7.5 示例：Linux 的 ext2 文件系统

ext2 文件系统可谓是 Linux 土生土长的文件系统。由于它是 ext（Extended File System）的完善，故而得名 ext2（The Second Extended File System）。ext2 具有很好的扩展性、高效性和安全性，在 Linux 世界里得到广泛应用。它大致有以下一些特点：

1. 支持 UNIX 所有标准的文件系统特征，包括普通文件（regular files）、目录、设备文件和链接文件等，这使得它很容易被 UNIX 程序员接受。事实上，ext2 的绝大多数的数据结构和系统调用与经典的 UNIX 一致。
2. 能够管理海量存储介质。支持多达 4TB 的数据，即一个分区的容量最大可达 4TB。
3. 支持长文件名，最多可达 255 个字符，并且可扩展到 1012 个字符。
4. 允许用户通过文件属性控制别的用户对文件的访问；目录下的文件继承目录的属性。
5. 支持文件系统数据“即时同步”特性，即内存中的数据一旦改变，立即更新硬盘上的数据使之一致。
6. 实现了“快速连接”（fast symbolic links）的方式，使得连接文件只需要存放 inode 的空间。
7. 允许用户定制文件系统的数据单元（block）的大小，可以是 1024、2048 或 4096 个字节，使之适应不同环境的要求。
8. 使用专用文件记录文件系统的状态和错误信息，供下一次系统启动时决定是否需要检查文件系统。

下面将介绍 ext2 的体系结构、关键的数据结构（包括超级块、组描述符、inode）、ext2 文件系统的具体操作的实现和数据块分配机制。

1. ext2 体系结构

与其它文件系统一样，ext2 文件系统也是由逻辑块的序列组成。除了第一个引导块外之外（1 个 block），ext2 文件系统将它所占用的逻辑分区划分为**块组（Block Group）**，每个块组保存着关于文件系统的备份信息（超级块和所有的组描述符）。实际上只有第一个块组的超级块内容才被文件管理系统读入。

ext2 文件系统的体系结构如图 7.25。

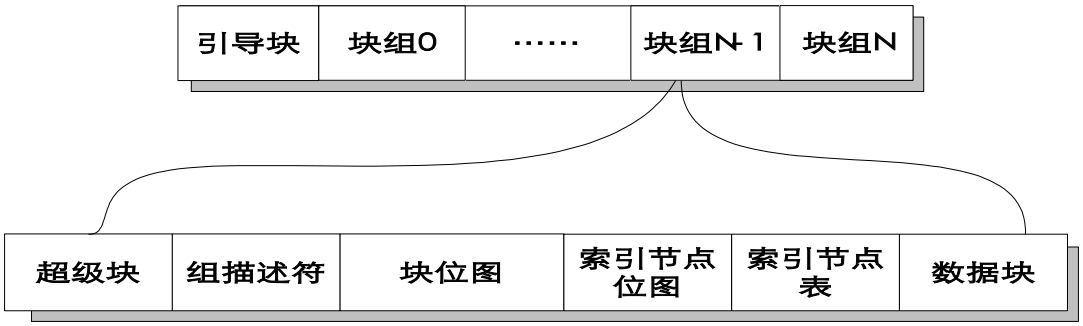

图 7.25 ext2 体系结构

- **超级块(super block)**：文件系统中最重要的结构，描述了整个系统的信息，如设备号、块大小、操作该文件系统的函数、安装路径等。
- **组描述符(group descriptor)**：记录所有块组的信息，如块组中的空闲块数、空闲节点数等。
- **块位图(block bitmap)**：每一个块组有一个对应的块位图，块位图中的每一位代表一个块，1 表示被使用，0 表示是空闲块。
- **inode 位图(inode bitmap)**：每一个块组有一个对应的 inode 位图，inode 位图中的每一位代表一个块，1 表示被使用，0 表示是空闲块。
- **inode 表(inode table)**：每一个文件用一个 inode 表示，inode 表存放该文件系统中所有的 inode。
- **数据块**：实际存放文件数据的块。

图 7.24 并不复杂，却涵盖了 ext2 数据布局的全局。一个块组包含一个超级块，块组中对应块的使用信息由组描述符维护。读者从下文可以得知，对于所有块组，它们的超级块和组描述符包含的信息是相同的。而块位图、inode 位图、inode 表、数据块与每一个块组相关。每个文件，无论是目录文件还是普通文件都用一个 inode 来描述。

在 ext2 文件系统中，所有数据块的长度相同，但是对于不同的 ext2 文件系统，数据块的长度可以不同。当然，对于给定的 ext2 文件系统，其块的大小在创建时就会固定下来。文件总是整块存储，不足一块的部分也占用一个数据块。例如，在数据块长度为 1024 字节的 ext2 文件系统中，一个长度为 1025 字节的文件就要占用 2 个数据块。

ext2 文件系统相关代码存放在 fs/ext2 目录下。include/linux/ext2_fs.h、ext2_fs_i.h 和 ext2_fs_sb.h 中也有 ext2 的重要数据结构定义。读者在阅读 ext2 的源代码时，经常看到很多数据结构之间的维护和转换方面的代码，可以参考 ext2 体系结构图理解这些过程的具体实现。

2. ext2 的关键数据结构

（1）超级块 super_block

每一个块组包含的超级块都是相同的。一般，只有块组 0 的超级块才读入内存。读者可能会问，为什么各个块组都需要包含超级块呢？原因很简单，其它超级块信息只作为备份。由此可见超级块对于维护整个文件系统的作用是至关重要的。

ext2 使用一个称为 ext2_super_block 的数据结构，它包含了文件系统内部的关键信息，其长度目前是 1024 个字节。ext2_super_block 中某些成员在文件系统创建时确定，另有一些则可根据文件系统管理者的实际要求在运行时改变。ext2_super_block 存在于硬盘中，供载入文件系统时读入相应的文件系统信息以建立相应的 VFS 超级块，其中包含文件块的大小之类的信息。当 Linux 将 ext2 文件系统载入内存中后，使用另一个 ext2_sb_info 数据结构来存放有关信息，这样对 ext2 文件系统核心数据的访问只需要在内存中操作即可。对超级块的访问是互斥的，即任意时刻最多只允许有一个进程拥有超级块访问权。

```
include/linux/ext2_fs.h, line 341
341 struct ext2_super_block {
342         __le32  s_inodes_count;         /* Inodes count */
343         __le32  s_blocks_count;         /* Blocks count */
344         __le32  s_r_blocks_count;       /* Reserved blocks count */
345         __le32  s_free_blocks_count;    /* Free blocks count */
346         __le32  s_free_inodes_count;    /* Free inodes count */
347         __le32  s_first_data_block;     /* First Data Block */
348         __le32  s_log_block_size;       /* Block size */
349         __le32  s_log_frag_size;        /* Fragment size */
350         __le32  s_blocks_per_group;     /* # Blocks per group */
351         __le32  s_frags_per_group;      /* # Fragments per group */
352         __le32  s_inodes_per_group;     /* # Inodes per group */
353         __le32  s_mtime;                /* Mount time */
354         __le32  s_wtime;                /* Write time */
355         __le16  s_mnt_count;            /* Mount count */
356         __le16  s_max_mnt_count;        /* Maximal mount count */
357         __le16  s_magic;                /* Magic signature */
358         __le16  s_state;                /* File system state */
     ...
411 };
```

s_inodes_count：　文件使用的文件节点数。
s_blocks_count：　文件块数。
s_r_blocks_count：保留未用的文件块数。
s_free_blocks_count：可用的文件块数。
s_free_inodes_count：可用的 inode 数目。
s_first_data_block：第一个数据块的位置。
s_log_block_size：用来计算 ext2 文件系统数据块的大小。
s_log_frag_size：　用来计算 ext2 文件系统文件碎片大小。

s_blocks_per_group：每个组的文件块的数目。
s_frags_per_group：每个组的碎片数目。
s_inodes_per_group：每个组的 inode 总数。
s_mtime：　　最近被装载（mount）的时间。
s_wtime：　　最近被修改的时间。
s_mnt_count：　最近一次文件系统检查（fsck）后被装载的次数。
s_max_mnt_count：最大可被安装的次数。当达到这个数目时，ext2 文件系统必须被检查，以保证一致性。
s_magic：　　文件系统的标识。
s_state：　　文件系统的状态。

（2）组描述符 Group Descriptor

为了易于管理，ext2 将整个文件系统建筑在**块（block）**的基础之上。物理存储介质被逻辑分成小块的数据块（block），这也是所能被分配的最小存储单元。数据块的大小可以是 512、1024、2048 或 4096 个字节，但一旦文件系统创建完毕，数据块大小就不可改变。一定数目的连续分配的数据块被组织在一起形成一个 group，这使得 ext2 能够将相似的信息组织在相近的物理存储介质范围内。ext2 使用一个叫做 group descriptor 的结构来管理 block group。这就是块组描述符的由来。

组描述符和超级块一样，记录的信息与整个文件系统相关。当某一个组的超级块或 inode 受损时，这些信息可以用于恢复文件系统。因此，为了更好地维护文件系统，每个块组中都保存关于文件系统的备份信息（超级块和所有组描述符）。

块位图（block bitmap）记录本组内各个数据块的使用情况，其中每一个 bit 对应于一个数据块，0 表示空闲，非 0 表示已经占用。

```
include/linux/ext2_fs.h, line 136
136 struct ext2_group_desc
137 {
138         __le32  bg_block_bitmap;                /* Blocks bitmap block */
139         __le32  bg_inode_bitmap;                /* Inodes bitmap block */
140         __le32  bg_inode_table;         /* Inodes table block */
141         __le16  bg_free_blocks_count;   /* Free blocks count */
142         __le16  bg_free_inodes_count;   /* Free inodes count */
143         __le16  bg_used_dirs_count;     /* Directories count */
144         __le16  bg_pad;
145         __le32  bg_reserved[3];
146 };
```

bg_block_bitmap：存放 block bitmap 所在的 block 的索引。block bitmap 中的每一位（bit）用于记录每一个 block 的分配（used）或释放（free）。
bg_inode_bitmap：存放文件 inode 节点位图的块的索引，意义和结构与 bg_block_bitmap 相

似。

bg_inode_table：　文件 inode 节点表在硬盘中的第一个块的索引。

bg_free_blocks_count：可用的文件块数。

bg_free_inodes_count：可用的文件 inode 节点数。

bg_used_dirs_count：使用中的目录数。

bg_pad：　为了补齐上一个变量的后 16 位，32 位地址对齐。

（3）inode

ext2_inode 是 ext2 中非常重要的数据结构，它具有很多的用途，但最主要是用于管理和识别文件及目录。每一个 ext2_inode 结构包含文件的类型、操作权限、所有者、大小和分配给文件的数据块（data block）的索引。当用户请求对一个文件进行操作时，Linux 内核就将操作转化为相应的对物理存储介质的访问。Linux 在内存中使用 ext2_inode_info 来存放相应 ext2_inode 的信息，由 ext2_read_inode()函数将 ext2_inode 读入内存中生成。

ext2_inode 结构中有一项是一个指向一系列 block 的数组（见图 7.26），其大小 EXT2_N_BLOCKS 在文件系统编译时决定。对 ext2 现在所使用的 0.5b 版本而言，前 12（EXT2_IND_BLOCK = 12）个直接指向存放文件数据的 block 的索引。取 12 这个数是有根据的：研究表明 Linux 文件系统中绝大多数文件都很小，当被操作的范围在 12 个 block 内时，只需对 block 索引读取一次，这就大大提高了效率。第 13 个 block 索引指向一个 indirect block，indirect block 实际上包含了一列 block 的索引。如果 block 的大小是 1024 个字节，每个 block 索引占据 4 个字节，则从 block 13 到 block 268 大小范围内的数据需要两次操作方可访问到。相似的，第 14 个 block 索引指向一个 double indirect block（可以读写从 block 269 到 block 65804 大小范围内的数据）； 第 15 个 block 索引则指向一个由 double indirect block 组成的链表的表头。

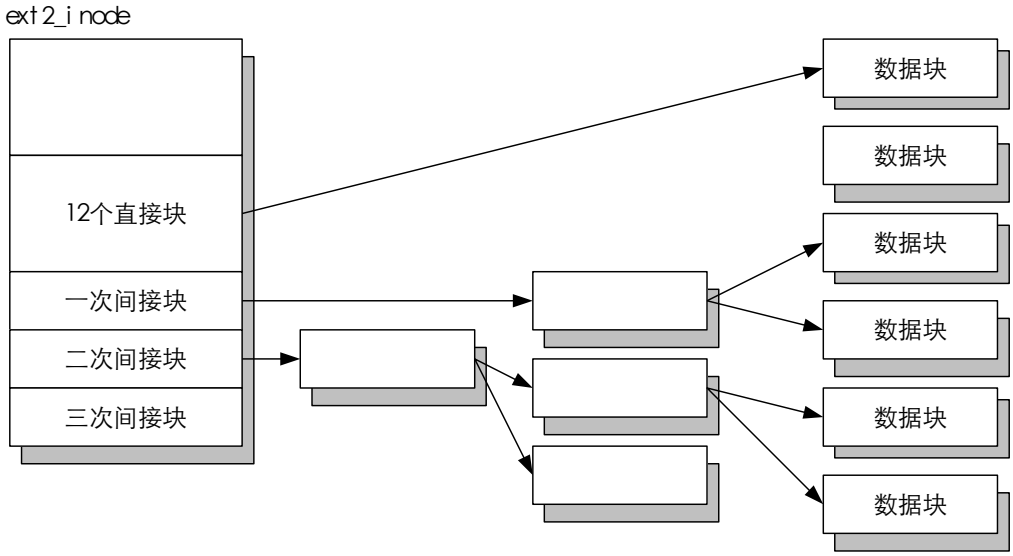

图 7.26 ext2_inode 结构

文件的 inode 结构，存在于外存中，供读入内存以建立 VFS inode。

```
include/linux/ext2_fs.h, line 211
211 struct ext2_inode {
212         __le16  i_mode;         /* File mode */
     ...
234         __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
     ...
263 };
```

i_mode: 文件模式，表示文件类型以及存取权限。在 ext2 中，inode 节点可以描述普通文件、目录文件、符号连接、块设备、字符设备或 FIFO 文件。

i_block: 文件块索引的数组，前 12 个指向物理块，后 3 个分别是一级、二级、三级间接指针。参见图 7.25。

由此可以粗略估计 ext2 的最大容量:

最大容量的计算和 block size 有关，三级间接指针所能寻址的最大 block 数目是：(block size / 4) ^{3} + (block size / 4) ^{2} + (block size / 4) + 11。

当 block 的为 1k 时，最大支持的磁盘容量约为（>）$2^{24} * 1k = 16G$;

当 block 的为 2k 时，最大支持的磁盘容量约为（>）$2^{27} * 2k = 256G$ ；

当 block 的为 4k 时，最大支持的磁盘容量约为（>）$2^{30} * 4k = 4T$。

这也是为什么我们在前面说当前版本的 ext2 所支持的最大分区的大小为 4T 的原因。

3. ext2 的操作实现

（1）超级块操作

```
fs/ext2/super.c, line 237
237 static struct super_operations ext2_sops = {
238         .alloc_inode    = ext2_alloc_inode,
239         .destroy_inode  = ext2_destroy_inode,
240         .read_inode     = ext2_read_inode,
241         .write_inode    = ext2_write_inode,
242         .put_inode      = ext2_put_inode,
243         .delete_inode   = ext2_delete_inode,
244         .put_super      = ext2_put_super,
245         .write_super    = ext2_write_super,
246         .statfs         = ext2_statfs,
```

```
247         .remount_fs     = ext2_remount,
248         .clear_inode    = ext2_clear_inode,
249         .show_options   = ext2_show_options,
...
254 };
```

ext2_read_inode 读文件节点操作。即将读入的 inode 的位置可以从入口传入的参数 inode 的相关属性计算得到。具体流程：从 inode->i_no 和 inode->i_sb 中求出文件块组号和所在组的描述符块的块号。从包含该文件块组的缓冲区中获取组描述符，再从描述符中计算得到文件数据所在的设备块号，将其读入缓存，最后将已经读入缓存的信息填入 inode。

ext2_write_super 写超级块操作。首先判断该超级块是否是只读的，对于可写的超级块 sb，获取其对应的 ext2 超级块，更新文件系统状态，更新安装时间等相关信息，清除超级块对应的“脏”标志（i_dirt）。

ext2_remount 重新安装文件系统。重新设定文件系统的读写状态。首先，解析安装进程的参数，判断安装参数是否已经发生变化。如果以读写的方式重新安装原为读写的文件系统，缓冲将被修改，并更改外存中的超级块，更新文件系统的载入时间。反之，如果以读写方式重新安装只读文件系统，设置文件系统超级块标志不再为只读。

（2）inode 操作

目录 inode 操作。

```
fs/ext2/namei.c, line 392
392 struct inode_operations ext2_dir_inode_operations = {
393         .create         = ext2_create,
394         .lookup         = ext2_lookup,
395         .link           = ext2_link,
396         .unlink         = ext2_unlink,
397         .symlink        = ext2_symlink,
398         .mkdir          = ext2_mkdir,
399         .rmdir          = ext2_rmdir,
400         .mknod          = ext2_mknod,
401         .rename         = ext2_rename,
...
408         .setattr        = ext2_setattr,
409         .permission     = ext2_permission,
410 };
```

ext2_create 建立新文件。首先为新文件建立新的 inode，指定索引文件节点的操作集为 ext2_file_inode_operations，通过 ext2_add_entry 为所在目录增添新目录项，如果有同

步要求，还需要将数据写回外存(由 ll_rw_block 完成)，最后填写 inode 信息。

ext2_mkdir 建立新目录。目录下生成名为“.”和“..”的两个子目录，分别指向当前目录和上一层目录，到此新建目录的连接数为 2，标识父目录缓冲区为“脏”，增添父目录连接数，最后填写 inode 信息。

ext2_rmdir 删除目录。判断该目录是否为空，非空目录不删除。找到包含该目录项的缓冲区，标识其为“脏”，如果要求同步操作，通过 ll_rw_block 将缓冲区刷新到设备上。重新设置对应的 inode 信息，包括大小为 0，引用数为 0 等，将该 inode 从目录索引中删除。

ext2_rename 重新命名文件。将原所在目录节点下的目录项重新命名为新目录下的目录项。首先完成权限和数据有效性检查，注意，在新文件节点加入新目录下成功之前，先减少新的文件节点的连接数，即去掉对上一层的目录的连接。所有更改过的缓冲区都标记为“脏”，如果要求同步操作，则立即将数据刷新到设备上。

4. ext2 数据块分配机制

在内存管理中，我们接触到了碎片问题。同样地，这个问题也存在于文件系统的管理当中。经过多次的读写操作后，属于同一文件的数据块可能会分散在文件系统的各个角落，导致对同一文件的串行访问效率降低。为了解决这个问题，ext2 有自己的数据块分配机制，我们看看它的具体内容。

ext2 文件系统为文件的扩展部分分配新数据块时，尽量先从文件原有数据块的附近寻找，至少使它们属于同一个块组。如果找不到，才从另外的块组寻找。

对一个文件进行写操作后，文件系统管理模块检查该文件的长度是否扩展了。如果扩展了，则需分配数据块。这时应首先锁定该文件系统的超级块，以保证其它进程不会读取错误的超级块信息。对超级块的申请采取 FCFS（First Come First Served）策略。

理想的选择是和文件最后一个数据块相连的数据块，如果该块已经分配，则从当前块相邻的 64 个块中寻找空块，否则从同一块组中寻找。再没找到就只好从别的块组中分配数据块。如果只能从其它块组搜索空闲数据块，则首先考虑 8 个一簇的连续块。

如果该文件系统采用了预分配机制，则当找到一个空块，接着的 8 个块都被保留（如果是空的）。如果分配的块使用了预分配的块，则需修改 i_prealloc_block 和 i_prealloc_count。

找到空闲块后应修改该数据块所在块组的块位图，分配一个数据缓冲区并初始化。初始化包括修正缓冲区 buffer_head 的设备号、块号，以及数据区清零。最后，超级块的 s_dirt 置位，说明超级块内容已更改，需写入设备。

文件被关闭后，被预分配但没用到的空块将被释放。