修改记录

版本	日期	作者
V0.01	10 年 9 月 7 日	徐申龙

目录

C++ 类结构

frameworks/base/camera/tests/CameraServiceTest/CameraServiceTest.cpp 为例，对 IHolder 接口的实现分析。设当前路径为 Android 工程根目录：

编译前准备

$. build/envsetup.sh

$adb remount

编译和更新设备中的库 libbinder

$mmm frameworks/base/libs/binder

$adb push out/target/product/generic/system/lib/libbinder.so /system/lib

$ DISABLE_AUTO_INSTALLCLEAN=true mmm frameworks/base/camera/tests/CameraServiceTest/

$ adb push out/target/product/generic/system/bin/CameraServiceTest /system/bin

$ DISABLE_AUTO_INSTALLCLEAN=true mmm frameworks/base/libs/camera/
$ adb push out/target/product/generic/system/lib/libcamera_client.so /system/lib

类结构解析

BnXXX; Native, just service； BpXXX: Proxy，is client。注意这两个模板类的声明：
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder

template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase

比如，形如 class BnMediaPlayerClient: public BnInterface<IMediaPlayerClient> 类声明，意味着 BnMediaPlayerClient 实际继承了 IMediaPlayerClient 类。
所以， class BnHolder : public BnInterface<IHolder>，
class BpHolder : public BpInterface<IHolder>
这种继承，说明了 BnHolder 和 BpHolder 都继承了 IHolder。

BpHolder 是负责把 IHolder 的各种方法封装到 transact 方法，然后以 Binder 传到服务器端。
BnHolder 的方法 onTransact 是 BpHolder 方法 transact 的对接方法，她把受到消息解析成 IHolder 的各个方法，
这些服务器的 IHolder 方法实现才是真正意思的接口实现。如果 BnHolder 不实现 onTransact，则使用 BBinder::onTransact。
但是，BnHolder 并没有实现 IHolder 的各个接口，仅仅是实现了 onTransact 的接口。真正实现 IHolder 的是 HolderService 类。

IBinder 与接口之间的转换

IInterface 的 asBinder（onAsBinder 是其辅助方法）是把 IInterface 类转换成 IBinder 类，与之相反，接口（如 IHolder）的方法 asInterface 是把 IBinder 转换为 IIterface。

asInterface 是用宏 DECLARE_META_INTERFACE、IMPLEMENT_META_INTERFACE 声明和定义的，每个接口（如 IHolder）必须使用这两个宏。
由于 BnInterface 既继承了 IBinder，又继承了 IHolder，他在这两种转换之间都是平滑的，直接返回自己的 this 即可。
对应 BpInterface，其本身是一个 IHolder 的派生类，同时，其父类 BpRefBase 有个 IBinder 常指针成员 mRemote，及相应的方法 remote 来获取之。

IInterface.h 对 onAsBinder 和 asInterface
onAsBinder 在 BnInterface 的实现就是返回 this（即自身）。而在 onAsBinder 的实现就是调用方法 remote();
asBinder 是 IIterface 直接实现了（IIterface.cpp），就是调用 onAsBinder，不用派生类重载。
asInterface 是在宏 IMPLEMENT_META_INTERFACE（IInterface.h 定义）中实现，他是一个接口，如 IHolder 的方法，
他首先让参数 IBinder 调用 queryLocalInterface，即查询本地接口，此处 local 与 native 一样，就是服务器端的意思，也就是看看这个 IBinder 类型的参数是不是指向一个 BnInterface 对象，如果是则直接返回 BnInterface 自己。否则，是一个普通 IBinder 对象，应该是客户端在查询，则新建一个 BpInterface 派生类的对象（如 BpHolder）。

服务启动
defaultServiceManager()->addService(String16("CameraServiceTest.Holder"),
new HolderService());
ProcessState::self()->startThreadPool();
spawPooledThread()
PoolThread::run() ，建立了一个新线程。

客户端获取服务
defaultServiceManager 返回 IServiceManager，通过 IServiceManager 的方法 getService 获取 IBinder，通过 interface_cast 把接口从 IBinder 类型转变为 IHolder 类型。
而 Interface_cast 是一个宏，最后，调用 IHolder::asInterface 接口。

我们知道，IBinder 是用于 IPC 的，那么，CameraServiceTest 是怎么做的呢？我们从 main 函数入手，他已开始判断参数个数，如果大于 1，则把第一个参数作为函数名、查表、执行之（runFunction(argv[1]);），然后退出。之后，才是运行服务（runHolderService）。所以，我们执行测试时，命令不能带参数。否则，服务没运行，怎么能测试呢！那么，这个判断代码段有什么含义呢？
再往下看，运行服务后，testConnect、testAllowConnectOnceOnly、testReconnect 、testLockUnlock 等测试，我修改了代码直接测试 testLockUnlock。一看 testLockUnlock 函数实现，遇到了调用 runInAnotherProcess，runInAnotherProcess 实现很简单，系统调

用 fork、加上库函数 execlp，就建立了子进程。fork 还好理解，但是再用了 execlp，子进程是要从 main 函数开始执行，而且，这个子进程的程序还就是 CameraServiceTest，而且不知参数 tag（const char *）有何用途，太乱了！不要急，我们回头看看 main 函数，其开头对程序参数的判断，就是对 tag 的判断，而且执行完 runFunction 的测试后，子进程就退出了，不会往下面执行。原来这段代码就是为了这些子进程服务的。
这样理解了，我们再看 Log 中 queryLocalInterface 的不同表现。interface_cast 进行类型转换的时候，有时使用了 IBinder 的 queryLocalInterface 实现 ，有时使用了 BnInterface 的 queryLocalInterface 实现。IBinder 的实际上就是返回 NULL，结果是 new BpInterface 对象；而
BnInterface 的 queryLocalInterface 是返回 BnInterface 自身指针（this）。这是为什么？还是回到 IPC 上面来看这个问题， interface_cast 转换时使用了 IBinder 的 queryLocalInterface 实现均是在子进程，相当于客户端；而使用了 BnInterface 的 queryLocalInterface 实现是在父进程，父进程是启动了服务的进程，所以对其来说 IPC 就是在自己进程内，返回自身即可。
再往深层次里说：

```
sp<IServiceManager> sm = defaultServiceManager();
ASSERT(sm != 0);
sp<IBinder> binder = sm->getService(String16("CameraServiceTest.Holder"));
```

同样是这段代码，在父进程（服务进程）和子进程（客户进程），获取的 binder 是不一样的，估计 defaultServiceManager 在父进程调用时，返回的对象，是一个 BnInterface 对象，如 CameraServiceTest 的 HolderService、ICameraService 对象；而在子进程的上下文中，defaultServiceManager 可能 new 了一个 IBinder 类实例。

Log

代码分布在如下文件中：
frameworks/base/include/binder/IInterface.h
frameworks/base/libs/camera/ICameraService.cpp
frameworks/base/camera/tests/CameraServiceTest/CameraServiceTest.cpp
frameworks/base/libs/binder/Binder.cpp
我在上面的代码中加了 log 消息，运行后的打印信息如下：

```
# CameraServiceTest
CameraServiceTest start, argv[0]:CameraServiceTest
void runHolderService()
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface android.os.IServiceManager,cls this =0x139e0
[this is IServiceManager asInterface obj:0x139e0, null obj:0x0]
queryLocalInterface failed, and new BpServiceManager
[--getCameraService -- interface_cast  =0x13ee8.]
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface android.hardware.ICameraService,cls this =0x13ee8
[this is ICameraService asInterface obj:0x13ee8, null obj:0x0]
```

==========BpCameraService constructor==========
queryLocalInterface failed, and new BpCameraService
[--getCameraService -- interface_cast done]
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface android.hardware.ICamera,cls this =0x145a0
[this is ICamera asInterface obj:0x145a0, null obj:0x0]
queryLocalInterface failed, and new BpCamera
void putTempObject(android::sp<android::IBinder>)
Hello , this is interface_cast, invoke asInterface
BnInterface queryLocalInterface desc CameraServiceTest.Holder, query CameraServiceTest.Holder, ptr is 0x13d68
[this is IHolder asInterface obj:0x13d6c, null obj:0x0]
getHolder done!!!

void runInAnotherProcess(const char*)
main argv[1] is testLockFailed
runFunction: testLockFailed
void testLockFailed()
android::sp<android::IBinder> getTempObject()
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface android.os.IServiceManager,cls this =0x139e0
[this is IServiceManager asInterface obj:0x139e0, null obj:0x0]
queryLocalInterface failed, and new BpServiceManager
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface CameraServiceTest.Holder,cls this =0x13e20
[this is IHolder asInterface obj:0x13e20, null obj:0x0]
queryLocalInterface failed, and new BpHolder
getHolder done!!!
!!!!!!!!!!!!!! BpHolder get!!!!!!!!!!!!!!!!!!!!
BnHolder onTransact!!!
==========holder service==========
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface android.hardware.ICamera,cls this =0x13d58
[this is ICamera asInterface obj:0x13d58, null obj:0x0]
queryLocalInterface failed, and new BpCamera
child process exit

void runInAnotherProcess(const char*)
main argv[1] is testLockUnlockSuccess
runFunction: testLockUnlockSuccess
void testLockUnlockSuccess()
android::sp<android::IBinder> getTempObject()

```
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface android.os.IServiceManager,cls this =0x139e0
[this is IServiceManager asInterface obj:0x139e0, null obj:0x0]
queryLocalInterface failed, and new BpServiceManager
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface CameraServiceTest.Holder,cls this =0x13e20
[this is IHolder asInterface obj:0x13e20, null obj:0x0]
queryLocalInterface failed, and new BpHolder
getHolder done!!!
!!!!!!!!!!!!!! BpHolder get!!!!!!!!!!!!!!!!!!!!
BnHolder onTransact!!!
====================holder service====================
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface android.hardware.ICamera,cls this =0x13d58
[this is ICamera asInterface obj:0x13d58, null obj:0x0]
queryLocalInterface failed, and new BpCamera
child process exit


 void runInAnotherProcess(const char*)
main argv[1] is testLockSuccess
runFunction: testLockSuccess
void testLockSuccess()
android::sp<android::IBinder> getTempObject()
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface android.os.IServiceManager,cls this =0x139e0
[this is IServiceManager asInterface obj:0x139e0, null obj:0x0]
queryLocalInterface failed, and new BpServiceManager
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface CameraServiceTest.Holder,cls this =0x13e20
[this is IHolder asInterface obj:0x13e20, null obj:0x0]
queryLocalInterface failed, and new BpHolder
getHolder done!!!
!!!!!!!!!!!!!! BpHolder get!!!!!!!!!!!!!!!!!!!!
BnHolder onTransact!!!
====================holder service====================
Hello , this is interface_cast, invoke asInterface
IBinder queryLocalInterface android.hardware.ICamera,cls this =0x13d58
[this is ICamera asInterface obj:0x13d58, null obj:0x0]
queryLocalInterface failed, and new BpCamera
child process exit

void clearTempObject()
```

Hello , this is interface_cast, invoke asInterface
BnInterface queryLocalInterface desc CameraServiceTest.Holder, query CameraServiceTest.Holder, ptr is 0x13d68
[this is IHolder asInterface obj:0x13d6c, null obj:0x0]
getHolder done!!!

Android IPC 通讯机制源码分析

Linux 系统中进程间通信的方式有:socket, named pipe,message queque, signal,share memory。Java 系统中的进程间通信方式有 socket, named pipe 等，android 应用程序理所当然可以应用 JAVA 的 IPC 机制实现进程间的通信，但我查看 android 的源码，在同一终端上的应用软件的通信几乎看不到这些 IPC 通信方式，取而代之的是 Binder 通信。Google 为什么要采用这种方式呢，这取决于 Binder 通信方式的高效率。 Binder 通信是通过 linux 的 binder driver 来实现的，Binder 通信操作类似线程迁移(thread migration)，两个进程间 IPC 看起来就象是一个进程进入另一个进程执行代码然后带着执行的结果返回。Binder 的用户空间为每一个进程维护着一个可用的线程池，线程池用于处理到来的 IPC 以及执行进程本地消息，Binder 通信是同步而不是异步。

Android 中的 Binder 通信是基于 Service 与 Client 的，所有需要 IBinder 通信的进程都必须创建一个 IBinder 接口，系统中有一个进程管理所有的 system service,Android 不允许用户添加非授权的 System service,当然现在源码开发了，我们可以修改一些代码来实现添加底层 system Service 的目的。对用户程序来说，我们也要创建 server,或者 Service 用于进程间通信，这里有一个 ActivityManagerService 管理 JAVA 应用层所有的 service 创建与连接(connect),disconnect,所有的 Activity 也是通过这个 service 来启动，加载的。ActivityManagerService 也是加载在 Systems Servcie 中的。

Android 虚拟机启动之前系统会先启动 service Manager 进程，service Manager 打开 binder 驱动，并通知 binder kernel 驱动程序这个进程将作为 System Service Manager，然后该进程将进入一个循环，等待处理来自其他进程的数据。用户创建一个 System service 后，通过 defaultServiceManager 得到一个远程 ServiceManager 的接口，通过这个接口我们可以调用 addService 函数将 System service 添加到 Service Manager 进程中，然后 client 可以通过 getService 获取到需要连接的目的 Service 的 IBinder 对象，这个 IBinder 是 Service 的 BBinder 在 binder kernel 的一个参考，所以 service IBinder 在 binder kernel 中不会存在相同的两个 IBinder 对象，每一个 Client 进程同样需要打开 Binder 驱动程序。对用户程序而言，我们获得这个对象就可以通过 binder kernel 访问 service 对象中的方法。Client 与 Service 在不同的进程中，通过这种方式实现了类似线程间的迁移的通信方式，对用户程序而言当调用 Service 返回的 IBinder 接口后，访问 Service 中的方法就如同调用自己的函数。
下图为 client 与 Service 建立连接的示意图

首先从 ServiceManager 注册过程来逐步分析上述过程是如何实现的。
ServiceMananger 进程注册过程源码分析：
Service Manager Process（Service_manager.c）：

Service_manager 为其他进程的 Service 提供管理，这个服务程序必须在 Android Runtime 起来之前运行，否则 Android JAVA Vm ActivityManagerService 无法注册。

```
int main(int argc, char **argv){
    struct binder_state *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;
        bs = binder_open(128*1024); //打开/dev/binder 驱动
        if (binder_become_context_manager(bs)) {//注册为 service manager in binder kernel
        LOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }
    svcmgr_handle = svcmgr;
```

```
    binder_loop(bs, svcmgr_handler);
    return 0;
}
```
首先打开 binder 的驱动程序然后通过 binder_become_context_manager 函数调用 ioctl 告诉 Binder Kernel 驱动程序这是一个服务管理进程，然后调用 binder_loop 等待来自其他进程的数据。BINDER_SERVICE_MANAGER 是服务管理进程的句柄，它的定义是：
```
/* the one magic object */
#define BINDER_SERVICE_MANAGER ((void*) 0)
```
如果客户端进程获取 Service 时所使用的句柄与此不符，Service Manager 将不接受 Client 的请求。客户端如何设置这个句柄在下面会介绍。
CameraSerivce 服务的注册(Main_mediaservice.c）
```
int main(int argc, char** argv){
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    LOGI("ServiceManager: %p", sm.get());
    AudioFlinger::instantiate();            //Audio 服务
    MediaPlayerService::instantiate();      //mediaPlayer 服务
    CameraService::instantiate();           //Camera 服务
    ProcessState::self()->startThreadPool(); //为进程开启缓冲池
    IPCThreadState::self()->joinThreadPool(); //将进程加入到缓冲池
}
```
CameraService.cpp
```
void CameraService::instantiate() {
    defaultServiceManager()->addService(
            String16("media.camera"), new CameraService());
}
```
创建 CameraService 服务对象并添加到 ServiceManager 进程中。
client 获取 remote IServiceManager IBinder 接口：
```
sp<IServiceManager> defaultServiceManager(){
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }
    return gDefaultServiceManager;
}
```
任何一个进程在第一次调用 defaultServiceManager 的时候 gDefaultServiceManager 值为 Null，所以该进程会通过 ProcessState::self 得到 ProcessState 实例。ProcessState 将打开 Binder 驱动。
ProcessState.cpp
```
sp<ProcessState> ProcessState::self()
{
    if (gProcess != NULL) return gProcess;

    AutoMutex _l(gProcessMutex);
    if (gProcess == NULL) gProcess = new ProcessState;
    return gProcess;
}
ProcessState::ProcessState()
: mDriverFD(open_driver()) //打开/dev/binder 驱动
{
```

```
//.....................
}
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
{
    if (supportsProcesses()) {
        return getStrongProxyForHandle(0);
    } else {
        return getContextObject(String16("default"), caller);
    }
}
```

Android 是支持 Binder 驱动的所以程序会调用 getStrongProxyForHandle。这里 handle 为 0，正好与 Service_manager 中的 BINDER_SERVICE_MANAGER 一致。

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;
    AutoMutex _l(mLock);
    handle_entry* e = lookupHandleLocked(handle);
        if (e != NULL) {
        // We need to create a new BpBinder if there isn't currently one, OR we
        // are unable to acquire a weak reference on this current one. See comment
        // in getWeakProxyForHandle() for more info about this.
        IBinder* b = e->binder; //第一次调用该函数 b 为 Null
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
            b = new BpBinder(handle);
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        } else {
            // This little bit of nastyness is to allow us to add a primary
            // reference to the remote proxy when this team doesn't have one
            // but another team is sending the handle to us.
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }
    return result;
}
```

第一次调用的时候 b 为 Null 所以会为 b 生成一 BpBinder 对象：

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
    LOGV("Creating BpBinder %p handle %d\n", this, mHandle);
        extendObjectLifetime(OBJECT_LIFETIME_WEAK);
    IPCThreadState::self()->incWeakHandle(handle);
}
void IPCThreadState::incWeakHandle(int32_t handle)
{
    LOG_REMOTEREFS("IPCThreadState::incWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_INCREFS);
    mOut.writeInt32(handle);
```

```
}
```
getContextObject 返回了一个 BpBinder 对象。
```
interface_cast<IServiceManager>(ProcessState::self()->getContextObject(NULL));
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```
将这个宏扩展后最终得到的是：
```
sp<IServiceManager> IServiceManager::asInterface(const sp<IBinder>& obj)
    {
        sp<IServiceManager> intr;
        if (obj != NULL) {
            intr = static_cast<IServiceManager*>(
                obj->queryLocalInterface(
                        IServiceManager::descriptor).get());
            if (intr == NULL) {
                intr = new BpServiceManager(obj);
            }
        }
        return intr;
}
```
返回一个 BpServiceManager 对象,这里 obj 就是前面我们创建的 BpBInder 对象。
client 获取 Service 的远程 IBinder 接口
以 CameraService 为例(camera.cpp):
```
const sp<ICameraService>& Camera::getCameraService()
{
    Mutex::Autolock _l(mLock);
    if (mCameraService.get() == 0) {
        sp<IServiceManager> sm = defaultServiceManager();
        sp<IBinder> binder;
        do {
            binder = sm->getService(String16("media.camera"));
            if (binder != 0)
                break;
            LOGW("CameraService not published, waiting...");
            usleep(500000); // 0.5 s
        } while(true);
        if (mDeathNotifier == NULL) {
            mDeathNotifier = new DeathNotifier();
        }
        binder->linkToDeath(mDeathNotifier);
        mCameraService = interface_cast<ICameraService>(binder);
    }
    LOGE_IF(mCameraService==0, "no CameraService!?");
    return mCameraService;
}
```
由前面的分析可知 sm 是 BpCameraService 对象：//应该为 BpServiceManager 对象
```
    virtual sp<IBinder> getService(const String16& name) const
    {
        unsigned n;
        for (n = 0; n < 5; n++){
            sp<IBinder> svc = checkService(name);
            if (svc != NULL) return svc;
```

```
            LOGI("Waiting for sevice %s...\n", String8(name).string());
            sleep(1);
        }
        return NULL;
    }
    virtual sp<IBinder> checkService( const String16& name) const
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        data.writeString16(name);
        remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);
        return reply.readStrongBinder();
}
```

这里的 remote 就是我们前面得到 BpBinder 对象。所以 checkService 将调用 BpBinder 中的 transact 函数：

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }
    return DEAD_OBJECT;
}
```

mHandle 为 0，BpBinder 继续往下调用 IPCThreadState:transact 函数将数据发给与 mHandle 相关联的 Service Manager Process。

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
   ..............................................................
    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
       ..............................
```

```
    return err;
}
```
通过 writeTransactionData 构造要发送的数据
```
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; //这个 handle 将传递到 service_manager
    tr.code = code;
    tr.flags = bindrFlags;
//。。。。。。。。。。。。。。。
}
```
waitForResponse 将调用 talkWithDriver 与对 Binder kernel 进行读写操作。当 Binder kernel 接收到数据后，service_mananger 线程的 ThreadPool 就会启动，service_manager 查找到 CameraService 服务后调用 binder_send_reply，将返回的数据写入 Binder kernel,Binder kernel。
```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    int32_t cmd;
    int32_t err;
      while (1) {
      if ((err=talkWithDriver()) < NO_ERROR) break;
//.....................................
}
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
  .....................................
#if defined(HAVE_ANDROID_OS)
      if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
         err = NO_ERROR;
      else
         err = -errno;
#else
      err = INVALID_OPERATION;
#endif
.....................................
}
```
通过上面的 ioctl 系统函数中 BINDER_WRITE_READ 对 binder kernel 进行读写。

Client A 与 Binder kernel 通信：kernel\drivers\android\Binder.c)
```
static int binder_open(struct inode *nodp, struct file *filp)
{
struct binder_proc *proc;
if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
  printk(KERN_INFO "binder_open: %d:%d\n", current->group_leader->pid, current->pid);
proc = kzalloc(sizeof(*proc), GFP_KERNEL);
if (proc == NULL)
                                    return -ENOMEM;
get_task_struct(current);
proc->tsk = current;  // 保存打开/dev/binder 驱动的当前进程任务数据结构
INIT_LIST_HEAD(&proc->todo);
init_waitqueue_head(&proc->wait);
proc->default_priority = task_nice(current);
mutex_lock(&binder_lock);
```

```
binder_stats.obj_created[BINDER_STAT_PROC]++;
hlist_add_head(&proc->proc_node, &binder_procs);
proc->pid = current->group_leader->pid;
INIT_LIST_HEAD(&proc->delivered_death);
filp->private_data = proc;
mutex_unlock(&binder_lock);
if (binder_proc_dir_entry_proc) {
    char strbuf[11];
    snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
  create_proc_read_entry(strbuf, S_IRUGO, binder_proc_dir_entry_proc, binder_read_proc_proc, proc); // 为当前进程创建一个 process 入口结构信息
}
return 0;
}
```

从这里可以知道每一个打开/dev/binder 的进程的信息都保存在 binder kernel 中，因而当一个进程调用 ioctl 与 kernel binder 通信时，binder kernel 就能查询到调用进程的信息。BINDER_WRITE_READ 是调用 ioctl 进程与 Binder kernel 通信一个非常重要的 command。大家可以看到在 IPCThreadState 中的 transact 函数这个函数中 call talkWithDriver 发送的 command 就是 BINDER_WRITE_READ 。

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
int ret;
struct binder_proc *proc = filp->private_data;
struct binder_thread *thread;
unsigned int size = _IOC_SIZE(cmd);
void __user *ubuf = (void __user *)arg;
/*printk(KERN_INFO "binder_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);*/
    // 将调用 ioctl 的进程挂起 caller 将挂起直到 service 返回
ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
if (ret)
  return ret;
mutex_lock(&binder_lock);
thread = binder_get_thread(proc);// 根据当 caller 进程消息获取该进程线程池数据结构
if (thread == NULL) {
    ret = -ENOMEM;
    goto err;
}
switch (cmd) {
case BINDER_WRITE_READ: { //IPcThreadState 中 talkWithDriver 设置 ioctl 的 CMD
    struct binder_write_read bwr;
    if (size != sizeof(struct binder_write_read)) {
        ret = -EINVAL;
        goto err;
    }
    if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
        ret = -EFAULT;
        goto err;
    }
    if (binder_debug_mask & BINDER_DEBUG_READ_WRITE)
        printk(KERN_INFO "binder: %d:%d write %ld at %08lx, read %ld at %08lx\n",
            proc->pid, thread->pid, bwr.write_size, bwr.write_buffer, bwr.read_size, bwr.read_buffer);
    if (bwr.write_size > 0) {
        ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer, bwr.write_size, &bwr.write_consumed);
```

```
                if (ret < 0) {
                    bwr.read_consumed = 0;
                    if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                        ret = -EFAULT;
                    goto err;
                }
            }
    if (bwr.read_size > 0) {// 数据写入到 caller process 。
        ret = binder_thread_read(proc, thread, (void __user *)bwr.read_buffer, bwr.read_size,
&bwr.read_consumed, filp->f_flags & O_NONBLOCK);
                if (!list_empty(&proc->todo))
        wake_up_interruptible(&proc->wait); // 恢复挂起的 caller 进程
                if (ret < 0) {
                    if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                        ret = -EFAULT;
                    goto err;
                }
            }
                                                    ……
}
Int binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,void __user *buffer, int size,
signed long *consumed)
{
uint32_t cmd;
void __user *ptr = buffer + *consumed;
void __user *end = buffer + size;
while (ptr < end && thread->return_error == BR_OK) {
    if (get_user(cmd, (uint32_t __user *)ptr))// 从 user 空间获取 cmd 数据到内核空间
                return -EFAULT;
            ptr += sizeof(uint32_t);
        if (_IOC_NR(cmd) < ARRAY_SIZE(binder_stats.bc)) {
                binder_stats.bc[_IOC_NR(cmd)]++;
                proc->stats.bc[_IOC_NR(cmd)]++;
                thread->stats.bc[_IOC_NR(cmd)]++;
            }
            switch (cmd) {
                case BC_INCREFS:
……
        case BC_TRANSACTION: //IPCThreadState 通过 writeTransactionData 设置该 cmd
            case BC_REPLY: {
    struct binder_transaction_data tr;
                if (copy_from_user(&tr, ptr, sizeof(tr)))
                    return -EFAULT;
                ptr += sizeof(tr);
            binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
                break;
            }
……
}
static void
binder_transaction(struct binder_proc *proc, struct binder_thread *thread,
struct binder_transaction_data *tr, int reply)
{
```

```
                                                                    .....
        if (reply) // cmd != BC_REPLY 不走这个 case
{
                                                                    .....
                                                                    }
                                                                    else
{
   if (tr->target.handle) { // 对于 service_manager 来说这个条件不满足 (handle == 0)
                                                                    .....
                                                                    }
  } else {// 这一段我们获取到了 service_mananger process 注册在 binder kernle 的进程信息
target_node = binder_context_mgr_node; //BINDER_SET_CONTEXT_MGR 注册了 service
                if (target_node == NULL) { //manager
                    return_error = BR_DEAD_REPLY;
                    goto err_no_context_mgr_node;
                                                                    }
                                                                    }
                    e->to_node = target_node->debug_id;
    target_proc = target_node->proc; // 得到目标进程 service_mananger 的结构
                if (target_proc == NULL) {
                    return_error = BR_DEAD_REPLY;
                    goto err_dead_binder;
                                                                    }
                                                                    .....
}
if (target_thread) {
                    e->to_thread = target_thread->pid;
                    target_list = &target_thread->todo;
     target_wait = &target_thread->wait; // 得到 service manager 挂起的线程
} else {
                    target_list = &target_proc->todo;
                    target_wait = &target_proc->wait;
}
.....
case BINDER_TYPE_BINDER:
case BINDER_TYPE_WEAK_BINDER: {
                                                                    .....
      ref = binder_get_ref_for_node(target_proc, node); // 在 Binder kernel 中创建
        ..... // 查找到的 service 参考
  } break;
.....
if (target_wait)
      wake_up_interruptible(target_wait); // 唤醒挂起的线程 处理 caller process 请求
.....// 处理命令可以看 svcmgr_handler
}
```

到这里我们已经通过 getService 连接到 service manager 进程了，service manager 进程得到请求后，如果他的状态是挂起的话，将被唤醒。现在我们来看一下 service manager 中的 binder_loop 函数。

Service_manager.c

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
                                                                    .....
   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); // 如果没有要处理的请求进程将挂起
                    if (res < 0) {
                        LOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
                        break;
                    }
res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);// 这里 func 就是
                    ..... //svcmgr_handler
                }
}
```

接收到数据处理的请求，这里进行解析并调用前面注册的回调函数查找 caller 请求的 service

```
int binder_parse(struct binder_state *bs, struct binder_io *bio,
                 uint32_t *ptr, uint32_t size, binder_handler func)
{
                    .....
                    switch(cmd) {
                    .....
                    case BR_TRANSACTION: {
                        struct binder_txn *txn = (void *) ptr;
                        if ((end - ptr) * sizeof(uint32_t) < sizeof(struct binder_txn)) {
                            LOGE("parse: txn too small!\n");
                            return -1;
                        }
                        binder_dump_txn(txn);
                        if (func) {
                            unsigned rdata[256/4];
                            struct binder_io msg;
                            struct binder_io reply;
            int res;
                            bio_init(&reply, rdata, sizeof(rdata), 4);
                            bio_init_from_txn(&msg, txn);
            res = func(bs, txn, &msg, &reply); // 找到 caller 请求的 service
        binder_send_reply(bs, &reply, txn->data, res);// 将找到的 service 返回给 caller
                        }
                        ptr += sizeof(*txn) / sizeof(uint32_t);
                        break;
                    .....
        }
}
void binder_send_reply(struct binder_state *bs,
                       struct binder_io *reply,
                       void *buffer_to_free,
                       int status)
{
                    struct {
                        uint32_t cmd_free;
                        void *buffer;
                        uint32_t cmd_reply;
                        struct binder_txn txn;
    } __attribute__((packed)) data;
```

```
                data.cmd_free = BC_FREE_BUFFER;
                data.buffer = buffer_to_free;
data.cmd_reply = BC_REPLY; // 将我们前面 binder_thread_write 中 cmd 替换为 BC_REPLY 就可以知
data.txn.target = 0;    // 道 service manager 如何将找到的 service 返回给 caller 了
                                                                    .....
    binder_write(bs, &data, sizeof(data)); // 调用 ioctl 与 binder kernel 通信
}
```

从这里走出去后，caller 该被唤醒了，client 进程就得到了所请求的 service 的 IBinder 对象在 Binder kernel 中的参考，这是一个远程 BBinder 对象。

连接建立后的 client 连接 Service 的通信过程：

```
virtual sp<ICamera> connect(const sp<ICameraClient>& cameraClient)
                                                                    {
                Parcel data, reply;
                    data.writeInterfaceToken(ICameraService::getInterfaceDescriptor());
                        data.writeStrongBinder(cameraClient->asBinder());
            remote()->transact(BnCameraService::CONNECT, data, &reply);
                return interface_cast<ICamera>(reply.readStrongBinder());
                                                                    }
```

向前面分析的这里 remote 是我们得到的 CameraService 的对象，caller 进程会切入到 CameraService。android 的每一个进程都会创建一个线程池，这个线程池用处理其他进程的请求。当没有数据的时候线程是挂起的，这时 binder kernel 唤醒了这个线程：

```
IPCThreadState::joinThreadPool(bool isMain)
{
  LOG_THREADPOOL("**** THREAD %p (PID %d) IS JOINING THE THREAD POOL\n", (void*)pthread_self(), getpid());
        mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);

                status_t result;
                    do {
                int32_t cmd;
            result = talkWithDriver();
            if (result >= NO_ERROR) {
        size_t IN = mIn.dataAvail(); //binder kernel 传递数据到 service
            if (IN < sizeof(int32_t)) continue;
            cmd = mIn.readInt32();
                IF_LOG_COMMANDS() {
            alog << "Processing top-level Command: "
                << getReturnString(cmd) << endl;
                                                                    }
        result = executeCommand(cmd); //service 执行 binder kernel 请求的命令
                                                                    }

        // Let this thread exit the thread pool if it is no longer
        // needed and it is not the main process thread.
            if(result == TIMED_OUT && !isMain) {
                                                                break;
                                                                    }
        } while (result != -ECONNREFUSED && result != -EBADF);
                                                                    .....
}
status_t IPCThreadState::executeCommand(int32_t cmd)
{
                BBinder* obj;
                RefBase::weakref_type* refs;
```

```
status_t result = NO_ERROR;

switch (cmd) {
.....
case BR_TRANSACTION:
{
binder_transaction_data tr;
result = mIn.read(&tr, sizeof(tr));
LOG_ASSERT(result == NO_ERROR,
"Not enough command data for brTRANSACTION");
if (result != NO_ERROR) break;

Parcel buffer;
buffer.ipcSetDataReference(
reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
tr.data_size,
reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
tr.offsets_size/sizeof(size_t), freeBuffer, this);

const pid_t origPid = mCallingPid;
const uid_t origUid = mCallingUid;

mCallingPid = tr.sender_pid;
mCallingUid = tr.sender_euid;

//LOGI(">>>> TRANSACT from pid %d uid %d\n", mCallingPid, mCallingUid);

Parcel reply;
.....
if (tr.target.ptr) {
sp<BBinder> b((BBinder*)tr.cookie); //service 中 Binder 对 象 即 CameraService
const status_t error = b->transact(tr.code, buffer, &reply, 0);// 将 调 用
if (error < NO_ERROR) reply.setError(error);//CameraService 的 onTransact 函 数

} else {
const status_t error = the_context_object->transact(tr.code, buffer, &reply, 0);
if (error < NO_ERROR) reply.setError(error);
}

//LOGI("<<<< TRANSACT from pid %d restore pid %d uid %d\n",
// mCallingPid, origPid, origUid);

if ((tr.flags & TF_ONE_WAY) == 0) {
LOG_ONEWAY("Sending reply to %d!", mCallingPid);
sendReply(reply, 0);
} else {
LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
}

mCallingPid = origPid;
mCallingUid = origUid;

IF_LOG_TRANSACTIONS() {
```

```
                                        TextOutput::Bundle _b(alog);
            alog << "BC_REPLY thr " << (void*)pthread_self() << " / obj "
              << tr.target.ptr << ": " << indent << reply << dedent << endl;
                                                                    }
                                                                .....
                                                                    }
                                                                break;
}
                                                                .....
if ((tr.flags & TF_ONE_WAY) == 0) {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
    sendReply(reply, 0); // 通过 binder kernel 返回数据到 caller 进程这个过程大家
        } else { // 参照前面的叙述自己分析一下
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
                                                                    }
                if (result != NO_ERROR) {
                                    mLastError = result;
                                                                    }
                                        return result;
}
调用 CameraService BBinder 对象中的 transact 函数：
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
                                                                .....
                                    switch (code) {
                                        case PING_TRANSACTION:
                                    reply->writeInt32(pingBinder());
                                                                break;
                                                            default:
                err = onTransact(code, data, reply, flags);
                                                                break;
                                                                    }
                                                                .....
                                        return err;
}
将调用 CameraService 的 onTransact 函数，CameraService 继承了 BBinder 。
status_t BnCameraService::onTransact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
                                        switch(code) {
                                    case CONNECT: {
                            CHECK_INTERFACE(ICameraService, data, reply);
        sp<ICameraClient> cameraClient = interface_cast<ICameraClient>(data.readStrongBinder());
        sp<ICamera> camera = connect(cameraClient); // 真正的处理函数
                                    reply->writeStrongBinder(camera->asBinder());
                                        return NO_ERROR;
                                            } break;
                                                            default:
                return BBinder::onTransact(code, data, reply, flags);
                                                                    }
}
```

至此完成了一次从 client 到 service 的通信。

设计一个多客户端的 Service

Service 可以连接不同的 Client,这里说的多客户端是指在 Service 中为不同的 client 创建不同的 IClient 接口，如果看过 AIDL 编程的话，应该清楚，Service 需要开放一个 IService 接口给客户端，我们通过 defaultServiceManager->getService 就可以得到相应的 service 一个 BpBinder 接口，通过这个接口调用 transact 函数就可以与 service 通信了，这样也就完成了一个简单的 service 与 client 程序了，但这里有个缺点就是，这个 IService 是对所有的 client 开放的，如果我们要对不同的 client 做区分的话，在建立连接的时候所有的 client 需要给 Service 一个特性，这样做也未尝不可，但会很麻烦。比如对 Camera 来说可能不止一个摄像头，摄像头的功能也不一样，这样做就比较麻烦了。其实我们完全可以参照 QT 中多客户端的设计方式，在 Service 中为每一个 Client 都创建一个 IClient 接口，IService 接口只用于 Serivce 与 Client 建立连接用。对于 Camera，如果存在多摄像头我们就可以在 Service 中为不同的 Client 打开不同的设备。

```
import android.os.IBinder;
import android.os.RemoteException;
public class TestServerServer extends android.app.testServer.ITestServer.Stub
{
int mClientCount = 0;
testServerClient mClient[];
@Override
public android.app.testServer.ITestClient.Stub connect(ITestClient client) throws RemoteException
{
// TODO Auto-generated method stub
testServerClient tClient = new testServerClient(this, client); // 为 Client 创建
mClient[mClientCount] = tClient; // 不同的 IClient
mClientCount ++;
System.out.printf("*** Server connect client is %d", client.asBinder());
return tClient;
}
@Override
public void receivedData(int count) throws RemoteException
{
// TODO Auto-generated method stub
}
Public static class testServerClient extends android.app.testServer.ITestClient.Stub{
public android.app.testServer.ITestClient mClient;
public TestServerServer mServer;
public testServerClient(TestServerServer tServer, android.app.testServer.ITestClient tClient) {
mServer = tServer;
mClient = tClient;
}
public IBinder asBinder() {
// TODO Auto-generated method stub
return this;
}
}
}
```

这仅仅是个 Service 的 demo 而已，如果添加这个作为 system Service 还得改一下 android 代码 avoid permission check!

总结：

假定一个 Client A 进程与 Service B 进程要建立 IPC 通信，通过前面的分析我们知道他的流程如下：

1：Service B 打开 Binder driver， 将自己的进程信息注册到 kernel 并为 Service 创建一个 binder_ref。

2：Service B 通过 Add_Service 将 Service 信息添加到 service_manager 进程

3：Service B 的 Thread pool 挂起等待 client 的请求

4：Client A 调用 open_driver 打开 Binder driver 将自己的进程信息注册到 kernel 并为 Service 创建一个

binder_ref
5: Client A 调用 defaultManagerService.getService 得到 Service B 在 kernel 中的 IBinder 对象
6：通过 transact 与 Binder kernel 通信，Binder Kernel 将 Client A 挂起。
7：Binder Kernel 恢复 Service B thread pool 线程，并在 joinThreadPool 中处理 Client 的请求
8: Binder Kernel 挂起 Service B 并将 Service B 返回的数据写到 Client A
9：Binder Kernle 恢复 Client A
Binder kernel driver 在 Client A 与 Service B 之间扮演着中间代理的角色。任何通过 transact 传递的 IBinder 对象都会在 Binder kernel 中创建一个与此相关联的独一无二的 BInder 对象，用于区分不同的 Client。

Android 的 IPC 机制 Binder 的各个部分

第一部分 Binder 的组成

1.1 驱动程序部分驱动程序的部分在以下的文件夹中：

kernel/include/linux/binder.h

kernel/drivers/android/binder.c

binder 驱动程序是一个 miscdevice，主设备号为 10，此设备号使用动态获得（MISC_DYNAMIC_MINOR），其设备的节点为：

/dev/binder

binder 驱动程序会在 proc 文件系统中建立自己的信息，其文件夹为/proc/binde，其中包含如下内容：

proc 目录：调用 Binder 各个进程的内容

state 文件：使用函数 binder_read_proc_state

stats 文件：使用函数 binder_read_proc_stats

transactions 文件：使用函数 binder_read_proc_transactions

transaction_log 文件：使用函数 binder_read_proc_transaction_log，其参数为 binder_transaction_log（类型为 struct binder_transaction_log）

failed_transaction_log 文件：使用函数 binder_read_proc_transaction_log 其参数为 binder_transaction_log_failed（类型为 struct binder_transaction_log）

在 binder 文件被打开后，其私有数据（private_data）的类型：

struct binder_proc

在这个数据结构中，主要包含了当前进程、进程 ID、内存映射信息、Binder 的统计信息和线程信息等。

在用户空间对 Binder 驱动程序进行控制主要使用的接口是 mmap、poll 和 ioctl，ioctl 主要使用的 ID 为：

```
#define BINDER_WRITE_READ        _IOWR('b', 1, struct binder_write_read)
#define BINDER_SET_IDLE_TIMEOUT   _IOW('b', 3, int64_t)
#define BINDER_SET_MAX_THREADS _IOW('b', 5, size_t)
#define BINDER_SET_IDLE_PRIORITY _IOW('b', 6, int)
#define BINDER_SET_CONTEXT_MGR _IOW('b', 7, int)
#define BINDER_THREAD_EXIT    _IOW('b', 8, int)
#define BINDER_VERSION           _IOWR('b', 9, struct binder_version)
```

BR_XXX 等宏为 BinderDriverReturnProtocol，表示 Binder 驱动返回协议。

BC_XXX 等宏为 BinderDriverCommandProtocol，表示 Binder 驱动命令协议。

binder_thread 是 Binder 驱动程序中使用的另外一个重要的数据结构，数据结构的定义如下所示：

```
struct binder_thread {
  struct binder_proc *proc;
   struct rb_node rb_node;
   int pid;
   int looper;
   struct binder_transaction *transaction_stack;
   struct list_head todo;
   uint32_t return_error;
   uint32_t return_error2;
   wait_queue_head_t wait;
   struct binder_stats stats;
};
```

binder_thread 的各个成员信息是从 rb_node 中得出。

BINDER_WRITE_READ 是最重要的 ioctl，它使用一个数据结构 binder_write_read 定义读写的数据。

```
struct binder_write_read {
   signed long write_size;
   signed long write_consumed;
   unsigned long write_buffer;
   signed long read_size;
   signed long read_consumed;
   unsigned long read_buffer;
};
```

1.2 servicemanager 部分

servicemanager 是一个守护进程，用于这个进程的和/dev/binder 通讯，从而达到管理系统中各个服务的作用。

可执行程序的路径：

/system/bin/servicemanager

开源版本文件的路径：

frameworks/base/cmds/servicemanager/binder.h
frameworks/base/cmds/servicemanager/binder.c
frameworks/base/cmds/servicemanager/service_manager.c

程序执行的流程：

open()：打开 binder 驱动
mmap()：映射一个 128*1024 字节的内存
ioctl(BINDER_SET_CONTEXT_MGR)：设置上下文为 mgr
进入主循环 binder_loop()
ioctl(BINDER_WRITE_READ)，读取
binder_parse()进入 binder 处理过程循环处理
binder_parse()的处理，调用返回值：

当处理 BR_TRANSACTION 的时候，调用 svcmgr_handler()处理增加服务、检查服务等工作。各种服务存放在一个链表（svclist）中。其中调用 binder_等开头的函数，又会调用 ioctl 的各种命令。

处理 BR_REPLY 的时候，填充 binder_io 类型的数据结

1.3 binder 的库的部分

binder 相关的文件作为 Android 的 uitls 库的一部分，这个库编译后的名称为 libutils.so，是 Android 系统中的一个公共库。

主要文件的路径如下所示：

frameworks/base/include/utils/*
frameworks/base/libs/utils/*

主要的类为：

RefBase.h :引用计数，定义类 RefBase。
Parcel.h :为在 IPC 中传输的数据定义容器，定义类 **Parcel**
IBinder.h：Binder 对象的抽象接口， 定义类 IBinder
Binder.h：Binder 对象的基本功能， 定义类 Binder 和 BpRefBase

BpBinder.h：BpBinder 的功能，定义类 BpBinder
IInterface.h：为抽象经过 Binder 的接口定义通用类，定义类 IInterface，类模板 BnInterface，类模板 BpInterface
ProcessState.h 表示进程状态的类，定义类 ProcessState
IPCThreadState.h 表示 IPC 线程的状态，定义类 **IPCThreadState**

在 IInterface.h 中定义的 BnInterface 和 BpInterface 是两个重要的模版，这是为各种程序中使用的。
BnInterface 模版的定义如下所示：

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

BnInterface 模版的定义如下所示：

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
                        BpInterface(const sp<IBinder>& remote);
protected:
virtual IBinder* onAsBinder();
};
```

这两个模版在使用的时候，起到得作用实际上都是双继承：使用者定义一个接口 INTERFACE，然后使用 BnInterface 和 BpInterface 两个模版结合自己的接口，构建自己的 BnXXX 和 BpXXX 两个类。
DECLARE_META_INTERFACE 和 IMPLEMENT_META_INTERFACE 两个宏用于帮助 BpXXX 类的实现：

```
#define DECLARE_META_INTERFACE(INTERFACE)                               \
static const String16 descriptor;                                       \
static sp<I##INTERFACE> asInterface(const sp<IBinder>& obj);            \
virtual String16 getInterfaceDescriptor() const;                        \
#define IMPLEMENT_META_INTERFACE(INTERFACE, NAME)                           \
const String16 I##INTERFACE::descriptor(NAME);                      \
String16 I##INTERFACE::getInterfaceDescriptor() const {             \
       return I##INTERFACE::descriptor;                                 \
}                                                                   \
sp<I##INTERFACE> I##INTERFACE::asInterface(const sp<IBinder>& obj)    \
{                                                                   \
       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;                                                 \
}
```

在定义自己的类的时候，只需要使用 DECLARE_META_INTERFACE 和 IMPLEMENT_META_INTERFACE 两个接口，并结合类的名称，就可以实现 BpInterface 中的 asInterface()和 getInterfaceDescriptor()两个函数。

第二部分 Binder 的运作

2.1 Binder 的工作机制

Service Manager 是一个守护进程，它复杂启动各个进程之间的服务，对于相关的两个需要通讯的进程，它们通过调用 libutil.so 库实现通讯，而真正通讯的截止，是内核空间中的一块共享内存。

2.2 从应用程序的角度看 Binder

从应用程序的角度看 Binder 一共有三个方面：

- Native 本地：例如 BnABC，这是一个需要被继承和实现的类。

  Proxy 代理：例如 BpABC，这是一个在接口框架中被实现，但是在接口中没有体现的类。

  客户端：例如客户端得到一个接口 ABC，在调用的时候实际上被调用的是 BpABC

- 本地功能（Bn）部分做的：

  实现 **BnABC:: BnTransact()**
  注册服务：IServiceManager：：AddService
  代理部分（Bp）做的：
  实现几个功能函数，调用 **BpABC::remote()->transact()**

  客户端做的：
  获得 ABC 接口，然后调用接口（实际上调用了 **BpABC**，继而通过 **IPC** 调用了 **BnABC**，然后调用了具体的功能）

  在程序的实现过程中 **BnABC** 和 **BpABC** 是双继承了接口 **ABC**。一般来说 **BpABC** 是一个实现类，这个实现类不需要在接口中体现，它实际上负责的只是通讯功能，不执行具体的功能；**BnABC** 则是一个接口类，需要一个真正工作的类来继承、实现它，这个类才是真正执行具体功能的类。

  在客户端中，从 **ISeriviceManager** 中获得一个 **ABC** 的接口，客户端调用这个接口，实际上是在调用 **BpABC**，而 **BpABC** 又通过 **Binder** 的 **IPC** 机制和 **BnABC** 通讯，**BnABC** 的实现类在后面执行。

  事实上，服务器的具体实现和客户端是两个不同的进程，如果不考虑进程间通讯的过程，从调用者的角度，似乎客户端在直接调用另外一个进程间的函数——当然这个函数必须是接口 **ABC** 中定义的。

2.3 ISericeManager 的作用

ISericeManager 涉及的两个文件是 ISericeManager.h 和 ISericeManager.cpp。这两个文件基本上是 ISericeManager。ISericeManager 是系统最先被启动的服务。非常值得注意的是：ISericeManager 本地功能并没有使现，它实际上由 ServiceManager 守护进程执行，而用户程序通过调用 BpServiceManager 来获得其他的服务。

在 ISericeManager.h 中定义了一个接口，用于得到默认的 ISericeManager：

```
sp<IServiceManager> defaultServiceManager();
```

这时得到的 ISericeManager 实际上是一个全局的 ISericeManager。

3.1 一个利用接口的具体实现

PermissionController 也是 libutils 中定义的一个有关权限控制的接口，它一共包含两个文件：IPermissionController.h 和 IPermissionController.cpp 这个结构在所有类的实现中都是类似的。

头文件 IPermissionController.h 的主要内容是定义 IPermissionController 接口和类 BnPermissionController：

```
class IPermissionController : public IInterface
{
public:
DECLARE_META_INTERFACE(PermissionController);
virtual bool checkPermission(const String16& permission,int32_t pid, int32_t uid) = 0;
enum {
        CHECK_PERMISSION_TRANSACTION = IBinder::FIRST_CALL_TRANSACTION
};
};
class BnPermissionController : public BnInterface<IPermissionController>
{
public:
virtual status_t onTransact( uint32_t code,
                             const Parcel& data,
                             Parcel* reply,
                             uint32_t flags = 0);
};
```

IPermissionController 是一个接口类，只有 checkPermission() 一个纯虚函数。BnPermissionController 继承了以 BnPermissionController 实例化模版类 BnInterface。因此，BnPermissionController，事实上 BnPermissionController 双继承了 BBinder 和 IPermissionController。

实现文件 IPermissionController.cpp 中，首先实现了一个 BpPermissionController。

```
class BpPermissionController : public BpInterface<IPermissionController>
{
public:
BpPermissionController(const sp<IBinder>& impl)
                       : BpInterface<IPermissionController>(impl)
{
}
```

```
virtual bool checkPermission(const String16& permission, int32_t pid, int32_t uid)
{
    Parcel data, reply;
    data.writeInterfaceToken(IPermissionController::
        getInterfaceDescriptor());
    data.writeString16(permission);
    data.writeInt32(pid);
    data.writeInt32(uid);
    remote()->transact(CHECK_PERMISSION_TRANSACTION, data, &reply);
    if (reply.readInt32() != 0) return 0;
    return reply.readInt32() != 0;
}
};
IMPLEMENT_META_INTERFACE(PermissionController, "android.os.IPermissionController");
```

BpPermissionController 继承了 BpInterface<IPermissionController>，它本身是一个已经实现的类，而且并没有在接口中体现。这个类按照格式写就可以，在实现 checkPermission()函数的过程中，使用 Parcel 作为传输数据的容器，传输中时候 transact()函数，其参数需要包含枚举值 CHECK_PERMISSION_TRANSACTION。IMPLEMENT_META_INTERFACE 用于扶助生成 。
BnPermissionController 中实现的 onTransact() 函数如下所示：

```
status_t BnPermissionController::BnTransact(
uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
switch(code) {
    case CHECK_PERMISSION_TRANSACTION: {
        CHECK_INTERFACE(IPermissionController, data, reply);
        String16 permission = data.readString16();
        int32_t pid = data.readInt32();
        int32_t uid = data.readInt32();
        bool res = checkPermission(permission, pid, uid);
        reply->writeInt32(0);
        reply->writeInt32(res ? 1 : 0);
        return NO_ERROR;
    } break;
    default:
        return BBinder::BnTransact(code, data, reply, flags);
}
}
```

在 onTransact()函数中根据枚举值判断数据使用的方式。注意，由于 BnPermissionController 也是继承了类 IPermissionController，但是纯虚函数 checkPermission()依然没有实现。因此这个 BnPermissionController 类并不能实例化，它其实也还是一个接口，需要一个实现类来继承它，那才是实现具体功能的类。

3.2 BnABC 的实现

本地服务启动后将形成一个守护进程，具体的本地服务是由一个实现类继承 BnABC 来实现的，这个服务的名称通常叫做 ABC。
在其中，通常包含了一个 instantiate()函数，这个函数一般按照如下的方式实现：

```
void ABC::instantiate() {
defaultServiceManager()->addService(
    String16("XXX.ABC"), new ABC());
}
```

按照这种方式，通过调用 defaultServiceManager()函数，将增加一个名为"XXX.ABC"的服务。
在这个 defaultServiceManager()函数中调用了：

```
ProcessState::self()->getContextObject(NULL));
IPCThreadState* ipc = IPCThreadState::self();
IPCThreadState::talkWithDriver()
```

在 ProcessState 类建立的过程中调用 open_driver()打开驱动程序，在 talkWithDriver()的执行过程中。

3.3 BpABC 调用的实现

BpABC 调用的过程主要通过 mRemote()->transact() 来传输数据，mRemote()是 BpRefBase 的成员，它是一个 IBinder。这个调用过程如下所示：

```
mRemote()->transact()
Process::self()
IPCThreadState::self()->transact()
writeTransactionData()
waitForResponse()
talkWithDriver()
ioctl(fd, BINDER_WRITE_READ, &bwr)
```

在 IPCThreadState::executeCommand()函数中，实现传输操作。

android IPC 通信机制中 BBinder 与 BpBinder 的区别

刚开始看 android 的 IPC 通信机制，BBinder 与 BpBinder 这两者容易混淆。其实这两者是很好区分，对于 service 来说继承了 BBinder（BnInterface）因为 BBinder 有 onTransact 消息处理函数，而对于与 service 通信的 client 来说需要继承 BpBinder(BpInterface)，因为 BpBinder 有消息传递函数 transcat。

以 cameraService 的 client 为例

Camera.cpp 中 getCameraService 函数取得远程 CameraService 的 IBinder 对象，然后通过 mCameraService = interface_cast<ICameraService>(binder);

进行重构得到了 BpCameraService 对象。而 BpCameraService 继承了 BpInterface。

cameraService：

defaultServiceManager()->addService(

String16("media.camera"), new CameraService()); 传入了 BBinder。

IPC 传递的过程中 IBinder 指针不可缺少，这个指针对一个进程来说就像是 socket 的 ID 一样，唯一的。所以不管这个 IBinder 是 BBinder 还是 BpBinder, 他们的都是在重构 BpBinder 或者 BBinder 的时候把 IBinder 作为参数传入。

--------Albert

2. How binder.java is connected with binder.c?

Through JNI calls to the Binder C++ classes which use IPCThreadState to interact with the driver.

Albertchen Android IPC 通讯机制源码分析

http://hi.baidu.com/albertchen521/blog/item/30c32d3f4bee993a71cf6ca0.html

http://hi.baidu.com/albertchen521/blog/item/822058d0f63ea2d4562c84a1.html

Android 的 IPC 机制 Binder 的各个部分

http://hi.baidu.com/android_fans/blog/item/8dd4e430882951af5fdf0e27.html

http://hi.baidu.com/android_fans/blog/item/7628aad2376674d8a8ec9a20.html

http://hi.baidu.com/android_fans/blog/item/c3db3858af67b12d2934f020.html