# 理解 binder--框架层

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。

## 一， 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。

# 二， 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。

# 三， 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());
```

}

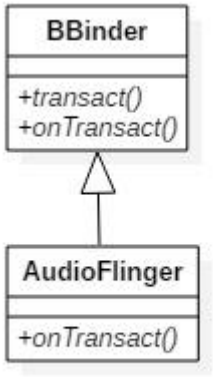
这是 Andorid1.0 时候的代码了，放到这儿就是为了理解简单，稍后我们给出新代码，不过除了增加了复杂性，也看不出新代码有多少优势。

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：

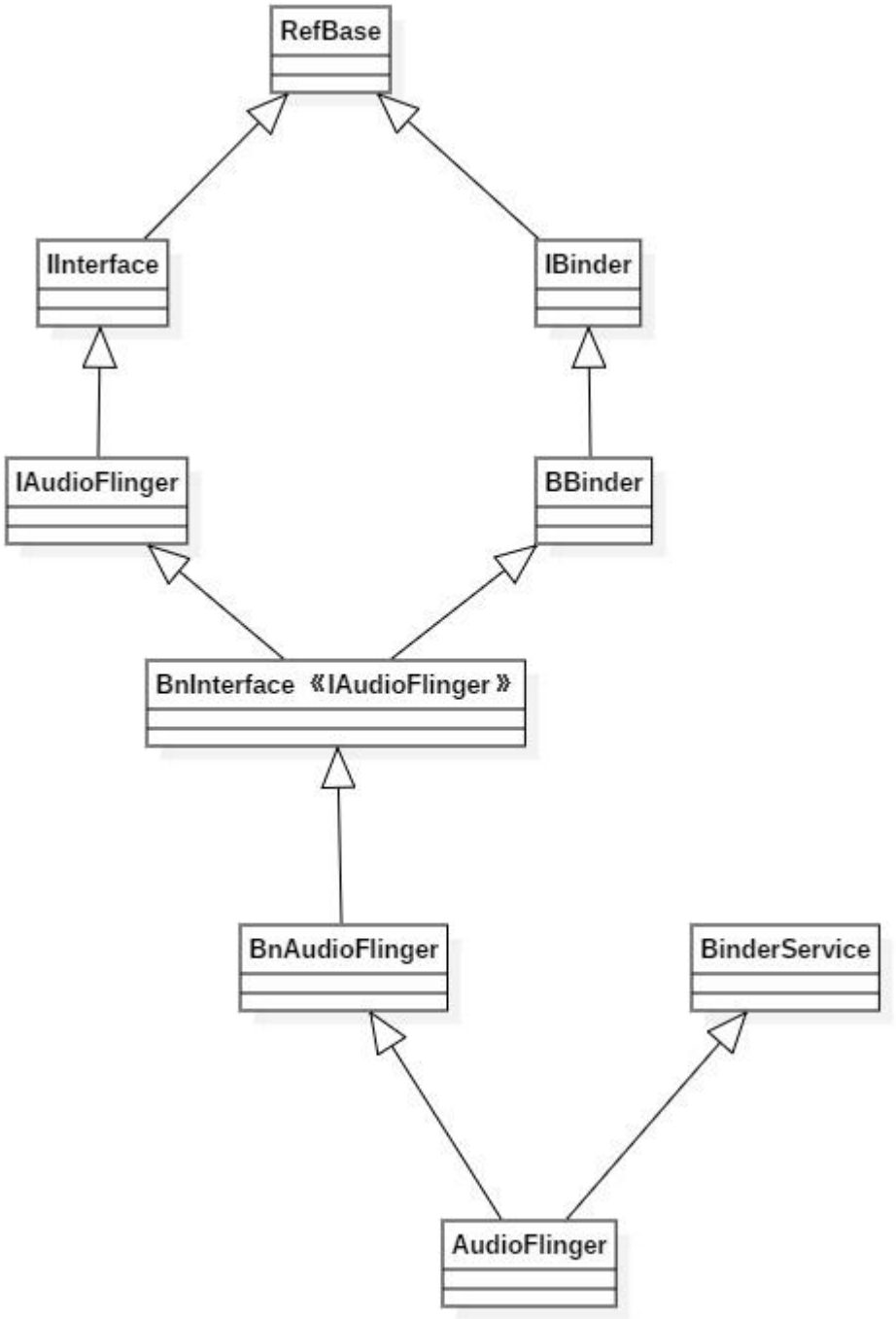
```
error = reinterpret_cast<BBinder*>(tr.cookie)->transact(tr.code, buffer,
                                                        &reply, tr.flags);
```

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。

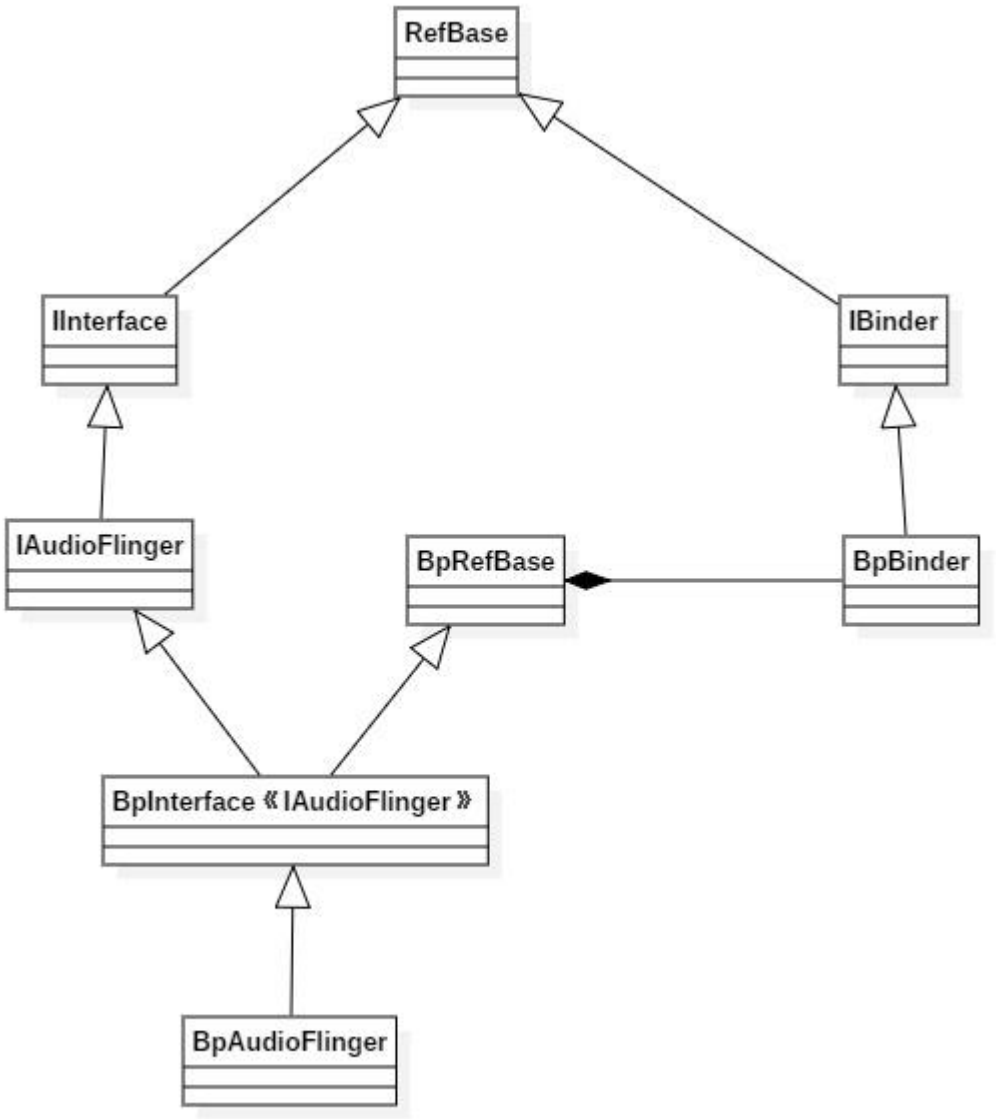
我们顺便再看看客户端发送消息的函数：

```
    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。

# 四， BpAudioFlinger




BpAudioFlinger 的理解，先从类继承关系开始吧。可以看到 BpAudioFlinger 没有从 IBinder 直接继承，而是通过 BpRefBase 来指向一个 IBinder 对象。这就是面向对象里所讨论的“is-a”和“has-a”的问题了。对于实现端来说，BnAudioFlinger 的指针对象被直接传给了内核，那

么它应该是一个 binder 了。但对于应用侧来说，它会从内核得到一个 handle，其实是服务端 binder 对象的一个编号数字，算不上是 binder 对象，所以这里是“has-a”的关系。这里只是顺便提提，面向对象的内容，自行理解吧。

其实对于应用侧，我们可以只关注于两件事，一是如何生成一个 Bp 对象，另外一个就是如何发送消息。

先来看第一个问题，先看看 BpAudioFlinger 如何来的，AudioSyste.cpp 中有下函数，用来得到 BpAudioFlinger:

```
const sp<IAudioFlinger> AudioSystem::get_audio_flinger()
{
    sp<IAudioFlinger> af;
    {
        Mutex::Autolock _l(gLock);
        if (gAudioFlinger == 0) {
            sp<IServiceManager> sm = defaultServiceManager();
            sp<IBinder> binder;

            binder = sm->getService(String16("media.audio_flinger"));
            binder->linkToDeath(gAudioFlingerClient);

            gAudioFlinger = interface_cast<IAudioFlinger>(binder);
        }
        af = gAudioFlinger;
    }
    return af;
}
```

这里看到，需要调用 ServiceManager 提供的 getService 函数，以及 IIterface.h 中定义的 interface_cast。

getService 是用来生成一个 BpBinder 对象，来看看 IServiceManager.cpp 中的代码（这里我截取了早一点版本上的代码，最新代码改成了 aidl 实现，我没有最新版本的编译环境，没有挨 aidl 生成的代码）

```
virtual sp<IBinder> getService(const String16& name) const
{
    unsigned n;
    for (n = 0; n < 5; n++){
        sp<IBinder> svc = checkService(name);
        if (svc != NULL) return svc;
        ALOGI("Waiting for service %s...\n", String8(name).string());
        sleep(1);
    }
    return NULL;
```

```
}

virtual sp<IBinder> checkService( const String16& name) const
{
    Parcel data, reply;
    data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
    data.writeString16(name);
    remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);
    return reply.readStrongBinder();
}
```

这里可以看到最后的 reply.readStrongBinder()返回了一个 Binder 对象，然后经过层层调用，最后调用到的函数是

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
```

感兴趣的可以自己追踪一下。这里最终传过来的是一个 handle，说过好多次了其实就是一个内核提供的一个索引值，然后用它生成一个 BpBinder 对象：

```
b = BpBinder::create(handle);
```

当然这里的叙述忽略了一些实现细节，比如在 ProcessState 中将保证每个 handle 只生成一个 BpBinder 对象。

BpBinder 对象有了，接下来应该看：

```
gAudioFlinger = interface_cast<IAudioFlinger>(binder);
```

interface_cast 代码：

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

最后调用到的其实是 IAudioFlinger 中的 asInterface，宏定义的函数：

```
::android::sp<I##INTERFACE> I##INTERFACE::asInterface(                  \
        const ::android::sp<::android::IBinder>& obj)                   \
{                                                                       \
    ::android::sp<I##INTERFACE> intr;                                   \
    if (obj != nullptr) {                                               \
        intr = static_cast<I##INTERFACE*>(                              \
            obj->queryLocalInterface(                                   \
                    I##INTERFACE::descriptor).get());                   \
        if (intr == nullptr) {                                          \
            intr = new Bp##INTERFACE(obj);                              \
        }                                                               \
    }                                                                   \
```

```
    return intr;                                                    \
}
```

BbBinder 没有实现 queryLocalInterface，所以调用的是默认实现，永远返回 nullptr。

```
sp<IInterface>  IBinder::queryLocalInterface(const String16& /*descriptor*/)
{
    return nullptr;
}
```

最终调用的函数是：

intr = new Bp##INTERFACE(obj);

其实就是 new BpAudioFlinger(obj)，用 BpBinder 对象构建 BpAudioFlinger。BpRefBase 中定义了 remote 函数，其实就是用它来得到 BpBinder 对象，然后 BpAudioFlinger 就可以通过 remote 来反问 BpBinder 对象了。

传送数据是通过 BpBinder 调用 IPCThreadState::transact 来实现。

到此框架层的主干代码基本介绍过了，还有就是 parcel.cpp 了。它是用来帮忙生成和读取传送的数据的。代码很长但不难，就不再这儿展开了。