


Android Binder通信机制

2013-06-14 19:14

1751人阅读

[评论\(2\)](#)[收藏](#)[举报](#) 分类： **【Android Binder通信】 (8)** ▼版权声明：
本

本文为博主原创文章，未经博主允许不得转载。

[目录\(?\)](#)[\[+\]](#)

Binder是Android系统进程间通信（IPC）方式之一。Linux已经拥有管道，system V IPC，socket等IPC手段，却还要倚赖Binder来实现进程间通信，说明Binder具有无可比拟的优势。深入了解Binder并将之与传统IPC做对比有助于我们深入领会进程间通信的实现和性能优化。本文将对Binder的设计细节做一个全面的阐述，首先通过介绍Binder通信模型和Binder通信协议了解Binder的设计需求；然后分别阐述Binder在系统不同部分的表述方式和起的作用；最后还会解释Binder在数据接收端的设计考虑，包括线程池管理，内存映射和等待队列管理等。通过本文对Binder的详细介绍以及与其它IPC通信方式的对比，读者将对Binder的优势和使用Binder作为Android主要IPC方式的原因有深入了解。应用程序虽然是以独立的进程来运行，但相互之间还是需要相互协作，彼此间又必须进行通信和数据共享，这就需要进程通信来完成。

Binder提供了以下功能：

- 1.用驱动程序来实现进程间通信；
- 2.通过共享内存来提供性能；
- 3.为进程请求分配线程池；
- 4.为对象引入了引用计数和跨进程对象引用映射；
- 5.进程间同步调用；

Binder操作类似于线程迁移，两个进程间通信看起来就像是一个进程进入另一个进程去执行代码，然后带着执行结果返回。Binder在用户空间为每一个进程维护着一个可用的线程池，用于处理到来的IPC以及执行进程的本地消息，Binder通信时同步的而不是异步的。

引言

基于Client-Server的通信方式广泛应用于从互联网和数据库访问到嵌入式手持设备内部通信等各个领域。智能手机平台特别是Android系统中，为了向应用开发者提供丰富多样的功能，这种通信方式更是无处不在，诸如媒体播放，视音频捕获，到各种让手机更智能的传感器（加速度，方位，温度，光亮度等）都由不同的Server负责管理，应用程序只需做为Client与这些Server建立连接便可以使用这些服务，花很少的时间和精力就能开发出令人眩目的功能。Client-Server方式的广泛采用对进程间通信（IPC）机制是一个挑战。目前linux支持的IPC包括传统的管道，System V IPC，即消息队列/共享内存/信号量，以及socket中只有socket支持Client-Server的通信方式。当然也可以在这些底层机制上架设一套协议来实现Client-Server通信，但这样增加了系统的复杂性，在手机这种条件复杂，资源稀缺的环境下可靠性也难以保证。

另一方面是传输性能。socket作为一款通用接口，其传输效率低，开销大，主要用在跨网络的进程间通信和本机上进程间的低速通信。消息队列和管道采用存储-转发方式，即数据先从发送方缓存区拷贝到内核开辟的缓存区中，然后再从内核缓存区拷贝到接收方缓存区，至少有两次拷贝过程。共享内存虽然无需拷贝，但控制复杂，难

以使用。

表 1 各种IPC方式数据拷贝次数

共享内存	0
Binder	1
Socket/管道/消息队列	2

还有一点是出于安全性考虑。Android作为一个开放式，拥有众多开发者的平台，应用程序的来源广泛，确保智能终端的安全是非常重要的。终端用户不希望从网上下载的程序在不知情的情况下偷窥隐私数据，连接无线网络，长期操作底层设备导致电池很快耗尽等等。传统IPC没有任何安全措施，完全依赖上层协议来确保。首先传统IPC的接收方无法获得对方进程可靠的UID/PID（用户ID/进程ID），从而无法鉴别对方身份。Android为每个安装好的应用程序分配了自己的UID，故进程的UID是鉴别进程身份的重要标志。使用传统IPC只能由用户在数据包里填入UID/PID，但这样不可靠，容易被恶意程序利用。可靠的身份标记只有由IPC机制本身在内核中添加。其次传统IPC访问接入点是开放的，无法建立私有通道。比如命名管道的名称，system V的键值，socket的ip地址或文件名都是开放的，只要知道这些接入点的程序都可以和对端建立连接，不管怎样都无法阻止恶意程序通过猜测接收方地址获得连接。

基于以上原因，Android需要建立一套新的IPC机制来满足系统对通信方式，传输性能和安全性的要求，这就是Binder。Binder基于 Client-Server通信模式，传输过程只需一次拷贝，为发送发添加UID/PID身份，既支持实名Binder也支持匿名Binder，安全性高。

面向对象的 Binder IPC

Binder使用Client-Server通信方式：一个进程作为Server提供诸如视频/音频解码，视频捕获，地址本查询，网络连接等服务；多个进程作为Client向Server发起服务请求，获得所需要的服务。要想实现Client-Server通信据必须实现以下两点：一是server 必须有确定的访问接入点或者说地址来接受Client的请求，并且Client可以通过某种途径获知Server的地址；二是制定Command- Reply协议来传输数据。例如在网络通信中Server的访问接入点就是Server主机的IP地址+端口号，传输协议为TCP协议。对Binder而言，Binder可以看成Server提供的实现某个特定服务的访问接入点， Client通过这个‘地址’向Server发送请求来使用该服务；对Client而言，Binder可以看成是通向Server的管道入口，要想和某个 Server通信首先必须建立这个管道并获得管道入口。

与其它IPC不同，Binder使用了面向对象的思想来描述作为访问接入点的Binder及其在Client中的入口：Binder是一个实体位于 Server中的对象，该对象提供了一套方法用以实现对服务的请求，就象类的成员函数。遍布于client中的入口可以看成指向这个binder对象的‘指针’，一旦获得了这个‘指针’就可以调用该方法访问server。在Client看来，通过Binder‘指针’调用其提供的方法和通过指针调用其它任何本地对象的方法并无区别，尽管前者的实体位于远端Server中，而后者实体位于本地内存中。‘指针’是C++的术语，而更通常的说法是引用，即Client通过Binder的引用访问Server。而软件领域另一个术语‘句柄’也可以用来表述Binder在Client中的存在方式。从通信的角度看，Client中的Binder也可以看作是Server Binder的‘代理’，在本地代表远端Server为Client提供服务。本文中会使用‘引用’或‘句柄’这个两广泛使用的术语。

面向对象思想的引入将进程间通信转化为通过对某个Binder对象的引用调用该方法，而其独特之处在于

Binder对象是一个可以跨进程引用的对象，它的实体位于一个进程中，而它的引用却遍布于系统的各个进程之中。最诱人的是，这个引用和java里引用一样既可以是强类型，也可以是弱类型，而且可以从一个进程传给其它进程，让大家都能访问同一Server，就象将一个对象或引用赋值给另一个引用一样。Binder模糊了进程边界，淡化了进程间通信过程，整个系统仿佛运行于同一个面向对象的程序之中。形形色色的Binder对象以及星罗棋布的引用仿佛粘接各个应用程序的胶水，这也是Binder 在英文里的原意。当然面向对象只是针对应用程序而言，对于Binder驱动和内核其它模块一样使用C语言实现，没有类和对象的概念。Binder驱动为面向对象的进程间通信提供底层支持。

Binder 通信模型

Binder框架定义了四个角色：Server，Client，ServiceManager（以后简称SMgr）以及驱动。其中 Server，Client，SMgr运行于用户空间，驱动运行于内核空间。这四个角色的关系和互联网类似：Server是服务器，Client是客户终端，SMgr是域名服务器（DNS），驱动是路由器。

Binder 驱动

和路由器一样，Binder驱动虽然默默无闻，却是通信的核心。尽管名叫‘驱动’，实际上和硬件设备没有任何关系，只是实现方式和设备驱动程序是一样的：它工作于内核态，提供open()，mmap()，poll()，ioctl()等标准文件操作，以字符驱动设备中的misc设备注册在设备目录 /dev下，用户通过/dev/binder访问该它。驱动负责进程之间Binder通信的建立，Binder在进程之间的传递，Binder引用计数管理，数据包在进程之间的传递和交互等一系列底层支持。驱动和应用程序之间定义了一套接口协议，主要功能由ioctl()接口实现，不提供 read()，write()接口，因为ioctl()灵活方便，且能够一次调用实现先写后读以满足同步交互，而不必分别调用write()和 read()。

ServiceManager与实名Binder

和DNS类似，ServiceManager的作用是将字符形式的Binder名字转化成Client中对该Binder的引用，使得Client能够通过Binder 名字获得对Server中Binder实体的引用。注册了名字的Binder叫实名Binder，就象每个网站除了有IP地址外都有自己的网址。Server创建了Binder实体，为其取一个字符形式，可读易记的名字，将这个Binder连同名字以数据包的形式通过Binder驱动发送给ServiceManager，通知ServiceManager注册一个名叫张三的Binder，它位于某个Server中。驱动为这个穿过进程边界的Binder创建位于内核中的实体节点以及ServiceManager对实体的引用，将名字及新建的引用传递给ServiceManager。ServiceManager收数据包后，从中取出名字和引用填入一张查找表中。

ServiceManager是一个进程，Server是另一个进程，Server向ServiceManager注册Binder必然会涉及进程间通信。当前实现的是进程间通信却又要用到进程间通信，这就好象蛋可以孵出鸡前提却是要找只鸡来孵蛋。Binder的实现比较巧妙：预先创造一只鸡来孵蛋。ServiceManager和其它进程同样采用Binder通信，ServiceManager是Server端，有自己的Binder实体，其它进程都是Client，需要通过这个 Binder的引用来实现Binder的注册，查询和获取。ServiceManager提供的Binder比较特殊，它没有名字也不需要注册，当一个进程使用 BINDER_SET_CONTEXT_MGR命令将自己注册成ServiceManager时Binder驱动会自动为它创建Binder实体。其次这个Binder的引用在所有Client中都固定为0而无须通过其它手段获得。也就是说，一个Server若要向ServiceManager注册自己Binder就必需通过0这个引用和ServiceManager的Binder通信。类比网络通信，0号引用就好比域名服务器的地址，你必须手工或动态配置好。要注意这里说的Client是相对ServiceManager而言的，一

个应用程序是个提供服务的Server，但对ServiceManager来说它仍然是个Client。

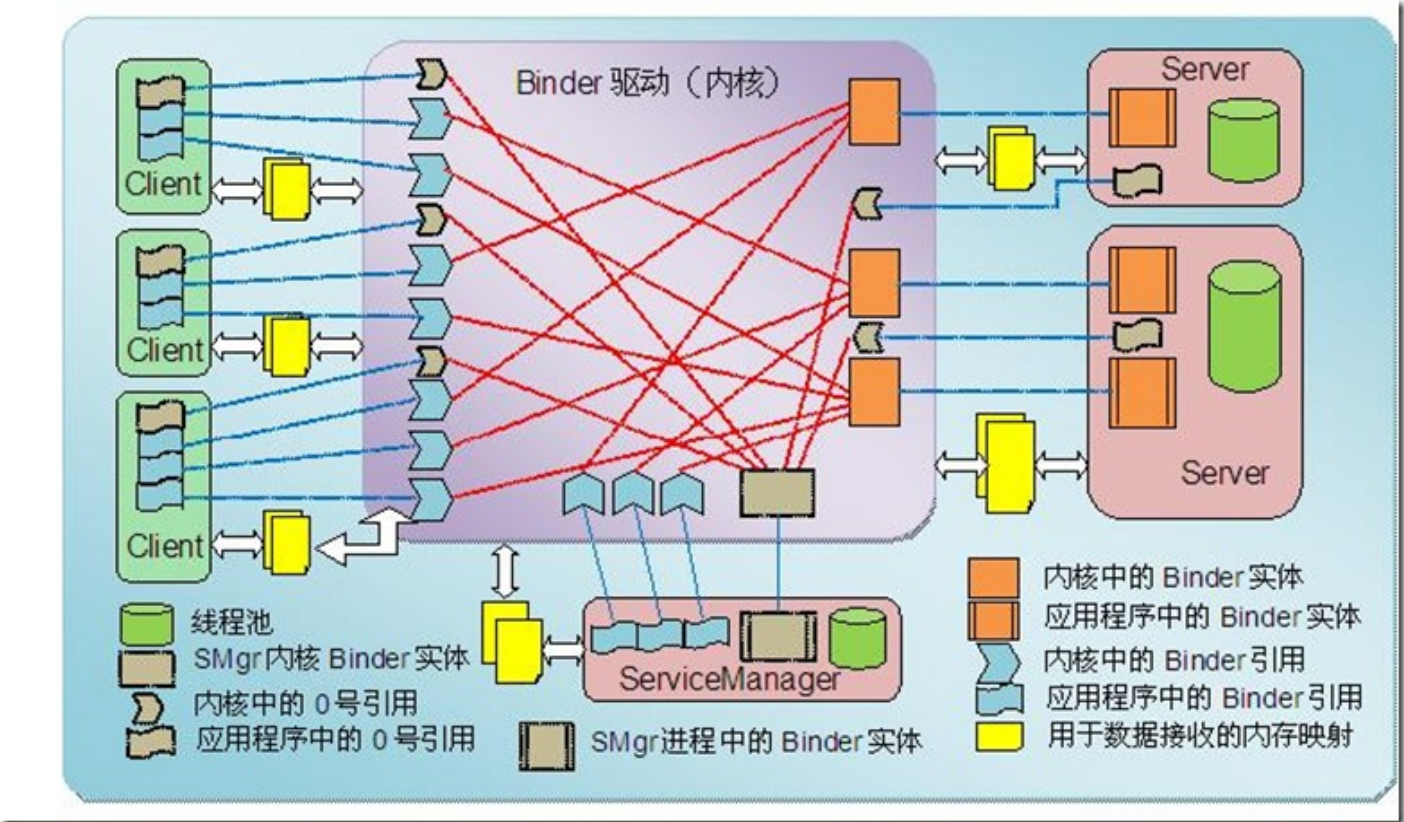
Client 获得实名Binder的引用

Server向ServiceManager注册了Binder实体及其名字后，Client就可以通过名字获得该Binder的引用了。Client也利用保留的0号引用向ServiceManager请求访问某个Binder：我申请获得名字叫张三的Binder的引用。

ServiceManager收到这个连接请求，从请求数据包里获得Binder的名字，在查找表里找到该名字对应的条目，从条目中取出Binder的引用，将该引用作为回复发送给发起请求的Client。从面向对象的角度，这个Binder对象现在有了两个引用：一个位于ServiceManager中，一个位于发起请求的Client中。如果接下来有更多的Client请求该Binder，系统中就会有更多的引用指向该Binder，就象java里一个对象存在多个引用一样。而且类似的这些指向Binder的引用是强类型，从而确保只要有引用Binder实体就不会被释放掉。通过以上过程可以看出，ServiceManager象个火车票代售点，收集了所有火车的车票，可以通过它购买到乘坐各趟火车的票，即得到某个Binder的引用。

匿名 Binder

并不是所有Binder都需要注册给ServiceManager。Server端可以通过已经建立的Binder连接将创建的Binder实体传给Client，当然这条已经建立的Binder连接必须是通过实名Binder实现。由于这个Binder没有向ServiceManager注册名字，所以是个匿名Binder。Client将会收到这个匿名Binder的引用，通过这个引用向位于Server中的实体发送请求。匿名Binder为通信双方建立一条私密通道，只要Server没有把匿名Binder发给别的进程，别的进程就无法通过穷举或猜测等任何方式获得该Binder的引用，向该Binder发送请求。



Binder 协议

Binder协议基本格式是（命令+数据），使用ioctl(fd, cmd, arg)函数实现交互。命令由参数cmd承载，数据由参

数arg承载，随cmd不同而不同。下表列举了所有命令及其所对应的数据：

表 2 Binder通信命令字

命令	含义	arg
BC_TRANSACTION BC_REPLY	BC_TRANSACTION用于Client向Server发送请求数据；BC_REPLY用于Server向Client发送回复（应答）数据。其后面紧接着一个binder_transaction_data结构体表明要写入的数据。	Struct binder_transaction_data
BC_ACQUIRE_RESULT BC_ATTEMPT_ACQUIRE	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。	指向需要释放的缓存区的指针；该指针位于收到的Binder数据包
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。	32位Binder引用号
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	void *ptr：Binder实体在用户空间中的指针 void *cookie：与该实体相关的附加数据
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。	uint32 *ptr：需要得到死亡通知的Binder引用 void **cookie：与死亡通知相关的信息，驱动会在发出死亡通知时返回给发出请求的进程。
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。	void **cookie

这其中最常用的命令是BINDER_WRITE_READ。该命令的参数包括两部分数据：一部分是向Binder写入的数据，一部分是要从Binder读出的数据，驱动程序先处理写部分再处理读部分。这样安排的好处是应用程序可以很灵活地处理命令的同步或异步。例如若要发送异步命令可以只填入写部分而将read_size置成0；若要只从Binder获得数据可以将写部分置空即write_size置成0；若要发送请求并同步等待返回数据可以将两部分都置上。

BINDER_WRITE_READ 之写操作

Binder写操作的数据时格式同样也是（命令+数据）。这时候命令和数据都存放在binder_write_read结构write_buffer域指向的内存空间里，多条命令可以连续存放。数据紧接着存放在命令后面，格式根据命令不同而不同。下表列举了Binder写操作支持的命令：

表 3 Binder写操作命令字

命令	含义	arg
BC_TRANSACTION BC_REPLY	BC_TRANSACTION用于Client向Server发送请求数据；BC_REPLY用于Server向Client发送回复（应答）数据。其后面紧接着一个binder_transaction_data结构体表明要写入的数据。	Struct binder_transaction_data
BC_ACQUIRE_RESULT BC_ATTEMPT_ACQUIRE	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。	指向需要释放的缓存区的指针；该指针位于收到的Binder数据包
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。	32位Binder引用号
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	void *ptr：Binder实体在用户空间中的指针 void *cookie：与该实体相关的附加数据
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。	uint32 *ptr：需要得到死亡通知的Binder引用 void **cookie：与死亡通知相关的信息，驱动会在发出死亡通知时返回给发出请求的进程。
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。	void **cookie

在这些命令中，最常用的是BC_TRANSACTION/BC_REPLY命令对，Binder数据通过这对命令发送给接收方。这对命令所承载的数据包由结构体struct binder_transaction_data定义。Binder交互有同步和异步之分，利用binder_transaction_data中 flag域区分。如果flag域的TF_ONE_WAY位为1则为异步交互，即Client端发送完请求交互即结束，Server端不再返回BC_REPLY数据包；否则Server会返回BC_REPLY数据包，Client端必须等待接收完该数据包方才完成一次交互。

BINDER_WRITE_READ 之读操作

从Binder里读出的数据格式和向Binder中写入的数据格式一样，采用（消息ID+数据）形式，并且多条消息可以连续存放。下表列举了从 Binder读出的命令字及其相应的参数：

表 4 Binder读操作消息ID

命令	含义	arg
BR_ERROR	发生内部错误（如内存分配失败）	
BR_OK 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_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。	binder_transaction_data
BR_ACQUIRE_RESULT BR_ATTEMPT_ACQUIRE BR_FINISHED	尚未实现	
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_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。	void *ptr：Binder实体在用户空间中的指针 void *cookie：与该实体相关的附加数据
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。	void **cookie：在使用BC_REQUEST_DEATH_NOTIFICATION注册死亡通知时的附加参数。 ^{[1] 载}
BR_FAILED_REPLY	如果发送非法引用号则返回该消息	

和写数据一样，其中最重要的消息是BR_TRANSACTION 或BR_REPLY，表明收到了一个格式为binder_transaction_data的请求数据包（BR_TRANSACTION）或返回数据包（BR_REPLY）。

Binder收发数据包结构binder_transaction_data

[cpp] C ?

```
01. struct binder_transaction_data {
02.     /* The first two are only used for bcTRANSACTION and brTRANSACTION,
03.      * identifying the target and contents of the transaction.
04.      */
05.     union {
06.         size_t  handle; /* target descriptor of command transaction */
07.         void    *ptr;   /* target descriptor of return transaction */
08.     } target;
09.     void        *cookie; /* target object cookie */
10.     unsigned int  code;   /* transaction command */
11.
12.     /* General information about the transaction. */
13.     unsigned int  flags;
14.     pid_t         sender_pid;
15.     uid_t         sender_euid;
16.     size_t        data_size; /* number of bytes of data */
17.     size_t        offsets_size; /* number of bytes of offsets */
18. }
```

```
19.     /* If this transaction is inline, the data immediately
20.      * follows here; otherwise, it ends with a pointer to
21.      * the data buffer.
22.      */
23.     union {
24.         struct {
25.             /* transaction data */
26.             const void *buffer;
27.             /* offsets from buffer to flat_binder_object structs */
28.             const void *offsets;
29.         } ptr;
30.         uint8_t buf[8];
31.     } data;
32. };
```

成员	含义
union { size_t handle; void *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）。
void *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。
unsigned int 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。
unsigned int 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，接收方不会返回任何数据。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。
size_t data_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。
size_t offsets_size	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 { const void *buffer; const void *offsets; } ptr; uint8_t buf[8]; } data;	data.buffer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。 这里有必要再强调一下offsets_size和data.offsets两个成员，这是Binder通信有别于其它IPC的地方。如前述，Binder采用面向对象的设计思想，一个Binder实体可以发送给其它进程从而建立许多跨进程的引用；另外这些引用也可以在进程之间传递，就象 java里将一个引用赋给另一个引用一样。为Binder在不同进程中建立引用必须有驱动的参与，由驱动在内核创建并注册相关的数据结构后接收方才能使用 该引用。而且这些引用可以是强类型，需要驱动为其维护引用计数。然而这些跨进程传递的Binder混杂在引用程序发送的数据包里，数据格式完全由用户定义，如果不把它们——标记出来告知驱动，驱动将无法从数据中将它们提取出来。于是就使用数组data.offsets存放用户数据中每个Binder相对 data.buffer的偏移量，用offsets_size表示这个数组的大小。驱动在发送数据包时会根据data.offsets和 offset_size将散落于data.buffer中的Binder找出来并——为它们创建相关的数据结构。在数据包中传输的Binder是类型为 struct flat_binder_object的结构体，对于接收方来说，该结构只相当于一个定长的消息头，真正的用户数据存放在data.buffer所指向的缓存区中。如果发送方在数据中内嵌了一个或多个Binder，接收到的数据包中同样会用data.offsets和 offset_size指出每个Binder的位置和总个数。不过通常接收方可以忽略这些信息，因为接收方是知道数据格式的，参考双方约定的格式定义就能知道这些Binder在什么位置。
--	--

Binder 的表述

考察一次Binder通信的全过程会发现，Binder存在于系统以下几个部分中：

应用程序进程：又分为Server进程和Client进程

Binder驱动：Server和Client有不同表述形

传输数据：由于Binder可以跨进程传递，需要在传输数据中予以表述

在系统不同部分，Binder实现的功能不同，表现形式也不一样的。接下来逐一探讨Binder在各部分所扮演的角色和使用的数据结构。

Binder 在应用程序中的表述

虽然Binder用到了面向对象的思想，但并不限制应用程序一定要使用面向对象的语言，无论是C语言还是C++语言都可以很容易的使用Binder 来通信。例如尽管Android主要使用java或C++，像SMgr这么重要的进程就是用C语言实现的。不过面向对象的方式表述起来更方便，所以本文假设应用程序是用面向对象语言实现的。Binder本质上只是一种底层通信方式，和具体服务没有关系。为了提供具体服务，Server必须提供一套接口函数以便Client通过远程访问使用各种服务。这时通常采用Proxy设计模式：将接口函数定义在一个抽象类中，Server和Client都会以该抽象类为基类实现所有接口函数，所不同的是Server端是真正的功能实现，而Client端是对这些函数远程调用请求的包装。如何将Binder和Proxy设计模式结合起来是应用程序实现面向对象Binder通信的根本问题。

Binder 在Server端的表述 - Binder实体

做为Proxy设计模式的基础，首先定义一个抽象接口类封装Server所有功能，其中包含一系列纯虚函数留待Server和Proxy各自实现。由于这些函数需要跨进程调用，须为其一一编号，从而Server可以根据收到的编号决定调用哪个函数。其次就要引入Binder了。Server端定义另一个Binder抽象类处理来自Client的Binder请求数据包，其中最重要的成员是虚函数onTransact()。该函数分析收到的数据包，调用相应的接口函数处理请求。接下来采用继承方式以接口类和Binder抽象类为基类构建Binder在Server中的实体，实现基类里所有的虚函数，包括公共接口函数以及数据包处理函数：onTransact()。这个函数的输入是来自Client的binder_transaction_data结构的数据包。前面提到，该结构里有个成员code，包含这次请求的接口函数编号。onTransact()将case-by-case地解析code值，从数据包里取出函数参数，调用接口类中相应的已经实现的公共接口函数。函数执行完毕，如果需要返回数据就再构建一个binder_transaction_data包将返回数据包填入其中。那么各个Binder实体的onTransact()又是什么时候调用呢？这就需要驱动参与了。前面说过，Binder实体须要以Binder传输结构flat_binder_object形式发送给其它进程才能建立Binder通信，而Binder实体指针就存放在该结构的handle域中。驱动根据Binder位置数组从传输数据中获取该Binder的传输结构，为它创建位于内核中的Binder节点，将Binder实体指针记录在该节点中。如果接下来有其它进程向该Binder发送数据，驱动会根据节点中记录的信息将Binder实体指针填入binder_transaction_data的target.ptr中返回给接收线程。接收线程从数据包中取出该指针，reinterpret_cast成Binder抽象类并调用onTransact()函数。由于这是个虚函数，不同的Binder实体中有各自的实现，从而可以调用到不同Binder实体提供的onTransact()。

Binder 在Client端的表述 - Binder引用

做为Proxy设计模式的一部分，Client端的Binder同样要继承Server提供的公共接口类并实现公共函数。但这不是真正的实现，而是对远程函数调用的包装：将函数参数打包，通过Binder向Server发送申请并等待返回值。为此Client端的Binder还要知道Binder实体的相关信息，即对Binder实体的引用。该引用或是由SMgr转发过来的，对实名Binder的引用或是由另一个进程直接发送过来的，匿名Binder的引用。由于继承了同样的公共接口类，Client Binder提供了与Server Binder一样的函数原型，使用户感觉不出Server是运行在本地还是远

端。Client Binder中，公共接口函数的实现方式是：创建一个binder_transaction_data数据包，将其对应的编码填入code域，将调用该函数所需的参数填入data.buffer指向的缓存中，并指明数据包的目的地址，那就是已经获得的对Binder实体的引用，填入数据包的target.handle中。注意这里和Server的区别：实际上target域是个联合体，包括ptr和handle两个成员，前者用于作为响应方的Server，指向Binder实体对应的内存空间；后者用于作为请求方的Client，存放Binder实体的引用，告知驱动数据包将路由给哪个实体。数据包准备好后，通过驱动接口发送出去。经过BC_TRANSACTION/BC_REPLY回合完成函数的远程调用并得到返回值。

Binder 在传输数据中的表述

Binder可以塞在数据包的有效数据中越进程边界从一个进程传递给另一个进程，这些传输中的Binder用结构 flat_binder_object表示，如下表所示：

```
[cpp]
01. struct flat_binder_object {
02.     /* 8 bytes for large_flat_header. */
03.     unsigned long    type;
04.     unsigned long    flags;
05.
06.     /* 8 bytes of data. */
07.     union {
08.         void         *binder;    /* local object */
09.         signed long  handle;    /* remote object */
10.     };
11.
12.     /* extra data associated with local object */
13.     void             *cookie;
14. };
```

成员	含义
unsigned long type	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，详见下节
unsigned long 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 { void *binder; signed long handle; };	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。
void *cookie	该域只对Binder实体有效，存放与该Binder有关的附加信息。

无论是Binder实体还是对实体的引用都从属与某个进程，所以该结构不能透明地在进程之间传输，必须有驱动的参与。例如当Server把Binder实体传递给Client时，在发送数据中，flat_binder_object中的type是BINDER_TYPE_BINDER，binder指向Server进程用户空间地址。如果透明传给接收端将毫无用处，驱动必须对数据流中的这个Binder做修改：将type改成BINDER_TYPE_HANDLE；为这个Binder在接收进程中创建位于内核中的引用并将引用号填入handle中。对于发送数据流中引用类型的Binder也要做同样转换。经过处理后接收进程

从数据流中取得的Binder引用才是有效的，才可以将其填入数据包binder_transaction_data的target.handle域，向Binder实体发送请求。

这样做也是出于安全性考虑：应用程序不能随便猜测一个引用号填入target.handle中就可以向Server请求服务了，因为驱动并没有为你在内核中创建该引用，必定会驱动被拒绝。唯有经过身份认证确认合法后，由‘权威机构’通过数据流授予你的Binder才能使用，因为这时驱动已经在内核中为你建立了引用，交给你的引用号是合法的。

BINDER_TYPE_BINDER

BINDER_TYPE_WEAK_BINDER

只有实体所在的进程能发送该类型的Binder。如果是第一次发送驱动将创建实体在内核中的节点，并保存binder，cookie，flag域。如果是第一次接收该Binder则创建实体在内核中的引用；将handle域替换为新建的引用号；将type域替换为 BINDER_TYPE_(WEAK_)HANDLE

BINDER_TYPE_HANDLE

BINDER_TYPE_WEAK_HANDLE

获得Binder引用的进程都能发送该类型Binder。驱动根据handle域提供的引用号查找建立在内核的引用。如果找到说明引用号合法，否则 拒绝该发送请求。如果收到的Binder实体位于接收进程中：将ptr域替换为保存在节点中的binder值；cookie替换为保存在节点中的cookie 值；type替换为BINDER_TYPE_(WEAK_)BINDER。

如果收到的Binder实体不在接收进程中：如果是第一次接收则创建实体在内核中的引用；将handle域替换为新建的引用号

BINDER_TYPE_FD 验证handle域中提供的打开文件号是否有效，无效则拒绝该发送请求。 在接收方创建新的打开文件号并将其与提供的打开文件描述结构绑定。

除了通常意义上用来通信的Binder，还有一种特殊的Binder：文件Binder。这种Binder的基本思想是：将文件看成Binder实体，进程打开的文件号看成Binder的引用。一个进程可以将它打开文件的文件号传递给另一个进程，从而另一个进程也打开了同一个文件，就象Binder 的引用在进程之间传递一样。

一个进程打开一个文件，就获得与该文件绑定的打开文件号。从Binder的角度，linux在内核创建的打开文件描述结构struct file是Binder的实体，打开文件号是该进程对该实体的引用。既然是Binder那么就可以在进程之间传递，故也可以用 flat_binder_object结构将文件Binder通过数据包发送至其它进程，只是结构中type域的值 为BINDER_TYPE_FD，表明 该Binder是文件Binder。而结构中的handle域则存放文件在发送方进程中的打开文件号。我们知道打开文件号是个局限于某个进程的值，一旦跨 进程就没有意义了。这一点和Binder实体用户指针或Binder引用号是一样的，若要跨进程同样需要驱动做转换。驱动在接收Binder的进程空间创 建一个新的打开文件号，将它与已有的打开文件描述结构struct file勾连上，从此该Binder实体又多了一个引用。新建的打开文件号覆盖flat_binder_object中原来的文件号交给接收进程。接收进 程利用它可以执行read()，write()等文件操作。

传个文件为啥要这么麻烦，直接将文件名用Binder传过去，接收方用open()打开不就行了吗？其实这还是有区别的。首先对同一个打开文件共享的层次不同：使用文件Binder打开的文件共享linux VFS中的struct file，struct dentry，struct inode结构，这意味着一个进程使用read()/write()/seek()改变了文件指针另一个进程的文件指针也会改变；而如果两个进程分别使用文件名打开同一文件则有各自的struct file结构，从而各自独立维护文件指针，互不干扰。其次是一些特殊设备文件要求在struct file一级共享才能使用，例如android的另一个驱动ashmem，它和Binder一样也是misc设备，用以实现进程间的共享内存。一个进程打开的ashmem文件只有通过文件Binder发送到另一个进程才能实现内存共享，这大大提高了内存共享的安全性，道理和Binder增强了IPC的安全性是一样的。

Binder 在驱动中的表述

驱动是Binder通信的核心，系统中所有的Binder实体以及每个实体在各个进程中的引用都登记在驱动中；驱动需要记录Binder引用 -> 实体之间多对一的关系；为引用找到对应的实体；在某个进程中为实体创建或查找到对应的引用；记录Binder的归属地（位于哪个进程中）；通过管理Binder的强/弱引用创建/销毁Binder实体等等。驱动里的Binder是什么时候创建的呢？前面提到过，为了实现实名Binder的注册，系统必须创建第一只鸡 - 为SMgr创建的，注册实名Binder专用的Binder实体，负责实名Binder注册过程中的进程间通信。既然创建了实体也要有对应的引用：驱动将所有进程中的0号引用都预留给该Binder实体，即一开始所有进程的0号引用都指注册实名Binder专用的Binder，无须特殊操作任何进程通过0号引用都可以注册实名Binder。接下来随着应用程序通过不断地注册实名Binder，不断向SMgr索要Binder的引用，不断将Binder从一个进程传递给另一个进程，越来越多的Binder以传输结构 - flat_binder_object的形式穿越驱动做跨进程的迁徙。由于binder_transaction_data中data.offset数组的存在，所有流经驱动的Binder都逃不过驱动的眼睛。Binder将对每个穿越进程边界的Binder做如下操作：检查传输结构的type域，如果是BINDER_TYPE_BINDER或BINDER_TYPE_WEAK_BINDER则创建Binder的实体；如果是BINDER_TYPE_HANDLE或BINDER_TYPE_WEAK_HANDLE则创建Binder的引用；如果是BINDER_TYPE_HANDLE则为进程打开文件，无须创建任何数据结构。详细过程可参考表7。随着越来越多的Binder实体或引用穿过驱动在进程间传递，驱动会在内核里创建越来越多的节点或引用，当然这个过程对用户来说是透明的。

Binder 实体在驱动中的表述

驱动中的Binder实体也叫‘节点’，隶属于提供实体的进程，由struct binder_node结构来表示：

```
[cpp] C #
01. struct binder_node {
02.     int debug_id;
03.     struct binder_work work;
04.     union {
05.         struct rb_node rb_node;
06.         struct hlist_node dead_node;
07.     };
08.     struct binder_proc *proc;
09.     struct hlist_head refs;
10.     int internal_strong_refs;
11.     int local_weak_refs;
```



```
12.     int local_strong_refs;
13.     void __user *ptr;
14.     void __user *cookie;
15.     unsigned has_strong_ref:1;
16.     unsigned pending_strong_ref:1;
17.     unsigned has_weak_ref:1;
18.     unsigned pending_weak_ref:1;
19.     unsigned has_async_transaction:1;
20.     unsigned accept_fds:1;
21.     unsigned min_priority:8;
22.     struct list_head async_todo;
23. };
```

成员	含义
int debug_id	用于调试
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。
void __user *ptr	指向用户空间Binder实体的指针，来自于flat_binder_object的binder成员
void __user *cookie	指向用户空间的附加指针，来自于flat_binder_object的cookie成员
unsigned has_strong_ref; unsigned pending_strong_ref; unsigned has_weak_ref; unsigned pending_weak_ref	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
unsigned 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 - async_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。
unsigned accept_fds	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。
int 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。
struct list_head async_todo	异步交互等待队列，用于分流发往本节点的异步交互包

每个进程都有一棵红黑树用于存放创建好的节点，以Binder在用户空间的指针作为索引。每当在传输数据中侦测到一个代表Binder实体的 flat_binder_object，先以该结构的binder指针为索引搜索红黑树；如果没找到就创建一个新节点添加到树中。由于对于同一个进程来说 内存地址是唯一的，所以不会重复建设造成混乱。

Binder 引用在驱动中的表述

和实体一样，Binder的引用也是驱动根据传输数据中的flat_binder_object创建的，隶属于获得该引用的进程，用struct binder_ref结构体表示：

```
[cpp] 01. struct binder_ref {
```

```
02.      /* Lookups needed: */
03.      /*   node + proc => ref (transaction) */
04.      /*   desc + proc => ref (transaction, inc/dec ref) */
05.      /*   node => refs + procs (proc exit) */
06.      int debug_id;
07.      struct rb_node rb_node_desc;
08.      struct rb_node rb_node_node;
09.      struct hlist_node node_entry;
10.      struct binder_proc *proc;
11.      struct binder_node *node;
12.      uint32_t desc;
13.      int strong;
14.      int weak;
15.      struct binder_ref_death *death;
16.  };
```

成员	含义
int debug_id	用于调试
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）为索引添入该树中。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）
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 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’。

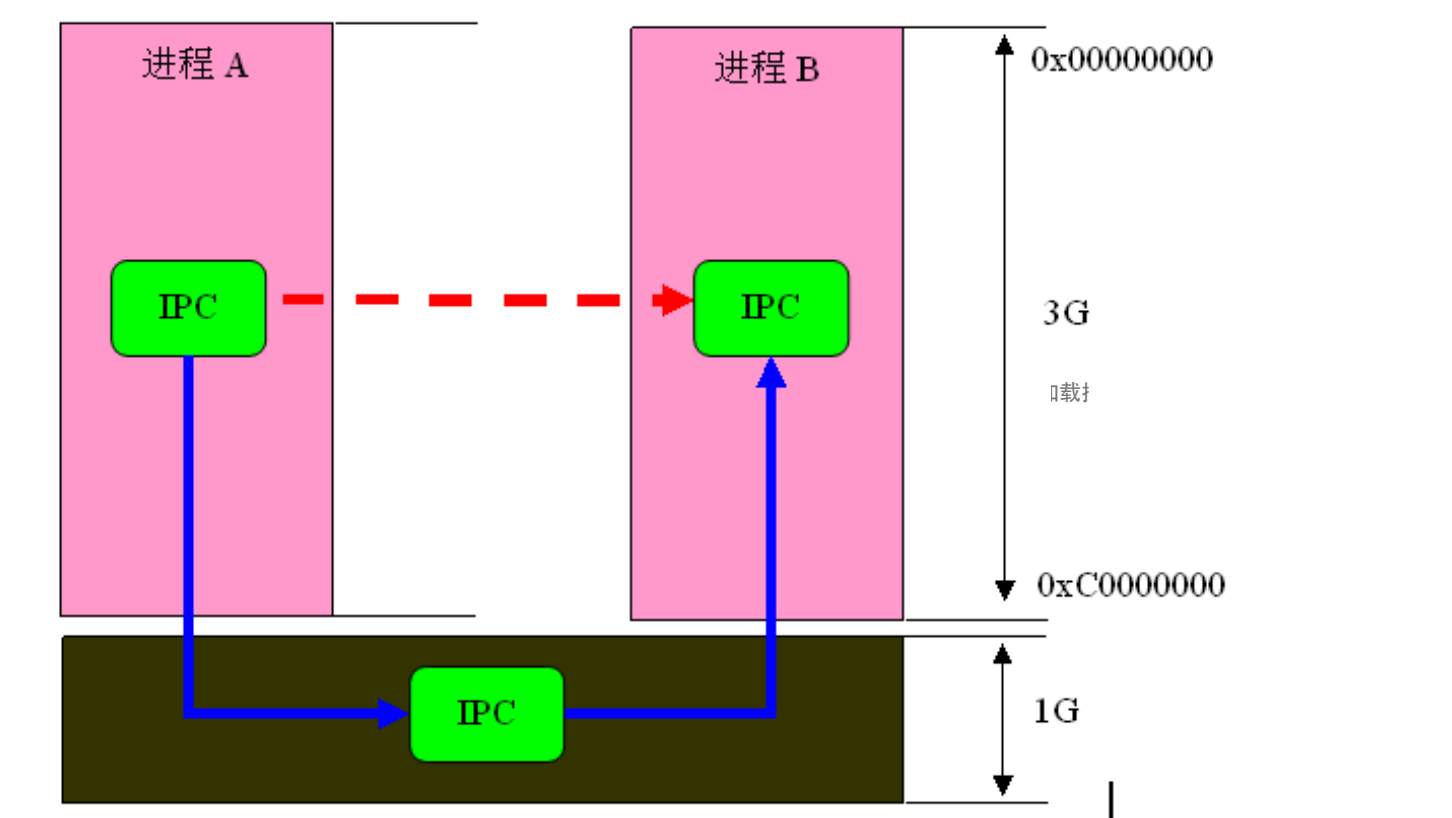
就象一个对象有很多指针一样，同一个Binder实体可能有很多引用，不同的是这些引用可能分布在不同的进程中。和实体一样，每个进程使用红黑树存放所有该进程正在使用的引用。但Binder的引用可以通过两个键值索引：

- 1) 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）。
- 2) 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保留给SMgr，其它值由驱动在创建引用时动态分配。向Binder 发送数据包时，应用程序通过将引用号填入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。

Binder 内存映射和接收缓存区管理

暂且撇开Binder，考虑一下传统的IPC方式中，数据是怎样从发送端到达接收端的呢？通常的做法是，发送方将准备好的数据存放在缓存区中，调用API通过系统调用进入内核中。内核服务程序在内核空间分配内存，将数据从

发送方缓存区复制到内核缓存区中。接收方读数据时也要提供一块缓存区，内核将数据从内核缓存区拷贝到接收方提供的缓存区中并唤醒接收线程，完成一次数据发送。这种存储-转发机制有两个缺陷：首先是效率低下，需要做两次拷贝：用户空间->内核空间->用户空间。Linux使用copy_from_user()和copy_to_user()实现这两个跨空间拷贝，在此过程中如果使用了高端内存（high memory），这种拷贝需要临时建立/取消页面映射，造成性能损失。其次是接收数据的缓存要由接收方提供，可接收方不知道到底要多大的缓存才够用，只能开辟尽量大的空间或先调用API接收消息头获得消息体大小，再开辟适当的空间接收消息体。两种做法都有不足，不是浪费空间就是浪费时间。



Binder采用一种全新策略：由Binder驱动负责管理数据接收缓存。我们注意到Binder驱动实现了mmap()系统调用，这对字符设备是比较特殊的，因为mmap()通常用在有物理存储介质的文件系统上，而象Binder这样没有物理介质，纯粹用来通信的字符设备没必要支持mmap()。Binder驱动当然不是为了在物理介质和用户空间做映射，而是用来创建数据接收的缓存空间。先看mmap()是如何使用的：

```
[cpp] 1 fd = open("/dev/binder", O_RDWR);
2 mmap(NULL, MAP_SIZE, PROT_READ, MAP_PRIVATE, fd, 0);
```

这样Binder的接收方就有了一片大小为MAP_SIZE的接收缓存区。mmap()的返回值是内存映射在用户空间的地址，不过这段空间是由驱动管理，用户不必也不能直接访问（映射类型为PROT_READ，只读映射）。

接收缓存区映射好后就可以做为缓存池接收和存放数据了。前面说过，接收数据包的结构为binder_transaction_data，但这只是消息头，真正有效负荷位于data.buffer所指向的内存中。这片内存不需要接收方提供，恰恰是来自mmap()映射的这片缓存池。在数据从发送方向接收方拷贝时，驱动会根据发送数据包的大小，使用最佳匹配算法从缓存池中找到一块大小合适的空间，将数据从发送缓存区复制过来。要注意的是，存放binder_transaction_data结构本身以及表4中所有消息的内存空间还是得由接收者提供，但这些数据大小固定，数

量也不多，不会给接收方造成不便。映射的缓存池要足够大，因为接收方的线程池可能会同时处理多条并发的交互，每条交互都需要从缓存池中获取目的存储区，一旦缓存池耗竭将产生导致无法预期的后果。

有分配必然有释放。接收方在处理完数据包后，就要通知驱动释放data.buffer所指向的内存区。在介绍Binder协议时已经提到，这是由命令BC_FREE_BUFFER完成的。

通过上面介绍可以看到，驱动为接收方分担了最为繁琐的任务：分配/释放大小不等，难以预测的有效负荷缓存区，而接收方只需要提供缓存来存放大小固定，可以预测的消息头即可。在效率上，由于mmap()分配的内存是映射在接收方用户空间里的，所有总体效果就相当于对有效负荷数据做了一次从发送方用户空间到接收方用户空间的直接数据拷贝，省去了内核中暂存这个步骤，提升了一倍的性能。顺便再提一点，Linux内核实际上没有从一个用户空间到另一个用户空间直接拷贝的函数，需要先用copy_from_user()拷贝到内核空间，再用copy_to_user()拷贝到另一个用户空间。为了实现用户空间到用户空间的拷贝，mmap()分配的内存除了映射进了接收方进程里，还映射进了内核空间。所以调用copy_from_user()将数据拷贝进内核空间也相当于拷贝进了接收方的用户空间，这就是Binder只需一次拷贝的‘秘密’。

Binder 接收线程管理

Binder通信实际上是位于不同进程中的线程之间的通信。假如进程S是Server端，提供Binder实体，线程T1从Client进程C1中通过Binder的引用向进程S发送请求。S为了处理这个请求需要启动线程T2，而此时线程T1处于接收返回数据的等待状态。T2处理完请求就会将处理结果返回给T1，T1被唤醒得到处理结果。在这过程中，T2仿佛T1在进程S中的代理，代表T1执行远程任务，而给T1的感觉就是象穿越到S中执行一段代码又回到了C1。为了使这种穿越更加真实，驱动会将T1的一些属性赋给T2，特别是T1的优先级nice，这样T2会使用与T1类似的时间完成任务。很多资料会用‘线程迁移’来形容这种现象，容易让人产生误解。一来线程根本不可能在进程之间跳来跳去，二来T2除了和T1优先级一样，其它没有相同之处，包括身份，打开文件，栈大小，信号处理，私有数据等。

对于Server进程S，可能会有许多Client同时发起请求，为了提高效率往往开辟线程池并发处理收到的请求。怎样使用线程池实现并发处理呢？这和具体的IPC机制有关。拿socket举例，Server端的socket设置为侦听模式，有一个专门的线程使用该socket侦听来自Client的连接请求，即阻塞在accept()上。这个socket就象一只生蛋的鸡，一旦收到来自Client的请求就会生一个蛋 - 创建新socket并从accept()返回。侦听线程从线程池中启动一个工作线程并将刚下的蛋交给该线程。后续业务处理就由该线程完成并通过这个单与Client实现交互。

可是对于Binder来说，既没有侦听模式也不会下蛋，怎样管理线程池呢？一种简单的做法是，不管三七二十一，先创建一堆线程，每个线程都用BINDER_WRITE_READ命令读Binder。这些线程会阻塞在驱动为该Binder的等待队列上，一旦有来自Client的数据驱动会从队列中唤醒一个线程来处理。这样做简单直观，省去了线程池，但一开始就创建一堆线程有点浪费资源。于是Binder协议设置了专门命令或消息帮助用户管理线程池，包括：

- 1) BINDER_SET_MAX_THREADS
- 2) BC_REGISTER_LOOP
- 3) BC_ENTER_LOOP
- 4) BC_EXIT_LOOP

·5) BR_SPAWN_LOOPER

首先要管理线程池就要知道池子有多大，应用程序通过INDER_SET_MAX_THREADS告诉驱动最多可以创建几个线程。以后每个线程在创建，进入主循环，退出主循环时都要分别使用BC_REGISTER_LOOP，BC_ENTER_LOOP，BC_EXIT_LOOP告知驱动，以便驱动收集和记录当前线程池的状态。每当驱动接收完数据包返回读Binder的线程时，都要检查一下是不是已经没有闲置线程了。如果是，而且线程总数不会超出线程池最大线程数，就会在当前读出的数据包后面再追加一条BR_SPAWN_LOOPER消息，告诉用户线程即将不够用了，请再启动一些，否则下一个请求可能不能及时响应。新线程一启动又会通过BC_xxx_LOOP告知驱动更新状态。这样只要线程没有耗尽，总是有空闲线程在等待队列中随时待命，及时处理请求。

关于工作线程的启动，Binder驱动还做了一点小小的优化。当进程P1的线程T1向进程P2发送请求时，驱动会先查看一下线程T1是否也正在处理来自P2某个线程请求但尚未完成（没有发送回复）。这种情况通常发生在两个进程都有Binder实体并互相对发时请求时。假如驱动在进程P2中发现了这样的线程，比如说T2，就会要求T2来处理T1的这次请求。因为T2既然向T1发送了请求尚未得到返回包，说明T2肯定（或将会）阻塞在读取返回包的状态。这时候可以让T2顺便做点事情，总比等在那里闲着好。而且如果T2不是线程池中的线程还可以为线程池分担部分工作，减少线程池使用率。

数据包接收队列与（线程）等待队列管理

通常数据传输的接收端有两个队列：数据包接收队列和（线程）等待队列，用以缓解供需矛盾。当超市里的进货（数据包）太多，货物会堆积在仓库里；购物的人（线程）太多，会排队等待在收银台，道理是一样的。在驱动中，每个进程有一个全局的接收队列，也叫to-do队列，存放不是发往特定线程的数据包；相应地有一个全局等待队列，所有等待从全局接收队列里收数据的线程在该队列里排队。每个线程有自己私有的to-do队列，存放发送给该线程的数据包；相应的每个线程都有各自私有等待队列，专门用于本线程等待接收自己to-do队列里的数据。虽然名叫队列，其实线程私有等待队列中最多只有一个线程，即它自己。

由于发送时没有特别标记，驱动怎么判断哪些数据包该送入全局to-do队列，哪些数据包该送入特定线程的to-do队列呢？这里有两条规则。规则1：Client发给Server的请求数据包都提交到Server进程的全局to-do队列。不过有个特例，就是上节谈到的Binder对工作线程启动的优化。经过优化，来自T1的请求不是提交给P2的全局to-do队列，而是送入了T2的私有to-do队列。规则2：对同步请求的返回数据包（由BC_REPLY发送的包）都发送到发起请求的线程的私有to-do队列中。如上面的例子，如果进程P1的线程T1发给进程P2的线程T2的是同步请求，那么T2返回的数据包将送进T1的私有to-do队列而不会提交到P1的全局to-do队列。

数据包进入接收队列的潜规则也就决定了线程进入等待队列的潜规则，即一个线程只要不接收返回数据包则应该在全局等待队列中等待新任务，否则就应该在其私有等待队列中等待Server的返回数据。还是上面的例子，T1在向T2发送同步请求后就必须等待在它私有等待队列中，而不是在P1的全局等待队列中排队，否则将得不到T2的返回的数据包。

这些潜规则是驱动对Binder通信双方施加的限制条件，体现在应用程序上就是同步请求交互过程中的线程一致性：1) Client端，等待返回包的线程必须是发送请求的线程，而不能由一个线程发送请求包，另一个线程等待接

收包，否则将收不到返回包；2) Server端，发送对应返回数据包的线程必须是收到请求数据包的线程，否则返回的数据包将无法送交发送请求的线程。这是因为返回数据包的目的 Binder不是用户指定的，而是驱动记录在收到请求数据包的线程里，如果发送返回包的线程不是收到请求包的线程驱动将无从知晓返回包将送往何处。

接下来探讨一下Binder驱动是如何递交同步交互和异步交互的。我们知道，同步交互和异步交互的区别是同步交互的发送（client）端在发出请求数据包后须要等待接收（Server）端的返回数据包，而异步交互的发送端发出请求数据包后交互即结束。对于这两种交互的请求数据包，驱动可以不管三七二十一，统统丢到接收端的to-do队列中一个个处理。但驱动并没有这样做，而是对异步交互做了限流，令其为同步交互让路，具体做法是：对于某个 Binder实体，只要有一个异步交互没有处理完毕，例如正在被某个线程处理或还在任意一条to-do队列中排队，那么接下来发给该实体的异步交互包将不再投递到to-do队列中，而是阻塞在驱动为该实体开辟的异步交互接收队列（Binder节点的async_todo域）中，但这期间同步交互依旧不受限制直接进入to-do队列获得处理。一直到该异步交互处理完毕下一个异步交互方可以脱离异步交互队列进入to-do队列中。之所以要这么做是因为同步交互的请求端需要等待返回包，必须迅速处理完毕以免影响请求端的响应速度，而异步交互属于‘发射后不管’，稍微延时一点不会阻塞其它线程。所以用专门队列将过多的异步交互暂存起来，以免突发大量异步交互挤占Server端的处理能力或耗尽线程池里的线程，进而阻塞同步交互。