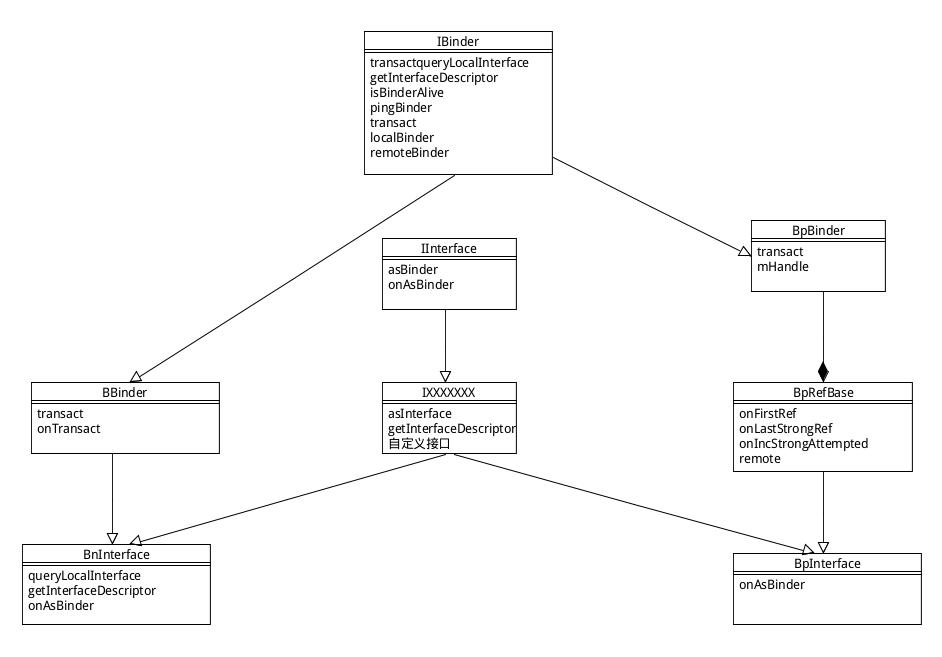
**Binder\_node**

**1.binder node的创建的过程：**

**Bn对象的结构：**



继承BBinder和派生于IInterface类子类，派生于IInterface类子类主要是产生一个对象通信时接口的虚函数表。

**BnInterface对象的创建：**

1. **New出继承BnXXX的服务对象。**

2.**创建ServiceManager的BpInterface:**

使用0作为handle的值来创建BpBinder(handle)。

使用创建的BpBinder对象来创建BpServiceManager对象。

3.**在BpServiceManager的addService函数**中通过Parcel对象来序列化创建的BnXXX服务对 象作为addService的参数，BnXXX服务对象被保存在flat\_binder\_object对象中。

方法：data.writeStrongBinder(service);

struct flat\_binder\_object {

\_\_u32 type; //flat\_binder\_object的类型：BINDER\_TYPE\_BINDER、BINDER\_TYPE\_HANDLE、BINDER\_TYPE\_FD

\_\_u32 flags;

union {

binder\_uintptr\_t binder;

\_\_u32 handle;

};

binder\_uintptr\_t cookie;//BnXXX服务对象的地址

};

4.**通过调用BpServiceManager**成员BpBinder的transact函数，在该函数中使用handle为0的参数来调用IPCThreadState::transact函数，在该函数中用传进来的参数封装binder\_transaction\_data数据结构，该数据结构主要是为驱动层处理BC\_TRANSACTION命令时的参数。

binder\_transaction\_data数据结构：

struct binder\_transaction\_data {

union {

\_\_u32 handle; //从servicemanager中获取的Bpbinder对象中handle值

binder\_uintptr\_t ptr;

} target;

binder\_uintptr\_t cookie; //Bn对象的地址

\_\_u32 code; //sevice端bn对象接口ID号

\_\_u32 flags;

pid\_t sender\_pid;

uid\_t sender\_euid;

binder\_size\_t data\_size;//BC\_TRANSACTION时从client端要传送到service端的参数大小

binder\_size\_t offsets\_size;//BC\_TRANSACTION时从client端要传送到service端的参数偏移

union {

struct {

binder\_uintptr\_t buffer;

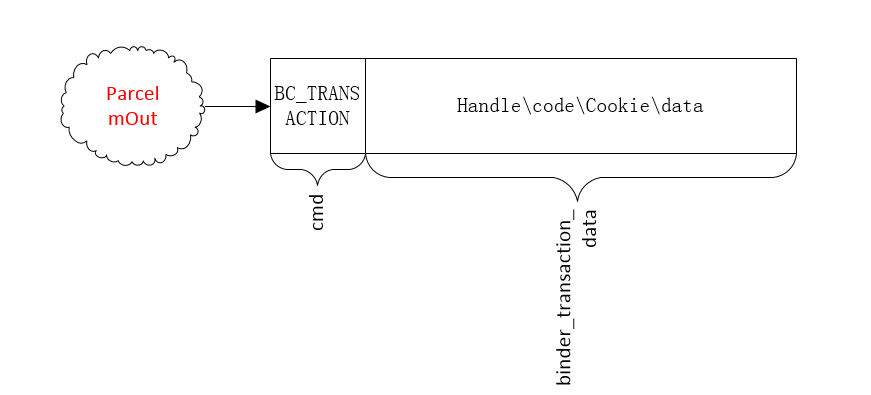
binder\_uintptr\_t offsets;

} ptr;//BC\_TRANSACTION时从client端要传送到service端的参数可能是个binder、handle、fd或者其他数据类型。

\_\_u8 buf[8];

} data;

};



1. **调用talkWithDriver函数**封装binder\_write\_read数据结构，该数据结构主要的功能是为驱动层的的binder\_ioctl函数中的BINDER\_WRITE\_READ处理准备参数。进入binder\_thread\_write函数。

struct binder\_write\_read {

//binder\_thread\_write函数使用的参数

binder\_size\_t write\_size; //Parcel mOut中的数据的大小

binder\_size\_t write\_consumed; //binder\_thread\_write函数已经读取数据的大小

binder\_uintptr\_t write\_buffer;//保存Parcel mOut中的数据

//binder\_thread\_read函数使用的参数

binder\_size\_t read\_size;//Parcel mIn中的数据的大小

binder\_size\_t read\_consumed;//binder\_thread\_read函数已经读取数据的大小

binder\_uintptr\_t read\_buffer;//保存Parcel mIn中的数据

};

1. binder\_thread\_write函数通过解析binder\_write\_read中的write\_buffer中的binder\_transaction\_data数据结构中的BC\_TRANSACTION命令来调用binder\_transaction函数。
2. **binder\_transaction函数解析：**

(1)tr->target.handle获取target\_node

在这里由于handle= 0，target\_node = binder\_context\_mgr\_node。

(2)通过target\_node获取target\_proc (为servicemanager的binder\_proc)

(3)通过target\_proc来获取target\_list和target\_wait

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

target\_wait = &target\_proc->wait;

(4)创建一个binder\_transaction对象。

struct binder\_transaction {

int debug\_id;

struct binder\_work work;

struct binder\_thread \*from;

struct binder\_transaction \*from\_parent;

struct binder\_proc \*to\_proc;

struct binder\_thread \*to\_thread;

struct binder\_transaction \*to\_parent;

unsigned need\_reply:1;

struct binder\_buffer \*buffer;

unsigned int code;

unsigned int flags;

long priority;

long saved\_priority;

kuid\_t sender\_euid;

};

1. .解析binder\_transaction\_data中的参数data中的数据，参数中flat\_binder\_object对象是BINDER\_TYPE\_BINDER的类型，通过fp->binder(Bn对象的弱引用), fp->cookie(Bn对象地址)为Bn服务对象在驱动层创建相应的binder\_node保存在binder\_proc中。

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;//binder\_node对应的进程

struct hlist\_head refs;

int internal\_strong\_refs;

int local\_weak\_refs;

int local\_strong\_refs;

binder\_uintptr\_t ptr;//bnInterface对象的弱引用

binder\_uintptr\_t cookie;//bnInterface对象的地址

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;

};

1. .通过binder\_get\_ref\_for\_node(target\_proc, node)函数在target\_proc进程中创建node binder\_ref，这里的target\_proc为serviceManager。 遍历binder\_proc中的refs\_by\_desc红黑树，将唯一的desc值赋给binder\_ref的desc中。将binder\_ref保存在serviceManager的binder\_proc中的refs\_by\_node红黑树中。

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; //保存binder\_ref的进程binder\_proc

struct binder\_node \*node;//对应的binder\_node

uint32\_t desc; //binder\_ref 中的handle从0递增，在servicemanager中其它进程的binder\_node 的binder\_ref handle的值是唯一的。

int strong;

int weak;

struct binder\_ref\_death \*death;

};

加入client端进程获取了一个service端进程的Bp对象，然后在client调用service端的接口，在该接口中传入一个client端创建的Bn对象，那么service端通过参数传过来的Bp对象也可以和client端的Bn对象进行通信。

也就是说一个进程只要通过binder通信的时候参数是个Bn服务对象，那么都会在本进程中创建binder\_node和在要进行binder通信的进程中创建binder\_ref. binder\_ref可以在servicemanager创建也可以在其它进程中创建。

(7).将fp->handle = ref->desc，同时将binder\_transaction加入到target\_list，唤醒在target\_wait上等待的线程。

1. **binder\_thread\_read函数的解析：**

Service\_manager主线程作为binder线程循环调用binder\_ioctl函数，并且阻塞在

ret = wait\_event\_freezable\_exclusive(proc->wait, binder\_has\_proc\_work(proc, thread));

函数上。

当等待队列proc->wait 被唤醒，从proc->todo中获取binder\_transaction对象，将BR\_TRANSACTION命令和binder\_transaction拷贝到参数binder\_write\_read中的read\_buffer中，返回到用户空间。

1. **servicemanager 的binder\_parse函数**中解析BR\_TRANSACTION命令，获取binder\_transaction的值，调用svcmgr\_handler函数进行处理SVC\_MGR\_ADD\_SERVICE 命令，创建svcinfo对象保存在svcinfo \*svclist链表中。

struct svcinfo

{

struct svcinfo \*next;

uint32\_t handle; //servicemanager中binder\_ref的desc

struct binder\_death death;

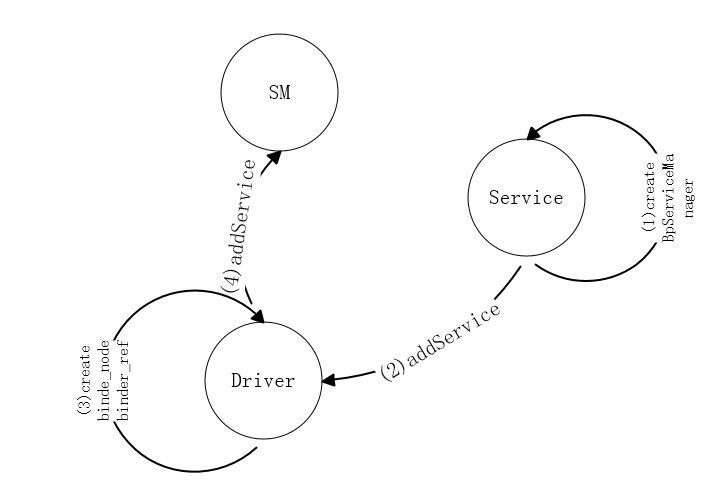
int allow\_isolated;

size\_t len;

uint16\_t name[0];//add service时InterfaceDescriptor

};

**addService函数作用的总结：**



在驱动层创建binder\_node、binder\_ref、binder\_ref的desc，将binder\_ref的desc和interface decription保存在serviceManager层。