Binder系列6—framework层分析

Nov 21, 2015

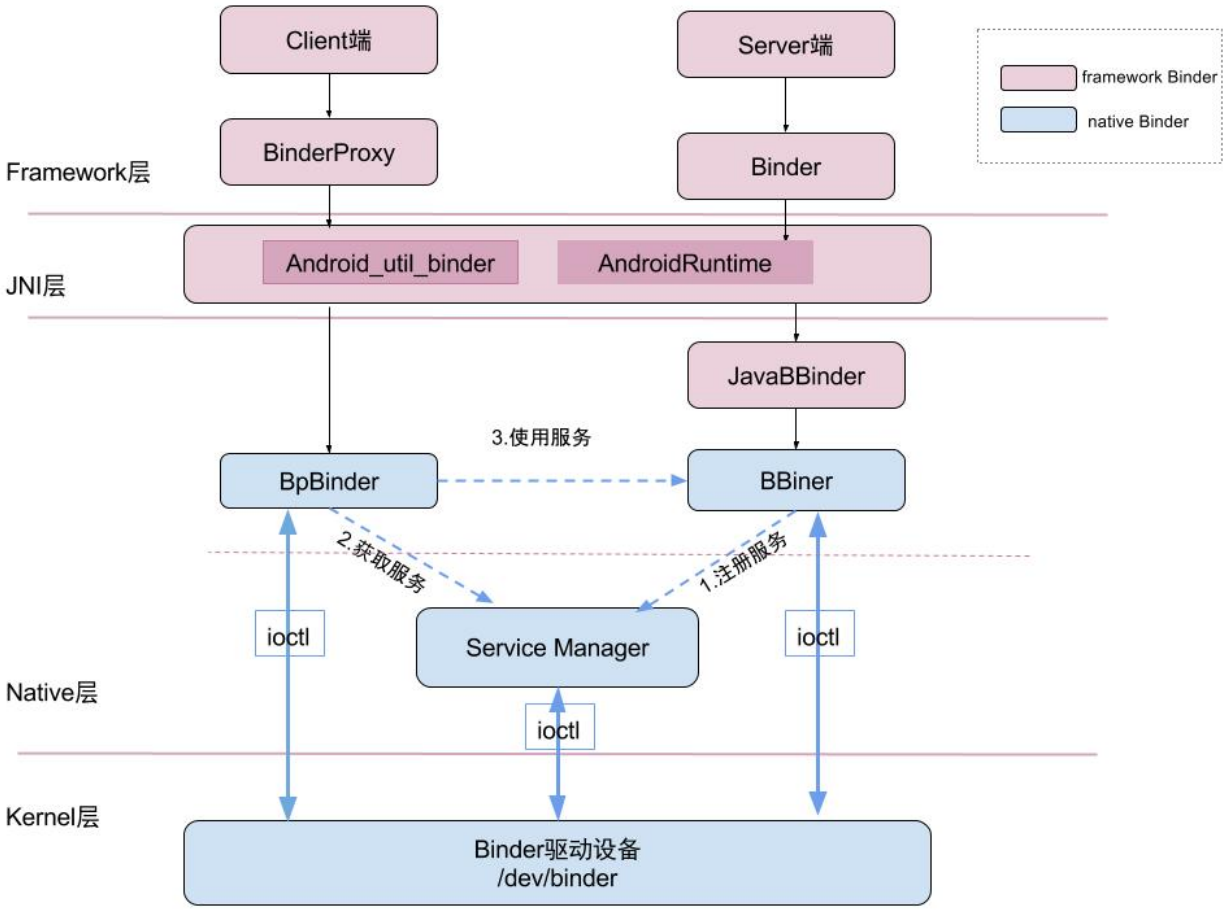
主要分析Binder在java层的框架逻辑

一、概述

1.1 Binder架构

binder在framework层，采用JNI技术来调用native(C/C++)层的binder架构，从而为上层应用程序提供服务。 看过binder系列之前的文章，我们知道native层中，binder是C/S架构，分为Bn端(Server)和Bp端(Client)。对于java层在命名与架构上非常相近，同样实现了一套IPC通信架构。

framework Binder架构图：



- 图中红色代表整个framework层 binder架构相关组件；
  - Binder类代表Server端，BinderProxy类代码Client端；
- 图中蓝色代表native层 binder架构相关组件；
- 上层framework层的binder逻辑，都是建立在native层架构的基础之上的，核心逻辑都是交予native层方法来处理。

1.2 相关源码

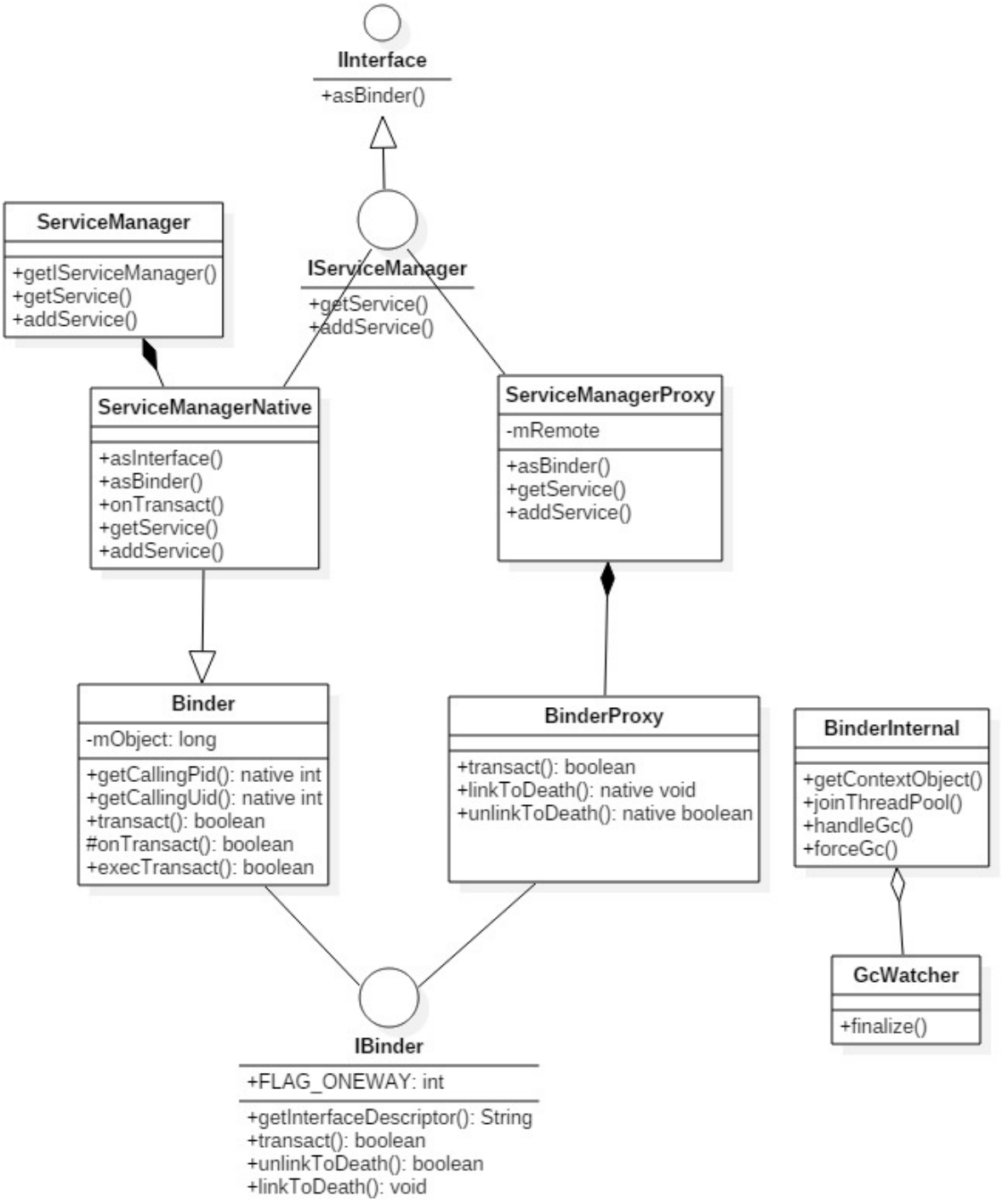
```
/framework/base/core/java/android/os/IInterface.java
/framework/base/core/java/android/os/IServiceManager.java
/framework/base/core/java/android/os/ServiceManager.java
/framework/base/core/java/android/os/ServiceManagerNative.java

/framework/base/core/java/android/os/IBinder.java
/framework/base/core/java/android/os/Binder.java
/framework/base/core/java/android/os/Parcel.java
/framework/base/core/java/com/android/internal/os/BinderInternal.java

/framework/base/core/jni/android_os_Binder.cpp
/framework/base/core/jni/android_os_Parcel.cpp
/framework/base/core/jni/AndroidRuntime.cpp
/framework/base/core/jni/android_util_Binder.cpp
```

Binder类与BinderProxy类 都位于Binder.java文件
ServiceManagerNative类与ServiceManagerProxy类 都位于ServiceManagerNative.java文件

1.3 类关系图



说明：

1. **ServiceManager**通过 getIServiceManager()，获取ServiceManagerNative对象； ServiceManager的addService(), getService()实际都是调用ServiceManagerNative类中相应的方法处理；
2. **ServiceManagerNative**通过asInterface()，获取ServiceManagerProxy对象；
3. **ServiceManagerProxy**的成员变量mRemote指向BinderProxy对象；
4. **Binder**的成员变量mObject和成员方法execTransact()用于native方法
5. **BinderInternal**内部有一个GcWatcher类，用于处理和调试与Binder相关的垃圾回收。

6. **IBinder**接口中常量FLAG_ONEWAY：客户端利用binder跟服务端通信是阻塞式的，但如果设置了FLAG_ONEWAY，这成为非阻塞的调用方式，客户端能立即返回，服务端采用回调方式来通知客户端完成情况。

一、初始化

在Android系统中， 虚拟机创建之初，会调用AndroidRuntime::startReg，进行jni方法的注册。

1.1 startReg

==> /framework/base/core/jni/AndroidRuntime.cpp

注册JNI方法

```
int AndroidRuntime::startReg(JNIEnv* env)
{
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateTh
readEtc);

    env->PushLocalFrame(200);

    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) { //注册j
ni方法
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);

    return 0;
}
```

其中gRegJNI是一个数组，记录所有需要注册的jni方法，其中有一项便是REG_JNI(register_android_os_Binder); Binder注册方法register_android_os_Binder位于android_util_Binder.cpp。

1.2 register_android_os_Binder

==> /framework/base/core/jni/android_util_Binder.cpp

注册Binder相关的jni方法

```
int register_android_os_Binder(JNIEnv* env)
{
    if (int_register_android_os_Binder(env) < 0)  // 注册Binder类的jni方法【见流程3】
        return -1;
    if (int_register_android_os_BinderInternal(env) < 0)// 注册BinderInternal类的jni方法【见流程4】
        return -1;
    if (int_register_android_os_BinderProxy(env) < 0) // 注册BinderProxy类的jni方法【见流程5】
        return -1;
    ...
    return 0;
}
```

1.3 注册 Binder

==> /framework/base/core/jni/android_util_Binder.cpp

注册 Binder类的jni方法

```
static int int_register_android_os_Binder(JNIEnv* env)
{
        //其中kBinderPathName = "android/os/Binder";
    jclass clazz = FindClassOrDie(env, kBinderPathName); //查找全路径名为android/os/Binder的类

    gBinderOffsets.mClass = MakeGlobalRefOrDie(env, clazz); //记录Binder类
    gBinderOffsets.mExecTransact = GetMethodIDOrDie(env, clazz, "execTransact", "(IJJI)Z");  //记录execTransact()方法
    gBinderOffsets.mObject = GetFieldIDOrDie(env, clazz, "mObject", "J");//记录mObject属性
    //方法注册
    return RegisterMethodsOrDie(env, kBinderPathName, gBinderMethods,
                NELEM(gBinderMethods));
}
```

(1) gBinderOffsets

gBinderOffsets是全局静态结构体，保存了binder类的 execTransact() 方法和 mObject 属性，这为JNI层访问Java层的对象提供的通道。

不是每一次都去查找binder对象信息，而是查询一次保存起来，是由于每次查询需要花费较多的CPU时间，尤其是频繁访问时，但用额外的结构体来保存这些信息，是以空间换时间的方法，能提高效率。

```
static struct bindernative_offsets_t
{
    // Class state.
    jclass mClass;
    jmethodID mExecTransact;

    // Object state.
    jfieldID mObject;

} gBinderOffsets;
```

(2)方法说明

- FindClassOrDie(env, kBinderPathName) 基本等价于 env->FindClass(kBinderPathName)
- MakeGlobalRefOrDie() 等价于 env->NewGlobalRef()
- GetMethodIDOrDie() 等价于 env->GetMethodID()
- GetFieldIDOrDie() 等价于 env->GeFieldID()
- RegisterMethodsOrDie() 等价于 Android::registerNativeMethods();

(3)其中gBinderMethods

```
static const JNINativeMethod gBinderMethods[] = {
     /* name, signature, funcPtr */
    { "getCallingPid", "()I", (void*)android_os_Binder_getCallingPid
},
    { "getCallingUid", "()I", (void*)android_os_Binder_getCallingUid
},
    { "clearCallingIdentity", "()J", (void*)android_os_Binder_clearCal
lingIdentity },
    { "restoreCallingIdentity", "(J)V", (void*)android_os_Binder_resto
reCallingIdentity },
    { "setThreadStrictModePolicy", "(I)V", (void*)android_os_Binder_se
tThreadStrictModePolicy },
    { "getThreadStrictModePolicy", "()I", (void*)android_os_Binder_get
ThreadStrictModePolicy },
    { "flushPendingCommands", "()V", (void*)android_os_Binder_flushPen
dingCommands },
    { "init", "()V", (void*)android_os_Binder_init },
    { "destroy", "()V", (void*)android_os_Binder_destroy },
    { "blockUntilThreadAvailable", "()V", (void*)android_os_Binder_blo
ckUntilThreadAvailable }
};
```

通过registerNativeMethods()，为Java层访问JNI层提供了通道。总之，该过程完成了Native层Binder与framework层Binder之间相互通信的桥梁。

1.4 注册BinderInternal

==> /framework/base/core/jni/android_util_Binder.cpp

注册BinderInternal类的jni方法

```
static int int_register_android_os_BinderInternal(JNIEnv* env)
{
        //其中kBinderInternalPathName = "com/android/internal/os/Binder
Internal"
    jclass clazz = FindClassOrDie(env, kBinderInternalPathName);

    gBinderInternalOffsets.mClass = MakeGlobalRefOrDie(env, clazz);
    gBinderInternalOffsets.mForceGc = GetStaticMethodIDOrDie(env, claz
z, "forceBinderGc", "()V");

    return RegisterMethodsOrDie(
        env, kBinderInternalPathName,
        gBinderInternalMethods, NELEM(gBinderInternalMethods));
}
```

gBinderInternalOffsets保存了BinderInternal的 `forceBinderGc()` 方法。

下面是BinderInternal类的JNI方法注册：

```
static const JNINativeMethod gBinderInternalMethods[] = {
    { "getContextObject", "()Landroid/os/IBinder;", (void*)android_o
s_BinderInternal_getContextObject },
    { "joinThreadPool", "()V", (void*)android_os_BinderInternal_joinTh
readPool },
    { "disableBackgroundScheduling", "(Z)V", (void*)android_os_BinderI
nternal_disableBackgroundScheduling },
    { "handleGc", "()V", (void*)android_os_BinderInternal_handleGc }
};
```

1.5 注册BinderProxy

==> /framework/base/core/jni/android_util_Binder.cpp

注册BinderProxy类的jni方法

```
static int int_register_android_os_BinderProxy(JNIEnv* env)
{
    //gErrorOffsets保存了Error类信息
    jclass clazz = FindClassOrDie(env, "java/lang/Error");
    gErrorOffsets.mClass = MakeGlobalRefOrDie(env, clazz);

        //gBinderProxyOffsets保存了BinderProxy类的信息
        //其中kBinderProxyPathName = "android/os/BinderProxy"
    clazz = FindClassOrDie(env, kBinderProxyPathName);
    gBinderProxyOffsets.mClass = MakeGlobalRefOrDie(env, clazz);
    gBinderProxyOffsets.mConstructor = GetMethodIDOrDie(env, clazz, "
<init>", "()V");
    gBinderProxyOffsets.mSendDeathNotice = GetStaticMethodIDOrDie(env,
clazz, "sendDeathNotice", "(Landroid/os/IBinder$DeathRecipient;)V");
    gBinderProxyOffsets.mObject = GetFieldIDOrDie(env, clazz, "mObjec
t", "J");
    gBinderProxyOffsets.mSelf = GetFieldIDOrDie(env, clazz, "mSelf",
"Ljava/lang/ref/WeakReference;");
    gBinderProxyOffsets.mOrgue = GetFieldIDOrDie(env, clazz, "mOrgue",
"J");

        //gClassOffsets保存了Class.getName()方法
    clazz = FindClassOrDie(env, "java/lang/Class");
    gClassOffsets.mGetName = GetMethodIDOrDie(env, clazz, "getName", "
()Ljava/lang/String;");

    return RegisterMethodsOrDie(
        env, kBinderProxyPathName,
        gBinderProxyMethods, NELEM(gBinderProxyMethods));
}
```

gBinderProxyOffsets保存了BinderProxy的构造方法，sendDeathNotice(), mObject, mSelf, mOrgue信息。

下面BinderProxy类的JNI方法注册：

```
static const JNINativeMethod gBinderProxyMethods[] = {
     /* name, signature, funcPtr */
    {"pingBinder",          "()Z", (void*)android_os_BinderProxy_pingB
inder},
    {"isBinderAlive",       "()Z", (void*)android_os_BinderProxy_isBin
derAlive},
    {"getInterfaceDescriptor", "()Ljava/lang/String;", (void*)androi
d_os_BinderProxy_getInterfaceDescriptor},
    {"transactNative",      "(ILandroid/os/Parcel;Landroid/os/Parce
l;I)Z", (void*)android_os_BinderProxy_transact},
    {"linkToDeath",         "(Landroid/os/IBinder$DeathRecipient;I)V",
(void*)android_os_BinderProxy_linkToDeath},
    {"unlinkToDeath",       "(Landroid/os/IBinder$DeathRecipient;I)Z",
(void*)android_os_BinderProxy_unlinkToDeath},
    {"destroy",             "()V", (void*)android_os_BinderProxy_destr
oy},
};
```

二、ServiceManager

首先分析ServiceManager.getIServiceManager

2.1 getIServiceManager

==> /framework/base/core/java/android/os/ServiceManager.java

获取Service manager

private static IServiceManager getIServiceManager() {

```
    if (sServiceManager != null) {
        return sServiceManager;
    }

    sServiceManager = ServiceManagerNative.asInterface(BinderInterna
l.getContextObject());
    return sServiceManager;
}
```

显然，ServiceManager采用了单例模式。

2.2 BinderInternal.getContextObject()

==>

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

```
public static final native IBinder getContextObject();
```

这是一个native方法，根据BinderInternal进行的jni注册方式，可知具体工作交给了下面方法：

==> /framework/base/core/jni/android_util_binder.cpp

```
static jobject android_os_BinderInternal_getContextObject(JNIEnv* env,
jobject clazz)
{
    sp<IBinder> b = ProcessState::self()->getContextObject(NULL);
    return javaObjectForIBinder(env, b);
}
```

对于ProcessState::self()->getContextObject()，在Binder系列3 —— 获取Service Manager (http://www.yuanhh.com/2015/11/08/binder-get-sm/)中详细介绍过。此处直接使用其结论： ProcessState::self()->getContextObject() 等价于 new BpBinder(0)；

2.3 javaObjectForIBinder

==> /framework/base/core/jni/android_util_binder.cpp

将IBinder对象转换为native层的对象，更准确地说BpBinder对象转换成一个BinderProxy对象。

```
jobject javaObjectForIBinder(JNIEnv* env, const sp<IBinder>& val)
{
    if (val == NULL) return NULL;

    if (val->checkSubclass(&gBinderOffsets)) { //返回false
        jobject object = static_cast<JavaBBinder*>(val.get())->objec
t();
        return object;
    }

    AutoMutex _l(mProxyLock);

    jobject object = (jobject)val->findObject(&gBinderProxyOffsets);
    if (object != NULL) { //第一次object为null
        jobject res = jniGetReferent(env, object);
        if (res != NULL) {
            return res;
        }
        android_atomic_dec(&gNumProxyRefs);
        val->detachObject(&gBinderProxyOffsets);
        env->DeleteGlobalRef(object);
    }

    object = env->NewObject(gBinderProxyOffsets.mClass, gBinderProxyOf
fsets.mConstructor);  //创建BinderProxy对象
    if (object != NULL) {
                //BinderProxy.mObject成员变量记录BpBinder对象
        env->SetLongField(object, gBinderProxyOffsets.mObject, (jlon
g)val.get());
        val->incStrong((void*)javaObjectForIBinder);

        jobject refObject = env->NewGlobalRef(
                env->GetObjectField(object, gBinderProxyOffsets.mSel
f));
                //将BinderProxy对象信息附加到BpBinder的成员变量mObjects中
        val->attachObject(&gBinderProxyOffsets, refObject,
                jnienv_to_javavm(env), proxy_cleanup);

        sp<DeathRecipientList> drl = new DeathRecipientList;
        drl->incStrong((void*)javaObjectForIBinder);
                //BinderProxy.mOrgue成员变量记录死亡通知对象
        env->SetLongField(object, gBinderProxyOffsets.mOrgue, reinterp
ret_cast<jlong>(drl.get()));

        android_atomic_inc(&gNumProxyRefs);
        incRefsCreated(env);
    }
    return object;
```

```
}
```

BinderProxy.mObject成员变量记录BpBinder对象.

到此，可知
ServiceManagerNative.asInterface(BinderInternal.getContextObject()) 等价于

```
ServiceManagerNative.asInterface(new BinderProxy())
```

2.4 ServiceManagerNative.asInterface

==> /framework/base/core/java/android/os/ServiceManagerNative.java

```
 static public IServiceManager asInterface(IBinder obj)
{
    if (obj == null) { //obj为BpBinder
        return null;
    }

    IServiceManager in = (IServiceManager)obj.queryLocalInterface(desc
riptor);
    if (in != null) { //in ==null
        return in;
    }

    return new ServiceManagerProxy(obj);
}
```

由此，可知ServiceManagerNative.asInterface(new BinderProxy()) 等价于

```
new ServiceManagerProxy(new BinderProxy())
```

2.5 小结

- ServiceManager.getIServiceManager最终等价于new ServiceManagerProxy(new BinderProxy())；
- framework层的ServiceManager的调用实际的工作确实交给远程接口ServiceManagerProxy的成员变量BinderProxy；
- 而BinderProxy通过jni方式，最终会调用BpBinder对象；可见上层binder架构的核心功能基本都是靠native架构的服务来完成的。

三、注册服务

3.1 addService

==> /framework/base/core/java/android/os/ServiceManager.java

```
public static void addService(String name, IBinder service, boolean al
lowIsolated) {
    try {
        getIServiceManager().addService(name, service, allowIsolated);
//【见3.2】
    } catch (RemoteException e) {
        Log.e(TAG, "error in addService", e);
    }
}
```

getIServiceManager()返回的是ServiceManagerProxy，故调用下面方法

3.2 ServiceManagerNative.addService

==> /framework/base/core/java/android/os/ServiceManagerNative.java

```
public void addService(String name, IBinder service, boolean allowIsol
ated)
        throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IServiceManager.descriptor);
    data.writeString(name);
    data.writeStrongBinder(service); 【见3.3】
    data.writeInt(allowIsolated ? 1 : 0);
    mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0); //mRemo
te为BinderProxy【见3.8】
    reply.recycle();
    data.recycle();
}
```

3.3 writeStrongBinder

==> /framework/base/core/java/android/os/Parcel.java

```
public writeStrongBinder(IBinder val){
        nativewriteStrongBinder(mNativePtr, val); 【见3.4】
}
```

这是一个native调用，进入下面方法。

3.4 android_os_Parcel_writeStrongBinder

==> /framework/base/core/jni/android_os_Parcel.cpp

```
static void android_os_Parcel_writeStrongBinder(JNIEnv* env, jclass cl
azz, jlong nativePtr, jobject object)
{
    Parcel* parcel = reinterpret_cast<Parcel*>(nativePtr); //将java层Pa
rcel转换为native层Parcel
    if (parcel != NULL) {
        const status_t err = parcel->writeStrongBinder(ibinderForJavaO
bject(env, object)); 【见3.5】
        if (err != NO_ERROR) {
            signalExceptionForError(env, clazz, err);
        }
    }
}
```

3.5 ibinderForJavaObject

==> /framework/base/core/jni/android_os_Binder.cpp

```
sp<IBinder> ibinderForJavaObject(JNIEnv* env, jobject obj)
{
    if (obj == NULL) return NULL;

    if (env->IsInstanceOf(obj, gBinderOffsets.mClass)) {
        JavaBBinderHolder* jbh = (JavaBBinderHolder*)
            env->GetLongField(obj, gBinderOffsets.mObject);
        return jbh != NULL ? jbh->get(env, obj) : NULL; 【见3.6】
    }

    if (env->IsInstanceOf(obj, gBinderProxyOffsets.mClass)) {
        return (IBinder*)env->GetLongField(obj, gBinderProxyOffsets.mO
bject);
    }
    return NULL;
}
```

3.6 JavaBBinderHolder.get()

==> /framework/base/core/jni/android_os_Binder.cpp

```
sp<JavaBBinder> get(JNIEnv* env, jobject obj)
{
    AutoMutex _l(mLock);
    sp<JavaBBinder> b = mBinder.promote();
    if (b == NULL) {
        b = new JavaBBinder(env, obj); //首次进来，创建JavaBBinder对
象【见3.7】
        mBinder = b;
    }
    return b;
}
```

JavaBBinderHolder有一个成员变量mBinder，保存当前创建的JavaBBinder对象，这是一个wp类型的，可能会被垃圾回收器给回收，所以每次使用前，都需要先判断是否存在。

3.7 new JavaBBinder()

==> /framework/base/core/jni/android_os_Binder.cpp

创建JavaBBinder，该对象继承于BBinder

```
JavaBBinder(JNIEnv* env, jobject object)
    : mVM(jnienv_to_javavm(env)), mObject(env->NewGlobalRef(object))
{
    android_atomic_inc(&gNumLocalRefs);
    incRefsCreated(env);
}
```

data.writeStrongBinder(service)最终等价于

```
parcel->writeStrongBinder(new JavaBBinder(env, obj));
```

3.8 BinderProxy.transact

==> /framework/base/core/java/android/os/Binder.java

回到ServiceManagerProxy.addService，其成员变量mRemote是BinderProxy。BinderProxy.transact如下：

```
public boolean transact(int code, Parcel data, Parcel reply, int flag
s) throws RemoteException {
        //用于检测Parcel大小是否大于800k
    Binder.checkParcel(this, code, data, "Unreasonably large binder bu
ffer");
    return transactNative(code, data, reply, flags);
}
```

3.9 android_os_BinderProxy_transact

==> /framework/base/core/jni/android_os_Binder.cpp

```
static jboolean android_os_BinderProxy_transact(JNIEnv* env, jobject o
bj,
    jint code, jobject dataObj, jobject replyObj, jint flags) // throw
s RemoteException
{
    if (dataObj == NULL) {
        jniThrowNullPointerException(env, NULL);
        return JNI_FALSE;
    }

    Parcel* data = parcelForJavaObject(env, dataObj); //java Parcel转为
native Parcel
    if (data == NULL) {
        return JNI_FALSE;
    }
    Parcel* reply = parcelForJavaObject(env, replyObj);
    if (reply == NULL && replyObj != NULL) {
        return JNI_FALSE;
    }
    //gBinderProxyOffsets.mObject中保存的是new BpBinder(0)对象
    IBinder* target = (IBinder*)
        env->GetLongField(obj, gBinderProxyOffsets.mObject);
    if (target == NULL) {
        jniThrowException(env, "java/lang/IllegalStateException", "Bin
der has been finalized!");
        return JNI_FALSE;
    }


    bool time_binder_calls;
    int64_t start_millis;
    if (kEnableBinderSample) { //默认为false
        time_binder_calls = should_time_binder_calls();
        if (time_binder_calls) {
            start_millis = uptimeMillis();
        }
    }
    //此处便是BpBinder::transact(),进入Binder驱动程序
    status_t err = target->transact(code, *data, reply, flags);

    if (kEnableBinderSample) {  //默认为false
        if (time_binder_calls) {
            conditionally_log_binder_call(start_millis, target, code);
        }
    }

    if (err == NO_ERROR) {
        return JNI_TRUE;
```

```
    } else if (err == UNKNOWN_TRANSACTION) {
        return JNI_FALSE;
    }

    signalExceptionForError(env, obj, err, true , data->dataSize());
    return JNI_FALSE;
}
```

BinderProxy.transact()，最终核心逻辑是交给BpBinder::transact()完成，在native Binder架构篇Binder系列4 —— 注册服务(addService)(http://www.yuanhh.com/2015/11/14/binder-add-service/)中有详细说明BpBinder工作原理。

小结

注册服务的方法，基本等价于下面：

```
public void addService(String name, IBinder service, boolean allowIsol
ated)
        throws RemoteException {
    ...
        Parcel data = Parcel.obtain(); //此处还需要将java层的Parcel转为Na
tive层的Parcel
        data->writeStrongBinder(new JavaBBinder(env, obj));
        BpBinder::transact(ADD_SERVICE_TRANSACTION, *data, reply, 0);
        ...
}
```

注册服务过程就是通过BpBinder来发送 ADD_SERVICE_TRANSACTION 命令，与实现与binder驱动进行数据交互。

四、获取服务

4.1 getService

==> /framework/base/core/java/android/os/ServiceManager.java

```
public static IBinder getService(String name) {
    try {
        IBinder service = sCache.get(name); //先从缓存中查看
        if (service != null) {
            return service;
        } else {
            return getIServiceManager().getService(name); 【见4.2】
        }
    } catch (RemoteException e) {
        Log.e(TAG, "error in getService", e);
    }
    return null;
}
```

其中sCache = new HashMap<String, IBinder>()以hashmap格式缓存已组成的名称。请求获取服务过程中，先从缓存中查询是否存在，如果缓存中不存在的话，再通过binder交互来查询相应的服务。

4.2 ServiceManagerNative.getService

==> /framework/base/core/java/android/os/ServiceManagerNative.java

```
public IBinder getService(String name) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IServiceManager.descriptor);
    data.writeString(name);
    mRemote.transact(GET_SERVICE_TRANSACTION, data, reply, 0); //mRemo
te为BinderProxy
    IBinder binder = reply.readStrongBinder(); 【见4.3】
    reply.recycle();
    data.recycle();
    return binder;
}
```

mRemote.transact()在前面，已经说明过，通过JNI调用，最终调用的是BpBinder::transact ( )方法。

4.3 readStrongBinder

==> /framework/base/core/java/android/os/Parcel.java

readStrongBinder的过程基本与前面的writeStrongBinder时逆过程。

```
static jobject android_os_Parcel_readStrongBinder(JNIEnv* env, jclass
clazz, jlong nativePtr)
{
    Parcel* parcel = reinterpret_cast<Parcel*>(nativePtr);
    if (parcel != NULL) {
        return javaObjectForIBinder(env, parcel->readStrongBinder());
    }
    return NULL;
}
```

javaObjectForIBinder在第 2.3小节 中已经介绍，javaObjectForIBinder(env, new BpBinder(handle));

4.4 小结

注册服务的方法，基本等价于下面：

```
public void addService(String name, IBinder service, boolean allowIsol
ated)
        throws RemoteException {
    ...
        Parcel reply = Parcel.obtain(); //此处还需要将java层的Parcel转为N
ative层的Parcel
        BpBinder::transact(GET_SERVICE_TRANSACTION, *data, reply, 0);
        IBinder binder = javaObjectForIBinder(env, new BpBinder(handl
e));
        ...
}
```

javaObjectForIBinder作用是 创建BinderProxy对象，并将BpBinder对象的地址保存到BinderProxy对象的mObjects中。

获取服务过程就是通过BpBinder来发送 ADD_SERVICE_TRANSACTION 命令，与实现与binder驱动进行数据交互。

(http://duoshuo.com/settings/avatar/)

说点什么吧…

分享到:

发布

多说 (http://duoshuo.com)

gityuan@gmail.com (mailto:gityuan@gmail.com) · Github

(https://github.com/yuanhuihui) · 天道酬勤 · © 2015 Yuanhh · Jekyll

(https://github.com/jekyll/jekyll) theme by HyG (https://github.com/Gaohaoyang)