

文件系统 (16)

u盘挂载 (8)

Linux内存管理 (0)

JAVA (3)

Android 技巧 (2)

Python (50)

Gradle (22)

RxJava (46)

Okhttp&Retrofit (12)

文章存档

2017年08月 (1)

2017年07月 (1)

2017年05月 (1)

2017年04月 (1)

2017年02月 (2)

展开

阅读排行

使用ObjectAnimator设置PackageManagerService (43305)

RxJava 中的map与flatMap (30701)

eclipse调试出现Failed to start the VM (25221)

android usb挂载分析---Ming (23929)

nginx地址重定向 (17296)

使用ValueAnimator设置TextView (16799)

adb install 流程 (16043)

python基础学习笔记<Week 1> (15523)

SensorService启动分析 (13817)

android usb挂载分析---Ming (13465)

评论排行

PackageManagerService (19)

linux设备驱动模型一三基 (16)

kernel升级 (15)

Android系统修改导航栏 (12)

adb 简要分析 (12)

Android dagger2使用 (9)

SensorService启动分析 (9)

android:allowTaskReparenting (8)

TS解析 (8)

linux i2c驱动 (8)

推荐文章

* CSDN邀请您来GitChat赚钱啦!

* 行为驱动开发（BDD）你准备好了吗?

* 如何更加安全、高效地利用开源项目?

* 程序员业余时间修炼指南

* DevOps 在公司项目中的实践落地

* Jenkins + Django 完整实战，细化到每一步操作

最新评论

Android dagger2使用

```
设备
31. #endif
32. #ifndef O_LARGEFILE
33. #define O_LARGEFILE 00100000 // 大文件标识
34. #endif
35. #ifndef O_DIRECTORY
36. #define O_DIRECTORY 00200000 // 必须是目录
37. #endif
38. #ifndef O_NOFOLLOW
39. #define O_NOFOLLOW 00400000 // 不获取连接文件
40. #endif
41. #ifndef O_NOATIME
42. #define O_NOATIME 01000000
43. #endif
44. #ifndef O_CLOEXEC
45. #define O_CLOEXEC 02000000 /* set close_on_exec */
46. #endif
47. #ifndef O_NDELAY
48. #define O_NDELAY O_NONBLOCK
49. #endif
```

当新创建一个文件时，需要指定mode参数，以下说明的格式如宏定义名称<实际常数>: 描述如下(include/linux/stat.h):

```
[cpp] view plain copy print ?
01. #define S_IRWXU 00700 //文件所有者有读写执行权限
02. #define S_IRUSR 00400 //文件所有者仅有读权限
03. #define S_IWUSR 00200 //文件所有者仅有写权限
04. #define S_IXUSR 00100 //文件所有者仅有执行权限
05.
06. #define S_IRWXG 00070 //组用户有读写执行权限
07. #define S_IRGRP 00040 //组用户仅有读权限
08. #define S_IWGRP 00020 //组用户仅有写权限
09. #define S_IXGRP 00010 //组用户仅有执行权限
10.
11. #define S_IRWXO 00007 //其他用户有读写执行权限
12. #define S_IROTH 00004 //其他用户仅有读权限
13. #define S_IWOTH 00002 //其他用户仅有写权限
14. #define S_IXOTH 00001 //其他用户仅有执行权限
```

系统调用号定义在arch/x86/include/asm/unistd_32.h中:

```
[cpp] view plain copy print ?
01. #define __NR_restart_syscall 0
02. #define __NR_exit 1
03. #define __NR_fork 2
04. #define __NR_read 3
05. #define __NR_write 4
06. #define __NR_open 5
07. #define __NR_close 6
08. #define __NR_waitpid 7
09. #define __NR_creat 8
10. #define __NR_link 9
11. #define __NR_unlink 10
12. #define __NR_execve 11
13. #define __NR_chdir 12
14. #define __NR_time 13
15. #define __NR_mknod 14
16. #define __NR_chmod 15
```

当open系统调用产生时，就会进入下面这个函数（）:

```
[cpp] view plain copy print ?
01. SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode)
02. {
03.     long ret;
04.
05.     /*检查是否应该不考虑用户层传递的标志、总是强行设置
06.     O_LARGEFILE标志。如果底层处理器的字长不是32位，就是这种
07.     情况*/
08.     if (force_o_largefile())
09.         flags |= O_LARGEFILE;
10.     /*实际工作*/
11.     ret = do_sys_open(AT_FDCWD, filename, flags, mode);
```

盛开的伤: 11

android usb挂载分析--ext4支持u
半半天涯_飞: 你好！我的一个硬
盘是ext4格式的文件系统，我在
Android系统中用文件管理等创
建文件夹等得到的...

RxJava 中的map与flatMap
_Chic: 真的不错。学习了。看
了几遍终于略懂12了

Android的优化
dfliang: 那些工具怎么下载？

Rxjava(结合类)-Zip
790870749:

Rxjava(结合类)-Zip
790870749:

Rxjava(结合类)-Zip
790870749: 还能输入1000个字
符

NanoHTTPD----SimpleWebSer
qq_38140615: 好东西！

android/support/v4/text/TextUtils
xzfengxi: 厉害了，我的哥 知道
为啥么

Amigo源码分析
我家果果: 楼主好，
AmigoPlugin.groovy这个方式能
改java代码方式去实现吗，入口
应该怎么做。

```
12. |         /* avoid REGPARM breakage on x86: */
13. |         asmlinkage_protect(3, ret, filename, flags, mode);
14. |         return ret;
15. |     }
```

我们看下SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode) 展开是怎么样的
首先看下宏SYSCALL_DEFINE3

```
[cpp] view plain copy print ?
01. | #define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)
```

再往下看SYSCALL_DEFINEx

```
[cpp] view plain copy print ?
01. | #define SYSCALL_DEFINEx(x, sname, ...) \
02. |     __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

再往下看__SYSCALL_DEFINEx

```
[cpp] view plain copy print ?
01. | #define __SYSCALL_DEFINEx(x, name, ...) \
02. |     asmlinkage long sys##name(__SC_DECL##x(__VA_ARGS__))
```

这里对对应__SC_DECL3

```
[cpp] view plain copy print ?
01. | #define __SC_DECL1(t1, a1) t1 a1
02. | #define __SC_DECL2(t2, a2, ...) t2 a2, __SC_DECL1(__VA_ARGS__)
03. | #define __SC_DECL3(t3, a3, ...) t3 a3, __SC_DECL2(__VA_ARGS__)
```

他们一步步展开SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode)代替进去，可以得到

```
[cpp] view plain copy print ?
01. | SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode)
02. | = SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)
03. | =asmlinkage long sys_open(__SC_DECL3(__VA_ARGS__))
04. | =asmlinkage long sys_open(const char __user* filename, int flags, int mode)
```

这个才是真正的函数原型

在sys_open里面继续调用do_sys_open完成 open操作

```
[cpp] view plain copy print ?
01. | long do_sys_open(int dfd, const char __user *filename, int flags, int mode)
02. | {
03. |     /*从进程地址空间读取该文件的路径名*/
04. |     char *tmp = getname(filename);
05. |     int fd = PTR_ERR(tmp);
06. |
07. |     if (!IS_ERR(tmp)) {
08. |         /*在内核中，每个打开的文件由一个文件描述符表示
09. |         该描述符在特定于进程的数组中充当位置索引(数组是
10. |         task_struct->files->fd_array)，该数组的元素包含了file结构，其中
11. |         包括每个打开文件的所有必要信息。因此，调用下面
12. |         函数查找一个未使用的文件描述符,返回的是上面
13. |         说的数组的下标*/
14. |         fd = get_unused_fd_flags(flags);
15. |         if (fd >= 0) {
16. |             /*fd获取成功则开始打开文件，此函数是主要完成打开功能的函数*/
17. |             //如果分配fd成功，则创建一个file对象
18. |             struct file *f = do_filp_open(dfd, tmp, flags, mode, 0);
19. |             if (IS_ERR(f)) {
20. |                 put_unused_fd(fd);
21. |                 fd = PTR_ERR(f);
22. |             } else {
23. |                 /*文件如果打开成功，调用fsnotify_open()函数，根据inode所指定的信息进行打开
24. |                 函数(参数为f)将该文件加入到文件监控的系统中。该系统是用来监控文件被打开，创建，
25. |                 读写，关闭，修改等操作的*/
26. |                 fsnotify_open(f->f_path.dentry);
27. |                 /*将文件指针安装在fd数组中
```

```

28.         将struct file *f加入到fd索引位置处的数组中。如果后续过程中，有对该文件描述符的
29.         操作的话，就会通过查找该数组得到对应的文件结构，而后在进行相关操作。*/
30.         fd_install(fd, f);
31.     }
32. }
33. putname(tmp);
34. }
35. return fd;
36. }

```

该函数主要分为如下几个步骤来完成打开文件的操作：

- 1.将文件名参数从用户态拷贝至内核，调用函数get_name();
- 2.从进程的文件表中找到一个空闲的文件表指针，调用了函数get_unused_fd_flags();
- 3.完成真正的打开操作，调用函数do_filp_open();
- 4.将打开的文件添加到进程的文件表数组中，调用函数fd_install();

getname函数主要的任务是将文件名filename从用户态拷贝至内核态

```

[cpp] view plain copy print ?
01. char * getname(const char __user * filename)
02. {
03.     char *tmp, *result;
04.
05.     result = ERR_PTR(-ENOMEM);
06.     tmp = __getname(); //从内核缓存中分配空间;
07.     if (tmp) {
08.         //将文件名从用户态拷贝至内核态;
09.         int retval = do_getname(filename, tmp);
10.
11.         result = tmp;
12.         if (retval < 0) { //如果拷贝失败，则调用__putname()释放__getname()中申请的空间;
13.             __putname(tmp);
14.             result = ERR_PTR(retval);
15.         }
16.     }
17.     audit_getname(result);
18.     return result;
19. }

```

get_unused_fd_flags实际调用的是alloc_fd

```

[cpp] view plain copy print ?
01. #define get_unused_fd_flags(flags) alloc_fd(0, (flags))

[cpp] view plain copy print ?
01. /*
02.  * allocate a file descriptor, mark it busy.
03.  */
04. int alloc_fd(unsigned start, unsigned flags)
05. {
06.     struct files_struct *files = current->files; //获得当前进程的files_struct 结构
07.     unsigned int fd;
08.     int error;
09.     struct fdtable *fdt;
10.
11.     spin_lock(&files->file_lock);
12. repeat:
13.     fdt = files_fdtable(files);
14.     fd = start;
15.     if (fd < files->next_fd) //从上一次打开的fd的下一个fd开始搜索空闲的fd
16.         fd = files->next_fd;
17.
18.     if (fd < fdt->max_fds) //寻找空闲的fd，返回值为空闲的fd
19.         fd = find_next_zero_bit(fdt->open_fds->fds_bits,
20.                                fdt->max_fds, fd);
21.     //如果有必要，即打开的fd超过max_fds,则需要expand当前进程的fd表;
22.     //返回值error<0表示出错，error=0表示无需expand，error=1表示进行了expand;
23.     error = expand_files(files, fd);
24.     if (error < 0)
25.         goto out;
26.
27.     /*

```

```

28.         * If we needed to expand the fs array we
29.         * might have blocked - try again.
30.         */
31.         //error=1表示进行了expand，那么此时需要重新去查找空闲的fd;
32.         if (error)
33.             goto repeat;
34.
35.         //设置下一次查找的起始fd，即本次找到的空闲的fd的下一个fd，记录在files->next_fd中;
36.         if (start <= files->next_fd)
37.             files->next_fd = fd + 1;
38.
39.         FD_SET(fd, fdt->open_fds);
40.         if (flags & O_CLOEXEC)
41.             FD_SET(fd, fdt->close_on_exec);
42.         else
43.             FD_CLR(fd, fdt->close_on_exec);
44.         error = fd;
45. #if 1
46.         /* Sanity check */
47.         if (rcu_dereference(fdt->fd[fd]) != NULL) {
48.             printk(KERN_WARNING "alloc_fd: slot %d not NULL!\n", fd);
49.             rcu_assign_pointer(fdt->fd[fd], NULL);
50.         }
51. #endif
52.
53. out:
54.     spin_unlock(&files->file_lock);
55.     return error;
56. }

```

该函数为需要打开的文件在当前进程内分配一个空闲的文件描述符fd，该fd就是open()系统调用的返回值do_filp_open函数的一个重要作用就是根据传递近来的权限进行分析，并且分析传递近来的路径名字，根据路径名逐个解析成dentry，并且通过dentry找到inode，inode就是记录着该文件相关的信息，包括文件的创建时间和文件属性所有者等等信息，根据这些信息就可以找到对应的文件操作方法。在这个过程当中有一个临时的结构体用于保存在查找过程中的相关信息，就是

[cpp] view plain copy print ?

```

01. struct nameidata {
02.     struct path path; //当前目录的dentry数据结构
03.     struct qstr last; //这个结构体也是临时性的，主要用来保存当前目录的名称，杂凑值。
04.     unsigned int flags;
05.     int last_type;
06.     unsigned depth; //连接文件的深度（可能一个连接文件跟到最后还是一个了连接文件）
07.     //用来保存连接文件的一些信息，下标表示连接文件的深度
08.     char *saved_names[MAX_NESTED_LINKS + 1];
09.
10.     /* Intent data */
11.     union {
12.         struct open_intent open;
13.     } intent;
14. };

```

[cpp] view plain copy print ?

```

01. struct file *do_filp_open(int dfd, const char *pathname,
02.     int open_flag, int mode, int acc_mode)
03. {
04.     struct file *filp;
05.     struct nameidata nd;
06.     int error;
07.     struct path path;
08.     int count = 0;
09.     int flag = open_to_namei_flags(open_flag); /*改变参数flag的值，具体做法是flag+1*/
10.     int force_reval = 0;
11.
12.     if (!(open_flag & O_CREAT))
13.         mode = 0;
14.
15.     /*
16.      * O_SYNC is implemented as __O_SYNC|O_DSYNC. As many places only
17.      * check for O_DSYNC if the need any syncing at all we enforce it's
18.      * always set instead of having to deal with possibly weird behaviour
19.      * for malicious applications setting only __O_SYNC.
20.      */

```

```

21.         if (open_flag & __O_SYNC)/*根据__O_SYNC标志来设置O_DSYNC 标志，用以防止恶意破坏程序*/
22.             open_flag |= O_DSYNC;
23.
24.         if (!acc_mode)/*设置访问权限*/
25.             acc_mode = MAY_OPEN | ACC_MODE(open_flag);
26.
27.         /* O_TRUNC implies we need access checks for write permissions */
28.         if (open_flag & O_TRUNC)/*根据 O_TRUNC标志设置写权限 */
29.             acc_mode |= MAY_WRITE;
30.
31.         /* Allow the LSM permission hook to distinguish append
32.            access from general write access. */
33.         if (open_flag & O_APPEND)/* 设置O_APPEND 标志*/
34.             acc_mode |= MAY_APPEND;
35.
36.         /* find the parent */
37.     reval:
38.         error = path_init(dfd, pathname, LOOKUP_PARENT, &nd);//初始化nd
39.         if (error)
40.             return ERR_PTR(error);
41.         if (force_reval)
42.             nd.flags |= LOOKUP_REVAL;
43.
44.         current->total_link_count = 0;
45.         error = link_path_walk(pathname, &nd);//路径名解析函数，将一个路径名最终转化为一个dentry
46.         if (error) {
47.             filp = ERR_PTR(error);
48.             goto out;
49.         }
50.         if (unlikely(!audit_dummy_context()) && (open_flag & O_CREAT))
51.             audit_inode(pathname, nd.path.dentry);
52.
53.         /*
54.          * We have the parent and last component.
55.          */
56.
57.         error = -ENFILE;
58.         filp = get_empty_filp();// 从进程文件表中获取一个未使用的文件结构指针，空则出错返回
59.         if (filp == NULL)
60.             goto exit_parent;
61.         nd.intent.open.file = filp;
62.         filp->f_flags = open_flag;
63.         nd.intent.open.flags = flag;
64.         nd.intent.open.create_mode = mode;
65.         nd.flags &= ~LOOKUP_PARENT;
66.         nd.flags |= LOOKUP_OPEN;
67.         if (open_flag & O_CREAT) {
68.             nd.flags |= LOOKUP_CREATE;
69.             if (open_flag & O_EXCL)
70.                 nd.flags |= LOOKUP_EXCL;
71.         }
72.         if (open_flag & O_DIRECTORY)
73.             nd.flags |= LOOKUP_DIRECTORY;
74.         if (!(open_flag & O_NOFOLLOW))
75.             nd.flags |= LOOKUP_FOLLOW;
76.         filp = do_last(&nd, &path, open_flag, acc_mode, mode, pathname);//返回一个file结构
77.         while (unlikely(!filp)) { /* trailing symlink *///符号链接
78.             struct path holder;
79.             struct inode *inode = path.dentry->d_inode;
80.             void *cookie;
81.             error = -ELOOP;
82.             /* S_ISDIR part is a temporary automount kludge */
83.             if (!(nd.flags & LOOKUP_FOLLOW) && !S_ISDIR(inode->i_mode))
84.                 goto exit_dput;
85.             if (count++ == 32)
86.                 goto exit_dput;
87.             /*
88.              * This is subtle. Instead of calling do_follow_link() we do
89.              * the thing by hands. The reason is that this way we have zero
90.              * link_count and path_walk() (called from ->follow_link)
91.              * honoring LOOKUP_PARENT. After that we have the parent and
92.              * last component, i.e. we are in the same situation as after
93.              * the first path_walk(). Well, almost - if the last component
94.              * is normal we get its copy stored in nd->last.name and we will
95.              * have to putname() it when we are done. Procfs-like symlinks
96.              * just set LAST_BIND.
97.              */
98.             nd.flags |= LOOKUP_PARENT;
99.             error = security_inode_follow_link(path.dentry, &nd);

```

```

100.         if (error)
101.             goto exit_dput;
102.         error = __do_follow_link(&path, &nd, &cookie); // 查找符号链接对应的目录中的最后一项
103.         if (unlikely(error)) {
104.             /* nd.path had been dropped */
105.             if (!IS_ERR(cookie) && inode->i_op->put_link)
106.                 inode->i_op->put_link(path.dentry, &nd, cookie);
107.             path_put(&path);
108.             release_open_intent(&nd);
109.             filp = ERR_PTR(error);
110.             goto out;
111.         }
112.         holder = path;
113.         nd.flags &= ~LOOKUP_PARENT;
114.         filp = do_last(&nd, &path, open_flag, acc_mode, mode, pathname);
115.         if (inode->i_op->put_link)
116.             inode->i_op->put_link(holder.dentry, &nd, cookie);
117.         path_put(&holder);
118.     }
119. out:
120.     if (nd.root.mnt)
121.         path_put(&nd.root);
122.     if (filp == ERR_PTR(-ESTALE) && !force_reval) {
123.         force_reval = 1;
124.         goto reval;
125.     }
126.     return filp; // 成功, 返回
127.
128. exit_dput:
129.     path_put_conditional(&path, &nd);
130.     if (!IS_ERR(nd.intent.open.file))
131.         release_open_intent(&nd);
132. exit_parent:
133.     path_put(&nd.path);
134.     filp = ERR_PTR(error);
135.     goto out;
136. }

```

当内核要访问一个文件的时候, 第一步要做的是找到这个文件, 而查找文件的过程在vfs里面是由link_path_walk函数来完成的, 在path_init的时候我们可以看到传进去的参数有一个LOOKUP_PARENT, 它的含义是查找最后一个分量名所在的目录。也就是当这个函数返回的时候, 我们得到了一个路径名中最后一个分量所在的目录。接着调用do_last返回最后一个分量对应的file指针, 我们关注一下这个函数

```

[cpp] view plain copy print ?

01. static struct file *do_last(struct nameidata *nd, struct path *path,
02.                             int open_flag, int acc_mode,
03.                             int mode, const char *pathname)
04. {
05.     struct dentry *dir = nd->path.dentry;
06.     struct file *filp;
07.     int error = -EISDIR;
08.
09.     switch (nd->last_type) { // 检查最后一段文件或目录名的属性情况
10.     case LAST_DOTDOT:
11.         follow_dotdot(nd);
12.         dir = nd->path.dentry;
13.     case LAST_DOT:
14.         if (nd->path.mnt->mnt_sb->s_type->fs_flags & FS_REVAL_DOT) {
15.             if (!dir->d_op->d_revalidate(dir, nd)) {
16.                 error = -ESTALE;
17.                 goto exit;
18.             }
19.         }
20.         /* fallthrough */
21.     case LAST_ROOT:
22.         if (open_flag & O_CREAT)
23.             goto exit;
24.         /* fallthrough */
25.     case LAST_BIND:
26.         audit_inode(pathname, dir);
27.         goto ok;
28.     }
29.
30.     /* trailing slashes? */
31.     if (nd->last.name[nd->last.len]) {

```

```

32.         if (open_flag & O_CREAT)
33.             goto exit;
34.         nd->flags |= LOOKUP_DIRECTORY | LOOKUP_FOLLOW;
35.     }
36.
37. /* just plain open? */
38. if (!(open_flag & O_CREAT)) { //没有创建标志, 即文件存在
39.     error = do_lookup(nd, &nd->last, path); //找到路径中最后一项对应的目录项
40.     if (error)
41.         goto exit;
42.     error = -ENOENT;
43.     if (!path->dentry->d_inode)
44.         goto exit_dput;
45.     if (path->dentry->d_inode->i_op->follow_link)
46.         return NULL;
47.     error = -ENOTDIR;
48.     if (nd->flags & LOOKUP_DIRECTORY) {
49.         if (!path->dentry->d_inode->i_op->lookup)
50.             goto exit_dput;
51.     }
52.     path_to_nameidata(path, nd); //赋值到nd结构
53.     audit_inode(pathname, nd->path.dentry);
54.     goto ok;
55. }
56.
57. /* OK, it's O_CREAT */
58. //文件不存在, 需要创建
59. mutex_lock(&dir->d_inode->i_mutex);
60.
61. path->dentry = lookup_hash(nd); //获取最后路径名中最后一项对应的目录项
62. path->mnt = nd->path.mnt;
63.
64. error = PTR_ERR(path->dentry);
65. if (IS_ERR(path->dentry)) {
66.     mutex_unlock(&dir->d_inode->i_mutex);
67.     goto exit;
68. }
69.
70. if (IS_ERR(nd->intent.open.file)) {
71.     error = PTR_ERR(nd->intent.open.file);
72.     goto exit_mutex_unlock;
73. }
74.
75. /* Negative dentry, just create the file */
76. if (!path->dentry->d_inode) { //没有索引节点与目录项关联
77.     /*
78.      * This write is needed to ensure that a
79.      * ro->rw transition does not occur between
80.      * the time when the file is created and when
81.      * a permanent write count is taken through
82.      * the 'struct file' in nameidata_to_filp().
83.      */
84.     error = mnt_want_write(nd->path.mnt);
85.     if (error)
86.         goto exit_mutex_unlock;
87.     error = __open_namei_create(nd, path, open_flag, mode); //创建相应的索引节点
88.     if (error) {
89.         mnt_drop_write(nd->path.mnt);
90.         goto exit;
91.     }
92.     filp = nameidata_to_filp(nd); //根据nameidata 得到相应的file结构*/
93.     mnt_drop_write(nd->path.mnt);
94.     if (!IS_ERR(filp)) {
95.         error = ima_file_check(filp, acc_mode);
96.         if (error) {
97.             fput(filp);
98.             filp = ERR_PTR(error);
99.         }
100.    }
101.    return filp;
102. }
103.
104. /*
105.  * It already exists.
106.  */
107. mutex_unlock(&dir->d_inode->i_mutex);
108. audit_inode(pathname, path->dentry);
109.
110. error = -EEXIST;

```



```

111.         if (open_flag & O_EXCL)
112.             goto exit_dput;
113.
114.         if (__follow_mount(path)) {
115.             error = -ELOOP;
116.             if (open_flag & O_NOFOLLOW)
117.                 goto exit_dput;
118.         }
119.
120.         error = -ENOENT;
121.         if (!path->dentry->d_inode)
122.             goto exit_dput;
123.
124.         if (path->dentry->d_inode->i_op->follow_link)
125.             return NULL;
126.
127.         path_to_nameidata(path, nd);
128.         error = -EISDIR;
129.         if (S_ISDIR(path->dentry->d_inode->i_mode))
130.             goto exit;
131.     ok:
132.         filp = finish_open(nd, open_flag, acc_mode); //完成文件打开操作
133.         return filp;
134.
135.     exit_mutex_unlock:
136.         mutex_unlock(&dir->d_inode->i_mutex);
137.     exit_dput:
138.         path_put_conditional(path, nd);
139.     exit:
140.         if (!IS_ERR(nd->intent.open.file))
141.             release_open_intent(nd);
142.         path_put(&nd->path);
143.         return ERR_PTR(error);
144. }

```

首先进行一些判断，然后看是否需要创建文件，如果需要创建的，则创建文件。如果文件存在的话，直接调用 `finish_open` 完成文件打开，我们这里关注下打开文件的

```

[cpp] view plain copy print ?

01. static struct file *finish_open(struct nameidata *nd,
02.                                int open_flag, int acc_mode)
03. {
04.     struct file *filp;
05.     int will_truncate;
06.     int error;
07.
08.     /*检测是否截断文件标志*/
09.     will_truncate = open_will_truncate(open_flag, nd->path.dentry->d_inode);
10.     if (will_truncate) { /*要截断的话就要获取写权限*/
11.         error = mnt_want_write(nd->path.mnt);
12.         if (error)
13.             goto exit;
14.     }
15.     //may_open执行权限检测、文件打开和truncate的操作
16.     error = may_open(&nd->path, acc_mode, open_flag);
17.     if (error) {
18.         if (will_truncate)
19.             mnt_drop_write(nd->path.mnt);
20.         goto exit;
21.     }
22.     filp = nameidata_to_filp(nd); /*根据nameidata 得到相应的file结构*/
23.     if (!IS_ERR(filp)) {
24.         error = ima_file_check(filp, acc_mode);
25.         if (error) {
26.             fput(filp);
27.             filp = ERR_PTR(error);
28.         }
29.     }
30.     if (!IS_ERR(filp)) {
31.         if (will_truncate) { //处理截断
32.             error = handle_truncate(&nd->path);
33.             if (error) {
34.                 fput(filp);
35.                 filp = ERR_PTR(error);
36.             }
37.         }

```

```

38.     }
39.     /*
40.      * It is now safe to drop the mnt write
41.      * because the filp has had a write taken
42.      * on its behalf.
43.      */
44.     if (will_truncate) //安全的放弃写权限
45.         mnt_drop_write(nd->path.mnt);
46.     return filp;
47.
48. exit:
49.     if (!IS_ERR(nd->intent.open.file))
50.         release_open_intent(nd);
51.     path_put(&nd->path);
52.     return ERR_PTR(error);
53. }

```

这里主要调用nameidata_to_filp得到相应的file结构

```

[cpp] view plain copy print ?
01. struct file *nameidata_to_filp(struct nameidata *nd)
02. {
03.     const struct cred *cred = current_cred();
04.     struct file *filp;
05.
06.     /* Pick up the filp from the open intent */
07.     filp = nd->intent.open.file; // 把相关 file结构的指针赋予 filp
08.     /* Has the filesystem initialised the file for us? */
09.     if (filp->f_path.dentry == NULL)
10.         filp = __dentry_open(nd->path.dentry, nd->path.mnt, filp,
11.                                NULL, cred);
12.     else
13.         path_put(&nd->path);
14.     return filp;
15. }

```

调用__dentry_open

```

[cpp] view plain copy print ?
01. static struct file *__dentry_open(struct dentry *dentry, struct vfsmount *mnt,
02.                                   struct file *f,
03.                                   int (*open)(struct inode *, struct file *),
04.                                   const struct cred *cred)
05. {
06.     struct inode *inode;
07.     int error;
08.
09.     f->f_mode = OPEN_FMODE(f->f_flags) | FMODE_LSEEK //初始化f_mode
10.               FMODE_READ | FMODE_WRITE;
11.     inode = dentry->d_inode;
12.     if (f->f_mode & FMODE_WRITE) {
13.         error = __get_file_write_access(inode, mnt);
14.         if (error)
15.             goto cleanup_file;
16.         if (!special_file(inode->i_mode))
17.             file_take_write(f);
18.     }
19.
20.     f->f_mapping = inode->i_mapping;
21.     f->f_path.dentry = dentry; //初始化目录项对象
22.     f->f_path.mnt = mnt; //初始化文件系统对象
23.     f->f_pos = 0;
24.     f->f_op = fops_get(inode->i_fop); //为文件操作建立起所有方法
25.     file_move(f, &inode->i_sb->s_files); //把文件对象插入到文件系统超级块的s_files字段所指向的
        打开文件的链表。
26.
27.     error = security_dentry_open(f, cred);
28.     if (error)
29.         goto cleanup_all;
30.
31.     if (!open && f->f_op) //传进来的open为NULL
32.         open = f->f_op->open;
33.     if (open) {
34.         error = open(inode, f);
35.         if (error)

```

```

36.         goto cleanup_all;
37.     }
38.     ima_counts_get(f);
39.
40.     f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC); //初始化f_f_flags
41.
42.     file_ra_state_init(&f->f_ra, f->f_mapping->host->i_mapping); //初始化预读的数据结构
43.
44.     /* NB: we're sure to have correct a_ops only after f_op->open */
45.     if (f->f_flags & O_DIRECT) { //检查直接IO操作是否可以作用于文件
46.         if (!f->f_mapping->a_ops ||
47.             ((!f->f_mapping->a_ops->direct_IO) &&
48.              (!f->f_mapping->a_ops->get_xip_mem))) {
49.             fput(f);
50.             f = ERR_PTR(-EINVAL);
51.         }
52.     }
53.
54.     return f;
55.
56. cleanup_all:
57.     fops_put(f->f_op);
58.     if (f->f_mode & FMODE_WRITE) {
59.         put_write_access(inode);
60.         if (!special_file(inode->i_mode)) {
61.             /*
62.              * We don't consider this a real
63.              * mnt_want/drop_write() pair
64.              * because it all happened right
65.              * here, so just reset the state.
66.              */
67.             file_reset_write(f);
68.             mnt_drop_write(mnt);
69.         }
70.     }
71.     file_kill(f);
72.     f->f_path.dentry = NULL;
73.     f->f_path.mnt = NULL;
74. cleanup_file:
75.     put_filp(f);
76.     dput(dentry);
77.     mntput(mnt);
78.     return ERR_PTR(error);
79. }

```

这里主要是进行一些赋值操作

对应于这里，传进来的open指针为NULL，如果相应file_operations结构存在的话就调用它的open函数

对于每个文件在创建的时候会赋值对其进行操作的file_operations结构，这个结构对于一类文件是一样的，例如对应于字符设备是chrdev_open

```

[cpp] view plain copy print ?
01. const struct file_operations def_chr_fops = {
02.     .open = chrdev_open,
03. };

```

但打开之后，我们可以重新获取它们的file_operations结构，这个是在注册设备驱动的时候为该类

也就是我们在驱动里面实现的，而前面的缺省file_operations就是为了完成这个转换的，def_chr_fops只起过渡作用，它的open方法要去找硬件驱动的支撑。

```

[cpp] view plain copy print ?
01. static int chrdev_open(struct inode *inode, struct file *filp)
02. {
03.     struct cdev *p;
04.     struct cdev *new = NULL;
05.     int ret = 0;
06.
07.     spin_lock(&cdev_lock);
08.     p = inode->i_cdev;
09.     if (!p) { /* 很显然，第一次打开的时候是NULL */
10.         struct kobject *kobj;
11.         int idx;
12.         spin_unlock(&cdev_lock);

```

```

13.         kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);/* 找到和设备号i_rdev对应的kobj,
    其实就是cdev了, 因为cdev中包含kobj; idx保存的是次设备号, 后面会分析kobj_lookup()函数 */
14.         if (!kobj)
15.             return -ENXIO;
16.         new = container_of(kobj, struct cdev, kobj);/*得到cdev
17.         spin_lock(&cdev_lock);
18.         /* Check i_cdev again in case somebody beat us to it while
19.            we dropped the lock. */
20.         p = inode->i_cdev;
21.         if (!p) {
22.             inode->i_cdev = p = new; /* 把找到的cdev保存到inode的icdev中 */
23.             list_add(&inode->i_devices, &p->list); /* inode加入到cdev的链表中 */
24.             new = NULL;
25.         } else if (!cdev_get(p))
26.             ret = -ENXIO;
27.         } else if (!cdev_get(p))
28.             ret = -ENXIO;
29.         spin_unlock(&cdev_lock);
30.         cdev_put(new);
31.         if (ret)
32.             return ret;
33.
34.         ret = -ENXIO;
35.         /*
36.         保存用户的fops, 以后你再调用read, write, ioctl系统调用的时候就直接使用了, 你懂的
37.         */
38.         filp->f_op = fops_get(p->ops);
39.         if (!filp->f_op) // 如果你没有注册fops
40.             goto out_cdev_put;
41.
42.         if (filp->f_op->open) { //判断open函数是否存在
43.             ret = filp->f_op->open(inode, filp); /* 调用用户的open函数, 我们前面写的驱动
44.             if (ret)
45.                 goto out_cdev_put;
46.         }
47.
48.         return 0;
49.
50.     out_cdev_put:
51.         cdev_put(p);
52.         return ret;
53.     }

```

在这个函数里, 我们重新 对f_op赋值了, 这里的f_op就是我们在写驱动时写的系统调用函数了。后面还调用了open方法

这里调用 kobj_lookup找到前面我们在注册驱动添加设备时添加的相应的kobj

```

[cpp] view plain copy print ?
01. struct kobject *kobj_lookup(struct kobj_map *domain, dev_t dev, int *index)
02. {
03.     struct kobject *kobj;
04.     struct probe *p;
05.     unsigned long best = ~0UL;
06.
07.     retry:
08.     mutex_lock(domain->lock);
09.     /* 根据主设备号和设备号查找它的一亩三分地。因为要支持2^12次方也就是4096个主设备号,
10.        但只使用了前255个主设备号索引, 所以这255个索引对应的probe结构都有一个单向
11.        链表保存着大于255的主设备号 (被255整除后的索引相等) */
12.     for (p = domain->probes[MAJOR(dev) % 255]; p; p = p->next) {
13.         struct kobject *(*probe)(dev_t, int *, void *);
14.         struct module *owner;
15.         void *data;
16.         // 比较, 看是否真找到了, 因为有链表存在
17.         if (p->dev > dev || p->dev + p->range - 1 < dev)
18.             continue;
19.         if (p->range - 1 >= best)
20.             break;
21.         if (!try_module_get(p->owner))
22.             continue;
23.         owner = p->owner;
24.         data = p->data; //data就是cdev
25.         probe = p->get;
26.         best = p->range - 1;
27.         *index = dev - p->dev; //得到次设备号
28.         /* 调用的lock就是exact_lock()函数, 增加对该字符设备驱动的引用, 防止被卸载什么的 */
29.         if (p->lock && p->lock(dev, data) < 0) {

```

```
30.         module_put(owner);
31.         continue;
32.     }
33.     mutex_unlock(domain->lock);
34.     /*调用的probe就是exact_match()函数，获取cdev的kobj指针 */
35.     kobj = probe(dev, index, data);
36.     /* Currently ->owner protects _only_ ->probe() itself. */
37.     module_put(owner);
38.     if (kobj)
39.         return kobj;
40.     goto retry;
41. }
42. mutex_unlock(domain->lock);
43. return NULL;
44. }
```

到这里do_filp_open的流程就基本完成了，即返回了一个file结构

```
[cpp] view plain copy print ?
01. void fd_install(unsigned int fd, struct file *file)
02. {
03.     struct files_struct *files = current->files;
04.     struct fdtable *fdt;
05.     spin_lock(&files->file_lock);
06.     fdt = files_fdtable(files); //获取fdtbale
07.     BUG_ON(fdt->fd[fd] != NULL);
08.     rcu_assign_pointer(fdt->fd[fd], file); //fd和file关系到fdtbale
09.     spin_unlock(&files->file_lock);
10. }
```

这样，我们的open系统调用就基本上完成了，我们得到一个fd，这个fd关联着一个file结构，这样以后，我们就可以通过这个fd结构操作相应的文件了
最后还是一样看下整个简略的流程图



顶 0 踩 0


上一篇 吉他基础乐理二
下一篇 吉他第三课

相关文章推荐


- linux硬件设备操作函数 `open(/dev/ietctl, O_RDWR...`
 - 深度学习部署系统构建-刘文志
 - 字符设备之`open()`与`release()`函数
 - 搜狗机器翻译技术分享-陈伟
 - 字符设备文件的打开操作 `open()`
 - Hadoop生态系统零基础入门
 - linux设备驱动模型一字符设备open系统调用流程
 - 最懂程序员的学习方式 TensorFlow入门
- 吉他基础乐理二
 - Retrofit 从入门封装到源码解析
 - Linux open系统调用流程
 - 程序员如何转型AI工程师-蒋涛
 - linux驱动学习3: 实现一简单完整驱动(包括`open`,`r...`
 - 应用层`open`如何调用驱动`open`函数的?
 - Linux应用程序访问字符设备驱动详细过程解析
 - android驱动 无法打开设备文件 解决

查看评论

2楼 [lewisgre](#) 2016-06-16 02:12发表

 最近开始在看ldd3,遇到驱动里面的open一头雾水不知道参数inode相关的是怎么来的，也在找open系统调用到驱动里面的open的调用关系，看了你的文章真对路，明天再仔细看看，非常感谢，对了，最后面的图片被删除看不到了。

Re: [new_abc](#) 2016-06-16 17:29发表

 回复lewisgre: 我也是从网上摘抄的，整理了一下，图片不小心删除了，没有了

1楼 [xiayuleWA](#) 2015-04-15 15:58发表

 写的真好

Re: [new_abc](#) 2015-06-04 09:09发表

 回复xiayuleWA: 谢谢。

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场