Android Binder

Mechanism for the various parts of Binder

Published: 2009-5-13 16:48 | Author: hanchao3c | Source: Android Developers

번역/정리: 고도리(http://www.aesop.or.kr), 2009/08/19 version

제 1부 Binder의 구성

1.1 driver분석

2.6.29 kernel의 경우는 다음과 같은 디렉토리에 위치한다.

drivers/staging/android/binder.h
drivers/staging/android/binder.c

binder는 miscdevice로 등록이 되며(major 10, minor는 dynamic), user space에서 나타나는 이름을 /dev/binder로 나타나게 된다.

binder 드라이버는 proc file system(/proc)을 이용해서 정보를 확인할 수 있으며 그 디렉토리는 /proc/binder이다.

이 디렉토리에는 다음과 같은 정보가 담기게 된다.

proc directory: Binder를 호출한 process의 내용
state: binder_read_proc_state() 함수를 이용해서 binder state 내용을 보여준다.
stats: binder_read_proc_stats()
transactions: binder_read_proc_transactions()
transaction_log: binder_read_proc_transaction_log(), 이 함수의 파라미터는 다음과 같다. binder_transaction_log (type struct binder_transaction_log)
failed_transaction_log: binder_read_proc_transaction_log(), 이 함수의 파라미터는 다음과 같다. binder_transaction_log_failed (type struct binder_transaction_log)

binder는 open되어 있고, binder를 사용하는데 사용하는 data는

struct binder_proc
이다.(drivers/staging/android/binder.c)

이 데이터 스터럭처에 포함되어 있는 주요 부분은

currnet process
process ID
memory mapping information

binder의 통계정보
thread 정보

user space에서 binder를 제어하는 방법은 mmap, poll, ioctl 등이다.

ioctl에서 사용되는 command는 다음과 같다.
(drivers/staging/android/binder.h)

CODE:
```
#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)
```

binder의 동작에는 driver의 return값고, driver로 command를 내려줘야할 경우가 있는데

binder.h에는 다음과 같은 enum형들이 존재한다.

CODE:
```
enum BinderDriverReturnProtocol {
    BR_ERROR = _IOR('r', 0, int),
    /*
     * int: error code
     */

    BR_OK = _IO('r', 1),
    /* No parameters! */

    BR_TRANSACTION = _IOR('r', 2, struct binder_transaction_data),
    BR_REPLY = _IOR('r', 3, struct binder_transaction_data),
......

enum BinderDriverCommandProtocol
{
```

```
    BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
    BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
...
```

binder driver에는 중요한 정보를 담고 있는 또 다른 structure가 있다.

그 정의는 다음과 같다.

CODE:

```
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; /* Write failed, return error code in read buf */
    uint32_t return_error2; /* Write failed, return error code in read */
        /* buffer. Used when sending a reply to a dead process that */
        /* we are also waiting on */
    wait_queue_head_t wait;
    struct binder_stats stats;
};
```

여기서 rb_node를 눈여겨 봐야한다(include/linux/rbtree.h...스케쥴링 관련)

BINDER_WRITE_READ ioctl 에 사용되는 structure는 다음과 같다.

CODE:

```
struct binder_write_read {
    signed long write_size; /* bytes to write */
    signed long write_consumed; /* bytes consumed by driver */
    unsigned long    write_buffer;
    signed long read_size;  /* bytes to read */
    signed long read_consumed;  /* bytes consumed by driver */
    unsigned long    read_buffer;
};
```

1.2 servicemanager 부분 분석

servicemanager는 daemon process이며 다양한 service를 제공하는 역할을 담당하며, /dev/binder 통신을 관리한다.

Android root filesystem에서 실행파일 path는 다음과 같고
/system/bin/servicemanager

소스는 이 위치에 존재한다.

commands/binder/binder.c
commands/binder/binder.h
commands/binder/service_manager.c

Open-source인 Android에서는 다음과 같은 위치에 존재한다.

frameworks/base/cmds/servicemanager/binder.h
frameworks/base/cmds/servicemanager/binder.c
frameworks/base/cmds/servicemanager/service_manager.c

service_manager.c 의 메인함수는 다음과 같다.

CODE;

```
int main(int argc, char **argv)
{
    struct binder_state *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;

    bs = binder_open(128*1024);

    if (binder_become_context_manager(bs)) {
        LOGE("cannot become context manager (%s)₩n", strerror(errno));
        return -1;
    }

    svcmgr_handle = svcmgr;
    binder_loop(bs, svcmgr_handler);
    return 0;
}
```

이 프로세스의 동작은 다음과 같다.

open (): open binder driver
mmap (): mapping of a 128 * 1024 bytes of memory
ioctl (BINDER_SET_CONTEXT_MGR): set the context for the mgr
메인 루프인 binder_loop ()에서는
ioctl (BINDER_WRITE_READ)를 이용해서 읽고
binder_parse()를 이용해서 binder를 다루는 루틴으로 들어간다.

binder_parse() 함수는 다음과 같은 처리를 한다.

ioctl()로 read된 data를 cmd 변수로 받고, BR_TRANSACTION처리에서는 service를 증가시키고, 감시하기 위하여

CODE:
```
void binder_loop(struct binder_state *bs, binder_handler func)
```

의 2번째 인자로 입력된 func를 호출한다(func가 svcmgr_handler()함수이다)
(func는 binder_parse()의 마지막 인자로 입력된다)

CODE:
```
int binder_parse(struct binder_state *bs, struct binder_io *bio,
                 uint32_t *ptr, uint32_t size, binder_handler func)
{
    int r = 1;
    uint32_t *end = ptr + (size / 4);

    while (ptr < end) {
        uint32_t cmd = *ptr++;
#if TRACE
        fprintf(stderr,"%s:₩n", cmd_name(cmd));
#endif
        switch(cmd) {
        case BR_TRANSACTION: {
            struct binder_txn *txn = (void *) ptr;
            if ((end - ptr) * sizeof(uint32_t) < sizeof(struct binder_txn)) {
                LOGE("parse: txn too small!₩n");
                return -1;
            }
            binder_dump_txn(txn);
            if (func) {
                unsigned rdata[256/4];
                struct binder_io msg;
```

```
            struct binder_io reply;
            int res;

            bio_init(&reply, rdata, sizeof(rdata), 4);
            bio_init_from_txn(&msg, txn);
            res = func(bs, txn, &msg, &reply); // 여기서 svcmgr_handler() 호출
            binder_send_reply(bs, &reply, txn->data, res);
        }
        ptr += sizeof(*txn) / sizeof(uint32_t);
        break;
    }
```

다양한 서비스들은 svclist 라는 linked list에 저장이 되며 이 리스트는 binder_ 로 시작하는 함수들에서 사용된다(ex> binder_acquire()로 입력되는 값으로 사용된다)

binder_parse()의 switch 문에서

BR_REPLY를 처리할 때 binder_io type의 data node를 채운다.

(참고: servicemanager 디렉토리의 Android.mk를 보면, bctest란 binder test program을 컴파일 할 수 있도록 되어 있다. 참고할 것

```
ifneq ($(TARGET_SIMULATOR),true)
LOCAL_PATH:= $(call my-dir)

#include $(CLEAR_VARS)
#LOCAL_SRC_FILES := bctest.c binder.c
#LOCAL_MODULE := bctest
#include $(BUILD_EXECUTABLE)

include $(CLEAR_VARS)
LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := service_manager.c binder.c
LOCAL_MODULE := servicemanager
include $(BUILD_EXECUTABLE)
endif
)
```

1.3 Binder의 핵심부분

binder 관련소스는 Android의 utils을 구성하는 핵심부분이다. 이 핵심부분은 컴파일 된 후는 libutils.so 를 구성하게 되고, 안드로이드 시스템에서의 public library로 사용된다.

파일의 경로는 다음과 같다.

include/utils/*
libs/utils/*

Android 소스(오픈소스 버전)에서는
frameworks/base/include/utils/*
frameworks/base/libs/utils/*

의 파일들이다.

메인 header들은 다음과 같다.

RefBase.h: Reference count를 다루는 RefBase class에 대한 정의.
Parcel.h: IPC transmission에 사용되는 data container에 대한 정의.
IBinder.h: Binder abstract object interface, IBinder Class에 대한 정의
Binder.h: binder, binder object의 기본적인 함수, BpRefBase에 대한 class를 정의.
BpBinder.h: BpBinder functions, BpBinder class에 대한 정의
IInterface.h: 일반적인 class를 통한 binder에 대한 추상적인 인터페이스를 정의
IInterface, class template BnInterface, class template BpInterface 에 대한 정의가 있음.
ProcessState.h: process의 state에 대한 class인 ProcessState를 정의
IPCThreadState.h: IPC thread의 상태를 정의, IPCThreadState class를 정의.

다양한 class 사이의 관계는 다음과 같다.

IInterface.h에 정의된 두 개의 중요한 class template 인 BnInterface와 BpInterface는 다양한 프로시저에서 사용된다(ex> camera interface에서 사용됨)

BnInterface template에 대한 정의는 다음과 같다.

CODE:
```
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
```

```
    virtual sp<IInterface>        queryLocalInterface(const String16& _descriptor);
    virtual String16              getInterfaceDescriptor() const;

protected:
    virtual IBinder*              onAsBinder();
};
```

BpInterface에 대한 정의는 다음과 같다.

CODE:

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
                                  BpInterface(const sp<IBinder>& remote);

protected:
    virtual IBinder*              onAsBinder();
};
```

사실상 두 template이 사용될 때는 계승을 받아서 사용하게 되는데, 두 개의 역할을 각각 하게 된다.

사용자 정의 interface인 INTERFACE가 두 template인 BnInterface와 BpInterface의 자체 인터페이스인 BpXXX, BnXXX를 만들 때 사용된다.

DECLARE_META_INTERFACE와 IMPLEMENT_META_INTERFACE 가 BpXXX interface를 만들 때 사용이 된다.

CODE:

```
//
-----------------------------------------------------------------------
--

#define DECLARE_META_INTERFACE(INTERFACE)                                ₩
    static const String16 descriptor;                                ₩
    static sp<I##INTERFACE> asInterface(const sp<IBinder>& obj);        ₩
    virtual String16 getInterfaceDescriptor() const;                 ₩

#define IMPLEMENT_META_INTERFACE(INTERFACE, NAME)
₩
```

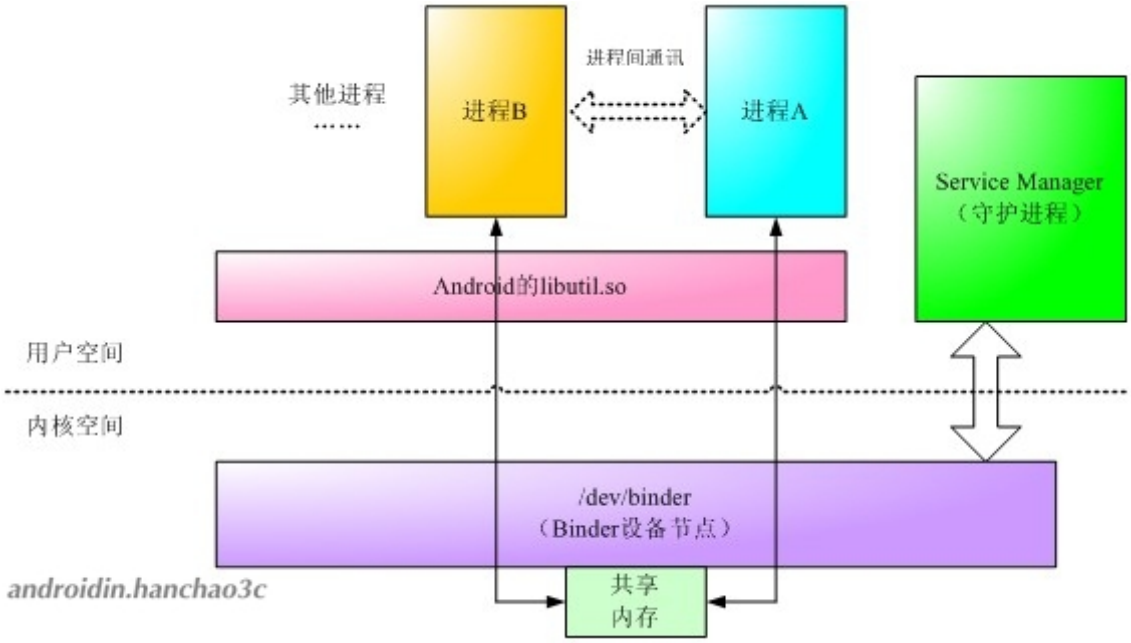
```
const String16 I##INTERFACE::descriptor(NAME);                    ₩
String16 I##INTERFACE::getInterfaceDescriptor() const {           ₩
    return I##INTERFACE::descriptor;                              ₩
}                                                                 ₩
sp<I##INTERFACE> I##INTERFACE::asInterface(const sp<IBinder>& obj)  ₩
{                                                                 ₩
    sp<I##INTERFACE> intr;                                        ₩
    if (obj != NULL) {                                            ₩
        intr = static_cast<I##INTERFACE*>(                        ₩
            obj->queryLocalInterface(                             ₩
                    I##INTERFACE::descriptor).get());             ₩
        if (intr == NULL) {                                       ₩
            intr = new Bp##INTERFACE(obj);                        ₩
        }                                                         ₩
    }                                                             ₩
    return intr;                                                  ₩
}                                                                 ₩
```

//

--

사용자 정의의 자체 class를 정의할 때 name과 위의 두 개의 매크로 인터페이스만 사용하면 된다.
BpInterface는 asInterface()와 getInterfaceDescriptor()함수만 설정함으로써 만들어질 수 있다.

2.1 Binder의 동작 메카니즘

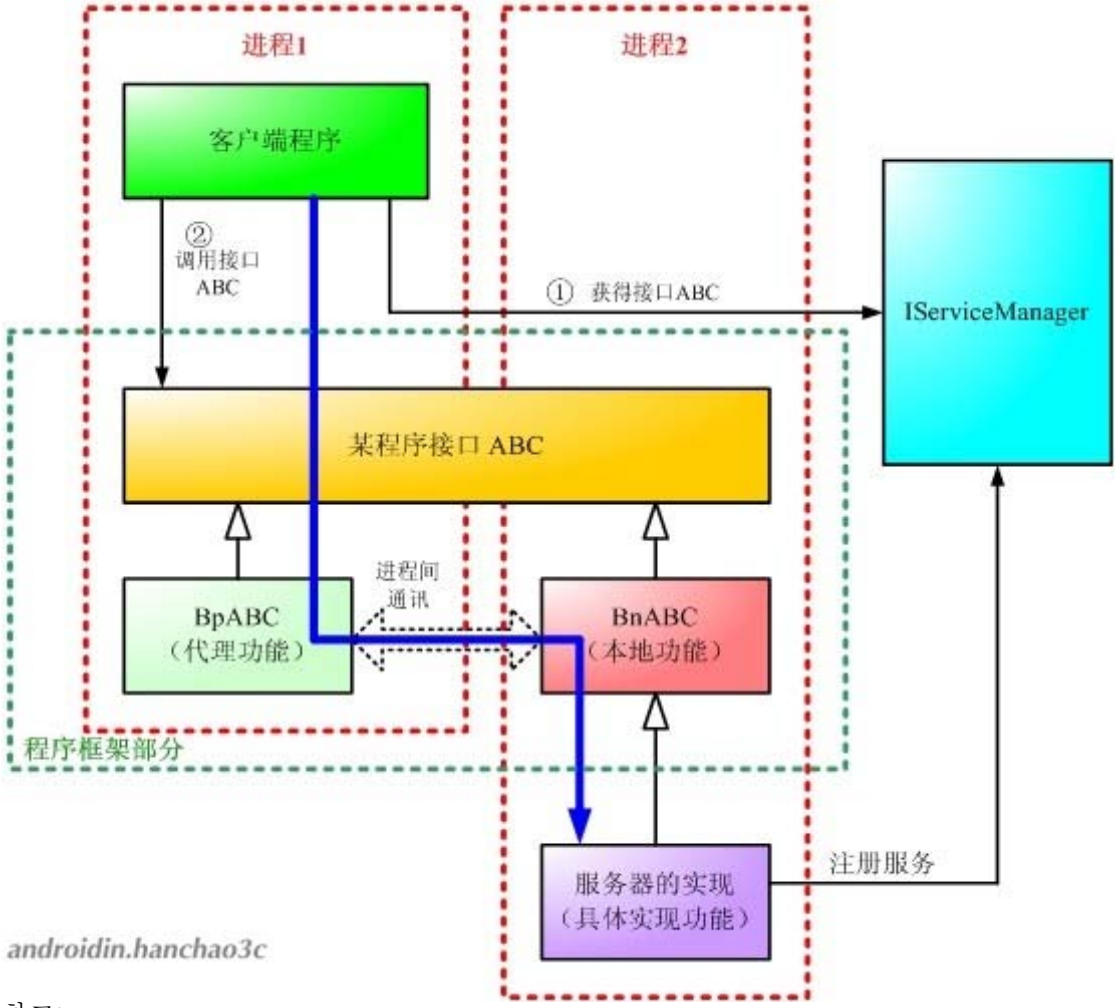


service manager는 process communication이 필요한 두 개의 process에 서비스를 시작하게 해주는 복잡한 daemon process이다. 두 개의 process는 libutil.so 를 호출함으로서 통신이 이루어지며, 실제 통신에서는 커널 space의 shared memory 영역을 사용하게 된다.

2.2 Application입장에서의 Binder

application 입장에서 바라봤을 경우에는 binder에 대해 2개의 관점으로 볼 수 있다.

■ Native local: for example BnABC, 이 class를 실제로 구현하기 위해서는 상속받는것이 필요하다.
■ Proxy agent: for example BpABC, 실현되어 있는 interface에 대한 framework이다. 하지만 인터페이스는 class를 그대로 반영하지는 않는다(not reflect)
■ client: for example, 실제로 BpABC를 호출했을 때 client interface ABC가 된다.



참고:
功能: 기능,작용 혹은 function
某程序接口: a program interface
注册服务: registration
程序框架部分: part framework
部分做的: 做 - 지을 주, 맡다, 책임을 지다.
几个: 몇 개
调用: 조용, 사용하다...호출하다(call)
客户: client
获得: 획득, 얻다 취득하다...

Native함수(local)인 Bn이 맡는 부분:
BnABC::onTransact()을 실행하고
IServiceManager::addService() 를 이용해서 service를 등록한다.

Proxy함수인 Bp 가 맡는 부분:
몇 개의 함수를 실행한다. BpABC::remote()->transact() 함수를 호출(해서 접속함)

client가 맡는 부분:
ABC에 대한 aceess를 획득하기 위하여, 인터페이스 함수를 호출한다(실은 BpABC를 호출한다. 그리고 나서 BnABC로의 IPC 통신을 하게 된다. 그리고, 관련함수들을 호출한다)

프로세스의 생성이 끝나면 BnABC와 BpABC 인터페이스는 ABC를 상속한다.
BpABC는 일반적인 방법으로 실행되며, 구현된 class는 실제 통신에의 응답을 반영하도록 작성할 필요는 없고, 실제 함수로도 구현되지 않는다(즉, 통신개념이니 실제 일은 BnABC쪽에서 한다는 말이다).
BnABC는 맡은 일을 수행하기 위해서는 class 상속이 실제로 필요한 인터페이스 클래스이다. 이 클래스는 특정 기능을 실행되기 위해 구현된 실제 함수들로 이루어져 있다.

IServiceManager로부터 service를 획득한 client ABC인터페이스는, client interface를 호출한다. 실제 이 호출은 BpABC에 대한 호출이다.
Binder의 IPC 메카니즘을 통해서 BpABC와 BnABC는 통신을 하게 되고, BnABC는 실제 특정 타입에 대한 기능을 실현하게 된다(특정 class기능 실행: ex> camera interface).

사실, 두 개의 프로세스는 서버와 클라이언트의 관계로 실제 단단하게 연결이 맺어지게 된다. 이것은 process간의 통신으로 생각할 필요가 없이, client에서 직접적으로 프로세스간의 함수를 호출하는 것처럼 보인다. 물론, 이 함수들은 반드시 ABC에 정의되어 있어야 한다(Bp, Bn.. 양쪽 다...)

2.3 IServiceManager 동작

IServiceManager는 IServiceManager.h와 IServiceManager.cpp에 해당 코드가 있다. IServiceManager는 ServiceManager daemon으로부터 실행이 되며, 사용자 프로그램은 BpServiceManager를 통해서 다른 서비스를 요청할 수 있다.

IServiceManager.h 에는 다음과 같은 default IServiceManager 인터페이스가 정의되어 있다.

CODE:
```
sp<IServiceManager> defaultServiceManager();
```

3.1 실제 어떻게 사용되는지에 대한 예를 보자

인터페이스 접근제어를 담당하는 PermissionController는 libutils에 정의되어 있으며

IPermissionController.h
IPermissionController.cpp

이 두 개의 파일에 해당 내용이 존재한다.

IPermissionController.h에는 주요 인터페이스에 대한 정의가 되어 있고, IPermissionController로부터 상속받은 BnPermissionController 에 대한 것이 정의 되어 있다.

CODE:
```
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<IPermissionController>
{
public:
    virtual status_t     onTransact( uint32_t code,
                                     const Parcel& data,
                                     Parcel* reply,
                                     uint32_t flags = 0);
};
```

IPermissionController class는 완전 virtual 함수의 형태를 갖는 checkPermission()만을 멤버로 가지고 있다.

BnPermissionController 는 BnInterface라는 template로부터 상속이 되었다. 그러므로,

BnPermissionController는 실제로 BBinder(BnInterface가 BBinder에서 상속되는 class이므로)와 IPermissionController class로부터 이중으로 상속이 된 것이다.

BpPermissionController는 IPermissionController.cpp에서 구현되었다.

CODE:

```
class BpPermissionController : public BpInterface<IPermissionController>
{
public:
    BpPermissionController(const sp<IBinder>& impl)
        : BpInterface<IPermissionController>(impl)
    {
    }

    virtual bool checkPermission(const String16& permission, int32_t pid, int32_t uid)
    {
        Parcel data, reply; // data 전송을 위한 Parcel class를 이용하였음
        data.writeInterfaceToken(IPermissionController::getInterfaceDescriptor());
        data.writeString16(permission);
        data.writeInt32(pid);
        data.writeInt32(uid);
        remote()->transact(CHECK_PERMISSION_TRANSACTION, data, &reply);
        // fail on exception
        if (reply.readInt32() != 0) return 0;
        return reply.readInt32() != 0;
    }
};

IMPLEMENT_META_INTERFACE(PermissionController, "android.os.IPermissionController");
```

IMPLEMENT_META_INTERFACE 매크로를 이용하여 인터페이스를 생성하였다.

BnPermissionController의 onTransact() 함수는 다음과 같이 구현이 되었다(서버 파트에서 호출되는).

```
status_t BnPermissionController::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    //printf("PermissionController received: "); data.print();
    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);
            // write exception
            reply->writeInt32(0);
            reply->writeInt32(res ? 1 : 0);
            return NO_ERROR;
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
```

onTransact()함수에서는 입력된 값에 따라 값을 switch case 문을 이용해서 검사해서 적용이 되도록 코드가 되어 있다.

Note:
BnPermissionController는 IPermissionController로부터 상속되었기 때문에, 순수 virtual 함수인 checkPermission()이 아직 실체화 되지 않은 상태이다.

BnPermissionController는 여기서는 초기화 되지 않으며 실제적으로는 인터페이스만 존재하게 된다. 이것은 상속을 받을 때 실제로 실체화가 되며 특정 형태의 함수로 세팅이 되게 된다(이 부분은 자세히 다시 봐야할 듯).

3.2 BnABC의 실체화

process에 대한 보호처리를 끝낸후에 시작되는 local service는 BnABC 상속에 의해 실제 class가 실체화 됨으로써 제공되게 된다.

service의 이름은 일반적으로 ABC 라고 이름을 붙이게 된다.

보통 ABC라는 service는 instantiate()라는 함수를 포함하고 있으며, 이 함수는 일반적으로 다음과 같은 형태가 된다.

CODE:
```
void ABC::instantiate() {
        defaultServiceManager()->addService(
        String16("XXX.ABC"), new ABC ());
}
```

이러한 방법으로 defaultServiceManager()을 호출함으로써, "XXX.ABC" services를 기존의 서비스들에 추가하게 된다.

이 defaultServiceManager()는 다음과 같은 함수들을 호출하게 된다.

```
ProcessState::self()->getContextObject(NULL));
IPCThreadState* ipc = IPCThreadState::self();
IPCThreadState::talkWithDriver()
```

ProcessState class는 생성자에서 open_driver()를 호출해서 talkWithDriver()를 구현한 프로세스에 접근할 수 있는 통로를 설정하게 된다.

3.3 BpABC 호출의 실체화

BpABC mainly through the process of calling mRemote () -> transact () to transmit data, mRemote () is a member of BpRefBase, it is a IBinder. The course of this call is as follows:

```
mRemote () -> transact ()
        Process:: self ()
        IPCThreadState:: self () -> transact ()
        writeTransactionData ()
        waitForResponse ()
        talkWithDriver ()
                ioctl (fd, BINDER_WRITE_READ, & bwr)
```

In IPCThreadState:: executeCommand () function to realize transfer operation.

BpABC는 주로 mRemote()->transact() 함수를 이용해서 데이터를 전송한다.
mRemote() 함수는 BpRefBase의 멤버로 이것이 바로 IBinder이다.
이 함수 호출의 순서는 다음과 같다.

```
mRemote()->transact()
       Process::self()
```

```
IPCThreadState::self()->transact()
writeTransactionData()
waitForResponse()
talkWithDriver()
        ioctl(fd, BINDER_WRITE_READ, &bwr)
```

IPCThreadState::executeCommand() 함수에서 실제 전송이 이루어진다.