修改记录

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 日期 | 作者 | 说明 |
| V0.01 | 10年9月7日 | 徐申龙 | 初始版本。 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

C++ 类结构 1

类结构解析 2

IBinder与接口之间的转换 3

Log 5

Android IPC 通讯机制源码分析 10

Android的IPC机制Binder的各个部分 12

类结构解析 2

IBinder与接口之间的转换 3

Log 5

Android IPC 通讯机制源码分析 10

Android的IPC机制Binder的各个部分 12

1. 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

1. 类结构解析
2. IBinder与接口之间的转换
3. Log
4. 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。

1. 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。

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>