Binder

1 Binder 是什么

Binder 是一种进程间通信机制，其实是提供远程过程调用（RPC）功能. 从英文字面上意思看，Binder 具有粘结剂的意思，那么它把什么东西粘结在一起呢？在 Android 系统的 Binder 机制中，由一系统组件组成，分别是 Client、Server、Service Manager 和 Binder 驱动程序，其中 Client、Server 和 Service Manager 运行在用户空间，Binder 驱动程序运行内核空间。Binder 就是一种把这四个组件粘合在一起的粘结剂了，其中，核心组件便是 Binder 驱动程序了，Service Manager 提供了辅助管理的功能，Client 和 Server 正是在 Binder 驱动和 Service Manager 提供的基础设施上，进行 Client-Server 之间的通信。Service Manager 和 Binder 驱动已经在 Android 平台中实现好，开发者只要按照规范实现自己的 Client 和 Server 组件就可以了。

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

IPC	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。

2 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。

面向对象思想的引入将进程间通信转化为通过对某个 Binder 对象的引用调用该对象的方法，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。

3 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 Binder 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。

3.1 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。

3.2 ServiceManager 与实名 Binder

和 DNS 类似，SMgr 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 地址外还有自己的网址。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 SMgr，通知 SMgr 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 SMgr 对实体的引用，将名字及新建的引用打包传递给 SMgr。SMgr 收数据包后，从中取出名字和引用填入一张查找表中。

细心的读者可能会发现其中的蹊跷：SMgr 是一个进程，Server 是另一个进程，Server 向 SMgr 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：SMgr 和其它进程同样采用 Binder 通信，SMgr 是 Server 端，有自己的 Binder 对象（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。SMgr 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 SMgr 时 Binder 驱动会自动为它创建 Binder 实体(这就是那只预先造好的鸡)。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 SMgr 注册自己 Binder 就必需通过 0 这个引用号和 SMgr 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 ,你必须预先手工或动态配置好。要注意这里说的 Client 是相对 SMgr 而言的，一个应用程序可能是个提供服务的 Server，但对 SMgr 来说它仍然是个 Client。

3.3 Client 获得实名 Binder 的引用

Server 向 SMgr 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 SMgr 请求访问某个 Binder：我申请获得名字叫张三的 Binder 的引用。SMgr 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 SMgr 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，SMgr 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-得到某个 Binder 的引用。

3.4 匿名 Binder

并不是所有 Binder 都需要注册给 SMgr 广而告之的。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 SMgr 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。

下图展示了参与 Binder 通信的所有角色，将在以后章节中一一提到。

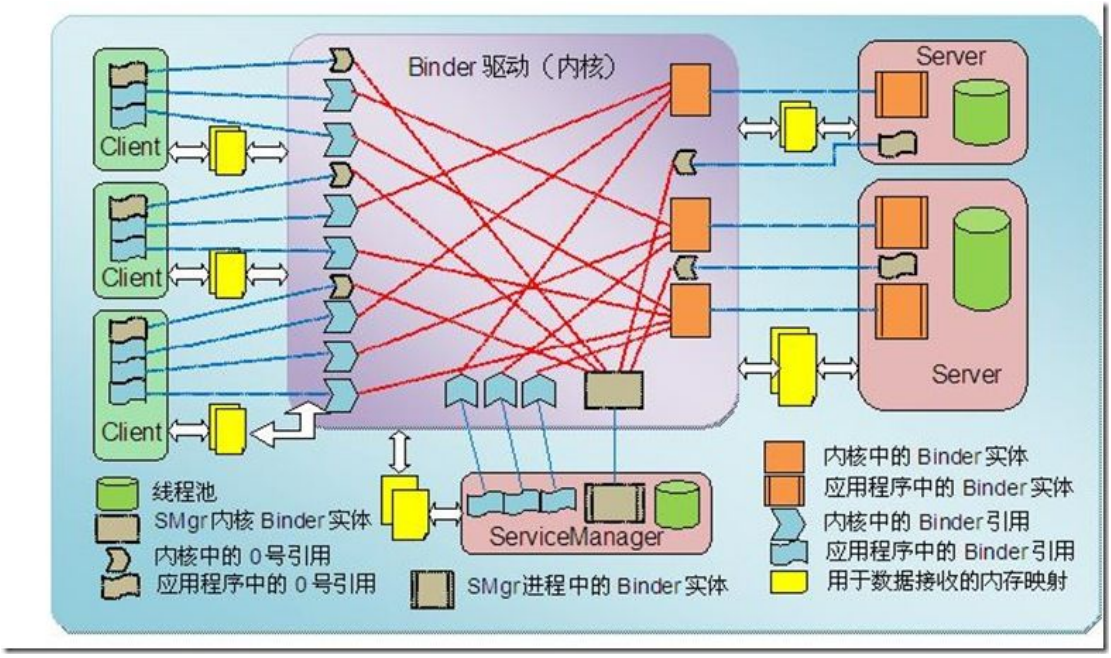

图 1 Binder 通信示例

4 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

BINDER_WRITE_READ	该命令向Binder写入或读取数据。参数分为两段：写部分和读部分。如果write_size不为0就先将write_buffer里的数据写入Binder；如果read_size不为0再从Binder中读取数据存入read_buffer中。write_consumed和read_consumed表示操作完成时Binder驱动实际写入或读出的数据个数。	struct binder_write_read {
		signed long write_size;
		signed long write_consumed;
		unsigned long write_buffer;
		signed long read_size;
		signed long read_consumed;
		unsigned long read_buffer;
		};

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。由于Client是并发向Server端发送请求的，Server端必须开辟线程池为这些并发请求提供服务。告知驱动线程池的最大值是为了让驱动发现线程数达到该值时不要再命令接收端启动新的线程。	int max_threads;
BINDER_SET_CONTEXT_MGR	将当前进程注册为SMgr。系统中同时只能存在一个SMgr。只要当前的SMgr没有调用close()关闭Binder驱动就不能有别的进程可以成为SMgr。	---
BINDER_THREAD_EXIT	通知Binder驱动当前线程退出了。Binder会为所有参与Binder通信的线程（包括Server线程池中的线程和Client发出请求的线程）建立相应的数据结构。这些线程在退出时必须通知驱动释放相应的数据结构。	---
BINDER_VERSION	获得Binder驱动的版本号。	---

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。

4.1 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

cmd	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数据包中
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

在这些命令中，最常用的是 BC_TRANSACTION/BC_REPLY 命令对，Binder 请求和应答数据就是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。

4.2 BINDER_WRITE_READ ：从 Binder 读出数据

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

消息	含义	参数
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	---

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）。

4.3 struct binder_transaction_data ：收发数据包结构

该结构是 Binder 接收/发送数据包的标准格式，每个成员定义如下：

表 5 Binder 收发数据包结构：binder_transaction_data

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，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。

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 混杂在应用程序发送的数据包里，数据格式由用户定义，如果不把它们一一标记出来告知驱动，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 找出来并一一为它们创

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，详见后文。

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。

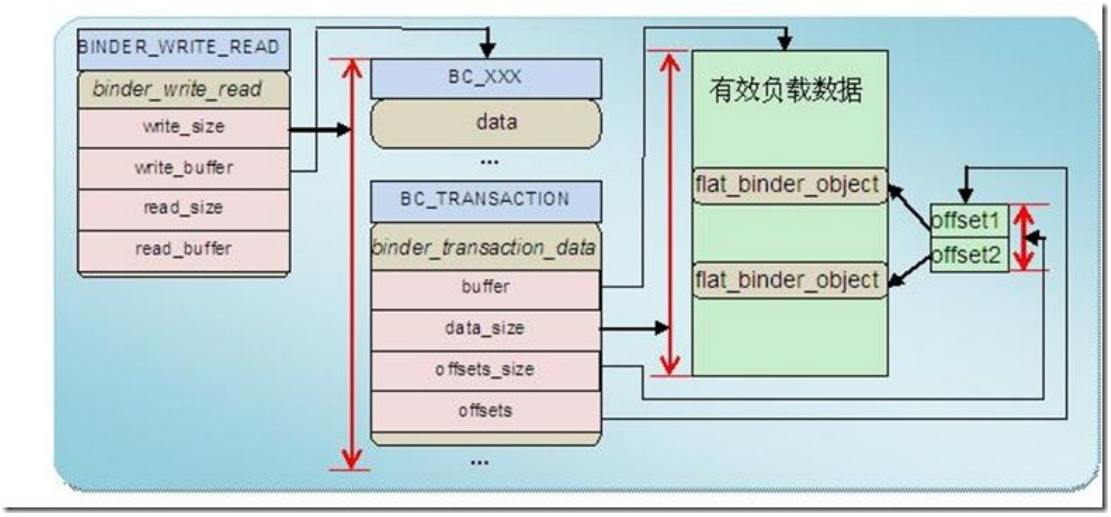

图 2 BINDER_WRITE_READ 数据包实例

5 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：分别位于 Server 进程和 Client 进程中

· Binder 驱动：分别管理为 Server 端的 Binder 实体和 Client 端的引用

· 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。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。

5.1 Binder 在应用程序中的表述

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。

5.1.1 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 Binde 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()。

5.1.2 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，对匿名 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，公共接口函数的包装方式是：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，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，前者用于接收数据包的 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。

5.2 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：

表 6 Binder 传输结构：flat_binder_object

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，必须经过驱动翻译。例如当 Server 把 Binder 实体传递给 Client 时，在发送数据流中，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 该成 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。对于发生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，必定会被驱动拒绝。唯有经过身份认证确认合法后，由‘权威机构’（Binder 驱动）亲手授予你的 Binder 才能使用，因为这时驱动已经在内核中为你使用该 Binder 做了注册，交给你的引用号是合法的。

下表总结了当 flat_binder_object 结构穿过驱动时驱动所做的操作：

表 7 驱动对 flat_binder_object 的操作

Binder 类型（ type 域）	在发送方的操作	在接收方的操作
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。

5.3 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，用于注册实名 Binder 的 Binder 实体，负责实名 Binder 注册过程中的进程间通信。既然创建了实体就要有对应的引用：驱动将所有进程中的 0 号引用都预留给该 Binder 实体 ,即所有进程的 0 号引用天然地都指向注册实名 Binder 专用的 Binder，无须特殊操作即可以使用 0 号引用来注册实名 Binder。接下来随着应用程序不断地注册实名 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 将对这些穿越进程边界的 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。

5.3.1 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：

表 8 Binder 节点描述结构：binder_node

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－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。

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。

5.3.2 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：

表 9 Binder 引用描述结构：binder_ref

成员	含义
int debug_id;	调试用
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）
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。

6 Binder 内存映射和接收缓存区管理

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：

```
fd = open("/dev/binder", O_RDWR);
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，真正的有效负荷位于 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，最大空间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’。

下次讲线程管理和死亡通知 ~ ~

7 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 会使用和 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 就象一只会生蛋的鸡，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 设置的等待队列上，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 协议引入了专门命令或消息帮助用户管理线程池，包括：

· INDER_SET_MAX_THREADS

· BC_REGISTER_LOOP

· BC_ENTER_LOOP

· BC_EXIT_LOOP

· 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。

8 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请求端(client)在发出请求数据包后须要等待应答端(Server)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。

9 总结

Binder 使用 Client-Server 通信方式，安全性好，简单高效，再加上其面向对象的设计思想，独特的接收缓存管理和线程池管理方式，成为 Android 进程间通信的中流砥柱。