<https://blog.csdn.net/freshui/article/details/54926111>

<https://blog.csdn.net/qq_33859911/article/details/88551527>

frameworks/av/media/libmedia

Binder驱动暴露给应用层的ioctl指令

|  |  |  |
| --- | --- | --- |
| BINDER\_WRITE\_READ | 读写数据使用 | struct binder\_write\_read |
| BINDER\_SET\_IDLE\_TIMEOUT |  |  |
| BINDER\_SET\_MAX\_THREADS | 该命令告知Binder驱动接收方（通常是Server端）线程池中最大的线程数 |  |
| BINDER\_SET\_IDLE\_PRIORITY |  |  |
| BINDER\_SET\_CONTEXT\_MGR | 将当前线程注册为service manager |  |
| BINDER\_THREAD\_EXIT |  |  |
| BINDER\_VERSION |  | struct binder\_version |
| BINDER\_GET\_NODE\_DEBUG\_INFO |  | struct binder\_node\_debug\_info |

struct binder\_write\_read {

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

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

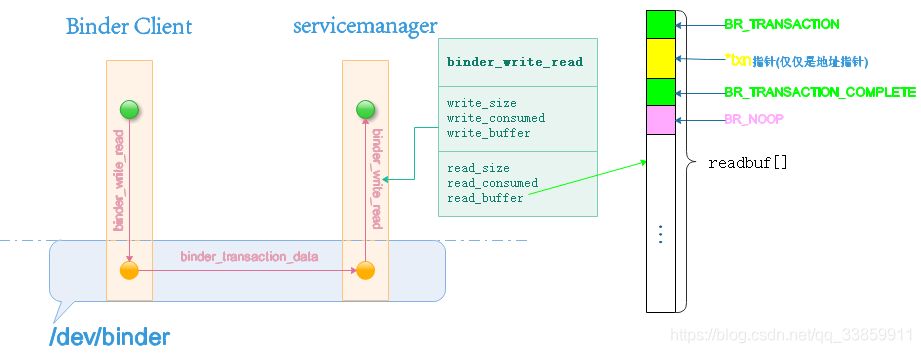
binder\_uintptr\_t write\_buffer;

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

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

binder\_uintptr\_t read\_buffer;

};



读写操作通过一条指令和一个结构体就可以实现，当然读写指令的内部会分成许多小的指令

下面是写指令的总结

|  |  |  |
| --- | --- | --- |
| BC\_TRANSACTION | Client向server发送请求数据 | struct binder\_transaction\_data |
| BC\_REPLY | Server向client发送回复数据 | struct binder\_transaction\_data |
| BC\_ACQUIRE\_RESULT | 暂未实现 |  |
| BC\_FREE\_BUFFER | 释放一块映射的内存。Binder接收方通过mmap()映射一块较大的内存空间，Binder驱动基于这片内存采用最佳匹配算法实现接收数据缓存的动态分配和释放，满足并发请求对接收缓存区的需求。应用程序处理完这片数据后必须尽快使用该命令释放缓存区，否则会因为缓存区耗尽而无法接收新数据。 | binder\_uintptr\_t |
| BC\_INCREFS  BC\_ACQUIRE  BC\_RELEASE  BC\_DECREFS | 这组命令增加或减少Binder的引用计数，用以实现强指针或弱指针的功能 |  |
| BC\_INCREFS\_DONE  BC\_ACQUIRE\_DONE | 第一次增加Binder实体引用计数时，驱动向Binder实体所在的进程发送BR\_INCREFS， BR\_ACQUIRE消息；Binder实体所在的进程处理完毕回馈BC\_INCREFS\_DONE，BC\_ACQUIRE\_DONE | struct binder\_ptr\_cookie |
| BC\_REGISTER\_LOOPER  BC\_ENTER\_LOOPER  BC\_EXIT\_LOOPER | 这组命令同BINDER\_SET\_MAX\_THREADS一道实现Binder驱动对接收方线程池管理。BC\_REGISTER\_LOOPER通知驱动线程池中一个线程已经创建了；BC\_ENTER\_LOOPER通知驱动该线程已经进入主循环，可以接收数据；BC\_EXIT\_LOOPER通知驱动该线程退出主循环，不再接收数据。 |  |
| BC\_REQUEST\_DEATH\_NOTIFICATION | 获得Binder引用的进程通过该命令要求驱动在Binder实体销毁得到通知。虽说强指针可以确保只要有引用就不会销毁实体，但这毕竟是个跨进程的引用，谁也无法保证实体由于所在的Server关闭Binder驱动或异常退出而消失，引用者能做的是要求Server在此刻给出通知 | struct binder\_handle\_cookie |
| BC\_DEAD\_BINDER\_DONE | 收到实体死亡通知书的进程在删除引用后用本命令告知驱动 | binder\_uintptr\_t |
| BC\_CLEAR\_DEATH\_NOTIFICATION |  | struct binder\_handle\_cookie |
| BC\_TRANSACTION\_SG |  | struct binder\_transaction\_data\_sg |
| BC\_REPLY\_SG |  | struct binder\_transaction\_data\_sg |

读指令（即返回指令的总结如下）

|  |  |  |
| --- | --- | --- |
| BR\_ERROR  BR\_OK | 操作完成 |  |
| BR\_TRANSACTION  BR\_REPLY | 这两条消息分别对应发送方的BC\_TRANSACTION和BC\_REPLY，表示当前接收的数据是请求还是回复 | struct binder\_transaction\_data |
| BR\_ACQUIRE\_RESULT  BR\_ATTEMPT\_ACQUIRE  BR\_FINISHED | 尚未实现  Msm4.4代码 |  |
| BR\_INCREFS  BR\_ACQUIRE  BR\_RELEASE  BR\_DECREFS | 这一组消息用于管理强/弱指针的引用计数。只有提供Binder实体的进程才能收到这组消息 | struct binder\_ptr\_cookie |
| BR\_DEAD\_REPLY | 交互过程中如果发现对方进程或线程已经死亡则返回该消息 |  |
| BR\_TRANSACTION\_COMPLETE | 发送方通过BC\_TRANSACTION或BC\_REPLY发送完一个数据包后，都能收到该消息做为成功发送的反馈。这和BR\_REPLY不一样，是驱动告知发送方已经发送成功，而不是Server端返回请求数据。所以不管同步还是异步交互接收方都能获得本消息。 |  |
| BR\_NOOP | 操作完成 |  |
| BR\_SPAWN\_LOOPER | 该消息用于接收方线程池管理。当驱动发现接收方所有线程都处于忙碌状态且线程池里的线程总数没有超过BINDER\_SET\_MAX\_THREADS设置的最大线程数时，向接收方发送该命令要求创建更多线程以备接收数据 |  |
| BR\_DEAD\_BINDER  BR\_CLEAR\_DEATH\_NOTIFICATION\_DONE | 向获得Binder引用的进程发送Binder实体死亡通知书；收到死亡通知书的进程接下来会返回BC\_DEAD\_BINDER\_DONE做确认 | binder\_uintptr\_t |
| BR\_FAILED\_REPLY | 如果发送非法引用号则返回该消息 |  |

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;

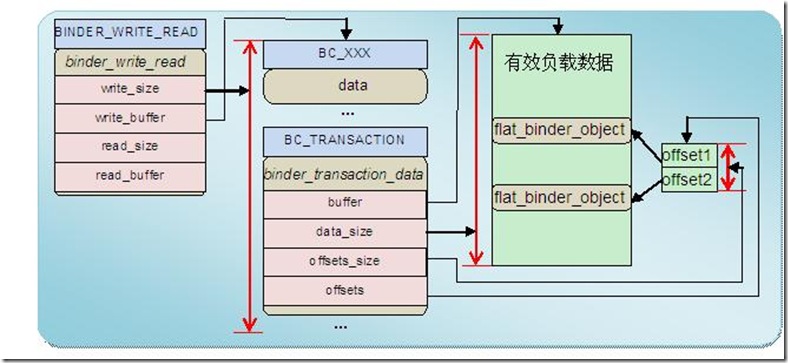
} ptr;

\_\_u8 buf[8];

} data;

};

|  |  |
| --- | --- |
| union {  /\* target descriptor of command transaction \*/  \_\_u32 handle;  /\* target descriptor of return transaction \*/  binder\_uintptr\_t ptr;  } target; | 对于发送数据包的一方，该成员指明发送目的地。由于目的是在远端，所以这里填入的是对Binder实体的引用，存放在target.handle中。如前述，Binder的引用在代码中也叫句柄（handle）。  当数据包到达接收方时，驱动已将该成员修改成Binder实体，即指向Binder对象内存的指针，使用target.ptr来获得。该指针是接收方在将Binder实体传输给其它进程时提交给驱动的，驱动程序能够自动将发送方填入的引用转换成接收方Binder对象的指针，故接收方可以直接将其当做对象指针来使用（通常是将其reinterpret\_cast成相应类） |
| binder\_uintptr\_t cookie | 发送方忽略该成员；接收方收到数据包时，该成员存放的是创建Binder实体时由该接收方自定义的任意数值，做为与Binder指针相关的额外信息存放在驱动中。驱动基本上不关心该成员。 |
| \_\_u32 code | 该成员存放收发双方约定的命令码，驱动完全不关心该成员的内容。通常是Server端定义的公共接口函数的编号 |
| \_\_u32 flags | 与交互相关的标志位，其中最重要的是TF\_ONE\_WAY位。如果该位置上表明这次交互是异步的，Server端不会返回任何数据。驱动利用该位来决定是否构建与返回有关的数据结构。另外一位TF\_ACCEPT\_FDS是出于安全考虑，如果发起请求的一方不希望在收到的回复中接收文件形式的Binder可以将该位置上。因为收到一个文件形式的Binder会自动为数据接收方打开一个文件，使用该位可以防止打开文件过多。 |
| pid\_t sender\_pid;  uid\_t sender\_euid; | 该成员存放发送方的进程ID和用户ID，由驱动负责填入，接收方可以读取该成员获知发送方的身份。 |
| binder\_size\_t data\_size;  binder\_size\_t offsets\_size; | 该成员表示data.buffer指向的缓冲区存放的数据长度。发送数据时由发送方填入，表示即将发送的数据长度；在接收方用来告知接收到数据的长度。  驱动一般情况下不关心data.buffer里存放什么数据，但如果有Binder在其中传输则需要将其相对data.buffer的偏移位置指出来让驱动知道。有可能存在多个Binder同时在数据中传递，所以须用数组表示所有偏移位置。本成员表示该数组的大小。 |
| 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; | data.bufer存放要发送或接收到的数据；data.offsets指向Binder偏移位置数组，该数组可以位于data.buffer中，也可以在另外的内存空间中，并无限制。buf[8]是为了无论保证32位还是64位平台，成员data的大小都是8个字节。 |



/\*

\* This is the flattened representation of a Binder object for transfer

\* between processes. The 'offsets' supplied as part of a binder transaction

\* contains offsets into the data where these structures occur. The Binder

\* driver takes care of re-writing the structure type and data as it moves

\* between processes.

\*/

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;

};

Flat\_binder\_object各成员变量解析

|  |  |
| --- | --- |
| struct binder\_object\_header hdr; | 表明该Binder的类型，包括以下几种：  BINDER\_TYPE\_BINDER：表示传递的是Binder实体，并且指向该实体的引用都是强类型；  BINDER\_TYPE\_WEAK\_BINDER：表示传递的是Binder实体，并且指向该实体的引用都是弱类型；  BINDER\_TYPE\_HANDLE：表示传递的是Binder强类型的引用  BINDER\_TYPE\_WEAK\_HANDLE：表示传递的是Binder弱类型的引用  BINDER\_TYPE\_FD：表示传递的是文件形式的Binder |
| \_\_u32 flags | 该域只对第一次传递Binder实体时有效，因为此刻驱动需要在内核中创建相应的实体节点，有些参数需要从该域取出：  第0-7位：代码中用FLAT\_BINDER\_FLAG\_PRIORITY\_MASK取得，表示处理本实体请求数据包的线程的最低优先级。当一个应用程序提供多个实体时，可以通过该参数调整分配给各个实体的处理能力。  第8位：代码中用FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS取得，置1表示该实体可以接收其它进程发过来的文件形式的Binder。由于接收文件形式的Binder会在本进程中自动打开文件，有些Server可以用该标志禁止该功能，以防打开过多文件。 |
| union {  binder\_uintptr\_t binder; /\* local object \*/  \_\_u32 handle; /\* remote object \*/  }; | 当传递的是Binder实体时使用binder域，指向Binder实体在应用程序中的地址。  当传递的是Binder引用时使用handle域，存放Binder在进程中的引用号。 |
| binder\_uintptr\_t cookie | 该域只对Binder实体有效，存放与该Binder有关的附加信息 |

内核当中代表binder实体的结构体叫做 struct binder\_node，结构体如下

struct binder\_node {

int debug\_id;

spinlock\_t lock;

struct binder\_work work; 当本节点引用计数发生改变，需要通知所属进程时，通过该成员挂入所属进程的to-do队列里，唤醒所属进程执行Binder实体引用计数的修改。

union {

struct rb\_node rb\_node;

struct hlist\_node dead\_node;

};每个进程都维护一棵红黑树，以Binder实体在用户空间的指针，即本结构的ptr成员为索引存放该进程所有的Binder实体。这样驱动可以根据Binder实体在用户空间的指针很快找到其位于内核的节点。rb\_node用于将本节点链入该红黑树中。

销毁节点时须将rb\_node从红黑树中摘除，但如果本节点还有引用没有切断，就用dead\_node将节点隔离到另一个链表中，直到通知所有进程切断与该节点的引用后，该节点才可能被销毁。

struct binder\_proc \*proc;本成员指向节点所属的进程，即提供该节点的进程。

struct hlist\_head refs;本成员是队列头，所有指向本节点的引用都链接在该队列里。这些引用可能隶属于不同的进程。通过该队列可以遍历指向该节点的所有引用。

int internal\_strong\_refs;用以实现强指针的计数器：产生一个指向本节点的强引用该计数就会加1。

int local\_weak\_refs;驱动为传输中的Binder设置的弱引用计数。如果一个Binder打包在数据包中从一个进程发送到另一个进程，驱动会为该Binder增加引用计数，直到接收进程通过BC\_FREE\_BUFFER通知驱动释放该数据包的数据区为止。

int local\_strong\_refs;驱动为传输中的Binder设置的强引用计数。同上。

int tmp\_refs;

binder\_uintptr\_t ptr;指向用户空间Binder实体的指针，来自于flat\_binder\_object的binder成员

binder\_uintptr\_t cookie;指向用户空间的附加指针，来自于flat\_binder\_object的cookie成员

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;

};这一组标志用于控制驱动与Binder实体所在进程交互式修改引用计数

struct {

/\*

\* invariant after initialization

\*/

u8 sched\_policy:2;

u8 inherit\_rt:1;

u8 accept\_fds:1;表明节点是否同意接受文件方式的Binder，来自flat\_binder\_object中flags成员的FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS位。由于接收文件Binder会为进程自动打开一个文件，占用有限的文件描述符，节点可以设置该位拒绝这种行为。

u8 min\_priority;设置处理Binder请求的线程的最低优先级。发送线程将数据提交给接收线程处理时，驱动会将发送线程的优先级也赋予接收线程，使得数据即使跨了进程也能以同样优先级得到处理。不过如果发送线程优先级过低，接收线程将以预设的最小值运行。

该域的值来自于flat\_binder\_object中flags成员。

};

bool has\_async\_transaction;该成员表明该节点在to-do队列中有异步交互尚未完成。驱动将所有发送往接收端的数据包暂存在接收进程或线程开辟的to-do队列里。对于异步交互，驱动做了适当流控：如果to-do队列里有异步交互尚待处理则该成员置1，这将导致新到的异步交互存放在本结构成员 – asynch\_todo队列中，而不直接送到to-do队列里。目的是为同步交互让路，避免长时间阻塞发送端。

struct list\_head async\_todo;

};

Binder的引用在驱动当中使用结构体

struct binder\_ref 来表示

/\*\*

\* struct binder\_ref - struct to track references on nodes

\* @data: binder\_ref\_data containing id, handle, and current refcounts

\* @rb\_node\_desc: node for lookup by @data.desc in proc's rb\_tree

\* @rb\_node\_node: node for lookup by @node in proc's rb\_tree

\* @node\_entry: list entry for node->refs list in target node

\* (protected by @node->lock)

\* @proc: binder\_proc containing ref

\* @node: binder\_node of target node. When cleaning up a

\* ref for deletion in binder\_cleanup\_ref, a non-NULL

\* @node indicates the node must be freed

\* @death: pointer to death notification (ref\_death) if requested

\* (protected by @node->lock)

\*

\* Structure to track references from procA to target node (on procB). This

\* structure is unsafe to access without holding @proc->outer\_lock.

\*/

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 binder\_ref\_data {

int debug\_id;

uint32\_t desc;本结构的引用号

int strong;强引用计数

int weak;弱引用计数

};

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

struct rb\_node rb\_node\_desc;每个进程有一棵红黑树，进程所有引用以引用号（即本结构的desc域）为索引添入该树中。本成员用做链接到该树的一个节点。

struct rb\_node rb\_node\_node;每个进程又有一棵红黑树，进程所有引用以节点实体在驱动中的内存地址（即本结构的node域）为所引添入该树中。本成员用做链接到该树的一个节点。

struct hlist\_node node\_entry;该域将本引用做为节点链入所指向的Binder实体结构binder\_node中的refs队列

struct binder\_proc \*proc;本引用所属的进程

struct binder\_node \*node;本引用所指向的节点（Binder实体）

struct binder\_ref\_death \*death;应用程序向驱动发送BC\_REQUEST\_DEATH\_NOTIFICATION或BC\_CLEAR\_DEATH\_NOTIFICATION命令从而当Binder实体销毁时能够收到来自驱动的提醒。该域不为空表明用户订阅了对应实体销毁的‘噩耗’

};

就象一个对象有很多指针一样，同一个Binder实体可能有很多引用，不同的是这些引用可能分布在不同的进程中。和实体一样，每个进程使用红黑树存放所有正在使用的引用。不同的是Binder的引用可以通过两个键值索引：

· 对应实体在内核中的地址。注意这里指的是驱动创建于内核中的binder\_node结构的地址，而不是Binder实体在用户进程中的地址。实体在内核中的地址是唯一的，用做索引不会产生二义性；但实体可能来自不同用户进程，而实体在不同用户进程中的地址可能重合，不能用来做索引。驱动利用该红黑树在一个进程中快速查找某个Binder实体所对应的引用（一个实体在一个进程中只建立一个引用）。

· 引用号。引用号是驱动为引用分配的一个32位标识，在一个进程内是唯一的，而在不同进程中可能会有同样的值，这和进程的打开文件号很类似。引用号将返回给应用程序，可以看作Binder引用在用户进程中的句柄。除了0号引用在所有进程里都固定保留给SMgr，其它值由驱动动态分配。向Binder发送数据包时，应用程序将引用号填入binder\_transaction\_data结构的target.handle域中表明该数据包的目的Binder。驱动根据该引用号在红黑树中找到引用的binder\_ref结构，进而通过其node域知道目标Binder实体所在的进程及其它相关信息，实现数据包的路由。

