**Binder机制（native）**

**文档修改记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **版号** | **修改日期** | **修改内容** | **修改人** |
| 01 | 2017-04-21 | Binder机制（native）文档编写 | 刘祚亮 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

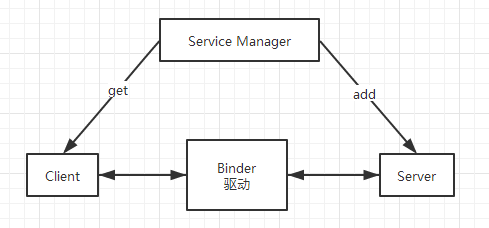
一、Binder的优势

Binder相比于传统的Linux进程间通信机制而言，简化了系统的复杂性，提高了通信的传输性能，传输过程中只需一次拷贝，为发送方添加UID/PID身份，既支持实名Binder也支持匿名Binder(实名binder就是在service manager中注册过的，匿名binder就是没有向servicemanager提交注册的binder，但匿名Binder必须是建立在一个实名Binder之上的，首先client和server通过实名binder建立联系，然后把匿名binder通过这个实名通道“传递过去”，对方也可以正确获取service的代理对象Bpxxx)，安全性高。

1. 通信模型

1. 基本模型

Binder框架定义了四个角色:Server、Client、ServiceManager以及Binder驱动。其中Server、Client、ServiceManager运行于用户空间，驱动运行于内核空间。如下图所示：



1. 通信协议

基本格式是：命令+数据，使用ioctl(fd,cmd,arg)函数实现交互。Fd是指文件描述符，cmd则是命令，arg指的是数据。

1. Binder的表述
2. Binder在应用程序中的表述
   1. Binder在Server端的表述-Binder实体

首先定义一个抽象接口类封装Server所有功能，其中包含一系列纯虚函数留待Server和Proxy各自实现。 由于这些函数需要跨进程调用，须为其一一编号，从而Server可以根据收到的编号决定调用哪个函数。其次就要引入Binder了。Server端定义另 一个Binder抽象类处理来自Client的Binder请求数据包，其中最重要的成员是虚函数onTransact()。该函数分析收到的数据包，调 用相应的接口函数处理请求。接下来采用继承方式以接口类和Binder抽象类为基类构建Binder在Server中的实体，实现基类里所有的虚函数，包括公共接口函数以及数 据包处理函数：onTransact()。

* 1. Binder在Client端的表述-Bidner引用

Client端的Binder同样要继承Server提供的公共接口类并实现公共函数。但这不是真正的实现，而是对远程函数调用的包装：将函数参数打包，通过Binder向Server发送申请并等待返回值。为此Client端的Binder还要知道Binder实体的相关信息，即对Binder实体的引用。该引用或是由SMgr转发过来的，对实名Binder的引用或是由另一个进程直接发送过来的，匿名 Binder的引用。

1. Binder在传输数据中的表述
   1. 文件形式的Binder

将文件看成Binder实体，进程打开的文件号看成Binder的引用。一个进程可以将它打开文件的文件号传递给另一个进程，从而另一个进程也打开了同一个文件，就象Binder 的引用在进程之间传递一样。

1. Binder在驱动中的表述
   1. Binder实体在驱动中的表述

每个进程都有一棵红黑树用于存放创建好的节点，以Binder在用户空间的指针作为索引。每当在传输数据中侦测到一个代表Binder实体的 flat\_binder\_object，先以该结构的binder指针为索引搜索红黑树；如果没找到就创建一个新节点添加到树中。由于对于同一个进程来说 内存地址是唯一的，所以不会重复建设造成混乱。

* 1. Binder引用在驱动中的表述

就像一个对象有很多指针一样，同一个Binder实体可能有很多引用，不同的是这些引用可能分布在不同的进程中。和实体一样，每个进程使用红黑树存放所有该进程正在使用的引用。驱动利用该红黑树在一个进程中快速查找某个Binder实体所对应的引用。

1. Binder内存映射和接收缓存管理

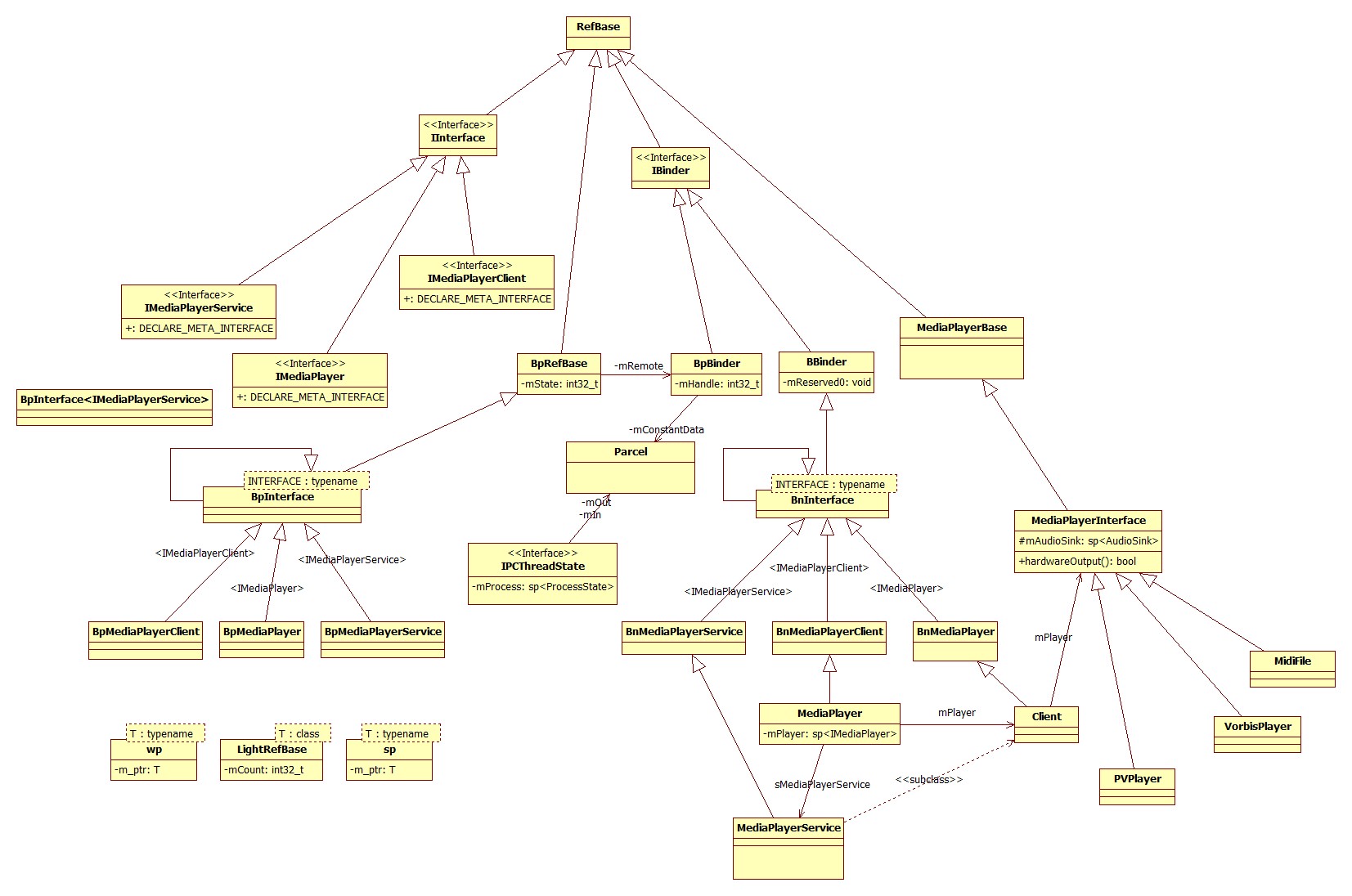
Binder采用一种全新策略：由Binder驱动负责管理数据接收缓存。Binder驱动当然不是为了在物理介质和用户空间做映射，而是用来创建数据接收的缓存空间。

fd = open(“dev/binder”,O\_RDWR);

mmap(NULL,MAP\_SIZE,PROT\_READ,MAP\_SIZE,fd,0);

为了实现用户空间到用户空间的拷贝，mmap()分配的内存,实质是共享内存，该内存数据除了映射进接收进程里，还映射进了内核空间。这就是Binder只需一次拷贝的秘密。

1. 继承关系图



1. 基本组成
2. Binder驱动

Binder驱动工作于内核态，提供open()、mmap()、poll()、ioctl()等标准文件操作，以字符驱动设备中的misc设备注册在设备目录/dev下，用户通过/dev/binder访问它。驱动负责进程之间Binder通信的建立，Binder在进程之间的传递，Binder引用计数管理，数据包在进程之间的传递与交互等一系列底层支持。

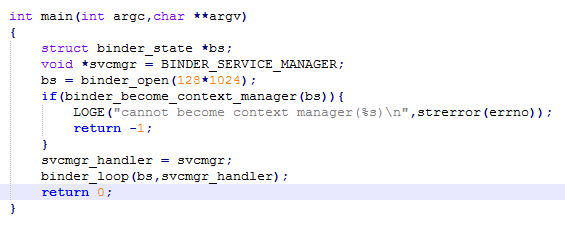
1. ServiceManager

ServiceManager是Android进程间通信（IPC）机制的一个Binder守护进程，Android系统中Service信息都是先add到ServiceManager中，由ServiceManager来集中管理，这样就可以查询当前系统有哪些服务。而且,Android系统中某个服务例如MediaPlayerService的客户端想要和MediaPlayerService通讯的话，必须先向ServiceManager查询MediaPlayerService的信息，然后通过ServiceManager返回的东西再来和MediaPlayerService交互。 概括为：

1. MediaPlayerService要向ServiceManager注册。

2. MediaPlayerClient查询当前注册在SM中的MediaPlayerService的信息。

3. 根据这个信息，MediaPlayerClient和MediaPlayerService进行交互。 ServiceManager的handle标示是0,所以只要往handle是 0 的服务发送消息了，最终都会被传递到ServiceManager中去。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。



大体过程是这样的：

1） 打开/dev/binder文件：open(“dev/binder”.O\_RDWR);

2） 建立128K内存映射：mmap(NULL,mapsize,PROT\_READ,

MAP\_PRIVATE,bs->fd,0);

3） 通知Binder驱动程序它是守护进程: binder\_become\_context

\_manager(bs);

4） 进入循环等待请求列表：binder\_loop(bs,svcmgr\_handler);

3. Server

获取到ServiceManager对象，然后通过addService函数将指定的Service注册到ServiceManager中，供Client获取Binder实体的引用。

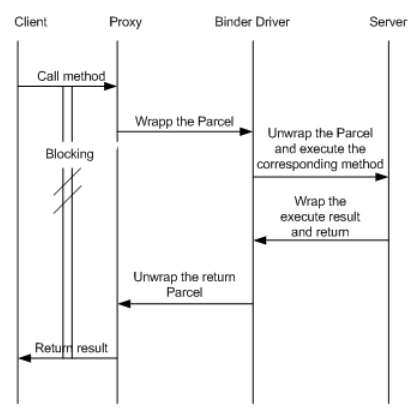
1. Client

获取到ServiceManager对象，然后通过getService函数获得服务端的IBinder实体引用之后，通过interface\_cast函数将IBinder实体引用转化成所需服务的对象，最后调用服务端的相应方法。

四、通信原理

Binder通信是一种Client-Server的通信结构：

从表面上来看，是Client通过获得一个Server的代理接口，对Server进行直接调用。实际上，代理接口中定义的方法与Server中定义的方法是一一对应的。Client调用某个代理接口中的方法时，代理接口的方法会将Client传递的参数打包成为Parcel对象，代理接口将该Parcel发送给内核中的binder driver。Server会读取binder driver中的请求数据，如果是发送给自己的，解析Parcel数据包对象，处理并将结果返回。整个调用过程是一个同步过程，在Server处理的时候，Client会block住。如图：



1. 基本对象

1. Binder

1） IBinder

IBinder是Android对Binder机制进行一个抽象，定义成一个IBinder接口，该接口是对跨进程对象的抽象，在C/C++和Java层都有定义。IBinder定义了一套使用Binder机制来实现客户程序与服务器的通信协议。

一个普通对象只能在当前进程中被访问，如果希望它能被其他进程访问，就必须实现IBinder接口。IBinder接口可以指向本地对象，也可以指向远程对象，关键就在于IBinder接口中的transact函数。如果IBinder指向的是一个服务端代理，那么transact只是负责把请求发送给服务器；如果IBinder指向的是一个服务端，那么transact只负责提供服务即可。因此，不管是服务端还是服务端代理对象，都必须实现该接口，这样才能进行Binder通信。

1. BpBinder

BpBinder是服务端代理对象，即远程对象在当前进程的代理。实际上，它也是Binder通信存在于客户端的进程，它实现了IBinder接口，它的transact函数的实现，在示例代码remote()->transact(DOSYSTEM, data, &reply);中，其中remote()函数返回的是一个BpBinder的一个对象引用，因此代码中真正的函数实现如下所示：

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;

}

它实际上只是简单地调用了IPCThreadState::self()的transact函数，将请求通过内核模块发送给了服务端，服务端处理完请求之后，沿原路返回结果给调用者。注意transact方法是同步方法，将会挂起客户进程的当前线程，直到Service把请求处理完成并返回结果。

1. BBinder

服务端同样需要实现IBinder接口，这里我们以Android默认的服务端实现类(BBinder)为例进行介绍，其中transact的实现：

status\_t BBinder::transact(uint32\_t code,const Parcel& data,Parcel reply,uint32\_t flags)

{

data.setDataPosition(0);

status\_t err = NO\_ERROR;

switch (code) {

case PING\_TRANSACTION:

reply->writeInt32(pingBinder());

break;

default:

err = onTransact(code, data, reply, flags);

break;

}

if (reply != NULL) {

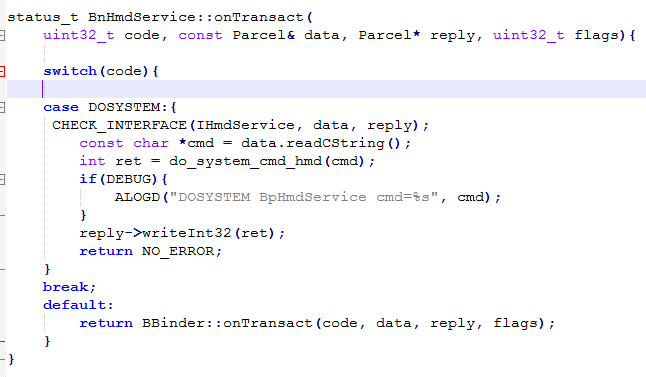
reply->setDataPosition(0);

}

return err;

}

//其中，PING\_TRANSACTION请求用来检查对象是否还存在，这里简单地把 pingBinder的返回值返回给调用者，其他的请求交给onTransact处理。onTransact是BBinder中声明的一个protected类型的虚函数，这个要求它的子类去实现。如图



1. IPCThreadState

每一个线程中都会有一个IPCThreadState对象，它主要负责Binder数据读取、写入和请求处理框架。

1. ProcessState

每个进程只有一个ProcessState对象，ProcessState是一个单例类型，其作用是维护当前进程中的所有Service代理。一个客户端进程可能需要多个Service的服务，这样可能会创建多个Service代理，客户端进程中的ProcessState对象就负责维护这些Service代理。

六、基本类和接口

1. IInterface

为server端提供接口，它的子类声明了server能够实现的所有的方法。为client端的代理接口BpINTERFACE和server端的BnINTERFACE的共同接口类，这个共同接口类的目的就是保证service方法在C-S两端的一致性。

1. BpRefBase

BpRefBase负责管理Client端调用getService函数来查询Service Manager中注册的Service而获得的Binder实体的引用，也就是管理BpBinder。

1. BnINTERFACE(继承自INTERFACE)

在定义android native端的service时，每个service均继承自BnINTERFACE<XXX>(XXX为service name)。BnINTERFACE类型定义了一个onTransact函数，这个函数负责解包收到的Parcel并执行client端的请求的方法。

1. BpINTERFACE

为了能够使client能够像本地调用一样调用一个远程Server，Server端需要向Client提供一个接口，Client在在这个接口的基础上创建一个BpINTERFACE，使用这个对象，Client的应用能够想本地调用一样直接调用Server端的方法。BpINTERFACE实现了Service中各方法的本地操作，将每个方法的参数以Parcel的形式发送给BD。

1. Parcel

Parcel是binder IPC中的最基本的通信单元，它存储C-S间函数调用的参数.但是Parcel只能存储基本的数据类型，如果是复杂的数据类型的话，在存储时，需要将其拆分为基本的数据类型来存储。

1. Server和Client获得Service Manager过程分析

在示例代码中，Client和Server获取Service Manager都是通过调用函数defaultServiceManager()函数来获取。

1） defaultServiceManager()函数获取ServiceManager对象的过程分析

sp<IServiceManager> defaultServiceManager();--->interface\_cast<IServiceManager>(ProcessState::s elf()->getContextObject(NULL));----->interface\_cast <IServiceManager> (new BpBinder(0));------>调用了IServiceManager::asInterface函数。 IServiceManager::asInterface是通过DECLARE\_META\_INTERFACE

(ServiceManager)宏中声明的。------>最终是通过new BpServiceManag

er(new BpBinder(0))。

1. Server获得Service Manager之后的执行过程分析

1. 继承关系

在示例代码中，HmdService继承于 BnHmdService，BnHmdService继承于 BnInterface<IHmdService> , BnHmdService实际是继承了IHmdService和BBinder类。IHmdService类又分别继承了 IInterface 和IBinder类，IInterface 和IBinder类又同时继承了 RefBase类。

2. Service获取到SericeManager对象之后的执行过程分析

Service拿到Service Manager之后，调用IServiceManager::addService这个接口来和Binder驱动程序交互了，即调用BpServiceManager::addServ

ice。而BpServiceManager::addService()又会调用通过其基类BpRefBase的 成员函数remote()获得原先创建的BpBinder实例，接着调用BpBinder::tran

Sact()函数。在BpBinder::transact()函数中，又会调用IPCThreadState::tran

sact()函数，这里就是最终与Binder驱动程序交互的地方。IPCThreadSta

te有一个ProcessState类型的成员变量mProcess，而mProcess有一个成员 变量mDriverFD,它是设备文件/dev/binder的打开文件描述符，因此， IPCThreadState就相当于间接在拥有了设备文件/dev/binder的打开文件描 述符，于是，便可以与Binder驱动程序进行交互。

1. Client获取到ServiceManager之后的执行过程分析

1. Client通过defaultServiceManager()函数获取到ServiceManager对象之后，通过调用getService()函数来获取HmdService这个Server的远程接口。目标是要获得一个称为BpHmdService对象的IHmdService接口。BpHmdService继承于BpInterface<IHmdService>类，即BpHmdService继承了IHmdService类和BpRefBase类，这两个类又分别继承了RefBase类。BpRefBase类有一个成员变量mRemote，它的类型为IBinder，实际是一个BpBinder对象。BpBinder类使用了IPCThreadState类来与Binder驱动程序进行交互，而IPCThreadState类有一个成员变量mProcess，它的类型为ProcessState，IPCThreadState类借助ProcessState类来打开Binder设备文件/dev/binder，因此，它可以和Binder驱动程序进行交互。BpMediaPlayerService的构造函数有一个参数 impl，它的类型为 const sp<IBinder>& ,这个实际上就是一个BpBinder对象。

2. Client调用getService()函数可获得到远程Server的Binder实体引用，然后通过函数interface\_cast<IHmdService>()函数将获取得的Binder引用转化成IHmdService的客户端Service的引用（如下图9-1）。然后使用IHmdService客户端的Service引用去调用相应的执行函数，相应的函数中会将数据写到Parcel结构中去，然后通过remote()->transact(code,data,&reply)(其实remote()函数返回的是Server的Binder的一个实体引用BpBinder)函数的调用（如下图9-2），实质上就会调用IPCThreadState::transact(handle,code,data,&reply,flags)---

-->writeTransactionData()(将要传输的数据写入到IPCThreadState的成员变量 mOut中)------>waitForResponse()(这个函数通过 IPCThreadState::talkWithDriv

er()与驱动程序进行交互)。

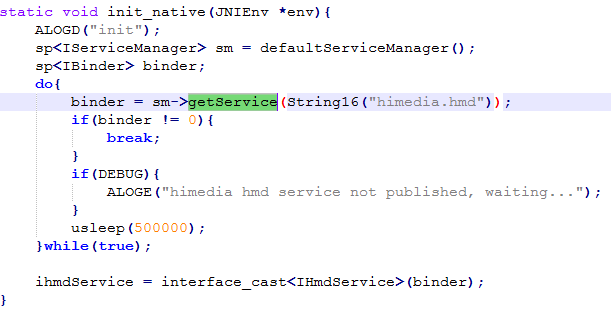


图 9-1

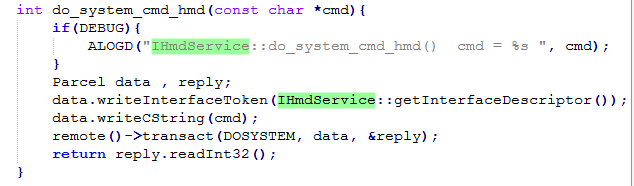


图 9-2

十、基本实现

如图10-1，该代码块中IHmdService继承的IInterface，是客户端和服务端的共同接口，进而保证客户端和服务端的执行方法是一致的。BnInterface是服务端接口,通过继承关系，表明BnHmdService是Binder进程间通信里的服务端。onTransact()函数是响应接收到数据的函数调用。宏DECLARE\_META\_INTERFACE进行声明，这样会添加静态字段descriptor，静态方法asInterface，虚方法getInterfaceDescriptor，以及构造函数和析构函数。另外只需要使用IMPLEMENT\_META\_INTERFACE(INTERFACE, NAME)来即可定义用宏DECLARE\_META\_INTERFACE声明的这些方法和字段。

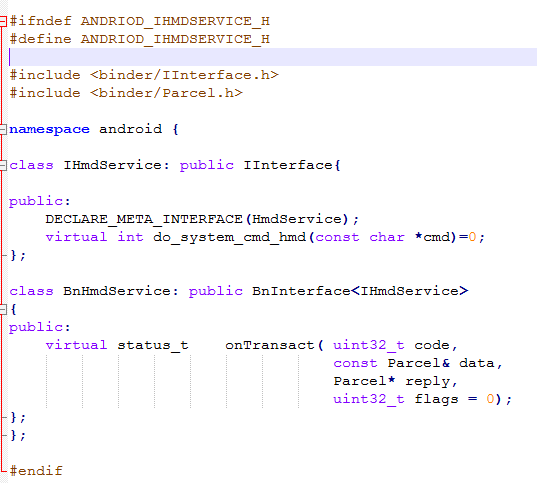
****

图 10-1

服务端真正实现的声明，instantiate()将service注册到ServiceManager服务管理中的函数声明。do\_system\_cmd\_hmd()执行系统命令的函数声明（如图10-2）。

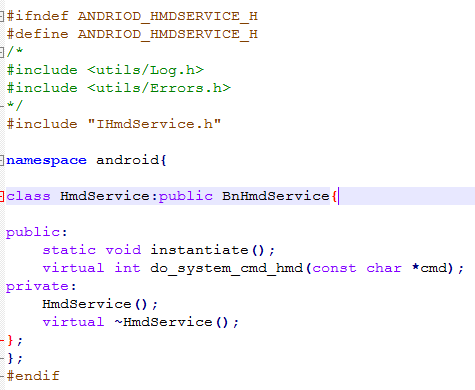


图 10-2

服务端的真正函数实现。Instantiate函数是将HmdService服务添加到ServiceManager中进行管理，其中“himedia.hmd”是HmdService的描述字符。do\_system\_cmd函数是引用的外部函数，来自于libhmd.so中。do\_system\_cmd\_hmd函数是服务端的函数实现（如图10-3）。

图 10-3

客户端的实现和服务端的onTransact()函数的实现，onTransact()函数是响应客户端的transact()函数的发送的Parcel数据。IMPLEMENT\_META\_INTERFACE()是对头文件中DECLARE\_META\_INTERFACE宏声明的实现（如图 10-4）。



图 10-4

如图10-5所示，此为客户端和服务端所对应的libhmdservice.so文件的mk文件。

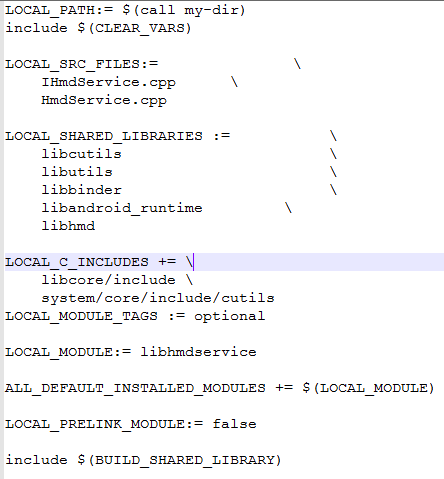


图 10-5

如图10-6所示，此为libhmd.so文件的实现，system()是Android系统中执行系统命令的函数。main()函数是生成可执行bin文件的函数执行入口，是为了测试system()函数是否可正确执行。



图 10-6

如图10-7所示，此为libhmd.so的mk文件，include $(BUILD\_SHARED\_LIBRARY)表明编译生成一个动态(so)库文件，将其换成 include($BUILD\_EXECUTABLE)，同是将libhmd的实现代码中将main()函数取消注释，则生成的是可执行bin文件。

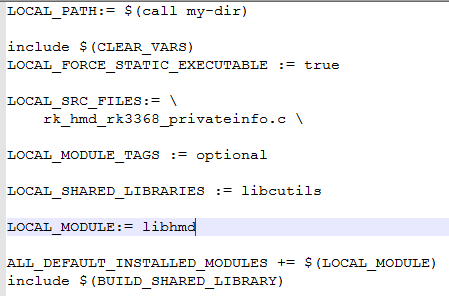


图 10-7

以下代码块是应用层（如图10-8）与JNI的实现（如图10-9），init\_native函数主要是从ServiceManager中获取指定的Service。do\_system\_cmd\_native函数是应用执行系统命令。

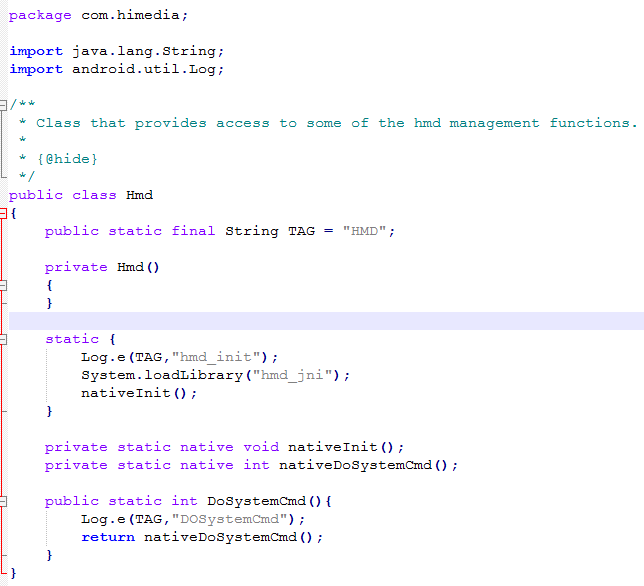


图 10-8

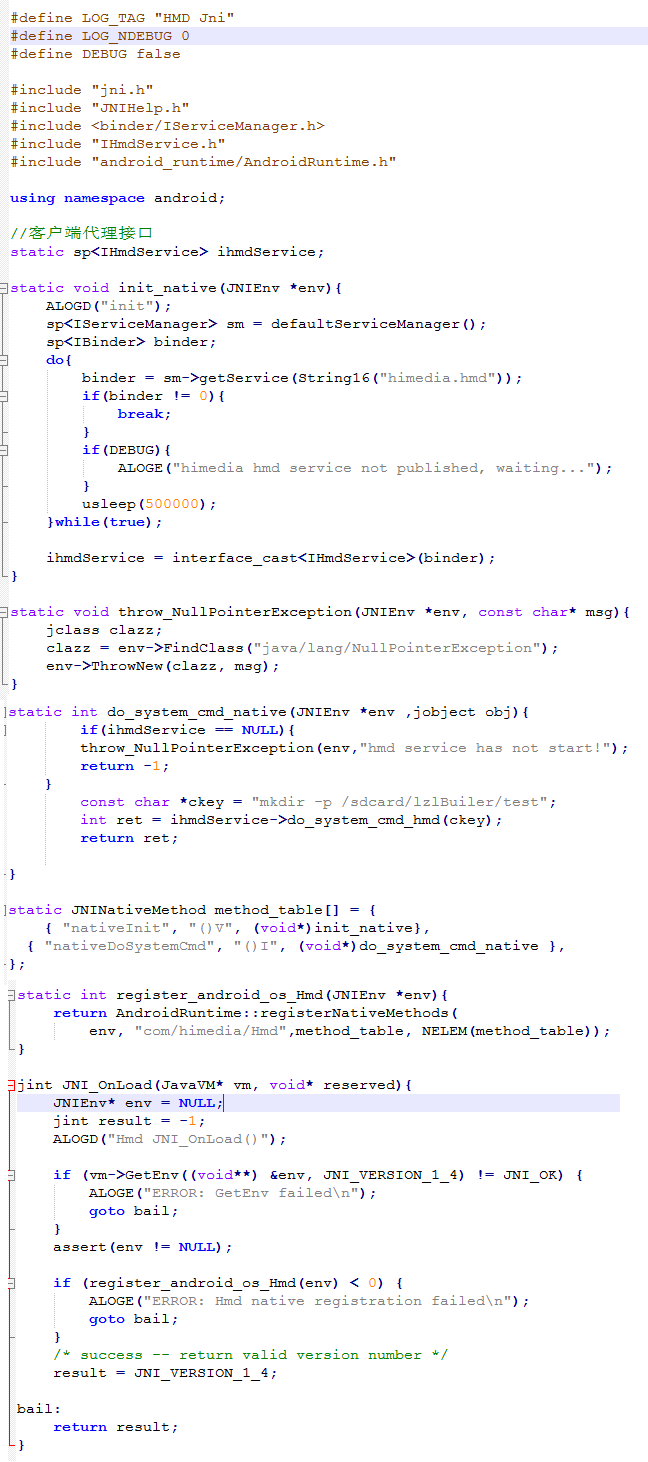


图 10-9

如图10-10所示，此为生成libhmd\_jni.so的mk文件。

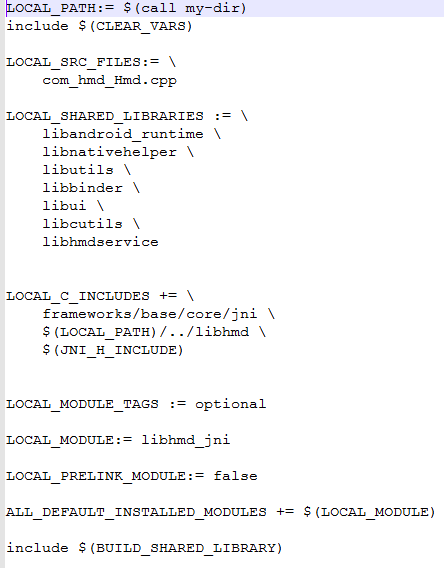


图 10-10

如图10-11所示，此文件是在系统启动时，将该指定的服务注册到ServiceManager，使得客户端能够通过getService函数获取到该Service对象。从而实现客户端和服务端的交互。图10-12中的mk文件，是为了将图11中的代码编译成可执行bin文件的mk文件。



图 10-11

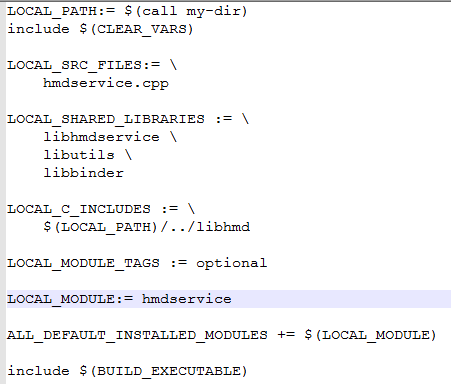


图 10-12

特别说明：

由于此Service是自定义的Service，并不是系统自带的，因此得在相应的文件中将该Service进行相应的添加。(以Amlogic T968为例)

1. 把himedia放到frameworks/base目录下
2. 在在X:\T968\_0422\device\customer\p313\device.mk增加

########################

#add by cg make hmd binder

PRODUCT\_PACKAGES += \

hmdservice \

libhmd\_jni \

libhmdservice \

hmd\_interface

################

3. 在X:\T968\_0422\build\core\pathmap.mk

在FRAMEWORKS\_BASE\_SUBDIRS := \ 下增加

himedia/external \

himedia/hmd \

4. 在X:\T968\_0422\device\amlogic\common\products\tv\init.amlogic.rc增加

#add by cg for hmd binder

service hmd /system/bin/hmdservice

class main

root

gruseroup root

ioprio rt 4

oneshot

总结：

代码执行流程：

1. 在系统启动时会解析加载init.rc文件，将自定义服务注册到ServiceManager进行管理（addService）。
2. 上层应用调用相应的native接口，从ServiceManager中获取到指定的服务引用（getService）。
3. 获取到服务端的Binder实体引用后，通过调用函数interface\_cast()函数将Binder实体引用转换为IHmdService的对象。
4. 调用执行系统命令的应用层方法（nativeDoSystemCmd），进而执行IHmdService客户端的相应包装方法（如do\_system\_cmd\_hmd方法），并将要传递的数据封装在Parcel中。
5. 调用remote()->transact()函数将封装在Parcel的数据传递到Binder Driver驱动中。
6. 服务端调用onTransact()函数，根据客户端传递的code来获取数据和执行相应的操作。

数据传输流程，见下图

