8.1. EXT2 磁盘数据的组织

一个物理磁盘可以划分为多个磁盘分区，每个磁盘分区可以从逻辑上看成是从 0 开始编号的大量扇区，各自可以格式化成不同类型的文件系统（例如 EXT2、NTFS 等）。如果格式化成 EXT2 文件系统，则其内部按照 EXT2 的规范，将磁盘盘块的组织成超级块、组描述符和位图、索引节点、目录等管理数据，放在分区前端称为元数据区，剩余空间用于保存文件数据。下面来学习和分析 EXT2 内部是如何组织的细节。

8.1.1. 整体布局

格式化成 ext2 文件系统的分区上，这些盘块分成两大类，一类是存放文件内容数据的叫作数据盘块，另一类是保存元数据（管理数据）的叫作元数据盘块。但是它们的物理布局不是一刀切地简单分成两段，而是首先分成多个同尺寸块组（block group）的形式，在块组内再分成前后两部分——前面是元数据盘块后面是数据盘块。此外，一个硬盘的可引导分区的第一个盘块是为引导系统而保留的，通常称为引导块（引导扇区），启动扇区并不是所有分区都需要的，非启动分区不需要这样的块。EXT2 文件系统的基本布局如图 8-1 图~~ 8-1~~图~~ 8-1~~所示。

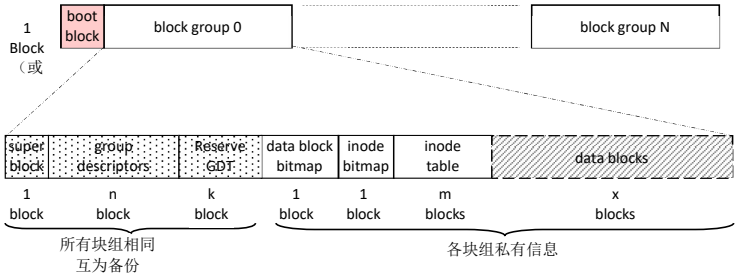

图 8-1 EXT2 文件系统布局

每个块组有很多连续盘块组成，块组内部还有自己的组织结构。每个块组中包含如下用途的盘块：

- 一个超级块（super block），各个块组中的超级块都是相同的（互为备份），记录文件系统全局性的管理信息和属性；
- 一个块组描述符表（GDT，group descripter table）每个块组中的块组描述符都是相同的（互为备份），记录各个块组的管理信息和相关属性；
- 一个数据盘块位图（data block bitmap），记录着本块组数据盘块（data blocks）的使用情况；
- 一个索引节点盘块位图（inode bitmap），记录着本块组索引节点表（inode table）的使用情况；

- 一个索引节点表（inode table），每个表项记录一个文件（普通文件或目录文件）的属性及数据块所在位置（含 12 个直接索引项、1 个间接级索引、1 个二级间接索引和 1 个三级间接索引）。
- 大量数据盘块（data blocks），记录着各种文件数据，使用情况由前面的数据盘块位图记录，因此一个块组的数据盘块数量不得超过数据盘块位图的记录能力。

其中超级块和块组描述符是很重要的，因此超级块和块组描述符是有多个备份的——出现错误时可以恢复。早期 EXT2 在每个块组中都作了备份，后续版本的 ext2 则采用稀疏超级块（sparse spuerblock）技术，超级块只在 0、1 块组，以及编号为 3/5/7 次幂的块组中，以节约超级块的备份空间。

联系操作系统理论课程中的概念，EXT2 的空闲盘块管理方式使用的是位图方式，而分配给一个文件的数据盘块，则是使用索引节点的方式进行管理。

8.1.2. 超级块

超级块是文件系统的起点信息，记录了文件系统全局性的一些信息。磁盘上的 EXT2 超级块是一个 struct ext2_super_block 结构体，共 1024 字节，参见代码 8-1 代码~~ 8-1~~代码~~ 8-1~~。其中各个字段使用了__u8、__u16 和__u32 表示不同位数的无符号数，__s8、__s16 和__s32 表示不同位数的有符号数，__le16、__le32 表示按小端（little-endian）顺序双字节和四字节数据，__be16 和__be32 表示大端（big-endian）的双字节和四字节数据。为了将磁盘字节序转换成 CPU 字节序，在 linux-3.13/include/uapi/linux/byteorder/big_endian.h 和 linux-3.13/include/uapi/linux/byteorder/ little_endian.h 中定义了相关的转换例程。

代码 8-1 ext2_super_block (linux-3.13/fs/ext2/ext2.h)

```
410 struct ext2_super_block {
411         __le32  s_inodes_count;         /* Inodes count */ 索引节点总数
412         __le32  s_blocks_count;         /* Blocks count */ 盘块的总数
413         __le32  s_r_blocks_count;       /* Reserved blocks count */ 保留的盘块数
414         __le32  s_free_blocks_count;    /* Free blocks count */      空闲盘块数
415         __le32  s_free_inodes_count;    /* Free inodes count */      空闲索引节点数
416         __le32  s_first_data_block;     /* First Data Block */  第一个数据盘块号
417         __le32  s_log_block_size;       /* Block size */    盘块大小(以2的幂次表示)
418         __le32  s_log_frag_size;        /* Fragment size */     片的大小
419         __le32  s_blocks_per_group;     /* # Blocks per group */ 块组内部的盘块数量
420         __le32  s_frags_per_group;      /* # Fragments per group */块组中的片数
421         __le32  s_inodes_per_group;     /* # Inodes per group */ 块组内的索引节点数
422         __le32  s_mtime;                /* Mount time */    最后一次安装挂载时间
423         __le32  s_wtime;                /* Write time */    最后一次写操作的时间
424         __le16  s_mnt_count;            /* Mount count */  安装挂载次数(检查后清零)
425         __le16  s_max_mnt_count;        /* Maximal mount count */最大挂载次数，超过则触发文件系统检查
426         __le16  s_magic;                /* Magic signature */   文件系统魔数（标志）
427         __le16  s_state;                /* File system state */      文件系统状态
428         __le16  s_errors;           /* Behaviour when detecting errors */出错时的行为
429         __le16  s_minor_rev_level;    /* minor revision level */ 文件系统的次版本号
430         __le32  s_lastcheck;          /* time of last check */上一次文件系统检查时间
431         __le32  s_checkinterval;      /* max. time between checks */      两次检查之间的间隔
```

```
  __le32  s_creator_os;           /* OS */  创建该文件系统的操作系统
  __le32  s_rev_level;            /* Revision level */    版本号
   __le16  s_def_resuid;    /* Default uid for reserved blocks */保留块的缺省
UID
   __le16  s_def_resgid;   /* Default gid for reserved blocks */保留块的缺省用
户组 ID
  /*
   * These fields are for EXT2_DYNAMIC_REV superblocks only.
   *
   * Note: the difference between the compatible feature set and
   * the incompatible feature set is that if there is a bit set
   * in the incompatible feature set that the kernel doesn't
   * know about, it should refuse to mount the filesystem.
   *
   * e2fsck's requirements are more strict; if it doesn't know
   * about a feature in either the compatible or incompatible
   * feature set, it must abort and not try to meddle with
   * things it doesn't understand...
   */
   __le32  s_first_ino;             /* First non-reserved inode */第一个非保留
的索引节点号
  __le16   s_inode_size;          /* size of inode structure */ 索引节点大小
   __le16  s_block_group_nr;       /* block group # of this superblock */  本
超级块的块组号
   __le32  s_feature_compat;       /* compatible feature set */具有兼容特点的
位图
   __le32  s_feature_incompat;     /* incompatible feature set */具有非兼容特
点的位图
  __le32  s_feature_ro_compat;    /* readonly-compatible feature set */只读特
点的位图
  __u8    s_uuid[16];             /* 128-bit uuid for volume */128 位的文件系
统标识符
 char    s_volume_name[16];      /* volume name */       卷名
 char    s_last_mounted[64];     /* directory where last mounted */    最  后
一个安装点路径名
  __le32  s_algorithm_usage_bitmap; /* For compression */      用于压缩
  /*
   * Performance hints.  Directory preallocation should only
   * happen if the EXT2_COMPAT_PREALLOC flag is on.
   */
   __u8    s_prealloc_blocks;      /* Nr of blocks to try to preallocate*/预分
配的盘块数
   __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */  目录的预
分配盘块数
  __u16   s_padding1;              （填充位，无具体功能）
  /*
   * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
   */                                  以下是 EXT3 日志相关的
  __u8    s_journal_uuid[16];     /* uuid of journal superblock */
  __u32   s_journal_inum;         /* inode number of journal file */
  __u32   s_journal_dev;          /* device number of journal file */
  __u32   s_last_orphan;          /* start of list of inodes to delete */
  __u32   s_hash_seed[4];         /* HTREE hash seed */
  __u8    s_def_hash_version;     /* Default hash version to use */
  __u8    s_reserved_char_pad;
  __u16   s_reserved_word_pad;
  __le32  s_default_mount_opts;
  __le32  s_first_meta_bg;        /* First metablock block group */
  __u32   s_reserved[190];        /* Padding to the end of the block */
```

```
480 };
```

保留的盘块可以让超级用户在磁盘空间紧缺时仍有可用的盘块。文件系统状态 s_state 字段为 0 表示已经安装挂载或者未安全地卸载，为 1 表示正常卸载，为 2 则表示包含错误。

8.1.3. 块组描述符

一个 EXT2 格式的磁盘文件系统内含多个块组，每个块组都有自己的描述符，对应结构体 ext2_group_desc，具体参见代码 8-2 代码 8-2 代码 8-2。

代码 8-2 ext2_group_desc (linux-3.13/fs/ext2/ext2.h)

```
194 struct ext2_group_desc
195 {
196         __le32  bg_block_bitmap;    /* Blocks bitmap block */数据盘块位图所在的盘块号
197         __le32  bg_inode_bitmap;    /* Inodes bitmap block */索引节点位图所在的盘块号
198         __le32  bg_inode_table; /* Inodes table block */索引节点表的起点所在的盘块号
199         __le16  bg_free_blocks_count;   /* Free blocks count */      空闲盘块数
200         __le16  bg_free_inodes_count;   /* Free inodes count */      空闲索引节点数
201         __le16  bg_used_dirs_count;     /* Directories count */      在用目录个数
202         __le16  bg_pad;                     字节对齐的填充
203         __le32  bg_reserved[3];             保留（使得结构体整体占用 32 字节）
204 };
```

当分配索引节点和数据盘块时，涉及到空闲盘块计数值 bg_free_block_count、空闲索引节点计数值 bg_free_inodes_count 和在用目录计数值 bg_used_dirs_count 字段。

数据盘块位图的块号在 bg_block_bitmap 指出，索引节点位图盘块号由 bg_inode_bitmap 指出，索引节点起点块号由 bg_inode_table 指出。

8.1.4. 索引节点

作为磁盘文件系统，ext2 只有静态的文件概念，也就是索引节点所代表的文件，而没有 VFS 中的 struct file 的动态打开的文件概念。文件读写操作先经过一个地址空间 address_space 的中间层，此时的逻辑文件是具有简单的线性的编址空间。但是将逻辑文件映射到磁盘上时，就必须由 ext2 索引节点提供必要的信息——用于从线性的逻辑文件编址空间转换到离散的盘块号的映射，最终才能落实到盘块的读写。ext2 的盘上索引节点是 struct ext2_inode，主要记录数据盘块的位置（VFS 的 inode 没有数据盘块位置信息）和访问权限等信息，其数据结构如代码 8-3 代码 8-3 代码 8-3 所示。

代码 8-3 ext2_inode (Linux-3.13/fs/ext2/ext2.h)

```
297 struct ext2_inode {
298         __le16  i_mode;         /* File mode */   文件类型和访问权限
299         __le16  i_uid;          /* Low 16 bits of Owner Uid */  所有者的UID(低16位)
300         __le32  i_size;         /* Size in bytes */             文件长度（字节）
301         __le32  i_atime;        /* Access time */               访问时间戳
302         __le32  i_ctime;        /* Creation time */             创建时间戳
303         __le32  i_mtime;        /* Modification time */         修改时间戳
304         __le32  i_dtime;        /* Deletion Time */             删除时间戳
305         __le16  i_gid;          /* Low 16 bits of Group Id */   所有者 GID（低 16 位）
306         __le16  i_links_count;  /* Links count */               硬链接计数
307         __le32  i_blocks;       /* Blocks count */              文件长度(block 计数)
```

```
308         __le32  i_flags;        /* File flags */                文件标志
309         union {
310                 struct {
311                         __le32  l_i_reserved1;
312                 } linux1;                                       Linux 中特定的信息之 1
313                 struct {
314                         __le32  h_i_translator;
315                 } hurd1;
316                 struct {
317                         __le32  m_i_reserved1;
318                 } masix1;
319         } osd1;                         /* OS dependent 1 */
320         __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */ 数据盘块指针数组
321         __le32  i_generation;   /* File version (for NFS) */
322         __le32  i_file_acl;     /* File ACL */                  文件访问控制列表
323         __le32  i_dir_acl;      /* Directory ACL */             目录访问控制列表
324         __le32  i_faddr;        /* Fragment address */
325         union {
326                 struct {
327                         __u8    l_i_frag;       /* Fragment number */
328                         __u8    l_i_fsize;      /* Fragment size */
329                         __u16   i_pad1;
330                         __le16  l_i_uid_high;   /* these 2 fields    */
331                         __le16  l_i_gid_high;   /* were reserved2[0] */
332                         __u32   l_i_reserved2;
333                 } linux2;                                       Linux 中特定的信息之 2
334                 struct {
335                         __u8    h_i_frag;       /* Fragment number */
336                         __u8    h_i_fsize;      /* Fragment size */
337                         __le16  h_i_mode_high;
338                         __le16  h_i_uid_high;
339                         __le16  h_i_gid_high;
340                         __le32  h_i_author;
341                 } hurd2;
342                 struct {
343                         __u8    m_i_frag;       /* Fragment number */
344                         __u8    m_i_fsize;      /* Fragment size */
345                         __u16   m_pad1;
346                         __u32   m_i_reserved2[2];
347                 } masix2;
348         } osd2;                         /* OS dependent 2 */
349 };
```

一个块组中的大量 ext2 的索引节点在磁盘上构成 inode table，表中每个元素（ext2 索引节点）代表一个文件，每个索引节点内含 i_block[]数组用于记录文件的数据块所在位置，i_block[EXT2_N_BLOCKS]虽然称为数组，但是它的元素属性并不完全一样，其具体含义如下：

1） 前 12 项 i_block[0]~i_block[11]用于记录文件数据盘块的编号；

2） 后面的 i_block[12]所记录的盘块并不用于存放文件数据，而是存放的一个间接索引表，表中的每一个项才是文件数据盘块的编号。因此第 i_block[12]指向间接索引块。

3） 在后面的 i_block[13]所记录的盘块则用作二次间接索引，也就是说由 i_block[13]所指向的盘块并不能直接用作索引，而是需要做两次间接索引才能获得数据盘块；

4） 最后一项 i_block[14]记录的盘块则是用作三次间接索引。

图 8-2 ~~图 8-2~~ 图 8-2 展示了 inode table、i_block[]和文件数据盘块等关系，图中标出了直接索引（折线箭头）、一次间接索引（实弧线箭头）、二次间接索引（虚弧线箭头）和三次间

接索引（实弧线箭头）的示意。i_block[12]指向一个一次间接块，将该盘块读出可以当作一个数据盘块号数组。i_block[13]指向一个二次间接块，二次间接块中的每一个项（盘块号）所指向的盘块不是数据盘块，而是再次作为间接索引使用（相当于一个间接盘块）。

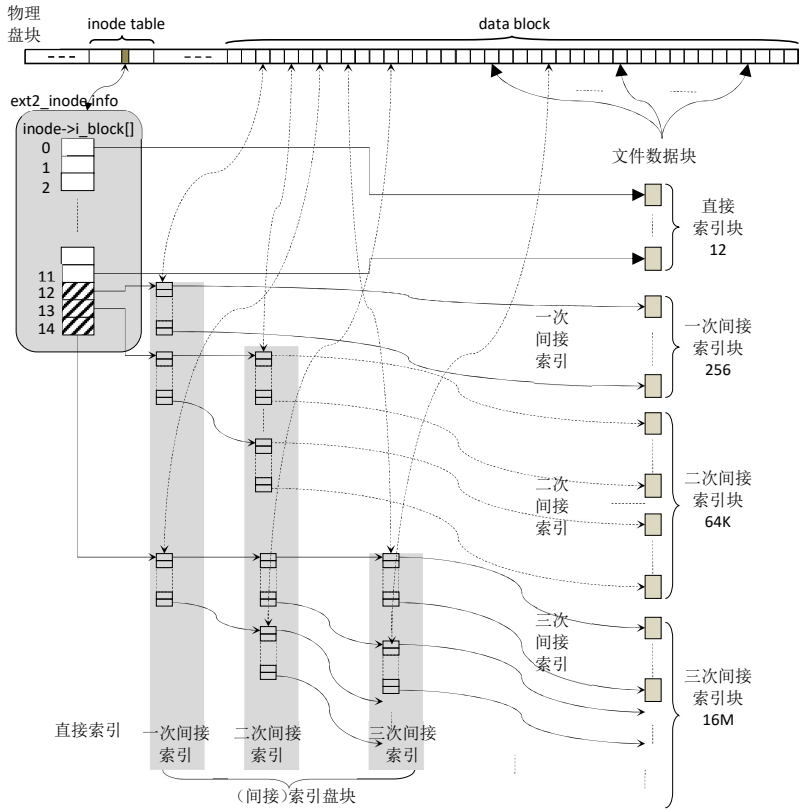

图 8-2 ext2 数据块索引示意图

如果盘块大小为 1KB（相当于 2 个 512B 的物理扇区），那么一个盘块就可以存入 256 个盘块号（32 位）。当文件不超过 12KB 时，该文件的所有盘块可以由 i_block[0]~i_block[11]记录。当文件大于 12KB 时就需要用 i_block[12]作间接索引，由 i_block[12]所指向的盘块可以切割成 256 个盘块号，因此可以索引 256 个盘块共 256KB。因此当文件大小在 12KB~（12+256）KB 的范围时，前 12KB 由直接索引寻址，后面 256KB 需要通过间接索引来确定物理盘块号。如果文件继续增大，就需要用到二次间接索引，此时新增加的二级索引容量为 256*256*1KB=64KKB=64MB，所以当文件大小在（12+256）KB~（12+256+64K）KB 的时候，数据块分成 12KB 的直接索引部分，256KB 的一次间接索引部分和 64MB 的二次索引部分。文件继续增大就需要使用三次间接索引，此时文件最大容量为（12+256+64K+16M）KB。上述例子见图 8-2 ~~图 8-2~~ 图 8-2 右侧数字，请注意图中的索引盘块虽然存储了管理数据，但它们与普通数据盘块同样处于图 8-1 ~~图 8-1~~ 图 8-1 中的 data blocks 数据盘块区。

当一个三次间接的文件尺寸大于 32 位系统的指针所能访问的最大范围时，需要引入专门方案来访问大文件，这是不仅影响标准库例程、也影响到内核代码。

读者可以看到索引节点不仅包含了文件的各种管理信息，还记录了文件数据所在的盘块号，这就是为什么在前面我们说索引节点才是真正意义上代表文件，而不是文件对象 file 和目录项对象 dentry 的原因。

8.1.5. 目录结构

目录用于将整个文件系统的文件形成按路径名访问的树形组织结构。ext2 文件系统并没有单独形成用于目录数据的盘块区，而是将目录与普通文件一样存放在 data blocks 区域的盘块。每一个目录都作为一个文件，所以也是使用盘上索引节点管理其内部的数据盘块——只不过其内容是用于存放所（包含的文件或目录的）目录项，通过上一级目录项将其类型标识为“目录”以区分普通文件。根目录有固定的起点（例如 ext2 文件系统跟目录的索引号位 2），因此根目录文件就不需要上级目录来定位了。

目录在存储管理上和普通文件相同——也是通过索引节点来管理目录项数据所在的盘块的，只不过盘块数据内容含有多条目录项数据结构，每个目录项使用 ext2_dir_entry 结构体来描述，ext2_dir_entry 和 ext2_dir_entry_2 如代码 8-4 代码 8-4 代码 8-4。

代码 8-4 ext2_dir_entry 和 ext2_dir_entry_2 (linux-3.13 /fs/ext2/ext2.h)

```
578 struct ext2_dir_entry {
579         __le32  inode;                  /* Inode number */ 目录项对应的索引节点
580         __le16  rec_len;                /* Directory entry length */ 目录项长度
581         __le16  name_len;               /* Name length */       名字长度
582         char    name[];                 /* File name, up to EXT2_NAME_LEN */
583 };
…
591 struct ext2_dir_entry_2 {
592         __le32  inode;                  /* Inode number */
593         __le16  rec_len;                /* Directory entry length */
594         __u8    name_len;               /* Name length */
595         __u8    file_type;               文件类型（见下面 603 行）
596         char    name[];                 /* File name, up to EXT2_NAME_LEN */
597 };
…
603 enum {           ext2_dir_entry_2 中的 file_type 枚举类型
604         EXT2_FT_UNKNOWN         = 0,
605         EXT2_FT_REG_FILE        = 1,          普通文件
606         EXT2_FT_DIR             = 2,          目录文件
607         EXT2_FT_CHRDEV          = 3,          字符设备文件
608         EXT2_FT_BLKDEV          = 4,          块设备文件
609         EXT2_FT_FIFO            = 5,          命名管道 FIFO
610         EXT2_FT_SOCK            = 6,          SOCK
611         EXT2_FT_SYMLINK         = 7,          符号链接
612         EXT2_FT_MAX
613 };
```

ext2_dir_entry 和 ext2_dir_entry_2 的差异在于，老版本中 ext2_dir_entry 中的 name_len 为无符号短整数，而新版中则改为 8 位的无符号字符，腾出一半用作文件类型。目录项的类型包括：普通文件、目录、字符设备、块设备、FIFO、SOCK、符号链接以及未知类型。文件类型不是定义在 inode 中的，而是在目录项中（见代码 8-4 代码 8-4 代码 8-4 的 603 行）。字符设

备、块设备和 FIFO 文件等类型的文件，不需要在磁盘上占用空间，即它们不需要数据盘块来存放文件内容——只需要目录项和相应的索引节点即可。

在 inode 成员中存放该目录项对应的文件 inode 号（可以有多个目录项通过硬链接指向同一个磁盘文件的 inode）。由于文件名是不定长的，所以目录项长度也不定——使用 rec_len 来记录。文件名的长度由 name_len 指出。文件名存在 name[]数组中，长度不超过 EXT2_NAME_LEN（即 255）个字符。目录项的长度是以 4 的整数倍字节递增的，图 8-3 图 8-3 图 8-3 是一个目录文件所含的目录项的示意图，我们以根目录为例。

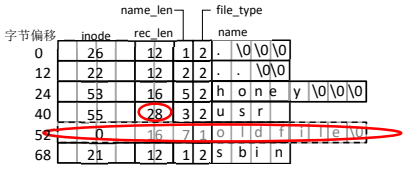

图 8-3 ext2 目录文件中的目录项

例如想查看/honey 目录里面有什么子目录和文件，就需要将 53 号 inode 对应的文件数据内容按照目录项结构进行解析，从而得出/honey 目录文件中的所有目录项（可能是子目录，也可能是普通文件或特殊文件等）。

对于软链接而言，如果目标文件路径名较短，则直接存储在索引节点的 i_blocks[]数组中（共有 15x4=60 字节）。否则将软连接的路径名字符串存储在文件数据盘块中。

8.2. EXT2 文件系统的创建

出于方便操作的原因，本节将使用环回设备从而不需要使用 U 盘或硬盘设备。也就是说我们将一个普通文件通过环回设备接口，模拟成一个磁盘块设备。

8.2.1. 分配磁盘空间

首先，创建一个大小为 512MB 的文件，后面再将它变成环回（loopback）设备并在其上创建文件系统。

屏显 8-1 用 dd 命令创建 512MB 大小的文件

```
[root@localhost exp6-filesystem]# dd if=/dev/zero of=bean bs=1k count=512000
记录了 512000+0 的读入
记录了 512000+0 的写出
524288000 字节(524 MB)已复制，1.15052 秒，456 MB/秒
```

通过 ll 命令检查文件大小，确保它分配了足够的空间。

```
[root@localhost exp6-filesystem]# ll bean
-rw-r--r--. 1 root root 524288000 3 月   6 16:03 bean
```

或者用-h 参数来执行 ll 命令，其中-h 是自适应显示数据大小的单位（K/M/G）。

```
[root@localhost exp6-filesystem]# ll -h bean
-rw-r--r--. 1 root root 500M 3 月   6 16:03 bean
```

8.2.2. 创建环回设备

通过 losetup 将制定文件设置为环回设备，从而满足我们创建文件系统需要“块设备”的基本需求。

```
[root@localhost exp6-filesystem]# losetup /dev/loop0 bean
```

然后通过/proc/partitions 检查这个环回设备是否创建成功：

```
[root@localhost exp6-filesystem]# cat /proc/partitions
major minor  #blocks  name

  11        0      57978 sr0
   8        0   16777216 sda
   8        1    1048576 sda1
   8        2   15727616 sda2
 253        0   14045184 dm-0
 253        1    1679360 dm-1
   7        0     512000 loop0
```

8.2.3. 创建 EXT2 文件系统

然后使用 mke2fs 命令在该设备上建立 EXT2 文件系统。

屏显 8-2 mke2fs 命令的输出

```
[root@localhost exp6-filesystem]# mke2fs /dev/loop0
mke2fs 1.42.9 (28-Dec-2013)
Discarding device blocks: 完成
文件系统标签=
OS type: Linux
块大小=1024 (log=0)
分块大小=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
128016 inodes, 512000 blocks
25600 blocks (5.00%) reserved for the super user
第一个数据块=1
Maximum filesystem blocks=67633152
63 block groups
8192 blocks per group, 8192 fragments per group
2032 inodes per group
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729, 204801, 221185, 401409
Allocating group tables: 完成
正在写入 inode 表: 完成
Writing superblocks and filesystem accounting information: 完成
```

8.2.4. 挂载文件系统

首先创建挂载点（目录，该目录会被挂载对象所覆盖）

```
[root@localhost exp6-filesystem]# mkdir /mnt/bean
```

然后使用 mount 命令挂载/dev/loop0 设备（该设备上已经创建了 EXT2 文件系统）

```
[root@localhost exp6-filesystem]# mount -t ext2 /dev/loop0 /mnt/bean
```

此时，可以通过不带参数的 mount 命令查看挂载结果：

```
[root@localhost exp6-filesystem]# mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime,seclabel)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
……
/dev/loop0 on /mnt/bean type ext2 (rw,relatime,seclabel)
```

正确挂载后，可以正常时用了。我们如果在该挂载点目录下执行 ll 命令可以看到一个空的目录。

```
[root@localhost exp6-filesystem]# cd /mnt/bean/
[root@localhost bean]# ll
总用量 12
drwx------. 2 root root 12288 3 月   6 16:05 lost+found
```

8.3. 查看 EXT2 磁盘数据

在着手查看文件系统的元数据细节前，请读者回顾一下图 8-1 图 8-1 图 8-1 关于 EXT2 文件系统整体布局的示意图。

8.3.1. 布局信息

在 mke2fs 时，将会告知把超级块放到那些块组，例如屏显 8-2 屏显 8-2 屏显 8-2 里面就给出了备份用的超级块所在位置的相应信息：

```
Superblock backups stored on blocks:
8193, 24577, 40961, 57345, 73729, 204801, 221185, 401409
```

对于屏显 8-2 屏显 8-2 屏显 8-2 的块组是 8192，因此上面的块号对应于块组 0/1/3/5/7/9/25/27/49 的第一个块（超级块的备份位置是根据块组号为 3/5/7 的幂来决定的）。

下面我们用 dumpe2fs 命令来查看前面创建的 EXT2 文件系统的超级块的信息：

屏显 8-3 超级块和块组描述符信息

```
[root@localhost bean]# dumpe2fs /dev/loop0
dumpe2fs 1.42.9 (28-Dec-2013)
Filesystem volume name:   <none>
Last mounted on:          <not available>
Filesystem UUID:          c35a602b-b89c-4d51-870c-46f33d335658
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      ext_attr resize_inode dir_index filetype sparse_super
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         not clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              128016
Block count:              512000
Reserved block count:     25600
Free blocks:              493526
Free inodes:              128005
First block:              1
Block size:               1024
Fragment size:            1024
Reserved GDT blocks:      256
Blocks per group:         8192
Fragments per group:      8192
Inodes per group:         2032
Inode blocks per group:   254
Filesystem created:       Mon Mar  6 16:05:41 2017
Last mount time:          Mon Mar  6 16:06:34 2017
Last write time:          Mon Mar  6 16:06:34 2017
Mount count:              1
Maximum mount count:      -1
Last checked:             Mon Mar  6 16:05:41 2017
Check interval:           0 (<none>)
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               128
Default directory hash:   half_md4
Directory Hash Seed:      ed77b8db-bb3a-40ce-9152-25c0ca2c3e92
```

```
Group 0: (Blocks 1-8192)
  主 superblock at 1, Group descriptors at 2-3
  保留的 GDT 块位于 4-259
  Block bitmap at 260 (+259), Inode bitmap at 261 (+260)
  Inode 表位于 262-515 (+261)
  7663 free blocks, 2021 free inodes, 2 directories
  可用块数: 530-8192
  可用 inode 数: 12-2032
Group 1: (Blocks 8193-16384)
  备份 superblock at 8193, Group descriptors at 8194-8195
  保留的 GDT 块位于 8196-8451
  Block bitmap at 8452 (+259), Inode bitmap at 8453 (+260)
  Inode 表位于 8454-8707 (+261)
  7677 free blocks, 2032 free inodes, 0 directories
  可用块数: 8708-16384
  可用 inode 数: 2033-4064
Group 2: (Blocks 16385-24576)
  Block bitmap at 16385 (+0), Inode bitmap at 16386 (+1)
  Inode 表位于 16387-16640 (+2)
  7936 free blocks, 2032 free inodes, 0 directories
  可用块数: 16641-24576
  可用 inode 数: 4065-6096
Group 3: (Blocks 24577-32768)
  备份 superblock at 24577, Group descriptors at 24578-24579
  保留的 GDT 块位于 24580-24835
  Block bitmap at 24836 (+259), Inode bitmap at 24837 (+260)
  Inode 表位于 24838-25091 (+261)
  7677 free blocks, 2032 free inodes, 0 directories
  可用块数: 25092-32768
  可用 inode 数: 6097-8128

......

Group 5: (Blocks 40961-49152)
  备份 superblock at 40961, Group descriptors at 40962-40963
  保留的 GDT 块位于 40964-41219
  Block bitmap at 41220 (+259), Inode bitmap at 41221 (+260)
  Inode 表位于 41222-41475 (+261)
  7677 free blocks, 2032 free inodes, 0 directories
  可用块数: 41476-49152
  可用 inode 数: 10161-12192

......

Group 62: (Blocks 507905-511999)
  Block bitmap at 507905 (+0), Inode bitmap at 507906 (+1)
  Inode 表位于 507907-508160 (+2)
  3839 free blocks, 2032 free inodes, 0 directories
  可用块数: 508161-511999
  可用 inode 数: 125985-128016
```

上面的屏显 8-3 屏显 8-3 屏显 8-3 是 dumpe2fs 工具给我们输出的信息，如果读者想进一步验证这些信息是否正确，那么我们可以直接从/dev/loop0 设备读入超级块然后按照代码 8-1 代码 8-1 代码 8-1 来解读相关信息。

```
[root@localhost bean]# dd if=/dev/loop0 bs=1k count=261 |od -tx1 -Ax > /tmp/dump_sp_hex
记录了 261+0 的读入
记录了 261+0 的写出
```

267264 字节(267 kB)已复制，0.0449364 秒，5.9 MB/秒

然后用 cat /tmp/dump_sp_hex 或者 vi /tmp/dump_sp_hex 命令查看，可以获得屏显 8-4 屏显 8-4 屏显 8-4 所示的输出。

屏显 8-4 超级块的信息

```
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00          0KB 偏移处对应“块 0”
*
000400 10 f4 01 00 00 d0 07 00 00 64 00 00 d6 87 07 00          1KB 偏移处对应“块 1”
000410 05 f4 01 00 01 00 00 00 00 00 00 00 00 00 00 00
000420 00 20 00 00 00 20 00 00 f0 07 00 00 8a 18 bd 58
000430 8a 18 bd 58 01 00 ff ff 53 ef 00 00 01 00 00 00
000440 55 18 bd 58 00 00 00 00 00 00 00 00 01 00 00 00
000450 00 00 00 00 0b 00 00 00 80 00 00 00 38 00 00 00
000460 02 00 00 00 01 00 00 00 c3 5a 60 2b b8 9c 4d 51
000470 87 0c 46 f3 3d 33 56 58 00 00 00 00 00 00 00 00
000480 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
0004c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
0004d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0004e0 00 00 00 00 00 00 00 00 00 00 00 00 ed 77 b8 db
0004f0 bb 3a 40 ce 91 52 25 c0 ca 2c 3e 92 01 00 00 00
000500 0c 00 00 00 00 00 00 00 55 18 bd 58 00 00 00 00
000510 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
000560 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000570 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
000800 04 01 00 00 05 01 00 00 06 01 00 00 ef 1d e5 07          2KB 偏移处对应“块 2”
000810 02 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00
000820 04 21 00 00 05 21 00 00 06 21 00 00 fd 1d f0 07
……
```

最左边一列是 16 进制地址，000400=1K 就对应是文件第 1K 个字节，000800 =2K 对应文件的第 2K 个字节。超级块前面有启动块，因此超级块从 000400 字节偏移处开始。

对照代码 8-1 代码 8-1 代码 8-1，第一个字段叫 s_inodes_count， 占四个字节。屏显 8-4 屏显 8-4 屏显 8-4 从 1K 开始前四个字节是 10 f4 01 00。EXT2 设计者为了支持文件系统的可移动，规定磁盘上一律是 little-endian（数据读入内存中时，kernel 来负责把格式转成 cpu 的本机格式），于是对应的数值是 0x0001f410。对照屏显 8-3 屏显 8-3 屏显 8-3，0x0001f410=128016，两者相符。再来看看另一个字段 free_blocks_count，起始位置是超级块的第 12 字节，即 00040c 地址。屏显 8-4 屏显 8-4 屏显 2-3 的 0x0004120c 字节偏移处是 d6 87 07 00， 0x000787d6 = 493526，和 dumpe2fs 的输出屏显 8-3 屏显 8-3 显 2-2 的 Free blocks 给出的一样。其他的字段可以由读者自己查看。

由于超级块只占 1 个 block 块，当 blocksize 为 4K 的时候，这个块大多数空间是浪费的（为 0）。

设置了格式：字体颜色：红色

设置了格式：字体颜色：红色

设置了格式：字体颜色：红色

设置了格式：字体颜色：红色

8.3.2. 块组描述符

屏显 8-3 屏显 8-3 屏显 8-3 显示块组 0 和块组 1 都有块组描述符，其中块组 0 显示“Group descriptors at 2-3”而块组 1 显示“Group descriptors at 8194-8195”。我们可以查看一下块组 0（Group 0）里面第 2 和第 3 个块存储的块组描述符，也就说从 0x000800～0x001000 的内容。

用 cat /tmp/dump_sp_hex 或者 vi /tmp/dump_sp_hex 命令查看，获得所示的内容（衔接屏显 8-4 屏显~~8-4~~屏显~~8-4~~的结尾处）

屏显 8-5 块组描述符信息

```
000800 04 01 00 00 05 01 00 00 06 01 00 00 ef 1d e5 07    --此处开始时块组 0 的描述符
000810 02 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00
000820 04 21 00 00 05 21 00 00 06 21 00 00 fd 1d f0 07    --此处开始时块组 1 的描述符
000830 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00
000840 01 40 00 00 02 40 00 00 03 40 00 00 00 1f f0 07    --此处开始时块组 2 的描述符
000850 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00
000860 04 61 00 00 05 61 00 00 06 61 00 00 fd 1d f0 07    ……
000870 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00
000880 01 80 00 00 02 80 00 00 03 80 00 00 00 1f f0 07
000890 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00
0008a0 04 a1 00 00 05 a1 00 00 06 a1 00 00 fd 1d f0 07
0008b0 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00
0008c0 01 c0 00 00 02 c0 00 00 03 c0 00 00 00 1f f0 07
0008d0 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00
0008e0 04 e1 00 00 05 e1 00 00 06 e1 00 00 fd 1d f0 07
```

对照代码 8-1 代码~~8-1~~代码~~8-1~~，第一个字段 bg_block_bitmap 纪录了数据盘块位图所在的盘块号，04 01 00 00 转换成可读的十进制是 0x104=259，表示数据块位图位于第 259 块。第二个字段 bg_inode_bitmap 记录了索引节点位图所在的盘块号，05 01 00 00 转换成刻度的数值是 0x106=260，表示索引节点位图位于第 260 块。第四个字段 bg_free_blocks_count 记录了空闲数据盘块数量，0x1def=7663 表示有 7663 个空闲数据块。上述信息和 dumpe2fs 输出得出来的信息是一致的，其他字段请读者自行检验。

块组描述符以数组的形式存储在 K 个快上，对于我们的/dev/loop0 只有 63 个块组，每个组块需要 32 个字节（共 2016 字节），只需要 2 个 1KB 的块的空间就足够存储它们了。由于块组描述符和超级块一样是由多个备份的，也就是说其他存储组描述符的两个 block，信息和块组 0 中的组描述符的两个 block 是一样的，读者可以自行读入其他块组上的块组描述符以检验。

数据块的位图只占用 1 个块，一个块有 1024 字节，每字节 8 个 bit，共有 8192 个 bit，可记录最多 8192 个块。

8.3.3. 索引节点与文件内容

我们首先来学习如何在“文件的索引节点号已知”的情况下，将文件内容读入。

索引节点的定位

在/dev/loop0 设备上根据索引节点号来定位索引节点需要分两个步骤，首先确定该索引节点属于哪个块组，然后在该块组内的索引节点表中找到所需的索引节点内容。

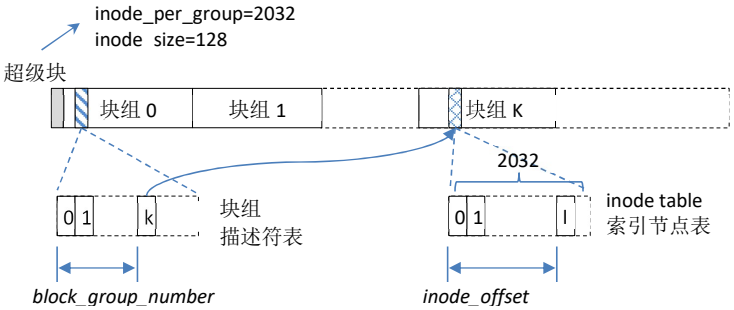

图 8-4 索引节点定位过程

为了确定文件索引节点的起点位置就有先知道索引节点所在的块组。那么根据索引节点号要如何知道索引节点所在的块组呢，这里还需要一个文件系统的信息，就是代码 8-1 代码 8-1 代码 8-1 的 421 行 s_inodes_per_group，利用这个信息根据公式 8-1 公式 8-1 公式 8-1 就可以知道索引节点所在块组编号（索引节点从 1 开始编号，块组从 0 开始编号）。

$$block\_group\_number = (inode\_number-1) / inodes\_per\_group$$

公式 8-1

其中 *inode_number* 是目标 inode 索引节点号，*inodes_per_group* 是每个块组所拥有的最大 inode 数量。

在确定了块组号以后，就可以找到该块组上的索引节点表 inode table 的起点位置，但是表内的偏移仍需进一步计算确定。计算文件索引节点在本块组内部索引节点表中的偏移可以使用公式来计算。

$$inode\_offset = (inode\_number-1) \% inodes\_per\_group$$

公式 8-2

其中，*inode_offset* 是 inode 索引节点在本块组内部 inode table 中的偏移量（以 inode 为单位计，不是按字节计），这个 offset 就相当于数组（此处即为 inode table）中的下标，范围是从零到最大索引节点数量减一）。

结合起点和偏移量可以得到索引节点在磁盘上的绝对位置（按字节计算）：索引节点所在位置=索引节点表的起点位置+内部偏移量，如公式 8-3 公式 8-3 公式 8-3 所示。

$$inode\_location\_in\_byte = inode\_talbe\_location + inode\_offset * inode\_size$$
$$= inode\_talbe\_location + ((inode\_number-1) \% inodes\_per\_group) * inode\_size$$

公式 8-3

其中文件索引节点所在的块组的索引节点表的起点位置，这个起点位置被记录在该块组的组描述符中，代码 8-2 代码 8-2 代码 8-2 的 198 行 bg_inode_table。索引节点大小 inode_size 信息被存放在超级块中，代码 8-1 代码 8-1 代码 8-1 的 450 行 s_inode_size。

下面我们来查看文件的索引节点号，这个可以简单通过命令 ls -i 获得，如屏显 8-6 屏显 8-6 屏显 8-6 所示。

屏显 8-6 查看文件的索引节点号

```
[root@localhost bean]# ls -i
12 temp.txt
```

在文件名前的数字就是文件的索引节点号，下面我们来展示如何通过这个索引节点号 12 来定位该索引节点在磁盘上的位置。

我们前面已经用 dumpe2fs 命令获得公式 8-1 公式 8-1 公式 8-1、公式 8-2 公式 8-2 公式 8-2 和公式 8-3 公式 8-3 公式 8-3 所需的信息，如屏显 8-3 屏显 8-3 屏显 8-3 所示。此时，*inodes_per_group*=2032，*inode_size*=128，然后结合文件 temp.txt 的索引节点号 *inode_number*=12。

首先根据公式 8-1 公式 8-1 公式 8-1 计算出索引节点 12 在块组 0，然后从屏显 8-3 屏显 8-3 屏显 8-3，我们也知道了块组 0 的 inode table 位于 262-515 (+261)，即按字节计的起点位置 *inode_talbe_location* = 262*1024=0x41800。在根据公式 8-3 公式 8-3 公式 8-3 就可以准确计算出文件 temp.txt 索引节点的在磁盘中的位置（以字节计），即 262*1024+（（12-1）%2032）*128 = 0x41D8000x41d80。

虽然 dumpe2fs 工具帮助我们提取了有用的信息，但我们还是尝试用直接读取裸设备的方式获得上面全部信息——通过 dd 工具直接读取。如屏显 8-4 屏显 8-4 屏显 8-4，为了获得代码 8-1 代码 8-1 代码 8-1 的 421 行 s_inodes_per_group，我们要知道 s_inodes_per_group 在超级块中的字节偏移量，分析超级块结构体后可以知道 s_inodes_per_group 在超级块中的偏移量为 40 字节。根据屏显 8-4 屏显 8-4 屏显 8-4，超级块的开始地址为 0x400，则 s_inodes_per_group 的位置就是 0x400+0x28（40）=0x428，具体数据为“f0 07 00 00”，按小端读写即为 0x70f=2032，与 dumpe2fs 输出的信息一致。其他信息也是同理获得。

接下来，就开始读取文件 temp.txt 的索引节点信息。前面我们已经计算出了文件 temp.txt 的索引节点位置为 0x41D8000x41d80。可用 dd if=/dev/loop0 bs=1k count=2042 |od -tx1 -Ax > /tmp/dump_sp_hex0 命令读入相应的数据如屏显 8-13 屏显 8-13 屏显 8-13。

屏显 8-7 文件/tmp/dump_sp_hex0 部分内容（索引节点表）

```
041d80 a4 81 00 00 00 60 00 00 ea e8 c9 58 ea e8 c9 58
041d90 ea e8 c9 58 00 00 00 00 00 00 01 00 32 00 00 00
041da0 00 00 00 00 01 00 00 00 01 04 00 00 02 04 00 00
041db0 03 04 00 00 04 04 00 00 05 04 00 00 06 04 00 00
041dc0 07 04 00 00 08 04 00 00 09 04 00 00 0a 04 00 00
041dd0 0b 04 00 00 0c 04 00 00 12 02 00 00 00 00 00 00
041de0 00 00 00 00 f9 33 b3 16 00 00 00 00 00 00 00 00
041df0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

还需要做一个简单验证 0x41D80 是文件 temp.txt 的索引节点的起始位置。我们通过验证文件大小来侧面验证。文件的大小记录在索引节点，具体是代码 8-3 代码 8-3 代码 8-3 的 300 行 i_size，由 indoe 的结构体可以 i_size 在结构体的偏移量为 4 字节，即如果索引节点的开始地址是 0x41D80，则 i_size 的位置就为 0x41D80+0x4 = 0x41D84，从屏显 8-7 屏显 8-7 屏显 8-7 可以获得具体数据为 00 60 00 00，按小端规则解读得 0x600=24576，与 ls 命令查看到的文件长度一致，如屏显 8-11 屏显 8-11 屏显 8-11 所示。

文件数据的定位

在正确读入 temp.txt 文件的取索引节点内容后，现在可以根据索引节点中的 i_block[]数组查找数据块在/dev/loop0 设备上的位置——即利用代码 8-3 代码 8-3 代码 8-3 的 320 行 i_block[EXT2_N_BLOCKS]信息来确定文件数据所在的盘块号。

出于简洁的原因，本节只讨论计算直接索引和一次间接索引的方法。参考图 8-2 图 8-2 图 8-2 可以知道 i_block 数组 0-11 记录的是直接索引的数据盘块号，i_block[12]记录的是一次间接索引的数据盘块。二次间接索引和三次间接索引的原理相似，读者可以自行推导。

对于直接索引的定位方式，i_block[i]*1024 即为数据盘块在/dev/loop0 设备上的物理地址（按字节计）。间接索引的定位方式中， i_block[12]*1024 存放着一次间接数据盘块号，该盘块内部最多可以存放 256 个数据盘块号（即一次间接索引）。所以建立一个数组 i_block_two[256]来存放这 256 个数据盘块号。在获得这 256 个数据盘块号时，要注意以小端读取的方式读取数据。通过这个 i_block_two[256]后，就可以计算出真正的数据存放位置，即 i_block_two[i]*1024。

接下来开始以文件 temp.txt 为例，首先计算 i_block[]的开始地址，因为 i_block[]首地址在结构体中的偏移量为 40 字节，所以 i_block[]首地址在 0x41d80+0x28（40）= 0x41da8。根据屏显 8-7 屏显 8-7 屏显 8-7 可知 i_block[0]的地址为 0x041da8，数值为 0x401，这对应着 temp.txt 的首 1024 字节所在的盘块号。继续检查 i_block[1]~i_block[11]记录的盘块，发现它们是连续的——盘块 0x402~0x40c，也就是说该文件的前 12 个盘块是连续存放的。用 dd 命令或 C 语言程序读入/dev/loop0 对应 12 个盘块的内容，对应 0x401*0x400=0x100400 起始位置，内容如屏显 8-8 屏显 8-8 屏显 8-8 所示。

屏显 8-8 temp.txt 直接索引数据块（前 12 个盘块）

```
100400 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
*
100800 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62
*
100c00 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
*
101000 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64
*
101400 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65
*
101800 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66
*
101c00 67 67 67 67 67 67 67 67 67 67 67 67 67 67 67 67
*
102000 68 68 68 68 68 68 68 68 68 68 68 68 68 68 68 68
*
102400 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69 69
*
102800 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a 6a
*
102c00 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b
*
103000 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c
```

原始文件 temp.txt 存放的是 a~x（a 的 ascii 码是 0x61），每种字母重复 1024 个顺序排列。然后留意直接索引的最后一个数据是 0x6c（对应字母表第 12 个字母 l）。根据屏显 8-7 屏显 8-7 屏显 8-7 可以获得 i_block[12]的地址为 0x41dd8，数值为 0x00000212，即一次间接索引位于 0x212*0x400=0x84800 位置。如入该盘块，内容如屏显 8-9 屏显 8-9 屏显 8-9 所示。

屏显 8-9 一次间接索引所在的盘块内容（内部为 256 个盘块号）

```
084800 0d 04 00 00 0e 04 00 00 0f 04 00 00 10 04 00 00
084810 31 02 00 00 32 02 00 00 33 02 00 00 34 02 00 00
084820 35 02 00 00 36 02 00 00 37 02 00 00 38 02 00 00
084830 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

里面记录的才是实际存放文件数据的盘块号，每 4 字节记录一个盘块号，所以看到第一个存放的盘块号 0x0000040d，对应 0x40d*0x400=0x103400 位置。读入对应盘块内容，如屏显 8-10 屏显~~ 8-10~~屏显~~ 8-10~~所示。

屏显 8-10 第一个一次间接索引的数据块盘内容

```
103400 6d 6d 6d 6d 6d 6d 6d 6d 6d 6d 6d 6d 6d 6d 6d 6d
*
```

正好是上面 0x6c 的下一个字母 0x6d（即字母表第 13 个字母 m），后续的文件数据，读者可以自行操作实践并对比检验。

8.3.4. 目录结构

在掌握了 EXT2 如何通过索引节点信息找到文件数据盘块的过程后，我们继续观察 EXT2 的目录结构。

首先，我们在 bean 目录下创建目录 directory，此时/mnt/bean 目录内部如屏显 8-11 屏显~~ 8-11~~屏显~~ 8-11~~所示。

屏显 8-11 创建 directory 目录后的/mnt/bean 目录

```
[root@localhost bean]# ll
total 58
drwxr-xr-x.     2 root root  1024 Mar 23 21:36 directory
-rw-r--r--.     1 root root 42551 Mar 22 15:53 hello.h
drwx------.     2 root root 12288 Mar 22 15:53 lost+found
```

然后再在 directory 目录下创建 3 个普通文件（regular file，使用 touch 命令完成），如屏显 8-12 屏显~~ 8-12~~屏显~~ 8-12~~所示。

屏显 8-12 在/mnt/bean/directory 中创建三个普通文件

```
[root@localhost directory]# ll
total 3
-rw-r--r--. 1 root root 0 Mar 22 16:00 file1
-rw-r--r--. 1 root root 0 Mar 22 16:00 file2
-rw-r--r--. 1 root root 0 Mar 22 16:00 file3
```

作了任何文件的改变（例如新建或删除了文件，或者对文件进行了些操作），请读者接着执行一个 sync 命令将这些改变落实到设备上（本实验使用的是环回设备），否则有可能通过/dev/loop0 观察不到修改变化。

目录文件的索引节点定位

由于目录是一种特殊文件，如果想读入它的内容也需要先获得它的 inode 索引节点号。使用 ls –i 命令查看 directory 的索引节点（inode）号，如屏显 8-13 屏显~~ 8-13~~屏显~~ 8-13~~所示。

屏显 8-13 用 ls –i 查看文件的索引节点号

```
[root@localhost bean]# ls -ia
77217 .  2 ..  77217 directory      12 hello.h      11 lost+found
```

从屏显 8-13 屏显~~ 8-13~~屏显~~ 8-13~~可见，目录 directory 的 inode 号为 77217。下面将逐步完成目录文件的索引节点定位：1）所在块组；2）对应块组上的索引节点表内偏移。第一步将根据公式，计算目录文件索引节点所在块组号 *block_group_numer*，如公式 8-1 公式~~ 8-1~~公式

8-1 所示。本例子中，前面已经通过 dumpe2fs 命令的输出内容中找到“Inodes per group”项为 2032，如屏显 8-3 屏显 8-3 屏显 8-3 所示。代入相应的数值可以得到，目录 directory 的块组号 *block_group_number* 是 38。

然后再定位 77217 号索引节点在块组 38 内部的位置。根据公式 8-2 公式 8-2 公式 8-2 计算目录文件索引节点的真实位置，代入相应的数值可以得到偏移值 inode_offset 是 0。

读入目录文件的索引节点

接下来，查找 inode 所在块组（38 号块组）的 inode_table 起始块号。使用 dumpe2fs /dev/loop0 输出的结果中查找第 38 项，如所示。

屏显 8-14 第 38 号块组的信息

```
Group 38: (Blocks 311297-319488)
  Block bitmap at 311297 (+0), Inode bitmap at 311298 (+1)
  Inode table at 311299-311552 (+2)
  7935 free blocks, 2028 free inodes, 1 directories
  Free blocks: 311554-319488
  Free inodes: 77221-79248
```

可以看到块组号为 38 的 inode table 起始块号为 311299，再乘以 1024 并转为十六进制即 0x13000c00，这个数值就是 inode table 在/dev/loop0 中的地址。用 dd if=/dev/loop0 bs=1k skip=311299 count=1 | od –tx1 –Ax > directory_i_table 命令将 inode table 的第一个盘块读入到 directory_i_talbe 文件中，或编写 C 程序且 lseek()到 0x13000c00 字节偏移处再读入该盘块。查看到/dev/loop0 在 0x13000c00 以及后面邻近的内容如屏显 8-15 屏显 8-15 屏显 8-15 所示。

屏显 8-15 38 号块组的 inode tabel 内容（第 0 项就是 77217 号索引节点内容）

```
0x13000c00 ed 41 00 00 00 04 00 00 51 b5 d7 58 50 b5 d7 58
0x13000c10 50 b5 d7 58 00 00 00 00 00 00 02 00 04 00 00 00
0x13000c20 00 00 00 00 04 00 00 00 01 c1 04 00 00 00 00 00
0x13000c30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x13000c40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x13000c50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

由于 77217 号索引节点在块组 38 内部的偏移为 0，也就是说第 0 项就是我们要找的索引节点。根据代码 8-3 代码 8-3 代码 8-3 第 300 行可知，ext2_inode 的第三个字段 i_size 是文件大小，在上面输出内容中查找对应位置，可以看到数值为 0x400，即 1024（这与上面屏显 8-11 屏显 8-11 屏显 8-11 中 ll 指令给出的 directory 大小一致）。为了得到 directory 目录文件的内容，还要根据代码 8-3 代码 8-3 代码 8-3 第 320 行推算出 ext2_inode 的 i_block[]数组的起始位置，可以看到一个数值 0x065c01（屏显 8-15 屏显 8-15 屏显 8-15 中灰色标注）即为 i_block[0]（直接块）。由于该文件只占一个盘块，因此后面的 i_block[1]~i_block[11]都为 0。

由于 i_block[0]数值是块号，如果使用 dd 命令则可以直接使用该数值（0x04c101= 311553），例如 dd if=/dev/loop0 bs=1k skip=311553 count=1 | od –tx1 –Ax > directory_content 将该盘块数据保存到 directory_content 文件中（od 命令使用 16 进制输出）。如果是通过编写 C 程序打开裸设备/dev/loop0 则需要用 lseek()到指定字节偏移处（还需要乘以 1024，得到地址 0x13040400）。下面给出了/dev/loop0 在 0x13040400 及后面邻近内容，如屏显 8-16 屏显 8-16 屏显 8-16 所示。

屏显 8-16 directory 目录文件的内容

```
0x13040400 a1 2d 01 00 0c 00 01 02 2e 00 00 00 02 00 00 00
0x13040410 0c 00 02 02 2e 2e 00 00 a2 2d 01 00 10 00 05 01
0x13040420 66 69 6c 65 31 00 00 00 a3 2d 01 00 10 00 05 01
0x13040430 66 69 6c 65 32 00 00 00 a4 2d 01 00 c8 03 05 01
0x13040440 66 69 6c 65 33 00 00 00 00 00 00 00 00 00 00 00
0x13040450 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x13040460 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x13040470 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x13040480 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x13040490 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

目录项解读

读取目录文件内容后，只差对其内容的进行解读了。根据代码 8-4 代码 8-4 代码 8-4 的 ext2_dir_entry_2 结构体定义，获得 directory 目录文件内部的各条目录项内容如下：

1） 首 4 字节的数据是目录文件第一个目录项的 inode 号 0x012da1=77217，该目录是目录自身（也就是常说的“.”目录）。这个目录项指向的文件的 inode 号为 77217，即是 directory 自身。这跟前面所用屏显 8-13 屏显 8-13 屏显 8-13 的 ls –i 命令所查看到的 inode 一致。

2） 接着的 2 字节的数据是 rec_len 属性 0xC=12，说明第一个目录项的大小为 12 字节，这个属性是正确分割目录文件内容的重要信息，否则无法定位下一条目录项（目录项大小是不固定的）。

3） 接下来的 1 字节数据是 name_len，这里是为了正确读取 name[]所需要的信息，0x1=1。

4） 再接下来的 1 字节数据是 file_type，0x2 代表这个目录项指向的文件是目录文件（文件类型定义请参见代码 8-4 代码 8-4 代码 8-4 的 603 行）。

5） 再接下来，该目录的所有数据都是名字字符串 name[]，然后前面知道了 name_len 为 1，所以我们只看 name[]的第一个字节是数据，0x2e 这里就要用 ASCII 解读，0x2e=“.”。

继续分析可以知道，第二个项是“..”目录（即父目录/mnt/bean）索引节点号是 0x02=2、目录项长度 0x0c=12 字节、文件名字符串长 0x02=2 字节；第三个项是“file1”文件，其索引节点号是 0x012da2=77218、目录项长度 0x10=16 字节、文件名字符串长 0x05=5 字节；第四个项是“file2”文件，其索引节点号是 0x012da3=77219、目录项长度 0x10=16 字节、文件名字符串长 0x05=5 字节；第五个项是“file3”文件，其索引节点号是 0x012da4=77220、目录项长度 0x03c8=968 字节、文件名字符串长 0x05=5 字节。

因此，相应的目录结构应该如图 8-5 图 8-5 图 8-5 所示，解读结果和屏显 8-13 屏显 8-13 屏显 8-13 一致。

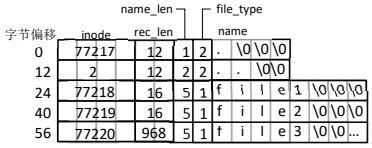

图 8-5 dirctory 中的目录项示意图

接下来，我们试着再在目录 directory 下面创建管道文件（用 mkfifo 命令）、目录文件、节点文件（用 mknod 命令创建的），然后用 ll 命令查看获得如屏显 8-17 屏显~~ 8-17~~屏显~~ 8-17~~所示的输出。

设置了格式：字体：五号，非加粗，图案：清除（白色）

屏显 8-17 在/mnt/bean/directory 中的管道文件、目录文件、节点文件

```
[root@localhost directory]# ll
total 11
drwxr-xr-x. 2 root root     1024 Mar 28 15:56 dir1
drwxr-xr-x. 2 root root     1024 Mar 28 15:56 dir2
drwxr-xr-x. 2 root root     1024 Mar 28 15:56 dir3
prw-r--r--. 1 root root        0 Mar 28 15:56 fifo1
-rw-r--r--. 1 root root        0 Mar 26 20:34 file1
-rw-r--r--. 1 root root        0 Mar 27 18:35 file2
-rw-r--r--. 1 root root        0 Mar 27 18:35 file3
crw-r--r--. 1 root root 200, 200 Mar 28 16:06 node1
```

然后通过 ls –i 查看 inode 索引号，如屏显 8-18 屏显~~ 8-18~~屏显~~ 8-18~~所示。

屏显 8-18 在/mnt/bean/directory 中文件的索引节点号

```
[root@localhost directory]# ls -i
77221 dir1  77223 dir3   77218 file1  77220 file3 77222 dir2  77224 fifo1  77219 file2  77225 node1
```

再次查看/dev/loop0 在地址 0x13040400 以及后面几条地址的输出内容（即/mnt/bean/directory 目录文件的内容），如屏显 8-19 屏显~~ 8-19~~屏显~~ 8-19~~所示。

屏显 8-19 directory 创建其他文件后目录文件的内容

```
0x13040400 a1 2d 01 00 0c 00 01 02 2e 00 00 00 02 00 00 00
0x13040410 0c 00 02 02 2e 2e 00 00 a2 2d 01 00 10 00 05 01
0x13040420 66 69 6c 65 31 00 00 00 a3 2d 01 00 10 00 05 01
0x13040430 66 69 6c 65 32 00 00 00 a4 2d 01 00 10 00 05 01
0x13040440 66 69 6c 65 33 00 00 00 a5 2d 01 00 0c 00 04 02
0x13040450 64 69 72 31 a6 2d 01 00 0c 00 04 02 64 69 72 32
0x13040460 a7 2d 01 00 0c 00 04 02 64 69 72 33 a8 2d 01 00
0x13040470 10 00 05 05 66 69 66 6f 31 00 00 00 a9 2d 01 00
0x13040480 84 03 05 03 6e 6f 64 65 31 00 00 00 00 00 00 00
0x13040490 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

可见，目录 directory 所存放的数据中，前面几个关键字段（对应目录项”.”、”..”、”file1”、”file2”以及”file3”）都没有变化，而在”file3”的目录项之后，又添加了一些数据，接下来进行对”file3”后面数据的解读：

根据代码 8-4 的 ext2_dir_entry_2 结构体定义，分析可知道：

下一个文件的索引节点是 0x012da5=77221、目录项长度是 0x0c=12、名字长度 0x04=4、文件类型 0x2=2（根据代码 8-4 的 file_type 枚举类型定义可知，该文件是目录）、名字通过查 ASCII 码表可以得到”dir1”；

以同样方式解读后面的数据：下一个文件索引节点是 77222、目录项长度为 12、名字长度为 4、文件类型为 2(目录)、文件名是”dir2”；

再下一个文件索引节点是 77223、目录项长度为 12、名字长度为 4、文件类型为 2(目录)、文件名为”dir3”；

再下一个文件索引节点是 77224、目录项长度为 16、名字长度为 5、文件类型为 5(FIFO)文件名为”fifo1”；

最后一个文件索引节点是 77225、目录项长度为 900、名字长度为 5、文件类型为 3(字符设备文件)、文件名为”node1”

字符设备、块设备和 FIFO 文件等类型的文件，不需要在磁盘上占用空间，即它们不需要数据盘块来存放文件内容——只需要目录项和相应的索引节点即可。