Linux虚拟文件系统

http://www.ilinuxkernel.com

目 录

图目录

1 概述

Linux支持多种不同文件系统，要实现这个目的，就要将对各种不同文件系统和管理纳入到一个统一的框架中，让内核中的文件系统界面成为一条文件系统“总线”，使用户程序可以通过同一个文件系统操作界面，也就是同一组系统调用，对各种不同的文件系统（以及文件）进行操作。这样，就可以对用户程序隐去各种不同文件系统的细节，为用户程序提供一个统一的、抽象的、虚拟的文件系统界面，这就是所谓“虚拟文件系统”VFS（Virtual Filesystem Switch）。这个抽象界面主要由一组标准的、抽象的文件操作构成，以系统调用的形式提供于用户程序，如read（）、write（）、lseek（）等等。这样，用户程序就可以把所有的文件都看作一致的、抽象的“VFS文件”，通过这些系统调用对文件进行操作，而无需关心具体的文件属于什么文件系统以及具体文件系统的设计和实现。例如，在Linux操作系统中，可以将DOS格式的磁盘或分区（即文件系统）“安装”到系统中，然后用户程序可以安完全相同的方式访问这些文件，就好像它们也是ext2格式的文件一样。

在我们使用C语言编写应用程序时，相信会经常使用到write（）系统调用；就是向一个文件中写入数据。write（）函数的原型为

ssize_t write(int fd, const void *buf, size_t count);

在用户程序的write（f, &buf, len），向文件描述符为f的文件中，写入len个字节数据，待写入的数据存放在buf中。下图为write（）将数据写入硬件上的简易流程。我们看到首先通过虚拟文件系统VFS，然后根据不同文件系统的write（）方法将数据写入物理设备上。

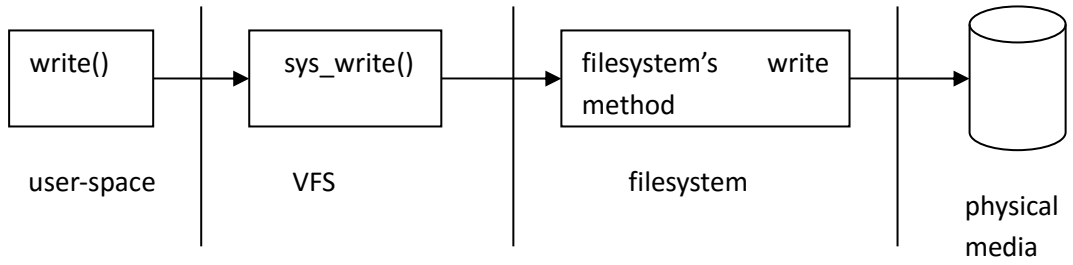

图1 write（）流程图

本文的内容组织如下：首先介绍VFS的四个主要对象和相关操作，然后介绍和文件系统、进程相关的数据结构和操作。

2 VFS主要对象及其数据结构

虚拟文件系统（VFS）支持的文件系统可以分成主要三种类型

- 基于磁盘（disk-based）的文件系统

管理本地磁盘和模拟磁盘的设备，基于磁盘的文件系统有：

- ✓ Linux的Ext2、Ext3和ReiserFS
- ✓ Unix文件系统的变种，如sysv、UFS、MINIX和VERITAS VxFS。
- ✓ 微软的文件系统，如MS-DOS、VFAT、NFTS
- ✓ ISO9660 CD-ROM文件系统和UDF（Universal Disk Format）DVD文件系统
- ✓ 其他文件系统如IBM OS/2中的HPFS、Apple's Macintosh的HFS、AFFS和ADFS等。

- 网络文件系统

它允许方便访问网络上其他计算机的文件系统。VFS支持的网络文件系统有NFS、Coda、AFS（Andrew filesystem）、CIFS（Common Internet Filesystem）和NCP（Novell's NetWare Core Protocol）。

- 特殊文件系统

它不管理磁盘空间。/proc就是一个特殊的文件系统。

虚拟文件系统有四个主要对象类型：

（1）spuerblock

表示特定加载的文件系统。

（2）inode

表示特定的文件。

（3）dentry

表示一个目录项，路径的一个组成部分。

（4）file

表示进程打开的一个文件。

上面四个对象，都有相应的操作方法

（1）super_operations

文件系统的方法，如read_inode（）和sync_fs（）。

（2）inode_operations

文件的方法，如create（）和link（）。

（3）dentry_operations

目录项的方法，如d_compare（）和d_delete（）。

（4）file

进程在打开文件上方法，如read（）和write（）。

以上是四个主要的VFS对象，存在其他的VFS对象。如每个注册的文件系统由file_system_type结构表示，这个对象描述文件系统和它的功能。进而每个挂载点由vfsmount数据结构表示，这个对象包含挂载的位置和标志信息。另外有三个数据结构和进程有关，来表示进程使用的文件系统和文件信息，它们是file_struct，fs_struct和namespace。

2.1 Unix文件系统

Unix提供了四个基本的文件系统相关抽象：文件、目录项、inode和挂载点。

文件系统是在特定的结构上，存储层次化的数据。文件系统包含文件、目录和相关的控制信息。在文件系统上的操作通常有：创建、删除和挂载。在Unix中，文件系统挂载在全局层次结构（global hierarchy）中的一个节点上，这个全局层次结构称为名空间（namespace）。这可以使用所有加载的文件系统，看上去上树上的一个节点。

文件是有顺序的字节流。每个文件都有一个可阅读（human-readable）的文件名。常用的文件操作包括读、写、创建和删除。

文件存放在目录里。目录和文件夹类似，它通常包含文件；目录也可以有子目录。目录交织形成路径，路径的每个组成部分被称为目录项（directory entry）。如“/root/ssd/chen”，根目录/，目录root、ssd和文件chen都是目录项。在Unix中，目录事实上也是普通的文件，只是文件的内容就是目录中的所有文件名。对于VFS来说，目录就是一个文件，所有对文件的操作也可用在目录上。

对于不熟悉Unix或Linux的人来说，inode会感觉比较陌生。Unix系统将文件数据和文件本身信息（属性）分开，文件访问权限、大小、拥有者、创建时间等。这些信息称为file

metadata，并且和文件数据分开存储，这就是inode。

文件系统的控制信息存放在超级块（superblock）中。超级块是包含文件系统信息的数据结构，有时这些文件系统信息称之为filesystem metadata。

Linux的虚拟文件系统就是基于这些概念实现的。非Unix的文件系统，如FAT或NTFS，仍可以在Linux中使用，只要这些文件系统提供这些概念的形式即可。

2.2 超级块对象

超级块（spuerblock）对象由各自的文件系统实现，用来存储文件系统的信息。这个对象对应为文件系统超级块或者文件系统控制块，它存储在磁盘特定的扇区上。不是基于磁盘的文件系统（基于内存的虚拟文件系统，如sysfs）临时生成超级块，并保存在内存中。

超级块对象由结构struct super_block表示，定义在include/linux/fs.h中。

```
01371: struct super_block {
01372:     struct list_head    s_list;       /* Keep this first */
01373:     dev_t               s_dev;        /* search index; _not_ kdev_t */
01374:     unsigned long       s_blocksize;
01375:     unsigned char       s_blocksize_bits;
01376:     unsigned char       s_dirt;
01377:     loff_t              s_maxbytes;   /* Max file size */
01378:     struct file_system_type    *s_type;
01379:     const struct super_operations*s_op;
01380:     const struct dquot_operations      *dq_op;
01381:     const struct quotactl_ops   *s_qcop;
01382:     const struct export_operations   *s_export_op;
01383:     unsigned long       s_flags;
01384:     unsigned long       s_magic;
01385:     struct dentry       *s_root;
01386:     struct rw_semaphore    s_umount;
01387:     struct mutex        s_lock;
01388:     int             s_count;
01389:     int             s_need_sync;
01390:     atomic_t        s_active;
01391: #ifdef CONFIG_SECURITY
01392:     void                *s_security;
01393: #endif
01394:     struct xattr_handler       **s_xattr;
01395:
01396:     struct list_head    s_inodes; /* all inodes */
01397:     struct hlist_head     s_anon;       /* anonymous dentries for (nfs) exporting */
01398:     struct list_head    s_files;
01399:     /* s_dentry_lru and s_nr_dentry_unused are protected by dcache_lock */
01400:     struct list_head    s_dentry_lru;     /* unused dentry lru */
01401:     int             s_nr_dentry_unused;     /* # of dentry on lru */
01402:
01403:     struct block_device    *s_bdev;
01404:     struct backing_dev_info *s_bdi;
01405:     struct mtd_info        *s_mtd;
01406:     struct list_head    s_instances;
```

```
01407:     struct quota_info    s_dquot;     /* Diskquota specific options */
01408:
01409:     int            s_frozen;
01410:     wait_queue_head_t     s_wait_unfrozen;
01411:
01412:     char s_id[32];                /* Informational name */
01413:
01414:     void              *s_fs_info;     /* Filesystem private info */
01415:     fmode_t              s_mode;
01416:
01417:     /*
01418:      * The next field is for VFS *only*. No filesystems have any business
01419:      * even looking at it. You had been warned.
01420:      */
01421:     struct mutex s_vfs_rename_mutex;       /* Kludge */
01422:
01423:     /* Granularity of c/m/atime in ns.
01424:        Cannot be worse than a second */
01425:     u32       s_time_gran;
01426:
01427:     /*
01428:      * Filesystem subtype.  If non-empty the filesystem type field
01429:      * in /proc/mounts will be "type.subtype"
01430:      */
01431:     char *s_subtype;
01432:
01433:     /*
01434:      * Saved mount options for lazy filesystems using
01435:      * generic_show_options()
01436:      */
01437:     char *s_options;
01438: } ? end super_block ? ;
01439:
```

创建、管理和销毁超级块对象的代码都在fs/super.c文件中。超级块对象通过alloc_super（）函数创建并初始化。在文件系统安装时，内核会调用该函数以便从磁盘读取文件系统超级块。

这里介绍一下super_block结构体中成员变量的含义：

s_list：指向超级块链表的指针；

s_dev：设备标识符；

s_blocksize：以字节为单位的块大小；

s_old_blocksize：以位为单位的旧的块大小；

s_blocksize_bits：以位为单位的块大小；

s_dirt：修改（脏）标志；

s_maxbytes：文件大小上限；

s_type：文件系统类型；

s_op：超级块方法；

dq_op：磁盘限额方法；

s_qcop：限额控制方法；

s_export_op：导出方法；

s_flags：登录标志；

s_magic：文件系统的魔数；

s_root：目录登录点；

s_umount：卸载信号量；

s_lock：超级块信号量；

s_count：超级块引用计数；

s_syncing：文件系统同步标志；

s_need_sync_fs：尚未同步标志；

s_active：活动引用计数；

s_security：安全模块；

s_dirty：脏节点链表；

s_io：回写链表；

s_anon：匿名目录项；

s_files：被分配文件链表；

s_bdev：相关的块设备；

s_instances：该类型文件系统；

s_dquot：限额相关选项；

s_id：文本名字；

s_fs_info：文件系统特殊信息；

s_vfs_rename_sem：重命名信号量；

s_time_gran：时间戳的粒度（单位为纳秒）。

2.2.1 超级块操作

超级块对象中最重要的一个域是s_op，它指向超级块的操作函数表。超级块操作函数表由super_operations结构体表示，定义在文件include/linux/fs.h中。

```
01611: struct super_operations {
01612:     struct inode *(*alloc_inode)(struct super_block *sb);
01613:     void (*destroy_inode)(struct inode *);
```

```
01614:
01615:      void (*dirty_inode) (struct inode *);
01616:      int (*write_inode) (struct inode *, struct writeback_control   *wbc);
01617:      void (*drop_inode) (struct inode *);
01618:      void (*delete_inode) (struct inode *);
01619:      void (*put_super) (struct super_block *);
01620:      void (*write_super) (struct super_block *);
01621:      int (*sync_fs)(struct super_block *sb, int wait);
01622:      int (*freeze_fs) (struct super_block *);
01623:      int (*unfreeze_fs) (struct super_block *);
01624:      int (*statfs) (struct dentry *, struct kstatfs *);
01625:      int (*remount_fs) (struct super_block *, int *, char *);
01626:      void (*clear_inode) (struct inode *);
01627:      void (*umount_begin)   (struct super_block *);
01628:
01629:      int (*show_options)(struct seq_file *, struct vfsmount *);
01630:      int (*show_stats)(struct   seq_file *, struct vfsmount *);
01631: #ifdef CONFIG_QUOTA
01632:      ssize_t (*quota_read)(struct   super_block *, int, char *, size_t, loff_t);
01633:      ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
01634: #endif
01635:      int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
01636: } ? end super_operations ? ;
01637:
```

该结构体中的每一项都是指向超级块操作函数的指针，超级块操作函数执行文件系统和索引节点的底层操作。

当文件系统需要对其超级块执行操作时，首先要在超级块对象中寻找需要的操作方法。比如一个文件系统要写自己的超级块，需要调用：

sb->s_op->write_super(sb)；

下面是super_operations中超级块操作函数的说明

alloc_inode(sb)

该方法在给定的超级块下创建并初始化一个新的索引节点对象。

destroy_inode(inode)

该函数用于释放给定的索引节点。

read_inode(inode)

该函数以inode->i_ino为索引，从磁盘上读取索引节点，并填充内存中对应的索引节点结构的剩余部分。

dirty_inode(inode)

VFS在索引节点脏时，会调用此函数。日志文件系统（如ext3）执行该函数进行日志更新。

write_inode(inode, wait)

该函数用于将给定的索引节点写入磁盘。wait参数指明写操作是否需要同步。

put_inode(inode)

该函数用于释放给定索引节点。

drop_inode(inode)

在最后一个指向索引节点的引用被释放后，VFS会调用该函数。VFS只要简单地删除这个索引节点后，普通Unix文件系统没有定义这个函数。注意调用者必须持有inode_lock锁。

delete_inode(inode)

该函数用户从磁盘上删除给定的索引节点。

put_super(sb)

在卸载文件系统时由VFS调用，用来释放超级块。

write_super(sb)

用给定的超级块更新磁盘上的超级块。VFS通过该函数对内存中的超级块和磁盘中的超级块进行同步。

sync_fs(sb, wait)

使文件系统的数据元与磁盘上的文件系统同步。wait参数指定操作是否同步。

write_super_lockfs(sb)

首先禁止对文件系统作改变，再使用给定的超级块更新磁盘上的超级块。目前LVM（逻

辑卷管理）会调用该函数。

unlockfs(sb)

对文件系统解除锁定，它是write_super_lockfs()的逆操作。

statfs(sb, statfs)

VFS通过调用该函数，获取文件系统状态。指定文件系统相关的统计信息放置在statfs中。

remount_fs(sb, flags, data)

当指定新的安装选项重新安装文件系统时，VFS会调用此函数。

clear_inode(inode)

VFS调用该函数释放索引节点，并清空包含相关数据的所有页面。

unmount_begin(sb)

VFS调用该函数中断安装操作。该函数被网络文件系统使用，如NFS。

所有以上函数都是由VFS在进程上下文中调用。必要时，它们都可以阻塞。这其中的一些函数是可选的：在超级块操作表中，文件系统可以将不需要的函数指针设置成NULL。如果VFS发现操作函数指针是NULL，那它要么就会调用通用函数执行相应操作，要么什么也不做，如何选择取决于具体函数。

2.3 索引节点对象

索引节点对象包含了内核在操作文件或目录时需要的全部信息。对于Unix文件系统来说，这些信息可以从磁盘索引节点直接读入。如果一个文件系统没有索引节点，那么，不管这些相关信息在磁盘上是怎么存放的，文件系统都必须从中提取这些信息。

索引节点对象由inode结构体表示，定义在文件include/linux/fs.h。

```
00766: struct inode {
00767:     struct hlist_node    i_hash;
00768:     struct list_head     i_list;      /* backing dev IO list */
00769:     struct list_head     i_sb_list;
00770:     struct list_head     i_dentry;
00771:     unsigned long        i_ino;
00772:     atomic_t             i_count;
00773:     unsigned int         i_nlink;
00774:     uid_t                i_uid;
00775:     gid_t                i_gid;
00776:     dev_t                i_rdev;
00777:     u64                  i_version;
00778:     loff_t               i_size;
00779: #ifdef __NEED_I_SIZE_ORDERED
00780:     seqcount_t           i_size_seqcount;
00781: #endif
00782:     struct timespec      i_atime;
00783:     struct timespec      i_mtime;
00784:     struct timespec      i_ctime;
00785:     blkcnt_t             i_blocks;
00786:     unsigned int         i_blkbits;
00787:     unsigned short       i_bytes;
00788:     umode_t              i_mode;
00789:     spinlock_t           i_lock;     /* i_blocks, i_bytes, maybe i_size */
00790:     struct mutex         i_mutex;
00791:     struct rw_semaphore  i_alloc_sem;
00792:     const struct inode_operations*i_op;
00793:     const struct file_operations    *i_fop;     /* former ->i_op->default_file_ops */
00794:     struct super_block     *i_sb;
00795:     struct file_lock     *i_flock;
00796:     struct address_space    *i_mapping;
00797:     struct address_space    i_data;
00798: #ifdef CONFIG_QUOTA
00799:     struct dquot         *i_dquot[MAXQUOTAS];
00800: #endif
00801:     struct list_head     i_devices;
00802:     union {
00803:         struct pipe_inode_info     *i_pipe;
00804:         struct block_device     *i_bdev;
00805:         struct cdev          *i_cdev;
00806:     };
00807:
00808:     __u32            i_generation;
00809:
00810: #ifdef CONFIG_FSNOTIFY
00811:     __u32            i_fsnotify_mask;   /* all events this inode cares about */
00812:     struct hlist_head    i_fsnotify_mark_entries; /* fsnotify mark entries */
00813: #endif
00814:
00815: #ifdef CONFIG_INOTIFY
00816:     struct list_head     inotify_watches;   /* watches on this inode */
00817:     struct mutex         inotify_mutex;      /* protects the watches list */
00818: #endif
00819:
00820:     unsigned long        i_state;
00821:     unsigned long        dirtied_when;     /* jiffies of first dirtying */
00822:
00823:     unsigned int         i_flags;
00824:
00825:     atomic_t             i_writecount;
00826: #ifdef CONFIG_SECURITY
00827:     void             *i_security;
00828: #endif
```

```
00829: #ifdef CONFIG_FS_POSIX_ACL
00830:     struct posix_acl    *i_acl;
00831:     struct posix_acl    *i_default_acl;
00832: #endif
00833:     void            *i_private; /* fs or device private pointer */
00834: } ? end inode ? ;
00835:
```

一个索引节点代表文件系统中的一个文件，它也可以是设备或管道这样的特殊文件。因此索引节点结构体中有一些和特殊文件相关的项，比如i_pipe项就指向一个代表命名管道的数据结构。如果索引节点并非代表一个命名管道，那么该项就被简单地设置为NULL。和特殊文件相关的项还有i_devices、i_bdev和i_cdev等。

inode结构体中各成员变量的含义如下：

i_hash：散列表；

i_list：索引节点链表；

i_dentry：目录项链表；

i_ino：节点号；

i_count：引用计数；

i_mode：访问权限控制；

i_nlink：硬连接数；

i_uid：使用者的id；

i_gid：使用者的组id；

i_rdev：实际设备标志符；

i_size：以字节为单位的文件大小；

i_atime：最后访问时间；

i_mtime：最后修改时间；

i_ctime：最后改变时间；

i_blkbits：以位为单位的块大小；

i_blksize：以字节为单位的块大小；

i_version：版本号；

i_blocks：文件的块数；

i_bytes：使用的字节数；

i_lock：自旋锁；

i_alloc_sem：嵌入在i_sem内部；

i_sem：索引节点信号量；

i_op：索引节点操作表；

i_fop：默认的索引节点操作；

i_sb：相关的超级块；

i_flock：文件锁链表；

i_mapping：相关的地址映射；

i_data：设备地址映射；

i_dquot：节点的磁盘限额；

i_devices：块设备链表；

i_pipe：管道信息；

i_bdev：块设备驱动；

i_dnotify_mask：目录通知掩码；

i_dnotify：目录通知；

i_state：状态标志；

dirtied_when：首次修改时间；

i_flags：文件系统标志；

i_writecount：写进程使用的计数；

i_security：指向inode节点的安全结构；

u.generic_ip：指向私有数据；

i_size_seqcount：SMP系统中使用的顺序计数，来获取i_size的一致值；

2.3.1 索引节点操作

和超级块操作一样，索引节点对象中的inode_operations项也非常重要，因为它描述了VFS用以操作索引节点对象的所有方法，这些方法由文件系统实现。与超级块类似，对索引节点的操作调用方式如下：

i->i_op->truncate(i)

i指向给定的索引节点，truncate（）函数是由索引节点i所在的文件系统提供操作。inode_operations结构体定义在文件include/linux/fs.h中。

```
01566: struct inode_operations {
01567:     int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
01568:     struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
01569:     int (*link) (struct dentry *,struct inode *,struct dentry *);
01570:     int (*unlink) (struct inode *,struct dentry *);
01571:     int (*symlink) (struct inode *,struct dentry *,const char *);
01572:     int (*mkdir) (struct inode *,struct dentry *,int);
01573:     int (*rmdir) (struct inode *,struct dentry *);
01574:     int (*mknod) (struct inode *,struct dentry *,int,dev_t);
01575:     int (*rename) (struct inode *, struct dentry *,
01576:             struct inode *, struct dentry *);
01577:     int (*readlink) (struct dentry *, char __user *,int);
01578:     void * (*follow_link) (struct dentry *, struct nameidata *);
01579:     void (*put_link) (struct dentry *, struct nameidata *, void *);
01580:     void (*truncate) (struct inode *);
01581:     int (*permission) (struct inode *, int);
01582:     int (*check_acl)(struct   inode *, int);
01583:     int (*setattr) (struct dentry *, struct iattr *);
01584:     int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
01585:     int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
01586:     ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
01587:     ssize_t (*listxattr) (struct dentry *, char *, size_t);
01588:     int (*removexattr) (struct dentry *, const char *);
01589:     void (*truncate_range)(struct inode *, loff_t, loff_t);
01590:     long (*fallocate)(struct inode *inode, int mode, loff_t offset,
01591:             loff_t len);
01592:     int (*fiemap)(struct inode *, struct fiemap_extent_info   *, u64 start,
01593:             u64 len);
01594: } ? end inode_operations ? ;
01595:
```

create(dir, dentry, mode, nameidata)

VFS通过系统调用create（）和open（）来调用该函数，从而为dentry对象创建一个新的索引节点。在创建时，必须使用mode指定的初始模式。

lookup(dir, dentry, nameidata)

在特定目录中寻找索引节点，该索引节点要对应于dentry中给出的文件名。

link(old_dentry, dir, new_dentry)

该函数被系统调用link（）调用，用来创建硬连接。硬连接名称由dentry参数指定，连接对象是dir目录中的old_dentry目录项所代表的文件。

unlink(dir, dentry)

该函数被unlink（）调用，从目录dir中删除由目录项dentry指定的索引节点对象。

symlink(dir, dentry, symname)

该函数由系统调用symlink（）调用，创建符号连接。该符号连接名由symname指定，连接对象是dir目录中的dentry目录项。

mkdir(dir, dentry, mode)

该函数由系统调用mkdir（）调用，创建一个新目录。创建时使用mode指定的初始模式。

rmdir(dir, dentry)

该函数由系统调用rmdir（）调用，删除dir目录下的dentry目录项代表的文件。

mknod(dir, dentry, mode, rdev)

该函数由系统调用mknod（）调用，创建特殊文件（设备文件、命名管道或套接字）。要创建的文件放在dir目录中，其目录项为dentry，关联的设备为rdev，初始权限由mode指定。

rename(old_dir, old_dentry, new_dir, new_dentry)

VFS调用该函数来移动文件。文件源路径在old_dir目录中，源文件由old_dentry目录所指定，目标路径在new_dir目录中，目标文件由new_dentry指定。

readlink(dentry, buffer, buflen)

该函数由系统调用readlink（）调用，拷贝数据到特定的缓冲buffer中。拷贝的数据来自dentry指定的符号连接，拷贝大小最大可达buflen字节。

follow_link(inode, nameidata)

由VFS调用，从一个符号连接查找它指向的索引节点。由dentry指向的连接被解析，其结果存放在由nd指向的nameidata结构中。

put_link(dentry, nameidata)

在follow_link（）调用之后，该函数由VFS调用进行清除工作。

truncate(inode)

由VFS调用，修改文件的大小。在调用前，索引节点的i_size项必须被设置为预期的大小。

permission(inode, mask, nameidata)

该函数用来检查给定的inode所代表的文件是否运行特定的访问模式。

setattr(dentry, iattr)

该函数被notify_change（）调用，在修改索引节点后，通知发生了“改变事件“。

getattr(mnt, dentry, kstat)

在通知索引节点需要从磁盘更新时，VFS会调用该函数。

setxattr(dentry, name, value, size, flags)

由VFS调用，给dentry指定的文件设置扩展属性。属性名为name，值为value。

getxattr(dentry, name, buffer, size)

向value中拷贝给定文件的扩展属性name对应的数值。

listxattr(dentry, buffer, size)

该函数将特定文件的所有属性列表拷贝到一个缓冲列表中。

removexattr(dentry, name)

该函数从给定文件中删除指定的属性。

2.4 目录项对象

VFS把目录当作文件对待，所以在路径/bin/ls，bin和ls都属于文件-bin是特殊的目录文件，而ls是一个普通文件，路径中的每个组成部分都由一个索引节点对象表示。虽然它们可以统一由索引节点表示，但VFS经常需要执行目录相关的操作，比如路径名查找等。路径名超找需要解析路径中的每一个组成部分，不但要确定它有效，而且还需要进一步寻找路径中的下一个部分。

为了方便查找，VFS引入目录项的概念。每个dentry代表路径中一个特定部分。对于/bin/ls来说，/、bin和ls都是目录项对象。前面是两个目录，最后一个是普通文件。在路径中，包括普通文件在内，每一个部分都是目录项对象。解析一个路径是一个耗时的、常规的字符串比较过程。

目录项对象由dentry结构体表示，定义在文件include/linux/dache.h中。

```
00089: struct dentry {
00090:     atomic_t d_count;
00091:     unsigned int d_flags;       /* protected by d_lock */
00092:     spinlock_t d_lock;          /* per dentry lock */
00093:     int d_mounted;              /* obsolete, ->d_flags is now used for this */
00094:     struct inode *d_inode;          /* Where the name belongs to - NULL is
00095:                                 * negative */
00096:     /*
00097:      * The next three fields are touched by ____d_lookup.   Place them here
00098:      * so they all fit in a cache line.
00099:      */
00100:     struct hlist_node d_hash;    /* lookup hash list */
00101:     struct dentry *d_parent;/* parent directory */
00102:     struct qstr d_name;
00103:
00104:     struct list_head d_lru;         /* LRU list */
00105:     /*
00106:      * d_child and d_rcu can share memory
00107:      */
00108:     union {
00109:         struct list_head d_child; /* child of parent list */
00110:         struct rcu_head d_rcu;
00111:     } d_u;
00112:     struct list_head d_subdirs;    /* our children */
00113:     struct list_head d_alias; /* inode alias list */
00114:     unsigned long d_time;           /* used by d_revalidate */
00115:     const struct dentry_operations   *d_op;
00116:     struct super_block *d_sb;    /* The root of the dentry tree */
00117:     void *d_fsdata;             /* fs-specific data */
00118:
00119:     unsigned char d_iname[DNAME_INLINE_LEN_MIN]; /* small names */
00120: } ? end dentry ? ;
00121:
```

不同于前面的两个对象，目录项对象没有对应的磁盘数据结构，VFS根据字符串形式的路径名临时创建它。而且由于目录项对象并非真正保存在磁盘上，所以目录项结构体没有是否被修改的标志。

dentry数据结构中，各成员变量含义如下：

d_count：使用计数；

d_vfs_flags：目录项缓存标志；

d_lock：单目录项锁；

d_inode：相关的索引节点；

d_lru：未用的链表；

d_child：父目录中目录项对象的链表；

d_subdirs：子目录；

d_alias：索引节点的别名链表；

d_time：重新生效时间；

d_op：目录项操作表；

d_sb：文件超级块；

d_flags：目录项标识；

d_fsdata：文件系统特殊的数据；

d_rcu：RCU锁；

d_cookie：cookie；

d_parent：父目录的目录项对象；

d_name：目录项的名字；

d_hash：散列表；

d_bucket：散列表头；

d_iname：短文件名。

2.4.1 目录项状态

目录项对象有三种有效状态：被使用、未被使用和负状态。

一个被使用的目录项对应一个有效的索引节点（即d_inode指向相应的索引节点），并且表明该对象存在一个或多个使用者（即d_count为正值）。一个目录项处于被使用状态，意味着它正被VFS使用并且指向有效的索引节点，因此不能被丢弃。

一个未被使用的目录项对应一个有效的索引节点（d_inode指向一个索引节点），但是应指明VFS当前并未使用它（d_count为0）。该目录项对象仍然指向一个有效对象，而且被保留在缓存中以便需要时再使用它。由于该目录项不会过早地被销毁，所以在以后再需要使用它时，不必重新创建，从而使路径查找更迅速。但如果要回收内存的话，可以销毁未使用的目录项。

一个负状态的目录项没有对用的有效索引节点（d_inode为NULL），因为索引节点已被删除，或路径不再正确，但是目录项仍然保留，以便快速解析以后的路径查询。虽然负状态的目录项有些用处，但是如果需要的话，可以销毁它，因为毕竟在实际中很少用到它。目

录项对象释放后也可以保存到slab对象缓存中去。

2.4.2 目录项缓存

如果VFS层遍历路径名中所有的元素，并将它们逐个地解析成目录项对象，这是非常耗时的工作。所以内核将目录项对象缓存在目录项缓存（即dcache）中。

目录项缓存包括三个主要部分：

（1）“被使用”目录项链表

该链表通过索引节点对象中的i_dentry项链接相关的索引节点，因为一个给定的索引节点可能有多个链接，所以就可能有多个目录项对象，因此用一个链表来连接它们。

（2）“最近被使用的”双向链表

该链表含由未被使用的和负状态的目录项对象。由于该链表以时间顺序插入，所以链头的节点是最新数据。当内核必须通过删除节点项回收内存时，会从链尾删除节点项。

（3）散列表和相应的散列函数用来快速将给定路径解析为相关目录项对象

散列表由数组dentry_hashtable表示，其中每一个元素都是一个指向具体相同键值的目录项对象的指针。数组的大小取决于系统中物理内存的大小。

实际的散列值由d_hash（）函数计算，它是内核提供给文件系统唯一的散列函数。

查找散列表要通过d_lookup（）函数，如果该函数在dcache中发现了与相匹配的目录项对象，则返回匹配的对象；否则返回NULL。

dache在一定意义上也提供对索引节点的缓存。和目录项对象相关的索引节点不会被释放，因为目录项会让相关索引节点的使用计数为正，这样就可以确保索引节点留在内存中。只要目录项被缓存，其相应的索引节点也就被缓存了。

2.4.3 目录项操作

dentry_operations结构体指明了VFS操作目录项的所有方法。该结构体定义在include/linux/dcache.h中

```
00134: struct dentry_operations {
00135:     int (*d_revalidate)(struct dentry *, struct nameidata *);
00136:     int (*d_hash) (struct dentry *, struct qstr *);
00137:     int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
00138:     int (*d_delete)(struct  dentry *);
00139:     void (*d_release)(struct   dentry *);
```

```
00140:      void (*d_iput)(struct dentry *, struct inode *);
00141:      char *(*d_dname)(struct   dentry *, char *, int);
00142: #ifndef __GENKSYMS__
00143:      struct vfsmount *(*d_automount)(struct path *);
00144:      int (*d_manage)(struct   dentry *, bool);
00145: #endif
00146: };
00147:
```

dentry_operations操作函数的含义如下：

d_revalidate(dentry, nameidata)

该函数判断目录对象是否有效。VFS准备从dcache中使用一个目录项时，会调用该函数。大部分文件系统将该方法设置为NULL，因为它们认为dcache中的目录项对象总是有效的。

d_hash(dentry, name)

该函数为目录项生成散列值，当目录项要加入散列表中时，VFS调用该函数。

d_compare(dir, name1, name2)

VFS调用该函数来比较name1和name2两个文件名。多数文件系统使用VFS的默认操作，仅仅做字符串比较。对于有些文件系统，比如FAT，简单的字符串比较不能满足其需要。因为FAT文件系统不区分大小写。

d_delete(dentry)

当目录项对象的d_count计数值等于0时，VFS调用该函数。注意使用该函数需要加dcache_lock锁。

d_release(dentry)

当目录项对象要被释放时，VFS调用该函数，默认情况下，它什么也不做。

d_iput(dentry, ino)

当一个目录项对象丢失了相关索引节点时，VFS调用该函数。默认情况下VFS会调用iput（）函数释放索引节点。如果文件系统重载了该函数，那么执行此文件系统特殊的工作外，还必须调用iput（）函数。

2.5 文件对象

VFS最后一个主要对象是文件对象。文件对象表示进程已打开的文件。如果我们站在用户空间的角度考虑VFS，文件对象会首先进入我们的视野。进程直接处理的是文件，而不是超级块、索引节点或目录项。文件对象包含我们非常熟悉的信息（如访问模式、当前偏移等），同样道理，文件操作和我们非常熟悉的系统调用read（）和write（）等也很类似。

文件对象是已打开的文件在内存中的表示。该对象（不是物理文件）由相应的open（）系统调用创建，由close（）系统调用销毁，所有这些文件相关的调用实际上都是文件操作表中定义的方法。因为多个进程可以同时打开和操作同一个文件，所以同一个文件也可能存在多个对应的文件对象。文件对象仅仅在进程观点上代表已打开文件，它反过来指向目录项对象（反过来指向索引节点），其实只有目录项对象才表示已打开的实际文件。虽然同一文件对应的文件对象不是唯一的，但对应的索引节点和目录项则是唯一的。

文件对象由file结构体表示，定义在文件include/linux/fs.h中

```
00961: struct file {
00962:     /*
00963:      * fu_list becomes invalid after file_free is called and queued via
00964:      * fu_rcuhead for RCU freeing
00965:      */
00966:     union {
00967:         struct list_head    fu_list;
00968:         struct rcu_head     fu_rcuhead;
00969:     } f_u;
00970:     struct path         f_path;
00971: #define f_dentry    f_path.dentry
00972: #define f_vfsmnt f_path.mnt
00973:     const struct file_operations    *f_op;
00974:     spinlock_t      f_lock;     /* f_ep_links, f_flags, no IRQ */
00975:     atomic_long_t       f_count;
00976:     unsigned int        f_flags;
00977:     fmode_t             f_mode;
00978:     loff_t          f_pos;
00979:     struct fown_struct      f_owner;
00980:     const struct cred       *f_cred;
00981:     struct file_ra_state    f_ra;
00982:
00983:     u64         f_version;
00984: #ifdef CONFIG_SECURITY
00985:     void            *f_security;
00986: #endif
00987:     /* needed for tty driver, and maybe others */
00988:     void            *private_data;
00989:
00990: #ifdef CONFIG_EPOLL
00991:     /* Used by fs/eventpoll.c to link all the hooks to this file */
00992:     struct list_head    f_ep_links;
00993: #endif /* #ifdef CONFIG_EPOLL */
00994:     struct address_space    *f_mapping;
00995: #ifdef CONFIG_DEBUG_WRITECOUNT
00996:     unsigned long f_mnt_write_state;
00997: #endif
```

```
00998: } ? end file ? ;
```

类似于目录项对象，文件对象实际上没有对应的磁盘数据。所以结构体中没有代表其对象是否为脏，是否需要回写磁盘的标志。文件对象通过f_dentry指针指向相关的目录项对象。目录项会指向相关的索引节点，索引节点会记录文件是否脏的。

file数据结构各成员变量的含义如下：

f_list：文件对象链表；

f_dentry：相关目录项对象；

f_vfsmnt：相关的安装文件系统；

f_op：文件操作表；

f_count：文件对象的使用计数；

f_flags：当打开文件时所指定的标志；

f_mode：文件的访问模式；

f_pos：文件当前的位移量；

f_owner：通过信号进行异步I/O数据的传送；

f_uid：用户的UID；

f_gid：用户的GID；

f_error：错误码；

f_ra：预读状态；

f_version：版本号；

f_security：安全模块；

private_data：tty设备hook；

f_ep_links：事件池链表；

f_ep_lock：事件池锁；

f_mapping：页缓存映射。

2.5.1 文件操作

和VFS的其他对象一样，文件操作在文件对象中也非常重要。和file结构体相关的操作与系统调用很类似，这些操作是标准Unix系统调用的基础。

文件对象的操作由file_operations结构体表示，定义在文件include/linux/fs.h。

```
01537: struct file_operations {
01538:     struct module *owner;
01539:     loff_t (*llseek) (struct file *, loff_t, int);
01540:     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
01541:     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
01542:     ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
01543:     ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
01544:     int (*readdir) (struct file *, void *, filldir_t);
01545:     unsigned int (*poll) (struct file *, struct poll_table_struct *);
01546:     int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
01547:     long (*unlocked_ioctl)   (struct file *, unsigned int, unsigned long);
01548:     long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
01549:     int (*mmap) (struct file *, struct vm_area_struct *);
01550:     int (*open) (struct inode *, struct file *);
01551:     int (*flush) (struct file *, fl_owner_t id);
01552:     int (*release) (struct inode *, struct file *);
01553:     int (*fsync) (struct file *, struct dentry *, int datasync);
01554:     int (*aio_fsync) (struct kiocb *, int datasync);
01555:     int (*fasync) (int, struct file *, int);
01556:     int (*lock) (struct file *, int, struct file_lock *);
01557:     ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
01558:     unsigned long (*get_unmapped_area)(struct file *, unsigned long,
01558:                unsigned long, unsigned long, unsigned long);
01559:     int (*check_flags)(int);
01560:     int (*flock) (struct file *, int, struct file_lock *);
01561:     ssize_t (*splice_write)(struct pipe_inode_info  *, struct file *, loff_t *, size_t,
                unsigned int);
01562:     ssize_t (*splice_read)(struct   file *, loff_t *, struct pipe_inode_info  *, size_t,
                unsigned int);
01563:     int (*setlease)(struct  file *, long, struct file_lock **);
01564: } ? end file_operations ? ;
01565:
```

具体的文件系统可以为每一种操作做专门的实现，如果存在通用操作，也可以使用通用操作。一般在基于Unix的文件系统上，这些通用操作效果都不错。并不要求实际文件系统实现文件操作函数表中的所有方法，对不需要的操作可以将该函数指针设置为NULL。

对于file_operations中的函数，我们在C语言中经常使用，这里不多作介绍。

3 与文件系统相关的数据结构

除了以上几种VFS基础对象外，内核还使用了另外一些标准数据结构来管理文件系统的其他相关数据结构。第一个结构体是file_system_type，用来描述各种特定文件系统类型，比如ext2和JFFS2。第二个结构体是vfsmount，用来描述一个安装文件系统的实例。

因为Linux支持多种文件系统，所以内核必须由一个特殊的结构来描述每种文件系统的功能和行为，file_system_type结构体定义在include/linux/fs.h中

```
01795: struct file_system_type {
01796:     const char *name;
01797:     int fs_flags;
```

```
01798:      int (*get_sb) (struct file_system_type *, int,
01799:                     const char *, void *, struct vfsmount *);
01800:      void (*kill_sb) (struct super_block *);
01801:      struct module *owner;
01802:      struct file_system_type * next;
01803:      struct list_head fs_supers;
01804:
01805:      struct lock_class_key s_lock_key;
01806:      struct lock_class_key s_umount_key;
01807:
01808:      struct lock_class_key i_lock_key;
01809:      struct lock_class_key i_mutex_key;
01810:      struct lock_class_key i_mutex_dir_key;
01811:      struct lock_class_key i_alloc_sem_key;
01812: };
01813:
```

get_sb（）函数从磁盘上读取超级块，并且在文件系统被安装时，在内存中组装超级块对象。剩余的函数描述文件系统的属性。

每种文件系统，不管多少个实例安装到系统中，还是根本没有安装到系统中，都只有一个file_system_type结构。

当文件系统被实际安装时，将有一个vfsmount结构体在安装点被创建。该结构体用来代表文件系统的实例，即代表一个安装点。

vfsmount结构被定义在include/linux/mount.h中：

```
00039: struct vfsmount {
00040:      struct list_head mnt_hash;
00041:      struct vfsmount *mnt_parent;      /* fs we are mounted on */
00042:      struct dentry *mnt_mountpoint;        /* dentry of mountpoint */
00043:      struct dentry *mnt_root;      /* root of the mounted tree */
00044:      struct super_block *mnt_sb;       /* pointer to superblock */
00045:      struct list_head mnt_mounts;      /* list of children, anchored here */
00046:      struct list_head mnt_child;    /* and going through their mnt_child */
00047:      int mnt_flags;
00048:      __u32 rh_reserved;        /* for use with fanotify */
00049:      struct hlist_head rh_reserved2;       /* for use with fanotify */
00050:      const char *mnt_devname;      /* Name of device e.g. /dev/dsk/hda1 */
00051:      struct list_head mnt_list;
00052:      struct list_head mnt_expire;      /* link in fs-specific expiry list */
00053:      struct list_head mnt_share; /* circular list of shared mounts */
00054:      struct list_head mnt_slave_list;/*    list of slave mounts */
00055:      struct list_head mnt_slave;    /* slave list entry */
00056:      struct vfsmount *mnt_master;      /* slave is on master->mnt_slave_list */
00057:      struct mnt_namespace *mnt_ns;      /* containing namespace */
00058:      int mnt_id;                /* mount identifier */
00059:      int mnt_group_id;          /* peer group identifier */
00060:      /*
00061:       * We put mnt_count & mnt_expiry_mark at the end of struct vfsmount
00062:       * to let these frequently modified fields in a separate cache line
00063:     * (so that reads of mnt_flags wont ping-pong on SMP machines)
00064:     */
```

理清文件系统和所有其他安装点间的关系，是维护所有安装点链表中最复杂的工作。所以vfsmount结构体中维护的各种链表就是为了能够跟踪这些关系信息。

vfsmount结构还保存了在安装时指定的标志信息，该信息存储在mnt_flags域中。表1列出了标准的安装标志。

表1 标准挂载标志表

Flag	Description
MNT_NOSUID	Forbids setuid and setgid flags on binaries on this filesystem
MNT_NODEV	Forbids access to device files on this filesystem
MNT_NOEXEC	Forbids execution of binaries on this filesystem

安装那些管理员不充分信任的移动设备时，这些标志很有用处。

4 和进程相关的数据结构

系统中的每一个进程都有自己的一组打开的文件，如根文件系统、当前工作目录、安装点等。有三个数据结构将VFS层和系统的进程紧密联系在一起，它们分别是：files_struct、fs_struct和namespace。

4.1 files_struct

files_struct结构体定义在文件include/linux/file.h中。该结构体由进程描述符中的files域指向。所有与每个进程相关的信息，如打开的文件及文件描述符都包含在其中。

```
00040: /*
00041:  * Open file table structure
00042:  */
00043: struct files_struct {
00044:     /*
00045:      * read mostly part
00046:      */
00047:     atomic_t count;
00048:     struct fdtable *fdt;
00049:     struct fdtable fdtab;
00050:     /*
00051:      * written part on a separate cache line in SMP
00052:      */
00053:     spinlock_t file_lock ____cacheline_aligned_in_smp;
00054:     int next_fd;
00055:     struct embedded_fd_set close_on_exec_init;
00056:     struct embedded_fd_set open_fds_init;
00057:     struct file * fd_array[NR_OPEN_DEFAULT];
00058: };
```

fd数组指针指向已打开的文件对象链表，默认情况下，指向fd_array数组。因为

NR_OPEN_DEFAULT等于32，所以该数组可以容纳32个文件对象。如果一个进程所打开的文件对象超过32个，内核将分配一个新数组，并且将fd指针指向它。所以对适当数量的文件对象访问会执行得很快，因为它是对对静态数组的操作；如果一个进程打开的文件数量过多，那么内核就需要建立新数组。如果系统中由大量的进程都要打开超过32个文件，为了优化性能，管理员可以适当增大NR_OPEN_DEFAULT的预定义值。

files_struct结构体中各成员变量的含义如下：

count：结构的使用计数；

file_lock：保护该结构体的锁；

max_fds：文件对象数的上限；

max_fdset：文件描述符的上限；

next_fd：下一个文件描述符；

fd：全部文件对象数组；

close_on_exec：exec（）关闭的文件描述符；

open_fds：指向打开文件的描述符；

close_on_exec_init：exec（）关闭的初始文件；

open_fd_init：文件描述符的初始集合；

fd_array：默认的文件对象数组。

4.2 fs_struct

和进程相关的第二个结构体是fs_struct。该结构由进程描述符的fs域指向。它包含文件系统和进程相关的信息，定义在文件include/linux/fs_struct.h中，

```
00006: struct fs_struct {
00007:     int users;
00008:     rwlock_t lock;
00009:     int umask;
00010:     int in_exec;
00011:     struct path root, pwd;
00012: };
```

该结构体包含了当前进程的当前工作目录（pwd）和根目录。

fs_struct结构体中有六个指针。前三个是dentry结构指针，就是root、pwd和altroot。这些指针各自指向代表着一个“目录项“的dentry数据结构，里面记录着文件的各项属性，如文件名、访问权限等等。其中pwd则指向进程当前所在的目录；而root所指向的dentry结构代表着本进程的“根目录”，那就是当用户登录进入系统时所“看到“的根目录；至于altroot

则为用户设置的“替换根目录”。实际运行时这三个目录不一定在同一个文件系统中。所以后三个指针就各自指向代表着这些“安装”的vfsmount数据结构。注意：fs_struct结构中的信息都是与文件系统和进程相关的，带有全局性的，而与具体的已打开的文件没有什么关系。