请先了解linux vfs

Ext2的一般特征

前面对虚拟文件系统VFS的讨论就告一段落了，VFS主要是提供一个系统调用接口，然后将相关文件系统对象与具体的文件系统串联起来。从本文开始，我们将选择一个具体的文件系统进行研究，这个文件系统就是第二扩展文件系统（Ext2）。为啥要特别学习这个东西呢？因为ext2是 Linux所固有的，事实上已在每个Linux系统中得以使用，因此我们自然要对Ext2进行讨论。此外，Ext2在对现代文件系统的高性能支持方面也显 示出很多良好的实践性。固然，Linux支持的其他文件系统有很多有趣的特点，但比较扩展文件系统是Linux中的老大，头头，所以我们并不对其他文件系 统一一讨论了。

类Unix操作系统使用多种文件系统。尽管所有这些文件系统都有少数POSIX API（如state()）所需的共同的属性子集，但每种文件系统的实现方式是不同的。

Linux的第一个版本是基于MINIX文件系统的。当Linux成熟时，引入了扩展文件系统(**Extended Filesystem**, Ext FS)，它包含了几个重要的扩展但提供的性能不令人满意。在1994年引入了第二扩展文件系统（Ext2）；它除了包含几个新的特点外，还相当高性能和稳 定，Ext2及它的下代文件系统Ext3、e3fs等已成为广泛使用的Linux文件系统。

Ext2文件系统具有以下一般特点：

1、当创建Ext2文件系统时，系统管理员可以根据预期的文件平均长度来选择最佳的块大小（从1024B——4096B）。例如，当文件的平均长度小于几千字节时，块的大小为1024B是最佳的，因为这会产生较少的内部碎片——也就是文件长度与存放块的磁盘分区有较少的不匹配。另一方面，大的块对于 大于几千字节的文件通常比较合合适，因为这样的磁盘传送较少，因而减轻了系统的开销。

2、当创建Ext2文件系统时，系统管理员可以根据在给定大小的分区上预计存放的文件数来选择给该分区分配多少个索引节点。这可以有效地利用磁盘的空间。

3、文件系统把磁盘块分为组。每组包含存放在相邻磁道上的数据块和索引节点。正是这种结构，使得可以用较少的磁盘平均寻道时间对存放在一个单独块组中的文件并行访问。

4、在磁盘数据块被实际使用之前，文件系统就把这些块预分配给普通文件。因此当文件的大小增加时，因为物理上相邻的几个块已被保留，这就减少了文件的碎片。

5、支持快速符号链接。如果符号链接表示一个短路径名（小于或等于60个字符），就把它存放在索引节点中而不用通过由一个数据块进行转换。

此外，Ext2还包含了一些使它既健壮又灵活的特点：

1、文件更新策略的谨慎实现将系统崩溃的影响减到最少。我们只举一个例子来体现这个优点：例如，当给文件创建一个硬链接时，首先增加磁盘索引节点中 的硬链接计数器，然后把这个新的名字加到合适的目录中。在这种方式下，如果在更新索引节点后而改变这个目录之前出现一个硬件故障，这样即使索引节点的计数 器产生错误，但目录是一致的。因此，尽管删除文件时无法自动收回文件的数据块，但并不导致灾难性的后果。如果这种处理的顺序相反（更新索引节点前改变目 录），同样的硬件故障将会导致危险的不一致，删除原始的硬链接就会从磁盘删除它的数据块，但新的目录项将指向一个不存在的索引节点。如果那个索引节点号以 后又被另外的文件所使用，那么向这个旧目录的写操作将毁坏这个新的文件。

2、在启动时支持对文件系统的状态进行自动的一致性检查。这种检查是由外部程序e2fsck完成的，这个外部程序不仅可以在系统崩溃之后被激活，也 可以在一个预定义的文件系统安装数（每次安装操作之后对计数器加1）之后被激活，或者在自从最近检查以来所花的预定义时间之后被激活。

3、支持不可变（immutable）的文件（不能修改、删除和更名）和仅追加（append-only）的文件（只能把数据追加在文件尾）。

4、既与Unix System V Release 4（SVR4）相兼容，也与新文件的用户组ID的BSD语义相兼容。在SVR4中，新文件采用创建它的进程的用户组ID；而在BSD中，新文件继承包含它 的目录的用户组ID。Ext2包含一个安装选项，由你指定采用哪种语义。

即使Ext2文件系统是如此成熟、稳定的程序，也还要考虑引入另外几个负面特性。目前，一些负面特性已新的文件系统或外部补丁避免了。另外一些还仅仅处于计划阶段，但在一些情况下，已经在Ext2的索引节点中为这些特性引入新的字段。最重要的一些特点如下：

块片（block fragmentation）

系统管理员对磁盘的访问通常选择较大的块，因为计算机应用程序常常处理大文件。因此，在大块上存放小文件就会浪费很多磁盘空间。这个问题可以通过把几个文件存放在同一块的不同片上来解决。

透明地处理压缩和加密文件

这些新的选项（创建一个文件时必须指定）将允许用户透明地在磁盘上存放压缩和（或）加密的文件版本。

逻辑删除

一个undelete选项将允许用户在必要时很容易恢复以前已删除的文件内容。

日志

日志避免文件系统在被突然卸载（例如，作为系统崩溃的后果）时对其自动进行的耗时检查。

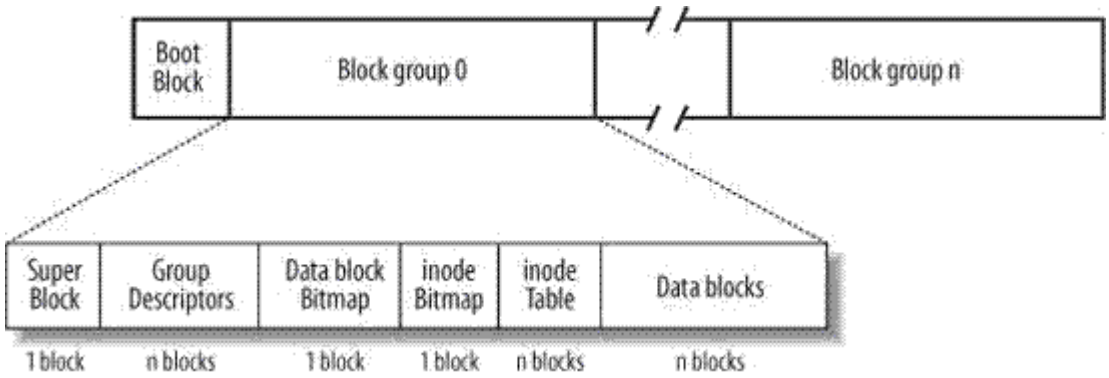
实际上，这些特点没有一个正式地包含在Ext2文件系统中。有人可能说Ext2是这种成功的牺牲品；直到几年前，它仍然是大多数Linux发布公司采用的首选文件系统，每天有成千上万的用户在使用它，这些用户会对用其他文件系统来代替Ext2的任何企图产生质疑。

Ext2中缺少的最突出的功能就是日志，日志是高可用服务器必需的功能。为了平顺过渡，日志没有引入到Ext2文件系统；但是，我们在后面 “Ext3文件系统”中会讨论，完全与Ext2兼容的一种新文件系统已经创建，这种文件系统提供了日志。不真正需要日志的用户可以继续使用良好而老式的Ext2文件系统，而其他用户可能采用这种新的文件系统。现在发行的大部分系统采用Ext3作为标准的文件系统。

Ext2磁盘数据结构

要学习ext2文件系统，首先要学习ext2的磁盘设计，也就是说，ext2将一个磁盘分区格式化成一个什么样子。本文就来介绍一下一个ext2文件系统的磁盘分区布局，重点关注它的数据结构。

任何Ext2分区中的第一个块从不受Ext2文件系统的管理，因为这一块是为分区的**引导扇区**所保留的。Ext2分区的其余部分被分成**块组**（block group），每个块组的分布图如图所示。正如你从图中所看到的，一些数据结构正好可以放在一块中，而另一些可能需要更多的块。在Ext2文件系统中的所有块组大小相同并被顺序存放，因此，内核可以从块组的整数索引很容易地得到磁盘中一个块组的位置：



由于内核尽可能地把属于同一个文件的数据块存放在同一块组中，所以块组减少了文件碎片。**块组**中的每个块包含下列信息之一：

1.文件系统的超级块的一个拷贝

2.一组块组描述符的拷贝

3.一个数据块位图

4.一个索引节点位图

5.一个索引节点表

6.属于文件的一大块数据，即数据块

如果一个块中不包含任何有意义的信息，就说这个块是空闲的。

从上图中可以看出，超级块与组描述符被复制到每个块组中。

注意:块组由很多‘块’组成.

其实呢，只有块组0中所包含超级块和组描述符才由内核使用，而其余的超级块和组描述符都保持不变；事实上，内核甚至不考虑它们。当e2fsck程序对Ext2文件系统的状态执行一致性检查时，就引用存放在块组0中的超级块和组描述符，然后把它们拷贝到其他所有的块组中。如果出现数据损坏，并且块组0 中的主超级块和主描述符变为无效，那么，系统管理员就可以命令e2fsck引用存放在某个块组（除了第一个块组）中的超级块和组描述符的旧拷贝。通常情况 下，这些多余的拷贝所存放的信息足以让e2fsck把Ext2分区带回到一个一致的状态。

那么有多少块组呢？这取决于分区的大小和块的大小。其主要限制在于块位图，因为**块位图必须存放在一个单独的块中**。块位图用来标识一个组中块的占用和 空闲状况。所以，每组中至多可以有8×b个块，b是以字节为单位的块大小。例如，一个块是 1024 Byte，那么，一个块的位图就有8192个位，一个块组正好就对应8192个块(位图中的一个bit描述一个块)。

举例说明，让我们考虑一下32GB的Ext2分区，换算成KB就是33554432，块的大小为4KB。在这种情况下，每个4KB的块位图描述 32KB个数据块，即128MB。因此，最多需要33554432 / 4096 * 32 = 256个块组。显然，块越大，块组数越小。

磁盘超级块

Ext2在磁盘上的超级块存放在一个**ext2_super_block**结构中，它的字段在下面列出（为了确保Ext2和Ext3文件系统之间的兼容性，ext2_super_block数据结构包含了一些Ext3特有的字段，我们省略号省了）(注:它与应VFS中的super_block)：

```
struct ext2_super_block {
    __le32    s_inodes_count;       /* 索引节点的总数 */
    __le32    s_blocks_count;       /* 块总数（所有的块） */
    __le32    s_r_blocks_count;     /* 保留的块数 */
    __le32    s_free_blocks_count;  /* 空闲块数 */
    __le32    s_free_inodes_count;  /* 空闲索引节点数 */
    __le32    s_first_data_block;   /* 第一个使用的块号（总为1） */
    __le32    s_log_block_size;          /* 块的大小 */
    __le32    s_log_frag_size;           /* 片的大小 */
    __le32    s_blocks_per_group;      /* 每组中的块数 */
    __le32    s_frags_per_group;        /* 每组中的片数 */
    __le32    s_inodes_per_group;     /* 每组中的索引节点数 */
    __le32    s_mtime;              /* 最后一次安装操作的时间 */
    __le32    s_wtime;              /* 最后一次写操作的时间 */
    __le16    s_mnt_count;          /* 被执行安装操作的次数 */
    __le16    s_max_mnt_count;      /* 检查之前安装操作的次数 */
    __le16    s_magic;              /* 魔术签名 */
    __le16    s_state;              /* 文件系统状态标志 */
    __le16    s_errors;             /* 当检测到错误时的行为 */
    __le16    s_minor_rev_level;    /* 次版本号 */
    __le32    s_lastcheck;          /* 最后一次检查的时间 */
    __le32    s_checkinterval;      /* 两次检查之间的时间间隔 */
    __le32    s_creator_os;         /* 创建文件系统的操作系统 */
    __le32    s_rev_level;          /* 主版本号 */
    __le16    s_def_resuid;          /* 保留块的缺省UID */
    __le16    s_def_resgid;          /* 保留块的缺省用户组ID */
    /*
     * 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;		/* 第一个非保留的索引节点号 */
	__le16	s_inode_size;		/* 磁盘上索引节点结构的大小 */
	__le16	s_block_group_nr;	/* 这个超级块的块组号 */
	__le32	s_feature_compat;	/* 具有兼容特点的位图 */
	__le32	s_feature_incompat;	/* 具有非兼容特点的位图 */
	__le32	s_feature_ro_compat;	/* 只读兼容特点的位图 */
	__u8	s_uuid[16];		/* 128位文件系统标识符 */
	char	s_volume_name[16];	/* 卷名 */
	char	s_last_mounted[64];	/* 最后一个安装点的路径名 */
	__le32	s_algorithm_usage_bitmap; /* 用于压缩 */
	/*
	 * Performance hints.  Directory preallocation should only
	 * happen if the EXT2_COMPAT_PREALLOC flag is on.
	 */
	__u8	s_prealloc_blocks;	/* 预分配的块数 */
	__u8	s_prealloc_dir_blocks;	/* 为目录预分配的块数 */
	__u16	s_padding1;		/* 按字对齐 */
	……
	__u32	s_reserved[190];	/* 用null填充1024字节 */
};
```

__u8、__u16及__u32数据类型分别表示长度为8、16及32位的无符号数，而__s8、__s16及__s32数据类型表示长度为8、16及32位的有符号数。为清晰地表示磁盘上字或双字中字节的存放顺序，内核又使用了__le16、__le32、__be16和__be32数据类型，前两种类型分别表示字或双字的“小尾（little-endian）”排序方式（低阶字节在高位地址），而后两种类型分别表示字或双字的“大尾（big-endian）”排序方式（高阶字节在高位地址）：

```
typedef __signed__ char __s8;
typedef unsigned char __u8;

typedef __signed__ short __s16;
typedef unsigned short __u16;

typedef __signed__ int __s32;
typedef unsigned int __u32;
```

```
typedef __signed__ long long __s64;
typedef unsigned long long __u64;

typedef __u16 __bitwise __le16;
typedef __u16 __bitwise __be16;
typedef __u32 __bitwise __le32;
typedef __u32 __bitwise __be32;
typedef __u64 __bitwise __le64;
typedef __u64 __bitwise __be64;
```

有细心的朋友可以数一数这些u或者le后面的数字，然后加起来后除以8，可以发现刚刚小于512。不错，512B正是最小的块大小，再小一点一个块就装不下ext2_super_block了。所有我推断，为什么这里要用这些u、s、或者le然后后面加数字的形式，就是方便开发者避免超过一个块大小而看起来方便一点的原因。

s_inodes_count字段存放索引节点的个数，而s_blocks_count字段存放Ext2文件系统的块的个数。

s_log_block_size字段以2的幂次方表示块的大小，用1024字节作为单位。因此，0表示1024字节的块，1表示2048字节的块，如此等等。目前s_log_frag_size字段与slog_block_size字段相等，因为块片还没有实现。

s_blocks_per_group、s_frags_per_group与s_inodes_per_group字段分别存放每个块组中的块数、片数及索引节点数。

一些磁盘块保留给超级用户（或由s_def_resuid和s_def_resgid字段挑选给某一其他用户或用户组）。即使当普通用户没有空闲块可用时，系统管理员也可以用这些块继续使用Ext2文件系统。

s_mnt_count、s_max_mnt_count、s_lastcheck及s_checkinterval字段使系统启动时自动地检查 Ext2文件系统。在预定义的安装操作数完成之后，或自最后一次一致性检查以来预定义的时间已经用完，这些字段就导致e2fsck执行（两种检查可以一起 进行）。如果Ext2文件系统还没有被全部卸载(例如系统崩溃以后)，或内核在其中发现一些错误，则一致性检查在启动时要强制进行。如果Ext2文件系统 被安装或未被全部卸载，则s_state字段存放的值为0；如果被正常卸载，则这个字段的值为1；如果包含错误，则值为2。

块组描述符和位图

每个块组都有自己的组描述符，它是一个ext2_group_desc结构：

```
struct ext2_group_desc
{
     __le32     bg_block_bitmap;              /* 块位图的块号 */
     __le32     bg_inode_bitmap;             /* 索引节点位图的块号 */
     __le32     bg_inode_table;       /* 第一个索引节点表块的块号 */
     __le16     bg_free_blocks_count;  /* 组中空闲块的个数 */
     __le16     bg_free_inodes_count;   /* 组中空闲索引节点的个数 */
     __le16     bg_used_dirs_count;     /* 组中目录的个数 */
     __le16     bg_pad;                  /* 按字对齐 */
     __le32     bg_reserved[3];         /* 用null填充24个字节 */
};
```

注意，每个块组都有n个块组描述符的拷贝，n等于块的个数，也就是刚才例子中的256个块组。没看懂的再仔细回忆回忆前面的那个块图。

当分配新索引节点和数据块时，会用到bg_free_blocks_count、bg_free_inodes_count、和 bg_used_dirs_count字段。这些字段确定在最合适的块中给每个数据结构进行分配位图中位的序列，其中值0表示对应的索引节点块或数据块是空闲的，1表示占用。因为每个位图必须存放在一个单独的块中，又因为块的大小可以是1024、2048或4096字节，因此，一个单独的位图描述 8192、16384或32768个块的状态。

注意有两种位图:块位图,inode位图.

磁盘索引节点表

索引节点表由一连串连续的块组成，其中每一块包含索引节点的一个预定义号。索引点表第一个块的块号存放在组描述符的bg_inode_table字段中。共有n个块，n等于块组的数量（256）。

所有索引节点的大小相同，即128字节。一个1024字节的块可以包含8个索引节点，一个4096字节的块可以包含32个索引节点。为了计算出索引 节点表占用了多少块，用一个组中的索引节点总数（存放在超级块的s_inodes_per_group字段中）除以每块的索引节点数。

每个Ext2索引节点为ext2_inode结构：

```
struct ext2_inode {
     __le16     i_mode;          /* 文件类型和访问权限 */
     __le16     i_uid;          /* 拥有者标识符 */
     __le32     i_size;          /* 以字节为单位的文件长度 */
     __le32     i_atime;     /* 最后一次访问文件的时间 */
     __le32     i_ctime;     /* 索引节点最后改变的时间 */
```

```
    __le32    i_mtime;     /* 文件内容最后改变的时间 */
    __le32    i_dtime;     /* 文件删除的时间 */
    __le16    i_gid;          /* 用户组标识符低16位 */
    __le16    i_links_count;     /* 硬链接计数器 */
    __le32    i_blocks;     /* 文件的数据块数 */
    __le32    i_flags;     /* 文件标志 */
    union {
         struct {
              __le32  l_i_reserved1;
         } linux1;
         struct {
              __le32  h_i_translator;
         } hurd1;
         struct {
              __le32  m_i_reserved1;
         } masix1;
    } osd1;                    /* 特定的操作系统信息 1 */
    __le32    i_block[EXT2_N_BLOCKS]; /* 指向数据块的指针 */
    __le32    i_generation;     /* 文件版本（当网络文件系统访问文件
时） */
    __le32    i_file_acl;     /* 文件访问控制列表 */
    __le32    i_dir_acl;     /* 目录访问控制列表 */
    __le32    i_faddr;     /* 片的地址 */
    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;                         /* 特定的操作系统信息 2 */
};
```

与POSIX规范相关的很多字段类似于VFS索引节点对象的相应字段，这已在“VFS文件系统对象”中讨论过。其余的字段与Ext2的特殊实现相关，主要处理块的分配。

特别地，i_size字段存放以字节为单位的文件的有效长度，而i_blocks字段存放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（因为还没实现片）且 i_size可能小于512 x i_blocks。另一方面，一个文件可能包含有洞。在那种情况下，i_size可能大于512 x i_blocks。

i_block字段是具有EXT2_N_BLOCKS（通常是15）个指针元素的一个数组，每个元素指向分配给文件的数据块：

```
#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)
```

留给i_size字段的32位把文件的大小限制到4GB。事实上，i size字段的最高位没有使用，因此，文件的最大长度限制为2GB。然而，Ext2文件系统包含一种“脏技巧”，允许像AMD的Opteron和IBM的 PowerPC G5这样的64位体系结构使用大型文件。从本质上说，索引节点的i_dir_acl字段（普通文件没有使用）表示i_size字段的32位扩展。因此，文 件的大小作为64位整数存放在索引节点中。Ext2文件系统的64位版本与32位版本在某种程度上兼容，因为在64位体系结构上创建的Ext2文件系统可 以安装在32位体系结构上，反之亦然。但是，在32位体系结构上不能访问大型文件，除非以O_LARGEFILE标志打开文件（参“VFS系统调用的实现”）。

回忆一下，VFS模型要求每个文件有不同的索引节点号。在Ext2中，没有必要在磁盘上存放文件的索引节点号与相应块号之间的转换，因为后者的值可 以从块组号和它在索引节点表中的相对位置而得出。例如，假设每个块组包含4096个索引节点，我们想知道索引节点13021在磁盘上的地址。在这种情况 下，这个索引节点属于第三个块组，它的磁盘地址存放在相应索引节点表的第733个表项中（13021 - 4096 -4096）。所以，inode数据结构的索引节点号i_ino是Ext2文件系统用来快速搜索磁盘上合适的索引节点描述符的一个非常关键的字段。

访问控制列表

很早以前访问控制列表就被建议用来改善Unix文件系统的保护机制。其将文件的权限分成三类：**拥有者、组和其他**，访问控制列表（access control list，ACL）可以与其他文件关联。有了这种列表，用户可以为他的文件限定可以访问的用户（或用户组）名称以及相应的权限。

Linux 2.6通过索引节点的增强属性完整实现ACL。实际上，增强属性主要就是为了支持ACL才引入的。因此，能让你处理文件ACL的库函数chacl()、setfacl()和getfacl()是通过setxattr()和getxattr()系统调用实现的。

不幸的是，在POSIX 1003.1系列标准内，定义安全增强属性的工作组所完成的成果从没有正式成为新的POSIX标准。因此现在，不同的类Unix文件系统都支持ACL，但不同的实现之间有一些微小的差别。

各种文件类型如何使用磁盘块

Ext2所认可的文件类型（普通文件、管道文件等）以不同的方式使用数据块。有些文件不存放数据，因此根本就不需要数据块。我们来讨论一些文件类型的存储要求：

普通文件

普通文件是最常见的情况，我们要重点关注它。但普通文件只有在开始有数据时才需要数据块。普通文件在刚创建时是空的，并不需要数据块；也可以用truncate()或open()系统调用清空它。这两种情况是相同的，例如，当你发出一个包含字符串 > filename的shell命令时，shell创建一个空文件或截断一个现有文件。

目录

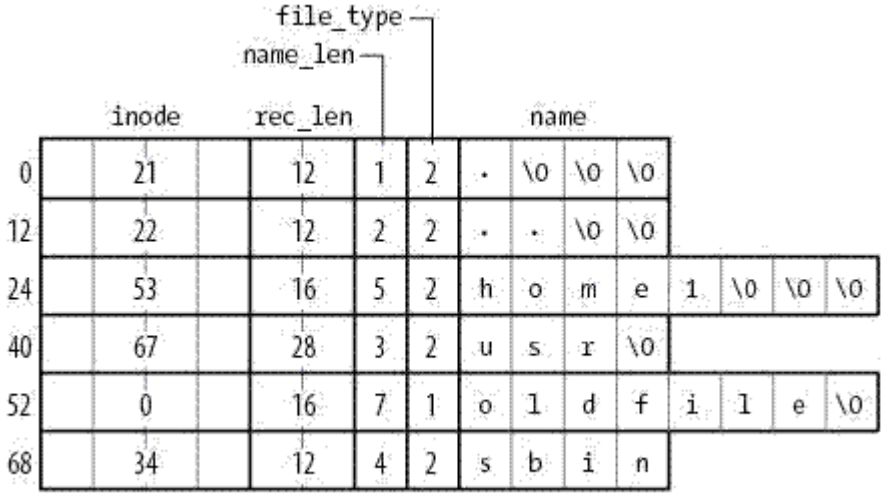
Ext2以一种特殊的文件实现了目录，这种文件的数据块把文件名和相应的索引节点号存放在一起。特别说明的是，这样的数据块包含了类型为ext2_dir_entry_2的结构：

```
#define EXT2_NAME_LEN 255
struct ext2_dir_entry_2 {
    __le32    inode;          /* 索引节点号 */
    __le16    rec_len;        /* 目录项长度 */
    __u8      name_len;       /* 文件名长度 */
    __u8      file_type;          /* 文件类型 */
```

```
    char    name[EXT2_NAME_LEN];    /* 文件名 */
};
```

因为该结构最后一个name字段是最大为EXT2_NAME_LEN（通常是255）个字符的变长数组，因此这个结构的长度是可变的。此外，因为效 率的原因，目录项的长度总是4的倍数，并在必要时用null字符（\0）填充文件名的末用的name_len字段存放实际的文件名长度（参见下图）。

注意:此处的目录项与VFS中的dentry不同. 这里的目录项是指目录文件中的一个item(项). 目录文件中的一个item的类型是ext2_dir_entry_2. ext2_dir_entry2实例的长度是可变的,其长度由其rec_len成员指定,而其name成员的实际长度由name_len指定.(这里我们看到了对ext2_dir_entry_2的hack)



file_type字段存放指定文件类型的值（见下表）：

文件类型	描述
0	未知
1	普通文件
2	目录
3	字符设备
4	块设备
5	命名管道
6	套接字
7	符号链接

rec_len字段可以被解释为指定一个有效目录项的指针：它是偏移量，与目录项的起始地址相加就得到下一个有效目录的起始地址。为了删除一个目录 项，把它的inode字段置为0并适当地增加前一个有效目录项rec_len字段的值就足够了。

仔细看一下上边图中的rec_len字段，你会发现 oldfile项已被删除，因为usr的rec_len字段被置为12+16（usr和oldfile目录项的长度）。

符号链接

如前所述，如果符号链接的路径名**小于等于60**个字符，就把它存放在索引节点的i_blocks字段，该字段是由15个4字节整数组成的数组，因此无需数据块。但是，如果路径名大于60个字符，就需要一个单独的数据块。

设备文件、管道和套接字

这些类型的文件不需要数据块。所有必要的信息都存放在索引节点中。

Ext2的超级块对象

前面我们详细讨论了将一个磁盘分区格式化成ext2文件系统后，一个分区的布局，重点介绍了超级快、块组、位图和索引节点等内容。那么，内核如何跟这些ext2文件系统的对象打交道呢？这个，才是我们研究存储的人应该重点关注的对象，从本文开始，我们就来重点讨论。

首先，当安装Ext2文件系统时（执行诸如mount -t ext2 /dev/sda2 /mnt/test的命令），存放在Ext2分区的磁盘数据结构中的大部分信息将被拷贝到RAM中，从而使内核避免了后来的很多读操作。那么一些数据结构 如何经常更新呢？让我们考虑一些基本的操作：

1、当一个新文件被创建时，必须减少磁盘中Ext2超级块中s_free_inodes_count字段的值和相应的组描述符中bg_free_inodes_count字段的值。

2、如果内核给一个现有的文件追加一些数据，以使分配给它的数据块数因此也增加，那么就必须修改Ext2超级块中s_free_blocks_count字段的值和组描述符中bg_free_blocks_count字段的值。

3、即使仅仅重写一个现有文件的部分内容，也要对Ext2超级块的s_wtime字段进行更新。

因为所有的Ext2磁盘数据结构都存放在Ext2磁盘分区的块中，因此，内核利用页高速缓存来保持它们最新（参见“磁盘高速缓存”博文）。下面我们就一项一项的来讲解内核如何跟他们打交道，本篇博文首先介绍ext2超级块对象。

在“VFS文件对象”中我们介绍过，VFS超级块的s_fs_info字段指向一个文件系统信息的数据结构：

```
struct super_block {
      ……

      void    *s_fs_info; /* Filesystem private info */

      ……
};
```

对于Ext2文件系统，该字段指向**ext2_sb_info类型**的结构：

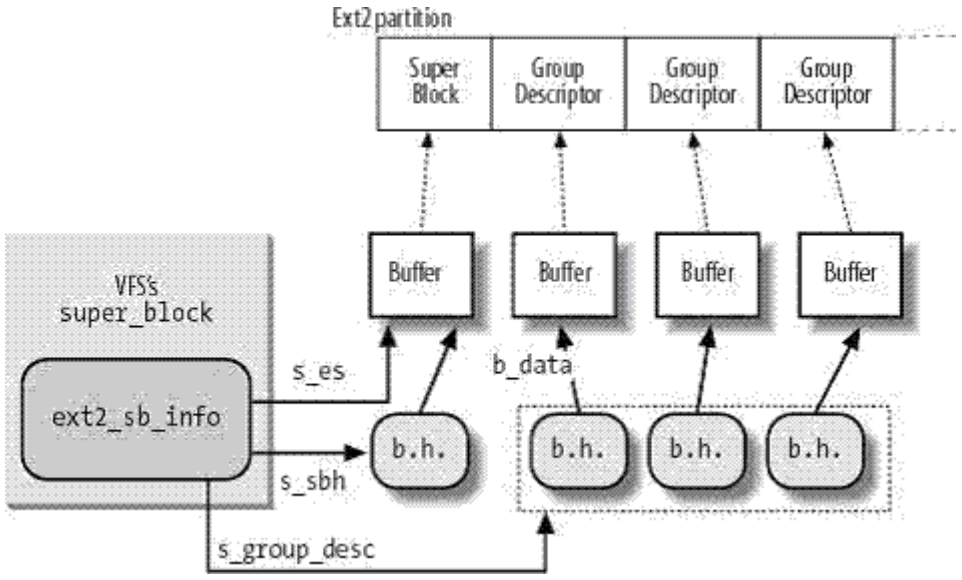
```
struct ext2_sb_info {
  unsigned long s_frag_size;    /* Size of a fragment in bytes */
  unsigned long s_frags_per_block;  /* Number of fragments per block */
  unsigned long s_inodes_per_block; /* Number of inodes per block */
  unsigned long s_frags_per_group;  /* Number of fragments in a group */
  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 */
  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;
  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;
};
```

注: ext2_sb_info来自于ext2_super_block(这是inode的磁盘表示).

其含如下信息:
- 磁盘超级块中的大部分字段
- s_sbh指针，指向包含磁盘超级块的缓冲区的缓冲区首部
- s_es指针，指向磁盘超级块所在的缓冲区
- 组描述符的个数s_desc_per_block，可以放在一个块中
- s_group_desc指针，指向一个缓冲区（包含组描述符的缓冲区）首部数组（只用一项就够了）
- 其他与安装状态、安装选项等有关的数据

下图表示的是与Ext2超级块和组描述符有关的缓冲区与缓冲区首部和ext2_sb_info数据结构之间的关系。



要看懂上面那个图以及后面的文字，建议大家先去看一下“页高速缓存”的相关内容。当内核安装Ext2文件系统时（mount命令），它调用ext2_fill_super()函数来为数据结构分配空间，并写入从磁盘读取的数据（再看看“文件系统安装”）：

```
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);
     unsigned long logic_sb_block;
     unsigned long offset = 0;
     unsigned long def_mount_opts;
     int blocksize = BLOCK_SIZE;
     int db_count;
     int i, j;
     __le32 features;

     sbi = kmalloc(sizeof(*sbi), GFP_KERNEL);
     if (!sbi)
          return -ENOMEM;
     sb->s_fs_info = sbi;
     memset(sbi, 0, sizeof(*sbi));

     /*
      * 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) {
     printk ("EXT2-fs: unable to set blocksize\n");
     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;
}

if (!(bh = sb_bread(sb, logic_sb_block))) {
     printk ("EXT2-fs: unable to read superblock\n");
     goto failed_sbi;
}
/*
 * Note: s_es must be initialized as soon as possible because
 *       some ext2 macro-instructions depend on its value
 */
es = (struct ext2_super_block *) (((char *)bh->b_data) + offset);
sbi->s_es = es;
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);
if (def_mount_opts & EXT2_DEFM_XATTR_USER)
```

```
            set_opt(sbi->s_mount_opt, XATTR_USER);
        if (def_mount_opts & EXT2_DEFM_ACL)
            set_opt(sbi->s_mount_opt, POSIX_ACL);

        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_RO)
            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);

        if (!parse_options ((char *) data, sbi))
            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)))
            printk("EXT2-fs warning: feature flags set on rev 0 fs, "
                   "running e2fsck is recommended\n");
        /*
         * 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) {
            printk("EXT2-fs: %s: couldn't mount because of "
                   "unsupported optional features (%x).\n",
                   sb->s_id, 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))){
            printk("EXT2-fs: %s: couldn't mount RDWR because of "
```

```
                "unsupported optional features (%x).\n",
                sb->s_id, le32_to_cpu(features));
        goto failed_mount;
    }

    blocksize = BLOCK_SIZE <<
le32_to_cpu(sbi->s_es->s_log_block_size);

    if ((ext2_use_xip(sb)) && ((blocksize != PAGE_SIZE) ||
                    (sb->s_blocksize != blocksize))) {
        if (!silent)
            printk("XIP: Unsupported blocksize\n");
        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)) {
            printk(KERN_ERR "EXT2-fs: blocksize too small for
device.\n");
            goto failed_sbi;
        }

        logic_sb_block = (sb_block*BLOCK_SIZE) / blocksize;
        offset = (sb_block*BLOCK_SIZE) % blocksize;
        bh = sb_bread(sb, logic_sb_block);
        if(!bh) {
            printk("EXT2-fs: Couldn't read superblock on "
                   "2nd try.\n");
            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)) {
            printk ("EXT2-fs: Magic mismatch, very weird !\n");
            goto failed_mount;
        }
    }

    sb->s_maxbytes = ext2_max_size(sb->s_blocksize_bits);
```

```
if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV) {
     sbi->s_inode_size = EXT2_GOOD_OLD_INODE_SIZE;
     sbi->s_first_ino = EXT2_GOOD_OLD_FIRST_INO;
} else {
     sbi->s_inode_size = le16_to_cpu(es->s_inode_size);
     sbi->s_first_ino = le32_to_cpu(es->s_first_ino);
     if ((sbi->s_inode_size < EXT2_GOOD_OLD_INODE_SIZE) ||
         (sbi->s_inode_size & (sbi->s_inode_size - 1)) ||
         (sbi->s_inode_size > blocksize)) {
          printk ("EXT2-fs: unsupported inode size: %d\n",
               sbi->s_inode_size);
          goto failed_mount;
     }
}

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;
sbi->s_mount_state = le16_to_cpu(es->s_state);
sbi->s_addr_per_block_bits =
     log2 (EXT2_ADDR_PER_BLOCK(sb));
sbi->s_desc_per_block_bits =
     log2 (EXT2_DESC_PER_BLOCK(sb));

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

```
     if (sb->s_blocksize != bh->b_size) {
          if (!silent)
               printk ("VFS: Unsupported blocksize on dev "
                    "%s.\n", sb->s_id);
          goto failed_mount;
     }

     if (sb->s_blocksize != sbi->s_frag_size) {
          printk ("EXT2-fs: fragsize %lu != blocksize %lu (not supported
yet)\n",
               sbi->s_frag_size, sb->s_blocksize);
          goto failed_mount;
     }

     if (sbi->s_blocks_per_group > sb->s_blocksize * 8) {
          printk ("EXT2-fs: #blocks per group too big: %lu\n",
               sbi->s_blocks_per_group);
          goto failed_mount;
     }
     if (sbi->s_frags_per_group > sb->s_blocksize * 8) {
          printk ("EXT2-fs: #fragments per group too big: %lu\n",
               sbi->s_frags_per_group);
          goto failed_mount;
     }
     if (sbi->s_inodes_per_group > sb->s_blocksize * 8) {
          printk ("EXT2-fs: #inodes per group too big: %lu\n",
               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) +
                           EXT2_BLOCKS_PER_GROUP(sb) - 1) /
                           EXT2_BLOCKS_PER_GROUP(sb);
     db_count = (sbi->s_groups_count + EXT2_DESC_PER_BLOCK(sb) - 1) /
             EXT2_DESC_PER_BLOCK(sb);
     sbi->s_group_desc = kmalloc (db_count * sizeof (struct buffer_head
*), GFP_KERNEL);
     if (sbi->s_group_desc == NULL) {
          printk ("EXT2-fs: not enough memory\n");
          goto failed_mount;
     }
     bgl_lock_init(&sbi->s_blockgroup_lock);
```

```
	sbi->s_debts = kmalloc(sbi->s_groups_count * sizeof(*sbi->s_debts),
			       GFP_KERNEL);
	if (!sbi->s_debts) {
		printk ("EXT2-fs: not enough memory\n");
		goto failed_mount_group_desc;
	}
	memset(sbi->s_debts, 0, sbi->s_groups_count *
sizeof(*sbi->s_debts));
	for (i = 0; i < db_count; i++) {
		block = descriptor_loc(sb, logic_sb_block, i);
		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]);
			printk ("EXT2-fs: unable to read group descriptors\n");
			goto failed_mount_group_desc;
		}
	}
	if (!ext2_check_descriptors (sb)) {
		printk ("EXT2-fs: group descriptors corrupted!\n");
		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);

	percpu_counter_init(&sbi->s_freeblocks_counter,
			    ext2_count_free_blocks(sb));
	percpu_counter_init(&sbi->s_freeinodes_counter,
			    ext2_count_free_inodes(sb));
	percpu_counter_init(&sbi->s_dirs_counter,
			    ext2_count_dirs(sb));
	/*
	 * set up enough so that it can read an inode
	 */
	sb->s_op = &ext2_sops;
	sb->s_export_op = &ext2_export_ops;
	sb->s_xattr = ext2_xattr_handlers;
	root = iget(sb, EXT2_ROOT_INO);
	sb->s_root = d_alloc_root(root);
	if (!sb->s_root) {
		iput(root);
		printk(KERN_ERR "EXT2-fs: get root inode failed\n");
		goto failed_mount3;
```

```
        }
        if (!S_ISDIR(root->i_mode) || !root->i_blocks || !root->i_size) {
                dput(sb->s_root);
                sb->s_root = NULL;
                printk(KERN_ERR "EXT2-fs: corrupt root inode, run e2fsck\n");
                goto failed_mount3;
        }
        if (EXT2_HAS_COMPAT_FEATURE(sb, EXT3_FEATURE_COMPAT_HAS_JOURNAL))
                ext2_warning(sb, __FUNCTION__,
                        "mounting ext3 filesystem as ext2");
        ext2_setup_super (sb, es, sb->s_flags & MS_RDONLY);
        return 0;

cantfind_ext2:
        if (!silent)
                printk("VFS: Can't find an ext2 filesystem on dev %s.\n",
                       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);
        return -EINVAL;
}
```

注意，要读懂上面的代码请好好理解页高速缓存相关的内容，因为ext2磁盘超级快ext2_super_block缓存于bh->b_data的某个位置，而对应的块号就是1（0号是引导块）。这里是对该函数的一个简要说明，只强调缓冲区与描述符的内存分配：

1. 分配一个ext2_sb_info描述符，将其地址当作参数传递并存放在超级块sb的s_fs_info字段：

```
        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);
unsigned long logic_sb_block;
unsigned long offset = 0;
unsigned long def_mount_opts;
int blocksize = BLOCK_SIZE;
int db_count;
int i, j;
__le32 features;

sbi = kmalloc(sizeof(*sbi), GFP_KERNEL);
if (!sbi)
     return -ENOMEM;
sb->s_fs_info = sbi;
memset(sbi, 0, sizeof(*sbi));
```

2. 调用__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冲区中。在“在页高速缓存中搜索块”一博我们讨论过，如果 一个块已在页高速缓存的缓冲区页而且是最新的，那么无需再分配。将缓冲区首部地址存放在Ext2超级块对象sbi的s_sbh字段：

```
if (!(bh = sb_bread(sb, logic_sb_block))) {
     printk ("EXT2-fs: unable to read superblock\n");
     goto failed_sbi;
}
//这里的offset其实就是0。
es = (struct ext2_super_block *) (((char *)bh->b_data) + offset);
sbi->s_es = es;
……
```

3. 分配一个数组用于存放缓冲区首部指针，每个组描述符一个，把该数组地址存放在ext2_sb_info的s_group_desc字段。

```
db_count = (sbi->s_groups_count + EXT2_DESC_PER_BLOCK(sb) - 1) /
        EXT2_DESC_PER_BLOCK(sb);
sbi->s_group_desc = kmalloc (db_count * sizeof (struct buffer_head
*), GFP_KERNEL);
```

4. 分配一个字节数组，每组一个字节，把它的地址存放在ext2_sb_info描述符的s_debts字段（参见后面的“管理ext2磁盘空间”博文）：

```
sbi->s_debts = kmalloc(sbi->s_groups_count * sizeof(*sbi->s_debts),
                GFP_KERNEL);
if (!sbi->s_debts) {
     printk ("EXT2-fs: not enough memory\n");
     goto failed_mount_group_desc;
}
```

```
     memset(sbi->s_debts, 0, sbi->s_groups_count *
sizeof(*sbi->s_debts));
```

5. 重复调用__bread()分配缓冲区，从磁盘读人包含Ext2组描述符的块。把缓冲区首部地址存放在上一步得到的s_group_desc数组中：

```
     for (i = 0; i < db_count; i++) {
          block = descriptor_loc(sb, logic_sb_block, i);
          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]);
               printk ("EXT2-fs: unable to read group descriptors\n");
               goto failed_mount_group_desc;
          }
     }
```

6. 最后安装好sb->s_op，sb->s_export_op，超级块增强属性。因为准备为根目录分配一个索引节点和目录项对象，必须把s_op字段为超级块建立好，从而能够从磁盘读入根索引节点对象：

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

注意，EXT2_ROOT_INO是2，也就是它的根节点位于第一个块组的第三个位置上。很显然，ext2_fill super()函数返回后，有很多关键的ext2磁盘数据结构的内容都保存在内存里了，例如ext2_super_block、第根索引节点等，只有当 Ext2文件系统卸载时才会被释放。当内核必须修改Ext2超级块的字段时，它只要把新值写入相应缓冲区内的相应位置然后将该缓冲区标记为脏即可，有看页 高速缓存事情就是这么简单！

Ext2的索引节点对象

上文我们详细分析了VFS如何将具体文件系统ext2的超级快对象缓存到内存中，主要是利用了ext2_sb_info数据结构。ext2_fill_super函数最后的时候，会将传入该函数的super_block类型的参数sb赋予以下的值：

```
sb->s_op = &ext2_sops;//超级块支持的操作
```

其中，ext2_sops是将super_block的super_operations初始化以下方法的集合：

```
static struct super_operations ext2_sops = {
	.alloc_inode	= ext2_alloc_inode,
	.destroy_inode	= ext2_destroy_inode,
	.read_inode	= ext2_read_inode,
	.write_inode	= ext2_write_inode,
	.put_inode	= ext2_put_inode,
	.delete_inode	= ext2_delete_inode,
	.put_super	= ext2_put_super,
	.write_super	= ext2_write_super,
	.statfs		= ext2_statfs,
	.remount_fs	= ext2_remount,
	.clear_inode	= ext2_clear_inode,
	.show_options	= ext2_show_options,
#ifdef CONFIG_QUOTA
	.quota_read	= ext2_quota_read,
	.quota_write	= ext2_quota_write,
#endif
};
```

本文，我们来讨论VFS如何将ext2的索引节点缓存到内存中。

在打开文件时，要执行路径名查找。对于不在目录项高速缓存内的路径名元素，会创建一个新的目录项对象和索引节点对象（参见“标准路经名查找”）。当VFS访问一个Ext2磁盘索引节点时，它会创建一个ext2_inode_info类型的索引节点描述符：

```
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 ued 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;

	/*
	 * i_next_alloc_block is the logical (file-relative) number of the
	 * most-recently-allocated block in this file.  Yes, it is misnamed.
	 * We use this for detecting linearly ascending allocation requests.
	 */
	__u32	i_next_alloc_block;

	/*
	 * i_next_alloc_goal is the *physical* companion to
i_next_alloc_block.
	 * it the the physical block number of the block which was
most-recently
	 * allocated to this file.  This give us the goal (target) for the
next
	 * allocation when we detect linearly ascending requests.
	 */
	__u32	i_next_alloc_goal;
	__u32	i_prealloc_block;
	__u32	i_prealloc_count;
	__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
#ifdef CONFIG_EXT2_FS_POSIX_ACL
	struct posix_acl	*i_acl;
	struct posix_acl	*i_default_acl;
#endif
	rwlock_t i_meta_lock;
	struct inode	vfs_inode;
};
```

注:ext2_inode_info来自于ext2_inode(inode的磁盘表示).

该描述符包含下列信息：
- 存放在vfs mode字段的整个VFS索引节点对象；
- ext2磁盘索引节点对象结构中的大部分字段（不保存在VFS索引节点中的那些字段）；
- 块组中索引节点对应的索引i_block_group；
- i_next_alloc_block和i_next_alloc_goal分别存放着最近为文件分配的磁盘块的逻辑块号和物理块号；
- i_prealloc_block和i_prealloc_count字段，用于数据块预分配；
- xattr_sem字段，一个读写信号量，允许增强属性与文件数据同时读入；
- i_acl和i_default_acl字段，指向文件的访问控制列表。

当处理Ext2文件时,alloc_inode超级块方法是由ext2_alloc_inode()函数实现：

```
static struct inode *ext2_alloc_inode(struct super_block *sb)
{
	struct ext2_inode_info *ei;
	ei = (struct ext2_inode_info *)kmem_cache_alloc(ext2_inode_cachep,
SLAB_KERNEL);
	if (!ei)
		return NULL;
#ifdef CONFIG_EXT2_FS_POSIX_ACL
	ei->i_acl = EXT2_ACL_NOT_CACHED;
	ei->i_default_acl = EXT2_ACL_NOT_CACHED;
#endif
	ei->vfs_inode.i_version = 1;
	return &ei->vfs_inode;
}
```

该函数的实现极其简单，它首先从ext2_inode_cachep slab分配器高速缓存得到一个ext2_inode_info数据结构，然后返回在这个ext2_inode_info数据结构中的索引节点对象的地址ei->vfs_inode。

我们看到，跟ext2_sb_info和ext2_group_desc不同，ext2_inode_info根本没有对应的 buffer_head字段，也就是说，它只是把ext2_inode的某些字段拷贝进来，对位于磁盘的ext2_inode进行动态缓存.

下面，我们就 把ext2数据结构的VFS映像做个总结：

类型	磁盘数据结构	内存数据结构	缓存模式
Superblock	ext2_super_block	ext2_sb_info	总是缓存

Group descriptor	ext2_group_desc	ext2_group_desc	总是缓存
Block bitmap	Bit array in block	Bit array in buffer	动态缓存
inode bitmap	Bit array in block	Bit array in buffer	动态缓存
inode	ext2_inode	ext2_inode_info	动态缓存
Data block	Array of bytes	VFS buffer	动态缓存
Free inode	ext2_inode	None	从不缓存
Free block	Array of bytes	None	从不缓存

创建Ext2文件系统

在磁盘上创建一个文件系统通常有两个阶段。
第一步格式化磁盘，以使磁盘驱动程序可以将块号转换成对应的磁道和扇区，从而可以读和写磁盘上 的物理块。现在的硬磁盘已经由厂家预先格式化，因此不需要重新格式化；在Linux上可以使用superformat或fdformat等实用程序对软盘进行格式化。
第二步才涉及创建文件系统，这意味着建立前文中详细描述的那些磁盘文件系统数据结构。

Ext2文件系统是由C程序mke2fs创建的。mke2fs采用下列缺省选项，用户可以用命令行的标志修改这些选项：

- 块大小：1024字节（小文件系统的缺省值）
- 片大小：等于块的大小（因为块的分片还没有实现）
- 所分配的索引节点个数：每8192字节的组分配一个索引节点
- 保留块的百分比：5%

mke2fs程序执行下列操作：

1. 初始化超级块和组描述符。

2. 作为选择，检查分区是否包含有缺陷的块；如果有，就创建一个有缺陷块的链表。

3. 对于每个块组，保留存放超级块、组描述符、索引节点表及两个位图所需要的所有磁盘块。

4. 把索引节点位图和每个块组的数据映射位图都初始化为0。

5. 初始化每个块组的索引节点表。

6. 创建/root目录。

7. 创建lost+found目录，由e2fsck使用这个目录把丢失和找到的缺陷块连接起来。

8. 在前两个已经创建的目录所在的块组中，更新块组中的索引节点位图和数据块位图。

9. 把有缺陷的块（如果存在）组织起来放在lost+found目录中。

让我们看一下mke2fs是如何以缺省选项初始化Ext2的1.44 MB软盘的。

软盘一旦被安装，VFS就把它看作由1412个块组成的一个卷，每块大小为1024字节。为了查看磁盘的内容，我们可以执行如下Unix命令：

```
[root@localhost]# dd if=/dev/fd0 bs=1k count=1440 | od -tx1 -Ax > /tmp/dump_hex
```

从而获得了AMP目录下的一个文件，这个文件包含十六进制的软盘内容的转储。

通过查看dump_hex文件我们可以看到，由于软盘有限的容量1.44MB（约等于1474560个字节），一共需要1440个块，那么一个单独的块组描述符就足够了。我们还注意到保留的块数为72（1440块的5%）。

软盘中的第一个块是引导块，第二个块是超级快ext2_super_block，第三个块是组描述符ext2_group_desc，第四个块是数 据块的位图（所以ext2_group_desc中的bg_block_bitmap等于4），第五个块是索引节点的位图（所以ext2_group_desc中的bg_inode_bitmap等于5）。

根据缺省选项,索引节点表必须为每8192个字节设置一个索引节点,也就是有184个索引节点存放在紧挨着索引节点的位图那个块的后23个块中，也就是第6~28个块。

下表是对一个软盘执行mke2fs程序后的磁盘分布：

块　　内容
0　　引导块
1　　超级块
2　　仅包含一个组描述符的块
3　　数据块位图
4　　inode 位图
5-27　inode 表，其中：1到10号inode保留（2号inode是根目录的inode）；11号inode是lost+found；12-184号inode空闲
28　　根目录（包括 .、..、和lost+found)
29　　lost+found 目录（包括 . and ..)
30-40 预分配给lost+found目录的保留的块
41-1439
空闲块

Ext2的方法总结

在“VFS对象”中所描述的关于VFS的很多方法在Ext2都有相应的实现。因为对所有的方法都进行讨论需要整整一本书，因此我们仅仅简单地回顾一下在Ext2中所实现的方法。一旦你真正搞明白了磁盘和内存数据结构，你就应当能理解实现这些方法的Ext2函数的代码。

Ext2超级块的操作

在“Ext2的索引节点对象”我们已经分析了Ext2超级块的操作之一：ext2_alloc_inode，这里在对Ext2超级块的操作做一个综合性的总结。很多VFS超级块操作在Ext2中都有具体的实现，这些方法为alloc_inode、destroy_inode、read_inode、 write_inode、delete_inode、put_super、write_super、statfs、remount_fs和 clear_inode。超级块方法的地址存放在ext2_sops指针数组：

```
static struct super_operations ext2_sops = {
      .alloc_inode      = ext2_alloc_inode,
      .destroy_inode      = ext2_destroy_inode,
      .read_inode      = ext2_read_inode,
      .write_inode      = ext2_write_inode,
      .put_inode      = ext2_put_inode,
```

```
	.delete_inode	= ext2_delete_inode,
	.put_super	= ext2_put_super,
	.write_super	= ext2_write_super,
	.statfs		= ext2_statfs,
	.remount_fs	= ext2_remount,
	.clear_inode	= ext2_clear_inode,
	.show_options	= ext2_show_options,
#ifdef CONFIG_QUOTA
	.quota_read	= ext2_quota_read,
	.quota_write	= ext2_quota_write,
#endif
};
```

Ext2索引节点的操作

一些VFS索引节点的操作在Ext2中都有具体的实现，这取决于索引节点所指的文件类型。

如果**文件类型是普通文件**，则对应inode的方法存放在ext2_file_inode_operations表中：

```
struct inode_operations ext2_file_inode_operations = {
	.truncate	= ext2_truncate,
#ifdef CONFIG_EXT2_FS_XATTR
	.setxattr	= generic_setxattr,
	.getxattr	= generic_getxattr,
	.listxattr	= ext2_listxattr,
	.removexattr	= generic_removexattr,
#endif
	.setattr	= ext2_setattr,
	.permission	= ext2_permission,
	.fiemap		= ext2_fiemap,
};
```

注意，普通文件的inode操作有很多没有实现VFS索引节点所规定的那些动作。

如果**文件类型是目录**，那么对应inode的方法存放在ext2_dir_inode_operations表中：

```
struct inode_operations ext2_dir_inode_operations = {
	.create		= ext2_create,
	.lookup		= ext2_lookup,
	.link		= ext2_link,
	.unlink		= ext2_unlink,
```

```
	.symlink	= ext2_symlink,
	.mkdir		= ext2_mkdir,
	.rmdir		= ext2_rmdir,
	.mknod		= ext2_mknod,
	.rename		= ext2_rename,
#ifdef CONFIG_EXT2_FS_XATTR
	.setxattr	= generic_setxattr,
	.getxattr	= generic_getxattr,
	.listxattr	= ext2_listxattr,
	.removexattr	= generic_removexattr,
#endif
	.setattr	= ext2_setattr,
	.permission	= ext2_permission,
};
```

如果**文件类型是符号链接**，那么实际上是有两种符号链接的：快速符号链接（路径名全部存放在索引节点内）与普通符号链接（较长的路径名）。因此，有两套索引节点操作，分别存放在ext2_fast_symlink_inode_operations和ext2_symlink_inode_operations表中：

```
struct inode_operations ext2_symlink_inode_operations = {
	.readlink	= generic_readlink,
	.follow_link	= page_follow_link_light,
	.put_link	= page_put_link,
#ifdef CONFIG_EXT2_FS_XATTR
	.setxattr	= generic_setxattr,
	.getxattr	= generic_getxattr,
	.listxattr	= ext2_listxattr,
	.removexattr	= generic_removexattr,
#endif
};

struct inode_operations ext2_fast_symlink_inode_operations = {
	.readlink	= generic_readlink,
	.follow_link	= ext2_follow_link,
#ifdef CONFIG_EXT2_FS_XATTR
	.setxattr	= generic_setxattr,
	.getxattr	= generic_getxattr,
	.listxattr	= ext2_listxattr,
	.removexattr	= generic_removexattr,
#endif
};
```

如果索引节点指的是一个字符设备文件、块设备文件或管道文件，那么这种索引节点的操作部依赖于具体的文件系统，其分别位于 chrdev_inode_operations、blkdev_inode_operations和fifo_inode_operations表中。 这些数据结构在新的内核版本中已经淘汰了，我们就不去详细描述他们了。

Ext2的文件操作

针对Ext2文件系统特定的文件操作，其实现VFS文件对象的具体方法地址存放在ext2_file_operations表中：

```
const struct file_operations ext2_file_operations = {
      .llseek            = generic_file_llseek,
      .read              = generic_file_read,
      .write             = generic_file_write,
      .aio_read      = generic_file_aio_read,
      .aio_write      = generic_file_aio_write,
      .ioctl             = ext2_ioctl,
      .mmap              = generic_file_mmap,
      .open              = generic_file_open,
      .release       = ext2_release_file,
      .fsync             = ext2_sync_file,
      .readv             = generic_file_readv,
      .writev             = generic_file_writev,
      .sendfile      = generic_file_sendfile,
      .splice_read       = generic_file_splice_read,
      .splice_write      = generic_file_splice_write,
};
```

我们看到，大部分方法都有generic字符，说明这些方法不仅只被ext2文件系统使用，而且是个很通用的方法，如果你有兴趣去看看 ext3_file_operations或ext4_file_operations你会发现，他们也大量使用了这些通用的方法。Ext2最主要的读写 文件方法的read和write方法分别通过generic_file_read和generic_file_write函数实现。这两个函数我们会在以 后的博文中重点讨论他们。

Ext2索引节点分配

文件在磁盘的存储不同于程序员所看到的文件，主要表现在两个方面：块可以分散在磁盘上（尽管文件系统尽力保持块连续存放以提高访问速度），以及程序员看到的文件似乎比实际的文件大，这是因为程序可以把洞引入文件（通过lseek ()系统调用）。

从本文开始，我们将介绍Ext2文件系统如何管理磁盘空间，也就是说，如何分配和释放索引节点和数据块。有两个主要的问题必须考虑：

（1）空间管理必须**尽力避免文件碎片**，也就是说，避免文件在物理上存放于几个小的、不相邻的盘块上。文件碎片增加了对文件的连续读操作的平均时间，因为在读操作期间，磁头必须频繁地重新定位。这个问题类似于在内存管理中的“伙伴系统算法”博文中所讨论的RAM的外部碎片问题。

（2）空间管理**必须考虑效率**，也就是说，内核应该能从文件的偏移量快速地导出Ext2分区上相应的逻辑块号。为了达到此目的，内核应该尽可能地限制对磁盘上寻址表的访问次数，因为对该表的访问会极大地增加文件的平均访问时间。

创建索引节点

ext2_new_inode()函数创建Ext2磁盘的索引节点，返回相应的索引节点对象的地址（或失败时为NULL）。该函数谨慎地选择存放该新索引节点的块组；它将无联系的目录散放在不同的组，而且同时把文件存放在父目录的同一组。为了平衡普通文件数与块组中的目录数，Ext2为每一个块组引 入“债（debt）”参数。

ext2_new_inode函数有两个参数：dir，所创建索引节点父目录对应的索引节点对象的地址，新创建的索引节点必须插入到这个目录中，成 为其中的一个目录项；mode，要创建的索引节点的类型。后一个参数还包含一个MS_SYNCHRONOUS标志，该标志请求当前进程一直挂起，直到索引 节点被分配成功或失败。该函数代码如下：

```
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);
        if (!inode)
                return ERR_PTR(-ENOMEM);

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

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

        for (i = 0; i < sbi->s_groups_count; i++) {
                gdp = ext2_get_group_desc(sb, group, &bh2);
                brelse(bitmap_bh);
                bitmap_bh = read_inode_bitmap(sb, group);
                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);
                if (ino >= EXT2_INODES_PER_GROUP(sb)) {
                        /*
                         * 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;
            }
            if (ext2_set_bit_atomic(sb_bgl_lock(sbi, group),
                                    ino, bitmap_bh->b_data)) {
                /* 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;
        }

        /*
         * Scanned all blockgroups.
         */
        err = -ENOSPC;
        goto fail;
got:
        mark_buffer_dirty(bitmap_bh);
        if (sb->s_flags & MS_SYNCHRONOUS)
            sync_dirty_buffer(bitmap_bh);
        brelse(bitmap_bh);

        ino += group * EXT2_INODES_PER_GROUP(sb) + 1;
        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_mod(&sbi->s_freeinodes_counter, -1);
	if (S_ISDIR(mode))
		percpu_counter_inc(&sbi->s_dirs_counter);

	spin_lock(sb_bgl_lock(sbi, group));
	gdp->bg_free_inodes_count =
			cpu_to_le16(le16_to_cpu(gdp->bg_free_inodes_count) -
1);
	if (S_ISDIR(mode)) {
		if (sbi->s_debts[group] < 255)
			sbi->s_debts[group]++;
		gdp->bg_used_dirs_count =
			cpu_to_le16(le16_to_cpu(gdp->bg_used_dirs_count) +
1);
	} 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);
	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;
	inode->i_mtime = inode->i_atime = inode->i_ctime =
CURRENT_TIME_SEC;
	memset(ei->i_data, 0, sizeof(ei->i_data));
	ei->i_flags = EXT2_I(dir)->i_flags & ~EXT2_BTREE_FL;
	if (S_ISLNK(mode))
		ei->i_flags &= ~(EXT2_IMMUTABLE_FL|EXT2_APPEND_FL);
	/* dirsync is only applied to directories */
	if (!S_ISDIR(mode))
		ei->i_flags &= ~EXT2_DIRSYNC_FL;
	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_group = group;
        ei->i_next_alloc_block = 0;
        ei->i_next_alloc_goal = 0;
        ei->i_prealloc_block = 0;
        ei->i_prealloc_count = 0;
        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);
        insert_inode_hash(inode);

        if (DQUOT_ALLOC_INODE(inode)) {
                err = -EDQUOT;
                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;

fail_free_drop:
        DQUOT_FREE_INODE(inode);

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

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

1. 调用new_inode()分配一个新的VFS索引节点对象，并把它的i_sb字段初始化为存放在dir->i_sb中的超级块地址。然后把它追加到正在用的索引节点链表与超级块链表中：

```
struct inode *new_inode(struct super_block *sb)
{
	/* 32 bits for compatability mode stat calls */
	static unsigned int last_ino;
	struct inode * inode;

	spin_lock_prefetch(&inode_lock);

	inode = alloc_inode(sb);
	if (inode) {
		spin_lock(&inode_lock);
		inodes_stat.nr_inodes++;
		list_add(&inode->i_list, &inode_in_use);
		list_add(&inode->i_sb_list, &sb->s_inodes);
		inode->i_ino = ++last_ino;
		inode->i_state = 0;
		spin_unlock(&inode_lock);
	}
	return inode;
}

static struct inode *alloc_inode(struct super_block *sb)
{
	static const struct address_space_operations empty_aops;
	static struct inode_operations empty_iops;
	static const struct file_operations empty_fops;
	struct inode *inode;

	if (sb->s_op->alloc_inode)
		inode = sb->s_op->alloc_inode(sb);
	else
		inode = (struct inode *) kmem_cache_alloc(inode_cachep,
SLAB_KERNEL);
```

```
	if (inode) {
		struct address_space * const mapping = &inode->i_data;

		inode->i_sb = sb;
		inode->i_blkbits = sb->s_blocksize_bits;
		inode->i_flags = 0;
		atomic_set(&inode->i_count, 1);
		inode->i_op = &empty_iops;
		inode->i_fop = &empty_fops;
		inode->i_nlink = 1;
		atomic_set(&inode->i_writecount, 0);
		inode->i_size = 0;
		inode->i_blocks = 0;
		inode->i_bytes = 0;
		inode->i_generation = 0;
#ifdef CONFIG_QUOTA
		memset(&inode->i_dquot, 0, sizeof(inode->i_dquot));
#endif
		inode->i_pipe = NULL;
		inode->i_bdev = NULL;
		inode->i_cdev = NULL;
		inode->i_rdev = 0;
		inode->i_security = NULL;
		inode->dirtied_when = 0;
		if (security_inode_alloc(inode)) {
			if (inode->i_sb->s_op->destroy_inode)
				inode->i_sb->s_op->destroy_inode(inode);
			else
				kmem_cache_free(inode_cachep, (inode));
			return NULL;
		}

		mapping->a_ops = &empty_aops;
		 mapping->host = inode;
		mapping->flags = 0;
		mapping_set_gfp_mask(mapping, GFP_HIGHUSER);
		mapping->assoc_mapping = NULL;
		mapping->backing_dev_info = &default_backing_dev_info;

		/*
		 * If the block_device provides a backing_dev_info for client
		 * inodes then use that.  Otherwise the inode share the
bdev's
		 * backing_dev_info.
		 */
```

```
            if (sb->s_bdev) {
                  struct backing_dev_info *bdi;

                  bdi = sb->s_bdev->bd_inode_backing_dev_info;
                  if (!bdi)
                        bdi =
sb->s_bdev->bd_inode->i_mapping->backing_dev_info;
                  mapping->backing_dev_info = bdi;
            }
            inode->i_private = 0;
            inode->i_mapping = mapping;
      }
      return inode;
}
```

还记得sb->s_op->alloc_inode(sb)吧，对，就是“Ext2的索引节点对象”中提到的ext2_alloc_inode函数，我们就不再赘述了。注意，VFS的inode对象是嵌入在ext2_inode_info描述符中的，当执 行了ext2_alloc_inode函数以后，new_inode()函数内部就会有一个未初始化的inode结构。随后，new_inode()函数 将这个inode分别插入到以inode_in_use和sb->s_inodes为首的循环链表中。

2. 回到ext2_new_inode中，接下来：

```
      struct ext2_inode_info *ei = EXT2_I(inode);
      struct ext2_sb_info *sbi = EXT2_SB(sb);
      struct ext2_super_block *es = sbi->s_es;
      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);
```

我们看到，与分配的inode相关的磁盘索引节点映像、磁盘超级快映像和磁盘超级快对象分别由内部变量ei、sbi和es指向。随后如果新的索引节 点是一个目录，函数就调用find_group_orlov(sb, dir)为目录找到一个合适的块组（安装ext2时，如果带一个参数OLDALLOC，则会强制内核使用一种简单、老式的方式分配块组：find_group_dir(sb, dir)）。find_group_orlov函数的实现比较简单，我们只是概要地介绍一下其中的相关代码，主要描述其中的逻辑关系。为一个目录分配块组 总的策略是，尽量把一个该目录的目录项对应的普通文件的所以块放在同一个块组中。该函数执行如下试探法：

a. 以文件系统根root为父目录的目录项应该分散在各个块组。这样，函数在这些块组中去查找一个组，这个组中的空闲索引节点数的比例和空闲块数的比例比

平均 值高（avefreei = freei / ngroups，avefreeb = free_blocks / ngroups）。如果没有这样的组则跳到第c步。

b. 如果满足下列条件，嵌套目录（父目录不是文件系统根root）就应被存放到父目录块组：

- 该组没有包含太多的目录（desc->bg_used_dirs_count < best_ndir）
- 该组有足够多的空闲索引节点（desc->bg_free_inodes_count < min_inodes）
- 改组有足够多的空闲块（desc->bg_free_blocks_count) < min_blocks）
- 该组有一点小“债”。（块组的债存放在ext2_sb_info描述符的s_debts字段所指向的计数器数组中。每当一个新目录加入，债加1；每当其他类型的文件加入，债减1）

这个“债”有个最大值：max_debt = EXT2_BLOCKS_PER_GROUP(sb) / max(blocks_per_dir, BLOCK_COST)，即每个组的块数与每个目录的块数之比值。注意，这里是表示目录文件所对应的块数，不是目录项对应的普通文件，其最大值为 BLOCK_COST，即256。

如果父目录组parent_group没有满足这些条件，那么选择第一个上述满足条件的组。如果没有满足条件的组，则跳到第c步。

c. 这是一个“退一步”原则（fallback），当找不到合适的组时使用。函数从包含父目录的块组开始选择第一个满足条件的块组，这个条件是：它的空闲索引节点数比每块组空闲索引节点数的平均值大。

d. find_group_orlov函数最后返回分配给该目录的块组号（group）

3. 如果新索引节点不是个目录，则调用find_group_other(sb, dir)，在有空闲索引节点的块组中给它分配一个。该函数从包含父目录的组开始往下找。具体逻辑如下：

a. 从包含父目录dir的块组开始，执行快速的对数查找。这种算法要查找log(n)个块组，这里n是块组总数。该算法一直向前查找直到找到一个可用的块组， 具体如下：如果我们把开始的块组称为i，那么，该算法要查找的块组为i mod (n)，i+1 mod (n)，i+1+2 mod (n)，i+1+2+4 mod (n)，等等。大家是不是觉得这个东西是不是似曾相识呀，不错，就是伙伴系统算法的思想：

```
int parent_group = EXT2_I(parent)->i_block_group;
int ngroups = EXT2_SB(sb)->s_groups_count;
group = (group + parent->i_ino) % ngroups;
```

```
        for (i = 1; i < ngroups; i <<= 1) {
                group += i;
                if (group >= ngroups)
                        group -= ngroups;
                desc = ext2_get_group_desc (sb, group, &bh);
                if (desc && le16_to_cpu(desc->bg_free_inodes_count) &&
                             le16_to_cpu(desc->bg_free_blocks_count))
                        goto found;
        }
```

b.如果该算法没有找到含有空闲索引节点的块组，就从包含父目录dir的块组开始执行彻底的线性查找：

```
        group = parent_group;
        for (i = 0; i < ngroups; i++) {
                if (++group >= ngroups)
                        group = 0;
                desc = ext2_get_group_desc (sb, group, &bh);
                if (desc && le16_to_cpu(desc->bg_free_inodes_count))
                        goto found;
        }
```

4. 好啦，得到了所分配的块组号group，回到ext2_new_inode函数中，下一步要做的事是设置位图。于是调用read_inode_bitmap()得到所选块组的索引节点位图，并从中寻找第一个空闲位，这样就得到了第一个空闲磁盘索引节点号：

```
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);
        if (!desc)
                goto error_out;

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

我们看到，read_inode_bitmap函数接收超级块和组号作为参数，获得超级快的块组描述符结构，然后将其索引节点位图地址对应的那个块缓存到页高速缓存中。最后将该高速缓存描述符buffer_head返回。

5. 接下来ext2_new_inode函数要做的事，是分配磁盘索引节点：把索引节点位图中的相应位置位：
ext2_set_bit_atomic(sb_bgl_lock(sbi, group), ino, bitmap_bh->b_data)

并把含有这个位图的页高速缓存标记为脏：
mark_buffer_dirty(bitmap_bh);

此外，如果文件系统安装时指定了MS_SYNCHRONOUS标志，则调用sync_dirty_buffer(bitmap_bh)开始I/O写操作并等待，直到写操作终止。

brelse(bitmap_bh)，递减bitmap_bh的引用数。

6. 减小ext2_sb_info数据结构中的s_freeinodes_counter字段；而且如果新索引节点是目录，则增大ext2_sb_info数据结构的s_dirs counter字段。

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

7. 减小组描述符的bg_free_inodes_count字段。如果新的索引节点是一个目录，则增加bg_used_dirs_count字段，并把含有这个组描述符的缓冲区标记为脏：

```
gdp = ext2_get_group_desc(sb, group, &bh2);
gdp->bg_free_inodes_count =
cpu_to_le16(le16_to_cpu(gdp->bg_free_inodes_count) - 1);
if (S_ISDIR(mode)) {
      if (sbi->s_debts[group] < 255)
            sbi->s_debts[group]++;
      gdp->bg_used_dirs_count =
            cpu_to_le16(le16_to_cpu(gdp->bg_used_dirs_count) +
1);
} else {
      if (sbi->s_debts[group])
            sbi->s_debts[group]--;
}
sb->s_dirt = 1;
mark_buffer_dirty(bh2);
inode->i_uid = current->fsuid;
```

注意，依据索引节点指向的是普通文件或目录，相应增减超级块内s_debts数组中的组计数器。

8. 初始化这个索引节点对象的字段。特别是，设置索引节点号i_no，并把xtime.tv_sec的值拷贝到i_atime、i_mtime及i_ctime。把这个块组的索引赋给

ext2_inode_info结构的i_block_group字段：

```
ei->i_block_group = group;
```

9. 初始化这个索引节点对象的访问控制列表（ACL）：

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

10. 将新索引节点对象插入散列表inode_hashtable（insert_inode_hash(inode)，其数据结构详情请参考《把Linux中的VFS对象串联起来》），调用mark_inode_dirty()把该索引节点对象移进超级块脏索引节点链表：

```
static inline void insert_inode_hash(struct inode *inode) {
      __insert_inode_hash(inode, inode->i_ino);
}
void __insert_inode_hash(struct inode *inode, unsigned long hashval)
{
      struct hlist_head *head = inode_hashtable + hash(inode->i_sb,
hashval);
      spin_lock(&inode_lock);
      hlist_add_head(&inode->i_hash, head);
      spin_unlock(&inode_lock);
}
static inline void hlist_add_head(struct hlist_node *n, struct
hlist_head *h)
{
      struct hlist_node *first = h->first;
      n->next = first;
      if (first)
            first->pprev = &n->next;
      h->first = n;
      n->pprev = &h->first;
}

static inline void mark_inode_dirty(struct inode *inode)
{
      __mark_inode_dirty(inode, I_DIRTY);
}
void __mark_inode_dirty(struct inode *inode, int flags)
{
      struct super_block *sb = inode->i_sb;

      /*
       * Don't do this for I_DIRTY_PAGES - that doesn't actually
       * dirty the inode itself
```

```
 */
if (flags & (I_DIRTY_SYNC | I_DIRTY_DATASYNC)) {
      if (sb->s_op->dirty_inode)
            sb->s_op->dirty_inode(inode);
}

/*
 * make sure that changes are seen by all cpus before we test i_state
 * -- mikulas
 */
smp_mb();

/* avoid the locking if we can */
if ((inode->i_state & flags) == flags)
      return;

if (unlikely(block_dump)) {
      struct dentry *dentry = NULL;
      const char *name = "?";

      if (!list_empty(&inode->i_dentry)) {
            dentry = list_entry(inode->i_dentry.next,
                              struct dentry, d_alias);
            if (dentry && dentry->d_name.name)
                  name = (const char *) dentry->d_name.name;
      }

      if (inode->i_ino || strcmp(inode->i_sb->s_id, "bdev"))
            printk(KERN_DEBUG
                   "%s(%d): dirtied inode %lu (%s) on %s\n",
                   current->comm, current->pid, inode->i_ino,
                   name, inode->i_sb->s_id);
}

spin_lock(&inode_lock);
if ((inode->i_state & flags) != flags) {
      const int was_dirty = inode->i_state & I_DIRTY;

      inode->i_state |= flags;

      /*
       * If the inode is locked, just update its dirty state.
       * The unlocker will place the inode on the appropriate
       * superblock list, based upon its state.
       */
```

```
            if (inode->i_state & I_LOCK)
                  goto out;

            /*
             * Only add valid (hashed) inodes to the superblock's
             * dirty list.  Add blockdev inodes as well.
             */
            if (!S_ISBLK(inode->i_mode)) {
                  if (hlist_unhashed(&inode->i_hash))
                        goto out;
            }
            if (inode->i_state & (I_FREEING|I_CLEAR))
                  goto out;

            /*
             * If the inode was already on s_dirty or s_io, don't
             * reposition it (that would break s_dirty time-ordering).
             */
            if (!was_dirty) {
                  inode->dirtied_when = jiffies;
                  list_move(&inode->i_list, &sb->s_dirty);
            }
      }
out:
      spin_unlock(&inode_lock);
}
```

11. 调用ext2_preread_inode()从磁盘读入包含该索引节点的块,将它存入页高速缓存。进行这种预读是因为最近创建的索引节点可能会被很快写入。

```
static void ext2_preread_inode(struct inode *inode)
{
      unsigned long block_group;
      unsigned long offset;
      unsigned long block;
      struct buffer_head *bh;
      struct ext2_group_desc * gdp;
      struct backing_dev_info *bdi;

      bdi = inode->i_mapping->backing_dev_info;
      if (bdi_read_congested(bdi))
            return;
      if (bdi_write_congested(bdi))
            return;
```

```
	block_group = (inode->i_ino - 1) /
EXT2_INODES_PER_GROUP(inode->i_sb);
	gdp = ext2_get_group_desc(inode->i_sb, block_group, &bh);
	if (gdp == NULL)
		return;

	/*
	 * Figure out the offset within the block group inode table
	 */
	offset = ((inode->i_ino - 1) % EXT2_INODES_PER_GROUP(inode->i_sb))
*
				EXT2_INODE_SIZE(inode->i_sb);
	block = le32_to_cpu(gdp->bg_inode_table) +
				(offset >> EXT2_BLOCK_SIZE_BITS(inode->i_sb));
	sb_breadahead(inode->i_sb, block);
}
static inline void
sb_breadahead(struct super_block *sb, sector_t block)
{
	__breadahead(sb->s_bdev, block, sb->s_blocksize);
}
```

12. 返回新索引节点对象inode的地址。

删除索引节点

内核调用ext2_free_inode()函数删除一个磁盘索引节点，把磁盘索引节点表示为索引节点对象，其地址作为参数来传递。内核在进行一系 列的清除操作（包括清除内部数据结构和文件中的数据）之后调用这个函数。具体来说，它在下列操作完成之后才执行：索引节点对象已经从散列表中删除，指向这 个索引节点的最后一个硬链接已经从适当的目录中删除，文件的长度截为0以回收它的所有数据块。

函数执行下列操作：

1. 调用clear_inode()，它依次执行如下步骤：

a.删除与索引节点关联的“间接”脏缓冲区。它们都存放在一个链表中，该链表的首部在address_space对象inode->i_data的private_list字段。

b.如果索引节点的I_LOCK标志置位，则说明索引节点中的某些缓冲区正处于I/O数据传送中；于是，函数挂起当前进程，直到这些1/0数据传送结束。

c.调用超级块对象的clear_inode方法（如果已定义），但Ext2文件系统没定义这个方法。

d.如果索引节点指向一个设备文件，则从设备的索引节点链表中删除索引节对象，这个链表要么在cdev字符设备描述符的cdev字段，要么在block_device块设备描述符的bd_inodes段。

e.把索引节点的状态置为I_CLEAR（表示索引节点对象的内容不再有意义）。

2. 从每个块组的索引节点号和索引节点数计算包含这个磁盘索引节点的块组的索引。

3. 调用read_inode_bitmap()得到索引节点位图。

4. 增加组描述符的bg_free_inodes_count字段。如果删除的索引节点是一个目录，那么也要减小bg_used_dirs_count字段。把这个组描述符所在的缓冲区标记为脏。

5. 如果删除的索引节点是一个目录，就减小ext2_sb_info结构的s_dirs_counter字段，把超级块的s_dirt标志置1，并把它所在的缓冲区标记为脏。

6. 清除索引节点位图中这个磁盘索引节点对应的位，并把包含这个位图的缓冲区标记为脏。此外，如果文件系统以MS_SYNCHRONIZE标志安装，则sync_dirty_buffer()并等待，直到在位图缓冲区上的写操作终止。

Ext2数据块分配

跟索引节点一样，Ext2也对磁盘数据块进行分配与释放。在详细分析相关代码之前，先引出两个重要的预备，一个是数据块寻址，一个是文件的洞

数据块寻址

每个非空的普通文件都由一组数据块组成。这些块或者由文件内的相对位置（它们的文件块号）来标识，或者由磁盘分区内的位置（它们的逻辑块号）来标识。

从文件内的偏移量 f 导出相应数据块的逻辑块号需要两个步骤：

1. 从**偏移量f**导出**文件的块号**，即在偏移量f处的字符所在的块索引。

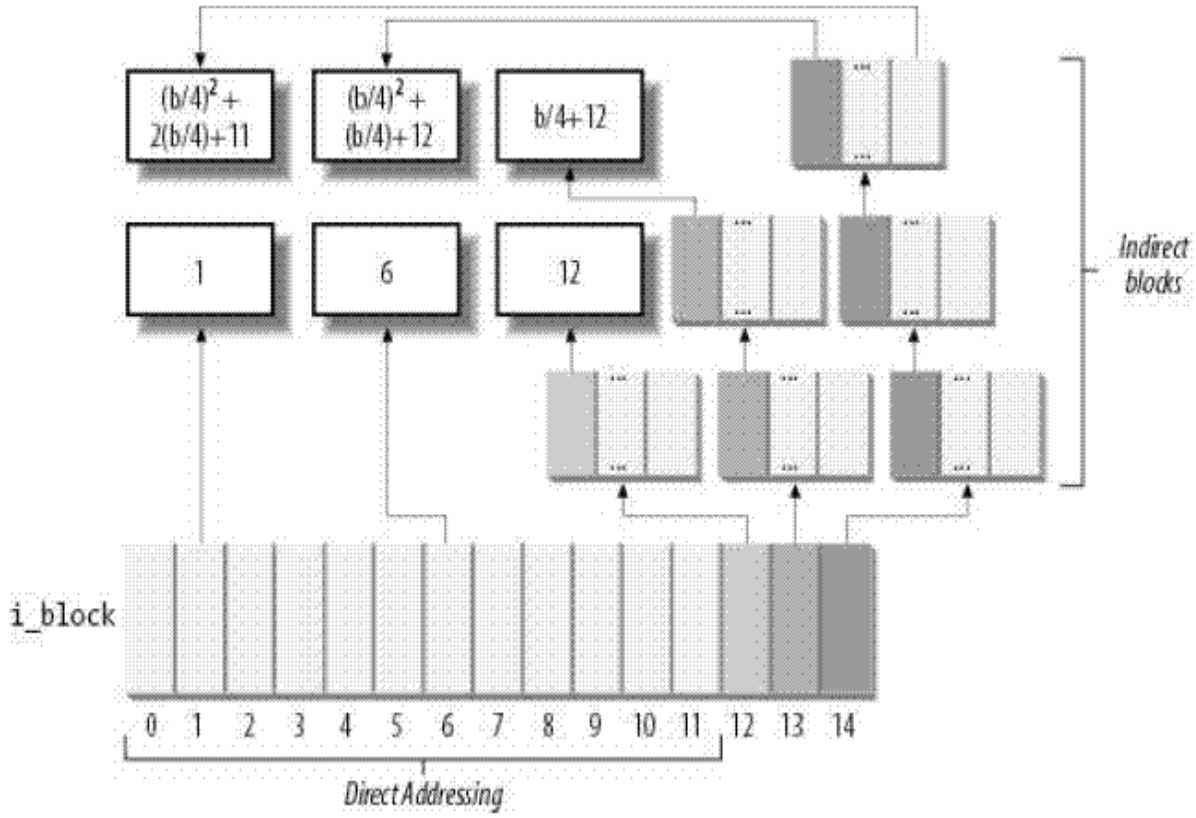
2. 把文件的块号转化为**相应的逻辑块号**。

因为Unix文件不包含任何控制字符，因此，导出文件的第f个字符所在的文件块号当容易的，只是用f除以文件系统块的大小，并取整即可。

例如，让我们假定块的大小为4KB。如果f小于4096，那么这个字符就在文件的第一数据块中，其文件的块号为0。如果f等于或大于4096而小于8192，则这个字符就在文件块号为1的数据块中，以此类推。

得到了文件的块号是第一步。但是，由于Ext2文件的数据块在磁盘上不必是相邻的,因此把文件的块号转化为相应的逻辑块号可不是这么直截了当的了。因此，Ext2文件系统必须提供一种方法,用这种方法可以在磁盘上建立每个文件块号与相应逻辑块号之间的关系。在索引节点内部部分实现了这种映射（回到了 AT&T Unix的早期版本）。这种映射也涉及一些包含额外指针的专用块，这些块用来处理大型文件的索引节点的扩展。

ext2磁盘索引节点ext2_inode的i_block字段是一个有EXT2_N_BLOCKS个元素且包含逻辑块号的数组。在下面的讨论中， 我们假定EXT2_N_BLOCKS的默认值为15（实际上到2.6.18这个值都一直是15）。如图所示，这个数组表示一个大型数据结构的初始化部分。



正如从图中所看到的，数组的15个元素有4种不同的类型：

- 最初的12个元素产生的逻辑块号与文件最初的12个块对应，即对应的文件块号从0 - 11。

- 下标12中的元素包含一个块的逻辑块号（叫做间接块），这个块中存放着一个表示逻辑块号的二级数组。这个数组的元素对应的文件块号从12 到 b/4+11，这里b是文件系统的块大小（每个逻辑块号占4个字节，因此我们在式子中用4作除数，如果块大小是4096，则该数组对应文件块号从12到 1035）。因此，内核为了查找指向一个块的指针必须先访问这个元素，然后，在这个块中找到另一个指向最终块（包含文件内容）的指针。注：可表示1024个块.

- 下标13中的元素包含一个间接块的逻辑块号，而这个块包含逻辑块号的一个二级数组，这个二级数组的数组项依次指向三级数组，这个三级数组存放的才是文件块 号对应的逻辑块号，范围从b/4+12到(b/4)^2+(b/4)+11。如果块大小是4096，则范围是从1036到1049611。注：可表示1M个块.

- 最后，下标14中的元素使用三级间接索引，第四级数组中存放的才是文件块号对应的逻辑块号，范围从(b/4)^2+(b/4)+12到(b/4)^3+(b/4)^2+(b/4)+11。注：可表示1G个块.

在图中，块内的数字表示相应的文件块号。箭头（表示存放在数组元素中的逻辑块号）指示了内核如何通过间接块找到包含文件实际内容的块。

注意这种机制是如何支持小文件的。如果文件需要的数据块小于12，那么两次磁盘访问就可以检索到任何数据：一次是读磁盘索引节点i_block数组 的一个元素，另一次是读所需要的数据块。对于大文件来说，可能需要三四次的磁盘访问才能找到需要的块。实际上，这是一种最坏的估计，因为目录项、索引节 点、页高速缓存都有助于极大地减少实际访问磁盘的次数。

还要注意文件系统的块大小是如何影响寻址机制的，因为大的块允许Ext2把更多的逻辑块号存放在一个单独的块中。例如，如果块的大小是1024字节，并且文件包含的数据最多为268KB，那么，通过直接映射可以访问文件最初的12KB数据，通过简单的间接映射可以访问剩的13KB到268KB的数 据。大于2GB的大型文件通过指定O_LARGEFILE打开标志必须在32位体系结构上进行打开。

文件的洞

文件的洞（file hole）是普通文件的一部分，它是一些空字符但没有存放在磁盘的任何数据块中。洞是Unix文件一直存在的一个特点。例如，下列的Unix命令创建了第一个字节是洞的文件：

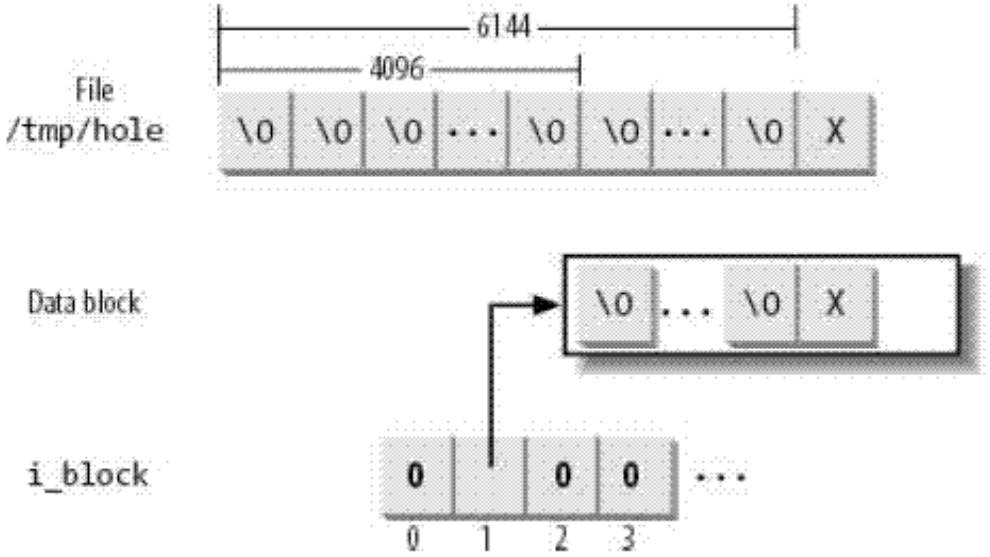
```
[root@localhost]# echo -n "X" | dd of=/tmp/hole bs=1024 seek=6
```

现在，/tmp/hole有6145个字符（6144个空字符加一个X字符），然而，这个文件在磁盘上只占一个数据块。

引入文件的洞是为了避免磁盘空间的浪费。它们因此被广泛地用在数据库应用中，更一般地说，用于在文件上进行散列的所有应用。

文件洞在Ext2中的实现是基于动态数据块的分配的：只有当进程需要向一个块写数据时，才真正把这个块分配给文件。每个索引节点的i_size字段定义程序所看到的文件大小，包括洞，而i_blocks字段存放分配给文件有效的数据块数（以512字节为单位）。

在前面dd命令的例子中，假定/tmp/hole文件创建在块大小为4096的Ext2分区上。其相应磁盘索引节点的**i_size字段**存放的数为 6145，而**i_blocks字段**存放的数为8（因为每4096字节的逻辑块包含8个512字节的物理块）。i_block数组的第二个元素（对应块的文 件块号为1）存放已分配块的逻辑块号，而数组中的其他元素都为空（参看下图）。



分配数据块

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。如果块不存在，该函数就自动为文件分配块。 请记住，每当内核在Ext2普通文件上执行读或写操作时就调用这个函数；显然，这个函数只在页高速缓存内没有相应的块时才被调用。

ext2_get_block()函数处理在刚才“数据块寻址”描述的数据结构，并在必要时调用ext2_alloc_block()函数在Ext2分区真正搜索一个空闲块。如果需要，该函数还为间接寻址分配相应的块。

为了减少文件的碎片，Ext2文件系统尽力在已分配给文件的最后一个块附近找一个新块分配给该文件。如果失败，Ext2文件系统又在包含这个文件索引节点的块组中搜寻一个新的块。如果还是失败，作为最后一个办法，可以从其他一个块组中获得空闲块。

Ext2文件系统使用数据块的预分配策略。文件并不仅仅获得所需要的块，而是获得一组多达8个邻接的块。ext2_inode_info结构的 i_prealloc_count字段存放预分配给某一文件但还没有使用的数据块的数量，而i_prealloc_block字段存放下一次要使用的预分 配块的逻辑块号。当下列情况发生时，释放预分配而一直没有使用的块：当文件被关闭时，当文件被缩短时，或者当一个写操作相对于引发块预分配的写操作不是顺 序的时。

ext2_alloc_block()函数接收的参数为指向索引节点对象的指针、目标（goal）和存放错误码的变量地址。目标是一个逻辑块号，表示新块的首选位置：

```
static unsigned long ext2_alloc_block (struct inode * inode, unsigned long goal, int *err)
```

```
{
#ifdef EXT2FS_DEBUG
      static unsigned long alloc_hits, alloc_attempts;
#endif
      unsigned long result;


#ifdef EXT2_PREALLOCATE
      struct ext2_inode_info *ei = EXT2_I(inode);
      write_lock(&ei->i_meta_lock);
      if (ei->i_prealloc_count &&
          (goal == ei->i_prealloc_block || goal + 1 ==
ei->i_prealloc_block))
      {
            result = ei->i_prealloc_block++;
            ei->i_prealloc_count--;
            write_unlock(&ei->i_meta_lock);
            ext2_debug ("preallocation hit (%lu/%lu).\n",
                        ++alloc_hits, ++alloc_attempts);
      } else {
            write_unlock(&ei->i_meta_lock);
            ext2_discard_prealloc (inode);
            ext2_debug ("preallocation miss (%lu/%lu).\n",
                        alloc_hits, ++alloc_attempts);
            if (S_ISREG(inode->i_mode)) /* 如果是普通文件 */
                  result = ext2_new_block (inode, goal,
                         &ei->i_prealloc_count,
                         &ei->i_prealloc_block, err);
            else  /* 如果是目录或符号链接 */
                  result = ext2_new_block(inode, goal, NULL, NULL, err);
      }
#else
      result = ext2_new_block (inode, goal, 0, 0, err);
#endif
      return result;
}
```

代码很容易看懂，如果先前有预分配，则直接返回ei->i_prealloc_block++，没有，则丢弃所有剩余的预分配块ext2_discard_prealloc(inode)，并调用ext2_new_block函数分配一个块：

```
unsigned long ext2_new_block(struct inode *inode, unsigned long goal,
                  u32 *prealloc_count, u32 *prealloc_block, int *err)
{
```

```
	struct buffer_head *bitmap_bh = NULL;
	struct buffer_head *gdp_bh;	/* bh2 */
	struct ext2_group_desc *desc;
	int group_no;			/* i */
	int ret_block;			/* j */
	int group_idx;			/* k */
	unsigned long target_block;	/* tmp */
	unsigned long block = 0;
	struct super_block *sb = inode->i_sb;
	struct ext2_sb_info *sbi = EXT2_SB(sb);
	struct ext2_super_block *es = sbi->s_es;
	unsigned group_size = EXT2_BLOCKS_PER_GROUP(sb);
	unsigned prealloc_goal = es->s_prealloc_blocks;
	unsigned group_alloc = 0, es_alloc, dq_alloc;
	int nr_scanned_groups;

	if (!prealloc_goal--)
		prealloc_goal = EXT2_DEFAULT_PREALLOC_BLOCKS - 1;
	if (!prealloc_count || *prealloc_count)
		prealloc_goal = 0;

	if (DQUOT_ALLOC_BLOCK(inode, 1)) {
		*err = -EDQUOT;
		goto out;
	}

	while (prealloc_goal && DQUOT_PREALLOC_BLOCK(inode,
prealloc_goal))
		prealloc_goal--;

	dq_alloc = prealloc_goal + 1;
	es_alloc = reserve_blocks(sb, dq_alloc);
	if (!es_alloc) {
		*err = -ENOSPC;
		goto out_dquot;
	}

	ext2_debug ("goal=%lu.\n", goal);

	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);
	group_no = (goal - le32_to_cpu(es->s_first_data_block)) /
group_size;
	desc = ext2_get_group_desc (sb, group_no, &gdp_bh);
```

```
	if (!desc) {
		/*
		 * gdp_bh may still be uninitialised.  But
group_release_blocks
		 * will not touch it because group_alloc is zero.
		 */
		goto io_error;
	}

	group_alloc = group_reserve_blocks(sbi, group_no, desc,
				gdp_bh, es_alloc);
	if (group_alloc) {
		ret_block = ((goal - le32_to_cpu(es->s_first_data_block)) %
				group_size);
		brelse(bitmap_bh);
		bitmap_bh = read_block_bitmap(sb, group_no);
		if (!bitmap_bh)
			goto io_error;

		ext2_debug("goal is at %d:%d.\n", group_no, ret_block);

		ret_block = grab_block(sb_bgl_lock(sbi, group_no),
				bitmap_bh->b_data, group_size, ret_block);
		if (ret_block >= 0)
			goto got_block;
		group_release_blocks(sb, group_no, desc, gdp_bh,
group_alloc);
		group_alloc = 0;
	}

	ext2_debug ("Bit not found in block group %d.\n", group_no);

	/*
	 * Now search the rest of the groups.  We assume that
	 * i and desc correctly point to the last group visited.
	 */
	nr_scanned_groups = 0;
retry:
	for (group_idx = 0; !group_alloc &&
			group_idx < sbi->s_groups_count; group_idx++) {
		group_no++;
		if (group_no >= sbi->s_groups_count)
			group_no = 0;
		desc = ext2_get_group_desc(sb, group_no, &gdp_bh);
		if (!desc)
```

```
                goto io_error;
            group_alloc = group_reserve_blocks(sbi, group_no, desc,
                                    gdp_bh, es_alloc);
        }
        if (!group_alloc) {
            *err = -ENOSPC;
            goto out_release;
        }
        brelse(bitmap_bh);
        bitmap_bh = read_block_bitmap(sb, group_no);
        if (!bitmap_bh)
            goto io_error;

        ret_block = grab_block(sb_bgl_lock(sbi, group_no),
bitmap_bh->b_data,
                        group_size, 0);
        if (ret_block < 0) {
            /*
             * If a free block counter is corrupted we can loop inifintely.
             * Detect that here.
             */
            nr_scanned_groups++;
            if (nr_scanned_groups > 2 * sbi->s_groups_count) {
                ext2_error(sb, "ext2_new_block",
                        "corrupted free blocks counters");
                goto io_error;
            }
            /*
             * Someone else grabbed the last free block in this blockgroup
             * before us.  Retry the scan.
             */
            group_release_blocks(sb, group_no, desc, gdp_bh,
group_alloc);
            group_alloc = 0;
            goto retry;
        }

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

        target_block = ret_block + group_no * group_size +
                le32_to_cpu(es->s_first_data_block);
```

```
	if (target_block == le32_to_cpu(desc->bg_block_bitmap) ||
	    target_block == le32_to_cpu(desc->bg_inode_bitmap) ||
	    in_range(target_block, le32_to_cpu(desc->bg_inode_table),
		     sbi->s_itb_per_group))
		ext2_error (sb, "ext2_new_block",
			    "Allocating block in system zone - "
			    "block = %lu", target_block);

	if (target_block >= le32_to_cpu(es->s_blocks_count)) {
		ext2_error (sb, "ext2_new_block",
			    "block(%d) >= blocks count(%d) - "
			    "block_group = %d, es == %p ", ret_block,
			le32_to_cpu(es->s_blocks_count), group_no, es);
		goto io_error;
	}
	block = target_block;

	/* OK, we _had_ allocated something */
	ext2_debug("found bit %d\n", ret_block);

	dq_alloc--;
	es_alloc--;
	group_alloc--;

	/*
	 * Do block preallocation now if required.
	 */
	write_lock(&EXT2_I(inode)->i_meta_lock);
	if (group_alloc && !*prealloc_count) {
		unsigned n;

		for (n = 0; n < group_alloc && ++ret_block < group_size; n++)
{
			if (ext2_set_bit_atomic(sb_bgl_lock(sbi, group_no),
						ret_block,
						(void*) bitmap_bh->b_data))
				break;
		}
		*prealloc_block = block + 1;
		*prealloc_count = n;
		es_alloc -= n;
		dq_alloc -= n;
		group_alloc -= n;
	}
	write_unlock(&EXT2_I(inode)->i_meta_lock);
```

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

        ext2_debug ("allocating block %d. ", block);

        *err = 0;
out_release:
        group_release_blocks(sb, group_no, desc, gdp_bh, group_alloc);
        release_blocks(sb, es_alloc);
out_dquot:
        DQUOT_FREE_BLOCK(inode, dq_alloc);
out:
        brelse(bitmap_bh);
        return block;

io_error:
        *err = -EIO;
        goto out_release;
}
```

ext2_new_block()函数用下列策略在Ext2分区内搜寻一个空闲块：

1. 如果传递给ext2_alloc_block()的首选块（目标块）是空闲的，就分配它。

2. 如果目标为忙，就检查首选块后的其余块之中是否有空闲的块。

3. 如果在首选块附近没有找到空闲块，就从包含目标的块组开始，查找所有的块组，对每个块组有：
    a. 寻找至少有8个相邻空闲块的一个组块。
    b. 如果没有找到这样的一组块，就寻找一个单独的空闲块。

下面我们就来详细分析这个函数，其接收的参数为：
- inode：指向被分配块的文件的索引节点
- goal：由ext2_alloc_block传递过来的目标块号
- prealloc_count：指向打算预分配块的计数器的指针
- prealloc_block：指向预分配的第一个块的位置
- err：存放错误码的变量地址

只要找到一个空闲块,搜索就结束,返回该块的块号。在结束前,ext2_new_block()函数还尽力在找到的空闲块附近的块中找8个空闲块进 行预分配,并把磁盘索引节点的i_prealloc_block和i_prealloc_count字段置为适当的块位置及块数。函数执行以下步骤：

1. 首先初始化一些内部变量：

```
struct buffer_head *bitmap_bh = NULL;
struct buffer_head *gdp_bh;      /* bh2 */
struct ext2_group_desc *desc;
……
unsigned long block = 0;
struct super_block *sb = inode->i_sb;
struct ext2_sb_info *sbi = EXT2_SB(sb);
struct ext2_super_block *es = sbi->s_es;
unsigned group_size = EXT2_BLOCKS_PER_GROUP(sb);
unsigned prealloc_goal = es->s_prealloc_blocks;
unsigned group_alloc = 0, es_alloc, dq_alloc;
```

这些内部变量各有各的含义，其中，bitmap_bh是位图的缓存；gdp_bh是块组缓存;block是当前搜索到的块号;sb是VFS 超级快结构,由inode的i_sb字段得出；sbi是磁盘超级块的内存对象描述符，由VFS超级快的s_fs_info字段得到；es是磁盘超级快对 象，由超级快内存对象描述符的s_es字段得到；group_size是磁盘块组的以块为单位的大小，由超级快的s_blocks_per_group字 段得到；

prealloc_goal是已预分配的块数,由磁盘超级快对象的s_prealloc_blocks字段得到；group_alloc表示同一个组分配块的数量。

2. 如果prealloc_goal减1为0了，则说明预分配的块已经用完，则：

```
if (!prealloc_goal--)
      prealloc_goal = EXT2_DEFAULT_PREALLOC_BLOCKS - 1;
if (!prealloc_count || *prealloc_count)
      prealloc_goal = 0;
```

其中EXT2_DEFAULT_PREALLOC_BLOCKS为8，也就是需要重新预分配8个块。当然，如果传递进来的参数prealloc_count为空或者是0，则说明不是普通文件，或者没有启动预分配机制，则prealloc_goal设置为0 。

3. 对配额进行检查，分配的目标块超过了用户配额，则将出错对象err设置为-EDQUOT;如果预分配超过了用户配额,则将预分配数量减至配额以内;检查完 毕后，将预分配数量赋给dq_alloc内部变量，再执行reserve_blocks函数检查预分配的块dq_alloc是否到达了保留块，如果是则减 去所处保留块的数量。如果得到的结果es_alloc为0了，则将出错对象err设置为-ENOSPC，表示no space：

```
if (DQUOT_ALLOC_BLOCK(inode, 1)) {
      *err = -EDQUOT;
      goto out;
}
while (prealloc_goal && DQUOT_PREALLOC_BLOCK(inode,
prealloc_goal))
```

```
            prealloc_goal--;
      dq_alloc = prealloc_goal + 1;
      es_alloc = reserve_blocks(sb, dq_alloc);
      if (!es_alloc) {
            *err = -ENOSPC;
            goto out_dquot;
      }
```

4. 如果目标块小于1（es->s_first_data_block总为1，查看“Ext2磁盘数据结构”博文），或者大于整个文件系统所有块的大小，则将goal设置为1。

```
      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);
```

5. 得到goal所对应的块组号，并根据块组号获得对应的组描述符：

```
      group_no = (goal - le32_to_cpu(es->s_first_data_block)) /
group_size;
      desc = ext2_get_group_desc (sb, group_no, &gdp_bh);
```

这里面的ext2_get_group_desc接收三个参数，sb是VFS超级快，group_no是块组号，&gdp_bh是该块组对应的页高速缓存：

```
struct ext2_group_desc * ext2_get_group_desc(struct super_block * sb,
                                    unsigned int block_group,
                                    struct buffer_head ** bh)
{
      unsigned long group_desc;
      unsigned long offset;
      struct ext2_group_desc * desc;
      struct ext2_sb_info *sbi = EXT2_SB(sb);
……

      group_desc = block_group >> EXT2_DESC_PER_BLOCK_BITS(sb);
      offset = block_group & (EXT2_DESC_PER_BLOCK(sb) - 1);
      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;
}
```

函数里面的group_desc内部变量通过block_group参数获得组描述符数组的位置下标，ext2磁盘组描述符 ext2_group_desc缓存于sbi->s_group_desc[group_desc]->b_data的某个位置，因为总是缓 存的，参考“Ext2的索引节点对象”最后那个表。

6. desc指向了组描述符以后，事情就好办了。执行 group_reserve_blocks(sbi, group_no, desc, gdp_bh, es_alloc)查看goal对应的那个组内是否有连续es_alloc个空闲的块，当然，组内如果一个空闲的块都没有，就返回0。如果空闲的块小于 es_alloc，就返回可以分配的块数，并修改组描述符的空闲块数，然后把组描述符对应的缓存标记为脏：

```
static int group_reserve_blocks(struct ext2_sb_info *sbi, int group_no,
      struct ext2_group_desc *desc, struct buffer_head *bh, int count)
{
      unsigned free_blocks;

      if (!desc->bg_free_blocks_count)
            return 0;

      spin_lock(sb_bgl_lock(sbi, group_no));
      free_blocks = le16_to_cpu(desc->bg_free_blocks_count);
      if (free_blocks < count)
            count = free_blocks;
      desc->bg_free_blocks_count = cpu_to_le16(free_blocks - count);
      spin_unlock(sb_bgl_lock(sbi, group_no));
      mark_buffer_dirty(bh);
      return count;
}
```

7. 好了，group_alloc局部变量就等于group_reserve_blocks的返回值。此时此刻，group_alloc为goal期望的那个 块可分配的数量。那么接下来就要做一个判断，如果group_alloc大于0，则说明首选块是空闲的，就分配它：

```
      if (group_alloc) {
            ret_block = ((goal - le32_to_cpu(es->s_first_data_block)) %
                              group_size);
            bitmap_bh = read_block_bitmap(sb, group_no);
            if (!bitmap_bh)
                  goto io_error;

            ret_block = grab_block(sb_bgl_lock(sbi, group_no),
                        bitmap_bh->b_data, group_size, ret_block);
```

```
            if (ret_block >= 0)
                  goto got_block;
            group_release_blocks(sb, group_no, desc, gdp_bh,
group_alloc);
            group_alloc = 0;
      }
```

read_block_bitmap读取超级块sb对应块组group_no的那个位图，并把这个位图动态缓存到bitmap_bh页高速缓存中：

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

      desc = ext2_get_group_desc (sb, block_group, NULL);
      if (!desc)
            goto error_out;
      bh = sb_bread(sb, le32_to_cpu(desc->bg_block_bitmap));
      if (!bh)
            ext2_error (sb, "read_block_bitmap",
                        "Cannot read block bitmap - "
                        "block_group = %d, block_bitmap = %u",
                        block_group,
le32_to_cpu(desc->bg_block_bitmap));
error_out:
      return bh;
}
```

随后调用grab_block(sb_bgl_lock(sbi, group_no), bitmap_bh->b_data, group_size, ret_block)检查一下goal所对应的那个数据块位图对应的位是否为0，如果是，则把它的块号赋给相对位置ret_block；如果不是，则ext2_find_next_zero_bit在组内分配一个空闲块的块号到ret_block。当然，如果ret_block大于等于0，则说明这个 组内有空闲块，则跳到got_block；如果小于0，就说明这个组的块已经分配完了，那么就把刚才给组描述符增加的那些值减回来，并把 group_alloc重新设置为0。

8. 如果group_alloc为0，则说明组内没有goal期望的那个块可分配的数量那么多的块，就到retry程序段，到其他组去找es_alloc个空闲块，具体代码跟前边一样。

9. 如果得到这个块了，那么ret_block就是这个连续块的第一个块的相对组头的位置，到got_block程序段，此时此刻，group_no是该快所在的块组号，随后：

```
target_block = ret_block + group_no * group_size +
le32_to_cpu(es->s_first_data_block);
```

target_block就是要分配的实际逻辑块号（相对于目标块号）。got_block程序段随后对这个逻辑块号进行一系列检查，包括这个块是否已经存放了索引节点、位图、组描述符等内容了。当然，肯定不会出现这些问题的，因为这些都是系统bug。

10. 设置位图（注意，是一个组内的预分配连续块都设置），并把bitmap_bh标记为脏。

```
      write_lock(&EXT2_I(inode)->i_meta_lock);
      if (group_alloc && !*prealloc_count) {
            unsigned n;

            for (n = 0; n < group_alloc && ++ret_block < group_size; n++)
{
                  if (ext2_set_bit_atomic(sb_bgl_lock(sbi, group_no),
                                    ret_block,
                                    (void*) bitmap_bh->b_data))
                        break;
            }
            *prealloc_block = block + 1;
            *prealloc_count = n;
            es_alloc -= n;
            dq_alloc -= n;
            group_alloc -= n;
      }
      write_unlock(&EXT2_I(inode)->i_meta_lock);
      mark_buffer_dirty(bitmap_bh);
```

11. 最后返回这个物理块：

```
      brelse(bitmap_bh);
      block = target_block;
      return block
```

关于linux如何把磁盘上的数据缓存到RAM中参考linux磁盘高速缓存