Android IPC mechanism (3): Talking about the use of Binder

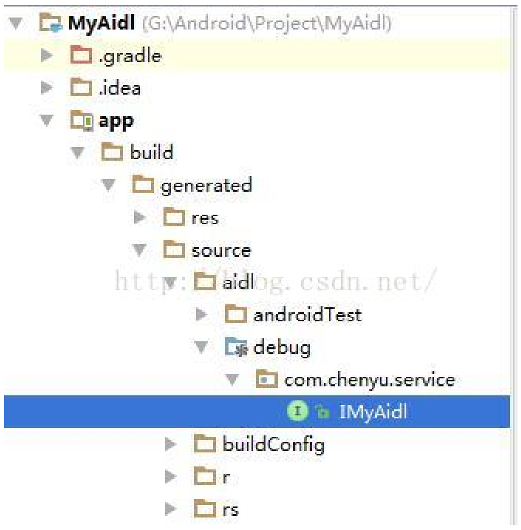
blog.csdn.net/a553181867/article/details/51125124

I. Introduction

In the previous blog Android IPC mechanism (2): The basic use of AIDL , the author describes a major way of communication between Android processes, using AIDL for communication, and introduces the basic use of AIDL. In fact, AIDL uses Binder for cross-process communication. Binder is a cross-process communication method in Android. The underlying implementation principle is more complicated. It is limited to the author level and cannot be discussed in detail. Therefore, this article mainly talks about AIDL. For example, let's talk about how to use Binder.

Second, the principle

In the previous article, I created an IMyAidl.aidl file, the interface file, which was compiled and generated a .java file, which is in the gen directory:



Open the file and get the following code:

```
<span style="font-size:18px;">
package com.chenyu.service;
public interface IMyAidl extends android.os.IInterface {
    public static abstract class Stub extends android.os.Binder implements com.chenyu.service.IMyAidl {
        ......
    public void addPerson(com.chenyu.service.Person person) throws android.os.RemoteException;
    public java.util.List<com.chenyu.service.Person> getPersonList() throws android.os.RemoteException;</span>
}
```

Some of them are omitted, and we first understand them in general and then go deeper.
(1) In general, the java file is an interface, inherits the IInterface interface, and then declares a static internal abstract class: Stub, then two methods, you can see that these two methods are the original IMyAidl Two methods declared in the .aidl file.

(2) We look back at the Stub class, which inherits Binder and implements IMyAidl . This class implements its own interface! So it is conceivable that the addPerson and getPersonList methods declared by the interface will be implemented in the Stub class. How to implement it, we expand the Stub class:

```
public static abstract class Stub extends android.os.Binder implements com.chenyu.service.IMyAidl {
        private static final java.lang.String DESCRIPTOR = "com.chenyu.service.IMyAidl";
        public Stub() {<span style="white-space:pre">  </span>
            this.attachInterface(this, DESCRIPTOR);
        }
        public static com.chenyu.service.IMyAidl asInterface(android.os.IBinder obj) {
            if ((obj == null)) {
                return null;
            }
            android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
            if (((iin != null) && (iin instanceof com.chenyu.service.IMyAidl))) {
                return ((com.chenyu.service.IMyAidl) iin);
            }
            return new com.chenyu.service.IMyAidl.Stub.Proxy(obj);
        }
        @Override
        public android.os.IBinder asBinder() {<span style="white-space:pre">  </span>
            return this;
        }
        @Override
        public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags) throws android.os.RemoteException {<span style="white-space:pre">    </span>
            switch (code) {
```

```
            case INTERFACE_TRANSACTION: {
                reply.writeString(DESCRIPTOR);
                return true;
            }
            case TRANSACTION_addPerson: {
                data.enforceInterface(DESCRIPTOR);
                com.chenyu.service.Person _arg0;
                if ((0 != data.readInt())) {
                    _arg0 = com.chenyu.service.Person.CREATOR.createFromParcel(data);
                } else {
                    _arg0 = null;
                }
                this.addPerson(_arg0);
                reply.writeNoException();
                return true;
            }
            case TRANSACTION_getPersonList: {
                data.enforceInterface(DESCRIPTOR);
                java.util.List<com.chenyu.service.Person> _result = this.getPersonList();
                reply.writeNoException();
                reply.writeTypedList(_result);
                return true;
            }
        }
        return super.onTransact(code, data, reply, flags);
    }
    private static class Proxy implements com.chenyu.service.IMyAidl {<span style="white-space:pre"> </span>
        ...
    }
    static final int TRANSACTION_addPerson = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
    static final int TRANSACTION_getPersonList = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);
}
```

(3) From top to bottom, we analyze the effects of each method or variable one by one:
1Stub() constructor: This method calls the attachInterface() method of the parent class Binder to associate the current Interface with the Binder. Since the parameter DESCRIPTOR is passed, the current Interface is uniquely identified.

2asInterface(IBinder obj) : Static method, passing an interface object , where is the object passed in? Let's take a look at the client code of the previous chapter blog:

```
public void onServiceConnected(ComponentName name, IBinder service) {
            Log.d("cylog", "onServiceConnected success");
            iMyAidl=IMyAidl.Stub.asInterface(service);
        }
```

Here, you can see that the IMyAidl.Stub.asInterface(service) method is called, which is the above method No. 2, and the service is passed in. Let's look down:

```
public static com.chenyu.service.IMyAidl asInterface(android.os.IBinder obj) {
            if ((obj == null)) {
                return null;
            }
            android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
            if (((iin != null) && (iin instanceof com.chenyu.service.IMyAidl))) {
                return ((com.chenyu.service.IMyAidl) iin);
            }
            return new com.chenyu.service.IMyAidl.Stub.Proxy(obj);
        }
```

First, it is judged whether obj is valid. If it is invalid, it directly returns Null, indicating that the connection between the client and the server has failed. Then call the obj.queryLocalInterface(DESCRIPTOR) method to assign values to the IInterface object.

Note that the DESCRIPTOR parameter is passed again. It can be guessed that this method should look for a method related to the current Interface. Let's look at the queryLocalInterface of the IBinder interface.)method:

```
/**
     * Attempt to retrieve a local implementation of an interface
     * for this Binder object.  If null is returned, you will need
     * to instantiate a proxy class to marshall calls through
     * the transact() method.
     */
    public IInterface queryLocalInterface(String descriptor);
```

Probably means that, according to the value of the descriptor, try to retrieve a local interface for the Binder, where local means the current process , if the return value is null, then you should instantiate a proxy class. After understanding the obj.queryLocalInterface(DESCRIPTOR) method, we return to the asInterface(obj) method again , and continue to look down: then an if judgment, mainly to determine whether the client and the server are in the same process, if they are in the same process, Then directly return the Stub object itself, if not the same process, then a new Proxy proxy class (mentioned below) will be created .

3asBinder(): This method is used to return the current object itself. 4onTransact(int code, Parcel data, Parcel reply, int flags): This method generally runs in the Binder thread pool in the server, that is, the remote request will be processed in this method. The passed code value is used to determine the client's request target, either addPerson or getPersonList. Let's take the request target as addPerson() as an example, and extract the main function body as follows:

```
case TRANSACTION_addPerson: {
                    data.enforceInterface(DESCRIPTOR);
                    com.chenyu.service.Person _arg0;
                    if ((0 != data.readInt())) {
                        _arg0 = com.chenyu.service.Person.CREATOR.createFromParcel(data);
                    } else {
                        _arg0 = null;
                    }
                    this.addPerson(_arg0);
                    reply.writeNoException();
                    return true;
                }
```

First, the object is declared _arg0 Person class, and then to data as a parameter called the Person class CREATOR.createFromParcel method, deserialization generate the Person class, which is why classes that implement the interface should also realize Parcelable CREATOR , original The deserialization method is called here. Next, call this.addPerson(_arg0) method, note: this represents the current Binder object, then the addPerson(_arg0) method called by Binder is actually called by the service bound to Binder, that is, the server calls Its own addPerson method. For the sake of convenience, let's review the code of the server on the previous article:

```
private IBinder iBinder= new IMyAidl.Stub() {
        @Override
        public void addPerson(Person person) throws RemoteException {
            persons.add(person);
        }
        @Override
        public List<Person> getPersonList() throws RemoteException {
            return persons;
        }
    };
```

Did you understand it all at once? The interface implemented by IMyAidl.Stub(), where the methods are implemented on the server side: addPerson and getPersonList(). When in the Binder thread pool, the this.addPerson() method is called, and the addPerson method of the server is actually called back. How is the underlying layer implemented? It is limited to the author's level. I don't understand it for a while, so I will learn more about Binder later. The working mechanism answers this question.

Ok, back to the current class, let's move on:
5private static class Proxy: There is a private static inner class, which will be covered in detail later.

The last two lines of code are two constants, which mark the two methods, the code values mentioned above.

(4) The Proxy class also implements the IMyAidl interface, and implements the methods of addPerson and getPersonList. Where is the Proxy class instantiated? In the above (3) 2, when the client and the server are not in the same process, the proxy class is instantiated and returned to the client. What is a proxy class? The so-called agent, that is, an intermediary, the instance obtained by the client, can operate some functions of the server, so that the client thinks that he has obtained the instance of the server, in fact, not, just get a proxy on the server. Next we expand the class and look inside:

```
private static class Proxy implements com.chenyu.service.IMyAidl {
            private android.os.IBinder mRemote;
            Proxy(android.os.IBinder remote) {
                mRemote = remote;
            }
            @Override
            public android.os.IBinder asBinder() {
                return mRemote;
            }
            public java.lang.String getInterfaceDescriptor() {
                return DESCRIPTOR;
            }
            @Override
            public void addPerson(com.chenyu.service.Person person) throws
android.os.RemoteException {
```

```
            android.os.Parcel _data = android.os.Parcel.obtain();
            android.os.Parcel _reply = android.os.Parcel.obtain();
            try {
                _data.writeInterfaceToken(DESCRIPTOR);
                if ((person != null)) {
                    _data.writeInt(1);
                    person.writeToParcel(_data, 0);
                } else {
                    _data.writeInt(0);
                }
                mRemote.transact(Stub.TRANSACTION_addPerson, _data, _reply, 0);
                _reply.readException();
            } finally {
                _reply.recycle();
                _data.recycle();
            }
        }
        @Override
        public java.util.List<com.chenyu.service.Person> getPersonList() throws android.os.RemoteException {
            android.os.Parcel _data = android.os.Parcel.obtain();
            android.os.Parcel _reply = android.os.Parcel.obtain();
            java.util.List<com.chenyu.service.Person> _result;
            try {
                _data.writeInterfaceToken(DESCRIPTOR);
                mRemote.transact(Stub.TRANSACTION_getPersonList, _data, _reply, 0);<span style="white-space:pre"> </span>
                _reply.readException();
                _result = _reply.createTypedArrayList(com.chenyu.service.Person.CREATOR);
            } finally {
                _reply.recycle();
                _data.recycle();
            }
            return _result;
        }
    }
```

Here focus on the implementation of two interface methods: addPerson and getPersonList, these two methods have appeared multiple times, and the two methods implemented in the proxy class are running on the client! ! ! The main implementation process is this: when the client gets the proxy class, call the addPerson or getPersonList method, first create the input parcel object _data and the output parcel object _reply, then call the first code to call transact to initiate the RPC remote Request, at the same time the current thread will be suspended , at this time, the server's onTransact will be called, that is, the above mentioned (3) No. 4 code, when the server finishes processing the request, it will return data, the current thread continues to execute and knows to return _result result.

At this point, one of the ways of IPC - the principle of AIDL has been analyzed, and then **summarize** :

1. The client issues a binding request, and the server and the client are bound to the same Binder. The client executes the asInterface() method. If the client and the server are in the same process, the Stub object itself is returned directly. If it is in a different process, the Stub.proxy proxy class object is returned.

2. The client sends a remote request (addPerson or getPersonList). At this time, the client thread hangs. After the Binder gets the data, the data is processed. If it is in a different process, the data is written to Parcel and the Transact method is called.

3. The onTransact method is triggered. The method runs in the Binder thread pool. The method calls the interface method implemented by the server. When the data is processed, the reply value is returned. After the Binder returns to the client, the client thread is woken up.

Third, optimization

Finally, let me talk about how to optimize AIDL. As mentioned above, after the client sends the request, it will be suspended. This means that if the data processing time is too long, the thread will not wait for wakeup, which is very serious. If the request is sent in the UI thread, it will directly lead to ANR, so we need to send an asynchronous request in the child thread , in order to avoid ANR. Another point is that Binder may have died unexpectedly. If Binder accidentally dies, the child thread may hang all the time, so we need to enable the reconnection service. There are two methods, one is to set the DeathRecipient listener for Binder , and the other is to reconnect the service in onServiceDisconnected .

Reference book: "Android Development Art Exploration" Ren Yugang, first edition in September 2015