文件系统层次分析

由上而下主要分为用户层、VFS层、文件系统层、缓存层、块设备层、磁盘驱动层、磁盘物理层

1. 用户层：最上面用户层就是我们日常使用的各种程序，需要的接口主要是文件的创建、删除、打开、关闭、写、读等。
2. VFS层：我们知道Linux分为用户态和内核态，用户态请求硬件资源需要调用System Call通过内核态去实现。用户的这些文件相关操作都有对应的System Call函数接口，接口调用 VFS对应的函数。
3. 文件系统层：不同的文件系统实现了VFS的这些函数，通过指针注册到VFS里面。所以，用户的操作通过VFS转到各种文件系统。文件系统把文件读写命令转化为对磁盘LBA的操作，起了一个翻译和磁盘管理的作用。
4. 缓存层：文件系统底下有缓存，Page Cache，加速性能。对磁盘LBA的读写数据缓存到这里。
5. 块设备层：块设备接口Block Device是用来访问磁盘LBA的层级，读写命令组合之后插入到命令队列，磁盘的驱动从队列读命令执行。Linux设计了电梯算法等对很多LBA的读写进行优化排序，尽量把连续地址放在一起。
6. 磁盘驱动层：磁盘的驱动程序把对LBA的读写命令转化为各自的协议，比如变成ATA命令，SCSI命令，或者是自己硬件可以识别的自定义命令，发送给磁盘控制器。Host Based SSD甚至在块设备层和磁盘驱动层实现了FTL，变成对Flash芯片的操作。
7. 磁盘物理层：读写物理数据到磁盘介质。

2.挂载描述符

一个文件系统，只有挂载到内存中目录树的一个目录下，进程才能访问这个文件系统。每次挂载文件系统，虚拟文件系统就会创建一个挂载描述符。挂载描述符用来描述文件系统的一个挂载实例，同一个存储设备上的文件系统可以多次挂载，每次挂载到不同的目录下。结构体为struc mount，在fs/mount.h文件中：

```
struct mount { struct hlist_node mnt_hash;//??? struct mount
 *mnt_parent;//?????? struct dentry *mnt_mountpoint;//??????
 struct vfsmount mnt;//?????????????????????? union {  struc
t rcu_head mnt_rcu;  struct llist_node mnt_llist; };#ifdef C
ONFIG_SMP struct mnt_pcp __percpu *mnt_pcp;#else int mnt_cou
nt; int mnt_writers;#endif struct list_head mnt_mounts;//???
????? struct list_head mnt_child;//??????mnt_mounts struct l
ist_head mnt_instance;//???????????????????? const char *mnt
_devname;//????????????/dev/hda1  struct list_head mnt_list;
 struct list_head mnt_expire; /* link in fs-specific expiry
list */ struct list_head mnt_share;//????????? struct list_h
ead mnt_slave_list;//??????? struct list_head mnt_slave;//??
??????? struct mount *mnt_master;//????????? struct mnt_name
space *mnt_ns;//???????? struct mountpoint *mnt_mp;//????? s
truct hlist_node mnt_mp_list;//?????????????????????? struct
 list_head mnt_umounting; /* list entry for umount propagati
on */#ifdef CONFIG_FSNOTIFY struct fsnotify_mark_connector _
_rcu *mnt_fsnotify_marks; __u32 mnt_fsnotify_mask;#endif int
 mnt_id;//??id int mnt_group_id;???id int mnt_expiry_mark;
/* true if marked for expiry */ struct hlist_head mnt_pins;
struct fs_pin mnt_umount; struct dentry *mnt_ex_mountpoint;}
 __randomize_layout;
```

3.文件系统类型

因为每种文件系统的超级块的格式不同，所以每种文件系统需要向虚拟文件系统注册文件系统类型 file_system_type，并且实现 mount
方法用来读取和解析超级块。结构体为struct file_system_type
，在include/linux/fs.h文件中：

```
struct file_system_type { const char *name;//?????? int fs_f
```

```
lags;//????#define FS_REQUIRES_DEV  1  /* ?????????? */#defi
ne FS_BINARY_MOUNTDATA 2 /*?????????????nfs */#define FS_HAS
_SUBTYPE  4 /* ????????????fuse */#define FS_USERNS_MOUNT  8
 /* Can be mounted by userns root */#define FS_RENAME_DOES_D
_MOVE 32768 /* FS will handle d_move() during rename() inter
nally. */ //??????????????????????? struct dentry *(*mount)
(struct file_system_type *, int,          const char *, void
*); //??????????? void (*kill_sb) (struct super_block *); st
ruct module *owner;//?????????????????THIS_MODULE struct fil
e_system_type * next;//????????????????? struct hlist_head f
s_supers;//????????????? struct lock_class_key s_lock_key; s
truct lock_class_key s_umount_key; struct lock_class_key s_v
fs_rename_key; struct lock_class_key s_writers_key[SB_FREEZE
_LEVELS]; struct lock_class_key i_lock_key; struct lock_clas
s_key i_mutex_key; struct lock_class_key i_mutex_dir_key;};
```

4.索引节点

在文件系统中，每个文件对应一个索引节点，而且一个索引节点只有文件被访问才会在内存中创建。索引节点描述了两类数据信息，1.文件的属性，也称为元数据；2.文件数据的存储位置。当内核访问存储设备上一个文件的时候，会在内核中创建和初始化一个节点，结构体为struct inode，在include/linux/fs.h文件中：

```
struct inode { umode_t   i_mode;//??????? unsigned short  i_
opflags;// kuid_t   i_uid;//???id kgid_t   i_gid;//???id uns
igned int  i_flags;//??????#ifdef CONFIG_FS_POSIX_ACL //????
???? struct posix_acl *i_acl; struct posix_acl *i_default_ac
l;#endif const struct inode_operations *i_op;//?????????? st
ruct super_block *i_sb;//????????? struct address_space *i_m
apping;//??????????????#ifdef CONFIG_SECURITY void   *i_secu
rity;//??????#endif /* Stat data, not accessed from path wal
king */ unsigned long  i_ino;//????? /*  * Filesystems may o
nly read i_nlink directly.  They shall use the  * following
functions for modification:  *  *    (set|clear|inc|drop)_nl
ink  *    inode_(inc|dec)_link_count  */ union {  const unsi
gned int i_nlink;//?????  unsigned int __i_nlink; }; dev_t
 i_rdev;//??????? loff_t   i_size;//???? struct timespec64 i
_atime;//?????? struct timespec64 i_mtime;//?????? struct ti
mespec64 i_ctime;//?????? spinlock_t  i_lock;//??? unsigned
```

```
short           i_bytes;//?????? u8   i_blkbits;//????????? u
8   i_write_hint;// blkcnt_t  i_blocks;//?????#ifdef __NEED_
I_SIZE_ORDERED seqcount_t  i_size_seqcount;//?i_size??????#e
ndif /* Misc */ unsigned long  i_state;//?????? struct rw_se
maphore i_rwsem;//????? unsigned long  dirtied_when; /* jiff
ies of first dirtying */ unsigned long  dirtied_time_when;//
????????? struct hlist_node i_hash;//?????????? struct list_
head i_io_list; /* backing dev IO list */#ifdef CONFIG_CGROU
P_WRITEBACK struct bdi_writeback *i_wb;  /* the associated c
group wb */ /* foreign inode detection, see wbc_detach_inode
() */ int   i_wb_frn_winner; u16   i_wb_frn_avg_time; u16
i_wb_frn_history;#endif struct list_head i_lru;//???????? st
ruct list_head i_sb_list; struct list_head i_wb_list;//?????
??? union {  struct hlist_head i_dentry;//?????  struct rcu_
head  i_rcu;//????? }; atomic64_t  i_version;//??? atomic_t
 i_count;//???? atomic_t  i_dio_count;//??io???? atomic_t  i
_writecount;//????#ifdef CONFIG_IMA atomic_t  i_readcount;//
????#endif const struct file_operations *i_fop;//???????? st
ruct file_lock_context *i_flctx;// struct address_space i_da
ta;//?????? struct list_head i_devices;//????? union {  stru
ct pipe_inode_info *i_pipe;//????  struct block_device *i_bd
ev;//?????  struct cdev  *i_cdev;//??????  char   *i_link;//
???  unsigned  i_dir_seq;//????? }; __u32   i_generation;#if
def CONFIG_FSNOTIFY __u32   i_fsnotify_mask; /* all events t
his inode cares about */ struct fsnotify_mark_connector __rc
u *i_fsnotify_marks;//????????#endif#if IS_ENABLED(CONFIG_FS
_ENCRYPTION) struct fscrypt_info *i_crypt_info;//????#endif
void   *i_private; /* fs or device private pointer */} __ran
domize_layout;
```

索引文件分为以下几种类型，在i_flags参数中区分：

- （1）普通文件（regular file）：就是我们通常说的文件，是狭义的文件。
- （2）目录：目录是一种特殊的文件，这种文件的数据是由目录项组成的，每个目录项 存储一个子目录或文件的名称以及对应的索引节点号。
- （3）符号链接（也称为软链接）：这种文件的数据是另一个文件的路径。
- （4）字符设备文件。
- （5）块设备文件。
- （6）命名管道（FIFO）。

- （7）套接字（socket）。

内核支持两种链接：

- （1）软链接，也称为符号链接，这种文件的数据是另一个文件的路径。
- （2）硬链接，相当于给一个文件取了多个名称，多个文件名称对应同一个索引节点，索引节点的成员 i_nlink 是硬链接计数。

索引节点的操作函数也很重要：

```
struct inode_operations { //???????????? struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int); const char * (*get_link) (struct dentry *, struct inode *, struct delayed_call *);//????inode??? int (*permission) (struct inode *, int);//?????inode??????  //?????????? struct posix_acl * (*get_acl)(struct inode *, int); int (*readlink) (struct dentry *, char __user *,int); //???????????open???? int (*create) (struct inode *,struct dentry *, umode_t, bool);  //?????????link???? int (*link) (struct dentry *,struct inode *,struct dentry *);  //?????????????????unlink???? int (*unlink) (struct inode *,struct dentry *);  //?????? int (*symlink) (struct inode *,struct dentry *,const char *); int (*mkdir) (struct inode *,struct dentry *,umode_t);//???? int (*rmdir) (struct inode *,struct dentry *);?//???? int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);//???????????????????  //???? int (*rename) (struct inode *, struct dentry *,  struct inode *, struct dentry *, unsigned int);   int (*setattr) (struct dentry *, struct iattr *);//?????? int (*getattr) (const struct path *, struct kstat *, u32, unsigned int);//?????? ssize_t (*listxattr) (struct dentry *, char *, size_t);  // int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,        u64 len); int (*update_time)(struct inode *, struct timespec64 *, int);//?????? int (*atomic_open)(struct inode *, struct dentry *,        struct file *, unsigned open_flag,       umode_t create_mode); int (*tmpfile) (struct inode *, struct dentry *, umode_t); int (*set_acl)(struct inode *, struct posix_acl *, int);//????????} ____cacheline_aligned;
```

5.目录项

目录项对象代表一个目录项，明明linux有个说法是一切皆文件，那为什么还要有目录项对象呢？因为虽然全部可以统一有索引节点表示，但是VFS需要经常执行目录相关操作，比如路径查找等，需要解析路径的每一个组成部分，不但要确保它有效，还需要进一步查找下一部分。解析一个路径并且遍历是一个耗时的，目录对象的引入会使得这个过程非常简单。目录项对象有struct dentry表示，在include/linux/dcache.h文件中：

```
struct dentry { /* RCU lookup touched fields */ unsigned int
 d_flags;//??????????d_lock?? seqcount_t d_seq;//????????? s
truct hlist_bl_node d_hash;//???????? struct dentry *d_paren
t;//????????? struct qstr d_name;//????? struct inode *d_ino
de;//??????????      unsigned char d_iname[DNAME_INLINE_LEN]
;//???????? /* Ref lookup also touches following */ struct l
ockref d_lockref;//??????? const struct dentry_operations *d
_op;//????????? struct super_block *d_sb;//????????? unsigne
d long d_time;//???? void *d_fsdata;//???????? union {  stru
ct list_head d_lru; //??????  wait_queue_head_t *d_wait;//??
?? }; struct list_head d_child;//??????????? struct list_hea
d d_subdirs;//??????????? /*  * d_alias and d_rcu can share
memory  */ union {  struct hlist_node d_alias;//????????  st
ruct hlist_bl_node d_in_lookup_hash; /* only for in-lookup o
nes */   struct rcu_head d_rcu;//rcu?? } d_u;} __randomize_l
ayout;
```

和前面两个对象不同，目录项对象没有对应的磁盘结构，VFS根据字符串形式的路径现场创建它，由于目录项对象没有真正的保存在磁盘中，目录项没有修改标志、回写等。目录项有三种状态：

- 被使用：一个被使用的目录项对象对有一个有效的索引节点，并且该对象至少有一个使用者
- 未被使用：一个未被使用的目录项对象也对有一个有效的索引节点，但是该对象没有使用者（d_count为0）
- 负状态：一个负状态的目录项对象没有一个有效的索引节点，可能因为索引节点被删除，也可能路径不正确

如果VFS遍历路径名中的所有元素并解析成目录项对象，还要达到最深层次，这是非常费力的事情，会浪费大量时间，所以VFS会对目录项对象遍历解析，解析完毕

后会缓存在dcache中，缓存主要包括三部分：

- 被使用目录项链表，存放被使用的目录项
- 最近被使用双向链表，存放未被使用和负状态的目录项，并且安装使用时间排序
- 散列表，用于快速匹配目录项

现在看看目录项的操作函数：

```
struct dentry_operations { //??????????? int (*d_revalidate)(struct dentry *, unsigned int); int (*d_weak_revalidate)(struct dentry *, unsigned int);  //?????????????????????? int (*d_hash)(const struct dentry *, struct qstr *);  //?????????????????????????????????? int (*d_compare)(const struct dentry *,   unsigned int, const char *, const struct qstr *);     //?????????d_count?0??? int (*d_delete)(const struct dentry *);  //??????????? int (*d_init)(struct dentry *);  //?????????????????? void (*d_release)(struct dentry *); void (*d_prune)(struct dentry *);  //?????????????????? void (*d_iput)(struct dentry *, struct inode *); char *(*d_dname)(struct dentry *, char *, int); struct vfsmount *(*d_automount)(struct path *); int (*d_manage)(const struct path *, bool); struct dentry *(*d_real)(struct dentry *, const struct inode *);} ____cacheline_aligned;
```

6.文件对象

文件对象代表进程打开的文件，包括普通文件和目录文件，是用户直接操作的对象，也是用户最熟悉的对象。他是由open创建，close撤销使用的。通过文件对象，VFS可以找到其对应的目录项和索引节点，从而找到所在的超级块。文件对象实际上没有对应的磁盘结构，他的作用是连接用户和VFS，给与用户操作文件的方法，从而实现间接操作磁盘文件。文件对象由struct file结构体表示，在在include/linux/fs.h文件中：

```
struct file { union {  struct llist_node fu_llist;//?????? struct rcu_head  fu_rcuhead;//????rcu?? } f_u; struct path f_path;//??????? struct inode  *f_inode;//????????? const struct file_operations *f_op;//????????? /*  * Protects f_ep_links, f_flags.  * Must not be taken from IRQ context.  */ sp
```

```
inlock_t  f_lock;//????? enum rw_hint  f_write_hint;//??????
?? atomic_long_t  f_count;//???????? unsigned int   f_flags;
//???????????? fmode_t   f_mode;//???????????????? struct mut
ex  f_pos_lock;//f_pos???? loff_t   f_pos;//???????? struct
fown_struct f_owner;//???????????IO???? const struct cred *f
_cred;//?????? struct file_ra_state f_ra;//?????? u64   f_ve
rsion;//???#ifdef CONFIG_SECURITY void   *f_security;//????#
endif /* needed for tty driver, and maybe others */ void   *
private_data;//???????????tty??#ifdef CONFIG_EPOLL /* Used b
y fs/eventpoll.c to link all the hooks to this file */ struc
t list_head f_ep_links;//????? struct list_head f_tfile_llin
k;//??????????#endif /* #ifdef CONFIG_EPOLL */ struct addres
s_space *f_mapping;//??????? errseq_t  f_wb_err;//??????????
?} __randomize_layout
```

文件对象的操作方法函数很重要，由struct file_operations表示：

```
struct file_operations { struct module *owner; //???????????
?????llseek??? loff_t (*llseek) (struct file *, loff_t, int)
; ssize_t (*read) (struct file *, char __user *, size_t, lof
f_t *);//? ssize_t (*write) (struct file *, const char __use
r *, size_t, loff_t *);//?  //????????????????????? ssize_t
(*read_iter) (struct kiocb *, struct iov_iter *);//??? ssize
_t (*write_iter) (struct kiocb *, struct iov_iter *);//??? i
nt (*iterate) (struct file *, struct dir_context *); int (*i
terate_shared) (struct file *, struct dir_context *);  //???
???????????????poll?? __poll_t (*poll) (struct file *, stru
ct poll_table_struct *);  //?????????????????ioctl?? long (*
unlocked_ioctl) (struct file *, unsigned int, unsigned long)
; long (*compat_ioctl) (struct file *, unsigned int, unsigne
d long);  //???????? int (*mmap) (struct file *, struct vm_a
rea_struct *); unsigned long mmap_supported_flags; int (*ope
n) (struct inode *, struct file *);//?? int (*flush) (struct
 file *, fl_owner_t id);//?? int (*release) (struct inode *,
 struct file *);//?? int (*fsync) (struct file *, loff_t, lo
ff_t, int datasync);//????? int (*fasync) (int, struct file
*, int);//??????? int (*lock) (struct file *, int, struct fi
le_lock *);//????? ssize_t (*sendpage) (struct file *, struc
t page *, int, size_t, loff_t *, int);  //???????????? unsig
```

```
ned long (*get_unmapped_area)(struct file *, unsigned long,
unsigned long, unsigned long, unsigned long); int (*check_fl
ags)(int);//??flags?????????nfs?????O_APPEND?O_DIRECT??  //?
???????flock??? int (*flock) (struct file *, int, struct fi
le_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 *, lo
ng, struct file_lock **, void **); long (*fallocate)(struct
file *file, int mode, loff_t offset,     loff_t len); void (
*show_fdinfo)(struct seq_file *m, struct file *f);#ifndef CO
NFIG_MMU unsigned (*mmap_capabilities)(struct file *);#endif
 ssize_t (*copy_file_range)(struct file *, loff_t, struct fi
le *,   loff_t, size_t, unsigned int); int (*clone_file_rang
e)(struct file *, loff_t, struct file *, loff_t,   u64); int
 (*dedupe_file_range)(struct file *, loff_t, struct file *,
loff_t,   u64); int (*fadvise)(struct file *, loff_t, loff_t
, int);} __randomize_layout;
```

总结：

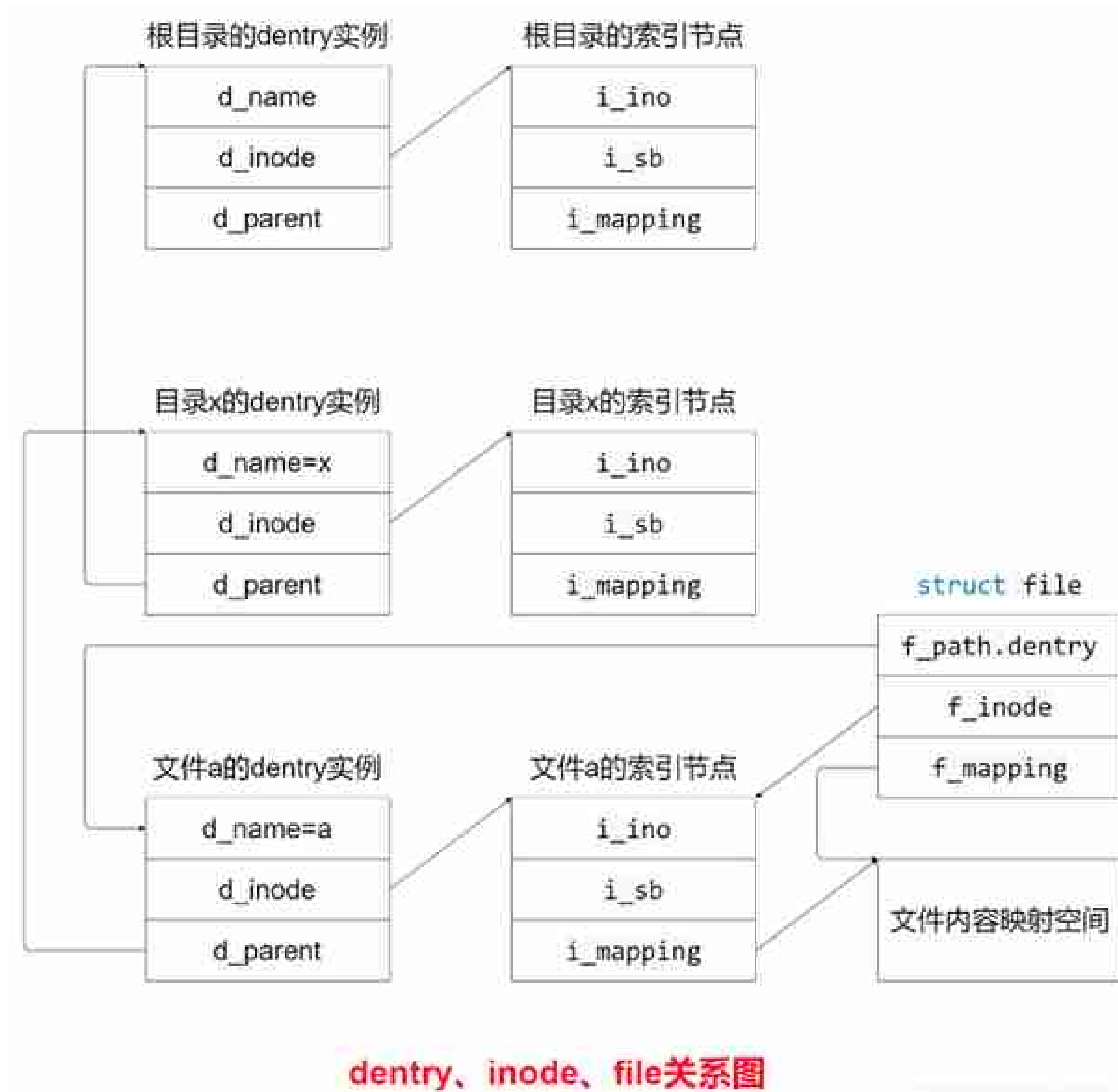
我们在进程中挂载了一个文件系统，也就是说找到了这个超级块super_block结构体，可以通过super_block结构体中的s_inodes（索引）找到对应的文件，同时，在遍历inodes的时候，会自动的解析inode路径的每一个组成部分，组成struct dentry（目录项），方便系统使用树的形式表示inode（索引）之间的关系。这样子我们打开了这个磁盘挂载的目录就可以看到磁盘的目录和文件了，我们打开了一个索引，系统会创建一个struct
file（文件）结构体，这就是我们平时操作一个文件的方式了。

相反，我们在进程中操作一个文件（struct file），可以通过struct file中的struct inode参数找到其索引，进而找到超级块（struct
super_block），这样子，VFS就知道要操作的文件系统和索引了。

上面的是VFS对上层的通道，对底层又会是怎么样子呢？其实，在内核初始化的时候，会注册了一种文件系统类型，VFS挂载这种文件系统会根据super_block结构体中struct file_system_type *s_type，找到这个文件系统类型，内核才可以根据文件系统类型来调用对应超级块的操作函数，因为每一种文件系统类型的超级块操作函数具体是实现是不同的。然后在挂载这种文件系统的时候，会创建一个struct mount 实例，当然，比如有两个u盘，要挂载两个，就要有两个struct mount

实例。

补充一下，file_system_type、 super block、 mount的关系图：



文章链接：https://mp.weixin.qq.com/s/Z2jcUzpPHeBcobRrUl8wfw

转载自：人人极客社区

文章来源：文件系统专栏 | 之文件系统架构

版权申明：本文来源于网络，免费传达知识，版权归原作者所有。如涉及作品版权问题，请联系我进行删除。