的首地址，size表示传输数据的大小，consumed表示已经处理完的数据位置，因此待处理的数据就介于ptr = buffer + *consumed与end = buffer + size之间，应用程序可以向Binder驱动程序一次发送多个Binder命令，它们都是按命令+数据的方式进行存放的，binder_thread_write函数遍历ptr与end指针之间的待处理数据，依次读取Binder命令及该命令对应的数据，并根据Binder命令做相应的处理。

对于命令BC_FREE_BUFFER的处理如下：

[cpp]

```cpp
case BC_FREE_BUFFER: {
    void __user *data_ptr;
    struct binder_buffer *buffer;

    if (get_user(data_ptr, (void * __user *)ptr))
        return -EFAULT;
    ptr += sizeof(void *);

    buffer = binder_buffer_lookup(proc, data_ptr);
    if (buffer == NULL) {
        break;
    }
    if (!buffer->allow_user_free) {
        break;
    }
    if (buffer->transaction) {
        buffer->transaction->buffer = NULL;
        buffer->transaction = NULL;
    }
    if (buffer->async_transaction && buffer->target_node) {
        BUG_ON(!buffer->target_node->has_async_transaction);
        if (list_empty(&buffer->target_node->async_todo))
            buffer->target_node->has_async_transaction = 0;
        else
            list_move_tail(buffer->target_node->async_todo.next, &thread->todo);
    }
    binder_transaction_buffer_release(proc, buffer, NULL);
    binder_free_buf(proc, buffer);
    break;
}
```

首先通过get_user(data_ptr, (void * __user *)ptr)从ptr指向的数据中取出在binder_send_reply函数中设置的buffer_to_free，该指针指向要释放的内核缓冲区，然后通过binder_buffer_lookup函数从当前进程即ServiceManager进程的binder_proc中查找要释放的内核缓冲区binder_buffer，查找过程如下：

[cpp]

```cpp
static struct binder_buffer *binder_buffer_lookup(struct binder_proc *proc,
                          void __user *user_ptr)
{
    struct rb_node *n = proc->allocated_buffers.rb_node;
    struct binder_buffer *buffer;
    struct binder_buffer *kern_ptr;

    kern_ptr = user_ptr - proc->user_buffer_offset
        - offsetof(struct binder_buffer, data);

    while (n) {
        buffer = rb_entry(n, struct binder_buffer, rb_node);
        BUG_ON(buffer->free);

        if (kern_ptr < buffer)
            n = n->rb_left;
        else if (kern_ptr > buffer)
            n = n->rb_right;
        else
            return buffer;
    }
    return NULL;
}
```

user_ptr是指向传输数据的用户空间地址，通过user_ptr - proc->user_buffer_offset可以计算出传输数据在内核空间的地址，proc->user_buffer_offset在介绍内核缓冲区分配时已经接收了，该变量记录了用户空间与内核空间之间的偏移，offsetof(struct binder_buffer, data)是计算出结构体binder_buffer中的data成员变量在该结构体中的偏移量，最后通过

```cpp
kern_ptr = user_ptr - proc->user_buffer_offset - offsetof(struct binder_buffer, data)
```

可以得到保存传输数据的binder_buffer在内核空间的地址值，然后变量当前进程的binder_proc的allocated_buffers红黑树，这颗红黑树挂载了所有分配的内核缓冲区binder_buffer，在这颗红黑树中查找需要释放空间的内核缓冲区是否存在，如果存在则释放该内核缓冲区

```cpp
if (buffer->transaction) {
    buffer->transaction->buffer = NULL;
    buffer->transaction = NULL;
}
if (buffer->async_transaction && buffer->target_node) {
    BUG_ON(!buffer->target_node->has_async_transaction);
    if (list_empty(&buffer->target_node->async_todo))
        buffer->target_node->has_async_transaction = 0;
    else
        list_move_tail(buffer->target_node->async_todo.next, &thread->todo);
}
binder_transaction_buffer_release(proc, buffer, NULL);
binder_free_buf(proc, buffer);
```

接下来分析BC_REPLY的处理过程：

```cpp
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;
}
```

处理过程很简单，就是调用binder_transaction函数向服务查询请求进程发送服务查询结果，binder_transaction函数非常复杂，也是Binder驱动中数据传输的重要函数，在Android IPC数据在内核空间中的发送过程分析中有详细的分析过程，这里再次详细分析一下该函数

```cpp
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_work *tcomplete;
    size_t *offp, *off_end;
    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;
    struct binder_transaction *in_reply_to = NULL;
    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;

    if (reply) {
        in_reply_to = thread->transaction_stack;
        if (in_reply_to == NULL) {
            return_error = BR_FAILED_REPLY;
            goto err_empty_call_stack;
        }
        binder_set_nice(in_reply_to->saved_priority);
        if (in_reply_to->to_thread != thread) {
            return_error = BR_FAILED_REPLY;
            in_reply_to = NULL;
            goto err_bad_call_stack;
        }
        thread->transaction_stack = in_reply_to->to_parent;
        target_thread = in_reply_to->from;
        if (target_thread == NULL) {
            return_error = BR_DEAD_REPLY;
            goto err_dead_binder;
        }
        if (target_thread->transaction_stack != in_reply_to) {
            return_error = BR_FAILED_REPLY;
            in_reply_to = NULL;
            target_thread = NULL;
            goto err_dead_binder;
        }
        target_proc = target_thread->proc;
    } else {
        if (tr->target.handle) {
            struct binder_ref *ref;
            ref = binder_get_ref(proc, tr->target.handle);
            if (ref == NULL) {
                return_error = BR_FAILED_REPLY;
                goto err_invalid_target_handle;
            }
            target_node = ref->node;
        } else {
            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;
        target_proc = target_node->proc;
        if (target_proc == NULL) {
            return_error = BR_DEAD_REPLY;
            goto err_dead_binder;
        }
        if (!(tr->flags & TF_ONE_WAY) && thread->transaction_stack) {
            struct binder_transaction *tmp;
```

```
            tmp = thread->transaction_stack;
            if (tmp->to_thread != thread) {
                return_error = BR_FAILED_REPLY;
                goto err_bad_call_stack;
            }
            while (tmp) {
                if (tmp->from && tmp->from->proc == target_proc)
                    target_thread = tmp->from;
                tmp = tmp->from_parent;
            }
        }
    }
    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;
    }
    e->to_proc = target_proc->pid;

    /* TODO: reuse incoming transaction for reply */
    t = kzalloc(sizeof(*t), GFP_KERNEL);
    if (t == NULL) {
        return_error = BR_FAILED_REPLY;
        goto err_alloc_t_failed;
    }
    binder_stats_created(BINDER_STAT_TRANSACTION);

    tcomplete = kzalloc(sizeof(*tcomplete), GFP_KERNEL);
    if (tcomplete == NULL) {
        return_error = BR_FAILED_REPLY;
        goto err_alloc_tcomplete_failed;
    }
    binder_stats_created(BINDER_STAT_TRANSACTION_COMPLETE);

    t->debug_id = ++binder_last_id;
    e->debug_id = t->debug_id;

    if (!reply && !(tr->flags & TF_ONE_WAY))
        t->from = thread;
    else
        t->from = NULL;
    t->sender_euid = proc->tsk->cred->euid;
    t->to_proc = target_proc;
    t->to_thread = target_thread;
    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));
    if (t->buffer == NULL) {
        return_error = BR_FAILED_REPLY;
        goto err_binder_alloc_buf_failed;
    }
    t->buffer->allow_user_free = 0;
    t->buffer->debug_id = t->debug_id;
    t->buffer->transaction = t;
    t->buffer->target_node = target_node;
    if (target_node)
        binder_inc_node(target_node, 1, 0, NULL);
```

```
136.
137.     offp = (size_t *)(t->buffer->data + ALIGN(tr->data_size, sizeof(void *)));
138.
139.     if (copy_from_user(t->buffer->data, tr->data.ptr.buffer, tr->data_size)) {
140.         return_error = BR_FAILED_REPLY;
141.         goto err_copy_data_failed;
142.     }
143.     if (copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_size)) {
144.         return_error = BR_FAILED_REPLY;
145.         goto err_copy_data_failed;
146.     }
147.     if (!IS_ALIGNED(tr->offsets_size, sizeof(size_t))) {
148.         return_error = BR_FAILED_REPLY;
149.         goto err_bad_offset;
150.     }
151.     off_end = (void *)offp + tr->offsets_size;
152.     for (; offp < off_end; offp++) {
153.         struct flat_binder_object *fp;
154.         if (*offp > t->buffer->data_size - sizeof(*fp) ||
155.             t->buffer->data_size < sizeof(*fp) ||
156.             !IS_ALIGNED(*offp, sizeof(void *))) {
157.             return_error = BR_FAILED_REPLY;
158.             goto err_bad_offset;
159.         }
160.         fp = (struct flat_binder_object *)(t->buffer->data + *offp);
161.         switch (fp->type) {
162.         case BINDER_TYPE_BINDER:
163.         case BINDER_TYPE_WEAK_BINDER: {
164.             struct binder_ref *ref;
165.             struct binder_node *node = binder_get_node(proc, fp->binder);
166.             if (node == NULL) {
167.                 node = binder_new_node(proc, fp->binder, fp->cookie);
168.                 if (node == NULL) {
169.                     return_error = BR_FAILED_REPLY;
170.                     goto err_binder_new_node_failed;
171.                 }
172.                 node->min_priority = fp->flags & FLAT_BINDER_FLAG_PRIORITY_MASK;
173.                 node->accept_fds = !!(fp->flags & FLAT_BINDER_FLAG_ACCEPTS_FDS);
174.             }
175.             if (fp->cookie != node->cookie) {
176.                 goto err_binder_get_ref_for_node_failed;
177.             }
178.             ref = binder_get_ref_for_node(target_proc, node);
179.             if (ref == NULL) {
180.                 return_error = BR_FAILED_REPLY;
181.                 goto err_binder_get_ref_for_node_failed;
182.             }
183.             if (fp->type == BINDER_TYPE_BINDER)
184.                 fp->type = BINDER_TYPE_HANDLE;
185.             else
186.                 fp->type = BINDER_TYPE_WEAK_HANDLE;
187.             fp->handle = ref->desc;
188.             binder_inc_ref(ref, fp->type == BINDER_TYPE_HANDLE,
189.                            &thread->todo);
190.         } break;
191.         case BINDER_TYPE_HANDLE:
192.         case BINDER_TYPE_WEAK_HANDLE: {
193.             struct binder_ref *ref = binder_get_ref(proc, fp->handle);
194.             if (ref == NULL) {
195.                 return_error = BR_FAILED_REPLY;
196.                 goto err_binder_get_ref_failed;
197.             }
```

```
            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);
                if (new_ref == NULL) {
                    return_error = BR_FAILED_REPLY;
                    goto err_binder_get_ref_for_node_failed;
                }
                fp->handle = new_ref->desc;
                binder_inc_ref(new_ref, fp->type == BINDER_TYPE_HANDLE, NULL);
            }
        } break;

        case BINDER_TYPE_FD: {
            int target_fd;
            struct file *file;

            if (reply) {
                if (!(in_reply_to->flags & TF_ACCEPT_FDS)) {
                    return_error = BR_FAILED_REPLY;
                    goto err_fd_not_allowed;
                }
            } else if (!target_node->accept_fds) {
                return_error = BR_FAILED_REPLY;
                goto err_fd_not_allowed;
            }

            file = fget(fp->handle);
            if (file == NULL) {
                return_error = BR_FAILED_REPLY;
                goto err_fget_failed;
            }
            target_fd = task_get_unused_fd_flags(target_proc, O_CLOEXEC);
            if (target_fd < 0) {
                fput(file);
                return_error = BR_FAILED_REPLY;
                goto err_get_unused_fd_failed;
            }
            task_fd_install(target_proc, target_fd, file);
            fp->handle = target_fd;
        } break;

        default:
            return_error = BR_FAILED_REPLY;
            goto err_bad_object_type;
        }
    }
    if (reply) {
        BUG_ON(t->buffer->async_transaction != 0);
        binder_pop_transaction(target_thread, in_reply_to);
    } else if (!(t->flags & TF_ONE_WAY)) {
        BUG_ON(t->buffer->async_transaction != 0);
        t->need_reply = 1;
        t->from_parent = thread->transaction_stack;
        thread->transaction_stack = t;
```

```
    } else {
        BUG_ON(target_node == NULL);
        BUG_ON(t->buffer->async_transaction != 1);
        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;
    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;

err_get_unused_fd_failed:
err_fget_failed:
err_fd_not_allowed:
err_binder_get_ref_for_node_failed:
err_binder_get_ref_failed:
err_binder_new_node_failed:
err_bad_object_type:
err_bad_offset:
err_copy_data_failed:
    binder_transaction_buffer_release(target_proc, t->buffer, offp);
    t->buffer->transaction = NULL;
    binder_free_buf(target_proc, t->buffer);
err_binder_alloc_buf_failed:
    kfree(tcomplete);
    binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
err_alloc_tcomplete_failed:
    kfree(t);
    binder_stats_deleted(BINDER_STAT_TRANSACTION);
err_alloc_t_failed:
err_bad_call_stack:
err_empty_call_stack:
err_dead_binder:
err_invalid_target_handle:
err_no_context_mgr_node:
    {
        struct binder_transaction_log_entry *fe;
        fe = binder_transaction_log_add(&binder_transaction_log_failed);
        *fe = *e;
    }
    BUG_ON(thread->return_error != BR_OK);
    if (in_reply_to) {
        thread->return_error = BR_TRANSACTION_COMPLETE;
        binder_send_failed_reply(in_reply_to, return_error);
    } else
        thread->return_error = return_error;
}
```

参数proc是当前ServiceManager进程的binder_proc描述，而thread也是ServiceManager的Binder线程描述，参数tr是指要传输的数据，reply是一个标志位，标示当前是否是BC_REPLY命令下的数据传输，因此在这个场景下reply = 1，因此函数binder_transaction首先执行以下代码片段：

```
[cpp]
01. //从ServiceManager的Binder线程事务堆栈中取出客户进程向ServiceManager进程请求服务查询的事务项
```

```
in_reply_to = thread->transaction_stack;
if (in_reply_to == NULL) {
    return_error = BR_FAILED_REPLY;
    goto err_empty_call_stack;
}
//保存当前线程优先级到binder_transaction的saved_priority成员中
binder_set_nice(in_reply_to->saved_priority);
//判断客户进程向ServiceManager进程请求服务查询的事务项发送的目标线程是否为当前线程
if (in_reply_to->to_thread != thread) {
    return_error = BR_FAILED_REPLY;
    in_reply_to = NULL;
    goto err_bad_call_stack;
}
thread->transaction_stack = in_reply_to->to_parent;
//设置目标线程target_thread为向ServiceManager进程发送请求服务查询事务项的线程
target_thread = in_reply_to->from;
if (target_thread == NULL) {
    return_error = BR_DEAD_REPLY;
    goto err_dead_binder;
}
if (target_thread->transaction_stack != in_reply_to) {
    return_error = BR_FAILED_REPLY;
    in_reply_to = NULL;
    target_thread = NULL;
    goto err_dead_binder;
}
target_proc = target_thread->proc;

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;
}
```

这里主要是找到数据回复的目标进程及目标线程

target_thread = thread->transaction_stack->from;

target_proc = target_thread->proc;

target_list = &target_thread->todo;

target_wait = &target_thread->wait;

接着创建事务项t，用来封装整个数据的传输过程，同时创建一个binder_work项tcomplete

```
[cpp]

t = kzalloc(sizeof(*t), GFP_KERNEL);
if (t == NULL) {
    return_error = BR_FAILED_REPLY;
    goto err_alloc_t_failed;
}
binder_stats_created(BINDER_STAT_TRANSACTION);

tcomplete = kzalloc(sizeof(*tcomplete), GFP_KERNEL);
if (tcomplete == NULL) {
    return_error = BR_FAILED_REPLY;
    goto err_alloc_tcomplete_failed;
}
binder_stats_created(BINDER_STAT_TRANSACTION_COMPLETE);
```

使用参数tr来初始化事务项t

```cpp
t->debug_id = ++binder_last_id;
e->debug_id = t->debug_id;

if (!reply && !(tr->flags & TF_ONE_WAY))
    t->from = thread;
else
    t->from = NULL;
t->sender_euid = proc->tsk->cred->euid;
t->to_proc = target_proc;
t->to_thread = target_thread;
t->code = tr->code;
t->flags = tr->flags;
t->priority = task_nice(current);
//根据此次传输的数据大小分配内核缓冲区
t->buffer = binder_alloc_buf(target_proc, tr->data_size,tr->offsets_size, !reply && (t->flags & TF_ONE_WAY));
if (t->buffer == NULL) {
    return_error = BR_FAILED_REPLY;
    goto err_binder_alloc_buf_failed;
}
t->buffer->allow_user_free = 0;
t->buffer->debug_id = t->debug_id;
t->buffer->transaction = t;
t->buffer->target_node = target_node;
if (target_node)
    binder_inc_node(target_node, 1, 0, NULL);
```

这里为数据传输事务项t分配了内核缓冲区来保存传输的数据，因此需要将tr中的数据拷贝到传输事务项t的内核缓冲区中

```cpp
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)) {
    return_error = BR_FAILED_REPLY;
    goto err_copy_data_failed;
}
if (copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_size)) {
    return_error = BR_FAILED_REPLY;
    goto err_copy_data_failed;
}
if (!IS_ALIGNED(tr->offsets_size, sizeof(size_t))) {
    return_error = BR_FAILED_REPLY;
    goto err_bad_offset;
}
```

因为ServiceManager进程完成服务查询后，将查询结果封装为一个Binder描述结构flat_binder_object发送回服务查询请求进程，该结构的内容如下：

```cpp
obj->flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
obj->type = BINDER_TYPE_HANDLE;
obj->pointer = ptr;
obj->cookie = 0;
```

因此在传输的数据中存在一个flat_binder_object对象，flat_binder_object的存储方式在Android 数据Parcel序列化过程源码分

析中已经详细介绍了。binder_transaction函数通过遍历flat_binder_object对象的偏移数组来找出每一个flat_binder_object

```cpp
off_end = (void *)offp + tr->offsets_size;
for (; offp < off_end; offp++) {
    struct flat_binder_object *fp;
    if (*offp > t->buffer->data_size - sizeof(*fp) ||
        t->buffer->data_size < sizeof(*fp) ||
        !IS_ALIGNED(*offp, sizeof(void *))) {
        return_error = BR_FAILED_REPLY;
        goto err_bad_offset;
    }
    fp = (struct flat_binder_object *)(t->buffer->data + *offp);
    switch (fp->type) {
    case BINDER_TYPE_BINDER:
    case BINDER_TYPE_WEAK_BINDER:
        break;
    case BINDER_TYPE_HANDLE:
    case BINDER_TYPE_WEAK_HANDLE:
        break;
    case BINDER_TYPE_FD:
        break;
    default:
        return_error = BR_FAILED_REPLY;
        goto err_bad_object_type;
    }
}
```

由于ServiceManager进程返回

的flat_binder_object类型为BINDER_TYPE_HANDLE，binder_transaction函数对该类型的Binder对象处理如下：

```cpp
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);
        if (new_ref == NULL) {
            return_error = BR_FAILED_REPLY;
            goto err_binder_get_ref_for_node_failed;
        }
        fp->handle = new_ref->desc;
        binder_inc_ref(new_ref, fp->type == BINDER_TYPE_HANDLE, NULL);
    }
} break;
```

此时的proc代表ServiceManager进程，在Android服务注册完整过程源码分析中介绍了Binder驱动为注册的Binder本地对象在内核缓冲区创建对应的Binder节点，并且为ServiceManager进程创建引用该Binder节点的Binder引用对象，并保存到ServiceManager进程的binder_proc的红黑树中，因此这里通过binder_get_ref函数就可以从ServiceManager进程的binder_proc中根据Binder引用对象句柄值查找到对应的Binder引用对象，查找过程如下：

```cpp
static struct binder_ref *binder_get_ref(struct binder_proc *proc,
                     uint32_t desc)
{
    struct rb_node *n = proc->refs_by_desc.rb_node;
    struct binder_ref *ref;

    while (n) {
        ref = rb_entry(n, struct binder_ref, rb_node_desc);

        if (desc < ref->desc)
            n = n->rb_left;
        else if (desc > ref->desc)
            n = n->rb_right;
        else
            return ref;
    }
    return NULL;
}
```

该函数实现比较简单，就是从binder_proc的refs_by_desc这颗红黑树中根据Binder引用对象的句柄值查找出Binder引用对象。找到该Binder引用对象后，判断该Binder引用对象所引用的Binder节点所属进程是否为目标进程，Binder节点所属进程是指注册该Binder实体对应服务的进程，对于Java服务来说，这些服务对应内核Binder节点所属进程就是SystemServer进程，而这里的target_proc却是请求ServiceManager进程查询服务的进程，比如Android应用程序进程，因此ref->node->proc不等于target_proc，于是

```cpp
struct binder_ref *new_ref;
new_ref = binder_get_ref_for_node(target_proc, ref->node);
if (new_ref == NULL) {
    return_error = BR_FAILED_REPLY;
    goto err_binder_get_ref_for_node_failed;
}
fp->handle = new_ref->desc;
binder_inc_ref(new_ref, fp->type == BINDER_TYPE_HANDLE, NULL);
```

函数binder_get_ref_for_node是为当前请求服务查询进程创建Binder引用对象，同时修改传输的`flat_binder_object`结构体的`handle`句柄值，关于`Binder`引用对象的创建过程如下：

```cpp
static struct binder_ref *binder_get_ref_for_node(struct binder_proc *proc,
                          struct binder_node *node)
{
    struct rb_node *n;
    struct rb_node **p = &proc->refs_by_node.rb_node;
    struct rb_node *parent = NULL;
    struct binder_ref *ref, *new_ref;

    while (*p) {
        parent = *p;
```

```
        ref = rb_entry(parent, struct binder_ref, rb_node_node);
        if (node < ref->node)
            p = &(*p)->rb_left;
        else if (node > ref->node)
            p = &(*p)->rb_right;
        else
            return ref;
    }
    new_ref = kzalloc(sizeof(*ref), GFP_KERNEL);
    if (new_ref == NULL)
        return NULL;
    binder_stats_created(BINDER_STAT_REF);
    new_ref->debug_id = ++binder_last_id;
    new_ref->proc = proc;
    new_ref->node = node;
    rb_link_node(&new_ref->rb_node_node, parent, p);
    rb_insert_color(&new_ref->rb_node_node, &proc->refs_by_node);

    new_ref->desc = (node == binder_context_mgr_node) ? 0 : 1;
    for (n = rb_first(&proc->refs_by_desc); n != NULL; n = rb_next(n)) {
        ref = rb_entry(n, struct binder_ref, rb_node_desc);
        if (ref->desc > new_ref->desc)
            break;
        new_ref->desc = ref->desc + 1;
    }

    p = &proc->refs_by_desc.rb_node;
    while (*p) {
        parent = *p;
        ref = rb_entry(parent, struct binder_ref, rb_node_desc);
        if (new_ref->desc < ref->desc)
            p = &(*p)->rb_left;
        else if (new_ref->desc > ref->desc)
            p = &(*p)->rb_right;
        else
            BUG();
    }
    rb_link_node(&new_ref->rb_node_desc, parent, p);
    rb_insert_color(&new_ref->rb_node_desc, &proc->refs_by_desc);
    if (node) {
        hlist_add_head(&new_ref->node_entry, &node->refs);
    }
    return new_ref;
}
```

该函数首先从目标进程target_proc的refs_by_node红黑树中根据Binder节点查找对应的Binder引用对象，如果没有查找到，则为目标进程创建一个引用该Binder节点的Binder引用对象，修改该Binder引用对象的描述符，并且插入到目标进程的refs_by_node红黑树中，接着binder_transaction函数将设置发送给请求服务查询进程的flat_binder_object结构体的句柄值为新创建的Binder引用对象的描述符，flat_binder_object结构体最终会传送到请求服务查询进程的用户空间，请求服务查询进程在用户空间通过该句柄值即可从当前进程在内核空间的描述符binder_proc中查找到所查询服务的Binder引用对象，通过Binder引用对象找到Binder节点，然后通过Binder节点找到服务的Binder本地对象，这样就可以调用服务的接口函数了，真正实现RPC远程调用。

Binder本地对象的寻址过程：

handle值 -> 当前进程在内核空间的Binder引用对象 -> 服务在内核空间对应的Binder节点 -> 服务在用户空间的Binder本地对象

此时传输的flat_binder_object结构体内容如下：

```cpp
fp->flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
fp->type = BINDER_TYPE_HANDLE;
fp->handle = new_ref->desc;
fp->cookie = 0;
```

binder_transaction函数接着调用binder_pop_transaction(target_thread, in_reply_to)来释放事务项in_reply_to的内存空间，然后将创建的事物项t挂载到目标线程的待处理队列todo中，同时将传输完成事务项tcomplete挂载到当前线程的待处理队列中，并唤醒目标线程

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

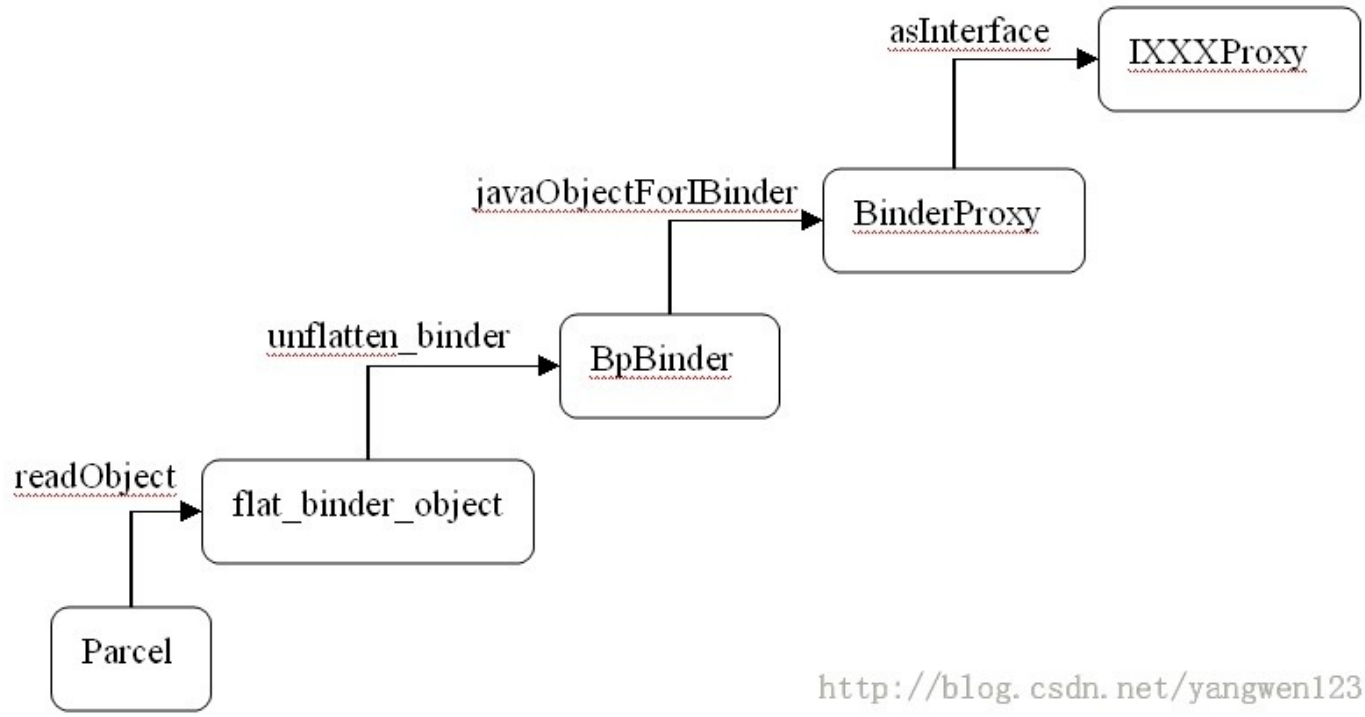
ServiceManager的当前Binder线程层层返回到binder_loop函数中，再次通过ioctl进入Binder驱动程序中，因为bwr.read_size > 0 而bwr.write_size = 0，因此binder_ioctl函数会调用binder_thread_read函数读取客户进程发送过来的请求数据，由于传输完成事务项tcomplete挂载到当前线程的待处理队列todo中，此时todo队列不为空，因此binder_thread_read函数对tcomplete事务项的处理过程为：

```cpp
case BINDER_WORK_TRANSACTION_COMPLETE: {
    cmd = BR_TRANSACTION_COMPLETE;
    if (put_user(cmd, (uint32_t __user *)ptr))
        return -EFAULT;
    ptr += sizeof(uint32_t);

    binder_stat_br(proc, thread, cmd);
    list_del(&w->entry);
    kfree(w);
    binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
} break;
```

处理完该事务项后，函数层层返回到binder_loop函数中，又再次通过ioctl进入Binder驱动程序中，同样执行binder_thread_read，但此时线程的待处理队列todo为空，因此当前线程会睡眠等待客户端的请求。前面介绍到ServiceManager进程通过binder_transaction函数将服务查询结果封装在一个事务项t中并挂载到服务查询请求进程的Binder线程待处理队列todo中，然后唤醒服务查询请求线程，由于该线程此时正睡眠在binder_thread_read函数中，线程被唤醒后，将读取到ServiceManager进程发送过来的服务查询结果，并返回到用户空间中。

客户进程获取服务代理对象

服务查询进程在用户空间通过IBinder binder = reply.readStrongBinder()来获得ServiceManager进程查询到的服务代理对象。关于readStrongBinder()函数的详细分析请看Android 数据Parcel序列化过程源码分析



函数首先从ServiceManager进程返回来的Parcel对象中取出flat_binder_object，然后根据Binder引用对象的描述符创建BpBinder对象，在创建Java层负责通信的BinderProxy对象，最后创建和业务相关的XXXProxy对象，这样就得到了所查询得到的服务的代理对象。通过该代理对象就可以向该服务的本地对象发送RPC远程调用请求。

总结一下Android服务查询过程

1）客户进程将要查询的服务名称发送给ServiceManager进程；

2）ServiceManager进程根据服务名称在自身用户空间中的全局服务链表中查找对应的服务，并得到ServiceManager进程引用该服务的句柄值；

3）ServiceManager进程将查询得到的句柄值发送给Binder驱动；

4）Binder驱动根据句柄值在ServiceManager进程的binder_proc中查找Binder引用对象；

5）如果服务查询进程不是注册该服务的进程，则在服务查询进程的binder_proc中根据Binder节点查找服务查询进程引用该Binder节点的Binder引用对象，反之，将该服务对应的Binder本地对象地址发送给服务查询进程；

6）如果服务查询进程不是注册该服务的进程并且第一次查询该服务，Binder驱动会为服务查询进程创建引用该服务Binder节点的Binder引用对象，并将该引用对象句柄值返回到客户进程的用户空间中；

7）客户进程得到本进程引用服务的Binder引用对象的句柄值后，创建服务代理对象；

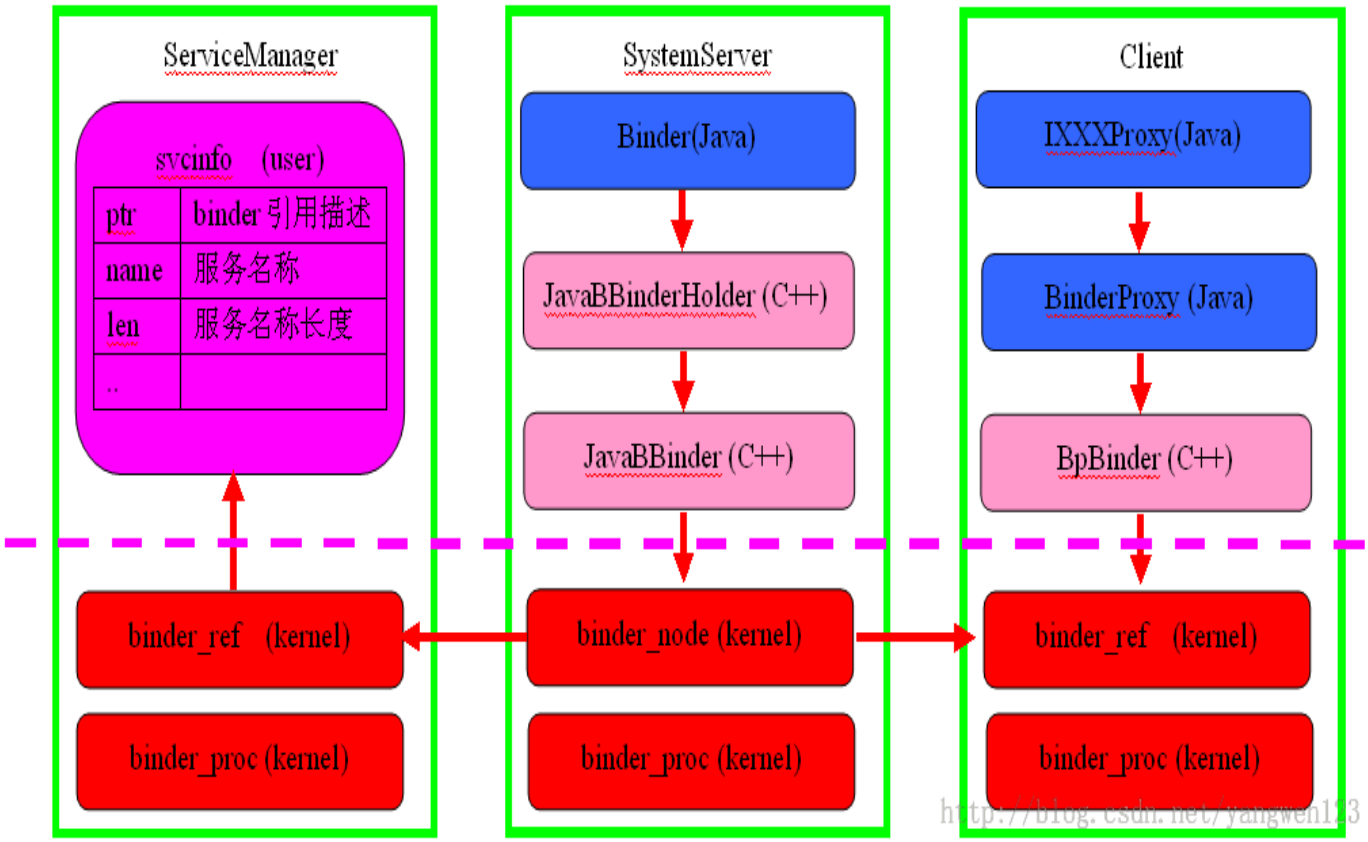
不是注册服务的进程请求查询服务：

name ——> hanlder(servicemanager) ——> binder_ref(servicemanager) ——> binder_node ——>binder_ref(client) ——>hanlder(client)

服务注册进程查询服务：

name ——> hanlder(servicemanager) ——> binder_ref(servicemanager) ——> binder_node ——> JavaBBinder

ServiceManager、服务注册进程、服务查询进程之间的关系如图：


ServiceManager
svcinfo (user)
ptr
binder引用描述
name
服务名称
len
服务名称长度
..
binder_ref (kernel)
binder_proc (kernel)
SystemServer
Binder(Java)
JavaBBinderHolder (C++)
JavaBBinder (C++)
binder_node (kernel)
binder_proc (kernel)
Client
IXXXProxy(Java)
BinderProxy (Java)
BpBinder (C++)
binder_ref (kernel)
binder_proc (kernel)