**getService-通过MediaPlayerService分析binder机制**

**1、getMediaPlayerService**

frameworks/av/media/libmedia/IMediaDeathNotifier.cpp

IMediaDeathNotifier::getMediaPlayerService()

{

sp<IServiceManager> sm = defaultServiceManager();//之前有分析过，返回BpServiceManager

sp<IBinder> binder;

binder = sm->getService(String16("media.player"));//从ServiceManager中取出MediaPlayerService在内核binder驱动中的binder引用描述符

DeathNotifier = new DeathNotifier();

binder->linkToDeath(sDeathNotifier);//分析怎样成功绑定死亡通知函数

sMediaPlayerService = interface\_cast<IMediaPlayerService>(binder)//得到BpMediaPlayerService

}

**2、getService(String16("media.player"))**

frameworks/native/libs/binder/IServiceManager.cpp

**2.1、MediaPlayer进程**

**2.1.1、sm->getService(String16("media.player"))即是调用BpServiceManager的getService：**

1、for循环调用checkService，最多5次，成功得到IBinder对象指针后跳出循环

2、checkService

data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());//写入描述符"android.os.IServiceManager"，校验用的。

data.writeString16(name);//写入待获取的service的name。

remote()->transact(CHECK\_SERVICE\_TRANSACTION, data, &reply);

相当于BpBinder(0) transact，接着走到IPCThreadState::self()->transact()，和分析addService时类似，与binder驱动通信之前的数据处理流程一模一样，这里简述一下，大致流程：

3、通过writeTransactionData把数据写入IPCThreadState的mOut中，进入waitForResponse，通过talkWithDriver中的系统调用ioctl与binder通信，数据经过几次封装最终填进bwr里。接下来MediaPlayer进程陷入内核态。cmd是BC\_TRANSACTION、BINDER\_WRITE\_READ

**2.1.2、binder驱动处理：**

函数调用流程binder\_ioctl、binder\_ioctl\_write\_read、copy\_from\_user、binder\_thread\_write、copy\_from\_user、binder\_transaction。

binder\_transaction工作：

1、根据0描述符，获取ServiceManager的binder实体，把ServiceManager进程设置为目标进程，分配一个待处理事务t给目标进程，分配一个待完成工作tcomplete到当前MediaPlayer进程，目的是告诉MediaPlayer，Binder驱动已经收到它的getService请求。最后，调用wake\_up\_interruptible()将Service Manager唤醒。

binder\_thread\_write 处理完后返回到binder\_ioctl\_write\_read继续，走到binder\_thread\_read()，读出它刚才的待完成工作tcomplete，处理完之后，bwr.read\_buffer中就包含了两个指令：BR\_NOOP和BR\_TRANSACTION\_COMPLETE。

层层返回，退出内核态，回到IPCThreadState talkWithDriver，清空mOut，读出返回的数据到mIn。接着退出到waitForResponse，它会处理mIn中的数据，这里有两个指令BR\_NOOP和BR\_TRANSACTION\_COMPLETE。和addService时处理的流程一样，最终MediaPlayer线程再次调用ioctl(BINDER\_WRITE\_READ)进行只读操作；执行到binder\_thread\_read()时，由于此时MediaPlayer线程的待处理工作队列为空，因此MediaPlayer线程会进入wait\_event\_freezable\_exclusive，中断等待状态。待ServiceManager进程处理完MediaPlayer的请求之后，就会将MediaPlayer线程唤醒。

**2.2、ServiceManager进程**

上一节MediaPlayer进程已经休眠，这一节看看ServiceManager进程中的处理，怎样唤醒MediaPlayer进程并把什么样的结果返回给它?

本来ServiceManager线程sleep在binder\_thread\_read中的wait\_event\_freezable\_exclusive，被MediaPlayer唤醒后，继续执行binder\_thread\_read。进入while循环，读取待处理事务t。

**回到《1-Bn-通过ServiceManager分析binder机制-ServiceManager服务启动》分析getService。**

**2.3、ServiceManager返回唤醒MediaPlayer**

从《1-Bn-通过ServiceManager分析binder机制-ServiceManager服务启动》可知，返回的是一个新建的MediaPlayerService的binder引用描述符。

看看驱动中被唤醒后的工作：和addService时，MediaPlayerService进程被唤醒的流程是一样的。把数据copy到用户空间后，函数调用就层层返回了。

**2.3.1、IPCThreadState::waitForResponse**

直接从这里分析，被唤醒后从talkWithDriver退出，读取mIn中命令和数据走到case BR\_REPLY，走到reply->ipcSetDataReference，把返回的数据保存在parcel（reply）中的mData里。

层层返回，到checkService的reply.readStrongBinder()

unflatten\_binder(ProcessState::self(), \*this, &val);

{

const flat\_binder\_object\* flat = in.readObject(false);

if (flat) {

switch (flat->type) {

case BINDER\_TYPE\_BINDER:

\*out = reinterpret\_cast<IBinder\*>(flat->cookie);

return finish\_unflatten\_binder(NULL, \*flat, in);

case BINDER\_TYPE\_HANDLE:还记得ServiceManager返回的数据吧，type是BINDER\_TYPE\_HANDLE，然后还有一个handle值。

\*out = proc->getStrongProxyForHandle(flat->handle);//会走到这里

return finish\_unflatten\_binder(

static\_cast<BpBinder\*>(out->get()), \*flat, in);

}

}

return BAD\_TYPE;

}

这个函数读出binder返回的数据类型flat\_binder\_object，里面保存了MediaPlayerService在驱动中的binder引用的handle和type，看看getStrongProxyForHandle (flat->handle)，进入ProcessState::getStrongProxyForHandle，第一次进来的话e->binder肯定是NULL，所以有b = new BpBinder(handle);e->binder = b;新建一个BpBinder，他的mHandle就是binder驱动中新建的MediaPlayerService的binder引用的描述符，用一个handle就把内核中的binder引用和用户空间的BpBinder关联起来了。

sm->getService(String16("media.player"))最终得到的是一个BpBinder，handle是该进程在驱动中保存的MediaPlayerService的binder引用描述符。

ServiceManager的内核binder引用描述符是0，其实它在binder驱动中没有binder引用，只有一个实体，通过0可以直接找到ServiceManager的全局变量Binder实体对象static struct binder\_node \*binder\_context\_mgr\_node。

**3、linkToDeath**

由上面分析结果，binder->linkToDeath(sDeathNotifier)中的binder为BpBinder(handle)

sDeathNotifier是一个DeathNotifier类对象。

class DeathNotifier: public IBinder::DeathRecipient

{

public:

DeathNotifier() {}

virtual ~DeathNotifier();

virtual void binderDied(const wp<IBinder>& who);

};

DeathNotifier继承IBinder::DeathRecipient（这是一个纯虚类），只有一个纯虚方法binderDied，被DeathNotifier覆盖，看看DeathNotifier的binderDied方法：靠IMediaDeathNotifier的静态数据成员sObitRecipients调用外部类IMediaDeathNotifier的died方法。外部类的构造函数调用addObitRecipient，把this指针保存在sObitRecipients容器中。外部类IMediaDeathNotifier的died方法被MediaPlayer类覆盖。如果调用sDeathNotifier的binderDied方法，最终会走到MediaPlayer。

看看BpBinder的linkToDeath：

新建Obituary类型的对象，把sDeathNotifier保存在成员recipient中，之后添加到BpBinder对象的数据成员：容器mObituaries里面。

第一次进来需要透过IPCThreadState发送注册消息到binder驱动。调用IPCThreadState requestDeathNotification。

**3.1、requestDeathNotification**

填充mOut，cmd为BC\_REQUEST\_DEATH\_NOTIFICATION，还有BpBinder的mHandle和BpBinder本身。

**3.2、flushCommands**

调用talkWithDriver(false)与binder驱动通信，bwr.write\_size = mOut.dataSize()，bwr.read\_size = 0

**3.2.1、把mOut填入binder\_write\_read bwr。**

然后ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr)使进程陷入内核态。

**3.2.2、看看binder驱动的处理**

binder\_thread\_write中的case BC\_REQUEST\_DEATH\_NOTIFICATION，binder\_get\_ref：根据传进来的handle得到驱动中的MediaPlayerService binder引用。

判断ref->death是否为NULL，不为空，说明注册过了，直接返回。

如果是NULL，新建一个binder\_ref\_death death

death->cookie = cookie;// cookie中保存的是BpBinder本身

ref->death = death;//这样ref的death就不为NULL了。

\*consumed = ptr - buffer;修改consumed的值后返回。

总结：linkToDeath就是传了一个用户空间的BpBinder指针下去，给binder驱动中对应的binder引用的death赋值，使它不为空。

**4、怎样回调到注册的sDeathNotifier**

binder驱动程序将设备文件/dev/binder的释放操作方法设置为函数binder\_release。Server进程在启动时，会调用函数open来打开设备文件/dev/binder。

**4.1、**在正常情况下，它退出时会调用函数close来关闭设备文件/dev/binder，这时候就会触发函数binder\_releasse被调用；

**4.2、**如果Server进程异常退出，即它没有正常关闭设备文件/dev/binder，那么内核就会负责关闭它，这个时候也会触发函数binder\_release被调用。

因此，Binder驱动程序就可以在函数binder\_release中检查进程退出时，是否有Binder本地对象在里面运行。如果有，就说明它们是死亡了的Binder本地对象。

**4.3、**看看binder\_release，从filp->private\_data中拿到死亡进程的proc信息，接着到binder\_defer\_work，这里使用了一个工作队列binder\_deferred\_workqueue，看看static DECLARE\_WORK(binder\_deferred\_work, binder\_deferred\_func)，进入binder\_deferred\_func

binder\_deferred\_release {

while ((n = rb\_first(&proc->nodes))) {

struct binder\_node \*node;

node = rb\_entry(n, struct binder\_node, rb\_node);

nodes++;

rb\_erase(&node->rb\_node, &proc->nodes);

incoming\_refs = binder\_node\_release(node, incoming\_refs);

}

}

这里查找死亡进程中的所有binder实体，一个进程可能有很多服务，比如mediaserver。对每一个binder node调用binder\_node\_release。

binder\_node\_release {

hlist\_for\_each\_entry(ref, &node->refs, node\_entry) {

refs++;

if (!ref->death)

continue;

death++;

if (list\_empty(&ref->death->work.entry)) {

ref->death->work.type = BINDER\_WORK\_DEAD\_BINDER;

list\_add\_tail(&ref->death->work.entry,

&ref->proc->todo);

wake\_up\_interruptible(&ref->proc->wait);

}

}

}

binder\_node\_release中查找binder实体对应的所有binder引用，因为每一个binder实体对应多个进程中的binder引用。如果引用中注册过death指针，会给引用所在的进程分发一个事务，类型是BINDER\_WORK\_DEAD\_BINDER，添加到它的todo列表中，然后把该进程唤醒。唤醒后binder\_thread\_read进入while，读取todo中待处理事务，进入case BINDER\_WORK\_DEAD\_BINDER，这里会填充数据bwr，并且cmd赋值为BR\_DEAD\_BINDER，数据为death->cookie(之前linkToDeath时传下来的BpBinder指针)。把todo清空后返回到binder\_ioctl\_write\_read，执行copy\_to\_user，把bwr数据拷贝到用户空间。

**4.4、调用返回到IPCThreadState::waitForResponse**

根据mIn中的cmd走到executeCommand {

case BR\_DEAD\_BINDER:

{

BpBinder \*proxy = (BpBinder\*)mIn.readPointer();//读出death->cookie

proxy->sendObituary();//执行sendObituary

mOut.writeInt32(BC\_DEAD\_BINDER\_DONE);//填充mOut，发送给binder驱动，告诉驱动，该进程已收到死亡通知。

mOut.writePointer((uintptr\_t)proxy);//填入BpBinder

} break;

}

**4.5、sendObituary**

调用binder->linkToDeath时binder的sendObituary。那个binder对象的mObituaries保存了DeathRecipient recipient，linkToDeath的参数sDeathNotifier。

self->clearDeathNotification(mHandle, this);self->flushCommands();告诉驱动清理掉注册过的死亡通知。

看看驱动中的处理：cmd是BC\_CLEAR\_DEATH\_NOTIFICATION

1、ref->death = NULL把ref中的death清空

2、death->work.type = BINDER\_WORK\_CLEAR\_DEATH\_NOTIFICATION，在todo中添加一个事务。如果线程中断休眠了则唤醒它，如果正在loop则只是添加这个事务到它的todo列表中。\*consumed = ptr - buffer;更新\*consumed值

3、binder\_thread\_read中读出2中设置的事务，cmd设为BR\_CLEAR\_DEATH\_NOTIFICATION\_DONE，清除该事件后返回。

接着走到：if (obits != NULL) {

const size\_t N = obits->size();

for (size\_t i=0; i<N; i++) {

reportOneDeath(obits->itemAt(i));

}

delete obits;

}

从Obituaries类型容器mObituaries取出所有link过的DeathRecipient对象，并一一调用reportOneDeath{recipient->binderDied(this);}，终于看见binderDied的调用了。DeathNotifier覆盖了binderDied方法。根据前面的介绍，DeathNotifier中binderDied方法会调用外部类的died方法，最后走到MediaPlayer的died函数，MediaPlayer::died{notify(MEDIA\_ERROR, MEDIA\_ERROR\_SERVER\_DIED, 0)}，通知app MediaPlayerService挂掉了。over。

**5、unlinkToDeath**

self->clearDeathNotification(mHandle, this);

self->flushCommands();

cmd为BC\_CLEAR\_DEATH\_NOTIFICATION，数据有Handle和BpBinder指针。类似于linkToDeath，看看binder驱动中的处理，只是cmd不同。

前面linkToDeath时有这一步。

**6、sMediaPlayerService = interface\_cast<IMediaPlayerService>(binder)**

这里很熟悉了，参考《2-Bp-通过ServiceManager分析binder机制-defaultServiceManager》，里面有模板interface\_cast的详细介绍。

他的参数binder，就是上面分析过的，是一个根据MediaPlayerService在该进程中的内核binder引用描述符创建的BpBinder对象，注意：ServiceManager进程中也有一个MediaPlayerService的binder引用，属于ServiceManager进程，和这个不同，虽然他们都指向同一个binder node实体，但是本身属于不同的进程。

sMediaPlayerService = new BpMediaPlayerService(BpBinder(mHandle))。

**7、延伸、**

本章只关注了binder，关于BpMediaPlayerService和BnMediaPlayerService类本身的分析参考《Bn与Bp MediaPlayerService》。