Android服务注册完整过程源码分析

标签： service server android SystemServer servicemanager

2013-07-03 20:55 2556人阅读 评论(4) 收藏 举报

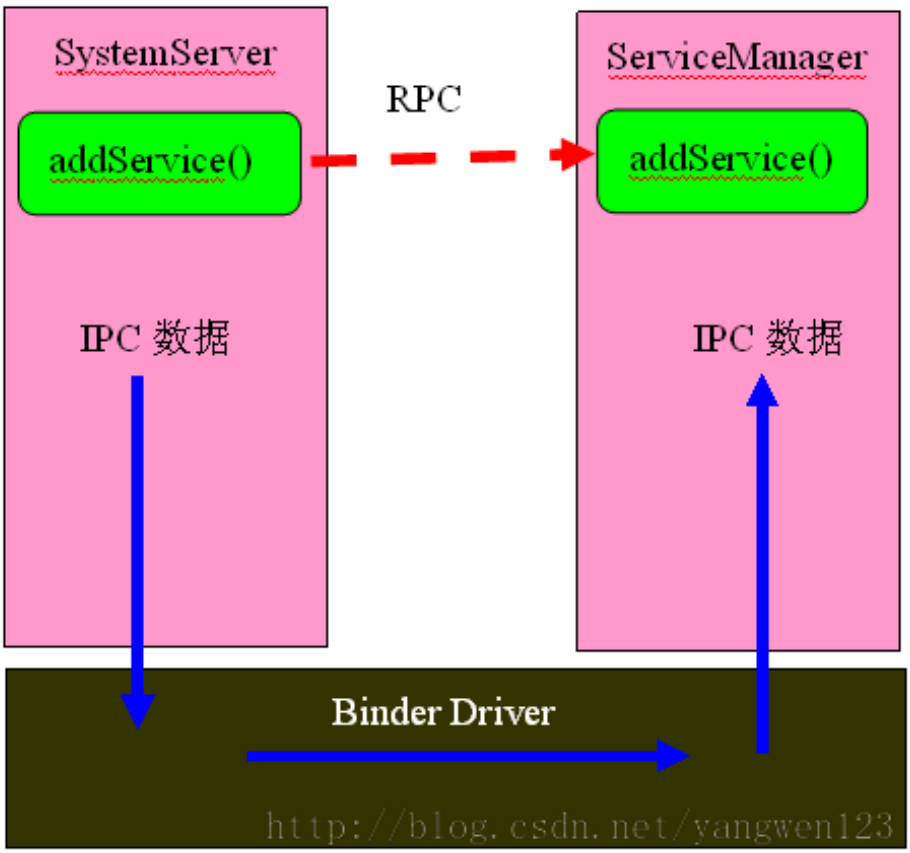
分类：

【Android Binder通信】（8）

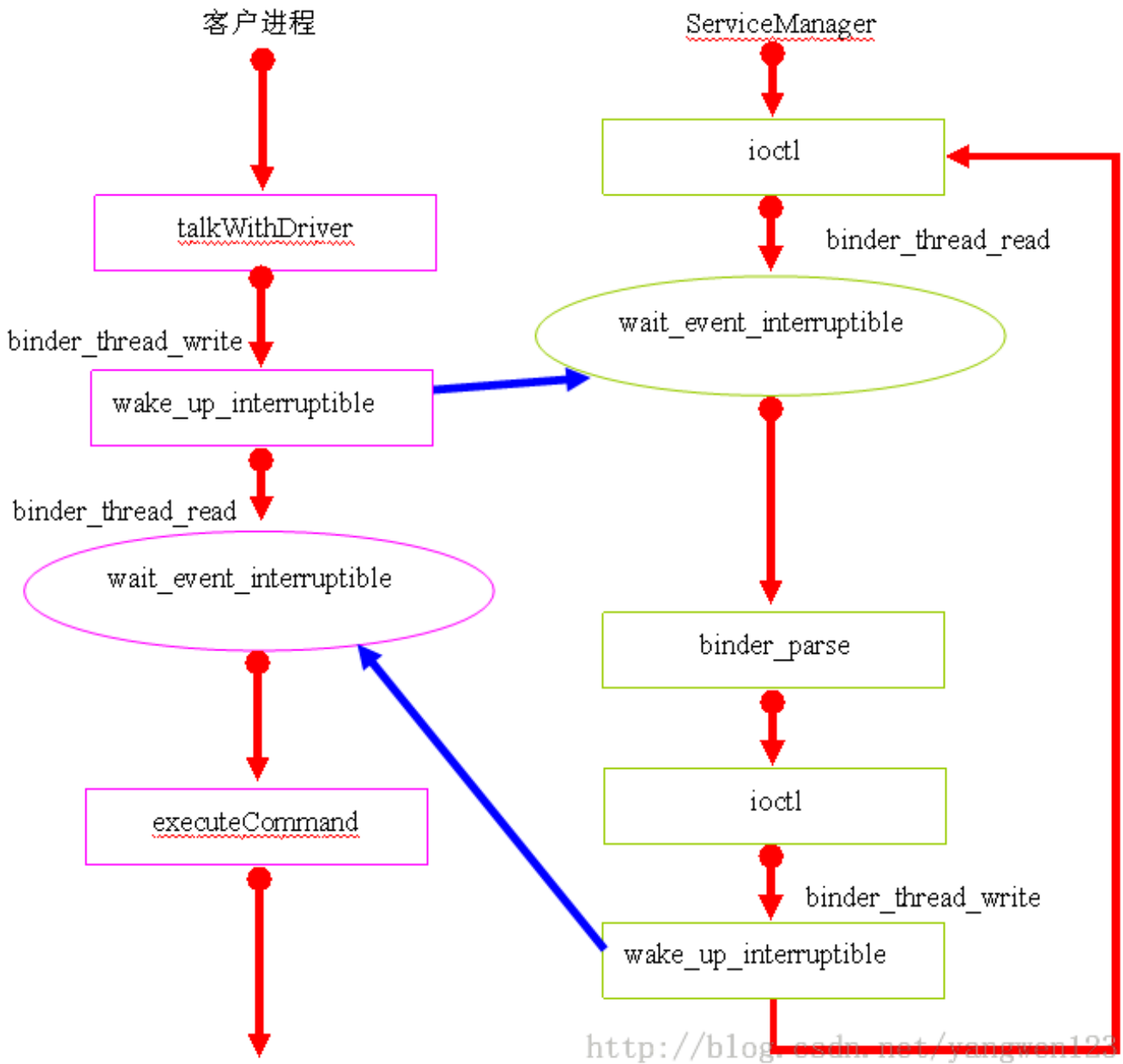
目录(?) [-]

前面从不同片段分析了Android的Binder通信机制，本文结合前面介绍的内容，对整个Android的Binder通信过程进行一次完整的分析。分析以AudioService服务的注册过程为例。

由于Android中的所有Java服务都驻留在SystemServer进程中，在SystemServer启动的时候，通过创建ServerThread线程来注册所有的Java服务，AudioService也不例外，因此AudioService的注册过程其实就是SystemServer进程与ServiceManager进程之间的一次远程RPC调用过程。



进程间通信过程的具体步骤如下图所示：



客户进程向目标进程发送服务注册信息

```cpp
if (!"0".equals(SystemProperties.get("system_init.startaudioservice"))) {
    try {
        Slog.i(TAG, "Audio Service");
        ServiceManager.addService(Context.AUDIO_SERVICE, new AudioService(context)); //AUDIO_SERVI
    } catch (Throwable e) {
        reportWtf("starting Audio Service", e);
    }
}
```

通过ServiceManager.addService(Context.AUDIO_SERVICE, new AudioService(context))向ServiceManager进程注册一个AudioService服务。

```java
public static void addService(String name, IBinder service) {
```

```
    try {
        getIServiceManager().addService(name, service, false);
    } catch (RemoteException e) {
        Log.e(TAG, "error in addService", e);
    }
}
```

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，该函数用于得到IServiceManager的远程代理ServiceManagerProxy接口对象，因此调用ServiceManagerProxy对象的addService函数来完成服务注册过程

```
[java]
public void addService(String name, IBinder service, boolean allowIsolated)
        throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IServiceManager.descriptor);
    data.writeString(name);
    data.writeStrongBinder(service);
    data.writeInt(allowIsolated ? 1 : 0);
    mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0);
    reply.recycle();
    data.recycle();
}
```

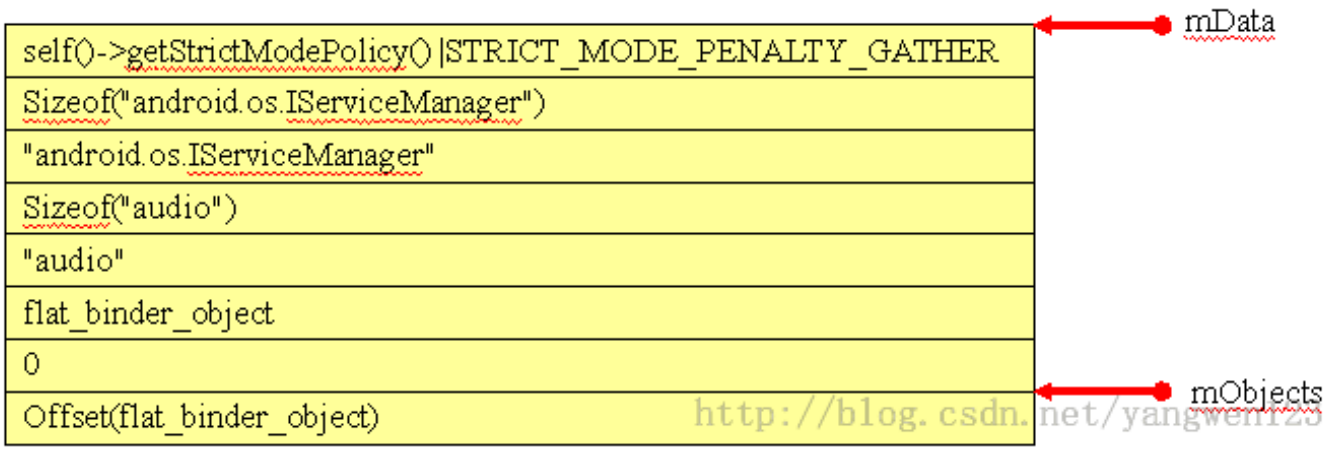
发送的数据为：

data.writeInterfaceToken("android.os.IServiceManager");

data.writeString("audio");

data.writeStrongBinder(new AudioService(context));

data.writeInt(0);



关于Parcel数据序列化问题请阅读Android 数据Parcel序列化过程源码分析 AudioService是一个Binder对象，其flat_binder_object结构体描述如下：

```
[cpp]
obj.flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
obj.type = BINDER_TYPE_BINDER;
obj.binder = local->getWeakRefs();
```

```
04.  obj.cookie = local;
```

由于数据传输的目标进程时ServiceManager进程，在Android系统中，ServiceManager的引用句柄值规定为0，因此ServiceManager进程的通信Binder代理对象为new BpBinder(0)，其对应的Java层的Binder代理对象mRemote = new BinderProxy(new BpBinder(0)),于是这里调用BinderProxy的transact()函数来传输数据,Android请求注册服务过程源码分析中通过图说明了BinderProxy与BpBinder之间的关系，BinderProxy通过其成员变量mObject来保存其对应的BpBinder对象地址。BinderProxy对象的transact函数的定义如下：

```
[cpp]
01.  public native boolean transact(int code, Parcel data, Parcel reply,int flags) throws RemoteExcepti
```

这是一个本地函数，其对应的JNI函数的实现为：

frameworks\base\core\jni\android_util_Binder.cpp

```
[cpp]
01.  static jboolean android_os_BinderProxy_transact(JNIEnv* env, jobject obj,
02.          jint code, jobject dataObj, jobject replyObj, jint flags) // throws RemoteException
03.  {
04.      if (dataObj == NULL) {
05.          jniThrowNullPointerException(env, NULL);
06.          return JNI_FALSE;
07.      }
08.
09.      Parcel* data = parcelForJavaObject(env, dataObj);
10.      if (data == NULL) {
11.          return JNI_FALSE;
12.      }
13.      Parcel* reply = parcelForJavaObject(env, replyObj);
14.      if (reply == NULL && replyObj != NULL) {
15.          return JNI_FALSE;
16.      }
17.
18.      IBinder* target = (IBinder*)env->GetIntField(obj, gBinderProxyOffsets.mObject);
19.      if (target == NULL) {
20.          jniThrowException(env, "java/lang/IllegalStateException", "Binder has been finalized!");
21.          return JNI_FALSE;
22.      }
23.
24.      ALOGV("Java code calling transact on %p in Java object %p with code %d\n",
25.              target, obj, code);
26.
27.      // Only log the binder call duration for things on the Java-level main thread.
28.      // But if we don't
29.      const bool time_binder_calls = should_time_binder_calls();
30.
31.      int64_t start_millis;
32.      if (time_binder_calls) {
33.          start_millis = uptimeMillis();
34.      }
35.      //printf("Transact from Java code to %p sending: ", target); data->print();
```

```
    status_t err = target->transact(code, *data, reply, flags);
    //if (reply) printf("Transact from Java code to %p received: ", target); reply->print();
    if (time_binder_calls) {
        conditionally_log_binder_call(start_millis, target, code);
    }

    if (err == NO_ERROR) {
        return JNI_TRUE;
    } else if (err == UNKNOWN_TRANSACTION) {
        return JNI_FALSE;
    }

    signalExceptionForError(env, obj, err, true /*canThrowRemoteException*/);
    return JNI_FALSE;
}
```

该函数在Android请求注册服务过程源码分析中已经详细分析过来，首先通过BinderProxy的成员变量mObject取得C++层的BpBinder对象new BpBinder(0) 的地址，并使用该对象来真正传输数据

[cpp]

```
status_t BpBinder::transact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

code = ADD_SERVICE_TRANSACTION

mHandle = 0

flags = 0

该函数最终调用IPCThreadState的transact来完成数据传输

[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) {
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
```

```
    }

    if ((flags & TF_ONE_WAY) == 0) {
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }

    return err;
}
```

函数通过writeTransactionData()函数将上面的数据写入到IPCThreadState的成员变量mOut中，writeTransactionData(BC_TRANSACTION, TF_ACCEPT_FDS, 0, GET_SERVICE_TRANSACTION, data, NULL)

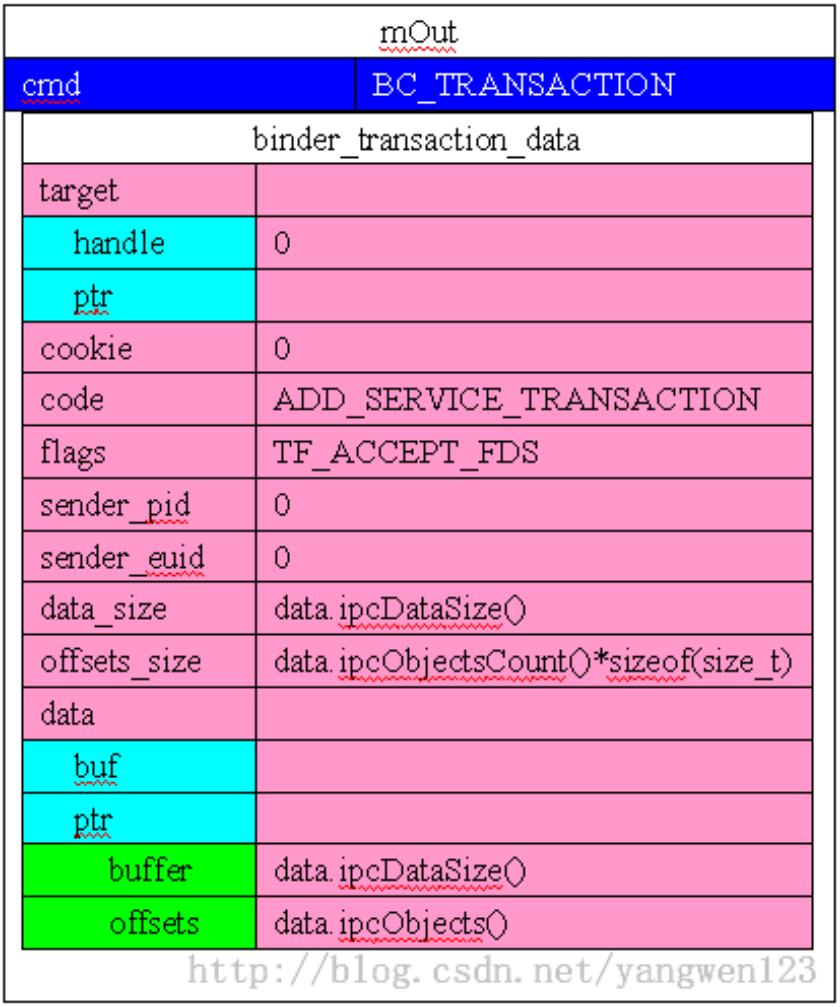
```
[cpp]
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
    int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
{
    binder_transaction_data tr;

    tr.target.handle = handle;
    tr.code = code;
    tr.flags = binderFlags;
    tr.cookie = 0;
    tr.sender_pid = 0;
    tr.sender_euid = 0;

    const status_t err = data.errorCheck();
    if (err == NO_ERROR) {
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    } else if (statusBuffer) {
        tr.flags |= TF_STATUS_CODE;
        *statusBuffer = err;
        tr.data_size = sizeof(status_t);
        tr.data.ptr.buffer = statusBuffer;
        tr.offsets_size = 0;
        tr.data.ptr.offsets = NULL;
    } else {
        return (mLastError = err);
    }

    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));

    return NO_ERROR;
}
```



然后调用函数waitForResponse将mOut中的数据发送到Binder驱动中，并等待服务进程返回执行结果

[cpp]

```cpp
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    int32_t cmd;
    int32_t err;
    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;

        cmd = mIn.readInt32();

        switch (cmd) {
        case BR_TRANSACTION_COMPLETE:
        case BR_DEAD_REPLY:
        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;
}
```

使用函数talkWithDriver()与Binder驱动交互

```
[cpp]

status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    ALOG_ASSERT(mProcess->mDriverFD >= 0, "Binder driver is not opened");
    binder_write_read bwr;
                                                                                载
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;

    bwr.write_size = outAvail;
    bwr.write_buffer = (long unsigned int)mOut.data();

    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;
    }
    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;

    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    do {
#if defined(HAVE_ANDROID_OS)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        else
            err = -errno;
#else
        err = INVALID_OPERATION;
#endif
    } while (err == -EINTR);

    if (err >= NO_ERROR) {
        if (bwr.write_consumed > 0) {
            if (bwr.write_consumed < (ssize_t)mOut.dataSize())
                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;
}
```

将数据发送与接收容器Parce封装在binder_write_read结构体中，最后通过ioctl系统调用进入Binder驱动，此时的数据为：

cmd = BINDER_WRITE_READ

bwr.write_size = outAvail;

bwr.write_buffer = (long unsigned int)mOut.data();

bwr.read_size = mIn.dataCapacity();

bwr.read_buffer = (long unsigned int)mIn.data();

bwr.read_consumed = 0;

因为write_size大于0，因此在此次的ioctl函数的BINDER_WRITE_READ命令下只执行Binder数据写操作

```
[cpp]
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int ret;
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread;
    unsigned int size = _IOC_SIZE(cmd);
    void __user *ubuf = (void __user *)arg;

    /*printk(KERN_INFO "binder_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);*/

    ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
    if (ret)
        return ret;

    mutex_lock(&binder_lock);
    thread = binder_get_thread(proc);
    if (thread == NULL) {
        ret = -ENOMEM;
        goto err;
    }

    switch (cmd) {
    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;
```

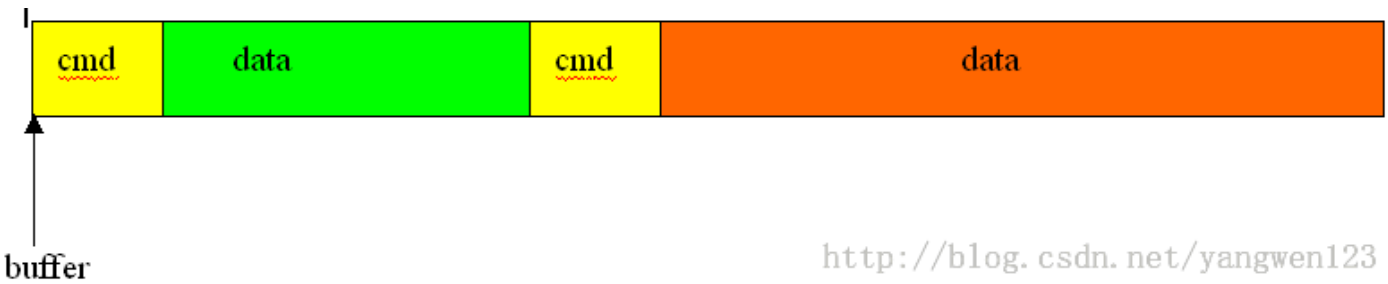
```
        }
        if (bwr.write_size > 0) {
            ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer, bwr.write_siz
            if (ret < 0) {
                bwr.read_consumed = 0;
                if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                    ret = -EFAULT;
                goto err;
            }
        }
        if (bwr.read_size > 0) {
            ret = binder_thread_read(proc, thread, (void __user *)bwr.read_buffer, bwr.read_size,
>f_flags & O_NONBLOCK);
            if (!list_empty(&proc->todo))
                wake_up_interruptible(&proc->wait);
            if (ret < 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;
    }
    default:
        ret = -EINVAL;
        goto err;
    }
    ret = 0;
err:
    if (thread)
        thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
    mutex_unlock(&binder_lock);
    wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
    if (ret && ret != -ERESTARTSYS)
        printk(KERN_INFO "binder: %d:%d ioctl %x %lx returned %d\n", proc->pid, current-
>pid, cmd, arg, ret);
    return ret;
}
```

binder_ioctl函数在Android IPC数据在内核空间中的发送过程分析中已经详细介绍了，根据传进来的参数可知，这里只执行binder_thread_write数据写操作。

```
[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;
        //变量用户空间的buffer，取出所有的Binder命令及对应的数据并处理
```

```
    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_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;
}
```

在数据发送Parcel对象中可以发送多个Binder命令



此次发送到Binder驱动的命令只有一个，因此在遍历buffer时，只能取出cmd = BINDER_WRITE_READ，该命令下发送的数据为：

binder_transaction_data	
target	
handle	0
ptr	
cookie	0
code	ADD_SERVICE_TRANSACTION
flags	TF_ACCEPT_FDS
sender_pid	0
sender_euid	0
data_size	data.ipcDataSize()
offsets_size	data.ipcObjectsCount()*sizeof(size_t)
data	
buf	
ptr	
buffer	data.ipcDataSize()
offsets	data.ipcObjects()

binder_thread_write函数在Android IPC数据在内核空间中的发送过程分析中也详细介绍了，由于Binder命令为BC_TRANSACTION，因此会调用binder_transaction函数来传输Binder实体对象到目标进程。binder_transaction函数首先将参数binder_transaction_data来封装一个工作事务binder_transaction

```cpp
struct binder_transaction *t;
//创建一个新的事务项
t = kzalloc(sizeof(*t), GFP_KERNEL);
//创建一个完成事务项
tcomplete = kzalloc(sizeof(*tcomplete), GFP_KERNEL);
binder_stats_created(BINDER_STAT_TRANSACTION_COMPLETE);

t->debug_id = ++binder_last_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;
offp = (size_t *)(t->buffer->data + ALIGN(tr->data_size, sizeof(void *)));
if (copy_from_user(t->buffer->data, tr->data.ptr.buffer, tr->data_size)) {

}
if (copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_size)) {

}
```

接下来遍历Parcel对象中的所有flat_binder_object结构体，因为这里传输了一个AudioService Binder对象，其对应的flat_binder_object如下：

```
[cpp]
obj.flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
obj.type = BINDER_TYPE_BINDER;
obj.binder = local->getWeakRefs();
obj.cookie = local;
```

这是一个Binder实体对象，在传输过程中对Binder实体对象的处理过程如下：

```
[cpp]
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) {
    //如果此次传输的是Binder实体对象，即服务注册
    case BINDER_TYPE_BINDER:
    case BINDER_TYPE_WEAK_BINDER: {
        struct binder_ref *ref;
        //通过BBinder的mRefs在当前binder_proc中查找该Binder实体对应的Binder节点
        struct binder_node *node = binder_get_node(proc, fp->binder);
        //第一次传输该Binder实体对象时，Binder驱动中不存在对应的Binder节点
        if (node == NULL) {
            //为传输的Binder实体对象创建对应的Binder节点,从此该Binder对象在Binder驱动程序中就存在对应的Binder节点了
            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;
        }
        //为目标进程也就是ServiceManager进程创建一个该Binder节点的Binder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;
        }
        //修改传输的flat_binder_object对象的类型
        if (fp->type == BINDER_TYPE_BINDER)
            fp->type = BINDER_TYPE_HANDLE;
        else
            fp->type = BINDER_TYPE_WEAK_HANDLE;
        //设置传输的flat_binder_object的handle为Binder引用的描述符
        fp->handle = ref->desc;
        binder_inc_ref(ref, fp->type == BINDER_TYPE_HANDLE,&thread->todo);
    } break;
    //如果此次传输的是Binder引用对象，即服务查询
    default:
        return_error = BR_FAILED_REPLY;
        goto err_bad_object_type;
    }
}
```

函数首先从当前进程的binder_proc中查找该Binder对象在内核中的Binder节点

```
[cpp]
struct binder_node *node = binder_get_node(proc, fp->binder);
```

参数proc为当前进程即SystemServer进程的binder_proc，参数fp->binder为传输的Binder实体对象内部的弱引用对象地址，binder_get_node函数就是从SystemServer进程中根据Binder实体对象内部的弱引用对象地址查找该Binder实体对象在内核空间中的Binder节点，其实现如下：

```
[cpp]
static struct binder_node *binder_get_node(struct binder_proc *proc,
                        void __user *ptr)
{
    struct rb_node *n = proc->nodes.rb_node;
    struct binder_node *node;

    while (n) {
        node = rb_entry(n, struct binder_node, rb_node);

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

该函数实现比较简单，就是从binder_proc的nodes红黑树中查找指定的binder_node节点，因为是第一次传输AudioService对象，因此在内核空间中不存在该对象对应的Binder节点，于是调用函数binder_new_node在内核空间中为该Binder实体对象创建对应的Binder节点，在后续传输该Binder实体对象时就可以查找到了

```cpp
static struct binder_node *binder_new_node(struct binder_proc *proc,
                       void __user *ptr,
                       void __user *cookie)
{
    struct rb_node **p = &proc->nodes.rb_node;
    struct rb_node *parent = NULL;
    struct binder_node *node;

    while (*p) {
        parent = *p;
        node = rb_entry(parent, struct binder_node, rb_node);

        if (ptr < node->ptr)
            p = &(*p)->rb_left;
        else if (ptr > node->ptr)
            p = &(*p)->rb_right;
        else
            return NULL;
    }
    //创建一个binder_node节点
    node = kzalloc(sizeof(*node), GFP_KERNEL);
    if (node == NULL)
        return NULL;
    binder_stats_created(BINDER_STAT_NODE);
    //将该binder_node节点挂载到binder_proc中
    rb_link_node(&node->rb_node, parent, p);
    rb_insert_color(&node->rb_node, &proc->nodes);
    //初始化binder_node节点
    node->debug_id = ++binder_last_id;
    node->proc = proc;
    node->ptr = ptr;//保存Binder实体对象内部的弱引用对象地址
    node->cookie = cookie;//保存Binder实体对象的地址
    node->work.type = BINDER_WORK_NODE;
    INIT_LIST_HEAD(&node->work.entry);
    INIT_LIST_HEAD(&node->async_todo);
    return node;
}
```

函数为AudioService这个服务Binder实体对象在内核空间中创建了对应的Binder节点，并且将该Binder实体对象的用户空间地址保存到了其对应的Binder节点中，这样就可以通过内核空间的Binder节点找到对应的用户空间的Binder实体对象，同时将创建的该Binder节点挂载到SystemServer进程的binder_proc中。

然后为ServiceManager进程创建该Binder节点的Binder引用对象

```cpp
ref = binder_get_ref_for_node(target_proc, node);
```

target_proc此时是ServiceManager进程的binder_proc，参数node是前面为注册的AudioService这个Binder实体对象创建的内核空间的Binder节点。调用binder_get_ref_for_node函数为ServiceManager进程创建一个AudioService对应的Binder节点的Binder引用对象

```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;
     //从proc中查找是否已经存在相同的Binder引用对象了
    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;
    }
    //如果binder_proc中无法查找到指定的Binder引用对象，这为该进程创建一个Binder引用对象
    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;
    //保存该Binder引用对象所引用的Binder节点，这样就可以通过Binder引用对象找到对应的Binder节点
    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);
    //判断Binder节点是否是ServiceManager的Binder节点，如果是，则设置该Binder引用对象的描述符为0，否则设置为1
    new_ref->desc = (node == binder_context_mgr_node) ? 0 : 1;
    //重新调整该Binder引用对象的描述符，确保binder_proc中的所有Binder引用对象的描述符是唯一的
    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);
```

```
    return new_ref;
}
```

函数首先从binder_proc的红黑树中查找是否存在引用node这个Binder节点的Binder引用对象，如果没有，则创建一个引用node这个Binder节点的Binder引用对象，同时设置该Binder引用对象的描述符，同一个进程中的每个Binder引用对象都有唯一的描述符。在Android系统中，服务的注册过程其实就是为用户空间的Binder实体对象在内核空间中创建对应的Binder节点，并且在ServiceManager进程的binder_proc中创建引用该Binder节点的引用对象，同时将该引用对象的描述符保存到ServiceManager进程的用户空间的链表中。

```[cpp]
if (fp->type == BINDER_TYPE_BINDER)
    fp->type = BINDER_TYPE_HANDLE;
else
    fp->type = BINDER_TYPE_WEAK_HANDLE;
fp->handle = ref->desc;
```

由于此时注册的是AudioService Binder实体对象，因此fp的值被修改为：

```[cpp]
fp->flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
fp->type = BINDER_TYPE_HANDLE;
fp->cookie = local;
fp->handle = ref->desc;
```

然后将事务t挂载到目标进程的待处理队列，将完成事务tcomplete挂载到当前Binder线程的待处理队列中

```[cpp]
//设置t事务的binder_work类型
t->work.type = BINDER_WORK_TRANSACTION;
//以binder_work的形式挂载到目标进程的待处理队列中
list_add_tail(&t->work.entry, target_list);
//设置tcomplete事务的binder_work类型
tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
//以binder_work的形式挂载到当前Binder线程的待处理队列中
list_add_tail(&tcomplete->entry, &thread->todo);
```

binder_transaction t		
int	debug_id	++binder_last_id
binder_work	work	
	type	BINDER_WORK_TRANSACTION
	entry	target_list
binder_thread	from	thread
binder_transaction	from_parent	thread->transaction_stack
binder_transaction	to_parent	
binder_proc	to_proc	target_proc
unsigned	need_reply	1
binder_thread	to_thread	target_thread
binder_buffer	buffer	
	entry	
	rb_node	
	free	0
	allow_user_free	0
	async_transaction	
	debug_id	++binder_last_id
	transaction	t
	target_node	target_node
	data_size	tr->data_size
	offsets_size	tr->offsets_size
	data	tr->data.ptr.buffer
unsigned	code	tr->code
unsigned	flags	tr->flags
long	priority	task_nice(current)
long	saved_priority	
uid_t	sender_euid	proc->tsk->cred->euid

然后唤醒目标进程

```
[cpp]
01. wake_up_interruptible(target_wait);
```

由于mHandle =0

```
[cpp]
01. target_node = binder_context_mgr_node;
02. target_proc = target_node->proc;
03. target_list = &target_proc->todo;
04. target_wait = &target_proc->wait;
```

此时的目标进程就是ServiceManager进程，ServiceManager进程此时睡眠在binder_thread_read函数中，被唤醒后继续执行binder_thread_read函数，同时客户端进程将从刚才执行的binder_transaction函数返回到binder_ioctl函数，由于bwr.read_size = mIn.dataCapacity()，于是进入binder_thread_read函数：

```
[cpp]
01. static int binder_thread_read(struct binder_proc *proc,
02.                   struct binder_thread *thread,
03.                   void  __user *buffer, int size,
04.                   signed long *consumed, int non_block)
05. {
06.     void __user *ptr = buffer + *consumed;
07.     void __user *end = buffer + size;
08.
09.     int ret = 0;
10.     int wait_for_proc_work;
11.     // *consumed == 0
12.     if (*consumed == 0) {
13.         //写入一个值BR_NOOP到参数ptr指向的缓冲区中去
14.         if (put_user(BR_NOOP, (uint32_t __user *)ptr))
15.             return -EFAULT;
16.         ptr += sizeof(uint32_t);
17.     }
18. retry:
19.     //由于tcomplete事务被挂载到了当前Binder线程的待处理队列中，因此wait_for_proc_work = false
20.     wait_for_proc_work = thread->transaction_stack == NULL && list_empty(&thread->todo);
21.     //由于在初始化binder_thread时return_error被设置为BR_OK，因此这里条件不成立
22.     if (thread->return_error != BR_OK && ptr < end) {
23.         if (thread->return_error2 != BR_OK) {
24.             if (put_user(thread->return_error2, (uint32_t __user *)ptr))
25.                 return -EFAULT;
26.             ptr += sizeof(uint32_t);
27.             if (ptr == end)
28.                 goto done;
29.             thread->return_error2 = BR_OK;
30.         }
31.         if (put_user(thread->return_error, (uint32_t __user *)ptr))
32.             return -EFAULT;
33.         ptr += sizeof(uint32_t);
34.         thread->return_error = BR_OK;
35.         goto done;
36.     }
37.     //设置binder线程为等待状态
38.     thread->looper |= BINDER_LOOPER_STATE_WAITING;
39.     //如果当前线程没有事务需要处理，则增加proc->ready_threads计数
40.     if (wait_for_proc_work)
41.         proc->ready_threads++;
42.     mutex_unlock(&binder_lock);
43.     if (wait_for_proc_work) {
44.         if (!(thread-
>looper & (BINDER_LOOPER_STATE_REGISTERED | BINDER_LOOPER_STATE_ENTERED))) {
45.             wait_event_interruptible(binder_user_error_wait,binder_stop_on_user_error < 2);
46.         }
47.         //调用binder_set_nice函数设置当前线程的优先级别为proc->default_priority
48.         binder_set_nice(proc->default_priority);
49.         //文件打开模式为非阻塞模式，函数就直接返回-EAGAIN，要求用户重新执行ioctl
```

```
        if (non_block) {
            if (!binder_has_proc_work(proc, thread))
                ret = -EAGAIN;
        } else
            ret = wait_event_interruptible_exclusive(proc->wait, binder_has_proc_work(proc, thread));
    } else {
        if (non_block) {
            if (!binder_has_thread_work(thread))
                ret = -EAGAIN;
        } else
            //当前线程就通过wait_event_interruptible_exclusive函数进入休眠状态，等待请求到来再唤醒了。
            ret = wait_event_interruptible(thread->wait, binder_has_thread_work(thread));
    }
```

函数通过wait_event_interruptible_exclusive睡眠等待目标进程发送服务注册结果，到此和Binder驱动交互的客户端当前Binder线程就进入睡眠等待状态了。接下来分析目标进程被唤醒后的执行过程。

目标进程IPC数据接收过程

ServiceManager 进程启动源码分析分析了ServiceManager进程的启动过程，ServiceManager启动完成后最终将睡眠等待客户进程的请求，前面分析了当客户进程将IPC数据发送给目标进程后，会唤醒正在睡眠等待客户请求的目标进程，当ServiceManager进程被唤醒后，会继续执行睡眠等待wait_event_interruptible之后的代码，这部分代码如下：

```cpp
mutex_lock(&binder_lock);
//在ServiceManager 进程启动源码分析中已经分析了wait_for_proc_work = true
if (wait_for_proc_work)
    //指定线程去处理客户请求，因此剩余的空闲线程数量减1
    proc->ready_threads--;
//清除线程等待标志位
thread->looper &= ~BINDER_LOOPER_STATE_WAITING;
if (ret)
    return ret;

while (1) {
    uint32_t cmd;
    struct binder_transaction_data tr;
    struct binder_work *w;
    struct binder_transaction *t = NULL;
    //如果当前线程的待处理队列不为空
    if (!list_empty(&thread->todo))
        //从待处理队列中取出binder_work，事务项是通过binder_work挂载到待处理队列的
        w = list_first_entry(&thread->todo, struct binder_work, entry);
    //如果当前进程的待处理队列不为空
    else if (!list_empty(&proc->todo) && wait_for_proc_work)
        //从待处理队列中取出binder_work
        w = list_first_entry(&proc->todo, struct binder_work, entry);
    else {
        if (ptr - buffer == 4 && !(thread-
```

```
>looper & BINDER_LOOPER_STATE_NEED_RETURN)) /* no data added */
            goto retry;
        break;
    }

    if (end - ptr < sizeof(tr) + 4)
        break;
    //处理不同类型的binder_work
    switch (w->type) {
    case BINDER_WORK_TRANSACTION: {
        //取出客户进程发送过来的事务
        t = container_of(w, struct binder_transaction, work);
    } break;
    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;
    case BINDER_WORK_NODE: {
        struct binder_node *node = container_of(w, struct binder_node, work);
        uint32_t cmd = BR_NOOP;
        const char *cmd_name;
        int strong = node->internal_strong_refs || node->local_strong_refs;
        int weak = !hlist_empty(&node->refs) || node->local_weak_refs || strong;
        if (weak && !node->has_weak_ref) {
            cmd = BR_INCREFS;
            cmd_name = "BR_INCREFS";
            node->has_weak_ref = 1;
            node->pending_weak_ref = 1;
            node->local_weak_refs++;
        } else if (strong && !node->has_strong_ref) {
            cmd = BR_ACQUIRE;
            cmd_name = "BR_ACQUIRE";
            node->has_strong_ref = 1;
            node->pending_strong_ref = 1;
            node->local_strong_refs++;
        } else if (!strong && node->has_strong_ref) {
            cmd = BR_RELEASE;
            cmd_name = "BR_RELEASE";
            node->has_strong_ref = 0;
        } else if (!weak && node->has_weak_ref) {
            cmd = BR_DECREFS;
            cmd_name = "BR_DECREFS";
            node->has_weak_ref = 0;
        }
        if (cmd != BR_NOOP) {
            if (put_user(cmd, (uint32_t __user *)ptr))
                return -EFAULT;
            ptr += sizeof(uint32_t);
            if (put_user(node->ptr, (void * __user *)ptr))
                return -EFAULT;
            ptr += sizeof(void *);
```

```
            if (put_user(node->cookie, (void * __user *)ptr))
                return -EFAULT;
            ptr += sizeof(void *);
            binder_stat_br(proc, thread, cmd);
        } else {
            list_del_init(&w->entry);
            if (!weak && !strong) {
                kfree(node);
                binder_stats_deleted(BINDER_STAT_NODE);
            } else {

            }
        }
    } break;
    case BINDER_WORK_DEAD_BINDER:
    case BINDER_WORK_DEAD_BINDER_AND_CLEAR:
    case BINDER_WORK_CLEAR_DEATH_NOTIFICATION: {
        struct binder_ref_death *death;
        uint32_t cmd;
        death = container_of(w, struct binder_ref_death, work);
        if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION)
            cmd = BR_CLEAR_DEATH_NOTIFICATION_DONE;
        else
            cmd = BR_DEAD_BINDER;
        if (put_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
        if (put_user(death->cookie, (void * __user *)ptr))
            return -EFAULT;
        ptr += sizeof(void *);
        if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION) {
            list_del(&w->entry);
            kfree(death);
            binder_stats_deleted(BINDER_STAT_DEATH);
        } else
            list_move(&w->entry, &proc->delivered_death);
        if (cmd == BR_DEAD_BINDER)
            goto done; /* DEAD_BINDER notifications can cause transactions */
    } break;
    }

    if (!t)
        continue;

    BUG_ON(t->buffer == NULL);
    if (t->buffer->target_node) {
        //取出客户进程发送过来的事务的目标Binder实体节点
        struct binder_node *target_node = t->buffer->target_node;
        //初始化binder_transaction_data
        tr.target.ptr = target_node->ptr;
        tr.cookie =  target_node->cookie;
        t->saved_priority = task_nice(current);
        if (t->priority < target_node->min_priority &&
            !(t->flags & TF_ONE_WAY))
            binder_set_nice(t->priority);
        else if (!(t->flags & TF_ONE_WAY) ||
             t->saved_priority > target_node->min_priority)
```

```
            binder_set_nice(target_node->min_priority);
        cmd = BR_TRANSACTION;
    } else {
        tr.target.ptr = NULL;
        tr.cookie = NULL;
        cmd = BR_REPLY;
    }
    tr.code = t->code;
    tr.flags = t->flags;
    tr.sender_euid = t->sender_euid;

    if (t->from) {
        struct task_struct *sender = t->from->proc->tsk;
        tr.sender_pid = task_tgid_nr_ns(sender,current->nsproxy->pid_ns);
    } else {
        tr.sender_pid = 0;
    }

    tr.data_size = t->buffer->data_size;
    tr.offsets_size = t->buffer->offsets_size;
    tr.data.ptr.buffer = (void *)t->buffer->data + proc->user_buffer_offset;
    tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data_size,sizeof(void *));
    //向ptr写入Binder命令头
    if (put_user(cmd, (uint32_t __user *)ptr))
        return -EFAULT;
    ptr += sizeof(uint32_t);
    //向ptr写入binder_transaction_data
    if (copy_to_user(ptr, &tr, sizeof(tr)))
        return -EFAULT;
    ptr += sizeof(tr);
    binder_stat_br(proc, thread, cmd);
    list_del(&t->work.entry);
    t->buffer->allow_user_free = 1;
    if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
        t->to_parent = thread->transaction_stack;
        t->to_thread = thread;
        //将客户进程发送过来的事务项设置到当前线程的事务堆栈中
        thread->transaction_stack = t;
    } else {                                                    □载扌
        t->buffer->transaction = NULL;
        kfree(t);
        binder_stats_deleted(BINDER_STAT_TRANSACTION);
    }
    break;
}

done:
*consumed = ptr - buffer;                                       □载扌
if (proc->requested_threads + proc->ready_threads == 0 &&
    proc->requested_threads_started < proc->max_threads &&
    (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
     BINDER_LOOPER_STATE_ENTERED)) /* the user-space code fails to */
     /*spawn a new thread if we leave this out */) {
    proc->requested_threads++;                                  □载扌
    if (put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffer))
        return -EFAULT;
}
```

```
196.  return 0;
```

ServiceManager进程被唤醒后，首先从当前进程或线程的待处理队列todo中取出binder_work，这是客户进程在发送IPC数据时以binder_work的形式挂载到了目标进程或线程的todo队列中，这里就从中取出来，因为客户进程在将一个事务添加到目标进程的待处理队列前将该事务封装成了binder_work形式，并且设置了该binder_work的类型为BINDER_WORK_TRANSACTION

```
[cpp]
01.  //设置事务类型为BINDER_WORK_TRANSACTION
02.  t->work.type = BINDER_WORK_TRANSACTION;
03.  //将事务添加到目标进程或线程的待处理队列中
04.  list_add_tail(&t->work.entry, target_list);
```

根据取出的binder_work来得到客户进程发送过来的事务binder_transaction

```
[cpp]
01.  t = container_of(w, struct binder_transaction, work);
```

然后从该事务中取出IPC数据，并封装成binder_transaction_data结构体

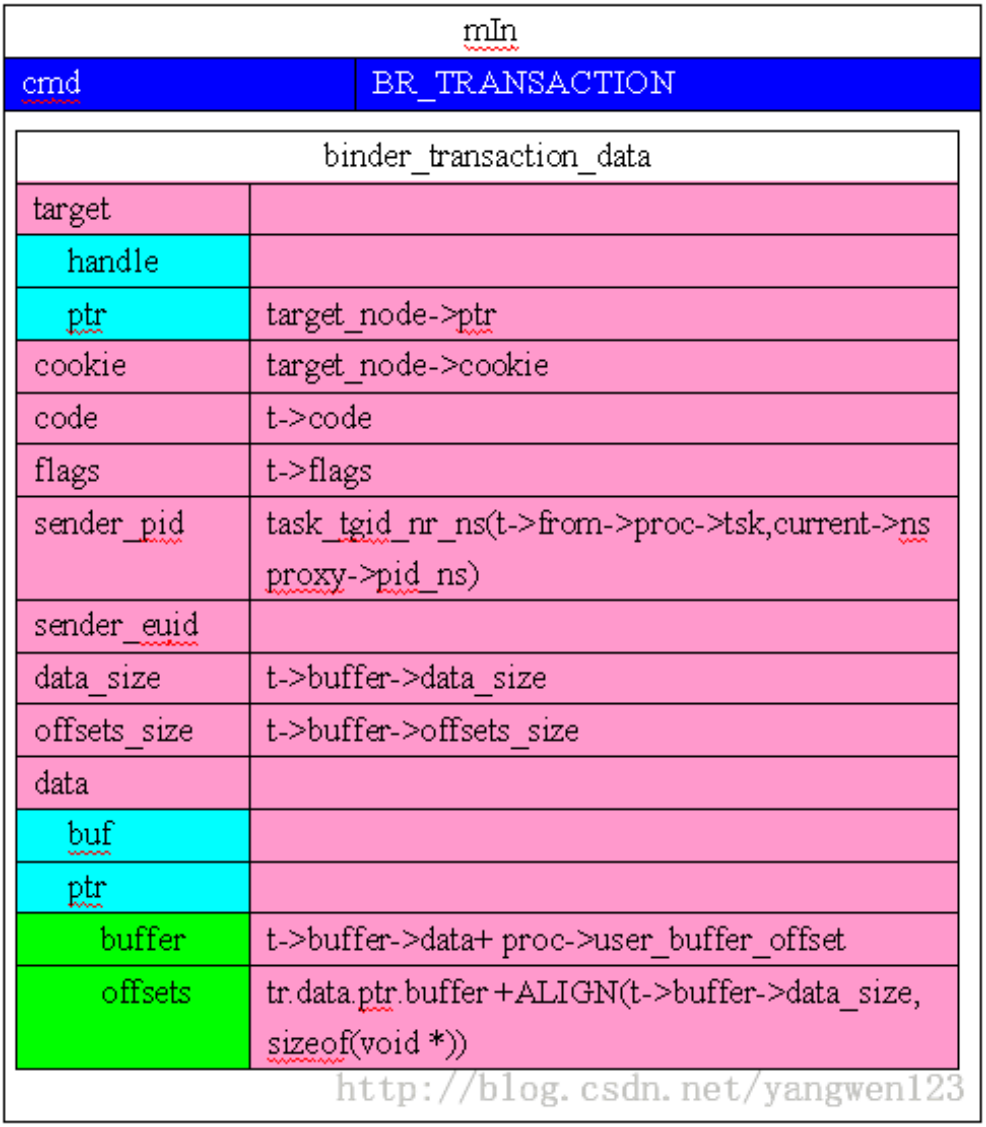
```
[cpp]
01.  if (t->buffer->target_node) {
02.      //取出客户进程发送过来的事务的目标Binder实体节点
03.      struct binder_node *target_node = t->buffer->target_node;
04.      //初始化binder_transaction_data
05.      tr.target.ptr = target_node->ptr;
06.      tr.cookie =  target_node->cookie;
07.      t->saved_priority = task_nice(current);
08.      if (t->priority < target_node->min_priority &&
09.          !(t->flags & TF_ONE_WAY))
10.          binder_set_nice(t->priority);
11.      else if (!(t->flags & TF_ONE_WAY) ||
12.           t->saved_priority > target_node->min_priority)
13.          binder_set_nice(target_node->min_priority);
14.      cmd = BR_TRANSACTION;
15.  } else {
16.      tr.target.ptr = NULL;
17.      tr.cookie = NULL;
18.      cmd = BR_REPLY;
19.  }
20.  tr.code = t->code;
21.  tr.flags = t->flags;
22.  tr.sender_euid = t->sender_euid;
23.
24.  if (t->from) {
25.      struct task_struct *sender = t->from->proc->tsk;
26.      tr.sender_pid = task_tgid_nr_ns(sender,current->nsproxy->pid_ns);
27.  } else {
28.      tr.sender_pid = 0;
29.  }
30.
31.  tr.data_size = t->buffer->data_size;
```

```
32. tr.offsets_size = t->buffer->offsets_size;
33. tr.data.ptr.buffer = (void *)t->buffer->data + proc->user_buffer_offset;
34. tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data_size,sizeof(void *));
```

binder_transaction_data	
target	
handle	
ptr	target_node->ptr
cookie	target_node->cookie
code	t->code
flags	t->flags
sender_pid	task_tgid_nr_ns(t->from->proc->tsk,current->nsproxy->pid_ns)
sender_euid	
data_size	t->buffer->data_size
offsets_size	t->buffer->offsets_size
data	
buf	
ptr	
buffer	t->buffer->data+ proc->user_buffer_offset
offsets	tr.data.ptr.buffer +ALIGN(t->buffer->data_size, sizeof(void *))

然后写入用户空间的buffer中

```
[cpp]
01. //向ptr写入Binder命令头
02. if (put_user(cmd, (uint32_t __user *)ptr))
03.     return -EFAULT;
04. ptr += sizeof(uint32_t);
05. //向ptr写入binder_transaction_data
06. if (copy_to_user(ptr, &tr, sizeof(tr)))
07.     return -EFAULT;
08. ptr += sizeof(tr);
```



这样ServiceManager进程在用户空间就真正得到了客户进程发送过来的服务注册信息。接着将客户进程发送过来的事务项添加到当前线程的事务堆栈中，交给当前线程处理。

```cpp
//将客户进程发送过来的事务项设置到当前线程的事务堆栈中
thread->transaction_stack = t;
```

到此ServiceManager进程就从binder_thread_read函数中返回，从binder_ioctl函数中返回到用户空间中的binder_loop函数中去

```cpp
void binder_loop(struct binder_state *bs, binder_handler func)
{
    int res;
    struct binder_write_read bwr;
    unsigned readbuf[32];
    bwr.write_size = 0;
    bwr.write_consumed = 0;
    bwr.write_buffer = 0;
    readbuf[0] = BC_ENTER_LOOPER;
    binder_write(bs, readbuf, sizeof(unsigned));
    for (;;) {
```

```
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (unsigned) readbuf;
        //ServiceManager进程接受到客户进程请求返回
        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
        if (res < 0) {
            ALOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
            break;
        }
        //解析并处理客户进程发送过来的IPC数据
        res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
        if (res == 0) {
            ALOGE("binder_loop: unexpected reply?!\n");
            break;
        }
        if (res < 0) {
            ALOGE("binder_loop: io error %d  %s\n", res, strerror(errno));
            break;
        }
    }
}
```

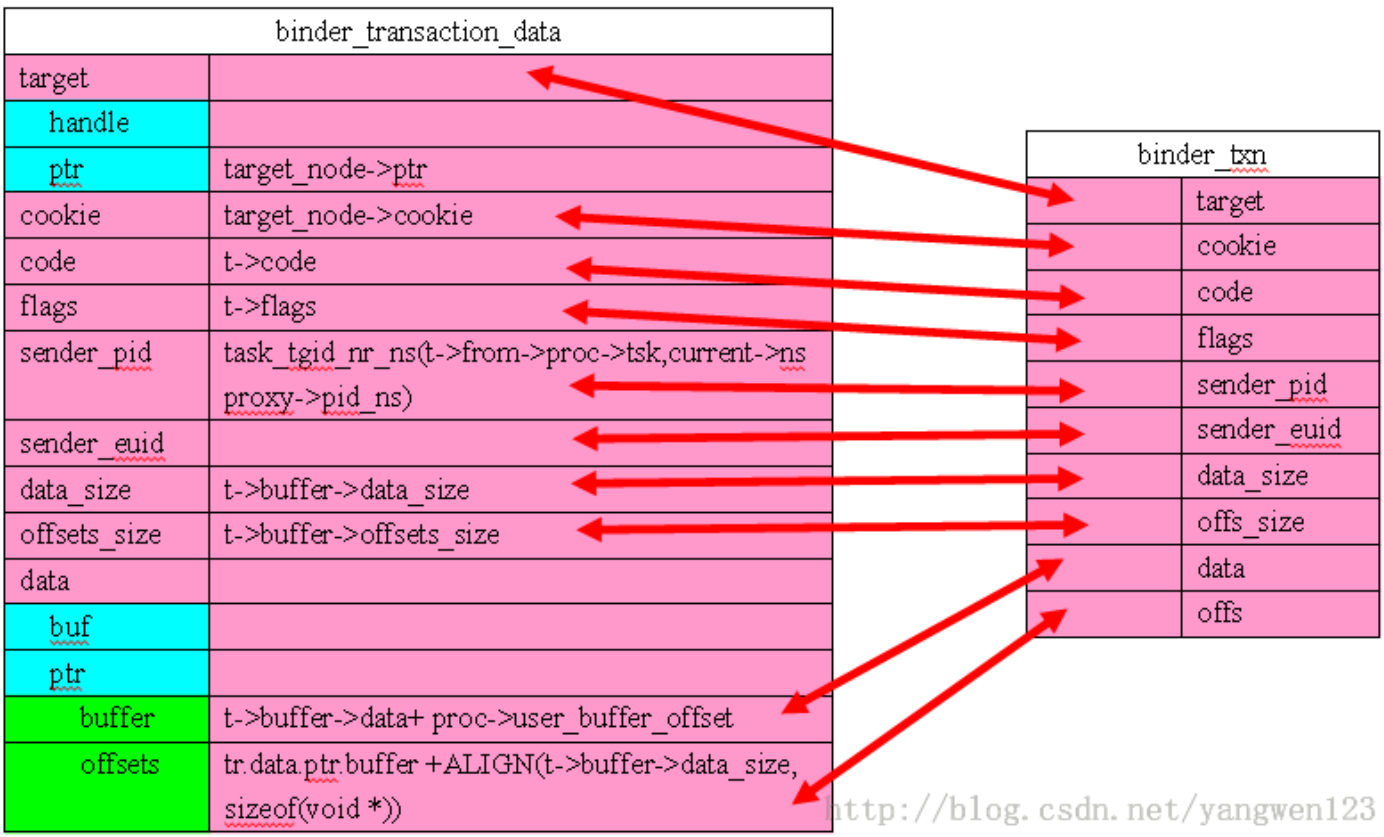
ServiceManager进程接收到客户进程的服务注册请求后，从ioctl函数中返回，并得到客户进程发送过来的注册信息，然后调用binder_parse函数进行数据解析，并传入回调函数svcmgr_handler的指针。

```
[cpp]

int binder_parse(struct binder_state *bs, struct binder_io *bio,
                 uint32_t *ptr, uint32_t size, binder_handler func)
{
    int r = 1;
    uint32_t *end = ptr + (size / 4);
    //处理不同的Binder命令头
    while (ptr < end) {
        uint32_t cmd = *ptr++;
        switch(cmd) {
        case BR_NOOP:
        case BR_TRANSACTION_COMPLETE:
        case BR_INCREFS:
        case BR_ACQUIRE:
        case BR_RELEASE:
        case BR_DECREFS:
        case BR_TRANSACTION:
        case BR_REPLY:
        case BR_DEAD_BINDER:
        case BR_FAILED_REPLY:
        case BR_DEAD_REPLY:
        default:
            return -1;
        }
    }
    return r;
}
```

此时的cmd=BR_TRANSACTION

```cpp
case BR_TRANSACTION: {
    //客户进程发送过来的IPC数据
    struct binder_txn *txn = (void *) ptr;
    if ((end - ptr) * sizeof(uint32_t) < sizeof(struct binder_txn)) {
        ALOGE("parse: txn too small!\n");
        return -1;
    }
    binder_dump_txn(txn);
    if (func) {
        unsigned rdata[256/4];
        struct binder_io msg;
        struct binder_io reply;
        int res;
        bio_init(&reply, rdata, sizeof(rdata), 4);
        bio_init_from_txn(&msg, txn);
        //调用回调函数处理IPC数据
        res = func(bs, txn, &msg, &reply);
        //向客户进程发送处理结果
        binder_send_reply(bs, &reply, txn->data, res);
    }
    ptr += sizeof(*txn) / sizeof(uint32_t);
    break;
}
```



函数首先初始化结构体数据，然后调用通过binder_loop函数传进来的回调函数svcmgr_handler来完成服务查询工作，将查询结果通过binder_send_reply函数发送给客户端进程。

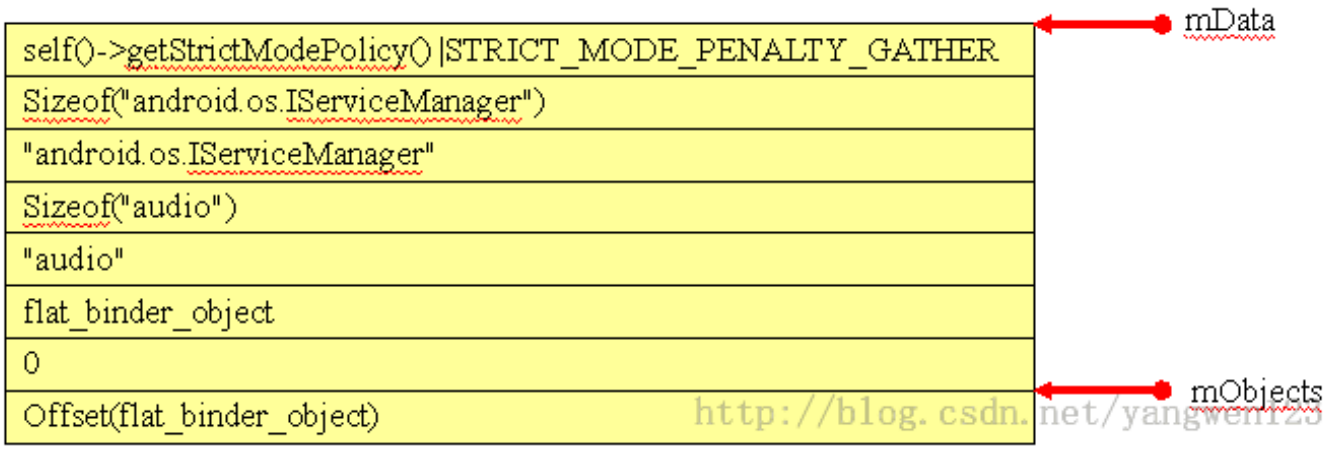
目标进程服务注册过程

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;
```

msg:



ServiceManager进程对服务的注册是通过回调函数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 self()->getStrictModePolicy() |STRICT_MODE_PENALTY_GATHER
    strict_policy = bio_get_uint32(msg);
    //读取字符串，在AudioService服务注册时，发送了以下数据:
    //data.writeInterfaceToken("android.os.IServiceManager");
    //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;
    default:
        ALOGE("unknown code %d\n", txn->code);
        return -1;
    }
    bio_put_uint32(reply, 0);
    return 0;
}
```

在客户端和服务端都统一定义了对应的函数调用码

客户端进程：

```
[cpp]

enum {
    GET_SERVICE_TRANSACTION = IBinder::FIRST_CALL_TRANSACTION,
    CHECK_SERVICE_TRANSACTION,
    ADD_SERVICE_TRANSACTION,
    LIST_SERVICES_TRANSACTION,
};
```

服务端进程：

```
[cpp]

enum {
    SVC_MGR_GET_SERVICE = 1,
    SVC_MGR_CHECK_SERVICE,
    SVC_MGR_ADD_SERVICE,
    SVC_MGR_LIST_SERVICES,
};
```

ServiceManager进程注册服务过程：

```cpp
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;
```

函数bio_get_ref()是从msg中取出flat_binder_object结构体的成员binder 的值

```cpp
void *bio_get_ref(struct binder_io *bio)
{
    struct binder_object *obj;
    //从传进来的参数bio中取出客户进程发送过来的flat_binder_object结构体，并用binder_object结构来表示
    obj = _bio_get_obj(bio);
    if (!obj)
        return 0;
    //如果客户进程发送过来的flat_binder_object类型为BINDER_TYPE_HANDLE
    if (obj->type == BINDER_TYPE_HANDLE)
        //返回Binder引用描述符
        return obj->pointer;

    return 0;
}
```

在Android 数据Parcel序列化过程源码分析中介绍了writeStrongBinder函数将Binder实体对象封装为flat_binde_robject结构体写入到Parcel对象中，在前面的binder_transaction()函数中，如果传输的是一个Binder实体对象，首先会在内核空间为该Binder实体对象创建Binder节点，同时为ServiceManager进程创建引用该Binder节点的Binder引用对象，并且修改该Binder引用对象在内核空间的描述flat_binder_object的类型及句柄值为：

```cpp
obj->flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
obj->type = BINDER_TYPE_HANDLE;
obj->cookie = local;
obj->handle = ref->desc;
```

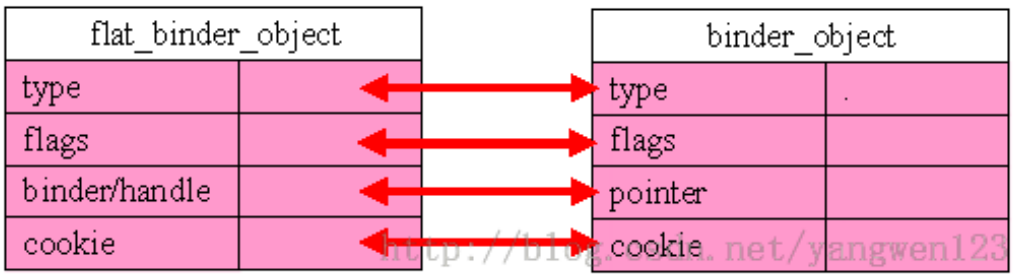
flat_binder_object对象的查找过程

```cpp
static struct binder_object *_bio_get_obj(struct binder_io *bio)
{
    unsigned n;
    unsigned off = bio->data - bio->data0;
    //遍历flat_binder_object对象地址偏移数组
    for (n = 0; n < bio->offs_avail; n++) {
        if (bio->offs[n] == off)
```

```
            //读取flat_binder_object对象，并用binder_object结构体来表示
            return bio_get(bio, sizeof(struct binder_object));
    }

    bio->data_avail = 0;
    bio->flags |= BIO_F_OVERFLOW;
    return 0;
}
```

flat_binder_object结构与binder_object结构之间的关系

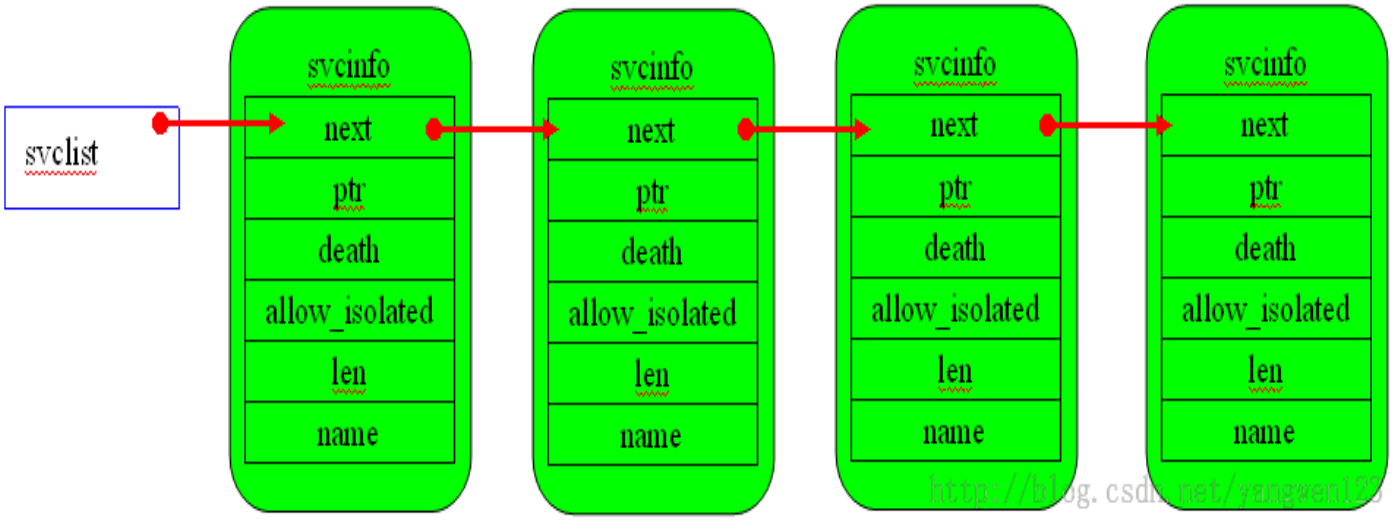


svcmgr_handler函数最后调用do_add_service函数来注册服务。

```
[cpp]

int do_add_service(struct binder_state *bs,
                   uint16_t *s, unsigned len,
                   void *ptr, unsigned uid, int allow_isolated)
{
    /*
        s = "audio"
        len = sizeof("audio")
        allow_isolated = 0
    */
    struct svcinfo *si;
    if (!ptr || (len == 0) || (len > 127))
        return -1;
    //权限检查，通过allowed数组设置了各个服务的权限值
    if (!svc_can_register(uid, s)) {
        ALOGE("add_service('%s',%p) uid=%d - PERMISSION DENIED\n",str8(s), ptr, uid);
        return -1;
    }
    //从服务列表svclist中根据服务名称查找服务
    si = find_svc(s, len);
    //服务已经被注册了
    if (si) {
        if (si->ptr) {
            ALOGE("add_service('%s',%p) uid=%d - ALREADY REGISTERED, OVERRIDE\n",str8(s), ptr, uic
            svcinfo_death(bs, si);
        }
        si->ptr = ptr;
    //服务未注册，创建一个新的服务，初始化并插入到svclist链表中
    } else {
        si = malloc(sizeof(*si) + (len + 1) * sizeof(uint16_t));
        if (!si) {
            ALOGE("add_service('%s',%p) uid=%d - OUT OF MEMORY\n",str8(s), ptr, uid);
            return -1;
```

```
        }
        si->ptr = ptr;
        si->len = len;
        memcpy(si->name, s, (len + 1) * sizeof(uint16_t));
        si->name[len] = '\0';
        si->death.func = svcinfo_death;
        si->death.ptr = si;
        si->allow_isolated = allow_isolated;
        si->next = svclist;
        svclist = si;
    }
    binder_acquire(bs, ptr);
    binder_link_to_death(bs, ptr, &si->death);
    return 0;
}
```



服务注册完成后，返回到svcmgr_handler函数中，并执行最后一条语句

```
[cpp]
bio_put_uint32(reply, 0);
```

设置reply指向的地址空间值为0，最后又返回到函数binder_parse中，通过binder_send_reply(bs, &reply, txn->data, res)语句向客户进程发送服务注册结果，当服务注册成功svcmgr_handler函数返回0,因此binder_send_reply(bs, &reply, txn->data, 0)

目标进程向客户进程发送服务注册结果

```
[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));
}
```

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

函数调用binder_write将服务注册结果通过Binder驱动发送给客户进程，前面介绍了客户进程此时正睡眠在Binder线程数据读取中

```
[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;
}
```

发送的数据为：

bwr.write_size = sizeof(data);

bwr.write_consumed = 0;

bwr.write_buffer = (unsigned) data;

bwr.read_size = 0;

bwr.read_consumed = 0;

bwr.read_buffer = 0;

通过ioctl命令控制函数再次进入到Binder驱动程序中，并进入BINDER_WRITE_READ命令处理过程中，由于bwr.write_size > 0而bwr.read_size = 0，因此在binder_ioctl中只调用binder_thread_write函数进行数据发送。

```
[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:
        case BC_TRANSACTION:
        case BC_REPLY: {
        default:
            return -EINVAL;
        }
```

```
            *consumed = ptr - buffer;
        }
        return 0;
    }
```

binder_thread_write函数循环从buffer中读取命令cmd，然后对不同的命令进行不同的处理，从上图data的存储结构可以看出，binder_thread_write函数首先读取到BC_FREE_BUFFER命令，BC_FREE_BUFFER命令的处理如下：

```
[cpp]
case BC_FREE_BUFFER: {
    void __user *data_ptr;
    struct binder_buffer *buffer;
    //读取buffer的地址
    if (get_user(data_ptr, (void * __user *)ptr))
        return -EFAULT;
    //移动ptr指针的位置
    ptr += sizeof(void *);
    //从当前客户进程binder_proc中查找指定的binder_buffer
    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);
    }
    //释放该binder_buffer
    binder_transaction_buffer_release(proc, buffer, NULL);
    binder_free_buf(proc, buffer);
    break;
}
```

首先从当前进程的binder_proc中查找出指定的binder_buffer，然后判断是否可以释放该binder_buffer的内存空间，然后调用binder_free_buf函数释放该内核缓冲区。处理完BC_FREE_BUFFER命令后，binder_thread_write函数继续遍历buffer，从而取出第二个命令BC_REPLY，以下是对BC_REPLY命令的处理过程：

```
[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;
    }
```

首先从用户空间的buffer中取出binder_txn数据结构，并保持到binder_transaction_data中，binder_txn和binder_transaction_data结构体之间的对应关系在前面已经通过图来说明了。将用户空间的binder_txn拷贝到内核空间的binder_transaction_data中后，调用binder_transaction函数进行夸进程数据发送。关于binder_transaction函数的详细介绍请查看Android IPC数据在内核空间中的发送过程分析。通过binder_transaction函数将参数binder_transaction_data来封装一个工作事务binder_transaction，然后唤醒正在睡眠等待的客户线程。服务注册完成并将结果发送回客户进程后，返回到函数binder_loop中

```
[cpp]
void binder_loop(struct binder_state *bs, binder_handler func)
{
    int res;
    struct binder_write_read bwr;
    unsigned readbuf[32];
    bwr.write_size = 0;
    bwr.write_consumed = 0;
    bwr.write_buffer = 0;
    readbuf[0] = BC_ENTER_LOOPER;                                    □载扌
    binder_write(bs, readbuf, sizeof(unsigned));
    for (;;) {
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (unsigned) readbuf;
        //ServiceManager进程接受到客户进程请求返回
        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
        if (res < 0) {
            ALOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
            break;
        }
        //解析并处理客户进程发送过来的IPC数据
        res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
        if (res == 0) {
            ALOGE("binder_loop: unexpected reply?!\n");
            break;
        }
        if (res < 0) {
            ALOGE("binder_loop: io error %d  %s\n", res, strerror(errno));
            break;
        }
    }
}
```

binder_loop()函数闭环执行ioctl()和binder_parse()函数，当binder_parse()函数返回时，binder_loop()又在一次通过ioctl系统调用进入到Binder驱动中，此时发送的数据为：

bwr.write_size = 0;

bwr.write_consumed = 0;

bwr.write_buffer = 0;

bwr.read_size = sizeof(readbuf);

bwr.read_consumed = 0;

bwr.read_buffer = (unsigned) readbuf;

由于bwr.read_size >0，因此在binder_ioctl中，只执行binder_thread_read(), ServiceManager进程在Binder驱动中读取数据时，又会通过wait_event_interruptible()函数睡眠等待客户端发送请求。

客户进程接收目标进程发送过来的执行结果

客户线程被ServiceManager唤醒后，继续执行binder_thread_read函数，读取ServiceManager进程发送过来的binder_transaction事务，并从该事务中取出binder_transaction_data结构，因为客户进程发送的数据就是通过binder_transaction_data来描述的。客户线程从binder_thread_read函数返回到binder_ioctl函数，并一路向上返回到用户空间的talkWithDriver()函数，最后返回到waitForResponse()函数执行talkWithDriver()语句之后的代码

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

    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;

        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;
        //读取目标进程发送过来的命令
        cmd = mIn.readInt32();
        //处理不同的binder命令
        switch (cmd) {
        case BR_TRANSACTION_COMPLETE:
            if (!reply && !acquireResult) goto finish;
            break;

        case BR_DEAD_REPLY:
            err = DEAD_OBJECT;
            goto finish;

        case BR_FAILED_REPLY:
            err = FAILED_TRANSACTION;
            goto finish;

        case BR_ACQUIRE_RESULT:
            {
                ALOG_ASSERT(acquireResult != NULL, "Unexpected brACQUIRE_RESULT");
                const int32_t result = mIn.readInt32();
                if (!acquireResult) continue;
                *acquireResult = result ? NO_ERROR : INVALID_OPERATION;
            }
            goto finish;

        case BR_REPLY:
            {
```

```
                binder_transaction_data tr;
                err = mIn.read(&tr, sizeof(tr));
                ALOG_ASSERT(err == NO_ERROR, "Not enough command data for brREPLY");
                if (err != NO_ERROR) goto finish;

                if (reply) {
                    if ((tr.flags & TF_STATUS_CODE) == 0) {
                        reply->ipcSetDataReference(
                            reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                            tr.data_size,
                            reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                            tr.offsets_size/sizeof(size_t),
                            freeBuffer, this);
                    } else {
                        err = *static_cast<const status_t*>(tr.data.ptr.buffer);
                        freeBuffer(NULL,
                            reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                            tr.data_size,
                            reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                            tr.offsets_size/sizeof(size_t), this);
                    }
                } else {
                    freeBuffer(NULL,
                        reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                        tr.data_size,
                        reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                        tr.offsets_size/sizeof(size_t), this);
                    continue;
                }
            }
            goto finish;

        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;
}
```

在前面介绍目标进程向客户进程发送服务注册结果知道，ServiceManager进程发送了两个命令及参数到Binder驱动中，命令分别是BC_FREE_BUFFER和BC_REPLY

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

命令BC_FREE_BUFFER在Binder驱动中就完成了对内涵缓冲区的释放；执行命令BC_REPLY时，通过binder_transaction()函数将该命令下的数据发送给了客户进程，因此当客户进程被唤醒，并读取Binder数据时，既可以读取到BC_REPLY命令及其发送过来的数据，对命令BC_REPLY的处理过程如下：

```cpp
case BR_REPLY:
    {
        binder_transaction_data tr;
        err = mIn.read(&tr, sizeof(tr));
        ALOG_ASSERT(err == NO_ERROR, "Not enough command data for brREPLY");
        if (err != NO_ERROR) goto finish;

        if (reply) {
            if ((tr.flags & TF_STATUS_CODE) == 0) {
                reply->ipcSetDataReference(
                    reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                    tr.data_size,
                    reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                    tr.offsets_size/sizeof(size_t),
                    freeBuffer, this);
            } else {
                err = *static_cast<const status_t*>(tr.data.ptr.buffer);
                freeBuffer(NULL,
                    reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                    tr.data_size,
                    reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                    tr.offsets_size/sizeof(size_t), this);
            }
        } else {
```

```
            freeBuffer(NULL,
                reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                tr.data_size,
                reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                tr.offsets_size/sizeof(size_t), this);
            continue;
        }
    }
    goto finish;
```

这样客户进程就得到了目标进程ServiceManager发送过来的服务注册结果了，Android服务注册过程可以总结为以下几个步骤：

1）在Java层将服务名称及服务对应的Binder对象写入到Parcel对象中；

2）在C++层为该服务创建对应的Binder实体对象JavaBBinder，并且将上层发送过来的数据序列化到C++的Parcel对象中，同时使用flat_binder_object结构体来描述Binder实体对象；

3）通过ServiceManager的Binder代理对象将数据发送到Binder驱动中；

4）Binder驱动在内核空间中为传输的Binder实体对象创建对应的Binder节点；

5）Binder驱动在内核空间中为ServiceManager进程创建引用该服务Binder节点的Binder引用对象；

6）修改Binder驱动中传输的flat_binder_object的类型和描述符；

7）将服务名称和为ServiceManager进程创建的Binder引用对象的描述符注册到ServiceManager进程中；