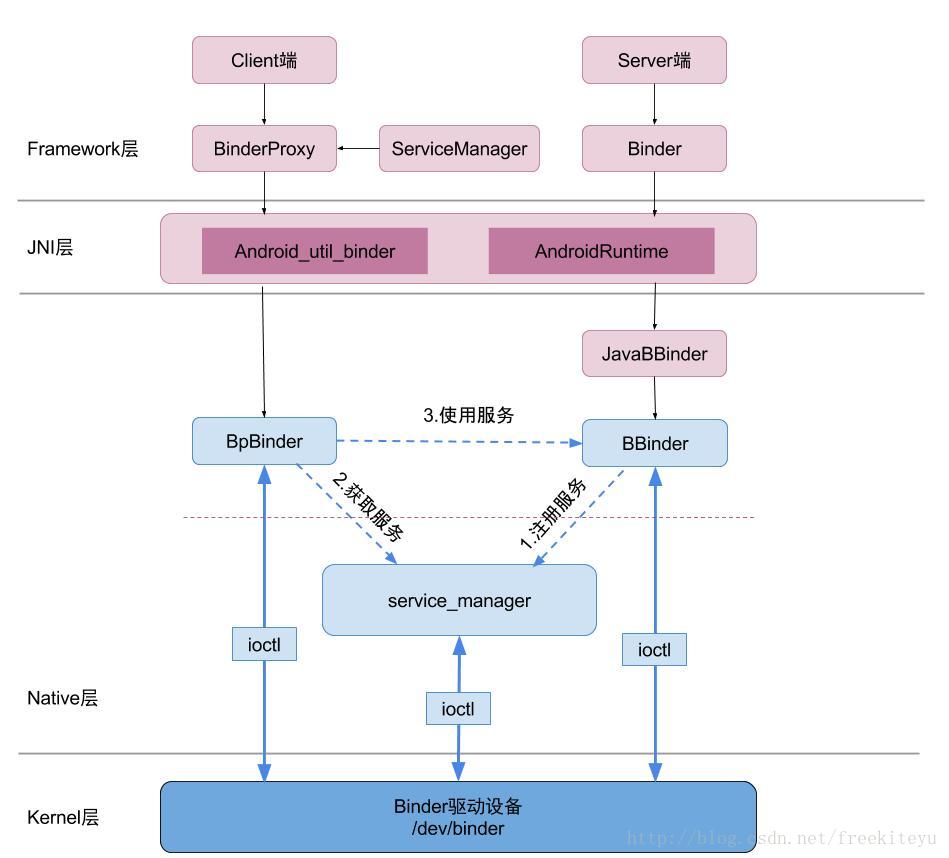
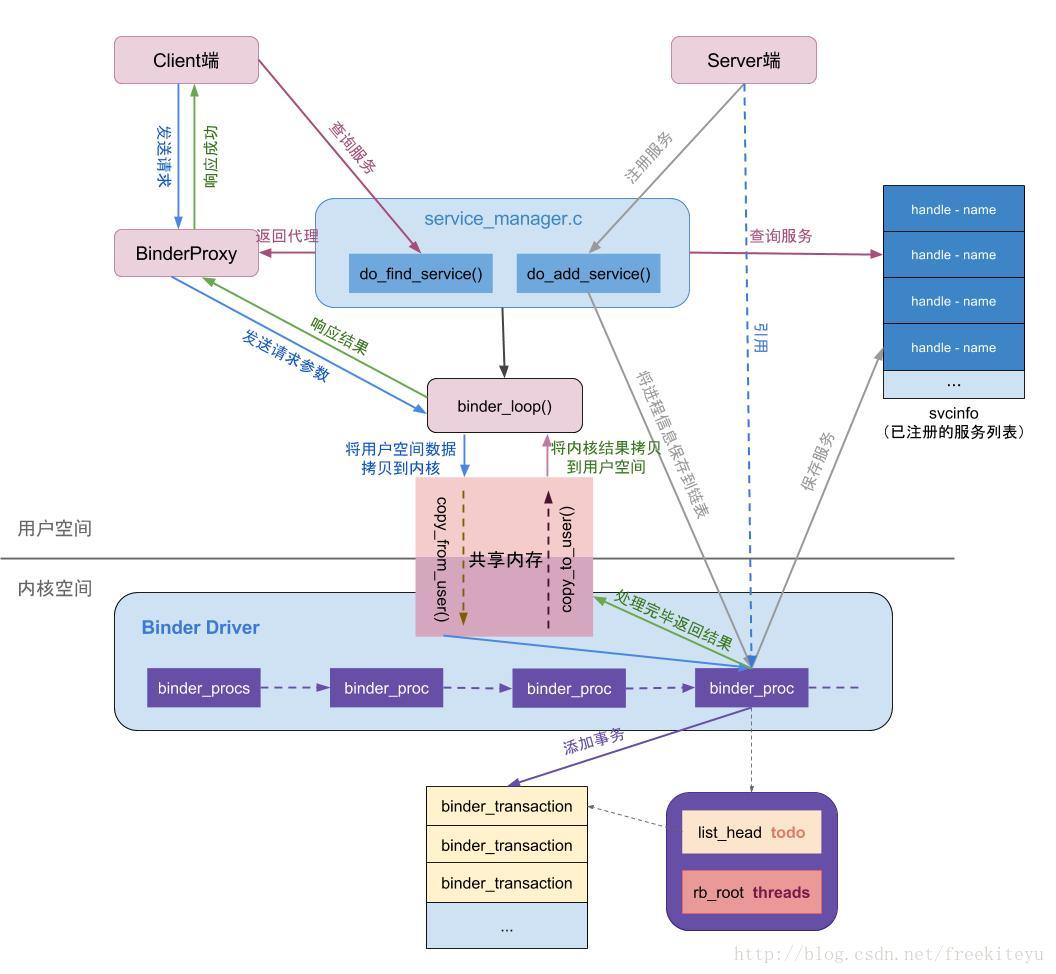
# Binder

**Binder架构**



**Binder机制**



首先需要注册服务端，只有注册了服务端，客户端才有通讯的目标，服务端通过 ServiceManager 注册服务，注册的过程就是向 Binder 驱动的全局链表 binder\_procs 中插入服务端的信息（binder\_proc 结构体，每个 binder\_proc 结构体中都有 todo 任务队列），然后向 ServiceManager 的 svcinfo 列表中缓存一下注册的服务。

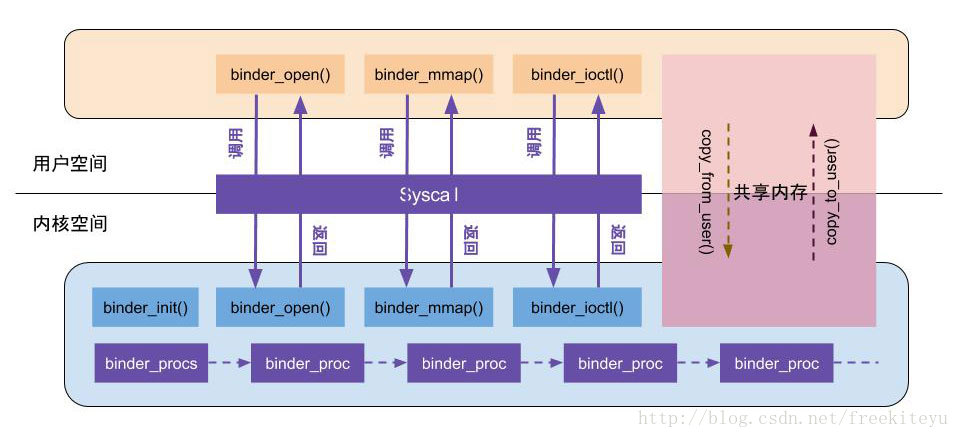
获取服务端的方式就是通过 ServiceManager 向 svcinfo 列表中查询一下返回服务端的代理，svcinfo 列表就是所有已注册服务的通讯录，保存了所有注册的服务信息。

有了服务端的引用我们就可以向服务端发送请求了，通过 BinderProxy 将我们的请求参数发送给 ServiceManager，通过共享内存的方式使用内核方法 copy\_from\_user() 将我们的参数先拷贝到内核空间，这时我们的客户端进入等待状态，然后 Binder 驱动向服务端的 todo 队列里面插入一条事务，执行完之后把执行结果通过 copy\_to\_user() 将内核的结果拷贝到用户空间（这里只是执行了拷贝命令，并没有拷贝数据，binder只进行一次拷贝），唤醒等待的客户端并把结果响应回来，这样就完成了一次通讯。

**Binder 驱动**

内核空间和用户空间是怎么交互的

Kernel space 是 Linux 内核的运行空间，User space 是用户程序的运行空间。 为了安全，它们是隔离的，即使用户的程序崩溃了，内核也不受影响。



Kernel space 是 Linux 内核的运行空间，可以执行任意命令，调用系统的一切资源。

User space 只能执行简单的运算，不能直接调用系统资源，必须通过系统接口（又称 system call），才能向内核发出指令。

用户空间访问内核空间的唯一方式就是系统调用；通过这个统一入口接口，所有的资源访问都是在内核的控制下执行，以免导致对用户程序对系统资源的越权访问，从而保障了系统的安全和稳定。

当一个任务（进程）执行系统调用而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）此时处理器处于特权级最高的（0级）内核代码中执行。当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。即此时处理器在特权级最低的（3级）用户代码中运行。处理器在特权等级高的时候才能执行那些特权CPU指令。

**内核模块/驱动**

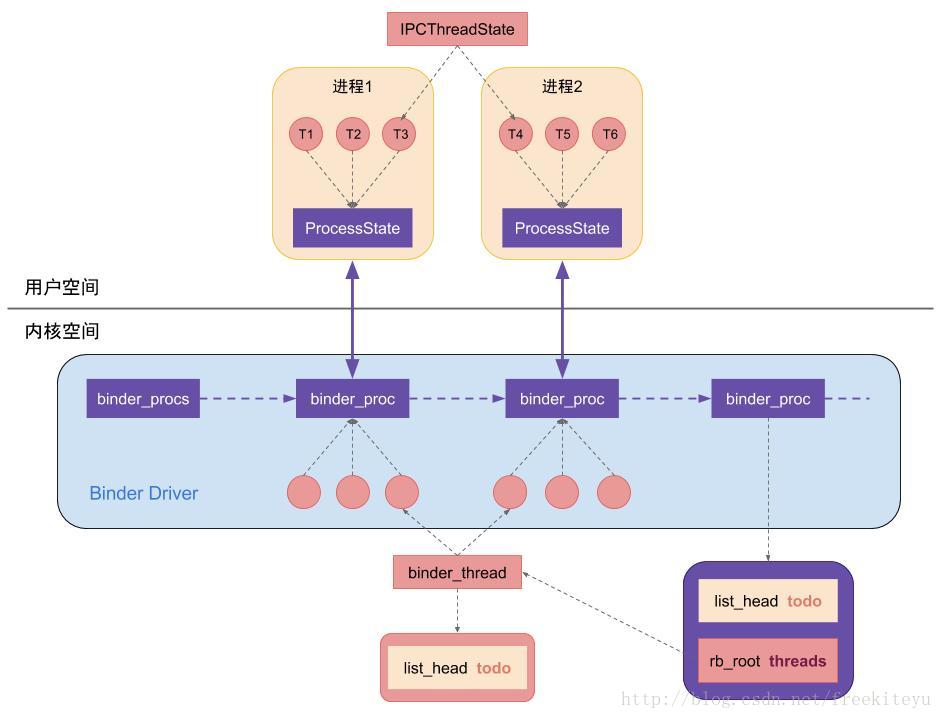
通过系统调用，用户空间可以访问内核空间，那么如果一个用户空间想与另外一个用户空间进行通信怎么办呢？很自然想到的是让操作系统内核添加支持；传统的 Linux 通信机制，比如 Socket，管道等都是内核支持的；但是 Binder 并不是 Linux 内核的一部分，它是怎么做到访问内核空间的呢？ Linux 的动态可加载内核模块（Loadable Kernel Module，LKM）机制解决了这个问题；模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间运行。这样，Android系统可以通过添加一个内核模块运行在内核空间，用户进程之间的通过这个模块作为桥梁，就可以完成通信了。

在 Android 系统中，这个运行在内核空间的，负责各个用户进程通过 Binder 通信的内核模块叫做 Binder 驱动;

驱动就是操作硬件的接口，为了支持Binder通信过程，Binder 使用了一种“硬件”，因此这个模块被称之为驱动。

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()。Binder驱动的代码位于linux目录的drivers/misc/binder.c中。

**Binder进程与线程**

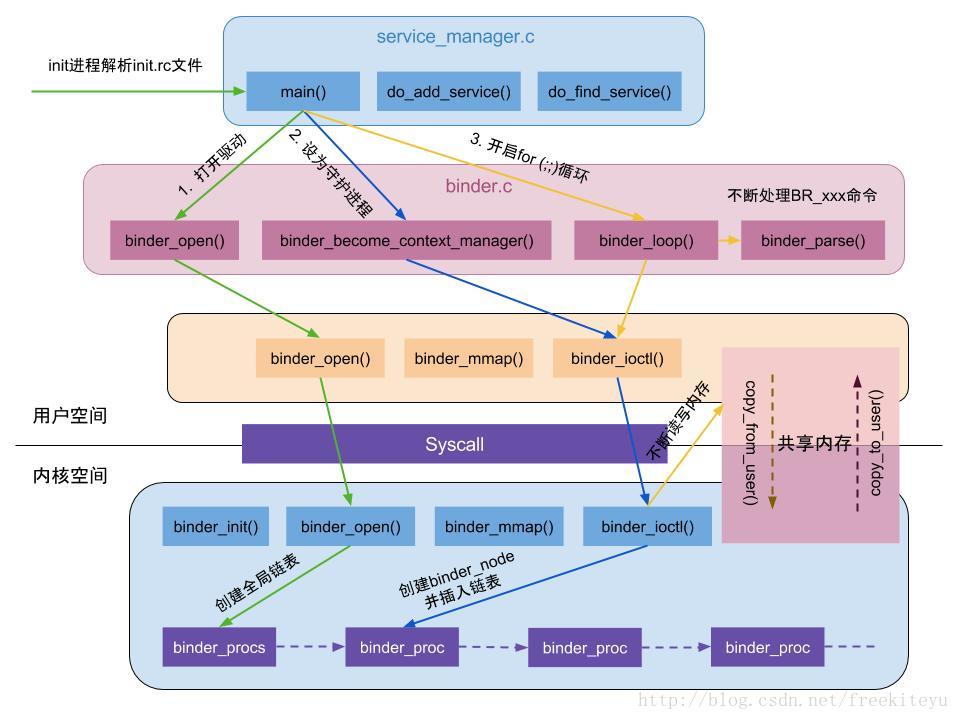


binder 驱动层的每一个 binder\_proc 结构体都与用户空间的一个用于 binder 通信的进程一一对应，且每个进程有且只有一个 ProcessState 对象，这是通过单例模式来保证的。在每个进程中可以有很多个线程，每个线程对应一个 IPCThreadState 对象，IPCThreadState 对象也是单例模式，即一个线程对应一个 IPCThreadState 对象，在 Binder 驱动层也有与之相对应的结构，那就是 Binder\_thread 结构体。在 binder\_proc 结构体中通过成员变量 rb\_root threads，来记录当前进程内所有的 binder\_thread。

Binder 线程池：每个 Server 进程在启动时创建一个 binder 线程池，并向其中注册一个 Binder 线程；之后 Server 进程也可以向 binder 线程池注册新的线程，或者 Binder 驱动在探测到没有空闲 binder 线程时主动向 Server 进程注册新的的 binder 线程。对于一个 Server 进程有一个最大 Binder 线程数限制，默认为16个 binder 线程，例如 Android 的 system\_server 进程就存在16个线程。对于所有 Client 端进程的 binder 请求都是交由 Server 端进程的 binder 线程来处理的。

**ServiceManager启动**

ServiceManager 的作用很简单就是提供了查询服务和注册服务的功能。下面我们来看一下 ServiceManager 启动的过程。



ServiceManager 的启动是系统在开机时，init 进程解析 init.rc 文件调用 service\_manager.c 中的 main() 方法入口启动的。 native 层有一个 binder.c 封装了一些与 Binder 驱动交互的方法。

ServiceManager 的启动分为三步，首先打开驱动创建全局链表 binder\_procs，然后将自己当前进程信息保存到 binder\_procs 链表，最后开启 loop 不断的处理共享内存中的数据，并处理 BR\_xxx 命令（ioctl 的命令，BR 可以理解为 binder reply 驱动处理完的响应）。

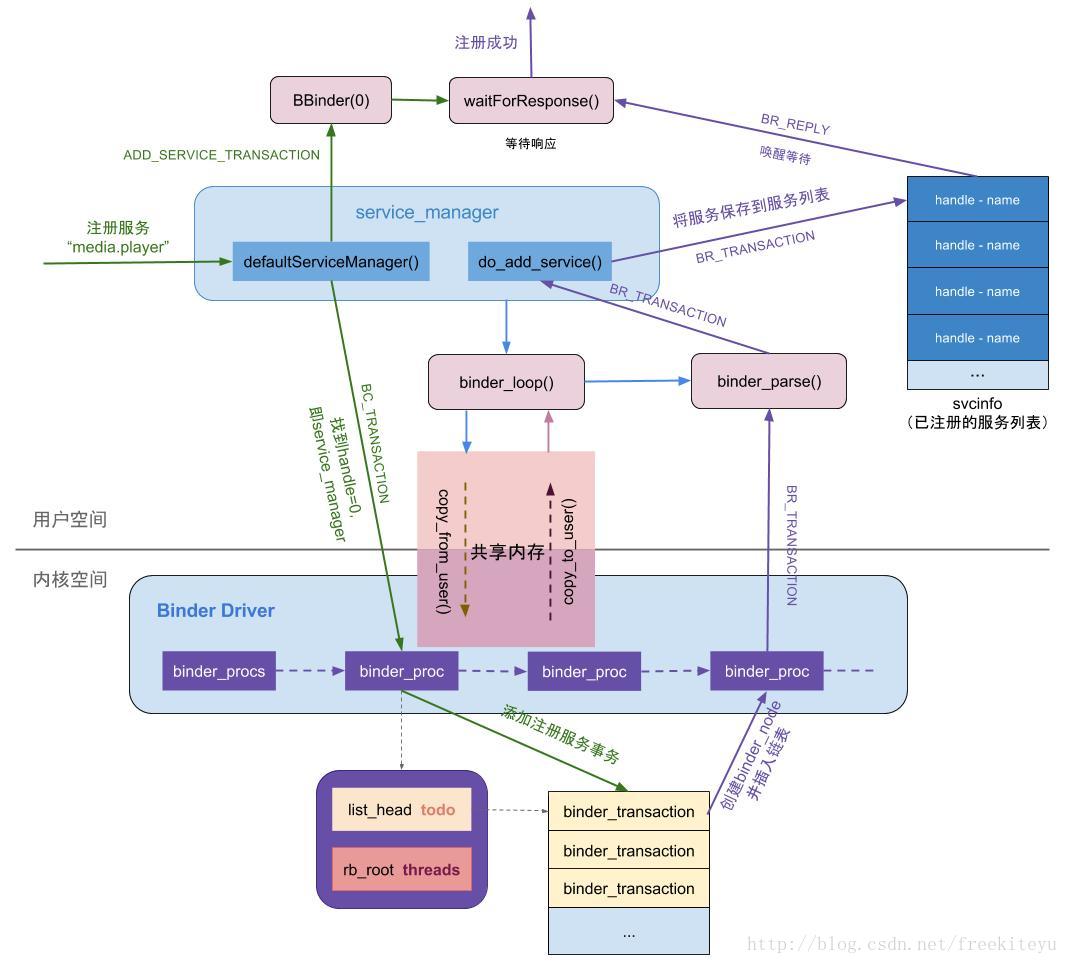
**ServiceManager注册服务**

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：SMgr和其它进程同样采用Binder通信，ServiceManager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。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（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ServiceManager注册自己Binder就必需通过0这个引用号和ServiceManager的Binder通信。

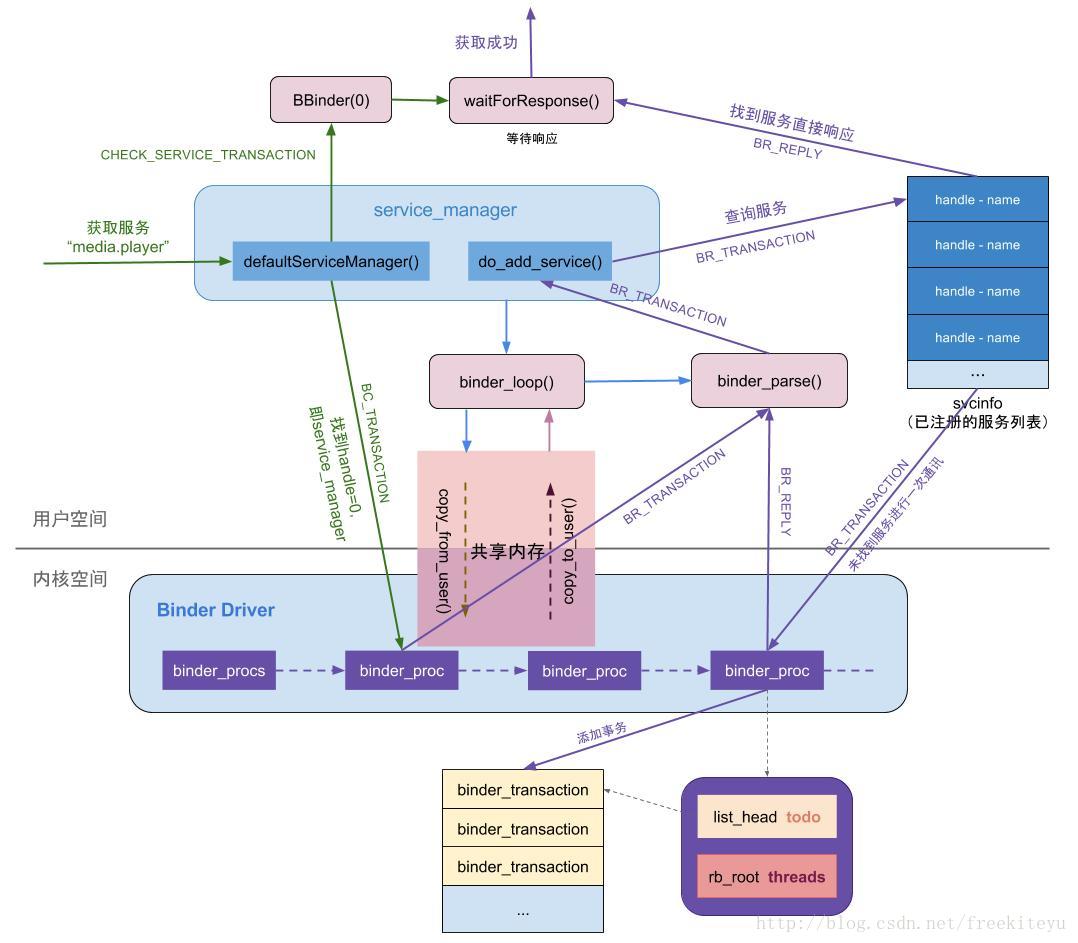
注册 MediaPlayerService 服务端，我们通过 ServiceManager 的 addService() 方法来注册服务。

首先 ServiceManager 向 Binder 驱动发送 BC\_TRANSACTION 命令（ioctl 的命令，BC 可以理解为 binder client 客户端发过来的请求命令）携带 ADD\_SERVICE\_TRANSACTION 命令，同时注册服务的线程进入等待状态 waitForResponse()。 Binder 驱动收到请求命令向 ServiceManager 的 todo 队列里面添加一条注册服务的事务。事务的任务就是创建服务端进程 binder\_node 信息并插入到 binder\_procs 链表中。

事务处理完之后发送 BR\_TRANSACTION 命令，ServiceManager 收到命令后向 svcinfo 列表中添加已经注册的服务。最后发送 BR\_REPLY 命令唤醒等待的线程，通知注册成功



**ServiceManager获取服务**

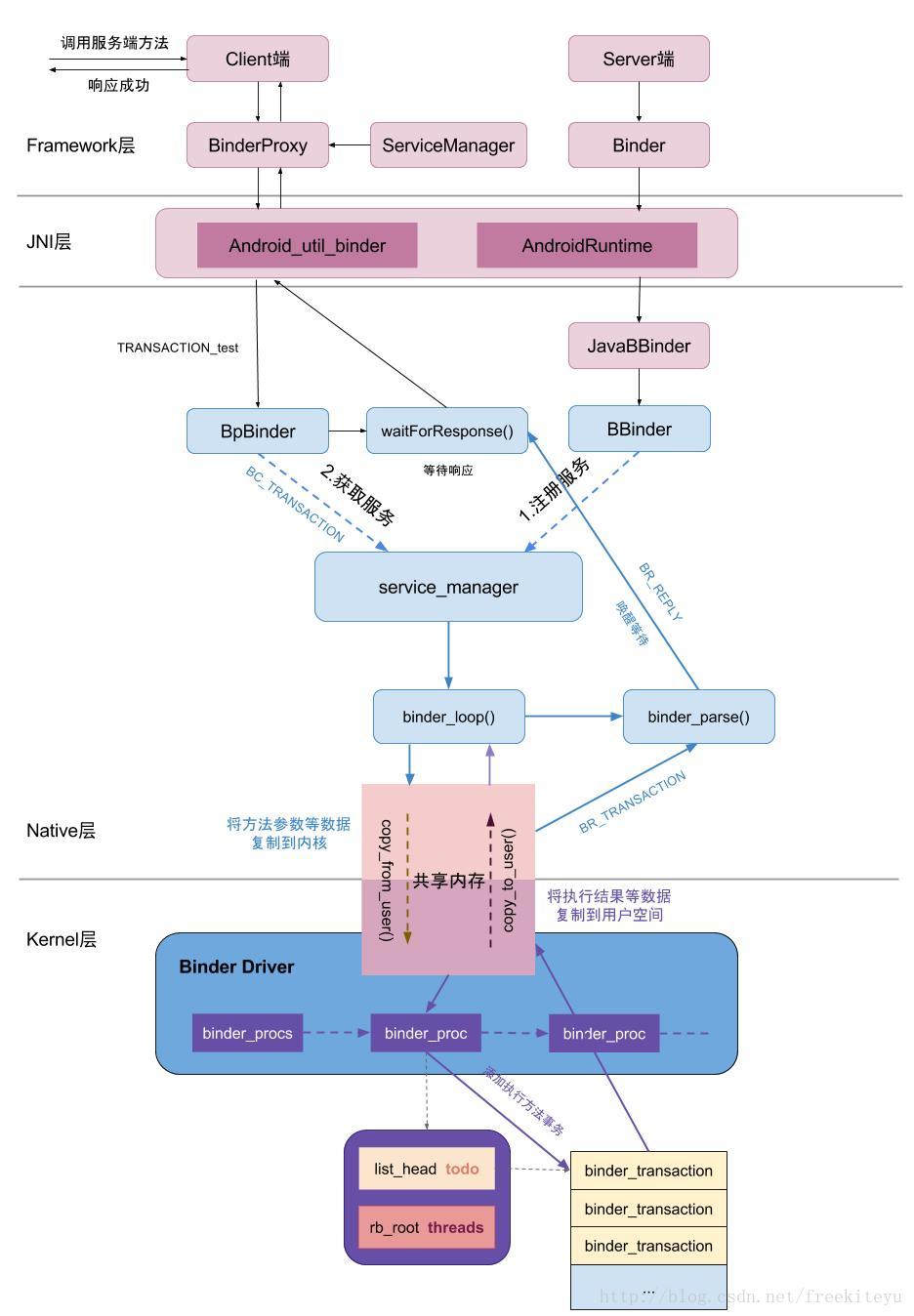


获取服务的过程与注册类似，相反的过程。通过 ServiceManager 的 getService() 方法来注册服务。

首先 ServiceManager 向 Binder 驱动发送 BC\_TRANSACTION 命令携带 CHECK\_SERVICE\_TRANSACTION 命令，同时获取服务的线程进入等待状态 waitForResponse()。

Binder 驱动收到请求命令向 ServiceManager 的发送 BC\_TRANSACTION 查询已注册的服务，查询到直接响应 BR\_REPLY 唤醒等待的线程。若查询不到将与 binder\_procs 链表中的服务进行一次通讯再响应。

**进行一次完整通信**



我们在使用 Binder 时基本都是调用 framework 层封装好的方法，AIDL 就是 framework 层提供的傻瓜式是使用方式。假设服务已经注册完，我们来看看客户端怎么执行服务端的方法。

首先我们通过 ServiceManager 获取到服务端的 BinderProxy 代理对象，通过调用 BinderProxy 将参数，方法标识（例如：TRANSACTION\_test，AIDL中自动生成）传给 ServiceManager，同时客户端线程进入等待状态。

ServiceManager 将用户空间的参数等请求数据复制到内核空间，并向服务端插入一条执行执行方法的事务。事务执行完通知 ServiceManager 将执行结果从内核空间复制到用户空间，并唤醒等待的线程，响应结果，通讯结束。

**Binder优势**

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。

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。

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。

**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。

**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。

|  |  |  |
| --- | --- | --- |
| **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。

**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：

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 |
| 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 |

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。

**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：

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 | --- |
| 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。 |
| 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）。

**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：

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。 |
| 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.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。 |