程序 12-4 linux/fs/inode.c

```
1  /*
2   *  linux/fs/inode.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8  #include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
9
10 #include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据，
                              // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <linux/mm.h>      // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
13 #include <asm/system.h>    // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
14
   // 设备数据块总数指针数组。每个指针项指向指定主设备号的总块数数组 hd_sizes[]。该总
   // 块数数组每一项对应子设备号确定的一个子设备上所拥有的数据块总数（1 块大小 = 1KB）。
15 extern int *blk_size[];
16
17 struct m_inode inode_table[NR_INODE]={{0,},};     // 内存中 i 节点表（NR_INODE=32 项）。
18
19 static void read_inode(struct m_inode * inode);  // 读指定 i 节点号的 i 节点信息，297 行。
20 static void write_inode(struct m_inode * inode); // 写 i 节点信息到高速缓冲中，324 行。
21
   //// 等待指定的 i 节点可用。
   // 如果 i 节点已被锁定，则将当前任务置为不可中断的等待状态，并添加到该 i 节点的等待队
   // 列 i_wait 中。直到该 i 节点解锁并明确地唤醒本任务。
22 static inline void wait_on_inode(struct m_inode * inode)
23 {
24         cli();
25         while (inode->i_lock)
26                 sleep_on(&inode->i_wait);          // kernel/sched.c，第 199 行。
27         sti();
28 }
29
   //// 对 i 节点上锁（锁定指定的 i 节点）。
   // 如果 i 节点已被锁定，则将当前任务置为不可中断的等待状态，并添加到该 i 节点的等待队
   // 列 i_wait 中。直到该 i 节点解锁并明确地唤醒本任务。然后对其上锁。
30 static inline void lock_inode(struct m_inode * inode)
31 {
32         cli();
33         while (inode->i_lock)
34                 sleep_on(&inode->i_wait);
35         inode->i_lock=1;                           // 置锁定标志。
36         sti();
37 }
38
   //// 对指定的 i 节点解锁。
   // 复位 i 节点的锁定标志，并明确地唤醒等待在此 i 节点等待队列 i_wait 上的所有进程。
39 static inline void unlock_inode(struct m_inode * inode)
40 {
41         inode->i_lock=0;
```

```
42          wake_up(&inode->i_wait);                // kernel/sched.c，第 204 行。
43  }
44
    //// 释放设备 dev 在内存 i 节点表中的所有 i 节点。
    // 扫描内存中的 i 节点表数组，如果是指定设备使用的 i 节点就释放之。
45  void invalidate_inodes(int dev)
46  {
47          int i;
48          struct m_inode * inode;
49
    // 首先让指针指向内存 i 节点表数组首项。然后扫描 i 节点表指针数组中的所有 i 节点。针对
    // 其中每个 i 节点，先等待该 i 节点解锁可用（若目前正被上锁的话），再判断是否属于指定
    // 设备的 i 节点。如果是指定设备的 i 节点，则看看它是否还被使用着，即其引用计数是否不
    // 为 0。若是则显示警告信息。然后释放之，即把 i 节点的设备号字段 i_dev 置 0。第 50 行上
    // 的指针赋值 "0+inode_table " 等同于 "inode_table"、"&inode_table[0] "。不过这样写
    // 可能更明了一些。
50          inode = 0+inode_table;                  // 指向 i 节点表指针数组首项。
51          for(i=0 ; i<NR_INODE ; i++,inode++) {
52                  wait_on_inode(inode);           // 等待该 i 节点可用（解锁）。
53                  if (inode->i_dev == dev) {
54                          if (inode->i_count)     // 若其引用数不为 0，则显示出错警告。
55                                  printk("inode in use on removed disk\n\r");
56                          inode->i_dev = inode->i_dirt = 0; // 释放 i 节点(置设备号为 0)。
57                  }
58          }
59  }
60
    //// 同步所有 i 节点。
    // 把内存 i 节点表中所有 i 节点与设备上 i 节点作同步操作。
61  void sync_inodes(void)
62  {
63          int i;
64          struct m_inode * inode;
65
    // 首先让内存 i 节点类型的指针指向 i 节点表首项，然后扫描整个 i 节点表中的节点。针对
    // 其中每个 i 节点，先等待该 i 节点解锁可用（若目前正被上锁的话），然后判断该 i 节点
    // 是否已被修改并且不是管道节点。若是这种情况则将该 i 节点写入高速缓冲区中。缓冲区
    // 管理程序 buffer.c 会在适当时机将它们写入盘中。
66          inode = 0+inode_table;                  // 让指针首先指向 i 节点表指针数组首项。
67          for(i=0 ; i<NR_INODE ; i++,inode++) {   // 扫描 i 节点表指针数组。
68                  wait_on_inode(inode);           // 等待该 i 节点可用（解锁）。
69                  if (inode->i_dirt && !inode->i_pipe)  // 若 i 节点已修改且不是管道节点，
70                          write_inode(inode);             // 则写盘（实际是写入缓冲区中）。
71          }
72  }
73
    //// 文件数据块映射到盘块的处理操作。（block 位图处理函数，bmap - block map）
    // 参数：inode - 文件的 i 节点指针；block - 文件中的数据块号；create - 创建块标志。
    // 该函数把指定的文件数据块 block 对应到设备上逻辑块上，并返回逻辑块号。如果创建标志
    // 置位，则在设备上对应逻辑块不存在时就申请新磁盘块，返回文件数据块 block 对应在设备
    // 上的逻辑块号（盘块号）。
74  static int _bmap(struct m_inode * inode,int block,int create)
75  {
```

```
76          struct buffer_head * bh;
77          int i;
78
    // 首先判断参数文件数据块号 block 的有效性。如果块号小于 0，则停机。如果块号大于直接
    // 块数 + 间接块数 + 二次间接块数，超出文件系统表示范围，则停机。
79          if (block<0)
80                  panic("_bmap: block<0");
81          if (block >= 7+512+512*512)
82                  panic("_bmap: block>big");
    // 然后根据文件块号的大小值和是否设置了创建标志分别进行处理。如果该块号小于 7，则使
    // 用直接块表示。如果创建标志置位，并且 i 节点中对应该块的逻辑块（区段）字段为 0，则
    // 向相应设备申请一磁盘块（逻辑块），并且将盘上逻辑块号（盘块号）填入逻辑块字段中。
    // 然后设置 i 节点改变时间，置 i 节点已修改标志。 最后返回逻辑块号。 函数 new_block()
    // 定义在 bitmap.c 程序中第 76 行开始处。
83          if (block<7) {
84                  if (create && !inode->i_zone[block])
85                          if (inode->i_zone[block]=new_block(inode->i_dev)) {
86                                  inode->i_ctime=CURRENT_TIME; // ctime - Change time。
87                                  inode->i_dirt=1;             // 设置已修改标志。
88                          }
89                  return inode->i_zone[block];
90          }
    // 如果该块号>=7，且小于 7+512，则说明使用的是一次间接块。下面对一次间接块进行处理。
    // 如果是创建，并且该 i 节点中对应间接块字段 i_zone[7]是 0，表明文件是首次使用间接块，
    // 则需申请一磁盘块用于存放间接块信息，并将此实际磁盘块号填入间接块字段中。 然后设
    // 置 i 节点已修改标志和修改时间。 如果创建时申请磁盘块失败，则此时 i 节点间接块字段
    // i_zone[7]为 0，则返回 0。或者不是创建，但 i_zone[7]原来就为 0，表明 i 节点中没有间
    // 接块，于是映射磁盘块失败，返回 0 退出。
91          block -= 7;
92          if (block<512) {
93                  if (create && !inode->i_zone[7])
94                          if (inode->i_zone[7]=new_block(inode->i_dev)) {
95                                  inode->i_dirt=1;
96                                  inode->i_ctime=CURRENT_TIME;
97                          }
98                  if (!inode->i_zone[7])
99                          return 0;
    // 现在读取设备上该 i 节点的一次间接块。并取该间接块上第 block 项中的逻辑块号（盘块
    // 号）i。每一项占 2 个字节。如果是创建并且间接块的第 block 项中的逻辑块号为 0 的话，
    // 则申请一磁盘块，并让间接块中的第 block 项等于该新逻辑块块号。然后置位间接块的已
    // 修改标志。如果不是创建，则 i 就是需要映射（寻找）的逻辑块号。
100                 if (!(bh = bread(inode->i_dev,inode->i_zone[7])))
101                         return 0;
102                 i = ((unsigned short *) (bh->b_data))[block];
103                 if (create && !i)
104                         if (i=new_block(inode->i_dev)) {
105                                 ((unsigned short *) (bh->b_data))[block]=i;
106                                 bh->b_dirt=1;
107                         }
    // 最后释放该间接块占用的缓冲块，并返回磁盘上新申请或原有的对应 block 的逻辑块块号。
108                 brelse(bh);
109                 return i;
110         }
```

```
    // 若程序运行到此，则表明数据块属于二次间接块。其处理过程与一次间接块类似。下面是对
    // 二次间接块的处理。首先将 block 再减去间接块所容纳的块数（512）。然后根据是否设置
    // 了创建标志进行创建或寻找处理。如果是新创建并且 i 节点的二次间接块字段为 0，则需申
    // 请一磁盘块用于存放二次间接块的一级块信息，并将此实际磁盘块号填入二次间接块字段
    // 中。之后，置 i 节点已修改编制和修改时间。同样地，如果创建时申请磁盘块失败，则此
    // 时 i 节点二次间接块字段 i_zone[8]为 0，则返回 0。或者不是创建，但 i_zone[8]原来就
    // 为 0，表明 i 节点中没有间接块，于是映射磁盘块失败，返回 0 退出。
111         block -= 512;
112         if (create && !inode->i_zone[8])
113                 if (inode->i_zone[8]=new_block(inode->i_dev)) {
114                         inode->i_dirt=1;
115                         inode->i_ctime=CURRENT_TIME;
116                 }
117         if (!inode->i_zone[8])
118                 return 0;
    // 现在读取设备上该 i 节点的二次间接块。并取该二次间接块的一级块上第 (block/512)
    // 项中的逻辑块号 i。如果是创建并且二次间接块的一级块上第 (block/512) 项中的逻辑
    // 块号为 0 的话，则需申请一磁盘块（逻辑块）作为二次间接块的二级块 i，并让二次间接
    // 块的一级块中第 (block/512)项等于该二级块的块号 i。然后置位二次间接块的一级块已
    // 修改标志。并释放二次间接块的一级块。如果不是创建，则 i 就是需要映射（寻找）的逻
    // 辑块号。
119         if (!(bh=bread(inode->i_dev,inode->i_zone[8])))
120                 return 0;
121         i = ((unsigned short *)bh->b_data)[block>>9];
122         if (create && !i)
123                 if (i=new_block(inode->i_dev)) {
124                         ((unsigned short *) (bh->b_data))[block>>9]=i;
125                         bh->b_dirt=1;
126                 }
127         brelse(bh);
    // 如果二次间接块的二级块块号为 0，表示申请磁盘块失败或者原来对应块号就为 0，则返
    // 回 0 退出。否则就从设备上读取二次间接块的二级块，并取该二级块上第 block 项中的逻
    // 辑块号（与上 511 是为了限定 block 值不超过 511）。
128         if (!i)
129                 return 0;
130         if (!(bh=bread(inode->i_dev,i)))
131                 return 0;
132         i = ((unsigned short *)bh->b_data)[block&511];
    // 如果是创建并且二级块的第 block 项中逻辑块号为 0 的话，则申请一磁盘块（逻辑块），
    // 作为最终存放数据信息的块。并让二级块中的第 block 项等于该新逻辑块块号(i)。然后
    // 置位二级块的已修改标志。
133         if (create && !i)
134                 if (i=new_block(inode->i_dev)) {
135                         ((unsigned short *) (bh->b_data))[block&511]=i;
136                         bh->b_dirt=1;
137                 }
    // 最后释放该二次间接块的二级块，返回磁盘上新申请的或原有的对应 block 的逻辑块块号。
138         brelse(bh);
139         return i;
140 }
141
    //// 取文件数据块 block 在设备上对应的逻辑块号。
    // 参数：inode - 文件的内存 i 节点指针；block - 文件中的数据块号。
```

```
    // 若操作成功则返回对应的逻辑块号，否则返回 0。
142 int bmap(struct m_inode * inode,int block)
143 {
144         return _bmap(inode,block,0);
145 }
146
    //// 取文件数据块 block 在设备上对应的逻辑块号。如果对应的逻辑块不存在就创建一块。
    // 并返回设备上对应的逻辑块号。
    // 参数：inode - 文件的内存 i 节点指针；block - 文件中的数据块号。
    // 若操作成功则返回对应的逻辑块号，否则返回 0。
147 int create_block(struct m_inode * inode, int block)
148 {
149         return _bmap(inode,block,1);
150 }
151
    //// 放回（放置）一个 i 节点(回写入设备)。
    // 该函数主要用于把 i 节点引用计数值递减 1，并且若是管道 i 节点，则唤醒等待的进程。
    // 若是块设备文件 i 节点则刷新设备。并且若 i 节点的链接计数为 0，则释放该 i 节点占用
    // 的所有磁盘逻辑块，并释放该 i 节点。
152 void iput(struct m_inode * inode)
153 {
    // 首先判断参数给出的 i 节点的有效性，并等待 inode 节点解锁（如果已上锁的话）。如果 i
    // 节点的引用计数为 0，表示该 i 节点已经是空闲的。内核再要求对其进行放回操作，说明内
    // 核中其他代码有问题。于是显示错误信息并停机。
154         if (!inode)
155                 return;
156         wait_on_inode(inode);
157         if (!inode->i_count)
158                 panic("iput: trying to free free inode");
    // 如果是管道 i 节点，则唤醒等待该管道的进程，引用次数减 1，如果还有引用则返回。否则
    // 释放管道占用的内存页面，并复位该节点的引用计数值、已修改标志和管道标志，并返回。
    // 对于管道节点，inode->i_size 存放着内存页地址。参见 get_pipe_inode()，231，237 行。
159         if (inode->i_pipe) {
160                 wake_up(&inode->i_wait);
161                 wake_up(&inode->i_wait2);          //
162                 if (--inode->i_count)
163                         return;
164                 free_page(inode->i_size);
165                 inode->i_count=0;
166                 inode->i_dirt=0;
167                 inode->i_pipe=0;
168                 return;
169         }
    // 如果 i 节点对应的设备号 = 0，则将此节点的引用计数递减 1，返回。例如用于管道操作的
    // i 节点，其 i 节点的设备号为 0。
170         if (!inode->i_dev) {
171                 inode->i_count--;
172                 return;
173         }
    // 如果是块设备文件的 i 节点，此时逻辑块字段 0（i_zone[0]）中是设备号，则刷新该设备。
    // 并等待 i 节点解锁。
174         if (S_ISBLK(inode->i_mode)) {
175                 sync_dev(inode->i_zone[0]);
```

```
176                 wait_on_inode(inode);
177         }
    // 如果 i 节点的引用计数大于 1，则计数递减 1 后就直接返回（因为该 i 节点还有人在用，不能
    // 释放），否则就说明 i 节点的引用计数值为 1（因为第 157 行已经判断过引用计数是否为零）。
    // 如果 i 节点的链接数为 0，则说明 i 节点对应文件被删除。于是释放该 i 节点的所有逻辑块，
    // 并释放该 i 节点。函数 free_inode()用于实际释放 i 节点操作，即复位 i 节点对应的 i 节点位
    // 图比特位，清空 i 节点结构内容。
178 repeat:
179         if (inode->i_count>1) {
180                 inode->i_count--;
181                 return;
182         }
183         if (!inode->i_nlinks) {
184                 truncate(inode);
185                 free_inode(inode);          // bitmap.c 第 108 行开始处。
186                 return;
187         }
    // 如果该 i 节点已作过修改，则回写更新该 i 节点，并等待该 i 节点解锁。由于这里在写 i 节
    // 点时需要等待睡眠，此时其他进程有可能修改该 i 节点，因此在进程被唤醒后需要再次重复
    // 进行上述判断过程（repeat）。
188         if (inode->i_dirt) {
189                 write_inode(inode);         /* we can sleep - so do again */
190                 wait_on_inode(inode);       /* 因为我们睡眠了，所以需要重复判断 */
191                 goto repeat;
192         }
    // 程序若能执行到此，则说明该 i 节点的引用计数值 i_count 是 1、链接数不为零，并且内容
    // 没有被修改过。因此此时只要把 i 节点引用计数递减 1，返回。此时该 i 节点的 i_count=0，
    // 表示已释放。
193         inode->i_count--;
194         return;
195 }
196
    //// 从 i 节点表（inode_table）中获取一个空闲 i 节点项。
    // 寻找引用计数 count 为 0 的 i 节点，并将其写盘后清零，返回其指针。引用计数被置 1。
197 struct m_inode * get_empty_inode(void)
198 {
199         struct m_inode * inode;
200         static struct m_inode * last_inode = inode_table;   // 指向 i 节点表第 1 项。
201         int i;
202
    // 在初始化 last_inode 指针指向 i 节点表头一项后循环扫描整个 i 节点表。如果 last_inode
    // 已经指向 i 节点表的最后 1 项之后，则让其重新指向 i 节点表开始处，以继续循环寻找空闲
    // i 节点项。如果 last_inode 所指向的 i 节点的计数值为 0，则说明可能找到空闲 i 节点项。
    // 让 inode 指向该 i 节点。如果该 i 节点的已修改标志和锁定标志均为 0，则我们可以使用该
    // i 节点，于是退出 for 循环。
203         do {
204                 inode = NULL;
205                 for (i = NR_INODE; i ; i--) {               // NR_INODE = 32。
206                         if (++last_inode >= inode_table + NR_INODE)
207                                 last_inode = inode_table;
208                         if (!last_inode->i_count) {
209                                 inode = last_inode;
210                                 if (!inode->i_dirt && !inode->i_lock)
```

```
211                                     break;
212                             }
213                     }
    // 如果没有找到空闲 i 节点（inode = NULL），则将 i 节点表打印出来供调试使用，并停机。
214                     if (!inode) {
215                             for (i=0 ; i<NR_INODE ; i++)
216                                     printk("%04x: %6d\t",inode_table[i].i_dev,
217                                             inode_table[i].i_num);
218                             panic("No free inodes in mem");
219                     }
    // 等待该 i 节点解锁（如果又被上锁的话）。如果该 i 节点已修改标志被置位的话，则将该
    // i 节点刷新（同步）。因为刷新时可能会睡眠，因此需要再次循环等待该 i 节点解锁。
220                     wait_on_inode(inode);
221                     while (inode->i_dirt) {
222                             write_inode(inode);
223                             wait_on_inode(inode);
224                     }
225             } while (inode->i_count);
    // 如果 i 节点又被其他占用的话（i 节点的计数值不为 0 了），则重新寻找空闲 i 节点。否则
    // 说明已找到符合要求的空闲 i 节点项。则将该 i 节点项内容清零，并置引用计数为 1，返回
    // 该 i 节点指针。
226             memset(inode,0,sizeof(*inode));
227             inode->i_count = 1;
228             return inode;
229     }
230
    //// 获取管道节点。
    // 首先扫描 i 节点表，寻找一个空闲 i 节点项，然后取得一页空闲内存供管道使用。然后将得
    // 到的 i 节点的引用计数置为 2(读者和写者)，初始化管道头和尾，置 i 节点的管道类型表示。
    // 返回为 i 节点指针，如果失败则返回 NULL。
231 struct m_inode * get_pipe_inode(void)
232 {
233             struct m_inode * inode;
234
    // 首先从内存 i 节点表中取得一个空闲 i 节点。如果找不到空闲 i 节点则返回 NULL。然后为该
    // i 节点申请一页内存，并让节点的 i_size 字段指向该页面。如果已没有空闲内存，则释放该
    // i 节点，并返回 NULL。
235             if (!(inode = get_empty_inode()))
236                     return NULL;
237             if (!(inode->i_size=get_free_page())) {     // 节点的 i_size 字段指向缓冲区。
238                     inode->i_count = 0;
239                     return NULL;
240             }
    // 然后设置该 i 节点的引用计数为 2，并复位复位管道头尾指针。i 节点逻辑块号数组 i_zone[]
    // 的 i_zone[0]和 i_zone[1]中分别用来存放管道头和管道尾指针。最后设置 i 节点是管道 i 节
    // 点标志并返回该 i 节点号。
241             inode->i_count = 2;     /* sum of readers/writers */   /* 读/写两者总计 */
242             PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0;  // 复位管道头尾指针。
243             inode->i_pipe = 1;                          // 置节点为管道使用的标志。
244             return inode;
245     }
246
    //// 取得一个 i 节点。
```

```
    // 参数：dev - 设备号；nr - i 节点号。
    // 从设备上读取指定节点号的 i 节点结构内容到内存 i 节点表中，并且返回该 i 节点指针。
    // 首先在位于高速缓冲区中的 i 节点表中搜寻，若找到指定节点号的 i 节点则在经过一些判断
    // 处理后返回该 i 节点指针。否则从设备 dev 上读取指定 i 节点号的 i 节点信息放入 i 节点表
    // 中，并返回该 i 节点指针。
247 struct m_inode * iget(int dev,int nr)
248 {
249         struct m_inode * inode, * empty;
250
    // 首先判断参数有效性。若设备号是 0，则表明内核代码问题，显示出错信息并停机。然后预
    // 先从 i 节点表中取一个空闲 i 节点备用。
251         if (!dev)
252                 panic("iget with dev==0");
253         empty = get_empty_inode();
    // 接着扫描 i 节点表。寻找参数指定节点号 nr 的 i 节点。并递增该节点的引用次数。如果当
    // 前扫描 i 节点的设备号不等于指定的设备号或者节点号不等于指定的节点号，则继续扫描。
254         inode = inode_table;
255         while (inode < NR_INODE+inode_table) {
256                 if (inode->i_dev != dev || inode->i_num != nr) {
257                         inode++;
258                         continue;
259                 }
    // 如果找到指定设备号 dev 和节点号 nr 的 i 节点，则等待该节点解锁（如果已上锁的话）。
    // 在等待该节点解锁过程中，i 节点表可能会发生变化。所以再次进行上述相同判断。如果发
    // 生了变化，则再次重新扫描整个 i 节点表。
260                 wait_on_inode(inode);
261                 if (inode->i_dev != dev || inode->i_num != nr) {
262                         inode = inode_table;
263                         continue;
264                 }
    // 到这里表示找到相应的 i 节点。于是将该 i 节点引用计数增 1。然后再作进一步检查，看它
    // 是否是另一个文件系统的安装点。若是则寻找被安装文件系统根节点并返回。如果该 i 节点
    // 的确是其他文件系统的安装点，则在超级块表中搜寻安装在此 i 节点的超级块。如果没有找
    // 到，则显示出错信息，并放回本函数开始时获取的空闲节点 empty，返回该 i 节点指针。
265                 inode->i_count++;
266                 if (inode->i_mount) {
267                         int i;
268
269                         for (i = 0 ; i<NR_SUPER ; i++)
270                                 if (super_block[i].s_imount==inode)
271                                         break;
272                         if (i >= NR_SUPER) {
273                                 printk("Mounted inode hasn't got sb\n");
274                                 if (empty)
275                                         iput(empty);
276                                 return inode;
277                         }
    // 执行到这里表示已经找到安装到 inode 节点的文件系统超级块。于是将该 i 节点写盘放回，
    // 并从安装在此 i 节点上的文件系统超级块中取设备号，并令 i 节点号为 ROOT_INO，即为 1。
    // 然后重新扫描整个 i 节点表，以获取该被安装文件系统的根 i 节点信息。
278                         iput(inode);
279                         dev = super_block[i].s_dev;
280                         nr = ROOT_INO;
```

```
281                         inode = inode_table;
282                         continue;
283                 }
    // 最终我们找到了相应的 i 节点。因此可以放弃本函数开始处临时申请的空闲 i 节点，返回
    // 找到的 i 节点指针。
284                 if (empty)
285                         iput(empty);
286                 return inode;
287         }
    // 如果我们在 i 节点表中没有找到指定的 i 节点，则利用前面申请的空闲 i 节点 empty 在 i
    // 节点表中建立该 i 节点。并从相应设备上读取该 i 节点信息，返回该 i 节点指针。
288         if (!empty)
289                 return (NULL);
290         inode=empty;
291         inode->i_dev = dev;                     // 设置 i 节点的设备。
292         inode->i_num = nr;                      // 设置 i 节点号。
293         read_inode(inode);
294         return inode;
295 }
296
    //// 读取指定 i 节点信息。
    // 从设备上读取含有指定 i 节点信息的 i 节点盘块，然后复制到指定的 i 节点结构中。 为了
    // 确定 i 节点所在的设备逻辑块号（或缓冲块），必须首先读取相应设备上的超级块，以获取
    // 用于计算逻辑块号的每块 i 节点数信息 INODES_PER_BLOCK。 在计算出 i 节点所在的逻辑块
    // 号后，就把该逻辑块读入一缓冲块中。然后把缓冲块中相应位置处的 i 节点内容复制到参数
    // 指定的位置处。
297 static void read_inode(struct m_inode * inode)
298 {
299         struct super_block * sb;
300         struct buffer_head * bh;
301         int block;
302
    // 首先锁定该 i 节点，并取该节点所在设备的超级块。
303         lock_inode(inode);
304         if (!(sb=get_super(inode->i_dev)))
305                 panic("trying to read inode without dev");
    // 该 i 节点所在的设备逻辑块号 = (启动块 + 超级块) + i 节点位图占用的块数 + 逻辑块位
    // 图占用的块数 + (i 节点号-1)/每块含有的 i 节点数。虽然 i 节点号从 0 开始编号，但第 1
    // 个 0 号 i 节点不用，并且磁盘上也不保存对应的 0 号 i 节点结构。因此存放 i 节点的盘块的
    // 第 1 块上保存的是 i 节点号是是 1--16 的 i 节点结构而不是 0--15 的。因此在上面计算 i 节
    // 点号对应的 i 节点结构所在盘块时需要减 1，即：B =(i 节点号-1)/每块含有 i 节点结构数。
    // 例如，节点号 16 的 i 节点结构应该在 B=(16-1)/16 = 0 的块上。 这里我们从设备上读取该
    // i 节点所在的逻辑块，并复制指定 i 节点内容到 inode 指针所指位置处。
306         block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
307                 (inode->i_num-1)/INODES_PER_BLOCK;
308         if (!(bh=bread(inode->i_dev,block)))
309                 panic("unable to read i-node block");
310         *(struct d_inode *)inode =
311                 ((struct d_inode *)bh->b_data)
312                         [(inode->i_num-1)%INODES_PER_BLOCK];
    // 最后释放读入的缓冲块，并解锁该 i 节点。对于块设备文件，还需要设置 i 节点的文件最大
    // 长度值。
313         brelse(bh);
```

```
314         if (S_ISBLK(inode->i_mode)) {
315                 int i = inode->i_zone[0];   // 对于块设备文件，i_zone[0]中是设备号。
316                 if (blk_size[MAJOR(i)])
317                         inode->i_size = 1024*blk_size[MAJOR(i)][MINOR(i)];
318                 else
319                         inode->i_size = 0x7fffffff;
320         }
321         unlock_inode(inode);
322 }
323
    //// 将 i 节点信息写入缓冲区中。
    // 该函数把参数指定的 i 节点写入缓冲区相应的缓冲块中，待缓冲区刷新时会写入盘中。为了
    // 确定 i 节点所在的设备逻辑块号（或缓冲块），必须首先读取相应设备上的超级块，以获取
    // 用于计算逻辑块号的每块 i 节点数信息 INODES_PER_BLOCK。 在计算出 i 节点所在的逻辑块
    // 号后，就把该逻辑块读入一缓冲块中。然后把 i 节点内容复制到缓冲块的相应位置处。
324 static void write_inode(struct m_inode * inode)
325 {
326         struct super_block * sb;
327         struct buffer_head * bh;
328         int block;
329
    // 首先锁定该 i 节点，如果该 i 节点没有被修改过或者该 i 节点的设备号等于零，则解锁该
    // i 节点，并退出。对于没有被修改过的 i 节点，其内容与缓冲区中或设备中的相同。 然后
    // 获取该 i 节点的超级块。
330         lock_inode(inode);
331         if (!inode->i_dirt || !inode->i_dev) {
332                 unlock_inode(inode);
333                 return;
334         }
335         if (!(sb=get_super(inode->i_dev)))
336                 panic("trying to write inode without device");
    // 该 i 节点所在的设备逻辑块号 = (启动块 + 超级块) + i 节点位图占用的块数 + 逻辑块位
    // 图占用的块数 + (i 节点号-1)/每块含有的 i 节点数。  我们从设备上读取该 i 节点所在的
    // 逻辑块，并将该 i 节点信息复制到逻辑块对应该 i 节点的项位置处。
337         block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
338                 (inode->i_num-1)/INODES_PER_BLOCK;
339         if (!(bh=bread(inode->i_dev,block)))
340                 panic("unable to read i-node block");
341         ((struct d_inode *)bh->b_data)
342                 [(inode->i_num-1)%INODES_PER_BLOCK] =
343                         *(struct d_inode *)inode;
    // 然后置缓冲区已修改标志，而 i 节点内容已经与缓冲区中的一致，因此修改标志置零。然后
    // 释放该含有 i 节点的缓冲区，并解锁该 i 节点。
344         bh->b_dirt=1;
345         inode->i_dirt=0;
346         brelse(bh);
347         unlock_inode(inode);
348 }
349
```