[Android系统进程间通信Binder机制在应用程序框架层的Java接口源代码分析](http://blog.csdn.net/luoshengyang/article/details/6642463)

在前面几篇文章中，我们详细介绍了Android系统进程间通信机制Binder的原理，并且深入分析了系统提供的Binder运行库和驱动程序的源代码。细心的读者会发现，这几篇文章分析的Binder接口都是基于C/C++语言来实现的，但是我们在编写应用程序都是基于Java语言的，那么，我们如何使用Java语言来使用系统的Binder机制来进行进程间通信呢？这就是本文要介绍的Android系统应用程序框架层的用Java语言来实现的Binder接口了。

       熟悉Android系统的读者，应该能想到应用程序框架中的基于Java语言的Binder接口是通过JNI来调用基于C/C++语言的Binder运行库来为Java应用程序提供进程间通信服务的了。JNI在Android系统中用得相当普遍，SDK中的Java接口API很多只是简单地通过JNI来调用底层的C/C++运行库从而为应用程序服务的。

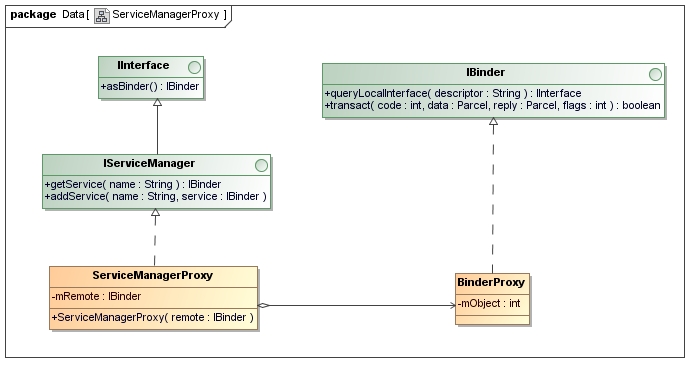
       这里，我们仍然是通过具体的例子来说明Binder机制在应用程序框架层中的Java接口，主要就是Service Manager、Server和Client这三个角色的实现了。通常，在应用程序中，我们都是把Server实现为Service的形式，并且通过IServiceManager.addService接口来把这个Service添加到Service Manager，Client也是通过IServiceManager.getService接口来获得Service接口，接着就可以使用这个Service提供的功能了，这个与运行时库的Binder接口是一致的。

       前面我们[学习Android硬件抽象层](http://blog.csdn.net/luoshengyang/article/details/6567257)时，曾经在应用程序框架层中提供了一个硬件访问服务HelloService，这个Service运行在一个独立的进程中充当Server的角色，使用这个Service的Client运行在另一个进程中，它们之间就是通过Binder机制来通信的了。这里，我们就使用HelloService这个例子来分析Android系统进程间通信Binder机制在应用程序框架层的Java接口源代码。所以希望读者在阅读下面的内容之前，先了解一下前面[在Ubuntu上为Android系统的Application Frameworks层增加硬件访问服务](http://blog.csdn.net/luoshengyang/article/details/6578352)这篇文章。

       这篇文章通过五个情景来学习Android系统进程间通信Binder机制在应用程序框架层的Java接口：1. 获取Service Manager的Java远程接口的过程；2. HelloService接口的定义；3. HelloService的启动过程；4. Client获取HelloService的Java远程接口的过程；5.  Client通过HelloService的Java远程接口来使用HelloService提供的服务的过程。

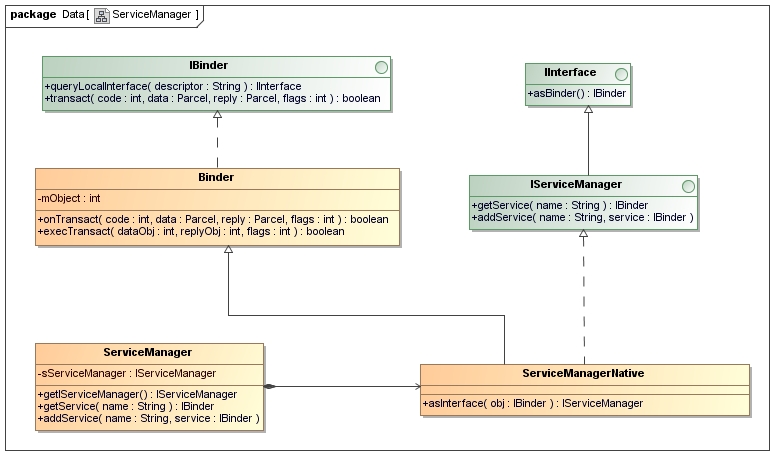
       一.  获取Service Manager的Java远程接口

       我们要获取的Service Manager的Java远程接口是一个ServiceManagerProxy对象的IServiceManager接口。我们现在就来看看ServiceManagerProxy类是长什么样子的：



         这里可以看出，ServiceManagerProxy类实现了IServiceManager接口，IServiceManager提供了getService和addService两个成员函数来管理系统中的Service。从ServiceManagerProxy类的构造函数可以看出，它需要一个BinderProxy对象的IBinder接口来作为参数。因此，要获取Service Manager的Java远程接口ServiceManagerProxy，首先要有一个BinderProxy对象。下面将会看到这个BinderProxy对象是如何获得的。

         再来看一下是通过什么路径来获取Service Manager的Java远程接口ServiceManagerProxy的。这个主角就是ServiceManager了，我们也先看一下ServiceManager是长什么样子的：



        ServiceManager类有一个静态成员函数getIServiceManager，它的作用就是用来获取Service Manager的Java远程接口了，而这个函数又是通过ServiceManagerNative来获取Service Manager的Java远程接口的。

        接下来，我们就看一下ServiceManager.getIServiceManager这个函数的实现，这个函数定义在frameworks/base/core/java/android/os/ServiceManager.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **final** **class** ServiceManager {
2. ......
3. **private** **static** IServiceManager sServiceManager;
4. ......
5. **private** **static** IServiceManager getIServiceManager() {
6. **if** (sServiceManager != **null**) {
7. **return** sServiceManager;
8. }
10. // Find the service manager
11. sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());
12. **return** sServiceManager;
13. }
14. ......
15. }

        如果其静态成员变量sServiceManager尚未创建，那么就调用ServiceManagerNative.asInterface函数来创建。在调用ServiceManagerNative.asInterface函数之前，首先要通过BinderInternal.getContextObject函数来获得一个BinderProxy对象。

        我们来看一下BinderInternal.getContextObject的实现，这个函数定义在frameworks/base/core/java/com/android/internal/os/BinderInternal.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **class** BinderInternal {
2. ......
3. /\*\*
4. \* Return the global "context object" of the system.  This is usually
5. \* an implementation of IServiceManager, which you can use to find
6. \* other services.
7. \*/
8. **public** **static** **final** **native** IBinder getContextObject();
10. ......
11. }

        这里可以看出，BinderInternal.getContextObject是一个JNI方法，它实现在frameworks/base/core/jni/android\_util\_Binder.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **static** jobject android\_os\_BinderInternal\_getContextObject(JNIEnv\* env, jobject clazz)
2. {
3. sp<IBinder> b = ProcessState::self()->getContextObject(NULL);
4. **return** javaObjectForIBinder(env, b);
5. }

       这里看到我们熟悉的ProcessState::self()->getContextObject函数，具体可以参考[浅谈Android系统进程间通信（IPC）机制Binder中的Server和Client获得Service Manager接口之路](http://blog.csdn.net/luoshengyang/article/details/6627260)一文。ProcessState::self()->getContextObject函数返回一个BpBinder对象，它的句柄值是0，即下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

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

       相当于是：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. sp<IBinder> b = **new** BpBinder(0);

       接着调用javaObjectForIBinder把这个BpBinder对象转换成一个BinderProxy对象：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. jobject javaObjectForIBinder(JNIEnv\* env, **const** sp<IBinder>& val)
2. {
3. **if** (val == NULL) **return** NULL;
5. **if** (val->checkSubclass(&gBinderOffsets)) {
6. // One of our own!
7. jobject object = **static\_cast**<JavaBBinder\*>(val.get())->object();
8. //printf("objectForBinder %p: it's our own %p!\n", val.get(), object);
9. **return** object;
10. }
12. // For the rest of the function we will hold this lock, to serialize
13. // looking/creation of Java proxies for native Binder proxies.
14. AutoMutex \_l(mProxyLock);
16. // Someone else's...  do we know about it?
17. jobject object = (jobject)val->findObject(&gBinderProxyOffsets);
18. **if** (object != NULL) {
19. jobject res = env->CallObjectMethod(object, gWeakReferenceOffsets.mGet);
20. **if** (res != NULL) {
21. LOGV("objectForBinder %p: found existing %p!\n", val.get(), res);
22. **return** res;
23. }
24. LOGV("Proxy object %p of IBinder %p no longer in working set!!!", object, val.get());
25. android\_atomic\_dec(&gNumProxyRefs);
26. val->detachObject(&gBinderProxyOffsets);
27. env->DeleteGlobalRef(object);
28. }
30. object = env->NewObject(gBinderProxyOffsets.mClass, gBinderProxyOffsets.mConstructor);
31. **if** (object != NULL) {
32. LOGV("objectForBinder %p: created new %p!\n", val.get(), object);
33. // The proxy holds a reference to the native object.
34. env->SetIntField(object, gBinderProxyOffsets.mObject, (**int**)val.get());
35. val->incStrong(object);
37. // The native object needs to hold a weak reference back to the
38. // proxy, so we can retrieve the same proxy if it is still active.
39. jobject refObject = env->NewGlobalRef(
40. env->GetObjectField(object, gBinderProxyOffsets.mSelf));
41. val->attachObject(&gBinderProxyOffsets, refObject,
42. jnienv\_to\_javavm(env), proxy\_cleanup);
44. // Note that a new object reference has been created.
45. android\_atomic\_inc(&gNumProxyRefs);
46. incRefsCreated(env);
47. }
49. **return** object;
50. }

        在介绍这个函数之前，先来看两个变量gBinderOffsets和gBinderProxyOffsets的定义。

        先看gBinderOffsets的定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **static** **struct** bindernative\_offsets\_t
2. {
3. // Class state.
4. jclass mClass;
5. jmethodID mExecTransact;
7. // Object state.
8. jfieldID mObject;
10. } gBinderOffsets;

       简单来说，gBinderOffsets变量是用来记录上面第二个类图中的Binder类的相关信息的，它是在注册Binder类的JNI方法的int\_register\_android\_os\_Binder函数初始化的：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **const** **char**\* **const** kBinderPathName = "android/os/Binder";
3. **static** **int** int\_register\_android\_os\_Binder(JNIEnv\* env)
4. {
5. jclass clazz;
7. clazz = env->FindClass(kBinderPathName);
8. LOG\_FATAL\_IF(clazz == NULL, "Unable to find class android.os.Binder");
10. gBinderOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
11. gBinderOffsets.mExecTransact
12. = env->GetMethodID(clazz, "execTransact", "(IIII)Z");
13. assert(gBinderOffsets.mExecTransact);
15. gBinderOffsets.mObject
16. = env->GetFieldID(clazz, "mObject", "I");
17. assert(gBinderOffsets.mObject);
19. **return** AndroidRuntime::registerNativeMethods(
20. env, kBinderPathName,
21. gBinderMethods, NELEM(gBinderMethods));
22. }

       再来看gBinderProxyOffsets的定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **static** **struct** binderproxy\_offsets\_t
2. {
3. // Class state.
4. jclass mClass;
5. jmethodID mConstructor;
6. jmethodID mSendDeathNotice;
8. // Object state.
9. jfieldID mObject;
10. jfieldID mSelf;
12. } gBinderProxyOffsets;

       简单来说，gBinderProxyOffsets是用来变量是用来记录上面第一个图中的BinderProxy类的相关信息的，它是在注册BinderProxy类的JNI方法的int\_register\_android\_os\_BinderProxy函数初始化的：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **const** **char**\* **const** kBinderProxyPathName = "android/os/BinderProxy";
3. **static** **int** int\_register\_android\_os\_BinderProxy(JNIEnv\* env)
4. {
5. jclass clazz;
7. clazz = env->FindClass("java/lang/ref/WeakReference");
8. LOG\_FATAL\_IF(clazz == NULL, "Unable to find class java.lang.ref.WeakReference");
9. gWeakReferenceOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
10. gWeakReferenceOffsets.mGet
11. = env->GetMethodID(clazz, "get", "()Ljava/lang/Object;");
12. assert(gWeakReferenceOffsets.mGet);
14. clazz = env->FindClass("java/lang/Error");
15. LOG\_FATAL\_IF(clazz == NULL, "Unable to find class java.lang.Error");
16. gErrorOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
18. clazz = env->FindClass(kBinderProxyPathName);
19. LOG\_FATAL\_IF(clazz == NULL, "Unable to find class android.os.BinderProxy");
21. gBinderProxyOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
22. gBinderProxyOffsets.mConstructor
23. = env->GetMethodID(clazz, "<init>", "()V");
24. assert(gBinderProxyOffsets.mConstructor);
25. gBinderProxyOffsets.mSendDeathNotice
26. = env->GetStaticMethodID(clazz, "sendDeathNotice", "(Landroid/os/IBinder$DeathRecipient;)V");
27. assert(gBinderProxyOffsets.mSendDeathNotice);
29. gBinderProxyOffsets.mObject
30. = env->GetFieldID(clazz, "mObject", "I");
31. assert(gBinderProxyOffsets.mObject);
32. gBinderProxyOffsets.mSelf
33. = env->GetFieldID(clazz, "mSelf", "Ljava/lang/ref/WeakReference;");
34. assert(gBinderProxyOffsets.mSelf);
36. **return** AndroidRuntime::registerNativeMethods(
37. env, kBinderProxyPathName,
38. gBinderProxyMethods, NELEM(gBinderProxyMethods));
39. }

        回到前面的javaObjectForIBinder函数中，下面这段代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **if** (val->checkSubclass(&gBinderOffsets)) {
2. // One of our own!
3. jobject object = **static\_cast**<JavaBBinder\*>(val.get())->object();
4. //printf("objectForBinder %p: it's our own %p!\n", val.get(), object);
5. **return** object;
6. }

        前面说过，这里传进来的参数是一个BpBinder的指针，而BpBinder::checkSubclass继承于父类IBinder::checkSubclass，它什么也不做就返回false。

        于是函数继续往下执行：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. jobject object = (jobject)val->findObject(&gBinderProxyOffsets);

        由于这个BpBinder对象是第一创建，它里面什么对象也没有，因此，这里返回的object为NULL。

        于是函数又继续往下执行：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. object = env->NewObject(gBinderProxyOffsets.mClass, gBinderProxyOffsets.mConstructor);

        这里，就创建了一个BinderProxy对象了。创建了之后，要把这个BpBinder对象和这个BinderProxy对象关联起来：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. env->SetIntField(object, gBinderProxyOffsets.mObject, (**int**)val.get());

        就是通过BinderProxy.mObject成员变量来关联的了，BinderProxy.mObject成员变量记录了这个BpBinder对象的地址。

        接下去，还要把它放到BpBinder里面去，下次就要使用时，就可以在上一步调用BpBinder::findObj把它找回来了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. val->attachObject(&gBinderProxyOffsets, refObject,
2. jnienv\_to\_javavm(env), proxy\_cleanup);

        最后，就把这个BinderProxy返回到android\_os\_BinderInternal\_getContextObject函数，最终返回到最开始的ServiceManager.getIServiceManager函数中来了，于是，我们就获得一个BinderProxy对象了。

        回到ServiceManager.getIServiceManager中，从下面语句返回：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());

        相当于是：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. sServiceManager = ServiceManagerNative.asInterface(**new** BinderProxy());

       接下去就是调用ServiceManagerNative.asInterface函数了，这个函数定义在frameworks/base/core/java/android/os/ServiceManagerNative.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **abstract** **class** ServiceManagerNative ......
2. {
3. ......
4. **static** **public** IServiceManager asInterface(IBinder obj)
5. {
6. **if** (obj == **null**) {
7. **return** **null**;
8. }
9. IServiceManager in =
10. (IServiceManager)obj.queryLocalInterface(descriptor);
11. **if** (in != **null**) {
12. **return** in;
13. }
15. **return** **new** ServiceManagerProxy(obj);
16. }
17. ......
18. }

       这里的参数obj是一个BinderProxy对象，它的queryLocalInterface函数返回null。因此，最终以这个BinderProxy对象为参数创建一个ServiceManagerProxy对象。

       返回到ServiceManager.getIServiceManager中，从下面语句返回：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. sServiceManager = ServiceManagerNative.asInterface(**new** BinderProxy());

       就相当于是：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. sServiceManager = **new** ServiceManagerProxy(**new** BinderProxy());

      于是，我们的目标终于完成了。

      总结一下，就是在Java层，我们拥有了一个Service Manager远程接口ServiceManagerProxy，而这个ServiceManagerProxy对象在JNI层有一个句柄值为0的BpBinder对象与之通过gBinderProxyOffsets关联起来。

      这样获取Service Manager的Java远程接口的过程就完成了。

      二. HelloService接口的定义

      前面我们在学习Android系统的硬件抽象层（HAL）时，在[在Ubuntu上为Android系统的Application Frameworks层增加硬件访问服务](http://blog.csdn.net/luoshengyang/article/details/6578352)这篇文章中，我们编写了一个硬件服务HelloService，它的服务接口定义在frameworks/base/core/java/android/os/IHelloService.aidl文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **package** android.os;
3. **interface** IHelloService
4. {
5. **void** setVal(**int** val);
6. **int** getVal();
7. }

       这个服务接口很简单，只有两个函数，分别用来读写硬件寄存器。

       注意，这是一个aidl文件，编译后会生成一个IHelloService.java。我们来看一下这个文件的内容隐藏着什么奥秘，可以这么神奇地支持进程间通信。

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. /\*
2. \* This file is auto-generated.  DO NOT MODIFY.
3. \* Original file: frameworks/base/core/java/android/os/IHelloService.aidl
4. \*/
5. **package** android.os;
6. **public** **interface** IHelloService **extends** android.os.IInterface
7. {
8. /\*\* Local-side IPC implementation stub class. \*/
9. **public** **static** **abstract** **class** Stub **extends** android.os.Binder **implements** android.os.IHelloService
10. {
11. **private** **static** **final** java.lang.String DESCRIPTOR = "android.os.IHelloService";
12. /\*\* Construct the stub at attach it to the interface. \*/
13. **public** Stub()
14. {
15. **this**.attachInterface(**this**, DESCRIPTOR);
16. }
18. /\*\*
19. \* Cast an IBinder object into an android.os.IHelloService interface,
20. \* generating a proxy if needed.
21. \*/
22. **public** **static** android.os.IHelloService asInterface(android.os.IBinder obj)
23. {
24. **if** ((obj==**null**)) {
25. **return** **null**;
26. }
27. android.os.IInterface iin = (android.os.IInterface)obj.queryLocalInterface(DESCRIPTOR);
28. **if** (((iin!=**null**)&&(iin **instanceof** android.os.IHelloService))) {
29. **return** ((android.os.IHelloService)iin);
30. }
31. **return** **new** android.os.IHelloService.Stub.Proxy(obj);
32. }
34. **public** android.os.IBinder asBinder()
35. {
36. **return** **this**;
37. }
39. @Override
40. **public** **boolean** onTransact(**int** code, android.os.Parcel data, android.os.Parcel reply, **int** flags) **throws** android.os.RemoteException
41. {
42. **switch** (code)
43. {
44. **case** INTERFACE\_TRANSACTION:
45. {
46. reply.writeString(DESCRIPTOR);
47. **return** **true**;
48. }
49. **case** TRANSACTION\_setVal:
50. {
51. data.enforceInterface(DESCRIPTOR);
52. **int** \_arg0;
53. \_arg0 = data.readInt();
54. **this**.setVal(\_arg0);
55. reply.writeNoException();
56. **return** **true**;
57. }
58. **case** TRANSACTION\_getVal:
59. {
60. data.enforceInterface(DESCRIPTOR);
61. **int** \_result = **this**.getVal();
62. reply.writeNoException();
63. reply.writeInt(\_result);
64. **return** **true**;
65. }
66. }
67. **return** **super**.onTransact(code, data, reply, flags);
68. }
70. **private** **static** **class** Proxy **implements** android.os.IHelloService
71. {
72. **private** android.os.IBinder mRemote;
74. Proxy(android.os.IBinder remote)
75. {
76. mRemote = remote;
77. }
79. **public** android.os.IBinder asBinder()
80. {
81. **return** mRemote;
82. }
84. **public** java.lang.String getInterfaceDescriptor()
85. {
86. **return** DESCRIPTOR;
87. }
89. **public** **void** setVal(**int** val) **throws** android.os.RemoteException
90. {
91. android.os.Parcel \_data = android.os.Parcel.obtain();
92. android.os.Parcel \_reply = android.os.Parcel.obtain();
93. **try** {
94. \_data.writeInterfaceToken(DESCRIPTOR);
95. \_data.writeInt(val);
96. mRemote.transact(Stub.TRANSACTION\_setVal, \_data, \_reply, 0);
97. \_reply.readException();
98. }
99. **finally** {
100. \_reply.recycle();
101. \_data.recycle();
102. }
103. }
105. **public** **int** getVal() **throws** android.os.RemoteException
106. {
107. android.os.Parcel \_data = android.os.Parcel.obtain();
108. android.os.Parcel \_reply = android.os.Parcel.obtain();
109. **int** \_result;
110. **try** {
111. \_data.writeInterfaceToken(DESCRIPTOR);
112. mRemote.transact(Stub.TRANSACTION\_getVal, \_data, \_reply, 0);
113. \_reply.readException();
114. \_result = \_reply.readInt();
115. }
116. **finally** {
117. \_reply.recycle();
118. \_data.recycle();
119. }
120. **return** \_result;
121. }
122. }
124. **static** **final** **int** TRANSACTION\_setVal = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 0);
125. **static** **final** **int** TRANSACTION\_getVal = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 1);
126. }
128. **public** **void** setVal(**int** val) **throws** android.os.RemoteException;
129. **public** **int** getVal() **throws** android.os.RemoteException;
130. }

        这里我们可以看到IHelloService.aidl这个文件编译后的真面目，原来就是根据IHelloService接口的定义生成相应的Stub和Proxy类，这个就是我们熟悉的Binder机制的内容了，即实现这个HelloService的Server必须继续于这里的IHelloService.Stub类，而这个HelloService的远程接口就是这里的IHelloService.Stub.Proxy对象获得的IHelloService接口。接下来的内容，我们就可以看到IHelloService.Stub和IHelloService.Stub.Proxy是怎么创建或者使用的。

        三. HelloService的启动过程

        在讨论HelloService的启动过程之前，我们先来看一下实现HelloService接口的Server是怎么定义的。

        回忆[在Ubuntu上为Android系统的Application Frameworks层增加硬件访问服务](http://blog.csdn.net/luoshengyang/article/details/6578352)一文，我们在frameworks/base/services/java/com/android/server目录下新增了一个HelloService.java文件：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **package** com.android.server;
3. **import** android.content.Context;
4. **import** android.os.IHelloService;
5. **import** android.util.Slog;
7. **public** **class** HelloService **extends** IHelloService.Stub {
8. **private** **static** **final** String TAG = "HelloService";
10. HelloService() {
11. init\_native();
12. }
14. **public** **void** setVal(**int** val) {
15. setVal\_native(val);
16. }
18. **public** **int** getVal() {
19. **return** getVal\_native();
20. }
22. **private** **static** **native** **boolean** init\_native();
23. **private** **static** **native** **void** setVal\_native(**int** val);
24. **private** **static** **native** **int** getVal\_native();
25. }

        这里，我们可以看到，HelloService继续了IHelloService.Stub类，它通过本地方法调用实现了getVal和setVal两个函数。我们不关心这两个函数的具体实现，有兴趣的读者可以参考[在Ubuntu上为Android系统的Application Frameworks层增加硬件访问服务](http://blog.csdn.net/luoshengyang/article/details/6578352)一文。  
        有了HelloService这个Server类后，下一步就是考虑怎么样把它启动起来了。在frameworks/base/services/java/com/android/server/SystemServer.java文件中，定义了SystemServer类。SystemServer对象是在系统启动的时候创建的，它被创建的时候会启动一个线程来创建HelloService，并且把它添加到Service Manager中去。

       我们来看一下这部份的代码：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **class** ServerThread **extends** Thread {
2. ......
4. @Override
5. **public** **void** run() {
7. ......
9. Looper.prepare();
11. ......
13. **try** {
14. Slog.i(TAG, "Hello Service");
15. ServiceManager.addService("hello", **new** HelloService());
16. } **catch** (Throwable e) {
17. Slog.e(TAG, "Failure starting Hello Service", e);
18. }
20. ......
22. Looper.loop();
24. ......
25. }
26. }
28. ......
30. **public** **class** SystemServer
31. {
32. ......
34. /\*\*
35. \* This method is called from Zygote to initialize the system. This will cause the native
36. \* services (SurfaceFlinger, AudioFlinger, etc..) to be started. After that it will call back
37. \* up into init2() to start the Android services.
38. \*/
39. **native** **public** **static** **void** init1(String[] args);
41. ......
43. **public** **static** **final** **void** init2() {
44. Slog.i(TAG, "Entered the Android system server!");
45. Thread thr = **new** ServerThread();
46. thr.setName("android.server.ServerThread");
47. thr.start();
48. }
49. ......
50. }

        这里，我们可以看到，在ServerThread.run函数中，执行了下面代码把HelloService添加到Service Manager中去。这里我们关注把HelloService添加到Service Manager中去的代码：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **try** {
2. Slog.i(TAG, "Hello Service");
3. ServiceManager.addService("hello", **new** HelloService());
4. } **catch** (Throwable e) {
5. Slog.e(TAG, "Failure starting Hello Service", e);
6. }

         通过调用ServiceManager.addService把一个HelloService实例添加到Service Manager中去。

         我们先来看一下HelloService的创建过程：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **new** HelloService();

         这个语句会调用HelloService类的构造函数，而HelloService类继承于IHelloService.Stub类，IHelloService.Stub类又继承了Binder类，因此，最后会调用Binder类的构造函数：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **class** Binder **implements** IBinder {
2. ......
4. **private** **int** mObject;
6. ......

9. **public** Binder() {
10. init();
11. ......
12. }

15. **private** **native** **final** **void** init();

18. ......
19. }

        这里调用了一个JNI方法init来初始化这个Binder对象，这个JNI方法定义在frameworks/base/core/jni/android\_util\_Binder.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **static** **void** android\_os\_Binder\_init(JNIEnv\* env, jobject clazz)
2. {
3. JavaBBinderHolder\* jbh = **new** JavaBBinderHolder(env, clazz);
4. **if** (jbh == NULL) {
5. jniThrowException(env, "java/lang/OutOfMemoryError", NULL);
6. **return**;
7. }
8. LOGV("Java Binder %p: acquiring first ref on holder %p", clazz, jbh);
9. jbh->incStrong(clazz);
10. env->SetIntField(clazz, gBinderOffsets.mObject, (**int**)jbh);
11. }

        它实际上只做了一件事情，就是创建一个JavaBBinderHolder对象jbh，然后把这个对象的地址保存在上面的Binder类的mObject成员变量中，后面我们会用到。

        回到ServerThread.run函数中，我们再来看一下ServiceManager.addService函数的实现：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **final** **class** ServiceManager {
2. ......
4. **private** **static** IServiceManager sServiceManager;
6. ......
8. **public** **static** **void** addService(String name, IBinder service) {
9. **try** {
10. getIServiceManager().addService(name, service);
11. } **catch** (RemoteException e) {
12. Log.e(TAG, "error in addService", e);
13. }
14. }
16. ......
18. }

         这里的getIServiceManager函数我们在前面已经分析过了，它返回的是一个ServiceManagerProxy对象的IServiceManager接口。因此，我们进入到ServiceManagerProxy.addService中去看看：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **class** ServiceManagerProxy **implements** IServiceManager {
2. **public** ServiceManagerProxy(IBinder remote) {
3. mRemote = remote;
4. }
6. ......
8. **public** **void** addService(String name, IBinder service)
9. **throws** RemoteException {
10. Parcel data = Parcel.obtain();
11. Parcel reply = Parcel.obtain();
12. data.writeInterfaceToken(IServiceManager.descriptor);
13. data.writeString(name);
14. data.writeStrongBinder(service);
15. mRemote.transact(ADD\_SERVICE\_TRANSACTION, data, reply, 0);
16. reply.recycle();
17. data.recycle();
18. }
20. ......
22. **private** IBinder mRemote;
23. }

       这里的Parcel类是用Java来实现的，它跟我们前面几篇文章介绍Binder机制时提到的用C++实现的Parcel类的作用是一样的，即用来在两个进程之间传递数据。

       这里我们关注是如何把参数service写到data这个Parcel对象中去的：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. data.writeStrongBinder(service);

       我们来看看Parcel.writeStrongBinder函数的实现：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **final** **class** Parcel {
2. ......
4. /\*\*
5. \* Write an object into the parcel at the current dataPosition(),
6. \* growing dataCapacity() if needed.
7. \*/
8. **public** **final** **native** **void** writeStrongBinder(IBinder val);
10. ......
11. }

        这里的writeStrongBinder函数又是一个JNI方法，它定义在frameworks/base/core/jni/android\_util\_Binder.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **static** **void** android\_os\_Parcel\_writeStrongBinder(JNIEnv\* env, jobject clazz, jobject object)
2. {
3. Parcel\* parcel = parcelForJavaObject(env, clazz);
4. **if** (parcel != NULL) {
5. **const** status\_t err = parcel->writeStrongBinder(ibinderForJavaObject(env, object));
6. **if** (err != NO\_ERROR) {
7. jniThrowException(env, "java/lang/OutOfMemoryError", NULL);
8. }
9. }
10. }

       这里的clazz参数是一个Java语言实现的Parcel对象，通过parcelForJavaObject把它转换成C++语言实现的Parcel对象。这个函数的实现我们就不看了，有兴趣的读者可以研究一下，这个函数也是实现在frameworks/base/core/jni/android\_util\_Binder.cpp这个文件中。  
       这里的object参数是一个Java语言实现的Binder对象，在调用C++语言实现的Parcel::writeStrongBinder把这个对象写入到parcel对象时，首先通过ibinderForJavaObject函数把这个Java语言实现的Binder对象转换为C++语言实现的JavaBBinderHolder对象：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. sp<IBinder> ibinderForJavaObject(JNIEnv\* env, jobject obj)
2. {
3. **if** (obj == NULL) **return** NULL;
5. **if** (env->IsInstanceOf(obj, gBinderOffsets.mClass)) {
6. JavaBBinderHolder\* jbh = (JavaBBinderHolder\*)
7. env->GetIntField(obj, gBinderOffsets.mObject);
8. **return** jbh != NULL ? jbh->get(env) : NULL;
9. }
11. **if** (env->IsInstanceOf(obj, gBinderProxyOffsets.mClass)) {
12. **return** (IBinder\*)
13. env->GetIntField(obj, gBinderProxyOffsets.mObject);
14. }
16. LOGW("ibinderForJavaObject: %p is not a Binder object", obj);
17. **return** NULL;
18. }

         我们知道，这里的obj参数是一个Binder类的实例，因此，这里会进入到第一个if语句中去。

         在前面创建HelloService对象，曾经在调用到HelloService的父类Binder中，曾经在JNI层创建了一个JavaBBinderHolder对象，然后把这个对象的地址保存在Binder类的mObject成员变量中，因此，这里把obj对象的mObject成员变量强制转为JavaBBinderHolder对象。

         到了这里，这个函数的功课还未完成，还剩下最后关键的一步：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **return** jbh != NULL ? jbh->get(env) : NULL;

        这里就是jbh->get这个语句了。

        在JavaBBinderHolder类中，有一个成员变量mBinder，它的类型为JavaBBinder，而JavaBBinder类继承于BBinder类。在前面学习Binder机制的C++语言实现时，我们在[Android系统进程间通信（IPC）机制Binder中的Server启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6629298)这篇文章中，曾经介绍过，IPCThreadState类负责与Binder驱动程序进行交互，它把从Binder驱动程序读出来的请求作简单的处理后，最后把这个请求扔给BBinder的onTransact函数来进一步处理。

        这里，我们就是要把JavaBBinderHolder里面的JavaBBinder类型Binder实体添加到Service Manager中去，以便使得这个HelloService有Client来请求服务时，由Binder驱动程序来唤醒这个Server线程，进而调用这个JavaBBinder类型Binder实体的onTransact函数来进一步处理，这个函数我们在后面会继续介绍。

       先来看一下JavaBBinderHolder::get函数的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **class** JavaBBinderHolder : **public** RefBase
2. {
3. ......
5. JavaBBinderHolder(JNIEnv\* env, jobject object)
6. : mObject(object)
7. {
8. ......
9. }
11. ......
13. sp<JavaBBinder> get(JNIEnv\* env)
14. {
15. AutoMutex \_l(mLock);
16. sp<JavaBBinder> b = mBinder.promote();
17. **if** (b == NULL) {
18. b = **new** JavaBBinder(env, mObject);
19. mBinder = b;
20. ......
21. }
23. **return** b;
24. }
26. ......
28. jobject         mObject;
29. wp<JavaBBinder> mBinder;
30. };

       这里是第一次调用get函数，因此，会创建一个JavaBBinder对象，并且保存在mBinder成员变量中。注意，这里的mObject就是上面创建的HelloService对象了，这是一个Java对象。这个HelloService对象最终也会保存在JavaBBinder对象的成员变量mObject中。

       回到android\_os\_Parcel\_writeStrongBinder函数中，下面这个语句：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **const** status\_t err = parcel->writeStrongBinder(ibinderForJavaObject(env, object));

       相当于是：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **const** status\_t err = parcel->writeStrongBinder((JavaBBinderHodler\*)(obj.mObject));

       因此，这里的效果相当于是写入了一个JavaBBinder类型的Binder实体到parcel中去。这与我们前面介绍的Binder机制的C++实现是一致的。

       接着，再回到ServiceManagerProxy.addService这个函数中，最后它通过其成员变量mRemote来执行进程间通信操作。前面我们在介绍如何获取Service Manager远程接口时提到，这里的mRemote成员变量实际上是一个BinderProxy对象，因此，我们再来看看BinderProxy.transact函数的实现：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **final** **class** BinderProxy **implements** IBinder {
2. ......
4. **public** **native** **boolean** transact(**int** code, Parcel data, Parcel reply,
5. **int** flags) **throws** RemoteException;
7. ......
8. }

       这里的transact成员函数又是一个JNI方法，它定义在frameworks/base/core/jni/android\_util\_Binder.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **static** jboolean android\_os\_BinderProxy\_transact(JNIEnv\* env, jobject obj,
2. jint code, jobject dataObj,
3. jobject replyObj, jint flags)
4. {
5. ......
7. Parcel\* data = parcelForJavaObject(env, dataObj);
8. **if** (data == NULL) {
9. **return** JNI\_FALSE;
10. }
11. Parcel\* reply = parcelForJavaObject(env, replyObj);
12. **if** (reply == NULL && replyObj != NULL) {
13. **return** JNI\_FALSE;
14. }
16. IBinder\* target = (IBinder\*)
17. env->GetIntField(obj, gBinderProxyOffsets.mObject);
18. **if** (target == NULL) {
19. jniThrowException(env, "java/lang/IllegalStateException", "Binder has been finalized!");
20. **return** JNI\_FALSE;
21. }
23. ......
25. status\_t err = target->transact(code, \*data, reply, flags);
27. ......
29. **if** (err == NO\_ERROR) {
30. **return** JNI\_TRUE;
31. } **else** **if** (err == UNKNOWN\_TRANSACTION) {
32. **return** JNI\_FALSE;
33. }
35. signalExceptionForError(env, obj, err);
36. **return** JNI\_FALSE;
37. }

        这里传进来的参数dataObj和replyObj是一个Java接口实现的Parcel类，由于这里是JNI层，需要把它转换为C++实现的Parcel类，它们就是通过我们前面说的parcelForJavaObject函数进行转换的。

        前面我们在分析如何获取Service Manager远程接口时，曾经说到，在JNI层中，创建了一个BpBinder对象，它的句柄值为0，它的地址保存在gBinderProxyOffsets.mObject中，因此，这里通过下面语句得到这个BpBinder对象的IBinder接口：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. IBinder\* target = (IBinder\*)
2. env->GetIntField(obj, gBinderProxyOffsets.mObject);

        有了这个IBinder接口后，就和我们前面几篇文章介绍Binder机制的C/C++实现一致了。

        最后，通过BpBinder::transact函数进入到Binder驱动程序，然后Binder驱动程序唤醒Service Manager响应这个ADD\_SERVICE\_TRANSACTION请求：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

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

       具体可以参考[Android系统进程间通信（IPC）机制Binder中的Server启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6629298)一文。需要注意的是，这里的data包含了一个JavaBBinderHolder类型的Binder实体对象，它就代表了我们上面创建的HelloService。Service Manager收到这个ADD\_SERVICE\_TRANSACTION请求时，就会把这个Binder实体纳入到自己内部进行管理。  
       这样，实现HelloService的Server的启动过程就完成了。

       四. Client获取HelloService的Java远程接口的过程

        前面我们在学习Android系统硬件抽象层（HAL）时，在[在Ubuntu上为Android系统内置Java应用程序测试Application Frameworks层的硬件服务](http://blog.csdn.net/luoshengyang/article/details/6580267)这篇文章中，我们创建了一个应用程序，这个应用程序作为一个Client角色，借助Service Manager这个Java远程接口来获得HelloService的远程接口，进而调用HelloService提供的服务。

        我们看看它是如何借助Service Manager这个Java远程接口来获得HelloService的远程接口的。在Hello这个Activity的onCreate函数，通过IServiceManager.getService函数来获得HelloService的远程接口：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **class** Hello **extends** Activity **implements** OnClickListener {
2. ......
4. **private** IHelloService helloService = **null**;
6. ......
8. @Override
9. **public** **void** onCreate(Bundle savedInstanceState) {
11. helloService = IHelloService.Stub.asInterface(
12. ServiceManager.getService("hello"));
13. }
15. ......
16. }

        我们先来看ServiceManager.getService的实现。前面我们说过，这里实际上是调用了ServiceManagerProxy.getService函数：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **class** ServiceManagerProxy **implements** IServiceManager {
2. **public** ServiceManagerProxy(IBinder remote) {
3. mRemote = remote;
4. }
6. ......
8. **public** IBinder getService(String name) **throws** RemoteException {
9. Parcel data = Parcel.obtain();
10. Parcel reply = Parcel.obtain();
11. data.writeInterfaceToken(IServiceManager.descriptor);
12. data.writeString(name);
13. mRemote.transact(GET\_SERVICE\_TRANSACTION, data, reply, 0);
14. IBinder binder = reply.readStrongBinder();
15. reply.recycle();
16. data.recycle();
17. **return** binder;
18. }
20. ......
22. **private** IBinder mRemote;
23. }

         最终通过mRemote.transact来执行实际操作。我们在前面已经介绍过了，这里的mRemote实际上是一个BinderProxy对象，它的transact成员函数是一个JNI方法，实现在frameworks/base/core/jni/android\_util\_Binder.cpp文件中的android\_os\_BinderProxy\_transact函数中。

        这个函数前面我们已经看到了，这里就不再列出来了。不过，当这个函数从：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

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

       这里的reply变量里面就包括了一个HelloService的引用了。注意，这里的reply变量就是我们在ServiceManagerProxy.getService函数里面传进来的参数reply，它是一个Parcel对象。

       回到ServiceManagerProxy.getService函数中，从下面语句返回：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. mRemote.transact(GET\_SERVICE\_TRANSACTION, data, reply, 0);

       接着，就通过下面语句将这个HelloService的引用读出来：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. IBinder binder = reply.readStrongBinder();

       我们看看Parcel.readStrongBinder的实现：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **final** **class** Parcel {
2. ......
4. /\*\*
5. \* Read an object from the parcel at the current dataPosition().
6. \*/
7. **public** **final** **native** IBinder readStrongBinder();
9. ......
10. }

        它也是一个JNI方法，实现在frameworks/base/core/jni/android\_util\_Binder.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **static** jobject android\_os\_Parcel\_readStrongBinder(JNIEnv\* env, jobject clazz)
2. {
3. Parcel\* parcel = parcelForJavaObject(env, clazz);
4. **if** (parcel != NULL) {
5. **return** javaObjectForIBinder(env, parcel->readStrongBinder());
6. }
7. **return** NULL;
8. }

       这里首先把Java语言实现的Parcel对象class转换成C++语言实现的Parcel对象parcel，接着，通过parcel->readStrongBinder函数来获得一个Binder引用。

       我们在前面学习Binder机制时，在[Android系统进程间通信（IPC）机制Binder中的Client获得Server远程接口过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6633311)这篇文章中，曾经分析过这个函数，它最终返回来的是一个BpBinder对象，因此，下面的语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **return** javaObjectForIBinder(env, parcel->readStrongBinder());

       就相当于是：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **return** javaObjectForIBinder(env, **new** BpBinder(handle));

       这里的handle就是HelloService这个Binder实体在Client进程中的句柄了，它是由Binder驱动程序设置的，上层不用关心它的值具体是多少。至于javaObjectForIBinder这个函数，我们前面介绍如何获取Service Manager的Java远程接口时已经有详细介绍，这里就不累述了，它的作用就是创建一个BinderProxy对象，并且把刚才获得的BpBinder对象的地址保存在这个BinderProxy对象的mObject成员变量中。

       最后返回到Hello.onCreate函数中，从下面语句返回：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. helloService = IHelloService.Stub.asInterface(
2. ServiceManager.getService("hello"));

      就相当于是：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. helloService = IHelloService.Stub.asInterface(**new** BinderProxy()));

      回忆一下前面介绍IHelloService接口的定义时，IHelloService.Stub.asInterface是这样定义的：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **interface** IHelloService **extends** android.os.IInterface
2. {
3. /\*\* Local-side IPC implementation stub class. \*/
4. **public** **static** **abstract** **class** Stub **extends** android.os.Binder **implements** android.os.IHelloService
5. {
6. ......
8. **public** **static** android.os.IHelloService asInterface(android.os.IBinder obj)
9. {
10. **if** ((obj==**null**)) {
11. **return** **null**;
12. }
13. android.os.IInterface iin = (android.os.IInterface)obj.queryLocalInterface(DESCRIPTOR);
14. **if** (((iin!=**null**)&&(iin **instanceof** android.os.IHelloService))) {
15. **return** ((android.os.IHelloService)iin);
16. }
17. **return** **new** android.os.IHelloService.Stub.Proxy(obj);
18. }
20. ......
21. }
22. }

        这里的obj是一个BinderProxy对象，它的queryLocalInterface返回null，于是调用下面语句获得HelloService的远程接口：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **return** **new** android.os.IHelloService.Stub.Proxy(obj);

        相当于是：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **return** **new** android.os.IHelloService.Stub.Proxy(**new** BinderProxy());

        这样，我们就获得了HelloService的远程接口了，它实质上是一个实现了IHelloService接口的IHelloService.Stub.Proxy对象。

        五. Client通过HelloService的Java远程接口来使用HelloService提供的服务的过程

        上面介绍的Hello这个Activity获得了HelloService的远程接口后，就可以使用它的服务了。

        我们以使用IHelloService.getVal函数为例详细说明。在Hello::onClick函数中调用了IHelloService.getVal函数：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **class** Hello **extends** Activity **implements** OnClickListener {
2. ......
4. @Override
5. **public** **void** onClick(View v) {
6. **if**(v.equals(readButton)) {
7. **int** val = helloService.getVal();
8. ......
9. }
10. **else** **if**(v.equals(writeButton)) {
11. ......
12. }
13. **else** **if**(v.equals(clearButton)) {
14. ......
15. }
16. }
18. ......
19. }

        通知前面的分析，我们知道，这里的helloService接口实际上是一个IHelloService.Stub.Proxy对象，因此，我们进入到IHelloService.Stub.Proxy类的getVal函数中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **interface** IHelloService **extends** android.os.IInterface
2. {
3. /\*\* Local-side IPC implementation stub class. \*/
4. **public** **static** **abstract** **class** Stub **extends** android.os.Binder **implements** android.os.IHelloService
5. {
7. ......
9. **private** **static** **class** Proxy **implements** android.os.IHelloService
10. {
11. **private** android.os.IBinder mRemote;
13. ......
15. **public** **int** getVal() **throws** android.os.RemoteException
16. {
17. android.os.Parcel \_data = android.os.Parcel.obtain();
18. android.os.Parcel \_reply = android.os.Parcel.obtain();
19. **int** \_result;
20. **try** {
21. \_data.writeInterfaceToken(DESCRIPTOR);
22. mRemote.transact(Stub.TRANSACTION\_getVal, \_data, \_reply, 0);
23. \_reply.readException();
24. \_result = \_reply.readInt();
25. }
26. **finally** {
27. \_reply.recycle();
28. \_data.recycle();
29. }
30. **return** \_result;
31. }
32. }
34. ......
35. **static** **final** **int** TRANSACTION\_getVal = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 1);
36. }
38. ......
39. }

        这里我们可以看出，实际上是通过mRemote.transact来请求HelloService执行TRANSACTION\_getVal操作。这里的mRemote是一个BinderProxy对象，这是我们在前面获取HelloService的Java远程接口的过程中创建的。

        BinderProxy.transact函数是一个JNI方法，我们在前面已经介绍过了，这里不再累述。最过调用到Binder驱动程序，Binder驱动程序唤醒HelloService这个Server。前面我们在介绍HelloService的启动过程时，曾经提到，HelloService这个Server线程被唤醒之后，就会调用JavaBBinder类的onTransact函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **class** JavaBBinder : **public** BBinder
2. {
3. JavaBBinder(JNIEnv\* env, jobject object)
4. : mVM(jnienv\_to\_javavm(env)), mObject(env->NewGlobalRef(object))
5. {
6. ......
7. }
9. ......
11. **virtual** status\_t onTransact(
12. uint32\_t code, **const** Parcel& data, Parcel\* reply, uint32\_t flags = 0)
13. {
14. JNIEnv\* env = javavm\_to\_jnienv(mVM);
16. ......
18. jboolean res = env->CallBooleanMethod(mObject, gBinderOffsets.mExecTransact,
19. code, (int32\_t)&data, (int32\_t)reply, flags);
21. ......
23. **return** res != JNI\_FALSE ? NO\_ERROR : UNKNOWN\_TRANSACTION;
24. }
26. ......
28. JavaVM\* **const**   mVM;
29. jobject **const**   mObject;
30. };

         前面我们在介绍HelloService的启动过程时，曾经介绍过，JavaBBinder类里面的成员变量mObject就是HelloService类的一个实例对象了。因此，这里通过语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. jboolean res = env->CallBooleanMethod(mObject, gBinderOffsets.mExecTransact,
2. code, (int32\_t)&data, (int32\_t)reply, flags);

         就调用了HelloService.execTransact函数，而HelloService.execTransact函数继承了Binder类的execTransact函数：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **class** Binder **implements** IBinder {
2. ......
4. // Entry point from android\_util\_Binder.cpp's onTransact
5. **private** **boolean** execTransact(**int** code, **int** dataObj, **int** replyObj, **int** flags) {
6. Parcel data = Parcel.obtain(dataObj);
7. Parcel reply = Parcel.obtain(replyObj);
8. // theoretically, we should call transact, which will call onTransact,
9. // but all that does is rewind it, and we just got these from an IPC,
10. // so we'll just call it directly.
11. **boolean** res;
12. **try** {
13. res = onTransact(code, data, reply, flags);
14. } **catch** (RemoteException e) {
15. reply.writeException(e);
16. res = **true**;
17. } **catch** (RuntimeException e) {
18. reply.writeException(e);
19. res = **true**;
20. } **catch** (OutOfMemoryError e) {
21. RuntimeException re = **new** RuntimeException("Out of memory", e);
22. reply.writeException(re);
23. res = **true**;
24. }
25. reply.recycle();
26. data.recycle();
27. **return** res;
28. }
29. }

         这里又调用了onTransact函数来作进一步处理。由于HelloService类继承了IHelloService.Stub类，而IHelloService.Stub类实现了onTransact函数，HelloService类没有实现，因此，最终调用了IHelloService.Stub.onTransact函数：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **interface** IHelloService **extends** android.os.IInterface
2. {
3. /\*\* Local-side IPC implementation stub class. \*/
4. **public** **static** **abstract** **class** Stub **extends** android.os.Binder **implements** android.os.IHelloService
5. {
6. ......
8. @Override
9. **public** **boolean** onTransact(**int** code, android.os.Parcel data, android.os.Parcel reply, **int** flags) **throws** android.os.RemoteException
10. {
11. **switch** (code)
12. {
13. ......
14. **case** TRANSACTION\_getVal:
15. {
16. data.enforceInterface(DESCRIPTOR);
17. **int** \_result = **this**.getVal();
18. reply.writeNoException();
19. reply.writeInt(\_result);
20. **return** **true**;
21. }
22. }
23. **return** **super**.onTransact(code, data, reply, flags);
24. }
26. ......
28. }
29. }

         函数最终又调用了HelloService.getVal函数：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. **public** **class** HelloService **extends** IHelloService.Stub {
2. ......
4. **public** **int** getVal() {
5. **return** getVal\_native();
6. }
8. ......
9. **private** **static** **native** **int** getVal\_native();
10. }

       最终，经过层层返回，就回到IHelloService.Stub.Proxy.getVal函数中来了，从下面语句返回：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. mRemote.transact(Stub.TRANSACTION\_getVal, \_data, \_reply, 0);

       并将结果读出来：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6642463)

1. \_result = \_reply.readInt();

       最后将这个结果返回到Hello.onClick函数中。

       这样，Client通过HelloService的Java远程接口来使用HelloService提供的服务的过程就介绍完了。

       至此，Android系统进程间通信Binder机制在应用程序框架层的Java接口源代码分析也完成了，整个Binder机制的学习就结束了。

       重新学习Android系统进程间通信Binder机制，请回到[Android进程间通信（IPC）机制Binder简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6618363)一文。