内容目录

[clearCallingIdentity/restoreCallingIdentity 1](#__RefHeading___Toc7917_695259283)

[binderDied 1](#__RefHeading___Toc7919_695259283)

[oneway 2](#__RefHeading___Toc7921_695259283)

[一 binder调用 overview 2](#__RefHeading___Toc1045_894002420)

[1.1 C程序示例 2](#__RefHeading___Toc1047_894002420)

[1.1.1 框架分析 2](#__RefHeading___Toc1049_894002420)

[1.1.2 编写程序 3](#__RefHeading___Toc1051_894002420)

[1.2 驱动情景分析 3](#__RefHeading___Toc1053_894002420)

[1.2.1数据结构浅析 3](#__RefHeading___Toc1055_894002420)

[1.2.2 打印数据交互过程 3](#__RefHeading___Toc1057_894002420)

[1.2.3 服务注册过程概述\_分析 4](#__RefHeading___Toc1059_894002420)

[1.2.4 服务获取过程、使用过程、transaction机制、多线程 7](#__RefHeading___Toc1205_571979078)

[1.3 C++实现binder 7](#__RefHeading___Toc1207_571979078)

[1.3.1 coding 7](#__RefHeading___Toc2463_1209659968)

[1.3.2 编译调试 7](#__RefHeading___Toc2465_1209659968)

[1.3.3 Binder系统\_C++实现2\_内部机制\_回顾关键点.jpg 8](#__RefHeading___Toc2467_1209659968)

[1.3.4 Binder系统\_C++实现3\_内部机制\_代理类BpXXX分析.jpg 8](#__RefHeading___Toc2469_1209659968)

[1.3.5 Binder系统\_C++实现4\_内部机制\_数据传输.jpg 8](#__RefHeading___Toc2471_1209659968)

[1.3.6 Binder系统\_C++实现5\_内部机制\_添加服务.jpg 9](#__RefHeading___Toc2473_1209659968)

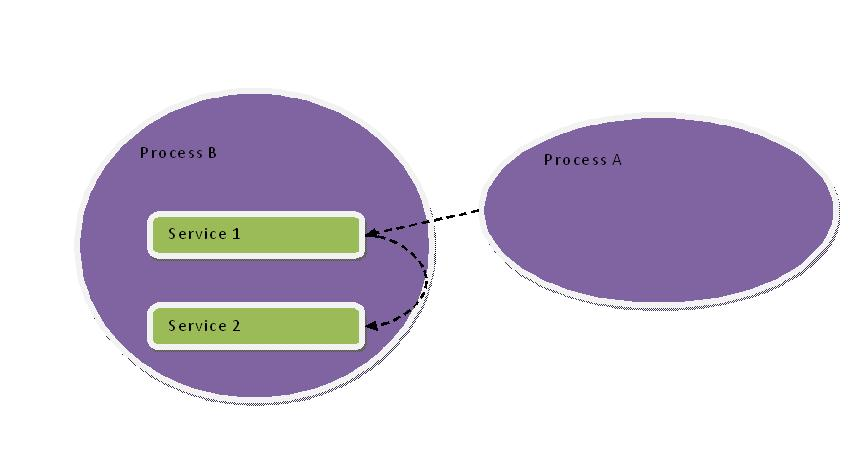
[1.4 JAVA实现binder 9](#__RefHeading___Toc2475_1209659968)

[1.4.1 Binder系统\_JAVA实现2\_分层.jpg 9](#__RefHeading___Toc2477_1209659968)

[1.4.2 Binder系统\_JAVA实现3\_内部机制.jpg 10](#__RefHeading___Toc2479_1209659968)

### clearCallingIdentity/restoreCallingIdentity

客户端A调用服务端B的服务，然后B又调用自己的服务或者方法

1. process B在被process A IPC调用时, process B需知道process A的UID和PID，来检查process A的访问权限，此时mCallingUid和mCallingPid保存的是process A的UID和PID。

2. 在IPC远程调用process B的过程中，process B的方法调用了同进程中的service的接口，process B既是调用方也是被调用方，虽然这个过程比较无聊，但是鉴于IPC过程的不透明性，因此process B仍然需要进行权限检测。

在B进程自己调用自己资源的时候可通过如下两个方法

public static final native long clearCallingIdentity();

public static final native void restoreCallingIdentity(long token);

先将A的UID、PID进行备份并且替换B的UID、PID，开始操作B的资源

token = clearCallingIdentity();

调用完成后，再恢复成A的UID、PID

### binderDied

**1. 原理**

当Binder非正常消亡的时候，会导致远程调用失败，这样客户端功能就会受到影响。

解决：给Binder设置一个死亡代理，当Binder死亡时，我们就会收到通知，这个时候可以重新发起连接。

**2. 制作**

**2.1 前期准备**

客户端：MainActivity.java

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

createService();

}

/\*连接Service端，获取mIBookManger\*/

private void createService(){

ServiceConnection connection = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

//初始化mIBookManger

mIBookManager = IBookManager.Stub.asInterface(service);

}

@Override

public void onServiceDisconnected(ComponentName name) {

}

};

Intent intent = new Intent(this,BookService.class);

bindService(intent,connection,BIND\_AUTO\_CREATE);

}

Service端：BookService.java

**2.2使用**

创建IBinder.DeathRecipient接口，重写其中的binderDied()，当Binder死亡时候，回调该方法。

private IBinder.DeathRecipient mDeathRecipient = new IBinder.DeathRecipient() {

@Override

public void binderDied() {

if (mIBookManager != null){

//解除绑定当前接口

mIBookManager.asBinder().unlinkToDeath(mDeathRecipient,0);

}

mIBookManager = null;

createService();

}

};

当然解除绑定之后，还需要绑定接口

/\*连接Service端，获取mIBookManger\*/

private void createService(){

...(连接之前的代码)

try {

//绑定接口,等待回调

mIBookManager.asBinder().linkToDeath(mDeathRecipient,0);

} catch (RemoteException e) {

e.printStackTrace();

}

}

### oneway

oneway 关键字用于修改远程调用的行为。

1.本地调用

如果 oneway 用于本地调用，则不会有任何影响，调用仍是同步调用。

2.远程调用

使用该关键字时，远程调用不会阻塞；它只是发送事务数据并立即返回。接口的实现最终接收此调用时，是以正常远程调用形式将其作为来自 Binder 线程池的常规调用进行接收。

注意，IBinder接口类中定义了一个名为FLAG\_ONEWAY的整型变量，该变量的意义非常重要。当客户端利用Binder机制发起一个跨进程的函数调用时，调用方（即客户端）一般会阻塞，直到服务端返回结果。这种方式和普通的函数调用是一样的。但是在调用Binder函数时，如果指明了FLAG\_ONEWAY标志，则调用方只要把请求发送到Binder驱动即可返回，而不用等待服务端的结果，这就是一种所谓的非阻塞方式。在Native层中，涉及的Binder调用基本都是阻塞的，但是在Java层的framework中，使用FLAG\_ONEWAY进行Binder调用的情况非常多，以后经常会碰到。

思考：使用FLAG\_ONEWAY进行函数调用的程序在设计上有什么特点？这里简单分析一下：对于使用FLAG\_ONEWAY的函数来说，客户端仅向服务端发出了请求，但是并不能确定服务端是否处理了该请求。所以，客户端一般会向服务端注册一个回调（同样是跨进程的Binder调用），一旦服务端处理了该请求，就会调用此回调函数来通知客户端处理结果。当然，这种回调函数也大多采用FLAG\_ONEWAY的方式。

# 一 binder调用 overview

## 1.1 C程序示例

git clone https://github.com/weidongshan/APP\_0003\_Binder\_C\_App.git

### 1.1.1 框架分析

frameworks/native/cmds/servicemanager/bctest.c

注册服务的过程:

a. binder\_open

b. binder\_call(bs, &msg, &reply, 0, SVC\_MGR\_ADD\_SERVICE)

// 含有服务的名字

// 它会含有servicemanager回复的数据

// 0表示servicemanager

// code: 表示要调用servicemanager中的"addservice函数"

获取服务的过程:

a. binder\_open

b. binder\_call(bs, &msg, &reply, target, SVC\_MGR\_CHECK\_SERVICE)

// 含有服务的名字

// 它会含有servicemanager回复的数据, 表示提供服务的进程

// 0表示servicemanager

// code: 表示要调用servicemanager中的"getservice函数"

Binder系统\_C程序框架分析了binder\_call函数对binder\_io的解析处理，交给binder\_write\_read的write\_buffer/read\_buffer。

### 1.1.2 编写程序

服务端：test\_server.c

客服端：test\_client.c

## 1.2 驱动情景分析

git clone https://github.com/weidongshan/DRV\_0003\_Binder.git

### 1.2.1数据结构浅析

test\_client.c

handle = svcmgr\_lookup(bs, svcmgr, "goodbye");

handle = svcmgr\_lookup(bs, svcmgr, "hello");

handle是进程A对进程B提供的服务的引用。

Binder系统\_驱动情景分析1\_IPC数据交互过程.jpg

### 1.2.2 打印数据交互过程

应用程序，BC\_TRANSACTION、BC\_REPLY写，BR\_TRANSACTION、BR\_REPLY读。只有这四个涉及到进程间的交互，其他的BC和BR是和驱动之间的交互用于改变和报告状态。

res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);

drivers/android/binder.c

static long binder\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg)

switch (cmd) {

case BINDER\_WRITE\_READ:

ret = binder\_ioctl\_write\_read(filp, cmd, arg, thread);

case BC\_TRANSACTION:

binder\_transaction(proc, thread, &tr, cmd == BC\_REPLY, 0);

}

static int binder\_ioctl\_write\_read(struct file \*filp,

unsigned int cmd, unsigned long arg,

struct binder\_thread \*thread) {

ret = binder\_thread\_write(proc, thread, bwr.write\_buffer, bwr.write\_size, &bwr.write\_consumed);

ret = binder\_thread\_read(proc, thread, bwr.read\_buffer, bwr.read\_size, &bwr.read\_consumed, filp->f\_flags & O\_NONBLOCK);

}

static int binder\_thread\_write(struct binder\_proc \*proc,

struct binder\_thread \*thread,

binder\_uintptr\_t binder\_buffer, size\_t size,

binder\_size\_t \*consumed) {

/\*print info: proc'name, proc id, thread id, cmd'name\*/

**printk**("%s (%d, %d), %s : %s\n", proc-tsk->comm, proc->pid, thread->pid, \_\_FUNCTION\_\_, cmd\_name(cmd));//cmd\_name来自客户端APP binder.c

switch (cmd) {}

}

static int binder\_thread\_read(struct binder\_proc \*proc,

struct binder\_thread \*thread,

binder\_uintptr\_t binder\_buffer, size\_t size,

binder\_size\_t \*consumed, int non\_block) {

/\*print info: proc'name, proc id, thread id, cmd'name\*/

**printk**("%s (%d, %d), %s : %s\n", proc-tsk->comm, proc->pid, thread->pid, \_\_FUNCTION\_\_, cmd\_name(BR\_NOOP));

if (put\_user(BR\_NOOP, (uint32\_t \_\_user \*)ptr))//在所有的put\_user前面打印

**printk**("%s (%d, %d), %s , print data :\n", proc-tsk->comm, proc->pid, thread->pid, \_\_FUNCTION\_\_);

hexdump(t->buffer->data, tr.data\_size);

}

static void binder\_transaction(struct binder\_proc \*proc,

struct binder\_thread \*thread,

struct binder\_transaction\_data \*tr, int reply,

binder\_size\_t extra\_buffers\_size) {

**binder\_debug**(BINDER\_DEBUG\_TRANSACTION,

"%d:%d BC\_REPLY %d -> %d:%d, data %016llx-%016llx size %lld-%lld-%lld\n",

proc->pid, thread->pid, t->debug\_id,

target\_proc->pid, target\_thread->pid,

(u64)tr->data.ptr.buffer,

(u64)tr->data.ptr.offsets,

(u64)tr->data\_size, (u64)tr->offsets\_size,

(u64)extra\_buffers\_size);

else

**binder\_debug**(BINDER\_DEBUG\_TRANSACTION,

"%d:%d BC\_TRANSACTION %d -> %d - node %d, data %016llx-%016llx size %lld-%lld-%lld\n",

proc->pid, thread->pid, t->debug\_id,

target\_proc->pid, target\_node->debug\_id,

(u64)tr->data.ptr.buffer,

(u64)tr->data.ptr.offsets,

(u64)tr->data\_size, (u64)tr->offsets\_size,

(u64)extra\_buffers\_size);

。。。

if (copy\_from\_user(t->buffer->data, (const void \_\_user \*)(uintptr\_t)

tr->data.ptr.buffer, tr->data\_size)) {

binder\_user\_error("%d:%d got transaction with invalid data ptr\n",

proc->pid, thread->pid);

return\_error = BR\_FAILED\_REPLY;

return\_error\_param = -EFAULT;

return\_error\_line = \_\_LINE\_\_;

goto err\_copy\_data\_failed;

}

/\* print data\*/

**printk**("%s (%d, %d), %s , print data :\n", proc-tsk->comm, proc->pid, thread->pid, \_\_FUNCTION\_\_);

hexdump(t->buffer->data, tr->data\_size);//hexdump来自客户端APP binder.c

if (copy\_from\_user(offp, (const void \_\_user \*)(uintptr\_t)

tr->data.ptr.offsets, tr->offsets\_size)) {

binder\_user\_error("%d:%d got transaction with invalid offsets ptr\n",

proc->pid, thread->pid);

return\_error = BR\_FAILED\_REPLY;

return\_error\_param = -EFAULT;

return\_error\_line = \_\_LINE\_\_;

goto err\_copy\_data\_failed;

}

}

### 1.2.3 服务注册过程概述\_分析

./service\_manager &

[ 32.566620] service\_manager (1362, 1362), binder\_thread\_write : BC\_ENTER\_LOOPER

[ 32.566712] service\_manager (1362, 1362), binder\_thread\_read : BR\_NOOP

//往下就休眠了，等待其他服务向他注册消息

./test\_server &

[ 38.320197] test\_server (1363, 1363), binder\_thread\_write : BC\_TRANSACTION

[ 38.320284] binder: 1363:1363 BC\_TRANSACTION 2 -> 1362 - node 1, data beca6a5c-beca6a4c size 96-4

[ 38.320383] test\_server (1363, 1363), binder\_transaction , print data :

[ 38.320454] 0000: 00 . 00 . 00 . 00 . 1a . 00 . 00 . 00 . 61 a 00 . 6e n 00 . 64 d 00 . 72 r 00 . – bio\_put\_string16\_x(&msg, SVC\_MGR\_NAME);

[ 38.329064] 0016: 6f o 00 . 69 i 00 . 64 d 00 . 2e . 00 . 6f o 00 . 73 s 00 . 2e . 00 . 49 I 00 .

[ 38.337917] 0032: 53 S 00 . 65 e 00 . 72 r 00 . 76 v 00 . 69 i 00 . 63 c 00 . 65 e 00 . 4d M 00 .

[ 38.346771] 0048: 61 a 00 . 6e n 00 . 61 a 00 . 67 g 00 . 65 e 00 . 72 r 00 . 00 . 00 . 00 . 00 .

[ 38.355646] 0064: 05 . 00 . 00 . 00 . 68 h 00 . 65 e 00 . 6c l 00 . 6c l 00 . 6f o 00 . 00 . 00 . – bio\_put\_string16\_x(&msg, name);

[ 38.364475] 0080: 85 . 2a \* 62 b 73 s 7f . 01 . 00 . 00 . e0 . 88 . 00 . 00 . 00 . 00 . 00 . 00 . – bio\_put\_obj(&msg, ptr); obj, type, binder, cookie

[ 38.373350] test\_server (1363, 1363), binder\_thread\_read : BR\_NOOP

[ 38.379521] service\_manager (1362, 1362), binder\_thread\_read : BR\_TRANSACTION

[ 38.386633] service\_manager (1362, 1362), binder\_thread\_read , print data :

[ 38.393567] 0000: 00 . 00 . 00 . 00 . 1a . 00 . 00 . 00 . 61 a 00 . 6e n 00 . 64 d 00 . 72 r 00 .

[ 38.402410] 0016: 6f o 00 . 69 i 00 . 64 d 00 . 2e . 00 . 6f o 00 . 73 s 00 . 2e . 00 . 49 I 00 .

[ 38.411263] 0032: 53 S 00 . 65 e 00 . 72 r 00 . 76 v 00 . 69 i 00 . 63 c 00 . 65 e 00 . 4d M 00 .

[ 38.420127] 0048: 61 a 00 . 6e n 00 . 61 a 00 . 67 g 00 . 65 e 00 . 72 r 00 . 00 . 00 . 00 . 00 .

[ 38.428971] 0064: 05 . 00 . 00 . 00 . 68 h 00 . 65 e 00 . 6c l 00 . 6c l 00 . 6f o 00 . 00 . 00 .

[ 38.437824] 0080: 85 . 2a \* 68 h 73 s 7f . 01 . 00 . 00 . 01 . 00 . 00 . 00 . 00 . 00 . 00 . 00 .

[ 38.447188] test\_server (1363, 1363), binder\_thread\_read : BR\_INCREFS

[ 38.453114] test\_server (1363, 1363), binder\_thread\_read : BR\_ACQUIRE

[ 38.459748] test\_server (1363, 1363), binder\_thread\_read : BR\_TRANSACTION\_COMPLETE

[ 38.467270] service\_manager (1362, 1362), binder\_thread\_write : BC\_ACQUIRE

[ 38.474004] test\_server (1363, 1363), binder\_thread\_read : BR\_NOOP

[ 38.480122] service\_manager (1362, 1362), binder\_thread\_write : BC\_REQUEST\_DEATH\_NOTIFICATION

[ 38.488626] service\_manager (1362, 1362), binder\_thread\_write : BC\_FREE\_BUFFER

[ 38.495828] service\_manager (1362, 1362), binder\_thread\_write : BC\_REPLY

[ 38.502538] binder: 1362:1362 BC\_REPLY 5 -> 1363:1363, data bed9fa4c-bed9fa3c size 4-0

[ 38.510404] service\_manager (1362, 1362), binder\_transaction , print data :

[ 38.517345] 0000: 00 . 00 . 00 . 00 .

[ 38.520994] test\_server (1363, 1363), binder\_thread\_read : BR\_REPLY

[ 38.527250] test\_server (1363, 1363), binder\_thread\_read , print data :

[ 38.540003] 0000: 00 . 00 . 00 . 00 .

[ 38.543642] service\_manager (1362, 1362), binder\_thread\_read : BR\_NOOP

[ 38.550253] test\_server (1363, 1363), binder\_thread\_write : BC\_FREE\_BUFFER

[ 38.557063] service\_manager (1362, 1362), binder\_thread\_read : BR\_TRANSACTION\_COMPLETE

[ 38.566481] service\_manager (1362, 1362), binder\_thread\_read : BR\_NOOP

[ 38.571459] test\_server (1363, 1363), binder\_thread\_write : BC\_ENTER\_LOOPER

[ 38.578368] test\_server (1363, 1363), binder\_thread\_read : BR\_NOOP

svcmgr: add\_service('hello'), handle = 1

ret = svcmgr\_publish(bs, svcmgr, "hello", hello\_service\_handler);

int svcmgr\_publish(struct binder\_state \*bs, uint32\_t target, const char \*name, void \*ptr)

{

int status;

unsigned iodata[512/4];

struct binder\_io msg, reply;

**//1. 构造数据**

**//a. 构造binder\_io**

//下面函数对应到用户空间binder.c里面，

bio\_init(&msg, iodata, sizeof(iodata), 4);//前十六个字节空出来offs

bio\_put\_uint32(&msg, 0); //32对应4字节，放入了00 00 00 00data->

bio\_put\_string16\_x(&msg, SVC\_MGR\_NAME);//"android.os.IServiceManager"，放入长度len=0x1A(4字节)，字符串用两个字节存放一个字符6100 6e00 6400 ...

bio\_put\_string16\_x(&msg, name);//"hello"，05000000 6800 6500…-<data

bio\_put\_obj(&msg, ptr);//前十六个字节，有四个位置可以一一指向构造出来的 flat\_binder\_object

{

struct flat\_binder\_object \*obj;

obj->flags = 0x7f | FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS;

obj->type = BINDER\_TYPE\_BINDER;//实体或者是引用，只有test\_server服务提供者能传实体，client、service\_manager只能传引用

obj->binder = (uintptr\_t)ptr;//传入的handler: flag实体对应实体(处理函数的地址)，应用对应引用

obj->cookie = 0;

}

if (binder\_call(bs, &msg, &reply, target, SVC\_MGR\_ADD\_SERVICE))

int binder\_call(struct binder\_state \*bs,

struct binder\_io \*msg,

struct binder\_io \*reply,

uint32\_t target, uint32\_t code)

{

int res;

struct binder\_write\_read bwr;

struct {

uint32\_t cmd;

struct binder\_transaction\_data txn;

} \_\_attribute\_\_((packed)) writebuf;

unsigned readbuf[32];

if (msg->flags & BIO\_F\_OVERFLOW) {

fprintf(stderr,"binder: txn buffer overflow\n");

goto fail;

}

**//b. 转为 binder\_transaction\_data**

writebuf.cmd = BC\_TRANSACTION;//四大类型之一

writebuf.txn.target.handle = target;//0-servicemanager

writebuf.txn.code = code;//SVC\_MGR\_ADD\_SERVICE-注册服务

writebuf.txn.flags = 0;

writebuf.txn.data\_size = msg->data - msg->data0;///放入msg数据

writebuf.txn.offsets\_size = ((char\*) msg->offs) - ((char\*) msg->offs0);//data-><-data 大小

writebuf.txn.data.ptr.buffer = (uintptr\_t)msg->data0;//data

writebuf.txn.data.ptr.offsets = (uintptr\_t)msg->offs0;//offs

**//c. 放入 binder\_write\_read**

bwr.write\_size = sizeof(writebuf);

bwr.write\_consumed = 0;

bwr.write\_buffer = (uintptr\_t) &writebuf;

hexdump(msg->data0, msg->data - msg->data0);

for (;;) {

bwr.read\_size = sizeof(readbuf);

bwr.read\_consumed = 0;

bwr.read\_buffer = (uintptr\_t) readbuf;

**//2. 发送数据ioctl**

res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);

}

return -1;

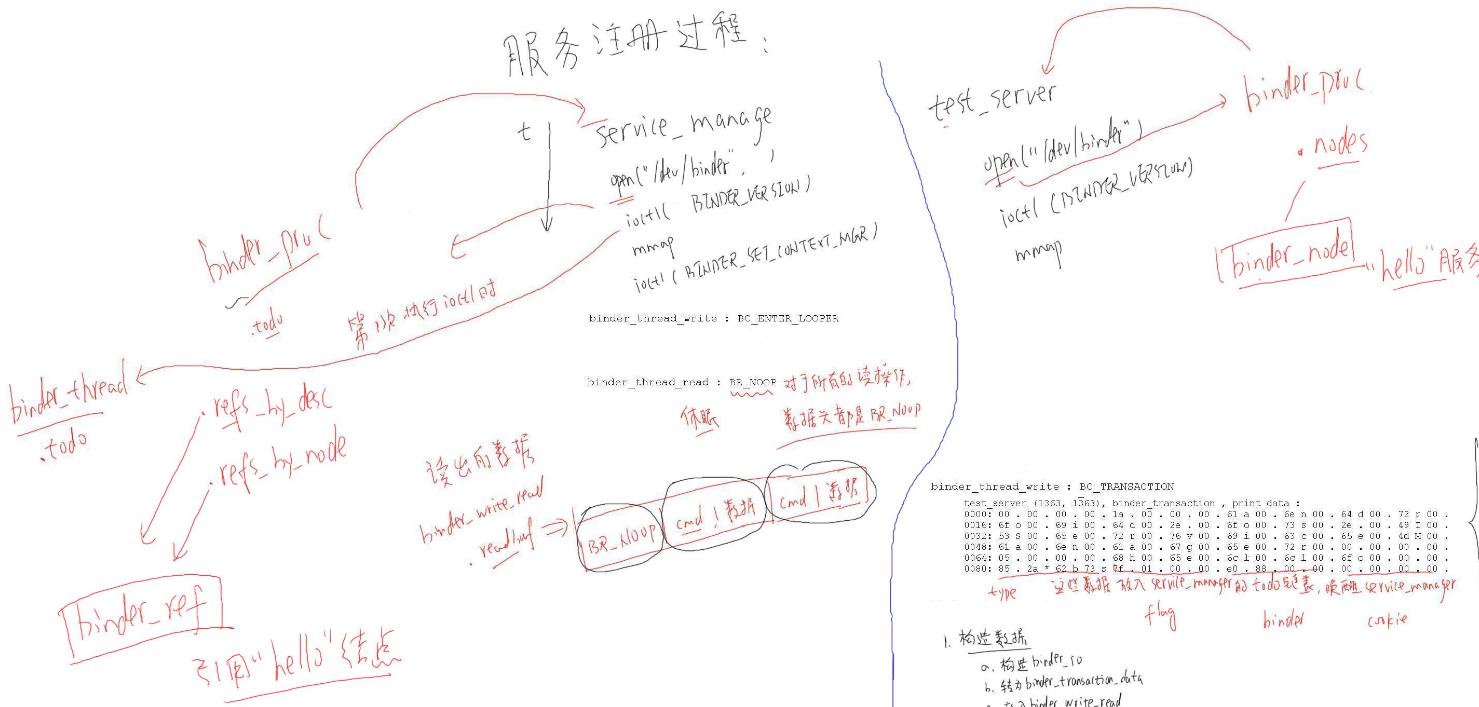
status = bio\_get\_uint32(&reply);

binder\_done(bs, &msg, &reply);

return status;

}

3. 进入驱动，binder\_ioctl把数据访日service\_manager进程的todo链表，并唤醒它



a. 根据handle找到目的进程service\_manager

b. 把数据copy\_from\_user放到service\_manager的mmap的空间

c. 处理offset数据 flat\_binder\_object

a. 构造binder\_node给test\_server，node = binder\_new\_node(proc, fp->binder, fp->cookie);

b. 构造binder\_ref给service\_manager

c. 增加引用计数

static void binder\_transaction(struct binder\_proc \*proc,

struct binder\_thread \*thread,

struct binder\_transaction\_data \*tr, int reply)

{

copy\_from\_user();

struct flat\_binder\_object \*fp;

case BINDER\_TYPE\_BINDER:

case BINDER\_TYPE\_WEAK\_BINDER: {

node = binder\_new\_node(proc, fp->binder, fp->cookie);//构造binder\_node给test\_server

ref = binder\_get\_ref\_for\_node(target\_proc, node);//构造binder\_ref给service\_manager

if (fp->type == BINDER\_TYPE\_BINDER)

fp->type = BINDER\_TYPE\_HANDLE;//实体改成引用

}

fp->handle = ref->desc;//修改成引用号1，即第一个引用，service\_manager通过1(desc)找到第一个binder\_ref，再找到binder\_node

struct flat\_binder\_object {

\_\_u32 type;

\_\_u32 flags;

union {

binder\_uintptr\_t binder;

\_\_u32 handle;

};

binder\_uintptr\_t cookie;

};

binder\_inc\_ref(ref, fp->type == BINDER\_TYPE\_HANDLE, &thread->todo);//增加引用计数

}

struct binder\_proc {

struct hlist\_node proc\_node;

struct rb\_root threads;

struct rb\_root nodes;

struct rb\_root refs\_by\_desc;//通过handle找到binder\_ref

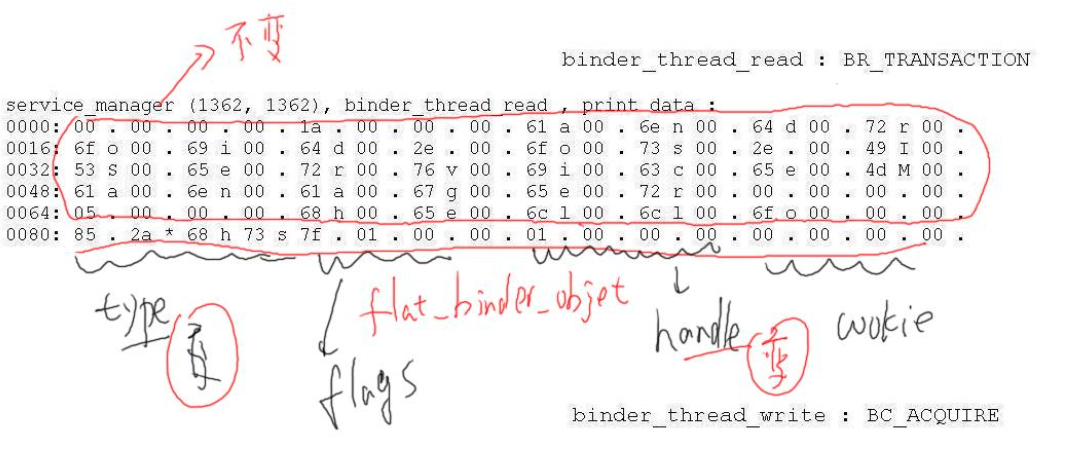
struct rb\_root refs\_by\_node;//通过binder\_node找到binder\_ref

。。。

};

binder\_ref.desc //handle=1，这个进程第一个binder\_ref结构体

binder\_ref.node //指向test\_server的binder\_node



service\_manage.c将驱动修改后的数据取出来，do\_add\_service，分配内存，服务引用1，服务名称hello，加入死亡通知，然后bio\_put\_uint32(reply, 0)，这样test\_server可以读到四个0。

### 1.2.4 服务获取过程、使用过程、transaction机制、多线程

Binder系统\_驱动情景分析3\_服务获取过程.jpg，讲的比1.2.2相对少很多，因为代码流程类似。

Binder系统\_驱动情景分析4\_服务使用过程.jpg，传进来的hello\_service\_handler、goodbye\_service\_handler存放在struct binder\_node->ptr里面

Binder系统\_驱动情景分析7\_binder\_server的多线程，test\_server.c设置最大线程binder\_set\_maxthreads，以及处理BR\_SPAWN\_LOOPER。

## 1.3 C++实现binder

本章第三小节非常重要。

### 1.3.1 coding

Binder系统\_C++实现1\_编程.jpg

# wuj @ wuj in ~/workspace/studying/FramworkFromAndroidOin2018/Binder/APP\_0004\_Binder\_CPP\_App on git:b1b1c13 o [14:44:36]

$ ll

总用量 24K

-rw-rw-r-- 1 wuj wuj 1.2K 5月 11 14:43 BnHelloService.cpp //Binder native

-rw-rw-r-- 1 wuj wuj 777 5月 11 14:43 BpHelloService.cpp //Bindr proxy

-rw-rw-r-- 1 wuj wuj 984 5月 11 14:43 IHelloService.h

-rw-rw-r-- 1 wuj wuj 48 5月 11 14:43 README.md

-rw-rw-r-- 1 wuj wuj 1.3K 5月 11 14:43 test\_client.cpp

-rw-rw-r-- 1 wuj wuj 776 5月 11 14:43 test\_server.cpp

### 1.3.2 编译调试

**1) 选定代码版本**

$ git checkout v2

之前的 HEAD 位置是 b1b1c13... v1, has not been tested, can not run

HEAD 目前位于 8565db2... v2, can run

**2) 在工作环境下编译和push**

wuj@wuj:~/workspace/codes/M01/O-M01-MASTER/LINUX/android/frameworks/testing/APP\_0004\_Binder\_CPP\_App$ mmm .

# wuj @ wuj in ~/workspace/codes/M01/O-M01-MASTER/LINUX/android/out/target/product/M01\_AE/system/bin [15:46:40] C:1

$ adb push test\_server /system/bin

test\_server: 1 file pushed. 1.3 MB/s (24996 bytes in 0.019s)

# wuj @ wuj in ~/workspace/codes/M01/O-M01-MASTER/LINUX/android/out/target/product/M01\_AE/system/bin [15:46:49]

$ adb push test\_client /system/bin

test\_client: 1 file pushed. 3.4 MB/s (20684 bytes in 0.006s)

**3) 调试输出**

M01\_AE:/system/bin # ./test\_server &

[1] 10308

M01\_AE:/system/bin # ./test\_client hello

M01\_AE:/system/bin # ./test\_client hello wujian

# wuj @ wuj in ~/tmp [15:47:31]

$ adb shell logcat | grep HelloService

05-11 15:48:16.990 10308 10313 I HelloService: say hello : 0

05-11 15:48:16.991 11080 11080 I HelloService: client call sayhello

05-11 15:48:23.461 10308 10313 I HelloService: say hello to wujian : 0

05-11 15:48:23.462 11452 11452 I HelloService: client call sayhello\_to, cnt = 1

### 1.3.3 Binder系统\_C++实现2\_内部机制\_回顾关键点.jpg

<https://www.jianshu.com/p/95e61dcaa1fe>

Client要去获得Server的一个API服务，这里 Client通过IDemoServer的BinderProxy的handle值能够轻松找到 binder\_ref，然后再通过 binder\_ref的node可以找到binder\_node，而这个binder\_node就是Server端进程中的IDemoServer的binder\_node，接着通过binder\_node又可以很轻松的找到Server进程 binder\_proc，最后将transaction插入到Server进程binder\_proc中，然后binder驱动唤醒Server进程。

跟视频流程一致，wds的总结:

 server注册服务时，对每个服务都提供不同的ptr/cookie

在驱动程序里对每个服务都构造一个binder\_node，它也含有ptr/cookie

 client使用服务前要先getService: 会在驱动程序里对该服务构造一个binder\_ref

binder\_ref含有desc，node成员(指向服务的binder\_node节点)，desc是整数，node指向对应服务的binder\_node

 使用服务时，client构造数据，调用ioctl: 数据里含有handle

 驱动程序根据handle找到binder\_ref(desc==handle)，找到binder\_node，找到binder\_proc，再找到server，从binder\_node取出ptr/cookie连同那些数据发给server

 server根据ptr/cookie知道要调用哪一个服务，．．．．

 最核心函数: ioctl

client的最核心数据是: handle

server的最核心数据是: ptr/cookie

### 1.3.4 Binder系统\_C++实现3\_内部机制\_代理类BpXXX分析.jpg

**4.1 获得BpServiceManager对象的过程:**

defaultServiceManager构造了一个BpServiceManager对象，其中它的mRemote = new BpBinder(0); // mRemote->mHandle=0

**4.2 获得BpHelloService对象的过程:**

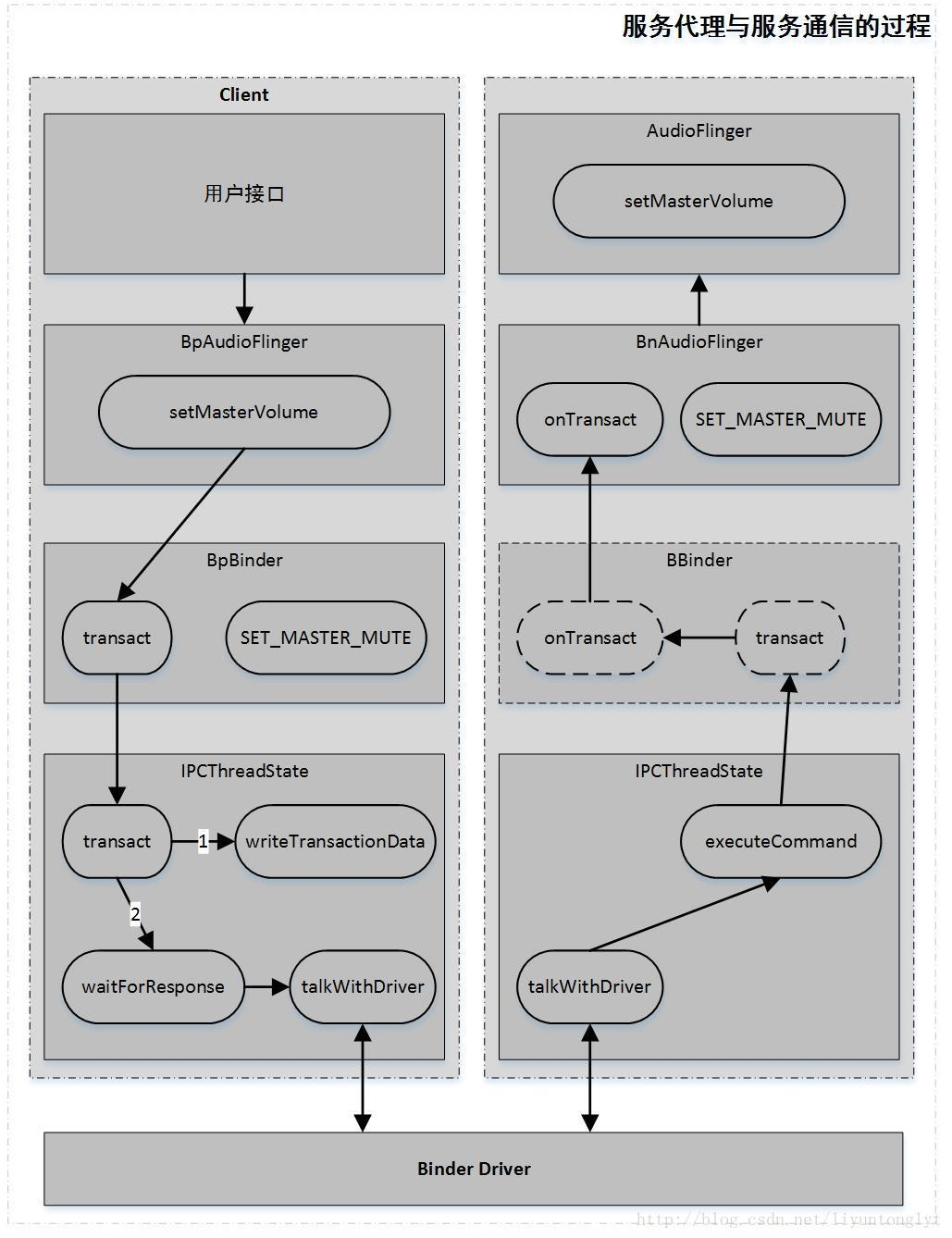
调用BpServiceManager的getService函数获得一个flat\_binder\_object，从中取出handle, 创建一个BpBinder(handle)，然后使用interface\_cast使用这个BpBinder创建一个BpHelloService对象

**4.3 代理类如何发送数据:**

ioctl，数据里含有handle，含有其他构造的参数，构造好数据之后，调用:

remote()->transact(...)

IPCThreadState::self()->transact(mHandle, code, data, reply, flags);



### 1.3.5 Binder系统\_C++实现4\_内部机制\_数据传输.jpg

test\_server.cpp

/\* 循环体 \*/

ProcessState::self()->startThreadPool();//子线程：对象每个线程有自己的IPCThreadState对象 10313，通过pthread\_setspecific/pthread\_getspecific各自维护自己的value

IPCThreadState::self()->joinThreadPool();//主线程：每个进程只有一个ProcessState 10308

M01\_AE:/system/bin # ps -AT | grep test\_server

root 10308 10308 7426 8412 1808 binder\_thread\_read f6253598 S test\_server

M01\_AE:/system/bin # ls /proc/10308/task/

10308 10313

M01\_AE:/system/bin # ps -AT | grep 10308

root 10308 10308 7426 8412 1808 binder\_thread\_read f6253598 S test\_server

root 10308 10313 7426 8412 1808 binder\_thread\_read f6253598 S Binder:10308\_1

**5.1 addService**

对于不同服务，构造flat\_binder\_object结构体，里面的.binder/.cookie对于不同的服务它的值不一样

sm->addService(String16("hello"), new BnHelloService());

data.writeStrongBinder(service); // service = new BnHelloService();

flatten\_binder(ProcessState::self(), val, this); // val = service = new BnHelloService();

flat\_binder\_object obj; // 参数 binder = val = service = new BnHelloService();

IBinder \*local = binder->localBinder(); // =this = new BnHelloService();

obj.type = BINDER\_TYPE\_BINDER;

obj.binder = reinterpret\_cast<uintptr\_t>(local->getWeakRefs());

obj.cookie = reinterpret\_cast<uintptr\_t>(local); // new BnHelloService();

**5.2 server如何分辨client想使用哪一个服务**

server收到数据里含有flat\_binder\_object结构体，

它可以根据.binder/.cookie分析client想使用哪一个服务

把.cookie转换为BnXXXX对象，然后调用它的函数:

// 根据cookie构造了一个BBinder指针, 实际上是指向某个BnXXX对象

sp<BBinder> b((BBinder\*)tr.cookie);

// 然后调用它的transact函数

error = b->transact(tr.code, buffer, &reply, tr.flags);

err = onTransact(code, data, reply, flags); // 就会调用到BnXXX里实现的onTransact

// 它就会根据code值来调用不同的函数

**5.3 怎么调用到HelloService所提供的函数**

见5.2最后一行

### 1.3.6 Binder系统\_C++实现5\_内部机制\_添加服务.jpg

## 1.4 JAVA实现binder

单独编译aidl

cp IHelloService.aidl frameworks/base/core/java/android/os/

frameworks/base/Android.mk

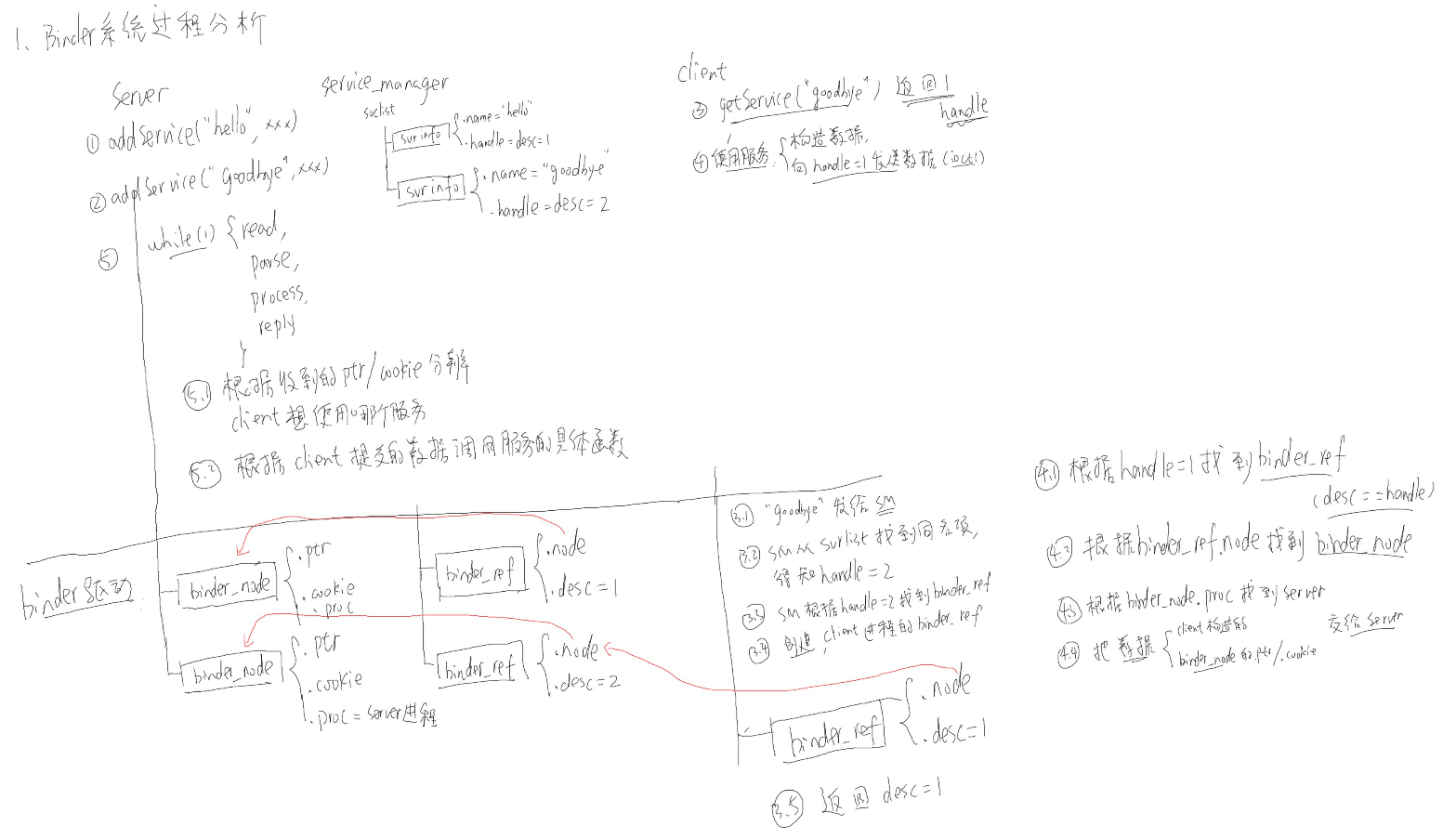
core/java/android/os/IHelloService.aidl

mmm frameworks/base/

out/target/common/obj/JAVA\_LIBRARIES/framework\_intermediates/core/java/android/os/IHelloService.java

### 1.4.1 Binder系统\_JAVA实现2\_分层.jpg

用户空间C和binder驱动交互的复习



### 1.4.2 Binder系统\_JAVA实现3\_内部机制.jpg

3. 用java实现hello服务\_测试

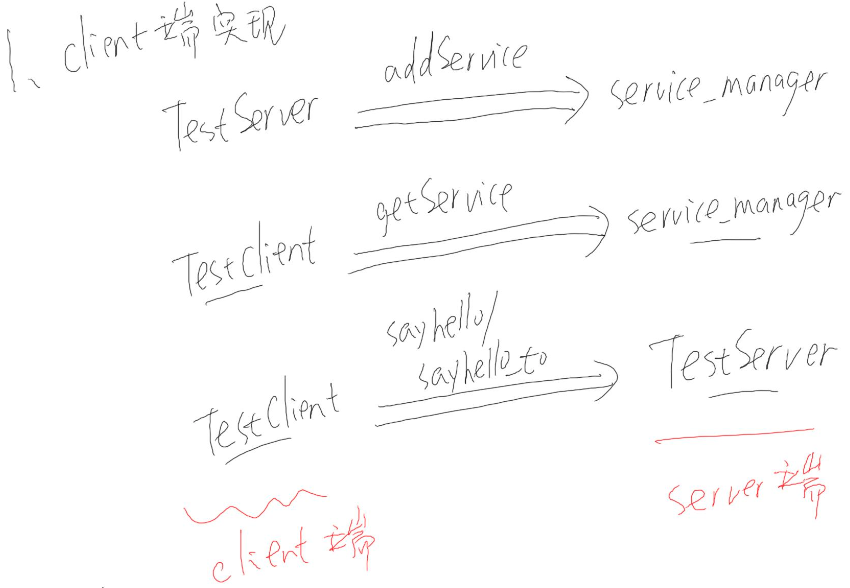
logcat TestServer:\* TestClient:\* HelloService:\* \*:S &

CLASSPATH=/mnt/android\_fs/TestServer.jar app\_process / TestServer &

CLASSPATH=/mnt/android\_fs/TestClient.jar app\_process / TestClient hello

CLASSPATH=/mnt/android\_fs/TestClient.jar app\_process / TestClient hello weidongshan

**1 client端实现**

****

**1.1 addService / getService**

APP\_0005\_Binder\_JAVA\_App/TestServer.java

ServiceManager.addService("hello", new HelloService());

frameworks/base/core/java/android/os/ServiceManager.java

public static void addService(String name, IBinder service) {

try {

getIServiceManager().addService(name, service, false);

} catch (RemoteException e) {

Log.e(TAG, "error in addService", e);

}

}

private static IServiceManager getIServiceManager() {

if (sServiceManager != null) {

return sServiceManager;

}

// Find the service manager，最终返回的是return new ServiceManagerProxy(obj);

sServiceManager = ServiceManagerNative.asInterface(Binder.allowBlocking(BinderInternal.getContextObject()));

return sServiceManager;

}

所以，最后的结果是

addService→new ServiceManagerProxy(obj).addService

getService→new ServiceManagerProxy(obj).getService

使用mRemote.transact()发送数据

**1.2 sayhello / sayhello\_to**

IHelloService.Stub.Proxy，使用mRemote.transact()发送数据

**1.3 统一使用mRemote.transact来发送数据，三要素：源、目的、数据**

先把结论上上来！

对于addService/getService，对于mRemote指向一个JavaBinderProxy对象，它的mObject指向一个C++的BpBinder对象，这个BpBinder的mHandle=0

对于sayhello/sayhello\_to，mRemote是一个JavaBinderProxy对象，它的mObject指向一个C++的BpBinder对象，这BpBinder的mHandle=1来自getService("hello")

代码分析：

**1.3.1 ServiceManagerProxy中mRemote的构造(用于addService/getService)**

ServiceManagerNative.asInterface(Binder.allowBlocking(BinderInternal.getContextObject()));

**a. BinderInternal.getContextObject()**

它是一个JNI调用，得到了一个Java BinderProxy对象, 其中mObject指向new BpBinder(0);

android\_os\_BinderInternal\_getContextObject {// android\_util\_Binder.cpp

sp<IBinder> b = ProcessState::self() -> getContextObject(NULL);

→ 返回的是getStrongProxyForHandle(0);

→ b = new BpBinder(handle);// mHandle = 0

return javaObjectForIBinder(env, b);//参数b，b = new BpBinder(0)，mHandle = 0

→ object = env->NewObject(gBinderProxyOffsets.mClass, gBinderProxyOffsets.mConstructor);//使用C代码调用NewObject来创建JAVA BinderProxy对象

env->SetLongField(object, gBinderProxyOffsets.mObject, (jlong)val.get());//设置该对象的mObject = val.get = b = new BpBinder(0)

return object;

}

**b. ServiceManagerNative.asInterface**

static public IServiceManager asInterface(IBinder obj) {// obj = BinderProxy对象

return new ServiceManagerProxy(obj);//mRemote = obj = BinderProxy对象, 其中mObject指向new BpBinder(0);

}

**1.3.2 hello服务里的mRemote如何构造**

APP\_0005\_Binder\_JAVA\_App/TestClient.java

a. IBinder binder = ServiceManager.getService("hello");//它的返回值就是一个java BinderProxy对象, 其中的mObject=new BpBinder(handle)

ServiceManagerNative.java

public IBinder getService(String name) throws RemoteException {

Parcel data = Parcel.obtain();

Parcel reply = Parcel.obtain();

data.writeInterfaceToken(IServiceManager.descriptor);

data.writeString(name);

mRemote.transact(GET\_SERVICE\_TRANSACTION, data, reply, 0);

IBinder binder = reply.readStrongBinder();

{

return nativeReadStrongBinder(mNativePtr);//frameworks/base/core/java/android/os/Parcel.java

//是一个jni调用，对应的代码

->android\_os\_Parcel\_readStrongBinder

{ //把java Parce对象转换为c++ Parcel对象，

client程序向sevice\_manager发出getService请求，

得到一个回复reply, 它里面含有flat\_binder\_object

它被封装成一个c++ Parcel对象

Parcel\* parcel = reinterpret\_cast<Parcel\*>(nativePtr);

//它会创建一个java BinderProxy对象, 其中的mObject=new BpBinder(handle)对象

return javaObjectForIBinder(env, parcel->readStrongBinder());

{

readNullableStrongBinder(&val);

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

->

{

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

{

const flat\_binder\_object\* flat = in.readObject<flat\_binder\_object>();

if (flat) {

switch (flat->hdr.type) {

case BINDER\_TYPE\_BINDER:

\*out = reinterpret\_cast<IBinder\*>(flat->cookie);

return finish\_unflatten\_binder(NULL, \*flat, in);

case BINDER\_TYPE\_HANDLE:

\*out = proc->getStrongProxyForHandle(flat->handle);//最终b = new BpBinder(handle);

return finish\_unflatten\_binder(

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

}

}

return BAD\_TYPE;

}

}

}

}

}

reply.recycle();

data.recycle();

return binder;

}

b. IHelloService svr = IHelloService.Stub.asInterface(binder);

return new IHelloService.Stub.Proxy(obj); //obj = 步骤a得到的binder，mRemote = remote;

**1.3.3** private android.os.IBinder mRemote; mRemote就是一个java BinderProxy 对象

看一下mRemote.transact()

transactNative(code, data, reply, flags);

static jboolean android\_os\_BinderProxy\_transact(JNIEnv\* env, jobject obj, jint code, jobject dataObj, jobject replyObj, jint flags)

{

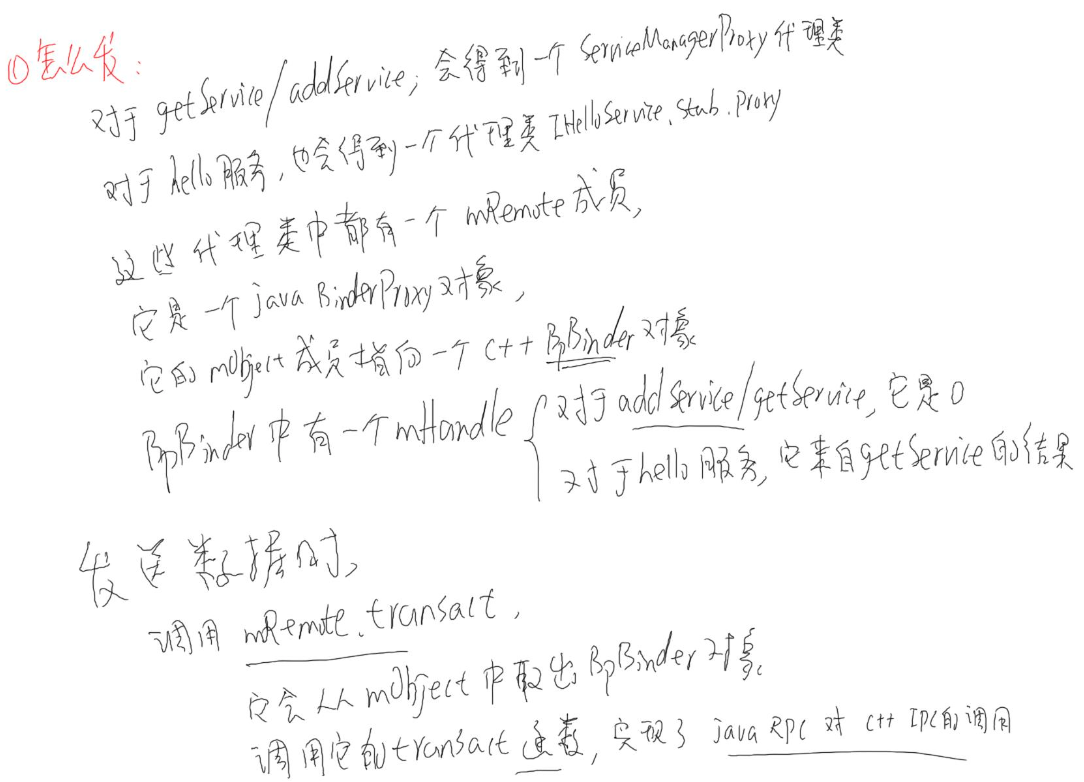
// 从java BinderProxy对象中把mObject取出, 它就是一个BpBinder对象

IBinder\* target = (IBinder\*)env->GetLongField(obj, gBinderProxyOffsets.mObject);

// 然后调用BpBinder的transact函数

status\_t err = target->transact(code, \*data, reply, flags);

}



**2 server端实现**

继上面的问题，我们总结了1、“怎么发”，现在需要解决另外两个问题2、server怎么读到数据，3、怎么调用某服务的RPC层的onTransact函数

**2.1 server怎么读到数据**

logcat TestServer:\* TestClient:\* HelloService:\* \*:S &

CLASSPATH=/mnt/android\_fs/TestServer.jar app\_process / TestServer &

CLASSPATH=/mnt/android\_fs/TestClient.jar app\_process / TestClient hello

CLASSPATH=/mnt/android\_fs/TestClient.jar app\_process / TestClient hello weidongshan

app\_process: frameworks\base\cmds\app\_process\app\_main.cpp

使用app\_process来启动server进程，会先创建子线程:

AppRuntime::onStarted()

→ proc->startThreadPool();

→ spawnPooledThread(true);

→ {

sp<Thread> t = new PoolThread(isMain);

t->run(name.string());// 它会创建子线程, 并执行threadLoop

}

→ IPCThreadState::self()->joinThreadPool(mIsMain);//frameworks/native/libs/binder/IPCThreadState.cpp

{

do {

result = getAndExecuteCommand();

→ {

result = talkWithDriver();

result = executeCommand(cmd);

→ //对于BR\_TRANSACTION数据

sp<BBinder> b((BBinder\*)tr.cookie);//cookie什么时候设置的

error = b->transact(tr.code, buffer, &reply, tr.flags);

}

} while(...)

}

**2.2 怎么调用某服务的RPC层的onTransact函数**

**a. 在addService时设置.ptr/.cookie**

a.1 new HelloService()是JAVA对象

a.2 处理数据时把.cookie转换成BBinder对象, 它是c++对象

所以: addService中肯定会把JAVA对象转换成一个BBinder派生类对象，存在.cookie里

**b. 读取到的数据里含有.ptr/ .cookie**

它会把cookie转换成BBinder对象，调用它的transact对象

**结论:**

a.1 addService会通过JNI调用c++函数:

创建一个BBinder派生类JavaBBinder对象,

它的.mObject指向JAVA对象: new HelloService()

它含有onTransact函数

把这个对象存入.cookie(最终存入binder驱动中该服务对应的binder\_node.cookie)

a.2 server进程从驱动中读到数据，里面含有.cookie

把它转换为BBinder对象，

调用它的transact函数

它会调用到派生类JavaBBinder中定义的onTransact函数

a.3 JavaBBinder中定义的onTransact函数(c++)

它通过JNI调用java Binder的execTransact方法，

然后调用Binder派生类IHelloService.Stub中定义的onTransact函数(JAVA)

a.4 IHelloService.Stub中定义的onTransact函数(JAVA):

分析数据

调用sayhello/sayhello\_to

**源码阅读：**

a.1 ServiceManager.addService("hello", new HelloService());

ServiceManagerProxy.addService:

// Parcel.java

data.writeStrongBinder(service);

// 它是一个JNI调用，对应android\_os\_Parcel\_writeStrongBinder(c++)

nativeWriteStrongBinder(mNativePtr, val); // val = service = new HelloService()

a.2 android\_os\_Parcel\_writeStrongBinder(c++)

猜测

// 它会构造一个JavaBBinder对象(c++)，.mObject=new HelloService() JAVA对象

// 然后让.cookie=JavaBBinder对象(c++)

验证

// 把Java Parcel转换为c++ Parcel

Parcel\* parcel = reinterpret\_cast<Parcel\*>(nativePtr);

// .cookie = ibinderForJavaObject(env, object)得到一个JavaBBinder对象

parcel->writeStrongBinder(ibinderForJavaObject(env, object))

a.3 ibinderForJavaObject(env, object) //object = new HelloService()

// 把一个Java对象(new HelloService())转换为c++ IBinder对象

JavaBBinderHolder\* jbh = (JavaBBinderHolder\*)env->GetLongField(obj, gBinderOffsets.mObject);

return jbh != NULL ? jbh->get(env, obj) : NULL;

-> b = new JavaBBinder(env, obj); // obj = new HelloService()

-> mObject = new HelloService()

a.4 从驱动中得过了.cookie, 它是一个JavaBBinder对象，调用它的transact函数，导致JavaBBinder对象的onTransact被调用

JavaBBinder::onTransact (调用java里的某个函数)

// mObject指向 HelloService对象

// gBinderOffsets.mExecTransact指向: java Binder类中的execTransact方法

// 调用HelloService(派生自Binder)对象中的execTransact方法

jboolean res = env->CallBooleanMethod(mObject, gBinderOffsets.mExecTransact,

code, reinterpret\_cast<jlong>(&data), reinterpret\_cast<jlong>(reply), flags);

a.5 java Binder execTransact:

res = onTransact(code, data, reply, flags);

调用HelloService中的onTransact方法(来自IHelloService.Stub)

分辨数据

调用sayhello/sayhello\_to

