開源中國社區
開源項目發現、使用和交流平台

- 項目
- 討論
- 代碼
- 資訊
- 翻譯
- 博客
- Android
- 招聘

當前訪客身份：遊客[ 登錄 | 加入開源中國 ] 你有0新留言

在 27048 款开源软件中搜

軟件

軟件

搜索

悠然紅茶 關注此人

關注(0) 粉絲(18) 積分(6)
求真求是
. 發送留言 . 請教問題

**博客分類**

- Android Frameworks 4.x (5)
- 日常記錄(0)
- 轉貼的文章(0)

**閱讀排行**

1. 1. 紅茶一杯話Binder（傳輸機制篇 上）
2. 2. 紅茶一杯話Binder（傳輸機制篇 中）
3. 3. 紅茶一杯話Binder（ServiceManager篇）
4. 4. 紅茶一杯話Binder（初始篇）
5. 5. AlarmManager研究

**最新評論**

- @Jessie0227：寫的太好了!!請問何時會有下一篇呢(期待中) 查看»
- @公子無憂：請教個問題,BC和BR的命令是什麼關係,分別什麼時候... 查看»
- @xkk609：分析比較深入。 查看»
- @RenKaidi：不明覺厲！ 查看»
- @enull：Mark一下，自學中，感謝。 查看»
- @xway：準備空下來的時候學習下Android開發，[illegible]認真的... 查看»
- @徐慶-neo：贊一個，非常好的文章 查看»
- @翠屏阿姨：我是沙發，曾經看過沒看懂，今天趁著這篇文章再看... 查看»
- @simonws：fucking source code 查看»
- @悠然紅茶：引用來自「simonws」的評論你是怎麼研究的？無他... 查看»

**訪客統計**

- **今日訪問： 3**
- **昨日訪問： 7**
- **本週訪問： 19**
- **本月訪問： 10**
- **所有訪問： 2285**

空間 » 博客 » Android Frameworks 4.x »博客正文

# 紅茶一杯話Binder（ServiceManager篇）



# 紅茶一杯話Binder

## （ServiceManager篇）

侯亮

# 1.先說一個大概

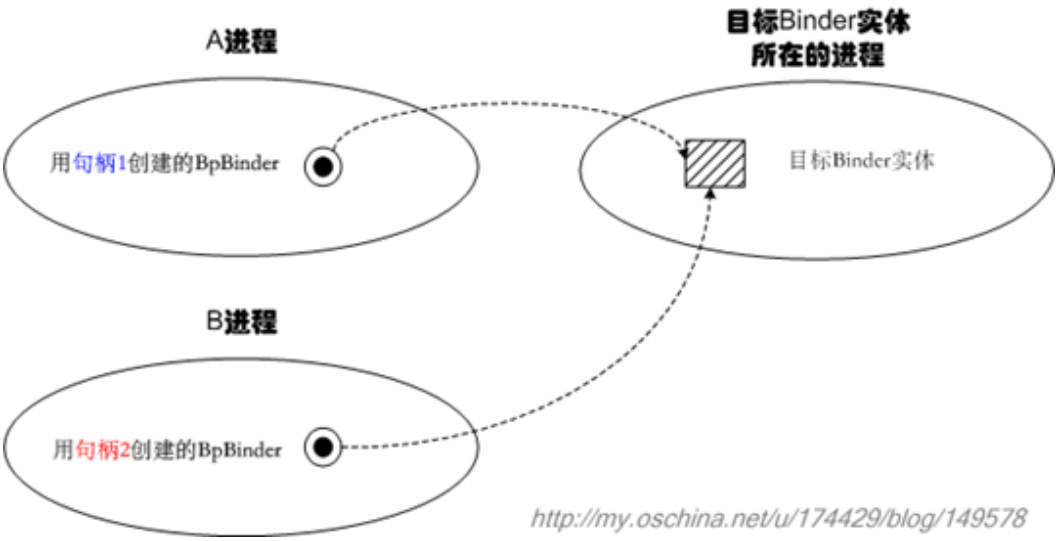
Android平台的一個基本設計理念是構造一個相對平坦的功能集合，這些功能可能會身處於不同的進程中，然而卻可以高效地整合到一起，實現不同的用戶需求。這就必須打破過去各個孤立App所形成的天然藩籬。為此，Android提供了Binder機制。

在Android中，系統提供的服務被包裝成一個個系統級service，這些service往往會在設備啟動之時添加進Android系統。在上一篇文檔中，我們已經瞭解了BpBinder和BBinder的概念，而service實體的底層說到底就是一個BBinder實體。

我們知道，如果某個程序希望享受系統提供的服務，它就必須調用系統提供的外部接口，向系統發出相應的請求。因此，Android中的程序必須先拿到和某個系統service對應的代理接口，然後才能通過這個接口，享受系統提供的服務。說白了就是我們得先拿到一個和目標service對應的合法BpBinder。

然而，該怎麼獲取和系統service對應的代理接口呢？Android是這樣設計的：先啟動一個特殊的系統服務，叫作Service Manager Service（簡稱SMS），它的基本任務就是管理其他系統服務。其他系統服務在系統啟動之時，就會向SMS註冊自己，於是SMS先記錄下與那個service對應的名字和句柄值。有了句柄值就可以用來創建合法的BpBinder了。只不過在實際的代碼中，SMS並沒有用句柄值創建出BpBinder，這個其實沒什麼，反正指代目標service實體的目的已經達到了。後續當某程序需要享受某系統服務時，它必須先以「特定手法」獲取SMS代理接口，並經由這個接口查詢出目標service對應的合法Binder句柄，然後再創建出合法的BpBinder對象。

在此，我們有必要交代一下「Binder句柄」的作用。句柄說穿了是個簡單的整數值，用來告訴Binder驅動我們想找的目標Binder實體是哪個。但是請注意，句柄只對發起端進程和Binder驅動有意義，A進程的句柄直接拿到B進程，是沒什麼意義的。也就是說，不同進程中指代相同Binder實體的句柄值可能是不同的。示意圖如下：




SMS記錄了所有系統service所對應的Binder句柄，它的核心功能就是維護好這些句柄值。後續，

當用戶進程需要獲取某個系統service的代理時，SMS就會在內部按service名查找到合適的句柄值，並「邏輯上」傳遞給用戶進程，於是用戶進程會得到一個新的合法句柄值，這個新句柄值可能在數值上和SMS所記錄的句柄值不同，然而，它們指代的卻是同一個Service實體。句柄的合法性是由Binder驅動保證的，這一點我們不必擔心。

前文我們提到要以「特定手法」獲取SMS代理接口，這是什麼意思呢？在IServiceManager.cpp文件中，我們可以看到一個defaultServiceManager()函數，代碼如下：

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

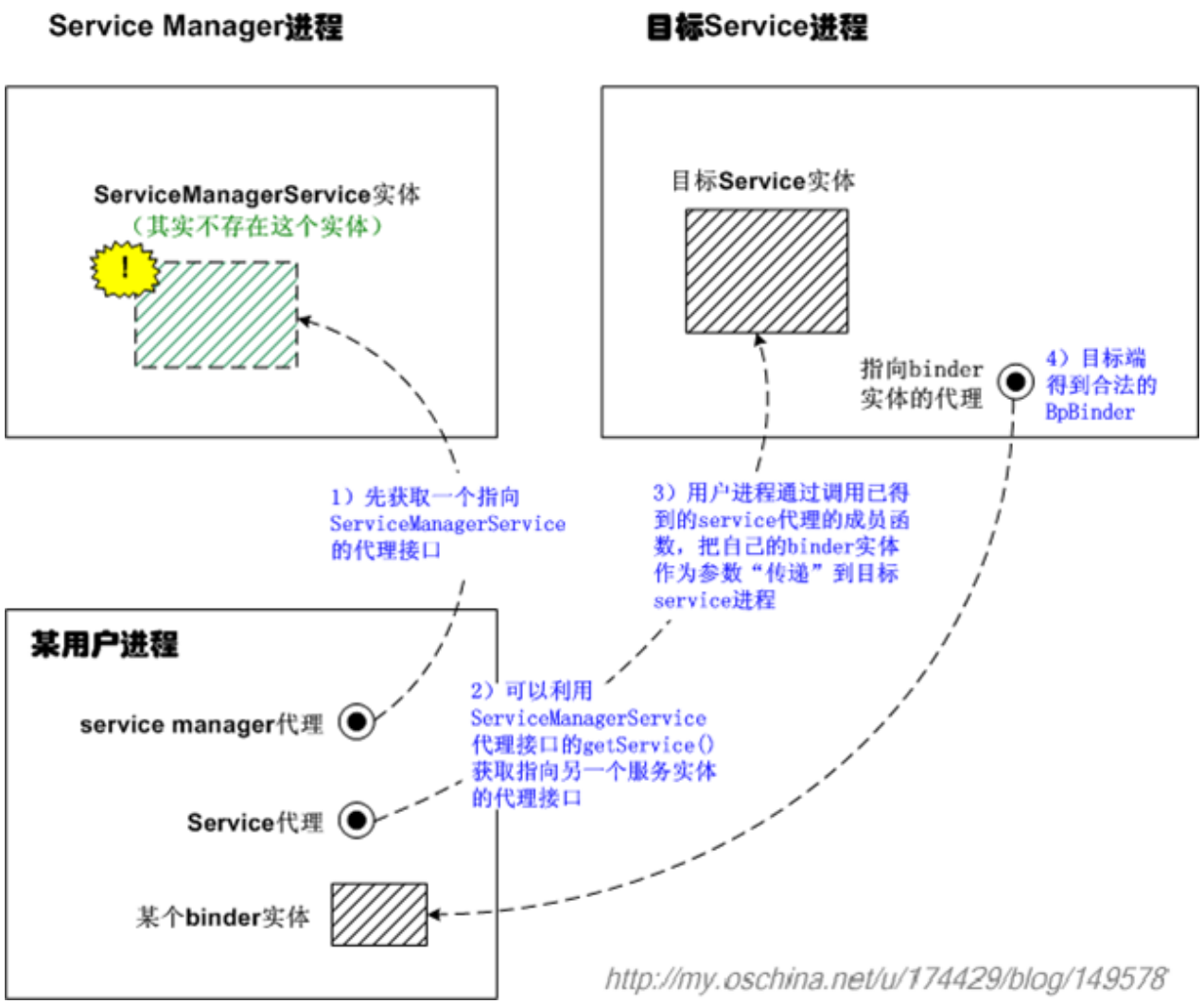
```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL)
         return gDefaultServiceManager;
    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL)
        {
            gDefaultServiceManager = interface_cast<IServiceManager>(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

    return gDefaultServiceManager;
}
```

這個函數裡調用interface_cast的地方是用一句getContextObject(NULL)來獲取BpBinder對象的。我們先不深入講解這個函數，只需要知道這一句裡的getContextObject(NULL)實際上相當於new BpBinder(0)就可以了。噢，看來要得到BpBinder對象並不複雜嘛，直接new就好了。然而，我之所以使用「特定手法」一詞，是因為這種直接new BpBinder(xxx)的做法，只能用於獲取SMS的代理接口。大家可不要想當然地隨便用這種方法去創建其他服務的代理接口噢。

在Android裡，對於Service Manager Service這個特殊的服務而言，其對應的代理端的句柄值已經預先定死為0了，所以我們直接new BpBinder(0)拿到的就是個合法的BpBinder，其對端為「Service Manager Service實體」（至少目前可以先這麼理解）。那麼對於其他「服務實體」對應的代理，句柄值又是多少呢？使用方又該如何得到這個句柄值呢？我們總不能隨便蒙一個句柄值吧。正如我們前文所述，要得到某個服務對應的BpBinder，主要得借助Service Manager Service系統服務，查詢出一個合法的Binder句柄，並進而創建出合法的BpBinder。

這裡有必要澄清一下，利用SMS獲取合法BpBinder的方法，並不是Android中得到BpBinder的唯一方法。另一種方法是，「起始端」經由一個已有的合法BpBinder，將某個binder實體或代理對像作為跨進程調用的參數，「傳遞」給「目標端」，這樣目標端也可以拿到一個合法的BpBinder。

我們把以上介紹的知識繪製成示意圖，如下：




請順著圖中標出的1）、2）、3）、4）序號，讀一下圖中的説明。

在跨進程通信方面，所謂的「傳遞」一般指的都是邏輯上的傳遞，所以應該打上引號。事實上，binder實體對像是不可能完全打包並傳遞到另一個進程的，而且也沒有必要這麼做。目前我們只需理解，binder架構會保證「傳遞」動作的目標端可以拿到一個和binder實體對像對應的代理對象即可。詳細情況，要到分析binder驅動的部分再闡述。

既然SMS承擔著讓客戶端獲取合法BpBinder的責任，那麼它的重要性就不言而喻了。現在我們就來詳細看看具體如何使用它。

# 2.具體使用Service Manager Service

## 2.1 必須先得到IServiceManager代理接口

要獲取某系統service的代理接口，必須先得到IServiceManager代理接口。還記得前文C++代碼中獲取IServiceManager代理接口的句子嗎？

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

我們在前一篇文檔中已經介紹過interface_cast了，現在再貼一下這個函數的代碼：

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast( const sp<IBinder>& obj)
{
```

```
    return INTERFACE::asInterface(obj);
}
```

也就是說，其實調用的是IServiceManager::asInterface(obj)，而這個obj參數就是new BpBinder(0)得到的對象。當然，這些都是C++層次的概念，Java層次把這些概念都包裝起來了。

在Java層次，是這樣獲取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，又出現了一個asInterface，看來Java層次和C++層的代碼在本質上是一致的。

ServiceManagerNative的asInterface()代碼如下：

```
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);
}
```

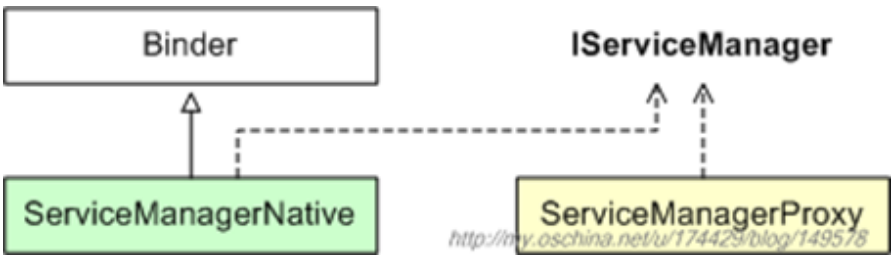
目前我們只需瞭解，用戶進程在調用到getIServiceManager()時，最終會走到return new ServiceManagerProxy(obj)即可。

哎呀，又出現了兩個名字：ServiceManagerProxy和ServiceManagerNative。簡單地說：

1） ServiceManagerProxy就是IServiceManager代理接口；

2） ServiceManagerNative顯得很雞肋；

它們的繼承關係圖如下：



下面我們分別來說明。

### 2.1.1 ServiceManagerProxy就是IServiceManager代理接口

用戶要訪問Service Manager Service服務，必須先拿到IServiceManager代理接口，而ServiceManagerProxy就是代理接口的實現。這個從前文代碼中的new ServiceManagerProxy(obj)一句就可以看出來了。ServiceManagerProxy的構造函數內部會把obj參數記錄到mRemote域中：

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

mRemote的定義是：

```
private IBinder mRemote;
```

其實說白了，mRemote的核心包裝的就是句柄為0的BpBinder對象，這個應該很容易理解。

日後，當我們通過IServiceManager代理接口訪問SMS時，其實調用的就是ServiceManagerProxy的成員函數。比如getService()、checkService()等等。

### 2.1.2 ServiceManagerNative顯得很雞肋

另一方面，ServiceManagerNative就顯得很雞肋了。

ServiceManagerNative是個抽像類：

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

它繼承了Binder，實現了IServiceManager，然而卻是個虛有其表的class。它唯一有用的大概就是前文列出的那個靜態成員函數asInterface()了，而其他成員函數（像onTransact()）就基本上沒什麼用。

如果我們花點兒時間在工程裡搜索一下ServiceManagerNative，會發現根本找不到它的子類。一個沒有子類的抽像類不就是虛有其表嗎。到頭來我們發現，關於ServiceManagerNative的用法只有一種，就是：

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

用一下它的asInterface()靜態函數而已。

為什麼會這樣呢？我想這可能是某種歷史的遺蹟吧。同理，我們看它的onTransact()函數，也會發現裡面調用的類似addService()那樣的函數，也都是找不到對應的實現體的。當然，因為ServiceManagerNative本身是個抽像類，所以即便它沒有實現IServiceManager的addService()等成員函數，也是可以編譯通過的。

這裡透出一個信息，既然Java層的ServiceManagerNative沒什麼大用處，是不是表示C++層也缺少對應的SMS服務實體呢？在後文我們可以看到，的確是這樣的，Service Manager Service在C++層被實現成一個獨立的進程，而不是常見的Binder實體。

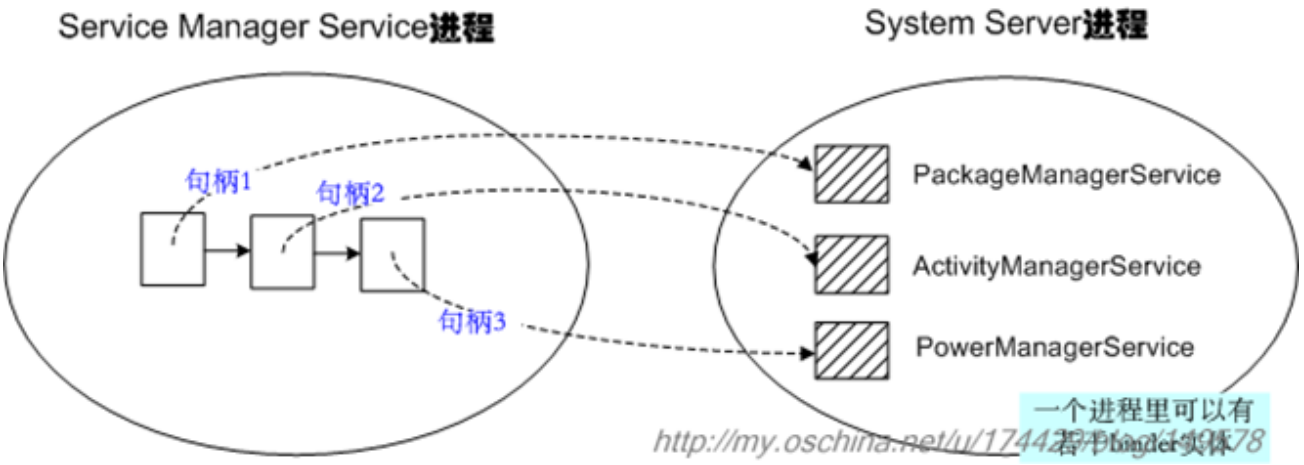
## 2.2 通過addService()來註冊系統服務

我們還是回過頭接著說對於IServiceManager接口的使用吧。最重要的當然是註冊系統服務。比如在System Server進程中，是這樣註冊PowerManagerService系統服務的：

```
public  void run()
```

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

addService()的第一個參數就是所註冊service的名字，比如上面的POWER_SERVICE對應的字符串就是"power"。第二個參數傳入的是service Binder實體。Service實體在Service Manager Service一側會被記錄成相應的句柄值，如圖：




有關addService()內部機理，我們會在後文講述，這裡先不細說。

## 2.3 通過getService()來獲取某系統服務的代理接口

除了註冊系統服務，Service Manager Service的另一個主要工作就是讓用戶進程可以獲取系統service的代理接口，所以其getService()函數就非常重要了。

其實，ServiceManagerProxy中的getService()等成員函數，僅僅是把語義整理進parcel，並通過mRemote將parcel傳遞到目標端而已。所以我們只看看getService()就行了，其他的函數都大同小異。

```
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;
}
```

傳遞的語義就是GET_SERVICE_TRANSACTION，非常簡單。mRemote從本質上看就是句柄為0的BpBinder，所以binder驅動很清楚這些語義將去向何方。

關於Service Manager Service的使用，我們就先說這麼多。下面我們要開始探索SMS內部的運作機制了。

# 3.Service Manager Service的運作機制

# 3.1 Service Manager Service服務的啟動

既然前文説ServiceManagerNative虛有其表，而且沒有子類，那麼Service Manager Service服務的真正實現代碼位於何處呢？答案就在init.rc腳本裡。關於init.rc的詳細情況，可參考其他闡述Android啟動流程的文檔，此處不再贅述。

init.rc腳本中，在描述zygote service之前就已經寫明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是一種native service。這種native service都是需要用C/C++編寫的。Service Manager Service對應的實現代碼位於frameworks/base/cmds/servicemanager/Service_manager.c文件中。這個文件中有每個C程序員都熟悉的main()函數，其編譯出的可執行程序就是/system/bin/servicemanager。

另外，還有一個干擾我們視線的cpp文件，名為IServiceManager.cpp，位於frameworks/base/libs/binder/目錄中，這個文件裡的BnServiceManager應該和前文的ServiceManagerNative類似，它的onTransact()也不起什麼作用。

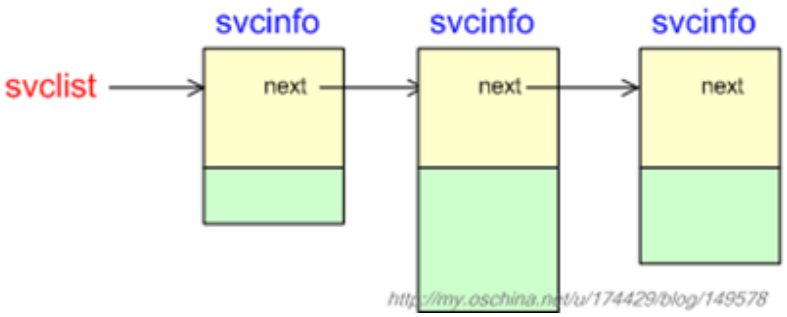
# 3.2 Service Manager Service是如何管理service句柄的？

在C語言層次，簡單地説並不存在一個單獨的ServiceManager結構。整個service管理機制都被放在一個獨立的進程裡了，該進程對應的實現文件就是Service_manager.c。

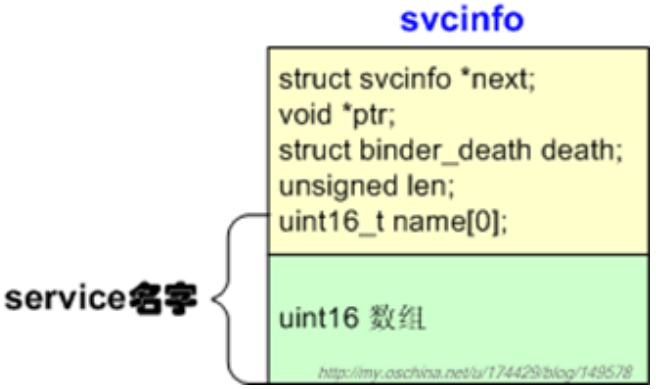
進程裡有一個全局性的svclist變量：

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

它記錄著所有添加進系統的「service代理」信息，這些信息被組織成一條單向煉表，我們不妨稱這條煉表為「服務向量表」。示意圖如下：

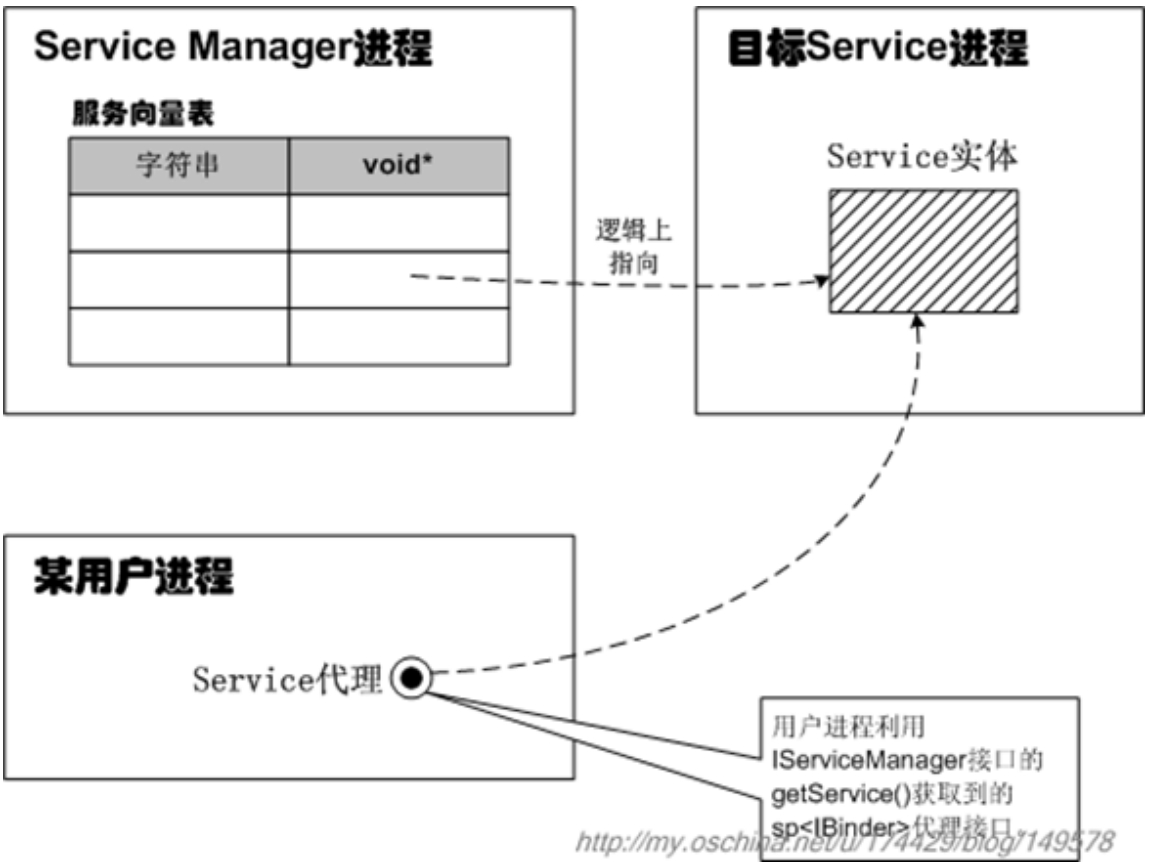



煉表節點類型為svcinfo。



因為svcinfo裡要記錄下service的名字字符串，所以它需要的buffer長度是(len + 1) * sizeof(uint16_t)，記得要留一個'\0'的結束位置。另外，svcinfo的ptr域，實際上記錄的就是系統service對應的binder句柄值。

日後，當應用調用getService()獲取系統服務的代理接口時，SMS就會搜索這張「服務向量表」，查找是否有節點能和用戶傳來的服務名匹配，如果能查到，就返回對應的sp<IBinder>，這個接口在遠端對應的實體就是「目標Service實體」。如此一來，系統中就會出現如下關係：




# 3.3 Service Manager Service的主程序（C++層）

要更加深入地瞭解Service Manager進程的運作，我們必須研究其主程序。參考代碼是frameworks\base\cmds\servicemanager\Service_manager.c。

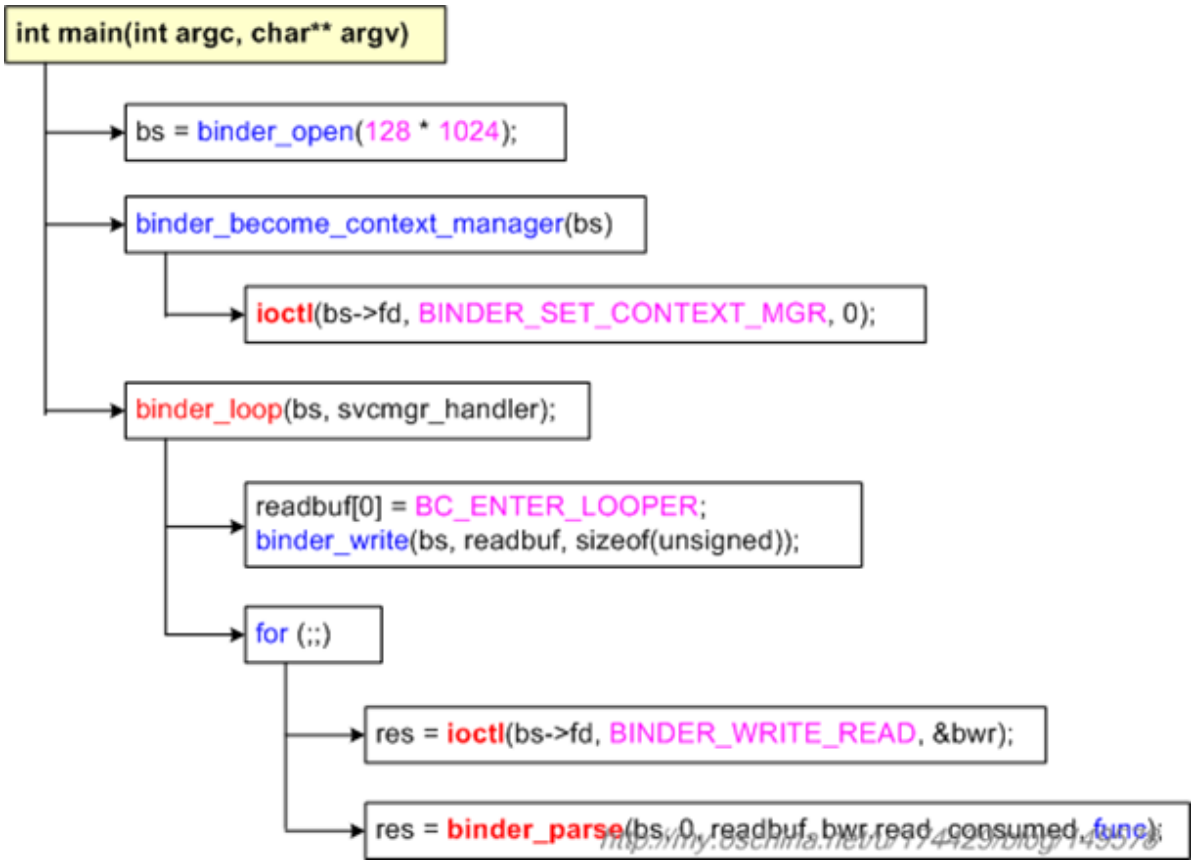
Service_manager.c中的main()函數如下：

```
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()函數一開始就打開了binder驅動，然後調用binder_become_context_manager()讓自己成為整個系統中唯一的上下文管理器，其實也就是service管理器啦。接著main()函數調用binder_loop()進入無限循環，不斷監聽並解析binder驅動發來的命令。

binder_loop()中解析驅動命令的函數是binder_parse()，其最後一個參數func來自於binder_loop()的最後一個參數——svcmgr_handler函數指針。這個svcmgr_handler()應該算是Service Manager Service的核心回調函數了。

為了方便查看，我把main()函數以及其間接調用的ioctl()語句繪製成如下的調用關係圖：



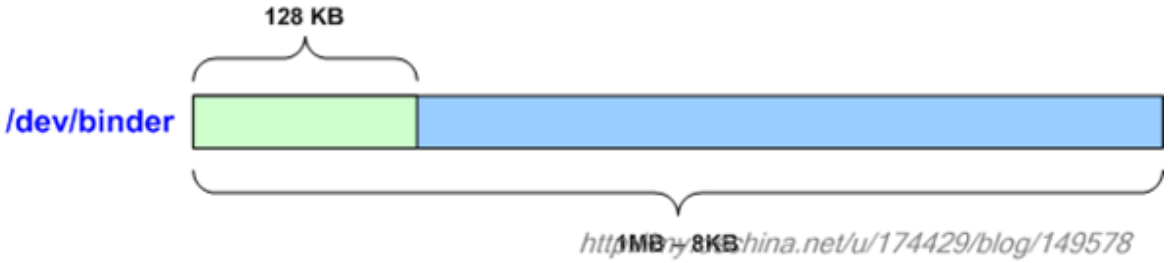

下面我們逐個分析其中調用的函數。

## 3.3.1 binder_open()

Service Manager Service必須先調用binder_open()來打開binder驅動，驅動文件為「/dev/binder」。binder_open()的代碼截選如下：

```
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()的參數mapsize表示它希望把binder驅動文件的多少字節映射到本地空間。可以看到，Service Manager Service和普通進程所映射的binder大小並不相同。它把binder驅動文件的128K字節映射到內存空間，而普通進程則會映射binder文件裡的BINDER_VM_SIZE（即1M減去8K）字節。




具體的映射動作由mmap()一句完成，該函數將binder驅動文件的一部分映射到進程空間。mmap()的函數原型如下：

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

該函數會把「參數fd所指代的文件」中的一部分映射到進程空間去。這部分文件內容以offset為起始位置，以len為字節長度。其中，參數offset表明從文件起始處開始算起的偏移量。參數prot表明對這段映射空間的訪問權限，可以是PROT_READ（可讀）、PROT_WRITE（可寫）、PROT_EXEC（可執行）、PROT_NONE（不可訪問）。參數addr用於指出文件應被映射到進程空間的起始地址，一般指定為空指針，此時會由內核來決定起始地址。

binder_open()的返回值類型為binder_state*，裡面記錄著剛剛打開的binder驅動文件句柄以及mmap()映射到的最終目標地址。

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

以後，SMS會不斷讀取這段映射空間，並做出相應的動作。

## 3.3.2 binder_become_context_manager()

我們前面已經說過，binder_become_context_manager()的作用是讓當前進程成為整個系統中唯一的上下文管理器，即service管理器。其代碼非常簡單：

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

僅僅是把BINDER_SET_CONTEXT_MGR發送到binder驅動而已。驅動中與ioctl()對應的binder_ioctl()是這樣處理的：

```
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 ;
    . . . . . .
    . . . . . .
}
```

代碼的意思很明確，要為整個系統的上下文管理器專門生成一個binder_node節點，並記入靜態變量binder_context_mgr_node。

我們在這裡多說兩句，一般情況下，應用層的每個binder實體都會在binder驅動層對應一個binder_node節點，然而binder_context_mgr_node比較特殊，它沒有對應的應用層binder實體。在整個系統裡，它是如此特殊，以至於系統規定，任何應用都必須使用句柄0來跨進程地訪問它。現在大家可以回想一下前文在獲取SMS接口時說到的那句new BpBinder(0)，是不是能加深一點兒理解。

## 3.3.3 binder_loop()

我們再回到SMS的main()函數。

接下來的binder_loop()會先向binder驅動發出了BC_ENTER_LOOPER命令，接著進入一個for循環不斷調用ioctl()讀取發來的數據，接著解析這些數據。參考代碼在：

【frameworks/base/cmds/servicemanager/Binder.c】（注意！這個Binder.c文件不是binder驅動層那個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()的參數func，它的值是svcmgr_handler()函數指針。而且這個參數會進一步傳遞給binder_parse()。

### 3.3.3.1 BC_ENTER_LOOPER

binder_loop()中發出BC_ENTER_LOOPER命令的目的，是為了告訴binder驅動「本線程要進入循環狀態了」。在binder驅動中，凡是用到跨進程通信機制的線程，都會對應一個binder_thread節點。這裡的BC_ENTER_LOOPER命令會導致這個節點的looper狀態發生變化：

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

有關binder_thread的細節，也會在闡述Binder驅動一節進行說明。

### 3.3.3.2 binder_parse()

在binder_loop()進入for循環之後，最顯眼的就是那句binder_parse()了。binder_parse()負責解析從binder驅動讀來的數據，其代碼截選如下：

```
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;
}
```

從前文的代碼我們可以看到，binder_loop()聲明了一個128字節的buffer（即unsigned readbuf[32]），每次用BINDER_WRITE_READ命令從驅動讀取一些內容，並傳入binder_parse()。

binder_parse()在合適的時機，會回調其func參數（binder_handler func）指代的回調函數，即前文說到的svcmgr_handler()函數。

binder_loop()就這樣一直循環下去，完成了整個service manager service的工作。

# 4.Service Manager Service解析收到的命令

現在，我們專門用一個小節來說說Service Manager Service內循環解析命令時的一些細節。我們要確定binder_loop()從驅動側讀到的數據到底如何解析？我們重貼一下binder_parse()的聲明部分：

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

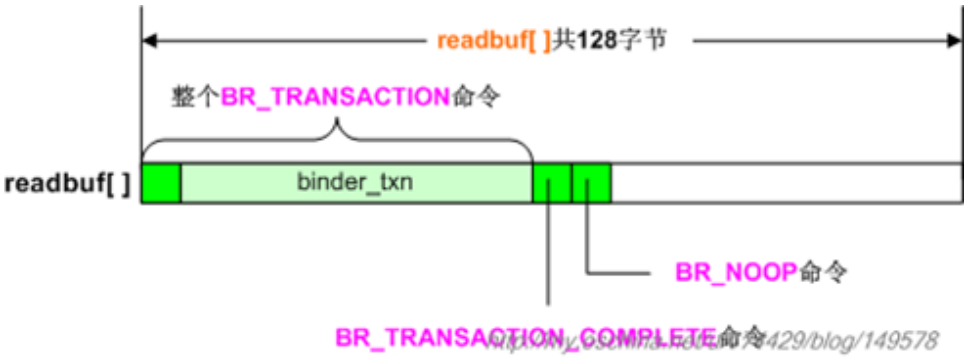
之前利用ioctl()讀取到的數據都記錄在第三個參數ptr所指的緩衝區中，數據大小由size參數記錄。其實這個buffer就是前文那個128字節的buffer。

從驅動層讀取到的數據，實際上是若干BR命令。每個BR命令是由一個命令號(uint32)以及若干相關數據組成的，不同BR命令的長度可能並不一樣。如下表所示：

| BR 命令 | 需進一步讀取的uint32 數 |
|---|---|
| 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()操作所讀取的數據，可能會包含多個BR命令，所以binder_parse()需要用一個while循環來解析buffer中所有的BR命令。我們隨便畫個示意圖，如下：




圖中的buffer中含有3條BR命令，分別為BR_TRANSACTION、BR_TRANSACTION_COMPLETE、BR_NOOP命令。一般而言，我們最關心的就是BR_TRANSACTION命令啦，因此前文截選的binder_parse()代碼，主要摘錄了處理BR_TRANSACTION命令的代碼，該命令的命令號之後跟著的是一個binder_txn結構。現在我們來詳細看這個結構。

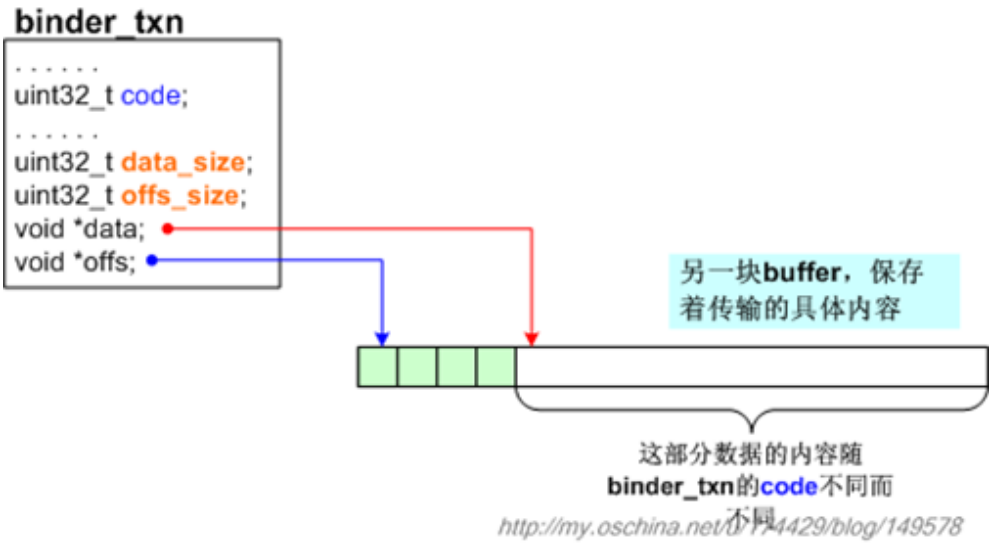
# 4.1 解析binder_txn信息

binder_txn的定義如下：

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

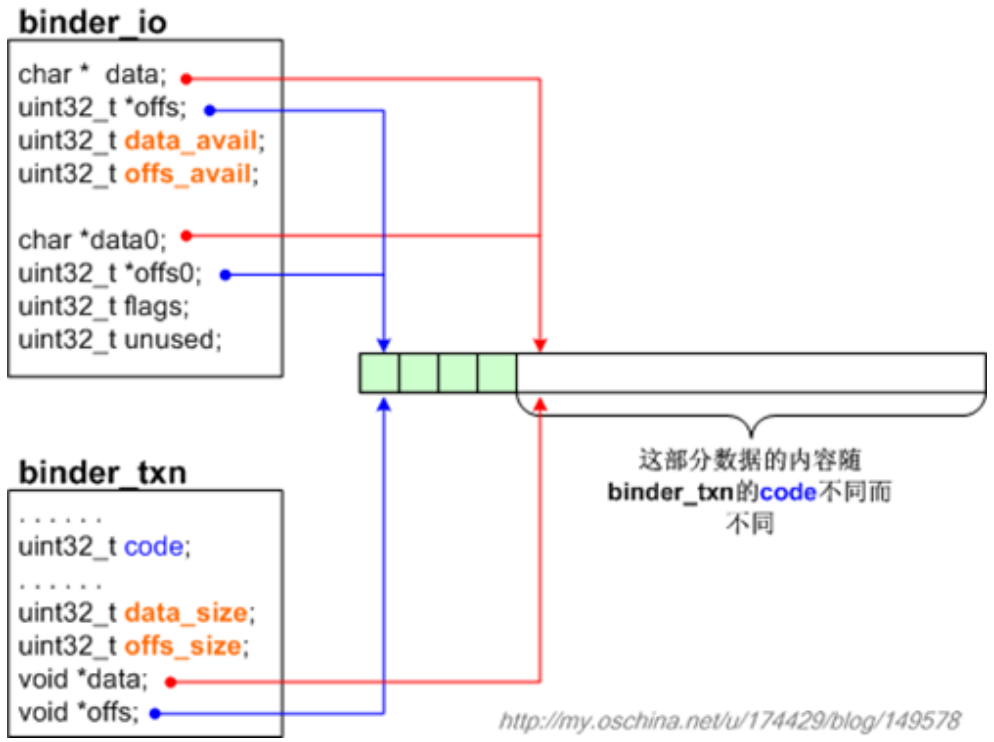
```
struct binder_txn
{
    void *target;
     void *cookie;
    uint32_t code;          //所傳輸的語義碼
    uint32_t flags;

    uint32_t sender_pid;
    uint32_t sender_euid;

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

binder_txn說明了transaction到底在傳輸什麼語義，而語義碼就記錄在其code域中。不同語義碼需要攜帶的數據也是不同的，這些數據由data域指定。示意圖如下：




簡單地說，我們從驅動側讀來的binder_txn只是一種「傳輸控制信息」，它本身並不包含傳輸的具體內容，而只是指出具體內容位於何處。現在，工作的重心要轉到如何解析傳輸的具體內容了，即binder_txn的data域所指向的那部分內容。

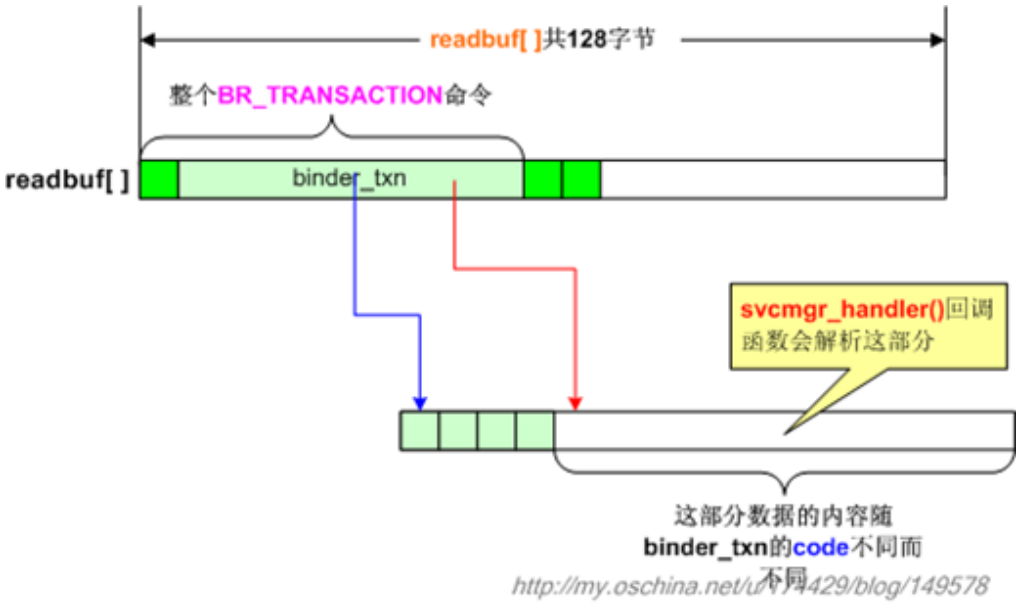
為瞭解析具體內容，binder_parse()聲明了兩個類型為binder_io的局部變量：msg和reply。從binder_io這個類型的名字，我們就可以看出要用它來讀取binder傳遞來的數據了。其實，為了便於讀取binder_io所指代的內容，工程提供了一系列以bio_打頭的輔助函數。在讀取實際數據之前，我們必須先調用bio_init_from_txn()，把binder_io變量（比如msg變量）和binder_txn所指代的緩衝區聯繫起來。示意圖如下：




從圖中可以看到，binder_io結構已經用binder_txn結構初始化了自己，以後我們就可以調用類似bio_get_uint32()、bio_get_string16()這樣的函數，來讀取這塊buffer了。

# 4.2 svcmgr_handler()回調函數

初始化後的binder_io數據，就可以傳給svcmgr_handler()回調函數做進一步的解析了。




此時我們可以調用下面這些輔助函數進行讀寫：

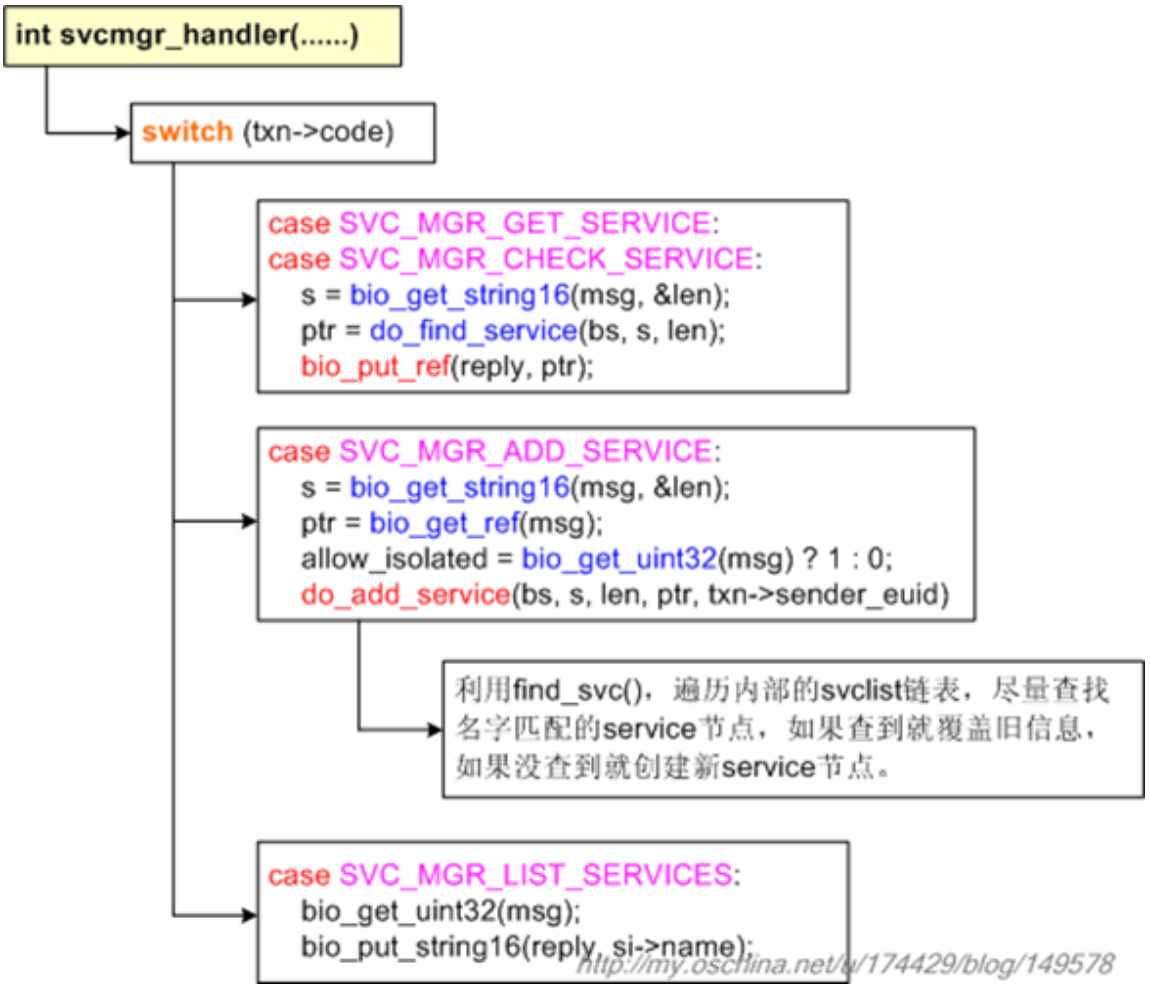
```
void bio_put_uint32( struct binder_io *bio, uint32_t n)
 void bio_put_obj( struct binder_io *bio, void *ptr)
uint32_t bio_get_uint32( struct binder_io *bio)
uint16_t *bio_get_string16( struct binder_io *bio, unsigned *sz)
 void *bio_get_ref( struct binder_io *bio)
. . . . . .
```

其中，bio_get_xxx()函數在讀取數據時，是以binder_io的data域為讀取光標的，每讀取一些數據，data值就會增加，並且data_avail域會相應減少。而data0域的值則保持不變，一直指著數據區最開始的位置，它的作用就是作為計算偏移量的基準值。

bio_get_uint32()非常簡單，會從binder_io.data所指的地方，讀取4個字節的內容。bio_get_string16()就稍微複雜一點兒，先要讀取一個32bits的整數，這個整數值就是字符串的長度，因為字符串都要包含最後一個'\0'，所以需要讀取((len + 1) * sizeof(uint16_t))字節的內容。還有一個是bio_get_ref()，它會讀取一個binder_object結構。binder_object的定義如下：

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

在svcmgr_handler()函數中，一個傳輸語義碼（txn->code）可能會對應幾次bio_get操作，比如後文我們要說的SVC_MGR_ADD_SERVICE語義碼。具體情況請大家參考svcmgr_handler()的代碼。svcmgr_handler()的調用示意圖如下：



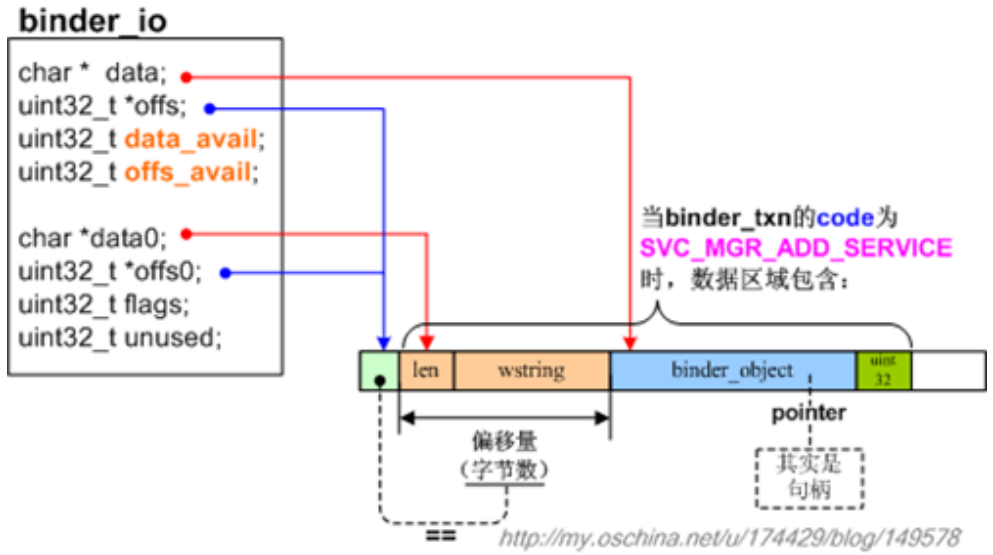

## 4.2.1 如何解析add service

我們先研究add service的動作。前文我們已經介紹過，service manager進程裡有一個全局性的svclist變量，記錄著所有添加進系統的「service代理」信息，這些信息被組織成一條單向煉表，即「服務向量表」。現在我們要看service manager是如何向這張表中添加新節點的。

假設某個服務進程調用Service Manager Service接口，向其註冊service。這個註冊動作到最後就會走到svcmgr_handler()的case SVC_MGR_ADD_SERVICE分支。此時會先獲取三個數據，而後再調用do_add_service()函數，代碼如下：

```
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);
```

也就是說，當binder_txn的code為SVC_MGR_ADD_SERVICE時，binder_txn所指的數據區域中應該包含一個字符串，一個binder對像以及一個uint32數據。示意圖如下：




其中那個binder_object，記錄的就是新註冊的service所對應的代理信息。此時binder_object的pointer域實際上已經不是指針值了，而是一個binder句柄值。

do_add_service()的函數截選如下：

```
struct svcinfo *svclist = 0;     //核心service煉表（即服務向量表）

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 {
        //新創建一個svcinfo節點。
        si = malloc( sizeof (*si) + (len + 1) * sizeof (uint16_t));
        if (!si) {
            return -1;
        }
        si->ptr = ptr;     //在svcinfo節點的ptr域中，記錄下service對應的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;

        // 把新節點插入svclist煉表
        si->next = svclist;
        svclist = si;
    }

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

現在我們來解讀這部分代碼。首先，並不是隨便找個進程就能向系統註冊service噢。do_add_service()函數一開始先調用svc_can_register()，判斷髮起端是否可以註冊service。如果不可以，do_add_service()就返回-1值。svc_can_register()的代碼如下：

```
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;
}
```

上面的代碼表示，如果發起端是root進程或者system server進程的話，是可以註冊service的，另外，那些在allowed[]數組中有明確記錄的用戶進程，也是可以註冊service的，至於其他絕大部分普通進程，很抱歉，不允許註冊service。在以後的軟件開發中，我們有可能需要編寫新的帶service的用戶進程（uid不為0或AID_SYSTEM），並且希望把service註冊進系統，此時不要忘了修改allowed[]數組。下面是allowed[]數組的一部分截選：

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

接下來，do_add_service()開始嘗試在service煉表裡查詢對應的service是否已經添加過了。如果可以查到，那麼就不用生成新的service節點了。否則就需要在煉表起始處再加一個新節點。節點類型為svcinfo。請注意上面代碼的si->ptr = ptr一句，此時的ptr參數其實來自於前文所說的binder_object的pointer域。

為了說明問題，我們重新列一下剛剛的case SVC_MGR_ADD_SERVICE代碼：

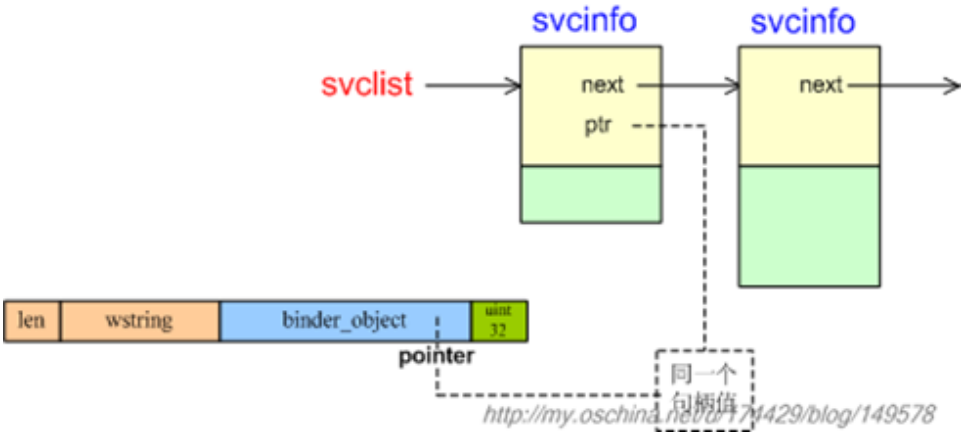
```
case SVC_MGR_ADD_SERVICE:
    s = bio_get_string16(msg, &len);
    ptr = bio_get_ref(msg);
    allow_isolated = bio_get_uint32(msg) ? 1 : 0;
    if (do_add_service(bs, s, len, ptr, txn->sender_euid, allow_isolated))
         return -1;
     break ;
```

那個ptr來自於bio_get_ref(msg)，而bio_get_ref()的實現代碼如下：

```
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;
}
```

因為現在是要向service manager註冊服務，所以obj->type一定是BINDER_TYPE_HANDLE，也就是說會返回binder_object的pointer域。這個域的類型雖為void*，實際上換成uint32可能更合適。通過

這個binder句柄值，我們最終可以找到遠端的具體service實體。




## 4.2.2 如何解析get service

現在我們接著來看get service動作。我們知道，在service被註冊進service manager之後，其他應用都可以調用ServiceManager的getService()來獲取相應的服務代理，並調用代理的成員函數。這個getService()函數最終會向service manager進程發出SVC_MGR_GET_SERVICE命令，並由svcmgr_handler()函數這樣處理：

```
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;
```

一開始從msg中讀取希望get的服務名，然後調用do_find_service()函數查詢服務名對應的句柄值，最後把句柄值寫入reply。do_find_service()的代碼如下：

```
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務代理對應的句柄值（si->ptr）。而svcmgr_handler()在拿到這個句柄值後，會把它寫入reply對象：

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

bio_put_ref()的代碼如下：

```
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()一句說明會從reply所關聯的buffer中劃分出一個binder_object區域，然後開始對這個區域寫值。於是BINDER_TYPE_HANDLE賦給了obj->type，句柄值賦給了obj->pointer。另外，reply所關聯的buffer只是binder_parse()裡的局部數組噢：

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

大家應該還記得svcmgr_handler()是被binder_parse()回調的，當svcmgr_handler()返回後，會接著把整理好的reply對象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，發送給發起查找的一方。

binder_send_reply()的代碼如下：

```
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));
}
```

觀察代碼中最後那幾行，看來還是在擺弄reply所指代的那個buffer。當初binder_parse()在創建reply對象之時，就給它初始化了一個局部buffer，即前文所說的unsigned rdata [256/4]，在svcmgr_handler()中又調用bio_put_ref()在這個buffer中開闢了一塊binder_object，並在其中賦予了ptr句柄。現在終於要向binder驅動傳遞reply信息了，此時調用的binder_write()的代碼如下：

```
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，又見ioctl()，數據就在bwr.write_buffer，數據裡打出了兩個binder命令，BC_FREE_BUFFER和BC_REPLY。

這些數據被傳遞給get service動作的發起端，雖然這些數據會被binder驅動做少許修改，不過語義是不會變的，於是發起端就獲得了所查service的合法句柄。

# 5.小結

至此，有關ServiceManager的基本知識就大體交代完畢了，文行於此，暫告段落。必須承認，受限於個人的認識和文章的篇幅，我不可能涉及其中所有的細節，這裡只能摘其重點進行闡述。如果以後又發現什麼有趣的東西，我還會補充進來。

*如需轉載本文內容，請註明出處。*

*謝謝。*

- « 紅茶一杯話Binder（初始篇）
- 紅茶一杯話Binder（傳輸機制篇_上）»






# 共有3 條網友評論

- 
  1樓：simonws發表於2013-08-09 23:13 回覆此評論
  你是怎麼研究的？
- 2樓：悠然紅茶發表於2013-08-10 19:24 回覆此評論

  
  **引用來自「simonws」的評論**

  你是怎麼研究的？

  無他法，唯讀代碼爾。呵呵。
- 
  3樓：simonws發表於2013-08-11 23:28 回覆此評論
  fucking source code

發表評論

文明上網，理性發言

回到頁首 | 回到評論列表

**關閉相關文章閱讀**


- 2013/08/02 紅茶一杯話Binder（初始篇）
- 2013/08/12 紅茶一杯話Binder（傳輸機制篇_上）...
- 2013/08/15 紅茶一杯話Binder（傳輸機制篇_中）...
- 2013/08/04 Android Binder的使用和設計[androi...
- 2012/06/02 Android Binder IPC分析...