Android's best understanding of the Binder mechanism

Reprinted: http://weishu.me/2016/01/12/binder-index-for-newer/

Binder study guide

Published in 2016-01-12 | 34011 readings

It is no exaggeration to say that Binder is one of the most important features of the Android system; as its name is "adhesive", it is a bridge between various components of the system, and the open design of the Android system is also largely Benefit from this and its convenient cross-process communication mechanism.

Understanding Binder has a very important role in understanding the entire Android system. The four components of the Android system, AMS, PMS and other system services are all linked to Binder. If you don't know Binder well, it is difficult to understand these system mechanisms. Just floating on the surface and not knowing Binder, you are embarrassed to say that you will develop Android; to penetrate Android, Binder is a step that must be taken.

Now there are a lot of information on the Internet to introduce Binder. I think the best two are as follows:

Among them, "Binder Design and Implementation" explains the Binder mechanism in the Android system from a macroscopic point of view. The article is like a flowing stream; if you have a certain understanding of Binder and then look at the article, there is a feeling of getting through the second line of the governor; Every time you look at it, you will be deeper. Lao Luo's series of articles deeply analyzes the implementation details of Binder from the perspective of system source code; it has great reference significance; whenever there is doubt about the details of Binder, a look at his book will be solved.

But unfortunately, the Binder mechanism can not be explained clearly in a few words. When you come up, the source code is likely to be deeply immersed in details. Lao Luo’s article is not generally long. If you don’t understand it, it is easy to fall asleep. If you just read the Binder design directly, then it is completely unintelligible; therefore, the above two articles are not friendly to beginners, this article will not go into the source details, nor will it The design of Binder is very talkative; the key points are as follows:

1. Some Linux prerequisites
2. What exactly is Binder?
3. How is the Binder mechanism cross-process?
4. What is the basic flow of a Binder communication?
5. In-depth understanding of the Binder of the Java layer

After reading this article, you should know the AIDL of the Java layer, and you will have a general understanding of Binder. If you go deeper, you have to rely on yourself. The Binder learning path recommended by me is as follows:

1. First learn to use AIDL for cross-process communication (in short, remote service)
2. Read this article
3. Look at Android documentation, `Parcel, IBinder, Binder` classes that involve cross-process communication
4. Do not rely on AIDL tools, handwritten remote service to complete cross-process communication
5. See "Binder Design and Implementation"
6. Look at Lao Luo's blog or book (the book structure is clearer)
7. Look at "Binder Design and Implementation"
8. Learn Linux system related knowledge; see the source code yourself.

background knowledge

In order to understand Binder we first clarify some concepts. Why do you need cross-process communication (IPC), how do you communicate across processes? Why is Binder?

Since the Android system is based on the Linux kernel, it is necessary to understand the relevant knowledge.

Process isolation

> Process isolation is a set of different hardware and software technologies designed to protect processes in the operating system from interference. This technique is to avoid the situation where process A writes to process B. The isolation implementation of the process uses a virtual address space. The virtual address of process A is different from the virtual address of process B, thus preventing process A from writing data information to process B.

The above is from Wikipedia; data is not shared between different processes of the operating system; for each process, it is naive to think that it has exclusive access to the entire system, completely unaware of the existence of other processes; (about virtual address, please Self-referencing) Therefore, one process needs to communicate with another process and needs some system mechanism to complete.

User space/kernel space

For a detailed explanation, please refer to the Kernel Space Definition ; simply understand the following:

The Linux Kernel is the core of the operating system. It is independent of normal applications and has access to protected memory space as well as access to all underlying hardware devices.

For Kernel, such a high security level, obviously does not allow other applications to call or access casually, so you need to provide some protection mechanism for Kernel. This protection mechanism is used to tell those applications, you can only access certain Licensed resources, unlicensed resources are refused to be accessed, so Kernel and the upper application are isolated, called Kernel Space and User Space.

System call / kernel mode / user mode

Although logically extracting user space and kernel space; but inevitably, there are always some user space that needs to access the kernel's resources; for example, the application accesses files, the network is a very common thing, what should I do?

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

The only way for user space to access kernel space is through **system calls** . Through this unified ingress interface, all resource access is performed under the control of the kernel, so as to avoid unauthorized access to system resources by user programs, thus ensuring system security and stable. User software is not good, what if they mess up and play the system badly? Therefore, certain privileged operations must be handed over to a secure and reliable kernel for execution.

When a task (process) executes a system call and is executed in kernel code, we say that the process is in kernel run state (or simply kernel mode) and the processor is executed in the privileged level (level 0) kernel code. When the process is executing the user's own code, it is said to be in the user's running state (user mode). That is, the processor is now running in the lowest privileged level (level 3) user code. The processor can execute those privileged CPU instructions when the privilege level is high.

Kernel module/driver

User space can access kernel space through system calls, so what if a user space wants to communicate with another user space? It is natural to think of adding support to the operating system kernel; traditional Linux communication mechanisms, such as Socket, pipeline, etc. are supported by the kernel; but Binder is not part of the Linux kernel, how does it access the kernel space? Linux's Dynamic Loadable Kernel Module (LKM) mechanism solves this problem; a module is a program with independent functions that can be compiled separately, but not independently. It is linked to the kernel as part of the kernel running in kernel space at runtime. In this way, the Android system can run in the kernel space by adding a kernel module, and the communication between the user processes through this module can complete the communication.

In the Android system, the kernel module that runs in the kernel space and is responsible for the communication of each user process through the Binder is called the **Binder driver** ;

> A driver is generally referred to as a device driver (Device Driver), a special program that allows a computer to communicate with a device. Equivalent to the hardware interface, the operating system can only control the work of the hardware device through this interface;

The driver is the interface for operating the hardware. In order to support the Binder communication process, Binder uses a kind of "hardware", so this module is called a driver.

Why use Binder?

The Linux kernel used by Android has a lot of cross-process communication mechanisms, such as pipeline, System V, Socket, etc. Why do you need to create a separate Binder? There are two main points, performance and security. On mobile devices, the widespread use of cross-process communication definitely imposes strict requirements on the communication mechanism itself; Binder is more efficient than the traditional Socket method; in addition, the traditional process communication method does not strictly enforce the identity of the two parties. The verification is only performed on the upper layer protocol; for example, the Socket communication ip address is manually filled in by the client, and can be forged; and the Binder mechanism supports the identity check of both parties from the protocol itself, thereby greatly improving the security. Sex. This is also the basis of the Android permissions model.

Binder communication model

For the two sides of the cross-process communication, we call it the Server process (Server) and the Client process (Client); because of the existence of process isolation, there is no way to communicate between them in a simple way, then how does the Binder mechanism work? ?

Recall the process of our communication in daily life: suppose A and B want to communicate, the medium of communication is to make a call (A is Client, B is Server); A wants to call B, must know the number of B, how is this number? Get it? **Address book** .

This address book is a table; the content is roughly:

```
1 B ->
  1234567
2 6
  C ->
  1233435
  4
```

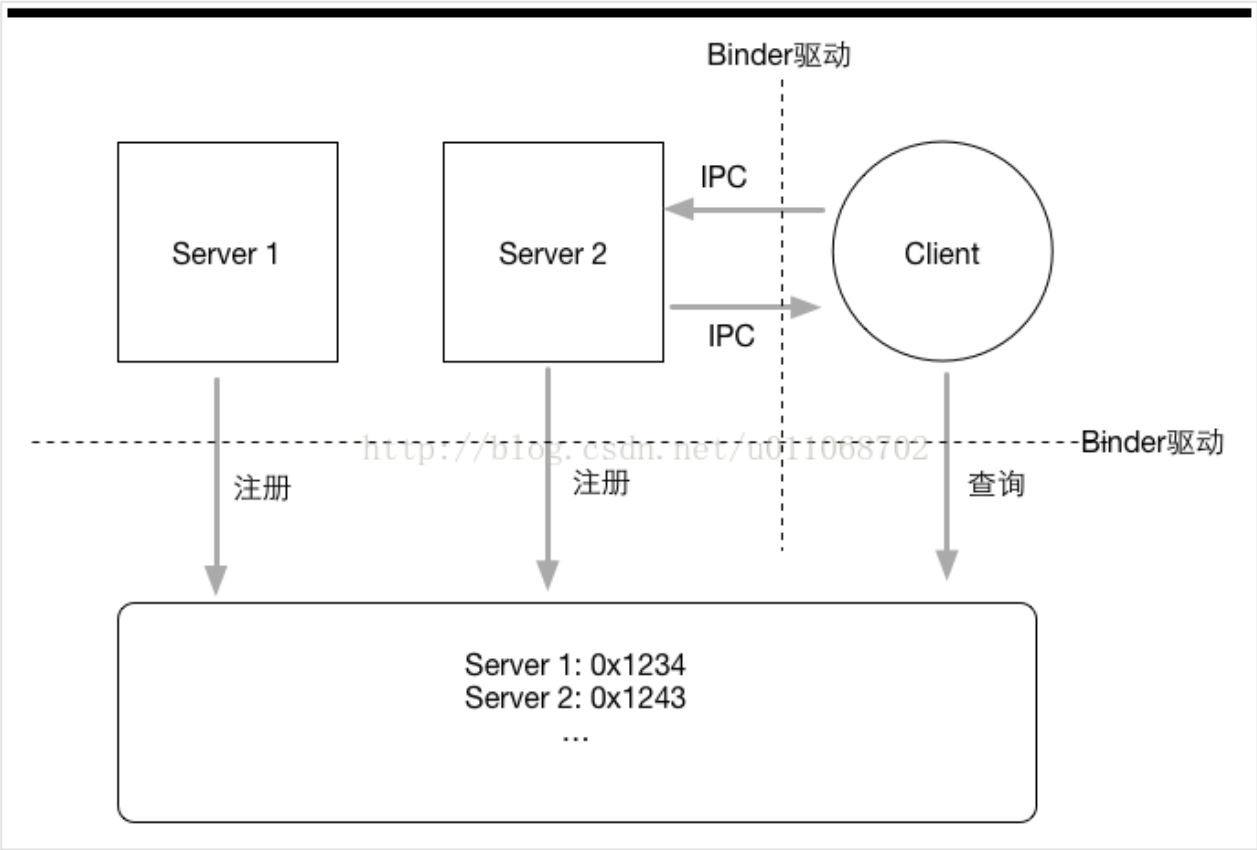
Check the address book first, get the number of B; in order to communicate; otherwise, how do you know what number should be dialed? Recall the old telephone. If A wants to make a call to B, he must first connect to the call center to explain the call to B. When the

call center helps him call B; when the connection is established, the communication is completed.

In addition, it is impossible to complete the communication by telephone and address book, and there is no base station support; the information cannot be communicated at all.

We see that the process of a telephone communication has two hidden roles in addition to the two sides of the communication: the address book and the base station. Binder communication mechanism is the same: two processes running in user space to complete communication, must rely on the help of the kernel, this program running inside the kernel is called **Binder driver** , its function is similar to the base station; the address book is called a **ServiceManager** Something (SM for short)

OK, Binder's communication model is as simple as this:



The entire communication steps are as follows:

1. SM is established (establishing the address book); first, there is a process to apply to the driver as SM; after the driver agrees, the SM process is responsible for managing the Service (note that this is Service instead of Server, because if the communication process is reversed, then the original client The Client will also become the server server. However, the address book is still empty at this time, and there is no one number.
2. Each server registers with SM (improves the address book); after each server-side

process starts, it reports to SM, I am zhangsan, please return me 0x1234 (this address has no practical meaning, analogy); other server processes are in this order; SM has established a table, corresponding to the name and address of each server; it is like B and A met, said to save my number, and then call me to dial 10086;
3. Client wants to communicate with Server, first ask SM; please tell me how to contact zhangsan, SM will give him a number 0x1234 after receiving it; after receiving the client, happy to use this number to dial the server's phone, so he started communication. .

So what did the Binder driver do? Here, the communication between the Client and the SM, as well as the communication between the Client and the Server, will be driven, and the driver is behind the scenes, but it does the most important work. The driver is the core of the entire communication process, so the secrets of completing cross-process communication are all hidden inside the driver; this we will discuss later.

OK, the above is the basic model of the entire Binder communication; doing a simple analogy, of course, there are some inappropriate places (such as the communication record in the real world, everyone has one, but the SM system has only one; there are many base stations, But there is only one driver;) but this is the whole; we see that the entire communication model is very simple.

Binder mechanism cross-process principle

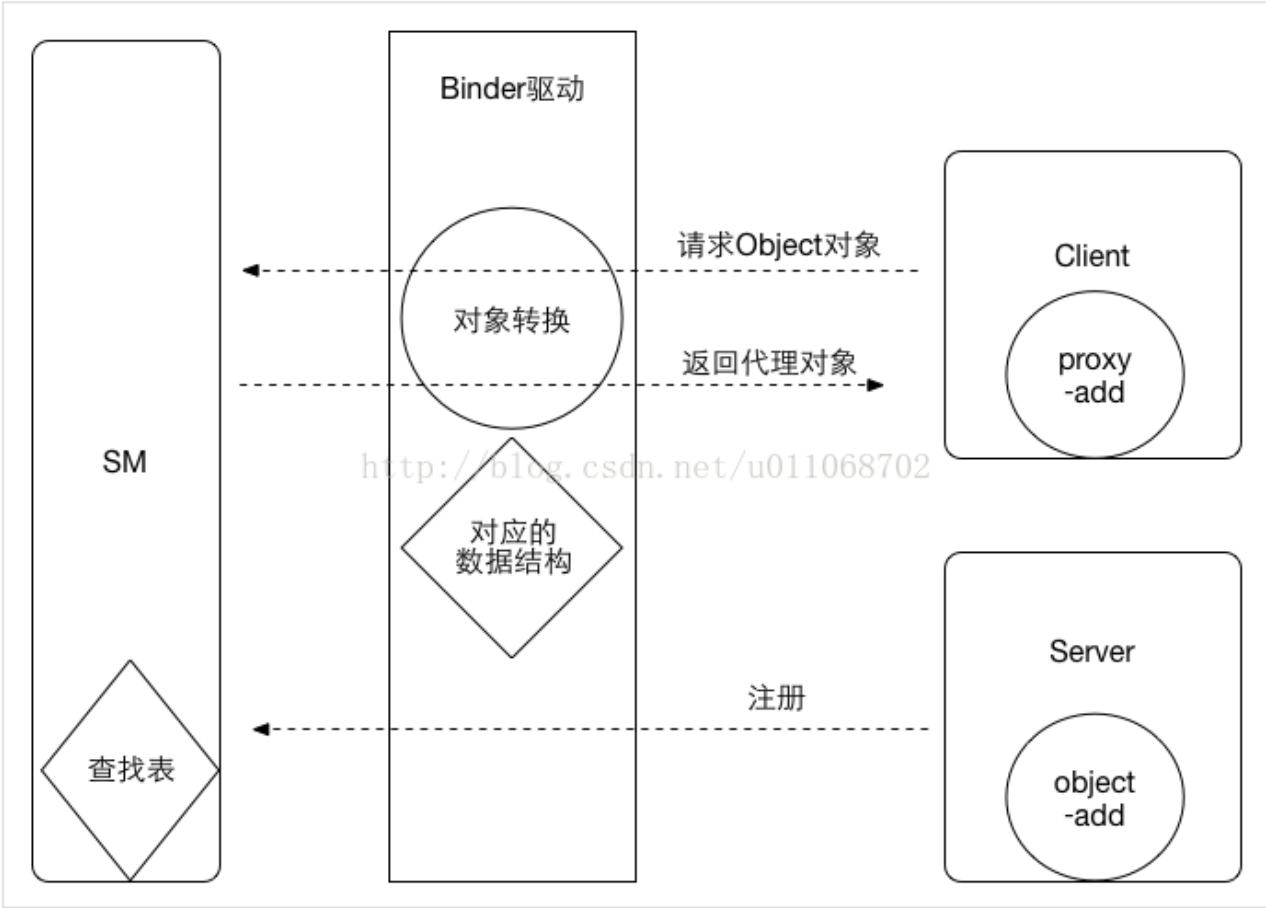
The communication model of Binder is given above, pointing out the four roles of the communication process: Client, Server, SM, driver; But we still don't know how the **Client communicates with the Server** .

How do two processes A and B running in user space complete communication? The kernel can access all the data of A and B; therefore, the easiest way is to do the relay through the kernel; if process A wants to send data to process B, then copy the data of A to the kernel space first, then the kernel space. Data copy to B is complete; user space to operate kernel space, you need to pass the system call; just, there are two system calls: `copy_from_user` , `copy_to_user` .

However, the Binder mechanism is not doing this. Speaking of this paragraph is to show that inter-process communication is not a mysterious thing. So, how does the Binder mechanism implement cross-process communication?

The Binder driver did everything for us.

Suppose the Client process wants to call `object` a method of the server process object `add` ; for this cross-process communication process, let's see how the Binder mechanism does. (Communication is a broad concept. As long as one process can call the method of an object in another process, it is very easy to complete what communication content is done.)



First, the Server process needs to register with the SM; telling who it is and what capabilities it has; in this scenario, Server tells SM that it calls `zhangsan` , it has an `object` object that can perform `add` operations; then SM creates a table: `zhangsan` the name corresponds to Process Server;

Then the Client queries the SM: I need to contact an object `zhangsan` in the process called the name `object` ; this time the key is: the data communicated between the processes will pass through the driver running in the kernel space, the driver does a little while the data flows. hands and feet, it does not give Client process returns a real `object` object, but returns with a look `object` exactly the same proxy object `objectProxy` , this `objectProxy` also has a `add` method, but this `add` method does not Server process inside `object` an object `add` method that ability; `objectProxy` in `add` just a puppet, The only thing it does is wrap the parameters and give them to the driver. (Here we simplify the process of SM, see below)

However, the Client process does not know that the object returned to it by the driver has been manipulated. After all, the camouflage is too similar, such as fake replacement. The Client happily holds the `objectProxy` object and then calls the `add` method; we said that this `add` does nothing, directly wrap the parameters and forward them directly to the Binder driver.

The driver received this message and found it to be this `objectProxy` ; a lookup table would understand: I used to `objectProxy` replace it and `object` send it to the client. It should actually access the `object` object `add` method; then the Binder driver notifies the server process and *calls your object object. The* `add` *method, then send the result to me* , the Sever process receives the message, after the call back the result to the driver, the driver then returns the result to the `Client` process; then the entire process is completed.

Because the driver returns the `objectProxy` same as the original in the Server process `object` , it feels like it is **directly passing the object object in the Server process to the Client process** ; therefore, we can say that the **Binder object is an object that can be passed across processes.**

But in fact, we know that Binder cross-process transfer does not really transfer an object to another process; the transfer process seems to be when Binder crosses the process, it leaves a real body in one process, and morphs in another process. A shadow (this shadow can be many); the operation of the Client process is actually the operation of the shadow, the shadow uses the Binder driver to finally let the real body complete the operation.

It is very important to understand this; be sure to understand it carefully. In addition, the Android system implements this mechanism using the *proxy mode* . If the access to the Binder is in the same process (no cross-process is required), then the original Binder entity is directly returned; if it is in a different process, then give him a Proxy object (shadow); we can see many of these implementations in the system source code and the generated code of AIDL.

In addition, in order to simplify the whole process, we hide the operation of SM part of the driver; in fact, since SM and Server are usually not in one process, the process of registering the server to SM is also inter-process communication, and the driver will also perform a black-box operation on this process. The server-side object that exists in the SM is actually a proxy object. When the client queries the SM, the driver returns another proxy object to the client. There are only one local object in the Sever process, and all other processes have their own agents.

A summary of the sentence is: The **client process is only the agent that holds the server; the proxy object assists the driver to complete the cross-process communication.**

What exactly is Binder?

We often refer to Binder, so what exactly is Binder?

Binder's design uses object-oriented thinking, in the four roles of the Binder communication model; their representatives are "Binder", so for the Binder communication users, the Binder in the Server and the Binder in the Client are not What

is different, a Binder object represents all, it does not care about the details of the implementation, not even care about the driver and the existence of SM; this is abstraction.

- In the usual sense, Binder refers to a communication mechanism; we say that AIDL uses Binder for communication, which refers to the **IPC mechanism of Binder** .
- For the Server process, Binder refers to the **Binder local object.**
- For the Client, Binder refers to the **Binder proxy object** , which is only a remote proxy of the **Binder local object** ; the operation of this Binder proxy object will be finally forwarded to the Binder local object through the driver; for the use of a Binder object In fact, it does not need to care whether this is a Binder proxy object or a Binder local object; there is no difference between the operation of the proxy object and the operation of the local object.
- For the transfer process, Binder is an object that can be passed across processes; the Binder driver performs special processing on objects with cross-process delivery capabilities: automatic conversion of proxy objects and local objects.

> The introduction of object-oriented thinking transforms inter-process communication into a method of calling the object by reference to a Binder object. The unique feature is that the Binder object is an object that can be referenced across processes, and its entity (local object) is located. In a process, its references (proxy objects) are spread across the various processes of the system. The most tempting thing is that this reference can be either a strong type or a weak type, as well as a reference in java, and can be passed from one process to another, so that everyone can access the same server, just like an object or reference. Assigning to another reference is the same. Binder obscures the process boundary and dilutes the interprocess communication process. The whole system seems to run in the same object-oriented program. Binder objects of all kinds and dotted references seem to glue the glue of each application, which is the original meaning of Binder in English.

Drive inside Binder

We now know that the Binder object in the Server process refers to the Binder local object, the object inside the Client is worth the Binder proxy object; when the Binder object is passed across the process, the Binder driver will automatically complete the conversion of these two types; The Binder driver must store the information about each Binder object that spans the process. In the driver, the representative of the Binder local object is called `binder_node` a data structure, and the Binder proxy object is `binder_ref` represented by a representative; some places directly refer to the Binder local object. The Binder entity directly refers to the Binder proxy object as a Binder reference (handle), which actually refers to the representation of the Binder object in the driver; the reader can understand the meaning.

OK, now I understand the communication model of Binder, and I also understand what the Binder object represents in the components of the communication process.

In-depth understanding of the Binder of the Java layer

IBinder/IInterface/Binder/BinderProxy/Stub

When we use the AIDL interface, we often come into contact with these classes, so what does each class represent?

- IBinder is an interface that represents **a cross-process transfer capability** ; as long as the interface is implemented, the object can be passed across processes; this is driven by the underlying support; when the cross-process data flow is driven, the driver It will recognize the data of the IBinder type, thus automatically completing the conversion of the Binder local object and the Binder proxy object of different processes.
- IBinder is responsible for data transfer, then the client and server call contract (no interface here to avoid confusion)? The IInterface here represents what the remote server object has. Specifically, it is the interface inside the aidl.
- The Binder class of the Java layer represents the **Binder local object** . The BinderProxy class is an internal class of the Binder class, which represents the local agent of the Binder object of the remote process; both of these classes inherit from IBinder, so they have the ability to transfer across processes; in fact, when crossing the process, the Binder driver The conversion of these two objects is done automatically.
- When using AIDL, the compilation tool will generate a static inner class for Stub; this class inherits Binder, indicating that it is a Binder local object, which implements the IInterface interface, indicating that it has the ability of the remote server to promise to the Client; Stub is an abstract class. The specific implementation of IInterface needs to be done manually. Here we use the policy mode.

AIDL process analysis

Now let's use an AIDL to analyze what the various roles have done in the whole communication process, and how AIDL completes the communication. (If you are not familiar with AIDL, please check the official documentation first)

First set a simple aidl interface:

```
1
   Package
2  com.example.test.app;
   interface  the ICompute  {
3  int the Add ( int A, int B) ; }

4

5
```

Then compile with the compiler tool, you can get the corresponding ICompute.java class, look at the code generated by the system:

```
1  Package com.example.test.app;

2  Public  interface  ICompute  extends  android . os . IInterface  {
       * Local-side IPC implementation stub class.      */ public static abstract class Stub extends android . os .
3  Binder implements com . example . test . app . ICompute { private static final java .lang.String DESCRIPTOR
   = "com.example.test.app.ICompute" ;
4

5


6


7          . * Construct The AT Stub The interface to the attach IT
           * /
8  public the Stub () { the this .attachInterface ( the this , DESCRIPTOR);         }

9


1
0          * Cast an IBinder object into an com.example.test.app.ICompute interface,
           * generating a proxy if needed.
1          */
1  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
1  ) && (iin instanceof com.example.test.app. ICompute))) { return ((com.example.test.app.ICompute) iin);
2  }

1
3


1
4


1             Return  new com.example.test.app.ICompute.Stub.Proxy(obj);
5          }

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

1
7
           @Override
1  public boolean onTransact ( int code, android.os.Parcel data, android.os.Parcel reply, int flags) throws
8  android.os.RemoteException { switch (code) { case INTERFACE_TRANSACTION: {
   reply.writeString(DESCRIPTOR); return True ;                 } case TRANSACTION_add: {
1  data.enforceInterface(DESCRIPTOR); int _arg0;                    _arg0 = data.readInt(); int _arg1;
9  _arg1 = data.readInt(); int

2
0


2
1


2
2


2
3                  = _result the this .add (_arg0, _arg1);
                   reply.writeNoException ();
2                  reply.writeInt (_result);
4  return to true ;               }            } return Super .onTransact (code, Data, Reply, the flags);         }

2
5


2
6          Private  static  class  Proxy  implements  com . example . test . app . ICompute  {
   private android.os.IBinder mRemote;
```

```
2
7           Proxy(android.os.IBinder remote) {
               mRemote = remote;
2           }
8
            @Override
2  public android.os. IBinder asBinder () { return mRemote;            }
9


3
0           Public java.lang. String getInterfaceDescriptor ()  {
   return DESCRIPTOR;            }
3
1


3           * Demonstrates some basic types that you can use as parameters
2           * and return values in AIDL.
            */
3  @Override public int add ( int a, int b) throws android.os.RemoteException {               android.os.Parcel
3  _data = android .os.Parcel.obtain();              android.os.Parcel _reply = android.os.Parcel.obtain(); int
   _result; try {                  _data.writeInterfaceToken(DESCRIPTOR);                  _data.writeInt(a);
3  _data.writeInt(b );
4


3
5


3
6


3                  mRemote.transact(Stub.TRANSACTION_add, _data, _reply, 0 );
7                  _reply.readException();
                   _result = _reply.readInt();
3               } finally {
8                  _reply.recycle();
                   _data.recycle();
3               }
9  return _result;            }        }

4
0
          Static  final  int TRANSACTION_add = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0 );
4      }
1


4      * Demonstrates that some types you CAN use Basic Parameters AS
2      * and return values in the AIDL.
       * /
4  Public int the Add ( int A, int B) throws android.os.RemoteException ; }
3


4
4


4
5


4
6


4
7


4
8


4
9


5
0


5
1


5
```

2

5
3

5
4

5
5

5
6

5
7

5
8

5
9

6
0

6
1
i
s

6
2
i
s

6
3
i
s

6
4

6
5

6
6

6
7

6
8

6
9

7
0

7
1
i
s

7
2

7
3
i
s

7

4

7
5

7
6

7
7

7
8

7
9

8
0

8
1

8
2

8
3

8
4

8
5

8
6

8
7

8
8

8
9

9
0

9
1
i
s

9
2

9
3

9
4

9
5

9
6

9
7

9
8

```
9
9

1
0
0

1
0
1

1
0
2

1
0
3

1
0
4

1
0
5

1
0
6

1
0
7
```

After the system has generated this file for us, we only need to inherit the abstract class ICompute.Stub, implement its methods, and then implement AIDL in the Service's onBind method. This stub class is very important, so let's see what it does.

The Stub class inherits from Binder, meaning that the Stub is actually a Binder local object, and then implements the ICompute interface. ICompute itself is an IInterface, so it carries some kind of client's required capabilities (here is the method `add`). This class has an inner class proxy, which is the Binder proxy object.

Then look at the `asInterface` method, we bind a Service, in the callback of onServiceConnecttion, is to get a remote service through this method, what does this method do?

```
1
   * Cast an IBinder object into an com.example.test.app.ICompute interface,
2  * generating a proxy if needed.
   */
3  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
4  (((iin != null ) && (iin instanceof com.example.test.app. ICompute))) { return
   ((com.example.test.app.ICompute) iin);     } return new
5
6
7
8      com.example.test.app.ICompute.Stub.Proxy(obj);
   }
9
1
0
1
1
1
2
1
3
1
4
```

First look at the function parameter `IBinder` type obj, this object is driven to us, if it is a Binder local object, then it is the Binder type, if it is a Binder proxy object, that is the `BinderProxy` type; then, as the above automatically generated documentation, It will try to find the Binder local object. If it is found, the client and the server are all in the same process. This parameter is directly a local object, directly forcing the type conversion and then returning. If not found, it is a remote object (in another process) Then you need to create a Binde proxy object, let the Binder proxy implement access to the remote object. In general, if it is communicating with a remote Service object, then the returned here must be a Binder proxy object, the IBinder parameter is actually BinderProxy;

Let's take a look at our `add` implementation of the aidl method; in the Stub class, it `add` is an abstract method, we need to inherit this class and implement it; if the Client and Server are in the same process, then this method is called directly; then, if Remote

call, what happened in the middle? How does the client call the method to the server?

We know, for remote method invocation, is done by Binder agent, in this case there is a `Proxy` class; `Proxy` for `add` methods to achieve the following:

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

5

6

7

8

9          _data.recycle();
       }
1  return _result; }
0

1
1

1
2

1
3

1
4

1
5

1
6

1
7

1
8
```

It first `Parcel` serializes the data and then calls the `transact` method; `transact` what exactly does this do? This `Proxy` class `asInterface` is created in the method, as mentioned earlier, if it is a Binder agent, then the IBinder returned by the driver is actually `BinderProxy` , so `Proxy` the `mRemote` actual type in our class should be `BinderProxy` ; we look at `BinderProxy` the `transact` method: (Binder.java inner class)

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

This is a local method; its implementation is in the native layer, specifically in the frameworks/base/core/jni/android_util_Binder.cpp file, which performs a series of function calls. The call chain is too long and is not given here. What you need to know is that it finally calls the `talkWithDriver` function; see the name of the function, you know that the communication process is handed over to the driver; this function finally passes the `ioctl` system call, the client process falls into the kernel state, and `add` the thread that the client calls the method hangs and waits for return. After the driver completes a series of operations, the server process is awakened, and the `onTransact` function of the local object of the server process is called (actually completed by the server-side thread pool). Let's look at the `onTransact` method of the Binder local object (this is the method `Stub` inside the class):

```
1  @Override
   public  boolean  onTransact ( int code, android.os.Parcel data, android.os.Parcel reply, int flags)  throws
2  android.os.RemoteException {
   switch (code) { case INTERFACE_TRANSACTION: {            reply.writeString(DESCRIPTOR); return True ;
3  } case TRANSACTION_add: {            data.enforceInterface(DESCRIPTOR); int _arg0;            _arg0 =
   data.readInt(); int _arg1;            _arg1 = data.readInt(); int _result = this
4
5
6
7
8
9            .add (_arg0, _arg1);
             reply.writeNoException ();
1            reply.writeInt (_result);
0  return to true ;         }     } return Super .onTransact (code, Data, Reply, the flags); }
1
1
1
2
1
3
1
4
1
5
1
6
1
7
1
8
1
9
2
0
2
1
```

In the Server process, `onTransact` according to the call number (each AIDL function has a number, in the process of cross-process, will not pass the function, but pass the number to indicate which function is called) call the relevant function; in this example, call Binder The `add` method of the local object ; this method returns the result to the driver, which wakes up the thread inside the suspended Client process and returns the result. So a cross-process call is completed.

At this point, you should have a certain understanding of the various classes and roles in the AIDL communication method; it is always a fixed pattern: an object that needs to be passed across processes must inherit from IBinder, if it is a Binder local object , then

must inherit Binder to implement IInterface, if it is a proxy object, then IInterface is implemented and IBinder reference is held;

Proxy is not the same as Stub, although they are both Binder and IInterface, the difference is that Stub uses inheritance (is relationship), Proxy uses combination (has relationship). They all implement all the IInterface functions, except that the Stub uses the strategy mode to call the virtual function (to be implemented by the subclass), while the Proxy uses the combined mode. Why does Stub use inheritance and Proxy uses combination? In fact, Stub itself is an IBinder (Binder), which itself is an object that can be transmitted across process boundaries, so it has to inherit IBinder to implement the transact function to get the ability to cross processes (this ability is given by the driver). The Proxy class uses a combination because it doesn't care what it is. It doesn't need to be transported across processes. It only needs to have this capability. To have this capability, you only need to keep a reference to IBinder. If you make this process an analogy, in the feudal society, Stub is like the emperor, you can order the world, he is born with this right (don't say preaching feudal superstitions.) If one wants to order the world, yes, "挟天子以令The princes." Why don't you go to the emperor yourself? First, it is not necessary in general. When the emperor actually has a lot of restrictions, isn't it? I am now in charge of the world, and can be unconstrained (Java single inheritance); Second, the name is not right, I am not special (Binder), you have to say I can not say, can not do it rebel. Finally, if you want to be an emperor, that is asBinder. In the Stub class, asBinder returns this, and in the Proxy it returns a reference to the combined class IBinder.

Then go through the system's ActivityManagerServer source code, you know which class is what role: IACtivityManager is an IInterface, which represents the capabilities of the remote Service, ActivityManagerNative refers to the Binder local object (similar to the Stub class generated by the AIDL tool), this The class is an abstract class, its implementation is `ActivityManagerService` ; therefore, the final operation of AMS will enter `ActivityManagerService` this real implementation; and if you look closely, ActivityManagerNative.java has a non-public class ActivityManagerProxy, which represents the Binder proxy object; is not with AIDL The model is exactly the same? So what `ActivityManager` is it? He is just a management class, and you can see that the real operation is forwarded to `ActivityManagerNative` the implementation that is handed to him `ActivityManagerService` .

OK, this article is here, to understand Binder in depth, you need to work hard; those native layers and the calling process inside the driver, writing it out of the article is meaningless, you need to track it yourself; then you can:

1. Look at Android documentation, `Parcel, IBinder, Binder` classes that involve cross-process communication;
2. Do not rely on AIDL tools, handwritten remote service to complete cross-process communication
3. See "Binder Design and Implementation"
4. Look at Lao Luo's blog or book (the book structure is clearer)
5. Look at "Binder Design and Implementation"

6. Learn Linux system related knowledge; see the source code yourself.