# 理解Binder--内核层

要理解android系统，绕不开binder，网上有很多介绍binder的资料，其中不乏“史上最强解析”，“三分钟带你理解”之类的文章。不过看多少资料，最后都会回归到陆游的这句诗：纸上得来终觉浅，绝知此事要躬行。要理解一段程序，最后还是要回归到理解源码上来。也许你期望一篇醍醐灌顶，读过后深刻理解binder的文章，这篇显然不是，只是希望能提供一些理解源码的方式。

## 内核层的binder

提到内核代码，也许很多人有畏难情绪，不过binder的内核代码并不算多，在v.4.13之前的内核版本中，需要理解的只有一个文件。

你可以在<https://www.kernel.org/> 去下载内核代码。然后在 drivers/android 目录下存放的就是binder的代码。

|  |  |
| --- | --- |
| 文件名 |  |
| [Kconfig](https://elixir.bootlin.com/linux/v4.13.16/source/drivers/android/Kconfig) | [1364 bytes](https://elixir.bootlin.com/linux/v4.13.16/source/drivers/android/Kconfig) |
| [Makefile](https://elixir.bootlin.com/linux/v4.13.16/source/drivers/android/Makefile) | [96 bytes](https://elixir.bootlin.com/linux/v4.13.16/source/drivers/android/Makefile) |
| [binder.c](https://elixir.bootlin.com/linux/v4.13.16/source/drivers/android/binder.c) | [121569 bytes](https://elixir.bootlin.com/linux/v4.13.16/source/drivers/android/binder.c) |
| [binder\_trace.h](https://elixir.bootlin.com/linux/v4.13.16/source/drivers/android/binder_trace.h) | [8979 bytes](https://elixir.bootlin.com/linux/v4.13.16/source/drivers/android/binder_trace.h) |

图-1 ： v.4.13之前的binder目录

|  |  |
| --- | --- |
| 文件名 |  |
| [Kconfig](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/Kconfig) | [1787 bytes](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/Kconfig) |
| [Makefile](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/Makefile) | [266 bytes](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/Makefile) |
| [binder.c](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder.c) | [181284 bytes](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder.c) |
| [binder\_alloc.c](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder_alloc.c) | [32309 bytes](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder_alloc.c) |
| [binder\_alloc.h](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder_alloc.h) | [6108 bytes](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder_alloc.h) |
| [binder\_alloc\_selftest.c](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder_alloc_selftest.c) | [8153 bytes](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder_alloc_selftest.c) |
| [binder\_internal.h](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder_internal.h) | [4067 bytes](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder_internal.h) |
| [binder\_trace.h](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder_trace.h) | [11183 bytes](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binder_trace.h) |
| [binderfs.c](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binderfs.c) | [19208 bytes](https://elixir.bootlin.com/linux/v5.4-rc2/source/drivers/android/binderfs.c) |

图-2 ： 当前版本（v5.4）的binder

新版本中将共享内存相关的代码从binder.c中提取到了binder\_alloc.c中，然后添加了binderfs.c。但主要实现还是在binder.c中。

### 从隐喻开始

《代码大全》中强调了隐喻的重要性，之前并不以为然。因为在开始学c的时候，接触过多种针对指针的比喻，但对于理解指针其实并没多大用处。远没有《inside the c++ object》中几张简易的图来的透测。不过对于一个庞杂的系统，比如linux 内核，要想理解系统的方方面面，对于大多数人来说是件不现实的事情。如果有些建议的模型辅助，对于理解代码也有很大的帮助。隐喻从困扰我自己的用户空间和内核空间开始。

#### 用户空间和内核空间

一般介绍虚拟内存的文章中，都可以看到这样的论述：32位的系统中，每个进程都单独拥有4G的虚拟内存。0-3G为用户空间，3-4G为内核空间。每个进程的虚拟空间都是私有的，彼此不能互相访问。忽略进程管理，内存分配等细节，其实用户空间和内核空间其实并不复杂。

Proc 2

Proc 1

Proc ...

用户空间

Kernel

内核空间

图3 - 内核与用户空间

如图所示，所有的进程拥有同一个内核空间。在用户空间各个进程之间不能相互访问，但在内核空间就没有这个限制了。所以在内核空间可以看到这样的代码：

static HLIST\_HEAD(binder\_procs);

这行代码在binder.c中，在内核空间定义了一个全局的静态变量。这个全局变量是一个list，用来存储所有使用binder的进程，它对于所有进程而言都是可见的。

我们可以将内核空间看做是一个服务器，这里保存了系统所有的信息，比如磁盘文件，外接端口。客户空间通过特定的接口才可以得到这些信息。就和微信一个道理，微信的服务端保存了所有的注册用户信息以及朋友圈的文章等内容，客服端只有访问微信服务端才能获取到这些信息。

#### 进程间通信

各个进程都拥有自己独立的虚拟空间，彼此不能访问。但是现实环境中，进程间免不了相互交互。比如linux 运行bash是一个进程，通过bash运行一个logcat，结果刷屏了，一直停不下来。这时候，你只能通过ctrl + c 等方式强制关掉logcat。bash 和 logcat 是两个不同的进程，bash是无法直接关掉logcat的，那么这时候bash 会发一个信号给logcat，让它停止执行。这其实就是进程间通信的一个例子。

进程间通信的方式有信号量，管道，共享内存，当然还包括socket等方式。Binder是Android特有的进程间通信方式。刚开始写代码的时候，要完成的一份代码中涉及到了两个程序间共享信息。那时候我可能还没搞明白进程是啥吧，有人建议可以创建一个文件，一个程序写文件，另一个程序读文件。当时这个想法被我们报以鄙视的目光，其实这也真不是解决问题的方法。不过进程间的通信的基本原理其实和共建一个文件没多少差别，差别是内核帮我们解决了读写保护，同步等问题。

Proc 2

Proc 1

Proc ...

用户空间

Kernel

binder\_procs

binder\_proc...

binder\_proc2

binder\_proc1

内核空间

图4 - Binder内核示意

如图所示，在内核空间中有一个静态列表 binder\_proces,所有使用binder的进程都会有一个binder\_proc对象存放在内核中。而binder\_proc中都包含有进程间通信所必须的信息。

### 一切皆文件

“一切皆文件”是kernel的设计的一个哲学概念。所有的资源都可以通过文件的方式访问，都是按照open,close的方式来处理。比如如果需要使用binder，第一步就是打开它：

int fd = open(“/dev/binder”, O\_RDWR | O\_CLOEXEC);

这里可以看到通过open的方式来打开dev目录下的一个binder文件。首先来简单看看binder文件是怎么来的。

一切皆文件，对客户空间来说是一个统一的调用方式，对于kernel来说就需要有一个统一的实现框架。Linux的驱动程序模块提供了这样的框架，借助于框架的代码，很容易就可以为驱动提供统一的访问接口。Binder是被作为misc device来开发的。打开binder.c来看看具体的实现，看内核代码很多时候可以从下往上看，最开始的代码往往在最下面。比如你在binder.c的最后可以看到这行代码：

device\_initcall(binder\_init);

device\_initcall是内核里提供的宏定义，涉及到一些gcc的编译技巧，具体我不是太明白，只是知道它可以保证在系统启动的时候调用到 binder\_init函数。

static int \_\_init binder\_init(void)

{

int ret;

char \*device\_name, \*device\_tmp;

char \*device\_names = NULL;

if (strcmp(binder\_devices\_param, "") != 0) {

/\*

\* Copy the module\_parameter string, because we don't want to

\* tokenize it in-place.

\*/

device\_names = kstrdup(binder\_devices\_param, GFP\_KERNEL);

if (!device\_names) {

ret = -ENOMEM;

goto err\_alloc\_device\_names\_failed;

}

device\_tmp = device\_names;

while ((device\_name = strsep(&device\_tmp, ","))) {

ret = init\_binder\_device(device\_name);

if (ret)

goto err\_init\_binder\_device\_failed;

}

}

ret = init\_binderfs();

return ret;

}

代码中去除了用于调试和错误处理的代码，这段代码本身也不复杂。就是将binder\_devices\_param的内容拷贝到device\_names,然后通过分号分隔字符串，再逐个调用 init\_binder\_device函数。在文件的开头部分，可以看到binder\_devices\_param的定义：

static char \*binder\_devices\_param = CONFIG\_ANDROID\_BINDER\_DEVICES;

CONFIG\_ANDROID\_BINDER\_DEVICES 是编译参数，编译kernel的时候可以自行设置。一般设置为：

#define CONFIG\_ANDROID\_BINDER\_DEVICES “binder,hwbinder,vndbinder”。 也就是说会调用三次init\_binder\_device，init\_binder\_device中传入“binder”，会生成一个文件“/dev/binder”，调用三次，生成了另两个文件“/dev/hwbinder”和“/dev/vndbinder”。这里生成三个文件是为了不同类型的进程只与同类型的进程通信，比如android自身框架内的进行选择“/dev/binder”，而第三方硬件相关的可以选择“/dev/hwbinder”。这部分内容了解binder的实现后，并不难理解。后续再说。另外：

ret = init\_binderfs();

是后来增加的实现。前面说了通过配置CONFIG\_ANDROID\_BINDER\_DEVICES可以得到多个设备文件，但是需要在编译内核之前就设定好。在某些应用中，这个还不够，binderfs就是用来动态的添加设备文件的。这部分的内容不做展开，有兴趣的可以自行了解。

static int \_\_init init\_binder\_device(const char \*name)

{

int ret;

struct binder\_device \*binder\_device;

binder\_device = kzalloc(sizeof(\*binder\_device), GFP\_KERNEL);

if (!binder\_device)

return -ENOMEM;

binder\_device->miscdev.fops = &binder\_fops;

binder\_device->miscdev.minor = MISC\_DYNAMIC\_MINOR;

binder\_device->miscdev.name = name;

binder\_device->context.binder\_context\_mgr\_uid = INVALID\_UID;

binder\_device->context.name = name;

mutex\_init(&binder\_device->context.context\_mgr\_node\_lock);

ret = misc\_register(&binder\_device->miscdev);

if (ret < 0) {

kfree(binder\_device);

return ret;

}

hlist\_add\_head(&binder\_device->hlist, &binder\_devices);

return ret;

}

init\_binder\_device的作用是根据传入的参数来生成对应的设备文件，比如/dev/binder。misc\_register帮助实现这个过程，生成的文件需要支持open,close等方法，这里由

binder\_device->miscdev.fops = &binder\_fops;

来指定这些方法在内核中的实现。

const struct file\_operations binder\_fops = {

.owner = THIS\_MODULE,

.poll = binder\_poll,

.unlocked\_ioctl = binder\_ioctl,

.compat\_ioctl = binder\_ioctl,

.mmap = binder\_mmap,

.open = binder\_open,

.flush = binder\_flush,

.release = binder\_release,

}

binder\_fops相当于函数指针，当用户空间调用open的时候，转到内核会调用到binder\_open。binder并不支持所有的文件操作，比如read,seek等并不支持。所支持的操作就这里所列的这些。

### 打开文件

之前就说过，binder是按照文件的方式来处理的。这里来关注一下打开文件的函数binder\_open，和这里处理的主要数据对象binder\_proc。

#### binder\_open 函数

static int binder\_open(struct inode \*nodp, struct file \*filp)

{

struct binder\_proc \*proc;

struct binder\_device \*binder\_dev;

proc = kzalloc(sizeof(\*proc), GFP\_KERNEL);

if (proc == NULL)

return -ENOMEM;

spin\_lock\_init(&proc->inner\_lock);

spin\_lock\_init(&proc->outer\_lock);

get\_task\_struct(current->group\_leader);

proc->tsk = current->group\_leader;

INIT\_LIST\_HEAD(&proc->todo);

proc->default\_priority = task\_nice(current);

/\* binderfs stashes devices in i\_private \*/

if (is\_binderfs\_device(nodp))

binder\_dev = nodp->i\_private;

else

binder\_dev = container\_of(filp->private\_data,

struct binder\_device, miscdev);

proc->context = &binder\_dev->context;

binder\_alloc\_init(&proc->alloc);

binder\_stats\_created(BINDER\_STAT\_PROC);

proc->pid = current->group\_leader->pid;

INIT\_LIST\_HEAD(&proc->delivered\_death);

INIT\_LIST\_HEAD(&proc->waiting\_threads);

filp->private\_data = proc;

mutex\_lock(&binder\_procs\_lock);

hlist\_add\_head(&proc->proc\_node, &binder\_procs);

mutex\_unlock(&binder\_procs\_lock);

return 0;

}

从代码开始的 proc = kzalloc(sizeof(\*proc), GFP\_KERNEL); 到函数结束部分的

hlist\_add\_head(&proc->proc\_node, &binder\_procs); 这个函数的作用就是生成一个binder\_proc对象，并把它保存到全局变量binder\_procs中去。关于binder\_proc结构中的元素，下面再详细讲。先来看看binder\_open中传入的两个参数：

**·**  inode

磁盘上有众多文件，不可能将所有的文件都加载到内存中。文件系统创建一个索引结构，这个结构里包含文件的必要信息：文件名，文件在硬盘上的扇区号，创建者等。索引结构是存储在硬盘的特殊位置，以便系统启动的时候可以加载。不使用binderfs的时候，没有用到inode。

if (is\_binderfs\_device(nodp))

binder\_dev = nodp->i\_private;

else

binder\_dev = container\_of(filp->private\_data,

struct binder\_device, miscdev);

用到binderfs的时候会从nodep去取binder\_device对象。

**·**  file

binder\_dev = container\_of(filp->private\_data,

struct binder\_device, miscdev);

这里会从filp中提取一个binder\_device对象，先看一下binder\_device的定义：

struct binder\_device {

struct hlist\_node hlist;

struct miscdevice miscdev;

struct binder\_context context;

struct inode \*binderfs\_inode;

};

对于hlist\_node,rb\_node,rb\_root等是内核中用来实现列表，红黑树等而定义的结构体。这些结构体里一般只是定义了一些指向其它节点的指针，比如：

struct hlist\_node {

struct hlist\_node \*next, \*\*pprev;

};

这部分内容感兴趣的，可以自己翻看一下。网上应该可以找到相关的文章。

struct miscdevice miscdev ,binder是作为miscdevice类型的驱动开发的，所以必须包含一个miscdevice对象，这个对象会被传入驱动的框架代码进行处理。在init\_binder\_device函数中，

ret = misc\_register(&binder\_device->miscdev);

就是将刚生成的binder\_device对象中的misdev传入框架代码去处理。而filp就是将这个对象再带回来。这里的container\_of 是内核开发中用到的小技巧，filp->private\_data指向的是binder\_device中的miscdev对象，这里想用的是binder\_device对象，misdev对象对于binder\_device对象有一个偏移，container\_of就是算出这个偏移，移动指针指向binder\_device对象的开头。

当 CONFIG\_ANDROID\_BINDER\_DEVICES 为 “binder,hwbinder,vndbinder”的时候，系统会生成三个binder\_device对象，每个binder\_device都有自己的binder\_context 对象，而同一个设备的话，都会指向同一个binder\_context对象。

而在binderfs方式中，会生成一个inode对象。

#### binder\_proc

struct binder\_proc {

//所有的binder\_proc会作为列表，保存在binder\_procs中

// bidner\_procs是全局静态变量，定义在文件开头

// static HLIST\_HEAD(binder\_procs);

struct hlist\_node proc\_node;

//用于记录使用binder功能的线程

struct rb\_root threads;

//保存属于该进程的binder\_node

struct rb\_root nodes;

//保存属于别的进程的binder\_node

//这里有两个红黑树，其实是同一个内容，两种排序方式

struct rb\_root refs\_by\_desc;

struct rb\_root refs\_by\_node;

//等待操作的进程

struct list\_head waiting\_threads;

//进程ID以及该进程的task\_struct

int pid;

struct task\_struct \*tsk;

//用于binder\_flush和binder\_release操作

struct hlist\_node deferred\_work\_node;

int deferred\_work;

bool is\_dead;

//等待处理的work

struct list\_head todo;

struct binder\_stats stats;

//分发死亡通知

struct list\_head delivered\_death;

//进程可以开多个线程去等待内核给出的消息

//当内核里等待的消息数量多余进程中的等待线程数的时候，

//内核可以通知进程增加线程数。

//以下三个变量用来记录线程数

int max\_threads;

int requested\_threads;

int requested\_threads\_started;

int tmp\_ref;

long default\_priority;

struct dentry \*debugfs\_entry;

// 针对于共享内存的操作

struct binder\_alloc alloc;

// 同一个binder\_device对象所共享对象

struct binder\_context \*context;

spinlock\_t inner\_lock;

spinlock\_t outer\_lock;

};

当调用open的时候，内核会为该进程创建一个binder\_proc对象，保存到全局变量binder\_procs中。

这里为binder\_proc结构中的大部分变量标注了用处，不过具体的作用，在后续代码分析中会变得清晰。

我想过一个问题，一个进程是否可以打开多次binder设备？应该是可以得，不过分析代码的时候带着这个思路去分析，会无形的增加复杂度。幸好在android的框架代码里封装了对binder的处理，保证一个进程只会打开一次binder设备，所以这个问题可以略过了。

### Binder的参数设定

Binder的目的是进程间的通信，不过在通信之前还需要做一些设置，另外还需要外部程序的帮忙。

#### 4.1 Sevice Manager

binder的内核显然不愿意所有的工作都由自己来做，所以分担了一部分工作给用户进程，也就是这里要讲的service manager。Service manger的主要工作是其它进程注册它们需要提供给外部使用的服务，要使用这些服务的进程可以方便的找些服务。这个有点象微信的注册服务，你想找人聊天的时候，先申请一个账号，然后其它人就可以通过服务器来找到你。

ServiceManager的代码可以在android的源码目录：

frameworks/native/cmds/servicemanager

中找到。

这节主要关注service manager启动时进行的处理：

int main(int argc, char\*\* argv) {

if (argc > 2) {

LOG(FATAL) << "usage: " << argv[0] << " [binder driver]";

}

const char\* driver = argc == 2 ? argv[1] : "/dev/binder";

sp<ProcessState> ps = ProcessState::initWithDriver(driver);

ps->setThreadPoolMaxThreadCount(0);

ps->setCallRestriction(ProcessState::CallRestriction::FATAL\_IF\_NOT\_ONEWAY);

sp<ServiceManager> manager = new ServiceManager(std::make\_unique<Access>());

IPCThreadState::self()->setTheContextObject(manager);

ps->becomeContextManager(nullptr, nullptr);

IPCThreadState::self()->joinThreadPool();

// should not be reached

return EXIT\_FAILURE;

}

其中 sp<ProcessState> ps = ProcessState::initWithDriver(driver); 的作用就是调用open打开设备。这节主要关注 setThreadPoolMaxThreadCount 和 becomeContextManager两个函数调用，其它代码除了joinThreadPool，大多与内核没啥关系，是自身代码维护的需要。

**·** setThreadPoolMaxThreadCount

以下的代码位于 frameworks/native/libs/binder中，关于binder框架用到的native代码都在这个目录下。

status\_t ProcessState::setThreadPoolMaxThreadCount(size\_t maxThreads) {

status\_t result = NO\_ERROR;

if (ioctl(mDriverFD, BINDER\_SET\_MAX\_THREADS, &maxThreads) != -1) {

mMaxThreads = maxThreads;

} else {

result = -errno;

ALOGE("Binder ioctl to set max threads failed: %s", strerror(-result));

}

return result;

}

这段代码，我们关注的是 ioctl(mDriverFD, BINDER\_SET\_MAX\_THREADS, &maxThreads) 这句。 mDriverFD 是open 打开设备后返回的文件编号，ioctl向设备发送一个BINDER\_SET\_MAX\_THREADS命令。

**·** becomeContextManager

bool ProcessState::becomeContextManager(context\_check\_func checkFunc, void\* userData)

{

AutoMutex \_l(mLock);

mBinderContextCheckFunc = checkFunc;

mBinderContextUserData = userData;

flat\_binder\_object obj {

.flags = FLAT\_BINDER\_FLAG\_TXN\_SECURITY\_CTX,

};

int result = ioctl(mDriverFD, BINDER\_SET\_CONTEXT\_MGR\_EXT, &obj);

// fallback to original method

if (result != 0) {

android\_errorWriteLog(0x534e4554, "121035042");

int dummy = 0;

result = ioctl(mDriverFD, BINDER\_SET\_CONTEXT\_MGR, &dummy);

}

if (result == -1) {

mBinderContextCheckFunc = nullptr;

mBinderContextUserData = nullptr;

ALOGE("Binder ioctl to become context manager failed: %s\n", strerror(errno));

}

return result == 0;

}

这里与内核的交互也是通过ioctl来实现，如果BINDER\_SET\_CONTEXT\_MGR\_EXT失败，那么就再执行BINDER\_SET\_CONTEXT\_MGR。BINDER\_SET\_CONTEXT\_MGR是旧的实现，BINDER\_SET\_CONTEXT\_MGR\_EXT是后来增加的。

从以上两个函数可以看到，它们都是通过ioctl来与内核交互的，所以我们的关注点也来到了ioctl的内核实现binder\_ioctl。

#### 4.2 binder\_thread

先看看binder\_ioctl开始的代码：

static long binder\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg)

{

int ret;

struct binder\_proc \*proc = filp->private\_data;

struct binder\_thread \*thread;

unsigned int size = \_IOC\_SIZE(cmd);

void \_\_user \*ubuf = (void \_\_user \*)arg;

/\*pr\_info("binder\_ioctl: %d:%d %x %lx\n",

proc->pid, current->pid, cmd, arg);\*/

binder\_selftest\_alloc(&proc->alloc);

trace\_binder\_ioctl(cmd, arg);

ret = wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);

if (ret)

goto err\_unlocked;

thread = binder\_get\_thread(proc);

if (thread == NULL) {

ret = -ENOMEM;

goto err;

}

每次运行binder\_ioctl都会调用 thread = binder\_get\_thread(proc)，我们来看看binder\_thread是什么内容。

##### · binder\_thread结构

struct binder\_thread {

// 属于哪个binder\_proc

struct binder\_proc \*proc;

// 红黑树节点，用来将该对象加入到binder\_proc的threads树中

struct rb\_node rb\_node;

// 用来将该对象加入到binder\_proc的waiting\_threads列表

struct list\_head waiting\_thread\_node;

int pid;

// 标记当前线程是否进入循环

int looper; /\* only modified by this thread \*/

// 执行flush，release之后，线程不需要等待结果，就可以返回。

// 这时候该值设置成true

bool looper\_need\_return; /\* can be written by other thread \*/

// 辅助进程内数据传输

struct binder\_transaction \*transaction\_stack;

// 该进程当前等待的work

struct list\_head todo;

// 是否有任务要处理

bool process\_todo;

struct binder\_error return\_error;

struct binder\_error reply\_error;

wait\_queue\_head\_t wait;

struct binder\_stats stats;

atomic\_t tmp\_ref;

bool is\_dead;

};

以上是binder\_thread的基本定义，这里描述一下binder\_thread的主要作用。

我们知道，每个进程都可以有多个线程，每个线程都可以对binder进行操作。这里将每个操作的进程都记录了下来，然后保存到了binder\_procs的threads中，threads是个红黑树对象。

记录每个线程的目的是：

A, 当一个线程对另外一个进程传输消息时，当传输内容结束，那么剩下的工作就转移到了对方进程，当前线程就需要等待。当对方进程处理完毕的时候，就需要唤醒该线程。

1. 如果当前进程是服务端，那么它可以有多个线程等待别的进程要求的服务。系统需要知道那些线程是可以用来处理请求的。

##### · binder\_get\_thread 函数

static struct binder\_thread \*binder\_get\_thread(struct binder\_proc \*proc)

{

struct binder\_thread \*thread;

struct binder\_thread \*new\_thread;

binder\_inner\_proc\_lock(proc);

thread = binder\_get\_thread\_ilocked(proc, NULL);

binder\_inner\_proc\_unlock(proc);

if (!thread) {

new\_thread = kzalloc(sizeof(\*thread), GFP\_KERNEL);

if (new\_thread == NULL)

return NULL;

binder\_inner\_proc\_lock(proc);

thread = binder\_get\_thread\_ilocked(proc, new\_thread);

binder\_inner\_proc\_unlock(proc);

if (thread != new\_thread)

kfree(new\_thread);

}

return thread;

}

binder\_get\_thread 通过binder\_get\_thread\_ilocked去查找当前的线程有没有记录，如果没有，那就新生成一个binder\_thread对象：

new\_thread = kzalloc(sizeof(\*thread), GFP\_KERNEL);

然后再次调用 binder\_get\_thread\_ilocked。

static struct binder\_thread \*binder\_get\_thread\_ilocked(

struct binder\_proc \*proc, struct binder\_thread \*new\_thread)

{

struct binder\_thread \*thread = NULL;

struct rb\_node \*parent = NULL;

struct rb\_node \*\*p = &proc->threads.rb\_node;

//红黑树的查找部分

while (\*p) {

parent = \*p;

thread = rb\_entry(parent, struct binder\_thread, rb\_node);

if (current->pid < thread->pid)

p = &(\*p)->rb\_left;

else if (current->pid > thread->pid)

p = &(\*p)->rb\_right;

else

return thread;

}

// new\_thread 的初始化部分

if (!new\_thread)

return NULL;

thread = new\_thread;

binder\_stats\_created(BINDER\_STAT\_THREAD);

thread->proc = proc;

thread->pid = current->pid;

atomic\_set(&thread->tmp\_ref, 0);

init\_waitqueue\_head(&thread->wait);

INIT\_LIST\_HEAD(&thread->todo);

rb\_link\_node(&thread->rb\_node, parent, p);

rb\_insert\_color(&thread->rb\_node, &proc->threads);

thread->looper\_need\_return = true;

thread->return\_error.work.type = BINDER\_WORK\_RETURN\_ERROR;

thread->return\_error.cmd = BR\_OK;

thread->reply\_error.work.type = BINDER\_WORK\_RETURN\_ERROR;

thread->reply\_error.cmd = BR\_OK;

INIT\_LIST\_HEAD(&new\_thread->waiting\_thread\_node);

return thread;

}

##### 红黑树的查找部分

内核的rbtree.h并没有提供完成的红黑树实现，红黑树的查找需要调用者自己实现：

struct rb\_node \*\*p = &proc->threads.rb\_node;

得到binder\_proc中的threads节点，然后进入while循环

thread = rb\_entry(parent, struct binder\_thread, rb\_node);

然后转换rb\_node到binder\_thread对象，然后

if (current->pid < thread->pid)

current是当前的进程结构，可以得到pid，在与红黑树种取出的pid进行比较，判断是否包含当前线程。

这部分可以多关注一下，后续还有几个红黑树，都是类似的实现。

##### new\_thread 的初始化

这部分代码大部分是结构初始化的内容，并不复杂。这里关注一下如何将新节点插入到红黑树中：

rb\_link\_node(&thread->rb\_node, parent, p);

rb\_insert\_color(&thread->rb\_node, &proc->threads);

这里的parent是刚才在红黑书查找过程中得到的左或者右子树为空的节点，新节点就是要作为它的子节点。而p 就是parent的左或右节点的地址指针，\*p也就是parent的子点点所指向的内容是NULL，现在需要它指向thread->rb\_node。

以上是把新节点添加到了红黑数，而后的rb\_insert\_color会再去调整树结构。

binder\_thread的相关信息先介绍到这儿，后续的应用到还有提到。这里需要注意的是binder\_thread其实只负责记录线程相关的一些信息，线程运行并由它负责，线程其实由用户空间主动启动的。binder\_proc其实也类似。

### 4.3 设置最大线程数

之前提到过，用户程序通过ioctl与内核交互：

ioctl(mDriverFD, BINDER\_SET\_MAX\_THREADS, &maxThreads)

其中BINDER\_SET\_MAX\_THREADS 是内核预定义的命令类型。binder所支持的命令类型在内核代码的 include/uapi/linux/android/binder.h 中：

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

#define BINDER\_SET\_IDLE\_TIMEOUT \_IOW('b', 3, \_\_s64)

#define BINDER\_SET\_MAX\_THREADS \_IOW('b', 5, \_\_u32)

#define BINDER\_SET\_IDLE\_PRIORITY \_IOW('b', 6, \_\_s32)

#define BINDER\_SET\_CONTEXT\_MGR \_IOW('b', 7, \_\_s32)

#define BINDER\_THREAD\_EXIT \_IOW('b', 8, \_\_s32)

#define BINDER\_VERSION \_IOWR('b', 9, struct binder\_version)

#define BINDER\_GET\_NODE\_DEBUG\_INFO \_IOWR('b', 11, struct binder\_node\_debug\_info)

#define BINDER\_GET\_NODE\_INFO\_FOR\_REF \_IOWR('b', 12, struct binder\_node\_info\_for\_ref)

#define BINDER\_SET\_CONTEXT\_MGR\_EXT \_IOW('b', 13, struct flat\_binder\_object)

以上是binder可以支持的所有ioctl 命令，不过 BINDER\_SET\_IDLE\_TIMEOUT 和 BINDER\_SET\_IDLE\_PRIORITY并没有实现。

\_IOWR其实应用位操作生成一个整数，感兴趣的话自己看看吧。

先捡一个简单的例子，看看binder\_ioctl的运行：

case BINDER\_SET\_MAX\_THREADS: {

int max\_threads;

if (copy\_from\_user(&max\_threads, ubuf,

sizeof(max\_threads))) {

ret = -EINVAL;

goto err;

}

binder\_inner\_proc\_lock(proc);

proc->max\_threads = max\_threads;

binder\_inner\_proc\_unlock(proc);

break;

}

这部分代码不复杂，首先通过copy\_from\_user将用户空间的变量拷贝到内核变量max\_threads，内核空间也不可以直接使用用户空间的变量。然后将该变量赋值到binder\_proc的max\_threads。

由于可能有多个线程来设置线程数，所有这里使用了binder\_proc的进程内部锁 binder\_inner\_proc\_lock 和binder\_inner\_proc\_unlock。 锁对于理解代码没有多大的障碍，暂时忽略这些内容也可以。

这里先简单介绍一下max\_threads的用处，当一个进程是作为服务端为其它服务提供服务的时候（比如Service Manager），就可以提供多个工作线程等待其它进程的访问。但当访问的线程数多于工作线程的时候，内核端可以计算出这个情形，内核就会发一个BR\_SPAWN\_LOOPER消息给工作进程。但如果当前的进程数已经超过max\_threads的设置时，就不会再发该命令了。

### 4.4 Context Manager

之前说过每个binder设备都会有一个binder\_context 对象，这里讲的就是对于binder\_context的处理。那么看看binder\_context有什么内容：

struct binder\_context {

struct binder\_node \*binder\_context\_mgr\_node;

struct mutex context\_mgr\_node\_lock;

kuid\_t binder\_context\_mgr\_uid;

const char \*name;

};

其中最重要的内容就是binder\_node对象了，先来看看binder\_node 是什么：

#### · binder\_node 对象

struct binder\_node {

int debug\_id;

spinlock\_t lock;

// 需要处理的工作列表

struct binder\_work work;

// 该对象将被保存到binder\_proc的nodes树，

// 或者全局变量binder\_dead\_nodes中，以备被删除

union {

struct rb\_node rb\_node;

struct hlist\_node dead\_node;

};

// 属于哪个binder\_proc对象

struct binder\_proc \*proc;

// 当有进程要用到该node时候，会生成一个binder\_ref对象

// 生成的binder\_ref对象，会添加到这个列表

struct hlist\_head refs;

// 这几个多是关于该node的引用

int internal\_strong\_refs;

int local\_weak\_refs;

int local\_strong\_refs;

// 在一个内部函数中对binder\_node进行处理的时候，该变量加1

// 以防在函数运行中，binder\_node对象被删除

int tmp\_refs;

// 用户空间的指针和cookie

binder\_uintptr\_t ptr;

binder\_uintptr\_t cookie;

//以下都是方便内部处理的标记量

struct {

/\*

\* bitfield elements protected by

\* proc inner\_lock

\*/

u8 has\_strong\_ref:1;

u8 pending\_strong\_ref:1;

u8 has\_weak\_ref:1;

u8 pending\_weak\_ref:1;

};

struct {

/\*

\* invariant after initialization

\*/

u8 accept\_fds:1;

u8 txn\_security\_ctx:1;

u8 min\_priority;

};

// 异步传输标记和列表

bool has\_async\_transaction;

struct list\_head async\_todo;

};

binder\_node是什么？我们先截取部分android框架的代码，最新的android代码,都按照binder框架提供的类进行了包装，新代码特点是更难理解了，之前简单的一行代码，变成了几个类。这里攫取的是很久很久以前的代码......

int main(int argc, char\*\* argv)

{

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

sp<IServiceManager> sm = defaultServiceManager();

ALOGI("ServiceManager: %p", sm.get());

AudioFlinger::instantiate();

MediaPlayerService::instantiate();

CameraService::instantiate();

AudioPolicyService::instantiate();

ProcessState::self()->startThreadPool();

IPCThreadState::self()->joinThreadPool();

}

以上是audioserver的代码，可以在 frameworks/av/media/audioserver/ 找到。这里截取的是旧代码，新代码功能一样，就是变复杂了。

audioserver是用来播放音声文件的服务器，其中它包含AudioFlinger,AudioPolicyService等服务。AudioFlinger是播放pcm格式数据的服务。

void AudioFlinger::instantiate() {

defaultServiceManager()->addService(

String16("media.audio\_flinger"), new AudioFlinger());

}

以上代码在frameworks/av/services/audioflinger目录下。这里也是截取的旧代码，新代码的instantiate提取到基类去做了，但是功能是一样的。为了方便解释，这里选择了旧代码。

这部分代码的实现，我们后续再说，这里关注的是binder\_node里保存的是什么，addService中传入了两个参数: String16("media.audio\_flinger")内核不关注，会转运到service manager中去。但是 new AudioFlinger() 会被内核保存下来，保存的地方就是binder\_node。

再看 binder\_node中的 binder\_uintptr\_t ptr，就是用来存放new AudioFlinger()的，也就是指向AudioFlinger对象的指针。

好吧，啰嗦这么多，其实就是想让你知道binder\_node中保存的是一个对象指针。

再看audioserver中还有AudioPolicyService等服务，也就意味着会有多个binder\_node，如之前提到的它们将作为红黑树存放在binder\_proc中。

binder\_node的介绍先到此为止，因为现在还没有办法将它述说清楚。其实完全理解binder\_node后，binder也就理解了。

#### · 作为管理者的binder\_node

每个binder设备都有一个特殊的binder\_node，也就是binder\_context中的binder\_context\_mgr\_node。拥有这个节点的进程就需要肩负起辅助内核管理各个服务的义务，android系统中其实就是service manager。

回到binder\_ioctl函数：

case BINDER\_SET\_CONTEXT\_MGR\_EXT: {

struct flat\_binder\_object fbo;

if (copy\_from\_user(&fbo, ubuf, sizeof(fbo))) {

ret = -EINVAL;

goto err;

}

ret = binder\_ioctl\_set\_ctx\_mgr(filp, &fbo);

if (ret)

goto err;

break;

}

case BINDER\_SET\_CONTEXT\_MGR:

ret = binder\_ioctl\_set\_ctx\_mgr(filp, NULL);

if (ret)

goto err;

break;

只有service manager 应该调用 BINDER\_SET\_CONTEXT\_MGR\_EXT 或者 BINDER\_SET\_CONTEXT\_MGR 命令。BINDER\_SET\_CONTEXT\_MGR\_EXT是后续添加的命令，两者的差别是客户空间有没有传入一个flat\_binder\_object对象，flat\_binder\_object结构暂且不展开，它里面包含的其实就是客户空间中的一个对象的指针（如刚才提到的new AudioFlinger()）。

static int binder\_ioctl\_set\_ctx\_mgr(struct file \*filp,

struct flat\_binder\_object \*fbo)

{

int ret = 0;

struct binder\_proc \*proc = filp->private\_data;

struct binder\_context \*context = proc->context;

struct binder\_node \*new\_node;

kuid\_t curr\_euid = current\_euid();

mutex\_lock(&context->context\_mgr\_node\_lock);

// 如果binder\_context\_mgr\_node已经存在，报错返回

if (context->binder\_context\_mgr\_node) {

pr\_err("BINDER\_SET\_CONTEXT\_MGR already set\n");

ret = -EBUSY;

goto out;

}

// 安全设置的问题，不是太了解

ret = security\_binder\_set\_context\_mgr(proc->tsk);

if (ret < 0)

goto out;

if (uid\_valid(context->binder\_context\_mgr\_uid)) {

if (!uid\_eq(context->binder\_context\_mgr\_uid, curr\_euid)) {

pr\_err("BINDER\_SET\_CONTEXT\_MGR bad uid %d != %d\n",

from\_kuid(&init\_user\_ns, curr\_euid),

from\_kuid(&init\_user\_ns,

context->binder\_context\_mgr\_uid));

ret = -EPERM;

goto out;

}

} else {

context->binder\_context\_mgr\_uid = curr\_euid;

}

// 生成新的binder\_node对象

new\_node = binder\_new\_node(proc, fbo);

if (!new\_node) {

ret = -ENOMEM;

goto out;

}

binder\_node\_lock(new\_node);

new\_node->local\_weak\_refs++;

new\_node->local\_strong\_refs++;

new\_node->has\_strong\_ref = 1;

new\_node->has\_weak\_ref = 1;

context->binder\_context\_mgr\_node = new\_node;

binder\_node\_unlock(new\_node);

binder\_put\_node(new\_node);

out:

mutex\_unlock(&context->context\_mgr\_node\_lock);

return ret;

}

这段代码不复杂，除了安全设置那块，我不太了解，不做理会。还有一个调用到的函数 binder\_new\_node， 其实和binder\_get\_thread的实现雷同。都是针对红黑树的查询插入。可以自行理解。

这里比较麻烦的就是local\_weak\_refs这些指引变量了，这个也在后面做说明。

## 数据传输

这就到了最核心的部分了，binder的作用其实就是进程间数据的传输。还是从客户空间的代码入手，开始新的旅程：

void AudioFlinger::instantiate() {

defaultServiceManager()->addService(

String16("media.audio\_flinger"), new AudioFlinger());

}

这是之前用到过的注册服务器的代码，其实再复杂的传输，原理都是一致的，理解了这部分代码，其实也就理解了binder了。

客户进程的代码自有一套逻辑，也需要展开去说，所以这里先略去这部分代码，我把与内核交互的代码列出来：

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

与内核交互的就是这行代码，向内核端发送一个BINDER\_WRITE\_READ命令，外加一个binder\_write\_read变量。不借助于外部代码，只好在内核中解析这部分内容。

### 5.1 binder\_write\_read 对象

struct binder\_write\_read {

binder\_size\_t write\_size; /\* bytes to write \*/

binder\_size\_t write\_consumed; /\* bytes consumed by driver \*/

binder\_uintptr\_t write\_buffer;

binder\_size\_t read\_size; /\* bytes to read \*/

binder\_size\_t read\_consumed; /\* bytes consumed by driver \*/

binder\_uintptr\_t read\_buffer;

};

write代表要内核解析的数据，read表示希望在内核处理完以后返回给客户空间的数据。

#### 5.2 binder\_ioctl\_write\_read 函数

case BINDER\_WRITE\_READ:

ret = binder\_ioctl\_write\_read(filp, cmd, arg, thread);

if (ret)

goto err;

break;

以上是binder\_ioctl中的函数，如果命令是BINDER\_WRITE\_READ ,将调用到binder\_ioctl\_write\_read函数。

static int binder\_ioctl\_write\_read(struct file \*filp,

unsigned int cmd, unsigned long arg,

struct binder\_thread \*thread)

{

int ret = 0;

struct binder\_proc \*proc = filp->private\_data;

unsigned int size = \_IOC\_SIZE(cmd);

void \_\_user \*ubuf = (void \_\_user \*)arg;

struct binder\_write\_read bwr;

if (size != sizeof(struct binder\_write\_read)) {

ret = -EINVAL;

goto out;

}

if (copy\_from\_user(&bwr, ubuf, sizeof(bwr))) {

ret = -EFAULT;

goto out;

}

binder\_debug(BINDER\_DEBUG\_READ\_WRITE,

"%d:%d write %lld at %016llx, read %lld at %016llx\n",

proc->pid, thread->pid,

(u64)bwr.write\_size, (u64)bwr.write\_buffer,

(u64)bwr.read\_size, (u64)bwr.read\_buffer);

if (bwr.write\_size > 0) {

ret = binder\_thread\_write(proc, thread,

bwr.write\_buffer,

bwr.write\_size,

&bwr.write\_consumed);

trace\_binder\_write\_done(ret);

if (ret < 0) {

bwr.read\_consumed = 0;

if (copy\_to\_user(ubuf, &bwr, sizeof(bwr)))

ret = -EFAULT;

goto out;

}

}

if (bwr.read\_size > 0) {

ret = binder\_thread\_read(proc, thread, bwr.read\_buffer,

bwr.read\_size,

&bwr.read\_consumed,

filp->f\_flags & O\_NONBLOCK);

trace\_binder\_read\_done(ret);

binder\_inner\_proc\_lock(proc);

if (!binder\_worklist\_empty\_ilocked(&proc->todo))

binder\_wakeup\_proc\_ilocked(proc);

binder\_inner\_proc\_unlock(proc);

if (ret < 0) {

if (copy\_to\_user(ubuf, &bwr, sizeof(bwr)))

ret = -EFAULT;

goto out;

}

}

binder\_debug(BINDER\_DEBUG\_READ\_WRITE,

"%d:%d wrote %lld of %lld, read return %lld of %lld\n",

proc->pid, thread->pid,

(u64)bwr.write\_consumed, (u64)bwr.write\_size,

(u64)bwr.read\_consumed, (u64)bwr.read\_size);

if (copy\_to\_user(ubuf, &bwr, sizeof(bwr))) {

ret = -EFAULT;

goto out;

}

out:

return ret;

}

binder\_ioctl\_write\_read 的作用其实就是在 bwr.write\_size > 0 的时候调用binder\_thread\_write， 在bwr.read\_size > 0 的时候调用binder\_thread\_read。

addService里包含有write数据，会调用到binder\_thread\_write。

#### 5.3 binder\_thread\_write 函数

##### A.写操作的的命令列表

对于写操作，binder也提供了命令列表。

include/uapi/linux/android/binder.h 中有

enum binder\_driver\_command\_protocol {

BC\_TRANSACTION = \_IOW('c', 0, struct binder\_transaction\_data),

BC\_REPLY = \_IOW('c', 1, struct binder\_transaction\_data),

这里只截取了最重要的传输和回复命令，其它的可以自行参考。\_IOW 之前提到过，BC\_TRANSACTION,BC\_REPLY实际上就是整数。

顺便提一下这里的BC是binder command的缩写，以免和后续提到的BR\_混淆。

##### 获取命令

static int binder\_thread\_write(struct binder\_proc \*proc,

struct binder\_thread \*thread,

binder\_uintptr\_t binder\_buffer, size\_t size,

binder\_size\_t \*consumed)

{

uint32\_t cmd;

struct binder\_context \*context = proc->context;

void \_\_user \*buffer = (void \_\_user \*)(uintptr\_t)binder\_buffer;

void \_\_user \*ptr = buffer + \*consumed;

void \_\_user \*end = buffer + size;

while (ptr < end && thread->return\_error.cmd == BR\_OK) {

int ret;

if (get\_user(cmd, (uint32\_t \_\_user \*)ptr))

return -EFAULT;

ptr += sizeof(uint32\_t);

trace\_binder\_command(cmd);

这里函数中传进来的参数binder\_buffer, size 和 consumed对应于binder\_write\_read结构中的write\_buffer，write\_size和write\_consumed。也就是客户空间传进来的内容。然后这部分内容被转换成了ptr。

get\_user(cmd, (uint32\_t \_\_user \*)ptr)

get\_user直接从ptr中取出一个整数，这个整数就是对应的命令。binder的代码有些随意，还缺少做够的文档，如果不看代码，用户空间的框架代码根本无法开发。当然也有可能开发者是同一批人。

while (ptr < end && thread->return\_error.cmd == BR\_OK)

也就是说，一次可以传输过来多个命令进来，所以需要多次去取命令执行。

##### 执行命令

本来想在这儿展开看看强弱指针的处理，后来瞅瞅太麻烦了。还是后续单独说吧。这儿还是按着主线介绍。

case BC\_TRANSACTION:

case BC\_REPLY: {

struct binder\_transaction\_data tr;

if (copy\_from\_user(&tr, ptr, sizeof(tr)))

return -EFAULT;

ptr += sizeof(tr);

binder\_transaction(proc, thread, &tr,

cmd == BC\_REPLY, 0);

break;

}

可以看到BC\_TRANSACTION 和 BC\_REPLY 在一个函数里被处理了。那个程序员的故事里都在说功能不要耦合，听起来也像个神话故事。这也就意味着这个函数足够长，嗯，需要足够的耐心......

还好内核定义了一个binder\_transaction\_data的结构，后续要处理的就是这个结构体了。

#### 5.4 binder\_transaction 函数

这将是我们解析的最长的一个函数了，所以集中起精神吧。

##### 相关的数据结构

这里用到一些数据结构，阅读函数之前先简要的了解一下。

###### ·binder\_transaction\_data

struct binder\_transaction\_data {

/\* The first two are only used for bcTRANSACTION and brTRANSACTION,

\* identifying the target and contents of the transaction.

\*/

union {

/\* target descriptor of command transaction \*/

\_\_u32 handle;

/\* target descriptor of return transaction \*/

binder\_uintptr\_t ptr;

} target;

binder\_uintptr\_t cookie; /\* target object cookie \*/

\_\_u32 code; /\* transaction command \*/

/\* General information about the transaction. \*/

\_\_u32 flags;

pid\_t sender\_pid;

uid\_t sender\_euid;

binder\_size\_t data\_size; /\* number of bytes of data \*/

binder\_size\_t offsets\_size; /\* number of bytes of offsets \*/

/\* If this transaction is inline, the data immediately

\* follows here; otherwise, it ends with a pointer to

\* the data buffer.

\*/

union {

struct {

/\* transaction data \*/

binder\_uintptr\_t buffer;

/\* offsets from buffer to flat\_binder\_object structs \*/

binder\_uintptr\_t offsets;

} ptr;

\_\_u8 buf[8];

} data;

};

binder\_transaction\_data 与客服端交互使用的数据结构，传输所要的数据都在这里面。

###### ·binder\_transaction

struct binder\_transaction {

int debug\_id;

struct binder\_work work;

struct binder\_thread \*from;

struct binder\_transaction \*from\_parent;

struct binder\_proc \*to\_proc;

struct binder\_thread \*to\_thread;

struct binder\_transaction \*to\_parent;

unsigned need\_reply:1;

/\* unsigned is\_dead:1; \*/ /\* not used at the moment \*/

//共享内存

struct binder\_buffer \*buffer;

unsigned int code;

unsigned int flags;

long priority;

long saved\_priority;

kuid\_t sender\_euid;

struct list\_head fd\_fixups;

binder\_uintptr\_t security\_ctx;

/\*\*

\* @lock: protects @from, @to\_proc, and @to\_thread

\*

\* @from, @to\_proc, and @to\_thread can be set to NULL

\* during thread teardown

\*/

spinlock\_t lock;

};

binder\_transaction\_data的数据会被转化到binder\_transaction中，然后通知目标进程去处理。binder\_transaction是内核内部的数据结构。

为了理解binder\_transaction，来看看binder\_work是啥？

struct binder\_work {

struct list\_head entry;

enum {

BINDER\_WORK\_TRANSACTION = 1,

BINDER\_WORK\_TRANSACTION\_COMPLETE,

BINDER\_WORK\_RETURN\_ERROR,

BINDER\_WORK\_NODE,

BINDER\_WORK\_DEAD\_BINDER,

BINDER\_WORK\_DEAD\_BINDER\_AND\_CLEAR,

BINDER\_WORK\_CLEAR\_DEATH\_NOTIFICATION,

} type;

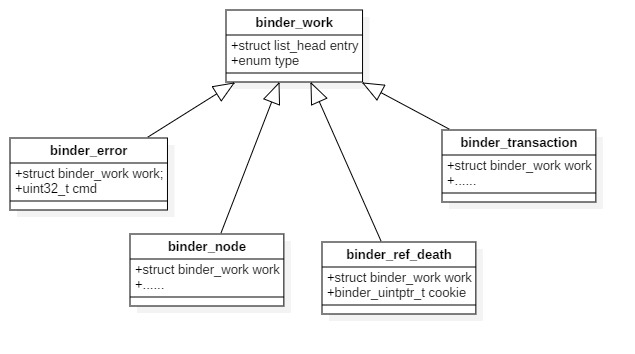
};

struct list\_head {

struct list\_head \*next, \*prev;

};

这里把list\_head也列了出来，可以看到它就是两个指针，作用就是把binder\_work给链接起来。binder\_work里包含一个list\_head entry对象，还有一个枚举值，两个变量。对binder\_work的使用，有点象面向对象里的继承方法，这里简单的描画一下：



如上图所示，binder\_error，binder\_node就像是binder\_work的继承类，这样他们就可以作为一个列表链接起来。而它们的连接的起点就是binder\_thread或者binder\_proc中的：

struct list\_head todo;

#### B, binder\_transaction 函数分解

binder\_transaction函数总共有700来行，不过也不需要太恐怖，其实还是可以分成不同的功能块来理解的。

在提醒一次，我们这里是按照

defaultServiceManager()->addService(

String16("media.audio\_flinger"), new AudioFlinger());

的实现来讲解代码，所以代码里很多分支在讲解过程中将不去除。

##### B1， 寻找target\_node 和 target\_thread

static void binder\_transaction(struct binder\_proc \*proc,

struct binder\_thread \*thread,

struct binder\_transaction\_data \*tr, int reply,

binder\_size\_t extra\_buffers\_size)

{

struct binder\_proc \*target\_proc = NULL;

struct binder\_thread \*target\_thread = NULL;

struct binder\_node \*target\_node = NULL;

if (reply) {

// 当客户进程处理完外部的请求后，会返回一个BC\_REPLY命令

// 这里我们先考虑BC\_TRANSACTION的处理，这部分略去。

} else {

// tr->target.handle 为0，对应的是管理者binder\_node

if (tr->target.handle) {

// handle的处理涉及到binder\_ref的实现，关于它下面会有专门的章节

} else {

mutex\_lock(&context->context\_mgr\_node\_lock);

target\_node = context->binder\_context\_mgr\_node;

if (target\_node)

target\_node = binder\_get\_node\_refs\_for\_txn(

target\_node, &target\_proc,

&return\_error);

else

return\_error = BR\_DEAD\_REPLY;

mutex\_unlock(&context->context\_mgr\_node\_lock);

if (target\_node && target\_proc == proc) {

goto err\_invalid\_target\_handle;

}

}

binder\_inner\_proc\_lock(proc);

// 这段代码检查binder\_thread的todo列表，如果列表中已经有BINDER\_WORK\_TRANSACTION

// 函数将直接返回错误信息。理论上不会有这种错误，除非是binder本身的代码有问题。

w = list\_first\_entry\_or\_null(&thread->todo,

struct binder\_work, entry);

if (!(tr->flags & TF\_ONE\_WAY) && w &&

w->type == BINDER\_WORK\_TRANSACTION) {

goto err\_bad\_todo\_list;

}

// 这里是寻找target\_thread的过程，看下面的图片说明。

if (!(tr->flags & TF\_ONE\_WAY) && thread->transaction\_stack) {

struct binder\_transaction \*tmp;

tmp = thread->transaction\_stack;

if (tmp->to\_thread != thread) {

goto err\_bad\_call\_stack;

}

while (tmp) {

struct binder\_thread \*from;

spin\_lock(&tmp->lock);

from = tmp->from;

if (from && from->proc == target\_proc) {

atomic\_inc(&from->tmp\_ref);

target\_thread = from;

spin\_unlock(&tmp->lock);

break;

}

spin\_unlock(&tmp->lock);

tmp = tmp->from\_parent;

}

}

binder\_inner\_proc\_unlock(proc);

}

以之前提到过的内容，内核会为用户空间的对象创建binder\_node对象，binder\_transaction的第一步就是找到这个对象。当前我们考虑的是 tr->target.handle为0 的情况，这是个特殊情况，其实对应的就是

target\_node = context->binder\_context\_mgr\_node。

有了target\_node，还需要确定target\_proc，这个其实就是binder\_node中的proc，代码如下：

static struct binder\_node \*binder\_get\_node\_refs\_for\_txn(

struct binder\_node \*node,

struct binder\_proc \*\*procp,

uint32\_t \*error)

{

struct binder\_node \*target\_node = NULL;

binder\_node\_inner\_lock(node);

if (node->proc) {

target\_node = node;

binder\_inc\_node\_nilocked(node, 1, 0, NULL);

binder\_inc\_node\_tmpref\_ilocked(node);

node->proc->tmp\_ref++;

\*procp = node->proc;

} else

\*error = BR\_DEAD\_REPLY;

binder\_node\_inner\_unlock(node);

return target\_node;

}

这段函数其实就是得到proc，如果没有proc，那么target\_node也会返回NULL。其中的

node->proc->tmp\_ref++;

tmp\_ref在临时使用到proc对象的时候会加1，以防proc被其它线程删除。当然有对应的减1函数：

static void binder\_proc\_dec\_tmpref(struct binder\_proc \*proc)

{

binder\_inner\_proc\_lock(proc);

proc->tmp\_ref--;

if (proc->is\_dead && RB\_EMPTY\_ROOT(&proc->threads) &&

!proc->tmp\_ref) {

binder\_inner\_proc\_unlock(proc);

binder\_free\_proc(proc);

return;

}

binder\_inner\_proc\_unlock(proc);

}

在 binder\_transaction 的最后几行，你可以看到对 binder\_proc\_dec\_tmpref的调用。

在了解寻找target\_thread的代码前，简单的看一下以下的图形（我自己是通过在纸上画以下的简图来熟悉这部分内容的，感兴趣的画，自己可以试着画画）：

线程3

线程2

线程1

BC\_TRANSACTION

BC\_TRANSACTION

这个图其实表达的意思很简单，就是说一个线程1可以发送BC\_TRANSACTION命令给线程2，而线程2在收到命令后可能的处理是再发送一个BC\_TRANSACTION给其它线程。

再回头看看这部分代码：

struct binder\_transaction \*tmp;

tmp = thread->transaction\_stack;

这里涉及到binder\_thread中的transaction\_stack对象，而transaction\_stack是binder\_transaction类型。我们首先看看transaction\_stack是从哪儿来的。

在 binder\_transaction函数的开头，我们可以看到一个定义：

struct binder\_transaction \*t;

这里先告诉大家，这个t将会保存到 transaction\_stack 中，所以我们这里要追踪t的具体内容了，代码就需要跳着去看了，当前还在binder\_transaction函数中：

if (!reply && !(tr->flags & TF\_ONE\_WAY))

t->from = thread;

else

t->from = NULL;

t->sender\_euid = task\_euid(proc->tsk);

t->to\_proc = target\_proc;

t->to\_thread = target\_thread;

t->code = tr->code;

t->flags = tr->flags;

t->priority = task\_nice(current);

TF\_ONE\_WAY表示该次传输，不需要返回结果，不需要返回的话t->from也不需要记录。

这里是一些简单的复值操作。再往下查，找transaction\_stack关键字：

} else if (!(t->flags & TF\_ONE\_WAY)) {

BUG\_ON(t->buffer->async\_transaction != 0);

binder\_inner\_proc\_lock(proc);

/\*

\* Defer the TRANSACTION\_COMPLETE, so we don't return to

\* userspace immediately; this allows the target process to

\* immediately start processing this transaction, reducing

\* latency. We will then return the TRANSACTION\_COMPLETE when

\* the target replies (or there is an error).

\*/

binder\_enqueue\_deferred\_thread\_work\_ilocked(thread, tcomplete);

t->need\_reply = 1;

t->from\_parent = thread->transaction\_stack;

thread->transaction\_stack = t;

binder\_inner\_proc\_unlock(proc);

if (!binder\_proc\_transaction(t, target\_proc, target\_thread)) {

binder\_inner\_proc\_lock(proc);

binder\_pop\_transaction\_ilocked(thread, t);

binder\_inner\_proc\_unlock(proc);

goto err\_dead\_proc\_or\_thread;

}

}

观察这段代码，当前我们关注的其实就两行：

t->from\_parent = thread->transaction\_stack;

thread->transaction\_stack = t;

对比以上的图，线程1发送请求时，thread->transaction\_stack应该为NULL，执行以上代码后，thread->transaction\_stack为t，也就是开头定义的binder\_transaction对象，而t->from\_parent为NULL。然后t将最为参数传入到 binder\_proc\_transaction 中进行处理，这时候t将被放入target\_proc或者target\_thread的todo列表去处理。这里可以看到thread，target\_thread其实都有指针指向了binder\_transaction对象。

略过一些细节，我们知道线程1生成的binder\_transaction对象被传入到线程2，而线程2将开始它的处理，这里是在函数binder\_thread\_read函数中，依然我们只关注binder\_transaction的生成（这里如果还是不太好理解的话，可以先放放，回头再看）：

case BINDER\_WORK\_TRANSACTION: {

binder\_inner\_proc\_unlock(proc);

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

} break;

binder\_thread\_read函数中会把之前保存的binder\_transaction对象处todo列表中取出来，中间的处理过程，我们略去了，看后面的处理过程：

if (cmd != BR\_REPLY && !(t->flags & TF\_ONE\_WAY)) {

binder\_inner\_proc\_lock(thread->proc);

t->to\_parent = thread->transaction\_stack;

t->to\_thread = thread;

thread->transaction\_stack = t;

binder\_inner\_proc\_unlock(thread->proc);

}

首先记住我们现在运行在线程2中，在线程1中设置了from\_parent,线程2中开始处理to\_parent，类似的处理。处理前thread->transaction\_stack为NULL，而后它指向了t。其实线程1和线程2的transaction\_stack指向的是同一个对象。

以上描述的是线程2内核层的处理，而后它的处理会转到用户层。这时候用户层处理以后一般会返回一个BC\_REPLY给线程1，流程结束。不过我们这儿讲的是另外一个可能性，也就是线程2要发送一个BC\_TRANSACTION给线程3。

那么这时候我们的分析又回到了binder\_transaction函数，但你应该记着这时候的主角是线程2。中间的过程我们都略过了，直接看到这儿：

t->from\_parent = thread->transaction\_stack;

thread->transaction\_stack = t;

线程2的 thread->transaction\_stack 之前已经被设置成了前一次传输的t,所以t->from\_parent 将不再为NULL,而是上次传输的对象。

这里讲的这么多，其实就是transaction\_stack的由来，你也看到了它有理由被命名为堆栈。

返回头，我们再次看寻找target\_thread的代码：

// 这里是寻找target\_thread的过程，看下面的图片说明。

if (!(tr->flags & TF\_ONE\_WAY) && thread->transaction\_stack) {

struct binder\_transaction \*tmp;

tmp = thread->transaction\_stack;

if (tmp->to\_thread != thread) {

goto err\_bad\_call\_stack;

}

while (tmp) {

struct binder\_thread \*from;

spin\_lock(&tmp->lock);

from = tmp->from;

if (from && from->proc == target\_proc) {

atomic\_inc(&from->tmp\_ref);

target\_thread = from;

spin\_unlock(&tmp->lock);

break;

}

spin\_unlock(&tmp->lock);

tmp = tmp->from\_parent;

}

先来看 tmp = thread->transaction\_stack; 这里你还是对照着上面的图来看，线程1运行的时候 tmp的值为NULL, 所以后续的代码不会执行了。

线程2执行的时候，它首先运行的是binder\_thread\_read，这时候它的transaction\_stack已经不为NULL。而后再由客户进程发起另外一个BC\_TRANSACTION命令，再次执行binder\_transcation函数，这时候tmp就不再为空。

tmp不为空，只有上面描述的一种可能性，就是连续调用BC\_TRANSACTION命令，所以这时候 tmp->to\_thread 也只有一种可能就是当前线程（图里的线程2）。所以这里的判断只是预防性判断，理论上除了内核代码错误，不会发生。

为了理解以下的while循环，重新画一下上面的图：

线程2

线程1

BC\_TRANSACTION

BC\_TRANSACTION

　这里变成了两个线程，线程２收到线程１的命令后，直接向线程１发送一个

BC\_TRANSACTION命令。这种情况下才会有from->proc == target\_proc的情况发生。

而后：

target\_thread = from;

这里的目的是线程１发送了命令给线程２，必然在等待线程２的执行。而线程２又需要线程１所在的进程的处理，那么内核认为线程１可以不再闲着了，起来干点活吧。绕这么一圈，其实目的就是这个，当然这里的实现极端依赖于客户空间代码的实现。

##### B2， 共享内存，binder\_transaction结构中binder\_buffer的处理

如果你从其它资料上参考过binder的优点，应该看到过内存间拷贝数据只需要一次。这里我们就来看看这个优点。

还记得binder提供mmap命令吧？

客户空间通过open打开binder设备后，都会调用mmap分配一段共享内存出来。关于这部分内存的具体用途代码基本都在binder\_alloc.c中，这里不会全部展开介绍这部分代码：一是因为这需要相当大的篇幅，二就是这部分代码不影响对binder整体的理解。

还是利用之前两个进程间传送命令的图形来解释这个问题吧：

a, 进程1 向进程2 发送一个BC\_TRANSACTION命令，这时候代码运行到了binder\_transaction，这时候是运行在进程1的内核空间。

b, binder\_transaction 函数的struct binder\_transaction\_data \*tr中，有以下成员变量：

union {

struct {

/\* transaction data \*/

binder\_uintptr\_t buffer;

/\* offsets from buffer to flat\_binder\_object structs \*/

binder\_uintptr\_t offsets;

} ptr;

\_\_u8 buf[8];

} data;

buffer 是一个指针，指向要传给进程2的数据，offsets是针对于buffer的一个偏移，从这个偏移开始保存flat\_binder\_object对象。

以下面的代码为例：

void AudioFlinger::instantiate() {

defaultServiceManager()->addService(

String16("media.audio\_flinger"), new AudioFlinger());

}

buffer开始处保存的是字符串“media.audio\_flinger”,offsets处保存的就是一个flat\_binder\_object对象：

struct flat\_binder\_object {

struct binder\_object\_header hdr;

\_\_u32 flags;

/\* 8 bytes of data. \*/

union {

binder\_uintptr\_t binder; /\* local object \*/

\_\_u32 handle; /\* remote object \*/

};

/\* extra data associated with local object \*/

binder\_uintptr\_t cookie;

};

而flat\_binder\_object对象的binder指向的是new AudioFlinger 返回的指针。

c, 这时候 binder\_transaction\_data中的buffer指针指向的进程1的用户空间指针，内核无法直接使用。另外这部分数据进程2的用户空间需要进行处理。所以这部分数据被拷贝到了进程2生成的共享空间。

这部分内容将被放入binder\_transaction结构的buffer中，所以可以看到以下的代码：

t->buffer = binder\_alloc\_new\_buf(&target\_proc->alloc, tr->data\_size,

tr->offsets\_size, extra\_buffers\_size,

!reply && (t->flags & TF\_ONE\_WAY));

注意到这里的buffer是由target\_proc的alloc分配的。

if (binder\_alloc\_copy\_user\_to\_buffer(

&target\_proc->alloc,

t->buffer, 0,

(const void \_\_user \*)

(uintptr\_t)tr->data.ptr.buffer,

tr->data\_size)) {

goto err\_copy\_data\_failed;

}

if (binder\_alloc\_copy\_user\_to\_buffer(

&target\_proc->alloc,

t->buffer,

ALIGN(tr->data\_size, sizeof(void \*)),

(const void \_\_user \*)

(uintptr\_t)tr->data.ptr.offsets,

tr->offsets\_size)) {

goto err\_copy\_data\_failed;

}

分别拷贝tr->data.ptr.buffer和tr->data.ptr.offsets的内容到binder\_transaction对象的buffer中。

这时候binder\_transaction中的buffer可供内核和进程2的用户空间使用。

d, 后续binder\_transcation函数将对binder\_transaction中的offsets对象进行调整。这个是下一部分要讲的内容。

e, 当binder\_tranaction作为一个work被进程2 的binder\_thread\_read处理的时候，共享内存中的内容将作为指针被直接传到客户空间。

可以关注一下binder\_thread\_read中的以下代码：

trd->data\_size = t->buffer->data\_size;

trd->offsets\_size = t->buffer->offsets\_size;

trd->data.ptr.buffer = (uintptr\_t)t->buffer->user\_data;

trd->data.ptr.offsets = trd->data.ptr.buffer +

ALIGN(t->buffer->data\_size,

sizeof(void \*));

##### B.3 针对binder\_object对象的处理

往下追踪binder\_transaction，来到了一个大型的循环结构：

off\_start\_offset = ALIGN(tr->data\_size, sizeof(void \*));

buffer\_offset = off\_start\_offset;

off\_end\_offset = off\_start\_offset + tr->offsets\_size;

sg\_buf\_offset = ALIGN(off\_end\_offset, sizeof(void \*));

sg\_buf\_end\_offset = sg\_buf\_offset + extra\_buffers\_size;

off\_min = 0;

for (buffer\_offset = off\_start\_offset; buffer\_offset < off\_end\_offset;

buffer\_offset += sizeof(binder\_size\_t)) {

struct binder\_object\_header \*hdr;

size\_t object\_size;

struct binder\_object object;

binder\_size\_t object\_offset;

binder\_alloc\_copy\_from\_buffer(&target\_proc->alloc,

&object\_offset,

t->buffer,

buffer\_offset,

sizeof(object\_offset));

object\_size = binder\_get\_object(target\_proc, t->buffer,

object\_offset, &object);

这个循环是针对binder\_transaction\_data中，我们之前提到过的offsets来处理的，也就是例子里的new AudioFlinger对象。看代码知道，类似的对象可能有多个，所以有循环。

继续看函数：

hdr = &object.hdr;

这里可以得到一个hdr对象，结构类型是：

struct binder\_object\_header {

\_\_u32 type;

};

type的值可以是：

enum {

BINDER\_TYPE\_BINDER = B\_PACK\_CHARS('s', 'b', '\*', B\_TYPE\_LARGE),

BINDER\_TYPE\_WEAK\_BINDER = B\_PACK\_CHARS('w', 'b', '\*', B\_TYPE\_LARGE),

BINDER\_TYPE\_HANDLE = B\_PACK\_CHARS('s', 'h', '\*', B\_TYPE\_LARGE),

BINDER\_TYPE\_WEAK\_HANDLE = B\_PACK\_CHARS('w', 'h', '\*', B\_TYPE\_LARGE),

BINDER\_TYPE\_FD = B\_PACK\_CHARS('f', 'd', '\*', B\_TYPE\_LARGE),

BINDER\_TYPE\_FDA = B\_PACK\_CHARS('f', 'd', 'a', B\_TYPE\_LARGE),

BINDER\_TYPE\_PTR = B\_PACK\_CHARS('p', 't', '\*', B\_TYPE\_LARGE),

};

这里简单描述一下这些枚举的意义：

BINDER\_TYPE\_BINDER和BINDER\_TYPE\_HANDLE的区别，还是addService的例子，进程new AudioFlinger，是进程自身空间的对象，new AudioFlinger属于BINDER\_TYPE\_BINDER。而别的进程中的对象（binder\_node），就需要用BINDER\_TYPE\_HANDLE来定义。比如我们的例子中 tr->target.handle的值是0，指向的是service manager中的管理binder\_node。

BINDER\_TYPE\_FD 和BINDER\_TYPE\_FDA应该是针对文件的操作，我没发现它的实际应用，这里不展开。BINDER\_TYPE\_PTR可能是纯指针处理，也没发现实际应用，不展开。

剩下的两个WEAK处理，我也没发现具体应用，可能和java有关吧，后续碰到再说吧。这里先不展开了，反正这里处理其实一样。

然后就是switch大展身手的地方了：

switch (hdr->type) {

case BINDER\_TYPE\_BINDER:

case BINDER\_TYPE\_WEAK\_BINDER: {

struct flat\_binder\_object \*fp;

fp = to\_flat\_binder\_object(hdr);

ret = binder\_translate\_binder(fp, t, thread);

if (ret < 0) {

goto err\_translate\_failed;

}

binder\_alloc\_copy\_to\_buffer(&target\_proc->alloc,

t->buffer, object\_offset,

fp, sizeof(\*fp));

} break;

目光转移到 binder\_translate\_binder，来看看它在做什么：

static int binder\_translate\_binder(struct flat\_binder\_object \*fp,

struct binder\_transaction \*t,

struct binder\_thread \*thread)

{

struct binder\_node \*node;

struct binder\_proc \*proc = thread->proc;

struct binder\_proc \*target\_proc = t->to\_proc;

struct binder\_ref\_data rdata;

int ret = 0;

node = binder\_get\_node(proc, fp->binder);

if (!node) {

node = binder\_new\_node(proc, fp);

if (!node)

return -ENOMEM;

}

ret = binder\_inc\_ref\_for\_node(target\_proc, node,

fp->hdr.type == BINDER\_TYPE\_BINDER,

&thread->todo, &rdata);

if (ret)

goto done;

if (fp->hdr.type == BINDER\_TYPE\_BINDER)

fp->hdr.type = BINDER\_TYPE\_HANDLE;

else

fp->hdr.type = BINDER\_TYPE\_WEAK\_HANDLE;

fp->binder = 0;

fp->handle = rdata.desc;

fp->cookie = 0;

done:

binder\_put\_node(node);

return ret;

}

binder\_get\_node 和 binder\_new\_node似曾相识，其实就是将fp->binder加入到binder\_proc的nodes中去。

if (fp->hdr.type == BINDER\_TYPE\_BINDER)

fp->hdr.type = BINDER\_TYPE\_HANDLE;

else

fp->hdr.type = BINDER\_TYPE\_WEAK\_HANDLE;

fp->binder = 0;

fp->handle = rdata.desc;

fp->cookie = 0;

再看这几行代码，BINDER\_TYPE\_BINDER被改成了 BINDER\_TYPE\_HANDLE,这是因为binder\_transaction对象将被目标进程处理，那时候就不是BINDER而是HANDLE了。所以fp->binder被设置成了0，而fp->handel 被设置成了rdata.desc。然后我们就需要回过头来看rdata.desc是如何来的了。

##### B.4 binder\_ref

接着上节，我们来看看fp->handle中保存的内容，这里就需要了解binder\_ref结构了。

struct binder\_ref\_data {

int debug\_id;

uint32\_t desc;

int strong;

int weak;

};

struct binder\_ref {

/\* Lookups needed: \*/

/\* node + proc => ref (transaction) \*/

/\* desc + proc => ref (transaction, inc/dec ref) \*/

/\* node => refs + procs (proc exit) \*/

struct binder\_ref\_data data;

struct rb\_node rb\_node\_desc;

struct rb\_node rb\_node\_node;

struct hlist\_node node\_entry;

struct binder\_proc \*proc;

struct binder\_node \*node;

struct binder\_ref\_death \*death;

};

binder\_ref中的node节点是指向它所引用的binder\_node节点，rb\_node\_desc与 rb\_node\_node用于将binder\_ref对象加入binder\_proc中的refs\_by\_desc与refs\_by\_node所指向的红黑树，加入两棵树是为了不同的情形下查询方便。其中refs\_by\_desc是以binder\_ref\_data中的desc来做关键字的，也就意味着同一个binder\_roc中，desc是唯一的。

而我们之前看到的handle，其实就是desc的值。另外binder\_node也需要记录指向它的binder\_ref有哪些， 所以这里还有一个node\_entry变量。

node 1

ref 1

ref 2

refs\_by\_desc

refs\_by\_node

nodes

ref 3

node 2

node 3

进程1 进程2

代表binder\_node中的rb\_node，形成一棵红黑树。由进程1中的nodes指向这棵红黑树。

代表binder\_ref中的node节点，它指向binder\_node对象。另外node\_entry会与binder\_node中的refs连成一个链表，这里的表示简化了，需要多画几个进程。

代表binder\_ref中的rb\_node\_node节点，形成一棵红黑树。由进程2中的refs\_by\_node指向它。

代表binder\_ref中的rb\_node\_desc节点，形成一棵红黑树。由进程2中的refs\_by\_desc指向它。

这里是个简化的图，其实一个进程里可能有node也有ref，ref可以指向不同的进程内的node等等情形。不过原理大体就是这样了。

然后binder\_translate\_handle等函数就可以自行理解了，还有函数开始部分的：

if (tr->target.handle) {

struct binder\_ref \*ref;

/\*

\* There must already be a strong ref

\* on this node. If so, do a strong

\* increment on the node to ensure it

\* stays alive until the transaction is

\* done.

\*/

binder\_proc\_lock(proc);

ref = binder\_get\_ref\_olocked(proc, tr->target.handle,

true);

if (ref) {

target\_node = binder\_get\_node\_refs\_for\_txn(

ref->node, &target\_proc,

&return\_error);

} else {

binder\_user\_error("%d:%d got transaction to invalid handle\n",

proc->pid, thread->pid);

return\_error = BR\_FAILED\_REPLY;

}

binder\_proc\_unlock(proc);

}

通过handle 找到target\_node，都不再难理解。

##### B.5 binder\_proc\_transaction函数

binder\_transaction函数的主要工作就是生成一个binder\_transaction对象，最后将生成的对象传递给目标进程。然后最后一步的任务就由binder\_proc\_transaction来完成：

static bool binder\_proc\_transaction(struct binder\_transaction \*t,

struct binder\_proc \*proc,

struct binder\_thread \*thread)

{

struct binder\_node \*node = t->buffer->target\_node;

bool oneway = !!(t->flags & TF\_ONE\_WAY);

bool pending\_async = false;

BUG\_ON(!node);

binder\_node\_lock(node);

if (oneway) {

BUG\_ON(thread);

if (node->has\_async\_transaction) {

pending\_async = true;

} else {

node->has\_async\_transaction = true;

}

}

binder\_inner\_proc\_lock(proc);

if (proc->is\_dead || (thread && thread->is\_dead)) {

binder\_inner\_proc\_unlock(proc);

binder\_node\_unlock(node);

return false;

}

if (!thread && !pending\_async)

thread = binder\_select\_thread\_ilocked(proc);

if (thread)

binder\_enqueue\_thread\_work\_ilocked(thread, &t->work);

else if (!pending\_async)

binder\_enqueue\_work\_ilocked(&t->work, &proc->todo);

else

binder\_enqueue\_work\_ilocked(&t->work, &node->async\_todo);

if (!pending\_async)

binder\_wakeup\_thread\_ilocked(proc, thread, !oneway /\* sync \*/);

binder\_inner\_proc\_unlock(proc);

binder\_node\_unlock(node);

return true;

}

这段代码的主要功能就是，如果没有指定的thread的话，就为它找一个thread，通过binder\_select\_thread\_ilocked来实现。如果有了指定线程将work加入它的todo列表：binder\_enqueue\_thread\_work\_ilocked。否则根据是否是oneway加入到proc的todo或者async\_todo列表。最后就是唤醒目标线程了：binder\_wakeup\_thread\_ilocked。

### 5.5 binder\_thread\_read 函数

上面的描述的内容理解后，binder\_thread\_read的内容就容易理解了。

通过 binder\_wait\_for\_work 等待当前线程或者进程被唤醒；通过binder\_dequeue\_work\_head\_ilocked去获取需要处理的work；这里主要处理的是BINDER\_WORK\_TRANSACTION类型的work，不过它也不介意接点私活，比如error处理等；然后它会对得到的work进行一些处理，就可以返回到用户空间了。

内核层的讲述就到此为止了，有些疑问可能在框架层中提到。