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。

一，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	1364 bytes
Makefile	96 bytes
binder.c	121569 bytes
binder_trace.h	8979 bytes

图-1 ： v.4.13 之前的 binder 目录

文件名

Kconfig	1787 bytes
Makefile	266 bytes
binder.c	181284 bytes
binder_alloc.c	32309 bytes
binder_alloc.h	6108 bytes
binder_alloc_selftest.c	8153 bytes
binder_internal.h	4067 bytes
binder_trace.h	11183 bytes
binderfs.c	19208 bytes

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。

1. 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。

1.1 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。

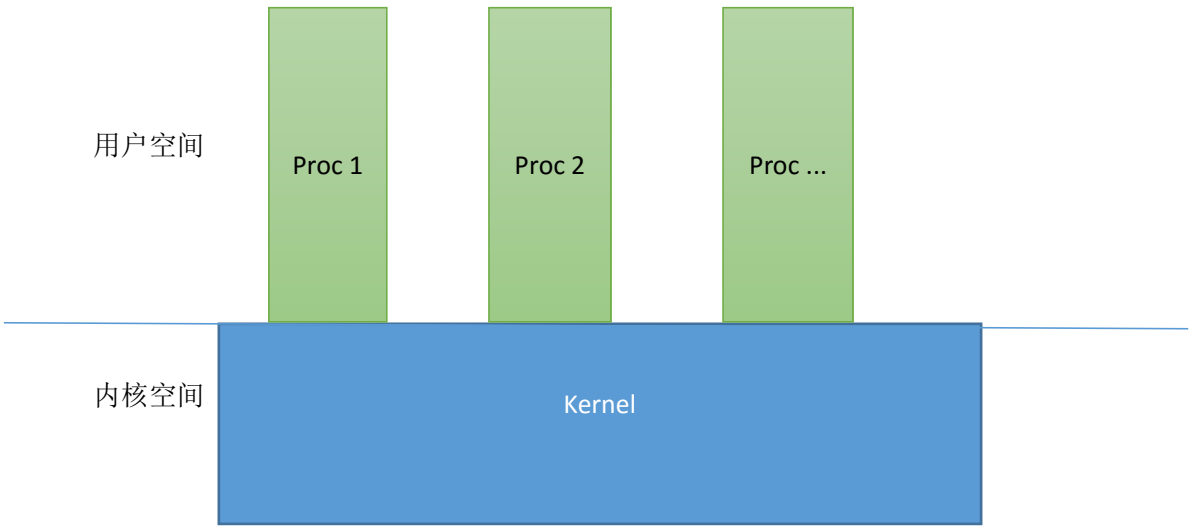


图 3 - 内核与用户空间

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。

1.2 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。

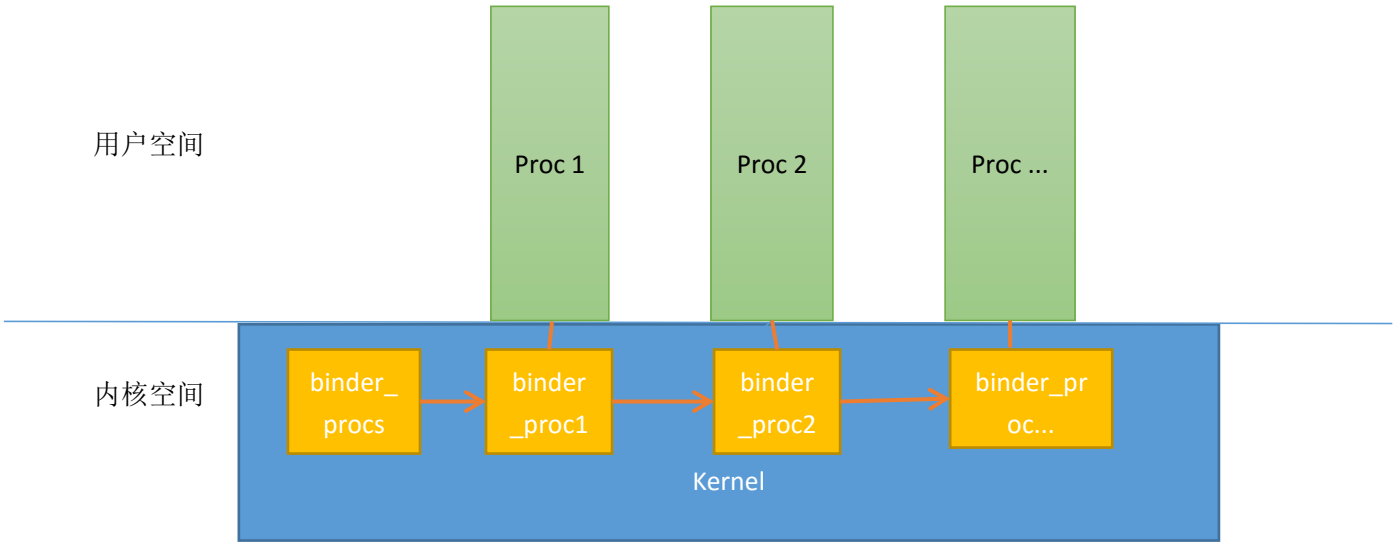


图 4 - Binder 内核示意

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。

2. 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。

3. 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。

3.1 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。

3.2 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。

4. 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。

B，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;
}
```

A， 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。

B， 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。

5. 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。

B. 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。

C. 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。

A， 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
```

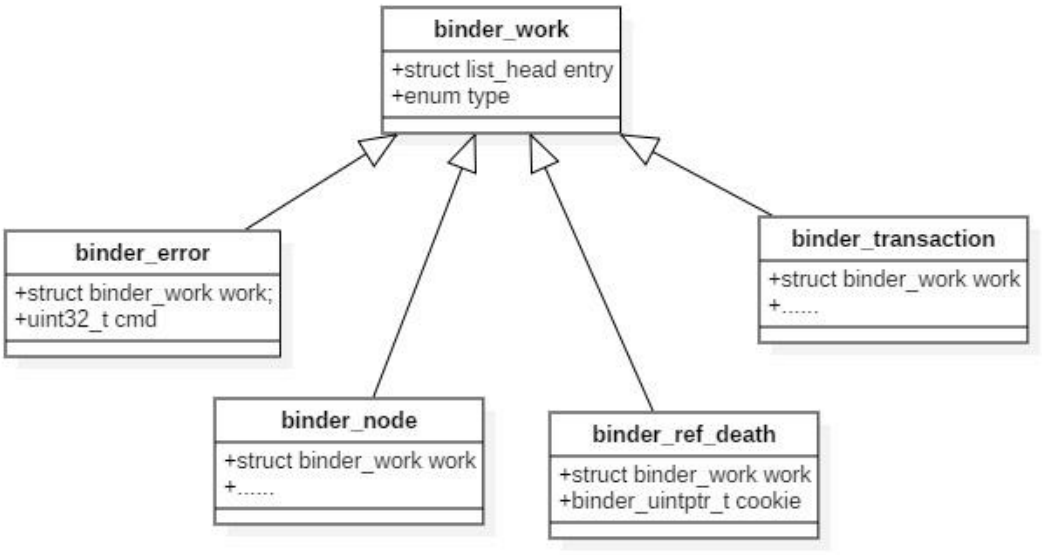
```
    struct binder_buffer *buffer;
    unsigned int    code;
    unsigned int    flags;
    long priority;
    long saved_priority;
    kuid_t      sender_euid;
    struct list_head fd_fixups;
    binder_uintptr_t security_ctx;
    /**
     * @lock:    protects @from, @to_proc, and @to_thread
     *
     * @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

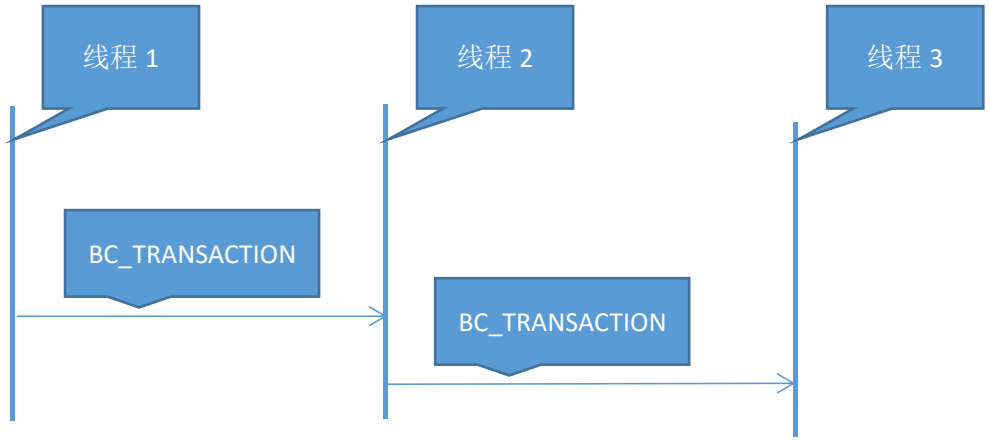
node->proc->tmp_ref++;

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）：



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;
```

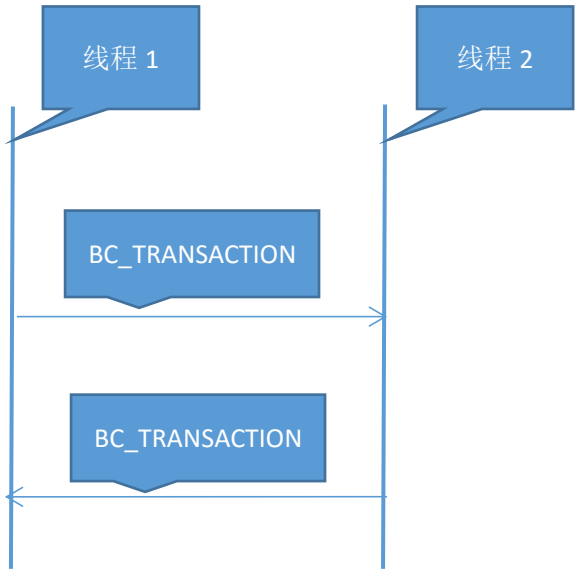
```
tmp = thread->transaction_stack;
if (tmp->to_thread != thread) {
        goto err_bad_call_stack;
}
while (tmp) {
        struct binder_thread *from;

        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：



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。

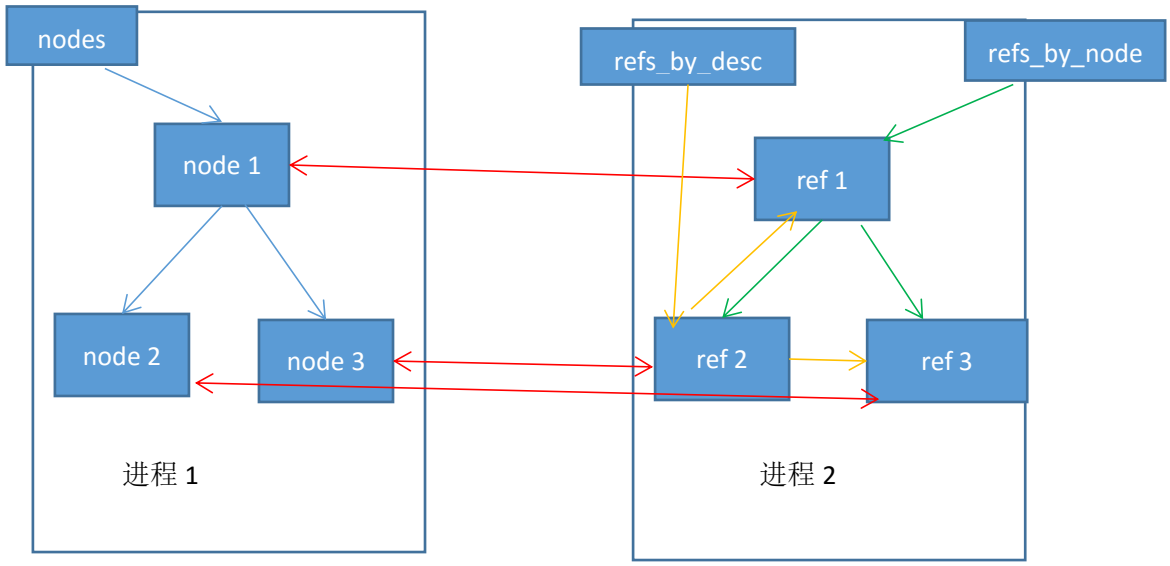
B.4 binder_ref

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。



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。