**先大概描述下Android中S/C架构的基本面貌**

由于采用linux内核，所以用户空间中得进程地址空间都是独立的，如果用户空间进程AB之间想进行通信,可以先将用户空间进程A要发送信息发送到内核，由于内核中得地址空间对于每个进程都是一样的，所以用户空间进程B可以读取内核收到的信息。

Android中得binder机制就是将这种调用进行了封装,将用户空间的进程分为三类，client,service,ManagerService

其中的ManagerService作用是记录所有的service信息，client访问service前先到ManagerService处去查询有无此服务

**常用有如下情形1：添加service服务**

**过程**

第一步：获取ManagerService服务代理BpBinder(0)和BpManagerService

第二步：通过**BpManagerService**中得addService方法将服务的一些信息打包，然后通过**BpBinder(0)**将信息发送到内核，等待ManagerService进程来读取，因为ManagerService进程中有一个Loop一直读取内核信息，所以很快就获取了service发给内核的信息，ManagerService进程就将service服务的一些信息添加到维护的列表中，将添加的结果发送给内核，然后service等到内核有回复后返回，将返回的信息解码分析。

第三步：执行一个Loop来从内核读取client的请求

**代码**

如下是添加一个service的代码

第一步：

**sp<ProcessState> proc(ProcessState::self());**

**sp<IServiceManager> sm = defaultServiceManager();**

第二步：

**defaultServiceManager()->addService(String16("media.player"), new MediaPlayerService());**

第三步：

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

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

**代码分析**

**sp<ProcessState> proc(ProcessState::self());**

创建一个ProcessState实例

ProcessState::mDriverFD=open\_driver()

ProcessState::mVMstart = mmap(…,mDriverFD,…);

mmap(0,…,mDriverFD,0);//只是让通信更快，可以当作没有，直接用read/write(mDriverFD);

**sp<IServiceManager> sm = defaultServiceManager();**

主要执行如下代码

sp<IServiceManager> gDefaultServiceManager = interface\_cast<IServiceManager>(PorcessState::self()->getContextObject(NULL));

其中PorcessState::self()->getContextObject(NULL)就是执行PorcessState::getStrongProxyForHandle(0);其主要是根据给定的参数(此处为0)创建对应的sp<IBinder> b = new BpBinder(0);

这边先说明interface\_case这个模版类，这个模版类作用是将服务xxxService对应的**BpBinder(x)**转化为对应的**BpxxxService** 如下：

sp<IxxxService> **BpxxxService** = interface\_cast<IxxxService>(new **BpBinder(x)**);

其本质是调用给定类型参数的asInterface方法，也就是IxxxService::asInterface(new BpBinder(x)), **IxxxService继承public IInterface**

IxxxService用宏DECLARE\_META\_INTERFACE(xxxService)和IMPLEMENT\_META\_INTERFACE(xxxService,”android.os.IxxxService”)来实现asInterface方法，展开后如 new BpxxxService(new BpBinder(x))，这样就从BpBinder(x)得到了BpxxxService

此处new BpxxxService(new BpBinder(x))构造函数中会将new BpBinder(x)给BpxxxService的基类BpRefBase的类型为IBinder \*成员mRemote也就是mRemote=new BpBinder(x);

如本例中给定的类型参数是IServiceManager，所以上述等价于

sp<IServiceManager> gDefaultServiceManager = IServiceManager::asInterface(new BpBinder(0)); IServiceManager继承public IInterface

其中用宏DECLARE\_META\_INTERFACE(ServiceManager) 和IMPLEMENT\_META\_INTERFACE(ServiceManager,”android.os.IServiceManager”)来实现asInterface方法,这个asInterface方法最终执行new BpServiceManager(new BpBinder(0));

也就是**sp<IServiceManager> sm = new BpServiceManager(new BpBinder(0));**

由此得到BpBinder(0)和BpServiceManager

从这里可以得出，如果自己创建服务**xxxService**,按如下步骤

创建类 class **IxxxService**:publice IInterface

{

DECLARE\_META\_INTERFACE(xxxService);//声明asInterface方法

virtual function();//虚函数，由BpxxxService实现

…

}

IMPLEMENT\_META\_INTERFACE(xxxService,”android.os.IxxxService”);//定义asInterface

展开相当于sp<IxxxService> IxxxService::asInterface(BpBinder){

return new BpxxxService

}

当有这个服务的Bpbinder(x)后可通过如下获取BpxxxService

sp<IxxxService> sm = new BpxxxService (new BpBinder(x));

注意:

1:BpxxxService类型为sp<IxxxService>

2:BpBinder(x)相当于xxxService对应binder实体的代理

**3:客户端为什么要通过BpBinder(X)再创建一个Bp**xxxService**呢？因为Bp**xxxService**对象的基类中的一个mRemote变量就是Bpbinder(x)，那为什么不直接用BpBinder(x)呢？因为Bp**xxxService**对象中还实现了一些BpBinder(x)没有的业务逻辑(实现基类I**xxxService**中的方法)如** BpServiceManager中提供了addService**方法，这些方法在服务端有对应的服务，如addService方法将请求的命令”addService”和数据打包成Parcel类型，然后通过BpBinder(x)发送给**xxxService，**其实先发给驱动中binder设备，**xxxService**服务中会有Loop循环一直读取binder设备的消息，读取消息后解析成命令+数据，然后根据命令如”addService”会调取对应的服务，然后将结果反馈给binder设备，客户端接收到binder设备的反馈后将返回的数据也解析成命令+数据，然后根据命令和数据执行相应的逻辑**

客户端 通过BpxxxService中方法将数据打包 -->Bpbinder(x)--> binder设备 --> 服务端Loop 获取数据解析成命令+数据-->调取命令指定的服务其中会涉及BBinder和BnxxxService，返回数据

客户端 解析返回的数据并执行相应逻辑 <--Bpbinder(x)<-- binder设备 <-- 服务端返回数据

由上可知，创建的服务还需创建BnxxxService和BpxxxService

class **BpxxxService** : public BpInterface<IxxxService>

{

…

vitural function(){

Parcel data,reply;

data.writeInterfaceToken(IxxxService::getInterfaceDescriptor());

data.writeInt32(pid);

remote()->transact(**命令码**，data,&reply);

}

}

class **BnxxxService**: public BnInterface<IxxxService>

{

public:

vitrual status\_t onTransact(uint32\_t code,const Parcel& data,Parcel\* reply,uint32\_t flags = 0);

}

IMPLEMENT\_META\_INTERFACE(xxxService,”android.xxxService.IxxxService”);

status\_t Bn**xxxService**::onTransact(uint32\_t code,const Parcel& data,Parcel\* reply,uint32\_t flags)

{

switch(code){

case 命令码：

CHECK\_INTERFACE(IxxxService,data,reply);

**function()//自己实现的逻辑**

break;

…

}

**defaultServiceManager()->addService(String16("media.player"), new MediaPlayerService());**

通过**BpServiceManager**(类型为sp<IServiceManager>)将数据打包成含”addService”命令，类型为**parcel** **data**的数据，调用BpBinder(0)这个对象的transact(),最终调到IPCThreadState::self()->transact(),

下面分析IPCThreadState，你暂时只需要知道每个线程有一个IPCThreadState实例，实例有属性mIn,mOut，其中mIn用来接受来自binder设备的parcel类型数据，mOut用来存储发往binder设备的parcel类型数据，IPCThreadState::self()->transact()最终完成与binder设备的交互，这个接口中先调用writeTransactionData把parcel类型的数据和int32\_t类型的命令码封装成**binder\_transaction\_data**类型数据放到IPCThreadState实例的mOut中去，接下来IPCThreadState::self()->transact()会调用IPCThreadState::self()->waitForResponse接口,其中会**调用talkWithDriver()与binder设备交换数据**(talkwithDriver主要是实现是ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr))，将返回来的数据存于IPCThreadState实例的mIn,然后**调用executeCommand()根据mIn中得命令码和数据来解析**，来调用各种业务逻辑

ServiceManager这个服务的Loop循环实现代码见下面：

struct binder\_state \*bs;

void \*svcmgr = BINDER\_SERVICE\_MANAGER;

bs = binder\_open(128\*1024);//应该是打开binder设备吧

binder\_become\_context\_manager(bs) //成为manager

svcmgr\_handle = svcmgr;

binder\_loop(bs, svcmgr\_handler);//Loop

new MediaPlayerService()，MediaPlayerService继承BnMediaPlayService，BnMediaPlayService继承BnInterface, BnInterface继承BBinder ,BBinder继承 IBinder,所以MediaPlayerService继承了IBinder,addService第二个参数类型是IBinder 实例一个MediaPlayService也就是实例一个**BnMediaPlaySerivce**

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

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

创建一个MediaPlayerService服务后（也就是创建一个BnMediaPlaySerivce)后，将其加入serviceManage维护的一个列表中，接下来，BnMediaPlaySerivce也该起到服务的作用，创建一个Loop等待客户端client的请求，所以应该也有一个循环，也就是如下两段代码的作用

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

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

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

其调用

sp<Thread> t = new PoolThread(isMain);isMain是TRUE

t->run(buf);//这句话会导致t->threadLoop调用，这里面执行了IPCThreadState::self()->joinThreadPool(mIsMain);

所以归根结底最后都调用**IPCThreadState::self()->joinThreadPool(mIsMain);**

这里面主要是执行一个循环，

do {

int32\_t cmd;

result = talkWithDriver();

result = executeCommand(cmd);

}

} while (result != -ECONNREFUSED && result != -EBADF);

**总结：**调用talkWithDriver()与binder设备交换数据(talkwithDriver主要是实现是ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr))，将返回来的数据存于IPCThreadState实例的mIn,然后调用executeCommand()根据mIn中得命令码和数据来解析，来调用自己(此处是**BnMediaPlaySerivce**)的各种业务逻辑，例如如果受到的命令是BR\_TRANSACTION则会调用reinterpret\_cast<BBinder\*>(tr.cookie)->transact(tr.code, buffer, &reply, tr.flags);结果调用**BnMediaPlaySerivce**的oNTransact方法

**这里面会根据获取的mIn中得**命令码和数据来调用不同的业务逻辑**,然后派发到派生类MediaService(自己创建的)的函数，由他们完成实际的工作**

**常用有如下情形2：client访问某个服务service**

**过程**

第一步：获取ManagerService服务代理BpBinder(0)和BpManagerService

第二步：通过**BpManagerService**中得getService方法将服务的一些信息打包，然后通过**BpBinder(0)**将信息发送到内核，等待ManagerService进程来读取，因为ManagerService进程中有一个Loop一直读取内核信息，所以很快就获取了client发给内核的信息，ManagerService进程将client想知道的东西发送给内核，client等到内核有回复后返回，将返回的信息解码分析已得知有无某个service服务，如果有这个service则返回的信息中有这个service的Bpbinder(x),client可以通过sp<Iservice> Bpservice = interface\_cast<Iservice>( Bpbinder(x));来获取BpService, BpService中有与service对应的服务

第三步：访问服务

client访问这些服务前先调BpService中对应特定服务其将数据及命令打包,通过Bpbinder(x)将信息发送到内核，应为service也有一个Loop一直在读取内核信息，所以service很快就获取了client发来的信息，解析后根据信息中得命令码将数据发送到Bnservice对象包含的方法中进行解析，然后将处理后的数据发送给内核，client等到内核的受到的消息后返回，将返回的消息解码然后处理

**代码**

如下是client获取service的的代码

第一步：

**sp<ProcessState> proc(ProcessState::self());**

**sp<IServiceManager> sm=defaultServiceManager();**

见上

第二步：

**sp<IBinder> binder = sm->getServices(String16(“media.player”));**

**sp<IMediaPlayService> sMediaPlayService = interface\_cast<IMediaPlayService>(binder);**

见上

第三步：