内容目录

[1 Binder通信机制架构 1](#__RefHeading___Toc211_123530771)

[2 Binder驱动架构 3](#__RefHeading___Toc213_123530771)

[3 ServiceManager的作用 5](#__RefHeading___Toc215_123530771)

[4 从源码分析ServiceManger向Binder注册的过程。 6](#__RefHeading___Toc217_123530771)

[4.1 serviceManger运行时序图 6](#__RefHeading___Toc219_123530771)

[5 client获取ServiceManager的过程 9](#__RefHeading___Toc221_123530771)

[6 client通过ServiceManager的获取服务的过程 12](#__RefHeading___Toc223_123530771)

[7 具体服务运行流程以MediaPlayerService为例 13](#__RefHeading___Toc225_123530771)

[8 Binder消息发送流程 14](#__RefHeading___Toc227_123530771)

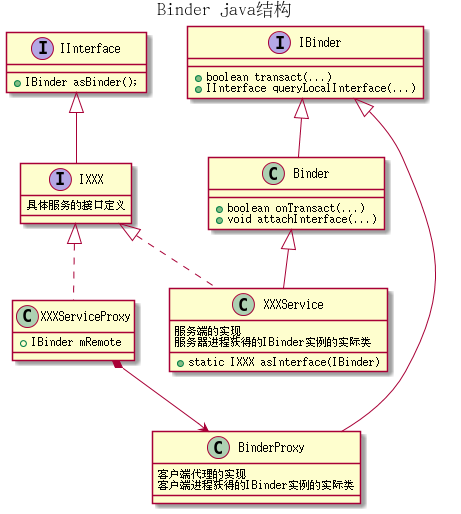
**1 Binder通信机制架构**

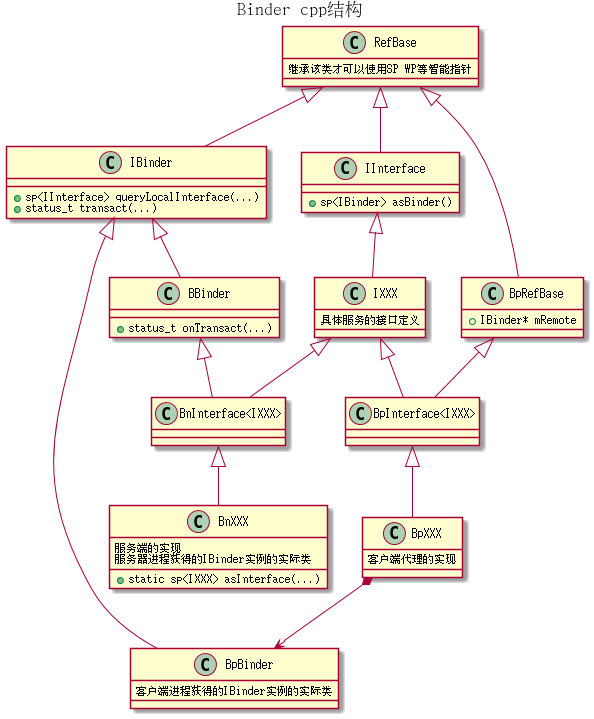
Binder机制是android的IPC通信机制

在java层以及cpp层都有具体的实现

cpp层类图

java层类图





两个地方的实现方式都是一致的。

1 服务端要实现onTransact(...)函数，这是具体的服务接口的实现的地方

客户端的所有接口的实现方式都是通过transact(...)函数发送请求给Binder驱动，然后Binder驱动会将请求交予服务端进程的onTransact(...)进行具体实现。

因此代理XXXServiceProxy具体完成的是接口参数的打包，BinderProxy完成的是请求的发送

2 对于服务端进程，使用的IBinder对象是XXXService的实例

而客户端则获取到的IBinder对象是BinderProxy的实例

例如通过ServiceManager.getService()获取到的都是 BinderProxy的实例

通过startService方式启动的自定义服务获取到的也都是BinderProxy的实例

3 服务端的实现XXXService需要注意的是

接口IXXX asInterface(IBinder obj)的实现要考虑到再服务进程和客户端进程调用的时候，返回的对象是不同的，在服务端进程直接返回XXXService实例就号了，而客户端进程则使用代理XXXServiceProxy实例对象。具体的方式为，先调用obj.queryLocalInterface()获取本地实例，如果有，说明是服务端进程，直接返回，没有则生成代理对象返回

Binder{

IInterface queryLocalInterface(...){ … return this;}//类似这样的实现

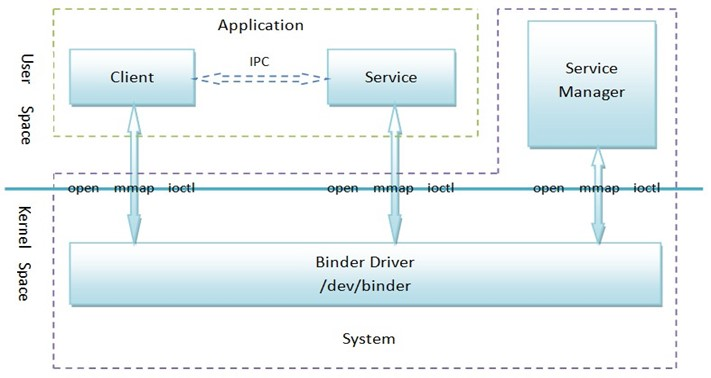
}

BinderProxy{

IInterface queryLocalInterface(...){return null;}//类似这样的实现

}

**2 Binder驱动架构**

****

Binder是一种高效且易用的IPC机制，提供远程过程调用（RPC）功能：

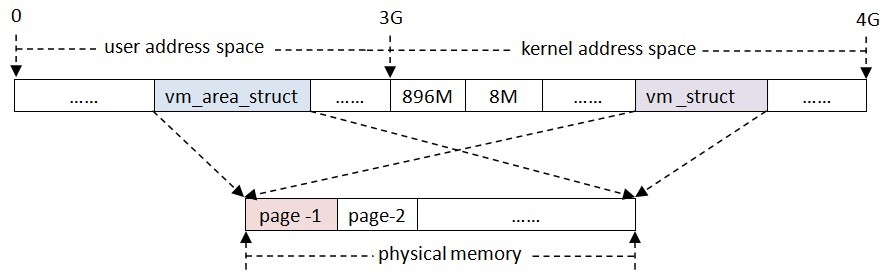
* 一次数据拷贝
* Client/Server通信模型
* 既可用作进程间通信，也可用作进程内通信

        在Android系统的Binder机制中，由一系统组件组成，分别是Client、Server、Service Manager和Binder驱动程序（**其中Client、Server和Service Manager运行在用户空间，Binder驱动程序运行内核空间**）。Binder就是一种把这四个组件粘合在一起的粘结剂了，其中，核心组件便是Binder驱动程序了，**Service Manager提供了辅助管理的功能，Client和Server正是在Binder驱动和Service Manager提供的基础设施上，进行Client-Server之间的通信**。

Client、Server和Service Manager实现在用户空间中，Binder驱动程序实现在内核空间中

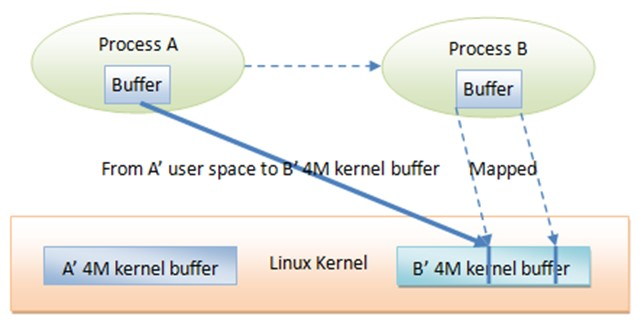
1. Binder驱动程序和Service Manager在Android平台中已经实现，开发者只需要在用户空间实现自己的Client和Server
2. **Binder驱动程序提供设备文件/dev/binder与用户空间交互，Client、Server和Service Manager通过open和ioctl文件操作函数与Binder驱动程序进行通信**
3. Client和Server之间的进程间通信通过Binder驱动程序间接实现
4. **Service Manager是一个守护进程，用来管理Server，并向Client提供查询Server接口的能力**

**对用client和server端来说，Binder就是一个文件，使用open函数打开就可以使用了，为了方便数据拷贝，需要使用mmap函数进行内存映射，大概的作用就是，把内核的一段缓存与进程的一段缓存对应起来，两个地方用的同一段内存空间，所以客户端向服务端发送数据的时候，只需要拷贝一次。ioctl是对驱动进行设置的函数，可以设置驱动的线程数等等参数，还有向驱动发送数据也是使用这个函数**

Binder驱动为**每一个进程分配4M的内核缓冲区**（物理页面，可以是不连续），用作数据传输：

* 4M内核缓冲区所对应的物理页面除了映射在内核空间之外，还会被映射在进程的用户空间（同时使用进程虚拟地址空间和内核虚拟地址空间来映射同一个物理页面）

**也就是说，每个进程在使用binder的时候都会使用mmap进行内存映射。然后其他进程与之通信，向其发送数据的时候，发送到这段内存就好了**

****

**3 ServiceManager的作用**

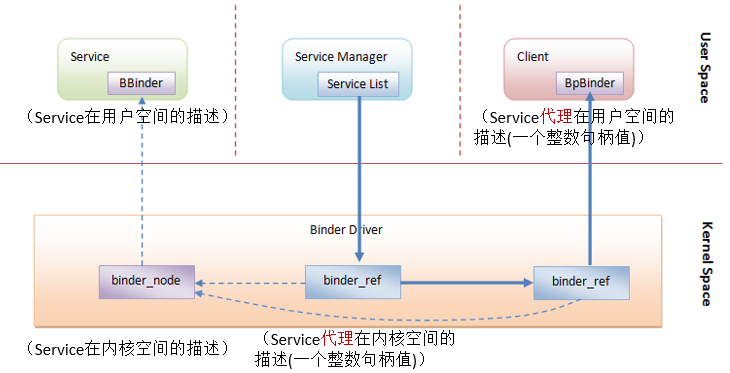
ServiceManager也是一个服务，不过这个服务是用来管理其他服务的，相当于服务的服务。

1 ServiceManger会在android开机的时候启动，ServiceManager会通知Binder将自己注册。

2 Server进程启动时，将在本进程内运行的Service注册到Service Manager中，并且启动一个Binder线程池，用来接收Client进程请求。

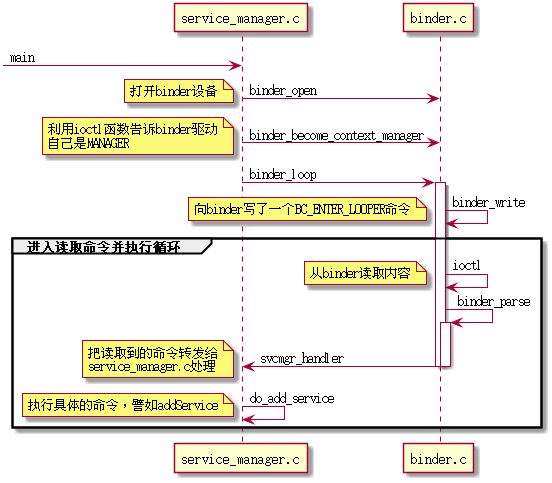
3 Client想要获取服务都是通过ServiceManger来得到。

**那么一个Client如何获得ServiceManager这个服务呢。答案就是，每个服务都在Binder驱动里面对应一个句柄（可以简单看做是数组序号），而ServiceManager就是句柄为0的那个。Client只要通过Binder向句柄为0的那个服务发送请求ServiceManger就能收到。同样的，其他服务也会对应一个句柄，不过具体是那个句柄只要向ServiceManager去问就好了**



**4 从源码分析ServiceManger向Binder注册的过程。**

**4.1 serviceManger运行时序图**

1 首先要说明的是，ServiceManager的客户端代理类确实是BpServiceManager但是，其服务端却不是BnServiceManager实现，而是在/frameworks/native/cmds/servicemanager/service\_manager.c实现的

int main(int argc, char \*\*argv)

{

struct binder\_state \*bs;

void \*svcmgr = BINDER\_SERVICE\_MANAGER;

bs = **binder\_open(128\*1024);//打开binder驱动**

if (**binder\_become\_context\_manager(bs))** {**//注册自身成为ServiceManger**

ALOGE("cannot become context manager (%s)\n", strerror(errno));

return -1;

}

svcmgr\_handle = svcmgr;

**binder\_loop(bs, svcmgr\_handler);//进入消息循环**

return 0;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////

struct binder\_state \*binder\_open(unsigned mapsize)

{

struct binder\_state \*bs;

bs = malloc(sizeof(\*bs));

if (!bs) {

errno = ENOMEM;

return 0;

}

bs->fd = **open("/dev/binder", O\_RDWR);//打开binder**

if (bs->fd < 0) {

fprintf(stderr,"binder: cannot open device (%s)\n",

strerror(errno));

goto fail\_open;

}

bs->mapsize = mapsize;

bs->mapped = **mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0);**

**//映射内存**

………//省略

return 0;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////

int binder\_become\_context\_manager(struct binder\_state \*bs)

{

return **ioctl(bs->fd, BINDER\_SET\_CONTEXT\_MGR, 0);//给Binder发了一个注册命令过去**

}

////////////////////////////////////////////////////////////////////////////////////////////////////////

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);//读取请求**

if (res < 0) {

ALOGE("binder\_loop: ioctl failed (%s)\n", strerror(errno));

break;

}

res = **binder\_parse(bs, 0, readbuf, bwr.read\_consumed, func)**;**//解析请求，这个函数就不做说明了，具体流程就是，先把数据解析，然后把命令分发给service\_manager.c的svcmgr\_handler函数进行具体的命令处理**

if (res == 0) {

ALOGE("binder\_loop: unexpected reply?!\n");

break;

}

if (res < 0) {

ALOGE("binder\_loop: io error %d %s\n", res, strerror(errno));

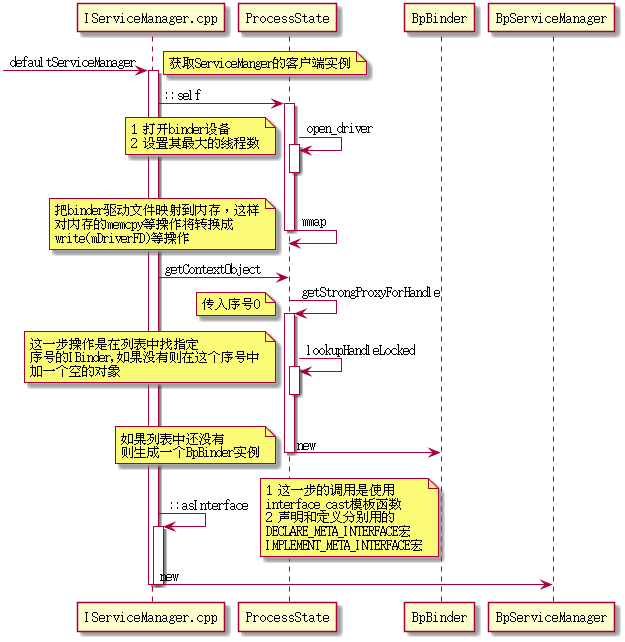
break;

}

}

}

**5 client获取ServiceManager的过程**



1 所有的client端获取ServiceManager实例都是调用IServiceManager.cpp的defaultServiceManager函数得到。

2 ProcessState管理着与Binder驱动的交互。整个进程只有一个，因此是单例的，调用其self函数获得其实例，在其构造函数中，ProcessState首先调用open\_driver函数打开Binder驱动，然后进程mmap内存映射

3 调用ProcessState的getContextObject(NULL)函数获得ServiceManager的BpBinder实例，由“**Binder通信机制架构**”这一节我们知道，这个BpBinder主要的功能是进行 参数的传输，也就是通信层。

4 使用interface\_cast宏生成BpServiceManger对象，这个就是客户端需要的ServiceManger的代理对象。

注意，这个interface\_cast宏展开其实就是

template<typename INTERFACE>

inline sp<INTERFACE> interface\_cast(const sp<IBinder>& obj)

{

return INTERFACE::asInterface(obj);

}

展开就是

sp<IServiceManager> interface\_cast(const sp<IBinder>& obj)

{

return IServiceManager::asInterface(obj);

}

也就是调用而这个函数的声明使用的宏是

DECLARE\_META\_INTERFACE

定义用的宏是

IMPLEMENT\_META\_INTERFACE

具体的代码就不解释了，这个asInterface函数就是吧BpBinder作为参数生成了一个BpServiceManager的实例

////////////////////////////////////////////////////////////////////////////////////////////////////////

sp<IServiceManager> defaultServiceManager()

{

if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

{

AutoMutex \_l(gDefaultServiceManagerLock);

while (gDefaultServiceManager == NULL) {

gDefaultServiceManager = **interface\_cast<IServiceManager>(**

**ProcessState::self()→getContextObject(NULL));//获得ServiceManager的BpBinder实例，然后使用IServiceManager::asInterface函数获得BpServiceManager实例**

if (gDefaultServiceManager == NULL)

sleep(1);

}

}

return gDefaultServiceManager;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////

ProcessState::ProcessState()

: mDriverFD(**open\_driver()**)**//注意这里打开了Binder驱动**

, mVMStart(MAP\_FAILED)

, mManagesContexts(false)

, mBinderContextCheckFunc(NULL)

, mBinderContextUserData(NULL)

, mThreadPoolStarted(false)

, mThreadPoolSeq(1)

{

if (mDriverFD >= 0) {

// XXX Ideally, there should be a specific define for whether we

// have mmap (or whether we could possibly have the kernel module

// availabla).

#if !defined(HAVE\_WIN32\_IPC)

// mmap the binder, providing a chunk of virtual address space to receive transactions.

**mVMStart = mmap(0, BINDER\_VM\_SIZE, PROT\_READ, MAP\_PRIVATE |** **MAP\_NORESERVE, mDriverFD, 0);//进行内存映射**

if (mVMStart == MAP\_FAILED) {

// \*sigh\*

ALOGE("Using /dev/binder failed: unable to mmap transaction memory.\n");

close(mDriverFD);

mDriverFD = -1;

}

#else

mDriverFD = -1;

#endif

}

LOG\_ALWAYS\_FATAL\_IF(mDriverFD < 0, "Binder driver could not be opened. Terminating.");

}

////////////////////////////////////////////////////////////////////////////////////////////////////////

static int open\_driver()

{

**int fd = open("/dev/binder", O\_RDWR);//打开Binder驱动**

if (fd >= 0) {

fcntl(fd, F\_SETFD, FD\_CLOEXEC);//close fd at execl() a new process, but not close at fork() sub process

int vers;

status\_t result = ioctl(fd, BINDER\_VERSION, &vers);

if (result == -1) {

ALOGE("Binder ioctl to obtain version failed: %s", strerror(errno));

close(fd);

fd = -1;

}

if (result != 0 || vers != BINDER\_CURRENT\_PROTOCOL\_VERSION) {

ALOGE("Binder driver protocol does not match user space protocol!");

close(fd);

fd = -1;

}

size\_t maxThreads = 15;

result = **ioctl(fd, BINDER\_SET\_MAX\_THREADS, &maxThreads);//设置Binder驱动最大线程数量**

if (result == -1) {

ALOGE("Binder ioctl to set max threads failed: %s", strerror(errno));

}

} else {

ALOGW("Opening '/dev/binder' failed: %s\n", strerror(errno));

}

return fd;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////

sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)

{

return **getStrongProxyForHandle(0);//0是ServiceManager在Binder中的句柄号**

}

////////////////////////////////////////////////////////////////////////////////////////////////////////

sp<IBinder> ProcessState::getStrongProxyForHandle(int32\_t handle)

{

sp<IBinder> result;

AutoMutex \_l(mLock);

**handle\_entry\* e = lookupHandleLocked(handle);//这里先到缓存列表里面找，没找到就先加一个空的进入列表**

if (e != NULL) {

IBinder\* b = e->binder;

if (b == NULL || !e->refs->attemptIncWeak(this)) {

if (handle == 0) {

Parcel data;

status\_t status = **IPCThreadState::self()->transact(**

**0, IBinder::PING\_TRANSACTION, data, NULL, 0);//看看该服务在不在运行**

if (status == DEAD\_OBJECT)

return NULL;

}

b = **new BpBinder(handle); //使用句柄生成 BpBinder对象返回**

e->binder = b;

if (b) e->refs = b->getWeakRefs();

result = b;

} else {

result.force\_set(b);

e->refs->decWeak(this);

}

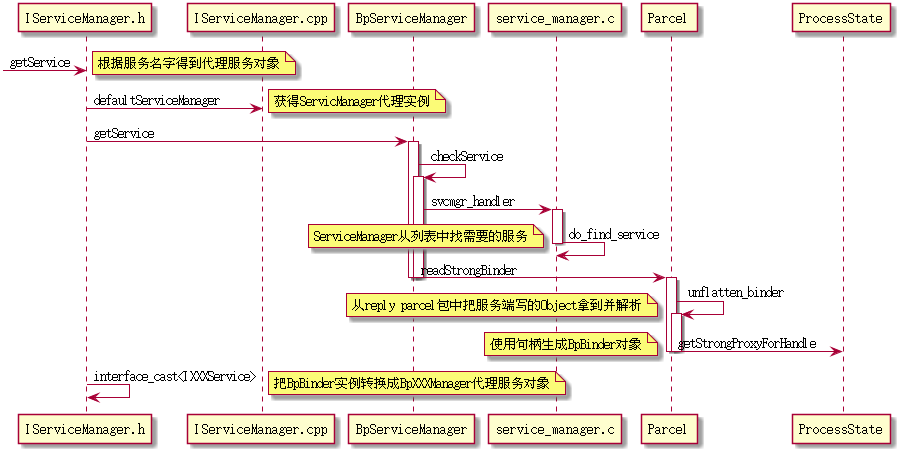
}

return result;

}

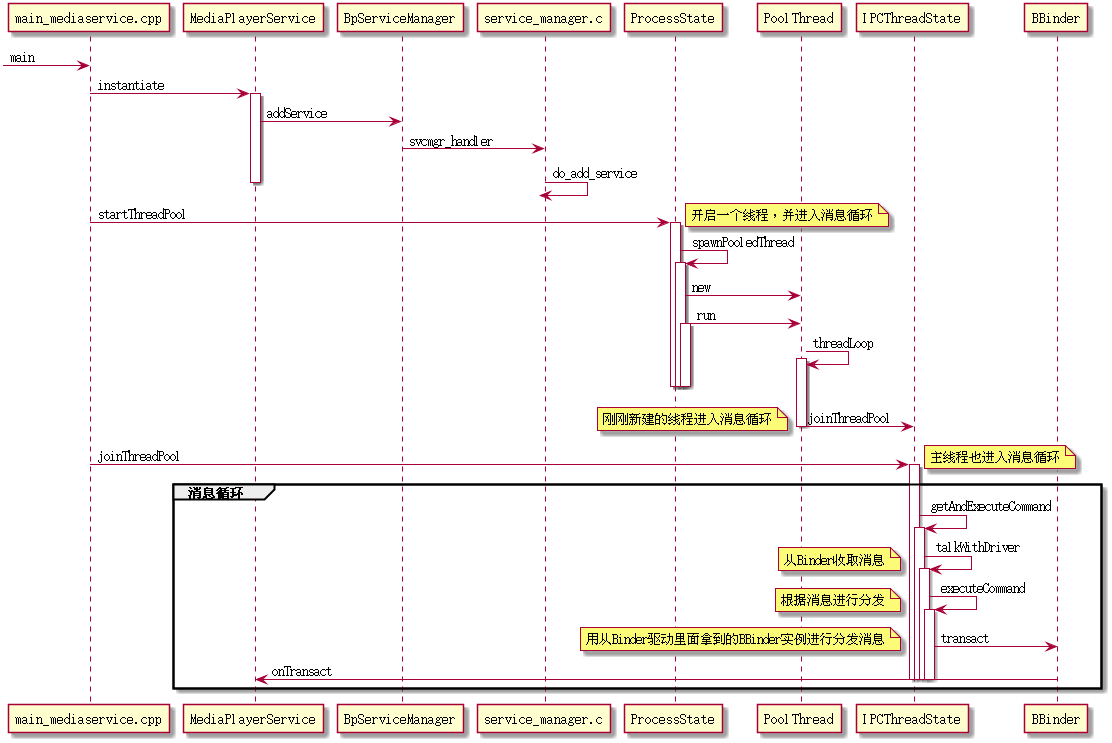
////////////////////////////////////////////////////////////////////////////////////////////////////////

**6 client通过ServiceManager的获取服务的过程**

1 客户端要得到某个Service代理对象，只要调用IServiceManager.h里面的getServcie函数，这个函数是一个模板函数，他会返回一个BpXXXServiceManager对象

2 首先getServcie会获取ServiceManager代理服务实例，然后向服务端发送checkService请求，服务端如果找到要求的服务就会返回对应句柄。然后客户端通过Parcel的readStrongBinder函数将句柄转换成BpBinder对象。

**7 具体服务运行流程以MediaPlayerService为例**

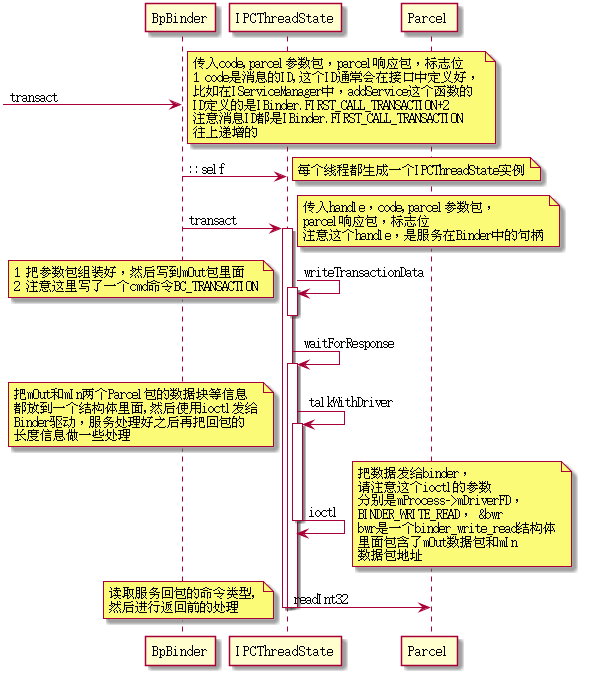
**主要分为三步：**

**1 将实现服务的BnXXXService对象通过addService函数添加到ServiceManager;**

**2 调用ProcessState的startThreadPool函数，新建一个线程，然后进入消息循环**

**3 调用IPCThreadState的joinThreadPool函数，主线程也进入消息循环**

**8 Binder消息发送流程**

****

1 IPCThreadState负责具体的与Binder驱动数据交互的部分，他有一个mOut parcel包，一个mIn parcel包，发送的时候IPCThreadState会吧这两个包的信息，以及他们的数据部分放到binder\_write\_read结构体里面，然后发送到Binder,这样从服务端那里访问到的内存就是属于mOut和mIn的。（归功于mmap内存映射）

2 writeTransactionData负责吧传入的参数包data 放到mOut里面，在使用ioctl通讯结束，如果服务返回的命令类型是BR\_REPLAY，那么，就会吧mIn包里面的数据放到replay 包里面去。