**static final int** TRANSACTION\_onSuccess = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 0);

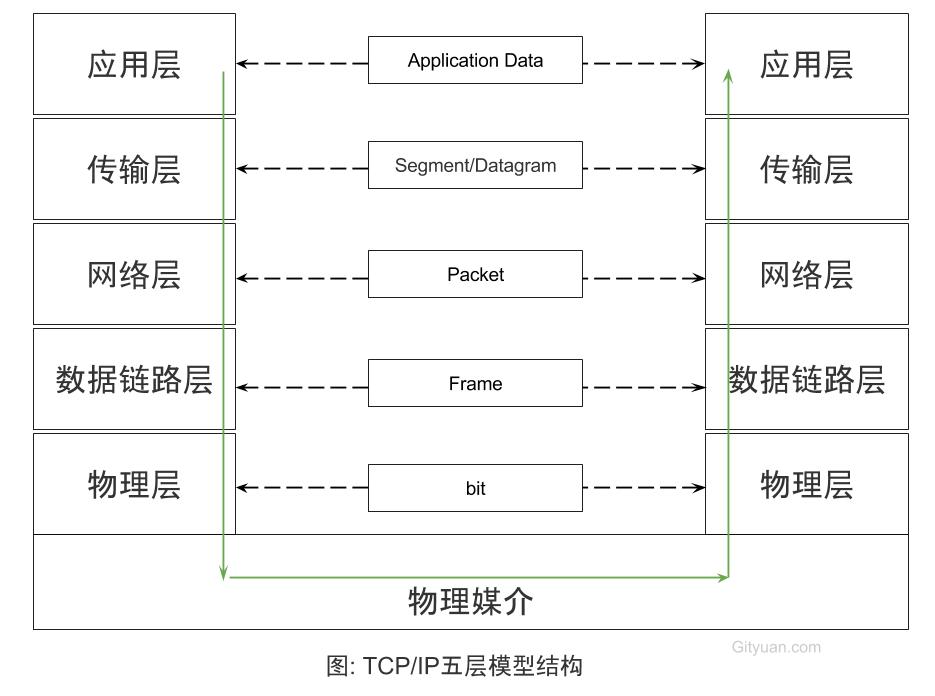
onSuccess

# 彻底理解Binder架构

### 引言

#### 1.1 Binder架构的思考

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:

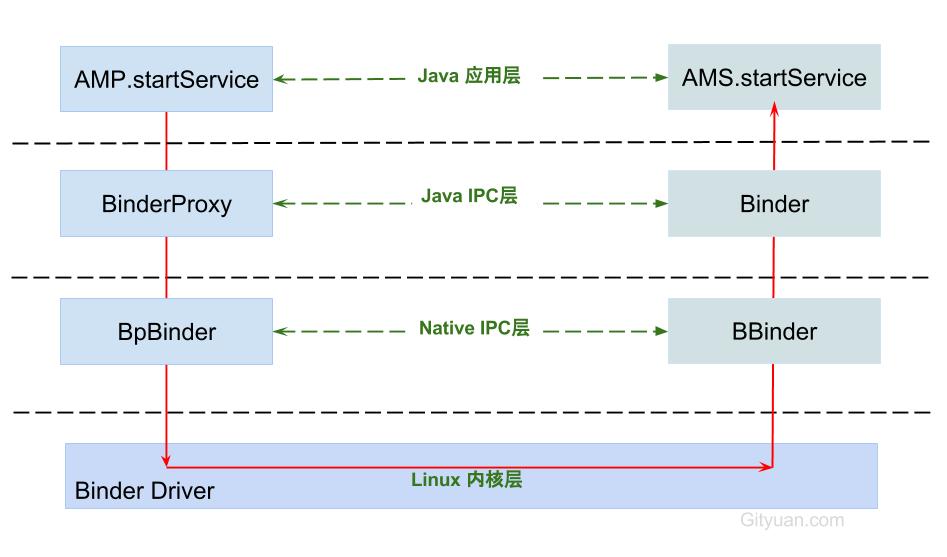


* 应用层: 直接为用户提供服务;
* 传输层: 传输的是报文(TCP数据)或者用户数据报(UDP数据)
* 网络层: 传输的是包(Packet), 例如路由器
* 数据链路层: 传输的是帧(Frame), 例如以太网交换机
* 物理层: 相邻节点间传输bit, 例如集线器,双绞线等

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:

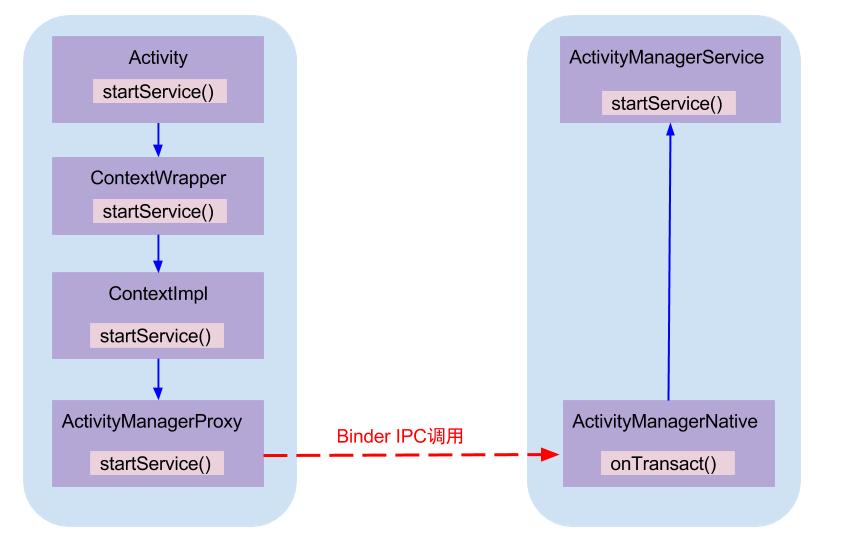


* **Java应用层:** 对于上层应用通过调用AMP.startService, 完全可以不用关心底层,经过层层调用,最终必然会调用到AMS.startService.
* **Java IPC层:** Binder通信是采用C/S架构, Android系统的基础架构便已设计好Binder在Java framework层的Binder客户类BinderProxy和服务类Binder;
* **Native IPC层:** 对于Native层,如果需要直接使用Binder(比如media相关), 则可以直接使用BpBinder和BBinder(当然这里还有JavaBBinder)即可, 对于上一层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.

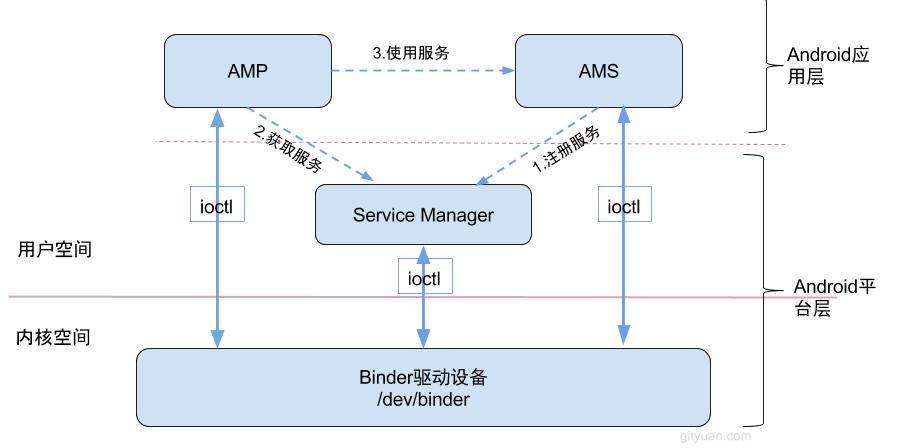
Binder系统如此庞大, 那么这里需要寻求一个出发点来穿针引线, 一窥视Binder全貌. 那么本文将从全新的视角,以[startService流程分析](http://gityuan.com/2016/03/06/start-service/)为例子来说说Binder所其作用. 首先在发起方进程调用AMP.startService，经过binder驱动，最终调用系统进程AMS.startService,如下图:



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]

//此处obj = BinderProxy, descriptor = "android.app.IActivityManager"; [见流程2.4.5]

**static public** IActivityManager asInterface(IBinder obj) {  
 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;

}

...

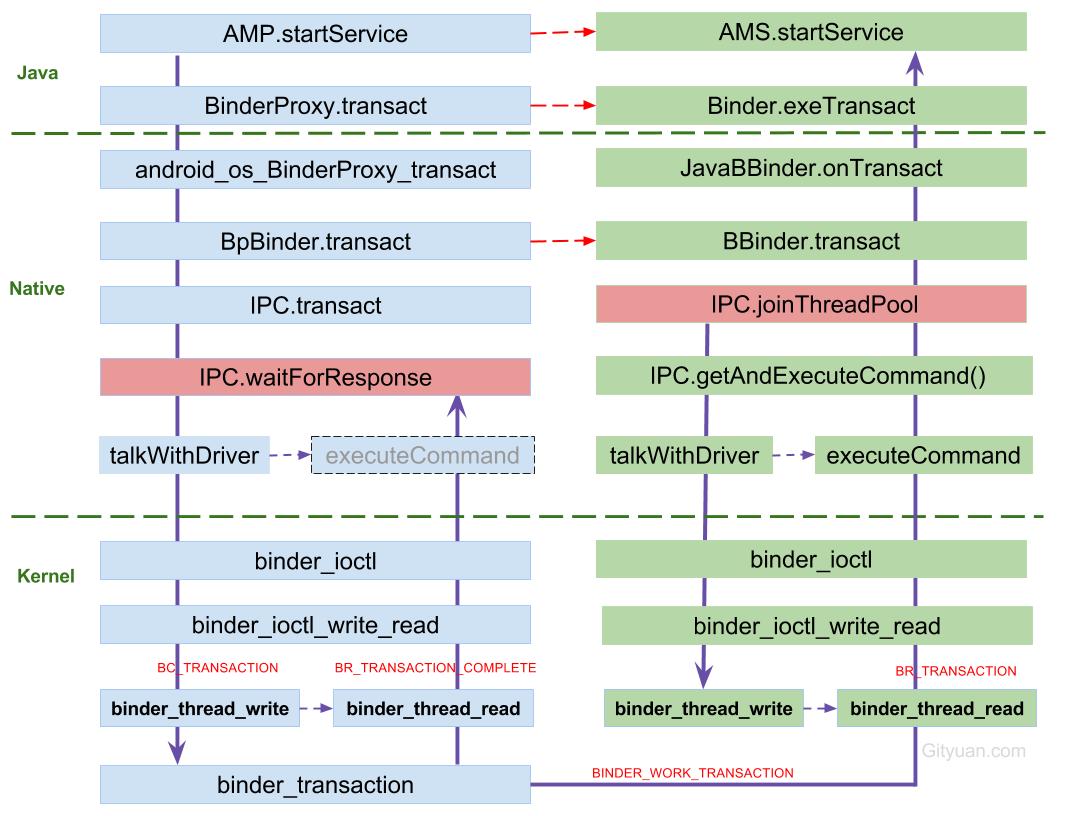
return 0;

}

## 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

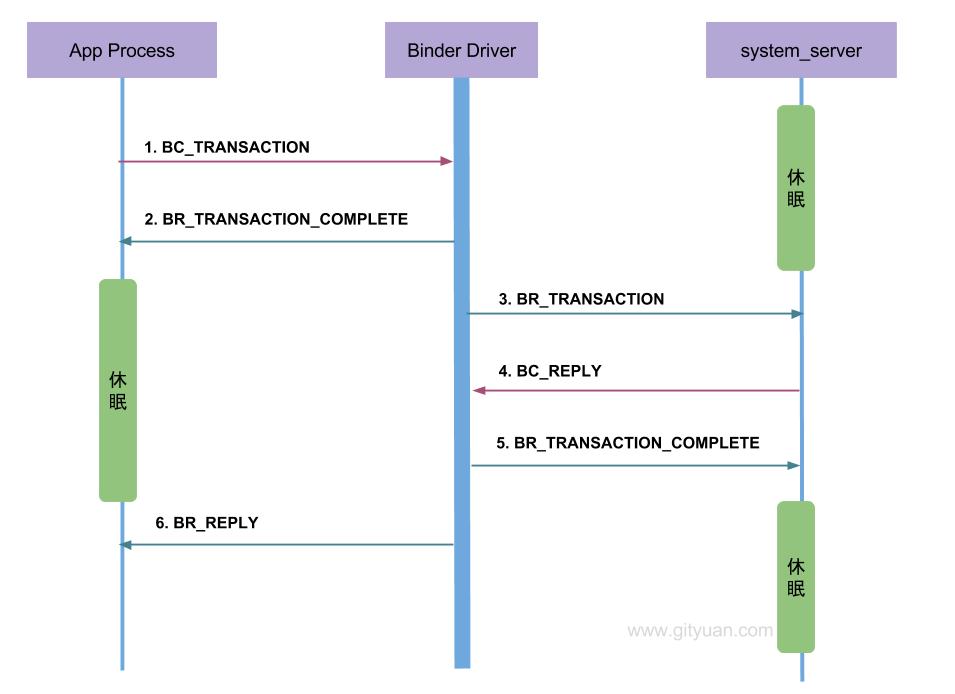
从通信流程角度来看整个过程:

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

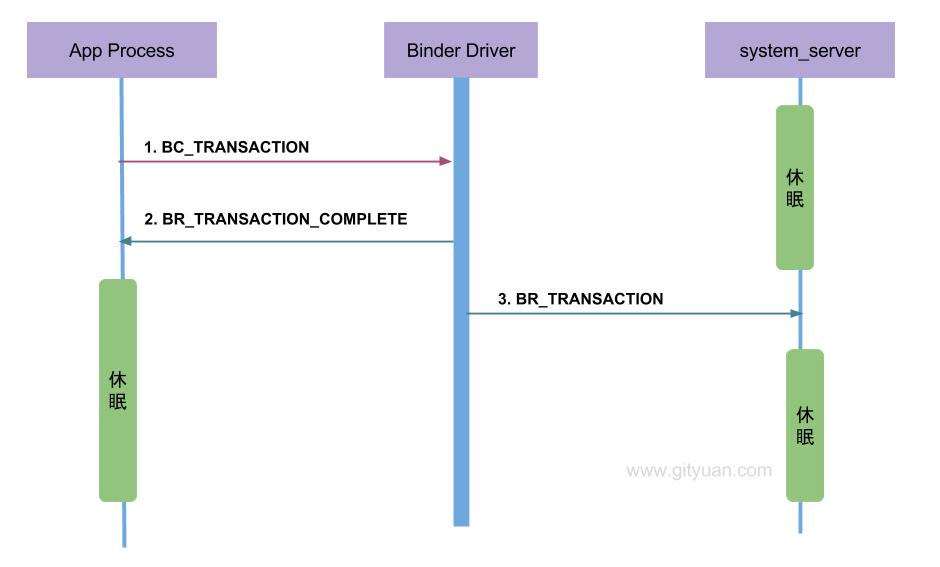
从通信协议的角度来看这个过程:



* 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.

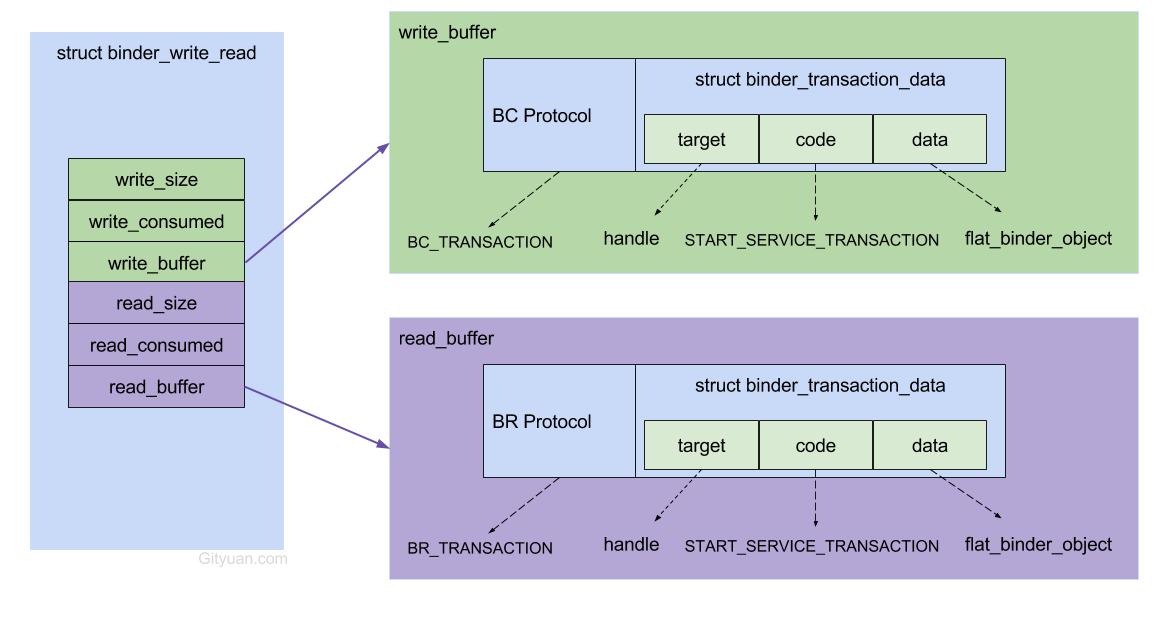
oneway与非oneway: 都是需要等待Binder Driver的回应消息BR\_TRANSACTION\_COMPLETE. 主要区别在于oneway的通信收到BR\_TRANSACTION\_COMPLETE则返回,而不会再等待BR\_REPLY消息的到来. 另外，oneway的binder IPC则接收端无法获取对方的pid.

### 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/>