ServiceManager

概述:SM的功能就像是互联网中的DNS服务器,和DNS自身也是服务器一样SM自身也是一个标准的BinderServer,它的IP地址是0.SM提供了[名称 BinderService句柄]对应关系的备案注册功能和通过名称获取对应的远程BinderServer句柄等功能.

SM启动:

 /[system](http://androidxref.com/9.0.0_r3/xref/system/)/[core](http://androidxref.com/9.0.0_r3/xref/system/core/)/[rootdir](http://androidxref.com/9.0.0_r3/xref/system/core/rootdir/)/[init.rc](http://androidxref.com/9.0.0_r3/xref/system/core/rootdir/init.rc).

on post-fs

[317](http://androidxref.com/9.0.0_r3/xref/system/core/rootdir/init.rc" \l "317) load\_system\_props

[318](http://androidxref.com/9.0.0_r3/xref/system/core/rootdir/init.rc" \l "318) # start essential services

[319](http://androidxref.com/9.0.0_r3/xref/system/core/rootdir/init.rc" \l "319) start logd

[320](http://androidxref.com/9.0.0_r3/xref/system/core/rootdir/init.rc" \l "320) start servicemanager

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[native](http://androidxref.com/9.0.0_r3/xref/frameworks/native/)/[cmds](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/)/[servicemanager](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/)/[service\_manager.c](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/service_manager.c)

int main(int argc, char\*\* argv){

struct binder\_state \*bs;

union selinux\_callback cb;

char \*driver;

if (argc > 1) {

driver = argv[1];

} else {

driver = "/dev/binder";

}

//打开binder设备,做好初始化,并接收返回的记录了SM中与binder相关的所有信息的结构体

bs = binder\_open(driver, 128\*1024);

//把自己设置为Binder管家,DNS服务器的角色

if (binder\_become\_context\_manager(bs)) {

ALOGE("cannot become context manager (%s)\n", strerror(errno));

return -1;

}

//开启循环,等待客户端的注册查询等请求

binder\_loop(bs, svcmgr\_handler);

return 0;

}

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[native](http://androidxref.com/9.0.0_r3/xref/frameworks/native/)/[cmds](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/)/[servicemanager](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/)/[binder.c](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/binder.c)

struct binder\_state \*binder\_open(const char\* driver, size\_t mapsize){

//结构体,记录了SM中与binder相关的所有信息,初始化后返回

struct binder\_state \*bs;

struct binder\_version vers;

bs = malloc(sizeof(\*bs));

bs->fd = open(driver, O\_RDWR | O\_CLOEXEC); //打开binder驱动节点并赋值给接结构体

bs->mapsize = mapsize;

//通过mmap拿到映射的地址,SM自己设置的大小为128\*1024，即128k并赋值给接结构体

bs->mapped = mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0);

return bs;

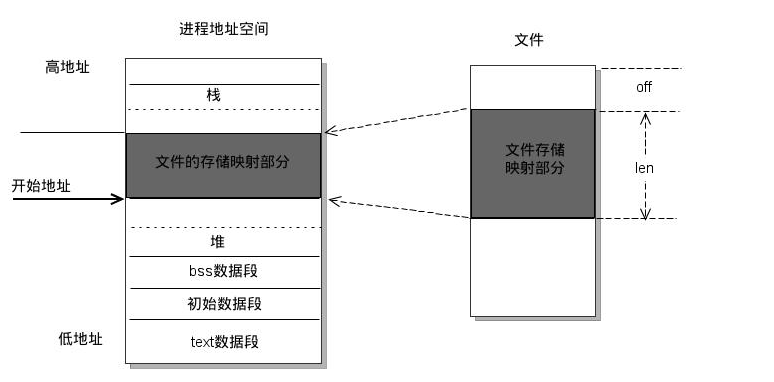
}

关于mmap:

mmap将一个文件或者其它对象映射进内存。当使用mmap映射文件到进程后，就可以直接操作这段虚拟地址进行文件的读写等操作，不必再调用read，write等系统调用,直接对该段内存写时不会写入超过当前文件大小的内容。

用户空间mmap()函数void \*mmap(void \*start, size\_t length, int prot, int flags,int fd, off\_t offset)，下面就其参数解释如下：

* start：用户进程中要映射的用户空间的起始地址，通常为NULL（由内核来指定）
* length：要映射的内存区域的大小
* prot：期望的内存保护标志
* flags：指定映射对象的类型
* fd：文件描述符（由open函数返回）
* offset：设置在内核空间中已经分配好的的内存区域中的偏移，例如文件的偏移量
* 返回值：mmap()返回被映射区的指针，该指针就是需要映射的内核空间在用户空间的虚拟地址



bs->mapped = mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0);

NULL:由binder驱动决定被映射到进程空间中内存的起始地址

Mapsize:映射的内存大小为128kb

PROT\_READ:映映射区为只读

MAP\_PRIVATE:映射区的改变是私有的,不需要保存文件

bs->fd: 文件描述符,由open函数返回

0：从文件的起始地址开始映射

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[native](http://androidxref.com/9.0.0_r3/xref/frameworks/native/)/[cmds](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/)/[servicemanager](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/)/[binder.c](http://androidxref.com/9.0.0_r3/xref/frameworks/native/cmds/servicemanager/binder.c)

int binder\_become\_context\_manager(struct binder\_state \*bs){

//发送BINDER\_SET\_CONTEXT\_MGR命令使自己成为BinderServer管家

return ioctl(bs->fd, BINDER\_SET\_CONTEXT\_MGR, 0);

}

// int ioctl(int fd, ind cmd, …);

功能简述:在计算机中ioctl(input/output control)是一个专用于设备输入输出操作的系统调用,该调用传入一个跟设备有关的请求码，系统调用的功能完全取决于请求码

fd:户程序打开设备时使用 open函数返回的文件标示符此处代表binder驱动文件描述符

cmd::户程序对设备的 控制命令,此处是通过BINDER\_SET\_CONTEXT\_MGR命令让自己成BinderServer管家

省略号是一些补充参数,一般最多一个,这个参数的有无和cmd的意义相关

binder\_loop循环等待客户端请求

//从获取ServiceManager的BinderServer出发:

/frameworks/base/core/java/android/os/ServiceManager.java

IBinder iBinder = ServiceManager .getService(String name);

/\*\* Returns a reference to a service with the given name.

\*@param name the name of the service to get

\* @return a reference to the service, or <code>null</code> if the service doesn't exist

\*/

public static IBinder getService(String name) {

//从返回的结果来看可以认为是返回了rawGetService(name)对象

return Binder.allowBlocking(rawGetService(name));

}

private static IBinder rawGetService(String name) throws RemoteException {

//这里分两步: 第一步获取ServerManager Binder Server对应的本地对象BpBinder

第二步调用发起远程调用

final IBinder binder = getIServiceManager().getService(name);

return binder;

}

private static IServiceManager getIServiceManager() {

// Find the service manager

IServiceManager sServiceManager = ServiceManagerNative

.asInterface(Binder.allowBlocking(BinderInternal.getContextObject()));

return sServiceManager;

}

/frameworks/base/core/java/com/android/internal/os/BinderInternal.java

/\*\*

\* Return the global "context object" of the system. This is usually

\* an implementation of IServiceManager, which you can use to find

\* other services.

\*/

public static final native IBinder getContextObject();

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[base](http://androidxref.com/9.0.0_r3/xref/frameworks/base/)/[core](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/)/[jni](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/jni/)/[android\_util\_Binder.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/jni/android_util_Binder.cpp)

static const JNINativeMethod gBinderInternalMethods[] = {

/\* name, signature, funcPtr \*/

{ "getContextObject", "()Landroid/os/IBinder;", (void\*)android\_os\_BinderInternal\_getContextObject }

}

//第一步获取ServiceManager BinderServcer对应的本地BpBinder

//第二步把BpBinder转成java对象返回给调用者作为真正发起远程调用的对象

static jobject android\_os\_BinderInternal\_getContextObject(JNIEnv\* env, jobject clazz){

sp<IBinder> b = ProcessState::self()->getContextObject(NULL);

return javaObjectForIBinder(env, b);

}

/frameworks/native/libs/binder/ProcessState.cpp

sp<ProcessState> ProcessState::self(){

if (gProcess != NULL) {

return gProcess;//一个进程只创建一个ProcessState

}

//创建Process对象,把binder驱动文件路径作为参数

gProcess = new ProcessState("/dev/binder");

return gProcess;

}

// ProcessState的构造方法,一个进程中只有一个ProcessState但有多个IPCThreadState

ProcessState::ProcessState(const char \*driver) : mDriverName(String8(driver)),

mDriverFD(open\_driver(driver))//打开binder驱动,并将返回的驱动文件描述符赋值给成员mDriverFD

, mVMStart(MAP\_FAILED), mThreadCountLock(PTHREAD\_MUTEX\_INITIALIZER)

, mThreadCountDecrement(PTHREAD\_COND\_INITIALIZER) , mExecutingThreadsCount(0)

, mMaxThreads(DEFAULT\_MAX\_BINDER\_THREADS) , mStarvationStartTimeMs(0)

, mManagesContexts(false), mBinderContextCheckFunc(NULL), mBinderContextUserData(NULL)

, mThreadPoolStarted(false), mThreadPoolSeq(1){

// mmap the binder, providing a chunk of virtual address space to receive transactions.

//提供一块虚拟地址去接收transaction,返回的是虚拟地址的起始位置

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

}

//获取ServiceManager的Binder Server时参数为null

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

//获取ServiceManager的Binder Server时handle值为0

return getStrongProxyForHandle(0);

}

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

sp<IBinder> result;

handle\_entry\* e = lookupHandleLocked(handle);//向列表查找handle对应的节点

if (e != NULL) {

IBinder\* b = e->binder;

if (b == NULL || !e->refs->attemptIncWeak(this)) {//节点中没有要找的对象或者对找到的Binder Server的本地对象增加应用计数失败都会重新创建一个BpBinder

if (handle == 0) {

// Special case for context manager...

// The context manager is the only object for which we create

// a BpBinder proxy without already holding a reference.

// Perform a dummy transaction to ensure the context manager

// is registered before we create the first local reference

// to it (which will occur when creating the BpBinder).

// If a local reference is created for the BpBinder when the

// context manager is not present, the driver will fail to

// provide a reference to the context manager, but the

// driver API does not return status.

Parcel data;

//从注释看,这一步的作用是为了确认在获取ServiceManager的远程BinderServer对应的本地BpBinder的时候,远程BinderServer已经注册到驱动,可以访问到BinderServer

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

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

if (status == DEAD\_OBJECT)

return NULL;

}

b = BpBinder::create(handle);//创建BpBinder,这个handle值是0

e->binder = b; ;//对handle\_entry的binder赋值为BpBinder

if (b) e->refs = b->getWeakRefs();//对handle\_entry的refs赋值

result = b;//将BpBinder作为结果返回

}

return result;

}

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

return &mHandleToObject.editItemAt(handle);

}

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[native](http://androidxref.com/9.0.0_r3/xref/frameworks/native/)/[include](http://androidxref.com/9.0.0_r3/xref/frameworks/native/include/)/[binder](http://androidxref.com/9.0.0_r3/xref/frameworks/native/include/binder/)/[ProcessState.h](http://androidxref.com/9.0.0_r3/xref/frameworks/native/include/binder/ProcessState.h)

Vector<handle\_entry> mHandleToObject;// vector列表

//列表中的数据结构是handle\_entry对象

struct handle\_entry {

IBinder\* binder;//实际上是BpBinder

RefBase::weakref\_type\* refs;对Bpbinder的应用机型进行计数的对象

};

BpBinder\* BpBinder::create(int32\_t handle) {

return new BpBinder(handle, trackedUid);//这个handle值是0

}

// 给BpBinder 的成员mHandle值赋值为0

BpBinder::BpBinder(int32\_t handle, int32\_t trackedUid): mHandle(handle) , mAlive(1) , mObitsSent(0)

, mObituaries(NULL) , mTrackedUid(trackedUid){

IPCThreadState::self()->incWeakHandle handle, this);

}

//这个函数实现了线程之间的数据隔离,类似于java层的ThreadLocal作用

IPCThreadState\* IPCThreadState::self(){

return new IPCThreadState;

}

IPCThreadState::IPCThreadState():

//这里会返回之前创建的ProcessState mProcess(ProcessState::self()),mStrictModePolicy(0),mLastTransactionBinderFlags(0){

pthread\_setspecific(gTLS, this);//线程数据隔离

clearCaller();

mIn.setDataCapacity(256);//mIn是Parcel类型的数据,用来接收Binder驱动发来的数据

mOut.setDataCapacity(256); //Out是Parcel类型的数据,用来存储发给Binder驱动的数据

}

jobject javaObjectForIBinder(JNIEnv\* env, const sp<IBinder>& val){

//val是BpBinder

BinderProxyNativeData\* nativeData = gNativeDataCache;

if (nativeData == nullptr) {

nativeData = new BinderProxyNativeData();

}

//这个函数通过JNI调用了BinderProxy的getInstance 方法并用结构体nativeData和传进来的BpBinder作为参数

jobject object = env->CallStaticObjectMethod(gBinderProxyOffsets.mClass,

gBinderProxyOffsets.mGetInstance, (jlong) nativeData, (jlong) val.get());

// New BinderProxy; we still have exclusive access.

nativeData->mOrgue = new DeathRecipientList;

//指向传进来的BpBinder对象, java层的BinderProxy类绑定C++层BpBinder类

nativeData->mObject = val;

return object;

}

// We aggregate native pointer fields for BinderProxy in a single object to allow

// management with a single NativeAllocationRegistry, and to reduce the number of JNI

// Java field accesses. This costs us some extra indirections here.

struct BinderProxyNativeData {

// Both fields are constant and not null once javaObjectForIBinder returns this as

// part of a BinderProxy.

// The native IBinder proxied by this BinderProxy.

sp<IBinder> mObject;

// Death recipients for mObject. Reference counted only because DeathRecipients

// hold a weak reference that can be temporarily promoted.

sp<DeathRecipientList> mOrgue; // Death recipients for mObject.

};

static int int\_register\_android\_os\_BinderProxy(JNIEnv\* env){

const char\* const kBinderProxyPathName = "android/os/BinderProxy";

gBinderProxyOffsets.mGetInstance = GetStaticMethodIDOrDie(env, clazz, "getInstance",

"(JJ)Landroid/os/BinderProxy;")

}

/\*\*

\* Return a BinderProxy for IBinder.

\* @param nativeData C++ pointer to (possibly still empty) BinderProxyNativeData.

\* Takes ownership of nativeData iff <result>.mNativeData == nativeData, or if

\* we exit via an exception. If neither applies, it's the callers responsibility to

\* recycle nativeData.

\* @param iBinder C++ pointer to IBinder. Does not take ownership of referenced object.

\*/

/frameworks/base/core/java/android/os/Binder.java

private static BinderProxy getInstance(long nativeData, long iBinder) {

//iBinder是BpBinder

result = sProxyMap.get(iBinder);//从缓存拿

if (result != null) {

return result;

}

result = new BinderProxy(nativeData);

//sProxyMap的 key为BpBinder,value为BinderProxy建立了对应关系

sProxyMap.set(iBinder, result);

return result;

}

/\*\*

\* C++ pointer to BinderProxyNativeData. That consists of strong pointers to the

\* native IBinder object, and a DeathRecipientList.//这个变量指向C++层的结构体BinderProxyNativeData

\*/

private final long mNativeData;

private BinderProxy(long nativeData) {

mNativeData = nativeData;//赋值

}

//回过头看:到这里只完成了BinderInternal.getContextObject()这一步,返回的是一个BinderProxy

(继承自IBinder)对应着C++层的BpBinder,内部打开了binder驱动,映射了数据传递地址空间,创建了进

行跨进程通信的ProcessState和IPCThreadState,完成了进行跨进程通信的准备

private static IServiceManager getIServiceManager() {

// Find the service manager

IServiceManager sServiceManager = ServiceManagerNative

.asInterface(Binder.allowBlocking(BinderInternal.getContextObject()));

return sServiceManager;

}

/\*\*

\* Cast a Binder object into a service manager interface, generating

\* a proxy if needed.

\*/

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[base](http://androidxref.com/9.0.0_r3/xref/frameworks/base/)/[core](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/)/[java](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/)/[android](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/)/[os](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/os/)/[ServiceManagerNative.java](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/os/ServiceManagerNative.java)

static public IServiceManager asInterface(IBinder obj){

//obj是BinderProxy,BinderProxy对应着C++层的BpBinder,这个BpBinder的handle值是0

//返回的是一个java层的代理ServiceManagerProxy

return new ServiceManagerProxy(obj);

}

public ServiceManagerProxy(IBinder remote) {

mRemote = remote;// 把传进来的BinderProxy赋值给mRemote

}

//回过头看:到这里getIServiceManager()已经走完返回的是ServiceManagerProxy对象

final IBinder binder = getIServiceManager().getService(name);

//ServiceManagerProxy对象的getService(String name)方法

public IBinder getService(String name) throws RemoteException {

Parcel.obtain();获取Parcel对象,是通过JNI获取的,指向在C++创建的Parcel对象

Parcel data = Parcel.obtain();//用于发送给binder驱动数据的对象

Parcel reply = Parcel.obtain();//创建用于接收binder驱动数据的对象

data.writeInterfaceToken(IServiceManager.descriptor);//此接口标识

data.writeString(name);//要查询的服务名称如”activity”

mRemote.transact(GET\_SERVICE\_TRANSACTION, data, reply, 0);//同步调用,调用的是BinderProxy对象的transact方法,BinderProxy是Binder的内部类, 在这一步之前都是在客户端进程的调用,这个函数中将真正发起跨进程调用.

IBinder binder = reply.readStrongBinder();//从reply中读取返回的BinderServer对象

reply.recycle();

data.recycle();

return binder;

}

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[base](http://androidxref.com/9.0.0_r3/xref/frameworks/base/)/[core](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/)/[java](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/)/[android](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/)/[os](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/os/)/[IServiceManager.java](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/os/IServiceManager.java)

static final String descriptor = "android.os.IServiceManager";

/frameworks/base/core/java/android/os/Binder.java

public boolean transact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {

…

return transactNative(code, data, reply, flags);

}

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[base](http://androidxref.com/9.0.0_r3/xref/frameworks/base/)/[core](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/)/[jni](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/jni/)/[android\_util\_Binder.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/jni/android_util_Binder.cpp)

static jboolean android\_os\_BinderProxy\_transact(JNIEnv\* env, jobject obj,

jint code, jobject dataObj, jobject replyObj, jint flags) // throws RemoteException{

// parcelForJavaObject:因为java层的Parcel是通过JNI来创建的,每一个java层的Parcel对象对应

C++层的一个Parcel对象, 这个方法是把这个java层的Parcel对象还原成其对应的C++层的Parcel 对象

/ Parcel\* data = parcelForJavaObject(env, dataObj);//

Parcel\* reply = parcelForJavaObject(env, replyObj);

//obj是BinderProxy对象,获取到BinderProxy对象的BinderProxyNativeData结构体后取出成员//mObject的值,这个值指向的是BpBinder

IBinder\* target = getBPNativeData(env, obj)->mObject.get();

//通过BpBinder发起远程调用

status\_t err = target->transact(code, \*data, reply, flags);

}

Java层Parcel类通过JNI对应的C++层类:/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[base](http://androidxref.com/9.0.0_r3/xref/frameworks/base/)/[core](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/)/[jni](http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/jni/)/android\_os\_Parcel.cpp

Java层Parcel对象对应C++层对象实现类: /[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[native](http://androidxref.com/9.0.0_r3/xref/frameworks/native/)/[libs](http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/)/[binder](http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/binder/)/[Parcel.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/binder/Parcel.cpp)

//Parcel是跨进程进行数据传递的载体,有打包的意思,可以把一个进程某个对象占据一段内存的数据打包起来,传到另外一个进程后可以复现这个这个对象

Parcel部分重要方法:

dataSize()获取当前已经存储的数据大小

dataPosition()数据的当前位置,类似于游标

setDataCapacity(int size)设置Parcel的空间大小,Parcel存储的数据不能大于这个值)

setDataPosition(int pos)改变Pracel的读写位置,必须介于0和dataSize()之间

dataAvail()当前Parcel中可读数据大小

dataCapacity()当前Parcel的存储能力

BinderProxyNativeData\* getBPNativeData(JNIEnv\* env, jobject obj) {

return (BinderProxyNativeData \*) env->GetLongField(obj, gBinderProxyOffsets.mNativeData);

}

static int int\_register\_android\_os\_BinderProxy(JNIEnv\* env){

const char\* const kBinderProxyPathName = "android/os/BinderProxy";

clazz = FindClassOrDie(env, kBinderProxyPathName);

//获取java层指向C++BinderProxyNativeData结构体的指针

gBinderProxyOffsets.mNativeData = GetFieldIDOrDie(env, clazz, "mNativeData", "J");

}

 /[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[native](http://androidxref.com/9.0.0_r3/xref/frameworks/native/)/[libs](http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/)/[binder](http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/binder/)/[BpBinder.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/binder/BpBinder.cpp)

status\_t BpBinder::transact(uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags){

//真正发起远程调用的是TPCThreadState

//这里的handle值是0,code是GET\_SERVICE\_TRANSACTION,data包含要查询参数,flags是0表示 允许回复中包含文件描述符

status\_t status = IPCThreadState::self()->transact(mHandle, code, data, reply, flags);

…

}

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[native](http://androidxref.com/9.0.0_r3/xref/frameworks/native/)/[libs](http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/)/[binder](http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/binder/)/[IPCThreadState.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/binder/IPCThreadState.cpp)

status\_t IPCThreadState::transact(int32\_t handle, uint32\_t code, const Parcel& data,

Parcel\* reply, uint32\_t flags){

status\_t err;

//将数据打包成binder可以接受的数据格式

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

err = waitForResponse(reply);//这个函数将与Binder驱动进行真正的交互

return err;

}

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;//与binder驱动交互的结构体

tr.target.ptr = 0; /\* Don't pass uninitialized stack data to a remote process \*/

tr.target.handle = handle;

tr.code = code;

tr.flags = binderFlags;

tr.cookie = 0;

tr.sender\_pid = 0;

tr.sender\_euid = 0;

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

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

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

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

// mOut是Parcel类型的变量是java层通过JNI传进来的,用来存储与Binder驱动交互的数据后面还会进行一次包装

mOut.writeInt32(cmd);//写入命令BC\_TRANSACTION

mOut.write(&tr, sizeof(tr));/ /写入数据

return NO\_ERROR;

}

struct binder\_transaction\_data {

union {

\_\_u32 handle; //binder\_ref（即handle）

binder\_uintptr\_t ptr; //Binder\_node的内存地址

} target; //RPC目标

binder\_uintptr\_t cookie; //BBinder指针

\_\_u32 code; //远程调用代码，代表Client与Server双方约定的命令码

\_\_u32 flags; //标志位，比如TF\_ONE\_WAY代表异步,即不等待Server端回复

pid\_t sender\_pid; //发送端进程的pid

uid\_t sender\_euid; //发送端进程的uid

binder\_size\_t data\_size; //data数据的总大小

binder\_size\_t offsets\_size; //IPC对象的大小

union {

struct {

binder\_uintptr\_t buffer; //数据区起始地址

binder\_uintptr\_t offsets; //数据区IPC对象偏移量

} ptr;

\_\_u8 buf[8];

} data; //RPC数据

};

//将对这函数进行拆分分析

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

int32\_t err;

while (1) {

if ((err=talkWithDriver()) < NO\_ERROR) break;

cmd = (uint32\_t)mIn.readInt32()//关于Binder驱动返回的cmd的处理下次再说

…

return err;

}

status\_t IPCThreadState::talkWithDriver(bool doReceive){

binder\_write\_read bwr;//binder驱动规定格式数据,读写都是这个格式,会对之前的数据进一步打包

// Is the read buffer empty?

//mIn有上一轮IO读出尚未解析的数据,数据位置值大于Parcel中已经存储的数据,因此needRead=true

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

// We don't want to write anything if we are still reading

// from data left in the input buffer and the caller

// has requested to read the next data.//如果正在读就不去写

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

bwr.write\_size = outAvail;

//把要发送的数据赋值给bwr的字段write\_buffer,写入binder驱动的数据都会在这个字段里面

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

// This is what we'll read.

if (doReceive && needRead) {// doReceive默认为true,需要读取出数据的话就去读Binder返回的数据

bwr.read\_size = mIn.dataCapacity();//大小为256

bwr.read\_buffer = (uintptr\_t)mIn.data();//Binder驱动返回的数据都存在mIn.data中并赋值给.read\_buffer这个字段

} else {

bwr.read\_size = 0;

bwr.read\_buffer = 0;//没有要读的数据

}

// Return immediately if there is nothing to do.没有发生数据的读写就直接返回继续循环

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

bwr.write\_consumed = 0;

bwr.read\_consumed = 0;

status\_t err;

do {

if (ioctl(mProcess->mDriverFD, BINDER\_WRITE\_READ, &bwr) >= 0)//真正与驱动交互的函数

err = NO\_ERROR;

} while (err == -EINTR);

if (err >= NO\_ERROR) {

if (bwr.write\_consumed > 0) {

//如果write\_consumed大于0说明Binder驱动消耗了写入的数据,如果消耗的数据量小于总的

//写入数据量就移除掉消耗的这部分数据,否则就是驱动消耗了所有的数据,就把mOut的数据量

//设置为0,这一步是为了让客户端进程的数据与驱动那边的数据保持同步状态,就像两个碗,一个

//碗A里面有水,一个碗B里面没有水,从A往B里面倒水,B增加多少A就会少多少.

if (bwr.write\_consumed < mOut.dataSize())

mOut.remove(0, bwr.write\_consumed);

else {

mOut.setDataSize(0);

processPostWriteDerefs();

}

}

//如果read\_consumed大于0说明Binder驱动读取到了数据,把数据写入到mInt中

if (bwr.read\_consumed > 0) {

mIn.setDataSize(bwr.read\_consumed);

mIn.setDataPosition(0);

return NO\_ERROR;

}

return err;

}

/external/kernel-headers/original/uapi/linux/android/binder.h

struct binder\_write\_read {

/\*\* 输入数据 从用户空间传输到Binder驱动程序的数据

\* 数据协议代码为命令协议码，由binder\_driver\_command\_protocol定义

\*/

// 写入的大小

binder\_size\_t write\_size; /\* bytes to write \*/

// 记录了驱动从缓冲区已经取出的数据量

binder\_size\_t write\_consumed; /\* bytes consumed by driver \*/

// 指向一个用户空间缓冲区的地址，里面的内容即为输入数据，大小由write\_size指定

binder\_uintptr\_t write\_buffer;

/\*\* 输出数据,从Binder驱动程序,返回给用户空间的数据

\* 数据协议代码为返回协议代码,由binder\_driver\_return\_protocol定义

\*/

//读出的大小

binder\_size\_t read\_size; /\* bytes to read \*/

// read\_buffer中已经读取的数据量

binder\_size\_t read\_consumed; /\* bytes consumed by driver \*/

// 指向一个用户缓冲区一个地址，里面保存输出的数据

binder\_uintptr\_t read\_buffer;

}

数据结构图:

