文件系统

by zenhumany

2012-05-19——2012-06-03

目录

1. 概述

本章中所涉及的源代码全部来至 linux 内核 2.6.34。

操作系统最重要的部分莫过于文件系统和进程管理了。一个完整的 linux 操作系统，通常包含数千到数百万个文件。

每种操作系统都至少有一种“标准文件系统”，提供一些功能，可以高效而可靠的执行所需的任务。Linux 中标准的文件系统为 ext2/ext3。

为了支持各种文件系统，且在同时允许访问其他操作系统的文件，linux 内核在用户进程和文件系统中引入了一个抽象层。该抽象层成为虚拟文件系统（VFS，Virtual File System）。

虚拟文件系统接口和数据结构构成了一个结构，各个文件系统都在该框架下运转。

本章将讲解虚拟文件系统的接口和相关数据结构，同事讨论在该框架下实现的具体文件系统 ext2。

2. 虚拟文件系统和相关数据结构

Linux 系统能够快速的成长，与其支持各种类型的文件系统不无关系。Linux 除了本身的文件系统 ext2 外，可以支持其他各种不同文件系统。要实现这个目的，就需要将对各种不同文件系统的操作和管理纳入到一个统一的框架中，然内核中的文件系统界面成为一条“总线”，使得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进行操作。这就是 VFS 所要达到的目的。下面看和 VFS 相关的数据结构。

2.1 inode文件索引节点

在 linux 系统中，磁盘上的一个文件由 indoe 索引节点唯一的标识。

include/linux/fs.h

```
struct inode {
        struct hlist_node       i_hash;
        struct list_head        i_list;         /* backing dev IO list */
        struct list_head        i_sb_list;
        struct list_head        i_dentry;
        unsigned long           i_ino;
        atomic_t                i_count;
        unsigned int            i_nlink;
        uid_t                   i_uid;
        gid_t                   i_gid;
        dev_t                   i_rdev;
        unsigned int            i_blkbits;
        u64                     i_version;
        loff_t                  i_size;
#ifdef __NEED_I_SIZE_ORDERED
        seqcount_t              i_size_seqcount;
#endif
        struct timespec         i_atime;
        struct timespec         i_mtime;
        struct timespec         i_ctime;
        blkcnt_t                i_blocks;
        unsigned short          i_bytes;
        umode_t                 i_mode;
        spinlock_t              i_lock; /* i_blocks, i_bytes, maybe i_size */
        struct mutex            i_mutex;
        struct rw_semaphore     i_alloc_sem;
        const struct inode_operations   *i_op;
        const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
        struct super_block      *i_sb;
        struct file_lock        *i_flock;
```

```
        struct address_space        *i_mapping;
        struct address_space        i_data;
#ifdef CONFIG_QUOTA
        struct dquot            *i_dquot[MAXQUOTAS];
#endif
        struct list_head        i_devices;
        union {
                struct pipe_inode_info  *i_pipe;
                struct block_device     *i_bdev;
                struct cdev             *i_cdev;
                struct cdev             *i_cdev;
        };

        __u32                   i_generation;

#ifdef CONFIG_FSNOTIFY
        __u32                   i_fsnotify_mask; /* all events this inode cares about */
        struct hlist_head       i_fsnotify_mark_entries; /* fsnotify mark entries */
#endif

#ifdef CONFIG_INOTIFY
        struct list_head        inotify_watches; /* watches on this inode */
        struct mutex            inotify_mutex;  /* protects the watches list */
#endif

        unsigned long           i_state;
        unsigned long           dirtied_when;   /* jiffies of first dirtying */

        unsigned int            i_flags;

        atomic_t                i_writecount;
```

```
#ifdef CONFIG_SECURITY
        void                    *i_security;
#endif
#ifdef CONFIG_FS_POSIX_ACL
        struct posix_acl        *i_acl;
        struct posix_acl        *i_default_acl;
#endif
        void                    *i_private; /* fs or device private pointer */
};
```

- ■i_hash：散列链表指针
- ■i_list：索引节点当前转台的链表指针
- ■i_sb_list：超级块索引节点指针
- ■i_dentry：引用索引节点的目录对象链表头
- ■i_ino：索引节点在磁盘上的索引节点号。
- ■i_op：为具体文件系统和 inode 相关的操作提供一个插口
- ■i_fop：为具体文件系统和文件相关的操作提供一个插口
- ■i_sb：指向该索引节点所属的超级块对象。
- ■i_mapping：指向缓存 address_space 对象的指针，该结构是文件系统缓存的关键。
- ■i_data：嵌入在 inode 中的 address_space 对象。

每个 inode 的成员可以分为两类，一类为描述文件状态的元数据，一类为保存实际文件的数据段。

每个 inode 都有一个“i 节点号”i_ino，在同一文件系统中该节点号是唯一的。

inode 结构中有两个设备号，即 i_dev 和 i_cdev。除了特殊文件外，一个索引节点总的存储在某个设备上，这就是 i_dev。其次，如果索引节点代表的不是常规文件，而是某个设备，那就要有个设备号，这个就是 i_rdev。

就像人可以有别名一样，文件也可以有别名，也就是将一个以创建的文件“连接”到另一个文件名，可以沿着 i_dentry 找到所有这个文件的“别名”。

在应用层，总是通过文件名来标识文件，表示文件名的数据结构式 dentry。

2.2 目录项缓存dentry

为了加快文件的访问速度，内核需要缓存已经打开过的文件的信息，用来缓存文件信息的数据结构就是 dentry。

include/linux/dcache.h

```
struct dentry {
        atomic_t d_count;
        unsigned int d_flags;           /* protected by d_lock */
        spinlock_t d_lock;          /* per dentry lock */
        int d_mounted;
        struct inode *d_inode;          /* Where the name belongs to - NULL is
                                         * negative */
        /*
         * The next three fields are touched by __d_lookup.  Place them here
         * so they all fit in a cache line.
         */
        struct hlist_node d_hash;       /* lookup hash list */
        struct dentry *d_parent;        /* parent directory */
        struct qstr d_name;

        struct list_head d_lru;         /* LRU list */
        /*
         * d_child and d_rcu can share memory
         */
        union {
                struct list_head d_child;       /* child of parent list */
                struct rcu_head d_rcu;
        } d_u;
```

```
        struct list_head d_subdirs; /* our children */
        struct list_head d_alias;      /* inode alias list */
        unsigned long d_time;            /* used by d_revalidate */
        const struct dentry_operations *d_op;
        struct super_block *d_sb;      /* The root of the dentry tree */
        void *d_fsdata;              /* fs-specific data */

        unsigned char d_iname[DNAME_INLINE_LEN_MIN];       /* small names */
};
```

- d_inode：指向目录项对应的 inode。
- d_parent：指向父目录。
- d_name：文件名。
- d_subdirs：对目录而言，子目录项的链表的头。
- d_alias：用于与同一索引节点（别名）相关的目录项链表的指针。
- d_op：为具体的文件系统提供目录项操作接口。
- d_sb：文件超级快对象。
- d_iname：短文件名存储空间

```
struct qstr {
        unsigned int hash;
        unsigned int len;
        const unsigned char *name;
};
```

- hash：文件名 hash 值
- len：文件名长度
- name：指向存放文件名空间的指针。

2.3 文件对象

文件对象描述进程与一个打开文件的交互，是打开文件的一个上下文。文件对象在磁盘上没有对应的映射。

```
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;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
#define f_dentry        f_path.dentry
#define f_vfsmnt        f_path.mnt
        const struct file_operations    *f_op;
        spinlock_t              f_lock;  /* f_ep_links, f_flags, no IRQ */
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hooks to this file */
```

```
	struct list_head	f_ep_links;
#endif /* #ifdef CONFIG_EPOLL */
	struct address_space	*f_mapping;
#ifdef CONFIG_DEBUG_WRITECOUNT
	unsigned long f_mnt_write_state;
#endif
};
```

include/linux/path.h

```
struct dentry;
struct vfsmount;

struct path {
	struct vfsmount *mnt;
	struct dentry *dentry;
};
```

- f_path.dentry：指向 file 对应的 dentry 结构
- f_paht.mnt：指向 file 所在文件的文件系统类型。
- f_op：为具体的文件系统提供文件操作插口。
- f_pos：文件当前的读写位置，也就是文件的上下文。
- f_mapping：指向文件地址空间对象的指针。
- private_data：指向特定文件系统或设备驱动程序所需的数据的指针

2.4 特定于进程的数据结构

用户进程都是通过一个文件描述符（就是整数）来在一个进程内唯一的标识打开的文件。

这就需要内核可以在用户进程中的描述符和内核之间内部使用结构建立一种关联。

每个进程的 task_struct 都包含了用于完成该工作的成员。

Include/linux/sched.h

```
struct task_struct {
……
```

```
/* filesystem information */
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
/* namespaces */
        struct nsproxy *nsproxy;
/* signal handlers */
……
};
```

fs：文件系统信息

files：打开文件信息

2.4.1 fs_struct

include/linux/fs_struct.h

```
struct fs_struct {
        int users;
        rwlock_t lock;
        int umask;
        int in_exec;
        struct path root, pwd;
};
```

include/linux/path.h

```
struct dentry;
struct vfsmount;

struct path {
        struct vfsmount *mnt;
        struct dentry *dentry;
};
```

fs_struct 中的连个类型为 path 的变量 root，pwd 为关键的结构。

path 结构中 dentry 指向一个目录项，mnt 表示 dentry 代表的目录项的安装的数据结构 vfsmount。

pwd.dentry,root.dentry 分别表示进程当前所在的目录，进程的“根目录”，也就是用户登录系统时看到的目录。这两个目录有可能并不在同一文件系统中，进程的根目录通常是安装于“/”节点上的 ext2 文件系统，而当前工作目录则可能安装于/dosc 的一个 DOS 文件系统。在文件系统中这些安装点起着非常重要的作用，所以 pwd.mnt，root.mnt 分别指向对应 dentry 的安装点。

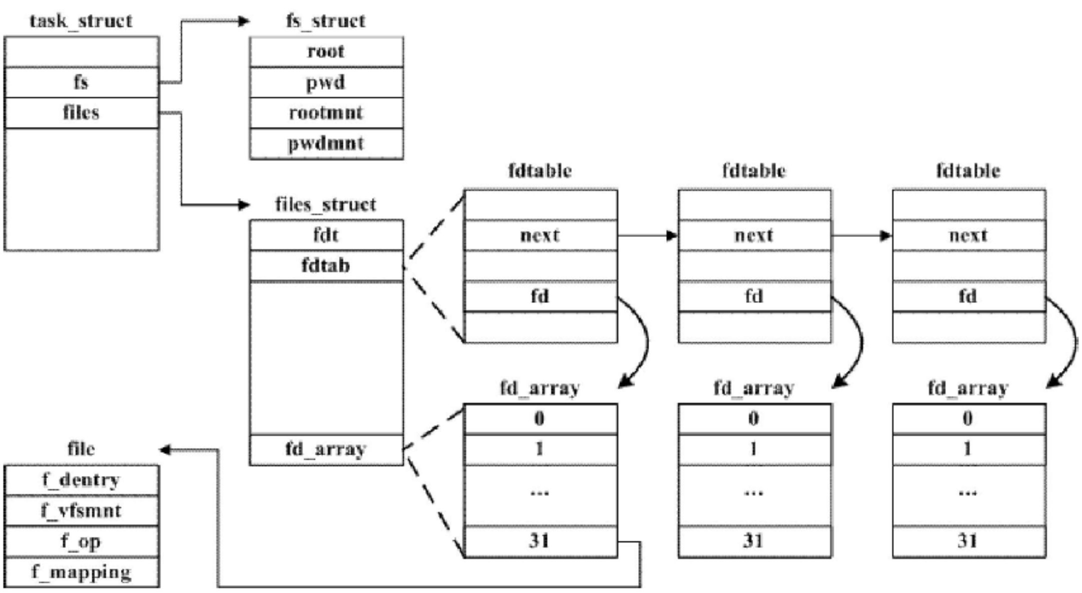
2.4.2 files_struct

```
/*
 * Open file table structure
 */
struct files_struct {
  /*
   * read mostly part
   */
    atomic_t count;
    struct fdtable *fdt;
    struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
    spinlock_t file_lock ____cacheline_aligned_in_smp;
    int next_fd;
    struct embedded_fd_set close_on_exec_init;
    struct embedded_fd_set open_fds_init;
    struct file * fd_array[NR_OPEN_DEFAULT];
};
```

fdtab 嵌入到 files_struct 中，并由 fdt 指向它。

```
struct fdtable {
        unsigned int max_fds;
        struct file ** fd;          /* current fd array */
        fd_set *close_on_exec;
        fd_set *open_fds;
        struct rcu_head rcu;
        struct fdtable *next;
};
```

fd 指向 file* fd_array[]的数组，当 fd_array 中没有空闲的元素时，扩充 fdtable，并由 next 连接。



2.5 文件系统file_system_type

内核支持的每一种文件系统都有一个 file_system_type 实例。

```
struct file_system_type {
        const char *name;
        int fs_flags;
        int (*get_sb) (struct file_system_type *, int,
                        const char *, void *, struct vfsmount *);
        void (*kill_sb) (struct super_block *);
        struct module *owner;
```

```
        struct file_system_type * next;
        struct list_head fs_supers;

        struct lock_class_key s_lock_key;
        struct lock_class_key s_umount_key;

        struct lock_class_key i_lock_key;
        struct lock_class_key i_mutex_key;
        struct lock_class_key i_mutex_dir_key;
        struct lock_class_key i_alloc_sem_key;
};
```

name：文件系统的名称

get_sb：获取超级快的函数。

2.6 安装点vfsmount

把一个设备安装到一个目录节点时需要用到一个 vfsmount 数据结构作为“连接件”。

```
struct vfsmount {
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent;        /* fs we are mounted on */
        struct dentry *mnt_mountpoint;    /* dentry of mountpoint */
        struct dentry *mnt_root;        /* root of the mounted tree */
        struct super_block *mnt_sb; /* pointer to superblock */
        struct list_head mnt_mounts;        /* list of children, anchored here */
        struct list_head mnt_child; /* and going through their mnt_child */
        int mnt_flags;
        /* 4 bytes hole on 64bits arches */
        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 vfsmount *mnt_master;       /* slave is on master->mnt_slave_list */
        struct mnt_namespace *mnt_ns;      /* containing namespace */
        int mnt_id;                  /* mount identifier */
        int mnt_group_id;            /* peer group identifier */
        /*
         * We put mnt_count & mnt_expiry_mark at the end of struct vfsmount
         * to let these frequently modified fields in a separate cache line
         * (so that reads of mnt_flags wont ping-pong on SMP machines)
         */
        atomic_t mnt_count;
        int mnt_expiry_mark;             /* true if marked for expiry */
        int mnt_pinned;
        int mnt_ghosts;
#ifdef CONFIG_SMP
        int __percpu *mnt_writers;
#else
        int mnt_writers;
#endif
};
```

- ■mnt_parent：指向安装点所在设备当初安装时的 vfsmount，也就是上一层的 vfsmount。
- ■mnt_mountpoint，mnt_root：安装点的 dentry 数据结构，mount_root 则指向所安装设备上根目录的 dentry 结构，这连个目录在目录树中处于同一层次。
- ■mnt_sb：所安装设备的超级块。

■mnt_mounts，mnt_child：当上一层的 vfsmount 存在时，通过 mnt_child 链入到 mnt_mounts 中。

■mnt_list：系统中有个总的 vfsmount 结构队列 vfsmntlist，所有以安装的 vfsmount 结构都通过 mnt_list 链入到 vfsmntlist 中。

2.7 super_block超级块

文件系统的超级块记录了文件系统的大部分信息。

```
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;
        int             s_need_sync;
        atomic_t                s_active;
#ifdef CONFIG_SECURITY
        void                            *s_security;
#endif
        struct xattr_handler    **s_xattr;
```

```

	struct list_head	s_inodes;	/* all inodes */
	struct hlist_head	s_anon;		/* anonymous dentries for (nfs) exporting */
	struct list_head	s_files;
	/* s_dentry_lru and s_nr_dentry_unused are protected by dcache_lock */
	struct list_head	s_dentry_lru;	/* unused dentry lru */
	int			s_nr_dentry_unused;	/* # of dentry on lru */

	struct block_device	*s_bdev;
	struct backing_dev_info *s_bdi;
	struct mtd_info		*s_mtd;
	struct list_head	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 */

	void 			*s_fs_info;	/* Filesystem private info */
	fmode_t			s_mode;

	/* Granularity of c/m/atime in ns.
	   Cannot be worse than a second */
	u32		   s_time_gran;

	/*
	 * The next field is for VFS *only*. No filesystems have any business
	 * even looking at it. You had been warned.
	 */
```

```
        struct mutex s_vfs_rename_mutex;        /* Kludge */

        /*
         * Filesystem subtype.  If non-empty the filesystem type field
         * in /proc/mounts will be "type.subtype"
         */
        char *s_subtype;

        /*
         * Saved mount options for lazy filesystems using
         * generic_show_options()
         */
        char *s_options;
};
```

■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_blocksize_bits：以位为单位的块大小

■s_dirt：是否为脏标识符

■s_maxbytes：文件的最长长度

■s_op：具体文件系统超级块插口

■type：文件系统类型

■s_root：文件系统根目录的目录项对象

■s_inodes：所有索引节点的链表头

■s_anon：用于处理远程网络文件系统的匿名目录项的链表。

■s_files：文件对象的链表。

■s_fs_info：指向具体的文件系统的超级块信息。

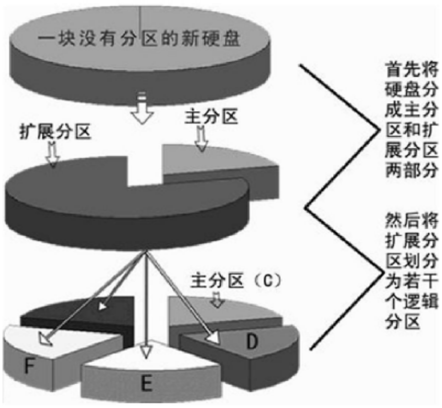
3. ext2 文件系统

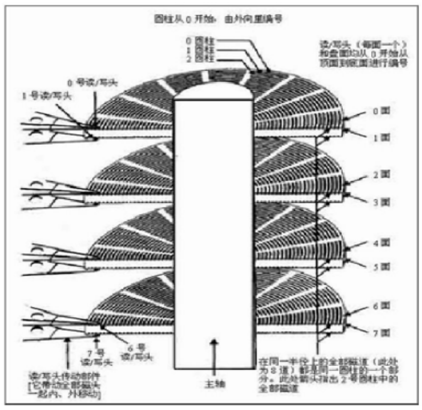
虚拟文件系统（VFS）接口和数据结构构成了一个框架，需要具体的文件系统来填充框架中的内容。本节以 linux 文件系统 ext2 为例讲解。

3.1 Ext2 磁盘数据结构

要学习 EXT2 文件系统，首先需要学习 EXT2 的磁盘设计。也就是 EXT2 将一个磁盘分区格式化成一个什么样子。

现代的磁盘驱动器都是多片的，所以不同的盘面上的相同磁道合在一起就形成了“柱面”（cylinder）的概念。从磁盘读出多个记录块时，如果从同一柱面中读取就比较快，因为这种情况下不需要移动磁头（磁头组）。互相连续的记录块实际上分布在同一柱面的各个盘面上，只有将一个柱面用完





后才会进入下一个柱面。所以，在许多文件系统中都把整个设备划分为若干柱面组，将反映着盘面存储空间的组织与管理的信息分散后就近存储于各个柱面中。相比之下，早期的文件系统往往将这些信息集中存储在一起，使得磁头在文件访问时来回“疲于奔命”而降低了效率。

ext2 文件系统在硬盘上的布局如下：

启动块	块组 0	块组 1	……	块组 n

每一个块组的划分如下：

超级块	组描述符	数据位图	inode 位图	inode 表	数据块

在 linux 下面，目录也是作为一种特定的文件来实现的。

- 图中组描述块记录着全部的组描述符结构，具体占用的块数取决于设备的大小。

- 数据块位图是本组的记录块位图，每一位对应一个记录块，1 表示已分配，0 表示空闲。
- Inode 位图是本块组中 inode 分配的记录位图，一般 inode 位图小于一个记录块。
- Inode 表：本块组中用来存放 inode 的记录块集合
- 数据库：本块组中用来存放数据块的记录块集合。

3.1.1 磁盘超级块

磁盘超级块描叙了磁盘格式化后各方面的信息，存储了用于管理磁盘的元信息。

EXT2 在磁盘上的超级块的数据结构为 ext2_super_block。

include/linux/ext2_fs.h

```
struct ext2_super_block {
        __le32  s_inodes_count;         /* Inodes count */
        __le32  s_blocks_count;         /* Blocks count */
        __le32  s_r_blocks_count;       /* Reserved blocks count */
        __le32  s_free_blocks_count;    /* Free blocks count */
        __le32  s_free_inodes_count;    /* Free inodes count */
        __le32  s_first_data_block; /* First Data Block */
        __le32  s_log_block_size;       /* Block size */
        __le32  s_log_frag_size;        /* Fragment size */
        __le32  s_blocks_per_group; /* # Blocks per group */
        __le32  s_frags_per_group;  /* # Fragments per group */
        __le32  s_inodes_per_group; /* # Inodes per group */
        __le32  s_mtime;                /* Mount time */
        __le32  s_wtime;                /* Write time */
        __le16  s_mnt_count;            /* Mount count */
        __le16  s_max_mnt_count;        /* Maximal mount count */
        __le16  s_magic;                /* Magic signature */
        __le16  s_state;                /* File system state */
        __le16  s_errors;               /* Behaviour when detecting errors */
```

```
__le16  s_minor_rev_level;  /* minor revision level */
__le32  s_lastcheck;        /* time of last check */
__le32  s_checkinterval;    /* max. time between checks */
__le32  s_creator_os;       /* OS */
__le32  s_rev_level;        /* Revision level */
__le16  s_def_resuid;       /* Default uid for reserved blocks */
__le16  s_def_resgid;       /* Default gid for reserved blocks */
/*
 * These fields are for EXT2_DYNAMIC_REV superblocks only.
 *
 * Note: the difference between the compatible feature set and
 * the incompatible feature set is that if there is a bit set
 * in the incompatible feature set that the kernel doesn't
 * know about, it should refuse to mount the filesystem.
 *
 * e2fsck's requirements are more strict; if it doesn't know
 * about a feature in either the compatible or incompatible
 * feature set, it must abort and not try to meddle with
 * things it doesn't understand...
__le32  s_first_ino;        /* First non-reserved inode */
__le16   s_inode_size;      /* size of inode structure */
__le16  s_block_group_nr;   /* block group # of this superblock */
__le32  s_feature_compat;   /* compatible feature set */
__le32  s_feature_incompat;     /* incompatible feature set */
__le32  s_feature_ro_compat;    /* readonly-compatible feature set */
__u8     s_uuid[16];        /* 128-bit uuid for volume */
char     s_volume_name[16];  /* volume name */
char     s_last_mounted[64];    /* directory where last mounted */
__le32  s_algorithm_usage_bitmap; /* For compression */
/*
 * Performance hints.  Directory preallocation should only
```

```
	 * happen if the EXT2_COMPAT_PREALLOC flag is on.
	 */
	__u8	s_prealloc_blocks;	/* Nr of blocks to try to preallocate*/
	__u8	s_prealloc_dir_blocks;	/* Nr to preallocate for dirs */
	__u16	s_padding1;
	/*
	 * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
	 */
	__u8	s_journal_uuid[16]; /* uuid of journal superblock */
	__u32	s_journal_inum;		/* inode number of journal file */
	__u32	s_journal_dev;		/* device number of journal file */
	__u32	s_last_orphan;		/* start of list of inodes to delete */
	__u32	s_hash_seed[4];		/* HTREE hash seed */
	__u8	s_def_hash_version; /* Default hash version to use */
	__u8	s_reserved_char_pad;
	__u16	s_reserved_word_pad;
	__le32	s_default_mount_opts;
	__le32	s_first_meta_bg;	/* First metablock block group */
	__u32	s_reserved[190];	/* Padding to the end of the block */
};
```

- s_inodes_count：索引节点的总数
- s_blocks_count：块总数
- s_r_blocks_count：保留的块数
- s_free_blocks_count：空闲的块数
- s_free_indoes_count：空闲的索引节点数
- s_first_data_block：第一个使用的块号
- s_log_block_size：块的大小
- s_log_frag_size：片的大小
- s_blocks_per_group：每组中的块数
- s_frags_per_group：没组中的片数
- s_inodes_per_group：每组中的索引节点数

- s_mtime：最后一次安装操作的时间
- s_wtime：最后一次写操作的时间
- s_mnt_count：被执行安装操作的次数
- s_max_mnt_count：检查之前安装操作的次数
- s_magic：文件系统魔术签名
- s_state：文件系统的状态标志
- s_first_ino：第一个非保留的索引节点号
- s_inode_size：磁盘上索引节点结构的大小
- s_block_group_nr：这个超级快的块组号
- s_prealloc_blocks：预分配的块数
- s_prealloc_dir_blocks：为目录预分配的块数。

3.1.2 组描述符

每个块组都有一个组描述符，用于描述本块组的元信息。

```
/*
 * Structure of a blocks group descriptor
 */
struct ext2_group_desc
{
        __le32  bg_block_bitmap;        /* Blocks bitmap block */
        __le32  bg_inode_bitmap;        /* Inodes bitmap block */
        __le32  bg_inode_table;     /* Inodes table block */
        __le16  bg_free_blocks_count;   /* Free blocks count */
        __le16  bg_free_inodes_count;   /* Free inodes count */
        __le16  bg_used_dirs_count; /* Directories count */
        __le16  bg_pad;
        __le32  bg_reserved[3];
};
```

- bg_block_bitmap：块位图的块号
- bg_inode_bitmap：索引节点位图的块号

- bg_inode_table：索引节点表块的起始块号
- bg_free_blocks_count：组中空闲块的个数
- bg_free_inodes_count：组中空闲索引节点的个数
- bg_used_dirs_count：组中目录的个数

3.1.3 索引节点

索引节点记录了磁盘上一个文件的元信息。

```
/*
 * Structure of an inode on the disk
 */
struct ext2_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Creation time */
        __le32  i_mtime;        /* Modification time */
        __le32  i_dtime;        /* Deletion Time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_flags;        /* File flags */
        union {
                struct {
                        __le32  l_i_reserved1;
                } linux1;
                struct {
                        __le32  h_i_translator;
                } hurd1;
                struct {
```

```
                __le32  m_i_reserved1;
        } masix1;
    } osd1;                 /* OS dependent 1 */
    __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
    __le32  i_generation;   /* File version (for NFS) */
    __le32  i_file_acl; /* File ACL */
    __le32  i_dir_acl;  /* Directory ACL */
    __le32  i_faddr;    /* Fragment address */
    union {
        struct {
            __u8    l_i_frag;   /* Fragment number */
            __u8    l_i_fsize;  /* Fragment size */
            __u16   i_pad1;
            __le16  l_i_uid_high;   /* these 2 fields    */
            __le16  l_i_gid_high;   /* were reserved2[0] */
            __u32   l_i_reserved2;
        } linux2;
        struct {
            __u8    h_i_frag;   /* Fragment number */
            __u8    h_i_fsize;  /* Fragment size */
            __le16  h_i_mode_high;
            __le16  h_i_uid_high;
            __le16  h_i_gid_high;
            __le32  h_i_author;
        } hurd2;
        struct {
            __u8    m_i_frag;   /* Fragment number */
            __u8    m_i_fsize;  /* Fragment size */
            __u16   m_pad1;
            __u32   m_i_reserved2[2];
        } masix2;
```

```
        } osd2;                    /* OS dependent 2 */
};
```

- i_mode：文件类型和访问权限
- i_uid：拥有者标示符
- i_size：以字节为单位的文件长度
- i_atime：最后一次访问文件的时间
- i_ctime：索引节点最后改变的时间
- i_mtime：文件内容最后改变的时间
- i_dtime：文件删去的时间
- i_gid：用户组标识符低 16 位
- i_links_count：硬链接计数器
- i_blocks：文件的数据库数
- i_flags：文件标志
- i_block：指向数据块的指针

i_size 存放的是以字节为单位的文件的有效长度，而 i_blocks 字段存放的是已分配给文件的数据块数（以 512 字节为单位）。

i_size 和 i_blocks 的值没有必然的联系。因为一个文件总是存放在整数块中，一个非空文件至少接受一个数据块且 i_size 可能小于 512*i_blocks。另一方面，一个文件可能包含有空洞，i_size 可能大于 512*i_blocks。

i_blocks 数 inode 结构中最重要的一个字段，因为它记录了内存中文件的块号（i_blocks 的下标）与磁盘逻辑块号（i_blocks 数组中的元素）之间的对应关系。

3.1.4 目录项

目录项记录了文件名称与索引节点之间的关系。在 linux 下，目录项也是做为一个文件来实现的，该文件的内容是一个个 ext2_dir_entry_2 的结构。

```
/*
 * The new version of the directory entry.  Since EXT2 structures are
 * stored in intel byte order, and the name_len field could never be
 * bigger than 255 chars, it's safe to reclaim the extra byte for the
 * file_type field.
```

```
 */
struct ext2_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

- inode:索引节点号
- rec_len：本目录项的长度。
- name_len：文件名长度
- file_type：文件类型
- name：保存文件名的空间

因为文件名的长度不一致，所以 ext2_dir_entry_2 在硬盘上的实际大小根据目录的长度而变，rec_len 记录了本目录项的实际大小。

3.2 VFS接口数据结构

3.1 节中分析了 EXT2 文件系统在硬盘上的相关结构，内核如何与这些数据结构建立联系呢，这些是通过 VFS 接口数据来实现的，VFSZ 中的接口数据只存在于内存中，为了与磁盘上的数据结构区分，称磁盘上对应的数据结构在内存中的实现为内存数据结构。

3.2.1 EXT2 超级块对象（内存）

VFS 超级块 super_block 有一个字段 s_fs_info 指向一个文件系统信息的数据结构。对于 EXT2 文件系统，该字段指向 ext2_sb_info 类型的结构：

include/linux/ext2_fs_sb.h

```
/*
 * second extended-fs super-block data in memory
 */
struct ext2_sb_info {
        unsigned long s_frag_size;    /* Size of a fragment in bytes */
```

```
unsigned long s_frags_per_block;/* Number of fragments per block */
unsigned long s_inodes_per_block;/* Number of inodes per block */
unsigned long s_frags_per_group;/* Number of fragments in a group */
unsigned long s_blocks_per_group;/* Number of blocks in a group */
unsigned long s_inodes_per_group;/* Number of inodes in a group */
unsigned long s_itb_per_group;   /* Number of inode table blocks per group */
unsigned long s_gdb_count;   /* Number of group descriptor blocks */
unsigned long s_desc_per_block; /* Number of group descriptors per block */
unsigned long s_groups_count;    /* Number of groups in the fs */
unsigned long s_overhead_last;   /* Last calculated overhead */
unsigned long s_blocks_last;      /* Last seen block count */
struct buffer_head * s_sbh; /* Buffer containing the super block */
struct ext2_super_block * s_es; /* Pointer to the super block in the buffer */
struct buffer_head ** s_group_desc;
unsigned long   s_mount_opt;
unsigned long s_sb_block;
uid_t s_resuid;
gid_t s_resgid;
unsigned short s_mount_state;
unsigned short s_pad;
int s_addr_per_block_bits;
int s_desc_per_block_bits;
int s_inode_size;
int s_first_ino;
spinlock_t s_next_gen_lock;
u32 s_next_generation;
unsigned long s_dir_count;
u8 *s_debts;
struct percpu_counter s_freeblocks_counter;
```

```
struct percpu_counter s_freeinodes_counter;
struct percpu_counter s_dirs_counter;
struct blockgroup_lock *s_blockgroup_lock;
/* root of the per fs reservation window tree */
spinlock_t s_rsv_window_lock;
struct rb_root s_rsv_window_root;
struct ext2_reserve_window_node s_rsv_window_head;
};
```

同磁盘超级块的字段

s_indoes_per_block：每块可以包含的 inode 数目

s_itb_per_group：块组中 indoe 表占用的块数。

s_gdb_count：块组中组描述符占用的块数。

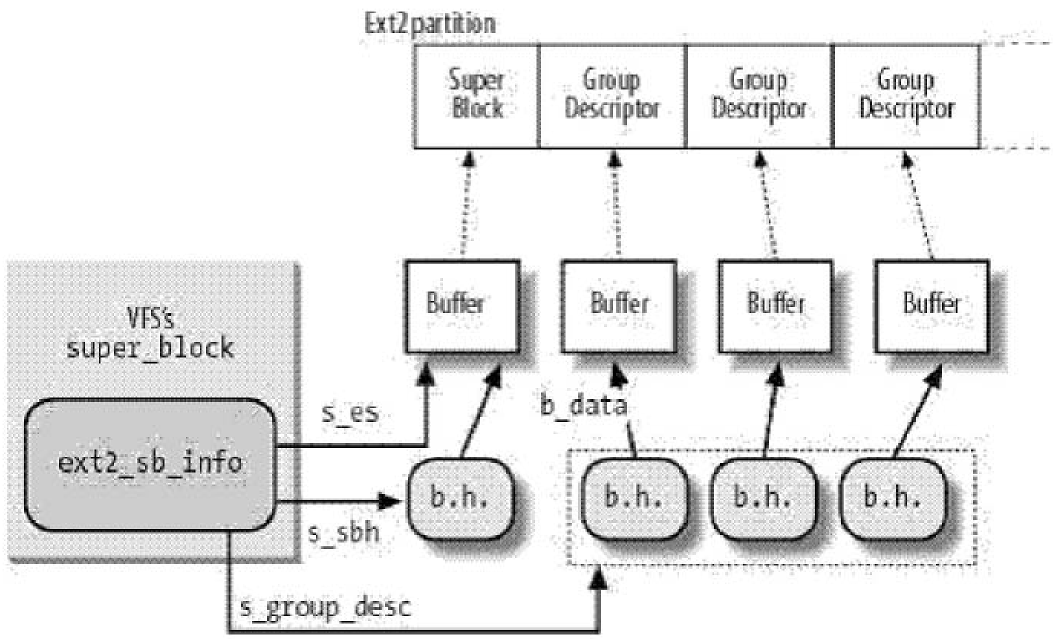
s_desc_per_block：每块中可以包含的组描述符数目

s_groups_count：文件系统中的块组数

s_sbh：指向包含超级块的块缓存

s_es：指向磁盘上的超级块结构

s_group_desc：指向包含组描述符的缓存



3.2.2 EXT2 索引节点对象（内存）

```
/*
 * second extended file system inode data in memory
 */
struct ext2_inode_info {
        __le32  i_data[15];
        __u32   i_flags;
        __u32   i_faddr;
        __u8    i_frag_no;
        __u8    i_frag_size;
        __u16   i_state;
        __u32   i_file_acl;
        __u32   i_dir_acl;
        __u32   i_dtime;

        /*
         * i_block_group is the number of the block group which contains
         * this file's inode.  Constant across the lifetime of the inode,
         * it is used for making block allocation decisions - we try to
         * place a file's data blocks near its inode block, and new inodes
         * near to their parent directory's inode.
         */
        __u32   i_block_group;

        /* block reservation info */
        struct ext2_block_alloc_info *i_block_alloc_info;

        __u32   i_dir_start_lookup;
#ifdef CONFIG_EXT2_FS_XATTR
        /*
```

```
         * Extended attributes can be read independently of the main file
         * data. Taking i_mutex even when reading would cause contention
         * between readers of EAs and writers of regular file data, so
         * instead we synchronize on xattr_sem when reading or changing
         * EAs.
         */
        struct rw_semaphore xattr_sem;
#endif
        rwlock_t i_meta_lock;

        /*
         * truncate_mutex is for serialising ext2_truncate() against
         * ext2_getblock().   It also protects the internals of the inode's
         * reservation data structures: ext2_reserve_window and
         * ext2_reserve_window_node.
         */
        struct mutex truncate_mutex;
        struct inode       vfs_inode;
        struct list_head i_orphan;    /* unlinked but open inodes */
};
```

- i_data：与磁盘索引节点中 i_block 对应。
- i_block_group：包含该索引节点的块组号

4. ext2 辅助操作函数

4.1 ext2_fill_super函数

ext2_fill_super 函数分配必要的数据结构，并从磁盘中读入磁盘超级块对象用来初始化 super_block。

```
static int ext2_fill_super(struct super_block *sb, void *data, int silent)
```

```
{
        struct buffer_head * bh;
        struct ext2_sb_info * sbi;
        struct ext2_super_block * es;
        struct inode *root;
        unsigned long block;
        unsigned long sb_block = get_sb_block(&data);
```

sb_block 超级块的起始块号

```
        unsigned long logic_sb_block;
        unsigned long offset = 0;
        unsigned long def_mount_opts;
        long ret = -EINVAL;
        int blocksize = BLOCK_SIZE;
        int db_count;
        int i, j;
        __le32 features;
        int err;

        sbi = kzalloc(sizeof(*sbi), GFP_KERNEL);
```

分配 ext2 超级块内存数据结构

```
        if (!sbi)
                return -ENOMEM;

        sbi->s_blockgroup_lock =
                kzalloc(sizeof(struct blockgroup_lock), GFP_KERNEL);
        if (!sbi->s_blockgroup_lock) {
                kfree(sbi);
                return -ENOMEM;
        }
        sb->s_fs_info = sbi;
```

初始化 sb 的 s_fs_info 指针

```
sbi->s_sb_block = sb_block;

/*
 * See what the current blocksize for the device is, and
 * use that as the blocksize.   Otherwise (or if the blocksize
 * is smaller than the default) use the default.
 * This is important for devices that have a hardware
 * sectorsize that is larger than the default.
 */
blocksize = sb_min_blocksize(sb, BLOCK_SIZE);
if (!blocksize) {
        ext2_msg(sb, KERN_ERR, "error: unable to set blocksize");
        goto failed_sbi;
}

/*
 * If the superblock doesn't start on a hardware sector boundary,
 * calculate the offset.
 */
if (blocksize != BLOCK_SIZE) {
        logic_sb_block = (sb_block*BLOCK_SIZE) / blocksize;
        offset = (sb_block*BLOCK_SIZE) % blocksize;
} else {
        logic_sb_block = sb_block;
}
```

首先确定设备上记录块的大小。EXT2 文件系统的记录块一般大小是 1K 字节，但是为提高读写效率也可以采用 2K 字节或者 4K 字节。内核中有一个以主设备号为下标的指针数组 hardsect_size。如果这个数组中设置了值，且大于 BLOCK_SIZE，则以该值为准。

超级块通常是设备上的 1 号记录块（即第二个记录块），所以 sb_block 设置为 1，在记录块大小为 BLOCK_SIZE 的设备上其逻辑块号

logic_sb_block 也为 1。但在记录块大于 BLOCK_SIZE 的设备上，由于超级块的大小仍为 BLOCK_SIZE，就要通过计算确定其所在的记录块，以及在块内的偏移。此时超级块虽然称为“块”，但实际上自身记录块中的一部分。确定了这两个参数后，就可以通过 sb_bread 将超级块读入内存了。

```

	if (!(bh = sb_bread(sb, logic_sb_block))) {
		ext2_msg(sb, KERN_ERR, "error: unable to read superblock");
		goto failed_sbi;
	}
```

调用 sb_bread 在缓冲区中分配一个缓冲区和缓冲区首部。然后从磁盘读入超级块在缓冲区中。

```
	es = (struct ext2_super_block *) (((char *)bh->b_data) + offset);
```

es 指向磁盘超级块在内存中的地址。

```
	sbi->s_es = es;
```

设置 sbi 的 s_es 字段。

```
	sb->s_magic = le16_to_cpu(es->s_magic);

	if (sb->s_magic != EXT2_SUPER_MAGIC)
		goto cantfind_ext2;

	/* Set defaults before we parse the mount options */
	def_mount_opts = le32_to_cpu(es->s_default_mount_opts);
	if (def_mount_opts & EXT2_DEFM_DEBUG)
		set_opt(sbi->s_mount_opt, DEBUG);
	if (def_mount_opts & EXT2_DEFM_BSDGROUPS)
		set_opt(sbi->s_mount_opt, GRPID);
	if (def_mount_opts & EXT2_DEFM_UID16)
		set_opt(sbi->s_mount_opt, NO_UID32);
#ifdef CONFIG_EXT2_FS_XATTR
	if (def_mount_opts & EXT2_DEFM_XATTR_USER)
		set_opt(sbi->s_mount_opt, XATTR_USER);
```

```
#endif
#ifdef CONFIG_EXT2_FS_POSIX_ACL
	if (def_mount_opts & EXT2_DEFM_ACL)
		set_opt(sbi->s_mount_opt, POSIX_ACL);
#endif

	if (le16_to_cpu(sbi->s_es->s_errors) == EXT2_ERRORS_PANIC)
		set_opt(sbi->s_mount_opt, ERRORS_PANIC);
	else if (le16_to_cpu(sbi->s_es->s_errors) == EXT2_ERRORS_CONTINUE)
		set_opt(sbi->s_mount_opt, ERRORS_CONT);
	else
		set_opt(sbi->s_mount_opt, ERRORS_RO);
	sbi->s_resuid = le16_to_cpu(es->s_def_resuid);
	sbi->s_resgid = le16_to_cpu(es->s_def_resgid);

	set_opt(sbi->s_mount_opt, RESERVATION);

	if (!parse_options((char *) data, sb))
		goto failed_mount;

	sb->s_flags = (sb->s_flags & ~MS_POSIXACL) |
		((EXT2_SB(sb)->s_mount_opt & EXT2_MOUNT_POSIX_ACL) ?
		 MS_POSIXACL : 0);

	ext2_xip_verify_sb(sb); /* see if bdev supports xip, unset
				    EXT2_MOUNT_XIP if not */

	if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV &&
		(EXT2_HAS_COMPAT_FEATURE(sb, ~0U) ||
		 EXT2_HAS_RO_COMPAT_FEATURE(sb, ~0U) ||
		 EXT2_HAS_INCOMPAT_FEATURE(sb, ~0U)))
```

```
		ext2_msg(sb, KERN_WARNING,
			"warning: feature flags set on rev 0 fs, "
			"running e2fsck is recommended");
	/*
	 * Check feature flags regardless of the revision level, since we
	 * previously didn't change the revision level when setting the flags,
	 * so there is a chance incompat flags are set on a rev 0 filesystem.
	 */
	features = EXT2_HAS_INCOMPAT_FEATURE(sb, ~EXT2_FEATURE_INCOMPAT_SUPP);
	if (features) {
		ext2_msg(sb, KERN_ERR,	"error: couldn't mount because of "
		       "unsupported optional features (%x)",
			le32_to_cpu(features));
		goto failed_mount;
	}
	if (!(sb->s_flags & MS_RDONLY) &&
	    (features = EXT2_HAS_RO_COMPAT_FEATURE(sb, ~EXT2_FEATURE_RO_COMPAT_SUPP))){
		ext2_msg(sb, KERN_ERR, "error: couldn't mount RDWR because of "
		       "unsupported optional features (%x)",
		       le32_to_cpu(features));
		goto failed_mount;
	}
	blocksize = BLOCK_SIZE << le32_to_cpu(sbi->s_es->s_log_block_size);
```

计算真正的块大小。

```

	if (ext2_use_xip(sb) && blocksize != PAGE_SIZE) {
		if (!silent)
			ext2_msg(sb, KERN_ERR,
```

```
                    "error: unsupported blocksize for xip");
            goto failed_mount;
    }

    /* If the blocksize doesn't match, re-read the thing.. */
    if (sb->s_blocksize != blocksize) {
            brelse(bh);

            if (!sb_set_blocksize(sb, blocksize)) {
                    ext2_msg(sb, KERN_ERR, "error: blocksize is too small");
                    goto failed_sbi;
            }
```

如果之前设置的块大小与真实的块大小不等，则重新设置 sb->s_blocksize 字段。

```

            logic_sb_block = (sb_block*BLOCK_SIZE) / blocksize;
            offset = (sb_block*BLOCK_SIZE) % blocksize;
            bh = sb_bread(sb, logic_sb_block);
```

重新读入块。

```
            if(!bh) {
                    ext2_msg(sb, KERN_ERR, "error: couldn't read"
                            "superblock on 2nd try");
                    goto failed_sbi;
            }
            es = (struct ext2_super_block *) (((char *)bh->b_data) + offset);
            sbi->s_es = es;
            if (es->s_magic != cpu_to_le16(EXT2_SUPER_MAGIC)) {
                    ext2_msg(sb, KERN_ERR, "error: magic mismatch");
                    goto failed_mount;
            }
```

设置相应字段。

```
907     }
```

885-907 行是对记录块大小的修正。前面已经确定了记录块的大小，但那未必是第一手资料。现在有了来至设备超级块的的信息，该信息更为准确，应该以此为准。

现在有一个问题了，既然原来的参数不对，那么根据不正确的参数读入的超级块为何是正确的呢？既然读入长度超级块时正确的，何必有重读一次呢？

上述问题的原因在于，在 sb_block 等于 1 的前提下计算出来的 logic_sb_block 和 offset 只有两种结果。当 sb->s_blocksize 大于 BLOCK_SIZE 时，logic_sb_block 总是 0 而 offset 为 BLOCK_SIZE,而与 sb->s_blocksize 的具体数值无关。当 sb->s_blocksize 等于 BLOCK_SIZE 时，logic_sb_block 为 1，offset 为 0。所以，只要记录块大小等于 BLOCK_SIZE 时超级块存放在第二块上，而在记录块大小大于 BLOCK_SIZE 时，则除将超级块的放在第二块的开头处外再在第一块中偏移为 BLOCK_SIZE 处放上一个副本，就不会报错了。这里重新读一遍，只不过是让缓冲区中含有整个记录块，而不只是超级块而已。

```
908
909     sb->s_maxbytes = ext2_max_size(sb->s_blocksize_bits);
910
911     if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV) {
912         sbi->s_inode_size = EXT2_GOOD_OLD_INODE_SIZE;
913         sbi->s_first_ino = EXT2_GOOD_OLD_FIRST_INO;
914     } else {
915         sbi->s_inode_size = le16_to_cpu(es->s_inode_size);
916         sbi->s_first_ino = le32_to_cpu(es->s_first_ino);
917         if ((sbi->s_inode_size < EXT2_GOOD_OLD_INODE_SIZE) ||
918             !is_power_of_2(sbi->s_inode_size) ||
919             (sbi->s_inode_size > blocksize)) {
920             ext2_msg(sb, KERN_ERR,
921                 "error: unsupported inode size: %d",
922                 sbi->s_inode_size);
923             goto failed_mount;
924         }
```

```
	}

	sbi->s_frag_size = EXT2_MIN_FRAG_SIZE <<
				   le32_to_cpu(es->s_log_frag_size);
	if (sbi->s_frag_size == 0)
		goto cantfind_ext2;
	sbi->s_frags_per_block = sb->s_blocksize / sbi->s_frag_size;

	sbi->s_blocks_per_group = le32_to_cpu(es->s_blocks_per_group);
	sbi->s_frags_per_group = le32_to_cpu(es->s_frags_per_group);
	sbi->s_inodes_per_group = le32_to_cpu(es->s_inodes_per_group);

	if (EXT2_INODE_SIZE(sb) == 0)
		goto cantfind_ext2;
	sbi->s_inodes_per_block = sb->s_blocksize / EXT2_INODE_SIZE(sb);
	if (sbi->s_inodes_per_block == 0 || sbi->s_inodes_per_group == 0)
		goto cantfind_ext2;
	sbi->s_itb_per_group = sbi->s_inodes_per_group /
					sbi->s_inodes_per_block;
	sbi->s_desc_per_block = sb->s_blocksize /
					sizeof (struct ext2_group_desc);
	sbi->s_sbh = bh;
```

设置 s_sbh 指针指向超级块的缓存。

```
	sbi->s_mount_state = le16_to_cpu(es->s_state);
	sbi->s_addr_per_block_bits =
		ilog2 (EXT2_ADDR_PER_BLOCK(sb));
	sbi->s_desc_per_block_bits =
		ilog2 (EXT2_DESC_PER_BLOCK(sb));

```

```
if (sb->s_magic != EXT2_SUPER_MAGIC)
        goto cantfind_ext2;

if (sb->s_blocksize != bh->b_size) {
        if (!silent)
                ext2_msg(sb, KERN_ERR, "error: unsupported blocksize");
        goto failed_mount;
}

if (sb->s_blocksize != sbi->s_frag_size) {
        ext2_msg(sb, KERN_ERR,
                "error: fragsize %lu != blocksize %lu"
                "(not supported yet)",
                sbi->s_frag_size, sb->s_blocksize);
        goto failed_mount;
}

if (sbi->s_blocks_per_group > sb->s_blocksize * 8) {
        ext2_msg(sb, KERN_ERR,
                "error: #blocks per group too big: %lu",
                sbi->s_blocks_per_group);
        goto failed_mount;
}
if (sbi->s_frags_per_group > sb->s_blocksize * 8) {
        ext2_msg(sb, KERN_ERR,
                "error: #fragments per group too big: %lu",
                sbi->s_frags_per_group);
        goto failed_mount;
}
if (sbi->s_inodes_per_group > sb->s_blocksize * 8) {
        ext2_msg(sb, KERN_ERR,
```

```
                "error: #inodes per group too big: %lu",
                sbi->s_inodes_per_group);
            goto failed_mount;
    }

    if (EXT2_BLOCKS_PER_GROUP(sb) == 0)
        goto cantfind_ext2;
    sbi->s_groups_count = ((le32_to_cpu(es->s_blocks_count) -
                le32_to_cpu(es->s_first_data_block) - 1)
                    / EXT2_BLOCKS_PER_GROUP(sb)) + 1;
```

计算所有文件系统中块组的总数目。计算公式：

（总的块数—起始的数据块块号-1）/块组中包含的块数 + 1

```
    db_count = (sbi->s_groups_count + EXT2_DESC_PER_BLOCK(sb) - 1) /
               EXT2_DESC_PER_BLOCK(sb);
```

db_count 表示存放 s_groups_count个块组描述符所需要的记录块数。

```
    sbi->s_group_desc = kmalloc (db_count * sizeof (struct buffer_head *), GFP_KERNEL);
```

初始化 s_group_sesc 函数指针。指向一个数组，数组中的每个元素都是一个类型为 struct buffer_head*的指针，数组的元素个数为 db_count。

```
    if (sbi->s_group_desc == NULL) {
        ext2_msg(sb, KERN_ERR, "error: not enough memory");
        goto failed_mount;
    }
    bgl_lock_init(sbi->s_blockgroup_lock);
    sbi->s_debts = kcalloc(sbi->s_groups_count, sizeof(*sbi->s_debts), GFP_KERNEL);
    if (!sbi->s_debts) {
        ext2_msg(sb, KERN_ERR, "error: not enough memory");
        goto failed_mount_group_desc;
    }
    for (i = 0; i < db_count; i++) {
```

```
        block = descriptor_loc(sb, logic_sb_block, i);
```

得到块组的起始块号。

```
        sbi->s_group_desc[i] = sb_bread(sb, block);
        if (!sbi->s_group_desc[i]) {
            for (j = 0; j < i; j++)
                brelse (sbi->s_group_desc[j]);
            ext2_msg(sb, KERN_ERR,
                "error: unable to read group descriptors");
            goto failed_mount_group_desc;
        }
    }
```

叫块组描述符缓存指针保存在 s_group_desc 数组中。

```
    if (!ext2_check_descriptors (sb)) {
        ext2_msg(sb, KERN_ERR, "group descriptors corrupted");
        goto failed_mount2;
    }
    sbi->s_gdb_count = db_count;
    get_random_bytes(&sbi->s_next_generation, sizeof(u32));
    spin_lock_init(&sbi->s_next_gen_lock);

    /* per fileystem reservation list head & lock */
    spin_lock_init(&sbi->s_rsv_window_lock);
    sbi->s_rsv_window_root = RB_ROOT;
    /*
     * Add a single, static dummy reservation to the start of the
     * reservation window list --- it gives us a placeholder for
     * append-at-start-of-list which makes the allocation logic
     * _much_ simpler.
     */
    sbi->s_rsv_window_head.rsv_start = EXT2_RESERVE_WINDOW_NOT_ALLOCATED;
```

```
sbi->s_rsv_window_head.rsv_end = EXT2_RESERVE_WINDOW_NOT_ALLOCATED;
sbi->s_rsv_window_head.rsv_alloc_hit = 0;
sbi->s_rsv_window_head.rsv_goal_size = 0;
ext2_rsv_window_add(sb, &sbi->s_rsv_window_head);

err = percpu_counter_init(&sbi->s_freeblocks_counter,
                ext2_count_free_blocks(sb));
if (!err) {
    err = percpu_counter_init(&sbi->s_freeinodes_counter,
                ext2_count_free_inodes(sb));
}
if (!err) {
    err = percpu_counter_init(&sbi->s_dirs_counter,
                ext2_count_dirs(sb));
}
if (err) {
    ext2_msg(sb, KERN_ERR, "error: insufficient memory");
    goto failed_mount3;
}
/*
 * set up enough so that it can read an inode
 */
```

超级块已安装了足够的信息，可以读取节点了。

```
sb->s_op = &ext2_sops;
sb->s_export_op = &ext2_export_ops;
sb->s_xattr = ext2_xattr_handlers;
root = ext2_iget(sb, EXT2_ROOT_INO);
```

读取根节点。根节点的索引节点号是固定的，在 EXT2 中为 EXT2_ROOT_INO（2）。

```
if (IS_ERR(root)) {
```

```
		ret = PTR_ERR(root);
		goto failed_mount3;
	}
	if (!S_ISDIR(root->i_mode) || !root->i_blocks || !root->i_size) {
		iput(root);
		ext2_msg(sb, KERN_ERR, "error: corrupt root inode, run e2fsck");
		goto failed_mount3;
	}
```

根索引节点必须表示满足如下条件：

- 必须是一个目录索引
- 文件的块数不能为 0
- 文件大小不能为 0

```

	sb->s_root = d_alloc_root(root);
```

初始化根目录项。调用 d_alloc_root 函数，完成根目录项的设置。

```
/**
 * d_alloc_root - allocate root dentry
 * @root_inode: inode to allocate the root for
 *
 * Allocate a root ("/") dentry for the inode given. The inode is
 * instantiated and returned. %NULL is returned if there is insufficient
 * memory or the inode passed is %NULL.
 */

struct dentry * d_alloc_root(struct inode * root_inode)
{
	struct dentry *res = NULL;

	if (root_inode) {
		static const struct qstr name = { .name = "/", .len = 1 };
```

设置根目录的名称为“/”，名称长度为 1。

```

	res = d_alloc(NULL, &name);
	if (res) {
		res->d_sb = root_inode->i_sb;
		res->d_parent = res;
```

根目录的父目录指向自己。

```
		d_instantiate(res, root_inode);
```

安装根目录。

```
	}
}
return res;
}
EXPORT_SYMBOL(d_alloc_root);
```

```
if (!sb->s_root) {
	iput(root);
	ext2_msg(sb, KERN_ERR, "error: get root inode failed");
	ret = -ENOMEM;
	goto failed_mount3;
}
if (EXT2_HAS_COMPAT_FEATURE(sb,
EXT3_FEATURE_COMPAT_HAS_JOURNAL))
	ext2_msg(sb, KERN_WARNING,
		"warning: mounting ext3 filesystem as ext2");
ext2_setup_super (sb, es, sb->s_flags & MS_RDONLY);
return 0;

cantfind_ext2:
if (!silent)
```

```
		ext2_msg(sb, KERN_ERR,
			"error: can't find an ext2 filesystem on dev %s.",
			sb->s_id);
	goto failed_mount;
failed_mount3:
	percpu_counter_destroy(&sbi->s_freeblocks_counter);
	percpu_counter_destroy(&sbi->s_freeinodes_counter);
	percpu_counter_destroy(&sbi->s_dirs_counter);
failed_mount2:
	for (i = 0; i < db_count; i++)
		brelse(sbi->s_group_desc[i]);
failed_mount_group_desc:
	kfree(sbi->s_group_desc);
	kfree(sbi->s_debts);
failed_mount:
	brelse(bh);
failed_sbi:
	sb->s_fs_info = NULL;
	kfree(sbi->s_blockgroup_lock);
	kfree(sbi);
	return ret;
}
```

总结一下 ext2_fill_super 所作的工作如下：

- 分配一个 ext2_sb_info（ext2 超级块内存对象），将其地址保存到 sb 的 s_fs_info 字段中。
- 调用 sb_bread 在缓冲区页中分配一个缓冲区和缓冲区首部，然后从磁盘读入超级块存放在缓存中。
- 根据超级块的内容，调整之前设置的块记录大小，有必要的话重新读入超级块所在的记录块。
- 分配一个数组用来存放缓冲区首部指针，这些指针用来指向内存中的组描述符。该数组存放在 s_group_desc 中。

- 重复调用 sb_bread 分配缓冲区，从磁盘读入包含 EXT2 组描述符的块，把缓冲区首部地址存放在上一部得到的 s_group_sesc 数组中。
- 安装 sb 中的 s_op，s_export_op 指针，准备好根目录，分配一个索引节点和目录项对象。

4.2 inode的分配 ext2_new_inode

ext2 函数创建目标文件在存储设备上的索引节点和在内存中的 inode 结构。

/fs/ext2/ialloc.c

```
struct inode *ext2_new_inode(struct inode *dir, int mode)
{
        struct super_block *sb;
        struct buffer_head *bitmap_bh = NULL;
        struct buffer_head *bh2;
        int group, i;
        ino_t ino = 0;
        struct inode * inode;
        struct ext2_group_desc *gdp;
        struct ext2_super_block *es;
        struct ext2_inode_info *ei;
        struct ext2_sb_info *sbi;
        int err;

        sb = dir->i_sb;
        inode = new_inode(sb);
```

调用如下的 new_inode 函数，分配一个 inode 结构。

```
struct inode *new_inode(struct super_block *sb)
{
        /*
         * On a 32bit, non LFS stat() call, glibc will generate an EOVERFLOW
```

```
         * error if st_ino won't fit in target struct field. Use 32bit counter
         * here to attempt to avoid that.
         */
        static unsigned int last_ino;
        struct inode *inode;

        spin_lock_prefetch(&inode_lock);

        inode = alloc_inode(sb);
        if (inode) {
                spin_lock(&inode_lock);
                __inode_add_to_lists(sb, NULL, inode);
                inode->i_ino = ++last_ino;
```

设置 inode 的索引节点号

```
                inode->i_state = 0;
                spin_unlock(&inode_lock);
        }
        return inode;
}
```

现代的块设备通常都是很大的。为了提高访问效率，就把存储介质划分为许多“块组”。一般来说，文件就应该与其所在目录存储在同一块组中，这样可以提高访问效率。另一方面，文件的内容和文件的索引节点也应该存储在同一块组中，所以在创建文件系统（格式化）时已经注意到了每个块组在索引节点和记录数块数量之间的比例。这个比例来自于统计信息，取决于平均的文件大小。此外，每一个块组中平均有多少个目录，也就是所每个目录平均有多少文件，大致也有个比例。所以，如果要创建的是文件，就应该首先考虑将它的索引节点分配在与其所在目录所处的块组中。如果要创建的是目录，则要考虑将来是否能够将其属下的文件都容纳在同一块组中，所以应该找一个器空闲节点的数量超过真个设备上的平均值这么一个块组，而不惜离开器父节点所在的块组。下面看这是如何实现的。

```
        if (!inode)
```

```
		return ERR_PTR(-ENOMEM);

	ei = EXT2_I(inode);
	sbi = EXT2_SB(sb);
	es = sbi->s_es;
```

ei 指向 ext2 在内存中的 inode。

```
	if (S_ISDIR(mode)) {
		if (test_opt(sb, OLDALLOC))
			group = find_group_dir(sb, dir);
		else
			group = find_group_orlov(sb, dir);
	} else
		group = find_group_other(sb, dir);
```

根据所要创建的是目录还是文件，调用不同的函数计算 inode 应该在的块组。具体的函数下面介绍。

```

	if (group == -1) {
		err = -ENOSPC;
		goto fail;
	}

	for (i = 0; i < sbi->s_groups_count; i++) {
		gdp = ext2_get_group_desc(sb, group, &bh2);
```

根据块组号，得到组描述符 gdp。

```
		brelse(bitmap_bh);
		bitmap_bh = read_inode_bitmap(sb, group);
```

得到本块组的索引位图记录块。

```
		if (!bitmap_bh) {
			err = -EIO;
			goto fail;
		}
```

```
        ino = 0;

repeat_in_this_group:
        ino = ext2_find_next_zero_bit((unsigned long *)bitmap_bh->b_data,
                                      EXT2_INODES_PER_GROUP(sb), ino);
```

查找位图中第一个下一个为 0 的 bit 位，并记录其与索引位图起始位的距离 ino。

```
        if (ino >= EXT2_INODES_PER_GROUP(sb)) {
```

如果 ino 大于库组中位图的数目

```
                /*
                 * Rare race: find_group_xx() decided that there were
                 * free inodes in this group, but by the time we tried
                 * to allocate one, they're all gone.   This can also
                 * occur because the counters which find_group_orlov()
                 * uses are approximate.   So just go and search the
                 * next block group.
                 */
                if (++group == sbi->s_groups_count)
                        group = 0;
                continue;
```

本组中找不到合适的，则循环前进到下一个块组。

```
        }
        if (ext2_set_bit_atomic(sb_bgl_lock(sbi, group),
                                ino, bitmap_bh->b_data)) {
```

Inode 被别人抢占，则继续到下一个块组。

```
                /* we lost this inode */
                if (++ino >= EXT2_INODES_PER_GROUP(sb)) {
                        /* this group is exhausted, try next group */
                        if (++group == sbi->s_groups_count)
                                group = 0;
```

```
                        continue;
                }
                /* try to find free inode in the same group */
                goto repeat_in_this_group;
        }
        goto got;
```

运行导次，则找到了 inode 索引。

```
    }

/*
     * Scanned all blockgroups.
     */
    err = -ENOSPC;
    goto fail;
got:
    mark_buffer_dirty(bitmap_bh);
```

由于修改了位图索引，必须将其缓存设置为 dirty，以便回写到磁盘。

```
    if (sb->s_flags & MS_SYNCHRONOUS)
        sync_dirty_buffer(bitmap_bh);
    brelse(bitmap_bh);

    ino += group * EXT2_INODES_PER_GROUP(sb) + 1;
```

计算 inode 索引的在文件系统中的全局编号。

```
    if (ino < EXT2_FIRST_INO(sb) || ino > le32_to_cpu(es->s_inodes_count)) {
        ext2_error (sb, "ext2_new_inode",
                "reserved inode or inode > inodes count - "
                "block_group = %d,inode=%lu", group,
                (unsigned long) ino);
        err = -EIO;
        goto fail;
```

```
}

percpu_counter_add(&sbi->s_freeinodes_counter, -1);
if (S_ISDIR(mode))
	percpu_counter_inc(&sbi->s_dirs_counter);

spin_lock(sb_bgl_lock(sbi, group));
le16_add_cpu(&gdp->bg_free_inodes_count, -1);
```

减少块组中空闲节点的计数。

```
if (S_ISDIR(mode)) {
	if (sbi->s_debts[group] < 255)
		sbi->s_debts[group]++;
	le16_add_cpu(&gdp->bg_used_dirs_count, 1);
```

增加块组中目录项的计数。

```
} else {
	if (sbi->s_debts[group])
		sbi->s_debts[group]--;
}
spin_unlock(sb_bgl_lock(sbi, group));

sb->s_dirt = 1;
mark_buffer_dirty(bh2);
```

设置超级块、组描述符所在缓存为脏。

```
inode->i_uid = current_fsuid();
if (test_opt (sb, GRPID))
	inode->i_gid = dir->i_gid;
else if (dir->i_mode & S_ISGID) {
	inode->i_gid = dir->i_gid;
	if (S_ISDIR(mode))
		mode |= S_ISGID;
} else
```

```
		inode->i_gid = current_fsgid();
	inode->i_mode = mode;

	inode->i_ino = ino;
	inode->i_blocks = 0;
```

设置 inode 索引节点号，文件的块数。

```
	inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME_SEC;
	memset(ei->i_data, 0, sizeof(ei->i_data));
```

设置 i_data（记录文件块号的数组）为空。

```
	ei->i_flags =
		ext2_mask_flags(mode, EXT2_I(dir)->i_flags & EXT2_FL_INHERITED);
	ei->i_faddr = 0;
	ei->i_frag_no = 0;
	ei->i_frag_size = 0;
	ei->i_file_acl = 0;
	ei->i_dir_acl = 0;
	ei->i_dtime = 0;
	ei->i_block_alloc_info = NULL;
	ei->i_block_group = group;
	ei->i_dir_start_lookup = 0;
	ei->i_state = EXT2_STATE_NEW;
	ext2_set_inode_flags(inode);
	spin_lock(&sbi->s_next_gen_lock);
	inode->i_generation = sbi->s_next_generation++;
	spin_unlock(&sbi->s_next_gen_lock);
	if (insert_inode_locked(inode) < 0) {
		err = -EINVAL;
		goto fail_drop;
	}
```

```

	dquot_initialize(inode);
	err = dquot_alloc_inode(inode);
	if (err)
		goto fail_drop;

	err = ext2_init_acl(inode, dir);
	if (err)
		goto fail_free_drop;

	err = ext2_init_security(inode,dir);
	if (err)
		goto fail_free_drop;

	mark_inode_dirty(inode);
	ext2_debug("allocating inode %lu\n", inode->i_ino);
	ext2_preread_inode(inode);
	return inode;
```

返回 inode。

```

fail_free_drop:
	dquot_free_inode(inode);

fail_drop:
	dquot_drop(inode);
	inode->i_flags |= S_NOQUOTA;
	inode->i_nlink = 0;
	unlock_new_inode(inode);
	iput(inode);
	return ERR_PTR(err);

```

```
fail:
        make_bad_inode(inode);
        iput(inode);
        return ERR_PTR(err);
}
```

4.2.1 ext2_get_group_desc

根据块组号的到块组描述符结构。

```
struct ext2_group_desc * ext2_get_group_desc(struct super_block * sb,
                                             unsigned int block_group,
                                             struct buffer_head ** bh)
```

参数

- sb：超级块
- block_group：块组号
- buffer_head：bh（out 参数），记录 block_group 对应的组描述符的缓存指针。

```
{
        unsigned long group_desc;
        unsigned long offset;
        struct ext2_group_desc * desc;
        struct ext2_sb_info *sbi = EXT2_SB(sb);

        if (block_group >= sbi->s_groups_count) {
                ext2_error (sb, "ext2_get_group_desc",
                            "block_group >= groups_count - "
                            "block_group = %d, groups_count = %lu",
                            block_group, sbi->s_groups_count);

                return NULL;
        }

```

```
	group_desc = block_group >> EXT2_DESC_PER_BLOCK_BITS(sb);
```

记录块的索引

```
	offset = block_group & (EXT2_DESC_PER_BLOCK(sb) - 1);
```

记录块中的偏移

```
	if (!sbi->s_group_desc[group_desc]) {
		ext2_error (sb, "ext2_get_group_desc",
			    "Group descriptor not loaded - "
			    "block_group = %d, group_desc = %lu, desc = %lu",
			     block_group, group_desc, offset);
		return NULL;
	}

	desc = (struct ext2_group_desc *) sbi->s_group_desc[group_desc]->b_data;
	if (bh)
		*bh = sbi->s_group_desc[group_desc];
	return desc + offset;
}
```

4.2.2 read_inode_bitmap

根据块组号读取该块组中索引节点位图所在的记录块。

```
static struct buffer_head *
read_inode_bitmap(struct super_block * sb, unsigned long block_group)
{
	struct ext2_group_desc *desc;
	struct buffer_head *bh = NULL;

	desc = ext2_get_group_desc(sb, block_group, NULL);
```

得到组描述符

```
	if (!desc)
```

```
        goto error_out;

    bh = sb_bread(sb, le32_to_cpu(desc->bg_inode_bitmap));
```

根据组描述符中 bg_inode_bitmap 字段记录的块号，读取索引节点位图记录块。

```
    if (!bh)
        ext2_error(sb, "read_inode_bitmap",
                "Cannot read inode bitmap - "
                "block_group = %lu, inode_bitmap = %u",
                block_group, le32_to_cpu(desc->bg_inode_bitmap));
error_out:
    return bh;
}
```

4.2.3 find_group_orlov

如果创建的是目录，则通过 find_group_orlov 函数查找分配索引节点的块组号。

```
static int find_group_orlov(struct super_block *sb, struct inode *parent)
{
    int parent_group = EXT2_I(parent)->i_block_group;
    struct ext2_sb_info *sbi = EXT2_SB(sb);
    struct ext2_super_block *es = sbi->s_es;
    int ngroups = sbi->s_groups_count;
    int inodes_per_group = EXT2_INODES_PER_GROUP(sb);
    int freei;
    int avefreei;
    int free_blocks;
    int avefreeb;
    int blocks_per_dir;
    int ndirs;
    int max_debt, max_dirs, min_blocks, min_inodes;
```

```
int group = -1, i;
struct ext2_group_desc *desc;

freei = percpu_counter_read_positive(&sbi->s_freeinodes_counter);
avefreei = freei / ngroups;
```

avefreei：每组的平均的空闲 inode

```
free_blocks = percpu_counter_read_positive(&sbi->s_freeblocks_counter);
avefreeb = free_blocks / ngroups;
```

avefreeb：每组平均空闲块数

```
ndirs = percpu_counter_read_positive(&sbi->s_dirs_counter);
```

```

if ((parent == sb->s_root->d_inode) ||
    (EXT2_I(parent)->i_flags & EXT2_TOPDIR_FL)) {
```

父目录为根节点：以文件系统根 root 为父目录的目录项应该分散在各个块组中。

```
    struct ext2_group_desc *best_desc = NULL;
    int best_ndir = inodes_per_group;
    int best_group = -1;

    get_random_bytes(&group, sizeof(group));
    parent_group = (unsigned)group % ngroups;
    for (i = 0; i < ngroups; i++) {
        group = (parent_group + i) % ngroups;
        desc = ext2_get_group_desc (sb, group, NULL);
        if (!desc || !desc->bg_free_inodes_count)
            continue;
        if (le16_to_cpu(desc->bg_used_dirs_count) >= best_ndir)
            continue;
```

```
                        if (le16_to_cpu(desc->bg_free_inodes_count) < avefreei)
                                continue;
                        if (le16_to_cpu(desc->bg_free_blocks_count) < avefreeb)
                                continue;
                        best_group = group;
                        best_ndir = le16_to_cpu(desc->bg_used_dirs_count);
                        best_desc = desc;
                }
```

遍历块组，在其中查找一个：

- 空闲的索引节点大于平均索引节点
- 空闲的块数比平均的块数大。
- 已经使用的目录数比平均的目录树小。

```
                if (best_group >= 0) {
                        desc = best_desc;
                        group = best_group;
                        goto found;
                }
                goto fallback;
        }

        if (ndirs == 0)
                ndirs = 1;	/* percpu_counters are approximate... */

        blocks_per_dir = (le32_to_cpu(es->s_blocks_count)-free_blocks) / ndirs;

        max_dirs = ndirs / ngroups + inodes_per_group / 16;
        min_inodes = avefreei - inodes_per_group / 4;
        min_blocks = avefreeb - EXT2_BLOCKS_PER_GROUP(sb) / 4;

        max_debt = EXT2_BLOCKS_PER_GROUP(sb) / max(blocks_per_dir, BLOCK_COST);
```

```
	if (max_debt * INODE_COST > inodes_per_group)
		max_debt = inodes_per_group / INODE_COST;
	if (max_debt > 255)
		max_debt = 255;
	if (max_debt == 0)
		max_debt = 1;

	for (i = 0; i < ngroups; i++) {
		group = (parent_group + i) % ngroups;
		desc = ext2_get_group_desc (sb, group, NULL);
		if (!desc || !desc->bg_free_inodes_count)
			continue;
		if (sbi->s_debts[group] >= max_debt)
			continue;
		if (le16_to_cpu(desc->bg_used_dirs_count) >= max_dirs)
			continue;
		if (le16_to_cpu(desc->bg_free_inodes_count) < min_inodes)
			continue;
		if (le16_to_cpu(desc->bg_free_blocks_count) < min_blocks)
			continue;
		goto found;
	}
```

遍历块组，找一个：

- 负债小
- 使用的目录少
- 空闲节点大于 min_inodes
- 空闲块数大于 min_blocks

```

fallback:
	for (i = 0; i < ngroups; i++) {
```

```
		group = (parent_group + i) % ngroups;
		desc = ext2_get_group_desc (sb, group, NULL);
		if (!desc || !desc->bg_free_inodes_count)
			continue;
		if (le16_to_cpu(desc->bg_free_inodes_count) >= avefreei)
			goto found;
	}

```

退一步计划，从包含父目录中的块组开始，选择第一个满足：空闲索引节点数比平均空闲索引节点数大的块组。

```
	if (avefreei) {
		/*
		 * The free-inodes counter is approximate, and for really small
		 * filesystems the above test can fail to find any blockgroups
		 */
		avefreei = 0;
		goto fallback;
	}
```

如果退一步计划失败，且 avefreei 不为 0，则设置 avefreei 为 0，跳转到退一步计划中。

```

	return -1;

found:
	return group;
}
```

4.2.4 find_group_other

如果创建的是文件，则调用 find_group_other 来计算块组号。

```
static int find_group_other(struct super_block *sb, struct inode *parent)
{
```

```
int parent_group = EXT2_I(parent)->i_block_group;
int ngroups = EXT2_SB(sb)->s_groups_count;
struct ext2_group_desc *desc;
int group, i;

/*
 * Try to place the inode in its parent directory
 */
group = parent_group;
desc = ext2_get_group_desc (sb, group, NULL);
if (desc && le16_to_cpu(desc->bg_free_inodes_count) &&
        le16_to_cpu(desc->bg_free_blocks_count))
    goto found;
```

父目录中查找。

```

/*
 * We're going to place this inode in a different blockgroup from its
 * parent.   We want to cause files in a common directory to all land in
 * the same blockgroup.   But we want files which are in a different
 * directory which shares a blockgroup with our parent to land in a
 * different blockgroup.
 *
 * So add our directory's i_ino into the starting point for the hash.
 */
group = (group + parent->i_ino) % ngroups;

/*
 * Use a quadratic hash to find a group with a free inode and some
 * free blocks.
 */
for (i = 1; i < ngroups; i <<= 1) {
```

```
        group += i;
        if (group >= ngroups)
            group -= ngroups;
        desc = ext2_get_group_desc (sb, group, NULL);
        if (desc && le16_to_cpu(desc->bg_free_inodes_count) &&
                le16_to_cpu(desc->bg_free_blocks_count))
            goto found;
    }
```

如果父目录中没有找到，则使用指数跳跃方式查找。

```

    /*
     * That failed: try linear search for a free inode, even if that group
     * has no free blocks.
     */
    group = parent_group;
    for (i = 0; i < ngroups; i++) {
        if (++group >= ngroups)
            group = 0;
        desc = ext2_get_group_desc (sb, group, NULL);
        if (desc && le16_to_cpu(desc->bg_free_inodes_count))
            goto found;
    }
```

线性查找。

```

    return -1;

found:
    return group;
}
```

4.3 ext2_get_block 数据块分配

当内核要分配一个数据块来保存 EXT2 普通文件的数据时，就调用 ext2_get_block 函数。

4.3.1 文件内块号到磁盘设备上块号的映射

从文件内块号到设备上块号的映射，最简单最迅速的方法是使用一个以文件内块号为下标的线性数组，并且将数组置与索引节点 inode 结构中。这样就需要很大的数组，从而是索引节点和 inode 的结构变大。

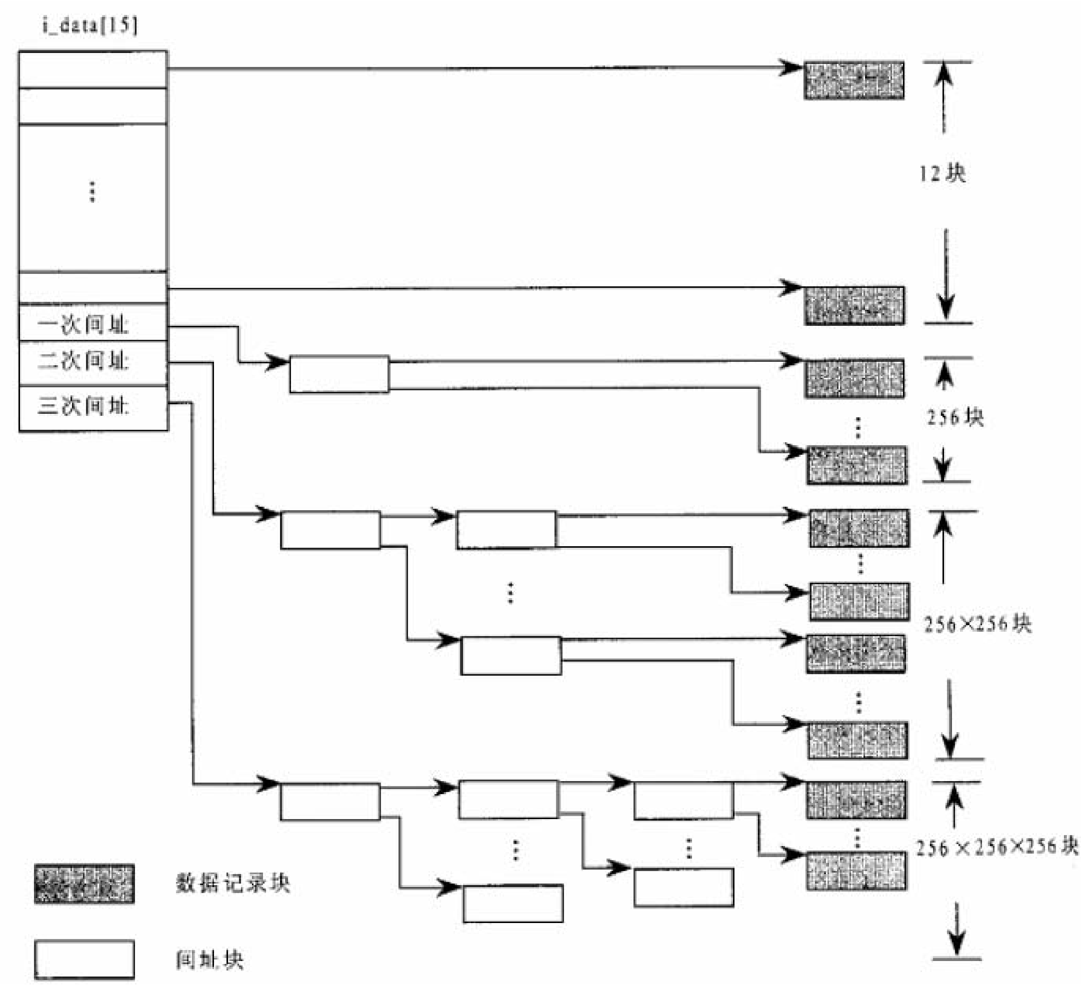
另一种方法是使用间接寻址，也就是将上述的数组分块放在设备上本来可用来存储数据的若干记录块中，而将这些记录块的块号放在索引节点 inode 结构中。这些记录块虽然在设备上的数据区中，却并不构成文件的本身内容，而只是一些管理信息。由于索引节点和（inode 结构）应该是固定大小的，所以当文件较大时还要将这种间接寻址的结构框架做成树状或链状，这样才能随着文件本身的大小而扩展器容量，但是这种方式解决了容量的问题，但是降低了运行效率。

基于上述考虑，UNIX 早期的文件系统采用了一种折衷的方法，使用直接与间接相结合。基本思想是将文件的记录块分为几个部分来实现。

第一部分是个以文件内块号为下标的数组，这是采用直接映射的部分，对于较小的文件这一部分就够用了。由于根据文件内块号就可以在 inode 结构的数组中直接找到相应的设备上的块号，所以效率高。

至于较大的文件，开头的部分可以直接映射，但是当文件超出这部分的容量时，超出的那部分就必须采用间接寻址了。

EXT2 文件系统的直接映射的这部分为 12 个记录块。前面看到 ext2_inode_info 结构中，有个大小为 15 的整型数组 i_data，其开头 12 个元素即用于此目的。当文件大小超过这部分内容时，数组中第 13 个元素指向一个记录块，这个记录块的内容也是一个整型数组，其中的每一个元素都指向一个设备上的记录块。第 14 个元素用于二次间接寻址，第 15 个元素用于三次间接寻址。



多重间接示意图

4.3.2 ext2_get_block

```
720 int ext2_get_block(struct inode *inode, sector_t iblock, struct buffer_head *bh_result, int create)
```

参数

Iblock：所处理的记录块在文件中的逻辑块号

create：是否需要创建的标志

bh_result：目标块的缓存区头。

```
721 {
722         unsigned max_blocks = bh_result->b_size >> inode->i_blkbits;
```

计算最大的块数

```
	int ret = ext2_get_blocks(inode, iblock, max_blocks,
				bh_result, create);
```

调用 ext2_get_blocks 实现具体的功能。

```
	if (ret > 0) {
		bh_result->b_size = (ret << inode->i_blkbits);
		ret = 0;
	}
	return ret;

}
```

ext2_get_blocks

```
/*
 * Allocation strategy is simple: if we have to allocate something, we will
 * have to go the whole way to leaf. So let's do it before attaching anything
 * to tree, set linkage between the newborn blocks, write them if sync is
 * required, recheck the path, free and repeat if check fails, otherwise
 * set the last missing link (that will protect us from any truncate-generated
 * removals - all blocks on the path are immune now) and possibly force the
 * write on the parent block.
 * That has a nice additional property: no special recovery from the failed
 * allocations is needed - we simply release blocks and do not touch anything
 * reachable from inode.
 *
 * `handle' can be NULL if create == 0.
 *
 * return > 0, # of blocks mapped or allocated.
 * return = 0, if plain lookup failed.
 * return < 0, error case.
```

```
 */
static int ext2_get_blocks(struct inode *inode,
                        sector_t iblock, unsigned long maxblocks,
                        struct buffer_head *bh_result,
                        int create)
{
        int err = -EIO;
        int offsets[4];
        Indirect chain[4];
        Indirect *partial;
        ext2_fsblk_t goal;
        int indirect_blks;
        int blocks_to_boundary = 0;
        int depth;
        struct ext2_inode_info *ei = EXT2_I(inode);
        int count = 0;
        ext2_fsblk_t first_block = 0;

        depth =
ext2_block_to_path(inode,iblock,offsets,&blocks_to_boundary);
```

ext2_block_to_path 根据文件中的逻辑块号，完成两项任务：

- 计算映射的深度。
- 计算每一层映射中使用的位移量，即数组的下标，保存在 offsets 数组中。

```

        if (depth == 0)
                return (err);
```

如果 ext2_block_to_path 返回 0，表示出错。

```

        partial = ext2_get_branch(inode, depth, offsets, chain, &err);
```

ext2_get_branch 从磁盘上逐层读入用于间接映射的记录块。

ext2_get_branch 返回值有两种情况：

- 如果顺利的完成了映射则返回值为 NULL。
- 如果在某一层上发现映射表内的相应项为 0，则说明这个表项（记录块）原来不存在，现在因为写操作而需要扩充文件的大小。此时返回的 Indirect 结构的指针，表示映射在此处“断裂”了。

```
/* Simplest case - block found, no allocation needed */
if (!partial) {
```

顺利完成映射

```
        first_block = le32_to_cpu(chain[depth - 1].key);
```

硬盘上起始块的块号

```
        clear_buffer_new(bh_result); /* What's this do? */
        count++;
        /*map more blocks*/
        while (count < maxblocks && count <= blocks_to_boundary) {
            ext2_fsblk_t blk;

            if (!verify_chain(chain, chain + depth - 1)) {
```

检测多层映射之间的映射关系是否正确

```
                /*
                 * Indirect block might be removed by
                 * truncate while we were reading it.
                 * Handling of that case: forget what we've
                 * got now, go to reread.
                 */
                err = -EAGAIN;
                count = 0;
                break;
            }
            blk = le32_to_cpu(*(chain[depth-1].p + count));
```

获得与起始索引出相差 count 个元素的索引号。

```
            if (blk == first_block + count)
```

```
621                 count++;
```

如果数组的两个下标之差与对应的数组元素之差相等，则 count++。

```
622             else
623                 break;
```

否则跳出循环。

```
624         }
```

605 到 624 层的循环，用来检测在文件中连续的块在磁盘上的逻辑块也连续的情况。

```
625         if (err != -EAGAIN)
626             goto got_it;
627     }
628
629     /* Next simple case - plain lookup or failed read of indirect block */
630     if (!create || err == -EIO)
631         goto cleanup;
632
633     mutex_lock(&ei->truncate_mutex);
634     /*
635      * If the indirect block is missing while we are reading
636      * the chain(ext3_get_branch() returns -EAGAIN err), or
637     * if the chain has been changed after we grab the semaphore,
638      * (either because another process truncated this branch, or
639      * another get_block allocated this branch) re-grab the chain to see if
640      * the request block has been allocated or not.
641      *
642      * Since we already block the truncate/other get_block
643      * at this point, we will have the current copy of the chain when we
644      * splice the branch into the tree.
645      */
646     if (err == -EAGAIN || !verify_chain(chain, partial)) {
```

```
647             while (partial > chain) {
648                     brelse(partial->bh);
649                     partial--;
650             }
651             partial = ext2_get_branch(inode, depth, offsets, chain, &err);
652             if (!partial) {
653                     count++;
654                     mutex_unlock(&ei->truncate_mutex);
655                     if (err)
656                             goto cleanup;
657                     clear_buffer_new(bh_result);
658                     goto got_it;
659             }
660     }
```

646-647 行：如果 chain 已经改变，需要重新调用 ext2_get_branch 来填充 chain。

```
661
662     /*
663      * Okay, we need to do block allocation.  Lazily initialize the block
664      * allocation info here if necessary
665     */
666     if (S_ISREG(inode->i_mode) && (!ei->i_block_alloc_info))
667             ext2_init_block_alloc_info(inode);
668
669     goal = ext2_find_goal(inode, iblock, partial);
```

为目标记录块分配一个建议的块号 goal

```
670
671     /* the number of blocks need to allocate for [d,t]indirect blocks */
672     indirect_blks = (chain + depth) - partial - 1;
```

中间路径需要分配的块的数目

```
673     /*
```

```
	 * Next look up the indirect map to count the totoal number of
	 * direct blocks to allocate for this branch.
	 */
	count = ext2_blks_to_allocate(partial, indirect_blks,
					maxblocks, blocks_to_boundary);
```

计算直接数据块的数目

```
	/*
	 * XXX ???? Block out ext2_truncate while we alter the tree
	 */
	err = ext2_alloc_branch(inode, indirect_blks, &count, goal,
				offsets + (partial - chain), partial);
```

ext2_alloc_branch 完成如下的功能：

- 设备上具体记录块的分配（目标记录块和可能用于间接映射的中间记录块）。
- 映射的建立

```

	if (err) {
		mutex_unlock(&ei->truncate_mutex);
		goto cleanup;
	}

	if (ext2_use_xip(inode->i_sb)) {
		/*
		 * we need to clear the block
		 */
		err = ext2_clear_xip_target (inode,
			le32_to_cpu(chain[depth-1].key));
		if (err) {
			mutex_unlock(&ei->truncate_mutex);
			goto cleanup;
		}
```

```
	}

	ext2_splice_branch(inode, iblock, partial, indirect_blks, count);
```

将断裂的键给安装上。

```
	mutex_unlock(&ei->truncate_mutex);
	set_buffer_new(bh_result);
got_it:
	map_bh(bh_result, inode->i_sb, le32_to_cpu(chain[depth-1].key));
	if (count > blocks_to_boundary)
		set_buffer_boundary(bh_result);
	err = count;
	/* Clean up and exit */
	partial = chain + depth - 1;	/* the whole chain */
cleanup:
	while (partial > chain) {
		brelse(partial->bh);
		partial--;
	}
	return err;
}
```

4.3.2.1 ext2_block_to_path

```
static int ext2_block_to_path(struct inode *inode,
			long i_block, int offsets[4], int *boundary)
{
	int ptrs = EXT2_ADDR_PER_BLOCK(inode->i_sb);
	int ptrs_bits = EXT2_ADDR_PER_BLOCK_BITS(inode->i_sb);
	const long direct_blocks = EXT2_NDIR_BLOCKS,
		indirect_blocks = ptrs,
		double_blocks = (1 << (ptrs_bits * 2));
```

```
int n = 0;
int final = 0;

if (i_block < 0) {
        ext2_msg(inode->i_sb, KERN_WARNING,
                "warning: %s: block < 0", __func__);
} else if (i_block < direct_blocks) {
        offsets[n++] = i_block;
        final = direct_blocks;
} else if ( (i_block -= direct_blocks) < indirect_blocks) {
        offsets[n++] = EXT2_IND_BLOCK;
        offsets[n++] = i_block;
        final = ptrs;
} else if ((i_block -= indirect_blocks) < double_blocks) {
        offsets[n++] = EXT2_DIND_BLOCK;
        offsets[n++] = i_block >> ptrs_bits;
        offsets[n++] = i_block & (ptrs - 1);
        final = ptrs;
} else if (((i_block -= double_blocks) >> (ptrs_bits * 2)) < ptrs) {
        offsets[n++] = EXT2_TIND_BLOCK;
        offsets[n++] = i_block >> (ptrs_bits * 2);
        offsets[n++] = (i_block >> ptrs_bits) & (ptrs - 1);
        offsets[n++] = i_block & (ptrs - 1);
        final = ptrs;
} else {
        ext2_msg(inode->i_sb, KERN_WARNING,
                "warning: %s: block is too big", __func__);
}
if (boundary)
        *boundary = final - 1 - (i_block & (ptrs - 1));

```

```
        return n;
}
```

该函数逻辑比较简单，根据文件中的块号计算在磁盘上映射的深度以及每层中对应数组的下标。

4.3.2.2 ext2_get_branch

在介绍该函数之前，先讲解一下对应的数据结构。

```
typedef struct {
        __le32   *p;
        __le32   key;
        struct buffer_head *bh;
} Indirect;
```

Indirect 结构用来记录映射的中间层。该结构根据 offset 来填充。

p 指向本层记录块映射表中相应的表项

key 是该表项的值

bh 指向缓冲区的指针

辅助函数 add_chain 建立链接，verify_chain 检测链接是否正确。

```
static inline void add_chain(Indirect *p, struct buffer_head *bh, __le32 *v)
{
        p->key = *(p->p = v);
        p->bh = bh;
}

static inline int verify_chain(Indirect *from, Indirect *to)
{
        while (from <= to && from->key == *from->p)
                from++;
        return (from > to);
}
```

```
static Indirect *ext2_get_branch(struct inode *inode,
```

```
                        int depth,
                        int *offsets,
                        Indirect chain[4],
                        int *err)
{
        struct super_block *sb = inode->i_sb;
        Indirect *p = chain;
        struct buffer_head *bh;

        *err = 0;
        /* i_data is not going away, no lock needed */
        add_chain (chain, NULL, EXT2_I(inode)->i_data + *offsets);
```

第一层映射的填充。

```
        if (!p->key)
                goto no_block;
```

如果下标对应的数组元素值为 0，表示链接断开。

```
        while (--depth) {
                bh = sb_bread(sb, le32_to_cpu(p->key));
```

读取中间目录的记录块

```
                if (!bh)
                        goto failure;
                read_lock(&EXT2_I(inode)->i_meta_lock);
                if (!verify_chain(chain, p))
                        goto changed;
```

如果间接映射的记录块有变，跳转到 changed。

```
                add_chain(++p, bh, (__le32*)bh->b_data + *++offsets);
```

建立映射关系

```
                read_unlock(&EXT2_I(inode)->i_meta_lock);
                if (!p->key)
                        goto no_block;
```

如果对以的记录为 0，跳转到 no_block。

```
	}
	return NULL;
changed:
	read_unlock(&EXT2_I(inode)->i_meta_lock);
	brelse(bh);
	*err = -EAGAIN;
	goto no_block;
failure:
	*err = -EIO;
no_block:
	return p;
}
```

4.3.2.3 ext2_find_goal

```
static inline ext2_fsblk_t ext2_find_goal(struct inode *inode, long block,
					  Indirect *partial)
{
	struct ext2_block_alloc_info *block_i;
	block_i = EXT2_I(inode)->i_block_alloc_info;
	/*
	 * try the heuristic for sequential allocation,
	 * failing that at least try to get decent locality.
	 */
	if (block_i && (block == block_i->last_alloc_logical_block + 1)
		&& (block_i->last_alloc_physical_block != 0)) {
		return block_i->last_alloc_physical_block + 1;
	}
```

参数 block 为文件内的逻辑块号，goal 用来返回所建议的设备上的块号。从本文件的角度当然希望所有的记录块在设备上都是连续的。为此目的，ext2_block_alloc_info 中的 last_alloc_logical_block 和 last_alloc_physical_block 用于次目的。last_alloc_logical_block 用于记录最后一次分配的逻辑块号，last_alloc_physical_block 用于记录设备上最后一次分配的块号。在正常的情况下对文件的扩充是顺序的，所以每次的文件内块号与前一次的连续，而理想的设备上块号也同样连续，二种平行的推进。当然，这只是从特定的文件角度提供的建议块号，能否实现还要看条件是否允许，不过内核会尽量满足要求，不能满足也会尽可能靠近建议值的块号分配。

但是，文件内逻辑块号也有可能不连续，也就是说文件的扩充是跳跃的，也的逻辑块号与文件原有的最后一个逻辑块号之间有“空洞”，这时调用 ext2_find_near 函数。

```
        return ext2_find_near(inode, partial);
```

调用 ext2_find_near 来完成分配块的查找。

```
}
```

ext2_find_near

```
static ext2_fsblk_t ext2_find_near(struct inode *inode, Indirect *ind)
{
        struct ext2_inode_info *ei = EXT2_I(inode);
        __le32 *start = ind->bh ? (__le32 *) ind->bh->b_data : ei->i_data;
        __le32 *p;
        ext2_fsblk_t bg_start;
        ext2_fsblk_t colour;

        /* Try to find previous block */
        for (p = ind->p - 1; p >= start; p--)
                if (*p)
                        return le32_to_cpu(*p);
```

首先将起点 start 设置成指向当前映射表的（映射过程中首次发现断裂的那个映射表）的起点，然后在当前映射表内往回搜索。如果要分配的是空洞后面的第一个记录块，那就需要往回找到空洞之前的表项所对应的物理块号，并以此为建议块号。当然，这个物理块号已经被使用，这个要求无法满足（记住，该函数给出的只是建议块号，便于内核在该块号的附件范围查找）。内核在分配物理记录块时会在位图中从这里开始往前搜索，就近分配物理记录块。

```

	/* No such thing, so let's try location of indirect block */
	if (ind->bh)
		return ind->bh->b_blocknr;
```

当空洞在间接映射表的开头处时，往回搜索在本映射中找不到空洞之前的表项，就以间接映射表本身所在的记录块号作为建议块号。

```

	/*
	 * It is going to be refered from inode itself? OK, just put it into
	 * the same cylinder group then.
	 */
	bg_start = ext2_group_first_block_no(inode->i_sb, ei->i_block_group);
	colour = (current->pid % 16) *
			(EXT2_BLOCKS_PER_GROUP(inode->i_sb) / 16);
	return bg_start + colour;
```

空洞在文件的开头，以索引节点所在块组的第一个数据记录块作为建议号。

```
}
```

4.3.2.4 ext2_blks_to_allocate

计算数据项需要分配的块数。

```
static int
ext2_blks_to_allocate(Indirect * branch, int k, unsigned long blks,
		int blocks_to_boundary)
```

```
{
	unsigned long count = 0;

	/*
	 * Simple case, [t,d]Indirect block(s) has not allocated yet
	 * then it's clear blocks on that path have not allocated
	 */
	if (k > 0) {
```

k>0 表示需要分配中间目录块。

```
		/* right now don't hanel cross boundary allocation */
		if (blks < blocks_to_boundary + 1)
			count += blks;
		else
			count += blocks_to_boundary + 1;
		return count;
```

返回直接分配的块数。

```
	}

	count++;
	while (count < blks && count <= blocks_to_boundary
		&& le32_to_cpu(*(branch[0].p + count)) == 0) {
		count++;
```

直接分配，返回连续分配的块数。

```
	}
	return count;
}
```

4.3.2.5 ext2_alloc_branch

```
static int ext2_alloc_branch(struct inode *inode,
			int indirect_blks, int *blks, ext2_fsblk_t goal,
			int *offsets, Indirect *branch)
```

参数：

- indirect_blks：表示还有几次映射需要建立，实际上就是需要分配几个记录块。
- branch：指向 chain[]中从映射断裂后开始的那一部分。
- offsets：指向 offsets 中的相应部分

```
{
        int blocksize = inode->i_sb->s_blocksize;
        int i, n = 0;
        int err = 0;
        struct buffer_head *bh;
        int num;
        ext2_fsblk_t new_blocks[4];
```

保存分配的块号

```
        ext2_fsblk_t current_block;

        num = ext2_alloc_blocks(inode, goal, indirect_blks,
                                *blks, new_blocks, &err);
        if (err)
                return err;

        branch[0].key = cpu_to_le32(new_blocks[0]);
        /*
         * metadata blocks and data blocks are allocated.
         */
        for (n = 1; n <= indirect_blks;  n++) {
                /*
                 * Get buffer_head for parent block, zero it out
                 * and set the pointer to new one, then send
                 * parent to disk.
                 */
```

```
bh = sb_getblk(inode->i_sb, new_blocks[n-1]);
```

在内存中分配缓存。

```
branch[n].bh = bh;
lock_buffer(bh);
memset(bh->b_data, 0, blocksize);
```

设置为 0

```
branch[n].p = (__le32 *) bh->b_data + offsets[n];
branch[n].key = cpu_to_le32(new_blocks[n]);
*branch[n].p = branch[n].key;
```

设置 Indirect 中的各个元素。

```
if ( n == indirect_blks) {
```

表示直接数据块

```
        current_block = new_blocks[n];
        /*
         * End of chain, update the last new metablock of
         * the chain to point to the new allocated
         * data blocks numbers
         */
        for (i=1; i < num; i++)
                *(branch[n].p + i) = cpu_to_le32(++current_block);
```

设置连续的数据块

```
}
set_buffer_uptodate(bh);
unlock_buffer(bh);
mark_buffer_dirty_inode(bh, inode);
/* We used to sync bh here if IS_SYNC(inode).
 * But we now rely upon generic_write_sync()
 * and b_inode_buffers.  But not for directories.
 */
if (S_ISDIR(inode->i_mode) && IS_DIRSYNC(inode))
        sync_dirty_buffer(bh);
```

设置相应项的标志，将内存中的修改写回硬盘。

```
	}
	*blks = num;
	return err;
}
```

4.3.2.6 ext2_alloc_blocks

```
/**
 *  ext2_alloc_blocks: multiple allocate blocks needed for a branch
 *  @indirect_blks: the number of blocks need to allocate for indirect
 *              blocks
 *
 *  @new_blocks: on return it will store the new block numbers for
 *  the indirect blocks(if needed) and the first direct block,
 *  @blks:   on return it will store the total number of allocated
 *          direct blocks
 */
static int ext2_alloc_blocks(struct inode *inode,
                        ext2_fsblk_t goal, int indirect_blks, int blks,
                        ext2_fsblk_t new_blocks[4], int *err)
{
```

参数:

goal：建议的块号

indirect_blks：中间目录需要分配的块数

blks：数据记录块的块数

new_blocks：指向 offset 数组中链接断开处的元素。

```
	int target, i;
	unsigned long count = 0;
	int index = 0;
	ext2_fsblk_t current_block = 0;
	int ret = 0;
```

```

/*
 * Here we try to allocate the requested multiple blocks at once,
 * on a best-effort basis.
 * To build a branch, we should allocate blocks for
 * the indirect blocks(if not allocated yet), and at least
 * the first direct block of this branch.  That's the
 * minimum number of blocks need to allocate(required)
 */
target = blks + indirect_blks;
```

target 总共需要分配的块数。

```

while (1) {
```

调用 ext2_new_blocks 循环分配需要的块数，知道满足要求为止。

```
        count = target;
```

count 表示需要分配的块数

```
        /* allocating blocks for indirect blocks and direct blocks */
        current_block = ext2_new_blocks(inode,goal,&count,err);
```

分配块数。返回值 current_block 表示分配块的起始块号。参数 count 是一个 in/out 参数，[in]count 表示需要分配的块数，[out]count 表示实际分配到的块数。

```
        if (*err)
                goto failed_out;

        target -= count;
```

target 表示剩余的需要分配的块数

```
        /* allocate blocks for indirect blocks */
        while (index < indirect_blks && count) {
                new_blocks[index++] = current_block++;
                count--;
        }
```

设置间接分配块的块号。

```
        if (count > 0)
            break;
    }
```

count>0 表示需要分配的块数以完成，跳出循环。

```
    /* save the new block number for the first direct block */
    new_blocks[index] = current_block;
```

数据块的起始块号

```
    /* total number of blocks allocated for direct blocks */
    ret = count;
```

数据块分配的总块数。

```
    *err = 0;
    return ret;
failed_out:
    for (i = 0; i <index; i++)
        ext2_free_blocks(inode, new_blocks[i], 1);
    return ret;
}
```

ext2_alloc_blocks 函数完成实际的分配任务，分配的最大的块的数目是 indirect_blks+blks，最小的块的数目是 indirect_blcks + 1，也就是该函数努力分配块，但是当分配不到更多的块时，至少要保证分配中间目录的块和 1 个数据块。

4.3.2.7 ext2_new_blocks

fs/ext2/balloc.c

分配时首先满足“客户”需要，如果所建议的记录块还空闲着就把它分配出去。否则，若果所建议的块号已经分配，就试图在它附件 32 个记录块中的范围内分配。还不行就向前在本块组的位图中搜索，先找位图中整个字节都是 8 个记录块的空闲区间，若达不到目的再降格以求。最后，如果实在找不到，则在整个设备的范围内寻找和分配。

```
/*
 * ext2_new_blocks() -- core block(s) allocation function
 * @inode:          file inode
 * @goal:           given target block(filesystem wide)
 * @count:          target number of blocks to allocate
 * @errp:           error code
 *
 * ext2_new_blocks uses a goal block to assist allocation.   If the goal is
 * free, or there is a free block within 32 blocks of the goal, that block
 * is allocated.   Otherwise a forward search is made for a free block; within
 * each block group the search first looks for an entire free byte in the block
 * bitmap, and then for any free bit if that fails.
 * This function also updates quota and i_blocks field.
 */
ext2_fsblk_t ext2_new_blocks(struct inode *inode, ext2_fsblk_t goal,
                    unsigned long *count, int *errp)
{
        struct buffer_head *bitmap_bh = NULL;
        struct buffer_head *gdp_bh;
        int group_no;
        int goal_group;
```

```
ext2_grpblk_t grp_target_blk;      /* blockgroup relative goal block */
ext2_grpblk_t grp_alloc_blk;       /* blockgroup-relative allocated block*/
ext2_fsblk_t ret_block;        /* filesyetem-wide allocated block */
int bgi;                       /* blockgroup iteration index */
int performed_allocation = 0;
ext2_grpblk_t free_blocks;    /* number of free blocks in a group */
struct super_block *sb;
struct ext2_group_desc *gdp;
struct ext2_super_block *es;
struct ext2_sb_info *sbi;
struct ext2_reserve_window_node *my_rsv = NULL;
struct ext2_block_alloc_info *block_i;
unsigned short windowsz = 0;
unsigned long ngroups;
unsigned long num = *count;
int ret;

*errp = -ENOSPC;
sb = inode->i_sb;
if (!sb) {
	printk("ext2_new_blocks: nonexistent device");
	return 0;
}

/*
 * Check quota for allocation of this block.
 */
ret = dquot_alloc_block(inode, num);
if (ret) {
	*errp = ret;
	return 0;
```

```
	}

	sbi = EXT2_SB(sb);
	es = EXT2_SB(sb)->s_es;
	ext2_debug("goal=%lu.\n", goal);
	/*
	 * Allocate a block from reservation only when
	 * filesystem is mounted with reservation(default,-o reservation), and
	 * it's a regular file, and
	 * the desired window size is greater than 0 (One could use ioctl
	 * command EXT2_IOC_SETRSVSZ to set the window size to 0 to turn off
	 * reservation on that particular file)
	 */
	block_i = EXT2_I(inode)->i_block_alloc_info;
	if (block_i) {
		windowsz = block_i->rsv_window_node.rsv_goal_size;
		if (windowsz > 0)
			my_rsv = &block_i->rsv_window_node;
	}

	if (!ext2_has_free_blocks(sbi)) {
		*errp = -ENOSPC;
		goto out;
	}

	/*
	 * First, test whether the goal block is free.
	 */
	if (goal < le32_to_cpu(es->s_first_data_block) ||
		goal >= le32_to_cpu(es->s_blocks_count))
```

```
		goal = le32_to_cpu(es->s_first_data_block);
```

按需调整 goal 号

```
	group_no = (goal - le32_to_cpu(es->s_first_data_block)) /
			EXT2_BLOCKS_PER_GROUP(sb);
	goal_group = group_no;
```

计算 goal 所在的块组号。

```
retry_alloc:
	gdp = ext2_get_group_desc(sb, group_no, &gdp_bh);
```

得到 goal 所在的块组的组描述符

```
	if (!gdp)
		goto io_error;

	free_blocks = le16_to_cpu(gdp->bg_free_blocks_count);
	/*
	 * if there is not enough free blocks to make a new resevation
	 * turn off reservation for this allocation
	 */
	if (my_rsv && (free_blocks < windowsz)
		&& (free_blocks > 0)
		&& (rsv_is_empty(&my_rsv->rsv_window)))
		my_rsv = NULL;

	if (free_blocks > 0) {
		grp_target_blk = ((goal - le32_to_cpu(es->s_first_data_block)) %
				EXT2_BLOCKS_PER_GROUP(sb));
		bitmap_bh = read_block_bitmap(sb, group_no);
		if (!bitmap_bh)
			goto io_error;
		grp_alloc_blk = ext2_try_to_allocate_with_rsv(sb, group_no,
					bitmap_bh, grp_target_blk,
					my_rsv, &num);
```

该函数进行块分配，返回值 grp_alloc_blk 表示分配的块号在本组内的序号。num 是一个 in/out 参数，[in]count 表示需要分配的块数目，[out]count 表示实际分配的块数。grp_target_blk 表示建议的分配块号。

```
		if (grp_alloc_blk >= 0)
			goto allocated;
	}
```

如果本组内有空闲块，则在本组内分配。

```

	ngroups = EXT2_SB(sb)->s_groups_count;
	smp_rmb();

	/*
	 * Now search the rest of the groups.  We assume that
	 * group_no and gdp correctly point to the last group visited.
	 */
	for (bgi = 0; bgi < ngroups; bgi++) {
		group_no++;
		if (group_no >= ngroups)
			group_no = 0;
		gdp = ext2_get_group_desc(sb, group_no, &gdp_bh);
		if (!gdp)
			goto io_error;

		free_blocks = le16_to_cpu(gdp->bg_free_blocks_count);
		/*
		 * skip this group if the number of
		 * free blocks is less than half of the reservation
		 * window size.
		 */
		if (my_rsv && (free_blocks <= (windowsz/2)))
			continue;
```

```
		brelse(bitmap_bh);
		bitmap_bh = read_block_bitmap(sb, group_no);
		if (!bitmap_bh)
			goto io_error;
		/*
		 * try to allocate block(s) from this group, without a goal(-1).
		 */
		grp_alloc_blk = ext2_try_to_allocate_with_rsv(sb, group_no,
					bitmap_bh, -1, my_rsv, &num);
```

该函数进行块分配，返回值 grp_alloc_blk 表示分配的块号在本组内的序号。num 是一个 in/out 参数，[in]count 表示需要分配的块数目，[out]count 表示实际分配的块数。grp_target_blk 表示建议的分配块号。

```
		if (grp_alloc_blk >= 0)
			goto allocated;
	}
```

本组内分配失败，则遍历所有块组分配。

```
	/*
	 * We may end up a bogus ealier ENOSPC error due to
	 * filesystem is "full" of reservations, but
	 * there maybe indeed free blocks avaliable on disk
	 * In this case, we just forget about the reservations
	 * just do block allocation as without reservations.
	 */
	if (my_rsv) {
		my_rsv = NULL;
		windowsz = 0;
		group_no = goal_group;
		goto retry_alloc;
		goto retry_alloc;
	}
```

```
	/* No space left on the device */
	*errp = -ENOSPC;
	goto out;

allocated:

	ext2_debug("using block group %d(%d)\n",
			group_no, gdp->bg_free_blocks_count);

	ret_block = grp_alloc_blk + ext2_group_first_block_no(sb, group_no);
```

返回分配的块的起始块号

```

	if (in_range(le32_to_cpu(gdp->bg_block_bitmap), ret_block, num) ||
	    in_range(le32_to_cpu(gdp->bg_inode_bitmap), ret_block, num) ||
	    in_range(ret_block, le32_to_cpu(gdp->bg_inode_table),
		      EXT2_SB(sb)->s_itb_per_group) ||
	    in_range(ret_block + num - 1,
le32_to_cpu(gdp->bg_inode_table),
		      EXT2_SB(sb)->s_itb_per_group)) {
		ext2_error(sb, "ext2_new_blocks",
			    "Allocating block in system zone - "
			    "blocks from "E2FSBLK", length %lu",
			    ret_block, num);
		/*
		 * ext2_try_to_allocate marked the blocks we allocated as in
		 * use.  So we may want to selectively mark some of the blocks
		 * as free
		 */
		goto retry_alloc;
	}
```

块的合理性检查

```
	performed_allocation = 1;

	if (ret_block + num - 1 >= le32_to_cpu(es->s_blocks_count)) {
		ext2_error(sb, "ext2_new_blocks",
			    "block("E2FSBLK") >= blocks count(%d) - "
			    "block_group = %d, es == %p ", ret_block,
			le32_to_cpu(es->s_blocks_count), group_no, es);
		goto out;
	}

	group_adjust_blocks(sb, group_no, gdp, gdp_bh, -num);
```

调整相关数据结构中空闲块的记录字段

```
	percpu_counter_sub(&sbi->s_freeblocks_counter, num);

	mark_buffer_dirty(bitmap_bh);
```

设置 bitmap 对应的缓存区为脏，将数据回写到硬盘。

```
	if (sb->s_flags & MS_SYNCHRONOUS)
		sync_dirty_buffer(bitmap_bh);

	*errp = 0;
	brelse(bitmap_bh);
	dquot_free_block(inode, *count-num);
	*count = num;
	return ret_block;
```

返回实际分配的块数目，连续块的起始块号。

```
io_error:
	*errp = -EIO;
out:
	/*
```

```
         * Undo the block allocation
         */
        if (!performed_allocation)
                dquot_free_block(inode, *count);
        brelse(bitmap_bh);
        return 0;
}
```

4.3.2.8 ext2_try_to_allocate_with_rsv

```
/**
 * ext2_try_to_allocate_with_rsv()
 * @sb:                 superblock
 * @group:              given allocation block group
 * @bitmap_bh:          bufferhead holds the block bitmap
 * @grp_goal:           given target block within the group
 * @count:              target number of blocks to allocate
 * @my_rsv:             reservation window
 *
 * This is the main function used to allocate a new block and its reservation
 * window.
 *
 * Each time when a new block allocation is need, first try to allocate from
 * its own reservation.   If it does not have a reservation window, instead of
 * looking for a free bit on bitmap first, then look up the reservation list to
 * see if it is inside somebody else's reservation window, we try to allocate a
 * reservation window for it starting from the goal first. Then do the block
 * allocation within the reservation window.
```

```
 *
 * This will avoid keeping on searching the reservation list again and
 * again when somebody is looking for a free block (without
 * reservation), and there are lots of free blocks, but they are all
 * being reserved.
 *
 * We use a red-black tree for the per-filesystem reservation list.
 */
static ext2_grpblk_t
ext2_try_to_allocate_with_rsv(struct super_block *sb, unsigned int group,
                        struct buffer_head *bitmap_bh, ext2_grpblk_t grp_goal,
                        struct ext2_reserve_window_node * my_rsv,
                        unsigned long *count)
{
        ext2_fsblk_t group_first_block, group_last_block;
        ext2_grpblk_t ret = 0;
        unsigned long num = *count;

/*
         * we don't deal with reservation when
         * filesystem is mounted without reservation
         * or the file is not a regular file
         * or last attempt to allocate a block with reservation turned on failed
         */
        if (my_rsv == NULL) {
                return ext2_try_to_allocate(sb, group, bitmap_bh,
                                                grp_goal, count, NULL);
        }
        /*
         * grp_goal is a group relative block number (if there is a goal)
         * 0 <= grp_goal < EXT2_BLOCKS_PER_GROUP(sb)
```

```
	 * first block is a filesystem wide block number
	 * first block is the block number of the first block in this group
	 */
	group_first_block = ext2_group_first_block_no(sb, group);
	group_last_block = group_first_block + (EXT2_BLOCKS_PER_GROUP(sb) - 1);
	/*
	 * Basically we will allocate a new block from inode's reservation
	 * window.
	 *
	 * We need to allocate a new reservation window, if:
	 * a) inode does not have a reservation window; or
	 * b) last attempt to allocate a block from existing reservation
	 *    failed; or
	 * c) we come here with a goal and with a reservation window
	 *
	 * We do not need to allocate a new reservation window if we come here
	 * at the beginning with a goal and the goal is inside the window, or
	 * we don't have a goal but already have a reservation window.
	 * then we could go to allocate from the reservation window directly.
	 */
	while (1) {
		if (rsv_is_empty(&my_rsv->rsv_window) || (ret < 0) ||
			!goal_in_my_reservation(&my_rsv->rsv_window,
						grp_goal, group, sb)) {
			if (my_rsv->rsv_goal_size < *count)
				my_rsv->rsv_goal_size = *count;
			ret = alloc_new_reservation(my_rsv, grp_goal, sb,
							group, bitmap_bh);
			if (ret < 0)
```

```
                        break;          /* failed */

                if (!goal_in_my_reservation(&my_rsv->rsv_window,
                                        grp_goal, group, sb))
                        grp_goal = -1;
        } else if (grp_goal >= 0) {
                int curr = my_rsv->rsv_end -
                                (grp_goal + group_first_block) + 1;

                if (curr < *count)
                        try_to_extend_reservation(my_rsv, sb,
                                        *count - curr);
        }

        if ((my_rsv->rsv_start > group_last_block) ||
                        (my_rsv->rsv_end < group_first_block)) {
                rsv_window_dump(&EXT2_SB(sb)->s_rsv_window_root, 1);
                BUG();
        }
        ret = ext2_try_to_allocate(sb, group, bitmap_bh, grp_goal,
                                &num, &my_rsv->rsv_window);
```

调用 ext2_try_to_allocate 分配块。

```
        if (ret >= 0) {
                my_rsv->rsv_alloc_hit += num;
                *count = num;
                break;                  /* succeed */
        }
        num = *count;
    }
    return ret;
}
```

4.3.2.9 ext2_try_to_allocate

```
static int
ext2_try_to_allocate(struct super_block *sb, int group,
                        struct buffer_head *bitmap_bh, ext2_grpblk_t grp_goal,
                        unsigned long *count,
                        struct ext2_reserve_window *my_rsv)
{
        ext2_fsblk_t group_first_block;
                ext2_grpblk_t start, end;
        unsigned long num = 0;

        /* we do allocation within the reservation window if we have a window */
        if (my_rsv) {
                group_first_block = ext2_group_first_block_no(sb, group);
                if (my_rsv->_rsv_start >= group_first_block)
                        start = my_rsv->_rsv_start - group_first_block;
                else
                        /* reservation window cross group boundary */
                        start = 0;
                end = my_rsv->_rsv_end - group_first_block + 1;
                if (end > EXT2_BLOCKS_PER_GROUP(sb))
                        /* reservation window crosses group boundary */
                        end = EXT2_BLOCKS_PER_GROUP(sb);
                if ((start <= grp_goal) && (grp_goal < end))
                        start = grp_goal;
                else
                        grp_goal = -1;
        } else {
```

设置块位图中对应的 bit 位，表示分配块号。

```
		if (grp_goal > 0)
			start = grp_goal;
		else
			start = 0;
		end = EXT2_BLOCKS_PER_GROUP(sb);
	}

	BUG_ON(start > EXT2_BLOCKS_PER_GROUP(sb));

repeat:
	if (grp_goal < 0) {
		grp_goal = find_next_usable_block(start, bitmap_bh, end);
		if (grp_goal < 0)
			goto fail_access;
		if (!my_rsv) {
			int i;

			for (i = 0; i < 7 && grp_goal > start &&
					!ext2_test_bit(grp_goal - 1,
							bitmap_bh->b_data);
					i++, grp_goal--)
				;
		}
	}
	start = grp_goal;

	if (ext2_set_bit_atomic(sb_bgl_lock(EXT2_SB(sb), group), grp_goal,
						bitmap_bh->b_data)) {
```

设置块位图中对应的 bit 位，表示分配块号。

```
		/*
		 * The block was allocated by another thread, or it was
```

```
		 * allocated and then freed by another thread
		 */
		start++;
		grp_goal++;
		if (start >= end)
			goto fail_access;
		goto repeat;
	}
	num++;
	grp_goal++;
	while (num < *count && grp_goal < end
		&& !ext2_set_bit_atomic(sb_bgl_lock(EXT2_SB(sb), group),
					grp_goal, bitmap_bh->b_data)) {
		num++;
		grp_goal++;
	}
```

设置块位图中的 bit 位表示分配块

```
	*count = num;
	return grp_goal - num;
fail_access:
	*count = num;
	return -1;
}
```

5. ext2 文件系统的具体实现

本小结中以 ext2 文件系统为例，具体分析文件系统的实现，主要内容包含如下几部分：

- 从文件路径到索引节点(inode)
- 文件系统的安装与拆卸
- 文件的打开与关闭

- 文件的读与写
- 文件的映射

5.1 文件路径到索引节点

文件打开、读写都涉及到从文件路径到索引节点的转换，所以这个部分是其他内容的基础。

ext2 文件系统在硬盘上的布局如下：

启动块	块组 0	块组 1	……	块组 n

每一个块组的划分如下：

超级块	组描述符	数据位图	inode 位图	inode 表	数据块

在 linux 下面，目录也是作为一种特定的文件来实现的。

5.1.1 相关数据结构

5.1.1.1 ext2 文件节点ext2_inode

vfs 的 inode 结构是内存中用来表示索引节点的数据结构，它要兼容各种文件系统，与磁盘上 indoe 的索引节点结构不同。对于 ext2 文件系统而言，磁盘上 indoe 结构就是 ext2_inode。

include/linux/ext2_fs.h

```
/*
 * Structure of an inode on the disk
 */
struct ext2_inode {
        __le16   i_mode;         /* File mode */
        __le16   i_uid;          /* Low 16 bits of Owner Uid */
        __le32   i_size;         /* Size in bytes */
        __le32   i_atime;        /* Access time */
        __le32   i_ctime;        /* Creation time */
        __le32   i_mtime;        /* Modification time */
        __le32   i_dtime;        /* Deletion Time */
        __le16   i_gid;          /* Low 16 bits of Group Id */
        __le16   i_links_count;  /* Links count */
```

```
	__le32	i_blocks;	/* Blocks count */
	__le32	i_flags;	/* File flags */
	union {
		struct {
			__le32	l_i_reserved1;
		} linux1;
		struct {
			__le32	h_i_translator;
		} hurd1;
		struct {
			__le32	m_i_reserved1;
		} masix1;
	} osd1;				/* OS dependent 1 */
	__le32	i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
	__le32	i_generation;	/* File version (for NFS) */
	__le32	i_file_acl; /* File ACL */
	__le32	i_dir_acl;	/* Directory ACL */
	__le32	i_faddr;	/* Fragment address */
	union {
		struct {
			__u8	l_i_frag;	/* Fragment number */
			__u8	l_i_fsize;	/* Fragment size */
			__u16	i_pad1;
			__le16	l_i_uid_high;	/* these 2 fields	*/
			__le16	l_i_gid_high;	/* were reserved2[0] */
			__u32	l_i_reserved2;
		} linux2;
		struct {
			__u8	h_i_frag;	/* Fragment number */
			__u8	h_i_fsize;	/* Fragment size */
			__le16	h_i_mode_high;
```

```
                        __le16    h_i_uid_high;
                        __le16    h_i_gid_high;
                        __le32    h_i_author;
                } hurd2;
                struct {
                        __u8      m_i_frag;     /* Fragment number */
                        __u8      m_i_fsize;    /* Fragment size */
                        __u16     m_pad1;
                        __u32     m_i_reserved2[2];
                } masix2;
        } osd2;                     /* OS dependent 2 */
};
```

```
/*
 * Constants relative to the data blocks
 */
#define EXT2_NDIR_BLOCKS            12
#define EXT2_IND_BLOCK              EXT2_NDIR_BLOCKS
#define EXT2_DIND_BLOCK             (EXT2_IND_BLOCK + 1)
#define EXT2_TIND_BLOCK             (EXT2_DIND_BLOCK + 1)
#define EXT2_N_BLOCKS               (EXT2_TIND_BLOCK + 1)
```

重点看 i_block 结构，i_block 是从文件逻辑块号到磁盘块号转换的关键点。

i_block 是一个数组，通过文件的逻辑块号作为数组下表，索引到该下表的元素，也就是逻辑块号对应的磁盘块号。（为了尽量简单，这里没有考虑间接索引的情况）。

```
/*
 * The new version of the directory entry.  Since EXT2 structures are
 * stored in intel byte order, and the name_len field could never be
 * bigger than 255 chars, it's safe to reclaim the extra byte for the
 * file_type field.
 */
```

```
struct ext2_dir_entry_2 {
        __le32  inode;                /* Inode number */
        __le16  rec_len;              /* Directory entry length */
        __u8    name_len;             /* Name length */
        __u8    file_type;
        char    name[EXT2_NAME_LEN];  /* File name */
};
```

- inode:索引节点号
- rec_len：文件实际长度
- name_len：文件名长度
- file_type：文件类型
- name：保存文件名的空间

5.1.2 文件路径到索引节点的实现概述

以/home/zenhumany/test.txt 为例，看看是如何通过路径名找到 test.txt 对应的索引节点的。

首先，系统先根据“/”的索引节点（根 inode，至于这一个索引节点哪来的，下面介绍）可以找到“/”在硬盘上的内容，根据索引节点中的 i_block 数组，可以找到文件在硬盘上的所有块号，也就得到了该文件的所有内容。

找到硬盘上的内容后，将其读入内存。前面说过，目录也是文件，是一种特需的文件，其中保存的是目录项。对 ext2 文件系统而言，里面保存的是类型为 ext2_dir_entry_2 的实例。通过根索引节点可以的到根文件（目录文件）的内容，在其中搜索名为 home 的目录项，可以得到 home 对应的 ext2_dir_entry_2 目录项，根据其 inode 元素，可以得到 home 对应的文件内容。

在 home 的文件内容中，查找 zenhumany 的目录项，可以得到 zenhumany 对应的 ext2_dir_entry_2 实例，根据 indoe 元素，可以得到 zenhumany 的文件内容，在其中搜索 test.txt，可以得到 test.txt 的 ext2_dir_entry_2 实例，根据 inode,可以得到 test.txt 对应的索引号，也就可以找到 test.txt 的文件内容了。

前面留了一个疑问，“/”的索引节点是如何来的。要访问一个文件，先要访问一个目录，才能根据文件名从目录中找到给文件的目录项，进而找到其 i 节点。可是目录自身也是文件，它本身的目录项又在另一个目录项中，这样一来就成先有鸡还是先有蛋的问题了。

linux 解决办法是：“/”目录，或者说根设备上的根目录的目录项不在其他目录中，可以通过在一个固定的位置上或者通过一个固定的算法找到，从这个目录出发，可以找到系统中任何一个文件。linux 中每一个“文件系统”，即每一格式化成某种文件系统的存储设备上都有一个根目录，同时都有一个“超级块”，根目录的位置以及文件系统中的其他一些参数就记录在超级块中。超级块在设备上的逻辑位置都是固定的。

5.1.3 具体实现

fs/namei.c

```
/* Returns 0 and nd will be valid on success; Retuns error, otherwise. */
static int do_path_lookup(int dfd, const char *name,
                        unsigned int flags, struct nameidata *nd)
{
        int retval = path_init(dfd, name, flags, nd);
        if (!retval)
                retval = path_walk(name, nd);
        if (unlikely(!retval && !audit_dummy_context() && nd->path.dentry &&
                        nd->path.dentry->d_inode))
                audit_inode(name, nd->path.dentry);
        if (nd->root.mnt) {
                path_put(&nd->root);
                nd->root.mnt = NULL;
        }
        return retval;
}

int path_lookup(const char *name, unsigned int flags,
```

```
                 struct nameidata *nd)
{
        return do_path_lookup(AT_FDCWD, name, flags, nd);
}
```

由路径到索引节点实现的入口函数是 path_lookup，其调用 do_path_lookup 来实现具体功能。

do_path_lookup 调用连个函数来实现具体功能

- path_init：确定搜索路径的起始 dentry 结构。
- path_walk：根据 path_init 中提供的起始 dentry 结构，查找目录项的对应结构。

5.1.3.1 path_init

参数 nd 的类型为 nameidata，它是一个临时变量，用来在查找 inode 的过程中。

```
struct nameidata {
        struct path path;
        struct qstr last;
        struct path root;
        unsigned int      flags;
        int        last_type;
        unsigned       depth;
        char *saved_names[MAX_NESTED_LINKS + 1];

        /* Intent data */
        union {
                struct open_intent open;
        } intent;
};
```

path：当前路径的 path 实例

root：进程更目录的 path 实例。

```
static int path_init(int dfd, const char *name, unsigned int flags, struct nameidata *nd)
{
	int retval = 0;
	int fput_needed;
	struct file *file;

	nd->last_type = LAST_ROOT; /* if there are only slashes... */
	nd->flags = flags;
	nd->depth = 0;
	nd->root.mnt = NULL;

	if (*name=='/') {
		set_root(nd);
		nd->path = nd->root;
		path_get(&nd->root);
	} else if (dfd == AT_FDCWD) {
		struct fs_struct *fs = current->fs;
		read_lock(&fs->lock);
		nd->path = fs->pwd;
		path_get(&fs->pwd);
		read_unlock(&fs->lock);
	} else {
		struct dentry *dentry;

		file = fget_light(dfd, &fput_needed);
		retval = -EBADF;
		if (!file)
			goto out_fail;

		dentry = file->f_path.dentry;

```

```
                retval = -ENOTDIR;
                if (!S_ISDIR(dentry->d_inode->i_mode))
                        goto fput_fail;

                retval = file_permission(file, MAY_EXEC);
                if (retval)
                        goto fput_fail;

                nd->path = file->f_path;
                path_get(&file->f_path);

                fput_light(file, fput_needed);
        }
        return 0;

fput_fail:
        fput_light(file, fput_needed);
out_fail:
        return retval;
}
```

当路径名是以“/”开始的，则认为是从当前进程的根目录开始操作。将 nd->path 设置为 fs->root。否则，将 nd->path 设置为 fs->pwd。

5.1.3.2 path_walk

```
static int path_walk(const char *name, struct nameidata *nd)
{
        struct path save = nd->path;
        int result;

```

```
982         current->total_link_count = 0;
983
984         /* make sure the stuff we saved doesn't go away */
985         path_get(&save);
986
987         result = link_path_walk(name, nd);
988         if (result == -ESTALE) {
989                 /* nd->path had been dropped */
990                 current->total_link_count = 0;
991                 nd->path = save;
992                 path_get(&nd->path);
993                 nd->flags |= LOOKUP_REVAL;
994                 result = link_path_walk(name, nd);
995         }
996
997         path_put(&save);
998
999         return result;
1000 }
```

987 行：调用 link_path_walk 实现具体功能。

link_path_walk：

该函数的逻辑比较简单，就是在一个 for 循环中处理每一个目录项，而每一次循环中针对目录项的不同做不同的处理：

- 目录项为“..”，由 follow_dotdot 处理。
- 目录项为正常目录，由 do_lookup 处理。

```
/*
 * Name resolution.
 * This is the basic name resolution function, turning a pathname into
 * the final dentry. We expect 'base' to be positive and a directory.
 *
 * Returns 0 and nd will have valid dentry and mnt on success.
```

```
 * Returns error and drops reference to input namei data on failure.
 */
static int link_path_walk(const char *name, struct nameidata *nd)
{
	struct path next;
	struct inode *inode;
	int err;
	unsigned int lookup_flags = nd->flags;

	while (*name=='/')
		name++;
	if (!*name)
		goto return_reval;

	inode = nd->path.dentry->d_inode;
	if (nd->depth)
		lookup_flags = LOOKUP_FOLLOW | (nd->flags & LOOKUP_CONTINUE);
```

如果是 nd->depth > 0，则表示 link_path_walk 为嵌套调用。

```

	/* At this point we know we have a real path component. */
```

如果路径名是

```
	for(;;) {
		unsigned long hash;
		struct qstr this;
		unsigned int c;

		nd->flags |= LOOKUP_CONTINUE;
		err = exec_permission(inode);
		if (err)
			break;
```

```
	this.name = name;
```

this.name 中保存本次循环中 name 的起始地址。

```
	c = *(const unsigned char *)name;

	hash = init_name_hash();
	do {
		name++;
		hash = partial_name_hash(c, hash);
		c = *(const unsigned char *)name;
	} while (c && (c != '/'));
```

向前移动指针 name，知道 c 为空或者遇到“/”。

```
	this.len = name - (const char *) this.name;
	this.hash = end_name_hash(hash);
```

保存本次查找目录的目录长度。

```

	/* remove trailing slashes? */
	if (!c)
		goto last_component;
	while (*++name == '/');
	if (!*name)
		goto last_with_slashes;

	/*
	 * "." and ".." are special - ".." especially so because it has
	 * to be able to know about the current root directory and
	 * parent relationships.
	 */
	if (this.name[0] == '.') switch (this.len) {
		default:
```

```
				break;
			case 2:
				if (this.name[1] != '.')
					break;
				follow_dotdot(nd);
				inode = nd->path.dentry->d_inode;
				/* fallthrough */
			case 1:
				continue;
		}
```

查看本次目录需要查找的是否为“.”或者“..”,如果是“.”，表示当前目录，则本次循环接触，跳转到下一次循环。如果是“..”，表示父目录，调用函数 follow_dotdot，查找父目录索引节点。

如果以“.”开头，但有不是上面两种情况，则路径有问题，跳出循环。

```
		/* This does the actual lookups.. */
		err = do_lookup(nd, &this, &next);
```

如果是正常的目录，调用 do_lookup 实现查找功能。

```
		if (err)
			break;

		err = -ENOENT;
		inode = next.dentry->d_inode;
		if (!inode)
			goto out_dput;
```

查找到了目录项，对应的目录项没有索引节点，则目录项有问题，跳转到 out_dput。

```

		if (inode->i_op->follow_link) {
			err = do_follow_link(&next, nd);
			if (err)
				goto return_err;
```

```
891                 err = -ENOENT;
892                 inode = nd->path.dentry->d_inode;
893                 if (!inode)
894                         break;
```

如果目录项是一个链接，则调用 do_follow_link 函数。

```
895         } else
896                 path_to_nameidata(&next, nd);
897         err = -ENOTDIR;
898         if (!inode->i_op->lookup)
899                 break;
900         continue;
```

进入下一轮循环。

```
901         /* here ends the main loop */
902
903 last_with_slashes:
904         lookup_flags |= LOOKUP_FOLLOW | LOOKUP_DIRECTORY;
```

如果是以“/”结尾，则设置查找参数为 LOOKUP_FOLLOW | LOOKUP_DIRECTORY

```
905 last_component:
906         /* Clear LOOKUP_CONTINUE iff it was previously unset */
907         nd->flags &= lookup_flags | ~LOOKUP_CONTINUE;
908         if (lookup_flags & LOOKUP_PARENT)
909                 goto lookup_parent;
```

908-909 行：如果要查找的是父节点，则最后一个目录项不用查找了，直接跳转到 lookup_parent

```
910         if (this.name[0] == '.') switch (this.len) {
911                 default:
912                         break;
913                 case 2:
914                         if (this.name[1] != '.')
915                                 break;
```

```
916                     follow_dotdot(nd);
917                     inode = nd->path.dentry->d_inode;
918                     /* fallthrough */
919                 case 1:
920                     goto return_reval;
```

919-920 行：和中间目录项处理不同，中间目录项时跳转到下一次循环，这里跳出循环。

```
921             }
922             err = do_lookup(nd, &this, &next);
923             if (err)
924                 break;
925             inode = next.dentry->d_inode;
926             if (follow_on_final(inode, lookup_flags)) {
927                 err = do_follow_link(&next, nd);
928                 if (err)
929                     goto return_err;
930                 inode = nd->path.dentry->d_inode;
931             } else
932                 path_to_nameidata(&next, nd);
933             err = -ENOENT;
934             if (!inode)
935                 break;
936             if (lookup_flags & LOOKUP_DIRECTORY) {
937                 err = -ENOTDIR;
938                 if (!inode->i_op->lookup)
939                     break;
940             }
941             goto return_base;
```

906-941 行：处理最后一个目录项。

```
942 lookup_parent:
```

```
943             nd->last = this;
```

```
944             nd->last_type = LAST_NORM;
945             if (this.name[0] != '.')
946                     goto return_base;
947             if (this.len == 1)
948                     nd->last_type = LAST_DOT;
949             else if (this.len == 2 && this.name[1] == '.')
950                     nd->last_type = LAST_DOTDOT;
951             else
952                     goto return_base;
```

942 行：如果要查找的是父节点，则

```
953 return_reval:
954             /*
955              * We bypassed the ordinary revalidation routines.
956              * We may need to check the cached dentry for staleness.
957              */
958             if (nd->path.dentry && nd->path.dentry->d_sb &&
959                     (nd->path.dentry->d_sb->s_type->fs_flags &
FS_REVAL_DOT)) {
960                     err = -ESTALE;
961                     /* Note: we do not d_invalidate() */
962                     if (!nd->path.dentry->d_op->d_revalidate(
963                                     nd->path.dentry, nd))
964                             break;
965             }
966 return_base:
967             return 0;
```

正确返回

```
968 out_dput:
969             path_put_conditional(&next, nd);
970             break;
971         }
```

```
972         path_put(&nd->path);
973 return_err:
974         return err;
```

错误返回。

```
975 }
```

5.1.3.2.1 回退到父目录follow_dotdot

```
670 static __always_inline void follow_dotdot(struct nameidata *nd)
671 {
672         set_root(nd);
```

设置 nd->root 为当前进程的 root。

```
673
674         while(1) {
675                 struct dentry *old = nd->path.dentry;
676
677                 if (nd->path.dentry == nd->root.dentry &&
678                     nd->path.mnt == nd->root.mnt) {
679                         break;
680                 }
681                 if (nd->path.dentry != nd->path.mnt->mnt_root) {
682                         /* rare case of legitimate dget_parent()... */
683                         nd->path.dentry = dget_parent(nd->path.dentry);
684                         dput(old);
685                         break;
686                 }
687                 if (!follow_up(&nd->path))
688                         break;
```

当 nd->path.dentry 所在的设备为根设备时，follow_up 返回 0，则执行到 688 行，跳出循环。

当 nd->path.dentry 所在的设备不是跟设备时，nd->path.dentry 为父设备中的安装点，nd->path.mnt 为父设备的 vfsmount。此时 follow_up 返回 1,688 行不执行，跳转到 674 行继续循环。

为啥 follow_up 会做如此的设计呢。因为 nd->path.dentry 已经为其所在设备的根设备了，并且 nd->path.mnt 所代表的设备也就是根设备了，所以没法再往上会退了，故返回 0，跳出循环。follow_up 返回 1，也就是 nd->path.dentry 所在的设备不是根设备，由 vfsmount 中 mnt_mountpoint（父设备安装点）与 mnt_mount(子设备根节点)表示同一层级，而 follow_dotdot 需要回退一级，所以 follow_up 返回 1，进而 where 循环继续运行，回退到 mnt_mountpoint 中的父节点。

```
689        }
690        follow_mount(&nd->path);
```

follow_mount 处理 nd->path.dentry 为安装点的情况。

```
691 }
```

该函数三种情况来处理：

- 第一种情况，已经到达节点 nd->path.dentry 就是本进程的根节点，且到达节点的安装点 nd->path.mnt 与本进程根节点的安装点相同，此时就不能在往上跑了，保持 nd->path.dentry 不变。（677-680）
- 第二种情况，已经到达的节点 nd->path.dentry 与其父节点在同一设备上。且 nd->path.dentry 不是根节点，这种情况处理比较简单，去 path.dentry 的父目录项即可。681 行为何能保证这一点能(因为 nd->path 所在的安装点就是 nd->path.mnt，所以 nd->path 与 nd->paht.mnt->mnt_root 一定在同一设备上）。（681-686）
- 第三中情况，已经到达的节点就是该节点所在设备的根节点，在往上跑一层就要跑到另一个设备上去了。这种情况交由 follow_up 处理。(687 行）

当将一个存储设备“安装”到另一个设备上的某个节点时，内核会分配和设置一个 vfsmount 设备，通过这个结构上的 mnt_mountpoint 和 mnt_root 将这两个设备节点关联起来。vfsmount 结构中 mnt_parent 指向“父设备”。mnt_mountpoint 指向安装点，mnt_root 指向安装设备（子设备）的根目录。

```
50 struct vfsmount {
```

```
struct list_head mnt_hash;
struct vfsmount *mnt_parent;       /* fs we are mounted on */
struct dentry *mnt_mountpoint;    /* dentry of mountpoint */
struct dentry *mnt_root;       /* root of the mounted tree */
struct super_block *mnt_sb; /* pointer to superblock */
```

follow_up 函数

```
int follow_up(struct path *path)
```

```
{
        struct vfsmount *parent;
        struct dentry *mountpoint;
        spin_lock(&vfsmount_lock);
        parent = path->mnt->mnt_parent;
```

parent 父设备安装点

```
        if (parent == path->mnt) {
                spin_unlock(&vfsmount_lock);
                return 0;
        }
```

如果设备 mnt 与 mnt->mnt_parent 相同，则该设备为根设备，所以返回 0。

```
        mntget(parent);
        mountpoint = dget(path->mnt->mnt_mountpoint);
        spin_unlock(&vfsmount_lock);
        dput(path->dentry);
        path->dentry = mountpoint;
```

设置 path->dentry 为父设备中的安装点

```
        mntput(path->mnt);
        path->mnt = parent;
```

设置 path->mnt 为父设备的 vfsmount。

```
        return 1;
```

返回 1。

```
}
```

follow_mount 函数

```
static void follow_mount(struct path *path)
{
        while (d_mountpoint(path->dentry)) {
                struct vfsmount *mounted = lookup_mnt(path);
                if (!mounted)
                        break;
                dput(path->dentry);
                mntput(path->mnt);
                path->mnt = mounted;
                path->dentry = dget(mounted->mnt_root);
        }
}
```

d_mountpoint 函数检测 path->dentry 是否为安装点，如果是，调用 lookup_mnt 搜索目录项高速缓存中已安装文件系统的根目录，并把 nd->dentry 和 nd->mnt 更新为相应已安装文件系统的安装点和安装系统对象地址。然后重复整个操作（几个文件系统可以安装在同一个安装点上）。本质上讲，由于进程可能从某个文件系统的目录开始路径名的查找，而该目录被另一个安装在其父目录上的文件系统所隐藏，那么当需要回到父目录时，则调用 follow_mount 函数。

5.1.3.2.2 目录查找函数 do_lookup

该函数是实现由中间目录名到 inode 节点转换的具体函数。

```
/*
 *  It's more convoluted than I'd like it to be, but... it's still fairly
 *  small and for now I'd prefer to have fast path as straight as possible.
 *  It _is_ time-critical.
```

```
 */
static int do_lookup(struct nameidata *nd, struct qstr *name,
                     struct path *path)
{
        struct vfsmount *mnt = nd->path.mnt;
        struct dentry *dentry, *parent;
        struct inode *dir;
        /*
         * See if the low-level filesystem might want
         * to use its own hash..
         */
        if (nd->path.dentry->d_op && nd->path.dentry->d_op->d_hash) {
                int err = nd->path.dentry->d_op->d_hash(nd->path.dentry, name);
                if (err < 0)
                        return err;
        }

        dentry = __d_lookup(nd->path.dentry, name);
```

在 nd->path.dentry 对应的缓存中查找名称为 name 的子 dentry 项。

```
        if (!dentry)
                goto need_lookup;
```

如果在缓存中没有找到，则调整到 need_lookup 函数，此时需要从磁盘中读入相关信息，建立 dentry 缓存。

```
        if (dentry->d_op && dentry->d_op->d_revalidate)
                goto need_revalidate;
```

如果 dentry 的 d_op-> d_revalidate 非空，则调整到验证代码。

```
done:
        path->mnt = mnt;
        path->dentry = dentry;
```

查找到 dentry 后，设置 path 相关变量。

```
	__follow_mount(path);
```

如果 path->dentry 为安装点，则调用__follow_mount 函数前进到子设备的根目录。

```
	return 0;

need_lookup:
	parent = nd->path.dentry;
	dir = parent->d_inode;

	mutex_lock(&dir->i_mutex);
	/*
	 * First re-do the cached lookup just in case it was created
	 * while we waited for the directory semaphore..
	 *
	 * FIXME! This could use version numbering or similar to
	 * avoid unnecessary cache lookups.
	 *
	 * The "dcache_lock" is purely to protect the RCU list walker
	 * from concurrent renames at this point (we mustn't get false
	 * negatives from the RCU list walk here, unlike the optimistic
	 * fast walk).
	 *
	 * so doing d_lookup() (with seqlock), instead of lockfree __d_lookup
	 */
	dentry = d_lookup(parent, name);
```

再次在缓存中查找。

```
	if (!dentry) {
		struct dentry *new;

		/* Don't create child dentry for a dead directory. */
		dentry = ERR_PTR(-ENOENT);
```

```
        if (IS_DEADDIR(dir))
                goto out_unlock;

        new = d_alloc(parent, name);
```

在内存中分配一个新的 dentry 结构。

```
        dentry = ERR_PTR(-ENOMEM);
        if (new) {
                dentry = dir->i_op->lookup(dir, new, nd);
```

根据具体的文件系统，选择具体的查找函数，对于 ext2 而言，就是 ext2_lookup 函数。

```
                if (dentry)
                        dput(new);
                else
                        dentry = new;
        }
out_unlock:
        mutex_unlock(&dir->i_mutex);
        if (IS_ERR(dentry))
                goto fail;
        goto done;
    }

    /*
     * Uhhuh! Nasty case: the cache was re-populated while
     * we waited on the semaphore. Need to revalidate.
     */
    mutex_unlock(&dir->i_mutex);
    if (dentry->d_op && dentry->d_op->d_revalidate) {
        dentry = do_revalidate(dentry, nd);
        if (!dentry)
                dentry = ERR_PTR(-ENOENT);
```

```
	}
```

如果是在缓存中找到的 dentry，需要验证。

```
	if (IS_ERR(dentry))
		goto fail;
	goto done;

need_revalidate:
	dentry = do_revalidate(dentry, nd);
```

验证

```
	if (!dentry)
		goto need_lookup;
```

验证失败，跳转到真正查找的地方

```
	if (IS_ERR(dentry))
		goto fail;
	goto done;

fail:
	return PTR_ERR(dentry);
}
```

do_lookup 函数现在缓存中查找（d_lookup）name 对应的 dentry 是否存在，如果存在，在返回该 dentry。 否则，就需要先分配一个 dentry 结构，然后调用具体的文件系统查找函数，从磁盘中读取相关内容建立内存中的 dentry 数据结构。

缓存中搜索 dentry 结构

```
struct dentry * d_lookup(struct dentry * parent, struct qstr * name)
{
	struct dentry * dentry = NULL;
	unsigned long seq;

```

```
        do {
                seq = read_seqbegin(&rename_lock);
                dentry = __d_lookup(parent, name);
                if (dentry)
            break;
    } while (read_seqretry(&rename_lock, seq));
    return dentry;
}
EXPORT_SYMBOL(d_lookup);

struct dentry * __d_lookup(struct dentry * parent, struct qstr * name)
{
```

参数 parent 表示父目录，name 表示需要在父目录中查找的名称。

```
    unsigned int len = name->len;
    unsigned int hash = name->hash;
    const unsigned char *str = name->name;
    struct hlist_head *head = d_hash(parent,hash);
    struct dentry *found = NULL;
    struct hlist_node *node;
    struct dentry *dentry;

    rcu_read_lock();

    hlist_for_each_entry_rcu(dentry, node, head, d_hash) {
        struct qstr *qstr;

        if (dentry->d_name.hash != hash)
            continue;
        if (dentry->d_parent != parent)
            continue;
```

判断当前的 dentry 是否满足要求，满足要求的条件是：

- hash 值与传递进来的 hash 值相同
- dentry 是 parent 的子目录

```

	spin_lock(&dentry->d_lock);

	/*
	 * Recheck the dentry after taking the lock - d_move may have
	 * changed things.  Don't bother checking the hash because we're
	 * about to compare the whole name anyway.
	 */
	if (dentry->d_parent != parent)
		goto next;

	/* non-existing due to RCU? */
	if (d_unhashed(dentry))
		goto next;

	/*
	 * It is safe to compare names since d_move() cannot
	 * change the qstr (protected by d_lock).
	 */
	qstr = &dentry->d_name;
	if (parent->d_op && parent->d_op->d_compare) {
		if (parent->d_op->d_compare(parent, qstr, name))
			goto next;
	} else {
		if (qstr->len != len)
			goto next;
		if (memcmp(qstr->name, str, len))
			goto next;
```

```
        }
```

名称，长度的进一步检验

```

                atomic_inc(&dentry->d_count);
                found = dentry;
                spin_unlock(&dentry->d_lock);
                break;
next:
                spin_unlock(&dentry->d_lock);
        }
        rcu_read_unlock();

return found;
}
```

分配一个新的 dentry

```
/**
 * d_alloc  -  allocate a dcache entry
 * @parent: parent of entry to allocate
 * @name: qstr of the name
 *
 * Allocates a dentry. It returns %NULL if there is insufficient memory
 * available. On a success the dentry is returned. The name passed in is
 * copied and the copy passed in may be reused after this call.
 */

struct dentry *d_alloc(struct dentry * parent, const struct qstr *name)
{
```

参数 parent 为新分配目录项的父目录。

name 为新分配目录项的名称。

```
        struct dentry *dentry;
```

```
char *dname;

dentry = kmem_cache_alloc(dentry_cache, GFP_KERNEL);
```

从 dentry_cache 缓存中分配 dentry

```
if (!dentry)
    return NULL;

if (name->len > DNAME_INLINE_LEN-1) {
    dname = kmalloc(name->len + 1, GFP_KERNEL);
    if (!dname) {
        kmem_cache_free(dentry_cache, dentry);
        return NULL;
    }
} else   {
    dname = dentry->d_iname;
}
dentry->d_name.name = dname;

dentry->d_name.len = name->len;
dentry->d_name.hash = name->hash;
memcpy(dname, name->name, name->len);
dname[name->len] = 0;

atomic_set(&dentry->d_count, 1);
dentry->d_flags = DCACHE_UNHASHED;
spin_lock_init(&dentry->d_lock);
dentry->d_inode = NULL;
dentry->d_parent = NULL;
dentry->d_sb = NULL;
dentry->d_op = NULL;
dentry->d_fsdata = NULL;
```

```
        dentry->d_mounted = 0;
        INIT_HLIST_NODE(&dentry->d_hash);
        INIT_LIST_HEAD(&dentry->d_lru);
        INIT_LIST_HEAD(&dentry->d_subdirs);
        INIT_LIST_HEAD(&dentry->d_alias);
```

初始化 dentry 结构相关变量

```

        if (parent) {
                dentry->d_parent = dget(parent);
```

设置 dentry 父节点

```
        } else {
                INIT_LIST_HEAD(&dentry->d_u.d_child);
        }

        spin_lock(&dcache_lock);
        if (parent)
                list_add(&dentry->d_u.d_child, &parent->d_subdirs);
        dentry_stat.nr_dentry++;
        spin_unlock(&dcache_lock);

        return dentry;
}
EXPORT_SYMBOL(d_alloc);
```

5.1.3.3 ext2 目录查找函数ext2_lookup

在 do_lookup 的 756 行中有如下的调用，

```
756                 dentry = dir->i_op->lookup(dir, new, nd);
```

最终会转换为 ext2_lookup 函数。

fs/ext2/namei.c

```
/*
```

```
 * Methods themselves.
 */

static struct dentry *ext2_lookup(struct inode * dir, struct dentry *dentry, struct nameidata *nd)
{
	struct inode * inode;
	ino_t ino;

	if (dentry->d_name.len > EXT2_NAME_LEN)
		return ERR_PTR(-ENAMETOOLONG);

	ino = ext2_inode_by_name(dir, &dentry->d_name);
```

有名称查找对应的 inode 索引。

```
	inode = NULL;
	if (ino) {
		inode = ext2_iget(dir->i_sb, ino);
```

通过 indoe 索引，从磁盘读入相关信息，在内存中建立 inode 结构。

```
		if (unlikely(IS_ERR(inode))) {
			if (PTR_ERR(inode) == -ESTALE) {
				ext2_error(dir->i_sb, __func__,
						"deleted inode referenced: %lu",
						(unsigned long) ino);
				return ERR_PTR(-EIO);
			} else {
				return ERR_CAST(inode);
			}
		}
	}
	return d_splice_alias(inode, dentry);
```

安装 inode 到 dentry 结构中。

```
}
```

5.1.3.3.1 name到索引节点号ext2_inode_by_name

```
ino_t ext2_inode_by_name(struct inode *dir, struct qstr *child)
{
        ino_t res = 0;
        struct ext2_dir_entry_2 *de;
        struct page *page;

        de = ext2_find_entry (dir, child, &page);
        if (de) {
                res = le32_to_cpu(de->inode);
```

得到 inode 节点的索引号

```
                ext2_put_page(page);
        }
        return res;
}
```

查找磁盘上目录项的相关内容

```
/*
 *   ext2_find_entry()
 *
 * finds an entry in the specified directory with the wanted name. It
 * returns the page in which the entry was found (as a parameter - res_page),
 * and the entry itself. Page is returned mapped and unlocked.
 * Entry is guaranteed to be valid.
 */
struct ext2_dir_entry_2 *ext2_find_entry (struct inode * dir,
                        struct qstr *child, struct page ** res_page)
```

```
{
	const char *name = child->name;
	int namelen = child->len;
	unsigned reclen = EXT2_DIR_REC_LEN(namelen);
	unsigned long start, n;
	unsigned long npages = dir_pages(dir);
	struct page *page = NULL;
	struct ext2_inode_info *ei = EXT2_I(dir);
	ext2_dirent * de;
	int dir_has_error = 0;

	if (npages == 0)
		goto out;

	/* OFFSET_CACHE */
	*res_page = NULL;

	start = ei->i_dir_start_lookup;
	if (start >= npages)
		start = 0;
	n = start;
	do {
		char *kaddr;
		page = ext2_get_page(dir, n, dir_has_error);
```

从父目录文件中读取一页内容，其中可能涉及到磁盘读取操作。

```
		if (!IS_ERR(page)) {
			kaddr = page_address(page);
			de = (ext2_dirent *) kaddr;
```

Kaddr 可以看做是一个 ext2_dir_entry_2 数组

```
			kaddr += ext2_last_byte(dir, n) - reclen;
			while ((char *) de <= kaddr) {
```

```
				if (de->rec_len == 0) {
					ext2_error(dir->i_sb, __func__,
						"zero-length directory entry");
					ext2_put_page(page);
					goto out;
				}
				if (ext2_match (namelen, name, de))
```

检测是否为需要的元素

```
					goto found;
				de = ext2_next_entry(de);
			}
			ext2_put_page(page);
		} else
			dir_has_error = 1;

		if (++n >= npages)
			n = 0;
		/* next page is past the blocks we've got */
		if (unlikely(n > (dir->i_blocks >> (PAGE_CACHE_SHIFT - 9)))) {
			ext2_error(dir->i_sb, __func__,
				"dir %lu size %lld exceeds block count %llu",
				dir->i_ino, dir->i_size,
				(unsigned long long)dir->i_blocks);
			goto out;
		}
	} while (n != start);
out:
	return NULL;

found:
	*res_page = page;
```

```
	ei->i_dir_start_lookup = n;
	return de;
}
```

5.1.3.3.2 索引号到索引节点 ext2_iget

```
struct inode *ext2_iget (struct super_block *sb, unsigned long ino)
{
	struct ext2_inode_info *ei;
	struct buffer_head * bh;
	struct ext2_inode *raw_inode;
	struct inode *inode;
	long ret = -EIO;
	int n;

	inode = iget_locked(sb, ino);
	if (!inode)
		return ERR_PTR(-ENOMEM);
	if (!(inode->i_state & I_NEW))
		return inode;

	ei = EXT2_I(inode);
```

从 inode 中提取出 ext2_inode 信息。

```
	ei->i_block_alloc_info = NULL;

	raw_inode = ext2_get_inode(inode->i_sb, ino, &bh);
```

该函数是重点，获得原始的 inode 信息。

```
	if (IS_ERR(raw_inode)) {
		ret = PTR_ERR(raw_inode);
		goto bad_inode;
	}
```

```

	inode->i_mode = le16_to_cpu(raw_inode->i_mode);
	inode->i_uid = (uid_t)le16_to_cpu(raw_inode->i_uid_low);
	inode->i_gid = (gid_t)le16_to_cpu(raw_inode->i_gid_low);
	if (!(test_opt (inode->i_sb, NO_UID32))) {
		inode->i_uid |= le16_to_cpu(raw_inode->i_uid_high) << 16;
		inode->i_gid |= le16_to_cpu(raw_inode->i_gid_high) << 16;
	}
	inode->i_nlink = le16_to_cpu(raw_inode->i_links_count);
	inode->i_size = le32_to_cpu(raw_inode->i_size);
	inode->i_atime.tv_sec = (signed)le32_to_cpu(raw_inode->i_atime);
	inode->i_ctime.tv_sec = (signed)le32_to_cpu(raw_inode->i_ctime);
	inode->i_mtime.tv_sec = (signed)le32_to_cpu(raw_inode->i_mtime);
	inode->i_atime.tv_nsec = inode->i_mtime.tv_nsec = inode->i_ctime.tv_nsec = 0;
	ei->i_dtime = le32_to_cpu(raw_inode->i_dtime);
	/* We now have enough fields to check if the inode was active or not.
	 * This is needed because nfsd might try to access dead inodes
	 * the test is that same one that e2fsck uses
	 * NeilBrown 1999oct15
	 */
	if (inode->i_nlink == 0 && (inode->i_mode == 0 || ei->i_dtime)) {
		/* this inode is deleted */
		brelse (bh);
		ret = -ESTALE;
		goto bad_inode;
	}
	inode->i_blocks = le32_to_cpu(raw_inode->i_blocks);
	ei->i_flags = le32_to_cpu(raw_inode->i_flags);
	ei->i_faddr = le32_to_cpu(raw_inode->i_faddr);
	ei->i_frag_no = raw_inode->i_frag;
	ei->i_frag_size = raw_inode->i_fsize;
```

```
	ei->i_file_acl = le32_to_cpu(raw_inode->i_file_acl);
	ei->i_dir_acl = 0;
	if (S_ISREG(inode->i_mode))
		inode->i_size |= ((__u64)le32_to_cpu(raw_inode->i_size_high)) << 32;
	else
		ei->i_dir_acl = le32_to_cpu(raw_inode->i_dir_acl);
	ei->i_dtime = 0;
	inode->i_generation = le32_to_cpu(raw_inode->i_generation);
	ei->i_state = 0;
	ei->i_block_group = (ino - 1) / EXT2_INODES_PER_GROUP(inode->i_sb);
	ei->i_dir_start_lookup = 0;

	/*
	 * NOTE! The in-memory inode i_data array is in little-endian order
	 * even on big-endian machines: we do NOT byteswap the block numbers!
	 */
	for (n = 0; n < EXT2_N_BLOCKS; n++)
		ei->i_data[n] = raw_inode->i_block[n];
```

将磁盘中 i_block 数组拷贝到 inode 中。

```

	if (S_ISREG(inode->i_mode)) {
		inode->i_op = &ext2_file_inode_operations;
		if (ext2_use_xip(inode->i_sb)) {
			inode->i_mapping->a_ops = &ext2_aops_xip;
			inode->i_fop = &ext2_xip_file_operations;
		} else if (test_opt(inode->i_sb, NOBH)) {
			inode->i_mapping->a_ops = &ext2_nobh_aops;
			inode->i_fop = &ext2_file_operations;
		} else {
```

```
			inode->i_mapping->a_ops = &ext2_aops;
			inode->i_fop = &ext2_file_operations;
		}
	} else if (S_ISDIR(inode->i_mode)) {
		inode->i_op = &ext2_dir_inode_operations;
		inode->i_fop = &ext2_dir_operations;
		if (test_opt(inode->i_sb, NOBH))
			inode->i_mapping->a_ops = &ext2_nobh_aops;
		else
			inode->i_mapping->a_ops = &ext2_aops;
	} else if (S_ISLNK(inode->i_mode)) {
		if (ext2_inode_is_fast_symlink(inode)) {
			inode->i_op = &ext2_fast_symlink_inode_operations;
			nd_terminate_link(ei->i_data, inode->i_size,
				sizeof(ei->i_data) - 1);
		} else {
			inode->i_op = &ext2_symlink_inode_operations;
			if (test_opt(inode->i_sb, NOBH))
				inode->i_mapping->a_ops = &ext2_nobh_aops;
			else
				inode->i_mapping->a_ops = &ext2_aops;
		}
	} else {
		inode->i_op = &ext2_special_inode_operations;
		if (raw_inode->i_block[0])
			init_special_inode(inode, inode->i_mode,
			   old_decode_dev(le32_to_cpu(raw_inode->i_block[0])));
		else
			init_special_inode(inode, inode->i_mode,
			   new_decode_dev(le32_to_cpu(raw_inode->i_block[1])));
	}
```

```
	brelse (bh);
	ext2_set_inode_flags(inode);
	unlock_new_inode(inode);
	return inode;

bad_inode:
	iget_failed(inode);
	return ERR_PTR(ret);
}
```

本函数的重点是 ext2_get_inode 函数，它根据 inode 索引节点号的到磁盘上的原始 indoe 信息，然后用此初始化内存中的 inode 结构。

```
static struct ext2_inode *ext2_get_inode(struct super_block *sb, ino_t ino,
					struct buffer_head **p)
{
	struct buffer_head * bh;
	unsigned long block_group;
	unsigned long block;
	unsigned long offset;
	struct ext2_group_desc * gdp;

	*p = NULL;
	if ((ino != EXT2_ROOT_INO && ino < EXT2_FIRST_INO(sb)) ||
	    ino > le32_to_cpu(EXT2_SB(sb)->s_es->s_inodes_count))
	    goto Einval;

	block_group = (ino - 1) / EXT2_INODES_PER_GROUP(sb);
```

计算块组号

```
	gdp = ext2_get_group_desc(sb, block_group, NULL);
```

得到块组

```
	if (!gdp)
```

```
                goto Egdp;
        /*
         * Figure out the offset within the block group inode table
         */
        offset = ((ino - 1) % EXT2_INODES_PER_GROUP(sb)) * EXT2_INODE_SIZE(sb);
```

节点在块内偏移

```
        block = le32_to_cpu(gdp->bg_inode_table) +
                (offset >> EXT2_BLOCK_SIZE_BITS(sb));
```

节点在磁盘上的块号

```
        if (!(bh = sb_bread(sb, block)))
                goto Eio;
```

调用底层驱动函数，读取指定块。

```

        *p = bh;
        offset &= (EXT2_BLOCK_SIZE(sb) - 1);
        return (struct ext2_inode *) (bh->b_data + offset);
```

返回

```

Einval:
        ext2_error(sb, "ext2_get_inode", "bad inode number: %lu",
                   (unsigned long) ino);
        return ERR_PTR(-EINVAL);
Eio:
        ext2_error(sb, "ext2_get_inode",
                   "unable to read inode block - inode=%lu, block=%lu",
                   (unsigned long) ino, block);
Egdp:
        return ERR_PTR(-EIO);
}
```

5.1.3.4 符号链接处理do_follow_link

在由名称到节点的转换过程中，还需检测最后得到的目录项是否为链接，如果是链接的话，调用 do_follow_walk 函数处理。

考虑如下的情况，如果/home/henry 是指向/home/zenhumany 的一个符号链接，那么，/home/henry/test.txt 必须由内核解析为/home/zenhumany/test.txt。在此，内核指向两个操作。

第一个查找操作：解析到/home/henry 时，发现 henry 是一个链接，内核必须将其链接内容/home/zenhumany 提取出来。进入第二个查找操作。

第二个查找操作，解析/home/zenhumany 目录，当得到 zenhumany 的 dentry 的目录项后，转入到第一个查找操作。

第一个查找操作从 zenhumany 的 dentry 中继续查找 test.txt 的目录项。

然而，难以驾驭的递归本质上是危险的。例如，假定一个符号链接指向自己。当然，解析含有这样符号链接的路径名可能导致无休止的递归调用流，这又依次引发内核栈的溢出。当前进程的描述符中的 link_count 字段用来避免这种问题：每次递归执行前增加这个字段的值，执行之后减少其值。如果该字段的值达到 6，整个循环操作就以错误码结束。因此，符号链接嵌套的层数不超过 5。

另外，当前进程的描述符中的 total_link_count 字段记录在原查找操作中有多少符号链接（甚至非嵌套的）被跟踪。如果这个计数器的值到 40，则查找操作中止。没有这个计数器，怀有恶意的用户就可能创建一个病态的路径名，让其中包含很多连续的符号链接，使内核在无休止的查找操作中冻结。

```
static inline int do_follow_link(struct path *path, struct nameidata *nd)
{
        void *cookie;
        int err = -ELOOP;
        if (current->link_count >= MAX_NESTED_LINKS)
                goto loop;
        if (current->total_link_count >= 40)
                goto loop;
        BUG_ON(nd->depth >= MAX_NESTED_LINKS);
```

```
        cond_resched();
        err = security_inode_follow_link(path->dentry, nd);
        if (err)
                goto loop;
        current->link_count++;
```

增加符号链接计数，防止嵌套的层数过的导致内核溢出。

```
        current->total_link_count++;
```

原查找路径中总的符号链接数。

```
        nd->depth++;
        err = __do_follow_link(path, nd, &cookie);
```

调用__do_follow_link 处理。

```
        if (!IS_ERR(cookie) && path->dentry->d_inode->i_op->put_link)
                path->dentry->d_inode->i_op->put_link(path->dentry, nd, cookie);
        path_put(path);
        current->link_count--;
        nd->depth--;
```

减少链接引用计数。

```
        return err;
loop:
        path_put_conditional(path, nd);
        path_put(&nd->path);
        return err;
}
```

```
static __always_inline int
__do_follow_link(struct path *path, struct nameidata *nd, void **p)
{
        int error;
        struct dentry *dentry = path->dentry;

```

```
	touch_atime(path->mnt, dentry);
	nd_set_link(nd, NULL);

	if (path->mnt != nd->path.mnt) {
		path_to_nameidata(path, nd);
		dget(dentry);
	}
	mntget(path->mnt);
	nd->last_type = LAST_BIND;
	*p = dentry->d_inode->i_op->follow_link(dentry, nd);
	error = PTR_ERR(*p);
	if (!IS_ERR(*p)) {
		char *s = nd_get_link(nd);
```

得到符号链接所表示的目录。

```
		error = 0;
		if (s)
			error = __vfs_follow_link(nd, s);
		else if (nd->last_type == LAST_BIND) {
			error = force_reval_path(&nd->path, nd);
			if (error)
				path_put(&nd->path);
		}
	}
	return error;
}
```

```
static __always_inline int __vfs_follow_link(struct nameidata *nd, const char *link)
{
	if (IS_ERR(link))
		goto fail;
```

```

	if (*link == '/') {
		set_root(nd);
		path_put(&nd->path);
		nd->path = nd->root;
		path_get(&nd->root);
	}

	return link_path_walk(link, nd);
```

从符号链接的目录开始调用 link_path_walk。大家记得 do_follow_link 是从 link_path_walk 中调用的，现在回调 link_path_walk 函数，构成递归调用。

```
fail:
	path_put(&nd->path);
	return PTR_ERR(link);
}
```

5.2 文件系统的安装与拆卸

5.2.1 概述

在一个块设备上按一定的格式建立起文件系统的时候，或者系统引导之初，设备上的文件和节点都还是不可访问的。也就是说，还不能按一定的路径名访问其中的特定的节点或文件（虽然作为“设备”是可访问的）。只有把它“安装”到计算机系统的文件系统中的某个节点上，才能使设备上的文件和节点成为可访问的。经过安装以后，设备上的“文件系统”就成为整个文件系统的一部分，或者说一个子系统。一般而言，文件系统就像一颗倒立的树，不过由于可能存在着的节点间的“连接”和“符号连接”而不一定是严格意义上的“树”。最初时，整个系统中只有一个节点，那就是整个文件系统的“根”节点，这个节点存在于内存中，而不在任何具体的设备上。系统在初始化时将一个“根设备”安装到节点“/”上，这个设备上的文件系统就成了整个系统中原始的、基本的文件系统（所以才称为根设备）。此后，就可以由超级用户进程通过系统调用 mount( )把其他的子系统安装到已经存在于文件系统中的空

闲节点上，使整个文件系统得以扩展，当不再需要使用某个子系统时，或者在关闭系统之前，则通过系统调用 umount 把已经安装的设备逐个“拆卸”下来。

系统调用 mount 将一个可访问的块设备（代表设备本身）安装到一个可访问的节点（设备安装点）上。所谓“可访问”是指该节点或者文件已经存在于已安装的文件系统中，可以通过路径名寻访。Uinx（以及 linux）将设备看作一种特殊的文件，并在文件系统中有代表着具体设备的节点，称为“设备”文件，通常都在目录“/dev”中。例如 IDE 硬盘上的第一个分区就是/dev/hda1。每个设备文件实际上只是一个索引节点，节点中提供了设备的“设备号”，由“主设备号”和“次设备号”两部分构成。其中主设备号指明了设备的种类，或者更确切地说是指明了应该使用哪一组驱动程序。同一个物理的设备，如果有两组不同的驱动程序，在逻辑上就被视作两种不同的设备而在文件系统中有两个不同的“设备文件”。次设备号则指明了该设备时同种设备中的第几个。所以，只要找到代表着某个设备的索引节点，就知道该怎样读/写这个设备了。既然是一个“可访问”的块设备，那为什么还要安装呢？答案是在安装之前可访问的只是这个设备，通常是作为一个线性的无结构的字节流来访问的，称为“原始设备”（raw device）。而在设备上的文件系统则是不可访问的。经过安装之后，设备上的文件系统就称为可访问的了。

这里有一个文件就是：系统调用 mount 要求被安装的块设备在安装之前就是可访问的，那根设备怎么办？在安装根设备之前，系统中只有一个“/”节点，根本就不存在可访问的块设备啊。其实，明白了“可访问的块设备”的含义就好解决这个问题了。“可访问的块设备”只是提供了设备的主设备号和次设备号，如果可以通过其他途径得到设备的主设备号和次设备号（比喻通过内核启动参数），那么同样可以安装设备上的文件系统。根设备的安装就是通过这种方式来的。事实上，根据情况的不同，内核中有三个函数是用于设备安装的，那就是 sys_mount，mount_root 以及 kem_mount。

5.2.2 sys_mount 文件系统的安装

```
2130 SYSCALL_DEFINE5(mount, char __user *, dev_name, char __user *,
dir_name,
2131            char __user *, type, unsigned long, flags, void __user *, data)
```

参数：

- dev_name：设备名称（如/dev/hda1）
- dir_name：安装点路径

■ type：安装类型

```
{
	int ret;
	char *kernel_type;
	char *kernel_dir;
	char *kernel_dev;
	unsigned long data_page;

	ret = copy_mount_string(type, &kernel_type);
	if (ret < 0)
		goto out_type;

	kernel_dir = getname(dir_name);
	if (IS_ERR(kernel_dir)) {
		ret = PTR_ERR(kernel_dir);
		goto out_dir;
	}

	ret = copy_mount_string(dev_name, &kernel_dev);
	if (ret < 0)
		goto out_dev;

	ret = copy_mount_options(data, &data_page);
	if (ret < 0)
		goto out_data;
```

将对应的参数从用户空间拷贝到内核空间。

```

	ret = do_mount(kernel_dev, kernel_dir, kernel_type, flags,
		(void *) data_page);
```

do_mount 实现具体功能。

```

```

```
        free_page(data_page);
out_data:
        kfree(kernel_dev);
out_dev:
        putname(kernel_dir);
out_dir:
        kfree(kernel_type);
out_type:
        return ret;
}
```

5.2.2.1 do_mount

```
long do_mount(char *dev_name, char *dir_name, char *type_page,
                  unsigned long flags, void *data_page)
{
        struct path path;
        int retval = 0;
        int mnt_flags = 0;

        /* Discard magic */
        if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
                flags &= ~MS_MGC_MSK;

        /* Basic sanity checks */

        if (!dir_name || !*dir_name || !memchr(dir_name, 0, PAGE_SIZE))
                return -EINVAL;

        if (data_page)
                ((char *)data_page)[PAGE_SIZE - 1] = 0;
```

```

	/* ... and get the mountpoint */
	retval = kern_path(dir_name, LOOKUP_FOLLOW, &path);
```

得到安装点

```
	if (retval)
		return retval;

	retval = security_sb_mount(dev_name, &path,
				   type_page, flags, data_page);
```

安全性检查

```
	if (retval)
		goto dput_out;

	/* Default to relatime unless overriden */
	if (!(flags & MS_NOATIME))
		mnt_flags |= MNT_RELATIME;

	/* Separate the per-mountpoint flags */
	if (flags & MS_NOSUID)
		mnt_flags |= MNT_NOSUID;
	if (flags & MS_NODEV)
		mnt_flags |= MNT_NODEV;
	if (flags & MS_NOEXEC)
		mnt_flags |= MNT_NOEXEC;
	if (flags & MS_NOATIME)
		mnt_flags |= MNT_NOATIME;
	if (flags & MS_NODIRATIME)
		mnt_flags |= MNT_NODIRATIME;
	if (flags & MS_STRICTATIME)
		mnt_flags &= ~(MNT_RELATIME | MNT_NOATIME);
	if (flags & MS_RDONLY)
```

```
        mnt_flags |= MNT_READONLY;

    flags &= ~(MS_NOSUID | MS_NOEXEC | MS_NODEV | MS_ACTIVE |
           MS_NOATIME | MS_NODIRATIME | MS_RELATIME| MS_KERNMOUNT |
           MS_STRICTATIME);

    if (flags & MS_REMOUNT)
        retval = do_remount(&path, flags & ~MS_REMOUNT, mnt_flags,
                    data_page);
    else if (flags & MS_BIND)
        retval = do_loopback(&path, dev_name, flags & MS_REC);
    else if (flags & (MS_SHARED | MS_PRIVATE | MS_SLAVE | MS_UNBINDABLE))
        retval = do_change_type(&path, flags);
    else if (flags & MS_MOVE)
        retval = do_move_mount(&path, dev_name);
    else
        retval = do_new_mount(&path, type_page, flags, mnt_flags,
                      dev_name, data_page);
```

根据不同的安装方式，选择不同的调用函数。

MS_REMOUNT 表示所要求的只是改变一个原已安装的设备的安装方式。例如，原来是按“只读”方式安装的，而现在要改为“可写”方式。这种操作成为“重安装”。

MS_MOVE：改变安装路径。

在这里，只关系 do_new_mount 函数。

```
dput_out:
    path_put(&path);
    return retval;
}
```

5.2.2.2 do_new_mount

```
static int do_new_mount(struct path *path, char *type, int flags,
                        int mnt_flags, char *name, void *data)
{
        struct vfsmount *mnt;

        if (!type)
                return -EINVAL;

        /* we need capabilities... */
        if (!capable(CAP_SYS_ADMIN))
                return -EPERM;

        lock_kernel();
        mnt = do_kern_mount(type, flags, name, data);
```

调用 do_kern_mount 完成具体功能。

```
        unlock_kernel();
        if (IS_ERR(mnt))
                return PTR_ERR(mnt);

        return do_add_mount(mnt, path, mnt_flags, NULL);
```

将一个 mount 挂入安装树。

```
}
```

5.2.2.2.1 do_kern_mount

```
struct vfsmount *
do_kern_mount(const char *fstype, int flags, const char *name, void *data)
{
        struct file_system_type *type = get_fs_type(fstype);
```

根据文件系统的名称获得 file_system_type 结构

```
	struct vfsmount *mnt;
	if (!type)
		return ERR_PTR(-ENODEV);
	mnt = vfs_kern_mount(type, flags, name, data);
```

调用 vfs_kern_mount。

```
	if (!IS_ERR(mnt) && (type->fs_flags & FS_HAS_SUBTYPE) &&
		!mnt->mnt_sb->s_subtype)
		mnt = fs_set_subtype(mnt, fstype);
	put_filesystem(type);
	return mnt;
}
EXPORT_SYMBOL_GPL(do_kern_mount);
```

vfs_kern_mount

```
struct vfsmount *
vfs_kern_mount(struct file_system_type *type, int flags, const char *name, void *data)
{
	struct vfsmount *mnt;
	char *secdata = NULL;
	int error;

	if (!type)
		return ERR_PTR(-ENODEV);

	error = -ENOMEM;
	mnt = alloc_vfsmnt(name);
```

分配一个 vfsmount 结构。

```
	if (!mnt)
		goto out;
```

```
	if (flags & MS_KERNMOUNT)
		mnt->mnt_flags = MNT_INTERNAL;

	if (data && !(type->fs_flags & FS_BINARY_MOUNTDATA)) {
		secdata = alloc_secdata();
		if (!secdata)
			goto out_mnt;

		error = security_sb_copy_data(data, secdata);
		if (error)
			goto out_free_secdata;
	}

	error = type->get_sb(type, flags, name, data, mnt);
```

调用 ext2_get_sb 函数读入设备的超级块内容。

```
	if (error < 0)
		goto out_free_secdata;
	BUG_ON(!mnt->mnt_sb);
	WARN_ON(!mnt->mnt_sb->s_bdi);

	error = security_sb_kern_mount(mnt->mnt_sb, flags, secdata);
	if (error)
		goto out_sb;

	/*
	 * filesystems should never set s_maxbytes larger than MAX_LFS_FILESIZE
	 * but s_maxbytes was an unsigned long long for many releases. Throw
	 * this warning for a little while to try and catch filesystems that
	 * violate this rule. This warning should be either removed or
```

```
         * converted to a BUG() in 2.6.34.
         */
        WARN((mnt->mnt_sb->s_maxbytes < 0), "%s set sb->s_maxbytes to "
                "negative value (%lld)\n", type->name, mnt->mnt_sb->s_maxbytes);

        mnt->mnt_mountpoint = mnt->mnt_root;
        mnt->mnt_parent = mnt;
```

设置 mnt 的相关字段。

```
        up_write(&mnt->mnt_sb->s_umount);
        free_secdata(secdata);
        return mnt;
out_sb:
        dput(mnt->mnt_root);
        deactivate_locked_super(mnt->mnt_sb);
out_free_secdata:
        free_secdata(secdata);
out_mnt:
        free_vfsmnt(mnt);
out:
        return ERR_PTR(error);
}
```

对于 ext2 文件系统，函数会调用 ext2_get_sb。

```
static int ext2_get_sb(struct file_system_type *fs_type,
        int flags, const char *dev_name, void *data, struct vfsmount *mnt)
{
        return get_sb_bdev(fs_type, flags, dev_name, data, ext2_fill_super, mnt);
}
```

get_sb_bdev

```
int get_sb_bdev(struct file_system_type *fs_type,
	int flags, const char *dev_name, void *data,
	int (*fill_super)(struct super_block *, void *, int),
	struct vfsmount *mnt)
{
	struct block_device *bdev;
	struct super_block *s;
	fmode_t mode = FMODE_READ;
	int error = 0;

	if (!(flags & MS_RDONLY))
		mode |= FMODE_WRITE;

	bdev = open_bdev_exclusive(dev_name, mode, fs_type);
```

该函数根据设备名找到设备对应的 block_device 结构。

```
	if (IS_ERR(bdev))
		return PTR_ERR(bdev);

	/*
	 * once the super is inserted into the list by sget, s_umount
	 * will protect the lockfs code from trying to start a snapshot
	 * while we are mounting
	 */
	mutex_lock(&bdev->bd_fsfreeze_mutex);
	if (bdev->bd_fsfreeze_count > 0) {
		mutex_unlock(&bdev->bd_fsfreeze_mutex);
		error = -EBUSY;
		goto error_bdev;
	}
```

```
	s = sget(fs_type, test_bdev_super, set_bdev_super, bdev);
```

调用 sget()搜索文件系统的超级块对象链表。 如果找到一个与块设备相关的超级块，则返回它的地址。否则，分配并初始化一个新的超级块对象，把它插入到文件系统链表和超级块全局链表中，并返回其地址。

```
	mutex_unlock(&bdev->bd_fsfreeze_mutex);
	if (IS_ERR(s))
		goto error_s;

	if (s->s_root) {
```

如果不是新超级块。

```
		if ((flags ^ s->s_flags) & MS_RDONLY) {
			deactivate_locked_super(s);
			error = -EBUSY;
			goto error_bdev;
		}

		close_bdev_exclusive(bdev, mode);
	} else {
		char b[BDEVNAME_SIZE];

		s->s_flags = flags;
		s->s_mode = mode;
		strlcpy(s->s_id, bdevname(bdev, b), sizeof(s->s_id));
		sb_set_blocksize(s, block_size(bdev));
		error = fill_super(s, data, flags & MS_SILENT ? 1 : 0);
```

如果是新超级块，则调用 ext2_fill_super 函数来从硬盘读入超级块。ext2_fill_super 在 4.1 中已经介绍。

```
		if (error) {
			deactivate_locked_super(s);
			goto error;
		}
```

```
	}

	simple_set_mnt(mnt, s);
	return 0;

error_s:
	error = PTR_ERR(s);
error_bdev:
	close_bdev_exclusive(bdev, mode);
error:
	return error;
}
```

Preceding lines of the listing (continuation):

```
		s->s_flags |= MS_ACTIVE;
		bdev->bd_super = s;
```

5.2.2.2.2 do_add_mount

在 do_new_mount 中，调用 do_kern_mount 后，待安装设备的 super_block 已经解决了，这边已经没有问题了。现在会来看看安装点这边。在读入超级块之前，已经通过 path_init 和 path_walk 找到了安装点的 dentry 结构、inode 结构以及 vfsmount 结构。现在，需要考虑一个问题：从设备上读入超级块是一个漫长的过程，当前进程在等待从设备上读入的过程中几乎肯定要睡眠，这样就可能会发生另一个设备捷足先登抢先将另一个设备安装到了同一个安装点上。如何探测这种情况，在 do_add_mount 函数有处理。

```
int do_add_mount(struct vfsmount *newmnt, struct path *path,
		int mnt_flags, struct list_head *fslist)
{
	int err;

	mnt_flags &= ~(MNT_SHARED | MNT_WRITE_HOLD | MNT_INTERNAL);
```

```
	down_write(&namespace_sem);
	/* Something was mounted here while we slept */
	while (d_mountpoint(path->dentry) &&
	       follow_down(path))
		;
```

处理安装点是否已经被安装。如果安装点上已经有设备，从 follow_down 上可以看出，调用 follow_down 函数前进到已安装设备上的根目录，并且要通过 while 循环进一步检测新的安装点（是否已有设备安装在其上），直到尽头，即前进到不再有设备安装的某个设备上的根节点为准。已安装设备的根目录下一般有内容，是否可以把一个设备安装到一个非空的目录节点上呢？这个是可以的。

```
	err = -EINVAL;
	if (!(mnt_flags & MNT_SHRINKABLE) && !check_mnt(path->mnt))
		goto unlock;

	/* Refuse the same filesystem on the same mount point */
	err = -EBUSY;
	if (path->mnt->mnt_sb == newmnt->mnt_sb &&
		path->mnt->mnt_root == path->dentry)
		goto unlock;

	err = -EINVAL;
	if (S_ISLNK(newmnt->mnt_root->d_inode->i_mode))
		goto unlock;
```

如果要安装的文件系统已经被安装或者安装点是一个符号链接，则返回错误。

```

	newmnt->mnt_flags = mnt_flags;
	if ((err = graft_tree(newmnt, path)))
```

将新安装的文件系统对象插入到 namespace 链表、散列表中。

```
		goto unlock;
```

```
	if (fslist) /* add to the specified expiration list */
		list_add_tail(&newmnt->mnt_expire, fslist);

	up_write(&namespace_sem);
	return 0;

unlock:
	up_write(&namespace_sem);
	mntput(newmnt);
	return err;
}
```

d_mountpoint 函数检测目录 dentry 是否已经有设备安装在其上。

include/linux/dcache.h

```
static inline int d_mountpoint(struct dentry *dentry)
{
	return dentry->d_mounted;
}
```

5.2.2.2.2.1 follow_down函数

fs/namei.c

```
/* no need for dcache_lock, as serialization is taken care in
 * namespace.c
 */
int follow_down(struct path *path)
{
	struct vfsmount *mounted;

	mounted = lookup_mnt(path);
	if (mounted) {
```

```
                dput(path->dentry);
                mntput(path->mnt);
                path->mnt = mounted;
                path->dentry = dget(mounted->mnt_root);
```

前进到安装点的根目录。

```
                return 1;
        }
        return 0;
}
```

lookup_mnt

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *child_mnt;
        spin_lock(&vfsmount_lock);
        if ((child_mnt = __lookup_mnt(path->mnt, path->dentry, 1)))
```

根据 mnt，dentry 查找安装点 vfsmount 的子 vfsmount。

```
                mntget(child_mnt);
        spin_unlock(&vfsmount_lock);
        return child_mnt;
}
```

__lookup_mnt

```
/*
 * find the first or last mount at @dentry on vfsmount @mnt depending on
 * @dir. If @dir is set return the first mount else return the last mount.
 */
struct vfsmount *__lookup_mnt(struct vfsmount *mnt, struct dentry *dentry,
                              int dir)
{
        struct list_head *head = mount_hashtable + hash(mnt, dentry);
```

```
418     struct list_head *tmp = head;
419     struct vfsmount *p, *found = NULL;
420
421     for (;;) {
422             tmp = dir ? tmp->next : tmp->prev;
423             p = NULL;
424             if (tmp == head)
425                     break;
426             p = list_entry(tmp, struct vfsmount, mnt_hash);
427             if (p->mnt_parent == mnt && p->mnt_mountpoint == dentry) {
428                     found = p;
429                     break;
430             }
```

子安装点必须满足两个条件：

- 子 vfsmount 的 mnt_parent 必须指向父 vfsmount。
- 子 vfsmount 的 mnt_mountpoint 必须指向安装点。

```
431     }
432     return found;
433 }
```

427-430 行代码解析

把一个设备安装到一个目录节点时要用一个 vfsmount 数据结构作为“连接件”，该数据结构定义如下。

```
struct vfsmount {
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent;        /* fs we are mounted on */
        struct dentry *mnt_mountpoint;   /* dentry of mountpoint */
        struct dentry *mnt_root;        /* root of the mounted tree */
        struct super_block *mnt_sb; /* pointer to superblock */
        struct list_head mnt_mounts;        /* list of children, anchored here */
        struct list_head mnt_child; /* and going through their mnt_child */
```

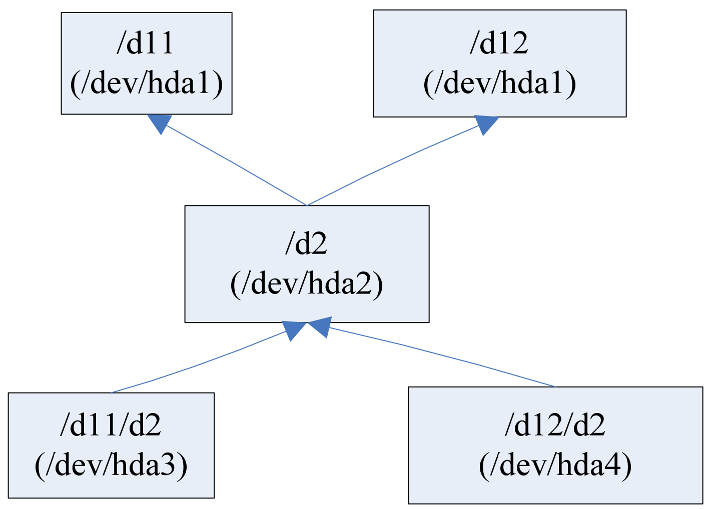
```
        int mnt_flags;
        /* 4 bytes hole on 64bits arches */
        const char *mnt_devname;       /* Name of device e.g. /dev/dsk/hda1 */
        struct list_head mnt_list;
        struct list_head mnt_expire;       /* link in fs-specific expiry list */
        struct list_head mnt_share; /* circular list of shared mounts */
        struct list_head mnt_slave_list;/* list of slave mounts */
        struct list_head mnt_slave; /* slave list entry */
        struct vfsmount *mnt_master;       /* slave is on master->mnt_slave_list */
        struct mnt_namespace *mnt_ns;      /* containing namespace */
        int mnt_id;                 /* mount identifier */
        int mnt_group_id;            /* peer group identifier */
        /*
         * We put mnt_count & mnt_expiry_mark at the end of struct vfsmount
         * to let these frequently modified fields in a separate cache line
         * (so that reads of mnt_flags wont ping-pong on SMP machines)
         */
        atomic_t mnt_count;
        int mnt_expiry_mark;              /* true if marked for expiry */
        int mnt_pinned;
        int mnt_ghosts;
#ifdef CONFIG_SMP
        int __percpu *mnt_writers;
#else
        int mnt_writers;
#endif
};
```

- mnt_mountpoint，mount_root：mnt_mountpoing 指向安装点的 dentry 数据结构,mount_root 指向所安装设备上的根目录的 dentry 数据结构。而则在安装点和安装设备上建立连接。

- dentry 结构中没有直接指向 vfsmount 数据结构的指针，有一个队列头 d_subdirs，这是因为安装点和设备之间是一对多的关系，在同一个安装点上可以安装多个设备。
- mnt_sb 指向所安装设备的超级块 super_block 数据结构。反之，在所安装设备的 super_block 数据结构中并没有直接指向 vfsmount 的数据结构，而是有个队列头 s_instalnces，因为设备与安装点也是一对多的关系，同一个设备可以安装到多个安装点上。
- 指针 mnt_parent 指向安装点所在设备当初安装时的 vfsmount 数据结构，就是上一层的 vfsmount。

所安装设备的 super_block 数据结构与作为“连接件”的 vfsmount 之间可以是一对多的关系，这个很容易理解。可是，安装点 dentry 与 vfsmount 也是一对多的关系，这个就不容易理解了。

通过一个例子来看这个问题：



如上图所示：

假定有/dev/hda1，/dev/hda2，/dev/hda3，/dev/hda4 四个设备，/dev/hda1 为根设备（这四个设备文件节点都在/dev/hda1 上的/dev 目录下），并且，在/dev/hda1 的根目录下有两个空闲的目录节点/d11 和/d12，而在/dev/hda2 的根目录下则有个空闲的目录节点 d2。现在将/dev/hda2 分别安装到/d11 和/d12 上去，这当然是可以的。这样，/dll/d2 和/d12/d2 两个路径通往同一个物理的目录节点。然后，把/dev/hda3 安装到/d11/d2 上，这样/d11/d2 就代表了着/dev/hda3 了。可是/d12/d2 呢？显然应该是空的，因为/d12 代表着一棵独立的子树。再把、/dev/hda4 安装到/d12/d2 上，这也是允许的。

现在，/dev/hda2 上的目录节点 d2 就安装两个设备了，从而有两个 vfsmount 数据结构在其 dentry 结构的 d_ subdirs 队列中。

现在有个问题，当沿路径名搜索时，发现 d2 是个安装点而要前进到所安装的设备上时，怎么知道是前进到/dev/hda3 还是/dev/hda4 呢？显然需要根据搜索路径时的上下文来决定。具体地，要看是顺着/dev/hda2 的那一次安装（每一次安装都有一个 vfsmount）（/d11/d2 或/d12/d2）搜索下来，而上一层的 vfsmount 数据结构就是这个上下文。

5.2.2.2.2.2 graft_tree

```
static int graft_tree(struct vfsmount *mnt, struct path *path)
{
        int err;
        if (mnt->mnt_sb->s_flags & MS_NOUSER)
                return -EINVAL;

        if (S_ISDIR(path->dentry->d_inode->i_mode) !=
                    S_ISDIR(mnt->mnt_root->d_inode->i_mode))
                return -ENOTDIR;

        err = -ENOENT;
        mutex_lock(&path->dentry->d_inode->i_mutex);
        if (cant_mount(path->dentry))
                goto out_unlock;

        err = security_sb_check_sb(mnt, path);
        if (err)
                goto out_unlock;

        err = -ENOENT;
        if (!d_unlinked(path->dentry))
                err = attach_recursive_mnt(mnt, path, NULL);
out_unlock:
        mutex_unlock(&path->dentry->d_inode->i_mutex);
```

```
	if (!err)
		security_sb_post_addmount(mnt, path);
	return err;
}
```

```
static int attach_recursive_mnt(struct vfsmount *source_mnt,
			struct path *path, struct path *parent_path)
{
	LIST_HEAD(tree_list);
	struct vfsmount *dest_mnt = path->mnt;
	struct dentry *dest_dentry = path->dentry;
	struct vfsmount *child, *p;
	int err;

	if (IS_MNT_SHARED(dest_mnt)) {
		err = invent_group_ids(source_mnt, true);
		if (err)
			goto out;
	}
	err = propagate_mnt(dest_mnt, dest_dentry, source_mnt, &tree_list);
	if (err)
		goto out_cleanup_ids;

	spin_lock(&vfsmount_lock);

	if (IS_MNT_SHARED(dest_mnt)) {
		for (p = source_mnt; p; p = next_mnt(p, source_mnt))
			set_mnt_shared(p);
	}
	if (parent_path) {
		detach_mnt(source_mnt, parent_path);
```

```
        attach_mnt(source_mnt, path);
        touch_mnt_namespace(parent_path->mnt->mnt_ns);
    } else {
        mnt_set_mountpoint(dest_mnt, dest_dentry, source_mnt);
        commit_tree(source_mnt);
    }

    list_for_each_entry_safe(child, p, &tree_list, mnt_hash) {
        list_del_init(&child->mnt_hash);
        commit_tree(child);
    }
    spin_unlock(&vfsmount_lock);
    return 0;

out_cleanup_ids:
    if (IS_MNT_SHARED(dest_mnt))
        cleanup_group_ids(source_mnt, NULL);
out:
    return err;
}
```

安装块设备到安装点。

```
void mnt_set_mountpoint(struct vfsmount *mnt, struct dentry *dentry,
                struct vfsmount *child_mnt)
{
    child_mnt->mnt_parent = mntget(mnt);
    child_mnt->mnt_mountpoint = dget(dentry);
    dentry->d_mounted++;
}
```

5.2.3 根文件系统的安装

在 5.2 的开头讲了根文件系统的安装比较特殊。下面来看看具体的实现。

根文件系统的安装时系统初始化中重要的一部分，这个过程比较复杂。

前面讲过，根文件系统的设备号是通过内核参数传递过来的。

当系统启动时，内核就要在变量 ROOT_DEV 中寻找包含根文件系统的磁盘主设备号：

//init/Do_mounts.c

dev_t ROOT_DEV;

当编译内核时，或者向最初的启动装入程序传递一个合适的“root”选项时，根文件系统可以被指定为/dev 目录下的一个设备文件。类似地，根文件系统的安装标志存放在

root_mountflags 变量中：

//init/Do_mounts.c

int root_mountflags = MS_RDONLY | MS_SILENT;

用户可以指定这些标志，或者通过对已编译的内核映像使用 rdev 外部程序，或者向最初的启动装入程序传递一个合适的 rootflags 选项来达到。

安装根文件系统分两个阶段：

（1）内核安装特殊 rootfs 文件系统，该文件系统仅提供一个作为初始安装点的空目录。

（2）内核在空目录上安装实际根文件系统。

为什么内核不怕麻烦，要在安装实际根文件系统之前安装 rootfs 文件系统呢？这是因为，rootfs 文件系统允许内核容易地改变实际根文件系统。实际上，在大多数情况下，系统初始化是内核会逐个地安装和卸载几个根文件系统。例如，一个发布版的初始启动光盘可能把具有一组最小驱动程序的内核装人 RAM 中，内核把存放在 ramdisk 中的一个最小的文件系统作为根安装。接下来，在这个初始根文件系统中的程序探测系统的硬件（例如，它们判断硬盘是否是 EIDE、SCSI 等等），装入所有必需的内核模块，并从物理块设备重新安装根文件系统。

5.2.3.1 安装rootfs文件系统

在内核初始化过程中，会调用 mnt_init 函数。

```
2321 void __init mnt_init(void)
2322 {
```

```
	unsigned u;
	int err;

	init_rwsem(&namespace_sem);

	mnt_cache = kmem_cache_create("mnt_cache", sizeof(struct vfsmount),
			0, SLAB_HWCACHE_ALIGN | SLAB_PANIC, NULL);

	mount_hashtable = (struct list_head *)__get_free_page(GFP_ATOMIC);

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

	printk("Mount-cache hash table entries: %lu\n", HASH_SIZE);

	for (u = 0; u < HASH_SIZE; u++)
		INIT_LIST_HEAD(&mount_hashtable[u]);

	err = sysfs_init();
	if (err)
		printk(KERN_WARNING "%s: sysfs_init error: %d\n",
			__func__, err);
	fs_kobj = kobject_create_and_add("fs", NULL);
	if (!fs_kobj)
		printk(KERN_WARNING "%s: kobj create error\n", __func__);
	init_rootfs();
	init_mount_tree();
```

init_rootfs 和 init_mount_tree 函数实现 rootfs 文件系统的注册等初始化步骤。

```
}
```

fs/ramfs/inode.c

```
static struct file_system_type rootfs_fs_type = {
        .name           = "rootfs",
        .get_sb         = rootfs_get_sb,
        .kill_sb        = kill_litter_super,
};
```

init_rootfs

```
int __init init_rootfs(void)
{
        int err;

        err = bdi_init(&ramfs_backing_dev_info);
        if (err)
                return err;

        err = register_filesystem(&rootfs_fs_type);
```

注册 rootfs 文件系统。

```
        if (err)
                bdi_destroy(&ramfs_backing_dev_info);

        return err;
}
```

init_mount_tree

```
static void __init init_mount_tree(void)
{
        struct vfsmount *mnt;
        struct mnt_namespace *ns;
        struct path root;

```

```
    mnt = do_kern_mount("rootfs", 0, "rootfs", NULL);
```

安装 rootfs 文件系统。

```
    if (IS_ERR(mnt))
        panic("Can't create rootfs");
    ns = create_mnt_ns(mnt);
    if (IS_ERR(ns))
        panic("Can't allocate initial namespace");

    init_task.nsproxy->mnt_ns = ns;
    get_mnt_ns(ns);

    root.mnt = ns->root;
    root.dentry = ns->root->mnt_root;

    set_fs_pwd(current->fs, &root);
    set_fs_root(current->fs, &root);
}
```

5.2.3.2 实际根文件系统安装

在内核初始化的最后阶段，会调用 prepare_namespace 函数进行实际根文件系统的安装。

init/do_mounts.c

```
void __init prepare_namespace(void)
{
    int is_floppy;

    if (root_delay) {
        printk(KERN_INFO "Waiting %dsec before mounting root device...\n",
               root_delay);
```

```
		ssleep(root_delay);
	}

	/*
	 * wait for the known devices to complete their probing
	 *
	 * Note: this is a potential source of long boot delays.
	 * For example, it is not atypical to wait 5 seconds here
	 * for the touchpad of a laptop to initialize.
	 */
	wait_for_device_probe();

	md_run_setup();

	if (saved_root_name[0]) {
		root_device_name = saved_root_name;
```

把 root_device_name 变量置为从启动参数“root”中获取的设备文件名。同样，把 ROOT_DEV 变量置为同一设备文件的主设备号和次设备号。

```
		if (!strncmp(root_device_name, "mtd", 3) ||
			!strncmp(root_device_name, "ubi", 3)) {
			mount_block_root(root_device_name, root_mountflags);
```

调用 mount_block_root()函数，将最常用的块设备作为 rootfs 文件系统的子文件系统。

```
			goto out;
		}
		ROOT_DEV = name_to_dev_t(root_device_name);
		if (strncmp(root_device_name, "/dev/", 5) == 0)
			root_device_name += 5;
	}

	if (initrd_load())
```

```
		goto out;
```

如果加载用户指定的设备失败，则根据内核配置参数（是否设置了加载 initrd 的参数），决定是否加载 initrd。

```

	/* wait for any asynchronous scanning to complete */
	if ((ROOT_DEV == 0) && root_wait) {
		printk(KERN_INFO "Waiting for root device %s...\n",
			saved_root_name);
		while (driver_probe_done() != 0 ||
			(ROOT_DEV = name_to_dev_t(saved_root_name)) == 0)
			msleep(100);
		async_synchronize_full();
	}

	is_floppy = MAJOR(ROOT_DEV) == FLOPPY_MAJOR;

	if (is_floppy && rd_doload && rd_load_disk(0))
		ROOT_DEV = Root_RAM0;

	mount_root();//大部分时间会调用该函数。
```

如果上面都失败，调用 mount_root 进一步努力。mount_root 可以看做是 mount_block_root 封装，只是将文件的设备号变为文件路径。

```
out:
	devtmpfs_mount("dev");
	sys_mount(".", "/", NULL, MS_MOVE, NULL);
```

将当前的目录移动到根目录下。

```
	sys_chroot(".");
}
```

5.2.3.2.1 mount_block_root

```
void __init mount_block_root(char *name, int flags)
{
        char *fs_names = __getname_gfp(GFP_KERNEL
                | __GFP_NOTRACK_FALSE_POSITIVE);
        char *p;
#ifdef CONFIG_BLOCK
        char b[BDEVNAME_SIZE];
#else
        const char *b = name;
#endif

        get_fs_names(fs_names);
retry:
        for (p = fs_names; *p; p += strlen(p)+1) {
                int err = do_mount_root(name, p, flags, root_mount_data);
```

完成实际的操作。

```
                switch (err) {
                        case 0:
                                goto out;
                        case -EACCES:
                                flags |= MS_RDONLY;
                                goto retry;
                        case -EINVAL:
                                continue;
                }
                        /*
                 * Allow the user to distinguish between failed sys_open
                 * and bad superblock on root device.
                 * and give them a list of the available devices
                 */
```

```
#ifdef CONFIG_BLOCK
		__bdevname(ROOT_DEV, b);
#endif
		printk("VFS: Cannot open root device \"%s\" or %s\n",
				root_device_name, b);
		printk("Please append a correct \"root=\" boot option; here are the available partitions:\n");

		printk_all_partitions();
#ifdef CONFIG_DEBUG_BLOCK_EXT_DEVT
		printk("DEBUG_BLOCK_EXT_DEVT is enabled, you need to specify "
			"explicit textual name for \"root=\" boot option.\n");
#endif
		panic("VFS: Unable to mount root fs on %s", b);
	}

	printk("List of all partitions:\n");
	printk_all_partitions();
	printk("No filesystem could mount root, tried: ");
	for (p = fs_names; *p; p += strlen(p)+1)
		printk(" %s", p);
	printk("\n");
#ifdef CONFIG_BLOCK
	__bdevname(ROOT_DEV, b);
#endif
	panic("VFS: Unable to mount root fs on %s", b);
out:
	putname(fs_names);
}
```

do_mount_root

```
static int __init do_mount_root(char *name, char *fs, int flags, void *data)
{
        int err = sys_mount(name, "/root", fs, flags, data);
```

将设备安装到 rootfs 的/root 目录下。

```
        if (err)
                return err;

        sys_chdir("/root");
```

切换当前目录为/root。

```
        ROOT_DEV = current->fs->pwd.mnt->mnt_sb->s_dev;
        printk("VFS: Mounted root (%s filesystem)%s on device %u:%u.\n",
               current->fs->pwd.mnt->mnt_sb->s_type->name,
               current->fs->pwd.mnt->mnt_sb->s_flags & MS_RDONLY ?
               " readonly" : "", MAJOR(ROOT_DEV), MINOR(ROOT_DEV));
        return 0;
}
```

5.2.3.2.2 mount_root

```
void __init mount_root(void)
{
#ifdef CONFIG_ROOT_NFS
        if (MAJOR(ROOT_DEV) == UNNAMED_MAJOR) {
                if (mount_nfs_root())
                        return;

                printk(KERN_ERR "VFS: Unable to mount root fs via NFS, trying floppy.\n");
                ROOT_DEV = Root_FD0;
        }
#endif
```

```
#ifdef CONFIG_BLK_DEV_FD
        if (MAJOR(ROOT_DEV) == FLOPPY_MAJOR) {
                /* rd_doload is 2 for a dual initrd/ramload setup */
                if (rd_doload==2) {
                        if (rd_load_disk(1)) {
                                ROOT_DEV = Root_RAM1;
                                root_device_name = NULL;
                        }
                } else
                        change_floppy("root floppy");
        }
#endif
#ifdef CONFIG_BLOCK
        create_dev("/dev/root", ROOT_DEV);
```

在 rootfs 文件系统上创建文件/dev/root，文件表示一个设备文件，inode 中存放的是文件的设备号。

```
        mount_block_root("/dev/root", root_mountflags);
```

调用 mount_block_root 函数安装设备文件到 rootfs 的/root 目录下。

```
#endif
}
```

5.2.4 sys_umount文件系统的拆卸

```
SYSCALL_DEFINE2(umount, char __user *, name, int, flags)
{
        struct path path;
        int retval;
        int lookup_flags = 0;

        if (flags & ~(MNT_FORCE | MNT_DETACH | MNT_EXPIRE |
UMOUNT_NOFOLLOW))
```

```
                return -EINVAL;

        if (!(flags & UMOUNT_NOFOLLOW))
                lookup_flags |= LOOKUP_FOLLOW;

        retval = user_path_at(AT_FDCWD, name, lookup_flags, &path);
        if (retval)
                goto out;
        retval = -EINVAL;
        if (path.dentry != path.mnt->mnt_root)
                goto dput_and_out;
        if (!check_mnt(path.mnt))
                goto dput_and_out;

        retval = -EPERM;
        if (!capable(CAP_SYS_ADMIN))
                goto dput_and_out;

        retval = do_umount(path.mnt, flags);
dput_and_out:
        /* we mustn't call path_put() as that would clear mnt_expiry_mark */
        dput(path.dentry);
        mntput_no_expire(path.mnt);
out:
        return retval;
}
```

5.2.4.1 do_umount

```
static int do_umount(struct vfsmount *mnt, int flags)
```

```
{
        struct super_block *sb = mnt->mnt_sb;
        int retval;
        LIST_HEAD(umount_list);

        retval = security_sb_umount(mnt, flags);
        if (retval)
                return retval;

        /*
         * Allow userspace to request a mountpoint be expired rather than
         * unmounting unconditionally. Unmount only happens if:
         *  (1) the mark is already set (the mark is cleared by mntput())
         *  (2) the usage count == 1 [parent vfsmount] + 1 [sys_umount]
         */
        if (flags & MNT_EXPIRE) {
                if (mnt == current->fs->root.mnt ||
                        flags & (MNT_FORCE | MNT_DETACH))
                        return -EINVAL;

                if (atomic_read(&mnt->mnt_count) != 2)
                        return -EBUSY;

                if (!xchg(&mnt->mnt_expiry_mark, 1))
                        return -EAGAIN;
        }

        /*
         * If we may have to abort operations to get out of this
         * mount, and they will themselves hold resources we must
         * allow the fs to do things. In the Unix tradition of
```

```
         * 'Gee thats tricky lets do it in userspace' the umount_begin
         * might fail to complete on the first run through as other tasks
         * must return, and the like. Thats for the mount program to worry
         * about for the moment.
         */

        if (flags & MNT_FORCE && sb->s_op->umount_begin) {
                sb->s_op->umount_begin(sb);
        }

        /*
         * No sense to grab the lock for this test, but test itself looks
         * somewhat bogus. Suggestions for better replacement?
         * Ho-hum... In principle, we might treat that as umount + switch
         * to rootfs. GC would eventually take care of the old vfsmount.
         * Actually it makes sense, especially if rootfs would contain a
         * /reboot - static binary that would close all descriptors and
         * call reboot(9). Then init(8) could umount root and exec /reboot.
         */
        if (mnt == current->fs->root.mnt && !(flags & MNT_DETACH)) {
                /*
                 * Special case for "unmounting" root ...
                 * we just try to remount it readonly.
                 */
                down_write(&sb->s_umount);
                if (!(sb->s_flags & MS_RDONLY))
                        retval = do_remount_sb(sb, MS_RDONLY, NULL, 0);
                up_write(&sb->s_umount);
                return retval;
        }
```

如果要卸载的文件系统是根文件系统，且用户并不要求真正地把它卸载下来则调用 do_remount_sb()重新安装根文件系统为只读并终止

```

	down_write(&namespace_sem);
	spin_lock(&vfsmount_lock);
	event++;

	if (!(flags & MNT_DETACH))
		shrink_submounts(mnt, &umount_list);

	retval = -EBUSY;
	if (flags & MNT_DETACH || !propagate_mount_busy(mnt, 2)) {
		if (!list_empty(&mnt->mnt_list))
			umount_tree(mnt, 1, &umount_list);
```

如果已安装文件系统不包含任何子安装文件系统的安装点，或者用户要求强制卸载文件系统，则调用 umount_tree()卸载文件系统（及其所有子文件系统）

```
		retval = 0;
	}
	spin_unlock(&vfsmount_lock);
	if (retval)
		security_sb_umount_busy(mnt);
	up_write(&namespace_sem);
	release_mounts(&umount_list);
	return retval;
}
```

5.3 文件的打开与关闭

5.3.1 文件的打开do_sys_open

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode)
{
        long ret;

        if (force_o_largefile())
                flags |= O_LARGEFILE;

        ret = do_sys_open(AT_FDCWD, filename, flags, mode);
        /* avoid REGPARM breakage on x86: */
        asmlinkage_protect(3, ret, filename, flags, mode);
        return ret;
}
```

```
long do_sys_open(int dfd, const char __user *filename, int flags, int mode)
{
        char *tmp = getname(filename);
        int fd = PTR_ERR(tmp);

        if (!IS_ERR(tmp)) {
                fd = get_unused_fd_flags(flags);
```

取得一个为使用的文件号

```
                if (fd >= 0) {
                        struct file *f = do_filp_open(dfd, tmp, flags, mode, 0);
```

具体功能函数

```
                        if (IS_ERR(f)) {
```

```
                                put_unused_fd(fd);
                                fd = PTR_ERR(f);
                        } else {
                                fsnotify_open(f->f_path.dentry);
                                fd_install(fd, f);
```

安装 file 结构到 fd 对应的位置。

```
                        }
                }
                putname(tmp);
        }
        return fd;
}
```

do_filp_open

```
/*
 * Note that the low bits of the passed in "open_flag"
 * are not the same as in the local variable "flag". See
 * open_to_namei_flags() for more details.
 */
struct file *do_filp_open(int dfd, const char *pathname,
                int open_flag, int mode, int acc_mode)
{
        struct file *filp;
        struct nameidata nd;
        int error;
        struct path path;
        int count = 0;
        int flag = open_to_namei_flags(open_flag);
        int force_reval = 0;

        if (!(open_flag & O_CREAT))
```

```
		mode = 0;

	/*
	 * O_SYNC is implemented as __O_SYNC|O_DSYNC.  As many places only
	 * check for O_DSYNC if the need any syncing at all we enforce it's
	 * always set instead of having to deal with possibly weird behaviour
	 * for malicious applications setting only __O_SYNC.
	 */
	if (open_flag & __O_SYNC)
		open_flag |= O_DSYNC;

	if (!acc_mode)
		acc_mode = MAY_OPEN | ACC_MODE(open_flag);

	/* O_TRUNC implies we need access checks for write permissions */
	if (open_flag & O_TRUNC)
		acc_mode |= MAY_WRITE;

	/* Allow the LSM permission hook to distinguish append
	   access from general write access. */
	if (open_flag & O_APPEND)
		acc_mode |= MAY_APPEND;

	/* find the parent */
reval:
	error = path_init(dfd, pathname, LOOKUP_PARENT, &nd);
	if (error)
		return ERR_PTR(error);
	if (force_reval)
		nd.flags |= LOOKUP_REVAL;

```

```
current->total_link_count = 0;
error = link_path_walk(pathname, &nd);
```

调用 path_init，link_path_walk 到父目录。

```
if (error) {
        filp = ERR_PTR(error);
        goto out;
}
if (unlikely(!audit_dummy_context()) && (open_flag & O_CREAT))
        audit_inode(pathname, nd.path.dentry);

/*
 * We have the parent and last component.
 */

error = -ENFILE;
filp = get_empty_filp();
if (filp == NULL)
        goto exit_parent;
nd.intent.open.file = filp;
filp->f_flags = open_flag;
nd.intent.open.flags = flag;
nd.intent.open.create_mode = mode;
nd.flags &= ~LOOKUP_PARENT;
nd.flags |= LOOKUP_OPEN;
if (open_flag & O_CREAT) {
        nd.flags |= LOOKUP_CREATE;
        if (open_flag & O_EXCL)
                nd.flags |= LOOKUP_EXCL;
}
if (open_flag & O_DIRECTORY)
        nd.flags |= LOOKUP_DIRECTORY;
```

```
	if (!(open_flag & O_NOFOLLOW))
		nd.flags |= LOOKUP_FOLLOW;
```

设置相应的处理参数。

```
	filp = do_last(&nd, &path, open_flag, acc_mode, mode, pathname);
```

处理最后一个节点。

```
	while (unlikely(!filp)) { /* trailing symlink */
```

处理符号链接。

```
		struct path holder;
		struct inode *inode = path.dentry->d_inode;
		void *cookie;
		error = -ELOOP;
		/* S_ISDIR part is a temporary automount kludge */
		if (!(nd.flags & LOOKUP_FOLLOW) && !S_ISDIR(inode->i_mode))
			goto exit_dput;
		if (count++ == 32)
			goto exit_dput;
		/*
		 * This is subtle. Instead of calling do_follow_link() we do
		 * the thing by hands. The reason is that this way we have zero
		 * link_count and path_walk() (called from ->follow_link)
		 * honoring LOOKUP_PARENT.  After that we have the parent and
		 * last component, i.e. we are in the same situation as after
		 * the first path_walk().  Well, almost - if the last component
		 * is normal we get its copy stored in nd->last.name and we will
		 * have to putname() it when we are done. Procfs-like symlinks
		 * just set LAST_BIND.
		 */
		nd.flags |= LOOKUP_PARENT;
		error = security_inode_follow_link(path.dentry, &nd);
```

```
		if (error)
			goto exit_dput;
		error = __do_follow_link(&path, &nd, &cookie);
		if (unlikely(error)) {
			/* nd.path had been dropped */
			if (!IS_ERR(cookie) && inode->i_op->put_link)
				inode->i_op->put_link(path.dentry, &nd, cookie);
			path_put(&path);
			release_open_intent(&nd);
			filp = ERR_PTR(error);
			goto out;
		}
		holder = path;
		nd.flags &= ~LOOKUP_PARENT;
		filp = do_last(&nd, &path, open_flag, acc_mode, mode, pathname);
```

调用 do_last 来处理函数。

```
		if (inode->i_op->put_link)
			inode->i_op->put_link(holder.dentry, &nd, cookie);
		path_put(&holder);
	}
out:
	if (nd.root.mnt)
		path_put(&nd.root);
	if (filp == ERR_PTR(-ESTALE) && !force_reval) {
		force_reval = 1;
		goto reval;
	}
	return filp;

exit_dput:
```

```
	path_put_conditional(&path, &nd);
	if (!IS_ERR(nd.intent.open.file))
		release_open_intent(&nd);
exit_parent:
	path_put(&nd.path);
	filp = ERR_PTR(error);
	goto out;
}
```

do_last 函数

```
static struct file *do_last(struct nameidata *nd, struct path *path,
				int open_flag, int acc_mode,
				int mode, const char *pathname)
{
	struct dentry *dir = nd->path.dentry;
	struct file *filp;
	int error = -EISDIR;

	switch (nd->last_type) {
	case LAST_DOTDOT:
		follow_dotdot(nd);
		dir = nd->path.dentry;
		if (nd->path.mnt->mnt_sb->s_type->fs_flags & FS_REVAL_DOT) {
			if (!dir->d_op->d_revalidate(dir, nd)) {
				error = -ESTALE;
				goto exit;
			}
		}
		/* fallthrough */
	case LAST_DOT:
```

```
	case LAST_ROOT:
		if (open_flag & O_CREAT)
			goto exit;
		/* fallthrough */
	case LAST_BIND:
		audit_inode(pathname, dir);
		goto ok;
	}
```

特需节点处理

```

	/* trailing slashes? */
	if (nd->last.name[nd->last.len]) {
		if (open_flag & O_CREAT)
			goto exit;
		nd->flags |= LOOKUP_DIRECTORY | LOOKUP_FOLLOW;
	}

	/* just plain open? */
	if (!(open_flag & O_CREAT)) {
		error = do_lookup(nd, &nd->last, path);
		if (error)
			goto exit;
		error = -ENOENT;
		if (!path->dentry->d_inode)
			goto exit_dput;
		if (path->dentry->d_inode->i_op->follow_link)
			return NULL;
		error = -ENOTDIR;
		if (nd->flags & LOOKUP_DIRECTORY) {
			if (!path->dentry->d_inode->i_op->lookup)
				goto exit_dput;
```

```
		}
		path_to_nameidata(path, nd);
		audit_inode(pathname, nd->path.dentry);
		goto ok;
	}
```

处理非创建的打开

```

	/* OK, it's O_CREAT */
	mutex_lock(&dir->d_inode->i_mutex);

	path->dentry = lookup_hash(nd);
	path->mnt = nd->path.mnt;

	error = PTR_ERR(path->dentry);
	if (IS_ERR(path->dentry)) {
		mutex_unlock(&dir->d_inode->i_mutex);
		goto exit;
	}

	if (IS_ERR(nd->intent.open.file)) {
		error = PTR_ERR(nd->intent.open.file);
		goto exit_mutex_unlock;
	}

	/* Negative dentry, just create the file */
	if (!path->dentry->d_inode) {
		/*
		 * This write is needed to ensure that a
		 * ro->rw transition does not occur between
		 * the time when the file is created and when
		 * a permanent write count is taken through
```

```
		 * the 'struct file' in nameidata_to_filp().
		 */
		error = mnt_want_write(nd->path.mnt);
		if (error)
			goto exit_mutex_unlock;
		error = __open_namei_create(nd, path, open_flag, mode);
		if (error) {
			mnt_drop_write(nd->path.mnt);
			goto exit;
		}
		filp = nameidata_to_filp(nd);
		mnt_drop_write(nd->path.mnt);
		if (!IS_ERR(filp)) {
			error = ima_file_check(filp, acc_mode);
			if (error) {
				fput(filp);
				filp = ERR_PTR(error);
			}
		}
		return filp;
	}
```

创建文件处理

```

	/*
	 * It already exists.
	 */
	mutex_unlock(&dir->d_inode->i_mutex);
	audit_inode(pathname, path->dentry);

	error = -EEXIST;
	if (open_flag & O_EXCL)
```

```
			goto exit_dput;

	if (__follow_mount(path)) {
		error = -ELOOP;
		if (open_flag & O_NOFOLLOW)
			goto exit_dput;
	}

	error = -ENOENT;
	if (!path->dentry->d_inode)
		goto exit_dput;

	if (path->dentry->d_inode->i_op->follow_link)
		return NULL;

	path_to_nameidata(path, nd);
	error = -EISDIR;
	if (S_ISDIR(path->dentry->d_inode->i_mode))
		goto exit;
ok:
	filp = finish_open(nd, open_flag, acc_mode);
```

inode 创建成功后，分配 file 结构。

```
	return filp;

exit_mutex_unlock:
	mutex_unlock(&dir->d_inode->i_mutex);
exit_dput:
	path_put_conditional(path, nd);
exit:
	if (!IS_ERR(nd->intent.open.file))
		release_open_intent(nd);
```

```
        path_put(&nd->path);
        return ERR_PTR(error);
}
```

5.3.1.1 __open_namei_create

```
/*
 * Be careful about ever adding any more callers of this
 * function.   Its flags must be in the namei format, not
 * what get passed to sys_open().
 */
static int __open_namei_create(struct nameidata *nd, struct path *path,
                                int open_flag, int mode)
{
        int error;
        struct dentry *dir = nd->path.dentry;

        if (!IS_POSIXACL(dir->d_inode))
                mode &= ~current_umask();
        error = security_path_mknod(&nd->path, path->dentry, mode, 0);
        if (error)
                goto out_unlock;
        error = vfs_create(dir->d_inode, path->dentry, mode, nd);
out_unlock:
        mutex_unlock(&dir->d_inode->i_mutex);
        dput(nd->path.dentry);
        nd->path.dentry = path->dentry;
        if (error)
                return error;
        /* Don't check for write permission, don't truncate */
        return may_open(&nd->path, 0, open_flag & ~O_TRUNC);
```

```
}
```

vfs_create

```
int vfs_create(struct inode *dir, struct dentry *dentry, int mode,
		struct nameidata *nd)
{
	int error = may_create(dir, dentry);

	if (error)
		return error;

	if (!dir->i_op->create)
		return -EACCES; /* shouldn't it be ENOSYS? */
	mode &= S_IALLUGO;
	mode |= S_IFREG;
	error = security_inode_create(dir, dentry, mode);
	if (error)
		return error;
	error = dir->i_op->create(dir, dentry, mode, nd);
	if (!error)
		fsnotify_create(dir, dentry);
```

文件系统创建通知。

```
	return error;
}
```

ext2_create

```
/*
 * By the time this is called, we already have created
 * the directory cache entry for the new file, but it
 * is so far negative - it has no inode.
 *
```

```
 * If the create succeeds, we fill in the inode information
 * with d_instantiate().
 */
static int ext2_create (struct inode * dir, struct dentry * dentry, int mode, struct nameidata *nd)
{
	struct inode *inode;

	dquot_initialize(dir);

	inode = ext2_new_inode(dir, mode);
```

在磁盘上分配一个 inode 节点，该函数在前面已分析。

```
	if (IS_ERR(inode))
		return PTR_ERR(inode);

	inode->i_op = &ext2_file_inode_operations;
	if (ext2_use_xip(inode->i_sb)) {
		inode->i_mapping->a_ops = &ext2_aops_xip;
		inode->i_fop = &ext2_xip_file_operations;
	} else if (test_opt(inode->i_sb, NOBH)) {
		inode->i_mapping->a_ops = &ext2_nobh_aops;
		inode->i_fop = &ext2_file_operations;
	} else {
		inode->i_mapping->a_ops = &ext2_aops;
		inode->i_fop = &ext2_file_operations;
	}
```

设置 inode 的相关字段

```
	mark_inode_dirty(inode);
```

设置 inode 为脏，回写到磁盘。

```
	return ext2_add_nondir(dentry, inode);
```

将目录项写入到父目录项中，同事将父目录的 inode 设置为脏。

```
}
```

5.3.2 文件关闭sys_close

```
SYSCALL_DEFINE1(close, unsigned int, fd)
{
	struct file * filp;
	struct files_struct *files = current->files;
	struct fdtable *fdt;
	int retval;

	spin_lock(&files->file_lock);
	fdt = files_fdtable(files);
	if (fd >= fdt->max_fds)
		goto out_unlock;
	filp = fdt->fd[fd];
	if (!filp)
		goto out_unlock;
	rcu_assign_pointer(fdt->fd[fd], NULL);
	FD_CLR(fd, fdt->close_on_exec);
	__put_unused_fd(files, fd);
	spin_unlock(&files->file_lock);
	retval = filp_close(filp, files);

	/* can't restart close syscall because file table entry was cleared */
	if (unlikely(retval == -ERESTARTSYS ||
		     retval == -ERESTARTNOINTR ||
		     retval == -ERESTARTNOHAND ||
		     retval == -ERESTART_RESTARTBLOCK))
		retval = -EINTR;

	return retval;
```

```
out_unlock:
        spin_unlock(&files->file_lock);
        return -EBADF;
}
EXPORT_SYMBOL(sys_close);
```

```
int filp_close(struct file *filp, fl_owner_t id)
{
        int retval = 0;
        if (!file_count(filp)) {
                printk(KERN_ERR "VFS: Close: file count is 0\n");
                return 0;
        }
        if (filp->f_op && filp->f_op->flush)
                retval = filp->f_op->flush(filp, id);
        dnotify_flush(filp, id);
        locks_remove_posix(filp, id);
        fput(filp);
        return retval;
}
```

5.4 文件的读写

5.4.1 概述

在打开文件后，或者说建立起进程与文件之间的“连接”后，才能对文件进行读写。为了调高效率，复杂一些的操作系统对文件的读/写都是带缓存的，linux 也不例外。像 VFS 一样，Linux 文件系统的缓冲机制也是它的一

大特色。所谓缓冲，是指系统为最近刚读/写过的文件在内核中保留的一份副本，以便当再次需要已经缓存在副本中的内容时就不必再临时从设备上读入，而需要写的时候则可以先写入到副本中，待系统较为空闲时再重副本写入设备。在多进程的系统中，由于同一文件可能为多个进程所共享，缓冲的作用就更显著了。

然而，1>怎样实现缓冲，2>在哪一个层次上实现缓冲，却是一个值得仔细考虑的问题。在系统中处于最高层的是进程，这一层可以称为“应用层”，在用户空间运行的，在这里代表着目标文件的是“打开文件号”。在这一层中提供缓冲似乎最贴近文件内容的使用者，但是那样就需要用户的介入，从而不能做到对使用者的“透明”。并且缓冲的内容不能为其它进程所共享，显然是不妥的。在应用层以下是“文件层”，又可细分为 VFS 层和具体的文件系统层，在下面就是“设备层”了。这些层都在内核中，所以在这些层上实现缓冲都可以达到对用户透明的目标。设备层最贴近设备，即文件内容的“源头“的地方，在这里实现缓冲显然是可行的，事实上早期的 Unix 内核中的文件缓冲就是以数据块缓冲的形式在这一层上实现的。但是，设备层上的缓冲区离使用者的距离太远了一点，特别是当文件层有分为 VFS 层和具体文件系统子层是，每次读/写都要穿越这么多界面深入到设备层就难免会耗时。

在文件层中有三种主要的数据结构，就是 file、dentry、inode。

一个 file 代表着一个文件的上下文，不但不同的进程可以在同一个文件上建立不同的上下文，就是同一个进程也可以同事打开一个文件多次而建立起多个上下文。如果在 file 结构中设备一个缓冲区队列，那么缓存区中的内容虽然贴近这个特定上下文的使用者，却不便于在多个进程中共享，甚至不便于在同一个进程打开的不同上下文“共享”。这显然是不合适的，需要把这些缓存区公用的地方提取出来。

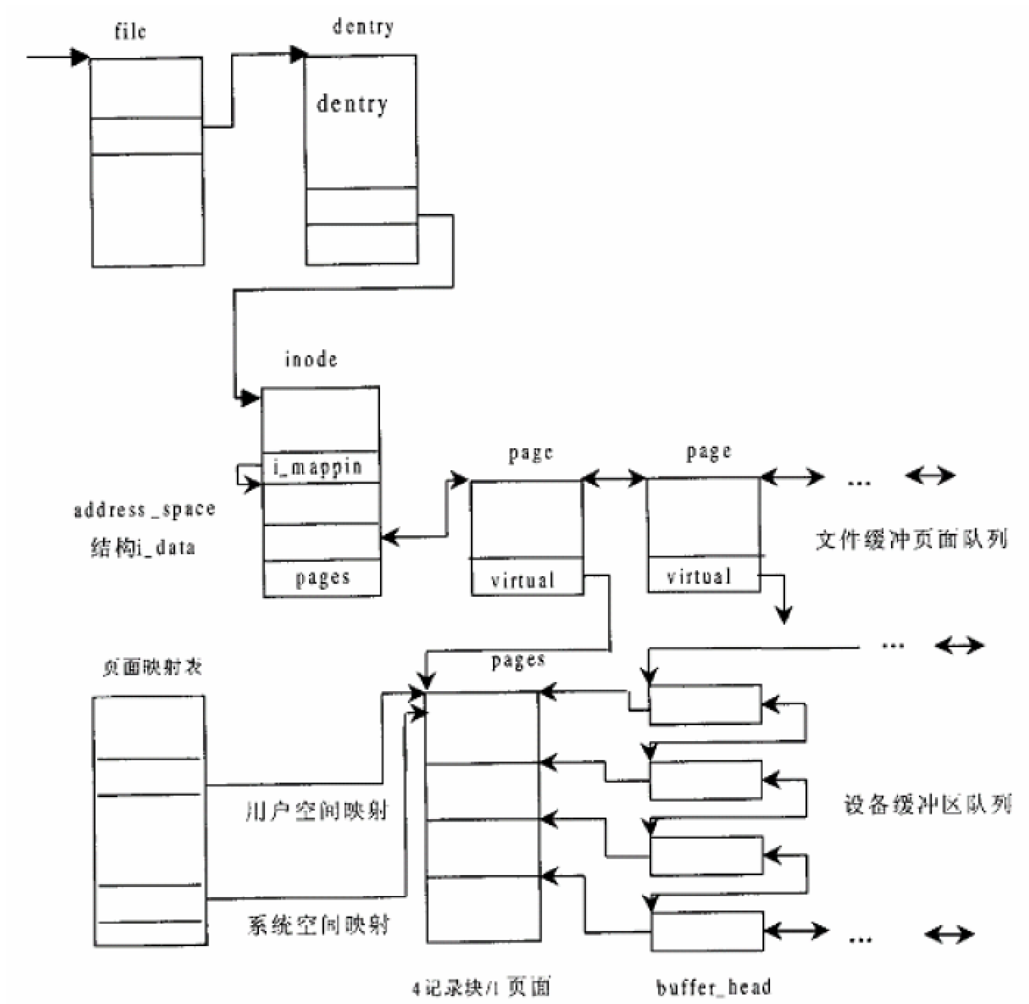
那么 dentry 结构如何？这个数据结构并不属于某一个上下文，也不属于某一个进程，可以为所有的进程和上下文共享。可是，dentry 结构与目标文件并不是一对一的关系，通过文件连接（硬连接），可以为已经存在的文件建立“别名”。一个 dentry 结构知识唯一的代表着这个文件系统中的一个节点，也就是一个路径名，但是多个节点可以同时代表同一个文件。

显然，在 inode 数据结构中设置一个缓冲区队列是最合适不过的了，首先，inode 结构与文件是一对一的关系，即使一个文件有多个路径名，最后也归纳到一个 inode 结构上。其次，一个文件中的内容是不能由其他文件共享的，在同一时间里，设备上的每一个记录块都只能至多属于一个文件（或者空闲），将载有同一文件内容的缓冲区都放在其所属文件的 inode 结构中

是很自然的是。因此，inode 结构中设置了一个指针 i_mapping，它指向一个 address_space 数据结构（通常这个数据结构就是 inode 结构中的 i_data0），缓存区队列就在这个数据结构中。

不过，挂在缓冲区队列中的并不是记录块而是内存页面。也就是说，文件的内容并不是以记录块为单位，而是已页面为单位进行缓冲的。如果记录块的大小为 1k 字节，那么一个页面就相当于 4 个记录块。为什么要这么做呢？这是为了将文件内容的缓冲与文件的内存映射结合在一起。在内存管理中讲过，一个进程可以通过 mmap 将一个文件映射到它的用户空间。建立了这样的映射后，就可以像访问内存一样访问这个文件。如果将文件的内容以页面为单位进行缓存，放在属于该文件的 inode 结构的缓冲队列中，那么只需要相应地设置进程的内存映射表，就可以自然地将这些缓冲页面映射到进程的用户空间中。这样，在按常规的文件访问一个文件时，可以通过 read 和 write 系统调用目标文件的 inode 结构访问这些缓冲页面；而通过内存映射机制访问这个文件时，就可以经由页面映射表直接读写这些缓冲着的页面。当目标页面不存在内存中时，常规文件通过系统调用 read,write 的底层将其从设备上读入，而内存映射机制则通过“缺页异常”的服务将目标页面从设备上读入。也就是所，同一个缓冲页面可以满足两方面的要求，文件系统的缓冲机制和文件的内存映射机制巧妙地结合在一起了。

可是，尽管以页面为单位的缓冲对于文件层来说确实是一个很好的选择，对于设备层则不那么合适了。对设备层而言，最自然的当然是以记录块为单位进行缓冲，因为设备的读/写都是以记录块为单位的。不过，从磁盘上读/写时主要的时间都花在准备工作（如磁头组的定位）上，一旦准备好了后读一个记录块与接连读几个记录块相差不大，而且每次只读写一个记录块反而是不经济的。所以每次读写若干连续的记录块、以页面为单位缓冲也并不成问题。另一方面，如果以页面为单位缓冲，而一个页面相当于若干个记录块，那么无论是对于缓冲页面还是对于记录块缓冲区，其控制和附加信息（如链接指针等）显然应该游离于页面之外，这些信息不应该映射到进程的用户空间。这个问题也不难解决，在“缓冲区头部”即 buffer_head 数据结构中有一个 b_data 指向缓冲区，而 buffer_head 结构本身不在缓冲区中。所以，在设备层中只要保持一些 buffer_head 结构，让它们的 b_data 指针分别指向缓冲区页面中的相应位置就可以了。这样，以页面为单位为文件内容建立缓冲是“一箭三雕”，如下示意图。



在这样一个结构框架中，一旦所欲访问的内容已经在缓冲页面队列中，读文件的效率就很高了。只要找到文件的 inode 结构（file->dentry->inode）就找到了缓冲区页面队列，从队列中找到相应的页面就可以读出了。

那么，写操作又如何呢？如前所述，一旦目标记录块已经存在与缓冲页面中，写操作只是把内容写到该缓冲页面中，所以从发动写操作的进程的角度看数度也是很快的。至于改变了内容的缓冲页面，则由系统负责在 CPU 较为空闲写入设备。

5.4.2 相关数据结构

5.4.2.1 inode

在 linux 系统中，磁盘上的一个文件由 indoe 索引节点唯一的标识。

include/linux/fs.h

```
struct inode {
 ……
        struct address_space      *i_mapping;
        struct address_space      i_data;
……
};
```

■i_mapping：指向缓存 address_space 对象的指针，该结构是文件系统缓存的关键。

■i_data：嵌入在 inode 中的 address_space 对象。

5.4.2.2 page

页帧代表系统内存的最小单位，对内存中的每个页都会创建 struct page 的一个实例。

内核设计需要尽量保证该结构尽量小。

由于 page 结构会由于很多地方，所以其中有很多字段在某种情况下有用，而在另外的情况下无用，为了保证 page 字段尽量小，内核使用联合的方式将同一内存出的内容做不同的解释。

```
/*
 * Each physical page in the system has a struct page associated with
 * it to keep track of whatever it is we are using the page for at the
 * moment. Note that we have no way to track which tasks are using
 * a page, though if it is a pagecache page, rmap structures can tell us
 * who is mapping it.
 */
struct page {
        unsigned long flags;            /* Atomic flags, some possibly
                                         * updated asynchronously */
        atomic_t _count;                /* Usage count, see below. */
        union {
                atomic_t _mapcount; /* Count of ptes mapped in mms,
```

```
                                * to show when page is mapped
                                * & limit reverse map searches.
                                */
                struct {                /* SLUB */
                        u16 inuse;
                        u16 objects;
                };
        };
        union {
                struct {
                unsigned long private;          /* Mapping-private opaque data:
                                        * usually used for buffer_heads
                                        * if PagePrivate set; used for
                                        * swp_entry_t if PageSwapCache;
                                        * indicates order in the buddy
                                        * system if PG_buddy is set.
                                        */
                struct address_space *mapping;  /* If low bit clear, points to
                                        * inode address_space, or NULL.
                                        * If page mapped as anonymous
                                        * memory, low bit is set, and
                                        * it points to anon_vma object:
                                        * see PAGE_MAPPING_ANON below.
                                        */
                };
#if USE_SPLIT_PTLOCKS
                spinlock_t ptl;
#endif
                struct kmem_cache *slab;        /* SLUB: Pointer to slab */
                struct page *first_page;        /* Compound tail pages */
        };
```

```
	union {
		pgoff_t index;		/* Our offset within mapping. */
		void *freelist;		/* SLUB: freelist req. slab lock */
	};
	struct list_head lru;		/* Pageout list, eg. active_list
					 * protected by zone->lru_lock !
					 */
	/*
	 * On machines where all RAM is mapped into kernel address space,
	 * we can simply calculate the virtual address. On machines with
	 * highmem some memory is mapped into kernel virtual memory
	 * dynamically, so we need a place to store that address.
	 * Note that this field could be 16 bits on x86 ... ;)
	 *
	 * Architectures with slow multiplication can define
	 * WANT_PAGE_VIRTUAL in asm/page.h
	 */
#if defined(WANT_PAGE_VIRTUAL)
	void *virtual;			/* Kernel virtual address (NULL if
					   not kmapped, ie. highmem) */
#endif /* WANT_PAGE_VIRTUAL */
#ifdef CONFIG_WANT_PAGE_DEBUG_FLAGS
	unsigned long debug_flags;	/* Use atomic bitops on this */
#endif

#ifdef CONFIG_KMEMCHECK
	/*
	 * kmemcheck wants to track the status of each byte in a page; this
	 * is a pointer to such a status block. NULL if not tracked.
	 */
	void *shadow;
```

```
#endif
};
```

- mapping 指定了页帧所在的地址空间 address_space。
- index 字段表示在所有者的地址空间中以页大小为单位的偏移，也就是所有者的磁盘映射中页中数据的位置。
- virtual 指向本页在内核中的虚拟地址

在页高速缓存中查找页使用 mapping、index 这两个字段。

5.4.2.3 address_space

页高速缓冲的核心数据结构是 address_space，它嵌入在页所有者的索引节点对象的数据结构。

```
struct address_space {
        struct inode            *host;          /* owner: inode, block_device */
        struct radix_tree_root  page_tree;      /* radix tree of all pages */
        spinlock_t              tree_lock;      /* and lock protecting it */
        unsigned int            i_mmap_writable;/* count VM_SHARED mappings */
        struct prio_tree_root   i_mmap;         /* tree of private and shared mappings */
        struct list_head        i_mmap_nonlinear;/*list VM_NONLINEAR mappings */
        spinlock_t              i_mmap_lock;    /* protect tree, count, list */
        unsigned int            truncate_count; /* Cover race condition with truncate */
        unsigned long           nrpages;        /* number of total pages */
        pgoff_t                 writeback_index;/* writeback starts here */
        const struct address_space_operations *a_ops;   /* methods */
        unsigned long           flags;          /* error bits/gfp mask */
        struct backing_dev_info *backing_dev_info; /* device readahead, etc */
        spinlock_t              private_lock;   /* for use by the address_space */
```

```
        struct list_head        private_list;     /* ditto */
        struct address_space        *assoc_mapping; /* ditto */
} __attribute__((aligned(sizeof(long))));
```

- host：指向拥有该对象的索引节点的指针（如果存在）
- page_tree：拥有者页的基数的根。
- i_mmap_writable：地址空间中共享内存映射的个数
- i_mmap：radix 优先搜索树的根

5.4.2.4 buffer_head

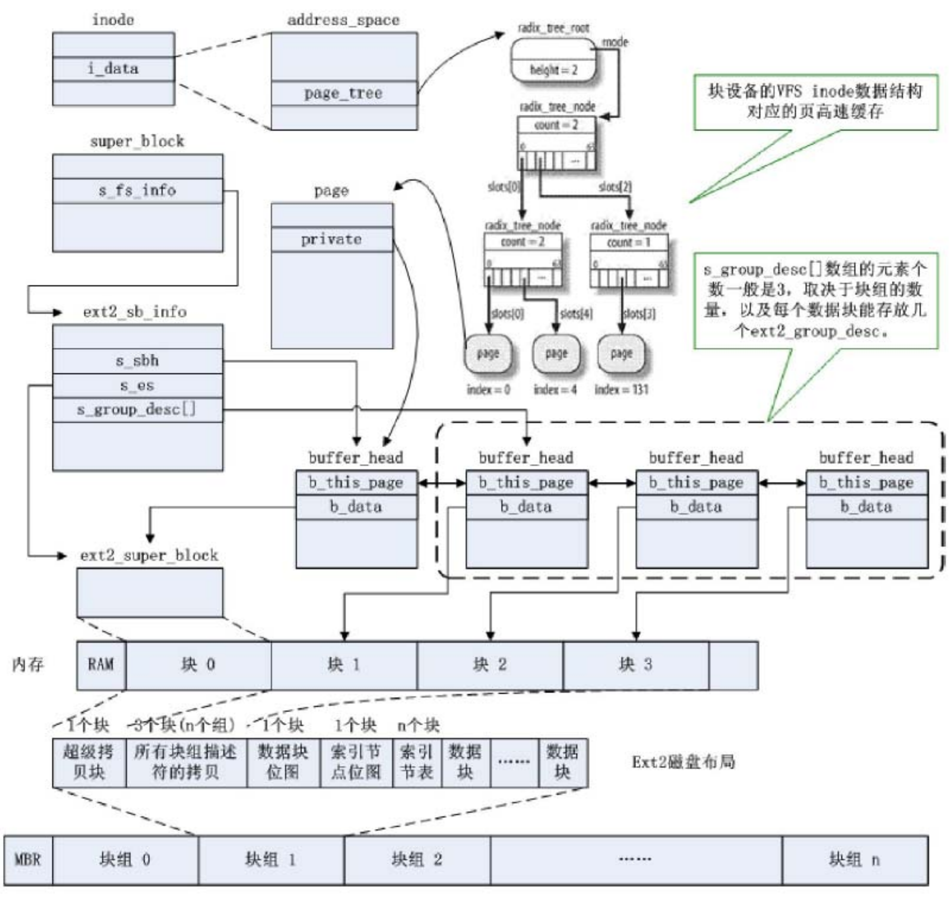
```
/*
 * Historically, a buffer_head was used to map a single block
 * within a page, and of course as the unit of I/O through the
 * filesystem and block layers.   Nowadays the basic I/O unit
 * is the bio, and buffer_heads are used for extracting block
 * mappings (via a get_block_t call), for tracking state within
 * a page (via a page_mapping) and for wrapping bio submission
 * for backward compatibility reasons (e.g. submit_bh).
 */
struct buffer_head {
        unsigned long b_state;          /* buffer state bitmap (see above) */
        struct buffer_head *b_this_page;/* circular list of page's buffers */
        struct page *b_page;            /* the page this bh is mapped to */

        sector_t b_blocknr;         /* start block number */
        size_t b_size;                  /* size of mapping */
        char *b_data;                   /* pointer to data within the page */

        struct block_device *b_bdev;
        bh_end_io_t *b_end_io;          /* I/O completion */
        void *b_private;                /* reserved for b_end_io */
        struct list_head b_assoc_buffers; /* associated with another mapping */
```

```
        struct address_space *b_assoc_map;    /* mapping this buffer is
                                                 associated with */
        atomic_t b_count;            /* users using this buffer_head */
};
```

- b_state：缓冲区状态标志
- b_this_page：指向缓冲区页的链表中的下一个元素的指针
- b_page：指向拥有该块的缓冲区页的描述符指针
- b_blocknr：与块设备相关的块号（起始逻辑块号）
- b_size：块大小
- b_data：块在缓冲区页内的位置
- b_bdev：指向块设备描述符的指针
- b_end_io：I/O 完成方法
- b_private：指向 I/O 完成方法的数据的指针
- b_assoc_buffers：为与某个索引节点相关的间接块的链表提供的指针
- b_count：块使用计数器。



各种数据结构关联图

5.4.3 文件的读sys_read

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
        struct file *file;
        ssize_t ret = -EBADF;
        int fput_needed;

        file = fget_light(fd, &fput_needed);
        if (file) {
                loff_t pos = file_pos_read(file);
                ret = vfs_read(file, buf, count, &pos);
                file_pos_write(file, pos);
                fput_light(file, fput_needed);
        }

        return ret;
}
```

5.4.3.1 vfs_read

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
        ssize_t ret;

        if (!(file->f_mode & FMODE_READ))
                return -EBADF;
        if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
                return -EINVAL;
        if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
                return -EFAULT;
```

```

	ret = rw_verify_area(READ, file, pos, count);
	if (ret >= 0) {
		count = ret;
		if (file->f_op->read)
			ret = file->f_op->read(file, buf, count, pos);
		else
			ret = do_sync_read(file, buf, count, pos);
		if (ret > 0) {
			fsnotify_access(file->f_path.dentry);
			add_rchar(current, ret);
		}
		inc_syscr(current);
	}

	return ret;
}
```

5.4.3.2 do_sync_read

```
ssize_t do_sync_read(struct file *filp, char __user *buf, size_t len, loff_t *ppos)
{
	struct iovec iov = { .iov_base = buf, .iov_len = len };
	struct kiocb kiocb;
	ssize_t ret;

	init_sync_kiocb(&kiocb, filp);
	kiocb.ki_pos = *ppos;
	kiocb.ki_left = len;
	kiocb.ki_nbytes = len;

```

```
	for (;;) {
		ret = filp->f_op->aio_read(&kiocb, &iov, 1, kiocb.ki_pos);
		if (ret != -EIOCBRETRY)
			break;
		wait_on_retry_sync_kiocb(&kiocb);
	}

	if (-EIOCBQUEUED == ret)
		ret = wait_on_sync_kiocb(&kiocb);
	*ppos = kiocb.ki_pos;
	return ret;
}
```

5.4.3.3 generic_file_aio_read

该函数是所有文件系统实现同步和异步读取操作所使用的通用例程。

```
ssize_t
generic_file_aio_read(struct kiocb *iocb, const struct iovec *iov,
		unsigned long nr_segs, loff_t pos)
{
	struct file *filp = iocb->ki_filp;
	ssize_t retval;
	unsigned long seg;
	size_t count;
	loff_t *ppos = &iocb->ki_pos;

	count = 0;
	retval = generic_segment_checks(iov, &nr_segs, &count, VERIFY_WRITE);
	if (retval)
		return retval;

```

```
/* coalesce the iovecs and go direct-to-BIO for O_DIRECT */
if (filp->f_flags & O_DIRECT) {
	loff_t size;
	struct address_space *mapping;
	struct inode *inode;

	mapping = filp->f_mapping;
	inode = mapping->host;
	if (!count)
		goto out; /* skip atime */
	size = i_size_read(inode);
	if (pos < size) {
		retval = filemap_write_and_wait_range(mapping, pos,
				pos + iov_length(iov, nr_segs) - 1);
		if (!retval) {
			retval = mapping->a_ops->direct_IO(READ, iocb,
						iov, pos, nr_segs);
		}
		if (retval > 0)
			*ppos = pos + retval;
		if (retval) {
			file_accessed(filp);
			goto out;
		}
	}
}

for (seg = 0; seg < nr_segs; seg++) {
	read_descriptor_t desc;

	desc.written = 0;
```

```
		desc.arg.buf = iov[seg].iov_base;
		desc.count = iov[seg].iov_len;
		if (desc.count == 0)
			continue;
		desc.error = 0;
		do_generic_file_read(filp, ppos, &desc, file_read_actor);
		retval += desc.written;
		if (desc.error) {
			retval = retval ?: desc.error;
			break;
		}
		if (desc.count > 0)
			break;
	}
```

for 循环中完成文件的读取操作。

```
out:
	return retval;
}
EXPORT_SYMBOL(generic_file_aio_read);
```

5.4.3.4 do_generic_file_read

```
static void do_generic_file_read(struct file *filp, loff_t *ppos,
		read_descriptor_t *desc, read_actor_t actor)
{
	struct address_space *mapping = filp->f_mapping;
```

地址空间

```
	struct inode *inode = mapping->host;
	struct file_ra_state *ra = &filp->f_ra;
	pgoff_t index;
	pgoff_t last_index;
	pgoff_t prev_index;
```

```
	unsigned long offset;		/* offset into pagecache page */
	unsigned int prev_offset;
	int error;

	index = *ppos >> PAGE_CACHE_SHIFT;
```

ppos 在缓存中对于的页面的索引号。

```
	prev_index = ra->prev_pos >> PAGE_CACHE_SHIFT;
	prev_offset = ra->prev_pos & (PAGE_CACHE_SIZE-1);
	last_index = (*ppos + desc->count + PAGE_CACHE_SIZE-1) >> PAGE_CACHE_SHIFT;
	offset = *ppos & ~PAGE_CACHE_MASK;

	for (;;) {
		struct page *page;
		pgoff_t end_index;
		loff_t isize;
		unsigned long nr, ret;

		cond_resched();
find_page:
		page = find_get_page(mapping, index);
```

在地址空间中查找对应的页。

```
		if (!page) {
```

如果页不在缓存中，则调用 page_cache_sync_readahead 预读函数。

```
			page_cache_sync_readahead(mapping,
					ra, filp,
					index, last_index - index);
```

从磁盘中读取相应内容。

```
			page = find_get_page(mapping, index);
			if (unlikely(page == NULL))
				goto no_cached_page;
```

```
		}
```

预读之后再次查询，如果还是没有找到，跳转到 no_cached_page 处。

```
		if (PageReadahead(page)) {
			page_cache_async_readahead(mapping,
					ra, filp, page,
					index, last_index - index);
		}
```

需要预读，则调用 page_cache_async_readhead 函数预读。

```
		if (!PageUptodate(page)) {
```

如果 page 已经存在于缓存中，但是 PG_uptodata 没有置为，说明数据没有更新，需要重磁盘读入。

```
			if (inode->i_blkbits == PAGE_CACHE_SHIFT ||
					!mapping->a_ops->is_partially_uptodate)
				goto page_not_up_to_date;
			if (!trylock_page(page))
				goto page_not_up_to_date;
```

跳转到 page_not_up_to_data。

```
			if (!mapping->a_ops->is_partially_uptodate(page,
								desc, offset))
				goto page_not_up_to_date_locked;
			unlock_page(page);
		}
page_ok:
		/*
		 * i_size must be checked after we know the page is Uptodate.
		 *
		 * Checking i_size after the check allows us to calculate
		 * the correct value for "nr", which means the zero-filled
		 * part of the page is not copied back to userspace (unless
		 * another truncate extends the file - this is desired though).
		 */
```

```
	isize = i_size_read(inode);
	end_index = (isize - 1) >> PAGE_CACHE_SHIFT;
	if (unlikely(!isize || index > end_index)) {
		page_cache_release(page);
		goto out;
	}

	/* nr is the maximum number of bytes to copy from this page */
	nr = PAGE_CACHE_SIZE;
	if (index == end_index) {
		nr = ((isize - 1) & ~PAGE_CACHE_MASK) + 1;
		if (nr <= offset) {
			page_cache_release(page);
			goto out;
		}
	}
	nr = nr - offset;

	/* If users can be writing to this page using arbitrary
	 * virtual addresses, take care about potential aliasing
	 * before reading the page on the kernel side.
	 */
	if (mapping_writably_mapped(mapping))
		flush_dcache_page(page);

	/*
	 * When a sequential read accesses a page several times,
	 * only mark it as accessed the first time.
	 */
	if (prev_index != index || offset != prev_offset)
```

```
1057                mark_page_accessed(page);
1058            prev_index = index;
1059
1060            /*
1061             * Ok, we have the page, and it's up-to-date, so
1062             * now we can copy it to user space...
1063             *
1064             * The actor routine returns how many bytes were actually used..
1065             * NOTE! This may not be the same as how much of a user buffer
1066             * we filled up (we may be padding etc), so we can only update
1067             * "pos" here (the actor routine has to update the user buffer
1068             * pointers and the remaining count).
1069             */
1070            ret = actor(desc, page, offset, nr);
1071            offset += ret;
1072            index += offset >> PAGE_CACHE_SHIFT;
1073            offset &= ~PAGE_CACHE_MASK;
1074            prev_offset = offset;
1075
1076            page_cache_release(page);
1077            if (ret == nr && desc->count)
1078                continue;
```

1018-1078：当要读取的内容已经存在与缓存中时，指向这段代码，代码比较简单，通过传过来的 actor 参数，将缓存中的内容读入用户传递过来的缓存区中。

```
1079            goto out;
1080
1081 page_not_up_to_date:
1082            /* Get exclusive access to the page ... */
1083            error = lock_page_killable(page);
1084            if (unlikely(error))
1085                goto readpage_error;
```

```

page_not_up_to_date_locked:
		/* Did it get truncated before we got the lock? */
		if (!page->mapping) {
			unlock_page(page);
			page_cache_release(page);
			continue;
		}

		/* Did somebody else fill it already? */
		if (PageUptodate(page)) {
			unlock_page(page);
			goto page_ok;
		}

readpage:
		/* Start the actual read. The read will unlock the page. */
		error = mapping->a_ops->readpage(filp, page);
```

调用具体文件系统的函数从磁盘中读取内容到缓存。

```

		if (unlikely(error)) {
			if (error == AOP_TRUNCATED_PAGE) {
				page_cache_release(page);
				goto find_page;
			}
			goto readpage_error;
		}

if (!PageUptodate(page)) {
			error = lock_page_killable(page);
			if (unlikely(error))
```

```
                        goto readpage_error;
                if (!PageUptodate(page)) {
                        if (page->mapping == NULL) {
                                /*
                                 * invalidate_mapping_pages got it
                                 */
                                unlock_page(page);
                                page_cache_release(page);
                                goto find_page;
                        }
                        unlock_page(page);
                        shrink_readahead_size_eio(filp, ra);
                        error = -EIO;
                        goto readpage_error;
                }
                unlock_page(page);
        }

        goto page_ok;

readpage_error:
        /* UHHUH! A synchronous read error occurred. Report it */
        desc->error = error;
        page_cache_release(page);
        goto out;

no_cached_page:
        /*
         * Ok, it wasn't cached, so we need to create a new
         * page..
         */
```

```
		page = page_cache_alloc_cold(mapping);
```

分配一个页面

```
		if (!page) {
			desc->error = -ENOMEM;
			goto out;
		}
		error = add_to_page_cache_lru(page, mapping,
						index, GFP_KERNEL);
```

将页面添加到对应的 address_space 中。

```
		if (error) {
			page_cache_release(page);
			if (error == -EEXIST)
				goto find_page;
			desc->error = error;
			goto out;
		}
		goto readpage;
```

跳转到 readpage 读取页面。

```
	}

out:
	ra->prev_pos = prev_index;
	ra->prev_pos <<= PAGE_CACHE_SHIFT;
	ra->prev_pos |= prev_offset;

	*ppos = ((loff_t)index << PAGE_CACHE_SHIFT) + offset;
	file_accessed(filp);
}
```

5.4.3.5 ext2_readpage

```
static int ext2_readpage(struct file *file, struct page *page)
```

```
{
        return mpage_readpage(page, ext2_get_block);
```

调用 mpage_readpage 完成进一步操作，参数 ext2_get_block 函数获得硬盘上指定的块，该函数之前已经分析过。

```
}
```

5.4.3.6 mpage_readpage

```
/*
 * This isn't called much at all
 */
int mpage_readpage(struct page *page, get_block_t get_block)
{
        struct bio *bio = NULL;
        sector_t last_block_in_bio = 0;
        struct buffer_head map_bh;
        unsigned long first_logical_block = 0;

        map_bh.b_state = 0;
        map_bh.b_size = 0;
        bio = do_mpage_readpage(bio, page, 1, &last_block_in_bio,
                        &map_bh, &first_logical_block, get_block);
        if (bio)
                mpage_bio_submit(READ, bio);
        return 0;
}
EXPORT_SYMBOL(mpage_readpage);
```

初始化 buffer_head map_bh 相关字段，调用 do_mpage_readpage 函数创建一个 bio 请求，该请求指明了要读取数据库在磁盘的位置、数据库的数量以及拷贝该数据的目标位置——缓存区中的 page 信息，然后调用 mpage_bio_submit 函数处理请求。

5.4.3.7 do_mpage_readpage

```
struct bio {
sector_t bi_sector; //块 I/O 操作的第一个磁盘扇区
struct bio *bi_next; //链接到请求队列中的下一个 bio
struct block_device *bi_bdev;//指向块设备描述符的指针
unsigned long bi_flags; //bio 的状态标志
unsigned long bi_rw; //IO 操作标志，即这次 I.O 是读或写
unsigned short bi_vcnt; /* bio 的 bio_vec 数组中段的数目 */
unsigned short bi_idx; /* bio 的 bio_vec 数组中段的当前索引值 */
unsigned short bi_phys_segments; //合并之后 bio 中物理段的数目
unsigned short bi_hw_segments; //合并之后硬件段的数目
unsigned int bi_size; /* 需要传送的字节数 */
unsigned int bi_hw_front_size;// 硬件段合并算法使用
unsigned int bi_hw_back_size;// 硬件段合并算法使用
unsigned int bi_max_vecs; /* bio 的 bio vec 数组中允许的最大段数 */
struct bio_vec *bi_io_vec; /*指向 bio 的 bio_vec 数组中的段的指针 */
bio_end_io_t *bi_end_io; /* bio 的 I/O 操作结束时调用的方法 */
atomic_t bi_cnt; /* bio 的引用计数器 */
void *bi_private; //通用块层和块设备驱动程序的 I/O 完成方法使用的指针
bio_destructor_t *bi_destructor;//释放 bio 时调用的析构方法（通常是
bio_destructor()方法）
};
```

```
static struct bio *
do_mpage_readpage(struct bio *bio, struct page *page, unsigned nr_pages,
                sector_t *last_block_in_bio, struct buffer_head *map_bh,
                unsigned long *first_logical_block, get_block_t get_block)
{
        struct inode *inode = page->mapping->host;
        const unsigned blkbits = inode->i_blkbits;
        const unsigned blocks_per_page = PAGE_CACHE_SIZE >> blkbits;
```

计算每页高速缓存能存放的块数目。

```
176 const unsigned blocksize = 1 << blkbits;
177 sector_t block_in_file;
178 sector_t last_block;
179 sector_t last_block_in_file;
180 sector_t blocks[MAX_BUF_PER_PAGE];
181 unsigned page_block;
182 unsigned first_hole = blocks_per_page;
183 struct block_device *bdev = NULL;
184 int length;
185 int fully_mapped = 1;
186 unsigned nblocks;
187 unsigned relative_block;
188
189 if (page_has_buffers(page))
190         goto confused;
191
192 block_in_file = (sector_t)page->index << (PAGE_CACHE_SHIFT - blkbits);
193 last_block = block_in_file + nr_pages * blocks_per_page;
194 last_block_in_file = (i_size_read(inode) + blocksize - 1) >> blkbits;
195 if (last_block > last_block_in_file)
196         last_block = last_block_in_file;
```

缓存是以页为单位的，而记录块是以块为单位的，192 行记录 page 在文件内部的块号（也就是页面号乘以 4），last_block 记录所要读取内容的最后一个块号。如果 last_block 超过了文件系统的最大块号，则 last_block 赋值为文件系统最大块号。

```
197 page_block = 0;
199 /*
200   * Map blocks using the result from the previous get_blocks call first.
201   */
202 nblocks = map_bh->b_size >> blkbits;
```

```
	if (buffer_mapped(map_bh) && block_in_file > *first_logical_block &&
			block_in_file < (*first_logical_block + nblocks)) {
```

如果页面已经被映射，且页中第一块的文件块号位于传递进来的 first_logical_block 参数和一个 buffer_heand 最后一个块之间，则对未映射部分进行处理。

```
		unsigned map_offset = block_in_file - *first_logical_block;
		unsigned last = nblocks - map_offset;

		for (relative_block = 0; ; relative_block++) {
			if (relative_block == last) {
				clear_buffer_mapped(map_bh);
				break;
			}
			if (page_block == blocks_per_page)
				break;
			blocks[page_block] = map_bh->b_blocknr + map_offset +
						relative_block;
			page_block++;
			block_in_file++;
		}
		bdev = map_bh->b_bdev;
	}

	/*
	 * Then do more get_blocks calls until we are done with this page.
	 */
	map_bh->b_page = page;
	while (page_block < blocks_per_page) {
		map_bh->b_state = 0;
		map_bh->b_size = 0;

		if (block_in_file < last_block) {
```

```
        map_bh->b_size = (last_block-block_in_file) << blkbits;
        if (get_block(inode, block_in_file, map_bh, 0))
```

调用 get_block 将在文件中的逻辑块号转换为磁盘中的逻辑块号，保存在 buffer_head 中的 b_blocknr 字段中。

```
            goto confused;
        *first_logical_block = block_in_file;
    }

    if (!buffer_mapped(map_bh)) {
        if (first_hole == blocks_per_page)
            first_hole = page_block;
        page_block++;
        block_in_file++;
```

如果 buffer_mapped 没被映射，page_block++，block_in_file++，跳转到循环开头处继续。

```
        continue;
    }

    /* some filesystems will copy data into the page during
     * the get_block call, in which case we don't want to
     * read it again.   map_buffer_to_page copies the data
     * we just collected from get_block into the page's buffers
     * so readpage doesn't have to repeat the get_block call
     */
    if (buffer_uptodate(map_bh)) {
        map_buffer_to_page(page, map_bh, page_block);
        goto confused;
    }

    if (first_hole != blocks_per_page)
        goto confused;          /* hole -> non-hole */
```

```

		/* Contiguous blocks? */
		if (page_block && blocks[page_block-1] != map_bh->b_blocknr-1)
			goto confused;
		nblocks = map_bh->b_size >> blkbits;
		for (relative_block = 0; ; relative_block++) {
			if (relative_block == nblocks) {
				clear_buffer_mapped(map_bh);
				break;
			} else if (page_block == blocks_per_page)
				break;
			blocks[page_block] = map_bh->b_blocknr+relative_block;
			page_block++;
			block_in_file++;
		}
		bdev = map_bh->b_bdev;
	}

	if (first_hole != blocks_per_page) {
		zero_user_segment(page, first_hole << blkbits, PAGE_CACHE_SIZE);
		if (first_hole == 0) {
			SetPageUptodate(page);
			unlock_page(page);
			goto out;
		}
	} else if (fully_mapped) {
		SetPageMappedToDisk(page);
	}

	/*
	 * This page will go to BIO.    Do we need to send this BIO off first?
```

```
	 */
	if (bio && (*last_block_in_bio != blocks[0] - 1))
		bio = mpage_bio_submit(READ, bio);

alloc_new:
	if (bio == NULL) {
		bio = mpage_alloc(bdev, blocks[0] << (blkbits - 9),
				min_t(int, nr_pages, bio_get_nr_vecs(bdev)),
				GFP_KERNEL);
		if (bio == NULL)
			goto confused;
	}

	length = first_hole << blkbits;
	if (bio_add_page(bio, page, length, 0) < length) {
		bio = mpage_bio_submit(READ, bio);
		goto alloc_new;
	}
```

分配一个 BIO，进行适当的初始化。

```

	relative_block = block_in_file - *first_logical_block;
	nblocks = map_bh->b_size >> blkbits;
	if ((buffer_boundary(map_bh) && relative_block == nblocks) ||
		(first_hole != blocks_per_page))
		bio = mpage_bio_submit(READ, bio);
```

调用 mpage_bio_submit 处理读请求。完成读文件的实际操作。

```
	else
		*last_block_in_bio = blocks[blocks_per_page - 1];
out:
	return bio;

```

```
confused:
	if (bio)
		bio = mpage_bio_submit(READ, bio);
	if (!PageUptodate(page))
	        block_read_full_page(page, get_block);
	else
		unlock_page(page);
	goto out;
}
```

5.4.4 sys_write 文件的写

fs/read_write.c

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
		size_t, count)
```

参数：

- fd：用户空间的文件句柄号
- buf：用户空间保存写入内容的缓冲区指针
- count：写入的内容的长度，以字节为单位

```
{
	struct file *file;
	ssize_t ret = -EBADF;
	int fput_needed;

	file = fget_light(fd, &fput_needed);
```

根据文件句柄号得到 file 结构。

```
	if (file) {
		loff_t pos = file_pos_read(file);
```

得到文件的上下文 pos，pos 即表示文件当前索引与文件起始处的偏移量。

```
		ret = vfs_write(file, buf, count, &pos);
```

读文件。

```
        file_pos_write(file, pos);
```

设置文件的上下文。

```
        fput_light(file, fput_needed);
    }

    return ret;
}
```

5.4.4.1 vfs_write

```
ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;

    if (!(file->f_mode & FMODE_WRITE))
        return -EBADF;
    if (!file->f_op || (!file->f_op->write && !file->f_op->aio_write))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_READ, buf, count)))
        return -EFAULT;
```

写操作必要条件检测

```

    ret = rw_verify_area(WRITE, file, pos, count);
```

检测需要写入的内容是否被加锁

```
    if (ret >= 0) {
        count = ret;
        if (file->f_op->write)
            ret = file->f_op->write(file, buf, count, pos);
```

调用具体文件系统的写入函数进行写操作。对于 ext2 而言就是 do_sync_write 函数。

```
                else
                        ret = do_sync_write(file, buf, count, pos);
                if (ret > 0) {
                        fsnotify_modify(file->f_path.dentry);
                        add_wchar(current, ret);
                }
                inc_syscw(current);
        }

        return ret;
}
```

5.4.4.2 do_sync_write

```
ssize_t do_sync_write(struct file *filp, const char __user *buf, size_t len, loff_t *ppos)
{
        struct iovec iov = { .iov_base = (void __user *)buf, .iov_len = len };
        struct kiocb kiocb;
        ssize_t ret;

        init_sync_kiocb(&kiocb, filp);
        kiocb.ki_pos = *ppos;
        kiocb.ki_left = len;
        kiocb.ki_nbytes = len;

        for (;;) {
                ret = filp->f_op->aio_write(&kiocb, &iov, 1, kiocb.ki_pos);
```

调用该函数完成写入操作。

```
		if (ret != -EIOCBRETRY)
			break;
		wait_on_retry_sync_kiocb(&kiocb);
	}

	if (-EIOCBQUEUED == ret)
		ret = wait_on_sync_kiocb(&kiocb);
	*ppos = kiocb.ki_pos;
	return ret;
}

EXPORT_SYMBOL(do_sync_write);
```

5.4.4.3 generic_file_aio_write

mm/filemap.c

```
ssize_t generic_file_aio_write(struct kiocb *iocb, const struct iovec *iov,
		unsigned long nr_segs, loff_t pos)
{
	struct file *file = iocb->ki_filp;
	struct inode *inode = file->f_mapping->host;
	ssize_t ret;

	BUG_ON(iocb->ki_pos != pos);

	mutex_lock(&inode->i_mutex);
	ret = __generic_file_aio_write(iocb, iov, nr_segs, &iocb->ki_pos);
	mutex_unlock(&inode->i_mutex);

	if (ret > 0 || ret == -EIOCBQUEUED) {
		ssize_t err;
```

```

                err = generic_write_sync(file, pos, ret);
                if (err < 0 && ret > 0)
                        ret = err;
        }
        return ret;
}
EXPORT_SYMBOL(generic_file_aio_write);
```

5.4.4.4 __generic_file_aio_write

```
ssize_t __generic_file_aio_write(struct kiocb *iocb, const struct iovec *iov,
                                 unsigned long nr_segs, loff_t *ppos)
{
        struct file *file = iocb->ki_filp;
        struct address_space * mapping = file->f_mapping;
        size_t ocount;          /* original count */
        size_t count;           /* after file limit checks */
        struct inode    *inode = mapping->host;
        loff_t          pos;
        ssize_t         written;
        ssize_t         err;

        ocount = 0;
        err = generic_segment_checks(iov, &nr_segs, &ocount, VERIFY_READ);
        if (err)
                return err;

        count = ocount;
        pos = *ppos;
```

调用该函数进行写入操作。

```

	vfs_check_frozen(inode->i_sb, SB_FREEZE_WRITE);

	/* We can write back this queue in page reclaim */
	current->backing_dev_info = mapping->backing_dev_info;
	written = 0;

	err = generic_write_checks(file, &pos, &count, S_ISBLK(inode->i_mode));
	if (err)
		goto out;

	if (count == 0)
		goto out;

	err = file_remove_suid(file);
	if (err)
		goto out;

	file_update_time(file);

	/* coalesce the iovecs and go direct-to-BIO for O_DIRECT */
	if (unlikely(file->f_flags & O_DIRECT)) {
		loff_t endbyte;
		ssize_t written_buffered;

		written = generic_file_direct_write(iocb, iov, &nr_segs, pos,
							ppos, count, ocount);
```

调用该函数进行写入操作。

```
		if (written < 0 || written == count)
			goto out;
```

```
	/*
	 * direct-io write to a hole: fall through to buffered I/O
	 * for completing the rest of the request.
	 */
	pos += written;
	count -= written;
	written_buffered = generic_file_buffered_write(iocb, iov,
					nr_segs, pos, ppos, count,
					written);
	/*
	 * If generic_file_buffered_write() retuned a synchronous error
	 * then we want to return the number of bytes which were
	 * direct-written, or the error code if that was zero.  Note
	 * that this differs from normal direct-io semantics, which
	 * will return -EFOO even if some bytes were written.
	 */
	if (written_buffered < 0) {
		err = written_buffered;
		goto out;
	}

	/*
	 * We need to ensure that the page cache pages are written to
	 * disk and invalidated to preserve the expected O_DIRECT
	 * semantics.
	 */
	endbyte = pos + written_buffered - written - 1;
	err = filemap_write_and_wait_range(file->f_mapping, pos, endbyte);
	if (err == 0) {
		written = written_buffered;
		invalidate_mapping_pages(mapping,
```

```
                                        pos >> PAGE_CACHE_SHIFT,
                                        endbyte >> PAGE_CACHE_SHIFT);
                } else {
                        /*
                         * We don't know how much we wrote, so just return
                         * the number of bytes which were direct-written
                         */
                }
        } else {
                written = generic_file_buffered_write(iocb, iov, nr_segs,
                                pos, ppos, count, written);
        }
out:
        current->backing_dev_info = NULL;
        return written ? written : err;
}
EXPORT_SYMBOL(__generic_file_aio_write);
```

5.4.4.5 generic_file_direct_write

```
ssize_t
generic_file_direct_write(struct kiocb *iocb, const struct iovec *iov,
                unsigned long *nr_segs, loff_t pos, loff_t *ppos,
                size_t count, size_t ocount)
{
        struct file *file = iocb->ki_filp;
        struct address_space *mapping = file->f_mapping;
        struct inode    *inode = mapping->host;
        ssize_t         written;
        size_t          write_len;
        pgoff_t         end;
```

```
	if (count != ocount)
		*nr_segs = iov_shorten((struct iovec *)iov, *nr_segs, count);

	write_len = iov_length(iov, *nr_segs);
	end = (pos + write_len - 1) >> PAGE_CACHE_SHIFT;

	written = filemap_write_and_wait_range(mapping, pos, pos + write_len - 1);
	if (written)
		goto out;

	/*
	 * After a write we want buffered reads to be sure to go to disk to get
	 * the new data.  We invalidate clean cached page from the region we're
	 * about to write.  We do this *before* the write so that we can return
	 * without clobbering -EIOCBQUEUED from ->direct_IO().
	 */
	if (mapping->nrpages) {
		written = invalidate_inode_pages2_range(mapping,
					pos >> PAGE_CACHE_SHIFT, end);
		/*
		 * If a page can not be invalidated, return 0 to fall back
		 * to buffered write.
		 */
		if (written) {
			if (written == -EBUSY)
				return 0;
			goto out;
		}
	}
```

```

	written = mapping->a_ops->direct_IO(WRITE, iocb, iov, pos, *nr_segs);

	/*
	 * Finally, try again to invalidate clean pages which might have been
	 * cached by non-direct readahead, or faulted in by get_user_pages()
	 * if the source of the write was an mmap'ed region of the file
	 * we're writing.  Either one is a pretty crazy thing to do,
	 * so we don't support it 100%.  If this invalidation
	 * fails, tough, the write still worked...
	 */
	if (mapping->nrpages) {
		invalidate_inode_pages2_range(mapping,
					      pos >> PAGE_CACHE_SHIFT, end);
	}

	if (written > 0) {
		loff_t end = pos + written;
		if (end > i_size_read(inode) && !S_ISBLK(inode->i_mode)) {
			i_size_write(inode,  end);
			mark_inode_dirty(inode);
		}
		*ppos = end;
	}
out:
	return written;
}
EXPORT_SYMBOL(generic_file_direct_write);
```

5.4.4.6 filemap_write_and_wait_range

```
int filemap_write_and_wait_range(struct address_space *mapping,
                                 loff_t lstart, loff_t lend)
{
        int err = 0;

        if (mapping->nrpages) {
                err = __filemap_fdatawrite_range(mapping, lstart, lend,
                                                 WB_SYNC_ALL);
                /* See comment of filemap_write_and_wait() */
                if (err != -EIO) {
                        int err2 = filemap_fdatawait_range(mapping,
                                                lstart, lend);
                        if (!err)
                                err = err2;
                }
        }
        return err;
}
EXPORT_SYMBOL(filemap_write_and_wait_range);
```

5.4.4.7 __filemap_fdatawrite_range

```
int __filemap_fdatawrite_range(struct address_space *mapping, loff_t start,
                               loff_t end, int sync_mode)
{
        int ret;
        struct writeback_control wbc = {
                .sync_mode = sync_mode,
                .nr_to_write = LONG_MAX,
```

```
        .range_start = start,
        .range_end = end,
    };
```

分配一个 writeback_control 结构.

```

    if (!mapping_cap_writeback_dirty(mapping))
        return 0;

    ret = do_writepages(mapping, &wbc);
```

调用 do_writepages。

```
    return ret;
}
```

5.4.4.8 mm/page-writeback.c

```
int do_writepages(struct address_space *mapping, struct writeback_control *wbc)
{
    int ret;

    if (wbc->nr_to_write <= 0)
        return 0;
    if (mapping->a_ops->writepages)
        ret = mapping->a_ops->writepages(mapping, wbc);
```

调用 ext2_writepages 函数。

```
    else
        ret = generic_writepages(mapping, wbc);
    return ret;
}
```

5.4.4.9 ext2_writepages

```
static int
ext2_writepages(struct address_space *mapping, struct writeback_control *wbc)
{
        return mpage_writepages(mapping, wbc, ext2_get_block);
```

传递 ext2_get_block 参数。

```
}
```

5.4.4.10 mpage_writepages

```
int
mpage_writepages(struct address_space *mapping,
                struct writeback_control *wbc, get_block_t get_block)
{
        int ret;

        if (!get_block)
                ret = generic_writepages(mapping, wbc);
        else {
                struct mpage_data mpd = {
                        .bio = NULL,
                        .last_block_in_bio = 0,
                        .get_block = get_block,
                        .use_writepage = 1,
                };
```

初始化一个 mpage_data 结构

```

                ret = write_cache_pages(mapping, wbc, __mpage_writepage, &mpd);
```

传入产生__mpage_writepage 调用 write_cache_pages 函数。

```
                if (mpd.bio)
                        mpage_bio_submit(WRITE, mpd.bio);
```

bio 请求准备好后，则调用 mpage_bio_submit 进行写入操作。

```
        }
        return ret;
}
EXPORT_SYMBOL(mpage_writepages);
```

5.4.4.11 write_cache_pages

```
int write_cache_pages(struct address_space *mapping,
                      struct writeback_control *wbc, writepage_t writepage,
                      void *data)
{
        int ret = 0;
        int done = 0;
        struct pagevec pvec;
        int nr_pages;
        pgoff_t uninitialized_var(writeback_index);
        pgoff_t index;
        pgoff_t end;            /* Inclusive */
        pgoff_t done_index;
        int cycled;
        int range_whole = 0;
        long nr_to_write = wbc->nr_to_write;

        pagevec_init(&pvec, 0);
        if (wbc->range_cyclic) {
                writeback_index = mapping->writeback_index; /* prev offset */
                index = writeback_index;
                if (index == 0)
```

```
			cycled = 1;
		else
			cycled = 0;
		end = -1;
	} else {
		index = wbc->range_start >> PAGE_CACHE_SHIFT;
		end = wbc->range_end >> PAGE_CACHE_SHIFT;
		if (wbc->range_start == 0 && wbc->range_end == LLONG_MAX)
			range_whole = 1;
		cycled = 1; /* ignore range_cyclic tests */
	}
retry:
	done_index = index;
	while (!done && (index <= end)) {
		int i;

		nr_pages = pagevec_lookup_tag(&pvec, mapping, &index,
					PAGECACHE_TAG_DIRTY,
					min(end - index, (pgoff_t)PAGEVEC_SIZE-1) + 1);
		if (nr_pages == 0)
			break;

		for (i = 0; i < nr_pages; i++) {
			struct page *page = pvec.pages[i];

			/*
			 * At this point, the page may be truncated or
			 * invalidated (changing page->mapping to NULL), or
			 * even swizzled back from swapper_space to tmpfs file
			 * mapping. However, page->index will not change
			 * because we have a reference on the page.
```

```
			 */
			if (page->index > end) {
				/*
				 * can't be range_cyclic (1st pass) because
				 * end == -1 in that case.
				 */
				done = 1;
				break;
			}

			done_index = page->index + 1;

			lock_page(page);

			/*
			 * Page truncated or invalidated. We can freely skip it
			 * then, even for data integrity operations: the page
			 * has disappeared concurrently, so there could be no
			 * real expectation of this data interity operation
			 * even if there is now a new, dirty page at the same
			 * pagecache address.
			 */
			if (unlikely(page->mapping != mapping)) {
continue_unlock:
				unlock_page(page);
				continue;
			}

			if (!PageDirty(page)) {
				/* someone wrote it for us */
				goto continue_unlock;
```

```
        }

        if (PageWriteback(page)) {
                if (wbc->sync_mode != WB_SYNC_NONE)
                        wait_on_page_writeback(page);
                else
                        goto continue_unlock;
        }

        BUG_ON(PageWriteback(page));
        if (!clear_page_dirty_for_io(page))
                goto continue_unlock;

        ret = (*writepage)(page, wbc, data);
        if (unlikely(ret)) {
                if (ret == AOP_WRITEPAGE_ACTIVATE) {
                        unlock_page(page);
                        ret = 0;
                } else {
                        /*
                         * done_index is set past this page,
                         * so media errors will not choke
                         * background writeout for the entire
                         * file. This has consequences for
                         * range_cyclic semantics (ie. it may
                         * not be suitable for data integrity
                         * writeout).
                         */
                        done = 1;
                        break;
                }
```

```
			}

			if (nr_to_write > 0) {
				nr_to_write--;
				if (nr_to_write == 0 &&
				    wbc->sync_mode == WB_SYNC_NONE) {
					/*
					 * We stop writing back only if we are
					 * not doing integrity sync. In case of
					 * integrity sync we have to keep going
					 * because someone may be concurrently
					 * dirtying pages, and we might have
					 * synced a lot of newly appeared dirty
					 * pages, but have not synced all of the
					 * old dirty pages.
					 */
					done = 1;
					break;
				}
			}
		}
		pagevec_release(&pvec);
		cond_resched();
	}
	if (!cycled && !done) {
		/*
		 * range_cyclic:
		 * We hit the last page and there is more work to be done: wrap
		 * back to the start of the file
		 */
		cycled = 1;
```

```
			index = 0;
			end = writeback_index - 1;
			goto retry;
	}
	if (!wbc->no_nrwrite_index_update) {
		if (wbc->range_cyclic || (range_whole && nr_to_write > 0))
			mapping->writeback_index = done_index;
		wbc->nr_to_write = nr_to_write;
	}

	return ret;
}
EXPORT_SYMBOL(write_cache_pages);
```

5.4.4.12 __mpage_writepage

```
static int __mpage_writepage(struct page *page, struct writeback_control *wbc,
			void *data)
{
	struct mpage_data *mpd = data;
	struct bio *bio = mpd->bio;
	struct address_space *mapping = page->mapping;
	struct inode *inode = page->mapping->host;
	const unsigned blkbits = inode->i_blkbits;
	unsigned long end_index;
	const unsigned blocks_per_page = PAGE_CACHE_SIZE >> blkbits;
	sector_t last_block;
	sector_t block_in_file;
	sector_t blocks[MAX_BUF_PER_PAGE];
	unsigned page_block;
	unsigned first_unmapped = blocks_per_page;
```

```
struct block_device *bdev = NULL;
int boundary = 0;
sector_t boundary_block = 0;
struct block_device *boundary_bdev = NULL;
int length;
struct buffer_head map_bh;
loff_t i_size = i_size_read(inode);
int ret = 0;

if (page_has_buffers(page)) {
        struct buffer_head *head = page_buffers(page);
        struct buffer_head *bh = head;
        /* If they're all mapped and dirty, do it */
        page_block = 0;
        do {
                BUG_ON(buffer_locked(bh));
                if (!buffer_mapped(bh)) {
                        /*
                         * unmapped dirty buffers are created by
                         * __set_page_dirty_buffers -> mmapped data
                         */
                        if (buffer_dirty(bh))
                                goto confused;
                        if (first_unmapped == blocks_per_page)
                                first_unmapped = page_block;
                        continue;
                }

                if (first_unmapped != blocks_per_page)
                        goto confused;    /* hole -> non-hole */

```

```
498                     if (!buffer_dirty(bh) || !buffer_uptodate(bh))
499                             goto confused;
500                     if (page_block) {
501                             if (bh->b_blocknr != blocks[page_block-1] + 1)
502                                     goto confused;
503                     }
504                     blocks[page_block++] = bh->b_blocknr;
505                     boundary = buffer_boundary(bh);
506                     if (boundary) {
507                             boundary_block = bh->b_blocknr;
508                             boundary_bdev = bh->b_bdev;
509                     }
510                     bdev = bh->b_bdev;
511             } while ((bh = bh->b_this_page) != head);
512
513             if (first_unmapped)
514                     goto page_is_mapped;
515
516             /*
517              * Page has buffers, but they are all unmapped. The page was
518              * created by pagein or read over a hole which was handled by
519              * block_read_full_page().  If this address_space is also
520              * using mpage_readpages then this can rarely happen.
521              */
522             goto confused;
523     }
```

475-523 如果 page 已经有对应的块缓存，则进行处理。

```
524
525     /*
526      * The page has no buffers: map it to disk
527      */
```

```
	BUG_ON(!PageUptodate(page));
	block_in_file = (sector_t)page->index << (PAGE_CACHE_SHIFT - blkbits);
	last_block = (i_size - 1) >> blkbits;
	map_bh.b_page = page;
```

文件中写入的块号范围计算[block_in_file，last_block]。

```
	for (page_block = 0; page_block < blocks_per_page; ) {
```

for 循环依次写入每个块。

```

		map_bh.b_state = 0;
		map_bh.b_size = 1 << blkbits;
		if (mpd->get_block(inode, block_in_file, &map_bh, 1))
```

将文件中的逻辑块号转换为磁盘中的逻辑块号。

```
			goto confused;
		if (buffer_new(&map_bh))
			unmap_underlying_metadata(map_bh.b_bdev,
						map_bh.b_blocknr);
		if (buffer_boundary(&map_bh)) {
			boundary_block = map_bh.b_blocknr;
			boundary_bdev = map_bh.b_bdev;
		}
		if (page_block) {
			if (map_bh.b_blocknr != blocks[page_block-1] + 1)
				goto confused;
		}
		blocks[page_block++] = map_bh.b_blocknr;
		boundary = buffer_boundary(&map_bh);
		bdev = map_bh.b_bdev;
		if (block_in_file == last_block)
			break;
		block_in_file++;
```

```
	}
	BUG_ON(page_block == 0);

	first_unmapped = page_block;

page_is_mapped:
	end_index = i_size >> PAGE_CACHE_SHIFT;
	if (page->index >= end_index) {
		/*
		 * The page straddles i_size.   It must be zeroed out on each
		 * and every writepage invocation because it may be mmapped.
		 * "A file is mapped in multiples of the page size.   For a file
		 * that is not a multiple of the page size, the remaining memory
		 * is zeroed when mapped, and writes to that region are not
		 * written out to the file."
		 */
		unsigned offset = i_size & (PAGE_CACHE_SIZE - 1);

		if (page->index > end_index || !offset)
			goto confused;
		zero_user_segment(page, offset, PAGE_CACHE_SIZE);
	}

	/*
	 * This page will go to BIO.   Do we need to send this BIO off first?
	 */
	if (bio && mpd->last_block_in_bio != blocks[0] - 1)
		bio = mpage_bio_submit(WRITE, bio);

alloc_new:
	if (bio == NULL) {
```

```
        bio = mpage_alloc(bdev, blocks[0] << (blkbits - 9),
                bio_get_nr_vecs(bdev), GFP_NOFS|__GFP_HIGH);
        if (bio == NULL)
            goto confused;
    }

    /*
     * Must try to add the page before marking the buffer clean or
     * the confused fail path above (OOM) will be very confused when
     * it finds all bh marked clean (i.e. it will not write anything)
     */
    length = first_unmapped << blkbits;
    if (bio_add_page(bio, page, length, 0) < length) {
        bio = mpage_bio_submit(WRITE, bio);
        goto alloc_new;
    }
```

创建并设置 bio。

```

    /*
     * OK, we have our BIO, so we can now mark the buffers clean.   Make
     * sure to only clean buffers which we know we'll be writing.
     */
    if (page_has_buffers(page)) {
        struct buffer_head *head = page_buffers(page);
        struct buffer_head *bh = head;
        unsigned buffer_counter = 0;

        do {
            if (buffer_counter++ == first_unmapped)
                break;
            clear_buffer_dirty(bh);
```

```
			bh = bh->b_this_page;
		} while (bh != head);

		/*
		 * we cannot drop the bh if the page is not uptodate
		 * or a concurrent readpage would fail to serialize with the bh
		 * and it would read from disk before we reach the platter.
		 */
		if (buffer_heads_over_limit && PageUptodate(page))
			try_to_free_buffers(page);
	}

	BUG_ON(PageWriteback(page));
	set_page_writeback(page);
	unlock_page(page);
	if (boundary || (first_unmapped != blocks_per_page)) {
		bio = mpage_bio_submit(WRITE, bio);
```

提交 bio 写请求。

```
		if (boundary_block) {
			write_boundary_block(boundary_bdev,
					boundary_block, 1 << blkbits);
		}
	} else {
		mpd->last_block_in_bio = blocks[blocks_per_page - 1];
	}
	goto out;

confused:
	if (bio)
		bio = mpage_bio_submit(WRITE, bio);

```

```
        if (mpd->use_writepage) {
                ret = mapping->a_ops->writepage(page, wbc);
        } else {
                ret = -EAGAIN;
                goto out;
        }
        /*
         * The caller has a ref on the inode, so *mapping is stable
         */
        mapping_set_error(mapping, ret);
out:
        mpd->bio = bio;
        return ret;
}
```

6. 参考书籍

[1] linux 内核 2.6.34 源码

[2] 深入了解 linux 内核 （第三版）

[3] linxu 内核源代码情景分析 毛德超

[4] 深入 linxu 内核架构

[5] http://blog.csdn.net/yunsongice(网络资源)