Weishu's Notes

Binder学习指南

发表于 2016-01-12 | 55555次阅读

毫不夸张地说，Binder是Android系统中最重要的特性之一；正如其名“粘合剂”所喻，它是系统间各个组件的桥梁，Android系统的开放式设计也很大程度上得益于这种及其方便的跨进程通信机制。

理解Binder对于理解整个Android系统有着非常重要的作用，Android系统的四大组件，AMS，PMS等系统服务无一不与Binder挂钩；如果对Binder不甚了解，那么就很难了解这些系统机制，从而仅仅浮游与表面，不懂Binder你都不好意思说自己会Android开发；要深入Android，Binder是必须迈出的一步。

现在网上有不少资料介绍Binder，个人觉得最好的两篇如下：

1. Binder设计与实现
2. Android进程间通信（IPC）机制Binder简要介绍和学习计划系列

其中，《Binder设计与实现》以一种宏观的角度解释了Android系统中的Binder机制，文章如行云流水；如果对于Binder有一定的了解再来看着篇文章，有一种打通任督二脉的感觉；每看一次理解就深一层。老罗的系列文章则从系统源码角度深入分析了Binder的实现细节；具有很大的参考意义；每当对于Binder细节有疑惑，看一看他的书就迎刃而解。

但是遗憾的是，Binder机制终究不是三言两语就能解释清楚的，一上来就扒出源码很可能深陷细节无法自拔，老罗的文章那不是一般的长，如果看不懂强行看很容易睡着；勉强看完还是云里雾里；相反如果直接大谈特谈Binder的设计，那么完全就是不知所云；因此上述两篇文章对于初学者并不友好，本文不会深入源码细节，也不会对于Binder的设计高谈阔论；重点如下：

1. 一些Linux的预备知识
2. Binder到底是什么？
3. Binder机制是如何跨进程的？
4. 一次Binder通信的基本流程是什么样？
5. 深入理解Java层的Binder

读完本文，你应该对于Java层的AIDL了如指掌，对于Binder也会有一个大体上的认识；再深入学习就得靠自己了，本人推荐的Binder学习路径如下：

1. 先学会熟练使用AIDL进行跨进程通信（简单来说就是远程Service）
2. 看完本文
3. 看Android文档， `Parcel, IBinder, Binder` 等涉及到跨进程通信的类
4. 不依赖AIDL工具，手写远程Service完成跨进程通信

5. 看《Binder设计与实现》
6. 看老罗的博客或者书（书结构更清晰）
7. 再看《Binder设计与实现》
8. 学习Linux系统相关知识；自己看源码。

背景知识

为了理解Binder我们先澄清一些概念。为什么需要跨进程通信（IPC），怎么做到跨进程通信？为什么是Binder？

由于Android系统基于Linux内核，因此有必要了解相关知识。

进程隔离

> 进程隔离是为保护操作系统中进程互不干扰而设计的一组不同硬件和软件的技术。这个技术是为了避免进程A写入进程B的情况发生。 进程的隔离实现，使用了虚拟地址空间。进程A的虚拟地址和进程B的虚拟地址不同，这样就防止进程A将数据信息写入进程B。

以上来自维基百科；操作系统的不同进程之间，数据不共享；对于每个进程来说，它都天真地以为自己独享了整个系统，完全不知道其他进程的存在；(有关虚拟地址，请自行查阅）因此一个进程需要与另外一个进程通信，需要某种系统机制才能完成。

用户空间/内核空间

详细解释可以参考Kernel Space Definition；简单理解如下：

Linux Kernel是操作系统的核心，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。

对于Kernel这么一个高安全级别的东西，显然是不容许其它的应用程序随便调用或访问的，所以需要对Kernel提供一定的保护机制，这个保护机制用来告诉那些应用程序，你只可以访问某些许可的资源，不许可的资源是拒绝被访问的，于是就把Kernel和上层的应用程序抽像的隔离开，分别称之为Kernel Space和User Space。

系统调用/内核态/用户态

虽然从逻辑上抽离出用户空间和内核空间；但是不可避免的的是，总有那么一些用户空间需要访问内核的资源；比如应用程序访问文件，网络是很常见的事情，怎么办呢？

> Kernel space can be accessed by user processes only through the use of system calls.

用户空间访问内核空间的唯一方式就是**系统调用**；通过这个统一入口接口，所有的资源访问都是在内核的控制下执行，以免导致对用户程序对系统资源的越权访问，从而保障了系统的安全和稳定。用户软件良莠不齐，要是它们乱搞把系统玩坏了怎么办？因此对于某些特权操作必须交给安全可靠的内核来执行。

当一个任务（进程）执行系统调用而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）此时处理器处于特权级最高的（0级）内核代码中执行。当进程在执行用户自己的代码时，则称其处于用

户运行态（用户态）。即此时处理器在特权级最低的（3级）用户代码中运行。处理器在特权等级高的时候才能执行那些特权CPU指令。

内核模块/驱动

通过系统调用，用户空间可以访问内核空间，那么如果一个用户空间想与另外一个用户空间进行通信怎么办呢？很自然想到的是让操作系统内核添加支持；传统的Linux通信机制，比如Socket，管道等都是内核支持的；但是Binder并不是Linux内核的一部分，它是怎么做到访问内核空间的呢？Linux的动态可加载内核模块（Loadable Kernel Module，LKM）机制解决了这个问题；模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间运行。这样，Android系统可以通过添加一个内核模块运行在内核空间，用户进程之间的通过这个模块作为桥梁，就可以完成通信了。

在Android系统中，这个运行在内核空间的，负责各个用户进程通过Binder通信的内核模块叫做**Binder驱动**;

> 驱动程序一般指的是设备驱动程序（Device Driver），是一种可以使计算机和设备通信的特殊程序。相当于硬件的接口，操作系统只有通过这个接口，才能控制硬件设备的工作；

驱动就是操作硬件的接口，为了支持Binder通信过程，Binder使用了一种“硬件”，因此这个模块被称之为驱动。

好了，说了这么多枯燥的概念，看张美图缓解一下。

Android使用的Linux内核拥有着非常多的跨进程通信机制，比如管道，System V，Socket等；为什么还需要单独搞一个Binder出来呢？主要有两点，性能和安全。在移动设备上，广泛地使用跨进程通信肯定对通信机制本身提出了严格的要求；Binder相对出传统的Socket方式，更加高效；另外，传统的进程通信方式对于通信双方的身份并没有做出严格的验证，只有在上层协议上进行架设；比如Socket通信ip地址是客户端手动填入的，都可以进行伪造；而Binder机制从协议本身就支持对通信双方做身份校检，因而大大提升了安全性。这个也是Android权限模型的基础。

Binder通信模型

对于跨进程通信的双方，我们姑且叫做Server进程（简称Server），Client进程（简称Client）；由于进程隔离的存在，它们之间没办法通过简单的方式进行通信，那么Binder机制是如何进行的呢？

回想一下日常生活中我们通信的过程：假设A和B要进行通信，通信的媒介是打电话（A是Client，B是Server）；A要给B打电话，必须知道B的号码，这个号码怎么获取呢？**通信录**.

这个通信录就是一张表；内容大致是：

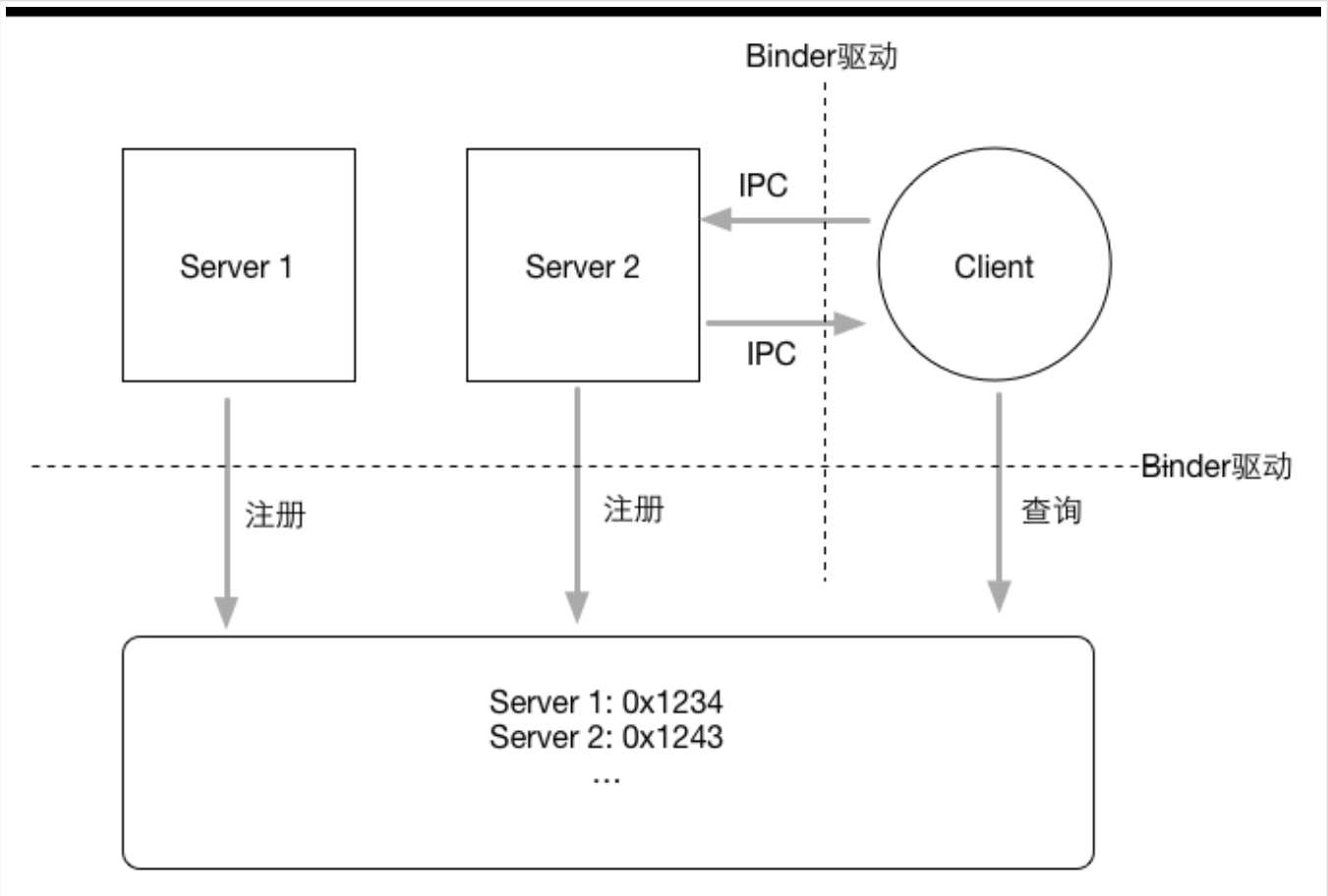
```
1  B -> 12345676
2  C -> 12334354
```

先查阅通信录，拿到B的号码；才能进行通信；否则，怎么知道应该拨什么号码？回想一下古老的电话机，如果A要给B打电话，必须先连接通话中心，说明给我接通B的电话；这时候通话中心帮他呼叫B；连接建立，就完成了通信。

另外，光有电话和通信录是不可能完成通信的，没有基站支持；信息根本无法传达。

我们看到，一次电话通信的过程除了通信的双方还有两个隐藏角色：通信录和基站。Binder通信机制也是一样：两个运行在用户空间的进程要完成通信，必须借助内核的帮助，这个运行在内核里面的程序叫做**Binder驱动**，它的功能类似于基站；通信录呢，就是一个叫做**ServiceManager**的东西（简称SM）

OK，Binder的通信模型就是这么简单，如下图：



整个通信步骤如下：

1. SM建立(建立通信录)；首先有一个进程向驱动提出申请为SM；驱动同意之后，SM进程负责管理Service（注意这里是Service而不是Server，因为如果通信过程反过来的话，那么原来的客户端Client也会成为服务端Server）不过这时候通信录还是空的，一个号码都没有。
2. 各个Server向SM注册(完善通信录)；每个Server端进程启动之后，向SM报告，我是zhangsan, 要找我请返回0x1234(这个地址没有实际意义，类比)；其他Server进程依次如此；这样SM就建立了一张表，对应着各个Server的名字和地址；就好比B与A见面了，说存个我的号码吧，以后找我拨打10086；
3. Client想要与Server通信，首先询问SM；请告诉我如何联系zhangsan，SM收到后给他一个号码0x1234；Client收到之后，开心滴用这个号码拨通了Server的电话，于是就开始通信了。

那么Binder驱动干什么去了呢？这里Client与SM的通信，以及Client与Server的通信，都会经过驱动，驱动在背后默默无闻，但是做着最重要的工作。驱动是整个通信过程的核心，因此完成跨进程通信的秘密全部隐藏在驱动里面；这个我们稍后讨论。

OK，上面就是整个Binder通信的基本模型；做了一个简单的类比，当然也有一些不恰当的地方，(比如通信录现实中每个人都有一个，但是SM整个系统只有一个；基站也有很多个，但是驱动只有一个）；但是整体上就是这样的；我们看到其实整个通信模型非常简单。

Binder机制跨进程原理

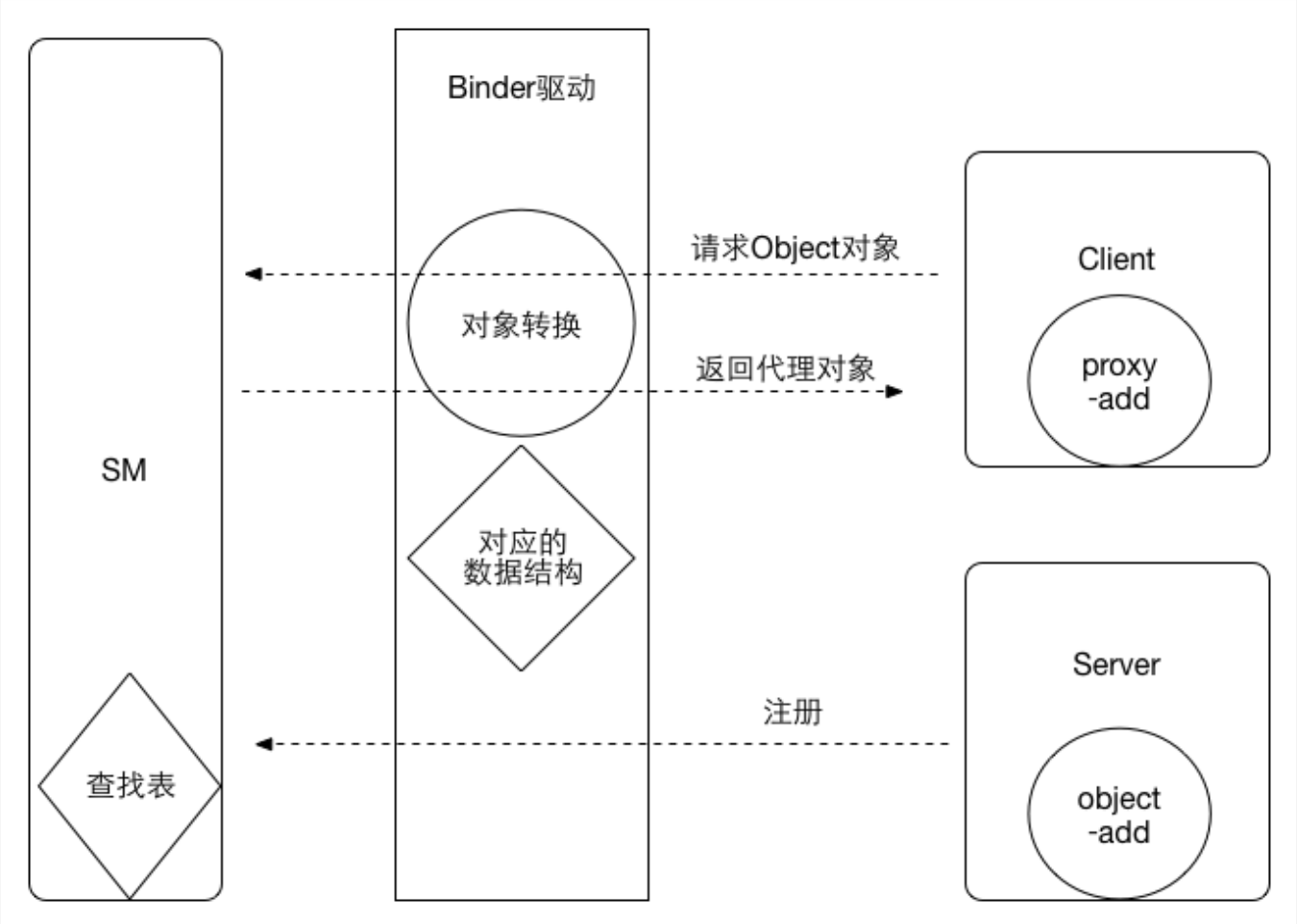
上文给出了Binder的通信模型，指出了通信过程的四个角色: Client, Server, SM, driver; 但是我们仍然不清楚**Client到底是如何与Server完成通信的**。

两个运行在用户空间的进程A和进程B如何完成通信呢？内核可以访问A和B的所有数据；所以，最简单的方式是通过内核做中转；假设进程A要给进程B发送数据，那么就先把A的数据copy到内核空间，然后把内核空间对应的数据copy到B就完成了；用户空间要操作内核空间，需要通过系统调用；刚好，这里就有两个系统调用： `copy_from_user`，`copy_to_user` 。

但是，Binder机制并不是这么干的。讲这么一段，是说明进程间通信并不是什么神秘的东西。那么，Binder机制是如何实现跨进程通信的呢？

Binder驱动为我们做了一切。

假设Client进程想要调用Server进程的 `object` 对象的一个方法 `add` ;对于这个跨进程通信过程，我们来看看Binder机制是如何做的。（通信是一个广泛的概念，只要一个进程能调用另外一个进程里面某对象的方法，那么具体要完成什么通信内容就很容易了。）



Alt text

首先，Server进程要向SM注册；告诉自己是谁，自己有什么能力；在这个场景就是Server告诉SM，它叫 `zhangsan` ，它有一个 `object` 对象，可以执行 `add` 操作；于是SM建立了一张表： `zhangsan` 这个名字对应进程Server;

然后Client向SM查询：我需要联系一个名字叫做 `zhangsan` 的进程里面的 `object` 对象；这时候关键来了：进程之间通信的数据都会经过运行在内核空间里面的驱动，驱动在数据流过的时候做了一点手脚，它并不会给

Client进程返回一个真正的 `object` 对象，而是返回一个看起来跟 `object` 一模一样的代理对象 `objectProxy` ，这个 `objectProxy` 也有一个 `add` 方法，但是这个 `add` 方法没有Server进程里面 `object` 对象的 `add` 方法那个能力； `objectProxy` 的 `add` 只是一个傀儡，它唯一做的事情就是把参数包装然后交给驱动。(这里我们简化了SM的流程，见下文)

但是Client进程并不知道驱动返回给它的对象动过手脚，毕竟伪装的太像了，如假包换。Client开开心心地拿着 `objectProxy` 对象然后调用 `add` 方法；我们说过，这个 `add` 什么也不做，直接把参数做一些包装然后直接转发给Binder驱动。

驱动收到这个消息，发现是这个 `objectProxy` ；一查表就明白了：我之前用 `objectProxy` 替换了 `object` 发送给Client了，它真正应该要访问的是 `object` 对象的 `add` 方法；于是Binder驱动通知Server进程，*调用你的object对象的 `add` 方法，然后把结果发给我*，Sever进程收到这个消息，照做之后将结果返回驱动，驱动然后把结果返回给 `Client` 进程；于是整个过程就完成了。

由于驱动返回的 `objectProxy` 与Server进程里面原始的 `object` 是如此相似，给人感觉好像是**直接把Server进程里面的对象object传递到了Client进程**；因此，我们可以说**Binder对象是可以进行跨进程传递的对象**

但事实上我们知道，Binder跨进程传输并不是真的把一个对象传输到了另外一个进程；传输过程好像是Binder跨进程穿越的时候，它在一个进程留下了一个真身，在另外一个进程幻化出一个影子（这个影子可以很多个）；Client进程的操作其实是对于影子的操作，影子利用Binder驱动最终让真身完成操作。

理解这一点非常重要；务必仔细体会。另外，Android系统实现这种机制使用的是*代理模式*, 对于Binder的访问，如果是在同一个进程（不需要跨进程），那么直接返回原始的Binder实体；如果在不同进程，那么就给他一个代理对象（影子）；我们在系统源码以及AIDL的生成代码里面可以看到很多这种实现。

另外我们为了简化整个流程，隐藏了SM这一部分驱动进行的操作；实际上，由于SM与Server通常不在一个进程，Server进程向SM注册的过程也是跨进程通信，驱动也会对这个过程进行暗箱操作：SM中存在的Server端的对象实际上也是代理对象，后面Client向SM查询的时候，驱动会给Client返回另外一个代理对象。Sever进程的本地对象仅有一个，其他进程所拥有的全部都是它的代理。

一句话总结就是：**Client进程只不过是持有了Server端的代理；代理对象协助驱动完成了跨进程通信。**

OK，该休息一下了。

Binder到底是什么？

我们经常提到Binder，那么Binder到底是什么呢？

Binder的设计采用了面向对象的思想，在Binder通信模型的四个角色里面；他们的代表都是“Binder”，这样，对于Binder通信的使用者而言，Server里面的Binder和Client里面的Binder没有什么不同，一个Binder对象就代表了所有，它不用关心实现的细节，甚至不用关心驱动以及SM的存在；这就是抽象。

- 通常意义下，Binder指的是一种通信机制；我们说AIDL使用Binder进行通信，指的就是**Binder这种IPC机制**。

- 对于Server进程来说，Binder指的是**Binder本地对象**
- 对于Client来说，Binder指的是**Binder代理对象**，它只是**Binder本地对象**的一个远程代理；对这个Binder代理对象的操作，会通过驱动最终转发到Binder本地对象上去完成；对于一个拥有Binder对象的使用者而言，它无须关心这是一个Binder代理对象还是Binder本地对象；对于代理对象的操作和对本地对象的操作对它来说没有区别。
- 对于传输过程而言，Binder是可以进行跨进程传递的对象；Binder驱动会对具有跨进程传递能力的对象做特殊处理：自动完成代理对象和本地对象的转换。

> 面向对象思想的引入将进程间通信转化为通过对某个Binder对象的引用调用该对象的方法，而其独特之处在于Binder对象是一个可以跨进程引用的对象，它的实体（本地对象）位于一个进程中，而它的引用（代理对象）却遍布于系统的各个进程之中。最诱人的是，这个引用和java里引用一样既可以是强类型，也可以是弱类型，而且可以从一个进程传给其它进程，让大家都能访问同一Server，就象将一个对象或引用赋值给另一个引用一样。Binder模糊了进程边界，淡化了进程间通信过程，整个系统仿佛运行于同一个面向对象的程序之中。形形色色的Binder对象以及星罗棋布的引用仿佛粘接各个应用程序的胶水，这也是Binder在英文里的原意。

驱动里面的Binder

我们现在知道，Server进程里面的Binder对象指的是Binder本地对象，Client里面的对象值得是Binder代理对象；在Binder对象进行跨进程传递的时候，Binder驱动会自动完成这两种类型的转换；因此Binder驱动必然保存了每一个跨越进程的Binder对象的相关信息；在驱动中，Binder本地对象的代表是一个叫做 `binder_node` 的数据结构，Binder代理对象是用 `binder_ref` 代表的；有的地方把Binder本地对象直接称作Binder实体，把Binder代理对象直接称作Binder引用（句柄），其实指的是Binder对象在驱动里面的表现形式；读者明白意思即可。

OK，现在大致了解Binder的通信模型，也了解了Binder这个对象在通信过程中各个组件里面到底表示的是什么。

深入理解Java层的Binder

IBinder/IInterface/Binder/BinderProxy/Stub

我们使用AIDL接口的时候，经常会接触到这些类，那么这每个类代表的是什么呢？

- IBinder是一个接口，它代表了**一种跨进程传输的能力**；只要实现了这个接口，就能将这个对象进行跨进程传递；这是驱动底层支持的；在跨进程数据流经驱动的时候，驱动会识别IBinder类型的数据，从而自动完成不同进程Binder本地对象以及Binder代理对象的转换。
- IBinder负责数据传输，那么client与server端的调用契约（这里不用接口避免混淆）呢？这里的IInterface代表的就是远程server对象具有什么能力。具体来说，就是aidl里面的接口。
- Java层的Binder类，代表的其实就是**Binder本地对象**。BinderProxy类是Binder类的一个内部类，它代表远程进程的Binder对象的本地代理；这两个类都继承自IBinder, 因而都具有跨进程传输的能力；实际上，在跨越进程的时候，Binder驱动会自动完成这两个对象的转换。
- 在使用AIDL的时候，编译工具会给我们生成一个Stub的静态内部类；这个类继承了Binder, 说明它是一个Binder本地对象，它实现了IInterface接口，表明它具有远程Server承诺给Client的能力；Stub是一个抽象类，具体的IInterface的相关实现需要我们手动完成，这里使用了策略模式。

AIDL过程分析

现在我们通过一个AIDL的使用，分析一下整个通信过程中，各个角色到底做了什么，AIDL到底是如何完成通信的。（如果你连AIDL都不熟悉，请先查阅官方文档）

首先定一个一个简单的aidl接口：

```
// ICompute.aidl
package com.example.test.app;
interface ICompute {
     int add(int a, int b);
}
```

然后用编译工具编译之后，可以得到对应的ICompute.java类，看看系统给我们生成的代码：

```
package com.example.test.app;

public interface ICompute extends android.os.IInterface {
    /**
     * Local-side IPC implementation stub class.
     */
    public static abstract class Stub extends android.os.Binder implements com.example.test.a
        private static final java.lang.String DESCRIPTOR = "com.example.test.app.ICompute";

        /**
         * Construct the stub at attach it to the interface.
         */
        public Stub() {
            this.attachInterface(this, DESCRIPTOR);
        }

        /**
         * Cast an IBinder object into an com.example.test.app.ICompute interface,
         * generating a proxy if needed.
         */
        public static com.example.test.app.ICompute asInterface(android.os.IBinder obj) {
            if ((obj == null)) {
                return null;
            }
            android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
            if (((iin != null) && (iin instanceof com.example.test.app.ICompute))) {
                return ((com.example.test.app.ICompute) iin);
            }
            return new com.example.test.app.ICompute.Stub.Proxy(obj);
        }

        @Override
        public android.os.IBinder asBinder() {
            return this;
        }
```

```
36
37          @Override
38          public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply,
39              switch (code) {
40                  case INTERFACE_TRANSACTION: {
41                      reply.writeString(DESCRIPTOR);
42                      return true;
43                  }
44                  case TRANSACTION_add: {
45                      data.enforceInterface(DESCRIPTOR);
46                      int _arg0;
47                      _arg0 = data.readInt();
48                      int _arg1;
49                      _arg1 = data.readInt();
50                      int _result = this.add(_arg0, _arg1);
51                      reply.writeNoException();
52                      reply.writeInt(_result);
53                      return true;
54                  }
55              }
56              return super.onTransact(code, data, reply, flags);
57          }
58
59          private static class Proxy implements com.example.test.app.ICompute {
60              private android.os.IBinder mRemote;
61
62              Proxy(android.os.IBinder remote) {
63                  mRemote = remote;
64              }
65
66              @Override
67              public android.os.IBinder asBinder() {
68                  return mRemote;
69              }
70
71              public java.lang.String getInterfaceDescriptor() {
72                  return DESCRIPTOR;
73              }
74
75              /**
76               * Demonstrates some basic types that you can use as parameters
77               * and return values in AIDL.
78               */
79              @Override
80              public int add(int a, int b) throws android.os.RemoteException {
81                  android.os.Parcel _data = android.os.Parcel.obtain();
82                  android.os.Parcel _reply = android.os.Parcel.obtain();
83                  int _result;
84                  try {
85                      _data.writeInterfaceToken(DESCRIPTOR);
86                      _data.writeInt(a);
87                      _data.writeInt(b);
```

```
88                     mRemote.transact(Stub.TRANSACTION_add, _data, _reply, 0);
89                     _reply.readException();
90                     _result = _reply.readInt();
91                 } finally {
92                     _reply.recycle();
93                     _data.recycle();
94                 }
95                 return _result;
96             }
97         }
98 
99         static final int TRANSACTION_add = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
100    }
101
102    /**
103     * Demonstrates some basic types that you can use as parameters
104     * and return values in AIDL.
105     */
106    public int add(int a, int b) throws android.os.RemoteException;
107 }
```

系统帮我们生成了这个文件之后，我们只需要继承ICompute.Stub这个抽象类，实现它的方法，然后在Service 的onBind方法里面返回就实现了AIDL。这个Stub类非常重要，具体看看它做了什么。

Stub类继承自Binder，意味着这个Stub其实自己是一个Binder本地对象，然后实现了ICompute接口，ICompute本身是一个IInterface，因此他携带某种客户端需要的能力（这里是方法 add）。此类有一个内部类Proxy，也就是Binder代理对象；

然后看看 asInterface 方法，我们在bind一个Service之后，在onServiceConnecttion的回调里面，就是通过这个方法拿到一个远程的service的，这个方法做了什么呢？

```
1  /**
2   * Cast an IBinder object into an com.example.test.app.ICompute interface,
3   * generating a proxy if needed.
4   */
5  public static com.example.test.app.ICompute asInterface(android.os.IBinder obj) {
6      if ((obj == null)) {
7          return null;
8      }
9      android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
10     if (((iin != null) && (iin instanceof com.example.test.app.ICompute))) {
11         return ((com.example.test.app.ICompute) iin);
12     }
13     return new com.example.test.app.ICompute.Stub.Proxy(obj);
14 }
```

首先看函数的参数 IBinder 类型的obj，这个对象是驱动给我们的，如果是Binder本地对象，那么它就是Binder类型，如果是Binder代理对象，那就是 BinderProxy 类型；然后，正如上面自动生成的文档所说，它会试着查找Binder本地对象，如果找到，说明Client和Server都在同一个进程，这个参数直接就是本地对象，

直接强制类型转换然后返回，如果找不到，说明是远程对象（处于另外一个进程）那么就需要创建一个Binde代理对象，让这个Binder代理实现对于远程对象的访问。一般来说，如果是与一个远程Service对象进行通信，那么这里返回的一定是一个Binder代理对象，这个IBinder参数的实际上是BinderProxy;

再看看我们对于aidl的 add 方法的实现；在Stub类里面， add 是一个抽象方法，我们需要继承这个类并实现它；如果Client和Server在同一个进程，那么直接就是调用这个方法；那么，如果是远程调用，这中间发生了什么呢？Client是如何调用到Server的方法的？

我们知道，对于远程方法的调用，是通过Binder代理完成的，在这个例子里面就是 Proxy 类； Proxy 对于 add 方法的实现如下：

```
Override
public int add(int a, int b) throws android.os.RemoteException {
    android.os.Parcel _data = android.os.Parcel.obtain();
    android.os.Parcel _reply = android.os.Parcel.obtain();
    int _result;
    try {
        _data.writeInterfaceToken(DESCRIPTOR);
        _data.writeInt(a);
        _data.writeInt(b);
        mRemote.transact(Stub.TRANSACTION_add, _data, _reply, 0);
        _reply.readException();
        _result = _reply.readInt();
    } finally {
        _reply.recycle();
        _data.recycle();
    }
    return _result;
}
```

它首先用 Parcel 把数据序列化了，然后调用了 transact 方法；这个 transact 到底做了什么呢？这个 Proxy 类在 asInterface 方法里面被创建，前面提到过，如果是Binder代理那么说明驱动返回的IBinder实际是 BinderProxy，因此我们的 Proxy 类里面的 mRemote 实际类型应该是 BinderProxy ；我们看看 BinderProxy 的 transact 方法：(Binder.java的内部类)

```
public native boolean transact(int code, Parcel data, Parcel reply,
            int flags) throws RemoteException;
```

这是一个本地方法；它的实现在native层，具体来说在frameworks/base/core/jni/android_util_Binder.cpp文件，里面进行了一系列的函数调用，调用链实在太长这里就不给出了；要知道的是它最终调用到了 talkWithDriver 函数；看这个函数的名字就知道，通信过程要交给驱动完成了；这个函数最后通过 ioctl 系统调用，Client进程陷入内核态，Client调用 add 方法的线程挂起等待返回；驱动完成一系列的操作之后唤醒Server进程，调用了Server进程本地对象的 onTransact 函数（实际上由Server端线程池完成）。我们再看Binder本地对象的 onTransact 方法（这里就是 Stub 类里面的此方法）：

```
@Override
public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags
```

```
    switch (code) {
        case INTERFACE_TRANSACTION: {
            reply.writeString(DESCRIPTOR);
            return true;
        }
        case TRANSACTION_add: {
            data.enforceInterface(DESCRIPTOR);
            int _arg0;
            _arg0 = data.readInt();
            int _arg1;
            _arg1 = data.readInt();
            int _result = this.add(_arg0, _arg1);
            reply.writeNoException();
            reply.writeInt(_result);
            return true;
        }
    }
    return super.onTransact(code, data, reply, flags);
}
```

在Server进程里面， `onTransact` 根据调用号（每个AIDL函数都有一个编号，在跨进程的时候，不会传递函数，而是传递编号指明调用哪个函数）调用相关函数；在这个例子里面，调用了Binder本地对象的 add 方法；这个方法将结果返回给驱动，驱动唤醒挂起的Client进程里面的线程并将结果返回。于是一次跨进程调用就完成了。

至此，你应该对AIDL这种通信方式里面的各个类以及各个角色有了一定的了解；它总是那么一种固定的模式：一个需要跨进程传递的对象一定继承自IBinder，如果是Binder本地对象，那么一定继承Binder实现IInterface，如果是代理对象，那么就实现了IInterface并持有了IBinder引用；

Proxy与Stub不一样，虽然他们都既是Binder又是IInterface，不同的是Stub采用的是继承（is 关系），Proxy采用的是组合（has 关系）。他们均实现了所有的IInterface函数，不同的是，Stub又使用策略模式调用的是虚函数（待子类实现），而Proxy则使用组合模式。为什么Stub采用继承而Proxy采用组合？事实上，Stub本身is一个IBinder（Binder），它本身就是一个能跨越进程边界传输的对象，所以它得继承IBinder实现transact这个函数从而得到跨越进程的能力（这个能力由驱动赋予）。Proxy类使用组合，是因为他不关心自己是什么，它也不需要跨越进程传输，它只需要拥有这个能力即可，要拥有这个能力，只需要保留一个对IBinder的引用。如果把这个过程做一个类比，在封建社会，Stub好比皇帝，可以号令天下，他生而具有这个权利（不要说宣扬封建迷信。。）如果一个人也想号令天下，可以，“挟天子以令诸侯”。为什么不自己去当皇帝， 一，一般情况没必要，当了皇帝 实限制也蛮多的是不是？我现在既能掌管天下，又能不受约束（Java单继承）；二，名不正言不顺啊，我本来特么就不是（Binder），你非要我是说不过去，搞不好还会造反。最后呢，如果想当皇帝也可以，那就是asBinder了。在Stub类里面，asBinder返回this，在Proxy里面返回的是持有的组合类IBinder的引用。

再去翻阅系统的ActivityManagerServer的源码，就知道哪一个类是什么角色了：IActivityManager是一个IInterface，它代表远程Service具有什么能力，ActivityManagerNative指的是Binder本地对象（类似AIDL工具生成的Stub类），这个类是抽象类，它的实现是 `ActivityManagerService` ；因此对于AMS的最终操作都会进入 `ActivityManagerService` 这个真正实现；同时如果仔细观察，ActivityManagerNative.java里面有一个非公开类ActivityManagerProxy, 它代表的就是Binder代理对象；是不是跟AIDL模型一模一样呢？那么

`ActivityManager` 是什么？他不过是一个管理类而已，可以看到真正的操作都是转发给 `ActivityManagerNative` 进而交给他的实现 `ActivityManagerService` 完成的。

OK，本文就讲到这里了，要深入理解Binder，需要自己下功夫；那些native层以及驱动里面的调用过程，用文章写出来根本没有意义，需要自己去跟踪；接下来你可以：

1. 看Android文档， `Parcel, IBinder, Binder` 等涉及到跨进程通信的类；
2. 不依赖AIDL工具，手写远程Service完成跨进程通信
3. 看《Binder设计与实现》
4. 看老罗的博客或者书（书结构更清晰）
5. 再看《Binder设计与实现》
6. 学习Linux系统相关知识；自己看源码。

#android #binder

你真的了解AsyncTask？ ASCII Art：使用纯文本流程图