<https://blog.csdn.net/luoshengyang/article/details/6629298>

<https://blog.csdn.net/universus/article/details/6211589>

Binder使用了面向对象的思想来描述作为访问接入点的Binder及其在Client中的入口：Binder是一个实体位于Server中的对象，该对象提供了一套方法用以实现对服务的请求，就象类的成员函数。遍布于client中的入口可以看成指向这个binder对象的‘指针’，一旦获得了这个‘指针’就可以调用该对象的方法访问server。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。

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。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。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。Client也利用保留的0号引用向SMgr请求访问某个Binder., 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。

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。

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。

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。

1. Service Manager成为Binder守护进程

一是打开Binder设备文件；二是告诉Binder驱动程序自己是Binder上下文管理者，即我们前面所说的守护进程；三是进入一个无穷循环，充当Server的角色，等待Client的请求。

Binder通信机制使用句柄来代表远程接口。Open函数创建一个struct binder\_proc数据结构来保存打开设备文件/dev/binder的进程的上下文信息。binder\_proc分别挂会四个红黑树下，用两种方式来组织红黑树，一种是以句柄作来key值来组织，一种是以引用的实体节点的地址值作来key值来组织。

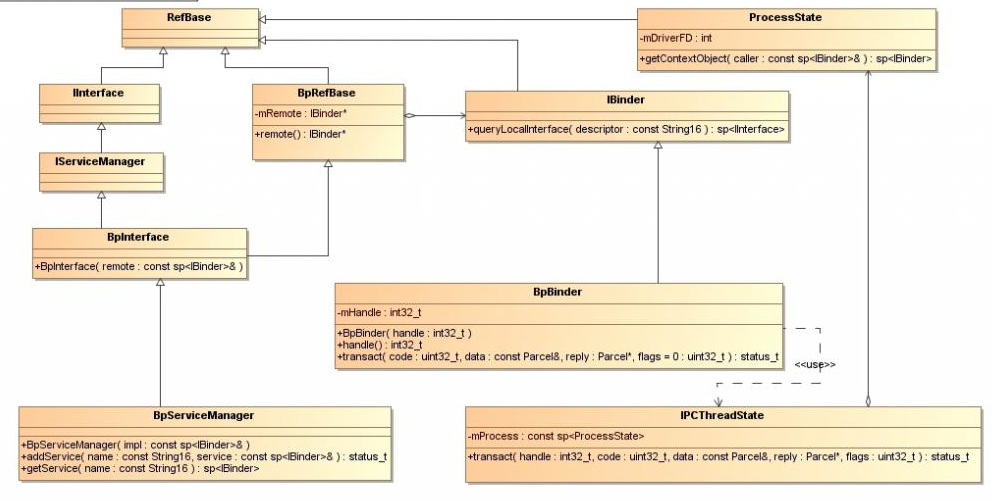
在Linux中，struct vm\_area\_struct表示的虚拟地址是给进程使用的，而struct vm\_struct表示的虚拟地址是给内核使用的，它们对应的物理页面都可以是不连续的。

这里为什么会同时使用进程虚拟地址空间和内核虚拟地址空间来映射同一个物理页面呢？这就是Binder进程间通信机制的精髓所在了，同一个物理页面，一方映射到进程虚拟地址空间，一方面映射到内核虚拟地址空间，这样，进程和内核之间就可以减少一次内存拷贝了，提到了进程间通信效率。

1. 获得Service Manager接口之路

对于普通的Server来说，Client如果想要获得Server的远程接口，那么必须通过Service Manager远程接口提供的getService接口来获得，这本身就是一个使用Binder机制来进行进程间通信的过程。而对于Service Manager这个Server来说，Client如果想要获得Service Manager远程接口，却不必通过进程间通信机制来获得，因为Service Manager远程接口是一个特殊的Binder引用，它的引用句柄一定是0。

BpBinder类通过IPCThreadState类来和Binder驱动程序并互，而IPCThreadState又通过它的成员变量mProcess来打开/dev/binder设备文件，mProcess成员变量的类型为ProcessState。ProcessState类打开设备/dev/binder之后，将打开文件描述符保存在mDriverFD成员变量中，以供后续使用。

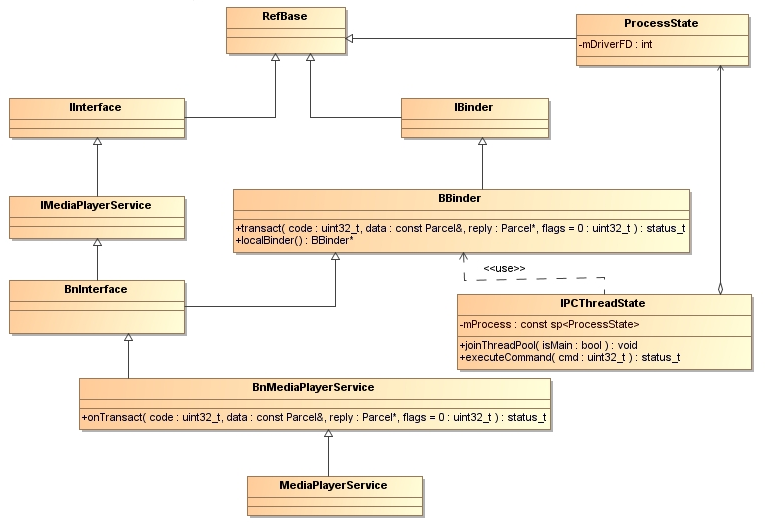


创建Service Manager远程接口的过程了，最终目的是要创建一个BpServiceManager实例，并且返回它的IServiceManager接口。

  对Server来说，就是调用IServiceManager::addService这个接口来和Binder驱动程序交互了，即调用BpServiceManager::addService 。而BpServiceManager::addService又会调用通过其基类BpRefBase的成员函数remote获得原先创建的BpBinder实例，接着调用BpBinder::transact成员函数。在BpBinder::transact函数中，又会调用IPCThreadState::transact成员函数，这里就是最终与Binder驱动程序交互的地方了。

三、Binder中的Server启动过程

BnMediaPlayerService是一个Binder Native类，用来处理Client请求的。



实际上，BnMediaPlayerService并不是直接接收到Client处发送过来的请求，而是使用了IPCThreadState接收Client处发送过来的请求，而IPCThreadState又借助了ProcessState类来与Binder驱动程序交互。IPCThreadState接收到了Client处的请求后，就会调用BBinder类的transact函数，并传入相关参数，BBinder类的transact函数最终调用BnMediaPlayerService类的onTransact函数，于是，就开始真正地处理Client的请求了。

因此binder->localBinder返回一个BBinder指针，如果进程间传输的数据间带有Binder对象的时候，Binder驱动程序需要作进一步的处理，以维护各个Binder实体的一致性。

那么Binder驱动程序是怎么处理这些Binder对象的？

在第二中，这里又调用了IPCThreadState::transact进执行实际的操作。函数首先调用writeTransactionData函数准备好一个struct binder\_transaction\_data结构体变量，这个是等一下要传输给Binder驱动程序的。初始化本地变量tr，将tr的内容保存在IPCThreadState的成员变量mOut中。主要调用了talkWithDriver函数来与Binder驱动程序进行交互，把这个Binder实体MediaPlayerService交给target\_proc，也就是Service Manager来管理，也就是说Service Manager要引用这个MediaPlayerService了，于是通过binder\_get\_ref\_for\_node为MediaPlayerService创建一个引用，并且通过binder\_inc\_ref来增加这个引用计数，防止这个引用还在使用过程当中就被销毁。

 之前分配了一个待处理事务t和一个待完成工作项tcomplete，并执行初始化工作， 注意，这里的事务t是要交给target\_proc处理的，在这个场景之下，就是Service Manager了。就是在Service Manager的进程空间中分配一块内存来保存用户传进入的参数了。

最后，把待处理事务加入到target\_list列表中去： 并且把待完成工作项加入到本线程的todo等待执行列表中去：目标进程有事情可做了，于是唤醒它：   这里就是要唤醒Service Manager进程了。

Binder进程间通信机制的精髓所在：

1. tr.data.ptr.buffer = (void \*)t->buffer->data + proc->user\_buffer\_offset;
2. tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data\_size, sizeof(void \*));

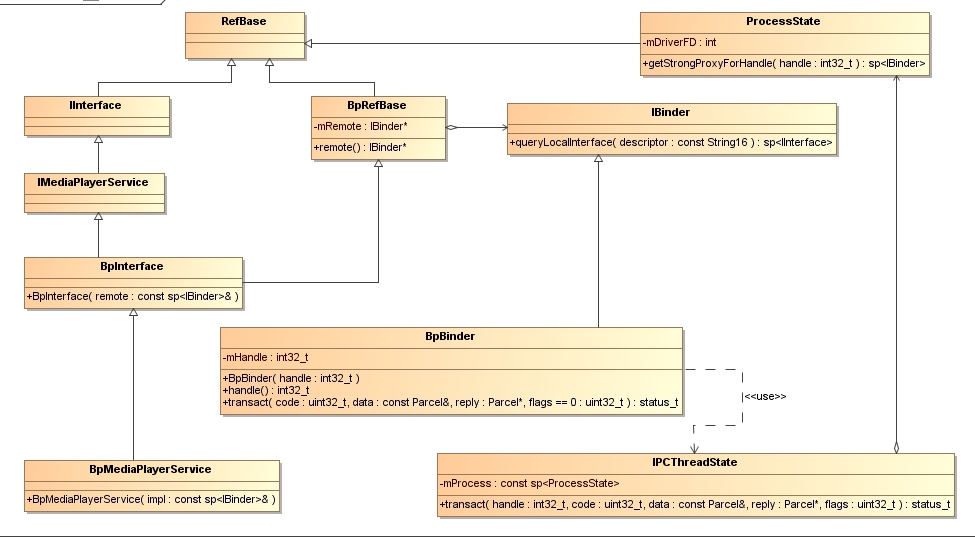
 t->buffer->data所指向的地址是内核空间的，现在要把数据返回给Service Manager进程的用户空间，而Service Manager进程的用户空间是不能访问内核空间的数据的，所以这里要作一下处理。怎么处理呢？我们在学面向对象语言的时候，对象的拷贝有深拷贝和浅拷贝之分，深拷贝是把另外分配一块新内存，然后把原始对象的内容搬过去，浅拷贝是并没有为新对象分配一块新空间，而只是分配一个引用，而个引用指向原始对象。Binder机制用的是类似浅拷贝的方法，通过在用户空间分配一个虚拟地址，然后让这个用户空间虚拟地址与 t->buffer->data这个内核空间虚拟地址指向同一个物理地址，这样就可以实现浅拷贝了。

   和前面一样，分别把t和tcomplete分别放在target\_list和thread->todo队列中，这里的target\_list指的就是最初调用IServiceManager::addService的MediaPlayerService的Server主线程的的thread->todo队列了，而thread->todo指的是Service Manager中用来回复IServiceManager::addService请求的线程。最后，唤醒等待在target\_wait队列上的线程了，就是最初调用IServiceManager::addService的MediaPlayerService的Server主线程了。上面讲到调用IServiceManager::addService的MediaPlayerService的Server主线程被唤醒了，于是，重新执行binder\_thread\_read函数：

最终都是调用了IPCThreadState::joinThreadPool函数，这个函数最终是在一个无穷循环中，通过调用talkWithDriver函数来和Binder驱动程序进行交互，实际上就是调用talkWithDriver来等待Client的请求，然后再调用executeCommand来处理请求，而在executeCommand函数中，最终会调用BBinder::transact来真正处理Client的请求，最终会调用onTransact函数来处理。在这个场景中，BnMediaPlayerService继承了BBinder类，并且重载了onTransact函数，因此，这里实际上是调用了BnMediaPlayerService::onTransact函数。

四、Binder中的Client获得Server远程接口

我们在获取Service Manager远程接口时，最终是获得了一个BpServiceManager对象的IServiceManager接口。类似地，我们要获得MediaPlayerService的远程接口，实际上就是要获得一个称为BpMediaPlayerService对象的IMediaPlayerService接口。现在，我们就先来看一下BpMediaPlayerService的类图：



BpMediaPlayerService的构造函数有一个参数impl，它的类型为const sp<IBinder>&，从上面的描述中，这个实际上就是一个BpBinder对象。这样，要创建一个BpMediaPlayerService对象，首先就要有一个BpBinder对象。再来看BpBinder类的构造函数，它有一个参数handle，类型为int32\_t，这个参数的意义就是请求MediaPlayerService这个远程接口的进程对MediaPlayerService这个Binder实体的引用了。因此，获取MediaPlayerService这个远程接口的本质问题就变为从Service Manager中获得MediaPlayerService的一个句柄了。

通过sm->getService接口来不断尝试获得名称为“media.player”的Service，即MediaPlayerService。为什么要通过这无穷循环来得MediaPlayerService呢？因为这时候MediaPlayerService可能还没有启动起来，所以这里如果发现取回来的binder接口为NULL，就睡眠0.5秒，然后再尝试获取，这是获取Service接口的标准做法。

通过Parcel::writeInterfaceToken往data写入一个RPC头，就是写往data里面写入了一个整数和一个字符串“android.os.IServiceManager”  Service Manager来处理CHECK\_SERVICE\_TRANSACTION请求之前，会先验证一下这个RPC头，看看是否正确。接着再往data写入一个字符串name，这里就是“media.player”了。

ProcessState会把使用过的Binder远程接口（BpBinder）缓存起来，这样下次从Service Manager那里请求得到相同的句柄（Handle）时就可以直接返回这个Binder远程接口了，不用再创建一个出来。

intr = new BpMediaPlayerService(new BpBinder(handle));

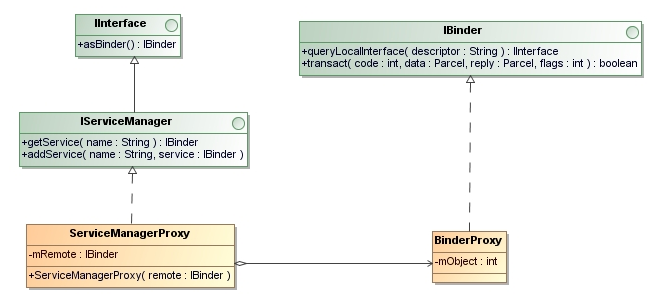
   因此，我们最终就得到了一个BpMediaPlayerService对象，达到我们最初的目标。 有了这个BpMediaPlayerService这个远程接口之后，MediaPlayer就可以调用MediaPlayerService的服务了。

五、Binder机制在应用程序框架层的Java接口

应用程序框架中的基于Java语言的Binder接口是通过JNI来调用基于C/C++语言的Binder运行库来为Java应用程序提供进程间通信服务的了。

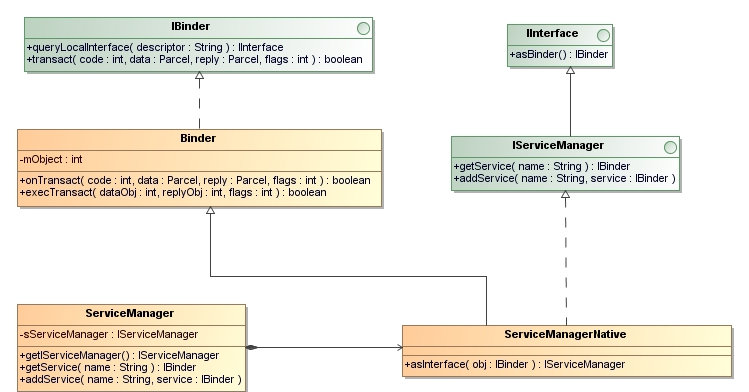
1. **获取Service Manager的Java远程接口**

我们要获取的Service Manager的Java远程接口是一个ServiceManagerProxy对象的IServiceManager接口



 再来看一下是通过什么路径来获取Service Manager的Java远程接口ServiceManagerProxy的。在调用ServiceManagerNative.asInterface函数之前，首先要通过BinderInternal.getContextObject函数来获得一个BinderProxy对象。BinderInternal.getContextObject是一个JNI方法，它实现在frameworks/base/core/jni/android\_util\_Binder.cpp文件中，ProcessState::self()->getContextObject函数返回一个BpBinder对象，它的句柄值是0，接着调用javaObjectForIBinder把这个BpBinder对象转换成一个BinderProxy对象。最终以这个BinderProxy对象为参数创建一个ServiceManagerProxy对象。

就是在Java层，我们拥有了一个Service Manager远程接口ServiceManagerProxy，而这个ServiceManagerProxy对象在JNI层有一个句柄值为0的BpBinder对象与之通过gBinderProxyOffsets关联起来。



 ServiceManager类有一个静态成员函数getIServiceManager，它的作用就是用来获取Service Manager的Java远程接口了，而这个函数又是通过ServiceManagerNative来获取Service Manager的Java远程接口的。