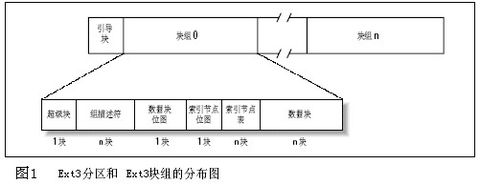
**Ext2 文件系统的硬盘布局+深入理解ext2/ext3文件系统真相**

摘要：我们对于ext2/ext3文件系统的了解大多来自操作系统的课本或者《understanding linux kernel》这本书，但是最近做了一个项目，需要根据文件的inode结点号找到inode本身。因此需要对ext2/ext3文件系统做深入的理解。在做的过程中才发现，原来书上讲的理论和现实的差距太大了。可能是因为书比较早，或者书上只是讲了一个概念模型，以至从实际工程的角度来看，课本上有好多的东西简直可以说是错误的。本文就把在工程中发现的问题和ext2/ext3文件系统的真相展示出来。

**第一章：经典而又经典的图其实是错的**

首先说明ext3文件系统和ext2文件系统的区别。ext3文件系统是带有日志的ext2文件系统，在设计时就秉承尽可能与ext2文件系统兼容的理念，因此他的文件系统数据结构与ext2文件系统的本质上是相同的。事实上可以把一个ext3文件系统卸载后做为ext2文件系统重新安装。反之也可以把创建日志的ext2文件系统做为ext3文件系统挂载。

ext2文件系统的磁盘数据结构如下图所示：



 除了硬盘分区中的第一块作为引导块所保留，不受Ext3文件系统管理以外，其余部分分成块组（blockgroup），每个快组的分布下图所示，由于内核尽可能把属于一个文件的数据块存放在同一块中，所以组块减少了文件碎片。块组中的每块包含下列信息：

   超级块：文件系统超级块的一个拷贝

   组描述符：一组块组描述符的拷贝

   数据块位图：一个数据块位图

   索引节点位图：一组索引节点

   索引节点表：一个索引节点位图

   数据块：存放文件数据

其中超级块描述整个文件系统的信息，包括此文件系统共有多少个块组，每个块组包含多少个块，包含的结点的个数，每个结点的大小。

组描述符表描述的是每个组的情况，每个组有一个组描述符，大小为32字节。文件系统中所有的组描述符放在一起构成了组描述符表。

组描述符表之后是数据块位图和索引结点位图各占一个块。随后就是这个组的索引结点表，每个索引结点占用128个字节。

块组的大小是由数据块位图决定的。假设文件系统的数据块大小为b字节（可以配置为1K，2K，4K），则每个数据块位图可以描述的磁盘块大小为 b \* 8 ，即8 \* b \* b 个字节大小。通常情况下块大小为4K，则一个块组包含 32k个物理块。需要说明是这32k个块并不全是数据块，而是包含元数据块和数据块的总和。这样看似备份的元数据浪费了数据块，但这种浪费换来的是查找性能上的提升。

块组大小确定之后，整个系统中的块组数就可以确定了，即磁盘总容量/每个块组的容量。

确定块组的重要的意义在于确定inode号。在整个文件系统中inode号是统一编址的，因此可以根据inode号确定这个inode所在的组号以及组内的偏移，进而找到这个inode。

根据上图，inode结点的地址的字节偏移应该按如下的方法计算：

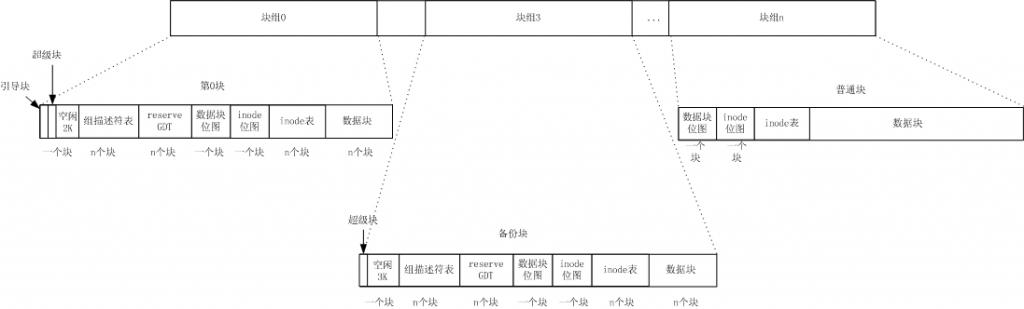
inode地址 = ((inode\_no / inode\_per\_group) \* block\_per\_group + (1 + n + 1 +１)) \* block\_size + ( inode\_no % inode\_per\_group ) \* inode\_size + 1024

其中inode\_no为结点号，inode\_per\_group为每个块组含有多少个结点， block\_per\_group为每个块组的块的个数， (1 + n + 1 +１) 表示一个超级块，一个数据块位图，一个索引结点位图，n表示块组的个数，最后加上引导块占用的1K个字节。

但是实际的情况是这个公式是错误的，按照这个公式不可能读到索引结点。

**第二章：寻找inode结点**

产生错误的原因在于课本上所讲的数据结构是错误的，或者说只是一个逻辑上的模型，与实际在物理磁盘上存放的数据结构有很大的差别。首先看一张真正的磁盘数据结构图：



对比和上一张图的不同，真实的磁盘布局中有三种块组，一种是块组0，一种备份块组，一种是普通块组。

块组0即包含引导块的块组，通常系统使用元数据信息都是从这个块组中读取，只有当这个块组中的元数据发生错误时才会从备份的块组中恢复。

由此产生了和课本的冲突，并不是所有的块组都会备份一份元数据信息，只有部分块组会备份。但是哪些块组会备份呢？只有当块组号是3，5，7的幂的块组才会备份元数据，其他块组则是做为普通块组只存放数据，不备份元数据。

和课本的另一个冲突，引导块占据1K的空间，他虽然不被文件系统管理，但他确实占据了磁盘0块组的第一个块。所以教科书上的画法是不妥当的，或者说教科书只是画出了一种逻辑的磁盘结构，并不反应现实。在0块组的第一个块中，引导块和超级块各占了1K的空间，留下了2K的空闲。而在其他的备份块组中则没有引导块的备份，只是在第一个块中备份了一个超级块。

和课本的下一个冲突，在实际中出现了一个课本上没有提到过的区域，reserve GDT,而且还是一个占用了不少块的区域。这个区域是为了以后扩容用的。当以后为文件系统扩容时，可能人增加新的块组，块组描述符表也会相应增加，这个reserve GDT就是为那时使用的。这个区域占用多少个块我还没有找到计算方法，但是在super block结构中有一个域指示了这个区域占用的块数，可以直接读出。

至此，我们得到计算inode结点字节偏移的真正方法：

inode\_address = ((inode\_no / inode\_per\_group) \* block\_per\_group + (1+n+n+1+1) \* is\_backup\_group ) \* block\_size + (inode\_no % inode\_per\_group) \* inode\_size

这就是真实的ext3文件系统的磁盘布局。到此我们可以拿到inode，可以进行查找操作了。

**第三章：问题才刚刚开始**

**第一个问题：inode结点号真的是对的吗？**

每个文件有一个inode结点，这个结点对应一个inode结点号。这个结点号可以由ls –i命令得到，也可以用debugfs 工具得到。但是这个inode结点号真的就是在磁盘物理空间上的结点顺序号吗? 看了内核中相应的一段代码才知道，这个inode结点号是不能直接拿来用的，必须减1才能用。减1是因为没有0号结点，我们拿到的结点号是从1开始计数的。

**第二个问题：读写大文件**

第二个问题是当把设备做为文件打开时，可能会因为设备过大导致文件无法支持，但是这时做的lseek， read操作会全部正确返回，但这时操作的结果是错误的，因为大的数据被截断了。解决方法是使用lseek64,或者\_llseek。

Ext2 文件系统的硬盘布局  
  
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/

Linux文件系统类型全接触   
随着Linux的不断发展，Linux所支持的文件系统类型也在迅速扩充。伴随着 Linux 2.4 版本的发行，出现了大量的文件系统可能性，其中包括 ReiserFS、XFS、JFS 和其它文件系统。每一个不同版本的Linux所支持的文件系统类型种类都有所不同，如何知道自己的Linux发行版本的类型哪？可以这样操作：（以笔者使用的Mandrake Linux 8.2为例,）   
以超级用户权限登陆Linux，进入/Lib/modules/2.4.18-6mdk/kernel/fs目录执行命令（不同Linux发行版本的Fs目录有些不同你可以用查找FS文件夹的方法找到它，见图－1）：   
＃ls   
  
图－1 Mandrake Linux 支持的文件系统类型   
/Lib/modules/2.4.18-6mdk/kernel/fs中查出当前系统所支持的文件系统种类。从图－1中可以看到笔者使用的Mandrake Linux 8.2支持的文件系统非常多。Linux系统核心支持十多种文件系统类型：jfs、 ReiserFS、ext、ext2、ext3、iso9660、xfs、 minx、msdos、umsdos、Vfat、NTFS、Hpfs、Nfs、smb、sysv、proc等。   
  
这里我们对最常用的几个文件系统的发展情况和优缺点作详细介绍：ext、ext2、ext3、jsf、 、xfs、ReiserFS。   
一、 ext   
ext是第一个专门为Linux的文件系统类型，叫做扩展文件系统。它在1992年4月完成的。它为Linux的发展取得了重要作用。但是在性能和兼容性上存在许多缺陷。现在已经很少使用了。   
二、 ext2   
ext2是为解决ext文件系统的缺陷而设计的可扩展的高性能的文件系统。又被称为二级扩展文件系统。它是在1993年发布的,设计者是Rey Card。ext2是Linux文件系统类型中使用最多的格式。并且在速度和CPU利用率上较突出，是 GNU/Linux 系统中标准的文件系统，其特点为存取文件的性能极好，对于中小型的文件更显示出优势，这主要得利于其簇快取层的优良设计。Ext2 可以支持256字节的长文件名，其单一文件大小与文件系统本身的容量上限与文件系统本身的簇大小有关，在一般常见的Intel x86兼容处理器的系统中，簇最大为 4KB, 则单一文件大小上限为 2048GB, 而文件系统的容量上限为 6384GB。尽管Linux可以支持种类繁多的文件系统，但是2000年以前几乎所有的Linux发行版都用ext2作为默认的文件系统。　   
ext2的缺点：ext2的设计者主要考虑的是文件系统性能方面的问题。ext2在写入文件内容的同时并没有同时写入文件的meta-data（和文件有关的信息，例如：权限、所有者以及创建和访问时间）。换句话说，Linux先写入文件的内容，然后等到有空的时候才写入文件的meta-data。这样若出现写入文件内容之后但在写入文件的meta-data之前系统突然断电，就可能造成在文件系统就会处于不一致的状态。在一个有大量文件操作的系统中出现这种情况会导致很严重的后果。另外但由于目前核心 2.4 所能使用的单一分割区最大只有 2048GB，尽管文件系统的容量上限为 6384G但是实际上能使用的文件系统容量最多也只有 2048GB。   
三、 ext3   
ext3是由开放资源社区开发的日志文件系统，主要开发人员是Stephen tweedie。ext3被设计成是ext2的升级版本，尽可能地方便用户从ext2fs向ext3fs迁移。ext3在ext2的基础上加入了记录元数据的日志功能，努力保持向前和向后的兼容性。这个文件系统被称为ext2的下一个版本。也就是在保有目前 ext2 的格式之下再加上日志功能。ext3是一种日志式文件系统。日志式文件系统的优越性在于：由于文件系统都有快取层参与运作，如不使用时必须将文件系统卸下，以便将快取层的资料写回磁盘中。因此每当系统要关机时，必须将其所有的文件系统全部卸下后才能进行关机。如果在文件系统尚未卸下前就关机 (如停电) 时，下次重开机后会造成文件系统的资料不一致，故这时必须做文件系统的重整工作，将不一致与错误的地方修复。然而，此一重整的工作是相当耗时的，特别是容量大的文件系统，而且也不能百分之百保证所有的资料都不会流失。故这在大型的伺服器上可能会造成问题。   
ext3的缺点：其最大的缺点是没有现代文件系统所具有的能提高文件数据处理速度和解压的高性能，另外使用ext3文件系统时要注意硬盘限额问题,在这个问题解决之前,不推荐在重要的企业应用上采用ext3+disk quota（磁盘配额）。 　   
四、 jsf   
jsf提供了基于日志的字节级文件系统，该文件系统是为面向事务的高性能系统而开发的。jsf（Journaled File System Technology for Linux）的开发者包括AIX（IBM的Unix）的jsf的主要开发者。在AIX上，jfs已经经受住了考验。它是可靠、快速和容易使用的。2000年2月，ibm宣布在一个开放资源许可证下，移植linux版的JSF文件系统。JSFs也是一个有大量用户安装使用的企业级文件系统。它具有可伸缩性和健壮性，与非日志文件系统相比，它的优点是其快速重启能力：Jfs 能够在几秒或几分钟内就把文件系统恢复到一致状态。虽然 jsf 主要是为满足服务器（从单处理器系统到高级多处理器和群集系统）的高吞吐量和可靠性需求而设计的，jsf 还可用于想得到高性能和可靠性的客户机配置因为在系统崩溃时，jsf 能提供快速文件系统重启时间，所以它是因特网文件服务器的关键技术。使用数据库日志处理技术，jsf 能在几秒或几分钟之内把文件系统恢复到一致状态。而在非日志文件系统中，文件恢复可能花费几小时或几天。   
jsf的缺点：使用jsf日志文件系统，性能上会有一定损失，系统资源占用的比率也偏高。是因为当它保持一个日志时，系统需要写许多数据。   
五、ReiserFS   
ReiserFS的第一次公开亮相是在1997年7月23日，Hans Reiser把他的基于平衡树结构的ReiserFS文件系统在网上公布。ReiserFS 3.6.x（作为 Linux 2.4 一部分的版本）是由 Hans Reiser 和他的在Namesys 的开发组共同开发设计的。Hans 和他的组员们相信最好的文件系统是那些能够有助于创建独立的共享环境或者命名空间的文件系统，应用程序可以在其中更直接、有效和有力地相互作用。为了实现这一目标，文件系统就应该满足其使用者对性能和功能方面的需要。那样，使用者就能够继续直接地使用文件系统，而不必建造运行在文件系统之上（如数据库之类）的特殊目的层。ReiserFS 使用了特殊的优化 b\* 平衡树（每个文件系统一个）来组织所有的文件系统数据。这为其自身提供了非常不错的性能改进，也能够减轻文件系统设计上的人为约束。例如，现在一个目录下可以容纳 ext00,000 个子目录。另一个使用 b\* 树的好处就是 ReiserFS 能够像大多其它的下一代文件系统一样，根据需要动态地分配索引节，而不必在文件系统创建时建立固定的索引节。这有助于文件系统更灵活地适应其面临的各种存储需要，同时提供附加的空间有效率。   
Reiserfs被看作是一个更加激进和现代的文件系统。传统的UNIX文件系统是按盘块来进行空间分配的，对于目录和文件等的查找使用了简单的线性查找。这些设计在当时是合适的，但随着磁盘容量的增大和应用需求的增加，传统文件系统在存储效率，速度和功能上已显落后。在reiserfs的下一版reiser4中还提供了对事务的支持。在http://www.namesys.com/v4/v4.html 中有reiser4的介绍和一个简单的reiser4的性能测试。   
ReiserFS的缺点：ReiserFS一个最受人批评的缺点是每升级一个版本，都将要将磁盘重新格式化一次。你可以在http://www.namesys.com/ 网站了解关于 ReiserFS 的更多信息。   
六、Xfs   
xfs是一种非常优秀的日志文件系统，它是SGI公司设计的。xfs被称为业界最先进的、最具可升级性的文件系统技术。它是一个全64位,快速、稳固的日志文件系统，多年用于SGI的IRIX操作系统。sgi决定支持Linux社区，将关键的基本架构技术授权于Linux。它以开放资源形式发布了他们自己拥有的xfs的源代码，并开始进行移植。此工作进展得很快，目前已进入beta版阶段。作为一个64位文件系统，xfs可以支持超大数量的文件（9g×1gb，甚至更大的18g×1gb），可在大型 2d 和 3d 数据方面提供显著的性能。xfs有能力预测其它文件系统薄弱环节，同时xfs提供了在不妨碍性能的情况下增强可靠性和快速的事故恢复。SGI的xfs可为linux和开放资源社区带来的新特性有：可升级性：xfs被设计成可升级，以面对大多数的存储容量和i/o存储需求，可处理大型文件和包含巨大数量文件的大型目录，满足二十一世纪快速增长的磁盘需求。xfs有能力动态地为文件分配索引空间，使系统形成高效支持大数量文件的能力。在它的支持下，用户可使用1exabyte （1g×1gb） 大的文件，远远大于现在最大的文件系统。优秀的i/o 性能：典型的现代服务器使用大型的条带式磁盘阵列，以提供达数gb/秒的总带宽。xfs可以很好地满足I/O请求的大小和并发I/O请求的数量。　xfs可作为root文件系统,并被lilo支持.在NFS服务器上使用也没问题.支持软件磁盘阵列（RAID）和虚拟集群（LVM）。SGI最新发布xfs为 1.0.1版.(在：http://http://oss.sgi.com/projects/xfs/ 可以下载它)。   
xfs的缺点：由于xfs比较复杂，实施起来有一些难度，所以目前xfs主要应用于Linux企业应用的高端。   
其他文件系统简介：   
Minix：Llnux支持的第一个文件系统，对用户有很多限制而且性能低下。有些没有时间标记，其文件名最长l 4个字符。minix 文件系统最大的缺点是最大只能使用64M 的硬盘分区，所以在目前已经没有人使用它了。   
Xia：Minix文件系统修正后的版本。在一定程度上解决了文件名和文件系统大小的局限。但没有新的特色，目前很少有人使用。   
Msdos：msdos 是在Dos、Windows和某些OS/2 操作系统上使用的一种文件系统，其名称采用“8+3”的形式，即8个字符的文件名加上3个字符的扩展名。         
umsdos：  Linux下的扩展msdos文件系统驱动，支持长文件名、所有者、允许权限、连接和设备文件。允许一个普通的msdo s文件系统用于Linux，而且无须为它建立单独的分区。   
iso9660：  标准CDROM文件系统，通用的Rock Ridge增强系统，允许长文件名。   
Vfat:vfat是Windows9x 和　Windows NT/2000下使用的一种Dos文件系统，其在　Dos 文件系统的基础上增加了对长文件名的支持。   
   Nfs：  Sun公司推出的网络文件系统，允许多台计算机之间共享同一文件系统，易于从所有这些计算机上存取文件。   
   Hpfs： High Performance File System（HPFS） 高性能文件系统（HPFS） HPFS是Microsoft的LAN Manager中的文件系统，同时也是IBM的LAN Server和OS／2的文件系统。HPFS能访问较大的硬盘驱动器，提供更多的组织特性并改善了文件系统的安全特性。   
Smb:smb是一种支持 Windows for workgroups、Windows NT 和Lan Manager的基于SMB协议的网络操作系统。   
Sysv: sysv文件系统实际上是System V/Coherent 在Linux平台上的文件系统。   
Ncpfs:ncpfs是一种Novell NetWare 使用的NCP协议的网络操作系统。   
Proc:proc是Linux 系统中作为一种伪文件系统出现的，它用来作为连接内核数据结构的界面。   
NTFS：微软Windows NT内核的系列操作系统支持的、一个特别为网络和磁盘配额、文件加密等管理安全特性设计的磁盘格式。