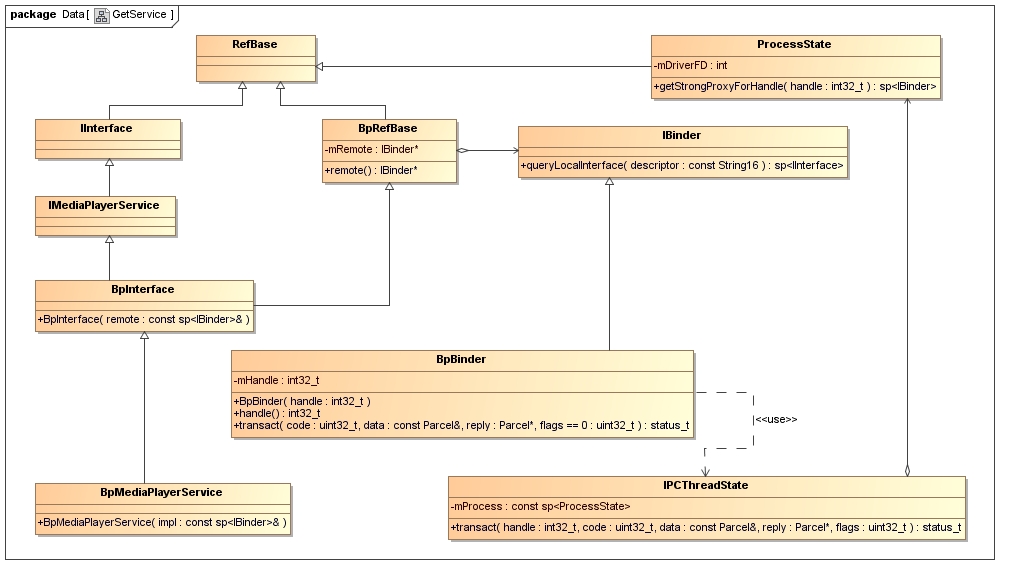
[Android系统进程间通信（IPC）机制Binder中的Client获得Server远程接口过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6633311)

在上一篇文章中，我们分析了Android系统进程间通信机制Binder中的Server在启动过程使用Service Manager的addService接口把自己添加到Service Manager守护过程中接受管理。在这一篇文章中，我们将深入到Binder驱动程序源代码去分析Client是如何通过Service Manager的getService接口中来获得Server远程接口的。Client只有获得了Server的远程接口之后，才能进一步调用Server提供的服务。

        这里，我们仍然是通过Android系统中自带的多媒体播放器为例子来说明Client是如何通过IServiceManager::getService接口来获得MediaPlayerService这个Server的远程接口的。假设计读者已经阅读过前面三篇文章[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)、[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)和[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)，即假设Service Manager和MediaPlayerService已经启动完毕，Service Manager现在等待Client的请求。

        这里，我们要举例子说明的Client便是MediaPlayer了，它声明和实现在frameworks/base/include/media/mediaplayer.h和frameworks/base/media/libmedia/mediaplayer.cpp文件中。MediaPlayer继承于IMediaDeathNotifier类，这个类声明和实现在frameworks/base/include/media/IMediaDeathNotifier.h和frameworks/base/media/libmedia//IMediaDeathNotifier.cpp文件中，里面有一个静态成员函数getMeidaPlayerService，它通过IServiceManager::getService接口来获得MediaPlayerService的远程接口。

        在介绍IMediaDeathNotifier::getMeidaPlayerService函数之前，我们先了解一下这个函数的目标。看来前面[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)这篇文章的读者知道，我们在获取Service Manager远程接口时，最终是获得了一个BpServiceManager对象的IServiceManager接口。类似地，我们要获得MediaPlayerService的远程接口，实际上就是要获得一个称为BpMediaPlayerService对象的IMediaPlayerService接口。现在，我们就先来看一下BpMediaPlayerService的类图：



        从这个类图可以看到，BpMediaPlayerService继承于BpInterface<IMediaPlayerService>类，即BpMediaPlayerService继承了IMediaPlayerService类和BpRefBase类，这两个类又分别继续了RefBase类。BpRefBase类有一个成员变量mRemote，它的类型为IBinder，实际是一个BpBinder对象。BpBinder类使用了IPCThreadState类来与Binder驱动程序进行交互，而IPCThreadState类有一个成员变量mProcess，它的类型为ProcessState，IPCThreadState类借助ProcessState类来打开Binder设备文件/dev/binder，因此，它可以和Binder驱动程序进行交互。

       BpMediaPlayerService的构造函数有一个参数impl，它的类型为const sp<IBinder>&，从上面的描述中，这个实际上就是一个BpBinder对象。这样，要创建一个BpMediaPlayerService对象，首先就要有一个BpBinder对象。再来看BpBinder类的构造函数，它有一个参数handle，类型为int32\_t，这个参数的意义就是请求MediaPlayerService这个远程接口的进程对MediaPlayerService这个Binder实体的引用了。因此，获取MediaPlayerService这个远程接口的本质问题就变为从Service Manager中获得MediaPlayerService的一个句柄了。

       现在，我们就来看一下IMediaDeathNotifier::getMeidaPlayerService的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. // establish binder interface to MediaPlayerService
2. /\*static\*/**const** sp<IMediaPlayerService>&
3. IMediaDeathNotifier::getMediaPlayerService()
4. {
5. LOGV("getMediaPlayerService");
6. Mutex::Autolock \_l(sServiceLock);
7. **if** (sMediaPlayerService.get() == 0) {
8. sp<IServiceManager> sm = defaultServiceManager();
9. sp<IBinder> binder;
10. **do** {
11. binder = sm->getService(String16("media.player"));
12. **if** (binder != 0) {
13. **break**;
14. }
15. LOGW("Media player service not published, waiting...");
16. usleep(500000); // 0.5 s
17. } **while**(**true**);
19. **if** (sDeathNotifier == NULL) {
20. sDeathNotifier = **new** DeathNotifier();
21. }
22. binder->linkToDeath(sDeathNotifier);
23. sMediaPlayerService = interface\_cast<IMediaPlayerService>(binder);
24. }
25. LOGE\_IF(sMediaPlayerService == 0, "no media player service!?");
26. **return** sMediaPlayerService;
27. }

        函数首先通过defaultServiceManager函数来获得Service Manager的远程接口，实际上就是获得BpServiceManager的IServiceManager接口，具体可以参考[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。总的来说，这里的语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. sp<IServiceManager> sm = defaultServiceManager();

        相当于是：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. sp<IServiceManager> sm = **new** BpServiceManager(**new** BpBinder(0));

        这里的0表示Service Manager的远程接口的句柄值是0。

        接下去的while循环是通过sm->getService接口来不断尝试获得名称为“media.player”的Service，即MediaPlayerService。为什么要通过这无穷循环来得MediaPlayerService呢？因为这时候MediaPlayerService可能还没有启动起来，所以这里如果发现取回来的binder接口为NULL，就睡眠0.5秒，然后再尝试获取，这是获取Service接口的标准做法。  
        我们来看一下BpServiceManager::getService的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **class** BpServiceManager : **public** BpInterface<IServiceManager>
2. {
3. ......
5. **virtual** sp<IBinder> getService(**const** String16& name) **const**
6. {
7. unsigned n;
8. **for** (n = 0; n < 5; n++){
9. sp<IBinder> svc = checkService(name);
10. **if** (svc != NULL) **return** svc;
11. LOGI("Waiting for service %s...\n", String8(name).string());
12. sleep(1);
13. }
14. **return** NULL;
15. }
17. **virtual** sp<IBinder> checkService( **const** String16& name) **const**
18. {
19. Parcel data, reply;
20. data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
21. data.writeString16(name);
22. remote()->transact(CHECK\_SERVICE\_TRANSACTION, data, &reply);
23. **return** reply.readStrongBinder();
24. }
26. ......
27. };

         BpServiceManager::getService通过BpServiceManager::checkService执行操作。

         在BpServiceManager::checkService中，首先是通过Parcel::writeInterfaceToken往data写入一个RPC头，这个我们在[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)一文已经介绍过了，就是写往data里面写入了一个整数和一个字符串“android.os.IServiceManager”， Service Manager来处理CHECK\_SERVICE\_TRANSACTION请求之前，会先验证一下这个RPC头，看看是否正确。接着再往data写入一个字符串name，这里就是“media.player”了。回忆一下[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)这篇文章，那里已经往Service Manager中注册了一个名字为“media.player”的MediaPlayerService。

        这里的remote()返回的是一个BpBinder，具体可以参考[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，于是，就进行到BpBinder::transact函数了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. }

        这里的mHandle = 0，code = CHECK\_SERVICE\_TRANSACTION，flags = 0。

        这里再进入到IPCThread::transact函数中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. }

         首先是调用函数writeTransactionData写入将要传输的数据到IPCThreadState的成员变量mOut中去：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. }

        结构体binder\_transaction\_data在上一篇文章[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)已经介绍过，这里不再累述，这个结构体是用来描述要传输的参数的内容的。这里着重描述一下将要传输的参数tr里面的内容，handle = 0，code =  CHECK\_SERVICE\_TRANSACTION，cmd = BC\_TRANSACTION，data里面的数据分别为：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

       这是在BpServiceManager::checkService函数里面写进去的，其中前两个是RPC头，Service Manager在收到这个请求时会验证这两个参数是否正确，这点前面也提到了。IPCThread->getStrictModePolicy默认返回0，STRICT\_MODE\_PENALTY\_GATHER定义为：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. // Note: must be kept in sync with android/os/StrictMode.java's PENALTY\_GATHER
2. #define STRICT\_MODE\_PENALTY\_GATHER 0x100

       我们不关心这个参数的含义，这不会影响我们分析下面的源代码，有兴趣的读者可以研究一下。这里要注意的是，要传输的参数不包含有Binder对象，因此tr.offsets\_size = 0。要传输的参数最后写入到IPCThreadState的成员变量mOut中，包括cmd和tr两个数据。

       回到IPCThread::transact函数中，由于(flags & TF\_ONE\_WAY) == 0为true，即这是一个同步请求，并且reply  != NULL，最终调用：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. err = waitForResponse(reply);

       进入到waitForResponse函数中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. }

        这个函数通过IPCThreadState::talkWithDriver与驱动程序进行交互：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. ......
28. // Return immediately if there is nothing to do.
29. **if** ((bwr.write\_size == 0) && (bwr.read\_size == 0)) **return** NO\_ERROR;
31. bwr.write\_consumed = 0;
32. bwr.read\_consumed = 0;
33. status\_t err;
34. **do** {
35. ......
36. #if defined(HAVE\_ANDROID\_OS)
37. **if** (ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr) >= 0)
38. err = NO\_ERROR;
39. **else**
40. err = -errno;
41. #else
42. err = INVALID\_OPERATION;
43. #endif
44. ......
45. } **while** (err == -EINTR);
47. ......
49. **if** (err >= NO\_ERROR) {
50. **if** (bwr.write\_consumed > 0) {
51. **if** (bwr.write\_consumed < (ssize\_t)mOut.dataSize())
52. mOut.remove(0, bwr.write\_consumed);
53. **else**
54. mOut.setDataSize(0);
55. }
56. **if** (bwr.read\_consumed > 0) {
57. mIn.setDataSize(bwr.read\_consumed);
58. mIn.setDataPosition(0);
59. }
61. ......
63. **return** NO\_ERROR;
64. }
66. **return** err;
67. }

        这里的needRead为true，因此，bwr.read\_size大于0；outAvail也大于0，因此，bwr.write\_size也大于0。函数最后通过：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

        进入到Binder驱动程序的binder\_ioctl函数中。注意，这里的mProcess->mDriverFD是在我们前面调用defaultServiceManager函数获得Service Manager远程接口时，打开的设备文件/dev/binder的文件描述符，mProcess是IPCSThreadState的成员变量。

        Binder驱动程序的binder\_ioctl函数中，我们只关注BINDER\_WRITE\_READ命令相关的逻辑：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. **default**:
66. ret = -EINVAL;
67. **goto** err;
68. }
69. ret = 0;
70. err:
71. ......
72. **return** ret;
73. }

        这里的filp->private\_data的值是在defaultServiceManager函数创建ProcessState对象时，在ProcessState构造函数通过open文件操作函数打开设备文件/dev/binder时设置好的，它表示的是调用open函数打开设备文件/dev/binder的进程上下文信息，这里将它取出来保存在proc本地变量中。

        这里的thread本地变量表示当前线程上下文信息，通过binder\_get\_thread函数获得。在前面执行ProcessState构造函数时，也会通过ioctl文件操作函数进入到这个函数，那是第一次进入到binder\_ioctl这里，因此，调用binder\_get\_thread时，表示当前进程上下文信息的proc变量还没有关于当前线程的上下文信息，因此，会为proc创建一个表示当前线程上下文信息的thread，会保存在proc->threads表示的红黑树结构中。这里调用binder\_get\_thread就可以直接从proc找到并返回了。

        进入到BINDER\_WRITE\_READ相关的逻辑。先看看BINDER\_WRITE\_READ的定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. #define BINDER\_WRITE\_READ           \_IOWR('b', 1, struct binder\_write\_read)

        这里可以看出，BINDER\_WRITE\_READ命令的参数类型为struct binder\_write\_read：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **struct** binder\_write\_read {
2. **signed** **long** write\_size; /\* bytes to write \*/
3. **signed** **long** write\_consumed; /\* bytes consumed by driver \*/
4. unsigned **long**   write\_buffer;
5. **signed** **long** read\_size;  /\* bytes to read \*/
6. **signed** **long** read\_consumed;  /\* bytes consumed by driver \*/
7. unsigned **long**   read\_buffer;
8. };

        这个结构体的含义可以参考[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。这里首先是通过copy\_from\_user函数把用户传进来的参数的内容拷贝到本地变量bwr中。  
        从上面的调用过程，我们知道，这里bwr.write\_size是大于0的，因此进入到binder\_thread\_write函数中，我们只关注BC\_TRANSACTION相关的逻辑：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. }
30. ......
31. **default**:
32. printk(KERN\_ERR "binder: %d:%d unknown command %d\n", proc->pid, **thread**->pid, cmd);
33. **return** -EINVAL;
34. }
35. \*consumed = ptr - buffer;
36. }
37. **return** 0;
38. }

        这里再次把用户传出来的参数拷贝到本地变量tr中，tr的类型为struct binder\_transaction\_data，这个就是前面我们在IPCThreadState::writeTransactionData写入的内容了。

        接着进入到binder\_transaction函数中，不相关的代码我们忽略掉：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. **if** (!(tr->flags & TF\_ONE\_WAY) && **thread**->transaction\_stack) {
38. ......
39. }
40. }
41. **if** (target\_thread) {
42. ......
43. } **else** {
44. target\_list = &target\_proc->todo;
45. target\_wait = &target\_proc->wait;
46. }
47. ......
49. /\* TODO: reuse incoming transaction for reply \*/
50. t = kzalloc(**sizeof**(\*t), GFP\_KERNEL);
51. **if** (t == NULL) {
52. return\_error = BR\_FAILED\_REPLY;
53. **goto** err\_alloc\_t\_failed;
54. }
55. binder\_stats.obj\_created[BINDER\_STAT\_TRANSACTION]++;
57. tcomplete = kzalloc(**sizeof**(\*tcomplete), GFP\_KERNEL);
58. **if** (tcomplete == NULL) {
59. return\_error = BR\_FAILED\_REPLY;
60. **goto** err\_alloc\_tcomplete\_failed;
61. }
62. binder\_stats.obj\_created[BINDER\_STAT\_TRANSACTION\_COMPLETE]++;
64. t->debug\_id = ++binder\_last\_id;
66. ......

69. **if** (!reply && !(tr->flags & TF\_ONE\_WAY))
70. t->from = **thread**;
71. **else**
72. t->from = NULL;
73. t->sender\_euid = proc->tsk->cred->euid;
74. t->to\_proc = target\_proc;
75. t->to\_thread = target\_thread;
76. t->code = tr->code;
77. t->flags = tr->flags;
78. t->priority = task\_nice(current);
79. t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,
80. tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));
81. **if** (t->buffer == NULL) {
82. return\_error = BR\_FAILED\_REPLY;
83. **goto** err\_binder\_alloc\_buf\_failed;
84. }
85. t->buffer->allow\_user\_free = 0;
86. t->buffer->debug\_id = t->debug\_id;
87. t->buffer->transaction = t;
88. t->buffer->target\_node = target\_node;
89. **if** (target\_node)
90. binder\_inc\_node(target\_node, 1, 0, NULL);
92. offp = (**size\_t** \*)(t->buffer->data + ALIGN(tr->data\_size, **sizeof**(**void** \*)));
94. **if** (copy\_from\_user(t->buffer->data, tr->data.ptr.buffer, tr->data\_size)) {
95. ......
96. return\_error = BR\_FAILED\_REPLY;
97. **goto** err\_copy\_data\_failed;
98. }
100. ......
102. **if** (reply) {
103. ......
104. } **else** **if** (!(t->flags & TF\_ONE\_WAY)) {
105. BUG\_ON(t->buffer->async\_transaction != 0);
106. t->need\_reply = 1;
107. t->from\_parent = **thread**->transaction\_stack;
108. **thread**->transaction\_stack = t;
109. } **else** {
110. ......
111. }
113. t->work.type = BINDER\_WORK\_TRANSACTION;
114. list\_add\_tail(&t->work.entry, target\_list);
115. tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;
116. list\_add\_tail(&tcomplete->entry, &**thread**->todo);
117. **if** (target\_wait)
118. wake\_up\_interruptible(target\_wait);
119. **return**;
121. ......
122. }

        注意，这里的参数reply = 0，表示这是一个BC\_TRANSACTION命令。  
        前面我们提到，传给驱动程序的handle值为0，即这里的tr->target.handle = 0，表示请求的目标Binder对象是Service Manager，因此有：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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;

        其中binder\_context\_mgr\_node是在Service Manager通知Binder驱动程序它是守护过程时创建的。

        接着创建一个待完成事项tcomplete，它的类型为struct binder\_work，这是等一会要保存在当前线程的todo队列去的，表示当前线程有一个待完成的事务。紧跟着创建一个待处理事务t，它的类型为struct binder\_transaction，这是等一会要存在到Service Manager的todo队列去的，表示Service Manager当前有一个事务需要处理。同时，这个待处理事务t也要存放在当前线程的待完成事务transaction\_stack列表中去：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. t->from\_parent = **thread**->transaction\_stack;
2. **thread**->transaction\_stack = t;

        这样表明当前线程还有事务要处理。

        继续往下看，就是分别把tcomplete和t放在当前线程thread和Service Manager进程的todo队列去了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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);

        最后，Service Manager有事情可做了，就要唤醒它了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. wake\_up\_interruptible(target\_wait);

        前面我们提到，此时Service Manager正在等待Client的请求，也就是Service Manager此时正在进入到Binder驱动程序的binder\_thread\_read函数中，并且休眠在target->wait上，具体参考[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。  
        这里，我们暂时忽略Service Manager被唤醒之后的情景，继续看当前线程的执行。  
        函数binder\_transaction执行完成之后，就一路返回到binder\_ioctl函数里去了。函数binder\_ioctl从binder\_thread\_write函数调用处返回后，发现bwr.read\_size大于0，于是就进入到binder\_thread\_read函数去了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. }

       函数首先是写入一个操作码BR\_NOOP到用户传进来的缓冲区中去。

      回忆一下上面的binder\_transaction函数，这里的thread->transaction\_stack != NULL，并且thread->todo也不为空，所以线程不会进入休眠状态。

      进入while循环中，首先是从thread->todo队列中取回待处理事项w，w的类型为BINDER\_WORK\_TRANSACTION\_COMPLETE，这也是在binder\_transaction函数里面设置的。对BINDER\_WORK\_TRANSACTION\_COMPLETE的处理也很简单，只是把一个操作码BR\_TRANSACTION\_COMPLETE写回到用户传进来的缓冲区中去。这时候，用户传进来的缓冲区就包含两个操作码了，分别是BR\_NOOP和BINDER\_WORK\_TRANSACTION\_COMPLETE。

      binder\_thread\_read执行完之后，返回到binder\_ioctl函数中，将操作结果写回到用户空间中去：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

       最后就返回到IPCThreadState::talkWithDriver函数中了。

       IPCThreadState::talkWithDriver函数从下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

       返回后，首先是清空之前写入Binder驱动程序的内容：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **if** (bwr.write\_consumed > 0) {
2. **if** (bwr.write\_consumed < (ssize\_t)mOut.dataSize())
3. mOut.remove(0, bwr.write\_consumed);
4. **else**
5. mOut.setDataSize(0);
6. }

       接着是设置从Binder驱动程序读取的内容：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **if** (bwr.read\_consumed > 0) {
2. mIn.setDataSize(bwr.read\_consumed);
3. mIn.setDataPosition(0);
4. }

       然后就返回到IPCThreadState::waitForResponse去了。IPCThreadState::waitForResponse函数的处理也很简单，就是处理刚才从Binder驱动程序读入内容了。从前面的分析中，我们知道，从Binder驱动程序读入的内容就是两个整数了，分别是BR\_NOOP和BR\_TRANSACTION\_COMPLETE。对BR\_NOOP的处理很简单，正如它的名字所示，什么也不做；而对BR\_TRANSACTION\_COMPLETE的处理，就分情况了，如果这个请求是异步的，那个整个BC\_TRANSACTION操作就完成了，如果这个请求是同步的，即要等待回复的，也就是reply不为空，那么还要继续通过IPCThreadState::talkWithDriver进入到Binder驱动程序中去等待BC\_TRANSACTION操作的处理结果。

      这里属于后一种情况，于是再次通过IPCThreadState::talkWithDriver进入到Binder驱动程序的binder\_ioctl函数中。不过这一次在binder\_ioctl函数中，bwr.write\_size等于0，而bwr.read\_size大于0，于是再次进入到binder\_thread\_read函数中。这时候thread->transaction\_stack仍然不为NULL，不过thread->todo队列已经为空了，因为前面我们已经处理过thread->todo队列的内容了，于是就通过下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

      进入休眠状态了，等待Service Manager的唤醒。

      现在，我们终于可以回到Service Manager被唤醒之后的过程了。前面我们说过，Service Manager此时正在binder\_thread\_read函数中休眠中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. t->saved\_priority = task\_nice(current);
70. **if** (t->priority < target\_node->min\_priority &&
71. !(t->flags & TF\_ONE\_WAY))
72. binder\_set\_nice(t->priority);
73. **else** **if** (!(t->flags & TF\_ONE\_WAY) ||
74. t->saved\_priority > target\_node->min\_priority)
75. binder\_set\_nice(target\_node->min\_priority);
76. cmd = BR\_TRANSACTION;
77. } **else** {
78. ......
79. }
80. tr.code = t->code;
81. tr.flags = t->flags;
82. tr.sender\_euid = t->sender\_euid;
84. **if** (t->from) {
85. **struct** task\_struct \*sender = t->from->proc->tsk;
86. tr.sender\_pid = task\_tgid\_nr\_ns(sender, current->nsproxy->pid\_ns);
87. } **else** {
88. ......
89. }
91. tr.data\_size = t->buffer->data\_size;
92. tr.offsets\_size = t->buffer->offsets\_size;
93. tr.data.ptr.buffer = (**void** \*)t->buffer->data + proc->user\_buffer\_offset;
94. tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data\_size, **sizeof**(**void** \*));
96. **if** (put\_user(cmd, (uint32\_t \_\_user \*)ptr))
97. **return** -EFAULT;
98. ptr += **sizeof**(uint32\_t);
99. **if** (copy\_to\_user(ptr, &tr, **sizeof**(tr)))
100. **return** -EFAULT;
101. ptr += **sizeof**(tr);
103. ......
105. list\_del(&t->work.entry);
106. t->buffer->allow\_user\_free = 1;
107. **if** (cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)) {
108. t->to\_parent = **thread**->transaction\_stack;
109. t->to\_thread = **thread**;
110. **thread**->transaction\_stack = t;
111. } **else** {
112. ......
113. }
114. **break**;
115. }
117. done:
119. \*consumed = ptr - buffer;
120. ......
121. **return** 0;
122. }

        这里就是从语句中唤醒了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. ret = wait\_event\_interruptible\_exclusive(proc->wait, binder\_has\_proc\_work(proc, **thread**));

        Service Manager唤醒过来看，继续往下执行，进入到while循环中。首先是从proc->todo中取回待处理事项w。这个事项w的类型是BINDER\_WORK\_TRANSACTION，这是上面调用binder\_transaction的时候设置的，于是通过w得到待处理事务t：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

        接下来的内容，就把cmd和t->buffer的内容拷贝到用户传进来的缓冲区去了，这里就是Service Manager从用户空间传进来的缓冲区了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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);

        注意，这里先是把t->buffer的内容拷贝到本地变量tr中，再拷贝到用户空间缓冲区去。关于t->buffer内容的拷贝，请参考[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)一文，它的一个关键地方是Binder驱动程序和Service Manager守护进程共享了同一个物理内存的内容，拷贝的只是这个物理内存在用户空间的虚拟地址回去：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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** \*));

       对于Binder驱动程序这次操作来说，这个事项就算是处理完了，就要从todo队列中删除了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. list\_del(&t->work.entry);

       紧接着，还不放删除这个事务，因为它还要等待Service Manager处理完成后，再进一步处理，因此，放在thread->transaction\_stack队列中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

       还要注意的一个地方是，上面写入的cmd = BR\_TRANSACTION，告诉Service Manager守护进程，它要做什么事情，后面我们会看到相应的分析。

       这样，binder\_thread\_read函数就处理完了，回到binder\_ioctl函数中，同样是操作结果写回到用户空间的缓冲区中去：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

       最后，就返回到frameworks/base/cmds/servicemanager/binder.c文件中的binder\_loop函数去了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. }

        这里就是从下面的语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);

        返回来了。接着就进入binder\_parse函数处理从Binder驱动程序里面读取出来的数据：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. **switch**(cmd) {
10. ......
11. **case** BR\_TRANSACTION: {
12. **struct** binder\_txn \*txn = (**void** \*) ptr;
13. ......
14. **if** (func) {
15. unsigned rdata[256/4];
16. **struct** binder\_io msg;
17. **struct** binder\_io reply;
18. **int** res;
20. bio\_init(&reply, rdata, **sizeof**(rdata), 4);
21. bio\_init\_from\_txn(&msg, txn);
22. res = func(bs, txn, &msg, &reply);
23. binder\_send\_reply(bs, &reply, txn->data, res);
24. }
25. ptr += **sizeof**(\*txn) / **sizeof**(uint32\_t);
26. **break**;
27. }
28. ......
29. **default**:
30. LOGE("parse: OOPS %d\n", cmd);
31. **return** -1;
32. }
33. }
35. **return** r;
36. }

         前面我们说过，Binder驱动程序写入到用户空间的缓冲区中的cmd为BR\_TRANSACTION，因此，这里我们只关注BR\_TRANSACTION相关的逻辑。

         这里用到的两个数据结构struct binder\_txn和struct binder\_io可以参考前面一篇文章[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)，这里就不复述了。

         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/6633311)

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/6633311)

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/6633311)

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. //    LOGI("target=%p code=%d pid=%d uid=%d\n",
13. //         txn->target, txn->code, txn->sender\_pid, txn->sender\_euid);
15. **if** (txn->target != svcmgr\_handle)
16. **return** -1;
18. // Equivalent to Parcel::enforceInterface(), reading the RPC
19. // header with the strict mode policy mask and the interface name.
20. // Note that we ignore the strict\_policy and don't propagate it
21. // further (since we do no outbound RPCs anyway).
22. strict\_policy = bio\_get\_uint32(msg);
23. s = bio\_get\_string16(msg, &len);
24. **if** ((len != (**sizeof**(svcmgr\_id) / 2)) ||
25. memcmp(svcmgr\_id, s, **sizeof**(svcmgr\_id))) {
26. fprintf(stderr,"invalid id %s\n", str8(s));
27. **return** -1;
28. }
30. **switch**(txn->code) {
31. **case** SVC\_MGR\_GET\_SERVICE:
32. **case** SVC\_MGR\_CHECK\_SERVICE:
33. s = bio\_get\_string16(msg, &len);
34. ptr = do\_find\_service(bs, s, len);
35. **if** (!ptr)
36. **break**;
37. bio\_put\_ref(reply, ptr);
38. **return** 0;
40. ......
41. }
42. **default**:
43. LOGE("unknown code %d\n", txn->code);
44. **return** -1;
45. }
47. bio\_put\_uint32(reply, 0);
48. **return** 0;
49. }

        这里， Service Manager要处理的code是SVC\_MGR\_CHECK\_SERVICE，这是在前面的BpServiceManager::checkService函数里面设置的。

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

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

       这里的语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

       其中，会验证一下传进来的第二个参数，即"android.os.IServiceManager"是否正确，这个是验证RPC头，注释已经说得很清楚了。

       最后，就是调用do\_find\_service函数查找是存在名称为"media.player"的服务了。回忆一下前面一篇文章[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)，MediaPlayerService已经把一个名称为"media.player"的服务注册到Service Manager中，所以这里一定能找到。我们看看do\_find\_service这个函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **void** \*do\_find\_service(**struct** binder\_state \*bs, uint16\_t \*s, unsigned len)
2. {
3. **struct** svcinfo \*si;
4. si = find\_svc(s, len);
6. //    LOGI("check\_service('%s') ptr = %p\n", str8(s), si ? si->ptr : 0);
7. **if** (si && si->ptr) {
8. **return** si->ptr;
9. } **else** {
10. **return** 0;
11. }
12. }

       这里又调用了find\_svc函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **struct** svcinfo \*find\_svc(uint16\_t \*s16, unsigned len)
2. {
3. **struct** svcinfo \*si;
5. **for** (si = svclist; si; si = si->next) {
6. **if** ((len == si->len) &&
7. !memcmp(s16, si->name, len \* **sizeof**(uint16\_t))) {
8. **return** si;
9. }
10. }
11. **return** 0;
12. }

       就是在svclist列表中查找对应名称的svcinfo了。

       然后返回到do\_find\_service函数中。回忆一下前面一篇文章[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)，这里的si->ptr就是指MediaPlayerService这个Binder实体在Service Manager进程中的句柄值了。

       回到svcmgr\_handler函数中，调用bio\_put\_ref函数将这个Binder引用写回到reply参数。我们看看bio\_put\_ref的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **void** bio\_put\_ref(**struct** binder\_io \*bio, **void** \*ptr)
2. {
3. **struct** binder\_object \*obj;
5. **if** (ptr)
6. obj = bio\_alloc\_obj(bio);
7. **else**
8. obj = bio\_alloc(bio, **sizeof**(\*obj));
10. **if** (!obj)
11. **return**;
13. obj->flags = 0x7f | FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS;
14. obj->type = BINDER\_TYPE\_HANDLE;
15. obj->pointer = ptr;
16. obj->cookie = 0;
17. }

        这里很简单，就是把一个类型为BINDER\_TYPE\_HANDLE的binder\_object写入到reply缓冲区中去。这里的binder\_object就是相当于是flat\_binder\_obj了，具体可以参考[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)一文。

        再回到svcmgr\_handler函数中，最后，还写入一个0值到reply缓冲区中，表示操作结果码：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. bio\_put\_uint32(reply, 0);

        最后返回到binder\_parse函数中，调用binder\_send\_reply函数将操作结果反馈给Binder驱动程序：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. }

        注意，这里的status参数为0。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Binder驱动程序，它的SVC\_MGR\_CHECK\_SERVICE操作已经完成了,要查询的服务的句柄值也是保存在data.txn.data，操作结果码是0，也是保存在data.txn.data中。  
        再来看binder\_write函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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，首先是执行BC\_FREE\_BUFFER命令，这个命令的执行在前面一篇文章[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)已经介绍过了，这里就不再累述了。

        我们重点关注BC\_REPLY命令的执行：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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/6633311)

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. ......
27. **thread**->transaction\_stack = in\_reply\_to->to\_parent;
28. target\_thread = in\_reply\_to->from;
29. ......
30. target\_proc = target\_thread->proc;
31. } **else** {
32. ......
33. }
34. **if** (target\_thread) {
35. e->to\_thread = target\_thread->pid;
36. target\_list = &target\_thread->todo;
37. target\_wait = &target\_thread->wait;
38. } **else** {
39. ......
40. }

43. /\* TODO: reuse incoming transaction for reply \*/
44. t = kzalloc(**sizeof**(\*t), GFP\_KERNEL);
45. **if** (t == NULL) {
46. return\_error = BR\_FAILED\_REPLY;
47. **goto** err\_alloc\_t\_failed;
48. }
49. binder\_stats.obj\_created[BINDER\_STAT\_TRANSACTION]++;
51. tcomplete = kzalloc(**sizeof**(\*tcomplete), GFP\_KERNEL);
52. **if** (tcomplete == NULL) {
53. return\_error = BR\_FAILED\_REPLY;
54. **goto** err\_alloc\_tcomplete\_failed;
55. }
56. ......
58. **if** (!reply && !(tr->flags & TF\_ONE\_WAY))
59. t->from = **thread**;
60. **else**
61. t->from = NULL;
62. t->sender\_euid = proc->tsk->cred->euid;
63. t->to\_proc = target\_proc;
64. t->to\_thread = target\_thread;
65. t->code = tr->code;
66. t->flags = tr->flags;
67. t->priority = task\_nice(current);
68. t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,
69. tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));
70. **if** (t->buffer == NULL) {
71. return\_error = BR\_FAILED\_REPLY;
72. **goto** err\_binder\_alloc\_buf\_failed;
73. }
74. t->buffer->allow\_user\_free = 0;
75. t->buffer->debug\_id = t->debug\_id;
76. t->buffer->transaction = t;
77. t->buffer->target\_node = target\_node;
78. **if** (target\_node)
79. binder\_inc\_node(target\_node, 1, 0, NULL);
81. offp = (**size\_t** \*)(t->buffer->data + ALIGN(tr->data\_size, **sizeof**(**void** \*)));
83. **if** (copy\_from\_user(t->buffer->data, tr->data.ptr.buffer, tr->data\_size)) {
84. binder\_user\_error("binder: %d:%d got transaction with invalid "
85. "data ptr\n", proc->pid, **thread**->pid);
86. return\_error = BR\_FAILED\_REPLY;
87. **goto** err\_copy\_data\_failed;
88. }
89. **if** (copy\_from\_user(offp, tr->data.ptr.offsets, tr->offsets\_size)) {
90. binder\_user\_error("binder: %d:%d got transaction with invalid "
91. "offsets ptr\n", proc->pid, **thread**->pid);
92. return\_error = BR\_FAILED\_REPLY;
93. **goto** err\_copy\_data\_failed;
94. }
95. ......
97. off\_end = (**void** \*)offp + tr->offsets\_size;
98. **for** (; offp < off\_end; offp++) {
99. **struct** flat\_binder\_object \*fp;
100. ......
101. fp = (**struct** flat\_binder\_object \*)(t->buffer->data + \*offp);
102. **switch** (fp->type) {
103. ......
104. **case** BINDER\_TYPE\_HANDLE:
105. **case** BINDER\_TYPE\_WEAK\_HANDLE: {
106. **struct** binder\_ref \*ref = binder\_get\_ref(proc, fp->handle);
107. **if** (ref == NULL) {
108. ......
109. return\_error = BR\_FAILED\_REPLY;
110. **goto** err\_binder\_get\_ref\_failed;
111. }
112. **if** (ref->node->proc == target\_proc) {
113. ......
114. } **else** {
115. **struct** binder\_ref \*new\_ref;
116. new\_ref = binder\_get\_ref\_for\_node(target\_proc, ref->node);
117. **if** (new\_ref == NULL) {
118. return\_error = BR\_FAILED\_REPLY;
119. **goto** err\_binder\_get\_ref\_for\_node\_failed;
120. }
121. fp->handle = new\_ref->desc;
122. binder\_inc\_ref(new\_ref, fp->type == BINDER\_TYPE\_HANDLE, NULL);
123. ......
124. }
125. } **break**;
127. ......
128. }
129. }
131. **if** (reply) {
132. BUG\_ON(t->buffer->async\_transaction != 0);
133. binder\_pop\_transaction(target\_thread, in\_reply\_to);
134. } **else** **if** (!(t->flags & TF\_ONE\_WAY)) {
135. ......
136. } **else** {
137. ......
138. }
140. t->work.type = BINDER\_WORK\_TRANSACTION;
141. list\_add\_tail(&t->work.entry, target\_list);
142. tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;
143. list\_add\_tail(&tcomplete->entry, &**thread**->todo);
144. **if** (target\_wait)
145. wake\_up\_interruptible(target\_wait);
146. **return**;
148. ......
149. }

        这次进入binder\_transaction函数的情形和上面介绍的binder\_transaction函数的情形基本一致，只是这里的proc、thread和target\_proc、target\_thread调换了角色，这里的proc和thread指的是Service Manager进程，而target\_proc和target\_thread指的是刚才请求SVC\_MGR\_CHECK\_SERVICE的进程。

        那么，这次是如何找到target\_proc和target\_thread呢。首先，我们注意到，这里的reply等于1，其次，上面我们提到，Binder驱动程序在唤醒Service Manager，告诉它有一个事务t要处理时，事务t虽然从Service Manager的todo队列中删除了，但是仍然保留在transaction\_stack中。因此，这里可以从thread->transaction\_stack找回这个等待回复的事务t，然后通过它找回target\_proc和target\_thread：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. in\_reply\_to = **thread**->transaction\_stack;
2. target\_thread = in\_reply\_to->from;
3. target\_list = &target\_thread->todo;
4. target\_wait = &target\_thread->wait;

       再接着往下看，由于Service Manager返回来了一个Binder引用，所以这里要处理一下，就是中间的for循环了。这是一个BINDER\_TYPE\_HANDLE类型的Binder引用，这是前面设置的。先把t->buffer->data的内容转换为一个struct flat\_binder\_object对象fp，这里的fp->handle值就是这个Service在Service Manager进程里面的引用值了。接通过调用binder\_get\_ref函数得到Binder引用对象struct binder\_ref类型的对象ref：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **struct** binder\_ref \*ref = binder\_get\_ref(proc, fp->handle);

       这里一定能找到，因为前面MediaPlayerService执行IServiceManager::addService的时候把自己添加到Service Manager的时候，会在Service Manager进程中创建这个Binder引用，然后把这个Binder引用的句柄值返回给Service Manager用户空间。

       这里面的ref->node->proc不等于target\_proc，因为这个Binder实体是属于创建MediaPlayerService的进程的，而不是请求这个服务的远程接口的进程的，因此，这里调用binder\_get\_ref\_for\_node函数为这个Binder实体在target\_proc创建一个引用：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **struct** binder\_ref \*new\_ref;
2. new\_ref = binder\_get\_ref\_for\_node(target\_proc, ref->node);

       然后增加引用计数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. binder\_inc\_ref(new\_ref, fp->type == BINDER\_TYPE\_HANDLE, NULL);

      这样，返回数据中的Binder对象就处理完成了。注意，这里会把fp->handle的值改为在target\_proc中的引用值：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. fp->handle = new\_ref->desc;

     这里就相当于是把t->buffer->data里面的Binder对象的句柄值改写了。因为这是在另外一个不同的进程里面的Binder引用，所以句柄值当然要用新的了。这个值最终是要拷贝回target\_proc进程的用户空间去的。

      再往下看：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **if** (reply) {
2. BUG\_ON(t->buffer->async\_transaction != 0);
3. binder\_pop\_transaction(target\_thread, in\_reply\_to);
4. } **else** **if** (!(t->flags & TF\_ONE\_WAY)) {
5. ......
6. } **else** {
7. ......
8. }

       这里reply等于1，执行binder\_pop\_transaction函数把当前事务in\_reply\_to从target\_thread->transaction\_stack队列中删掉，这是上次调用binder\_transaction函数的时候设置的，现在不需要了，所以把它删掉。

       再往后的逻辑就跟前面执行binder\_transaction函数时候一样了，这里不再介绍。最后的结果就是唤醒请求SVC\_MGR\_CHECK\_SERVICE操作的线程：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

       这样，Service Manger回复调用SVC\_MGR\_CHECK\_SERVICE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/6633311)

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/6633311)

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。  
      上面讲到调用请求SVC\_MGR\_CHECK\_SERVICE操作的线程被唤醒了，于是，重新执行binder\_thread\_read函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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. }

        就是从下面这个调用：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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

       被唤醒过来了。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返回来了一个Binder引用和一个结果码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数据了。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/6633311)

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/6633311)

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。接着就把结果保存在reply了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

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**);

       我们简单看一下Parcel::ipcSetDataReference函数的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **void** Parcel::ipcSetDataReference(**const** uint8\_t\* data, **size\_t** dataSize,
2. **const** **size\_t**\* objects, **size\_t** objectsCount, release\_func relFunc, **void**\* relCookie)
3. {
4. freeDataNoInit();
5. mError = NO\_ERROR;
6. mData = **const\_cast**<uint8\_t\*>(data);
7. mDataSize = mDataCapacity = dataSize;
8. //LOGI("setDataReference Setting data size of %p to %lu (pid=%d)\n", this, mDataSize, getpid());
9. mDataPos = 0;
10. LOGV("setDataReference Setting data pos of %p to %d\n", **this**, mDataPos);
11. mObjects = **const\_cast**<**size\_t**\*>(objects);
12. mObjectsSize = mObjectsCapacity = objectsCount;
13. mNextObjectHint = 0;
14. mOwner = relFunc;
15. mOwnerCookie = relCookie;
16. scanForFds();
17. }

        上面提到，返回来的数据中有一个Binder引用，因此，这里的mObjectSize等于1，这个Binder引用对应的位置记录在mObjects成员变量中。

        从这里层层返回，最后回到BpServiceManager::checkService函数中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. **virtual** sp<IBinder> BpServiceManager::checkService( **const** String16& name) **const**
2. {
3. Parcel data, reply;
4. data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
5. data.writeString16(name);
6. remote()->transact(CHECK\_SERVICE\_TRANSACTION, data, &reply);
7. **return** reply.readStrongBinder();
8. }

        这里就是从：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. remote()->transact(CHECK\_SERVICE\_TRANSACTION, data, &reply);

        返回来了。我们接着看一下reply.readStrongBinder函数的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. sp<IBinder> Parcel::readStrongBinder() **const**
2. {
3. sp<IBinder> val;
4. unflatten\_binder(ProcessState::self(), \***this**, &val);
5. **return** val;
6. }

        这里调用了unflatten\_binder函数来构造一个Binder对象：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. status\_t unflatten\_binder(**const** sp<ProcessState>& proc,
2. **const** Parcel& in, sp<IBinder>\* out)
3. {
4. **const** flat\_binder\_object\* flat = in.readObject(**false**);
6. **if** (flat) {
7. **switch** (flat->type) {
8. **case** BINDER\_TYPE\_BINDER:
9. \*out = **static\_cast**<IBinder\*>(flat->cookie);
10. **return** finish\_unflatten\_binder(NULL, \*flat, in);
11. **case** BINDER\_TYPE\_HANDLE:
12. \*out = proc->getStrongProxyForHandle(flat->handle);
13. **return** finish\_unflatten\_binder(
14. **static\_cast**<BpBinder\*>(out->get()), \*flat, in);
15. }
16. }
17. **return** BAD\_TYPE;
18. }

        这里的flat->type是BINDER\_TYPE\_HANDLE，因此调用ProcessState::getStrongProxyForHandle函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. sp<IBinder> ProcessState::getStrongProxyForHandle(int32\_t handle)
2. {
3. sp<IBinder> result;
5. AutoMutex \_l(mLock);
7. handle\_entry\* e = lookupHandleLocked(handle);
9. **if** (e != NULL) {
10. // We need to create a new BpBinder if there isn't currently one, OR we
11. // are unable to acquire a weak reference on this current one.  See comment
12. // in getWeakProxyForHandle() for more info about this.
13. IBinder\* b = e->binder;
14. **if** (b == NULL || !e->refs->attemptIncWeak(**this**)) {
15. b = **new** BpBinder(handle);
16. e->binder = b;
17. **if** (b) e->refs = b->getWeakRefs();
18. result = b;
19. } **else** {
20. // This little bit of nastyness is to allow us to add a primary
21. // reference to the remote proxy when this team doesn't have one
22. // but another team is sending the handle to us.
23. result.force\_set(b);
24. e->refs->decWeak(**this**);
25. }
26. }
28. **return** result;
29. }

       这里我们可以看到，ProcessState会把使用过的Binder远程接口（BpBinder）缓存起来，这样下次从Service Manager那里请求得到相同的句柄（Handle）时就可以直接返回这个Binder远程接口了，不用再创建一个出来。这里是第一次使用，因此，e->binder为空，于是创建了一个BpBinder对象：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. b = **new** BpBinder(handle);
2. e->binder = b;
3. **if** (b) e->refs = b->getWeakRefs();
4. result = b;

       最后，函数返回到IMediaDeathNotifier::getMediaPlayerService这里，从这个语句返回：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. binder = sm->getService(String16("media.player"));

        这里，就相当于是：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. binder = **new** BpBinder(handle);

        最后，函数调用：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. sMediaPlayerService = interface\_cast<IMediaPlayerService>(binder);

        到了这里，我们可以参考一下前面一篇文章[浅谈Android系统进程间通信（IPC）机制Binder中的Server和Client获得Service Manager](http://blog.csdn.net/luoshengyang/article/details/6627260)，就会知道，这里的interface\_cast实际上最终调用了IMediaPlayerService::asInterface函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. android::sp<IMediaPlayerService> IMediaPlayerService::asInterface(**const** android::sp<android::IBinder>& obj)
2. {
3. android::sp<IServiceManager> intr;
4. **if** (obj != NULL) {
5. intr = **static\_cast**<IMediaPlayerService\*>(
6. obj->queryLocalInterface(IMediaPlayerService::descriptor).get());
7. **if** (intr == NULL) {
8. intr = **new** BpMediaPlayerService(obj);
9. }
10. }
11. **return** intr;
12. }

        这里的obj就是BpBinder，而BpBinder::queryLocalInterface返回NULL，因此就创建了一个BpMediaPlayerService对象：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6633311)

1. intr = **new** BpMediaPlayerService(**new** BpBinder(handle));

        因此，我们最终就得到了一个BpMediaPlayerService对象，达到我们最初的目标。

        有了这个BpMediaPlayerService这个远程接口之后，MediaPlayer就可以调用MediaPlayerService的服务了。

        至此，Android系统进程间通信（IPC）机制Binder中的Client如何通过Service Manager的getService函数获得Server远程接口的过程就分析完了，Binder机制的学习就暂告一段落了。

        不过，细心的读者可能会发现，我们这里介绍的Binder机制都是基于C/C++语言实现的，但是我们在编写应用程序都是基于Java语言的，那么，我们如何使用Java语言来使用系统的Binder机制来进行进程间通信呢？这就是下一篇文章要介绍的内容了，敬请关注。