第十八章

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，Linux支持的其他文件系统有很多有趣的特点，但因篇幅限制，我们并不对此一一讨论。

在“Ext2的一般特征”一节中引入Ext2后，我们会像在其他章节中一样，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，因此必须考虑同一数据结构的两种形式：“Ext2磁盘数据结构”一节说明把Ext2存放在磁盘上的数据结构，而“Ext2的内存数据结构”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Ext2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，Ext3是Ext2文件系统的发展。

Ext2的一般特征

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，还相当高效和稳定，Ext2及它的下代文件系统Ext3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它的磁盘分区有较少的不匹配（参见第八章中的“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。
- 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进行访问。
- 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。
- 支持快速符号链接（参见第一章中的“硬链接和软链接”一节）。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读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：

- 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。如果这种操作的顺序相反（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（*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）。
- 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。

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”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。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

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（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 18-1 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。

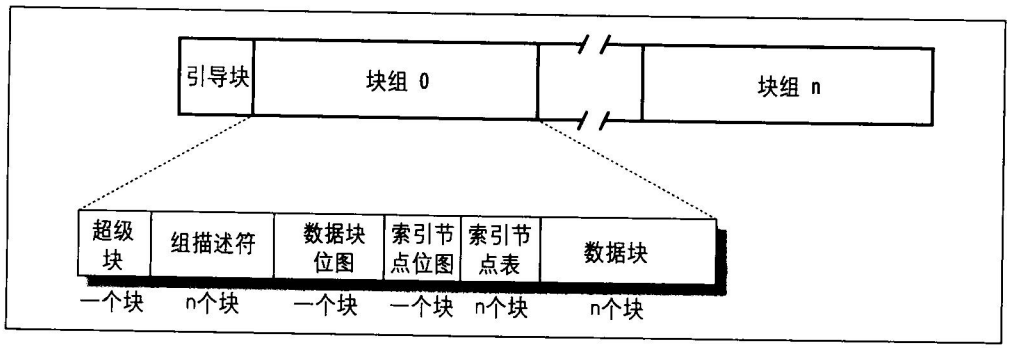

图 18-1：Ext2 分区和 Ext2 块组的分布图

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：

- 文件系统的超级块的一个拷贝
- 一组块组描述符的拷贝
- 一个数据块位图
- 一个索引节点位图
- 一个索引节表
- 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。

从图 18-1 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。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 $8 \times b$ 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 是分区所包含的总块数。

举例说明，让我们考虑一下 32GB 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 256 个块组。显然，块的大小越小，块组数越大。

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 18-1 中列出（注 1）。__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）。

表 18-1：Ext2 超级块的字段

类型	字段	说明
__le32	s_inodes_count	索引节点的总数
__le32	s_blocks_count	以块为单位的文件系统的大小
__le32	s_r_blocks_count	保留的块数
__le32	s_free_blocks_count	空闲块计数器
__le32	s_free_inodes_count	空闲索引节点计数器
__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

注 1：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，它们并未在表 18-1 中列出。

表 18-1：Ext2 超级块的字段（续）

类型	字段	说明
__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	安装操作计数器
__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	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	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
__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 [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
__le32	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 [204]	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，如此等等。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；如果被正常卸载，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

每个块组都有自己的组描述符，它是一个 ext2_group_desc 结构，它的字段在表 18-2 中列出。

表 18-2：Ext2 组描述符的字段

类型	字段	说明
__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 [3]	bg_reserved	用 null 填充 24 个字节

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。位图是位的序列，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。

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。

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。

每个Ext2索引节点为ext2_innode结构，其字段如表18-3所示。

表18-3：Ext2磁盘索引节点的字段

类型	字段	说明
__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
__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	osd1	特定的操作系统信息
__le32 [EXT2_N_BLOCKS]	i_block	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使用)
__le32	i_file_acl	文件访问控制列表

表18-3：Ext2磁盘索引节点的字段（续）

类型	字段	说明
__le32	i_dir_acl	目录访问控制列表
__le32	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 × 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_blocks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（参见第十二章“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合适的索引节点描述符的一个关键字。

索引节点的增强属性

Ext2索引节点的格式对于文件系统设计者就好像一件紧身衣，索引节点的长度必须是2的幂，以免造成存放索引节点表的块内碎片。实际上，一个Ext2索引节点的128个字符空间中充满了信息，只有少许空间可以增加新的字段。另一方面，将索引节点的长度增加至256不仅相当浪费，而且使用不同索引节点长度的Ext2文件系统之间还会造成兼容问题。

引入增强属性（*extended attribute*）就是要克服上面的问题。这些属性存放在索引节点之外的磁盘块中。索引节点的`i_file_acl`字段指向一个存放增强属性的块。具用同样增强属性的不同索引节点可以共享同一个块。

每个增强属性有一个名称和值。两者都编码为变长字符数组，并由`ext2_xattr_entry`描述符来确定。图18-2表示Ext2中增强属性块的结构。每个属性分成两部分：在块首部的是`ext2_xattr_entry`描述符与属性名，而属性值则在块尾部。块前面的表项按照属性名称排序，而值的位置是固定的，因为它们是由属性的分配次序决定的。

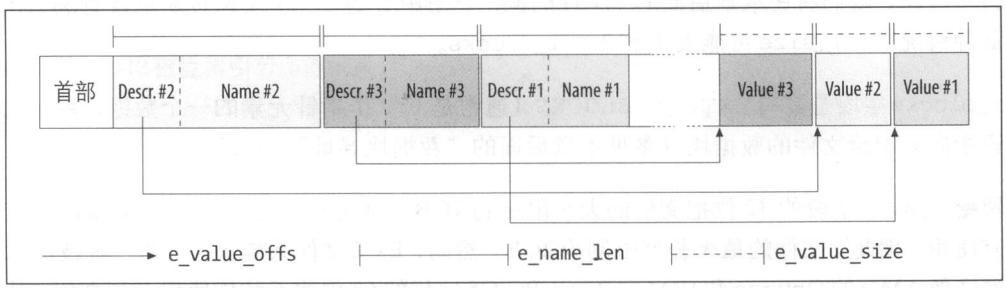

图18-2：含增强属性的块的结构

有很多系统调用用来设置、取得、列表和删除一个文件的增强属性。系统调用`setxattr()`、`lsetxattr()`和`fsetxattr()`设置文件的增强属性，它们在符号链接的处理与文件限定的方式（或者传递路径名或者是文件描述符）上根本不同。类似地，系统调用`getxattr()`、`lgetxattr()`和`fgetxattr()`返回增强属性的值。系统调用`listxattr()`、`llistxattr()`和`flistxattr()`则列出一个文件的所有增强属性。最后，系统调用`removexattr()`、`lremovexattr()`和`fremovexattr()`从文件删除一个增强属性。

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。不是将文件的用户分成三类：拥有者、组和其他，访问控制列表（*access control list, ACL*）可以与每个

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()`就是通过上一节中介绍的 `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。本节讨论每种文件类型的存储要求，如表 18-4 所示。

表 18-4：Ext2 文件类型

文件类型	说明
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。表18-5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18-3）。

表18-5：Ext2目录项中的字段

类型	字段	说明
__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 [EXT2_NAME_LEN]	name	文件名

file_type字段存放指定文件类型的值（见表18-4）。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18-3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）。

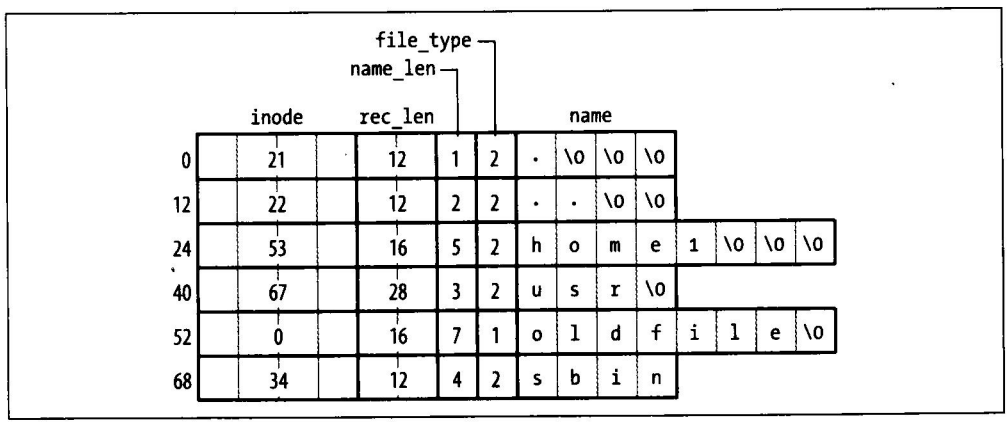

图18-3：Ext2目录的一个例子

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 字段的值和相应的组描述符中 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，内核利用页高速缓存来保持它们最新（参见第十五章中的“把脏页写入磁盘”一节）。

对于与 Ext2 文件系统以及文件相关的每种数据类型，表 18-6 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决定使用多大容量高速缓存的经验方法。频繁更新的数据总是存放在高速缓存，也就是说，这些数据一直存放在内存并包含在页高速缓存中，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 18-6：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

表 18-6：Ext2 数据结构的 VFS 映像（续）

类型	磁盘数据结构	内存数据结构	缓存模式
块位图	块中的位数组	缓冲区中的位数组	动态
索引节点位图	块中的位数组	缓冲区中的位数组	动态
索引节点	ext2_inode	ext2_inode_info	动态
数据块	字节数组	VFS 缓冲区	动态
空闲索引节点	ext2_inode	无	从不缓存
空闲块	字节数组	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。相反，“总是缓存”的数据也总在RAM中，这样就不必从磁盘读数据了（但是，数据必须周期性地写回磁盘）。除了这两种极端模式外，还有一种动态模式。在动态模式下，只要相应的对象（索引节点、数据块或位图）还在使用，它就保存在高速缓存中；而当文件关闭或数据块被删除后，页框回收算法会从高速缓存中删除有关数据。

有意思的是，索引节点与块位图并不永久保存在内存里，而是需要时从磁盘读。有了页高速缓存，最近使用的磁盘块保存在内存里，这样可以避免很多磁盘读（参见第十五章“把块存放在页高速缓存中”一节）（注 2）。

Ext2 的超级块对象

在第十二章“超级块对象”一节我们介绍过，VFS超级块的s_fs_info字段指向一个包含文件系统信息的数据结构。对于Ext2，该字段指向ext2_sb_info类型的结构，它包含如下信息：

- 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（通常一项就够）

注 2：在Linux 2.4和早期的版本中，最近使用的索引节点和块位图被保存在特殊高速缓存的有限空间里。

- 其他与安装状态、安装选项等有关的数据

图18-4 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。

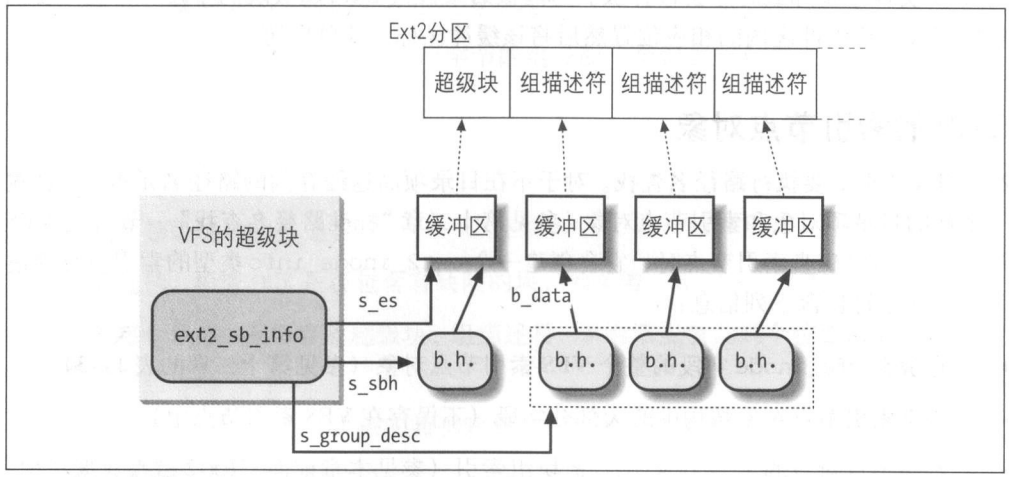

图 18-4：ext2_sb_info 数据结构

当内核安装Ext2文件系统时，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（参见第十二章“安装普通文件系统”一节）。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的s_fs_info字段。
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。在第十五章的“在页高速缓存中搜索块”一节我们讨论过，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 超级块对象的 s_sbh 字段。
3. 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（参见本章后面的“创建索引节点”一节）。
4. 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。
5. 重复调用__bread()分配缓冲区，从磁盘读入包含 Ext2 组描述符的块。把缓冲区首部地址存放在上一步得到的 s_group_desc 数组中。

6. 为根目录分配一个索引节点和目录项对象，为超级块建立相应的字段，从而能够从磁盘读入根索引节点对象。

很显然，ext2_fill_super()函数返回后，分配的所有数据结构都保存在内存里，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。

Ext2的索引节点对象

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（参见第十二章“标准路经名查找”一节）。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。该描述符包含下列信息：

- 存放在vfs_inode字段的整个VFS索引节点对象（参见第十二章的表12-3）
- 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）
- 索引节点对应的i_block_group块组索引（参见本章前面“Ext2磁盘数据结构”一节）
- i_next_alloc_block和i_next_alloc_goal字段，分别存放着最近为文件分配的磁盘块的逻辑块号与物理块号
- i_prealloc_block和i_prealloc_count字段，用于数据块预分配（参见本章后面“分配数据块”一节）
- 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ACL

当处理Ext2文件时，alloc_inode超级块方法是由ext2_alloc_inode()函数实现的。它首先从ext2_inode_cachep slab分配器高速缓存得到一个ext2_inode_info描述符，然后返回在这个ext2_inode_info描述符中的索引节点对象的地址。

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。第二步格式化磁盘，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*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：块的大小（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：

```
$ 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 3）。

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 23 个块中。

注 3： 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。

表 18-7 总结了按缺省选项如何在软盘上建立 Ext2 文件系统。

表 18-7：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
5～27	索引节点表：5～10 —— 保留（2 是 root）；11 —— lost+found；12～184 —— 空闲
28	根目录（包括“.”、“..”及“lost+found”）
29	*lost+found* 目录（包括“.”和“..”）
30～40	给 *lost+found* 目录预分配保留的块
41～1439	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，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

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。

Ext2 的普通文件和目录文件的索引节点操作见表 18-8。每个方法的目的在第十二章的“索

引节点对象”一节有介绍。表中没有列出普通文件和目录中未定义的方法（NULL 指针）。回忆一下，如果方法未定义，VFS 要么调用通用函数，要么什么也不做。普通文件与目录的Ext2方法地址分别存放在ext2_file_inode_operations和ext2_dir_inode_operations表中。

表 18-8：Ext2 普通文件与目录的索引节点操作

VFS 索引节点操作	普通文件	目录
create	NULL	ext2_create()
lookup	NULL	ext2_lookup()
link	NULL	ext2_link()
unlink	NULL	ext2_unlink()
symlink	NULL	ext2_symlink()
mkdir	NULL	ext2_mkdir()
rmdir	NULL	ext2_rmdir()
mknod	NULL	ext2_mknod()
rename	NULL	ext2_rename()
truncate	ext2_truncate()	NULL
permission	ext2_permission()	ext2_permission()
setattr	ext2_setattr()	ext2_setattr()
setxattr	generic_setxattr()	generic_setxattr()
getxattr	generic_getxattr()	generic_getxattr()
listxattr	ext2_listxattr()	ext2_listxattr()
removexattr	generic_removexattr()	generic_removexattr()

Ext2 的符号链接的索引节点操作见表 18-9（省略未定义的方法）。实际上有两种符号链接：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。

表 18-9：Ext2 的快速及普通符号链接的索引节点操作

VFS 索引节点操作	快速符号链接	普通符号链接
readlink	generic_readlink()	generic_readlink()
follow_link	ext2_follow_link()	page_follow_link_light()
put_link	NULL	page_put_link()

表 18-9：Ext2 的快速及普通符号链接的索引节点操作（续）

VFS 索引节点操作	快速符号链接	普通符号链接
setxattr	generic_setxattr()	generic_setxattr()
getxattr	generic_getxattr()	generic_getxattr()
listxattr	ext2_listxattr()	ext2_listxattr()
removexattr	generic_removexattr()	generic_removexattr()

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18-10列出了Ext2文件系统特定的文件操作。正如你看到的，一些VFS方法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 18-10：Ext2 文件操作

VFS 文件操作	Ext2 方法
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()

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“从文件中读取数据”和“写入文件”两节进行了描述。

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）。

在本节，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4）。这个问题类似于在第八章的“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。

该函数作用于两个参数：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（参见第十二章“索引节点对象”一节）。

注4：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。

2. 如果新的索引节点是一个目录，函数就调用find_group_orlov()为目录找到一个合适的块组（注5）。该函数执行如下试探法：

   a. 以文件系统根root为父目录的目录应该分散在各个组。这样，函数在这些块组去查找一个组，它的空闲索引节点数和空闲块数比平均值高。如果没有这样的组则跳到第2c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组：

      - 该组没有包含太多的目录
      - 该组有足够多的空闲索引节点
      - 该组有一点小“债”。（块组的债存放在一个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，债加一；每当其他类型的文件加入，债减一）

      如果父目录组不满足这些条件，那么选择第一个满足条件的组。如果没有满足条件的组，则跳到第2c步。

   c. 这是一个“退一步”原则，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。

3. 如果新索引节点不是个目录，则调用find_group_other()，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$i \bmod (n)$，$i+1 \bmod (n)$，$i+1+2 \bmod (n)$，$i+1+2+4 \bmod (n)$，等等。

   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。

4. 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位，这样就得到了第一个空闲磁盘索引节点号。

5. 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（参见第十二章中的“安装普通文件系统”一节），则调用sync_dirty_buffer()开始I/O写操作并等待，直到写操作终止。

注5：安装Ext2文件系统还可以带有一个选项标志，它强制内核使用一个更简单、更老式的分配策略，该策略是由find_group_dir()函数实现的。

6. 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。
7. 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. 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。
9. 将超级块的s_dirt标志置1，并把包含它的缓冲区标记为脏。
10. 把VFS超级块对象的s_dirt字段置1。
11. 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。关于这些字段的含义请参考表18-3。
12. 初始化这个索引节点对象的访问控制列表（ACL）。
13. 将新索引节点对象插入散列表inode_hashtable，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（参见第十二章“索引节点对象”一节）。
14. 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。
15. 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. 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（参见后面“数据块寻址”一节）。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（参见第十五章“address_space对象”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. 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点对象，这个链表要么在cdev字符设备描述符的cdev字段（参见第十三章“字符设备驱动程序”一节），要么在block_device块设备描述符的bd_inodes字段（参见第十四章“块设备”一节）。

e. 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）。

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

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（参见第十四章的“块设备的处理”一节）。

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，以此类推。

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直截了当的。

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。

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18-5所示，这个数组表示一个大型数据结构的初始化部分。正如从图中所看到的，数组的15个元素有4种不同的类型：

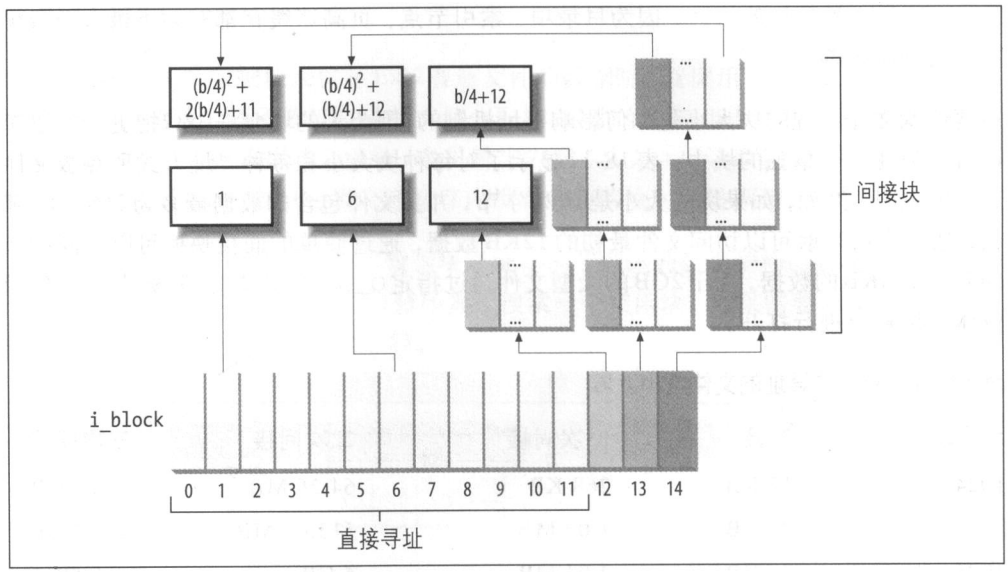

图18-5：对文件的数据块进行寻址的数据结构

- 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～*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）。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。

- 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 $b/4+12 \sim (b/4)^2+(b/4)+11$。
- 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 $(b/4)^2+(b/4)+12 \sim (b/4)^3+(b/4)^2+(b/4)+11$。

在图18-5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。表18-11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13～268KB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。

表18-11：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	~4 TB

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。

```
$ 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（参看图18-6）。

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（参见第十六章“从文件中读取数据”和“写入文件”两节）；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“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（参见图18-5）。

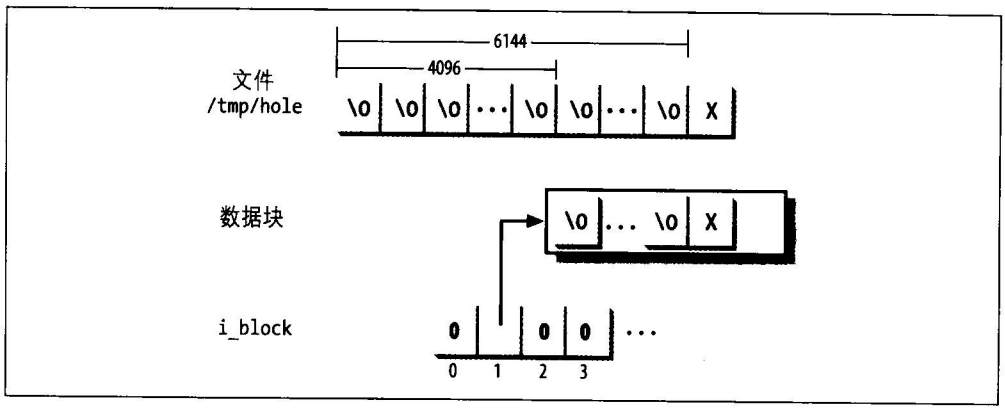

图18-6：起始部分有洞的文件

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。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：
   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。

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。

释放数据块

当进程删除一个文件或把它的长度截为 0 时，其所有数据块必须回收。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间接寻址用的块的位置。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
: 文件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. 增加块组描述符的 bg_free_blocks_count 字段，并把相应的缓冲区标记为脏。
4. 增加磁盘超级块的 s_free_blocks_count 字段，并把相应的缓冲区标记为脏，把超级块对象的 s_dirt 标记置位。
5. 如果 Ext2 文件系统安装时设置了 MS_SYNCHRONOUS 标志，则调用 sync_dirty_buffer()并等待，直到对这个位图缓冲区的写操作终止。

Ext3 文件系统

在本节我们将简单描述从 Ext2 发展而来的增强型文件系统，即 Ext3。这个新的文件系统在设计时曾秉持两个简单的概念：

- 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，本章前面几节的很多描述也适用于Ext3。因此，本节我们集中于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）的一种设计选择证明是不相称的。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文件系统在安装之前都要进行检查；如果它没有被正常卸载，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`字段中。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常卸载，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，随着文件系统达到几百个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（*journal*）的最新磁盘写操作。系统出现故障后，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，把数据提交到日志），块就被写入文件系统。当发往文件系统的I/O数据传送终止时（把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，这将会使拷贝操作中断，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位图块。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磁盘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被记入日志。这种模式减少了丢失每个文件修改的机会，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被记入日志。然而，Ext3文件系统把元数据和相关的数据块进行分组，以便在元数据之前把数据块写入磁盘。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被记入日志；这是在其他日志文件系统中发现的方法，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，为了在*/jdisk*安装点对存放在*/dev/sda2*分区上的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，该文件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*，*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典型地使用同一磁盘来把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影响。换言之，JDB也必须保护自己免受任何系统故障引起的日志损坏。

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；一般来说，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
: 包括几个原子操作处理，同时，原子操作处理的日志记录对*e2fsck*标记为有效。

日志记录

日志记录（*log record*）本质上是文件系统将要发出的一个低级操作的描述。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，这种小标签存放块在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”模式），那么，内核把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，并把BH_JBD标志置位（参见第十五章“块缓冲区和缓冲区首部”一节）。

原子操作处理

修改文件系统的任一系统调用通常都被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很多低级的操作。

现在，仅仅想象一下，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（*transaction*）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。

一个事务的所有日志记录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都提交给文件系统时才回收该事务所使用的块。

事务一旦被创建，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5s。
- 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从物理上写入日志，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，但在日志中，事务仍然被标记为完成。

在任何时刻，日志可能包含多个事务，但其中只有一个处于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。

如果事务完成，说明所有日志记录已被写入日志，但是一部分相应的缓冲区还没有写入文件系统。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。

日志如何工作

让我们用一个例子来试图解释日志如何工作：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。那将会涉及太多问题！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，这已在第十六章“写入文件”一节进行了描述。
2. generic_file_write()函数几次调用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。后续的调用确认进程描述符的journal_info字段已经被置位，并使用这个处理。
4. ext3_prepare_write()函数调用第十六章已描述过的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()负责准备文件页的缓冲区和缓冲区首部。
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务以老的内容提交。

- 在更新元数据块所在的缓冲区之后，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通常并不包含在索引节点的缓冲区的脏链表中，因此，这些缓冲区并不由第十五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，函数如何实现这个方法取决于Ext3文件系统的安装方式：

   - 如果Ext3文件系统已经以“日志”模式安装，那么commit_write方法是由ext3_journalled_commit_write()函数实现的，它对页中的每个数据（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表中，但不包含在拥有者索引节点的脏链表中；此外，相应的日志记录写入日志。最后，ext3_journalled_commit_write()调用journal_stop通知JBD层原子操作处理已关闭。

   - 如果Ext3文件系统已经以“预定”模式安装，那么commit_write方法是由ext3_ordered_commit_write()函数实现的，它对页中的每个数据缓冲区调用journal_dirty_data()函数以把缓冲区插入到活动事务的适当链表中。JDB层确保在事务中的元数据缓冲区写入之前这个链表中的所有缓冲区写入磁盘。没有日志记录写入日志。然后，ext3_ordered_commit_write()函数执行第十五章描述的常规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表中。最后，ext3_ordered_commit_write()调用journal_stop()通知JBD层原子操作处理已关闭。

   - 如果Ext3文件系统已经以“写回”模式安装，那么commit_write方法是由ext3_writeback_commit_write()函数实现的，它执行第十五章描述的常规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表中。然后，ext3_ writeback_commit_write()调用journal_stop()通知JBD层原子操作处理已关闭。

8. 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，我们的事务才完成。然后，执行journal_commit_transaction()。

9. 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。

10. 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，则也为所有的数据缓冲区激活I/O数据传送）。
11. 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。