Binder驱动和ServiceManager对数据的处理

/[drivers](http://androidxref.com/kernel_3.18/xref/drivers/)/[staging](http://androidxref.com/kernel_3.18/xref/drivers/staging/)/[android](http://androidxref.com/kernel_3.18/xref/drivers/staging/android/)/[binder.c](http://androidxref.com/kernel_3.18/xref/drivers/staging/android/binder.c)

static int \_\_init binder\_init(void){

int ret;

//创建名为binder的工作队列

binder\_deferred\_workqueue = create\_singlethread\_workqueue("binder");

…

ret = misc\_register(&binder\_miscdev); // 注册misc设备

return ret;

}

device\_initcall(binder\_init);//binder驱动初始化

// 注册misc设备

static struct miscdevice binder\_miscdev = {

.minor = MISC\_DYNAMIC\_MINOR,

.name = "binder",//驱动名称

.fops = &binder\_fops//驱动支持的文件操作

};

//binder驱动暴露的接口

static const struct file\_operations binder\_fops = {

.owner = THIS\_MODULE,

.poll = binder\_poll,

.unlocked\_ioctl = binder\_ioctl,

.compat\_ioctl = binder\_ioctl,

.mmap = binder\_mmap,

.open = binder\_open,

.flush = binder\_flush,

.release = binder\_release,

};

/\*上层调用的open对应binder\_open.创建binder\_proc对象,并把当前进程等信息保存到binder\_proc对象,

\*该对象管理IPC所需的各种信息;再把binder\_proc对象保存到文件指针filp,以及把binder\_proc加入到

\*全局链表binder\_procs,每个进程都有一个独立的binder\_procs;

\*/

static HLIST\_HEAD(binder\_procs);

static int binder\_open(struct inode \*nodp, struct file \*filp){

struct binder\_proc \*proc; // binder进程

proc = kzalloc(sizeof(\*proc), GFP\_KERNEL); // 为binder\_proc结构体分配kernel内存空间get\_task\_struct(current);

proc->tsk = current; //将当前线程的task保存到binder进程的tsk

INIT\_LIST\_HEAD(&proc->todo); //初始化todo列表

init\_waitqueue\_head(&proc->wait); //初始化wait队列

proc->default\_priority = task\_nice(current); //将当前进程的nice值转换为进程优先级binder\_lock(\_\_func\_\_); //同步锁，因为binder支持多线程访问binder\_stats\_created(BINDER\_STAT\_PROC); //BINDER\_PROC对象创建数加1

hlist\_add\_head(&proc->proc\_node, &binder\_procs); //将proc\_node节点添加到binder\_procs的队列的头部

proc->pid = current->group\_leader->pid;//进程的pid

INIT\_LIST\_HEAD(&proc->delivered\_death); //初始化已分发的死亡通知列表

filp->private\_data = proc; //file文件指针的private\_data变量指向binder\_proc数据

binder\_unlock(\_\_func\_\_); //释放同步锁

return 0;

}

/frameworks/native/libs/binder/IPCThreadState.cpp

IPCThreadState::talkWithDriver -> ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr)

/drivers/staging/android/binder.c

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

int ret;

struct binder\_proc \*proc = filp->private\_data;//取出进程对应的binder\_proc结构体

struct binder\_thread \*thread;

...

binder\_lock(\_\_func\_\_);

//从binder\_proc中查找binder\_thread,如果当前线程已经加入到proc的线程队列则直接返回,

//如果不存在则创建binder\_thread,并将当前线程添加到当前的proc

thread = binder\_get\_thread(proc);取出

switch (cmd) {

case BINDER\_WRITE\_READ:

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

...

return ret;

}

static int binder\_ioctl\_write\_read(struct file \*filp,unsigned int cmd, unsigned long arg,struct binder\_thread \*thread){

int ret = 0;

struct binder\_proc \*proc = filp->private\_data;

unsigned int size = \_IOC\_SIZE(cmd);

void \_\_user \*ubuf = (void \_\_user \*)arg;

struct binder\_write\_read bwr;

…

//从用户空间复制数据,这里把数据复制到驱动中的bwr结构体中

if (copy\_from\_user(&bwr, ubuf, sizeof(bwr))) {}

if (bwr.write\_size > 0) {//如果需要写数据就调用binder\_thread\_write写

ret = binder\_thread\_write(proc, thread,bwr.write\_buffer, bwr.write\_size,

&bwr.write\_consumed);

}

if (bwr.read\_size > 0) {//如果请求读数据就调用binder\_thread\_read读

ret = binder\_thread\_read(proc, thread, bwr.read\_buffer, bwr.read\_size,

&bwr.read\_consumed,filp->f\_flags & O\_NONBLOCK);

f (!list\_empty(&proc->todo))

//如果这个进程中的有需要处理的事务就唤醒该进程,一开始是为空的所

//以走不到这里

wake\_up\_interruptible(&proc->wait);

}

if (copy\_to\_user(ubuf, &bwr, sizeof(bwr))) {}

…

}

copy\_from\_user 函数的目的是从用户空间拷贝数据到内核空间.失败返回没有 被拷贝的字节数,成功返回 0.

不管是读取还是写入数据Binder只是个中间人的作用,真正处理请求的是Binder Client和Binder Server通信双方.

参数含义:

proc:调用者进程.

Thread:调用者线程

binder\_buffer:Binder Client要写入的数据在talkWithDriver中被设置为bwr.write\_buffer = (uintptr\_t)mOut.data();也就是writeTransactionData时对mOut写入的数据,有两个部分 mOut.writeInt32(cmd);//写入命令BC\_TRANSACTION

mOut.write(&tr, sizeof(tr));/ /写入binder\_transaction\_data结构体数据

size: bwr.write\_size写入数据的大小

consumed: bwr.write\_consumed已经被Binder驱动消耗的数据大小

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){

uint32\_t cmd;

void \_\_user \*buffer = (void \_\_user \*)(uintptr\_t)binder\_buffer;

void \_\_user \*ptr = buffer + \*consumed;//数据的起点略过已经处理的部分

void \_\_user \*end = buffer + size;//数据的终点

//可能不止一个cmd和其对应的数据,所以循环取出这些命令和数据

while (ptr < end && thread->return\_error == BR\_OK) {

if (get\_user(cmd, (uint32\_t \_\_user \*)ptr))//取出一个cmd

return -EFAULT;

ptr += sizeof(uint32\_t);//跳过cmd占据的空间,下面取出的就是cmd对应的数据了

switch (cmd) {

..

case BC\_TRANSACTION://这里是这个命令

case BC\_REPLY: {//当SM处理完请求把结果发给驱动的时候会有使用这个命令

struct binder\_transaction\_data tr;

//把数据复制到tr结构体中,与用户进程传进来的数据结构相同

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

return -EFAULT;

//跳过数据区,意味着一个命令和这个命令对应的数据区已经取完,接着去取下一个

ptr += sizeof(tr);

binder\_transaction(proc, thread, &tr, cmd == BC\_REPLY);//对取出的数据处理

break;

}

}

}

}

// BC\_TRANSACTION和BC\_REPLY命令都会由这个函数处理,要区分开,这里reply为false

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

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

下面对这个函数分段分析,先看一下几个重要的成员

struct binder\_transaction \*t;//表示一个transaction事务

/\*\*tcomplete代表一个未完成的操作,一个transaction会涉及到两个进程A和B,当A向B发送了

\*个请求,B需要一段时间来执行, 这个时候对于A来说就是一个未完成的操作,直到B返回了

\*结果,Binder驱动会再次启动A继续执行\*/

struct binder\_work \*tcomplete;//当前线程一个未完成的transaction

binder\_size\_t \*offp, \*off\_end;

binder\_size\_t off\_min;

struct binder\_proc \*target\_proc;//目标进程,这个场景指的是ServiceManager进程

struct binder\_thread \*target\_thread = NULL;//目标线程, 这个场景指的是ServiceManager线程

struct binder\_node \*target\_node = NULL;

struct list\_head \*target\_list;

wait\_queue\_head\_t \*target\_wait;

struct binder\_transaction \*in\_reply\_to = NULL;

struct binder\_transaction\_log\_entry \*e;

if (reply) {

…

//BC\_REPLY命令才会走这个分支这里略过

} else {

if (tr->target.handle) {

struct binder\_ref \*ref;

//handle不为0的情况通过这个方法拿到远程服务对应的node

ref = binder\_get\_ref(proc, tr->target.handle) ;

target\_node = ref->node;

} else {

//传进来的handle值是0,走这个分支,拿到ServiceManager对应的node

target\_node = binder\_context\_mgr\_node;

}

target\_proc = target\_node->proc;//获取目标进程

//调用者线程还没有transaction\_stack任务栈所以这个场景不会走到这个分支

if (!(tr->flags & TF\_ONE\_WAY) && thread->transaction\_stack) {

…

}

}

if (target\_thread) {// target\_thread为空

…

} else {

//找出目标进程的todo和wait列表

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

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

}

//生成一个binder\_transaction,用于描述本次要进行的transaction,最后会将其加入目标进程的//target\_thread->todo列表中,等到目标进程唤醒之后可以从这个队列中取出要做的工作

t = kzalloc(sizeof(\*t), GFP\_KERNEL);

//生成一个binder\_work变量,用于描述调用者线程有一个未完成的transaction,最后会被加入调用者线程

//的todo列表中

tcomplete = kzalloc(sizeof(\*tcomplete), GFP\_KERNEL);

//为transaction成员赋值

if (!reply && !(tr->flags & TF\_ONE\_WAY))//此处flags是0 ,TF\_ONE\_WAY是0x01所以会走这个分支

t->from = thread;//设置发起请求的线程为调用者线程

else

t->from = NULL;

t->sender\_euid = task\_euid(proc->tsk);

t->to\_proc = target\_proc;//设置目标进程,对应ServiceManager

t->to\_thread = target\_thread;//设置目标线程,对应ServiceManager

t->code = tr->code;//请求码是GET\_SERVICE\_TRANSACTION

t->flags = tr->flags;//0

t->priority = task\_nice(current);//迁移优先级

//完成本次transaction申请的内存,这块内存是在目标进程的内核空间分配的

t->buffer = binder\_alloc\_buf(target\_proc,

tr->data\_size,tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));

t->buffer->allow\_user\_free = 0;

t->buffer->debug\_id = t->debug\_id;

t->buffer->transaction = t;//记录事务

t->buffer->target\_node = target\_node;//记录目标节点

if (target\_node)

binder\_inc\_node(target\_node, 1, 0, NULL);

offp = (binder\_size\_t \*)(t->buffer->data +ALIGN(tr->data\_size, sizeof(void \*)));

if (copy\_from\_user(t->buffer->data, (const void \_\_user \*)(uintptr\_t) tr->data.ptr.buffer, tr->data\_size)) {…}

if (copy\_from\_user(offp, (const void \_\_user \*)(uintptr\_t) tr->data.ptr.offsets, tr->offsets\_size)){…}

申请到t->buffer内存后把调用者进程用户空间的数据复制到目标进程的内核空间中去,因为这块内存指向的地址空间和目标进程的用户空间是共享的,所以只需要一次复制就把数据从Binder Client复制到Binder Server中了.

理解这个过程要从mmap说起,在与ServerManager发生IPC之前 ,一共有两次mmap操作,一次是在ServiceManager服务启动的时候,一次是在请求进程创建ProcessState的时候:

ServiceManager进程:

/frameworks/native/cmds/servicemanager/service\_manager.c

int main(int argc, char\*\* argv){

…

bs = binder\_open(driver, 128\*1024);

…

}

/frameworks/native/cmds/servicemanager/binder.c

struct binder\_state \*binder\_open(const char\* driver, size\_t mapsize){

…

bs->mapped = mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0);

…

}

请求进程:

/frameworks/native/libs/binder/ProcessState.cpp

// ProcessState的构造方法,一个进程中只有一个ProcessState但有多个IPCThreadState

ProcessState::ProcessState(const char \*driver){

…

// mmap the binder, providing a chunk of virtual address space to receive transactions.

//提供一块虚拟地址去接收transaction,返回的是虚拟地址的起始位置

mVMStart = mmap(0, BINDER\_VM\_SIZE, PROT\_READ, MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0);

…

}

进程分为用户空间和内核空间,这两种空间虚拟地址空间所在范围不同,mmap会依照用户空间虚拟地址的大小为内核空间申请相同大小的空间,并让用户空间虚拟地址和内核空间虚拟地址指向同一块物理地址即做物理内存映射,还会计算出用户空间虚拟地址和内核空间虚拟地址的地址差,这样拷贝数据的时候如果把数据拷贝到进程的内核空间的虚拟地址的时候也就拷贝到了进程的用户空间,而用户空间要访问这段数据,只要把内核空间的虚拟地址加上地址差就可以得到用户空间的虚拟地址,这样用户空间拿到这个地址就可以访问到数据了.

for (; offp < off\_end; offp++) {

struct flat\_binder\_object \*fp;

if (reply) {

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

} else if (!(t->flags & TF\_ONE\_WAY)) {//会走这个分支

t->need\_reply = 1;//need\_reply设置为1

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

thread->transaction\_stack = t; //调用者记录这个事务

} else {}

}

t->work.type = BINDER\_WORK\_TRANSACTION;//本次transaction的类型

list\_add\_tail(&t->work.entry, target\_list);把此transaction对应的binder\_work加入到目标进程的todo列表

tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;//当前进程认为的transaction类型

list\_add\_tail(&tcomplete->entry, &thread->todo);//加入到调用者线程的todo列表,表示有一个未完成的任务

if (target\_wait)//如果目标的wait列表不为空,就唤醒目标进程,这里指的是ServiceManager

wake\_up\_interruptible(target\_wait);

return;

ServiceManager被唤醒后Binder Client的处理:

调用者在完成binder\_transaction后,会先返回binder\_thread\_write,再返回binder\_ioctl,这样write请求已经完成,紧接着会判断用户是否有read请求:

if (bwr.read\_size > 0) {

ret = binder\_thread\_read(proc, thread, bwr.read\_buffer, bwr.read\_size,&bwr.read\_consumed,

filp->f\_flags & O\_NONBLOCK);

if (!list\_empty(&proc->todo))

wake\_up\_interruptible(&proc->wait);

}

根据传参,read\_size是大于0的,具体逻辑由binder\_thread\_read处理,这个函数和binder\_thread\_write的请求类似.

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)

proc和thread:分别是调用者进程和调用者线程,binder\_buffer是调用者提供的数据写入的空间,也就是IPCThreadState中的mIn数据存储空间,size是调用者希望读取的数据大小,consumed对应bwr.read\_consumed = 0:

void \_\_user \*buffer = (void \_\_user \*)(uintptr\_t)binder\_buffer;//取出写入空间

void \_\_user \*ptr = buffer + \*consumed;//数据的起点

void \_\_user \*end = buffer + size;//数据的终点

int ret = 0;

int wait\_for\_proc\_work;

if (\*consumed == 0) {

if (put\_user(BR\_NOOP, (uint32\_t \_\_user \*)ptr))//写入一个BR\_NOOP命令

return –EFAULT;

ptr += sizeof(uint32\_t);

}

…

while (1) {

uint32\_t cmd;

struct binder\_transaction\_data tr;

struct binder\_work \*w;

struct binder\_transaction \*t = NULL;

//注意,之前的binder\_thread\_write方法在调用者线程todo列表中加入了一个transaction,类型是//BINDER\_WORK\_TRANSACTION\_COMPLETE;

if (!list\_empty(&thread->todo)) {

w = list\_first\_entry(&thread->todo, struct binder\_work, entry);//取出这个transaction work

} else if (!list\_empty(&proc->todo) && wait\_for\_proc\_work) {

} else {}

**}**

switch (w->type) {

case BINDER\_WORK\_TRANSACTION\_COMPLETE: {

cmd = BR\_TRANSACTION\_COMPLETE;

if (put\_user(cmd, (uint32\_t \_\_user \*)ptr))//又写入了一个BR\_TRANSACTION\_COMPLETE命令 return –EFAULT;

} break;

….

}

put\_user(x,p)

put\_user用于将内核空间的一个简单类型变量x拷贝到p所指向的用户空间,该函数可以自动判断变量的类型,如果执行成功则返回0,否则返回-EFAULT

这样binder\_thead\_read读取了两个命令,然后回到binder\_ioctl再回到TPCThreadState中的talkWithDriver

/frameworks/native/libs/binder/IPCThreadState.cpp

status\_t IPCThreadState::waitForResponse(Parcel \*reply, status\_t \*acquireResult){

uint32\_t cmd;

int32\_t err;

while (1) {

if ((err=talkWithDriver()) < NO\_ERROR) break;

if (mIn.dataAvail() == 0) continue;如果没有可用数据继续循环

…

cmd = (uint32\_t)mIn.readInt32();

switch (cmd) {

case BR\_TRANSACTION\_COMPLETE:

if (!reply && !acquireResult) goto finish;

break;

default:

err = executeCommand(cmd);//BR\_NOOP会走到这个分支因为前面没有这个case

if (err != NO\_ERROR) goto finish;

break;

}

}

…

return err;

}

status\_t IPCThreadState::executeCommand(int32\_t cmd){

switch ((uint32\_t)cmd) {

…

case BR\_NOOP://什么都没做

break;

…

return result;

}

执行完这两个命令后会继续循环,但第二次循环bwr.write\_size = 0,而bwr.read\_size 还是大于0,所以在调用进入Binder驱动的binder\_ioctl中后就不会再执行binder\_thread\_write了,可直接进入binder\_thread\_read中

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){

void \_\_user \*buffer = (void \_\_user \*)(uintptr\_t)binder\_buffer;

void \_\_user \*ptr = buffer + \*consumed;

void \_\_user \*end = buffer + size;

int ret = 0;

int wait\_for\_proc\_work;

if (\*consumed == 0) {

if (put\_user(BR\_NOOP, (uint32\_t \_\_user \*)ptr))//首先韩式先写入一个BR\_NOOP

return –EFAULT;

ptr += sizeof(uint32\_t);

}

retry:

//由于在binder\_thread\_write中对这两个值都有赋值所以这个变量为false

wait\_for\_proc\_work = thread->transaction\_stack == NULL &&list\_empty(&thread->todo);

thread->looper |= BINDER\_LOOPER\_STATE\_WAITING;

if (wait\_for\_proc\_work) {

…

} else {

if (non\_block) {

if (!binder\_has\_thread\_work(thread))

ret = -EAGAIN;

} else

//会进入这个分支,至此ServiceManager进程被唤醒后调用者进程进入等待

ret = wait\_event\_freezable(thread->wait, binder\_has\_thread\_work(thread));

}

}

ServiceManager被唤醒后的操作:

ServiceManager启动的时候进行了一系列的初始化操作,包括open\_driver,mmap,让自己成为Binder Server大管家即在驱动注册自己成为binder\_context\_mgr\_node,然后开启循环,等待请求,解析请求.

/frameworks/native/cmds/servicemanager/service\_manager.c

int main(int argc, char\*\* argv){

struct binder\_state \*bs;

union selinux\_callback cb;

char \*driver;

driver = “/dev/binder”;

bs = binder\_open(driver, 128\*1024);

if (binder\_become\_context\_manager(bs)) {//把自己设置为Binder管家,DNS服务器的角色

ALOGE(“cannot become context manager (%s)\n”, strerror(errno));

return -1;

}

…

binder\_loop(bs, svcmgr\_handler); //开启循环,等待客户端的注册查询等请求

return 0;

}

void binder\_loop(struct binder\_state \*bs, binder\_handler func){

int res;

struct binder\_write\_read bwr;//binder驱动交互所需的数据结构体

uint32\_t readbuf[32];

//一开始启动时不会写入数据

bwr.write\_size = 0;

bwr.write\_consumed = 0;

bwr.write\_buffer = 0;

readbuf[0] = BC\_ENTER\_LOOPER;//告知binder驱动进入循环

binder\_write(bs, readbuf, sizeof(uint32\_t));

for (;;) {

bwr.read\_size = sizeof(readbuf);

bwr.read\_consumed = 0;

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

res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);//与binder驱动交互

res = binder\_parse(bs, 0, (uintptr\_t) readbuf, bwr.read\_consumed, func);//解析请求

}

}

因为ServiceManager启动的时间很早,一开始ServiceManager与驱动进行交互的时候没有写入数据,只会读取数据,所以在binder驱动中只会走到binder\_thread\_read方法

/drivers/staging/android/binder.c

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){

void \_\_user \*ptr = buffer + \*consumed;

…

//由于一开始就没有对这两个值都有赋值所以这个变量为true

wait\_for\_proc\_work = thread->transaction\_stack == NULL &&list\_empty(&thread->todo);

if (wait\_for\_proc\_work) {

…

if (non\_block) {

…

} else

//一开始走到这里会进入等待,等待Binder Client请求

ret = wait\_event\_freezable\_exclusive(proc->wait, binder\_has\_proc\_work(proc, thread));

} else {

}

//后来Binder Client发送getService()请求的时候会唤醒ServiceManager,所以就继续走下面的逻辑

while (1) {

uint32\_t cmd;

struct binder\_transaction\_data tr;

struct binder\_work \*w;

struct binder\_transaction \*t = NULL;

//Binder Client发出请求,在binder\_thread\_write中向Binder Server进程的todo列表添加

//了一个transaction,所以这里不为空

if (!list\_empty(&thread->todo)) {

w = list\_first\_entry(&thread->todo, struct binder\_work, entry);//取出binder\_work

} else if (!list\_empty(&proc->todo) && wait\_for\_proc\_work) {

w = list\_first\_entry(&proc->todo, struct binder\_work,entry);

} else {}

switch (w->type) {

//Binder Client发出请求时写入的transaction binder\_work类型

case BINDER\_WORK\_TRANSACTION: {

//根据为binder\_work找出binder\_transaction

t = container\_of(w, struct binder\_transaction, work);

} break;

container\_of在Linux内核中是一个常用的宏,用于从包含在某个结构中的指针获得结构本身的指针,通俗地讲就是通过结构体变量中某个成员的首地址进而获得整个结构体变量的首地址.

接口:

container\_of(ptr, type, member)

 ptr:表示结构体中member的地址

 type:表示结构体类型

 member:表示结构体中的成员

通过ptr的地址可以返回结构体的首地址.

//Binder Client发请求时对这个参数赋值为ServiceManager在驱动中对应的节点

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

struct binder\_node \*target\_node = t->buffer->target\_node;

//把数据提取出来准备发送给用户空间的ServiceManager

tr.target.ptr = target\_node->ptr;

tr.cookie = target\_node->cookie;

t->saved\_priority = task\_nice(current);

if (t->priority < target\_node->min\_priority &&

!(t->flags & TF\_ONE\_WAY))

binder\_set\_nice(t->priority);

else if (!(t->flags & TF\_ONE\_WAY) ||

t->saved\_priority > target\_node->min\_priority)

binder\_set\_nice(target\_node->min\_priority);

cmd = BR\_TRANSACTION;//命令参数

} else {

tr.target.ptr = 0;

tr.cookie = 0;

cmd = BR\_REPLY;

}

tr.code = t->code;

tr.flags = t->flags;

tr.sender\_euid = from\_kuid(current\_user\_ns(), t->sender\_euid);

if (t->from) {

struct task\_struct \*sender = t->from->proc->tsk;

tr.sender\_pid = task\_tgid\_nr\_ns(sender,task\_active\_pid\_ns(current));

} else {

tr.sender\_pid = 0;

}

tr.data\_size = t->buffer->data\_size;//data区域大小

tr.offsets\_size = t->buffer->offsets\_size;//offs区域大小

//Binder Server用户空间数据区域起始地址 = Binder Server内核空间的数据区域的起始地址+

//用户空间虚地址与内核空间虚拟地址的差值

tr.data.ptr.buffer = (binder\_uintptr\_t)((uintptr\_t)t->buffer->data +proc->user\_buffer\_offset);

// Binder Server用户空间offs区域起始地址= Binder Server用户空间数据区域起始地址+

//数据区域的大小

tr.data.ptr.offsets = tr.data.ptr.buffer +ALIGN(t->buffer->data\_size, sizeof(void \*));

if (put\_user(cmd, (uint32\_t \_\_user \*)ptr))//向ServiceManager的用户空间发送命令

ptr += sizeof(uint32\_t);

/\*将tr中的数据拷贝到ptr指向的Binder Server用户空间的readbuf

\*这个不是 copy 数据的,而是 copy binder\_transaction\_data 这个数据结构,只不过这个数据结构里

\*有指向数据的地址,所以这个不算在 binder 数据传递的复制次数中,也就是说就算传递比较大的

\*数据,这次复制只是复制一个数据结构的大小.根据前面的分析,binder\_thread\_read 返回

\*binder\_ioctl 就返回了\*/

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

return –EFAULT;

//由于前面修改了cmd为BR\_TRANSACTION所以会走到这个分支

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

t->to\_parent = thread->transaction\_stack;//唤醒后的操作,记录的是自己

t->to\_thread = thread//唤醒后的操作,记录的是自己

//SM线程事务栈记录这个binder\_transaction,t中包含请求者信息,用于在SM回复时找到

//指定请求者进行回复

thread->transaction\_stack = t;

} else {}

}

回到Binder Server的用户空间, 读到请求后开始解析请求:

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[native](http://androidxref.com/9.0.0_r3/xref/frameworks/native/)/[cmds](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/)/[servicemanager](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/)/[binder.c](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/binder.c)

int binder\_parse(struct binder\_state \*bs, struct binder\_io \*bio,

uintptr\_t ptr, size\_t size, binder\_handler func){

…

while (ptr < end) {

…

switch(cmd) {

case BR\_TRANSACTION: {

//把传进来的指针转成结构体struct binder\_transaction\_data

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

binder\_dump\_txn(txn);

if (func) {

unsigned rdata[256/4];

struct binder\_io msg;

struct binder\_io reply;

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

//把数据转换成binder\_io类型的结构体msg

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

//func是之前在[service\_manager.c](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/service_manager.c)中binder\_loop传进来的svcmgr\_handler函数

res = func(bs, txn, &msg, &reply);//请求交给这个函数处理

//把结果写回binder驱动

binder\_send\_reply(bs, &reply, txn->data.ptr.buffer, res);

}

}

break;

}

}

return r;

}

struct binder\_io{

char \*data; /\* 数据区当前地址,从该指针指向的内存空间处开始读或写; \*/

binder\_size\_t \*offs; /\* offs区域当前地址,数组偏移地址 \*/

size\_t data\_avail; /\* 数据区域中可以有效利用的数据长度,即数据区剩余空间\*/

size\_t offs\_avail; /\* offs区域中可以存放的数组元素个数 , 即offs区域剩余空间\*/

char \*data0; /\* 数据区的起始地址 \*/

binder\_size\_t \*offs0; /\* offs区域的起始地址 \*/

uint32\_t flags;

uint32\_t unused;

};

binder\_io对数据划分了两个区域data区和offs,每个区又有起始地址data0,offs0和当前地址data,0ffs, 这个数据结构是用来存储flat\_binder\_object数据的

void bio\_init(struct binder\_io \*bio, void \*data, size\_t maxdata, size\_t maxoffs){

size\_t n = maxoffs \* sizeof(size\_t);//最大偏移16个字节

bio->data = bio->data0 = (char \*) data + n;

bio->offs = bio->offs0 = data;

bio->data\_avail = maxdata – n;

bio->offs\_avail = maxoffs;//

bio->flags = 0;

}

//处理请求的函数

/frameworks/native/cmds/servicemanager/service\_manager.c

int svcmgr\_handler(struct binder\_state \*bs,struct binder\_transaction\_data \*txn,struct binder\_io \*msg,

struct binder\_io \*reply){

struct svcinfo \*si;

uint16\_t \*s;

size\_t len;

uint32\_t handle;

uint32\_t strict\_policy;

int allow\_isolated;

uint32\_t dumpsys\_priority;

switch(txn->code) {

…

case SVC\_MGR\_GET\_SERVICE:

case SVC\_MGR\_CHECK\_SERVICE:

s = bio\_get\_string16(msg, &len); //从msg中拿到data数据也就是Binder Server的名称

//SVC\_MGR\_GET\_SERVICE是查询binder服务的命令由Binder Client发出的

//查找符合要求的Binder Server,返回值handle是什么东东,要看添加Binder Server的时候做了什么.

Handle = do\_find\_service(s, len, txn->sender\_euid, txn->sender\_pid);

bio\_put\_ref(reply, handle);/写入到reply中返回给客户端

return 0;

case SVC\_MGR\_ADD\_SERVICE:

s = bio\_get\_string16(msg, &len);//获取服务名称

handle = bio\_get\_ref(msg);//获取handle值

allow\_isolated = bio\_get\_uint32(msg) ? 1 : 0;

dumpsys\_priority = bio\_get\_uint32(msg);

//如果请求的添加服务就会走到这个分支,执行添加逻辑

if (do\_add\_service(bs, s, len, handle, txn->sender\_euid, allow\_isolated, dumpsys\_priority,txn->sender\_pid))

return -1;

break;

bio\_put\_uint32(reply, 0);

return 0;

}

uint32\_t bio\_get\_ref(struct binder\_io \*bio){

struct flat\_binder\_object \*obj;

obj = \_bio\_get\_obj(bio);

if (obj->hdr.type == BINDER\_TYPE\_HANDLE)

return obj->handle;//返回的是一个handle值

return 0;

}

int do\_add\_service(struct binder\_state \*bs, const uint16\_t \*s, size\_t len, uint32\_t handle,

uid\_t uid, int allow\_isolated, uint32\_t dumpsys\_priority, pid\_t spid) {

struct svcinfo \*si;//是一个链表

si = find\_svc(s, len);//查找是否已经添加过了

if (si) {

si->handle = handle;//如果已经添加过了就更新hanlde值

} else {

//没有添加就新增一个包含所有传进来的必要信息的svcinfo

si = malloc(sizeof(\*si) + (len + 1) \* sizeof(uint16\_t));

si->handle = handle;

si->len = len;

memcpy(si->name, s, (len + 1) \* sizeof(uint16\_t));

si->name[len] = ‘\0’;

si->death.func = (void\*) svcinfo\_death;

si->death.ptr = si;

si->allow\_isolated = allow\_isolated;

si->dumpsys\_priority = dumpsys\_priority;

si->next = svclist;

svclist = si;//插入到表头

}

…

return 0;

}

struct svcinfo \*find\_svc(const uint16\_t \*s16, size\_t len){

struct svcinfo \*si;

for (si = svclist; si; si = si->next) {

//遍历内部链表如果找到了相同名称的svcinfo就返回

if ((len == si->len) && !memcmp(s16, si->name, len \* sizeof(uint16\_t))) {

return si;

}

}

return NULL;

}

//ServiceManager内部维护着一个svclist列表,数据结构是svcinfo的结构体

struct svcinfo{

struct svcinfo \*next;//链表结构,存储所有BinderServer

uint32\_t handle; //BinderServer对应的handle值

struct binder\_death death; //BinderServer死亡代理

int allow\_isolated;

uint32\_t dumpsys\_priority;

size\_t len; //BinderServer对应名称的长度

uint16\_t name[0]; //是个数组.BinderServer对应名称

};

关于memcmp函数:

原型:int memcmp(const void \*str1, const void \*str2, size\_t n));

参数:

str1–指向内存块的指针。

Str2–指向内存块的指针。

n–要被比较的字节数。

功能: 比较内存区域str1和str2的前n个字节。

返回到case SVC\_MGR\_GET\_SERVICE:

uint32\_t do\_find\_service(const uint16\_t \*s, size\_t len, uid\_t uid, pid\_t spid){

struct svcinfo \*si = find\_svc(s, len);//找到相同名称的svcinfo

…

return si->handle;//找到的话返回的是对应的handle值

}

void bio\_put\_ref(struct binder\_io \*bio, uint32\_t handle){

//传进来的bio是reply,handle是查找得到的

struct flat\_binder\_object \*obj;

if (handle)

obj = bio\_alloc\_obj(bio);//如果找到了handle就会走这里,根据reply创建一个flat\_binder\_object

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

obj->hdr.type = BINDER\_TYPE\_HANDLE;//这个类型后面还会用到

obj->handle = handle;//把handle保存到obj中

obj->cookie = 0;

}

static struct flat\_binder\_object \*bio\_alloc\_obj(struct binder\_io \*bio){

struct flat\_binder\_object \*obj;

//分配obj的地址指向reply的data区域,因此向obj中写数据就相当于在reply的data区域写数据

obj = bio\_alloc(bio, sizeof(\*obj));

if (obj && bio->offs\_avail) {//如果分配成功就在offs区域记录data区域中的变化

// reply的offs区域是一个size\_t型数组

bio->offs\_avail--;//reply的offs区域可用数组空间减少1

// 每个元素记录data区域中flat\_binder\_object相对于data0的偏移量

\*bio->offs++ = ((char\*) obj) – ((char\*) bio->data0);

return obj;

}

bio->flags |= BIO\_F\_OVERFLOW;

return NULL;

}

static void \* bio\_alloc(struct binder\_io \*bio, size\_t size){

size = (size + 3) & (~3);

if (size > bio->data\_avail) {

bio->flags |= BIO\_F\_OVERFLOW;

return NULL;

} else {

void \*ptr = bio->data;//可以看到返回的obj指向reply的data区域

bio->data += size;//reply的data区域大小增加size,size指的是flat\_binder\_object的大小

bio->data\_avail -= size;//reply的data的可用区域减少size大小

return ptr;

}

}

由以上分析,bio即reply中有一块flat\_binder\_object区域,类型为BINDER\_TYPE\_HANDLE,flat\_binder\_object->handle指向了从SM中查找到的Service指针,这个指针最终将被转换成IBinder.

到这里SM端数据已经查找完毕,准备发往驱动,回到binder\_send\_reply

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[native](http://androidxref.com/9.0.0_r3/xref/frameworks/native/)/[cmds](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/)/[servicemanager](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/)/[binder.c](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/binder.c)

void binder\_send\_reply(struct binder\_state \*bs, struct binder\_io \*reply, binder\_uintptr\_t buffer\_to\_free,int status){

struct {

uint32\_t cmd\_free;

binder\_uintptr\_t buffer;

uint32\_t cmd\_reply;

struct binder\_transaction\_data txn;

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

data.cmd\_free = BC\_FREE\_BUFFER;

data.buffer = buffer\_to\_free;

data.cmd\_reply = BC\_REPLY;

data.txn.target.ptr = 0;

data.txn.cookie = 0;

data.txn.code = 0;

if (status) {

…//没有错误就不会走这个分支

} else {

data.txn.flags = 0;

data.txn.data\_size = reply->data – reply->data0;//reply中的data区域数据大小

data.txn.offsets\_size = ((char\*) reply->offs) – ((char\*) reply->offs0); ;//reply中的offs区域数据大小

data.txn.data.ptr.buffer = (uintptr\_t)reply->data0; ;//reply中的data区域数据起始地址

data.txn.data.ptr.offsets = (uintptr\_t)reply->offs0; //reply中的offs区域数据起始地址

}

binder\_write(bs, &data, sizeof(data));//发送给驱动

}

//通过ioctl与binder驱动交互,遵循binder\_write\_read结构体数据格式

int binder\_write(struct binder\_state \*bs, void \*data, size\_t len){

struct binder\_write\_read bwr;

int res;

bwr.write\_size = len;

bwr.write\_consumed = 0;

bwr.write\_buffer = (uintptr\_t) data;//填充数据

bwr.read\_size = 0;

bwr.read\_consumed = 0;

bwr.read\_buffer = 0;

// BINDER\_WRITE\_READ即可以读也可以写,read\_size为0 ,write\_buffer不为0所以这个操作是写

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

return res;

}

和读的流程一样oictl---->binder\_ioctl----> binder\_ioctl\_write\_read----> binder\_thread\_write----> binder\_transaction,首先会进入binder\_thread\_write-命令为BC\_REPLY所以也会交给binder\_transaction处理:

/drivers/staging/android/binder.c

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

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

//与前面的BC\_TRANSACTION一样只不过发起方是相反, BC\_TRANSACTION的发起者是

//Binder Client,BC\_REPLY的发起者是Binder Server SM

..

if (reply) {//为true

//取出之前在binder\_thread\_read中存入的t即binder\_transaction事务,这个结构体存储的是请

//求者信息

in\_reply\_to = thread->transaction\_stack;

target\_thread = in\_reply\_to->from;//找到请求者线程

target\_proc = target\_thread->proc;//找到请求者进程

}

if (target\_thread) {

target\_list = &target\_thread->todo;//找到请求者的transaction队列

target\_wait = &target\_thread->wait; //找到请求者的wait队列

} else {…}

t = kzalloc(sizeof(\*t), GFP\_KERNEL);//发给请求者的一个binder\_transaction任务

tcomplete = kzalloc(sizeof(\*tcomplete), GFP\_KERNEL);//代表SM端的一个未完成的任务

if (!reply && !(tr->flags & TF\_ONE\_WAY))

else

t->from = NULL;

t->sender\_euid = task\_euid(proc->tsk);

t->to\_proc = target\_proc;

t->to\_thread = target\_thread;

t->code = tr->code;

t->flags = tr->flags;

t->priority = task\_nice(current);

//根据SM传输的数据大小在目标进程即调用者进程中分配相同大小的空间

t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));

t->buffer->allow\_user\_free = 0;

t->buffer->transaction = t;

t->buffer->target\_node = target\_node;

// offp是offs区域起始地址即data区域起始地址 + data区域大小

offp = (binder\_size\_t \*)(t->buffer->data +ALIGN(tr->data\_size, sizeof(void \*)));

//拷贝data区域

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

//拷贝offs区域

if (copy\_from\_user(offp, (const void \_\_user \*)(uintptr\_t)tr->data.ptr.offsets, tr->offsets\_size)

//SM的回复中包换flat\_binder\_object这里进行还原

off\_end = (void \*)offp + tr->offsets\_size;//offs区域起始地址加上offs区域大小就是数据的结束位置

off\_min = 0;

for (; offp < off\_end; offp++) {

struct flat\_binder\_object \*fp;//

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

switch (fp->type) {

//遍历SM中维护的红黑树根据handle找到相对应binder\_ref

struct binder\_ref \*ref = binder\_get\_ref(proc, fp->handle);//

if (ref->node->proc == target\_proc) {//同一进程的处理

…

} else {

struct binder\_ref \*new\_ref;

//这个函数主要是根据请求者进程创建一个新的binder\_ref节点

new\_ref = binder\_get\_ref\_for\_node(target\_proc, ref->node);

fp->handle = new\_ref->desc;

}

}

static struct binder\_ref \*binder\_get\_ref(struct binder\_proc \*proc, uint32\_t desc){

struct rb\_node \*n = proc->refs\_by\_desc.rb\_node;//找到SM进程的红黑树

struct binder\_ref \*ref;

//遍历红黑树,返回要找的binder\_ref

while (n) {

ref = rb\_entry(n, struct binder\_ref, rb\_node\_desc);

if (desc < ref->desc)

n = n->rb\_left;

else if (desc > ref->desc)

n = n->rb\_right;

else

return ref;

}

return NULL;

}

if (reply) {

binder\_pop\_transaction(target\_thread, in\_reply\_to);//事务出栈操作

} else if (!(t->flags & TF\_ONE\_WAY)) {…} else {…}

t->work.type = BINDER\_WORK\_TRANSACTION;//发给请求者的业务类型

list\_add\_tail(&t->work.entry, target\_list);//加入到请求者事务队列

tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE; //SM业务类型

//加入到SM事务队列中, 代表SM有一个未完成的事务

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

if (target\_wait)

wake\_up\_interruptible(target\_wait);//唤醒请求者进程

return;

}

回想一下:之前请求者进程即Binder Client在经过waitForResponse-(IPCThreadState.cpp)---->talkWithDriver(TPCThreadState.cpp)----->ioctl(IPCThreadState.cpp)------>binder\_ioctl(Binder.c)------>binder\_ioctl\_write\_read(Binder.c)------->binder\_thread\_write(Binder.c)------>binder\_thread\_read(Binder.c)后进入了睡眠,现在SM已经给了回复,请求者进程被唤醒后需要处理回复的结果

SM也给自己加入了一个事务,用于通知自己请求者的请求已经处理完毕,会再次进入休眠状态,等待请求者发起新的请求.

请求者是在binder\_thread\_read方法中ret = wait\_event\_freezable(thread->wait, binder\_has\_thread\_work(thread));这行代码被休眠的,被唤醒后会接着走后面的代码:

while (1) {

uint32\_t cmd;

struct binder\_transaction\_data tr;

struct binder\_work \*w;

struct binder\_transaction \*t = NULL;

if (!list\_empty(&thread->todo)) {

w = list\_first\_entry(&thread->todo, struct binder\_work,entry);//找出之前加入的事务

} else if (!list\_empty(&proc->todo) && wait\_for\_proc\_work) {} else {}

switch (w->type) {

case BINDER\_WORK\_TRANSACTION:

//根据binder\_transaction结构体中的变量,还原结构体

t = container\_of(w, struct binder\_transaction, work);

break;

}

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

//target\_node在SM调用进入Binder驱动中时没有被赋值

} else {

tr.target.ptr = 0;

tr.cookie = 0;

cmd = BR\_REPLY;

}

tr.data\_size = t->buffer->data\_size;//数据区域大小

tr.offsets\_size = t->buffer->offsets\_size;//size区域大小

//请求者用户空间data区域起始地址=请求者内核空间data区域起始地址+用户空间虚地址与内核//空间虚拟地址的差值

tr.data.ptr.buffer = (binder\_uintptr\_t)((uintptr\_t)t->buffer->data +proc->user\_buffer\_offset);

//请求者用户空间offs区域起始地址 = 请求者用户空间data区域起始地址+data区域数据大小

tr.data.ptr.offsets = tr.data.ptr.buffer +ALIGN(t->buffer->data\_size, sizeof(void \*));

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

ptr += sizeof(uint32\_t);

if (copy\_to\_user(ptr, &tr, sizeof(tr)))//与前面一样只是复制binder\_transaction\_data结构体

ptr += sizeof(tr);

}

请求者进程在waitForResponse中收到BR\_REPLY后:

status\_t IPCThreadState::waitForResponse(Parcel \*reply, status\_t \*acquireResult){

While(1){

case BR\_REPLY: {

binder\_transaction\_data tr;

err = mIn.read(&tr, sizeof(tr));//把mIn中的数据读取到结构体中

if (reply) {

if ((tr.flags & TF\_STATUS\_CODE) == 0) {

reply->ipcSetDataReference(

reinterpret\_cast<const uint8\_t\*>(tr.data.ptr.buffer),//SM传递的data区域

tr.data\_size,//data区域大小

//offs区域,存的是flat\_binder\_object相对于data位置的偏移量

reinterpret\_cast<const binder\_size\_t\*>(tr.data.ptr.offsets),

tr.offsets\_size/sizeof(binder\_size\_t),//获取flat\_binder\_object的个数,此场景个数是1

freeBuffer, this);

} else if.(){…} else {…}

}

}

}

void Parcel::ipcSetDataReference(const uint8\_t\* data, size\_t dataSize,

const binder\_size\_t\* objects, size\_t objectsCount, release\_func relFunc, void\* relCookie){

binder\_size\_t minOffset = 0;

freeDataNoInit();

mError = NO\_ERROR;

mData = const\_cast<uint8\_t\*>(data);

mDataSize = mDataCapacity = dataSize;

mDataPos = 0;

mObjects = const\_cast<binder\_size\_t\*>(objects);

mObjectsSize = mObjectsCapacity = objectsCount;

mNextObjectHint = 0;

mObjectsSorted = false;

mOwner = relFunc;

mOwnerCookie = relCookie;

scanForFds();

//以上是把SM回复的数据填充到Parcel中,Parcel是IPC数据的载体

}

到这里transact的过程已经完成了,数据已经填充到了reply中,回到发起调用的代码:

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[base](http://androidxref.com/9.0.0_r3/xref/frameworks/base/)/[core](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/)/[java](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/)/[android](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/)/[os](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/os/)/[ServiceManagerNative.java](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/os/ServiceManagerNative.java)

//ServiceManagerProxy对象的getService(String name)方法

public IBinder getService(String name) throws RemoteException {

Parcel.obtain();获取Parcel对象,是通过JNI获取的,指向在C++创建的Parcel对象

Parcel data = Parcel.obtain();//用于发送给binder驱动数据的对象

Parcel reply = Parcel.obtain();//创建用于接收binder驱动数据的对象

data.writeInterfaceToken(IServiceManager.descriptor);//此接口标识

data.writeString(name);//要查询的服务名称如”activity”

mRemote.transact(GET\_SERVICE\_TRANSACTION, data, reply, 0);//同步调用,调用的是BinderProxy对象的transact方法,BinderProxy是Binder的内部类, 在这一步之前都是在客户端进程的调用,这个函数中将真正发起跨进程调用.

IBinder binder = reply.readStrongBinder();//从reply中读取flat\_binder\_object

reply.recycle();

data.recycle();

return binder;

}

/frameworks/native/libs/binder/Parcel.cpp

接下来就开始从reply中读取数据reply.readStrongBinder();通过JNI调用来到C++层的Parcel

sp<IBinder> Parcel::readStrongBinder() const{

sp<IBinder> val;

readNullableStrongBinder(&val);

return val;

}

status\_t Parcel::readNullableStrongBinder(sp<IBinder>\* val) const{

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(false);

if (flat) {

switch (flat->hdr.type) {

case BINDER\_TYPE\_BINDER://同一个进程

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

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

case BINDER\_TYPE\_HANDLE://不同进程,由于是跨进程调用所以会走这里

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

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

}

}

return BAD\_TYPE;

}

const flat\_binder\_object\* Parcel::readObject(bool nullMetaData) const{

const size\_t DPOS = mDataPos;

if ((DPOS+sizeof(flat\_binder\_object)) <= mDataSize) {

//根据地址还原flat\_binder\_object

const flat\_binder\_object\* obj= reinterpret\_cast<const flat\_binder\_object\*>(mData+DPOS);

mDataPos = DPOS + sizeof(flat\_binder\_object);//移动data数据的游标

}

return NULL;

}

/frameworks/native/libs/binder/ProcessState.cpp

//这个函数之前在查找SM远程对象的时候分析过,这里查找的是普通远程对象,handle值不是0

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

sp<IBinder> result;

handle\_entry\* e = lookupHandleLocked(handle);//本地查找记录

if (e != NULL) {

IBinder\* b = e->binder;

if (b == NULL || !e->refs->attemptIncWeak(this)) {//如果没有找对对应的IBinder或增加应用计数失败

if (handle == 0) {

//不会走这个分支查找SM的时候才会走这里

}

b = BpBinder::create(handle);//根据handle值新建一个BpBinder

e->binder = b;

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

result = b;

} else { }

}

return result;//返回BpBinder,返回给调用者时被强制转换成了IBbinder,即远程服务在本地的代理

}

这样ServerManager.getService()整个流程就走完了.