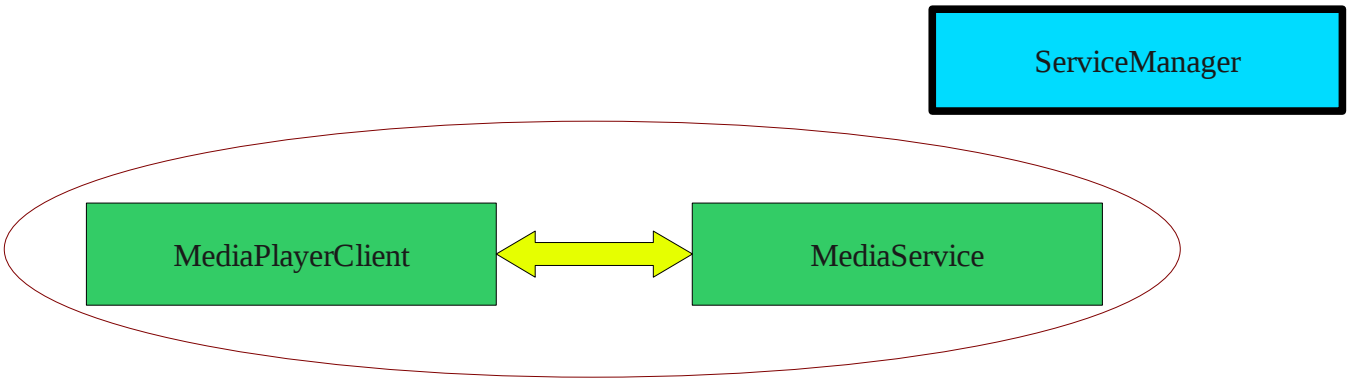


framework\base\Media\MediaServer\Main_mediaserver.cpp

```
int main(int argc, char** argv)
{
    //获得一个 ProcessState 实例
     sp<ProcessState> proc(ProcessState::self());
     //得到一个 ServiceManager 对象
   sp<IServiceManager> sm = defaultServiceManager();     //理解为指向 IServiceManager 的指针
   MediaPlayerService::instantiate();              //初始化 MediaPlayerService 服务
   ProcessState::self()->startThreadPool();       //启动 Process 的线程池
   IPCThreadState::self()->joinThreadPool();   //将自己加入到刚才的线程池
}
```

1.ProcessState:

framework\base\libs\binder\ProcessState.cpp

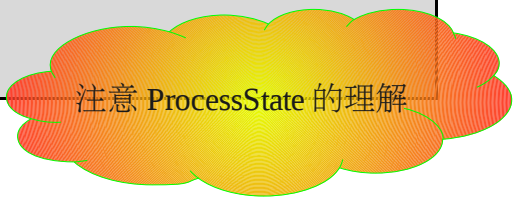
```
sp<ProcessState> ProcessState::self()
{
   if (gProcess != NULL) return gProcess;
   AutoMutex _l(gProcessMutex);      //锁保护
   if (gProcess == NULL) gProcess = new ProcessState;      //创建一个 ProcessState 对象
     return gProcess;
}
```

构造函数

再来看看 ProcessState 构造函数

```
ProcessState::ProcessState()                    //映射+open binder
    : mDriverFD(open_driver( ))
    , mVMStart(MAP_FAILED)
    , 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);
    }
    ...
}
```

```
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;
}
```



2.defaultServiceManager:

framework\base\libs\binder\IServiceManager.cpp

```
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;
}
```

Gprocess ->getContextObject

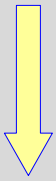
new BpBinder

实际返回的是： BpServiceManager

一个进程获得上下文意味着什么？

Gprocess ->getContextObject

```
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
        return getStrongProxyForHandle(0);          //注意，这里传入 0
    }
}
```

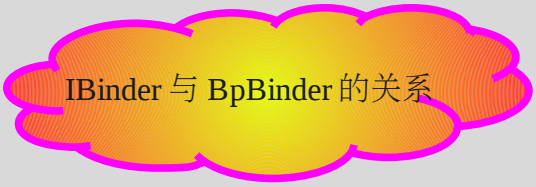


```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;
    AutoMutex _l(mLock);
    handle_entry* e = lookupHandleLocked(handle);
```

Handle == 0

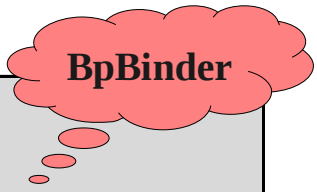
```
    struct handle_entry {
        IBinder* binder;
        RefBase::weakref_type* refs;
    };
```

```
    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);    //创建了一个新的 BpBinder，并返回
            e->binder = b;
            result = b;
        }....
    }
    return result;
}
```



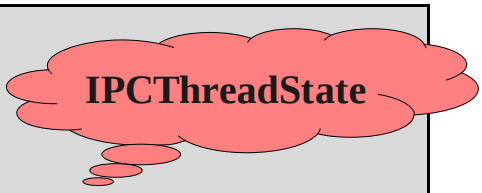

```
gDefaultServiceManager = interface_cast<IServiceManager>(
                                    ProcessState::self()->getContextObject(NULL));

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

framework\base\libs\binder\BpBinder.cpp




```
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);
}
```

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;  线程间不共享这些空间
        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;
}
```

```
IPCThreadState::IPCThreadState()
    : mProcess(ProcessState::self()), mMyThreadId(androidGetTid())
{
    pthread_setspecific(gTLS, this);
    clearCaller();
    mIn.setDataCapacity(256);
    mOut.setDataCapacity(256);
}
```

ProcessState

要获得服务进程的上下文，
就要 new 个 BpBinder 。
最后，
Gprocess->(IBinder)

BpBinder → IPCThreadState →

```
gDefaultServiceManager = interface_cast<IServiceManager>(
                                ProcessState::self()->getContextObject(NULL));

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

总之要返回一个
IBinder *

framework/base/include/binder/IInterface.h

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

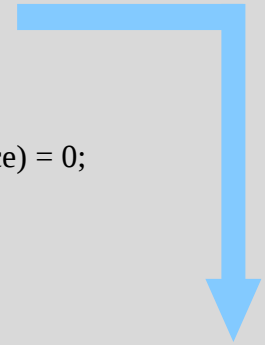
组合，这里就能生成 **BpServiceManager**

所以，上面等价于：

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

framework/base/include/binder/IServiceManager.h

```
class IServiceManager : public IInterface
{
public:
    DECLARE_META_INTERFACE(ServiceManager);
     virtual status_t   addService ( const String16& name,
                                     const sp<IBinder>& service) = 0;
};
```



```
#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;          //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
IServiceManager ();                                                 \
virtual ~IserviceManager();          //增加构造和虚析购函数...
```

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() { }                                  \
```

转化后:

```
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() { }


android::sp<IServiceManager> IServiceManager::asInterface(
         const android::sp<android::IBinder>& obj)
   {
      android::sp<IServiceManager> intr;
      if (obj != NULL) {
         ....
```

```
            intr = new BpServiceManager(obj);
        }
    }
    return intr;
}
```

小总结：

sp<IServiceManager> sm = defaultServiceManager();

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。

```
int main(int argc, char** argv)
{
    //获得一个 ProcessState 实例
    sp<ProcessState> proc(ProcessState::self());
    //得到一个 ServiceManager 对象: BpServiceManager
    sp<IServiceManager> sm = defaultServiceManager();  //理解为指向 IServiceManager 的指针

   MediaPlayerService::instantiate();          //初始化 MediaPlayerService 服务
   ProcessState::self()->startThreadPool();    //启动 Process 的线程池
   IPCThreadState::self()->joinThreadPool();   //将自己加入到刚才的线程池
}
```

下面开始添加 **Media** 服务。。。

framework\base\media\libmediaplayerservice\libMediaPlayerService.cpp

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

有了服务进程，为进程赋予任务

要创建服务 BnMediaPlayerService —派生出→ MediaPlayerService

BpServiceManager->addService() —通知→ ServiceManager

```
MediaPlayerService::MediaPlayerService()
{
    LOGV("MediaPlayerService created");
    mNextConnId = 1;
}
```

addService：

```
virtual status_t  addService (const String16& name,
                              const sp<IBinder>& service)
    {
        Parcel data;          //发送到 BnServiceManager 的命令包
        Parcel reply;
//先把 Interface 名字写进去，也就是 android.os.IServiceManager
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
```

String16("media.player"), new MediaPlayerService()

IMPLEMENT_META_INTERFACE

```
//再把新 service 的名字写进去 叫 media.player
        data.writeString16(name);
//把新服务 MediaPlayerService 写到命令中
        data.writeStrongBinder(service);

//调用 remote 的 transact 函数
                        remote(){ return mRemote; }    ------>    BpBinder
        status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
        return err == NO_ERROR ? reply.readInt32() : err;
      }
```

framework\base\libs\binder\BpBinder.cpp

```
status_t BpBinder::transact(  uint32_t code,   const Parcel& data,   Parcel* reply,   uint32_t flags )
{
        status_t status = IPCThreadState::self()->transact(
                                                mHandle, code, data, reply, flags );
     // mHandle=0，code=ADD_SERVICE_TRANSACTION，data 是命令包，reply 是回复包，flags=0
        if (status == DEAD_OBJECT)  mAlive = 0;
        return status;
    }
...
}
```

IPCThreadState::self()- 是什么
和 BpBinder 的关系

```
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);
    }                                                    cmd

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

```
//发送数据给谁呢——当然是 binder
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();
    }
     ....
     //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));

    return NO_ERROR;
}
```

```
status_t IPCThreadState::waitForResponse(Parcel *reply,  status_t *acquireResult)
{
   int32_t cmd;
   int32_t err;
     while (1) {
      //与 binder 终于发生了关系：把 mOut 发出去，然后从 driver 中读到数据放到 mIn 中
       if (( err= talkWithDriver () ) < NO_ERROR) break;
       err = mIn.errorCheck();
       if (err < NO_ERROR) break;
       if (mIn.dataAvail() == 0) continue;
       cmd = mIn.readInt32();
       switch (cmd) {
       case BR_TRANSACTION_COMPLETE:
          if (!reply && !acquireResult) goto finish;
          break;
  .....
   return err;
}
```

talkWithDriver：

```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
        binder_write_read    bwr;
        // 把 mOut 数据发出，而后 mIn 数据会赋值给 bwr
        status_t err;
        do {
           if ( ioctl(mProcess->mDriverFD,  BINDER_WRITE_READ,  &bwr) >= 0 )
               err = NO_ERROR;
```

/dev/binder

```
            else
                err = -errno;
        } while (err == -EINTR);
        //到这里，回复数据就在 bwr 中了，bwr 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;
}
```

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 ***Bn***ServiceManager，然后收到回复。

虚构的

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();
}
```

MediaPlayerService 是一个 BnMediaPlayerService，

那么它是不是应该等着 BpMediaPlayerService 来和他交互呢?

framework/base/cmds/servicemanger.c

ServiceManager

```
int main(int argc, char **argv)
{
    struct binder_state  *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;
    bs = binder_open(128*1024);          //打开 binder 设备

    binder_become_context_manager (bs)   //成为 manager：   ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);

    svcmgr_handle = svcmgr;
    binder_loop (bs, svcmgr_handler);          //处理 BpServiceManager 发过来的命令
}
```

```
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);
```

```
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 (;;) {

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

```
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))  //真正添加 BnMediaService 信息
            return -1;
        break;
...
}
```

```
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;          //svclist 是一个列表，保存了当前注册到 ServiceManager 中的信息
    binder_acquire(bs, ptr);   //Service 退出，系统通知下好释放上面 malloc 出来的资源
    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。

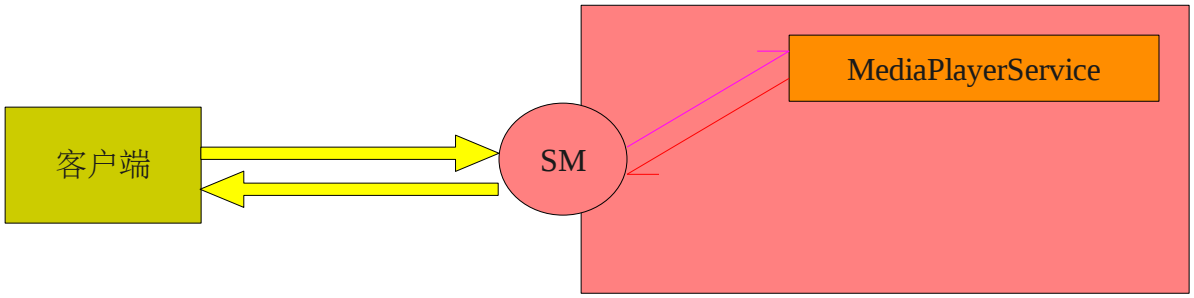
阶段性总结：

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。

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。

DefaultServiceManager 得到 **BpServiceManager**，

然后 **MediaPlayerService** 实例化后，调用 **BpServiceManager->addService** 函数。

ServiceManager 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。

ServiceManager 的 ***binder_looper*** 函数，专门等着从 **binder** 中接收请求。

BnServiceManager：各种服务都从这里派生。

ServiceManager 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。

MediaPlayerService 打开 binder

——MediaPlayerService 的构造函数没有看到显式的打开 binder 设备

sp<ProcessState> proc(ProcessState::self()) 打开过

打开 binder 设备的地方和进程相关
一个进程打开一个就可以了

class ***MediaPlayerService*** : public **BnMediaPlayerService**

1 2

class **BnMediaPlayerService**: public BnInterface<IMediaPlayerService>

{ 3

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**

//难道是下面两个?

ProcessState::self()->startThreadPool();

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

```
void ProcessState :: startThreadPool()
{
  ...
   spawnPooledThread(true);
}
```

```
PoolThread::PoolThread(bool isMain)
      : mIsMain(isMain)
  {

  }
```

```
  void ProcessState::spawnPooledThread(bool isMain)
  {
     sp<Thread> t = new PoolThread(isMain);  //isMain 是 TRUE


    // 创建线程池，然后 run 起来
     t->run(buf);
   }
```

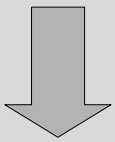
```
Thread::Thread(bool canCallJava)      //canCallJava 默认值是 true
    :   mCanCallJava(canCallJava),
        mThread(thread_id_t(-1)),
        mLock("Thread::mLock"),
        mStatus(NO_ERROR),
        mExitPending(false), mRunning(false)
{

}
```

```
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);
    }
```

```
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();       //调用自己的 threadLoop
           }
        }
}
virtual bool PoolThread  ::threadLoop()
    {
          IPCThreadState::self()->joinThreadPool(mIsMain);
          return false;
    }

// 主线程和工作线程都调用了 joinThreadPool。
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;
//来了一个命令，解析成 BR_TRANSACTION,然后读取后续的信息
case BR_TRANSACTION:
        {
            binder_transaction_data  tr;
            result = mIn.read(&tr, sizeof(tr));
            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);
}
```



```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
            err = onTransact (code, data, reply, flags);
            return err;
}
```

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

```
class BnMediaPlayerService: public BnInterface<IMediaPlayerService>
                                    class BnInterface : public IMediaPlayerService, public BBinder
```

终于水落石出，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;
```

BnXXX 的 onTransact 函数收取命令

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。

MediaPlayerClient

—— MediaPlayerClient 如何与 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
//这个 binder 只是用来和 binder 设备通讯用的，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：

实现自己的 Service

```
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

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;
}
```

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;
        }
        case SET_XXX: {
        }
//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
```

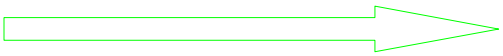
至此，Binder 就算分析完了，大家看完后，应该能做到以下几点：

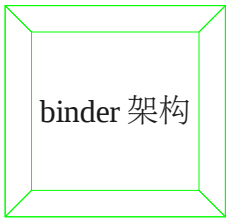
如果需要写自己的 Service 的话，总得知道系统是怎么个调用你的函数，恩。对。有 2 个线程在那不停得从 binder 设备中收取命令，然后调用你的函数呢。恩，这是个多线程问题。

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。

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

```
定义接口:
class IXXX: public IInterface
{
    public:
    DECLARE_META_INTERFACE(XXX);
    virtual getXXX() = 0;
    virtual setXXX() = 0;
}
```






```
class BnXXX: public BnInterface<IXXX>
{
public:
    virtual status_t    onTransact( uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);
};
```

```
IMPLEMENT_META_INTERFACE(XXX, "android.xxx.IXXX");
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;
        }
        case SET_XXX: {
        }
}
```

```
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;
}
vitural setXXX()
{
}
```