Android服务函数远程调用源码分析

标签： android service transact IPCThreadState binder

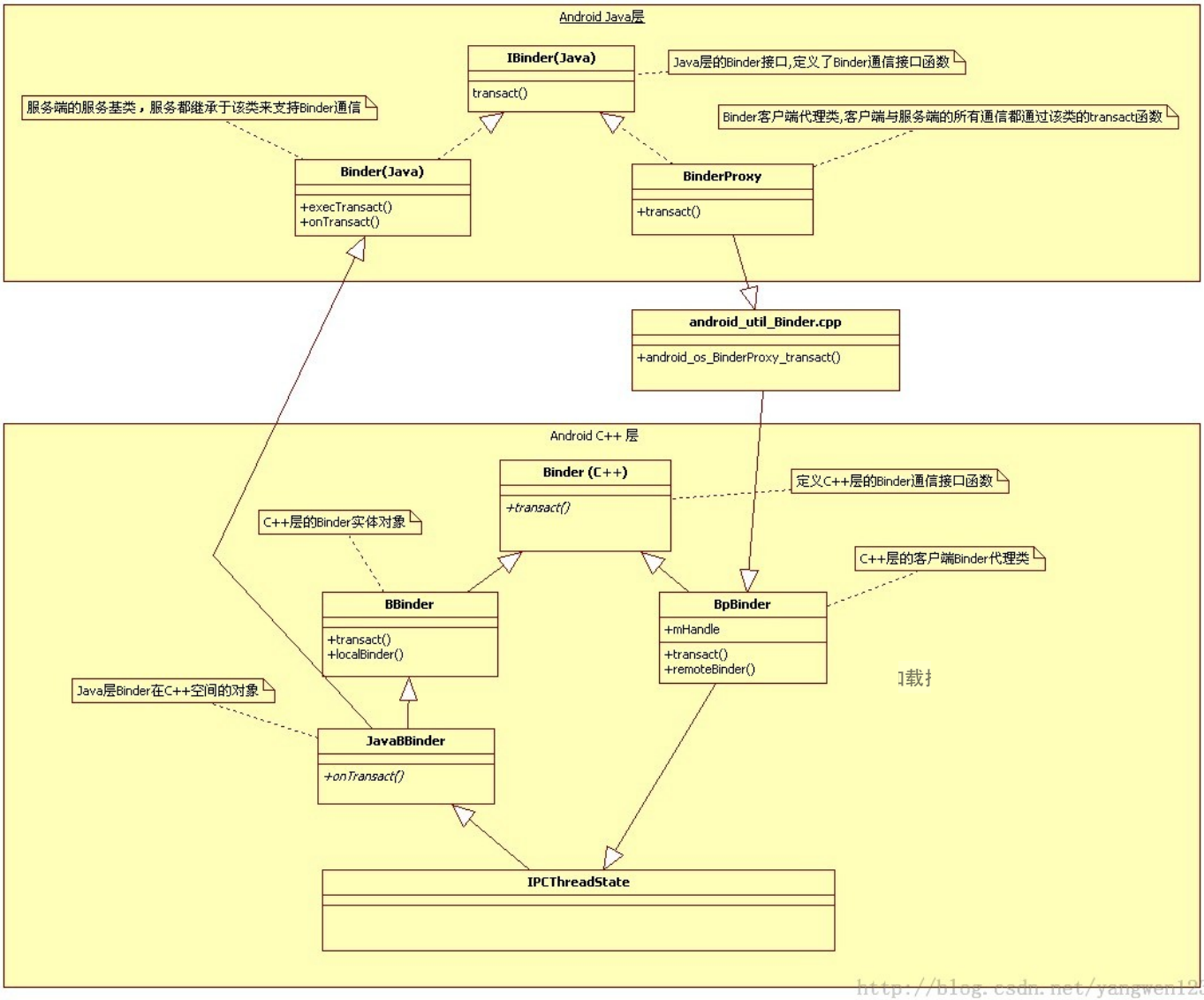
2013-07-12 19:59 1556人阅读 评论(0) 收藏 举报

分类：

【Android Binder通信】 (8)

目录(?) [-]

在Android服务查询完整过程源码分析中介绍了客户进程向ServiceManager进程查询服务的完整过程，ServiceManager进程根据服务名称在自身维护的服务链表中查找ServiceManager进程引用该服务在内核空间的Binder节点的Binder引用对象描述符，根据该描述符找到ServiceManager在内核空间对该服务Binder节点的Binder引用对象，在根据Binder引用对象找到引用的Binder节点，如果当前服务查询进程不是服务注册进程，则在内核空间中为当前进程创建引用服务Binder节点的Binder引用对象，并将该引用对象的句柄值返回到用户空间中，在用户空间中创建和通信相关的BpBinder及和业务相关的BpXXXService。得到服务代理对象后就可以通过RPC远程调用服务功能函数，和本地调用一样。本文在Android服务注册完整过程源码分析和Android服务查询完整过程源码分析的基础上以ActivityManager类的getRunningTasks函数为例进一步分析RPC远程函数调用过程。



Android Binder通信框架图

客户进程发送RPC调用请求

```[java]
01. public List<RunningTaskInfo> getRunningTasks(int maxNum, int flags, IThumbnailReceiver receiver)
02.         throws SecurityException {
03.     try {
04.         return ActivityManagerNative.getDefault().getTasks(maxNum, flags, receiver);
05.     } catch (RemoteException e) {
06.         // System dead, we will be dead too soon!
07.         return null;
08.     }
09. }
```

ActivityManagerNative.getDefault()通过单例模式创建ActivityManagerProxy代理对象

```[java]
01. static public IActivityManager getDefault() {
02.     return gDefault.get();
03. }
```

```
04.
05. private static final Singleton<IActivityManager> gDefault = new Singleton<IActivityManager>
    () {
06.     protected IActivityManager create() {
07.         IBinder b = ServiceManager.getService("activity");
08.         IActivityManager am = asInterface(b);
09.         return am;
10.     }
11. };
```

ServiceManager.getService("activity")在Android服务查询完整过程源码分析这篇文章中详细分析了，Binder驱动会为当前服务查询进程在内核空间创建Binder引用对象，并将该Binder引用对象的句柄值返回到当前进程的用户空间中，在当前进程的用户空间中根据该句柄值创建C++层的Binder通信代理对象BpBinder及Java层的Binder通信代理对象BinderProxy，同时创建与业务相关的代理对象XXXProxy对象。这里的变量b就是BinderProxy对象，通过asInterface()函数创建与特定业务相关的代理对象ActivityManagerProxy

```
[java]
01. static public IActivityManager asInterface(IBinder obj) {
02.     if (obj == null) {
03.         return null;
04.     }
05.     IActivityManager in =(IActivityManager)obj.queryLocalInterface(descriptor);
06.     if (in != null) {
07.         return in;
08.     }
09.     return new ActivityManagerProxy(obj);
10. }
```

ActivityManagerNative.getDefault()最终返回ActivityManagerProxy对象，从ServiceManager进程查询返回来的当前进程引用服务的Binder引用对象的句柄值保存在BpBinder对象的mHandle成员变量中。ActivityManagerNative.getDefault().getTasks(maxNum, flags, receiver)最终调用ActivityManagerProxy的getTasks()函数，该函数定义如下：

```
[java]
01. public List getTasks(int maxNum, int flags,IThumbnailReceiver receiver) throws RemoteException {
02.     Parcel data = Parcel.obtain();
03.     Parcel reply = Parcel.obtain();
04.     data.writeInterfaceToken(IActivityManager.descriptor);
05.     data.writeInt(maxNum);
06.     data.writeInt(flags);
07.     data.writeStrongBinder(receiver != null ? receiver.asBinder() : null);
08.     mRemote.transact(GET_TASKS_TRANSACTION, data, reply, 0);
09.     reply.readException();
10.     ArrayList list = null;
11.     int N = reply.readInt();
12.     if (N >= 0) {
13.         list = new ArrayList();
14.         while (N > 0) {
15.             ActivityManager.RunningTaskInfo info =
16.                     ActivityManager.RunningTaskInfo.CREATOR
17.                     .createFromParcel(reply);
```

```
            list.add(info);
            N--;
        }
    }
    data.recycle();
    reply.recycle();
    return list;
}
```

当前进程向服务进程发送的数据包括服务描述符和函数调用参数：

data.writeInterfaceToken(IActivityManager.descriptor);

data.writeInt(maxNum);

data.writeInt(flags);

data.writeStrongBinder(receiver != null ? receiver.asBinder() : null);

mRemote是BinderProxy对象，其transact()函数是本地函数，对于的JNI函数实现如下：

```[java]
static jboolean android_os_BinderProxy_transact(JNIEnv* env, jobject obj,
        jint code, jobject dataObj, jobject replyObj, jint flags) // throws RemoteException
{
    if (dataObj == NULL) {
        jniThrowNullPointerException(env, NULL);
        return JNI_FALSE;
    }

    Parcel* data = parcelForJavaObject(env, dataObj);
    if (data == NULL) {
        return JNI_FALSE;
    }
    Parcel* reply = parcelForJavaObject(env, replyObj);
    if (reply == NULL && replyObj != NULL) {
        return JNI_FALSE;
    }

    IBinder* target = (IBinder*)env->GetIntField(obj, gBinderProxyOffsets.mObject);
    if (target == NULL) {
        jniThrowException(env, "java/lang/IllegalStateException", "Binder has been finalized!");
        return JNI_FALSE;
    }

    // Only log the binder call duration for things on the Java-level main thread.
    // But if we don't
    const bool time_binder_calls = should_time_binder_calls();

    int64_t start_millis;
    if (time_binder_calls) {
        start_millis = uptimeMillis();
    }

    status_t err = target->transact(code, *data, reply, flags);

    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;
}
```

函数首先将Java层的Parcel对象转换为C++层的Parcel对象，然后通过Java层的BinderProxy成员变量mObject取得其对应的C++层的BpBinder对象地址，最后调用BpBinder对象的transact函数向服务进程发送函数调用码及函数参数

```
[cpp]
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

BpBinder对象直接使用IPCThreadState对象的transact()函数来发送参数信息，因为当前BpBinder对象的mHandle成员变量中保存了当前进程引用服务Binder节点的Binder引用对象句柄值。这里将该句柄值一起传到IPCThreadState对象的transact()函数中，一起发送到Binder驱动中，Binder驱动通过该句柄值实现Binder节点的寻址。

```
[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;
}
```

writeTransactionData()函数在Android IPC数据在内核空间中的发送过程分析有详细的介绍，该函数就是将函数调用码、函数参数、Binder引用对象句柄值保存到binder_transaction_data结构体中，然后将Binder命令和该结构体写入到IPCThreadState对象的成员变量mOut这个Parcel对象中。waitForResponse()就是将mOut发送到Binder驱动中，并等待服务进程返回函数执行结果。

```
[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:
            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;
}
```

该函数通过talkWithDriver()将前面打包好的数据发送到Binder驱动中，然后返回并读取IPCThreadState的成员变量mIn这个Parcel对象，

```cpp
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    LOG_ASSERT(mProcess->mDriverFD >= 0, "Binder driver is not opened");

    binder_write_read bwr;
    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();

    // This is what we'll read.
    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (long unsigned int)mIn.data();
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }
    // Return immediately if there is nothing to do.
    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;
}
```

该函数首先将发送的数据设置到binder_write_read结构体中

```
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();
```

最后通过ioctl进入到Binder驱动中，ioctl函数的执行将导致binder_ioctl()函数的调用。

```
[cpp]
01. static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
02. {
03.     int ret;
04.     struct binder_proc *proc = filp->private_data;
05.     struct binder_thread *thread;
06.     unsigned int size = _IOC_SIZE(cmd);
07.     void __user *ubuf = (void __user *)arg;
08.     ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
09.     if (ret)
10.         return ret;
11.
12.     mutex_lock(&binder_lock);
13.     thread = binder_get_thread(proc);
14.     if (thread == NULL) {
15.         ret = -ENOMEM;
16.         goto err;
17.     }
18.
19.     switch (cmd) {
20.     case BINDER_WRITE_READ: {
21.         struct binder_write_read bwr;
22.         if (size != sizeof(struct binder_write_read)) {
23.             ret = -EINVAL;
24.             goto err;
25.         }
26.         if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
27.             ret = -EFAULT;
28.             goto err;
29.         }
30.         if (bwr.write_size > 0) {
31.             ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer, bwr.write_siz
32.             if (ret < 0) {
33.                 bwr.read_consumed = 0;
34.                 if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
35.                     ret = -EFAULT;
36.                 goto err;
37.             }
38.         }
39.         if (bwr.read_size > 0) {
40.             ret = binder_thread_read(proc, thread, (void __user *)bwr.read_buffer, bwr.read_size,
    >f_flags & O_NONBLOCK);
41.             if (!list_empty(&proc->todo))
42.                 wake_up_interruptible(&proc->wait);
43.             if (ret < 0) {
44.                 if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
45.                     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;
}
```

由于bwr.write_size和bwr.read_size都大于0，因此binder_ioctl函数先执行Binder驱动写操作然后在执行Binder线程读操作。

```
[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;

    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;
}
```

由于在IPCThreadState的transact()函数中设置的Binder命令为BC_TRANSACTION，因此这里直接调用binder_transaction函数进行输出传输。该函数的具体分析请看Android IPC数据在内核空间中的发送过程分析。binder_transaction函数相当复杂，这里是Binder驱动的精华所在，其实现如下：

```
[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) {
        ...
    } 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 {
            ...
```

```
        }
        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);

    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;
    }
    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;
        }
    }
    if (reply) {
        ...
    } 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;
}
```

该函数主要包括三个部分内容：

1）查找服务注册进程与线程：通过当前进程引用服务Binder节点的Binder引用对象句柄值从当前进程中查找到Binder引用对象，在根据该Binder引用对象找到服务对应的Binder节点，然后通过Binder节点找到注册服务的目标进程及目标线程，查找过程为：

句柄值——> Binder引用对象 ——> Binder节点 ——> 服务注册进程

2）根据参数binder_transaction_data中的数据创建并初始化一个事务项t和一个完成事务项tcomplete，同时根据传输的Binder对象类型，修改Binder描述；

3）将事务项挂载到目标进程或目标线程的待处理队列中，同时将完成事务项挂载到当前Binder线程的待处理队列中；

4）唤醒目标进程或者目标线程；

该函数将发送的数据封装到事务项中并挂载到目标进程的待处理队列中同时唤醒目标进程后，层层返回到binder_ioctl()函数，然后继续执行Binder驱动读操作。在执行binder_thread_read函数时，由于前面挂载了一个完成事务项到当前线程的待处理队列中，因此Binder驱动会执行该完成事务项

```
[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);
    binder_debug(BINDER_DEBUG_TRANSACTION_COMPLETE,
                 "binder: %d:%d BR_TRANSACTION_COMPLETE\n",
                 proc->pid, thread->pid);
```

```
11.
12.     list_del(&w->entry);
13.     kfree(w);
14.     binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
15. } break;
```

处理过程比较简单，首先向当前进程的用户空间返回一个BR_TRANSACTION_COMPLETE命令，然后释放该事务项。执行完binder_thread_read函数后，返回向上返回到用户空间的talkWithDriver()中执行以下操作：

[cpp]

```
01. if (err >= NO_ERROR) {
02.     if (bwr.write_consumed > 0) {
03.         if (bwr.write_consumed < (ssize_t)mOut.dataSize())
04.             mOut.remove(0, bwr.write_consumed);
05.         else
06.             mOut.setDataSize(0);
07.     }
08.     if (bwr.read_consumed > 0) {
09.         mIn.setDataSize(bwr.read_consumed);
10.         mIn.setDataPosition(0);
11.     }
12.     return NO_ERROR;
13. }
```

调整数据发送和接收缓存区，由于数据已经发送到Binder驱动中了，因此mOut中没有需要处理的数据了，然后继续返回到waitForResponse函数中

[cpp]

```
01. status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
02. {
03.     int32_t cmd;
04.     int32_t err;
05.     while (1) {
06.         if ((err=talkWithDriver()) < NO_ERROR) break;
07.         err = mIn.errorCheck();
08.         if (err < NO_ERROR) break;
09.         if (mIn.dataAvail() == 0) continue;
10.         cmd = mIn.readInt32();
11.         switch (cmd) {
12.         case BR_TRANSACTION_COMPLETE:
13.             if (!reply && !acquireResult) goto finish;
14.             break;
15.         default:
16.             err = executeCommand(cmd);
17.             if (err != NO_ERROR) goto finish;
18.             break;
19.         }
20.     }
21. finish:
22.     if (err != NO_ERROR) {
23.         if (acquireResult) *acquireResult = err;
24.         if (reply) reply->setError(err);
25.         mLastError = err;
```

```
    }
    return err;
}
```

该函数首先读取Binder驱动发送上来的BR_TRANSACTION_COMPLETE命令，如果是同步传输，则再次调用talkWithDriver()函数进入Binder驱动，等待服务进程返回函数远程调用的执行结果，如果是异步传输，则退出waitForResponse函数。由于前面从Binder驱动返回到talkWithDriver函数后，数据发送缓存区mOut的大小被设置为0

mOut.setDataSize(0)

因此此时传入到Binder驱动的数据为：

bwr.write_size = 0;
bwr.write_buffer = (long unsigned int)mOut.data();
bwr.read_size = mIn.dataCapacity();
bwr.read_buffer = (long unsigned int)mIn.data();

由于bwr.write_size = 0，因此此次通过ioctl调用binder_ioctl()函数只执行Binder读操作

```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;
    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.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;
}
```

对于binder_thread_read函数，这里不在详细分析，在执行过程中，当前Binder线程睡眠在当前进程或线程的等待队列中，等待服务进程执行完调用函数，并将执行结果返回。

服务进程接收函数调用请求

在Android应用程序启动Binder线程源码分析一文中介绍了，所有的Android应用程序在启动的时候都会通过joinThreadPool()函数来向Binder驱动注册一个Binder线程，等待接收客户端的请求，这就是为什么Android应用天然支持Binder进程间通信机制的原因。Android应用程序启动Binder线程源码分析介绍了应用程序向Binder驱动注册一个Binder线程，并且该线程在执行binder_thread_read函数时睡眠等待客户进程的请求，在前面客户进程发送RPC调用请求中介绍了客户进程向服务进程发送RPC远程函数调用请求，因此服务进程将被唤醒来执行服务函数，服务Binder线程被唤醒后，继续执行binder_thread_read函数接收客户进程发送过来的函数调用参数信息。

```
[cpp]
while (1) {
    uint32_t cmd;
    struct binder_transaction_data tr;
    struct binder_work *w;
    struct binder_transaction *t = NULL;

    if (!list_empty(&thread->todo))
```

```
        w = list_first_entry(&thread->todo, struct binder_work, entry);
    else if (!list_empty(&proc->todo) && wait_for_proc_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;

    switch (w->type) {
        case BINDER_WORK_TRANSACTION: {
            t = container_of(w, struct binder_transaction, work);
        } break;
    }

    if (!t)
        continue;

    BUG_ON(t->buffer == NULL);
    if (t->buffer->target_node) {
        struct binder_node *target_node = t->buffer->target_node;
        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 *));

    if (put_user(cmd, (uint32_t __user *)ptr))
        return -EFAULT;
    ptr += sizeof(uint32_t);
```

```
    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;
        thread->transaction_stack = t;
    } else {
        t->buffer->transaction = NULL;
        kfree(t);
        binder_stats_deleted(BINDER_STAT_TRANSACTION);
    }
    break;
}
```

首先查找当前线程或进程的todo队列是否为空，如果不为空，取出客户进程挂载在当前进程或线程todo队列中的事务项，然后根据事务项的类型分别处理，前面介绍到客户进程会将需要发送的数据封装成BINDER_WORK_TRANSACTION类型的事务工作项挂载到服务进程的todo队列中，因此这里取出该事务工作项后，从中取出事务项t，并根据该事务项t封装成另一个事务项tr，需要注意的是，在这里实现了Binder实体对象的寻址

```
[cpp]
if (t->buffer->target_node) {
    struct binder_node *target_node = t->buffer->target_node;
    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;
}
```

并向服务进程的用户空间发送BR_TRANSACTION命令以及将事务项tr的内容拷贝到服务进程的用户空间，然后服务Binder线程从binder_thread_read函数中返回的服务进程的用户空间中执行waitForResponse()函数

```
[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) {
            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switch分支中并没有对BR_TRANSACTION命令的处理，因此函数会调用executeCommand(cmd)函数来处理BR_TRANSACTION命令，处理过程如下：

```
[cpp]
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    RefBase::weakref_type* refs;
    status_t result = NO_ERROR;

    switch (cmd) {
        case BR_TRANSACTION:
            {
                binder_transaction_data tr;
                result = mIn.read(&tr, sizeof(tr));
                ALOG_ASSERT(result == NO_ERROR,"Not enough command data for brTRANSACTION");
                if (result != NO_ERROR) break;

                Parcel buffer;
                buffer.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);

                const pid_t origPid = mCallingPid;
                const uid_t origUid = mCallingUid;

                mCallingPid = tr.sender_pid;
                mCallingUid = tr.sender_euid;
                mOrigCallingUid = tr.sender_euid;

                int curPrio = getpriority(PRIO_PROCESS, mMyThreadId);
                if (gDisableBackgroundScheduling) {
                    if (curPrio > ANDROID_PRIORITY_NORMAL) {
                        setpriority(PRIO_PROCESS, mMyThreadId, ANDROID_PRIORITY_NORMAL);
                    }
                } else {
```

```
                    if (curPrio >= ANDROID_PRIORITY_BACKGROUND) {
                        set_sched_policy(mMyThreadId, SP_BACKGROUND);
                    }
                }
                Parcel reply;
                if (tr.target.ptr) {
                    sp<BBinder> b((BBinder*)tr.cookie);
                    const status_t error = b->transact(tr.code, buffer, &reply, tr.flags);
                    if (error < NO_ERROR) reply.setError(error);

                } else {
                    const status_t error = the_context_object->transact(tr.code, buffer, &reply, tr.flags);
                    if (error < NO_ERROR) reply.setError(error);
                }

                if ((tr.flags & TF_ONE_WAY) == 0) {
                    LOG_ONEWAY("Sending reply to %d!", mCallingPid);
                    sendReply(reply, 0);
                } else {
                    LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
                }

                mCallingPid = origPid;
                mCallingUid = origUid;
                mOrigCallingUid = origUid;
            }
            break;

        default:
            printf("*** BAD COMMAND %d received from Binder driver\n", cmd);
            result = UNKNOWN_ERROR;
            break;
    }

    if (result != NO_ERROR) {
        mLastError = result;
    }
    return result;
}
```

首先使用Parcel对象的ipcSetDataReference函数将binder_transaction_data结构体中的数据设置到Parcel对象buffer中，根据Binder本地对象地址得到请求服务的Binder本地对象BBinder，并调用服务Binder本地对象的transact函数将客户进程发送过来的函数调用信息传送到服务进程的上层，并且将函数执行结果reply发送会客户进程。

```
[cpp]

status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    data.setDataPosition(0);

    status_t err = NO_ERROR;
    switch (code) {
```

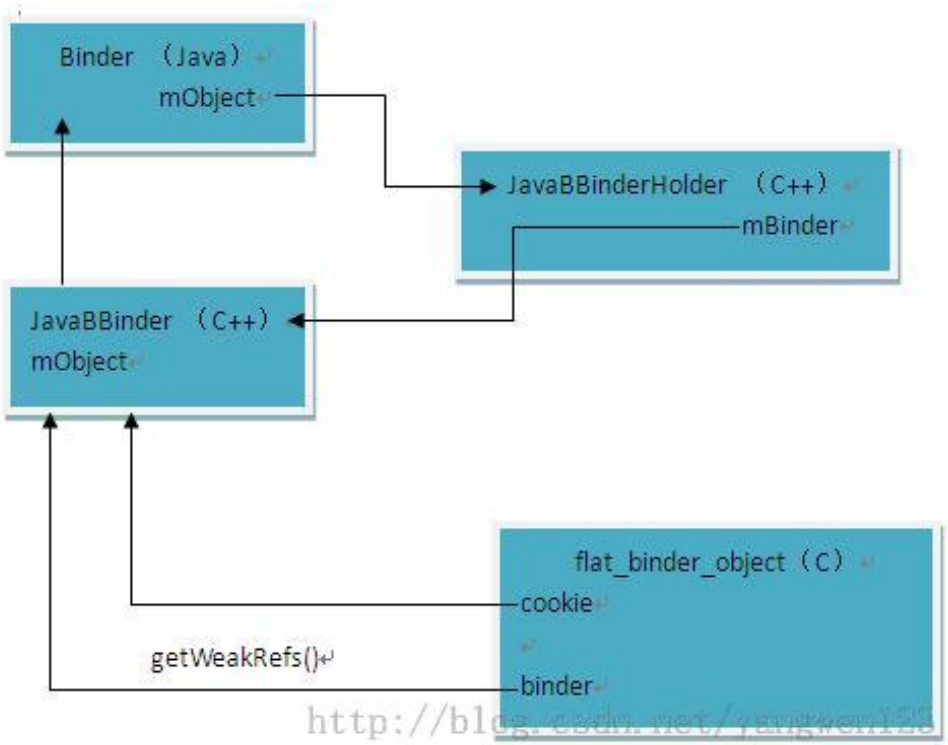
```
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
            err = onTransact(code, data, reply, flags);
            break;
    }

    if (reply != NULL) {
        reply->setDataPosition(0);
    }

    return err;
}
```

在switch分支中并没有对客户进程发送过来的函数调用码的处理，因此调用onTransact()函数进一步处理，由于在服务注册时，为服务创建的Binder本地对象是JavaBBinder，有关服务注册过程请查看Android服务注册完整过程源码分析，JavaBBinder是BBinder的子类，并且重写了BBinder的onTransact函数，因此BBinder的transact函数会调用BBinder子类JavaBBinder的onTransact()函数来执行客户端请求的函数调用。

```
[cpp]
virtual status_t onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0)
{
    JNIEnv* env = javavm_to_jnienv(mVM);

    ALOGV("onTransact() on %p calling object %p in env %p vm %p\n", this, mObject, env, mVM);

    IPCThreadState* thread_state = IPCThreadState::self();
    const int strict_policy_before = thread_state->getStrictModePolicy();
    thread_state->setLastTransactionBinderFlags(flags);

    jboolean res = env-
>CallBooleanMethod(mObject, gBinderOffsets.mExecTransact,code, (int32_t)&data, (int32_t)reply, fla
    jthrowable excep = env->ExceptionOccurred();

    if (excep) {
        report_exception(env, excep,"*** Uncaught remote exception!  "
            "(Exceptions are not yet supported across processes.)");
        res = JNI_FALSE;
        /* clean up JNI local ref -- we don't return to Java code */
        env->DeleteLocalRef(excep);
    }
    const int strict_policy_after = thread_state->getStrictModePolicy();
    if (strict_policy_after != strict_policy_before) {
        // Our thread-local...
        thread_state->setStrictModePolicy(strict_policy_before);
        // And the Java-level thread-local...
        set_dalvik_blockguard_policy(env, strict_policy_before);
    }
    jthrowable excep2 = env->ExceptionOccurred();
    if (excep2) {
        report_exception(env, excep2,
            "*** Uncaught exception in onBinderStrictModePolicyChange");
```

```
33.         /* clean up JNI local ref -- we don't return to Java code */
34.         env->DeleteLocalRef(excep2);
35.     }
36.     // Need to always call through the native implementation of
37.     // SYSPROPS_TRANSACTION.
38.     if (code == SYSPROPS_TRANSACTION) {
39.         BBinder::onTransact(code, data, reply, flags);
40.     }
41.     return res != JNI_FALSE ? NO_ERROR : UNKNOWN_TRANSACTION;
42. }
```

该函数通过JNI调用服务对应的Java层的Binder对象的execTransact函数，关于Java层的Binder对象与C++层的JavaBBinder对象之间的关系在Android 数据Parcel序列化过程源码分析中已经分析了



在构造Binder本地对象JavaBBinder对象时，创建了Java层Binder对象的全局引用对象，并保存在mObject中，在此通过mObject得到Java层的Binder对象，并调用其execTransact函数

```
[cpp]
01. private boolean execTransact(int code, int dataObj, int replyObj,
02.         int flags) {
03.     Parcel data = Parcel.obtain(dataObj);
04.     Parcel reply = Parcel.obtain(replyObj);
05.     // theoretically, we should call transact, which will call onTransact,
06.     // but all that does is rewind it, and we just got these from an IPC,
07.     // so we'll just call it directly.
08.     boolean res;
09.     try {
10.         res = onTransact(code, data, reply, flags);
11.     } catch (RemoteException e) {
12.         reply.setDataPosition(0);
13.         reply.writeException(e);
14.         res = true;
```

```
    } catch (RuntimeException e) {
        reply.setDataPosition(0);
        reply.writeException(e);
        res = true;
    } catch (OutOfMemoryError e) {
        RuntimeException re = new RuntimeException("Out of memory", e);
        reply.setDataPosition(0);
        reply.writeException(re);
        res = true;
    }
    reply.recycle();
    data.recycle();
    return res;
}
```

由于每个服务都是Java层Binder类的子类对象，并且都重写了Binder类的onTransact()方法，对于ActivityManager服务，其服务类是ActivityManagerNative

```
[cpp]
public abstract class ActivityManagerNative extends Binder implements IActivityManager
{
 public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
            throws RemoteException {
        switch (code) {
            ...
        }
    }
}
```

ActivityManagerNative类是一个抽象类，且是Binder的子类，证实了Java服务都是Binder类的子类对象，这一特性决定了Android服务进程都支持Binder进程间通信机制。ActivityManagerNative和客户进程使用的代理类ActivityManagerProxy都实现了IActivityManager接口，IActivityManager接口定义了客户端与服务端的业务接口函数，由于ActivityManagerNative类重写了父类Binder的onTransact函数，因此ActivityManagerNative的onTransact()函数会被调用执行

```
[cpp]
public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
        throws RemoteException {
    switch (code) {
        case GET_TASKS_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            int maxNum = data.readInt();
            int fl = data.readInt();
            IBinder receiverBinder = data.readStrongBinder();
            IThumbnailReceiver receiver = receiverBinder != null
                ? IThumbnailReceiver.Stub.asInterface(receiverBinder)
                : null;
            List list = getTasks(maxNum, fl, receiver);
            reply.writeNoException();
            int N = list != null ? list.size() : -1;
            reply.writeInt(N);
            int i;
```

```
17.             for (i=0; i<N; i++) {
18.                 ActivityManager.RunningTaskInfo info =
19.                         (ActivityManager.RunningTaskInfo)list.get(i);
20.                 info.writeToParcel(reply, 0);
21.             }
22.             return true;
23.         }
24.         ....
25.     }
26.     return super.onTransact(code, data, reply, flags);
27. }
```

该函数根据客户进程发送过来的函数调用码调用相应的执行函数，由于ActivityManagerNative是一个抽象类，它仅实现IActivityManager接口的部分函数，其余函数由其子类来实现，ActivityManagerService类是ActivityManagerNative的子类，实现了所有IActivityManager接口定义的函数，因此onTransact函数在执行函数调用码为GET_TASKS_TRANSACTION时，将调用之类ActivityManagerService的getTasks()函数来真正实现任务查询工作，最后将函数执行的结果通过相反的路径发送给客户进程，至此关于Android服务函数远程调用过程就分析完了。总结一下：

1) 客户进程中使用的服务代理类和服务进程中的服务类实现相同的接口，这样在客户端和服务端具有想匹配的调用函数；

2) 客户端通过服务代理类将指定函数的调用信息及调用码打包到Parcel对象中，并通过IPCThreadState发送到Binder驱动中；

3) Binder驱动根据服务代理对象的句柄值，在内核空间中找到为客户进程创建的Binder引用对象；

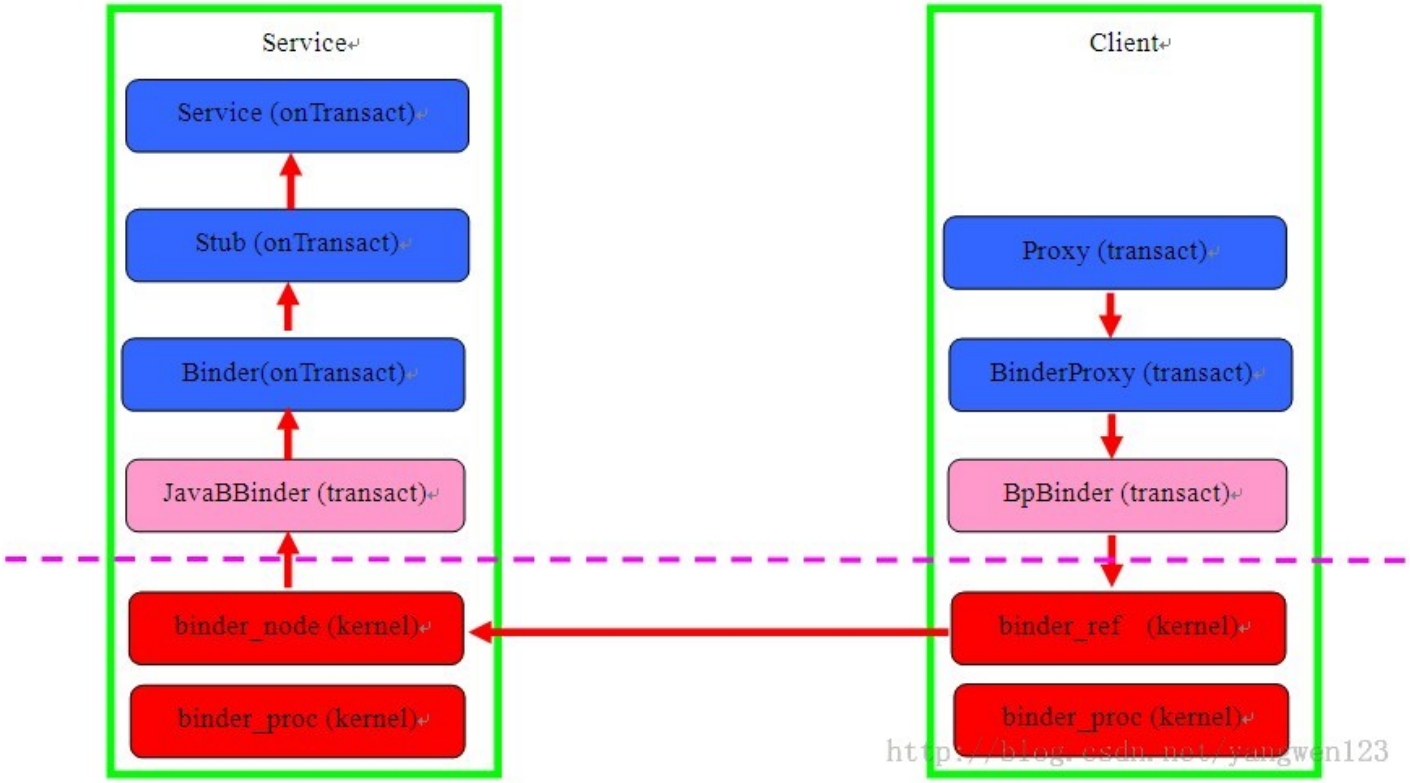
4) 根据Binder引用对象找到服务在内核空间中的Binder节点；

5) 通过Binder节点找到服务在服务进程的用户空间的Binder本地对象地址；

6) Binder驱动唤醒服务进程中的Binder线程，服务线程返回到用户空间执行Binder对象的transact函数；

7) 服务本地对象JavaBBinder通过JNI方式调用Java层的服务实现函数。

8) 服务进程将执行结构按照相反的路径返回给客户进程；


Service
Service (onTransact)
Stub (onTransact)
Binder(onTransact)
JavaBBinder (transact)
binder_node (kernel)
binder_proc (kernel)
Client
Proxy (transact)
BinderProxy (transact)
BpBinder (transact)
binder_ref (kernel)
binder_proc (kernel)
http://blog.csdn.net/yangwen123