


# Android's best understanding of the Binder mechanism

 [blog.csdn.net/u011068702/article/details/53610683](http://blog.csdn.net/u011068702/article/details/53610683)

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
```

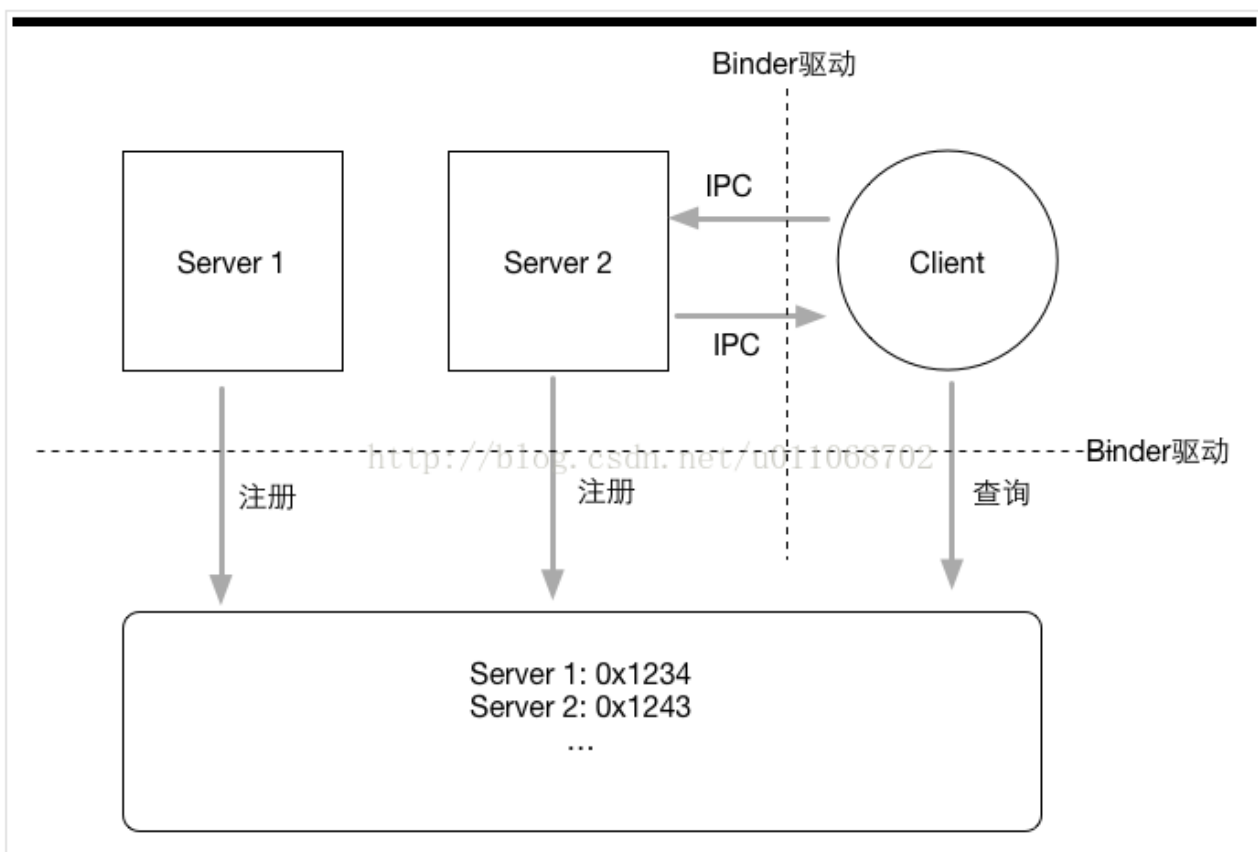
Check the address book first, get the number of B; in order to communicate; otherwise, how do you know what number should be dialed? Recall the old telephone. If A wants to make a call to B, he must first connect to the call center to explain the call to B. When the

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** .

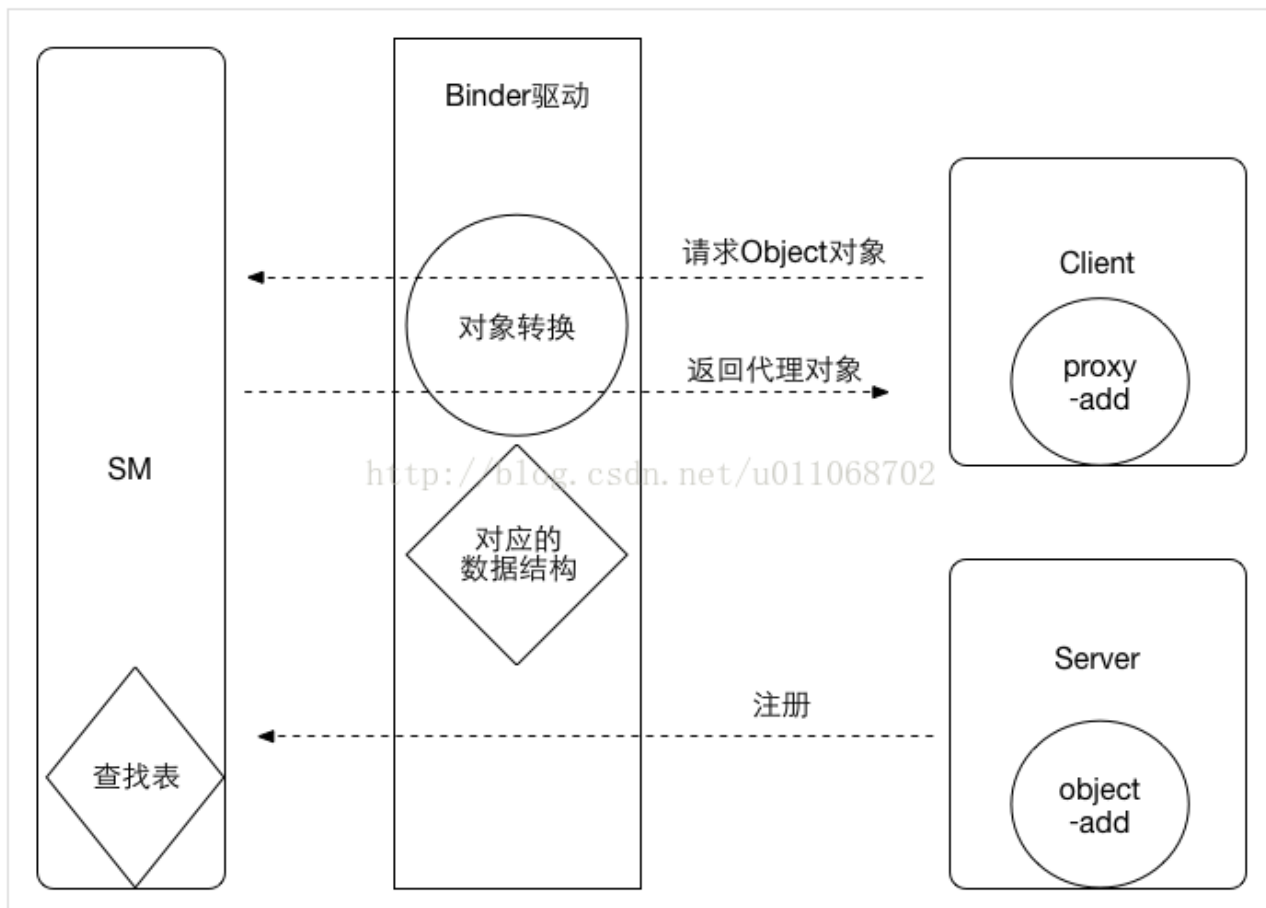
How do two processes A and B running in user space complete communication? The kernel can access all the data of A and B; therefore, the easiest way is to do the relay through the kernel; if process A wants to send data to process B, then copy the data of A to the kernel space first, then the kernel 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 Server 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 Server 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;
3 interface the ICompute {
4     int the Add ( int A, int B ) ; }
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
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658

```

```

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 `Interface`, 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 `onServiceConnection`, 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  */
4  public static com.example.test.app. ICompute asInterface (android.os.IBinder obj) {
5      if ((obj == null )) { return null ;    }    android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR); if
6      (((iin != null ) && (iin instanceof com.example.test.app. ICompute))) { return
7      ((com.example.test.app.ICompute) iin);    } return new
8      com.example.test.app.ICompute.Stub.Proxy(obj);
9  }
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

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
2 public int add ( int a, int b) throws android.os.RemoteException {
3     android.os.Parcel _data = android.os.Parcel.obtain();
4     android.os.Parcel _reply = android.os.Parcel.obtain( );
5     int _result; the try { _data.writeIntInterfaceToken (DESCRIPTOR); _data.writeInt (A);
6     _data.writeInt (B); mRemote.transact (Stub.TRANSACTION_add, the _data, _reply, 0 );
7     _reply.readException (); _result = _reply.readInt(); } finally { _reply.recycle();
8
9     _data.recycle();
10 }
11 return _result; }
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

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
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, "挟天子以令诸侯." 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.