**[Android的IPC机制Binder的详解汇总](http://mintelong.javaeye.com/blog/458369)**

关键字: android、binder

第一部分 Binder的组成   
1.1 驱动程序部分驱动程序的部分在以下的文件夹中： 

kernel/include/linux/binder.h

kernel/drivers/android/binder.c

    binder驱动程序是一个miscdevice，主设备号为10，此设备号使用动态获得（MISC\_DYNAMIC\_MINOR），其设备的节点为：   
/dev/binder   
    binder驱动程序会在proc文件系统中建立自己的信息，其文件夹为/proc/binder，其中包含如下内容：   
proc目录：调用Binder各个进程的内容   
state文件：使用函数binder\_read\_proc\_state   
stats文件：使用函数binder\_read\_proc\_stats   
transactions文件：使用函数binder\_read\_proc\_transactions   
transaction\_log文件：使用函数binder\_read\_proc\_transaction\_log，其参数为binder\_transaction\_log (类型为struct binder\_transaction\_log)   
failed\_transaction\_log文件：使用函数binder\_read\_proc\_transaction\_log 其参数为   
binder\_transaction\_log\_failed (类型为struct binder\_transaction\_log)   
  
    在binder文件被打开后，其私有数据（private\_data）的类型：   
struct binder\_proc   
    在这个数据结构中，主要包含了当前进程、进程ID、内存映射信息、Binder的统计信息和线程信息等。   
    在用户空间对Binder驱动程序进行控制主要使用的接口是mmap、poll和ioctl，ioctl主要使用的ID为： 

#define BINDER\_WRITE\_READ \_IOWR('b', 1, struct binder\_write\_read)

#define BINDER\_SET\_IDLE\_TIMEOUT \_IOW('b', 3, int64\_t)

#define BINDER\_SET\_MAX\_THREADS \_IOW('b', 5, size\_t)

#define BINDER\_SET\_IDLE\_PRIORITY \_IOW('b', 6, int)

#define BINDER\_SET\_CONTEXT\_MGR \_IOW('b', 7, int)

#define BINDER\_THREAD\_EXIT \_IOW('b', 8, int)

#define BINDER\_VERSION \_IOWR('b', 9, struct binder\_version)

    BR\_XXX等宏为BinderDriverReturnProtocol，表示Binder驱动返回协议。   
    BC\_XXX等宏为BinderDriverCommandProtocol，表示Binder驱动命令协议。   
    binder\_thread是Binder驱动程序中使用的另外一个重要的数据结构，数据结构的定义如下所示： 

struct binder\_thread {

struct binder\_proc \*proc;

struct rb\_node rb\_node;

int pid;

int looper;

struct binder\_transaction \*transaction\_stack;

struct list\_head todo;

uint32\_t return\_error;

uint32\_t return\_error2;

wait\_queue\_head\_t wait;

struct binder\_stats stats;

};

    binder\_thread 的各个成员信息是从rb\_node中得出。   
    BINDER\_WRITE\_READ是最重要的ioctl，它使用一个数据结构binder\_write\_read定义读写的数据。 

struct binder\_write\_read {

signed long write\_size;

signed long write\_consumed;

unsigned long write\_buffer;

signed long read\_size;

signed long read\_consumed;

unsigned long read\_buffer;

};

1.2 servicemanager部分        servicemanager是一个守护进程，用于这个进程的和/dev/binder通讯，从而达到管理系统中各个服务的作用。   
        可执行程序的路径：   
        /system/bin/servicemanager          
开源版本文件的路径： 

frameworks/base/cmds/servicemanager/binder.h

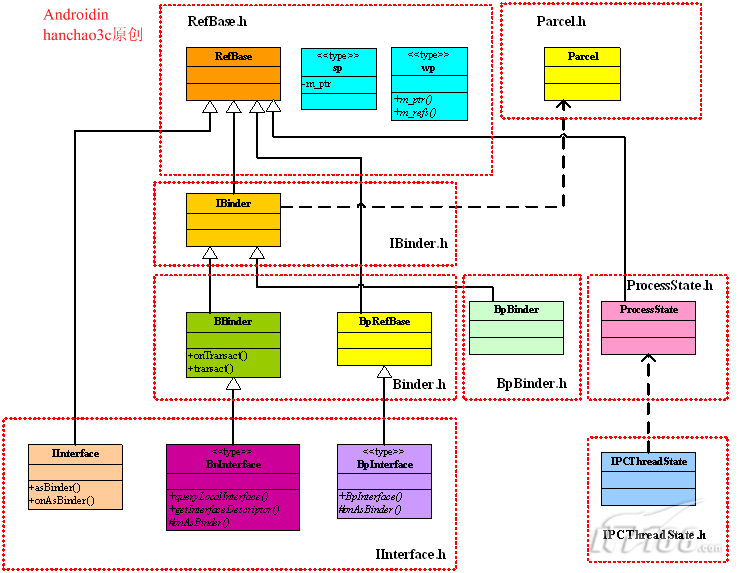
frameworks/base/cmds/servicemanager/binder.c

frameworks/base/cmds/servicemanager/service\_manager.c

       程序执行的流程：   
  
open()：打开binder驱动   
  
mmap()：映射一个128\*1024字节的内存   
  
ioctl(BINDER\_SET\_CONTEXT\_MGR)：设置上下文为mgr   
       进入主循环binder\_loop()   
             ioctl(BINDER\_WRITE\_READ)，读取   
                       binder\_parse()进入binder处理过程循环处理   
         binder\_parse()的处理，调用返回值：   
        当处理BR\_TRANSACTION的时候，调用svcmgr\_handler()处理增加服务、检查服务等工作。各种服务存放在一个链表（svclist）中。其中调用binder\_等开头的函数，又会调用ioctl的各种命令。   
        处理BR\_REPLY的时候，填充binder\_io类型的数据结   
1.3 binder的库的部分   
    binder相关的文件作为Android的uitls库的一部分，这个库编译后的名称为libutils.so，是Android系统中的一个公共库。   
    主要文件的路径如下所示： 

frameworks/base/include/utils/\*

frameworks/base/libs/utils/\*

     
    主要的类为：   
RefBase.h :   
    引用计数，定义类RefBase。   
Parcel.h :   
    为在IPC中传输的数据定义容器，定义类Parcel   
IBinder.h：   
    Binder对象的抽象接口， 定义类IBinder   
Binder.h：   
    Binder对象的基本功能， 定义类Binder和BpRefBase   
BpBinder.h：   
BpBinder的功能，定义类BpBinder   
IInterface.h：   
为抽象经过Binder的接口定义通用类，   
    定义类IInterface，类模板BnInterface，类模板BpInterface   
ProcessState.h   
    表示进程状态的类，定义类ProcessState   
IPCThreadState.h   
    表示IPC线程的状态，定义类IPCThreadState   
各个类之间的关系如下所示：   
  
  
  
    在IInterface.h中定义的BnInterface和BpInterface是两个重要的模版，这是为各种程序中使用的。   
BnInterface模版的定义如下所示： 

template

class BnInterface : public INTERFACE, public BBinder

{

public:

virtual sp queryLocalInterface(const String16& \_descriptor);

virtual String16 getInterfaceDescriptor() const;

protected:

virtual IBinder\* onAsBinder();

};

BnInterface模版的定义如下所示：

template

class BpInterface : public INTERFACE, public BpRefBase

{

public:

BpInterface(const sp& remote);

protected:

virtual IBinder\* onAsBinder();

};

         这两个模版在使用的时候，起到得作用实际上都是双继承：使用者定义一个接口INTERFACE，然后使用BnInterface和BpInterface两个模版结合自己的接口，构建自己的BnXXX和BpXXX两个类。   
         DECLARE\_META\_INTERFACE和IMPLEMENT\_META\_INTERFACE两个宏用于帮助BpXXX类的实现： 

#define DECLARE\_META\_INTERFACE(INTERFACE) \

static const String16 descriptor; \

static sp asInterface(const sp& obj); \

virtual String16 getInterfaceDescriptor() const; \

#define IMPLEMENT\_META\_INTERFACE(INTERFACE, NAME) \

const String16 I##INTERFACE::descriptor(NAME); \

String16 I##INTERFACE::getInterfaceDescriptor() const { \

return I##INTERFACE::descriptor; \

} \

sp I##INTERFACE::asInterface(const sp& obj) \

{ \

sp intr; \

if (obj != NULL) { \

intr = static\_cast( \

obj->queryLocalInterface( \

I##INTERFACE::descriptor).get()); \

if (intr == NULL) { \

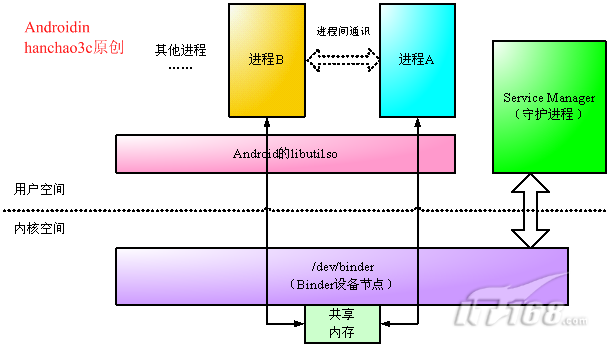
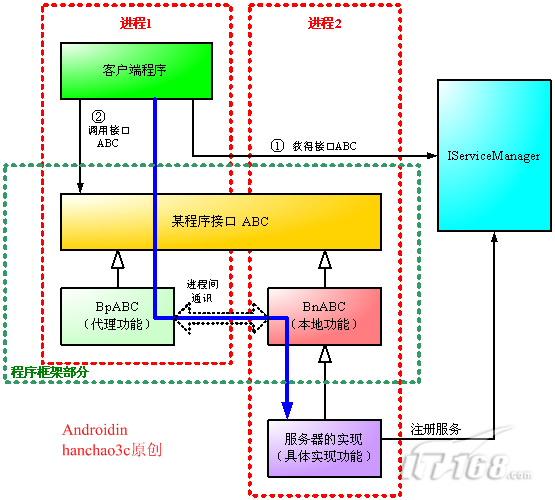
intr = new Bp##INTERFACE(obj); \

} \

} \

return intr; \

}

    
   在定义自己的类的时候，只需要使用DECLARE\_META\_INTERFACE和IMPLEMENT\_META\_INTERFACE两个接口，并   
结合类的名称，就可以实现BpInterface中的asInterface()和getInterfaceDescriptor()两个函数。   
第二部分 Binder的运作   
　　2.1 Binder的工作机制   
      Service Manager是一个守护进程，它负责启动各个进程之间的服务，对于相关的两个需要通讯的进程，它们通过调用libutil.so库实现通讯，而真正通讯的机制，是内核空间中的一块共享内存。   
        
   
  
　　2.2 从应　　用程序的角度看Binder   
  
　　从应用程序的角度看Binder一共有三个方面：   
　　Native 本地：例如BnABC，这是一个需要被继承和实现的类。   
　　Proxy 代理：例如BpABC，这是一个在接口框架中被实现，但是在接口中没有体现的类。   
　　客户端：例如客户端得到一个接口ABC，在调用的时候实际上被调用的是BpABC   
  
   
  
本地功能（Bn）部分做的：   
    实现BnABC:: BnTransact()   
    注册服务：IServiceManager：：AddService   
代理部分（Bp）做的：   
    实现几个功能函数，调用BpABC::remote()->transact()   
客户端做的：   
    获得ABC接口，然后调用接口（实际上调用了BpABC，继而通过IPC调用了BnABC，然后调用了具体的功能）   
  
       在程序的实现过程中BnABC和BpABC是双继承了接口ABC。一般来说BpABC是一个实现类，这个实现类不需要在接口中体现，它实际上负责的只是通讯功能，不执行具体的功能；BnABC则是一个接口类，需要一个真正工作的类来继承、实现它，这个类才是真正执行具体功能的类。   
       在客户端中，从ISeriviceManager中获得一个ABC的接口，客户端调用这个接口，实际上是在调用BpABC，而BpABC又通过Binder的IPC机制和BnABC通讯，BnABC的实现类在后面执行。   
　　事实上，   
服务器   
的具体实现和客户端是两个不同的进程，如果不考虑进程间通讯的过程，从调用者的角度，似乎客户端在直接调用另外一个进程间的函数——当然这个函数必须是接口ABC中定义的。   
　　2.3 ISericeManager的作用   
  
    ISericeManager涉及的两个文件是ISericeManager.h和ISericeManager.cpp。这两个文件基本上是   
ISericeManager。ISericeManager是系统最先被启动的服务。非常值得注意的是：ISericeManager本地功能并没有使   
现，它实际上由ServiceManager守护进程执行，而用户程序通过调用BpServiceManager来获得其他的服务。   
      在ISericeManager.h中定义了一个接口，用于得到默认的ISericeManager：   
        sp defaultServiceManager();   
     这时得到的ISericeManager实际上是一个全局的ISericeManager。   
第三部分 程序中Binder的具体实现   
　　3.1 一个利用接口的具体实现   
    PermissionController也是libutils中定义的一个有关权限控制的接口，它一共包含两个文件：IPermissionController.h和IPermissionController.cpp这个结构在所有类的实现中都是类似的。   
     头文件IPermissionController.h的主要内容是定义IPermissionController接口和类BnPermissionController： 

class IPermissionController : public IInterface

{

public:

DECLARE\_META\_INTERFACE(PermissionController);

virtual bool checkPermission(const String16& permission,int32\_t pid, int32\_t uid) = 0;

enum {

CHECK\_PERMISSION\_TRANSACTION = IBinder::FIRST\_CALL\_TRANSACTION

};

};

class BnPermissionController : public BnInterface

{

public:

virtual status\_t onTransact( uint32\_t code,

const Parcel& data,

Parcel\* reply,

uint32\_t flags = 0);

};

    IPermissionController是一个接口类，只有checkPermission()一个纯虚函数。   
  
  
BnPermissionController继承了以BnPermissionController实例化模版类BnInterface。因   
此，BnPermissionController，事实上BnPermissionController双继承了BBinder和   
IPermissionController。   
    实现文件IPermissionController.cpp中，首先实现了一个BpPermissionController。 

class BpPermissionController : public BpInterface

{

public:

BpPermissionController(const sp& impl)

: BpInterface(impl)

{

}

virtual bool checkPermission(const String16& permission, int32\_t pid, int32\_t uid)

{

Parcel data, reply;

data.writeInterfaceToken(IPermissionController::

getInterfaceDescriptor());

data.writeString16(permission);

data.writeInt32(pid);

data.writeInt32(uid);

remote()->transact(CHECK\_PERMISSION\_TRANSACTION, data, &reply);

if (reply.readInt32() != 0) return 0;

return reply.readInt32() != 0;

}

};

IMPLEMENT\_META\_INTERFACE(PermissionController, "android.os.IPermissionController");   
  
  
BpPermissionController继承了BpInterface，它本身是一个   
已经实现的类，而且并没有在接口中体现。这个类按照格式写就可以，在实现checkPermission()函数的过程中，使用Parcel作为传输数据   
的容器，传输中时候transact()函数，其参数需要包含枚举值CHECK\_PERMISSION\_TRANSACTION。   
IMPLEMENT\_META\_INTERFACE用于扶助生成。   
    BnPermissionController中实现的onTransact()函数如下所示： 

status\_t BnPermissionController:: BnTransact(

uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags)

{

switch(code) {

case CHECK\_PERMISSION\_TRANSACTION: {

CHECK\_INTERFACE(IPermissionController, data, reply);

String16 permission = data.readString16();

int32\_t pid = data.readInt32();

int32\_t uid = data.readInt32();

bool res = checkPermission(permission, pid, uid);

reply->writeInt32(0);

reply->writeInt32(res ? 1 : 0);

return NO\_ERROR;

} break;

default:

return BBinder:: BnTransact(code, data, reply, flags);

}

}

     在onTransact()函数中根据枚举值判断数据使用的方式。注意，由于BnPermissionController也是继承了类   
IPermissionController，但是纯虚函数checkPermission()依然没有实现。因此这个   
BnPermissionController类并不能实例化，它其实也还是一个接口，需要一个实现类来继承它，那才是实现具体功能的类。   
　　3.2 BnABC的实现   
    本地服务启动后将形成一个守护进程，具体的本地服务是由一个实现类继承BnABC来实现的，这个服务的名称通常叫做ABC。   
    在其中，通常包含了一个instantiate()函数，这个函数一般按照如下的方式实现：   
void ABC::instantiate() {   
    defaultServiceManager()->addService(   
            String16("XXX.ABC"), new ABC ());   
}   
    按照这种方式，通过调用defaultServiceManager()函数，将增加一个名为"XXX.ABC"的服务。   
    在这个defaultServiceManager()函数中调用了：   
ProcessState::self()->getContextObject(NULL));   
    IPCThreadState\* ipc = IPCThreadState::self();   
   IPCThreadState::talkWithDriver()   
在ProcessState 类建立的过程中调用open\_driver()打开   
驱动   
程序，在talkWithDriver()的执行过程中。   
　　3.3 BpABC调用的实现   
    BpABC调用的过程主要通过mRemote()->transact() 来传输数据，mRemote()是BpRefBase的成员，它是一个IBinder。这个调用过程如下所示： 

mRemote()->transact()

Process::self()

IPCThreadState::self()->transact()

writeTransactionData()

waitForResponse()

talkWithDriver()

ioctl(fd, BINDER\_WRITE\_READ, &bwr)

    在IPCThreadState::executeCommand()函数中，实现传输操作。