第4章 ext文件系统机制原理剖析

分类: Linux 基础篇

将磁盘进行分区，分区是将磁盘按柱面进行物理上的划分。划分好分区后还要进行格式化，然后再挂载才能使用(不考虑其他方法)。格式化分区的过程其实就是创建文件系统。

文件系统的类型有很多种，如CentOS 5和CentOS 6上默认使用的ext2/ext3/ext4，CentOS 7上默认使用的xfs，windows上的NTFS，光盘类的文件系统ISO9660，MAC上的混合文件系统HFS，网络文件系统NFS，Oracle研发的btrfs，还有老式的FAT/FAT32等。

本文将非常全面且详细地介绍ext家族的文件系统，中间还非常详细地介绍了inode、软链接、硬链接、数据存储方式以及操作文件的理论，基本上看完本文，对文件系统的宏观理解将再无疑惑。ext家族的文件系统有ext2/ext3/ext4，ext3是有日志的ext2改进版，ext4对相比ext3做了非常多的改进。虽然xfs/btrfs等文件系统有所不同，但它们只是在实现方式上不太同，再加上属于自己的特性而已。

4.1 文件系统的组成部分

4.1.1 block的出现

硬盘最底层的读写IO一次是一个扇区512字节，如果要读写大量文件，以扇区为单位肯定很慢很消耗性能，所以硬盘使用了一个称作逻辑块的概念。逻辑块是逻辑的，由磁盘驱动器负责维护和操作，它并非是像扇区一样物理划分的。一个逻辑块的大小可能包含一个或多个扇区，每个逻辑块都有唯一的地址，称为LBA。有了逻辑块之后，磁盘控制器对数据的操作就以逻辑块为单位，一次读写一个逻辑块，磁盘控制器知道如何将逻辑块翻译成对应的扇区并读写数据。

到了Linux操作系统层次，通过文件系统提供了一个也称为块的读写单元，文件系统数据块的大小一般为1024bytes(1K)或2048bytes(2K)或4096bytes(4K)。文件系统数据块也是逻辑概念，是文件系统层次维护的，而磁盘上的逻辑数据块是由磁盘控制器维护的，文件系统的IO管理器知道如何将它的数据块翻译成磁盘维护的数据块地址LBA。对于使用文件系统的IO操作来说，比如读写文件，这些IO的基本单元是文件系统上的数据块，一次读写一个文件系统数据块。比如需要读一个或多个块时，文件系统的IO管理器首先计算这些文件系统块对应在哪些磁盘数据块，也就是计算出LBA，然后通知磁盘控制器要读取哪些块的数据，硬盘控制器将这些块翻译成扇区地址，然后从扇区中读取数据，再通过硬盘控制器将这些扇区数据重组写入到内存中去。

本文既然是讨论文件系统的，那么重点自然是在文件系统上而不是在磁盘上，所以后文出现的block均表示的是文件系统的数据块而不是磁盘维护的逻辑块。

文件系统block的出现使得在文件系统层面上读写性能大大提高，也大量减少了碎片。但是它的副作用是可能造成空间浪费。由于文件系统以block为读写单元，即使存储的文件只有1K大小也将占用一个block，剩余的空间完全是浪费的。在某些业务需求下可能大量存储小文件，这会浪费大量的空间。

尽管有缺点，但是其优点足够明显，在当下硬盘容量廉价且追求性能的时代，使用block是一定的。

4.1.2 inode的出现

如果存储的1个文件占用了大量的block读取时会如何？假如block大小为1KB，仅仅存储一个10M的文件就需要10240个block，而且这些blocks很可能在位置上是不连续在一起的(不相邻)，读取该文件时难道要从前向后扫描整个文件系统的块，然后找出属于该文件的块吗？显然是不应该这么做的，因为太慢太傻瓜式了。再考虑一下，读取一个只占用1个block的文件，难道只读取一个block就结束了吗？并不是，仍然是扫描整个文件系统的所有block，因为它不知道什么时候扫描到，扫描到了它也不知道这个文件是不是已经完整而不需要再扫描其他的block。

另外，每个文件都有属性(如权限、大小、时间戳等)，这些属性类的元数据存储在哪里呢？难道也和文件的数据部分存储在块中吗？如果一个文件占用多个block那是不是每个属于该文件的block都要存储一份文件元数据？但是如果不在每个block中存储元数据文件系统又怎么知道某一个block是不是属于该文件呢？但是显然，每个数据block中都存储一份元数据太浪费空间。

文件系统设计者当然知道这样的存储方式很不理想，所以需要优化存储方式。如何优化？对于这种类似的问题的解决方法是使用索引，通过扫描索引找到对应的数据，而且索引可以存储部分数据。

在文件系统上索引技术具体化为索引节点(index node)，在索引节点上存储的部分数据即为文件的属性元数据及其他少量信息。一般来说索引占用的空间相比其索引的文件数据而言占用的空间就小得多，扫描它比扫描整个数据要快得多，否则索引就没有存在的意义。这样一来就解决了前面所有的问题。

在文件系统上的术语中，索引节点称为inode。在inode中存储了inode号（注，inode中并未存储inode num，但为了方便理解，这里暂时认为它存储了inode号）、文件类型、权限、文件所有者、大小、时间戳等元数据信息，最重要的是还存储了指向属于该文件block的指针，这样读取inode就可以找到属于该文件的block，进而读取这些block并获得该文件的数据。由于后面还会介绍一种指针，为了方便称呼和区分，暂且将这个inode记录中指向文件data block的指针称之为block指针。以下是ext2文件系统中inode包含的信息示例：

```
Inode: 12   Type: regular    Mode:  0644   Flags: 0x0
Generation: 1454951771    Version: 0x00000000:00000001
User:     0   Group:     0   Size: 5
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x5b628db2:15e0aff4 -- Thu Aug  2 12:50:58 2018
 atime: 0x5b628db2:15e0aff4 -- Thu Aug  2 12:50:58 2018
 mtime: 0x5b628db2:15e0aff4 -- Thu Aug  2 12:50:58 2018
crtime: 0x5b628db2:15e0aff4 -- Thu Aug  2 12:50:58 2018
Size of extra inode fields: 28
BLOCKS:
(0):1024
TOTAL: 1
```

一般inode大小为128字节或256字节，相比那些MB或GB计算的文件数据而言小得多的多，但也要知道可能一个文件大小小于inode大小，例如只占用1个字节的文件。

4.1.3 bmap出现

在向硬盘存储数据时，文件系统需要知道哪些块是空闲的，哪些块是已经占用了的。最笨的方法当然是从前向后扫描，遇到空闲块就存储一部分，继续扫描直到存储完所有数据。

优化的方法当然也可以考虑使用索引，但是仅仅1G的文件系统就有1KB的block共1024*1024=1048576个，这仅仅只是1G，如果是100G、500G甚至更大呢，仅仅使用索引索引的数量和空间占用也将极大，这时就出现更高一级的优化方法：使用块位图(bitmap简称bmap)。

位图只使用0和1标识对应block是空闲还是被占用，0和1在位图中的位置和block的位置一一对应，第一位标识第一个块，第二个位标识第二个块，依次下去直到标记完所有的block。

考虑下为什么块位图更优化。在位图中1个字节8个位，可以标识8个block。对于一个block大小为1KB、容量为1G的文件系统而言，block数量有1024*1024个，所以在位图中使用1024*1024个位共1024*1024/8=131072字节=128K，即1G的文件只需要128个block做位图就能完成一一对应。通过扫描这100多个block就能知道哪些block是空闲的，速度提高了非常多。

但是要注意，bmap的优化针对的是写优化，因为只有写才需要找到空闲block并分配空闲block。对于读而言，只要通过inode找到了block的位置，cpu就能迅速计算出block在物理磁盘上的地址，cpu的计算速度是极快的，计算block地址的时间几乎可以忽略，那么读速度基本认为是受硬盘本身性能的影响而与文件系统无关。大多数稍大一点的文件可能都会存储在不连续的block上，而且使用了一段时间的文件系统可能会有不少碎片，这时硬盘的随机读取性能直接决定读数据的速度，这也是机械硬盘速度相比固态硬盘慢的多的多的原因之一，而且固态硬盘的随机读和连续读取速度几乎是一致的，对它来说，文件系统碎片的多少并不会影响读取速度。

虽然bmap已经极大的优化了扫描，但是仍有其瓶颈：如果文件系统是100G呢？100G的文件系统要使用128*100=12800个1KB大小的block，这就占用了12.5M的空间了。试想完全扫描12800个很可能不连续的block这也是需要占用一些时间的，虽然快但是扛不住每次存储文件都要扫描带来的巨大开销。

所以需要再次优化，如何优化？简而言之就是将文件系统划分开形成块组，至于块组的介绍放在后文。

4.1.4 inode表的出现

回顾下inode相关信息：inode存储了inode号（注，同前文，inode中并未存储inode num）、文件属性元数据、指向文件占用的block的指针；每一个inode占用128字节或256字节。

现在又出现问题了，一个文件系统中可以说有无数多个文件，每一个文件都对应一个inode，难道每一个仅128字节的inode都要单独占用一个block进行存储吗？这太浪费空间了。

所以更优的方法是将多个inode合并存储在block中，对于128字节的inode，一个block存储8个inode，对于256字节的inode，一个block存储4个inode。这就使得每个存储inode的块都不浪费。

在ext文件系统上，将这些物理上存储inode的block组合起来，在逻辑上形成一张inode表(inode table)来记录所有的inode。

举个例子，每一个家庭都要向派出所登记户口信息，通过户口本可以知道家庭住址，而每个镇或街道的派出所将本镇或本街道的所有户口整合在一起，要查找某一户地址时，在派出所就能快速查找到。inode table就是这里的派出所。它的内容如下图所示。

Inode Number	File Type	Permission	Link count	UID	GID	size	...	pointer
1	-	644	1	500	500			
2	d	755	1	0	0			
...	...	...	...	...	...	...	...	...
n	-	644	2	501	501			

再细细一思考，就能发现一个大的文件系统仍将占用大量的块来存储inode，想要找到其中的一个inode记录也需要不小的开销，尽管它们已经形成了一张逻辑上的表，但扛不住表太大记录太多。那么如何快速找到inode，这同样是需要优化的，优化的方法是将文件系统的block进行分组划分，每个组中都存有本组inode table范围、bmap等。

4.1.5 imap的出现

前面说bmap是块位图，用于标识文件系统中哪些block是空闲哪些block是占用的。

对于inode也一样，在存储文件(Linux中一切皆文件)时需要为其分配一个inode号。但是在格式化创建文件系统后所有的inode号都已被事先计算好（创建文件系统时会为每个块组计算好该块组拥有哪些inode号），因此产生了问题：要为文件分配哪一个inode号呢？又如何知道某一个inode号是否已经被分配了呢？

既然是"是否被占用"的问题，使用位图是最佳方案，像bmap记录block的占用情况一样。标识inode号是否被分配的位图称为inodemap简称为imap。这时要为一个文件分配inode号只需扫描imap即可知道哪一个inode号是空闲的。

imap存在着和bmap和inode table一样需要解决的问题：如果文件系统比较大，imap本身就会很大，每次存储文件都要进行扫描，会导致效率不够高。同样，优化的方式是将文件系统占用的block划分成块组，每个块组有自己的imap范围。

4.1.6 块组的出现

前面一直提到的优化方法是将文件系统占用的block划分成块组(block group)，解决bmap、inode table和imap太大的问题。

在物理层面上的划分是将磁盘按柱面划分为多个分区，即多个文件系统；在逻辑层面上的划分是将文件系统划分成块组。每个文件系统包含多个块组，每个块组包含多个元数据区和数据区：元数据区就是存储bmap、inode table、imap等的数据；数据区就是存储文件数据的区域。注意块组是逻辑层面的概念，所以并不会真的在磁盘上按柱面、按扇区、按磁道等概念进行划分。

4.1.7 块组的划分

块组在文件系统创建完成后就已经划分完成了，也就是说元数据区bmap、inode table和imap等信息占用的block以及数据区占用的block都已经划分好了。那么文件系统如何知道一个块组元数据区包含多少个block，数据区又包含多少block呢？

它只需确定一个数据——每个block的大小，再根据bmap至多只能占用一个完整的block的标准就能计算出块组如何划分。如果文件系统非常小，所有的bmap总共都不能占用完一个block，那么也只能空闲bmap的block了。

每个block的大小在创建文件系统时可以人为指定，不指定也有默认值。

假如现在block的大小是1KB，一个bmap完整占用一个block能标识1024*8= 8192个block(当然这8192个block是数据区和元数据区共8192个，因为元数据区分配的block也需要通过bmap来标识)。每个block是1K，每个块组是8192K即8M，创建1G的文件系统需要划分1024/8=128个块组，如果是1.1G的文件系统呢？128+12.8=128+13=141个块组。

每个组的block数目是划分好了，但是每个组设定多少个inode号呢？inode table占用多少block呢？这需要由系统决定了，因为描述"每多少个数据区的block就为其分配一个inode号"的指标默认是我们不知道的，当然创建文件系统时也可以人为指定这个指标或者百分比例。见后文"inode深入"。

使用dumpe2fs可以将ext类的文件系统信息全部显示出来，当然bmap是每个块组固定一个block的不用显示，imap比bmap更小所以也只占用1个block不用显示。

下图是一个文件系统的部分信息，在这些信息的后面还有每个块组的信息，其实这里面的很多信息都可以通过几个比较基本的元数据推导出来。

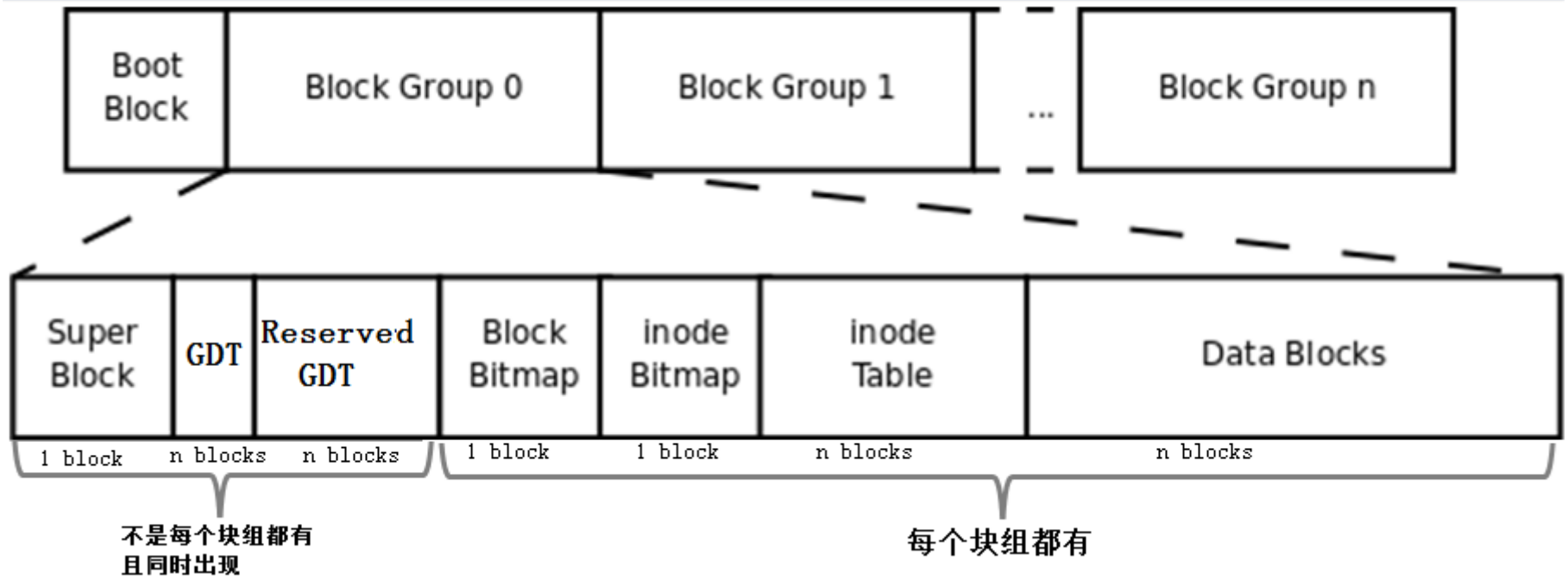
从这张表中能计算出文件系统的大小，该文件系统共4667136个blocks，每个block大小为4K，所以文件系统大小为4667136*4/1024/1024=17.8GB。

也能计算出分了多少个块组，因为每一个块组的block数量为32768，所以块组的数量为4667136/32768=142.4即143个块组。由于块组从0开始编号，所以最后一个块组编号为Group 142。如下图所示是最后一个块组的信息。

```
Group 142: (Blocks 4653056-4667135) [ITABLE_ZEROED]
  Checksum 0x9401, unused inodes 0
  Block bitmap at 4194318 (+4294508558), Inode bitmap
  Inode table at 4201476-4201985 (+4294515716)
  14080 free blocks, 8160 free inodes, 0 directories
  Free blocks: 4653056-4667135
  Free inodes: 1158721-1166880
```

4.2 文件系统的完整结构

将上文描述的bmap、inode table、imap、数据区的blocks和块组的概念组合起来就形成了一个文件系统，当然这还不是完整的文件系统。完整的文件系统如下图



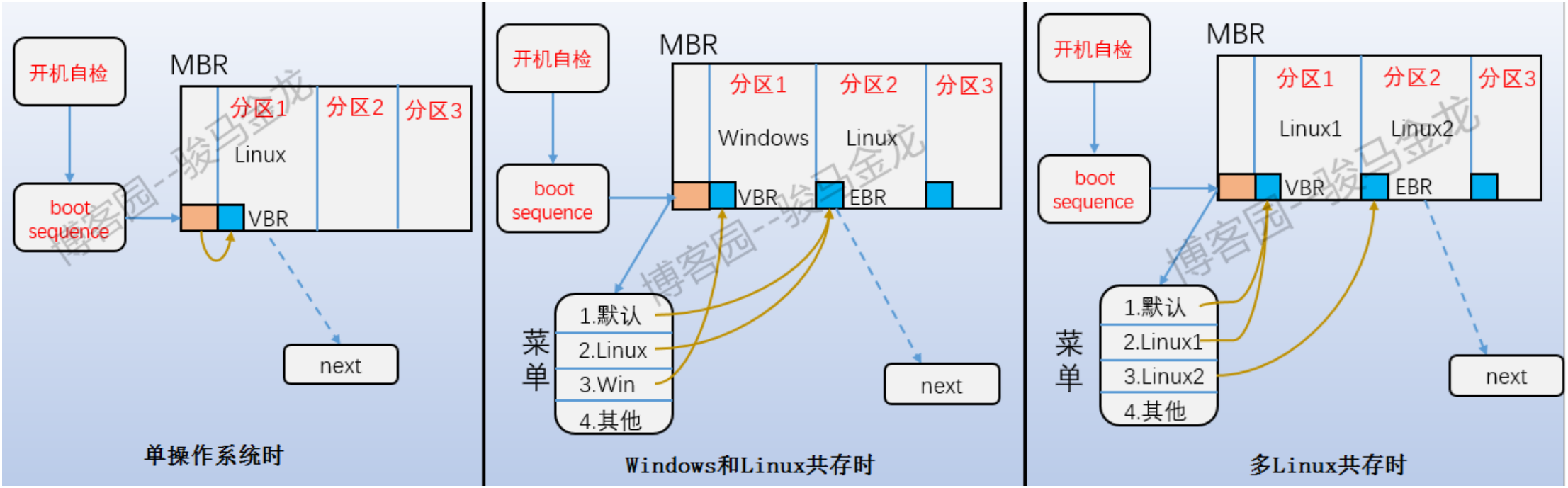
首先，该图中多了Boot Block、Super Block、GDT、Reserver GDT这几个概念。下面会分别介绍它们。

然后，图中指明了块组中每个部分占用的block数量，除了superblock、bmap、imap能确定占用1个block，其他的部分都不能确定占用几个block。

最后，图中指明了Superblock、GDT和Reserved GDT是同时出现且不一定存在于每一个块组中的，也指明了bmap、imap、inode table和data blocks是每个块组都有的。

4.2.1 引导块

即上图中的Boot Block部分，也称为boot sector。它位于分区上的第一个块，占用1024字节，并非所有分区都有这个boot sector，只有装了操作系统的主分区和装了操作系统的逻辑分区才有。里面存放的也是boot loader，这段boot loader称为VBR(主分区装操作系统时)或EBR(扩展分区装操作系统时)，这里的Boot loader和mbr上的boot loader是存在交错关系的。开机启动的时候，首先加载mbr中的bootloader，然后定位到操作系统所在分区的boot serctor上加载此处的boot loader。如果是多系统，加载mbr中的bootloader后会列出操作系统菜单，菜单上的各操作系统指向它们所在分区的boot sector上。它们之间的关系如下图所示。



但是，这种方式的操作系统菜单早已经弃之不用了，而是使用grub来管理启动菜单。尽管如此，在安装操作系统时，仍然有一步是选择boot loader安装位置的步骤。

4.2.2 超级块(superblock)

既然一个文件系统会分多个块组，那么文件系统怎么知道分了多少个块组呢？每个块组又有多少block多少inode号等等信息呢？还有，文件系统本身的属性信息如各种时间戳、block总数量和空闲数量、inode总数量和空闲数量、当前文件系统是否正常、什么时候需要自检等等，它们又存储在哪里呢？

毫无疑问，这些信息必须要存储在block中。存储这些信息占用1024字节，所以也要一个block，这个block称为超级块(superblock)，它的block号可能为0也可能为1。如果block大小为1K，则引导块正好占用一个block，这个block号为0，所以superblock的号为1；如果block大小大于1K，则引导块和超级块同置在一个block中，这个block号为0。总之superblock的起止位置是第二个1024(1024-2047)字节。

使用df命令读取的就是每个文件系统的superblock，所以它的统计速度非常快。相反，用du命令查看一个较大目录的已用空间就非常慢，因为不可避免地要遍历整个目录的所有文件。

```
[root@xuexi ~]# df -hT
Filesystem     Type    Size  Used Avail Use% Mounted on
/dev/sda3      ext4     18G  1.7G   15G  11% /
tmpfs          tmpfs   491M     0  491M   0% /dev/shm
/dev/sda1      ext4    190M   32M  149M  18% /boot
```

superblock对于文件系统而言是至关重要的，超级块丢失或损坏必将导致文件系统的损坏。所以旧式的文件系统将超级块备份到每一个块组中，但是这又有所空间浪费，所以ext2文件系统只在块组0、1和3、5、7幂次方的块组中保存超级块的信息，如Group9、Group25等。尽管保存了这么多的superblock，但是文件系统只使用第一个块组即Group0中超级块信息来获取文件系统属性，只有当Group0上的superblock损坏或丢失才会找下一个备份超级块复制到Group0中来恢复文件系统。

下图是一个ext4文件系统的superblock的信息，ext家族的文件系统都能使用dumpe2fs -h获取。

```
Filesystem volume name:   <none>
Last mounted on:          /
Filesystem UUID:          e1b90643-6b1f-4e0b-ae41-049765784435
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype
needs_recovery extent flex_bg sparse_super large_file huge_file uninit_bg dir_nlink
extra_isize
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              1166880
Block count:              4667136
Reserved block count:     233356
Free blocks:              3963754
Free inodes:              1106146
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      1022
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         8160
Inode blocks per group:   510
Flex block group size:    16
Filesystem created:       Thu Feb 18 20:27:46 2016
Last mount time:          Fri Oct  7 21:26:58 2016
Last write time:          Thu Feb 18 20:59:06 2016
Mount count:              20
Maximum mount count:      -1
Last checked:             Thu Feb 18 20:27:46 2016
Check interval:           0 (<none>)
Lifetime writes:          4112 MB
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               256
Required extra isize:     28
Desired extra isize:      28
Journal inode:            8
Default directory hash:   half_md4
Directory Hash Seed:      c9b3ad52-775e-491f-82ff-c808c5373893
Journal backup:           inode blocks
Journal features:         journal_incompat_revoke
Journal size:             128M
Journal length:           32768
Journal sequence:         0x00004e39
Journal start:            13134
```

4.2.3 块组描述符表(GDT)

既然文件系统划分了块组，那么每个块组的信息和属性元数据又保存在哪里呢?

ext文件系统每一个块组信息使用32字节描述，这32个字节称为块组描述符，所有块组的块组描述符组成块组描述符表GDT(group descriptor table)。

虽然每个块组都需要块组描述符来记录块组的信息和属性元数据，但是不是每个块组中都存放了块组描述符。ext文件系统的存储方式是：将它们组成一个GDT，并将该GDT存放于某些块组中，存放GDT的块组和存放superblock和备份superblock的块相同，也就是说它们是同时出现在某一个块组中的。读取时也总是读取Group0中的块组描述符表信息。

假如block大小为4KB的文件系统划分了143个块组，每个块组描述符32字节，那么GDT就需要143*32=4576字节即两个block来存放。这两个GDT block中记录了所有块组的块组信息，且存放GDT的块组中的GDT都是完全相同的。

下图是一个块组描述符的信息(通过dumpe2fs获取)。

```
Group 142: (Blocks 4653056-4667135) [ITABLE_ZEROED]
  Checksum 0x9401, unused inodes 0
  Block bitmap at 4194318 (+4294508558), Inode bitmap
  Inode table at 4201476-4201985 (+4294515716)
  14080 free blocks, 8160 free inodes, 0 directories
  Free blocks: 4653056-4667135
  Free inodes: 1158721-1166880
```

4.2.4 保留GDT(Reserved GDT)

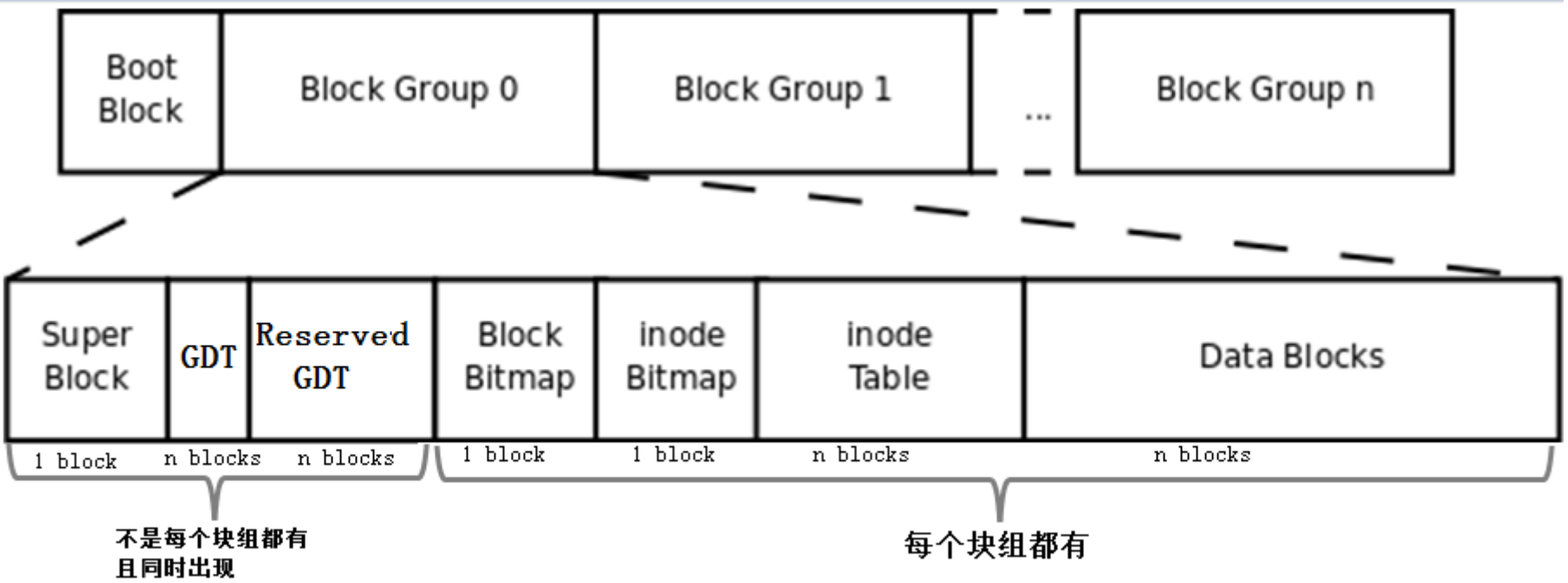
保留GDT用于以后扩容文件系统使用，防止扩容后块组太多，使得块组描述符超出当前存储GDT的blocks。保留GDT和GDT总是同时出现，当然也就和superblock同时出现了。

例如前面143个块组使用了2个block来存放GDT，但是此时第二个block还空余很多空间，当扩容到一定程度时2个block已经无法再记录块组描述符了，这时就需要分配一个或多个Reserved GDT的block来存放超出的块组描述符。

由于新增加了GDT block，所以应该让每一个保存GDT的块组都同时增加这一个GDT block，所以将保留GDT和GDT存放在同一个块组中可以直接将保留GDT变换为GDT而无需使用低效的复制手段备份到每个存放GDT的块组。

同理，新增加了GDT需要修改每个块组中superblock中的文件系统属性，所以将superblock和Reserved GDT/GDT放在一起又能提升效率。

4.3 Data Block



如上图，除了Data Blocks其他的部分都解释过了。data block是直接存储数据的block，但事实上并非如此简单。

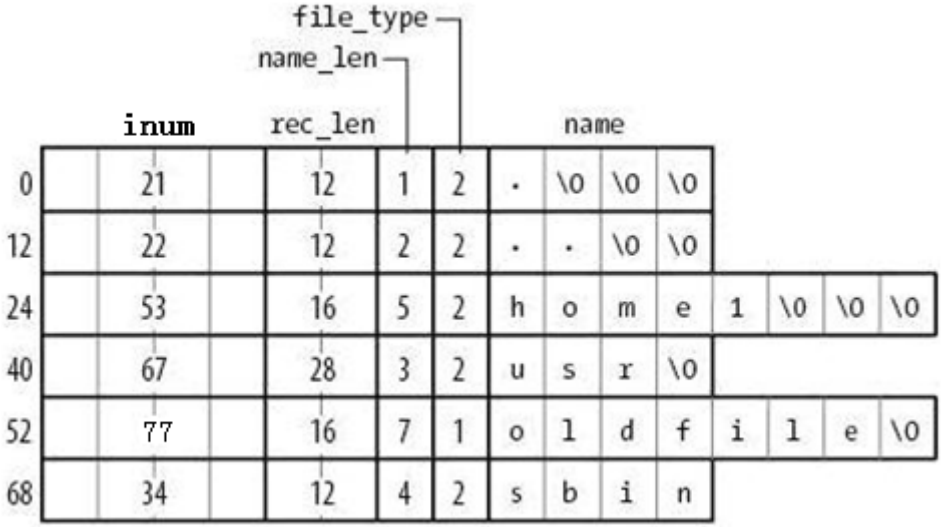
数据所占用的block由文件对应inode记录中的block指针找到，不同的文件类型，数据block中存储的内容是不一样的。以下是Linux中不同类型文件的存储方式。

- 对于常规文件，文件的数据正常存储在数据块中。
- 对于目录，该目录下的所有文件和一级子目录的目录名存储在数据块中。
  - **文件名和inode号不是存储在其自身的inode中，而是存储在其所在目录的data block中。**
- 对于符号链接，如果目标路径名较短则直接保存在inode中以便更快地查找，如果目标路径名较长则分配一个数据块来保存。
- 设备文件、FIFO和socket等特殊文件没有数据块，设备文件的主设备号和次设备号保存在inode中。

常规文件的存储就不解释了，下面分别解释特殊文件的存储方式。

4.3.1 目录文件的data block

目录的data block的内容如下图所示。



由图可知，在目录文件的数据块中存储了其下的文件名、目录名、目录本身的相对名称"."和上级目录的相对名称"..", 还存储了这些文件名对应的inode号、目录项长度rec_len、文件名长度name_len和文件类型file_type。注意到除了文件本身的inode记录了文件类型，其所在的目录的数据块也记录了文件类型。由于rec_len只能是4的倍数，所以需要使用"\0"来填充name_len不够凑满4倍数的部分。至于rec_len具体是什么，只需知道它是一种偏移即可。

需要注意的是，inode table中的inode自身并没有存储每个inode的inode号，它是存储在目录的data block中的，通过inode号可以计算并索引到inode table中该inode号对应的inode记录，可以认为这个inode号是一个inode指针 (当然，并非真的是指针，但有助于理解通过inode号索引找到对应inode的这个过程，后文将在需要的时候使用inode指针这个词来表示inode号。至此，已经知道了两种指针：一种是inode table中每个inode记录指向其对应data block的block指针，一个此处的“inode指针”)。

除了inode号，目录的data block中还使用数字格式记录了文件类型，数字格式和文件类型的对应关系如下图。

编码	文件类型
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

注意到目录的data block中前两行存储的是目录本身的相对名称"."和上级目录的相对名称"..", 它们实际上是目录本身的硬链接和上级目录的硬链接。硬链接的本质后面说明。

4.3.2 如何根据inode号找到inode

前面提到过，inode结构自身并没有保存inode号（同样，也没有保存文件名），那么inode号保存在哪里呢？目录的data block中保存了该目录中每个文件的inode号。

另一个问题，既然inode中没有inode号，那么如何根据目录data block中的inode号找到inode table中对应的inode呢？

实际上，只要有了inode号，就可以计算出inode表中对应该inode号的inode结构。在创建文件系统的时候，每个块组中的起始inode号以及inode table的起始地址都已经确定了，所以只要知道inode号，就能知道这个inode号和该块组起始inode号的偏移数量，再根据每个inode结构的大小(256字节或其它大小)，就能计算出来对应的inode结构。

所以，目录的data block中的inode number和inode table中的inode是通过计算的方式一一映射起来的。从另一个角度上看，目录data block中的inode number是找到inode table中对应inode记录的唯一方式。

考虑一种比较特殊的情况：目录data block的记录已经删除，但是该记录对应的inode结构仍然存在于inode table中。这种inode称为孤儿inode（orphan inode）：存在于inode table中，但却无法再索引到它。因为目录中已经没有该inode对应的文件记录了，所以其它进程将无法找到该inode，也就无法根据该inode找到该文件之前所占用的data block，这正是创建便删除所实现的真正临时文件，该临时文件只有当前进程和子进程才能访问。

4.3.3 符号链接存储方式

符号链接即为软链接，类似于Windows操作系统中的快捷方式，它的作用是指向原文件或目录。

软链接之所以也被称为特殊文件的原因是：它一般情况下不占用data block，仅仅通过它对应的inode记录就能将其信息描述完成；符号链接的大小是其指向目标路径占用的字符个数，例如某个符号链接的指向方式为"rmt --> ../sbin/rmt"，则其文件大小为11字节；只有当符号链接指向的目标的路径名较长(60个字节)时文件系统才会划分一个data block给它；它的权限如何也不重要，因它只是一个指向原文件的"工具"，最终决定是否能读写执行的权限由原文件决定，所以很可能ls -l查看到的符号链接权限为777。

注意，软链接的block指针存储的是目标文件名。也就是说，链接文件的一切都依赖于其目标文件名。这就解释了为什么/mnt的软链接/tmp/mnt在/mnt挂载文件系统后，通过软链接就能进入/mnt所挂载的文件系统。究其原因，还是因为其目标文件名"/mnt"并没有改变。

例如以下筛选出了/etc/下的符号链接，注意观察它们的权限和它们占用的空间大小。

```
[root@xuexi ~]# ll /etc/ | grep '^l'
lrwxrwxrwx.  1 root root     56 Feb 18  2016 favicon.png -> /usr/share/icons/hicolor/16x16/apps/system-logo-icon.png
lrwxrwxrwx.  1 root root     22 Feb 18  2016 grub.conf -> ../boot/grub/grub.conf
lrwxrwxrwx.  1 root root     11 Feb 18  2016 init.d -> rc.d/init.d
lrwxrwxrwx.  1 root root      7 Feb 18  2016 rc -> rc.d/rc
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc0.d -> rc.d/rc0.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc1.d -> rc.d/rc1.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc2.d -> rc.d/rc2.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc3.d -> rc.d/rc3.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc4.d -> rc.d/rc4.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc5.d -> rc.d/rc5.d
lrwxrwxrwx.  1 root root     10 Feb 18  2016 rc6.d -> rc.d/rc6.d
lrwxrwxrwx.  1 root root     13 Feb 18  2016 rc.local -> rc.d/rc.local
lrwxrwxrwx.  1 root root     15 Feb 18  2016 rc.sysinit -> rc.d/rc.sysinit
lrwxrwxrwx.  1 root root     14 Feb 18  2016 redhat-release -> centos-release
lrwxrwxrwx.  1 root root     11 Apr 10  2016 rmt -> ../sbin/rmt
lrwxrwxrwx.  1 root root     14 Feb 18  2016 system-release -> centos-release
```

4.3.4 设备文件、FIFO、套接字文件

关于这3种文件类型的文件只需要通过inode就能完全保存它们的信息，它们不占用任何数据块，所以它们是特殊文件。

设备文件的主设备号和次设备号也保存在inode中。以下是/dev/下的部分设备信息。注意到它们的第5列和第6列信息，它们分别是主设备号和次设备号，主设备号标识每一种设备的类型，次设备号标识同种设备类型的不同编号；也注意到这些信息中没有大小的信息，因为设备文件不占用数据块所以没有大小的概念。

```
[root@xuexi ~]# ll /dev | tail
crw-rw---- 1 vcsa tty       7, 129 Oct  7 21:26 vcsa1
crw-rw---- 1 vcsa tty       7, 130 Oct  7 21:27 vcsa2
crw-rw---- 1 vcsa tty       7, 131 Oct  7 21:27 vcsa3
crw-rw---- 1 vcsa tty       7, 132 Oct  7 21:27 vcsa4
crw-rw---- 1 vcsa tty       7, 133 Oct  7 21:27 vcsa5
crw-rw---- 1 vcsa tty       7, 134 Oct  7 21:27 vcsa6
crw-rw---- 1 root root     10,  63 Oct  7 21:26 vga_arbiter
crw------- 1 root root     10,  57 Oct  7 21:26 vmci
crw-rw-rw- 1 root root     10,  56 Oct  7 21:27 vsock
crw-rw-rw- 1 root root      1,   5 Oct  7 21:26 zero
```

4.4 inode基础知识

每个文件都有一个inode，在将inode关联到文件后系统将通过inode号来识别文件，而不是文件名。并且访问文件时将先找到inode，通过inode中记录的block位置找到该文件。

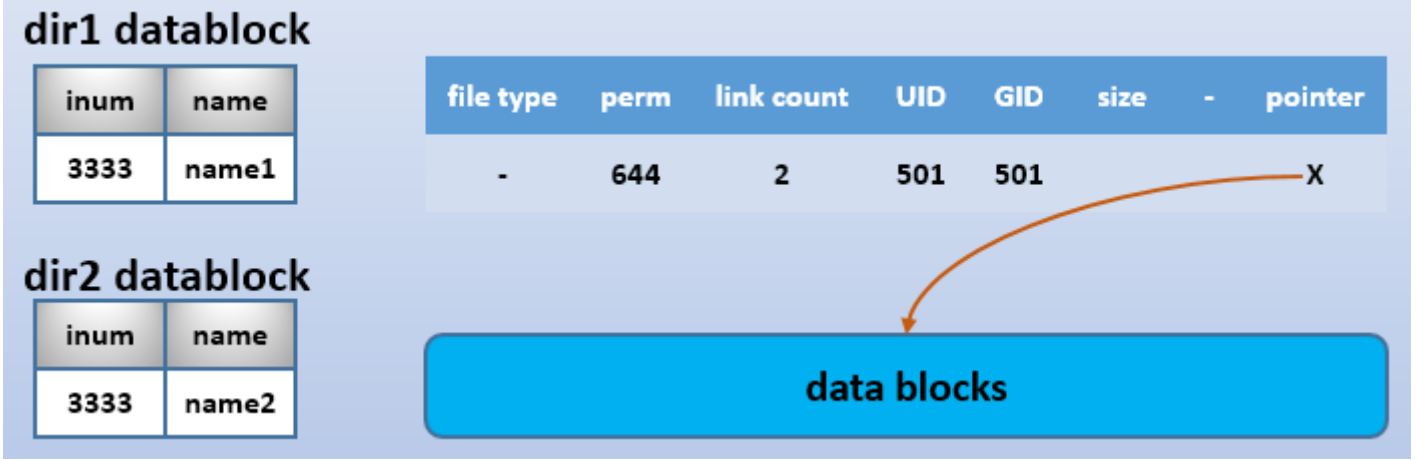
4.4.1 硬链接

虽然每个文件都有一个inode，但是存在一种可能：多个文件的inode相同，也就即inode号、元数据、block位置都相同，这是一种什么样的情况呢？能够想象这些inode相同的文件使用的都是同一条inode记录，所以代表的都是同一个文件，这些文件所在目录的data block中的inode号都是一样的，只不过各inode号对应的文件名互不相同而已。这种inode相同的文件在Linux中被称为"硬链接"。

硬链接文件的inode都相同，每个文件都有一个"硬链接数"的属性，使用ls -l的第二列就是被硬链接数，它表示的就是该文件有几个硬链接。

```
[root@xuexi ~]# ls -l
total 48
drwxr-xr-x  5 root root  4096 Oct 15 18:07 700
-rw-------. 1 root root  1082 Feb 18  2016 anaconda-ks.cfg
-rw-r--r--  1 root root   399 Apr 29  2016 Identity.pub
-rw-r--r--. 1 root root 21783 Feb 18  2016 install.log
-rw-r--r--. 1 root root  6240 Feb 18  2016 install.log.syslog
```

例如下图描述的是dir1目录中的文件name1及其硬链接dir2/name2，右边分别是它们的inode和datablock。这里也看出了硬链接文件之间唯一不同的就是其所在目录中的记录不同。注意下图中有一列Link Count就是标记硬链接数的属性。



每创建一个文件的硬链接，实质上是多一个指向该inode记录的inode指针，并且硬链接数加1。

删除文件的实质是删除该文件所在目录data block中的对应的inode行，所以也是减少硬链接次数，由于block指针是存储在inode中的，所以不是真的删除数据，如果仍有其他inode号链接到该inode，那么该文件的block指针仍然是可用的。当硬链接次数为1时再删除文件就是真的删除文件了，此时inode记录中block指针也将被删除。

不能跨分区创建硬链接，因为不同文件系统的inode号可能会相同，如果允许创建硬链接，复制到另一个分区时inode可能会和此分区已使用的inode号冲突。

硬链接只能对文件创建，无法对目录创建硬链接。之所以无法对目录创建硬链接，是因为文件系统已经把每个目录的硬链接创建好了，它们就是相对路径中的"."和".."，分别标识当前目录的硬链接和上级目录的硬链接。每一个目录中都会包含这两个硬链接，它包含了两个信息：(1)一个没有子目录的目录文件的硬链接数是2，其一是目录本身，即该目录datablock中的"."，其二是其父目录datablock中该目录的记录，这两者都指向同一个inode号；(2)一个包含子目录的目录文件，其硬链接数是2+子目录数，因为每个子目录都关联一个父目录的硬链接".."。很多人在计算目录的硬链接数时认为由于包含了"."和"..."，所以空目录的硬链接数是2，这是错误的，因为".."不是本目录的硬链接。另外，还有一个特殊的目录应该纳入考虑，即"/"目录，它自身是一个文件系统的入口，是自引用(下文中会解释自引用)的，所以"/"目录下的"."和".."的inode号相同，它自身不占用硬链接，因为其datablock中只记录inode号相同的"."和"..."，不再像其他目录一样还记录一个名为"/"的目录，所以"/"的硬链接数也是2+子目录数，但这个2是"."和".."的结果。

```
[root@xuexi ~]# ln /tmp /mydata
ln: `/tmp': hard link not allowed for directory
```

为什么文件系统自己创建好了目录的硬链接就不允许人为创建呢？从"."和".."的用法上考虑，如果当前目录为/usr，我们可以使用"./local"来表示/usr/local，但是如果我们人为创建了/usr目录的硬链接/tmp/husr，难道我们也要使用"/tmp/husr/local"来表示/usr/local吗？这其实已经是软链接的作用了。若要将其认为是硬链接的功能，这必将导致硬链接维护的混乱。

不过，通过mount工具的"--bind"选项，可以将一个目录挂载到另一个目录下，实现伪"硬链接"，它们的内容和inode号是完全相同的。

硬链接的创建方法：`ln file_target link_name`。

4.4.2 软链接

软链接就是字符链接，链接文件默认指的就是字符链接文件(注意不是字符设备)，使用"l"表示其类型。

硬链接不能跨文件系统创建，否则inode号可能会冲突。于是实现了软链接以便跨文件系统建立链接。既然是跨文件系统，那么软链接必须得有自己的inode号。

软链接在功能上等价与Windows系统中的快捷方式，它指向原文件，原文件损坏或消失，软链接文件就损坏。可以认为软链接inode记录中的指针内容是目标路径的字符串。

创建方式：`ln -s source_file softlink_name`，记住是source_file<--link_name的指向关系(反箭头)，以前我老搞错位置。

查看软链接的值：`readlink softlink_name`

在设置软链接的时候，source_file虽然不要求是绝对路径，但建议给绝对路径。是否还记得软链接文件的大小？它是根据软链接所指向路径的字符数计算的，例如某个符号链接的指向方式为"rmt --> ../sbin/rmt"，它的文件大小为11字节，也就是说只要建立了软链接后，软链接的指向路径是不会改变的，仍然是"../sbin/rmt"。如果此时移动软链接文件本身，它的指向是不会改变的，仍然是11个字符的"../sbin/rmt"，但此时该软链接父目录下可能根本就不存在/sbin/rmt，也就是说此时该软链接是一个被破坏的软链接。

4.5 inode深入

4.5.1 inode大小和划分

inode大小为128字节的倍数，最小为128字节。它有默认值大小，它的默认值由/etc/mke2fs.conf文件中指定。不同的文件系统默认值可能不同。

```
[root@xuexi ~]# cat /etc/mke2fs.conf
[defaults]
        base_features = sparse_super,filetype,resize_inode,dir_index,ext_attr
        enable_periodic_fsck = 1
        blocksize = 4096
        inode_size = 256
        inode_ratio = 16384

[fs_types]
        ext3 = {
                features = has_journal
        }
        ext4 = {
                features = has_journal,extent,huge_file,flex_bg,uninit_bg,dir_nlink,extra_isize
                inode_size = 256
        }
```

同样观察到这个文件中还记录了blocksize的默认值和inode分配比率inode_ratio。inode_ratio=16384表示每16384个字节即16KB就分配一个inode号，由于默认blocksize=4KB，所以每4个block就分配一个inode号。当然分配的这些inode号只是预分配，并不真的代表会全部使用，毕竟每个文件才会分配一个inode号。但是分配的inode自身会占用block，而且其自身大小256字节还不算小，所以inode号的浪费代表着空间的浪费。

既然知道了inode分配比率，就能计算出每个块组分配多少个inode号，也就能计算出inode table占用多少个block。

如果文件系统中大量存储电影等大文件，inode号就浪费很多，inode占用的空间也浪费很多。但是没办法，文件系统又不知道你这个文件系统是用来存什么样的数据，多大的数据，多少数据。

当然inode size、inode分配比例、block size都可以在创建文件系统的时候人为指定。

4.5.2 ext文件系统预留的inode号

Ext预留了一些inode做特殊特性使用，如下：某些可能并非总是准确，具体的inode号对应什么文件可以使用"find / -inum NUM"查看。

- Ext4的特殊inode
- Inode号　用途
- 0　不存在0号inode，可用于标识目录data block中已删除的文件
- 1　虚拟文件系统，如/proc和/sys
- **2　根目录**
- 3　ACL索引
- 4　ACL数据
- 5　Boot loader
- 6　未删除的目录
- 7　预留的块组描述符inode
- 8　日志inode
- 11　第一个非预留的inode，通常是lost+found目录

所以在ext4文件系统的dumpe2fs信息中，能观察到fisrt inode号可能为11也可能为12。

并且注意到"/"的inode号为2，这个特性在文件访问时会用上。

需要注意的是，每个文件系统都会分配自己的inode号，不同文件系统之间是可能会出现使用相同inode号文件的。例如：

```
[root@xuexi ~]# find / -ignore_readdir_race -inum 2 -ls
     2     4 dr-xr-xr-x  22 root     root         4096 Jun  9 09:56 /
     2     2 dr-xr-xr-x   5 root     root         1024 Feb 25 11:53 /boot
     2     0 c---------   1 root     root              Jun  7 02:13 /dev/pts/ptmx
     2     0 -rw-r--r--   1 root     root            0 Jun  6 18:13 /proc/sys/fs/binfmt_misc/status
     2     0 drwxr-xr-x   3 root     root            0 Jun  6 18:13 /sys/fs
```

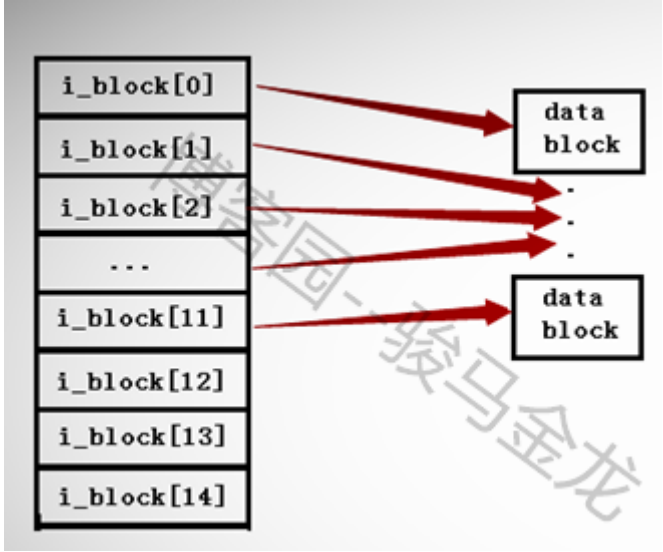
从结果中可见，除了根的Inode号为2，还有几个文件的inode号也是2，它们都属于独立的文件系统，有些是虚拟文件系统，如/proc和/sys。

4.5.3 ext2/3的inode直接、间接寻址

前文说过，inode中保存了blocks指针，但是一条inode记录中能保存的指针数量是有限的，否则就会超出inode大小(128字节或256字节)。

在ext2和ext3文件系统中，一个inode中最多只能有15个指针，每个指针使用i_block[n]表示。

前12个指针i_block[0]到i_block[11]是直接寻址指针，每个指针指向一个数据区的block。如下图所示。



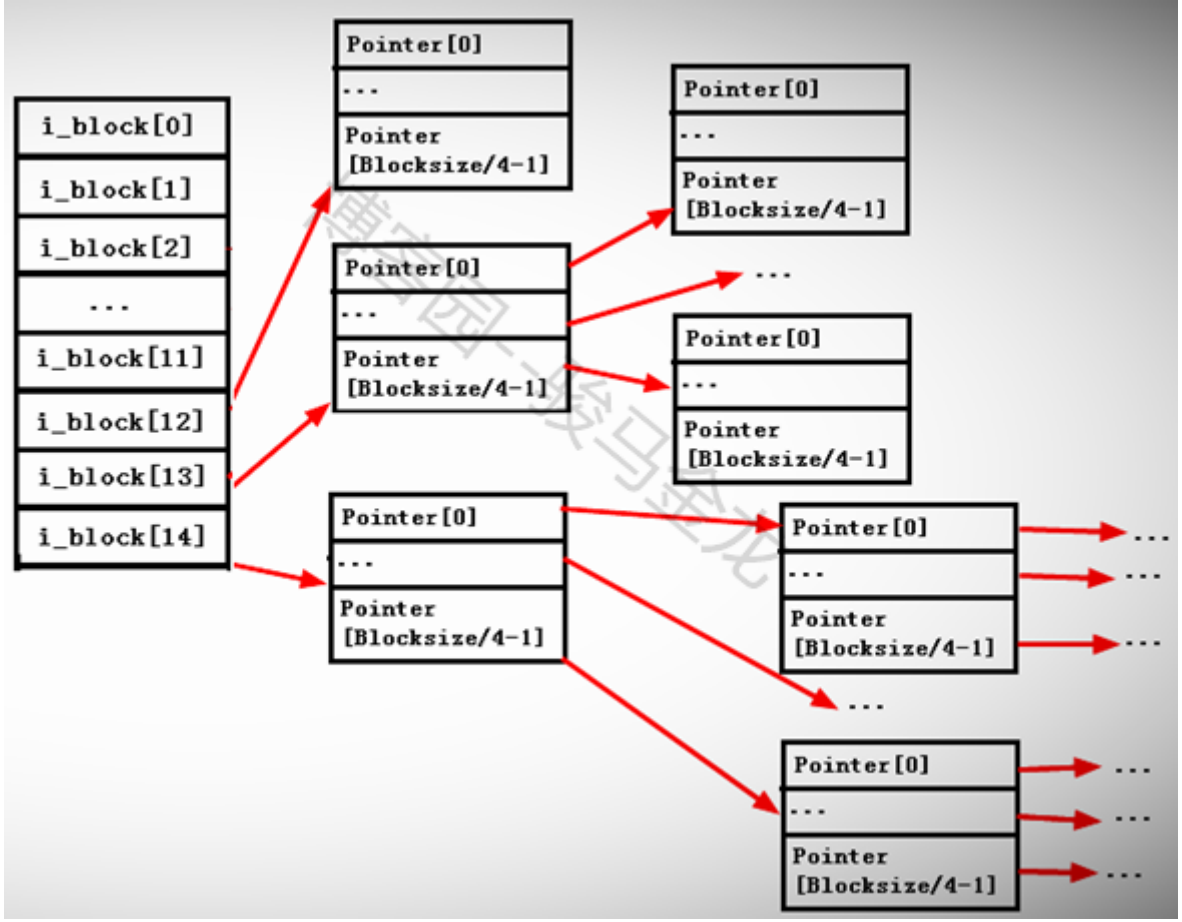
第13个指针i_block[12]是一级间接寻址指针，它指向一个仍然存储了指针的block即i_block[12] --> Pointerblock --> datablock。

第14个指针i_block[13]是二级间接寻址指针，它指向一个仍然存储了指针的block，但是这个block中的指针还继续指向其他存储指针的block，即i_block[13] --> Pointerblock1 --> PointerBlock2 --> datablock。

第15个指针i_block[14]是三级间接寻址指针，它指向一个任然存储了指针的block，这个指针block下还有两次指针指向。即i_block[13] --> Pointerblock1 --> PointerBlock2 --> PointerBlock3 --> datablock。

其中由于每个指针大小为4字节，所以每个指针block能存放的指针数量为BlockSize/4byte。例如blocksize为4KB，那么一个Block可以存放4096/4=1024个指针。

如下图。



为什么要分间接和直接指针呢？如果一个inode中15个指针全是直接指针，假如每个block的大小为1KB，那么15个指针只能指向15个block即15KB的大小，由于每个文件对应一个inode号，所以就限制了每个文件最大为15*1=15KB，这显然是不合理的。

如果存储大于15KB的文件而又不太大的时候，就占用一级间接指针i_block[12]，这时可以存放指针数量为1024/4+12=268，所以能存放268KB的文件。

如果存储大于268K的文件而又不太大的时候，就继续占用二级指针i_block[13]，这时可以存放指针数量为[1024/4]^2+1024/4+12=65804，所以能存放65804KB=64M左右的文件。

如果存放的文件大于64M，那么就继续使用三级间接指针i_block[14]，存放的指针数量为[1024/4]^3+[1024/4]^2+[1024/4]+12=16843020个指针，所以能存放16843020KB=16GB左右的文件。

如果blocksize=4KB呢？那么最大能存放的文件大小为([4096/4]^3+[4096/4]^2+[4096/4]+12)*4/1024/1024/1024=4T左右。

当然这样计算出来的不一定就是最大能存放的文件大小，它还受到另一个条件的限制。这里的计算只是表明一个大文件是如何寻址和分配的。

其实看到这里的计算数值，就知道ext2和ext3对超大文件的存取效率是低下的，它要核对太多的指针，特别是4KB大小的blocksize时。而ext4针对这一点就进行了优化，ext4使用extent的管理方式取代ext2和ext3的块映射，大大提高了效率也降低了碎片。

4.6 单文件系统中文件操作的原理

在Linux上执行删除、复制、重命名、移动等操作时，它们是怎么进行的呢？还有访问文件时是如何找到它的呢？其实只要理解了前文中介绍的几个术语以及它们的作用就很容易知道文件操作的原理了。

注：在这一小节所解释的都是在单个文件系统下的行为，在多个文件系统中如何请看下一个小节：多文件系统关联。

4.6.1 读取文件

当执行"cat /var/log/messages"命令在系统内部进行了什么样的步骤呢？该命令能被成功执行涉及了cat命令的寻找、权限判断以及messages文件的寻找和权限判断等等复杂的过程。这里只解释和本节内容相关的如何寻找到被cat的/var/log/messages文件。

- **找到根文件系统的块组描述符表所在的blocks，读取GDT(已在内存中)找到inode table的block号。**

因为GDT总是和superblock在同一个块组，而superblock总是在分区的第1024-2047个字节，所以很容易就知道第一个GDT所在的块组以及GDT在这个块组中占用了哪些block。

其实GDT早已经在内存中了，在系统开机的时候会挂载根文件系统，挂载的时候就已经将所有的GDT放进内存中。

- **在inode table的block中定位到根"/"的inode，找出"/"指向的data block。**

前文说过，ext文件系统预留了一些inode号，其中"/"的inode号为2，所以可以根据inode号直接定位根目录文件的data block。

- **在"/"的datablock中记录了var目录名和var的inode号，找到该inode记录，inode记录中存储了指向var的block指针，所以也就找到了var目录文件的data block。**

通过var目录的inode号，可以寻找到var目录的inode记录，但是在寻找的过程中，还需要知道该inode记录所在的块组以及所在的inode table，所以需要读取GDT，同样，GDT已经缓存到了内存中。

- **在var的data block中记录了log目录名和其inode号，通过该inode号定位到该inode所在的块组及所在的inode table，并根据该inode记录找到log的data block。**
- **在log目录文件的data block中记录了messages文件名和对应的inode号，通过该inode号定位到该inode所在的块组及所在的inode table，并根据该inode记录找到messages的data block。**
- **最后读取messages对应的datablock。**

将上述步骤中GDT部分的步骤简化后比较容易理解。如下:找到GDT-->找到"/"的inode-->找到/的数据块读取var的inode-->找到var的数据块读取log的inode-->找到log的数据块读取messages的inode-->找到messages的数据块并读取它们。

当然，在每次定位到inode记录后，都会先将inode记录加载到内存中，然后查看权限，如果权限允许，将根据block指针找到对应的data block。

4.6.2 删除、重命名和移动文件

注意这里是不跨越文件系统的操作行为。

- **删除文件分为普通文件和目录文件，知道了这两种类型的文件的删除原理，就知道了其他类型特殊文件的删除方法。**

对于删除普通文件：(1)找到文件的inode和data block(根据前一个小节中的方法寻找)；(2)将inode table中该inode记录中的data block指针删除；(3)在imap中将该文件的inode号标记为未使用；(4)在其所在目录的data block中将该文件名所在的记录行删除，删除了记录就丢失了指向inode的指针（实际上不是真的删除，直接删除的话会在目录data block的数

据结构中产生空洞，所以实际的操作是将待删除文件的inode号设置为特殊的值0，这样下次新建文件时就可以重用该行记录）；(5)将bmap中data block对应的block号标记为未使用。

对于删除目录文件：找到目录和目录下所有文件、子目录、子文件的inode和data block；在imap中将这些inode号标记为未使用；将bmap中将这些文件占用的 block号标记为未使用；在该目录的父目录的data block中将该目录名所在的记录行删除。需要注意的是，删除父目录data block中的记录是最后一步，如果该步骤提前，将报目录非空的错误，因为在该目录中还有文件占用。

关于上面的(2)-(5)：当(2)中删除data block指针后，将无法再找到这个文件的数据；当(3)标记inode号未使用，表示该inode号可以被后续的文件重用；当(4)删除目录data block中关于该文件的记录，真正的删除文件，外界再也定位也无法看到这个文件了；当(5)标记data block为未使用后，表示开始释放空间，这些data block可以被其他文件重用。

注意，在第(5)步之前，由于data block还未被标记为未使用，在superblock中仍然认为这些data block是正在使用中的。这表示尽管文件已经被删除了，但空间却还没有释放，df也会将其统计到已用空间中(df是读取superblock中的数据块数量，并计算转换为空间大小)。

什么时候会发生这种情况呢？当一个进程正在引用文件时将该文件删除，就会出现文件已删除但空间未释放的情况。这时步骤已经进行到(4)，外界无法再找到该文件，但由于进程在加载该文件时已经获取到了该文件所有的data block指针，该进程可以获取到该文件的所有数据，但却暂时不会释放该文件空间。直到该进程结束，文件系统才将未执行的步骤(5)继续完成。这也是为什么有时候du的统计结果比df小的原因，关于du和df统计结果的差别，详细内容见：详细分析du和df的统计结果为什么不一样。

- **重命名文件分为同目录内重命名和非同目录内重命名。非同目录内重命名实际上是移动文件的过程，见下文。**

同目录内重命名文件的动作仅仅只是修改所在目录data block中该文件记录的文件名部分，不是删除再重建的过程。

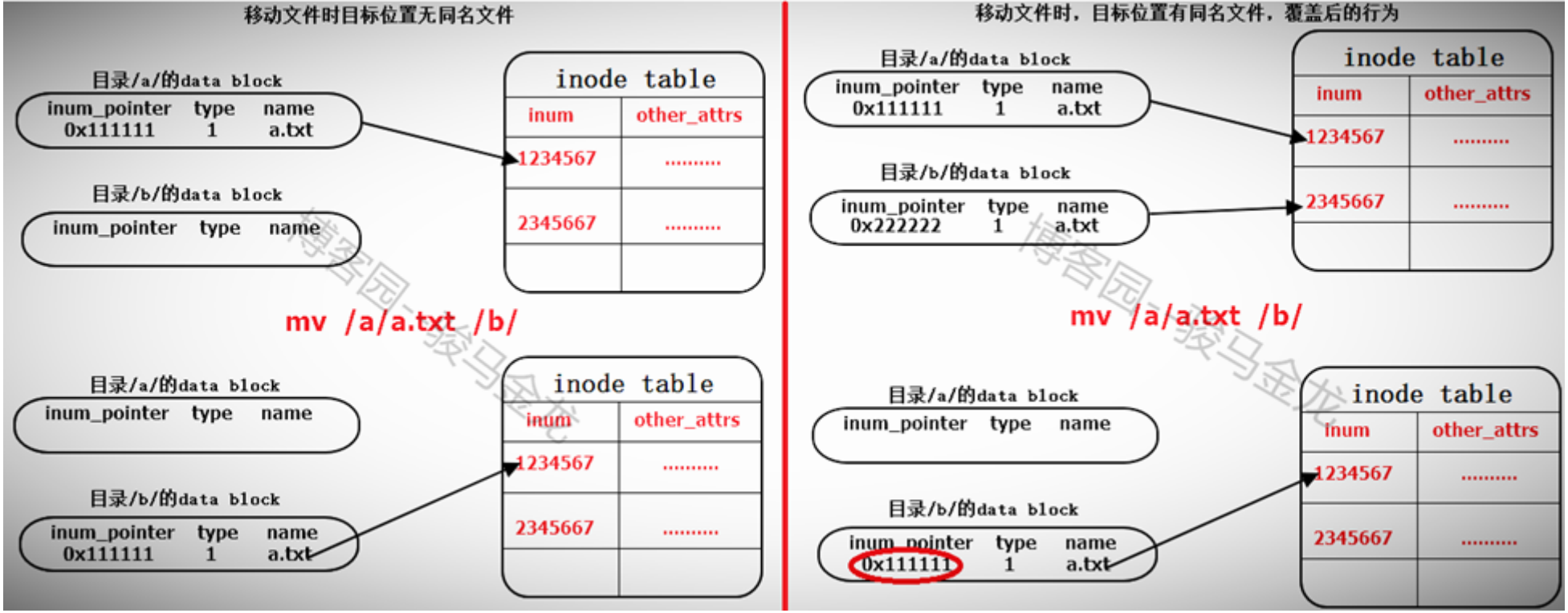
如果重命名时有文件名冲突(该目录内已经存在该文件名)，则提示是否覆盖。覆盖的过程是覆盖目录data block中冲突文件的记录。例如/tmp/下有a.txt和a.log，若将a.txt重命名为a.log，则提示覆盖，若选择覆盖，则/tmp的data block中关于a.log的记录被覆盖。

- **移动文件**

同文件系统下移动文件实际上是修改目标文件所在目录的data block，向其中添加一行指向inode table中待移动文件的inode指针，如果目标路径下有同名文件，则会提示是否覆盖，实际上是覆盖目录data block中冲突文件的记录，由于同名文件的inode记录指针被覆盖，所以无法再找到该文件的data block，也就是说该文件被标记为删除(如果多个硬链接数，则另当别论)。

所以在同文件系统内移动文件相当快，仅仅在所在目录data block中添加或覆盖了一条记录而已。也因此，移动文件时，文件的inode号是不会改变的。

对于不同文件系统内的移动，相当于先复制再删除的动作。见后文。



关于文件移动，在Linux环境下有一个非常经典网上却又没任何解释的问题：/tmp/a/a能覆盖为/tmp/a吗？答案是不能，但windows能。为什么不能？见mv的一个经典问题(mv的本质)。

4.6.3 存储和复制文件

- 对于文件存储
  - (1).读取GDT，找到各个(或部分)块组imap中未使用的inode号，并为待存储文件分配inode号；
  - (2).在inode table中完善该inode号所在行的记录；
  - (3).在目录的data block中添加一条该文件的相关记录；
  - (4).将数据填充到data block中。
    - 注意，填充到data block中的时候会调用block分配器：一次分配4KB大小的block数量，当填充完4KB的data block后会继续调用block分配器分配4KB的block，然后循环直到填充完所有数据。也就是说，如果存储一个100M的文件需要调用block分配器100*1024/4=25600次。
    - 另一方面，在block分配器分配block时，block分配器并不知道真正有多少block要分配，只是每次需要分配时就分配，在每存储一个data block前，就去bmap中标记一次该block已使用，它无法实现一次标记多个bmap位。这一点在ext4中进行了优化。
  - (5)填充完之后，去inode table中更新该文件inode记录中指向data block的寻址指针。

- 对于复制，完全就是另一种方式的存储文件。步骤和存储文件的步骤一样。

4.7 多文件系统关联

在单个文件系统中的文件操作和多文件系统中的操作有所不同。本文将对此做出非常详细的说明。

4.7.1 根文件系统的特殊性

这里要明确的是，任何一个文件系统要在Linux上能正常使用，必须挂载在某个已经挂载好的文件系统中的某个目录下，例如/dev/cdrom挂载在/mnt上，/mnt目录本身是在"/"文件系统下的。而且任意文件系统的一级挂载点必须是在根文件系统的某个目录下，因为只有"/"是自引用的。这里要说明挂载点的级别和自引用的概念。

假如/dev/sdb1挂载在/mydata上，/dev/cdrom挂载在/mydata/cdrom上，那么/mydata就是一级挂载点，此时/mydata已经是文件系统/dev/sdb1的入口了，而/dev/cdrom所挂载的目录/mydata/cdrom是文件系统/dev/sdb1中的某个目录，那么/mydata/cdrom就是二级挂载点。一级挂载点必须在根文件系统下，所以可简述为：文件系统2挂载在文件系统1中的某个目录下，而文件系统1又挂载在根文件系统中的某个目录下。

再解释自引用。首先要说的是，自引用的只能是文件系统，而文件系统表现形式是一个目录，所以自引用是指该目录的data block中，"."和".."的记录中的inode号都对应inode table中同一个inode记录，所以它们inode号是相同的，即互为硬链接。而根文件系统是唯一可以自引用的文件系统。

```
[root@xuexi /]# ll -ai /
total 102
     2 dr-xr-xr-x.  22 root root  4096 Jun  6 18:13 .
     2 dr-xr-xr-x.  22 root root  4096 Jun  6 18:13 ..
```

由此也能解释cd /.和cd /..的结果都还是在根下，这是自引用最直接的表现形式。

```
[root@xuexi tmp]# cd /.
[root@xuexi /]#
[root@xuexi tmp]# cd /..
[root@xuexi /]#
```

注意，根目录下的"."和".."都是"/"目录的硬链接，且其datablock中不记录名为"/"的条目，因此除去根目录下子目录数后的硬链接数为2。

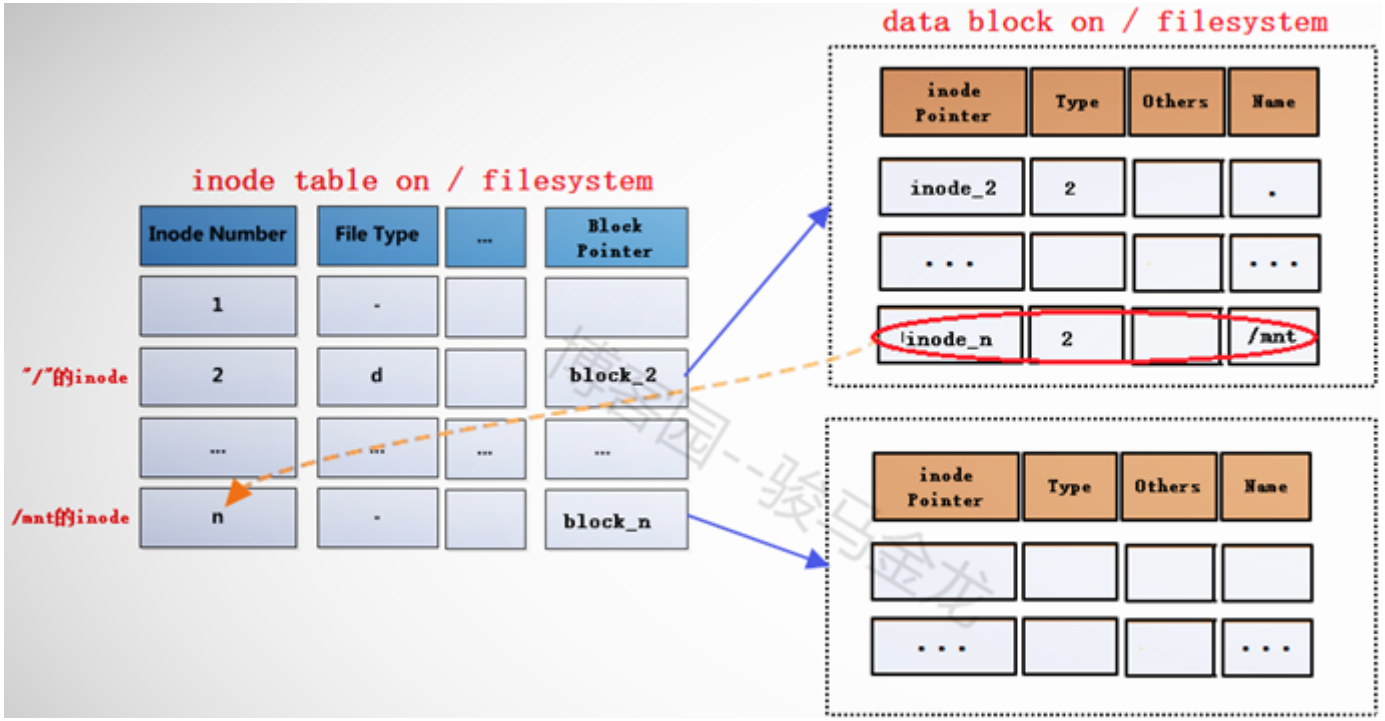
```
[root@server2 tmp]# a=$(ls -ld / | awk '{print $2}')
[root@server2 tmp]# b=$(ls -l / | grep "^d" |wc -l)
[root@server2 tmp]# echo $((a - b))
2
```

4.7.2 挂载文件系统的细节

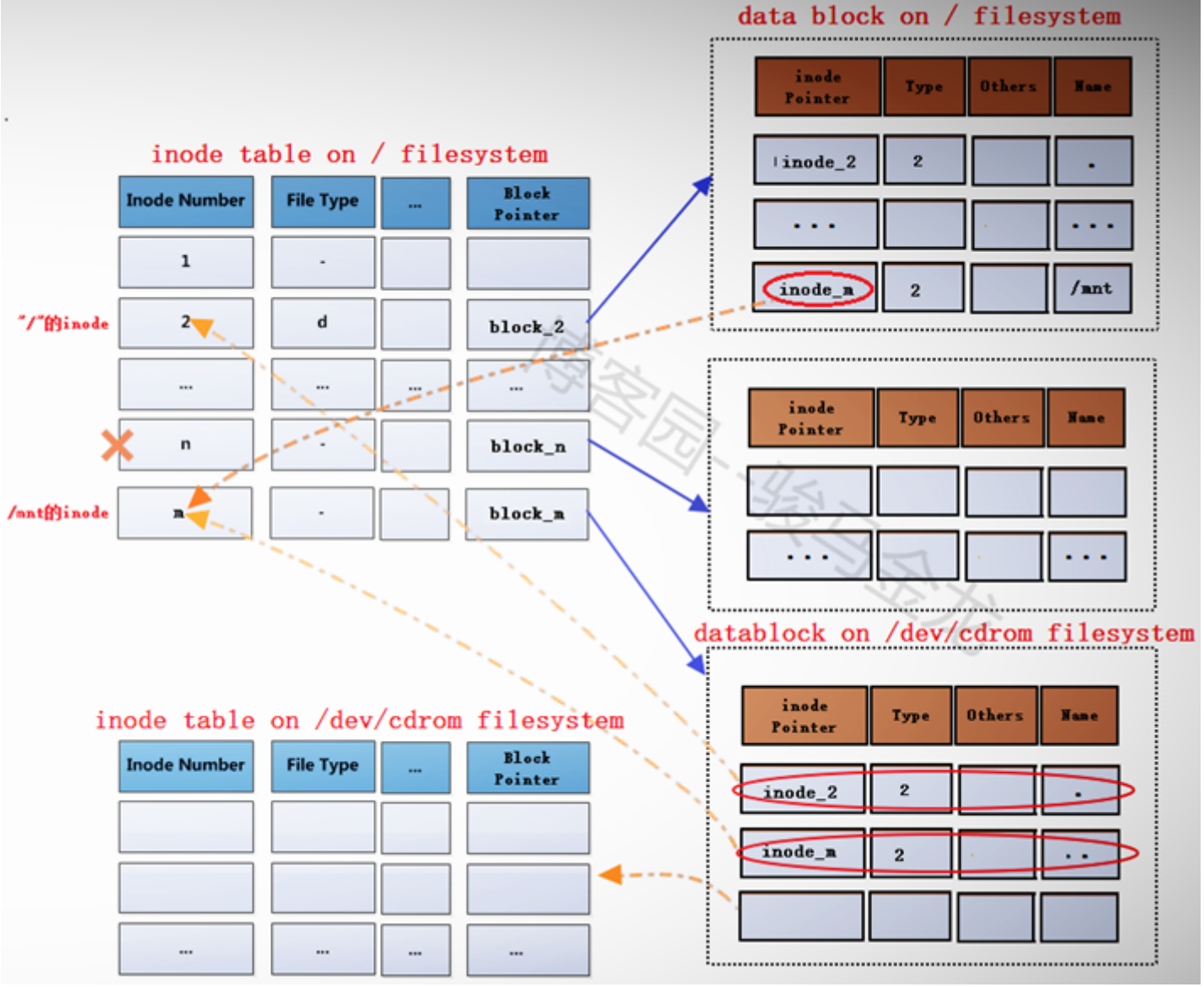
挂载文件系统到某个目录下，例如"mount /dev/cdrom /mnt"，挂载成功后/mnt目录中的文件全都暂时不可见了，且挂载后权限和所有者(如果指定允许普通用户挂载)等的都改变了，知道为什么吗?

下面就以通过"mount /dev/cdrom /mnt"为例，详细说明挂载过程中涉及的细节。

在将文件系统/dev/cdrom(此处暂且认为它是文件系统)挂载到挂载点/mnt之前，挂载点/mnt是根文件系统中的一个目录，"/"的data block中记录了/mnt的一些信息，其中包括inode号inode_n，而在inode table中，/mnt对应的inode记录中又存储了block指针block_n，此时这两个指针还是普通的指针。



当文件系统/dev/cdrom挂载到/mnt上后，/mnt此时就已经成为另一个文件系统的入口了，因此它需要连接两边文件系统的inode和data block。但是如何连接呢？如下图。



在根文件系统的inode table中，为/mnt重新分配一个inode记录m，该记录的block指针block_m指向文件系统/dev/cdrom中的data block。既然为/mnt分配了新的inode记录m，那么在"/"目录的data block中，也需要修改其inode指针为inode_m以指向m记录。同时，原来inode table中的inode记录n就被标记为暂时不可用。

block_m指向的是文件系统/dev/cdrom的data block，所以严格说起来，除了/mnt的元数据信息即inode记录m还在根文件系统上，/mnt的data block已经是在/dev/cdrom中的了。这就是挂载新文件系统后实现的跨文件系统，它将挂载点的元数据信息和数据信息分别存储在不同的文件系统上。

挂载完成后，将在/proc/self/{mounts,mountstats,mountinfo}这三个文件中写入挂载记录和相关的挂载信息，并会将/proc/self/mounts中的信息同步到/etc/mtab文件中，当然，如果挂载时加了-n参数，将不会同步到/etc/mtab。

而卸载文件系统，其实质是移除临时新建的inode记录(当然，在移除前会检查是否正在使用)及其指针，并将指针指回原来的inode记录，这样inode记录中的block指针也就同时生效而找回对应的data block了。由于卸载只是移除inode记录，所以使用挂载点和文件系统都可以实现卸载，因为它们是联系在一起的。

下面是分析或结论。

(1).挂载点挂载时的inode记录是新分配的。

挂载前挂载点/mnt的inode号

```
[root@server2 tmp]# ll -id /mnt
100663447 drwxr-xr-x. 2 root root 6 Aug 12  2015 /mnt

[root@server2 tmp]# mount /dev/cdrom /mnt
```

```
# 挂载后挂载点的inode号
[root@server2 tmp]# ll -id /mnt
1856 dr-xr-xr-x    8 root root  2048 Dec 10  2015 mnt
```

由此可以验证，inode号确实是重新分配的。

(2).挂载后，挂载点的内容将暂时不可见、不可用，卸载后文件又再次可见、可用。

```
# 在挂载前，向挂载点中创建几个文件
[root@server2 tmp]# touch /mnt/a.txt
[root@server2 tmp]# mkdir /mnt/abcdir
```

```
# 挂载
[root@server2 tmp]# mount /dev/cdrom /mnt

# 挂载后，挂载点中将找不到刚创建的文件
[root@server2 tmp]# ll /mnt
total 636
-r--r--r-- 1 root root     14 Dec 10  2015 CentOS_BuildTag
dr-xr-xr-x 3 root root   2048 Dec 10  2015 EFI
-r--r--r-- 1 root root    215 Dec 10  2015 EULA
-r--r--r-- 1 root root  18009 Dec 10  2015 GPL
dr-xr-xr-x 3 root root   2048 Dec 10  2015 images
dr-xr-xr-x 2 root root   2048 Dec 10  2015 isolinux
dr-xr-xr-x 2 root root   2048 Dec 10  2015 LiveOS
dr-xr-xr-x 2 root root 612352 Dec 10  2015 Packages
dr-xr-xr-x 2 root root   4096 Dec 10  2015 repodata
-r--r--r-- 1 root root   1690 Dec 10  2015 RPM-GPG-KEY-CentOS-7
-r--r--r-- 1 root root   1690 Dec 10  2015 RPM-GPG-KEY-CentOS-Testing-7
-r--r--r-- 1 root root   2883 Dec 10  2015 TRANS.TBL

# 卸载后，挂载点/mnt中的文件将再次可见
[root@server2 tmp]# umount /mnt
```

```
[root@server2 tmp]# ll /mnt
total 0
drwxr-xr-x 2 root root 6 Jun  9 08:18 abcdir
-rw-r--r-- 1 root root 0 Jun  9 08:18 a.txt
```

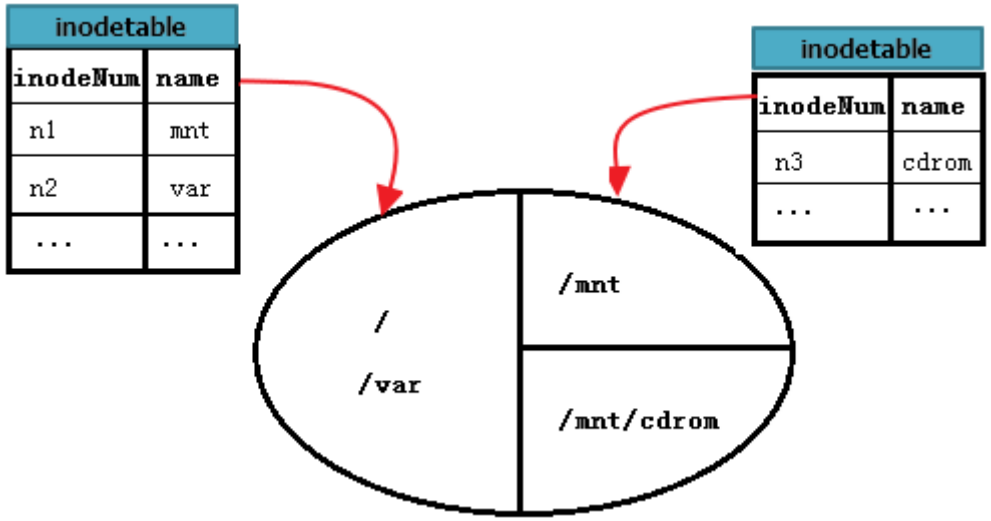
之所以会这样，是因为挂载文件系统后，挂载点原来的inode记录暂时被标记为不可用，关键是没有指向该inode记录的inode指针了。在卸载文件系统后，又重新启用挂载点原来的inode记录，"/"目录下的mnt的inode指针又重新指向该inode记录。

(3).挂载后，挂载点的元数据和data block是分别存放在不同文件系统上的。

(4).挂载点即使在挂载后，也还是属于源文件系统的文件。

4.7.3 多文件系统操作关联

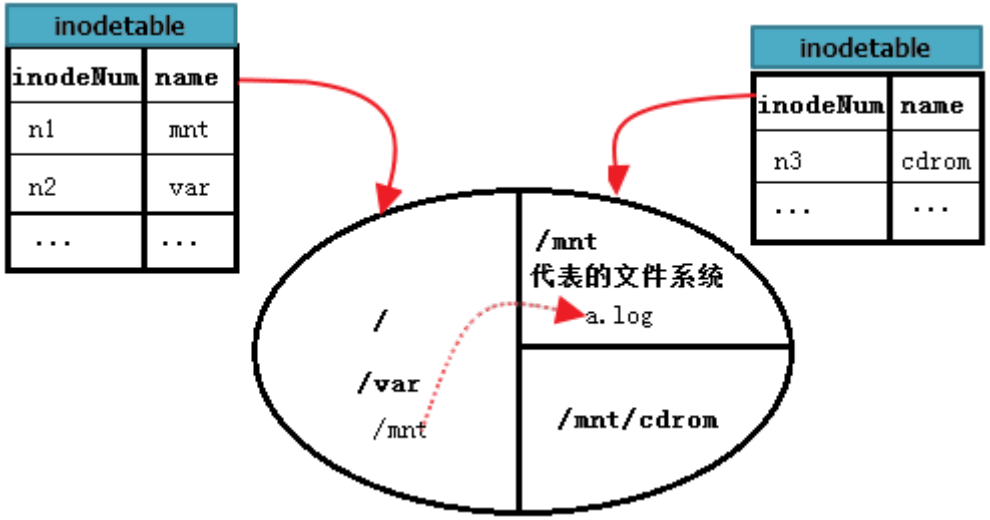
假如下图中的圆代表一块硬盘，其中划分了3个区即3个文件系统。其中根是根文件系统，/mnt是另一个文件系统A的入口，A文件系统挂载在/mnt上，/mnt/cdrom也是一个文件系统B的入口，B文件系统挂载在/mnt/cdrom上。每个文件系统都维护了一些inode table，这里假设图中的inode table是每个文件系统所有块组中的inode table的集合表。



如何读取/var/log/messages呢？这是和"/"在同一个文件系统的文件读取，在前面单文件系统中已经详细说明了。

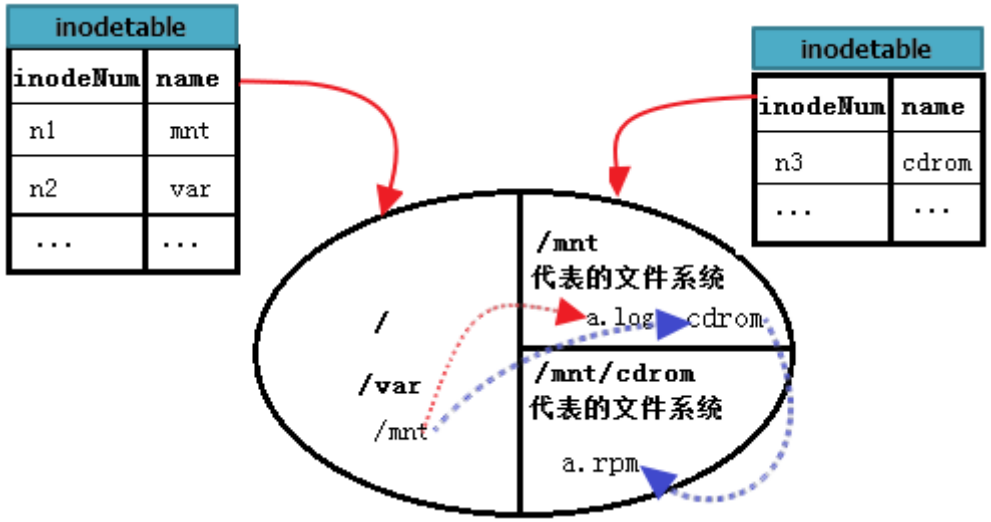
但如何读取A文件系统中的/mnt/a.log呢？首先，从根文件系统找到/mnt的inode记录，这是单文件系统内的查找；然后根据此inode记录的block指针，定位到/mnt的data block中，这些block是A文件系统的data block；然后从/mnt的data block中读取a.log记录，并根据a.log的inode指针定位到A文件系统的inode table中对应a.log的inode记录；最后从此inode记录的block指针找到a.log的data block。至此，就能读取到/mnt/a.log文件的内容。

下图能更完整的描述上述过程。



那么又如何读取/mnt/cdrom中的/mnt/cdrom/a.rpm呢？这里cdrom代表的文件系统B挂载点位于/mnt下，所以又多了一个步骤。先找到"/"，再找到根中的mnt，进入到mnt文件系统中，找到cdrom的data block，再进入到cdrom找到a.rpm。也就是说，mnt目录文件存放位置是根，cdrom目录文件存放位置是mnt，最后a.rpm存放的位置才是cdrom。

继续完善上图。如下。



4.8 ext3文件系统的日志功能

相比ext2文件系统，ext3多了一个日志功能。

在ext2文件系统中，只有两个区：数据区和元数据区。如果正在向data block中填充数据时突然断电，那么下一次启动时就会检查文件系统中数据和状态的一致性，这段检查和修复可能会消耗大量时间，甚至检查后无法修复。之所以会这样是因为文件系统在突然断电后，它不知道上次正在存储的文件的block从哪里开始、哪里结束，所以它会扫描整个文件系统进行排除(也许是这样检查的吧)。

而在创建ext3文件系统时会划分三个区：数据区、日志区和元数据区。每次存储数据时，先在日志区中进行ext2中元数据区的活动，直到文件存储完成后标记上commit才将日志区中的数据转存到元数据区。当存储文件时突然断电，下一次检查修复文件系统时，只需要检查日志区的记录，将bmap对应的data block标记为未使用，并把inode号标记未使用，这样就不需要扫描整个文件系统而耗费大量时间。

虽说ext3相比ext2多了一个日志区转写元数据区的动作而导致ext3相比ext2性能要差一点，特别是写众多小文件时。但是由于ext3其他方面的优化使得ext3和ext2性能几乎没有差距。

4.9 ext4文件系统

回顾前面关于ext2和ext3文件系统的存储格式，它使用block为存储单元，每个block使用bmap中的位来标记是否空闲，尽管使用划分块组的方法优化提高了效率，但是一个块组内部仍然使用bmap来标记该块组内的block。对于一个巨大的文件，扫描整个bmap都将是一件浩大的工程。另外在inode寻址方面，ext2/3使用直接和间接的寻址方式，对于三级间接指针，可能要遍历的指针数量是非常非常巨大的。

ext4文件系统的最大特点是在ext3的基础上使用区(extent，或称为段)的概念来管理。一个extent尽可能的包含物理上连续的一堆block。inode寻址方面也一样使用区段树的方式进行了改进。

默认情况下，EXT4不再使用EXT3的block mapping分配方式 ，而改为Extent方式分配。

以下是ext4文件系统中一个文件的inode属性示例，注意最后两行的EXTENTS。

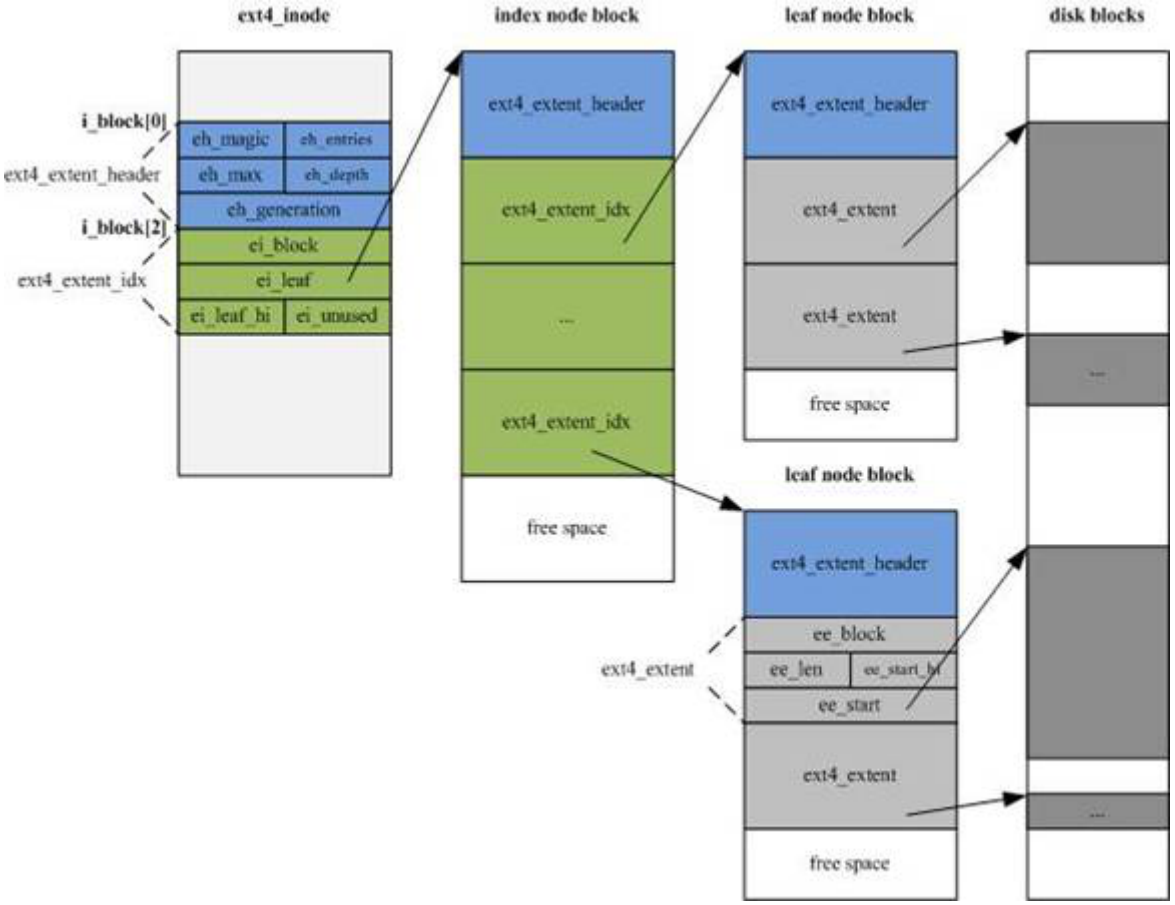
```
Inode: 12   Type: regular    Mode:  0644   Flags: 0x80000
Generation: 476513974    Version: 0x00000000:00000001
User:     0   Group:     0   Size: 11
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x5b628ca0:491d6224 -- Thu Aug  2 12:46:24 2018
 atime: 0x5b628ca0:491d6224 -- Thu Aug  2 12:46:24 2018
 mtime: 0x5b628ca0:491d6224 -- Thu Aug  2 12:46:24 2018
crtime: 0x5b628ca0:491d6224 -- Thu Aug  2 12:46:24 2018
Size of extra inode fields: 28
EXTENTS:
(0):33409
```

(1). 关于EXT4的结构特征

EXT4在总体结构上与EXT3相似，大的分配方向都是基于相同大小的块组，每个块组内分配固定数量的inode、可能的superblock(或备份)及GDT。

EXT4的inode 结构做了重大改变，为增加新的信息，大小由EXT3的128字节增加到默认的256字节，同时inode寻址索引不再使用EXT3的"12个直接寻址块+1个一级间接寻址块+1个二级间接寻址块+1个三级间接寻址块"的索引模式，而改为4个Extent片断流，每个片断流设定片断的起始block号及连续的block数量(有可能直接指向数据区，也有可能指向索引块区)。

片段流即下图中索引节点(inde node block)部分的绿色区域，每个15字节，共60字节。



(2). EXT4删除数据的结构更改。

EXT4删除数据后，会依次释放文件系统bitmap空间位、更新目录结构、释放inode空间位。

(3). ext4使用多block分配方式。

在存储数据时，ext3中的block分配器一次只能分配4KB大小的Block数量，而且每存储一个block前就标记一次bmap。假如存储1G的文件，blocksize是4KB，那么每存储完一个Block就将调用一次block分配器，即调用的次数为1024*1024/4KB=262144次，标记bmap的次数也为1024*1024/4=262144次。

而在ext4中根据区段来分配，可以实现调用一次block分配器就分配一堆连续的block，并在存储这一堆block前一次性标记对应的bmap。这对于大文件来说极大的提升了存储效率。

4.10 ext类的文件系统的缺点

最大的缺点是它在创建文件系统的时候就划分好一切需要划分的东西，以后用到的时候可以直接进行分配，也就是说它不支持动态划分和动态分配。对于较小的分区来说速度还好，但是对于一个超大的磁盘，速度是极慢极慢的。例如将一个几十T的磁盘阵列格式化为ext4文件系统，可能你会因此而失去一切耐心。

除了格式化速度超慢以外，ext4文件系统还是非常可取的。当然，不同公司开发的文件系统都各有特色，最主要的还是根据需求选择合适的文件系统类型。

4.11 虚拟文件系统VFS

每一个分区格式化后都可以建立一个文件系统，Linux上可以识别很多种文件系统，那么它是如何识别的呢？另外，在我们操作分区中的文件时，并没有指定过它是哪个文件系统的，各种不同的文件系统如何被我们用户以无差别的方式操作呢？这就是虚拟文件系统的作用。

虚拟文件系统为用户操作各种文件系统提供了通用接口，使得用户执行程序时不需要考虑文件是在哪种类型的文件系统上，应该使用什么样的系统调用来操作该文件。有了虚拟文件系统，只要将所有需要执行的程序调用VFS的系统调用就可以了，剩下的动作由VFS来帮忙完成。