Andorid Binder进程间通信---Binder本地对象，实体对象，引用对象，代理对象的引用计数

时间：2014-06-02 23:57:40 阅读：178 评论：0 收藏：0 [点我收藏+]

标签：des android class 代码 style int

本文参考《Android系统源代码情景分析》，作者罗升阳。

一、Binder库(libbinder)代码:

~/Android/frameworks/base/libs/binder

----BpBinder.cpp

----Parcel.cpp

----ProcessState.cpp

----Binder.cpp

----IInterface.cpp

----IPCThreadState.cpp

----IServiceManager.cpp

----Static.cpp

~/Android/frameworks/base/include/binder

----Binder.h

----BpBinder.h

----IInterface.h

----IPCThreadState.h

----IServiceManager.h

----IBinder.h

----Parcel.h

----ProcessState.h

驱动层代码:

～/Android//kernel/goldfish/drivers/staging/android

----binder.c

----binder.h

二、Binder代理对象的生命周期

Binder代理对象是一个类型为BpBinder的对象，它是在用户空间创建，并且运行在Client进程中。

Binder代理对象一方面会被运行在Client进程中的其他对象引用，另一方面它也会引用Binder驱动程序中的Binder引用对象。由于BpBinder类继承了RefBase类，因此，client进程中的其他对象可以简单地通过智能指针来引用这些Binder代理对象，以便可以控制它们的生命周期。由于Binder驱动程序中的Binder引用对象是运行在内核空间的，Binder代理对象就不能通过智能指针来引用它们，因此，Client进程就需要通过BC_ACQUIRE、BC_INCREFS、BC_RELEASE、BC_DECREFS四个协议来引用Binder驱动程序中的Binder引用对象。

Binder代理对象的创建过程如下所示：

~/Android/frameworks/base/libs/binder

----BpBinder.cpp

```
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);//增加弱引用计数
}
```

Binder代理对象的生命周期受到弱引用计数影响，接着调用当前线程内部的IPCThreadState对象的成员函数incWeakHandle来增加相应的Binder引用对象的弱引用计数。

~/Android/frameworks/base/libs/binder

---IPCThreadState.cpp

```
void IPCThreadState::incWeakHandle(int32_t handle)
{
    LOG_REMOTEREFS("IPCThreadState::incWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_INCREFS);//增加弱引用计数
```

```
    mOut.writeInt32(handle);
}
```

IPCThreadState类的成员函数incWeakHandle将增加Binder引用对象的弱引用计数的操作缓存在内部的一个成员变量mOut中，等到下次使用IO控制命令BINDER_WRITE_READ进入到Binder驱动程序时，再请求Binder驱动程序增加相应的Binder引用对象的弱引用计数。

另外，当一个Binder代理对象销毁时，当前线程就会调用其内部的IPCThreadState对象的成员函数decWeakHandle来减少相应的Binder引用对象的弱引用计数。

~/Android/frameworks/base/libs/binder

----BpBinder.cpp

```
BpBinder::~BpBinder()
{
    LOGV("Destroying BpBinder %p handle %d\n", this, mHandle);

    IPCThreadState* ipc = IPCThreadState::self();

    if (ipc) {
        ........
        ipc->decWeakHandle(mHandle);//减少弱引用计数
    }
}
```

~/Android/frameworks/base/libs/binder

---IPCThreadState.cpp

```
void IPCThreadState::decWeakHandle(int32_t handle)
{
    LOG_REMOTEREFS("IPCThreadState::decWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_DECREFS);//减少弱引用计数
    mOut.writeInt32(handle);
}
```

IPCThreadState类的成员函数decWeakHandle将减少Binder引用对象的弱引用计数的操作缓存在内部的一个成员变量mOut中，等到下次使用IO控制命令BINDER_WRITE_READ进入到Binder驱动程序时，再请求Binder驱动程序减少相应的Binder引用对象的弱引用计数。

前面提到，一个Binder代理对象可能会被用户空间的其他对象引用，当其他对象通过强指针来引用这些Binder代理对象时，Client进程就需要请求Binder驱动程序增加相应的Binder引用对象的强引用计数。为了减少Clinet进程和Binder驱动程序的交互开销，当一个Binder代理对象第一次被强指针引用时，Client进程才会请求Binder驱动程序增加相应的Binder引用对象的强引用计数；同时一个Binder代理对象不再被任何强指针引用时，Client进程就会请求Binder驱动程序减少相应的Binder引用对象的强引用计数。

当一个对象第一次被强指针引用时，它的成员函数onFirstRef就会被调用；而当一个对象不再被任何强指针引用时，它的成员函数onLastStrongRef就会被调用。

~/Android/frameworks/base/libs/binder

----BpBinder.cpp

```
void BpBinder::onFirstRef()
{
    ........
    IPCThreadState* ipc = IPCThreadState::self();
    if (ipc) ipc->incStrongHandle(mHandle);//增加强引用计数
}
```

它调用当前线程内部的IPCThreadState对象incStrongHandle来增加相应的Binder引用对象的强引用计数。

~/Android/frameworks/base/libs/binder

---IPCThreadState.cpp

```
void IPCThreadState::incStrongHandle(int32_t handle)
{
    .........
    mOut.writeInt32(BC_ACQUIRE);//增加强引用计数
    mOut.writeInt32(handle);
}
```

IPCThreadState类的成员函数incStrongHandle将增加Binder引用对象的强引用计数的操作缓存在内部的一个成员变量mOut中，等到下次使用IO控制命令BINDER_WRITE_READ进入到Binder驱动程序时，再请求Binder驱动程序增加相应的Binder引用对象的强引用计数。

~/Android/frameworks/base/libs/binder

----BpBinder.cpp

```
void BpBinder::onLastStrongRef(const void* id)
{
    .......
    IPCThreadState* ipc = IPCThreadState::self();
    if (ipc) ipc->decStrongHandle(mHandle);//减少强引用计数
}
```

~/Android/frameworks/base/libs/binder

---IPCThreadState.cpp

```
void IPCThreadState::decStrongHandle(int32_t handle)
{
    .........
    mOut.writeInt32(BC_RELEASE);//减少强引用计数
    mOut.writeInt32(handle);
}
```

IPCThreadState类的成员函数decStrongHandle将减少Binder引用对象的强引用计数的操作缓存在内部的一个成员变量mOut中，等到下次使用IO控制命令BINDER_WRITE_READ进入到Binder驱动程序时，再请求Binder驱动程序减少相应的Binder引用对象的强引用计数。

三、Binder引用对象的生命周期

Binder引用对象是一个类型为binder_ref的对象，它是在Binder驱动程序中创建的，并且被用户被用户空间中的Binder代理对象所引用。

继续上一节中，mOut已经存入了命令和命令对应的数据，使用IO控制命令BINDER_WRITE_READ进入到Binder驱动程序。

～/Android//kernel/goldfish/drivers/staging/android

----binder.c

```
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))
        return -EFAULT;
    ptr += sizeof(uint32_t);
    ......
    switch (cmd) {//获取的命令
    case BC_INCREFS:
    case BC_ACQUIRE:
    case BC_RELEASE:
    case BC_DECREFS: {
        uint32_t target;
        struct binder_ref *ref;
        const char *debug_string;

        if (get_user(target, (uint32_t __user *)ptr))//获取的句柄值
            return -EFAULT;
        ptr += sizeof(uint32_t);
        if (target == 0 && binder_context_mgr_node &&
            (cmd == BC_INCREFS || cmd == BC_ACQUIRE)) {
            ref = binder_get_ref_for_node(proc,
                    binder_context_mgr_node);//来获得一个与Service Manager对应的Binder引用对象。
如果Binder驱动程序之前尚未为Client进程创建过这个Binder引用对象，那么函数binder_get_ref_for_node就会为它创建
一个，以便Client进程可以引用它
            if (ref->desc != target) {
                binder_user_error("binder: %d:"
                    "%d tried to acquire "
                    "reference to desc 0, "
                    "got %d instead\n",
                    proc->pid, thread->pid,
```

```
                        ref->desc);
                }
        } else
                ref = binder_get_ref(proc, target);//来获取对应的Binder引用对象
        if (ref == NULL) {
                binder_user_error("binder: %d:%d refcou"
                        "nt change on invalid ref %d\n",
                        proc->pid, thread->pid, target);
                break;
        }
        switch (cmd) {
        case BC_INCREFS://增加弱引用计数
                debug_string = "IncRefs";
                binder_inc_ref(ref, 0, NULL);
                break;
        case BC_ACQUIRE://增加强引用计数
                debug_string = "Acquire";
                binder_inc_ref(ref, 1, NULL);
                break;
        case BC_RELEASE://减少强引用计数
                debug_string = "Release";
                binder_dec_ref(ref, 1);
                break;
        case BC_DECREFS://增加弱引用计数
        default:
                debug_string = "DecRefs";
                binder_dec_ref(ref, 0);
                break;
        }
        ........
}
.......
```

```
            *consumed = ptr - buffer;
        }
        return 0;
}
```

Binder驱动程序根据Client进程传过来的句柄值找到要修改引用计数的Binder引用对象。这个句柄值是Binder驱动程序为Client进程创建的，用来关联用户空间的Binder代理对象和内核空间的Binder引用对象。在Binder驱动程序中，有一个特殊的句柄值，它的值等于0，用来描述一个与Service Manager关联的Binder引用对象。

首先获得由Client进程传进来的句柄值，并且保存在变量target中。接着判断该句柄值是否等于0，如果等于0，而且存在binder_context_mgr_node，并且执行的操作是增加它的强引用计数或者弱引用计数。这时就会调用函数binder_get_ref_for_node来获得一个与Service Manager对应的Binder引用对象。如果Binder驱动程序之前尚未为Client进程创建过这个Binder引用对象，那么函数binder_get_ref_for_node就会为它创建一个，以便Client进程可以引用它。

如果Client进程传进来的句柄值不等于0，那么就会调用函数binder_get_ref来获取对应的Binder引用对象。如果对应的Binder引用对象不存在，那么函数binder_get_ref不会根据这个句柄值来创建一个Binder引用对象返回给调用者。因为句柄值本来就是在创建Binder引用对象的时候分配的，即先有Binder引用对象，再有句柄值，只不过等于0的句柄比较特殊，它允许先有句柄值，再有Binder引用对象。

得到了目标Binder引用对象ref之后，switch语句根据不同的协议来增加后者减少它的强引用计数或者弱引用计数。

增加引用计数：

～/Android//kernel/goldfish/drivers/staging/android

----binder.c

```
static int
binder_inc_ref(
        struct binder_ref *ref, int strong, struct list_head *target_list)
{
        int ret;
        if (strong) {//增加强引用计数
                if (ref->strong == 0) {//第一次增加强引用计数
                        ret = binder_inc_node(ref->node, 1, 1, target_list);//增加它所引用的Binder实体对象的外部强引用计数internal_strong_refs
                        if (ret)
                                return ret;
```

```
            }
            ref->strong++;
        } else {//增加弱引用计数
            if (ref->weak == 0) {//第一次增加弱引用计数
                ret = binder_inc_node(ref->node, 0, 1, target_list);//增加它所引用的Binder实体对象的外部弱引用计数internal_week_refs
                if (ret)
                    return ret;
            }
            ref->weak++;
        }
        return 0;
}
```

第一次参数ref用来描述要操作的Binder引用对象；第二个参数strong用来描述要增加Binder引用对象ref的强引用计数还是弱引用计数；第三个参数target_list指向一个目标进程或者目标线程的todo队列，当它不为NULL时，就表示增加了Binder引用对象ref的引用计数之后，要相应的增加与它所引用的Binder实体对象对应的Binder本地对象的引用计数。

如果strong的值为1，就要增加Binder引用对象ref的强引用计数，就是增加它的成员变量strong的值。如果一个Binder引用对象的强引用计数由0变1，那么就需要调用函数binder_inc_node来增加它所引用的Binder实体对象的外部强引用计数internal_strong_refs，避免它还在被一个Binder引用对象引用的情况下被销毁。

如果strong的值为0，就要增加Binder引用对象ref的弱引用计数，就是增加它的成员变量week的值。如果一个Binder引用对象的弱引用计数由0变1，那么就需要调用函数binder_inc_node来增加它所引用的Binder实体对象的外部弱引用计数internal_week_refs，避免它还在被一个Binder引用对象引用的情况下被销毁。

减少引用计数：

～/Android//kernel/goldfish/drivers/staging/android

----binder.c

```
static int
binder_dec_ref(struct binder_ref *ref, int strong)
{
    if (strong) {//减少强引用计数
        if (ref->strong == 0) {
```

```
                binder_user_error("binder: %d invalid dec strong, "
                                  "ref %d desc %d s %d w %d\n",
                                  ref->proc->pid, ref->debug_id,
                                  ref->desc, ref->strong, ref->weak);
                return -EINVAL;
            }
            ref->strong--;
            if (ref->strong == 0) {//第一次减少强引用计数
                int ret;
                ret = binder_dec_node(ref->node, strong, 1);
                if (ret)
                    return ret;
            }
        } else {//减少弱引用计数
            if (ref->weak == 0) {
                binder_user_error("binder: %d invalid dec weak, "
                                  "ref %d desc %d s %d w %d\n",
                                  ref->proc->pid, ref->debug_id,
                                  ref->desc, ref->strong, ref->weak);
                return -EINVAL;
            }
            ref->weak--;
        }
        if (ref->strong == 0 && ref->weak == 0)//强引用计数和弱引用计数同时为0
            binder_delete_ref(ref);//销毁binder_ref对象
        return 0;
    }
```

第一个参数ref用来描述要操作的Binder引用对象；第二个参数strong用来描述是要减少Binder引用对象ref的强引用计数还是弱引用计数。

如果参数strong的值等于1，就说明要减少Binder引用对象ref的强引用计数，也就是减少它的成员变量strong的值。如果一个Binder引用对象的强引用计数由1变成0，那么就需要调用函数binder_dec_node来减少它所引用的Binder实体对象的外部强引用计数internal_strong_refs，表示它不再被该Binder引用对象通过

外部强引用计数来引用了。

如果参数strong的值等于0，就说明要减少Binder引用对象ref的弱引用计数，也就是减少它的成员变量week的值。

一个Binder引用对象的生命周期同时受到它的强引用计数和弱引用计数影响，因此都它们的都等于0时，就需要调用函数binder_delete_ref来销毁该binder_ref对象了。

binder_delete_ref实现如下：

～/Android//kernel/goldfish/drivers/staging/android

----binder.c

```
static void
binder_delete_ref(struct binder_ref *ref)
{
    .......
    rb_erase(&ref->rb_node_desc, &ref->proc->refs_by_desc);//将它从这两个红黑树中移除
    rb_erase(&ref->rb_node_node, &ref->proc->refs_by_node);//将它从这两个红黑树中移除
    if (ref->strong)//一个Binder引用对象是被强行销毁的
        binder_dec_node(ref->node, 1, 1);//减少它所引用的Binder实体对象的外部强引用计数internal_strong_refs
    hlist_del(&ref->node_entry);//它所引用的Binder实体对象的Binder引用列表中删除
    binder_dec_node(ref->node, 0, 1);//减少一个即将要销毁的Binder引用对象所引用的Binder实体对象的外部弱引用计数
    if (ref->death) {//如果注册了死亡通知
        ........
        list_del(&ref->death->work.entry);//删除以及销毁该死亡接受通知
        kfree(ref->death);
        ........
    }
    kfree(ref);//销毁Binder引用对象ref
    .......
}
```

一个Binder引用对象分别保存在其宿主进程的两个红黑树中，首先将它从这两个红黑树中移除。同时一个Binder引用对象还保存在它所引用的Binder实体对象的Binder引用列表中，因此调用hlist_del将它从该列表删除。

如果一个Binder引用对象是被强行销毁的，即在它的强引用计数还大于0的情况下销毁，那么需要调用函

数binder_dec_node来减少它所引用的Binder实体对象的外部强引用计数internal_strong_refs，因为当该Binder引用对象销毁后，它所引用的Binder实体对象就少了一个Binder引用对象通过外部强引用计数来引用了。

然后调用了函数binder_dec_node来减少一个即将要销毁的Binder引用对象所引用的Binder实体对象的外部弱引用计数。我们知道，一个Binder实体对象是没有一个类似于internal_strong_refs的外部弱引用计数，这里调用函数binder_dec_node只是为了检查是否需要减少相对应的Binder本地对象的弱引用计数。

最后检查即将要销毁的Binder引用对象是否注册有一个死亡接受通知。如果是，就删除以及销毁该死亡接受通知。然后调用kfree来销毁Binder引用对象ref。

四、Binder实体对象的生命周期

Binder实体对象是一个类型为binder_node的对象，它是在Binder驱动程序中创建的，并且被Binder驱动程序中Binder引用对象所引用。

当Client进程第一次引用一个Binder实体对象时，Binder驱动程序就会在内部为它创建一个Binder引用对象。

在Binder驱动程序中，增加一个Binder实体对象的引用计数是通过函数binder_inc_node来实现的。如下：

～/Android//kernel/goldfish/drivers/staging/android

----binder.c

```
static int
binder_inc_node(struct binder_node *node, int strong, int internal,
                struct list_head *target_list)
{
        if (strong) {
                if (internal) {//增加Binder实体对象node的外部强引用计数internal_strong_ref
                        ......
                        node->internal_strong_refs++;
                } else
                        node->local_strong_refs++;//增加Binder实体对象node的外部强引用计数local_strong_refs
                if (!node->has_strong_ref && target_list) {//没有增加对应的Binder本地对象的强引用计数
                        list_del_init(&node->work.entry);
                        list_add_tail(&node->work.entry, target_list);//加入一个工作项来通知该进程增加对应的Binder本地对象的强引用计数
                }
        } else {
```

```
            if (!internal)
                node->local_weak_refs++;//要增加Binder实体对象node的内部强引用计数local_week_refs
            if (!node->has_weak_ref && list_empty(&node->work.entry)) {//没有增加对应的Binder本地对象的弱引用计数
                if (target_list == NULL) {
                    printk(KERN_ERR "binder: invalid inc weak node "
                        "for %d\n", node->debug_id);
                    return -EINVAL;
                }
                list_add_tail(&node->work.entry, target_list);//加入一个工作项来通知该进程增加对应的Binder本地对象的弱引用计数
            }
        }
        return 0;
}
```

第一个参数node表示要增加引用计数的Binder实体对象；第二个参数strong表示要增加强引用计数还是弱引用计数；第三个参数internal用来区分增加的是内部引用计数还是外部引用计数；第四个参数target_list指向一个目标进程或者目标线程的todo队列，当它不为NULL时，就表示增加了Binder实体对象node的引用计数之后，要相应地增加它所引用的Binder本地对象的引用计数。

内部引用计数是相对于该Binder实体对象所在的Server进程而言的，而外部引用计数是相对于引用了该Binder实体对象的Client进程而言的。

当Binder驱动程序请求Server进程中某一个Binder本地对象来执行一个操作时，它就会增加与该Binder本地对象对应的Binder实体对象的内部引用计数，避免该Binder实体对象被过早地销毁。

而当一个Client进程通过一个Binder引用对象来引用一个Binder实体对象时，Binder驱动程序就会增加它的外部引用计数，也是避免该Binder实体对象被过早地销毁。 internal_strong_refs是一个外部强引用计数，用来描述有多少个Binder引用对象是通过强引用计数来引用该Binder实体对象的。为什么没有一个弱引用计数internal_weak_refs？因为，Binder实体对象的Binder引用对象列表refs的大小就已经隐含了它的外部弱引用计数。

了解了这些基础知识后，函数binder_inc_node的实现原理就容易理解了。

如果参数strong为1，internal为1，表示要增加Binder实体对象node的外部强引用计数internal_strong_refs。

如果参数strong为1，internal为0，表示要增加Binder实体对象node的外部强引用计数local_strong_refs。

无论哪种情况，如果此时没有增加对应的Binder本地对象的强引用计数，即它的成员变量has_strong_ref也等于0，并且target_list不为NULL。那么就加入一个工作项来通知该进程增加对应的Binder本地对象的强引用计数。

如果参数strong为0，internal为0，表示要增加Binder实体对象node的内部强引用计数local_week_refs。

如果参数strong为0，internal为1，因为，Binder实体对象的Binder引用对象列表refs的大小就已经隐含了它的外部弱引用计数，所以不会增加外部弱引用计数。

无论哪种情况，如果此时没有增加对应的Binder本地对象的弱引用计数，即它的成员变量has_week_ref也等于0，并且工作项为空，那么就加入一个工作项来通知该进程增加对应的Binder本地对象的弱引用计数。

在Binder驱动程序中，减少一个Binder实体对象的引用计数是通过函数binder_dec_node来实现的，如下：

～/Android//kernel/goldfish/drivers/staging/android

----binder.c

```
static int
binder_dec_node(struct binder_node *node, int strong, int internal)
{
    if (strong) {
        if (internal)
            node->internal_strong_refs--;//减少Binder实体对象node的外部强引用计数internal_strong_refs
        else
            node->local_strong_refs--;//减少Binder实体对象node的内部强引用计数local_strong_refs
        if (node->local_strong_refs || node->internal_strong_refs)
            return 0;
    } else {
        if (!internal)
            node->local_weak_refs--;//减少Binder实体对象node的内部强引用计数local_week_refs
        if (node->local_weak_refs || !hlist_empty(&node->refs))
            return 0;
    }
    if (node->proc && (node->has_strong_ref || node->has_weak_ref)) {//说明Binder实体对象node的强引用计数或者弱引用计数等于0了，这时候如果它的成员变量has_strong_ref和has_week_ref的其中有一个等于1
        if (list_empty(&node->work.entry)) {//是否已经将一个类型为BINDER_WORK_NODE的工作项添加到要减少
```

```
引用计数的Binder本地对象所在进程的todo队列中
                list_add_tail(&node->work.entry, &node->proc->todo);//添加该工作项
                wake_up_interruptible(&node->proc->wait);//调用函数wake_up_interruptible唤醒该进程来处理该
工作项
            }
        } else {//要么是Binder实体对象node的宿主进程结构体为NULL，要么是它的成员函数has_strong_ref和has_weak_
ref都等于0
            if (hlist_empty(&node->refs) && !node->local_strong_refs &&
                !node->local_weak_refs) {//如果Binder实体对象node的所有引用计数均等于0，销毁该Binder实体对象no
de
                list_del_init(&node->work.entry);
                if (node->proc) {
                    rb_erase(&node->rb_node, &node->proc->nodes);//它是保存在其宿主进程结构体的Binder实
体对象列表中，然后把它删除
                    .....
                } else {
                    hlist_del(&node->dead_node);//该Binder实体对象node之前保存在一个死亡Binder实体对象列表
中，然后把它删除
                    ......
                }
                kfree(node);//kfree来销毁Binder实体对象node
                ......
            }
        }

        return 0;
}
```

第一个参数node表示要减少引用计数的Binder实体对象；第二个参数strong表示要减少强引用计数还是弱引用计数；第三个参数internal用来区分减少的是内部引用计数，还是外部引用计数。

如果参数strong为1，internal为1，表示要减少Binder实体对象node的外部强引用计数

internal_strong_refs。

如果参数strong为1，internal为0，表示要减少Binder实体对象node的内部强引用计数local_strong_refs。

如果参数strong为0，internal为0，表示要减少Binder实体对象node的内部强引用计数local_week_refs。

如果参数strong为0，internal为1，因为，在调用binder_dec_node之前，就已经将一个对应的Binder引用对象从Binder实体对象node的Binder引用对象列表中删除了，即相当于减少了Binder实体对象node的外部弱引用计数，因此，就没有减少相应计数的操作。

如果binder_dec_node减少了Binder实体对象node的强引用计数或者弱引用计数之后，如果它的强引用计数或者弱引用计数不等于0，那么函数binder_dec_node就不往下处理了。否则，就需要减少与Binder实体对象node对应的Binder本地对象的引用计数。

```
if (node->proc && (node->has_strong_ref || node->has_weak_ref)) {
```

执行到该函数时，说明Binder实体对象node的强引用计数或者弱引用计数等于0了，这时候如果它的成员变量has_strong_ref和has_week_ref的其中有一个等于1，那么就需要减少与Binder实体对象node对应的Binder本地对象的强引用计数或者弱引用计数，因为此时Binder实体对象node不需要通过强引用计数或者弱引用计数来引用相对应的Binder本地对象，但是之前增加过它的强引用计数或者弱引用计数。

然后判断是否已经将一个类型为BINDER_WORK_NODE的工作项添加到要减少引用计数的Binder本地对象所在进程的todo队列中。如果不是，那么就添加该工作项，调用函数wake_up_interruptible唤醒该进程来处理该工作项，以便马上执行减少相应的Binder本地对象的引用计数的工作。

如果上面列出的函数为false，那么要么是Binder实体对象node的宿主进程结构体为NULL，要么是它的成员函数has_strong_ref和has_weak_ref都等于0。这时候如果Binder实体对象node的所有引用计数均等于0，那么就需要销毁该Binder实体对象node。

如果发现它的宿主进程结构体为NULL，那就说明该Binder实体对象node之前保存在一个死亡Binder实体对象列表中，然后把它删除。

如果发现它的宿主进程结构体不为NULL，那就说明它是保存在其宿主进程结构体的Binder实体对象列表中，然后把它删除。

最后kfree来销毁Binder实体对象node。

五、Binder本地对象的生命周期

Binder本地对象是一个类型为BBinder的对象，它是在用户空间中创建的，并且运行在Server进程中，Binder本地对象一方面会被运行在Server进程中的其他对象引用，另一方面也会被Binder驱动程序中的Binder实体对象引用。由于BBinder类继承了RefBase类，因此Server进程中的其他对象可以简单地通过智能指针来引用这些Binder本地对象，以便控制它们的生命周期。由于Binder驱动程序中的Binder实体对象是运行在内核空间的，它不能通过智能指针来引用运行在用户空间的Binder本地对象。

总结来说，Binder驱动程序就是通过BR_INCREFS、BR_ACQUIRE、BR_DECREFS、BR_RELEASE

协议来引用运行在Server进程中的Binder本地对象，相关的代码实现在函数binder_thread_read中，如下所示：

～/Android//kernel/goldfish/drivers/staging/android

----binder.c

```
static int
binder_thread_read(struct binder_proc *proc, struct binder_thread *thread,
	void  __user *buffer, int size, signed long *consumed, int non_block)
{
	.....
	while (1) {
		uint32_t cmd;
		.......
		struct binder_work *w;
		.......

		if (!list_empty(&thread->todo))
			w = list_first_entry(&thread->todo, struct binder_work, entry);
		else if (!list_empty(&proc->todo) && wait_for_proc_work)
			w = list_first_entry(&proc->todo, struct binder_work, entry);
		else {
			.....
		}

		.......
		switch (w->type) {
		......
		case BINDER_WORK_NODE: {
			struct binder_node *node = container_of(w, struct binder_node, work);
			uint32_t cmd = BR_NOOP;
			const char *cmd_name;
			int strong = node->internal_strong_refs || node->local_strong_refs;//如果有强引用计数，那么就将
变量strong的值设置为1；否则为0
```

```
int weak = !hlist_empty(&node->refs) || node->local_weak_refs || strong;//有弱引用计数，那么就将变量weak的值设置为1；否则就设置为0
if (weak && !node->has_weak_ref) {//说明该Binder实体对象已经引用了一个Binder本地对象，但是还没有增加它的弱引用计数
    cmd = BR_INCREFS;
    cmd_name = "BR_INCREFS";//BR_INCREFS协议来请求增加对应的Binder本地对象的弱引用计数
    node->has_weak_ref = 1;
    node->pending_weak_ref = 1;
    node->local_weak_refs++;
} else if (strong && !node->has_strong_ref) {//该Binder实体对象已经引用了一个Binder本地对象，但是还没有增加它的强引用计数
    cmd = BR_ACQUIRE;
    cmd_name = "BR_ACQUIRE";//BR_ACQUIRE协议来请求增加对应的Binder本地对象的强引用计数
    node->has_strong_ref = 1;
    node->pending_strong_ref = 1;
    node->local_strong_refs++;
} else if (!strong && node->has_strong_ref) {//该Binder实体对象已经不再引用了一个Binder本地对象，但是还没有减少它的强引用计数
    cmd = BR_RELEASE;//使用BR_RELEASE协议来请求减少对应的Binder本地对象的强引用计数
    cmd_name = "BR_RELEASE";
    node->has_strong_ref = 0;
} else if (!weak && node->has_weak_ref) {//该Binder实体对象已经不再引用了一个Binder本地对象，但是还没有减少它的弱引用计数
    cmd = BR_DECREFS;//BR_DECREFS协议来请求减少对应的Binder本地对象的弱引用计数
    cmd_name = "BR_DECREFS";
    node->has_weak_ref = 0;
}
if (cmd != BR_NOOP) {//然后将前面准备好的协议以及协议内容写入到由Server进程所提供的一个用户空间缓冲
    if (put_user(cmd, (uint32_t __user *)ptr))//命令
        return -EFAULT;
    ptr += sizeof(uint32_t);
```

```
                    if (put_user(node->ptr, (void * __user *)ptr))//Binder本地对象内部的一个弱引用计数对象
                        return -EFAULT;
                    ptr += sizeof(void *);
                    if (put_user(node->cookie, (void * __user *)ptr))//Binder本地对象的地址
                        return -EFAULT;
                    ptr += sizeof(void *);

                    binder_stat_br(proc, thread, cmd);
                    if (binder_debug_mask & BINDER_DEBUG_USER_REFS)
                        printk(KERN_INFO "binder: %d:%d %s %d u%p c%p\n",
                            proc->pid, thread->pid, cmd_name, node->debug_id, node->ptr, node->cookie);
                } else {
                    list_del_init(&w->entry);
                    .......
                }
            } break;
        .........
        return 0;
}
```

首先检查该Binder实体对象是否有强引用计数和弱引用计数，如果有强引用计数，那么就将变量strong的值设置为1；否则为0。

同样，如果Binder实体对象有弱引用计数，那么就将变量weak的值设置为1；否则就设置为0。

如果变量weak的值等于1，并且目标Binder实体对象的成员变量has_weak_ref的值等于0，那么就说明该Binder实体对象已经引用了一个Binder本地对象，但是还没有增加它的弱引用计数。所以就使用BR_INCREFS协议来请求增加对应的Binder本地对象的弱引用计数。

如果变量strong的值等于1，并且目标Binder实体对象的成员变量has_strong_ref的值等于0，那么就说明该Binder实体对象已经引用了一个Binder本地对象，但是还没有增加它的强引用计数。所以就使用BR_ACQUIRE协议来请求增加对应的Binder本地对象的强引用计数。

如果变量strong的值等于0，并且目标Binder实体对象的成员变量has_strong_ref的值等于1，那么就说明该Binder实体对象已经不再引用了一个Binder本地对象，但是还没有减少它的强引用计数。所以就使用BR_RELEASE协议来请求减少对应的Binder本地对象的强引用计数。

如果变量weak的值等于0，并且目标Binder实体对象的成员变量has_weak_ref的值等于1，那么就说明该Binder实体对象已经不再引用了一个Binder本地对象，但是还没有减少它的弱引用计数。所以就使用

BR_DECREFS协议来请求减少对应的Binder本地对象的弱引用计数。

然后将前面准备好的协议以及协议内容写入到由Server进程所提供的一个用户空间缓冲区，然后返回到Server进程的用户空间。Server进程调用IPCThreadState类的成员函数executeCommand中处理BR_INCREFS、BR_ACQUIRE、BR_DECREFS、BR_RELEASE这四个协议。如下：

~/Android/frameworks/base/libs/binder

----IPCThreadState.cpp

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

    switch (cmd) {
    ......
    case BR_ACQUIRE://BR_ACQUIRE协议来请求增加对应的Binder本地对象的强引用计数
        refs = (RefBase::weakref_type*)mIn.readInt32();//Binder本地对象内部的一个弱引用计数对象
        obj = (BBinder*)mIn.readInt32();//Binder本地对象的地址
        ........
        obj->incStrong(mProcess.get());
        ........
        mOut.writeInt32(BC_ACQUIRE_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_RELEASE://使用BR_RELEASE协议来请求减少对应的Binder本地对象的强引用计数
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ........
        mPendingStrongDerefs.push(obj);
        break;
```

```
    case BR_INCREFS://BR_INCREFS协议来请求增加对应的Binder本地对象的弱引用计数
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        refs->incWeak(mProcess.get());
        mOut.writeInt32(BC_INCREFS_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_DECREFS://用BR_DECREFS协议来请求减少对应的Binder本地对象的弱引用计数
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ........
        mPendingWeakDerefs.push(refs);
        break;

    ..........

    return result;
}
```

对于BR_ACQUIRE、BR_INCREFS，Server进程会马上增加目标Binder本地对象的强引用计数和弱引用计数，并且使用BC_ACQUIRE_DONE和BC_INCREFS_DONE协议来通知Binder驱动程序，它已经增加了相应的Binder本地对象的引用计数了。

对于BR_DECREFS、BR_RELEASE，Server进程却不急于去处理，而是将它们缓存在IPCThreadState类的成员变量mPendingStrongDerefs和mPendingWeakDerefs中，等到Server进程下次使用IO控制命令BINDER_WRITE_READ进入Binder驱动程序之前，再来处理它们。因为增加一个Binder本地对象的引用计数是一件紧急的事情，必须马上处理。否则该Binder本地对象就可能提前被销毁。相反，减少一个Binder本地对象的引用计数是一件不重要的事情，至多就是延迟了Binder本地对象的生命周期，这样做的好处就是可以让Server进程优先去处理其他更重要的事情。

Andorid Binder进程间通信---Binder本地对象，实体对象，引用对象，代理对象的引用计数,布布扣,bubuko.com

Andorid Binder进程间通信---Binder本地对象，实体对象，引用对象，代理对象的引用计数

标签：des android class 代码 style int