深入分析Android Binder 驱动

标签： Binder Android IPC Driver 2013-07-13 11:27 4511人阅读 评论(6) 收藏 举报

分类： 【Android Binder通信】（8）

目录(?) [-]

Binder通信是基于Service和Client的，所有需要IBinder通信的进程都必须创建一个IBinder接口。系统使用一个名为ServiceManager的收获进程管理着系统中的各个服务，它负责监听是否有其他程序向其发送请求，如果有请求就响应，如果没有，则继续监听等待。每个服务都要在ServiceManager中注册，而请求服务的客户端则向ServiceManager请求服务。在Android虚拟机启动之前，系统会先启动ServiceManager进程，ServiceManager会打开并通知Binder驱动程序自己将作为系统的服务管理者，然后ServiceManager进入一个循环，等待处理来自其他进程的数据。Android Binder是一种在Android里广泛使用的一种远程过程调用接口。从结构上来说Android Binder系统是一种服务器/客户机模式，包括Binder Server、Binder Client和Android Binder驱动，实际的数据传输就是通过Android Binder驱动来完成的，这里我们就来详细的介绍Android Binder驱动程序。

通常来说，Binder是Android系统中的内部进程通讯（IPC）之一。在Android系统中共有三种IPC机制，分别是：

- 标准Linux Kernel IPC接口
- 标准D-BUS接口
- Binder接口

尽管Google宣称Binder具有更加简洁、快速，消耗更小内存资源的优点，但并没有证据表明D-BUS就很差。实际上D-BUS可能会更合适些，或许只是当时Google并没有注意到它吧，或者Google不想使用GPL协议的D-BUS库。我们不去探究具体的原因了，你只要清楚Android系统中支持了多个IPC接口，而且大部分程序使用的是我们并不熟悉的Binder接口。

Binder是OpenBinder的Google精简实现，它包括一个Binder驱动程序、一个Binder服务器及Binder客户端（？）。这里我们只要介绍内核中的Binder驱动的实现。

对于Android Binder，它也可以称为是Android系统的一种RPC（远程过程调用）机制，因为Binder实现的功能就是在本地“执行”其他服务进程的功能的函数调用。不管是IPC也好，还是RPC也好，我们所要知道的就是Android Binder的功能是如何实现的。

Binder驱动原理

Binder驱动是作为一个特殊字符型设备存在，设备节点为/dev/binder，遵循Linux设备驱动模型。在驱动实现过程中，主要通过binder_ioctl函数与用户空间的进程交换数据。BINDER_WRITE_READ用来读写数据，数据包中有个cmd用于区分不同的请求。

binder_thread_write函数用于发送请求或返回结果，而binder_thread_read函数用于读取结果。在binder_thread_write函数中调用binder_transaction函数来转发请求并返回结果。当服务进程收到请求时，binder_transaction函数会通过对象的handle找到对象所在进程，如果handle为0，就认为请求的是ServiceManager进程。

Android Binder协议

Android 的Binder机制是基于OpenBinder来实现的，是一个OpenBinder的Linux实现。Android Binder的协议定义在binder.h头文件中，Android的通讯就是基于这样的一个协议的。

Binder Type

Android定义了五个（三大类）Binder类型，如下：

```cpp
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 Object

进程间传输的数据被称为Binder对象（Binder Object），它是一个flat_binder_object，定义如下：

```cpp
struct flat_binder_object {
    /* 8 bytes for large_flat_header. */
    unsigned long        type;
    unsigned long        flags;
    /* 8 bytes of data. */
    union {
        void        *binder;    /* local object */
        signed long    handle;        /* remote object */
    };
    /* extra data associated with local object */
    void            *cookie;
};
```

其中，类型字段描述了Binder对象的类型，flags描述了传输方式，比如同步、异步等。

```cpp
enum transaction_flags {
    TF_ONE_WAY    = 0x01,    /* this is a one-way call: async, no return */
    TF_ROOT_OBJECT    = 0x04,    /* contents are the component's root object */
    TF_STATUS_CODE    = 0x08,    /* contents are a 32-bit status code */
    TF_ACCEPT_FDS    = 0x10,    /* allow replies with file descriptors */
};
```

传输的数据是一个复用数据联合体，对于BINDER类型，数据就是一个binder本地对象，如果是HANDLE类型，这数据就是一个远程的handle对象。该如何理解本地binder对象和远程handle对象呢？其实它们都指向同一个对象，不过是从不同的角度来说。举例来说，假如A有个对象X，对于A来说，X就是一个本地的binder对象；如果B想访问A的X对象，这对于B来说，X就是一个handle。因此，从根本上来说handle和binder都指向X。本地对象还可以带有额外的数据，保存在cookie中。

Binder对象的传递是通过binder_transaction_data来实现的，即Binder对象实际是封装在binder_transaction_data结构体中。

binder_transaction_data

这个数据结构才是真正要传输的数据。它的定义如下：

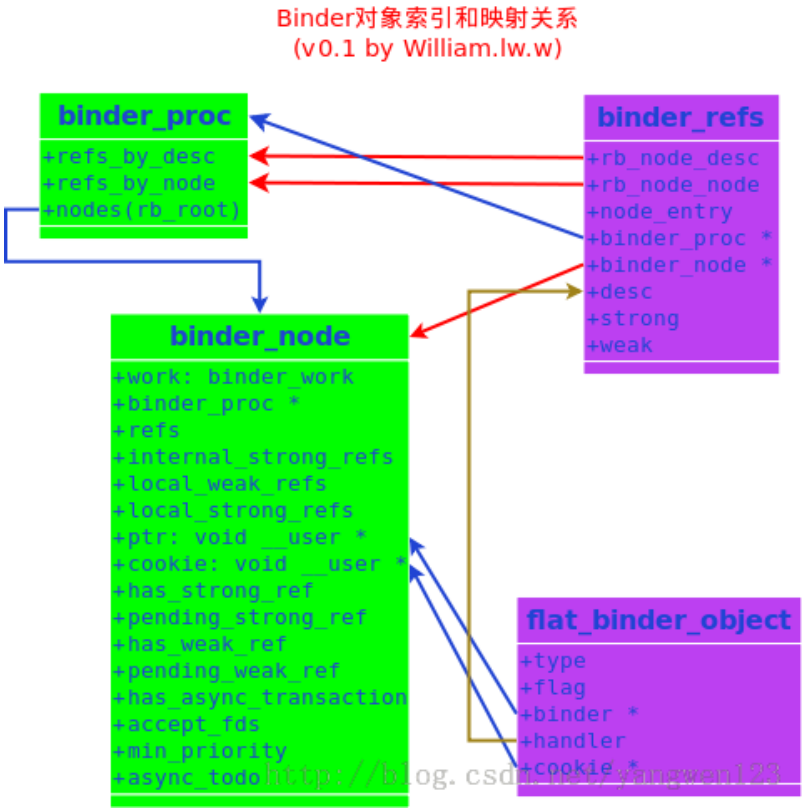
```cpp
struct binder_transaction_data {
    /* The first two are only used for bcTRANSACTION and brTRANSACTION,
     * identifying the target and contents of the transaction.
     */
    union {
        size_t    handle;    /* target descriptor of command transaction */
        void    *ptr;    /* target descriptor of return transaction */
    } target;
    void        *cookie;    /* target object cookie */
    unsigned int    code;        /* transaction command */
    /* General information about the transaction. */
    unsigned int    flags;
    pid_t        sender_pid;
    uid_t        sender_euid;
    size_t        data_size;    /* number of bytes of data */
    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 */
            const void    *buffer;
            /* offsets from buffer to flat_binder_object structs */
            const void    *offsets;
        } ptr;
        uint8_t    buf[8];
    } data;
};
```

结构体中的数据成员target是一个复合联合体对象，请参考前面的关于binder本地对象及handle远程对象的描述。code是一个命令，描述了请求Binder对象执行的操作。

对象的索引和映射

Binder中有两种索引，一是本地进程地址空间的一个地址，另一个是一个抽象的32位句柄（HANDLE），它们之间是互斥的：所有的进程本地对象的索引都是本地进程的一个地址(address, ptr, binder)，所有的远程进程的对象的索引都是一个句柄（handle）。对于发送者进程来说，索引就是一个远端对象的一个句柄，当Binder对象数据被发送到远端接收进程时，远端接受进程则会认为索引是一个本地对象地址，因此从第三方的角度来说，尽管名称不同，对于一次Binder调用，两种索引指的是同一个对象，Binder驱动则负责两种索引的映射，这样才能把数据发送给正确的进程。

对于Android的Binder来说，对象的索引和映射是通过binder_node和binder_ref两个核心数据结构来完成的，对于Binder本地对象，对象的Binder地址保存在binder_node->ptr里，对于远程对象，索引就保存在binder_ref->desc里，每一个binder_node都有一个binder_ref对象与之相联系，他们就是是通过ptr和desc来做映射的，如下图：



flat_binder_object就是进程间传递的Binder对象，每一个flat_binder_object对象内核都有一个唯一的binder_node对象，这个对象挂接在binder_proc的一颗二叉树上。对于一个binder_node对象，内核也会有一个唯一的binder_ref对象，可以这么理解，binder_ref的desc唯一的映射到binder_node的ptr和cookie上，同时也唯一的映射到了flat_binder_object的handler上。而binder_ref又按照node和desc两种方式映射到binder_proc对象上，也就是可以通过binder_node对象或者desc两种方式在binder_proc上查找到binder_ref或binder_node。所以，对于flat_binder_object对象来说，它的binder+cookie和handler指向了同一个binder_node对象上，即同一个binder对象。

BinderDriverCommandProtocol

Binder驱动的命令协议（BC_命令），定义了Binder驱动支持的命令格式及数据定义（协议）。不同的命令所带有的数据是不同的。Binder的命令由binder_write_read数据结构描述，它是ioctl命令（BINDER_WRITE_READ）的参数。

```cpp
struct binder_write_read {
    signed long    write_size;    /* bytes to write */
    signed long    write_consumed;    /* bytes consumed by driver */
    unsigned long    write_buffer;
    signed long    read_size;    /* bytes to read */
    signed long    read_consumed;    /* bytes consumed by driver */
    unsigned long    read_buffer;
};
```

对于写操作，write_buffer包含了一系列请求线程执行的Binder命令；对于读（返回）操作，read_buffer包含了一系列线程执行后填充的返回值。Binder命令（BC_）用于BINDER_WRITE_READ的write操作。

Binder的BC的命令格式是：| CMD | Data|| CMD | Data|..

```cpp
enum BinderDriverCommandProtocol {
    BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
    BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
    /*
     * binder_transaction_data: the sent command.
     */

    BC_ACQUIRE_RESULT = _IOW('c', 2, int),
    /*
     * not currently supported
```

```
     * int:  0 if the last BR_ATTEMPT_ACQUIRE was not successful.
     * Else you have acquired a primary reference on the object.
     */

    BC_FREE_BUFFER = _IOW('c', 3, int),
    /*
     * void *: ptr to transaction data received on a read
     */

    BC_INCREFS = _IOW('c', 4, int),
    BC_ACQUIRE = _IOW('c', 5, int),
    BC_RELEASE = _IOW('c', 6, int),
    BC_DECREFS = _IOW('c', 7, int),
    /*
     * int: descriptor
     */

    BC_INCREFS_DONE = _IOW('c', 8, struct binder_ptr_cookie),
    BC_ACQUIRE_DONE = _IOW('c', 9, struct binder_ptr_cookie),
    /*
     * void *: ptr to binder
     * void *: cookie for binder
     */

    BC_ATTEMPT_ACQUIRE = _IOW('c', 10, struct binder_pri_desc),
    /*
     * not currently supported
     * int: priority
     * int: descriptor
     */

    BC_REGISTER_LOOPER = _IO('c', 11),
    /*
     * No parameters.
     * Register a spawned looper thread with the device.
     */

    BC_ENTER_LOOPER = _IO('c', 12),
    BC_EXIT_LOOPER = _IO('c', 13),
    /*
     * No parameters.
     * These two commands are sent as an application-level thread
     * enters and exits the binder loop, respectively.  They are
     * used so the binder can have an accurate count of the number
     * of looping threads it has available.
     */

    BC_REQUEST_DEATH_NOTIFICATION = _IOW('c', 14, struct binder_ptr_cookie),
    /*
     * void *: ptr to binder
     * void *: cookie
     */

    BC_CLEAR_DEATH_NOTIFICATION = _IOW('c', 15, struct binder_ptr_cookie),
    /*
     * void *: ptr to binder
     * void *: cookie
     */

    BC_DEAD_BINDER_DONE = _IOW('c', 16, void *),
    /*
     * void *: cookie
     */
};
```

BinderDriverReturnProtocol

Binder驱动的响应（返回，BR_）协议，定义了Binder命令的数据返回格式。同Binder命令协议一样，Binder驱动返回协议也是通过

BINDER_WRITE_READ ioctl命令实现的，不同的是它是read操作。

Binder BR的命令格式是：| CMD | Data|| CMD | Data|..

```cpp
enum BinderDriverReturnProtocol {
    BR_ERROR = _IOR('r', 0, int),
    /*
     * int: error code
     */

    BR_OK = _IO('r', 1),
    /* No parameters! */

    BR_TRANSACTION = _IOR('r', 2, struct binder_transaction_data),
    BR_REPLY = _IOR('r', 3, struct binder_transaction_data),
    /*
     * binder_transaction_data: the received command.
     */

    BR_ACQUIRE_RESULT = _IOR('r', 4, int),
    /*
     * not currently supported
     * int: 0 if the last bcATTEMPT_ACQUIRE was not successful.
     * Else the remote object has acquired a primary reference.
     */

    BR_DEAD_REPLY = _IO('r', 5),
    /*
     * The target of the last transaction (either a bcTRANSACTION or
     * a bcATTEMPT_ACQUIRE) is no longer with us.  No parameters.
     */

    BR_TRANSACTION_COMPLETE = _IO('r', 6),
    /*
     * No parameters... always refers to the last transaction requested
     * (including replies).  Note that this will be sent even for
     * asynchronous transactions.
     */

    BR_INCREFS = _IOR('r', 7, struct binder_ptr_cookie),
    BR_ACQUIRE = _IOR('r', 8, struct binder_ptr_cookie),
    BR_RELEASE = _IOR('r', 9, struct binder_ptr_cookie),
    BR_DECREFS = _IOR('r', 10, struct binder_ptr_cookie),
    /*
     * void *:  ptr to binder
     * void *: cookie for binder
     */

    BR_ATTEMPT_ACQUIRE = _IOR('r', 11, struct binder_pri_ptr_cookie),
    /*
     * not currently supported
     * int: priority
     * void *: ptr to binder
     * void *: cookie for binder
     */

    BR_NOOP = _IO('r', 12),
    /*
     * No parameters.  Do nothing and examine the next command.  It exists
     * primarily so that we can replace it with a BR_SPAWN_LOOPER command.
     */

    BR_SPAWN_LOOPER = _IO('r', 13),
    /*
     * No parameters.  The driver has determined that a process has no
     * threads waiting to service incomming transactions.  When a process
     * receives this command, it must spawn a new service thread and
```

```
     * register it via bcENTER_LOOPER.
     */

    BR_FINISHED = _IO('r', 14),
    /*
     * not currently supported
     * stop threadpool thread
     */

    BR_DEAD_BINDER = _IOR('r', 15, void *),
    /*
     * void *: cookie
     */
    BR_CLEAR_DEATH_NOTIFICATION_DONE = _IOR('r', 16, void *),
    /*
     * void *: cookie
     */

    BR_FAILED_REPLY = _IO('r', 17),
    /*
     * The the last transaction (either a bcTRANSACTION or
     * a bcATTEMPT_ACQUIRE) failed (e.g. out of memory).  No parameters.
     */
};
```

驱动接口

Android Binder设备驱动接口函数是*device_initcall(binder_init);*

我们知道一般来说设备驱动的接口函数是module_init和module_exit，这么做是为了同时兼容支持静态编译的驱动模块（buildin）和动态编译的驱动模块（module）。但是Android的Binder驱动显然不想支持动态编译的驱动，如果你需要将Binder驱动修改为动态的内核模块，可以直接将device_initcall修改为module_init，但不要忘了增加module_exit的驱动卸载接口函数。

binder_init

初始化函数首先创建了一个内核工作队列对象（workqueue），用于执行可以延期执行的工作任务：

```
[cpp]
static struct workqueue_struct *binder_deferred_workqueue;
binder_deferred_workqueue = create_singlethread_workqueue("binder");
```

挂在这个workqueue上的work是binder_deferred_work，定义如下。当内核需要执行work任务时，就通过workqueue来调度执行这个work了。

```
[cpp]
static DECLARE_WORK(binder_deferred_work, binder_deferred_func);
queue_work(binder_deferred_workqueue, &binder_deferred_work);
```

既然说到了binder_deferred_work，这里有必要来进一步说明一下，binder_deferred_work是在函数binder_defer_work里调度的：

```
[cpp]
static void binder_defer_work(struct binder_proc *proc, enum binder_deferred_state defer)
{
    mutex_lock(&binder_deferred_lock);
    proc->deferred_work |= defer;
    if (hlist_unhashed(&proc->deferred_work_node)) {
        hlist_add_head(&proc->deferred_work_node,
                &binder_deferred_list);
        queue_work(binder_deferred_workqueue, &binder_deferred_work);
    }
    mutex_unlock(&binder_deferred_lock);
}
```

deferred_work有三种类型，分别是BINDER_DEFERRED_PUT_FILES，BINDER_DEFERRED_FLUSH和BINDER_DEFERRED_RELEASE。它们都操作在binder_proc对象上。

```cpp
enum binder_deferred_state {
    BINDER_DEFERRED_PUT_FILES    = 0x01,
    BINDER_DEFERRED_FLUSH        = 0x02,
    BINDER_DEFERRED_RELEASE      = 0x04,
};
```

初始化函数接着使用proc_mkdir创建了一个Binder的proc文件系统的根节点（binder_proc_dir_entry_root，/proc/binder），并为binder创建了binder proc节点（binder_proc_dir_entry_proc，/proc/binder/proc），然后Binder驱动使用misc_register把自己注册为一个Misc设备（/dev/misc/binder）。最后，如果驱动成功的创建了/proc/binder根节点，就调用create_proc_read_entry创建只读proc文件：/proc/binder/state，/proc/binder/stats，/proc/binder/transactions，/proc/binder/transaction_log，/proc/binder/failed_transaction_log。

用户接口

驱动程序的一个主要同能就是向用户空间的程序提供操作接口，这个接口是标准的，对于Android Binder驱动，包含的接口有：

– Proc接口（/proc/binder）

. */proc/binder/state*

. */proc/binder/stats*

. */proc/binder/transactions*

. */proc/binder/transaction_log*

./proc/binder/failed_transaction_log

. */proc/binder/proc/*

-设备接口（/dev/binder）

. *binder_open*

. *binder_release*

. *binder_flush*

. *binder_mmap*

. *binder_poll*

. *binder_ioctl*

这些内核接口函数是在驱动程序的初始化函数(binder_init)中初始化的

binder_open

通常来说，驱动程序的open函数是用户调用驱动接口来使用驱动功能的第一个函数，称为入口函数。同其他驱动一样，对于Android驱动，任何一个进程及其内的所有线程都可以打开一个binder设备。首先来看看Binder驱动是如何打开设备的。

首先，binder驱动分配内存以保存binder_proc数据结构。然后，binder填充binder_proc数据（初始化），增加当前线程/进程的引用计数并赋值给tsk

```cpp
get_task_struct(current);
proc->tsk = current;
```

初始化binder_proc的队列及默认优先级

```cpp
INIT_LIST_HEAD(&proc->todo);
init_waitqueue_head(&proc->wait);
proc->default_priority = task_nice(current);
```

增加BINDER_STAT_PROC的对象计数，并把创建的binder_proc对象添加到全局的binder_proc哈希列表中，这样任何一个进程就都可以

访问到其他进程的binder_proc对象了。

```cpp
binder_stats.obj_created[BINDER_STAT_PROC]++;
hlist_add_head(&proc->proc_node, &binder_procs);
```

把当前进程/线程的线程组的pid（pid指向线程id）赋值给proc的pid字段，可以理解为一个进程id（thread_group指向线程组中的第一个线程的task_struct结构）。同时把binder_proc对象指针赋值给filp的private_data对象保存起来。

```cpp
proc->pid = current->group_leader->pid;
INIT_LIST_HEAD(&proc->delivered_death);
filp->private_data = proc;
```

最后，在bindr proc目录中创建只读文件（/proc/binder/proc/$pid）用来输出当前binder proc对象的状态。这里要注意的是，proc->pid字段，按字面理解它应该是保存当前进程/线程的id，但实际上它保存的是线程组的pid，也就是线程组中的第一个线程的pid（等于tgid，进程id）。这样当一个进程或线程打开一个binder设备时，驱动就会在内核中为其创建binder_proc结构来保存打开此设备的进程/线程信息。

binder_release

binder_release是一个驱动的出口函数，当进程退出(exit)时，进程需要显示或隐式的调用release函数来关闭打开的文件。Release函数一般来清理进程的内核数据，释放申请的内存。Binder驱动的release函数相对比较简单，它删除open是创建的binder proc文件，然后调度一个workqueue来释放这个进程/线程的binder_proc对象（BINDER_DEFERRED_RELEASE）。这里要提的一点就是使用workqueue（deffered）可以提高系统的响应速度和性能，因为Android Binder的release及flush等操作是一个复杂费事的任务，而且也没有必要在系统调用里完成它，因此最好的方法是延迟执行这个费时的任务。其实在中断处理函数中我们经常要把一些耗时的操作放到底半部中处理（bottom half），这是一样的道理。当然真正的释放工作是在binder_deferred_release函数里完成的.

binder_flush

flush操作在关闭一个设备文件描述符拷贝时被调用，Binder的flush函数十分简单，它只是简单的调度一个workqueue执行BINDER_DEFERRED_FLUSH操作。flush操作比较简单，内核只是唤醒所有睡眠在proc对象及其thread对象上的所有函数。

binder_mmap

mmap（memory map）用于把设备内存映射到用户进程地址空间中，这样就可以像操作用户内存那样操作设备内存。（还有一种说法，mmap用于把用户进程地址空间映射到设备内存上，尽管说法不同，但是说的是同一个事情）

Binder设备对内存映射是有些限制的，比如binder设备最大能映射4M的内存区域；binder不能映射具有写权限的内存区域。

不同于一般的设备驱动，大多的设备映射的设备内存是设备本身具有的，或者在驱动初始化时由vmalloc或kmalloc等内核内存函数分配的，Binder的设备内存是在mmap操作时分配的，分配的方法是先在内核虚拟映射表上获取一个可以使用的区域，然后分配物理页，并把物理页映射到获取的虚拟空间上。由于设备内存是在mmap操作中实现的，因此每个进程/线程只能做映射操作一次，其后的操作都会返回错误。

具体来说，binder_mmap首先检查mmap调用是否合法，即是否满足binder内存映射的条件，主要检查映射内存的大小、flags，是否是第一次调用。

```cpp
if ((vma->vm_end - vma->vm_start) > SZ_4M)
vma->vm_end = vma->vm_start + SZ_4M;
if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
…
}
vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;
if (proc->buffer) {
…
}
```

然后，binder驱动从系统申请可用的虚拟内存空间（注意不是物理内存空间），这是通过get_vm_area内核函数实现的：（get_vm_area

是一个内核，功能是在内核中申请并保留一块连续的内核虚拟内存空间区域）

```cpp
area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
proc->buffer = area->addr;
proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;
```

然后根据请求映射的内存空间大小，分配binder核心数据结构binder_proc的pages成员，它主要用来保存指向申请的物理页的指针。

```cpp
proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma->vm_start) / PAGE_SIZE), GFP_KERNEL);
proc->buffer_size = vma->vm_end - vma->vm_start;
```

一切都准备就绪了，现在开始分配物理内存（page）了。这是通过binder驱动的帮助函数binder_update_page_range来实现的。尽管名字上称为update page，但在这里它是在分配物理页并映射到刚才保留的虚拟内存空间上。当然根据参数，它也可以释放物理页面。在这里，Binder使用alloc_page分配页面，使用map_vm_area为分配的内存做映射关系，使用vm_insert_page把分配的物理页插入到用户vma区域。函数传递的参数很有意识，我们来看一下：

```cpp
binder_update_page_range(proc, 1, proc->buffer, proc->buffer + PAGE_SIZE, vma);
```

开始地址是proc->buffer很容易理解，但是也许你会奇怪为什么结束地址是proc->buffer+PAGE_SIZE？这是因为，在这里并没有全部分配物理内存，其实只是分配了一个页的物理内存（第一，函数是在分配物理页，就是说物理内容的分配是以页面为单位的，因此所谓的开始地址和结束地址是用来计算需要分配多少物理页面的，这是开始地址和结束地址的含义。第二，前面已经提到mmap的最大物理内存是4M，因此需要的最多的pages就是1K，一个指针是4个字节，因此最大的page指针buffer就是4K，也就是一个页的大小。第三，不管申请mmap物理内存是多大，内核总是分配一个页的指针数据，也就是每次都分配最多的物理页。）下面来看看物理页是如何分配的：

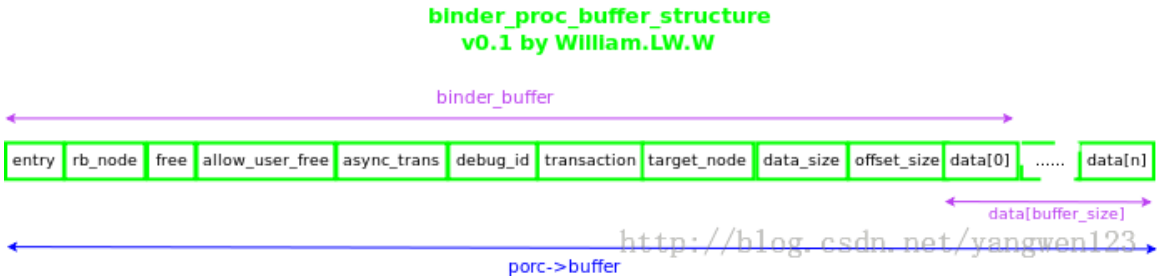
```cpp
*page = alloc_page(GFP_KERNEL | __GFP_ZERO);
tmp_area.addr = page_addr;
tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
page_array_ptr = page;
ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
user_page_addr = (uintptr_t)page_addr + proc->user_buffer_offset;
ret = vm_insert_page(vma, user_page_addr, page[0]);
```

注：alloc_page, map_vm_area和vm_insert_page都是Linux内核中内存相关函数。

物理页分配完后，这个物理内存就交给binder_buffer来管理了，刚刚申请的物理内存以binder_buffer的方式放到proc->buffers链表里。

```cpp
struct binder_buffer *buffer;
buffer = proc->buffer;
INIT_LIST_HEAD(&proc->buffers);
list_add(&buffer->entry, &proc->buffers);
buffer->free = 1;
binder_insert_free_buffer(proc, buffer);
proc->free_async_space = proc->buffer_size / 2;
```

binder_buffer和proc->buffer的关系如下图：



当然，这里会使用到的核心数据结构binder_proc，用到的主要域是

buffer 记录binder_proc的内核虚拟地址的首地址

buffer_size 记录binder_proc的虚拟地址的大小

user_buffer_offset 记录binder_proc的用户地址偏移，即用户进程vma地址与binder申请的vma地址的偏差

pages 记录指向binder_proc物理页（struct page）的指针（二维指针）

files 记录进程的struct file_struct 结构

vma 记录用户进程的vma结构

binder_poll

poll函数是非阻塞型IO（select，poll调用）的内核驱动实现，所有支持非阻塞IO操作的设备驱动都需要实现poll函数。Binder的poll函数仅支持设备是否可非阻塞的读（POLLIN），这里有两种等待任务，一种是一种是proc_work，另一种是thread_work。同其他驱动的poll实现一样，这里也是通过调用poll_wait函数来实现的，这里就不多做叙述了。

```cpp
poll_wait(filp, &thread->wait, wait);
```

这里需要明确提的一点就是这里调用了下面的函数来取得当前进程/线程的thread对象：

```cpp
thread = binder_get_thread(proc);
```

特别要指出的，也是很重要的一点，就是这个函数实际上在为服务进程的线程池创建对应的thread对象。后面还会详细讲解这个函数。首先介绍一下的是这个函数会查找当前进程/线程的thread对象，thread对象根据pid值保存在：struct rb_node **p = &proc->threads.rb_node;的红黑树中，如果没有找到，线程就会创建一个thread对象并且根据pid的值把新创建的thread对象插入到红黑树中。

binder_ioctl

这个函数是Binder的最核心部分，Binder的功能就是通过ioctl命令来实现的。Binder的ioctl命令共有7个，定义在ioctl.h头文件中：

```cpp
#define BINDER_WRITE_READ            _IOWR('b', 1, struct binder_write_read)
#define BINDER_SET_IDLE_TIMEOUT      _IOW('b', 3, int64_t)
#define BINDER_SET_MAX_THREADS       _IOW('b', 5, size_t)
#define BINDER_SET_IDLE_PRIORITY     _IOW('b', 6, int)
#define BINDER_SET_CONTEXT_MGR       _IOW('b', 7, int)
#define BINDER_THREAD_EXIT           _IOW('b', 8, int)
#define BINDER_VERSION               _IOWR('b', 9, struct binder_version)
```

首先要说明的是BINDER_SET_IDLE_TIMEOUT 和 BINDER_SET_IDLE_PRIORITY在目前的Binder驱动中没有实现。

这里最重要的就是BINDER_WRITE_READ命令，它是Binder驱动核心的核心，Binder IPC机制主要是通过这个命令来实现的。下面我们首先来介绍简单的用于设置Binder驱动参数的几个ioctl命令，最后着重讲述BINDER_WRITE_READ命令。

来看这个函数的功能，当应用程序一进入ioctl调用的时候，驱动就检查是否有错误，如果有错误的话，应用程序将要睡眠到binder_user_error_wait的等待队列上，直到没有错误或是睡眠被信号中断。

```cpp
ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
if (ret)
    return ret;
```

注：wait_event_interruptible是一个内核等待队列函数，它是进程睡眠知道等待的条件为真，或是被signal唤醒而返回-ERESTARTSYS错误。

如果没有用户错误，那么驱动就调用函数binder_get_thread来取得或创建当前线程/进程的thread对象，关于这个函数马上就会介绍到了。这里要说明的是每一个线程/进程都有一个对应的thread对象。

1. BINDER_SET_MAX_THREADS

这个ioctl命令用于设置进程的Biner对象所支持的最大线程数。设置的值保存在binder_proc结构的max_threads成员里。

2. BINDER_SET_CONTEXT_MGR

在这里会引入Binder的另一个核心数据binder_node。从功能上看，只有一个进程/线程能成功设置binder_context_mgr_node对象，这个进程被称为Context Manager（context_mgr）。当然，也只有创建binder_context_mgr_node对象的Binder上下文管理进程/线程才有权限重新设置这个对象。进程的权限（cred->euid）保存在binder_context_mgr_uid对象里。

```cpp
binder_context_mgr_uid = current->cred->euid;
```

从接口的角度来说，这是一个进程想要成为一个Context Manager的唯一接口。一个Context Manager进程需要为binder_proc创建一个binder_node类型的节点。节点是通过*binder_new_node*函数来创建的，我们在后面在详细讲解这个函数。节点创建成功后内核会初始化节点的部分数据（*weak_ref*和*strong_ref*）。

```cpp
binder_context_mgr_node->local_weak_refs++;
binder_context_mgr_node->local_strong_refs++;
binder_context_mgr_node->has_strong_ref = 1;
binder_context_mgr_node->has_weak_ref = 1;
```


binder_proc成员node是binder_node的根节点，这是一棵红黑树（一种平衡二叉树）。现在来看看binder_new_node函数，它首先根据规则找到第一个页节点作为新插入的节点的父节点（规则就是binder_node的指针对象，后面还会遇到）。

```cpp
while (*p) {
    parent = *p;
    node = rb_entry(parent, struct binder_node, rb_node);
    if (ptr < node->ptr)
        p = &(*p)->rb_left;
    else if (ptr > node->ptr)
        p = &(*p)->rb_right;
    else
        return NULL;
}
```


找到节点了，那么调用内核函数创建并插入节点吧

```cpp
node = kzalloc(sizeof(*node), GFP_KERNEL);
binder_stats.obj_created[BINDER_STAT_NODE]++;
rb_link_node(&node->rb_node, parent, p);
rb_insert_color(&node->rb_node, &proc->nodes);
```


注：rb_link_node和rb_insert_color都是内核红黑树函数，rb_link_node是一个内联函数，它把新节点插入到红黑树中的指定父节点下。rb_insert_color节把已经插入到红黑树中的节点调整并融合到红黑树中（参考根据红黑树规则）。
插入完成后，做最后的初始化工作，这里着重说明两点，一是把binder_proc对象指针保存在binder_node对象里，二是初始化node对象的链表头指针。

```cpp
```

```
node->proc = proc;
node->ptr = ptr;
node->cookie = cookie;
node->work.type = BINDER_WORK_NODE;
INIT_LIST_HEAD(&node->work.entry);
INIT_LIST_HEAD(&node->async_todo);
```

这里还要说明一下的是，对于ContextManager对象来说，binder_node是*binder_context_mgr_node*，这个是全局变量；binder对象的索引（handler）固定为0。要记住这一点，后面还会遇到的。

3. BINDER_THREAD_EXIT

通过调用binder_free_thread终止并释放binder_thread对象及其binder_transaction事务。

4. BINDER_VERSION

读取当前Binder驱动支持的协议版本号。

5. BINDER_WRITE_READ

前面提到，这个ioctl命令才是Binder最核心的部分，Android Binder的IPC机制就是通过这个接口来实现的。我们在这里来介绍binder_thread对象，其实在前面已经见到过了，但是因为它与这个接口更加紧密，因此我们把它拿到这里来介绍。

每一个进程的binder_proc对象都有一个binder_thread对象队列（保存在proc->threads.rb_node节点队列里），每一个进程/线程的id(pid)就保存在线程自己的binder_thread结构的pid成员里。Binder_thread对象是在binder_get_thread函数中创建的，ioctl函数在入口处会调用它来取得或创建binder_thread对象：

```
[cpp]
thread = binder_get_thread(proc);
```

binder_thread对象保存在binder_proc对象的thread成员里，同binder_node一样，它是一棵红黑树。首先我们先来看一下binder_get_thread函数。这个函数还是比较简单的，它遍历thread树找到同当前进程相关的binder_thread对象，判断条件就是当前进程/线程的pid要等于thread对象里记录的pid，看下面的代码：

```
[cpp]
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
        break;
}
```

如果找到了binder_thread对象，就直接返回该对象。如果没有找到，就说明当前进程/线程的binder_thread对象还没有创建，创建一个新的binder_thread节点并插入到红黑树中，返回这个新创建的binder_thread对象。当然，这里还要对binder_thread对象最一个初始化工作

```
[cpp]
thread = kzalloc(sizeof(*thread), GFP_KERNEL);
binder_stats.obj_created[BINDER_STAT_THREAD]++;
thread->proc = proc;
thread->pid = current->pid;
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 |= BINDER_LOOPER_STATE_NEED_RETURN;
thread->return_error = BR_OK;
thread->return_error2 = BR_OK;
```

这里要说明的是，进程/线程有两个通过binder_get_thread创建进程/线程的binder_thread对象，一个是在调用binder_poll()的时候，另一个是在调用binder_ioctl的时候。

在介绍BINDER_WRITE_READ之前，我们先来看一下接口的参数，我们知道，每一个ioctl命令都可以有一个数据参数，BINDER_WRITE_READ的参数是一个binder_write_read类型的数据结构，它描述了可读写的数据，BINDER_WRITE_READ就是根据它的具体内容来做写操作或是读操作。

```cpp
struct binder_write_read {
    signed long    write_size;    /* bytes to write */
    signed long    write_consumed;    /* bytes consumed by driver */
    unsigned long    write_buffer;
    signed long    read_size;    /* bytes to read */
    signed long    read_consumed;    /* bytes consumed by driver */
    unsigned long    read_buffer;
};
```

其中，x_size表示有多少个字节需要读写，x_consumed表示了已经被内核读写了多少数据，x_buffer是指向读写数据的指针。具体的读写操作是通过两个核心函数来实现的，分别是binder_thread_write和binder_thread_read。当有写数据的时候，binder_thread_write函数就会被调用来发送binder命令，当有读数据的使用，binder_thread_read就会被调用来读取binder命令。

```cpp
if (bwr.write_size > 0) {
    ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer, bwr.write_size, &bwr.
}
if (bwr.read_size > 0) {
    ret = binder_thread_read(proc, thread, (void __user *)bwr.read_buffer, bwr.read_size, &bwr.rea
    filp->f_flags & O_NONBLOCK);
}
```

binder_thread_write

Binder协议的命令（BC_和BR_）就是通过这个BINDER_WRITE_READ ioctl接口实现的，驱动也按照Binder命令的格式（命令+参数）来分类处理，下面我们具体来看一下这些命令的实现。

bwr.write_buffer里保存的就是指向Binder协议命令的指针数据，首先，函数要得到这个数据的首尾地址，并且根据条件来处理所有可处理的Binder命令。

```cpp
void __user *ptr = buffer + *consumed;
void __user *end = buffer + size;
while (ptr < end && thread->return_error == BR_OK) {
    ......
```

按照Binder命令协议，首先读取Binder命令，由于buffer里只是指向命令的指针，实际数据还保存在用户空间，因此调用get_user函数从用户空间读取数据。取得命令后，先更新命令的状态信息，即增加命令的使用计数用于统计。

```cpp
if (get_user(cmd, (uint32_t __user *)ptr))
if (_IOC_NR(cmd) < ARRAY_SIZE(binder_stats.bc)) {
            binder_stats.bc[_IOC_NR(cmd)]++;
            proc->stats.bc[_IOC_NR(cmd)]++;
            thread->stats.bc[_IOC_NR(cmd)]++;
        }
```

然后根据读取的命令按照Binder命令的协议分类处理。首先来看看这四个命令：

- BC_INCREFS
- BC_ACQUIRE
- BC_RELEASE
- BC_DECREFS

这四个命令是Binder的binder_ref对象的操作命令，用于增加或减少对象的引用计数，其命令格式是：

cmd | desc

binder_ref是Android Binder驱动的另一个核心数据结构，用于描述Binder节点对象的对象索引，对象索引由binder_ref->desc来描述。节点对象的binder_ref索引是通过函数binder_get_ref_for_node来创建的，我们先来看看这个函数。

同其他binder对象一样，binder_ref也是保存在一个红黑树中的，函数首先在红黑树中查找是否已经为节点创建ref索引了：

```cpp
struct rb_node **p = &proc->refs_by_node.rb_node;
struct rb_node *parent = NULL;
struct binder_ref *ref, *new_ref;
while (*p) {
    parent = *p;
    ref = rb_entry(parent, struct binder_ref, rb_node_node);
    if (node < ref->node)
        p = &(*p)->rb_left;
    else if (node > ref->node)
        p = &(*p)->rb_right;
    else
        return ref;
}
```

如果还没有为节点对象创建ref索引，就为这个节点创建一个新的索引对象，并把binder_proc和binder_node对象赋值给索引对象，然后link到proc的refs_by_node红黑树中：

```cpp
new_ref = kzalloc(sizeof(*ref), GFP_KERNEL);
new_ref->proc = proc;
new_ref->node = node;
rb_link_node(&new_ref->rb_node_node, parent, p);
rb_insert_color(&new_ref->rb_node_node, &proc->refs_by_node);
```

然后为新的索引赋值，这里要说明的是管理节点（binder_context_mgr_node，也就是Context Manager进程的节点）的索引固定为0，非管理节点的索引从1开始递增，这样对于新创建索引的索引值desc，都是树中最大的索引值加1，然后在插入到proc->refs_by_desc红黑树中。这样对于一个索引，即可以通过node对像，也可以通过索引对象查找。

```cpp
new_ref->desc = (node == binder_context_mgr_node) ? 0 : 1;    //初始值
for (n = rb_first(&proc->refs_by_desc); n != NULL; n = rb_next(n)) {
    ref = rb_entry(n, struct binder_ref, rb_node_desc);
    if (ref->desc > new_ref->desc)
        break;
    new_ref->desc = ref->desc + 1;  //把节点树中最大的desc值加1，赋值给新节点的desc
}
/* 根据索引值desc，插入到refs_by_desc数中*/
p = &proc->refs_by_desc.rb_node;
while (*p) {
    parent = *p;
    ref = rb_entry(parent, struct binder_ref, rb_node_desc);
    if (new_ref->desc < ref->desc)
        p = &(*p)->rb_left;
    else if (new_ref->desc > ref->desc)
        p = &(*p)->rb_right;
    else
        BUG();
```

```
}
rb_link_node(&new_ref->rb_node_desc, parent, p);
rb_insert_color(&new_ref->rb_node_desc, &proc->refs_by_desc);
```

最后把新索引插入到节点对象的哈斯列表里，并返回新创建的binder_ref对象。再来看我们的binder_thread_write函数，前面我们已经读到了四个跟binder_ref相关的命令，取得这四个命令后，驱动进入这四个命令的处理分支。首先，按照协议，继续读取四个字节的target descriptor。target descriptor描述了命令作用的对象，即前面我们提到的binder_ref里的desc。

```
[cpp]
if (get_user(target, (uint32_t __user *)ptr))
```

得到target desc后，根据这个desc值查询得到它所代表的binder_ref对象，这里有两个处理流程，一个是对于*Context Manager*节点，如果是Context Manager节点，并且是BC_INCREFS或者BC_ACQUIRE操作的话，需要用binder_get_ref_for_node函数，因为有可能需要为Context Manager节点创建新的索引对象。如果不是的话，就直接调用binder_get_ref取得binder_ref对象。

```
[cpp]
if (target == 0 && binder_context_mgr_node &&
    (cmd == BC_INCREFS || cmd == BC_ACQUIRE)) {
    ref = binder_get_ref_for_node(proc, binder_context_mgr_node);
    if (ref->desc != target) {
        binder_user_error("binder: %d:%d tried to acquire reference to desc 0, got %d instead\n",
            proc->pid, thread->pid, ref->desc);
    }
} else
ref = binder_get_ref(proc, target);
```

得到target desc所代表的binder_ref对象后，就通过binder_ref增加节点对象的引用计数，这里又根据操作命令的不同分为Strong和Weak两种操作。对于BC_ACQUIRE/BC_RELEASE所谓的Strong操作，会增加/减少节点对象的strong计数，对于BC_INCREFS/BC_DECREFS所谓的Weak操作，会增加或者减少节点对象的weak计数。

```
[cpp]
case BC_INCREFS:
        binder_inc_ref(ref, 0, NULL);
        break;
case BC_ACQUIRE:
        binder_inc_ref(ref, 1, NULL);
        break;
case BC_RELEASE:
        binder_dec_ref(ref, 1);
        break;
case BC_DECREFS:
        binder_dec_ref(ref, 0);
```

我们继续来探讨一下binder_inc_ref和binder_dec_ref两个函数。这两个函数在管理节点索引的像，如果节点索引被使用（INCREFS活ACQUIRE），就增加索引对象的引用计数（strong++或者weak++），当然，对于第一个strong或者weak索引对象，还会相应的增加索引映射的节点对象的使用计数（还有一些其他的操作）。同理如果节点对象被释放（DECREFS和RELEASE），就减少索引对象的引用计数，如果strong和weak都减少到0了，就表示没有程序在使用这个索引对象，就可以删除索引了。

```
[cpp]
static int binder_inc_ref(struct binder_ref *ref, int strong, struct list_head *target_list)
{
    int ret;
    if (strong) {
        if (ref->strong == 0) {
            ret = binder_inc_node(ref->node, 1, 1, target_list);
            if (ret)
                return ret;
        }
        ref->strong++;
```

```
    } else {
        if (ref->weak == 0) {
            ret = binder_inc_node(ref->node, 0, 1, target_list);
            if (ret)
                return ret;
        }
        ref->weak++;
    }
    return 0;
}
static int binder_dec_ref(struct binder_ref *ref, int strong)
{
    if (strong) {
        if (ref->strong == 0) {
            binder_user_error("binder: %d invalid dec strong, "
                      "ref %d desc %d s %d w %d\n",
                      ref->proc->pid, ref->debug_id,
                      ref->desc, ref->strong, ref->weak);
            return -EINVAL;
        }
        ref->strong--;
        if (ref->strong == 0) {
            int ret;
            ret = binder_dec_node(ref->node, strong, 1);
            if (ret)
                return ret;
        }
    } else {
        if (ref->weak == 0) {
            binder_user_error("binder: %d invalid dec weak, "
                      "ref %d desc %d s %d w %d\n",
                      ref->proc->pid, ref->debug_id,
                      ref->desc, ref->strong, ref->weak);
            return -EINVAL;
        }
        ref->weak--;
        }
        if (ref->weak == 0) {
            int ret;
            ret = binder_dec_node(ref->node, strong, 1);
            if (ret)
                return ret;
        }
    }
    if (ref->strong == 0 && ref->weak == 0)
        binder_delete_ref(ref);
    return 0;
}
```

这里有一个问题，就是在减少ref->weak的引用计数的时候，为了保持函数逻辑的一致性，应该在ref->weak减少到0的时候调用*binder_dec_node(ref->node, strong, 1)*，比如代码中的红色部分。当然不调用也没关系，即使strong减少到0的时候也可以不调用这个函数，因为在binder_delete_ref函数里就有相关的处理，但是显然调用跟能保持函数逻辑的一致性。

注：binder_inc_node函数会增加节点的internal_strong_refs，local_strong_refs或者local_weak_refs的使用计数，并且把node->work.entry添加到链表target_list里；同理，binder_dec_node函数会减少internal_strong_refs，local_strong_refs或者local_weak_refs的使用计数，并删除节点的work.entry链表，这里不再详细描述。

下面我们来看另外两个命令，他们跟前面的四个命令是有关系的，

BC_INCREFS_DONE

BC_ACQUIRE_DONE

这两个命令在INCREFS或者ACQUIRE使用完的时候会发送，命令格式是：CMD | ptr | cookie

同前面的命令处理一样，首先按照协议从用户空间读取node_ptr和cookie数据，在根据读取的node->ptr查询到对应的节点对象。这个节点对象就是之前进程/线程根据自己的ptr和cookie创建的。这里不会创建节点对象，因此如果节点不存在或者cookie值不对，都直接返回错

误。

```cpp
if (get_user(node_ptr, (void * __user *)ptr))
            return -EFAULT;
ptr += sizeof(void *);
if (get_user(cookie, (void * __user *)ptr))
            return -EFAULT;
ptr += sizeof(void *);
node = binder_get_node(proc, node_ptr);
if (node == NULL) {
            binder_user_error("binder: %d:%d %s u%p no match\n",
            ......
            break;
}
if (cookie != node->cookie) {
            binder_user_error("binder: %d:%d %s u%p node %d"
            ......
            break;
}
```

然后根据命令，设置节点的pending_strong_ref或者pending_weak_ref为零，并且调用函数binder_dec_node来减少节点的使用计数，这两个命令就处理完成了。

```cpp
if (cmd == BC_ACQUIRE_DONE) {
    node->pending_strong_ref = 0;
}
else{
    node->pending_weak_ref = 0;
}
binder_dec_node(node, cmd == BC_ACQUIRE_DONE, 0);
```

这个命令是有关于Binder的buffer管理的，我们姑且现在这里做简单的介绍，详细的内容会在后面单独详细的讲解Android的Binder Buffer管理机制。

BC_FREE_BUFFER

命令格式是：CMD | data_ptr (*data_ptr指向在read操作中接收到的transaction data*)

同样，这个命令首先从用户空间读取data_ptr数据，然后根据data_ptr数据，在binder_proc对象里查找binder_buffer对象

```cpp
get_user(data_ptr, (void * __user *)ptr)
buffer = binder_buffer_lookup(proc, data_ptr);
```

首先药判断是否查找到buffer对象，或者buffer对象是否允许用户释放（buffer->allow_user_free）。满足这些条件后，做释放前的一些准备工作：将binder_buffer里的binder_transaction的buffer及transaction对象都设置为空，同时，根据*是否有async_transaction操作，将has_async_transaction设置为0，或者把async_todo.next移动到thread->todo的链表尾上。*

```cpp
if (buffer->transaction) {
        buffer->transaction->buffer = NULL;
        buffer->transaction = NULL;
}
if (buffer->async_transaction && buffer->target_node) {
        BUG_ON(!buffer->target_node->has_async_transaction);
        if (list_empty(&buffer->target_node->async_todo))
            buffer->target_node->has_async_transaction = 0;
        else
            list_move_tail(buffer->target_node->async_todo.next, &thread->todo);
}
```

最后调用函数释放binder_buffer对象：

```cpp
binder_transaction_buffer_release(proc, buffer, NULL);
binder_free_buf(proc, buffer);
```

BC_TRANSACTION

BC_REPLY

命令格式是：CMD | binder_transaction_data

这两个命令应该是Binder驱动里最最核心的两个命令了，Binder机制的数据传递过程就是在这两个命令里完成的。首先，同前面两个命令一样，根据Binder协议，驱动首先从用户空间读取命令的参数binder_transaction_data，然后调用binder_transaction函数，实际的数据传输就是在这个函数里完成的。

```cpp
struct binder_transaction_data tr;
if (copy_from_user(&tr, ptr, sizeof(tr)))
    return -EFAULT;
binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
```

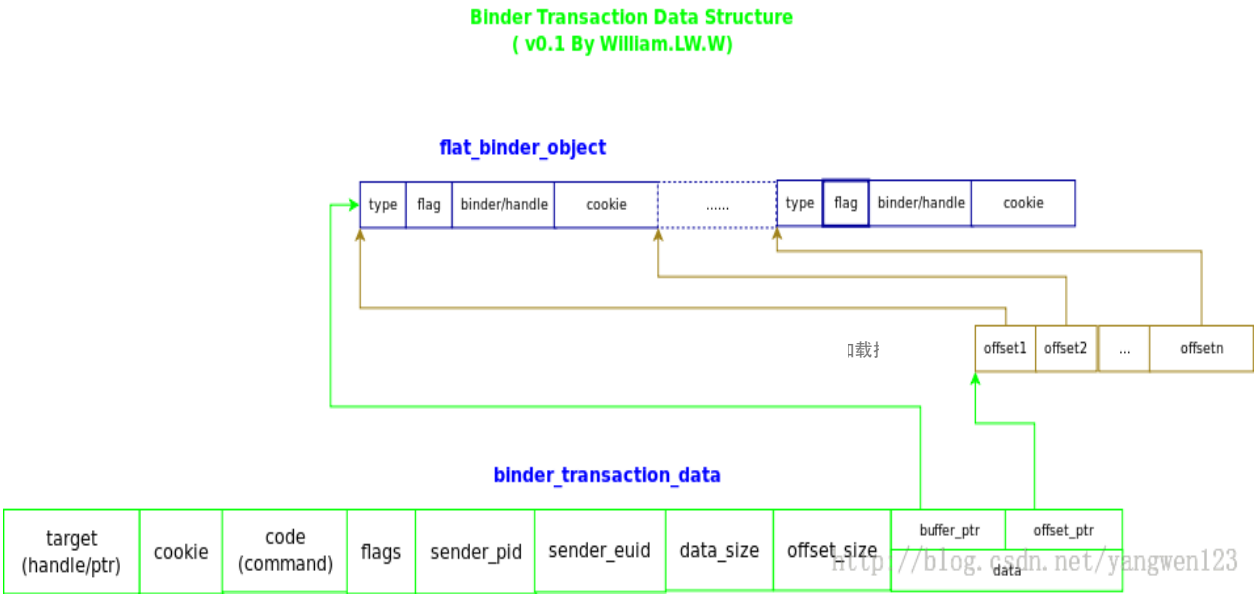
下面我们来看binder_transaction这个函数，根据命令的不同，函数有两个分支，一个是TRANSACTION处理，另一个是REPLY分支处理。这个函数的写法实在不敢恭维，由于混合了两种不同的分支处理，函数的易读性比较差，也导致这个函数比较大，大概有400多行。其实完全可以根据不同的功能，把公共的代码独立为一个或者几个函数，而把不同的功能拆分为几个不同函数，这样也许更好些。或者Google认为也许作为一个函数的话性能更好?

TRANSACTION处理流程

我们先来看Transaction命令的处理流程，这时，*structbinder_transaction_data*里的数据含义是：

tr->target.handle ->远程对象索引，即binder_ref里的desc

下面的图描述了binder_transaction_data里数据组织结构。Binder驱动要从用户空间读取这些数据发送给目标对象。



驱动程序首先根据binder_transaction_data数据里的目标/远程对象索引（target.handle）查找到目标节点对象，进而得到目标进程（binder_proc）对象。前面提到过，如果handle是0，就表示目标节点是Context Manger的节点。

```cpp
if (tr->target.handle) {
        struct binder_ref *ref;
        ref = binder_get_ref(proc, tr->target.handle);
        target_node = ref->node;
} else {
```

```
        target_node = binder_context_mgr_node;
}
target_proc = target_node->proc;
```

如果本次binder_transaction不是异步传输（flag不是TF_ONE_WAY），并且thread->transaction_stack（就是struct binder_transaction的链表）不是空，就说明目前正位于Binder传输的中间环节，需要根据from_parent查找到目标thread对象。在前面的操作中，thread->transaction_stack的to_thread被设置为它自己，因此，如果不是的话就说明整个thread的transaction堆栈有问题。如果没问题，就根据前面得到的target_proc回溯查找到target_thread。这里有一个疑问，就是当查找到的时候，程序不是中断查找，而是继续回溯查找，从逻辑上看就是要找到最靠近堆栈根的那个对象，是不是传输堆栈中一个thread对象可以被放到传输堆栈中多次，是这样么？

这里要解释一下什么是binder_transaction对象。可以这么理解，binder_transaction_data是binder传输对象的外部表示，应用于应用程序的，而binder_transaction是binder传输对象的内部表示，应用于内核binder驱动本身。binder_transaction对象都位于binder_thread的传输栈上，其本身是一个多级链表结构，描述了传输来源和传输目标，也记录了本次传输的信息，如binder_work、binder_buffer、binder命令等。

```cpp
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) {
            if (tmp->from && tmp->from->proc == target_proc)
                target_thread = tmp->from;
            tmp = tmp->from_parent;
        }
}
```

如果查找到target_thread，那么他就是目标线程，否则，target_proc就是传输的目标。根据传输的目标设置本次binder传输的目标等待队列(wait_queue)和本次binder_work需要挂载的列表(list)，也就是target_wait和target_list:

```cpp
if (target_thread) {
            e->to_thread = target_thread->pid;
            target_list = &target_thread->todo;
            target_wait = &target_thread->wait;
} else {
            target_list = &target_proc->todo;
            target_wait = &target_proc->wait;
}
```

回忆一下binder_poll函数，target_wait就是进程/线程在poll函数里的等待队列，也就是本次binder_transaction最后要唤醒的本次传输的目标进程/线程。到目前，target_node，target_thread，target_proc，target_wait和target_list都已经找到了。下面就该为此次传输分配新的binder_transaction对象和binder_work对象了，并根据当前的信息填充内容。首先填充基本内容：sender_euid、to_proc、to_thread、code、flag、priority，如果不是异步传输，就把当前的binder_thread对象赋值给from，否者from就为空，以便记录此次binder_transaction对象的来源。

```cpp
t = kzalloc(sizeof(*t), GFP_KERNEL);
tcomplete = kzalloc(sizeof(*tcomplete), GFP_KERNEL);
if (!reply && !(tr->flags & TF_ONE_WAY))
        t->from = thread;                               //同步传输
else
        t->from = NULL;
t->sender_euid = proc->tsk->cred->euid;
t->to_proc = target_proc;
t->to_thread = target_thread;
t->code = tr->code;
t->flags = tr->flags;
```

```
12.  t->priority = task_nice(current);
```

然后为binder_transaction分配buffer，这是一个binder_buffer对象，由函数binder_alloc_buf来分配，就如上面的图形所示，buffer的数据大小和offset数量都通过binder_transaction_data由用户空间传过来：

```
[cpp]
01.  t->buffer = binder_alloc_buf(target_proc, tr->data_size, tr->offsets_size, !reply && (t-
     >flags & TF_ONE_WAY));
```

回忆一下在前面的binder_mmap操作中，已经为proc->buffer分配了一个页的物理内存，并且以binder_buffer的形式添加到proc->free_buffers树中。binder_alloc_buf就是到proc->free_buffers树中查找是否有可用的物理内存空间，如果有，就在这个binder_buffer对象上分配物理页，并将这个binder_buffer从free_buffers上移到alloc_buffers上。当然如果binder_buffer对象的物理内存空间比实际需要的物理空间大的话，就将剩余的内存移出来分配给新的binder_buffer对象并添加到free_buffers树中供下次使用。如果成功的申请到了binder_buffer空间，就填充数据到binder_buffer上，并将用户空间的data和offset数据拷贝到binder_buffer的数据里。

```
[cpp]
01.  t->buffer->allow_user_free = 0;
02.  t->buffer->transaction = t;
03.  t->buffer->target_node = target_node;
04.  copy_from_user(t->buffer->data, tr->data.ptr.buffer, tr->data_size)
05.  offp = (size_t *)(t->buffer->data + ALIGN(tr->data_size, sizeof(void *)));
06.  copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_size)
```

有一点要注意到，就是data里的数据就是flat_binder_object，也就是进程/线程间传递的binder对象。到现在所有的数据都取得了，下一步就要根据binder协议来传输flat_binder_objec了。但是在传输之前还要做一个对象的索引和映射的转换，就是，如果binder对象，就要转换为reference对象，如果是reference对象，就要转换为binder对象。这是因为对于发送者和接收者来说，尽管是同一个对象，但确是从不同的角度来识别对象的。驱动会在buffer->data里遍历所有的flat_binder_object，根据对象的索引和映射法则修改flat_binder_object的type和bnder/handle数据。

第一，如果发送端是binder对象（BINDER或WEAK_BINDER），驱动就从本地proc里查找binder_node节点，如果节点还没有创建就创建一个新的节点对象

```
[cpp]
01.  struct binder_node *node = binder_get_node(proc, fp->binder);
02.      if (node == NULL) {
03.          node = binder_new_node(proc, fp->binder, fp->cookie);
04.  }
```

找到/创建了binder_node对象后，就根据这个node对象在目标进程(target_proc)里查找活创建这个节点的索引对象（binder_ref，也可称为参考对象）。有了binder_ref索引对象后，就讲索引ID（ref->desc）赋值给flat_binder_object，同时更改flat_binder_object的type为HANDLE或者WEAK_HANDLE，并增加索引和节点的引用计数。

```
[cpp]
01.  if (fp->type == BINDER_TYPE_BINDER)
02.          fp->type = BINDER_TYPE_HANDLE;
03.  else
04.          fp->type = BINDER_TYPE_WEAK_HANDLE;
05.  fp->handle = ref->desc;
06.  binder_inc_ref(ref, fp->type == BINDER_TYPE_HANDLE, &thread->todo);
```

第二，如果发送端是索引对象（HANDLE或WEAK_HANDLE），驱动就根据flat_binder_object里的handle从本地的proc里读取binder_ref对象：

```
[cpp]
01.  struct binder_ref *ref = binder_get_ref(proc, fp->handle);
```

如果找到binder_ref对象，并且相对的binder_node里的proc对象就是target_proc对象，那么这就是我们需要的索引对象了，为什么这么说

呢？这是因为节点对象只有一个，但是每个节点在不同的进程上都可能会有一个索引对象，所以对于一个固定的节点对象，索引对象可以有多个，但是对于给定的进程(binder_proc)，每个节点的索引对象是唯一的。类似鱼前面的binder对象，根据找到的索引对象转换flat_binder_object以指向对应的binder对象

```cpp
if (ref->node->proc == target_proc) {
        if (fp->type == BINDER_TYPE_HANDLE)
            fp->type = BINDER_TYPE_BINDER;
        else
            fp->type = BINDER_TYPE_WEAK_BINDER;
        fp->binder = ref->node->ptr;
        fp->cookie = ref->node->cookie;
        binder_inc_node(ref->node, fp->type == BINDER_TYPE_BINDER, 0, NULL);
}
```

如果找到的索引->节点->proc不是需要的target_proc对象，那么在target_proc上查找或者为节点创建索引对象

```cpp
new_ref = binder_get_ref_for_node(target_proc, ref->node);
        if (new_ref == NULL) {
            return_error = BR_FAILED_REPLY;
            goto err_binder_get_ref_for_node_failed;
        }
fp->handle = new_ref->desc;
binder_inc_ref(new_ref, fp->type == BINDER_TYPE_HANDLE, NULL);
```

第三，如果发送端是文件对象（BINDER_TYPE_FD），发送端要与接收端共享一个打开的文件，这个处理就简单的多了。驱动先根据发送端的文件描述符(fd)得到struct file对象，在根据struct file对象为目标进程分配新的文件描述符。

```cpp
file = fget(fp->handle);
target_fd = task_get_unused_fd_flags(target_proc, O_CLOEXEC);
task_fd_install(target_proc, target_fd, file);
```

所以的binder_flat_object都处理好了，下面就准备唤醒目标进程/线程了，但在此之前，还要设置本次传输是否需要回应（REPLY）：

```cpp
if (reply) {
        binder_pop_transaction(target_thread, in_reply_to);
} else if (!(t->flags & TF_ONE_WAY)) {                    //同步传输， 需要reply，在这里设置reply需要的参数
        t->need_reply = 1;
        t->from_parent = thread->transaction_stack;  //设置用于REPLY命令
        thread->transaction_stack = t;                          //设置用于REPLY命令
} else {                                                                                  //异步传输，不需要reply
        if (target_node->has_async_transaction) {
            target_list = &target_node->async_todo;
            target_wait = NULL;
        } else
            target_node->has_async_transaction = 1;
}
```

下面就是唤醒目标进程/线程了：

```cpp
t->work.type = BINDER_WORK_TRANSACTION;
list_add_tail(&t->work.entry, target_list);
tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
list_add_tail(&tcomplete->entry, &thread->todo);
if (target_wait)
    wake_up_interruptible(target_wait);
```

BC_REPLY处理流程

以上就是BC_TRANSACTION的处理流程。BC_REPLY的处理流程跟BC_TRANSACTION类似，只不过REPLY是在传输同步传输时接收端反馈的数据，原来的接收端变成了发送端，原来的发送端变成了接收端，因此在获取target_proc，target_thread，binder_transaction时不同而已，下面我们来看一下：

首先得到原来的binder_transaction对象：

```cpp
in_reply_to = thread->transaction_stack;
```

然后从in_reply_to里取得需要的数据：binder_transaction，target_thread。找到target_thread，也就找到了target_proc对象了。

```cpp
thread->transaction_stack = in_reply_to->to_parent;     //to_parent是在binder_thread_read里设置的
target_thread = in_reply_to->from;
target_proc = target_thread->proc;
```

第二点不同的是，在唤醒目标进程/线程之前，需要通过binder_pop_transaction函数来释放原来的binder_transaction对象：

```cpp
if (target_thread) {
        target_thread->transaction_stack = target_thread->transaction_stack->from_parent;
        t->from = NULL;
}
t->need_reply = 0;
if (t->buffer)
    t->buffer->transaction = NULL;
kfree(t);
```

BC_REGISTER_LOOPER

BC_ENTER_LOOPER

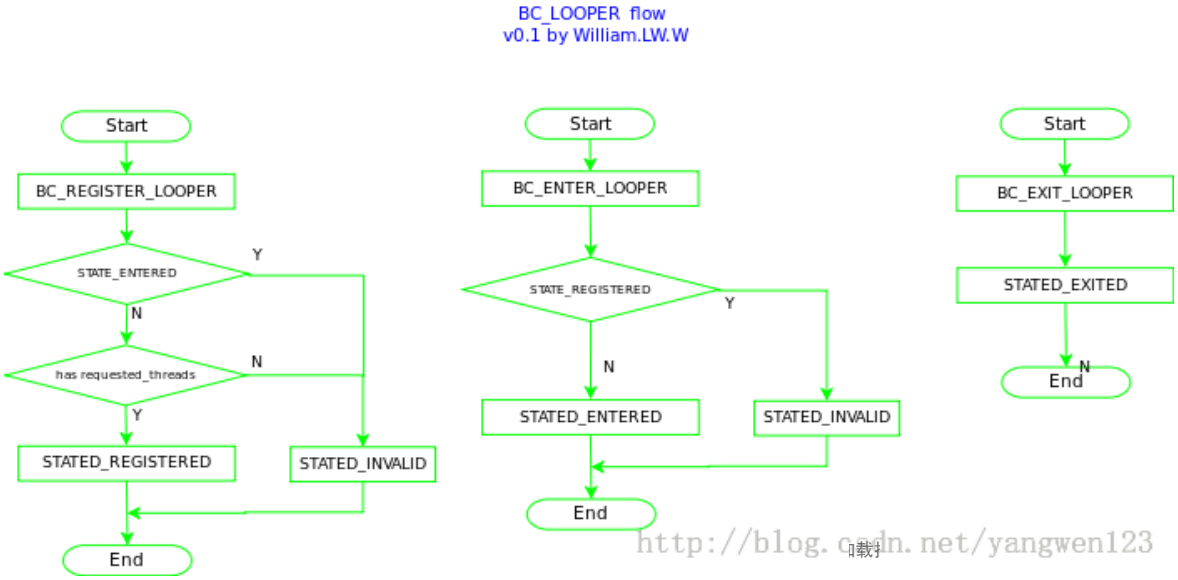
BC_EXIT_LOOPER

命令格式是：CMD

这三个命令用来设置binder looper的状态，binder looper一共有六种状态，而这三个命令主要用来设置其中的三种，如果出现错误则将状态设置为INVALID。

```cpp
enum {
    BINDER_LOOPER_STATE_REGISTERED  = 0x01,
    BINDER_LOOPER_STATE_ENTERED     = 0x02,
    BINDER_LOOPER_STATE_EXITED      = 0x04,
    BINDER_LOOPER_STATE_INVALID     = 0x08,
    BINDER_LOOPER_STATE_WAITING     = 0x10,
    BINDER_LOOPER_STATE_NEED_RETURN = 0x20
};
```

下面的图描述了不同命令时LOOPER状态的变化。从图中可以看出BINDER_LOOPER_STATE_REGISTERED和BINDER_LOOPER_STATE_ENTERED是互斥的。



BC_REQUEST_DEATH_NOTIFICATION

BC_CLEAR_DEATH_NOTIFICATION

命令格式： CMD | target_ptr | cookie

这两个命令通知目标进程/线程执行Request或者Clear DEATH_NOTIFICATION命令。首先本地进程/线程需要通知目标进程/线程执行Request命令，只有进入里Death状态的目标进程/线程才可以Clear DEATH_NOTIFICATION。

按照协议，驱动首先从用户空间读取目标索引对象的target_ptr和cookie值，再根据读取的target_ptr值取得目标节点的binder_ref索引对象。

```cpp
get_user(target, (uint32_t __user *)ptr)
get_user(cookie, (void __user * __user *)ptr)
ref = binder_get_ref(proc, target);
```

BC_REQUEST_DEATH_NOTIFICATION

首先我们来看BC_REQUEST_DEATH_NOTIFICATION这个命令。一个binder目标对象只能执行一次death操作，对于一个已经进入了death状态的目标对象来说，驱动认为已经执行成功了直接返回。否则，驱动就要创建一个新的binder_ref_death对象，并将这个death对象赋值给ref索引对象，然后唤醒目标进程/线程执行命令

```cpp
death = kzalloc(sizeof(*death), GFP_KERNEL);
INIT_LIST_HEAD(&death->work.entry);
death->cookie = cookie;
ref->death = death;
if (ref->node->proc == NULL) {
     ref->death->work.type = BINDER_WORK_DEAD_BINDER;
     if (thread->looper & (BINDER_LOOPER_STATE_REGISTERED | BINDER_LOOPER_STATE_ENTERED)) {
         list_add_tail(&ref->death->work.entry, &thread->todo);
      } else {
         list_add_tail(&ref->death->work.entry, &proc->todo);
         wake_up_interruptible(&proc->wait);
     }
}
```

BC_CLEAR_DEATH_NOTIFICATION

相反，对于BC_CLEAR_DEATH_NOTIFICATION命令，目标对象必须已经执行了Request Death操作，即ref索引对象里的death不能为空，否者就没有必要执行这个命令了。这个命令相对简单，从ref对象里取得death对象，并将ref->death置空，然后唤醒目标进程/线程执行命令。

```cpp
death = ref->death;
```

```
ref->death = NULL;
if (list_empty(&death->work.entry)) {
        death->work.type = BINDER_WORK_CLEAR_DEATH_NOTIFICATION;
        if (thread->looper & (BINDER_LOOPER_STATE_REGISTERED | BINDER_LOOPER_STATE_ENTERED)) {
            list_add_tail(&death->work.entry, &thread->todo);
        } else {
            list_add_tail(&death->work.entry, &proc->todo);
            wake_up_interruptible(&proc->wait);
        }
}
```

BC_DEAD_BINDER_DONE

命令格式： CMD |cookie

这个命令也是关于binder_ref_death的，按理说应该与前两个放在一起，在这里单独拿出来是因为他们的参数不同。按照协议，首先从用户空间读取death的cookie数据，因为cookie数据唯一的标识了binder_ref_death对象，因此可以根据cookie数据在proc对象上查找到对应的binder_ref_death对象。

```
[cpp]
get_user(cookie, (void __user * __user *)ptr)
list_for_each_entry(w, &proc->delivered_death, entry) {
        struct binder_ref_death *tmp_death = container_of(w, struct binder_ref_death, work);
        if (tmp_death->cookie == cookie) {
            death = tmp_death;
            break;
        }
}
```

找到death对象后，将death->work对象从proc或thread的todo列表上删除。如果death的work.type被设置为BINDER_WORK_DEAD_BINDER_AND_CLEAR了，就将它修改为BINDER_WORK_CLEAR_DEATH_NOTIFICATION，并唤醒thread或proc的等待队列处理它。

```
[cpp]
if (death->work.type == BINDER_WORK_DEAD_BINDER_AND_CLEAR) {
        death->work.type = BINDER_WORK_CLEAR_DEATH_NOTIFICATION;
        if (thread->looper & (BINDER_LOOPER_STATE_REGISTERED | BINDER_LOOPER_STATE_ENTERED)) {
            list_add_tail(&death->work.entry, &thread->todo);
        } else {
            list_add_tail(&death->work.entry, &proc->todo);
            wake_up_interruptible(&proc->wait);
        }
}
```

binder_thread_read

现在开始介绍binder_thread_read这个函数，前面提到在binder_ioctl的BINDER_WRITE_READ命令里，如果参数里的binder_write_read->read_size>0，那么就会调用binder_thread_read读取数据。简单的说，binder_thread_read就是处理挂在进程或者线程上的todo list，也就是binder_work。

前处理

binder_thread_read也有一个comsumed参数，由于记录本次处理过程中实际消耗了多少数据，显然对于一次数据处理，第一次进入此函数时，*consumed的数据为0，如果本次没有处理完成，用户程序还可以根据*consumed继续处理。Android binder的实现，对于一次处理流程，我们可以称之为事务，事务起始的时候都返回一个没有参数的BR_NOOP命令。

```
[cpp]
if (*consumed == 0) {
        if (put_user(BR_NOOP, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
}
```

那么对read函数来说是怎么区分处理proc->todo还是thread->todo呢？很简单，就是根据thread->transaction_stack和thread->todo，如果都是空就是proc->todo，否则就是thread->todo。

```cpp
wait_for_proc_work = thread->transaction_stack == NULL && list_empty(&thread->todo);
```

binder_thread_read有两种模式，一种是阻塞型，一种是非阻塞型，对于非阻塞型读操作，如果没有数据（也就是位于prco->todo或thread->todo上的binder_work），binder_thread_read就直接返回EAGAIN错误。而对于阻塞型读操作，程序就进入睡眠状态，等待有可操作的binder_work。

```cpp
if (wait_for_proc_work) {
        if (!(thread->looper & (BINDER_LOOPER_STATE_REGISTERED | BINDER_LOOPER_STATE_ENTERED))) {
            wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
        }
        binder_set_nice(proc->default_priority);
        if (non_block) {
            if (!binder_has_proc_work(proc, thread))
                ret = -EAGAIN;
        } else
            ret = wait_event_interruptible_exclusive(proc->wait, binder_has_proc_work(proc, thread));
} else {
        if (non_block) {
            if (!binder_has_thread_work(thread))
                ret = -EAGAIN;
        } else
            ret = wait_event_interruptible(thread->wait, binder_has_thread_work(thread));
}
```

这里提一下thread->looper的状态，需要注意的是，第一，在处理proc_work的时候，只有状态必须是BINDER_LOOPER_STATE_REGISTERED或者是BINDER_LOOPER_STATE_ENTERED；第二，不管处理的是thread->todo还是proc->todo，如果没有待处理的binder_work，那么状态被设置为BINDER_LOOPER_STATE_WAITING，否则就清除BINDER_LOOPER_STATE_WAITING标识。

对于待处理的binder_work，分为六种类型，分别是：

```cpp
struct binder_work {
    struct list_head entry;
    enum {
        BINDER_WORK_TRANSACTION = 1,
        BINDER_WORK_TRANSACTION_COMPLETE,
        BINDER_WORK_NODE,
        BINDER_WORK_DEAD_BINDER,
        BINDER_WORK_DEAD_BINDER_AND_CLEAR,
        BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
    } type;
};
```

看起来binder_work很简单，就只是含有type数据而已，entry只是用于链表处理的一个list内置对象，那么binder_work是怎么处理的呢？处理流程是这样，程序从thread->todo或者proc->todo上取得binder_work对象。根据BinderDriverReturnProtocol向用户空间发送BinderReturn命令（BR_），对于BR_TRANSACTION或者是BR_REPLY需要binder_transaction_data参数时，要知道binder_work是位于binder_transaction里的，所以根据binder_work就可以找到binder_transaction了。有了binder_transaction对象就可以找到所有需要的数据了，比如target_node, from_thread等，进而根据这些数据构造binder_transaction_data对象发送给用户空间应用程序。

查询binder_work

程序首先读取binder_work对象，程序的实现上是优先处理thread->todo的，只有函数进入的时候设置了wait_for_proc_work标识，才处理proc->todo。当还没有处理任何命令（仅仅发送了BR_NOOP命令给用户空间），并且thread->looper没有设置

BINDER_LOOPER_STATE_NEED_RETURN时，程序讲进入下一次的睡眠等待中，否则，程序不进入下一次睡眠而进入返回流程。

```cpp
if (!list_empty(&thread->todo))
        w = list_first_entry(&thread->todo, struct binder_work, entry);
else if (!list_empty(&proc->todo) && wait_for_proc_work)
        w = list_first_entry(&proc->todo, struct binder_work, entry);
else {
        if (ptr - buffer == 4 && !(thread->looper & BINDER_LOOPER_STATE_NEED_RETURN)) /* no data added */
                goto retry;
         break;
}
```

程序会循环的处理每一个连接在线程/进程上的binder_work对象，直到没有足够的空间保存binder_transaction_data时候才进入返回流程。

binder_work的处理

得到binder_work对象后，根据binder_work->type进入不同的处理过程：

1，如果是BINDER_WORK_TRANSACTION，则得到bnider_transaction对象

```cpp
t = container_of(w, struct binder_transaction, work);
```

2，如果是BINDER_WORK_TRANSACTION_COMPLETE，则发送BR_TRANSACTION_COMPLETE命令到用户空间通知程序TRANSACTION处理完成。并将work从链表上删除，释放为work分配的内存

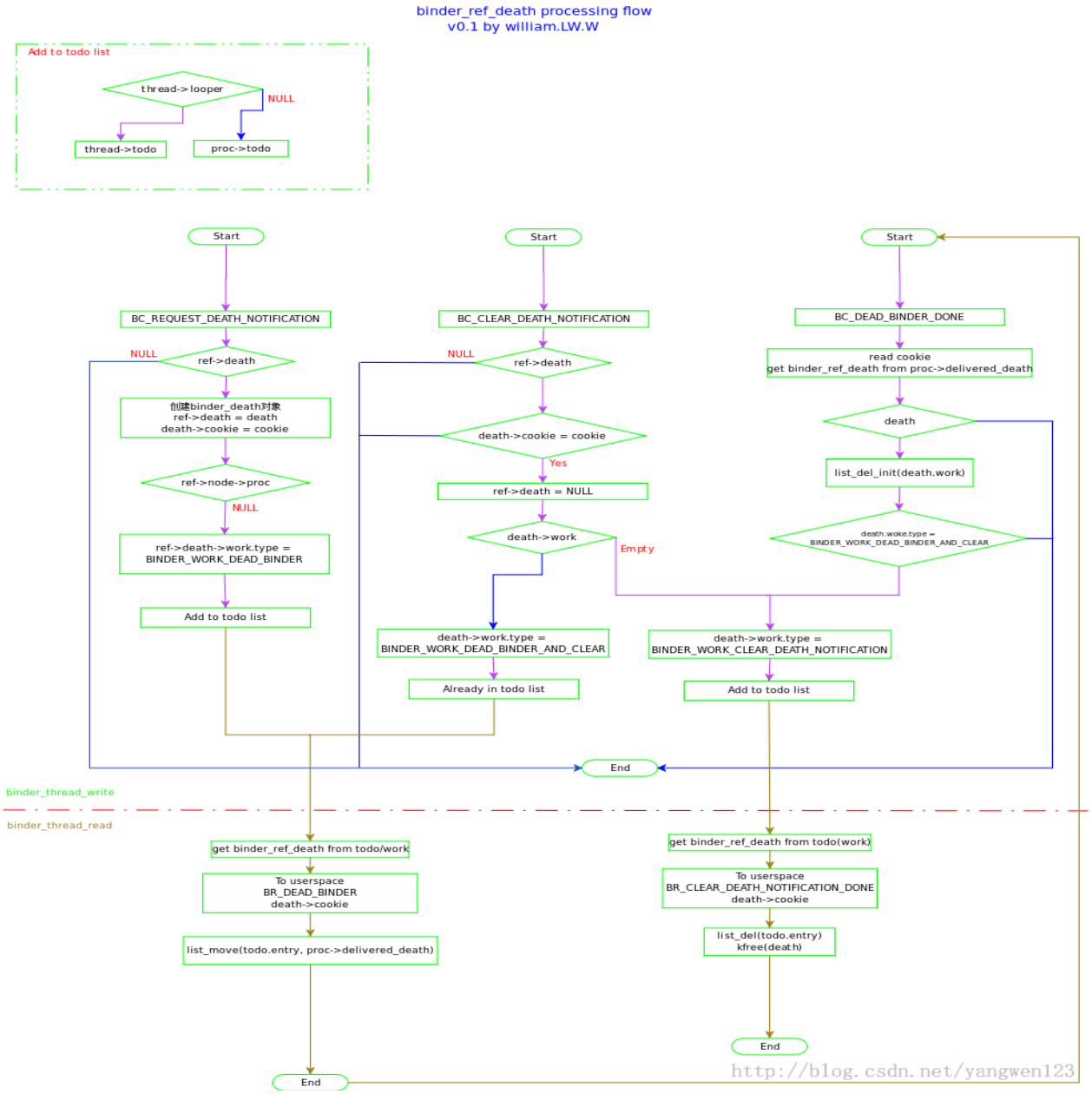
```cpp
cmd = BR_TRANSACTION_COMPLETE;
 if (put_user(cmd, (uint32_t __user *)ptr))
        return -EFAULT;
list_del(&w->entry);
kfree(w);
```

3，如果是BINDER_WORK_NODE，根据binder_work得到binder_node节点，然后根据节点的strond或者weak类型，增加/获取、减少/释放节点的索引，参数就是node->ptr和node->cookie数据。

```cpp
struct binder_node *node = container_of(w, struct binder_node, work);      //得到binder_node
int strong = node->internal_strong_refs || node->local_strong_refs;           //得到当前的strong
int weak = !hlist_empty(&node->refs) || node->local_weak_refs || strong;  //得到当前的weak，为什么要或上strong?
//构造命令BR_命令
if (weak && !node->has_weak_ref) {      //如果是wake但还没有设置has_weak_ref
        cmd = BR_INCREFS;
        cmd_name = "BR_INCREFS";
        node->has_weak_ref = 1;
        node->pending_weak_ref = 1;
        node->local_weak_refs++;
} else if (strong && !node->has_strong_ref) {      //如果是strong，但还没有设置has_strong_ref
        cmd = BR_ACQUIRE;
        cmd_name = "BR_ACQUIRE";
        node->has_strong_ref = 1;
        node->pending_strong_ref = 1;
        node->local_strong_refs++;
} else if (!strong && node->has_strong_ref) {       //如果没有strong，但是设置了has_strong_ref
        cmd = BR_RELEASE;
        cmd_name = "BR_RELEASE";
        node->has_strong_ref = 0;
} else if (!weak && node->has_weak_ref) {         //如果没有weak，但是设置了has_weak_ref
        cmd = BR_DECREFS;
        cmd_name = "BR_DECREFS";
```

```
        node->has_weak_ref = 0;
}
//如果有需要处理的数据
put_user(cmd, (uint32_t __user *)ptr)
put_user(node->ptr, (void * __user *)ptr)
put_user(node->cookie, (void * __user *)ptr)
```

4，如果是BINDER_WORK_DEAD_BINDER，BINDER_WORK_DEAD_BINDER_AND_CLEAR或者BINDER_WORK_CLEAR_DEATH_NOTIFICATION有关DEAD_BINDER的操作，我们在前面的biner_thread_write里已经介绍了相关的BC命令。由于read函数相对简单一些，不需要在详细描述了，请参考下面的binder_ref_death的处理流程（包括binder_thread_write部分）：



返回binder_transaction_data

如果是关于BINDER_WORK_TRANSACTION类型的数据处理，则需要像用户空间返回binder_transaction_data数据。前面提到已经得到binder_transaction对象了，如果是BC_TRANSACTION命令发过来的数据，则从binder_transaction里取得target_node数据，填充binder_transaction_data：

```
[cpp]
struct binder_node *target_node = t->buffer->target_node;
tr.target.ptr = target_node->ptr;
tr.cookie =  target_node->cookie;
```

如果是BC_REPLY发过来的命令，则发回的BR命令就是BR_REPLY，并且target_prt和cookie都是空：

```cpp
tr.target.ptr = NULL;
tr.cookie = NULL;
cmd = BR_REPLY;
```

驱动继续根据binder_transaction对象填充返回给用户空间的binder_transaction_data对象数据，然后发送给用户空间程序：

```cpp
put_user(cmd, (uint32_t __user *)ptr)
copy_to_user(ptr, &tr, sizeof(tr))
```

然后将处理完的binder_transaction从链表上移出去，并润许用户释放里面的buffer内存。

```cpp
list_del(&t->work.entry);
t->buffer->allow_user_free = 1;
```

如果是同步BR_TRANSACTION操作，就将当前的thread对象赋值给binder_transaction对象发挥给发送端进程/线程进行下一步的同步处理，否则就释放这个binder_transaction对象。

```cpp
if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
            t->to_parent = thread->transaction_stack;
            t->to_thread = thread;
            thread->transaction_stack = t;
} else {
            t->buffer->transaction = NULL;
            kfree(t);
}
```

到这里我们就详细的介绍了Android的用户接口实现，包括了最重要的binder IOCTL BINDER_WRITE_READ及其内部实现机制。从这里我们就可以详细的了解Android驱动程序是如果发送和接受数据的，从而实现Binder的RPC功能。