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，這樣認真的... 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**
- **本周訪問： 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（傳輸機制篇_上）

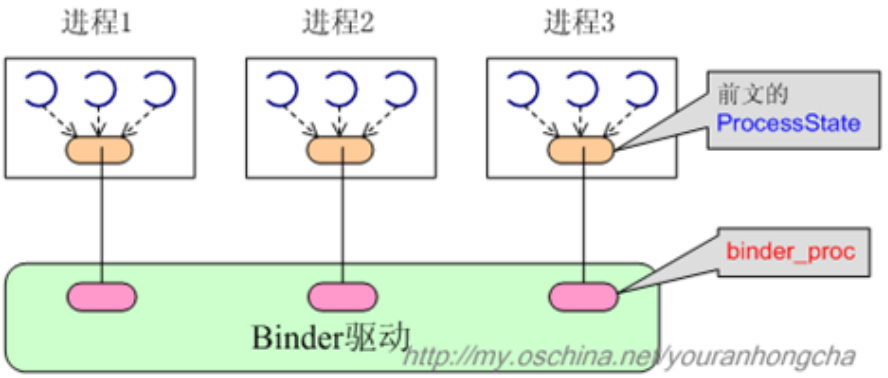
紅茶一杯話Binder

（傳輸機制篇_上）

侯亮

1 Binder是如何做到精確打擊的？

我們先問一個問題，binder機製到底是如何從代理對象找到其對應的binder實體呢？難道它有某種制導裝置嗎？要回答這個問題，我們只能靜下心來研究binder驅動的代碼。在本系列文檔的初始篇中，我們曾經介紹過ProcessState，這個結構是屬於應用層次的東西，僅靠它當然無法完成精確打擊。其實，在binder驅動層，還有個與之相對的結構，叫做binder_proc。為了說明問題，我修改了初始篇中的示意圖，得到下圖：



1.1 創建binder_proc

當構造ProcessState並打開binder驅動之時，會調用到驅動層的binder_open()函數，而binder_proc就是在binder_open()函數中創建的。新創建的binder_proc會作為一個節點，插入一個總鍊錶（binder_procs）中。具體代碼可參考kernel/drivers/staging/android/Binder.c。

驅動層的binder_open()的代碼如下：

```
static  int binder_open( struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc;

    . . . . . .
    proc = kzalloc( sizeof (*proc), GFP_KERNEL);

    get_task_struct(current);
    proc->tsk = current;
    . . . . . .
    hlist_add_head(&proc->proc_node, &binder_procs);
    proc->pid = current->group_leader->pid;
    . . . . . .
    filp->private_data = proc;
    . . . . . .
}
```

注意，新創建的binder_proc會被記錄在參數filp的private_data域中，以後每次執行binder_ioctl()，都會

從filp->private_data域重新讀取binder_proc的。

binder_procs總表的定義如下：

```
static HLIST_HEAD(binder_procs);
```

我們可以在List.h中看到HLIST_HEAD的定義：

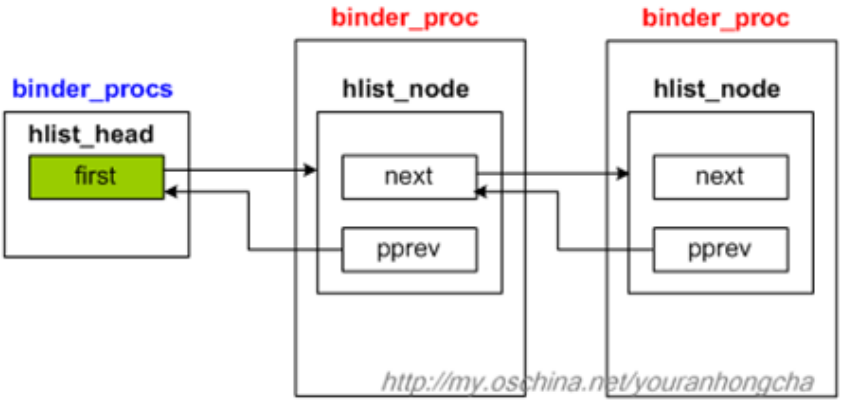
【kernel/include/linux/List.h】

```
#define HLIST_HEAD(name) struct hlist_head name = { .first = NULL }
```

於是binder_procs的定義相當於：

```
struct hlist_head binder_procs = { .first = NULL };
```

隨著後續不斷向binder_procs表中添加節點，這個表會不斷加長，示意圖如下：

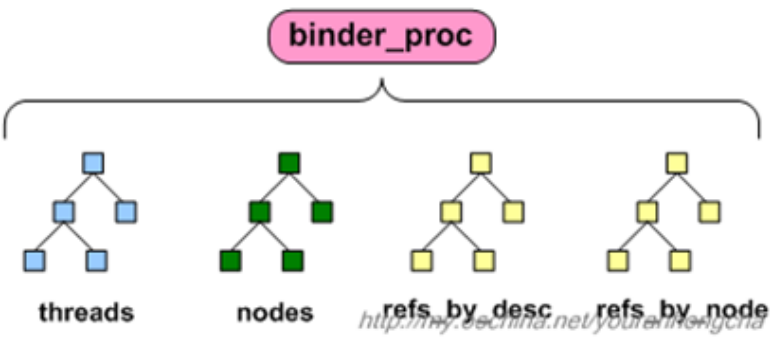


1.2 binder_proc中的4棵紅黑樹

binder_proc裡含有很多重要內容，不過目前我們只需關心其中的幾個域：

```
struct binder_proc
{
    struct hlist_node proc_node;
     struct rb_root threads;
     struct rb_root nodes;
     struct rb_root refs_by_desc;
     struct rb_root refs_by_node;
     int pid;
    . . . . . .
    . . . . . .
};
```

注意其中的那4個rb_root域，“rb”的意思是“red black”，可見binder_proc裡搞出了4個紅黑樹。

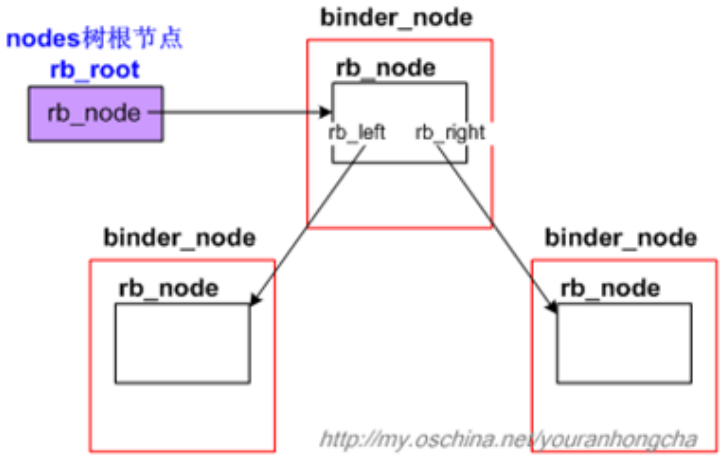


其中，nodes樹用於記錄binder實體，refs_by_desc樹和refs_by_node樹則用於記錄binder代理。之所以會有兩個代理樹，是為了便於快速查找，我們暫時只關心其中之一就可以了。threads樹用於記錄執行傳輸

動作的線程信息。

在一個進程中，有多少“被其他進程進行跨進程調用的”binder實體，就會在該進程對應的nodes樹中生成多少個紅黑樹節點。另一方面，一個進程要訪問多少其他進程的binder實體，則必須在其refs_by_desc樹中擁有對應的引用節點。

這4棵樹的節點類型是不同的，threads樹的節點類型為binder_thread，nodes樹的節點類型為binder_node，refs_by_desc樹和refs_by_node樹的節點類型相同，為binder_ref。這些節點內部都會包含rb_node子結構，該結構專門負責連接節點的工作，和前文的hlist_node有點兒異曲同工，這也是linux上一個常用的小技巧。我們以nodes樹為例，其示意圖如下：



rb_node和rb_root的定義如下：

```
struct rb_node
{
    unsigned long   rb_parent_color;
 #define RB_RED 0
 #define RB_BLACK 1
     struct rb_node *rb_right;
     struct rb_node *rb_left;
} __attribute__((aligned( sizeof ( long ))));
     /* The alignment might seem pointless, but allegedly CRIS needs it */

struct rb_root
{
    struct rb_node *rb_node;
};
```

binder_node的定義如下：

```
struct binder_node
{
    int debug_id;
     struct binder_work work;
    union {
        struct rb_node rb_node;
         struct hlist_node dead_node;
    };
    struct binder_proc *proc;
     struct hlist_head refs;
     int internal_strong_refs;
     int local_weak_refs;
     int local_strong_refs;
     void __user *ptr;          //注意這個域！
    void __user *cookie;      //注意這個域！
    unsigned has_strong_ref:1;
    unsigned pending_strong_ref:1;
    unsigned has_weak_ref:1;
    unsigned pending_weak_ref:1;
    unsigned has_async_transaction:1;
    unsigned accept_fds:1;
    unsigned min_priority:8;
```

```
    struct list_head async_todo;
};
```

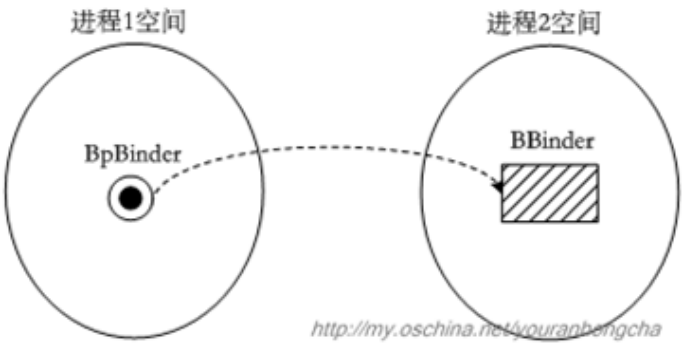
我們前文已經說過，nodes樹是用於記錄binder實體的，所以nodes樹中的每個binder_node節點，必須能夠記錄下相應binder實體的信息。因此請大家注意binder_node的ptr域和cookie域。

另一方面，refs_by_desc樹和refs_by_node樹的每個binder_ref節點則和上層的一個BpBinder對應，而且更重要的是，它必須具有和“目標binder實體的binder_node”進行關聯的信息。binder_ref的定義如下：

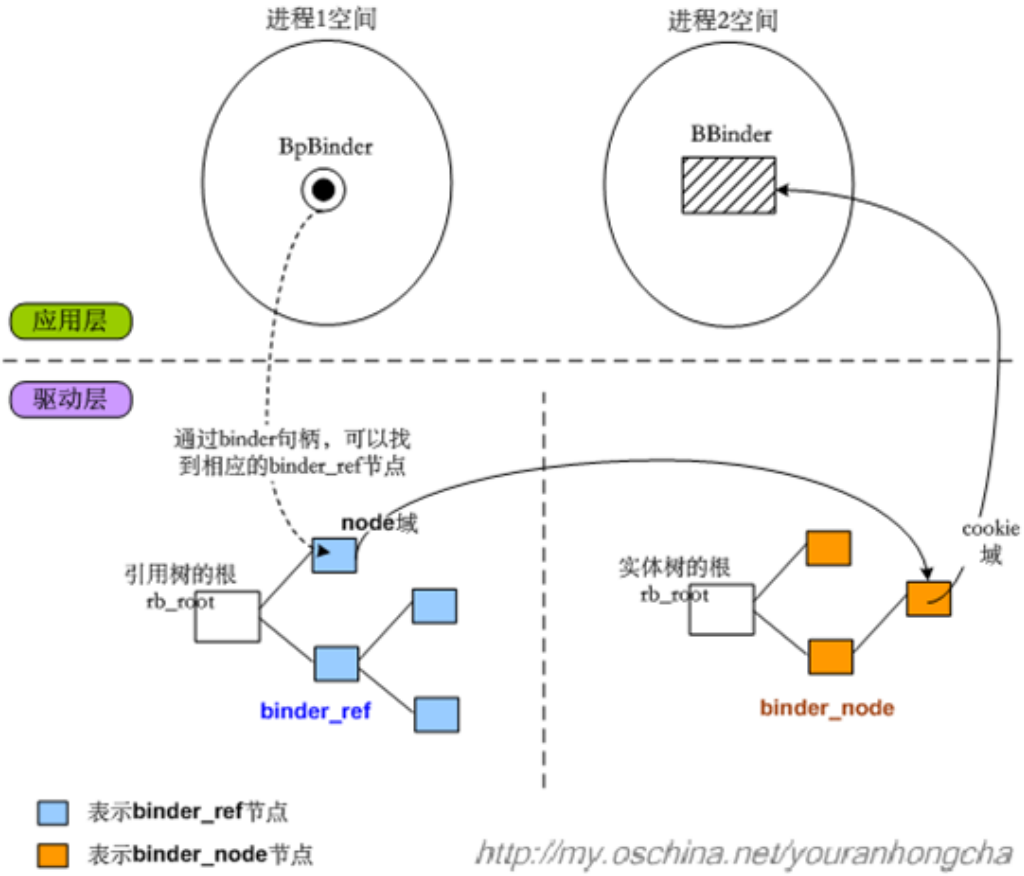
```
struct binder_ref
{
    int debug_id;
     struct rb_node rb_node_desc;
     struct rb_node rb_node_node;
     struct hlist_node node_entry;
     struct binder_proc *proc;
     struct binder_node *node;    //注意這個node域
    uint32_t desc;
    int strong;
     int weak;
     struct binder_ref_death *death;
};
```

請注意那個node域，它負責和binder_node關聯。另外，binder_ref中有兩個類型為rb_node的域：rb_node_desc域和rb_node_node域，它們分別用於連接refs_by_desc樹和refs_by_node。也就是說雖然binder_proc中有兩棵引用樹，但這兩棵樹用到的具體binder_ref節點其實是複用的。

大家應該還記得，在《初始篇》中我是這樣表達BpBinder和BBinder關係的：



現在，我們有了binder_ref和binder_node知識，可以再畫一張圖，來解釋BpBinder到底是如何和BBinder聯繫上的：



上圖只表示了從進程1向進程2發起跨進程傳輸的意思，其實反過來也是可以的，即進程2也可以通過自己的“引用樹”節點找到進程1的“實體樹”節點，並進行跨進程傳輸。大家可以自己補充上圖。

OK，現在我們可以更深入地說明binder句柄的作用了，比如進程1的BpBinder在發起跨進程調用時，向binder驅動傳入了自己記錄的句柄值，binder驅動就會在“進程1對應的binder_proc結構”的引用樹中查找和句柄值相符的binder_ref節點，一旦找到binder_ref節點，就可以通過該節點的node域找到對應的binder_node節點，這個目標binder_node當然是從屬於進程2的binder_proc啦，不過不要緊，因為binder_ref和binder_node都處於binder驅動的地址空間中，所以是可以用指針直接指向的。目標binder_node節點的cookie域，記錄的其實是進程2中BBinder的地址，binder驅動只需把這個值反映給應用層，應用層就可以直接拿到BBinder了。這就是Binder完成精確打擊的大體過程。

2 BpBinder和IPCThreadState

接下來我們來談談Binder傳輸機制。

在《初始篇》中，我們已經提到了BpBinder和ProcessState。當時只是說BpBinder是代理端的核心，主要負責跨進程傳輸，並且不關心所傳輸的內容。而ProcessState則是進程狀態的記錄器，它裡面記錄著打開binder驅動後得到的句柄值。因為我們並沒有進一步展開來討論BpBinder和ProcessState，所以也就沒有進一步打通BpBinder和ProcessState之間的關係。現在，我們試著補充一些內容。

作為代理端的核心，BpBinder總要通過某種方式和binder驅動打交道，才可能完成跨進程傳遞語義的工作。既然binder驅動對應的句柄在ProcessState中記著，那麼現在就要看BpBinder如何和ProcessState聯繫了。此時，我們需要提到IPCThreadState。

從名字上看，IPCThreadState是“和跨進程通信（IPC）相關的線程狀態”。那麼很顯然，一個具有多個線程的進程裡應該會有多個IPCThreadState對象了，只不過每個線程只需一個IPCThreadState對象而已。這有點兒“局部單例”的意思。所以，在實際的代碼中，IPCThreadState對像是存放在線程的局部存儲區（TLS）裡的。

2.1 BpBinder的transact()動作

每當我們利用BpBinder的transact()函數發起一次跨進程事務時，其內部其實是調用IPCThreadState對象的transact()。BpBinder的transact()代碼如下：

```
status_t BpBinder::transact(uint32_t code, const Parcel& data,
Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    if (mAlive)
    {
        status_t status = IPCThreadState::self()->transact(mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
         return status;
    }

    return DEAD_OBJECT;
}
```

當然，進程中的一個BpBinder有可能被多個線程使用，所以發起傳輸的IPCThreadState對象可能並不是同一個對象，但這沒有關係，因為這些IPCThreadState對象最終使用的是同一個ProcessState對象。

2.1.1 調用IPCThreadState的transact()

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    . . . . . .
        // 把data數據整理進內部的mOut包中
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    . . . . . .

    if ((flags & TF_ONE_WAY) == 0)
    {
        . . . . . .
        if (reply)
        {
            err = waitForResponse(reply);
        }
        else
        {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        . . . . . .
    }
    else
    {
        err = waitForResponse(NULL, NULL);
    }

    return err;
}
```

IPCThreadState::transact()會先調用writeTransactionData()函數將data數據整理進內部的mOut包中，這個函數的代碼如下：

```
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
                                              int32_t handle, uint32_t code,
                                              const Parcel& data, status_t* statusBuffer)
{
    binder_transaction_data tr;

    tr.target.handle = handle;
    tr.code = code;
    tr.flags = binderFlags;
```

```
    tr.cookie = 0;
    tr.sender_pid = 0;
    tr.sender_euid = 0;

    . . . . . .
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()* sizeof (size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    . . . . . .

    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof (tr));

    return NO_ERROR;
}
```

接著IPCThreadState::transact()會考慮本次發起的事務是否需要回复。“不需要等待回复的”事務，在其flag標誌中會含有TF_ONE_WAY，表示一去不回頭。而“需要等待回复的”，則需要在傳遞時提供記錄回复信息的Parcel對象，一般發起transact()的用戶會提供這個Parcel對象，如果不提供，transact()函數內部會臨時構造一個假的Parcel對象。

上面代碼中，實際完成跨進程事務的是waitForResponse()函數，這個函數的命名不太好，但我們也不必太在意，反正Android中寫得不好的代碼多了去了，又不只多這一處。waitForResponse()的代碼截選如下：

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    int32_t cmd;
    int32_t err;

    while (1)
    {
        // talkWithDriver()內部會完成跨進程事務
        if ((err = talkWithDriver()) < NO_ERROR)
             break ;

        // 事務的回复信息被記錄在mIn中，所以需要進一步分析這個回复
        . . . . . .
        cmd = mIn.readInt32();
        . . . . . .
        switch (cmd)
        {
        case BR_TRANSACTION_COMPLETE:
             if (!reply && !acquireResult) goto finish;
             break ;

        case BR_DEAD_REPLY:
            err = DEAD_OBJECT;
            goto finish;

        case BR_FAILED_REPLY:
            err = FAILED_TRANSACTION;
            goto finish;
        . . . . . .
        . . . . . .
        default :
             //注意這個executeCommand()噢，它會處理BR_TRANSACTION的。
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
             break ;
        }
    }

finish:
    . . . . . .
    return err;
}
```

2.1.2 talkWithDriver()

waitForResponse()中是通過調用talkWithDriver()來和binder驅動打交道的，說到底會調用ioctl()函數。因為ioctl()函數在傳遞BINDER_WRITE_READ語義時，既會使用“輸入buffer”，也會使用“輸出buffer”，所以IPCThreadState專門搞了兩個Parcel類型的成員變量：mIn和mOut。總之就是，mOut中的內容髮出去，發送後的回複寫進mIn。

talkWithDriver()的代碼截選如下：

```
status_t IPCThreadState::talkWithDriver( bool doReceive)
{
    . . . . . .
    binder_write_read bwr;

    . . . . . .
    bwr.write_size = outAvail;
    bwr.write_buffer = ( long unsigned int )mOut.data();
    . . . . . .
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = ( long unsigned int )mIn.data();
    . . . . . .
    . . . . . .
    do
    {
        . . . . . .
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        . . . . . .
    } while (err == -EINTR);

    . . . . . .
    . . . . . .
    return err;
}
```

看到了嗎？mIn和mOut的data會先整理進一個binder_write_read結構，然後再傳給ioctl()函數。而最關鍵的一句，當然就是那句ioctl()了。此時使用的文件描述符就是前文我們說的ProcessState中記錄的mDriverFD，說明是向binder驅動傳遞語義。BINDER_WRITE_READ表示我們希望讀寫一些數據。

至此，應用程序通過BpBinder向遠端發起傳輸的過程就交代完了，數據傳到了binder驅動，一切就看binder驅動怎麼做了。至於驅動層又做了哪些動作，我們留在下一篇文章再介紹。

如需轉載本文內容，請註明出處。

http://my.oschina.net/youranhongcha/blog/152233

謝謝。

- « 紅茶一杯話Binder（ServiceManager篇）
- 紅茶一杯話Binder（傳輸機制篇_中） »

共有3 條網友評論

- 
1樓：翠屏阿姨發表於2013-08-13 08:56 回复此評論
我是沙發，曾經看過沒看懂，今天趁著這篇文章再看看，大贊

- 
2樓：enull發表於2013-08-14 17:39 回复此評論
Mark一下，自學中，感謝。

- 
3樓：RenKaidi (Android)發表於2013-08-19 11:34 回复此評論
不明覺厲！

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）
- 2013/08/02 紅茶一杯話Binder（ServiceManager篇...
- 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...