Linux 文件系统详解

By hubert

————银河学院

目录

一、文件系统简介

文件系统是操作系统最为重要的一部分，它定义了磁盘上储存文件的方法和数据结构。文件系统是操作系统组织、存取和保存信息的重要手段，每种操作系统都有自己的文件系统，如 Windows 所用的文件系统主要有 FAT16、FAT32 和 NTFS，Linux 所用的文件系统主要有 ext2、ext3、ext4、ReiserFS 和 btrfs 等。

我们知道，一块磁盘，要先分区，然后再格式化，否则就无法使用。而这个格式化的过程，就是文件系统创建的过程，也可以这样理解，磁盘上的一个分区，就是一个文件系统。这个就像我们在使用 windows 系统的时候，把磁盘分区可以格式化成 FAT32 或者 NTFS，但所格式化的文件系统必需是你使用的系统所能认出来的。这就是为什么 NTFS 的文件系统，不能直接被 Linux 系统所认识。同样，windows 也不能认识 ext3/ext4 一样的道理。

关于文件系统的定义，没有一个标准的答案，如果有更高兴趣来研究文件系统的朋友，可能找相关的资料来查阅。本文只是带大家初步来了解认识 linux 文件系统。下面是一些和文件系统相关的术语，大家一起来了解一下吧。

存储介质：硬盘、光盘、软盘、Flash 盘、磁带、网络存储设备等。

磁盘的分区：这是针对大容量的存储设备来说的，主要是指硬盘；对于大硬盘，我们要合理地进行分区规划。

文件系统的创建：这个过程是存储设备建立文件系统的过程，一般也被称为格式化或初始化。

挂载（mount）：文件系统只有挂载才能使用，Linux 的操作系统是通过 mount 进行的，挂载文件系统时要有挂载点，比如我们在安装 Linux 的过程中，有时会提示我们分区，然后建立文件系统，接着是问你的挂载点是什么。我们在 Linux 系统的使用过程中，也会挂载其它的硬盘分区，也要选中挂载点，挂载点通常是一个空置的目录

文件系统结构：文件系统的是用来组织和排列文件存取的，所以它是可见的，在 Linux 中，我们可以通过 ls 等工具来查看其结构，在 Linux 系统中，我们见到的都是树形结构；比如操作系统安装在一个文件系统中，他表现为由/起始的树形结构。

二、Linux 常见文件系统格式介绍

ext1：第一个受 Linux 支持的文件系统是 Minix 文件系统。这个文件系统有严重的性能问题，因此出现了另一个针对 Linux 的文件系统，即扩展文件系统。第 1 个扩展文件系统（ext1）由 Remy Card 设计，并于 1992 年 4 月引入到 Linux 中。ext1 文件系统是第一个使用虚拟文件系统（VFS）交换的文件系统。支持的最大文件系统为 2GB。

ext2：第 2 个扩展文件系统（ext2）也是由 Remy Card 实现的，并于 1993 年 1 月引入到 Linux 中。它借鉴了当时文件系统（比如 Berkeley Fast File System [FFS]）的先进想法。ext2 支持的最大文件系统为 2TB，但是 2.6 内

核将该文件系统支持的最大容量提升到 32TB。

ext3：第 3 个扩展文件系统（ext3）是 Linux 文件系统的重大改进，尽管它在性能方面逊色于某些竞争对手。ext3 文件系统引入了日志概念，以在系统突然停止时提高文件系统的可靠性。虽然某些文件系统的性能更好（比如 Silicon Graphics 的 XFS 和 IBM® Journaled File System [JFS]），但 ext3 支持从使用 ext2 的系统进行就地（in-place）升级。ext3 由 Stephen Tweedie 实现，并于 2001 年 11 月引入。

ext4：2.6.28 内核是首个稳定的 ext4 文件系统。在性能、伸缩性和可靠性方面进行了大量改进。最值得一提的是，ext4 支持 1 EB 的文件系统。ext4 是由 Theodore Tso（ext3 的维护者）领导的开发团队实现的，并引入到 2.6.19 内核中。目前，它在 2.6.28 内核中已经很稳定（到 2008 年 12 月为止）。

ext4 从竞争对手那里借鉴了许多有用的概念。例如，在 JFS 中已经实现了使用区段（extent）来管理块。另一个与块管理相关的特性（延迟分配）已经在 XFS 和 Sun Microsystems 的 ZFS 中实现。在 ext4 文件系统中，您可以发现各种改进和创新。这些改进包括新特性（新功能）、伸缩性（打破当前文件系统的限制）和可靠性（应对故障），当然也包括性能的改善。

swap: 它是 Linux 中一种专门用于交换分区的 swap 文件系统。Linux 是使用这一整个分区作为交换空间。一般这个 swap 格式的交换分区是主内存的 2 倍。在内存不够时，Linux 会将部分数据写到交换分区上。

三、文件系统工作原理

文件系统的工作与操作系统的文件数据有关。现在的操作系统的文件数据除了文件实际内容外，通常含有非常多的属性，例如文件权限(rwx)与文件属性(所有者、用户组、时间参数等)。文件系统通常会将这两部份的数据分别存放在不同的区块，权限与属性放到 inode 中，数据则放到 block 区块中。另外，还有一个超级区块(super block)会记录整个文件系统的整体信息，包括 inode 与 block 的总量、使用量、剩余量等等等。

每个 inode 与 block 都有编号，至于这三个数据的意义可以简略说明如下：

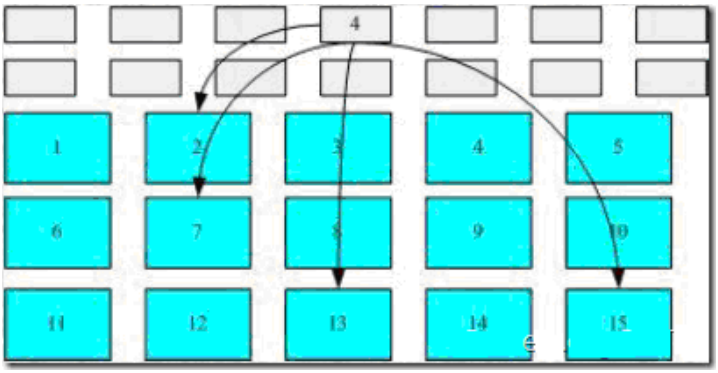
superblock：记录此 filesystem 的整体信息，包括 inode/block 的总量、使用量、剩余量， 以及文件系统的格式与相关信息等；

inode：记录文件的属性，一个文件占用一个 inode，同时记录此文件的数据所在的 block 号码；

block：实际记录文件的内容，若文件太大时，会占用多个 block 。

由于每个 inode 与 block 都有编号，而每个文件都会占用一个 inode，inode 内则有文件数据放置的 block 号码。因此，我们可以知道的是，如果能够找到文件的 inode 的话，那么自然就会知道这个文件所放置数据的 block 号码，当然也就能够读出该文件的实际数据了。这是个比较有效率的作法，因为如此一来我们的磁盘就能够在短时间内读取出全部的数据， 读写的效能比较好。

我们将 inode 与 block 区块用图解来说明一下，如下图所示，文件系统先格式化出 inode 与 block 的区块，假设某一个档案的属性与权限数据是放置到 inode 4 号(下图较小方格内) ,而这个 inode 记录了档案数据的实际放置点为 2, 7, 13, 15 这四个 block 号码，此时我们的操作系统就能够据此来排列磁盘的阅读顺序，可以一口气将四个 block 内容读出来！ 那么数据的读取就如同下图中的箭头所指定的模样了。inode/block 资料存取示意图：



这种数据存取的方法我们称为索引式文件系统(indexed allocation)。下面我们来看一下 windows 系统中的 FAT，这种格式的文件系统并没有 inode 存在，所以 FAT 没有办法将这个文件的所有 block 在一开始就读取出来。每个 block

号码都记录在前一个 block 当中， 他的读取方式有点像底下这样：

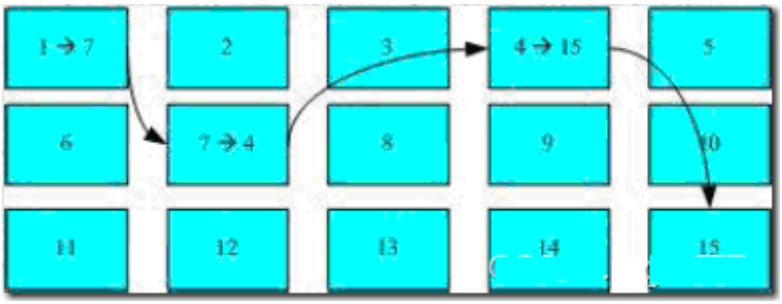


图 FAT 文件系统资料存取示意图

上图中我们假设文件的数据依序写入 1->7->4->15 号这四个 block 号码中， 但这个文件系统没有办法一口气就知道四个 block 的号码，他得要一个一个的将 block 读出后，才会知道下一个 block 在何处。如果同一个文件数据写入的 block 分散的太厉害时，则我们的磁盘读取头将无法在磁盘转一圈就读到所有的数据， 因此磁盘就会多转好几圈才能完整的读取到这个文件的内容。

这就是为什么在 windows 系统中常常需要碎片整理，需要碎片整理的原因就是文件写入的 block 太过于离散了，此时文件读取的效能将会变的很差。这个时候可以通过碎片整理将同一个文件所属的 blocks 汇整在一起，这样数据的读取会比较容易。

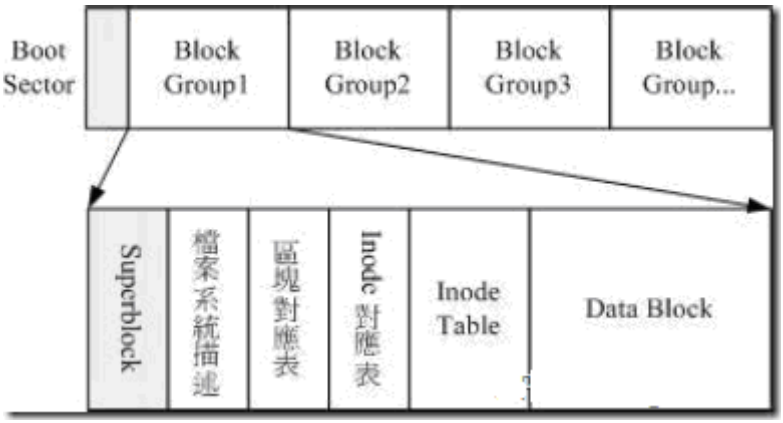
FAT 的文件系统需要不定期进行碎片整理，那么 ext 是否需要磁盘整理呢？

由于 Ext 是索引式文件系统，基本上不需要进行碎片整理。但是如果文件系统使用太久， 常常删除/编辑/新增文件时，那么还是可能会造成文件数据太过于离散的问题，此时或许会需要进行重整一下的。 不过，似乎没有多少人来进行 Linux 磁盘的碎片整理啊！

如上所说的，inode 的内容是记录文件的权限与相关属性，至于 block 区块则是在记录文件的实际内容。而且文件系统一开始就将 inode 与 block 规划好了，除非重新格式化(或者利用 resize2fs 等指令变更文件系统大小)，否则 inode 与 block 固定后就不再变动。但是如果仔细考虑一下，如果我的文档系统高达数百 GB 时， 那么将所有的 inode 与 block 通通放置在一起将是很不智的决定，因为 inode 与 block 的数量太庞大，不容易管理。 因此 ext2 文件系统在格式化的时候基本上是区分为多个区块群组 (block group) 的，每个区块群组都有独立的 inode/block/superblock 系统。

四、Ext2 文件系统结构详解

整体来说，Ext2 格式化后有点像底下这样：



ext2 文件系统示意图

在整体的规划当中，文件系统最前面有一个启动扇区(boot sector)，这个启动扇区可以安装开机管理程序，这是个非常重要的设计，这样我们就能够将不同的开机管理程序安装到个别的文件系统最前端，而不用覆盖整个硬盘唯一的 MBR， 这样也才能够制作出多重引导的环境啊！至于每一个区块群组(block group)的六个主要内容说明如下：

data block (资料区块)

data block 是用来放置文件内容数据地方，在 Ext2 文件系统中所支持的 block 大小有 1K, 2K 及 4K 三种而

已。在格式化时 block 的大小就固定了，且每个 block 都有编号，以方便 inode 的记录啦。不过要注意的是，由于 block 大小的差异，会导致该文件系统能够支持的最大磁盘容量与最大单一文件容量并不相同。 因为 block 大小而产生的 Ext2 文件系统限制如下图：

Block 大小	1KB	2KB	4KB
最大单一文件限制	16GB	256GB	2TB
最大文件系统总容量	2TB	8TB	16TB

你需要注意的是，虽然 Ext2 已经能够支持大于 2GB 以上的单一文件容量，不过某些应用程序依然使用旧的限制，也就是说，某些程序只能够识别到小于 2GB 以下的文件而已，这就跟文件系统无关了！

注意：

原则上，block 的大小与数量在格式化完就不能够再改变了(除非重新格式化)；

每个 block 内最多只能够放置一个档案的数据；

如果档案大于 block 的大小，则一个档案会占用多个 block 数量；

若档案小于 block ，则该 block 的剩余容量就不能够再被使用了(磁盘空间会浪费)；

若档案很大的话，就会占用更多的 block，将会降低文件系统的读写效能。

inode table (inode 表格)

再来讨论一下 inode 这个吧！如前所述 inode 的作用是记录文件的属性以及该文件实际数据是放置在哪几号 block 内！ 基本上，inode 记录的文件数据至少有底下这些：

该文件的存取模式(read/write/excute)；

该文件的拥有者与群组(owner/group)；

该文件的容量；

该文件建立或状态改变的时间(ctime)；

最近一次的读取时间(atime)；

最近修改的时间(mtime)；

定义文件特性的旗标(flag)，如 SetUID...；

该文件真正内容的指向 (pointer)；

inode 的数量与大小也是在格式化时就已经固定了，除此之外 inode 还有以下特色：

每个 inode 大小均固定为 128 bytes；

每个文件都仅会占用一个 inode；

文件系统能够建立的文件数量与 inode 的数量有关；

系统读取文件时需要先找到 inode，根据 inode 所记录的权限与用户是否符合，若符合才能读取 block 的内容。

Superblock (超级块)

Superblock 是记录整个 filesystem 相关信息的地方，没有 Superblock，就没有这个 filesystem 了。他记录的信息主要有：

* block 与 inode 的总量；

* 未使用与已使用的 inode / block 数量；

* block 与 inode 的大小 (block 为 1, 2, 4K，inode 为 128 bytes)；

* filesystem 的挂载时间、最近一次写入数据的时间、最近一次检验磁盘 (fsck) 的时间等文件系统的相关信息；

* 一个 valid bit 数值，若此文件系统已被挂载，则 valid bit 为 0 ，若未被挂载，则 valid bit 为 1 。

Superblock 是非常重要的 ,因为文件系统的基本信息都写在这里。一般来说， superblock 的大小为 1024bytes。

Filesystem Description (文件系统描述说明)

这个区段可以描述每个 block group 的开始与结束的 block 号码，以及说明每个区段 (superblock, bitmap, inodemap, data block) 分别介于哪一个 block 号码之间。

block bitmap (区块对照表)

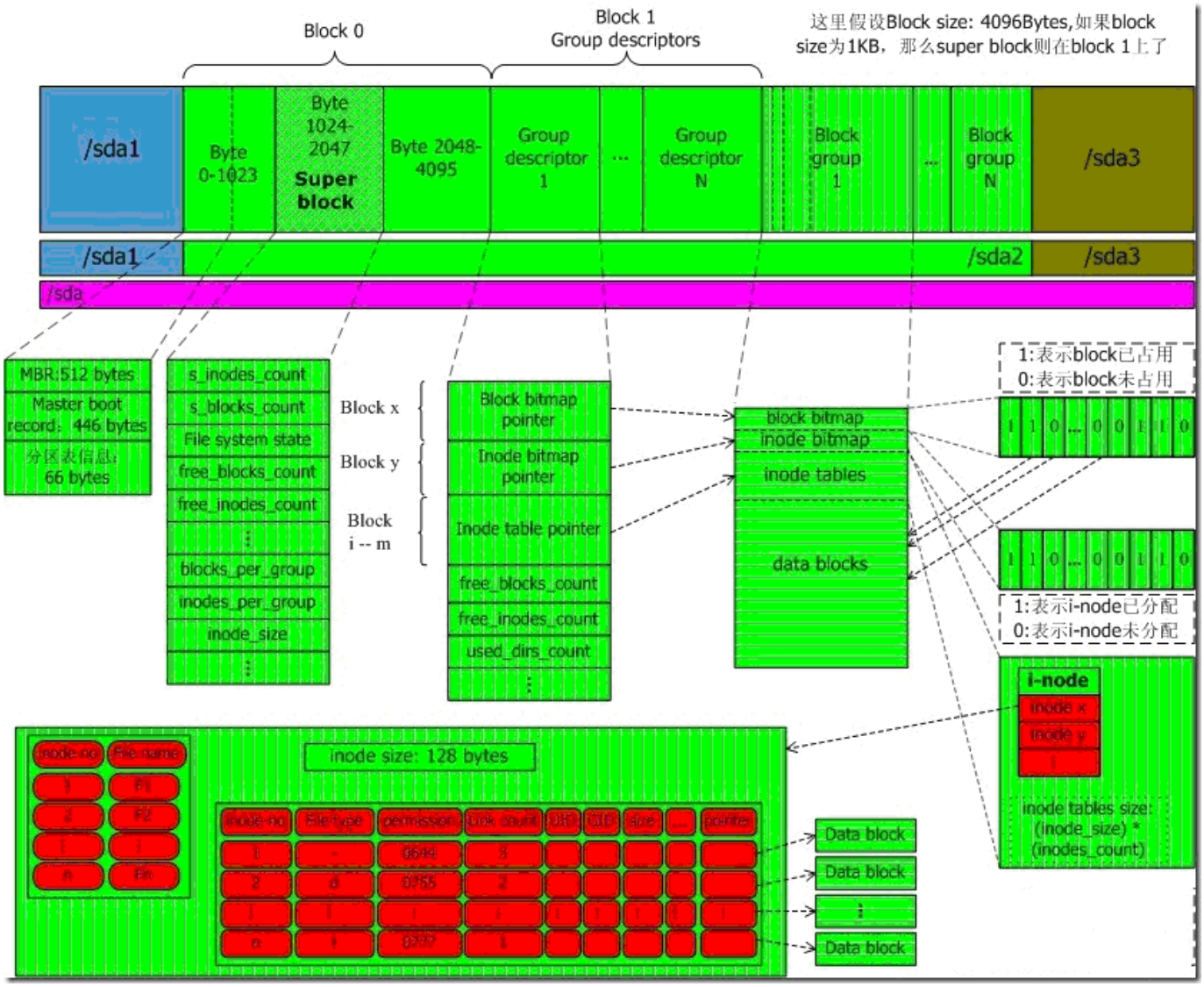
如果你想要新增文件时总会用到 block ，那要使用哪个 block 来记录呢？当然是选择空的 block 来记录新文件的数据。 那你怎么知道哪个 block 是空的？这就得要透过 block bitmap 的辅助了。从 block bitmap 当中可以知道哪些 block 是空的，因此我们的系统就能够很快速的找到可使用的空间来处文件。同样的，如果你删除某些文件时，那么那些档案原本占用的 block 号码就得要释放出来， 此时在 block bitmap 当中相对应到该 block 号码的标志就得要修改成为未使用的，这就是 bitmap 的功能。

inode bitmap (inode 对照表)

这个其实与 block bitmap 是类似的功能，只是 block bitmap 记录的是使用与未使用的 block 号码， 至于 inode bitmap 则是记录使用与未使用的 inode 号码。

最后给一张 EXT3 文件系统结构图，大家可以详细的来解读 EXT 文件系统。

注意：此图上有一点需要修正，就是 MBR 不在分区上，而是在磁盘的最开始位置！！！



五、新一代 Linux 文件系统 btrfs 简介

文件系统似乎是内核中比较稳定的部分，多年来，人们一直使用 ext2/3，目前 Linux 各发行版本采用的是 ext4 文件系统。ext 文件系统以其卓越的稳定性成为了事实上的 Linux 标准文件系统。近年来 ext2/3 暴露出了一些扩展性问题，于是便催生了 ext4 。在 2008 年发布的 Linux2.6.19 内核中集成了 ext4 的 dev 版本。 2.6.28 内核发布时，ext4 结束了开发版，开始接受用户的使用。似乎 ext 就将成为 Linux 文件系统的代名词。然而当您阅读很多有关 ext4 的文章时，会发现都不约而同地提到了 btrfs，并认为 ext4 将是一个过渡的文件系统。 ext4 的作者 Theodore Tso 也盛赞 btrfs 并认为 btrfs 将成为下一代 Linux 标准文件系统。 Oracle，IBM， Intel 等厂商也对 btrfs 表现出了极大的关注，投入了资金和人力。为什么 btrfs 如此受人瞩目呢。这便是本文想探讨的问题。

Kevin Bowling[1] 有一篇介绍各种文件系统的文章，在他看来，ext2/3 等文件系统属于“古典时期”。文件系统的新时代是 2005 年由 Sun 公司的 ZFS 开创的。ZFS 代表” last word in file system ”，意思是此后再也不需要开发其他的文件系统了。ZFS 的确带来了很多崭新的观念 ,对文件系统来讲是一个划时代的作品。如果您比较 btrfs 的特性，将会发现 btrfs 和 ZFS 非常类似。也许我们可以认为 btrfs 就是 Linux 社区对 ZFS 所作出的回应。从此往后在 Linux 中也终于有了一个可以和 ZFS 相媲美的文件系统。

btrfs 的特性

首先是**扩展性 (scalability)** 相关的特性，btrfs 最重要的设计目标是应对大型机器对文件系统的扩展性要求。Extent，B-Tree 和动态 inode 创建等特性保证了 btrfs 在大型机器上仍有卓越的表现，其整体性能而不会随着系统容量的增加而降低。

其次是**数据一致性 (data integrity)** 相关的特性。系统面临不可预料的硬件故障，Btrfs 采用 COW 事务技术来保证文件系统的一致性。 btrfs 还支持 checksum，避免了 silent corrupt 的出现。而传统文件系统则无法做到。

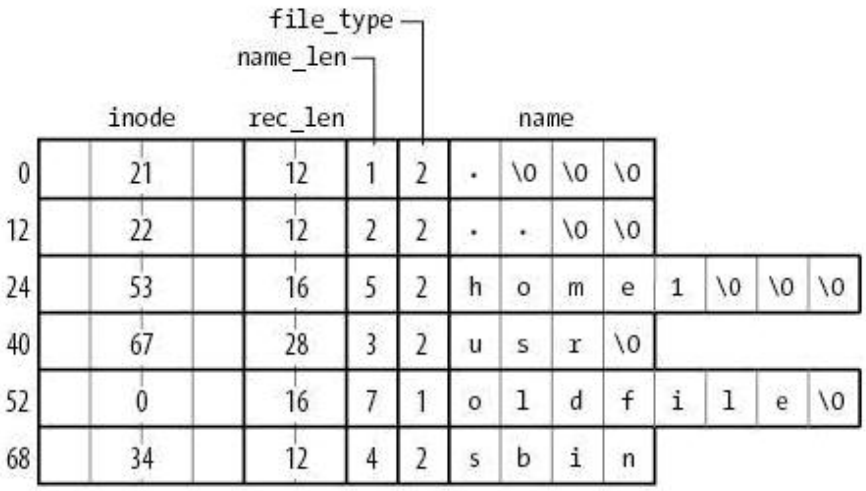
第三是和**多设备管理相关的特性。** Btrfs 支持创建快照 (snapshot)，和克隆 (clone) 。 btrfs 还能够方便的管理多个物理设备，使得传统的卷管理软件变得多余。

最后是其他难以归类的特性。这些特性都是比较先进的技术，能够显著提高文件系统的时间和空间性能，包括延迟分配，小文件的存储优化，目录索引等。

扩展性相关的特性 B-Tree

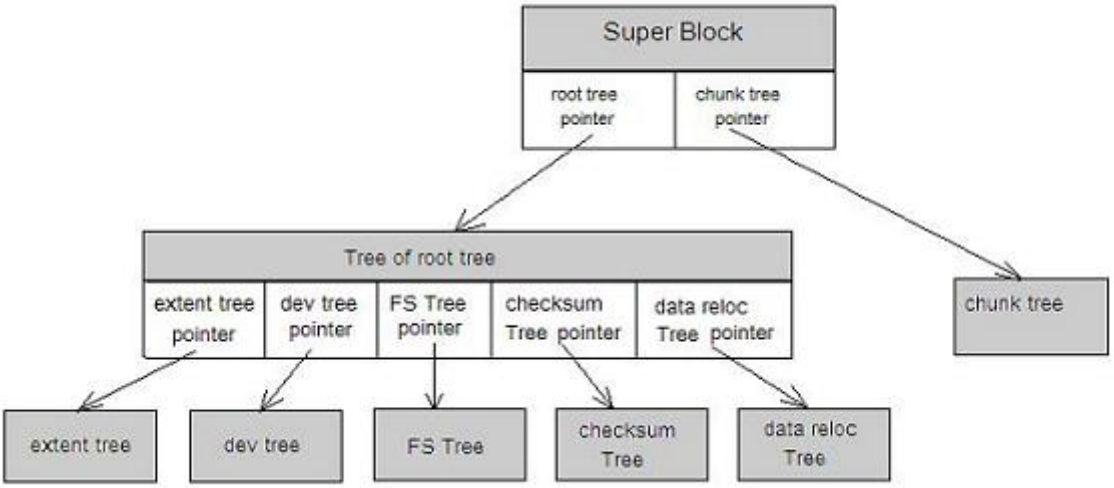
btrfs 文件系统中所有的 metadata 都由 BTree 管理。使用 BTree 的主要好处在于查找，插入和删除操作都很高效。可以说 BTree 是 btrfs 的核心。一味地夸耀 BTree 很好很高效也许并不能让人信服，但假如稍微花费一点儿时间看看 ext2/3 中元数据管理的实现方式，便可以反衬出 BTree 的优点。

妨碍 ext2/3 扩展性的一个问题来自其目录的组织方式。目录是一种特殊的文件，在 ext2/3 中其内容是一张线性表格。如下图所示：



上图展示了一个 ext2 目录文件的内容，该目录中包含四个文件。分别是 "home1"，"usr"，"oldfile" 和 "sbin" 。如果需要在该目录中查找目录 sbin，ext2 将遍历前三项，直至找到 sbin 这个字符串为止。这种结构在文件个数有限的情况下是比较直观的设计，但随着目录下文件数的增加，查找文件的时间将线性增长。 2003 年，ext3 设计者开发了目录索引技术，解决了这个问题。目录索引使用的数据结构就是 BTree 。如果同一目录下的文件数超过 2K，inode 中的 i_data 域指向一个特殊的 block 。在该 block 中存储着目录索引 BTree 。 BTree 的查找效率高于线性表，但为同一个元数据设计两种数据结构总是不太优雅。在文件系统中还有很多其他的元数据，用统一的 BTree 管理是非常简单而优美的设计。

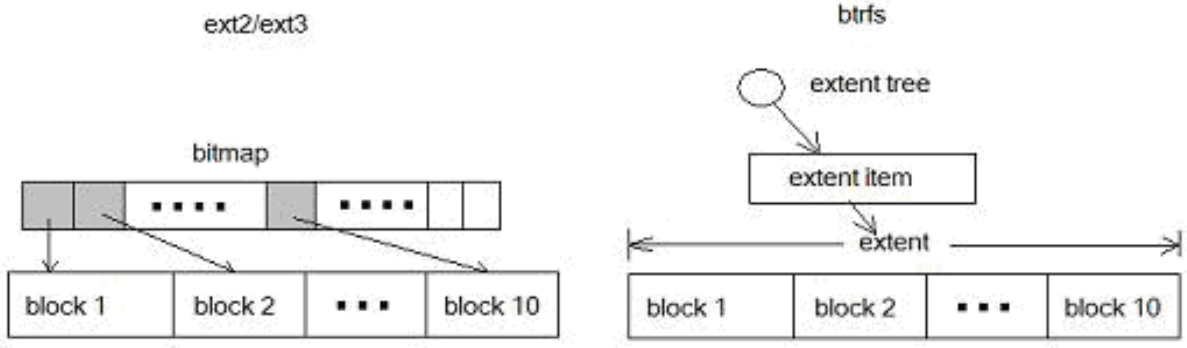
Btrfs 内部所有的元数据都采用 BTree 管理，拥有良好的可扩展性。 btrfs 内部不同的元数据由不同的 Tree 管理。在 superblock 中，有指针指向这些 BTree 的根。如下图所示：



FS Tree 管理文件相关的元数据，如 inode，dir 等； Chunk tree 管理设备，每一个磁盘设备都在 Chunk Tree 中有一个 item ； Extent Tree 管理磁盘空间分配，btrfs 每分配一段磁盘空间，便将该磁盘空间的信息插入到 Extent tree 。查询 Extent Tree 将得到空闲的磁盘空间信息； Tree of tree root 保存很多 BTree 的根节点。比如用户每建立一个快照，btrfs 便会创建一个 FS Tree 。为了管理所有的树，btrfs 采用 Tree of tree root 来保存所有树的根节点； checksum Tree 保存数据块的校验和。

基于 Extent 的文件存储

现代很多文件系统都采用了 extent 替代 block 来管理磁盘。 Extent 就是一些连续的 block，一个 extent 由起始的 block 加上长度进行定义。Extent 能有效地减少元数据开销。为了进一步理解这个问题，我们还是看看 ext2 中的反面例子。ext2/3 以 block 为基本单位，将磁盘划分为多个 block 。为了管理磁盘空间，文件系统需要知道哪些 block 是空闲的。 Ext 使用 bitmap 来达到这个目的。 Bitmap 中的每一个 bit 对应磁盘上的一个 block，当相应 block 被分配后，bitmap 中的相应 bit 被设置为 1 。这是很经典也很清晰的一个设计，但不幸的是当磁盘容量变大时，bitmap 自身所占用的空间也将变大。这就导致了扩展性问题，随着存储设备容量的增加，bitmap 这个元数据所占用的空间也随之增加。而人们希望无论磁盘容量如何增加，元数据不应该随之线形增加，这样的设计才具有可扩展性。下图比较了 block 和 extent 的区别：



在 ext2/3 中，10 个 block 需要 10 个 bit 来表示；在 btrfs 中则只需要一个元数据。对于大文件，extent 表现出了更加优异的管理性能。Extent 是 btrfs 管理磁盘空间的最小单位，由 extent tree 管理。Btrfs 分配 data 或 metadata 都需要查询 extent tree 以便获得空闲空间的信息。

动态 inode 分配

为了理解动态 inode 分配，还是需要借助 ext2/3。在 ext2 中 inode 区是被预先固定分配的，且大小固定，比如一个 100G 的分区中，inode table 区中只能存放 131072 个 inode，这就意味着不可能创建超过 131072 个文件，因为每一个文件都必须有一个唯一的 inode 。

为了解决这个问题，必须动态分配 inode 。每一个 inode 只是 BTree 中的一个节点，用户可以无限制地任意插入新的 inode，其物理存储位置是动态分配的。所以 btrfs 没有对文件个数的限制。

针对 SSD 的优化支持

SSD 是固态存储 Solid State Disk 的简称。在过去的几十年中，CPU/RAM 等器件的发展始终遵循着摩尔定律，但硬盘 HDD 的读写速率却始终没有飞跃式的发展。磁盘 IO 始终是系统性能的瓶颈。

SSD 采用 flash memory 技术，内部没有磁盘磁头等机械装置，读写速率大幅度提升。 flash memory 有一些不同于 HDD 的特性。 flash 在写数据之前必须先执行擦除操作；其次，flash 对擦除操作的次数有一定的限制，在目前的技术水平下，对同一个数据单元最多能进行约 100 万次擦除操作，因此，为了延长 flash 的寿命，应该将写操作平均到整个 flash 上。

SSD 在硬件内部的微代码中实现了 wear leveling 等分布写操作的技术，因此系统无须再使用特殊的 MTD 驱动和 FTL 层。虽然 SSD 在硬件层面做了很多努力，但毕竟还是有限。文件系统针对 SSD 的特性做优化不仅能提高 SSD 的使用寿命，而且能提高读写性能。 Btrfs 是少数专门对 SSD 进行优化的文件系统。 btrfs 用户可以使用 mount 参数打开对 SSD 的特殊优化处理。Btrfs 的 COW 技术从根本上避免了对同一个物理单元的反复写操作。如果用户打开了 SSD 优化选项，btrfs 将在底层的块空间分配策略上进行优化：将多次磁盘空间分配请求聚合成一个大小为 2M 的连续的块。大块连续地址的 IO 能够让固化在 SSD 内部的微代码更好的进行读写优化，提高 IO 性能。

多设备管理相关的特性

每个 Unix 管理员都曾面临为用户和各种应用分配磁盘空间的任务。多数情况下，人们无法事先准确地估计一个用户或者应用在未来究竟需要多少磁盘空间。磁盘空间被用尽的情况经常发生，此时人们不得不试图增加文件系统空间。传统的 ext2/3 无法应付这种需求。

很多卷管理软件被设计出来满足用户对多设备管理的需求，比如 LVM 。 Btrfs 集成了卷管理软件的功能，一方面简化了用户命令；另一方面提高了效率。

Btrfs 支持动态添加设备。用户在系统中增加新的磁盘之后，可以使用 btrfs 的命令将该设备添加到文件系统中。为了灵活利用设备空间，Btrfs 将磁盘空间划分为多个 chunk 。每个 chunk 可以使用不同的磁盘空间分配策略。比如某些 chunk 只存放 metadata，某些 chunk 只存放数据。一些 chunk 可以配置为 mirror，而另一些 chunk 则可以配置为 stripe 。这为用户提供了非常灵活的配置可能性。

Subvolume

Subvolume 是很优雅的一个概念。即把文件系统的一部分配置为一个完整的子文件系统，称之为 subvolume 。采用 subvolume，一个大的文件系统可以被划分为多个子文件系统，这些子文件系统共享底层的设备空间，在需要磁盘空间时便从底层设备中分配，类似应用程序调用 malloc() 分配内存一样。可以称之为存储池。这种模型有很多优点，比如可以充分利用 disk 的带宽，可以简化磁盘空间的管理等。所谓充分利用 disk 的带宽，指文件系统可以并行读写底层的多个 disk，这是因为每个文件系统都可以访问所有的 disk 。传统的文件系统不能共享底层的 disk 设备，无论是物理的还是逻辑的，因此无法做到并行读写。所谓简化管理，是相对于 LVM 等卷管理软件而言。采用存

储池模型，每个文件系统的大小都可以自动调节。而使用 LVM，如果一个文件系统的空间不够了，该文件系统并不能自动使用其他磁盘设备上的空闲空间，而必须使用 LVM 的管理命令手动调节。

Subvolume 可以作为根目录挂载到任意 mount 点。 subvolume 是非常有趣的一个特性，有很多应用。假如管理员只希望某些用户访问文件系统的一部分 ,比如希望用户只能访问 /var/test/ 下面的所有内容 ,而不能访问 /var/ 下面其他的内容。那么便可以将 /var/test 做成一个 subvolume 。 /var/test 这个 subvolume 便是一个完整的文件系统，可以用 mount 命令挂载。比如挂载到 /test 目录下，给用户访问 /test 的权限，那么用户便只能访问 /var/test 下面的内容了。

快照和克隆

快照是对文件系统某一时刻的完全备份。建立快照之后，对文件系统的修改不会影响快照中的内容。这是非常有用的一种技术。利用快照，管理员可以在时间点 T1 将数据库停止，对系统建立一个快照。这个过程一般只需要几秒钟，然后就可以立即重新恢复数据库服务。此后在任何时候，管理员都可以对快照的内容进行备份操作，而此时用户对数据库的修改不会影响快照中的内容。当备份完成，管理员便可以删除快照，释放磁盘空间。

快照一般是只读的，当系统支持可写快照，那么这种可写快照便被称为克隆。克隆技术也有很多应用。比如在一个系统中安装好基本的软件，然后为不同的用户做不同的克隆，每个用户使用自己的克隆而不会影响其他用户的磁盘空间。非常类似于 VM 虚拟机。Btrfs 支持 snapshot 和 clone 。这个特性极大地增加了 btrfs 的使用范围，用户不需要购买和安装昂贵并且使用复杂的卷管理软件。

软件 RAID

RAID 技术有很多非常吸引人的特性，比如用户可以将多个廉价的 IDE 磁盘组合为 RAID0 阵列，从而变成了一个大容量的磁盘； RAID1 和更高级的 RAID 配置还提供了数据冗余保护，从而使得存储在磁盘中的数据更加安全。Btrfs 很好的支持了软件 RAID，RAID 种类包括 RAID0,RAID1 和 RAID10.

Btrfs 缺省情况下对 metadata 进行 RAID1 保护。前面已经提及 btrfs 将设备空间划分为 chunk ,一些 chunk 被配置为 metadata，即只存储 metadata 。对于这类 chunk，btrfs 将 chunk 分成两个条带，写 metadata 的时候，会同时写入两个条带内，从而实现对 metadata 的保护。

压缩

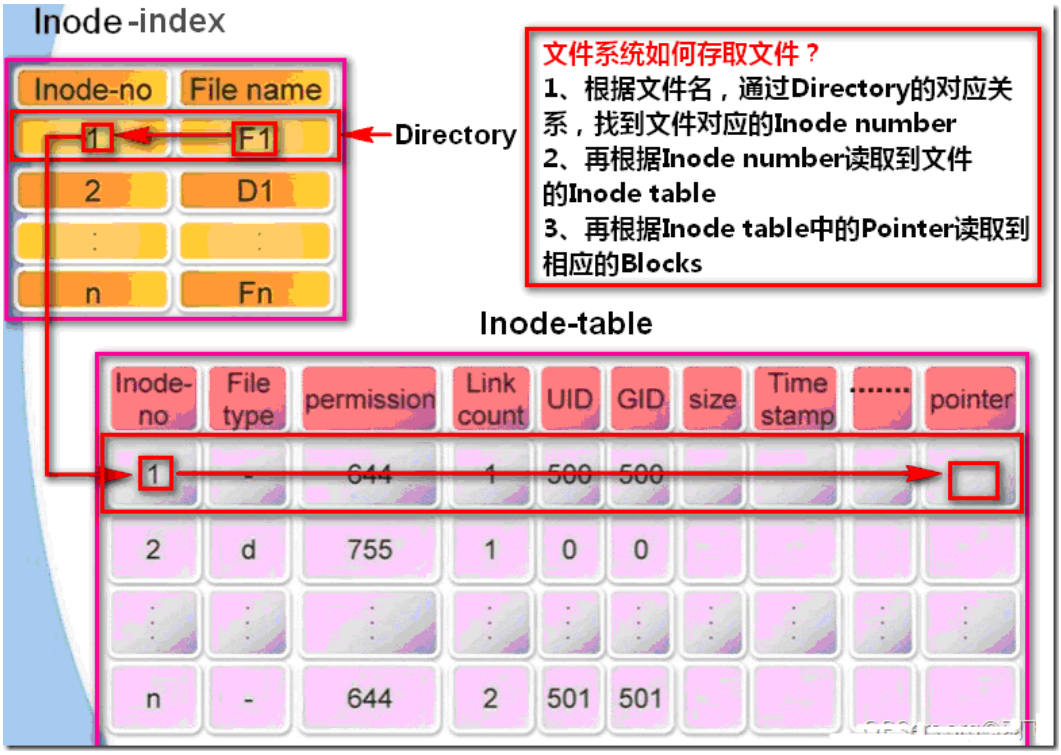
大家都曾使用过 zip，winrar 等压缩软件，将一个大文件进行压缩可以有效节约磁盘空间。 Btrfs 内置了压缩功能。通常人们认为将数据写入磁盘之前进行压缩会占用很多的 CPU 计算时间，必然降低文件系统的读写效率。但随着硬件技术的发展，CPU 处理时间和磁盘 IO 时间的差距不断加大。在某些情况下，花费一定的 CPU 时间和一些内存，但却能大大节约磁盘 IO 的数量，这反而能够增加整体的效率。

至此，我们对 btrfs 的很多特性进行了较为详细的探讨，但 btrfs 能提供的特性却并不止这些。 btrfs 正处于试验开发阶段，还将有更多的特性。Btrfs 也有一个重要的缺点，当 BTree 中某个节点出现错误时，文件系统将失去该节点之下的所有的文件信息。而 ext2/3 却避免了这种被称为" 错误扩散" 的问题。但无论怎样， btrfs 将是 Linux 未来最有希望的文件系统。

六、文件系统 inode &block 详解

在 Linux 文件系统中 ,很多人对 Inode 和 block 都不太明白 ,现在我和大家一起来先分享一下我对 Inode 的认识。首先我用一张 EXT3 文件系统结构图来进行说明吧。里面已经涉及到了 Inode 的相关信息——图的左下角部分，我们就专门对这一部分进行一下详解，希望通过今天的内容，能帮助大家更清楚的了解 Inode。

先看看 Inode 的结构图：



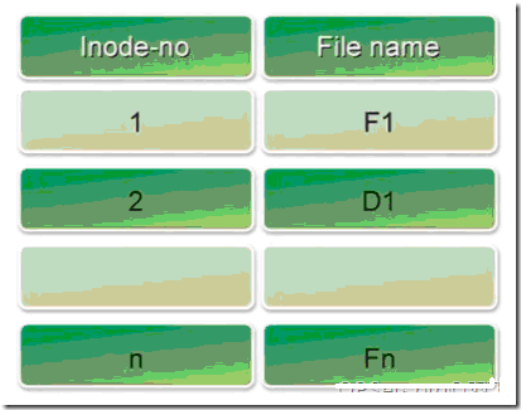
再来了解一下文件系统如何存取文件的：

1、根据文件名，通过 Directory 里的对应关系，找到文件对应的 Inode number

2、再根据 Inode number 读取到文件的 Inode table

3、再根据 Inode table 中的 Pointer 读取到相应的 Blocks

这里有一个重要的内容，就是 Directory，他不是我们通常说的目录，而是一个列表，记录了一个文件/目录名称对应的 Inode number。如下图：



Directory:

A directory is a mapping between the human name for the file and the computer’ s inode number.

所以说，这个 Directory 不是目录，我们可以看作是文件系统中的一个属性，只是用来关联文件名与 Inode number。这个一定要理解好，否则后面关于硬链接的内容，就不容易理解了。我在以前的教程中有讲过到，上图第二栏表示的是有多少文件连接到 inode

如果是一个文件，此时这一字段表示这个文件所具有的硬链接数，

如果是一个目录，则此字段表示该目录所含子目录的个数。

在 linux 系统中如果想要查看文件或目录的 inode 和 block 情况，我们可以使用 ls –li 或者 stat 命令，如下图：

```
[root@localhost ~]# stat install.log
  File: `install.log'
  Size: 35320           Blocks: 80         IO Block: 4096   regular file
Device: 803h/2051d      Inode: 1423490     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2010-03-10 00:37:35.000000000 +0800
Modify: 2010-03-10 01:13:34.000000000 +0800
Change: 2010-03-10 01:14:05.000000000 +0800
[root@localhost ~]#
```

使用 stat 命令获得的信息：除了通常的文件大小（也可以使用 ls -l 命令获得）以外，您还获得了该文件占用的块数。通常的 Linux 块(也就是 block size)大小为 512 字节，因此一个大小为 35320 字节的文件将占用(35320/512=) 68.98 个块。由于块都是完整占用，因此该文件使用了一些整数个数的块。无需猜测就可以获得确切的块数。

您还可以从以上输出中获得文件所有者的 GID 和 UID，以及权限的八进制表示形式 (0644/-rw-r--r--)。以上输出最有用的部分是文件访问时间戳信息，该文件被访问的时间是 2010-03-10 00:37:35(显示在“Access:”的旁边)，这是某个用户开始使用的时间。该文件的修改时间是 2010-03-10 01:13:34（显示在“Modify:”的旁边）。最后，“Change:”旁边的时间戳显示文件状态更改的时间。

stat 命令的一个重要参数 -f 显示了有关文件系统（而非文件）的信息：

```
[root@localhost ~]# stat -f install.log
  File: "install.log"
    ID: 0        Namelen: 255     Type: ext2/ext3
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 1879050    Free: 1226169    Available: 1129177
Inodes: Total: 1941120    Free: 1817236
[root@localhost ~]# _
```

现在是不是容易理解了？如果你还不是很明白，那么下面我们就在 RHEL 系统下的实例让大家明白。

Example 1 : inode 和软、硬链接

在根目录下创建一个 test 目录，我们进入此目录，进行如下操作。

```
[root@hubert ~]# mkdir test
[root@hubert ~]# cd test
[root@hubert test]#
[root@hubert test]# touch testfile
[root@hubert test]# mkdir testdir
[root@hubert test]#
[root@hubert test]# ls -li
total 12
1423646 drwxr-xr-x 2 root root 4096 Mar 10 20:25 testdir
1423645 -rw-r--r-- 1 root root    0 Mar 10 20:25 testfile
[root@hubert test]# _
```

查看到文件与目录的 Inode 和 inode count 分别为：

1423646 <——> 2 <——> testdir

1423645 <——> 1 <——> testfile

现在目录的链接数为 2，文件的链接数为 1。

为什么会这样呢？其实很好理解。对于目录而言，每个目录下肯定会有两个特殊目录，那就是.和..这两个目录，我们前面的课程中也讲到，.表示当前的目录，而..则是表示上层目录。我们也知道，在 Linux 系统中，是从根来开始查找的，要想找到某个目录，必需要先找到他的上层目录，所以说，空目录（严格的来说，不能叫空目录）是有两个链接到相应的 Inode number 的。作为文件很明显，他只有一个链接到相应的 Inode number。

下面我们就来看看这个链接数是如何改变的。继续上面的操作：

```
[root@hubert test]# ln testfile testfile.hard
[root@hubert test]# ln -s testfile testfile.soft
[root@hubert test]#
[root@hubert test]# ls -li
total 20
1423646 drwxr-xr-x 2 root root 4096 Mar 10 20:25 testdir
1423645 -rw-r--r-- 2 root root    0 Mar 10 20:25 testfile
1423645 -rw-r--r-- 2 root root    0 Mar 10 20:25 testfile.hard
1423647 lrwxrwxrwx 1 root root    8 Mar 10 20:28 testfile.soft -> testfile
[root@hubert test]#
```

如上图所示，使用 ln 命令对文件 testfile 建立一个硬链接和一个软链接，再 ls -li 查看文件和目录的属性。这时我们就发现，创建一个硬链接后，testfile 的 inode count 增加了一个。而且 testfile 和 testfile.hard 这两个的 Inode number 是一样的都是 1423645。这个硬链接就是重新创建了一个文件名对应到原文件的 Inode，实质就是在 Directory 中增加了一个新的对应关系。通过这个例子，你是不是更清楚了，这个 Inode count 的含义了。他就是指，一个 Inode 对应了多少个文件名。

Example 2：硬链接和软链接的特点

我们先来使用 watch 命令实时查看一下系统中所剩的 block 和 inode 的变化数量。

建议大家不要用 deumpe2fs 和 tune2fs 这两个命令，如果使用他们来查看的话，将会很郁闷——你会发现，你无论怎么创建文件或对文件写入内容，Inode 和 block 的值都不会变，除非你每操作一次，就重新启动一次系统，而用了上面的命令，就是实时监视他们的变化情况。关于 df 的命令使用，大家可以自行查看帮助进行学习。当然还有 du 这个命令，他们都和文件系统有关。

```
[root@hubert test]# watch -n 1 "df -i;df"


Every 1.0s: df -i;df                                    Wed Mar 10 20:37:15 2010

Filesystem            Inodes   IUsed   IFree IUse% Mounted on
/dev/sda3            1941120  123884 1817236    7% /
/dev/sda1              26104      34   26070    1% /boot
tmpfs                  31930       1   31929    1% /dev/shm
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda3              7516200   2611488   4516744  37% /
/dev/sda1               101086     11375     84492  12% /boot
tmpfs                   127720         0    127720   0% /dev/shm
```

我们再来 ln 创建一个硬链接，进行以下操作：

```
[root@hubert test]# ls -li
total 20
1423646 drwxr-xr-x 2 root root 4096 Mar 10 20:25 testdir
1423645 -rw-r--r-- 2 root root    0 Mar 10 20:25 testfile
1423645 -rw-r--r-- 2 root root    0 Mar 10 20:25 testfile.hard
1423647 lrwxrwxrwx 1 root root    8 Mar 10 20:28 testfile.soft -> testfile
[root@hubert test]#
[root@hubert test]# ln testfile testfile.hard1
[root@hubert test]#
[root@hubert test]# ls -li
total 24
1423646 drwxr-xr-x 2 root root 4096 Mar 10 20:25 testdir
1423645 -rw-r--r-- 3 root root    0 Mar 10 20:25 testfile
1423645 -rw-r--r-- 3 root root    0 Mar 10 20:25 testfile.hard
1423645 -rw-r--r-- 3 root root    0 Mar 10 20:25 testfile.hard1
1423647 lrwxrwxrwx 1 root root    8 Mar 10 20:28 testfile.soft -> testfile
[root@hubert test]#
```

可以再 ls -li 观察一下 Inode count 和 Inode number 的对应关系。

下面再用命令# watch -n 1 “df -i;df” 看看 inodes 和 blocks 的变化：

```
Every 1.0s: df -i;df                                    Wed Mar 10 20:46:20 2010

Filesystem            Inodes   IUsed   IFree IUse% Mounted on
/dev/sda3            1941120  123884 1817236    7% /
/dev/sda1              26104      34   26070    1% /boot
tmpfs                  31930       1   31929    1% /dev/shm
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda3              7516200   2611488   4516744  37% /
/dev/sda1               101086     11375     84492  12% /boot
tmpfs                   127720         0    127720   0% /dev/shm
```

我们发现，inodes 和 blocks 是没有减少的，所以说，硬链接是不会占用磁盘的空间的。如果说删除硬链接的话，就会改变 Inode count 的数量。

硬链接还有其他的两个特性：不能跨 Filesystem 也不能 link 目录。

下面再来看看这个软链接。

```
[root@hubert test]# ls -il testfile.soft testfile
1423645 -rw-r--r-- 3 root root 0 Mar 10 20:25 testfile
1423647 lrwxrwxrwx 1 root root 8 Mar 10 20:28 testfile.soft -> testfile
[root@hubert test]#
```

他的 Inode number 和原文件不一样。而且大小也发生了变化。可见，这个软链接是重新建立了一个文件，而文件是指向到原文件，而不是指向原 Inode。当然他会占用掉 inode 与 block。当我们删除了源文件后，链接文件不能独立存在，虽然仍保留文件名，但我们却不能查看软链接文件的内容了。但软链接是可以跨文件系统，而且是可以链接目录。他就相当于 windows 系统下的快捷方式一样。通过这个特性，我们可以通过软链接解决某个分区 inode conut 不足的问题(软链接到另一个 inode count 足够多的分区)。

Example：复制文件、移动文件和删除文件对 inode 的影响

我们先记录以下的信息：

```
[root@hubert test]# watch -n 1 "df -i;df"

Every 1.0s: df -i;df                                    Wed Mar 10 20:50:08 2010

Filesystem            Inodes   IUsed   IFree IUse% Mounted on
/dev/sda3            1941120  123884 1817236    7% /
/dev/sda1              26104      34   26070    1% /boot
tmpfs                  31930       1   31929    1% /dev/shm
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda3              7516200   2611488   4516744  37% /
/dev/sda1               101086     11375     84492  12% /boot
tmpfs                   127720         0    127720   0% /dev/shm
root@hubert test]# ls -li
otal 24
423646 drwxr-xr-x 2 root root 4096 Mar 10 20:25 testdir
423645 -rw-r--r-- 3 root root    0 Mar 10 20:25 testfile
423645 -rw-r--r-- 3 root root    0 Mar 10 20:25 testfile.hard
423645 -rw-r--r-- 3 root root    0 Mar 10 20:25 testfile.hard1
423647 lrwxrwxrwx 1 root root    8 Mar 10 20:28 testfile.soft -> testfile
root@hubert test]#
```

先看复制文件的情况

```
[root@hubert test]# cp testfile testfile.cp
[root@hubert test]#
[root@hubert test]# ls -li
total 28
1423646 drwxr-xr-x 2 root root 4096 Mar 10 20:25 testdir
1423645 -rw-r--r-- 3 root root    0 Mar 10 20:25 testfile
1423648 -rw-r--r-- 1 root root    0 Mar 10 20:55 testfile.cp
```

我们只对比这两个文件，发现 Inode number 不一样，我们再来看看 inodes 和 blocks 的剩余情况：

```
Every 1.0s: df -i;df                                    Wed Mar 10 20:54:12 2010

Filesystem            Inodes   IUsed   IFree IUse% Mounted on
/dev/sda3            1941120  123885 1817235    7% /
/dev/sda1              26104      34   26070    1% /boot
tmpfs                  31930       1   31929    1% /dev/shm
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda3              7516200   2611488   4516744  37% /
/dev/sda1               101086     11375     84492  12% /boot
tmpfs                   127720         0    127720   0% /dev/shm
```

发现 inodes 减少了一个，而 blocks 也少了，这就说明，复制文件是创建文件，并占 Inode 和 Block 的。文件创建过程是：先查找一个空的 Inode，写入新的 Inode table，创建 Directory，对应文件名，向 block 中写入文件内容。

关于移动文件和删除文件的实验，大家可以自己动手来实践吧。我直接给出相应的说明。

移动文件，分两种情况：

在同一个文件系统中移动文件时：创建一个新的文件名和 Inode 的对应关系（也就是在 Directory 中写入信息），然后在 Directory 中删除旧的信息，更新 CTIME，其他的信息如 Inode 等等均无任何影响。

在不同文件系统移动文件时：先查找一个空的 Inode，写入新的 Inode table，创建 Directory 中的对应关系，向 block 中写入文件内容，同时还会更改 CTIME。

删除文件：实质上就是减少 link count，当 link count 为 0 时，就表示这个 Inode 可以使用，并把 Block 标记为可以写,但并没有清除 Block 里面数据，除非是有新的数据需要用到这个 block。

inode 总结：

1、一个 Inode 对应一个文件，而一个文件根据其大小，会占用多块 blocks。

2、更为准确的来说，一个文件只对应一个 Inode。因为硬链接其实不是创建新文件，只是在 Directory 中写入了新的对应关系而已。

3、当我们删除文件的时候，只是把 Inode 标记为可用，文件在 block 中的内容是没有被清除的，只有在有新的文件需要占用 block 的时候，才会被覆盖。

文件系统相关命令总结：

#stat [-f] [filename]

#df –l [devices]

#mkfs [-t] [filesystem] [devices]

#tune2fs –l [devices]

#deumpe2fs

OK OK!!! 讲完了 inode 再来研究一下更加让人头疼的 block 吧！GO~~~

首先再次复习一下 inode 和 block 的关系：当我们在 linux 系统下面创建一个文件时，文件系统通常会将这两部份的数据分别存放在不同的区块 权限与属性放到 inode 中 数据则放到 block 区块中。另外 还有一个超级区块(super block)会记录整个文件系统的整体信息，包括 inode 与 block 的总量、使用量、剩余量等等等。

通常 Linux 的“block size”指的是 1024 bytes，Linux 用 1024-byte blocks 作为 buffer cache 的基本单位。但 linux 的不同文件系统的 block 却都不相同，这个就比较的纠结了，也是很容易把人搞晕的地方。例如 RHEL5 所使用的 ext3 文件系统中，block size 有三种，分别是 1K、2K、4K。使用 tune2fs 命令或 dumpe2fs 命令能够查看带文件系统的磁盘分区的相关信息，包括 block size。如下图所示：

```
[root@localhost ~]# fdisk -l

Disk /dev/sda: 8589 MB, 8589934592 bytes
255 heads, 63 sectors/track, 1044 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14          78      522112+  82  Linux swap / Solaris
/dev/sda3              79        1044     7759395   83  Linux
[root@localhost ~]#
[root@localhost ~]# tune2fs -l /dev/sda1 |grep "Block size"
Block size:               1024
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]# tune2fs -l /dev/sda3 |grep "Block size"
Block size:               4096
```

在上图中，我的 RHEL 系统/dev/sda1 是/boot 分区，Block size＝1K＝1024byte。而/dev/sda3 是/根分区，Block size=4K=4096byte。

其实本来这几个概念不是很难，主要是玛尼隔壁的他们的名字都相同，都叫“Block Size”，纠结啊~~~。

1. 物理硬盘上的 block size，应该是"sector size"，linux 的扇区大小是 512byte
2. 磁盘分区的"cylinder size"，用 fdisk 能够查看。
3. 有文件系统的分区的 block size，叫做"block size"，不同文件系统大小不一样，能够用 tune2fs 命令查看
4. 没有文件系统的分区的 block size，也叫“block size”，大小指的是 1024 byte

也就是说 ,磁盘里面的和文件系统里面的两个都叫 block size ,但是大小是不一样的 ,磁盘里面一般都是 512bytes, 文件系统里面的不一定是 512bytes 了。举例说明：

```
[root@localhost ~]# fdisk -l /dev/sda

Disk /dev/sda: 8589 MB, 8589934592 bytes
255 heads, 63 sectors/track, 1044 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14          78      522112+  82  Linux swap / Solaris
/dev/sda3              79        1044     7759395   83  Linux
```

红色的是磁盘里面的一个 block 的大小=512byes，蓝色的是文件系统里面的 block 的个数。在上面又有讲 ext 文件系统 block size 有 1K、2K、4K。

犯晕时：

sda1 的大小计算：13*8225280/1024/1024=101MB　104391*512/1024/1024=50MB　少一半 囧了 SB 了。明白之后：

#tune2fs -l /dev/sda1 | grep Block

sda1 的大小计算：13*8225280/1024/1024=196MB　104391*1024/1024/1024=101MB　对上了 OK 了。

blocak size 的计算还是有必要搞清楚滴，因为文件系统调优的时候会用到，还是弄明白的好啊。

七、文件系统优化

相关命令：

 # mkfs.ext3 [device] 把分区格式化成 ext3 文件系统

[-b block-size] [-i bytes-per-inode] [-m reserved-blocks-percentage]

-b <block size> 指定块大小，可以是 1024\2048\4096

-i <bytes-per-indoe> 指定每多少字节使用一个索引节点 inode，单位 byte

-m <percentage> 保留块的百分比

#tune2fs [device] 是一个调整 ext2/ext3 文件系统的工具

[-l] [-c max-mount-counts] [-i interval-between-checks[d|m|w]] [-m reserved-blocks-percentage] [-L volume-label] [-r reserved-blocks-count]

-l <device> 查看文件系统详细信息

-c <count> 设置强制自检的挂载次数，开启后每挂载一次 mount conut 就会加 1，超过次数后强制自检

-i <n day> 设置强制自检的时间间隔[d 天 m 月 w 周]

-m <percentage> 保留块的百分比

-j 将 ext2 文件系统转换为 ext3 类型的文件系统

-L <volume-label> 类似 e2label 的功能，可以修改文件系统的标签

-r 调整系统保留空间

#du –sh [file|dir] 查看文件或目录所占磁盘空间的大小

1、 block size 优化

在 linux 系统中，硬盘最小的存储单位是 block，硬盘的每个分区被格式化后，都会创建很多个 block 用于存储数据。在 RHEL 系统使用的 ext3 文件系统中 block 的大小可以是 1K、2K、4K。

```
[root@localhost ~]# tune2fs -l /dev/sda1 |grep "Block size"
Block size:               1024
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]# tune2fs -l /dev/sda3 |grep "Block size"
Block size:               4096
```

如上图所示，/dev/sda1 分区的 block size = 1024byte=1K ,而另一个/dev/sda3 分区的 block size = 4096=4K，这个就比较滴纠结了~~~都是 linux 下的 2 个不同的分区但是 block size 却不一样！!!

假设 block size = 1K ，如果一个文件有 4K，要使用 4 个 block；即使一个文件不到 1K，也会占用 1 个 block。

假设 block size = 4K，如果一个文件有 4K，只占用一个 block；即使一个文件不到 4K，也会占用 1 个 block，直到填满才使用下一个 block。

这样我们就可以得出一个结论 ,如果你的文件很大 ,而 block size 很小 ,这个文件就会被分割成很小的很多 block，这样分割的时间和寻址的时间都会花费比较多的时间导致磁盘读写效率很低，相反如果 block size 大点就会减少相应的时间！但并非 block 越大越好！如果你的 block 大小是 4K，而你的文件只有 1K，这样就会浪费 3/4 的磁盘空间。

也就是说如果文件很大 ,但是 block 很小 ,则会影响读取的速度 ;如果文件很小 ,但是 block 很大 ,则会浪费 block。鱼和熊掌不可兼得，无限纠结啊~~~所以在 linux 中如果既要提高磁盘读写效率，又要合理利用磁盘空间而不导致浪费的话，最好的方案就是根据分区类型和分区存储的文件的大小来来手动指定 block size 的大小！!!

Example：

在 VM 中添加一块新硬盘/dev/sdb，并划分 2 个大小都是 1000MB 的分区/dev/sdb1 和/dev/sdb2。

```
[root@hubert ~]# fdisk -l /dev/sdb

Disk /dev/sdb: 8589 MB, 8589934592 bytes
255 heads, 63 sectors/track, 1044 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1         123      987966   83  Linux
/dev/sdb2             124         246      987997+  83  Linux
[root@hubert ~]#
[root@hubert ~]# tune2fs -l /dev/sdb1 |grep "Block size"
Block size:               1024
```

然后对 sdb1 和 sdb2 进行格式化，并指定 block size，其中 sdb1 block size=1024；sdb2 block size=4096。

```
[root@hubert ~]# mkfs.ext3 -b 1024 /dev/sdb1
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
123904 inodes, 987964 blocks
49398 blocks (5.00%) reserved for the super user

[root@hubert ~]# mkfs.ext3 -b 4096 /dev/sdb2
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
123648 inodes, 246999 blocks
12349 blocks (5.00%) reserved for the super user
```

接下来，比较一下 sdb1 和 sdb2 不同的 block size 对磁盘使用的影响。操作如下

```
[root@hubert ~]# mkdir /mnt/sdb1 /mnt/sdb2
[root@hubert ~]# mount /dev/sdb1 /mnt/sdb1
[root@hubert ~]# mount /dev/sdb2 /mnt/sdb2
[root@hubert ~]#
[root@hubert ~]# mkdir /mnt/sdb1/dir1
[root@hubert ~]# mkdir /mnt/sdb2/dir2
[root@hubert ~]#
[root@hubert ~]# echo "hubert" > /mnt/sdb1/file1
[root@hubert ~]# echo "hubert" > /mnt/sdb2/file2
[root@hubert ~]#
[root@hubert ~]# ls -lh /mnt/sdb[12]
/mnt/sdb1:
total 4.0K
drwxr-xr-x 2 root root 1.0K Mar 10 22:26 dir1
-rw-r--r-- 1 root root    7 Mar 10 22:26 file1

/mnt/sdb2:
total 16K
drwxr-xr-x 2 root root 4.0K Mar 10 22:26 dir2
-rw-r--r-- 1 root root    7 Mar 10 22:26 file2
[root@hubert ~]#
```

通过上面操作可以观察到，在 sdb1 和 sdb2 下分别创建的两个文件 file1\file2 的大小都是 7byte，但是两个目录 dir1 和 dir2 的大小却不同，一个是 1k 另一个是 4k，再使用 du 命令看一下两个目录 dir1 和 dir2 占用磁盘空间大小：

```
[root@hubert ~]# du -sh /mnt/sdb1/dir1
2.0K    /mnt/sdb1/dir1
[root@hubert ~]# du -sh /mnt/sdb2/dir2
8.0K    /mnt/sdb2/dir2
```

为什么目录 dir1 占用 2 K,而目录 dir2 占用 8K 呢？？？因为 sdb1 的 block size=1024byte=1K ,所以 dir1 和 file1 两个占用磁盘空间就是 2K；而 sdb2 的 block size=4096byte=4K，所以 dir2 和 file2 两个占用磁盘空间就是 8 K 了。

2、Inode 优化

通过上一章的内容我们已经知道，文件系统中 inode 的作用是记录文件的属性以及该文件数据是放置在哪几号 block 内。inode 的数量与大小也是在格式化时就已经固定了；每个文件都仅会占用一个 inode；每个 inode 大小均固定为 128bytes；文件系统能够建立的文件数量与 inode 的数量有关；系统读取文件时需要先找到 inode，根据 inode 所记录的权限与用户是否符合，若符合才能读取 block 的内容。

在文件系统调优中，如果 inode 和 block 一致，对于小文件来说更优化，但是整个系统所能存储的文件数较少。如果改变 block size 使用多个 block 对应一个 inode 对于大文件来说更优化 ,可以最大限度节省磁盘空间 ,节省 inode 数量，整个文件系统所能存储的文件数就较多。

Example：

一个 inode 对应一个 block：

如下图所示，把 sdb1 指定 block size=1K，并且每 1 个 block 占用 1 个 inode 记录，这样有利于小文件存储。

```
[root@hubert ~]# mkfs.ext3 -b 1024 -i 1024 /dev/sdb1
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
988328 inodes, 987964 blocks
49398 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=68157440
121 block groups
8192 blocks per group, 8192 fragments per group
8168 inodes per group
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729, 204801, 221185, 401409,

[root@hubert ~]# tune2fs -l /dev/sdb1 |grep " count"
Inode count:              988328
Block count:              987964
```

Hubert

一个 inode 对应多个 block：

如下图所示，把 sdb2 指定 block size=4K，并且每 4 个 block 占用一个 inode 记录，这样有利于大文件存储

```
[root@hubert ~]# mkfs.ext3 -b 1024 -i 4096 /dev/sdb2
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
247808 inodes, 987996 blocks
49399 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=68157440
121 block groups
8192 blocks per group, 8192 fragments per group
2048 inodes per group
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729, 204801, 221185, 401409,

[root@hubert ~]# tune2fs -l /dev/sdb2 |grep " count"
Inode count:              247808
Block count:              987996
```

Hubert

3、保留块（reserved blocks）

保留块其实就是为系统管理员保留的磁盘管理空间，为防止磁盘空间剩余太少以致管理员无法编辑配置文件而设置的，默认是总数据块的 5%。在 mkfs.ext3 格式化时可以看到默认的保留块 5%，如下图所示：

```
[root@hubert ~]# mkfs.ext3 -b 1024 -i 4096 /dev/sdb1
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
247808 inodes, 987964 blocks
49398 blocks (5.00%) reserved for the super user
```

总inode数=247808，总block数=987564

为超级用户保留总block数的5%，也就是987564*0.05=49398

用 tune2fs 命令也可以查看到这 5%的保留块信息：

```
[root@hubert ~]# tune2fs -l /dev/sdb1 |grep Reserved
Reserved block count:     49398
Reserved GDT blocks:      256
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
[root@hubert ~]#
```

Hubert

Example：

修改 sdb2 保留块的数量占总数的 10%，再用 tune2fs 查看结果。

```
[root@hubert ~]# mkfs.ext3 -b 1024 -i 4096 -m 10 /dev/sdb2
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
247808 inodes, 987996 blocks
98799 blocks (10.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=68157440
121 block groups
8192 blocks per group, 8192 fragments per group
2048 inodes per group
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729, 204801, 221185, 401409, 663553

[root@hubert ~]#
[root@hubert ~]# tune2fs -l /dev/sdb2 |grep Reserved
Reserved block count:     98799
Reserved GDT blocks:      256
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
```

保留总块数的10%

Hubert

4、调整 ext2/ext3 文件系统强制自检的挂载次数

先查看一下没挂载过的分区 sdb1，显示挂载次数为 0,最大 20，如下图

```
[root@hubert ~]# tune2fs -l /dev/sdb1 |grep " count"
Inode count:              247808
Block count:              987964
Reserved block count:     49398
Mount count:              0
Maximum mount count:      20
[root@hubert ~]#
```

Hubert

挂载后再查看，

```
[root@hubert ~]# mount /dev/sdb1 /mnt
[root@hubert ~]#
[root@hubert ~]# tune2fs -l /dev/sdb1 |grep " count"
Inode count:              247808
Block count:              987964
Reserved block count:     49398
Mount count:              1
Maximum mount count:      20
```

上图得知挂载过 1 次了，达到 20 次的时候就会强制自检。

可使用命令 tune2fs -c 来修改强制挂载最大次数，-i 可以修改强制自检的时间间隔，-m 可修改保留块。

```
[root@hubert ~]# tune2fs -c 3 -i 7 -m 10 /dev/sdb1
tune2fs 1.39 (29-May-2006)
Setting maximal mount count to 3
Setting interval between checks to 604800 seconds
Setting reserved blocks percentage to 10% (98796 blocks)
[root@hubert ~]#
[root@hubert ~]# tune2fs -l /dev/sdb1 |grep "Maximum mount count"
Maximum mount count:      3
[root@hubert ~]# tune2fs -l /dev/sdb1 |grep "Check interval"
Check interval:           604800 (1 week)
[root@hubert ~]# tune2fs -l /dev/sdb1 |grep "Reserved block count"
Reserved block count:     98796
[root@hubert ~]# _
```

挂载3次，或每隔7天 进行强制自检，保留块10%

注意这个 Maximum mount count 值只在系统启动过程中起作用，把该值设为负值就永远不会自检了。或者把-i 设置为 0 也不会自检了。如果这样修改：tune2fs -c -1 -i 0 /dev/sdb1 挂载次数和时间都不能让它自检了，要想自检只能手工的 fsck 了。

5、文件系统检查工具 fsck&e2fsck

fsck 和 e2fsck 有点危险，识别 ext2 和 ext3 上有误差，尽量不用。fsck 是检查文件系统完整性的工具，并自动修复。fsck 和 mkfs 一样有两种用法：fsck -t ext2 /dev/sdb1 或者 fsck.ext2 /dev/sdb1。

linux 开机时会自动运行/etc/fstab 文件中的最后一个参数就是 fsck

```
[root@hubert ~]# cat /etc/fstab
LABEL=/                 /                       ext3    defaults        1 1
LABEL=/boot             /boot                   ext3    defaults        1 2
tmpfs                   /dev/shm                tmpfs   defaults        0 0
devpts                  /dev/pts                devpts  gid=5,mode=620  0 0
sysfs                   /sys                    sysfs   defaults        0 0
proc                    /proc                   proc    defaults        0 0
LABEL=SWAP-sda2         swap                    swap    defaults        0 0
[root@hubert ~]# _
```

如果想要关闭文件系统的 fsck 功能，只需要修改/etc/fstab 文件，把最后一个参数修改为 0 就可以了。