# 一、虚拟文件系统概述

虚拟文件系统VFS（也成虚拟文件交换）作为内核子系统，为用户空间程序提供了文件和文件系统相关的统一接口。通过VFS，应用程序可以使用相同接口完成不同介质上不同文件系统的数据读写操作，如图示例。



VFS是内核对所有实际文件系统（ext2\ext3\vfat\nfs等）操作接口的一层通用封装（故称虚拟）。通过这个抽象层，应用程序调用相同接口完成不同文件系统之间的数据操作，而底层可以支持各种文件系统，也可以轻松的新增文件系统而不会对应用程序有任何影响。

VFS之所以能够衔接各种各样的文件系统，是因为它定义了所有文件系统都支持的、基本的、概念上的接口和数据结构。实际文件系统也将自身的操作在形式上与VFS定义保持一致，而每个文件系统的细节则由各自实现。如下两个框图:

 

左图为VFS下用户操作设备的通用模型，右图为write操作的一个简单例子。

# 二、相关数据结构描述

VFS采用面向对象的设计思路，但是内核只有C语言，因此使用结构体方式实现即包含数据的同时又包含操作这些数据的函数指针。

VFS中有四个主要的对象类型：

A．超级块对象：代表一个具体的已安装文件系统（存在物理介质）

B．索引节点对象：代表一个具体的文件（存在物理介质）

C．目录项对象：代表一个目录项，是路径的一个组成部分（存在内存）

D．文件对象：代码由进程打开的文件（存在内存）

每个主要对象中都包含一个操作对象，这些操作对象描述了内核针对主要对象可以使用的方法：

A．super\_operations对象：包含内核针对特定文件系统所能调用的方法

B．inode\_operations对象：包含内核对特定文件所能调用的方法

C．dentry\_operations对象：包含内核对特定目录所能调用的方法

D．file\_operations对象：包含进程针对已打开文件所能调用的方法

## 1、超级块对象

各种文件系统都必须实现超级块对象，该对象用于存储特定文件系统的信息，通常对应于存放在磁盘特定扇区的文件系统超级块或文件系统控制块。对于并非基于磁盘的文件系统（基于内存的文件系统sysfs proc等），它们会在使用现场创建超级块并将其保存到内存中。

超级块对象由include/linux/fs.h中定义，其细节如下：

struct super\_block {

struct list\_head s\_list; /\* 指向所有超级块的链表 \*/

dev\_t s\_dev; /\* 设备标识符 \*/

unsigned char s\_blocksize\_bits; /\* 以位为单位的块大小 \*/

unsigned long s\_blocksize; /\* 以字节为单位的块大小 \*/

loff\_t s\_maxbytes; /\* 文件大小上限 \*/

struct file\_system\_type \*s\_type; /\* 文件系统类型 \*/

const struct super\_operations \*s\_op; /\* 超级块操作方法 \*/

const struct dquot\_operations \*dq\_op; /\* 磁盘限额操作方法 \*/

const struct quotactl\_ops \*s\_qcop; /\* 限额控制方法 \*/

const struct export\_operations \*s\_export\_op; /\* 导出方法 \*/

unsigned long s\_flags; /\* 挂载标志 \*/

unsigned long s\_magic; /\* 文件系统的幻数 \*/

struct dentry \*s\_root; /\* 目录挂载点 \*/

struct rw\_semaphore s\_umount; /\* 卸载信号量 \*/

int s\_count; /\* 超级块引用计数 \*/

atomic\_t s\_active; /\* 活动引用计数 \*/

const struct xattr\_handler \*\*s\_xattr; /\* 扩展的模块属性 \*/

struct list\_head s\_inodes; /\* inodes链表 \*/

struct hlist\_bl\_head s\_anon; /\* 匿名目录项 \*/

struct list\_head \_\_percpu \*s\_files; /\* 被分配文件链表，percpu变量 \*/

struct list\_head s\_mounts; /\* \*/

struct list\_head s\_dentry\_lru; /\* 未被使用目录项链表 \*/

int s\_nr\_dentry\_unused; /\* 链表中目录项的数目 \*/

spinlock\_t s\_inode\_lru\_lock \_\_\_\_cacheline\_aligned\_in\_smp;

struct list\_head s\_inode\_lru; /\* 未被使用inode链表 \*/

int s\_nr\_inodes\_unused; /\* 链表中inodes的数目 \*/

struct block\_device \*s\_bdev; /\* 相关块设备 \*/

struct backing\_dev\_info \*s\_bdi; /\* \*/

struct mtd\_info \*s\_mtd; /\* 存储磁盘信息 \*/

struct hlist\_node s\_instances; /\* 该类型文件系统 \*/

struct quota\_info s\_dquot; /\* 限额相关选项 \*/

struct sb\_writers s\_writers; /\* \*/

char s\_id[32]; /\* 文本名字 \*/

u8 s\_uuid[16]; /\* \*/

void \*s\_fs\_info; /\* 文件系统特殊信息 \*/

unsigned int s\_max\_links; /\* \*/

fmode\_t s\_mode; /\* 安装权限 \*/

u32 s\_time\_gran; /\* 时间戳粒度 \*/

struct mutex s\_vfs\_rename\_mutex; /\* 重命名信号量 \*/

char \*s\_subtype; /\* 子类型名称 \*/

char \_\_rcu \*s\_options; /\* 已安装选项 \*/

const struct dentry\_operations \*s\_d\_op; /\* \*/

int cleancache\_poolid;

struct shrinker s\_shrink; /\* \*/

atomic\_long\_t s\_remove\_count;

int s\_readonly\_remount;

};

创建、管理和撤销超级块对象的代码位于文件fs/super.c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，文件系统会调用该函数以便从磁盘读取文件系统超级块，并且将其信息填充到内存中的超级块对象中。

在上面一个重要结构体成员，红色部分s\_op，它指向超级块的操作函数表。其结构体定义在linux/fs.h中，每一个项的超级块操作函数执行文件系统和索引节点的底层操作，细节如下：

struct super\_operations {

struct inode \*(\*alloc\_inode)(struct super\_block \*sb); //申请和初始化一个新的索引节点对象

void (\*destroy\_inode)(struct inode \*); //释放索引节点

void (\*dirty\_inode) (struct inode \*, int flags); //VFS在索引节点脏（被修改）时会调用此函数

int (\*write\_inode) (struct inode \*, struct writeback\_control \*wbc); //将给定的索引节点写入磁盘，写入控制方式

int (\*drop\_inode) (struct 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

void (\*evict\_inode) (struct inode \*); //从icache中删除节点

void (\*put\_super) (struct super\_block \*); //卸载文件系统时，VFS调用它释放超级块

int (\*sync\_fs)(struct super\_block \*sb, int wait); //使文件系统的数据元与磁盘上的文件系统同步

int (\*freeze\_fs) (struct super\_block \*);

int (\*unfreeze\_fs) (struct super\_block \*);

int (\*statfs) (struct dentry \*, struct kstatfs \*); //获取文件系统状态

int (\*remount\_fs) (struct super\_block \*, int \*, char \*); //重新安装文件系统

void (\*umount\_begin) (struct super\_block \*); //网络文件系统（如NFS）使用，中断安装操作。

int (\*show\_options)(struct seq\_file \*, struct dentry \*);

int (\*show\_devname)(struct seq\_file \*, struct dentry \*);

int (\*show\_path)(struct seq\_file \*, struct dentry \*);

int (\*show\_stats)(struct seq\_file \*, struct dentry \*);

#ifdef CONFIG\_QUOTA

ssize\_t (\*quota\_read)(struct super\_block \*, int, char \*, size\_t, loff\_t);

ssize\_t (\*quota\_write)(struct super\_block \*, int, const char \*, size\_t, loff\_t);

#endif

int (\*bdev\_try\_to\_free\_page)(struct super\_block\*, struct page\*, gfp\_t);

int (\*nr\_cached\_objects)(struct super\_block \*);

void (\*free\_cached\_objects)(struct super\_block \*, int);

};

所有以上函数都是由VFS在进程上下文中调用。

## 2、索引节点对象

索引节点包含了内核在操作文件或目录时需要的全部信息。对于unix风格的文件系统，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，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信息，填充索引节点对象。一个索引节点对象代表文件系统中的一个文件，但是只有索引节点被访问时才在内存中创建，也可以是设备或管道文件的特殊文件。索引节点对象定义在linux/fs.h中，细节如下：

struct inode {

umode\_t i\_mode; //

unsigned short i\_opflags;

kuid\_t i\_uid; //使用者ID

kgid\_t i\_gid; //使用组ID

unsigned int i\_flags; //文件系统标志

const struct inode\_operations \*i\_op; //索引节点操作表

struct super\_block \*i\_sb; //相关的超级块

struct address\_space \*i\_mapping; //相关地址映射

unsigned long i\_ino; //节点号

union {

const unsigned int i\_nlink; //硬连接数

unsigned int \_\_i\_nlink;

};

dev\_t i\_rdev; //实际设备标识符

loff\_t i\_size; //以字节为单位的文件大小

struct timespec i\_atime; //最后访问时间

struct timespec i\_mtime; //最后修改时间

struct timespec i\_ctime; //最后改变时间

spinlock\_t i\_lock;

unsigned short i\_bytes; //使用的字节数

unsigned int i\_blkbits; //以位为单位的块大小

blkcnt\_t i\_blocks; //文件的块数

/\* Misc \*/

unsigned long i\_state; //状态标志

struct mutex i\_mutex;

unsigned long dirtied\_when; /\* 第一次脏数据时间，单位jiffies \*/

struct hlist\_node i\_hash; //散列表

struct list\_head i\_wb\_list; /\* backing dev IO list \*/

struct list\_head i\_lru; /\* inode LRU list \*/

struct list\_head i\_sb\_list; //超级块链表

union {

struct hlist\_head i\_dentry;

struct rcu\_head i\_rcu;

};

u64 i\_version;

atomic\_t i\_count; //引用计数

atomic\_t i\_dio\_count; //

atomic\_t i\_writecount;

const struct file\_operations \*i\_fop; /\* 缺省的索引节点操作 former ->i\_op->default\_file\_ops \*/

struct file\_lock \*i\_flock; //文件锁链表

struct address\_space i\_data; //设备地址映射

struct dquot \*i\_dquot[MAXQUOTAS]; //索引节点的磁盘限额

struct list\_head i\_devices; //块设备链表

union {

struct pipe\_inode\_info \*i\_pipe; //管道

struct block\_device \*i\_bdev; //块设备

struct cdev \*i\_cdev; //字符设备

};

\_\_u32 i\_generation;

void \*i\_private; /\* fs or device private pointer \*/

};

同样，索引节点对象中的i\_op是inode的所有操作项，其结构定义在linux/fs.h中，细节（按照cache对齐）如下：

struct inode\_operations {

struct dentry \* (\*lookup) (struct inode \*,struct dentry \*, unsigned int); //在特定目录中寻找节点

void \* (\*follow\_link) (struct dentry \*, struct nameidata \*); //从一个符号连接查找它指向的索引节点。

int (\*permission) (struct inode \*, int); //检查给定的inode所代表的文件是否允许特定的访问模式。

struct posix\_acl \* (\*get\_acl)(struct inode \*, int);

int (\*readlink) (struct dentry \*, char \_\_user \*,int); //拷贝数据到特定的缓冲buffer中，数据来自dentry指定的符号连接

void (\*put\_link) (struct dentry \*, struct nameidata \*, void \*); //清理工作

int (\*create) (struct inode \*,struct dentry \*, umode\_t, bool); //create或open调用该函数，创建一个新的索引节点。

int (\*link) (struct dentry \*,struct inode \*,struct dentry \*); //用来创建硬件连接

int (\*unlink) (struct inode \*,struct dentry \*); //从目录dir中删除由目录项dentry指定的索引节点对象

int (\*symlink) (struct inode \*,struct dentry \*,const char \*); //创建符号连接

int (\*mkdir) (struct inode \*,struct dentry \*,umode\_t); //创建目录

int (\*rmdir) (struct inode \*,struct dentry \*); //删除目录

int (\*mknod) (struct inode \*,struct dentry \*,umode\_t,dev\_t); //创建特殊文件

int (\*rename) (struct inode \*, struct dentry \*, //修改目录名

struct inode \*, struct dentry \*);

int (\*setattr) (struct dentry \*, struct iattr \*);

int (\*getattr) (struct vfsmount \*mnt, struct dentry \*, struct kstat \*);

int (\*setxattr) (struct dentry \*, const char \*,const void \*,size\_t,int);

ssize\_t (\*getxattr) (struct dentry \*, const char \*, void \*, size\_t);

ssize\_t (\*listxattr) (struct dentry \*, char \*, size\_t);

int (\*removexattr) (struct dentry \*, const char \*);

int (\*fiemap)(struct inode \*, struct fiemap\_extent\_info \*, u64 start,

u64 len);

int (\*update\_time)(struct inode \*, struct timespec \*, int);

int (\*atomic\_open)(struct inode \*, struct dentry \*,

struct file \*, unsigned open\_flag,

umode\_t create\_mode, int \*opened);

} \_\_\_\_cacheline\_aligned;

## 3、目录项对象

VFS把目录当作文件对待，所以在路径/bin/vi中，bin和vi都属于文件——bin是特殊的目录文件而vi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。为了方便查找操作，VFS引入目录项的概念。每个dentry代表路径中的一个特定部分。对前一个例子来说：/、bin和vi都属于目录项对象，前两个是目录，最后一个是普通文件。在路径中，每个部分（包括普通文件在内）都是目录项对象。

注意点：目录项没有对应的磁盘数据结构，VFS根据字符串形式的路径名创建在内存中，因此没有被修改（是否为脏，是否需要写回磁盘的标志）的标志。

目录项对象定义在文件<linux/dcache.h>中。下面给出该结构体和其中各项的描述详细：

struct dentry {

unsigned int d\_flags; /\* 目录项标识 protected by d\_lock \*/

seqcount\_t d\_seq; /\* 每个目录项的顺序锁 \*/

struct hlist\_bl\_node d\_hash; /\* 查找散列表lookup hash list \*/

struct dentry \*d\_parent; /\* 父目录的目录对象parent directory \*/

struct qstr d\_name; /\* 目录项名称 \*/

struct inode \*d\_inode; /\* 相关联的索引节点 \*/

unsigned char d\_iname[DNAME\_INLINE\_LEN]; /\* 短文件名 \*/

unsigned int d\_count; /\* 使用记数protected by d\_lock \*/

spinlock\_t d\_lock; /\* 目录项锁per dentry lock \*/

const struct dentry\_operations \*d\_op; /\* 目录项操作指针 \*/

struct super\_block \*d\_sb; /\* 目录树对应的超级块The root of the dentry tree \*/

unsigned long d\_time; /\* 目录重置时间used by d\_revalidate \*/

void \*d\_fsdata; /\* 文件特有数据fs-specific data \*/

struct list\_head d\_lru; /\* 最近最少使用链表LRU list \*/

/\*

\* d\_child and d\_rcu can share memory

\*/

union {

struct list\_head d\_child; /\* 目录项内部形成的链表 \*/

struct rcu\_head d\_rcu; /\* RCU加锁 \*/

} d\_u;

struct list\_head d\_subdirs; /\* 子目录链表our children \*/

struct hlist\_node d\_alias; /\* 索引节点别名链表inode alias list \*/

};

（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。

A．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）。

B．一个未被使用的目录项对应一个有效的索引节点（即d\_inode指向一个索引节点），但是应该指明VFS当前并未使用它（d\_count为0）。保留在缓存中，方便后续使用。

C．一个负状态的目录项没有对应的有效索引节点（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。

（2）目录项缓存

如果VFS遍历路径名中所有的元素并将它们逐个的解析成目录项对象，还要到达最深层目录，将是一件非常费力的工作，所以内核将目录对象缓存起来。目录项缓存分为三个部分：

A．“被使用的”目录项链表：通过索引节点对象中的成员i\_dentry项连接相关的索引节点

B．“最近被使用的”双向链表：含有未被使用的和负状态的目录项对象，总是在头部插入项目项，所以链头节点的数据总比链尾的数据要新。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。

C．散列表和相应的散列函数用快速地将给定路径解析为相关目录项对象。

只要目录项被缓存，其相应的索引节点也被缓存了；只要路径名在缓存中找到了，那么相应的索引节点肯定也在内存中缓存着。

（3）目录项操作

目录项操作函数定义在<include/linux/dcache.h>，其细节如下：

struct dentry\_operations {

int (\*d\_revalidate)(struct dentry \*, unsigned int); //判断目录项对象是否有效

int (\*d\_weak\_revalidate)(struct dentry \*, unsigned int); //

int (\*d\_hash)(const struct dentry \*, const struct inode \*, struct qstr \*); //该函数为目录项生成散列表，当目录项需要加入到散列表中时，VFS调用该函数

int (\*d\_compare)(const struct dentry \*, const struct inode \*,

const struct dentry \*, const struct inode \*,

unsigned int, const char \*, const struct qstr \*); //VFS调用该函数比较name1和name2这两个文件名

int (\*d\_delete)(const struct dentry \*); //当目录项对象的d\_count计数值等于0时，VFS调用该函数

void (\*d\_release)(struct dentry \*); //当目录项对象将要被释放时调用时，默认情况下它什么都不做。

void (\*d\_prune)(struct dentry \*); //

void (\*d\_iput)(struct dentry \*, struct inode \*); //当一个目录项对象丢失了其相关的索引节点时，调用该函数。

char \*(\*d\_dname)(struct dentry \*, char \*, int);

struct vfsmount \*(\*d\_automount)(struct path \*);

int (\*d\_manage)(struct dentry \*, bool);

} \_\_\_\_cacheline\_aligned;

## 4、文件对象

文件对象是进程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()系统调用撤销，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。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，它反过来指向目录对象（目录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。虽然一个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对象无疑是唯一的。

注意点：文件对象没有对应的磁盘数据。文件对象通过f\_path指针指向相关的目录项对象，目录项对象指向相关的索引节点，索引节点会记录文件是否脏。

文件对象结构体定义在<include/linux/fs.h>，细节如下：

struct file {

/\*

\* fu\_list becomes invalid after file\_free is called and queued via

\* fu\_rcuhead for RCU freeing

\*/

union {

struct list\_head fu\_list; //文件对象链表

struct rcu\_head fu\_rcuhead; //释放无效之后变成RCU链表

} f\_u;

struct path f\_path; //包含目录项

#define f\_dentry f\_path.dentry

struct inode \*f\_inode; /\* cached value \*/

const struct file\_operations \*f\_op; //文件操作表

/\*

\* Protects f\_ep\_links, f\_flags, f\_pos vs i\_size in lseek SEEK\_CUR.

\* Must not be taken from IRQ context.

\*/

spinlock\_t f\_lock; //单个文件结构锁

#ifdef CONFIG\_SMP

int f\_sb\_list\_cpu;

#endif

atomic\_long\_t f\_count; //文件对象的使用计数

unsigned int f\_flags; //当打开文件时所指定的标志

fmode\_t f\_mode; //文件的访问模式

loff\_t f\_pos; //文件当前的位移量（文件指针）

struct fown\_struct f\_owner; //拥有者通过信号进行I/O数据的传送

const struct cred \*f\_cred; //文件的信任状

struct file\_ra\_state f\_ra; //预读装填

u64 f\_version; //版本号

#ifdef CONFIG\_SECURITY

void \*f\_security;

#endif

/\* needed for tty driver, and maybe others \*/

void \*private\_data; //tty设备驱动的钩子，见MMZ例子存放私有数据结构

#ifdef CONFIG\_EPOLL

/\* Used by fs/eventpoll.c to link all the hooks to this file \*/

struct list\_head f\_ep\_links; //事件池链表

struct list\_head f\_tfile\_llink; //

#endif /\* #ifdef CONFIG\_EPOLL \*/

struct address\_space \*f\_mapping; //页缓存映射

#ifdef CONFIG\_DEBUG\_WRITECOUNT

unsigned long f\_mnt\_write\_state; //调试状态

#endif

};

文件操作列表在文件对象中非常重要。文件对象的操作有file\_operations结构体表示，在文件<linux/fs>中定义：

struct file\_operations {

struct module \*owner;

//更新偏移量指针，由系统调用llseek调用它

loff\_t (\*llseek) (struct file \*, loff\_t, int);

//从指定文件的offset偏移处读取count字节数据到buf中，同时更新文件指针，由系统调用read调用它。

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);

//将buf的count个字节数据写入到文件的offset偏移处，同时更新文件指针，由系统调用write调用它。

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);

//从iocb描述的文件里以同步方式读取数据，并将其写入由vector描述的count个缓冲区中去，同时增加文件的偏移量，由aio\_read调用它。

ssize\_t (\*aio\_read) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);

//以同步方式将vector描述的count个缓冲区中数据写入iocb描述的文件里，同时减小文件的偏移量，由aio\_write调用它。

ssize\_t (\*aio\_write) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);

//返回目录列表中的下一个目录，由系统调用readdir()调用它

int (\*readdir) (struct file \*, void \*, filldir\_t);

//睡眠等待给定文件活动，由系统调用poll()调用它

unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);

//与ioctl功能类似：给设备发送命令参数，当文件是一个被打开的设备节点时，可以通过它进行设置操作，由系统调用ioctl调用它。与ioctl不同点：不需要持有BKL(大内核锁)。

long (\*unlocked\_ioctl) (struct file \*, unsigned int, unsigned long);

//ioctl函数的可移植变种，被32位应用程序用在64位系统上。

long (\*compat\_ioctl) (struct file \*, unsigned int, unsigned long);

//将给定的文件映射到指定的地址空间上，由系统调用mmap调用它

int (\*mmap) (struct file \*, struct vm\_area\_struct \*);

//打开或创建一个新的文件对象，并将它和相应的索引节点对象关联起来，由系统调用open()调用它。

int (\*open) (struct inode \*, struct file \*);

//当已打开文件的引用计数减少时被调用，具体作用根据文件系统而定

int (\*flush) (struct file \*, fl\_owner\_t id);

//当文件的最后一个引用被注销时，该函数会被VFS调用。作用根据具体文件系统而定。

int (\*release) (struct inode \*, struct file \*);

//将给定文件所有被缓存数据写回磁盘，由系统调用fsync()调用它

int (\*fsync) (struct file \*, loff\_t, loff\_t, int datasync);

//将iocb描述的文件的所有被缓存数据写回磁盘，由系统调用aio\_fsync()调用它。

int (\*aio\_fsync) (struct kiocb \*, int datasync);

//用于打开或关闭异步IO的信号

int (\*fasync) (int, struct file \*, int);

//用于给指定文件上锁

int (\*lock) (struct file \*, int, struct file\_lock \*);

//从一个文件向另外一个文件发送数据

ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);

//获取未使用的地址空间来映射给定的文件

unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);

//检查flag，目前只有NFS使用，因为NFS不允许O\_APPEND和O\_DIRECT相结合

int (\*check\_flags)(int);

int (\*flock) (struct file \*, int, struct file\_lock \*);

ssize\_t (\*splice\_write)(struct pipe\_inode\_info \*, struct file \*, loff\_t \*, size\_t, unsigned int);

ssize\_t (\*splice\_read)(struct file \*, loff\_t \*, struct pipe\_inode\_info \*, size\_t, unsigned int);

int (\*setlease)(struct file \*, long, struct file\_lock \*\*);

long (\*fallocate)(struct file \*file, int mode, loff\_t offset,

loff\_t len);

int (\*show\_fdinfo)(struct seq\_file \*m, struct file \*f);

};

## 5、与文件系统相关数据结构

以上讲述了VFS的几个基础对象，除此之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：

1. file\_system\_type：描述各种特定文件类型（ext2/ext3等）
2. vfsmount：描述一个安装文件系统的实例

（1）file\_system\_type

内核通过file\_system\_type结构体来描述每种文件系统的功能和行为，以便支持众多不同的文件系统。其结构体在<linux/fs.h>中具体定义如下：

struct file\_system\_type {

const char \*name; //文件系统的名字

int fs\_flags; //文件系统类型标志

#define FS\_REQUIRES\_DEV 1

#define FS\_BINARY\_MOUNTDATA 2

#define FS\_HAS\_SUBTYPE 4

#define FS\_USERNS\_MOUNT 8 /\* Can be mounted by userns root \*/

#define FS\_USERNS\_DEV\_MOUNT 16 /\* A userns mount does not imply MNT\_NODEV \*/

#define FS\_RENAME\_DOES\_D\_MOVE 32768 /\* FS will handle d\_move() during rename() internally. \*/

//挂载安装文件系统时，获取相应的超级块信息

struct dentry \*(\*mount) (struct file\_system\_type \*, int,

const char \*, void \*);

//卸载后终止访问超级块

void (\*kill\_sb) (struct super\_block \*);

struct module \*owner; //文件系统模块

struct file\_system\_type \* next; //链表中下一个文件系统类型

struct hlist\_head fs\_supers; //超级块对象链表：每种文件系统，不管有多少个实例安装到系统中还是根本就没有安装到系统中，都只有一个file\_system\_type结构

struct lock\_class\_key s\_lock\_key;

struct lock\_class\_key s\_umount\_key;

struct lock\_class\_key s\_vfs\_rename\_key;

struct lock\_class\_key s\_writers\_key[SB\_FREEZE\_LEVELS];

struct lock\_class\_key i\_lock\_key;

struct lock\_class\_key i\_mutex\_key;

struct lock\_class\_key i\_mutex\_dir\_key;

};

（2）vfsmount

当文件系统被实际安装时（即挂载），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（相应挂载点），在< linux/mount.h >中定义：

struct vfsmount {

struct dentry \*mnt\_root; /\* root of the mounted tree 该文件系统的根目录项\*/

struct super\_block \*mnt\_sb; /\* pointer to superblock 该文件系统的超级块\*/

int mnt\_flags; //安装标志

};

## 6、与进程相关的数据结构

系统中每一个进程都有自己的一组打开的文件，像根文件系统、当前工作目录、安装点等。三个数据结构将VFS层和系统的进程紧密联系在一起，分别是file\_struct、fs\_struct和namespace结构体。在task\_struct中定义了相应的成员：

struct task\_struct {

…

/\* filesystem information \*/

struct fs\_struct \*fs;

/\* open file information \*/

struct files\_struct \*files;

/\* namespaces \*/

struct nsproxy \*nsproxy;

…

}

（1）fs\_struct

该结构由进程描述符中的fs成员指向，包含了与该进程相关的所有文件系统信息，在<linux/fs\_struct.h>定义：

struct fs\_struct {

int users; //用户数目

spinlock\_t lock; //保护该结构体的锁

seqcount\_t seq; //

int umask; //掩码

int in\_exec; //当前正在执行的文件

struct path root, pwd; //根目录路径和当前工作目录的路径

};

该结构体包含了当前进程的当前工作目录pwd和根目录。

（2）file\_struct

该结构体由进程描述符中的files成员指向，包含了与该进程相关的所有打开文件信息。在< linux/fdtable.h >定义：

struct files\_struct {

/\*

\* read mostly part

\*/

atomic\_t count; //结构体使用计数

struct fdtable \_\_rcu \*fdt; //指向其他fd表的指针

struct fdtable fdtab; //基fd表

/\*

\* written part on a separate cache line in SMP

\*/

spinlock\_t file\_lock \_\_\_\_cacheline\_aligned\_in\_smp; //单个文件锁

int next\_fd; //缓存下一个可用的fd

unsigned long close\_on\_exec\_init[1]; //exec()时关闭的文件描述符链表

unsigned long open\_fds\_init[1]; //打开的文件描述符链表

struct file \_\_rcu \* fd\_array[NR\_OPEN\_DEFAULT]; //缺省的文件对象数组

};

fd\_array数组指针指向已打开的文件对象，#define NR\_OPEN\_DEFAULT BITS\_PER\_LONG，在32位机器体系结构中这个宏的值为32，所以该数组可以容纳32个文件对象。如果进程打开的文件对象超过32个，内核将分配一个新数组并且由fdt指针指向它。

（3）namespace

该结构体由进程描述符中的nsproxy成员指向，包含了与进程相关的所有命名空间信息，在<>中定义：

struct nsproxy {

atomic\_t count;

struct uts\_namespace \*uts\_ns;

struct ipc\_namespace \*ipc\_ns;

struct mnt\_namespace \*mnt\_ns;

struct pid\_namespace \*pid\_ns;

struct net \*net\_ns;

};

struct mnt\_namespace {

atomic\_t count; //结构体使用计数

unsigned int proc\_inum;

struct mount \* root; //根目录的安装点对象

struct list\_head list; //安装点链表

struct user\_namespace \*user\_ns; //用户命名空间

u64 seq; /\* Sequence number to prevent loops \*/

wait\_queue\_head\_t poll;

int event;

};

默认情况下，所有的进程共享同样的命名空间（即从相同的挂载表中看到同一个文件系统层次结构）。只有在进程clone()操作时使用CLONE\_NEWS标志，才会给进程一个唯一的命名空间结构体的拷贝。因为大多数进程不提供这个标志，所有进程都继承其父进程的命名空间。因此，在大多数系统上只有一个命名空间。

## 7、关系图

### （1）前面几个数结构关系

前面描述了超级块、索引节点、目录项、文件等数据结构，他们之间的关系如图所示：



图中包含连个文件系统fs1和fs2，并且fs2的文件系统是挂载在fs1的dir2目录下。注册文件系统后，相应的文件系统就会放到全局链表中，链表头为file\_systems。安装或挂载文件系统实例后，超级块也会放到全局链表中，链表头为super\_blocks。当挂载一个文件系统时，实际就是创建挂载点（vfsmount）、超级块（super\_block）、目录项（dentry）、索引节点（inode）的过程，因此这个四个数据结构的地位很重要。

当进程打开文件时（如：task1打开file2或task2打开file1），会创建相依的file文件对象数据结构并指向相应的索引节点inode，这样就可以通过file来访问VFS的文件系统。

### （2）EXT文件系统磁盘布局

下图展示了EXT每个组块的分布图和相应的数据结构信息：



# 三、实例代码分析

## 1、文件系统初始化

Linux的文件系统初始化一般分为三个阶段：

A．vfs\_caches\_init负责挂载rootfs文件系统，并创建根挂载点目录：’/’

B．rest\_init()负责加载initrd文件，扩展VFS树，创建基本的文件系统目录拓扑

C．init程序负责挂载磁盘文件系统，并将文件系统的根目录从rootfs切换到磁盘文件系统

如图所示：



在嵌入式上linux的文件系统初始化分为两个阶段（后续代码主要讲述该方式）：

A．vfs\_caches\_init负责挂载rootfs文件系统，并创建根挂载点目录：’/’

B．rest\_init()将文件系统的根目录从rootfs切换到flash的mtd文件系统

如图所示：



通过mount命令查看挂载：

rootfs on / type rootfs (rw) //对应阶段A

/dev/root on / type squashfs (ro,relatime) //对应阶段B

lrwxrwxrwx 1 root root 9 Jan 1 00:00 /dev/root -> mtdblock4

简而言之：先将rootfs内存文件系统挂载到’/’目录，再将mtdblock4的squashfs文件系统挂载到rootfs文件系统下的/root目录，然后将/root目录替换roofs成为最终的’/’目录（实现方式见第二小节）

### （1）挂载rootfs文件系统

vfs\_caches\_init函数完成rootfs文件系统的挂载即如上mount查看到的：

rootfs on / type rootfs (rw)，其源码如下：

void \_\_init vfs\_caches\_init(unsigned long mempages)

{

unsigned long reserve;

/\* Base hash sizes on available memory, with a reserve equal to

150% of current kernel size \*/

reserve = min((mempages - nr\_free\_pages()) \* 3/2, mempages - 1);

mempages -= reserve;

names\_cachep = kmem\_cache\_create("names\_cache", PATH\_MAX, 0,

SLAB\_HWCACHE\_ALIGN|SLAB\_PANIC, NULL);

dcache\_init(); //目录项散列表初始化

inode\_init(); //索引节点散列表初始化

/\*根据总内存计算系统可以支持的最大文件数，计算方法：

一个文件需要分配索引节点和目录项内存粗略算1K，所有文件不超过中内存的10%

/proc/sys/fs/file-max就可以查看这个最大文件数

而ulimit –a则是当前进程支持的最大文件数

\*/

files\_init(mempages);

mnt\_init(); //注册sysfs文件系统和挂载rootfs文件系统

bdev\_cache\_init(); //完成bdev\_inode缓存申请，bdev文件系统的注册和挂载。内核用bdev文件系统管理设备索引节点，并在/dev中显示

chrdev\_init(); //完成字符设备散列表的初始化，散列表以主设备号为索引。

}

mnt\_init()函数是与VFS相关初始化的主要函数，其实现如下：

void \_\_init mnt\_init(void)

{

unsigned u;

int err;

init\_rwsem(&namespace\_sem);

//初始化mnt缓存

mnt\_cache = kmem\_cache\_create("mnt\_cache", sizeof(struct mount),

0, SLAB\_HWCACHE\_ALIGN | SLAB\_PANIC, NULL);

//申请挂载和挂载点的散列表

mount\_hashtable = (struct list\_head \*)\_\_get\_free\_page(GFP\_ATOMIC);

mountpoint\_hashtable = (struct list\_head \*)\_\_get\_free\_page(GFP\_ATOMIC);

if (!mount\_hashtable || !mountpoint\_hashtable)

panic("Failed to allocate mount hash table\n");

printk(KERN\_INFO "Mount-cache hash table entries: %lu\n", HASH\_SIZE);

for (u = 0; u < HASH\_SIZE; u++)

INIT\_LIST\_HEAD(&mount\_hashtable[u]);

for (u = 0; u < HASH\_SIZE; u++)

INIT\_LIST\_HEAD(&mountpoint\_hashtable[u]);

br\_lock\_init(&vfsmount\_lock);

/\*\*\*\*\*\*\*\*\*\*\*\*

static struct file\_system\_type sysfs\_fs\_type = {

.name = "sysfs",

.mount = sysfs\_mount,

.kill\_sb = sysfs\_kill\_sb,

.fs\_flags = FS\_USERNS\_MOUNT,

};

A．该函数调用register\_filesystem注册sysfs文件系统sysfs\_fs\_type，并加入到全局单链表file\_systems中

B．kern\_mount()—>vfs\_kern\_mount()完成struct mount挂载点创建和初始化并将挂载点的挂载项保存到全局变量sysfs\_mnt全局变量；调用sysfs\_fs\_type成员sysfs\_mount完成超级块、根目录”/”、根目录索引节点等数据结构的创建和初始化，将超级块添加到全局单链表super\_blocks中，把索引节点添加到inode\_hashtable和超级块的ionde链表中。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

err = sysfs\_init();

if (err)

printk(KERN\_WARNING "%s: sysfs\_init error: %d\n",

\_\_func\_\_, err);

/\* 在sysfs文件系统中创建fs目录，是一级目录所有第二个参数为NULL，即/sys/fs这个目录\*/

fs\_kobj = kobject\_create\_and\_add("fs", NULL);

if (!fs\_kobj)

printk(KERN\_WARNING "%s: kobj create error\n", \_\_func\_\_);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*

static struct file\_system\_type rootfs\_fs\_type = {

.name = "rootfs",

.mount = rootfs\_mount,

.kill\_sb = kill\_litter\_super,

};

该函数调用register\_filesystem注册根文件系统rootfs\_fs\_type，并加入到全局单链表file\_systems中

\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

init\_rootfs();

/\*\*\*\*\*\*\*\*\*\*\*\*\*

A．vfs\_kern\_mount完成挂载，过程同上

B．create\_mnt\_ns创建命名空间并设置该命名空间的挂载点为rootfs的挂载点，同时将rootfs的挂载点链接到该命名空间的双向链表中。

C．设置init\_task的命名空间，同时设置当前任务（即init\_task）的当前目录和根目录为rootfs的根目录”/”。

\*\*\*\*\*\*\*\*\*\*\*\*\*/

init\_mount\_tree();

}

上述函数执行完后，完成了sysfs和rootfs文件系统的挂载，但是只有rootfs处于init\_task进程的命名空间中，并且进程的fs\_struct数据结构的root和pwd都指向rootfs的根目录“/”，所以用户实际使用的是rootfs文件系统。rootfs为VFS提供了”/”根目录，所以文件操作和文件系统的挂载操作都可以在VFS上进行。其关系图如图所示：



### （2）rootfs切换到mtd文件系统

在kernel\_init()这个1号进程中完成rootfs切换到flash的mtd文件系统（本文使用squashfs）。kernel\_init()调用kernel\_init\_freeable()完成这个步骤：

static noinline void \_\_init kernel\_init\_freeable(void)

{

…

/\*

该函数完成各种编译进内核的模块加载，包括驱动、文件系统等

do\_initcalls完成各种配置的文件系统的注册，其中rootfs\_initcall(populate\_rootfs)对于三个阶段的完成initrd加载，而对于嵌入式二个阶段的啥也不做。

\*/

do\_basic\_setup();

…

if (sys\_access((const char \_\_user \*) ramdisk\_execute\_command, 0) != 0) {

ramdisk\_execute\_command = NULL;

prepare\_namespace(); //该函数完成了rootfs的切换

}

…

}

prepare\_namespace()函数的具体实现如下：

void \_\_init prepare\_namespace(void)

{

…

/\*

static int \_\_init root\_dev\_setup(char \*line)

{

strlcpy(saved\_root\_name, line, sizeof(saved\_root\_name));

return 1;

}

\_\_setup("root=", root\_dev\_setup);

saved\_root\_name由内核启动参数root=决定（这里为/dev/mtdblock4）

\*/

if (saved\_root\_name[0]) {

root\_device\_name = saved\_root\_name;

…

ROOT\_DEV = name\_to\_dev\_t(root\_device\_name);

if (strncmp(root\_device\_name, "/dev/", 5) == 0)

root\_device\_name += 5;

}

…

mount\_root(); //完成mtd新根文件系统挂载

out:

devtmpfs\_mount("dev"); //挂载dev的tmpfs

/\*

mount\_root将mtd设备文件挂载到/root目录作为新根文件系统，以下两步就是将新根文件系统替换rootfs，使其成为linux VFS的根目录。

mount命令展示：

rootfs on / type rootfs (rw)

/dev/root on / type squashfs (ro,relatime)

lrwxrwxrwx 1 root root 9 Jan 1 00:00 /dev/root -> mtdblock4

简而言之：先将rootfs内存文件系统挂载到’/’目录，再将mtdblock4的squashfs文件系统挂载到rootfs文件系统下的/root目录，然后将/root目录替换roofs成为最终的’/’目录（下面就是替换方法，”.”就是/root目录）

\*/

sys\_mount(".", "/", NULL, MS\_MOVE, NULL);

sys\_chroot(".");

}

mount\_root函数的具体实现如下：

void \_\_init mount\_root(void)

{

/\*

创建/dev/root设备节点，并软连接到mtd

lrwxrwxrwx 1 root root 9 Jan 1 00:00 /dev/root -> mtdblock4

\*/

create\_dev("/dev/root", ROOT\_DEV);

/\*

get\_fs\_names：获取mtd文件系统，通过\_\_setup("rootfstype=", fs\_names\_setup)内核参数指定，也可以有系统自动搜寻匹配。

do\_mount\_root：将mtd设备挂载到/root目录并cd到该新文件系统根目录

root\_mountflags：一般只读的挂载方式

\*/

mount\_block\_root("/dev/root", root\_mountflags);

}

## 2、挂载设备

这里主要分析mount命令的系统调用，以“mount –t vfat /dev/sda /home”的挂载过程为例进程分析。其执行过程如下：



（1）用户执行mount

当用户执行mount –t vfat /dev/sda /home时，标准C库通过swi触发软中断从用户态切换到内核态，进入mount的系统调用sys\_mount，细节见《linux内核之系统调用》一文。

（2）sys\_mount

该函数为mount的系统调用入口（在 fs/namespace.c中定义），会将参数从用户空间拷贝到内核空间，添加打印各参数打印如下：

kernel\_dev:/dev/sda, kernel\_dir:/home/, type:vfat, flags:0x8000

（3）do\_mount

该函数完成挂载的所有工作，调用kern\_path根据kernel\_dir查找挂载点，再根据flags选择合适的挂载方式，上面属于普通磁盘挂载会调用常用的do\_new\_mount。

（4）kern\_path

该函数主要根据kernel\_dir（即/home）路径查找dentry目录对象（即挂载点），将挂载点存放在struct path path中返回。调用do\_path\_lookup完成，其实现过程有三种情形，细节见下一节分析。

（5）do\_new\_mount

该函数完成最终的挂载，主要调用get\_fs\_type、vfs\_kern\_mount、do\_add\_mount。

（6）get\_fs\_type

该函数根据type(即vfat)，获取vfat注册到内核的file\_system\_type，如果内核没有注册即不支持该type则返回出错。在mount未指定具体的type（mount /dev/sda /home）时，busybox的mount源码会根据/proc/filesystems循环测试各个type，每测试一个type都是一次系统调用。

（7）vfs\_kern\_mount

该函数主要建立vfsmount对象和superblock对象。通过mount\_fs实现superblock对象建立，回调到实际file\_system\_type的mount实例，这里是vfat的mount，定义如下：

static struct file\_system\_type vfat\_fs\_type = {

.owner = THIS\_MODULE,

.name = "vfat",

/\* vfat\_mount调用mount\_bdev，必要时从设备上读取相应数据，填充superblock对象

\*/

.mount = vfat\_mount,

.kill\_sb = kill\_block\_super,

.fs\_flags = FS\_REQUIRES\_DEV,

};

（8）do\_add\_mount

将vfsmount加入到系统mount数和相应的hash\_table中，用步骤（4）的path替换原有的dentry目录对象，这样系统的目录内容就是挂载后的。它查找

## 3、路径查找三种情形

上节讲到寻找kernel\_dir挂载点路径是通过do\_path\_lookup函数完成，该函数主要由filename\_lookup—> path\_lookupat实现（sys\_open则通过path\_openat函数实现，路径查找部分类似）。其查找主要按照三种情形操作：

情况一：路径名在单个文件系统

情况二：路径名在多个文件系统

情况三：路径名在挂载点重复挂载多文件系统

下面以查找/mnt/mtd/ test.txt为例说明各种情况：

（1）情况一：路径名在单个文件系统

从根目录或当前目录开始，根据目录拓扑结构递归查找父目录。

如图所示，路径名只在单个文件系统squashfs中，递归查找父目录完成后，使用nd.path记录父目录mtd的位置,nd.last记录字符常量“test.txt”。nd是struct nameidata数据结构变量。



（2）情况二：路径名在多个文件系统

在查找时需要从当前文件系统的挂载点切换到最终文件系统的目录拓扑结构中，然后继续查找。

如图所示，该种情况路径名横跨了两种文件系统，vfat挂载在/mnt目录上，同时设置mnt为已挂载状态。路径查找到mnt目录时发现为已挂载状态，就从mnt切换到vfat的根目录，在vfat文件系统中继续查找路径，完成后保存到nd变量。



（3）情况三：路径名在挂载点重复挂载多文件系统

在查找时需要从当前文件系统的挂载点切换到最终文件系统的目录拓扑结构中，然后继续查找。

Linux支持同一挂载点挂载多个文件系统的操作，但是只有最后被挂载的文件系统才可见。mnt目录挂载了两个文件系统vfat和nfs；当成功挂载vfat后，设置挂载点mnt为已挂载状态；当挂载nfs时，发现mnt已挂载就从mnt切换到vfat根目录，将nfs挂载到vfat根目录并设置已挂载状态。

如图所示，在查找/mnt/mtd/ test.txt时，发现mnt已挂载状态，就从mnt切换到vfat根目录，而vfat根目录也为挂载状态，继续切换到nfs根目录；nfs根目录为非挂装状态，就在nfs文件系统中继续查找路径，完成后记录在nd变量。



## 4、文件操作

Linux下一切皆文件，将各种文件划分为：常规文件、目录文件、软连接文件、硬连接文件、特殊文件（设备文件、管道文件、sockt等），这些文件对应不同的系统操作函数：sys\_open()、sys\_mkdir()、sys\_link()、sys\_mknod()。

这里主要介绍open对应系统调用sys\_open代码，它根据flags对路径文件进行操作：简单打开、创建、清除、追加等。sys\_open在fs/open.c中定义，其关键函数调用如下：



get\_unused\_fd\_flags：从当前进程中申请一个新的文件描述符fd

do\_filp\_open：调用path\_openat()实现，完成路径搜索、文件操作（创建或打开等），返回相应file结构

fd\_install：将fd和file建立关联，用户就可以通过fd操作file文件。

get\_empty\_filp：从filp cache中申请一个file结构体

path\_init：主要是设置nd变量（struct nameidata结构变量），初始化检索的起始目录，判断起始目录是根目录还是当前目录，为link\_path\_walk查找路径做好准备

link\_path\_walk：关键字符串解析处理函数，分析解析字符串然后查找目录项hashtable找到下一级目录的inode节点，直到找到待操作文件目录。

do\_last：创建或者获取文件对应的inode对象并初始化file对象，这样就完成了该进程的一个文件对象。

下图描述了使用sys\_open新建常规文件的简单过程：



步骤1：get\_unused\_fd\_flags新建文件描述符

步骤2：get\_empty\_filp从cache申请file结构体

步骤3：do\_last()->lookup\_open()->lookup\_dcache()创建常规文件的目录项

步骤4：do\_last()->lookup\_open()->vfs\_create()创建常规文件的inode节点

步骤5：fd\_install将文件描述符指向file结构体，与inode建立关系

本文主要是记录linux虚拟文件系统部分的学习，比较浅显，参考了书籍和附录A的一些网站。

# 附录A

参考网站：

<http://blog.csdn.net/heikefangxian23/article/details/51579971>

http://blog.csdn.net/luomoweilan/article/details/17850377

http://blog.chinaunix.net/uid-31390099-id-5756239.html

http://blog.csdn.net/new\_abc/article/details/7685681

http://blog.chinaunix.net/uid-20522771-id-4419678.html