Android Binder 机制

学号　　1501210451

姓名　　张志康

专业　　集成电路工程

学苑　　科技四苑

2015 年 12 月 5 日

目录

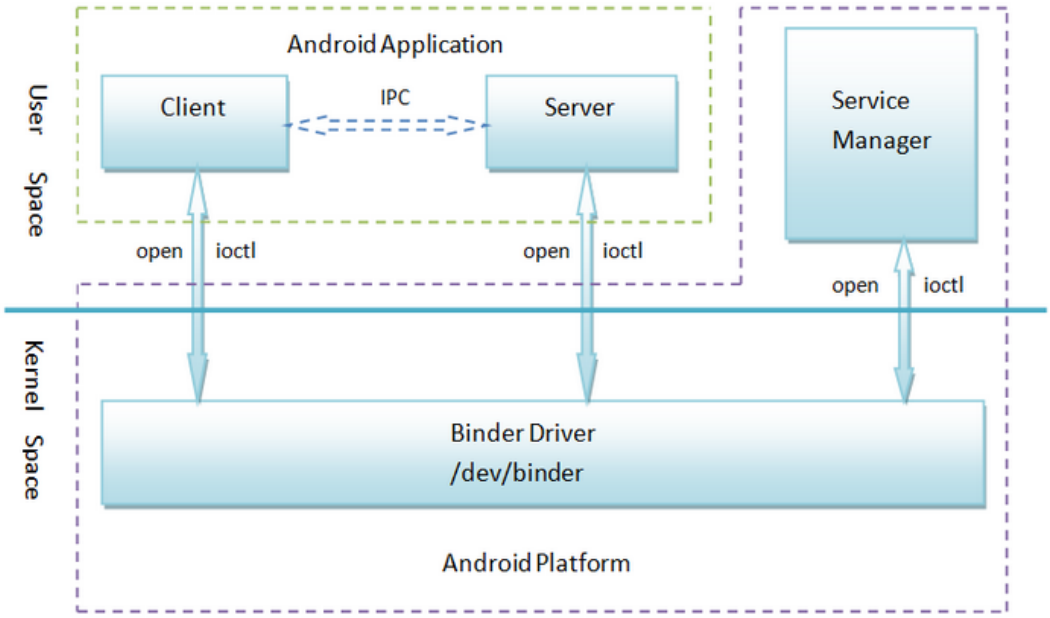
本文主要分析 native 层和 Java 层的 Android binder 通信机制。

binder 是 Android 最为常见的进程通信机制之一，其驱动和通信库是 binder 的核心，分别由 C 和 C++编写，应用程序通过 JNI 同底层库进行关联，也就是 native 层驱动和通信库通过 Java 层包装后被 Java 层调用。

一、native 层整体通信流程

1.通信流程概要

在探究 binder 通信流程之前，首先我们需要了解 Binder 机制的四个组件：Client、Server、Service Manager 和 Binder 驱动程序。关系如图：



应用程序最终目的是完成 Client 组件和 Server 组件之间的通信。ServiceManger 对于大家而言是一个公共接入点，0 便是 ServiceManger 的句柄值。

从表面看通信建立的流程便是注册和获取的过程：

1、client 通过参数（Parcel 包）传递进行通信请求；

2、在收到通信请求时，Server 组件需要通过 0 这个句柄值访问 ServiceManger，在 ServiceManger 中注册一个 binder 实体。并关联一个字符串；

3、Client 组件通过 0 这个标识去访问 ServiceManger，通过一个字符去查询 Server 组件的引用，此 ServiceManger 将 Server 注册的 binder 实体的一个引用传递给 Client 端，此时 client 便可根据这个引用同 server 进行通信了。

由以上可知，在收到请求时 server 将一个 binder 实体传递给 C 进程，而 client 得到的只是 binder 的一个引用，进而调用 binder 实体的函数。BpBinder 和 BBinder 分别代表 binder 的引用和实体，它们均继承自 IBinder 类。

在描述具体流程之前我们先来了解 binder 通信中需要用到的三个主要基类：

1.基类 IInterface：

为 server 端提供接口，它的子类声明了 service 能够实现的所有的方法；

2.基类 IBinder

BBinder 与 BpBinder 均为 IBinder 的子类，因此可以看出 IBinder 定义了 binder IPC 的通信协议，BBinder 与 BpBinder 在这个协议框架内进行的收和发操作，构建了基本的 binder IPC 机制。

3.基类 BpRefBase

client 端在查询 SM 获得所需的的 BpBinder 后，BpRefBase 负责管理当前获得的 BpBinder 实例。

2.ServiceManger

首先我们来了解一下在通信流程中 ServiceManger 所做的工作。

ServiceManger 是一个 linux 级进程，是一个 service 管理器（service 向 SM 注册是，service 就是一个 client，而 ServiceManger 便是 server），即我们前边提到的：每一个 service 被使用之前，均要向 ServiceManger 注册，客户端通过查询 ServiceManger 是否存在此服务来获取 service 的 handle（标识符）。

ServiceManger 入口函数为：service_manager.c

位于：/frameworks/base/cmds/servicemanager/

```
270int main(int argc, char **argv)
271{
272    struct binder_state *bs;
273    void *svcmgr = BINDER_SERVICE_MANAGER;
274
275    bs = binder_open(128*1024);
276
277    if (binder_become_context_manager(bs)) {
278        ALOGE("cannot become context manager (%s)\n",
strerror(errno));
279        return -1;
280    }
281
```

```
282     svcmgr_handle = svcmgr;
283     binder_loop(bs, svcmgr_handler);
284     return 0;
285}
```

主要工作：

1. 初始化 binder，打开/dev/binder 设备，在内存中为 binder 映射 128Kb 空间。

bs = binder_open(128*1024);

其中 binder_open 位于 binder.c 中，源代码为：

```
94struct binder_state *binder_open(unsigned mapsize)
95{
96     struct binder_state *bs;
97
98     bs = malloc(sizeof(*bs));
99     if (!bs) {
100         errno = ENOMEM;
101         return 0;
102     }
103
104     bs->fd = open("/dev/binder", O_RDWR);
105     ......
127     return 0;
128}
```

2. 指定 SM 对于代理 binder 的 handle 为 0，即 client 尝试同 SM 通信时创建一个 handle 为 0 的代理 binder。

void *svcmgr = BINDER_SERVICE_MANAGER;

svcmgr_handle = svcmgr;

其中 BINDER_SERVICE_MANAGER 在 binder.h 中被指定为 0：

#define BINDER_SERVICE_MANAGER ((void*) 0)

3. 通知 binder driver(BD)使 SM 成为 BD 的 context manager；

if (binder_become_context_manager(bs)) {

```
        LOGE("cannot become context manager (%s)/n", strerror(errno));
        return -1;
    }
```

binder_become_context_manager(bs)源码位于 binder.c 中：

```
int binder_become_context_manager(struct binder_state *bs)
{
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

4.进入一个死循环，不断读取内核的 binder driver，查看是否有对 service 的操作请求，如果有调用 svcmgr_handler 来处理请求操作：

binder_loop(bs, svcmgr_handler);

binder_loop(,)源码位于 binder.c 中：

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
    int res;
    struct binder_write_read bwr;
    unsigned readbuf[32];
    ……
    }
}
```

5.维护一个 svclist 列表来存储 service 的信息。

源码位于 service_manager.c：

```
int svcmgr_handler(struct binder_state *bs,
                   struct binder_txn *txn,
                   struct binder_io *msg,
                   struct binder_io *reply)
{
    struct svcinfo *si;
    ……
}
```

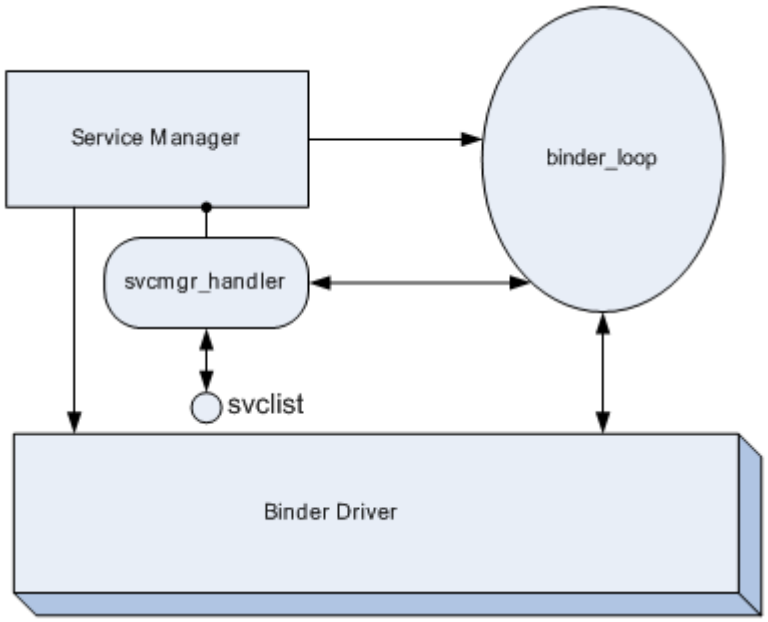


Figure 1 Service Manager 工作流程

3.ProcessState

ProcessState 是每个进程在使用 Binder 通信时都需要维护的，用来描述当前进程的 binder 状态。

ProcessState 主要完成两个功能：

1.创建一个 thread 负责与内核中的 binder 模块进行通信（Poolthread）。

在 Binder IPC 中，所有进程均会启动一个 thread 来负责与 binder 来直接通信，也就是不断读写 binder，这个线程主体是一个 IPCThreadState 对象（具体介绍见第 4 节）。

Poolthread 启动方式：ProcessState::self()->startThreadPool();

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

```
136void ProcessState::startThreadPool()
137{
138    AutoMutex _l(mLock);
139    if (!mThreadPoolStarted) {
140        mThreadPoolStarted = true;
141        spawnPooledThread(true);
142    }
143}
```

2.为知道的 handle 创建一个 BpBinder 对象，并管理进程中所有的 BpBinder 对象。

BpBinder 在第一节已经提到，其主要功能是负责 client 向 BD 发送调用请求的数据，是 client 端 binder 通信的核心，通过调用 transact 向 BD 发送调用请求的数据。

ProcessState 通过如下函数获取 BpBinder 对象：

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

```
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
90{
91    return getStrongProxyForHandle(0);
92}


sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
184{
185    sp<IBinder> result;
186
187    AutoMutex _l(mLock);
188
189    handle_entry* e = lookupHandleLocked(handle);
            ……
210    return result;
211}


ProcessState::handle_entry* ProcessState::lookupHandleLocked(int32_t
handle)
171{
172    const size_t N=mHandleToObject.size();
173    if (N <= (size_t)handle) {
174        handle_entry e;
175        e.binder = NULL;
176        e.refs = NULL;
```

```
177         status_t err = mHandleToObject.insertAt(e, N, handle+1-N);
178         if (err < NO_ERROR) return NULL;
179     }
180     return &mHandleToObject.editItemAt(handle);
181}
```

在获取 BpBinder 对象的过程中，ProcessState 会维护一个 BpBinder 的 vecto：mHandleToObject(具体调用过程见上述源代码)。

创建一个 BpBinder 实例时，回去查询 mHandleToObject，如果对应的 handler 以及有 binder 指针，就不再创建，否则创建并插入到 mHandlerToObject 中（具体代码见上述的 lookupHandleLocked）。

BpBinder 构造函数位于/frameworks/native/libs/binder/BpBinder.cpp：

```
BpBinder::BpBinder(int32_t handle)
90     : mHandle(handle)
91     , mAlive(1)
92     , mObitsSent(0)
93     , mObituaries(NULL)
94{
95     ALOGV("Creating BpBinder %p handle %d\n", this, mHandle);
96
97     extendObjectLifetime(OBJECT_LIFETIME_WEAK);
98     IPCThreadState::self()->incWeakHandle(handle);
99}
```

通过此构造函数我们可以发现：BpBinder 会将通信中 server 的 handle 记录下来。当有数据发送时，会把数据的发送目标通知 BD。

4.IPCThreadState

IPCThreadState 也是一个单例模式，由上边我们已知每个进程维护一个 ProcessState 实例，且 ProcessState 只启动一个 Pool thread，因此一个进程之后启动一个 Pool thread。

IPCThreadState 实际内容为：

```
void IPCThreadState::joinThreadPool(bool isMain)
```

```
421{
422    LOG_THREADPOOL("**** THREAD %p (PID %d) IS JOINING THE THREAD POOL\n",
(void*)pthread_self(), getpid());
423
424    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
429    set_sched_policy(mMyThreadId, SP_FOREGROUND);
431    status_t result;
432    do {
433        int32_t cmd;
436        if (mIn.dataPosition() >= mIn.dataSize()) {
437            size_t numPending = mPendingWeakDerefs.size();
438            if (numPending > 0) {
439                for (size_t i = 0; i < numPending; i++) {
440                    RefBase::weakref_type* refs = mPendingWeakDerefs[i];
441                    refs->decWeak(mProcess.get());
442                }
443                mPendingWeakDerefs.clear();
444            }
446            numPending = mPendingStrongDerefs.size();
447            if (numPending > 0) {
448                for (size_t i = 0; i < numPending; i++) {
449                    BBinder* obj = mPendingStrongDerefs[i];
450                    obj->decStrong(mProcess.get());
451                }
452                mPendingStrongDerefs.clear();
453            }
454        }
457        result = talkWithDriver();
458        if (result >= NO_ERROR) {
459            size_t IN = mIn.dataAvail();
```

```
460             if (IN < sizeof(int32_t)) continue;
461             cmd = mIn.readInt32();
462             IF_LOG_COMMANDS() {
463                 alog << "Processing top-level Command: "
464                     << getReturnString(cmd) << endl;
465             }
468             result = executeCommand(cmd);
469         }
482         if(result == TIMED_OUT && !isMain) {
483             break;
484         }
485     } while (result != -ECONNREFUSED && result != -EBADF);
486
487     LOG_THREADPOOL("**** THREAD %p (PID %d) IS LEAVING THE THREAD POOL
err=%p\n",
488         (void*)pthread_self(), getpid(), (void*)result);
489
490     mOut.writeInt32(BC_EXIT_LOOPER);
491     talkWithDriver(false);
492}
```

ProcessState 中有 2 个 Parcel 成员（mIn 和 mOut），由以上代码可见，Pool Thread 会不断查询 BD 中是否有数据可读，若有，则保存在 mIn；不停检查 mOut 是否有数据需要向 BD 发送，若有，则写入 BD。

根据第三节提到的：BpBinder 通过调用 transact 向 BD 发送调用请求的数据，也就是说 ProcessState 中生成的 BpBinder 实例通过调用 IPCThreadState 的 transact 函数来向 mOut 中写入数据，这样的话这个 binder IPC 过程的 client 端的调用请求的发送过程就讲述完毕。

IPCThreadState 有两个重要的函数，talkWithDriver 函数负责从 BD 读写数据，executeCommand 函数负责解析并执行 mIn 中的数据。

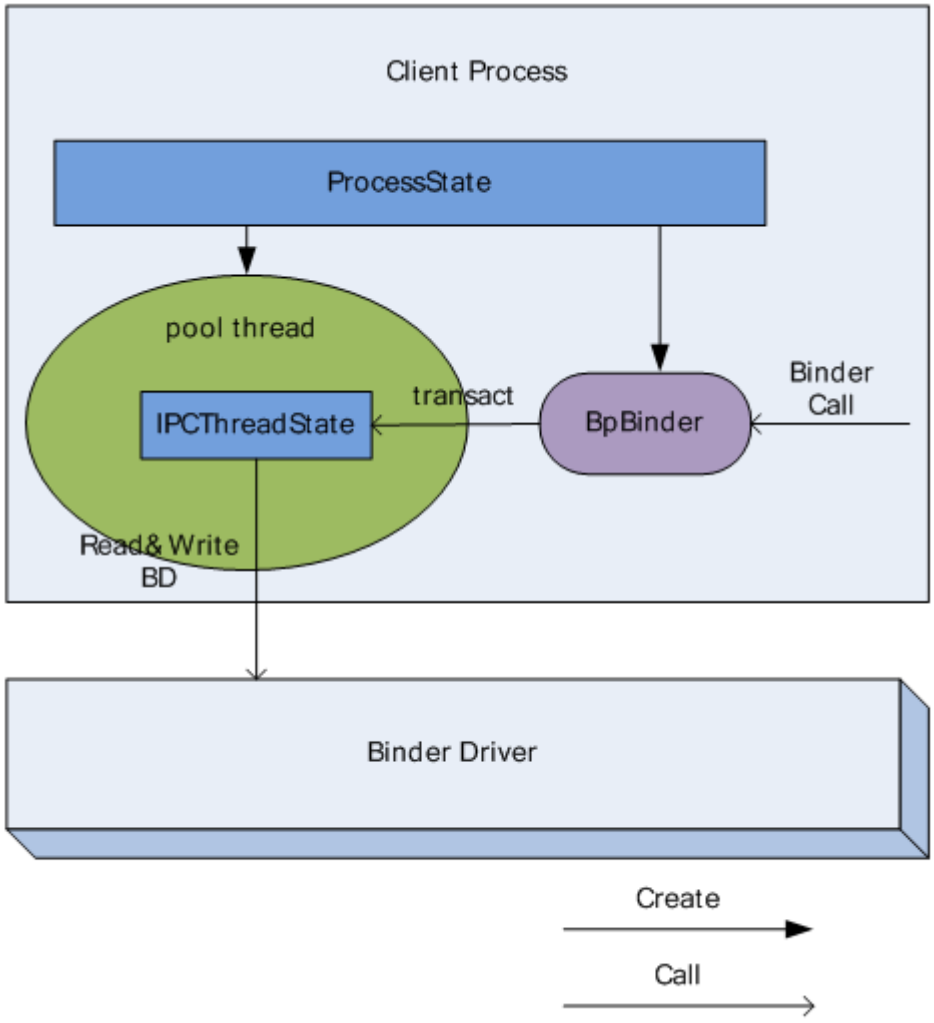


Figure 2 Client 端 binder 工作流程

5.两个接口类

1.BpINTERFACE

client 在获得 server 端 service 时，server 端向 client 提供一个接口，client 在这个接口基础上创建一个 BpINTERFACE，使用此对象，client 端的应用能够像本地调用一样直接调用 server 端的方法，而不必关系 binder IPC 实现。

BpINTERFACE 原型如下：

```
/frameworks/native/include/binder/IInterface.h
62template<typename INTERFACE>
63class BpInterface : public INTERFACE, public BpRefBase
64{
65public:
```

```
66                                  BpInterface(const sp<IBinder>& remote);
67
68protected:
69      virtual IBinder*            onAsBinder();
70};
```

可见，BpINTERFACE 继承自 INTERFACE、BpRefBase。

BpINTERFACE 既实现了 service 中各方法的本地操作，将每个方法的参数以 Parcel 的形式发送给 BD。同时又将 BpBinder 作为了自己的成员来管理，将 BpBinder 存储在 mRemote 中，BpServiceManager 通过调用 BpRefBase 的 remote()来获得 BpBinder 指针。

2. BnINTERFACE

同样位于/frameworks/native/include/binder/IInterface.h

```
49template<typename INTERFACE>
50class BnInterface : public INTERFACE, public BBinder
51{
52public:
53      virtual sp<IInterface>      queryLocalInterface(const String16&
 _descriptor);
54      virtual const String16&     getInterfaceDescriptor() const;
55
56protected:
57      virtual IBinder*            onAsBinder();
58};
```

由代码可知，BnInterface 继承自 INTERFACE、BBinder。

class BBinder：public IBinder，由此可见，server 端的 binder 操作及状态维护是通过 BBinder 来实现的。BBinder 即为 binder 的本质。

3.接口类总结

由上节的描述及刚才对于两个接口类源代码分析可知：BpBinder 是 client 端用于创建消息发送的机制，而 BBinder 是 server 端用于接口消息的通道。

BpBinder 是 client 创建的用于消息发送的代理，其 transact 函数用于向 IPCThreadState 发送消息，通知其有消息要发送给 BD，部分源代码如下：

/frameworks/native/libs/binder/BpBinder.cpp

```
status_t BpBinder::transact(
```

```
160     uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
161{
163     if (mAlive) {
164         status_t status = IPCThreadState::self()->transact(
165             mHandle, code, data, reply, flags);
166         if (status == DEAD_OBJECT) mAlive = 0;
167         return status;
168     }
170     return DEAD_OBJECT;
207         }
209         default:
210             return UNKNOWN_TRANSACTION;
211     }
212}
```

由 BBinder 的源码可知，其作用是当 IPCThreadState 收到 BD 消息时，通过 transact 方法将其传递给它的子类 BnSERVICE 的 onTransact 函数执行 server 端的操作。部分源码如下：

/frameworks/native/libs/binder/Binder.cpp

```
        status_t BBinder::transact(
98    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
99{
100     data.setDataPosition(0);
102     status_t err = NO_ERROR;
103     switch (code) {
104         case PING_TRANSACTION:
105             reply->writeInt32(pingBinder());
106             break;
107         default:
108             err = onTransact(code, data, reply, flags);
109             break;
```

```
110     }
112     if (reply != NULL) {
113         reply->setDataPosition(0);
114     }
116     return err;
117}
```

由上述可知，BpINTERFACE，BnINTERFACE 均来自同一接口类 IINTERFACE，由此保证了 service 方法在 C/S 两端的一致性。

6.writeStrongBinder 和 readStrongBinder

1. writeStrongBinder 是 client 将一个 binder 传送给 server 时需要调用的函数。

具体源码如下：

```
status_t Parcel::writeStrongBinder(const sp<IBinder>& val)
681{
682     return flatten_binder(ProcessState::self(), val, this);
683}
```

flatten_binder 为：

```
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();
        if (!local) {
            BpBinder *proxy = binder->remoteBinder();
            if (proxy == NULL) {
                LOGE("null proxy");
            }
```

```
            const int32_t handle = proxy ? proxy->handle() : 0;
            obj.type = BINDER_TYPE_HANDLE;
            obj.handle = handle;
            obj.cookie = NULL;
        } else {
            obj.type = BINDER_TYPE_BINDER;
            obj.binder = local->getWeakRefs();
            obj.cookie = local;
        }
    } else {
        obj.type = BINDER_TYPE_BINDER;
        obj.binder = NULL;
        obj.cookie = NULL;
    }
    return finish_flatten_binder(binder, obj, out);
}
```

下边举例说明，addService 源码为：

/frameworks/native/libs/binder/IServiceManager.cpp

```
    virtual status_t addService(const String16& name, const sp<IBinder>&
service,
155             bool allowIsolated)
156     {
157         Parcel data, reply;
158
data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
159         data.writeString16(name);
160         data.writeStrongBinder(service);
161         data.writeInt32(allowIsolated ? 1 : 0);
162         status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data,
&reply);
163         return err == NO_ERROR ? reply.readExceptionCode() : err;
```

```
164     }
```

由上述代码块可知，写入到 parcel 的 binder 类型为 BINDER_TYPE_BINDER，然而 SM 收到的 Service 的 binder 类型必须为 BINDER_TYPE_HANDLE 才会将其添加到 svclist 中，因此说，addService 开始传递的 binder 类型为 BINDER_TYPE_BINDER 然而 SM 收到的 binder 类型为 BINDER_TYPE_HANDLE，中间经历了一个改变，代码如下：

drivers/staging/android/Binder.c

```
static void binder_transaction(struct binder_proc *proc,
                               struct binder_thread *thread,
                               struct binder_transaction_data *tr, int reply){
……
 if (fp->type == BINDER_TYPE_BINDER)
        fp->type = BINDER_TYPE_HANDLE;
 else
        fp->type = BINDER_TYPE_WEAK_HANDLE;
fp->handle = ref->desc;
……
}
```

由以上函数可知，SM 只保存了 Service binder 的 handle 和 name，当 client 需要和 Service 通信时，如何才能获得 Service 得 binder 呢？需要由 readStrongBinder 来完成。

2. readStrongBinder

Client 向 server 请求时，server 向 BD 发送一个 binder 返回给 SM(保存 handle 和 name)，当 IPCThreadState 收到由返回的 parcel 时，client 通过这一函数将这个 server 返回给 SM 的 binder 读出。

源码为：

/frameworks/native/libs/binder/Parcel.cpp

```
sp<IBinder> Parcel::readStrongBinder() const
1041{
1042    sp<IBinder> val;
1043    unflatten_binder(ProcessState::self(), *this, &val);
1044    return val;
```

```
1045}
```

unflatten_binder 为：

```
status_t unflatten_binder(const sp<ProcessState>& proc,
237     const Parcel& in, sp<IBinder>* out)
238{
239     const flat_binder_object* flat = in.readObject(false);
240
241     if (flat) {
242         switch (flat->type) {
243             case BINDER_TYPE_BINDER:
244                 *out = static_cast<IBinder*>(flat->cookie);
245                 return finish_unflatten_binder(NULL, *flat, in);
246             case BINDER_TYPE_HANDLE:
247                 *out = proc->getStrongProxyForHandle(flat->handle);
248                 return finish_unflatten_binder(
249                     static_cast<BpBinder*>(out->get()), *flat, in);
250         }
251     }
252     return BAD_TYPE;
253}
```

由如上源码可知：发现如果 server 返回的 binder 类型为 BINDER_TYPE_BINDER 的话，直接获取这个 binder；如果 server 返回的 binder 类型为 BINDER_TYPE_HANDLE 时，那么需要重新创建一个 BpBinder 返回给 client。Client 通过获得 SMhandle 来重新构建代理 binder 与 server 进行通信。

至此，native 通信机制已构建完毕。

二、Java 层的 binder 机制

下边来解析一下 java 层对于 binder 的封装过程，分四部分来进行介绍：Java 层 ServiceManager 的结构、如何注册一个 Service、如何得到一个 Service、Service 代理对象方法的过程。

1. ServiceManager 的结构

在 Java 层，ServiceManager 的函数源码为：

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

```
    public final class ServiceManager {
36      }
49  public static IBinder getService(String name) {
50
61  }
62
70  public static void addService(String name, IBinder service) {
71
76  }
77
87  public static void addService(String name, IBinder service, boolean
allowIsolated) {
93  }
94
99  public static IBinder checkService(String name) {
111  }
112
116  public static String[] listServices() throws RemoteException {
117
123  }
124
133  public static void initServiceCache(Map<String, IBinder> cache) {
134
138  }
```

由源码可知，ServiceManager 没有继承其他类，下边我们来分析 ServiceManager 管理 binder 通信的流程。

2.在 Java 层注册 Service

通过 ServiceManager 的 addService()可注册自己，其传输了两个参数：String name, IBinder service，分别为 name 和 BBinder 的子类对象，跟 native 层 ServiceManager 中 Service 的注册方法相一致。

具体源码如下：

```
    public static void addService(String name, IBinder service) {
71      try {
72          getIServiceManager().addService(name, service, false);
73      } catch (RemoteException e) {
74          Log.e(TAG, "error in addService", e);
75      }
76  }
```

getIServiceManager().addService 表明将此操作请求转发给了 getIServiceManager()，返回一个 IServiceManger 类型的 sServiceManager 对象，源码如下：

```
    private static IServiceManager getIServiceManager() {
34      if (sServiceManager != null) {
35          return sServiceManager;
36      }
39      sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());
40      return sServiceManager;
41  }
```

BinderInternal.getContextObject 在 native 层得到 BpBinder 对象。

ServiceManagerNative.asInterface 将 BpBinder 封装为 Java 层可用的 ServiceManagerProxy 对象。

下面来通过源码具体分析 BpBinder 封装为 ServiceManagerProxy 的过程：

```
    static public IServiceManager asInterface(IBinder obj)
34  {
35      if (obj == null) {
36          return null;
```

```
37          }
38          IServiceManager in =
39              (IServiceManager)obj.queryLocalInterface(descriptor);
40          if (in != null) {
41              return in;
42          }
43
44          return new ServiceManagerProxy(obj);
45      }
```

由源码可知，通过 asInterface 的转换，BpBinder 对象生成了 ServiceManagerProxy 对象。也就是说 getIServiceManager()得到的是一个 ServiceManagerProxy 对象，那么 ServiceManagerProxy 又是什么，下边来具体分析一下。

```
class ServiceManagerProxy implements IServiceManager {
110     public ServiceManagerProxy(IBinder remote) {
111         mRemote = remote;
112     }
114     public IBinder asBinder() {
115         return mRemote;
116     }
118     public IBinder getService(String name) throws RemoteException {
119
128     }
130     public IBinder checkService(String name) throws RemoteException {
131
140     }
142     public void addService(String name, IBinder service, boolean allowIsolated)
143             throws RemoteException {
144
153     }
155     public String[] listServices() throws RemoteException {
```

```
156
182     }
184     public void setPermissionController(IPermissionController controller)
185             throws RemoteException {
186
193     }
195     private IBinder mRemote;
196}
```

由源码可知，ServiceManagerProxy 继承自 IServiceManager，提供 add、get、list、check 等方法。由以上分析可知，通过 getIServiceManager 的便可得到 ServiceManagerProxy 对象，调用其 addService 方法便可进行注册，addService 源码如下：

```
public void addService(String name, IBinder service, boolean allowIsolated)
143             throws RemoteException {
144         Parcel data = Parcel.obtain();
145         Parcel reply = Parcel.obtain();
146         data.writeInterfaceToken(IServiceManager.descriptor);
147         data.writeString(name);
148         data.writeStrongBinder(service);
149         data.writeInt(allowIsolated ? 1 : 0);
150         mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0);
151         reply.recycle();
152         data.recycle();
153     }
```

可知，将 name 和 Service 对象封装到 Parcel 中，调用 transact()方法送出，并将当前操作标记为 ADD_SERVICE_TRANSACTION，根据上一章提到的内容，transact()便会调用到 BpBinder 中，此时便进入到 native 层的使用，这部分内容已经在上一章节分析完毕，具体流程图如下：

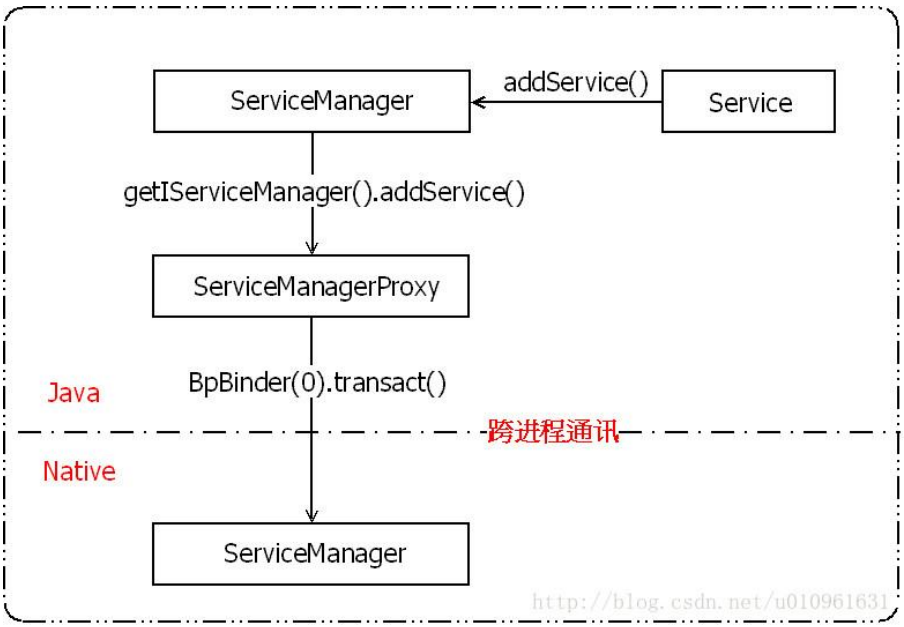


Figure 3 Java 层注册 Service 流程

3.客户端得到一个 Service

主要流程如下：通过 Java 层的 ServerManager 得到相应的 Service，然后通过 asInterface()将得到的对象转为客户端可直接调用的代理对象，然后调用代理对象的 updateAdnRecordsEfBySearch()方法。

具体分析如下：

首先，通过 ServerManager 得到相应的 BpBinder 对象。

源码位于 ServerManager.java 中

```
public static IBinder getService(String name) {
50          try {
51              IBinder service = sCache.get(name);
52              if (service != null) {
53                  return service;
54              } else {
55                  return getIServiceManager().getService(name);
56              }
57          } catch (RemoteException e) {
58              Log.e(TAG, "error in getService", e);
59          }
```

```
60          return null;
61      }
```

可见，调用 getIServiceManager()对象的 getService()方法，代码如下。

```
private static IServiceManager getIServiceManager() {
34          if (sServiceManager != null) {
35              return sServiceManager;
36          }
37
38          // Find the service manager
39          sServiceManager =
ServiceManagerNative.asInterface(BinderInternal.getContextObject());
40          return sServiceManager;
41      }
```

可知通过 IServiceManager 得到的是一个 ServiceManager 在 Java 层的代理对象，下边来分析此代理对象的 getService()方法。

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

```
public IBinder getService(String name) throws RemoteException {
119         Parcel data = Parcel.obtain();
120         Parcel reply = Parcel.obtain();
121         data.writeInterfaceToken(IServiceManager.descriptor);
122         data.writeString(name);
123         mRemote.transact(GET_SERVICE_TRANSACTION, data, reply, 0);
124         IBinder binder = reply.readStrongBinder();
125         reply.recycle();
126         data.recycle();
127         return binder;
128     }
```

可见，getService 请求被转交给 native 层，由上一章分析可知，native 层得到请求后会将目标 Service 的 BpBinder 返回给客户端，得到 BpBinder 对象后，通过 asInterface()得到一个 Proxy 对象，客户端便通过这个代理类调用服务端定义的各种方法。具体客户端得到 Service 的流程图如下：

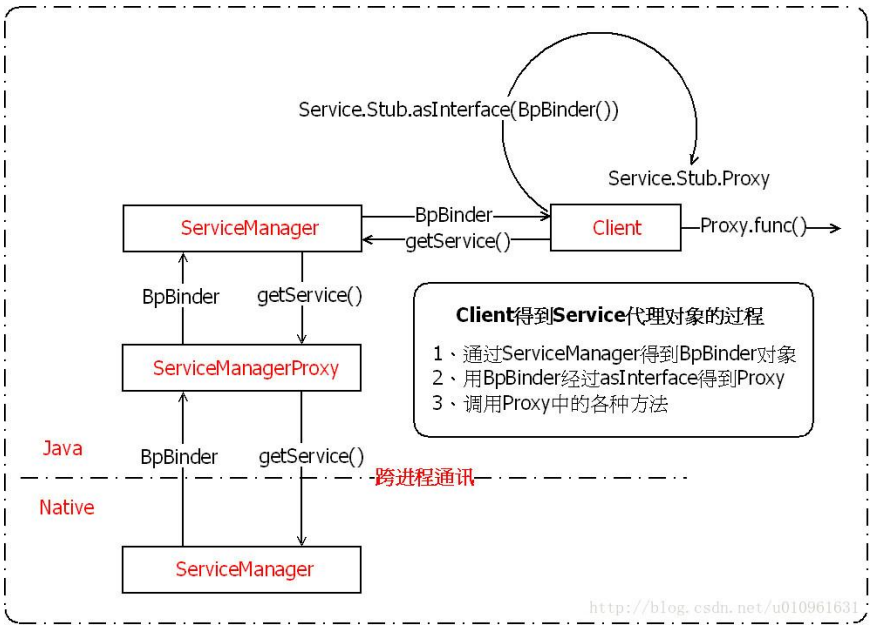


Figure 4 Java 层客户端得到 Service 流程

三、总结

Binder 通信整体流程图如下：

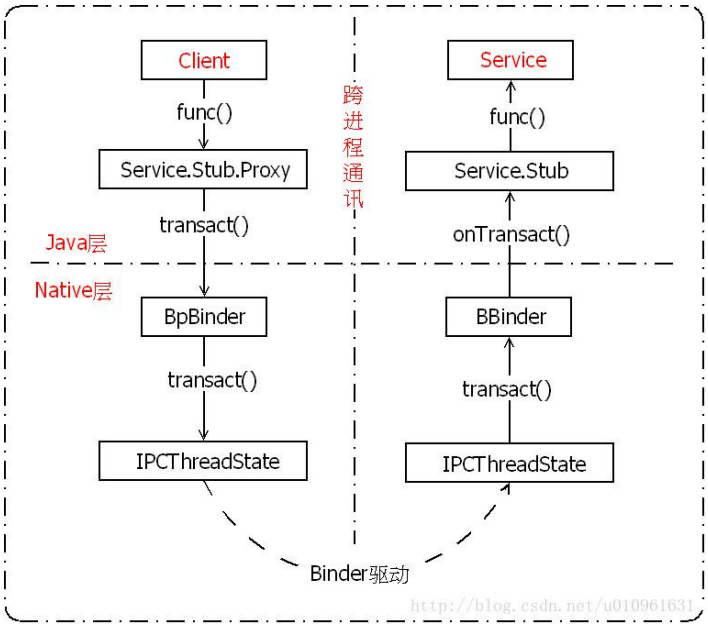


Figure 5 Binder 通信流程图