Android Binder攻关

|  |  |  |
| --- | --- | --- |
| 版本号 | 描述 | 日期(更新日期)/更新人 |
| 1.0 | Android Binder技术 | 2018/10/16 刘志保 |
|  |  |  |
| Github地址 | https://github.com/MMLoveMeMM/AngryPandaBinder.git | |

预备知识:

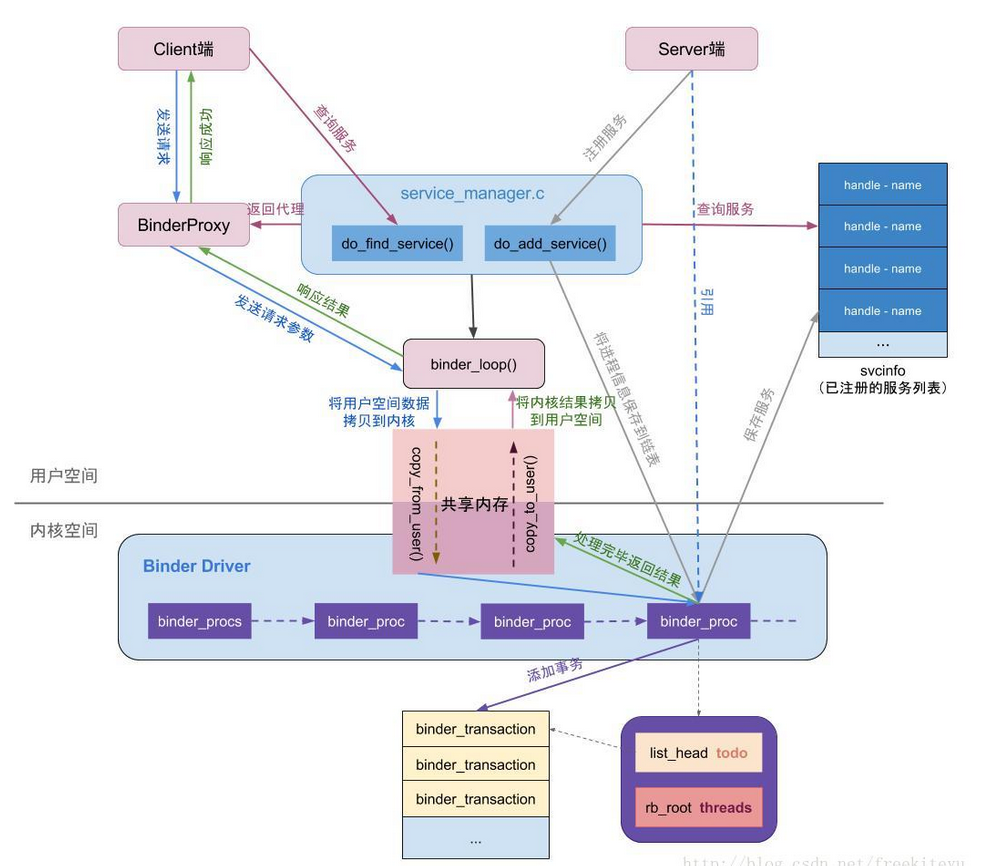
<1> : C++面向对象,以及宏定义,模板,指针转换;

<2> : Android基本架构;

<3> : C/S模型;

<4> : Android跨进程的种类方式;

先看看:[ https://blog.csdn.net/freekiteyu/article/details/70082302]



先接着” Android Service那些事.docx”文档.

分析思路:

<1> : 分析平时应用APP中AIDL自动生成的java文件;

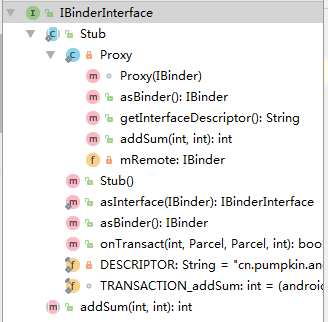
<2> : 自己根据上面的结构图,使用Java开发一个Binder跨进程服务C/S;

<3> : 自己使用C++模仿Java开发一个Binder跨进程服务C/S;

<4> : 自己使用C++模仿MediaPlayerService的Binder跨进程服务;

能够一步一步模仿出来,对熟悉整个跨进程调度过程起到非常大的辅助作用.

看看AngryPandaBinder工程代码,看看平时我们工程中使用aidl的时候是如何调用的.写了一个addSum作为样例.aidl都已经写好了,并build,在build目录下找到IBinderInterface.java文件.类结构如下:



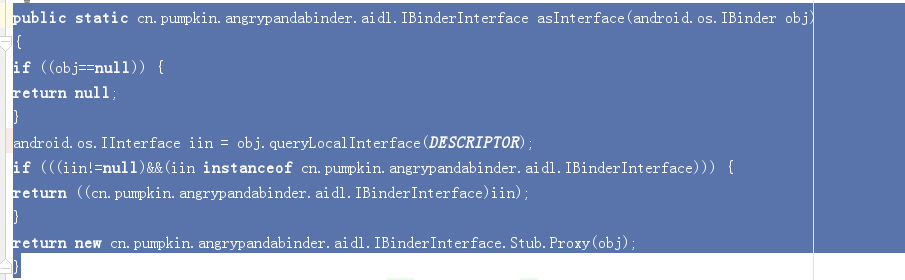
他们中的类关系:

IBinderInterface **extends** android.os.IInterface;

**public static abstract class** Stub **extends** android.os.Binder **implements** cn.pumpkin.angrypandabinder.aidl.IBinderInterface;

**private static class** Proxy **implements** cn.pumpkin.angrypandabinder.aidl.IBinderInterface

从上面继承实现关系来看,完全没有伦理道德,接口类中有两个子类,一个是public的,一个是private的.它们共同的特点就是implements了IBinderInterface接口(而这个接口父类是IInterface),那么接口就有一个实现接口,一个调用接口.实现接口就是在抽象类Stub中,而调用接口就在代理类Proxy中,这个代理类居然没有直接提供给应用,而是通过静态接口返回:



所以应用层实际都是通过这个Proxy类进行调用接口的.这个Proxy在构造方法时将从服务Service创建的Binder传入,对外提供addSum(…)接口调用,但是隐藏了Binder部分,实际Proxy里面实际操作是利用Binder的Transact方法将addSum发送给Service去运行.

下面是应用层的基本关系:

将创建的Binder传递给客户端

客户端

IInterface

Service创建Binder,并且映射到binder驱动

提供调用

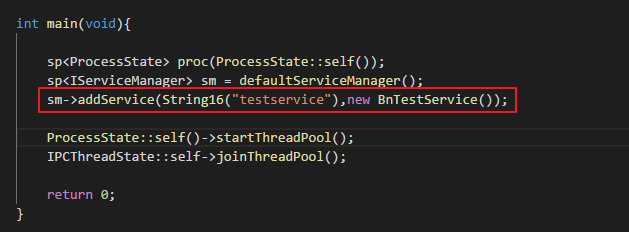
Service端实现

再来看看几个java和c++代码直接调用的非常简单的例子,做后面的铺垫:

参考工程AngryPandaBinder中的javabinder,cppbinder1,cppbinder2模块.

这里主要介绍一下cppbinder2中:

注册进去的是

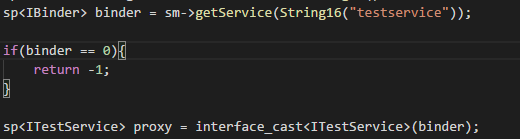


上面的代码如何写出来的,可以参考系统媒体播放的源代码main\_mediaserver.cpp .

红框第二个参数:



继承(实现)的是BnInterface<ITestService>的接口.但是获取出来的却是:

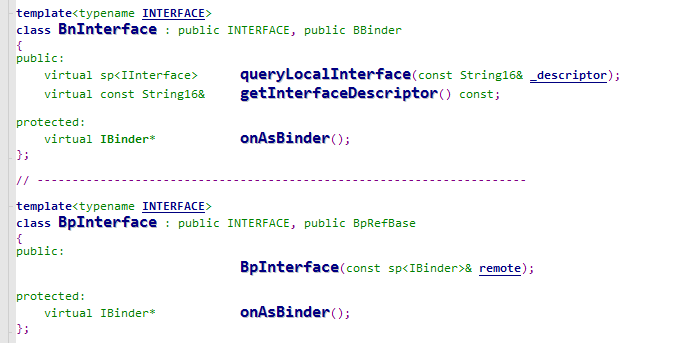


其实这个proxy准确来说是BpTestService:



是不是感觉添加进去的是Bn\*\*\*,出来却是Bp\*\*\*?

这里有几个重要的转正需要看熟IInterface.h



首先class BnTestService:public BnInterface<ITestService>

替换上面的模板,即

class BnTestService:public ITestService,public BBinder

同理:

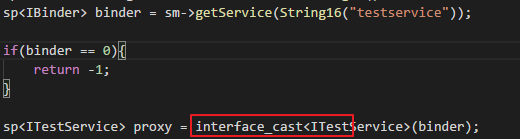
class BpTestService:public BpInterface<ITestService>

class BpTestService:public ITestService,public BpRefBase

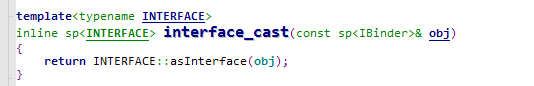
下面的宏是给出为什么可以Bn->Bp:



那么怎么样将Bp转成父(基)类:



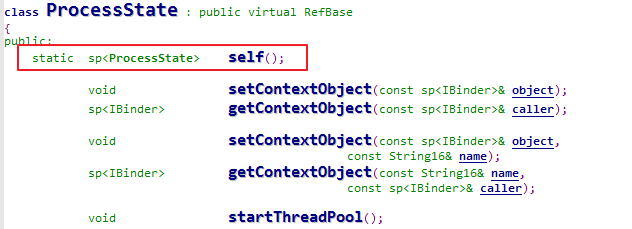
可以看看interface\_cast这个源码,你可以理解为指针转换,但是实际不仅仅是指针转换,准确说应该类地址对齐(将子类首地址对齐到基类[父类]):



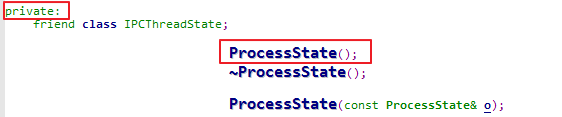
和java aidl自动生成的asInterface对比一下,一模一样.

下面还有两句程序:

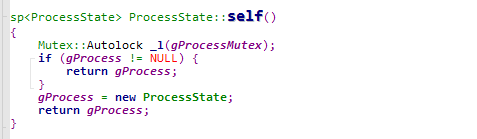
ProcessState::self()->startThreadPool();



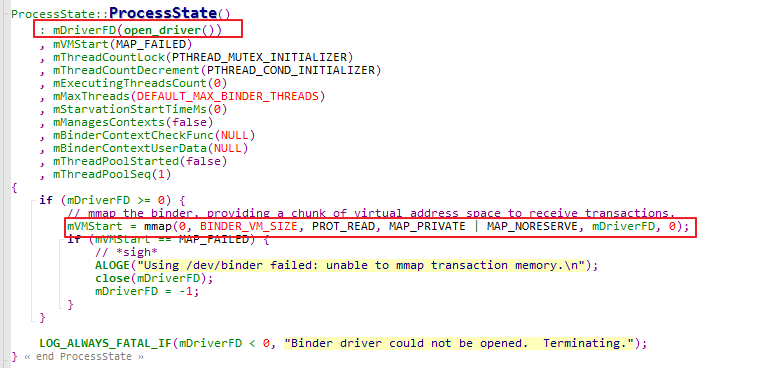
其中self()函数是个静态函数,并且ProcessState的构造函数是个私有的:



其实就是个单实例类.构造函数在self()中调用.



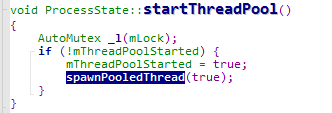
构造函数做了哪些事?



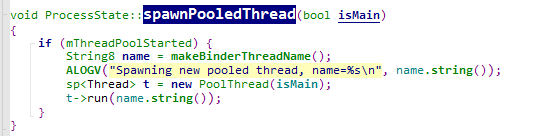
<1> : 打开binder驱动,参见第一个红框处(写的够隐蔽):mDriverFD(open\_driver());

<2> : mmap分配一块通信虚拟内存空间,用于交换数据等信息.

->startThreadPool();



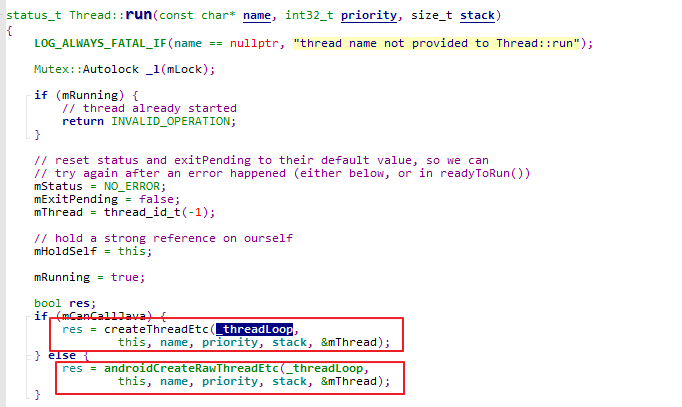
然后里面启动一个主线程(注意上mThreadPoolStarted=true):



其中PoolThread:



这个地方要注意threadLoop()何时调用,PoolThread继承Thread(android源代码system/core/libutils下的Threads.cpp),线程开始run()时:

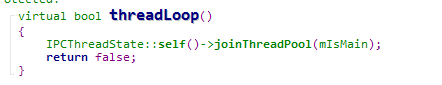


在继续看:

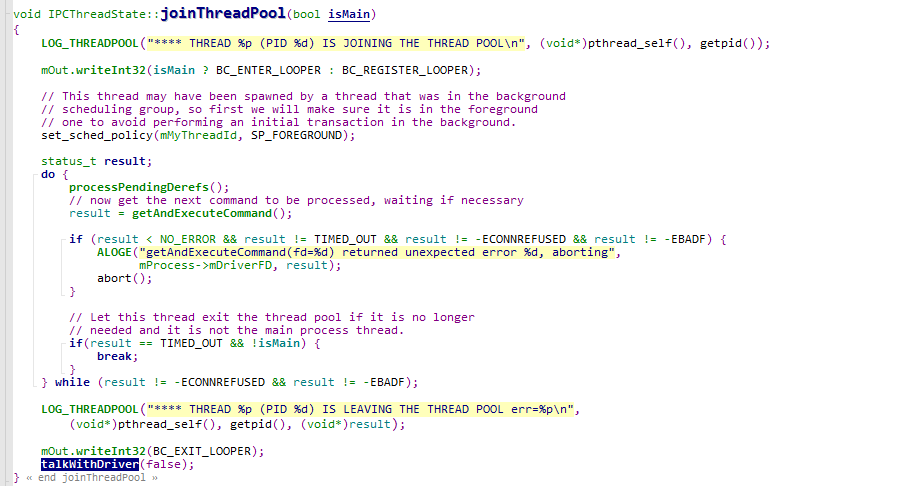


发现threadLoop()调用了,这个在Thread中实现,并且在run()函数调用的时候调用.

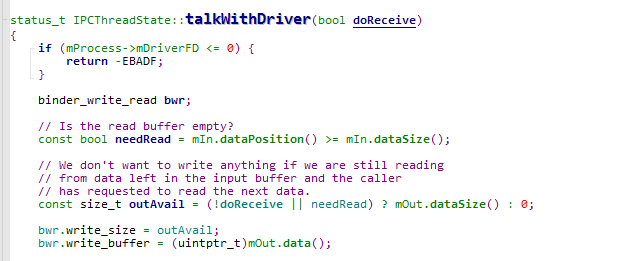
回到ProcessState.cpp中PoolThread类中如何实现threadLoop函数的:



终于见到IPCThreadState类了吧,这个类也是个单实例.这里的mISMain一直为true.



这个是在当前进程中去不断轮询是否有数据需要发送,注意do/while,死等任务哈,一旦有数据或者任务了,就调用talkWithDriver去与binder驱动进行通信.

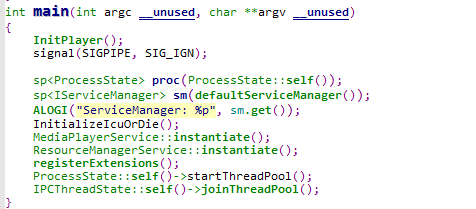


如果不需要返回doReceive就是false,如果有返回值就是true,真个流程基本就是这样的.

[另外一个参考: https://blog.csdn.net/gykimo/article/details/8901192]

根据上面的再来看看系统源码MediaPlayerService模块的AIDL通信,由浅入深,下面就会非常简单了!

Server比如xxxService,那么这个是提供某个模块的服务模块,我们要操作系统xxx模块和使用xxx的功能,那么一般需要找xxxService来帮忙,但是xxxService是一个独立进程,在系统启动的init里面启动了(具体参考android启动过程),要和xxxService进行沟通就需要一点手段,当然前提是xxxService需要在启动的时候同时留个后门,方便应用层调用.这个后门是什么呢?就是IBinder(即IBinder.h,系统binder目录下的代码要看一遍哈)的子类,例如MediaPlayerService(后面都是以MediaPlayer这个作为熟悉的过程),先大致看一下MediaPlayerService启动,打开main\_mediaplayer.cpp文件:



重点来看看这个”后面”是如何开的,看看关系:

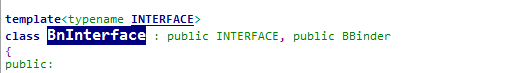
关系一:



继承关系二:



继承关系三:

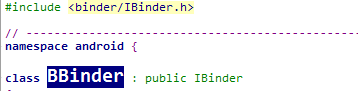


注释 :这个地方是个模板.那么BnInterface<IMediaPlayerService>代码模板类中完整代码即:

Class BnInterface : public IMediaPlayerService,public BBinder …

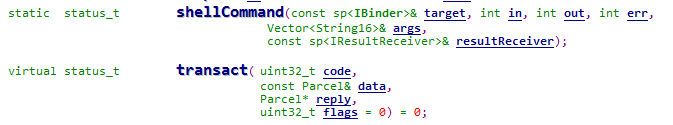
注意:不是IMediaPlayerService继承IBinder,IMediaPlayerService是继承RefBase的.即I\*\*\*的都是继承RefBase接口的,客户端和服务端都是对称的.

继承关系四:



继承关系五:

IBinder.h : 主要两个通信接口如下:



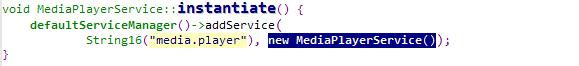
经过这么的继承关系,有什么用呢?

第一个 : IMediaPlayerService接口中是大量操作MediaPlayerService里面核心功能的接口,这些接口需要在子类被实现,以及binder的通信接口;

第二个 : 将拥有大量具有实际操作能力的接口类作为一个IBinder的一个子类保存到Binder分配的虚拟内存里面.Binder中保存了一份对应关系表单,简图如下:

|  |  |
| --- | --- |
| media.player | new MediaPlayerService() |

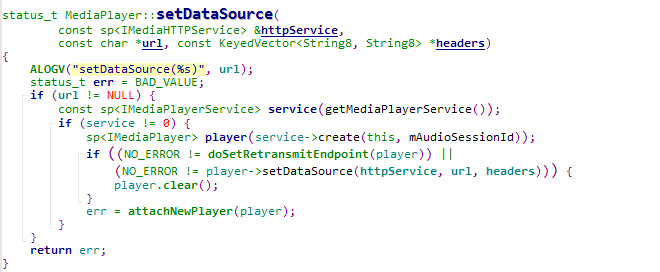
关系添加,添加和获取都是由系统IServiceManager类进行管理的:



看到了吗?它将new MediaPlayerService()这个对象(是对象,对象可是有实际操作能力的)保存到Binder里面去了,这个添加是在前面main\_mediaplayer.cpp的main启动的时候就添加了.

那另外一端的客户端(应用)如果去调用这个服务进行操作呢?

MediaPlayer.cpp



然后看看getMediaPlayerService()是如何获取:



注意看sm->getService(String16(“media.player”)),这个也是通过ServiceManager(具体参考service\_manager.c文件)管理类获取的,与前面的addService(String16(“media.player”,…))一一对应了.getService返回的是sp<IBinder>的指针,这个指针一般是指向的是IBinder的子类,比如MediaPlayerService就是IBinder的子类,这个关系一定要理清楚哈,不然addService进去的是new MediaPlayerService() 子类,getService出来的是sp<IBinder>,看起来似乎完全没关系,实际上就是指向new MediaPlayerService()子类,这个地方千万不能够断片呀.这个通过getService出来的sp<IBinder>是可以通过指针转换的(了解c++指针转换).这个地方没有转过弯,后面所有的源码都不用看了,一定看不懂的.拿到这个可是重点,因为你要让MediaPlayerService帮你做事,都需要这个返回的IBinder接口帮你做事,经过指针转换,IBinder的子类:

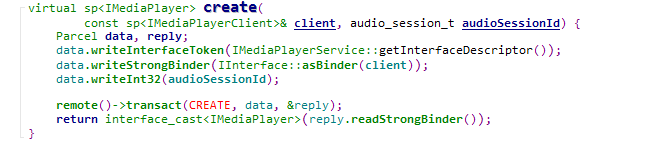


子类其实是sp<IMediaPlayerService>,这不就是MediaPlayerService的父类吗?

那这样有了,不就可以和MediaPlayerService进行交互了吗?通过这个接口类.

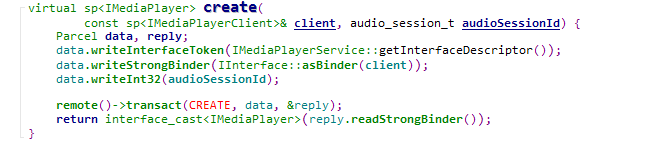


具体操作如下IMediaPlayerService.cpp代码:

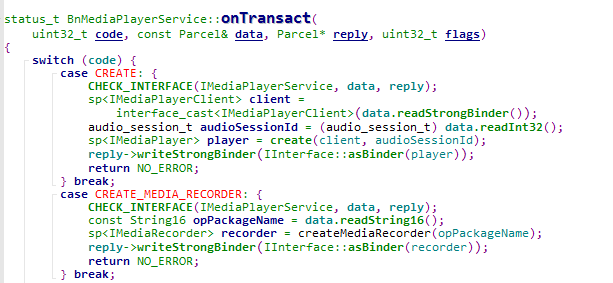


看返回,由于IMediaPlayer也是继承IBinder的,所以做一下指针转换,而MediaPlayer中全部是通过IMediaPlayer接口调用操作的,具体传输这里暂时不介绍,先搞懂他们这些关系户.

关系户搞清楚了,就看看如果交互通信.接着上面的程序,需要创建一个对象来播放媒体文件,创建代码如下,参考IMediaPlayerService.cpp:其中transact函数是用来发送的.

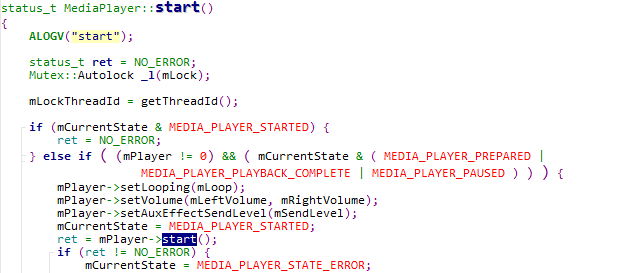


通过”发送”一个”CREATE”指令,并且附带数据过去,给谁呢?接受者是BnMediaPlayerService类,

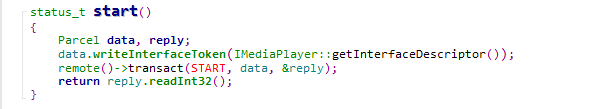


BnMediaPlayerService类不就就是服务Server的代理处理事务的类吗?

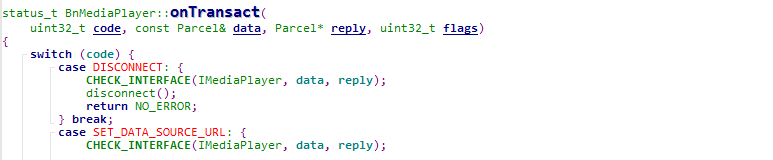
程序还是容易懂的,收到数据后,根据传递过来的IMediaPlayerClient,创建IMediaPlayer对象player对象,然后作为replay返回给上面发送的.这样在客户端那边就有了mPlayer这个对象了,通过这个mPlayer就可以调用准备开始播放等接口,如何:



根据上面,说白了,mPlayer就是IMediaPlayer的一个实例.然后看看start()具体操作(参考IMediaPlayer.cpp):



START指令发给谁处理呢?发给了BnMediaPlayer类.



Transact(…)和onTransact(…)可以说是一对,一个发送一个接收处理.后面的流程与最前面介绍的完全一致.