
Android 深入浅出之 Binder 机制

Android 深入浅出之 Binder 机制

一 说明

Android 系统最常见也是初学者最难搞明白的就是 Binder 了，很多很多的 Service 就是通过 Binder 机制来和客户端通讯交互的。所以搞明白 Binder 的话，在很大程度上就能理解程序运行的流程。

我们这里将以 MediaService 的例子来分析 Binder 的使用：

- ServiceManager，这是 Android OS 的整个服务的管理程序
- MediaService，这个程序里边注册了提供媒体播放的服务程序 MediaPlayerService，我们最后只分析这个
- MediaPlayerClient，这个是与 MediaPlayerService 交互的客户端程序

下面先讲讲 MediaService 应用程序。

二 MediaService 的诞生

MediaService 是一个应用程序，虽然 Android 搞了七七八八的 JAVA 之类的东西，但是在本质上，它还是一个完整的 Linux 操作系统，也还没有牛到什么应用程序都是 JAVA 写。所以，MS(MediaService)就是一个和普通的 C++应用程序一样的东西。MediaService 的源码文件在：framework\base\Media\MediaServer\Main_mediaserver.cpp 中。让我们看看到底是个什么玩意儿！

```
int main(int argc, char** argv)
{
//FT，就这么简单？？
//获得一个 ProcessState 实例
    sp<ProcessState> proc(ProcessState::self());
    //得到一个 ServiceManager 对象
    sp<IServiceManager> sm = defaultServiceManager();
    MediaPlayerService::instantiate();//初始化 MediaPlayerService 服务
    ProcessState::self()->startThreadPool();//看名字，启动 Process 的线程池？
    IPCThreadState::self()->joinThreadPool();//将自己加入到刚才的线程池？
}
```

其中，我们只分析 MediaPlayerService。

这么多疑问，看来我们只有一个个函数深入分析了。不过，这里先简单介绍下 sp 这个东西。

sp，究竟是 smart pointer 还是 strong pointer 呢？其实我后来发现不用太关注这个，就把它当做一个普通的指针看待，即 sp<IServiceManager>======》IServiceManager*吧。sp 是 google 搞出来的为了方便 C/C++程序员管理指针的分配和释放的一套方法，类似 JAVA 的什么 WeakReference 之类的。我个人觉得，要是自己写程序的话，不用这个东西也成。

好了，以后的分析中，sp<XXX>就看成是 XXX*就可以了。

2.1 ProcessState

第一个调用的函数是 ProcessState::self()，然后赋值给了 proc 变量，程序运行完，proc 会自动 delete 内部的内容，所以就自动释放了先前分配的资源。

ProcessState 位置在 framework\base\libs\binder\ProcessState.cpp

```
sp<ProcessState> ProcessState::self()
{
    if (gProcess != NULL) return gProcess;---->第一次进来肯定不走这儿
    AutoMutex _l(gProcessMutex);--->锁保护
    if (gProcess == NULL) gProcess = new ProcessState;--->创建一个 ProcessState 对象
     return gProcess;--->看见没，这里返回的是指针，但是函数返回的是 sp<xxx>，所以
     //把 sp<xxx>看成是 XXX*是可以的
}
再来看看 ProcessState 构造函数
//这个构造函数看来很重要
ProcessState::ProcessState()
    : mDriverFD(open_driver())----->Android 很多代码都是这么写的,稍不留神就没看见这里调用了一个很重要的函数
    , mVMStart(MAP_FAILED)//映射内存的起始地址
    , mManagesContexts(false)
    , mBinderContextCheckFunc(NULL)
    , mBinderContextUserData(NULL)
    , mThreadPoolStarted(false)
    , mThreadPoolSeq(1)
{
```

```
    if (mDriverFD >= 0) {
    //BIDNER_VM_SIZE 定义为(1*1024*1024) - (4096 *2) 1M-8K
        mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE |
MAP_NORESERVE,
                mDriverFD, 0);//这个需要你自己去 man mmap 的用法了，不过大概意思就是
               //将 fd 映射为内存，这样内存的 memcpy 等操作就相当于 write/read(fd)了
    }
    ...
}
```

最讨厌这种在构造 list 中添加函数的写法了，常常疏忽某个变量的初始化是一个函数调用的结果。

open_driver，就是打开/dev/binder 这个设备，这个是 android 在内核中搞的一个专门用于完成

进程间通讯而设置的一个虚拟的设备。BTW，说白了就是内核的提供的一个机制，这个和我们用 socket 加 NET_LINK 方式和内核通讯是一个道理。

```
static int open_driver()
{
    int fd = open("/dev/binder", O_RDWR);//打开/dev/binder
    if (fd >= 0) {
     ....
        size_t maxThreads = 15;
       //通过 ioctl 方式告诉内核，这个 fd 支持最大线程数是15个。
        result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);     }
    return fd;
```

好了，到这里 Process::self 就分析完了，到底干什么了呢？

- 打开/dev/binder 设备，这样的话就相当于和内核 binder 机制有了交互的通道
- 映射 fd 到内存，设备的 fd 传进去后，估计这块内存是和 binder 设备共享的

接下来，就到调用 defaultServiceManager()地方了。

2.2 defaultServiceManager

defaultServiceManager 位置在 framework\base\libs\binder\IServiceManager.cpp 中

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
    //又是一个单例，设计模式中叫 singleton。
    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
//真正的 gDefaultServiceManager 是在这里创建的喔
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }
  return gDefaultServiceManager;
}
-----》
gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
ProcessState::self，肯定返回的是刚才创建的 gProcess，然后调用它的 getContextObject，注意，传进去的是 NULL，即0
//回到 ProcessState 类，
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
{
    if (supportsProcesses()) {//该函数根据打开设备是否成功来判断是否支持 process，
    //在真机上肯定走这个
        return getStrongProxyForHandle(0);//注意，这里传入0
    }
}
----》进入到 getStrongProxyForHandle，函数名字怪怪的，经常严重阻碍大脑运转
//注意这个参数的命名，handle。搞过 windows 的应该比较熟悉这个名字，这是对
//资源的一种标示，其实说白了就是某个数据结构，保存在数组中，然后 handle 是它在这个
```

```
数组中的索引。--->就是这么一个玩意儿
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;
    AutoMutex _l(mLock);
     handle_entry* e = lookupHandleLocked(handle);--》哈哈，果然，从数组中查找对应
     索引的资源，lookupHandleLocked 这个就不说了，内部会返回一个 handle_entry
 下面是 handle_entry 的结构
/*
struct handle_entry {
                    IBinder* binder;--->Binder
                    RefBase::weakref_type* refs;-->不知道是什么，不影响.
                };
*/
    if (e != NULL) {
        IBinder* b = e->binder; -->第一次进来，肯定为空
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
            b = new BpBinder(handle); --->看见了吧，创建了一个新的 BpBinder
            e->binder = b;
            result = b;
        }....
    }
    return result;返回刚才创建的 BpBinder。
}
//到这里，是不是有点乱了？对，当人脑分析的函数调用太深的时候，就容易忘记。
我们是从 gDefaultServiceManager = interface_cast<IServiceManager>(
                    ProcessState::self()->getContextObject(NULL));
开始搞的，现在，这个函数调用将变成
gDefaultServiceManager = interface_cast<IServiceManager>(new BpBinder(0));
```

BpBinder 又是个什么玩意儿？Android 名字起得太眼花缭乱了。

因为还没介绍 Binder 机制的大架构，所以这里介绍 BpBinder 不合适，但是又讲到 BpBinder 了，不介绍 Binder 架构似乎又说不清楚....，sigh！

恩，还是继续把层层深入的函数调用栈化繁为简吧，至少大脑还可以工作。先看看 BpBinder 的构造函数把。

2.3 BpBinder

BpBinder 位置在 framework\base\libs\binder\BpBinder.cpp 中。

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle) //注意，接上述内容，这里调用的时候传入的是0
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
  IPCThreadState::self()->incWeakHandle(handle);//FT，竟然到 IPCThreadState::self()
}
这里一块说说吧，IPCThreadState::self 估计怎么着又是一个 singleton 吧？
//该文件位置在 framework\base\libs\binder\IPCThreadState.cpp
IPCThreadState* IPCThreadState::self()
{
    if (gHaveTLS) {//第一次进来为 false
restart:
            const pthread_key_t k = gTLS;
//TLS 是 Thread Local Storage 的意思，不懂得自己去 google 下它的作用吧。这里只需要
//知道这种空间每个线程有一个，而且线程间不共享这些空间，好处是？我就不用去搞什么
//同步了。在这个线程，我就用这个线程的东西，反正别的线程获取不到其他线程 TLS 中的
数据。===》这句话有漏洞，钻牛角尖的明白大概意思就可以了。
//从线程本地存储空间中获得保存在其中的 IPCThreadState 对象
//这段代码写法很晦涩，看见没，只有 pthread_getspecific,那么肯定有地方调用
// pthread_setspecific。
            IPCThreadState* st = (IPCThreadState*)pthread_getspecific(k);
            if (st) return st;
            return new IPCThreadState;//new 一个对象，
    }

    if (gShutdown) return NULL;
```

```
    pthread_mutex_lock(&gTLSMutex);
    if (!gHaveTLS) {
        if (pthread_key_create(&gTLS, threadDestructor) != 0) {
            pthread_mutex_unlock(&gTLSMutex);
            return NULL;
        }
        gHaveTLS = true;
    }
    pthread_mutex_unlock(&gTLSMutex);
    goto restart;//我 FT，其实 goto 没有我们说得那样卑鄙，汇编代码很多跳转语句的。
    //关键是要用好。
}
//这里是构造函数，在构造函数里边 pthread_setspecific
IPCThreadState::IPCThreadState()
    : mProcess(ProcessState::self()), mMyThreadId(androidGetTid())
{
    pthread_setspecific(gTLS, this);
    clearCaller();
    mIn.setDataCapacity(256);
    //mIn,mOut 是两个 Parcel，干嘛用的啊？把它看成是命令的 buffer 吧。再深入解释，又会大脑停摆的。
    mOut.setDataCapacity(256);
}
```

出来了，终于出来了....，恩，回到 BpBinder 那。

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle) //注意，接上述内容，这里调用的时候传入的是0
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
.....
```

```
    IPCThreadState::self()->incWeakHandle(handle);
    什么 incWeakHandle，不讲了..
}
```

喔，new BpBinder 就算完了。到这里，我们创建了些什么呢？

- ProcessState 有了。
- IPCThreadState 有了，而且是主线程的。
- BpBinder 有了，内部 handle 值为0

gDefaultServiceManager = interface_cast<IServiceManager>(new BpBinder(0));

终于回到原点了，大家是不是快疯掉了？

interface_cast，我第一次接触的时候，把它看做类似的 static_cast 一样的东西，然后死活也搞不明白 BpBinder*指针怎么能强转为 IServiceManager*，花了 n 多时间查看 BpBinder 是否和 IServiceManager 继承还是咋的....。

终于，我用 ctrl+鼠标(source insight)跟踪进入了 interface_cast

IInterface.h 位于 framework/base/include/binder/IInterface.h

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
所以，上面等价于：
inline sp<IServiceManager> interface_cast(const sp<IBinder>& obj)
{
    return IServiceManager::asInterface(obj);
}
看来，只能跟到 IServiceManager 了。
IServiceManager.h---》framework/base/include/binder/IServiceManager.h
看看它是如何定义的:
```

2.4 IServiceManager

```
class IServiceManager : public IInterface
{
//ServiceManager,字面上理解就是 Service 管理类，管理什么？增加服务，查询服务等
//这里仅列出增加服务 addService 函数
public:
    DECLARE_META_INTERFACE(ServiceManager);
     virtual status_t    addService( const String16& name,
                                                    const sp<IBinder>& service) = 0;
};
DECLARE_META_INTERFACE(ServiceManager)？？
怎么和MFC这么类似？微软的影响很大啊!知道MFC的,有DELCARE肯定有IMPLEMENT
果然，这两个宏 DECLARE_META_INTERFACE 和 IMPLEMENT_META_INTERFACE(INTERFACE, NAME)都在
刚才的 IInterface.h 中定义。我们先看看 DECLARE_META_INTERFACE 这个宏往 IServiceManager 加了什么？
下面是 DECLARE 宏
#define DECLARE_META_INTERFACE(INTERFACE)                                    \
    static const android::String16 descriptor;                                \
    static android::sp<I##INTERFACE> asInterface(                              \
                const android::sp<android::IBinder>& obj);                     \
    virtual const android::String16& getInterfaceDescriptor() const;     \
    I##INTERFACE();                                                                \
    virtual ~I##INTERFACE();
我们把它兑现到 IServiceManager 就是：
static const android::String16 descriptor; -->喔，增加一个描述字符串
static android::sp< IServiceManager > asInterface(const android::sp<android::IBinder>&
obj) ---》增加一个 asInterface 函数
virtual const android::String16& getInterfaceDescriptor() const; ---》增加一个 get 函数
估计其返回值就是 descriptor 这个字符串
IServiceManager ();                                                            \
virtual ~IServiceManager();增加构造和虚析购函数...
```

那 IMPLEMENT 宏在哪定义的呢？

见 IServiceManager.cpp。位于 framework/base/libs/binder/IServiceManager.cpp

```
IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager");
```

下面是这个宏的定义

```
#define IMPLEMENT_META_INTERFACE(INTERFACE, NAME)                                \
    const android::String16 I##INTERFACE::descriptor(NAME);                      \
    const android::String16&                                                     \
            I##INTERFACE::getInterfaceDescriptor() const {                       \
        return I##INTERFACE::descriptor;                                         \
    }                                                                            \
    android::sp<I##INTERFACE> I##INTERFACE::asInterface(                         \
            const android::sp<android::IBinder>& obj)                            \
    {                                                                            \
        android::sp<I##INTERFACE> intr;                                          \
        if (obj != NULL) {                                                       \
            intr = static_cast<I##INTERFACE*>(                                   \
                obj->queryLocalInterface(                                        \
                        I##INTERFACE::descriptor).get());                        \
            if (intr == NULL) {                                                  \
                intr = new Bp##INTERFACE(obj);                                   \
            }                                                                    \
        }                                                                        \
        return intr;                                                             \
    }                                                                            \
    I##INTERFACE::I##INTERFACE() { }                                             \
    I##INTERFACE::~I##INTERFACE() { }                                            \
```

很麻烦吧？尤其是宏看着头疼。赶紧兑现下吧。

```
const
android::String16 IServiceManager::descriptor("android.os.IServiceManager");
const android::String16& IServiceManager::getInterfaceDescriptor() const
  { return IServiceManager::descriptor;//返回上面那个 android.os.IServiceManager
```

```
    }
android::sp<IServiceManager> IServiceManager::asInterface(
                const android::sp<android::IBinder>& obj)
    {
        android::sp<IServiceManager> intr;
        if (obj != NULL) {
            intr = static_cast<IServiceManager *>(
                obj->queryLocalInterface(IServiceManager::descriptor).get());
            if (intr == NULL) {
                intr = new BpServiceManager(obj);
            }
        }
        return intr;
    }
    IServiceManager::IServiceManager () { }
    IServiceManager::~ IServiceManager() { }
```

哇塞，asInterface 是这么搞的啊，赶紧分析下吧，还是不知道 interface_cast 怎么把 BpBinder* 转成了 IServiceManager

我们刚才解析过的 interface_cast<IServiceManager>(new BpBinder(0)),

原来就是调用 asInterface(new BpBinder(0))

```
android::sp<IServiceManager> IServiceManager::asInterface(
                const android::sp<android::IBinder>& obj)
    {
        android::sp<IServiceManager> intr;
        if (obj != NULL) {
            ....
                intr = new BpServiceManager(obj);
//神呐，终于看到和 IServiceManager 相关的东西了，看来
//实际返回的是 BpServiceManager(new BpBinder(0));
            }
        }
        return intr;
}
```

BpServiceManager 是个什么玩意儿？p 是什么个意思？

2.5 BpServiceManager

终于可以讲解点架构上的东西了。p 是 proxy 即代理的意思，Bp 就是 BinderProxy，BpServiceManager，就是 SM 的 Binder 代理。既然是代理，那肯定希望对用户是透明的，那就是说头文件里边不会有这个 Bp 的定义。是吗？

果然，BpServiceManager 就在刚才的 IServiceManager.cpp 中定义。

```
class BpServiceManager : public BpInterface<IServiceManager>
//这种继承方式，表示同时继承 BpInterface 和 IServiceManager，这样 IServiceManger 的
addService 必然在这个类中实现
{
public:
//注意构造函数参数的命名 impl，难道这里使用了 Bridge 模式？真正完成操作的是 impl 对
象？
//这里传入的 impl 就是 new BpBinder(0)
    BpServiceManager(const sp<IBinder>& impl)
        : BpInterface<IServiceManager>(impl)
    {
    }
     virtual status_t addService(const String16& name, const sp<IBinder>& service)
    {
        待会再说..
    }
    基类 BpInterface 的构造函数（经过兑现后）
    //这里的参数又叫 remote，唉，真是害人不浅啊。
    inline BpInterface< IServiceManager >::BpInterface(const sp<IBinder>& remote)
        : BpRefBase(remote)
    {
    }
    BpRefBase::BpRefBase(const sp<IBinder>& o)
        : mRemote(o.get()), mRefs(NULL), mState(0)
    //o.get()，这个是 sp 类的获取实际数据指针的一个方法，你只要知道
```

```
    //它返回的是 sp<xxxx>中 xxx*指针就行
    {
    //mRemote 就是刚才的 BpBinder(0)
       ...
    }
```

好了，到这里，我们知道了：

sp<IServiceManager> sm = defaultServiceManager();返回的实际是 BpServiceManager，它的 remote 对象是 BpBinder，传入的那个 handle 参数是0。

现在重新回到 MediaService。

```
int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());
     sp<IServiceManager> sm = defaultServiceManager();
     //上面的讲解已经完了
     MediaPlayerService::instantiate();//实例化 MediaPlayerservice
     //看来这里有名堂！

     ProcessState::self()->startThreadPool();
     IPCThreadState::self()->joinThreadPool();
}
```

到这里，我们把 binder 设备打开了，得到一个 BpServiceManager 对象，这表明我们可以和 SM 打交道了，但是好像没干什么有意义的事情吧？

2.6 MediaPlayerService

那下面我们看看后续又干了什么？以 MediaPlayerService 为例。

它位于 framework\base\media\libmediaplayerservice\libMediaPlayerService.cpp

```
void MediaPlayerService::instantiate() {
    defaultServiceManager()->addService(
    //传进去服务的名字，传进去 new 出来的对象
                String16("media.player"), new MediaPlayerService());
```

```
}
MediaPlayerService::MediaPlayerService()
{
    LOGV("MediaPlayerService created");//太简单了
    mNextConnId = 1;
}
```

defaultServiceManager 返回的是刚才创建的 BpServiceManager

调用它的 addService 函数。

MediaPlayerService 从 BnMediaPlayerService 派生

```
class MediaPlayerService : public BnMediaPlayerService
```

FT，MediaPlayerService 从 BnMediaPlayerService 派生，BnXXX,BpXXX，快晕了。

Bn 是 Binder Native 的含义，是和 Bp 相对的，Bp 的 p 是 proxy 代理的意思，那么另一端一定有一个和代理打交道的东西，这个就是 Bn。

讲到这里会有点乱喔。先分析下，到目前为止都构造出来了什么。

- BpServiceManager
- BnMediaPlayerService

这两个东西不是相对的两端，从 BnXXX 就可以判断，BpServiceManager 对应的应该是 BnServiceManager，BnMediaPlayerService 对应的应该是 BpMediaPlayerService。

我们现在在哪里?对了，我们现在是创建了 BnMediaPlayerService，想把它加入到系统的中去。

喔，明白了。我创建一个新的 Service—BnMediaPlayerService，想把它告诉 ServiceManager。

那我怎么和 ServiceManager 通讯呢？恩，利用 BpServiceManager。所以嘛，我调用了 BpServiceManager 的 addService 函数！

为什么要搞个 ServiceManager 来呢？这个和 Android 机制有关系。所有 Service 都需要加入到 ServiceManager 来管理。同时也方便了 Client 来查询系统存在哪些 Service，没看见我们传入了字符串吗？这样就可以通过 Human Readable 的字符串来查找 Service 了。

---》感觉没说清楚...饶恕我吧。

2.7 addService

addService 是调用的 BpServiceManager 的函数。前面略去没讲，现在我们看看。

```
virtual status_t addService(const String16& name, const sp<IBinder>& service)
    {
        Parcel data, reply;
//data 是发送到 BnServiceManager 的命令包
//看见没？先把 Interface 名字写进去，也就是什么 android.os.IServiceManager
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
//再把新 service 的名字写进去 叫 media.player
        data.writeString16(name);
//把新服务 service—>就是 MediaPlayerService 写到命令中
        data.writeStrongBinder(service);
//调用 remote 的 transact 函数
        status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
        return err == NO_ERROR ? reply.readInt32() : err;
    }
```

我的天，remote()返回的是什么？

remote(){ return mRemote; }-->啊？找不到对应的实际对象了？？？

还记得我们刚才初始化时候说的：

“这里的参数又叫 remote，唉，真是害人不浅啊“

原来，这里的 mRemote 就是最初创建的 BpBinder..

好吧，到那里去看看：

```
BpBinder 的位置在 framework\base\libs\binder\BpBinder.cpp
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    //又绕回去了，调用 IPCThreadState 的 transact。
    //注意啊，这里的 mHandle 为0,code 是 ADD_SERVICE_TRANSACTION,data 是命令包
    //reply 是回复包，flags=0
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
```

```
            return status;
        }
...
}
```

再看看 IPCThreadState 的 transact 函数把

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err = data.errorCheck();

    flags |= TF_ACCEPT_FDS;

    if (err == NO_ERROR) {
        //调用 writeTransactionData 发送数据
       err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

      if ((flags & TF_ONE_WAY) == 0) {
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
      ....等回复
        err = waitForResponse(NULL, NULL);
  ....
    return err;
}
```

再进一步，瞧瞧这个...

```
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
```

```
    int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
{
    binder_transaction_data tr;

    tr.target.handle = handle;
    tr.code = code;
    tr.flags = binderFlags;

    const status_t err = data.errorCheck();
    if (err == NO_ERROR) {
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    }
    ....
    上面把命令数据封装成 binder_transaction_data，然后
    写到 mOut 中，mOut 是命令的缓冲区，也是一个 Parcel
    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));
    //仅仅写到了 Parcel 中，Parcel 好像没和/dev/binder 设备有什么关联啊？
    恩，那只能在另外一个地方写到 binder 设备中去了。难道是在？
    return NO_ERROR;
}
//说对了，就是在 waitForResponse 中
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    int32_t cmd;
    int32_t err;

    while (1) {
    //talkWithDriver，哈哈，应该是这里了
```

```
            if ((err=talkWithDriver()) < NO_ERROR) break;
            err = mIn.errorCheck();
            if (err < NO_ERROR) break;
            if (mIn.dataAvail() == 0) continue;
            //看见没？这里开始操作 mIn 了，看来 talkWithDriver 中
//把 mOut 发出去，然后从 driver 中读到数据放到 mIn 中了。
            cmd = mIn.readInt32();

            switch (cmd) {
            case BR_TRANSACTION_COMPLETE:
                if (!reply && !acquireResult) goto finish;
                break;
   .....
     return err;
}
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
binder_write_read bwr;
   //中间东西太复杂了，不就是把 mOut 数据和 mIn 接收数据的处理后赋值给 bwr 吗？
          status_t err;
          do {
     //用 ioctl 来读写
                if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
                    err = NO_ERROR;
                else
                    err = -errno;
       } while (err == -EINTR);
     //到这里，回复数据就在 bwr 中了，bmr 接收回复数据的 buffer 就是 mIn 提供的
                if (bwr.read_consumed > 0) {
                    mIn.setDataSize(bwr.read_consumed);
                    mIn.setDataPosition(0);
                }
```

```
    return NO_ERROR;
}
```

好了，到这里，我们发送 addService 的流程就彻底走完了。

BpServiceManager 发送了一个 addService 命令到 BnServiceManager，然后收到回复。

先继续我们的 main 函数。

```
int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
     MediaPlayerService::instantiate();
     ---》该函数内部调用 addService，把 MediaPlayerService 信息 add 到 ServiceManager 中
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

这里有个容易搞晕的地方：

MediaPlayerService 是一个 BnMediaPlayerService,那么它是不是应该等着

BpMediaPlayerService 来和他交互呢?但是我们没看见 MediaPlayerService 有打开 binder 设备的操作啊！

这个嘛，到底是继续 addService 操作的另一端 BnServiceManager 还是先说

BnMediaPlayerService 呢？

还是先说 BnServiceManager 吧。顺便把系统的 Binder 架构说说。

2.8 BnServiceManager

上面说了，defaultServiceManager 返回的是一个 BpServiceManager，通过它可以把命令请求发送到 binder 设备，而且 handle 的值为0。那么，系统的另外一端肯定有个接收命令的，那又是谁呢？

很可惜啊，BnServiceManager 不存在，但确实有一个程序完成了 BnServiceManager 的工作，那就是 service.exe(如果在 windows 上一定有 exe 后缀，叫 service 的名字太多了，这里加 exe 就表明它是一个程序)

位置在 framework/base/cmds/servicemanger.c 中。

```
int main(int argc, char **argv)
{
    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);//处理 BpServiceManager 发过来的命令
}
```

看看 binder_open 是不是和我们猜得一样？

```
struct binder_state *binder_open(unsigned mapsize)
{
    struct binder_state *bs;
    bs = malloc(sizeof(*bs));
   ....
    bs->fd = open("/dev/binder", O_RDWR);//果然如此
  ....
    bs->mapsize = mapsize;
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
  }
```

再看看 binder_become_context_manager

```
int binder_become_context_manager(struct binder_state *bs)
{
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);//把自己设为 MANAGER
}
```

binder_loop 肯定是从 binder 设备中读请求，写回复的这么一个循环吧？

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
    int res;
    struct binder_write_read bwr;
    readbuf[0] = BC_ENTER_LOOPER;
```

```
    binder_write(bs, readbuf, sizeof(unsigned));
    for (;;) {//果然是循环
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (unsigned) readbuf;

        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
      //哈哈，收到请求了，解析命令
        res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
 }
```

这个...后面还要说吗？？

恩，最后有一个类似 handleMessage 的地方处理各种各样的命令。这个就是 svcmgr_handler,就在 ServiceManager.c 中

```
int svcmgr_handler(struct binder_state *bs,
                   struct binder_txn *txn,
                   struct binder_io *msg,
                   struct binder_io *reply)
{
    struct svcinfo *si;
    uint16_t *s;
    unsigned len;
    void *ptr;

    s = bio_get_string16(msg, &len);
    switch(txn->code) {
    case SVC_MGR_ADD_SERVICE:
        s = bio_get_string16(msg, &len);
        ptr = bio_get_ref(msg);
        if (do_add_service(bs, s, len, ptr, txn->sender_euid))
            return -1;
        break;
...
```

其中，do_add_service 真正添加 BnMediaService 信息

```
int do_add_service(struct binder_state *bs,
                   uint16_t *s, unsigned len,
                   void *ptr, unsigned uid)
{
    struct svcinfo *si;
    si = find_svc(s, len);s 是一个 list
     si = malloc(sizeof(*si) + (len + 1) * sizeof(uint16_t));
        si->ptr = ptr;
         si->len = len;
         memcpy(si->name, s, (len + 1) * sizeof(uint16_t));
         si->name[len] = '\0';
         si->death.func = svcinfo_death;
         si->death.ptr = si;
         si->next = svclist;
         svclist = si; //看见没，这个 svclist 是一个列表，保存了当前注册到 ServiceManager
中的信息
    }
  binder_acquire(bs, ptr);//这个吗。当这个 Service 退出后，我希望系统通知我一下，好释放
上面 malloc 出来的资源。大概就是干这个事情的。
    binder_link_to_death(bs, ptr, &si->death);
    return 0;
}
```

喔，对于 addService 来说，看来 ServiceManager 把信息加入到自己维护的一个服务列表中了。

2.9 ServiceManager 存在的意义

为何需要一个这样的东西呢？

原来，Android 系统中 Service 信息都是先 add 到 ServiceManager 中，由 ServiceManager 来集中管理，这样就可以查询当前系统有哪些服务。而且,Android 系统中某个服务例如 MediaPlayerService 的客户端想要和 MediaPlayerService 通讯的话，必须先向 ServiceManager 查询 MediaPlayerService 的信息，然后通过 ServiceManager 返回的东西再来和 MediaPlayerService 交互。

毕竟，要是 MediaPlayerService 身体不好，老是挂掉的话，客户的代码就麻烦了，就不知道

后续新生的 MediaPlayerService 的信息了，所以只能这样：

- MediaPlayerService 向 SM 注册
- MediaPlayerClient 查询当前注册在 SM 中的 MediaPlayerService 的信息
- 根据这个信息，MediaPlayerClient 和 MediaPlayerService 交互

另外，ServiceManager 的 handle 标示是0，所以只要往 handle 是0的服务发送消息了，最终都会被传递到 ServiceManager 中去。

三 MediaService 的运行

上一节的知识，我们知道了：

- defaultServiceManager 得到了 BpServiceManager，然后 MediaPlayerService 实例化后，调用 BpServiceManager 的 addService 函数
- 这个过程中，是 service_manager 收到 addService 的请求，然后把对应信息放到自己保存的一个服务 list 中

到这儿，我们可看到，service_manager 有一个 binder_looper 函数，专门等着从 binder 中接收请求。虽然 service_manager 没有从 BnServiceManager 中派生，但是它肯定完成了 BnServiceManager 的功能。

同样，我们创建了 MediaPlayerService 即 BnMediaPlayerService，那它也应该：

- 打开 binder 设备
- 也搞一个 looper 循环，然后坐等请求

service，service，这个和网络编程中的监听 socket 的工作很像嘛！

好吧，既然 MediaPlayerService 的构造函数没有看到显示的打开 binder 设备，那么我们看看它的父类即 BnXXX 又到底干了些什么呢？

3.1 MediaPlayerService 打开 binder

```
class MediaPlayerService : public BnMediaPlayerService
// MediaPlayerService 从 BnMediaPlayerService 派生
//而 BnMediaPlayerService 从 BnInterface 和 IMediaPlayerService 同时派生
class BnMediaPlayerService: public BnInterface<IMediaPlayerService>
{
public:
```

```
    virtual status_t      onTransact( uint32_t code,
                                      const Parcel& data,
                                      Parcel* reply,
                                      uint32_t flags = 0);
};
```

看起来，BnInterface 似乎更加和打开设备相关啊。

```
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface>          queryLocalInterface(const String16& _descriptor);
    virtual const String16&         getInterfaceDescriptor() const;

protected:
    virtual IBinder*                onAsBinder();
};
```

兑现后变成

```
class BnInterface : public IMediaPlayerService, public BBinder
```

BBinder?BpBinder？是不是和 BnXXX 以及 BpXXX 对应的呢？如果是，为什么又叫 BBinder 呢？

```
BBinder::BBinder()
    : mExtras(NULL)
{
//没有打开设备的地方啊？
}
```

完了？难道我们走错方向了吗？难道不是每个 Service 都有对应的 binder 设备 fd 吗？

.....

回想下，我们的 Main_MediaService 程序，有哪里打开过 binder 吗？

```
int main(int argc, char** argv)
{
//对啊，我在 ProcessState 中不是打开过 binder 了吗？
```

```
sp<ProcessState> proc(ProcessState::self());
sp<IServiceManager> sm = defaultServiceManager();
MediaPlayerService::instantiate();
.....
```

3.2 looper

啊?原来打开 binder 设备的地方是和进程相关的啊？一个进程打开一个就可以了。那么，我在哪里进行类似的消息循环 looper 操作呢？

```
...
//难道是下面两个？
ProcessState::self()->startThreadPool();
IPCThreadState::self()->joinThreadPool();
看看 startThreadPool 吧
void ProcessState::startThreadPool()
{
 ...
    spawnPooledThread(true);
}
void ProcessState::spawnPooledThread(bool isMain)
{
    sp<Thread> t = new PoolThread(isMain);isMain 是 TRUE
//创建线程池，然后 run 起来，和 java 的 Thread 何其像也。
    t->run(buf);
 }
PoolThread 从 Thread 类中派生，那么此时会产生一个线程吗？看看 PoolThread 和 Thread 的构造吧
PoolThread::PoolThread(bool isMain)
        : mIsMain(isMain)
    {
    }
Thread::Thread(bool canCallJava)//canCallJava 默认值是 true
    :   mCanCallJava(canCallJava),
```

```
            mThread(thread_id_t(-1)),
            mLock("Thread::mLock"),
            mStatus(NO_ERROR),
            mExitPending(false), mRunning(false)
{
}
```

喔，这个时候还没有创建线程呢。然后调用 PoolThread::run，实际调用了基类的 run。

```
status_t Thread::run(const char* name, int32_t priority, size_t stack)
{
 bool res;
    if (mCanCallJava) {
            res = createThreadEtc(_threadLoop,//线程函数是_threadLoop
                        this, name, priority, stack, &mThread);
    }
//终于，在 run 函数中，创建线程了。从此
 主线程执行
IPCThreadState::self()->joinThreadPool();
```

新开的线程执行_threadLoop

我们先看看_threadLoop

```
int Thread::_threadLoop(void* user)
{
    Thread* const self = static_cast<Thread*>(user);
    sp<Thread> strong(self->mHoldSelf);
    wp<Thread> weak(strong);
    self->mHoldSelf.clear();

    do {
 ...
            if (result && !self->mExitPending) {
                        result = self->threadLoop();哇塞，调用自己的 threadLoop
                }
            }
```

我们是 PoolThread 对象，所以调用 PoolThread 的 threadLoop 函数

```
virtual bool PoolThread ::threadLoop()
    {
//mIsMain 为 true。
//而且注意，这是一个新的线程，所以必然会创建一个
新的 IPCThreadState 对象（记得线程本地存储吗？TLS），然后
IPCThreadState::self()->joinThreadPool(mIsMain);
            return false;
    }
```

主线程和工作线程都调用了 joinThreadPool，看看这个干嘛了！

```
void IPCThreadState::joinThreadPool(bool isMain)
{
      mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
      status_t result;
    do {
            int32_t cmd;
              result = talkWithDriver();
              result = executeCommand(cmd);
             }
          } while (result != -ECONNREFUSED && result != -EBADF);

    mOut.writeInt32(BC_EXIT_LOOPER);
    talkWithDriver(false);
}
```

看到没？有 loop 了，但是好像是有两个线程都执行了这个啊！这里有两个消息循环？

下面看看 executeCommand

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
BBinder* obj;
    RefBase::weakref_type* refs;
    status_t result = NO_ERROR;
case BR_TRANSACTION:
```

```
            {
                binder_transaction_data tr;
                result = mIn.read(&tr, sizeof(tr));
//来了一个命令，解析成 BR_TRANSACTION,然后读取后续的信息
            Parcel reply;
                 if (tr.target.ptr) {
//这里用的是 BBinder。
                    sp<BBinder> b((BBinder*)tr.cookie);
                    const status_t error = b->transact(tr.code, buffer, &reply, 0);
}
让我们看看 BBinder 的 transact 函数干嘛了
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
就是调用自己的 onTransact 函数嘛
err = onTransact(code, data, reply, flags);
return err;
}
```

BnMediaPlayerService 从 BBinder 派生，所以会调用到它的 onTransact 函数

终于水落石出了，让我们看看 BnMediaPlayerServcice 的 onTransact 函数。

```
status_t BnMediaPlayerService::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
// BnMediaPlayerService 从 BBinder 和 IMediaPlayerService 派生，所有 IMediaPlayerService
//看到下面的 switch 没？所有 IMediaPlayerService 提供的函数都通过命令类型来区分
//
    switch(code) {
        case CREATE_URL: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            create 是一个虚函数，由 MediaPlayerService 来实现!!
             sp<IMediaPlayer> player = create(
```

```
                pid, client, url, numHeaders > 0 ? &headers : NULL);


            reply->writeStrongBinder(player->asBinder());
            return NO_ERROR;
        } break;
```

其实，到这里，我们就明白了。BnXXX 的 onTransact 函数收取命令，然后派发到派生类的函数，由他们完成实际的工作。

说明：

这里有点特殊，startThreadPool 和 joinThreadPool 完后确实有两个线程，主线程和工作线程，而且都在做消息循环。为什么要这么做呢？他们参数 isMain 都是 true。不知道 google 搞什么。难道是怕一个线程工作量太多，所以搞两个线程来工作？这种解释应该也是合理的。

网上有人测试过把最后一句屏蔽掉，也能正常工作。但是难道主线程提出了，程序还能不退出吗？这个...管它的，反正知道有两个线程在那处理就行了。

四 MediaPlayerClient

这节讲讲 MediaPlayerClient 怎么和 MediaPlayerService 交互。

使用 MediaPlayerService 的时候，先要创建它的 BpMediaPlayerService。我们看看一个例子

```
IMediaDeathNotifier::getMediaPlayerService()
{
        sp<IServiceManager> sm = defaultServiceManager();
        sp<IBinder> binder;
        do {
//向 SM 查询对应服务的信息，返回 binder
binder = sm->getService(String16("media.player"));
            if (binder != 0) {
                break;
             }
             usleep(500000); // 0.5 s
        } while(true);


//通过 interface_cast，将这个 binder 转化成 BpMediaPlayerService
```

```
//注意，这个 binder 只是用来和 binder 设备通讯用的，实际
//上和 IMediaPlayerService 的功能一点关系都没有。
//还记得我说的 Bridge 模式吗?BpMediaPlayerService 用这个 binder 和 BnMediaPlayerService
//通讯。
    sMediaPlayerService = interface_cast<IMediaPlayerService>(binder);
    }
    return sMediaPlayerService;
}
```

为什么反复强调这个 Bridge？其实也不一定是 Bridge 模式，但是我真正想说明的是：

Binder 其实就是一个和 binder 设备打交道的接口，而上层 IMediaPlayerService 只不过把它当做一个类似 socket 使用罢了。我以前经常把 binder 和上层类 IMediaPlayerService 的功能混到一起去。

当然，你们不一定会犯这个错误。但是有一点请注意：

4.1 Native 层

刚才那个 getMediaPlayerService 代码是 C++层的，但是整个使用的例子确实 JAVA->JNI 层的调用。如果我要写一个纯 C++的程序该怎么办？

```
int main()
{
 getMediaPlayerService();直接调用这个函数能获得 BpMediaPlayerService 吗？
不能，为什么？因为我还没打开 binder 驱动呐！但是你在 JAVA 应用程序里边却有 google 已经替你
封装好了。
所以，纯 native 层的代码，必须也得像下面这样处理：
sp<ProcessState> proc(ProcessState::self());//这个其实不是必须的，因为
//好多地方都需要这个，所以自动也会创建.
getMediaPlayerService();
还得起消息循环呐，否则如果 Bn 那边有消息通知你，你怎么接受得到呢？
ProcessState::self()->startThreadPool();
//至于主线程是否也需要调用消息循环，就看个人而定了。不过一般是等着接收其他来源的消息，例如 socket 发来的命令，然后控制 MediaPlayerService 就可以了。
}
```

五 实现自己的 Service

好了，我们学习了这么多 Binder 的东西，那么想要实现一个自己的 Service 该咋办呢？

如果是纯 C++程序的话，肯定得类似 main_MediaService 那样干了。

```
int main()
{
 sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    sm->addService("service.name",new XXXService());
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

看看 XXXService 怎么定义呢？

我们需要一个 Bn，需要一个 Bp，而且 Bp 不用暴露出来。那么就在 BnXXX.cpp 中一起实现好了。

另外，XXXService 提供自己的功能，例如 getXXX 调用

5.1定义 XXX 接口

XXX 接口是和 XXX 服务相关的，例如提供 getXXX，setXXX 函数，和应用逻辑相关。

需要从 IInterface 派生

```
class IXXX: public IInterface
{
public:
    DECLARE_META_INTERFACE(XXX);申明宏
    virtual getXXX() = 0;
    virtual setXXX() = 0;
}这是一个接口。
```

5.2定义 BnXXX 和 BpXXX

为了把 IXXX 加入到 Binder 结构，需要定义 BnXXX 和对客户端透明的 BpXXX。

其中 BnXXX 是需要有头文件的。BnXXX 只不过是把 IXXX 接口加入到 Binder 架构中来，

而不参与实际的 getXXX 和 setXXX 应用层逻辑。

这个 BnXXX 定义可以和上面的 IXXX 定义放在一块。分开也行。

```
class BnXXX: public BnInterface<IXXX>
{
public:
    virtual status_t      onTransact( uint32_t code,
                                      const Parcel& data,
                                      Parcel* reply,
                                      uint32_t flags = 0);
//由于 IXXX 是个纯虚类，而 BnXXX 只实现了 onTransact 函数，所以 BnXXX 依然是
一个纯虚类
};
```

有了 DECLARE，那我们在某个 CPP 中 IMPLEMNT 它吧。那就在 IXXX.cpp 中吧。

```
IMPLEMENT_META_INTERFACE(XXX, "android.xxx.IXXX");//IMPLEMENT 宏


status_t BnXXX::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case GET_XXX: {
            CHECK_INTERFACE(IXXX, data, reply);
           读请求参数
           调用虚函数 getXXX()
            return NO_ERROR;
        } break; //SET_XXX 类似
```

BpXXX 也在这里实现吧。

```
class BpXXX: public BpInterface<IXXX>
{
public:
    BpXXX (const sp<IBinder>& impl)
```

```
        : BpInterface< IXXX >(impl)
    {
    }
    vitural getXXX()
    {
      Parcel data, reply;
      data.writeInterfaceToken(IXXX::getInterfaceDescriptor());
       data.writeInt32(pid);
       remote()->transact(GET_XXX, data, &reply);
       return;
    }
    //setXXX 类似
```

至此，Binder 就算分析完了，大家看完后，应该能做到以下几点：

- 如果需要写自己的 Service 的话，总得知道系统是怎么个调用你的函数，恩。对。有2个线程在那不停得从 binder 设备中收取命令，然后调用你的函数呢。恩，这是个多线程问题。

- 如果需要跟踪 bug 的话，得知道从 Client 端调用的函数，是怎么最终传到到远端的 Service。这样，对于一些函数调用，Client 端跟踪完了，我就知道转到 Service 去看对应函数调用了。反正是同步方式。也就是 Client 一个函数调用会一直等待到 Service 返回为止。