Android Binder From Deep

Leo shecenon@gmail.com

2012-06-02

目录

1 Binder 基本架构

1.1 Binder 通信模型

Binder 框架定义了四个角色：客户端，服务端，ServiceManager 以及 Binder 驱动。其中客户端，服务端，ServiceManager 运行于用户空间，驱动运行于内核空间。ServiceManager 相当于域名服务器，驱动相当于路由器。

1.2 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 1: Binder 协议

BINDER_WRITE_READ	该命令向 Binder 写入或读取数据。参数见代码 1.2.1。
BINDER_SET_MAX_THREADS	设置进程的最大生成线程 max_threads, 参见子节 4.3
BINDER_SET_CONTEXT_MGR	设置当前进程为 service manager, 用全局变量记录下来。[a]
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(包括服务端线程池中的线程和客户端发出请求的线程)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。[b]
BINDER_VERSION	获取 binder 协议版本。

[a] binder 是一个服务和客户通讯的协议, 客户为了能跟服务搭上线, 需要有个地方能查找服务的实际所在, 只有在获取服务的实际所在才能提出后面的各种要求, 而为客户牵线的便是这个 service manager 进程。系统中同时只能存在一个 service manager。只要当前的 service manager 没有调用 close() 关闭 Binder 驱动就不能有别的进程可以成为 service manager。

[b] 当前线程退出 binder 驱动, 清理其在 binder_get_thread 中创建的节点, 同时会给那些跟他 transaction 的线程发个 BR_DEAD_REPLY 表明自己挂了。

1.2.1 Binder 读写命令

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。

```
1  struct binder_write_read {
2          signed long     write_size;     /* bytes to write */
3          signed long     write_consumed; /* bytes consumed by driver */
4          unsigned long   write_buffer;
5          signed long     read_size;      /* bytes to read */
6          signed long     read_consumed;  /* bytes consumed by driver */
7          unsigned long   read_buffer;
8  };
```

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_buffer7 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。

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 2: Binder 操作命令

BC_TRANSACTION BC_REPLY	用于客户端向服务端发送请求数据子节 3.1 用于服务端向客户端发送回复数据。
BC_FREE_BUFFER	释放一块映射的内存。子节 4.6
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。数据是 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 [a]
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 子节 4.3。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。[b]
BC_ACQUIRE_RESULT BC_ATTEMPT_ACQUIRE	暂未实现

[a] 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

[b] 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。

1.2.2 Binder 的返回值

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 3: Binder 返回值协议

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。参见代码 3.1
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	用于管理强/弱指针的引用计数。本地 Binder 的进程才能收到这组消息。
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。
BR_FAILED_REPLY	如果发送非法引用号则返回该消息.

2 Binder Library

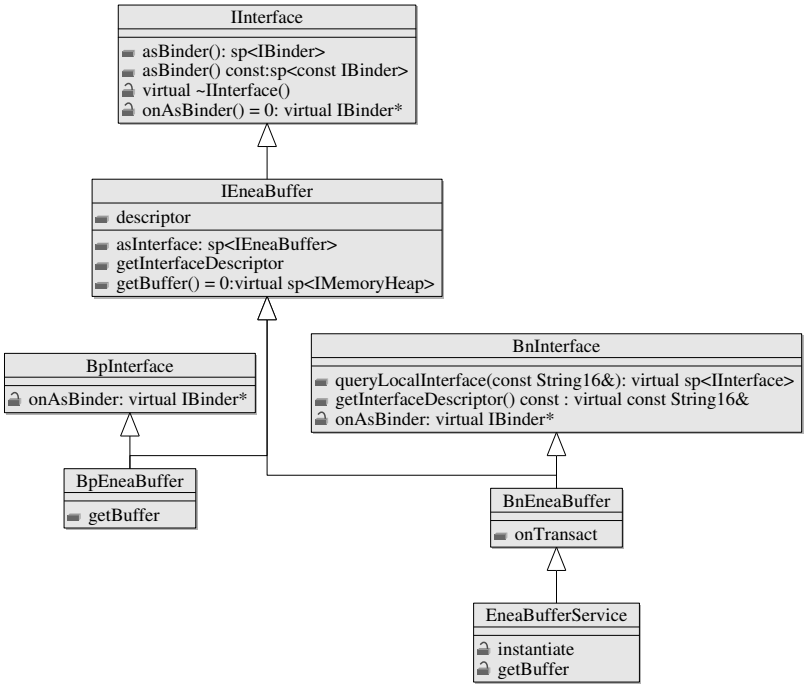
Binder 本质上只是一种底层通信方式,和具体服务没有关系。为了提供具体服务,服务端必须提供一套接口函数以便客户端通过远程访问使用各种服务。这时通常采用 Proxy 设计模式:将接口函数定义在一个抽象类中,服务端和客户端都会以该抽象类为基类实现所有接口函数,所不同的是服务端是真正的功能实现,而客户端是对这些函数远程调用请求的包装。

我们要明白两件事:一是我们用户如何使用,使用 Binder 的架构; 二是本地 Binder 和远程 Binder 是如何通讯。这都是和我们具体工作密切相关的问题,明白的这些基本的问题,我们阅读代码会更流畅,书写的代码更有保证。

2.1 实例:用户看到的 Binder 和接口

我们以接口 IEneaBuffer 为例,代码在附录节 A,

表 4: IEneaBuffer 类图



BnInterface/BpInterface 都是模板类[1], 注意这两个模板类的声明：

```
1   template<typename INTERFACE>
2   class BnInterface : public INTERFACE, public BBinder {
      :
3     :
4   }
5
6   template<typename INTERFACE>
7   class BpInterface : public INTERFACE, public BpRefBase {
      :
8     :
9   }
```

以类 IEneaBuffer 实例化模板类：

class BnEneaBuffer : public BnInterface<IEneaBuffer>
class BpEneaBuffer : public BpInterface<IEneaBuffer>

这种继承就说明了 BnEneaBuffer 和 BpEneaBuffer 都继承了 IEneaBuffer。

IEneaBuffer 定义的方法, 包括独有的成员函数[2] getBuffer, 还有 asInterface 、asBinder、getInterfaceDescriptor 共有成员函数[3]。共有成员函数对于 IEneaBuffer 的派生类而言都是一样的实现, 所以在 IEneaBuffer 类实现了共有成员[4], 不劳烦派生类实现或重载。而独有的成员函数需要派生

[1] Bn 意味着 Native, 代表 Binder 的本地端, 即服务端；Bp 是 Proxy, 表示客户端的 Binder 引用, 代理 Binder。

[2] 独有、共有, 是指不同接口之间比较的。或者说是业务成员函数, 支撑/基础成员函数

[3] 调用 DECLARE_META_INTERFACE 宏就声明了

[4] IMPLEMENT_META_INTERFACE 宏就是干这件事的

类分别实现。

BnEneaBuffer 实现了 onTransact 函数，onTransact 实现里有对 IEneaBuffer 独有成员函数（即 getBuffer）的调用，但 BnEneaBuffer 并未实现她。这是 Android 常见的反向控制。而 EneaBufferService 继承了 BnEneaBuffer，并实现了 IEneaBuffer 接口的所有函数。EneaBufferService 是 IEneaBuffer 接口的具体类。

所以，BnEneaBuffer 是不能实例化的，EneaBufferService 才能实例化。也正因为这个原因，IEneaBuffer，BnEneaBuffer 一起定义在头文件里，而 BpEneaBuffer、EneaBufferService 就直接在源文件里声明、实现了。

BpEneaBuffer 就比较简单，实现了 IEneaBuffer 的独有接口，因此她是个具体类，可以实例化的，但我们没有看到 BpEneaBuffer 使用的影子，把其注释调后发现，IEneaBuffer::asInterface子节 2.3使用了它。

注意到 IInterface，IBinder 的纯析构函数，有关纯析构函数的讨论请参见 节 B。

2.2 分身术：本地 BBinder 与远程 BpBinder

深入分析接口时，必然涉及 Binder，讲 Binder 首先明确本地 Binder 和远程 Binder。本地 Binder 就是 BBinder、BnInterface 派生类，例子中的就是 IEneaBuffer、BnEneaBuffer 派生类 EneaBufferService 实例，而远程 Binder 就是 BpBinder 类的实例。这里暗示，BpBinder 类可以实例化对象，而 BBinder 是抽象类，不能实例化的。

IBinder[9] 实现了成员函数 localBinder 和 remoteBinder，都返回 NULL。而派生类 BBinder 仅仅重载了 localBinder，返回自身的 this。派生类 BpBinder 仅仅重载了 remoteBinder，也是返回了自身的 this。而 localBinder 返回的类型是 BBinder*，remoteBinder 返回的类型是 BpBinder*。从中可见一斑，BBinder 及其派生类是本地 Binder，BpBinder 是远程 Binder。

IBinder 也实现了 queryLocalInterface 成员函数，也是返回了 NULL，此函数的返回类型是 IInterface。BpBinder 并没有重载此函数，所以此类的此成员返回 NULL。而 BBinder 虽然没有重载此函数，但是其派生类 BnInterface 重载了她，返回的是 this，BBinder 的派生具体类如 EneaBufferService 没有重载，那 BBinder 对象的此函数必然返回 this。

代码 1: queryLocalInterface 的实现

```
1  /*frameworks/base/include/binder/IInterface.h*/
2  template<typename INTERFACE>
3  inline sp<IInterface> BnInterface<INTERFACE>::queryLocalInterface(
4          const String16& _descriptor)
5  {
6      if (_descriptor == INTERFACE::descriptor) return this;
7      return NULL;
8  }
9
10 /*frameworks/base/libs/binder/Binder.cpp*/
11 sp<IInterface>  IBinder::queryLocalInterface(const String16& descriptor)
```

```
12  {
13      return NULL;
14  }
```

BpBinder 对象的 localInterface、localBinder 均返回了 NULL。而 BBinder 对象 (即类 EneaBufferService 的实例) 的 localInterface、localBinder 均返回了自身 this。这显示接口的双重属性是 IBinder，也是 IInterface。

2.3 转换：传输和使用

这两个属性之间转换，IBinder::asBinder 是把 IEneaBuffer 具体的派生类转换成 IBinder 类，以便 IPC 传输；IEneaBuffer::asInterface 是把 IBinder 派生类转换为 IEneaBuffer，以便使用对象的功能。

asInterface成员函数是 IEneaBuffer 实现了，参数是 IBinder 对象。他现调用参数的函数 queryLocalInterface，本地 Binder(如 BnInterface 派生类 EneaBufferService)，得到 EneaBufferService 对象的 this，此时 asInterface 返回这个 this，即 BnEneaBuffer 对象。远程 Binder (BpBinder 的实例)，返回 NULL，asInterface 实例化 BpInterface 派生类的对象(如 BpEneaBuffer)并返回。interface_cast 是对 asInterface 的保证，利用了 C++ 语言的特性，方便使用。

asBinder 是 IIterface 直接实现了(IIterface.cpp)，就是调用 onAsBinder，派生类不重载 asBinder，而是实现 onAsBinder，这也是反向控制。BnInterface 的 onAsBinder 实现就是返回 this 。BpInterface 的 onAsBinder 实现就是调用方法 remote()，这个 BpRefBase 方法返回常指针成员 mRemote。mRemote 是 BpEneaBuffer 构造函数的参数经 BpInterface 传给 BpRefBase 保存的，BpEneaBuffer 实例化就在 asInterface，所以 BpEneaBuffer 的实例化参数是个远程 Binder 即 BpBinder。

代码 2: BpInterface 的构造函数

```
1   /*frameworks/base/include/binder/IInterface.h*/
2   template<typename INTERFACE>
3   inline BpInterface<INTERFACE>::BpInterface(const sp<IBinder>& remote)
4       : BpRefBase(remote)
5   {
6   }
7
8   BpRefBase::BpRefBase(const sp<IBinder>& o)
9       : mRemote(o.get()), mRefs(NULL), mState(0)
10  {
11      extendObjectLifetime(OBJECT_LIFETIME_WEAK);
12
13      if (mRemote) {
14          mRemote->incStrong(this);           // Removed on first IncStrong().  incStrong
15          mRefs = mRemote->createWeak(this);  // Held for our entire lifetime.
16      }
17  }
```

小结一下，本地 binder 转换是很方便，她具备双重属性，BnEneaBuffer 既继承了 IBinder，又继承了 IEneaBuffer，他在这两种转换之间都是平滑的，直接返回自己的 this 即可。从远程 binder 转换得到 IEneaBuffer 对象

是新建 BpEneaBuffer 对象, 从 IEneaBuffer 对象得到的远程 Binder 对象是 BpEneaBuffer 父类 BpRefBase 保存的 BpBinder 对象。

2.4 如何创建

但是, 从客户角度来讲, 他所直接接触的并不是本地 binder 还是远程 binder 而是本地接口和远程接口。我们需要知道本地接口和远程接口是如何接上头的。

从远程 binder 的构造函数, 我们可以知道, 远程 binder 和本地 binder 建立联系的纽带是所谓的 handle, 这是 32 位的数值, 其中 0 是有特殊含义, 表示 binder manager 的, 我们定义的 binder 是从 1 开始编号的。

服务进程启动后, 实例化 EneaBufferService 对象并把它加到 Service Manager, 就真在提供服务。

```
1  defaultServiceManager()->addService(String16("vendor.enea.Buffer"),
2      new EneaBufferService());
3  ProcessState::self()->startThreadPool();
4  IPCThreadState::self()->joinThreadPool();
```

这是本地 Binder 的实例化的地方了, 一般情况下, 也是唯一的一个地方。当然, 这也是本地接口实例化的地方。好像在 addService 时, service manager 创建了一个 BpBidner？

远程 Binder 的实例化在 ProcessState 的 getStrongProxyForHandle 、和 getWeakProxyForHandle。具体来讲, Parcel 类 acquire_object、release_object、unflatten_binder 函数在处理 BINDER_TYPE_HANDLE 类型的 binder 时会调用 getStrongProxyForHandle, 处理 BINDER_TYPE_WEAK_HANDLE 类型时调用 getWeakProxyForHandle 等等来实例化本地 binder。

参见 2.5, 再用这个 binder 调用接口 asInterface 来获取接口, 对于本地 binder 来说就是返回自身, 而对于远程 binder, 如 子节 2.3 所述, asInterface 是本地接口(如 BpEneaBuffer)唯一的实例化地方。

但是本地接口话的时候, 不仅仅是靠 binder 句柄, 里面还有文章的, 跟驱动引用计数有关。回顾一下 代码 2, BpRefBase mRemote->incStrong(this); 实际调用的流程 :RefBase::incStrong -> BpBinder::onFirstRef -> IPCThreadState::incStrongHandle 发出了一个 BC_ACQUIRE 的命令

2.5 如何使用接口

Binder 是 IPC, 所以典型的情况是一个服务进程, 一个客户进程, 背后隐藏了的 Service Manager。

其实不管服务进程, 还是客户进程, 使用接口的对象, 首先要获得 IServiceManager 对象, 再获得服务, 即接口的 Binder 对象, 然后转型为接口, 方可使用, 相比之下多了三步。

```
1  static sp<IEneaBuffer> eneaBuffer = NULL;
2
3  sp<IServiceManager> sm = defaultServiceManager();
4  sp<IBinder> binder = sm->getService(String16("vendor.enea.Buffer"));
5  if (binder != 0) {
6      eneaBuffer = IEneaBuffer::asInterface(binder);
7  }
8
9  if (eneaBuffer == NULL) {
10     receiverMemBase = eneaBuffer->getBuffer();
11 }
```

客户还是服务进程,都的首先创建一个 BpServiceManager,因为他们都得和 servicemanager 通讯。客户进程和服务进程,都要调用 defaultServiceManager 就必须实例化一个 BpBinder 对象,这个对象是 Service Manager 的远程 Binder,而且 handle 号是 0。过程是：defaultServiceManager 调用 ProcessState::getContexObject,后者以参数 0 调用 ProcessState::getStrongProxyForHandle。

服务进程并没有保存 EneaBufferService 对象,这个对象保存在了 Service Manager 里,所以服务进程也不直接调用接口的各种方法,它也是个" 客户端"。但是服务进程和客户进程获得的 Binder 对象还是不一样的,前者是本地 Binder,而后者是远程 Binder。也就是说客户与服务进程在调用 getService 返回的 Binder 对象是不同的。

Service manager 有必要澄清一下. Binder 库定义了 IServiceManager、BpServiceManager、BnServiceManager。但是 BnServiceManager 派生类 ServiceManager [13] 只被程序 runtime 使用,这个程序只在模拟器上使用。真机上,服务管理器的程序叫 servicemanager,代码在 [12] 里。服务进程或客户进程调用了 IServiceManager 的接口后,经 BpServiceManager 代理入驱动后,最后传给了 servicemanager。

需要注意是 servicemanager 里未采用 IServiceManager 里成员的编码宏名 我们查看 service_manager.c 的函数 find_svc,发现 getService

IServiceManager 定义的宏名	servicemanager 对应的宏名
GET_SERVICE_TRANSACTION	SVC_MGR_GET_SERVICE
CHECK_SERVICE_TRANSACTION	SVC_MGR_CHECK_SERVICE
ADD_SERVICE_TRANSACTION	SVC_MGR_ADD_SERVICE
LIST_SERVICES_TRANSACTION	SVC_MGR_LIST_SERVICES

根本不区分本地 binder 还是远程 binder。回头看 BpServiceManager,检查 checkService,定位可疑点是 reply.readStrongBinder(),即 Parcel 类的 readStrongBinder 函数,进一步是 unflatten_binder 。unflatten_binder 显然是对本地 binder 和远程 binder 做了区分的。如同 flatten_binder 对本地 binder 和远程 binder 作了区分的。

写入流程：writeStrongBinder flatten_binder finish_flatten_binder writeObject 读取流程：readStrongBinder unflatten_binder readObject

另外,当服务未启动时,getService 会等待,输出日志"Waiting for service xxx" ,系统初始化 SurfaceFlinger 会出现这种情况。

2.6 ProcessState

代码 2.4 Binder 库中有两个状态的类, 其含义可以从其名字可知：ProcessState 是表示进程状态的库类 IPCThreadState 是处理线程状态及线程池的类。服务进程要启动线程池, 客户进程一般不需要的。调用

ProcessState::self()->startThreadPool();

就可以启动线程池了。ProcessState 有个类 PoolThread , 是 Thread 派生类, 其主要是实现了 threaLoop(), 这也是 Thread 派生类必做的, 因为这是 Thread 的纯虚函数。PoolThread::threaLoop 就是调用了

IPCThreadState::self()->joinThreadPool();

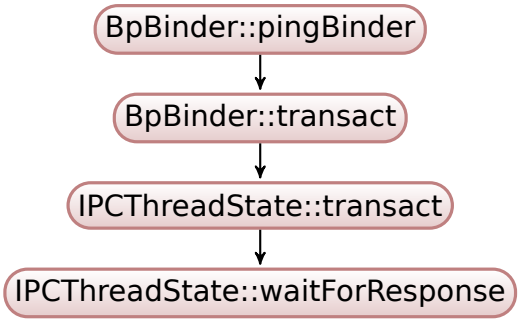
startThreadPool 调用 spawnPooledThread 实例化 PoolThread, 并调用 run 函数运行这个线程(run 的三个参数都有默认值的), 这个线程设为主线程。Thread::run 函数会调用 Thread::_threadLoop , 这是线程的入口函数, 参数就是 Thread 派生类的 this 指针, _threadLoop 会调用派生类的 threadLoop 函数。ProcessState 没有保留 PoolThread 的对象, 其引用不会为 0. 所以, _threadLoop 的循环不会自动退出。

服务进程在进程池里有两个 Binder 线程, 其中一个 ProcessState::startThreadPool 调用 ProcessState::spawnPooledThread 启动的, 这个是主线程。主线程的的 threadLoop 会进入 IPCThreadState::joinThreadPool, 发了 BC_ENTER_LOOPER, 这是驱动会返回 BR_SPAWN_LOOPER, 让用户进程创建一个工作进程, 此工作进程调用 IPCThreadState::joinThreadPool 向驱动发 BC_REGISTER_LOOPER 注册自己, binder 线程是向驱动发 BC_ENTER_LOOPER /BC_REGISTER_LOOPER 注册的, 而一般的进程是不会这么做的。

spawnPooledThread 是生出的线程的, 可以使用命令 ps -t 命令显示的, Binder Thread #1 的线程就是她的杰作。客户端是否也使用了 ProcessState 及 IPCThreadState？IPCThreadState 在客户进程还是用到了, 但是并未看到线程池的支持。因为客户端使用了 BpEneaBuffer, 其父类 BpRefBase 保存的 BpBinder 类的实现中使用了 IPCThreadState 的函数, 如 BpBinder::onFirstRef, BpBinder::onLastStrongRef, BpBinder::transact。

我们关注的是 IPCThreadState::joinThreadPool 的循环. joinThreadPool 首先调用 mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER); 第一次循环, 下列条件不成立 :mIn.dataPosition() >= mIn.dataSize() 其下代码块跳过之后, 就是 result = talkWithDriver(); 如执行无误, 从 mIn 中取出命令(命令的是 int32_t, 取命令前先判断 mIn 的数据长度是否够装命令), 交给 executeCommand 执行。检查了 talkWithDriver 和 executeCommand 没有发现等待, 如 sleep,wait 等命令。这个看似是简单的 for 循环, 其实并不是的, 服务器不会做简单的循环, 奥秘在 talkWithDriver 中的 ioctl, 这是个阻塞调用, joinThreadPool 在没有事情做的时候, 就在这个 ioctl 里等待, 直到有客户请求的到来。

客户的请求流程是：



waitForResponse 也是调用了 talkWithDriver(以接收方式, 即参数为 true), 他也以 BINDER_WRITE_READ 方式调用 ioctl, 此时发生了奇妙的事, 客户进程被阻塞了, 而先前在等待的服务进程激活了, 即发生所谓的" 进程迁移", [5] 服务进程, 就进入了, executeCommand, 对应上述的命令是 BR_TRANSACTION, 那么服务进程, executeCommand 调用本地 binder 的函数 transact, 后者调用 BBinder::pingBinder, 后调用 sendReply, 而 sendReply 又会调用 waitForResponse , 类似开始的客户进程, 服务进程被阻塞, 客户进程被疏通。

要注意的是, talkWithDriver 在 joinThreadPool 是处理了顶层的返回命令, 而在 waitForResponse 里处理是底下的返回命令。waitForResponse 在一个无限循环里调用 talkWithDriver , 事实上在处理一个客户请求中, waitForResponse 要多次调用 talkWithDriver 才能处理完成这个客户请求。返回命令也可以理解为中间命令吧！

假如线程退出循环, 发命令告诉驱动：mOut.writeInt32(BC_EXIT_LOOPER); talkWithDriver(false);

IPCThreadState::joinThreadPool 对于主线程有所区别的

mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER); 子节 4.4

IPCThreadState::executeCommand BR_SPAWN_LOOPER

case BR_SPAWN_LOOPER: mProcess->spawnPooledThread(false); break;

2.7 Binder 库的一些其他讨论

Binder 自身默认的几个操作, 定义在 IBinder.h 里：

```
1  FIRST_CALL_TRANSACTION  = 0x00000001,
2  LAST_CALL_TRANSACTION   = 0x00ffffff,
3
4  PING_TRANSACTION        = B_PACK_CHARS('_','P','N','G'),
5  DUMP_TRANSACTION        = B_PACK_CHARS('_','D','M','P'),
6  INTERFACE_TRANSACTION   = B_PACK_CHARS('_', 'N', 'T', 'F'),
7
8
9
```

[5] 别的 binder子节 1.2 命令是不会发生进程迁移的吧？

```
10  status_t talkWithDriver(bool doReceive=true);
11
12  result = talkWithDriver(); 等价于
13  result = talkWithDriver(true); 表示从  // Binder 读取数据
14
15   result = talkWithDriver(false); 表示往// Binder 里写数据
16  void IPCThreadState::flushCommands() 就是对其包装,往  Binder 里写数据
```

Binder.cpp 类 BBinder,BBinder::transact 是处理入口,并处理了 PING_-TRANSACTION BBinder::onTransact 处理了 DUMP_TRANSACTION , INTERFACE_TRANSACTION DUMP_TRANSACTION 在 dumpsys 里用到的(framework/base/cmd)

3 库和驱动的交互

然这要对 binder 有一定了解之后,改变一些写法。上面的是代码静态,静态分析,跟着代码走,比较累哦。动态分析,让代码按我们的思路来走。

定义在 binder.h binder_transaction_data、flat_binder_object 是驱动跟用户态的 c++ binder 库交互的结构体

3.1 binder_transaction_data

在命令中,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。而在回复中,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)。

```
1   struct binder_transaction_data {
2           /* The first two are only used for bcTRANSACTION and brTRANSACTION,
3            * identifying the target and contents of the transaction.
4            */
5           union {
6                   size_t  handle; /* target descriptor of command transaction */
7                   void    *ptr;   /* target descriptor of return transaction */
8           } target;
9           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  };
```

- target
  对于数据发送方，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 引用，存放在 target.handle 中。当数据包到达接收方时，驱动已将 Binder 对象的指针填入 target.ptr。
- cookie
  发送方忽略该成员；接收方收到数据包时，该成员存放的是驱动中与 Binder 指针相关的额外信息。
- code
  接口独有成员函数的编号。
- flags
  交互标志。TF_ONE_WAY 位为 1 则为异步交互，客户端发送完请求交互即结束，服务端不再返回 BC_REPLY 数据包；否则服务端会返回 BC_REPLY 数据包，客户端必须等待接收完该数据包方才完成一次交互。TF_ACCEPT_FDS 位是出于安全考虑，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 可以将其置位。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过多文件。
- sender_pid
  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。
- 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。
- offsets_size
  多个 Binder 在数据中传递，需要标出每个在 data.buffer 里偏移，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。
- 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 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。一个本地 Binder 可以发送给其它进程从而建立许多跨进程的引用 (即远程 Binder)；另外这些引用也可以在进程之间传递。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。

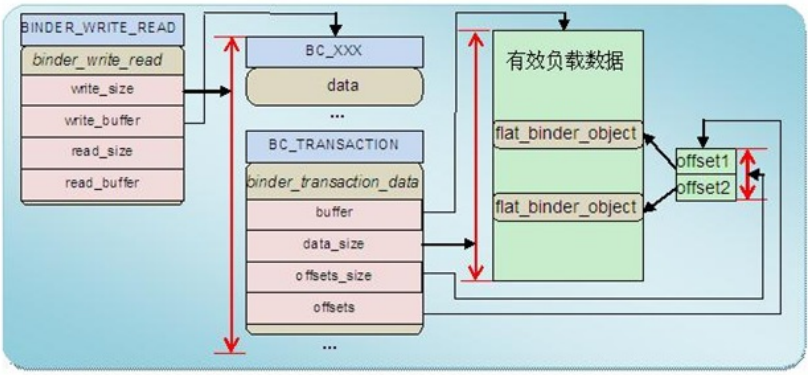


表 5: BINDER_WRITE_READ 数据包结构

3.2 flat_binder_object

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表格 5表示，如下表所示：

代码 3: flat_binder_object

```
1  /*
2   * This is the flattened representation of a Binder object for transfer
3   * between processes.  The 'offsets' supplied as part of a binder transaction
4   * contains offsets into the data where these structures occur.  The Binder
5   * driver takes care of re-writing the structure type and data as it moves
6   * between processes.
7   */
8  struct flat_binder_object {
9          /* 8 bytes for large_flat_header. */
10         unsigned long           type;
11         unsigned long           flags;
12
13         /* 8 bytes of data. */
14         union {
15                 void            *binder;        /* local object */
16                 signed long     handle;         /* remote object */
17         };
18
19         /* extra data associated with local object */
20         void                    *cookie;
21 };
```

type

type 表示 binder 的类型, 一共 5 种：BINDER_TYPE_BINDER 和 BINDER_TYPE_WEAK_BINDER 都是本地 binder , 对应于 binder_node, BINDER_TYPE_HANDLE 和 BINDER_TYPE_WEAK_HANDLE 都是远程 binder , 对应于 binder_desc 。BINDER_TYPE_FD 表示文件描述符。

binder/handle

联合体 binder/handle 就对应于 binder_node 的 ptr 或者 binder_desc 的 desc。传递本地 Binder 时使用 binder 域, 指向 Binder 在服务进程中的地址。传递远程 Binder 时使用 handle 域, 存放 Binder 的引用号。

cookie

cookie 在 BINDER 类型时就是 binder_node 的 cookie, 存放与该 Binder 有关的附加信息。在 HANDLE 时是 null, 没啥用。

flags

7~0 bits 设置 binder_node 线程优先级, 远程进程的对 binder 的操作请求会由binder 本地进程的一个线程完成, 而这个优先级就是用来设置该线程处理这个binder 事物时的优先级。

8th bit 是用来表示这个 binder 是否接受文件描述符的, 不支持文件描述符的 binder 是没法用 BINDER_TYPE_FD 的。

俗话说一个巴掌拍不响, 如果没有这个结构体, 光有驱动里的 binder_node 几个结构体, 跨进程的指针也是不好实现的, 因为无论 ptr 还是 desc 只是个 32 比特的数(假设是 32 位系统), 进程光靠这个数是没法区分这是本地还是远程 binder, 而驱动光知道 proc 和一个数也是没法清楚用户程序要的是本地还是远程 binder , 需要 type 才能分清楚是远程还是本地 binder 。虽然对于事物请求 (transaction) , 驱动和应用程序可以假定目标 binder 指的远程 binder , 但是想要在两个进程之间传递一个 binder 就无法做到了, 而这个正是 service manager 做的事情, android 各种服务程序和应用程序之间也是通过这种方式传递对象 binder 的。

正如前面所说3.1, 这个结构体主要是用来在进程间传递 binder , 只会发生在进程间 transaction 时, transaction 数据里的 offsets 和 offsets_size 便是用来存放这个结构体的, 当 transaction 发生时, 驱动会修改这里面的数据, 把请求方进程的 binder 转换成 transaction 目标进程的里的 binder , 可能是远程到本地, 本地到远程甚至是远程到远程 binder 的转换。

驱动不仅仅只是完成 binder 的转换和传递, 必要时还得创建 binder_node 和 binder_ref, 分两种情况。一个是 ioctl(BINDER_SET_CONTEXT_MGR), 这是 binder 管理进程[6]把自己注册为 binder 的服务程序。而进程与管理进程通信时, 如果还没有管理进程的远程 binder 时, 便会创建一个 binder_ref。其二, 当 transaction 传递一个本地 binder 时, 如果该 binder 在驱动中并没有记录, 驱动为其创建一个 binder_node[7]。不论传递远程还是

[6] 即 service manager 进程, 这个 binder_node 不是对象, ptr 和 cookie 都为 0

[7] 调用 binder_new_node

本地 binder，如果该 binder 在目标进程中没有记录，驱动为目标进程创建一个 binder_ref[8]。

例如当服务端把本地 Binder 传递给客户端时，在发送数据流中，flat_binder_object 中的 type 是 BINDER_TYPE_BINDER，binder 是binder 对象用户空间地址。驱动必须对数据流中的这个 Binder 做修改：type 设为 BINDER_TYPE_HANDLE；在接收进程中创建或寻找对应的 binder_ref 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才可以将其填入数据包 binder_transaction_data 的 target.handle 域，向本地 Binder 对象发送请求。

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

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代码 3.3，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。

3.3 binder_node

struct binder_node 是 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：

```
1  struct binder_node {
2      int debug_id;
3      struct binder_work work;
4      union {
5          struct rb_node rb_node;
6          struct hlist_node dead_node;
7      };
8      struct binder_proc *proc;
9      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;
```

[8] 调用 binder_get_ref_for_node

```
19      unsigned has_async_transaction:1;
20      unsigned accept_fds:1;
21      unsigned min_priority:8;
22      struct list_head async_todo;
23  };
```

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。

表 6: 结构成员解析

work 子节 4.5	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。
rb_node dead_node	每个进程都维护一棵红黑树，以 Binder 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。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。
proc	节点所属的进程，即提供该节点的进程。
refs	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。
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
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。
local_strong_refs	驱动为传输中的 Binder 设置的强引用计数。同上。
ptr	指向用户空间 Binder 的指针，来自于 flat_binder_object 的 binder 成员

cookie	指向用户空间的附加指针, 来自于 flat_binder_object 的 cookie 成员. 其实 cookie 才是真正的指针, 而 ptr 是应用层为了引用计数弄出来的一个东西, 对于驱动来说, 两者有些重复
has_strong_ref pending_strong_ref has_weak_ref 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
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。
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。
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。
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
debug_id	除了 binder_node , 还有几类信息也有 debug_id , 这几类的 debug_id 在驱动里从 1 开始统一排的。其实也可以用 debug_id 这个一维的索引来表示 binder , 不过效率会比红黑树低, 所以它只能当做 debug 信息用。

binder_proc 里的 nodes 树是用来遍历进程的 binder_node 的, 为了快速查找, nodes 是颗二叉搜索数, key 是 binder_node 的 ptr 。按照 linux 的惯例, binder_node 里需要有个成员, 才能把它加到树或者列表里, 而这个

成员便是 rb_node , 通过 rb_node , 可以把 binder_node 与 binder_proc 的 nodes 关联, 从而实现对一个进程的本地 binder 的管理。

另外, 这个 rb_node 是一个 union, 另一个名字叫做 dead_node , 这个身份是为了处理死亡的 binder 的, 当一个进程结束时, 他的本地 binder 自然也就挂了, binder_node 也就没必要存在了, 但是由于别的进程可能正在使用这个 binder , 所以一时半会, 驱动还没法直接移除这个 binder_node , 不过由于进程挂了, 驱动没法继续把这个 binder_node 挂靠在对应的 binder_proc 下, 只好把它挂靠在 binder_dead_nodes 下面。不过, 这个 binder_dead_nodes 其实也没啥意义, 也就是打印/proc/binder/state 时用了一下, 看看当前有多少已经死亡但没移除的 binder_node 。驱动里真正移除一个死亡的 binder_node 是靠引用计数的机制来完成的。

3.4 binder_ref

binder 在远程进程中的表示, 也就是 Binder 引用在驱动中的表述.

```
1  struct binder_ref {
2          /* Lookups needed: */
3          /*   node + proc => ref (transaction)
              */
4          /*   desc + proc => ref (transaction,
              inc/dec ref) */
5          /*   node => refs + procs (proc exit)
              */
6          int debug_id;
7          struct rb_node rb_node_desc;
8          struct rb_node rb_node_node;
9          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 };
```

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 8: Binder 引用描述结构 :binder_ref

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。
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。
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. 由于一个本地 binder 可以被多个进程使用, 于是一个 binder_node 可能有多个 binder_ref 来对应, 所以 binder_node 中有个 refs 的链表用来记录他的远程表示, 而 node_entry 就是用于这个链表的, 这个没用红黑树, 因为这个表一般比较小, 而且用不着搜索。
proc	本引用所属的进程
node	本引用所指向本地 binder 的 binder_node

desc	本结构的引用号, 即 handle, 她和 proc 一起就能确定一个远程 binder
strong	强引用计数
weak	弱引用计数
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’。

rb_node_desc 用于 binder_proc 的 refs_by_desc , rb_node_node 用于 binder_proc 的 refs_by_node , 这两都是二叉搜索树, 前一个 key 是 desc , 后一个 key 是 node 。如果已知 binder 的远程表示 proc + desc , 就可以用 refs_by_desc 查找, 如果知道本地表示则用 refs_by_node 查找, 看看某个远程 proc 是否已经具备该 binder_node 的远程表示。

一个本地 Binder 可能有很多远程 Binder 引用, 这些引用可能分布在不同的进程中。每个进程使用红黑树存放所有正在使用的引用。Binder 的引用可以通过两个键值索引：

- 本地 Binder 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, 而不是本地 Binder 在用户进程中的地址。本地 binder 内核地址是唯一的, 用做索引不会产生二义性；但本地 binder 用户态地址在其他进程中可能与某些指针重合, 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。

3.5 数据结构的关系

总结一下, binder 是跨进程的指针, 有两种形式, 一是在本地进程里, 他就是个进程指针, 驱动中记为 binder_node , 其中 proc 为它的本地进程, ptr/cookie 为进程里的指针, 另一种是在远程进程里, 他就是个从 0 开始

增加的数(按该进程中远程 binder 出场顺序记录), 驱动里记为 binder_desc , 其中 proc 为远程进程, desc 为序号。驱动中维护远程 binder 和本地 binder 的转换, 包括远程 (proc, desc) 与本地 (proc, ptr) 的互相转换, 也包括远程 (proc, desc) 和另一个远程 (proc, desc) 的互相转换, 当然也支持本地到本地的转换(这种情况就是完全相同的一个东西), 不过这种转换驱动里是不会发生的, 上层的 c++ 的 binder 库中已经识别了 binder 是不是本地的, 本地的 binder 自行就处理了, 不需要经过驱动这层额外开销。

下面列几个相关的函数, 以下省略 struct 关键字。

- binder_node* binder_get_node(binder_proc *proc, void __user *ptr)

  给定 ptr(binder_node 中的 ptr) 和进程 proc, 在 proc 的 nodes 树中查找并获取 binder_node 结构, 如果这个 proc 的这个指针在驱动里还没有建立本地 binder, 则返回 null。

- binder_node* binder_new_node(binder_proc *proc, void __user *ptr, void __user *cookie)

  给定 ptr 和 cookie 以及进程 proc, 在驱动中创建该指针的本地 binder 项, 并添加到 proc 的 nodes 树中。

- binder_ref *binder_get_ref(binder_proc *proc, uint32_t desc)

  给定 desc 和 proc, 获取远程 binder 的表示 binder_desc, 如果没有则返回 null。

- binder_ref *binder_get_ref_for_node(binder_proc *proc, binder_node *node)

  给定 proc 和 node, 获取某个本地 binder 在该 proc 进程里的远程表示, 如果该本地 binder 在该 proc 中还没有建立远程表示, 则创建它并返回。此函数会用到并可能更新 proc 的 refs_by_desc, refs_by_node 树以及 node 的 refs 链表。

- binder_delete_ref(binder_ref *ref)

  删除给定的远程 binder。并不存在 binder_delete_node 函数, node 的删除是在 node 的引用计数减到 0 时发生的, 被整合到了 binder_dec_node 函数。

4 进程通信的管理

4.1 进程和 binder_proc

用来表示进程的

```
1  struct binder_proc {
2      struct hlist_node proc_node;
3      struct rb_root threads;
4      struct rb_root nodes;
5      struct rb_root refs_by_desc;
6      struct rb_root refs_by_node;
7      int pid;
8      struct vm_area_struct *vma;
9      struct task_struct *tsk;
10     struct files_struct *files;
11     struct hlist_node deferred_work_node;
12     int deferred_work;
13     void *buffer;
14     ptrdiff_t user_buffer_offset;
15     struct list_head buffers;
16     struct rb_root free_buffers;
17     struct rb_root allocated_buffers;
18     size_t free_async_space;
19     struct page **pages;
20     size_t buffer_size;
21     uint32_t buffer_free;
22     struct list_head todo;
23     wait_queue_head_t wait;
24     struct binder_stats stats;
25     struct list_head delivered_death;
26     int max_threads;
27     int requested_threads;
28     int requested_threads_started;
29     int ready_threads;
30     long default_priority;
31     struct dentry *debugfs_entry;
32 };
```

首先, 这个结构体得能表示一个进程以及记录一个进程的资源, 成员变量 pid, tsk, files 就是用来干这些的, 另外还有 threads 和 page, buffer 之类的成员也是用来表示进程资源的。
其次, 既然进程在两种存在方式里都是第一维, 那么通过它, 我们应该能够得到进程所有的 binder(本地的以及远程的), 他的 nodes, refs_by_desc, refs_by_node 就是用来实现这个的, 这三个都是红黑树根节点。

驱动里都是以 binder_proc 为单元来分配和记录资源的, 驱动会给每个进程创建一个/proc/binder/proc/*pid* 文件, 用来展示进程相关的 binder 信息, 而进程的 binder 清理工作则是通过标记量 deferred_work 以及链表 deferred_work_node 实现的。至于 transaction 需要的内存空间则是通过一系列名字包含 buffer 和 page 的成员变量完成的,

debugfs_entry 对应于 proc 文件/proc/binder/proc/*pid*。delivered_death 则是用于死亡通知的, 而 stats 是用于统计 binder 信息的,/proc/binder/stats

threads , 这个是用来记录进程中与 binder 有一腿的线程的, 与线程相关的还有各种含 thread 的变量以及 todo, wait 和 default_priority 。

4.2 线程和 binder_thread

Linux 的线程是轻量级的进程。Binder 记录线程的数据结构是结构体 binder_thread

```
1  struct binder_thread {
2          struct binder_proc *proc;
3          struct rb_node rb_node;
4          int pid;
5          int looper;
6          struct binder_transaction *transaction_stack;
7          struct list_head todo;
8          uint32_t return_error; /* Write failed, return error code in read buf */
9          uint32_t return_error2; /* Write failed, return error code in read */
10                 /* buffer. Used when sending a reply to a dead process that */
11                 /* we are also waiting on */
12         wait_queue_head_t wait;
13         struct binder_stats stats;
14 };
```

proc 用来记录该线程所属进程的 binder_proc 结构体, pid 记录线程的 id , stats 同 proc 的, 用于统计线程的 binder 信息, return_error 和

return_error2 用来记录 ioctl 的一些错误信息, transaction_stack 用于 transaction , looper 记录线程的状态, 与后面的线程池相关。wait 和 todo 则跟线程的工作相关。

rb_node 用来关联 binder_proc 的 threads 树, 这也是一棵二叉搜索树, key 很显然便是线程的 pid 。有一个很重要的函数跟这棵树相关：

```
1  binder_thread *binder_get_thread(binder_proc *proc)
```

任何一个线程执行 binder 驱动的 poll 和 ioctl 操作时, 驱动都会从文件结构体获取 binder_proc 信息, 然后尝试从这个进程信息里获取当前线程的信息, 如果没找到, 则为当前线程创建一个 binder_thread , 并记录到进程的 threads 中。所以, 进程的 threads 只是记录了那些跟 binder 有一腿的线程的, 而不会记录进程的所有线程。

4.3 进程池

写它之前, 先简单的提一下 ioctl(BINDER _WRITE_READ) 。这个 ioctl 有两步, 第一步是 write , 会传给驱动一些以 BC_ 开头的命令, 让驱动去执行, 一次可以传多个命令。第二步是 read, 会从驱动获取一些以 BR_ 开头的回复, 也可能是多个, 上层 binder 库可以根据这些回复做相应的操作, read 这一步有可能阻塞当前线程, 直到驱动给他回复并唤醒它。因为可能会有许多客户端同时发起请求, 服务进程为了提高效率往往开辟线程池并发处理收到的请求。

所谓的线程池跟平时说的线程区别并不大, 进程会产生一些线程, 让他们做无限循环, 在循环里等待, 直到外界有工作传来, 该线程便被唤醒, 待到该线程干完活之后, 又继续循环, 再次进入等待状态, 直到下一个工作来临。不同的是, 传统的线程池的工作是本地进程里的某个不加入线程池的线程给的, 而 android 的线程池等待的工作则是由别的进程里的线程给的。这个工作由别的进程里的线程通过 BINDER_WRITE_READ 的 write 步骤提交给驱动, 线程池中的线程则一直在循环做着 BINDER_WRITE_READ , 他们的 read 步骤会阻塞, 直到驱动把别的线程通过 write 写过来的工作传给他们, 线程池中的线程的 read 得以返回, 便根据得到的 BR 进行相应的操作, 之后再次 BINDER_WRITE_READ , 通过 write 把结果传回驱动, 然后线程又可能再次 BINDER_WRITE_READ 阻塞在 read 阶段等待新的工作。

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。

驱动除了转交别的进程的工作，也可能自己产生工作给线程池做，同样是在线程池的 read 阶段返回给线程池。这一类包括一堆引用计数的工作，以及接下来要写的 LOOPER 工作。

4.4 线程池的循环

所谓的 LOOPER 就是线程池的无限循环，先来看看 LOOPER 相关的枚举量：

代码 4: Binder Looper state

```
1  enum {
2      BINDER_LOOPER_STATE_REGISTERED = 0x01,
3      BINDER_LOOPER_STATE_ENTERED = 0x02,
4      BINDER_LOOPER_STATE_EXITED = 0x04,
5      BINDER_LOOPER_STATE_INVALID = 0x08,
6      BINDER_LOOPER_STATE_WAITING = 0x10,
7      BINDER_LOOPER_STATE_NEED_RETURN = 0x20
8  };
```

这些标记(以下枚举量省略前缀 BINDER_LOOPER_STATE_)会用于 proc_thread 的 looper 变量，用来表明线程的状态，下面会一一提到。

如果一个线程想加入线程池，获取进程 work 时，需要发送 BC_ENTER_LOOPER 或者 BC_REGISTER_LOOPER 命令给驱动以表明自己进入了线程池循环，可以接受进程 work，驱动会根据命令设置线程的 looper 的 ENTERED 或者 REGISTERED 标志位。而一个线程如果要退出线程池，需要发送 BC_EXIT_LOOPER 给驱动，此时 looper 的 EXITED 标记被设置。一个线程(不一定加入了线程池)如果没有事情可做，在 read 阶段阻塞时，WAITING 标记被设置。

而 NEED_RETURN 状态则是所有线程的初始状态，这个状态如果被设置，线程不管有没有 work 要做，read 都将立刻返回，而这个标记在每次 ioctl 后都被清零，就是说这个标记正常情况只在第一次 ioctl 时起作用，而几乎所有的线程的第一次 ioctl 都是 write 命令 BC_ENTER_LOOPER 或者 BC_REGISTER_LOOPER 把自己加入线程池，所以不管线程池有没有 work 可做，这些线程在把自己注册到线程池后都会返回，这样用户程序才好得知线程是否成功加入线程池。

有一个例外，任何一个进程用于 open 驱动的那个线程，一般是进程的主线程，binder 库中，这个线程在 open 驱动之后会进行 ioctl(BINDER_VERSION) 用来检测驱动版本，便产生了副作用，此 ioctl 清除了主线程的 NEED_RETURN 标记，导致主线程在 ENTER_LOOPER 会被阻塞。于是有一个有趣的现象，应用程序里的主线程如果调用 joinThreadPool , BC_ENTER_LOOPER 时不会立刻返回，也就不会生成一个新的线程，而程序里用 startThreadPool 新起的线程在 BC_ENTER_LOOPER 后会返回，而且几乎肯定会再生成一个新的线程。至于为啥会生成一个新的线程，后面写 BC_ENTER_LOOPER 和 BC_REGISTER_LOOPER 的区别时会写到，那里也将涉及到最后一个状态 INVALID 。

关于 NEED_RETURN 还有一个非正常情况，那就是文件操作中的 flush，flush 情况下会设置所有线程的 NEED_RETURN 标记并唤醒 WAITING 中的线程，这样所有的线程的 read 都会得到返回。

BC_ENTER_LOOPER 和 BC_REGISTER_LOOPER 都是用来将线程注册到驱动，以便他们能够获取进程 work 。前者适用于应用程序自己主动产生的线程，比如主线程以及 startThreadPool 产生的第一个线程。

任何一个线程在 read 阶段返回时，会检查进程的 ready_threads，这个变量表示当前进程有多少个线程正在等待进程 work，这类线程必然处于 WAITING 状态(反过来不一定成立 ^-^)。如果当前没有线程等待进程 work，驱动在 read 的返回 BR 队列最前面加上一个 BR_SPAWN_LOOPER 通知该线程，" 线程池已经没有可用于处理进程 work 的线程了，你得赶紧给我产生一个新的"。线程在 read 返回之后便会根据该 BR 创建一个新的线程，并让它 BC_REGISTER_LOOPER 加入线程池并待命，然后自己才去处理这次 read 到的真正 work，这样可以缩短线程池的真空期。

这里有点小优化，如果驱动已经要求某个线程生成一个新的线程，在这个线程还没加入线程池前，驱动不会在别的线程 read 返回时再次要求生成新线程，这样可以避免要求进程生成过多的线程。这件事情是通过进程的 requested_threads 变量实现的，在 read 返回 BR_SPAWN_LOOPER 时，此变量被增加，在新的线程 BC_REGISTER_LOOPER 成功时此变量被减小，用来表明生成线程的工作已经完成，而通过这种方式生成的线程数会被记录到进程的 reqeusted_threads_started，这种方式生成的线程数不能超过进程的 max_threads，此数通过 ioctl(BINDER_SET_MAX_THREADS) 设置，binder 库中设置为 15。注意，这个 MAX 并不约束进程的线程数，也不约束线程池中的线程数，仅仅约束在驱动要求下生成的线程数，即通过 BC_REGISTER_LOOPER 加入线程池的线程数。

如果到了 MAX 后，线程池中的线程还不够响应远程请求时，用户进程可以自行生成新的线程并用 BC_ENTER_LOOPER 注册到线程池用来提供服务。不过一般没必要这么做，线程多了并不见得能改善服务速度，还不如让请求等待着，直到线程池已有的线程完成手头工作再去响应新的请求。

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。

最后一个状态 INVALID 便是与这两种 BC 相关，如果一个 ENTERED 的线程发起 BC_REGISTER_LOOPER 或者 REGISTERD 的线程发起 BC_ENTER_LOOPER，都会被标记为 INVALID 状态。而一个线程发起 BC_REGISTER_LOOPER 时，驱动会检查 requested_threads，如果它发现自己并没

有要求哪个线程生成新的线程并让其 BC_REGISTER_LOOPER, 也会将该线程标记为 INVALID。

4.5 todo 列表和 binder_work

进程 IPC, 经线程池, 交给线程, 线程会把工作放在队列, 最后分解成 work。通常数据传输的接收端有两个队列：工作队列和等待队列, 用以缓解供需矛盾。

工作队列分进程和线程, 分别记录在 binder_proc22和 binder_thread7的 todo 列表。等待队列也分进程和线程, binder_proc wait23队列以及 binder_thread wait12队列里。其实, 等待队列里只有一个线程的, 不管是进程还是线程的。

进程 work 可能被线程池中任意一个线程通过 read 获取, 而线程 work 只会给指定线程获取。

work 发往进程的工作队列还是线程的工作队列, 有两条规则。

- 规则 1：客户端发给服务端的请求数据包都提交到服务端的进程工作队列。除非 4.4中的情况, 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)都发送到发起请求方的线程工作队列中。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。

工作队列规则决定了等待队列规则, 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：

- 客户端, 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；
- 服务端, 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。

work 本身用 binder_work 表示, 如下：

```
1  struct binder_work {
2      struct list_head entry;
3      enum {
4          BINDER_WORK_TRANSACTION = 1,
5          BINDER_WORK_TRANSACTION_COMPLETE,
6          BINDER_WORK_NODE,
7          BINDER_WORK_DEAD_BINDER,
8          BINDER_WORK_DEAD_BINDER_AND_CLEAR,
9          BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
10     } type;
11 };
```

entry 是用来把 work 加入到相应 todo 里的。而 type(以下略去前缀 BINDER_WORK_)则是 work 的性质。

TRANSACTION 类型的 work 就是前面多次提到的 transaction, android 里进程之间普通的请求就是这个东西, 这里先简单说下流程, 以后再详写实现。

通常, 一个线程需要另一个进程的 binder 进行服务时, 它会 write BC_TRANSCATION 向某个 binder 来发起 transaction, 驱动会创建一个 TRANSACTION 类型的 work, 并把它加到 binder 所在进程 work 里, 然后唤醒线程池中的一个线程, 让他去完成这个请求。binder 完成任务时, 所在的线程会 write BC_REPLY 发回 transaction 完成的结果, 此时驱动也会创建一个 TRANSACTION 类型的 work, 并把它加到 transaction 发起者的线程 work 里, 然后唤醒发起线程。这一来一去有个很大不同之处便是接受者, 发起 transaction 时面对是整个线程池, 所以 work 会被加到目标进程的 work 里, 而 transaction 回复时是知道发起者线程的, 所以是加到线程 work 中。

在上述过程中, BC_TRANSACTION 和 BC_REPLY 的 write 完成时, 驱动会立刻往写入命令的线程 work 中增加一个 TRANSACTION_COMPLETE 类型的 work, 以便让线程再接下来的 read 阶段能够立刻返回。binder 库中的 transact 在发出 BC_TRANSACTION 并收到这个 BR 后, 会立刻再次进行 iocl(BINDER_WRITE_READ) 进入等待状态, 直到远端的 binder 实际完成 transaction, 所以看起来线程就像是在发出 transaction 后便阻塞了。

NODE 类型的 work 是用于引用计数的, 当 binder_node 的引用计数发生某些变化时, 驱动就会往进程或者线程的 work 里添加这个 binder_node 的 work, 还记得 binder_node 里的 work 成员变量么。

DEAD_BINDER, DEAD_BINDER_AND_CLEAR, CLEAR_DEATH_NOTIFICATION 这三种类型是用于死亡通知的。

最后要提一点, 驱动在处理各种 work 时, 会考虑当前线程 LOOPER 状态, 如果没有 ENTERED 或者 REGISTERED 的, 便一定会把 work 添加到进程 work 里, 否则很可能加到当前线程的 work 里, 这个得看实际情况了。

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。驱动没有简单把异步交互直接扔往 todo 列表, 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，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。

总结一下 binder 和进程，线程，线程池的关系。binder 本质是个跨进程的指针，只需要跟进程挂钩，所以它可以被进程的任意一个线程处理，于是 android 这种服务于其他进程的线程池便有了存在的根基。android 中一个本地 binder 一般就对应着一个服务，会有很多请求者，所以用线程池来响应是比较合适的。各种 LOOPER 状态使得应用程序的线程池能够很好的投影到驱动里，保持高度的一致性。而驱动要求线程 spawn 线程的机制与 binder 库的无缝结合，使得高层用户不用关心线程池的容量，高层用户只需要知道线程池他不是一个线程在战斗就行了，线程池搞不定时他会自己再生出一个线程来帮手的。而进程中的远程 binder 一般却是活在特定线程中，他们是服务请求的发起者，所以无所谓用线程池来响应请求，而且他们一般是由应用程序某个特定线程调度，用来完成特殊命令的，这样的线程是不愿意让别的线程获取自己发出的请求的答复的，所以远程 binder 虽然是活在进程里，但是他永远住在某个特定的线程，从不搬家。至于 work 的性质，添加和移除，则跟 transaction，引用以及死亡通知相关，以后再详写。

4.6 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() 系统调用小节 5.3.2，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：

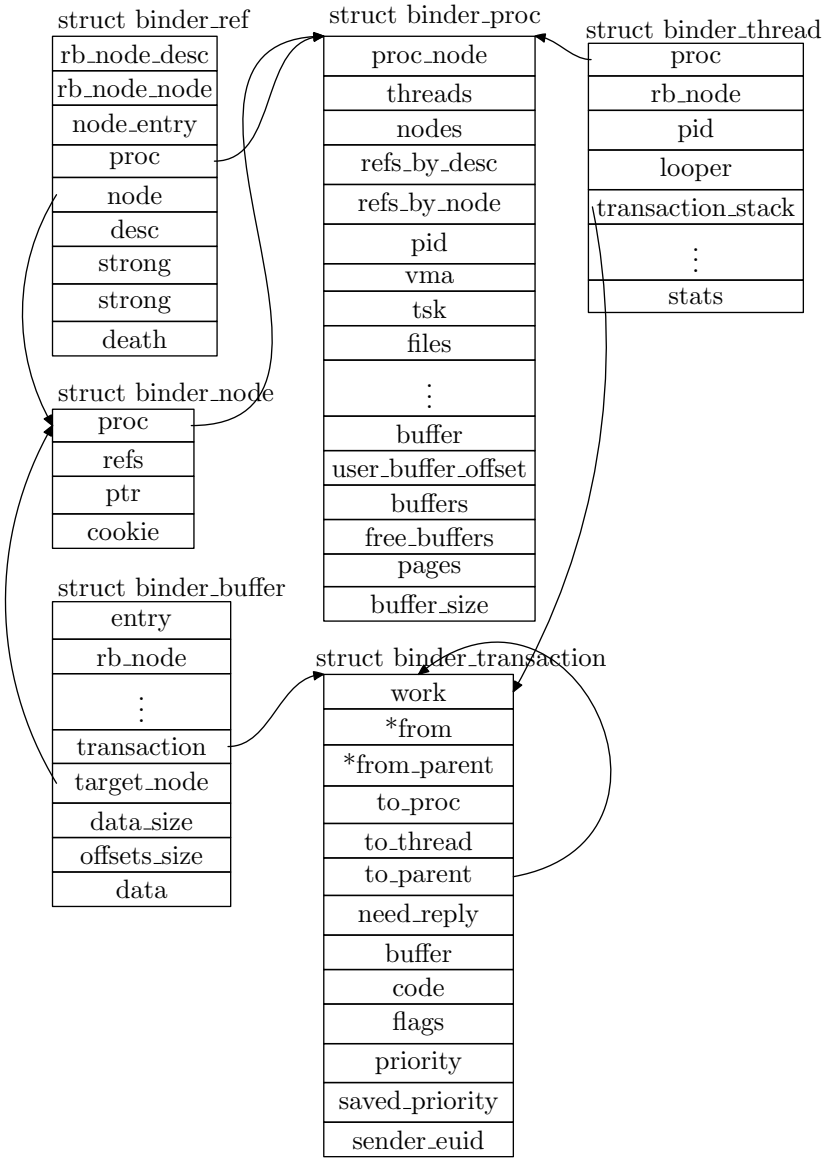
```
fd = open("/dev/binder", O_RDWR);
mmap(NULL, MAP_SIZE, PROT_READ, MAP_PRIVATE, fd, 0);
```

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 结构本身以及表格 3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 表格 2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’。



5 Binder 驱动的函数

5.1 binder_init 函数

- binder_init 调用 create_singlethread_workqueue("binder") 创建一个 workqueue 来做一些延迟工作。
- proc 文件系统中建立目录 binder 和 binder/proc。
- 注册 binder 驱动。
- /proc/binder 下建立几个 proc 文件:

- state, stats, transaction, stransaction_log, failed_transaction_log
  这些个 proc 文件和/binder/proc/*pid*都可以读出 binder 相关的状态和统计信息。
- /binder/proc/*pid*
  这些文件是各个 binder 进程的信息，会在 open 函数中建立。这些个 proc 文件能帮助理解 binder，可惜的是，有长度限制，最多只会打印一个内存页面那么多字节，对于 node 或者 ref 多的进程，信息就显示不全，比如 system_server 的 binder 信息。具体每个文件的信息等把 binder 的数据结构写完了再加上。

5.2 module 参数

参数可在/sys/module/binder/parameters 下找到，可读可写。

- debug_mask 调试标志位，参考 enum BINDER_DEBUG_XXX，相应 bit 置 1 能看到相应的调试信息，都是 KERN_INFO 级别。不过 adb 下是看不到 printk 的，得 dmesg 或者用串口看。这个参数的设置可以帮助理解 Binder 驱动的工作流程。
- proc_no_lock
  查看 proc 文件时是否加 Binder 全局锁，一般不加，查看 proc 一般也不需要得到最准确的信息，没必要锁着，影响 Binder 工作。
- stop_on_user_error 设置为 0 时，不起作用，设置为 1 时，如果某个进程调用 binder 时发生错误，则该进程 sleep，以后别的进程再调用 binder 的 ioctl 也会 sleep，此时你若往这个参数写入一个小于 2 的数，进程会被唤醒。设置为 2 以上的值，那悲剧了，ioctl 直接睡觉。该参数默认为 0。

5.3 文件操作

static const struct file_operations binder_fops

5.3.1 binder_open

- 创建 binder_proc 结构体，用来记录调用进程的信息，并进行相关初始化工作。将该结构体指针记录到 filp->private_data，这样以后别的文件操作都能得到当前进程的 Binder 信息。
- 建立/proc/binder/pid 文件，用于查询进程的 binder 信息。

5.3.2 binder_mmap

内核内存空间中分配最大 4M 的连续空间给进程，并关联到进程的用户空间。

- 之所以要连续的，是为了方便的维护与用户空间地址的映射，内核地址只需要加上一个偏移量 proc->user_buffer_offset 就能得到用户地址了。
- 该空间用于 transaction 交互数据用。
  假设 A 是服务程序，B 是客户程序，B 发出请求后，驱动会把 B 传送的数据从 B 的用户空间拷贝到内核中分配给 A 进程的内存空间，然后通知 A 进程工作，由于内核空间的内存已经通过 mmap 映射到 A 的用户空间，驱动只需要给 A 进程一个指向内核空间的用户指针即可，A 就能获得从 B 考过来的数据，于是节省了一次从驱动到 A 用户空间的数据拷贝。
- 由于该操作负责分配空间给进程，所以使用 Binder 的进程必须调用一次，以设定其用于 transaction 的总 buffer 空间，一般在 open 之后就会执行，否则进行 transaction 服务会得到 BR_FAILED_REPLY 的错误，如果空间不够用的时候，同样会得到这个错误。
- 这个操作还会保存进程的 files_struct 到 binder_struct 中用于 binder 的 fd 操作。

此函数的大致操作流程如下：[9]

- get_vm_area 获取内核空间的一段连续虚拟地址
- kzalloc 一段空间给 proc->pages 用于保存页表节点，
- binder_update_page_range 调用 alloc_page 分配页面并用 map_vm_area 关联到内核的虚拟地址，
- 同时用 vm_insert_page 把分配到的页面添加到进程的用户空间。

5.3.3 binder_poll

- 检查是否有 proc 或 thread 的工作要做，有的话返回 POLLIN。
- 没有的话进行 poll_wait。
- 醒来之后再次检查，有工作要做的话返回 POLLIN，否则返回 0。

5.3.4 binder_ioctl

- 进入时检查 binder_stop_on_user_error子节 5.2，确定是否 wait。
- binder_get_thread(proc)，获取或者初始化当前线程在 binder 中的节点。binder_proc 中有颗红黑树 threads 用结构体 binder_thread

[9] 注意，mmap 的时候只分配了第一个页面，其余的页面在以后调用 binder_alloc_buf 的时候才分配，也算是一种 page on demand 吧。

记录了各个 thread 的信息, 这是颗按 pid 排列的二叉查找树。比较重要的初始化, 每个 binder_thread 的 looper 的 BINDER_LOOPER_STATE_NEED_RETURN 标记被设置, 这将使得任何一个线程的第一次 ioctl 一定回返回, 而不会阻塞在 read 阶段。binder write/read 操作的时候再详细说明。

- 各种 binder ioctl 操作。
- 清空线程 looper 的 BINDER_LOOPER_STATE_NEED_RETURN, 没有这个标记之后, 以后的 ioctl 就有可能阻塞在 read 阶段, 写 binder writeread 操作的时候再详细说明。
- 返回前再次检查 binder_stop_on_user_error, 确定是否 wait。

5.3.5 binder_flush

- 调用 binder_defer_work(proc, BINDER_DEFERED_FLUSH) 把 flush 的工作交给 binder_init 中创建的 workqueue 来延迟执行。
- workqueue 最终会执行 binder_deferred_flush 来进行 flush。flush 函数会设置该进程的每个线程的 looper 标记的 BINDER_LOOPER_STATE_NEED_RETURN 位, 前面提过, 该标记会导致 read 操作立刻返回, 当然, 还需要唤醒那些已经睡觉的线程, 不然光设置一个标记也没用。总之, 这个操作就是让所有的线程不再睡觉, 主要是针对那些等待 read 的线程。

5.3.6 binder_release

- 删除/binder/proc/*pid*。
- 调用 binder_defer_work(proc, BINDER_DEFERED_RELEASE) 把 release 的工作交给 binder_init 中创建的 workqueue 来延迟执行。
- workqueue 最终会执行 binder_deferred_release 来释放该进程在 binder 驱动中相关的内容, 包括 page, buffer, node, ref, thread, proc 。node 的释放是将其先放到 dead_node 表中, 并想那些注册过死亡通知的线程发出死亡通知, 而 dead_node 需要等到该 node 所有的 ref 都删除时才会真的删除。

6 utils 库 :指针和引用计数

A Binder Example

代码 5: Interface header: IEneaBuffer.h

```
1  /*
2   * IEneabuffer.h
3   *  Created on: 19 mars 2010  Author: Zingo Andersen
4   *  License: Public Domain (steal and use what you like)
5   *
```

```
6    * Buffer classes to handle the binder communication   */
7
8   #ifndef IENEABUFFER_H_
9   #define IENEABUFFER_H_
10
11  #include <utils/RefBase.h>
12  #include <binder/IInterface.h>
13  #include <binder/Parcel.h>
14  #include <binder/IMemory.h>
15  #include <utils/Timers.h>
16
17  namespace android {
18      class IEneaBuffer: public IInterface {
19          public:
20            DECLARE_META_INTERFACE(EneaBuffer);
21            virtual sp<IMemoryHeap>  getBuffer() = 0;
22      };
23
24      /* ---- Server side ---- */
25      class BnEneaBuffer: public BnInterface<IEneaBuffer> {
26          public:
27            virtual status_t    onTransact( uint32_t code,
28                                            const Parcel& data,
29                                            Parcel* reply,
30                                            uint32_t flags = 0);
31      };
32
33  }; // namespace android
34  #endif /* IENEABUFFER_H_ */
```

代码 6: Interface class: IEneaBuffer.cpp

```
1   /*
2    * IEneaBuffer.cpp
3    *  Created on: 19 mars 2010  Author: Zingo Andersen
4    *  License: Public Domain (steal and use what you like)
5    *
6    * Buffer classes to handle the binder communication
7    */
8
9   //#define LOG_TAG "IEneaBuffer"
10  #include <utils/Log.h>
11  #include <stdint.h>
12  #include <sys/types.h>
13  #include <binder/MemoryHeapBase.h>
14  #include <IEneaBuffer.h>
15
16  namespace android {
17
18  enum {
19      GET_BUFFER = IBinder::FIRST_CALL_TRANSACTION
20  };
21
22  /* ---- Client side ---- */
23  class BpEneaBuffer: public BpInterface<IEneaBuffer> {
24  public:
25    BpEneaBuffer(const sp<IBinder>& impl) : BpInterface<IEneaBuffer>(impl) {
26    }
27
28    sp<IMemoryHeap> getBuffer() {
29      Parcel data, reply;
30      sp<IMemoryHeap> memHeap = NULL;
31      data.writeInterfaceToken(IEneaBuffer::getInterfaceDescriptor());
32      // This will result in a call to the onTransact()
33      // method on the server in it's context (from it's binder threads)
34      remote()->transact(GET_BUFFER, data, &reply);
35      memHeap = interface_cast<IMemoryHeap> (reply.readStrongBinder());
36      return memHeap;
37    }
38  };
39
40  IMPLEMENT_META_INTERFACE(EneaBuffer, "android.vendor.IEneaBuffer");
41
42  /* ---- Server side ---- */
43  status_t BnEneaBuffer::onTransact(uint32_t code, const Parcel& data, Parcel* reply,
      uint32_t flags) {
44    switch (code) {
45      case GET_BUFFER: {
46        CHECK_INTERFACE(IEneaBuffer, data, reply);
47        sp<IMemoryHeap> Data = getBuffer();
48        if (Data != NULL) {
49          reply->writeStrongBinder(Data->asBinder());
50        }
```

```
51          return NO_ERROR;
52          break;
53        }
54        default:
55          return BBinder::onTransact(code, data, reply, flags);
56      }
57    }
58    }; // namespace android
```

Then use it in your server class by inherit the server class with something like this:

代码 7: Server command: EneaBufferServer.cpp

```
1   /*
2    * EneaBufferServer.cpp
3    *  Created on: 19 mars 2010  Author: Zingo Andersen
4    *  License: Public Domain (steal and use what you like)
5    *
6    *  The Server will create a shared area that the client will then use
7    *  The server will initiate the first int in the area and print it's value every
8    *  5s. If you start the client in parallell it will try to change this value
9    *  (value=value+1).
10   */
11
12  #include "IEneaBuffer.h"
13  #include <binder/MemoryHeapBase.h>
14  #include <binder/IServiceManager.h>
15  #include <binder/IPCThreadState.h>
16
17  namespace android {
18
19      #define MEMORY_SIZE 10*1024  /* 10Kb shared memory*/
20
21      class EneaBufferService : public BnEneaBuffer {
22          public:
23            static void instantiate();
24            EneaBufferService();
25            virtual ~EneaBufferService();
26            virtual sp<IMemoryHeap> getBuffer();
27          private:
28            sp<MemoryHeapBase> mMemHeap;
29      };
30
31      sp<IMemoryHeap> EneaBufferService::getBuffer() {
32        return mMemHeap;
33      }
34
35      void EneaBufferService::instantiate() {
36        status_t status;
37        status = defaultServiceManager()->addService(String16("vendor.enea.Buffer"),
38                                                     new EneaBufferService());
39      }
40
41      EneaBufferService::EneaBufferService() {
42        //The memory is allocated using a MemoryHeapBase, and thereby is using ashmem
43        mMemHeap = new MemoryHeapBase(MEMORY_SIZE);
44        unsigned int *base = (unsigned int *) mMemHeap->getBase();
45        *base=0xdeadcafe; //Initiate first value in buffer
46      }
47
48      EneaBufferService::~EneaBufferService() {
49        mMemHeap = 0;
50      }
51
52      static sp<IMemoryHeap> receiverMemBase;
53
54      unsigned int * getBufferMemPointer(void) {
55        static sp<IEneaBuffer> eneaBuffer;
56
57        /* Get the buffer service */
58        if (eneaBuffer == NULL) {
59          sp<IServiceManager> sm = defaultServiceManager();
60          sp<IBinder> binder;
61          binder = sm->getService(String16("vendor.enea.Buffer"));
62          if (binder != 0) {
63            eneaBuffer = IEneaBuffer::asInterface(binder);
64          }
65        }
66        if (eneaBuffer == NULL) {
67          LOGE("The buffer service is not published");
```

```
68            return (unsigned int *)-1; /* return an errorcode... */
69          } else {
70            receiverMemBase = eneaBuffer->getBuffer();
71            return (unsigned int *) receiverMemBase->getBase();
72          }
73        }
74    }
75
76    using namespace android;
77
78    int main(int argc, char** argv) {
79        unsigned int *base;
80
81        EneaBufferService::instantiate();
82
83        //Create binder threads for this "server"
84        ProcessState::self()->startThreadPool();
85        LOGD("Server is up and running");
86
87        base = getBufferMemPointer();
88
89        for(;;) {
90         LOGD("EneaBufferServer base=%p Data=0x%x", base,*base);
91         sleep(5);
92        }
93        // wait for threads to stop
94        //   IPCThreadState::self()->joinThreadPool();
95        return 0;
96    }
```

You need add the following line to register the service in the file frameworks/base/cmds/servicemanager/service_manager.c

```
1   { AID_MEDIA, "vendor.enea.Buffer" },
```

On the client process side you can use something like the code snippet below to get the memory pointer.

NOTE: MemoryHeapBase is based on strong pointer and will be "magical" ref counted and removed thats why the client put the object pointer in a static variable in the example below to keep it from being removed.

代码 8: EneaBufferClient.cpp

```
1   /*
2    * EneaBufferClient.cpp
3    *  Created on: 19 mars 2010  Author: Zingo Andersen
4    *  License: Public Domain (steal and use what you like)
5    *
6    *  Get the shared memory buffer from the server and change the first int value
7    *  by adding one to it. The Server should be running in parallell pleas view
8    *  the logcat for the result
9    */
10
11
12  #include "IEneaBuffer.h"
13  #include <binder/MemoryHeapBase.h>
14  #include <binder/IServiceManager.h>
15
16  namespace android {
17  static sp<IMemoryHeap> receiverMemBase;
18
19  unsigned int * getBufferMemPointer(void) {
20    static sp<IEneaBuffer> eneaBuffer = 0;
21
22    /* Get the buffer service */
23    if (eneaBuffer == NULL) {
24      sp<IServiceManager> sm = defaultServiceManager();
25      sp<IBinder> binder;
26      binder = sm->getService(String16("vendor.enea.Buffer"));
27      if (binder != 0) {
28        eneaBuffer = IEneaBuffer::asInterface(binder);
29      }
30    }
```

```
31      if (eneaBuffer == NULL) {
32        LOGE("The EneaBufferServer is not published");
33        return (unsigned int *)-1; /* return an errorcode... */
34      } else {
35        receiverMemBase = eneaBuffer->getBuffer();
36        return (unsigned int *) receiverMemBase->getBase();
37      }
38    }
39
40    }
41
42    using namespace android;
43
44    int main(int argc, char** argv) {
45     // base could be on same address as Servers base but this
46     // is purely by luck do NEVER rely on this. Linux memory
47     // management may put it wherever it likes.
48        unsigned int *base = getBufferMemPointer();
49        if(base != (unsigned int *)-1) {
50            LOGD("EneaBufferClient base=%p Data=0x%x\n",base, *base);
51            *base = (*base)+1;
52            LOGD("EneaBufferClient base=%p Data=0x%x CHANGED\n",base, *base);
53            receiverMemBase = 0;
54        } else {
55         LOGE("Error shared memory not available\n");
56        }
57        return 0;
58    }
```

And my Android.mk file for this:

代码 9: Android.mk

```
1   # Ashmem shared buffer example
2   # Created on: 19 mars 2010  Author: Zingo Andersen
3   # License: Public Domain (steal and use what you like)
4
5   LOCAL_PATH:= $(call my-dir)
6
7   #
8   # BufferServer
9   #
10  include $(CLEAR_VARS)
11
12  LOCAL_SRC_FILES:=         \
13    IEneaBuffer.cpp       \
14    EneaBufferServer.cpp \
15
16  LOCAL_SHARED_LIBRARIES:= libcutils libutils libbinder
17  LOCAL_MODULE:= EneaBufferServer
18  LOCAL_CFLAGS+= -DLOG_TAG=\"EneaBufferServer\"
19  LOCAL_PRELINK_MODULE:=false
20  include $(BUILD_EXECUTABLE)
21
22  #
23  # BufferClient
24  #
25  include $(CLEAR_VARS)
26
27  LOCAL_SRC_FILES:=         \
28    IEneaBuffer.cpp       \
29    EneaBufferClient.cpp \
30
31  LOCAL_SHARED_LIBRARIES:= libcutils libutils libbinder
32  LOCAL_MODULE:= EneaBufferClient
33  LOCAL_CFLAGS+= -DLOG_TAG=\"EneaBufferClient\"
34  LOCAL_PRELINK_MODULE:=false
35  include $(BUILD_EXECUTABLE)
```

B 纯虚析构函数

为虚析构函数工作的方式是：最底层的派生类的析构函数最先被调用，然后各个基类的析构函数被调用。这就是说，即使是抽象类，编译器也要产生对基类析构函数的调用，所以要保证为它提供函数体。如果不这么做，链接器就会检测出来，最后还是得回去把它添上。

C++ 的多态性是通过虚函数来实现的，虚函数的出现使得动态链接成为可能。

基于构造函数的特点，不能将构造函数定义为虚函数，但可以将析构函数定义为虚函数。

一般情况：当派生类的对象从内存中撤销时，会先调用派生类的析构函数，然后自动调用基类的析构函数，如此看来析构函数也没有必要定义为虚函数。

但如考虑如下这种情况，如果使用基类指针指向派生类的对象，而这个派生类对象恰好是用 new 运算创建的，这种情况下会如何呢？当程序使用 delete 运算撤销派生类对象时，这时只会调用基类的析构函数，而没有调用派生类的析构函数。如果使用的是虚析构函数的话，就不一样了，所以定义虚析构函数有时候还是很有必要的。

代码 10: VirtualDestructor.cpp

```
1
2   using namespace std;
3   #include <iostream>
4
5   class ClxBase {
6       public:
7           ClxBase() {} ;
8
9           virtual void DoSomething() { cout << "Do something in class ClxBase!" << endl; };
10      //protected:
11          virtual ~ClxBase() {} ;
12  };
13
14  class ClxDerived : public ClxBase {
15      public:
16          ClxDerived() {} ;
17          ~ClxDerived() { cout << "Output from the destructor of class ClxDerived!" << endl;
                } ;
18
19          void DoSomething() { cout<<"Do something in class ClxDerived!"<<endl;};
20  };
21
22
23  int main (int argc, char ** argv) {
24      ClxBase *pTest = new ClxDerived;
25      pTest->DoSomething();
26      delete pTest;
27
28      cout << endl << "next " << endl;
29
30      ClxDerived my;
31      my.DoSomething();
32
33      cout << "ok, default" << endl;
34      return 0;
35  }
```

编译运行代码：

```
$ g++ virtualdestructure.cpp -o v
$ chmod a+x v
$ ./v
```

代码 11: ~ClxBase() 运行结果

```
1   Do something in class ClxDerived!
2
3   next
4   Do something in class ClxDerived!
5   ok, default
6   Output from the destructor of class ClxDerived!
```

代码 12: virtual ~ClxBase() 运行结果

```
1  Do something in class ClxDerived!
2  Output from the destructor of class ClxDerived!
3
4  next
5  Do something in class ClxDerived!
6  ok, default
7  Output from the destructor of class ClxDerived!
```

类 ClxDerived 的析构函数根本没有被调用！一般情况下类的析构函数里面都是释放内存资源，而析构函数不被调用的话就会造成内存泄漏。当然，如果在析构函数中做了其他工作的话，那你的所有努力也都是白费力气。

所以，文章开头的那个问题的答案就是——这样做是为了当用一个基类的指针删除一个派生类的对象时，派生类的析构函数会被调用。

当然，并不是要把所有类的析构函数都写成虚函数。因为当类里面有虚函数的时候，编译器会给类添加一个虚函数表，里面来存放虚函数指针，这样就会增加类的存储空间。所以，只有当一个类被用来作为基类的时候，才把析构函数写成虚函数。

参考文献

[1] android 的 binder 驱动进程, 线程, 线程池,

[2] android 的 binder 驱动 7 - fd 和死亡通知

[3] android 的 binder 驱动 6 - 引用计数

[4] android 的 binder 驱动 5 - buffer

[5] android 的 binder 驱动 4 - BINDER_WRITE_READ 和 transaction

[6] Android Bander 设计与实现 - 设计篇

[7] drivers/staging/android/: binder.c, binder.h

[8] frameworks/base/include/binder/: IBinder.h

[9] frameworks/base/libs/binder/: Binder.cpp BpBinder.cpp

[10] frameworks/base/include/utils

[11] frameworks/base/libs/utils

[12] frameworks/base/cmds/servicemanager

[13] frameworks/base/cmds/runtime/ServiceManager.cpp

索引