# TODO

E/SELinux: avc: denied { add } for service=protocol pid=621 uid=1000 scontext=u:r:system\_server:s0 tcontext=u:object\_r:default\_android\_service:s0 tclass=service\_manager permissive=1

亲自动手从驱动程序到app？？

Aidl技术原理

http://gityuan.com/2015/10/31/binder-prepare/

<http://gityuan.com/2017/05/01/binder_exception/>

<http://gityuan.com/2016/10/29/binder-thread-pool/>

http://gityuan.com/2016/10/03/binder\_linktodeath/

<http://gityuan.com/2016/10/02/binder-died/>

Android Binder机制原理（史上最强理解，没有之一）（转）

<https://www.cnblogs.com/qingchen1984/p/5212755.html>

https://blog.csdn.net/freekiteyu/article/details/70082302

每个进程所允许Binder线程数，google提供的默认最大线程数为16个，受限于CPU，由于线程数太多，增加系统负载，并且每个进程默认分配的（1M-8K）大小的内存。

# 概述

## IPC原理

从进程角度来看IPC机制



每个Android的进程，只能运行在自己进程所拥有的虚拟地址空间。对应一个4GB的虚拟地址空间，其中3GB是用户空间，1GB是内核空间，当然内核空间的大小是可以通过参数配置调整的。对于用户空间，不同进程之间彼此是不能共享的，而内核空间却是可共享的。Client进程向Server进程通信，恰恰是利用进程间可共享的内核内存空间来完成底层通信工作的，Client端与Server端进程往往采用ioctl等方法跟内核空间的驱动进行交互。

## Binder原理

Binder通信采用C/S架构，从组件视角来说，包含Client、Server、ServiceManager以及binder驱动，其中ServiceManager用于管理系统中的各种服务。架构图如下所示：



可以看出无论是注册服务和获取服务的过程都需要ServiceManager，需要注意的是此处的Service Manager是指Native层的ServiceManager（C++），并非指framework层的ServiceManager(Java)。ServiceManager是整个Binder通信机制的大管家，是Android进程间通信机制Binder的守护进程，要掌握Binder机制，首先需要了解系统是如何首次启动Service Manager。当Service Manager启动之后，Client端和Server端通信时都需要先获取Service Manager接口，才能开始通信服务。

图中Client/Server/ServiceManage之间的相互通信都是基于Binder机制。既然基于Binder机制通信，那么同样也是C/S架构，则图中的3大步骤都有相应的Client端与Server端。

**注册服务(addService)**：Server进程要先注册Service到ServiceManager。该过程：Server是客户端，ServiceManager是服务端。

**获取服务(getService)**：Client进程使用某个Service前，须先向ServiceManager中获取相应的Service。该过程：Client是客户端，ServiceManager是服务端。

使用服务：Client根据得到的Service信息建立与Service所在的Server进程通信的通路，然后就可以直接与Service交互。该过程：client是客户端，server是服务端。

图中的Client,Server,Service Manager之间交互都是虚线表示，是由于它们彼此之间不是直接交互的，而是都通过与Binder驱动进行交互的，从而实现IPC通信方式。其中Binder驱动位于内核空间，Client,Server,Service Manager位于用户空间。Binder驱动和Service Manager可以看做是Android平台的基础架构，而Client和Server是Android的应用层，开发人员只需自定义实现client、Server端，借助Android的基本平台架构便可以直接进行IPC通信。

Binder框架定义了四个角色：Server，Client，ServiceManager（以后简称SMgr）以及Binder驱动。其中Server，Client，SMgr运行于用户空间，驱动运行于内核空间。这四个角色的关系和互联网类似：Server是服务器，Client是客户终端，SMgr是域名服务器（DNS），驱动是路由器

通讯过程

binder通信是一种client-server的通信结构，

1.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；

2.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；

3.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；

4.代理接口将该Parcel发送给内核中的binder driver.

5.server会读取binder driver中的请求数据，如果是发送给自己的，解包Parcel对象，处理并将结果返回；

6.整个的调用过程是一个同步过程，在server处理的时候，client会block住

## C/S模式

BpBinder(客户端)和BBinder(服务端)都是Android中Binder通信相关的代表，它们都从IBinder类中派生而来，关系图如下：



client端：BpBinder.transact()来发送事务请求；

server端：BBinder.onTransact()会接收到相应事务。

。

## 设计一个基于CS模型的binder的业务框架

鼎鼎有名的4大组件，就是基于上述模型来展开的，广播机制无非是在这个基础上加了加了各种检测

## 提纲

在后续的Binder源码分析过程中所涉及的源码，会有部分的精简，主要是去掉所有log输出语句，已减少代码篇幅过于长。通过前面的介绍，下面罗列一下关于Binder系列文章的提纲：

* [Binder系列1—Binder Driver初探](http://gityuan.com/2015/11/01/binder-driver/)
* [Binder系列2—Binder Driver再探](http://gityuan.com/2015/11/02/binder-driver-2/)
* [Binder系列3—启动Service Manager](http://gityuan.com/2015/11/07/binder-start-sm/)
* [Binder系列4—获取Service Manager](http://gityuan.com/2015/11/08/binder-get-sm/)
* [Binder系列5—注册服务(addService)](http://gityuan.com/2015/11/14/binder-add-service/)
* [Binder系列6—获取服务(getService)](http://gityuan.com/2015/11/15/binder-get-service/)
* [Binder系列7—framework层分析](http://gityuan.com/2015/11/21/binder-framework/)
* [Binder系列8—如何使用Binder](http://gityuan.com/2015/11/22/binder-use/)
* [Binder系列9—如何使用AIDL](http://gityuan.com/2015/11/23/binder-aidl/)
* [Binder系列10—总结](http://gityuan.com/2015/11/28/binder-summary/)

文章是从底层驱动往上层写的，这并不适合大家的理解，建议读者还是从上层往底层看。下面说说这个系列文章之间的彼此联系，也是对你阅读顺序的一个建议

首先阅读[Binder系列5—注册服务(addService)](http://gityuan.com/2015/11/14/binder-add-service/)和[Binder系列6—获取服务(getService)](http://gityuan.com/2015/11/15/binder-get-service/)，这两个过程都需要于ServiceManager打交道，那么这两个过程在开始之前都需要[Binder系列4—获取Service Manager](http://gityuan.com/2015/11/08/binder-get-sm/)，既然要获取Service Manager，那么就需要先[Binder系列3—启动Service Manager](http://gityuan.com/2015/11/07/binder-start-sm/)。在看Binder服务的注册和获取这两个过程中，不断追溯下去，最终调用到底层Binder底层驱动，这时需要了解[Binder系列1—Binder Driver初探](http://gityuan.com/2015/11/01/binder-driver/)和[Binder系列2—Binder Driver再探](http://gityuan.com/2015/11/02/binder-driver-2/)。

看完Binder系列1~系列6，那么对Binder的整个流程会有一个比较清晰的认知，这还只是停留在Native层(C/C++)。接下来，可以看看上层[Binder系列7—framework层分析](http://gityuan.com/2015/11/21/binder-framework/)的Binder架构情况，Java层 Binder架构的核心逻辑都是交由Native架构来完成，更多的是对Binder的一个封装过程，只有真正理解了Native层Binder架构，才能算掌握的Binder。

前面的这些都是讲述Binder整个流程以及原理，再接下来你可能想要自己写一套Binder的C/S架构服务。如果你是**系统工程师**可能会比较关心Native层和framework层分别该如何实现自己的自定义的Binder通信服务，见[Binder系列8—如何使用Binder](http://gityuan.com/2015/11/22/binder-use/)；如果你是**应用开发工程师**则应该更关心App是如何使用Binder的，那么可以查看文章[Binder系列9—如何使用AIDL](http://gityuan.com/2015/11/23/binder-aidl/)。

最后是对Binder的一个简单总结[Binder系列10—总结](http://gityuan.com/2015/11/28/binder-summary/)

## 源码目录

从上之下, 整个Binder架构所涉及的总共有以下5个目录

/framework/base/core/java/ (Java)

/framework/base/core/jni/ (JNI)

/framework/native/libs/binder (Native)

/framework/native/cmds/servicemanager/ (Native)

/kernel/drivers/staging/android (Driver)

### Java framework

/framework/base/core/java/android/os/

- IInterface.java

- IBinder.java

- Parcel.java

- IServiceManager.java

- ServiceManager.java

- ServiceManagerNative.java

- Binder.java

/framework/base/core/jni/

- android\_os\_Parcel.cpp

- AndroidRuntime.cpp

- android\_util\_Binder.cpp (核心类)

### Native framework

/framework/native/libs/binder

- IServiceManager.cpp

- BpBinder.cpp

- Binder.cpp

- IPCThreadState.cpp (核心类)

- ProcessState.cpp (核心类)

/framework/native/include/binder/

- IServiceManager.h

- IInterface.h

/framework/native/cmds/servicemanager/

- service\_manager.c

- binder.c

### Kernel

/kernel/drivers/staging/android/

- binder.c

- uapi/binder.h

# 启动ServiceManager

基于Android 6.0的源码剖析， 本文详细地讲解了ServiceManager启动流程

framework/native/cmds/servicemanager/

- service\_manager.c

- binder.c

kernel/drivers/ (不同Linux分支路径略有不同)

- staging/android/binder.c

- android/binder.c

## 概述

ServiceManager是Binder IPC通信过程中的守护进程，本身也是一个Binder服务，但并没有采用libbinder中的多线程模型来与Binder驱动通信，而是自行编写了binder.c直接和Binder驱动来通信，并且只有一个循环binder\_loop来进行读取和处理事务，这样的好处是简单而高效。

**Service Manager**：是系统中一个独立的进程，它是整个Binder机制的守护进程，用来管理开发者创建的各种Server，并且向Client提供查询Server远程接口的功能。

ServiceManager本身工作相对简单，其功能：查询和注册服务。 对于Binder IPC通信过程中，其实更多的情形是BpBinder和BBinder之间的通信，比如ActivityManagerProxy和ActivityManagerService之间的通信等。

### 流程图



启动过程主要以下几个阶段：

* 打开binder驱动：binder\_open；
* 注册成为binder服务的大管家：binder\_become\_context\_manager；
* 进入无限循环，处理client端发来的请求：binder\_loop；

### 启动

init启动**Service Manager**：

frameworks/native/cmds/servicemanager/servicemanager.rc

service servicemanager /system/bin/servicemanager  
 class core animation  
 user system  
 group system readproc  
 critical  
 onrestart restart healthd  
 onrestart restart zygote  
 onrestart restart audioserver  
 onrestart restart media  
 onrestart restart surfaceflinger  
 onrestart restart inputflinger  
 onrestart restart drm  
 onrestart restart cameraserver  
 writepid /dev/cpuset/system-background/tasks

在init进程启动的过程中会去解析init.rc文件

可以看出， servicemanager服务还是很重要的，它带了critical标志，即如果该进程连续crash几次，系统会进入恢复模式。而且如果该进程挂了，系统还会重启zygote、surfaceflinger等系统关键进程。其重要性真是不言而喻

其所对应的可执行程序/system/bin/servicemanager，所对应的源文件是service\_manager.c，进程名为/system/bin/servicemanager。

既然Service Manager组件是用来管理Server并且向Client提供查询Server远程接口的功能，那么，Service Manager就必然要和Server以及Client进行通信了。我们知道，Service Manger、Client和Server三者分别是运行在独立的进程当中，这样它们之间的通信也属于进程间通信了，而且也是采用Binder机制进行进程间通信。因此，Service Manager在充当Binder机制的守护进程的角色的同时，也在充当Server的角色，然而，它是一种特殊的Server，下面我们将会看到它的特殊之处

源代码位于frameworks/base/cmds/servicemanager目录下，主要是由binder.h、binder.c和service\_manager.c三个文件组成。Service Manager的入口位于service\_manager.c文件中的main函数：main函数如下：

Android系统进程间通信机制Binder的总体架构由Client、Server、ServiceManager和驱动程序Binder四个组件构成。今天主要来看看ServiceManager的实现吧。

### 类图

1. /\*
2. struct binder\_state
3. {
4. int fd; // /dev/binder的文件描述符
5. void \*mapped;   // mmap把设备文件/dev/binder映射到进程空间的起始地址
6. unsigned mapsize;   // mmap映射的大小
7. };

## service\_manager.c

main函数主要有三个功能：一是打开Binder设备文件；二是告诉Binder驱动程序自己是Binder上下文管理者，即我们前面所说的守护进程；三是进入一个无穷循环，充当Server的角色，等待Client的请求

1. **int** main(**int** argc, **char** \*\*argv)
2. {
3. /\*
4. struct binder\_state \*bs;
6. /\*
7. /\* the one magic object \*/
8. #define BINDER\_SERVICE\_MANAGER ((void\*) 0)
9. 一个magic对象，用0来标识一个Service Manager
10. \*/
11. **void** \*svcmgr = BINDER\_SERVICE\_MANAGER;
13. // 1.打开/dev/binder设备，并mmap映射之，大小为128K
14. bs = binder\_open(128\*1024);
16. // 2.告诉binder驱动我是Service Manager进程
17. //成为上下文管理者 【见小节2.3】
18. **if** (binder\_become\_context\_manager(bs)) {
19. ALOGE("cannot become context manager (%s)\n", strerror(errno));
20. **return** -1;
21. }
23. svcmgr\_handle = svcmgr;
25. // 3.循环接受并处理消息
26. binder\_loop(bs, svcmgr\_handler);
27. **return** 0;
28. }

宏BINDER\_SERVICE\_MANAGER定义frameworks/base/cmds/servicemanager/binder.h文件中：

1. /\* the one magic object \*/
2. #define BINDER\_SERVICE\_MANAGER ((void\*) 0)

它表示Service Manager的句柄为0。Binder通信机制使用句柄来代表远程接口，这个句柄的意义和Windows编程中用到的句柄是差不多的概念。前面说到，Service Manager在充当守护进程的同时，它充当Server的角色，当它作为远程接口使用时，它的句柄值便为0，这就是它的特殊之处，其余的Server的远程接口句柄值都是一个大于0 而且由Binder驱动程序自动进行分配的

## openbinder驱动

首先来看看binder\_open的实现，代码很简单，就不详细说明了：

[-> servicemanager/binder.c]

**[cpp]** [view plain](http://blog.csdn.net/hu3167343/article/details/38441119) [copy](http://blog.csdn.net/hu3167343/article/details/38441119)

1. **struct** binder\_state \*binder\_open(unsigned mapsize)
2. {
3. **struct** binder\_state \*bs;
5. bs = malloc(**sizeof**(\*bs));
6. **if** (!bs) {
7. errno = ENOMEM;
8. **return** 0;
9. }
10. //通过系统调用陷入内核，打开Binder设备驱动
11. bs->fd = open("/dev/binder", O\_RDWR | O\_CLOEXEC);
12. **if** (bs->fd < 0) {
13. fprintf(stderr,"binder: cannot open device (%s)\n",
14. strerror(errno));
15. **goto** fail\_open;
16. }
18. //通过系统调用，ioctl获取binder版本信息
19. if ((ioctl(bs->fd, BINDER\_VERSION, &vers) == -1) ||
20. (vers.protocol\_version != BINDER\_CURRENT\_PROTOCOL\_VERSION)) {
21. goto fail\_open; //内核空间与用户空间的binder不是同一版本
22. }
23. bs->mapsize = mapsize;
24. //通过系统调用，mmap内存映射，mmap必须是page的整数倍
25. bs->mapped = mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0);
26. **if** (bs->mapped == MAP\_FAILED) {
27. fprintf(stderr,"binder: cannot map device (%s)\n",
28. strerror(errno));
29. **goto** fail\_map;  // binder设备内存无法映射
30. }
32. **return** bs;
34. fail\_map:
35. close(bs->fd);
36. fail\_open:
37. free(bs);
38. **return** 0;
39. }

打开binder驱动相关操作:

先调用open()打开binder设备，open()方法经过系统调用，进入Binder驱动

然后调用方法binder\_open()，该方法会在Binder驱动层创建一个binder\_proc对象，再将binder\_proc对象赋值给fd->private\_data，同时放入全局链表binder\_procs。再通过ioctl()检验当前binder版本与Binder驱动层的版本是否一致。

调用mmap()进行内存映射，同理mmap()方法经过系统调用，对应于Binder驱动层的binder\_mmap()方法，该方法会在Binder驱动层创建Binder\_buffer对象，并放入当前binder\_proc的proc->buffers链表

### binder\_state

[-> servicemanager/binder.c]

struct binder\_state

{

int fd; // dev/binder的文件描述符

void \*mapped; //指向mmap的内存地址

size\_t mapsize; //分配的内存大小，默认为128KB

};

## binder\_become\_context\_manager注册为服务进程

接着来看binder\_become\_context\_manager的实现：

**[cpp]** [view plain](http://blog.csdn.net/hu3167343/article/details/38441119) [copy](http://blog.csdn.net/hu3167343/article/details/38441119)

1. **int** binder\_become\_context\_manager(**struct** binder\_state \*bs)
2. {
3. **return** ioctl(bs->fd, BINDER\_SET\_CONTEXT\_MGR, 0);
4. }

binder\_become\_context\_manager原来是通过ioctl的系统调用来告诉binder驱动自己是MGR进程

//通过ioctl，传递BINDER\_SET\_CONTEXT\_MGR指令【见小节2.3.1】

成为上下文的管理者，整个系统中只有一个这样的管理者。 通过ioctl()方法经过系统调用，对应于Binder驱动层的[binder\_ioctl()](http://gityuan.com/2015/11/01/binder-driver/#binderioctl)方法.

### binder\_ioctl

[-> kernel/drivers/android/binder.c]

static long binder\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg) {

binder\_lock(\_\_func\_\_);

switch (cmd) {

case BINDER\_SET\_CONTEXT\_MGR:

ret = binder\_ioctl\_set\_ctx\_mgr(filp);//【见小节2.3.2】

break;

}

case :...

}

binder\_unlock(\_\_func\_\_);

}

根据参数BINDER\_SET\_CONTEXT\_MGR，最终调用binder\_ioctl\_set\_ctx\_mgr()方法，这个过程会持有binder\_main\_lock。

**2.3.2 binder\_ioctl\_set\_ctx\_mgr**

[-> kernel/drivers/android/binder.c]

static int binder\_ioctl\_set\_ctx\_mgr(struct file \*filp)

{

int ret = 0;

struct binder\_proc \*proc = filp->private\_data;

kuid\_t curr\_euid = current\_euid();

//保证只创建一次mgr\_node对象

if (binder\_context\_mgr\_node != NULL) {

ret = -EBUSY;

goto out;

}

if (uid\_valid(binder\_context\_mgr\_uid)) {

...

} else {

//设置当前线程euid作为Service Manager的uid

binder\_context\_mgr\_uid = curr\_euid;

}

//创建ServiceManager实体【见小节2.3.3】

binder\_context\_mgr\_node = binder\_new\_node(proc, 0, 0);

...

binder\_context\_mgr\_node->local\_weak\_refs++;

binder\_context\_mgr\_node->local\_strong\_refs++;

binder\_context\_mgr\_node->has\_strong\_ref = 1;

binder\_context\_mgr\_node->has\_weak\_ref = 1;

out:

return ret;

}

进入binder驱动，在Binder驱动中定义的静态变量

// service manager所对应的binder\_node;

static struct binder\_node \*binder\_context\_mgr\_node;

// 运行service manager的线程uid

static kuid\_t binder\_context\_mgr\_uid = INVALID\_UID;

创建了全局的binder\_node对象binder\_context\_mgr\_node，并将binder\_context\_mgr\_node的强弱引用各加1.

**2.3.3 binder\_new\_node**

[-> kernel/drivers/android/binder.c]

static struct binder\_node \*binder\_new\_node(struct binder\_proc \*proc,

binder\_uintptr\_t ptr,

binder\_uintptr\_t cookie)

{

struct rb\_node \*\*p = &proc->nodes.rb\_node;

struct rb\_node \*parent = NULL;

struct binder\_node \*node;

//首次进来为空

while (\*p) {

parent = \*p;

node = rb\_entry(parent, struct binder\_node, rb\_node);

if (ptr < node->ptr)

p = &(\*p)->rb\_left;

else if (ptr > node->ptr)

p = &(\*p)->rb\_right;

else

return NULL;

}

//给新创建的binder\_node 分配内核空间

node = kzalloc(sizeof(\*node), GFP\_KERNEL);

if (node == NULL)

return NULL;

binder\_stats\_created(BINDER\_STAT\_NODE);

// 将新创建的node对象添加到proc红黑树；

rb\_link\_node(&node->rb\_node, parent, p);

rb\_insert\_color(&node->rb\_node, &proc->nodes);

node->debug\_id = ++binder\_last\_id;

node->proc = proc;

node->ptr = ptr;

node->cookie = cookie;

node->work.type = BINDER\_WORK\_NODE; //设置binder\_work的type

INIT\_LIST\_HEAD(&node->work.entry);

INIT\_LIST\_HEAD(&node->async\_todo);

return node;

}

在Binder驱动层创建[binder\_node结构体](http://gityuan.com/2015/11/01/binder-driver/#bindernode)对象，并将当前binder\_proc加入到binder\_node的node->proc。并创建binder\_node的async\_todo和binder\_work两个队列。

## binder\_loop消息循环

最后，调用binder\_loop开始接受消息：

1. **void** binder\_loop(**struct** binder\_state \*bs, binder\_handler func)
2. {
3. **int** res;
5. **struct** binder\_write\_read bwr;
6. unsigned readbuf[32];
8. bwr.write\_size = 0;
9. bwr.write\_consumed = 0;
10. bwr.write\_buffer = 0;
12. // 通知binder驱动，我要开始循环接受消息了。\
13. //将BC\_ENTER\_LOOPER命令发送给binder驱动，让Service Manager进入循环 【见小节2.4.1】
14. readbuf[0] = BC\_ENTER\_LOOPER;
15. binder\_write(bs, readbuf, **sizeof**(unsigned));
17. // 开始循环接收数据
18. **for** (;;) {
19. bwr.read\_size = **sizeof**(readbuf);
20. bwr.read\_consumed = 0;
21. bwr.read\_buffer = (unsigned) readbuf;
23. // 通过ioctl获得驱动中的数据
24. res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);
26. **if** (res < 0) {
27. ALOGE("binder\_loop: ioctl failed (%s)\n", strerror(errno));
28. **break**;
29. }
31. // 获得之后开始解析处理
32. res = binder\_parse(bs, 0, readbuf, bwr.read\_consumed, func);
33. **if** (res == 0) {
34. ALOGE("binder\_loop: unexpected reply?!\n");
35. **break**;
36. }
37. **if** (res < 0) {
38. ALOGE("binder\_loop: io error %d %s\n", res, strerror(errno));
39. **break**;
40. }
41. }
42. }

进入循环读写操作，由main()方法传递过来的参数func指向svcmgr\_handler。

binder\_write通过ioctl()将BC\_ENTER\_LOOPER命令发送给binder驱动，此时bwr只有write\_buffer有数据，进入binder\_thread\_write()方法。 接下来进入for循环，执行ioctl()，此时bwr只有read\_buffer有数据，那么进入binder\_thread\_read()方法。

#### binder\_write

[-> servicemanager/binder.c]

int binder\_write(struct binder\_state \*bs, void \*data, size\_t len) {

struct binder\_write\_read bwr;

int res;

bwr.write\_size = len;

bwr.write\_consumed = 0;

bwr.write\_buffer = (uintptr\_t) data; //此处data为BC\_ENTER\_LOOPER

bwr.read\_size = 0;

bwr.read\_consumed = 0;

bwr.read\_buffer = 0;

//【见小节2.4.2】

res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);

return res;

}

根据传递进来的参数，初始化bwr，其中write\_size大小为4，write\_buffer指向缓冲区的起始地址，其内容为BC\_ENTER\_LOOPER请求协议号。通过ioctl将bwr数据发送给binder驱动，则调用其binder\_ioctl方法，如下：

**2.4.2 binder\_ioctl**

[-> kernel/drivers/android/binder.c]

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;

ret = wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);

...

binder\_lock(\_\_func\_\_);

thread = binder\_get\_thread(proc); //获取binder\_thread

switch (cmd) {

case BINDER\_WRITE\_READ: //进行binder的读写操作

ret = binder\_ioctl\_write\_read(filp, cmd, arg, thread); //【见小节2.4.3】

if (ret)

goto err;

break;

case ...

}

ret = 0;

err:

if (thread)

thread->looper &= ~BINDER\_LOOPER\_STATE\_NEED\_RETURN;

binder\_unlock(\_\_func\_\_);

...

return ret;

}

**2.4.3 binder\_ioctl\_write\_read**

[-> kernel/drivers/android/binder.c]

static int binder\_ioctl\_write\_read(struct file \*filp,

unsigned int cmd, unsigned long arg,

struct binder\_thread \*thread)

{

int ret = 0;

struct binder\_proc \*proc = filp->private\_data;

void \_\_user \*ubuf = (void \_\_user \*)arg;

struct binder\_write\_read bwr;

if (copy\_from\_user(&bwr, ubuf, sizeof(bwr))) { //把用户空间数据ubuf拷贝到bwr

ret = -EFAULT;

goto out;

}

if (bwr.write\_size > 0) { //此时写缓存有数据【见小节2.4.4】

ret = binder\_thread\_write(proc, thread,

bwr.write\_buffer, bwr.write\_size, &bwr.write\_consumed);

...

}

if (bwr.read\_size > 0) { //此时读缓存无数据

...

}

if (copy\_to\_user(ubuf, &bwr, sizeof(bwr))) { //将内核数据bwr拷贝到用户空间ubuf

ret = -EFAULT;

goto out;

}

out:

return ret;

}

此处将用户空间的binder\_write\_read结构体 拷贝到内核空间.

**2.4.4 binder\_thread\_write**

[-> kernel/drivers/android/binder.c]

static int binder\_thread\_write(struct binder\_proc \*proc, struct binder\_thread \*thread, binder\_uintptr\_t binder\_buffer, size\_t size, binder\_size\_t \*consumed) {

uint32\_t cmd;

void \_\_user \*buffer = (void \_\_user \*)(uintptr\_t)binder\_buffer;

void \_\_user \*ptr = buffer + \*consumed;

void \_\_user \*end = buffer + size;

while (ptr < end && thread->return\_error == BR\_OK) {

get\_user(cmd, (uint32\_t \_\_user \*)ptr); //获取命令

switch (cmd) {

case BC\_ENTER\_LOOPER:

//设置该线程的looper状态

thread->looper |= BINDER\_LOOPER\_STATE\_ENTERED;

break;

case ...;

}

} }

从bwr.write\_buffer拿出cmd数据,此处为BC\_ENTER\_LOOPER. 可见上层本次调用binder\_write()方法，主要是完成设置当前线程的looper状态为BINDER\_LOOPER\_STATE\_ENTERED。

## binder\_parse

1. **int** binder\_parse(**struct** binder\_state \*bs, **struct** binder\_io \*bio,
2. uint32\_t \*ptr, uint32\_t size, binder\_handler func)
3. {
4. **int** r = 1;
5. uint32\_t \*end = ptr + (size / 4);
7. **while** (ptr < end) {
8. uint32\_t cmd = \*ptr++;
9. **switch**(cmd) {
10. **case** BR\_NOOP:  ////无操作，退出循环
11. **break**;
12. **case** BR\_TRANSACTION\_COMPLETE:
13. **break**;
14. **case** BR\_INCREFS:
15. **case** BR\_ACQUIRE:
16. **case** BR\_RELEASE:
17. **case** BR\_DECREFS:
18. #if TRACE
19. fprintf(stderr,"  %08x %08x\n", ptr[0], ptr[1]);
20. #endif
21. ptr += 2;
22. **break**;
23. **case** BR\_TRANSACTION: {
24. **struct** binder\_txn \*txn = (**void** \*) ptr;
25. **if** ((end - ptr) \* **sizeof**(uint32\_t) < **sizeof**(**struct** binder\_txn)) {
26. ALOGE("parse: txn too small!\n");
27. **return** -1;
28. }
29. binder\_dump\_txn(txn);  ///从txn解析出binder\_io信息
30. **if** (func) {
31. unsigned rdata[256/4];
32. **struct** binder\_io msg;
33. **struct** binder\_io reply;
34. **int** res;
36. bio\_init(&reply, rdata, **sizeof**(rdata), 4);
37. bio\_init\_from\_txn(&msg, txn);
38. res = func(bs, txn, &msg, &reply);
39. binder\_send\_reply(bs, &reply, txn->data, res);
40. }
41. ptr += **sizeof**(\*txn) / **sizeof**(uint32\_t);
42. **break**;
43. }
44. **case** BR\_REPLY: {
45. **struct** binder\_txn \*txn = (**void**\*) ptr;
46. **if** ((end - ptr) \* **sizeof**(uint32\_t) < **sizeof**(**struct** binder\_txn)) {
47. ALOGE("parse: reply too small!\n");
48. **return** -1;
49. }
50. binder\_dump\_txn(txn);
51. **if** (bio) {
52. bio\_init\_from\_txn(bio, txn);
53. bio = 0;
54. } **else** {
55. /\* todo FREE BUFFER \*/
56. }
57. ptr += (**sizeof**(\*txn) / **sizeof**(uint32\_t));
58. r = 0;
59. **break**;
60. }
61. **case** BR\_DEAD\_BINDER: {
62. **struct** binder\_death \*death = (**void**\*) \*ptr++;
63. // binder死亡消息【见小节3.3】
64. death->func(bs, death->ptr);
65. **break**;
66. }
67. **case** BR\_FAILED\_REPLY:
68. r = -1;
69. **break**;
70. **case** BR\_DEAD\_REPLY:
71. r = -1;
72. **break**;
73. **default**:
74. ALOGE("parse: OOPS %d\n", cmd);
75. **return** -1;
76. }
77. }
79. **return** r;
80. }

如果接收到了消息，那么binder\_parse会经过简单的转换func，最后还是会调用回调函数svcmgr\_handler:

解析binder信息，此处参数ptr指向BC\_ENTER\_LOOPER，func指向svcmgr\_handler。故有请求到来，则调用svcmgr\_handler。

### bio\_init

[-> servicemanager/binder.c]

void bio\_init(struct binder\_io \*bio, void \*data,

size\_t maxdata, size\_t maxoffs)

{

size\_t n = maxoffs \* sizeof(size\_t);

if (n > maxdata) {

...

}

bio->data = bio->data0 = (char \*) data + n;

bio->offs = bio->offs0 = data;

bio->data\_avail = maxdata - n;

bio->offs\_avail = maxoffs;

bio->flags = 0;

}

其中

struct binder\_io

{

char \*data; /\* pointer to read/write from \*/

binder\_size\_t \*offs; /\* array of offsets \*/

size\_t data\_avail; /\* bytes available in data buffer \*/

size\_t offs\_avail; /\* entries available in offsets array \*/

char \*data0; //data buffer起点位置

binder\_size\_t \*offs0; //buffer偏移量的起点位置

uint32\_t flags;

uint32\_t unused;

};

### 2.5.2 bio\_init\_from\_txn

[-> servicemanager/binder.c]

void bio\_init\_from\_txn(struct binder\_io \*bio, struct binder\_transaction\_data \*txn)

{

bio->data = bio->data0 = (char \*)(intptr\_t)txn->data.ptr.buffer;

bio->offs = bio->offs0 = (binder\_size\_t \*)(intptr\_t)txn->data.ptr.offsets;

bio->data\_avail = txn->data\_size;

bio->offs\_avail = txn->offsets\_size / sizeof(size\_t);

bio->flags = BIO\_F\_SHARED;

}

将readbuf的数据赋给bio对象的data

## svcmgr\_handler

svcmgr\_handler

1. **int** svcmgr\_handler(**struct** binder\_state \*bs,
2. **struct** binder\_txn \*txn,
3. **struct** binder\_io \*msg,
4. **struct** binder\_io \*reply)
5. {
6. **struct** svcinfo \*si;
7. uint16\_t \*s;
8. unsigned len;
9. **void** \*ptr;
10. uint32\_t strict\_policy;
11. **int** allow\_isolated;
13. // 确认消息是发给server manager
14. **if** (txn->target != svcmgr\_handle)
15. **return** -1;
17. …….
19. **switch**(txn->code) {
20. **case** SVC\_MGR\_GET\_SERVICE:
21. **case** SVC\_MGR\_CHECK\_SERVICE:
22. // 获得当前服务名，s变量为服务名
23. s = bio\_get\_string16(msg, &len);
24. /\*
25. Server manager在其进程空间维护了一个service的全局链表svclist，
26. do\_find\_service通过服务名来查找该服务是否存在，并返回给客户端
27. \*/
28. ptr = do\_find\_service(bs, s, len, txn->sender\_euid);
29. **if** (!ptr)
30. **break**;
31. bio\_put\_ref(reply, ptr);
32. **return** 0;
34. // 其他的server进程来添加服务
35. **case** SVC\_MGR\_ADD\_SERVICE:
36. s = bio\_get\_string16(msg, &len);
37. ptr = bio\_get\_ref(msg);
38. allow\_isolated = bio\_get\_uint32(msg) ? 1 : 0;
39. **if** (do\_add\_service(bs, s, len, ptr, txn->sender\_euid, allow\_isolated))
40. **return** -1;
41. **break**;
43. // 枚举所有已经注册了的服务名
44. **case** SVC\_MGR\_LIST\_SERVICES: {
45. unsigned n = bio\_get\_uint32(msg);
47. si = svclist;
48. **while** ((n-- > 0) && si)
49. si = si->next;
50. **if** (si) {
51. bio\_put\_string16(reply, si->name);
52. **return** 0;
53. }
54. **return** -1;
55. }
56. **default**:
57. ALOGE("unknown code %d\n", txn->code);
58. **return** -1;
59. }
61. bio\_put\_uint32(reply, 0);
62. **return** 0;
63. }

该方法的功能：查询服务，注册服务，以及列举所有服务

svcinfo

struct svcinfo

{

struct svcinfo \*next;

uint32\_t handle; //服务的handle值

struct binder\_death death;

int allow\_isolated;

size\_t len; //名字长度

uint16\_t name[0]; //服务名

};

每一个服务用svcinfo结构体来表示，该handle值是在注册服务的过程中，由服务所在进程那一端所确定的。

## 核心工作

servicemanager的核心工作就是注册服务和查询服务。

### SVC\_MGR\_GET\_SERVICE

**case** SVC\_MGR\_CHECK\_SERVICE:  
 s = bio\_get\_string16(msg, &len);  
 **if** (s == NULL) {  
 **return** -1;  
 }  
 handle = do\_find\_service(s, len, txn->sender\_euid, txn->sender\_pid);  
 **if** (!handle)  
 **break**;  
 bio\_put\_ref(reply, handle);

#### do\_find\_service

[-> service\_manager.c]

uint32\_t do\_find\_service(struct binder\_state \*bs, const uint16\_t \*s, size\_t len, uid\_t uid, pid\_t spid)

{

//查询相应的服务 【见小节3.1.1】

struct svcinfo \*si = find\_svc(s, len);

if (!si || !si->handle) {

return 0;

}

if (!si->allow\_isolated) {

uid\_t appid = uid % AID\_USER;

//检查该服务是否允许孤立于进程而单独存在

if (appid >= AID\_ISOLATED\_START && appid <= AID\_ISOLATED\_END) {

return 0;

}

}

//服务是否满足查询条件

if (!svc\_can\_find(s, len, spid)) {

return 0;

}

return si->handle;

}

查询到目标服务，并返回该服务所对应的handle

##### 3.1.1 find\_svc

struct svcinfo \*find\_svc(const uint16\_t \*s16, size\_t len)

{

struct svcinfo \*si;

for (si = svclist; si; si = si->next) {

//当名字完全一致，则返回查询到的结果

if ((len == si->len) &&

!memcmp(s16, si->name, len \* sizeof(uint16\_t))) {

return si;

}

}

return NULL;

}

从svclist服务列表中，根据服务名遍历查找是否已经注册。当服务已存在svclist，则返回相应的服务名，否则返回NULL。

当找到服务的handle, 则调用bio\_put\_ref(reply, handle)，将handle封装到reply.

#### 3.1.2 bio\_put\_ref

***void bio\_put\_ref(struct binder\_io \*bio, uint32\_t handle) {***

***struct flat\_binder\_object \*obj;***

***if (handle)***

***obj = bio\_alloc\_obj(bio); //[见小节3.1.3]***

***else***

***obj = bio\_alloc(bio, sizeof(\*obj));***

***if (!obj)***

***return;***

***obj->flags = 0x7f | FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS;***

***obj->type = BINDER\_TYPE\_HANDLE; //返回的是HANDLE类型***

***obj->handle = handle;***

***obj->cookie = 0;***

***}***

##### 3.1.3 bio\_alloc\_obj

static struct flat\_binder\_object \*bio\_alloc\_obj(struct binder\_io \*bio)

{

struct flat\_binder\_object \*obj;

obj = bio\_alloc(bio, sizeof(\*obj));//[见小节3.1.4]

if (obj && bio->offs\_avail) {

bio->offs\_avail--;

\*bio->offs++ = ((char\*) obj) - ((char\*) bio->data0);

return obj;

}

bio->flags |= BIO\_F\_OVERFLOW;

return NULL;

}

##### 3.1.4 bio\_alloc

static void \*bio\_alloc(struct binder\_io \*bio, size\_t size)

{

size = (size + 3) & (~3);

if (size > bio->data\_avail) {

bio->flags |= BIO\_F\_OVERFLOW;

return NULL;

} else {

void \*ptr = bio->data;

bio->data += size;

bio->data\_avail -= size;

return ptr;

}

}

### SVC\_MGR\_ADD\_SERVICE

**case** SVC\_MGR\_ADD\_SERVICE:  
 s = bio\_get\_string16(msg, &len);  
 **if** (s == NULL) {  
 **return** -1;  
 }  
 handle = bio\_get\_ref(msg);  
 allow\_isolated = bio\_get\_uint32(msg) ? 1 : 0;  
 **if** (do\_add\_service(bs, s, len, handle, txn->sender\_euid,  
 allow\_isolated, txn->sender\_pid))  
 **return** -1;  
 **break**;

#### do\_add\_service

发过

1. **int** do\_add\_service(**struct** binder\_state \*bs,
2. uint16\_t \*s, unsigned len,
3. **void** \*ptr, unsigned uid, **int** allow\_isolated)
4. {
5. **struct** svcinfo \*si;
6. // 参数检查
7. **if** (!ptr || (len == 0) || (len > 127))
8. **return** -1;
10. // 根据uid和服务名判断是否可以被注册
11. **if** (!svc\_can\_register(uid, s)) {
12. ALOGE("add\_service('%s',%p) uid=%d - PERMISSION DENIED\n",
13. str8(s), ptr, uid);
14. **return** -1;
15. }
17. // 根据服务名，遍历svclist服务链表
18. si = find\_svc(s, len);
19. **if** (si) {
20. **if** (si->ptr) {
21. ALOGE("add\_service('%s',%p) uid=%d - ALREADY REGISTERED, OVERRIDE\n",
22. str8(s), ptr, uid);
23. ////服务已注册时，释放相应的服务
24. svcinfo\_death(bs, si);  // 清理前一个服务
25. }
26. si->ptr = ptr;   // 赋值新服务
27. } **else** {
28. si = malloc(**sizeof**(\*si) + (len + 1) \* **sizeof**(uint16\_t));
29. **if** (!si) {  //内存不足，无法分配足够内存
30. ALOGE("add\_service('%s',%p) uid=%d - OUT OF MEMORY\n",
31. str8(s), ptr, uid);
32. **return** -1;
33. }
34. si->ptr = ptr;
35. si->len = len;  //内存拷贝服务信息
36. memcpy(si->name, s, (len + 1) \* **sizeof**(uint16\_t));
37. si->name[len] = '\0';
38. si->death.func = svcinfo\_death;  // 如果server进程挂了，那么调用该函数处理
39. si->death.ptr = si;
40. si->allow\_isolated = allow\_isolated;
41. si->next = svclist;
42. svclist = si;  // svclist保存所有已注册的服务
43. }
44. //以BC\_ACQUIRE命令，handle为目标的信息，通过ioctl发送给binder驱动
45. binder\_acquire(bs, ptr);
46. //以BC\_REQUEST\_DEATH\_NOTIFICATION命令的信息，通过ioctl发送给binder驱动，主要用于清理内存等收尾工作。[见小节3.3]
47. binder\_link\_to\_death(bs, ptr, &si->death);
48. **return** 0;
49. }

do\_add\_service首先会调用svc\_can\_register函数检测当前要添加的服务是否被允许，函数如下：

##### svc\_can\_register

用户app无法注册服务，因此system\_uid=1000才可以！

**static int** svc\_can\_register(**const** uint16\_t \*name, size\_t name\_len, pid\_t spid, uid\_t uid)  
{  
 **const char** \*perm = **"add"**;  
  
 **if** (multiuser\_get\_app\_id(uid) >= AID\_APP) {//10000  
 **return** 0; */\* Don't allow apps to register services \*/* }  
  
 **return** check\_mac\_perms\_from\_lookup(spid, uid, perm, str8(name, name\_len)) ? 1 : 0;  
}

system/core/libcutils/include/private/android\_filesystem\_config.h

#define AID\_APP 10000 */\** ***TODO: switch users over to AID\_APP\_START \*/***#define AID\_APP\_START 10000 */\* first app user \*/*#define AID\_APP\_END 19999 */\* last app user \*/*

allowed数组内容如下

**[cpp]** [view plain](http://blog.csdn.net/hu3167343/article/details/38441119) [copy](http://blog.csdn.net/hu3167343/article/details/38441119)

1. **static** **struct** {
2. unsigned uid;
3. **const** **char** \*name;
4. } allowed[] = {
5. { AID\_MEDIA, "media.audio\_flinger" },
6. { AID\_MEDIA, "media.log" },
7. { AID\_MEDIA, "media.player" },
8. { AID\_MEDIA, "media.camera" },
9. { AID\_MEDIA, "media.audio\_policy" },
10. { AID\_DRM,   "drm.drmManager" },
11. { AID\_NFC,   "nfc" },
12. { AID\_BLUETOOTH, "bluetooth" },
13. { AID\_RADIO, "radio.phone" },
14. { AID\_RADIO, "radio.sms" },
15. { AID\_RADIO, "radio.phonesubinfo" },
16. { AID\_RADIO, "radio.simphonebook" },
17. /\* TODO: remove after phone services are updated: \*/
18. { AID\_RADIO, "phone" },
19. { AID\_RADIO, "sip" },
20. { AID\_RADIO, "isms" },
21. { AID\_RADIO, "iphonesubinfo" },
22. { AID\_RADIO, "simphonebook" },
23. { AID\_MEDIA, "common\_time.clock" },
24. { AID\_MEDIA, "common\_time.config" },
25. { AID\_KEYSTORE, "android.security.keystore" },
26. };

##### svcinfo\_death

[-> service\_manager.c]

***void svcinfo\_death(struct binder\_state \*bs, void \*ptr) {***

***struct svcinfo \*si = (struct svcinfo\* ) ptr;***

***if (si->handle) {***

***binder\_release(bs, si->handle);***

***si->handle = 0;***

***}***

***}***

##### binder\_acquire

**3.2.3 bio\_get\_ref**

[-> servicemanager/binder.c]

***uint32\_t bio\_get\_ref(struct binder\_io \*bio) {***

***struct flat\_binder\_object \*obj;***

***obj = \_bio\_get\_obj(bio);***

***if (!obj)***

***return 0;***

***if (obj->type == BINDER\_TYPE\_HANDLE)***

***return obj->handle;***

***return 0;***

***}***

### binder\_link\_to\_death

[-> servicemanager/binder.c]

void binder\_link\_to\_death(struct binder\_state \*bs, uint32\_t target, struct binder\_death \*death) {

struct {

uint32\_t cmd;

struct binder\_handle\_cookie payload;

} \_\_attribute\_\_((packed)) data;

data.cmd = BC\_REQUEST\_DEATH\_NOTIFICATION;

data.payload.handle = target;

data.payload.cookie = (uintptr\_t) death;

binder\_write(bs, &data, sizeof(data)); //[见小节3.3.1]

}

binder\_write经过跟小节2.4.1一样的方式, 进入Binder driver后,直接调用后进入binder\_thread\_write, 处理BC\_REQUEST\_DEATH\_NOTIFICATION命令

**3.3.1 binder\_ioctl\_write\_read**

[-> kernel/drivers/android/binder.c]

static int binder\_ioctl\_write\_read(struct file \*filp,

unsigned int cmd, unsigned long arg,

struct binder\_thread \*thread)

{

int ret = 0;

struct binder\_proc \*proc = filp->private\_data;

void \_\_user \*ubuf = (void \_\_user \*)arg;

struct binder\_write\_read bwr;

if (copy\_from\_user(&bwr, ubuf, sizeof(bwr))) { //把用户空间数据ubuf拷贝到bwr

ret = -EFAULT;

goto out;

}

if (bwr.write\_size > 0) { //此时写缓存有数据【见小节3.3.2】

ret = binder\_thread\_write(proc, thread,

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 out;

}

}

if (bwr.read\_size > 0) { //此时读缓存有数据【见小节3.3.3】

ret = binder\_thread\_read(proc, thread, bwr.read\_buffer,

bwr.read\_size,

&bwr.read\_consumed,

filp->f\_flags & O\_NONBLOCK);

if (!list\_empty(&proc->todo)) //进程todo队列不为空,则唤醒该队列中的线程

wake\_up\_interruptible(&proc->wait);

if (ret < 0) {

if (copy\_to\_user(ubuf, &bwr, sizeof(bwr)))

ret = -EFAULT;

goto out;

}

}

if (copy\_to\_user(ubuf, &bwr, sizeof(bwr))) { //将内核数据bwr拷贝到用户空间ubuf

ret = -EFAULT;

goto out;

}

out:

return ret;

}

**3.3.2 binder\_thread\_write**

[-> kernel/drivers/android/binder.c]

***static int binder\_thread\_write(struct binder\_proc \*proc, struct binder\_thread \*thread, binder\_uintptr\_t binder\_buffer, size\_t size, binder\_size\_t \*consumed) {***

***uint32\_t cmd;***

***struct binder\_context \*context = proc->context;***

***void \_\_user \*buffer = (void \_\_user \*)(uintptr\_t)binder\_buffer;***

***void \_\_user \*ptr = buffer + \*consumed; //ptr指向小节3.2.3中bwr中write\_buffer的data.***

***void \_\_user \*end = buffer + size;***

***while (ptr < end && thread->return\_error == BR\_OK) {***

***get\_user(cmd, (uint32\_t \_\_user \*)ptr); //获取BC\_REQUEST\_DEATH\_NOTIFICATION***

***ptr += sizeof(uint32\_t);***

***switch (cmd) {***

***case BC\_REQUEST\_DEATH\_NOTIFICATION:{ //注册死亡通知***

***uint32\_t target;***

***void \_\_user \*cookie;***

***struct binder\_ref \*ref;***

***struct binder\_ref\_death \*death;***

***get\_user(target, (uint32\_t \_\_user \*)ptr); //获取target***

***ptr += sizeof(uint32\_t);***

***get\_user(cookie, (void \_\_user \* \_\_user \*)ptr); //获取death***

***ptr += sizeof(void \*);***

***ref = binder\_get\_ref(proc, target); //拿到目标服务的binder\_ref***

***if (cmd == BC\_REQUEST\_DEATH\_NOTIFICATION) {***

***if (ref->death) {***

***break; //已设置死亡通知***

***}***

***death = kzalloc(sizeof(\*death), GFP\_KERNEL);***

***INIT\_LIST\_HEAD(&death->work.entry);***

***death->cookie = cookie;***

***ref->death = death;***

***if (ref->node->proc == NULL) { //当目标binder服务所在进程已死,则发送死亡通知***

***ref->death->work.type = BINDER\_WORK\_DEAD\_BINDER;***

***//当前线程为binder线程,则直接添加到当前线程的todo队列. 接下来,进入[小节3.2.6]***

***if (thread->looper & (BINDER\_LOOPER\_STATE\_REGISTERED | BINDER\_LOOPER\_STATE\_ENTERED)) {***

***list\_add\_tail(&ref->death->work.entry, &thread->todo);***

***} else {***

***list\_add\_tail(&ref->death->work.entry, &proc->todo);***

***wake\_up\_interruptible(&proc->wait);***

***}***

***}***

***} else {***

***...***

***}***

***} break;***

***case ...;***

***}***

***\*consumed = ptr - buffer;***

***} }***

此方法中的proc, thread都是指当前servicemanager进程的信息. 此时TODO队列有数据,则进入binder\_thread\_read.

那么哪些场景会向队列增加BINDER\_WORK\_DEAD\_BINDER事务呢? 那就是当binder所在进程死亡后,会调用binder\_release方法, 然后调用binder\_node\_release.这个过程便会发出死亡通知的回调.

**3.3.3 binder\_thread\_read**

static int binder\_thread\_read(struct binder\_proc \*proc,

struct binder\_thread \*thread,

binder\_uintptr\_t binder\_buffer, size\_t size,

binder\_size\_t \*consumed, int non\_block)

...

//只有当前线程todo队列为空，并且transaction\_stack也为空，才会开始处于当前进程的事务

if (wait\_for\_proc\_work) {

...

ret = wait\_event\_freezable\_exclusive(proc->wait, binder\_has\_proc\_work(proc, thread));

} else {

...

ret = wait\_event\_freezable(thread->wait, binder\_has\_thread\_work(thread));

}

binder\_lock(\_\_func\_\_); //加锁

if (wait\_for\_proc\_work)

proc->ready\_threads--; //空闲的binder线程减1

thread->looper &= ~BINDER\_LOOPER\_STATE\_WAITING;

while (1) {

uint32\_t cmd;

struct binder\_transaction\_data tr;

struct binder\_work \*w;

struct binder\_transaction \*t = NULL;

//从todo队列拿出前面放入的binder\_work, 此时type为BINDER\_WORK\_DEAD\_BINDER

if (!list\_empty(&thread->todo)) {

w = list\_first\_entry(&thread->todo, struct binder\_work,

entry);

} else if (!list\_empty(&proc->todo) && wait\_for\_proc\_work) {

w = list\_first\_entry(&proc->todo, struct binder\_work,

entry);

}

switch (w->type) {

case BINDER\_WORK\_DEAD\_BINDER: {

struct binder\_ref\_death \*death;

uint32\_t cmd;

death = container\_of(w, struct binder\_ref\_death, work);

if (w->type == BINDER\_WORK\_CLEAR\_DEATH\_NOTIFICATION)

...

else

cmd = BR\_DEAD\_BINDER; //进入此分支

put\_user(cmd, (uint32\_t \_\_user \*)ptr);//拷贝到用户空间[见小节3.3.4]

ptr += sizeof(uint32\_t);

//此处的cookie是前面传递的svcinfo\_death

put\_user(death->cookie, (binder\_uintptr\_t \_\_user \*)ptr);

ptr += sizeof(binder\_uintptr\_t);

if (w->type == BINDER\_WORK\_CLEAR\_DEATH\_NOTIFICATION) {

...

} else

list\_move(&w->entry, &proc->delivered\_death);

if (cmd == BR\_DEAD\_BINDER)

goto done;

} break;

}

}

...

return 0;

}

将命令BR\_DEAD\_BINDER写到用户空间, 此处的cookie是前面传递的svcinfo\_death. 当binder\_loop下一次 执行binder\_parse的过程便会处理该消息。

**3.3.4 binder\_parse**

[-> servicemanager/binder.c]

int binder\_parse(struct binder\_state \*bs, struct binder\_io \*bio, uintptr\_t ptr, size\_t size, binder\_handler func) {

int r = 1;

uintptr\_t end = ptr + (uintptr\_t) size;

while (ptr < end) {

uint32\_t cmd = \*(uint32\_t \*) ptr;

ptr += sizeof(uint32\_t);

switch(cmd) {

case BR\_DEAD\_BINDER: {

struct binder\_death \*death = (struct binder\_death \*)(uintptr\_t) \*(binder\_uintptr\_t \*)ptr;

ptr += sizeof(binder\_uintptr\_t);

// binder死亡消息【见小节3.3.5】

death->func(bs, death->ptr);

break;

}

...

}

}

return r;

}

由小节3.2的 si->death.func = (void\*) svcinfo\_death; 可知此处 death->func便是执行svcinfo\_death()方法.

**3.3.5 svcinfo\_death**

[-> service\_manager.c]

***void svcinfo\_death(struct binder\_state \*bs, void \*ptr) {***

***struct svcinfo \*si = (struct svcinfo\* ) ptr;***

***if (si->handle) {***

***binder\_release(bs, si->handle);***

***si->handle = 0;***

***}***

***}***

**3.3.6 binder\_release**

[-> service\_manager.c]

void binder\_release(struct binder\_state \*bs, uint32\_t target) {

uint32\_t cmd[2];

cmd[0] = BC\_RELEASE;

cmd[1] = target;

binder\_write(bs, cmd, sizeof(cmd));

}

向Binder Driver写入BC\_RELEASE命令, 最终进入Binder Driver后执行binder\_dec\_ref(ref, 1)来减少binder node的引用.

**3.4 binder\_send\_reply**

[-> servicemanager/binder.c]

void binder\_send\_reply(struct binder\_state \*bs, struct binder\_io \*reply, binder\_uintptr\_t buffer\_to\_free, int status) {

struct {

uint32\_t cmd\_free;

binder\_uintptr\_t buffer;

uint32\_t cmd\_reply;

struct binder\_transaction\_data txn;

} \_\_attribute\_\_((packed)) data;

data.cmd\_free = BC\_FREE\_BUFFER; //free buffer命令

data.buffer = buffer\_to\_free;

data.cmd\_reply = BC\_REPLY; // reply命令

data.txn.target.ptr = 0;

data.txn.cookie = 0;

data.txn.code = 0;

if (status) {

data.txn.flags = TF\_STATUS\_CODE;

data.txn.data\_size = sizeof(int);

data.txn.offsets\_size = 0;

data.txn.data.ptr.buffer = (uintptr\_t)&status;

data.txn.data.ptr.offsets = 0;

} else {

data.txn.flags = 0;

data.txn.data\_size = reply->data - reply->data0;

data.txn.offsets\_size = ((char\*) reply->offs) - ((char\*) reply->offs0);

data.txn.data.ptr.buffer = (uintptr\_t)reply->data0;

data.txn.data.ptr.offsets = (uintptr\_t)reply->offs0;

}

//向Binder驱动通信

binder\_write(bs, &data, sizeof(data));

}

当小节2.5执行binder\_parse方法，先调用svcmgr\_handler()，再然后执行binder\_send\_reply过程。该方法会调用 [小节2.4.1] binder\_write进入binder驱动后，将BC\_FREE\_BUFFER和BC\_REPLY命令协议发送给Binder驱动，向client端发送reply. 其中data的数据区中保存的是TYPE为HANDLE.

### 核心策略(check\_mac\_perms\_from\_lookup)

//

**static char** \*service\_manager\_context;  
**static struct** selabel\_handle\* sehandle;

**static bool** check\_mac\_perms\_from\_lookup(pid\_t spid, uid\_t uid, **const char** \*perm, **const char** \*name)  
{  
 **bool** allowed;  
 **char** \*tctx = NULL;  
  
 **if** (!sehandle) {  
 ALOGE(**"SELinux: Failed to find sehandle. Aborting service\_manager.\n"**);  
 abort();  
 }  
  
 **if** (selabel\_lookup(sehandle, &tctx, name, 0) != 0) {  
 ALOGE(**"SELinux: No match for %s in service\_contexts.\n"**, name);  
 **return false**;  
 }  
  
 allowed = check\_mac\_perms(spid, uid, tctx, perm, name);  
 freecon(tctx);  
 **return** allowed;  
}

#### selabel\_lookup(sehandle, &tctx, name, 0)

#### check\_mac\_perms

**static bool** check\_mac\_perms(pid\_t spid, uid\_t uid, **const char** \*tctx, **const char** \*perm, **const char** \*name)  
{  
 **char** \*sctx = NULL;  
 **const char** \***class** = **"service\_manager"**;  
 **bool** allowed;  
 **struct** audit\_data ad;  
  
 **if** (getpidcon(spid, &sctx) < 0) {  
 ALOGE(**"SELinux: getpidcon(pid=%d) failed to retrieve pid context.\n"**, spid);  
 **return false**;  
 }  
  
 ad.pid = spid;  
 ad.uid = uid;  
 ad.name = name;  
  
 **int** result = selinux\_check\_access(sctx, tctx, **class**, perm, (**void** \*) &ad);  
 allowed = (result == 0);  
  
 freecon(sctx);  
 **return** allowed;  
}

## 总结

ServiceManger集中管理系统内的所有服务，通过权限控制进程是否有权注册服务,通过字符串名称来查找对应的Service; 由于ServiceManger进程建立跟所有向其注册服务的死亡通知, 那么当服务所在进程死亡后, 会只需告知ServiceManager. 每个Client通过查询ServiceManager可获取Server进程的情况，降低所有Client进程直接检测会导致负载过重。

### ServiceManager启动流程：

打开binder驱动，并调用mmap()方法分配128k的内存映射空间：binder\_open();

通知binder驱动使其成为守护进程：binder\_become\_context\_manager()；

验证selinux权限，判断进程是否有权注册或查看指定服务；

进入循环状态，等待Client端的请求：binder\_loop()。

注册服务的过程，根据服务名称，但同一个服务已注册，重新注册前会先移除之前的注册信息；

死亡通知: 当binder所在进程死亡后,会调用binder\_release方法,然后调用binder\_node\_release.这个过程便会发出死亡通知的回调.

### ServiceManager核心功能

注册服务：记录服务名和handle信息，保存到svclist列表；

查询服务：根据服务名查询相应的的handle信息。

### Service Manager存在的意义

* Service Manager能集中管理系统内的所有服务，它能被施加权限控制，并不是任何进程都能注册服务的。
* Service Manager支持通过字符串名称来查找对应的Service。
* 由于各种原因的影响，Server进程可能生死无常。如果有了Service Manager做统一的管理，那么Client只要向Service Manager做查询，就能得到Server的最新信息。

用来管理开发者创建的各种Server，并且向Client提供查询Server远程接口的功能。

<http://blog.csdn.net/hu3167343/article/details/38441119>

# 获取ServiceManager

基于Android 6.0的源码剖析， 本文详细地讲解defaultServiceManager流程

framework/native/libs/binder/

- ProcessState.cpp

- BpBinder.cpp

- Binder.cpp

- IServiceManager.cpp

framework/native/include/binder/

- IServiceManager.h

- IInterface.h

## 概述

获取Service Manager是通过defaultServiceManager()方法来完成，当进程注册服务(addService)或 获取服务(getService)的过程之前，都需要先调用defaultServiceManager()方法来获取gDefaultServiceManager对象。对于gDefaultServiceManager对象，如果存在则直接返回；如果不存在则创建该对象，创建过程包括调用open()打开binder驱动设备，利用mmap()映射内核的地址空间。

### 流程图



### defaultServiceManager

g

[-> IServiceManager.cpp]

sp<IServiceManager> defaultServiceManager()

{

if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

{

AutoMutex \_l(gDefaultServiceManagerLock); //加锁

while (gDefaultServiceManager == NULL) {

//【见下文小节二,三,四】

gDefaultServiceManager = interface\_cast<IServiceManager>(

ProcessState::self()->getContextObject(NULL));

if (gDefaultServiceManager == NULL)

sleep(1);

}

}

return gDefaultServiceManager;

}

获取ServiceManager对象采用单例模式，当gDefaultServiceManager存在，则直接返回，否则创建一个新对象。 发现与一般的单例模式不太一样，里面多了一层while循环

这是google 在2013年1月Todd Poynor提交的修改。当尝试创建或获取ServiceManager时，ServiceManager可能尚未准备就绪，这时通过sleep 1秒后，循环尝试获取直到成功。gDefaultServiceManager的创建过程,可分解为以下3个步骤：

ProcessState::self()：用于获取ProcessState对象(也是单例模式)，每个进程有且只有一个ProcessState对象，存在则直接返回，不存在则创建，详情见【小节二】;

getContextObject()： 用于获取BpBinder对象，对于handle=0的BpBinder对象，存在则直接返回，不存在才创建，详情见【小节三】;

interface\_cast<IServiceManager>()：用于获取BpServiceManager对象，详情见【小节四】;

**二. 获取ProcessState对象**

**2.1 ProcessState::self**

[-> ProcessState.cpp]

sp<ProcessState> ProcessState::self()

{

Mutex::Autolock \_l(gProcessMutex);

if (gProcess != NULL) {

return gProcess;

}

//实例化ProcessState 【见小节2.2】

gProcess = new ProcessState;

return gProcess;

}

获得ProcessState对象: 这也是**单例模式**，从而保证每一个进程只有一个ProcessState对象。其中gProcess和gProcessMutex是保存在Static.cpp类的全局变量。

**2.2 初始化ProcessState**

[-> ProcessState.cpp]

ProcessState::ProcessState()

: mDriverFD(open\_driver()) // 打开Binder驱动【见小节2.3】

, mVMStart(MAP\_FAILED)

, mThreadCountLock(PTHREAD\_MUTEX\_INITIALIZER)

, mThreadCountDecrement(PTHREAD\_COND\_INITIALIZER)

, mExecutingThreadsCount(0)

, mMaxThreads(DEFAULT\_MAX\_BINDER\_THREADS)

, mManagesContexts(false)

, mBinderContextCheckFunc(NULL)

, mBinderContextUserData(NULL)

, mThreadPoolStarted(false)

, mThreadPoolSeq(1)

{

if (mDriverFD >= 0) {

//采用内存映射函数mmap，给binder分配一块虚拟地址空间,用来接收事务

mVMStart = mmap(0, BINDER\_VM\_SIZE, PROT\_READ, MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0);

if (mVMStart == MAP\_FAILED) {

close(mDriverFD); //没有足够空间分配给/dev/binder,则关闭驱动

mDriverFD = -1;

}

}

}

* ProcessState的单例模式的惟一性，因此一个进程只打开binder设备一次,其中ProcessState的成员变量mDriverFD记录binder驱动的fd，用于访问binder设备。
* BINDER\_VM\_SIZE = (1\*1024\*1024) - (4096 \*2), binder分配的默认内存大小为1M-8k。
* DEFAULT\_MAX\_BINDER\_THREADS = 15，binder默认的最大可并发访问的线程数为16。

**2.3 open\_driver**

[-> ProcessState.cpp]

static int open\_driver()

{

// 打开/dev/binder设备，建立与内核的Binder驱动的交互通道

int fd = open("/dev/binder", O\_RDWR);

if (fd >= 0) {

fcntl(fd, F\_SETFD, FD\_CLOEXEC);

int vers = 0;

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

if (result == -1) {

close(fd);

fd = -1;

}

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

close(fd);

fd = -1;

}

size\_t maxThreads = DEFAULT\_MAX\_BINDER\_THREADS;

// 通过ioctl设置binder驱动，能支持的最大线程数

result = ioctl(fd, BINDER\_SET\_MAX\_THREADS, &maxThreads);

if (result == -1) {

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

}

} else {

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

}

return fd;

}

open\_driver作用是打开/dev/binder设备，设定binder支持的最大线程数。关于binder驱动的相应方法，见文章[Binder Driver初探](http://gityuan.com/2015/11/01/binder-driver/)。

**三. 获取BpBinder对象**

**3.1 getContextObject**

[-> ProcessState.cpp]

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

{

return getStrongProxyForHandle(0); //【见小节3.2】

}

获取handle=0的IBinder

**3.2 getStrongProxyForHandle**

[-> ProcessState.cpp]

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

{

sp<IBinder> result;

AutoMutex \_l(mLock);

//查找handle对应的资源项【见小节3.3】

handle\_entry\* e = lookupHandleLocked(handle);

if (e != NULL) {

IBinder\* b = e->binder;

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

if (handle == 0) {

Parcel data;

//通过ping操作测试binder是否准备就绪

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

0, IBinder::PING\_TRANSACTION, data, NULL, 0);

if (status == DEAD\_OBJECT)

return NULL;

}

//当handle值所对应的IBinder不存在或弱引用无效时，则创建BpBinder对象【见小节3.4】

b = new BpBinder(handle);

e->binder = b;

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

result = b;

} else {

result.force\_set(b);

e->refs->decWeak(this);

}

}

return result;

}

当handle值所对应的IBinder不存在或弱引用无效时会创建BpBinder，否则直接获取。 针对handle==0的特殊情况，通过PING\_TRANSACTION来判断是否准备就绪。如果在context manager还未生效前，一个BpBinder的本地引用就已经被创建，那么驱动将无法提供context manager的引用。

**3.3 lookupHandleLocked**

[-> ProcessState.cpp]

ProcessState::handle\_entry\* ProcessState::lookupHandleLocked(int32\_t handle)

{

const size\_t N=mHandleToObject.size();

//当handle大于mHandleToObject的长度时，进入该分支

if (N <= (size\_t)handle) {

handle\_entry e;

e.binder = NULL;

e.refs = NULL;

//从mHandleToObject的第N个位置开始，插入(handle+1-N)个e到队列中

status\_t err = mHandleToObject.insertAt(e, N, handle+1-N);

if (err < NO\_ERROR) return NULL;

}

return &mHandleToObject.editItemAt(handle);

}

根据handle值来查找对应的handle\_entry,handle\_entry是一个结构体，里面记录IBinder和weakref\_type两个指针。当handle大于mHandleToObject的Vector长度时，则向该Vector中添加(handle+1-N)个handle\_entry结构体，然后再返回handle向对应位置的handle\_entry结构体指针。

**3.4 创建BpBinder**

[-> BpBinder.cpp]

BpBinder::BpBinder(int32\_t handle)

: mHandle(handle)

, mAlive(1)

, mObitsSent(0)

, mObituaries(NULL)

{

extendObjectLifetime(OBJECT\_LIFETIME\_WEAK); //延长对象的生命时间

IPCThreadState::self()->incWeakHandle(handle); //handle所对应的bindle弱引用 + 1

}

创建BpBinder对象中会将handle相对应Binder的弱引用增加1.

**四. 获取BpServiceManager**

**4.1 interface\_cast**

[-> IInterface.h]

template<typename INTERFACE>

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

{

return INTERFACE::asInterface(obj); //【见小节4.2】

}

这是一个模板函数，可得出，interface\_cast<IServiceManager>() 等价于 IServiceManager::asInterface()。接下来,再来说说asInterface()函数的具体功能。

**4.2 IServiceManager::asInterface**

对于asInterface()函数，通过搜索代码，你会发现根本找不到这个方法是在哪里定义这个函数的, 其实是通过模板函数来定义的，通过下面两个代码完成的：

//位于IServiceManager.h文件 【见小节4.3】

DECLARE\_META\_INTERFACE(ServiceManager)

//位于IServiceManager.cpp文件 【见小节4.4】

IMPLEMENT\_META\_INTERFACE(ServiceManager,"android.os.IServiceManager")

接下来，再说说这两行代码分别完成的功能：

**4.3 DECLARE\_META\_INTERFACE**

[-> IInterface.h]

#define DECLARE\_META\_INTERFACE(INTERFACE) \

static const android::String16 descriptor; \

static android::sp<I##INTERFACE> asInterface( \

const android::sp<android::IBinder>& obj); \

virtual const android::String16& getInterfaceDescriptor() const; \

I##INTERFACE(); \

virtual ~I##INTERFACE(); \

位于IServiceManager.h文件中,INTERFACE=ServiceManager展开即可得：

[-> IServiceManager.h]

static const android::String16 descriptor;

static android::sp< IServiceManager > asInterface(const android::sp<android::IBinder>& obj)

virtual const android::String16& getInterfaceDescriptor() const;

IServiceManager ();

virtual ~IServiceManager();

该过程主要是声明asInterface(),getInterfaceDescriptor()方法.

**4.4 IMPLEMENT\_META\_INTERFACE**

[-> IInterface.h]

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

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

const android::String16& \

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

return I##INTERFACE::descriptor; \

} \

android::sp<I##INTERFACE> I##INTERFACE::asInterface( \

const android::sp<android::IBinder>& obj) \

{ \

android::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; \

} \

I##INTERFACE::I##INTERFACE() { } \

I##INTERFACE::~I##INTERFACE() { } \

位于IServiceManager.cpp文件中,INTERFACE=ServiceManager, NAME=”android.os.IServiceManager”展开即可得：

[-> IServiceManager.cpp]

const android::String16 IServiceManager::descriptor(“android.os.IServiceManager”);

const android::String16& IServiceManager::getInterfaceDescriptor() const

{

return IServiceManager::descriptor;

}

android::sp<IServiceManager> IServiceManager::asInterface(const android::sp<android::IBinder>& obj)

{

android::sp<IServiceManager> intr;

if(obj != NULL) {

intr = static\_cast<IServiceManager \*>(

obj->queryLocalInterface(IServiceManager::descriptor).get());

if (intr == NULL) {

intr = new BpServiceManager(obj); //【见小节4.5】

}

}

return intr;

}

IServiceManager::IServiceManager () { }

IServiceManager::~ IServiceManager() { }

不难发现，[小节4.2]的IServiceManager::asInterface() 等价于 new BpServiceManager()。在这里，更确切地说应该是new BpServiceManager(BpBinder)。

**4.5 BpServiceManager实例化**

创建BpServiceManager对象的过程，会先初始化父类对象：

**4.5.1 BpServiceManager初始化**

[-> IServiceManager.cpp]

BpServiceManager(const sp<IBinder>& impl)

: BpInterface<IServiceManager>(impl)

{ }

**4.5.2 BpInterface初始化**

[-> IInterface.h]

inline BpInterface<INTERFACE>::BpInterface(const sp<IBinder>& remote)

:BpRefBase(remote)

{ }

**4.5.3 BpRefBase初始化**

[-> Binder.cpp]

BpRefBase::BpRefBase(const sp<IBinder>& o)

: mRemote(o.get()), mRefs(NULL), mState(0)

{

extendObjectLifetime(OBJECT\_LIFETIME\_WEAK);

if (mRemote) {

mRemote->incStrong(this);

mRefs = mRemote->createWeak(this);

}

}

new BpServiceManager()，在初始化过程中，比较重要工作的是类BpRefBase的mRemote指向new BpBinder(0)，从而BpServiceManager能够利用Binder进行通过通信。

**五. 总结**

defaultServiceManager 等价于 new BpServiceManager(new BpBinder(0));

ProcessState::self()主要工作：

* 调用open()，打开/dev/binder驱动设备；
* 再利用mmap()，创建大小为1M-8K的内存地址空间；
* 设定当前进程最大的最大并发Binder线程个数为16。

BpServiceManager巧妙将通信层与业务层逻辑合为一体，

* 通过继承接口IServiceManager实现了接口中的业务逻辑函数；
* 通过成员变量mRemote= new BpBinder(0)进行Binder通信工作。
* BpBinder通过handler来指向所对应BBinder, 在整个Binder系统中handle=0代表ServiceManager所对应的BBinder。

**5.1 模板函数**

Native层的Binder架构,通过如下两个宏,非常方便地创建了new Bp##INTERFACE(obj):

//用于申明asInterface(),getInterfaceDescriptor()

#define DECLARE\_META\_INTERFACE(INTERFACE)

#define IMPLEMENT\_META\_INTERFACE(INTERFACE, NAME) //用于实现上述两个方法

例如:

// 实现BPServiceManager对象

IMPLEMENT\_META\_INTERFACE(ServiceManager,"android.os.IServiceManager")

等价于:

const android::String16 IServiceManager::descriptor(“android.os.IServiceManager”);

const android::String16& IServiceManager::getInterfaceDescriptor() const

{

return IServiceManager::descriptor;

}

android::sp<IServiceManager> IServiceManager::asInterface(const android::sp<android::IBinder>& obj)

{

android::sp<IServiceManager> intr;

if(obj != NULL) {

intr = static\_cast<IServiceManager \*>(

obj->queryLocalInterface(IServiceManager::descriptor).get());

if (intr == NULL) {

intr = new BpServiceManager(obj);

}

}

return intr;

}

IServiceManager::IServiceManager () { }

IServiceManager::~ IServiceManager() { }

# 注册服务

F

framework/native/libs/binder/

- Binder.cpp

- BpBinder.cpp

- IPCThreadState.cpp

- ProcessState.cpp

- IServiceManager.cpp

- IInterface.cpp

- Parcel.cpp

frameworks/native/include/binder/

- IInterface.h (包括BnInterface, BpInterface)

## 概述

### media服务注册main\_mediaserver

media入口函数是main\_mediaserver.cpp中的main()方法，代码如下：

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

{

...

signal(SIGPIPE, SIG\_IGN);

//获得ProcessState实例对象【见小节2.1】

sp<ProcessState> proc(ProcessState::self());

//获取BpServiceManager对象

sp<IServiceManager> sm = defaultServiceManager();

AudioFlinger::instantiate();

//注册多媒体服务 【见小节3.1】

MediaPlayerService::instantiate();

ResourceManagerService::instantiate();

CameraService::instantiate();

AudioPolicyService::instantiate();

SoundTriggerHwService::instantiate();

RadioService::instantiate();

registerExtensions();

**//启动Binder线程池**

ProcessState::self()->startThreadPool();

**//当前线程加入到线程池**

IPCThreadState::self()->joinThreadPool();

}

F

过程说明:

1. [获取ServiceManager](http://gityuan.com/2015/11/08/binder-get-sm/#defaultservicemanager): 讲解了defaultServiceManager()返回的是BpServiceManager对象， 用于跟servicemanager进程通信;
2. [理解Binder线程池的管理](http://gityuan.com/2016/10/29/binder-thread-pool/), 讲解了startThreadPool和joinThreadPool过程.

本文的重点就是讲解Native层服务注册的过程.

### 类图

在Native层的服务以media服务为例，来说一说服务注册过程，先来看看media的整个的类关系图。 点击查看大图



图解：

蓝色代表的是注册MediaPlayerService服务所涉及的类

绿色代表的是Binder架构中与Binder驱动通信过程中的最为核心的两个类；

紫色代表的是注册服务和获取服务的公共接口/父类；

### 时序图

Fdg

先通过一幅图来说说，media服务启动过程是如何向servicemanager注册服务的。

点击查看大图



## ProcessState

ProcessState.cpp

### ProcessState::self

ProcessState.cpp

sp<ProcessState> ProcessState::self()

{

Mutex::Autolock \_l(gProcessMutex);

if (gProcess != NULL) {

return gProcess;

}

//实例化ProcessState 【见小节2.2】

gProcess = new ProcessState;

return gProcess;

}

获得ProcessState对象: 这也是单例模式，从而保证每一个进程只有一个ProcessState对象。其中gProcess和gProcessMutex是保存在Static.cpp类的全局变量。

### ProcessState()构造函数

G

[-> ProcessState.cpp]

ProcessState::ProcessState()

: mDriverFD(open\_driver()) // 打开Binder驱动【见小节2.3】

, mVMStart(MAP\_FAILED)

, mThreadCountLock(PTHREAD\_MUTEX\_INITIALIZER)

, mThreadCountDecrement(PTHREAD\_COND\_INITIALIZER)

, mExecutingThreadsCount(0)

, mMaxThreads(DEFAULT\_MAX\_BINDER\_THREADS)

, mManagesContexts(false)

, mBinderContextCheckFunc(NULL)

, mBinderContextUserData(NULL)

, mThreadPoolStarted(false)

, mThreadPoolSeq(1)

{

if (mDriverFD >= 0) {

//采用内存映射函数mmap，给binder分配一块虚拟地址空间【见小节2.4】

mVMStart = mmap(0, BINDER\_VM\_SIZE, PROT\_READ, MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0);

if (mVMStart == MAP\_FAILED) {

close(mDriverFD); //没有足够空间分配给/dev/binder,则关闭驱动

mDriverFD = -1;

}

}

}

* ProcessState的单例模式的惟一性，因此一个进程只打开binder设备一次,其中ProcessState的成员变量mDriverFD记录binder驱动的fd，用于访问binder设备。
* BINDER\_VM\_SIZE = (1\*1024\*1024) - (4096 \*2), binder分配的默认内存大小为1M-8k。
* DEFAULT\_MAX\_BINDER\_THREADS = 15，binder默认的最大可并发访问的线程数为16。

#### open\_driver

[-> ProcessState.cpp]

static int open\_driver()

{

// 打开/dev/binder设备，建立与内核的Binder驱动的交互通道

int fd = open("/dev/binder", O\_RDWR);

if (fd >= 0) {

fcntl(fd, F\_SETFD, FD\_CLOEXEC);

int vers = 0;

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

if (result == -1) {

close(fd);

fd = -1;

}

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

close(fd);

fd = -1;

}

size\_t maxThreads = DEFAULT\_MAX\_BINDER\_THREADS;

// 通过ioctl设置binder驱动，能支持的最大线程数

result = ioctl(fd, BINDER\_SET\_MAX\_THREADS, &maxThreads);

if (result == -1) {

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

}

} else {

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

}

return fd;

}

open\_driver作用是打开/dev/binder设备，设定binder支持的最大线程数。关于binder驱动的相应方法，见文章[Binder Driver初探](http://gityuan.com/2015/11/01/binder-driver/)。

ProcessState采用单例模式，保证每一个进程都只打开一次Binder Driver。

#### mmap

d】

//原型

void\* mmap(void\* addr, size\_t size, int prot, int flags, int fd, off\_t offset) //此处 mmap(0, BINDER\_VM\_SIZE, PROT\_READ, MAP\_PRIVATE | MAP\_NORESERVE, mDriverFD, 0);

参数说明：

addr: 代表映射到进程地址空间的起始地址，当值等于0则由内核选择合适地址，此处为0；

size: 代表需要映射的内存地址空间的大小，此处为1M-8K；

prot: 代表内存映射区的读写等属性值，此处为PROT\_READ(可读取);

flags: 标志位，此处为MAP\_PRIVATE(私有映射，多进程间不共享内容的改变)和 MAP\_NORESERVE(不保留交换空间)

fd: 代表mmap所关联的文件描述符，此处为mDriverFD；

offset：偏移量，此处为0。

mmap()经过系统调用，执行[binder\_mmap](http://gityuan.com/2015/11/01/binder-driver/)过程。

## 服务注册

### Instantiate

G

[-> MediaPlayerService.cpp]

void MediaPlayerService::instantiate() {

//注册服务【见小节3.2】

defaultServiceManager()->addService(String16("media.player"), new MediaPlayerService());

}

注册服务MediaPlayerService：由defaultServiceManager()返回的是BpServiceManager，同时会创建ProcessState对象和BpBinder对象。 故此处等价于调用BpServiceManager->addService。其中MediaPlayerService位于libmediaplayerservice库.

### BpSM.addService

G

[-> IServiceManager.cpp ::BpServiceManager]

virtual status\_t addService(const String16& name, const sp<IBinder>& service, bool allowIsolated) {

Parcel data, reply; //Parcel是数据通信包

//写入头信息"android.os.IServiceManager"

data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());

data.writeString16(name); // name为 "media.player"

data.writeStrongBinder(service); // MediaPlayerService对象【见小节3.2.1】

data.writeInt32(allowIsolated ? 1 : 0); // allowIsolated= false

//remote()指向的是BpBinder对象【见小节3.3】

status\_t err = remote()->transact(ADD\_SERVICE\_TRANSACTION, data, &reply);

return err == NO\_ERROR ? reply.readExceptionCode() : err;

}

服务注册过程：向ServiceManager注册服务MediaPlayerService，服务名为”media.player”

#### writeStrongBinder

g

[-> parcel.cpp]

status\_t Parcel::writeStrongBinder(const sp<IBinder>& val)

{

return flatten\_binder(ProcessState::self(), val, this);

}

#### flatten\_binder

g

status\_t flatten\_binder(const sp<ProcessState>& /\*proc\*/,

const sp<IBinder>& binder, Parcel\* out)

{

flat\_binder\_object obj;

obj.flags = 0x7f | FLAT\_BINDER\_FLAG\_ACCEPTS\_FDS;

if (binder != NULL) {

IBinder \*local = binder->localBinder(); //本地Binder不为空

if (!local) {

BpBinder \*proxy = binder->remoteBinder();

const int32\_t handle = proxy ? proxy->handle() : 0;

obj.type = BINDER\_TYPE\_HANDLE;

obj.binder = 0;

obj.handle = handle;

obj.cookie = 0;

} else { //进入该分支

obj.type = BINDER\_TYPE\_BINDER;

obj.binder = reinterpret\_cast<uintptr\_t>(local->getWeakRefs());

obj.cookie = reinterpret\_cast<uintptr\_t>(local);

}

} else {

...

}

//【见小节3.2.3】

return finish\_flatten\_binder(binder, obj, out);

}

将Binder对象扁平化，转换成flat\_binder\_object对象。

对于Binder实体，则cookie记录Binder实体的指针；

对于Binder代理，则用handle记录Binder代理的句柄；

关于localBinder，代码见Binder.cpp。

BBinder\* **BBinder**::localBinder()

{

return this;

}

BBinder\* **IBinder**::localBinder()

{

return NULL;

}

#### finish\_flatten\_binder

f

inline static status\_t finish\_flatten\_binder(

const sp<IBinder>& , const flat\_binder\_object& flat, Parcel\* out)

{

return out->writeObject(flat, false);

}

将flat\_binder\_object写入out。

### BpBinder::transact

Fg

[-> BpBinder.cpp]

status\_t BpBinder::transact(

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

{

if (mAlive) {

// code=ADD\_SERVICE\_TRANSACTION【见小节3.4】

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

mHandle, code, data, reply, flags);

if (status == DEAD\_OBJECT) mAlive = 0;

return status;

}

return DEAD\_OBJECT;

}

Binder代理类调用transact()方法，真正工作还是交给IPCThreadState来进行transact工作。先来 看看IPCThreadState::self的过程。

#### IPCThreadState::self

F

[-> IPCThreadState.cpp]

IPCThreadState\* IPCThreadState::self()

{

if (gHaveTLS) {

restart:

const pthread\_key\_t k = gTLS;

IPCThreadState\* st = (IPCThreadState\*)pthread\_getspecific(k);

if (st) return st;

return new IPCThreadState; //初始IPCThreadState 【见小节3.3.2】

}

if (gShutdown) return NULL;

pthread\_mutex\_lock(&gTLSMutex);

if (!gHaveTLS) { //首次进入gHaveTLS为false

if (pthread\_key\_create(&gTLS, threadDestructor) != 0) { //创建线程的TLS

pthread\_mutex\_unlock(&gTLSMutex);

return NULL;

}

gHaveTLS = true;

}

pthread\_mutex\_unlock(&gTLSMutex);

goto restart;

}

TLS是指Thread local storage(线程本地储存空间)，每个线程都拥有自己的TLS，并且是私有空间，线程之间不会共享。通过pthread\_getspecific/pthread\_setspecific函数可以获取/设置这些空间中的内容。从线程本地存储空间中获得保存在其中的IPCThreadState对象

#### IPCThreadState初始化

D

[-> IPCThreadState.cpp]

IPCThreadState::IPCThreadState()

: mProcess(ProcessState::self()),

mMyThreadId(gettid()),

mStrictModePolicy(0),

mLastTransactionBinderFlags(0)

{

pthread\_setspecific(gTLS, this);

clearCaller();

mIn.setDataCapacity(256);

mOut.setDataCapacity(256);

}

每个线程都有一个IPCThreadState，每个IPCThreadState中都有一个mIn、一个mOut。成员变量mProcess保存了ProcessState变量(每个进程只有一个)。

mIn 用来接收来自Binder设备的数据，默认大小为256字节；

mOut用来存储发往Binder设备的数据，默认大小为256字节。

### IPC::transact

[-> IPCThreadState.cpp]

status\_t IPCThreadState::transact(int32\_t handle,

uint32\_t code, const Parcel& data,

Parcel\* reply, uint32\_t flags)

{

status\_t err = data.errorCheck(); //数据错误检查

flags |= TF\_ACCEPT\_FDS;

....

if (err == NO\_ERROR) { // 传输数据 【见小节3.5】

err = writeTransactionData(BC\_TRANSACTION, flags, handle, code, data, NULL);

}

...

if ((flags & TF\_ONE\_WAY) == 0) {

if (reply) {

//等待响应 【见小节3.6】

err = waitForResponse(reply);

} else {

Parcel fakeReply;

err = waitForResponse(&fakeReply);

}

} else {

//oneway，则不需要等待reply的场景

err = waitForResponse(NULL, NULL);

}

return err;

}

IPCThreadState进行transact事务处理分3部分：

errorCheck() //数据错误检查

writeTransactionData() // 传输数据

waitForResponse() //f等待响应

### IPC.writeTransactionData

F

[-> IPCThreadState.cpp]

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.ptr = 0;

tr.target.handle = handle; // handle = 0

tr.code = code; // code = ADD\_SERVICE\_TRANSACTION

tr.flags = binderFlags; // binderFlags = 0

tr.cookie = 0;

tr.sender\_pid = 0;

tr.sender\_euid = 0;

// data为记录Media服务信息的Parcel对象

const status\_t err = data.errorCheck();

if (err == NO\_ERROR) {

tr.data\_size = data.ipcDataSize(); // mDataSize

tr.data.ptr.buffer = data.ipcData(); //mData

tr.offsets\_size = data.ipcObjectsCount()\*sizeof(binder\_size\_t); //mObjectsSize

tr.data.ptr.offsets = data.ipcObjects(); //mObjects

} else if (statusBuffer) {

...

} else {

return (mLastError = err);

}

mOut.writeInt32(cmd); //cmd = BC\_TRANSACTION

mOut.write(&tr, sizeof(tr)); //写入binder\_transaction\_data数据

return NO\_ERROR;

}

其中handle的值用来标识目的端，注册服务过程的目的端为service manager，此处handle=0所对应的是binder\_context\_mgr\_node对象，正是service manager所对应的binder实体对象。binder\_transaction\_data结构体是binder驱动通信的数据结构，该过程最终是把Binder请求码BC\_TRANSACTION和binder\_transaction\_data结构体写入到mOut。

transact过程，先写完binder\_transaction\_data数据，其中Parcel data的重要成员变量：

mDataSize:保存再data\_size，binder\_transaction的数据大小；

mData: 保存在ptr.buffer, binder\_transaction的数据的起始地址；

mObjectsSize:保存在ptr.offsets\_size，记录着flat\_binder\_object结构体的个数；

mObjects: 保存在offsets, 记录着flat\_binder\_object结构体在数据偏移量；

接下来执行waitForResponse()方法。

### IPC.waitForResponse

F

IPCThreadState.cpp]

status\_t IPCThreadState::waitForResponse(Parcel \*reply, status\_t \*acquireResult)

{

int32\_t cmd;

int32\_t err;

while (1) {

if ((err=talkWithDriver()) < NO\_ERROR) break; // 【见小节3.7】

...

if (mIn.dataAvail() == 0) continue;

cmd = mIn.readInt32();

switch (cmd) {

case BR\_TRANSACTION\_COMPLETE: ...

case BR\_DEAD\_REPLY: ...

case BR\_FAILED\_REPLY: ...

case BR\_ACQUIRE\_RESULT: ...

case BR\_REPLY: ...

goto finish;

default:

err = executeCommand(cmd); //【见小节3.x】

if (err != NO\_ERROR) goto finish;

break;

}

}

...

return err;

}

在waitForResponse过程, 首先执行BR\_TRANSACTION\_COMPLETE；另外，目标进程收到事务后，处理BR\_TRANSACTION事务。 然后发送给当前进程，再执行BR\_REPLY命令

### IPC.talkWithDriver

G

[-> IPCThreadState.cpp]

status\_t IPCThreadState::talkWithDriver(bool doReceive)

{

...

binder\_write\_read bwr;

const bool needRead = mIn.dataPosition() >= mIn.dataSize();

const size\_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;

bwr.write\_size = outAvail;

bwr.write\_buffer = (uintptr\_t)mOut.data();

if (doReceive && needRead) {

//接收数据缓冲区信息的填充。如果以后收到数据，就直接填在mIn中了。

bwr.read\_size = mIn.dataCapacity();

bwr.read\_buffer = (uintptr\_t)mIn.data();

} else {

bwr.read\_size = 0;

bwr.read\_buffer = 0;

}

//当读缓冲和写缓冲都为空，则直接返回

if ((bwr.write\_size == 0) && (bwr.read\_size == 0)) return NO\_ERROR;

bwr.write\_consumed = 0;

bwr.read\_consumed = 0;

status\_t err;

do {

//通过ioctl不停的读写操作，跟Binder Driver进行通信

if (ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr) >= 0)

err = NO\_ERROR;

...

} while (err == -EINTR); //当被中断，则继续执行

...

return err;

}

[binder\_write\_read结构体](http://gityuan.com/2015/11/01/binder-driver/#binderwriteread)用来与Binder设备交换数据的结构, 通过ioctl与mDriverFD通信，是真正与Binder驱动进行数据读写交互的过程。 主要是操作mOut和mIn变量。

ioctl()经过系统调用后进入Binder Driver.

## Binder Driver

ioctl -> binder\_ioctl -> binder\_ioctl\_write\_read

### binder\_ioctl\_write\_read

f

[-> binder.c]

static int binder\_ioctl\_write\_read(struct file \*filp,

unsigned int cmd, unsigned long arg,

struct binder\_thread \*thread)

{

struct binder\_proc \*proc = filp->private\_data;

void \_\_user \*ubuf = (void \_\_user \*)arg;

struct binder\_write\_read bwr;

//将用户空间bwr结构体拷贝到内核空间

copy\_from\_user(&bwr, ubuf, sizeof(bwr));

...

if (bwr.write\_size > 0) {

//将数据放入目标进程【见小节4.2】

ret = binder\_thread\_write(proc, thread,

bwr.write\_buffer,

bwr.write\_size,

&bwr.write\_consumed);

...

}

if (bwr.read\_size > 0) {

//读取自己队列的数据 【见小节】

ret = binder\_thread\_read(proc, thread, bwr.read\_buffer,

bwr.read\_size,

&bwr.read\_consumed,

filp->f\_flags & O\_NONBLOCK);

if (!list\_empty(&proc->todo))

wake\_up\_interruptible(&proc->wait);

...

}

//将内核空间bwr结构体拷贝到用户空间

copy\_to\_user(ubuf, &bwr, sizeof(bwr));

...

}

### binder\_thread\_write

d

static int binder\_thread\_write(struct binder\_proc \*proc,

struct binder\_thread \*thread,

binder\_uintptr\_t binder\_buffer, size\_t size,

binder\_size\_t \*consumed)

{

uint32\_t cmd;

void \_\_user \*buffer = (void \_\_user \*)(uintptr\_t)binder\_buffer;

void \_\_user \*ptr = buffer + \*consumed;

void \_\_user \*end = buffer + size;

while (ptr < end && thread->return\_error == BR\_OK) {

//拷贝用户空间的cmd命令，此时为BC\_TRANSACTION

if (get\_user(cmd, (uint32\_t \_\_user \*)ptr)) -EFAULT;

ptr += sizeof(uint32\_t);

switch (cmd) {

case BC\_TRANSACTION:

case BC\_REPLY: {

struct binder\_transaction\_data tr;

//拷贝用户空间的binder\_transaction\_data

if (copy\_from\_user(&tr, ptr, sizeof(tr))) return -EFAULT;

ptr += sizeof(tr);

// 见小节4.3】

binder\_transaction(proc, thread, &tr, cmd == BC\_REPLY);

break;

}

...

}

\*consumed = ptr - buffer;

}

return 0;

}

### binder\_transaction

f

static void binder\_transaction(struct binder\_proc \*proc,

struct binder\_thread \*thread,

struct binder\_transaction\_data \*tr, int reply){

struct binder\_transaction \*t;

struct binder\_work \*tcomplete;

...

if (reply) {

...

}else {

if (tr->target.handle) {

...

} else {

// handle=0则找到servicemanager实体

target\_node = binder\_context\_mgr\_node;

}

//target\_proc为servicemanager进程

target\_proc = target\_node->proc;

}

if (target\_thread) {

...

} else {

//找到servicemanager进程的todo队列

target\_list = &target\_proc->todo;

target\_wait = &target\_proc->wait;

}

t = kzalloc(sizeof(\*t), GFP\_KERNEL);

tcomplete = kzalloc(sizeof(\*tcomplete), GFP\_KERNEL);

//非oneway的通信方式，把当前thread保存到transaction的from字段

if (!reply && !(tr->flags & TF\_ONE\_WAY))

t->from = thread;

else

t->from = NULL;

t->sender\_euid = task\_euid(proc->tsk);

t->to\_proc = target\_proc; //此次通信目标进程为servicemanager进程

t->to\_thread = target\_thread;

t->code = tr->code; //此次通信code = ADD\_SERVICE\_TRANSACTION

t->flags = tr->flags; // 此次通信flags = 0

t->priority = task\_nice(current);

//从servicemanager进程中分配buffer

t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,

tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));

t->buffer->allow\_user\_free = 0;

t->buffer->transaction = t;

t->buffer->target\_node = target\_node;

if (target\_node)

binder\_inc\_node(target\_node, 1, 0, NULL); //引用计数加1

offp = (binder\_size\_t \*)(t->buffer->data + ALIGN(tr->data\_size, sizeof(void \*)));

//分别拷贝用户空间的binder\_transaction\_data中ptr.buffer和ptr.offsets到内核

copy\_from\_user(t->buffer->data,

(const void \_\_user \*)(uintptr\_t)tr->data.ptr.buffer, tr->data\_size);

copy\_from\_user(offp,

(const void \_\_user \*)(uintptr\_t)tr->data.ptr.offsets, tr->offsets\_size);

off\_end = (void \*)offp + tr->offsets\_size;

for (; offp < off\_end; offp++) {

struct flat\_binder\_object \*fp;

fp = (struct flat\_binder\_object \*)(t->buffer->data + \*offp);

off\_min = \*offp + sizeof(struct flat\_binder\_object);

switch (fp->type) {

case BINDER\_TYPE\_BINDER:

case BINDER\_TYPE\_WEAK\_BINDER: {

struct binder\_ref \*ref;

//【见4.3.1】

struct binder\_node \*node = binder\_get\_node(proc, fp->binder);

if (node == NULL) {

//服务所在进程 创建binder\_node实体【见4.3.2】

node = binder\_new\_node(proc, fp->binder, fp->cookie);

...

}

//servicemanager进程binder\_ref【见4.3.3】

ref = binder\_get\_ref\_for\_node(target\_proc, node);

...

//调整type为HANDLE类型

if (fp->type == BINDER\_TYPE\_BINDER)

fp->type = BINDER\_TYPE\_HANDLE;

else

fp->type = BINDER\_TYPE\_WEAK\_HANDLE;

fp->binder = 0;

fp->handle = ref->desc; //设置handle值

fp->cookie = 0;

binder\_inc\_ref(ref, fp->type == BINDER\_TYPE\_HANDLE,

&thread->todo);

} break;

case :...

}

if (reply) {

..

} else if (!(t->flags & TF\_ONE\_WAY)) {

//BC\_TRANSACTION 且 非oneway,则设置事务栈信息

t->need\_reply = 1;

t->from\_parent = thread->transaction\_stack;

thread->transaction\_stack = t;

} else {

...

}

//将BINDER\_WORK\_TRANSACTION添加到目标队列，本次通信的目标队列为target\_proc->todo

t->work.type = BINDER\_WORK\_TRANSACTION;

list\_add\_tail(&t->work.entry, target\_list);

//将BINDER\_WORK\_TRANSACTION\_COMPLETE添加到当前线程的todo队列

tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;

list\_add\_tail(&tcomplete->entry, &thread->todo);

//唤醒等待队列，本次通信的目标队列为target\_proc->wait

if (target\_wait)

wake\_up\_interruptible(target\_wait);

return;

}

注册服务的过程，传递的是BBinder对象，故[小节3.2.1]的writeStrongBinder()过程中localBinder不为空， 从而flat\_binder\_object.type等于BINDER\_TYPE\_BINDER。

服务注册过程是在服务所在进程创建binder\_node，在servicemanager进程创建binder\_ref。 对于同一个binder\_node，每个进程只会创建一个binder\_ref对象。

向servicemanager的binder\_proc->todo添加BINDER\_WORK\_TRANSACTION事务，接下来进入ServiceManager进程

#### binder\_get\_node

d

static struct binder\_node \*binder\_get\_node(struct binder\_proc \*proc,

binder\_uintptr\_t ptr)

{

struct rb\_node \*n = proc->nodes.rb\_node;

struct binder\_node \*node;

while (n) {

node = rb\_entry(n, struct binder\_node, rb\_node);

if (ptr < node->ptr)

n = n->rb\_left;

else if (ptr > node->ptr)

n = n->rb\_right;

else

return node;

}

return NULL;

}

从binder\_proc来根据binder指针ptr值，查询相应的binder\_node。

#### binder\_new\_node

d

static struct binder\_node \*binder\_new\_node(struct binder\_proc \*proc,

binder\_uintptr\_t ptr,

binder\_uintptr\_t cookie)

{

struct rb\_node \*\*p = &proc->nodes.rb\_node;

struct rb\_node \*parent = NULL;

struct binder\_node \*node;

... //红黑树位置查找

//给新创建的binder\_node 分配内核空间

node = kzalloc(sizeof(\*node), GFP\_KERNEL);

// 将新创建的node添加到proc红黑树；

rb\_link\_node(&node->rb\_node, parent, p);

rb\_insert\_color(&node->rb\_node, &proc->nodes);

node->debug\_id = ++binder\_last\_id;

node->proc = proc;

node->ptr = ptr;

node->cookie = cookie;

node->work.type = BINDER\_WORK\_NODE; //设置binder\_work的type

INIT\_LIST\_HEAD(&node->work.entry);

INIT\_LIST\_HEAD(&node->async\_todo);

return node;

}

#### binder\_get\_ref\_for\_node

df

static struct binder\_ref \*binder\_get\_ref\_for\_node(struct binder\_proc \*proc,

struct binder\_node \*node)

{

struct rb\_node \*n;

struct rb\_node \*\*p = &proc->refs\_by\_node.rb\_node;

struct rb\_node \*parent = NULL;

struct binder\_ref \*ref, \*new\_ref;

//从refs\_by\_node红黑树，找到binder\_ref则直接返回。

while (\*p) {

parent = \*p;

ref = rb\_entry(parent, struct binder\_ref, rb\_node\_node);

if (node < ref->node)

p = &(\*p)->rb\_left;

else if (node > ref->node)

p = &(\*p)->rb\_right;

else

return ref;

}

//创建binder\_ref

new\_ref = kzalloc\_preempt\_disabled(sizeof(\*ref));

new\_ref->debug\_id = ++binder\_last\_id;

new\_ref->proc = proc; //记录进程信息

new\_ref->node = node; //记录binder节点

rb\_link\_node(&new\_ref->rb\_node\_node, parent, p);

rb\_insert\_color(&new\_ref->rb\_node\_node, &proc->refs\_by\_node);

//计算binder引用的handle值，该值返回给target\_proc进程

new\_ref->desc = (node == binder\_context\_mgr\_node) ? 0 : 1;

//从红黑树最最左边的handle对比，依次递增，直到红黑树遍历结束或者找到更大的handle则结束。

for (n = rb\_first(&proc->refs\_by\_desc); n != NULL; n = rb\_next(n)) {

//根据binder\_ref的成员变量rb\_node\_desc的地址指针n，来获取binder\_ref的首地址

ref = rb\_entry(n, struct binder\_ref, rb\_node\_desc);

if (ref->desc > new\_ref->desc)

break;

new\_ref->desc = ref->desc + 1;

}

// 将新创建的new\_ref 插入proc->refs\_by\_desc红黑树

p = &proc->refs\_by\_desc.rb\_node;

while (\*p) {

parent = \*p;

ref = rb\_entry(parent, struct binder\_ref, rb\_node\_desc);

if (new\_ref->desc < ref->desc)

p = &(\*p)->rb\_left;

else if (new\_ref->desc > ref->desc)

p = &(\*p)->rb\_right;

else

BUG();

}

rb\_link\_node(&new\_ref->rb\_node\_desc, parent, p);

rb\_insert\_color(&new\_ref->rb\_node\_desc, &proc->refs\_by\_desc);

if (node) {

hlist\_add\_head(&new\_ref->node\_entry, &node->refs);

}

return new\_ref;

}

handle值计算方法规律：

每个进程binder\_proc所记录的binder\_ref的handle值是从1开始递增的；

所有进程binder\_proc所记录的handle=0的binder\_ref都指向service manager；

同一个服务的binder\_node在不同进程的binder\_ref的handle值可以不同

## ServiceManager

F

由[Binder系列3—启动ServiceManager](http://gityuan.com/2015/11/07/binder-start-sm/)已介绍其原理，循环在binder\_loop()过程， 会调用binder\_parse()方法

### binder\_parse

servicemanager/binder.c]

int binder\_parse(struct binder\_state \*bs, struct binder\_io \*bio, uintptr\_t ptr, size\_t size, binder\_handler func) {

int r = 1;

uintptr\_t end = ptr + (uintptr\_t) size;

while (ptr < end) {

uint32\_t cmd = \*(uint32\_t \*) ptr;

ptr += sizeof(uint32\_t);

switch(cmd) {

case BR\_TRANSACTION: {

struct binder\_transaction\_data \*txn = (struct binder\_transaction\_data \*) ptr;

...

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); //从txn解析出binder\_io信息

// 收到Binder事务 【见小节5.2】

res = func(bs, txn, &msg, &reply);

// 发送reply事件【见小节5.4】

binder\_send\_reply(bs, &reply, txn->data.ptr.buffer, res);

}

ptr += sizeof(\*txn);

break;

}

case : ...

}

return r;

}

### svcmgr\_handler

fd

[-> service\_manager.c]

int svcmgr\_handler(struct binder\_state \*bs, struct binder\_transaction\_data \*txn, struct binder\_io \*msg, struct binder\_io \*reply) {

struct svcinfo \*si;

uint16\_t \*s;

size\_t len;

uint32\_t handle;

uint32\_t strict\_policy;

int allow\_isolated;

...

strict\_policy = bio\_get\_uint32(msg);

s = bio\_get\_string16(msg, &len);

...

switch(txn->code) {

case SVC\_MGR\_ADD\_SERVICE:

s = bio\_get\_string16(msg, &len);

...

handle = bio\_get\_ref(msg); //获取handle

allow\_isolated = bio\_get\_uint32(msg) ? 1 : 0;

//注册指定服务 【见小节5.3】

if (do\_add\_service(bs, s, len, handle, txn->sender\_euid,

allow\_isolated, txn->sender\_pid))

return -1;

break;

case :...

}

bio\_put\_uint32(reply, 0);

return 0;

}

### do\_add\_service

f

[-> service\_manager.c]

int do\_add\_service(struct binder\_state \*bs,

const uint16\_t \*s, size\_t len,

uint32\_t handle, uid\_t uid, int allow\_isolated,

pid\_t spid)

{

struct svcinfo \*si;

if (!handle || (len == 0) || (len > 127))

return -1;

//权限检查

if (!svc\_can\_register(s, len, spid)) {

return -1;

}

//服务检索

si = find\_svc(s, len);

if (si) {

if (si->handle) {

svcinfo\_death(bs, si); //服务已注册时，释放相应的服务

}

si->handle = handle;

} else {

si = malloc(sizeof(\*si) + (len + 1) \* sizeof(uint16\_t));

if (!si) { //内存不足，无法分配足够内存

return -1;

}

si->handle = handle;

si->len = len;

memcpy(si->name, s, (len + 1) \* sizeof(uint16\_t)); //内存拷贝服务信息

si->name[len] = '\0';

si->death.func = (void\*) svcinfo\_death;

si->death.ptr = si;

si->allow\_isolated = allow\_isolated;

si->next = svclist; // svclist保存所有已注册的服务

svclist = si;

}

//以BC\_ACQUIRE命令，handle为目标的信息，通过ioctl发送给binder驱动

binder\_acquire(bs, handle);

//以BC\_REQUEST\_DEATH\_NOTIFICATION命令的信息，通过ioctl发送给binder驱动，主要用于清理内存等收尾工作。

binder\_link\_to\_death(bs, handle, &si->death);

return 0;

}

svcinfo记录着服务名和handle信息，保存到svclist列表

### binder\_send\_reply

[-> servicemanager/binder.c]

void binder\_send\_reply(struct binder\_state \*bs, struct binder\_io \*reply, binder\_uintptr\_t buffer\_to\_free, int status) {

struct {

uint32\_t cmd\_free;

binder\_uintptr\_t buffer;

uint32\_t cmd\_reply;

struct binder\_transaction\_data txn;

} \_\_attribute\_\_((packed)) data;

data.cmd\_free = BC\_FREE\_BUFFER; //free buffer命令

data.buffer = buffer\_to\_free;

data.cmd\_reply = BC\_REPLY; // reply命令

data.txn.target.ptr = 0;

data.txn.cookie = 0;

data.txn.code = 0;

if (status) {

...

} else {

data.txn.flags = 0;

data.txn.data\_size = reply->data - reply->data0;

data.txn.offsets\_size = ((char\*) reply->offs) - ((char\*) reply->offs0);

data.txn.data.ptr.buffer = (uintptr\_t)reply->data0;

data.txn.data.ptr.offsets = (uintptr\_t)reply->offs0;

}

//向Binder驱动通信

binder\_write(bs, &data, sizeof(data));

}

binder\_write进入binder驱动后，将BC\_FREE\_BUFFER和BC\_REPLY命令协议发送给Binder驱动， 向client端发送reply.

## 总结

F

服务注册过程(addService)核心功能：在服务所在进程创建binder\_node，在servicemanager进程创建binder\_ref。 其中binder\_ref的desc再同一个进程内是唯一的：

每个进程binder\_proc所记录的binder\_ref的handle值是从1开始递增的；

所有进程binder\_proc所记录的handle=0的binder\_ref都指向service manager；

同一个服务的binder\_node在不同进程的binder\_ref的handle值可以不同；

Media服务注册的过程涉及到MediaPlayerService(作为Client进程)和Service Manager(作为Service进程)，通信流程图如下所示：



过程分析：

1. MediaPlayerService进程调用ioctl()向Binder驱动发送IPC数据，该过程可以理解成一个事务binder\_transaction(记为T1)，执行当前操作的线程binder\_thread(记为thread1)，则T1->from\_parent=NULL，T1->from = thread1，thread1->transaction\_stack=T1。其中IPC数据内容包含：
   * Binder协议为BC\_TRANSACTION；
   * Handle等于0；
   * RPC代码为ADD\_SERVICE；
   * RPC数据为”media.player”。
2. Binder驱动收到该Binder请求，生成BR\_TRANSACTION命令，选择目标处理该请求的线程，即ServiceManager的binder线程(记为thread2)，则 T1->to\_parent = NULL，T1->to\_thread = thread2。并将整个binder\_transaction数据(记为T2)插入到目标线程的todo队列；
3. Service Manager的线程thread2收到T2后，调用服务注册函数将服务”media.player”注册到服务目录中。当服务注册完成后，生成IPC应答数据(BC\_REPLY)，T2->form\_parent = T1，T2->from = thread2, thread2->transaction\_stack = T2。
4. Binder驱动收到该Binder应答请求，生成BR\_REPLY命令，T2->to\_parent = T1，T2->to\_thread = thread1, thread1->transaction\_stack = T2。 在MediaPlayerService收到该命令后，知道服务注册完成便可以正常使用。

整个过程中，BC\_TRANSACTION和BR\_TRANSACTION过程是一个完整的事务过程；BC\_REPLY和BR\_REPLY是一个完整的事务过程。 到此，其他进行便可以获取该服务，使用服务提供的方法，下一篇文章将会讲述[如何获取服务](http://gityuan.com/2015/11/15/binder-get-service/)。

<http://gityuan.com/2015/11/14/binder-add-service/>

# 获取服务(getService

在Native层的服务注册，我们选择以media为例来展开讲解，先来看看media的类关系图

## 类图



图解：

* 蓝色: 代表获取MediaPlayerService服务相关的类；
* 绿色: 代表Binder架构中与Binder驱动通信过程中的最为核心的两个类；
* 紫色: 代表[注册服务](http://gityuan.com/2015/11/14/binder-add-service/)和获取服务的公共接口/父类；

## 获取Media服务

### getMediaPlayerService

g

[-> framework/av/media/libmedia/IMediaDeathNotifier.cpp]

sp<IMediaPlayerService>&

IMediaDeathNotifier::getMediaPlayerService()

{

Mutex::Autolock \_l(sServiceLock);

if (sMediaPlayerService == 0) {

sp<IServiceManager> sm = defaultServiceManager(); //获取ServiceManager

sp<IBinder> binder;

do {

//获取名为"media.player"的服务 【见2.2】

binder = sm->getService(String16("media.player"));

if (binder != 0) {

break;

}

usleep(500000); // 0.5s

} while (true);

if (sDeathNotifier == NULL) {

sDeathNotifier = new DeathNotifier(); //创建死亡通知对象

}

//将死亡通知连接到binder 【见流程14】

binder->linkToDeath(sDeathNotifier);

sMediaPlayerService = interface\_cast<IMediaPlayerService>(binder);

}

return sMediaPlayerService;

}

其中defaultServiceManager()过程在上一篇文章获取ServiceManager已讲过，返回BpServiceManager。

在请求获取名为”media.player”的服务过程中，采用不断循环获取的方法。由于MediaPlayerService服务可能还没向ServiceManager注册完成或者尚未启动完成等情况，故则binder返回为NULL，休眠0.5s后继续请求，直到获取服务为止。

### BpSM.getService

[-> IServiceManager.cpp ::BpServiceManager]

virtual sp<IBinder> getService(const String16& name) const

{

unsigned n;

for (n = 0; n < 5; n++){

sp<IBinder> svc = checkService(name); //【见2.3】

if (svc != NULL) return svc;

sleep(1);

}

return NULL;

}

通过BpServiceManager来获取MediaPlayer服务：检索服务是否存在，当服务存在则返回相应的服务，当服务不存在则休眠1s再继续检索服务。该循环进行5次。为什么是循环5次呢，这估计跟Android的ANR时间为5s相关。如果每次都无法获取服务，循环5次，每次循环休眠1s，忽略checkService()的时间，差不多就是5s的时间

### BpSM.checkService

G

[-> IServiceManager.cpp ::BpServiceManager]

virtual sp<IBinder> checkService( const String16& name) const

{

Parcel data, reply;

//写入RPC头

data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());

//写入服务名

data.writeString16(name);

remote()->transact(CHECK\_SERVICE\_TRANSACTION, data, &reply); //【见2.4】

return reply.readStrongBinder(); //【见小节2.9】

}

检索指定服务是否存在, 其中remote()为BpBinder

### BpBinder::transact

F

[-> BpBinder.cpp]

status\_t BpBinder::transact(

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

{

if (mAlive) {

//【见流程2.5】

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

mHandle, code, data, reply, flags);

if (status == DEAD\_OBJECT) mAlive = 0;

return status;

}

return DEAD\_OBJECT;

}

Binder代理类调用transact()方法，真正工作还是交给IPCThreadState来进行transact工作，

#### IPCThreadState::self

[-> IPCThreadState.cpp]

IPCThreadState\* IPCThreadState::self()

{

if (gHaveTLS) {

restart:

const pthread\_key\_t k = gTLS;

IPCThreadState\* st = (IPCThreadState\*)pthread\_getspecific(k);

if (st) return st;

return new IPCThreadState; //初始IPCThreadState 【见小节2.4.2】

}

if (gShutdown) return NULL;

pthread\_mutex\_lock(&gTLSMutex);

if (!gHaveTLS) { //首次进入gHaveTLS为false

if (pthread\_key\_create(&gTLS, threadDestructor) != 0) { //创建线程的TLS

pthread\_mutex\_unlock(&gTLSMutex);

return NULL;

}

gHaveTLS = true;

}

pthread\_mutex\_unlock(&gTLSMutex);

goto restart;

}

TLS是指Thread local storage(线程本地储存空间)，每个线程都拥有自己的TLS，并且是私有空间，线程之间不会共享。通过pthread\_getspecific/pthread\_setspecific函数可以获取/设置这些空间中的内容。从线程本地存储空间中获得保存在其中的IPCThreadState对象。

#### IPCThreadState初始化

Df

[-> IPCThreadState.cpp]

IPCThreadState::IPCThreadState()

: mProcess(ProcessState::self()),

mMyThreadId(gettid()),

mStrictModePolicy(0),

mLastTransactionBinderFlags(0)

{

pthread\_setspecific(gTLS, this);

clearCaller();

mIn.setDataCapacity(256);

mOut.setDataCapacity(256);

}

每个线程都有一个IPCThreadState，每个IPCThreadState中都有一个mIn、一个mOut。成员变量mProcess保存了ProcessState变量(每个进程只有一个)。

mIn 用来接收来自Binder设备的数据，默认大小为256字节；

mOut用来存储发往Binder设备的数据，默认大小为256字节。

### IPC::transact

[-> IPCThreadState.cpp]

[-> IPCThreadState.cpp]

status\_t IPCThreadState::transact(int32\_t handle,

uint32\_t code, const Parcel& data,

Parcel\* reply, uint32\_t flags)

{

status\_t err = data.errorCheck(); //数据错误检查

flags |= TF\_ACCEPT\_FDS;

....

if (err == NO\_ERROR) {

// 传输数据 【见流程2.6】

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) { //flags=0进入该分支

if (reply) {

//等待响应 【见流程2.7】

err = waitForResponse(reply);

} else {

Parcel fakeReply;

err = waitForResponse(&fakeReply);

}

} else {

//不需要响应消息的binder则进入该分支

err = waitForResponse(NULL, NULL);

}

return err;

}

**2.6 IPC.writeTransactionData**

[-> IPCThreadState.cpp]

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.ptr = 0;

tr.target.handle = handle; // handle = 0

tr.code = code; // code = CHECK\_SERVICE\_TRANSACTION

tr.flags = binderFlags; // binderFlags = 0

tr.cookie = 0;

tr.sender\_pid = 0;

tr.sender\_euid = 0;

// data为记录Media服务信息的Parcel对象

const status\_t err = data.errorCheck();

if (err == NO\_ERROR) {

tr.data\_size = data.ipcDataSize(); // mDataSize

tr.data.ptr.buffer = data.ipcData(); //mData

tr.offsets\_size = data.ipcObjectsCount()\*sizeof(binder\_size\_t); //mObjectsSize

tr.data.ptr.offsets = data.ipcObjects(); //mObjects

} else if (statusBuffer) {

...

} else {

return (mLastError = err);

}

mOut.writeInt32(cmd); //cmd = BC\_TRANSACTION

mOut.write(&tr, sizeof(tr)); //写入binder\_transaction\_data数据

return NO\_ERROR;

}

其中handle的值用来标识目的端，注册服务过程的目的端为service manager，此处handle=0所对应的是binder\_context\_mgr\_node对象，正是service manager所对应的binder实体对象。[binder\_transaction\_data结构体](http://gityuan.com/2015/11/01/binder-driver/#bindertransactiondata)是binder驱动通信的数据结构，该过程最终是把Binder请求码BC\_TRANSACTION和binder\_transaction\_data结构体写入到mOut。

**2.7 IPC.waitForResponse**

[-> IPCThreadState.cpp]

status\_t IPCThreadState::waitForResponse(Parcel \*reply, status\_t \*acquireResult)

{

int32\_t cmd;

int32\_t err;

while (1) {

if ((err=talkWithDriver()) < NO\_ERROR) break; // 【见流程2.8】

err = mIn.errorCheck();

if (err < NO\_ERROR) break;

if (mIn.dataAvail() == 0) continue;

cmd = mIn.readInt32();

switch (cmd) {

case BR\_TRANSACTION\_COMPLETE: ...

case BR\_DEAD\_REPLY: ...

case BR\_FAILED\_REPLY: ...

case BR\_ACQUIRE\_RESULT: ...

case BR\_REPLY:

{

binder\_transaction\_data tr;

err = mIn.read(&tr, sizeof(tr));

if (reply) {

if ((tr.flags & TF\_STATUS\_CODE) == 0) {

reply->ipcSetDataReference(

reinterpret\_cast<const uint8\_t\*>(tr.data.ptr.buffer),

tr.data\_size,

reinterpret\_cast<const binder\_size\_t\*>(tr.data.ptr.offsets),

tr.offsets\_size/sizeof(binder\_size\_t),

freeBuffer, this);

} else {

...

}

}

}

goto finish;

default:

err = executeCommand(cmd);

if (err != NO\_ERROR) goto finish;

break;

}

}

...

return err;

}

**2.8 IPC.talkWithDriver**

[-> IPCThreadState.cpp]

status\_t IPCThreadState::talkWithDriver(bool doReceive)

{

...

binder\_write\_read bwr;

const bool needRead = mIn.dataPosition() >= mIn.dataSize();

const size\_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;

bwr.write\_size = outAvail;

bwr.write\_buffer = (uintptr\_t)mOut.data();

if (doReceive && needRead) {

//接收数据缓冲区信息的填充。如果以后收到数据，就直接填在mIn中了。

bwr.read\_size = mIn.dataCapacity();

bwr.read\_buffer = (uintptr\_t)mIn.data();

} else {

bwr.read\_size = 0;

bwr.read\_buffer = 0;

}

//当读缓冲和写缓冲都为空，则直接返回

if ((bwr.write\_size == 0) && (bwr.read\_size == 0)) return NO\_ERROR;

bwr.write\_consumed = 0;

bwr.read\_consumed = 0;

status\_t err;

do {

//通过ioctl不停的读写操作，跟Binder Driver进行通信【2.8.1】

if (ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr) >= 0)

err = NO\_ERROR;

...

} while (err == -EINTR); //当被中断，则继续执行

...

return err;

}

[binder\_write\_read结构体](http://gityuan.com/2015/11/01/binder-driver/#binderwriteread)用来与Binder设备交换数据的结构, 通过ioctl与mDriverFD通信，是真正与Binder驱动进行数据读写交互的过程。 先向service manager进程发送查询服务的请求(BR\_TRANSACTION)，见[Binder系列3—启动ServiceManager](http://gityuan.com/2015/11/07/binder-start-sm/)。当service manager进程收到该命令后，会执行do\_find\_service() 查询服务所对应的handle，然后再binder\_send\_reply()应答 发起者，发送BC\_REPLY协议，然后调用binder\_transaction()，再向服务请求者的Todo队列 插入事务。

接下来，再看看binder\_transaction过程。

**2.8.1 binder\_transaction**

static void binder\_transaction(struct binder\_proc \*proc,

struct binder\_thread \*thread,

struct binder\_transaction\_data \*tr, int reply){

//根据各种判定，获取以下信息：

struct binder\_thread \*target\_thread； //目标线程

struct binder\_proc \*target\_proc； //目标进程

struct binder\_node \*target\_node； //目标binder节点

struct list\_head \*target\_list； //目标TODO队列

wait\_queue\_head\_t \*target\_wait； //目标等待队列

...

//分配两个结构体内存

struct binder\_transaction \*t = kzalloc(sizeof(\*t), GFP\_KERNEL);

struct binder\_work \*tcomplete = kzalloc(sizeof(\*tcomplete), GFP\_KERNEL);

//从target\_proc分配一块buffer

t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,

for (; offp < off\_end; offp++) {

switch (fp->type) {

case BINDER\_TYPE\_BINDER: ...

case BINDER\_TYPE\_WEAK\_BINDER: ...

case BINDER\_TYPE\_HANDLE:

case BINDER\_TYPE\_WEAK\_HANDLE: {

struct binder\_ref \*ref = binder\_get\_ref(proc, fp->handle,

fp->type == BINDER\_TYPE\_HANDLE);

...

//此时运行在servicemanager进程，故ref->node是指向服务所在进程的binder实体，

//而target\_proc为请求服务所在的进程，此时并不相等。

if (ref->node->proc == target\_proc) {

if (fp->type == BINDER\_TYPE\_HANDLE)

fp->type = BINDER\_TYPE\_BINDER;

else

fp->type = BINDER\_TYPE\_WEAK\_BINDER;

fp->binder = ref->node->ptr;

fp->cookie = ref->node->cookie; //BBinder服务的地址

binder\_inc\_node(ref->node, fp->type == BINDER\_TYPE\_BINDER, 0, NULL);

} else {

struct binder\_ref \*new\_ref;

//请求服务所在进程并非服务所在进程，则为请求服务所在进程创建binder\_ref

new\_ref = binder\_get\_ref\_for\_node(target\_proc, ref->node);

fp->binder = 0;

fp->handle = new\_ref->desc; //重新赋予handle值

fp->cookie = 0;

binder\_inc\_ref(new\_ref, fp->type == BINDER\_TYPE\_HANDLE, NULL);

}

} break;

case BINDER\_TYPE\_FD: ...

}

}

//分别target\_list和当前线程TODO队列插入事务

t->work.type = BINDER\_WORK\_TRANSACTION;

list\_add\_tail(&t->work.entry, target\_list);

tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;

list\_add\_tail(&tcomplete->entry, &thread->todo);

if (target\_wait)

wake\_up\_interruptible(target\_wait);

return;

}

这个过程非常重要，分两种情况来说：

1. 当请求服务的进程与服务属于不同进程，则为请求服务所在进程创建binder\_ref对象，指向服务进程中的binder\_node;
2. 当请求服务的进程与服务属于同一进程，则不再创建新对象，只是引用计数加1，并且修改type为BINDER\_TYPE\_BINDER或BINDER\_TYPE\_WEAK\_BINDER。

**2.8.2 binder\_thread\_read**

binder\_thread\_read（...）{

...

//当线程todo队列有数据则执行往下执行；当线程todo队列没有数据，则进入休眠等待状态

ret = wait\_event\_freezable(thread->wait, binder\_has\_thread\_work(thread));

...

while (1) {

uint32\_t cmd;

struct binder\_transaction\_data tr;

struct binder\_work \*w;

struct binder\_transaction \*t = NULL;

//先从线程todo队列获取事务数据

if (!list\_empty(&thread->todo)) {

w = list\_first\_entry(&thread->todo, struct binder\_work, entry);

// 线程todo队列没有数据, 则从进程todo对获取事务数据

} else if (!list\_empty(&proc->todo) && wait\_for\_proc\_work) {

...

}

switch (w->type) {

case BINDER\_WORK\_TRANSACTION:

//获取transaction数据

t = container\_of(w, struct binder\_transaction, work);

break;

case : ...

}

//只有BINDER\_WORK\_TRANSACTION命令才能继续往下执行

if (!t) continue;

if (t->buffer->target\_node) {

...

} else {

tr.target.ptr = NULL;

tr.cookie = NULL;

cmd = BR\_REPLY; //设置命令为BR\_REPLY

}

tr.code = t->code;

tr.flags = t->flags;

tr.sender\_euid = t->sender\_euid;

if (t->from) {

struct task\_struct \*sender = t->from->proc->tsk;

//当非oneway的情况下,将调用者进程的pid保存到sender\_pid

tr.sender\_pid = task\_tgid\_nr\_ns(sender, current->nsproxy->pid\_ns);

} else {

...

}

tr.data\_size = t->buffer->data\_size;

tr.offsets\_size = t->buffer->offsets\_size;

tr.data.ptr.buffer = (void \*)t->buffer->data +

proc->user\_buffer\_offset;

tr.data.ptr.offsets = tr.data.ptr.buffer +

ALIGN(t->buffer->data\_size,

sizeof(void \*));

//将cmd和数据写回用户空间

put\_user(cmd, (uint32\_t \_\_user \*)ptr);

ptr += sizeof(uint32\_t);

copy\_to\_user(ptr, &tr, sizeof(tr));

ptr += sizeof(tr);

list\_del(&t->work.entry);

t->buffer->allow\_user\_free = 1;

if (cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)) {

...

} else {

t->buffer->transaction = NULL;

kfree(t); //通信完成则运行释放

}

break;

}

done:

\*consumed = ptr - buffer;

if (proc->requested\_threads + proc->ready\_threads == 0 &&

proc->requested\_threads\_started < proc->max\_threads &&

(thread->looper & (BINDER\_LOOPER\_STATE\_REGISTERED |

BINDER\_LOOPER\_STATE\_ENTERED))) {

proc->requested\_threads++;

// 生成BR\_SPAWN\_LOOPER命令，用于创建新的线程

put\_user(BR\_SPAWN\_LOOPER, (uint32\_t \_\_user \*)buffer)；

}

return 0;

}

**2.9 readStrongBinder**

[-> Parcel.cpp]

sp<IBinder> Parcel::readStrongBinder() const

{

sp<IBinder> val;

//【见小节2.9.1】

unflatten\_binder(ProcessState::self(), \*this, &val);

return val;

}

**2.9.1 unflatten\_binder**

[-> Parcel.cpp]

***status\_t unflatten\_binder(const sp<ProcessState>& proc, const Parcel& in, sp<IBinder>\* out) {***

***const flat\_binder\_object\* flat = in.readObject(false);***

***if (flat) {***

***switch (flat->type) {***

***case BINDER\_TYPE\_BINDER:***

***// 当请求服务的进程与服务属于同一进程***

***\*out = reinterpret\_cast<IBinder\*>(flat->cookie);***

***return finish\_unflatten\_binder(NULL, \*flat, in);***

***case BINDER\_TYPE\_HANDLE:***

***//请求服务的进程与服务属于不同进程【见2.9.2】***

***\*out = proc->getStrongProxyForHandle(flat->handle);***

***//创建BpBinder对象***

***return finish\_unflatten\_binder(***

***static\_cast<BpBinder\*>(out->get()), \*flat, in);***

***}***

***}***

***return BAD\_TYPE;***

***}***

**2.9.2 getStrongProxyForHandle**

[-> ProcessState.cpp]

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

{

sp<IBinder> result;

AutoMutex \_l(mLock);

//查找handle对应的资源项[2.9.3]

handle\_entry\* e = lookupHandleLocked(handle);

if (e != NULL) {

IBinder\* b = e->binder;

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

...

//当handle值所对应的IBinder不存在或弱引用无效时，则创建BpBinder对象

b = new BpBinder(handle);

e->binder = b;

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

result = b;

} else {

result.force\_set(b);

e->refs->decWeak(this);

}

}

return result;

}

readStrongBinder的功能是flat\_binder\_object解析并创建BpBinder对象.

**2.9.3 lookupHandleLocked**

ProcessState::handle\_entry\* ProcessState::lookupHandleLocked(int32\_t handle)

{

const size\_t N=mHandleToObject.size();

//当handle大于mHandleToObject的长度时，进入该分支

if (N <= (size\_t)handle) {

handle\_entry e;

e.binder = NULL;

e.refs = NULL;

//从mHandleToObject的第N个位置开始，插入(handle+1-N)个e到队列中

status\_t err = mHandleToObject.insertAt(e, N, handle+1-N);

if (err < NO\_ERROR) return NULL;

}

return &mHandleToObject.editItemAt(handle);

}

根据handle值来查找对应的handle\_entry.

## 总结

Fd

请求服务(getService)过程，就是向servicemanager进程查询指定服务，当执行binder\_transaction()时，会区分请求服务所属进程情况。

当请求服务的进程与服务属于不同进程，则为请求服务所在进程创建binder\_ref对象，指向服务进程中的binder\_node;

最终readStrongBinder()，返回的是BpBinder对象；

当请求服务的进程与服务属于同一进程，则不再创建新对象，只是引用计数加1，并且修改type为BINDER\_TYPE\_BINDER或BINDER\_TYPE\_WEAK\_BINDER。

最终readStrongBinder()，返回的是BBinder对象的真实子类；

<http://gityuan.com/2015/11/15/binder-get-service/>

# 使用Binder

自定义binder架构的 client/ server组件

Bn意味着Binder Native 端 Bp是Binder Proxy端， 这两端会实现相同的接口，但Proxy端只是通过binder ipc发送一个binder transaction， native端是真正做事情，再将结果返回。 Android用此机制实现高效的远程调用。

## 设计模式

## Native层(C)Binder

源码结构：

* ClientDemo.cpp: 客户端程序
* ServerDemo.cpp：服务端程序
* IMyService.h：自定义的MyService服务的头文件
* IMyService.cpp：自定义的MyService服务
* Android.mk：源码build文件

### 服务端

#include "IMyService.h"

int main() {

//获取service manager引用

sp < IServiceManager > sm = defaultServiceManager();

//注册名为"service.myservice"的服务到service manager

sm->addService(String16("service.myservice"), new BnMyService());

ProcessState::self()->startThreadPool(); //启动线程池

IPCThreadState::self()->joinThreadPool(); //把主线程加入线程池

return 0;

}

将名为”service.myservice”的BnMyService服务添加到ServiceManager，并启动服务

### 客户端

Ds

#include "IMyService.h"

int main() {

//获取service manager引用

sp < IServiceManager > sm = defaultServiceManager();

//获取名为"service.myservice"的binder接口

sp < IBinder > binder = sm->getService(String16("service.myservice"));

//将biner对象转换为强引用类型的IMyService

sp<IMyService> cs = interface\_cast < IMyService > (binder);

//利用binder引用调用远程sayHello()方法

cs->sayHello();

return 0;

}

获取名为”service.myservice”的服务，再进行类型，最后调用远程方法sayHello()

### 创建MyService

#### IMyService.h

namespace android

{

class IMyService : public IInterface

{

public:

DECLARE\_META\_INTERFACE(MyService); //使用宏，申明MyService

virtual void sayHello()=0; //定义方法

};

//定义命令字段

enum

{

HELLO = 1,

};

//申明客户端BpMyService

class BpMyService: public BpInterface<IMyService> {

public:

BpMyService(const sp<IBinder>& impl);

virtual void sayHello();

};

//申明服务端BnMyService

class BnMyService: public BnInterface<IMyService> {

public:

virtual status\_t onTransact(uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags = 0);

virtual void sayHello();

};

}

主要功能：

申明IMyService

申明BpMyService（Binder客户端）

申明BnMyService（Binder的服务端）

#### IMyService.cpp

电饭锅和

原sad

大师傅大师傅

#include "IMyService.h"

namespace android

{

//使用宏，完成MyService定义

IMPLEMENT\_META\_INTERFACE(MyService, "android.demo.IMyService");

//客户端

BpMyService::BpMyService(const sp<IBinder>& impl) :

BpInterface<IMyService>(impl) {

}

// 实现客户端sayHello方法

void BpMyService::sayHello() {

printf("BpMyService::sayHello\n");

Parcel data, reply;

data.writeInterfaceToken(IMyService::getInterfaceDescriptor());

remote()->transact(HELLO, data, &reply);

printf("get num from BnMyService: %d\n", reply.readInt32());

}

//服务端，接收远程消息，处理onTransact方法

status\_t BnMyService::onTransact(uint\_t code, const Parcel& data,

Parcel\* reply, uint32\_t flags) {

switch (code) {

case HELLO: { //收到HELLO命令的处理流程

printf("BnMyService:: got the client hello\n");

CHECK\_INTERFACE(IMyService, data, reply);

sayHello();

reply->writeInt32(2015);

return NO\_ERROR;

}

break;

default:

break;

}

return NO\_ERROR;

}

// 实现服务端sayHello方法

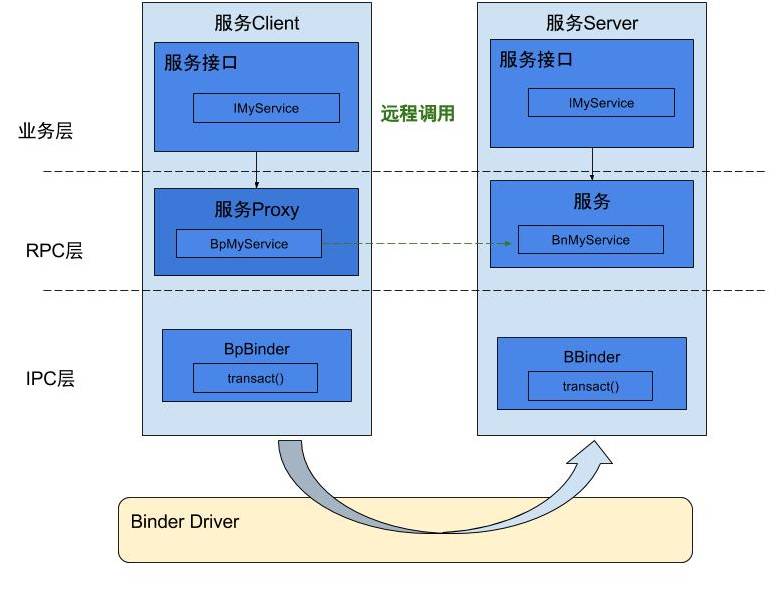
void BnMyService::sayHello() {

printf("BnMyService::sayHello\n");

};

1. }

### 原理图



### 运行

#### Android.mk

# ClientDemo

# =========================================================

LOCAL\_PATH := $(call my-dir)

include $(CLEAR\_VARS)

LOCAL\_SHARED\_LIBRARIES := \

libcutils \

libutils \

libbinder

LOCAL\_MODULE := ClientDemo

LOCAL\_SRC\_FILES := \

IMyService.cpp \

ClientDemo.cpp

LOCAL\_MODULE\_TAGS := optional

include $(BUILD\_EXECUTABLE)

# ServerDemo

# =========================================================

include $(CLEAR\_VARS)

LOCAL\_SHARED\_LIBRARIES := \

libcutils \

libutils \

libbinder

LOCAL\_MODULE := ServerDemo

LOCAL\_SRC\_FILES := \

IMyService.cpp \

ServerDemo.cpp

LOCAL\_MODULE\_TAGS := optional

include $(BUILD\_EXECUTABLE)

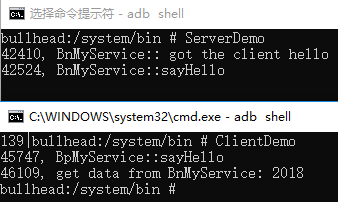
**(1)编译生成** 利用Android.mk编译上述代码，在Android的源码中，通过mm编译后，可生成两个可执行文件ServerDemo，ClientDemo。

#### **(2)执行**

首先将这两个ServerDemo，ClientDemo可执行文件push到手机

adb root  
adb remount  
adb push Z:\android-8.0.0\_r1\out\target\product\bullhead\system\bin\ServerDemo /system/bin/  
adb push Z:\android-8.0.0\_r1\out\target\product\bullhead\system\bin\ClientDemo /system/bin/

#### **(3)结果**



## Framework层（Java层）Binder

源码结构：

Server端

1. ServerDemo.java：可执行程序
2. IMyService.java: 定义IMyService接口
3. MyService.java：定义MyService

Client端

1. ClientDemo.java：可执行程序
2. IMyService.java: 与Server端完全一致
3. MyServiceProxy.java：定义MyServiceProxy

### Server端

X

**(1)ServerDemo.java**

可执行程序

***public class ServerDemo {***

***public static void main(String[] args) {***

***System.out.println("MyService Start");***

***//准备Looper循环执行***

***Looper.prepareMainLooper();***

***//设置为前台优先级***

***android.os.Process.setThreadPriority(android.os.Process.THREAD\_PRIORITY\_FOREGROUND);***

***//注册服务***

***ServiceManager.addService("MyService", new MyService());***

***Looper.loop();***

***}***

***}***

**(2)IMyService.java**

定义sayHello()方法，DESCRIPTOR属性

public interface IMyService extends IInterface {

static final java.lang.String DESCRIPTOR = "com.gityuan.frameworkBinder.MyServer";

public void sayHello(String str) throws RemoteException ;

static final int TRANSACTION\_say = android.os.IBinder.FIRST\_CALL\_TRANSACTION;

}

**(3)MyService.java**

public class MyService extends Binder implements IMyService{

public MyService() {

this.attachInterface(this, DESCRIPTOR);

}

@Override

public IBinder asBinder() {

return this;

}

/\*\* 将MyService转换为IMyService接口 \*\*/

public static com.gityuan.frameworkBinder.IMyService asInterface( android.os.IBinder obj) {

if ((obj == null)) {

return null;

}

android.os.IInterface iInterface = obj.queryLocalInterface(DESCRIPTOR);

if (((iInterface != null)&&(iInterface instanceof com.gityuan.frameworkBinder.IMyService))){

return ((com.gityuan.frameworkBinder.IMyService) iInterface);

}

return null;

}

/\*\* 服务端，接收远程消息，处理onTransact方法 \*\*/

@Override

protected boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {

switch (code) {

case INTERFACE\_TRANSACTION: {

reply.writeString(DESCRIPTOR);

return true;

}

case TRANSACTION\_say: {

data.enforceInterface(DESCRIPTOR);

String str = data.readString();

sayHello(str);

reply.writeNoException();

return true;

}}

return super.onTransact(code, data, reply, flags);

}

/\*\* 自定义sayHello()方法 \*\*/

@Override

public void sayHello(String str) {

System.out.println("MyService:: Hello, " + str);

}

}

### Client端

**(1)ClientDemo.java**

可执行程序

***public class ClientDemo {***

***public static void main(String[] args) throws RemoteException {***

***System.out.println("Client start");***

***IBinder binder = ServiceManager.getService("MyService"); //获取名为"MyService"的服务***

***IMyService myService = new MyServiceProxy(binder); //创建MyServiceProxy对象***

***myService.sayHello("binder"); //通过MyServiceProxy对象调用接口的方法***

***System.out.println("Client end");***

***}***

***}***

**(2)IMyService.java**

与Server端的IMyService是一致，基本都是拷贝一份过来。

**(3)MyServiceProxy.java**

public class MyServiceProxy implements IMyService {

private android.os.IBinder mRemote; //代表BpBinder

public MyServiceProxy(android.os.IBinder remote) {

mRemote = remote;

}

public java.lang.String getInterfaceDescriptor() {

return DESCRIPTOR;

}

/\*\* 自定义的sayHello()方法 \*\*/

@Override

public void sayHello(String str) throws RemoteException {

android.os.Parcel \_data = android.os.Parcel.obtain();

android.os.Parcel \_reply = android.os.Parcel.obtain();

try {

\_data.writeInterfaceToken(DESCRIPTOR);

\_data.writeString(str);

mRemote.transact(TRANSACTION\_say, \_data, \_reply, 0);

\_reply.readException();

} finally {

\_reply.recycle();

\_data.recycle();

}

}

@Override

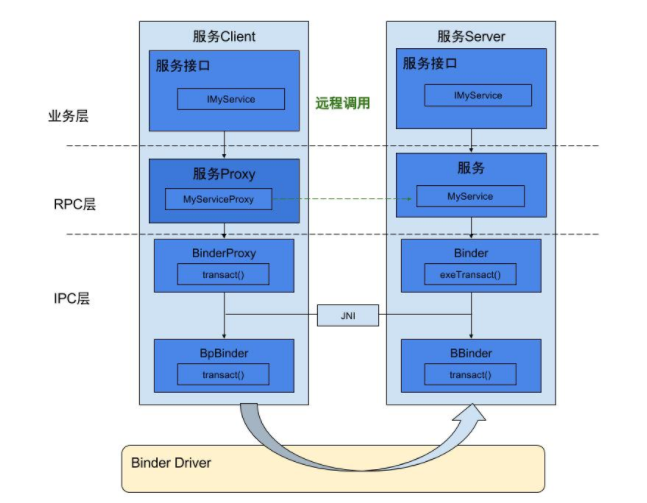
public IBinder asBinder() {

return mRemote;

}

}

### 原理图



### 运行

首先将ServerDemo，ClientDemo可执行文件，以及ServerDemo.jar，ClientDemo.jar都push到手机

adb push ServerDemo.jar /system/framework

adb push ClientDemo.jar /system/framework

## App层

### C端

mBinder = ServiceManager.getService(“mybinder”);

Parcel data = Parcel.obtain();

data.writeString(rcPath);

Parcel reply = Parcel.obtain();  
**try** {  
 mBinder.transact(keyCode, data, reply, 0);  
} **catch** (RemoteException e) {  
 e.printStackTrace();  
}

### S端

**mBinder** = **new** UpBinder();  
**try** {*//普通app，会报这个异常！！* ServiceManager.*addService*(***“***mybinder***”***, **mBinder**);  
}**catch** (Exception e){  
   
}

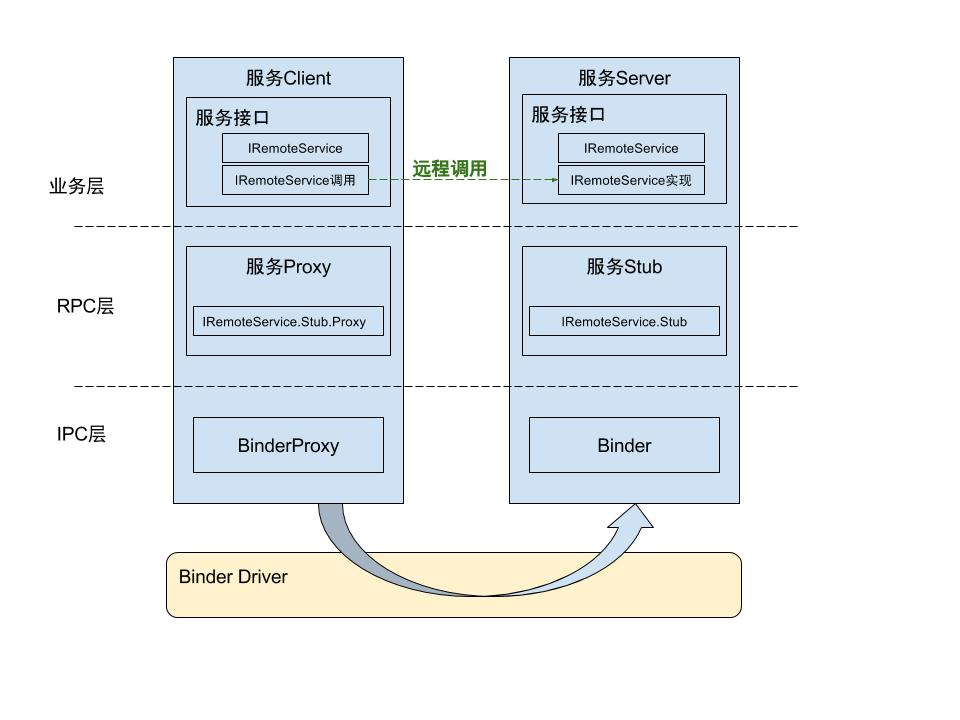
**public class** UpBinder **extends** Binder {  
 **private static final** String ***TAG*** = **"UpBinder"**;  
 @Override  
 **protected boolean** onTransact(**int** code, Parcel data, Parcel reply,  
 **int** flags) **throws** RemoteException {**if**( code == TransactionCodeProtocol.***OsUpIndexUi***){  
 handleUpgradeIndexUi();  
 }   
 **return true**;  
 }

}

## AIDL（system\_server）写法

采用AIDL技术，是原理还是利用framework binder的架构。本文的实例AIDL会自动生成一个与之相对应的IRemoteService.java文件，

所以本质还是没有改变。如下：



类图模板

<http://gityuan.com/2015/11/23/binder-aidl/>

### java

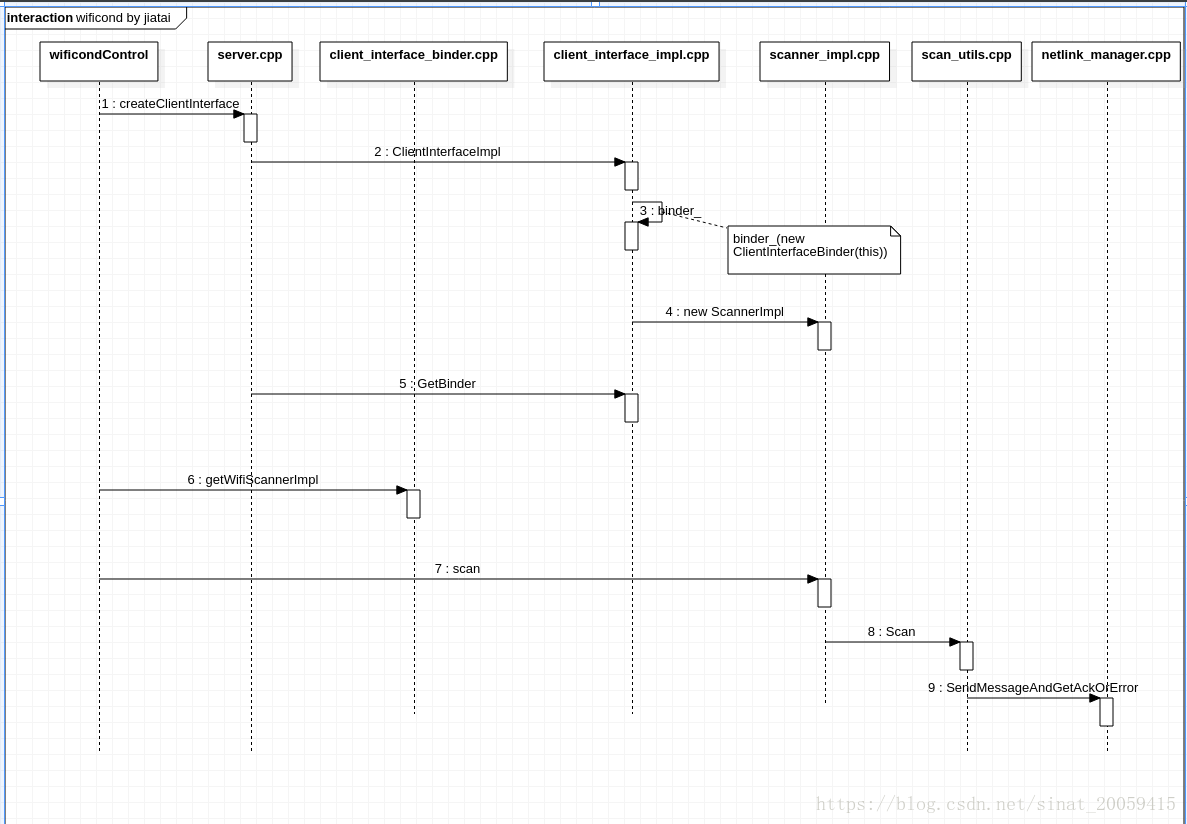
### aidl-cpp

参考https://www.jianshu.com/p/5f2ae92291af 或者 https://android.googlesource.com/platform/system/tools/aidl/+/brillo-m10-dev/docs/aidl-cpp.md

是有aidl-cpp这回事的，可以完成java和C++之间的IPC通信。

前面三个类比于java中的aidl和自动生成的实现类，c++多了.h和.cpp，对应于.java。

---------------------



请附上博文链接！

## 参考

Binder系列8—如何使用Binder

<http://gityuan.com/2015/11/22/binder-use/>

# Binder IPC的权限控制

基于Android 6.0的源码剖析， 分析Binder IPC通信的权限控制方法clearCallingIdentity和restoreCallingIdentity的原理和用途。

/frameworks/base/core/java/android/os/Binder.java

/frameworks/base/core/jni/android\_util\_Binder.cpp

/frameworks/native/libs/binder/IPCThreadState.cpp

## 概述

在[Binder系列](http://gityuan.com/2015/10/31/binder-prepare/)中通过十篇文章，深入探讨了Android M的Binder IPC机制。看过Android系统源代码的朋友，一定看到过Binder.clearCallingIdentity()和Binder.restoreCallingIdentity()这两个方法，其定义在Binder.java文件：

//作用是清空远程调用端的uid和pid，用当前本地进程的uid和pid替代；

public static final native long clearCallingIdentity();

//作用是恢复远程调用端的uid和pid信息，正好是`clearCallingIdentity`的反过程;

public static final native void restoreCallingIdentity(long token);

这两个方法涉及的uid和pid，每个线程都有自己独一无二的IPCThreadState对象，记录当前线程的pid和uid，可通过方法Binder.getCallingPid()和Binder.getCallingUid()获取相应的pid和uid。

clearCallingIdentity(), restoreCallingIdentity()这两个方法使用过程都是成对使用的，这两个方法配合使用，用于权限控制检测功能。

从定义这两个方法是native方法，通过[Binder的JNI调用](http://gityuan.com/2015/11/21/binder-framework/#registerandroidosbinder)，在android\_util\_Binder.cpp文件中定义了native方法所对应的jni方法。

### 2.1 clearCallingIdentity

[-> android\_util\_Binder.cpp]

static jlong android\_os\_Binder\_clearCallingIdentity(JNIEnv\* env, jobject clazz)

{

//调用IPCThreadState类的方法执行

return IPCThreadState::self()->clearCallingIdentity();

}

#### 2.1.1 IPC.clearCallingIdentity

[-> IPCThreadState.cpp]

int64\_t IPCThreadState::clearCallingIdentity()

{

int64\_t token = ((int64\_t)mCallingUid<<32) | mCallingPid;

clearCaller();

return token;

}

void IPCThreadState::clearCaller()

{

mCallingPid = getpid(); //当前进程pid赋值给mCallingPid

mCallingUid = getuid(); //当前进程uid赋值给mCallingUid

}

* mCallingUid(记为UID)，保存Binder IPC通信的调用方进程的Uid；
* mCallingPid(记为PID)，保存Binder IPC通信的调用方进程的Pid；

UID和PID是IPCThreadState的成员变量， 都是32位的int型数据，通过移位操作，将UID和PID的信息保存到token，其中高32位保存UID，低32位保存PID。然后调用clearCaller()方法将当前本地进程pid和uid分别赋值给PID和UID，最后返回token。

### restoreCallingIdentity

[-> android\_util\_Binder.cpp]

static void android\_os\_Binder\_restoreCallingIdentity(JNIEnv\* env, jobject clazz, jlong token)

{

//token记录着uid信息，将其右移32位得到的是uid

int uid = (int)(token>>32);

if (uid > 0 && uid < 999) {

//目前Android中不存在小于999的uid，当uid<999则抛出异常。

char buf[128];

jniThrowException(env, "java/lang/IllegalStateException", buf);

return;

}

//调用IPCThreadState类的方法执行

IPCThreadState::self()->restoreCallingIdentity(token);

}

#### 2.2.1 IPC.restoreCallingIdentity

[-> IPCThreadState.cpp]

void IPCThreadState::restoreCallingIdentity(int64\_t token)

{

mCallingUid = (int)(token>>32);

mCallingPid = (int)token;

}

从token中解析出PID和UID，并赋值给相应的变量。该方法正好是clearCallingIdentity的反过程。

### 2.3 getCallingPid

[-> android\_util\_Binder.cpp]

static jint android\_os\_Binder\_getCallingPid(JNIEnv\* env, jobject clazz)

{

return IPCThreadState::self()->getCallingPid();

}

#### 2.3.1 IPC.getCallingPid

[-> IPCThreadState.cpp]

pid\_t IPCThreadState::getCallingPid() const

{

return mCallingPid;

}

uid\_t IPCThreadState::getCallingUid() const

{

return mCallingUid;

}

### 2.4 远程调用

#### 2.4.1 binder\_thread\_read

binder\_thread\_read（）{

while (1) {

struct binder\_work \*w;

switch (w->type) {

case BINDER\_WORK\_TRANSACTION:

t = container\_of(w, struct binder\_transaction, work);

break;

case :...

}

if (!t)

continue; //只有BR\_TRANSACTION,BR\_REPLY才会往下执行

tr.code = t->code;

tr.flags = t->flags;

tr.sender\_euid = t->sender\_euid; //mCallingUid

if (t->from) {

struct task\_struct \*sender = t->from->proc->tsk;

//当非oneway的情况下,将调用者进程的pid保存到sender\_pid

tr.sender\_pid = task\_tgid\_nr\_ns(sender,current->nsproxy->pid\_ns);

} else {

//当oneway的的情况下,则该值为0

tr.sender\_pid = 0;

}

...

}

#### 2.4.2 IPC.executeCommand

status\_t IPCThreadState::executeCommand(int32\_t cmd)

{

BBinder\* obj;

RefBase::weakref\_type\* refs;

status\_t result = NO\_ERROR;

switch ((uint32\_t)cmd) {

case BR\_TRANSACTION:

{

const pid\_t origPid = mCallingPid;

const uid\_t origUid = mCallingUid;

mCallingPid = tr.sender\_pid; //设置调用者pid

mCallingUid = tr.sender\_euid;//设置调用者uid

...

reinterpret\_cast<BBinder\*>(tr.cookie)->transact(tr.code, buffer,

&reply, tr.flags);

mCallingPid = origPid; //恢复原来的pid

mCallingUid = origUid; //恢复原来的uid

}

case :...

}

}

关于mCallingPid、mCallingUid的修改过程:是在每次Binder Call的远程进程在执行binder\_thread\_read()过程， 会设置pid和uid. 然后在IPCThreadState的transact收到BR\_TRANSACION则会修改mCallingPid、mCallingUid。

这里需要注意的是，当oneway的的情况下mCallingPid=0，不过mCallingUid可以拿到正确值。

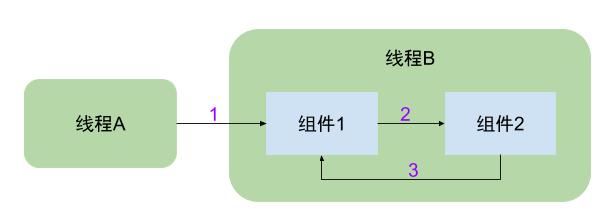
## 三、用途

### 3.1 场景分析

**场景：**首先线程A通过Binder远程调用线程B，然后线程B通过Binder调用当前线程的另一个service或者activity之类的组件。

**分析：**

1. 线程A通过Binder远程调用线程B：则线程B的IPCThreadState中的mCallingUid和mCallingPid保存的就是线程A的UID和PID。这时在线程B中调用Binder.getCallingPid()和Binder.getCallingUid()方法便可获取线程A的UID和PID，然后利用UID和PID进行权限比对，判断线程A是否有权限调用线程B的某个方法。
2. 线程B通过Binder调用当前线程的某个组件：此时线程B是线程B某个组件的调用端，则mCallingUid和mCallingPid应该保存当前线程B的PID和UID，故需要调用clearCallingIdentity()方法完成这个功能。当线程B调用完某个组件，由于线程B仍然处于线程A的被调用端，因此mCallingUid和mCallingPid需要恢复成线程A的UID和PID，这是调用restoreCallingIdentity()即可完成。



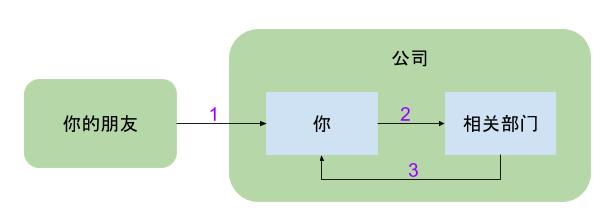
一句话：图中过程2（调用组件2开始之前）执行clearCallingIdentity()，过程3（调用组件2结束之后）执行restoreCallingIdentity()。

### 3.2 类比分析

看完场景分析，估计还有不少朋友感到迷惑，为何需要这两个方法来多此一举，直接检测最初调用端的权限不就行了吗？为了更加形象明了地说明其用途，下面用一个生活中的场景来类比说明。

**场景：**假如你的朋友请你帮忙，给她(他)到你的公司以内部价购买公司的某个产品。

**分析：**这个过程分为两个阶段



* 第一阶段：你的朋友请你帮忙的过程，这个过程并不一定所有朋友都会帮的，这时就需要一个权限检测，那么在你的朋友”远程调用”你执行任务时，你会记录他的”Identity”信息（比如是性别），有了信息那么就可以权限检测，不妨令权限规则是如果这个朋友是女性则答应帮忙，否则就认定权限不够拒绝执行（可能黑客会想到先去一趟泰国，权限控制可能相应需要打补丁了），若答应帮忙则进入第二阶段，否则直接返回。
* 第二阶段：你向自己所在公司的相关部门内购产品的过程，这个过程也并不是所有人都能权限能够内购的，只有自己公司的员工才行，否则你的朋友也不会找你帮忙了。 这个过程同样需要权限检测，但是”Identity”保存的是性别女的信息，公司内购产品如果也以性别来判断，那岂不是公司的所有男员工没有权限内购，那这公司就有点太坑了，这明显不符合实情。 clearCallingIdentity()是时候该登场了，在第二阶段开始之前，先执行clearCallingIdentity()过程，也就是把”Identity”信息清空，替换为你的信息（比如员工编码ITCode之类的），那公司相关部门通过ITCode就可以直接判断是否允许内购某产品。当第二阶段完成后，也就是你已经购买到了公司产品，这时你需要将产品交付给你的朋友，需要restoreCallingIdentity，恢复”Identity”为女的信息，这样就嗯呢该顺便交付给你的女朋友。如果不恢复信息，还是原来的ITCode，你交付的朋友可能是男的，另有其人，这样就不科学了。

相信到此，大家应该都能明白这两个方法的作用，缺一不可，而且要成对出现。

### 3.3 实例分析

上述过程主要在system\_server进程的各个线程中比较常见（普通的app应用很少出现），比如system\_server进程中的ActivityManagerService子线程，代码如下：

[–>ActivityManagerService.java]

@Override

public final void attachApplication(IApplicationThread thread) {

synchronized (this) {

//获取远程Binder调用端的pid

int callingPid = Binder.getCallingPid();

//清除远程Binder调用端uid和pid信息，并保存到origId变量

final long origId = Binder.clearCallingIdentity();

attachApplicationLocked(thread, callingPid);

//通过origId变量，还原远程Binder调用端的uid和pid信息

Binder.restoreCallingIdentity(origId);

}

}

文章[startService流程分析](http://gityuan.com/2016/02/21/start-service/#activitymanagerproxyattachapplication)中有讲到attachApplication()的调用。该方法一般是system\_server进程的子线程调用远程进程时使用，而attachApplicationLocked方法则是在同一个线程中，故需要在调用该方法前清空远程调用者的uid和pid，调用结束后恢复远程调用者的uid和pid。

# 总结

## Binder概述

1. 从IPC角度来说：Binder是Android中的一种跨进程通信方式，该通信方式在linux中没有，是Android独有；
2. 从Android Driver层：Binder还可以理解为一种虚拟的物理设备，它的设备驱动是/dev/binder；
3. 从Android Native层：Binder是创建Service Manager以及BpBinder/BBinder模型，搭建与binder驱动的桥梁；
4. 从Android Framework层：Binder是各种Manager（ActivityManager、WindowManager等）和相应xxxManagerService的桥梁；
5. 从Android APP层：Binder是客户端和服务端进行通信的媒介，当bindService的时候，服务端会返回一个包含了服务端业务调用的 Binder对象，通过这个Binder对象，客户端就可以获取服务端提供的服务或者数据，这里的服务包括普通服务和基于AIDL的服务。

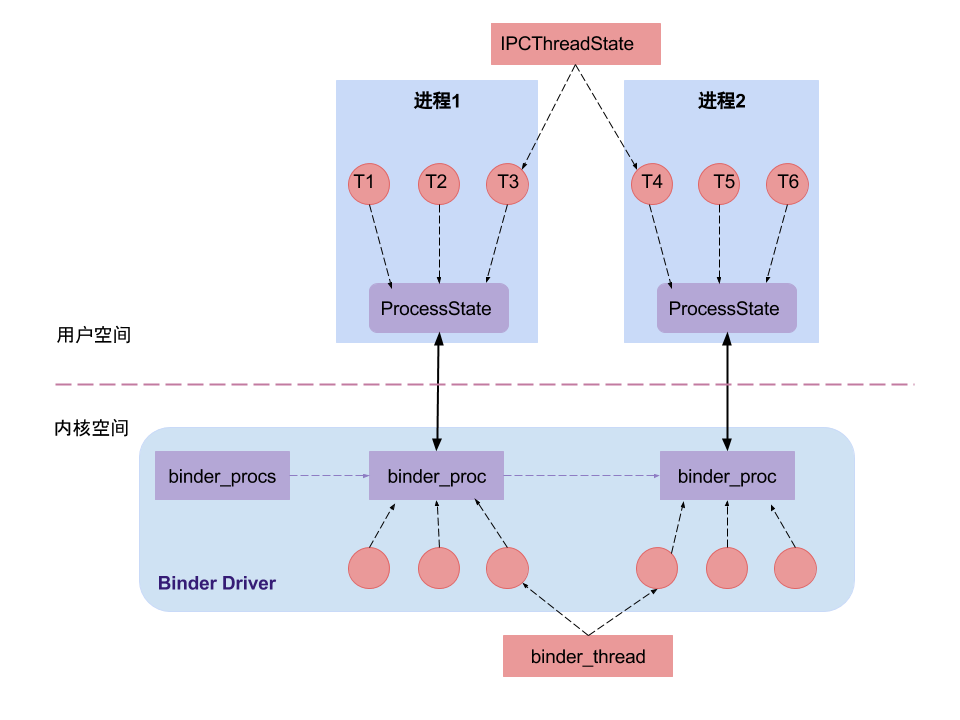
## Binder架构

Sd



Binder在整个Android系统中有这举足轻重的地位，在Native层有一套完整的binder通信的C/S架构(图中的蓝色)，Bpinder作为客户端，BBinder作为服务端。基于naive层的Binder框架，Java也有一套镜像功能的binder C/S架构，通过JNI技术，与native层的binder对应，Java层的binder功能最终都是交给native的binder来完成。从kernel到native，jni，framework层的架构所涉及的所有有关类和方法见

## Binder进程与线程



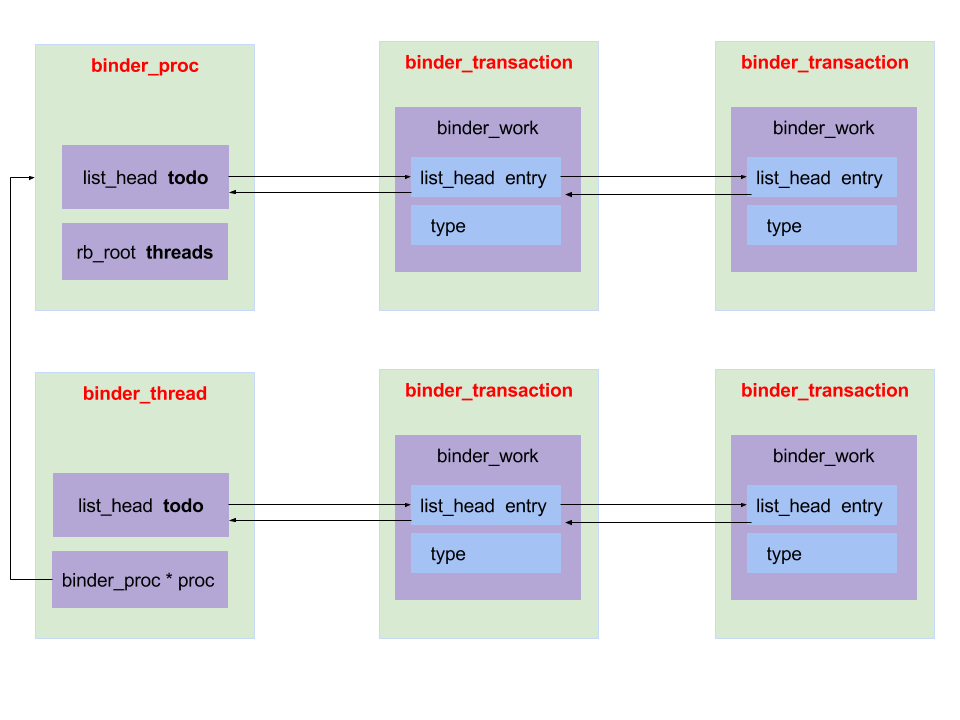
对于底层Binder驱动，通过binder\_procs链表记录所有创建的binder\_proc结构体，binder驱动层的每一个binder\_proc结构体都与用户空间的一个用于binder通信的进程一一对应，且每个进程有且只有一个ProcessState对象，这是通过单例模式来保证的。在每个进程中可以有很多个线程，每个线程对应一个IPCThreadState对象，IPCThreadState对象也是单例模式，即一个线程对应一个IPCThreadState对象，在Binder驱动层也有与之相对应的结构，那就是Binder\_thread结构体。在binder\_proc结构体中通过成员变量rb\_root threads，来记录当前进程内所有的binder\_thread。

Binder线程池：每个Server进程在启动时会创建一个binder线程池，并向其中注册一个Binder线程；之后Server进程也可以向binder线程池注册新的线程，或者Binder驱动在探测到没有空闲binder线程时会主动向Server进程注册新的的binder线程。对于一个Server进程有一个最大Binder线程数限制，默认为16个binder线程，例如Android的system\_server进程就存在16个线程。对于所有Client端进程的binder请求都是交由Server端进程的binder线程来处理的。

## Binder传输过程

Binder IPC机制，就是指在进程间传输数据（binder\_transaction\_data），一次数据的传输，称为事务（binder\_transaction）。对于多个不同进程向同一个进程发送事务时，这个同一个进程或线程的事务需要串行执行，在Binder驱动中为binder\_proc和binder\_thread都有todo队列。

也就是说对于进程间的通信，就是发送端把binder\_transaction节点，插入到目标进程或其子线程的todo队列中，等目标进程或线程不断循环地从todo队列中取出数据并进行相应的操作。

在Binder驱动层，每个接收端进程都有一个todo队列，用于保存发送端进程发送过来的binder请求，这类请求可以由接收端进程的任意一个空闲的binder线程处理；接收端进程存在一个或多个binder线程，在每个binder线程里都有一个todo队列，也是用于保存发送端进程发送过来的binder请求，这类请求只能由当前binder线程来处理。binder线程在空闲时进入可中断的休眠状态，当自己的todo队列或所属进程的todo队列有新的请求到来时便会唤醒，如果是由所需进程唤醒的，那么进程会让其中一个线程处理响应的请求，其他线程再次进入休眠状态。

## Binder路由

先来看看Native Binder IPC的两个重量级对象：BpBinder(客户端)和BBinder(服务端)都是Android中Binder通信相关的代表，它们都从IBinder类中派生而来，关系图如下：



* Binder有一个重要方法queryLocalInterface， 默认返回值为NULL；
  + BBinder/BpBinder都没有实现，默认返回NULL；BnInterface重写该方法；
  + BinderProxy(Java)默认返回NULL；Binder(Java)重写该方法；
* IInterface有一个重要方法asBinder；
* IInterface子类(服务端)会有一个方法asInterface；

Native层通过宏IMPLEMENT\_META\_INTERFACE来完成asInterface实现和descriptor的赋值过程；

对于Java层跟Native一样，也有完全对应的一套对象和方法:

* 例如ActivityManagerNative， 通过实现asInterface方法，以及其通过其构造函数 调用attachInterface()，完成descriptor的赋值过程。
* 再如AIDL全自动生成asInterface和descriptor赋值过程。

同一个进程，请求binder服务，不需要创建binder\_ref,BpBinder等这些对象，但是是否需要经过binder call，取决于descriptor是否设置。 这就涉及到Java服务Native使用，或许Native服务在Java层使用，需要格外注意。

**binder的路由原理**：BpBinder发送端，根据handler，在当前binder\_proc中，找到相应的binder\_ref，由binder\_ref再找到目标binder\_node实体，由目标binder\_node再找到目标进程binder\_proc。简单地方式是直接把binder\_transaction节点插入到binder\_proc的todo队列中，完成传输过程。

对于binder驱动来说应尽可能地把binder\_transaction节点插入到目标进程的某个线程的todo队列，效率更高。当binder驱动可以找到合适的线程，就会把binder\_transaction节点插入到相应线程的todo队列中，如果找不到合适的线程，就把节点之间插入binder\_proc的todo队列。

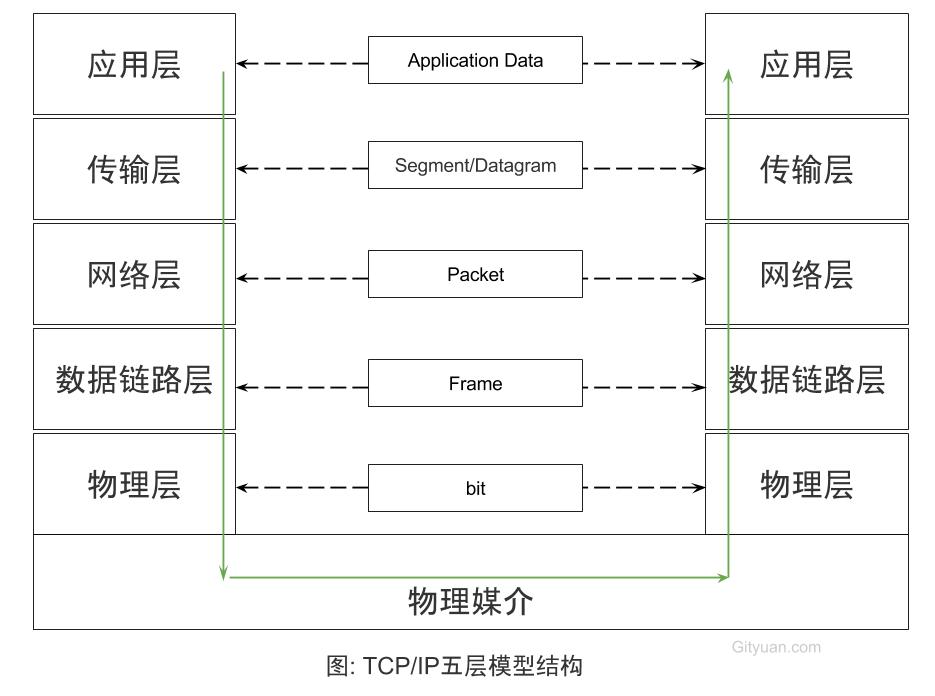
# 彻底理解Android Binder通信架构

### 引言

#### 1.1 Binder架构的思考

Android内核是基于Linux系统, 而Linux现存多种进程间IPC方式:管道, 消息队列, 共享内存, 套接字, 信号量, 信号. 为什么Android非要用Binder来进行进程间通信呢. 从我个人的理解角度, 曾尝试着在知乎回答同样一个问题 [为什么Android要采用Binder作为IPC机制？](https://www.zhihu.com/question/39440766/answer/89210950). 这是Gityuan第一次认认真真地在知乎上回答问题, 收到很多网友的点赞与回复, 让我很受鼓舞, 也决心分享更多优先地文章回报读者和粉丝, 为Android圈贡献自己的微薄之力。

在说到Binder架构之前, 先简单说说大家熟悉的TCP/IP的五层通信体系结构:

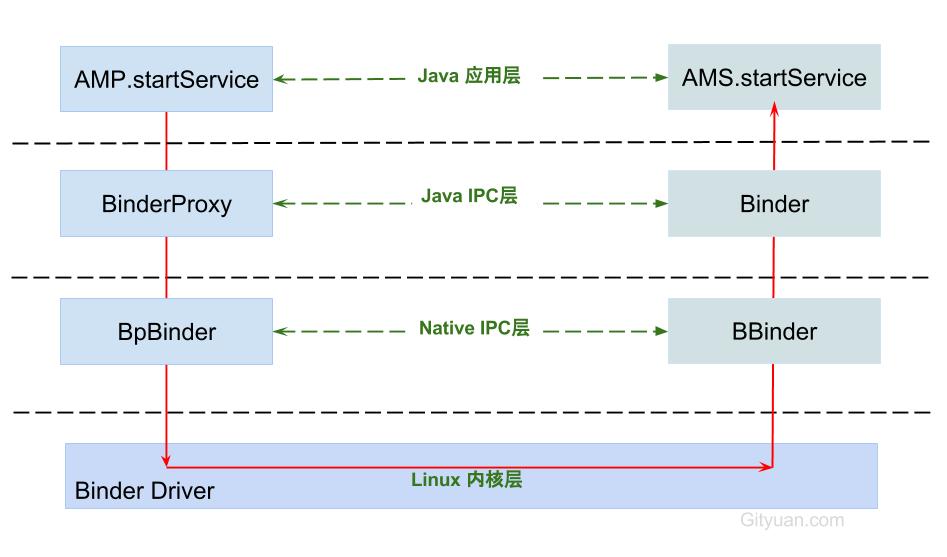


* 应用层: 直接为用户提供服务;
* 传输层: 传输的是报文(TCP数据)或者用户数据报(UDP数据)
* 网络层: 传输的是包(Packet), 例如路由器
* 数据链路层: 传输的是帧(Frame), 例如以太网交换机
* 物理层: 相邻节点间传输bit, 例如集线器,双绞线等

这是经典的五层TPC/IP协议体系, 这样分层设计的思想, 让每一个子问题都设计成一个独立的协议, 这协议的设计/分析/实现/测试都变得更加简单:

* 层与层具有独立性, 例如应用层可以使用传输层提供的功能而无需知晓其实现原理;
* 设计灵活, 层与层之间都定义好接口, 即便层内方法发生变化,只有接口不变, 对这个系统便毫无影响;
* 结构的解耦合, 让每一层可以用更适合的技术方案, 更合适的语言;
* 方便维护, 可分层调试和定位问题;

Binder架构也是采用分层架构设计, 每一层都有其不同的功能:

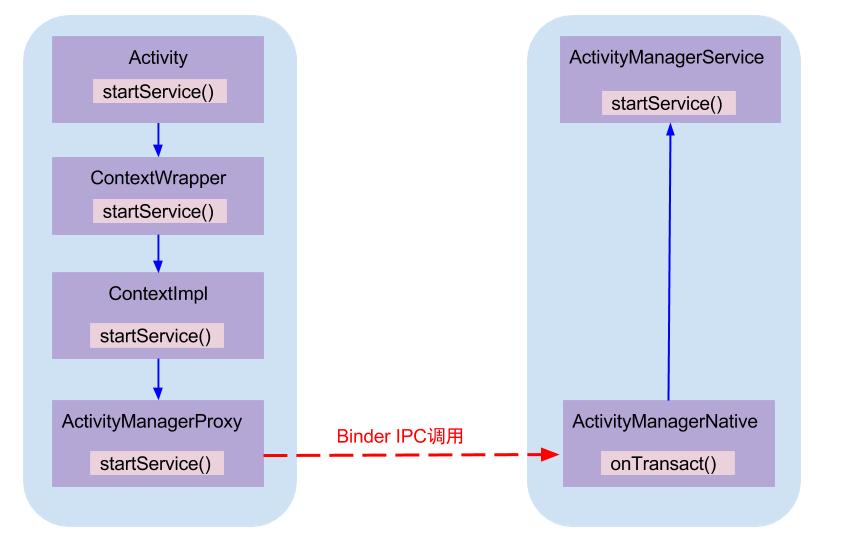


* **Java应用层:** 对于上层应用通过调用AMP.startService, 完全可以不用关心底层,经过层层调用,最终必然会调用到AMS.startService.
* **Java IPC层:** Binder通信是采用C/S架构, Android系统的基础架构便已设计好Binder在Java framework层的Binder客户类BinderProxy和服务类Binder;
* **Native IPC层:** 对于Native层,如果需要直接使用Binder(比如media相关), 则可以直接使用BpBinder和BBinder(当然这里还有JavaBBinder)即可, 对于上一层Java IPC的通信也是基于这个层面.
* **Kernel物理层:** 这里是Binder Driver, 前面3层都跑在用户空间,对于用户空间的内存资源是不共享的,每个Android的进程只能运行在自己进程所拥有的虚拟地址空间, 而内核空间却是可共享的. 真正通信的核心环节还是在Binder Driver.

### 分析起点

前面通过一个[Binder系列-开篇](http://gityuan.com/2015/10/31/binder-prepare/)来从源码讲解了Binder的各个层面, 但是Binder牵涉颇为广泛, 几乎是整个Android架构的顶梁柱, 虽说用了十几篇文章来阐述Binder的各个过程. 但依然还是没有将Binder IPC(进程间通信)的过程彻底说透.

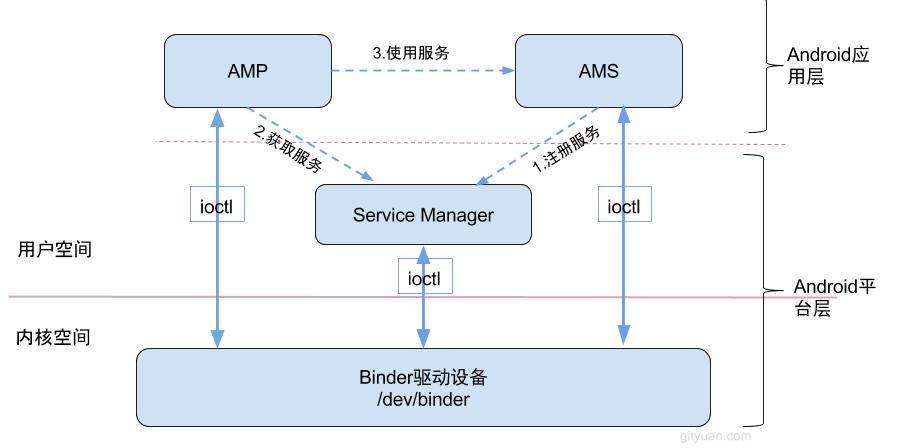
Binder系统如此庞大, 那么这里需要寻求一个出发点来穿针引线, 一窥视Binder全貌. 那么本文将从全新的视角,以[startService流程分析](http://gityuan.com/2016/03/06/start-service/)为例子来说说Binder所其作用. 首先在发起方进程调用AMP.startService，经过binder驱动，最终调用系统进程AMS.startService,如下图:



AMP和AMN都是实现了IActivityManager接口,AMS继承于AMN. 其中AMP作为Binder的客户端,运行在各个app所在进程, AMN(或AMS)运行在系统进程system\_server.

### Binder IPC原理

Binder通信采用C/S架构，从组件视角来说，包含Client、Server、ServiceManager以及binder驱动，其中ServiceManager用于管理系统中的各种服务。下面说说startService过程所涉及的Binder对象的架构图：



可以看出

这3大过程每一次都是一个完整的Binder IPC过程, 接下来从源码角度, 仅介绍**第3过程使用服务**, 即展开AMP.startService是如何调用到AMS.startService的过程.

## 二. 通信过程

### 2.1 AMP.startService

[-> ActivityManagerNative.java ::ActivityManagerProxy]

public ComponentName startService(IApplicationThread caller, Intent service, String resolvedType, String callingPackage, int userId) throws RemoteException {

//获取或创建Parcel对象【见小节2.2】

Parcel data = Parcel.obtain();

Parcel reply = Parcel.obtain();

data.writeInterfaceToken(IActivityManager.descriptor);

data.writeStrongBinder(caller != null ? caller.asBinder() : null);

service.writeToParcel(data, 0);

//写入Parcel数据 【见小节2.3】

data.writeString(resolvedType);

data.writeString(callingPackage);

data.writeInt(userId);

//通过Binder传递数据【见小节2.5】

mRemote.transact(START\_SERVICE\_TRANSACTION, data, reply, 0);

//读取应答消息的异常情况

reply.readException();

//根据reply数据来创建ComponentName对象

ComponentName res = ComponentName.readFromParcel(reply);

//【见小节2.2.3】

data.recycle();

reply.recycle();

return res;

}

主要功能:

* 获取或创建两个Parcel对象,data用于发送数据，reply用于接收应答数据.
* 将startService相关数据都封装到Parcel对象data, 其中descriptor = “android.app.IActivityManager”;
* 通过Binder传递数据,并将应答消息写入reply;
* 读取reply应答消息的异常情况和组件对象;

### 2.2 Parcel.obtain

[-> Parcel.java]

public static Parcel obtain() {

final Parcel[] pool = sOwnedPool;

synchronized (pool) {

Parcel p;

//POOL\_SIZE = 6

for (int i=0; i<POOL\_SIZE; i++) {

p = pool[i];

if (p != null) {

pool[i] = null;

return p;

}

}

}

//当缓存池没有现成的Parcel对象，则直接创建[见流程2.2.1]

return new Parcel(0);

}

sOwnedPool是一个大小为6，存放着parcel对象的缓存池,这样设计的目标是用于节省每次都创建Parcel对象的开销。obtain()方法的作用：

1. 先尝试从缓存池sOwnedPool中查询是否存在缓存Parcel对象，当存在则直接返回该对象;
2. 如果没有可用的Parcel对象，则直接创建Parcel对象。

#### 2.2.1 new Parcel

[-> Parcel.java]

private Parcel(long nativePtr) {

//初始化本地指针

init(nativePtr);

}

private void init(long nativePtr) {

if (nativePtr != 0) {

mNativePtr = nativePtr;

mOwnsNativeParcelObject = false;

} else {

// 首次创建,进入该分支[见流程2.2.2]

mNativePtr = nativeCreate();

mOwnsNativeParcelObject = true;

}

}

nativeCreate这是native方法,经过JNI进入native层, 调用android\_os\_Parcel\_create()方法.

#### 2.2.2 android\_os\_Parcel\_create

[-> android\_os\_Parcel.cpp]

static jlong android\_os\_Parcel\_create(JNIEnv\* env, jclass clazz) {

Parcel\* parcel = new Parcel();

return reinterpret\_cast<jlong>(parcel);

}

创建C++层的Parcel对象, 该对象指针强制转换为long型, 并保存到Java层的mNativePtr对象. 创建完Parcel对象利用Parcel对象写数据. 接下来以writeString为例.

#### 2.2.3 Parcel.recycle

public final void recycle() {

//释放native parcel对象

freeBuffer();

final Parcel[] pool;

//根据情况来选择加入相应池

if (mOwnsNativeParcelObject) {

pool = sOwnedPool;

} else {

mNativePtr = 0;

pool = sHolderPool;

}

synchronized (pool) {

for (int i=0; i<POOL\_SIZE; i++) {

if (pool[i] == null) {

pool[i] = this;

return;

}

}

}

}

将不再使用的Parcel对象放入缓存池，可回收重复利用，当缓存池已满则不再加入缓存池。这里有两个Parcel线程池,mOwnsNativeParcelObject变量来决定:

* mOwnsNativeParcelObject=true, 即调用不带参数obtain()方法获取的对象, 回收时会放入sOwnedPool对象池;
* mOwnsNativeParcelObject=false, 即调用带nativePtr参数的obtain(long)方法获取的对象, 回收时会放入sHolderPool对象池;

### 2.3 writeString

[-> Parcel.java]

public final void writeString(String val) {

//[见流程2.3.1]

nativeWriteString(mNativePtr, val);

}

#### 2.3.1 nativeWriteString

[-> android\_os\_Parcel.cpp]

static void android\_os\_Parcel\_writeString(JNIEnv\* env, jclass clazz, jlong nativePtr, jstring val) {

Parcel\* parcel = reinterpret\_cast<Parcel\*>(nativePtr);

if (parcel != NULL) {

status\_t err = NO\_MEMORY;

if (val) {

const jchar\* str = env->GetStringCritical(val, 0);

if (str) {

//[见流程2.3.2]

err = parcel->writeString16(

reinterpret\_cast<const char16\_t\*>(str),

env->GetStringLength(val));

env->ReleaseStringCritical(val, str);

}

} else {

err = parcel->writeString16(NULL, 0);

}

if (err != NO\_ERROR) {

signalExceptionForError(env, clazz, err);

}

}

}

#### 2.3.2 writeString16

[-> Parcel.cpp]

status\_t Parcel::writeString16(const char16\_t\* str, size\_t len)

{

if (str == NULL) return writeInt32(-1);

status\_t err = writeInt32(len);

if (err == NO\_ERROR) {

len \*= sizeof(char16\_t);

uint8\_t\* data = (uint8\_t\*)writeInplace(len+sizeof(char16\_t));

if (data) {

//数据拷贝到data所指向的位置

memcpy(data, str, len);

\*reinterpret\_cast<char16\_t\*>(data+len) = 0;

return NO\_ERROR;

}

err = mError;

}

return err;

}

**Tips:** 除了writeString(),在Parcel.java中大量的native方法, 都是调用android\_os\_Parcel.cpp相对应的方法, 该方法再调用Parcel.cpp中对应的方法.   
调用流程: Parcel.java –> android\_os\_Parcel.cpp –> Parcel.cpp.

frameworks/base/core/java/android/os/Parcel.java

frameworks/base/core/jni/android\_os\_Parcel.cpp

frameworks/native/libs/binder/Parcel.cpp

简单说,就是

### 2.4 mRemote究竟为何物

mRemote的出生,要出先说说ActivityManagerProxy对象(简称AMP)创建说起, AMP是通过ActivityManagerNative.getDefault()来获取的.

#### 2.4.1 AMN.getDefault

[-> ActivityManagerNative.java]

static public IActivityManager getDefault() {

// [见流程2.4.2]

return gDefault.get();

}

gDefault的数据类型为Singleton<IActivityManager>, 这是一个单例模式, 接下来看看Singleto.get()的过程

#### 2.4.2 gDefault.get

public abstract class Singleton<IActivityManager> {

public final IActivityManager get() {

synchronized (this) {

if (mInstance == null) {

//首次调用create()来获取AMP对象[见流程2.4.3]

mInstance = create();

}

return mInstance;

}

}

}

首次调用时需要创建,创建完之后保持到mInstance对象,之后可直接使用.

#### 2.4.3 gDefault.create

private static final Singleton<IActivityManager> gDefault = new Singleton<IActivityManager>() {

protected IActivityManager create() {

//获取名为"activity"的服务

IBinder b = ServiceManager.getService("activity");

//创建AMP对象[见流程2.4.4]

IActivityManager am = asInterface(b);

return am;

}

};

文章[Binder系列7—framework层分析](http://gityuan.com/2015/11/21/binder-framework/#section-4)，可知ServiceManager.getService(“activity”)返回的是指向目标服务AMS的代理对象BinderProxy对象，由该代理对象可以找到目标服务AMS所在进程

#### 2.4.4 AMN.asInterface

[-> ActivityManagerNative.java]

public abstract class ActivityManagerNative extends Binder implements IActivityManager {

static public IActivityManager asInterface(IBinder obj) {

if (obj == null) {

return null;

}

//此处obj = BinderProxy, descriptor = "android.app.IActivityManager"; [见流程2.4.5]

IActivityManager in = (IActivityManager)obj.queryLocalInterface(descriptor);

if (in != null) { //此处为null

return in;

}

//[见流程2.4.6]

return new ActivityManagerProxy(obj);

}

...

}

此时obj为BinderProxy对象, 记录着远程进程system\_server中AMS服务的binder线程的handle.

#### 2.4.5 queryLocalInterface

[Binder.java]

public class Binder implements IBinder {

//对于Binder对象的调用,则返回值不为空

public IInterface queryLocalInterface(String descriptor) {

//mDescriptor的初始化在attachInterface()过程中赋值

if (mDescriptor.equals(descriptor)) {

return mOwner;

}

return null;

}

}

//由上一小节[2.4.4]调用的流程便是此处,返回Null

final class BinderProxy implements IBinder {

//BinderProxy对象的调用, 则返回值为空

public IInterface queryLocalInterface(String descriptor) {

return null;

}

}

对于Binder IPC的过程中, 同一个进程的调用则会是asInterface()方法返回的便是本地的Binder对象;对于不同进程的调用则会是远程代理对象BinderProxy.

#### 2.4.6 创建AMP

[-> ActivityManagerNative.java :: AMP]

class ActivityManagerProxy implements IActivityManager {

public ActivityManagerProxy(IBinder remote) {

mRemote = remote;

}

}

可知mRemote便是指向AMS服务的BinderProxy对象。

### 2.5 mRemote.transact

[-> Binder.java ::BinderProxy]

final class BinderProxy implements IBinder {

public boolean transact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {

//用于检测Parcel大小是否大于800k

Binder.checkParcel(this, code, data, "Unreasonably large binder buffer");

//【见2.6】

return transactNative(code, data, reply, flags);

}

}

mRemote.transact()方法中的code=START\_SERVICE\_TRANSACTION, data保存了descriptor，caller, intent, resolvedType, callingPackage, userId这6项信息。

transactNative是native方法，经过jni调用android\_os\_BinderProxy\_transact方法。

### 2.6 android\_os\_BinderProxy\_transact

[-> android\_util\_Binder.cpp]

static jboolean android\_os\_BinderProxy\_transact(JNIEnv\* env, jobject obj,

jint code, jobject dataObj, jobject replyObj, jint flags)

{

...

//将java Parcel转为c++ Parcel

Parcel\* data = parcelForJavaObject(env, dataObj);

Parcel\* reply = parcelForJavaObject(env, replyObj);

//gBinderProxyOffsets.mObject中保存的是new BpBinder(handle)对象

IBinder\* target = (IBinder\*) env->GetLongField(obj, gBinderProxyOffsets.mObject);

...

//此处便是BpBinder::transact()【见小节2.7】

status\_t err = target->transact(code, \*data, reply, flags);

...

//最后根据transact执行具体情况，抛出相应的Exception

signalExceptionForError(env, obj, err, true , data->dataSize());

return JNI\_FALSE;

}

gBinderProxyOffsets.mObject中保存的是BpBinder对象, 这是开机时Zygote调用AndroidRuntime::startReg方法来完成jni方法的注册.

其中register\_android\_os\_Binder()过程就有一个初始并注册BinderProxy的操作,完成gBinderProxyOffsets的赋值过程. 接下来就进入该方法.

### 2.7 BpBinder.transact

[-> BpBinder.cpp]

status\_t BpBinder::transact(

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

{

if (mAlive) {

// 【见小节2.8】

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

mHandle, code, data, reply, flags);

if (status == DEAD\_OBJECT) mAlive = 0;

return status;

}

return DEAD\_OBJECT;

}

IPCThreadState::self()采用单例模式，保证每个线程只有一个实例对象。

### 2.8 IPC.transact

[-> IPCThreadState.cpp]

status\_t IPCThreadState::transact(int32\_t handle,

uint32\_t code, const Parcel& data,

Parcel\* reply, uint32\_t flags)

{

status\_t err = data.errorCheck(); //数据错误检查

flags |= TF\_ACCEPT\_FDS;

....

if (err == NO\_ERROR) {

// 传输数据 【见小节2.9】

err = writeTransactionData(BC\_TRANSACTION, flags, handle, code, data, NULL);

}

if (err != NO\_ERROR) {

if (reply) reply->setError(err);

return (mLastError = err);

}

// 默认情况下,都是采用非oneway的方式, 也就是需要等待服务端的返回结果

if ((flags & TF\_ONE\_WAY) == 0) {

if (reply) {

//reply对象不为空 【见小节2.10】

err = waitForResponse(reply);

}else {

Parcel fakeReply;

err = waitForResponse(&fakeReply);

}

} else {

err = waitForResponse(NULL, NULL);

}

return err;

}

transact主要过程:

* 先执行writeTransactionData()已向Parcel数据类型的mOut写入数据，此时mIn还没有数据；
* 然后执行waitForResponse()方法，循环执行，直到收到应答消息. 调用talkWithDriver()跟驱动交互，收到应答消息，便会写入mIn, 则根据收到的不同响应吗，执行相应的操作。

此处调用waitForResponse根据是否有设置TF\_ONE\_WAY的标记:

* 当已设置oneway时, 则调用waitForResponse(NULL, NULL);
* 当未设置oneway时, 则调用waitForResponse(reply) 或 waitForResponse(&fakeReply)

### 2.9 IPC.writeTransactionData

[-> IPCThreadState.cpp]

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.ptr = 0;

tr.target.handle = handle; // handle指向AMS

tr.code = code; // START\_SERVICE\_TRANSACTION

tr.flags = binderFlags; // 0

tr.cookie = 0;

tr.sender\_pid = 0;

tr.sender\_euid = 0;

const status\_t err = data.errorCheck();

if (err == NO\_ERROR) {

// data为startService相关信息

tr.data\_size = data.ipcDataSize(); // mDataSize

tr.data.ptr.buffer = data.ipcData(); // mData指针

tr.offsets\_size = data.ipcObjectsCount()\*sizeof(binder\_size\_t); //mObjectsSize

tr.data.ptr.offsets = data.ipcObjects(); //mObjects指针

}

...

mOut.writeInt32(cmd); //cmd = BC\_TRANSACTION

mOut.write(&tr, sizeof(tr)); //写入binder\_transaction\_data数据

return NO\_ERROR;

}

将数据写入mOut

### 2.10 IPC.waitForResponse

status\_t IPCThreadState::waitForResponse(Parcel \*reply, status\_t \*acquireResult)

{

int32\_t cmd;

int32\_t err;

while (1) {

if ((err=talkWithDriver()) < NO\_ERROR) break; // 【见小节2.11】

err = mIn.errorCheck();

if (err < NO\_ERROR) break; //当存在error则退出循环

//每当跟Driver交互一次，若mIn收到数据则往下执行一次BR命令

if (mIn.dataAvail() == 0) continue;

cmd = mIn.readInt32();

switch (cmd) {

case BR\_TRANSACTION\_COMPLETE:

//只有当不需要reply, 也就是oneway时 才会跳出循环,否则还需要等待.

if (!reply && !acquireResult) goto finish; break;

case BR\_DEAD\_REPLY:

err = DEAD\_OBJECT; goto finish;

case BR\_FAILED\_REPLY:

err = FAILED\_TRANSACTION; goto finish;

case BR\_REPLY: ... goto finish;

default:

err = executeCommand(cmd); //【见小节2.12】

if (err != NO\_ERROR) goto finish;

break;

}

}

finish:

if (err != NO\_ERROR) {

if (reply) reply->setError(err); //将发送的错误代码返回给最初的调用者

}

return err;

}

在这个过程中, 收到以下任一BR\_命令，处理后便会退出waitForResponse()的状态:

* BR\_TRANSACTION\_COMPLETE: binder驱动收到BC\_TRANSACTION事件后的应答消息; 对于oneway transaction,当收到该消息,则完成了本次Binder通信;
* BR\_DEAD\_REPLY: 回复失败，往往是线程或节点为空. 则结束本次通信Binder;
* BR\_FAILED\_REPLY:回复失败，往往是transaction出错导致. 则结束本次通信Binder;
* BR\_REPLY: Binder驱动向Client端发送回应消息; 对于非oneway transaction时,当收到该消息,则完整地完成本次Binder通信;

除了以上命令，其他命令的处理流程【见小节2.12】

### 2.11 IPC.talkWithDriver

//mOut有数据，mIn还没有数据。doReceive默认值为true

status\_t IPCThreadState::talkWithDriver(bool doReceive)

{

binder\_write\_read bwr;

const bool needRead = mIn.dataPosition() >= mIn.dataSize();

const size\_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;

bwr.write\_size = outAvail;

bwr.write\_buffer = (uintptr\_t)mOut.data();

if (doReceive && needRead) {

//接收数据缓冲区信息的填充。当收到驱动的数据，则写入mIn

bwr.read\_size = mIn.dataCapacity();

bwr.read\_buffer = (uintptr\_t)mIn.data();

} else {

bwr.read\_size = 0;

bwr.read\_buffer = 0;

}

// 当同时没有输入和输出数据则直接返回

if ((bwr.write\_size == 0) && (bwr.read\_size == 0)) return NO\_ERROR;

bwr.write\_consumed = 0;

bwr.read\_consumed = 0;

status\_t err;

do {

//ioctl执行binder读写操作，经过syscall，进入Binder驱动。调用Binder\_ioctl【小节3.1】

if (ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr) >= 0)

err = NO\_ERROR;

else

err = -errno;

...

} while (err == -EINTR);

if (err >= NO\_ERROR) {

if (bwr.write\_consumed > 0) {

if (bwr.write\_consumed < mOut.dataSize())

mOut.remove(0, bwr.write\_consumed);

else

mOut.setDataSize(0);

}

if (bwr.read\_consumed > 0) {

mIn.setDataSize(bwr.read\_consumed);

mIn.setDataPosition(0);

}

return NO\_ERROR;

}

return err;

}

[binder\_write\_read结构体](http://gityuan.com/2015/11/01/binder-driver/#binderwriteread)用来与Binder设备交换数据的结构, 通过ioctl与mDriverFD通信，是真正与Binder驱动进行数据读写交互的过程。

### 2.12 IPC.executeCommand

status\_t IPCThreadState::executeCommand(int32\_t cmd)

{

BBinder\* obj;

RefBase::weakref\_type\* refs;

status\_t result = NO\_ERROR;

switch ((uint32\_t)cmd) {

case BR\_ERROR: ...

case BR\_OK: ...

case BR\_ACQUIRE: ...

case BR\_RELEASE: ...

case BR\_INCREFS: ...

case BR\_TRANSACTION: ... //Binder驱动向Server端发送消息

case BR\_DEAD\_BINDER: ...

case BR\_CLEAR\_DEATH\_NOTIFICATION\_DONE: ...

case BR\_NOOP: ...

case BR\_SPAWN\_LOOPER: ... //创建新binder线程

default: ...

}

}

再回到【小节2.11】，可知ioctl()方法经过syscall最终调用到Binder\_ioctl()方法.

## 三、Binder driver

#### 3.1 binder\_ioctl

[-> Binder.c]

由【小节2.11】传递过出来的参数 cmd=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;

//当binder\_stop\_on\_user\_error>=2时，则该线程加入等待队列并进入休眠状态. 该值默认为0

ret = wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);

...

binder\_lock(\_\_func\_\_);

//查找或创建binder\_thread结构体

thread = binder\_get\_thread(proc);

...

switch (cmd) {

case BINDER\_WRITE\_READ:

//【见小节3.2】

ret = binder\_ioctl\_write\_read(filp, cmd, arg, thread);

break;

...

}

ret = 0;

err:

if (thread)

thread->looper &= ~BINDER\_LOOPER\_STATE\_NEED\_RETURN;

binder\_unlock(\_\_func\_\_);

wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);

return ret;

}

首先,根据传递过来的文件句柄指针获取相应的binder\_proc结构体, 再从中查找binder\_thread,如果当前线程已经加入到proc的线程队列则直接返回， 如果不存在则创建binder\_thread，并将当前线程添加到当前的proc.

* 当返回值为-ENOMEM，则意味着内存不足，往往会出现创建binder\_thread对象失败;
* 当返回值为-EINVAL，则意味着CMD命令参数无效；

#### 3.2 binder\_ioctl\_write\_read

static int binder\_ioctl\_write\_read(struct file \*filp,

unsigned int cmd, unsigned long arg,

struct binder\_thread \*thread)

{

int ret = 0;

struct binder\_proc \*proc = filp->private\_data;

unsigned int size = \_IOC\_SIZE(cmd);

void \_\_user \*ubuf = (void \_\_user \*)arg;

struct binder\_write\_read bwr;

if (size != sizeof(struct binder\_write\_read)) {

ret = -EINVAL;

goto out;

}

//将用户空间bwr结构体拷贝到内核空间

if (copy\_from\_user(&bwr, ubuf, sizeof(bwr))) {

ret = -EFAULT;

goto out;

}

if (bwr.write\_size > 0) {

//将数据放入目标进程【见小节3.3】

ret = binder\_thread\_write(proc, thread,

bwr.write\_buffer,

bwr.write\_size,

&bwr.write\_consumed);

//当执行失败，则直接将内核bwr结构体写回用户空间，并跳出该方法

if (ret < 0) {

bwr.read\_consumed = 0;

if (copy\_to\_user\_preempt\_disabled(ubuf, &bwr, sizeof(bwr)))

ret = -EFAULT;

goto out;

}

}

if (bwr.read\_size > 0) {

//读取自己队列的数据 【见小节3.5】

ret = binder\_thread\_read(proc, thread, bwr.read\_buffer,

bwr.read\_size,

&bwr.read\_consumed,

filp->f\_flags & O\_NONBLOCK);

//当进程的todo队列有数据,则唤醒在该队列等待的进程

if (!list\_empty(&proc->todo))

wake\_up\_interruptible(&proc->wait);

//当执行失败，则直接将内核bwr结构体写回用户空间，并跳出该方法

if (ret < 0) {

if (copy\_to\_user\_preempt\_disabled(ubuf, &bwr, sizeof(bwr)))

ret = -EFAULT;

goto out;

}

}

if (copy\_to\_user(ubuf, &bwr, sizeof(bwr))) {

ret = -EFAULT;

goto out;

}

out:

return ret;

}

此时arg是一个binder\_write\_read结构体，mOut数据保存在write\_buffer，所以write\_size>0，但此时read\_size=0。首先,将用户空间bwr结构体拷贝到内核空间,然后执行binder\_thread\_write()操作.

#### 3.3 binder\_thread\_write

static int binder\_thread\_write(struct binder\_proc \*proc,

struct binder\_thread \*thread,

binder\_uintptr\_t binder\_buffer, size\_t size,

binder\_size\_t \*consumed)

{

uint32\_t cmd;

void \_\_user \*buffer = (void \_\_user \*)(uintptr\_t)binder\_buffer;

void \_\_user \*ptr = buffer + \*consumed;

void \_\_user \*end = buffer + size;

while (ptr < end && thread->return\_error == BR\_OK) {

//拷贝用户空间的cmd命令，此时为BC\_TRANSACTION

if (get\_user(cmd, (uint32\_t \_\_user \*)ptr)) -EFAULT;

ptr += sizeof(uint32\_t);

switch (cmd) {

case BC\_TRANSACTION:

case BC\_REPLY: {

struct binder\_transaction\_data tr;

//拷贝用户空间的binder\_transaction\_data

if (copy\_from\_user(&tr, ptr, sizeof(tr))) return -EFAULT;

ptr += sizeof(tr);

// 见小节3.4】

binder\_transaction(proc, thread, &tr, cmd == BC\_REPLY);

break;

}

...

}

\*consumed = ptr - buffer;

}

return 0;

}

不断从binder\_buffer所指向的地址获取cmd, 当只有BC\_TRANSACTION或者BC\_REPLY时, 则调用binder\_transaction()来处理事务.

#### 3.4 binder\_transaction

发送的是BC\_TRANSACTION时，此时reply=0。

static void binder\_transaction(struct binder\_proc \*proc,

struct binder\_thread \*thread,

struct binder\_transaction\_data \*tr, int reply){

struct binder\_transaction \*t;

struct binder\_work \*tcomplete;

binder\_size\_t \*offp, \*off\_end;

binder\_size\_t off\_min;

struct binder\_proc \*target\_proc;

struct binder\_thread \*target\_thread = NULL;

struct binder\_node \*target\_node = NULL;

struct list\_head \*target\_list;

wait\_queue\_head\_t \*target\_wait;

struct binder\_transaction \*in\_reply\_to = NULL;

if (reply) {

...

}else {

if (tr->target.handle) {

struct binder\_ref \*ref;

// 由handle 找到相应 binder\_ref, 由binder\_ref 找到相应 binder\_node

ref = binder\_get\_ref(proc, tr->target.handle);

target\_node = ref->node;

} else {

target\_node = binder\_context\_mgr\_node;

}

// 由binder\_node 找到相应 binder\_proc

target\_proc = target\_node->proc;

}

if (target\_thread) {

e->to\_thread = target\_thread->pid;

target\_list = &target\_thread->todo;

target\_wait = &target\_thread->wait;

} else {

//首次执行target\_thread为空

target\_list = &target\_proc->todo;

target\_wait = &target\_proc->wait;

}

t = kzalloc(sizeof(\*t), GFP\_KERNEL);

tcomplete = kzalloc(sizeof(\*tcomplete), GFP\_KERNEL);

//非oneway的通信方式，把当前thread保存到transaction的from字段

if (!reply && !(tr->flags & TF\_ONE\_WAY))

t->from = thread;

else

t->from = NULL;

t->sender\_euid = task\_euid(proc->tsk);

t->to\_proc = target\_proc; //此次通信目标进程为system\_server

t->to\_thread = target\_thread;

t->code = tr->code; //此次通信code = START\_SERVICE\_TRANSACTION

t->flags = tr->flags; // 此次通信flags = 0

t->priority = task\_nice(current);

//从目标进程target\_proc中分配内存空间【3.4.1】

t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,

tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));

t->buffer->allow\_user\_free = 0;

t->buffer->transaction = t;

t->buffer->target\_node = target\_node;

if (target\_node)

binder\_inc\_node(target\_node, 1, 0, NULL); //引用计数加1

//binder对象的偏移量

offp = (binder\_size\_t \*)(t->buffer->data + ALIGN(tr->data\_size, sizeof(void \*)));

//分别拷贝用户空间的binder\_transaction\_data中ptr.buffer和ptr.offsets到目标进程的binder\_buffer

copy\_from\_user(t->buffer->data,

(const void \_\_user \*)(uintptr\_t)tr->data.ptr.buffer, tr->data\_size);

copy\_from\_user(offp,

(const void \_\_user \*)(uintptr\_t)tr->data.ptr.offsets, tr->offsets\_size);

off\_end = (void \*)offp + tr->offsets\_size;

for (; offp < off\_end; offp++) {

struct flat\_binder\_object \*fp;

fp = (struct flat\_binder\_object \*)(t->buffer->data + \*offp);

off\_min = \*offp + sizeof(struct flat\_binder\_object);

switch (fp->type) {

...

case BINDER\_TYPE\_HANDLE:

case BINDER\_TYPE\_WEAK\_HANDLE: {

//处理引用计数情况

struct binder\_ref \*ref = binder\_get\_ref(proc, fp->handle);

if (ref->node->proc == target\_proc) {

if (fp->type == BINDER\_TYPE\_HANDLE)

fp->type = BINDER\_TYPE\_BINDER;

else

fp->type = BINDER\_TYPE\_WEAK\_BINDER;

fp->binder = ref->node->ptr;

fp->cookie = ref->node->cookie;

binder\_inc\_node(ref->node, fp->type == BINDER\_TYPE\_BINDER, 0, NULL);

} else {

struct binder\_ref \*new\_ref;

new\_ref = binder\_get\_ref\_for\_node(target\_proc, ref->node);

fp->handle = new\_ref->desc;

binder\_inc\_ref(new\_ref, fp->type == BINDER\_TYPE\_HANDLE, NULL);

}

} break;

...

default:

return\_error = BR\_FAILED\_REPLY;

goto err\_bad\_object\_type;

}

}

if (reply) {

//BC\_REPLY的过程

binder\_pop\_transaction(target\_thread, in\_reply\_to);

} else if (!(t->flags & TF\_ONE\_WAY)) {

//BC\_TRANSACTION 且 非oneway,则设置事务栈信息

t->need\_reply = 1;

t->from\_parent = thread->transaction\_stack;

thread->transaction\_stack = t;

} else {

//BC\_TRANSACTION 且 oneway,则加入异步todo队列

if (target\_node->has\_async\_transaction) {

target\_list = &target\_node->async\_todo;

target\_wait = NULL;

} else

target\_node->has\_async\_transaction = 1;

}

//将BINDER\_WORK\_TRANSACTION添加到目标队列,即target\_proc->todo

t->work.type = BINDER\_WORK\_TRANSACTION;

list\_add\_tail(&t->work.entry, target\_list);

//将BINDER\_WORK\_TRANSACTION\_COMPLETE添加到当前线程队列，即thread->todo

tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;

list\_add\_tail(&tcomplete->entry, &thread->todo);

//唤醒等待队列，本次通信的目标队列为target\_proc->wait

if (target\_wait)

wake\_up\_interruptible(target\_wait);

return;

}

主要功能:

1. 查询目标进程的过程： handle -> binder\_ref -> binder\_node -> binder\_proc
2. 将BINDER\_WORK\_TRANSACTION添加到目标队列target\_list:
   * call事务， 则目标队列target\_list=target\_proc->todo;
   * reply事务，则目标队列target\_list=target\_thread->todo;
   * async事务，则目标队列target\_list=target\_node->async\_todo.
3. 数据拷贝
   * 将用户空间binder\_transaction\_data中ptr.buffer和ptr.offsets拷贝到目标进程的binder\_buffer->data；
   * 这就是只拷贝一次的真理所在；
4. 设置事务栈信息
   * BC\_TRANSACTION且非oneway, 则将当前事务添加到thread->transaction\_stack；
5. 事务分发过程：
   * 将BINDER\_WORK\_TRANSACTION添加到目标队列(此时为target\_proc->todo队列);
   * 将BINDER\_WORK\_TRANSACTION\_COMPLETE添加到当前线程thread->todo队列;
6. 唤醒目标进程target\_proc开始执行事务。

该方法中proc/thread是指当前发起方的进程信息，而binder\_proc是指目标接收端进程。 此时当前线程thread的todo队列已经有事务, 接下来便会进入binder\_thread\_read来处理相关的事务.

#### 3.4.1 binder\_alloc\_buf

static struct binder\_buffer \*binder\_alloc\_buf(struct binder\_proc \*proc,

size\_t data\_size, size\_t offsets\_size, int is\_async)

{

struct rb\_node \*n = proc->free\_buffers.rb\_node;

struct binder\_buffer \*buffer;

size\_t buffer\_size;

struct rb\_node \*best\_fit = NULL;

void \*has\_page\_addr;

void \*end\_page\_addr;

size\_t size;

..

size = ALIGN(data\_size, sizeof(void \*)) + ALIGN(offsets\_size, sizeof(void \*));

if (is\_async && proc->free\_async\_space < size + sizeof(struct binder\_buffer)) {

return NULL; // 剩余可用的异步空间，小于所需的大小

}

while (n) { //从binder\_buffer的红黑树中查找大小相等的buffer块

buffer = rb\_entry(n, struct binder\_buffer, rb\_node);

buffer\_size = binder\_buffer\_size(proc, buffer);

if (size < buffer\_size) {

best\_fit = n;

n = n->rb\_left;

} else if (size > buffer\_size)

n = n->rb\_right;

else {

best\_fit = n;

break;

}

}

...

if (n == NULL) {

buffer = rb\_entry(best\_fit, struct binder\_buffer, rb\_node);

buffer\_size = binder\_buffer\_size(proc, buffer);

}

has\_page\_addr =(void \*)(((uintptr\_t)buffer->data + buffer\_size) & PAGE\_MASK);

if (n == NULL) {

if (size + sizeof(struct binder\_buffer) + 4 >= buffer\_size)

buffer\_size = size;

else

buffer\_size = size + sizeof(struct binder\_buffer);

}

//末端地址

end\_page\_addr = (void \*)PAGE\_ALIGN((uintptr\_t)buffer->data + buffer\_size);

...

//分配物理页

if (binder\_update\_page\_range(proc, 1,

(void \*)PAGE\_ALIGN((uintptr\_t)buffer->data), end\_page\_addr, NULL))

return NULL;

rb\_erase(best\_fit, &proc->free\_buffers);

buffer->free = 0;

binder\_insert\_allocated\_buffer(proc, buffer);

if (buffer\_size != size) {

struct binder\_buffer \*new\_buffer = (void \*)buffer->data + size;

list\_add(&new\_buffer->entry, &buffer->entry);

new\_buffer->free = 1;

binder\_insert\_free\_buffer(proc, new\_buffer);

}

buffer->data\_size = data\_size;

buffer->offsets\_size = offsets\_size;

buffer->async\_transaction = is\_async;

if (is\_async) { //调整异步可用内存空间大小

proc->free\_async\_space -= size + sizeof(struct binder\_buffer);

}

return buffer;

}

#### 3.5 binder\_thread\_read

binder\_thread\_read（）{

//当已使用字节数为0时，将BR\_NOOP响应码放入指针ptr

if (\*consumed == 0) {

if (put\_user(BR\_NOOP, (uint32\_t \_\_user \*)ptr))

return -EFAULT;

ptr += sizeof(uint32\_t);

}

retry:

//binder\_transaction()已设置transaction\_stack不为空，则wait\_for\_proc\_work为false.

wait\_for\_proc\_work = thread->transaction\_stack == NULL &&

list\_empty(&thread->todo);

thread->looper |= BINDER\_LOOPER\_STATE\_WAITING;

if (wait\_for\_proc\_work)

proc->ready\_threads++; //进程中空闲binder线程加1

//只有当前线程todo队列为空，并且transaction\_stack也为空，才会开始处于当前进程的事务

if (wait\_for\_proc\_work) {

if (non\_block) {

...

} else

//当进程todo队列没有数据,则进入休眠等待状态

ret = wait\_event\_freezable\_exclusive(proc->wait, binder\_has\_proc\_work(proc, thread));

} else {

if (non\_block) {

...

} else

//当线程todo队列有数据则执行往下执行；当线程todo队列没有数据，则进入休眠等待状态

ret = wait\_event\_freezable(thread->wait, binder\_has\_thread\_work(thread));

}

if (wait\_for\_proc\_work)

proc->ready\_threads--; //退出等待状态, 则进程中空闲binder线程减1

thread->looper &= ~BINDER\_LOOPER\_STATE\_WAITING;

...

while (1) {

uint32\_t cmd;

struct binder\_transaction\_data tr;

struct binder\_work \*w;

struct binder\_transaction \*t = NULL;

//先从线程todo队列获取事务数据

if (!list\_empty(&thread->todo)) {

w = list\_first\_entry(&thread->todo, struct binder\_work, entry);

// 线程todo队列没有数据, 则从进程todo对获取事务数据

} else if (!list\_empty(&proc->todo) && wait\_for\_proc\_work) {

w = list\_first\_entry(&proc->todo, struct binder\_work, entry);

} else {

//没有数据,则返回retry

if (ptr - buffer == 4 &&

!(thread->looper & BINDER\_LOOPER\_STATE\_NEED\_RETURN))

goto retry;

break;

}

switch (w->type) {

case BINDER\_WORK\_TRANSACTION:

//获取transaction数据

t = container\_of(w, struct binder\_transaction, work);

break;

case BINDER\_WORK\_TRANSACTION\_COMPLETE:

cmd = BR\_TRANSACTION\_COMPLETE;

//将BR\_TRANSACTION\_COMPLETE写入\*ptr，并跳出循环。

put\_user(cmd, (uint32\_t \_\_user \*)ptr)；

list\_del(&w->entry);

kfree(w);

break;

case BINDER\_WORK\_NODE: ... break;

case BINDER\_WORK\_DEAD\_BINDER:

case BINDER\_WORK\_DEAD\_BINDER\_AND\_CLEAR:

case BINDER\_WORK\_CLEAR\_DEATH\_NOTIFICATION: ... break;

}

//只有BINDER\_WORK\_TRANSACTION命令才能继续往下执行

if (!t)

continue;

if (t->buffer->target\_node) {

//获取目标node

struct binder\_node \*target\_node = t->buffer->target\_node;

tr.target.ptr = target\_node->ptr;

tr.cookie = target\_node->cookie;

t->saved\_priority = task\_nice(current);

...

cmd = BR\_TRANSACTION; //设置命令为BR\_TRANSACTION

} else {

tr.target.ptr = NULL;

tr.cookie = NULL;

cmd = BR\_REPLY; //设置命令为BR\_REPLY

}

tr.code = t->code;

tr.flags = t->flags;

tr.sender\_euid = t->sender\_euid;

if (t->from) {

struct task\_struct \*sender = t->from->proc->tsk;

//当非oneway的情况下,将调用者进程的pid保存到sender\_pid

tr.sender\_pid = task\_tgid\_nr\_ns(sender,

current->nsproxy->pid\_ns);

} else {

//当oneway的的情况下,则该值为0

tr.sender\_pid = 0;

}

tr.data\_size = t->buffer->data\_size;

tr.offsets\_size = t->buffer->offsets\_size;

tr.data.ptr.buffer = (void \*)t->buffer->data + proc->user\_buffer\_offset;

tr.data.ptr.offsets = tr.data.ptr.buffer +

ALIGN(t->buffer->data\_size, sizeof(void \*));

//将cmd和数据写回用户空间

if (put\_user(cmd, (uint32\_t \_\_user \*)ptr))

return -EFAULT;

ptr += sizeof(uint32\_t);

if (copy\_to\_user(ptr, &tr, sizeof(tr)))

return -EFAULT;

ptr += sizeof(tr);

list\_del(&t->work.entry);

t->buffer->allow\_user\_free = 1;

if (cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)) {

t->to\_parent = thread->transaction\_stack;

t->to\_thread = thread;

thread->transaction\_stack = t;

} else {

t->buffer->transaction = NULL;

kfree(t); //通信完成,则运行释放

}

break;

}

done:

\*consumed = ptr - buffer;

//当满足请求线程加已准备线程数等于0，已启动线程数小于最大线程数(15)，

//且looper状态为已注册或已进入时创建新的线程。

if (proc->requested\_threads + proc->ready\_threads == 0 &&

proc->requested\_threads\_started < proc->max\_threads &&

(thread->looper & (BINDER\_LOOPER\_STATE\_REGISTERED |

BINDER\_LOOPER\_STATE\_ENTERED))) {

proc->requested\_threads++;

// 生成BR\_SPAWN\_LOOPER命令，用于创建新的线程

put\_user(BR\_SPAWN\_LOOPER, (uint32\_t \_\_user \*)buffer)；

}

return 0;

}

该方法功能说明:

此处wait\_for\_proc\_work是指当前线程todo队列为空，并且transaction\_stack也为空,该值为true.

1. 当wait\_for\_proc\_work = false, 则进入线程的等待队列thread->wait, 直到thread->todo队列有事务才往下执行;
   * 获取并处理thread->todo队列中的事务;将相应的cmd和数据写回用户空间.
2. 当wait\_for\_proc\_work = true, 则进入线程的等待队列proc->wait, 直到proc->todo队列有事务才往下执行;
   * 获取并处理proc->todo队列中的事务;将相应的cmd和数据写回用户空间.

到这里,可能有人好奇,对于[小节3.4]介绍了target\_list有3种, 这里只会处理前2种:thread->todo, proc->todo.那么对于 target\_node->async\_todo的处理过程时间呢? [见小节5.4]

#### 3.6 下一步何去何从

1. 执行完binder\_thread\_write方法后, 通过binder\_transaction()首先写入BINDER\_WORK\_TRANSACTION\_COMPLETE写入当前线程.
2. 这时bwr.read\_size > 0, 回到binder\_ioctl\_write\_read方法, 便开始执行binder\_thread\_read();
3. 在binder\_thread\_read()方法, 将获取cmd=BR\_TRANSACTION\_COMPLETE, 再将cmd和数据写回用户空间;
4. 一次Binder\_ioctl完成,接着回调用户空间方法talkWithDriver(),刚才的数据以写入mIn.
5. 这时mIn有可读数据, 回到【小节2.10】IPC.waitForResponse()方法,完成BR\_TRANSACTION\_COMPLETE过程. 如果本次transaction采用非oneway方式, 这次Binder通信便完成, 否则还是要等待Binder服务端的返回。

对于startService过程, 采用的便是非oneway方式,那么发起者进程还会继续停留在waitForResponse()方法,继续talkWithDriver()，然后休眠在binder\_thread\_read()的wait\_event\_freezable()过程，等待当前线程的todo队列有数据的到来，即等待收到BR\_REPLY消息.

由于在前面binder\_transaction()除了向自己所在线程写入了BINDER\_WORK\_TRANSACTION\_COMPLETE, 还向目标进程(此处为system\_server)写入了BINDER\_WORK\_TRANSACTION命令，那么接下里介绍system\_server进程的工作。

## 四. 回到用户空间

system\_server的binder线程是如何运转的，那么就需要从Binder线程的创建开始说起， Binder线程的创建有两种方式：

* ProcessState::self()->startThreadPool();
* IPCThreadState::self()->joinThreadPool();

从文章[addService 小节4.1](http://gityuan.com/2015/11/14/binder-add-service/)，可知，调用链如下： startThreadPool()过程会创建新Binder线程，再经过层层调用也会进入joinThreadPool()方法。 system\_server的binder线程从IPC.joinThreadPool –> IPC.getAndExecuteCommand() -> IPC.talkWithDriver() ,但talkWithDriver收到事务之后, 便进入IPC.executeCommand()方法。

接下来从joinThreadPool说起：

### 4.1 IPC.joinThreadPool

void IPCThreadState::joinThreadPool(bool isMain)

{

mOut.writeInt32(isMain ? BC\_ENTER\_LOOPER : BC\_REGISTER\_LOOPER);

set\_sched\_policy(mMyThreadId, SP\_FOREGROUND);

status\_t result;

do {

processPendingDerefs(); //处理对象引用

result = getAndExecuteCommand();//获取并执行命令【见小节4.2】

if (result < NO\_ERROR && result != TIMED\_OUT && result != -ECONNREFUSED && result != -EBADF) {

ALOGE("getAndExecuteCommand(fd=%d) returned unexpected error %d, aborting",

mProcess->mDriverFD, result);

abort();

}

//对于binder非主线程不再使用，则退出

if(result == TIMED\_OUT && !isMain) {

break;

}

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

mOut.writeInt32(BC\_EXIT\_LOOPER);

talkWithDriver(false);

}

### 4.2 IPC.getAndExecuteCommand

status\_t IPCThreadState::getAndExecuteCommand()

{

status\_t result;

int32\_t cmd;

result = talkWithDriver(); //该Binder Driver进行交互

if (result >= NO\_ERROR) {

size\_t IN = mIn.dataAvail();

if (IN < sizeof(int32\_t)) return result;

cmd = mIn.readInt32(); //读取命令

pthread\_mutex\_lock(&mProcess->mThreadCountLock);

mProcess->mExecutingThreadsCount++;

pthread\_mutex\_unlock(&mProcess->mThreadCountLock);

result = executeCommand(cmd); //【见小节4.3】

pthread\_mutex\_lock(&mProcess->mThreadCountLock);

mProcess->mExecutingThreadsCount--;

pthread\_cond\_broadcast(&mProcess->mThreadCountDecrement);

pthread\_mutex\_unlock(&mProcess->mThreadCountLock);

set\_sched\_policy(mMyThreadId, SP\_FOREGROUND);

}

return result;

}

此时system\_server的binder线程空闲便是停留在binder\_thread\_read()方法来处理进程/线程新的事务。 由【小节3.4】可知收到的是BINDER\_WORK\_TRANSACTION命令, 再经过inder\_thread\_read()后生成命令cmd=BR\_TRANSACTION.再将cmd和数据写回用户空间。

### 4.3 IPC.executeCommand

status\_t IPCThreadState::executeCommand(int32\_t cmd)

{

BBinder\* obj;

RefBase::weakref\_type\* refs;

status\_t result = NO\_ERROR;

switch ((uint32\_t)cmd) {

case BR\_TRANSACTION:

{

binder\_transaction\_data tr;

result = mIn.read(&tr, sizeof(tr)); //读取mIn数据

if (result != NO\_ERROR) break;

Parcel buffer;

//当buffer对象回收时，则会调用freeBuffer来回收内存【见小节4.3.1】

buffer.ipcSetDataReference(

reinterpret\_cast<const uint8\_t\*>(tr.data.ptr.buffer),

tr.data\_size,

reinterpret\_cast<const binder\_size\_t\*>(tr.data.ptr.offsets),

tr.offsets\_size/sizeof(binder\_size\_t), freeBuffer, this);

const pid\_t origPid = mCallingPid;

const uid\_t origUid = mCallingUid;

const int32\_t origStrictModePolicy = mStrictModePolicy;

const int32\_t origTransactionBinderFlags = mLastTransactionBinderFlags;

//设置调用者的pid和uid

mCallingPid = tr.sender\_pid;

mCallingUid = tr.sender\_euid;

mLastTransactionBinderFlags = tr.flags;

int curPrio = getpriority(PRIO\_PROCESS, mMyThreadId);

if (gDisableBackgroundScheduling) {

... //不进入此分支

} else {

if (curPrio >= ANDROID\_PRIORITY\_BACKGROUND) {

set\_sched\_policy(mMyThreadId, SP\_BACKGROUND);

}

}

Parcel reply;

status\_t error;

if (tr.target.ptr) {

//尝试通过弱引用获取强引用

if (reinterpret\_cast<RefBase::weakref\_type\*>(

tr.target.ptr)->attemptIncStrong(this)) {

// tr.cookie里存放的是BBinder子类JavaBBinder [见流程4.4]

error = reinterpret\_cast<BBinder\*>(tr.cookie)->transact(tr.code, buffer,

&reply, tr.flags);

reinterpret\_cast<BBinder\*>(tr.cookie)->decStrong(this);

} else {

error = UNKNOWN\_TRANSACTION;

}

} else {

error = the\_context\_object->transact(tr.code, buffer, &reply, tr.flags);

}

if ((tr.flags & TF\_ONE\_WAY) == 0) {

if (error < NO\_ERROR) reply.setError(error);

//对于非oneway, 需要reply通信过程,则向Binder驱动发送BC\_REPLY命令【见小节4.3.1】

sendReply(reply, 0);

}

//恢复pid和uid信息

mCallingPid = origPid;

mCallingUid = origUid;

...

}

break;

case ...

default:

result = UNKNOWN\_ERROR;

break;

}

if (result != NO\_ERROR) {

mLastError = result;

}

return result;

}

* 对于oneway的场景, 执行完本次transact()则全部结束.
* 对于非oneway, 需要reply的通信过程,则向Binder驱动发送BC\_REPLY命令【见小节5.1】

#### 4.3.1 ipcSetDataReference

[-> Parcel.cpp]

void Parcel::ipcSetDataReference(const uint8\_t\* data, size\_t dataSize,

const binder\_size\_t\* objects, size\_t objectsCount, release\_func relFunc, void\* relCookie)

{

binder\_size\_t minOffset = 0;

freeDataNoInit(); //【见小节4.3.2】

mError = NO\_ERROR;

mData = const\_cast<uint8\_t\*>(data);

mDataSize = mDataCapacity = dataSize;

mDataPos = 0;

mObjects = const\_cast<binder\_size\_t\*>(objects);

mObjectsSize = mObjectsCapacity = objectsCount;

mNextObjectHint = 0;

mOwner = relFunc;

mOwnerCookie = relCookie;

for (size\_t i = 0; i < mObjectsSize; i++) {

binder\_size\_t offset = mObjects[i];

if (offset < minOffset) {

mObjectsSize = 0;

break;

}

minOffset = offset + sizeof(flat\_binder\_object);

}

scanForFds();

}

该方法的功能，Parcel成员变量说明：

* mData：parcel数据起始地址
* mDataSize：parcel数据大小
* mObjects：flat\_binder\_object地址偏移量
* mObjectsSize：parcel中flat\_binder\_object个数
* mOwner：释放函数freebuffer
* mOwnerCookie：释放函数所需信息

#### 4.3.2 freeDataNoInit

[-> Parcel.cpp]

void Parcel::freeDataNoInit()

{

if (mOwner) {

mOwner(this, mData, mDataSize, mObjects, mObjectsSize, mOwnerCookie);

} else { //mOwner为空， 进入该分支

releaseObjects(); //【见小节4.3.3】

if (mData) {

pthread\_mutex\_lock(&gParcelGlobalAllocSizeLock);

if (mDataCapacity <= gParcelGlobalAllocSize) {

gParcelGlobalAllocSize = gParcelGlobalAllocSize - mDataCapacity;

} else {

gParcelGlobalAllocSize = 0;

}

if (gParcelGlobalAllocCount > 0) {

gParcelGlobalAllocCount--;

}

pthread\_mutex\_unlock(&gParcelGlobalAllocSizeLock);

free(mData);

}

if (mObjects) free(mObjects);

}

}

#### 4.3.3 releaseObjects

void Parcel::releaseObjects()

{

const sp<ProcessState> proc(ProcessState::self());

size\_t i = mObjectsSize;

uint8\_t\* const data = mData;

binder\_size\_t\* const objects = mObjects;

while (i > 0) {

i--;

const flat\_binder\_object\* flat

= reinterpret\_cast<flat\_binder\_object\*>(data+objects[i]);

//【见小节4.3.4】

release\_object(proc, \*flat, this, &mOpenAshmemSize);

}

}

#### 4.3.4 release\_object

static void release\_object(const sp<ProcessState>& proc, const flat\_binder\_object& obj, const void\* who, size\_t\* outAshmemSize) {

switch (obj.type) {

case BINDER\_TYPE\_BINDER:

if (obj.binder) {

reinterpret\_cast<IBinder\*>(obj.cookie)->decStrong(who);

}

return;

case BINDER\_TYPE\_WEAK\_BINDER:

if (obj.binder)

reinterpret\_cast<RefBase::weakref\_type\*>(obj.binder)->decWeak(who);

return;

case BINDER\_TYPE\_HANDLE: {

const sp<IBinder> b = proc->getStrongProxyForHandle(obj.handle);

if (b != NULL) {

b->decStrong(who);

}

return;

}

case BINDER\_TYPE\_WEAK\_HANDLE: {

const wp<IBinder> b = proc->getWeakProxyForHandle(obj.handle);

if (b != NULL) b.get\_refs()->decWeak(who);

return;

}

case BINDER\_TYPE\_FD: {

...

return;

}

}

}

根据flat\_binder\_object的类型，来决定减少相应的强弱引用。

#### 4.3.5 ~Parcel

[-> Parcel.cpp]

当[小节4.3]executeCommand执行完成后， 便会释放局部变量Parcel buffer，则会析构Parcel。

Parcel::~Parcel()

{

freeDataNoInit();

}

void Parcel::freeDataNoInit()

{

if (mOwner) { //此处mOwner等于freeBuffer 【见小节4.3.6】

mOwner(this, mData, mDataSize, mObjects, mObjectsSize, mOwnerCookie);

} else {

...

}

}

接下来，进入IPC的freeBuffer过程。

#### 4.3.6 freeBuffer

[-> IPCThreadState.cpp]

void IPCThreadState::freeBuffer(Parcel\* parcel, const uint8\_t\* data,

size\_t /\*dataSize\*/,

const binder\_size\_t\* /\*objects\*/,

size\_t /\*objectsSize\*/, void\* /\*cookie\*/)

{

if (parcel != NULL) parcel->closeFileDescriptors();

IPCThreadState\* state = self();

state->mOut.writeInt32(BC\_FREE\_BUFFER);

state->mOut.writePointer((uintptr\_t)data);

}

向Binder驱动写入BC\_FREE\_BUFFER命令。

### 4.4 BBinder.transact

[-> Binder.cpp ::BBinder ]

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); //【见流程4.5】

break;

}

if (reply != NULL) {

reply->setDataPosition(0);

}

return err;

}

### 4.5 JavaBBinder.onTransact

[-> android\_util\_Binder.cpp]

virtual status\_t onTransact(

uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags = 0)

{

JNIEnv\* env = javavm\_to\_jnienv(mVM);

IPCThreadState\* thread\_state = IPCThreadState::self();

//调用Binder.execTransact [见流程4.6]

jboolean res = env->CallBooleanMethod(mObject, gBinderOffsets.mExecTransact,

code, reinterpret\_cast<jlong>(&data), reinterpret\_cast<jlong>(reply), flags);

jthrowable excep = env->ExceptionOccurred();

if (excep) {

res = JNI\_FALSE;

//发生异常, 则清理JNI本地引用

env->DeleteLocalRef(excep);

}

...

return res != JNI\_FALSE ? NO\_ERROR : UNKNOWN\_TRANSACTION;

}

还记得AndroidRuntime::startReg过程吗, 其中有一个过程便是register\_android\_os\_Binder(),该过程会把gBinderOffsets.mExecTransact便是Binder.java中的execTransact()方法.详见见[Binder系列7—framework层分析](http://gityuan.com/2015/11/21/binder-framework/)文章中的第二节初始化的过程.

另外,此处mObject是在服务注册addService过程,会调用writeStrongBinder方法, 将Binder对象传入了JavaBBinder构造函数的参数, 最终赋值给mObject. 在本次通信过程中Object为ActivityManagerNative对象.

此处斗转星移, 从C++代码回到了Java代码. 进入AMN.execTransact, 由于AMN继续于Binder对象, 接下来进入Binder.execTransact

### 4.6 Binder.execTransact

[Binder.java]

private boolean execTransact(int code, long dataObj, long replyObj, int flags) {

Parcel data = Parcel.obtain(dataObj);

Parcel reply = Parcel.obtain(replyObj);

boolean res;

try {

// 调用子类AMN.onTransact方法 [见流程4.7]

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

} catch (RemoteException e) {

if ((flags & FLAG\_ONEWAY) != 0) {

...

} else {

//非oneway的方式,则会将异常写回reply

reply.setDataPosition(0);

reply.writeException(e);

}

res = true;

} catch (RuntimeException e) {

if ((flags & FLAG\_ONEWAY) != 0) {

...

} else {

reply.setDataPosition(0);

reply.writeException(e);

}

res = true;

} catch (OutOfMemoryError e) {

RuntimeException re = new RuntimeException("Out of memory", e);

reply.setDataPosition(0);

reply.writeException(re);

res = true;

}

reply.recycle();

data.recycle();

return res;

}

当发生RemoteException, RuntimeException, OutOfMemoryError, 对于非oneway的情况下都会把异常传递给调用者.

### 4.7 AMN.onTransact

[-> ActivityManagerNative.java]

public boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {

switch (code) {

...

case START\_SERVICE\_TRANSACTION: {

data.enforceInterface(IActivityManager.descriptor);

IBinder b = data.readStrongBinder();

//生成ApplicationThreadNative的代理对象，即ApplicationThreadProxy对象

IApplicationThread app = ApplicationThreadNative.asInterface(b);

Intent service = Intent.CREATOR.createFromParcel(data);

String resolvedType = data.readString();

String callingPackage = data.readString();

int userId = data.readInt();

//调用ActivityManagerService的startService()方法【见流程4.8】

ComponentName cn = startService(app, service, resolvedType, callingPackage, userId);

reply.writeNoException();

ComponentName.writeToParcel(cn, reply);

return true;

}

}

### 4.8 AMS.startService

public ComponentName startService(IApplicationThread caller, Intent service, String resolvedType, String callingPackage, int userId) throws TransactionTooLargeException {

synchronized(this) {

...

ComponentName res = mServices.startServiceLocked(caller, service,

resolvedType, callingPid, callingUid, callingPackage, userId);

Binder.restoreCallingIdentity(origId);

return res;

}

}

历经千山万水, 总算是进入了AMS.startService. 当system\_server收到BR\_TRANSACTION的过程后，通信并没有完全结束，还需将服务启动完成的回应消息 告诉给发起端进程。

## 五. Reply流程

还记得前面【小节2.10】IPC.waitForResponse()过程，对于非oneway的方式，还仍在一直等待system\_server这边的响应呢，只有收到BR\_REPLY，或者BR\_DEAD\_REPLY，或者BR\_FAILED\_REPLY，再或许其他BR\_命令执行出错的情况下，该waitForResponse()才会退出。

BR\_REPLY命令是如何来的呢？【小节4.3】IPC.executeCommand()过程处理完BR\_TRANSACTION命令的同时，还会通过sendReply()向Binder Driver发送BC\_REPLY消息，接下来从该方法说起。

#### 5.1 IPC.sendReply

status\_t IPCThreadState::sendReply(const Parcel& reply, uint32\_t flags)

{

status\_t err;

status\_t statusBuffer;

//[见小节2.10]

err = writeTransactionData(BC\_REPLY, flags, -1, 0, reply, &statusBuffer);

if (err < NO\_ERROR) return err;

//[见小节5.3]

return waitForResponse(NULL, NULL);

}

先将数据写入mOut；再进waitForResponse，等待应答，此时同理也是等待BR\_TRANSACTION\_COMPLETE。 同理经过IPC.talkWithDriver -> binder\_ioctl -> binder\_ioctl\_write\_read -> binder\_thread\_write， 再就是进入binder\_transaction方法。

#### 5.2 BC\_REPLY

// reply =true

static void binder\_transaction(struct binder\_proc \*proc,

struct binder\_thread \*thread,

struct binder\_transaction\_data \*tr, int reply)

{

...

if (reply) {

in\_reply\_to = thread->transaction\_stack; //接收端的事务栈

...

thread->transaction\_stack = in\_reply\_to->to\_parent;

target\_thread = in\_reply\_to->from; //发起端的线程

//发起端线程不能为空

if (target\_thread == NULL) {

return\_error = BR\_DEAD\_REPLY;

goto err\_dead\_binder;

}

//发起端线程的事务栈 要等于 接收端的事务栈

if (target\_thread->transaction\_stack != in\_reply\_to) {

return\_error = BR\_FAILED\_REPLY;

in\_reply\_to = NULL;

target\_thread = NULL;

goto err\_dead\_binder;

}

target\_proc = target\_thread->proc; //发起端的进程

} else {

...

}

if (target\_thread) {

//发起端的线程

target\_list = &target\_thread->todo;

target\_wait = &target\_thread->wait;

} else {

...

}

t = kzalloc(sizeof(\*t), GFP\_KERNEL);

tcomplete = kzalloc(sizeof(\*tcomplete), GFP\_KERNEL);

...

if (!reply && !(tr->flags & TF\_ONE\_WAY))

t->from = thread;

else

t->from = NULL; //进入该分支

t->sender\_euid = task\_euid(proc->tsk);

t->to\_proc = target\_proc;

t->to\_thread = target\_thread;

t->code = tr->code;

t->flags = tr->flags;

t->priority = task\_nice(current);

// 发起端进程分配buffer

t->buffer = binder\_alloc\_buf(target\_proc, tr->data\_size,

tr->offsets\_size, !reply && (t->flags & TF\_ONE\_WAY));

...

t->buffer->allow\_user\_free = 0;

t->buffer->transaction = t;

t->buffer->target\_node = target\_node;

if (target\_node)

binder\_inc\_node(target\_node, 1, 0, NULL);

//分别拷贝用户空间的binder\_transaction\_data中ptr.buffer和ptr.offsets到内核

copy\_from\_user(t->buffer->data,

(const void \_\_user \*)(uintptr\_t)tr->data.ptr.buffer, tr->data\_size);

copy\_from\_user(offp,

(const void \_\_user \*)(uintptr\_t)tr->data.ptr.offsets, tr->offsets\_size);

...

if (reply) {

binder\_pop\_transaction(target\_thread, in\_reply\_to);

} else if (!(t->flags & TF\_ONE\_WAY)) {

...

} else {

...

}

//将BINDER\_WORK\_TRANSACTION添加到目标队列，本次通信的目标队列为target\_thread->todo

t->work.type = BINDER\_WORK\_TRANSACTION;

list\_add\_tail(&t->work.entry, target\_list);

//将BINDER\_WORK\_TRANSACTION\_COMPLETE添加到当前线程的todo队列

tcomplete->type = BINDER\_WORK\_TRANSACTION\_COMPLETE;

list\_add\_tail(&tcomplete->entry, &thread->todo);

//唤醒等待队列，本次通信的目标队列为target\_thread->wait

if (target\_wait)

wake\_up\_interruptible(target\_wait);

return;

binder\_transaction -> binder\_thread\_read -> IPC.waitForResponse，收到BR\_REPLY来回收buffer.

#### 5.3 BR\_REPLY

status\_t IPCThreadState::waitForResponse(Parcel \*reply, status\_t \*acquireResult)

{

int32\_t cmd;

int32\_t err;

while (1) {

if ((err=talkWithDriver()) < NO\_ERROR) break; // 【见小节2.11】

if (mIn.dataAvail() == 0) continue;

...

cmd = mIn.readInt32();

switch (cmd) {

...

case BR\_REPLY:

{

binder\_transaction\_data tr;

err = mIn.read(&tr, sizeof(tr));

if (err != NO\_ERROR) goto finish;

if (reply) {

...

} else {

// 释放buffer[见小节5.4]

freeBuffer(NULL,

reinterpret\_cast<const uint8\_t\*>(tr.data.ptr.buffer),

tr.data\_size,

reinterpret\_cast<const binder\_size\_t\*>(tr.data.ptr.offsets),

tr.offsets\_size/sizeof(binder\_size\_t), this);

continue;

}

}

goto finish;

default:

err = executeCommand(cmd);

...

break;

}

}

...

}

#### 5.4 IPC.freeBuffer

void IPCThreadState::freeBuffer(Parcel\* parcel, const uint8\_t\* data,

size\_t /\*dataSize\*/,

const binder\_size\_t\* /\*objects\*/,

size\_t /\*objectsSize\*/, void\* /\*cookie\*/)

{

if (parcel != NULL) parcel->closeFileDescriptors();

IPCThreadState\* state = self();

state->mOut.writeInt32(BC\_FREE\_BUFFER);

state->mOut.writePointer((uintptr\_t)data);

}

将BC\_FREE\_BUFFER写入mOut,再talkWithDriver()

##### 5.5 BC\_FREE\_BUFFER

static int binder\_thread\_write(struct binder\_proc \*proc,

struct binder\_thread \*thread,

binder\_uintptr\_t binder\_buffer, size\_t size,

binder\_size\_t \*consumed)

{

uint32\_t cmd;

void \_\_user \*buffer = (void \_\_user \*)(uintptr\_t)binder\_buffer;

void \_\_user \*ptr = buffer + \*consumed;

void \_\_user \*end = buffer + size;

while (ptr < end && thread->return\_error == BR\_OK) {

//拷贝用户空间的cmd命令，此时为BC\_FREE\_BUFFER

if (get\_user(cmd, (uint32\_t \_\_user \*)ptr)) -EFAULT;

ptr += sizeof(uint32\_t);

switch (cmd) {

case BC\_TRANSACTION:

case BC\_REPLY: ...

case BC\_FREE\_BUFFER: {

void \_\_user \*data\_ptr;

struct binder\_buffer \*buffer;

if (get\_user(data\_ptr, (void \* \_\_user \*)ptr)) return -EFAULT;

ptr += sizeof(void \*);

buffer = binder\_buffer\_lookup(proc, data\_ptr);

...

if (buffer->transaction) {

buffer->transaction->buffer = NULL;

buffer->transaction = NULL;

}

// binder\_buffer存在异步事务,且binder\_node不为空

if (buffer->async\_transaction && buffer->target\_node) {

if (list\_empty(&buffer->target\_node->async\_todo))

buffer->target\_node->has\_async\_transaction = 0;

else

//当异步队列async\_todo也不为空,则事务追加到该线程todo队列.

list\_move\_tail(buffer->target\_node->async\_todo.next, &thread->todo);

}

binder\_transaction\_buffer\_release(proc, buffer, NULL);

binder\_free\_buf(proc, buffer);

break;

}

}

\*consumed = ptr - buffer;

}

return 0;

}

接收端线程处理BC\_FREE\_BUFFER命令:

* 当binder\_buffer存在异步事务,当异步队列async\_todo也不为空,则事务追加到该线程todo队列.
* 释放当前的buffer.

##### 5.6 binder\_thread\_read

binder\_thread\_read（）{

...

while (1) {

uint32\_t cmd;

struct binder\_transaction\_data tr;

struct binder\_work \*w;

struct binder\_transaction \*t = NULL;

//从线程todo队列获取事务数据

if (!list\_empty(&thread->todo)) {

w = list\_first\_entry(&thread->todo, struct binder\_work, entry);

} else if (!list\_empty(&proc->todo) && wait\_for\_proc\_work) {

...

} else {

...

}

switch (w->type) {

case BINDER\_WORK\_TRANSACTION:

//获取transaction数据

t = container\_of(w, struct binder\_transaction, work);

break;

...

}

...

if (t->buffer->target\_node) {

//获取目标node

struct binder\_node \*target\_node = t->buffer->target\_node;

tr.target.ptr = target\_node->ptr;

tr.cookie = target\_node->cookie;

t->saved\_priority = task\_nice(current);

...

cmd = BR\_TRANSACTION; //设置命令为BR\_TRANSACTION

} else {

tr.target.ptr = NULL;

tr.cookie = NULL;

cmd = BR\_REPLY; //设置命令为BR\_REPLY

}

tr.code = t->code;

tr.flags = t->flags;

tr.sender\_euid = t->sender\_euid;

...

//将cmd和数据写回用户空间

if (put\_user(cmd, (uint32\_t \_\_user \*)ptr)) return -EFAULT;

ptr += sizeof(uint32\_t);

if (copy\_to\_user(ptr, &tr, sizeof(tr))) return -EFAULT;

ptr += sizeof(tr);

list\_del(&t->work.entry);

t->buffer->allow\_user\_free = 1;

if (cmd == BR\_TRANSACTION && !(t->flags & TF\_ONE\_WAY)) {

t->to\_parent = thread->transaction\_stack;

t->to\_thread = thread;

thread->transaction\_stack = t;

} else {

t->buffer->transaction = NULL;

kfree(t); //通信完成,则运行释放

}

break;

}

...

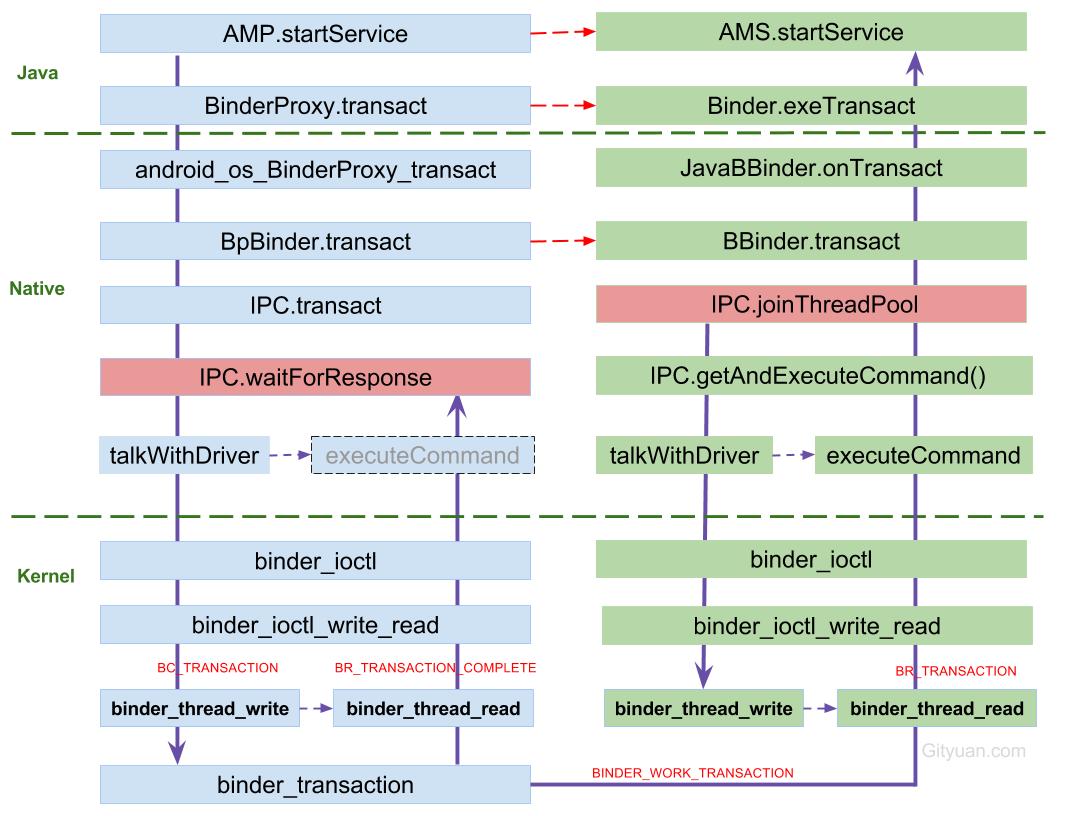
return 0;

}

## 六. 总结

本文详细地介绍如何从AMP.startService是如何通过Binder一步步调用进入到system\_server进程的AMS.startService. 整个过程涉及Java framework, native, kernel driver各个层面知识. 仅仅一个Binder IPC调用, 就花费了如此大篇幅来讲解, 可见系统之庞大. 整个过程的调用流程:

### 6.1 通信流程

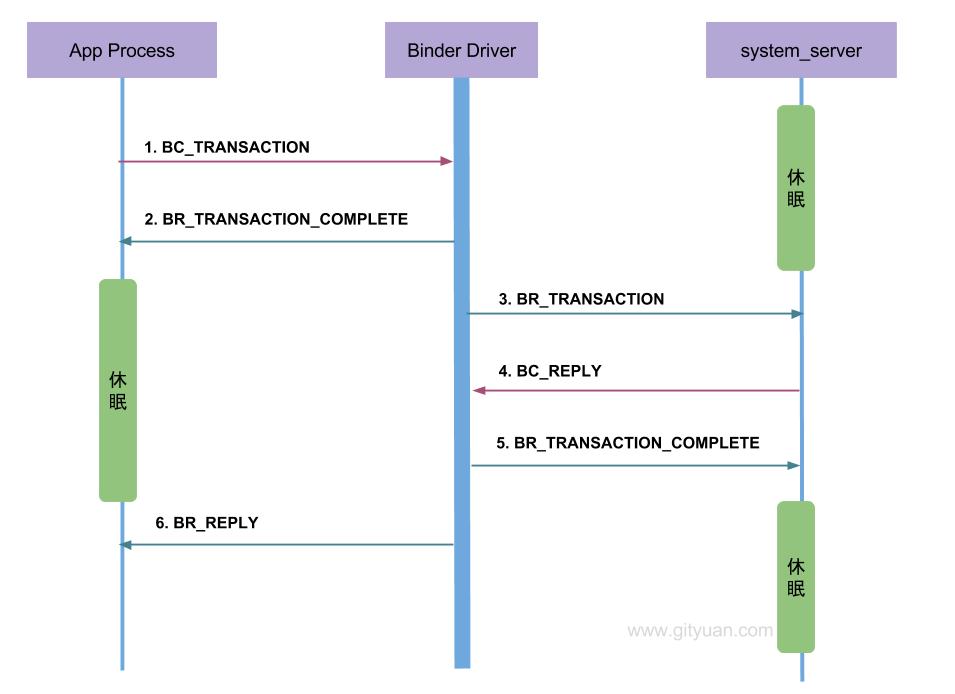
从通信流程角度来看整个过程:

图解:

1. 发起端线程向Binder Driver发起binder ioctl请求后, 便采用环不断talkWithDriver,此时该线程处于阻塞状态, 直到收到如下BR\_XXX命令才会结束该过程.
   * BR\_TRANSACTION\_COMPLETE: oneway模式下,收到该命令则退出
   * BR\_REPLY: 非oneway模式下,收到该命令才退出;
   * BR\_DEAD\_REPLY: 目标进程/线程/binder实体为空, 以及释放正在等待reply的binder thread或者binder buffer;
   * BR\_FAILED\_REPLY: 情况较多,比如非法handle, 错误事务栈, security, 内存不足, buffer不足, 数据拷贝失败, 节点创建失败, 各种不匹配等问题
   * BR\_ACQUIRE\_RESULT: 目前未使用的协议;
2. 左图中waitForResponse收到BR\_TRANSACTION\_COMPLETE,则直接退出循环, 则没有机会执行executeCommand()方法, 故将其颜色画为灰色. 除以上5种BR\_XXX命令, 当收到其他BR命令,则都会执行executeCommand过程.
3. 目标Binder线程创建后, 便进入joinThreadPool()方法, 采用循环不断地循环执行getAndExecuteCommand()方法, 当bwr的读写buffer都没有数据时,则阻塞在binder\_thread\_read的wait\_event过程. 另外,正常情况下binder线程一旦创建则不会退出.

### 6.2 通信协议

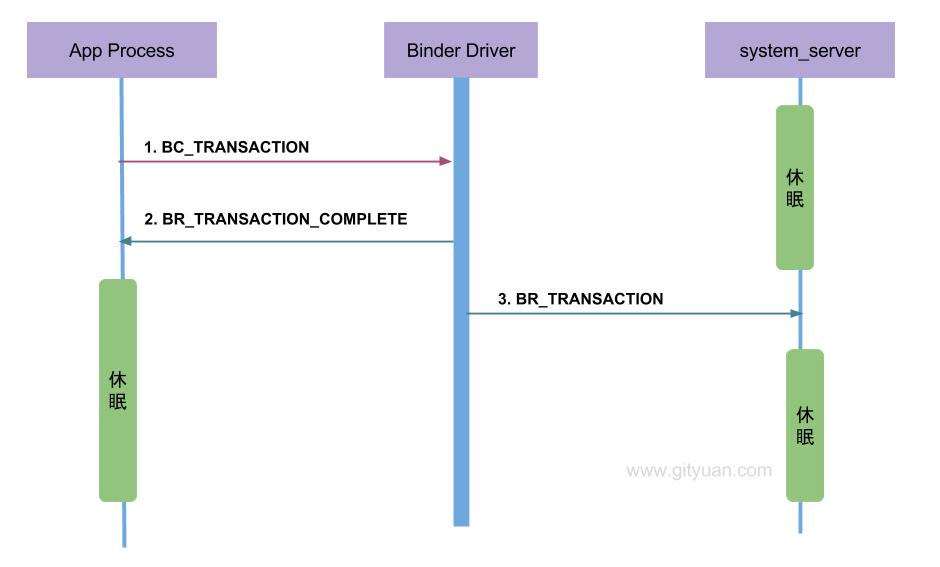
从通信协议的角度来看这个过程:



* Binder客户端或者服务端向Binder Driver发送的命令都是以BC\_开头,例如本文的BC\_TRANSACTION和BC\_REPLY, 所有Binder Driver向Binder客户端或者服务端发送的命令则都是以BR\_开头, 例如本文中的BR\_TRANSACTION和BR\_REPLY.
* 只有当BC\_TRANSACTION或者BC\_REPLY时, 才调用binder\_transaction()来处理事务. 并且都会回应调用者一个BINDER\_WORK\_TRANSACTION\_COMPLETE事务, 经过binder\_thread\_read()会转变成BR\_TRANSACTION\_COMPLETE.
* startService过程便是一个非oneway的过程, 那么oneway的通信过程如下所述.

### 6.3 说一说oneway

上图是非oneway通信过程的协议图, 下图则是对于oneway场景下的通信协议图:



当收到BR\_TRANSACTION\_COMPLETE则程序返回,有人可能觉得好奇,为何oneway怎么还要等待回应消息? 我举个例子,你就明白了.

你(app进程)要给远方的家人(system\_server进程)邮寄一封信(transaction), 你需要通过邮寄员(Binder Driver)来完成.整个过程如下:

1. 你把信交给邮寄员(BC\_TRANSACTION);
2. 邮寄员收到信后, 填一张单子给你作为一份回执(BR\_TRANSACTION\_COMPLETE). 这样你才放心知道邮递员已确定接收信, 否则就这样走了,信到底有没有交到邮递员手里都不知道,这样的通信实在太让人不省心, 长时间收不到远方家人的回信, 无法得知是在路的中途信件丢失呢,还是压根就没有交到邮递员的手里. 所以说oneway也得知道信是投递状态是否成功.
3. 邮递员利用交通工具(Binder Driver),将信交给了你的家人(BR\_TRANSACTION);

当你收到回执(BR\_TRANSACTION\_COMPLETE)时心里也不期待家人回信, 那么这便是一次oneway的通信过程.

如果你希望家人回信, 那便是非oneway的过程,在上述步骤2后并不是直接返回,而是继续等待着收到家人的回信, 经历前3个步骤之后继续执行:

1. 家人收到信后, 立马写了个回信交给邮递员BC\_REPLY;
2. 同样,邮递员要写一个回执(BR\_TRANSACTION\_COMPLETE)给你家人;
3. 邮递员再次利用交通工具(Binder Driver), 将回信成功交到你的手上(BR\_REPLY)

这便是一次完成的非oneway通信过程.

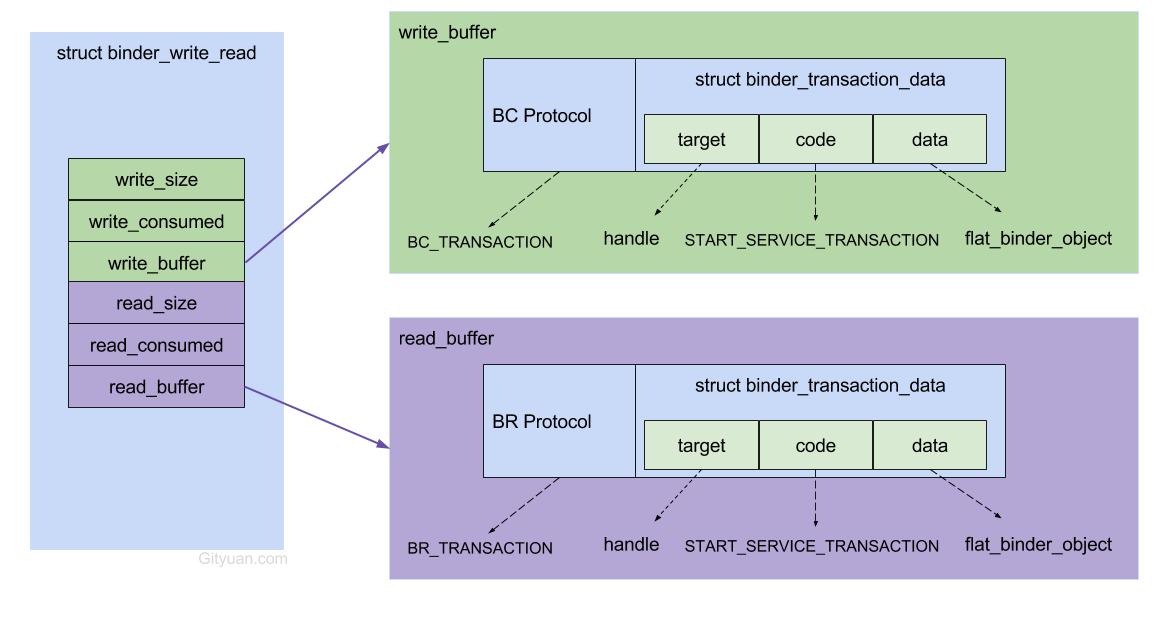
oneway与非oneway: 都是需要等待Binder Driver的回应消息BR\_TRANSACTION\_COMPLETE. 主要区别在于oneway的通信收到BR\_TRANSACTION\_COMPLETE则返回,而不会再等待BR\_REPLY消息的到来. 另外，oneway的binder IPC则接收端无法获取对方的pid.

### 6.4 小规律

* BC\_TRANSACTION + BC\_REPLY = BR\_TRANSACTION\_COMPLETE + BR\_DEAD\_REPLY + BR\_FAILED\_REPLY
* Binder线程只有当本线程的thread->todo队列为空，并且thread->transaction\_stack也为空，才会去处理当前进程的事务， 否则会继续处理或等待当前线程的todo队列事务。换句话说，就是只有当前线程的事务;
* binder\_thread\_write: 添加成员到todo队列;
* binder\_thread\_read: 消耗todo队列;
* 对于处于空闲可用的,或者Ready的binder线程是指停在binder\_thread\_read()的wait\_event地方的Binder线程;
* 每一次BR\_TRANSACTION或者BR\_REPLY结束之后都会调用freeBuffer().
* ProcessState.mHandleToObject记录着handle与对应的BpBinder信息。

整个过程copy once便是指binder\_transaction()过程把binder\_transaction\_data->data拷贝到目标进程的buffer。

### 6.5 数据流



* [2.1]AMP.startService：组装flat\_binder\_object对象等组成的Parcel data；
* [2.9]IPC.writeTransactionData：组装BC\_TRANSACTION和binder\_transaction\_data结构体，写入mOut;
* [2.11]IPC.talkWithDriver: 组装BINDER\_WRITE\_READ和binder\_write\_read结构体，通过ioctl传输到驱动层。

进入驱动后

* [3.3]binder\_thread\_write: 处理binder\_write\_read.write\_buffer数据
* [3.4]binder\_transaction: 处理write\_buffer.binder\_transaction\_data数据；
  + 创建binder\_transaction结构体，记录事务通信的线程来源以及事务链条等相关信息；
  + 分配binder\_buffer结构体，拷贝当前线程binder\_transaction\_data的data数据到binder\_buffer->data；
* [3.5]binder\_thread\_read: 处理binder\_transaction结构体数据
  + 组装cmd=BR\_TRANSACTION和binder\_transaction\_data结构体，写入binder\_write\_read.read\_buffer数据

回到用户空间

* [4.3]IPC.executeCommand：处理BR\_TRANSACTION命令, 将binder\_transaction\_data数据解析成BBinder.transact()所需的参数
* [4.7] AMN.onTransact: 层层回调，进入该方法，反序列化数据后，调用startService()方法。

## 參考

<https://gityuan.com/2016/09/04/binder-start-service/>

# binder调试分析

## Binder驱动调试

Binder IPC过程最终都交给Binder Driver来完成，这是真正干跨进程通信活的地方，那么意味着这里会有各种核心的通信log，比如binder open, mmap, ioctl等操作都可以通过某种方式来打开相应调试信息来分析。对于binder driver存在16类调试log开关，如下:

### debug\_mask

| **Log类型** | **mask值** | **解释** |
| --- | --- | --- |
| BINDER\_DEBUG\_USER\_ERROR | 1 | 用户使用错误 |
| BINDER\_DEBUG\_FAILED\_TRANSACTION | 2 | transaction失败 |
| BINDER\_DEBUG\_DEAD\_TRANSACTION | 4 | transaction死亡 |
| BINDER\_DEBUG\_OPEN\_CLOSE | 8 | **binder的open/close/mmap信息** |
| BINDER\_DEBUG\_DEAD\_BINDER | 16 | binder/node死亡信息 |
| BINDER\_DEBUG\_DEATH\_NOTIFICATION | 32 | binder死亡通知信息 |
| BINDER\_DEBUG\_READ\_WRITE | 64 | binder的read/write信息 |
| BINDER\_DEBUG\_USER\_REFS | 128 | binder引用计数 |
| BINDER\_DEBUG\_THREADS | 256 | **binder\_thread信息** |
| BINDER\_DEBUG\_TRANSACTION | 512 | transaction信息 |
| BINDER\_DEBUG\_TRANSACTION\_COMPLETE | 1024 | transaction完成信息 |
| BINDER\_DEBUG\_FREE\_BUFFER | 2048 | 可用buffer信息 |
| BINDER\_DEBUG\_INTERNAL\_REFS | 4096 | binder内部引用计数 |
| BINDER\_DEBUG\_BUFFER\_ALLOC | 8192 | **同步内存分配信息** |
| BINDER\_DEBUG\_PRIORITY\_CAP | 16384 | 调整binder线程的nice值 |
| BINDER\_DEBUG\_BUFFER\_ALLOC\_ASYNC | 32768 | **异步内存分配信息** |

每一项mask值通过将1左移N位，也就是等于2的倍数

### 调试开关

通过节点/sys/module/binder/parameters/debug\_mask来动态控制选择开启上表中的debug log.

(1)例如打开BINDER\_DEBUG\_OPEN\_CLOSE调试开关，则通过adb shell执行如下命令：

echo 8 > /sys/module/binder/parameters/debug\_mask

(2)再例如同时打开BINDER\_DEBUG\_FAILED\_TRANSACTION和BINDER\_DEBUG\_DEAD\_BINDER，将各个mask值相加即可，16+2 =18.

echo 18 > /sys/module/binder/parameters/debug\_mask

(3)要打开多个开关，只需将各个开关的mask值相加写入debug\_mask即可。打开调试开关后，可通过adb shell，执行cat /proc/kmsg | grep binder，即可查看相应的binder log信息。

### 原理

mask相加，其实现其实是利用或运算，通过一个变量控制16个开关，而不是采用16个变量，这是比较经典的设计方案。在binder Driver中通过下面语句完成节点控制debug的功能：

module\_param\_named(debug\_mask, binder\_debug\_mask, uint, S\_IWUSR | S\_IRUGO);

module\_param\_named的功能：

* 首先会生成/sys/module/binder/parameters/目录；
* module\_param\_named的第一个参数为debug\_mask，则会在parameters目录下创建debug\_mask文件节点；
* 当通过echo NUM > debug\_mask命令，会触发动态修改module\_param\_named的第二个参数binder\_debug\_mask值，这是个静态uint32\_t类型数据；
* 驱动中输出debug log都是通过调用binder\_debug()方法，该方法通过与binder\_debug\_mask变量做或运算来判断相应类型的log信息是否需要输出。

binder\_debug宏定义，如下：

#define binder\_debug(mask, x...) \

do { \

if (binder\_debug\_mask & mask) \

pr\_info(x); \

} while (0)

当然，也可以通过代码直接修改binder\_debug\_mask值来控制调试开关，默认值为：

binder\_debug\_mask = BINDER\_DEBUG\_USER\_ERROR |

BINDER\_DEBUG\_FAILED\_TRANSACTION | BINDER\_DEBUG\_DEAD\_TRANSACTION;

另外，在/sys/module/binder/parameters/目录还有另外两个节点，分别为proc\_no\_lock, stop\_on\_user\_error，其实现原理也基本差不多。

* **proc\_no\_lock节点**：与之对应binder驱动的binder\_debug\_no\_lock变量，这是bool类型变量，该开关含义为在输出某些统计调试方法中是否加锁；默认为N；
* **stop\_on\_user\_error节点**：与之对应binder驱动的binder\_stop\_on\_user\_error变量，这是int类型变量，另外，修改该节点还会触发调用binder\_set\_stop\_on\_user\_error()方法;该开关含义是指当触发BINDER\_DEBUG\_USER\_ERROR类型错误时是否让整个binder系统进入休眠等待状态，默认值为0，代表不会即便发生该类型错误系统不会被挂住，而是继续执行。

## 实战

### 3.1 BINDER\_DEBUG\_OPEN\_CLOSE

#### 3.1.1 log信息

当打开调试开关BINDER\_DEBUG\_OPEN\_CLOSE时，主要输出binder的open, mmap, close, flush, release方法中的log信息

具体kernel log，如下：

1. **binder\_open:** 4681:4681
2. **binder\_mmap:** 4681 b6b42000-b6c40000 (1016 K) vma 200071 pagep 79f
3. **binder:** 4681 close vm area b6b42000-b6c40000 (1016 K) vma 2220051 pagep 79f
4. **binder\_flush:** 4681 woke 0 threads
5. **binder\_release:** 4681 threads 1, nodes 0 (ref 0), refs 2, active transactions 0, buffers 1, pages 1

#### 3.1.2 解析

上面各行log所对应的信息项：

1. **binder\_open:** group\_leader->pid:pid
2. **binder\_mmap:** pid vm\_start-vm\_end (vm\_size K) vma vm\_flags pagep vm\_page\_prot
3. **binder:** pid close vm area vm\_start-vm\_end (vm\_size K) vma vm\_flags pagep vm\_page\_prot
4. **binder\_flush:** pid woke wake\_count threads
5. **binder\_release:** pid threads threads, nodes nodes (ref incoming\_refs), refs outgoing\_refs, active transactions active\_transactions, buffers buffers, pages page\_count

进一步说明其中部分关键词的含义：

* vm\_page\_prot:是指当前进程的VMA访问权限；
* wake\_count:是指该进程唤醒了处于BINDER\_LOOPER\_STATE\_WAITING休眠等待状态的线程个数；
* threads是指该进程中的线程个数；
* nodes代表该进程中创建binder\_node个数；
* incoming\_refs指向当前node的refs个数；
* outgoing\_refs指向其他进程的refs个数；
* active\_transactions是指当前进程中所有binder线程的transactions总和；
* buffers是指当前进程已分配的buffer个数；
* page\_count是指当前进程已分配的物理page个数。

#### 3.1.3 对应函数

上述log每一行相对应的函数：

1. binder\_open()
2. binder\_vma\_open() 或者 binder\_mmap()
3. binder\_vma\_close()
4. binder\_deferred\_flush() 由binder\_flush调用（见下方调用栈）
5. binder\_deferred\_release() 由binder\_release调用（见下方调用栈）

**binder\_flush调用栈：**

binder\_flush binder\_defer\_work(proc, BINDER\_DEFERRED\_FLUSH);

queue\_work(binder\_deferred\_workqueue, &binder\_deferred\_work);

binder\_deferred\_func //通过 DECLARE\_WORK(binder\_deferred\_work, binder\_deferred\_func);

binder\_deferred\_flush

**binder\_release调用栈：**

binder\_release binder\_defer\_work(proc, BINDER\_DEFERRED\_RELEASE);

queue\_work(binder\_deferred\_workqueue, &binder\_deferred\_work);

binder\_deferred\_func //通过 DECLARE\_WORK(binder\_deferred\_work, binder\_deferred\_func);

binder\_deferred\_release

当binder所在进程结束时会调用binder\_release。 binder\_open打开binder驱动/dev/binder，这是字符设备，获取文件描述符。在进程结束的时候会有一个关闭文件系统的过程中会调用驱动close方法，该方法相对应的是release()方法。

但并不是每个close系统调用都会触发调用release()方法. 只有真正释放设备数据结构才调用release(),内核维持一个文件结构被使用多少次的计数，即便是应用程序没有明显地关闭它打开的文件也适用: 内核在进程exit()时会释放所有内存和关闭相应的文件资源, 通过使用close系统调用最终也会release binder.

### 3.2 BINDER\_DEBUG\_DEAD\_BINDER

//debug\_id, node的引用次数，死亡通知个数   
binder: node 1078337 now dead, refs 1, death 0

//ref->proc->pid, ref->debug\_id, ref->desc(handle)   
binder: 13839 delete ref 1078335 desc 1 has death notification

//proc->pid, thread->pid, (u64)cookie, death   
binder: 1788:1805 BC\_DEAD\_BINDER\_DONE 9ce308c0 found f10a5400

### 3.3 BINDER\_DEBUG\_FREE\_BUFFER

**查询可用buffer：**

//proc->pid, thread->pid, (u64)data\_ptr, buffer->debug\_id, buffer->transaction   
binder: 463:5919 BC\_FREE\_BUFFER ub4641028 found buffer 1183795 for finished transaction   
binder: 277:2771 BC\_FREE\_BUFFER ub6c58028 found buffer 1183806 for active transaction

另外，buffer->transaction ? “active” : “finished”

位于方法binder\_thread\_write()

### 3.4 BINDER\_DEBUG\_BUFFER\_ALLOC\_ASYNC(异步)

**申请和释放异步buffer:**

//proc->pid, size, proc->free\_async\_space  
binder: 1788: binder\_alloc\_buf size 148 async free 520004  
binder: 1788: binder\_free\_buf size 148 async free 520192

**解析：**

* binder\_alloc\_buf：进程1788，申请148 Bytes，则该进程的可用异步空间大小520004 Bytes；
* binder\_free\_buf： 进程1788，释放148 Bytes，则该进程的可用异步空间大小520192 Bytes；

**内存大小计算：**

free\_async\_space = 520004 Bytes，再释放148 Bytes后，则可用大小应该是 520152 Bytes，这里却为520192 Bytes，这里多出来的40 Bytes是哪来得呢？这是因为binder\_free\_buf还会同时释放struct binder\_buffer，该结构体大小则为40 Bytes.

另外：buffer申请内存binder\_alloc\_buf和释放内存binder\_free\_buf，除了本身内存申请和释放，会同时伴随着binder\_buffer结构体的创建和释放，这便是每次操作40 Bytes差距所在。另外, 对于64位系统,binder\_buffer大小为80 Bytes.

**初始化值**

proc->free\_async\_space = proc->buffer\_size / 2 = (1M-8K)/2 = 520192 Bytes。当进程刚打开binder驱动，执行完binder\_mmap方法后，异步可用空间总大小为 520192 Bytes.

### 3.5 BINDER\_DEBUG\_BUFFER\_ALLOC(同步)

// 参数：proc->pid, size, buffer, buffer\_size   
binder: 1788: binder\_alloc\_buf size 76 got buffer c7800128 size 208   
binder: 1788: allocate pages c7801000-c7800000   
// 参数：proc->pid, new\_buffer\_size, new\_buffer   
binder: 1788: add free buffer, size 92, at c780019c   
binder: 1788: free pages c7801000-c7800000   
//参数：proc->pid, buffer, prev   
binder: 1788: merge free, buffer c780019c share page with c7800128

**解析：**

* binder\_alloc\_buf: 从proc->free\_buffers这棵红黑色树，找到一块大小大于并最接近76Bytes的buffer,该buffer大小为208Bytes;
* binder\_update\_page\_range：申请一个page大小的物理内存，地址为c7801000-c7800000。
* binder\_insert\_free\_buffer: 将空闲buffer添加到proc->free\_buffers；
* binder\_update\_page\_range：释放一个page大小的物理内存，地址为c7801000-c7800000。
* binder\_delete\_free\_buffer：在执行binder\_free\_buf()过程，合并释放的buffer，由于该buffer跟上一个buffer共享同一page，则无需释放。

## 四、 trace\_pipe

还有另一个方式也能调试Binder

cd /d/tracing/events/binder

进入该目录，会看到有很多binder相关的节点，需要通过开关来控制是否开启相应的调试开关。例如，要开启binder\_buffer内存分配过程的log：

echo 1 > /d/tracing/events/binder/binder\_transaction\_alloc\_buf/enable

查看信息：

cat /d/tracing/trace\_pipe

要关闭该信息：

echo 0 > /d/tracing/events/binder/binder\_transaction\_alloc\_buf/enable

当然也可以直接输出到文件：

adb shell cat /d/tracing/trace\_pipe > trace\_binder

这里可以开启的信息有很多，再比如：

echo 1 > /d/tracing/events/binder/enable

更多开关与功能，可自行探索。

## Binder子系统之调试分析(二)

<https://gityuan.com/2016/08/28/binder-debug-2/>

## Binder子系统之调试分析(三)

https://gityuan.com/2016/09/03/binder-debug-3/

## 參考

Binder子系统之调试分析(一)

<https://gityuan.com/2016/08/27/binder-debug/>

# 理解Refbase强弱引用

<http://gityuan.com/2015/12/05/android-refbase/>

# 10 binder与四大组件

<http://note.youdao.com/noteshare?id=4e5cd8395d29d2e8fc0dc4e026e5812a&sub=D711E7F7F24446A7867324E746876F08>

参考：http://blog.csdn.net/victory08/article/details/8696252

由于android系统中应用程序之间不能共享内存。因此，在不同应用程序之间交互数据（跨进程通讯）就稍微麻烦一些。在android SDK中提供了4种用于跨进程通讯的方式。这4种方式正好对应于android系统中4种应用程序组件：Activity、Content Provider、Broadcast和Service。其中Activity可以跨进程调用其他应用程序的Activity；Content Provider可以跨进程访问其他应用程序中的数据（以Cursor对象形式返回），当然，也可以对其他应用程序的数据进行增、删、改操 作；Broadcast可以向android系统中所有应用程序发送广播，而需要跨进程通讯的应用程序可以监听这些广播；Service和Content Provider类似，也可以访问其他应用程序中的数据，但不同的是，Content Provider返回的是Cursor对象，而Service返回的是Java对象，这种可以跨进程通讯的服务叫AIDL服务。

完整示例请参阅本文提供的源代码。

方式一：访问其他应用程序的Activity

Activity既可以在进程内（同一个应用程序）访问，也可以跨进程访问。如果想在同一个应用程序中访问Activity，需要指定Context对象和Activity的Class对象，代码如下：

[java] view plaincopy

Intent intent = new Intent(this, Test.class);

startActivity(intent);

Activity的跨进程访问与进程内访问略有不同。虽然它们都需要Intent对象，但跨进程访问并不需要指定Context对象和Activity的Class对象，而需要指定的是要访问的Activity所对应的Action（一个字符串）。有些Activity还需要指定一个Uri（通过Intent构造方法的第2个参数指定）。

在android系统中有很多应用程序提供了可以跨进程访问的Activity，例如，下面的代码可以直接调用拨打电话的Activity。

[java] view plaincopy

Intent callIntent = new Intent(Intent.ACTION\_CALL, Uri.parse("tel:12345678");

startActivity(callIntent);

执行上面的代码后，系统会自动拨号，界面如图1所示。

在调用拨号程序的代码中使用了一个Intent.ACTION\_CALL常量，该常量的定义如下：

[java] view plaincopy

public static final String ACTION\_CALL = "android.intent.action.CALL";

这个常量是一个字符串常量，也是我们在这节要介绍的跨进程调用Activity的关键。如果在应用程序中要共享某个Activity，需要为这个Activity指定一个字符串ID，也就是Action。也可以将这个Action看做这个Activity的key。在其他的应用程序中只要通过这个Action就可以找到与Action对应的Activity，并通过startActivity方法来启动这个Activity。

下面先来看一下如何将应用程序的Activity共享出来，读者可按如下几步来共享Activity：

1. 在AndroidManifest.xml文件中指定Action。指定Action要使用<action>标签，并在该标签的android:name属性中指定Action

2. 在AndroidManifest.xml文件中指定访问协议。在指定Uri（Intent类的第2个参数）时需要访问协议。访问协议需要使 用<data>标签的android:scheme属性来指定。如果该属性的值是“abc”，那么Uri就应该是“abc://Uri的主体 部分”，也就是说，访问协议是Uri的开头部分。

3. 通过getIntent().getData().getHost()方法获得协议后的Uri的主体部分。这个Host只是个称谓，并不一定是主机名。读者可以将其看成是任意的字符串。

4. 从Bundle对象中获得其他应用程序传递过来的数据。

5. 这一步当然是获得数据后做进一步的处理了。至于如何处理这些数据，就得根据具体的需求决定了。

下面来根据这些步骤共享一个Activity。首先建立一个android工程（ActionActivity），工程的主Activity是Main。在 本例中我们会共享这个Main类。首先打开AndroidManifest.xml文件，添加一个<activity>标签，并重新定义了Main的相应属性。AndroidManifest.xml文件的内容如下：

[java] view plaincopy

<!-- 重新配置Main -->

<activity android:name=".Main" android:label="@string/app\_name">

<intent-filter>

<action android:name="net.blogjava.mobile.MYACTION" />

<data android:scheme="info" />

<category android:name="android.intent.category.DEFAULT" />

</intent-filter>

</activity>

在配置AndroidManifest.xml时要注意，不能在同一个<activity>中配置多个动作，否则会覆盖MAIN动作以使该程序无法正常启动（虽然其他应用程序调用Main是正常的）。

从上面的代码可以看出，<action>标签的android:name属性值是net.blogjava.mobile.MYACTION，这就是Main自定义的动作。<data>标签指定了Url的协议。如果指定 了<data>标签的android:scheme属性值（info），则在调用Main时需要使用如下的URL:

[java] view plaincopy

info://任意字符串

一般<category>标签的android:name属性值可以设成android.intent.category.DEFAULT。

下面来看看如何在Main类的onCreate方法中获得其他应用程序传递过来的数据。

package net.blogjava.mobile.actionactivity;

... ...

public class Main extends Activity implements OnClickListener

{

private EditText editText;

@Override

public void onClick(View view)

{

// 单击按钮，会显示文本框中的内容（以Toast信息框形式显示）

Toast.makeText(this , editText.getText().toString(), Toast.LENGTH\_LONG)

.show();

}

@Override

public void onCreate(Bundle savedInstanceState)

{

super .onCreate(savedInstanceState);

setContentView(R.layout.main);

Button button = (Button) findViewById(R.id.button);

button.setOnClickListener(this );

editText = (EditText) findViewById(R.id.edittext);

// 获得其他应用程序传递过来的数据

if (getIntent().getData() != null )

{

// 获得Host，也就是info://后面的内容

String host = getIntent().getData().getHost();

Bundle bundle = getIntent().getExtras();

// 其他的应用程序会传递过来一个value值，在该应用程序中需要获得这个值

String value = bundle.getString("value" );

// 将Host和Value组合在一下显示在EditText组件中

editText.setText(host + ":" + value);

// 调用了按钮的单击事件，显示Toast信息提示框

onClick(button);

}

}

}

[java] view plaincopy

package net.blogjava.mobile.actionactivity;

... ...

public class Main extends Activity implements OnClickListener

{

private EditText editText;

@Override

public void onClick(View view)

{

// 单击按钮，会显示文本框中的内容（以Toast信息框形式显示）

Toast.makeText(this, editText.getText().toString(), Toast.LENGTH\_LONG)

.show();

}

@Override

public void onCreate(Bundle savedInstanceState)

{

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

Button button = (Button) findViewById(R.id.button);

button.setOnClickListener(this);

editText = (EditText) findViewById(R.id.edittext);

// 获得其他应用程序传递过来的数据

if (getIntent().getData() != null)

{

// 获得Host，也就是info://后面的内容

String host = getIntent().getData().getHost();

Bundle bundle = getIntent().getExtras();

// 其他的应用程序会传递过来一个value值，在该应用程序中需要获得这个值

String value = bundle.getString("value");

// 将Host和Value组合在一下显示在EditText组件中

editText.setText(host + ":" + value);

// 调用了按钮的单击事件，显示Toast信息提示框

onClick(button);

}

}

}

从上面的程序可以看出，首先通过getIntent().getData()来判断其他的应用程序是否传递了Uri（getData方法返回了一个Uri 对象）。如果运行该程序，Uri为null，因此，不会执行if语句里面的代码。当其他的应用程序传递了Uri对象后，系统会执行if语句里面的代码。当 运行ActionActivity后，在文本框中输入“Running”，单击“显示文本框的内容”按钮，会显示如图2所示的Toast提示信息框。

下面来看一下其他的应用程序是如何调用ActionActivity中的Main。新建一个android工程（InvokeActivity），并添加一个按钮，按钮的单击事件方法代码如下：

public void onClick(View view)

{

// 需要使用Intent类的第2个参数指定Uri

Intent intent = new Intent("net.blogjava.mobile.MYACTION" , Uri

.parse("info://调用其他应用程序的Activity" ));

// 设置value属性值

intent.putExtra("value" , "调用成功" );

// 调用ActionActivity中的Main

startActivity(intent);

}

[java] view plaincopy

public void onClick(View view)

{

// 需要使用Intent类的第2个参数指定Uri

Intent intent = new Intent("net.blogjava.mobile.MYACTION", Uri

.parse("info://调用其他应用程序的Activity"));

// 设置value属性值

intent.putExtra("value", "调用成功");

// 调用ActionActivity中的Main

startActivity(intent);

}

在运行InvokeActivity之前，先要运行ActionActivity以便在android模拟器中安装该程序。然后单击InvokeActivity中的按钮，就会显示如图3所示的效果。

当然，也可以使用startActivityForResult方法来启动其他应用程序的Activity，以便获得Activity的返回值。例如，可以将ActionActivity中Main类的onClick代码修改为下面的形式。

public void onClick(View view)

{

Toast.makeText(this , editText.getText().toString(), Toast.LENGTH\_LONG).show();

Intent intent = new Intent();

// 设置要返回的属性值

intent.putExtra("result" , editText.getText().toString());

// 设置返回码和Intent对象

setResult(2 , intent);

// 关闭Activity

finish();

}

[java] view plaincopy

public void onClick(View view)

{

Toast.makeText(this, editText.getText().toString(), Toast.LENGTH\_LONG).show();

Intent intent = new Intent();

// 设置要返回的属性值

intent.putExtra("result", editText.getText().toString());

// 设置返回码和Intent对象

setResult(2, intent);

// 关闭Activity

finish();

}

然后在InvokeActivity中使用下面的代码来调用Main。

intent = new Intent("net.blogjava.mobile.MYACTION" , Uri

.parse("info://调用其他应用程序的Activity" ));

// 传递数据

intent.putExtra("value" , "调用成功" );

startActivityForResult(intent, 1 ); // 1为请求码

[java] view plaincopy

intent = new Intent("net.blogjava.mobile.MYACTION", Uri

.parse("info://调用其他应用程序的Activity"));

// 传递数据

intent.putExtra("value", "调用成功");

startActivityForResult(intent, 1); // 1为请求码

要想接收Activity返回的值，需要覆盖onActivityResult事件方法，代码如下：

@Override

protected void onActivityResult(int requestCode, int resultCode, Intent data)

{

Toast.makeText(this , "返回值：" + data.getExtras().getString("result" ),

Toast.LENGTH\_LONG).show();

}

[java] view plaincopy

@Override

protected void onActivityResult(int requestCode, int resultCode, Intent data)

{

Toast.makeText(this, "返回值：" + data.getExtras().getString("result"),

Toast.LENGTH\_LONG).show();

}

当单击InvokeActivity中的相应按钮后，并且Main关闭后，会显示如图4所示的Toast信息提示框。

从本节的介绍可以看出，跨进程访问Activity（访问其他应用程序中的Activity）主要是通过一个Action来完成的，如果要传递数据，还需 要指定一个Uri。当然，传递数据也可以通过Intent来完成。传递数据的过程可以是双向的。如果要想从调用的Activity中返回数据，就需要使用startActivityForResult方法来启动Activity了。

## 方式二：Content Provider

Android应用程序可以使用文件或SqlLite数据库来存储数据。Content Provider提供了一种在多个应用程序之间数据共享的方式（跨进程共享数据）。应用程序可以利用Content Provider完成下面的工作

1. 查询数据

2. 修改数据

3. 添加数据

4. 删除数据

虽然Content Provider也可以在同一个应用程序中被访问，但这么做并没有什么意义。Content Provider存在的目的向其他应用程序共享数据和允许其他应用程序对数据进行增、删、改操作。

Android系统本身提供了很多Content Provider，例如，音频、视频、联系人信息等等。我们可以通过这些Content Provider获得相关信息的列表。这些列表数据将以Cursor对象返回。因此，从Content Provider返回的数据是二维表的形式。

对于访问Content Provider的程序，需要使用ContentResolver对象。该对象需要使用getContentResolver方法获得，

[java] view plaincopy

ContentResolver cr = getContentResolver();

与Activity一样，Content Provider也需要与一个URI对应。每一个Content Provider可以控制多个数据集，在这种情况下，每一个数据集会对应一个单独的URI。所有的URI必须以“content://”开头。

为了程序更容易维护，也为了简化程序代码，一般将URI定义成一个常量。例如，下面的常量表示系统的联系人电话号码。

[java] view plaincopy

android.provider.Contacts.Phones.CONTENT\_URI

下面来看一下编写Content Provider的具体步骤。

1. 编写一个继承于android.content.ContentProvider的子类。该类是ContentProvider的核心类。在该类中会实现query、insert、update及delete方法。实际上调用ContentResolver类的这4个方法就是调用ContentProvider类中与之要对应的方法。在本文中只介绍query。至于insert、update、delete和query的用法类 似。也是通过Uri传递参数，然后在这些方法中接收这些参数，并做进一步地处理。

2. 在AndroidManifest.xml文件中配置ContentProvider。要想唯一确定一个ContentProvider，需要指定这个ContentProvider的URI，除此之外，还需要指定URI所对应的ContentProvider类。这有些象Servlet的定义，除了要 指定Servlet对应的Web地址，还要指定这个地址所对应的Servlet类。

现在来看一下Uri的具体格式，先看一下如图5所示的URI。

下面对图5所示的URI的4个部分做一下解释。

A：Content Provider URI的固定前缀，也就是说，所有的URI必须以content://开头。

B：URI中最重要的部分。该部分是Content Provider的唯一标识。对于第三方应用程序来说，该部分最后使用完整的类名（包名+类名），以确保URI的唯一性。该部分需要在AndroidManifest.xml文件中<provider>标签中定义，代码如下：

<provider name=".TransportationProvider" authorities="com.example.transportationprovider"

. . . >

[java] view plaincopy

<provider name=".TransportationProvider" authorities="com.example.transportationprovider"

. . . >

C：这部分是URI的路径（path）。表示URI中各种被请求的数据。这部分是可选的， 如果Content Provider仅仅提供一种请求的数据，那么这部分可以省略。如果Content Provider要提供多种请求数据。就需要添加多个路径，甚至是子路径。例如，“land/bus”、“land/train”、“sea/ship” 就指定了3种可能提供的数据。

D：这部分也是可选的。如果要传递一个值给Content Provider，可以通过这部分传递。当然，如果不需要传值，这部分也可以省略，省略后的URI如下所示：

content://com.example.transportationprovider/trains

[java] view plaincopy

content://com.example.transportationprovider/trains

本例利用了《基于android SDK1.5的英文电子词典的实现》一文中实现的电子词典程序。通过ContentProvider，将电子词典的查词功能共享成Cursor对象。这样 其他的应用程序就可以通过ContentProvider来查词英文单词了。关于英文词典的具体实现细节，读者可以通过如下的地址查看《基于android SDK1.5的英文电子词典的实现》一文。

http://www. androidsdn.com/article/show/111

[java] view plaincopy

http://www.ophonesdn.com/article/show/111

在电子词典程序中需要一个DictionaryContentProvider类，该类是ContentProvider的子类。在该类中实现了query方法，并根据不同的URI来返回不同的结果。让我们先看一下DictionaryContentProvider类，然后再对这些代码做一些解 释。

... ...

public class DictionaryContentProvider extends ContentProvider

{

private static UriMatcher uriMatcher;

private static final String AUTHORITY = "net.blogjava.mobile.dictionarycontentprovider" ;

private static final int SINGLE\_WORD = 1 ;

private static final int PREFIX\_WORDS = 2 ;

public static final String DATABASE\_PATH = android.os.Environment

.getExternalStorageDirectory().getAbsolutePath()

+ "/dictionary" ;

public static final String DATABASE\_FILENAME = "dictionary.db" ;

private SQLiteDatabase database;

static

{

// 添加访问ContentProvider的Uri

uriMatcher = new UriMatcher(UriMatcher.NO\_MATCH);

uriMatcher.addURI(AUTHORITY, "single" , SINGLE\_WORD);

uriMatcher.addURI(AUTHORITY, "prefix/\*" , PREFIX\_WORDS);

}

// 该方法在Activity的onCreate方法之前调用

@Override

public boolean onCreate()

{

database = openDatabase();

return true ;

}

// 在本例中只实现了query方法，其他的方法（insert、update和delete）与query方法的实现

// 类似

@Override

public Cursor query(Uri uri, String[] projection, String selection,

String[] selectionArgs, String sortOrder)

{

Cursor cursor = null ;

switch (uriMatcher.match(uri))

{

case SINGLE\_WORD:

// 查找指定的单词

cursor = database.query("t\_words" , projection, selection,

selectionArgs, null , null , sortOrder);

break ;

case PREFIX\_WORDS:

String word = uri.getPathSegments().get(1 );

// 查找以指定字符串开头的单词集合

cursor = database

.rawQuery(

"select english as \_id, chinese from t\_words where english like ?" ,

new String[]

{ word + "%" });

break ;

default :

throw new IllegalArgumentException("<" + uri + ">格式不正确." );

}

return cursor;

}

... ...

}

[java] view plaincopy

... ...

public class DictionaryContentProvider extends ContentProvider

{

private static UriMatcher uriMatcher;

private static final String AUTHORITY = "net.blogjava.mobile.dictionarycontentprovider";

private static final int SINGLE\_WORD = 1;

private static final int PREFIX\_WORDS = 2;

public static final String DATABASE\_PATH = android.os.Environment

.getExternalStorageDirectory().getAbsolutePath()

+ "/dictionary";

public static final String DATABASE\_FILENAME = "dictionary.db";

private SQLiteDatabase database;

static

{

// 添加访问ContentProvider的Uri

uriMatcher = new UriMatcher(UriMatcher.NO\_MATCH);

uriMatcher.addURI(AUTHORITY, "single", SINGLE\_WORD);

uriMatcher.addURI(AUTHORITY, "prefix/\*", PREFIX\_WORDS);

}

// 该方法在Activity的onCreate方法之前调用

@Override

public boolean onCreate()

{

database = openDatabase();

return true;

}

// 在本例中只实现了query方法，其他的方法（insert、update和delete）与query方法的实现

// 类似

@Override

public Cursor query(Uri uri, String[] projection, String selection,

String[] selectionArgs, String sortOrder)

{

Cursor cursor = null;

switch (uriMatcher.match(uri))

{

case SINGLE\_WORD:

// 查找指定的单词

cursor = database.query("t\_words", projection, selection,

selectionArgs, null, null, sortOrder);

break;

case PREFIX\_WORDS:

String word = uri.getPathSegments().get(1);

// 查找以指定字符串开头的单词集合

cursor = database

.rawQuery(

"select english as \_id, chinese from t\_words where english like ?",

new String[]

{ word + "%" });

break;

default:

throw new IllegalArgumentException("<" + uri + ">格式不正确.");

}

return cursor;

}

... ...

}

关于DictionaryContentProvider类的代码需要做如下的解释。

1. 在DictionaryContentProvider类的开头定义的AUTHORITY是访问ContentProvider的URI的前半部分。

2. 访问ContentProvider的URI的后半部分由uriMatcher.addURI(...)方法指定。该方法的第1个参数就是AUTHORITY（Uri的前半部分），第2个参数是Uri的后半部分，第3个参数是与第2个参数值对应的代码。当其他的应用程序通过Uri访问ContentProvider时。系统解析Uri后，将addURI方法的第2个参数值转换成与之对应的代码（第3个参数值）。

3. addURI的第2个参数值可以使用通配符。例如，prefix/\*中的\*表示所有字符。prefix/abc、prefix/xxx都会匹配成功。

4. 访问ContentProvider的URI是addURI的第1个和第2个参数值的组件，例如，按着DictionaryContentProvider中设置的两个URI，可以分别匹配下面的两个URI。

content://net.blogjava.mobile.dictionarycontentprovider/single

content://net.blogjava.mobile.dictionarycontentprovider/prefix/wo

[java] view plaincopy

content://net.blogjava.mobile.dictionarycontentprovider/single

content://net.blogjava.mobile.dictionarycontentprovider/prefix/wo

要注意的是，访问ContentProvider的URI必须以“content://”开头。

5. 在query方法中建议使用SQLiteDatabase对象的query方法查询。因为query方法的参数正好和DictionaryContentProvider类中的query方法的参数对应，这样使用起来比较方便。

6. 由于安装了ContentProvider的应用程序会先调用ContentProvider的onCreate方法（该方法会在Activity的onCreate方法之前调用），因此，只需要将打开或复制数据库的方法（openDatabase）放在DictionaryContentProvider类中，并在onCreate方法中调用即可。

7. 在DictionaryContentProvider类中只实现了query方法。在该方法中判断了其他应用程序发送的是哪一个Uri。并进行相应的处理。这两个Uri一个是查询指定单词的，另外一个是查询以某个字符串开头的所有单词的（用于显示单词列表）。

下面在AndroidManifest.xml文件中配置DictionaryContentProvider类。

<provider android:name="DictionaryContentProvider"

android:authorities="net.blogjava.mobile.dictionarycontentprovider" />

[java] view plaincopy

<provider android:name="DictionaryContentProvider"

android:authorities="net.blogjava.mobile.dictionarycontentprovider" />

OK，现在来看看应用程序如何调用ContentProvider。调用ContentProvider的关键是使用getContentResolver方法来获得一个ContentResolver对象，并通过ContentResolver对象的query方法来 访问ContentProvider。

首先来定义两个访问ContentProvider的常量。

public final String DICTIONARY\_SINGLE\_WORD\_URI = "content://net.blogjava.mobile.dictionarycontentprovider/single" ;

public final String DICTIONARY\_PREFIX\_WORD\_URI = "content://net.blogjava.mobile.dictionarycontentprovider/prefix" ;

[java] view plaincopy

public final String DICTIONARY\_SINGLE\_WORD\_URI = "content://net.blogjava.mobile.dictionarycontentprovider/single";

public final String DICTIONARY\_PREFIX\_WORD\_URI = "content://net.blogjava.mobile.dictionarycontentprovider/prefix";

然后在查询按钮的单击事件中编写如下的代码来查询单词。

public void onClick(View view)

{

Uri uri = Uri.parse(DICTIONARY\_SINGLE\_WORD\_URI);

// 通过ContentProvider查询单词，并返回Cursor对象，然后的操作就和直接从数据中获得

// Cursor对象后的操作是一样的了

Cursor cursor = getContentResolver().query(uri, null , "english=?" ,

new String[]{ actvWord.getText().toString() }, null );

String result = "未找到该单词." ;

if (cursor.getCount() > 0 )

{

cursor.moveToFirst();

result = cursor.getString(cursor.getColumnIndex("chinese" ));

}

new AlertDialog.Builder(this ).setTitle("查询结果" ).setMessage(result)

.setPositiveButton("关闭" , null ).show();

}

[java] view plaincopy

public void onClick(View view)

{

Uri uri = Uri.parse(DICTIONARY\_SINGLE\_WORD\_URI);

// 通过ContentProvider查询单词，并返回Cursor对象，然后的操作就和直接从数据中获得

// Cursor对象后的操作是一样的了

Cursor cursor = getContentResolver().query(uri, null, "english=?",

new String[]{ actvWord.getText().toString() }, null);

String result = "未找到该单词.";

if (cursor.getCount() > 0)

{

cursor.moveToFirst();

result = cursor.getString(cursor.getColumnIndex("chinese"));

}

new AlertDialog.Builder(this).setTitle("查询结果").setMessage(result)

.setPositiveButton("关闭", null).show();

}

下面是显示单词列表的代码。

public void afterTextChanged(Editable s)

{

if ("" .equals(s.toString()))

return ;

Uri uri = Uri.parse(DICTIONARY\_PREFIX\_WORD\_URI + "/" + s.toString());

// 从ContentProvider中获得以某个字符串开头的所有单词的Cursor对象

Cursor cursor = getContentResolver().query(uri, null , null , null , null );

DictionaryAdapter dictionaryAdapter = new DictionaryAdapter(this ,

cursor, true );

actvWord.setAdapter(dictionaryAdapter);

}

[java] view plaincopy

public void afterTextChanged(Editable s)

{

if ("".equals(s.toString()))

return;

Uri uri = Uri.parse(DICTIONARY\_PREFIX\_WORD\_URI + "/" + s.toString());

// 从ContentProvider中获得以某个字符串开头的所有单词的Cursor对象

Cursor cursor = getContentResolver().query(uri, null, null, null, null);

DictionaryAdapter dictionaryAdapter = new DictionaryAdapter(this,

cursor, true);

actvWord.setAdapter(dictionaryAdapter);

}

现在来运行本例，会看到如图6所示的界面。当查询单词时会显示如图7所示的单词列表，查询出结果后，会显示如图8所示的界面。

## 方式三：广播（Broadcast）

广播是一种被动跨进程通讯的方式。当某个程序向系统发送广播时，其他的应用程序只能被动地接收广播数据。这就象电台进行广播一样，听众只能被动地收听，而不能主动与电台进行沟通。

在应用程序中发送广播比较简单。只需要调用sendBroadcast方法即可。该方法需要一个Intent对象。通过Intent对象可以发送需要广播的数据。

先建一个android工程：sendbroadcast。在XML布局文件中放两个组件：EditText和Button，当单击按钮后，会弹出显示EditText组件中文本的对话框，关闭对话框后， 会使用sendBroadcast方法发送消息，并将EditText组件的文本通过Intent对象发送出去。完整的代码如下：

package net.blogjava.mobile.sendbroadcast;

... ...

public class Main extends Activity implements OnClickListener

{

private EditText editText;

@Override

public void onClick(View view)

{

new AlertDialog.Builder(this ).setMessage(editText.getText().toString())

.setPositiveButton("确定" , null ).show();

// 通过Intent类的构造方法指定广播的ID

Intent intent = new Intent("net.blogjava.mobile.MYBROADCAST" );

// 将要广播的数据添加到Intent对象中

intent.putExtra("text" , editText.getText().toString());

// 发送广播

sendBroadcast(intent);

}

... ...

}

[java] view plaincopy

package net.blogjava.mobile.sendbroadcast;

... ...

public class Main extends Activity implements OnClickListener

{

private EditText editText;

@Override

public void onClick(View view)

{

new AlertDialog.Builder(this).setMessage(editText.getText().toString())

.setPositiveButton("确定", null).show();

// 通过Intent类的构造方法指定广播的ID

Intent intent = new Intent("net.blogjava.mobile.MYBROADCAST");

// 将要广播的数据添加到Intent对象中

intent.putExtra("text", editText.getText().toString());

// 发送广播

sendBroadcast(intent);

}

... ...

}

发送广播并不需要在AndroidManifest.xml文件中注册，但接收广播必须在AndroidManifest.xml文件中注册receiver。下面来编写一个接收广播的应用程序。首先建立一个android工程：receiver。然后编写一个MyReceiver类，该类是BroadcastReceiver的子类，代码如下：

package net.blogjava.mobile.receiver;

... ...

public class MyReceiver extends BroadcastReceiver

{

// 当sendbroadcast发送广播时，系统会调用onReceive方法来接收广播

@Override

public void onReceive(Context context, Intent intent)

{

// 判断是否为sendbroadcast发送的广播

if ("net.blogjava.mobile.MYBROADCAST" .equals(intent.getAction()))

{

Bundle bundle = intent.getExtras();

if (bundle != null )

{

String text = bundle.getString("text" );

Toast.makeText(context, "成功接收广播：" + text, Toast.LENGTH\_LONG).show();

}

}

}

}

[java] view plaincopy

package net.blogjava.mobile.receiver;

... ...

public class MyReceiver extends BroadcastReceiver

{

// 当sendbroadcast发送广播时，系统会调用onReceive方法来接收广播

@Override

public void onReceive(Context context, Intent intent)

{

// 判断是否为sendbroadcast发送的广播

if ("net.blogjava.mobile.MYBROADCAST".equals(intent.getAction()))

{

Bundle bundle = intent.getExtras();

if (bundle != null)

{

String text = bundle.getString("text");

Toast.makeText(context, "成功接收广播：" + text, Toast.LENGTH\_LONG).show();

}

}

}

}

当应用程序发送广播时，系统会调用onReceive方法来接收广播，并通过intent.getAction()方法返回广播的ID，也就是在发送广播时Intent构造方法指定的字符串。然后就可以从Bundle对象中获得相应的数据了。

最后还需要在AndroidManifest.xml文件中注册receiver，代码如下：

<!-- 注册receiver 

<receiver android:name="MyReceiver" >

<intent-filter>

<action android:name="net.blogjava.mobile.MYBROADCAST" />

</intent-filter>

</receiver>

[java] view plaincopy

<!-- 注册receiver 

<receiver android:name="MyReceiver">

<intent-filter>

<action android:name="net.blogjava.mobile.MYBROADCAST" />

</intent-filter>

</receiver>

在注册MyReceiver类时需要使用<receiver>标签，android:name属性指定MyReceiver类，<action>标签的android:name指定了广播的ID。

首先运行receiver程序，然后就可以关闭receiver程序了。接收广播并不依赖于程序的状态。就算程序关闭了，仍然可以接收广播。然后再启动sendbroadcast程序。并在文本框中输入“android”，然后单击按钮，会弹出一个显示文本框内容的对话框，如图9所示。当关闭对话框后，会 显示一个Toast信息提示框，这个信息框是由receiver程序弹出的。如图10所示。

方式四：AIDL服务

服务（Service）是android系统中非常重要的组件。Service可以脱离应用程序运行。也就是说，应用程序只起到一个启动Service的作用。一但Service被启动，就算应用程序关闭，Service仍然会在后台运行。

android系统中的Service主要有两个作用：后台运行和跨进程通讯。后台运行就不用说了，当Service启动后，就可以在Service对象中 运行相应的业务代码，而这一切用户并不会察觉。而跨进程通讯是这一节的主题。如果想让应用程序可以跨进程通讯，就要使用我们这节讲的AIDL服 务，AIDL的全称是Android Interface Definition Language，也就是说，AIDL实际上是一种接口定义语言。通过这种语言定义接口后，Eclipse插件（ODT）会自动生成相应的Java代码接 口代码。下面来看一下编写一个AIDL服务的基本步骤。

1. 在Eclipse工程的package目录中建立一个扩展名为aidl的文件。package目录就是Java类所在的目录。该文件的语法类似于Java代码。aidl文件中定义的是AIDL服务的接口。这个接口需要在调用AIDL服务的程序中访问。

2. 如果aidl文件的内容是正确的，Eclipse插件会自动生成一个Java接口文件（\*.java）。

3. 建立一个服务类（Service的子类）。

4. 实现由aidl文件生成的Java接口。

5. 在AndroidManifest.xml文件中配置AIDL服务，尤其要注意的是，<action>标签的android:name属性值就是客户端要引用该服务的ID，也就是Intent类构造方法的参数值。

现在我们来编写一个AIDL服务，首先建立一个android工程：aidlservice。在aidlservice工程中有一个Main类，在Main类所有的目录建立一个IMyService.aidl文件，内容如下：

[java] view plaincopy

package net.blogjava.mobile.aidlservice;

interface IMyService

{

String getValue(); // 为AIDL服务的接口方法，调用AIDL服务的程序需要调用该方法

}

在保存IMyService.aidl文件后，ODT会在gen目录下产生一个IMyService.java文件，读者可以不必管这个文件中的内容，也 不需要修改该文件的内容。这个文件是由ODT自动维护的，只要修改了IMyService.aidl文件的内容，IMyService.java文件的内 容就会随之改变。

然后建立一个MyService类，该类是Service的子类，代码如下：

package net.blogjava.mobile.aidlservice;

public class MyService extends Service

{

// IMyService.Stub类是根据IMyService.aidl文件生成的类，该类中包含了接口方法（getValue）

public class MyServiceImpl extends IMyService.Stub

{

@Override

public String getValue() throws RemoteException

{

return "从AIDL服务获得的值.";

}

}

@Override

public IBinder onBind(Intent intent)

{

// 该方法必须返回MyServiceImpl类的对象实例

return new MyServiceImpl();

}

}

最后需要在AndroidManifest.xml文件中配置MyService类，代码如下：

[java] view plaincopy

<!-- 注册服务-->

<service android:name=".MyService">

<intent-filter>

<!-- 指定调用AIDL服务的ID -->

<action android:name="net.blogjava.mobile.aidlservice.IMyService" />

</intent-filter>

</service>

下面来看看如何调用这个AIDL服务。首先建立一个android工程：aidlclient。然后将aidlservice工程中自动生成的IMyService.java文件复制到aidlclient工程中。在调用AIDL服务之前需要先使用bindService方法绑定AIDL服务。bindService方法需要一个ServiceConnection对象。ServiceConnection有一个onServiceConnected方法，当成功绑定AIDL服务且，该方法被调用。并通过service参数返回AIDL服务对象。下面是调用AIDL服务的完成代码。

[java] view plaincopy

package net.blogjava.mobile.aidlclient;

public class Main extends Activity implements OnClickListener

{

private IMyService myService = null;

// 创建ServiceConnection对象

private ServiceConnection serviceConnection = new ServiceConnection()

{

@Override

public void onServiceConnected(ComponentName name, IBinder service)

{

// 获得AIDL服务对象

myService = IMyService.Stub.asInterface(service);

try

{

// 调用AIDL服务对象中的getValue方法，并以对话框中显示该方法的返回值

new AlertDialog.Builder(Main.this).setMessage(

myService.getValue()).setPositiveButton("确定", null)

.show();

}

catch (Exception e)

{

}

}

@Override

public void onServiceDisconnected(ComponentName name)

{

}

};

@Override

public void onClick(View view)

{

// 绑定AIDL服务

bindService(new Intent("net.blogjava.mobile.aidlservice.IMyService"),

serviceConnection, Context.BIND\_AUTO\_CREATE);

}

... ...

}

在编写AIDL服务和客户端时要注意如下两点：

1. AIDL服务中的onBind方法必须返回AIDL接口对象（MyServiceImpl对象）。该对象也是onServiceConnected事件方法的第2个参数值。

2. bindService方法的第1个参数是Intent对象，该对象构造方法的参数需要指定AIDL服务的ID，也就是在AndroidManifest.xml文件中<service>标签的<action>子标签的android:name属性 的值。

现在先运行aidlservice程序，以便安装AIDL服务，然后运行aidlclient程序，并单击按钮，会显示如图11所示的对话框。对话框中的信息就是AIDL服务接口中getValue方法的返回值。

总结

本文介绍了4种跨进程通讯的方式：Activity、ContentProvider、Broadcast和AIDL Service。其中Activity可以跨进程调用其他应用程序的Activity；ContentProvider可以访问其他应用程序返回的Cursor对象；Broadcast采用的是被动接收的方法，也就是说，客户端只能接收广播数据，而不能向发送广播的程序发送信息。AIDL Service可以将程序中的某个接口公开，这样在其他的应用程序中就可以象访问本地对象一样访问AIDL服务对象了。这4种跨进程通讯的方式可以应用在 不同的场合，例如，在需要显示可视化的界面时可以用Activity，需要返回记录集时可以用ContentProvider。至于在应用程序中具体要用 到哪一种或几种方式进行跨进程通讯，读者可以根据实际情况进行选择。

转自： http://blog.csdn.net/nkmnkm/article/details/6451699

# IPC基础

Dfg

IPC是 Inter-Process Communication的缩写，含义为进程间通信或者跨进程通信，是指两个进程之间进行数据交换的过程。进程间通信的方式有：**Bundle、文件共享、AIDL、Messenger、ContentProvider和Socket**等。

Android系统中，每个应用程序是由Android的Activity，Service，Broadcast，ContentProvider这四剑客的中一个或多个组合而成，这四剑客所涉及的多进程间的通信底层都是依赖于Binder IPC机制。例如当进程A中的Activity要向进程B中的Service通信，这便需要依赖于Binder IPC。不仅于此，整个Android系统架构中，大量采用了Binder机制作为IPC（进程间通信）方案，当然也存在部分其他的IPC方式，比如Zygote通信便是采用socket。

Binder作为Android系统提供的一种IPC机制，无论从事系统开发还是应用开发，都应该有所了解，这是Android系统中最重要的组成，也是最难理解的一块知识点，错综复杂。要深入了解Binder机制，

IPC不是Android中所独有的，任何一个操作系统都需要相应的IPC机制，比如Windows上可以通过剪贴板等来进行进程间通信。Android是一种基于Linux内核的移动操作系统，它的进程间通信方式并不能完全继承自Linux，相反，它有自己的进程间通信方式。在Android中最有特色的进程间通信方式就是Binder了。

一个进程可以包含多个线程，因此进程和线程是包含与被包含的关系。最简单的情况下，一个进程中只可以有一个线程，即主线程，也叫UI线程。why we need 多进程，可能有很多，比如有些模块由于特殊原因需要运行在单独的进程中，或者为了加 大一个应用可使用的内存所以需要通过多进程来获取多份内存空间。Android对单个应用所使用的最大内存做了16M限制。另一种情况是当前应用需要向其他应用获取数据，由于是两个应用，所以必须采用跨进程的方式来获取所需的数据，甚至我们通过系统提供的ContentPra}ider去查询数据的时候

## 为什么Android要采用Binder作为IPC机制？

### Linux现有的所有进程间IPC方式：

 管道：在创建时分配一个page大小的内存，缓存区大小比较有限；

消息队列：信息复制两次，额外的CPU消耗；不合适频繁或信息量大的通信；

共享内存：无须复制，共享缓冲区直接付附加到进程虚拟地址空间，速度快；但进程间的同步问题操作系统无法实现，必须各进程利用同步工具解决；

套接字：作为更通用的接口，传输效率低，主要用于不通机器或跨网络的通信；

信号量：常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

信号: 不适用于信息交换，更适用于进程中断控制，比如非法内存访问，杀死某个进程等；

Android的内核也是基于Linux内核，为何不直接采用Linux现有的进程IPC方案呢，难道Linux社区那么多优秀人员都没有考虑到有Binder这样一个更优秀的方案，是google太过于牛B吗？事实是真相并非如此，请细细往下看，您就明白了。

-------------------------------------------------------------------------------------------------------------------------------------------

### 分析

接下来正面回答这个问题，从5个角度来展开对Binder的分析：

（1）从性能的角度

数据拷贝次数：Binder数据拷贝只需要一次，而管道、消息队列、Socket都需要2次，但共享内存方式一次内存拷贝都不需要；从性能角度看，Binder性能仅次于共享内存。

（2）从稳定性的角度

Binder是基于C/S架构的，简单解释下C/S架构，是指客户端(Client)和服务端(Server)组成的架构，Client端有什么需求，直接发送给Server端去完成，架构清晰明朗，Server端与Client端相对独立，稳定性较好；而共享内存实现方式复杂，没有客户与服务端之别， 需要充分考虑到访问临界资源的并发同步问题，否则可能会出现死锁等问题；从这稳定性角度看，Binder架构优越于共享内存。

仅仅从以上两点，各有优劣，还不足以支撑google去采用binder的IPC机制，那么更重要的原因是：

（3）从安全的角度

传统Linux IPC的接收方无法获得对方进程可靠的UID/PID，从而无法鉴别对方身份；而Android作为一个开放的开源体系，拥有非常多的开发平台，App来源甚广，因此手机的安全显得额外重要；对于普通用户，绝不希望从App商店下载偷窥隐射数据、后台造成手机耗电等等问题，传统Linux IPC无任何保护措施，完全由上层协议来确保。

Android为每个安装好的应用程序分配了自己的UID，故进程的UID是鉴别进程身份的重要标志，前面提到C/S架构，Android系统中对外只暴露Client端，Client端将任务发送给Server端，Server端会根据权限控制策略，判断UID/PID是否满足访问权限，目前权限控制很多时候是通过弹出权限询问对话框，让用户选择是否运行。Android 6.0，也称为Android M，在6.0之前的系统是在App第一次安装时，会将整个App所涉及的所有权限一次询问，只要留意看会发现很多App根本用不上通信录和短信，但在这一次性权限权限时会包含进去，让用户拒绝不得，因为拒绝后App无法正常使用，而一旦授权后，应用便可以胡作非为。

针对这个问题，google在Android M做了调整，不再是安装时一并询问所有权限，而是在App运行过程中，需要哪个权限再弹框询问用户是否给相应的权限，对权限做了更细地控制，让用户有了更多的可控性，但同时也带来了另一个用户诟病的地方，那也就是权限询问的弹框的次数大幅度增多。对于Android M平台上，有些App开发者可能会写出让手机异常频繁弹框的App，企图直到用户授权为止，这对用户来说是不能忍的，用户最后吐槽的可不光是App，还有Android系统以及手机厂商，有些用户可能就跳果粉了，这还需要广大Android开发者以及手机厂商共同努力，共同打造安全与体验俱佳的Android手机。

Android中权限控制策略有SELinux等多方面手段，下面列举从Binder的一个角度的权限控制：

Android源码的Binder权限是如何控制？ -Gityuan的回答

传统IPC只能由用户在数据包里填入UID/PID；另外，可靠的身份标记只有由IPC机制本身在内核中添加。其次传统IPC访问接入点是开放的，无法建立私有通道。从安全角度，Binder的安全性更高。

说到这，可能有人要反驳，Android就算用了Binder架构，而现如今Android手机的各种流氓软件，不就是干着这种偷窥隐射，后台偷偷跑流量的事吗？没错，确实存在，但这不能说Binder的安全性不好，因为Android系统仍然是掌握主控权，可以控制这类App的流氓行为，只是对于该采用何种策略来控制，在这方面android的确存在很多有待进步的空间，这也是google以及各大手机厂商一直努力改善的地方之一。在Android 6.0，google对于app的权限问题作为较多的努力，大大收紧的应用权限；另外，在Google举办的Android Bootcamp 2016大会中，google也表示在Android 7.0 （也叫Android N）的权限隐私方面会进一步加强加固，比如SELinux，Memory safe language(还在research中)等等，在今年的5月18日至5月20日，google将推出Android N。

话题扯远了，继续说Binder。

（4）从语言层面的角度

大家多知道Linux是基于C语言(面向过程的语言)，而Android是基于Java语言(面向对象的语句)，而对于Binder恰恰也符合面向对象的思想，将进程间通信转化为通过对某个Binder对象的引用调用该对象的方法，而其独特之处在于Binder对象是一个可以跨进程引用的对象，它的实体位于一个进程中，而它的引用却遍布于系统的各个进程之中。可以从一个进程传给其它进程，让大家都能访问同一Server，就像将一个对象或引用赋值给另一个引用一样。Binder模糊了进程边界，淡化了进程间通信过程，整个系统仿佛运行于同一个面向对象的程序之中。从语言层面，Binder更适合基于面向对象语言的Android系统，对于Linux系统可能会有点“水土不服”。

另外，Binder是为Android这类系统而生，而并非Linux社区没有想到Binder IPC机制的存在，对于Linux社区的广大开发人员，我还是表示深深佩服，让世界有了如此精湛而美妙的开源系统。也并非Linux现有的IPC机制不够好，相反地，经过这么多优秀工程师的不断打磨，依然非常优秀，每种Linux的IPC机制都有存在的价值，同时在Android系统中也依然采用了大量Linux现有的IPC机制，根据每类IPC的原理特性，因时制宜，不同场景特性往往会采用其下最适宜的。比如在Android OS中的Zygote进程的IPC采用的是Socket（套接字）机制，Android中的Kill Process采用的signal（信号）机制等等。而Binder更多则用在system\_server进程与上层App层的IPC交互。

(5) 从公司战略的角度

总所周知，Linux内核是开源的系统，所开放源代码许可协议GPL保护，该协议具有“病毒式感染”的能力，怎么理解这句话呢？受GPL保护的Linux Kernel是运行在内核空间，对于上层的任何类库、服务、应用等运行在用户空间，一旦进行SysCall（系统调用），调用到底层Kernel，那么也必须遵循GPL协议。

而Android 之父 Andy Rubin对于GPL显然是不能接受的，为此，Google巧妙地将GPL协议控制在内核空间，将用户空间的协议采用Apache-2.0协议（允许基于Android的开发商不向社区反馈源码），同时在GPL协议与Apache-2.0之间的Lib库中采用BSD证授权方法，有效隔断了GPL的传染性，仍有较大争议，但至少目前缓解Android，让GPL止步于内核空间，这是Google在GPL Linux下 开源与商业化共存的一个成功典范。

有了这些铺垫，我们再说说Binder的今世前缘

Binder是基于开源的 OpenBinder实现的，OpenBinder是一个开源的系统IPC机制,最初是由 Be Inc. 开发，接着由Palm, Inc.公司负责开发，现在OpenBinder的作者在Google工作，既然作者在Google公司，在用户空间采用Binder 作为核心的IPC机制，再用Apache-2.0协议保护，自然而然是没什么问题，减少法律风险，以及对开发成本也大有裨益的，那么从公司战略角度，Binder也是不错的选择。

另外，再说一点关于OpenBinder，在2015年OpenBinder以及合入到Linux Kernel主线 3.19版本，这也算是Google对Linux的一点回馈吧。

综合上述5点，可知Binder是Android系统上层进程间通信的不二选择。

### **D-Bus**

也采用C/S架构的IPC机制，D-Bus是在用户空间实现的方法，效率低，消息拷贝次数和上下文切换次数都明显多过于Binder。针对D-Bus这些缺陷，于是就产生了kdbus，这是D-Bus在内核实现版，效率得到提升，与Binder一样在内核作为字符设计，通过open()打开设备，mmap()映射内存。

（1）kdbus在进程间通信过程，Client端将消息在内存的消息队列，可以存储大量的消息，Server端不断从消息队里中取消息，大小只受限内存；

（2）Binder的机制是每次通信，会通信的进程或线程中的todo队里中增加binder事务，并且每个进程所允许Binder线程数，google提供的默认最大线程数为16个，受限于CPU，由于线程数太多，增加系统负载，并且每个进程默认分配的（1M-8K）大小的内存。

而kdbus对于内存消耗较大，同时也适合传输大量数据和大量消息的系统。Binder对CPU和内存的需求比较低，效率比较高，从而进一步说明Binder适合于移动系统Android，但是，也有一定缺点，就是不同利用Binder输出大数据，比如利用Binder传输几M大小的图片，便会出现异常，虽然有厂商会增加Binder内存，但是也不可能比系统默认内存大很多，否则整个系统的可用内存大幅度降低。

### Android Binder架构

Binder在Android系统中江湖地位非常之高。在Zygote孵化出system\_server进程后，在system\_server进程中出初始化支持整个Android framework的各种各样的Service，而这些Service从大的方向来划分，分为Java层Framework和Native Framework层(C++)的Service，几乎都是基于BInder IPC机制。

Java framework：作为Server端继承(或间接继承)于Binder类，Client端继承(或间接继承)于BinderProxy类。例如 ActivityManagerService(用于控制Activity、Service、进程等) 这个服务作为Server端，间接继承Binder类，而相应的ActivityManager作为Client端，间接继承于BinderProxy类。 当然还有PackageManagerService、WindowManagerService等等很多系统服务都是采用C/S架构；

Native Framework层：这是C++层，作为Server端继承(或间接继承)于BBinder类，Client端继承(或间接继承)于BpBinder。例如MediaPlayService(用于多媒体相关)作为Server端，继承于BBinder类，而相应的MediaPlay作为Client端，间接继承于BpBinder类。

总之，一句话"无Binder不Android"。

---------------------

作者：Gityuan

来源：CSDN

原文：https://blog.csdn.net/Gityuan/article/details/50815110

版权声明：本文为博主原创文章，转载请附上博文链接！

### 参考

<https://blog.csdn.net/Gityuan/article/details/50815110>

--------------------------------------------------------------------------------

## 多进程模式

在Android中使用多进程只有一种方法：给四大组件(Activity、Service、Receiver、ContentProvider)在AndroidManifest中指定android:process属性，除此之外没有其他办法(除去通过JNI在native层去fork):

● android:process=”:remote”，私有进程，这种方式的标记的进程名为包名:remote，“:“的含义是指要在当前的进程名前附加包名，其次，进程名以“：”开头的进程属于当前应用的私有进程，其他应用的组件不可以和它跑在同一个进程中。

● android:process=”com.example.wpp.remote”，全局进程，这种声明方式是完整的命名方式，不会附件包名信息，属于全局进程，其他应用通过ShareUID方式可以和它跑在同一个进程中

Android系统会为每个应用分配一个唯一的UID，具有相同UID的应用才能共享数据，而两个应用通过ShareUID跑在同一个进程需要这两个应用有相同的ShareUID并且签名相同才可以，在这种情况下，它们可以互相 访问对方的私有数据，比如data目录、组件信息等，不管它们是否泡在同一个进程中。当然如果它们跑在同一个进程中，那么除了能共享data目录、组件信息，还可以共享内存数据

使用多进程会造成如下几方面的问题：

1. 静态成员和单例模式完全失效 （每个进程都分配一个独立的虚拟机，有着不同的虚拟机空间）

2. 线程同步机制完全失效。（不管是锁对象还是锁全局类都无法保证线程同步，因为不同进程锁的对象也不同了）

3. SharedPreferences的可靠性下降

4. Application会多次创建。

## Q:多进程如果共享数据？

### 2.4.2使用文件共享

对数据同步要求不高的进程之间进行遁信，基本走java的路线了

oos = new ObjectOutputStream( new FileOutputStream(file));

oos.writeObject(user);

注:SharePreferences是一个轻量级的存储方案，但是系统对这个的读写存在一定的缓存策略，在多进程高并发的情况下，丢失数据的概率很高，因此进程之间不推荐用这种方式通信(之前和zhongping就是一次失败的情况)

### 2.4.3使用Messenger

服务端

messenger = new Messenger(new MyHandler());

客户端

messenger = new Messenger(iBinder);

m.replyTo = mGetReplyMessenger;

### 2.4.6 使用Socket

### Binder

--------------------------------------------------------------------------------

#### 序列化

基本概念：Serializable接口、Parcelable接口以及Binder

##### Serializable接口

SerialiaabIe是Java中的序列化, 其使用起来简单但是开销很大，序列化到**存储设备**中或者将对象序列化后通过网络传输

##### Parcelable接口

用来保存到本地的类特别好，取代了shareprefer

Parcelable是Android中的序列化方式，使用起来稍微麻烦点，但是它的**效率很高**，主要用在**内存序列化**

Parcelable接口是Android提供的新的序列化方式，Parcelable也是一个接口，下面的示例是一个典型的用法。

public class User implements Parcelable{

public int userId;

public String userName;

public boolean isMale;

public Book book;

public User(int userId, String userName, boolean isMale){

this.userId = userId;

this.userName = userName;

this.isMale = isMale;

}

public int describeContents(){

return 0;

}

public void writeToParcel(Parcel out, int flags){

out.writeInt(userId);

out.wirteString(userName);

out.wirteInt(isMale ? 1 : 0);

out.writeParcelable(book, 0);

}

public static final Parcelable.Creator<User> CREATOR = new Parcelable.Creator<User>(){

public User createFromParcel(Parcel in){

return new User(in);

}

public User[] new Array(int size){

return new User[size];

}

};

private User(Parcel in){

userId = in.readInt();

userName = in.readString();

isMale = in.readInt()==1;

book = in.readParcelable(Thread.currentThread().getContextClassLoader());

}

}

Parcelable方法说明



方法功能标记位createFromParcel(Parcel in)从序列化后的对象中创建原对象new Array(int size)创建指定长度的原始对象数组User(Parcel in)从序列化后的对象中创建原始对象writeToParcel(Parcel out, int flag)将当前对象写入序列化结构中，其中flag标识有两种值：0或者1.为1时标识当前对象需要作为返回值返回，不能立即释放资源，几乎所有情况都为0PARCELABLE\_WRITE\_RETURN\_VALUEdescribeContents返回当前对象的内容描述，如果还有文件描述符，返回1，否则返回0，几乎所有情况都返回0

系统已经为我们提供了许多实现了Parcelable接口的类，它们都是可以直接序列化的，比如Intent、Bundle、Bitmap等，同时List和Map也可以序列化，前提是它们里面每个元素都是可序列化的。

### 使用场景Binder

选择AIDL的使用场合

官方文档特别提醒我们何时使用AIDL是必要的：只有你允许客户端从不同的应用程序为了进程间的通信而去访问你的service，以及想在你的service处理多线程。

## Parcel深入理解

### 一、Android中的Parcel是什么

Parcel，翻译过来是“打包”的意思。打包干什么呢？是为了序列化。

如果要在进程之间传递一个整数，很简单，直接传就是行了；如果要传一个字符串，就稍微复杂了点：需先分配一块可以容纳字符串的 内存，然后将字符串复制到内存中，再传递（新手可能问：为啥不直接把字符串的引用传过去呢？学过C/C++的地球人都知道：进程有自己的内存地址空间，一 个进程中的1000地址可能在另一个进程中是100000，java对象的引用跟本上还是内存地址）；再如果要传递一个类的实例呢？也是先为类分配内存， 然后复制一份再传递可以吗？我认为不可以，我至少可以找到一个理由：类中成员除了属性还有方法，即使属性能完整传过去，但还有方法呢？方法是独立于类对象 存在的，所以到另一个进程中再引用同一个方法就要出错了，还是因为独立地址空间的原因。

Android开发中，很经常在各activity之间传递数据，而跟据Android的设计架构，即使同一个程序中的Activity都不一定运行在同 一个进程中，所以处理数据传递时你不能老假设两个activity都运行于同一进程，那么只能按进程间传递数据来处理，使之具有最广泛的适应性。

　 　那么到底如何在进程之间传递类对象呢？简单来说可以这样做：在进程Ａ中把类中的非默认值的属性和类的唯一标志打成包（这就叫序列化），把这个包传递到进 程Ｂ，进程Ｂ接收到包后，跟据类的唯一标志把类创建出来，然后把传来的属性更新到类对象中，这样进程Ａ和进程Ｂ中就包含了两个完全一样的类对象。

二、 探索Android中的Parcel机制（上）

转自：http://blog.csdn.net/caowenbin/article/details/6532217 (作者：曹文斌)

一．先从Serialize说起

我们都知道JAVA中的Serialize机制，译成串行化、序列化……，其作用是能将数据对象存入字节流当中，在需要时重新生成对象。主要应用是利用外部存储设备保存对象状态，以及通过网络传输对象等。

二．Android中的新的序列化机制

在Android系统中，定位为针对内存受限的设备，因此对性能要求更高，另外系统中采用了新的IPC（进程间通信）机制，必然 要求使用性能更出色的对象传输方式。在这样的环境下，Parcel被设计出来，其定位就是轻量级的高效的对象序列化和反序列化机制。

#### 三．Parcel类的背后

在Framework中有parcel类，源码路径是：

Frameworks/base/core/java/android/os/Parcel.java

典型的源码片断如下：

[html] view plain copy print ?

/\*\*

\* Write an integer value into the parcel at the current dataPosition(),

\* growing dataCapacity() if needed.

\*/

public final native void writeInt(int val);

/\*\*

\* Write a long integer value into the parcel at the current dataPosition(),

\* growing dataCapacity() if needed.

\*/

public final native void writeLong(long val);

从中我们看到，从这个源程序文件中我们看不到真正的功能是如何实现的，必须透过JNI往下走了。于是，Frameworks/base/core/jni/android\_util\_Binder.cpp中找到了线索

[html] view plain copy print ?

static void android\_os\_Parcel\_writeInt(JNIEnv\* env, jobject clazz, jint val)

{

Parcel\* parcel = parcelForJavaObject(env, clazz);

if (parcel != NULL) {

const status\_t err = parcel->writeInt32(val);

if (err != NO\_ERROR) {

jniThrowException(env, "java/lang/OutOfMemoryError", NULL);

}

}

}

static void android\_os\_Parcel\_writeLong(JNIEnv\* env, jobject clazz, jlong val)

{

Parcel\* parcel = parcelForJavaObject(env, clazz);

if (parcel != NULL) {

const status\_t err = parcel->writeInt64(val);

if (err != NO\_ERROR) {

jniThrowException(env, "java/lang/OutOfMemoryError", NULL);

}

}

}

#### Parcel指针

从这里我们可以得到的信息是函数的实现依赖于Parcel指针，因此还需要找到Parcel的类定义，注意，这里的类已经是用C++语言实现的了。

找到Frameworks/base/include/binder/parcel.h和Frameworks/base/libs/binder/parcel.cpp。终于找到了最终的实现代码了。

有兴趣的朋友可以自己读一下，不难理解，这里把基本的思路总结一下：

1. 整个读写全是在内存中进行，主要是通过malloc()、realloc()、memcpy()等内存操作进行，所以效率比JAVA序列化中使用外部存储器会高很多；

2. 读写时是4字节对齐的，可以看到#define PAD\_SIZE(s) (((s)+3)&~3)这句宏定义就是在做这件事情；

3. 如果预分配的空间不够时newSize = ((mDataSize+len)\*3)/2;会一次多分配50%；

4. 对于普通数据，使用的是mData内存地址，对于IBinder类型的数据以及FileDescriptor使用的是 mObjects内存地址。后者是通过flatten\_binder()和unflatten\_binder()实现的，目的是反序列化时读出的对象就是 原对象而不用重新new一个新对象。

#### 三、Parcel应用demo

上一篇中我们透过源码看到了Parcel背后的机制，本质上把它当成一个Serialize就可以了，只是它是在内存中完成的序列化和反序列化，利用的是连续的内存空间，因此会更加高效。

我们接下来要说的是Parcel类如何应用。就应用程序而言，最常见使用Parcel类的场景就是在Activity间传递数据。没错，在Activity间使用Intent传递数据的时候，可以通过Parcelable机制传递复杂的对象。

在下面的程序中，MyColor用于保存一个颜色值，MainActivity在用户点击屏幕时将MyColor对象设成红色， 传递到SubActivity中，此时SubActivity的TextView显示为红色的背景；当点击SubActivity时，将颜色值改为绿色， 返回MainActivity，期望的是MainActivity的TextView显示绿色背景。

来看一下MyColor类的实现代码：

[html] view plain copy print ?

package com.wenbin.test;

import android.graphics.Color;

import android.os.Parcel;

import android.os.Parcelable;

/\*\*

\* @author 曹文斌

\* http://blog.csdn.net/caowenbin

\*

\*/

public class MyColor implements Parcelable {

private int color=Color.BLACK;

MyColor(){

color=Color.BLACK;

}

MyColor(Parcel in){

color=in.readInt();

}

public int getColor(){

return color;

}

public void setColor(int color){

this.color=color;

}

@Override

public int describeContents() {

return 0;

}

@Override

public void writeToParcel(Parcel dest, int flags) {

dest.writeInt(color);

}

public static final Parcelable.Creator<MyColor> CREATOR

= new Parcelable.Creator<MyColor>() {

public MyColor createFromParcel(Parcel in) {

return new MyColor(in);

}

public MyColor[] newArray(int size) {

return new MyColor[size];

}

};

}

该类实现了Parcelable接口，提供了默认的构造函数，同时也提供了可从Parcel对象开始的构造函数，另外还实现了一个static的构造器用于构造对象和数组。代码很简单，不一一解释了。

再看MainActivity的代码：

[html] view plain copy print ?

package com.wenbin.test;

import android.app.Activity;

import android.content.Intent;

import android.graphics.Color;

import android.os.Bundle;

import android.view.MotionEvent;

/\*\*

\* @author 曹文斌

\* http://blog.csdn.net/caowenbin

\*

\*/

public class MainActivity extends Activity {

private final int SUB\_ACTIVITY=0;

private MyColor color=new MyColor();

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

}

@Override

protected void onActivityResult(int requestCode, int resultCode, Intent data) {

super.onActivityResult(requestCode, resultCode, data);

if (requestCode==SUB\_ACTIVITY){

if (resultCode==RESULT\_OK){

if (data.hasExtra("MyColor")){

color=data.getParcelableExtra("MyColor"); //Notice

findViewById(R.id.text).setBackgroundColor(color.getColor());

}

}

}

}

@Override

public boolean onTouchEvent(MotionEvent event){

if (event.getAction()==MotionEvent.ACTION\_UP){

Intent intent=new Intent();

intent.setClass(this, SubActivity.class);

color.setColor(Color.RED);

intent.putExtra("MyColor", color);

startActivityForResult(intent,SUB\_ACTIVITY);

}

return super.onTouchEvent(event);

}

}

下面是SubActivity的代码：

[html] view plain copy print ?

package com.wenbin.test;

import android.app.Activity;

import android.content.Intent;

import android.graphics.Color;

import android.os.Bundle;

import android.view.MotionEvent;

import android.widget.TextView;

/\*\*

\* @author 曹文斌

\* http://blog.csdn.net/caowenbin

\*

\*/

public class SubActivity extends Activity {

private MyColor color;

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

((TextView)findViewById(R.id.text)).setText("SubActivity");

Intent intent=getIntent();

if (intent!=null){

if (intent.hasExtra("MyColor")){

color=intent.getParcelableExtra("MyColor");

findViewById(R.id.text).setBackgroundColor(color.getColor());

}

}

}

@Override

public boolean onTouchEvent(MotionEvent event){

if (event.getAction()==MotionEvent.ACTION\_UP){

Intent intent=new Intent();

if (color!=null){

color.setColor(Color.GREEN);

intent.putExtra("MyColor", color);

}

setResult(RESULT\_OK,intent);

finish();

}

return super.onTouchEvent(event);

}

}

下面是main.xml的代码：

[html] view plain copy print ?

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="vertical"

android:layout\_width="fill\_parent"

android:layout\_height="fill\_parent"

>

<TextView

android:layout\_width="fill\_parent"

android:layout\_height="wrap\_content"

android:text="@string/hello"

android:id="@+id/text"

/>

</LinearLayout>

注意的是在MainActivity的onActivityResult()中，有一句 color=data.getParcelableExtra("MyColor")，这说明的是反序列化后是一个新的MyColor对象，因此要想使用 这个对象，我们做了这个赋值语句。

记得在上一篇《探索Android中的Parcel机制（上）》 中提到，如果数据本身是IBinder类型，那么反序列化的结果就是原对象，而不是新建的对象，很显然，如果是这样的话，在反序列化后在 MainActivity中就不再需要color=data.getParcelableExtra("MyColor")这句了。因此，换一种 MyColor的实现方法，令其中的int color成员变量使用IBinder类型的成员变量来表示。

新建一个BinderData类继承自Binder，代码如下：

[html] view plain copy print ?

package com.wenbin.test;

import android.os.Binder;

/\*\*

\* @author 曹文斌

\* http://blog.csdn.net/caowenbin

\*

\*/

public class BinderData extends Binder {

public int color;

}

修改MyColor的代码如下：

[html] view plain copy print ?

package com.wenbin.test;

import android.graphics.Color;

import android.os.Parcel;

import android.os.Parcelable;

/\*\*

\* @author 曹文斌

\* http://blog.csdn.net/caowenbin

\*

\*/

public class MyColor implements Parcelable {

private BinderData data=new BinderData();

MyColor(){

data.color=Color.BLACK;

}

MyColor(Parcel in){

data=(BinderData) in.readValue(BinderData.class.getClassLoader());

}

public int getColor(){

return data.color;

}

public void setColor(int color){

data.color=color;

}

@Override

public int describeContents() {

return 0;

}

@Override

public void writeToParcel(Parcel dest, int flags) {

dest.writeValue(data);

}

public static final Parcelable.Creator<MyColor> CREATOR

= new Parcelable.Creator<MyColor>() {

public MyColor createFromParcel(Parcel in) {

return new MyColor(in);

}

public MyColor[] newArray(int size) {

return new MyColor[size];

}

};

}

去掉MainActivity的onActivityResult()中的color=data.getParcelableExtra("MyColor")一句，变成：

[html] view plain copy print ?

@Override

protected void onActivityResult(int requestCode, int resultCode, Intent data) {

super.onActivityResult(requestCode, resultCode, data);

if (requestCode==SUB\_ACTIVITY){

if (resultCode==RESULT\_OK){

if (data.hasExtra("MyColor")){

findViewById(R.id.text).setBackgroundColor(color.getColor());

}

}

}

}

再次运行程序，结果符合预期。

以上就是Parcel在应用程序中的使用方法，与Serialize还是挺相似的，详细的资料当然还是要参考Android SDK的开发文档了。

#### 三、Android Parcel内存模型

android 中Parcel 的使用，他是一个存储基本数据类型和引用数据类型的容器，在andorid 中通过IBinder来绑定数据在进程间传递数据。

Parcel parcel = Parcel.obtain();// 获取一个Parcel 对象

下面就可以对其进行方法进行操作了，createXXX(),wirteXXX(),readXXX(),

其中 dataPosition(),返回当前Parcel 当前对象存储数据的偏移量，而setDataPosition(),设置当前Parcel 对象的偏移量，方便读取parcel 中的数据，可问题就出在我读取出来的数据要么是空（null）,要么永远是第一个偏移量处的值，存储和读取数据的。Parcel采用什么机制实现的，是以 什么形式的存储的，然后我才能任意对其操作，读取目标数据。

基本数据类型的取值范围，

boolean 1bit

short 16bit

int 32bit

long 64bit

float 32bit

double 64bit

char 16bit

byte 8bit

由此我 可以猜想，Parcel 32bit 作为基本单位存储写入的变量,4byte\*8=32bit,在内存中的引用地址变量是采用16进制进行编码，且作为偏移量，即偏移量是4的倍 数，0,4,8,12,16,20,24,28,32,36,40,44,48......4\*N,

f(x) = 4\*y｛y>=0&y是自然数｝

我想绝对不会出现向偏移量是3，6，9这样的数据。。。

由此我们可以推断出，无论他存储的是基本数据类型或引用数据类型的变量，都是以32bit基本单位作为偏移量，

parcel.writeInt(1);

parcel.writeInt(2);

parcel.writeInt(3);

parcel.writeInt(4);

parcel.writeInt(5);

parcel.writeInt(6);

parcel.writeInt(7);

parcel.writeInt(81011111);

parcel.writeFloat(1f);

parcel.writeFloat(1000000000000000000000000000000000000f);

parcel.writeXXX(), 每写一次数据，在32bit的空间里能够存储要放入的变量，怎只占一个偏移量，也就之一动4个位置，而当存储的数据如 parcel.writeFloat(1000000000000000000000000000000000000f);他就自动往后移动，

parcel.writeString("a");

parcel.writeString("b");

parcel.writeString("d");

parcel.writeString("c");

和

parcel.writeString("abcd"); 的区别。有此可见，他的内存的分配原来是这样的。

那我怎样才能把我存进去的书据依次的去出来呢？setDataPosition(),设置parcel 的偏移量，在readXXX(),读取数据

int size = parcel.dataSize();

int i = 0;

while (i <= size ) {

parcel.setDataPosition(i);

int curr\_int = parcel.readInt();

i+=4;

int j = 0;

j++;

}

由此可 见parcel 写入数据是按照32bit 为基本的容器，依次存储写入的数据，基本和引用（其实引用的也是有多个基本数据类型组合而成OBJECTS－属性｜方法），读取的时候我们就可以按照这种 规律根据目标数据的偏移量的位置（curr\_position），以及偏移量的大小(size)，,取出已经存进去的数据了

int i ＝ curr\_position；

while (i <= size ) {

parcel.setDataPosition(i);

int curr\_int = parcel.readXXXt();

i+=4;

int j = 0;

j++;

}

这样就ok 了

他的createXXX（）方法现在没用，用了在说吧！

总结一句话，java 中 基本数据类型的取值范围，引用类型的数据，相当于c中的指针，以及各进制之间的相互转换和灵活的引用，以及定制自己想要的任意进制数据类型。

四、 Android开发：什么是Parcel(2)

转自：http://blog.csdn.net/nkmnkm/article/details/6453391

上回书解释了IBinder,这回详细解释一下Parcel，以下是对android sdk 文档的翻议：

Parcel是一个容器，它主要用于存储序列化数据，然后可以通过Binder在进程间传递这些数据（要了解为什么要序列化，请参考：http://blog.csdn.net/nkmnkm/archive/2011/05/28/6451699.aspx）。Parcel可以包含原始数据类型（用各种对应的方法写入，比如writeInt(),writeFloat()等），可以包含Parcelable对象，它还包含了一个活动的IBinder对象的引用，这个引用导致另一端接收到一个指向这个IBinder的代理IBinder。

注：Parcel不是一般目的的序列化机制。这个类被设计用于高性能的IPC传输。因此不适合把Parcel写入永久化存储中，因为Parcel中的数据类型的实现的改变会导致旧版的数据不可读。

Parcel的一坨一坨的API用于解决不同类型数据的读写。这些函数们主要有六种类型。

１原始类

这类方法们主要读写原始数据类型。它们是：writeByte(byte), readByte(), writeDouble(double), readDouble(), writeFloat(float), readFloat(), writeInt(int), readInt(), writeLong(long), readLong(), writeString(String), readString(). 大多数其它数据的操作都是基于这些方法。

２原始数组类

这类方法用于读写原始数据组成的数组。在向数组写数据时先写入数组的长度再写入数据。读数组的方法可以将数据读到已存在的数组中，也可以创建并返回一个新数组。它们是：

writeBooleanArray(boolean[]), readBooleanArray(boolean[]), createBooleanArray()

writeByteArray(byte[]), writeByteArray(byte[], int, int), readByteArray(byte[]), createByteArray()

writeCharArray(char[]), readCharArray(char[]), createCharArray()

writeDoubleArray(double[]), readDoubleArray(double[]), createDoubleArray()

writeFloatArray(float[]), readFloatArray(float[]), createFloatArray()

writeIntArray(int[]), readIntArray(int[]), createIntArray()

writeLongArray(long[]), readLongArray(long[]), createLongArray()

writeStringArray(String[]), readStringArray(String[]), createStringArray().

writeSparseBooleanArray(SparseBooleanArray), readSparseBooleanArray().

3 Parcelable类

Parcelable为对象从Parcel中读写自己提供了极其高效的协议。你可以使用直接的方法 writeParcelable(Parcelable, int) 和 readParcelable(ClassLoader) 或 writeParcelableArray(T[], int) and readParcelableArray(ClassLoader) 进行读写。这些方法们把类的信息和数据都写入Parcel，以使将来能使用合适的类装载器重新构造类的实例。

还有一些方法提供了更高效的操作Parcelable们的途径，它们是：writeTypedArray(T[], int), writeTypedList(List), readTypedArray(T[], Parcelable.Creator) and readTypedList(List, Parcelable.Creator)。这些方法不会写入类的信息，取而代之的是：读取时必须能知道数据属于哪个类并传入正确的 Parcelable.Creator来创建对象而不是直接构造新对象。（更加高效的读写单个Parcelable对象的方法是：直接调用 Parcelable.writeToParcel()和Parcelable.Creator.createFromParcel()）

4 Bundles类

Bundles是一种类型安全的Map型容器，可用于存储任何不同类型的数据。它具有很多对讀写数据的性能优化，并且它的类型安全机制避免了当把它的数据 封送到Parcel中时由于类型错误引起的BUG的调试的麻烦，可以使用的方法为： writeBundle(Bundle), readBundle(), and readBundle(ClassLoader)。

5 活动对象类

Parcel的一个非同寻常的特性是读写活对象的能力。对于活动对象，它们的内容实际上并没有写入，而是仅写入了一个令牌来引用这个对象。当从Parcel中读取这个对象时，你不会获取一个新的对象实例，而是直接得到那个写入的对象。有两种活动对象可操作：

Binder对象。它是 Android跨进程通讯的基础。这种对象可被写入Parcel，并在读取时你将得到原始的对象或一个代理对象（可以想象：在进程内时得到原始的对象，在 进程间时得到代理对象）。可以使用的方法们是： writeStrongBinder(IBinder), writeStrongInterface(IInterface), readStrongBinder(), writeBinderArray(IBinder[]), readBinderArray(IBinder[]), createBinderArray(), writeBinderList(List), readBinderList(List), createBinderArrayList()。

FileDescriptor对象。 它代表了原始的Linux文件描述符，它可以被写入Parcel并在读取时返回一个ParcelFileDescriptor对象用于操作原始的文件描述 符。ParcelFileDescriptor是原始描述符的一个复制：对象和fd不同，但是都操作于同一文件流，使用同一个文件位置指针，等等。可以使 用的方法是：writeFileDescriptor(FileDescriptor), readFileDescriptor()。

6无类型容器类

一类final方法，用于读写标准的java容器类。这些方法们是：writeArray(Object[]), readArray(ClassLoader), writeList(List), readList(List, ClassLoader), readArrayList(ClassLoader), writeMap(Map), readMap(Map, ClassLoader), writeSparseArray(SparseArray), readSparseArray(ClassLoader)。

三、参考

1.android parcelable 详解

http://jingyan.baidu.com/article/f96699bbae42cd894e3c1b15.html

2. 探索Android中的Parcel

<http://my.oschina.net/zhuzihasablog/blog/102316>

## AIDL