2.1.2. 有趣的 VFS 和文件系统

VFS 在 linux 的出现 ,不得不惊叹是一个神来之笔 ,下面先来看一下内核代码中的 ext4 文件系统与 VFS 关系图。

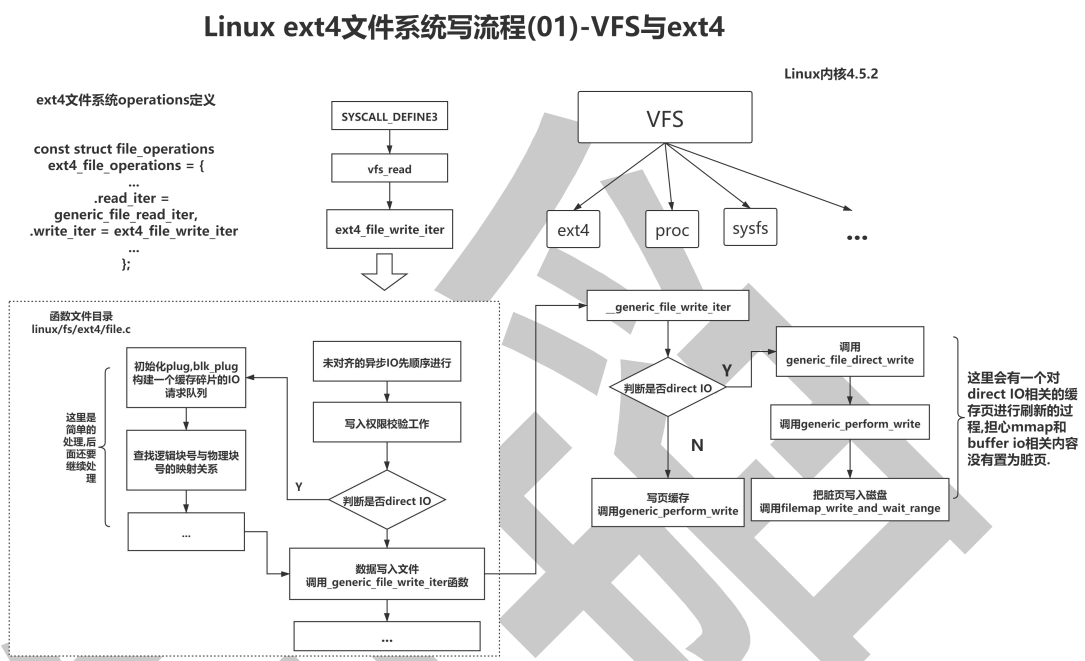

图 2.1.1-1 vfs 与 ext4 代码关系图

首先图中的一个读请求，是经过了 vfs_read 然后调用了 ext4_file_write_iter，而后者则是 ext4 文件系统注册在操作系统中的 FOP,这里在调用的时候，会根据文件系统注册的函数进行回调 ,然后该函数就是真正地进入了文件系统的操作逻辑了 ,这里也就是有了两个分支,就是前面提到的是否使用操作系统内核缓存的问题了。下面为了让大家更好地理解一下 VFS 的元素，将具体介绍一下内容。

2.1.2.1. VFS inode

为了更好地理解一下 inode 信息，下面给出一个普通文件的 stat 命令结果，该命令可

以列出文件的一些属性信息。

```
$ sudo ls -i test.sh
1328 test.sh


$ sudo stat test.sh
  File: test.sh
  Size: 614         Blocks: 19         IO Block: 1024   regular file
Device: 74h/116d    Inode: 1328        Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2020-11-07 03:21:30.777590296 +0000
Modify: 2020-11-07 03:21:26.485583231 +0000
Change: 2020-11-07 03:21:26.493583244 +0000
 Birth: -
```

这里 inode 号为 1328,这里还要留意一下 change,modify 和 access 三个时间，这里也就是 ctime,mtime 和 atime，在 glusterfs 中的文件，也是利用了这三个信息进行一些文件更新校验的工作。另外这里的 size，也是在 glusterfs 中使用的一个很重要的信息。

change: 也就是 ctime,最后一次改变文件或目录(属性)的时间，例如执行 chmod，chown 等命令。

modity: 也就是 mtime,是最后一次修改文件或目录（内容）的时间。

access: 也就是 atime,是最后一次访问文件或目录的时间。

```
$ sudo stat dir
  File: dir
  Size: 2           Blocks: 1          IO Block: 131072 directory
Device: 74h/116d    Inode: 131881      Links: 2
Access: (0755/drwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2021-06-03 03:21:17.577886842 +0000
Modify: 2021-06-03 03:21:17.577886842 +0000
Change: 2021-06-03 03:21:17.577886842 +0000
 Birth: -
```

与文件的不同，目录的 stat 信息中的 Links 注意是 2 的，其他信息则相似。当然 inode 中的信息远远不只这么少，还包括了其他很多的信息，例如包括了用户和所属组的权限，关

联的 superblock 等信息。具体想了解的可以看看 inode 的数据结构，该结构体定义在<linux/fs.h>中。

```
struct inode
{
    /* 哈希表 */
    struct hlist_node       i_hash;
    /* 索引节点链表 */
    struct list_head        i_list;
    /* 目录项链表 */
    struct list_head        i_dentry;
    ...
    uid_t                   i_uid;
    gid_t                   i_gid;
    ...
    struct timespec         i_atime;
    struct timespec         i_mtime;
    struct timespec         i_ctime;
    ...
    /* 索引节点操作表 */
    struct inode_operations *i_op;
    /* 默认的索引节点操作 */
    struct file_operations  *i_fop;
    /* 相关的超级块 */
    struct super_block      *i_sb;
    /* 文件锁链表 */
    struct file_lock        *i_flock;
    /* 相关的地址映射 */
    struct address_space    *i_mapping;
    ...
}
```

2.1.2.2. VFS superblock

对于超级块来说，可以使用 df -i 命令来简单查看一下操作系统层面的 inode 使用情况。

```
$ sudo df -i
Filesystem       Inodes   IUsed    IFree IUse% Mounted on
udev            2023350     620  2022730    1% /dev
```

```
tmpfs           2029590    1379  2028211    1% /run
/dev/nvme0n1p2 30498816 1481554 29017262    5% /
tmpfs           2029590    3222  2026368    1% /dev/shm
tmpfs           2029590       7  2029583    1% /run/lock
tmpfs           2029590      18  2029572    1% /sys/fs/cgroup
```

这里使用 df -i 命令可以直观地看到目前系统中使用的 inode 情况 ,同时当 inode 耗尽的时候，也是一个非常要注意的问题。这些在日常的操作系统维护中都是需要监控的内容。另外下面给出 superblock 的对象结构。

```
struct super_block {
        ...
        //关联设备
        dev_t   s_dev;
       //块大小
       unsigned long s_blocksize;
       ...
       //文件系统类型,常见的有 ext,xfs 等
       struct file_system_type *s_type;
      //超级块的操作函数
       const struct super_operations *s_op;
       ...
       //该 super_block 中所有 inode 的 i_sb_list 成员的双向链表
       struct list_head s_inodes;
       ...

}
```

2.1.2.3. VFS struct file

struct file 结构体定义在 include/linux/fs.h 中 ,其中每一个文件结构体表示被进程打开的一个文件，而系统中每个被打开的文件，都可以在内核空间中找到关联的一个 struct file 对象。这个对象由内核在打开文件时创建，并且传递给在文件上进程操作的函数。在文件的所有实例都关闭后，内核释放这个数据结构。下面先简单感受一下在操作系统中，文件打开后的变化。

```
[root@gfsclient01 ~]# lsof  |grep test
COMMAND    PID  TID    USER   FD      TYPE             DEVICE  SIZE/OFF       NODE NAME
....
vim       1827         root    4u      REG              253,0     12288  100663370 /root/.test.sh.swp
/root/.test.sh.swp
[root@gfsclient01 ~]# ps -ef |grep 1677
root      1827  1529  0 10:09 pts/0    00:00:00 vim test.sh
```

在 linux 中，打开了一个文件之后，其实并不是真正意义上直接对原文件进行修改的，而是会把原文件复制一份加上.swp 之后，对应的所有的操作都会在该临时的缓存文件中，而一旦修改完成进行合并的时候，才会替换到原文件。当然这里会有常见的文件冲突，就是当文件被打开之后，如果再次打开该文件，操作系统会检测到有临时缓存文件存在，那么就证明可能是有修改没完成，或者之前的修改中断了没有合并，这样会导致冲突，但是可以进行替换操作等恢复。

下面给出一种使用文件句柄来恢复数据的方式。当该文件有被进程引用 fd，那么就有机会恢复原来的数据。

```
[root@gfsclient01 ~]# lsof  |grep test_file.txt
tail      2165         root    3r      REG              253,0        10  100730896 /root/test_file.txt
[root@gfsclient01 ~]# ls -l /proc/2165/fd/3
lr-x------. 1 root root 64 Jun  3 10:20 /proc/2165/fd/3 -> /root/test_file.txt
[root@gfsclient01 ~]# cat  /proc/2165/fd/3
#123

321
[root@gfsclient01 ~]# rm -fr test_file.txt
[root@gfsclient01 ~]# cat  /proc/2165/fd/3 > abc.txt
[root@gfsclient01 ~]# cat abc.txt
#123

321
```

这里首先创建了一个名字为 test_file.txt 的文件，然后在一个终端使用 tail -f 来阅读该文件，这样可以保证文件一直在被打开的状态，同时 fd 中也可以看到是 read 状态。接着通过 lsof 找到该文件对应的引用进程，在 proc 下可以找到该文件的缓存内容，这样就可以进行数据恢复了。

同样的，为了进一步了解 struct file 的数据结构，该结构定义在内核代码中的 include/linux/fs.h 中,下面给出部分代码内容。

```
struct file {
    ...
    struct path           f_path;
    struct inode         *f_inode;
    //和文件关联的操作.
    const struct file_operations   *f_op;
    spinlock_t     f_lock;
    atomic_long_t      f_count;
   //文件标志,有 O_REONLY 和 O_SYNC 等.
    unsigned int       f_flags;
   //文件模式
    fmode_t        f_mode;
    struct mutex       f_pos_lock;
    //当前读写位置
    loff_t         f_pos;
    struct fown_struct f_owner;
    const struct cred               *f_cred;
    struct file_ra_state   f_ra;
     ...
}
```

2.1.2.4. VFS dentry

dentry，也就是目录项，是多个文件或者目录的链接，通过这个链接可以找寻到目录之下的文件或者是目录项。dentry 在文件系统里是极其重要的一个概念，dentry 结构体

在 linux 内核里也是用处广泛，这个结构体定义在 include/linux/dcache.h 里。

```
struct dentry {
        //目录项标志
    unsigned int d_flags;
    seqcount_t d_seq;
        //散列表表项的指针
    struct hlist_bl_node d_hash;
        //父目录的目录项
    struct dentry *d_parent;
    struct qstr d_name;
       // 与文件名关联的索引节点
    struct inode *d_inode;
    unsigned char d_iname[DNAME_INLINE_LEN];


    struct lockref d_lockref;
        //dentry 的操作函数
    const struct dentry_operations *d_op;
        //文件的超级块对象
    struct super_block *d_sb;
    unsigned long d_time;
    void *d_fsdata;

    struct list_head d_lru;
    //父列表的子级
    struct list_head d_child;
    struct list_head d_subdirs;

    union {
        struct hlist_node d_alias;
        struct rcu_head d_rcu;
    } d_u;
};
```

为了观察操作系统中目录的关系结构，可以使用 tree 命令进行查看。

```
[root@gfsclient01 ~]# tree /tmp/
/tmp/
├── ks-script-YbuVGl
├── systemd-private-4556ac58b212443eb846ec7ab9a7f059-chronyd.service-soC2Da
```

```
|   └── tmp
├── yum.log
└── yum_save_tx.2021-06-03.09-45.cN3Oby.yumtx

2 directories, 3 files
```

2.1.2.5. 文件对象

有了前面的四个核心基本元素，那么如果要打开一个文件，就必须要使用文件对象了，它的作用是描述进程和文件交互的关系，说白了就是要知道哪个进程在打开这个文件，并且打开读取的文件位置等信息。文件的结构定义如下所示。

```
struct file {
        //指向文件对应的 dentry 结构
        struct dentry                           *f_dentry;
         //指向文件所属于的文件系统的 vfsmount 对象,和挂载 有关
        struct vfsmount                        *f_vfsmnt;
        //指向文件的 file operation
        const struct  file_operations     *f_op;
        ...
       //记录进程对文件操作的位置。对文件读取前 n 字节，那么其指向 n+1 字节位置
        loff_t                                          f_pos;

        struct fown_struct                    f_owner;
        //表示文件的用户的 uid 和 gid
        unsigned     int                        f_uid;
        unsigned     int                        f_gid;
        //用于记录文件预读的设置
        struct file_ra    state                f_ra;

       //这个结构指是指向一个封装文件的读写缓存页面的结构
        struct   address_space           *f_mapping;

}
```