[浅谈Service Manager成为Android进程间通信（IPC）机制Binder守护进程之路](http://blog.csdn.net/luoshengyang/article/details/6621566)

上一篇文章[Android进程间通信（IPC）机制Binder简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6618363)简要介绍了Android系统进程间通信机制Binder的总体架构，它由Client、Server、Service Manager和驱动程序Binder四个组件构成。本文着重介绍组件Service Manager，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。

        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。

       与Service Manager相关的源代码较多，这里不会完整去分析每一行代码，主要是带着Service Manager是如何成为整个Binder机制中的守护进程这条主线来一步一步地深入分析相关源代码，包括从用户空间到内核空间的相关源代码。希望读者在阅读下面的内容之前，先阅读一下前一篇文章提到的两个参考资料[Android深入浅出之Binder机制](http://www.cnblogs.com/innost/archive/2011/01/09/1931456.html)和[Android Binder设计与实现](http://disanji.net/2011/02/28/android-bnder-design/)，熟悉相关概念和数据结构，这有助于理解下面要分析的源代码。

       Service Manager在用户空间的源代码位于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：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **int** main(**int** argc, **char** \*\*argv)
2. {
3. **struct** binder\_state \*bs;
4. **void** \*svcmgr = BINDER\_SERVICE\_MANAGER;
6. bs = binder\_open(128\*1024);
8. **if** (binder\_become\_context\_manager(bs)) {
9. LOGE("cannot become context manager (%s)\n", strerror(errno));
10. **return** -1;
11. }
13. svcmgr\_handle = svcmgr;
14. binder\_loop(bs, svcmgr\_handler);
15. **return** 0;
16. }

        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。进入这三个功能之间，先来看一下这里用到的结构体binder\_state、宏BINDER\_SERVICE\_MANAGER的定义：

        struct binder\_state定义在frameworks/base/cmds/servicemanager/binder.c文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **struct** binder\_state
2. {
3. **int** fd;
4. **void** \*mapped;
5. unsigned mapsize;
6. };

        fd是文件描述符，即表示打开的/dev/binder设备文件描述符；mapped是把设备文件/dev/binder映射到进程空间的起始地址；mapsize是上述内存映射空间的大小。

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

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

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。

        函数首先是执行打开Binder设备文件的操作binder\_open，这个函数位于frameworks/base/cmds/servicemanager/binder.c文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

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. /\* TODO: check version \*/
28. **return** bs;
30. fail\_map:
31. close(bs->fd);
32. fail\_open:
33. free(bs);
34. **return** 0;
35. }

       通过文件操作函数open来打开/dev/binder设备文件。设备文件/dev/binder是在Binder驱动程序模块初始化的时候创建的，我们先看一下这个设备文件的创建过程。进入到kernel/common/drivers/staging/android目录中，打开binder.c文件，可以看到模块初始化入口binder\_init：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **struct** file\_operations binder\_fops = {
2. .owner = THIS\_MODULE,
3. .poll = binder\_poll,
4. .unlocked\_ioctl = binder\_ioctl,
5. .mmap = binder\_mmap,
6. .open = binder\_open,
7. .flush = binder\_flush,
8. .release = binder\_release,
9. };
11. **static** **struct** miscdevice binder\_miscdev = {
12. .minor = MISC\_DYNAMIC\_MINOR,
13. .name = "binder",
14. .fops = &binder\_fops
15. };
17. **static** **int** \_\_init binder\_init(**void**)
18. {
19. **int** ret;
21. binder\_proc\_dir\_entry\_root = proc\_mkdir("binder", NULL);
22. **if** (binder\_proc\_dir\_entry\_root)
23. binder\_proc\_dir\_entry\_proc = proc\_mkdir("proc", binder\_proc\_dir\_entry\_root);
24. ret = misc\_register(&binder\_miscdev);
25. **if** (binder\_proc\_dir\_entry\_root) {
26. create\_proc\_read\_entry("state", S\_IRUGO, binder\_proc\_dir\_entry\_root, binder\_read\_proc\_state, NULL);
27. create\_proc\_read\_entry("stats", S\_IRUGO, binder\_proc\_dir\_entry\_root, binder\_read\_proc\_stats, NULL);
28. create\_proc\_read\_entry("transactions", S\_IRUGO, binder\_proc\_dir\_entry\_root, binder\_read\_proc\_transactions, NULL);
29. create\_proc\_read\_entry("transaction\_log", S\_IRUGO, binder\_proc\_dir\_entry\_root, binder\_read\_proc\_transaction\_log, &binder\_transaction\_log);
30. create\_proc\_read\_entry("failed\_transaction\_log", S\_IRUGO, binder\_proc\_dir\_entry\_root, binder\_read\_proc\_transaction\_log, &binder\_transaction\_log\_failed);
31. }
32. **return** ret;
33. }
35. device\_initcall(binder\_init);

        创建设备文件的地方在misc\_register函数里面，关于misc设备的注册，我们在[Android日志系统驱动程序Logger源代码分析](http://blog.csdn.net/luoshengyang/article/details/6595744)一文中有提到，有兴趣的读取不访去了解一下。其余的逻辑主要是在/proc目录创建各种Binder相关的文件，供用户访问。从设备文件的操作方法binder\_fops可以看出，前面的binder\_open函数执行语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. bs->fd = open("/dev/binder", O\_RDWR);

        就进入到Binder驱动程序的binder\_open函数了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **int** binder\_open(**struct** inode \*nodp, **struct** file \*filp)
2. {
3. **struct** binder\_proc \*proc;
5. **if** (binder\_debug\_mask & BINDER\_DEBUG\_OPEN\_CLOSE)
6. printk(KERN\_INFO "binder\_open: %d:%d\n", current->group\_leader->pid, current->pid);
8. proc = kzalloc(**sizeof**(\*proc), GFP\_KERNEL);
9. **if** (proc == NULL)
10. **return** -ENOMEM;
11. get\_task\_struct(current);
12. proc->tsk = current;
13. INIT\_LIST\_HEAD(&proc->todo);
14. init\_waitqueue\_head(&proc->wait);
15. proc->default\_priority = task\_nice(current);
16. mutex\_lock(&binder\_lock);
17. binder\_stats.obj\_created[BINDER\_STAT\_PROC]++;
18. hlist\_add\_head(&proc->proc\_node, &binder\_procs);
19. proc->pid = current->group\_leader->pid;
20. INIT\_LIST\_HEAD(&proc->delivered\_death);
21. filp->private\_data = proc;
22. mutex\_unlock(&binder\_lock);
24. **if** (binder\_proc\_dir\_entry\_proc) {
25. **char** strbuf[11];
26. snprintf(strbuf, **sizeof**(strbuf), "%u", proc->pid);
27. remove\_proc\_entry(strbuf, binder\_proc\_dir\_entry\_proc);
28. create\_proc\_read\_entry(strbuf, S\_IRUGO, binder\_proc\_dir\_entry\_proc, binder\_read\_proc\_proc, proc);
29. }
31. **return** 0;
32. }

         这个函数的主要作用是创建一个struct binder\_proc数据结构来保存打开设备文件/dev/binder的进程的上下文信息，并且将这个进程上下文信息保存在打开文件结构struct file的私有数据成员变量private\_data中，这样，在执行其它文件操作时，就通过打开文件结构struct file来取回这个进程上下文信息了。这个进程上下文信息同时还会保存在一个全局哈希表binder\_procs中，驱动程序内部使用。binder\_procs定义在文件的开头：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** HLIST\_HEAD(binder\_procs);

        结构体struct binder\_proc也是定义在kernel/common/drivers/staging/android/binder.c文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **struct** binder\_proc {
2. **struct** hlist\_node proc\_node;
3. **struct** rb\_root threads;
4. **struct** rb\_root nodes;
5. **struct** rb\_root refs\_by\_desc;
6. **struct** rb\_root refs\_by\_node;
7. **int** pid;
8. **struct** vm\_area\_struct \*vma;
9. **struct** task\_struct \*tsk;
10. **struct** files\_struct \*files;
11. **struct** hlist\_node deferred\_work\_node;
12. **int** deferred\_work;
13. **void** \*buffer;
14. **ptrdiff\_t** user\_buffer\_offset;
16. **struct** list\_head buffers;
17. **struct** rb\_root free\_buffers;
18. **struct** rb\_root allocated\_buffers;
19. **size\_t** free\_async\_space;
21. **struct** page \*\*pages;
22. **size\_t** buffer\_size;
23. uint32\_t buffer\_free;
24. **struct** list\_head todo;
25. wait\_queue\_head\_t wait;
26. **struct** binder\_stats stats;
27. **struct** list\_head delivered\_death;
28. **int** max\_threads;
29. **int** requested\_threads;
30. **int** requested\_threads\_started;
31. **int** ready\_threads;
32. **long** default\_priority;
33. };

        这个结构体的成员比较多，这里就不一一说明了，简单解释一下四个成员变量threads、nodes、 refs\_by\_desc和refs\_by\_node，其它的我们在遇到的时候再详细解释。这四个成员变量都是表示红黑树的节点，也就是说，binder\_proc分别挂会四个红黑树下。threads树用来保存binder\_proc进程内用于处理用户请求的线程，它的最大数量由max\_threads来决定；node树成用来保存binder\_proc进程内的Binder实体；refs\_by\_desc树和refs\_by\_node树用来保存binder\_proc进程内的Binder引用，即引用的其它进程的Binder实体，它分别用两种方式来组织红黑树，一种是以句柄作来key值来组织，一种是以引用的实体节点的地址值作来key值来组织，它们都是表示同一样东西，只不过是为了内部查找方便而用两个红黑树来表示。

         这样，打开设备文件/dev/binder的操作就完成了，接着是对打开的设备文件进行内存映射操作mmap：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. bs->mapped = mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0);

         对应Binder驱动程序的binder\_mmap函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **int** binder\_mmap(**struct** file \*filp, **struct** vm\_area\_struct \*vma)
2. {
3. **int** ret;
4. **struct** vm\_struct \*area;
5. **struct** binder\_proc \*proc = filp->private\_data;
6. **const** **char** \*failure\_string;
7. **struct** binder\_buffer \*buffer;
9. **if** ((vma->vm\_end - vma->vm\_start) > SZ\_4M)
10. vma->vm\_end = vma->vm\_start + SZ\_4M;
12. **if** (binder\_debug\_mask & BINDER\_DEBUG\_OPEN\_CLOSE)
13. printk(KERN\_INFO
14. "binder\_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
15. proc->pid, vma->vm\_start, vma->vm\_end,
16. (vma->vm\_end - vma->vm\_start) / SZ\_1K, vma->vm\_flags,
17. (unsigned **long**)pgprot\_val(vma->vm\_page\_prot));
19. **if** (vma->vm\_flags & FORBIDDEN\_MMAP\_FLAGS) {
20. ret = -EPERM;
21. failure\_string = "bad vm\_flags";
22. **goto** err\_bad\_arg;
23. }
24. vma->vm\_flags = (vma->vm\_flags | VM\_DONTCOPY) & ~VM\_MAYWRITE;
26. **if** (proc->buffer) {
27. ret = -EBUSY;
28. failure\_string = "already mapped";
29. **goto** err\_already\_mapped;
30. }
32. area = get\_vm\_area(vma->vm\_end - vma->vm\_start, VM\_IOREMAP);
33. **if** (area == NULL) {
34. ret = -ENOMEM;
35. failure\_string = "get\_vm\_area";
36. **goto** err\_get\_vm\_area\_failed;
37. }
38. proc->buffer = area->addr;
39. proc->user\_buffer\_offset = vma->vm\_start - (**uintptr\_t**)proc->buffer;
41. #ifdef CONFIG\_CPU\_CACHE\_VIPT
42. **if** (cache\_is\_vipt\_aliasing()) {
43. **while** (CACHE\_COLOUR((vma->vm\_start ^ (uint32\_t)proc->buffer))) {
44. printk(KERN\_INFO "binder\_mmap: %d %lx-%lx maps %p bad alignment\n", proc->pid, vma->vm\_start, vma->vm\_end, proc->buffer);
45. vma->vm\_start += PAGE\_SIZE;
46. }
47. }
48. #endif
49. proc->pages = kzalloc(**sizeof**(proc->pages[0]) \* ((vma->vm\_end - vma->vm\_start) / PAGE\_SIZE), GFP\_KERNEL);
50. **if** (proc->pages == NULL) {
51. ret = -ENOMEM;
52. failure\_string = "alloc page array";
53. **goto** err\_alloc\_pages\_failed;
54. }
55. proc->buffer\_size = vma->vm\_end - vma->vm\_start;
57. vma->vm\_ops = &binder\_vm\_ops;
58. vma->vm\_private\_data = proc;
60. **if** (binder\_update\_page\_range(proc, 1, proc->buffer, proc->buffer + PAGE\_SIZE, vma)) {
61. ret = -ENOMEM;
62. failure\_string = "alloc small buf";
63. **goto** err\_alloc\_small\_buf\_failed;
64. }
65. buffer = proc->buffer;
66. INIT\_LIST\_HEAD(&proc->buffers);
67. list\_add(&buffer->entry, &proc->buffers);
68. buffer->free = 1;
69. binder\_insert\_free\_buffer(proc, buffer);
70. proc->free\_async\_space = proc->buffer\_size / 2;
71. barrier();
72. proc->files = get\_files\_struct(current);
73. proc->vma = vma;
75. /\*printk(KERN\_INFO "binder\_mmap: %d %lx-%lx maps %p\n", proc->pid, vma->vm\_start, vma->vm\_end, proc->buffer);\*/
76. **return** 0;
78. err\_alloc\_small\_buf\_failed:
79. kfree(proc->pages);
80. proc->pages = NULL;
81. err\_alloc\_pages\_failed:
82. vfree(proc->buffer);
83. proc->buffer = NULL;
84. err\_get\_vm\_area\_failed:
85. err\_already\_mapped:
86. err\_bad\_arg:
87. printk(KERN\_ERR "binder\_mmap: %d %lx-%lx %s failed %d\n", proc->pid, vma->vm\_start, vma->vm\_end, failure\_string, ret);
88. **return** ret;
89. }

         函数首先通过filp->private\_data得到在打开设备文件/dev/binder时创建的struct binder\_proc结构。内存映射信息放在vma参数中，注意，这里的vma的数据类型是struct vm\_area\_struct，它表示的是一块连续的虚拟地址空间区域，在函数变量声明的地方，我们还看到有一个类似的结构体struct vm\_struct，这个数据结构也是表示一块连续的虚拟地址空间区域，那么，这两者的区别是什么呢？在Linux中，struct vm\_area\_struct表示的虚拟地址是给进程使用的，而struct vm\_struct表示的虚拟地址是给内核使用的，它们对应的物理页面都可以是不连续的。struct vm\_area\_struct表示的地址空间范围是0~3G，而struct vm\_struct表示的地址空间范围是(3G + 896M + 8M) ~ 4G。struct vm\_struct表示的地址空间范围为什么不是3G~4G呢？原来，3G ~ (3G + 896M)范围的地址是用来映射连续的物理页面的，这个范围的虚拟地址和对应的实际物理地址有着简单的对应关系，即对应0~896M的物理地址空间，而(3G + 896M) ~ (3G + 896M + 8M)是安全保护区域（例如，所有指向这8M地址空间的指针都是非法的），因此struct vm\_struct使用(3G + 896M + 8M) ~ 4G地址空间来映射非连续的物理页面。有关Linux的内存管理知识，可以参考[Android学习启动篇](http://blog.csdn.net/luoshengyang/article/details/6557518)一文提到的《Understanding the Linux Kernel》一书中的第8章。

        这里为什么会同时使用进程虚拟地址空间和内核虚拟地址空间来映射同一个物理页面呢？这就是Binder进程间通信机制的精髓所在了，同一个物理页面，一方映射到进程虚拟地址空间，一方面映射到内核虚拟地址空间，这样，进程和内核之间就可以减少一次内存拷贝了，提到了进程间通信效率。举个例子如，Client要将一块内存数据传递给Server，一般的做法是，Client将这块数据从它的进程空间拷贝到内核空间中，然后内核再将这个数据从内核空间拷贝到Server的进程空间，这样，Server就可以访问这个数据了。但是在这种方法中，执行了两次内存拷贝操作，而采用我们上面提到的方法，只需要把Client进程空间的数据拷贝一次到内核空间，然后Server与内核共享这个数据就可以了，整个过程只需要执行一次内存拷贝，提高了效率。

        binder\_mmap的原理讲完了，这个函数的逻辑就好理解了。不过，这里还是先要解释一下struct binder\_proc结构体的几个成员变量。buffer成员变量是一个void\*指针，它表示要映射的物理内存在内核空间中的起始位置；buffer\_size成员变量是一个size\_t类型的变量，表示要映射的内存的大小；pages成员变量是一个struct page\*类型的数组，struct page是用来描述物理页面的数据结构；user\_buffer\_offset成员变量是一个ptrdiff\_t类型的变量，它表示的是内核使用的虚拟地址与进程使用的虚拟地址之间的差值，即如果某个物理页面在内核空间中对应的虚拟地址是addr的话，那么这个物理页面在进程空间对应的虚拟地址就为addr + user\_buffer\_offset。

        再解释一下Binder驱动程序管理这个内存映射地址空间的方法，即是如何管理buffer ~ (buffer + buffer\_size)这段地址空间的，这个地址空间被划分为一段一段来管理，每一段是结构体struct binder\_buffer来描述：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **struct** binder\_buffer {
2. **struct** list\_head entry; /\* free and allocated entries by addesss \*/
3. **struct** rb\_node rb\_node; /\* free entry by size or allocated entry \*/
4. /\* by address \*/
5. unsigned free : 1;
6. unsigned allow\_user\_free : 1;
7. unsigned async\_transaction : 1;
8. unsigned debug\_id : 29;
10. **struct** binder\_transaction \*transaction;
12. **struct** binder\_node \*target\_node;
13. **size\_t** data\_size;
14. **size\_t** offsets\_size;
15. uint8\_t data[0];
16. };

        每一个binder\_buffer通过其成员entry按从低址到高地址连入到struct binder\_proc中的buffers表示的链表中去，同时，每一个binder\_buffer又分为正在使用的和空闲的，通过free成员变量来区分，空闲的binder\_buffer通过成员变量rb\_node连入到struct binder\_proc中的free\_buffers表示的红黑树中去，正在使用的binder\_buffer通过成员变量rb\_node连入到struct binder\_proc中的allocated\_buffers表示的红黑树中去。这样做当然是为了方便查询和维护这块地址空间了，这一点我们可以从其它的代码中看到，等遇到的时候我们再分析。

        终于可以回到binder\_mmap这个函数来了，首先是对参数作一些健康体检（sanity check），例如，要映射的内存大小不能超过SIZE\_4M，即4M，回到service\_manager.c中的main 函数，这里传进来的值是128 \* 1024个字节，即128K，这个检查没有问题。通过健康体检后，调用get\_vm\_area函数获得一个空闲的vm\_struct区间，并初始化proc结构体的buffer、user\_buffer\_offset、pages和buffer\_size和成员变量，接着调用binder\_update\_page\_range来为虚拟地址空间proc->buffer ~ proc->buffer + PAGE\_SIZE分配一个空闲的物理页面，同时这段地址空间使用一个binder\_buffer来描述，分别插入到proc->buffers链表和proc->free\_buffers红黑树中去，最后，还初始化了proc结构体的free\_async\_space、files和vma三个成员变量。

        这里，我们继续进入到binder\_update\_page\_range函数中去看一下Binder驱动程序是如何实现把一个物理页面同时映射到内核空间和进程空间去的：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **int** binder\_update\_page\_range(**struct** binder\_proc \*proc, **int** allocate,
2. **void** \*start, **void** \*end, **struct** vm\_area\_struct \*vma)
3. {
4. **void** \*page\_addr;
5. unsigned **long** user\_page\_addr;
6. **struct** vm\_struct tmp\_area;
7. **struct** page \*\*page;
8. **struct** mm\_struct \*mm;
10. **if** (binder\_debug\_mask & BINDER\_DEBUG\_BUFFER\_ALLOC)
11. printk(KERN\_INFO "binder: %d: %s pages %p-%p\n",
12. proc->pid, allocate ? "allocate" : "free", start, end);
14. **if** (end <= start)
15. **return** 0;
17. **if** (vma)
18. mm = NULL;
19. **else**
20. mm = get\_task\_mm(proc->tsk);
22. **if** (mm) {
23. down\_write(&mm->mmap\_sem);
24. vma = proc->vma;
25. }
27. **if** (allocate == 0)
28. **goto** free\_range;
30. **if** (vma == NULL) {
31. printk(KERN\_ERR "binder: %d: binder\_alloc\_buf failed to "
32. "map pages in userspace, no vma\n", proc->pid);
33. **goto** err\_no\_vma;
34. }
36. **for** (page\_addr = start; page\_addr < end; page\_addr += PAGE\_SIZE) {
37. **int** ret;
38. **struct** page \*\*page\_array\_ptr;
39. page = &proc->pages[(page\_addr - proc->buffer) / PAGE\_SIZE];
41. BUG\_ON(\*page);
42. \*page = alloc\_page(GFP\_KERNEL | \_\_GFP\_ZERO);
43. **if** (\*page == NULL) {
44. printk(KERN\_ERR "binder: %d: binder\_alloc\_buf failed "
45. "for page at %p\n", proc->pid, page\_addr);
46. **goto** err\_alloc\_page\_failed;
47. }
48. tmp\_area.addr = page\_addr;
49. tmp\_area.size = PAGE\_SIZE + PAGE\_SIZE /\* guard page? \*/;
50. page\_array\_ptr = page;
51. ret = map\_vm\_area(&tmp\_area, PAGE\_KERNEL, &page\_array\_ptr);
52. **if** (ret) {
53. printk(KERN\_ERR "binder: %d: binder\_alloc\_buf failed "
54. "to map page at %p in kernel\n",
55. proc->pid, page\_addr);
56. **goto** err\_map\_kernel\_failed;
57. }
58. user\_page\_addr =
59. (**uintptr\_t**)page\_addr + proc->user\_buffer\_offset;
60. ret = vm\_insert\_page(vma, user\_page\_addr, page[0]);
61. **if** (ret) {
62. printk(KERN\_ERR "binder: %d: binder\_alloc\_buf failed "
63. "to map page at %lx in userspace\n",
64. proc->pid, user\_page\_addr);
65. **goto** err\_vm\_insert\_page\_failed;
66. }
67. /\* vm\_insert\_page does not seem to increment the refcount \*/
68. }
69. **if** (mm) {
70. up\_write(&mm->mmap\_sem);
71. mmput(mm);
72. }
73. **return** 0;
75. free\_range:
76. **for** (page\_addr = end - PAGE\_SIZE; page\_addr >= start;
77. page\_addr -= PAGE\_SIZE) {
78. page = &proc->pages[(page\_addr - proc->buffer) / PAGE\_SIZE];
79. **if** (vma)
80. zap\_page\_range(vma, (**uintptr\_t**)page\_addr +
81. proc->user\_buffer\_offset, PAGE\_SIZE, NULL);
82. err\_vm\_insert\_page\_failed:
83. unmap\_kernel\_range((unsigned **long**)page\_addr, PAGE\_SIZE);
84. err\_map\_kernel\_failed:
85. \_\_free\_page(\*page);
86. \*page = NULL;
87. err\_alloc\_page\_failed:
88. ;
89. }
90. err\_no\_vma:
91. **if** (mm) {
92. up\_write(&mm->mmap\_sem);
93. mmput(mm);
94. }
95. **return** -ENOMEM;
96. }

        这个函数既可以分配物理页面，也可以用来释放物理页面，通过allocate参数来区别，这里我们只关注分配物理页面的情况。要分配物理页面的虚拟地址空间范围为(start ~ end)，函数前面的一些检查逻辑就不看了，直接看中间的for循环：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **for** (page\_addr = start; page\_addr < end; page\_addr += PAGE\_SIZE) {
2. **int** ret;
3. **struct** page \*\*page\_array\_ptr;
4. page = &proc->pages[(page\_addr - proc->buffer) / PAGE\_SIZE];
6. BUG\_ON(\*page);
7. \*page = alloc\_page(GFP\_KERNEL | \_\_GFP\_ZERO);
8. **if** (\*page == NULL) {
9. printk(KERN\_ERR "binder: %d: binder\_alloc\_buf failed "
10. "for page at %p\n", proc->pid, page\_addr);
11. **goto** err\_alloc\_page\_failed;
12. }
13. tmp\_area.addr = page\_addr;
14. tmp\_area.size = PAGE\_SIZE + PAGE\_SIZE /\* guard page? \*/;
15. page\_array\_ptr = page;
16. ret = map\_vm\_area(&tmp\_area, PAGE\_KERNEL, &page\_array\_ptr);
17. **if** (ret) {
18. printk(KERN\_ERR "binder: %d: binder\_alloc\_buf failed "
19. "to map page at %p in kernel\n",
20. proc->pid, page\_addr);
21. **goto** err\_map\_kernel\_failed;
22. }
23. user\_page\_addr =
24. (**uintptr\_t**)page\_addr + proc->user\_buffer\_offset;
25. ret = vm\_insert\_page(vma, user\_page\_addr, page[0]);
26. **if** (ret) {
27. printk(KERN\_ERR "binder: %d: binder\_alloc\_buf failed "
28. "to map page at %lx in userspace\n",
29. proc->pid, user\_page\_addr);
30. **goto** err\_vm\_insert\_page\_failed;
31. }
32. /\* vm\_insert\_page does not seem to increment the refcount \*/
33. }

        首先是调用alloc\_page来分配一个物理页面，这个函数返回一个struct page物理页面描述符，根据这个描述的内容初始化好struct vm\_struct tmp\_area结构体，然后通过map\_vm\_area将这个物理页面插入到tmp\_area描述的内核空间去，接着通过page\_addr + proc->user\_buffer\_offset获得进程虚拟空间地址，并通过vm\_insert\_page函数将这个物理页面插入到进程地址空间去，参数vma代表了要插入的进程的地址空间。  
       这样，frameworks/base/cmds/servicemanager/binder.c文件中的binder\_open函数就描述完了，回到frameworks/base/cmds/servicemanager/service\_manager.c文件中的main函数，下一步就是调用binder\_become\_context\_manager来通知Binder驱动程序自己是Binder机制的上下文管理者，即守护进程。binder\_become\_context\_manager函数位于frameworks/base/cmds/servicemanager/binder.c文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

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

       这里通过调用ioctl文件操作函数来通知Binder驱动程序自己是守护进程，命令号是BINDER\_SET\_CONTEXT\_MGR，没有参数。BINDER\_SET\_CONTEXT\_MGR定义为：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. #define BINDER\_SET\_CONTEXT\_MGR      \_IOW('b', 7, int)

       这样就进入到Binder驱动程序的binder\_ioctl函数，我们只关注BINDER\_SET\_CONTEXT\_MGR命令：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **long** binder\_ioctl(**struct** file \*filp, unsigned **int** cmd, unsigned **long** arg)
2. {
3. **int** ret;
4. **struct** binder\_proc \*proc = filp->private\_data;
5. **struct** binder\_thread \***thread**;
6. unsigned **int** size = \_IOC\_SIZE(cmd);
7. **void** \_\_user \*ubuf = (**void** \_\_user \*)arg;
9. /\*printk(KERN\_INFO "binder\_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);\*/
11. ret = wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);
12. **if** (ret)
13. **return** ret;
15. mutex\_lock(&binder\_lock);
16. **thread** = binder\_get\_thread(proc);
17. **if** (**thread** == NULL) {
18. ret = -ENOMEM;
19. **goto** err;
20. }
22. **switch** (cmd) {
23. ......
24. **case** BINDER\_SET\_CONTEXT\_MGR:
25. **if** (binder\_context\_mgr\_node != NULL) {
26. printk(KERN\_ERR "binder: BINDER\_SET\_CONTEXT\_MGR already set\n");
27. ret = -EBUSY;
28. **goto** err;
29. }
30. **if** (binder\_context\_mgr\_uid != -1) {
31. **if** (binder\_context\_mgr\_uid != current->cred->euid) {
32. printk(KERN\_ERR "binder: BINDER\_SET\_"
33. "CONTEXT\_MGR bad uid %d != %d\n",
34. current->cred->euid,
35. binder\_context\_mgr\_uid);
36. ret = -EPERM;
37. **goto** err;
38. }
39. } **else**
40. binder\_context\_mgr\_uid = current->cred->euid;
41. binder\_context\_mgr\_node = binder\_new\_node(proc, NULL, NULL);
42. **if** (binder\_context\_mgr\_node == NULL) {
43. ret = -ENOMEM;
44. **goto** err;
45. }
46. binder\_context\_mgr\_node->local\_weak\_refs++;
47. binder\_context\_mgr\_node->local\_strong\_refs++;
48. binder\_context\_mgr\_node->has\_strong\_ref = 1;
49. binder\_context\_mgr\_node->has\_weak\_ref = 1;
50. **break**;
51. ......
52. **default**:
53. ret = -EINVAL;
54. **goto** err;
55. }
56. ret = 0;
57. err:
58. **if** (**thread**)
59. **thread**->looper &= ~BINDER\_LOOPER\_STATE\_NEED\_RETURN;
60. mutex\_unlock(&binder\_lock);
61. wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);
62. **if** (ret && ret != -ERESTARTSYS)
63. printk(KERN\_INFO "binder: %d:%d ioctl %x %lx returned %d\n", proc->pid, current->pid, cmd, arg, ret);
64. **return** ret;
65. }

        继续分析这个函数之前，又要解释两个数据结构了，一个是struct binder\_thread结构体，顾名思久，它表示一个线程，这里就是执行binder\_become\_context\_manager函数的线程了。

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **struct** binder\_thread {
2. **struct** binder\_proc \*proc;
3. **struct** rb\_node rb\_node;
4. **int** pid;
5. **int** looper;
6. **struct** binder\_transaction \*transaction\_stack;
7. **struct** list\_head todo;
8. uint32\_t return\_error; /\* Write failed, return error code in read buf \*/
9. uint32\_t return\_error2; /\* Write failed, return error code in read \*/
10. /\* buffer. Used when sending a reply to a dead process that \*/
11. /\* we are also waiting on \*/
12. wait\_queue\_head\_t wait;
13. **struct** binder\_stats stats;
14. };

       proc表示这个线程所属的进程。struct binder\_proc有一个成员变量threads，它的类型是rb\_root，它表示一查红黑树，把属于这个进程的所有线程都组织起来，struct binder\_thread的成员变量rb\_node就是用来链入这棵红黑树的节点了。looper成员变量表示线程的状态，它可以取下面这几个值：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **enum** {
2. BINDER\_LOOPER\_STATE\_REGISTERED  = 0x01,
3. BINDER\_LOOPER\_STATE\_ENTERED     = 0x02,
4. BINDER\_LOOPER\_STATE\_EXITED      = 0x04,
5. BINDER\_LOOPER\_STATE\_INVALID     = 0x08,
6. BINDER\_LOOPER\_STATE\_WAITING     = 0x10,
7. BINDER\_LOOPER\_STATE\_NEED\_RETURN = 0x20
8. };

        其余的成员变量，transaction\_stack表示线程正在处理的事务，todo表示发往该线程的数据列表，return\_error和return\_error2表示操作结果返回码，wait用来阻塞线程等待某个事件的发生，stats用来保存一些统计信息。这些成员变量遇到的时候再分析它们的作用。

        另外一个数据结构是struct binder\_node，它表示一个binder实体：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **struct** binder\_node {
2. **int** debug\_id;
3. **struct** binder\_work work;
4. **union** {
5. **struct** rb\_node rb\_node;
6. **struct** hlist\_node dead\_node;
7. };
8. **struct** binder\_proc \*proc;
9. **struct** hlist\_head refs;
10. **int** internal\_strong\_refs;
11. **int** local\_weak\_refs;
12. **int** local\_strong\_refs;
13. **void** \_\_user \*ptr;
14. **void** \_\_user \*cookie;
15. unsigned has\_strong\_ref : 1;
16. unsigned pending\_strong\_ref : 1;
17. unsigned has\_weak\_ref : 1;
18. unsigned pending\_weak\_ref : 1;
19. unsigned has\_async\_transaction : 1;
20. unsigned accept\_fds : 1;
21. **int** min\_priority : 8;
22. **struct** list\_head async\_todo;
23. };

        rb\_node和dead\_node组成一个联合体。 如果这个Binder实体还在正常使用，则使用rb\_node来连入proc->nodes所表示的红黑树的节点，这棵红黑树用来组织属于这个进程的所有Binder实体；如果这个Binder实体所属的进程已经销毁，而这个Binder实体又被其它进程所引用，则这个Binder实体通过dead\_node进入到一个哈希表中去存放。proc成员变量就是表示这个Binder实例所属于进程了。refs成员变量把所有引用了该Binder实体的Binder引用连接起来构成一个链表。internal\_strong\_refs、local\_weak\_refs和local\_strong\_refs表示这个Binder实体的引用计数。ptr和cookie成员变量分别表示这个Binder实体在用户空间的地址以及附加数据。其余的成员变量就不描述了，遇到的时候再分析。

        现在回到binder\_ioctl函数中，首先是通过filp->private\_data获得proc变量，这里binder\_mmap函数是一样的。接着通过binder\_get\_thread函数获得线程信息，我们来看一下这个函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **struct** binder\_thread \*binder\_get\_thread(**struct** binder\_proc \*proc)
2. {
3. **struct** binder\_thread \***thread** = NULL;
4. **struct** rb\_node \*parent = NULL;
5. **struct** rb\_node \*\*p = &proc->threads.rb\_node;
7. **while** (\*p) {
8. parent = \*p;
9. **thread** = rb\_entry(parent, **struct** binder\_thread, rb\_node);
11. **if** (current->pid < **thread**->pid)
12. p = &(\*p)->rb\_left;
13. **else** **if** (current->pid > **thread**->pid)
14. p = &(\*p)->rb\_right;
15. **else**
16. **break**;
17. }
18. **if** (\*p == NULL) {
19. **thread** = kzalloc(**sizeof**(\***thread**), GFP\_KERNEL);
20. **if** (**thread** == NULL)
21. **return** NULL;
22. binder\_stats.obj\_created[BINDER\_STAT\_THREAD]++;
23. **thread**->proc = proc;
24. **thread**->pid = current->pid;
25. init\_waitqueue\_head(&**thread**->wait);
26. INIT\_LIST\_HEAD(&**thread**->todo);
27. rb\_link\_node(&**thread**->rb\_node, parent, p);
28. rb\_insert\_color(&**thread**->rb\_node, &proc->threads);
29. **thread**->looper |= BINDER\_LOOPER\_STATE\_NEED\_RETURN;
30. **thread**->return\_error = BR\_OK;
31. **thread**->return\_error2 = BR\_OK;
32. }
33. **return** **thread**;
34. }

        这里把当前线程current的pid作为键值，在进程proc->threads表示的红黑树中进行查找，看是否已经为当前线程创建过了binder\_thread信息。在这个场景下，由于当前线程是第一次进到这里，所以肯定找不到，即\*p == NULL成立，于是，就为当前线程创建一个线程上下文信息结构体binder\_thread，并初始化相应成员变量，并插入到proc->threads所表示的红黑树中去，下次要使用时就可以从proc中找到了。注意，这里的thread->looper = BINDER\_LOOPER\_STATE\_NEED\_RETURN。

        回到binder\_ioctl函数，继续往下面，有两个全局变量binder\_context\_mgr\_node和binder\_context\_mgr\_uid，它定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **struct** binder\_node \*binder\_context\_mgr\_node;
2. **static** uid\_t binder\_context\_mgr\_uid = -1;

        binder\_context\_mgr\_node用来表示Service Manager实体，binder\_context\_mgr\_uid表示Service Manager守护进程的uid。在这个场景下，由于当前线程是第一次进到这里，所以binder\_context\_mgr\_node为NULL，binder\_context\_mgr\_uid为-1，于是初始化binder\_context\_mgr\_uid为current->cred->euid，这样，当前线程就成为Binder机制的守护进程了，并且通过binder\_new\_node为Service Manager创建Binder实体：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **struct** binder\_node \*
2. binder\_new\_node(**struct** binder\_proc \*proc, **void** \_\_user \*ptr, **void** \_\_user \*cookie)
3. {
4. **struct** rb\_node \*\*p = &proc->nodes.rb\_node;
5. **struct** rb\_node \*parent = NULL;
6. **struct** binder\_node \*node;
8. **while** (\*p) {
9. parent = \*p;
10. node = rb\_entry(parent, **struct** binder\_node, rb\_node);
12. **if** (ptr < node->ptr)
13. p = &(\*p)->rb\_left;
14. **else** **if** (ptr > node->ptr)
15. p = &(\*p)->rb\_right;
16. **else**
17. **return** NULL;
18. }
20. node = kzalloc(**sizeof**(\*node), GFP\_KERNEL);
21. **if** (node == NULL)
22. **return** NULL;
23. binder\_stats.obj\_created[BINDER\_STAT\_NODE]++;
24. rb\_link\_node(&node->rb\_node, parent, p);
25. rb\_insert\_color(&node->rb\_node, &proc->nodes);
26. node->debug\_id = ++binder\_last\_id;
27. node->proc = proc;
28. node->ptr = ptr;
29. node->cookie = cookie;
30. node->work.type = BINDER\_WORK\_NODE;
31. INIT\_LIST\_HEAD(&node->work.entry);
32. INIT\_LIST\_HEAD(&node->async\_todo);
33. **if** (binder\_debug\_mask & BINDER\_DEBUG\_INTERNAL\_REFS)
34. printk(KERN\_INFO "binder: %d:%d node %d u%p c%p created\n",
35. proc->pid, current->pid, node->debug\_id,
36. node->ptr, node->cookie);
37. **return** node;
38. }

        注意，这里传进来的ptr和cookie均为NULL。函数首先检查proc->nodes红黑树中是否已经存在以ptr为键值的node，如果已经存在，就返回NULL。在这个场景下，由于当前线程是第一次进入到这里，所以肯定不存在，于是就新建了一个ptr为NULL的binder\_node，并且初始化其它成员变量，并插入到proc->nodes红黑树中去。

        binder\_new\_node返回到binder\_ioctl函数后，就把新建的binder\_node指针保存在binder\_context\_mgr\_node中了，紧接着，又初始化了binder\_context\_mgr\_node的引用计数值。

        这样，BINDER\_SET\_CONTEXT\_MGR命令就执行完毕了，binder\_ioctl函数返回之前，执行了下面语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **if** (**thread**)
2. **thread**->looper &= ~BINDER\_LOOPER\_STATE\_NEED\_RETURN;

       回忆上面执行binder\_get\_thread时，thread->looper = BINDER\_LOOPER\_STATE\_NEED\_RETURN，执行了这条语句后，thread->looper = 0。

       回到frameworks/base/cmds/servicemanager/service\_manager.c文件中的main函数，下一步就是调用binder\_loop函数进入循环，等待Client来请求了。binder\_loop函数定义在frameworks/base/cmds/servicemanager/binder.c文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **void** binder\_loop(**struct** binder\_state \*bs, binder\_handler func)
2. {
3. **int** res;
4. **struct** binder\_write\_read bwr;
5. unsigned readbuf[32];
7. bwr.write\_size = 0;
8. bwr.write\_consumed = 0;
9. bwr.write\_buffer = 0;
11. readbuf[0] = BC\_ENTER\_LOOPER;
12. binder\_write(bs, readbuf, **sizeof**(unsigned));
14. **for** (;;) {
15. bwr.read\_size = **sizeof**(readbuf);
16. bwr.read\_consumed = 0;
17. bwr.read\_buffer = (unsigned) readbuf;
19. res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);
21. **if** (res < 0) {
22. LOGE("binder\_loop: ioctl failed (%s)\n", strerror(errno));
23. **break**;
24. }
26. res = binder\_parse(bs, 0, readbuf, bwr.read\_consumed, func);
27. **if** (res == 0) {
28. LOGE("binder\_loop: unexpected reply?!\n");
29. **break**;
30. }
31. **if** (res < 0) {
32. LOGE("binder\_loop: io error %d %s\n", res, strerror(errno));
33. **break**;
34. }
35. }
36. }

       首先是通过binder\_write函数执行BC\_ENTER\_LOOPER命令告诉Binder驱动程序， Service Manager要进入循环了。

       这里又要介绍一下设备文件/dev/binder文件操作函数ioctl的操作码BINDER\_WRITE\_READ了，首先看定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. #define BINDER\_WRITE\_READ           \_IOWR('b', 1, struct binder\_write\_read)

       这个io操作码有一个参数，形式为struct binder\_write\_read：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **struct** binder\_write\_read {
2. **signed** **long** write\_size; /\* bytes to write \*/
3. **signed** **long** write\_consumed; /\* bytes consumed by driver \*/
4. unsigned **long**   write\_buffer;
5. **signed** **long** read\_size;  /\* bytes to read \*/
6. **signed** **long** read\_consumed;  /\* bytes consumed by driver \*/
7. unsigned **long**   read\_buffer;
8. };

      这里顺便说一下，用户空间程序和Binder驱动程序交互大多数都是通过BINDER\_WRITE\_READ命令的，write\_bufffer和read\_buffer所指向的数据结构还指定了具体要执行的操作，write\_bufffer和read\_buffer所指向的结构体是struct binder\_transaction\_data：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **struct** binder\_transaction\_data {
2. /\* The first two are only used for bcTRANSACTION and brTRANSACTION,
3. \* identifying the target and contents of the transaction.
4. \*/
5. **union** {
6. **size\_t**  handle; /\* target descriptor of command transaction \*/
7. **void**    \*ptr;   /\* target descriptor of return transaction \*/
8. } target;
9. **void**        \*cookie;    /\* target object cookie \*/
10. unsigned **int**    code;       /\* transaction command \*/
12. /\* General information about the transaction. \*/
13. unsigned **int**    flags;
14. pid\_t       sender\_pid;
15. uid\_t       sender\_euid;
16. **size\_t**      data\_size;  /\* number of bytes of data \*/
17. **size\_t**      offsets\_size;   /\* number of bytes of offsets \*/
19. /\* If this transaction is inline, the data immediately
20. \* follows here; otherwise, it ends with a pointer to
21. \* the data buffer.
22. \*/
23. **union** {
24. **struct** {
25. /\* transaction data \*/
26. **const** **void**  \*buffer;
27. /\* offsets from buffer to flat\_binder\_object structs \*/
28. **const** **void**  \*offsets;
29. } ptr;
30. uint8\_t buf[8];
31. } data;
32. };

       有一个联合体target，当这个BINDER\_WRITE\_READ命令的目标对象是本地Binder实体时，就使用ptr来表示这个对象在本进程中的地址，否则就使用handle来表示这个Binder实体的引用。只有目标对象是Binder实体时，cookie成员变量才有意义，表示一些附加数据，由Binder实体来解释这个个附加数据。code表示要对目标对象请求的命令代码，有很多请求代码，这里就不列举了，在这个场景中，就是BC\_ENTER\_LOOPER了，用来告诉Binder驱动程序， Service Manager要进入循环了。其余的请求命令代码可以参考kernel/common/drivers/staging/android/binder.h文件中定义的两个枚举类型BinderDriverReturnProtocol和BinderDriverCommandProtocol。

       flags成员变量表示事务标志：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **enum** transaction\_flags {
2. TF\_ONE\_WAY  = 0x01, /\* this is a one-way call: async, no return \*/
3. TF\_ROOT\_OBJECT  = 0x04, /\* contents are the component's root object \*/
4. TF\_STATUS\_CODE  = 0x08, /\* contents are a 32-bit status code \*/
5. TF\_ACCEPT\_FDS   = 0x10, /\* allow replies with file descriptors \*/
6. };

      每一个标志位所表示的意义看注释就行了，遇到时再具体分析。

      sender\_pid和sender\_euid表示发送者进程的pid和euid。

      data\_size表示data.buffer缓冲区的大小，offsets\_size表示data.offsets缓冲区的大小。这里需要解释一下data成员变量，命令的真正要传输的数据就保存在data.buffer缓冲区中，前面的一成员变量都是一些用来描述数据的特征的。data.buffer所表示的缓冲区数据分为两类，一类是普通数据，Binder驱动程序不关心，一类是Binder实体或者Binder引用，这需要Binder驱动程序介入处理。为什么呢？想想，如果一个进程A传递了一个Binder实体或Binder引用给进程B，那么，Binder驱动程序就需要介入维护这个Binder实体或者引用的引用计数，防止B进程还在使用这个Binder实体时，A却销毁这个实体，这样的话，B进程就会crash了。所以在传输数据时，如果数据中含有Binder实体和Binder引和，就需要告诉Binder驱动程序它们的具体位置，以便Binder驱动程序能够去维护它们。data.offsets的作用就在这里了，它指定在data.buffer缓冲区中，所有Binder实体或者引用的偏移位置。每一个Binder实体或者引用，通过struct flat\_binder\_object 来表示：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. /\*
2. \* This is the flattened representation of a Binder object for transfer
3. \* between processes.  The 'offsets' supplied as part of a binder transaction
4. \* contains offsets into the data where these structures occur.  The Binder
5. \* driver takes care of re-writing the structure type and data as it moves
6. \* between processes.
7. \*/
8. **struct** flat\_binder\_object {
9. /\* 8 bytes for large\_flat\_header. \*/
10. unsigned **long**       type;
11. unsigned **long**       flags;
13. /\* 8 bytes of data. \*/
14. **union** {
15. **void**        \*binder;    /\* local object \*/
16. **signed** **long** handle;     /\* remote object \*/
17. };
19. /\* extra data associated with local object \*/
20. **void**            \*cookie;
21. };

       type表示Binder对象的类型，它取值如下所示：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **enum** {
2. BINDER\_TYPE\_BINDER  = B\_PACK\_CHARS('s', 'b', '\*', B\_TYPE\_LARGE),
3. BINDER\_TYPE\_WEAK\_BINDER = B\_PACK\_CHARS('w', 'b', '\*', B\_TYPE\_LARGE),
4. BINDER\_TYPE\_HANDLE  = B\_PACK\_CHARS('s', 'h', '\*', B\_TYPE\_LARGE),
5. BINDER\_TYPE\_WEAK\_HANDLE = B\_PACK\_CHARS('w', 'h', '\*', B\_TYPE\_LARGE),
6. BINDER\_TYPE\_FD      = B\_PACK\_CHARS('f', 'd', '\*', B\_TYPE\_LARGE),
7. };

       flags表示Binder对象的标志，该域只对第一次传递Binder实体时有效，因为此刻驱动需要在内核中创建相应的实体节点，有些参数需要从该域取出。

       type和flags的具体意义可以参考[Android Binder设计与实现](http://disanji.net/2011/02/28/android-bnder-design/)一文。

       最后，binder表示这是一个Binder实体，handle表示这是一个Binder引用，当这是一个Binder实体时，cookie才有意义，表示附加数据，由进程自己解释。

       数据结构分析完了，回到binder\_loop函数中，首先是执行BC\_ENTER\_LOOPER命令：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. readbuf[0] = BC\_ENTER\_LOOPER;
2. binder\_write(bs, readbuf, **sizeof**(unsigned));

        进入到binder\_write函数中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **int** binder\_write(**struct** binder\_state \*bs, **void** \*data, unsigned len)
2. {
3. **struct** binder\_write\_read bwr;
4. **int** res;
5. bwr.write\_size = len;
6. bwr.write\_consumed = 0;
7. bwr.write\_buffer = (unsigned) data;
8. bwr.read\_size = 0;
9. bwr.read\_consumed = 0;
10. bwr.read\_buffer = 0;
11. res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);
12. **if** (res < 0) {
13. fprintf(stderr,"binder\_write: ioctl failed (%s)\n",
14. strerror(errno));
15. }
16. **return** res;
17. }

        注意这里的binder\_write\_read变量bwr，write\_size大小为4，表示write\_buffer缓冲区大小为4，它的内容是一个BC\_ENTER\_LOOPER命令协议号，read\_buffer为空。接着又是调用ioctl函数进入到Binder驱动程序的binder\_ioctl函数，这里我们也只是关注BC\_ENTER\_LOOPER相关的逻辑：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **long** binder\_ioctl(**struct** file \*filp, unsigned **int** cmd, unsigned **long** arg)
2. {
3. **int** ret;
4. **struct** binder\_proc \*proc = filp->private\_data;
5. **struct** binder\_thread \***thread**;
6. unsigned **int** size = \_IOC\_SIZE(cmd);
7. **void** \_\_user \*ubuf = (**void** \_\_user \*)arg;
9. /\*printk(KERN\_INFO "binder\_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);\*/
11. ret = wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);
12. **if** (ret)
13. **return** ret;
15. mutex\_lock(&binder\_lock);
16. **thread** = binder\_get\_thread(proc);
17. **if** (**thread** == NULL) {
18. ret = -ENOMEM;
19. **goto** err;
20. }
22. **switch** (cmd) {
23. **case** BINDER\_WRITE\_READ: {
24. **struct** binder\_write\_read bwr;
25. **if** (size != **sizeof**(**struct** binder\_write\_read)) {
26. ret = -EINVAL;
27. **goto** err;
28. }
29. **if** (copy\_from\_user(&bwr, ubuf, **sizeof**(bwr))) {
30. ret = -EFAULT;
31. **goto** err;
32. }
33. **if** (binder\_debug\_mask & BINDER\_DEBUG\_READ\_WRITE)
34. printk(KERN\_INFO "binder: %d:%d write %ld at %08lx, read %ld at %08lx\n",
35. proc->pid, **thread**->pid, bwr.write\_size, bwr.write\_buffer, bwr.read\_size, bwr.read\_buffer);
36. **if** (bwr.write\_size > 0) {
37. ret = binder\_thread\_write(proc, **thread**, (**void** \_\_user \*)bwr.write\_buffer, bwr.write\_size, &bwr.write\_consumed);
38. **if** (ret < 0) {
39. bwr.read\_consumed = 0;
40. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr)))
41. ret = -EFAULT;
42. **goto** err;
43. }
44. }
45. **if** (bwr.read\_size > 0) {
46. ret = binder\_thread\_read(proc, **thread**, (**void** \_\_user \*)bwr.read\_buffer, bwr.read\_size, &bwr.read\_consumed, filp->f\_flags & O\_NONBLOCK);
47. **if** (!list\_empty(&proc->todo))
48. wake\_up\_interruptible(&proc->wait);
49. **if** (ret < 0) {
50. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr)))
51. ret = -EFAULT;
52. **goto** err;
53. }
54. }
55. **if** (binder\_debug\_mask & BINDER\_DEBUG\_READ\_WRITE)
56. printk(KERN\_INFO "binder: %d:%d wrote %ld of %ld, read return %ld of %ld\n",
57. proc->pid, **thread**->pid, bwr.write\_consumed, bwr.write\_size, bwr.read\_consumed, bwr.read\_size);
58. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr))) {
59. ret = -EFAULT;
60. **goto** err;
61. }
62. **break**;
63. }
64. ......
65. **default**:
66. ret = -EINVAL;
67. **goto** err;
68. }
69. ret = 0;
70. err:
71. **if** (**thread**)
72. **thread**->looper &= ~BINDER\_LOOPER\_STATE\_NEED\_RETURN;
73. mutex\_unlock(&binder\_lock);
74. wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);
75. **if** (ret && ret != -ERESTARTSYS)
76. printk(KERN\_INFO "binder: %d:%d ioctl %x %lx returned %d\n", proc->pid, current->pid, cmd, arg, ret);
77. **return** ret;
78. }

       函数前面的代码就不解释了，同前面调用binder\_become\_context\_manager是一样的，只不过这里调用binder\_get\_thread函数获取binder\_thread，就能从proc中直接找到了，不需要创建一个新的。

       首先是通过copy\_from\_user(&bwr, ubuf, sizeof(bwr))语句把用户传递进来的参数转换成struct binder\_write\_read结构体，并保存在本地变量bwr中，这里可以看出bwr.write\_size等于4，于是进入binder\_thread\_write函数，这里我们只关注BC\_ENTER\_LOOPER相关的代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **int**
2. binder\_thread\_write(**struct** binder\_proc \*proc, **struct** binder\_thread \***thread**,
3. **void** \_\_user \*buffer, **int** size, **signed** **long** \*consumed)
4. {
5. uint32\_t cmd;
6. **void** \_\_user \*ptr = buffer + \*consumed;
7. **void** \_\_user \*end = buffer + size;
9. **while** (ptr < end && **thread**->return\_error == BR\_OK) {
10. **if** (get\_user(cmd, (uint32\_t \_\_user \*)ptr))
11. **return** -EFAULT;
12. ptr += **sizeof**(uint32\_t);
13. **if** (\_IOC\_NR(cmd) < ARRAY\_SIZE(binder\_stats.bc)) {
14. binder\_stats.bc[\_IOC\_NR(cmd)]++;
15. proc->stats.bc[\_IOC\_NR(cmd)]++;
16. **thread**->stats.bc[\_IOC\_NR(cmd)]++;
17. }
18. **switch** (cmd) {
19. ......
20. **case** BC\_ENTER\_LOOPER:
21. **if** (binder\_debug\_mask & BINDER\_DEBUG\_THREADS)
22. printk(KERN\_INFO "binder: %d:%d BC\_ENTER\_LOOPER\n",
23. proc->pid, **thread**->pid);
24. **if** (**thread**->looper & BINDER\_LOOPER\_STATE\_REGISTERED) {
25. **thread**->looper |= BINDER\_LOOPER\_STATE\_INVALID;
26. binder\_user\_error("binder: %d:%d ERROR:"
27. " BC\_ENTER\_LOOPER called after "
28. "BC\_REGISTER\_LOOPER\n",
29. proc->pid, **thread**->pid);
30. }
31. **thread**->looper |= BINDER\_LOOPER\_STATE\_ENTERED;
32. **break**;
33. ......
34. **default**:
35. printk(KERN\_ERR "binder: %d:%d unknown command %d\n", proc->pid, **thread**->pid, cmd);
36. **return** -EINVAL;
37. }
38. \*consumed = ptr - buffer;
39. }
40. **return** 0;
41. }

       回忆前面执行binder\_become\_context\_manager到binder\_ioctl时，调用binder\_get\_thread函数创建的thread->looper值为0，所以这里执行完BC\_ENTER\_LOOPER时，thread->looper值就变为BINDER\_LOOPER\_STATE\_ENTERED了，表明当前线程进入循环状态了。

       回到binder\_ioctl函数，由于bwr.read\_size == 0，binder\_thread\_read函数就不会被执行了，这样，binder\_ioctl的任务就完成了。

       回到binder\_loop函数，进入for循环：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **for** (;;) {
2. bwr.read\_size = **sizeof**(readbuf);
3. bwr.read\_consumed = 0;
4. bwr.read\_buffer = (unsigned) readbuf;
6. res = ioctl(bs->fd, BINDER\_WRITE\_READ, &bwr);
8. **if** (res < 0) {
9. LOGE("binder\_loop: ioctl failed (%s)\n", strerror(errno));
10. **break**;
11. }
13. res = binder\_parse(bs, 0, readbuf, bwr.read\_consumed, func);
14. **if** (res == 0) {
15. LOGE("binder\_loop: unexpected reply?!\n");
16. **break**;
17. }
18. **if** (res < 0) {
19. LOGE("binder\_loop: io error %d %s\n", res, strerror(errno));
20. **break**;
21. }
22. }

        又是执行一个ioctl命令，注意，这里的bwr参数各个成员的值：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. bwr.write\_size = 0;
2. bwr.write\_consumed = 0;
3. bwr.write\_buffer = 0;
4. readbuf[0] = BC\_ENTER\_LOOPER;
5. bwr.read\_size = **sizeof**(readbuf);
6. bwr.read\_consumed = 0;
7. bwr.read\_buffer = (unsigned) readbuf;

        再次进入到binder\_ioctl函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **long** binder\_ioctl(**struct** file \*filp, unsigned **int** cmd, unsigned **long** arg)
2. {
3. **int** ret;
4. **struct** binder\_proc \*proc = filp->private\_data;
5. **struct** binder\_thread \***thread**;
6. unsigned **int** size = \_IOC\_SIZE(cmd);
7. **void** \_\_user \*ubuf = (**void** \_\_user \*)arg;
9. /\*printk(KERN\_INFO "binder\_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);\*/
11. ret = wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);
12. **if** (ret)
13. **return** ret;
15. mutex\_lock(&binder\_lock);
16. **thread** = binder\_get\_thread(proc);
17. **if** (**thread** == NULL) {
18. ret = -ENOMEM;
19. **goto** err;
20. }
22. **switch** (cmd) {
23. **case** BINDER\_WRITE\_READ: {
24. **struct** binder\_write\_read bwr;
25. **if** (size != **sizeof**(**struct** binder\_write\_read)) {
26. ret = -EINVAL;
27. **goto** err;
28. }
29. **if** (copy\_from\_user(&bwr, ubuf, **sizeof**(bwr))) {
30. ret = -EFAULT;
31. **goto** err;
32. }
33. **if** (binder\_debug\_mask & BINDER\_DEBUG\_READ\_WRITE)
34. printk(KERN\_INFO "binder: %d:%d write %ld at %08lx, read %ld at %08lx\n",
35. proc->pid, **thread**->pid, bwr.write\_size, bwr.write\_buffer, bwr.read\_size, bwr.read\_buffer);
36. **if** (bwr.write\_size > 0) {
37. ret = binder\_thread\_write(proc, **thread**, (**void** \_\_user \*)bwr.write\_buffer, bwr.write\_size, &bwr.write\_consumed);
38. **if** (ret < 0) {
39. bwr.read\_consumed = 0;
40. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr)))
41. ret = -EFAULT;
42. **goto** err;
43. }
44. }
45. **if** (bwr.read\_size > 0) {
46. ret = binder\_thread\_read(proc, **thread**, (**void** \_\_user \*)bwr.read\_buffer, bwr.read\_size, &bwr.read\_consumed, filp->f\_flags & O\_NONBLOCK);
47. **if** (!list\_empty(&proc->todo))
48. wake\_up\_interruptible(&proc->wait);
49. **if** (ret < 0) {
50. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr)))
51. ret = -EFAULT;
52. **goto** err;
53. }
54. }
55. **if** (binder\_debug\_mask & BINDER\_DEBUG\_READ\_WRITE)
56. printk(KERN\_INFO "binder: %d:%d wrote %ld of %ld, read return %ld of %ld\n",
57. proc->pid, **thread**->pid, bwr.write\_consumed, bwr.write\_size, bwr.read\_consumed, bwr.read\_size);
58. **if** (copy\_to\_user(ubuf, &bwr, **sizeof**(bwr))) {
59. ret = -EFAULT;
60. **goto** err;
61. }
62. **break**;
63. }
64. ......
65. **default**:
66. ret = -EINVAL;
67. **goto** err;
68. }
69. ret = 0;
70. err:
71. **if** (**thread**)
72. **thread**->looper &= ~BINDER\_LOOPER\_STATE\_NEED\_RETURN;
73. mutex\_unlock(&binder\_lock);
74. wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);
75. **if** (ret && ret != -ERESTARTSYS)
76. printk(KERN\_INFO "binder: %d:%d ioctl %x %lx returned %d\n", proc->pid, current->pid, cmd, arg, ret);
77. **return** ret;
78. }

         这次，bwr.write\_size等于0，于是不会执行binder\_thread\_write函数，bwr.read\_size等于32，于是进入到binder\_thread\_read函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6621566)

1. **static** **int**
2. binder\_thread\_read(**struct** binder\_proc \*proc, **struct** binder\_thread \***thread**,
3. **void**  \_\_user \*buffer, **int** size, **signed** **long** \*consumed, **int** non\_block)
4. {
5. **void** \_\_user \*ptr = buffer + \*consumed;
6. **void** \_\_user \*end = buffer + size;
8. **int** ret = 0;
9. **int** wait\_for\_proc\_work;
11. **if** (\*consumed == 0) {
12. **if** (put\_user(BR\_NOOP, (uint32\_t \_\_user \*)ptr))
13. **return** -EFAULT;
14. ptr += **sizeof**(uint32\_t);
15. }
17. retry:
18. wait\_for\_proc\_work = **thread**->transaction\_stack == NULL && list\_empty(&**thread**->todo);
20. **if** (**thread**->return\_error != BR\_OK && ptr < end) {
21. **if** (**thread**->return\_error2 != BR\_OK) {
22. **if** (put\_user(**thread**->return\_error2, (uint32\_t \_\_user \*)ptr))
23. **return** -EFAULT;
24. ptr += **sizeof**(uint32\_t);
25. **if** (ptr == end)
26. **goto** done;
27. **thread**->return\_error2 = BR\_OK;
28. }
29. **if** (put\_user(**thread**->return\_error, (uint32\_t \_\_user \*)ptr))
30. **return** -EFAULT;
31. ptr += **sizeof**(uint32\_t);
32. **thread**->return\_error = BR\_OK;
33. **goto** done;
34. }

37. **thread**->looper |= BINDER\_LOOPER\_STATE\_WAITING;
38. **if** (wait\_for\_proc\_work)
39. proc->ready\_threads++;
40. mutex\_unlock(&binder\_lock);
41. **if** (wait\_for\_proc\_work) {
42. **if** (!(**thread**->looper & (BINDER\_LOOPER\_STATE\_REGISTERED |
43. BINDER\_LOOPER\_STATE\_ENTERED))) {
44. binder\_user\_error("binder: %d:%d ERROR: Thread waiting "
45. "for process work before calling BC\_REGISTER\_"
46. "LOOPER or BC\_ENTER\_LOOPER (state %x)\n",
47. proc->pid, **thread**->pid, **thread**->looper);
48. wait\_event\_interruptible(binder\_user\_error\_wait, binder\_stop\_on\_user\_error < 2);
49. }
50. binder\_set\_nice(proc->default\_priority);
51. **if** (non\_block) {
52. **if** (!binder\_has\_proc\_work(proc, **thread**))
53. ret = -EAGAIN;
54. } **else**
55. ret = wait\_event\_interruptible\_exclusive(proc->wait, binder\_has\_proc\_work(proc, **thread**));
56. } **else** {
57. **if** (non\_block) {
58. **if** (!binder\_has\_thread\_work(**thread**))
59. ret = -EAGAIN;
60. } **else**
61. ret = wait\_event\_interruptible(**thread**->wait, binder\_has\_thread\_work(**thread**));
62. }
63. .......
64. }

         传入的参数\*consumed == 0，于是写入一个值BR\_NOOP到参数ptr指向的缓冲区中去，即用户传进来的bwr.read\_buffer缓冲区。这时候，thread->transaction\_stack == NULL，并且thread->todo列表也是空的，这表示当前线程没有事务需要处理，于是wait\_for\_proc\_work为true，表示要去查看proc是否有未处理的事务。当前thread->return\_error == BR\_OK，这是前面创建binder\_thread时初始化设置的。于是继续往下执行，设置thread的状态为BINDER\_LOOPER\_STATE\_WAITING，表示线程处于等待状态。调用binder\_set\_nice函数设置当前线程的优先级别为proc->default\_priority，这是因为thread要去处理属于proc的事务，因此要将此thread的优先级别设置和proc一样。在这个场景中，proc也没有事务处理，即binder\_has\_proc\_work(proc, thread)为false。如果文件打开模式为非阻塞模式，即non\_block为true，那么函数就直接返回-EAGAIN，要求用户重新执行ioctl；否则的话，就通过当前线程就通过wait\_event\_interruptible\_exclusive函数进入休眠状态，等待请求到来再唤醒了。

        至此，我们就从源代码一步一步地分析完Service Manager是如何成为Android进程间通信（IPC）机制Binder守护进程的了。总结一下，Service Manager是成为Android进程间通信（IPC）机制Binder守护进程的过程是这样的：

        1. 打开/dev/binder文件：open("/dev/binder", O\_RDWR);

        2. 建立128K内存映射：mmap(NULL, mapsize, PROT\_READ, MAP\_PRIVATE, bs->fd, 0);

        3. 通知Binder驱动程序它是守护进程：binder\_become\_context\_manager(bs);

        4. 进入循环等待请求的到来：binder\_loop(bs, svcmgr\_handler);

        在这个过程中，在Binder驱动程序中建立了一个struct binder\_proc结构、一个struct  binder\_thread结构和一个struct binder\_node结构，这样，Service Manager就在Android系统的进程间通信机制Binder担负起守护进程的职责了。