Binder系列4—注册服务(addService)

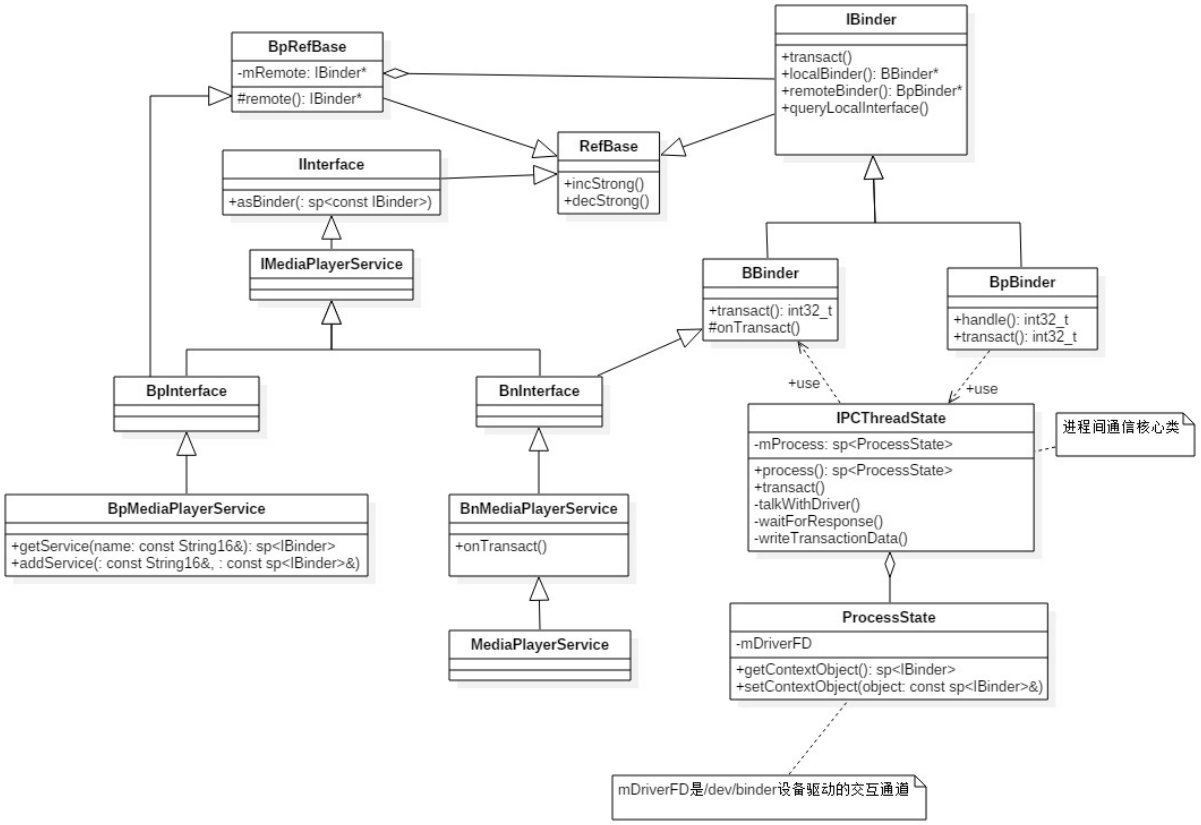
Nov 14, 2015

- 类关系图
- 源码分析
  - 源码入口
- 源码分析
  - [1] instantiate()
  - [3] addService
  - [4] BpBinder::transact
  - [5] IPCThreadState::self()
  - [6] new IPCThreadState
  - [7] transact
  - [8] writeTransactionData
  - [9] waitForResponse
  - [10] talkWithDriver
  - [11] executeCommand
  - [12]. BBinder::transact
  - [13]. BBinder::onTransact
  - [16] startThreadPool
  - [17] spawnPooledThread
  - [20] joinThreadPool()
  - [21]. getAndExecuteCommand

> 基于Android 6.0的源码剖析， 本文讲解如何向ServiceManager注册服务的过程。

类关系图

在Native层中，我们以media为例，来展开讲解，先来看看media的类关系图。



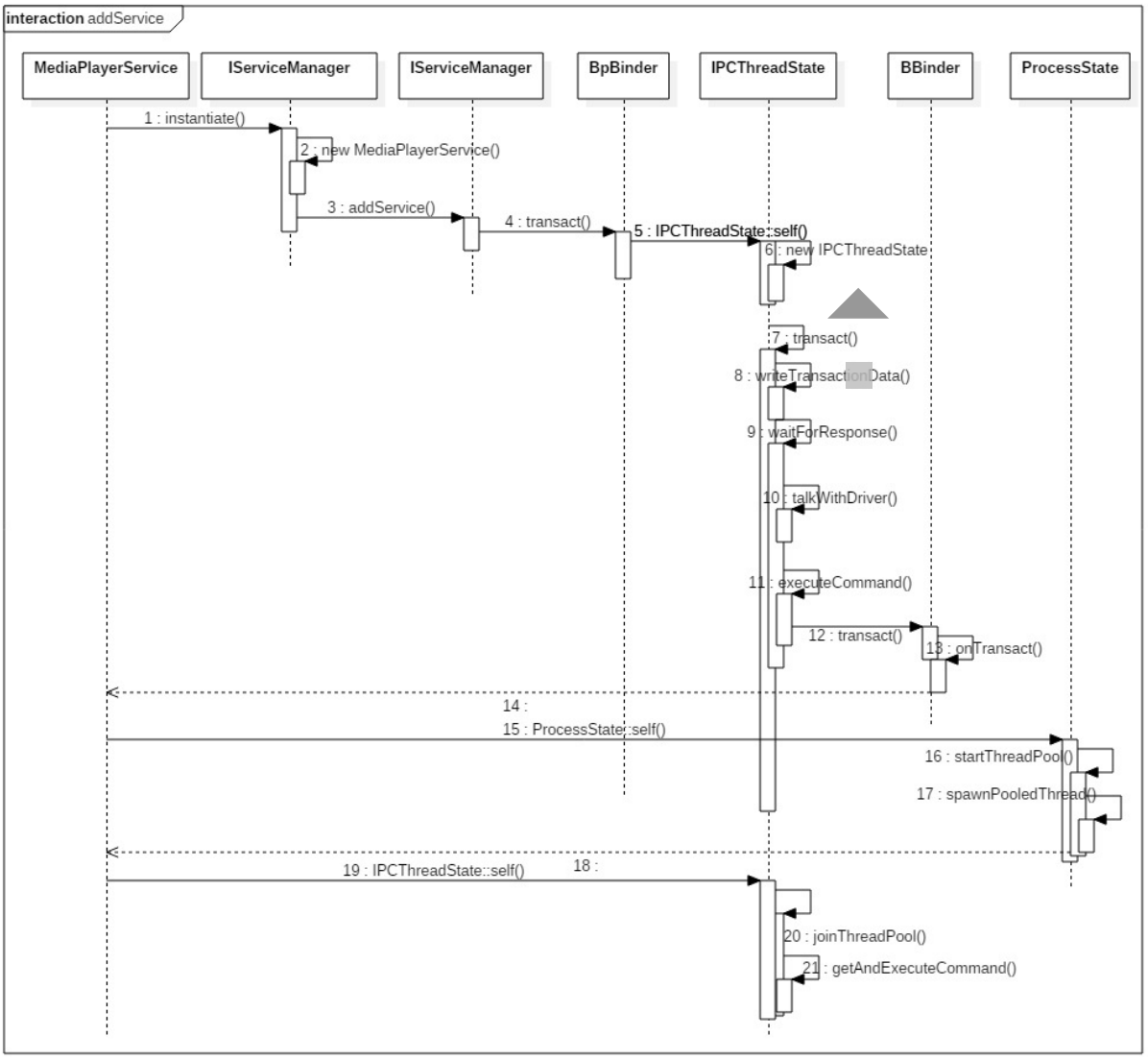
源码分析

相关源码

```
/framework/native/libs/binder/IServiceManager.cpp
/framework/native/libs/binder/BpBinder.cpp
/framework/native/libs/binder/IPCThreadState.cpp
/framework/native/libs/binder/Binder.cpp
/framework/native/libs/binder/ProcessState.cpp

/framework/av/media/libmediaplayerservice/MediaPlayerService.cpp
```

流程图



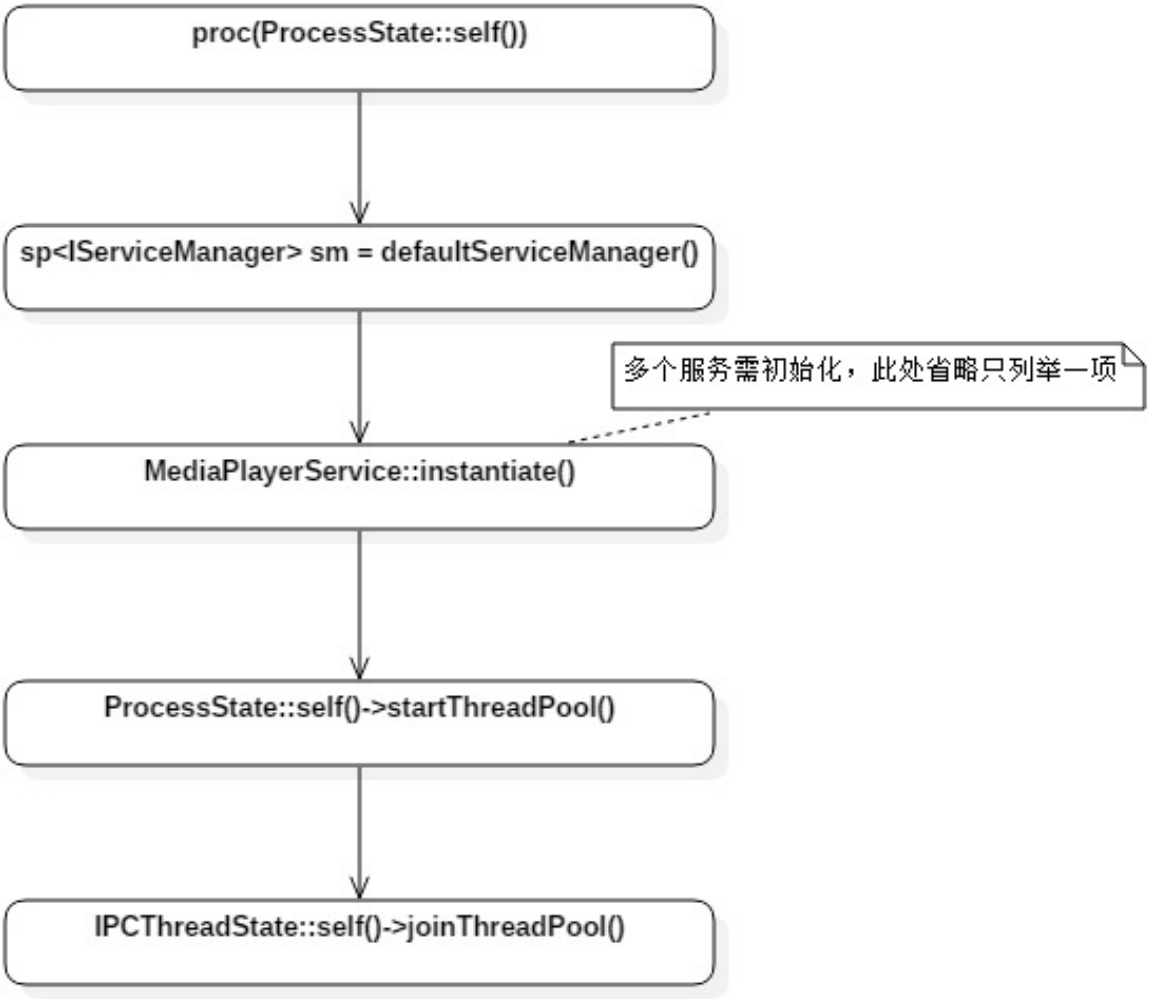
下面开始讲解每一个流程：

源码入口

main_mediaserver.cpp是可执行程序，入口函数main代码如下：

```
int main(int argc __unused, char** argv)
{
        ...
    InitializeIcuOrDie();  //初始化ICU，国际通用编码方案。
    sp<ProcessState> proc(ProcessState::self());       //获得ProcessState实例
    sp<IServiceManager> sm = defaultServiceManager(); //获取ServiceManager实例
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();                //多媒体服务   【见流程1~12】
    ResourceManagerService::instantiate();
    CameraService::instantiate();
    AudioPolicyService::instantiate();
    SoundTriggerHwService::instantiate();
    RadioService::instantiate();
    registerExtensions();
    ProcessState::self()->startThreadPool();        //创建线程池    【见流程16】
    IPCThreadState::self()->joinThreadPool();       //当前线程加入到线程池 【见流程20】
 }
```

这个过程主要分下面5个步骤：



上面的main方法，对于 `defaultServiceManager()` ，在前一篇文章Binder系列3——— 获取Service Manager (http://www.yuanhh.com/2015/11/08/binder-get-sm/)已经介绍，下面主要讲，后三步如下图：

源码分析

[1] instantiate()

`==> /framework/av/media/libmediaplayerservice/MediaPlayerService.cpp`

注册服务MediaPlayerService

```
void MediaPlayerService::instantiate() {
    defaultServiceManager()->addService(
            String16("media.player"), new MediaPlayerService()); 【见流程3】
}
```

由Binder系列3 —— 获取Service Manager (http://www.yuanhh.com/2015/11/08/binder-get-sm/)分析，可知defaultServiceManager()返回的是BpServiceManager。故此处等价于调用BpServiceManager->addService。
关于MediaPlayerService的初始化过程，此处就省略，后面有时间会单独介绍。

[3] addService

==> /framework/native/libs/binder/IServiceManager.cpp

服务注册

```
virtual status_t addService(const String16& name, const sp<IBinder>& service,
        bool allowIsolated)
{
    Parcel data, reply; //Parcel是数据通信包
    data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());   //写入RPC头信息
    data.writeString16(name);        // name为 "media.player"
    data.writeStrongBinder(service); // MediaPlayerService对象
    data.writeInt32(allowIsolated ? 1 : 0); // allowIsolated= false
    status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply); //【见流程4】
    return err == NO_ERROR ? reply.readExceptionCode() : err;
}
```

- 将名为”media.player”的MediaPlayerService服务注册到ServiceManager；
- RPC头信息 IServiceManager::getInterfaceDescriptor()为“android.os.IServiceManager”；
- remote()就是BpBinder()；

[4] BpBinder::transact

==> /framework/native/libs/binder/BpBinder.cpp

Binder代理类调用transact

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

真正工作交给IPCThreadState来进行transact工作，由【流程3】传递过来的参数：transact(ADD_SERVICE_TRANSACTION, data, &reply, 0);

[5] IPCThreadState::self()

==> /framework/native/libs/binder/IPCThreadState.cpp

获取IPCThreadState对象

```
IPCThreadState* IPCThreadState::self()
{
    if (gHaveTLS) {
restart:
        const pthread_key_t k = gTLS;
        IPCThreadState* st = (IPCThreadState*)pthread_getspecific(k);
        if (st) return st;
        return new IPCThreadState;  //初始IPCThreadState 【见流程6】
    }

    if (gShutdown) return NULL;

    pthread_mutex_lock(&gTLSMutex);
    if (!gHaveTLS) { //首次进入gHaveTLS为false
        if (pthread_key_create(&gTLS, threadDestructor) != 0) { //创建线程的TLS
            pthread_mutex_unlock(&gTLSMutex);
            return NULL;
        }
        gHaveTLS = true;
    }
    pthread_mutex_unlock(&gTLSMutex);
    goto restart;
}
```

TLS是指Thread local storage(线程本地储存空间)，每个线程都拥有自己的TLS，并且是私有空间，线程之间不会共享。通过pthread_getspecific/pthread_setspecific函数可以获取/设置这些空间中的内容。从线程本地存储空间中获得保存在其中的IPCThreadState对象。

[6] new IPCThreadState

`==> /framework/native/libs/binder/IPCThreadState.cpp`

创建IPCThreadState对象

```
IPCThreadState::IPCThreadState()
    : mProcess(ProcessState::self()),
      mMyThreadId(gettid()),
      mStrictModePolicy(0),
      mLastTransactionBinderFlags(0)
{
    pthread_setspecific(gTLS, this);
    clearCaller();
    mIn.setDataCapacity(256);
    mOut.setDataCapacity(256);
}
```

每个线程都有一个 `IPCThreadState` ，每个 `IPCThreadState` 中都有一个mIn、一个mOut。成员变量mProcess保存了ProcessState变量(每个进程只有一个)。

- mIn 用来接收来自Binder设备的数据，默认大小为256字节；
- mOut用来存储发往Binder设备的数据，默认大小为256字节。

[7] transact

`==> /framework/native/libs/binder/IPCThreadState.cpp`

IPCThreadState进行transact事务处理

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err = data.errorCheck(); //数据错误检查
    flags |= TF_ACCEPT_FDS;
    ....
    if (err == NO_ERROR) {
        err = writeTransactionData(BC_TRANSACTION, flags, handle, cod
e, data, NULL); // 传输数据
    }

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

    if ((flags & TF_ONE_WAY) == 0) { //flgs =0进入该分支
        if (reply) {
            err = waitForResponse(reply); //等待响应
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }

    } else {
        err = waitForResponse(NULL, NULL); //不需要响应消息的binder
    }

    return err;
}
```

工作分3部分：

- errorCheck() //数据错误检查
- writeTransactionData() // 传输数据
- waitForResponse() //f等待响应

由【流程4】传递过来的参数：transact (0，ADD_SERVICE_TRANSACTION, data, &reply, 0);

[8] writeTransactionData

==> /framework/native/libs/binder/IPCThreadState.cpp

将transaction数据写入到mOut

```
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t bi
nderFlags,
    int32_t handle, uint32_t code, const Parcel& data, status_t* statu
sBuffer)
{
    binder_transaction_data tr;

    tr.target.ptr = 0;
    tr.target.handle = handle; // handle=0
    tr.code = code;            // ADD_SERVICE_TRANSACTION
    tr.flags = binderFlags;    // 0
    tr.cookie = 0;
    tr.sender_pid = 0;
    tr.sender_euid = 0;

    const status_t err = data.errorCheck();
    if (err == NO_ERROR) {
        tr.data_size = data.ipcDataSize();  // data为Media服务相关的parc
el通信数据包
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()*sizeof(binder_siz
e_t);
        tr.data.ptr.offsets = data.ipcObjects();
    } else if (statusBuffer) {
        tr.flags |= TF_STATUS_CODE;
        *statusBuffer = err;
        tr.data_size = sizeof(status_t);
        tr.data.ptr.buffer = reinterpret_cast<uintptr_t>(statusBuffe
r);
        tr.offsets_size = 0;
        tr.data.ptr.offsets = 0;
    } else {
        return (mLastError = err);
    }

    mOut.writeInt32(cmd);          //cmd = BC_TRANSACTION
    mOut.write(&tr, sizeof(tr));  //写入binder_transaction_data数据

    return NO_ERROR;
}
```

由【流程7】传递过来的参数：writeTransactionData(BC_TRANSACTION, 0, 0, ADD_SERVICE_TRANSACTION, data, NULL)。
handle的值用来标识目的端，其中0是ServiceManager的标志。
`binder_transaction_data` 是和binder设备通信的数据结构，最终是把所有相关信息写到 `mOut` 。

[9] waitForResponse

==> /framework/native/libs/binder/IPCThreadState.cpp

不断循环地与Binder驱动设备交互，获取响应信息

【流程8】传递过来的参数：waitForResponse(&reply, NULL);

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acqu
ireResult)
{
    int32_t cmd;
    int32_t err;

    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break; // 【见流程10】
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;

        cmd = mIn.readInt32();

        switch (cmd) {
        case BR_TRANSACTION_COMPLETE:
            if (!reply && !acquireResult) goto finish;
            break;

        case BR_DEAD_REPLY:
            err = DEAD_OBJECT;
            goto finish;

        case BR_FAILED_REPLY:
            err = FAILED_TRANSACTION;
            goto finish;

        case BR_ACQUIRE_RESULT:
            {
                const int32_t result = mIn.readInt32();
                if (!acquireResult) continue;
                *acquireResult = result ? NO_ERROR : INVALID_OPERATIO
N;
            }
            goto finish;

        case BR_REPLY:
            {
                binder_transaction_data tr;
                err = mIn.read(&tr, sizeof(tr));
                if (err != NO_ERROR) goto finish;

                if (reply) {
                    if ((tr.flags & TF_STATUS_CODE) == 0) {
                        reply->ipcSetDataReference(
                            reinterpret_cast<const uint8_t*>(tr.data.p
tr.buffer),
                            tr.data_size,
```

```
                            reinterpret_cast<const binder_size_t*>(t
r.data.ptr.offsets),
                            tr.offsets_size/sizeof(binder_size_t),
                            freeBuffer, this);
                    } else {
                        err = *reinterpret_cast<const status_t*>(tr.da
ta.ptr.buffer);
                        freeBuffer(NULL,
                            reinterpret_cast<const uint8_t*>(tr.data.p
tr.buffer),
                            tr.data_size,
                            reinterpret_cast<const binder_size_t*>(t
r.data.ptr.offsets),
                            tr.offsets_size/sizeof(binder_size_t), thi
s);
                    }
                } else {
                    freeBuffer(NULL,
                        reinterpret_cast<const uint8_t*>(tr.data.ptr.b
uffer),
                        tr.data_size,
                        reinterpret_cast<const binder_size_t*>(tr.dat
a.ptr.offsets),
                        tr.offsets_size/sizeof(binder_size_t), this);
                    continue;
                }
            }
            goto finish;

        default:
            err = executeCommand(cmd);  //【见流程11】
            if (err != NO_ERROR) goto finish;
            break;
        }
    }

finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
    }

    return err;
}
```

[10] talkWithDriver

```
==> /framework/native/libs/binder/IPCThreadState.cpp
```

与Binder驱动交互，是真正往Binder设备写数据，与读取Binder设备数据的过程。

```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    if (mProcess->mDriverFD <= 0) {
        return -EBADF;
    }

    binder_write_read bwr;

    const bool needRead = mIn.dataPosition() >= mIn.dataSize();
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize()
: 0;

    bwr.write_size = outAvail;
    bwr.write_buffer = (uintptr_t)mOut.data();

    if (doReceive && needRead) {
                //接收数据缓冲区信息的填充。如果以后收到数据，就直接填在mIn中
了。
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (uintptr_t)mIn.data();
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }

    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERRO
R;

    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    do {
#if defined(HAVE_ANDROID_OS)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
//ioctl不停的读写操作
            err = NO_ERROR;
        else
            err = -errno;
#else
        err = INVALID_OPERATION;
#endif
        if (mProcess->mDriverFD <= 0) {
            err = -EBADF;
        }

    } while (err == -EINTR);

    if (err >= NO_ERROR) {
```

```
        if (bwr.write_consumed > 0) {
            if (bwr.write_consumed < mOut.dataSize())
                mOut.remove(0, bwr.write_consumed);
            else
                mOut.setDataSize(0);
        }
        if (bwr.read_consumed > 0) {
            mIn.setDataSize(bwr.read_consumed);
            mIn.setDataPosition(0);
        }
        return NO_ERROR;
    }
    return err;
}
```

`binder_write_read` 是用来与Binder设备交换数据的结构, 通过ioctl与mDriverFD通信，是真正与Binder驱动进行数据读写交互的过程。

[11] executeCommand

`==> /framework/native/libs/binder/IPCThreadState.cpp`

根据收到的响应消息，执行相应的操作

【流程9】传递过来的参数：executeCommand(BR_TRANSACTION)

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

    switch (cmd) {
    case BR_ERROR:
        result = mIn.readInt32();
        break;

    case BR_OK:
        break;

    case BR_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readPointer();
        obj = (BBinder*)mIn.readPointer();
        obj->incStrong(mProcess.get());
        mOut.writeInt32(BC_ACQUIRE_DONE);
        mOut.writePointer((uintptr_t)refs);
        mOut.writePointer((uintptr_t)obj);
        break;

    case BR_RELEASE:
        refs = (RefBase::weakref_type*)mIn.readPointer();
        obj = (BBinder*)mIn.readPointer();
        mPendingStrongDerefs.push(obj);
        break;

    case BR_INCREFS:
        refs = (RefBase::weakref_type*)mIn.readPointer();
        obj = (BBinder*)mIn.readPointer();
        refs->incWeak(mProcess.get());
        mOut.writeInt32(BC_INCREFS_DONE);
        mOut.writePointer((uintptr_t)refs);
        mOut.writePointer((uintptr_t)obj);
        break;

    case BR_DECREFS:
        refs = (RefBase::weakref_type*)mIn.readPointer();
        obj = (BBinder*)mIn.readPointer();
        mPendingWeakDerefs.push(refs);
        break;

    case BR_ATTEMPT_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readPointer();
        obj = (BBinder*)mIn.readPointer();
        const bool success = refs->attemptIncStrong(mProcess.get());
```

```
        mOut.writeInt32(BC_ACQUIRE_RESULT);
        mOut.writeInt32((int32_t)success);
        break;

    case BR_TRANSACTION:
        {
            binder_transaction_data tr;
            result = mIn.read(&tr, sizeof(tr));
            if (result != NO_ERROR) break;

            Parcel buffer;
            buffer.ipcSetDataReference(
                reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                tr.data_size,
                reinterpret_cast<const binder_size_t*>(tr.data.ptr.off
sets),
                tr.offsets_size/sizeof(binder_size_t), freeBuffer, thi
s);

            const pid_t origPid = mCallingPid;
            const uid_t origUid = mCallingUid;
            const int32_t origStrictModePolicy = mStrictModePolicy;
            const int32_t origTransactionBinderFlags = mLastTransactio
nBinderFlags;

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

            int curPrio = getpriority(PRIO_PROCESS, mMyThreadId);
            if (gDisableBackgroundScheduling) {
                if (curPrio > ANDROID_PRIORITY_NORMAL) {
                    setpriority(PRIO_PROCESS, mMyThreadId, ANDROID_PRI
ORITY_NORMAL);
                }
            } else {
                if (curPrio >= ANDROID_PRIORITY_BACKGROUND) {
                    set_sched_policy(mMyThreadId, SP_BACKGROUND);
                }
            }

            Parcel reply;
            status_t error;
                        // tr.cookie里存放的是BBinder，此处b是BBinder的实
现子类
            if (tr.target.ptr) {
                sp<BBinder> b((BBinder*)tr.cookie);
                    error = b->transact(tr.code, buffer, &reply, tr.fl
ags); //【见流程12】
```

```
            } else {
                error = the_context_object->transact(tr.code, buffer,
&reply, tr.flags);
            }

            if ((tr.flags & TF_ONE_WAY) == 0) {
                if (error < NO_ERROR) reply.setError(error);
                sendReply(reply, 0);
            }

            mCallingPid = origPid;
            mCallingUid = origUid;
            mStrictModePolicy = origStrictModePolicy;
            mLastTransactionBinderFlags = origTransactionBinderFlags;

        }
        break;

    case BR_DEAD_BINDER:
        {  //收到binder驱动发来的service死掉的消息，只有Bp端能收到
            BpBinder *proxy = (BpBinder*)mIn.readPointer();
            proxy->sendObituary();
            mOut.writeInt32(BC_DEAD_BINDER_DONE);
            mOut.writePointer((uintptr_t)proxy);
        } break;

    case BR_CLEAR_DEATH_NOTIFICATION_DONE:
        {
            BpBinder *proxy = (BpBinder*)mIn.readPointer();
            proxy->getWeakRefs()->decWeak(proxy);
        } break;

    case BR_FINISHED:
        result = TIMED_OUT;
        break;

    case BR_NOOP:
        break;

    case BR_SPAWN_LOOPER:
        mProcess->spawnPooledThread(false);//收到来自驱动的指示以创建一个
新线程，用于和Binder通信
        break;

    default:
        result = UNKNOWN_ERROR;
        break;
    }
```

```
    if (result != NO_ERROR) {
        mLastError = result;
    }

    return result;
}
```

[12]. BBinder::transact

==> /framework/native/libs/binder/Binder.cpp

服务端transact事务处理

```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    data.setDataPosition(0);

    status_t err = NO_ERROR;
    switch (code) {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
            err = onTransact(code, data, reply, flags); //【见流程13】
            break;
    }

    if (reply != NULL) {
        reply->setDataPosition(0);
    }

    return err;
}
```

[13]. BBinder::onTransact

==> /framework/native/libs/binder/Binder.cpp

服务端事务回调处理函数

```
status_t BBinder::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t /*flags*/)
{
    switch (code) {
        case INTERFACE_TRANSACTION:
            reply->writeString16(getInterfaceDescriptor());
            return NO_ERROR;

        case DUMP_TRANSACTION: {
            int fd = data.readFileDescriptor();
            int argc = data.readInt32();
            Vector<String16> args;
            for (int i = 0; i < argc && data.dataAvail() > 0; i++) {
               args.add(data.readString16());
            }
            return dump(fd, args);
        }

        case SYSPROPS_TRANSACTION: {
            report_sysprop_change();
            return NO_ERROR;
        }

        default:
            return UNKNOWN_TRANSACTION;
    }
}
```

[16] startThreadPool

==> /framework/native/libs/binder/ProcessState.cpp

先通过ProcessState::self()，来获取单例对象ProcessState，再进行启动线程池

```
void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);    //多线程同步 自动锁
    if (!mThreadPoolStarted) {
        mThreadPoolStarted = true;
        spawnPooledThread(true);  【见流程17】
    }
}
```

[17] spawnPooledThread

==> /framework/native/libs/binder/ProcessState.cpp

创建线程池

```
void ProcessState::spawnPooledThread(bool isMain)
{
    if (mThreadPoolStarted) {
        String8 name = makeBinderThreadName(); //获取Binder线程名
        sp<Thread> t = new PoolThread(isMain); //isMain=true
        t->run(name.string());
    }
}
```

- 获取Binder线程名，格式为 Binder_x，其中x为整数。每个进程中的binder编码是从1开始，依次递增;
- 在终端通过 ps -t | grep Binder ，能看到当前所有的Binder线程。

[20] joinThreadPool()

==> /framework/native/libs/binder/ProcessState.cpp

先通过IPCThreadState::self()，来获取单例对象IPCThreadState，再join到线程池中

```
void IPCThreadState::joinThreadPool(bool isMain)
{
    //isMain为true，则需要循环处理
    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
    set_sched_policy(mMyThreadId, SP_FOREGROUND); //设置前台调度策略

    status_t result;
    do {
        processPendingDerefs(); //清除队列的引用
        result = getAndExecuteCommand(); //处理下一条指令 【见流程20】

        if (result < NO_ERROR && result != TIMED_OUT && result != -ECO
NNREFUSED && result != -EBADF) {
            abort();
        }

        if(result == TIMED_OUT && !isMain) {
            break;
        }
    } while (result != -ECONNREFUSED && result != -EBADF);

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

将线程调度策略设置SP_FOREGROUND，当已启动的线程由后台的scheduling group创建，可以避免由后台线程优先级来执行初始化的transaction。

[21]. getAndExecuteCommand

==> /framework/native/libs/binder/IPCThreadState.cpp

获取并处理指令

```
status_t IPCThreadState::getAndExecuteCommand()
{
    status_t result;
    int32_t cmd;

    result = talkWithDriver(); //与binder进行交互
    if (result >= NO_ERROR) {
        size_t IN = mIn.dataAvail();
        if (IN < sizeof(int32_t)) return result;
        cmd = mIn.readInt32();

        pthread_mutex_lock(&mProcess->mThreadCountLock);
        mProcess->mExecutingThreadsCount++;
        pthread_mutex_unlock(&mProcess->mThreadCountLock);

        result = executeCommand(cmd);

        pthread_mutex_lock(&mProcess->mThreadCountLock);
        mProcess->mExecutingThreadsCount--;
        pthread_cond_broadcast(&mProcess->mThreadCountDecrement);
        pthread_mutex_unlock(&mProcess->mThreadCountLock);

        set_sched_policy(mMyThreadId, SP_FOREGROUND);
    }
    return result;
}
```

gityuan@gmail.com (mailto:gityuan@gmail.com) · Github

(https://github.com/yuanhuihui) · 天道酬勤 · © 2015 Yuanhh · Jekyll

(https://github.com/jekyll/jekyll) theme by HyG (https://github.com/Gaohaoyang)