struct file\_system\_type {

const char \*name;

int fs\_flags;

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

const char \*, void \*);

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

struct module \*owner;

struct file\_system\_type \* next;

struct hlist\_head fs\_supers;

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 i\_lock\_key;

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

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

};

name：文件系统的名称，是一个字符串，如reiserfs，ext4；

fs\_flags：

mount：

kill\_sb：

owner：

next：

fs\_supers：对每一个已经mount的文件系统，内存中都对应一个超级块，同一类型的文件系统可能对应多个分区，比如root分区和home分区都是ext4文件系统，则mount后都链接到fs\_supers为表头的链表中；

struct vfsmount {

struct dentry \*mnt\_root; /\* root of the mounted tree \*/

struct super\_block \*mnt\_sb; /\* pointer to superblock \*/

int mnt\_flags;

};

每个被安装的文件系统都对应一个mount结构。

struct mount {

struct list\_head mnt\_hash;

struct mount \*mnt\_parent;

struct dentry \*mnt\_mountpoint;

struct vfsmount mnt;

#ifdef CONFIG\_SMP

struct mnt\_pcp \_\_percpu \*mnt\_pcp;

atomic\_t mnt\_longterm; /\* how many of the refs are longterm \*/

#else

int mnt\_count;

int mnt\_writers;

#endif

struct list\_head mnt\_mounts; /\* list of children, anchored here \*/

struct list\_head mnt\_child; /\* and going through their mnt\_child \*/

struct list\_head mnt\_instance; /\* mount instance on sb->s\_mounts \*/

const char \*mnt\_devname; /\* Name of device e.g. /dev/dsk/hda1 \*/

struct list\_head mnt\_list;

struct list\_head mnt\_expire; /\* link in fs-specific expiry list \*/

struct list\_head mnt\_share; /\* circular list of shared mounts \*/

struct list\_head mnt\_slave\_list;/\* list of slave mounts \*/

struct list\_head mnt\_slave; /\* slave list entry \*/

struct mount \*mnt\_master; /\* slave is on master->mnt\_slave\_list \*/

struct mnt\_namespace \*mnt\_ns; /\* containing namespace \*/

#ifdef CONFIG\_FSNOTIFY

struct hlist\_head mnt\_fsnotify\_marks;

\_\_u32 mnt\_fsnotify\_mask;

#endif

int mnt\_id; /\* mount identifier \*/

int mnt\_group\_id; /\* peer group identifier \*/

int mnt\_expiry\_mark; /\* true if marked for expiry \*/

int mnt\_pinned;

int mnt\_ghosts;

}

mnt\_hash：所有被安装的文件系统使用了一个全局链表mount\_hashtable，以mnt\_hash链接到该链表中。哈希值与mount实例的地址和dentry结构的地址有关；

mnt\_parent：指向父文件系统的mount结构；

mnt\_mountpoint：是当前被安装文件系统的安装点在父目录中的dentry结构；

mnt：被安装文件系统的根目录信息，是一个vfsmount结构，被安装文件系统的根目录保存在mnt\_root中，超级块保存在mnt\_sb中，对被安装文件系统而言，一个文件系统只对应一个超级块实例，mnt\_flags设置通用的文件系统标志，与特定的文件系统无关。比如，如果没有具体的设备，则设置MNT\_NODEV标志；

mount\_count：计数器，当一个mount实例不再需要时，调用mntput将计数器减1，需要获取一个mount实例时，调用mntget，计数器会加1；

mnt\_mounts：子文件系统链表的表头；

mnt\_child：通过该字段链接到子文件系统链表；

mnt\_instance：

mnt\_devname：

mnt\_list：

mnt\_expire：所有可能自动过期的安装通过mnt\_expire链接到一个链表上；

mnt\_share：所有共享安装通过mnt\_share链接到一个双向循环链表；

mnt\_slave\_list：

mnt\_slave：

mnt\_master：mnt\_slave、mnt\_slave\_list和mnt\_master被用来实现从属安装（slave mount），主安装（master mount）将所有从属安装通过mnt\_slave链接到mnt\_slave\_list为表头的链表上，所有的从属安装通过mnt\_master指向其主安装；

mnt\_ns：被安装文件系统所属的命名空间；

mnt\_expiry\_mark：表示被安装的文件系统是否已经不再使用，取值为0或者1；

struct super\_block {

struct list\_head s\_list; /\* Keep this first \*/

dev\_t s\_dev; /\* search index; \_not\_ kdev\_t \*/

unsigned char s\_dirt;

unsigned char s\_blocksize\_bits;

unsigned long s\_blocksize;

loff\_t s\_maxbytes; /\* Max file size \*/

struct file\_system\_type \*s\_type;

const struct super\_operations \*s\_op;

const struct dquot\_operations \*dq\_op;

const struct quotactl\_ops \*s\_qcop;

const struct export\_operations \*s\_export\_op;

unsigned long s\_flags;

unsigned long s\_magic;

struct dentry \*s\_root;

struct rw\_semaphore s\_umount;

struct mutex s\_lock;

int s\_count;

atomic\_t s\_active;

#ifdef CONFIG\_SECURITY

void \*s\_security;

#endif

const struct xattr\_handler \*\*s\_xattr;

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

struct hlist\_bl\_head s\_anon; /\* anonymous dentries for (nfs) exporting \*/

#ifdef CONFIG\_SMP

struct list\_head \_\_percpu \*s\_files;

#else

struct list\_head s\_files;

#endif

struct list\_head s\_mounts; /\* list of mounts; \_not\_ for fs use \*/

/\* s\_dentry\_lru, s\_nr\_dentry\_unused protected by dcache.c lru locks \*/

struct list\_head s\_dentry\_lru; /\* unused dentry lru \*/

int s\_nr\_dentry\_unused; /\* # of dentry on lru \*/

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

struct list\_head s\_inode\_lru; /\* unused inode lru \*/

int s\_nr\_inodes\_unused; /\* # of inodes on lru \*/

struct block\_device \*s\_bdev;

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

struct mtd\_info \*s\_mtd;

struct hlist\_node s\_instances;

struct quota\_info s\_dquot; /\* Diskquota specific options \*/

int s\_frozen;

wait\_queue\_head\_t s\_wait\_unfrozen;

char s\_id[32]; /\* Informational name \*/

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

void \*s\_fs\_info; /\* Filesystem private info \*/

fmode\_t s\_mode;

u32 s\_time\_gran;

struct mutex s\_vfs\_rename\_mutex; /\* Kludge \*/

char \*s\_subtype;

char \_\_rcu \*s\_options;

const struct dentry\_operations \*s\_d\_op; /\* default d\_op for dentries \*/

int cleancache\_poolid;

struct shrinker s\_shrink; /\* per-sb shrinker handle \*/

atomic\_long\_t s\_remove\_count;

int s\_readonly\_remount;

};

s\_list：系统中所有的超级块都通过该成员，链接到全局链表super\_blocks上；

s\_dev：文件系统的设备号，包括了主设备号和次设备号；

s\_dirt：如果超级块被改变，则置为1，表示需要回写到磁盘；

s\_blocksize\_bits：

s\_blocksize：指定了文件系统的块大小，比如标准的ext2文件系统的标准块大小是1024字节，s\_blocksize则为1024，表示字节数，s\_blocksize\_bits则为10（210=1024）

s\_maxbytes：文件系统支持的最大文件长度，根据具体文件系统实现而定；

s\_type：指向具体的file\_system\_type实例；

s\_op：超级块的相关操作函数集，具体文件系统有其特定的实现。比如，对于ecryptfs文件系统为ecryptfs\_sops；对于ext4文件系统为ext4\_sops，对于ceph文件系统为ceph\_super\_ops；

dq\_op：

s\_qcop：

s\_export\_op：

s\_flags：

s\_magic：

s\_root：

s\_umount：

s\_lock：

s\_count：

s\_active：

s\_security：

s\_xattr：xattr\_handler结构指针，包含了处理扩展属性的函数指针；

s\_inodes：

s\_anon：

s\_files：指向file结构的双向链表，表示该超级块对应的文件系统打开的所有文件，卸载文件系统时会检查该链表，如果卸载时仍包含写打开的文件，则卸载会失败；

s\_mounts：

s\_dentry\_lru：

s\_nr\_dentry\_unused：

s\_inode\_lru\_lock：

s\_inode\_lru：

s\_nr\_inodes\_unused：

s\_bdev：文件系统所在的块设备block\_device结构；

s\_bdi：

s\_mtd：

s\_instances：通过该成员链接到同一类型文件系统的file\_system\_type结构中的fs\_supers链表上；

s\_dquot：

s\_frozen：

s\_wait\_unfrozen：

s\_id[32] ：

s\_uuid[16] ：

s\_fs\_info：指向具体文件系统内存超级块的指针。比如，对于ext4文件系统，ext4的file\_system\_type结构中mount操作为ext4\_mount，其中要调用ext4\_fill\_super，会把ext4\_sb\_info结构指针赋值给s\_fs\_info；

s\_mode：

s\_time\_gran：指定文件系统使用时间戳的粒度，单位为ns，即1s的1/109为一般情况s\_time\_gran在mount的时候，会调用sget查找或者创建超级块，再调用alloc\_super分配超级块，s\_time\_gran赋值为1000000000，根据具体文件系统的不同，还有可能为别的值。如ext4文件系统，为1 << (EXT4\_EPOCH\_BITS - 2)；对于ceph分布式文件系统，为1000，1000 ns == 1 us；对NTFS，为1；

s\_vfs\_rename\_mutex;

s\_subtype：

s\_options：

s\_d\_op：

cleancache\_poolid：

s\_shrink：

s\_remove\_count：

s\_readonly\_remount：

struct super\_operations {

struct inode \*(\*alloc\_inode)(struct super\_block \*sb);

void (\*destroy\_inode)(struct inode \*);

void (\*dirty\_inode) (struct inode \*, int flags);

int (\*write\_inode) (struct inode \*, struct writeback\_control \*wbc);

int (\*drop\_inode) (struct inode \*);

void (\*evict\_inode) (struct inode \*);

void (\*put\_super) (struct super\_block \*);

void (\*write\_super) (struct super\_block \*);

int (\*sync\_fs)(struct super\_block \*sb, int wait);

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

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

int (\*statfs) (struct dentry \*, struct kstatfs \*);

int (\*remount\_fs) (struct super\_block \*, int \*, char \*);

void (\*umount\_begin) (struct super\_block \*);

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);

};

alloc\_inode

destroy\_inode

dirty\_inode：将传递的inode标记为脏，表示其数据已经被修改；

write\_inode

drop\_inode

evict\_inode

put\_super

write\_super

sync\_fs

reeze\_fs

unfreeze\_fs

statfs

remount\_fs

umount\_begin

show\_options

show\_devname

show\_path

show\_stats

quota\_read

quota\_write

bdev\_try\_to\_free\_page

nr\_cached\_objects

free\_cached\_objects

mount系统调用

mount系统调用的入口点是sys\_mount，通过SYSCALL\_DEFINE5来定义的。首先把安装选项（类型、设备和选项）从用户空间复制到内核空间，然后调用do\_mount，根据安装时设置的不同标志选择不同的处理函数，如：

**MS\_REMOUNT**

do\_remount：通过重新remount安装，修改被安装文件系统的安装选项。

**MS\_BIND**

do\_loopback：通过loopback回环设备安装文件系统。

**MS\_SHARED | MS\_PRIVATE | MS\_SLAVE | MS\_UNBINDABLE**

do\_change\_type：处理共享、从属和非绑定安装。

**MS\_MOVE**

do\_move\_mount：是将被安装文件系统的安装点从原目录移到一个新的目录。

**通常情况**

do\_new\_mount：默认情况下处理普通的安装操作，不需要特殊标志位。

**do\_new\_mount**分为两部分：do\_kern\_mount和do\_add\_mount。

**do\_kern\_mount：**

do\_kern\_mount(type, flags, name, data)的第三个参数name是被安装的设备名。首先使用get\_fs\_type从file\_systems链表里获取file\_system\_type实例，然后通过vfs\_kern\_mount调用特定文件系统的mount读取磁盘文件系统的超级块，返回vfsmount结构，该结构里保存有super\_block超级块的指针。vfs\_kern\_mount会新分配一个mount结构，通过mnt\_instance成员链接到被安装文件系统根目录dentry指向的超级块的s\_mounts链表。

**vfs\_kern\_mount：**

该函数主要是调用mount\_fs，在mount\_fs中，会调用特定文件系统file\_system\_type的mount，读取文件系统的超级块，函数返回值是被安装文件系统根目录的dentry结构指针。

**do\_add\_mount：**

先作两个判断，一是同样的文件系统不能被安装到同一安装点；二是不能安装链接文件。do\_add\_mount的核心是调用graft\_tree(newmnt, path)，而graft\_tree则主要是调用attach\_recursive\_mnt。

**attach\_recursive\_mnt**

核心是**mnt\_set\_mountpoint**，设置安装点：

child\_mnt->mnt\_mountpoint = dget(dentry);

child\_mnt->mnt\_parent = mnt;

以及commit\_tree，通过mnt\_hash 成员链接到mount\_hashtable全局哈希表；通过mnt\_child链接到父文件系统的mnt\_mounts链表。