

Light.Moon

~( ^ ^ )~ 三月学长的根据地 ( ^ ^ )~



# Android Binder 分析——通信模型

By Mingming

© 2015 1月 28 更新日期:2017 2月 7

通信就要搞一些协议，binder 的比较简单，但是也有一个基本的模型，这里以最基本的一次 IPC 调用来说明一下。然后涉及的代码主要在（这里不列 java 层的代码了，java 层的代码前面原理篇分析过了，主要是挂马甲调用 native 的方法的）：

文章目录

1. 流程
2. 1、服务端等待请求
3. 2. 客户端发起 IPC 请求
4. 3. 服务端处理请求，并返回结果
5. 4. 客户端接收到服务端返回的数据
6. 总结

```
1  # native binder 头文件
2  frameworks/native/include/binder
3  # native binder 实现
4  frameworks/native/libs/binder
5
6  # kernel binder 驱动
7  kernel/drivers/staging/android/binder.h
8  kernel/drivers/staging/android/binder.c
9
```

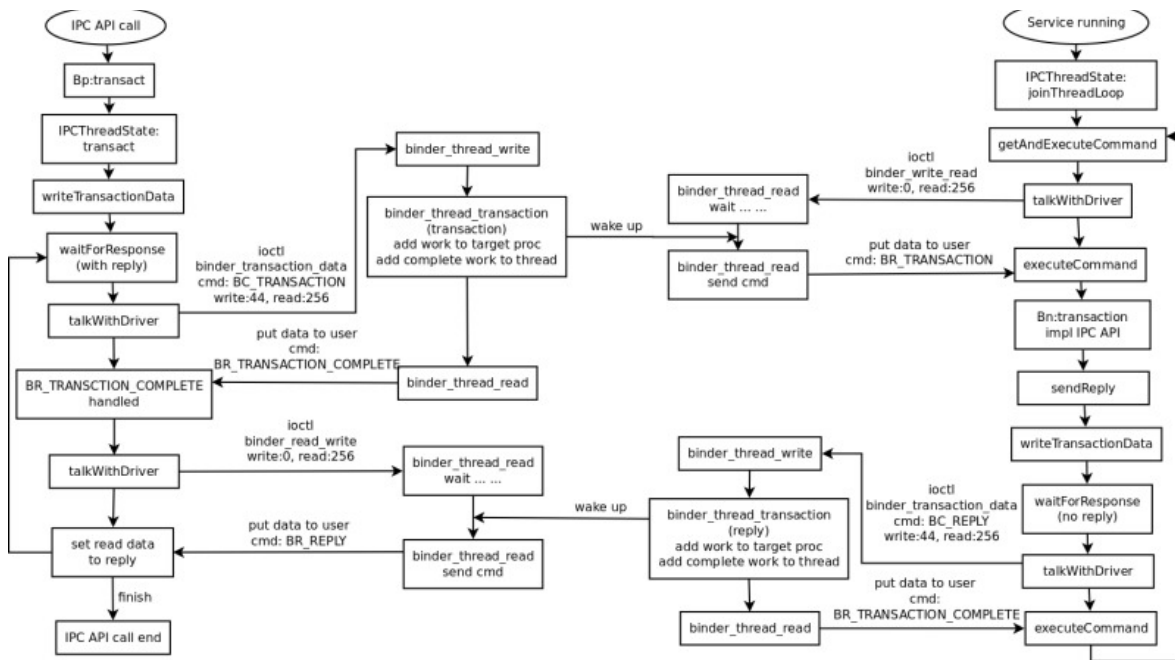
## 流程

先上张图先，图我尽量简化，只画了 IPC 调用相关的东西（是 4.4 来分析的，之前的版本有点点小区别）：

Client(proc)

Binder(Kernel)

Service(proc)



## 1、服务端等待请求

首先看图上右边的服务（service）部分。service 进程运行起来，然后通过调用 IPCThreadState 的 joinThreadLoop 在本线程中开始等待客户端请求的到来。这里有2个问题：第一个，system service 必须向 service manager 注册自己；第二个，服务端的多线程支持问题。这里都后面再说。

```

1  void IPCThreadState::joinThreadLoop(bool isMain)
2  {
3      LOG_THREADPOOL("**** THREAD %p (PID %d) IS JOINING THE THREA
4
5      mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPE
6
7      // This thread may have been spawned by a thread that was in
8      // scheduling group, so first we will make sure it is in the
9      // one to avoid performing an initial transaction in the bac
10     set_sched_policy(mMyThreadId, SP_FOREGROUND);
11
12     status_t result;
13     do {
14         processPendingDerefs();
15         // now get the next command to be processed, waiting if
16         result = getAndExecuteCommand();
17
18         if (result < NO_ERROR && result != TIMED_OUT && result !

```

```

19         ALOGE("getAndExecuteCommand(fd=%d) returned unexpect
20               mProcess->mDriverFD, result);
21         abort();
22     }
23
24     // Let this thread exit the thread pool if it is no long
25     // needed and it is not the main process thread.
26     if(result == TIMED_OUT && !isMain) {
27         break;
28     }
29     } while (result != -ECONNREFUSED && result != -EBADF);
30
31     LOG_THREADPOOL("**** THREAD %p (PID %d) IS LEAVING THE THREA
32                   (void*)pthread_self(), getpid(), (void*)result);
33
34     mOut.writeInt32(BC_EXIT_LOOPER);
35     talkWithDriver(false);
36 }
37

```

这里看得出，这个函数是个循环（等待——处理——等待 ... 一般服务器的模型都是这样），参数 isMain 表示这个 service thread 是不是主线程，这个东西后面再说。循环里面主要是调用 getAndExecuteCommand 来处理。

```

1  status_t IPCThreadState::getAndExecuteCommand()
2  {
3      status_t result;
4      int32_t cmd;
5
6      result = talkWithDriver();
7      if (result >= NO_ERROR) {
8          size_t IN = mIn.dataAvail();
9          if (IN < sizeof(int32_t)) return result;
10         cmd = mIn.readInt32();
11         IF_LOG_COMMANDS() {
12             alog << "Processing top-level Command: "
13                 << getReturnString(cmd) << endl;
14         }
15
16         result = executeCommand(cmd);
17
18         // After executing the command, ensure that the thread i
19         // foreground cgroup before rejoining the pool. The dri
20         // restoring the priority, but doesn't do anything with
21         // need to take care of that here in userspace. Note th
22         // sure to go in the foreground after executing a transa

```

```

23         // there are other callbacks into user code that could h
24         // our group so we want to make absolutely sure it is pu
25         set_sched_policy(mMyThreadId, SP_FOREGROUND);
26     }
27
28     return result;
29 }
30

```

getAndExecuteCommand 这个名字就可以看得出这个函数主要干2个事情，一个取请求数据、一个就是处理请求数据：正好又分了2个函数：talkWithDriver 和 executeCommand（下面那个好像是设置线程优先级，先不理这些东西先）。

先来看下 talkWithDriver：

```

1  status_t IPCThreadState::talkWithDriver(bool doReceive)
2  {
3      if (mProcess->mDriverFD <= 0) {
4          return -EBADF;
5      }
6
7      binder_write_read bwr;
8
9      // Is the read buffer empty?
10     const bool needRead = mIn.dataPosition() >= mIn.dataSize();
11
12     // We don't want to write anything if we are still reading
13     // from data left in the input buffer and the caller
14     // has requested to read the next data.
15     const size_t outAvail = (!doReceive || needRead) ? mOut.dat
16
17     bwr.write_size = outAvail;
18     bwr.write_buffer = (long unsigned int)mOut.data();
19
20     // This is what we'll read.
21     if (doReceive && needRead) {
22         bwr.read_size = mIn.dataCapacity();
23         bwr.read_buffer = (long unsigned int)mIn.data();
24     } else {
25         bwr.read_size = 0;
26         bwr.read_buffer = 0;
27     }
28
29     IF_LOG_COMMANDS() {

```

```

30         TextOutput::Bundle _b(aolog);
31         if (outAvail != 0) {
32             alog << "Sending commands to driver: " << indent;
33             const void* cmds = (const void*)bwr.write_buffer;
34             const void* end = ((const uint8_t*)cmds)+bwr.write_
35             alog << HexDump(cmds, bwr.write_size) << endl;
36             while (cmds < end) cmds = printCommand(aolog, cmds);
37             alog << dedent;
38         }
39         alog << "Size of receive buffer: " << bwr.read_size
40             << ", needRead: " << needRead << ", doReceive: " <<
41     }
42
43     // Return immediately if there is nothing to do.
44     if ((bwr.write_size == 0) && (bwr.read_size == 0)) return 0;
45
46     bwr.write_consumed = 0;
47     bwr.read_consumed = 0;
48     status_t err;
49     do {
50         IF_LOG_COMMANDS() {
51             alog << "About to read/write, write size = " << mOut
52         }
53 #if defined(HAVE_ANDROID_OS)
54         if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr)
55             err = NO_ERROR;
56         else
57             err = -errno;
58     #else
59         err = INVALID_OPERATION;
60     #endif
61         if (mProcess->mDriverFD <= 0) {
62             err = -EBADF;
63         }
64         IF_LOG_COMMANDS() {
65             alog << "Finished read/write, write size = " << mOut
66         }
67     } while (err == -EINTR);
68
69     IF_LOG_COMMANDS() {
70         alog << "Our err: " << (void*)err << ", write consumed:
71             << bwr.write_consumed << " (of " << mOut.dataSize()
72             << "), read consumed: " << bwr.read_consumed
73     }
74
75     if (err >= NO_ERROR) {
76         if (bwr.write_consumed > 0) {
77             if (bwr.write_consumed < (ssize_t)mOut.dataSize())
78                 mOut.remove(0, bwr.write_consumed);

```

```

79         else
80             mOut.setDataSize(0);
81     }
82     if (bwr.read_consumed > 0) {
83         mIn.setDataSize(bwr.read_consumed);
84         mIn.setDataPosition(0);
85     }
86     IF_LOG_COMMANDS() {
87         TextOutput::Bundle _b(aalog);
88         aalog << "Remaining data size: " << mOut.dataSize()
89         aalog << "Received commands from driver: " << indent
90         const void* cmds = mIn.data();
91         const void* end = mIn.data() + mIn.dataSize();
92         aalog << HexDump(cmds, mIn.dataSize()) << endl;
93         while (cmds < end) cmds = printReturnCommand(aalog,
94         aalog << dedent;
95     }
96     return NO_ERROR;
97 }
98
99 return err;
100 }
101

```

talkWithDriver 主要是调用 ioctl 去 kernel 的 binder 设备那里读数据。这里的 mDriverFD 是打开 binder 设备的文件描述符。在 ProcessState 的构造函数中会打开 binder 设备（一个进程只会开打一次，然后所有线程共用一个 fd）：

```

1  ProcessState::ProcessState()
2      : mDriverFD(open_driver())
3      ... ...
4  }
5  static int open_driver()
6  {
7      int fd = open("/dev/binder", O_RDWR);
8      if (fd >= 0) {
9          fcntl(fd, F_SETFD, FD_CLOEXEC);
10         int vers;
11         status_t result = ioctl(fd, BINDER_VERSION, &vers);
12         if (result == -1) {
13             ALOGE("Binder ioctl to obtain version failed: %s", s
14             close(fd);
15             fd = -1;
16         }
17         if (result != 0 || vers != BINDER_CURRENT_PROTOCOL_VERSI
18             ALOGE("Binder driver protocol does not match user sp

```

```

19         close(fd);
20         fd = -1;
21     }
22     size_t maxThreads = 15;
23     result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
24     if (result == -1) {
25         ALOGE("Binder ioctl to set max threads failed: %s",
26             strerror(errno));
27     } else {
28         ALOGW("Opening '/dev/binder' failed: %s\n", strerror(errno));
29     }
30     return fd;
31 }
32

```

回到 talkWithDriver, ioctl 调用的命令是 `BINDER_WRITE_READ`，参数是 `binder_write_read` 这个结构。这里每个 IPCThreadState (service 的线程) 都还有2个 Parcel 变量: mIn、mOut 分别用于打包发送和接收读取的数据。这些后面再分析 binder 的数据包的时候再说。这里只说前面要向 binder 发命令，就调用 mOut 写命令，要接收数据的话，talkWithDriver 的参数 doRevice 默认是 true, ioctl 能一次性完成读、写操作。

这里第一次 service 写是 0byte，读是 256byte (初始化的 mIn 的 Capacity 是 256，但是 service 开始并没有写入命令)。

然后 ioctl 就到了 kernel 的 binder 驱动里面：

```

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;
8
9      /* printk(KERN_INFO "binder_ioctl: %d:%d %x %lx\n", proc->pid,
10         proc->tid, cmd, arg); */
11     ret = wait_event_interruptible(binder_user_error_wait, binder_user_error_wait);
12     if (ret)
13         return ret;
14
15     mutex_lock(&binder_lock);
16     thread = binder_get_thread(proc);

```

```
17     if (thread == NULL) {
18         ret = -ENOMEM;
19         goto err;
20     }
21     switch (cmd) {
22     case BINDER_WRITE_READ: {
23         struct binder_write_read bwr;
24         if (size != sizeof(struct binder_write_read)) {
25             ret = -EINVAL;
26             goto err;
27         }
28         if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
29             ret = -EFAULT;
30             goto err;
31         }
32         binder_debug(BINDER_DEBUG_READ_WRITE,
33                     "binder: %d:%d write %ld at %08lx, read %ld at
34                     proc->pid, thread->pid, bwr.write_size, bwr.wri
35                     bwr.read_size, bwr.read_buffer);
36
37         if (bwr.write_size > 0) {
38             ret = binder_thread_write(proc, thread, (void __user
39             if (ret < 0) {
40                 bwr.read_consumed = 0;
41                 if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
42                     ret = -EFAULT;
43                 goto err;
44             }
45         }
46         if (bwr.read_size > 0) {
47             ret = binder_thread_read(proc, thread, (void __user
48             if (!list_empty(&proc->todo))
49                 wake_up_interruptible(&proc->wait);
50             if (ret < 0) {
51                 if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
52                     ret = -EFAULT;
53                 goto err;
54             }
55         }
56         binder_debug(BINDER_DEBUG_READ_WRITE,
57                     "binder: %d:%d wrote %ld of %ld, read return %l
58                     proc->pid, thread->pid, bwr.write_consumed, bwr
59                     bwr.read_consumed, bwr.read_size);
60         if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
61             ret = -EFAULT;
62             goto err;
63         }
64         break;
65     }
```



```

66
67     ... ..
68
69 }
70

```

关于 ioctl 的一些基本知识可以参看下我的这篇文章：

[\[转\] unlocked\\_ioctl 和堵塞 \(waitqueue\) 读写函数的实现](#)

这里稍微注意下，ioctl 里面 proc 表示本次调用的进程，thread 表示调用本次调用的线程（这2个主要是记录了进程号 pid 和线程号 tid）。然后第一次 write 的 size 是 0，所以没有处理 binder\_thread\_write。接下来是取数据：

binder\_thread\_read（这里可以看到出，kernel 是先处理 write 再处理 read 的，原因到后面就知道了）。

```

1  static int binder_thread_read(struct binder_proc *proc,
2                                struct binder_thread *thread,
3                                void __user *buffer, int size,
4                                signed long *consumed, int non_block)
5  {
6      void __user *ptr = buffer + *consumed;
7      void __user *end = buffer + size;
8
9      int ret = 0;
10     int wait_for_proc_work;
11
12     if (*consumed == 0) {
13         if (put_user(BR_NOOP, (uint32_t __user *)ptr))
14             return -EFAULT;
15         ptr += sizeof(uint32_t);
16     }
17
18     retry:
19     wait_for_proc_work = thread->transaction_stack == NULL &&
20         list_empty(&thread->todo);
21
22     if (thread->return_error != BR_OK && ptr < end) {
23         if (thread->return_error2 != BR_OK) {
24             if (put_user(thread->return_error2, (uint32_t __user *)ptr))
25                 return -EFAULT;
26             ptr += sizeof(uint32_t);
27             if (ptr == end)
28                 goto done;
29             thread->return_error2 = BR_OK;

```

```
30         }
31         if (put_user(thread->return_error, (uint32_t __user *)pt
32             return -EFAULT;
33         ptr += sizeof(uint32_t);
34         thread->return_error = BR_OK;
35         goto done;
36     }
37
38
39     thread->looper |= BINDER_LOOPER_STATE_WAITING;
40     if (wait_for_proc_work)
41         proc->ready_threads++;
42     mutex_unlock(&binder_lock);
43     if (wait_for_proc_work) {
44         if (!(thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
45             BINDER_LOOPER_STATE_ENTERED))) {
46             binder_user_error("binder: %d:%d ERROR: Thread waiti
47                 "for process work before calling BC_REGISTER_"
48                 "LOOPER or BC_ENTER_LOOPER (state %x)\n",
49                 proc->pid, thread->pid, thread->looper);
50             wait_event_interruptible(binder_user_error_wait,
51                 binder_stop_on_user_error < 2);
52         }
53         binder_set_nice(proc->default_priority);
54         if (non_block) {
55             if (!binder_has_proc_work(proc, thread))
56                 ret = -EAGAIN;
57         } else
58             ret = wait_event_interruptible_exclusive(proc->wait,
59     } else {
60         if (non_block) {
61             if (!binder_has_thread_work(thread))
62                 ret = -EAGAIN;
63         } else
64             ret = wait_event_interruptible(thread->wait, binder_
65     }
66     mutex_lock(&binder_lock);
67     if (wait_for_proc_work)
68         proc->ready_threads--;
69     thread->looper &= ~BINDER_LOOPER_STATE_WAITING;
70
71     if (ret)
72         return ret;
73
74     ... ..
75
76 }
77
```

来看下 `binder_thread_read` 的前面部分。这里 service 的 `wait_for_proc_work` 的值为 1，也是 true 的意思。一开始的 `thread->transaction_stack` 是 NULL 的，这个是表示这个线程的传递堆栈，类似于函数调用堆栈的东西，一开始没传递，这个堆栈肯定是空的（这个也后面再具体分析）。然后一开始 `thread->todo` 这链表也是空的，这个是表示这个线程上需要完成的工作，这个后面会知道是什么东西。

那根据上面的条件，这里 service 的 `ioctl` 调用就会跑到这里：

```
wait_event_interruptible_exclusive(proc->wait, binder_has_proc_work
```

`wait event` 是 kernel 里面的等待队列，在前面那篇 `ioctl` 的文章里有详细说明，它现在阻塞于 `proc->wait` 这个变量，如果唤醒 `proc->wait` 阻塞会结束。还有如果后面那个 `binder_has_proc_work` 返回值为 1（条件为 true）阻塞也会结束。但是这里 `binder_has_proc_work` 不为 true：

```
1  // 这个其实就是判断 proc 有没有需要完成的工作（前面那个是判断是 thread 的）
2  static int binder_has_proc_work(struct binder_proc *proc,
3                                  struct binder_thread *thread)
4  {
5      return !list_empty(&proc->todo) ||
6             (thread->looper & BINDER_LOOPER_STATE_NEED_RETURN);
7  }
8
```

所以这里 `binder_thread_read` 要不阻塞，要么跑进来的时候 `proc->todo` 不为空（就是该进程有需要处理的工作），要么就只能等待有人唤醒 `proc->wait`。这里对应图中 service 在等待客户端请求的到来，这个是在 kernel 中使用 `wait queue`（等待队列）实现的。

所以为什么先处理 `binder_thread_write`，因为 `binder_thread_read` 会阻塞。

## 2. 客户端发起 IPC 请求

服务端已经在阻塞等待了，现在来看看客户端（client）这边（图中的左边部

分)。client 调用一个 IPC 接口函数（例如调用 ActivityManager 的 startActivity 之类的），发起 IPC 调用，IPC 接口函数调用 Bp 的 transaction 函数，这些在前面的原理篇里有说过，获取 Bp 是通过 service manager 获取的（前面说 service 要在 service manager 注册就是为了 client 能通过 service manager 获取自己的 Bp），这里细节后面再说。

Bp 的 transaction 直接调用 IPCThreadState 的 transaction 函数（注意别头晕，servie、client 是共用一份代码的，加上前面 ioctl 读、写一个命令 -\_-||）：

```

1  status_t IPCThreadState::transact(int32_t handle,
2                                  uint32_t code, const Parcel& d
3                                  Parcel* reply, uint32_t flags)
4  {
5      status_t err = data.errorCheck();
6
7      flags |= TF_ACCEPT_FDS;
8
9      IF_LOG_TRANSACTIONS() {
10         TextOutput::Bundle _b(alog);
11         alog << "BC_TRANSACTION thr " << (void*)pthread_self() <
12             << handle << " / code " << TypeCode(code) << ": "
13             << indent << data << dedent << endl;
14     }
15
16     if (err == NO_ERROR) {
17         LOG_ONeway(">>>> SEND from pid %d uid %d %s", getpid(),
18             (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY"
19         err = writeTransactionData(BC_TRANSACTION, flags, handle
20     }
21
22     if (err != NO_ERROR) {
23         if (reply) reply->setError(err);
24         return (mLastError = err);
25     }
26
27     if ((flags & TF_ONE_WAY) == 0) {
28         #if 0
29         if (code == 4) { // relayout
30             ALOGI(">>>>> CALLING transaction 4");
31         } else {
32             ALOGI(">>>>> 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         ALOGI("<<<<<< RETURNING transaction 4");
44     } else {
45         ALOGI("<<<<<< RETURNING transaction %d", code);
46     }
47     #endif
48
49     IF_LOG_TRANSACTIONS() {
50         TextOutput::Bundle _b(alog);
51         alog << "BR_REPLY thr " << (void*)pthread_self() <<
52             << handle << ": ";
53         if (reply) alog << indent << *reply << dedent << end
54         else alog << "(none requested)" << endl;
55     }
56 } else {
57     err = waitForResponse(NULL, NULL);
58 }
59
60 return err;
61 }
62

```

这里也是分为2个部分：第一部分：writeTransactionData 向 service 写请求数据，第二部分：waitForResponse，等待 service 返回请求结果。不难理解，一个函数调用，一般都有返回值（就算是 void 的也需要等待 service 返回结果），所以要发起请求后，要等待 service 返回结果。

先来看下写请求数据：

```

1  status_t IPCThreadState::writeTransactionData(int32_t cmd, uint3
2      int32_t handle, uint32_t code, const Parcel& data, status_t*
3  {
4      binder_transaction_data tr;
5
6      tr.target.handle = handle;
7      tr.code = code;
8      tr.flags = binderFlags;
9      tr.cookie = 0;
10     tr.sender_pid = 0;

```

```

11     tr.sender_euid = 0;
12
13     const status_t err = data.errorCheck();
14     if (err == NO_ERROR) {
15         tr.data_size = data.ipcDataSize();
16         tr.data.ptr.buffer = data.ipcData();
17         tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
18         tr.data.ptr.offsets = data.ipcObjects();
19     } else if (statusBuffer) {
20         tr.flags |= TF_STATUS_CODE;
21         *statusBuffer = err;
22         tr.data_size = sizeof(status_t);
23         tr.data.ptr.buffer = statusBuffer;
24         tr.offsets_size = 0;
25         tr.data.ptr.offsets = NULL;
26     } else {
27         return (mLastError = err);
28     }
29
30     mOut.writeInt32(cmd);
31     mOut.write(&tr, sizeof(tr));
32
33     return NO_ERROR;
34 }
35

```

传过去的 cmd (命令) 是 `BC_TRANSACTION` 这个要记住。这里把上层 app 传递过来的数据 (Parcel 封装的, 主要是 IPC 的参数), 打包到

`binder_transaction_data` 这个数据结构中, 这个是 kernel binder 驱动 ioctl 参数里带的参数 (后面再说)。这里都是通过 mOut Parcel 打包的 (后面再说)。注意这里只是把数据包打好, 还没发送。

然后到 IPCThreadState transaction 的第二部分: waitForResponse 等待 service 返回处理结果。这里注意一下 transaction 有个判断:

```
if ((flags & TF_ONE_WAY) == 0)
```

这个是表示 IPC 调用需要不需要返回, 一般都是需要的, 所以这里 waitForResponse 的参数 reply 不是 NULL, 这个 reply 会返回给 IPC 的调用者, 里来带有返回的数据。

```
1 status_t IPCThreadState::waitForResponse(Parcel *reply, status_t
```

```

2   {
3       int32_t cmd;
4       int32_t err;
5
6       while (1) {
7           if ((err=talkWithDriver()) < NO_ERROR) break;
8
9       ... ..
10
11       }
12
13   }
14

```

这里看到有一个循环，至于为什么，到后面会知道的。循环开始调用 talkWithDriver，这个函数前面在 service 那里看过了，是调用 ioctl 去 binder 那里写数据和读数据。service 那里只是读了，但是没写。client 不一样了，前面 writeTransactionData 把 BC\_TRANSACTION 命令和参数全都打包好了，就通过 ioctl 发到 kernel 的 binder 那去了。

这里 binder\_ioctl 那会跑 binder\_thread\_write 了：

```

1   int binder_thread_write(struct binder_proc *proc, struct binder_
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;
7
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;
22

```

```

23         if (copy_from_user(&tr, ptr, sizeof(tr)))
24             return -EFAULT;
25         ptr += sizeof(tr);
26         binder_transaction(proc, thread, &tr, cmd == BC_REPL
27             break;
28     }
29     ... ..
30     default:
31         printk(KERN_ERR "binder: %d:%d unknown command %d\n"
32             proc->pid, thread->pid, cmd);
33         return -EINVAL;
34     }
35     *consumed = ptr - buffer;
36 }
37 return 0;
38 }
39

```

write 这里是一个 while 循环 ptr 是前面用户空间 (client) 打包传进入来的数据, end 是数据结束的地址。这么搞这一个循环, 其实是因为这个数据是支持打包多条命令的, 虽然这里没体现出来, 但是后面有些地方能体现出来, 例如说释放内存的命令有些时候就是和其其它命令打包在一起传进来的。取完数据, 数据指针会移动, 移动到 end 地址了, 就结束处处理了, 所以可以处理多条命令。感觉 binder 这里经常一次搞多种东西, 让代码上让人感觉怪怪的。主要看用的人了, 因为是支持打包多条命令的。

不过这里先关心我们之前 client 写入的 `BC_TRANSACTION` 命令。看下面的处理。传输命令交由 `binder_transaction` 函数处理:

```

1  static void binder_transaction(struct binder_proc *proc,
2                                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;
16
17     e = binder_transaction_log_add(&binder_transaction_log);
18     e->call_type = reply ? 2 : !(tr->flags & TF_ONE_WAY);
19     e->from_proc = proc->pid;
20     e->from_thread = thread->pid;
21     e->target_handle = tr->target.handle;
22     e->data_size = tr->data_size;
23     e->offsets_size = tr->offsets_size;
24
25     if (reply) {
26         ... ..
27     } else {
28         if (tr->target.handle) {
29             struct binder_ref *ref;
30             ref = binder_get_ref(proc, tr->target.handle);
31             if (ref == NULL) {
32                 binder_user_error("binder: %d:%d got "
33                                     "transaction to invalid handle\n",
34                                     proc->pid, thread->pid);
35                 return_error = BR_FAILED_REPLY;
36                 goto err_invalid_target_handle;
37             }
38             target_node = ref->node;
39         } else {
40             target_node = binder_context_mgr_node;
41             if (target_node == NULL) {
42                 return_error = BR_DEAD_REPLY;
43                 goto err_no_context_mgr_node;
44             }
45         }
46         e->to_node = target_node->debug_id;
47         target_proc = target_node->proc;
48         if (target_proc == NULL) {
49             return_error = BR_DEAD_REPLY;
50             goto err_dead_binder;
51         }
52         if (!(tr->flags & TF_ONE_WAY) && thread->transaction_sta
53             struct binder_transaction *tmp;
54             tmp = thread->transaction_stack;
55             if (tmp->to_thread != thread) {
56                 binder_user_error("binder: %d:%d got new "
57                                     "transaction with bad transaction stack"
58                                     ", transaction %d has target %d:%d\n",
59                                     proc->pid, thread->pid, tmp->debug_id,
60                                     tmp->to_proc ? tmp->to_proc->pid : 0,
61                                     tmp->to_thread ?
62                                     tmp->to_thread->pid : 0);
```

```

63         return_error = BR_FAILED_REPLY;
64         goto err_bad_call_stack;
65     }
66     while (tmp) {
67         if (tmp->from && tmp->from->proc == target_proc)
68             target_thread = tmp->from;
69         tmp = tmp->from_parent;
70     }
71 }
72 }
73 if (target_thread) {
74     e->to_thread = target_thread->pid;
75     target_list = &target_thread->todo;
76     target_wait = &target_thread->wait;
77 } else {
78     target_list = &target_proc->todo;
79     target_wait = &target_proc->wait;
80 }
81 e->to_proc = target_proc->pid;
82
83 ... ..
84
85 }
86

```

先来看看这个函数前面一段，前面有一个是否是 reply 的判断，前面

`BC_TRANSACTION` 命令传过来的是 true，所以走的下面分支。其实这段是查找服务端进程的。tr->target.handle 这个由最开始 client 所持有的 Bp 传进来的，通过这个 kernel binder 驱动可以找对应的 service 的 proc。这里具体后面再说，这里就简单知道通过 handle 找到远程目标 service 的 proc，还有这里走的是下面 target\_proc 的分支，`target_thread` 是后面 service 写返回值用的。还有注意，这里通过找到 `target_proc` 确定了 `target_wait`，回想下前面 service 在 `binder_thread_read` 那里 wait，就是 wait 这个变量。

接下去继续往下看：

```

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     binder_stats_created(BINDER_STAT_TRANSACTION);
8
9     tcomplete = kzalloc(sizeof(*tcomplete), GFP_KERNEL);
10    if (tcomplete == NULL) {
11        return_error = BR_FAILED_REPLY;
12        goto err_alloc_tcomplete_failed;
13    }
14    binder_stats_created(BINDER_STAT_TRANSACTION_COMPLETE);
15
16    t->debug_id = ++binder_last_id;
17    e->debug_id = t->debug_id;
18
19    int tmp_pid = -1;
20    if (target_thread) {
21        tmp_pid = target_thread->pid;
22    }
23
24    ... ...
25
26    if (!reply && !(tr->flags & TF_ONE_WAY))
27        t->from = thread;
28    else
29        t->from = NULL;
30    t->sender_euid = proc->tsk->cred->euid;
31    t->to_proc = target_proc;
32    t->to_thread = target_thread;
33    t->code = tr->code;
34    t->flags = tr->flags;
35    t->priority = task_nice(current);
36    t->buffer = binder_alloc_buf(target_proc, tr->data_size,
37        tr->offsets_size, !reply && (t->flags & TF_ONE_WAY));
38    if (t->buffer == NULL) {
39        return_error = BR_FAILED_REPLY;
40        goto err_binder_alloc_buf_failed;
41    }
42    t->buffer->allow_user_free = 0;
43    t->buffer->debug_id = t->debug_id;
44    t->buffer->transaction = t;
45    t->buffer->target_node = target_node;
46    if (target_node)
47        binder_inc_node(target_node, 1, 0, NULL);
48
49    offp = (size_t *)(t->buffer->data + ALIGN(tr->data_size, si
50
51    if (copy_from_user(t->buffer->data, tr->data.ptr.buffer, tr
52        binder_user_error("binder: %d:%d got transaction with i
53        "data ptr\n", proc->pid, thread->pid);
54    return_error = BR_FAILED_REPLY;
55    goto err_copy_data_failed;

```

```

56     }
57     if (copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_
58         binder_user_error("binder: %d:%d got transaction with i
59         "offsets ptr\n", proc->pid, thread->pid);
60         return_error = BR_FAILED_REPLY;
61         goto err_copy_data_failed;
62     }
63     if (!IS_ALIGNED(tr->offsets_size, sizeof(size_t))) {
64         binder_user_error("binder: %d:%d got transaction with "
65         "invalid offsets size, %zd\n",
66         proc->pid, thread->pid, tr->offsets_size);
67         return_error = BR_FAILED_REPLY;
68         goto err_bad_offset;
69     }
70     off_end = (void *)offp + tr->offsets_size;
71     for (; offp < off_end; offp++) {
72
73     ... ...
74
75     }
76     if (reply) {
77         BUG_ON(t->buffer->async_transaction != 0);
78         binder_pop_transaction(target_thread, in_reply_to);
79     } else if (!(t->flags & TF_ONE_WAY)) {
80         BUG_ON(t->buffer->async_transaction != 0);
81         t->need_reply = 1;
82         t->from_parent = thread->transaction_stack;
83         thread->transaction_stack = t;
84     } else {
85         BUG_ON(target_node == NULL);
86         BUG_ON(t->buffer->async_transaction != 1);
87         if (target_node->has_async_transaction) {
88             target_list = &target_node->async_todo;
89             target_wait = NULL;
90         } else
91             target_node->has_async_transaction = 1;
92     }
93     t->work.type = BINDER_WORK_TRANSACTION;
94     list_add_tail(&t->work.entry, target_list);
95     tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
96     list_add_tail(&tcomplete->entry, &thread->todo);
97     if (target_wait)
98         wake_up_interruptible(target_wait);
99     return;
100

```

这里创建了2个对象：`binder_transaction` 和 `binder_work`，后面的代码

就是填充这2个结构。 `binder_transaction` 主要是把目标 ( `target_proc` 或是 `target_thread` ) 和数据填写好。数据从用户层 `ioctl` 传过来的 `binder_transaction_data` ( 前面 `client writeTransactionData` 那里 ), 把 IPC 的参数打包封装起来了, 这里通过 `copy_from_user` 把这里数据再填写到 `binder_transaction` 带的数据结构的指针中。

下面又有一个 `for` 循环, 这个循环是处理 `Parcel` 保存的 `object` 的数据的, 其实就是 `Bp` 或是 `Bn`, 这个是和 `service manager` 通信用的, 这里先暂时不管。

这里还有一点, 保存了传递堆栈, `!(t->flags & TF_ONE_WAY)` 这里一般 IPC 都是需要返回的, 所以把当前的 `binder_transaction` 保存到 `transaction_stack` 中去了。后面 `service` 写返回值的时候回通过这个找到要返回的目标进程的。

最后, `binder_transaction` 的 `work type` 设置为 `BINDER_WORK_TRANSACTION` 然后 `binder_work` 设置为 `BINDER_WORK_TRANSACTION`, 分别加入到了 `target_list` 中 ( 这个这里是 `target_proc` 的 `todo list`, 也就是 `service` 的进程工作列表 ) 和 `thread` 的 `todo list` ( 也就是 `client` 发起 IPC 调用的线程的 `todo list` )。然后, 如果目标 `service` 进程在等待中 ( 前面第一部分确实在等待 ), 就唤醒它。 ( 回去看看最开始的图稍微好理解些 )

其实到这里, `client` 已经把 IPC 请求发出去了, 而 `service` 那把阻塞在 `binder_thread_read` 那里的也应该唤醒了, 开始执行后面的处理了。不过这里还是继续看 `client` 这边。

`client` 这边 `binder_thread_write` 处理完了, 就到 `binder_thread_read` 了。前面 `service` 那里分析过阻塞的情况 ( 前面说过了 `service`、`client` 代码都是共用的 )。这里因为前面 `binder_thread_write` 那里把一个 `binder_work` 插入到 `thread->todo` 中, 所以这里是不会阻塞的。这里会不会觉得有点奇怪, 应该正常的模型应该是要阻塞等待 `service` 那边返回结果才对, 不过先别着急。慢慢看:

```
1      while (1) {
2          uint32_t cmd;
3          struct binder_transaction_data tr;
```

```

4      struct binder_work *w;
5      struct binder_transaction *t = NULL;
6
7      if (!list_empty(&thread->todo))
8          w = list_first_entry(&thread->todo, struct binder_work
9      else if (!list_empty(&proc->todo) && wait_for_proc_work)
10         w = list_first_entry(&proc->todo, struct binder_work
11     else {
12         if (ptr - buffer == 4 && !(thread->looper & BINDER_L
13             goto retry;
14         break;
15     }
16
17     if (end - ptr < sizeof(tr) + 4)
18         break;
19
20     switch (w->type) {
21     case BINDER_WORK_TRANSACTION: {
22         t = container_of(w, struct binder_transaction, work)
23     } break;
24     case BINDER_WORK_TRANSACTION_COMPLETE: {
25         cmd = BR_TRANSACTION_COMPLETE;
26         if (put_user(cmd, (uint32_t __user *)ptr))
27             return -EFAULT;
28         ptr += sizeof(uint32_t);
29
30         binder_stat_br(proc, thread, cmd);
31         binder_debug(BINDER_DEBUG_TRANSACTION_COMPLETE,
32                     "binder: %d:%d BR_TRANSACTION_COMPLETE\n",
33                     proc->pid, thread->pid);
34
35         list_del(&w->entry);
36         kfree(w);
37         binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLET
38     } break;
39     ... ..
40 }
41
42 if (!t)
43     continue;
44
45 ... ..
46
47 }
48

```

有是一个 while 循环，前面已经有好多了，见怪不怪。前面把一个

`BINDER_WORK_TRANSACTION_COMPLETE` 的 `binder_work` 插入到了 `thread->todo` 中，所以这里的分支是 `switch` 那里的

`BINDER_WORK_TRANSACTION_COMPLETE`。这里只是把一个 `BR_TRANSACTION_COMPLETE` 返回给用户了（`put_user` 是 kernel 向用户空间写单个变量，`copy_user` 是传递一片数据）。然后就把这个 work 从 todo list 中删掉了。

然后走到下面，`t` 是 `NULL` 所以又回到循环开始地方，由于处理完了 work 就删掉了，所以这里取不到任何 todo work 了，就会走最后没那个 `else` 分支，然后这个

```
(ptr - buffer == 4 && !(thread->looper & BINDER_LOOPER_STATE_NEED_F
```

这个判断是没有任何读数据的时候，就是没做任何处理。前面最开始有向用户空间写入 `BR_NOOP` 的操作（可以回到 `service` 那里仔细看一下），所以如果没做任何处理，`ptr` 应该被移动了4个字节。但是前面处理了

`BINDER_WORK_TRANSACTION_COMPLETE` 有移动了4个字节，所以这里条件不满足。然后就 `break` 跳出循环了。然后 `binder_thread_read` 就结束了。这次 `ioctl` 也就结束了，然后又回到用户空间 `client` 那里。

这里会到前面 `client waitForResponse` 那里，从 `talkWithDriver` 返回了：

```

1      while (1) {
2          if ((err=talkWithDriver()) < NO_ERROR) break;
3          err = mIn.errorCheck();
4          if (err < NO_ERROR) break;
5          if (mIn.dataAvail() == 0) continue;
6
7          cmd = mIn.readInt32();
8
9          IF_LOG_COMMANDS() {
10             alog << "Processing waitForResponse Command: "
11                 << getReturnString(cmd) << endl;
12         }
13
14         switch (cmd) {
15             case BR_TRANSACTION_COMPLETE:
16                 if (!reply && !acquireResult) goto finish;
17                 break;
18

```

```

19     ... ..
20
21         default:
22             err = executeCommand(cmd);
23             if (err != NO_ERROR) goto finish;
24             break;
25         }
26     } // end while(1)
27
28     finish:
29         if (err != NO_ERROR) {
30             if (acquireResult) *acquireResult = err;
31             if (reply) reply->setError(err);
32             mLastError = err;
33         }
34

```

前面 kernel `binder_thread_read` 返回的 cmd 是

`BR_TRANSACTION_COMPLETE`，这边分支除了一个判断，要跳转到 finish 以外其它什么处理都没了。前面 client `waitForResponse` 是有传入参数 `reply` 的（这个是拿来存 IPC 函数的返回值的），所以这里只是 `break` 而已，没跳转去 finish（跳转到 finish 这个函数就结束了）。如果是不需要返回的，`reply` 传入 `NULL` 的话，接收到 kernel 的返回值，就直接退出了。

从这个含义来看，`BR_TRANSACTION_COMPLETE` 是 kernel binder 告诉调用者，命令发送已经处理完毕（前面 client 有对 kernel 发送 `BC_TRANSACTION` 命令）。所以 kernel 会把 `BINDER_WORK_TRANSACTION_COMPLETE` 的 work 插入到本线程的 todo list 中，为了就是告诉调用者，发送已经完成而已。而看样子，现阶段这个返回值好像没啥用，因为调用者没做啥处理。这个理解下设计者的意图，后面 service 还会有。

然后继续循环，然后调用 `talkWithDriver`，这里和前面第一部分 service 那里很像，因为前面把 `mOut` 中数据已经写完了，所以这里写入就是 0byte，读入 256byte，然后就和前面 service 一样了，阻塞在 `binder_thread_read` 那里等待 service 返回的值。这里才是符合前面所模型，client 发送了请求后，就要等待 service 的返回。这里是通过 client 那里 `waitForResponse` 的 while 来实现的，因为这个函数需要处理多个命令（kernel 的返回值），这里知道为什么这里要用循环了吧。



### 3. 服务端处理请求，并返回结果

client 在等待 service 的处理，我们回到 service 这边。之前 service 阻塞在 `binder_thread_read` 的 `ioctl` 调用那（回到图中右边部分），后面 client 发送了一个 IPC 请求，然后把一个 work 插入到了 service `proc`（`target_proc`）的 `todo list` 上，service 的阻塞就被唤醒了。我们接着看后面的处理：

```

1      while (1) {
2
3      ... ..
4
5          BUG_ON(t->buffer == NULL);
6          if (t->buffer->target_node) {
7              struct binder_node *target_node = t->buffer->target_
8              tr.target.ptr = target_node->ptr;
9              tr.cookie = target_node->cookie;
10             t->saved_priority = task_nice(current);
11             if (t->priority < target_node->min_priority &&
12                 !(t->flags & TF_ONE_WAY))
13                 binder_set_nice(t->priority);
14             else if (!(t->flags & TF_ONE_WAY) ||
15                      t->saved_priority > target_node->min_priority)
16                 binder_set_nice(target_node->min_priority);
17             cmd = BR_TRANSACTION;
18         } else {
19             tr.target.ptr = NULL;
20             tr.cookie = NULL;
21             cmd = BR_REPLY;
22         }
23         tr.code = t->code;
24         tr.flags = t->flags;
25         tr.sender_euid = t->sender_euid;
26
27         if (t->from) {
28             struct task_struct *sender = t->from->proc->tsk;
29             tr.sender_pid = task_tgid_nr_ns(sender,
30                                             current->nsproxy->pid_ns);
31         } else {
32             tr.sender_pid = 0;
33         }
34
35         tr.data_size = t->buffer->data_size;

```

```

36         tr.offsets_size = t->buffer->offsets_size;
37         tr.data.ptr.buffer = (void *)t->buffer->data +
38             proc->user_buffer_offset;
39         tr.data.ptr.offsets = tr.data.ptr.buffer +
40             ALIGN(t->buffer->data_size,
41                 sizeof(void *));
42
43         if (put_user(cmd, (uint32_t __user *)ptr))
44             return -EFAULT;
45         ptr += sizeof(uint32_t);
46         if (copy_to_user(ptr, &tr, sizeof(tr)))
47             return -EFAULT;
48         ptr += sizeof(tr);
49
50         binder_stat_br(proc, thread, cmd);
51         binder_debug(BINDER_DEBUG_TRANSACTION,
52             "binder: %d:%d %s %d %d:%d, cmd %d"
53             "size %zd-%zd ptr %p-%p\n",
54             proc->pid, thread->pid,
55             (cmd == BR_TRANSACTION) ? "BR_TRANSACTION" :
56             "BR_REPLY",
57             t->debug_id, t->from ? t->from->proc->pid : 0,
58             t->from ? t->from->pid : 0, cmd,
59             t->buffer->data_size, t->buffer->offsets_size,
60             tr.data.ptr.buffer, tr.data.ptr.offsets);
61
62         list_del(&t->work.entry);
63         t->buffer->allow_user_free = 1;
64         if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
65             t->to_parent = thread->transaction_stack;
66             t->to_thread = thread;
67             thread->transaction_stack = t;
68         } else {
69             t->buffer->transaction = NULL;
70             kfree(t);
71             binder_stats_deleted(BINDER_STAT_TRANSACTION);
72         }
73         break;
74     }
75

```

由于 client 把一个 work 插入到 service proc 中的 todo list 中，work type 是

`BINDER_WORK_TRANSACTION`，这个处理就自己把前面写入的

`binder_transaction` 这个数据结构给取出来了，所以 t 那个判断 `t != NULL`

就会继续走下面的处理（看前面 client 的代码分析）。

这里有个分支判断：t->buffer->target\_node 是否为 NULL，前面 client binder\_transaction 那里通过 Bp 的 handle 找到了目标（service）的 target node 了的（每个 Bn 都是一个 binder node，这个后面再说），然后把 target\_node 写入了 binder\_transaction 中，所以这里走 BR\_TRANSACTION 命令这个分支。这个分支把 target\_node 的 ptr 和 cookie 写了返回给 service 的数据中，这个东西就是 Bn 的地址指针（这个后面 service manager 那再具体说）。

然后下面，就是把 binder\_transaction 中的数据地址 copy 到准备返回给 service 的 binder\_transaction\_data 中。然后先是一个 put\_user 把 BR\_TRANSACTION 命令写给用户空间，后面 copy\_to\_user 把数据给写给用户空间。最后分支是 BR\_TRANSACTION 然后也需要返回，保存下传送堆栈。

之后 service 在 kernel 中的 ioctl 阻塞结束，返回到用户空间。之前在 getAndExecuteCommand 的 talkWithDriver 阻塞，现在继续往下执行，在 ioctl 读取到数据后，读取 kernel 返回的命令：

```

1  if (result >= NO_ERROR) {
2      size_t IN = mIn.dataAvail();
3      if (IN < sizeof(int32_t)) return result;
4      cmd = mIn.readInt32();
5      IF_LOG_COMMANDS() {
6          alog << "Processing top-level Command: "
7              << getReturnString(cmd) << endl;
8      }
9
10     result = executeCommand(cmd);
11
12     // After executing the command, ensure that the thread is re
13     // foreground cgroup before rejoining the pool. The driver
14     // restoring the priority, but doesn't do anything with cgrc
15     // need to take care of that here in userspace. Note that w
16     // sure to go in the foreground after executing a transactio
17     // there are other callbacks into user code that could have
18     // our group so we want to make absolutely sure it is put ba
19     set_sched_policy(mMyThreadId, SP_FOREGROUND);
20 }
21

```

之前 kernel 返回的是 `BR_TRANSACTION`，然后到 `executeCommand` 处理命令：

```

1  status_t IPCThreadState::executeCommand(int32_t cmd)
2  {
3      BBinder* obj;
4      RefBase::weakref_type* refs;
5      status_t result = NO_ERROR;
6
7      switch (cmd) {
8      ...
9
10     case BR_TRANSACTION:
11     {
12         binder_transaction_data tr;
13         result = mIn.read(&tr, sizeof(tr));
14         ALOG_ASSERT(result == NO_ERROR,
15             "Not enough command data for brTRANSACTION");
16         if (result != NO_ERROR) break;
17
18         Parcel buffer;
19         buffer.ipcSetDataReference(
20             reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
21             tr.data_size,
22             reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
23             tr.offsets_size/sizeof(size_t), freeBuffer, this);
24
25         const pid_t origPid = mCallingPid;
26         const uid_t origUid = mCallingUid;
27
28         mCallingPid = tr.sender_pid;
29         mCallingUid = tr.sender_euid;
30
31         int curPrio = getpriority(PRIO_PROCESS, mMyThreadId);
32         if (gDisableBackgroundScheduling) {
33             if (curPrio > ANDROID_PRIORITY_NORMAL) {
34                 // We have inherited a reduced priority from the caller
35                 // want to run in that state in this process
36                 // priority already (though not our scheduling policy)
37                 // it back to the default before invoking the command
38                 setpriority(PRIO_PROCESS, mMyThreadId, ANDROID_PRIORITY_NORMAL);
39             }
40         } else {
41             if (curPrio >= ANDROID_PRIORITY_BACKGROUND) {
42                 // We want to use the inherited priority from the caller
43                 // Ensure this thread is in the background
44                 // since the driver won't modify scheduling

```

```

45         // The scheduling group is reset to default
46         // once this method returns after the trans
47         set_sched_policy(mMyThreadId, SP_BACKGROUND
48     }
49 }
50
51 //ALOGI(">>> TRANSACT from pid %d uid %d\n", mCall
52
53 Parcel reply;
54 IF_LOG_TRANSACTIONS() {
55     TextOutput::Bundle _b(aalog);
56     aalog << "BR_TRANSACTION thr " << (void*)pthread
57         << " / obj " << tr.target.ptr << " / code "
58         << TypeCode(tr.code) << ": " << indent << t
59         << dedent << endl
60         << "Data addr = "
61         << reinterpret_cast<const uint8_t*>(tr.data
62         << ", offsets addr="
63         << reinterpret_cast<const size_t*>(tr.data.
64     }
65     if (tr.target.ptr) {
66         sp<BBinder> b((BBinder*)tr.cookie);
67         const status_t error = b->transact(tr.code, buf
68         if (error < NO_ERROR) reply.setError(error);
69
70     } else {
71         const status_t error = the_context_object->tran
72         if (error < NO_ERROR) reply.setError(error);
73     }
74
75     //ALOGI("<<<< TRANSACT from pid %d restore pid %d u
76     //      mCallingPid, origPid, origUid);
77
78     if ((tr.flags & TF_ONE_WAY) == 0) {
79         LOG_ONeway("Sending reply to %d!", mCallingPid)
80         sendReply(reply, 0);
81     } else {
82         LOG_ONeway("NOT sending reply to %d!", mCalling
83     }
84
85     mCallingPid = origPid;
86     mCallingUid = origUid;
87
88     IF_LOG_TRANSACTIONS() {
89         TextOutput::Bundle _b(aalog);
90         aalog << "BC_REPLY thr " << (void*)pthread_self(
91         << tr.target.ptr << ": " << indent << reply
92     }
93

```

```

94         }
95         break;
96
97     default:
98         printf("*** BAD COMMAND %d received from Binder driver\
99         result = UNKNOWN_ERROR;
100        break;
101    }
102
103    if (result != NO_ERROR) {
104        mLastError = result;
105    }
106
107    return result;
108 }
109

```

继续从 `mIn` 中把前面 kernel 写入的 `binder_transaction_data` 数据取出来，然后把 `buffer` 数据地址写入到 `Parcel` 中，这个相当于把 client 传入的 IPC 参数取出来了。

然后关键的来了，判断 `tr.target.ptr` 是否为 `NULL`，前面 kernel 把 service Bn 的指针地址写进去了，后面直接转化为 `BBinder*`，然后调用 `transaction` 方法。这个就想当于调用 service Bn 的 `transaction` 方法，最后参数调用到 service 中真正实现的函数，具体的看上一篇原理的分析（这里实现 IPC 调用，前面图中有一个流程 impl IPC API）。

在调用 `transaction` 的时候，有个引用参数 `reply`，service 的业务函数会把返回值通过 `Parcel` 打包好，然后后面那个判断前面见过很多次，是需要返回值的，所以调用 `sendReply` 把返回值，通过 kernel binder 发送给 client：

```

1  status_t IPCThreadState::sendReply(const Parcel& reply, uint32_t
2  {
3      status_t err;
4      status_t statusBuffer;
5      err = writeTransactionData(BC_REPLY, flags, -1, 0, reply, &s
6      if (err < NO_ERROR) return err;
7
8      return waitForResponse(NULL, NULL);
9  }
10

```

这个函数十分简洁，调用的2个函数前面（client 发命令那里）也都见过，而且顺序都一样，只不过参数不一样而已。writeTransactionData 打包的命令是 BC\_REPLY，而 waitForReponse 参数是 NULL，根据前面的分析，参数是 NULL 的话，那么得到 kernel 返回 BR\_TRANSACTION\_COMPLETE 函数就结束了（前面的图中是 no reply）。

这里就不贴代码了，和前面是一样的，writeTransactionData 把数据打好包后，waitForReponse 通过 talkWithDriver 的 ioctl 就到 kernel 里面了，这次 binder\_thread\_write 调用 binder\_transaction 的参数 reply 是 true，来看看有什么不一样（前面 client 是 false）：

```

1      if (reply) {
2          in_reply_to = thread->transaction_stack;
3          if (in_reply_to == NULL) {
4              binder_user_error("binder: %d:%d got reply transacti
5                  "with no transaction stack\n",
6                  proc->pid, thread->pid);
7              return_error = BR_FAILED_REPLY;
8              goto err_empty_call_stack;
9          }
10         binder_set_nice(in_reply_to->saved_priority);
11         if (in_reply_to->to_thread != thread) {
12             binder_user_error("binder: %d:%d got reply transacti
13                 "with bad transaction stack,"
14                 " transaction %d has target %d:%d\n",
15                 proc->pid, thread->pid, in_reply_to->debug_id,
16                 in_reply_to->to_proc ?
17                 in_reply_to->to_proc->pid : 0,
18                 in_reply_to->to_thread ?
19                 in_reply_to->to_thread->pid : 0);
20             return_error = BR_FAILED_REPLY;
21             in_reply_to = NULL;
22             goto err_bad_call_stack;
23         }
24         thread->transaction_stack = in_reply_to->to_parent;
25         target_thread = in_reply_to->from;
26         if (target_thread == NULL) {
27             return_error = BR_DEAD_REPLY;
28             goto err_dead_binder;
29         }
30         if (target_thread->transaction_stack != in_reply_to) {
31             binder_user_error("binder: %d:%d got reply transacti
32                 "with bad target transaction stack %d, "

```

```

33         "expected %d\n",
34         proc->pid, thread->pid,
35         target_thread->transaction_stack ?
36         target_thread->transaction_stack->debug_id : 0,
37         in_reply_to->debug_id);
38     return_error = BR_FAILED_REPLY;
39     in_reply_to = NULL;
40     target_thread = NULL;
41     goto err_dead_binder;
42 }
43     target_proc = target_thread->proc;
44 } else {
45     ... ..
46 }
47

```

这里主要的不同是寻找目标进程（线程）的不同，前面 client 那里寻找目标 service proc 是通过从 service manager 取到的 service Bp 的 handle 取到 service Bn 的 node，进而找到 service proc 的。这里却是通过前面 client 保存的 `transaction_stack`（传送堆栈）取到要返回的 client 的 thread 的。而且还根据这个校验这次的传送是否是有效的，就是保存的堆栈中的 `in_reply_to` 必须是本线程（就是前面 client 发送到的目标线程必须是自己），否则就认为是一次无效的调用，就不处理（这个可能对某些恶意注入、拦截有用吧）。

然后后面就和 client 那里一样了，创建 `binder_transaction` 和 `binder_work` 把数据和 work type 写到里面去。有一点，后面如果是 reply 则调用 `binder_pop_transaction` 把之前 client 保存堆栈出栈（因为这里是返回到 client，所以要出栈，之前 client 到 service 是入栈）。

然后也和前面 client 一样，插入了2个 work，一个插到 client 的 thread todo（这里是 thread 的 todo list，因为前面找到的 `target_thread` != NULL）里面，一个插到自己 thread 的 todo 里面，并唤醒在等待的 client 线程。

这里和前面一样，先不管 client 那边先，先继续看 service 这边。这边从 `binder_thread_write` 写完后，就到 `binder_thread_read` 了。和前面 client 一样，由于插入了 `BINDER_WORK_COMPLETE` 的 work 到 thread->todo list 所以这里不会阻塞。然后和 client 一样了，返回



`BR_TRANSACTION_COMPLETE` 给用户空间，然后退出循环，结束 `ioctl` 调用。

到用户空间，`waitForReponse` 前面说了，参数为 `NULL`，直接跳到 `finish`，结束，然后 `sendReply` 执行结束，`executeCommand` 执行结束，然后 `getAndExecuteCommand` 执行结束。这里本次 IPC 调用，`service` 端的工作就算是结束了。然后进入下一个 `joinThreadLoop` 循环，等待下一次 `client` 请求的到来。

## 4. 客户端接收到服务端返回的数据

`service` 端结束了，回到 `client` 这边（继续图的左边部分）。前面 `client` 为等待 `service` 返回的结果，阻塞在 `binder_thread_read` 那里。上一个部分，`service` 把返回数据用 `BC_REPLY` 写入 `kernel` 后，`client` 就被唤醒了，然后继续往下走：

```

1
2         BUG_ON(t->buffer == NULL);
3         if (t->buffer->target_node) {
4     ... ..
5     } else {
6         tr.target.ptr = NULL;
7         tr.cookie = NULL;
8         cmd = BR_REPLY;
9     }
10

```

前面取数据都一样了，然后主要是这里不一样，前面 `service` 那

`t->buffer->target_node` 是为 `NULL` 的，但是这里是 `NULL`（上面一部 `service` 是通过 `transaction_stack` 取的 `target thread`，所以没有 `target_node`），所以走下面。`Bn` 的地址直接设置为 `NULL`（`client` 那当然没 `Bn`，那里的是 `Bp`），返回的命令的是 `BR_REPLY`。后面的处理就和前面差不多了，方法就返回到用户空间了。

这里就从 `talkWithDriver` 回到了 `waitForReponse`，继续下面的处理：

```

1         while (1) {

```

```

2         if ((err=talkWithDriver()) < NO_ERROR) break;
3         err = mIn.errorCheck();
4         if (err < NO_ERROR) break;
5         if (mIn.dataAvail() == 0) continue;
6
7         cmd = mIn.readInt32();
8
9         IF_LOG_COMMANDS() {
10             alog << "Processing waitForResponse Command: "
11                 << getReturnString(cmd) << endl;
12         }
13
14         switch (cmd) {
15     ... ..
16         case BR_REPLY:
17             {
18                 binder_transaction_data tr;
19                 err = mIn.read(&tr, sizeof(tr));
20                 ALOG_ASSERT(err == NO_ERROR, "Not enough command");
21                 if (err != NO_ERROR) goto finish;
22
23                 if (reply) {
24                     if ((tr.flags & TF_STATUS_CODE) == 0) {
25                         reply->ipcSetDataReference(
26                             reinterpret_cast<const uint8_t*>(tr.
27                             tr.data_size,
28                             reinterpret_cast<const size_t*>(tr.o
29                             tr.offsets_size/sizeof(size_t),
30                             freeBuffer, this);
31                     } else {
32                         err = *static_cast<const status_t*>(tr.o
33                         freeBuffer(NULL,
34                             reinterpret_cast<const uint8_t*>(tr.
35                             tr.data_size,
36                             reinterpret_cast<const size_t*>(tr.o
37                             tr.offsets_size/sizeof(size_t), this
38                     }
39                 } else {
40                     freeBuffer(NULL,
41                         reinterpret_cast<const uint8_t*>(tr.data
42                         tr.data_size,
43                         reinterpret_cast<const size_t*>(tr.data.
44                         tr.offsets_size/sizeof(size_t), this);
45                     continue;
46                 }
47             }
48             goto finish;
49
50         default:

```

```

51         err = executeCommand(cmd);
52         if (err != NO_ERROR) goto finish;
53         break;
54     }
55 }
56
57 finish:
58     if (err != NO_ERROR) {
59         if (acquireResult) *acquireResult = err;
60         if (reply) reply->setError(err);
61         mLastError = err;
62     }
63

```

这里从 kernel 返回的命令是 `BR_REPLY`，处理也比较简单，就是把 service 写给 kernel 的数据指针设置给了 reply (Parcel) 而已。然后就跳出循环 finish 了。然后后面就是通过 Parcel 从 reply 中读出 service 打包的数据，作为函数的返回值。

到这里 client 发起的一次 IPC 调用就结束了。

## 总结

前面写了很多，稍微简化一下流程是：

1. service 运行，阻塞于 ioctl，等待 client 发起请求
2. client 通过 ioctl 发起 IPC 请求，等待 service 结果
  - 2.1. client send `BC_TRANSACTION` → kernel
  - 2.2. kernel return `BR_TRANSACTION_COMPLETE` → client
  - 2.3. client 阻塞于 ioctl，等待 service 返回结果
3. service 被唤醒，完成业务，返回结果
  - 3.1. kernel return `BR_TRANSACTION` → service
  - 3.2. service impl IPC call
  - 3.3. service send `BC_REPLY` → kernel
  - 3.4. kernel return `BR_TRANSACTION_COMPLETE` → service

#### 4. client 被唤醒，读取 service 返回结果，IPC 结束

4.1. kernel return `BR_REPLY` → client

4.2. IPC call end

可以看到 BC 开头的协议都是用户空间对 kernel 发送的，BR 开头的协议都是 kernel 返回给用户空间的。所以应用程序是通过 kernel 的 binder 驱动进行通信的（之前我搞混过，一开始我以为这些 `BC_XX`，`BR_XX` 是 client 发往 service，其实不是它们都只与 kernel 通信而已，不知道对方彼此的存在）。向 kernel 发送传送请求的命令（`BC_TRANSACTION`，`BC_REPLY`），kernel 会返回 `BR_TRANSACTION_COMPLETE` 告诉发送者，传送完成。

kernel binder 驱动 binder.h 中定义2个 enum，分别是：

`BinderDriverReturnProtocol` 和 `BinderDriverCommandProtocol`。里面除了上面提到的 `BC_TRANSACTION`，`BC_REPLY`，`BR_TRANSACTION_COMPLETE`，`BR_TRANSACTION`，`BR_REPLY` 还有很多别的命令，而且有一些还是没实现的（看注释有写）。其它一些用处，后面一些篇章会说到。其实名字还是挺形象的（BC、BR），只不过我一开始理解错了。

---

 [android](#)

 [Android Framework](#)



上一篇:

« [Android Binder 分析——原理](#)

下一篇:

» [Android Binder 分析——数据传递者（Parcel）](#)

分类

---

Android Development <sup>35</sup>

---

Android Framework <sup>47</sup>

---

Basics Knowledge <sup>11</sup>

---

Linux <sup>25</sup>

---

MiniGUI <sup>12</sup>

---

Other <sup>8</sup>

---

Server <sup>1</sup>

---

Window <sup>10</sup>

---

## 标签

---

Linux <sup>1</sup>   android <sup>82</sup>   basics <sup>11</sup>   install <sup>9</sup>   linux <sup>28</sup>   minigui <sup>13</sup>   opengl <sup>3</sup>  
other <sup>5</sup>   server <sup>1</sup>   shell <sup>5</sup>   window <sup>11</sup>

## 哥的后勤处 o(^▽^)o

---


-  QQ 空间
-  GitHub
-  战斗力
-  Makrdown

## 最近冒泡的小伙伴 ~(•'~'•)~

---

## 友情链接 o(^▽^)o

---

-  桃园小七的博客

Powered by hexo and Theme by Lightmoon © 2017 Mingming