android binder 讲解

下面进行详细讲述 Android Binder 机制问题，Binder 机制是通过驱动的形式来实现，其实驱动程序的部分是保存在源代码的以下的文件中。

Android Binder 机制大部分都是使用的 IPC，进程间通信机制有很多种，例如 linux 中可以采用管道，消息队列，信号，共享内存，socket 等，这些都可以实现进程间的通信。

Android Binder 机制通信是基于 Service 与 Client 的，有一个 ServiceManager 的守 护进程管理着系统的各个服务，它负责监听是否有其他程序向其发送请求。如果有请求就响应。每个服务都要在 ServiceManager 中注册，而请求服务 的客户端去 ServiceManager 请求服务。

binder 的通信操作类似线程迁移（thread migration），binder 的用户空间为每一个进程维护着一个可用的线程池, 用来处理到来的 IPC 以及执行本地消息。两个进程间通信就好像是一个进程进入另一个进程执行代码然后带着执行的结果返回，Android 和驱动程序通信采用 linux 的 ioctl 机制。下面先简单介绍一下 ioctl 机制。

什么是 ioctl

ioctl 是设备驱动程序中对设备的 I/O 通道进行管理的函数。所谓对 I/O 通道进行管理，就是对设备的一些特性进行控制，例如串口的传输波特率、 马达的转速等等。它的调用函数如下：int ioctl(int fd, ind cmd, …)；其中 fd 就是用户程序打开设备时使用 open 函数返回的文件标示符，cmd 就是用户程序对设备的控制命令，至于后面的省略号。

那是一些补充参数，一般最多一个，有或没有是和 cmd 的意义相关的。ioctl 函数是文件结构中的一个属性分量。就是说如果你的驱动程序提供了对 ioctl 的支持，用户就可以在用户程序中使用 ioctl 函数控制设备的 I/O 通道。

ioctl 的必要性

如果不用 ioctl 的话，也可以实现对设备 I/O 通道的控制，但那就太复杂了。例如，我们可以在驱动程序中实现 write 的时候检查一下是否有特殊 约定的数据流通过。如果有的话，那么后面就跟着控制命令（一般在 socket 编程中常常这样做）。但是如果这样做的话，会导致代码分工不明，程序结构混 乱。

程序员自己也会头昏眼花的。所以，我们就使用 ioctl 来实现控制的功能。要记住，用户程序所作的只是通过命令码告诉驱动程序它想做什么，至于怎么解释这些命令和怎么实现这些命令，这都是驱动程序要做的事情。

Android Binder 机制如 何实现在驱动程序中实现的 ioctl 函数体内，实际上是有一个 switch{case}结构，每一个 case 对应一个命令码，做出一些相应的操作。怎么实 现这些操作，这是每一个程序员自己的事情，因为设备都是特定的。关键在于怎么样组织命令码，因为在 ioctl 中命令码是唯一联系用户程序命令和驱动程序支 持的途径。命令码的组织是有一些讲究的。

因为我们一定要做到命令和设备是一一对应的，这样才不会将正确的命令发给错误的设备，或者是把错误的命令发给正确的设备。或者是把错误的命令发给错 误的设备。这些错误都会导致不可预料的事情发生，而当程序员发现了这些奇怪的事情的时候，再来调试程序查找错误，那将是非常困难的事情。

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：

Java 代码

1. kernel/include/linux/binder.h
2. 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：

Java 代码

```
1.  #define BINDER_WRITE_READ       _IOWR('b', 1, struct binder_write_read)
2.  #define BINDER_SET_IDLE_TIMEOUT  _IOW('b', 3, int64_t)
3.  #define BINDER_SET_MAX_THREADS   _IOW('b', 5, size_t)
4.  #define BINDER_SET_IDLE_PRIORITY _IOW('b', 6, int)
5.  #define BINDER_SET_CONTEXT_MGR   _IOW('b', 7, int)
6.  #define BINDER_THREAD_EXIT       _IOW('b', 8, int)
7.  #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：

Java 代码

```
1.  struct binder_thread {
2.      struct binder_proc *proc;
3.      struct rb_node rb_node;
4.      int pid;
5.      int looper;
6.      struct binder_transaction *transaction_stack;
7.      struct list_head todo;
8.      uint32_t return_error;
9.      uint32_t return_error2;
10.     wait_queue_head_t wait;
11.     struct binder_stats stats;
12. };
```

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。

Java 代码

```
1.  struct binder_write_read {
2.      signed long write_size;
3.      signed long write_consumed;
4.      unsigned long write_buffer;
5.      signed long read_size;
6.      signed long read_consumed;
7.      unsigned long read_buffer;
8.  };
```

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：

Java 代码

```
1.  frameworks/base/cmds/servicemanager/binder.h
2.  frameworks/base/cmds/servicemanager/binder.c
3.  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

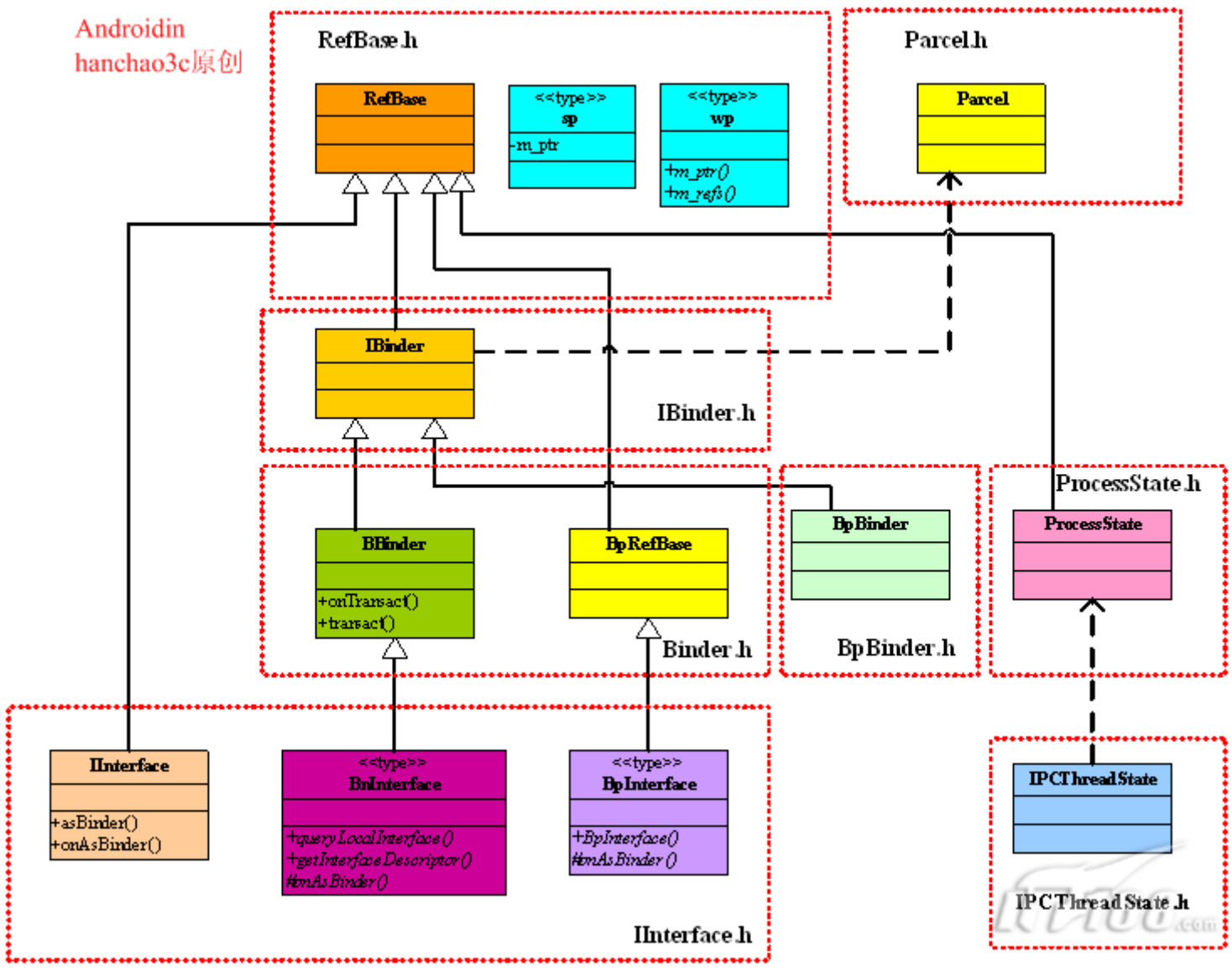
1.3 binder 的库的部分

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：

Java 代码

1. frameworks/base/include/utils/*
2. 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：

Java 代码

```
1.  template
2.  class BnInterface : public INTERFACE, public BBinder
3.  {
4.  public:
5.      virtual sp  queryLocalInterface(const String16& _descriptor);
6.      virtual String16        getInterfaceDescriptor() const;
7.  protected:
8.      virtual IBinder*        onAsBinder();
9.  };
10.     BnInterface 模版的定义如下所示：
11. template
12. class BpInterface : public INTERFACE, public BpRefBase
13. {
14. public:
15.                     BpInterface(const sp& remote);
16. protected:
17.     virtual IBinder*    onAsBinder();
18. };
```

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：

Java 代码

```
1.  #define DECLARE_META_INTERFACE(INTERFACE)                           \
2.      static const String16 descriptor;                               \
3.      static sp asInterface(const sp& obj);       \
4.      virtual String16 getInterfaceDescriptor() const;                \
5.  #define IMPLEMENT_META_INTERFACE(INTERFACE, NAME)                    \
6.      const String16 I##INTERFACE::descriptor(NAME);                  \
7.      String16 I##INTERFACE::getInterfaceDescriptor() const {         \
```

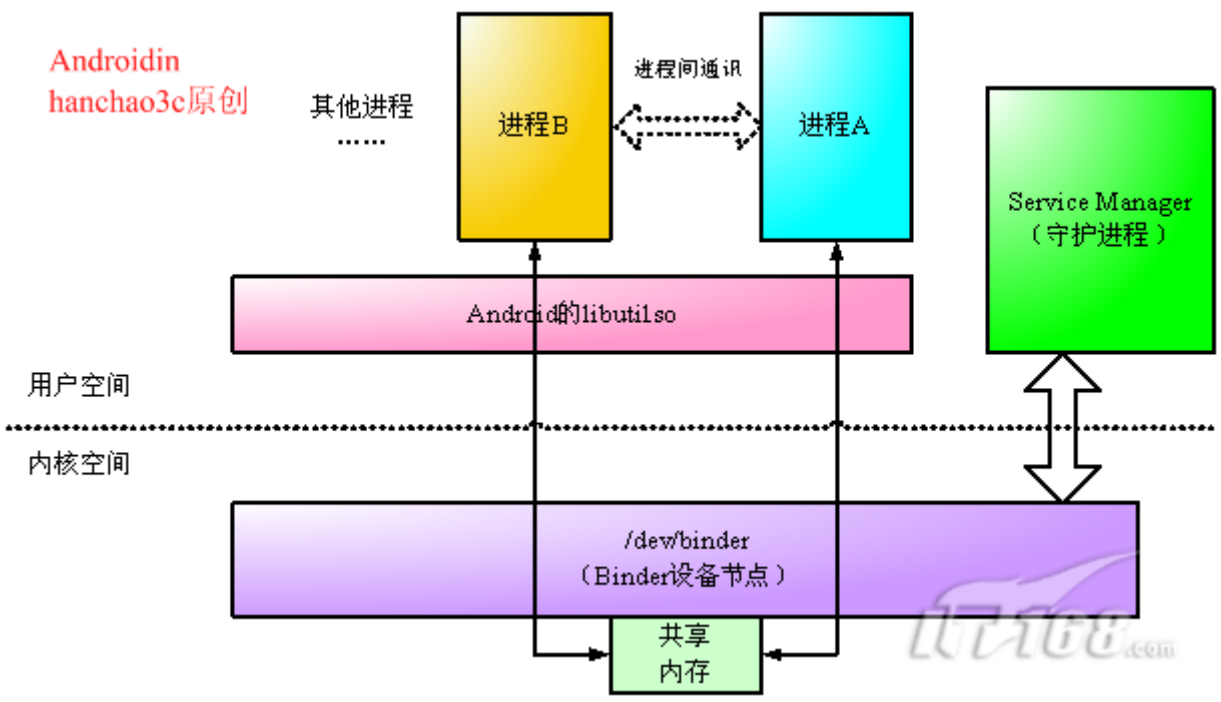
```
8.          return I##INTERFACE::descriptor;                          \
9.      }                                                             \
10.     sp I##INTERFACE::asInterface(const sp& obj)  \
11.     {                                                             \
12.         sp intr;                                  \
13.         if (obj != NULL) {                                        \
14.             intr = static_cast(                       \
15.                 obj->queryLocalInterface(                         \
16.                         I##INTERFACE::descriptor).get());         \
17.             if (intr == NULL) {                                   \
18.                 intr = new Bp##INTERFACE(obj);                    \
19.             }                                                     \
20.         }                                                         \
21.         return intr;                                              \
22.     }
```

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。



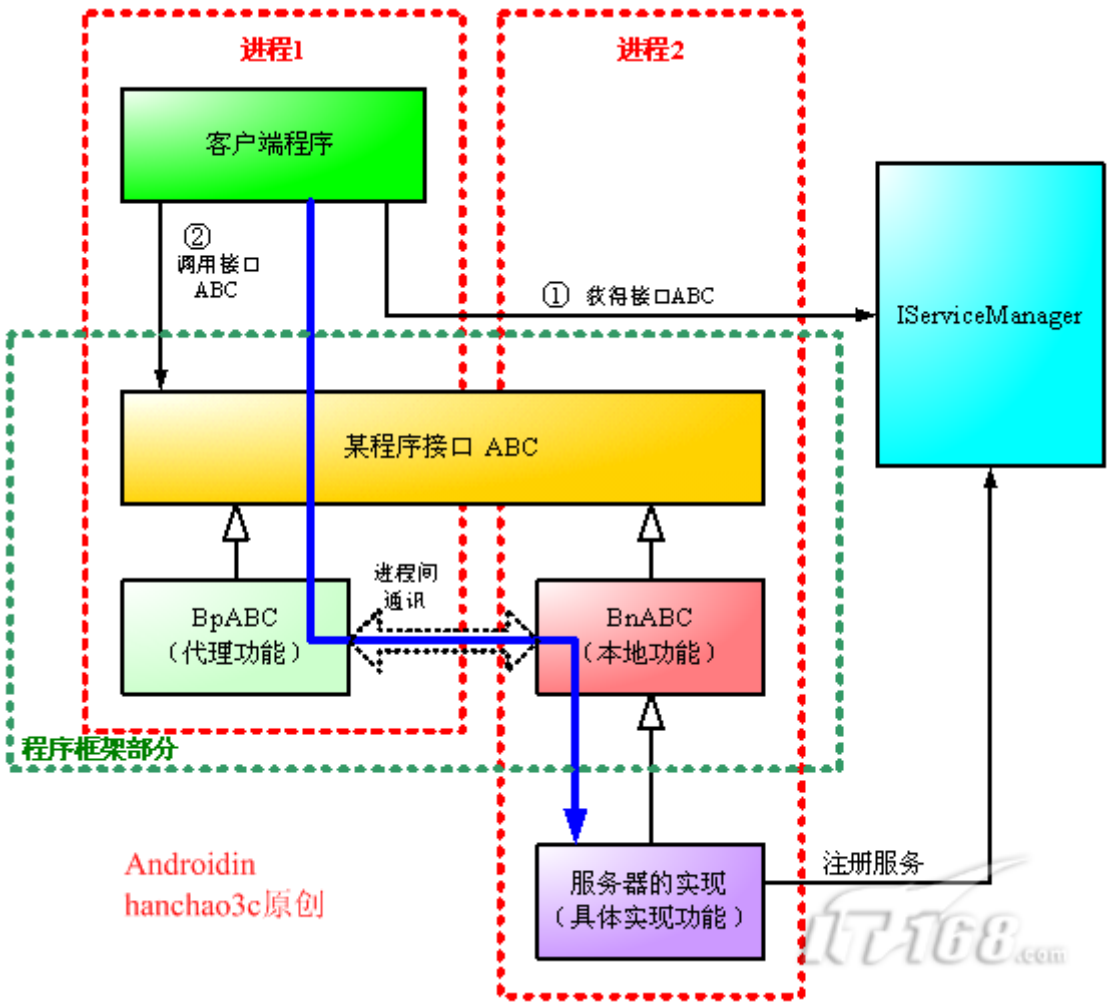
2.2 从应用程序的角度看 Binder

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：

Java 代码

```
1.  class IPermissionController : public IInterface
2.  {
3.  public:
4.      DECLARE_META_INTERFACE(PermissionController);
5.      virtual bool   checkPermission(const String16& permission,int32_t pid, int32_t uid) = 0;
6.      enum {
7.          CHECK_PERMISSION_TRANSACTION = IBinder::FIRST_CALL_TRANSACTION
8.      };
```

```
};
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。

Java 代码

```
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：

Java 代码

```
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);
```

```
13.             return NO_ERROR;
14.         } break;
15.         default:
16.             return BBinder:: BnTransact(code, data, reply, flags);
17.     }
18. }
```

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：

Java 代码

```
1.  mRemote()->transact()
2.      Process::self()
3.      IPCThreadState::self()->transact()
4.      writeTransactionData()
5.      waitForResponse()
6.      talkWithDriver()
7.      ioctl(fd, BINDER_WRITE_READ, &bwr)
```

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

o IBinder 接口

IBinder 接口是对跨进程的对象的抽象。普通对象在当前进程可以访问，如果希望对象能被其它进程访问，那就必须实现 IBinder 接口。IBinder 接口可以指向本地对象，也可以指向远程对象，调用者不需要关心指向的对象是本地的还是远程。

transact 是 IBinder 接口中一个比较重要的函数，它的函数原型如下：

```
1.  virtual status_t transact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0) = 0;
```

android 中的 IPC 的基本模型是基于客户/服务器(C/S)架构的。

客户端	请求通过内核模块中转	服务端

如果 IBinder 指向的是一个客户端代理，那 transact 只是把请求发送给服务器。服务端的 IBinder 的 transact 则提供了实际的服务。

o 客户端

BpBinder 是远程对象在当前进程的代理，它实现了 IBinder 接口。它的 transact 函数实现如下：

```
1.  status_t BpBinder::transact(
2.      uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
3.  {
4.      // Once a binder has died, it will never come back to life.
5.      if (mAlive) {
6.          status_t status = IPCThreadState::self()->transact(
7.              mHandle, code, data, reply, flags);
8.          if (status == DEAD_OBJECT) mAlive = 0;
9.          return status;
10.     }
11.
12.     return DEAD_OBJECT;
13. }
```

参数说明：

- code 是请求的 ID 号。
- data 是请求的参数。
- reply 是返回的结果。
- flags 一些额外的标识，如 FLAG_ONEWAY。通常为 0。

transact 只是简单的调用了 IPCThreadState::self()的 transact，在 IPCThreadState::transact 中：

```
1.  status_t IPCThreadState::transact(int32_t handle,
2.                                    uint32_t code, const Parcel& data,
3.                                    Parcel* reply, uint32_t flags)
4.  {
5.      status_t err = data.errorCheck();
6.
7.      flags |= TF_ACCEPT_FDS;
8.
9.      IF_LOG_TRANSACTIONS() {
10.         TextOutput::Bundle _b(alog);
11.         alog << "BC_TRANSACTION thr " << (void*)pthread_self() << " / hand "
12.             << handle << " / code " << TypeCode(code) << ": "
13.             << indent << data << dedent << endl;
14.     }
15.
16.     if (err == NO_ERROR) {
17.         LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
18.             (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
19.         err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
20.     }
21.
22.     if (err != NO_ERROR) {
23.         if (reply) reply->setError(err);
24.         return (mLastError = err);
25.     }
26.
27.     if ((flags & TF_ONE_WAY) == 0) {
28.         if (reply) {
29.             err = waitForResponse(reply);
30.         } else {
31.             Parcel fakeReply;
32.             err = waitForResponse(&fakeReply);
33.         }
34.
35.         IF_LOG_TRANSACTIONS() {
36.             TextOutput::Bundle _b(alog);
37.             alog << "BR_REPLY thr " << (void*)pthread_self() << " / hand "
38.                 << handle << ": ";
39.             if (reply) alog << indent << *reply << dedent << endl;
40.             else alog << "(none requested)" << endl;
```

```
41.         }
42.     } else {
43.         err = waitForResponse(NULL, NULL);
44.     }
45.
46.     return err;
47. }
48.
49. status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
50. {
51.     int32_t cmd;
52.     int32_t err;
53.
54.     while (1) {
55.         if ((err=talkWithDriver()) < NO_ERROR) break;
56.         err = mIn.errorCheck();
57.         if (err < NO_ERROR) break;
58.         if (mIn.dataAvail() == 0) continue;
59.
60.         cmd = mIn.readInt32();
61.
62.         IF_LOG_COMMANDS() {
63.             alog << "Processing waitForResponse Command: "
64.                 << getReturnString(cmd) << endl;
65.         }
66.
67.         switch (cmd) {
68.         case BR_TRANSACTION_COMPLETE:
69.             if (!reply && !acquireResult) goto finish;
70.             break;
71.
72.         case BR_DEAD_REPLY:
73.             err = DEAD_OBJECT;
74.             goto finish;
75.
76.         case BR_FAILED_REPLY:
77.             err = FAILED_TRANSACTION;
78.             goto finish;
79.
80.         case BR_ACQUIRE_RESULT:
81.             {
82.                 LOG_ASSERT(acquireResult != NULL, "Unexpected brACQUIRE_RESULT");
83.                 const int32_t result = mIn.readInt32();
84.                 if (!acquireResult) continue;
85.                 *acquireResult = result ? NO_ERROR : INVALID_OPERATION;
86.             }
87.             goto finish;
88.
89.         case BR_REPLY:
90.             {
91.                 binder_transaction_data tr;
92.                 err = mIn.read(&tr, sizeof(tr));
93.                 LOG_ASSERT(err == NO_ERROR, "Not enough command data for brREPLY");
94.                 if (err != NO_ERROR) goto finish;
95.
96.                 if (reply) {
97.                     if ((tr.flags & TF_STATUS_CODE) == 0) {
98.                         reply->ipcSetDataReference(
99.                             reinterpret_cast(tr.data.ptr.buffer),
100.                            tr.data_size,
101.                            reinterpret_cast(tr.data.ptr.offsets),
102.                            tr.offsets_size/sizeof(size_t),
103.                            freeBuffer, this);
104.                    } else {
105.                        err = *static_cast(tr.data.ptr.buffer);
106.                        freeBuffer(NULL,
107.                            reinterpret_cast(tr.data.ptr.buffer),
```

```
                    tr.data_size,
                    reinterpret_cast(tr.data.ptr.offsets),
                    tr.offsets_size/sizeof(size_t), this);
            }
        } else {
            freeBuffer(NULL,
                reinterpret_cast(tr.data.ptr.buffer),
                tr.data_size,
                reinterpret_cast(tr.data.ptr.offsets),
                tr.offsets_size/sizeof(size_t), this);
            continue;
        }
    }
    goto finish;

default:
    err = executeCommand(cmd);
    if (err != NO_ERROR) goto finish;
    break;
}
}

finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
    }

    return err;
}
```

这里 transact 把请求经内核模块发送了给服务端，服务端处理完请求之后，沿原路返回结果给调用者。这里也可以看出请求是同步操作，它会等待直到结果返回为止。

在 BpBinder 之上进行简单包装，我们可以得到与服务对象相同的接口，调用者无需要关心调用的对象是远程的还是本地的。拿 ServiceManager 来说：
(frameworks/base/libs/utils/IServiceManager.cpp)

```
class BpServiceManager : public BpInterface
{
public:
    BpServiceManager(const sp& impl)
        : BpInterface(impl)
    {
    }
...
    virtual status_t addService(const String16& name, const sp& service)
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        data.writeString16(name);
        data.writeStrongBinder(service);
        status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
        return err == NO_ERROR ? reply.readInt32() : err;
    }
...
};
```

BpServiceManager 实现了 IServiceManager 和 IBinder 两个接口，调用者可以把 BpServiceManager 的对象看作是一个 IServiceManager 对象或者 IBinder 对象。当调用者把 BpServiceManager 对象当作 IServiceManager 对象使 用时，所有的请求只是对 BpBinder::transact 的封装。这样的封装使得调用者不需要关心 IServiceManager 对象是本地的还是远 程的了。

客户通过 defaultServiceManager 函数来创建 BpServiceManager 对象：
(frameworks/base/libs/utils/IServiceManager.cpp)

```
1.  sp<IServiceManager> defaultServiceManager()
2.  {
3.      if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
4.
5.      {
6.          AutoMutex _l(gDefaultServiceManagerLock);
7.          if (gDefaultServiceManager == NULL) {
8.              gDefaultServiceManager = interface_cast<IServiceManager>(
9.                  ProcessState::self()->getContextObject(NULL));
10.         }
11.     }
12.
13.     return gDefaultServiceManager;
14. }
```

先通过 ProcessState::self()->getContextObject(NULL)创建一个 Binder 对象，然后通过 interface_cast 和 IMPLEMENT_META_INTERFACE(ServiceManager, “android.os.IServiceManager”)把 Binder 对象包装成 IServiceManager 对象。原理上等同于创建了一个 BpServiceManager 对象。

ProcessState::self()->getContextObject 调用 ProcessState::getStrongProxyForHandle 创建代理对象：

```
1.  sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
2.  {
3.      sp<IBinder> result;
4.
5.      AutoMutex _l(mLock);
6.
7.      handle_entry* e = lookupHandleLocked(handle);
8.
9.      if (e != NULL) {
10.         // We need to create a new BpBinder if there isn't currently one, OR we
11.         // are unable to acquire a weak reference on this current one.  See comment
12.         // in getWeakProxyForHandle() for more info about this.
13.         IBinder* b = e->binder;
14.         if (b == NULL || !e->refs->attemptIncWeak(this)) {
15.             b = new BpBinder(handle);
16.             e->binder = b;
17.             if (b) e->refs = b->getWeakRefs();
18.             result = b;
19.         } else {
20.             // This little bit of nastyness is to allow us to add a primary
21.             // reference to the remote proxy when this team doesn't have one
22.             // but another team is sending the handle to us.
23.             result.force_set(b);
24.             e->refs->decWeak(this);
25.         }
26.     }
27.
28.     return result;
29. }
```

如果 handle 为空，默认为 context_manager 对象，context_manager 实际上就是 ServiceManager。

o 服务端

服务端也要实现 IBinder 接口，BBinder 类对 IBinder 接口提供了部分默认实现，其中 transact 的实现如下：

```
1.  status_t BBinder::transact(
2.      uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
3.  {
4.      data.setDataPosition(0);
5.
6.      status_t err = NO_ERROR;
```

```
7.      switch (code) {
8.          case PING_TRANSACTION:
9.              reply->writeInt32(pingBinder());
10.             break;
11.         default:
12.             err = onTransact(code, data, reply, flags);
13.             break;
14.     }
15.
16.     if (reply != NULL) {
17.         reply->setDataPosition(0);
18.     }
19.
20.     return err;
21. }
```

PING_TRANSACTION 请求用来检查对象是否还存在，这里简单的把 pingBinder 的返回值返回给调用者。其它的请求交给 onTransact 处理。onTransact 是 BBinder 里声明的一个 protected 类型的虚函数，这个要求它的子类去实现。比如 CameraService 里的实现如下：

```
1.  status_t CameraService::onTransact(
2.      uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
3.  {
4.      // permission checks...
5.      switch (code) {
6.          case BnCameraService::CONNECT:
7.              IPCThreadState* ipc = IPCThreadState::self();
8.              const int pid = ipc->getCallingPid();
9.              const int self_pid = getpid();
10.             if (pid != self_pid) {
11.                 // we're called from a different process, do the real check
12.                 if (!checkCallingPermission(
13.                         String16("android.permission.CAMERA")))
14.                 {
15.                     const int uid = ipc->getCallingUid();
16.                     LOGE("Permission Denial: "
17.                             "can't use the camera pid=%d, uid=%d", pid, uid);
18.                     return PERMISSION_DENIED;
19.                 }
20.             }
21.             break;
22.     }
23.
24.     status_t err = BnCameraService::onTransact(code, data, reply, flags);
25.
26.     LOGD("+++ onTransact err %d code %d", err, code);
27.
28.     if (err == UNKNOWN_TRANSACTION || err == PERMISSION_DENIED) {
29.         // the 'service' command interrogates this binder for its name, and then supplies it
30.         // even for the debugging commands.  that means we need to check for it here, using
31.         // ISurfaceComposer (since we delegated the INTERFACE_TRANSACTION handling to
32.         // BnSurfaceComposer before falling through to this code).
33.
34.         LOGD("+++ onTransact code %d", code);
35.
36.         CHECK_INTERFACE(ICameraService, data, reply);
37.
38.         switch(code) {
39.         case 1000:
40.         {
41.             if (gWeakHeap != 0) {
42.                 sp h = gWeakHeap.promote();
43.                 IMemoryHeap *p = gWeakHeap.unsafe_get();
44.                 LOGD("CHECKING WEAK REFERENCE %p (%p)", h.get(), p);
45.                 if (h != 0)
```

```
46.                 h->printRefs();
47.             bool attempt_to_delete = data.readInt32() == 1;
48.             if (attempt_to_delete) {
49.                 // NOT SAFE!
50.                 LOGD("DELETING WEAK REFERENCE %p (%p)", h.get(), p);
51.                 if (p) delete p;
52.             }
53.             return NO_ERROR;
54.         }
55.     }
56.     break;
57.     default:
58.         break;
59.     }
60. }
61. return err;
62. }
```

由此可见，服务端的 onTransact 是一个请求分发函数，它根据请求码(code)做相应的处理。

o 消息循环

服务端(任何进程都可以作为服务端)有一个线程监听来自客户端的请求，并循环处理这些请求。

如果在主线程中处理请求，可以直接调用下面的函数：

```
1. IPCThreadState::self()->joinThreadPool(mIsMain);
```

如果想在非主线程中处理请求，可以按下列方式：

```
1. sp
2. pan>= ProcessState::self();
3. if (proc->supportsProcesses()) {
4.     LOGV("App process: starting thread pool.\n");
5.     proc->startThreadPool();
6. }
```

startThreadPool 的实现原理：

```
1. void ProcessState::startThreadPool()
2. {
3.     AutoMutex _l(mLock);
4.     if (!mThreadPoolStarted) {
5.         mThreadPoolStarted = true;
6.         spawnPooledThread(true);
7.     }
8. }
9.
10. void ProcessState::spawnPooledThread(bool isMain)
11. {
12.     if (mThreadPoolStarted) {
13.         int32_t s = android_atomic_add(1, &mThreadPoolSeq);
14.         char buf[32];
15.         sprintf(buf, "Binder Thread #%d", s);
16.         LOGV("Spawning new pooled thread, name=%s\n", buf);
17.         sp
18.  t = new PoolThread(isMain);
19.         t->run(buf);
20.     }
21. }
```

这里创建了 PoolThread 的对象，实现上就是创建了一个线程。所有的线程类都要实现 threadLoop 虚函数。PoolThread 的 threadLoop 的实现如下：

```
1.  virtual bool threadLoop()
2.  {
3.      IPCThreadState::self()->joinThreadPool(mIsMain);
4.      return false;
5.  }
```

上述代码，简而言之就是创建了一个线程，然后在线程里调用 IPCThreadState::self()->joinThreadPool 函数。

下面再看 joinThreadPool 的实现：

```
1.  do
2.  {
3.  ...
4.          result = talkWithDriver();
5.          if (result >= NO_ERROR) {
6.              size_t IN = mIn.dataAvail();
7.              if (IN < sizeof(int32_t)) continue;
8.              cmd = mIn.readInt32();
9.              IF_LOG_COMMANDS() {
10.                 alog << "Processing top-level Command: "
11.                     << getReturnString(cmd) << endl;
12.             }
13.             result = executeCommand(cmd);
14.         }
15. ...
16. while(...);
```

这个函数在循环中重复执行下列动作：

1. talkWithDriver 通过 ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr)读取请求和写回结果。
2. executeCommand 执行相应的请求

在 IPCThreadState::executeCommand(int32_t cmd)函数中：

1. 对于控制对象生命周期的请求，像 BR_ACQUIRE/BR_RELEASE 直接做了处理。
2. 对于 BR_TRANSACTION 请求，它调用被请求对象的 transact 函数。

按下列方式调用实际的对象：

```
1.  if (tr.target.ptr) {
2.      sp<BBinder> b((BBinder*)tr.cookie);
3.      const status_t error = b->transact(tr.code, buffer, &reply, 0);
4.      if (error < NO_ERROR) reply.setError(error);
5.
6.  } else {
7.      const status_t error = the_context_object->transact(tr.code, buffer, &reply, 0);
8.      if (error < NO_ERROR) reply.setError(error);
9.  }
```

如果 tr.target.ptr 不为空，就把 tr.cookie 转换成一个 Binder 对象，并调用它的 transact 函数。如果没有目标对象， 就调用 the_context_object 对象的 transact 函数。奇怪的是，根本没有谁对 the_context_object 进行初始 化，the_context_object 是空指针。原因是 context_mgr 的请求发给了 ServiceManager，所以根本不会走到 else 语句里来。

- **内核模块**

android 使用了一个内核模块 binder 来中转各个进程之间的消息。模块源代码放在 binder.c 里，它是一个字符驱动程序，主要通过 binder_ioctl 与用户空间的进程交换数据。其中 BINDER_WRITE_READ 用来读写数据，数据包中有一个 cmd 域用于区分不同的请求：

1. binder_thread_write 用于发送请求或返回结果。
2. binder_thread_read 用于读取结果。

从 binder_thread_write 中调用 binder_transaction 中转请求和返回结果，binder_transaction 的实现如下：

对请求的处理：

1. 通过对象的 handle 找到对象所在的进程，如果 handle 为空就认为对象是 context_mgr，把请求发给 context_mgr 所在的进程。
2. 把请求中所有的 binder 对象全部放到一个 RB 树中。
3. 把请求放到目标进程的队列中，等待目标进程读取。

如何成为 context_mgr 呢？内核模块提供了 BINDER_SET_CONTEXT_MGR 调用:

```
1.  static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
2.  {
3.      ...
4.      case BINDER_SET_CONTEXT_MGR:
5.          if (binder_context_mgr_node != NULL) {
6.              printk(KERN_ERR "binder: BINDER_SET_CONTEXT_MGR already set\n");
7.              ret = -EBUSY;
8.              goto err;
9.          }
10.         if (binder_context_mgr_uid != -1) {
11.             if (binder_context_mgr_uid != current->euid) {
12.                 printk(KERN_ERR "binder: BINDER_SET_"
13.                        "CONTEXT_MGR bad uid %d != %d\n",
14.                        current->euid,
15.                        binder_context_mgr_uid);
16.                 ret = -EPERM;
17.                 goto err;
18.             }
19.         } else
20.             binder_context_mgr_uid = current->euid;
21.         binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
22.         if (binder_context_mgr_node == NULL) {
23.             ret = -ENOMEM;
24.             goto err;
25.         }
26.         binder_context_mgr_node->local_weak_refs++;
27.         binder_context_mgr_node->local_strong_refs++;
28.         binder_context_mgr_node->has_strong_ref = 1;
29.         binder_context_mgr_node->has_weak_ref = 1;
30.         break;
```

ServiceManager（frameworks/base/cmds/servicemanager)通过下列方式成为了 context_mgr 进程：

```
1.  int binder_become_context_manager(struct binder_state *bs)
2.  {
3.      return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
4.  }
5.
6.  int main(int argc, char **argv)
7.  {
8.      struct binder_state *bs;
9.      void *svcmgr = BINDER_SERVICE_MANAGER;
10.
11.     bs = binder_open(128*1024);
12.
13.     if (binder_become_context_manager(bs)) {
14.         LOGE("cannot become context manager (%s)\n", strerror(errno));
15.         return -1;
16.     }
17.
```

```
18.     svcmgr_handle = svcmgr;
19.     binder_loop(bs, svcmgr_handler);
20.     return 0;
21. }
```

- **如何得到服务对象的 handle**

1. 服务提供者通过 defaultServiceManager 得到 ServiceManager 对象，然后调用 addService 向服务管理器注册。
2. 服务使用者通过 defaultServiceManager 得到 ServiceManager 对象，然后调用 getService 通过服务名称查找到服务对象的 handle。

- **如何通过服务对象的 handle 找到服务所在的进程**

0 表示服务管理器的 handle，getService 可以查找到系统服务的 handle。这个 handle 只是代表了服务对象，内核模块是如何通过 handle 找到服务所在的进程的呢？

1. 对于 ServiceManager: ServiceManager 调用了 binder_become_context_manager 使用自己成为 context_mgr，所有 handle 为 0 的请求都会被转发给 ServiceManager。
2. 对于系统服务和应用程序的 Listener，在第一次请求内核模块时(比如调用 addService)，内核模块在一个 RB 树中建立了服务对象和进程的对应关系。

```
1.  off_end = (void *)offp + tr->offsets_size;
2.  for (; offp < off_end; offp++) {
3.      struct flat_binder_object *fp;
4.      if (*offp > t->buffer->data_size - sizeof(*fp)) {
5.          binder_user_error("binder: %d:%d got transaction with "
6.              "invalid offset, %d\n",
7.              proc->pid, thread->pid, *offp);
8.          return_error = BR_FAILED_REPLY;
9.          goto err_bad_offset;
10.     }
11.     fp = (struct flat_binder_object *)(t->buffer->data + *offp);
12.     switch (fp->type) {
13.     case BINDER_TYPE_BINDER:
14.     case BINDER_TYPE_WEAK_BINDER: {
15.         struct binder_ref *ref;
16.         struct binder_node *node = binder_get_node(proc, fp->binder);
17.         if (node == NULL) {
18.             node = binder_new_node(proc, fp->binder, fp->cookie);
19.             if (node == NULL) {
20.                 return_error = BR_FAILED_REPLY;
21.                 goto err_binder_new_node_failed;
22.             }
23.             node->min_priority = fp->flags & FLAT_BINDER_FLAG_PRIORITY_MASK;
24.             node->accept_fds = !!(fp->flags & FLAT_BINDER_FLAG_ACCEPTS_FDS);
25.         }
26.         if (fp->cookie != node->cookie) {
27.             binder_user_error("binder: %d:%d sending u%p "
28.                 "node %d, cookie mismatch %p != %p\n",
29.                 proc->pid, thread->pid,
30.                 fp->binder, node->debug_id,
31.                 fp->cookie, node->cookie);
32.             goto err_binder_get_ref_for_node_failed;
33.         }
34.         ref = binder_get_ref_for_node(target_proc, node);
35.         if (ref == NULL) {
36.             return_error = BR_FAILED_REPLY;
37.             goto err_binder_get_ref_for_node_failed;
38.         }
39.         if (fp->type == BINDER_TYPE_BINDER)
40.             fp->type = BINDER_TYPE_HANDLE;
41.         else
42.             fp->type = BINDER_TYPE_WEAK_HANDLE;
43.         fp->handle = ref->desc;
44.         binder_inc_ref(ref, fp->type == BINDER_TYPE_HANDLE, &thread->todo);
```

```
45.             if (binder_debug_mask & BINDER_DEBUG_TRANSACTION)
46.                 printk(KERN_INFO "        node %d u%p -> ref %d desc %d\n",
47.                        node->debug_id, node->ptr, ref->debug_id, ref->desc);
48.         } break;
```

3. 请求服务时，内核先通过 handle 找到对应的进程，然后把请求放到服务进程的队列中。

o C 调用 JAVA

前面我们分析的是 C 代码的处理。对于 JAVA 代码，JAVA 调用 C 的函数通过 JNI 调用即可。从内核时读取请求是在 C 代码(executeCommand)里进行了，那如何在 C 代码中调用那些用 JAVA 实现的服务呢？

android_os_Binder_init 里的 JavaBBinder 对 Java 里的 Binder 对象进行包装。

JavaBBinder::onTransact 调用 Java 里的 execTransact 函数：

```
1.          jboolean res = env->CallBooleanMethod(mObject, gBinderOffsets.mExecTransact,
2.              code, (int32_t)&data, (int32_t)reply, flags);
3.          jthrowable excep = env->ExceptionOccurred();
4.          if (excep) {
5.              report_exception(env, excep,
6.                  "*** Uncaught remote exception!  "
7.                  "(Exceptions are not yet supported across processes.)");
8.              res = JNI_FALSE;
9.
10.             /* clean up JNI local ref -- we don't return to Java code */
11.             env->DeleteLocalRef(excep);
12.         }
```

o 广播消息

binder 不提供广播消息，不过可以 ActivityManagerService 服务来实现广播。(frameworks/base/core/java/android/app/ActivityManagerNative.java)

接收广播消息需要实现接口 BroadcastReceiver，然后调用 ActivityManagerProxy::registerReceiver 注册。

触发广播调用 ActivityManagerProxy::broadcastIntent。(应用程序并不直接调用它，而是调用 Context 对它的包装)

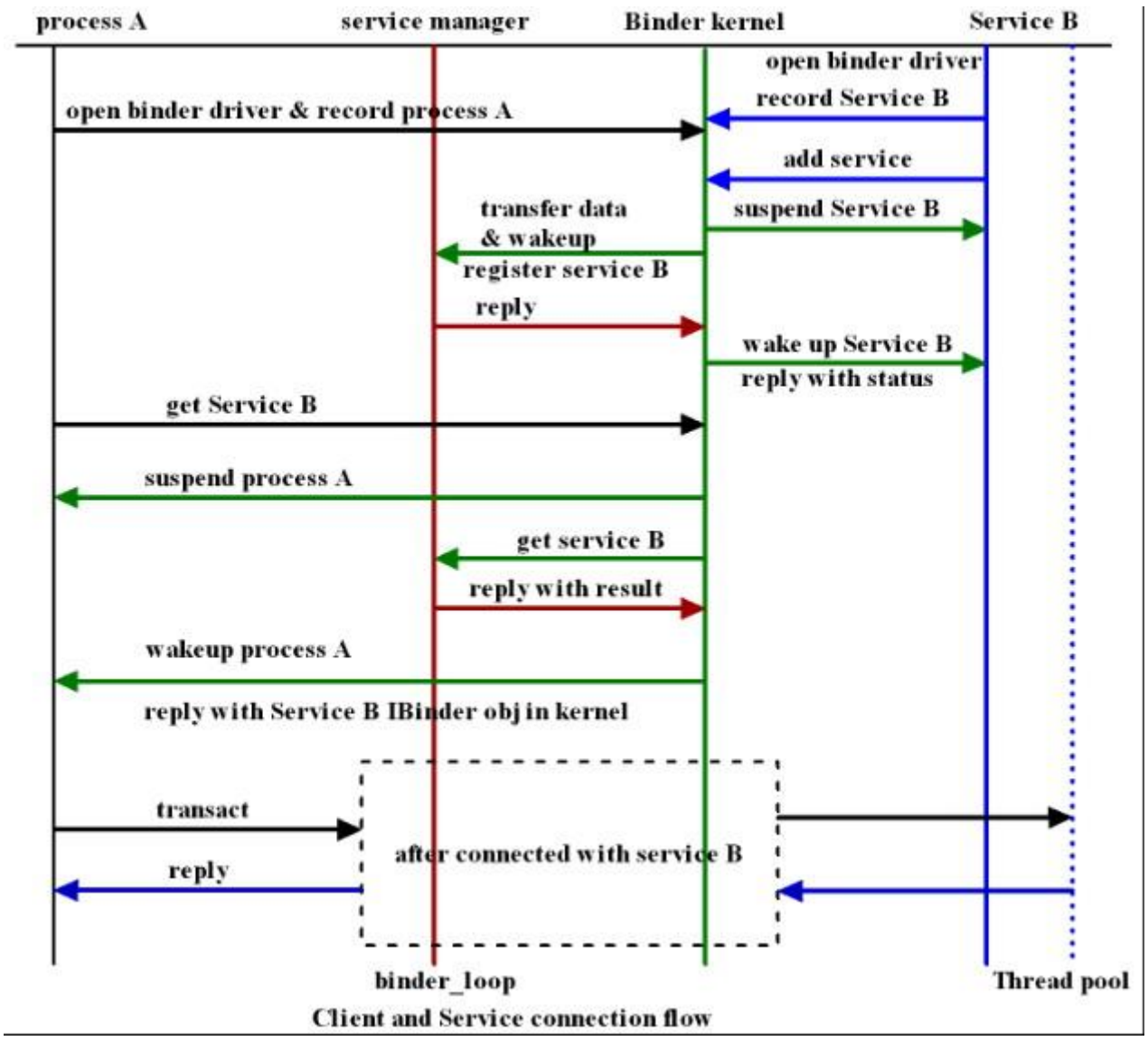
Binder 通信简介：

Linux 系统中进程间通信的方式有:socket, named pipe,message queque, signal,share memory。Java 系统中的进程间通信方式有 socket, named pipe 等，android 应用程序理所当然可以应用 JAVA 的 IPC 机制实现进程间的通信，但我查看 android 的源码，在同一终端上的应用软件的通 信几乎看不到这些 IPC 通信方式，取而代之的是 Binder 通信。Google 为什么要采用这种方式呢，这取决于 Binder 通信方式的高效率。 Binder 通信是通过 linux 的 binder driver 来实现的，Binder 通信操作类似线程迁移(thread migration)，两个进程间 IPC 看起来就象是一个进程进入另一个进程执行代码然后带着执行的结果返回。Binder 的用户空间为每一个进程维护着 一个可用的线程池，线程池用于处理到来的 IPC 以及执行进程本地消息，Binder 通信是同步而不是异步。

Android 中的 Binder 通信是基于 Service 与 Client 的，所有需要 IBinder 通信的进程都必须创建一个 IBinder 接口，系统中有一个进程管理所有 的 system service,Android 不允许用户添加非授权的 System service,当然现在源码开发了，我们可以修改一些代码来实现添加底层 system Service 的目的。对用户程序来说，我们也要创建 server,或者 Service 用于进程间通信，这里有一个 ActivityManagerService 管理 JAVA 应用层所有的 service 创建与连接(connect),disconnect,所有的 Activity 也是通过这个 service 来启动，加载的。ActivityManagerService 也是加载在 Systems Servcie 中的。

Android 虚拟机启动之前系统会先启动 service Manager 进程，service Manager 打开 binder 驱动，并通知 binder kernel 驱动程序这个进程将作为 System Service Manager，然后该进程将进入一个循环，等待处理来自其他进程的数据。用户创建一个 System service 后，通过 defaultServiceManager 得到一个远程 ServiceManager 的接口，通过这个接口我们可以调用 addService 函数将 System service 添加到 Service Manager 进程中，然后 client 可以通过 getService 获取到需要连接的目的 Service 的 IBinder 对象，这个 IBinder 是 Service 的 BBinder 在 binder kernel 的一个参考，所以 service IBinder 在 binder kernel 中不会存在相同的两个 IBinder 对象，每一个 Client 进程同样需要打开 Binder 驱动程序。对用户程序而言，我们获得这个对象就可 以通过 binder kernel 访问 service 对象中的方法。Client 与 Service 在不同的进程中，通过这种方式实现了类似线程间的迁移的通信方式，对用户程序 而言当调用 Service 返回的 IBinder 接口后，访问 Service 中的方法就如同调用自己的函数。

下图为 client 与 Service 建立连接的示意图



首先从 ServiceManager 注册过程来逐步分析上述过程是如何实现的。

ServiceMananger 进程注册过程源码分析：

Service Manager Process（Service_manager.c）：

Service_manager 为其他进程的 Service 提供管理，这个服务程序必须在 Android Runtime 起来之前运行，否则 Android JAVA Vm ActivityManagerService 无法注册。

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

       bs = binder_open(128*1024); //打开/dev/binder  驱动

       if (binder_become_context_manager(bs)) {//注册为 service manager in binder kernel
      LOGE("cannot become context manager (%s)\n", strerror(errno));
      return -1;
   }
   svcmgr_handle = svcmgr;
   binder_loop(bs, svcmgr_handler);
   return 0;
}
```

首 先打开 binder 的驱动程序然后通过 binder_become_context_manager 函数调用 ioctl 告诉 Binder Kernel 驱动程序这是一个服务管理进程，然后调用 binder_loop 等待来自其他进程的数据。BINDER_SERVICE_MANAGER 是服 务管理进程的句柄，它的定义是：

```
#define BINDER_SERVICE_MANAGER ((void*) 0)
```

如果客户端进程获取 Service 时所使用的句柄与此不符，Service Manager 将不接受 Client 的请求。客户端如何设置这个句柄在下面会介绍。

CameraSerivce 服务的注册(Main_mediaservice.c）

```
int main(int argc, char** argv)
{
   sp<ProcessState> proc(ProcessState::self());
   sp<IServiceManager> sm = defaultServiceManager();
   LOGI("ServiceManager: %p", sm.get());
   AudioFlinger::instantiate();            //Audio  服务
   MediaPlayerService::instantiate();      //mediaPlayer 服务
   CameraService::instantiate();           //Camera  服务
```

```
    ProcessState::self()->startThreadPool(); //为进程开启缓冲池
    IPCThreadState::self()->joinThreadPool(); //将进程加入到缓冲池
}

        CameraService.cpp
void CameraService::instantiate() {
    defaultServiceManager()->addService(
            String16("media.camera"), new CameraService());
}
```

创建 CameraService 服务对象并添加到 ServiceManager 进程中。

client 获取 remote IServiceManager IBinder 接口：

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }
    return gDefaultServiceManager;
}
```

任 何一个进程在第一次调用 defaultServiceManager 的时候 gDefaultServiceManager 值为 Null，所以该进程会通过 ProcessState::self 得到 ProcessState 实例。ProcessState 将打开 Binder 驱动。

ProcessState.cpp

```
sp<ProcessState> ProcessState::self()
{
    if (gProcess != NULL) return gProcess;

    AutoMutex _l(gProcessMutex);
    if (gProcess == NULL) gProcess = new ProcessState;
    return gProcess;
}

        ProcessState::ProcessState()
: mDriverFD(open_driver()) //打开/dev/binder 驱动
.....
{
}

        sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
{
    if (supportsProcesses()) {
        return getStrongProxyForHandle(0);
    } else {
        return getContextObject(String16("default"), caller);
    }
}
```

Android 是支持 Binder 驱动的所以程序会调用 getStrongProxyForHandle。这里 handle 为 0，正好与 Service_manager 中的 BINDER_SERVICE_MANAGER 一致。

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;
    AutoMutex _l(mLock);
    handle_entry* e = lookupHandleLocked(handle);

            if (e != NULL) {
        // We need to create a new BpBinder if there isn't currently one, OR we
        // are unable to acquire a weak reference on this current one. See comment
        // in getWeakProxyForHandle() for more info about this.
        IBinder* b = e->binder; //第一次调用该函数 b 为 Null
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
            b = new BpBinder(handle);
```

```
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        } else {
            // This little bit of nastyness is to allow us to add a primary
            // reference to the remote proxy when this team doesn't have one
            // but another team is sending the handle to us.
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }
    return result;
}
```

第一次调用的时候 b 为 Null 所以会为 b 生成一 BpBinder 对象：

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
    LOGV("Creating BpBinder %p handle %d\n", this, mHandle);

            extendObjectLifetime(OBJECT_LIFETIME_WEAK);
    IPCThreadState::self()->incWeakHandle(handle);
}

        void IPCThreadState::incWeakHandle(int32_t handle)
{
    LOG_REMOTEREFS("IPCThreadState::incWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_INCREFS);
    mOut.writeInt32(handle);
}
```

getContextObject 返回了一个 BpBinder 对象。

```
interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));

        template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

将这个宏扩展后最终得到的是：

```
sp<IServiceManager> IServiceManager::asInterface(const sp<IBinder>& obj)
    {
        sp<IServiceManager> intr;
        if (obj != NULL) {
            intr = static_cast<IServiceManager*>(
                obj->queryLocalInterface(
                        IServiceManager::descriptor).get());
            if (intr == NULL) {
                intr = new BpServiceManager(obj);
            }
        }
        return intr;
}
```

返回一个 BpServiceManager 对象,这里 obj 就是前面我们创建的 BpBInder 对象。

client 获取 Service 的远程 IBinder 接口

以 CameraService 为例(camera.cpp):

```
const sp<ICameraService>& Camera::getCameraService()
{
    Mutex::Autolock _l(mLock);
    if (mCameraService.get() == 0) {
        sp<IServiceManager> sm = defaultServiceManager();
        sp<IBinder> binder;
        do {
            binder = sm->getService(String16("media.camera"));
```

```
            if (binder != 0)
                break;
            LOGW("CameraService not published, waiting...");
            usleep(500000); // 0.5 s
        } while(true);
        if (mDeathNotifier == NULL) {
            mDeathNotifier = new DeathNotifier();
        }
        binder->linkToDeath(mDeathNotifier);
        mCameraService = interface_cast<ICameraService>(binder);
    }
    LOGE_IF(mCameraService==0, "no CameraService!?");
    return mCameraService;
}
```

由前面的分析可知 sm 是 BpCameraService 对象：//应该为 BpServiceManager 对象

```
    virtual sp<IBinder> getService(const String16& name) const
    {
        unsigned n;
        for (n = 0; n < 5; n++){
            sp<IBinder> svc = checkService(name);
            if (svc != NULL) return svc;
            LOGI("Waiting for sevice %s...\n", String8(name).string());
            sleep(1);
        }
        return NULL;
    }
    virtual sp<IBinder> checkService( const String16& name) const
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        data.writeString16(name);
        remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);
        return reply.readStrongBinder();
}
```

这里的 remote 就是我们前面得到 BpBinder 对象。所以 checkService 将调用 BpBinder 中的 transact 函数：

```
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;
}
```

mHandle 为 0，BpBinder 继续往下调用 IPCThreadState:transact 函数将数据发给与 mHandle 相关联的 Service Manager Process。

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
   ..............................................................
    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        if (reply) {
            err = waitForResponse(reply);
```

```
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
       ..............................

    return err;
}
```

通过 writeTransactionData 构造要发送的数据

```
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
    int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
{
    binder_transaction_data tr;

        tr.target.handle = handle; //这个 handle 将传递到 service_manager
    tr.code = code;
    tr.flags = bindrFlags;
。。。。。。。。。。。。。。。
}
```

waitForResponse 将调用 talkWithDriver 与对 Binder kernel 进行读写操作。当 Binder kernel 接收到数据后，service_mananger 线程的 ThreadPool 就会启动，service_manager 查找到 CameraService 服务后调用 binder_send_reply，将返回的数据写入 Binder kernel,Binder kernel。

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    int32_t cmd;
    int32_t err;

        while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;

.....................................................
}
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
   .....................................................
#if defined(HAVE_ANDROID_OS)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        else
            err = -errno;
#else
        err = INVALID_OPERATION;
#endif
.....................................................
}
```

通过上面的 ioctl 系统函数中 BINDER_WRITE_READ 对 binder kernel 进行读写。

Client A 与 Binder kernel 通信:

```
kernel\drivers\android\Binder.c）
static int binder_open(struct inode *nodp, struct file *filp)
{
struct binder_proc *proc;

        if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
    printk(KERN_INFO "binder_open: %d:%d\n", current->group_leader->pid, current->pid);

        proc = kzalloc(sizeof(*proc), GFP_KERNEL);
if (proc == NULL)
    return -ENOMEM;
get_task_struct(current);
proc->tsk = current;          //保存打开/dev/binder 驱动的当前进程任务数据结构
INIT_LIST_HEAD(&proc->todo);
init_waitqueue_head(&proc->wait);
proc->default_priority = task_nice(current);
mutex_lock(&binder_lock);
binder_stats.obj_created[BINDER_STAT_PROC]++;
```

```
hlist_add_head(&proc->proc_node, &binder_procs);
proc->pid = current->group_leader->pid;
INIT_LIST_HEAD(&proc->delivered_death);
filp->private_data = proc;
mutex_unlock(&binder_lock);

        if (binder_proc_dir_entry_proc) {
    char strbuf[11];
    snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
    create_proc_read_entry(strbuf, S_IRUGO, binder_proc_dir_entry_proc, binder_read_proc_proc, proc); //为当前进程创建一个
process 入口结构信息
}
return 0;
}
从 这里可以知道每一个打开/dev/binder 的进程的信息都保存在 binder kernel 中，因而当一个进程调用 ioctl 与 kernel binder 通信
时，binder kernel 就能查询到调用进程的信息。BINDER_WRITE_READ 是调用 ioctl 进程与 Binder kernel 通信一个非常重要的
command。大家可以看到在 IPCThreadState 中的 transact 函数这个函数中 call talkWithDriver 发送的 command 就是
BINDER_WRITE_READ。
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
int ret;
struct binder_proc *proc = filp->private_data;
struct binder_thread *thread;
unsigned int size = _IOC_SIZE(cmd);
void __user *ubuf = (void __user *)arg;

        /*printk(KERN_INFO "binder_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);*/
     //将调用 ioctl 的进程挂起  caller 将挂起直到  service  返回
ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
if (ret)
    return ret;

        mutex_lock(&binder_lock);
thread = binder_get_thread(proc);//根据当 caller 进程消息获取该进程线程池数据结构
if (thread == NULL) {
    ret = -ENOMEM;
    goto err;
}

        switch (cmd) {
case BINDER_WRITE_READ: { //IPcThreadState 中 talkWithDriver 设置 ioctl 的 CMD
    struct binder_write_read bwr;
    if (size != sizeof(struct binder_write_read)) {
     ret = -EINVAL;
     goto err;
    }
    if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
     ret = -EFAULT;
     goto err;
    }
    if (binder_debug_mask & BINDER_DEBUG_READ_WRITE)
     printk(KERN_INFO "binder: %d:%d write %ld at %08lx, read %ld at %08lx\n",
           proc->pid, thread->pid, bwr.write_size, bwr.write_buffer, bwr.read_size, bwr.read_buffer);
    if (bwr.write_size > 0) {
     ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer, bwr.write_size, &bwr.write_consumed);
     if (ret < 0) {
      bwr.read_consumed = 0;
      if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
       ret = -EFAULT;
      goto err;
     }
    }
    if (bwr.read_size > 0) {//数据写入到 caller process。
     ret = binder_thread_read(proc, thread, (void __user *)bwr.read_buffer, bwr.read_size, &bwr.read_consumed, filp->f_flags &
O_NONBLOCK);
     if (!list_empty(&proc->todo))
      wake_up_interruptible(&proc->wait); //恢复挂起的 caller 进程
```

```
    if (ret < 0) {
     if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
      ret = -EFAULT;
     goto err;
    }
   }
    .....................................
}

      Int binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,void __user *buffer, int size, signed long *consumed)
{
uint32_t cmd;
void __user *ptr = buffer + *consumed;
void __user *end = buffer + size;

      while (ptr < end && thread->return_error == BR_OK) {
   if (get_user(cmd, (uint32_t __user *)ptr))//从 user 空间获取 cmd 数据到内核空间
    return -EFAULT;
   ptr += sizeof(uint32_t);
   if (_IOC_NR(cmd) < ARRAY_SIZE(binder_stats.bc)) {
    binder_stats.bc[_IOC_NR(cmd)]++;
    proc->stats.bc[_IOC_NR(cmd)]++;
    thread->stats.bc[_IOC_NR(cmd)]++;
   }
   switch (cmd) {
   case BC_INCREFS:
.....................................
       case BC_TRANSACTION: //IPCThreadState 通过 writeTransactionData 设置该 cmd
   case BC_REPLY: {
    struct binder_transaction_data tr;

          if (copy_from_user(&tr, ptr, sizeof(tr)))
     return -EFAULT;
    ptr += sizeof(tr);
    binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
    break;
   }
.....................................
}

      static void
binder_transaction(struct binder_proc *proc, struct binder_thread *thread,
struct binder_transaction_data *tr, int reply)
{
     .....................................
    if (reply) // cmd != BC_REPLY  不走这个 case
{
        .....................................
   }
   else
{
   if (tr->target.handle) { //对于 service_manager 来说这个条件不满足(handle == 0)
    .....................................
    }
   } else {//这一段我们获取到了 service_mananger process  注册在 binder kernle 的进程信息
target_node = binder_context_mgr_node; //BINDER_SET_CONTEXT_MGR  注册了 service
    if (target_node == NULL) {            //manager
     return_error = BR_DEAD_REPLY;
     goto err_no_context_mgr_node;
    }
   }
   e->to_node = target_node->debug_id;
   target_proc = target_node->proc; //得到目标进程 service_mananger  的结构
   if (target_proc == NULL) {
    return_error = BR_DEAD_REPLY;
    goto err_dead_binder;
   }
```

```
    .....................
}
if (target_thread) {
    e->to_thread = target_thread->pid;
    target_list = &target_thread->todo;
    target_wait = &target_thread->wait; //得到 service manager 挂起的线程
} else {
    target_list = &target_proc->todo;
    target_wait = &target_proc->wait;
}
.............................................
case BINDER_TYPE_BINDER:
case BINDER_TYPE_WEAK_BINDER: {
     ...........................
     ref = binder_get_ref_for_node(target_proc, node); //在 Binder kernel 中创建
     ...........................                       //查找到的 service 参考
    } break;


        ..............................................
if (target_wait)
       wake_up_interruptible(target_wait);    //唤醒挂起的线程  处理 caller process 请求
.............................................//处理命令可以看 svcmgr_handler
}
```

到这里我们已经通过 getService 连接到 service manager 进程了，service manager 进程得到请求后，如果他的状态是挂起的话，将被唤醒。现在我们来看一下 service manager 中的 binder_loop 函数。

```
Service_manager.c
void binder_loop(struct binder_state *bs, binder_handler func)
{
    ...................................
    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); //如果没有要处理的请求进程将挂起
        if (res < 0) {
            LOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
            break;
        }
res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);//这里 func 就是
       ....................................              //svcmgr_handler
    }
}
```

接收到数据处理的请求，这里进行解析并调用前面注册的回调函数查找 caller 请求的 service

```
int binder_parse(struct binder_state *bs, struct binder_io *bio,
                 uint32_t *ptr, uint32_t size, binder_handler func)
{
       ....................................
      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);      //找到 caller 请求的 service
```

```
        binder_send_reply(bs, &reply, txn->data, res);//将找到的 service 返回给 caller
            }
            ptr += sizeof(*txn) / sizeof(uint32_t);
            break;
         ........
        }

       }
void binder_send_reply(struct binder_state *bs,
                       struct binder_io *reply,
                       void *buffer_to_free,
                       int status)
{
    struct {
        uint32_t cmd_free;
        void *buffer;
        uint32_t cmd_reply;
        struct binder_txn txn;
    } __attribute__((packed)) data;

            data.cmd_free = BC_FREE_BUFFER;
    data.buffer = buffer_to_free;
data.cmd_reply = BC_REPLY; //将我们前面 binder_thread_write 中 cmd 替换为 BC_REPLY 就可以知
data.txn.target = 0;        //道 service manager 如何将找到的 service 返回给 caller 了
   ......................
    binder_write(bs, &data, sizeof(data)); //调用 ioctl 与 binder kernel 通信
}
```

从这里走出去后，caller 该被唤醒了，client 进程就得到了所请求的 service 的 IBinder 对象在 Binder kernel 中的参考，这是一个远程 BBinder 对象。

连接建立后的 client 连接 Service 的通信过程：

```
virtual sp<ICamera> connect(const sp<ICameraClient>& cameraClient)
    {
        Parcel data, reply;
        data.writeInterfaceToken(ICameraService::getInterfaceDescriptor());
        data.writeStrongBinder(cameraClient->asBinder());
        remote()->transact(BnCameraService::CONNECT, data, &reply);
        return interface_cast<ICamera>(reply.readStrongBinder());
    }
```

向 前面分析的这里 remote 是我们得到的 CameraService 的对象，caller 进程会切入到 CameraService。android 的每 一个进程都会创建一个线程池，这个线程池用处理其他进程的请求。当没有数据的时候线程是挂起的，这时 binder kernel 唤醒了这个线程：

```
IPCThreadState::joinThreadPool(bool isMain)
{
    LOG_THREADPOOL("**** THREAD %p (PID %d) IS JOINING THE THREAD POOL\n", (void*)pthread_self(), getpid());

            mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
    status_t result;
    do {
        int32_t cmd;
        result = talkWithDriver();
        if (result >= NO_ERROR) {
            size_t IN = mIn.dataAvail();    //binder kernel 传递数据到 service
            if (IN < sizeof(int32_t)) continue;
            cmd = mIn.readInt32();
            IF_LOG_COMMANDS() {
                alog << "Processing top-level Command: "
                    << getReturnString(cmd) << endl;
            }
          result = executeCommand(cmd); //service  执行 binder kernel 请求的命令
        }

        // Let this thread exit the thread pool if it is no longer
        // needed and it is not the main process thread.
        if(result == TIMED_OUT && !isMain) {
            break;
        }
```

```
    } while (result != -ECONNREFUSED && result != -EBADF);
      .....................
}

       status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    RefBase::weakref_type* refs;
    status_t result = NO_ERROR;

switch (cmd) {
.....................
    case BR_TRANSACTION:
        {
            binder_transaction_data tr;
            result = mIn.read(&tr, sizeof(tr));
            LOG_ASSERT(result == NO_ERROR,
                "Not enough command data for brTRANSACTION");
            if (result != NO_ERROR) break;

            Parcel buffer;
            buffer.ipcSetDataReference(
                reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                tr.data_size,
                reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                tr.offsets_size/sizeof(size_t), freeBuffer, this);

            const pid_t origPid = mCallingPid;
            const uid_t origUid = mCallingUid;

            mCallingPid = tr.sender_pid;
            mCallingUid = tr.sender_euid;

            //LOGI(">>>> TRANSACT from pid %d uid %d\n", mCallingPid, mCallingUid);

            Parcel reply;
            .....................
            if (tr.target.ptr) {
  sp<BBinder> b((BBinder*)tr.cookie); //service 中 Binder 对象即 CameraService
   const status_t error = b->transact(tr.code, buffer, &reply, 0);//将调用
if (error < NO_ERROR) reply.setError(error);//CameraService 的 onTransact 函数

            } else {
                const status_t error = the_context_object->transact(tr.code, buffer, &reply, 0);
                if (error < NO_ERROR) reply.setError(error);
            }

            //LOGI("<<<< TRANSACT from pid %d restore pid %d uid %d\n",
            //     mCallingPid, origPid, origUid);

            if ((tr.flags & TF_ONE_WAY) == 0) {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
                sendReply(reply, 0);
            } else {
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
            }

            mCallingPid = origPid;
            mCallingUid = origUid;

            IF_LOG_TRANSACTIONS() {
                TextOutput::Bundle _b(alog);
                alog << "BC_REPLY thr " << (void*)pthread_self() << " / obj "
                    << tr.target.ptr << ": " << indent << reply << dedent << endl;
            }
        .....................................
}
```

```
        break;
}
    .....................................
if ((tr.flags & TF_ONE_WAY) == 0) {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
      sendReply(reply, 0); //通过 binder kernel 返回数据到 caller 进程这个过程大家
            } else {                //参照前面的叙述自己分析一下
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
            }
    if (result != NO_ERROR) {
        mLastError = result;
    }
    return result;
}
```

调用 CameraService BBinder 对象中的 transact 函数：

```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
   .....................
    switch (code) {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
            err = onTransact(code, data, reply, flags);
            break;
    }
    ...................
    return err;
}
```

将调用 CameraService 的 onTransact 函数，CameraService 继承了 BBinder。

```
status_t BnCameraService::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case CONNECT: {
            CHECK_INTERFACE(ICameraService, data, reply);
            sp<ICameraClient> cameraClient = interface_cast<ICameraClient>(data.readStrongBinder());
            sp<ICamera> camera = connect(cameraClient); //真正的处理函数
            reply->writeStrongBinder(camera->asBinder());
            return NO_ERROR;
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
```

至此完成了一次从 client 到 service 的通信。

设计一个多客户端的 Service

Service 可以连接不同的 Client,这里说的多客户端是指在 Service 中为不同的 client 创建不同的 IClient 接口，如果看过 AIDL 编程的话，应该清楚，Service 需要开放一个 IService 接口给客户端，我们通过 defaultServiceManager->getService 就可以得到相应的 service 一个 BpBinder 接口，通过这个接口调用 transact 函数就可以与 service 通信了，这样也就完成了一个简单的 service 与 client 程序了，但这里有个缺点就是，这个 IService 是对所有的 client 开放的，如果我们要对不同的 client 做区分的话，在建立连接的时候所有的 client 需要给 Service 一 个特性，这样做也未尝不可，但会很麻烦。比如对 Camera 来说可能不止一个摄像头，摄像头的功能也不一样，这样做就比较麻烦了。其实我们完全可以参照 QT 中多客户端的设计方式，在 Service 中为每一个 Client 都创建一个 IClient 接口，IService 接口只用于 Serivce 与 Client 建立连接用。对于 Camera，如果存在多摄像头我们就可以在 Service 中为不同的 Client 打开不同的设备。

```
import android.os.IBinder;
import android.os.RemoteException;
public class TestServerServer extends android.app.testServer.ITestServer.Stub
{
int mClientCount = 0;
testServerClient mClient[];
@Override
public android.app.testServer.ITestClient.Stub connect(ITestClient client) throws RemoteException
```

```
{
   // TODO Auto-generated method stub
testServerClient tClient = new testServerClient(this, client); //为 Client 创建
   mClient[mClientCount] = tClient;              //不同的 IClient
   mClientCount ++;
   System.out.printf("*** Server connect client is %d", client.asBinder());
   return tClient;
}

      @Override
public void receivedData(int count) throws RemoteException
{
   // TODO Auto-generated method stub

}
Public static class testServerClient extends android.app.testServer.ITestClient.Stub
{
   public android.app.testServer.ITestClient mClient;
   public TestServerServer mServer;
   public testServerClient(TestServerServer tServer, android.app.testServer.ITestClient tClient)
   {
    mServer = tServer;
    mClient = tClient;
   }
   public IBinder asBinder()
   {
    // TODO Auto-generated method stub
    return this;
   }
}
}
```

这仅仅是个 Service 的 demo 而已，如果添加这个作为 system Service 还得改一下 android 代码 avoid permission check!

总结：

假定一个 Client A 进程与 Service B 进程要建立 IPC 通信，通过前面的分析我们知道他的流程如下：

1：Service B 打开 Binder driver， 将自己的进程信息注册到 kernel 并为 Service 创建一个 binder_ref。
2：Service B 通过 Add_Service 将 Service 信息添加到 service_manager 进程
3：Service B 的 Thread pool 挂起 等待 client 的请求
4：Client A 调用 open_driver 打开 Binder driver 将自己的进程信息注册到 kernel 并为 Service 创建一个 binder_ref
5: Client A 调用 defaultManagerService.getService 得到 Service B 在 kernel 中的 IBinder 对象
6：通过 transact 与 Binder kernel 通信，Binder Kernel 将 Client A 挂起。
7：Binder Kernel 恢复 Service B thread pool 线程，并在 joinThreadPool 中处理 Client 的请求
8: Binder Kernel 挂起 Service B 并将 Service B 返回的数据写到 Client A
9：Binder Kernle 恢复 Client A

Binder kernel driver 在 Client A 与 Service B 之间扮演着中间代理的角色。任何通过 transact 传递的 IBinder 对象都会在 Binder kernel 中创建一个与此相关联的独一无二的 BInder 对象，用于区分不同的 Client。