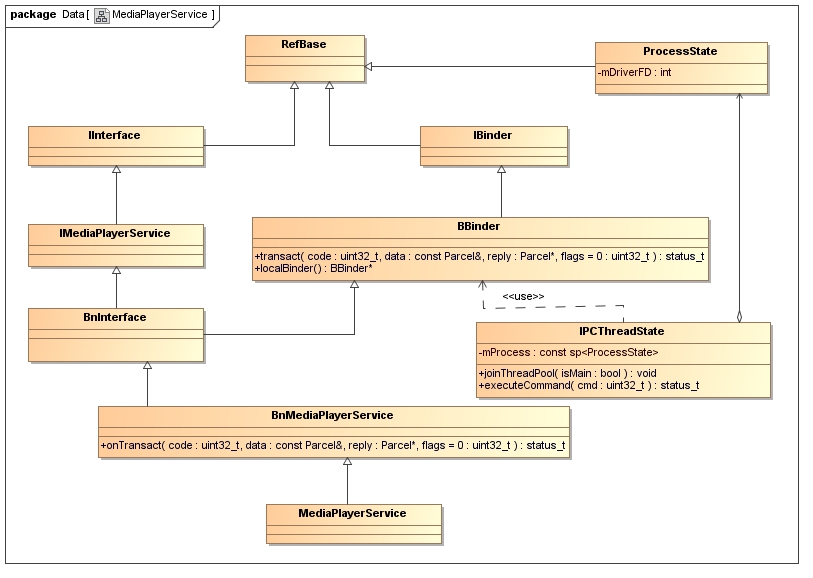
[Android系统进程间通信（IPC）机制Binder中的Server启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6629298)

在前面一篇文章[浅谈Android系统进程间通信（IPC）机制Binder中的Server和Client获得Service Manager接口之路](http://blog.csdn.net/luoshengyang/article/details/6627260)中，介绍了在Android系统中Binder进程间通信机制中的Server角色是如何获得Service Manager远程接口的，即defaultServiceManager函数的实现。Server获得了Service Manager远程接口之后，就要把自己的Service添加到Service Manager中去，然后把自己启动起来，等待Client的请求。本文将通过分析源代码了解Server的启动过程是怎么样的。

        本文通过一个具体的例子来说明Binder机制中Server的启动过程。我们知道，在Android系统中，提供了多媒体播放的功能，这个功能是以服务的形式来提供的。这里，我们就通过分析MediaPlayerService的实现来了解Media Server的启动过程。

        首先，看一下MediaPlayerService的类图，以便我们理解下面要描述的内容。



        我们将要介绍的主角MediaPlayerService继承于BnMediaPlayerService类，熟悉Binder机制的同学应该知道BnMediaPlayerService是一个Binder Native类，用来处理Client请求的。BnMediaPlayerService继承于BnInterface<IMediaPlayerService>类，BnInterface是一个模板类，它定义在frameworks/base/include/binder/IInterface.h文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **template**<**typename** INTERFACE>
2. **class** BnInterface : **public** INTERFACE, **public** BBinder
3. {
4. **public**:
5. **virtual** sp<IInterface>      queryLocalInterface(**const** String16& \_descriptor);
6. **virtual** **const** String16&     getInterfaceDescriptor() **const**;
8. **protected**:
9. **virtual** IBinder\*            onAsBinder();
10. };

       这里可以看出，BnMediaPlayerService实际是继承了IMediaPlayerService和BBinder类。IMediaPlayerService和BBinder类又分别继承了IInterface和IBinder类，IInterface和IBinder类又同时继承了RefBase类。

       实际上，BnMediaPlayerService并不是直接接收到Client处发送过来的请求，而是使用了IPCThreadState接收Client处发送过来的请求，而IPCThreadState又借助了ProcessState类来与Binder驱动程序交互。有关IPCThreadState和ProcessState的关系，可以参考上一篇文章[浅谈Android系统进程间通信（IPC）机制Binder中的Server和Client获得Service Manager接口之路](http://blog.csdn.net/luoshengyang/article/details/6627260)，接下来也会有相应的描述。IPCThreadState接收到了Client处的请求后，就会调用BBinder类的transact函数，并传入相关参数，BBinder类的transact函数最终调用BnMediaPlayerService类的onTransact函数，于是，就开始真正地处理Client的请求了。

      了解了MediaPlayerService类结构之后，就要开始进入到本文的主题了。

      首先，看看MediaPlayerService是如何启动的。启动MediaPlayerService的代码位于frameworks/base/media/mediaserver/main\_mediaserver.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **int** main(**int** argc, **char**\*\* argv)
2. {
3. sp<ProcessState> proc(ProcessState::self());
4. sp<IServiceManager> sm = defaultServiceManager();
5. LOGI("ServiceManager: %p", sm.get());
6. AudioFlinger::instantiate();
7. MediaPlayerService::instantiate();
8. CameraService::instantiate();
9. AudioPolicyService::instantiate();
10. ProcessState::self()->startThreadPool();
11. IPCThreadState::self()->joinThreadPool();
12. }

       这里我们不关注AudioFlinger和CameraService相关的代码。

       先看下面这句代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. sp<ProcessState> proc(ProcessState::self());

       这句代码的作用是通过ProcessState::self()调用创建一个ProcessState实例。ProcessState::self()是ProcessState类的一个静态成员变量，定义在frameworks/base/libs/binder/ProcessState.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. sp<ProcessState> ProcessState::self()
2. {
3. **if** (gProcess != NULL) **return** gProcess;
5. AutoMutex \_l(gProcessMutex);
6. **if** (gProcess == NULL) gProcess = **new** ProcessState;
7. **return** gProcess;
8. }

       这里可以看出，这个函数作用是返回一个全局唯一的ProcessState实例gProcess。全局唯一实例变量gProcess定义在frameworks/base/libs/binder/Static.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. Mutex gProcessMutex;
2. sp<ProcessState> gProcess;

       再来看ProcessState的构造函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. ProcessState::ProcessState()
2. : mDriverFD(open\_driver())
3. , mVMStart(MAP\_FAILED)
4. , mManagesContexts(**false**)
5. , mBinderContextCheckFunc(NULL)
6. , mBinderContextUserData(NULL)
7. , mThreadPoolStarted(**false**)
8. , mThreadPoolSeq(1)
9. {
10. **if** (mDriverFD >= 0) {
11. // XXX Ideally, there should be a specific define for whether we
12. // have mmap (or whether we could possibly have the kernel module
13. // availabla).
14. #if !defined(HAVE\_WIN32\_IPC)
15. // mmap the binder, providing a chunk of virtual address space to receive transactions.
16. mVMStart = mmap(0, BINDER\_VM\_SIZE, PROT\_READ, MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0);
17. **if** (mVMStart == MAP\_FAILED) {
18. // \*sigh\*
19. LOGE("Using /dev/binder failed: unable to mmap transaction memory.\n");
20. close(mDriverFD);
21. mDriverFD = -1;
22. }
23. #else
24. mDriverFD = -1;
25. #endif
26. }
27. **if** (mDriverFD < 0) {
28. // Need to run without the driver, starting our own thread pool.
29. }
30. }

        这个函数有两个关键地方，一是通过open\_driver函数打开Binder设备文件/dev/binder，并将打开设备文件描述符保存在成员变量mDriverFD中；二是通过mmap来把设备文件/dev/binder映射到内存中。

        先看open\_driver函数的实现，这个函数同样位于frameworks/base/libs/binder/ProcessState.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **int** open\_driver()
2. {
3. **if** (gSingleProcess) {
4. **return** -1;
5. }
7. **int** fd = open("/dev/binder", O\_RDWR);
8. **if** (fd >= 0) {
9. fcntl(fd, F\_SETFD, FD\_CLOEXEC);
10. **int** vers;
11. #if defined(HAVE\_ANDROID\_OS)
12. status\_t result = ioctl(fd, BINDER\_VERSION, &vers);
13. #else
14. status\_t result = -1;
15. errno = EPERM;
16. #endif
17. **if** (result == -1) {
18. LOGE("Binder ioctl to obtain version failed: %s", strerror(errno));
19. close(fd);
20. fd = -1;
21. }
22. **if** (result != 0 || vers != BINDER\_CURRENT\_PROTOCOL\_VERSION) {
23. LOGE("Binder driver protocol does not match user space protocol!");
24. close(fd);
25. fd = -1;
26. }
27. #if defined(HAVE\_ANDROID\_OS)
28. **size\_t** maxThreads = 15;
29. result = ioctl(fd, BINDER\_SET\_MAX\_THREADS, &maxThreads);
30. **if** (result == -1) {
31. LOGE("Binder ioctl to set max threads failed: %s", strerror(errno));
32. }
33. #endif
35. } **else** {
36. LOGW("Opening '/dev/binder' failed: %s\n", strerror(errno));
37. }
38. **return** fd;
39. }

        这个函数的作用主要是通过open文件操作函数来打开/dev/binder设备文件，然后再调用ioctl文件控制函数来分别执行BINDER\_VERSION和BINDER\_SET\_MAX\_THREADS两个命令来和Binder驱动程序进行交互，前者用于获得当前Binder驱动程序的版本号，后者用于通知Binder驱动程序，MediaPlayerService最多可同时启动15个线程来处理Client端的请求。

        open在Binder驱动程序中的具体实现，请参考前面一篇文章[浅谈Service Manager成为Android进程间通信（IPC）机制Binder守护进程之路](http://blog.csdn.net/luoshengyang/article/details/6621566)，这里不再重复描述。打开/dev/binder设备文件后，Binder驱动程序就为MediaPlayerService进程创建了一个struct binder\_proc结构体实例来维护MediaPlayerService进程上下文相关信息。

        我们来看一下ioctl文件操作函数执行BINDER\_VERSION命令的过程：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t result = ioctl(fd, BINDER\_VERSION, &vers);

        这个函数调用最终进入到Binder驱动程序的binder\_ioctl函数中，我们只关注BINDER\_VERSION相关的部分逻辑：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **long** binder\_ioctl(**struct** file \*filp, unsigned **int** cmd, unsigned **long** arg)
2. {
3. **int** ret;
4. **struct** binder\_proc \*proc = filp->private\_data;
5. **struct** binder\_thread \***thread**;
6. unsigned **int** size = \_IOC\_SIZE(cmd);
7. **void** \_\_user \*ubuf = (**void** \_\_user \*)arg;
9. /\*printk(KERN\_INFO "binder\_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);\*/
11. ret = wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);
12. **if** (ret)
13. **return** ret;
15. mutex\_lock(&binder\_lock);
16. **thread** = binder\_get\_thread(proc);
17. **if** (**thread** == NULL) {
18. ret = -ENOMEM;
19. **goto** err;
20. }
22. **switch** (cmd) {
23. ......
24. **case** BINDER\_VERSION:
25. **if** (size != **sizeof**(**struct** binder\_version)) {
26. ret = -EINVAL;
27. **goto** err;
28. }
29. **if** (put\_user(BINDER\_CURRENT\_PROTOCOL\_VERSION, &((**struct** binder\_version \*)ubuf)->protocol\_version)) {
30. ret = -EINVAL;
31. **goto** err;
32. }
33. **break**;
34. ......
35. }
36. ret = 0;
37. err:
38. ......
39. **return** ret;
40. }

        很简单，只是将BINDER\_CURRENT\_PROTOCOL\_VERSION写入到传入的参数arg指向的用户缓冲区中去就返回了。BINDER\_CURRENT\_PROTOCOL\_VERSION是一个宏，定义在kernel/common/drivers/staging/android/binder.h文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. /\* This is the current protocol version. \*/
2. #define BINDER\_CURRENT\_PROTOCOL\_VERSION 7

       这里为什么要把ubuf转换成struct binder\_version之后，再通过其protocol\_version成员变量再来写入呢，转了一圈，最终内容还是写入到ubuf中。我们看一下struct binder\_version的定义就会明白，同样是在kernel/common/drivers/staging/android/binder.h文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. /\* Use with BINDER\_VERSION, driver fills in fields. \*/
2. **struct** binder\_version {
3. /\* driver protocol version -- increment with incompatible change \*/
4. **signed** **long** protocol\_version;
5. };

        从注释中可以看出来，这里是考虑到兼容性，因为以后很有可能不是用signed long来表示版本号。

        这里有一个重要的地方要注意的是，由于这里是打开设备文件/dev/binder之后，第一次进入到binder\_ioctl函数，因此，这里调用binder\_get\_thread的时候，就会为当前线程创建一个struct binder\_thread结构体变量来维护线程上下文信息，具体可以参考[浅谈Service Manager成为Android进程间通信（IPC）机制Binder守护进程之路](http://blog.csdn.net/luoshengyang/article/details/6621566)一文。

        接着我们再来看一下ioctl文件操作函数执行BINDER\_SET\_MAX\_THREADS命令的过程：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. result = ioctl(fd, BINDER\_SET\_MAX\_THREADS, &maxThreads);

        这个函数调用最终进入到Binder驱动程序的binder\_ioctl函数中，我们只关注BINDER\_SET\_MAX\_THREADS相关的部分逻辑：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **long** binder\_ioctl(**struct** file \*filp, unsigned **int** cmd, unsigned **long** arg)
2. {
3. **int** ret;
4. **struct** binder\_proc \*proc = filp->private\_data;
5. **struct** binder\_thread \***thread**;
6. unsigned **int** size = \_IOC\_SIZE(cmd);
7. **void** \_\_user \*ubuf = (**void** \_\_user \*)arg;
9. /\*printk(KERN\_INFO "binder\_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);\*/
11. ret = wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);
12. **if** (ret)
13. **return** ret;
15. mutex\_lock(&binder\_lock);
16. **thread** = binder\_get\_thread(proc);
17. **if** (**thread** == NULL) {
18. ret = -ENOMEM;
19. **goto** err;
20. }
22. **switch** (cmd) {
23. ......
24. **case** BINDER\_SET\_MAX\_THREADS:
25. **if** (copy\_from\_user(&proc->max\_threads, ubuf, **sizeof**(proc->max\_threads))) {
26. ret = -EINVAL;
27. **goto** err;
28. }
29. **break**;
30. ......
31. }
32. ret = 0;
33. err:
34. ......
35. **return** ret;
36. }

        这里实现也是非常简单，只是简单地把用户传进来的参数保存在proc->max\_threads中就完毕了。注意，这里再调用binder\_get\_thread函数的时候，就可以在proc->threads中找到当前线程对应的struct binder\_thread结构了，因为前面已经创建好并保存在proc->threads红黑树中。

        回到ProcessState的构造函数中，这里还通过mmap函数来把设备文件/dev/binder映射到内存中，这个函数在[浅谈Service Manager成为Android进程间通信（IPC）机制Binder守护进程之路](http://blog.csdn.net/luoshengyang/article/details/6621566)一文也已经有详细介绍，这里不再重复描述。宏BINDER\_VM\_SIZE就定义在ProcessState.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. #define BINDER\_VM\_SIZE ((1\*1024\*1024) - (4096 \*2))

        mmap函数调用完成之后，Binder驱动程序就为当前进程预留了BINDER\_VM\_SIZE大小的内存空间了。

        这样，ProcessState全局唯一变量gProcess就创建完毕了，回到frameworks/base/media/mediaserver/main\_mediaserver.cpp文件中的main函数，下一步是调用defaultServiceManager函数来获得Service Manager的远程接口，这个已经在上一篇文章[浅谈Android系统进程间通信（IPC）机制Binder中的Server和Client获得Service Manager接口之路](http://blog.csdn.net/luoshengyang/article/details/6627260)有详细描述，读者可以回过头去参考一下。

        再接下来，就进入到MediaPlayerService::instantiate函数把MediaPlayerService添加到Service Manger中去了。这个函数定义在frameworks/base/media/libmediaplayerservice/MediaPlayerService.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **void** MediaPlayerService::instantiate() {
2. defaultServiceManager()->addService(
3. String16("media.player"), **new** MediaPlayerService());
4. }

        我们重点看一下IServiceManger::addService的过程，这有助于我们加深对Binder机制的理解。

        在上一篇文章[浅谈Android系统进程间通信（IPC）机制Binder中的Server和Client获得Service Manager接口之路](http://blog.csdn.net/luoshengyang/article/details/6627260)中说到，defaultServiceManager返回的实际是一个BpServiceManger类实例，因此，我们看一下BpServiceManger::addService的实现，这个函数实现在frameworks/base/libs/binder/IServiceManager.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **class** BpServiceManager : **public** BpInterface<IServiceManager>
2. {
3. **public**:
4. BpServiceManager(**const** sp<IBinder>& impl)
5. : BpInterface<IServiceManager>(impl)
6. {
7. }
9. ......
11. **virtual** status\_t addService(**const** String16& name, **const** sp<IBinder>& service)
12. {
13. Parcel data, reply;
14. data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
15. data.writeString16(name);
16. data.writeStrongBinder(service);
17. status\_t err = remote()->transact(ADD\_SERVICE\_TRANSACTION, data, &reply);
18. **return** err == NO\_ERROR ? reply.readExceptionCode()
19. }
21. ......
23. };

         这里的Parcel类是用来于序列化进程间通信数据用的。

         先来看这一句的调用：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());

         IServiceManager::getInterfaceDescriptor()返回来的是一个字符串，即"android.os.IServiceManager"，具体可以参考IServiceManger的实现。我们看一下Parcel::writeInterfaceToken的实现，位于frameworks/base/libs/binder/Parcel.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. // Write RPC headers.  (previously just the interface token)
2. status\_t Parcel::writeInterfaceToken(**const** String16& interface)
3. {
4. writeInt32(IPCThreadState::self()->getStrictModePolicy() |
5. STRICT\_MODE\_PENALTY\_GATHER);
6. // currently the interface identification token is just its name as a string
7. **return** writeString16(interface);
8. }

         它的作用是写入一个整数和一个字符串到Parcel中去。

         再来看下面的调用：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. data.writeString16(name);

        这里又是写入一个字符串到Parcel中去，这里的name即是上面传进来的“media.player”字符串。

        往下看：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. data.writeStrongBinder(service);

        这里定入一个Binder对象到Parcel去。我们重点看一下这个函数的实现，因为它涉及到进程间传输Binder实体的问题，比较复杂，需要重点关注，同时，也是理解Binder机制的一个重点所在。注意，这里的service参数是一个MediaPlayerService对象。

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t Parcel::writeStrongBinder(**const** sp<IBinder>& val)
2. {
3. **return** flatten\_binder(ProcessState::self(), val, **this**);
4. }

        看到flatten\_binder函数，是不是似曾相识的感觉？我们在前面一篇文章[浅谈Service Manager成为Android进程间通信（IPC）机制Binder守护进程之路](http://blog.csdn.net/luoshengyang/article/details/6621566)中，曾经提到在Binder驱动程序中，使用struct flat\_binder\_object来表示传输中的一个binder对象，它的定义如下所示：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. /\*
2. \* This is the flattened representation of a Binder object for transfer
3. \* between processes.  The 'offsets' supplied as part of a binder transaction
4. \* contains offsets into the data where these structures occur.  The Binder
5. \* driver takes care of re-writing the structure type and data as it moves
6. \* between processes.
7. \*/
8. **struct** flat\_binder\_object {
9. /\* 8 bytes for large\_flat\_header. \*/
10. unsigned **long**       type;
11. unsigned **long**       flags;
13. /\* 8 bytes of data. \*/
14. **union** {
15. **void**        \*binder;    /\* local object \*/
16. **signed** **long** handle;     /\* remote object \*/
17. };
19. /\* extra data associated with local object \*/
20. **void**            \*cookie;
21. };

        各个成员变量的含义请参考资料[Android Binder设计与实现](http://disanji.net/2011/02/28/android-bnder-design/)。

        我们进入到flatten\_binder函数看看：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t flatten\_binder(**const** sp<ProcessState>& proc,
2. **const** sp<IBinder>& binder, Parcel\* out)
3. {
4. flat\_binder\_object obj;
6. obj.flags = 0x7f | FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS;
7. **if** (binder != NULL) {
8. IBinder \*local = binder->localBinder();
9. **if** (!local) {
10. BpBinder \*proxy = binder->remoteBinder();
11. **if** (proxy == NULL) {
12. LOGE("null proxy");
13. }
14. **const** int32\_t handle = proxy ? proxy->handle() : 0;
15. obj.type = BINDER\_TYPE\_HANDLE;
16. obj.handle = handle;
17. obj.cookie = NULL;
18. } **else** {
19. obj.type = BINDER\_TYPE\_BINDER;
20. obj.binder = local->getWeakRefs();
21. obj.cookie = local;
22. }
23. } **else** {
24. obj.type = BINDER\_TYPE\_BINDER;
25. obj.binder = NULL;
26. obj.cookie = NULL;
27. }
29. **return** finish\_flatten\_binder(binder, obj, out);
30. }

        首先是初始化flat\_binder\_object的flags域：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. obj.flags = 0x7f | FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS;

        0x7f表示处理本Binder实体请求数据包的线程的最低优先级，FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS表示这个Binder实体可以接受文件描述符，Binder实体在收到文件描述符时，就会在本进程中打开这个文件。

       传进来的binder即为MediaPlayerService::instantiate函数中new出来的MediaPlayerService实例，因此，不为空。又由于MediaPlayerService继承自BBinder类，它是一个本地Binder实体，因此binder->localBinder返回一个BBinder指针，而且肯定不为空，于是执行下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. obj.type = BINDER\_TYPE\_BINDER;
2. obj.binder = local->getWeakRefs();
3. obj.cookie = local;

        设置了flat\_binder\_obj的其他成员变量，注意，指向这个Binder实体地址的指针local保存在flat\_binder\_obj的成员变量cookie中。

        函数调用finish\_flatten\_binder来将这个flat\_binder\_obj写入到Parcel中去：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **inline** **static** status\_t finish\_flatten\_binder(
2. **const** sp<IBinder>& binder, **const** flat\_binder\_object& flat, Parcel\* out)
3. {
4. **return** out->writeObject(flat, **false**);
5. }

       Parcel::writeObject的实现如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t Parcel::writeObject(**const** flat\_binder\_object& val, **bool** nullMetaData)
2. {
3. **const** **bool** enoughData = (mDataPos+**sizeof**(val)) <= mDataCapacity;
4. **const** **bool** enoughObjects = mObjectsSize < mObjectsCapacity;
5. **if** (enoughData && enoughObjects) {
6. restart\_write:
7. \***reinterpret\_cast**<flat\_binder\_object\*>(mData+mDataPos) = val;
9. // Need to write meta-data?
10. **if** (nullMetaData || val.binder != NULL) {
11. mObjects[mObjectsSize] = mDataPos;
12. acquire\_object(ProcessState::self(), val, **this**);
13. mObjectsSize++;
14. }
16. // remember if it's a file descriptor
17. **if** (val.type == BINDER\_TYPE\_FD) {
18. mHasFds = mFdsKnown = **true**;
19. }
21. **return** finishWrite(**sizeof**(flat\_binder\_object));
22. }
24. **if** (!enoughData) {
25. **const** status\_t err = growData(**sizeof**(val));
26. **if** (err != NO\_ERROR) **return** err;
27. }
28. **if** (!enoughObjects) {
29. **size\_t** newSize = ((mObjectsSize+2)\*3)/2;
30. **size\_t**\* objects = (**size\_t**\*)realloc(mObjects, newSize\***sizeof**(**size\_t**));
31. **if** (objects == NULL) **return** NO\_MEMORY;
32. mObjects = objects;
33. mObjectsCapacity = newSize;
34. }
36. **goto** restart\_write;
37. }

        这里除了把flat\_binder\_obj写到Parcel里面之内，还要记录这个flat\_binder\_obj在Parcel里面的偏移位置：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. mObjects[mObjectsSize] = mDataPos;

       这里因为，如果进程间传输的数据间带有Binder对象的时候，Binder驱动程序需要作进一步的处理，以维护各个Binder实体的一致性，下面我们将会看到Binder驱动程序是怎么处理这些Binder对象的。

       再回到BpServiceManager::addService函数中，调用下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t err = remote()->transact(ADD\_SERVICE\_TRANSACTION, data, &reply);

       回到[浅谈Android系统进程间通信（IPC）机制Binder中的Server和Client获得Service Manager接口之路](http://blog.csdn.net/luoshengyang/article/details/6627260)一文中的类图中去看一下，这里的remote成员函数来自于BpRefBase类，它返回一个BpBinder指针。因此，我们继续进入到BpBinder::transact函数中去看看：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t BpBinder::transact(
2. uint32\_t code, **const** Parcel& data, Parcel\* reply, uint32\_t flags)
3. {
4. // Once a binder has died, it will never come back to life.
5. **if** (mAlive) {
6. status\_t status = IPCThreadState::self()->transact(
7. mHandle, code, data, reply, flags);
8. **if** (status == DEAD\_OBJECT) mAlive = 0;
9. **return** status;
10. }
12. **return** DEAD\_OBJECT;
13. }

       这里又调用了IPCThreadState::transact进执行实际的操作。注意，这里的mHandle为0，code为ADD\_SERVICE\_TRANSACTION。ADD\_SERVICE\_TRANSACTION是上面以参数形式传进来的，那mHandle为什么是0呢？因为这里表示的是Service Manager远程接口，它的句柄值一定是0，具体请参考[浅谈Android系统进程间通信（IPC）机制Binder中的Server和Client获得Service Manager接口之路](http://blog.csdn.net/luoshengyang/article/details/6627260)一文。  
       再进入到IPCThreadState::transact函数，看看做了些什么事情：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t IPCThreadState::transact(int32\_t handle,
2. uint32\_t code, **const** Parcel& data,
3. Parcel\* reply, uint32\_t flags)
4. {
5. status\_t err = data.errorCheck();
7. flags |= TF\_ACCEPT\_FDS;
9. IF\_LOG\_TRANSACTIONS() {
10. TextOutput::Bundle \_b(alog);
11. alog << "BC\_TRANSACTION thr " << (**void**\*)pthread\_self() << " / hand "
12. << handle << " / code " << TypeCode(code) << ": "
13. << indent << data << dedent << endl;
14. }
16. **if** (err == NO\_ERROR) {
17. LOG\_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
18. (flags & TF\_ONE\_WAY) == 0 ? "READ REPLY" : "ONE WAY");
19. err = writeTransactionData(BC\_TRANSACTION, flags, handle, code, data, NULL);
20. }
22. **if** (err != NO\_ERROR) {
23. **if** (reply) reply->setError(err);
24. **return** (mLastError = err);
25. }
27. **if** ((flags & TF\_ONE\_WAY) == 0) {
28. #if 0
29. **if** (code == 4) { // relayout
30. LOGI(">>>>>> CALLING transaction 4");
31. } **else** {
32. LOGI(">>>>>> CALLING transaction %d", code);
33. }
34. #endif
35. **if** (reply) {
36. err = waitForResponse(reply);
37. } **else** {
38. Parcel fakeReply;
39. err = waitForResponse(&fakeReply);
40. }
41. #if 0
42. **if** (code == 4) { // relayout
43. LOGI("<<<<<< RETURNING transaction 4");
44. } **else** {
45. LOGI("<<<<<< RETURNING transaction %d", code);
46. }
47. #endif
49. IF\_LOG\_TRANSACTIONS() {
50. TextOutput::Bundle \_b(alog);
51. alog << "BR\_REPLY thr " << (**void**\*)pthread\_self() << " / hand "
52. << handle << ": ";
53. **if** (reply) alog << indent << \*reply << dedent << endl;
54. **else** alog << "(none requested)" << endl;
55. }
56. } **else** {
57. err = waitForResponse(NULL, NULL);
58. }
60. **return** err;
61. }

        IPCThreadState::transact函数的参数flags是一个默认值为0的参数，上面没有传相应的实参进来，因此，这里就为0。

        函数首先调用writeTransactionData函数准备好一个struct binder\_transaction\_data结构体变量，这个是等一下要传输给Binder驱动程序的。struct binder\_transaction\_data的定义我们在[浅谈Service Manager成为Android进程间通信（IPC）机制Binder守护进程之路](http://blog.csdn.net/luoshengyang/article/details/6621566)一文中有详细描述，读者不妨回过去读一下。这里为了方便描述，将struct binder\_transaction\_data的定义再次列出来：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **struct** binder\_transaction\_data {
2. /\* The first two are only used for bcTRANSACTION and brTRANSACTION,
3. \* identifying the target and contents of the transaction.
4. \*/
5. **union** {
6. **size\_t**  handle; /\* target descriptor of command transaction \*/
7. **void**    \*ptr;   /\* target descriptor of return transaction \*/
8. } target;
9. **void**        \*cookie;    /\* target object cookie \*/
10. unsigned **int**    code;       /\* transaction command \*/
12. /\* General information about the transaction. \*/
13. unsigned **int**    flags;
14. pid\_t       sender\_pid;
15. uid\_t       sender\_euid;
16. **size\_t**      data\_size;  /\* number of bytes of data \*/
17. **size\_t**      offsets\_size;   /\* number of bytes of offsets \*/
19. /\* If this transaction is inline, the data immediately
20. \* follows here; otherwise, it ends with a pointer to
21. \* the data buffer.
22. \*/
23. **union** {
24. **struct** {
25. /\* transaction data \*/
26. **const** **void**  \*buffer;
27. /\* offsets from buffer to flat\_binder\_object structs \*/
28. **const** **void**  \*offsets;
29. } ptr;
30. uint8\_t buf[8];
31. } data;
32. };

         writeTransactionData函数的实现如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t IPCThreadState::writeTransactionData(int32\_t cmd, uint32\_t binderFlags,
2. int32\_t handle, uint32\_t code, **const** Parcel& data, status\_t\* statusBuffer)
3. {
4. binder\_transaction\_data tr;
6. tr.target.handle = handle;
7. tr.code = code;
8. tr.flags = binderFlags;
10. **const** status\_t err = data.errorCheck();
11. **if** (err == NO\_ERROR) {
12. tr.data\_size = data.ipcDataSize();
13. tr.data.ptr.buffer = data.ipcData();
14. tr.offsets\_size = data.ipcObjectsCount()\***sizeof**(**size\_t**);
15. tr.data.ptr.offsets = data.ipcObjects();
16. } **else** **if** (statusBuffer) {
17. tr.flags |= TF\_STATUS\_CODE;
18. \*statusBuffer = err;
19. tr.data\_size = **sizeof**(status\_t);
20. tr.data.ptr.buffer = statusBuffer;
21. tr.offsets\_size = 0;
22. tr.data.ptr.offsets = NULL;
23. } **else** {
24. **return** (mLastError = err);
25. }
27. mOut.writeInt32(cmd);
28. mOut.write(&tr, **sizeof**(tr));
30. **return** NO\_ERROR;
31. }

        注意，这里的cmd为BC\_TRANSACTION。 这个函数很简单，在这个场景下，就是执行下面语句来初始化本地变量tr：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. tr.data\_size = data.ipcDataSize();
2. tr.data.ptr.buffer = data.ipcData();
3. tr.offsets\_size = data.ipcObjectsCount()\***sizeof**(**size\_t**);
4. tr.data.ptr.offsets = data.ipcObjects();

       回忆一下上面的内容，写入到tr.data.ptr.buffer的内容相当于下面的内容：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. writeInt32(IPCThreadState::self()->getStrictModePolicy() |
2. STRICT\_MODE\_PENALTY\_GATHER);
3. writeString16("android.os.IServiceManager");
4. writeString16("media.player");
5. writeStrongBinder(**new** MediaPlayerService());

       其中包含了一个Binder实体MediaPlayerService，因此需要设置tr.offsets\_size就为1，tr.data.ptr.offsets就指向了这个MediaPlayerService的地址在tr.data.ptr.buffer中的偏移量。最后，将tr的内容保存在IPCThreadState的成员变量mOut中。  
       回到IPCThreadState::transact函数中，接下去看，(flags & TF\_ONE\_WAY) == 0为true，并且reply不为空，所以最终进入到waitForResponse(reply)这条路径来。我们看一下waitForResponse函数的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t IPCThreadState::waitForResponse(Parcel \*reply, status\_t \*acquireResult)
2. {
3. int32\_t cmd;
4. int32\_t err;
6. **while** (1) {
7. **if** ((err=talkWithDriver()) < NO\_ERROR) **break**;
8. err = mIn.errorCheck();
9. **if** (err < NO\_ERROR) **break**;
10. **if** (mIn.dataAvail() == 0) **continue**;
12. cmd = mIn.readInt32();
14. IF\_LOG\_COMMANDS() {
15. alog << "Processing waitForResponse Command: "
16. << getReturnString(cmd) << endl;
17. }
19. **switch** (cmd) {
20. **case** BR\_TRANSACTION\_COMPLETE:
21. **if** (!reply && !acquireResult) **goto** finish;
22. **break**;
24. **case** BR\_DEAD\_REPLY:
25. err = DEAD\_OBJECT;
26. **goto** finish;
28. **case** BR\_FAILED\_REPLY:
29. err = FAILED\_TRANSACTION;
30. **goto** finish;
32. **case** BR\_ACQUIRE\_RESULT:
33. {
34. LOG\_ASSERT(acquireResult != NULL, "Unexpected brACQUIRE\_RESULT");
35. **const** int32\_t result = mIn.readInt32();
36. **if** (!acquireResult) **continue**;
37. \*acquireResult = result ? NO\_ERROR : INVALID\_OPERATION;
38. }
39. **goto** finish;
41. **case** BR\_REPLY:
42. {
43. binder\_transaction\_data tr;
44. err = mIn.read(&tr, **sizeof**(tr));
45. LOG\_ASSERT(err == NO\_ERROR, "Not enough command data for brREPLY");
46. **if** (err != NO\_ERROR) **goto** finish;
48. **if** (reply) {
49. **if** ((tr.flags & TF\_STATUS\_CODE) == 0) {
50. reply->ipcSetDataReference(
51. **reinterpret\_cast**<**const** uint8\_t\*>(tr.data.ptr.buffer),
52. tr.data\_size,
53. **reinterpret\_cast**<**const** **size\_t**\*>(tr.data.ptr.offsets),
54. tr.offsets\_size/**sizeof**(**size\_t**),
55. freeBuffer, **this**);
56. } **else** {
57. err = \***static\_cast**<**const** status\_t\*>(tr.data.ptr.buffer);
58. freeBuffer(NULL,
59. **reinterpret\_cast**<**const** uint8\_t\*>(tr.data.ptr.buffer),
60. tr.data\_size,
61. **reinterpret\_cast**<**const** **size\_t**\*>(tr.data.ptr.offsets),
62. tr.offsets\_size/**sizeof**(**size\_t**), **this**);
63. }
64. } **else** {
65. freeBuffer(NULL,
66. **reinterpret\_cast**<**const** uint8\_t\*>(tr.data.ptr.buffer),
67. tr.data\_size,
68. **reinterpret\_cast**<**const** **size\_t**\*>(tr.data.ptr.offsets),
69. tr.offsets\_size/**sizeof**(**size\_t**), **this**);
70. **continue**;
71. }
72. }
73. **goto** finish;
75. **default**:
76. err = executeCommand(cmd);
77. **if** (err != NO\_ERROR) **goto** finish;
78. **break**;
79. }
80. }
82. finish:
83. **if** (err != NO\_ERROR) {
84. **if** (acquireResult) \*acquireResult = err;
85. **if** (reply) reply->setError(err);
86. mLastError = err;
87. }
89. **return** err;
90. }

        这个函数虽然很长，但是主要调用了talkWithDriver函数来与Binder驱动程序进行交互：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t IPCThreadState::talkWithDriver(**bool** doReceive)
2. {
3. LOG\_ASSERT(mProcess->mDriverFD >= 0, "Binder driver is not opened");
5. binder\_write\_read bwr;
7. // Is the read buffer empty?
8. **const** **bool** needRead = mIn.dataPosition() >= mIn.dataSize();
10. // We don't want to write anything if we are still reading
11. // from data left in the input buffer and the caller
12. // has requested to read the next data.
13. **const** **size\_t** outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;
15. bwr.write\_size = outAvail;
16. bwr.write\_buffer = (**long** unsigned **int**)mOut.data();
18. // This is what we'll read.
19. **if** (doReceive && needRead) {
20. bwr.read\_size = mIn.dataCapacity();
21. bwr.read\_buffer = (**long** unsigned **int**)mIn.data();
22. } **else** {
23. bwr.read\_size = 0;
24. }
26. IF\_LOG\_COMMANDS() {
27. TextOutput::Bundle \_b(alog);
28. **if** (outAvail != 0) {
29. alog << "Sending commands to driver: " << indent;
30. **const** **void**\* cmds = (**const** **void**\*)bwr.write\_buffer;
31. **const** **void**\* end = ((**const** uint8\_t\*)cmds)+bwr.write\_size;
32. alog << HexDump(cmds, bwr.write\_size) << endl;
33. **while** (cmds < end) cmds = printCommand(alog, cmds);
34. alog << dedent;
35. }
36. alog << "Size of receive buffer: " << bwr.read\_size
37. << ", needRead: " << needRead << ", doReceive: " << doReceive << endl;
38. }
40. // Return immediately if there is nothing to do.
41. **if** ((bwr.write\_size == 0) && (bwr.read\_size == 0)) **return** NO\_ERROR;
43. bwr.write\_consumed = 0;
44. bwr.read\_consumed = 0;
45. status\_t err;
46. **do** {
47. IF\_LOG\_COMMANDS() {
48. alog << "About to read/write, write size = " << mOut.dataSize() << endl;
49. }
50. #if defined(HAVE\_ANDROID\_OS)
51. **if** (ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr) >= 0)
52. err = NO\_ERROR;
53. **else**
54. err = -errno;
55. #else
56. err = INVALID\_OPERATION;
57. #endif
58. IF\_LOG\_COMMANDS() {
59. alog << "Finished read/write, write size = " << mOut.dataSize() << endl;
60. }
61. } **while** (err == -EINTR);
63. IF\_LOG\_COMMANDS() {
64. alog << "Our err: " << (**void**\*)err << ", write consumed: "
65. << bwr.write\_consumed << " (of " << mOut.dataSize()
66. << "), read consumed: " << bwr.read\_consumed << endl;
67. }
69. **if** (err >= NO\_ERROR) {
70. **if** (bwr.write\_consumed > 0) {
71. **if** (bwr.write\_consumed < (ssize\_t)mOut.dataSize())
72. mOut.remove(0, bwr.write\_consumed);
73. **else**
74. mOut.setDataSize(0);
75. }
76. **if** (bwr.read\_consumed > 0) {
77. mIn.setDataSize(bwr.read\_consumed);
78. mIn.setDataPosition(0);
79. }
80. IF\_LOG\_COMMANDS() {
81. TextOutput::Bundle \_b(alog);
82. alog << "Remaining data size: " << mOut.dataSize() << endl;
83. alog << "Received commands from driver: " << indent;
84. **const** **void**\* cmds = mIn.data();
85. **const** **void**\* end = mIn.data() + mIn.dataSize();
86. alog << HexDump(cmds, mIn.dataSize()) << endl;
87. **while** (cmds < end) cmds = printReturnCommand(alog, cmds);
88. alog << dedent;
89. }
90. **return** NO\_ERROR;
91. }
93. **return** err;
94. }

        这里doReceive和needRead均为1，有兴趣的读者可以自已分析一下。因此，这里告诉Binder驱动程序，先执行write操作，再执行read操作，下面我们将会看到。

        最后，通过ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr)进行到Binder驱动程序的binder\_ioctl函数，我们只关注cmd为BINDER\_WRITE\_READ的逻辑：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **long** binder\_ioctl(**struct** file \*filp, unsigned **int** cmd, unsigned **long** arg)
2. {
3. **int** ret;
4. **struct** binder\_proc \*proc = filp->private\_data;
5. **struct** binder\_thread \***thread**;
6. unsigned **int** size = \_IOC\_SIZE(cmd);
7. **void** \_\_user \*ubuf = (**void** \_\_user \*)arg;
9. /\*printk(KERN\_INFO "binder\_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);\*/
11. ret = wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);
12. **if** (ret)
13. **return** ret;
15. mutex\_lock(&binder\_lock);
16. **thread** = binder\_get\_thread(proc);
17. **if** (**thread** == NULL) {
18. ret = -ENOMEM;
19. **goto** err;
20. }
22. **switch** (cmd) {
23. **case** BINDER\_WRITE\_READ: {
24. **struct** binder\_write\_read bwr;
25. **if** (size != **sizeof**(**struct** binder\_write\_read)) {
26. ret = -EINVAL;
27. **goto** err;
28. }
29. **if** (copy\_from\_user(&bwr, ubuf, **sizeof**(bwr))) {
30. ret = -EFAULT;
31. **goto** err;
32. }
33. **if** (binder\_debug\_mask & BINDER\_DEBUG\_READ\_WRITE)
34. printk(KERN\_INFO "binder: %d:%d write %ld at %08lx, read %ld at %08lx\n",
35. proc->pid, **thread**->pid, bwr.write\_size, bwr.write\_buffer, bwr.read\_size, bwr.read\_buffer);
36. **if** (bwr.write\_size > 0) {
37. ret = binder\_thread\_write(proc, **thread**, (**void** \_\_user \*)bwr.write\_buffer, bwr.write\_size, &bwr.write\_consumed);
38. **if** (ret < 0) {
39. bwr.read\_consumed = 0;
40. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr)))
41. ret = -EFAULT;
42. **goto** err;
43. }
44. }
45. **if** (bwr.read\_size > 0) {
46. ret = binder\_thread\_read(proc, **thread**, (**void** \_\_user \*)bwr.read\_buffer, bwr.read\_size, &bwr.read\_consumed, filp->f\_flags & O\_NONBLOCK);
47. **if** (!list\_empty(&proc->todo))
48. wake\_up\_interruptible(&proc->wait);
49. **if** (ret < 0) {
50. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr)))
51. ret = -EFAULT;
52. **goto** err;
53. }
54. }
55. **if** (binder\_debug\_mask & BINDER\_DEBUG\_READ\_WRITE)
56. printk(KERN\_INFO "binder: %d:%d wrote %ld of %ld, read return %ld of %ld\n",
57. proc->pid, **thread**->pid, bwr.write\_consumed, bwr.write\_size, bwr.read\_consumed, bwr.read\_size);
58. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr))) {
59. ret = -EFAULT;
60. **goto** err;
61. }
62. **break**;
63. }
64. ......
65. }
66. ret = 0;
67. err:
68. ......
69. **return** ret;
70. }

         函数首先是将用户传进来的参数拷贝到本地变量struct binder\_write\_read bwr中去。这里bwr.write\_size > 0为true，因此，进入到binder\_thread\_write函数中，我们只关注BC\_TRANSACTION部分的逻辑：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. binder\_thread\_write(**struct** binder\_proc \*proc, **struct** binder\_thread \***thread**,
2. **void** \_\_user \*buffer, **int** size, **signed** **long** \*consumed)
3. {
4. uint32\_t cmd;
5. **void** \_\_user \*ptr = buffer + \*consumed;
6. **void** \_\_user \*end = buffer + size;
8. **while** (ptr < end && **thread**->return\_error == BR\_OK) {
9. **if** (get\_user(cmd, (uint32\_t \_\_user \*)ptr))
10. **return** -EFAULT;
11. ptr += **sizeof**(uint32\_t);
12. **if** (\_IOC\_NR(cmd) < ARRAY\_SIZE(binder\_stats.bc)) {
13. binder\_stats.bc[\_IOC\_NR(cmd)]++;
14. proc->stats.bc[\_IOC\_NR(cmd)]++;
15. **thread**->stats.bc[\_IOC\_NR(cmd)]++;
16. }
17. **switch** (cmd) {
18. .....
19. **case** BC\_TRANSACTION:
20. **case** BC\_REPLY: {
21. **struct** binder\_transaction\_data tr;
23. **if** (copy\_from\_user(&tr, ptr, **sizeof**(tr)))
24. **return** -EFAULT;
25. ptr += **sizeof**(tr);
26. binder\_transaction(proc, **thread**, &tr, cmd == BC\_REPLY);
27. **break**;
28. }
29. ......
30. }
31. \*consumed = ptr - buffer;
32. }
33. **return** 0;
34. }

         首先将用户传进来的transact参数拷贝在本地变量struct binder\_transaction\_data tr中去，接着调用binder\_transaction函数进一步处理，这里我们忽略掉无关代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **void**
2. binder\_transaction(**struct** binder\_proc \*proc, **struct** binder\_thread \***thread**,
3. **struct** binder\_transaction\_data \*tr, **int** reply)
4. {
5. **struct** binder\_transaction \*t;
6. **struct** binder\_work \*tcomplete;
7. **size\_t** \*offp, \*off\_end;
8. **struct** binder\_proc \*target\_proc;
9. **struct** binder\_thread \*target\_thread = NULL;
10. **struct** binder\_node \*target\_node = NULL;
11. **struct** list\_head \*target\_list;
12. wait\_queue\_head\_t \*target\_wait;
13. **struct** binder\_transaction \*in\_reply\_to = NULL;
14. **struct** binder\_transaction\_log\_entry \*e;
15. uint32\_t return\_error;
17. ......
19. **if** (reply) {
20. ......
21. } **else** {
22. **if** (tr->target.handle) {
23. ......
24. } **else** {
25. target\_node = binder\_context\_mgr\_node;
26. **if** (target\_node == NULL) {
27. return\_error = BR\_DEAD\_REPLY;
28. **goto** err\_no\_context\_mgr\_node;
29. }
30. }
31. ......
32. target\_proc = target\_node->proc;
33. **if** (target\_proc == NULL) {
34. return\_error = BR\_DEAD\_REPLY;
35. **goto** err\_dead\_binder;
36. }
37. ......
38. }
39. **if** (target\_thread) {
40. ......
41. } **else** {
42. target\_list = &target\_proc->todo;
43. target\_wait = &target\_proc->wait;
44. }
46. ......
48. /\* TODO: reuse incoming transaction for reply \*/
49. t = kzalloc(**sizeof**(\*t), GFP\_KERNEL);
50. **if** (t == NULL) {
51. return\_error = BR\_FAILED\_REPLY;
52. **goto** err\_alloc\_t\_failed;
53. }
54. ......
56. tcomplete = kzalloc(**sizeof**(\*tcomplete), GFP\_KERNEL);
57. **if** (tcomplete == NULL) {
58. return\_error = BR\_FAILED\_REPLY;
59. **goto** err\_alloc\_tcomplete\_failed;
60. }
62. ......
64. **if** (!reply && !(tr->flags & TF\_ONE\_WAY))
65. t->from = **thread**;
66. **else**
67. t->from = NULL;
68. t->sender\_euid = proc->tsk->cred->euid;
69. t->to\_proc = target\_proc;
70. t->to\_thread = target\_thread;
71. t->code = tr->code;
72. t->flags = tr->flags;
73. t->priority = task\_nice(current);
74. t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,
75. tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));
76. **if** (t->buffer == NULL) {
77. return\_error = BR\_FAILED\_REPLY;
78. **goto** err\_binder\_alloc\_buf\_failed;
79. }
80. t->buffer->allow\_user\_free = 0;
81. t->buffer->debug\_id = t->debug\_id;
82. t->buffer->transaction = t;
83. t->buffer->target\_node = target\_node;
84. **if** (target\_node)
85. binder\_inc\_node(target\_node, 1, 0, NULL);
87. offp = (**size\_t** \*)(t->buffer->data + ALIGN(tr->data\_size, **sizeof**(**void** \*)));
89. **if** (copy\_from\_user(t->buffer->data, tr->data.ptr.buffer, tr->data\_size)) {
90. ......
91. return\_error = BR\_FAILED\_REPLY;
92. **goto** err\_copy\_data\_failed;
93. }
94. **if** (copy\_from\_user(offp, tr->data.ptr.offsets, tr->offsets\_size)) {
95. ......
96. return\_error = BR\_FAILED\_REPLY;
97. **goto** err\_copy\_data\_failed;
98. }
99. ......
101. off\_end = (**void** \*)offp + tr->offsets\_size;
102. **for** (; offp < off\_end; offp++) {
103. **struct** flat\_binder\_object \*fp;
104. ......
105. fp = (**struct** flat\_binder\_object \*)(t->buffer->data + \*offp);
106. **switch** (fp->type) {
107. **case** BINDER\_TYPE\_BINDER:
108. **case** BINDER\_TYPE\_WEAK\_BINDER: {
109. **struct** binder\_ref \*ref;
110. **struct** binder\_node \*node = binder\_get\_node(proc, fp->binder);
111. **if** (node == NULL) {
112. node = binder\_new\_node(proc, fp->binder, fp->cookie);
113. **if** (node == NULL) {
114. return\_error = BR\_FAILED\_REPLY;
115. **goto** err\_binder\_new\_node\_failed;
116. }
117. node->min\_priority = fp->flags & FLAT\_BINDER\_FLAG\_PRIORITY\_MASK;
118. node->accept\_fds = !!(fp->flags & FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS);
119. }
120. **if** (fp->cookie != node->cookie) {
121. ......
122. **goto** err\_binder\_get\_ref\_for\_node\_failed;
123. }
124. ref = binder\_get\_ref\_for\_node(target\_proc, node);
125. **if** (ref == NULL) {
126. return\_error = BR\_FAILED\_REPLY;
127. **goto** err\_binder\_get\_ref\_for\_node\_failed;
128. }
129. **if** (fp->type == BINDER\_TYPE\_BINDER)
130. fp->type = BINDER\_TYPE\_HANDLE;
131. **else**
132. fp->type = BINDER\_TYPE\_WEAK\_HANDLE;
133. fp->handle = ref->desc;
134. binder\_inc\_ref(ref, fp->type == BINDER\_TYPE\_HANDLE, &**thread**->todo);
135. ......
137. } **break**;
138. ......
139. }
140. }
142. **if** (reply) {
143. ......
144. } **else** **if** (!(t->flags & TF\_ONE\_WAY)) {
145. BUG\_ON(t->buffer->async\_transaction != 0);
146. t->need\_reply = 1;
147. t->from\_parent = **thread**->transaction\_stack;
148. **thread**->transaction\_stack = t;
149. } **else** {
150. ......
151. }
152. t->work.type = BINDER\_WORK\_TRANSACTION;
153. list\_add\_tail(&t->work.entry, target\_list);
154. tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;
155. list\_add\_tail(&tcomplete->entry, &**thread**->todo);
156. **if** (target\_wait)
157. wake\_up\_interruptible(target\_wait);
158. **return**;
159. ......
160. }

       注意，这里传进来的参数reply为0，tr->target.handle也为0。因此，target\_proc、target\_thread、target\_node、target\_list和target\_wait的值分别为：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. target\_node = binder\_context\_mgr\_node;
2. target\_proc = target\_node->proc;
3. target\_list = &target\_proc->todo;
4. target\_wait = &target\_proc->wait;

       接着，分配了一个待处理事务t和一个待完成工作项tcomplete，并执行初始化工作：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. /\* TODO: reuse incoming transaction for reply \*/
2. t = kzalloc(**sizeof**(\*t), GFP\_KERNEL);
3. **if** (t == NULL) {
4. return\_error = BR\_FAILED\_REPLY;
5. **goto** err\_alloc\_t\_failed;
6. }
7. ......
9. tcomplete = kzalloc(**sizeof**(\*tcomplete), GFP\_KERNEL);
10. **if** (tcomplete == NULL) {
11. return\_error = BR\_FAILED\_REPLY;
12. **goto** err\_alloc\_tcomplete\_failed;
13. }
15. ......
17. **if** (!reply && !(tr->flags & TF\_ONE\_WAY))
18. t->from = **thread**;
19. **else**
20. t->from = NULL;
21. t->sender\_euid = proc->tsk->cred->euid;
22. t->to\_proc = target\_proc;
23. t->to\_thread = target\_thread;
24. t->code = tr->code;
25. t->flags = tr->flags;
26. t->priority = task\_nice(current);
27. t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,
28. tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));
29. **if** (t->buffer == NULL) {
30. return\_error = BR\_FAILED\_REPLY;
31. **goto** err\_binder\_alloc\_buf\_failed;
32. }
33. t->buffer->allow\_user\_free = 0;
34. t->buffer->debug\_id = t->debug\_id;
35. t->buffer->transaction = t;
36. t->buffer->target\_node = target\_node;
37. **if** (target\_node)
38. binder\_inc\_node(target\_node, 1, 0, NULL);
40. offp = (**size\_t** \*)(t->buffer->data + ALIGN(tr->data\_size, **sizeof**(**void** \*)));
42. **if** (copy\_from\_user(t->buffer->data, tr->data.ptr.buffer, tr->data\_size)) {
43. ......
44. return\_error = BR\_FAILED\_REPLY;
45. **goto** err\_copy\_data\_failed;
46. }
47. **if** (copy\_from\_user(offp, tr->data.ptr.offsets, tr->offsets\_size)) {
48. ......
49. return\_error = BR\_FAILED\_REPLY;
50. **goto** err\_copy\_data\_failed;
51. }

         注意，这里的事务t是要交给target\_proc处理的，在这个场景之下，就是Service Manager了。因此，下面的语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,
2. tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));

         就是在Service Manager的进程空间中分配一块内存来保存用户传进入的参数了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (copy\_from\_user(t->buffer->data, tr->data.ptr.buffer, tr->data\_size)) {
2. ......
3. return\_error = BR\_FAILED\_REPLY;
4. **goto** err\_copy\_data\_failed;
5. }
6. **if** (copy\_from\_user(offp, tr->data.ptr.offsets, tr->offsets\_size)) {
7. ......
8. return\_error = BR\_FAILED\_REPLY;
9. **goto** err\_copy\_data\_failed;
10. }

         由于现在target\_node要被使用了，增加它的引用计数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (target\_node)
2. binder\_inc\_node(target\_node, 1, 0, NULL);

        接下去的for循环，就是用来处理传输数据中的Binder对象了。在我们的场景中，有一个类型为BINDER\_TYPE\_BINDER的Binder实体MediaPlayerService：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **switch** (fp->type) {
2. **case** BINDER\_TYPE\_BINDER:
3. **case** BINDER\_TYPE\_WEAK\_BINDER: {
4. **struct** binder\_ref \*ref;
5. **struct** binder\_node \*node = binder\_get\_node(proc, fp->binder);
6. **if** (node == NULL) {
7. node = binder\_new\_node(proc, fp->binder, fp->cookie);
8. **if** (node == NULL) {
9. return\_error = BR\_FAILED\_REPLY;
10. **goto** err\_binder\_new\_node\_failed;
11. }
12. node->min\_priority = fp->flags & FLAT\_BINDER\_FLAG\_PRIORITY\_MASK;
13. node->accept\_fds = !!(fp->flags & FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS);
14. }
15. **if** (fp->cookie != node->cookie) {
16. ......
17. **goto** err\_binder\_get\_ref\_for\_node\_failed;
18. }
19. ref = binder\_get\_ref\_for\_node(target\_proc, node);
20. **if** (ref == NULL) {
21. return\_error = BR\_FAILED\_REPLY;
22. **goto** err\_binder\_get\_ref\_for\_node\_failed;
23. }
24. **if** (fp->type == BINDER\_TYPE\_BINDER)
25. fp->type = BINDER\_TYPE\_HANDLE;
26. **else**
27. fp->type = BINDER\_TYPE\_WEAK\_HANDLE;
28. fp->handle = ref->desc;
29. binder\_inc\_ref(ref, fp->type == BINDER\_TYPE\_HANDLE, &**thread**->todo);
30. ......
32. } **break**;

        由于是第一次在Binder驱动程序中传输这个MediaPlayerService，调用binder\_get\_node函数查询这个Binder实体时，会返回空，于是binder\_new\_node在proc中新建一个，下次就可以直接使用了。

        现在，由于要把这个Binder实体MediaPlayerService交给target\_proc，也就是Service Manager来管理，也就是说Service Manager要引用这个MediaPlayerService了，于是通过binder\_get\_ref\_for\_node为MediaPlayerService创建一个引用，并且通过binder\_inc\_ref来增加这个引用计数，防止这个引用还在使用过程当中就被销毁。注意，到了这里的时候，t->buffer中的flat\_binder\_obj的type已经改为BINDER\_TYPE\_HANDLE，handle已经改为ref->desc，跟原来不一样了，因为这个flat\_binder\_obj是最终是要传给Service Manager的，而Service Manager只能够通过句柄值来引用这个Binder实体。

        最后，把待处理事务加入到target\_list列表中去：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. list\_add\_tail(&t->work.entry, target\_list);

        并且把待完成工作项加入到本线程的todo等待执行列表中去：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. list\_add\_tail(&tcomplete->entry, &**thread**->todo);

        现在目标进程有事情可做了，于是唤醒它：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (target\_wait)
2. wake\_up\_interruptible(target\_wait);

       这里就是要唤醒Service Manager进程了。回忆一下前面[浅谈Service Manager成为Android进程间通信（IPC）机制Binder守护进程之路](http://blog.csdn.net/luoshengyang/article/details/6621566)这篇文章，此时， Service Manager正在binder\_thread\_read函数中调用wait\_event\_interruptible进入休眠状态。

       这里我们先忽略一下Service Manager被唤醒之后的场景，继续MedaPlayerService的启动过程，然后再回来。

       回到binder\_ioctl函数，bwr.read\_size > 0为true，于是进入binder\_thread\_read函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **int**
2. binder\_thread\_read(**struct** binder\_proc \*proc, **struct** binder\_thread \***thread**,
3. **void**  \_\_user \*buffer, **int** size, **signed** **long** \*consumed, **int** non\_block)
4. {
5. **void** \_\_user \*ptr = buffer + \*consumed;
6. **void** \_\_user \*end = buffer + size;
8. **int** ret = 0;
9. **int** wait\_for\_proc\_work;
11. **if** (\*consumed == 0) {
12. **if** (put\_user(BR\_NOOP, (uint32\_t \_\_user \*)ptr))
13. **return** -EFAULT;
14. ptr += **sizeof**(uint32\_t);
15. }
17. retry:
18. wait\_for\_proc\_work = **thread**->transaction\_stack == NULL && list\_empty(&**thread**->todo);
20. .......
22. **if** (wait\_for\_proc\_work) {
23. .......
24. } **else** {
25. **if** (non\_block) {
26. **if** (!binder\_has\_thread\_work(**thread**))
27. ret = -EAGAIN;
28. } **else**
29. ret = wait\_event\_interruptible(**thread**->wait, binder\_has\_thread\_work(**thread**));
30. }
32. ......
34. **while** (1) {
35. uint32\_t cmd;
36. **struct** binder\_transaction\_data tr;
37. **struct** binder\_work \*w;
38. **struct** binder\_transaction \*t = NULL;
40. **if** (!list\_empty(&**thread**->todo))
41. w = list\_first\_entry(&**thread**->todo, **struct** binder\_work, entry);
42. **else** **if** (!list\_empty(&proc->todo) && wait\_for\_proc\_work)
43. w = list\_first\_entry(&proc->todo, **struct** binder\_work, entry);
44. **else** {
45. **if** (ptr - buffer == 4 && !(**thread**->looper & BINDER\_LOOPER\_STATE\_NEED\_RETURN)) /\* no data added \*/
46. **goto** retry;
47. **break**;
48. }
50. **if** (end - ptr < **sizeof**(tr) + 4)
51. **break**;
53. **switch** (w->type) {
54. ......
55. **case** BINDER\_WORK\_TRANSACTION\_COMPLETE: {
56. cmd = BR\_TRANSACTION\_COMPLETE;
57. **if** (put\_user(cmd, (uint32\_t \_\_user \*)ptr))
58. **return** -EFAULT;
59. ptr += **sizeof**(uint32\_t);
61. binder\_stat\_br(proc, **thread**, cmd);
62. **if** (binder\_debug\_mask & BINDER\_DEBUG\_TRANSACTION\_COMPLETE)
63. printk(KERN\_INFO "binder: %d:%d BR\_TRANSACTION\_COMPLETE\n",
64. proc->pid, **thread**->pid);
66. list\_del(&w->entry);
67. kfree(w);
68. binder\_stats.obj\_deleted[BINDER\_STAT\_TRANSACTION\_COMPLETE]++;
69. } **break**;
70. ......
71. }
73. **if** (!t)
74. **continue**;
76. ......
77. }
79. done:
80. ......
81. **return** 0;
82. }

        这里，thread->transaction\_stack和thread->todo均不为空，于是wait\_for\_proc\_work为false，由于binder\_has\_thread\_work的时候，返回true，这里因为thread->todo不为空，因此，线程虽然调用了wait\_event\_interruptible，但是不会睡眠，于是继续往下执行。

        由于thread->todo不为空，执行下列语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (!list\_empty(&**thread**->todo))
2. w = list\_first\_entry(&**thread**->todo, **struct** binder\_work, entry);

        w->type为BINDER\_WORK\_TRANSACTION\_COMPLETE，这是在上面的binder\_transaction函数设置的，于是执行：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **switch** (w->type) {
2. ......
3. **case** BINDER\_WORK\_TRANSACTION\_COMPLETE: {
4. cmd = BR\_TRANSACTION\_COMPLETE;
5. **if** (put\_user(cmd, (uint32\_t \_\_user \*)ptr))
6. **return** -EFAULT;
7. ptr += **sizeof**(uint32\_t);
9. ......
10. list\_del(&w->entry);
11. kfree(w);
13. } **break**;
14. ......
15. }

        这里就将w从thread->todo删除了。由于这里t为空，重新执行while循环，这时由于已经没有事情可做了，最后就返回到binder\_ioctl函数中。注间，这里一共往用户传进来的缓冲区buffer写入了两个整数，分别是BR\_NOOP和BR\_TRANSACTION\_COMPLETE。

        binder\_ioctl函数返回到用户空间之前，把数据消耗情况拷贝回用户空间中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr))) {
2. ret = -EFAULT;
3. **goto** err;
4. }

        最后返回到IPCThreadState::talkWithDriver函数中，执行下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (err >= NO\_ERROR) {
2. **if** (bwr.write\_consumed > 0) {
3. **if** (bwr.write\_consumed < (ssize\_t)mOut.dataSize())
4. mOut.remove(0, bwr.write\_consumed);
5. **else**
6. mOut.setDataSize(0);
7. }
8. **if** (bwr.read\_consumed > 0) {
9. <pre name="code" **class**="cpp">            mIn.setDataSize(bwr.read\_consumed);
10. mIn.setDataPosition(0);</pre>        }        ......        **return** NO\_ERROR;    }

        首先是把mOut的数据清空：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. mOut.setDataSize(0);

        然后设置已经读取的内容的大小：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. mIn.setDataSize(bwr.read\_consumed);
2. mIn.setDataPosition(0);

        然后返回到IPCThreadState::waitForResponse函数中。在IPCThreadState::waitForResponse函数，先是从mIn读出一个整数，这个便是BR\_NOOP了，这是一个空操作，什么也不做。然后继续进入IPCThreadState::talkWithDriver函数中。  
        这时候，下面语句执行后：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **const** **bool** needRead = mIn.dataPosition() >= mIn.dataSize();

        needRead为false，因为在mIn中，尚有一个整数BR\_TRANSACTION\_COMPLETE未读出。

       这时候，下面语句执行后：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **const** **size\_t** outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;

        outAvail等于0。因此，最后bwr.write\_size和bwr.read\_size均为0，IPCThreadState::talkWithDriver函数什么也不做，直接返回到IPCThreadState::waitForResponse函数中。在IPCThreadState::waitForResponse函数，又继续从mIn读出一个整数，这个便是BR\_TRANSACTION\_COMPLETE：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **switch** (cmd) {
2. **case** BR\_TRANSACTION\_COMPLETE:
3. **if** (!reply && !acquireResult) **goto** finish;
4. **break**;
5. ......
6. }

        reply不为NULL，因此，IPCThreadState::waitForResponse的循环没有结束，继续执行，又进入到IPCThreadState::talkWithDrive中。

        这次，needRead就为true了，而outAvail仍为0，所以bwr.read\_size不为0，bwr.write\_size为0。于是通过：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr)

        进入到Binder驱动程序中的binder\_ioctl函数中。由于bwr.write\_size为0，bwr.read\_size不为0，这次直接就进入到binder\_thread\_read函数中。这时候，thread->transaction\_stack不等于0，但是thread->todo为空，于是线程就通过：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. wait\_event\_interruptible(**thread**->wait, binder\_has\_thread\_work(**thread**));

        进入睡眠状态，等待Service Manager来唤醒了。

        现在，我们可以回到Service Manager被唤醒的过程了。我们接着前面[浅谈Service Manager成为Android进程间通信（IPC）机制Binder守护进程之路](http://blog.csdn.net/luoshengyang/article/details/6621566)这篇文章的最后，继续描述。此时， Service Manager正在binder\_thread\_read函数中调用wait\_event\_interruptible\_exclusive进入休眠状态。上面被MediaPlayerService启动后进程唤醒后，继续执行binder\_thread\_read函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **int**
2. binder\_thread\_read(**struct** binder\_proc \*proc, **struct** binder\_thread \***thread**,
3. **void**  \_\_user \*buffer, **int** size, **signed** **long** \*consumed, **int** non\_block)
4. {
5. **void** \_\_user \*ptr = buffer + \*consumed;
6. **void** \_\_user \*end = buffer + size;
8. **int** ret = 0;
9. **int** wait\_for\_proc\_work;
11. **if** (\*consumed == 0) {
12. **if** (put\_user(BR\_NOOP, (uint32\_t \_\_user \*)ptr))
13. **return** -EFAULT;
14. ptr += **sizeof**(uint32\_t);
15. }
17. retry:
18. wait\_for\_proc\_work = **thread**->transaction\_stack == NULL && list\_empty(&**thread**->todo);
20. ......
22. **if** (wait\_for\_proc\_work) {
23. ......
24. **if** (non\_block) {
25. **if** (!binder\_has\_proc\_work(proc, **thread**))
26. ret = -EAGAIN;
27. } **else**
28. ret = wait\_event\_interruptible\_exclusive(proc->wait, binder\_has\_proc\_work(proc, **thread**));
29. } **else** {
30. ......
31. }
33. ......
35. **while** (1) {
36. uint32\_t cmd;
37. **struct** binder\_transaction\_data tr;
38. **struct** binder\_work \*w;
39. **struct** binder\_transaction \*t = NULL;
41. **if** (!list\_empty(&**thread**->todo))
42. w = list\_first\_entry(&**thread**->todo, **struct** binder\_work, entry);
43. **else** **if** (!list\_empty(&proc->todo) && wait\_for\_proc\_work)
44. w = list\_first\_entry(&proc->todo, **struct** binder\_work, entry);
45. **else** {
46. **if** (ptr - buffer == 4 && !(**thread**->looper & BINDER\_LOOPER\_STATE\_NEED\_RETURN)) /\* no data added \*/
47. **goto** retry;
48. **break**;
49. }
51. **if** (end - ptr < **sizeof**(tr) + 4)
52. **break**;
54. **switch** (w->type) {
55. **case** BINDER\_WORK\_TRANSACTION: {
56. t = container\_of(w, **struct** binder\_transaction, work);
57. } **break**;
58. ......
59. }
61. **if** (!t)
62. **continue**;
64. BUG\_ON(t->buffer == NULL);
65. **if** (t->buffer->target\_node) {
66. **struct** binder\_node \*target\_node = t->buffer->target\_node;
67. tr.target.ptr = target\_node->ptr;
68. tr.cookie =  target\_node->cookie;
69. ......
70. cmd = BR\_TRANSACTION;
71. } **else** {
72. ......
73. }
74. tr.code = t->code;
75. tr.flags = t->flags;
76. tr.sender\_euid = t->sender\_euid;
78. **if** (t->from) {
79. **struct** task\_struct \*sender = t->from->proc->tsk;
80. tr.sender\_pid = task\_tgid\_nr\_ns(sender, current->nsproxy->pid\_ns);
81. } **else** {
82. tr.sender\_pid = 0;
83. }
85. tr.data\_size = t->buffer->data\_size;
86. tr.offsets\_size = t->buffer->offsets\_size;
87. tr.data.ptr.buffer = (**void** \*)t->buffer->data + proc->user\_buffer\_offset;
88. tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data\_size, **sizeof**(**void** \*));
90. **if** (put\_user(cmd, (uint32\_t \_\_user \*)ptr))
91. **return** -EFAULT;
92. ptr += **sizeof**(uint32\_t);
93. **if** (copy\_to\_user(ptr, &tr, **sizeof**(tr)))
94. **return** -EFAULT;
95. ptr += **sizeof**(tr);
97. ......
99. list\_del(&t->work.entry);
100. t->buffer->allow\_user\_free = 1;
101. **if** (cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)) {
102. t->to\_parent = **thread**->transaction\_stack;
103. t->to\_thread = **thread**;
104. **thread**->transaction\_stack = t;
105. } **else** {
106. t->buffer->transaction = NULL;
107. kfree(t);
108. binder\_stats.obj\_deleted[BINDER\_STAT\_TRANSACTION]++;
109. }
110. **break**;
111. }
113. done:
115. ......
116. **return** 0;
117. }

        Service Manager被唤醒之后，就进入while循环开始处理事务了。这里wait\_for\_proc\_work等于1，并且proc->todo不为空，所以从proc->todo列表中得到第一个工作项：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. w = list\_first\_entry(&proc->todo, **struct** binder\_work, entry);

        从上面的描述中，我们知道，这个工作项的类型为BINDER\_WORK\_TRANSACTION，于是通过下面语句得到事务项：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. t = container\_of(w, **struct** binder\_transaction, work);

       接着就是把事务项t中的数据拷贝到本地局部变量struct binder\_transaction\_data tr中去了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (t->buffer->target\_node) {
2. **struct** binder\_node \*target\_node = t->buffer->target\_node;
3. tr.target.ptr = target\_node->ptr;
4. tr.cookie =  target\_node->cookie;
5. ......
6. cmd = BR\_TRANSACTION;
7. } **else** {
8. ......
9. }
10. tr.code = t->code;
11. tr.flags = t->flags;
12. tr.sender\_euid = t->sender\_euid;
14. **if** (t->from) {
15. **struct** task\_struct \*sender = t->from->proc->tsk;
16. tr.sender\_pid = task\_tgid\_nr\_ns(sender, current->nsproxy->pid\_ns);
17. } **else** {
18. tr.sender\_pid = 0;
19. }
21. tr.data\_size = t->buffer->data\_size;
22. tr.offsets\_size = t->buffer->offsets\_size;
23. tr.data.ptr.buffer = (**void** \*)t->buffer->data + proc->user\_buffer\_offset;
24. tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data\_size, **sizeof**(**void** \*));

        这里有一个非常重要的地方，是Binder进程间通信机制的精髓所在：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. tr.data.ptr.buffer = (**void** \*)t->buffer->data + proc->user\_buffer\_offset;
2. tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data\_size, **sizeof**(**void** \*));

        t->buffer->data所指向的地址是内核空间的，现在要把数据返回给Service Manager进程的用户空间，而Service Manager进程的用户空间是不能访问内核空间的数据的，所以这里要作一下处理。怎么处理呢？我们在学面向对象语言的时候，对象的拷贝有深拷贝和浅拷贝之分，深拷贝是把另外分配一块新内存，然后把原始对象的内容搬过去，浅拷贝是并没有为新对象分配一块新空间，而只是分配一个引用，而个引用指向原始对象。Binder机制用的是类似浅拷贝的方法，通过在用户空间分配一个虚拟地址，然后让这个用户空间虚拟地址与 t->buffer->data这个内核空间虚拟地址指向同一个物理地址，这样就可以实现浅拷贝了。怎么样用户空间和内核空间的虚拟地址同时指向同一个物理地址呢？请参考前面一篇文章[浅谈Service Manager成为Android进程间通信（IPC）机制Binder守护进程之路](http://blog.csdn.net/luoshengyang/article/details/6621566)，那里有详细描述。这里只要将t->buffer->data加上一个偏移值proc->user\_buffer\_offset就可以得到t->buffer->data对应的用户空间虚拟地址了。调整了tr.data.ptr.buffer的值之后，不要忘记也要一起调整tr.data.ptr.offsets的值。

        接着就是把tr的内容拷贝到用户传进来的缓冲区去了，指针ptr指向这个用户缓冲区的地址：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (put\_user(cmd, (uint32\_t \_\_user \*)ptr))
2. **return** -EFAULT;
3. ptr += **sizeof**(uint32\_t);
4. **if** (copy\_to\_user(ptr, &tr, **sizeof**(tr)))
5. **return** -EFAULT;
6. ptr += **sizeof**(tr);

         这里可以看出，这里只是对作tr.data.ptr.bufferr和tr.data.ptr.offsets的内容作了浅拷贝。

         最后，由于已经处理了这个事务，要把它从todo列表中删除：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. list\_del(&t->work.entry);
2. t->buffer->allow\_user\_free = 1;
3. **if** (cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)) {
4. t->to\_parent = **thread**->transaction\_stack;
5. t->to\_thread = **thread**;
6. **thread**->transaction\_stack = t;
7. } **else** {
8. t->buffer->transaction = NULL;
9. kfree(t);
10. binder\_stats.obj\_deleted[BINDER\_STAT\_TRANSACTION]++;
11. }

         注意，这里的cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)为true，表明这个事务虽然在驱动程序中已经处理完了，但是它仍然要等待Service Manager完成之后，给驱动程序一个确认，也就是需要等待回复，于是把当前事务t放在thread->transaction\_stack队列的头部：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. t->to\_parent = **thread**->transaction\_stack;
2. t->to\_thread = **thread**;
3. **thread**->transaction\_stack = t;

         如果cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)为false，那就不需要等待回复了，直接把事务t删掉。

         这个while最后通过一个break跳了出来，最后返回到binder\_ioctl函数中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **long** binder\_ioctl(**struct** file \*filp, unsigned **int** cmd, unsigned **long** arg)
2. {
3. **int** ret;
4. **struct** binder\_proc \*proc = filp->private\_data;
5. **struct** binder\_thread \***thread**;
6. unsigned **int** size = \_IOC\_SIZE(cmd);
7. **void** \_\_user \*ubuf = (**void** \_\_user \*)arg;
9. ......
11. **switch** (cmd) {
12. **case** BINDER\_WRITE\_READ: {
13. **struct** binder\_write\_read bwr;
14. **if** (size != **sizeof**(**struct** binder\_write\_read)) {
15. ret = -EINVAL;
16. **goto** err;
17. }
18. **if** (copy\_from\_user(&bwr, ubuf, **sizeof**(bwr))) {
19. ret = -EFAULT;
20. **goto** err;
21. }
22. ......
23. **if** (bwr.read\_size > 0) {
24. ret = binder\_thread\_read(proc, **thread**, (**void** \_\_user \*)bwr.read\_buffer, bwr.read\_size, &bwr.read\_consumed, filp->f\_flags & O\_NONBLOCK);
25. **if** (!list\_empty(&proc->todo))
26. wake\_up\_interruptible(&proc->wait);
27. **if** (ret < 0) {
28. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr)))
29. ret = -EFAULT;
30. **goto** err;
31. }
32. }
33. ......
34. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr))) {
35. ret = -EFAULT;
36. **goto** err;
37. }
38. **break**;
39. }
40. ......
41. **default**:
42. ret = -EINVAL;
43. **goto** err;
44. }
45. ret = 0;
46. err:
47. ......
48. **return** ret;
49. }

         从binder\_thread\_read返回来后，再看看proc->todo是否还有事务等待处理，如果是，就把睡眠在proc->wait队列的线程唤醒来处理。最后，把本地变量struct binder\_write\_read bwr的内容拷贝回到用户传进来的缓冲区中，就返回了。

        这里就是返回到frameworks/base/cmds/servicemanager/binder.c文件中的binder\_loop函数了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **void** binder\_loop(**struct** binder\_state \*bs, binder\_handler func)
2. {
3. **int** res;
4. **struct** binder\_write\_read bwr;
5. unsigned readbuf[32];
7. bwr.write\_size = 0;
8. bwr.write\_consumed = 0;
9. bwr.write\_buffer = 0;
11. readbuf[0] = BC\_ENTER\_LOOPER;
12. binder\_write(bs, readbuf, **sizeof**(unsigned));
14. **for** (;;) {
15. bwr.read\_size = **sizeof**(readbuf);
16. bwr.read\_consumed = 0;
17. bwr.read\_buffer = (unsigned) readbuf;
19. res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);
21. **if** (res < 0) {
22. LOGE("binder\_loop: ioctl failed (%s)\n", strerror(errno));
23. **break**;
24. }
26. res = binder\_parse(bs, 0, readbuf, bwr.read\_consumed, func);
27. **if** (res == 0) {
28. LOGE("binder\_loop: unexpected reply?!\n");
29. **break**;
30. }
31. **if** (res < 0) {
32. LOGE("binder\_loop: io error %d %s\n", res, strerror(errno));
33. **break**;
34. }
35. }
36. }

       返回来的数据都放在readbuf中，接着调用binder\_parse进行解析：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **int** binder\_parse(**struct** binder\_state \*bs, **struct** binder\_io \*bio,
2. uint32\_t \*ptr, uint32\_t size, binder\_handler func)
3. {
4. **int** r = 1;
5. uint32\_t \*end = ptr + (size / 4);
7. **while** (ptr < end) {
8. uint32\_t cmd = \*ptr++;
9. ......
10. **case** BR\_TRANSACTION: {
11. **struct** binder\_txn \*txn = (**void** \*) ptr;
12. **if** ((end - ptr) \* **sizeof**(uint32\_t) < **sizeof**(**struct** binder\_txn)) {
13. LOGE("parse: txn too small!\n");
14. **return** -1;
15. }
16. binder\_dump\_txn(txn);
17. **if** (func) {
18. unsigned rdata[256/4];
19. **struct** binder\_io msg;
20. **struct** binder\_io reply;
21. **int** res;
23. bio\_init(&reply, rdata, **sizeof**(rdata), 4);
24. bio\_init\_from\_txn(&msg, txn);
25. res = func(bs, txn, &msg, &reply);
26. binder\_send\_reply(bs, &reply, txn->data, res);
27. }
28. ptr += **sizeof**(\*txn) / **sizeof**(uint32\_t);
29. **break**;
30. }
31. ......
32. **default**:
33. LOGE("parse: OOPS %d\n", cmd);
34. **return** -1;
35. }
36. }
38. **return** r;
39. }

        首先把从Binder驱动程序读出来的数据转换为一个struct binder\_txn结构体，保存在txn本地变量中，struct binder\_txn定义在frameworks/base/cmds/servicemanager/binder.h文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **struct** binder\_txn
2. {
3. **void** \*target;
4. **void** \*cookie;
5. uint32\_t code;
6. uint32\_t flags;
8. uint32\_t sender\_pid;
9. uint32\_t sender\_euid;
11. uint32\_t data\_size;
12. uint32\_t offs\_size;
13. **void** \*data;
14. **void** \*offs;
15. };

       函数中还用到了另外一个数据结构struct binder\_io，也是定义在frameworks/base/cmds/servicemanager/binder.h文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **struct** binder\_io
2. {
3. **char** \*data;            /\* pointer to read/write from \*/
4. uint32\_t \*offs;        /\* array of offsets \*/
5. uint32\_t data\_avail;   /\* bytes available in data buffer \*/
6. uint32\_t offs\_avail;   /\* entries available in offsets array \*/
8. **char** \*data0;           /\* start of data buffer \*/
9. uint32\_t \*offs0;       /\* start of offsets buffer \*/
10. uint32\_t flags;
11. uint32\_t unused;
12. };

       接着往下看，函数调bio\_init来初始化reply变量：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **void** bio\_init(**struct** binder\_io \*bio, **void** \*data,
2. uint32\_t maxdata, uint32\_t maxoffs)
3. {
4. uint32\_t n = maxoffs \* **sizeof**(uint32\_t);
6. **if** (n > maxdata) {
7. bio->flags = BIO\_F\_OVERFLOW;
8. bio->data\_avail = 0;
9. bio->offs\_avail = 0;
10. **return**;
11. }
13. bio->data = bio->data0 = data + n;
14. bio->offs = bio->offs0 = data;
15. bio->data\_avail = maxdata - n;
16. bio->offs\_avail = maxoffs;
17. bio->flags = 0;
18. }

       接着又调用bio\_init\_from\_txn来初始化msg变量：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **void** bio\_init\_from\_txn(**struct** binder\_io \*bio, **struct** binder\_txn \*txn)
2. {
3. bio->data = bio->data0 = txn->data;
4. bio->offs = bio->offs0 = txn->offs;
5. bio->data\_avail = txn->data\_size;
6. bio->offs\_avail = txn->offs\_size / 4;
7. bio->flags = BIO\_F\_SHARED;
8. }

      最后，真正进行处理的函数是从参数中传进来的函数指针func，这里就是定义在frameworks/base/cmds/servicemanager/service\_manager.c文件中的svcmgr\_handler函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **int** svcmgr\_handler(**struct** binder\_state \*bs,
2. **struct** binder\_txn \*txn,
3. **struct** binder\_io \*msg,
4. **struct** binder\_io \*reply)
5. {
6. **struct** svcinfo \*si;
7. uint16\_t \*s;
8. unsigned len;
9. **void** \*ptr;
10. uint32\_t strict\_policy;
12. **if** (txn->target != svcmgr\_handle)
13. **return** -1;
15. // Equivalent to Parcel::enforceInterface(), reading the RPC
16. // header with the strict mode policy mask and the interface name.
17. // Note that we ignore the strict\_policy and don't propagate it
18. // further (since we do no outbound RPCs anyway).
19. strict\_policy = bio\_get\_uint32(msg);
20. s = bio\_get\_string16(msg, &len);
21. **if** ((len != (**sizeof**(svcmgr\_id) / 2)) ||
22. memcmp(svcmgr\_id, s, **sizeof**(svcmgr\_id))) {
23. fprintf(stderr,"invalid id %s\n", str8(s));
24. **return** -1;
25. }
27. **switch**(txn->code) {
28. ......
29. **case** SVC\_MGR\_ADD\_SERVICE:
30. s = bio\_get\_string16(msg, &len);
31. ptr = bio\_get\_ref(msg);
32. **if** (do\_add\_service(bs, s, len, ptr, txn->sender\_euid))
33. **return** -1;
34. **break**;
35. ......
36. }
38. bio\_put\_uint32(reply, 0);
39. **return** 0;
40. }

         回忆一下，在BpServiceManager::addService时，传给Binder驱动程序的参数为：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. writeInt32(IPCThreadState::self()->getStrictModePolicy() | STRICT\_MODE\_PENALTY\_GATHER);
2. writeString16("android.os.IServiceManager");
3. writeString16("media.player");
4. writeStrongBinder(**new** MediaPlayerService());

         这里的语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. strict\_policy = bio\_get\_uint32(msg);
2. s = bio\_get\_string16(msg, &len);
3. s = bio\_get\_string16(msg, &len);
4. ptr = bio\_get\_ref(msg);

         就是依次把它们读取出来了，这里，我们只要看一下bio\_get\_ref的实现。先看一个数据结构struct binder\_obj的定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **struct** binder\_object
2. {
3. uint32\_t type;
4. uint32\_t flags;
5. **void** \*pointer;
6. **void** \*cookie;
7. };

        这个结构体其实就是对应struct flat\_binder\_obj的。

        接着看bio\_get\_ref实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **void** \*bio\_get\_ref(**struct** binder\_io \*bio)
2. {
3. **struct** binder\_object \*obj;
5. obj = \_bio\_get\_obj(bio);
6. **if** (!obj)
7. **return** 0;
9. **if** (obj->type == BINDER\_TYPE\_HANDLE)
10. **return** obj->pointer;
12. **return** 0;
13. }

       \_bio\_get\_obj这个函数就不跟进去看了，它的作用就是从binder\_io中取得第一个还没取获取过的binder\_object。在这个场景下，就是我们最开始传过来代表MediaPlayerService的flat\_binder\_obj了，这个原始的flat\_binder\_obj的type为BINDER\_TYPE\_BINDER，binder为指向MediaPlayerService的弱引用的地址。在前面我们说过，在Binder驱动驱动程序里面，会把这个flat\_binder\_obj的type改为BINDER\_TYPE\_HANDLE，handle改为一个句柄值。这里的handle值就等于obj->pointer的值。

        回到svcmgr\_handler函数，调用do\_add\_service进一步处理：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **int** do\_add\_service(**struct** binder\_state \*bs,
2. uint16\_t \*s, unsigned len,
3. **void** \*ptr, unsigned uid)
4. {
5. **struct** svcinfo \*si;
6. //    LOGI("add\_service('%s',%p) uid=%d\n", str8(s), ptr, uid);
8. **if** (!ptr || (len == 0) || (len > 127))
9. **return** -1;
11. **if** (!svc\_can\_register(uid, s)) {
12. LOGE("add\_service('%s',%p) uid=%d - PERMISSION DENIED\n",
13. str8(s), ptr, uid);
14. **return** -1;
15. }
17. si = find\_svc(s, len);
18. **if** (si) {
19. **if** (si->ptr) {
20. LOGE("add\_service('%s',%p) uid=%d - ALREADY REGISTERED\n",
21. str8(s), ptr, uid);
22. **return** -1;
23. }
24. si->ptr = ptr;
25. } **else** {
26. si = malloc(**sizeof**(\*si) + (len + 1) \* **sizeof**(uint16\_t));
27. **if** (!si) {
28. LOGE("add\_service('%s',%p) uid=%d - OUT OF MEMORY\n",
29. str8(s), ptr, uid);
30. **return** -1;
31. }
32. si->ptr = ptr;
33. si->len = len;
34. memcpy(si->name, s, (len + 1) \* **sizeof**(uint16\_t));
35. si->name[len] = '\0';
36. si->death.func = svcinfo\_death;
37. si->death.ptr = si;
38. si->next = svclist;
39. svclist = si;
40. }
42. binder\_acquire(bs, ptr);
43. binder\_link\_to\_death(bs, ptr, &si->death);
44. **return** 0;
45. }

        这个函数的实现很简单，就是把MediaPlayerService这个Binder实体的引用写到一个struct svcinfo结构体中，主要是它的名称和句柄值，然后插入到链接svclist的头部去。这样，Client来向Service Manager查询服务接口时，只要给定服务名称，Service Manger就可以返回相应的句柄值了。

        这个函数执行完成后，返回到svcmgr\_handler函数，函数的最后，将一个错误码0写到reply变量中去，表示一切正常：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. bio\_put\_uint32(reply, 0);

       svcmgr\_handler函数执行完成后，返回到binder\_parse函数，执行下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. binder\_send\_reply(bs, &reply, txn->data, res);

       我们看一下binder\_send\_reply的实现，从函数名就可以猜到它要做什么了，告诉Binder驱动程序，它完成了Binder驱动程序交给它的任务了。

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **void** binder\_send\_reply(**struct** binder\_state \*bs,
2. **struct** binder\_io \*reply,
3. **void** \*buffer\_to\_free,
4. **int** status)
5. {
6. **struct** {
7. uint32\_t cmd\_free;
8. **void** \*buffer;
9. uint32\_t cmd\_reply;
10. **struct** binder\_txn txn;
11. } \_\_attribute\_\_((packed)) data;
13. data.cmd\_free = BC\_FREE\_BUFFER;
14. data.buffer = buffer\_to\_free;
15. data.cmd\_reply = BC\_REPLY;
16. data.txn.target = 0;
17. data.txn.cookie = 0;
18. data.txn.code = 0;
19. **if** (status) {
20. data.txn.flags = TF\_STATUS\_CODE;
21. data.txn.data\_size = **sizeof**(**int**);
22. data.txn.offs\_size = 0;
23. data.txn.data = &status;
24. data.txn.offs = 0;
25. } **else** {
26. data.txn.flags = 0;
27. data.txn.data\_size = reply->data - reply->data0;
28. data.txn.offs\_size = ((**char**\*) reply->offs) - ((**char**\*) reply->offs0);
29. data.txn.data = reply->data0;
30. data.txn.offs = reply->offs0;
31. }
32. binder\_write(bs, &data, **sizeof**(data));
33. }

       从这里可以看出，binder\_send\_reply告诉Binder驱动程序执行BC\_FREE\_BUFFER和BC\_REPLY命令，前者释放之前在binder\_transaction分配的空间，地址为buffer\_to\_free，buffer\_to\_free这个地址是Binder驱动程序把自己在内核空间用的地址转换成用户空间地址再传给Service Manager的，所以Binder驱动程序拿到这个地址后，知道怎么样释放这个空间；后者告诉MediaPlayerService，它的addService操作已经完成了，错误码是0，保存在data.txn.data中。

       再来看binder\_write函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **int** binder\_write(**struct** binder\_state \*bs, **void** \*data, unsigned len)
2. {
3. **struct** binder\_write\_read bwr;
4. **int** res;
5. bwr.write\_size = len;
6. bwr.write\_consumed = 0;
7. bwr.write\_buffer = (unsigned) data;
8. bwr.read\_size = 0;
9. bwr.read\_consumed = 0;
10. bwr.read\_buffer = 0;
11. res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);
12. **if** (res < 0) {
13. fprintf(stderr,"binder\_write: ioctl failed (%s)\n",
14. strerror(errno));
15. }
16. **return** res;
17. }

       这里可以看出，只有写操作，没有读操作，即read\_size为0。

       这里又是一个ioctl的BINDER\_WRITE\_READ操作。直入到驱动程序的binder\_ioctl函数后，执行BINDER\_WRITE\_READ命令，这里就不累述了。

       最后，从binder\_ioctl执行到binder\_thread\_write函数，我们首先看第一个命令BC\_FREE\_BUFFER：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **int**
2. binder\_thread\_write(**struct** binder\_proc \*proc, **struct** binder\_thread \***thread**,
3. **void** \_\_user \*buffer, **int** size, **signed** **long** \*consumed)
4. {
5. uint32\_t cmd;
6. **void** \_\_user \*ptr = buffer + \*consumed;
7. **void** \_\_user \*end = buffer + size;
9. **while** (ptr < end && **thread**->return\_error == BR\_OK) {
10. **if** (get\_user(cmd, (uint32\_t \_\_user \*)ptr))
11. **return** -EFAULT;
12. ptr += **sizeof**(uint32\_t);
13. **if** (\_IOC\_NR(cmd) < ARRAY\_SIZE(binder\_stats.bc)) {
14. binder\_stats.bc[\_IOC\_NR(cmd)]++;
15. proc->stats.bc[\_IOC\_NR(cmd)]++;
16. **thread**->stats.bc[\_IOC\_NR(cmd)]++;
17. }
18. **switch** (cmd) {
19. ......
20. **case** BC\_FREE\_BUFFER: {
21. **void** \_\_user \*data\_ptr;
22. **struct** binder\_buffer \*buffer;
24. **if** (get\_user(data\_ptr, (**void** \* \_\_user \*)ptr))
25. **return** -EFAULT;
26. ptr += **sizeof**(**void** \*);
28. buffer = binder\_buffer\_lookup(proc, data\_ptr);
29. **if** (buffer == NULL) {
30. binder\_user\_error("binder: %d:%d "
31. "BC\_FREE\_BUFFER u%p no match\n",
32. proc->pid, **thread**->pid, data\_ptr);
33. **break**;
34. }
35. **if** (!buffer->allow\_user\_free) {
36. binder\_user\_error("binder: %d:%d "
37. "BC\_FREE\_BUFFER u%p matched "
38. "unreturned buffer\n",
39. proc->pid, **thread**->pid, data\_ptr);
40. **break**;
41. }
42. **if** (binder\_debug\_mask & BINDER\_DEBUG\_FREE\_BUFFER)
43. printk(KERN\_INFO "binder: %d:%d BC\_FREE\_BUFFER u%p found buffer %d for %s transaction\n",
44. proc->pid, **thread**->pid, data\_ptr, buffer->debug\_id,
45. buffer->transaction ? "active" : "finished");
47. **if** (buffer->transaction) {
48. buffer->transaction->buffer = NULL;
49. buffer->transaction = NULL;
50. }
51. **if** (buffer->async\_transaction && buffer->target\_node) {
52. BUG\_ON(!buffer->target\_node->has\_async\_transaction);
53. **if** (list\_empty(&buffer->target\_node->async\_todo))
54. buffer->target\_node->has\_async\_transaction = 0;
55. **else**
56. list\_move\_tail(buffer->target\_node->async\_todo.next, &**thread**->todo);
57. }
58. binder\_transaction\_buffer\_release(proc, buffer, NULL);
59. binder\_free\_buf(proc, buffer);
60. **break**;
61. }
63. ......
64. \*consumed = ptr - buffer;
65. }
66. **return** 0;
67. }

       首先通过看这个语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. get\_user(data\_ptr, (**void** \* \_\_user \*)ptr)

       这个是获得要删除的Buffer的用户空间地址，接着通过下面这个语句来找到这个地址对应的struct binder\_buffer信息：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. buffer = binder\_buffer\_lookup(proc, data\_ptr);

       因为这个空间是前面在binder\_transaction里面分配的，所以这里一定能找到。

       最后，就可以释放这块内存了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. binder\_transaction\_buffer\_release(proc, buffer, NULL);
2. binder\_free\_buf(proc, buffer);

       再来看另外一个命令BC\_REPLY：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **int**
2. binder\_thread\_write(**struct** binder\_proc \*proc, **struct** binder\_thread \***thread**,
3. **void** \_\_user \*buffer, **int** size, **signed** **long** \*consumed)
4. {
5. uint32\_t cmd;
6. **void** \_\_user \*ptr = buffer + \*consumed;
7. **void** \_\_user \*end = buffer + size;
9. **while** (ptr < end && **thread**->return\_error == BR\_OK) {
10. **if** (get\_user(cmd, (uint32\_t \_\_user \*)ptr))
11. **return** -EFAULT;
12. ptr += **sizeof**(uint32\_t);
13. **if** (\_IOC\_NR(cmd) < ARRAY\_SIZE(binder\_stats.bc)) {
14. binder\_stats.bc[\_IOC\_NR(cmd)]++;
15. proc->stats.bc[\_IOC\_NR(cmd)]++;
16. **thread**->stats.bc[\_IOC\_NR(cmd)]++;
17. }
18. **switch** (cmd) {
19. ......
20. **case** BC\_TRANSACTION:
21. **case** BC\_REPLY: {
22. **struct** binder\_transaction\_data tr;
24. **if** (copy\_from\_user(&tr, ptr, **sizeof**(tr)))
25. **return** -EFAULT;
26. ptr += **sizeof**(tr);
27. binder\_transaction(proc, **thread**, &tr, cmd == BC\_REPLY);
28. **break**;
29. }
31. ......
32. \*consumed = ptr - buffer;
33. }
34. **return** 0;
35. }

       又再次进入到binder\_transaction函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **void**
2. binder\_transaction(**struct** binder\_proc \*proc, **struct** binder\_thread \***thread**,
3. **struct** binder\_transaction\_data \*tr, **int** reply)
4. {
5. **struct** binder\_transaction \*t;
6. **struct** binder\_work \*tcomplete;
7. **size\_t** \*offp, \*off\_end;
8. **struct** binder\_proc \*target\_proc;
9. **struct** binder\_thread \*target\_thread = NULL;
10. **struct** binder\_node \*target\_node = NULL;
11. **struct** list\_head \*target\_list;
12. wait\_queue\_head\_t \*target\_wait;
13. **struct** binder\_transaction \*in\_reply\_to = NULL;
14. **struct** binder\_transaction\_log\_entry \*e;
15. uint32\_t return\_error;
17. ......
19. **if** (reply) {
20. in\_reply\_to = **thread**->transaction\_stack;
21. **if** (in\_reply\_to == NULL) {
22. ......
23. return\_error = BR\_FAILED\_REPLY;
24. **goto** err\_empty\_call\_stack;
25. }
26. binder\_set\_nice(in\_reply\_to->saved\_priority);
27. **if** (in\_reply\_to->to\_thread != **thread**) {
28. .......
29. **goto** err\_bad\_call\_stack;
30. }
31. **thread**->transaction\_stack = in\_reply\_to->to\_parent;
32. target\_thread = in\_reply\_to->from;
33. **if** (target\_thread == NULL) {
34. return\_error = BR\_DEAD\_REPLY;
35. **goto** err\_dead\_binder;
36. }
37. **if** (target\_thread->transaction\_stack != in\_reply\_to) {
38. ......
39. return\_error = BR\_FAILED\_REPLY;
40. in\_reply\_to = NULL;
41. target\_thread = NULL;
42. **goto** err\_dead\_binder;
43. }
44. target\_proc = target\_thread->proc;
45. } **else** {
46. ......
47. }
48. **if** (target\_thread) {
49. e->to\_thread = target\_thread->pid;
50. target\_list = &target\_thread->todo;
51. target\_wait = &target\_thread->wait;
52. } **else** {
53. ......
54. }

57. /\* TODO: reuse incoming transaction for reply \*/
58. t = kzalloc(**sizeof**(\*t), GFP\_KERNEL);
59. **if** (t == NULL) {
60. return\_error = BR\_FAILED\_REPLY;
61. **goto** err\_alloc\_t\_failed;
62. }

65. tcomplete = kzalloc(**sizeof**(\*tcomplete), GFP\_KERNEL);
66. **if** (tcomplete == NULL) {
67. return\_error = BR\_FAILED\_REPLY;
68. **goto** err\_alloc\_tcomplete\_failed;
69. }
71. **if** (!reply && !(tr->flags & TF\_ONE\_WAY))
72. t->from = **thread**;
73. **else**
74. t->from = NULL;
75. t->sender\_euid = proc->tsk->cred->euid;
76. t->to\_proc = target\_proc;
77. t->to\_thread = target\_thread;
78. t->code = tr->code;
79. t->flags = tr->flags;
80. t->priority = task\_nice(current);
81. t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,
82. tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));
83. **if** (t->buffer == NULL) {
84. return\_error = BR\_FAILED\_REPLY;
85. **goto** err\_binder\_alloc\_buf\_failed;
86. }
87. t->buffer->allow\_user\_free = 0;
88. t->buffer->debug\_id = t->debug\_id;
89. t->buffer->transaction = t;
90. t->buffer->target\_node = target\_node;
91. **if** (target\_node)
92. binder\_inc\_node(target\_node, 1, 0, NULL);
94. offp = (**size\_t** \*)(t->buffer->data + ALIGN(tr->data\_size, **sizeof**(**void** \*)));
96. **if** (copy\_from\_user(t->buffer->data, tr->data.ptr.buffer, tr->data\_size)) {
97. binder\_user\_error("binder: %d:%d got transaction with invalid "
98. "data ptr\n", proc->pid, **thread**->pid);
99. return\_error = BR\_FAILED\_REPLY;
100. **goto** err\_copy\_data\_failed;
101. }
102. **if** (copy\_from\_user(offp, tr->data.ptr.offsets, tr->offsets\_size)) {
103. binder\_user\_error("binder: %d:%d got transaction with invalid "
104. "offsets ptr\n", proc->pid, **thread**->pid);
105. return\_error = BR\_FAILED\_REPLY;
106. **goto** err\_copy\_data\_failed;
107. }
109. ......
111. **if** (reply) {
112. BUG\_ON(t->buffer->async\_transaction != 0);
113. binder\_pop\_transaction(target\_thread, in\_reply\_to);
114. } **else** **if** (!(t->flags & TF\_ONE\_WAY)) {
115. ......
116. } **else** {
117. ......
118. }
119. t->work.type = BINDER\_WORK\_TRANSACTION;
120. list\_add\_tail(&t->work.entry, target\_list);
121. tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;
122. list\_add\_tail(&tcomplete->entry, &**thread**->todo);
123. **if** (target\_wait)
124. wake\_up\_interruptible(target\_wait);
125. **return**;
126. ......
127. }

       注意，这里的reply为1，我们忽略掉其它无关代码。

       前面Service Manager正在binder\_thread\_read函数中被MediaPlayerService启动后进程唤醒后，在最后会把当前处理完的事务放在thread->transaction\_stack中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)) {
2. t->to\_parent = **thread**->transaction\_stack;
3. t->to\_thread = **thread**;
4. **thread**->transaction\_stack = t;
5. }

       所以，这里，首先是把它这个binder\_transaction取回来，并且放在本地变量in\_reply\_to中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. in\_reply\_to = **thread**->transaction\_stack;

       接着就可以通过in\_reply\_to得到最终发出这个事务请求的线程和进程：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. target\_thread = in\_reply\_to->from;
2. target\_proc = target\_thread->proc;

        然后得到target\_list和target\_wait：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. target\_list = &target\_thread->todo;
2. target\_wait = &target\_thread->wait;

       下面这一段代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. /\* TODO: reuse incoming transaction for reply \*/
2. t = kzalloc(**sizeof**(\*t), GFP\_KERNEL);
3. **if** (t == NULL) {
4. return\_error = BR\_FAILED\_REPLY;
5. **goto** err\_alloc\_t\_failed;
6. }

9. tcomplete = kzalloc(**sizeof**(\*tcomplete), GFP\_KERNEL);
10. **if** (tcomplete == NULL) {
11. return\_error = BR\_FAILED\_REPLY;
12. **goto** err\_alloc\_tcomplete\_failed;
13. }
15. **if** (!reply && !(tr->flags & TF\_ONE\_WAY))
16. t->from = **thread**;
17. **else**
18. t->from = NULL;
19. t->sender\_euid = proc->tsk->cred->euid;
20. t->to\_proc = target\_proc;
21. t->to\_thread = target\_thread;
22. t->code = tr->code;
23. t->flags = tr->flags;
24. t->priority = task\_nice(current);
25. t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,
26. tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));
27. **if** (t->buffer == NULL) {
28. return\_error = BR\_FAILED\_REPLY;
29. **goto** err\_binder\_alloc\_buf\_failed;
30. }
31. t->buffer->allow\_user\_free = 0;
32. t->buffer->debug\_id = t->debug\_id;
33. t->buffer->transaction = t;
34. t->buffer->target\_node = target\_node;
35. **if** (target\_node)
36. binder\_inc\_node(target\_node, 1, 0, NULL);
38. offp = (**size\_t** \*)(t->buffer->data + ALIGN(tr->data\_size, **sizeof**(**void** \*)));
40. **if** (copy\_from\_user(t->buffer->data, tr->data.ptr.buffer, tr->data\_size)) {
41. binder\_user\_error("binder: %d:%d got transaction with invalid "
42. "data ptr\n", proc->pid, **thread**->pid);
43. return\_error = BR\_FAILED\_REPLY;
44. **goto** err\_copy\_data\_failed;
45. }
46. **if** (copy\_from\_user(offp, tr->data.ptr.offsets, tr->offsets\_size)) {
47. binder\_user\_error("binder: %d:%d got transaction with invalid "
48. "offsets ptr\n", proc->pid, **thread**->pid);
49. return\_error = BR\_FAILED\_REPLY;
50. **goto** err\_copy\_data\_failed;
51. }

          我们在前面已经分析过了，这里不再重复。但是有一点要注意的是，这里target\_node为NULL，因此，t->buffer->target\_node也为NULL。

          函数本来有一个for循环，用来处理数据中的Binder对象，这里由于没有Binder对象，所以就略过了。到了下面这句代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. binder\_pop\_transaction(target\_thread, in\_reply\_to);

          我们看看做了什么事情：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **void**
2. binder\_pop\_transaction(
3. **struct** binder\_thread \*target\_thread, **struct** binder\_transaction \*t)
4. {
5. **if** (target\_thread) {
6. BUG\_ON(target\_thread->transaction\_stack != t);
7. BUG\_ON(target\_thread->transaction\_stack->from != target\_thread);
8. target\_thread->transaction\_stack =
9. target\_thread->transaction\_stack->from\_parent;
10. t->from = NULL;
11. }
12. t->need\_reply = 0;
13. **if** (t->buffer)
14. t->buffer->transaction = NULL;
15. kfree(t);
16. binder\_stats.obj\_deleted[BINDER\_STAT\_TRANSACTION]++;
17. }

        由于到了这里，已经不需要in\_reply\_to这个transaction了，就把它删掉。

        回到binder\_transaction函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. t->work.type = BINDER\_WORK\_TRANSACTION;
2. list\_add\_tail(&t->work.entry, target\_list);
3. tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;
4. list\_add\_tail(&tcomplete->entry, &**thread**->todo);

         和前面一样，分别把t和tcomplete分别放在target\_list和thread->todo队列中，这里的target\_list指的就是最初调用IServiceManager::addService的MediaPlayerService的Server主线程的的thread->todo队列了，而thread->todo指的是Service Manager中用来回复IServiceManager::addService请求的线程。

        最后，唤醒等待在target\_wait队列上的线程了，就是最初调用IServiceManager::addService的MediaPlayerService的Server主线程了，它最后在binder\_thread\_read函数中睡眠在thread->wait上，就是这里的target\_wait了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (target\_wait)
2. wake\_up\_interruptible(target\_wait);

        这样，Service Manger回复调用IServiceManager::addService请求就算完成了，重新回到frameworks/base/cmds/servicemanager/binder.c文件中的binder\_loop函数等待下一个Client请求的到来。事实上，Service Manger回到binder\_loop函数再次执行ioctl函数时候，又会再次进入到binder\_thread\_read函数。这时个会发现thread->todo不为空，这是因为刚才我们调用了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. list\_add\_tail(&tcomplete->entry, &**thread**->todo);

          把一个工作项tcompelete放在了在thread->todo中，这个tcompelete的type为BINDER\_WORK\_TRANSACTION\_COMPLETE，因此，Binder驱动程序会执行下面操作：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **switch** (w->type) {
2. **case** BINDER\_WORK\_TRANSACTION\_COMPLETE: {
3. cmd = BR\_TRANSACTION\_COMPLETE;
4. **if** (put\_user(cmd, (uint32\_t \_\_user \*)ptr))
5. **return** -EFAULT;
6. ptr += **sizeof**(uint32\_t);
8. list\_del(&w->entry);
9. kfree(w);
11. } **break**;
12. ......
13. }

        binder\_loop函数执行完这个ioctl调用后，才会在下一次调用ioctl进入到Binder驱动程序进入休眠状态，等待下一次Client的请求。

        上面讲到调用IServiceManager::addService的MediaPlayerService的Server主线程被唤醒了，于是，重新执行binder\_thread\_read函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **static** **int**
2. binder\_thread\_read(**struct** binder\_proc \*proc, **struct** binder\_thread \***thread**,
3. **void**  \_\_user \*buffer, **int** size, **signed** **long** \*consumed, **int** non\_block)
4. {
5. **void** \_\_user \*ptr = buffer + \*consumed;
6. **void** \_\_user \*end = buffer + size;
8. **int** ret = 0;
9. **int** wait\_for\_proc\_work;
11. **if** (\*consumed == 0) {
12. **if** (put\_user(BR\_NOOP, (uint32\_t \_\_user \*)ptr))
13. **return** -EFAULT;
14. ptr += **sizeof**(uint32\_t);
15. }
17. retry:
18. wait\_for\_proc\_work = **thread**->transaction\_stack == NULL && list\_empty(&**thread**->todo);
20. ......
22. **if** (wait\_for\_proc\_work) {
23. ......
24. } **else** {
25. **if** (non\_block) {
26. **if** (!binder\_has\_thread\_work(**thread**))
27. ret = -EAGAIN;
28. } **else**
29. ret = wait\_event\_interruptible(**thread**->wait, binder\_has\_thread\_work(**thread**));
30. }
32. ......
34. **while** (1) {
35. uint32\_t cmd;
36. **struct** binder\_transaction\_data tr;
37. **struct** binder\_work \*w;
38. **struct** binder\_transaction \*t = NULL;
40. **if** (!list\_empty(&**thread**->todo))
41. w = list\_first\_entry(&**thread**->todo, **struct** binder\_work, entry);
42. **else** **if** (!list\_empty(&proc->todo) && wait\_for\_proc\_work)
43. w = list\_first\_entry(&proc->todo, **struct** binder\_work, entry);
44. **else** {
45. **if** (ptr - buffer == 4 && !(**thread**->looper & BINDER\_LOOPER\_STATE\_NEED\_RETURN)) /\* no data added \*/
46. **goto** retry;
47. **break**;
48. }
50. ......
52. **switch** (w->type) {
53. **case** BINDER\_WORK\_TRANSACTION: {
54. t = container\_of(w, **struct** binder\_transaction, work);
55. } **break**;
56. ......
57. }
59. **if** (!t)
60. **continue**;
62. BUG\_ON(t->buffer == NULL);
63. **if** (t->buffer->target\_node) {
64. ......
65. } **else** {
66. tr.target.ptr = NULL;
67. tr.cookie = NULL;
68. cmd = BR\_REPLY;
69. }
70. tr.code = t->code;
71. tr.flags = t->flags;
72. tr.sender\_euid = t->sender\_euid;
74. **if** (t->from) {
75. ......
76. } **else** {
77. tr.sender\_pid = 0;
78. }
80. tr.data\_size = t->buffer->data\_size;
81. tr.offsets\_size = t->buffer->offsets\_size;
82. tr.data.ptr.buffer = (**void** \*)t->buffer->data + proc->user\_buffer\_offset;
83. tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data\_size, **sizeof**(**void** \*));
85. **if** (put\_user(cmd, (uint32\_t \_\_user \*)ptr))
86. **return** -EFAULT;
87. ptr += **sizeof**(uint32\_t);
88. **if** (copy\_to\_user(ptr, &tr, **sizeof**(tr)))
89. **return** -EFAULT;
90. ptr += **sizeof**(tr);
92. ......
94. list\_del(&t->work.entry);
95. t->buffer->allow\_user\_free = 1;
96. **if** (cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)) {
97. ......
98. } **else** {
99. t->buffer->transaction = NULL;
100. kfree(t);
101. binder\_stats.obj\_deleted[BINDER\_STAT\_TRANSACTION]++;
102. }
103. **break**;
104. }
106. done:
107. ......
108. **return** 0;
109. }

         在while循环中，从thread->todo得到w，w->type为BINDER\_WORK\_TRANSACTION，于是，得到t。从上面可以知道，Service Manager反回了一个0回来，写在t->buffer->data里面，现在把t->buffer->data加上proc->user\_buffer\_offset，得到用户空间地址，保存在tr.data.ptr.buffer里面，这样用户空间就可以访问这个返回码了。由于cmd不等于BR\_TRANSACTION，这时就可以把t删除掉了，因为以后都不需要用了。

         执行完这个函数后，就返回到binder\_ioctl函数，执行下面语句，把数据返回给用户空间：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr))) {
2. ret = -EFAULT;
3. **goto** err;
4. }

         接着返回到用户空间IPCThreadState::talkWithDriver函数，最后返回到IPCThreadState::waitForResponse函数，最终执行到下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t IPCThreadState::waitForResponse(Parcel \*reply, status\_t \*acquireResult)
2. {
3. int32\_t cmd;
4. int32\_t err;
6. **while** (1) {
7. **if** ((err=talkWithDriver()) < NO\_ERROR) **break**;
9. ......
11. cmd = mIn.readInt32();
13. ......
15. **switch** (cmd) {
16. ......
17. **case** BR\_REPLY:
18. {
19. binder\_transaction\_data tr;
20. err = mIn.read(&tr, **sizeof**(tr));
21. LOG\_ASSERT(err == NO\_ERROR, "Not enough command data for brREPLY");
22. **if** (err != NO\_ERROR) **goto** finish;
24. **if** (reply) {
25. **if** ((tr.flags & TF\_STATUS\_CODE) == 0) {
26. reply->ipcSetDataReference(
27. **reinterpret\_cast**<**const** uint8\_t\*>(tr.data.ptr.buffer),
28. tr.data\_size,
29. **reinterpret\_cast**<**const** **size\_t**\*>(tr.data.ptr.offsets),
30. tr.offsets\_size/**sizeof**(**size\_t**),
31. freeBuffer, **this**);
32. } **else** {
33. ......
34. }
35. } **else** {
36. ......
37. }
38. }
39. **goto** finish;
41. ......
42. }
43. }
45. finish:
46. ......
47. **return** err;
48. }

        注意，这里的tr.flags等于0，这个是在上面的binder\_send\_reply函数里设置的。最终把结果保存在reply了：

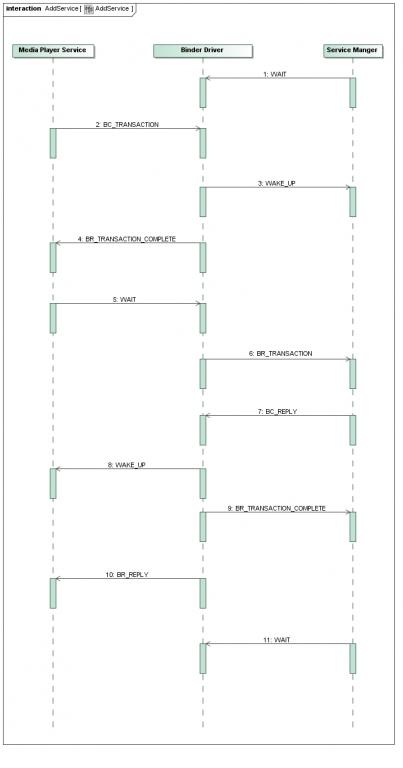
**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. reply->ipcSetDataReference(
2. **reinterpret\_cast**<**const** uint8\_t\*>(tr.data.ptr.buffer),
3. tr.data\_size,
4. **reinterpret\_cast**<**const** **size\_t**\*>(tr.data.ptr.offsets),
5. tr.offsets\_size/**sizeof**(**size\_t**),
6. freeBuffer, **this**);

       这个函数我们就不看了，有兴趣的读者可以研究一下。

       从这里层层返回，最后回到MediaPlayerService::instantiate函数中。

       至此，IServiceManager::addService终于执行完毕了。这个过程非常复杂，但是如果我们能够深刻地理解这一过程，将能很好地理解Binder机制的设计思想和实现过程。这里，对IServiceManager::addService过程中MediaPlayerService、ServiceManager和BinderDriver之间的交互作一个小结：



        回到frameworks/base/media/mediaserver/main\_mediaserver.cpp文件中的main函数，接下去还要执行下面两个函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. ProcessState::self()->startThreadPool();
2. IPCThreadState::self()->joinThreadPool();

        首先看ProcessState::startThreadPool函数的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **void** ProcessState::startThreadPool()
2. {
3. AutoMutex \_l(mLock);
4. **if** (!mThreadPoolStarted) {
5. mThreadPoolStarted = **true**;
6. spawnPooledThread(**true**);
7. }
8. }

       这里调用spwanPooledThread：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **void** ProcessState::spawnPooledThread(**bool** isMain)
2. {
3. **if** (mThreadPoolStarted) {
4. int32\_t s = android\_atomic\_add(1, &mThreadPoolSeq);
5. **char** buf[32];
6. sprintf(buf, "Binder Thread #%d", s);
7. LOGV("Spawning new pooled thread, name=%s\n", buf);
8. sp<Thread> t = **new** PoolThread(isMain);
9. t->run(buf);
10. }
11. }

       这里主要是创建一个线程，PoolThread继续Thread类，Thread类定义在frameworks/base/libs/utils/Threads.cpp文件中，其run函数最终调用子类的threadLoop函数，这里即为PoolThread::threadLoop函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **virtual** **bool** threadLoop()
2. {
3. IPCThreadState::self()->joinThreadPool(mIsMain);
4. **return** **false**;
5. }

       这里和frameworks/base/media/mediaserver/main\_mediaserver.cpp文件中的main函数一样，最终都是调用了IPCThreadState::joinThreadPool函数，它们的区别是，一个参数是true，一个是默认值false。我们来看一下这个函数的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. **void** IPCThreadState::joinThreadPool(**bool** isMain)
2. {
3. LOG\_THREADPOOL("\*\*\*\* THREAD %p (PID %d) IS JOINING THE THREAD POOL\n", (**void**\*)pthread\_self(), getpid());
5. mOut.writeInt32(isMain ? BC\_ENTER\_LOOPER : BC\_REGISTER\_LOOPER);
7. ......
9. status\_t result;
10. **do** {
11. int32\_t cmd;
13. .......
15. // now get the next command to be processed, waiting if necessary
16. result = talkWithDriver();
17. **if** (result >= NO\_ERROR) {
18. **size\_t** IN = mIn.dataAvail();
19. **if** (IN < **sizeof**(int32\_t)) **continue**;
20. cmd = mIn.readInt32();
21. ......
22. }
24. result = executeCommand(cmd);
25. }
27. ......
28. } **while** (result != -ECONNREFUSED && result != -EBADF);
30. .......
32. mOut.writeInt32(BC\_EXIT\_LOOPER);
33. talkWithDriver(**false**);
34. }

        这个函数最终是在一个无穷循环中，通过调用talkWithDriver函数来和Binder驱动程序进行交互，实际上就是调用talkWithDriver来等待Client的请求，然后再调用executeCommand来处理请求，而在executeCommand函数中，最终会调用BBinder::transact来真正处理Client的请求：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t IPCThreadState::executeCommand(int32\_t cmd)
2. {
3. BBinder\* obj;
4. RefBase::weakref\_type\* refs;
5. status\_t result = NO\_ERROR;
7. **switch** (cmd) {
8. ......
10. **case** BR\_TRANSACTION:
11. {
12. binder\_transaction\_data tr;
13. result = mIn.read(&tr, **sizeof**(tr));
15. ......
17. Parcel reply;
19. ......
21. **if** (tr.target.ptr) {
22. sp<BBinder> b((BBinder\*)tr.cookie);
23. **const** status\_t error = b->transact(tr.code, buffer, &reply, tr.flags);
24. **if** (error < NO\_ERROR) reply.setError(error);
26. } **else** {
27. **const** status\_t error = the\_context\_object->transact(tr.code, buffer, &reply, tr.flags);
28. **if** (error < NO\_ERROR) reply.setError(error);
29. }
31. ......
32. }
33. **break**;
35. .......
36. }
38. **if** (result != NO\_ERROR) {
39. mLastError = result;
40. }
42. **return** result;
43. }

        接下来再看一下BBinder::transact的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t BBinder::transact(
2. uint32\_t code, **const** Parcel& data, Parcel\* reply, uint32\_t flags)
3. {
4. data.setDataPosition(0);
6. status\_t err = NO\_ERROR;
7. **switch** (code) {
8. **case** PING\_TRANSACTION:
9. reply->writeInt32(pingBinder());
10. **break**;
11. **default**:
12. err = onTransact(code, data, reply, flags);
13. **break**;
14. }
16. **if** (reply != NULL) {
17. reply->setDataPosition(0);
18. }
20. **return** err;
21. }

       最终会调用onTransact函数来处理。在这个场景中，BnMediaPlayerService继承了BBinder类，并且重载了onTransact函数，因此，这里实际上是调用了BnMediaPlayerService::onTransact函数，这个函数定义在frameworks/base/libs/media/libmedia/IMediaPlayerService.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6629298)

1. status\_t BnMediaPlayerService::onTransact(
2. uint32\_t code, **const** Parcel& data, Parcel\* reply, uint32\_t flags)
3. {
4. **switch**(code) {
5. **case** CREATE\_URL: {
6. ......
7. } **break**;
8. **case** CREATE\_FD: {
9. ......
10. } **break**;
11. **case** DECODE\_URL: {
12. ......
13. } **break**;
14. **case** DECODE\_FD: {
15. ......
16. } **break**;
17. **case** CREATE\_MEDIA\_RECORDER: {
18. ......
19. } **break**;
20. **case** CREATE\_METADATA\_RETRIEVER: {
21. ......
22. } **break**;
23. **case** GET\_OMX: {
24. ......
25. } **break**;
26. **default**:
27. **return** BBinder::onTransact(code, data, reply, flags);
28. }
29. }

       至此，我们就以MediaPlayerService为例，完整地介绍了Android系统进程间通信Binder机制中的Server启动过程。Server启动起来之后，就会在一个无穷循环中等待Client的请求了。在下一篇文章中，我们将介绍Client如何通过Service Manager远程接口来获得Server远程接口，进而调用Server远程接口来使用Server提供的服务，敬请关注。