```
1  /*
2   *  linux/fs/super.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   * super.c contains code to handle the super-block tables.
9   */
   /*
    * super.c 程序中含有处理超级块表的代码。
    */
10 #include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
11 #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。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
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
15 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
16 #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。
17
   // 对指定设备执行高速缓冲与设备上数据的同步操作（fs/buffer.c，59 行）。
18 int sync_dev(int dev);
   // 等待击键（kernel/chr_drv/tty_io.c，140 行）。
19 void wait_for_keypress(void);
20
21 /* set_bit uses setb, as gas doesn't recognize setc */
   /* set_bit()使用了 setb 指令，因为汇编编译器 gas 不能识别指令 setc */
   //// 测试指定位偏移处比特位的值，并返回该原比特位值（应该取名为 test_bit()更妥帖）。
   // 嵌入式汇编宏。参数 bitnr 是比特位偏移值，addr 是测试比特位操作的起始地址。
   // %0 - ax(__res)，%1 - 0，%2 - bitnr，%3 - addr
   // 第 23 行定义了一个局部寄存器变量。该变量将被保存在 eax 寄存器中，以便于高效访问和
   // 操作。第 24 行上指令 bt 用于对比特位进行测试（Bit Test）。它会把地址 addr（%3）和
   // 比特位偏移量 bitnr（%2）指定的比特位的值放入进位标志 CF 中。 指令 setb 用于根据进
   // 位标志 CF 设置操作数%al。如果 CF = 1 则%al = 1，否则%al = 0。
22 #define set_bit(bitnr,addr) ({ \
23 register int __res __asm__("ax"); \
24 __asm__("bt %2,%3;setb %%al":"=a" (__res):"a" (0),"r" (bitnr),"m" (*(addr))); \
25 __res; })
26
27 struct super_block super_block[NR_SUPER];   // 超级块结构表数组（NR_SUPER = 8）。
28 /* this is initialized in init/main.c */
   /* ROOT_DEV 已在 init/main.c 中被初始化 */
29 int ROOT_DEV = 0;                           // 根文件系统设备号。
30
   // 以下 3 个函数（lock_super()、free_super()和 wait_on_super()）的作用与 inode.c 文
   // 件中头 3 个函数的作用雷同，只是这里操作的对象换成了超级块。
   //// 锁定超级块。
   // 如果超级块已被锁定，则将当前任务置为不可中断的等待状态，并添加到该超级块等待队
   // 列 s_wait 中。直到该超级块解锁并明确地唤醒本任务。然后对其上锁。
31 static void lock_super(struct super_block * sb)
32 {
```

```
33          cli();                                  // 关中断。
34          while (sb->s_lock)                      // 如果该超级块已经上锁，则睡眠等待。
35                  sleep_on(&(sb->s_wait));        // kernel/sched.c，第 199 行。
36          sb->s_lock = 1;                         // 给该超级块加锁（置锁定标志）。
37          sti();                                  // 开中断。
38  }
39
    //// 对指定超级块解锁。
    // 复位超级块的锁定标志，并明确地唤醒等待在此超级块等待队列 s_wait 上的所有进程。
    // 如果使用 ulock_super 这个名称则可能更妥帖。
40  static void free_super(struct super_block * sb)
41  {
42          cli();
43          sb->s_lock = 0;                         // 复位锁定标志。
44          wake_up(&(sb->s_wait));                 // 唤醒等待该超级块的进程。
45          sti();                                  // wake_up()在 kernel/sched.c，第 188 行。
46  }
47
    //// 睡眠等待超级块解锁。
    // 如果超级块已被锁定，则将当前任务置为不可中断的等待状态，并添加到该超级块的等待队
    // 列 s_wait 中。直到该超级块解锁并明确地唤醒本任务。
48  static void wait_on_super(struct super_block * sb)
49  {
50          cli();
51          while (sb->s_lock)                      // 如果超级块已经上锁，则睡眠等待。
52                  sleep_on(&(sb->s_wait));
53          sti();
54  }
55
    //// 取指定设备的超级块。
    // 在超级块表（数组）中搜索指定设备 dev 的超级块结构信息。若找到则返回超级块的指针，
    // 否则返回空指针。
56  struct super_block * get_super(int dev)
57  {
58          struct super_block * s;                 // s 是超级块数据结构指针。
59
    // 首先判断参数给出设备的有效性。若设备号为 0 则返回空指针。然后让 s 指向超级块数组
    // 起始处，开始搜索整个超级块数组，以寻找指定设备 dev 的超级块。 第 62 行上的指针赋
    // 值语句"s = 0+super_block" 等同于 "s = super_block"、"s = &super_block[0]"。
60          if (!dev)
61                  return NULL;
62          s = 0+super_block;
63          while (s < NR_SUPER+super_block)
    // 如果当前搜索项是指定设备的超级块，即该超级块的设备号字段值与函数参数指定的相同，
    // 则先等待该超级块解锁（若已被其他进程上锁的话）。在等待期间，该超级块项有可能被
    // 其他设备使用，因此等待返回之后需再判断一次是否是指定设备的超级块，如果是则返回
    // 该超级块的指针。否则就重新对超级块数组再搜索一遍，因此此时 s 需重又指向超级块数
    // 组开始处。
64                  if (s->s_dev == dev) {
65                          wait_on_super(s);
66                          if (s->s_dev == dev)
67                                  return s;
68                          s = 0+super_block;
```

```
    // 如果当前搜索项不是，则检查下一项。如果没有找到指定的超级块，则返回空指针。
 69                 } else
 70                         s++;
 71         return NULL;
 72 }
 73
    //// 释放（放回）指定设备的超级块。
    // 释放设备所使用的超级块数组项（置 s_dev=0），并释放该设备 i 节点位图和逻辑块位图所
    // 占用的高速缓冲块。如果超级块对应的文件系统是根文件系统，或者其某个 i 节点上已经安
    // 装有其他的文件系统，则不能释放该超级块。
 74 void put_super(int dev)
 75 {
 76         struct super_block * sb;
 77         int i;
 78
    // 首先判断参数的有效性和合法性。如果指定设备是根文件系统设备，则显示警告信息“根系
    // 统盘改变了，准备生死决战吧”，并返回。然后在超级块表中寻找指定设备号的文件系统超
    // 级块。如果找不到指定设备的超级块，则返回。另外，如果该超级块指明该文件系统所安装
    // 到的 i 节点还没有被处理过，则显示警告信息并返回。在文件系统卸载（umount）操作中，
    // s_imount 会先被置成 Null 以后才会调用本函数，参见第 192 行。
 79         if (dev == ROOT_DEV) {
 80                 printk("root diskette changed: prepare for armageddon\n\r");
 81                 return;
 82         }
 83         if (!(sb = get_super(dev)))
 84                 return;
 85         if (sb->s_imount) {
 86                 printk("Mounted disk changed - tssk, tssk\n\r");
 87                 return;
 88         }
    // 然后在找到指定设备的超级块之后，我们先锁定该超级块，再置该超级块对应的设备号字段
    // s_dev 为 0，也即释放该设备上的文件系统超级块。然后释放该超级块占用的其他内核资源，
    // 即释放该设备上文件系统 i 节点位图和逻辑块位图在缓冲区中所占用的缓冲块。下面常数符
    // 号 I_MAP_SLOTS 和 Z_MAP_SLOTS 均等于 8，用于分别指明 i 节点位图和逻辑块位图占用的磁
    // 盘逻辑块数。注意，若这些缓冲块内容被修改过，则需要作同步操作才能把缓冲块中的数据
    // 写入设备中。函数最后对该超级块解锁，并返回。
 89         lock_super(sb);
 90         sb->s_dev = 0;                          // 置超级块空闲。
 91         for(i=0;i<I_MAP_SLOTS;i++)
 92                 brelse(sb->s_imap[i]);
 93         for(i=0;i<Z_MAP_SLOTS;i++)
 94                 brelse(sb->s_zmap[i]);
 95         free_super(sb);
 96         return;
 97 }
 98
    //// 读取指定设备的超级块。
    // 如果指定设备 dev 上的文件系统超级块已经在超级块表中，则直接返回该超级块项的指针。
    // 否则就从设备 dev 上读取超级块到缓冲块中，并复制到超级块表中。并返回超级块指针。
 99 static struct super_block * read_super(int dev)
100 {
101         struct super_block * s;
102         struct buffer_head * bh;
```

```
103         int i,block;
104
    // 首先判断参数的有效性。如果没有指明设备，则返回空指针。然后检查该设备是否可更换
    // 过盘片（也即是否是软盘设备）。如果更换过盘，则高速缓冲区有关该设备的所有缓冲块
    // 均失效，需要进行失效处理，即释放原来加载的文件系统。
105         if (!dev)
106                 return NULL;
107         check_disk_change(dev);
    // 如果该设备的超级块已经在超级块表中，则直接返回该超级块的指针。否则，首先在超级
    // 块数组中找出一个空项(也即字段 s_dev=0 的项)。如果数组已经占满则返回空指针。
108         if (s = get_super(dev))
109                 return s;
110         for (s = 0+super_block ;; s++) {
111                 if (s >= NR_SUPER+super_block)
112                         return NULL;
113                 if (!s->s_dev)
114                         break;
115         }
    // 在超级块数组中找到空项之后，就将该超级块项用于指定设备 dev 上的文件系统。于是对
    // 该超级块结构中的内存字段进行部分初始化处理。
116         s->s_dev = dev;                         // 用于 dev 设备上的文件系统。
117         s->s_isup = NULL;
118         s->s_imount = NULL;
119         s->s_time = 0;
120         s->s_rd_only = 0;
121         s->s_dirt = 0;
    // 然后锁定该超级块，并从设备上读取超级块信息到 bh 指向的缓冲块中。超级块位于块设备
    // 的第 2 个逻辑块（1 号块）中，（第 1 个是引导盘块）。如果读超级块操作失败，则释放上
    // 面选定的超级块数组中的项（即置 s_dev=0），并解锁该项，返回空指针退出。否则就将设
    // 备上读取的超级块信息从缓冲块数据区复制到超级块数组相应项结构中。并释放存放读取信
    // 息的高速缓冲块。
122         lock_super(s);
123         if (!(bh = bread(dev,1))) {
124                 s->s_dev=0;
125                 free_super(s);
126                 return NULL;
127         }
128         *((struct d_super_block *) s) =
129                 *((struct d_super_block *) bh->b_data);
130         brelse(bh);
    // 现在我们从设备 dev 上得到了文件系统的超级块，于是开始检查这个超级块的有效性并从设
    // 备上读取 i 节点位图和逻辑块位图等信息。如果所读取的超级块的文件系统魔数字段不对，
    // 说明设备上不是正确的文件系统，因此同上面一样，释放上面选定的超级块数组中的项，并
    // 解锁该项，返回空指针退出。对于该版 Linux 内核，只支持 MINIX 文件系统 1.0 版本，其魔
    // 数是 0x137f。
131         if (s->s_magic != SUPER_MAGIC) {
132                 s->s_dev = 0;
133                 free_super(s);
134                 return NULL;
135         }
    // 下面开始读取设备上 i 节点位图和逻辑块位图数据。首先初始化内存超级块结构中位图空间。
    // 然后从设备上读取 i 节点位图和逻辑块位图信息，并存放在超级块对应字段中。i 节点位图
    // 保存在设备上 2 号块开始的逻辑块中，共占用 s_imap_blocks 个块。逻辑块位图在 i 节点位
```

```
    // 图所在块的后续块中，共占用 s_zmap_blocks 个块。
136         for (i=0;i<I_MAP_SLOTS;i++)                 // 初始化操作。
137                 s->s_imap[i] = NULL;
138         for (i=0;i<Z_MAP_SLOTS;i++)
139                 s->s_zmap[i] = NULL;
140         block=2;
141         for (i=0 ; i < s->s_imap_blocks ; i++)      // 读取设备中 i 节点位图。
142                 if (s->s_imap[i]=bread(dev,block))
143                         block++;
144                 else
145                         break;
146         for (i=0 ; i < s->s_zmap_blocks ; i++)      // 读取设备中逻辑块位图。
147                 if (s->s_zmap[i]=bread(dev,block))
148                         block++;
149                 else
150                         break;
    // 如果读出的位图块数不等于位图应该占有的逻辑块数，说明文件系统位图信息有问题，超级
    // 块初始化失败。因此只能释放前面申请并占用的所有资源，即释放 i 节点位图和逻辑块位图
    // 占用的高速缓冲块、释放上面选定的超级块数组项、解锁该超级块项，并返回空指针退出。
151         if (block != 2+s->s_imap_blocks+s->s_zmap_blocks) {
152                 for(i=0;i<I_MAP_SLOTS;i++)          // 释放位图占用的高速缓冲块。
153                         brelse(s->s_imap[i]);
154                 for(i=0;i<Z_MAP_SLOTS;i++)
155                         brelse(s->s_zmap[i]);
156                 s->s_dev=0;                         // 释放选定的超级块数组项。
157                 free_super(s);                      // 解锁该超级块项。
158                 return NULL;
159         }
    // 否则一切成功。另外，由于对于申请空闲 i 节点的函数来讲，如果设备上所有的 i 节点已经
    // 全被使用，则查找函数会返回 0 值。因此 0 号 i 节点是不能用的，所以这里将位图中第 1 块
    // 的最低比特位设置为 1，以防止文件系统分配 0 号 i 节点。同样的道理，也将逻辑块位图的
    // 最低位设置为 1。最后函数解锁该超级块，并返回超级块指针。
160         s->s_imap[0]->b_data[0] |= 1;
161         s->s_zmap[0]->b_data[0] |= 1;
162         free_super(s);
163         return s;
164 }
165
    //// 卸载文件系统（系统调用）。
    // 参数 dev_name 是文件系统所在设备的设备文件名。
    // 该函数首先根据参数给出的块设备文件名获得设备号，然后复位文件系统超级块中的相应字
    // 段，释放超级块和位图占用的缓冲块，最后对该设备执行高速缓冲与设备上数据的同步操作。
    // 若卸载操作成功则返回 0，否则返回出错码。
166 int sys_umount(char * dev_name)
167 {
168         struct m_inode * inode;
169         struct super_block * sb;
170         int dev;
171
    // 首先根据设备文件名找到对应的 i 节点，并取其中的设备号。设备文件所定义设备的设备号
    // 是保存在其 i 节点的 i_zone[0]中的。 参见后面 namei.c 程序中系统调用 sys_mknod()的代
    // 码第 445 行。另外，由于文件系统需要存放在块设备上，因此如果不是块设备文件，则放回
    // 刚申请的 i 节点 dev_i，返回出错码。
```

```
172         if (!(inode=namei(dev_name)))
173                 return -ENOENT;
174         dev = inode->i_zone[0];
175         if (!S_ISBLK(inode->i_mode)) {
176                 iput(inode);                            // fs/inode.c，第 150 行。
177                 return -ENOTBLK;
178         }
    // OK，现在上面为了得到设备号而取得的 i 节点已完成了它的使命，因此这里放回该设备文件
    // 的 i 节点。接着我们来检查一下卸载该文件系统的条件是否满足。如果设备上是根文件系统，
    // 则不能被卸载，返回忙出错号。
179         iput(inode);
180         if (dev==ROOT_DEV)
181                 return -EBUSY;
    // 如果在超级块表中没有找到该设备上文件系统的超级块，或者已找到但是该设备上文件系统
    // 没有安装过，则返回出错码。如果超级块所指明的被安装到的 i 节点并没有置位其安装标志
    // i_mount，则显示警告信息。 然后查找一下 i 节点表，看看是否有进程在使用该设备上的文
    // 件，如果有则返回忙出错码。
182         if (!(sb=get_super(dev)) || !(sb->s_imount))
183                 return -ENOENT;
184         if (!sb->s_imount->i_mount)
185                 printk("Mounted inode has i_mount=0\n");
186         for (inode=inode_table+0 ; inode<inode_table+NR_INODE ; inode++)
187                 if (inode->i_dev==dev && inode->i_count)
188                                 return -EBUSY;
    // 现在该设备上文件系统的卸载条件均得到满足，因此我们可以开始实施真正的卸载操作了。
    // 首先复位被安装到的 i 节点的安装标志，释放该 i 节点。然后置超级块中被安装 i 节点字段
    // 为空，并放回设备文件系统的根 i 节点，接着置超级块中被安装系统根 i 节点指针为空。
189         sb->s_imount->i_mount=0;
190         iput(sb->s_imount);
191         sb->s_imount = NULL;
192         iput(sb->s_isup);
193         sb->s_isup = NULL;
    // 最后我们释放该设备上的超级块以及位图占用的高速缓冲块，并对该设备执行高速缓冲与设
    // 备上数据的同步操作。然后返回 0（卸载成功）。
194         put_super(dev);
195         sync_dev(dev);
196         return 0;
197 }
198
    //// 安装文件系统（系统调用）。
    // 参数 dev_name 是设备文件名，dir_name 是安装到的目录名，rw_flag 被安装文件系统的可
    // 读写标志。将被加载的地方必须是一个目录名，并且对应的 i 节点没有被其他程序占用。
    // 若操作成功则返回 0，否则返回出错号。
199 int sys_mount(char * dev_name, char * dir_name, int rw_flag)
200 {
201         struct m_inode * dev_i, * dir_i;
202         struct super_block * sb;
203         int dev;
204
    // 首先根据设备文件名找到对应的 i 节点，以取得其中的设备号。对于块特殊设备文件，设备
    // 号在其 i 节点的 i_zone[0]中。另外，由于文件系统必须在块设备中，因此如果不是块设备
    // 文件，则放回刚取得的 i 节点 dev_i，返回出错码。
205         if (!(dev_i=namei(dev_name)))
```

```
206                 return -ENOENT;
207         dev = dev_i->i_zone[0];
208         if (!S_ISBLK(dev_i->i_mode)) {
209                 iput(dev_i);
210                 return -EPERM;
211         }
    // OK，现在上面为了得到设备号而取得的 i 节点 dev_i 已完成了它的使命，因此这里放回该设
    // 备文件的 i 节点。接着我们来检查一下文件系统安装到的目录名是否有效。于是根据给定的
    // 目录文件名找到对应的 i 节点 dir_i。 如果该 i 节点的引用计数不为 1（仅在这里引用），
    // 或者该 i 节点的节点号是根文件系统的节点号 1，则放回该 i 节点返回出错码。另外，如果
    // 该节点不是一个目录文件节点，则也放回该 i 节点，返回出错码。因为文件系统只能安装在
    // 一个目录名上。
212         iput(dev_i);
213         if (!(dir_i=namei(dir_name)))
214                 return -ENOENT;
215         if (dir_i->i_count != 1 || dir_i->i_num == ROOT_INO) {
216                 iput(dir_i);
217                 return -EBUSY;
218         }
219         if (!S_ISDIR(dir_i->i_mode)) {                  // 安装点需要是一个目录名。
220                 iput(dir_i);
221                 return -EPERM;
222         }
    // 现在安装点也检查完毕，我们开始读取要安装文件系统的超级块信息。如果读超级块操作失
    // 败，则放回该安装点 i 节点 dir_i 并返回出错码。一个文件系统的超级块会首先从超级块表
    // 中进行搜索，如果不在超级块表中就从设备上读取。
223         if (!(sb=read_super(dev))) {
224                 iput(dir_i);
225                 return -EBUSY;
226         }
    // 在得到了文件系统超级块之后，我们对它先进行检测一番。如果将要被安装的文件系统已经
    // 安装在其他地方，则放回该 i 节点，返回出错码。如果将要安装到的 i 节点已经安装了文件
    // 系统（安装标志已经置位），则放回该 i 节点，也返回出错码。
227         if (sb->s_imount) {
228                 iput(dir_i);
229                 return -EBUSY;
230         }
231         if (dir_i->i_mount) {
232                 iput(dir_i);
233                 return -EPERM;
234         }
    // 最后设置被安装文件系统超级块的“被安装到 i 节点”字段指向安装到的目录名的 i 节点。
    // 并设置安装位置 i 节点的安装标志和节点已修改标志。然后返回 0（安装成功）。
235         sb->s_imount=dir_i;
236         dir_i->i_mount=1;
237         dir_i->i_dirt=1;/* NOTE! we don't iput(dir_i) */ /*注意!这里没用 iput(dir_i)*/
238         return 0;       /* we do that in umount */       /* 这将在 umount 内操作 */
239 }
240
    //// 安装根文件系统。
    // 该函数属于系统初始化操作的一部分。函数首先初始化文件表数组 file_table[]和超级块表
    // （数组），然后读取根文件系统超级块，并取得文件系统根 i 节点。 最后统计并显示出根文
    // 件系统上的可用资源（空闲块数和空闲 i 节点数）。 该函数会在系统开机进行初始化设置时
```

```
    // （sys_setup()）调用（blk_drv/hd.c，157 行）。
241 void mount_root(void)
242 {
243         int i,free;
244         struct super_block * p;
245         struct m_inode * mi;
246
    // 若磁盘 i 节点结构不是 32 字节，则出错停机。该判断用于防止修改代码时出现不一致情况。
247         if (32 != sizeof (struct d_inode))
248                 panic("bad i-node size");
    // 首先初始化文件表数组（共 64 项，即系统同时只能打开 64 个文件）和超级块表。这里将所
    // 有文件结构中的引用计数设置为 0（表示空闲），并把超级块表中各项结构的设备字段初始
    // 化为 0（也表示空闲）。如果根文件系统所在设备是软盘的话，就提示“插入根文件系统盘，
    // 并按回车键”，并等待按键。
249         for(i=0;i<NR_FILE;i++)                          // 初始化文件表。
250                 file_table[i].f_count=0;
251         if (MAJOR(ROOT_DEV) == 2) {                     // 提示插入根文件系统盘。
252                 printk("Insert root floppy and press ENTER");
253                 wait_for_keypress();
254         }
255         for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {
256                 p->s_dev = 0;                           // 初始化超级块表。
257                 p->s_lock = 0;
258                 p->s_wait = NULL;
259         }
    // 做好以上“份外”的初始化工作之后，我们开始安装根文件系统。于是从根设备上读取文件
    // 系统超级块，并取得文件系统的根 i 节点（1 号节点）在内存 i 节点表中的指针。如果读根
    // 设备上超级块失败或取根节点失败，则都显示信息并停机。
260         if (!(p=read_super(ROOT_DEV)))
261                 panic("Unable to mount root");
262         if (!(mi=iget(ROOT_DEV,ROOT_INO)))              // 在 fs.h 中 ROOT_INO 定义为 1。
263                 panic("Unable to read root i-node");
    // 现在我们对超级块和根 i 节点进行设置。把根 i 节点引用次数递增 3 次。因为下面 266 行上
    // 也引用了该 i 节点。另外，iget() 函数中 i 节点引用计数已被设置为 1。然后置该超级块的
    // 被安装文件系统 i 节点和被安装到 i 节点字段为该 i 节点。再设置当前进程的当前工作目录
    // 和根目录 i 节点。此时当前进程是 1 号进程（init 进程）。
264         mi->i_count += 3 ;      /* NOTE! it is logically used 4 times, not 1 */
                                    /* 注意！从逻辑上讲，它已被引用了 4 次，而不是 1 次 */
265         p->s_isup = p->s_imount = mi;
266         current->pwd = mi;
267         current->root = mi;
    // 然后我们对根文件系统上的资源作统计工作。统计该设备上空闲块数和空闲 i 节点数。首先
    // 令 i 等于超级块中表明的设备逻辑块总数。然后根据逻辑块位图中相应比特位的占用情况统
    // 计出空闲块数。这里宏函数 set_bit()只是在测试比特位，而非设置比特位。"i&8191" 用于
    // 取得 i 节点号在当前位图块中对应的比特位偏移值。"i>>13" 是将 i 除以 8192，也即除一个
    // 磁盘块包含的比特位数。
268         free=0;
269         i=p->s_nzones;
270         while (-- i >= 0)
271                 if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
272                         free++;
    // 在显示过设备上空闲逻辑块数/逻辑块总数之后。我们再统计设备上空闲 i 节点数。首先令 i
    // 等于超级块中表明的设备上 i 节点总数+1。加 1 是将 0 节点也统计进去。然后根据 i 节点位
```

```
    // 图中相应比特位的占用情况计算出空闲 i 节点数。最后再显示设备上可用空闲 i 节点数和 i
    // 节点总数。
273         printk("%d/%d free blocks\n\r",free,p->s_nzones);
274         free=0;
275         i=p->s_ninodes+1;
276         while (-- i >= 0)
277                 if (!set_bit(i&8191,p->s_imap[i>>13]->b_data))
278                         free++;
279         printk("%d/%d free inodes\n\r",free,p->s_ninodes);
280 }
281
```