第九章　Ext2 文件系统

Ext2（第二扩充文件系统）是一种功能强大、易扩充、性能上进行了全面的优化的文件系统，也是当前 Linux 文件系统实际上的标准。

Linux 的第一个文件系统是 Minix，它原本是为 Minix 这个操作系统所使用的，Linus Torvalds 写 Linux 之前，学习的就是 Tanenbaum 所写的《Operating Systems Design And Implementation》，该书以 Tanenbaum 自己写的 Minix 系统为例，所以 Linus 就将 Minix 文件系统改写后用于 Linux，但这个文件系统有几个主要的缺限：

- 磁盘分区大小必须小于 64MB；
- 必须使用 14 个字符定长的文件名；
- 难于扩展。

在 VFS 被加入内核后，1992 年 4 月，第一个专门为 Linux 所写的文件系统 Ext（扩充文件系统）被加入了 0.96c 这个版本。Ext 使 Minix 的缺陷得以改进，一是它最大可支持 2GB 的磁盘分区，二是其文件名最长可达 255 个字符。但它仍有自己的缺陷：它使用链表管理未分配的数据块和节点，这样当文件系统投入使用后，链表变得杂乱无序，文件系统中会产生很多碎片。

1993 年，Remy Card 对 Ext 做出了改进，写成了 Ext2。Ext2 有如下几方面的特点。

- 它的节点中使用了 15 个数据块指针，这样它最大可支持 4TB 的磁盘分区。
- 它使用变长的目录项，这样既可以不浪费磁盘空间，又能支持最长 255 个字符的文件名。
- 使用位图来管理数据块和节点的使用情况，解决了 Ext 出现的问题。
- 最重要的一点是，它在磁盘上的布局做了改进，即使用了块组的概念，从而使数据的读和写更快、更有效，也便系统变得更安全可靠。
- 易于扩展。

9.1　基本概念

在上一章中，我们把 Ext2、Minix、Ext 等实际可使用的文件系统称为具体文件系统。具体文件系统管理的是一个逻辑空间，这个逻辑空间就像一个大的数组，数组的每个元素就是文件系统操作的基本单位——逻辑块，逻辑块是从 0 开始编号的，而且，逻辑块是连续的。与逻辑块相对的是物理块，物理块是数据在磁盘上的存取单位，也就是每进行一次 I/O 操作，最小传输的数据大小。我们知道数据是存储在磁盘的扇区中的，那么扇区是不是物理块呢？或者物理块是多大呢？这涉及到文件系统效率的问题。

如果物理块定的比较大，比如一个柱面大小，这时，即使是 1 个字节的文件都要占用整个一个柱面，假设 Linux 环境下文件的平均大小为 1KB，那么分配 32KB 的柱面将浪费 97%的磁盘空间，也就是说，大的存取单位将带来严重的磁盘空间浪费。另一方面，如果物理块过小，则意味着对一个文件的操作将进行更多次的寻道延迟和旋转延迟，因而读取由小的物理块组成的文件将非常缓慢！可见，时间效率和空间效率在本质上是相互冲突的。

因此，最优的方法是计算出 Linux 环境下文件的平均大小，然后将物理块大小定为最接近扇区的整数倍大小。在 Ext2 中，物理块的大小是可变化的，这取决于你在创建文件系统时的选择，之所以不限制大小，也正体现了 Ext2 的灵活性和可扩充性，一是因为要适应近年来文件的平均长度缓慢增长的趋势，二是为了适应不同的需要。比如，如果一个文件系统主要用于 BBS 服务，考虑到 BBS 上的文章通常很短小，所以，物理块选得小一点是恰当的。通常，Ext2 的物理块占一个或几个连续的扇区，显然，物理块的数目是由磁盘容量等硬件因素决定的。逻辑块与物理块的关系类似于虚拟内存中的页与物理内存中的页面的关系。

具体文件系统所操作的基本单位是逻辑块，只在需要进行 I/O 操作时才进行逻辑块到物理块的映射，这显然避免了大量的 I/O 操作，因而文件系统能够变得高效。逻辑块作为一个抽象的概念，它必然要映射到具体的物理块上去，因此，逻辑块的大小必须是物理块大小的整数倍，一般说来，两者是一样大的。

通常，一个文件占用的多个物理块在磁盘上是不连续存储的，因为如果连续存储，则经过频繁的删除、建立、移动文件等操作，最后磁盘上将形成大量的空洞，很快磁盘上将无空间可供使用。因此，必须提供一种方法将一个文件占用的多个逻辑块映射到对应的非连续存储的物理块上去，Ext2 等类文件系统是用索引节点解决这个问题的，具体实现方法后面再予以介绍。

为了更好地说明逻辑块和物理块的关系，我们来看一个例子。

假设用户要对一个已有文件进行写操作，用户进程必须先打开这个文件，file 结构记录了该文件的当前位置。然后用户把一个指向用户内存区的指针和请求写的字节数传送给系统，请求写操作，这时系统要进行两次映射。

（1）一组字节到逻辑块的映射。

这个映射过程就是找到起始字节到结束字节所占用的所有逻辑块号。这是因为在逻辑空间，文件传输的基本单位是逻辑块而不是字节。

（2）逻辑块到物理块的映射。

这个过程必须要用到索引节点结构，该结构中有一个物理块指针数组，以逻辑块号为索引，通过这些指针找到磁盘上的物理块，具体实现将在介绍 Ext2 索引节点时再进行介绍。

图 9.1 是由一组请求的字节到物理块的映射过程示意图。

有了逻辑块和物理块的概念，我们也就知道通常所说的数据块是指逻辑块，以下没有特别说明，块或数据块指的是逻辑块。

在 Ext2 中，还有一个重要的概念：片（fragment），它的作用是什么？

每个文件必然占用整数个逻辑块，除非每个文件大小都恰好是逻辑块的整数倍，否则最后一个逻辑块必然有空间未被使用，实际上，每个文件的最后一个逻辑块平均要浪费一半的空间，显然最终浪费的还是物理块。在一个有很多文件的系统中，这种浪费是很大的。Ext2 使用片来解决这个问题。

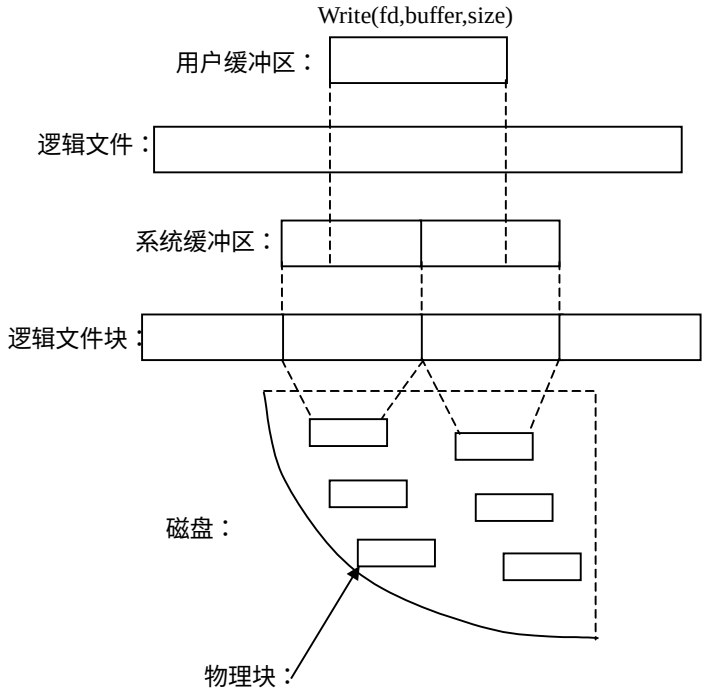

图 9.1　一组字节映射到物理块的示意图

片也是一个逻辑空间中的概念，其大小在 1KB 至 4KB 之间，但片的大小总是不大于逻辑块。假设逻辑块大小为 4KB，片大小为 1KB，物理块大小也是 1KB，当你要创建一个 3KB 大小的文件时，实际上分配给你了 3 个片，而不会给你一个逻辑块，当文件大小增加到 4KB 时，文件系统则分配一个逻辑块给你，而原来的四个片被清空。如果文件又增加到 5KB 时，则占用 1 个逻辑块和 1 个片。上述 3 种情况下，所占用的物理块分别是 3 个、4 个、5 个，如果不采用片，则要用到 4 个、4 个、8 个物理块，可见，使用片，减少了磁盘空间的浪费。当然，在物理块和逻辑块大小一样时，片就没有意义了。

由上面分析也可看出：

物理块大小<=片大小<=逻辑块大小

9.2　Ext2 的磁盘布局和数据结构

9.2.1　Ext2 的磁盘布局

文件系统的逻辑空间最终要通过逻辑块到物理块的映射转化为磁盘等介质上的物理空间，因此，对逻辑空间的组织和管理的好坏必然影响到物理空间的使用情况。一个文件系统，在磁盘上如何布局，要综合考虑以下几个方面的因素。

- 首先也是最重要的是要保证数据的安全性，也就是说当在向磁盘写数据时发生错误，要能保证文件系统不遭到破坏。
- 其次，数据结构要能高效地支持所有的操作。Ext2 中，最复杂的操作是硬链接操作。硬链接允许一个文件有多个名称，通过任何一个名称都将访问相同的数据。另一个比较复杂的操作是删除一个已打开的文件。
- 第三，磁盘布局应使数据查找的时间尽量短，以提高效率。驱动器查找分散的数据要比查找相邻的数据花多得多的时间。一个好的磁盘布局应该让相关的数据尽量连续分布。例如，同一个文件的数据应连续分布，并和包含该文件的目录文件相邻。
- 最后，磁盘布局应该考虑节省空间。虽然现在节省磁盘空间已不太重要，但也不应该无谓地浪费磁盘空间。

Ext2 的磁盘布局可以说综合考虑了以上几方面的因素，因此，它是一种高效、安全的文件系统。图 9.2 是 Ext2 的磁盘布局在逻辑空间中的映像。可以看出，它由一个引导块和重复的块组构成的，每个块组又由超级块、组描述符表、块位图、索引节点位图、索引节点表、数据区构成。引导块中含有可执行代码，启动计算机时，硬件从引导设备将引导块读入内存，然后执行它的代码。系统启动后，引导块不再使用。因此，引导块不属于文件系统管理。

以下对块组及块组中的数据结构的介绍可以充分领略 Ext2 磁盘布局的优越之处。

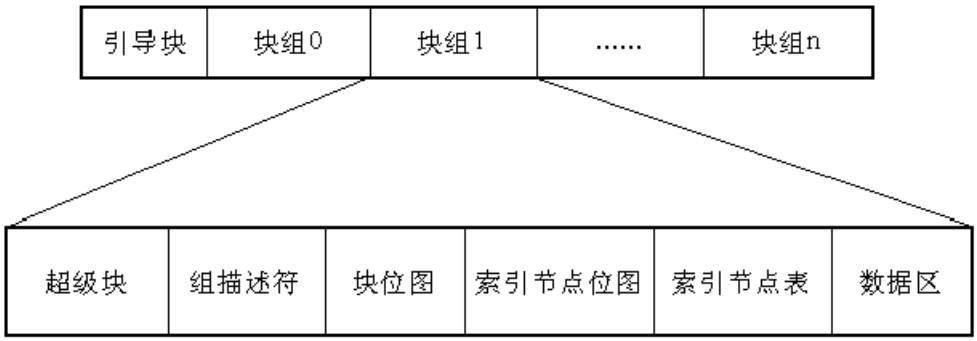

图 9.2　Ext2 磁盘布局在逻辑空间的映像

9.2.2　Ext2 的超级块

Ext2 超级块是用来描述 Ext2 文件系统整体信息的数据结构，是 Ext2 的核心所在。它是一个 ext2_super_block 数据结构（在 include/Linux/ext2_fs.h 中定义），其各个域及含义如下。

```
struct ext2_super_block
{    __u32     s_inodes_count;              /*文件系统中索引节点总数 */
     __u32     s_blocks_count;              /*文件系统中总块数 */
     __u32     s_r_blocks_count;        /*为超级用户保留的块数 */
     __u32     s_free_blocks_count;     /*文件系统中空闲块总数 */
     __u32     s_free_inodes_count;     /*文件系统中空闲索引节点总数*/
     __u32     s_first_data_block;          /* 文件系统中第一个数据块 */
     __u32     s_log_block_size;        /* 用于计算逻辑块大小 */
```

```
    __s32    s_log_frag_size;       /* 用于计算片大小 */
    __u32    s_blocks_per_group;        /* 每组中块数 */
    __u32    s_frags_per_group;     /* 每组中片数 */
    __u32    s_inodes_per_group;        /* 每组中索引节点数 */
    __u32    s_mtime;               /*最后一次安装操作的时间 */
    __u32    s_wtime;               /*最后一次对该超级块进行写操作的时间 */
    __u16    s_mnt_count;           /* 安装计数 */
    __s16    s_max_mnt_count;       /* 最大可安装计数 */
    __u16    s_magic;               /* 用于确定文件系统版本的标志 */
    __u16    s_state;               /* 文件系统的状态*/
    __u16    s_errors;              /* 当检测到有错误时如何处理 */
    __u16    s_minor_rev_level;         /* 次版本号 */
    __u32    s_lastcheck;           /* 最后一次检测文件系统状态的时间 */
    __u32    s_checkinterval;       /* 两次对文件系统状态进行检测的间隔时间 */
    __u32    s_rev_level;           /* 版本号 */
    __u16    s_def_resuid;          /* 保留块的默认用户标识号 */
    __u16    s_def_resgid;          /* 保留块的默认用户组标识号*/

  /*
   * 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...
   */
  __u32   s_first_ino;               /* 第一个非保留的索引节点 */
  __u16   s_inode_size;              /* 索引节点的大小 */
    __u16   s_block_group_nr;        /* 该超级块的块组号 */
    __u32   s_feature_compat;        /* 兼容特点的位图*/
    __u32   s_feature_incompat;      /* 非兼容特点的位图 */
    __u32   s_feature_ro_compat;     /* 只读兼容特点的位图*/
    __u8    s_uuid[16];              /* 128 位的文件系统标识号*/
    char    s_volume_name[16];       /* 卷名 */
    char    s_last_mounted[64];      /* 最后一个安装点的路径名 */
    __u32   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;   /* 给目录预分配的块数 */
    __u16   s_padding1;
    __u32   s_reserved[204];         /* 用 NULL 填充块的末尾 */
  };
```

从中我们可以看出，这个数据结构描述了整个文件系统的信息，下面对其中一些域作一些解释。

（1）文件系统中并非所有的块普通用户都可以使用，有一些块是保留给超级用户专用的，这些块的数目就是在 s_r_blocks_count 中定义的。一旦空闲块总数等于保留块数，普通用户无法再申请到块了。如果保留块也被使用，则系统就可能无法启动了。有了保留块，我们就可以确保一个最小的空间用于引导系统。

（2）逻辑块是从 0 开始编号的，对块大小为 1KB 的文件系统，s_first_data_block 为 1，对其他文件系统，则为 0。

（3）s_log_block_size 是一个整数，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。

同样,片的大小计算方法也是类似的,因为 Ext2 中还没有实现片,因此,s_log_frag_size 与 s_log_block_size 相等。

（4）Ext2 要定期检查自己的状态，它的状态取下面两个值之一。

```
#define EXT2_VALID_FS     0x0001
```

文件系统没有出错。

```
#define EXT2_ERROR_FS    0x0002
```

内核检测到错误。

s_lastcheck 就是用来记录最近一次检查状态的时间，而 s_checkinterval 则规定了两次检查状态的最大允许间隔时间。

（5）如果检测到文件系统有错误，则对 s_errors 赋一个错误值。一个好的系统应该能在错误发生时进行正确处理，有关 Ext2 如何处理错误将在后面介绍。

超级块被读入内存后，主要用于填写 VFS 的超级块，此外，它还要用来填写另外一个结构，这就是 ext2_super_info 结构，这一点我们可以从有关 Ext2 超级块的操作中看出，比如 ext2_read_super()。之所以要用到这个结构，是因为 VFS 的超级块必须兼容各种文件系统的不同的超级块结构,所以对某个文件系统超级块自己的特性必须用另一个结构保存于内存中，以加快对文件的操作，比如对 Ext2 来说，片就是它特有的，所以不能存储在 VFS 超级块中。Ext2 中的这个结构是 ext2_super_info，它其中的信息多是从磁盘上的索引节点计算得来的 。该结构定义于 include/Linux/ext2_fs_sb.h，下面是该结构及各个域含义：

```
struct ext2_sb_info
{
     unsigned long s_frag_size;                /* 片大小（以字节计） */
     unsigned long s_frags_per_block;          /* 每块中片数 */
     unsigned long s_inodes_per_block;         /* 每块中节点数*/
     unsigned long s_frags_per_group;          /* 每组中片数*/
     unsigned long s_blocks_per_group;         /* 每组中块数 */
     unsigned long s_inodes_per_group;         /*每组中节点数 */
     unsigned long s_itb_per_group;            /* 每组中索引节点表所占块数 */
     unsigned long s_db_per_group;             /* 每组中组描述符所在块数 */
     unsigned long s_desc_per_block;           /* 每块中组描述符数 */
     unsigned long s_groups_count;             /* 文件系统中块组数 */
     struct buffer_head * s_sbh;               /* 指向包含超级块的缓存 */
     struct buffer_head ** s_group_desc;   /*指向高速缓存中组描述符表块的
                                                指针数组的一个指针*/
```

```
    unsigned short s_loaded_inode_bitmaps;     /* 装入高速缓存中的节点位图块数*/
    unsigned short s_loaded_block_bitmaps;     /*装入高速缓存中的块位图块数*/
    unsigned long s_inode_bitmap_number[Ext2_MAX_GROUP_LOADED];
    struct buffer_head * s_inode_bitmap[Ext2_MAX_GROUP_LOADED];
    unsigned long s_block_bitmap_number[Ext2_MAX_GROUP_LOADED];
    struct buffer_head * s_block_bitmap[Ext2_MAX_GROUP_LOADED];
    int s_rename_lock;                         /*重命名时的锁信号量*/
    struct wait_queue * s_rename_wait;         /*指向重命名时的等待队列*/
    unsigned long  s_mount_opt;                /*安装选项*/
    unsigned short s_resuid;                   /*默认的用户标识号*/
    unsigned short s_resgid;                   /* 默认的用户组标识号*/
    unsigned short s_mount_state;              /*专用于管理员的安装选项*/
    unsigned short s_pad;                          /*填充*/
    int s_inode_size;                          /*节点的大小*/
    int s_first_ino;                               /*第一个节点号*/
};
```

s_block_bitmap_number[] 、 s_block_bitmap[] 、 s_inode_bitmap_number[] 、s_inode_bitmap[]是用来管理位图块高速缓存的，在介绍位图时再作说明。

另外，由于每个文件系统的组描述符表可能占多个块，这些块进入缓存后，用一个指针数组分别指向它们在缓存中的地址，而 s_group_desc 则是用来指向这个数组的，用相对于组描述符表首块的块数作索引，就可以找到指定的组描述符表块。

图 9.3 是 3 个与超级块相关的数据结构的关系示意图。

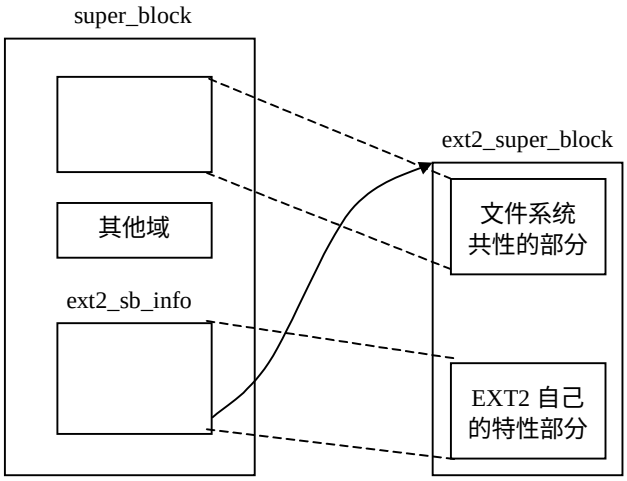

图 9.3　三种超级块结构的关系示意图

9.2.3　Ext2 的索引节点

Ext2 和 UNIX 类的文件系统一样，使用索引节点来记录文件信息。每一个普通文件和目录都有唯一的索引节点与之对应，索引节点中含有文件或目录的重要信息。当你要访问一个文件或目录时，通过文件或目录名首先找到与之对应的索引节点，然后通过索引节点得到文件或目录的信息及磁盘上的具体的存储位置。Ext2 的索引节点的数据结构叫 ext2_inode，在

include/Linux/ext2_fs.h 中定义，下面是其结构及各个域的含义（不同版本，该结构略有不同）。

```
struct ext2_inode {
    __u16 i_mode;                   /* 文件类型和访问权限 */
     __u16     i_uid;                    /* 文件拥有者标识号*/
     __u32     i_size;                   /* 以字节计的文件大小 */
     __u32     i_atime;                  /* 文件的最后一次访问时间 */
     __u32     i_ctime;                  /* 该节点最后被修改时间 */
     __u32     i_mtime;                  /* 文件内容的最后修改时间 */
     __u32     i_dtime;                  /* 文件删除时间 */
     __u16     i_gid;                    /* 文件的用户组标志符 */
     __u16     i_links_count;            /* 文件的硬链接计数 */
     __u32     i_blocks;                 /* 文件所占块数（每块以 512 字节计）*/
     __u32     i_flags;                  /* 打开文件的方式 */
   union                               /*特定操作系统的信息*/
   __u32   i_block[Ext2_N_BLOCKS];     /* 指向数据块的指针数组 */
     __u32     i_version;                /* 文件的版本号（用于 NFS） */
     __u32     i_file_acl;               /*文件访问控制表（已不再使用） */
     __u32     i_dir_acl;                /*目录访问控制表（已不再使用）*/
      __u8     l_i_frag;                 /* 每块中的片数 */
     __u32   i_faddr;                    /* 片的地址 */
      union                              /*特定操作系统信息*/
}
```

从中可以看出，索引节点是用来描述文件或目录信息的。

以下，对其中一些域作一定解释。

（1）前面说过，Ext2 通过索引节点中的数据块指针数组进行逻辑块到物理块的映射。在 Ext2 索引节点中，数据块指针数组共有 15 项，前 12 个为直接块指针，后 3 个分别为“一次间接块指针”、“二次间接块指针”、“三次间接块指针”，如图 9.4 所示。

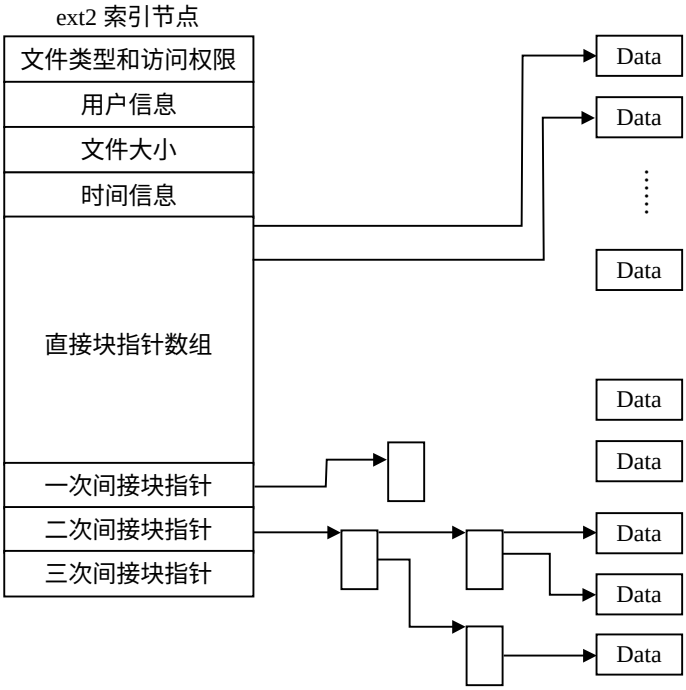

图 9.4　索引节点的数据块指针数组示意图

所谓“直接块”，是指该块直接用来存储文件的数据，而“一次间接块”是指该块不存储数据，而是存储直接块的地址，同样，“二次间接块”存储的是“一次间接块”的地址。这里所说的块，指的都是物理块。Ext2 默认的物理块大小为 1KB，块地址占 4 个字节（32 位），所以每个物理块可以存储 256 个地址。这样，文件大小最大可达 12KB+256KB+ 64MB+16GB。但实际上，Linux 是 32 位的操作系统，故文件大小最大只能为 4GB。

系统是以逻辑块号为索引查找物理块的。例如，要找到第 100 个逻辑块对应的物理块，因为 256+12>100+12，所以要用到一次间接块，在一次间接块中查找第 88 项，此项内容就是对应的物理块的地址。而如果要找第 1000 个逻辑块对应的物理块，由于 1000>256+12，所以要用到二次间接块了。

（2）索引节点的标志（flags）取下列几个值的可能组合。

EXT2_SECRM_FL　0x00000001

完全删除标志。设置这个标志后，删除文件时，随机数据会填充原来的数据块。

EXT2_UNRM_FL　0x00000002

可恢复标志。设置这个标志后，删除文件时，文件系统会保留足够信息，以确保文件仍能恢复（仅在一段时间内）。

EXT2_COMR_FL　0x00000004

压缩标志。设置这个标志后，表明该文件被压缩过。当访问该文件时，文件系统必须采用解压缩算法进行解压。

EXT2_SYNC_FL　0x00000008

同步更新标志。设置该标志后，则该文件必须和内存中的内容保持一致，对这种文件进行异步输入、输出操作是不允许的。这个标志仅用于节点本身和间接块。数据块总是异步写

入磁盘的。

除了这几个常用标志外，还有 12 个标志就不一一介绍了。

（3）索引节点在磁盘上是经过编号的。其中，有一些节点有特殊用途，用户不能使用。这些特殊节点也在 include/Linux/ext2_fs.h 中定义。

```
#define EXT2_BAD_INO                1
```

该节点所对应的文件中包含着该文件系统中坏块的链接表。

```
#define EXT2_ROOT_INO               2
```

该文件系统的根目录所对应的节点。

```
#define EXT2_IDX_INO            3
```

ACL（访问控制链表）节点。

```
#define EXT2_DATA_INO           4
```

ACL 节点。

```
#define EXT2_BOOT_LOADER_INO 5
```

用于引导系统的文件所对应的节点。

```
#define EXT2_UNDEL_DIR_INO      6
```

文件系统中可恢复的目录对应的节点。

没有特殊用途的第一个节点号为 11。

```
#define  EXT2_FIRST_INO         11
```

（4）文件的类型、访问权限、用户标识号、用户组标识号等将在后面介绍。

与 Ext2 超级块类似，当磁盘上的索引节点调入内存后，除了要填写 VFS 的索引节点外，系统还要根据它填写另一个数据结构，该结构叫 ext2_inode_info，其作用也是为了存储特定文件系统自己的特性，它在 include/Linux/ext2_fs_i.h 中定义如下：

```
struct ext2_inode_info
{
     __u32     i_data[15];                 /*数据块指针数组*/
     __u32     i_flags;                    /*打开文件的方式*/
     __u32     i_faddr;                    /*片的地址*/
     __u8 i_frag_no;                   /*如果用到片，则是第一个片号*/
     __u8 i_frag_size;                 /*片大小*/
      __u16    i_osync;                  /*同步*/
     __u32     i_file_acl;               /*文件访问控制链表*/
     __u32     i_dir_acl;                  /*目录访问控制链表*/
     __u32     i_dtime;                    /*文件的删除时间*/
     __u32     i_block_group;              /*索引节点所在的块组号*/
/******以下四个域是用于操作预分配块的************/
     __u32     i_next_alloc_block;
     __u32     i_next_alloc_goal;
     __u32     i_prealloc_block;
     __u32     i_prealloc_count;

  __u32   i_dir_start_lookup
     int     i_new_inode:1   /* Is a freshly allocated inode */
};
```

VFS 索引节点中是没有物理块指针数组的域，这个 Ext2 特有的域在调入内存后，就必须保存在 ext2_inode_info 这个结构中。此外，片作为 Ext2 比较特殊的地方，在

ext2_inode_info 中也保存了一些相关的域。另外，Ext2 在分配一个块时通常还要预分配几个连续的块，因为它判断这些块很可能将要被访问，所以采用预分配的策略可以减少磁头的寻道时间。这些用于预分配操作的域也被保存在 ext2_inode_info 结构中。

9.2.4　组描述符

块组中，紧跟在超级块后面的是组描述符表，其每一项称为组描述符，是一个叫 ext2_group_desc 的数据结构，共 32 字节。它是用来描述某个块组的整体信息的。

```
struct ext2_group_desc
{
    __u32    bg_block_bitmap;        /*组中块位图所在的块号 */
    __u32    bg_inode_bitmap;            /*组中索引节点位图所在块的块号 */
    __u32    bg_inode_table;             /*组中索引节点表的首块号 */
    __u16    bg_free_blocks_count;   /*组中空闲块数 */
    __u16    bg_free_inodes_count;   /* 组中空闲索引节点数 */
    __u16    bg_used_dirs_count;         /*组中分配给目录的节点数 */
    __u16    bg_pad;                 /*填充，对齐到字*/
    __u32 [3]  bg_reserved;          /*用 NULL 填充 12 个字节*/
}
```

每个块组都有一个相应的组描述符来描述它，所有的组描述符形成一个组描述符表，组描述符表可能占多个数据块。组描述符就相当于每个块组的超级块，一旦某个组描述符遭到破坏，整个块组将无法使用，所以组描述符表也像超级块那样，在每个块组中进行备份，以防遭到破坏。组描述符表所占的块和普通的数据块一样，在使用时被调入块高速缓存。

9.2.5　位图

在 Ext2 中，是采用位图来描述数据块和索引节点的使用情况的，每个块组中都有两个块，一个用来描述该组中数据块的使用情况，另一个描述该组中索引节点的使用情况。这两个块分别称为数据位图块和索引节点位图块。数据位图块中的每一位表示该组中一个块的使用情况，如果为 0，则表示相应数据块空闲，为 1，则表示已分配，索引节点位图块的使用情况类似。

Ext2 在安装后，用两个高速缓存分别来管理这两种位图块。每个高速缓存最多同时只能装入 Ext2_MAX_GROUP_LOADED 个位图块或索引节点块，当前该值定义为 8，所以也应该采用一些算法来管理这两个高速缓存，Ext2 中采用的算法类似于 LRU 算法。

前面说过，ext2_sp_info 结构中有 4 个域用来管理这两个高速缓存，其中 s_block_bitmap_number[]数组中存有进入高速缓存的位图块号（即块组号，因为一个块组中只有一个位图块），而 s_block_bitmap[]数组则存储了相应的块在高速缓存中的地址。

s_inode_bitmap_number[]和 s_inode_bitmap[]数组的作用类似上面。

我们通过一个具体的函数来看 Ext2 是如何通过这 4 个域管理位图块管理高速缓存的。在 Linux/fs/ext2/balloc.c 中，有一个函数 load__block_bitmap()，它用来调入指定的数据位图块，下面是它的执行过程。

(1) 如果指定的块组号大于块组数，出错，结束。

(2) 通过搜索 s_block_bitmap_number[]数组可知位图块是否已进入高速缓存，如果已进入，则结束，否则，继续；

(3) 如果块组数不大于 Ext2_MAX_GROUP_LOADED，高速缓存可以同时装入所有块组的数据块位图块，不用采用什么算法，只要找到 s_block_bitmap_number[]数组中第一个空闲元素，将块组号写入，然后将位图块调入高速缓存，最后将它在高速缓存中的地址写入 s_block_bitmap[]数组中。

(4) 如果块组数大于 Ext2_MAX_GROUP_LOADED，则需要采用以下算法：

- 首先通过 s_block_bitmap_number[]数组判断高速缓存是否已满，若未满，则操作过程类似上一步，不同之处在于需要将 s_block_bitmap_number[]数组各元素依次后移一位，而用空出的第一个元素存储块组号，s_block_bitmap[]也要做相同处理；
- 如果高速缓存已满，则将 s_block_bitmap[]数组最后一项所指的位图块从高速缓存中交换出去，然后调入所指定的位图块，最后对这两个数组做与上面相同的操作。

可以看出，这个算法很简单，就是对两个数组的简单操作，只是在块组数大于 Ext2_MAX_GROUP_LOADED 时，要求数组的元素按最近访问的先后次序排列，显然，这样也是为了更合理的进行高速缓存的替换操作。

9.2.6 索引节点表及实例分析

在两个位图块后面，就是索引节点表了，每个块组中的索引节点都存储在各自的索引节点表中，并且按索引节点号依次存储。索引节点表通常占好几个数据块，索引节点表所占的块使用时也像普通的数据块一样被调入块高速缓存。

有了以上几个概念和数据结构后，我们分析一个具体的例子，来看看这些数据结构是如何配合工作的。

在 fs/ext2/inode.c 中，有一个 ext2_read_inode()，用来读取指定的索引节点信息。其代码如下：

```
void ext2_read_inode  (struct inode * inode)
{
          struct buffer_head * bh;
          struct ext2_inode * raw_inode;
          unsigned long block_group;
          unsigned long group_desc;
          unsigned long desc;
          unsigned long block;
          unsigned long offset;
         struct ext2_group_desc * gdp;

          if  ( ( inode->i_ino  !=  EXT2_ROOT_INO  &&  inode->i_ino  !=  EXT2_ACL_IDX_INO  &&
inode->i_ino != EXT2_ACL_DATA_INO && inode->i_ino < EXT2_FIRST_INO (inode->i_sb) )  || inode->i_ino
> le32_to_cpu (inode->i_sb->u.ext2_sb.s_es->s_inodes_count) )
              {
                  ext2_error  (inode->i_sb, "ext2_read_inode",
                              "bad inode number: %lu", inode->i_ino) ;
```

```
        goto bad_inode;
}
block_group =  (inode->i_ino  -  1)  / EXT2_INODES_PER_GROUP (inode->i_sb) ;
 if  (block_group >= inode->i_sb->u.ext2_sb.s_groups_count)  {
        ext2_error  (inode->i_sb, "ext2_read_inode",
                     "group >= groups count") ;
        goto bad_inode;
}
 group_desc = block_group >> EXT2_DESC_PER_BLOCK_BITS (inode->i_sb) ;
 desc = block_group &  (EXT2_DESC_PER_BLOCK (inode->i_sb)  - 1) ;
 bh = inode->i_sb->u.ext2_sb.s_group_desc[group_desc];
 if  (!bh)  {
        ext2_error  (inode->i_sb, "ext2_read_inode",
                     "Descriptor not loaded") ;
        goto bad_inode;
 }

 gdp =  (struct ext2_group_desc *)  bh->b_data;
 /*
  * 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[desc].bg_inode_table)  +
        (offset >> EXT2_BLOCK_SIZE_BITS (inode->i_sb) ) ;
if  (! (bh = sb_bread (inode->i_sb, block) ) )  {
        ext2_error  (inode->i_sb, "ext2_read_inode",
                     "unable to read inode block - "
                     "inode=%lu, block=%lu", inode->i_ino, block) ;
        goto bad_inode;
 }
offset &=  (EXT2_BLOCK_SIZE (inode->i_sb)  - 1) ;
 raw_inode =  (struct ext2_inode *)   (bh->b_data + offset) ;

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 = le32_to_cpu (raw_inode->i_atime) ;
inode->i_ctime = le32_to_cpu (raw_inode->i_ctime) ;
inode->i_mtime = le32_to_cpu (raw_inode->i_mtime) ;
 inode->u.ext2_i.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 || inode->u.ext2_i.i_dtime) )  {
               /* this inode is deleted */
                brelse  (bh) ;
                goto bad_inode;
        }
        inode->i_blksize = PAGE_SIZE;   /* This is the optimal IO size  (for stat) , not
the fs block size */
        inode->i_blocks = le32_to_cpu (raw_inode->i_blocks) ;
        inode->i_version = ++event;
        inode->u.ext2_i.i_flags = le32_to_cpu (raw_inode->i_flags) ;
        inode->u.ext2_i.i_faddr = le32_to_cpu (raw_inode->i_faddr) ;
        inode->u.ext2_i.i_frag_no = raw_inode->i_frag;
        inode->u.ext2_i.i_frag_size = raw_inode->i_fsize;
        inode->u.ext2_i.i_file_acl = le32_to_cpu (raw_inode->i_file_acl) ;
        if  (S_ISREG (inode->i_mode) )
         inode->i_size |= ( (__u64) le32_to_cpu (raw_inode->i_size_high) ) <<32;
        else
              inode->u.ext2_i.i_dir_acl = le32_to_cpu (raw_inode->i_dir_acl) ;
        inode->i_generation = le32_to_cpu (raw_inode->i_generation) ;
        inode->u.ext2_i.i_prealloc_count = 0;
       inode->u.ext2_i.i_block_group = block_group;

        /*
         * 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  (block = 0; block < EXT2_N_BLOCKS; block++)
               inode->u.ext2_i.i_data[block] = raw_inode->i_block[block];

        if  (inode->i_ino == EXT2_ACL_IDX_INO ||
           inode->i_ino == EXT2_ACL_DATA_INO)
                /* Nothing to do */ ;
        else if  (S_ISREG (inode->i_mode) )  {
               inode->i_op = &ext2_file_inode_operations;
                inode->i_fop = &ext2_file_operations;
                inode->i_mapping->a_ops = &ext2_aops;
        } else if  (S_ISDIR (inode->i_mode) )  {
                inode->i_op = &ext2_dir_inode_operations;
                inode->i_fop = &ext2_dir_operations;
                inode->i_mapping->a_ops = &ext2_aops;
       } else if  (S_ISLNK (inode->i_mode) )  {
                if  (!inode->i_blocks)
                        inode->i_op = &ext2_fast_symlink_inode_operations;
               else {
                       inode->i_op = &page_symlink_inode_operations;
                        inode->i_mapping->a_ops = &ext2_aops;
               }
       } else
                init_special_inode (inode, inode->i_mode,
                                    le32_to_cpu (raw_inode->i_block[0]) ) ;
        brelse  (bh) ;
       inode->i_attr_flags = 0;
```

```
        if (inode->u.ext2_i.i_flags & EXT2_SYNC_FL) {
                inode->i_attr_flags |= ATTR_FLAG_SYNCRONOUS;
                inode->i_flags |= S_SYNC;
        }
        if (inode->u.ext2_i.i_flags & EXT2_APPEND_FL) {
                inode->i_attr_flags |= ATTR_FLAG_APPEND;
               inode->i_flags |= S_APPEND;
        }
        if (inode->u.ext2_i.i_flags & EXT2_IMMUTABLE_FL) {
                inode->i_attr_flags |= ATTR_FLAG_IMMUTABLE;
               inode->i_flags |= S_IMMUTABLE;
       }
        if (inode->u.ext2_i.i_flags & EXT2_NOATIME_FL) {
               inode->i_attr_flags |= ATTR_FLAG_NOATIME;
               inode->i_flags |= S_NOATIME;
       }
        return;

bad_inode:
        make_bad_inode (inode) ;
        return;
}
```

这个函数的代码有 200 多行，为了突出重点，下面是对该函数主要内容的描述。

- 如果指定的索引节点号是一个特殊的节点号（EXT2_ROOT_INO、EXT2_ACL_IDX_ INO 及 EXT2_ACL_DATA_INO），或者小于第一个非特殊用途的节点号，即 EXT2_FIRST_ INO（为 11），或者大于该文件系统中索引节点总数，则输出错误信息，并返回。
- 用索引节点号整除每组中索引节点数，计算出该索引节点所在的块组号。

即：block_group = (inode->i_ino - 1) / Ext2_INODES_PER_GROUP(inode->i_sb)。

- 找到该组的组描述符在组描述符表中的位置。因为组描述符表可能占多个数据块，所以需要确定组描述符在组描述符表的哪一块以及是该块中第几个组描述符。

即：group_desc = block_group >> Ext2_DESC_PER_BLOCK_BITS(inode->i_sb) 表示块组号整除每块中组描述符数，计算出该组的组描述符在组描述符表中的哪一块。我们知道，每个组描述符是 32 字节大小，在一个 1KB 大小的块中可存储 32 个组描述符。

- 块组号与每块中组的描述符数进行“ 与 ”运算，得到这个组描述符具体是该块中第几个描述符。即 desc = block_group & (Ext2_DESC_PER_BLOCK(inode->i_sb) - 1)。
- 有了 group_desc 和 desc，接下来在高速缓存中找这个组描述符就比较容易了。

即 ： bh = inode->i_sb->u.ext2_sb.s_group_desc[group_desc] ， 首 先 通 过 s_group_desc[]数组找到这个组描述符所在块在高速缓存中的缓冲区首部；然后通过缓冲区首部找到数据区，即 gdp = (struct ext2_group_desc *) bh->b_data。

- 找到组描述符后,就可以通过组描述符结构中的 bg_inode_tabl 找到索引节点表首块在高速缓存中的地址：

```
    offset = ( (inode->i_ino - 1) % Ext2_INODES_PER_GROUP (inode->i_sb) ) *
         Ext2_INODE_SIZE (inode->i_sb) /*计算该索引节点在块中的偏移位置*/ ;
block = le32_to_cpu (gdp[desc].bg_inode_table)  +
(offset >> Ext2_BLOCK_SIZE_BITS (inode->i_sb) ) /*计算索引节点所在块的地址*/。
```

- 代码中 le32_to_cpu（）、le16_to_cpu（）按具体 CPU 的要求进行数据的排列，在 i386 处理器上访问 Ext2 文件系统时这些函数不做任何事情。因为不同的处理器在存取数据时在字节的排列次序上有所谓“ big ending ”和“ little ending ”之分。例如，i386 就是“ little ending ”处理器，它在存储一个 16 位数据 0x1234 时，实际存储的却是 0x3412，对 32 位数据也是如此。这里索引节点号与块的长度都作为 32 位或 16 位无符号整数存储在磁盘上，而同一磁盘既可以安装在采用“ little ending ”方式的 CPU 机器上，也可能安装在采用“ big ending ”方式的 CPU 机器上，所以要选择一种形式作为标准。事实上，Ext2 采用的标准为“ little ending ”，所以，le32_to_cpu（）、le16_to_cpu（）函数不作任何转换。
- 计算出索引节点所在块的地址后，就可以调用 sb_bread（）通过设备驱动程序读入该块。从磁盘读入的索引节点为 ext2_Inode 数据结构，前面我们已经看到它的定义。磁盘上索引节点中的信息是原始的、未经加工的，所以代码中称之为 raw_ inode，即：raw_inode = (struct ext2_inode *) (bh->b_data + offset)
- 与磁盘索引节点 ext2_ inode 相对照，内存中 VFS 的 inode 结构中的信息则分为两部分，一部分是属于 VFS 层的，适用于所有的文件系统；另一部分则属于具体的文件系统，这就是 inode 中的那个 union，因具体文件系统的不同而赋予不同的解释。对 Ext2 来说，这部分数据就是前面介绍的 ext2_inode_info 结构。至于代表着符号链接的节点，则并没有文件内容（数据），所以正好用这块空间来存储链接目标的路径名。ext2_inode_info 结构的大小为 60 个字节。虽然节点名最长可达 255 个字节，但一般都不会太长，因此将符号链接目标的路径名限制在 60 个字节不至于引起问题。代码中 inode->u.*设置的就是 Ext2 文件系统的特定信息。
- 接着，根据索引节点所提供的信息设置 inode 结构中的 inode_operations 结构指针和 file_operations 结构指针，完成具体文件系统与虚拟文件系统 VFS 之间的连接。
- 目前 2.4 版内核并不支持存取控制表 ACL，因此，代码中只是为之留下了位置，而暂时没做任何处理。
- 另外，通过检查 inode 结构中的 mode 域来确定该索引节点是常规文件（S_ISREG）、目录（S_ISDIR）、符号链接（S_ISLNK）还是其他特殊文件而作不同的设置或处理。例如，对 Ext2 文件系统的目录节点，就将 i_op 和 i_fop 分配设置为 ext2_dir_inode_operations 和 ext2_dir_operations。而对于 Ext2 常规文件，则除 i_op 和 i_fop 以外，还设置了另一个指针 a_ops，它指向一个 address_apace_operation 结构，用于文件到内存空间的映射或缓冲。对特殊文件，则通过 init_special_inode（）函数加以检查和处理。

从这个读索引节点的过程可以看出，首先要寻找指定的索引节点 ，要找索引节点，必须先找组描述符，然后通过组描述符找到索引节点表，最后才是在这个索引节点表中找索引节点。当从磁盘找到索引节点以后，就要把其读入内存，并存放在 VFS 索引节点相关的域中。从这个实例的分析，读者可以仔细体会前面所介绍的各种数据结构的具体应用。

9.2.7 Ext2 的目录项及文件的定位

文件系统一个很重要的问题就是文件的定位，如何通过一个路径来找到一个文件的具体位置，就要依靠 ext2_dir_entry 这个结构。

1．Ext2 目录项结构

在 Ext2 中，目录是一种特殊的文件，它是由 ext2_dir_entry 这个结构组成的列表。这个结构是变长的，这样可以减少磁盘空间的浪费，但是，它还是有一定的长度方面的限制，一是文件名最长只能为 255 个字符。二是尽管文件名长度可以不限（在 255 个字符之内），但系统自动将之变成 4 的整数倍，不足的地方用 NULL 字符（\0）填充。目录中有文件和子目录，每一项对应一个 ext2_dir_entry。该结构在 include/Linux/ext2_fs.h 中定义如下：

```
 /*
  * Structure of a directory entry
  */
  #define EXT2_NAME_LEN 255
struct ext2_dir_entry {
        __u32   inode;                  /* Inode number */
        __u16   rec_len;                /* Directory entry length */
        __u16   name_len;               /* Name length */
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

这是老版本的定义方式，在 ext2_fs.h 中还有一种新的定义方式：

```
 /*
  * 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 {
        __u32   inode;                  /* Inode number */
        __u16   rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
       __u8    file_type;
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

其二者的差异在于，一是新版中结构名改为 ext2_dir_entry_2；二是老版本中 ext2_dir_entry 中的 name_len 为无符号短整数，而新版中则改为 8 位的无符号字符，腾出一半用作文件类型。目前已定义的文件类型为：

```
  *
  * Ext2 directory file types.  Only the low 3 bits are used.  The
  * other bits are reserved for now.
  */
enum {
        EXT2_FT_UNKNOWN,    /*未知*/
        EXT2_FT_REG_FILE,   /*常规文件*/
        EXT2_FT_DIR,            /*目录文件*/
        EXT2_FT_CHRDEV,     /*字符设备文件*/
        EXT2_FT_BLKDEV,     /*块设备文件*/
        EXT2_FT_FIFO,       /*命名管道文件*/
        EXT2_FT_SOCK,       /*套接字文件*/
        EXT2_FT_SYMLINK,    /*符号连文件*/
        EXT2_FT_MAX             /*文件类型的最大个数*/
```

};

2．各种文件类型如何使用数据块

我们说，不管哪种类型的文件，每个文件都对应一个 inode 结构，在 inode 结构中有一个指向数据块的指针 i_blaock，用来标识分配给文件的数据块。但是 Ext2 所定义的文件类型以不同的方式使用数据块。有些类型的文件不存放数据，因此，根本不需要数据块，下面对不同文件类型如何使用数据块给予说明。

（1）常规文件

常规文件是最常用的文件。常规文件在刚创建时是空的，并不需要数据块，只有在开始有数据时才需要数据块；可以用系统调用 truncate()清空一个常规文件。

（2）目录文件

Ext2 以一种特殊的方式实现了目录,这种文件的数据块中存放的就是 ext2_dir_entry_2 结构。如前所述，这个结构的最后一个域是可变长度数组，因此该结构的长度是可变的。

在 ext2_dir_entry_2 结构中，因为 rec_len 域是目录项的长度，把它与目录项的起始地址相加就得到下一个目录项的起始地址，因此说，rec_len 可以被解释为指向下一个有效目录项的指针。为了删除一个目录项，把 ext2_dir_entry_2 的 inode 域置为 0 并适当增加前一个有效目录项 rec_len 域的值就可以了。

（3）符号连

如果符号连的路径名小于 60 个字符，就把它存放在索引节点的 i_blaock 域，该域是由 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。

（4）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。

3．文件的定位

文件的定位是一个复杂的过程，我们先看一个具体的例子，然后再结合上面的数据结构具体介绍一下如何找到一个目录项的过程。

如果要找的文件为普通文件，则可通过文件所对应的索引节点找到文件的具体位置，如果是一个目录文件，则也可通过相应的索引节点找到目录文件具体所在，然后再从这个目录文件中进行下一步查找，来看一个具体的例子。

假设路径为/home/user1/file1，home 和 user1 是目录名，而 file1 为文件名。为了找到这个文件，有两种途径，一是从根目录开始查找，二是从当前目录开始查找。假设我们从根目录查找，则必须先找到根目录的节点，这个节点的位置在 VFS 中的超级块中已经给出，然后可找到根目录文件，其中必有 home 所对应的目录项，由此可先找到 home 节点，从而找到 home 的目录文件，然后依次是 user1 的节点和目录文件，最后，在该目录文件中的 file1 目录项中找到 file1 的节点，至此，已经可以找到 file1 文件的具体所在了。

目录中还有两个特殊的子目录：“．”和“．．”，分别代表当前目录和父目录。它们是无法被删除的，其作用就是用来进行相对路径的查找。

现在，我们来分析一下 fs/ext2/dir.c 中的函数 ext2_find_entry（），该函数从磁盘

上找到并读入当前节点的目录项，其代码及解释如下：

```
/*
 *      ext2_find_entry ()
 *
 * finds an entry in the specified directory with the wanted name. It
 * returns the page in which the entry was found, and the entry itself
 *  (as a parameter - res_dir) . Page is returned mapped and unlocked.
 * Entry is guaranteed to be valid.
 */
typedef struct ext2_dir_entry_2 ext2_dirent;
struct ext2_dir_entry_2 * ext2_find_entry  (struct inode * dir,
                        struct dentry *dentry, struct page ** res_page)
{
      const char *name = dentry->d_name.name;       /*目录项名*/
      int namelen = dentry->d_name.len;                 /*目录项名的长度*/
      unsigned reclen = EXT2_DIR_REC_LEN (namelen) ;    /*目录项的长度*/
      unsigned long start, n;
      unsigned long npages = dir_pages (dir) ;          /*把以字节为单位的文件大小
转换为物理页面数*/
        struct page *page = NULL;
        ext2_dirent * de;                                   /*de 为要返回的 Ext2 目录项
                                                            /*结构*/
        /* OFFSET_CACHE */
        *res_page = NULL;

       start = dir->u.ext2_i.i_dir_start_lookup;   /*目录项在内存的起始位置*/
        if  (start >= npages)
               start = 0;
        n = start;
       do {
               char *kaddr;
               page = ext2_get_page (dir, n) ;         /*从页面高速缓存中获得目录
项所在的页面*/
               if  (!IS_ERR (page) )  {
                     kaddr = page_address (page) ;     /*获得 page 所对应的内核
虚拟地址*/
                      de =  (ext2_dirent *)  kaddr;   /*获得该目录项结构的
起始地址*/
                        kaddr += PAGE_CACHE_SIZE - reclen; /* PAGE_CACHE_SIZE
的大小为 1 个页面的大小，假定所有的目录项结构都存放在一个页面内*/
                      while  ( (char *)  de <= kaddr)  {  /*循环查找，直到
  找到匹配的目录项*/
                              if  (ext2_match  (namelen, name, de) )
                                     goto found;
                             de = ext2_next_entry (de) ;
                      }
                     ext2_put_page (page) ;      /*释放目录项所在的页面*/
               }
               if  (++n >= npages)
                       n = 0;
       } while  (n != start) ;
        return NULL;
```

```
found:
        *res_page = page;
        dir->u.ext2_i.i_dir_start_lookup = n;
        return de;
}
```

通过代码的注释,读者应当很容易明理解其含义。

9.3 文件的访问权限和安全

Linux 作为一种网络操作系统，允许多个用户使用。为了保护用户的个人文件不被其他用户侵犯，Linux 提供了文件权限的机制。这种机制使得一个文件或目录归一个特定的用户所有，这个用户有权对他所拥有的文件或目录进行存取或其他操作，也可以设置其他用户对这些文件或目录的操作权限。

Linux 还用到了用户组的概念。每个用户在建立用户目录时都被放到至少一个用户组中（当然，系统管理员可以将用户编进多个用户组中）。用户组通常是根据使用计算机的用户的种类来划分的。例如，普通用户通常属于 usrs 组。另外，还有几个系统定义的组（如 bin 和 admin），系统使用这些组来控制对资源的访问。

普通文件的权限有 3 部分：读、写和执行。分别用“ r ”、“ w ”、“ x ”来表示。目录也有这 3 种表示方式，分别表示：列出目录内容、在目录中建立或删除文件、进入和退出目录。以一个例子来解释文件权限的概念。

我们用带-l 选项的 ls 命令来显示包括文件权限的长格式信息。

```
/ $  ls  -l   /home/user1/file1
-rw-r--r--     1      user1          usrs       490      JUN  28  21:50     file1
```

在这个文件的长格式信息中，第 1 列代表的就是文件的类型和权限。其中，第 1 个字母表示文件的类型，“ - ”表明这是一个普通文件。接下来 9 个字符每 3 个一组依次代表文件的所有者、所有者所在用户组和组外其他用户对该文件的访问权限，代表文件权限的 3 个字符依次是读、写和执行权限，当用户没有相应的权限时，系统在该权限对应的位置上用“ - ”表示。所以，本例表示用户 user1 自己对该文件有读和写的权限，但没有执行权限，而 user1 所在 usrs 组的其他用户和组外的用户对该文件只有读权限，没有写和执行权限。

有时候，我们会看到权限的执行位上出现了“ s ”，而不是“ x ”。这涉及到进程中的概念。一个进程，除了进程标识号（PID）外，还有 4 个标识号表示进程的权限。它们是：

- real user ID　　实际用户标识号
- real group ID　　实际用户组标识号
- effective user ID　　有效用户标识号
- effective group ID　　有效用户组标识号

实际用户标识号就是运行该进程的用户的 UID，实际用户组标识号就是运行该进程的用户的 GID。一般情况下，有效用户标识号、有效用户组标识号分别和实际用户标识号、实际用户组标识号相同。但如果设置 setuid 位和 setgid 位，则这两个标识号在对文件进行操作时自动转换成该文件所有者和所有组的标识号。setuid 位和 setgid 位设置与否，就是看文

件权限位上是否有“ s ”，如果用户的执行权限位为“ s ”，则设置了 setuid 位，如果用户组的执行权限位为“ s ”，则设置了 setgid 位。

这两个标识位的设置正确与否关系到整个系统的安全，所以非同小可，我们以两个例子来说明。

/bin/passwd 就是一个设置了 setuid 位的文件。当/bin/passwd 被执行时，它要在/etc 目录下找一个也叫 passwd 的文件，该文件是个文本文件，里面记录了全部用户的数据，如口令、用户标识号等。该文件显然不能让超级用户以外的人修改，所以该文件的存取权限是 rw-r--r--。但是，普通用户在更改它自己的口令时，必须改动这个文件，解决这个问题的方法便是设置 setuid 的位，使指令在被执行时有和 root 相同的权利，这样就可以修改这个文件了。

setuid 位的设立是有风险的！如果某个普通进程执行一个设置了 setuid 位，且所有者是“ root ”的文件，立刻具有了超级用户的权利，万一该程序有 BUG，就留下了被人入侵系统的可能。因此，把文件按性质分类，再用设置 setgid 位来达到分享的目标是一个不错的办法。

Linux 中有一个叫/etc/kmem 的字符设备文件，目的为存储一些核心程序要用到的数据。当用户注册到系统时所输入的口令都存在其中，所以这个文件不能给普通用户读写，以免给企图侵入系统者可乘之机，但将使用权限设为“ rw------- ”也不行，这样一般用户无法用 ps 等显示系统状态的指令（这类显示系统状态的指令必须要读取 kmem 的内容）。如果将存取权限设置成“ rw-r----- ”，再把组标识号改成与 kmem 一样，最后再设置 setgid 位，这样，无论谁执行 ps，该进程的组标识号都变得和 ps 一样，从而就能读取 kmem 的数据了。

作为一个系统管理员，能用 setgid 来代替 setuid 就尽量用 setgid，因为它的风险小得多！

还有一个不引人注意的现象，就是在其他用户的执行权限位上可能出现“ t ”，而不是“ x ”，这表示该文件被装入内存后，将一直保留在内存中，就像 dos 下 TSR 程序，这样可以减少程序的装入时间，增加程序的反应速度。因此，系统管理员可以对一些使用频率较高的工具程序设置该位。

创建一个新文件时，需要提供文件的访问权限位，但是，新文件的权限位并不就是根据这个数设置的，还必须参考另一个值：文件权限的掩码，这是一个 9 位的二进制数，对应于 9 个权限位，如果这个数某一位上置 1，则新创建文件相应权限位被强制置 0，而不管所提供的权限位是怎样的。这种机制使的一个新文件在创建时被限制为一个合适的权限。可以用 umask 命令看到这个值，它以 3 位八进制数显示，默认为 022，即 000010010，所以一个新文件创建时，除了文件拥有者外，其他用户是没有写权限的。在 VFS 的 fs_struct 结构中有一个 umask 域，存储的就是这个文件权限的掩码。

文件建立后，文件拥有者可以用 chmod 改变访问权限。每一组权限位的设置可以用一个八进制数表示，这样，用 3 个八进制数就可以表示 9 个权限位了。例如 chmod 755/home/user1/file1，则该文件的权限表示为 wxr-xr-xr。如果要设置 setuid、setgid 位，则需要用到第 4 个八进制数，该数为 4，则 setuid 位被设置；该数为 2，则 setgid 位被设置，该数为 1，则是其他用户的执行位被设置成“ t ”。

Ext2 索引节点中的 i_mode 就是用来存储文件的属性、用户标识号、用户组标识号和访

问权限等信息的，这是一个 16 位的字段，其中 0 到 8 位用来表示文件的访问权限，第 9 位用来控制文件是否驻留内存，10、11 位即 setgid、setuid 位，12 到 15 位的组合用来表示文件类型。如表 9.1 所示。

表 9.1　　i mode 字段各位的含义

位		符号常量	含义
12~15 位的组合	1100	S_IFSOCK	套接字
	1010	S_IFLNK	符号链接文件
	1000	S_IFREG	普通文件
	0110	S_IFBLK	块设备文件
	0100	S_IFDIR	目录
	0010	S_IFCHR	字符设备文件
	0001	S_IFIFO	命名管道
11		S_ISUID	用户
10		S_ISGID	用户组
9		S_ISVTX	文件是否驻留内存
0~8		rwx-rwx-rwx	文件的访问权限

这个表中的符号常量都是在 include/Linux/stat.h 中定义的。

9.4　链接文件

由前面有关索引节点和目录项的介绍，我们已经知道 Ext2 把文件名和文件信息分开存储，其中文件信息用索引节点来描述，目录项就是用来联系文件名和索引节点的。目录项中，每一对文件名和索引节点号的一个一一对应称为一个链接，这就使同一个索引节点号出现在多个链接中成为可能，也就是说，同一个索引节点号可以对应多个不同的文件名。这种链接称为硬链接，可以用 ln 命令为一个已存在的文件建立一个新的硬链接：

```
ln    /home/user1/file1    /home/user1/file2
```

建立了一个文件 file2，链接到 file1 上。file2 和 file1 有相同的索引节点号，也就是和 file1 共享同一个索引节点。在建立了一个新的硬链接后，这个索引节点中的 i_links_count 值将加 1，i_links_count 的值反映了链接到这个索引节点上的文件数。

使用硬链接的好处如下所示。

（1）由于在删除文件时，实际上先对 i_links_count 作减 1，如果 i_links_count 不为 0，则结束，即仅仅删除了一个硬链接，具体文件的数据并没有被删除。只有在 i_links_count 为 0 时，才真正将文件从磁盘上删除。这样，你可以对重要的文件作多个链接，防止文件被误

删除。

（2）允许用户在不进入某个目录的情况下对该目录下面的文件进行处理。

由于同一个文件系统中，索引节点号是系统用来辨认文件的唯一标志，而两个不同的文件系统中，可能有索引节点号一样的文件，所以硬链接仅允许在同一个文件系统上进行，要在多个文件系统之间建立链接，必须用到符号链接。

符号链接与硬链接最大的不同就在于它并不与索引节点建立链接，也就是说当为一个文件建立一个符号链接时，索引节点的链接计数并不变化。当你删除一个文件时，它的符号链接文件也就失去了作用，而当你删去一个文件的符号链接文件，对该文件本身并无影响。所以，有必要区分符号链接文件和硬链接文件，符号链接文件用“ l ”表示，另外，符号链接文件的索引节点号与原文件的索引节点号也是不同的。而硬链接只是普通文件。硬链接的一个缺陷是你无法简单地知道哪些文件是链接到同一个文件上的。而在符号链接中，可以看到它是指向哪个文件的。

最后，硬链接只能由超级用户建立，而普通用户可以建立符号链接。建立符号链接用 ls -s 命令。

因为内核为符号链接文件也创建一个索引节点，但它跟普通文件的索引节点所有不同。如前所述，代表着链接节点的文件没有数据，因此，关于符号链接的操作也就比较简单。对 Ext2 文件系统来说，只有 ext2_readlink() 和 ext2_follow_link() 函数，这是在 fs/ext2/symlink.c 中定义的：

```
struct inode_operations ext2_fast_symlink_inode_operations = {
         readlink:       ext2_readlink,
         follow_link:    ext2_follow_link,
};
ext2_readlink()函数的代码如下：
static int ext2_readlink (struct dentry *dentry, char *buffer, int buflen)
{
         char *s =  (char *) dentry->d_inode->u.ext2_i.i_data;
        return vfs_readlink (dentry, buffer, buflen, s) ;
}
```

如前所述，对于 Ext2 文件系统，连接目标的路径在 ext2_inode_info 结构（即 inode 结构的 union 域）的 i_data 域中存放，因此字符串 s 就存放有连接目标的路径名。vfs_readlink () 的代码在 fs/namei.c 中：

```
int vfs_readlink (struct dentry *dentry, char *buffer, int buflen, const char *link)
{
         int len;

         len = PTR_ERR (link) ;
        if  (IS_ERR (link) )
                 goto out;

         len = strlen (link) ;
         if  (len >  (unsigned)  buflen)
                 len = buflen;
        if  (copy_to_user (buffer, link, len) )
                 len = -EFAULT;
out:
```

```
        return len;
}
```

从代码可以看出，该函数比较简单，即把连接目标的路径名拷贝到用户空间的缓冲区中，并返回路径名的长度。

ext2_follow_link () 函数用于搜索符号连接所在的目标文件，其代码如下：

```
static int ext2_follow_link (struct dentry *dentry, struct nameidata *nd)
{
        char *s =  (char *) dentry->d_inode->u.ext2_i.i_data;
        return vfs_follow_link (nd, s) ;
}
```

这个函数与 ext2_readlink () 类似，值得注意的是，从 ext2_readlink () 中对 vfs_readlink () 的调用意味着从较低的层次（Ext2 文件系统）回到更高的 VFS 层次。为什么呢？这是因为符号链接的目标有可能在另一个不同的文件系统中，因此，必须通过 VFS 来中转，在 vfs_follow_link () 中必须要调用路径搜索函数 link_path_walk()来找到代表着连接对象的 dentry 结构，函数的代码如下：

```
static inline int vfs_follow_link (struct nameidata *nd, const char *link)
{
        int res = 0;
        char *name;
        if  (IS_ERR (link) )
                goto fail;

        if  (*link == '/')  {
                path_release (nd) ;
                if  (!walk_init_root (link, nd) )
                        /* weird __emul_prefix ()  stuff did it */
                        goto out;
        }
        res = link_path_walk (link, nd) ;
out:
        if  (current->link_count || res || nd->last_type!=LAST_NORM)
                return res;
        /*
         * If it is an iterative symlinks resolution in open_namei ()  we
        * have to copy the last component. And all that crap because of
        * bloody create ()  on broken symlinks. Furrfu...
         */
        name = __getname () ;
        if  (!name)
                return -ENOMEM;
        strcpy (name, nd->last.name) ;
        nd->last.name = name;
        return 0;
fail:
        path_release (nd) ;
        return PTR_ERR (link) ;
}
```

其中 nameidata 结构为：

```
 struct nameidata {
```

```
        struct dentry *dentry;
        struct vfsmount *mnt;
         struct qstr last;
        unsigned int flags;
         int last_type;
};
```

last_type 域的可能取值定义于 fs.h 中：

```
/*
* Type of the last component on LOOKUP_PARENT
*/
enum {LAST_NORM, LAST_ROOT, LAST_DOT, LAST_DOTDOT, LAST_BIND};
```

在路径的搜索过程中，这个域的值会随着路径名当前的搜索结果而变。例如，如果成功地找到了目标文件，那么这个域的值就变成了 LAST_NORM；而如果最后停留在一个“ . ”上，则变成 LAST_DOT。

Qstr 结构用来存放路径名中当前节点的名字、长度及哈希值，其定义于 include/linux/dcache.h 中：

```
*
* "quick string" -- eases parameter passing, but more importantly
* saves "metadata" about the string (ie length and the hash) .
*/

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

下面来对 vfs_follow_link () 函数的代码给予说明。

- 如果符号链接的路径名是以“ ／ ”开头的绝对路径，那就要通过 walk_init_root () 从根节点开始查找。
- 调用 link_path_walk () 函数查找符号链接所在目标文件对应的信息。从 link_path_walk()返回时，返回值为 0 表示搜索成功，此时，nameidata 结构中的指针 dentry 指向目标节点的 dentry 结构，指针 mnt 指向目标节点所在设备的安装结构，同时，这个结构中的 last_type 表示最后一个节点的类型，节点名则在类型为 qstr 结构的 last 中。该函数失败时，则函数返回值为一负的出错码，而 nameidata 结构中则提供失败的节点名等信息。
- vfs_follow_link () 返回值的含义与 ink_path_walk () 函数完全相同。

9.5　分配策略

当建立一个新文件或目录时，Ext2 必须决定在磁盘上的什么地方存储数据，也就是说，将哪些物理块分配给这个新文件或目录。一个好的分配物理块的策略，将导致文件系统性能的提高。一个好的思路是将相关的数据尽量存储在磁盘上相邻的区域，以减少磁头的寻道时间。Ext2 使用块组的优越性就体现出来了，因为，同一个组中的逻辑块所对应的物理块通常是相邻存储的。Ext2 企图将每一个新的目录分到它的父目录所在的组，因为在理论上，访问

完父目录后，接着要访问其子目录，例如对一个路径的解析。所以，将父目录和子目录放在同一个组是有必要的。它还企图将文件和它的目录项分在同一个组，因为目录访问常常导致文件访问。当然如果组已满，则文件或目录可能分在某一个未满的组中。

分配新块的算法如下所述。

(1) 文件的数据块尽量和它的索引节点在同一个组中。

(2) 每个文件的数据块尽量连续分配。

(3) 父目录和子目录尽量在一个块组中。

(4) 文件和它的目录项尽量在同一个块组中。

9.5.1 数据块寻址

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

从文件内的偏移量 f 导出相应数据块的逻辑块号需要以下两个步骤。

- 从偏移量 f 导出文件的块号，即偏移量 f 处的字符所在的块索引。
- 把文件的块号转化为相应的逻辑块号。

因为 Linux 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是相当容易的：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 1 个数据块中，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 的数据块中等等。

只用关心文件的块号确实不错。但是，由于 Ext2 文件的数据块在磁盘上并不是相邻的，因此把文件的块号转化为相应的逻辑块号可不是那么直接了当。

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，这种映射也包括一些专门的数据块，可以把这些数据块看成是用来处理大型文件的索引节点的扩展。

磁盘索引节点的 i_block 域是一个有 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，如图 9.4 所示，这个数组表示一个大型数据结构的初始化部分。正如你从图中所看到的，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 到 11。
- 索引 12 中的元素包含一个块的逻辑块号，这个块代表逻辑块号的一个二级数组。这个数组对应的文件块号从 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 做除数）。因此，内核必须先用指向一个块的指针访问这个元素，然后，用另一个指向包含文件最终内容的块的指针访问那个块。
- 索引 13 中的元素包含一个块的逻辑块号，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逻辑块号对应的文件块号，范围从 $b/4+12$ 到 $(b/4)^2+(b/4)+11$。
- 最后，索引 14 中的元素利用了三级间接索引：第四级数组中存放的才是逻辑块号对

应的文件块号，范围从$(b/4)^2+(b/4)+12$ 到$(b/4)^3+(b/4)^2+(b/4)+11$。

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，那么两次访问磁盘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 3～4 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、缓冲区及页高速缓存都有助于极大地减少实际访问磁盘的次数。

也要注意文件系统的块大小是如何影响寻址机制的，因为大的块大小允许 Ext2 把更多的逻辑块号存放在一个单独的块中。表 9.2 显示了对每种块大小和每种寻址方式所存放文件大小的上限。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余的 13KB 到 268KB 的数据。对于 4096 字节的块，两次间接就完全满足了对 2GB 文件的寻址（2GB 是 32 位体系结构上的 Ext2 文件系统所允许的最大值）。

表 9.2　　可寻址的文件数据块大小的界限

块大小	直接	一次间接	二次间接	三次间接
1024	12 KB	268 KB	63.55 MB	2 GB
2048	24 KB	1.02 MB	513.02 MB	2 GB
4096	48 KB	4.04 MB	2 GB	—

9.5.2　文件的洞

文件的洞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 Linux 命令创建了第一个字节是洞的文件。

```
$ echo -n "X" | dd of=/tmp/hole bs=1024 seek=6
```

现在，/tmp/hole 有 6145 个字符（6144 个 NULL 字符加一个 X 字符），然而，这个文件只占磁盘上一个数据块。

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

文件洞在 Ext2 的实现是基于动态数据块的分配：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 域定义程序所看到的文件大小，包括洞，而 i_blocks 域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 文件被创建在块大小为 4096 的 Ext2 分区上。其相应磁盘索引节点的 i_size 域存放的数为 6145，而 i_blocks 域存放的数为 8（因为每 4096 字节的块包含 8 个 512 字节的块）。i_block 数组的第 2 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。

9.5.3　分配一个数据块

当内核要分配一个新的数据块来保存 Ext2 普通文件的数据时，就调用

ext2_get_block( )函数。这个函数依次处理在“数据块寻址”部分所描述的那些数据结构，并在必要时调用 ext2_alloc_block( )函数在 Ext2 分区实际搜索一个空闲的块。

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。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 域存放预分配给某一文件但还没有使用的数据块数，而 i_prealloc_block 域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，即文件被关闭时，文件被删除时，或关于引发块预分配的写操作而言，有一个写操作不是顺序的时候，就释放预分配但一直没有使用的块。

下面我们来看一下 ext2_get_block( )函数，其代码在 fs/ext2/inode.c 中：

```
  /*
   * 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.
   */
static int ext2_get_block (struct inode *inode, long iblock, struct buffer_head *bh_result,
int create)
  {
        int err = -EIO;
        int offsets[4];
        Indirect chain[4];
        Indirect *partial;
        unsigned long goal;
        int left;
        int depth = ext2_block_to_path (inode, iblock, offsets) ;
        if  (depth == 0)
              goto out;

       lock_kernel () ;
reread:
        partial = ext2_get_branch (inode, depth, offsets, chain, &err) ;

        /* Simplest case - block found, no allocation needed */
        if  (!partial)  {
got_it:
              bh_result->b_dev = inode->i_dev;
              bh_result->b_blocknr = le32_to_cpu (chain[depth-1].key) ;
              bh_result->b_state |=  (1UL << BH_Mapped) ;
               /* Clean up and exit */
               partial = chain+depth-1; /* the whole chain */
```

```
                goto cleanup;
        }

        /* Next simple case - plain lookup or failed read of indirect block */
        if (!create || err == -EIO) {
cleanup:
                while (partial > chain) {
                        brelse (partial->bh) ;
                        partial--;
                }
                unlock_kernel () ;
out:
                return err;
        }

        /*
         * Indirect block might be removed by truncate while we were
         * reading it. Handling of that case (forget what we've got and
         * reread) is taken out of the main path.
         */
        if (err == -EAGAIN)
                goto changed;

        if (ext2_find_goal (inode, iblock, chain, partial, &goal) < 0)
                goto changed;

        left = (chain + depth) - partial;
        err = ext2_alloc_branch (inode, left, goal,
                                 offsets+ (partial-chain) , partial) ;
        if (err)
                goto cleanup;

        if (ext2_splice_branch (inode, iblock, chain, partial, left) < 0)
                goto changed;

        bh_result->b_state |= (1UL << BH_New) ;
        goto got_it;

changed:
        while (partial > chain) {
                brelse (partial->bh) ;
                partial--;
        }
        goto reread;
}
```

对这个函数，源代码的作者对分配策略给出了简单的注释，相信读者从会中领略一二。函数的参数 inode 指向文件的 inode 结构；参数 iblock 表示文件中的逻辑块号；参数 bh_result 为指向缓冲区首部的指针，buffer_head 结构已在上一章做了介绍；参数 create 表示是否需要创建。

其中 Indirect 结构在同一文件中定义如下：

```
typedef struct {
        u32     *p;
        u32     key;
        struct buffer_head *bh;
} Indirect
```

用数组 chain[4]描述 4 种不同的索引，即直接索引、一级间接索引、二级间接索引、三级间接索引。举例说明这个结构各个域的含义。如果文件内的块号为 8，则不需要间接索引，所以只用 chain[0]一个 Indirect 结构，p 指向直接索引表下标为 8 处，即 &inode->u.ext2_i.i_data[8]；而 key 则持有该表项的内容，即文件块号所对应的设备上的块号（类似于逻辑页面号与物理页面号的对应关系）；bh 为 NULL，因为没有用于间接索引的块。如果文件内的块号为 20，则需要一次间接索引，索引要用 chian[0]和 chain[1]两个表项。第一个表项 chian[0] 中，指针 bh 仍为 NULL，因为这一层没有用于间接索引的数据块；指针 p 指向 &inode->u.ext2_i.i_data[12]，即间接索引的表项；而 key 持有该项的内容，即对应设备的块号。chain[1]中的指针 bh 则指向进行间接索引的块所在的缓冲区，这个缓冲区的内容就是用作间接索引的一个整数数组，而 p 指向这个数组中下标为 8 处，而 key 则持有该项的内容。这样，根据具体索引的深度 depth，数组 chain[]中的最后一个元素，即 chain[depth-1].key，总是持有目标数据块的物理块号。而从 chain [] 中第 1 个元素 chain [0] 到具体索引的最后一个元素 chain [depth-1] ，则提供了具体索引的整个路径，构成了一条索引链，这也是数据名 chain 的由来。

了解了以上基本内容后，我们来看 ext2_get_block（）函数的具体实现代码。

- 首先调用 ext2_block_to_path（）函数，根据文件内的逻辑块号 iblock 计算出这个数据块落在哪个索引区间，要采用几重索引（1 表示直接）。如果返回值为 0，表示出错，因为文件内块号与设备上块号之间至少也得有一次索引。出错的原因可能是文件内块号太大或为负值。
- ext2_get_branch（）函数深化从 ext2_block_to_path（）所取得的结果，而这合在一起基本上完成了从文件内块号到设备上块号的映射。从 ext2_get_branch（）返回的值有两种可能。一是，如果顺利完成了映射则返回值为 NULL。二是，如果在某一索引级发现索引表内的相应表项为 0，则说明这个数据块原来并不存在，现在因为写操作而需要扩充文件的大小。此时，返回指向 Indirect 结构的指针，表示映射在此断裂。此外，如果映射的过程中出错，例如，读数据块失败，则通过 err 返回一个出错代码。
- 如果顺利完成了映射，就把所得结果填入缓冲区结构 bh_result 中，然后把映射过程中读入的缓冲区（用于间接索引）全部释放。
- 可是，如果 ext2_get_branch（）返回一个非 0 指针，那就说明映射在某一索引级上断裂了。根据映射的深度和断裂的位置，这个数据块也许是个用于间接索引的数据块，也许是最终的数据块。不管怎样，此时都应该为相应的数据块分配空间。
- 要分配空间，首先应该确定从物理设备上何处读取目标块。根据分配算法，所分配的数据块应该与上一次已分配的数据块在设备上连续存放。为此目的，在 ext2_inode_info 结构中设置了两个域 i_next_alloc_block 和 i_next_alloc_goal。前者用来记录下一次要分配的文件内块号，而后者则用来记录希望下一次能分配的设备上的块号。在正常情况下，对文件的扩充是顺序的，因此，每次所分配的文件内块号都与前一次的连续，而理想上来说，设

备上的块号也同样连续，二者平行地向前推进。这种理想的“建议块号”就是由 ext2_find_goal（）函数来找的。

- 设备上具体物理块的分配，以及文件内数据块与物理块之间映射的建立，都是调用 ext2_alloc_branch（）函数完成的。调用之前，先要算出还有几级索引需要建立。
- 从 ext2_alloc_branch（）返回以后，我们已经从设备上分配了所需的数据块，包括用于间接索引的中间数据块。但是，原先映射开始断开的最高层上所分配的数据块号只是记录了其 Indirect 结构中的 key 域，却并没有写入相应的索引表中。现在，就要把断开的“树枝”接到整个索引树上，同时，还需要对文件所属 inode 结构中的有关内容做一些调整。这些操作都是由 ext2_splice_branch（）函数完成。

到此为止，万事具备，则转到标号 got_it 处，把映射后的数据块连同设备号置入 bh_result 所指的缓冲区结构中，这就完成了数据块的分配。