本文主要讲述 Linux 上比较流行的 ext2 文件系统在硬盘分区上的详细布局情况。Ext2 文件系统加上日志支持的下一个版本是 ext3 文件系统，它和 ext2 文件系统在硬盘布局上是一样的，其差别仅仅是 ext3 文件系统在硬盘上多出了一个特殊的 inode（可以理解为一个特殊文件），用来记录文件系统的日志，也即所谓的 journal。由于本文并不讨论日志文件，所以本文的内容对于 ext2 和 ext3 都是适用的。

1 前言

本文的资料来源是 Linux 内核中 ext3 文件系统的源代码。为了便于读者查阅源代码，本文中一些关键的技术词汇都使用了内核源代码中所使用的英语单词，而没有使用相应的中文翻译。(这种方法是否恰当，还请读者朋友们指教。)

2 粗略的描述

对于 ext2 文件系统来说，硬盘分区首先被划分为一个个的 block，一个 ext2 文件系统上的每个 block 都是一样大小的，但是对于不同的 ext2 文件系统，block 的大小可以有区别。典型的 block 大小是 1024 bytes 或者 4096 bytes。这个大小在创建 ext2 文件系统的时候被决定，它可以由系统管理员指定，也可以由文件系统的创建程序根据硬盘分区的大小，自动选择一个较合理的值。这些 blocks 被聚在一起分成几个大的 block group。每个 block group 中有多少个 block 是固定的。

每个 block group 都相对应一个 group descriptor，这些 group descriptor 被聚在一起放在硬盘分区的开头部分，跟在 super block 的后面。所谓 super block，我们下面还要讲到。在这个 descriptor 当中有几个重要的 block 指针。我们这里所说的 block 指针，就是指硬盘分区上的 block 号数，比如，指针的值为 0，我们就说它是指向硬盘分区上的 block 0；指针的值为 1023，我们就说它是指向硬盘分区上的 block 1023。我们注意到，一个硬盘分区上的 block 计数是从 0 开始的，并且这个计数对于这个硬盘分区来说是全局性质的。

在 block group 的 group descriptor 中，其中有一个 block 指针指向这个 block group 的 block bitmap，block bitmap 中的每个 bit 表示一个 block，如果该 bit 为 0，表示该 block 中有数据，如果 bit 为 1，则表示该 block 是空闲的。注意，这个 block bitmap 本身也正好只有一个 block 那么大小。假设 block 大小为 S bytes,那么 block bitmap 当中只能记载 8*S 个 block 的情况(因为一个 byte 等于 8 个 bits,而一个 bit 对应一个 block)。这也就是说，一个 block group 最多只能有 8*S*S bytes 这么大。

在 block group 的 group descriptor 中另有一个 block 指针指向 inode bitmap，这个 bitmap 同样也是正好有一个 block 那么大，里面的每一个 bit 相对应一个 inode。硬盘上的一个 inode 大体上相对应于文件系统上的一个文件或者目录。关于 inode，我们下面还要进一步讲到。

在 block group 的 descriptor 中另一个重要的 block 指针，是指向所谓的 inode table。这个 inode table 就不止一个 block 那么大了。这个 inode table 就是这个 block group 中所聚集到的全部 inode 放在一起形成的。

一个 inode 当中记载的最关键的信息，是这个 inode 中的用户数据存放在什么地方。我们在前面提到，一个 inode 大体上相对应于文件系统中的一个文件，那么用户文件的内容存放在什么地方，这就是一个 inode 要回答的问题。一个 inode 通过提供一系列的 block 指针，来回答这个问题。这些 block 指针指向的 block，里面就存放了用户文件的内容。

2.1 回顾

现在我们回顾一下。硬盘分区首先被分为好多个 block。这些 block 聚在一起，被分成几组，也就是 block group。每个 block group 都有一个 group descriptor。所有这些 descriptor 被聚在一起，放在硬盘分区的开头部分，跟在 super block 的后面。从 group descriptor 我们可以通过 block 指针，找到这个 block group 的 inode table 和 block bitmap 等等。从 inode table 里面，我们就可以看到一个个的 inode 了。从一个 inode，我们通过它里面的 block 指针，就可以进而找到存放用户数据的那些 block。我们还要提一下，block 指针不是可以到处乱指的。一个 block group 的 block bitmap 和 inode bitmap 以及 inode table，都依次存放在这个 block group 的开头部分，而那些存放用户数据的 block 就紧跟在它们的后面。一个 block group 结束后，另一个 block group 又跟着开始。

3 详细的布局情况

3.1 Super Block

所谓 ext2 文件系统的 super block，就是硬盘分区开头（开头的第一个 byte 是 byte 0）从 byte 1024 开始往后的一部分数据。由于 block size 最小是 1024 bytes，所以 super block 可能是在 block 1 中（此时 block 的大小正好是 1024 bytes），也可能是在 block 0 中。

硬盘分区上 ext3 文件系统的 super block 的详细情况如下。其中 __u32 是表示 unsigned 不带符号的 32 bits 的数据类型，其余类推。这是 Linux 内核中所用到的数据类型，如果是开发用户空间（user-space）的程序，可以根据具体计算机平台的情况，用 unsigned long 等等来代替。下面列表中关于 fragments 的部分可以忽略，Linux 上的 ext3 文件系统并没有实现 fragments 这个特性。另外要注意，ext3 文件系统在硬盘分区上的数据是按照 Intel 的 Little-endian 格式存放的，如果是在 PC 以外的平台上开发 ext3 相关的程序，要特别注意这一点。如果只是在 PC 上做开发，倒不用特别注意。

```
struct ext3_super_block {
/*00*/ __u32 s_inodes_count;      /* inodes 计数 */
       __u32 s_blocks_count;      /* blocks 计数 */
       __u32 s_r_blocks_count;    /* 保留的 blocks 计数 */
       __u32 s_free_blocks_count; /* 空闲的 blocks 计数 */
/*10*/ __u32 s_free_inodes_count; /* 空闲的 inodes 计数 */
       __u32 s_first_data_block;  /* 第一个数据 block */
       __u32 s_log_block_size;    /* block 的大小 */
       __s32 s_log_frag_size;     /* 可以忽略 */
/*20*/ __u32 s_blocks_per_group;  /* 每 block group 的 block 数量 */
       __u32 s_frags_per_group;   /* 可以忽略 */
       __u32 s_inodes_per_group;  /* 每 block group 的 inode 数量 */
       __u32 s_mtime;             /* Mount time */
/*30*/ __u32 s_wtime;             /* Write time */
       __u16 s_mnt_count;         /* Mount count */
       __s16 s_max_mnt_count;     /* Maximal mount count */
       __u16 s_magic;             /* Magic 签名 */
       __u16 s_state;             /* File system state */
```

```
        __u16 s_errors;           /* Behaviour when detecting errors */
        __u16 s_minor_rev_level;  /* minor revision level */
/*40*/  __u32 s_lastcheck;        /* time of last check */
        __u32 s_checkinterval;    /* max. time between checks */
        __u32 s_creator_os;       /* 可以忽略 */
        __u32 s_rev_level;        /* Revision level */
/*50*/  __u16 s_def_resuid;       /* Default uid for reserved blocks */
        __u16 s_def_resgid;       /* Default gid for reserved blocks */
        __u32 s_first_ino;        /* First non-reserved inode */
        __u16 s_inode_size;       /* size of inode structure */
        __u16 s_block_group_nr;   /* block group # of this superblock */
        __u32 s_feature_compat;   /* compatible feature set */
/*60*/  __u32 s_feature_incompat;  /* incompatible feature set */
        __u32 s_feature_ro_compat; /* readonly-compatible feature set */
/*68*/  __u8  s_uuid[16];         /* 128-bit uuid for volume */
/*78*/  char  s_volume_name[16];  /* volume name */
/*88*/  char  s_last_mounted[64];  /* directory where last mounted */
/*C8*/  __u32 s_algorithm_usage_bitmap; /* 可以忽略 */
        __u8  s_prealloc_blocks;       /* 可以忽略 */
        __u8  s_prealloc_dir_blocks;   /* 可以忽略 */
        __u16 s_padding1;              /* 可以忽略 */
/*D0*/  __u8  s_journal_uuid[16]; /* uuid of journal superblock */
/*E0*/  __u32 s_journal_inum;     /* 日志文件的 inode 号数 */
        __u32 s_journal_dev;      /* 日志文件的设备号 */
        __u32 s_last_orphan;      /* start of list of inodes to delete */
/*EC*/  __u32 s_reserved[197];    /* 可以忽略 */
};
```

我们可以看到，super block 一共有 1024 bytes 那么大。在 super block 中，我们第一个要关心的字段是 magic 签名，对于 ext2 和 ext3 文件系统来说，这个字段的值应该正好等于 0xEF53。如果不等的话，那么这个硬盘分区上肯定不是一个正常的 ext2 或 ext3 文件系统。从这里，我们也可以估计到，ext2 和 ext3 的兼容性一定是很强的，不然的话，Linux 内核的开发者应该会为 ext3 文件系统另选一个 magic 签名才对。

在 super block 中另一个重要的字段是 s_log_block_size。从这个字段，我们可以得出真正的 block 的大小。我们把真正 block 的大小记作 B，B = 1 << (s_log_block_size + 10)，单位是 bytes。举例来说，如果这个字段是 0，那么 block 的大小就是 1024 bytes，这正好就是最小的 block 大小；如果这个字段是 2，那么 block 大小就是 4096 bytes。从这里我们就得到了 block 的大小这一非常重要的数据。

3.2 Group Descriptors

我们继续往下，看跟在 super block 后面的一堆 group descriptors。首先注意到 super block 是从 byte 1024 开始，一共有 1024 bytes 那么大。而 group descriptors 是从 super block 后面的第一个 block 开始。也就是说，如果 super block 是在 block 0，那么 group descriptors 就是从 block 1

开始；如果 super block 是在 block 1，那么 group descriptors 就是从 block 2 开始。因为 super block 一共只有 1024 bytes 那么大，所以不会超出一个 block 的边界。如果一个 block 正好是 1024 bytes 那么大的话，我们看到 group descriptors 就是紧跟在 super block 后面的了，没有留一点空隙。而如果一个 block 是 4096 bytes 那么大的话，那么在 group descriptors（从 byte 4096 开始）和 super block 的结尾之间，就有一定的空隙（4096 - 2048 bytes）。

那么硬盘分区上一共有多少个 block group，或者说一共有多少个 group descriptors，这我们要在 super block 中找答案。super block 中的 s_blocks_count 记录了硬盘分区上的 block 的总数，而 s_blocks_per_group 记录了每个 group 中有多少个 block。显然，文件系统上的 block groups 数量，我们把它记作 G，G = (s_blocks_count - s_first_data_block - 1) / s_blocks_per_group + 1。为什么要减去 s_first_data_block，因为 s_blocks_count 是硬盘分区上全部的 block 的数量，而在 s_first_data_block 之前的 block 是不归 block group 管的，所以当然要减去。最后为什么又要加一，这是因为尾巴上可能多出来一些 block，这些 block 我们要把它划在一个相对较小的 group 里面。

注意，硬盘分区上的所有这些 group descriptors 要能塞在一个 block 里面。也就是说 groups_count * descriptor_size 必须小于等于 block_size。

知道了硬盘分区上一共有多少个 block group，我们就可以把这么多个 group descriptors 读出来了。先来看看 group descriptor 是什么样子的。

```
struct ext3_group_desc
{
 __u32 bg_block_bitmap;      /* block 指针指向 block bitmap */
 __u32 bg_inode_bitmap;      /* block 指针指向 inode bitmap */
 __u32 bg_inode_table;       /* block 指针指向 inodes table */
 __u16 bg_free_blocks_count; /* 空闲的 blocks 计数 */
 __u16 bg_free_inodes_count; /* 空闲的 inodes 计数 */
 __u16 bg_used_dirs_count;   /* 目录计数 */
 __u16 bg_pad;               /* 可以忽略 */
 __u32 bg_reserved[3];       /* 可以忽略 */
};
```

每个 group descriptor 是 32 bytes 那么大。从上面，我们看到了三个关键的 block 指针，这三个关键的 block 指针，我们已经在前面都提到过了。

3.3 Inode

前面都准备好了以后，我们现在终于可以开始读取文件了。首先要读的，当然是文件系统的根目录。注意，这里所谓的根目录，是相对于这一个文件系统或者说硬盘分区而言的，它并不一定是整个 Linux 操作系统上的根目录。这里的这个 root 目录存放在一个固定的 inode 中，这就是文件系统上的 inode 2。需要提到 inode 计数同 block 计数一样，也是全局性质的。这里需要特别注意的是，inode 计数是从 1 开始的，而前面我们提到过 block 计数是从 0 开始，这个不同在开发程序的时候要特别留心。（这一奇怪的 inode 计数方法，曾经让本文作者大伤脑筋。）

那么，我们先来看一下得到一个 inode 号数以后，怎样读取这个 inode 中的用户数据。在 super

block 中有一个字段 s_inodes_per_group 记载了每个 block group 中有多少个 inode。用我们得到的 inode 号数除以 s_inodes_per_group，我们就知道了我们要的这个 inode 是在哪一个 block group 里面，这个除法的余数也告诉我们，我们要的这个 inode 是这个 block group 里面的第几个 inode；然后，我们可以先找到这个 block group 的 group descriptor，从这个 descriptor，我们找到这个 group 的 inode table，再从 inode table 找到我们要的第几个 inode，再以后，我们就可以开始读取 inode 中的用户数据了。

这个公式是这样的：block_group = (ino - 1) / s_inodes_per_group。这里 ino 就是我们的 inode 号数。而 offset = (ino - 1) % s_inodes_per_group，这个 offset 就指出了我们要的 inode 是这个 block group 里面的第几个 inode。

找到这个 inode 之后，我们来具体的看看 inode 是什么样的。

```
struct ext3_inode {
 __u16 i_mode;    /* File mode */
 __u16 i_uid;     /* Low 16 bits of Owner Uid */
 __u32 i_size;    /* 文件大小，单位是 byte */
 __u32 i_atime;   /* Access time */
 __u32 i_ctime;   /* Creation time */
 __u32 i_mtime;   /* Modification time */
 __u32 i_dtime;   /* Deletion Time */
 __u16 i_gid;     /* Low 16 bits of Group Id */
 __u16 i_links_count;          /* Links count */
 __u32 i_blocks;               /* blocks 计数 */
 __u32 i_flags;                /* File flags */
 __u32 l_i_reserved1;          /* 可以忽略 */
 __u32 i_block[EXT3_N_BLOCKS]; /* 一组 block 指针 */
 __u32 i_generation;           /* 可以忽略 */
 __u32 i_file_acl;             /* 可以忽略 */
 __u32 i_dir_acl;              /* 可以忽略 */
 __u32 i_faddr;                /* 可以忽略 */
 __u8  l_i_frag;               /* 可以忽略 */
 __u8  l_i_fsize;              /* 可以忽略 */
 __u16 i_pad1;                 /* 可以忽略 */
 __u16 l_i_uid_high;           /* 可以忽略 */
 __u16 l_i_gid_high;           /* 可以忽略 */
 __u32 l_i_reserved2;          /* 可以忽略 */
};
```

我们看到在 inode 里面可以存放 EXT3_N_BLOCKS（= 15）这么多个 block 指针。用户数据就从这些 block 里面获得。15 个 blocks 不一定放得下全部的用户数据，在这里 ext3 文件系统采取了一种分层的结构。这组 15 个 block 指针的前 12 个是所谓的 direct blocks，里面直接存放的就是用户数据。第 13 个 block，也就是所谓的 indirect block，里面存放的全部是 block 指针，这些 block 指针指向的 block 才被用来存放用户数据。第 14 个 block 是所谓的 double indirect block，里面存放的全是 block 指

针，这些 block 指针指向的 block 也被全部用来存放 block 指针，而这些 block 指针指向的 block，才被用来存放用户数据。第 15 个 block 是所谓的 triple indirect block，比上面说的 double indirect block 有多了一层 block 指针。作为练习，读者可以计算一下，这样的分层结构可以使一个 inode 中最多存放多少字节的用户数据。（计算所需的信息是否已经足够？还缺少哪一个关键数据？）

一个 inode 里面实际有多少个 block，这是由 inode 字段 i_size 再通过计算得到的。i_size 记录的是文件或者目录的实际大小，用它的值除以 block 的大小，就可以得出这个 inode 一共占有几个 block。注意上面的 i_blocks 字段，粗心的读者可能会以为是这一字段记录了一个 inode 中实际用到多少个 block,其实不是的。那么这一字段是干什么用的呢,读者朋友们可以借这个机会,体验一下阅读 Linux 内核源代码的乐趣。;-)

3.4 文件系统的目录结构

现在我们已经可以读取 inode 的内容了，再往后，我们将要读取文件系统上文件和目录的内容。读取文件的内容，只要把相应的 inode 的内容全部读出来就行了；而目录只是一种固定格式的文件，这个文件按照固定的格式记录了目录中有哪些文件，以及它们的文件名，和 inode 号数等等。

```
struct ext3_dir_entry_2 {
 __u32 inode;    /* Inode 号数 */
 __u16 rec_len;  /* Directory entry length */
 __u8  name_len; /* Name length */
 __u8  file_type;
 char  name[EXT3_NAME_LEN]; /* File name */
};
```

上面用到的 EXT3_NAME_LEN 是 255。注意，在硬盘分区上的 dir entry 不是固定长度的，每个 dir entry 的长度由上面的 rec_len 字段记录。

4 小结

有了以上的这些信息，我们就可以读取一个 ext3 文件系统的全部内容了。如果读者有 Windows 驱动程序开发的经验，从本文的信息，开发一个 Windows 下只读的 ext3 文件系统是可能的。但是要想又读又写，那还需要了解 Ext3 的日志文件的结构，而本文限于篇幅，并没有包括这方面的内容。

参考文献

1 Remy Card, Theodore Ts'o, Stephen Tweedie, Design and Implementation of the Second Extended Filesystem, http://web.mit.edu/tytso/www/linux/ext2intro.html

2 Linux Kernel 2.4.18 Source Code, http://lxr.linux.no/source/fs/ext3/

关于作者：赵蔚(zhaoway@public1.ptt.js.cn) ，是中国大陆第二个注册的 Debian GNU/Linux 义务开发人员。作者对于 LISP 和 Lambda Calculus 也有浓厚的兴趣。