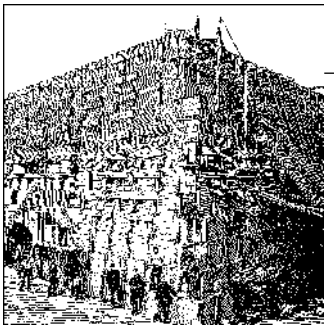

第十七章

Ext2 和 Ext3 文件系统

在本章，我们通过当与一个特定的文件系统交互时内核所关注的细节来结束对I/O和文件系统进行的广泛讨论。因为第二扩展文件系统（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，其他文件系统将包含新而有趣的需要，但它们是为其他操作系统设计的，因此我们在本书中不对各种文件系统和各种平台所具有的特性进行考察。

在“Ext2的一般特征”一节中引入Ext2后，会像其他章节一样，接着描述所需的数据结构。因为我们着眼于磁盘上存放数据的特定方式，所以必须考虑两种形式的数据结构：“磁盘数据结构”一节说明把Ext2存放在磁盘上的数据结构，而“内存数据结构”一节说明如何把磁盘上的数据结构复制到内存中。

然后，我们讨论Ext2文件系统上所执行的操作。在“创建文件系统”一节，我们讨论如何在磁盘分区创建Ext2。接下来的一节描述使用磁盘时内核所执行的操作。其中的大部分操作是为索引节点和数据块分配磁盘空间，这些操作相对比较低级。

在最后一节，我们将对Ext3文件系统给予简短描述，它是Ext2文件系统的改进版。

Ext2的一般特征

类Unix操作系统使用多个文件系统。尽管所有这些文件系统的文件都有少数POSIX API（如stat()）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（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（second Extended Filesystem，Ext2）；它除了包含几个新的特点外，还相当高效和强健，已成为广泛使用的Linux文件系统。

下列特点有助于 Ext2 的效率：

- 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块大小（从 1024 到 4096 字节）。例如，当文件的平均长度小于几千字节时，块的大小为 1024 字节是最佳的，因为这会产生较少的内部碎片，也就是文件长度与存放它的磁盘分区有较少的不匹配（参见第七章中的“内存区管理”一节，在那里讨论了动态内存的内部碎片）。另一方面，大的块对于大于几千字节的文件通常比较合适，因为这样的磁盘传送较少，因而减轻了系统的开销。
- 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。
- Ext2 文件系统把磁盘块分为组。每组包含存放在相邻磁道的数据块和索引节点。正是这种结构，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进行访问。
- 在磁盘数据块被实际使用之前，Ext2 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。
- 支持快速符号链接。如果符号链接的路径名（参见第一章中的“硬链接和软链接”一节）小于或等于 60 字节，就把它存放在索引节点中而不用通过读一个数据块进行转换。

此外，Ext2 还包含了一些使它既强健又灵活的特点：

- 文件更新策略的谨慎实现将系统崩溃的影响减到最少。例如，当给文件创建一个新的硬链接时，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错误，就会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，但并不会导致灾难性的后果。如果这种操作的顺序相反（更新索引节点前改变目录），同样的硬错误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，那么向这个旧目录项的写操作将毁坏这个新的文件。
- 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。

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

Ext2 文件系统是一种成熟、稳定的程序，近几年没有多大变化。不过，人们已经考虑引入几个另外的特性。一些特性已被实现并以外部补丁的形式来使用。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。

访问控制表（Access Control List，ACL）

访问控制表不是把一个文件的用户分为三类（文件主、同组用户、其他用户），而是对任何特定的用户或用户组，让每个文件与特定的存取权限相关联。

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

日志避免文件系统被突然卸载（例如，由于系统崩溃的结果）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包含在 Ext2 文件系统中。有人可能说 Ext2 是这种成功的牺牲品；但它仍然是大多数 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没有经过严格测试和广泛使用的文件系统来代替 Ext2 的任何企图产生质疑。

这种现象的显而易见的例子是日志，日志是高可用服务器所需的最引人注目的特点。日志还没有引入到 Ext2 文件系统；相反，我们在后面“Ext3 文件系统”一节会讨论，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。

Ext2 磁盘数据结构

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启动扇区所保留的（参见附录一）。Ext2 分区的其余部分分成块组（block group），每个块组的分布图如图 17-1 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。

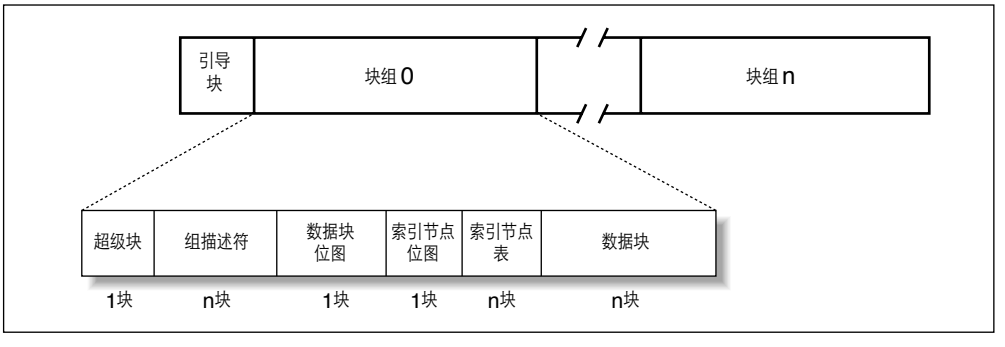

图 17-1：Ext2 分区和 Ext2 块组的分布图

由于内核尽可能地把属于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的碎片。块组中的每个块包含下列信息之一：

- 文件系统超级块的一个拷贝
- 一组块组描述符的拷贝
- 一个数据块位图 块组中对应的块是否被占用，指向8*b个块
- 一组索引节点
- 一个索引节点位图
- 属于文件的一大块数据；即一个数据块

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。

从图 17-1 中可以看出，超级块与组描述符被复制到每个块组中。只有块组 0 中所包含的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。

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，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。因此，块组的总数大约是 $s/(8\times b)$，这里 *s* 是分区的总块数。

4KB居多
8倍 1byte=8bit

举例说明，让我们考虑一下 8 GB 的 Ext2 分区，块的大小为 4KB。在这种情况下，每个 4KB 的块位图描述 32K 个数据块，即 128MB。因此，最多需要 64 个块组。显然，块的大小越小，块组数越大。 都在一个块组里

超级块

Ext2 在磁盘上的超级块存放在一个 `ext2_super_block` 结构中，它的字段在表 17-1 中列出。`__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`、`__s32` 数据类型表示长度为 8、16 及 32 位的有符号数。

表 17-1：Ext2 超级块的字段

类型	字段	描述
`__u32`	`s_inodes_count`	索引节点的总数
`__u32`	`s_blocks_count`	以块为单位的文件系统的大小
`__u32`	`s_r_blocks_count`	保留的块数
`__u32`	`s_free_blocks_count`	空闲块计数器
`__u32`	`s_free_inodes_count`	空闲索引节点计数器
`__u32`	`s_first_data_block`	第一个使用的块号（总为 1）
`__u32`	`s_log_block_size`	块的大小
`__s32`	`s_log_frag_size`	片的大小
`__u32`	`s_blocks_per_group`	每组中的块数
`__u32`	`s_frags_per_group`	每组中的片数
`__u32`	`s_inodes_per_group`	每组中的节点数

表 17-1：Ext2 超级块的字段（续）

类型	字段	描述
__u32	s_mtime	最后一次安装操作的时间
__u32	s_wtime	最后一次写操作的时间
__u16	s_mnt_count	安装操作计数器
__u16	s_max_mnt_count	检查之前安装操作的次数
__u16	s_magic	魔数签名
__u16	s_state	状态标志
__u16	s_errors	当检测到错误时的行为
__u16	s_minor_rev_level	次版本号
__u32	s_lastcheck	最后一次检查的时间
__u32	s_checkinterval	两次检查之间的时间间隔
__u32	s_creator_os	创建文件系统的操作系统
__u32	s_rev_level	版本号
__u16	s_def_resuid	保留块的缺省 UID
__u16	s_def_resgid	保留块的缺省 GID
__u32	s_first_ino	第一个非保留的索引节点号
__u16	s_inode_size	磁盘上索引节点结构的大小
__u16	s_block_group_nr	这个超级块的块组号
__u32	s_feature_compat	具有兼容特点的位图
__u32	s_feature_incompat	具有非兼容特点的位图
__u32	s_feature_ro_compat	只读兼容特点的位图
__u8 [16]	s_uuid	128 位文件系统标识符
char [16]	s_volume_name	卷名
char [64]	s_last_mounted	最后一个安装点的路径名
__u32	s_algorithm_usage_bitmap	用于压缩
__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 [1024]	s_reserved	用 null 填充 1024 字节

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，等等。目前s_log_frag_size字段与s_log_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，如果被全部卸载，则这个字段的值为1，如果包含错误则值为2。

组描述符和位图

每个块组有自己的组描述符，即ext2_group_desc结构，它的字段在表17-2中列出。

表17-2：Ext2组描述符的字段

类型	字段	描述
__u32	bg_block_bitmap	块位图的块号
__u32	bg_inode_bitmap	索引节点位图的块号
__u32	bg_inode_table	第一个索引节点表块的块号
__u16	bg_free_blocks_count	组中空闲块的个数
__u16	bg_free_inodes_count	组中索引节点的个数
__u16	bg_used_dirs_count	组中目录的个数
__u16	bg_pad	按字对齐
__u32 [3]	bg_reserved	用null填充24个字节

256位

n是多少？*组个数/块大小？

block number不是record number

是绝对地址 (64位?)

当分配新节点和数据块时用到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。位图是位的序列，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

1byte=8bit

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，因此，一个单独的位图描述 8192、16384 或 32768 个块的状态。

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。索引节点表第一个块的块号存放在组描述符的 bg_inode_table 字段中。

256字节
4096/256=16
计算n

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）除以每块中的索引节点数。 8192/16=512

每个 Ext2 索引节点是一个 ext2_inode 结构，它的字段在表 17-3 中列出。

表 17-3：Ext2 磁盘索引节点的字段

类型	字段	描述
__u16	i_mode	文件类型和访问权限
__u16	i_uid	拥有者标识符
__u32	i_size	以字节为单位的文件长度
__u32	i_atime	最后一次访问文件的时间
__u32	i_ctime	索引节点最后改变的时间
__u32	i_mtime	文件内容最后改变的时间
__u32	i_dtime	文件删除的时间
__u16	i_gid	组标识符
__u16	i_links_count	硬链接计数器
__u32	i_blocks	文件的数据块数
__u32	i_flags	文件标志
union	osd1	特定的操作系统信息
__u32 [EXT2_N_BLOCKS]	i_block	指向数据块的指针
__u32	i_version	文件版本（当网络文件系统访问文件时使用）
__u32	i_file_acl	文件访问控制表
__u32	i_dir_acl	目录访问控制表
__u32	i_faddr	片的地址
union	osd2	特定的操作系统信息

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，这已在第十二章的“索引节点对象”一节中讨论过。其余的字段与 Ext2 的特殊实现相关，主要处理块的分配。

特别的是，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收一个数据块（因为还没实现片）且i_size可能小于512×i_blocks。另一方面，我们将在本章后面的“文件的洞”一节中看到，一个文件可能包含有洞。在那种情况下，i_size可能大于 512×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（参见本章后面的“数据块寻址”一节）。

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”，允许像惠普的 Alpha 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 位版本有点兼容，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（参见第十二章的“open()系统调用”一节）。

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。正如你看到的，索引节点号是 Ext2 例程用来快速搜索磁盘上正确的索引节点描述符的一个关键。

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。这一节讨论每种文件类型的存储要求，如表 17-4 所示。

表 17-4：Ext2 文件类型

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

普通文件

普通文件是最常见的情况，本章主要关注它。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 string >*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以一种特殊类型的文件实现了目录，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。特别的是，这样的数据块包含了类型为 `ext2_dir_entry_2`的结构。表 17-5列出了这个结构的字段。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`）填充文件名的末尾。`name_len` 字段存放实际的文件名长度（参见图 17-2）。

表 17-5：Ext2 目录项中的字段

类型	字段	描述
`__u32`	`inode`	索引节点号
`__u16`	`rec_len`	目录项长度
`__u8`	`name_len`	文件名长度
`__u8`	`file_type`	文件类型
`char [EXT2_NAME_LEN]`	`name`	文件名

file_type字段存放指定文件类型的值（见表17-4）。rec_len字段可以被解释为指向下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项的起始地址。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。仔细看一下图17-2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）。

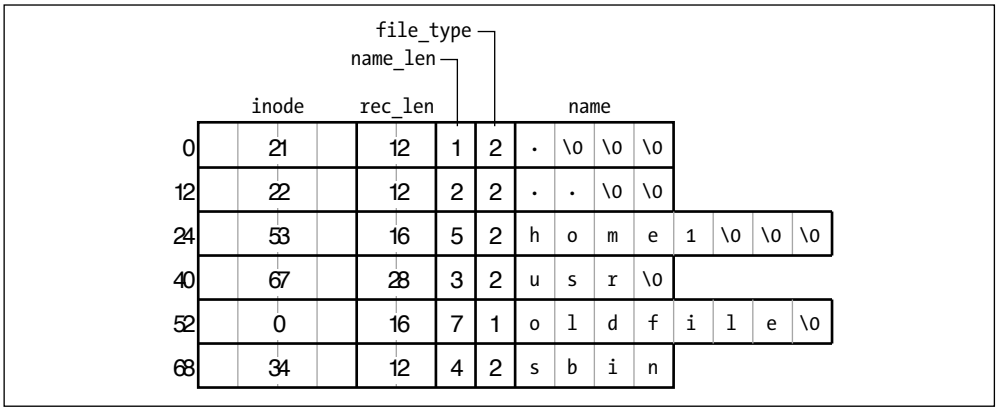

图17-2：EXT2目录的一个例子

符号链接

如前所述，如果符号链接的路径名达到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的内存数据结构

为了提高效率，当安装Ext2文件系统时，存放在Ext2分区的磁盘数据结构中的大部分信息被拷贝到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：

- 当一个新文件被创建时，必须减少Ext2超级块中s_free_inodes_count字段的值和适当的组描述符bg_free_inodes_count字段的值。

- 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。
- 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 分区的块中，因此，内核利用缓冲区高速缓存和页高速缓存来保持它们最新（参见第十四章中的“把脏缓冲区写入磁盘”一节）。

对于与 Ext2 文件系统以及文件相关的每种数据类型，表 17-6 详细说明了在磁盘上用来表示数据的数据结构、在内存中内核所使用的数据结构以及单凭经验来决定使用多大容量的高速缓存。频繁更新的数据总需要缓存，也就是说，这些数据一直存放在内存并包含在缓冲区高速缓存和页高速缓存中，直到相应的Ext2分区被卸载。内核通过让缓冲区的引用计数器一直大于 0 来达到此目的。

表 17-6：Ext2 数据结构的 VFS 映像

类型	磁盘数据结构	内存数据结构	缓存模式
超级块	ext2_super_block	ext2_sb_info	总是缓存
组描述符	ext2_group_desc	ext2_group_desc	总是缓存
块位图	块中的位数组	缓冲区中的位数组	固定限制
索引节点位图	块中的位数组	缓冲区中的位数组	固定限制
索引节点	ext2_inode	ext2_inode_info	动态
数据块	未指定	缓冲区页	动态
空闲索引节点	ext2_inode	无	从不缓存
空闲块	未指定	无	从不缓存

在任何高速缓存中不保存“从不缓存”的数据，因为这种数据表示无意义的信息。

在这些极端的模式中存在另外两种模式：固定限制和动态模式。在固定限制模式中，指定数量的数据结构保存在缓冲区高速缓存中；当超过这个数时，老的数据结构被刷新到磁盘。在动态模式中，只要相关的对象（索引节点或块）正在使用，其数据就保存在缓冲区高速缓存中；当相应的文件被关闭或块被删除时，shrink_caches()函数就从高速缓存中删除相关的数据。

ext2_sb_info 和 ext2_inode_info 结构

当 Ext2 文件系统被安装时，用类型为 ext2_sb_info 的结构装入 VFS 超级块的 u 字段（包含具体文件系统的数据），以便内核能找出与这个文件系统相关的内容。这个结构包含下列信息：

- 磁盘超级块字段的大部分内容
- 块位图高速缓存，由 s_inode_bitmap 和 s_inode_bitmap_number 数组记录（参见下一节）
- 索引节点位图高速缓存，由 s_inode_bitmap 和 s_inode_bitmap_number 数组记录（参见下一节）
- s_sbh 指针，指向磁盘超级块所在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，被压缩在一个块中
- s_group_desc 指针，指向组描述符所在的缓冲区的缓冲区首部的数组（通常，一个单独项就足够了）
- 其他与安装状态、安装选项等相关的数据

同样地，当属于 Ext2 文件的索引节点对象被初始化时，用 ext2_inode_info 类型的结构装载 u 字段，该结构包含下列信息：

- 在磁盘索引节点结构中而不在一般的 VFS 索引节点对象中的大部分字段（参见第十二章中的表 12-3）
- 片的大小和片数（还未使用）
- 索引节点所在块组的 i_block_group 块组索引（参见本章前面的“磁盘数据结构”一节）
- i_prealloc_block 和 i_prealloc_count 字段，在为数据块进行预分配中使用（参见本章后面的“分配数据块”一节）
- i_osync 字段，是一个标志，表示是否同步地更新磁盘索引节点

位图高速缓存

当安装一个 Ext2 文件系统时，内核给 Ext2 磁盘超级块分配一个缓冲区并从磁盘读取超级块的内容。只有当 Ext2 文件系统被卸载时才释放这个缓冲区。当内核必须修改 Ext2

超级块中的一个字段时，只是把新值写进相应缓冲区合适的位置，然后把这个缓冲区标记为“脏”。

遗憾的是，这种方法并不适合所有的Ext2磁盘数据结构。近年来磁盘容量的数十倍增加导致了索引节点的大小和数据块位图的数十倍增加，因此，我们已经处于这样一种状况，即把所有的位图同时保存在RAM中不再方便了。

例如，考虑4GB的磁盘，块的大小为1KB。因为每个位图填满一个单独块的所有位，因此其中的每个位图都描述8192个块的状态，即8MB的磁盘空间。块组数为4096 MB/8 MB=512个。因为每个块组既需要一个索引节点位图还需要一个数据块位图，因此在内存中存放所有的这1024个位图将需要1MB的RAM!

对任一安装的Ext2文件系统，限制Ext2描述符对内存需要所采用的解决方法就是使用大小为EXT2_MAX_GROUP_LOADED（通常为8）的两个高速缓存。一个高速缓存存放大部分最近访问的索引节点位图，而另一个高速缓存存放大部分最近访问的块位图。包含在某个高速缓存中的位图所在的缓冲区有一个大于0的引用计数器，因此shrink_mmap()从不释放这些缓冲区（参见第十六章中的“回收页框”一节）。相反，不在位图高速缓存中的位图所在的缓冲区有一个空的引用计数器，如果空闲内存变得紧缺时，就释放这些缓冲区。

每个高速缓存的实现都是通过有EXT2_MAX_GROUP_LOADED个元素的两个数组进行的。一个数组包含块组位图当前在高速缓存中的块组的索引，而另一个数组包含涉及这些位图的缓冲区首部指针。

ext2_sb_info结构存放属于索引节点位图高速缓存的数组；在s_inode_bitmap字段找到块组的索引，在s_inode_bitmap_number字段找到指向缓冲区首部的指针。块位图高速缓存的相应数组存放在s_block_bitmap和s_block_bitmap_number字段。

load_inode_bitmap()函数装入一个指定块组的索引节点位图，并返回找到的位图在高速缓存中的位置。

如果位图已不在位图高速缓存中，load_inode_bitmap()就调用read_inode_bitmap()。后一个函数从块组描述符的bg_inode_bitmap字段中获得包含这个位图的块号，然后调用bread()分配一个新的缓冲区，如果这个块已不在缓冲区高速缓存中就从磁盘读它。

如果Ext2分区的块组数小于或等于EXT2_MAX_GROUP_LOADED，把这个位图插入到高速缓存数组的某一位置，这个位置的索引总要与作为参数传递给load_inode_bitmap()函数的块组索引相匹配。

否则，如果块组数比高速缓存中的位置多，在必要时采用最近最少使用（LRU）策略从高速缓存中删除一个位图，并把需要的位图插入到高速缓存的第一个位置。图 17-3 显示了要引用块组 5 的三种可能的情况：请求的位图已在高速缓存中、位图不在高速缓存中但有空闲位置以及位图不在高速缓存中也没有空闲位置。

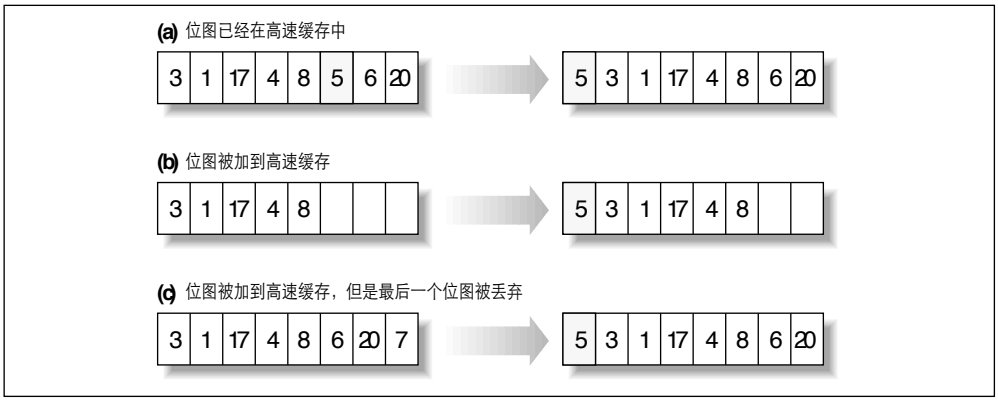

图 17-3：给高速缓存增加一个位图

函数 `load_block_bitmap()`和`read_block_bitmap()`非常类似于 `load_inode_bitmap()`和`read_inode_bitmap()`，但它们指的是 Ext2 分区的块位图高速缓存。

图 17-4 说明了一个安装的 Ext2 文件系统的内存数据结构。在我们的例子中，有三个块组，其描述符存放在磁盘的三个块中。因此，`ext2_sb_info`的`s_group_desc`字段指向由这三个缓冲区首部组成的一个数组。尽管内核可以在位图高速缓存中保存 2×`EXT2_MAX_GROUP_LOADED`个位图，甚至可以在缓冲区高速缓存中存放更多的位图，但我们仅仅显示了下标 2 的索引节点位图和下标 4 的块位图。

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读和写磁盘上的块。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* 实用程序对软盘进行格式化。第二步才涉及创建文件系统，这意味着建立本章前面详细描述的结构。

Ext2 文件系统是由实用程序 *mke2fs* 创建的。*mke2fs* 采用下列默认选项，用户可以用命令行的标志修改这些选项：

- 块大小：1024 字节

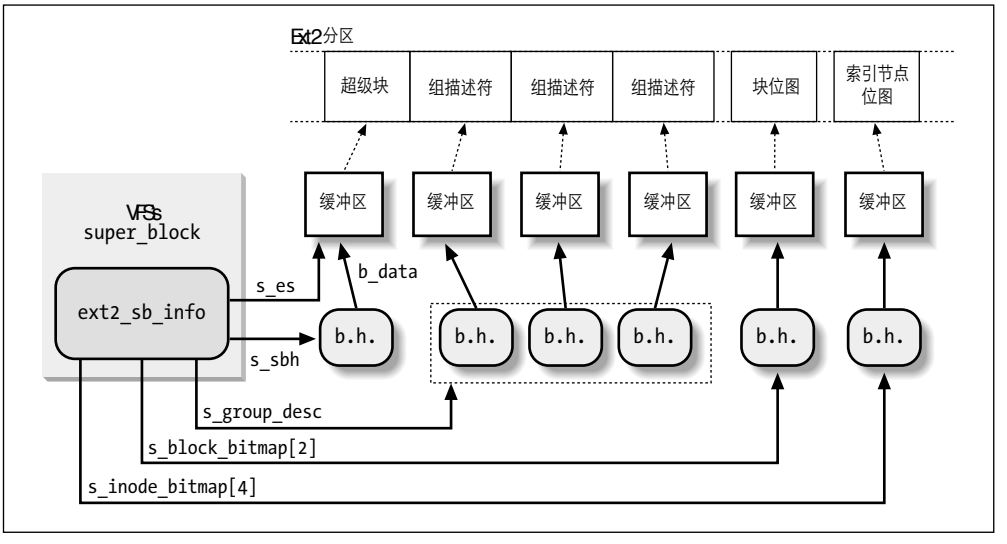

图 17-4：Ext2 的内存数据结构

- 片大小：块的大小（块的分片还没有实现）
- 所分配的索引节点个数：每4096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. 初始化超级块描述符和组描述符。
2. 可选地，检查分区是否包含有缺陷的块，如果有，创建一个有缺陷块的链接表。
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的块和有缺陷的块连接起来。
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 1.4 MB 软盘的。

软盘一旦被安装，VFS 就把它看作由 1390 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：

```
$ dd if=/dev/fd0 bs=1k count=1440 | od -tx1 -Ax > /tmp/dump_hex
```

从而获得了 */tmp* 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（注 1）。

通过查看 dump_hex文件我们可以看到，由于软盘有限的容量，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%），并且根据默认选项，索引节点表必须为每 4096 个字节设置一个索引节点，也就是 360 个索引节点存放在 45 个块中。

表 17-7 概述了按默认选项如何在软盘上建立 Ext2 文件系统。

表 17-7：软盘的 Ext2 块分配

块	内容
0	引导块
1	超级块
2	包含一个单独的块组描述符的块
3	数据块位图
4	索引节点位图
5~49	索引节点表：5~10: 保留； 11：*lost+found*； 12~360：空闲
50	根目录（包括 “ .”、 “..” 及 “*lost+found*”）
51	*lost+found* 目录（包括 “ .” 及 “ ..”）
52~62	给 *lost+found* 目录预分配保留的块
63~1439	空闲块

Ext2 的方法

在第十二章所描述的关于 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描述将需要整整一本书，所以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，就应当能理解实现这些方法的Ext2函数的代码。

注 1： 使用 *dumpesfs* 和 *debugfs* 实用程序也可以获得有关 Ext2 文件系统的一些信息。

Ext2 超级块的操作

很多VFS超级块操作在Ext2中都有具体实现，这些方法为read_inode，write_inode，put_inode，delete_inode，put_super，write_super，statfs及remount_fs。超级块方法的地址存放在 ext2_sops 指针数组中。

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。

如果索引节点指的是普通文件，则在ext2_file_inode_operations表中列出的所有操作都为 NULL 指针，不过，由 ext2_truncate()函数实现的截断操作例外。回想一下，当 Ext2 的相应方法没有定义时（NULL指针），VFS 就使用自己的通用函数。

如果索引节点指的是一个目录，则在ext2_dir_inode_operations表中列出的大多数索引节点操作都是由具体的 Ext2 函数实现的（见表 17-8）。

表 17-8：目录文件的 Ext2 索引节点的操作

VFS 索引节点操作	Ext2 目录索引节点的方法
create	ext2_create()
lookup	ext2_lookup()
link	ext2_link()
unlink	ext2_unlink()
symlink	ext2_symlink()
mkdir	ext2_mkdir()
rmdir	ext2_rmdir()
create	ext2_create()
mknod	ext2_mknod()
rename	ext2_rename()

如果索引节点指的是一个符号链接，而这种符号链接可以完全存放在索引节点本身，那么除了readlink和follow_link以外其他方法都为空，这两个方法是分别通过ext2_readlink()和ext2_follow_link()实现的。这些方法的地址存放在ext2_symlink_inode_operations表中。另一方面，如果索引节点指的是一个长的符号链接，这种符号链接必须存放在一个数据块内，那么，readlink和follow_link方法是通过通用的

page_readlink()和page_follow_link()函数实现的，这些函数的地址存放在page_symlink_inode_operations表中。

如果索引节点指的是一个字符设备文件、块设备文件或命名管道（参见第十九章中的“FIFO”一节），那么这种索引节点的操作不依赖于文件系统，其操作分别位于chrdev_inode_operations、blkdev_inode_operations和fifo_inode_operations表中。

Ext2 的文件操作

表17-9列出了Ext2文件系统特定的文件操作。正如你看到的，VFS方法的read和mmap是由很多文件系统共用的通用函数实现的。这些方法的地址存放在ext2_file_operations表中。

表 17-9：Ext2 的文件操作

VFS 的文件操作	Ext2 的方法
llseek	ext2_file_llseek()
read	generic_file_read()
write	ext2_file_write()
ioctl	ext2_ioctl()
mmap	generic_file_mmap()
open	ext2_file_open()
release	ext2_release_file()
fsync	ext2_sync_file()

注意，Ext2 的 read 和 write 方法是分别通过 generic_file_read()和generic_file_write()函数实现的。这两个函数在第十五章的“从文件读取”和“写入文件”两节中进行了描述。

管理 Ext2 磁盘空间

文件在磁盘的存储不同于程序员所看到的文件，这表现在两个方面：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时间），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）。

在这一节，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：

- 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、不相邻的磁盘块上。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（注2）。这个问题类似于在第七章的“伙伴系统算法”一节中所讨论的RAM的外部碎片问题。
- 空间管理必须考虑时效性，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存放的寻址表的访问次数，因为对该表的立即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`）。它作用于两个参数：`dir`，一个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；`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。该函数执行如下操作：

1. 调用`new_inode()`分配一个新的索引节点对象，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。
2. 对包含在父超级块中的`s_lock`信号量调用`down()`。我们知道，如果信号量处于忙状态，则内核挂起当前进程。
3. 如果新的索引节点是一个目录，应该尽力安排这个新节点以通过部分地填充块组而使目录都能均匀分散地存放。具体来说，在空闲索引节点数大于平均值（平均值是空闲索引节点总数除以块组的个数）的所有块组中，找一个空闲块最多的块组，在这个块组中分配新目录。
4. 如果新的索引节点不是目录，就在有空闲索引节点的块组中给它分配。块组的选择是从包含父目录的块组开始，一直查找下去，具体如下：
   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)，……

注2：请注意，把一个文件跨过块组进行分片是一件坏事，而为了在一个块中存放多个文件把块进行分片（还没实现）是一件好事，这二者之间是不同的。

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。

5. 调用load_inode_bitmap()得到所选块组的索引节点位图，并从中寻找第一个空位，这样就得到了第一个空闲磁盘索引节点号。

6. 分配磁盘索引节点：把索引节点位图中的相应位置位，并把含有这个位图的缓冲区标记为脏。此外，如果文件系统安装时指定了MS_SYNCHRONOUS标志，则调用ll_rw_block()并等待，直到写操作终止（参见第十二章中的“安装一个普通的文件系统”一节）。

7. 把组描述符的bg_free_inodes_count字段减1。如果新的索引节点是一个目录，则增加 bg_used_dirs_count 字段。把含有这个组描述符的缓冲区标记为脏。

8. 把磁盘超级块的s_free_inodes_count字段减1，并把包含它的缓冲区标记为脏。把 VFS 超级块对象的 s_dirt 字段置 1。

9. 初始化这个索引节点对象的字段。具体地说，设置索引节点号 i_ino，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。关于这些字段的含义请参考表 17-3。

10. 把新的索引节点对象插入到inode_hashtable，并调用mark_inode_dirty()把这个索引节点对象移到超级块的脏索引节点的链接表中（参见第十二章“索引节点对象”一节）。

11. 对包含在父超级块中的 s_lock信号量调用 up()。

12. 返回新索引节点对象的地址。

删除索引节点

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（参见本章后面“释放数据块”一节）。函数执行下列操作：

1. 对包含在父超级块中的 s_lock 信号量调用 down()以获得对超级块的互斥访问。

2. 调用 clear_inode()以执行下列操作：

a. 调用 invalidate_inode_buffers()从 i_dirty_buffers 和 i_dirty_data_buffers 链接表中删除属于这个索引节点的脏缓冲区（参见第十四章的“缓冲区首部数据结构”一节）。

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 I/O 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 文件系统没有定义这个方法。

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

3. 从索引节点号计算包含这个磁盘索引节点的块组的索引和每个块组的索引节点数。

4. 调用 oad_inode_bitmap()获得索引节点位图。

5. 把组描述符的 bg_free_inodes_count 字段加 1。如果删除的索引节点是一个目录，也要把 bg_used_dirs_count 字段减 1。把这个组描述符所在的缓冲区标记为脏。

6. 把磁盘超级块的 s_free_inodes_count 字段加 1，并把超级块所在的缓冲区标记为脏。把超级块对象的 s_dirt 字段置为 1。

7. 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，调用 ll_rw_block()并等待，直到在位图缓冲区上的写操作终止。

8. 对包含在父超级块中的 s_lock 信号量调用 up()。

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，在第十三章的“缓冲区首部”一节中做了解释）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是相当容易的，只需用 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。

只用关注文件的块号确实不错。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简单的。

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` 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。如图 17-5 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。
- 下标 12 中的元素包含一个块的逻辑块号，这个块表示逻辑块号的一个二级数组。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 做除数）。因此，内核为了查找指向数据块的指针必须先访问这个元素，然后，用另一个指向最终块（包含文件内容）的指针访问那个块。
- 下标 13 中的元素包含一个块的逻辑块号，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对应的逻辑块号，范围从 $b/4+12$ 到 $(b/4)2+(b/4)+11$。
- 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 $(b/4)^3+(b/4)^2+(b/4)+11$。

在图 17-5 中，块内的个数表示相应的文件块数。箭头（表示存放在数组元素中的逻辑块号）指示了内核如何找到包含文件实际内容的那个块。

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，因为索引节点、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。

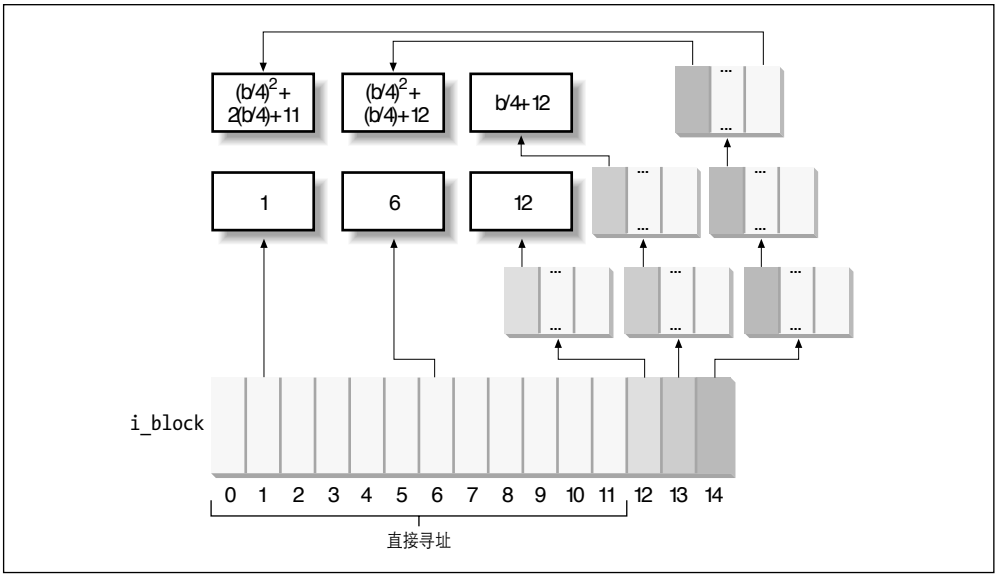

图 17-5：对文件的数据块进行寻址的数据结构

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。表17-10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13到268KB的数据。在32位体系结构上打开大于2GB的大型文件必须指定O_LARGEFILE打开标志。在任何情况下，Ext2文件系统都把文件大小的上限置为2TB减4096字节。

表 17-10：数据块寻址的文件大小上界

块大小	直接	一次间接	二次间接	三次间接
1024	12 KB	268 KB	64.26 MB	16.06 GB
2048	24 KB	1.02 MB	513.02 MB	256.5 GB
4096	48 KB	4.04 MB	4 GB	~2TB

文件的洞

文件的洞是普通文件的一部分，它包含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。

```
$ 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，用于在文件上进行散列的所有应用。

文件洞在Ext2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 字节的块包含 8 个 512 字节的块）。`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（参见图 17-6）。

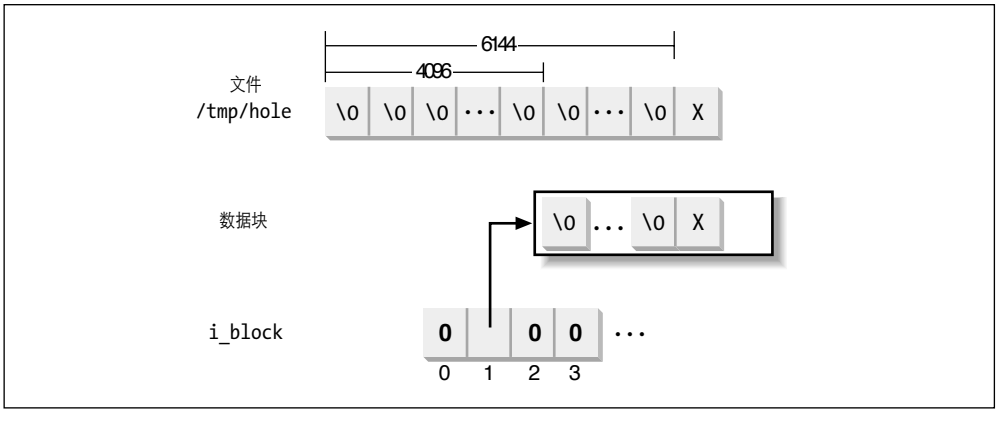

图 17-6：起始部分有洞的文件

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（参见第十五章“从文件读取”和“写入文件”两节）。

`ext2_get_block()`函数处理在“数据块寻址”一节描述的数据结构，并在必要时调用 `ext2_alloc_block()`函数在 Ext2 分区实际地搜索一个空闲块。

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字段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，释放预分配但一直没有使用的块：当文件被关闭时，当文件被截断时，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和目标（goal）。目标是一个逻辑块号，表示新块的首选位置。ext2_getblk()函数根据下列的试探法设置目标参数：

1. 如果正被分配的块与前面已分配的块有连续的文件块号，则目标就是前一块的逻辑块号加1。这很有意义，因为程序所看到的连续的块在磁盘上将会是相邻的。
2. 如果第一条规则不适用，并且至少给文件已分配了一个块，那么目标就是这些块的逻辑块号中的一个。更确切地说，目标是已分配块的逻辑块号，位于文件中待分配块之前。
3. 如果前面的规则都不适用，那么目标就是文件索引节点所在的块组中第一个块的逻辑块号（不必空闲）。

ext2_alloc_block()函数检查目标是否指向文件的预分配块中的一块。如果是，就分配相应的块并返回它的逻辑块号；否则，丢弃所有剩余的预分配块并调用ext2_new_block()。

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）是空闲的，就分配它。
2. 如果目标为忙，检查首选块后的64个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：
   a. 寻找至少有8个相邻空闲块的一组块。
   b. 如果没有找到这样的一组块，就寻找一个单独的空闲块。

只要找到一个空闲块，搜索就结束。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个空闲块作预分配，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。

释放数据块

当进程删除一个文件或把它的长度截为 0 时，其所有数据块必须收回。这是通过调用 ext2_truncate()函数（其参数是这个文件的索引节点对象的地址）来完成的。实际上，这个函数扫描磁盘索引节点的i_block数组以确定所有数据块的位置和用作间接寻址的所有块的位置。然后反复调用 ext2_free_blocks()函数释放这些块。

ext2_free_blocks()函数释放一组含有一个或多个相邻块的数据块。除 ext2_truncate()调用它外，当丢弃文件的预分配块时也主要调用它（参见前面的“分配数据块”一节）。函数参数如下：

inode
: 对文件进行描述的索引节点对象的地址

block
: 要释放的第一个块的逻辑块号

count
: 要释放的相邻块数

这个函数在超级块的 s_lock 信号量上调用 down()以获得对 Ext2 文件系统超级块的互斥访问，然后对每个要释放的块执行下列操作：

1. 获得要释放块所在块组的块位图
2. 把块位图中要释放的块的对应位清 0，并把位图所在的缓冲区标记为脏
3. 把块组描述符的 bg_free_blocks_count 字段加 1，并把相应的缓冲区标记为脏
4. 把超级块的 s_free_blocks_count 字段加 1，并把相应的缓冲区标记为脏，设置超级块对象的 s_dirt 标记
5. 如果 Ext2 文件系统安装时设置了 MS_SYNCHRONOUS 标志，则调用 ll_rw_block()并等待，直到对这个位图缓冲区的写操作终止

最后，这个函数调用 up()释放超级块的 s_lock 信号量。

Ext3 文件系统

在这一节我们将简单描述从 Ext2 发展来的增强型文件系统，即 Ext3。这个新的文件系统在设计时曾牢记两个简单的概念：

- 成为一个日志文件系统（参见下一节）
- 尽可能与原来的 Ext2 文件系统兼容

Ext3 完全达到了这两个目标。尤其是，它很大程度上是基于 Ext2 的，因此，它在磁盘上的数据结构从本质上与 Ext2 文件系统的数据结构是相同的。事实上，如果 Ext3 文件系统已经被彻底卸载，那么，就可以把它作为 Ext2 文件系统来重新安装；反之，创建 Ext2 文件系统的日志，并把它作为 Ext3 文件系统来重新安装也是一种简单、快速的操作。

由于 Ext3 与 Ext2 之间的兼容性，本章前一节的很多描述也适用于 Ext3。因此，我们集中讨论 Ext3 所提供的新特点 —— “日志”。

日志文件系统

随着磁盘变得越来越大，传统 Unix 文件系统（像 Ext2）的一种设计选择被证明是不适用的。从第十四章我们已经知道，对文件系统块的更新可能在内存保留相当长的时间后才刷新到磁盘。因此，像断电故障或系统崩溃这样不可预测的事件可能导致文件系统处于不一致状态。为了克服这个问题，每个传统的 Unix 文件系统都在安装之前要进行检查；如果它没有被正确卸载，那么，就有一个特定的程序执行彻底、耗时的检查，并修正磁盘上文件系统的所有数据结构。

例如，Ext2 文件系统的状态存放在磁盘上超级块的 `s_mount_state` 字段。由启动脚本调用 *e2fsck* 实用程序检查存放在这个字段中的值；如果它不等于 `EXT2_VALID_FS`，说明文件系统没有正确卸载，因此，*e2fsck* 开始检查文件系统的所有磁盘数据结构。

显然，检查文件系统一致性所花费的时间主要取决于要检查的文件数和目录数；因此，它也取决于磁盘的大小。如今，随着文件系统达到上百个 GB，一次一致性检查就可能花费数个小时。造成的停机时间对任何生产环境和高可用服务器都是无法接受的。

日志文件系统的目标就是避免对整个文件系统进行耗时的一致性检查，这是通过查看一个特殊的磁盘区达到的，因为这种磁盘区包含所谓日志的最新磁盘写操作。系统出现故障后重新安装日志文件系统只不过是几秒钟的事。

Ext3 日志文件系统

Ext3日志所隐含的思想就是对文件系统进行的任何高级修改都分两步进行。首先，把待写块的一个副本存放在日志中；其次，当发往日志的I/O数据传送完成时（简而言之，数据提交到日志），块就写入文件系统。当发往文件系统的I/O数据传送终止时（数据提交给文件系统），日志中的块副本就被丢弃。

当从系统故障中恢复时，*e2fsck* 程序区分下列两种情况：

提交到日志之前系统故障发生。与高级修改相关的块副本或者从日志中丢失，或者是不完整的；在这两种情况下，e2fsck 都忽略它们。

提交到日志之后系统故障发生。块的副本是有效的，且 *e2fsck* 把它们写入文件系统。

在第一种情况下，对文件系统的高级修改被丢失，但文件系统的状态还是一致的。在第二种情况下，*e2fsck* 应用于整个高级修改，因此，修正由于把未完成的 I/O 数据传送到文件系统而造成的任何不一致。

不要对日志文件系统抱有太多的期望。它只能确保系统调用级的一致性。例如，当你正在发出几个`write()`系统调用拷贝一个大型文件时发生了系统故障，这将会使拷贝操作崩溃，因此，复制的文件就会比原来的文件短。

因此，日志文件系统通常不把所有的块都拷贝到日志中。事实上，每个文件系统都由两种块组成：包含所谓元数据（metadata）的块和包含普通数据的块。在 Ext2 和 Ext3 的情形中，有六种元数据：超级块、块组描述符、索引节点、用于间接寻址的块（间接块）、数据位图块和索引节点块。其他的文件系统可能使用不同的元数据。

很多日志文件系统（如ReiserFS、SGI的XFS以及IBM的JFS）都限定自己把影响元数据的操作记入日志。事实上，元数据的日志记录足以恢复基于磁盘的文件系统数据结构的一致性。然而，因为文件的数据块不记入日志，因此就无法防止系统故障造成的文件内容的损坏。

不过，可以把Ext3文件系统配置为把影响文件系统元数据的操作和影响文件数据块的操作都记入日志。因为把每种写操作都记入日志会导致极大的性能损失，因此，Ext3让系统管理员决定应当把什么记入日志；具体来说，它提供三种不同的日志模式：

日志（Journal）

文件系统所有数据和元数据的改变都记入日志。这种模式减少了丢失每个文件所作修改的机会，但是它需要很多额外的磁盘访问。例如，当一个新文件被创建时，它的所有数据块都必须复制一份作为日志记录。这是最安全和最慢的Ext3日志模式。

预定（Ordered）
只有对文件系统元数据的改变才记入日志。然而，Ext3文件系统把元数据和相关的数据块进行分组，以便把元数据写入磁盘之前写入数据块。这样，就可以减少文件内数据损坏的机会；例如，确保增大文件的任何写访问都完全受日志的保护。这是缺省的Ext3日志模式。

写回（Writeback）
只有对文件系统元数据的改变才记入日志；这是在其他日志文件系统发现的方法，也是最快的模式。

Ext3文件系统的日志模式由*mount*系统命令的一个选项来指定。例如，为了对存放在*/dev/sda2*分区*/jdisk*安装点上的Ext3文件系统以“写回”模式进行安装，系统管理员可以键入如下命令：

```
# mount -t ext3 -o data=writeback /dev/sda2 /jdisk
```

日志块设备层

Ext3日志通常存放在名为*.journal*的隐藏文件中，位于文件系统的根目录。

Ext3文件系统本身不处理日志，而是利用所谓日志块设备（Journaling Block Device）或叫JBD的通用内核层。现在，只有Ext3使用JDB层，而其他文件系统可能在将来才使用它。

JDB层是相当复杂的软件部分。Ext3文件系统调用JDB例程以确保在系统万一出现故障时它的后续操作不会损坏磁盘数据结构。然而，JDB通常使用同一磁盘来把Ext3文件系统所做的改变记入日志，因此，它与Ext3一样易受系统故障的攻击。换言之，JDB也必须保护自己免受可能引起日志损坏的任何系统故障。

因此，Ext3与JDB之间的交互本质上基于三个基本单元：

日志记录
描述日志文件系统一个磁盘块的一次更新。

原子操作处理
包括文件系统的一次高级修改对应的日志记录；具体来说，修改文件系统的每个系统调用都引起一次单独的原子操作处理。

事务
包括几个原子操作，原子操作的日志记录对*e2fsck*同时也标记为有效。

日志记录

日志记录本质上是文件系统将要发出的低级操作的描述。在某些日志文件系统中，日志记录只包括操作所修改的字节范围及字节在文件系统中的起始位置。然而，JDB 层使用的日志记录由低级操作所修改的整个缓冲区组成。这种方式可能浪费很多日志空间（例如，当低级操作仅仅改变位图的一个位时），但是，它还是相当快的，因为 JBD 层直接对缓冲区和缓冲区首部进行操作。

因此，日志记录在日志内部表示为普通的数据块（或元数据）。但是。每个这样的块都是与类型为 `journal_block_tag_t` 的小标签相关联的，这种小标签存放块文件系统中的逻辑块号和几个状态标志。

随后，只要一个缓冲区得到 JBD 的关注，或者因为它属于日志记录，或者因为它是一个数据块，该数据块应当在相应的元数据之前刷新到磁盘（处于“预定”日志模式），内核都会把 `journal_head` 数据结构加入到缓冲区首部。在这种情况下，缓冲区首部的 `b_private` 字段存放 `journal_head` 数据结构的地址，并设置 `BH_JBD` 标志（参见第十三章“缓冲区首部”）。

原子操作处理

修改文件系统的任一系统调用都通常划分为操纵磁盘数据结构的一系列低级操作。

例如，假定 Ext3 必须满足用户把一个数据块追加到普通文件的请求。文件系统层必须确定文件的最后一个块，定位文件系统中的一个空闲块，更新适当块组内的数据块位图，存放新块的逻辑块号在文件的索引节点或间接寻址块中，写新块的内容，并在最后更新索引节点的几个字段。你可以看到，追加操作转换为对文件系统数据块和元数据块的很多低级操作。

现在，仅仅想像一下，如果在追加操作的中间一些低级操作已经执行，另一些还没有执行而系统出现了故障会发生什么事情。当然，对于影响两个或多个文件的高级操作（例如，把文件从一个目录移到另一个目录），情况会更糟。

为了防止数据损坏，Ext3 文件系统必须确保每个系统调用以原子的方式进行处理。原子操作处理（atomic operation handle）是对磁盘数据结构的一组低级操作，这组低级操作对应一个单独的高级操作。当从系统故障中恢复时，文件系统确保要么整个高级操作起作用，要么没有一个低级操作起作用。

任何原子操作处理都用类型为 `handle_t` 的描述符来表示。为了开始一个原子操作，Ext3 文件系统调用 `journal_start()` JBD 函数，该函数在必要时分配一个新的原子操作处理并把它插入到当前的事务中（见下一节）。因为对磁盘的任何低级操作都可能挂起进

程，因此，活动原子操作处理的地址存放在进程描述符的 journal_info字段中。为了通知原子操作已经完成，Ext3 文件系统调用 journal_stop()函数。

事务

出于效率的原因，JBD层对日志的处理采用分组的方法，即把属于几个原子操作处理的日志记录分组放在一个单独的事务中。此外，与一个处理相关的所有日志记录都必须包含在同一个事务中。

一个事物的所有日志记录都存放在日志的连续块中。JBD层把每个事务作为整体来处理。例如，只有当包含在一个事务的日志记录中的所有数据提交给文件系统时才回收该事务所使用的块。

事务一被创建，它就能接受新处理的日志记录。当下列情况之一发生时，事务就停止接受新处理：

- 固定的时间已经过去，典型情况下为 5 秒
- 日志中没有空闲块留给新处理

事务是由类型为 transaction_t 的描述符来表示的。其最重要的字段为 t_state，该字段描述事务的当前状态。

从本质上说，事务可以是：

完成的
: 包含在事务中的所有日志记录都已经从物理上写入日志。当从系统故障中恢复时，*e2fsck* 考虑日志中每个完成的事务，并把相应的块写入文件系统。在这种情况下，t_state 字段存放值 T_FINISHED。

未完成的
: 包含在事务中的日志记录至少还有一个没有物理上写入日志，或者新的日志记录还正在追加到事务中。在系统故障的情况下，存放在日志中的事务映像很可能不是最新的。因此，当从系统故障中恢复时，*e2fsck* 不信任日志中未完成的事务，并跳过它们。在这种情况下，i_state 存放下列值之一：

  T_RUNNING
  : 还在接受新的原子操作处理。

  T_LOCKED
  : 不接受新的原子操作处理，但其中的一些还没有完成。

T_FLUSH
所有的原子操作处理都已完成，但一些日志记录还正在写入日志。

T_COMMIT
原子操作处理的所有日志记录都已经写入磁盘，且在日志中把事务标记为完成。

在任一给定的实例中，日志可能包含多个事务。其中只有一个处于 T_RUNNING 状态——即它是活动事务（active transaction），所谓活动事务就是正在接受由 Ext3 文件系统发出的新原子操作处理的请求。

日志中的几个事务可能是未完成的，因为包含相关日志记录的缓冲区还没有写入日志。

只有当 JDB 层确认日志记录描述的所有缓冲区都已成功写入 Ext3 文件系统时，一个完成的事务才能从日志中删除。因此，日志可以包含至多一个未完成的事务和几个完成的事务。完成事务的日志记录已经写入日志，但是，相应的一些缓冲区还有待写入文件系统。

日志如何工作

让我们用一个例子来尝试解释日志如何工作：Ext3文件系统层接受向普通文件写一些数据块的请求。

你可能很容易猜到，我们不打算详细描述Ext3文件系统层和JDB层的每个单独操作。那将会涉及很多问题！但是，我们描述本质的操作：

1. write()系统调用服务例程触发与 Ext3 普通文件相关的文件对象的 write 方法。对于 Ext3 来说，这个方法是由 generic_file_write()函数实现的，这已在第十五章“写入文件”一节进行了描述。
2. generic_file_write()函数多次调用address_space对象的prepare_write方法，写方法涉及的每个数据页都调用一次。对 Ext3 来说，这个方法是由 ext3_prepare_write()函数实现的。
3. ext3_prepare_write()函数调用 journal_start() JBD 函数开始一个新的原子操作。这个原子操作处理被加到活动事务中。实际上，原子操作处理是在第一次调用journal_start()函数时创建的。接着的调用确认进程描述符的journal_info字段已经被设置，并使用这个处理。
4. ext3_prepare_write()函数调用第十五章已描述的 block_prepare_write()函数，传递给它的参数为 ext3_get_block()函数的地址。回想一下，block_prepare_write()负责为文件页的缓冲区和缓冲区首部做准备。

5. 当内核必须确定Ext3文件系统的逻辑块号时，就执行ext3_get_block()函数。这个函数实际上类似于ext2_get_block()，后者在前面“分配数据块”一节已经描述。但是，有一个主要的差异在于Ext3文件系统调用JDB层的函数来确保低级操作记入日志。

   - 在对Ext3文件系统的元数据块发出低级写操作之前，该函数调用journal_get_write_access()。后一个函数主要把元数据缓冲区加入到活动事务的链接表中。但是，它也必须检查元数据是否包含在日志的一个较老的未完成的事务中；在这种情况下，它把缓冲区复制一份以确保老的事物以老的内容提交。
   - 在更新元数据所在的缓冲区之后，Ext3文件系统调用journal_dirty_metadata()把元数据缓冲区移到活动事务的适当脏链接表中，并在日志中记录这一操作。

   注意，由JDB层处理的元数据缓冲区并不实际包含在索引节点的缓冲区的脏链接表中，因此，这些缓冲区并不由第十四章描述的正常磁盘高速缓存的刷新机制写入磁盘。

6. 如果Ext3文件系统已经以“日志”模式安装，则ext3_prepare_write()函数在写操作触及的每个缓冲区上也调用journal_get_write_access()。

7. 控制权回到generic_file_write()函数，该函数用存放在用户态地址空间的数据更新页，并调用address_space对象的commit_write方法。对于Ext3，这个方法是由ext3_commit_write()函数实现的。

8. 如果Ext3文件系统已经以“日志”模式安装，则ext3_commit_write()函数对页的每个数据（不是元数据）缓冲区调用journal_dirty_metadata()。这样，缓冲区就包含在活动事务的适当脏链接表中，但不包含在拥有者索引节点的脏链接表中；此外，相应的日志记录写入日志。

9. 如果Ext3文件系统已经以“预定”模式安装，则ext3_commit_write()对页中的每个数据缓冲区调用journal_dirty_data()函数来把缓冲区插入到活动事务的适当链接表中。JDB层确保这个链接表中的所有缓冲区在事务中的元数据缓冲区写入磁盘之前写入。没有日志记录写入日志。

10. 如果Ext3文件系统已经以“预定”或“写回”模式安装，则ext3_commit_write()函数执行第十五章描述的正常的generic_commit_write()函数，该函数把数据缓冲区插入拥有者索引节点的脏缓冲区链接表中。

11. 最后，ext3_commit_write()调用journal_stop()以通知JBD层原子操作处理被关闭。

12. write()系统调用的服务例程到此结束。但是，JDB 层还没有完成它的工作。终于，当事务的所有日志记录都物理地写入日志时，我们的事务完成。然后，执行journal_commit_transaction()。

13. 如果Ext3文件系统已经以“预定”模式安装，则journal_commit_transaction()函数为事务链接表包含的所有数据缓冲区激活I/O数据传送，并等待，直到数据传送终止。

14. journal_commit_transaction()函数为包含在事务中的所有元数据缓冲区激活I/O数据传送（如果Ext3以“日志”模式安装，也为所有的数据缓冲区激活I/O数据传送）。

15. 内核周期性地为日志中每个完成的事务激活检查点活动。检查点主要验证由journal_commit_transaction()触发的I/O数据传送是否已经成功结束。如果是，则从日志中删除事务。

当然，除非发生系统故障，否则，日志中的日志记录根本就没有什么积极作用。事实上，只有在系统发生故障时，*e2fsck* 实用程序才扫描存放在文件系统中的日志，并重新安排完成的事务中的日志记录所描述的所有写操作。