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。