如果已经对binder内核中的实现有了了解，binder的框架层的实现与内核层有着一种对应关系，理解起来并不复杂。之前说过，binder内核要借助于service manager的实现来完成功能。这里就从service manager开始来理解框架代码吧。

frameworks/native/cmds/servicemanager/main.cpp

int main(int argc, char\*\* argv) {

if (argc > 2) {

LOG(FATAL) << "usage: " << argv[0] << " [binder driver]";

}

const char\* driver = argc == 2 ? argv[1] : "/dev/binder";

sp<ProcessState> ps = ProcessState::initWithDriver(driver);

ps->setThreadPoolMaxThreadCount(0);

ps->setCallRestriction(ProcessState::CallRestriction::FATAL\_IF\_NOT\_ONEWAY);

sp<ServiceManager> manager = new ServiceManager(std::make\_unique<Access>());

IPCThreadState::self()->setTheContextObject(manager);

ps->becomeContextManager(nullptr, nullptr);

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

// should not be reached

return EXIT\_FAILURE;

}

以上是service manager的主函数，之前就曾引用过，这里我们从框架的角度来再认识一次。

## ProcessState

之前的描述中，我们已经强调过binder在内核中是借助于驱动程序的框架实现的，而它的表现形式就是一个文件。要使用binder，首先要做的一件事就是打开binder的设备文件—— “/dev/binder”。 在c++框架层，这个工作就是由ProcessState来实现的。

frameworks/native/libs/binder/ProcessState.cpp

static int open\_driver(const char \*driver)

{

int fd = open(driver, O\_RDWR | O\_CLOEXEC);

if (fd >= 0) {

int vers = 0;

status\_t result = ioctl(fd, BINDER\_VERSION, &vers);

if (result == -1) {

ALOGE("Binder ioctl to obtain version failed: %s", strerror(errno));

close(fd);

fd = -1;

}

if (result != 0 || vers != BINDER\_CURRENT\_PROTOCOL\_VERSION) {

ALOGE("Binder driver protocol(%d) does not match user space protocol(%d)! ioctl() return value: %d",

vers, BINDER\_CURRENT\_PROTOCOL\_VERSION, result);

close(fd);

fd = -1;

}

size\_t maxThreads = DEFAULT\_MAX\_BINDER\_THREADS;

result = ioctl(fd, BINDER\_SET\_MAX\_THREADS, &maxThreads);

if (result == -1) {

ALOGE("Binder ioctl to set max threads failed: %s", strerror(errno));

}

} else {

ALOGW("Opening '%s' failed: %s\n", driver, strerror(errno));

}

return fd;

}

以上是从ProcessState中截取出来的代码，其实ProcessState的主要工作都集中在这段函数里了。通过 int fd = open(driver, O\_RDWR | O\_CLOEXEC); 打开文件设备，通过ioctl确认版本和设置最大的线程数。再啰嗦一次，当内核发现接收命令的线程数不够的时候，它会发一个BC\_SPAWN\_LOOPER命令给客户空间，客户空间（其实这里就是ProcessState）就可以自行再启动一个线程去接收命令。这个你可以看看spawnPooledThread的实现。

ProcesState是通过单件的形式提供的，构造和析构函数都是private类型的，只能通过静态函数self或者initWithDriver 来实现对它的访问。保证了每个进程只能有一个ProcessState对象，也就只能调用一次 open 去打开文件。这也就意味着在内核层只会为进程生成一个binder\_proc。

回到service manager的代码，对于ProcessState的最后调用是：

IPCThreadState::self()->setTheContextObject(manager);

这个函数是我们之前说过的，帮助生成一个上下文binder\_node对象，客户端的对应，稍后会专门去讲。

## IPCThreadState

binder设备打开之后，就可以进行进程间的通信了。内核篇已经讲解过这个是靠ioctl发送BINDER\_WRITE\_READ命令来实现的。而我们可以在IPCThreadState.cpp中找到以下的代码：

ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr)

这段代码在IPCThreadState::talkWithDriver中，也只在这个函数里可以找到，也就是说所有通信相关的代码都需要经过talkWithDriver来实现。

talkWithDriver的代码不复杂，除了错误处理和log，就是调用ioctl了，之前已经展开过binder\_write\_read结构，所以这里也没必要再说了。

再次回到service manager的代码：

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

函数最后调用的是以上函数，这里的作用就很明显了，让进程进入循环等待要处理的命令。

void IPCThreadState::joinThreadPool(bool isMain)

{

mOut.writeInt32(isMain ? BC\_ENTER\_LOOPER : BC\_REGISTER\_LOOPER);

status\_t result;

do {

processPendingDerefs();

// now get the next command to be processed, waiting if necessary

result = getAndExecuteCommand();

if (result < NO\_ERROR && result != TIMED\_OUT && result != -ECONNREFUSED && result != -EBADF) {

mProcess->mDriverFD, result);

}

// Let this thread exit the thread pool if it is no longer

// needed and it is not the main process thread.

if(result == TIMED\_OUT && !isMain) {

break;

}

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

(void\*)pthread\_self(), getpid(), result);

mOut.writeInt32(BC\_EXIT\_LOOPER);

talkWithDriver(false);

}

这段函数中循环开始和结束前的 BC\_ENTER\_LOOPER， BC\_EXIT\_LOOPER的设置，是用来告诉内核该线程进入循环处理。processPendingDerefs 用来辅助客户进程空间的清理工作，这里都不再展开。往下看 getAndExecuteCommand 的实现：

status\_t IPCThreadState::getAndExecuteCommand()

{

status\_t result;

int32\_t cmd;

result = talkWithDriver();

if (result >= NO\_ERROR) {

size\_t IN = mIn.dataAvail();

if (IN < sizeof(int32\_t)) return result;

cmd = mIn.readInt32();

result = executeCommand(cmd);

}

return result;

}

我将这段代码中间的错误处理以及记录性能的代码都删了，剩下真正要执行的代码，其实就是talkWithDriver 与 executeCommand两个函数。talkWithDriver 之前说过了，当有消息发到当前线程的时候，ioctl会从内核层返回，而所获得传输数据就在binder\_write\_read 结构中，这里被保存到了mIn中。当得到数据以后，就可以执行executeCommand了 。

status\_t IPCThreadState::executeCommand(int32\_t cmd)

{

BBinder\* obj;

RefBase::weakref\_type\* refs;

status\_t result = NO\_ERROR;

switch ((uint32\_t)cmd) {

case BR\_ERROR:

result = mIn.readInt32();

break;

BINDER\_WRITE\_READ又支持不同的子命令，executeCommand就针对不同的命令就行处理。当然我们最关心的是对于BR\_TRANSACTION 的处理：

if (tr.target.ptr) {

// We only have a weak reference on the target object, so we must first try to

// safely acquire a strong reference before doing anything else with it.

if (reinterpret\_cast<RefBase::weakref\_type\*>(

tr.target.ptr)->attemptIncStrong(this)) {

error = reinterpret\_cast<BBinder\*>(tr.cookie)->transact(tr.code, buffer,

&reply, tr.flags);

reinterpret\_cast<BBinder\*>(tr.cookie)->decStrong(this);

} else {

error = UNKNOWN\_TRANSACTION;

}

} else {

error = the\_context\_object->transact(tr.code, buffer, &reply, tr.flags);

}

一样去除了错误处理，我们看到这里对tr.target.ptr做判断，ptr为0的时候，代表着是全局上下文binder\_node对象，它由service manager拥有，调用一下函数：

error = the\_context\_object->transact(tr.code, buffer, &reply, tr.flags);

而在其它情形下的调用是

error = reinterpret\_cast<BBinder\*>(tr.cookie)->transact(tr.code, buffer,

&reply, tr.flags);

这里看到tr.cookie将被转化为BBinder对象，然后调用transact函数。

tr.cookie的内容其实与tr.target.ptr相同，可以这么调用是因为在发送BR\_TRANSACTION命令前，需要发送BC\_INCREFS命令，而在处理BC\_INCREFS之后，tr.cookie为被设置成tr.target.ptr。（大概是这个意思，我就不去纠结这个问题了。）

这里我们看到了对BBinder对象的调用，而这是下一节相关的内容。

另外提的一点是IPCThreadState貌似是单件模式，与ProcessState类似的调用模式。事实上并不是，这里用到了线程局部存储，每个线程在第一次调用IPCThreadState的时候都会生成自己的IPCThreadState对象。不过在线程层面它又确实是单件，每个线程只会有一个IPCThreadState对象。

## BBinder

一会我们我看到BnInterface，BpInterface等，先简要描述一下。Bn 代表着Binder native，其实就是提供服务的一端，Bp 代表着Binder proxy，也就是请求服务的一方。

直接去看这部分的代码，有点绕，换个思路，如果换成我们自己去实现这部分代码该如何处理。这部分代码的说明将以AudioFlinger的代码来说明，因为Service Manager的代码改成了aidl实现，多少影响了熟悉的过程。另外我自己本身熟悉binder的目的就是为了熟悉AudiioFlinger。

还是从一段旧代码开始：

void AudioFlinger::instantiate() {

defaultServiceManager()->addService(

String16("media.audio\_flinger"), new AudioFlinger());

}

这是Andorid1.0时候的代码了，放到这儿就是为了理解简单，稍后我们给出新代码，不过除了增加了复杂性，也看不出新代码有多少优势。

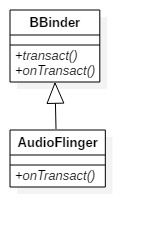
这段代码是AudioFlinger的初始化代码，可以看到这里new出一个AudioFlinger对象后，直接就传到了addService中。调用处甚至都没有保存新生成的对象，任其自身自灭了。（这里AudioFlinger对象只所以没有被销毁，和android的智能指针实现有关。感兴趣的是可以找sp的相关文章看看）我们知道这个对象将被内核保存，然后接收其它线程发送过来的消息。IPCThreadState中，我们介绍过，executeCommand中需要一个transact函数：

error = reinterpret\_cast<BBinder\*>(tr.cookie)->transact(tr.code, buffer,

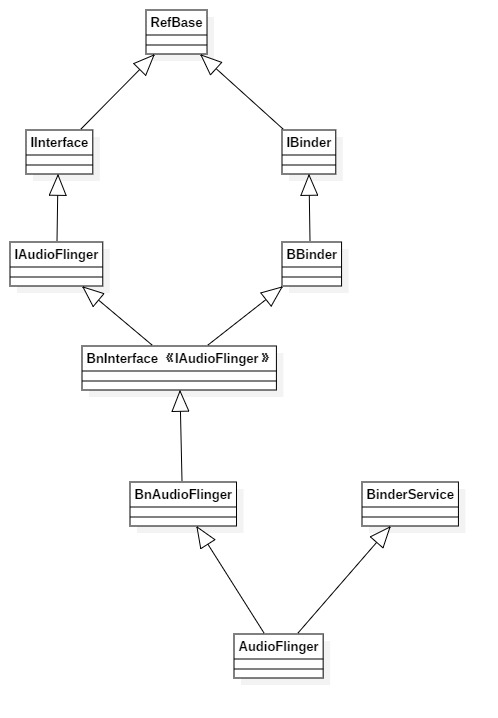
&reply, tr.flags);

一个进程中可以有多个Binder对象，比如mediaservice中有AudioFlinger对象，还有AudioPolicyService等对象。所以transact需要是个虚函数，每个binder对象都需要有自己的实现。

总结一下：所有的binder对象，其实只需要有一个transact虚函数，理论上就可以运行起来。所以最简洁的结构，如下图，Binder 就可以运行起来。



Binder框架的实现者可能认为这个太过简单了，所以往复杂里写了写代码，不过不要紧，复杂也没多少代码，一步一步来看。



依照上图，我们来看看，这些类都提供了什么。

### 3.1 IInterface

frameworks/native/libs/binder/include/binder/IInterface.h

class IInterface : public virtual RefBase

{

public:

IInterface();

static sp<IBinder> asBinder(const IInterface\*);

static sp<IBinder> asBinder(const sp<IInterface>&);

protected:

virtual ~IInterface();

virtual IBinder\* onAsBinder() = 0;

};

Binder的调用者和实现者其实使用的是同一个接口，框架的设计者认为它们需要从同一个接口继承，也就是这里的IIterface。这里实现了两个静态函数asBinder，实现者也就是Bn开始的类其实也用不上该函数。不过还是提供了实现，返回的其实就是this。

简单看一下：

sp<IBinder> IInterface::asBinder(const IInterface\* iface)

{

if (iface == nullptr) return nullptr;

return const\_cast<IInterface\*>(iface)->onAsBinder();

}

template<typename INTERFACE>

IBinder\* BnInterface<INTERFACE>::onAsBinder()

{

return this;

}

而对于调用测，也就是Bp开始的类，返回的是remote,这个我们后面会碰到：

template<typename INTERFACE>

inline IBinder\* BpInterface<INTERFACE>::onAsBinder()

{

return remote();

}

### 3.2 IAudioFlinger

IAudioFlinger直接继承于IIterface。

Frameworks/av/include/media/IAudioFlinger.h

class IAudioFlinger : public IInterface

{

public:

DECLARE\_META\_INTERFACE(AudioFlinger);

.......

};

IAudioFlinger 也是个接口类，不会真正实现，它为Bn和Bp提供了共同的通信接口，这个是我们后一部分的主要内容。

这里简单提一下DECLARE\_META\_INTERFACE，这是一个宏定义，在IInterface.h中。这里简单展开一下：

public:

static const ::android::String16 descriptor;

static ::android::sp<IAudioFlinger> asInterface(

const ::android::sp<::android::IBinder>& obj);

virtual const ::android::String16& getInterfaceDescriptor() const;

IAudioFlinger();

virtual ~IAudioFlinger();

static bool setDefaultImpl(std::unique\_ptr<IAudioFlinger> impl);

static const std::unique\_ptr<IAudioFlinger>& getDefaultImpl();

private:

static std::unique\_ptr<IAudioFlinger> default\_impl;

public:

可以看到就是用来定义变量和函数，后续我们会提一下asInterface。

而IMPLEMENT\_META\_INTERFACE就是用来展开这些函数的实现，可以自己替换看看。

### 3.3 IBinder

这里提供了一些binder可能用到的公共函数接口，比如pingBinder, dump等。最重要的就是：

virtual status\_t transact( uint32\_t code,

const Parcel& data,

Parcel\* reply,

uint32\_t flags = 0) = 0;

Bp 端用来传输数据，Bn端用来接收数据。

剩下的一些接口吧，很多对于Bn端来说并没有意义，比如：

// NOLINTNEXTLINE(google-default-arguments)

status\_t BBinder::linkToDeath(

const sp<DeathRecipient>& /\*recipient\*/, void\* /\*cookie\*/,

uint32\_t /\*flags\*/)

{

return INVALID\_OPERATION;

}

// NOLINTNEXTLINE(google-default-arguments)

status\_t BBinder::unlinkToDeath(

const wp<DeathRecipient>& /\*recipient\*/, void\* /\*cookie\*/,

uint32\_t /\*flags\*/, wp<DeathRecipient>\* /\*outRecipient\*/)

{

return INVALID\_OPERATION;

}

### 3.4 BBinder

BBinder呢，和IBinder看起来没多少区别，只是这一层不再是接口，而是将一些函数做了实现，比如上面抄到的linkToDeatch。另外就是新增了，

virtual status\_t onTransact( uint32\_t code,

const Parcel& data,

Parcel\* reply,

uint32\_t flags = 0);

这个，需要我们看看transact的实现：

// NOLINTNEXTLINE(google-default-arguments)

status\_t BBinder::transact(

uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags)

{

data.setDataPosition(0);

status\_t err = NO\_ERROR;

switch (code) {

case PING\_TRANSACTION:

err = pingBinder();

break;

case EXTENSION\_TRANSACTION:

err = reply->writeStrongBinder(getExtension());

break;

case DEBUG\_PID\_TRANSACTION:

err = reply->writeInt32(getDebugPid());

break;

default:

err = onTransact(code, data, reply, flags);

break;

}

// In case this is being transacted on in the same process.

if (reply != nullptr) {

reply->setDataPosition(0);

}

return err;

}

transact实现了一些基本的命令，而剩下的任务就由onTransact来接着去实现了，继承类只要实现onTransact就好了。

不过呢BBinder在onTransact中又加了几个命名，这个我也搞不清楚它为啥不放在transact中呢，所以继承类的中的onTransact中最后的代码都应该是：

default:

return BBinder::onTransact(code, data, reply, flags);

### 3.4 BnAudioFlinger

class BnAudioFlinger : public BnInterface<IAudioFlinger>

{

public:

virtual status\_t onTransact( uint32\_t code,

const Parcel& data,

Parcel\* reply,

uint32\_t flags = 0);

// Requests media.log to start merging log buffers

virtual void requestLogMerge() = 0;

};

这里可以看到,BnAudioFlinger继承自BnInterface<IAudioFlinger>, 它最重要的作用是是实现了onTransact函数。

不过在看onTransact之前，我们先看看BnInterface。

template<typename INTERFACE>

class BnInterface : public INTERFACE, public BBinder

{

public:

virtual sp<IInterface> queryLocalInterface(const String16& \_descriptor);

virtual const String16& getInterfaceDescriptor() const;

protected:

typedef INTERFACE BaseInterface;

virtual IBinder\* onAsBinder();

};

简化一下BnInterface，可以是：

class BnInterface : public IAudioFlinger, public BBinder

这样BnInterface就是继承了IAudioFlinger和BBinder，然后它最重要的任务就是重新实现了queryLocalInterface 和 getInterfaceDescriptor，以及onAsBinder。

这里之所以应用模板是因为 BnInterface<IAudioFlinger> 相当于重新定义了一个类，不用本模块的话，就需要定义很多新的类，比如BnInterfaceAudioFlinger。

相对于BnInterface的定义，Bp端的定义是BpInterface，这连个模板定义的目的是区别开IAudioFlinger ,毕竟客户端与服务端的实现还是有差别的。

### 3.5 BinderService

template<typename SERVICE>

class BinderService

{

public:

static status\_t publish(bool allowIsolated = false,

int dumpFlags = IServiceManager::DUMP\_FLAG\_PRIORITY\_DEFAULT) {

sp<IServiceManager> sm(defaultServiceManager());

return sm->addService(String16(SERVICE::getServiceName()), new SERVICE(),

allowIsolated,dumpFlags);

}

static void publishAndJoinThreadPool(

bool allowIsolated = false,

int dumpFlags = IServiceManager::DUMP\_FLAG\_PRIORITY\_DEFAULT) {

publish(allowIsolated, dumpFlags);

joinThreadPool();

}

static void instantiate() { publish(); }

static status\_t shutdown() { return NO\_ERROR; }

private:

static void joinThreadPool() {

sp<ProcessState> ps(ProcessState::self());

ps->startThreadPool();

ps->giveThreadPoolName();

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

}

};

BinderService的存在的最主要目的是提供一个静态instantiate 函数。然后就不需要自己再去写以下的代码：

void AudioFlinger::instantiate() {

defaultServiceManager()->addService(

String16("media.audio\_flinger"), new AudioFlinger());

}

### 3.6 BnAudioFlinger::onTransact

其实开始的时候就说过了，对于binder来说，最重要就是transact函数了。由于之前说过的修改，这里就成了onTransact函数。来看看这部分代码的实现。

status\_t BnAudioFlinger::onTransact(

uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags)

{

switch (code) {

case SET\_STREAM\_VOLUME:

onTransact最重要的工作还是借助于swtich选择不同的code来处理。看一个具体的实现：

case SET\_MODE: {

CHECK\_INTERFACE(IAudioFlinger, data, reply);

audio\_mode\_t mode = (audio\_mode\_t) data.readInt32();

reply->writeInt32( setMode(mode) );

return NO\_ERROR;

} break;

这里是客户端发送一个SET\_MODE命令过来，onTransact的实现其实也简单，读取客户端传来的值，调用虚函数setMode，然后返回。

我们顺便再看看客户端发送消息的函数：

virtual status\_t setMode(audio\_mode\_t mode)

{

Parcel data, reply;

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

data.writeInt32(mode);

remote()->transact(SET\_MODE, data, &reply);

return reply.readInt32();

}

以上是BpAudioFinger中的函数，发送请求其实就是将SET\_MODE作为code发送给服务端。