**Bn-通过ServiceManager分析binder机制**

**1、启动ServiceManager**

ServiceManager是通过注册在init.rc中，而被启动的，device/whaley/generic/etc/init.rc中定义如下：

service servicemanager /system/bin/servicemanager

class core //class为core的服务，系统一般先启动class为core的服务，然后再启动class为main的服务

user system //把用户名切换到system，默认是root

group system //把组名切换到system

critical //说明该服务是个对于设备很关键的服务，如果4分钟内退出大于4次，则系统将重启并进入recovery恢复模式

onrestart restart healthd //重启时执行后面的语句restart healthd

onrestart restart zygote

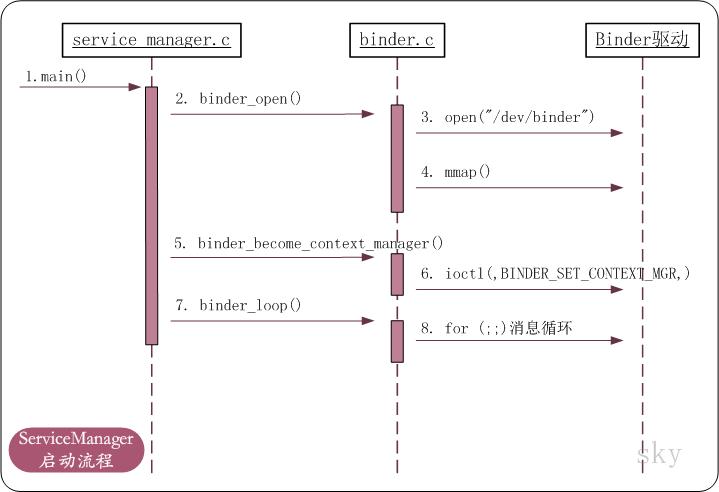
onrestart restart media

onrestart restart surfaceflinger

onrestart restart drm

servicemanager会在kernel启动完成后执行到android init.rc时自动启动，进入它的main函数。

**2、service\_manager.c，main函数分析：**



代码路径：frameworks/native/cmds/servicemanager/ service\_manager.c

frameworks/native/cmds/servicemanager/ binder.c

**2.1、binder\_open**

此方法主要和binder驱动交互，binder驱动路径：[device/hisilicon/bigfish/sdk/source/kernel/linux-3.18.y/drivers/staging/android/](http://androidxref.moretv.cn/source/xref/hisi-l-551/device/hisilicon/bigfish/sdk/source/kernel/linux-3.18.y/drivers/staging/android/)binder.c

驱动注册device\_initcall(binder\_init)，misc\_register(&binder\_miscdev)注册binder设备节点"/dev/binder"，绑定binder设备节点的操作方法binder\_fops：

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,

};

static struct miscdevice binder\_miscdev = {

.minor = MISC\_DYNAMIC\_MINOR,

.name = "binder",

.fops = &binder\_fops

};

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

{

ret = misc\_register(&binder\_miscdev);

}

device\_initcall(binder\_init);

**2.1.1、open("/dev/binder", O\_RDWR)**

打开binder设备，把句柄保存在struct binder\_state的fd中：

{

int fd;

void \*mapped;

size\_t mapsize;

};

Mapsize为映射内存的大小，这里是(128\*1024)，128kb。

bs->mapped = mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0)这里开始映射内存，并返回起始地址到mapped。

进入binder驱动看看open具体做了哪些工作，打开"/dev/binder"设备节点会走到binder驱动的方法.open = binder\_open。

binder驱动binder\_open：新建结构体binder\_proc变量指针proc，并为它分配连续的堆空间，binder\_proc是描述Binder进程的上下文信息结构体，这里就是ServiceManager这个服务进程。填充proc：tsk（线程号）、pid（进程号）等，初始化proc的待处理事务列表proc->todo和等待队列proc->wait。最后把proc保存在struct file \*filp->private\_data中，private\_data保存了ServiceManager的进程上下文环境信息，在fd中。

struct file \*filp：代表一个打开的文件，系统中每个被打开的文件在内核空间都有一个关联的struct file。它由内核在打开文件时创建，并传递给在文件上进行操作的任何函数。在文件的所有实例都关闭后，内核释放这个数据结构。用户空间binder\_state结构体中保存的句柄fd就是这个文件结构体的指针。

所以open("/dev/binder", O\_RDWR)作用是在内核中创建一个binder\_proc结构体变量保存ServiceManager本身的进程上下文信息，并把它保存在打开"/dev/binder"时所创建的文件结构体的private\_data中，最后将该进程上下文信息proc保存到"全局哈希表binder\_procs"中。

**2.2、mmap，mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0)**

分配128kb的内存，并将ServiceManager进程的虚拟地址和内核虚拟地址映射到同一块物理内存上，地址保存在filp->private\_data（binder\_proc，open时创建）中。这样ServiceManager进程的用户空间地址和内核空间地址就映射到了同一个物理页面，当Client中的数据拷贝到内核空间时，也就相当于拷贝到了ServiceManager进程中，binder驱动只需要传递一个指针给用户空间，不需要copy指针的内容数据。因此，在Binder通信机制中，数据传输时，只需要1次内存拷贝，即把client端的数据拷贝到binder实体（内核中）。

**2.3、binder\_become\_context\_manager**

通过ioctl走到binder驱动binder\_ioctl代码，第一次进来binder\_get\_thread中新建一个binder\_thread对象，里面保存当前进程信息，初始化待处理事务列表todo和等待队列wait，最后赋值到binder\_proc的threads红黑树中，就是线程信息了，thread->looper |= BINDER\_LOOPER\_STATE\_NEED\_RETURN;新建线程的时候有一个BINDER\_LOOPER\_STATE\_NEED\_RETURN值放进thread->looper里，在binder\_ioctl函数返回之前把此标志位清除if (thread) thread->looper &= ~BINDER\_LOOPER\_STATE\_NEED\_RETURN;

case BINDER\_SET\_CONTEXT\_MGR:

ret = binder\_ioctl\_set\_ctx\_mgr(filp);

ret = security\_binder\_set\_context\_mgr(proc->tsk);

break;

binder\_ioctl\_set\_ctx\_mgr第一次进来，会binder\_new\_node，新建一个binder\_node实例，保存进程信息proc和ptr（0）、cookie（0），work.type是BINDER\_WORK\_NOD，最后把新建的binder node保存在binder\_proc对象的proc->nodes红黑树中，此binder\_node实例同时保存在static struct binder\_node \*binder\_context\_mgr\_node中，这个全局变量就是ServiceManager对应的内核中的Binder实体了。

特别需要注意的是，对于每一个Server，Binder驱动都会为其分配一个binder\_node对象（内核中的Binder实体），并保存在proc->nodes红黑树中，所以通过进程结构体proc可以找到进程内所有的binder实体。对于ServiceManager这个Binder上下文管理者而言，Binder驱动更是会将它的Binder实体保存到全局变量binder\_context\_mgr\_node中。

**2.3、binder\_loop**

**2.3.1、binder\_write\_read**

这是 Binder通信中常用的数据结构体binder\_write\_read。bwr.write\_size>0，表示有数据发送给Binder驱动，而发送的数据就保存在bwr.write\_buffer中，binder\_write这里就是BC\_ENTER\_LOOPER指令，bwr.write\_consumed则表示已经被处理的数据的大小。bwr.read\_XXX则是用来保存Binder驱动返回的信息。

binder\_loop中的binder\_write(bs, readbuf, sizeof(uint32\_t))就是告诉binder驱动ServiceManager即将进入消息循环状态。驱动中只是把thread->looper的状态设为BINDER\_LOOPER\_STATE\_ENTERED，表示这个线程进入消息循环了。

随后进入for死循环，for循环里面因为bwr.write\_XXX都是0，所以是读取模式：

bwr.write\_size = 0;

bwr.write\_consumed = 0;

bwr.write\_buffer = 0;

bwr.read\_size = sizeof(readbuf);

bwr.read\_consumed = 0;

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

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

**2.3.2、看看Binder驱动中的处理**

case BINDER\_WRITE\_READ:

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

binder\_ioctl\_write\_read中主要流程：

1、copy\_from\_user，把用户空间的数据拷贝到内核空间。将ServiceManager中调用ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr)时的bwr数据拷贝到Binder驱动中的临时变量bwr中。

2、if (bwr.write\_size > 0) ret = binder\_thread\_write()

if (bwr.read\_size > 0) ret = binder\_thread\_read()

根据bwr.write\_size和bwr.read\_size判断是读还是写操作。

3、copy\_to\_user，把bwr内容从内核空间拷贝到用户空间。

**binder\_thread\_write：**

从brw.write\_buffer中读取4个字节作为cmd。这4个字节就是ServiceManager传递的指令BC\_ENTER\_LOOPER。在BC\_ENTER\_LOOPER对应的switch分支中，就是将BINDER\_LOOPER\_STATE\_ENTERED加入到thread->looper中。即告诉Binder驱动，ServiceManager进程进入了消息循环状态。

**binder\_thread\_read：**

ServiceManage的binder\_loop进入for循环后会走到binder驱动的binder\_thread\_read。

Todo是空，所以wait\_for\_proc\_work为true，thread->looper |= BINDER\_LOOPER\_STATE\_WAITING，looper已经有两个值了，是0x12。non\_block是默认值，根据逻辑会走到wait\_event\_interruptible\_exclusive(proc->wait, binder\_has\_proc\_work(proc, thread))。此时ServiceManager进程中没有待处理事务，因此binder\_has\_proc\_work(proc, thread)返回false。从而当前线程进入中断等待状态，ServiceManager进程陷入内核后休眠，等待其它进程将ServiceManager唤醒。

**2.4、binder\_parse**

**2.4.1、addService时，被client唤醒**

1、看看内核空间，从wait\_event\_interruptible\_exclusive处醒来，清除BINDER\_LOOPER\_STATE\_WAITING状态，进入while循环：

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

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

tr.cookie = target\_node->cookie;

这里得到一个binder实体，被MediaPlayerService进程加载到t中，是ServiceManager在驱动中的binder实体。并获取它的ptr和cookie，根据本文上面的分析，这两个值其实都是0。

tr.code = t->code;这code是addservice。

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

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

这里保存是那个进程发来的t事务，这里是MediaPlayerService进程，后面ServiceManager返回的时候会用到。

tr.data\_size = t->buffer->data\_size;

tr.offsets\_size = t->buffer->offsets\_size;

tr.data.ptr.buffer = (void \*)t->buffer->data + proc->user\_buffer\_offset;

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

这里用到了用户空间和内核空间虚拟地址共同映射到同一块物理内存的点，没有copy\_to\_user，直接修改指针，把内核指针变成用户指针，因为指向的内容是一样的。

将数据tr拷贝到用户空间，删除todo中的事务，更新bwr.read\_consumed的值，\*consumed = ptr – buffer。

看看cmd：休眠之前有一个BR\_NOOP，在while中获取目标进程binder node实体后又赋值了BR\_TRANSACTION。

2、用户空间进程从ioctl返回，由binder\_parse解析从内核空间读到的数据binder\_parse(bs, 0, (uintptr\_t) readbuf, bwr.read\_consumed, func)

bs：binder\_open得到的文件指针、映射的内存起始地址等信息

readbuf：binder驱动发送过来的数据

bwr.read\_consumed：表示已经处理过多少数据

func：svcmgr\_handler，主要功能do\_find\_service、do\_add\_service，service的添加和查找，把结果再返回给驱动中的Binder实体。注意：ServiceManager中只保存service的服务名和它的Binder实体的Binder引用的描述符；所以返回的只能是查找到的service的binder引用的描述符。

第一次进来得到cmd BR\_NOOP，什么都没有做，回到while再次把BR\_TRANSACTION读取出来，进入case BR\_TRANSACTION，执行下面函数：

res = func(bs, txn, &msg, &reply);这里进入svcmgr\_handler

binder\_send\_reply(bs, &reply, txn->data.ptr.buffer, res);这里把结果reply返回发送到Binder驱动。

**2.4.2、BR\_TRANSACTION时code流程，接2.4.1**

binder\_transaction\_data \*txn = (struct binder\_transaction\_data \*) ptr，这里把ptr转成binder\_transaction\_data 类型，struct binder\_txn是与binder\_transaction\_datad对应的结构体。

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

bio\_init\_from\_txn(&msg, txn); // 根据txn(Binder驱动反馈的信息)初始化msg

res = func(bs, txn, &msg, &reply); // 消息处理svcmgr\_handler

binder\_send\_reply(bs, &reply, txn->data, res); // 反馈消息给Binder驱动。

1、svcmgr\_handler

A、if (txn->target.handle != svcmgr\_handle) return -1，这里target.handle是0，继续向下执行。

b、从msg中获取MediaPlayerService 在addService 时封进去的数据IServiceManager::getInterfaceDescriptor()，与uint16\_t svcmgr\_id[] = {'a','n','d','r','o','i','d'

,'.','o','s','.','I','S','e','r','v','i','c','e','M','a','n','a','g','e','r'}对比。

c、继续走到case SVC\_MGR\_ADD\_SERVICE：

s = bio\_get\_string16(msg, &len);//这个s是”media.player”，MediaPlayerService addService的填进去的name

handle = bio\_get\_ref(msg);//获取MediaPlayerService在binder驱动中新建的binder引用的描述：obj->handle = binder\_ref->desc

2、do\_add\_service

先在svcinfo svclist（链表）中查找是否有name为media.player的service，有的话，如果不为0（不是ServiceManager），释放掉，执行svcinfo\_death，binder\_release(bs, si->handle)，写BC\_RELEASE和此handle到binder驱动，看看驱动中的处理：从当前proc的引用红黑树efs\_by\_desc中获取这个binder引用，然后把这个引用释放掉，就返回了。回到用户空间svcinfo\_death，重新赋值svcinfo si handle为0，到do\_add\_service再次赋值handle，让ServiceManager指向新建的binder引用描述符。

如果没有，新建svcinfo对象si，赋值MediaPlayerService的name和binder驱动传过来的引用描述符，注意它的death.func函数是svcinfo\_death，然后再将si添加到svclist链表。这样，MediaPlayerService就注册到Service Manager中了。

最后还有一个binder\_link\_to\_death。后面分析死亡通知的时候再讲。

3、binder\_send\_reply

ServiceManager处理完，发一个reply回binder驱动。BC\_FREE\_BUFFER和BC\_REPLY，

4、binder驱动

BC\_FREE\_BUFFER，释放内存。不介绍了。

看看BC\_REPLY：binder\_transaction

收到BC\_REPLY后，Binder驱动走进binder\_transaction，reply为真，target\_thread = in\_reply\_to->from;target\_proc = target\_thread->proc;获取目标进程，之前MediaPlayerService进程给ServiceManager发送事务t时把自己写在t中。

接下来新建struct binder\_transaction \*t和struct binder\_work \*tcomplete

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

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

t->to\_proc = target\_proc;

t->to\_thread = target\_thread;

t->code = tr->code;

和《3-通过MediaPlayerService分析binder机制-addService》中新建t事务把ServiceManager唤醒时类似。只不过这一次的目标进程变成了MediaPlayerService，这个t会加入到MediaPlayerService的todo列表中，t的work type是BINDER\_WORK\_TRANSACTION。新建的tcomplete事务是给当前进程的，即，ServiceManager，tcomplete的type是BINDER\_WORK\_TRANSACTION\_COMPLETE。拷贝ServiceManager返回的reply数据以及数据的偏移量到t->buffer->data、offp。由于reply中没有flat\_binder\_object数据，所以for循环没有执行，直接到了binder\_pop\_transaction，这里t->need\_reply = 0等，基本t是一个空的事务，只起到通知的作用。

将addService的反馈内容以待处理事务t的方式添加到MediaPlayerService的待处理事务队列当中，wake\_up\_interruptible (target\_wait)，并将MediaPlayerService进程唤醒，MediaPlayerService被唤醒后的操作可以参考**《3-通过MediaPlayerService分析binder机制-addService》**。Binder驱动数据处理完之后，现在ServiceManager进程要返回到用户空间了。

5、回到binder\_loop

循环，再次执行ioctl，当ServiceManager再次进入到Binder驱动，并通过binder\_ioctl()调用到binder\_thread\_read()时。由于此时的ServiceManager线程中有一个类型为BINDER\_WORK\_TRANSACTION\_COMPLETE的待处理事务；于是，便取出该事务进行执行。执行完毕之后，将该事务从Service Manager的待处理事务队列中删除，并反馈cmd=BR\_TRANSACTION\_COMPLETE信息给ServiceManager守护进程。ServiceManager守护进程收到Binder驱动的反馈后，解析出BR\_TRANSACTION\_COMPLETE，该指令什么也不做，直接break；它的目的是让ServiceManager知道，此次addService的反馈已经顺利完成。于是，ServiceManager继续它的循环；当它再次调用ioctl()，进而进入到Binder驱动中读取请求时；由于此时的待处理事务队列为空，再次进入wait\_event\_freezable\_exclusive，因此，ServiceManager会再次进入中断等待状态，等待Client的请求。

6、binder\_link\_to\_death

注意到在addservcie时还有一个**binder\_link\_to\_death**的调用，把binder\_death一起保存。发送BC\_REQUEST\_DEATH\_NOTIFICATION到binder驱动，发送的数据还有：添加Service的binder引用描述符handle以及si->death（这个结构体有两个元素，1、binder引用本身，2、函数指针svcinfo\_death），因为ServiceManager是专门管理service的守护进程，所以需要知道这个service的生死，这里link了一个死亡通知函数svcinfo\_death到service的binder引用上，这个binder引用就是ServiceManager进程中保存的被添加service的binder实体的引用，当service挂掉以后，驱动会根据这个引用的death函数通知到ServiceManager。

具体binder驱动的工作可以参考《**3-通过MediaPlayerService分析binder机制-addService**》，里面有一节详细介绍死亡通知在binder驱动中的处理。

这里这个死亡通知函数svcinfo\_death的功能：

在一个service挂掉之后，binder驱动走binder release，通知该service对应的所有binder引用（注册过death函数的所有引用）所在的进程处理死亡通知消息BR\_DEAD\_BINDER。

binder\_parse：case BR\_DEAD\_BINDER，直接调用了death->func，也就是svcinfo\_death，里面执行binder\_release，发送BC\_RELEASE到binder驱动，只是执行了binder\_dec\_ref，把这个引用删除，这样ServiceManager进程就没有死亡service的binder引用了。再来看看ServiceManager中保存的svcinfo链表，重置svcinfo里的handle为0，0不就是ServiceManager自己吗？所以如果一个service挂掉之后，再从ServiceManager中去getService返回ServiceManager自己的代理BpBinder（0）。一个service曾经crash过，再次注册时，会重新覆盖svclist中的同名service节点对象svcinfo，其实只是重新赋值了一个新的binder应用描述符mHandle，再次getService就可以得到正确的service了。

总结：正常情况下两个service是不能同名的。前一个注册的会被后一个覆盖，覆盖前先发送BC\_RELEASE到驱动，删除之前保存的binder引用。

此时ServiceManager中断等待、MediaPlayerService进程被唤醒，MediaPlayerService被唤醒后的操作参考**《3-通过MediaPlayerService分析binder机制-addService》**。

**2.4.3、getService时code流程：**

驱动中被唤醒后的流程和上一节addService时一模一样，最终填写两个cmd：BR\_NOOP、BR\_TRANSACTION

其它流程就不分析了，和addService时都一样，差异点在binder\_parse中走到svcmgr\_handler的case SVC\_MGR\_CHECK\_SERVICE：

case SVC\_MGR\_GET\_SERVICE:case SVC\_MGR\_CHECK\_SERVICE:，这里获取到name后直接用do\_find\_service根据name寻找service。

1、s = bio\_get\_string16(msg, &len)，得到service的name，这里其实是"media.player"。

2、do\_find\_service()调用find\_svc()进行查找。在find\_svc()中，从svclist链表中查找是否有名称等于"media.player"的svcinfo对象，找到并返回，没找到返回NULL。

3、do\_find\_service()返回找到的si->handle。这就是MediaPlayerService在Binder驱动中的Binder引用的描述符了。

4、svcmgr\_handler()会调用bio\_put\_ref()将该引用信息写入到flat\_binder\_object结构中，看看填写了什么：

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

obj->type = BINDER\_TYPE\_HANDLE;

obj->handle = handle;

obj->cookie = 0;

返回到binder\_parse，binder\_send\_reply，这里写了两个cmd：BC\_FREE\_BUFFER、BC\_REPLY，之后调用binder\_write调用ioctl（BINDER\_WRITE\_READ）陷入内核态，回到binder驱动。

5、BC\_FREE\_BUFFER不介绍了，释放内存的。BC\_REPLY：回到驱动的binder\_transaction()中，reply是true。

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

target\_thread = in\_reply\_to->from;

target\_proc = target\_thread->proc;

e->to\_thread = target\_thread->pid;

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

target\_wait = &target\_thread->wait;//拿到目标线程信息（proc、todo等）。

创建事务t和tcomplete。把reply的数据填写在t中相应的结构体指针里。进入for循环得到flat\_binder\_object结构体指针fp，和addService流程一样。只是这里有一个fp数据，type是BINDER\_TYPE\_HANDLE。

进入case BINDER\_TYPE\_HANDLE，通过binder\_get\_ref在当前进程的refs\_by\_desc红黑树中获取MediaPlayerService的binder引用，if (ref->node->proc == target\_proc) 这里不等，因为这里这个MediaPlayerService的ref->node实体指向的proc是MediaPlayerService进程，而target\_proc是MediaPlayer进程。

所以有：

new\_ref = binder\_get\_ref\_for\_node(target\_proc, ref->node);在MediaPlayer进程中查找MediaPlayerService node实体对应的binder引用，找不到会新建一个，同时加入到MediaPlayer进程的proc->refs\_by\_node和proc->refs\_by\_desc

fp->handle = new\_ref->desc;//把新建的binder引用描述符赋值到fp中。

t->work.type = BINDER\_WORK\_TRANSACTION;

tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;

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

if (target\_wait) wake\_up\_interruptible(target\_wait);

把目标进程MediaPlayer唤醒，它的todo中有一个事务t。

**注意这里是根据ref->node (MediaPlayerService的binder引用中保存的binder实体) 新建了一个MediaPlayerService的binder引用，并放进MediaPlayer进程proc保存。**

6、binder驱动开始返回指令和数据了，同样是两个事务。

分配一个待处理事务t给目标进程——Mediaplayer，唤醒它，binder驱动返回reply数据，回到《4-通过MediaPlayerService分析binder机制-getService》

分配一个待完成工作tcomplete到当前进程——ServiceManager，告诉它reply已经处理，ServiceManager进程处理完该事务后，再次进入中断等待状态，等待在binder驱动的wait\_event\_freezable\_exclusive处。

**2.5、总结：**

1、对于ServiceManager进程而言它打开了Binder设备文件，创建了自己的binder\_proc对象，并且将一块物理内存同时映射到ServiceManager的进程空间和binder内核空间。然后，它告诉Binder驱动自己是Binder上下文的管理者，新建并把自己的binder node实体设成全局变量。最后，进入消息循环，等待Client请求。

2、对于Binder驱动而言它初始化了ServiceManager对应的进程上下文环境(即binder\_proc对象的创建)，并将内核虚拟地址和进程虚拟地址映射到同一物理内存中。然后，新建当前线程对应的binder\_thread对象，并将其添加到binder\_proc->threads红黑树中。在得知ServiceManager是Binder上下文管理者后，建立ServiceManager对应的Binder实体，并将该Binder实体保存到全局变量中，同时保存到binder\_proc对象的proc->nodes红黑树中。最后在ServiceManager进入消息循环后，由于当前线程中没有事务可处理，则进入wait\_event\_freezable\_exclusive中断等待状态，等待其他进程将其唤醒。

3、addService时，根据Service的binder node实体在ServiceManager进程的refs\_by\_node红黑树中查找binder引用，第一次会新建一个binder引用，并添加到ServiceManager进程的refs\_by\_node和refs\_by\_desc中，这个引用属于ServiceManager进程。在ServiceManager用户空间，新建一个svcinfo节点保存service的name和binder引用描述符。

4、getService时，根据service的name在ServiceManager中获取binder引用描述符，进而找到service的binder node实体。根据binder实体在目标进程（调用getService的进程）的refs\_by\_node中查找对应的binder引用，第一次则新建一个binder引用，并添加到目标进程的refs\_by\_node和refs\_by\_desc中，这个引用属于该目标进程。驱动返回给目标进程的数据reply里就是这个引用的描述符。目标进程用户空间根据这个描述符创建BpBinder。

**3、延伸**

没看到关于binder\_ref：binder实体的引用在哪里创建？参考**《3-通过MediaPlayerService分析binder机制-addService》，可知**ServiceManager在内核binder驱动并没有binder\_node实体的引用binder\_ref对象，是根据0号handle直接获取全局变量binder\_node \*binder\_context\_mgr\_node来得到ServiceManager的binder实体。其它service的binder\_node和binder\_ref在addService时由binder驱动新建，并把binder\_node和service的对象指针关联起来，把binder\_ref放在ServiceManager的红黑树中。getService时新建一个binder ref放在调用getService的进程红黑树中。

**其实本文讲的是BnServiceManager本身及工作原理，与binder机制息息相关。**