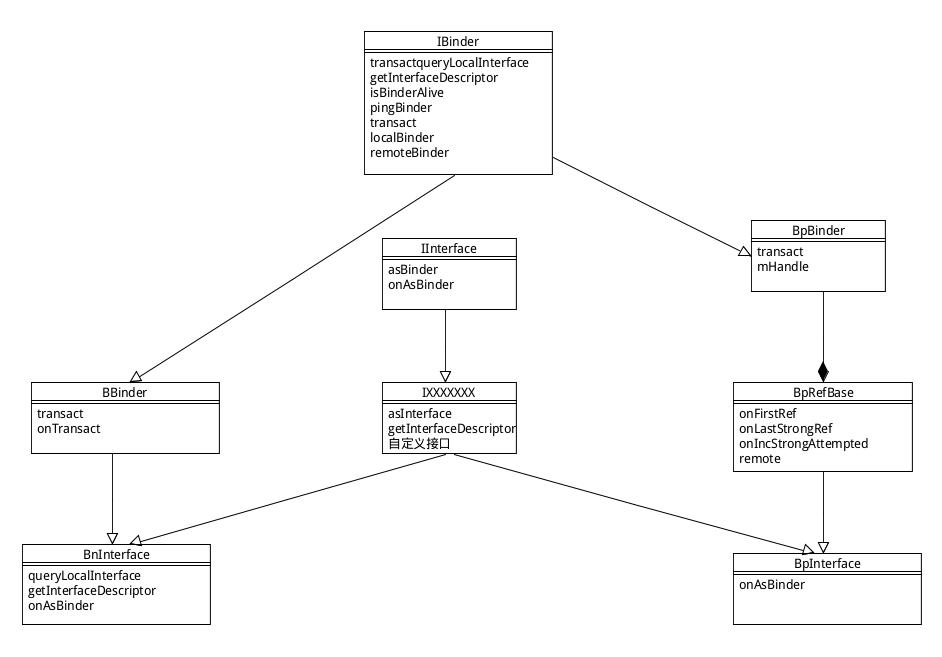
**Binder\_proxy**

1. **框架**



1. **创建BpXXX对象：**

BpXXXX = asInterface(sm->getService(name));

(1)通过从SM创建Bpbinder对象（sm->getService(name)）：

1.和bind\_node创建时调用servicemanager接口一样通过processState来创建BpServicemanager对象。

2.调用BpServicemanager的getService函数的接口，将name这个参数序列化到data中，调用到IPCThreadState::transact函数和binder驱动进行通信，最终通过binder\_context\_mgr\_node节点会调用到service\_manager进程中的接口。

**binder\_transaction三种传递情况：**

1.向servicemanager请求处理BC\_TRANSACTION命令。

2.向bnservice请求BC\_TRANSACTION命令。

3.处理BC\_REPLY命令。

第一种情况在binder\_transaction函数中运行路径：

target\_node = binder\_context\_mgr\_node; 获取请求的binder\_node;

target\_proc = target\_node->proc;为servicemanager的binder\_proc;

通过t->from = thread将当前请求的线程保存在binder\_transaction中;

由于target\_thread为空:

target\_list = &target\_proc->todo;

target\_wait = &target\_proc->wait;

创建binder\_transaction对象：

t->from = thread;//保存当前的binder\_thread 对象到binder\_transaction对象中

解析binder\_transaction\_data中的参数，Getservice中的参数在驱动层不需要处理。

保存binder\_transaction对象到thread->transaction\_stack：

t->need\_reply = 1;

t->from\_parent = thread->transaction\_stack;

thread->transaction\_stack = t;

将binder\_transaction加入到target\_list中同时唤醒target\_wait。

t->work.type = BINDER\_WORK\_TRANSACTION;

list\_add\_tail(&t->work.entry, target\_list);

if (target\_wait)

wake\_up\_interruptible(target\_wait);

在binder\_thread\_read函数中唤醒servicemanager的binder线程处理todo队列中的binder\_transaction对象，通过binder\_transaction对象来创建binder\_transaction\_data对象，将BR\_TRANSACTION命令和binder\_transaction\_data拷贝用户空间的参数中。

if (t->buffer->target\_node) {

......

cmd = BR\_TRANSACTION;

}

将binder\_transaction对象从todo队列中删除，将binder\_transaction对象保存在当前线程的binder\_thread的transaction\_stack。

if (cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)) {

t->to\_parent = thread->transaction\_stack;

t->to\_thread = thread;

thread->transaction\_stack = t;//保存t对象，为后面处理BC\_REPLY命令做准备。

}

返回到service\_manager的binder\_loop函数中，在该函数中调用binder\_parse来解析从驱动层传递的cmd和binder\_transaction\_data。在svcmgr\_handler函数通过client端的name参数获取在svclist链表中服务bn在servicemanager的desc的值,同时将查询的handle保存在reply中。

case BR\_TRANSACTION: {

struct binder\_transaction\_data \*txn = (struct binder\_transaction\_data \*) ptr;

bio\_init(&reply, rdata, sizeof(rdata), 4);

bio\_init\_from\_txn(&msg, txn);

res = func(bs, txn, &msg, &reply);//调用svcmgr\_handler函数处理SVC\_MGR\_CHECK\_SERVICE命令

binder\_send\_reply(bs, &reply, txn->data.ptr.buffer, res);//将handle保存在btd对象中，向驱动层发送BC\_REPLY命令

}

继续来到binder\_transaction函数中处理BC\_REPLY命令：

binder\_transaction三种传递情况中的第三种：

in\_reply\_to = thread->transaction\_stack;//获取在上一次保存在binder\_thread中传递的binder\_transaction对象。

target\_thread = in\_reply\_to->from;//获取保存的client请求的binder\_thread

target\_proc = target\_thread->proc;//获取client端的binder\_proc

获取client端的todo链表和wait等待队列：

if (target\_thread) {

e->to\_thread = target\_thread->pid;

target\_list = &target\_thread->todo;

target\_wait = &target\_thread->wait;

}

创建binder\_transaction对象，解析servicemanager传递的参数。

fp = (struct flat\_binder\_object \*)(t->buffer->data + \*offp);

off\_min = \*offp + sizeof(struct flat\_binder\_object);

switch (fp->type) {

case BINDER\_TYPE\_HANDLE:

struct binder\_ref\*ref=binder\_get\_ref(proc,fp->handle);//从servicemanager中通过handle获取对应的binder\_ref

if (ref->node->proc == target\_proc) {

} else {

struct binder\_ref \*new\_ref;

new\_ref = binder\_get\_ref\_for\_node(target\_proc, ref->node);//为client端创建binder\_node对应的binder\_ref和desc。

fp->handle = new\_ref->desc;//将desc保存在fp->handle中也就是binder\_transaction\_data的tr.data.ptr.buffer中

}

if (reply) {

binder\_pop\_transaction(target\_thread, in\_reply\_to);//释放binder\_transaction对象

}

将创建binder\_transaction对象加入到target\_thread的todo队列中，同时唤醒client中请求的等待队列。

t->work.type = BINDER\_WORK\_TRANSACTION;

list\_add\_tail(&t->work.entry, target\_list);

tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;

list\_add\_tail(&tcomplete->entry, &thread->todo);

if (target\_wait)

wake\_up\_interruptible(target\_wait);

在binder\_thread\_read函数中取出todo队列中的binder\_transaction对象，使用该对象创建binder\_transaction\_data对象。

struct binder\_transaction\_data tr;

tr.target.ptr = 0;

tr.cookie = 0;

cmd = BR\_REPLY;

tr.data\_size = t->buffer->data\_size;

tr.offsets\_size = t->buffer->offsets\_size;

tr.data.ptr.buffer = (binder\_uintptr\_t)(

(uintptr\_t)t->buffer->data +

proc->user\_buffer\_offset);

tr.data.ptr.offsets = tr.data.ptr.buffer +

ALIGN(t->buffer->data\_size,

sizeof(void \*));

if (put\_user(cmd, (uint32\_t \_\_user \*)ptr))

if (copy\_to\_user(ptr, &tr, sizeof(tr)))

kfree(t);

binder\_stats\_deleted(BINDER\_STAT\_TRANSACTION);

创建完binder\_transaction\_data返回到client端，在IPCThreadState::waitForResponse函数对BR\_REPLY命令进行处理，将驱动返回的handle保存到Parcel类型的reply中。返回到BpServiceManager的checkService函数中，在该函数掉用reply.readStrongBinder()函数来创建bpbinder对象。

virtual sp<IBinder> checkService( const String16& name) const

{

return reply.readStrongBinder();

}

sp<IBinder> Parcel::readStrongBinder() const

{

unflatten\_binder(ProcessState::self(), \*this, &val);

return val;

}

status\_t unflatten\_binder(const sp<ProcessState>& proc, const Parcel& in, sp<IBinder>\* out)

{

switch (flat->type) {

case BINDER\_TYPE\_HANDLE:

\*out = proc->getStrongProxyForHandle(flat->handle);

return finish\_unflatten\_binder(

static\_cast<BpBinder\*>(out->get()), \*flat, in);

}

sp<IBinder> ProcessState::getStrongProxyForHandle(int32\_t handle)

{

b = new BpBinder(handle); //使用从驱动层传过来的handle创建BpBinder对象。

e->binder = b;

if (b) e->refs = b->getWeakRefs();

result = b;

return result;

}

调用asInterface函数来创建BpInterface的子类：

#define DECLARE\_META\_INTERFACE(ServiceManager)

static const android::String16 descriptor;

static android::sp<IServiceManager> asInterface( const android::sp<android::IBinder>& obj);

virtual const android::String16& getInterfaceDescriptor() const;

IServiceManager();

virtual ~IServiceManager();

#define IMPLEMENT\_META\_INTERFACE(ServiceManager, "android.os.IServiceManager")

const android::String16 IServiceManager::descriptor("android.os.IServiceManager");

const android::String16&

IServiceManager::getInterfaceDescriptor() const {

return IServiceManager::descriptor;

}

android::sp<IServiceManager>IServiceManager::asInterface( const android::sp<android::IBinder>& obj)

{

android::sp<IServiceManager> intr;

if (obj != NULL) {

intr = static\_cast<IServiceManager\*>(

obj->queryLocalInterface(

IServiceManager::descriptor).get());

if (intr == NULL) {

intr = new BpServiceManager(obj);

}

}

return intr;

}

1. **发送请求**

通过handle查找到binder\_ref，通过查找到的binder\_ref获取binder\_node.在该节点中获取继承BnInterface的服务实体对象，然后调用其相应code的接口进行处理请求。

4. **binder\_transaction传递的三种数据类型：**

1.BINDER\_TYPE\_BINDER

binder\_new\_node：为当前进程创建binder\_node

binder\_get\_ref\_for\_node:为接收者创建binder\_ref

2.BINDER\_TYPE\_HANDLE

binder\_get\_ref:获取发送者的binder\_ref

binder\_get\_ref\_for\_node:为接收者创建binder\_ref

3.BINDER\_TYPE\_FD

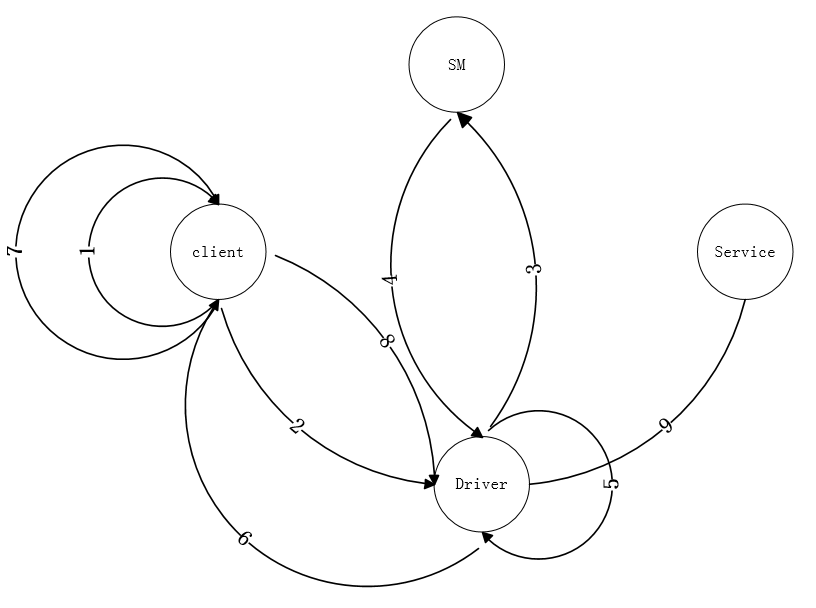
binder\_proc:struct files\_struct \*files; 保存当前进程所有的文件描述符。

Fget ：传递者：获取当前fd的struct file结构。

\_\_alloc\_fd: 为接受者：分配一个未使用的fd。

\_\_fd\_install:将传递者的struct file结构和接收者分配的fd关联起来。

**4.总结：**



1. 创建BpServiceManager对象。
2. 使用值为0的handle请求CHECK\_SERVICE\_TRANSACTION和Driver进行通信。
3. 获取sm的binder\_node调用到servicemanager的CHECK\_SERVICE\_TRANSACTION获取service在servicemanager中的handle。
4. 通过handle查找出servicemanager中的binder\_ref。
5. 通过binder\_ref获取service端的binder\_node，通过获取的binder\_node在client端创建binder\_ref和desc
6. 将desc以handle值传送给client端
7. 通过handle创建BpBinder对象和BnInterface对象。
8. 通过BpBinder对象中的handle在驱动中查找出相应的binder\_ref,通过binder\_ref 获取service端的binder\_node节点从而获取到service端的BnInterface对象进行client的请求的处理。