参数	作用
diagnostics.brick-log-level	控制 brick 端的日志级别,影响 glusterd 和 brick 日志输出
diagnostics.client-log-level	控制 client 日志输出级别

章节语:

在最开始，我们尝试着在部署了一下 glusterfs 集群之后，简单地创建和体验了一些不同类型的 volume 特点，去简单理解一下 glusterfs 的使用，而仲裁节点的出现，也是一个非常好的亮点，同时还去关心了一下集群运维中比较重要的日志文件内容。

附录引用:
[1] https://gluster.readthedocs.io/en/latest/
[2] https://gluster.readthedocs.io/en/latest/Administrator-Guide/Logging

2.第二章 文件系统的那些事

2.1. 文件系统的层次结构

生产环境的服务器中使用的基本上是 linux 操作系统,而操作系统中会涉及到进程模块,内存管理和文件系统等各大模块，那么对于一个正常的文件系统，如常见的 ext4 和 xfs 这些,一个普通的读写会经历哪些层次呢？这些都是非常值得关心的内容,下面请先看一张图。

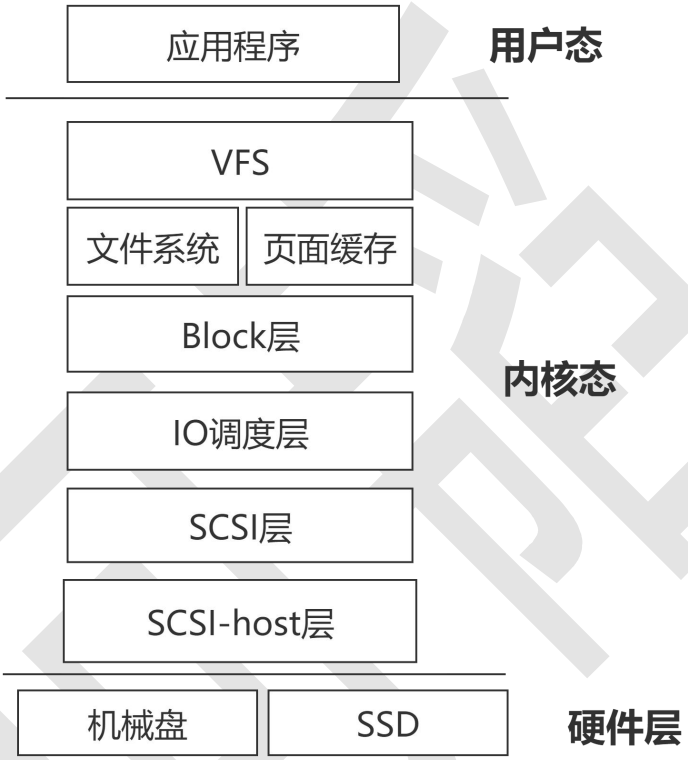


图 2.1-1 linux 文件系层次图

对于一个正常的应用服务，如果要读写本地磁盘数据，会向本地的文件系统发起请求,例如使用 mkdir 或者 touch 这样的 shell 命令进行操作，那么首先会经过一个叫 VFS 层，这个其实是 linux 操作系统中对于文件系统的一个规划,也就是说,不管是自定义的操作系统,还是一些 ext4 这些，都需要遵守的一些规范，而 VFS 中最核心的元素就是 inode,dentry,superblock 和 file 四个元素。

inode 是负责记录每个文件的一些元数据信息的(在 linux 系统中，一切皆文件，不管是目录还是 devices 设备，都会封装看成一个文件一样调用)。dentry 对象则是用于记录文件的结构关系的，也就是不同目录的上下级层级结构关系和树状关系等。 superblock 就是超级块,用于管理整个操作系统中 inode 资源整体使用情况等,如果把文件系统比作一个图书馆，那么图书馆里面的每一本书就是 block，而书的分类和标签信息这些就是 inode,superblock 就是统计整理整个图书馆的资源情况的。file 对象就是用于记录一下每一本书的租借情况，对于文件来说，这个文件是否被打开过，是否产生了文件句柄 fd(所谓的

文件句柄可以理解为，在 file 对象和打开的进程的数据结构中做了一些标记)等。

定义了 VFS 的规范之后，所有的文件系统的都要使用这四个元素进行管理文件系统，而每一个文件系统，在注册到操作系统里面时，还需要定义该文件系统的 fileOptions(这里可以理解为，每个文件系统的一些操作如何定义与具体实现，例如打开一个文件时，是否使用缓存，或者实现一些特殊的操作定义，在 glusterfs 中也有该实现，叫做 FOP)。 那么有了这些 FOP 之后，一个正常的读写请求就会到达该文件系统，根据调用的函数不同，这里就会区分到底请求是否使用缓存了，也就是是否使用操作系统的缓存，而使用这个缓存的优势与缺点也是非常明显的，如果想达到快速读取，那么如果缓存中有该数据，就不需要从磁盘中进行读取了，但是对于写入来说，尤其是数据库的一些写入，因为很多在应用的内存里面已经做了一次缓存，因此并不需要再次做缓存，而且数据库是对于写入 IO 是比较敏感的服务，为了加快写入速度，通常会不使用操作系统的缓存的。

不管是否使用缓存，对于一个读写请求来说，最后都是要到达 block 层的，而这一层的出现，其实就是为了封装所有的读写请求为一个 bio 对象。而为什么需要 block 层，一个很重要的原因就是，操作系统上面，可能会有多个不同的文件系统，不管任何文件系统最后想要对磁盘进行读写之前，都要进行统一的格式封装，而这就是 block 层的 bio 对象所需要做的事情了，同时在这里，对于读写请求来说，还有考虑能否合并请求，如果两个请求的写入磁盘位置是前后关联的，那么这里就可以进行合并操作了，如果请求可以进行合并的话，那么封装后的 bio 对象就会传递给调度队列了，接下来就是常见的调度算法进行操作了，有 Deadline 和 CFQ 等常见的算法。

当然对于 glusterfs 来说，也实现了一套 fuse，GlusterFS 是一个用户空间文件系统。GluserFS 开发人员选择这种方法是为了避免在 Linux 内核中使用模块。下图将简单展示一下 glusterfs 的 fuse，结合 linux 文件系统的内容，其实两者是有非常多的共同点的。

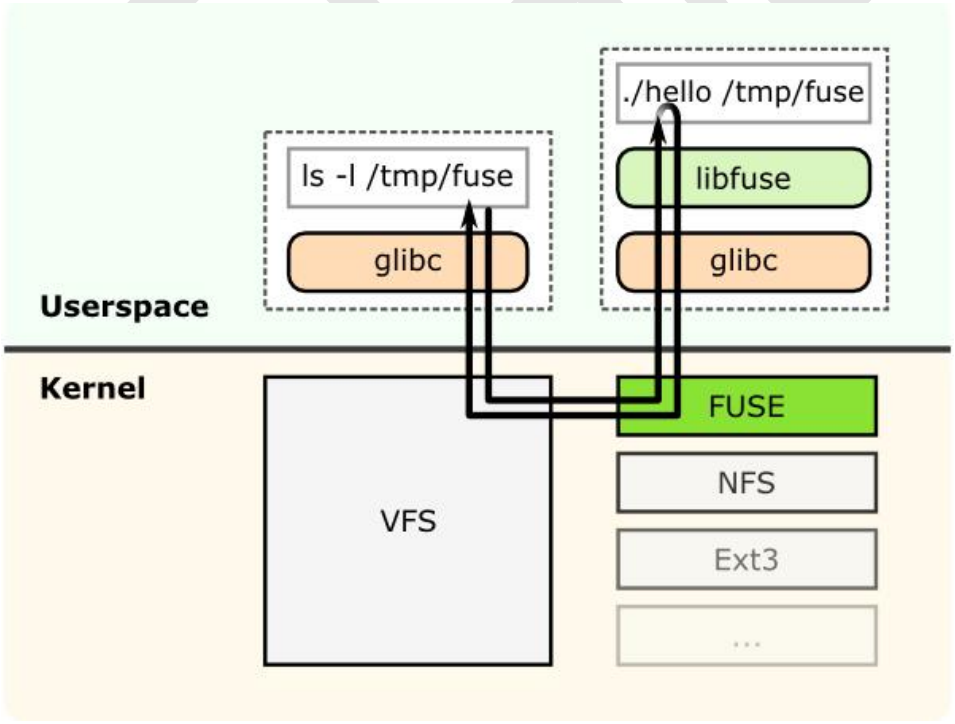


图 2.1-2 glusterfs fuse 架构图

接下来，将结合代码，具体介绍一下上面提到的内容，该部分内容是为了让大家更好地理解一些概念，如果已经熟悉该部分内容，可以跳过阅读。

2.1.1. 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。

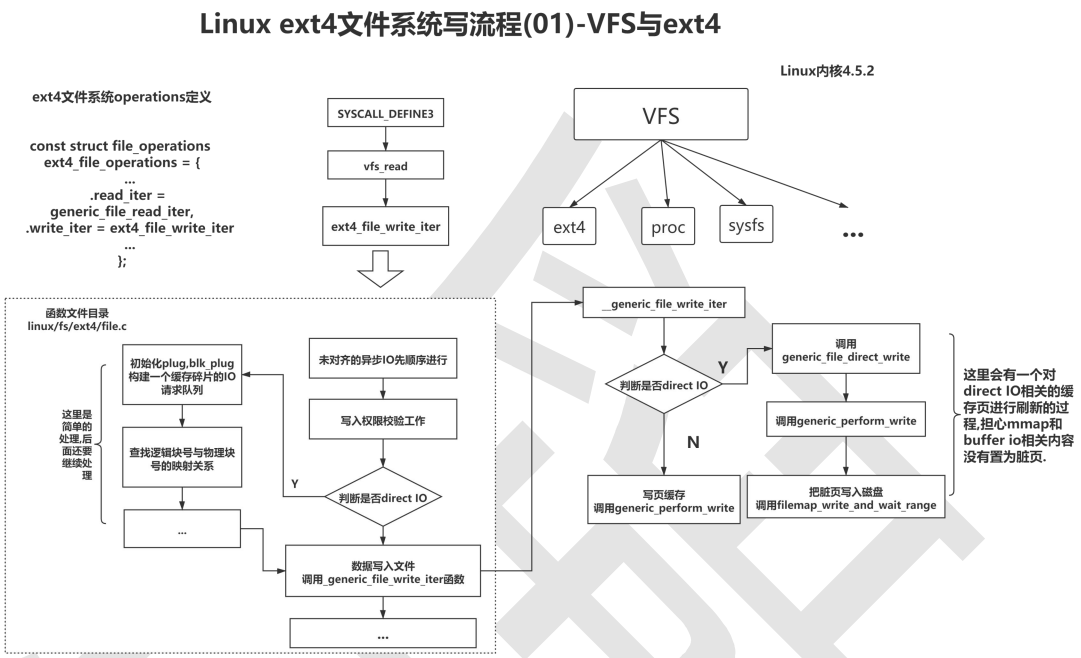


图 2.1.1-1　vfs 与 ext4 代码关系图

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。

2.1.1.1. 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;
    ...
}
```

2.1.1.2. 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;
    ...

}
```

2.1.1.3. 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;
    ...
}
```

2.1.1.4. 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
```

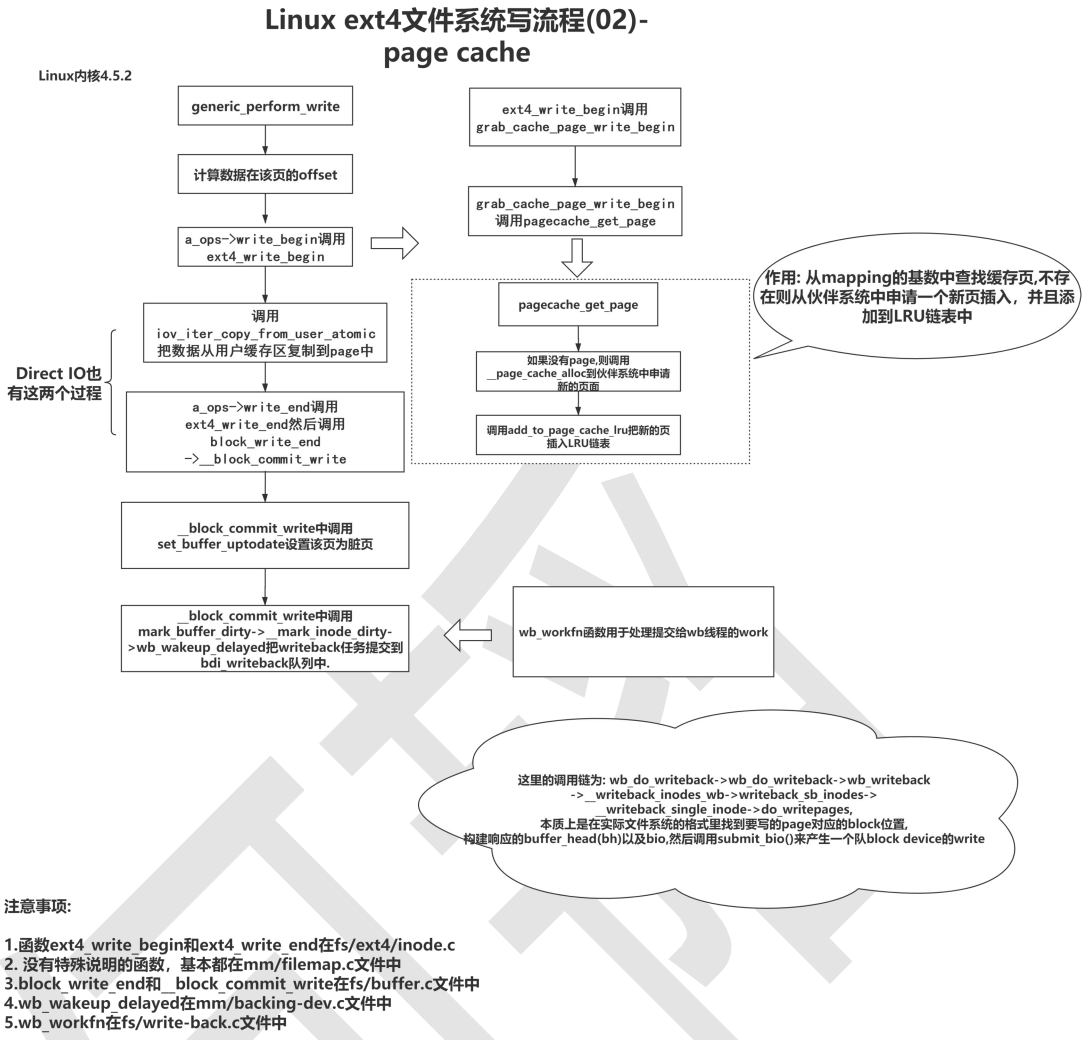
2.1.1.5. 文件对象

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;

}
```

2.1.2. page cahe 的作用

回到前面提的内容，对于一个读写请求来说，经过前面的 VFS 和文件系统的调用，那么下面就会进入到内核调用中，而如果要使用内核缓存，这里就有了 page cache 的存在目的了。下面先来简单看一张代码调用逻辑图，理解一下其中的思路。



对于一个读写请求, 如果这里发现在缓存中没有找到对应的数据, 那么就需要申请 page 缓存页了，而这里会涉及到 linux 操作系统的内存管理模块伙伴系统和 mapp 相关的内容。而所谓的伙伴系统，如果熟悉 java jvm 的话，其实这里的思想是类似的。伙伴系统是一个结合了 2 的方幂个分配器和空闲缓冲区合并计技术的内存分配方案，其基本思想很简单. 内存被分成含有很多页面的大块，每一块都是 2 个页面大小的方幂. 如果找不到想要的块，一个大块会被分成两部分，这两部分彼此就成为伙伴. 其中一半被用来分配，而另一半则空闲. 这些块在以后分配的过程中会继续被二分直至产生一个所需大小的块. 当一个块被最终释放时，其伙伴将被检测出来，如果伙伴也空闲则合并两者。

那么不管内存是如何管理的，申请出来的页面，最后会被存放到一个叫做 LRU 链表进行管理。在 Linux 中，操作系统对 LRU 的实现主要是基于一对双向链表：active 链表和 inactive 链表，这两个链表是 Linux 操作系统进行页面回收所依赖的关键数据结构，每个内存区域都存在一对这样的链表。顾名思义，那些经常被访问的处于活跃状态的页面会被放在 active 链表上，而那些虽然可能关联到一个或者多个进程，但是并不经常使用的页面则会被放到 inactive 链表上。页面会在这两个双向链表中移动，操作系统会根据页面的活跃程度来判断应该把页面放到哪个链表上。页面可能会从 active 链表上被转移到 inactive 链表上，也可能从 inactive 链表上被转移到 active 链表上，但是，这种转移并不是每次页面访问都会发生，页面的这种转移发生的间隔有可能比较长。那些最近最少使用的页面会被逐

个放到 inactive 链表的尾部。进行页面回收的时候，Linux 操作系统会从 inactive 链表的尾部开始进行回收。

而内核的内存管理部分，会有三个概念，分别是 node,zone 和 page,简单理解，page 就是一个数据页，zone 是一个区域分组，CPU 被划分为多个节点(node)，内存则被分簇，每个 CPU 对应一个本地物理内存，即一个 CPU-node 对应一个内存簇 bank，即每个内存簇被认为是一个节点。

有了这些缓存的数据页之后，那么数据就会从用户缓存区复制到 page 当中，这里就会涉及到内核调用，同时为了进一步优化，这里会有零拷贝的技术出现了。当然因为这里并不打算深入具体去了解每一部分的内容，因此如果感兴趣的话，可以自行查阅相关资料。

而当数据存放到内核缓存之后，那么写入的请求会把请求封装成一个 bio 对象，提交到 bdi_writeback 队列中，而这里会有一个 writeback 机制，也就是回写机制，下面就是 block 层的任务了。

2.1.3. Fuse block

为了更好地简单理解读写请求后面要做的事情，可以先看看下面这张图。

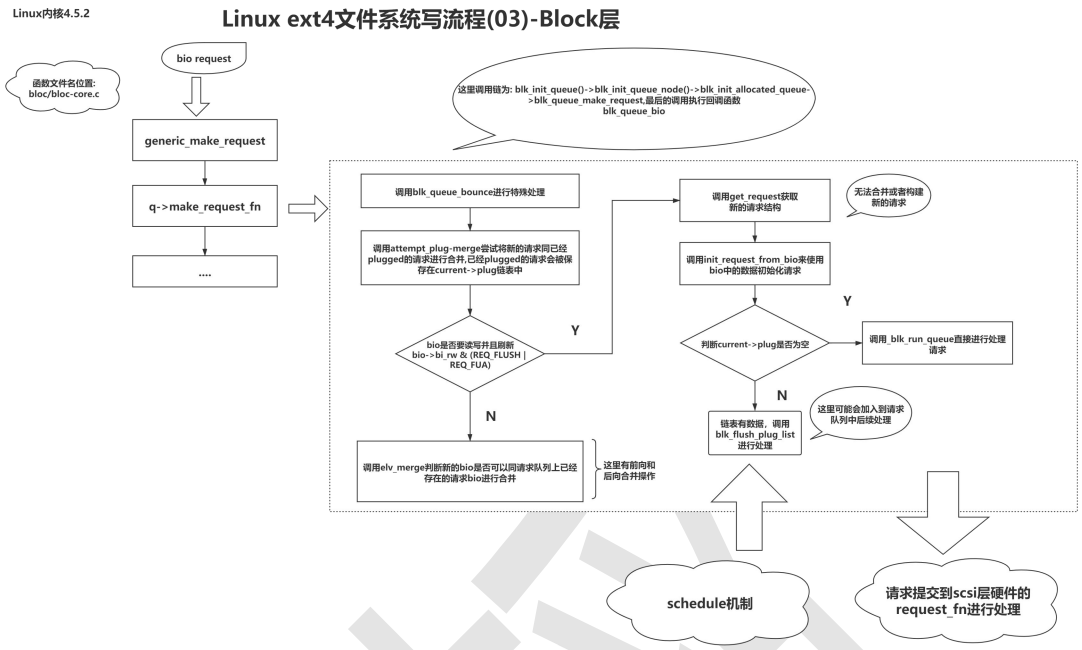


图 2.1.3-1 fuse block 层原理

对于一个 IO 请求，除了前面提到的把数据写入到缓存以外，其实还有一个叫直接 IO 的内容，那么关于 IO 我们常常还有听到同步与异步 IO 的内容，而异步 IO 则只能使用直接 IO 来实现的。另外对于一个 bio 请求来说，这里还会进行前向与后向的合并，之后会放入到调度队列里面，等待调度算法来进行调度处理。

最后，在经历了一系列的原理的理解之后，我们使用 strace 命令来了解一下，在 linux 系统中创建一个文件时使用的函数调用吧。

```
# strace touch 1.txt
execve("/usr/bin/touch", ["touch", "1.txt"], 0x7ffe4906edc8 /* 24 vars */) = 0
brk(NULL)                               = 0x13ef000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff5b1044000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=25560, ...}) = 0
mmap(NULL, 25560, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff5b103d000
close(3)                                = 0
open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`&\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2156352, ...}) = 0
mmap(NULL, 3985920, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ff5b0a56000
mprotect(0x7ff5b0c1a000, 2093056, PROT_NONE) = 0
mmap(0x7ff5b0e19000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c3000) = 0x7ff5b0e19000
mmap(0x7ff5b0e1f000, 16896, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7ff5b0e1f000
```

```
close(3)                        = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff5b103c000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff5b103a000
arch_prctl(ARCH_SET_FS, 0x7ff5b103a740) = 0
mprotect(0x7ff5b0e19000, 16384, PROT_READ) = 0
mprotect(0x60d000, 4096, PROT_READ)     = 0
mprotect(0x7ff5b1045000, 4096, PROT_READ) = 0
munmap(0x7ff5b103d000, 25560)           = 0
brk(NULL)                               = 0x13ef000
brk(0x1410000)                          = 0x1410000
brk(NULL)                               = 0x1410000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=106176928, ...}) = 0
mmap(NULL, 106176928, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff5aa513000
close(3)                        = 0
open("1.txt", O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK, 0666) = 3
dup2(3, 0)                      = 0
close(3)                        = 0
utimensat(0, NULL, NULL, 0)             = 0
close(0)                        = 0
close(1)                        = 0
close(2)                        = 0
exit_group(0)                   = ?
+++ exited with 0 +++
```

这里有常见的 mmap 和 brk 是和内存管理分配有关的函数, open 是文件打开的函数等, 其中 open 函数中的参数 O_CREAT 是创建并打开一个新文件,关于这些不同的函数的作用与参数意义，可以根据需要时去查阅接口文档。

章节语:

这一章我们主要是理解了一些 linux 文件系统中的概念，一个正常的读写请求所经过的层次结构，而这里面涉及到内存管理，进程调度等模块，本章也只是非常粗糙简略地讲解了 VFS 和缓存的一些内容，而这个章节的内容，也是为了方便理解后续 glusterfs fuse 中的一些参数做准备的, 因为 glusterfs fuse 的底层也是很多我们所熟悉的系统调用, 而且 glusterfs fuse 是一个用户文件系统。