Binder之一：Binder Driver概览：《android框架揭秘读书笔记》

1.第一部分 Binder之一：Binder Driver概览：《android框架揭秘读书笔记》

Binder Driver的分析

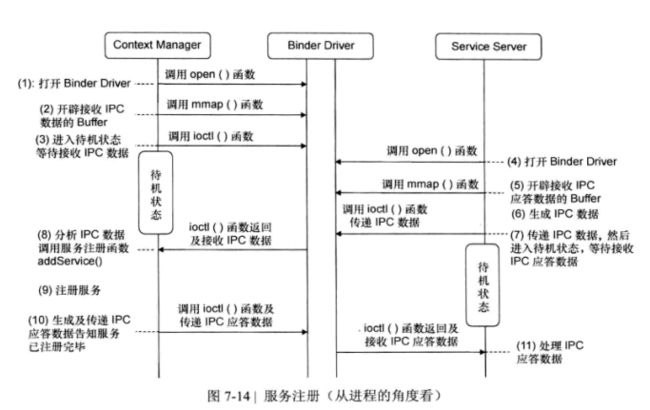
1.从进程的角度看服务的使用

客户端使用服务端要经历的三个阶段

* 服务注册（服务端与Context Manager之间的IPC）
* 服务检索（客户度与Context Manager之间的IPC）
* 服务使用（客户端与服务端之间的IPC）

1.1 服务注册

**服务注册：把服务注册到Context Manager的服务目录中**

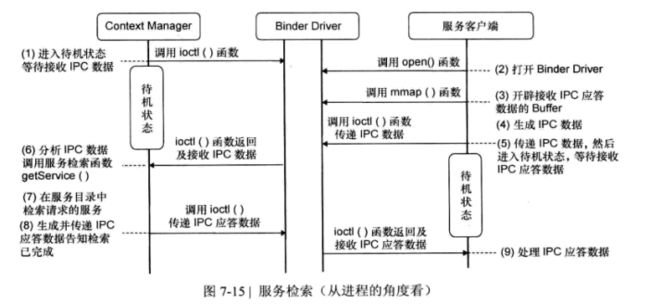
[](https://img.it610.com/image/info10/ddacf5c8055c4c428491abf41f2bb159.jpg)

服务注册.png

(1)-(3)servicemanager调用open函数，打开binder driver，然后调用mmap函数，在内核空间开辟一块用于接收ipc数据的buffer，再调用ioctl函数进入待机状态。  
(4)-(5)为了注册服务，服务端先打开binder driver，而后调用mmap函数，确定一块buffer，用于接收ipc应答数据。  
(6)服务端生成ipc数据，ipc数据包含rpc数据(保存要注册的服务名称)，prc代码(servicemanager的注册函数ADD\_SERVICE)，Handle(servicemanager的Handle值为0)三个部分  
(7)服务端调用ioctl函数向binder driver传递IPC数据，Binder Driver将数据再传递给Service Manager  
(8)-(9)Service Manager分析IPC数据中的RPC代码，并根据RPC代码调用相应的服务注册函数，而后使用RPC数据中的服务名称，将制定的服务注册到目录中。  
(10)服务注册完成后，Service Manager会生成应答数据，并传递给服务端，告知服务已经注册。

1.2 服务检索

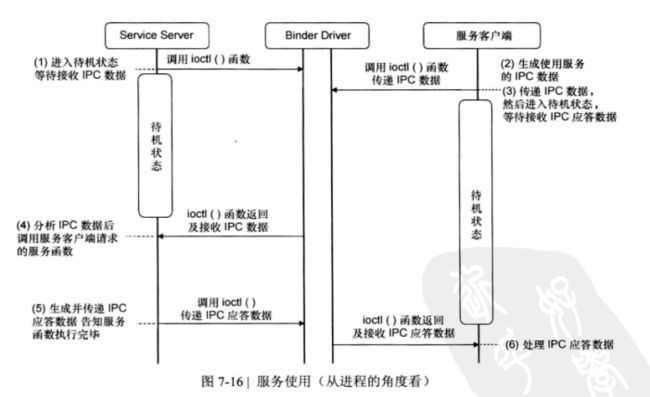
服务检索：客户端使用服务端的服务时，需要向ServiceManager请求服务的编号，这个过程叫服务检索。  
在该阶段，客户端会调用ServiceManager的服务检索函数，客户端首先生成调用ServiceManager的IPC数据，然后传递给Binder Driver，然后Binder Driver将IPC数据传递给ServiceManager，ServiceManager接收到IPC数据中的RPC代码调用相应的函数。

[](https://img.it610.com/image/info10/1a36a104b3f84c00b1bc3a54bd22b025.jpg)

服务检索.png

1:ServiceManager处于待机状态，等待接收IPC数据  
2-4：为了向ServiceManager传递IPC数据，客户端打开BinderDriver，而后调用mmap函数准备一块buffer，用于接收IPC应答数据，客户端生成IPC数据，IPC数据包括RPC数据（保存要使用的服务名称），RPC代码（ServiceManger的服务检索函数GET\_SERVICE），Handle（ServiceManager的handle为0）三部分。  
5：服务端调用ioctl函数向BinderDriver传递数据，BinderDriver根据IPC数据中的Handle，将IPC数据传递给ServiceManager  
6:-8：ServiceManager分析IPC数据，根据RPC代码调用服务检索函数，根据RPC数据中保存的服务名称在服务目录中检索服务，查找到服务BInder节点编号，ServiceManager将查找到的Binder节点编号插入到IPC应答数据中。  
9：客户端接收IPC应答数据，获得指定服务的Binder节点编号。

1.3 服务使用

[](https://img.it610.com/image/info10/3da5ddda83b54ad0b40a9237063d7279.jpg)

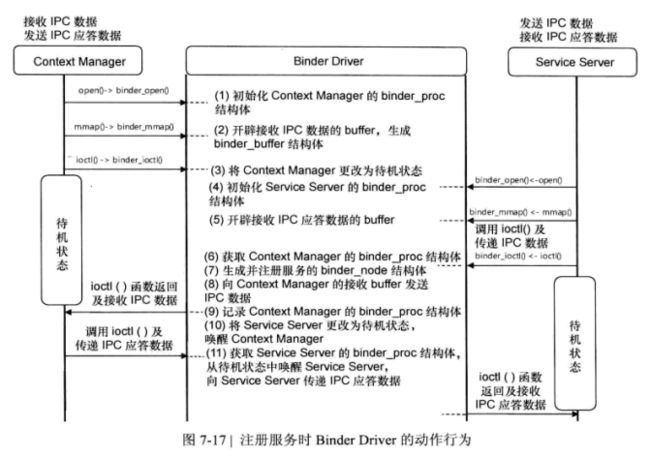
服务使用.png

1.服务端进入待机状态，等待接收IPC数据，并且当处理完接收的IPC数据之后，它将再次进入待机状态，等待接收新的数据  
2-3：为了使用服务，客户端首先生成IPC数据，IPC数据有三个部分组成，包括RPC代码（指定服务函数），RPC数据（服务函数的参数），Handle（指定哪个服务）；然后调用ioctl函数，将IPC数据传递给BinderDriver，最后BinderDriver根据IPC数据中的Handle把IPC数据传递给服务端  
4-5：服务端分析IPC数据中的RPC代码，根据RPC代码调用服务函数，IPC数据中的RPC数据是函数参数，在函数的内部使用，当指定的函数执行完毕之后，服务端会生成IPC应答数据，其中包括Binder节点编号，而后将IPC应答数据发送给客户端，告知指定的服务函数已经执行完成。  
6：客户端接收IPC数据，并进行处理

2.从BinderDriver角度看服务的使用：BinderDriver的运行过程分析

2.1服务注册

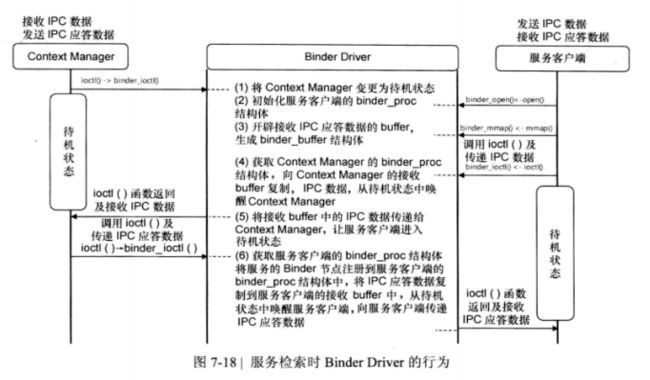
**ServiceManager**先于其他所有服务运行，它最先使用BinderDriver，进入待机状态，等待服务的注册和客户端对服务的检索请求。

[](https://img.it610.com/image/info10/b8069721643743da94b3c9e86d9bf660.jpg)

注册服务时BinderDriver的行为.png

1-3:描述的是ServiceManager进入待机状态前使用BinderDriver的过程；binder\_proc是一个结构体，这个结构体里包含了一系列指针，以便访问其他结构体，每个进程都有一个binder\_proc结构体，通过这个结构体，可以查找到另一个进程，也可以查找到进程接收IPC数据的buffer  
1：ServiceManager通过调用open函数调用binder\_open函数，BinderDriver使用binder\_open函数诶ServiceManager生成并初始化binder\_proc  
2.ServiceManager在内核开辟一块buffer，用于接收IPC数据：ServiceManger通过调用mmap函数调用BinderDriver的binder\_mmap函数，binder\_mmap函数在内核空间中分配一块用于接收IPC数据的buffer，并保存到binder\_buffer结构体，而binder\_buffer结构体会被注册到binder\_proc结构体里  
3.ServiceManger通过调用ioctl函数调用BinderDriver的binder\_ioctl函数，在binder\_ioctl中，ServiceManger将处于待机状态，知道另一个进程向其发送IPC数据。  
4.服务端通过open调用BinderDriver的binder\_open函数，BinderDriver为服务生成binder\_proc结构体，并将其初始化。  
5.服务端通过mmap函数调用BinderDriver的binder\_mmap函数开辟一块空间用于接收IPC应答数据的buffer，并将其保存到binder\_buffer结构体，而后服务端生成IPC数据，用来调用ServiceManager的服务注册函数。  
6.为了注册服务，服务端将IPC数据传递给BinderDriver；BinderDriver通过IPC数据中的Handle查找到ServiceManager的binde\_node和binder\_proc  
7.为注册的服务生成binder\_node结构体，BinderDriver将binder\_node分别注册到服务的binder\_proc和ServiceManager的binder\_proc中。  
8.BinderDriver将在第六步中查找到的ServiceManager的binder\_proc结构体中查找注册在binder\_buffer中的buffer，并传递IPC数据。  
9.BinderDriver会记下IPC数据发送进程也就是服务进程的binder\_proc结构体，以便在ServiceManager向服务进程发送应答数据时查找服务的binder\_proc。  
10.BinderDriver让服务处于待机状态，并将ServiceManager从待机状态唤醒，传递IPC数据，ServiceManger从BinderDriver中接收来自服务的IPC数据，ServiceManger根据接收到的数据注册指定的服务，而后生成IPC应答数据，并将其传递给BinderDriver。  
11.BinderDriver会在服务的binder\_proc结构体中查找binder\_buffer里的buffer，并传递IPC应答数据，唤醒服务端，服务端被唤醒后接收IPC数据，并根据IPC数据做相应处理。

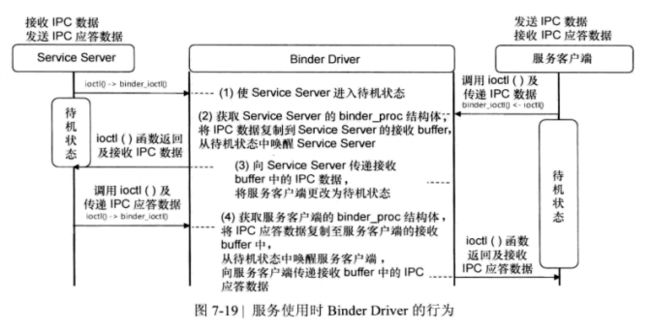
2.2 服务检索

[](https://img.it610.com/image/info10/5ea5660639ba43a499a9162b616e80f4.jpg)

服务检索中BinderDriver的行为.png

1.ServiceManager进入待机状态，等待接收IPC数据  
2-3客户端生成并初始化binder\_proc结构体，然后开辟一块buffer用于接收IPC数据  
4.客户端通过ioctl函数调用BinderDriver的binder\_ioctl函数，BinderDriver会查找Servicemanager的binder\_proc结构体，这里不生成binder\_node结构体，只是将IPC数据拷贝到ServiceManager的接收buffer中，并记下客户端的binder\_proc结构体，以便查找IPC应答数据的接收端。  
5.BinderDriver让客户端处于待机状态，并唤醒处于等待状态的ServiceManger，而后接收IPC数据，ServiceManager在服务目录中查找请求的服务，把服务的编号插入IPC数据中，并将IPC数据传递给BinderDriver。  
6.BinderDriver根据接收到的服务编号查找相应的binder\_node结构体，而后将查找到的binder\_node结构体注册到客户端的binder\_proc中，binder\_node结构体用于在服务使用阶段查找服务端的binder\_proc结构体。  
7.BinderDriver将在第六步中注册的binder\_node结构体的编号插入到IPC应答数据中，传递给客户端，唤醒客户端，在使用服务时，客户端会将编号作为Handle使用。

2.3服务使用

[](https://img.it610.com/image/info10/19bf6cb3f3f44eb69284127ea5d3e23c.jpg)

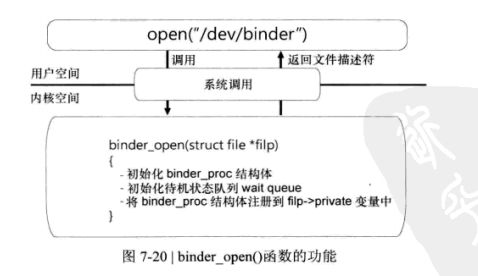
服务使用时BinderDriver的行为.png

1.服务端处于待机状态，用于接收IPC数据  
2.客户端把从服务检索阶段获取到的Binder节点编号保存到IPC数据的Handle中，生成IPC数据后传递给BinderDriver，BinderDriver根据IPC数据中的Handle查找相应的binder\_node结构体，并根据此查找到注册有binder\_node结构体的服务进程的binder\_proc结构体，然后通过注册在binder\_proc结构体中的binder\_buffer结构体，将IPC数据拷贝的buffer中  
3.BinderDriver让客户端进入待机状态，唤醒处于待机状态的服务端，服务端接收IPC数据，并通过IPC数据中的RPC代码RPC数据调用相应的服务函数  
4.服务函数执行完毕后，服务端会生成IPC应答数据，并将其传递给BinderDriver，然后BinderDriver将数据传递给客户端

3.BinderDriver的函数分析

3.1 binder\_open函数

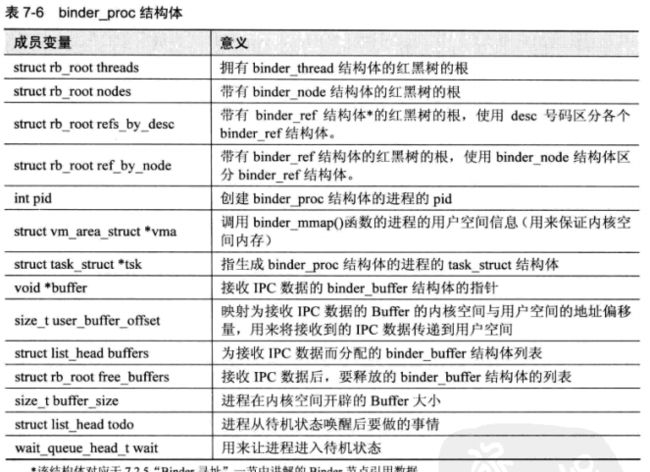
**binder\_open函数为打开BinderDriver的进程生成并初始化结构体binder\_proc；并且初始化等待队列（用来将进程切换到待机状态下)和todo队列（用来在进程被唤醒后执行其中的任务）**

[](https://img.it610.com/image/info10/571ca7338d48449d97d6a53493909050.jpg)

binder\_open函数的功能.png

3.1.1 binder\_proc结构体

**binder\_proc结构体用来管理BinderIPC所需要的各种信息，包括打开BinderDriver的进程信息，接收IPC数据的buffer信息等，此外，binder\_proc拥有其他结构体的指针，为访问其他结构体提供了方便，被称为BinderDriver的根结构体**

[](https://img.it610.com/image/info10/83a6d268c4e84c019f72314f6280fab2.jpg)

binder\_proc结构体.png

3.1.2 binder\_open源码分析

static HLIST\_HEAD(binder\_procs);

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

{

struct binder\_proc \*proc;

binder\_debug(BINDER\_DEBUG\_OPEN\_CLOSE, "binder\_open: %d:%d\n",

current->group\_leader->pid, current->pid);

//为binder\_proc开辟内存空间，也就是初始化binder\_proc

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

if (proc == NULL)

return -ENOMEM;

get\_task\_struct(current);

proc->tsk = current;//1

INIT\_LIST\_HEAD(&proc->todo);//2

init\_waitqueue\_head(&proc->wait);//2

proc->default\_priority = task\_nice(current);

mutex\_lock(&binder\_lock);

binder\_stats\_created(BINDER\_STAT\_PROC);

//把当前生成的proc加入到binder\_procs中

hlist\_add\_head(&proc->proc\_node, &binder\_procs);

proc->pid = current->group\_leader->pid;

INIT\_LIST\_HEAD(&proc->delivered\_death);

filp->private\_data = proc;//3

mutex\_unlock(&binder\_lock);

if (binder\_debugfs\_dir\_entry\_proc) {//4

char strbuf[11];

snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);

proc->debugfs\_entry = debugfs\_create\_file(strbuf, S\_IRUGO,

binder\_debugfs\_dir\_entry\_proc, proc, &binder\_proc\_fops);

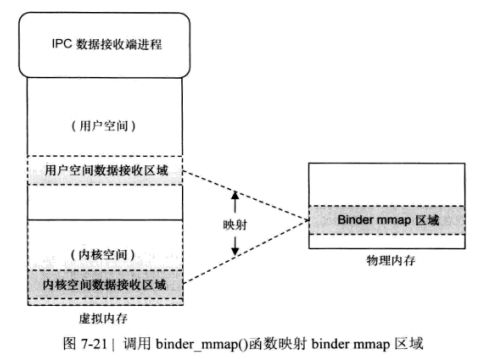
}

return 0;

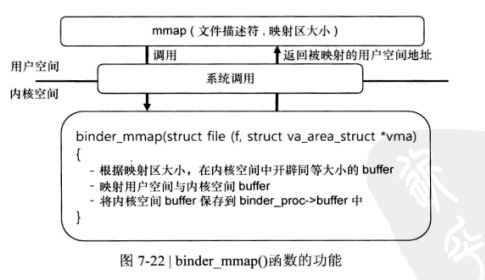
}

1.task\_struct结构体中包含打开BinderDriver的进程的信息，proc->tsk = current将task\_struct结构体变量current注册到binder\_proc结构体变量的tsk成员中，该变量用来输出当前正在进行BinderIPC的进程。  
2.BinderDriver接收到IPC数据后，将要执行的任务保存到proc->todo列表中，第二条语句初始化待机队列，以便将打开的BinderDriver进程切换到待机状态。  
3.将proc放到flip中，以便在后面使用，后面使用时只需要用proc=flip->private\_data就可以获取到binder\_proc  
4.在proc/binder/proc目录下生成文件，以便初始化binder\_proc结构体或显示与BInderIPC相关的信息，在Android启动后，查看该目录和目录下的文件，可以查看到BinderIPC的进程信息。

3.2 binder\_mmap函数

[](https://img.it610.com/image/info10/351ab188af8143dc8e724fc99c219a94.jpg)

映射.png

[](https://img.it610.com/image/info10/6af061e0a2d24908a5aec19cb20dcf9d.jpg)

binder\_mmap函数功能.png

static int binder\_mmap(struct file \*filp, struct vm\_area\_struct \*vma)

{

int ret;

struct vm\_struct \*area;

//1:

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

const char \*failure\_string;

struct binder\_buffer \*buffer;

if ((vma->vm\_end - vma->vm\_start) > SZ\_4M)

vma->vm\_end = vma->vm\_start + SZ\_4M;

binder\_debug(BINDER\_DEBUG\_OPEN\_CLOSE,

"binder\_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",

proc->pid, vma->vm\_start, vma->vm\_end,

(vma->vm\_end - vma->vm\_start) / SZ\_1K, vma->vm\_flags,

(unsigned long)pgprot\_val(vma->vm\_page\_prot));

if (vma->vm\_flags & FORBIDDEN\_MMAP\_FLAGS) {

ret = -EPERM;

failure\_string = "bad vm\_flags";

goto err\_bad\_arg;

}

vma->vm\_flags = (vma->vm\_flags | VM\_DONTCOPY) & ~VM\_MAYWRITE;

if (proc->buffer) {

ret = -EBUSY;

failure\_string = "already mapped";

goto err\_already\_mapped;

}

area = get\_vm\_area(vma->vm\_end - vma->vm\_start, VM\_IOREMAP);//2

if (area == NULL) {

ret = -ENOMEM;

failure\_string = "get\_vm\_area";

goto err\_get\_vm\_area\_failed;

}

proc->buffer = area->addr; //3

proc->user\_buffer\_offset = vma->vm\_start - (uintptr\_t)proc->buffer;//4

#ifdef CONFIG\_CPU\_CACHE\_VIPT

if (cache\_is\_vipt\_aliasing()) {

while (CACHE\_COLOUR((vma->vm\_start ^ (uint32\_t)proc->buffer))) {

printk(KERN\_INFO "binder\_mmap: %d %lx-%lx maps %p bad alignment\n", proc->pid, vma->vm\_start, vma->vm\_end, proc->buffer);

vma->vm\_start += PAGE\_SIZE;

}

}

#endif

proc->pages = kzalloc(sizeof(proc->pages[0]) \* ((vma->vm\_end - vma->vm\_start) / PAGE\_SIZE), GFP\_KERNEL);

if (proc->pages == NULL) {

ret = -ENOMEM;

failure\_string = "alloc page array";

goto err\_alloc\_pages\_failed;

}

proc->buffer\_size = vma->vm\_end - vma->vm\_start;

vma->vm\_ops = &binder\_vm\_ops;

vma->vm\_private\_data = proc;

//5

if (binder\_update\_page\_range(proc, 1, proc->buffer, proc->buffer + PAGE\_SIZE, vma)) {

ret = -ENOMEM;

failure\_string = "alloc small buf";

goto err\_alloc\_small\_buf\_failed;

}

buffer = proc->buffer;//6

INIT\_LIST\_HEAD(&proc->buffers);

list\_add(&buffer->entry, &proc->buffers);

buffer->free = 1;

binder\_insert\_free\_buffer(proc, buffer);//7

proc->free\_async\_space = proc->buffer\_size / 2;

barrier();

proc->files = get\_files\_struct(current);

proc->vma = vma;

/\*printk(KERN\_INFO "binder\_mmap: %d %lx-%lx maps %p\n",

proc->pid, vma->vm\_start, vma->vm\_end, proc->buffer);\*/

return 0;

err\_alloc\_small\_buf\_failed:

kfree(proc->pages);

proc->pages = NULL;

err\_alloc\_pages\_failed:

vfree(proc->buffer);

proc->buffer = NULL;

err\_get\_vm\_area\_failed:

err\_already\_mapped:

err\_bad\_arg:

printk(KERN\_ERR "binder\_mmap: %d %lx-%lx %s failed %d\n",

proc->pid, vma->vm\_start, vma->vm\_end, failure\_string, ret);

return ret;

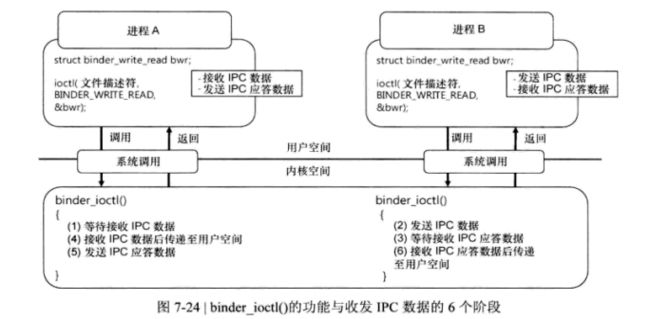
}

1.binder\_mmap函数的第一个参数是file结构体指针，它指向BinderDriver的文件描述符，file结构体指针的private\_data成员变量是当前进程的binder\_proc结构体，在binder\_open中已经被赋值；binder\_mmap第二个参数是用户空间的buffer信息，由mmap函数与内核空间映射在一起。  
2.get\_vm\_area是一个内核函数，通过该函数可以向系统申请一块可用的虚拟内存空间，也就是在内核中申请并保留一块连续的内核虚拟内存区域，若内核的VMALLOC区域有一块符合指定尺寸的空间，则生成一个vm\_struct结构体，在把空间的起始地址付给vm\_struct结构体的addr变量后返回  
3.调用get\_vm\_area函数将返回vm\_struct结构体，并将其赋值给area变量，此时area->addr中存放着内核空间的起始地址，使用proc->buffer=area->addr后，proc->buffer其实存放了内核空间中接收IPC数据的Buffer的起始地址。  
4.这行代码用来计算用户空间buffer地址与内核空间buffer地址的偏移。proc->user\_buffer\_offset是int类型变量，是一个负值，偏移信息用来告诉进程用户空间的buffer地址，在BinderDriver接收IPC数据后，用户空间的Buffer与保存IPC数据的内核空间映射在一起。  
5.调用binder\_update\_page\_range函数，分配物理页，并将存在于物理内存与虚拟内存的内核空间的接收Buffer与用户空间的接收Buffer映射起来  
6.该语句用来将IPC数据保存到binder\_buffer中  
7.binder\_insert\_free\_buffer函数，将bindeer\_buffer结构体注册到当前进程的binder\_proc结构体的free\_buffers中，free\_buffers变量是指接收IPC数据的buffer，根据IPC数据的大小，分配不同的free\_buffers，以便IPC接收IPC数据

3.3 binder\_ioctl函数

[](https://img.it610.com/image/info10/2281aa89f36443c4aa13417232642dbd.jpg)

binder\_ioctl所支持的命令.png

[](https://img.it610.com/image/info10/299ba285e24e446c962d34a1f5a6d027.jpg)

binder\_ioctl收发数据的流程.png

3.3.1 binder\_write\_read结构体

struct binder\_write\_read {

//用于发送数据

signed long write\_size; /\* bytes to write \*/

signed long write\_consumed; /\* bytes consumed by driver \*/

unsigned long write\_buffer;

//用于接收数据

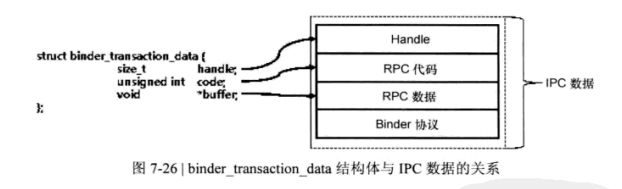
signed long read\_size; /\* bytes to read \*/

signed long read\_consumed; /\* bytes consumed by driver \*/

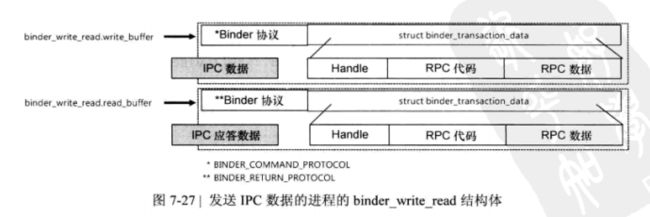
unsigned long read\_buffer;

};

3.3.2 binder\_transaction\_data结构体（Handle，RPC代码，RPC数据保存在该结构体中）

[](https://img.it610.com/image/info10/b2838377021d48d5af7bb7ab4db24e9f.jpg)

binder\_transaction\_data与IPC数据的关系.png

[](https://img.it610.com/image/info10/063faf70b7b94527b14feca1029ebf66.jpg)

binder\_write\_read与binder\_transaction\_data的关系.png

以发送IPC数据进程的buffer来分析  
binder\_write\_read结构体中包含着用户空间生成的IPC数据，BinderDriver也拥有一个相同的结构体，用户空间设置完binder\_write\_read结构体数据后，调用ioctl函数传递给BInderDrier，BinderDrier调用copy\_from\_user函数将用户空间中的数据拷贝到自身的binder\_write\_read结构体中，相反，这IPC应答阶段，BinderDrier将调用copy\_to\_user函数，将自身的binder\_write\_read结构体中的数据拷贝到用户空间。

struct binder\_transaction\_data {

/\* The first two are only used for bcTRANSACTION and brTRANSACTION,

\* identifying the target and contents of the transaction.

\*/

union {

size\_t handle; /\* target descriptor of command transaction \*/

void \*ptr; /\* target descriptor of return transaction \*/

} target;

void \*cookie; /\* target object cookie \*/

unsigned int code; /\* transaction command \*/

/\* General information about the transaction. \*/

unsigned int flags;

pid\_t sender\_pid;

uid\_t sender\_euid;

size\_t data\_size; /\* number of bytes of data \*/

size\_t offsets\_size; /\* number of bytes of offsets \*/

/\* If this transaction is inline, the data immediately

\* follows here; otherwise, it ends with a pointer to

\* the data buffer.

\*/

union {

struct {

/\* transaction data \*/

const void \*buffer;

/\* offsets from buffer to flat\_binder\_object structs \*/

const void \*offsets;

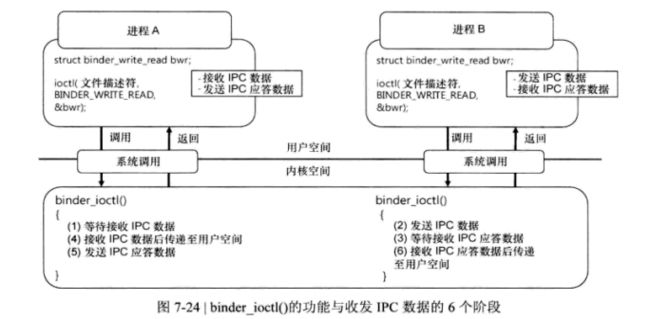
} ptr;

uint8\_t buf[8];

} data;

};

3.3.3 概览

[](https://img.it610.com/image/info10/09f4320dea21455fbaca04ab0340a21e.jpg)

binder\_ioctl收发数据的流程.png

Binder架构

http://gityuan.com/images/binder/java\_binder/java\_binder.jpg

Binder进程和线程

http://gityuan.com/images/binder/summary/binder\_proc\_relation.png