原 红茶一杯话Binder（ServiceManager篇）

发表于2年前(2013-08-02 22:01) 阅读（5217） | 评论（8） 26 人收藏此文章, 我要收藏

红茶一杯话Binder
(ServiceManager篇)

侯亮

1.先说一个大概

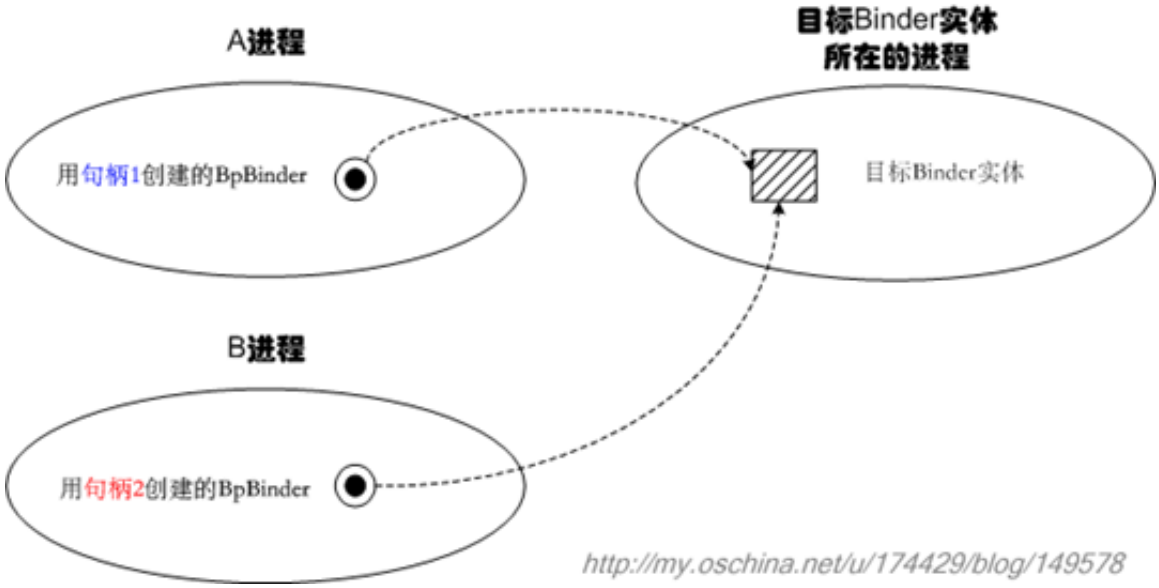
Android平台的一个基本设计理念是构造一个相对平坦的功能集合，这些功能可能会身处于不同的进程中，然而却可以高效地整合到一起，实现不同的用户需求。这就必须打破过去各个孤立App所形成的天然藩篱。为此，Android提供了Binder机制。

在Android中，系统提供的服务被包装成一个个系统级service，这些service往往会在设备启动之时添加进Android系统。在上一篇文档中，我们已经了解了BpBinder和BBinder的概念，而service实体的底层说到底就是一个BBinder实体。

我们知道，如果某个程序希望享受系统提供的服务，它就必须调用系统提供的外部接口，向系统发出相应的请求。因此，Android中的程序必须先拿到和某个系统service对应的代理接口，然后才能通过这个接口，享受系统提供的服务。说白了就是我们得先拿到一个和目标service对应的合法BpBinder。

然而，该怎么获取和系统service对应的代理接口呢？Android是这样设计的：先启动一个特殊的系统服务，叫作Service Manager Service（简称SMS），它的基本任务就是管理其他系统服务。其他系统服务在系统启动之时，就会向SMS注册自己，于是SMS先记录下与那个service对应的名字和句柄值。有了句柄值就可以用来创建合法的BpBinder了。只不过在实际的代码中，SMS并没有用句柄值创建出BpBinder，这个其实没什么，反正指代目标service实体的目的已经达到了。后续当某程序需要享受某系统服务时，它必须先以"特定手法"获取SMS代理接口，并经由这个接口查询出目标service对应的合法Binder句柄，然后再创建出合法的BpBinder对象。

在此，我们有必要交代一下"Binder句柄"的作用。句柄说穿了是个简单的整数值，用来告诉Binder驱动我们想找的目标Binder实体是哪个。但是请注意，句柄只对发起端进程和Binder驱动有意义，A进程的句柄直接拿到B进程，是没什么意义的。也就是说，不同进程中指代相同Binder实体的句柄值可能是不同的。示意图如下：



SMS记录了所有系统service所对应的Binder句柄，它的核心功能就是维护好这些句柄值。后续，当用户进程需要获取某个系统service的代理时，SMS就会在内部按service名查找到合适的句柄值，并"逻辑上"传递给用户进程，于是用户进程会得到一个新的合法句柄值，这个新句柄值可能在数值上和SMS所记录的句柄值不同，然而，它们指代的却是同一个Service实体。句柄的合法性是由Binder驱动保证的，这一点我们不必担心。

前文我们提到要以"特定手法"获取SMS代理接口，这是什么意思呢？在IServiceManager.cpp文件中，我们可以看到一个defaultServiceManager()函数，代码如下：

【frameworks/native/libs/binder/IServiceManager.cpp】

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL)
        return gDefaultServiceManager;
    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL)
        {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }

    return gDefaultServiceManager;
}
```

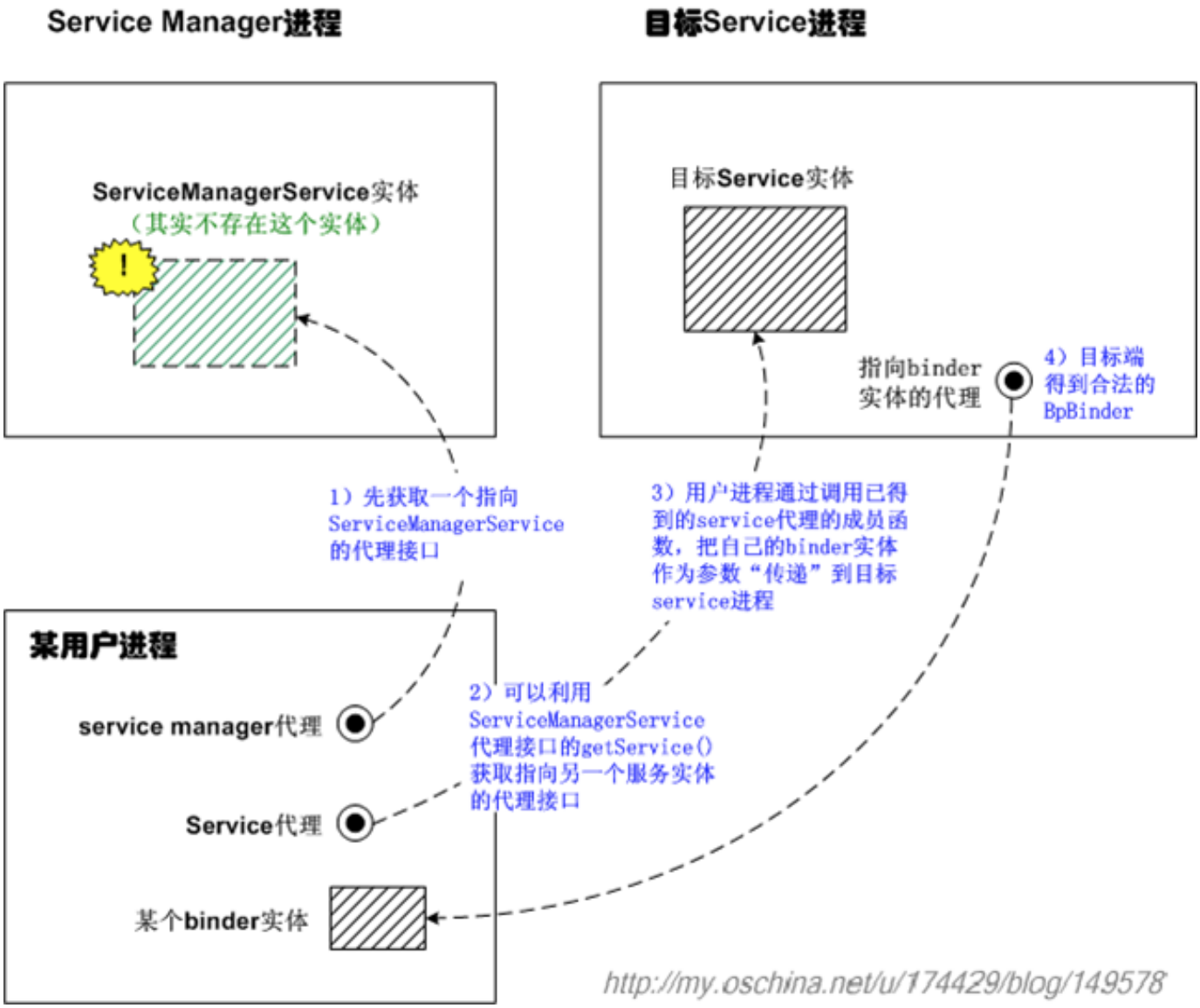
这个函数里调用interface_cast的地方是用一句getContextObject(NULL)来获取BpBinder对象的。我们先不深入讲解这个函数，只需要知道这一句里的getContextObject(NULL)实际上相当于new BpBinder(0)就可以了。噢，看来要得到BpBinder对象并不复杂嘛，直接new就好了。然而，我之所以使用"特定手法"一词，是因为这种直接new BpBinder(xxx)的做法，只能用于获取SMS的代理接口。大家可不要想当然地随便用这种方法去创建其他服务的代理接口噢。

在Android里，对于Service Manager Service这个特殊的服务而言，其对应的代理端的句柄值已经预先定死为0了，所以我们直接new BpBinder(0)拿到的就是个合法的BpBinder，其对端为"Service Manager Service实体"（至少目前可以先这么理解）。那么对于其他"服务实体"对应的代理，句柄值又是多少呢？使用方又该如何得到这个句柄值呢？我们总不能随便蒙一个句柄值吧。正如我们前文所述，要得到某个服务对应的

BpBinder，主要得借助Service Manager Service系统服务，查询出一个合法的Binder句柄，并进而创建出合法的BpBinder。

这里有必要澄清一下，利用SMS获取合法BpBinder的方法，并不是Android中得到BpBinder的唯一方法。另一种方法是，"起始端"经由一个已有的合法BpBinder，将某个binder实体或代理对象作为跨进程调用的参数，"传递"给"目标端"，这样目标端也可以拿到一个合法的BpBinder。

我们把以上介绍的知识绘制成示意图，如下：



请顺着图中标出的1）、2）、3）、4）序号，读一下图中的说明。

在跨进程通信方面，所谓的"传递"一般指的都是逻辑上的传递，所以应该打上引号。事实上，binder实体对象是不可能完全打包并传递到另一个进程的，而且也没有必要这么做。目前我们只需理解，binder架构会保证"传递"动作的目标端可以拿到一个和binder实体对象对应的代理对象即可。详细情况，要到分析binder驱动的部分再阐述。

既然SMS承担着让客户端获取合法BpBinder的责任，那么它的重要性就不言而喻了。现在我们就来详细看看具体如何使用它。

2.具体使用Service Manager Service

2.1 必须先得到IServiceManager代理接口

要获取某系统service的代理接口，必须先得到IServiceManager代理接口。还记得前文C++代码中获取IServiceManager代理接口的句子吗?

```
gDefaultServiceManager = interface_cast<IServiceManager>(
    ProcessState::self()->getContextObject(NULL));
```

我们在前一篇文档中已经介绍过interface_cast了，现在再贴一下这个函数的代码:

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

也就是说，其实调用的是IServiceManager::asInterface(obj)，而这个obj参数就是new BpBinder(0)得到的对象。当然，这些都是C++层次的概念，Java层次把这些概念都包装起来了。

在Java层次，是这样获取IServiceManager接口的:

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

```
private static IServiceManager getIServiceManager()
{
    if (sServiceManager != null) {
        return sServiceManager;
    }

    // Find the service manager
    sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject())
    return sServiceManager;
}
```

噢，又出现了一个asInterface，看来Java层次和C++层的代码在本质上是一致的。

ServiceManagerNative的asInterface()代码如下:

```
static public IServiceManager asInterface(IBinder obj)
{
    if (obj == null)
    {
        return null;
    }

    IServiceManager in = (IServiceManager)obj.queryLocalInterface(descriptor);
    if (in != null)
    {
        return in;
    }
```

```
    return new ServiceManagerProxy(obj);
}
```

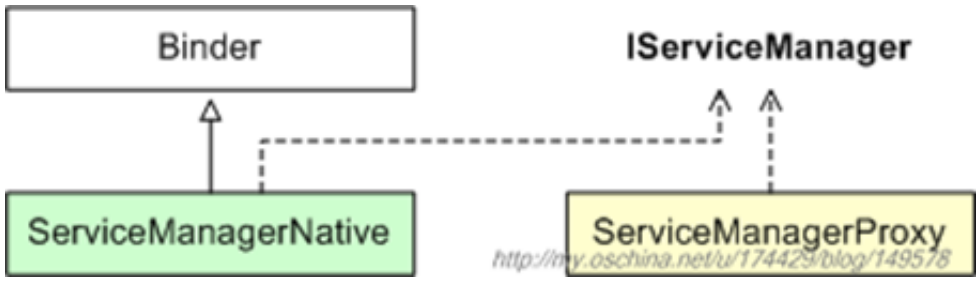
目前我们只需了解，用户进程在调用到getIServiceManager()时，最终会走到return new ServiceManagerProxy(obj)即可。

哎呀，又出现了两个名字：ServiceManagerProxy和ServiceManagerNative。简单地说：

1） ServiceManagerProxy就是IServiceManager代理接口；

2） ServiceManagerNative显得很鸡肋；

它们的继承关系图如下：



下面我们分别来说明。

2.1.1 ServiceManagerProxy就是IServiceManager代理接口

用户要访问Service Manager Service服务，必须先拿到IServiceManager代理接口，而ServiceManagerProxy就是代理接口的实现。这个从前文代码中的new ServiceManagerProxy(obj)一句就可以看出来了。ServiceManagerProxy的构造函数内部会把obj参数记录到mRemote域中：

```
public ServiceManagerProxy(IBinder remote)
{
    mRemote = remote;
}
```

mRemote的定义是：

```
private IBinder mRemote;
```

其实说白了，mRemote的核心包装的就是句柄为0的BpBinder对象，这个应该很容易理解。

日后，当我们通过IServiceManager代理接口访问SMS时，其实调用的就是ServiceManagerProxy的成员函数。比如getService()、checkService()等等。

2.1.2 ServiceManagerNative显得很鸡肋

另一方面，ServiceManagerNative就显得很鸡肋了。

ServiceManagerNative是个抽象类：

```
public abstract class ServiceManagerNative extends Binder implements IServiceManager
```

它继承了Binder，实现了IServiceManager，然而却是个虚有其表的class。它唯一有用的大概就是前文列出的那个静态成员函数asInterface()了，而其他成员函数（像onTransact()）就基本上没什么用。

如果我们花点儿时间在工程里搜索一下ServiceManagerNative，会发现根本找不到它的子类。一个没有子类的抽象类不就是虚有其表吗。到头来我们发现，关于ServiceManagerNative的用法只有一种，就是：

```
ServiceManagerNative.asInterface(BinderInternal.getContextObject());
```

用一下它的asInterface()静态函数而已。

为什么会这样呢？我想这可能是某种历史的遗迹吧。同理，我们看它的onTransact()函数，也会发现里面调用的类似addService()那样的函数，也都是找不到对应的实现体的。当然，因为ServiceManagerNative本身是个抽象类，所以即便它没有实现IServiceManager的addService()等成员函数，也是可以编译通过的。

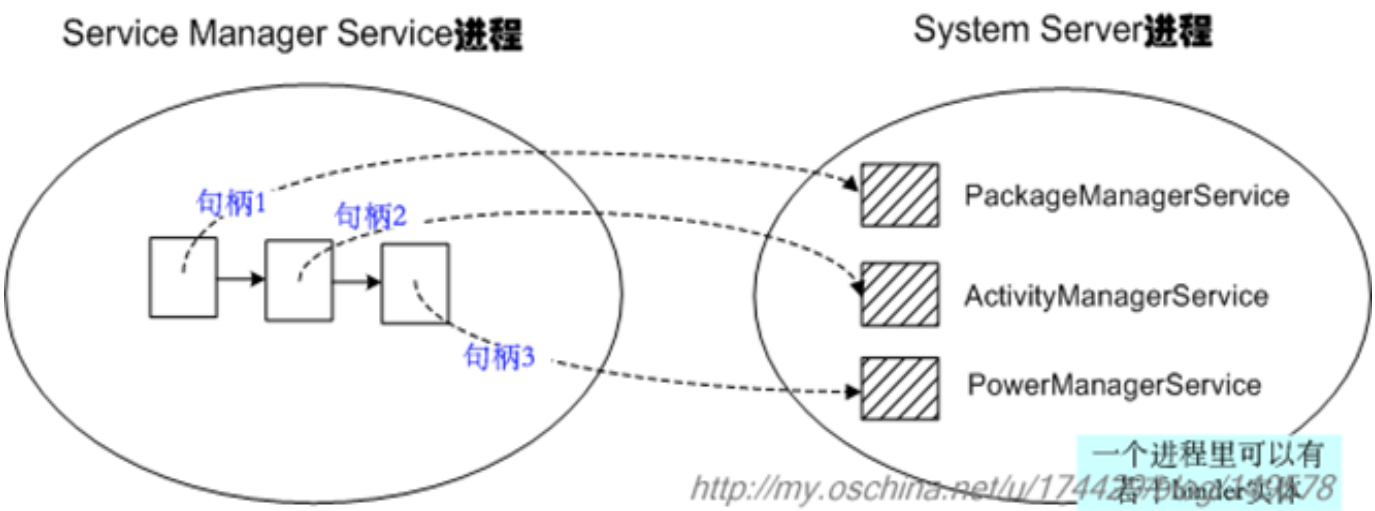
这里透出一个信息，既然Java层的ServiceManagerNative没什么大用处，是不是表示C++层也缺少对应的SMS服务实体呢？在后文我们可以看到，的确是这样的，Service Manager Service在C++层被实现成一个独立的进程，而不是常见的Binder实体。

2.2 通过addService()来注册系统服务

我们还是回过头接着说对于IServiceManager接口的使用吧。最重要的当然是注册系统服务。比如在System Server进程中，是这样注册PowerManagerService系统服务的：

```
public void run()
{
    . . . . . .
    power = new PowerManagerService();
    ServiceManager.addService(Context.POWER_SERVICE, power);
    . . . . . .
```

addService()的第一个参数就是所注册service的名字，比如上面的POWER_SERVICE对应的字符串就是"power"。第二个参数传入的是service Binder实体。Service实体在Service Manager Service一侧会被记录成相应的句柄值，如图：



有关addService()内部机理，我们会在后文讲述，这里先不细说。

2.3 通过getService()来获取某系统服务的代理接口

除了注册系统服务，Service Manager Service的另一个主要工作就是让用户进程可以获取系统service的代理接口，所以其getService()函数就非常重要了。

其实，ServiceManagerProxy中的getService()等成员函数，仅仅是把语义整理进parcel，并通过mRemote将parcel传递到目标端而已。所以我们只看看getService()就行了，其他的函数都大同小异。

```
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();
    reply.recycle();
    data.recycle();
    return binder;
}
```

传递的语义就是GET_SERVICE_TRANSACTION，非常简单。mRemote从本质上看就是句柄为0的BpBinder，所以binder驱动很清楚这些语义将去向何方。

关于Service Manager Service的使用，我们就先说这么多。下面我们要开始探索SMS内部的运作机制了。

3.Service Manager Service的运作机制

3.1 Service Manager Service服务的启动

既然前文说ServiceManagerNative虚有其表，而且没有子类，那么Service Manager Service服务的真正实现代码位于何处呢？答案就在init.rc脚本里。关于init.rc的详细情况，可参考其他阐述Android启动流程的文档，此处不再赘述。

init.rc脚本中，在描述zygote service之前就已经写明service manager service的信息了：

```
service servicemanager /system/bin/servicemanager
    user system
    critical
    onrestart restart zygote
    onrestart restart media
```

可以看到，servicemanager是一种native service。这种native service都是需要用C/C++编写的。Servi

ce Manager Service对应的实现代码位于frameworks/base/cmds/servicemanager/Service_manager.c文件中。这个文件中有每个C程序员都熟悉的main()函数，其编译出的可执行程序就是/system/bin/servicemanager。

另外，还有一个干扰我们视线的cpp文件，名为IServiceManager.cpp，位于frameworks/base/libs/binder/目录中，这个文件里的BnServiceManager应该和前文的ServiceManagerNative类似，它的onTransact()也不起什么作用。

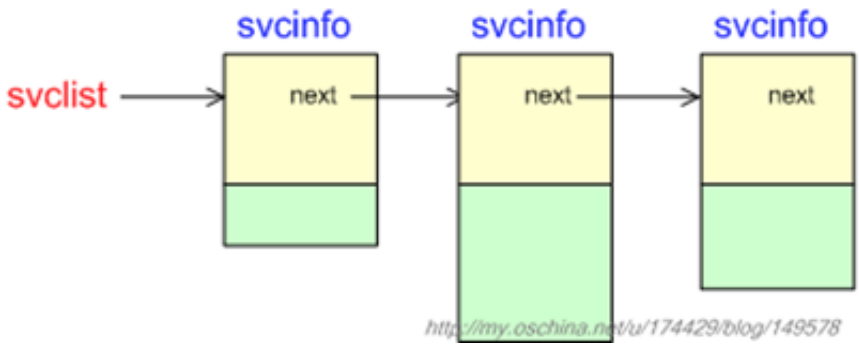
3.2 Service Manager Service是如何管理service句柄的?

在C语言层次，简单地说并不存在一个单独的ServiceManager结构。整个service管理机制都被放在一个独立的进程里了，该进程对应的实现文件就是Service_manager.c。

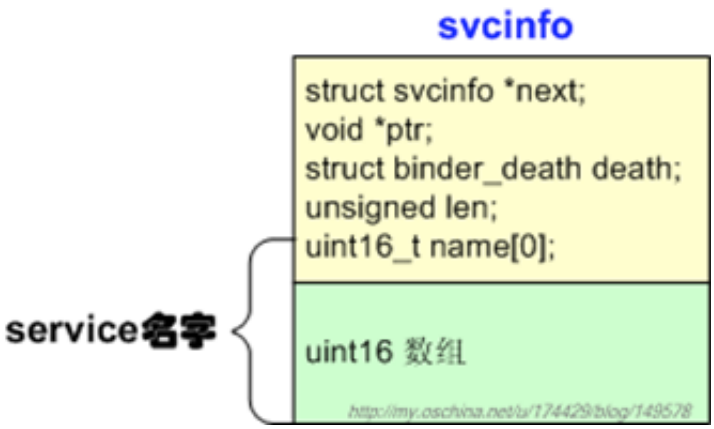
进程里有一个全局性的svclist变量：

```
struct svcinfo *svclist = 0;
```

它记录着所有添加进系统的"service代理"信息，这些信息被组织成一条单向链表，我们不妨称这条链表为"服务向量表"。示意图如下：

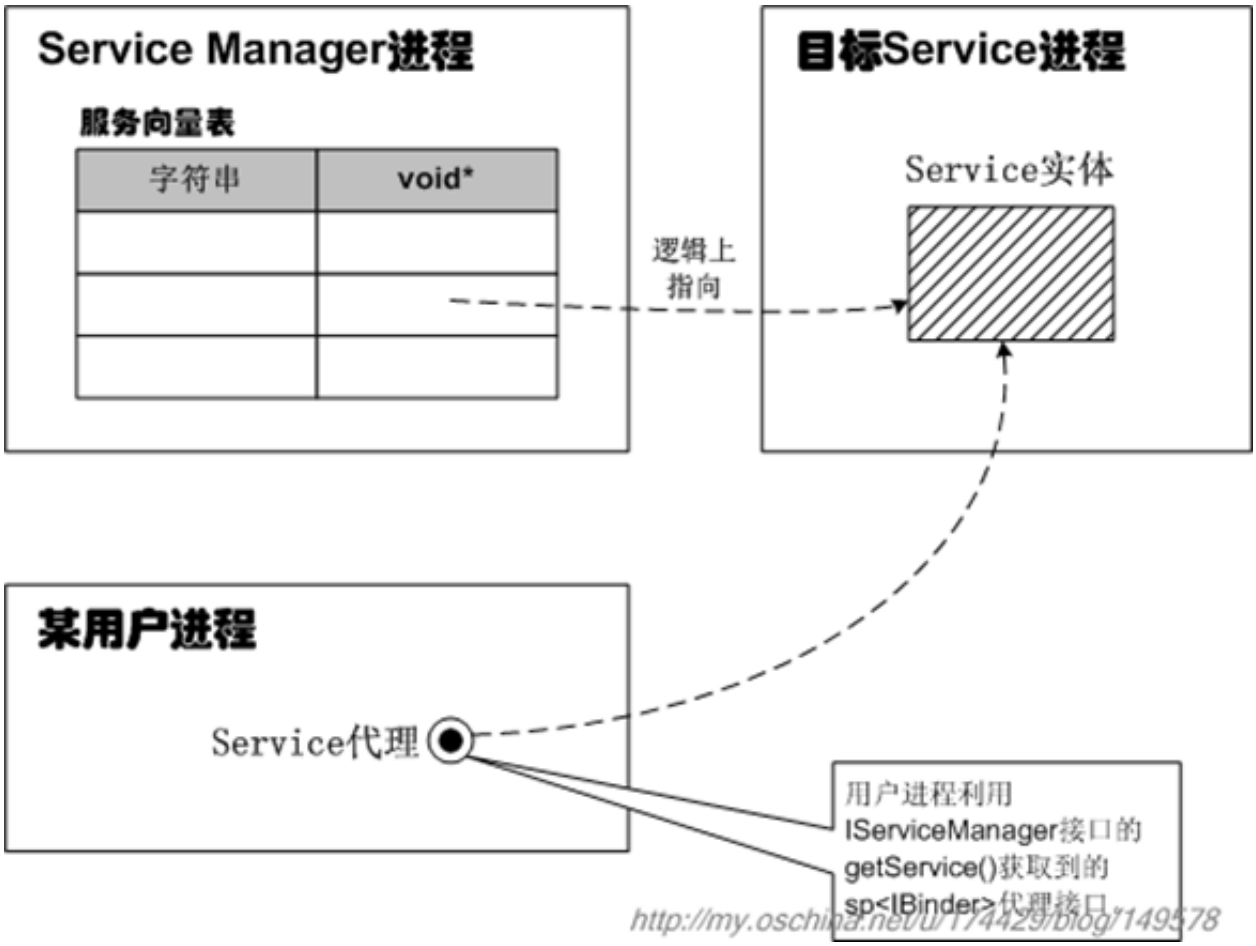


链表节点类型为svcinfo。



因为svcinfo里要记录下service的名字字符串，所以它需要的buffer长度是(len + 1) * sizeof(uint16_t)，记得要留一个'\0'的结束位置。另外，svcinfo的ptr域，实际上记录的就是系统service对应的binder句柄值。

日后，当应用调用getService()获取系统服务的代理接口时，SMS就会搜索这张"服务向量表"，查找是否有节点能和用户传来的服务名匹配，如果能查到，就返回对应的sp<IBinder>，这个接口在远端对应的实体就是"目标Service实体"。如此一来，系统中就会出现如下关系：



3.3 Service Manager Service的主程序（C++层）

要更加深入地了解Service Manager进程的运作，我们必须研究其主程序。参考代码是frameworks\base\cmds\servicemanager\Service_manager.c。

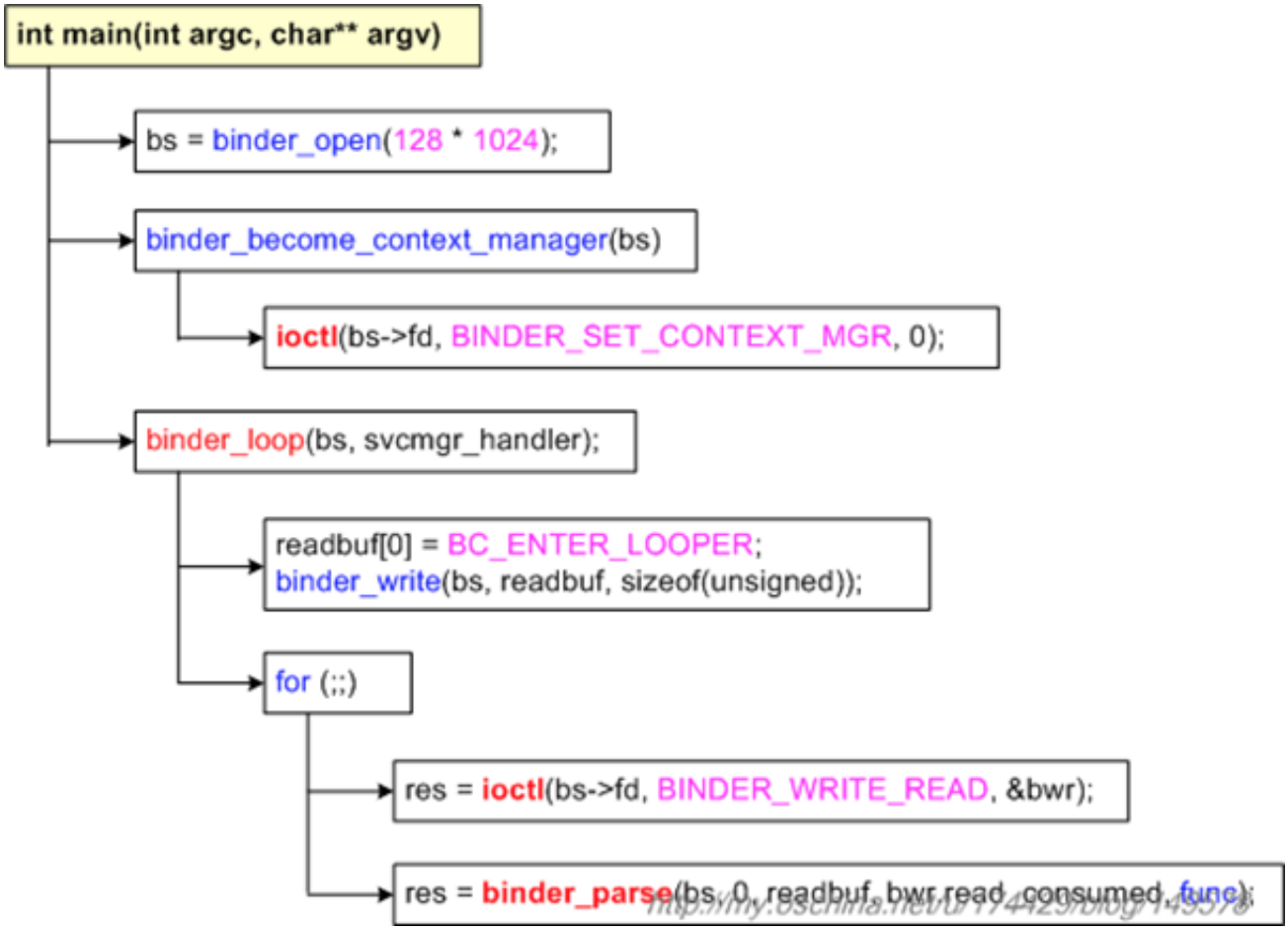
Service_manager.c中的main()函数如下：

```
int main(int argc, char **argv)
{
    struct binder_state *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;

    bs = binder_open(128*1024);

    if (binder_become_context_manager(bs)) {
        ALOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }

    svcmgr_handle = svcmgr;
    binder_loop(bs, svcmgr_handler);
    return 0;
}
```

main()函数一开始就打开了binder驱动，然后调用binder_become_context_manager()让自己成为整个系统中唯一的上下文管理器，其实也就是service管理器啦。接着main()函数调用binder_loop()进入无限循环，不断监听并解析binder驱动发来的命令。

binder_loop()中解析驱动命令的函数是binder_parse()，其最后一个参数func来自于binder_loop()的最后一个参数——svcmgr_handler函数指针。这个svcmgr_handler()应该算是Service Manager Service的核心回调函数了。

为了方便查看，我把main()函数以及其间接调用的ioctl()语句绘制成如下的调用关系图：



下面我们逐个分析其中调用的函数。

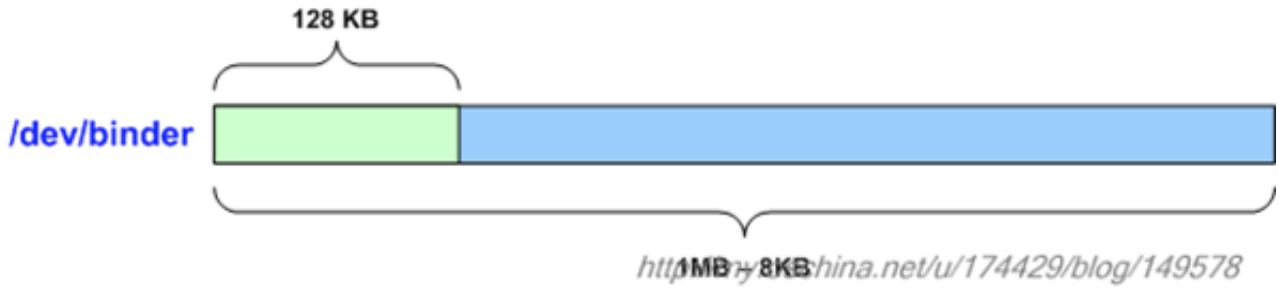
3.3.1 binder_open()

Service Manager Service必须先调用binder_open()来打开binder驱动，驱动文件为"/dev/binder"。binder_open()的代码截选如下：

```
struct binder_state * binder_open(unsigned mapsize)
{
    struct binder_state *bs;

    bs = malloc(sizeof(*bs));
    . . . . . .
    bs->fd = open("/dev/binder", O_RDWR);
    . . . . . .
    bs->mapsize = mapsize;
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
```

```
    . . . . . .
    return bs;
. . . . . .
}
```

binder_open()的参数mapsize表示它希望把binder驱动文件的多少字节映射到本地空间。可以看到，Service Manager Service和普通进程所映射的binder大小并不相同。它把binder驱动文件的128K字节映射到内存空间，而普通进程则会映射binder文件里的BINDER_VM_SIZE（即1M减去8K）字节。



具体的映射动作由mmap()一句完成，该函数将binder驱动文件的一部分映射到进程空间。mmap()的函数原型如下：

```
void* mmap ( void * addr , size_t len , int prot , int flags , int fd , off_t offset );
```

该函数会把"参数fd所指代的文件"中的一部分映射到进程空间去。这部分文件内容以offset为起始位置，以len为字节长度。其中，参数offset表明从文件起始处开始算起的偏移量。参数prot表明对这段映射空间的访问权限，可以是PROT_READ（可读）、PROT_WRITE （可写）、PROT_EXEC （可执行）、PROT_NONE（不可访问）。参数addr用于指出文件应被映射到进程空间的起始地址，一般指定为空指针，此时会由内核来决定起始地址。

binder_open()的返回值类型为binder_state*，里面记录着刚刚打开的binder驱动文件句柄以及mmap()映射到的最终目标地址。

```
struct binder_state
{
    int fd;
    void *mapped;
    unsigned mapsize;
};
```

以后，SMS会不断读取这段映射空间，并做出相应的动作。

3.3.2 binder_become_context_manager()

我们前面已经说过，binder_become_context_manager()的作用是让当前进程成为整个系统中唯一的上下文管理器，即service管理器。其代码非常简单：

```
int binder_become_context_manager(struct binder_state *bs)
{
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

仅仅是把BINDER_SET_CONTEXT_MGR发送到binder驱动而已。驱动中与ioctl()对应的binder_ioctl()是这样处理的：

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
     int ret;
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread;
    unsigned int size = _IOC_SIZE(cmd);
    void __user *ubuf = (void __user *)arg;

    . . . . . .
    . . . . . .
    case BINDER_SET_CONTEXT_MGR:
        . . . . . .
        . . . . . .
            binder_context_mgr_uid = current->cred->euid;

        binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
        if (binder_context_mgr_node == NULL)
        {
            ret = -ENOMEM;
            goto err;
        }
        binder_context_mgr_node->local_weak_refs++;
        binder_context_mgr_node->local_strong_refs++;
        binder_context_mgr_node->has_strong_ref = 1;
        binder_context_mgr_node->has_weak_ref = 1;
        break;
    . . . . . .
    . . . . . .
}
```

代码的意思很明确，要为整个系统的上下文管理器专门生成一个binder_node节点，并记入静态变量binder_context_mgr_node。

我们在这里多说两句，一般情况下，应用层的每个binder实体都会在binder驱动层对应一个binder_node节点，然而binder_context_mgr_node比较特殊，它没有对应的应用层binder实体。在整个系统里，它是如此特殊，以至于系统规定，任何应用都必须使用句柄0来跨进程地访问它。现在大家可以回想一下前文在获取SMS接口时说到的那句new BpBinder(0)，是不是能加深一点儿理解。

3.3.3 binder_loop()

我们再回到SMS的main()函数。

接下来的binder_loop()会先向binder驱动发出了BC_ENTER_LOOPER命令，接着进入一个for循环不断调用ioctl()读取发来的数据，接着解析这些数据。参考代码在：

【frameworks/base/cmds/servicemanager/Binder.c】（注意！这个Binder.c文件不是binder驱动层那个Binder.c文件噢。）

```
void binder_loop(struct binder_state *bs, binder_handler func)
```

```
{
    int res;
    struct binder_write_read bwr;
    unsigned readbuf[32];

    bwr.write_size = 0;
    bwr.write_consumed = 0;
    bwr.write_buffer = 0;

    readbuf[0] = BC_ENTER_LOOPER;
    binder_write(bs, readbuf, sizeof(unsigned));

    for (;;)
    {
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (unsigned) readbuf;

        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);

        if (res < 0) {
            LOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
            break;
        }

        res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
        if (res == 0) {
            LOGE("binder_loop: unexpected reply?!\n");
            break;
        }
        if (res < 0) {
            LOGE("binder_loop: io error %d %s\n", res, strerror(errno));
            break;
        }
    }
}
```

注意binder_loop()的参数func，它的值是svcmgr_handler()函数指针。而且这个参数会进一步传递给binder_parse()。

3.3.3.1 BC_ENTER_LOOPER

binder_loop()中发出BC_ENTER_LOOPER命令的目的，是为了告诉binder驱动“本线程要进入循环状态了”。在binder驱动中，凡是用到跨进程通信机制的线程，都会对应一个binder_thread节点。这里的BC_ENTER_LOOPER命令会导致这个节点的looper状态发生变化：

```
thread->looper |= BINDER_LOOPER_STATE_ENTERED;
```

有关binder_thread的细节，也会在阐述Binder驱动一节进行说明。

3.3.3.2 binder_parse()

在binder_loop()进入for循环之后，最显眼的就是那句binder_parse()了。binder_parse()负责解析从binder驱动读来的数据，其代码截选如下：

```
int binder_parse(struct binder_state *bs, struct binder_io *bio,
                 uint32_t *ptr, uint32_t size, binder_handler func)
{
    int r = 1;
    uint32_t *end = ptr + (size / 4);

    while (ptr < end)
    {
        uint32_t cmd = *ptr++;
        . . . . . .
        case BR_TRANSACTION:
        {
            struct binder_txn *txn = (void *) ptr;
            if ((end - ptr) * sizeof(uint32_t) < sizeof(struct binder_txn)) {
                ALOGE("parse: txn too small!\n");
                return -1;
            }
            binder_dump_txn(txn);
            if (func)
            {
                unsigned rdata[256/4];
                struct binder_io msg;
                struct binder_io reply;
                int res;

                bio_init(&reply, rdata, sizeof(rdata), 4);
                bio_init_from_txn(&msg, txn);
                res = func(bs, txn, &msg, &reply);
                binder_send_reply(bs, &reply, txn->data, res);
            }
            ptr += sizeof(*txn) / sizeof(uint32_t);
            break;
        }
        . . . . . .
        . . . . . .
    }

    return r;
}
```

从前文的代码我们可以看到，binder_loop()声明了一个128字节的buffer（即unsigned readbuf[32]），每次用BINDER_WRITE_READ命令从驱动读取一些内容，并传入binder_parse()。

binder_parse()在合适的时机，会回调其func参数（binder_handler func）指代的回调函数，即前文说到的svcmgr_handler()函数。

binder_loop()就这样一直循环下去，完成了整个service manager service的工作。

4.Service Manager Service解析收到的命令

现在，我们专门用一个小节来说说Service Manager Service内循环解析命令时的一些细节。我们要确定binder_loop()从驱动侧读到的数据到底如何解析？我们重贴一下binder_parse()的声明部分：

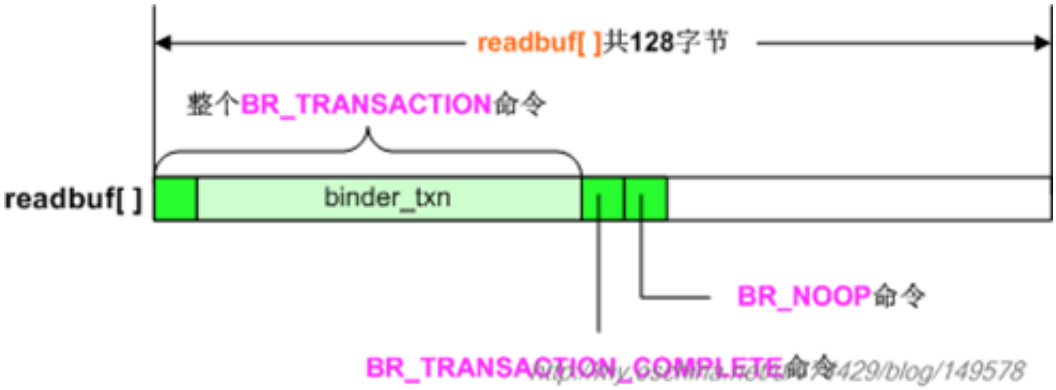
```
int binder_parse(struct binder_state *bs,
                 struct binder_io *bio,
                 uint32_t *ptr,
                 uint32_t size,
                 binder_handler func)
```

之前利用ioctl()读取到的数据都记录在第三个参数ptr所指的缓冲区中，数据大小由size参数记录。其实这个buffer就是前文那个128字节的buffer。

从驱动层读取到的数据，实际上是若干BR命令。每个BR命令是由一个命令号(uint32)以及若干相关数据组成的，不同BR命令的长度可能并不一样。如下表所示：

BR命令	需进一步读取的**uint32**数
BR_NOOP	0
BR_TRANSACTION_COMPLETE	0
BR_INCREFS	2
BR_ACQUIRE	2
BR_RELEASE	2
BR_DECREFS	2
BR_TRANSACTION	sizeof(**binder_txn**) / sizeof(uint32_t)
BR_REPLY	sizeof(**binder_txn**) / sizeof(uint32_t)
BR_DEAD_BINDER	1
BR_FAILED_REPLY	0
BR_DEAD_REPLY	0

每次ioctl()操作所读取的数据，可能会包含多个BR命令，所以binder_parse()需要用一个while循环来解析buffer中所有的BR命令。我们随便画个示意图，如下：



图中的buffer中含有3条BR命令，分别为BR_TRANSACTION、BR_TRANSACTION_COMPLETE、BR_NOOP命令。一般而言，我们最关心的就是BR_TRANSACTION命令啦，因此前文截选的binder_parse()代码，主要摘录了处理BR_TRANSACTION命令的代码，该命令的命令号之后跟着的是一个binder_txn结构。现在我们来详细看这个结构。

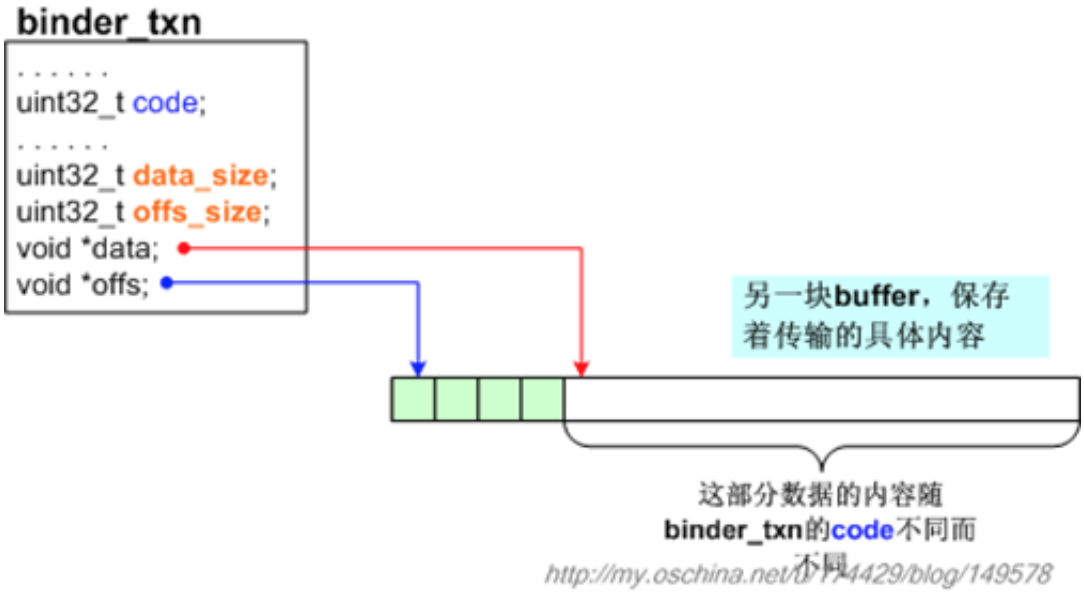
4.1 解析binder_txn信息

binder_txn的定义如下：

【frameworks/base/cmds/servicemanager/Binder.h】

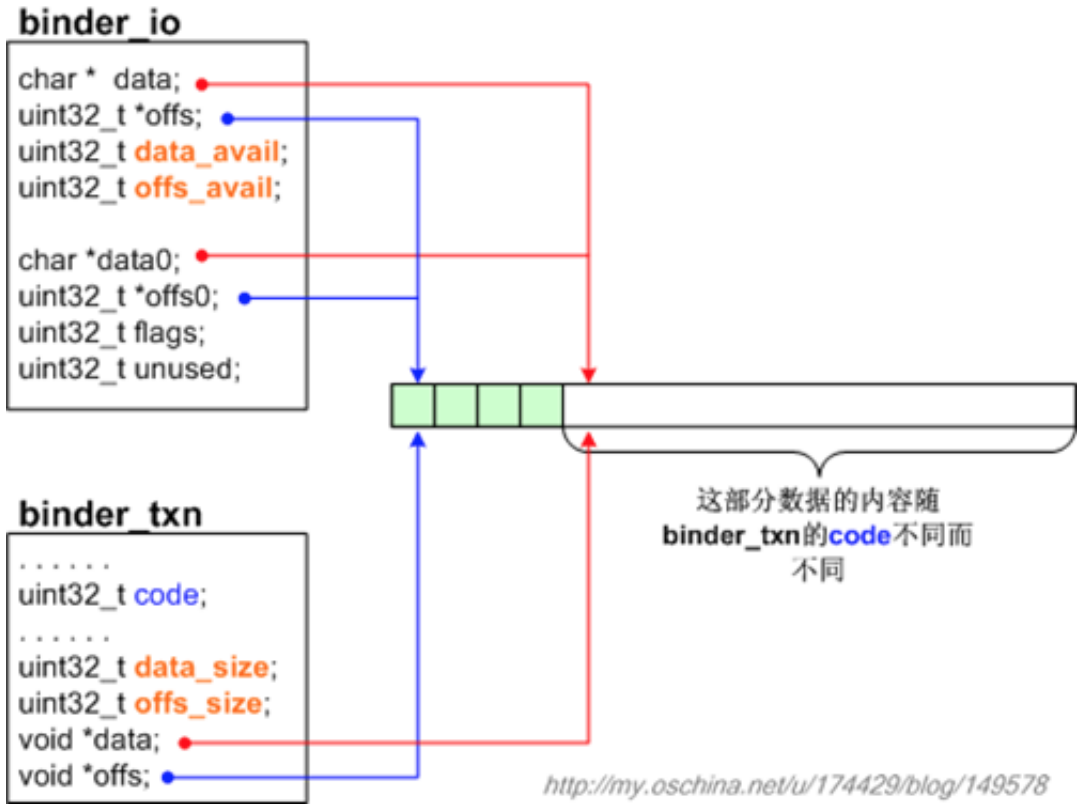
```
struct binder_txn
{
    void *target;
    void *cookie;
    uint32_t code;          // 所传输的语义码
    uint32_t flags;

    uint32_t sender_pid;
    uint32_t sender_euid;

    uint32_t data_size;
    uint32_t offs_size;
    void *data;
    void *offs;
};
```

binder_txn说明了transaction到底在传输什么语义，而语义码就记录在其code域中。不同语义码需要携带的数据也是不同的，这些数据由data域指定。示意图如下：



简单地说，我们从驱动侧读来的binder_txn只是一种"传输控制信息"，它本身并不包含传输的具体内容，而只是指出具体内容位于何处。现在，工作的重心要转到如何解析传输的具体内容了，即binder_txn的data域所指向的那部分内容。

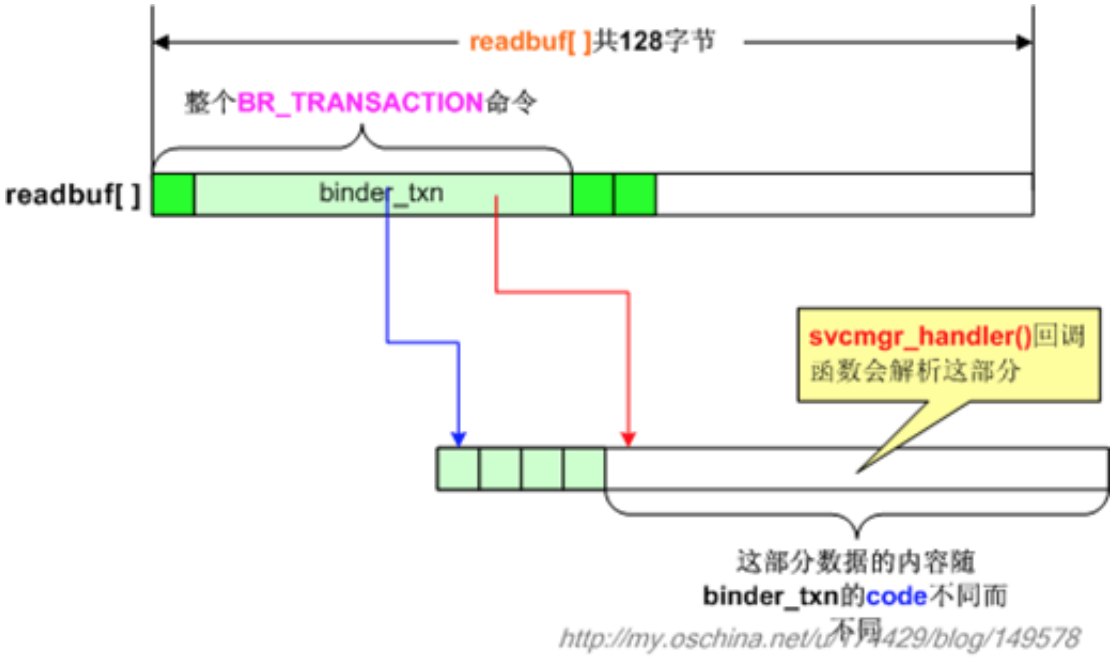
为了解析具体内容，binder_parse()声明了两个类型为binder_io的局部变量：msg和reply。从binder_io这个类型的名字，我们就可以看出要用它来读取binder传递来的数据了。其实，为了便于读取binder_io所指代的内容，工程提供了一系列以bio_打头的辅助函数。在读取实际数据之前，我们必须先调用bio_init_from_txn()，把binder_io变量（比如msg变量）和binder_txn所指代的缓冲区联系起来。示意图如下：



从图中可以看到，binder_io结构已经用binder_txn结构初始化了自己，以后我们就可以调用类似bio_get_uint32()、bio_get_string16()这样的函数，来读取这块buffer了。

4.2 svcmgr_handler()回调函数

初始化后的binder_io数据，就可以传给svcmgr_handler()回调函数做进一步的解析了。



此时我们可以调用下面这些辅助函数进行读写：

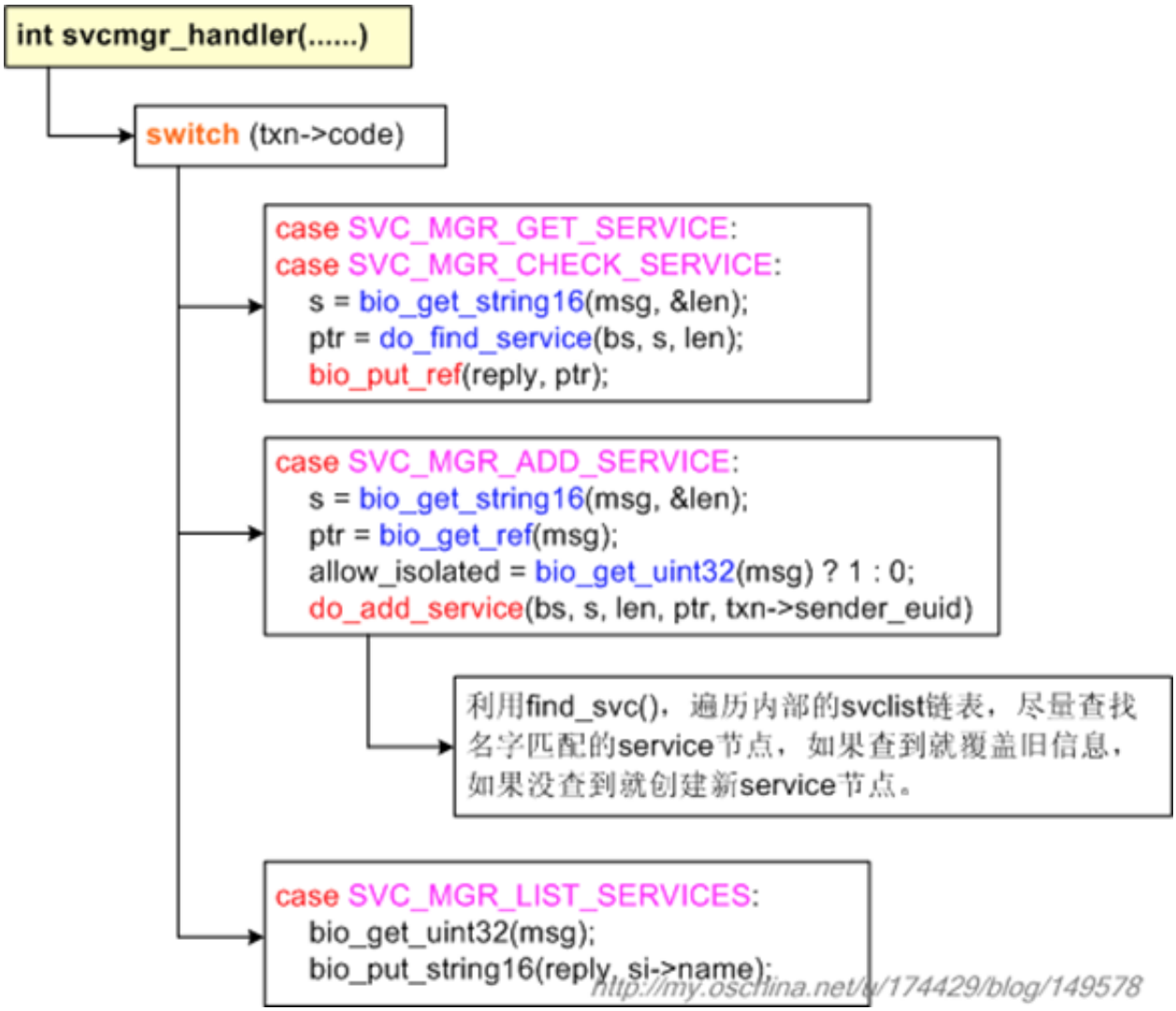
```
void bio_put_uint32(struct binder_io *bio, uint32_t n)
void bio_put_obj(struct binder_io *bio, void *ptr)
uint32_t bio_get_uint32(struct binder_io *bio)
uint16_t *bio_get_string16(struct binder_io *bio, unsigned *sz)
void *bio_get_ref(struct binder_io *bio)
. . . . . .
```

其中，bio_get_xxx()函数在读取数据时，是以binder_io的data域为读取光标的，每读取一些数据，data值就会增加，并且data_avail域会相应减少。而data0域的值则保持不变，一直指着数据区最开始的位置，它的作用就是作为计算偏移量的基准值。

bio_get_uint32()非常简单，会从binder_io.data所指的地方，读取4个字节的内容。bio_get_string16()就稍微复杂一点儿，先要读取一个32bits的整数，这个整数值就是字符串的长度，因为字符串都要包含最后一个'\0'，所以需要读取((len + 1) * sizeof(uint16_t))字节的内容。还有一个是bio_get_ref()，它会读取一个binder_object结构。binder_object的定义如下：

```
struct binder_object
{
    uint32_t type;
    uint32_t flags;
    void *pointer;
    void *cookie;
};
```

在svcmgr_handler()函数中，一个传输语义码（txn->code）可能会对应几次bio_get操作，比如后文我们要说的SVC_MGR_ADD_SERVICE语义码。具体情况请大家参考svcmgr_handler()的代码。svcmgr_handler()的调用示意图如下：



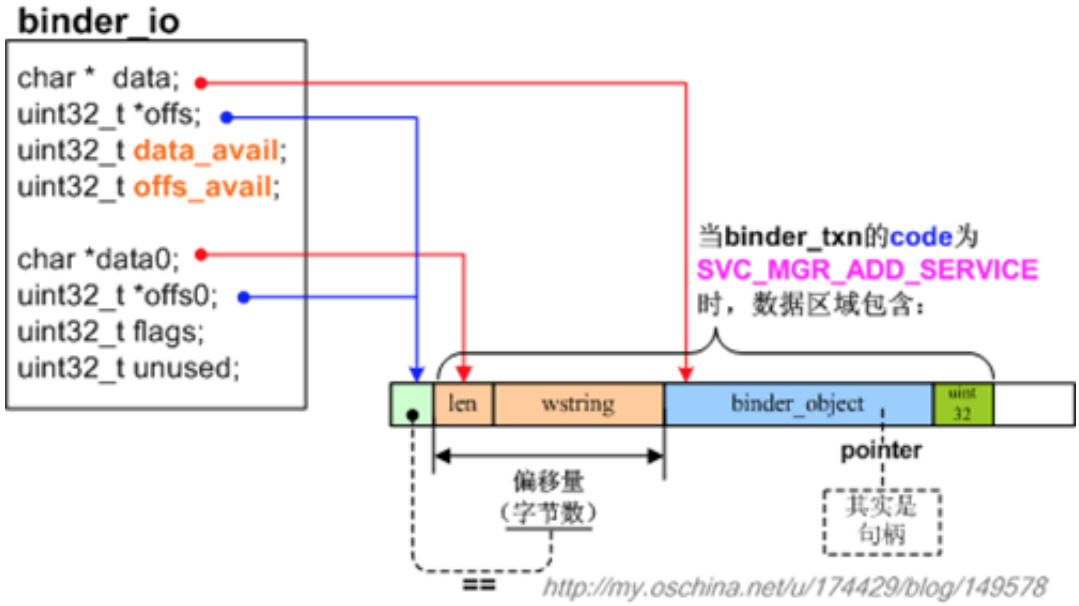
4.2.1 如何解析add service

我们先研究add service的动作。前文我们已经介绍过，service manager进程里有一个全局性的svclist变量，记录着所有添加进系统的"service代理"信息，这些信息被组织成一条单向链表，即"服务向量表"。现在我们要看service manager是如何向这张表中添加新节点的。

假设某个服务进程调用Service Manager Service接口，向其注册service。这个注册动作到最后就会走到svcmgr_handler()的case SVC_MGR_ADD_SERVICE分支。此时会先获取三个数据，而后再调用do_add_service()函数，代码如下：

```
uint16_t *    s;
void *        ptr;
. . . . . .
s = bio_get_string16(msg, &len);
ptr = bio_get_ref(msg);
allow_isolated = bio_get_uint32(msg) ? 1 : 0;
do_add_service(bs, s, len, ptr, txn->sender_euid);
```

也就是说，当binder_txn的code为SVC_MGR_ADD_SERVICE时，binder_txn所指的数据区域中应该包含一个字符串，一个binder对象以及一个uint32数据。示意图如下：



其中那个binder_object，记录的就是新注册的service所对应的代理信息。此时binder_object的pointer域实际上已经不是指针值了，而是一个binder句柄值。

do_add_service()的函数截选如下：

```
struct svcinfo *svclist = 0;     // 核心service链表（即服务向量表）

int do_add_service(struct binder_state *bs, uint16_t *s, unsigned len,
                        void *ptr, unsigned uid)
{
    struct svcinfo *si;

    if (!ptr || (len == 0) || (len > 127))
        return -1;

    if (!svc_can_register(uid, s)) {
        ALOGE("add_service('%s',%p) uid=%d - PERMISSION DENIED\n",
             str8(s), ptr, uid);
        return -1;
    }
    si = find_svc(s, len);
    if (si) {
        if (si->ptr) {
            svcinfo_death(bs, si);
        }
        si->ptr = ptr;
    } else {
        // 新创建一个svcinfo节点。
        si = malloc(sizeof(*si) + (len + 1) * sizeof(uint16_t));
        if (!si) {
            return -1;
        }
        si->ptr = ptr;     // 在svcinfo节点的ptr域中，记录下service对应的binder句柄值
        si->len = len;
        memcpy(si->name, s, (len + 1) * sizeof(uint16_t));
        si->name[len] = '\0';
        si->death.func = svcinfo_death;
        si->death.ptr = si;
```

```
        // 把新节点插入svclist链表
        si->next = svclist;
        svclist = si;
    }

    binder_acquire(bs, ptr);
    binder_link_to_death(bs, ptr, &si->death);
    return 0;
}
```

现在我们来解读这部分代码。首先，并不是随便找个进程就能向系统注册service噢。do_add_service()函数一开始先调用svc_can_register()，判断发起端是否可以注册service。如果不可以，do_add_service()就返回-1值。svc_can_register()的代码如下：

```
int svc_can_register(unsigned uid, uint16_t *name)
{
    unsigned n;

    if ((uid == 0) || (uid == AID_SYSTEM))
        return 1;

    for (n = 0; n < sizeof(allowed) / sizeof(allowed[0]); n++)
        if ((uid == allowed[n].uid) && str16eq(name, allowed[n].name))
            return 1;

    return 0;
}
```

上面的代码表示，如果发起端是root进程或者system server进程的话，是可以注册service的，另外，那些在allowed[]数组中有明确记录的用户进程，也是可以注册service的，至于其他绝大部分普通进程，很抱歉，不允许注册service。在以后的软件开发中，我们有可能需要编写新的带service的用户进程（uid不为0或AID_SYSTEM），并且希望把service注册进系统，此时不要忘了修改allowed[]数组。下面是allowed[]数组的一部分截选：

```
static struct {
    unsigned uid;
    const char *name;
} allowed[] = {
    { AID_MEDIA, "media.audio_flinger" },
    { AID_MEDIA, "media.player" },
    { AID_MEDIA, "media.camera" },
    . . . . . .
```

接下来，do_add_service()开始尝试在service链表里查询对应的service是否已经添加过了。如果可以查到，那么就不用生成新的service节点了。否则就需要在链表起始处再加一个新节点。节点类型为svcinfo。请注意上面代码的si->ptr = ptr一句，此时的ptr参数其实来自于前文所说的binder_object的pointer域。

为了说明问题，我们重新列一下刚刚的case SVC_MGR_ADD_SERVICE代码：

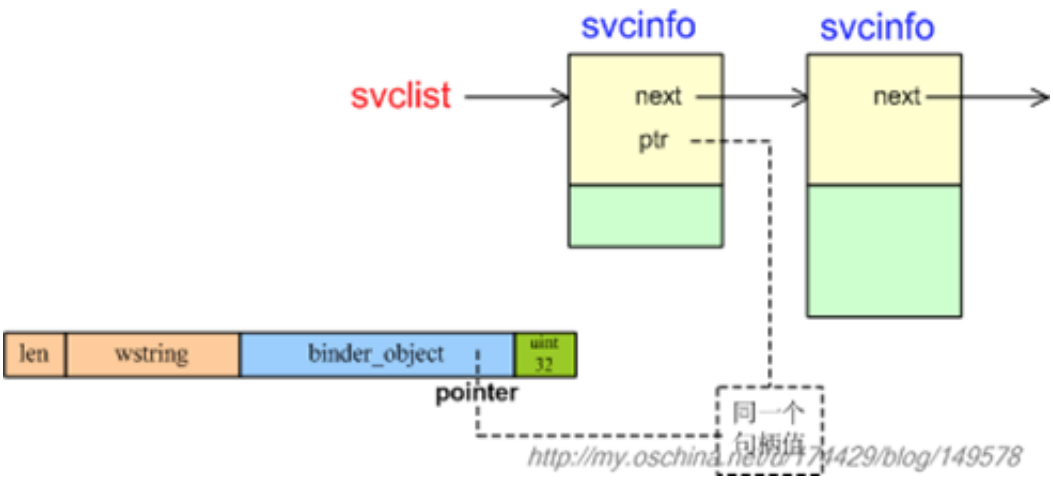
```
case SVC_MGR_ADD_SERVICE:
    s = bio_get_string16(msg, &len);
```

```
    ptr = bio_get_ref(msg);
    allow_isolated = bio_get_uint32(msg) ? 1 : 0;
    if (do_add_service(bs, s, len, ptr, txn->sender_euid, allow_isolated))
        return -1;
    break;
```

那个ptr来自于bio_get_ref(msg)，而bio_get_ref()的实现代码如下：

```
void *bio_get_ref(struct binder_io *bio)
{
    struct binder_object *obj;

    obj = _bio_get_obj(bio);
    if (!obj)
        return 0;

    if (obj->type == BINDER_TYPE_HANDLE)
        return obj->pointer;

    return 0;
}
```

因为现在是要向service manager注册服务，所以obj->type一定是BINDER_TYPE_HANDLE，也就是说会返回binder_object的pointer域。这个域的类型虽为void*，实际上换成uint32可能更合适。通过这个binder句柄值，我们最终可以找到远端的具体service实体。



4.2.2 如何解析get service

现在我们接着来看get service动作。我们知道，在service被注册进service manager之后，其他应用都可以调用ServiceManager的getService()来获取相应的服务代理，并调用代理的成员函数。这个getService()函数最终会向service manager进程发出SVC_MGR_GET_SERVICE命令，并由svcmgr_handler()函数这样处理：

```
switch(txn->code)
{
case SVC_MGR_GET_SERVICE:
case SVC_MGR_CHECK_SERVICE:
    s = bio_get_string16(msg, &len);
```

```
    ptr = do_find_service(bs, s, len, txn->sender_euid);
    if (!ptr)
        break;
    bio_put_ref(reply, ptr);
    return 0;
```

一开始从msg中读取希望get的服务名，然后调用do_find_service()函数查询服务名对应的句柄值，最后把句柄值写入reply。do_find_service()的代码如下：

```
void *do_find_service(struct binder_state *bs, uint16_t *s, unsigned len, unsigned uid)
{
    struct svcinfo *si;
    si = find_svc(s, len);

    if (si && si->ptr)
    {
        if (!si->allow_isolated)
        {
            unsigned appid = uid % AID_USER;
            if (appid >= AID_ISOLATED_START && appid <= AID_ISOLATED_END)
            {
                return 0;
            }
        }
        return si->ptr;     // 返回service代理的句柄!
    }
    else
    {
        return 0;
    }
}
```

可以看到，do_find_service()返回的就是所找到的服务代理对应的句柄值（si->ptr）。而svcmgr_handler()在拿到这个句柄值后，会把它写入reply对象：

```
bio_put_ref(reply, ptr);
```

bio_put_ref()的代码如下：

```
void bio_put_ref(struct binder_io *bio, void *ptr)
{
    struct binder_object *obj;

    if (ptr)
        obj = bio_alloc_obj(bio);
    else
        obj = bio_alloc(bio, sizeof(*obj));

    if (!obj)
        return;

    obj->flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
    obj->type = BINDER_TYPE_HANDLE;
    obj->pointer = ptr;
```

```
    obj->cookie = 0;
}
```

bio_alloc_obj()一句说明会从reply所关联的buffer中划分出一个binder_object区域，然后开始对这个区域写值。于是BINDER_TYPE_HANDLE赋给了obj->type，句柄值赋给了obj->pointer。另外，reply所关联的buffer只是binder_parse()里的局部数组噢：

```
unsigned rdata[256/4];
```

大家应该还记得svcmgr_handler()是被binder_parse()回调的，当svcmgr_handler()返回后，会接着把整理好的reply对象send出去：

```
                bio_init(&reply, rdata, sizeof(rdata), 4);
                bio_init_from_txn(&msg, txn);
                res = func(bs, txn, &msg, &reply);
                binder_send_reply(bs, &reply, txn->data, res);
```

也就是把查找到的信息，发送给发起查找的一方。

binder_send_reply()的代码如下：

```
void binder_send_reply(struct binder_state *bs, struct binder_io *reply,
                       void *buffer_to_free, int status)
{
    struct
    {
        uint32_t cmd_free;
        void *buffer;
        uint32_t cmd_reply;
        struct binder_txn txn;
    } __attribute__((packed)) data;

    data.cmd_free = BC_FREE_BUFFER;
    data.buffer = buffer_to_free;
    data.cmd_reply = BC_REPLY;
    data.txn.target = 0;
    data.txn.cookie = 0;
    data.txn.code = 0;

    if (status)
    {
        data.txn.flags = TF_STATUS_CODE;
        data.txn.data_size = sizeof(int);
        data.txn.offs_size = 0;
        data.txn.data = &status;
        data.txn.offs = 0;
    }
    else
    {
        data.txn.flags = 0;
        data.txn.data_size = reply->data - reply->data0;
```

```
        data.txn.offs_size = ((char*) reply->offs) - ((char*) reply->offs0);
        data.txn.data = reply->data0;
        data.txn.offs = reply->offs0;
    }
    binder_write(bs, &data, sizeof(data));
}
```

观察代码中最后那几行，看来还是在摆弄reply所指代的那个buffer。当初binder_parse()在创建reply对象之时，就给它初始化了一个局部buffer，即前文所说的unsigned rdata[256/4]，在svcmgr_handler()中又调用bio_put_ref()在这个buffer中开辟了一块binder_object，并在其中赋予了ptr句柄。现在终于要向binder驱动传递reply信息了，此时调用的binder_write()的代码如下：

```
int binder_write(struct binder_state *bs, void *data, unsigned len)
{
    struct binder_write_read bwr;
    int res;
    bwr.write_size = len;
    bwr.write_consumed = 0;
    bwr.write_buffer = (unsigned) data;
    bwr.read_size = 0;
    bwr.read_consumed = 0;
    bwr.read_buffer = 0;
    res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
    if (res < 0) {
        fprintf(stderr,"binder_write: ioctl failed (%s)\n",
                strerror(errno));
    }
    return res;
}
```

噢，又见ioctl()，数据就在bwr.write_buffer，数据里打出了两个binder命令，BC_FREE_BUFFER和BC_REPLY。

这些数据被传递给get service动作的发起端，虽然这些数据会被binder驱动做少许修改，不过语义是不会变的，于是发起端就获得了所查service的合法句柄。

5.小结

至此，有关ServiceManager的基本知识就大体交代完毕了，文行于此，暂告段落。必须承认，受限于个人的认识和文章的篇幅，我不可能涉及其中所有的细节，这里只能摘其重点进行阐述。如果以后又发现什么有趣的东西，我还会补充进来。