Android系统进程间通信机制Binder的总体架构由Client、Server、ServiceManager和驱动程序Binder四个组件构成。今天主要来看看ServiceManager的实现吧。

# TODO

ser

# Service Manager

## 基本概念

**Service Manager**：是系统中一个独立的进程，它是整个Binder机制的守护进程，用来管理开发者创建的各种Server，并且向Client提供查询Server远程接口的功能。

## 调用关系

init启动**Service Manager**：

在init进程启动的过程中会去解析init.rc文件

service servicemanager /system/bin/servicemanager

class core

user system

group system

critical

onrestart restart healthd

onrestart restart zygote

onrestart restart media

onrestart restart surfaceflinger

onrestartrestart drm

可以看出， servicemanager服务还是很重要的，它带了critical标志，即如果该进程连续crash几次，系统会进入恢复模式。而且如果该进程挂了，系统还会重启zygote、surfaceflinger等系统关键进程。其重要性真是不言而喻

## 源码分析

既然Service Manager组件是用来管理Server并且向Client提供查询Server远程接口的功能，那么，Service Manager就必然要和Server以及Client进行通信了。我们知道，Service Manger、Client和Server三者分别是运行在独立的进程当中，这样它们之间的通信也属于进程间通信了，而且也是采用Binder机制进行进程间通信，因此，Service Manager在充当Binder机制的守护进程的角色的同时，也在充当Server的角色，然而，它是一种特殊的Server，下面我们将会看到它的特殊之处

源代码位于frameworks/base/cmds/servicemanager目录下，主要是由binder.h、binder.c和service\_manager.c三个文件组成。Service Manager的入口位于service\_manager.c文件中的main函数：main函数如下：

### service\_manager.c

main函数主要有三个功能：一是打开Binder设备文件；二是告诉Binder驱动程序自己是Binder上下文管理者，即我们前面所说的守护进程；三是进入一个无穷循环，充当Server的角色，等待Client的请求

1. **int** main(**int** argc, **char** \*\*argv)
2. {
3. /\*
4. struct binder\_state
5. {
6. int fd; // /dev/binder的文件描述符
7. void \*mapped;   // mmap把设备文件/dev/binder映射到进程空间的起始地址
8. unsigned mapsize;   // mmap映射的大小
9. };
10. /\*
11. struct binder\_state \*bs;
13. /\*
14. /\* the one magic object \*/
15. #define BINDER\_SERVICE\_MANAGER ((void\*) 0)
16. 一个magic对象，用0来标识一个Service Manager
17. \*/
18. **void** \*svcmgr = BINDER\_SERVICE\_MANAGER;
20. // 打开/dev/binder设备，并mmap映射之，大小为128K
21. bs = binder\_open(128\*1024);
23. // 告诉binder驱动我是Service Manager进程
24. **if** (binder\_become\_context\_manager(bs)) {
25. ALOGE("cannot become context manager (%s)\n", strerror(errno));
26. **return** -1;
27. }
29. svcmgr\_handle = svcmgr;
31. // 循环接受并处理消息
32. binder\_loop(bs, svcmgr\_handler);
33. **return** 0;
34. }

宏BINDER\_SERVICE\_MANAGER定义frameworks/base/cmds/servicemanager/binder.h文件中：

1. /\* the one magic object \*/
2. #define BINDER\_SERVICE\_MANAGER ((void\*) 0)

它表示Service Manager的句柄为0。Binder通信机制使用句柄来代表远程接口，这个句柄的意义和Windows编程中用到的句柄是差不多的概念。前面说到，Service Manager在充当守护进程的同时，它充当Server的角色，当它作为远程接口使用时，它的句柄值便为0，这就是它的特殊之处，其余的Server的远程接口句柄值都是一个大于0 而且由Binder驱动程序自动进行分配的

### 打开binder驱动

首先来看看binder\_open的实现，代码很简单，就不详细说明了：

**[cpp]** [view plain](http://blog.csdn.net/hu3167343/article/details/38441119) [copy](http://blog.csdn.net/hu3167343/article/details/38441119)

1. **struct** binder\_state \*binder\_open(unsigned mapsize)
2. {
3. **struct** binder\_state \*bs;
5. bs = malloc(**sizeof**(\*bs));
6. **if** (!bs) {
7. errno = ENOMEM;
8. **return** 0;
9. }
11. bs->fd = open("/dev/binder", O\_RDWR);
12. **if** (bs->fd < 0) {
13. fprintf(stderr,"binder: cannot open device (%s)\n",
14. strerror(errno));
15. **goto** fail\_open;
16. }
18. bs->mapsize = mapsize;
19. bs->mapped = mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0);
20. **if** (bs->mapped == MAP\_FAILED) {
21. fprintf(stderr,"binder: cannot map device (%s)\n",
22. strerror(errno));
23. **goto** fail\_map;
24. }
26. **return** bs;
28. fail\_map:
29. close(bs->fd);
30. fail\_open:
31. free(bs);
32. **return** 0;
33. }

### 注册为服务进程

接着来看binder\_become\_context\_manager的实现：

**[cpp]** [view plain](http://blog.csdn.net/hu3167343/article/details/38441119) [copy](http://blog.csdn.net/hu3167343/article/details/38441119)

1. **int** binder\_become\_context\_manager(**struct** binder\_state \*bs)
2. {
3. **return** ioctl(bs->fd, BINDER\_SET\_CONTEXT\_MGR, 0);
4. }

binder\_become\_context\_manager原来是通过ioctl的系统调用来告诉binder驱动自己是MGR进程

### 消息循环

最后，调用binder\_loop开始接受消息：

1. **void** binder\_loop(**struct** binder\_state \*bs, binder\_handler func)
2. {
3. **int** res;
4. /\*
5. binder\_write\_read结构用来应用层和驱动层传递数据。
6. struct binder\_write\_read {
7. signed long write\_size;
8. signed long write\_consumed;
9. unsigned long write\_buffer;
10. signed long read\_size;
11. signed long read\_consumed;
12. unsigned long read\_buffer;
13. };
14. \*/
15. **struct** binder\_write\_read bwr;
16. unsigned readbuf[32];
18. bwr.write\_size = 0;
19. bwr.write\_consumed = 0;
20. bwr.write\_buffer = 0;
22. // 通知binder驱动，我要开始循环接受消息了。
23. readbuf[0] = BC\_ENTER\_LOOPER;
24. binder\_write(bs, readbuf, **sizeof**(unsigned));
26. // 开始循环接收数据
27. **for** (;;) {
28. bwr.read\_size = **sizeof**(readbuf);
29. bwr.read\_consumed = 0;
30. bwr.read\_buffer = (unsigned) readbuf;
32. // 通过ioctl获得驱动中的数据
33. res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);
35. **if** (res < 0) {
36. ALOGE("binder\_loop: ioctl failed (%s)\n", strerror(errno));
37. **break**;
38. }
40. // 获得之后开始解析处理
41. res = binder\_parse(bs, 0, readbuf, bwr.read\_consumed, func);
42. **if** (res == 0) {
43. ALOGE("binder\_loop: unexpected reply?!\n");
44. **break**;
45. }
46. **if** (res < 0) {
47. ALOGE("binder\_loop: io error %d %s\n", res, strerror(errno));
48. **break**;
49. }
50. }
51. }

### binder\_parse

的v

1. **int** binder\_parse(**struct** binder\_state \*bs, **struct** binder\_io \*bio,
2. uint32\_t \*ptr, uint32\_t size, binder\_handler func)
3. {
4. **int** r = 1;
5. uint32\_t \*end = ptr + (size / 4);
7. **while** (ptr < end) {
8. uint32\_t cmd = \*ptr++;
9. **switch**(cmd) {
10. **case** BR\_NOOP:
11. **break**;
12. **case** BR\_TRANSACTION\_COMPLETE:
13. **break**;
14. **case** BR\_INCREFS:
15. **case** BR\_ACQUIRE:
16. **case** BR\_RELEASE:
17. **case** BR\_DECREFS:
18. #if TRACE
19. fprintf(stderr,"  %08x %08x\n", ptr[0], ptr[1]);
20. #endif
21. ptr += 2;
22. **break**;
23. **case** BR\_TRANSACTION: {
24. **struct** binder\_txn \*txn = (**void** \*) ptr;
25. **if** ((end - ptr) \* **sizeof**(uint32\_t) < **sizeof**(**struct** binder\_txn)) {
26. ALOGE("parse: txn too small!\n");
27. **return** -1;
28. }
29. binder\_dump\_txn(txn);
30. **if** (func) {
31. unsigned rdata[256/4];
32. **struct** binder\_io msg;
33. **struct** binder\_io reply;
34. **int** res;
36. bio\_init(&reply, rdata, **sizeof**(rdata), 4);
37. bio\_init\_from\_txn(&msg, txn);
38. res = func(bs, txn, &msg, &reply);
39. binder\_send\_reply(bs, &reply, txn->data, res);
40. }
41. ptr += **sizeof**(\*txn) / **sizeof**(uint32\_t);
42. **break**;
43. }
44. **case** BR\_REPLY: {
45. **struct** binder\_txn \*txn = (**void**\*) ptr;
46. **if** ((end - ptr) \* **sizeof**(uint32\_t) < **sizeof**(**struct** binder\_txn)) {
47. ALOGE("parse: reply too small!\n");
48. **return** -1;
49. }
50. binder\_dump\_txn(txn);
51. **if** (bio) {
52. bio\_init\_from\_txn(bio, txn);
53. bio = 0;
54. } **else** {
55. /\* todo FREE BUFFER \*/
56. }
57. ptr += (**sizeof**(\*txn) / **sizeof**(uint32\_t));
58. r = 0;
59. **break**;
60. }
61. **case** BR\_DEAD\_BINDER: {
62. **struct** binder\_death \*death = (**void**\*) \*ptr++;
63. death->func(bs, death->ptr);
64. **break**;
65. }
66. **case** BR\_FAILED\_REPLY:
67. r = -1;
68. **break**;
69. **case** BR\_DEAD\_REPLY:
70. r = -1;
71. **break**;
72. **default**:
73. ALOGE("parse: OOPS %d\n", cmd);
74. **return** -1;
75. }
76. }
78. **return** r;
79. }

如果接收到了消息，那么binder\_parse会经过简单的转换func，最后还是会调用回调函数svcmgr\_handler:

### svcmgr\_handler

svcmgr\_handler

1. **int** svcmgr\_handler(**struct** binder\_state \*bs,
2. **struct** binder\_txn \*txn,
3. **struct** binder\_io \*msg,
4. **struct** binder\_io \*reply)
5. {
6. **struct** svcinfo \*si;
7. uint16\_t \*s;
8. unsigned len;
9. **void** \*ptr;
10. uint32\_t strict\_policy;
11. **int** allow\_isolated;
13. // 确认消息是发给server manager
14. **if** (txn->target != svcmgr\_handle)
15. **return** -1;
17. …….
19. **switch**(txn->code) {
20. **case** SVC\_MGR\_GET\_SERVICE:
21. **case** SVC\_MGR\_CHECK\_SERVICE:
22. // 获得当前服务名，s变量为服务名
23. s = bio\_get\_string16(msg, &len);
24. /\*
25. Server manager在其进程空间维护了一个service的全局链表svclist，
26. do\_find\_service通过服务名来查找该服务是否存在，并返回给客户端
27. \*/
28. ptr = do\_find\_service(bs, s, len, txn->sender\_euid);
29. **if** (!ptr)
30. **break**;
31. bio\_put\_ref(reply, ptr);
32. **return** 0;
34. // 其他的server进程来添加服务
35. **case** SVC\_MGR\_ADD\_SERVICE:
36. s = bio\_get\_string16(msg, &len);
37. ptr = bio\_get\_ref(msg);
38. allow\_isolated = bio\_get\_uint32(msg) ? 1 : 0;
39. **if** (do\_add\_service(bs, s, len, ptr, txn->sender\_euid, allow\_isolated))
40. **return** -1;
41. **break**;
43. // 枚举所有已经注册了的服务名
44. **case** SVC\_MGR\_LIST\_SERVICES: {
45. unsigned n = bio\_get\_uint32(msg);
47. si = svclist;
48. **while** ((n-- > 0) && si)
49. si = si->next;
50. **if** (si) {
51. bio\_put\_string16(reply, si->name);
52. **return** 0;
53. }
54. **return** -1;
55. }
56. **default**:
57. ALOGE("unknown code %d\n", txn->code);
58. **return** -1;
59. }
61. bio\_put\_uint32(reply, 0);
62. **return** 0;
63. }

### do\_add\_service

发过

1. **int** do\_add\_service(**struct** binder\_state \*bs,
2. uint16\_t \*s, unsigned len,
3. **void** \*ptr, unsigned uid, **int** allow\_isolated)
4. {
5. **struct** svcinfo \*si;
6. // 参数检查
7. **if** (!ptr || (len == 0) || (len > 127))
8. **return** -1;
10. // 根据uid和服务名判断是否可以被注册
11. **if** (!svc\_can\_register(uid, s)) {
12. ALOGE("add\_service('%s',%p) uid=%d - PERMISSION DENIED\n",
13. str8(s), ptr, uid);
14. **return** -1;
15. }
17. // 根据服务名，遍历svclist服务链表
18. si = find\_svc(s, len);
19. **if** (si) {
20. **if** (si->ptr) {
21. ALOGE("add\_service('%s',%p) uid=%d - ALREADY REGISTERED, OVERRIDE\n",
22. str8(s), ptr, uid);
23. svcinfo\_death(bs, si);  // 清理前一个服务
24. }
25. si->ptr = ptr;   // 赋值新服务
26. } **else** {
27. si = malloc(**sizeof**(\*si) + (len + 1) \* **sizeof**(uint16\_t));
28. **if** (!si) {
29. ALOGE("add\_service('%s',%p) uid=%d - OUT OF MEMORY\n",
30. str8(s), ptr, uid);
31. **return** -1;
32. }
33. si->ptr = ptr;
34. si->len = len;
35. memcpy(si->name, s, (len + 1) \* **sizeof**(uint16\_t));
36. si->name[len] = '\0';
37. si->death.func = svcinfo\_death;  // 如果server进程挂了，那么调用该函数处理
38. si->death.ptr = si;
39. si->allow\_isolated = allow\_isolated;
40. si->next = svclist;
41. svclist = si;
42. }
44. binder\_acquire(bs, ptr);
45. binder\_link\_to\_death(bs, ptr, &si->death);
46. **return** 0;
47. }

do\_add\_service首先会调用svc\_can\_register函数检测当前要添加的服务是否被允许，函数如下：

### svc\_can\_register

ghd

1. **int** svc\_can\_register(unsigned uid, uint16\_t \*name)
2. {
3. unsigned n;
4. // 如果uid为root或者system权限，那么直接放行
5. **if** ((uid == 0) || (uid == AID\_SYSTEM))
6. **return** 1;
8. // 否则，检查allowed数组
9. **for** (n = 0; n < **sizeof**(allowed) / **sizeof**(allowed[0]); n++)
10. **if** ((uid == allowed[n].uid) && str16eq(name, allowed[n].name))
11. **return** 1;
13. **return** 0;
14. }

allowed数组内容如下：

**[cpp]** [view plain](http://blog.csdn.net/hu3167343/article/details/38441119) [copy](http://blog.csdn.net/hu3167343/article/details/38441119)

1. **static** **struct** {
2. unsigned uid;
3. **const** **char** \*name;
4. } allowed[] = {
5. { AID\_MEDIA, "media.audio\_flinger" },
6. { AID\_MEDIA, "media.log" },
7. { AID\_MEDIA, "media.player" },
8. { AID\_MEDIA, "media.camera" },
9. { AID\_MEDIA, "media.audio\_policy" },
10. { AID\_DRM,   "drm.drmManager" },
11. { AID\_NFC,   "nfc" },
12. { AID\_BLUETOOTH, "bluetooth" },
13. { AID\_RADIO, "radio.phone" },
14. { AID\_RADIO, "radio.sms" },
15. { AID\_RADIO, "radio.phonesubinfo" },
16. { AID\_RADIO, "radio.simphonebook" },
17. /\* TODO: remove after phone services are updated: \*/
18. { AID\_RADIO, "phone" },
19. { AID\_RADIO, "sip" },
20. { AID\_RADIO, "isms" },
21. { AID\_RADIO, "iphonesubinfo" },
22. { AID\_RADIO, "simphonebook" },
23. { AID\_MEDIA, "common\_time.clock" },
24. { AID\_MEDIA, "common\_time.config" },
25. { AID\_KEYSTORE, "android.security.keystore" },
26. };

## Service Manager存在的意义

* Service Manager能集中管理系统内的所有服务，它能被施加权限控制，并不是任何进程都能注册服务的。
* Service Manager支持通过字符串名称来查找对应的Service。
* 由于各种原因的影响，Server进程可能生死无常。如果有了Service Manager做统一的管理，那么Client只要向Service Manager做查询，就能得到Server的最新信息。

用来管理开发者创建的各种Server，并且向Client提供查询Server远程接口的功能。

<http://blog.csdn.net/hu3167343/article/details/38441119>

# cmd

## 老罗

http://blog.csdn.net/qq\_35559358/article/details/78192863