openEuler内核编程

课程讲稿

第九章 第4讲

《**ext4文件系统详解**》

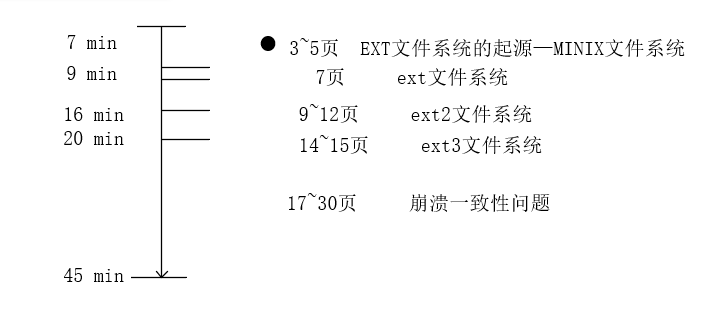
软件所制

第九章 第四讲 ext4文件系统详解

**学时：**2学时

**教学目的：** 了解ext文件系统的演化过程，了解崩溃一致性问题以及为解决这个问题提出的解决方法。了解ext4文件系统的数据结构、磁盘布局以及一些最新的文件系统

**课程时间线：**



**课外参考读物：**

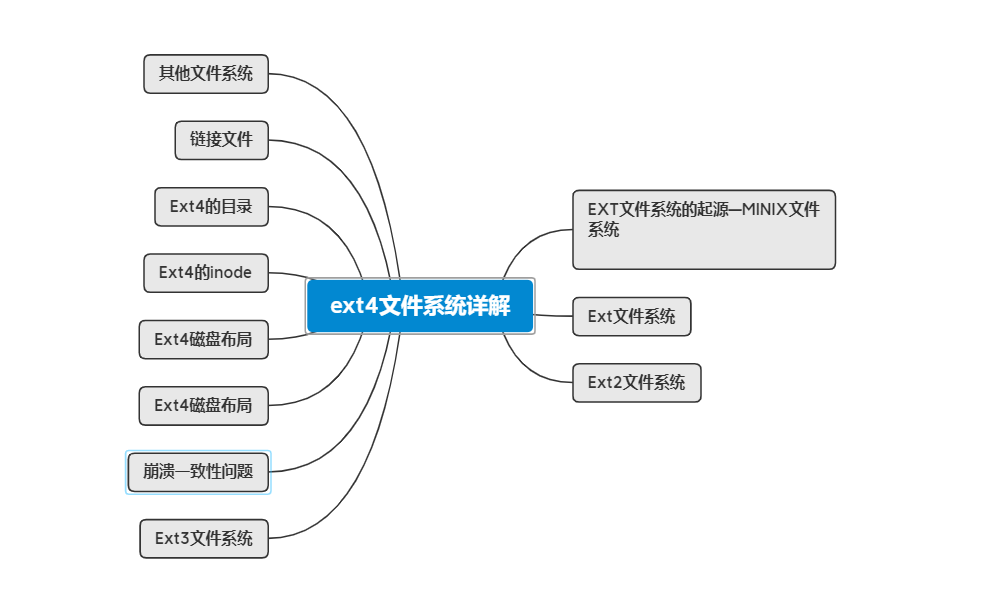
**1. https://www.jianshu.com/p/3355a35e7e0a**

**2. https://linux.cn/article-10000-1.html**

**3. 操作系统导论**

**4. https://ty-chen.github.io/linux-kernel-fs/**

**知识框图：**

****

**PPT讲稿：**



略



这一节和大家聊聊ext文件系统的前世今生，然后讲一下崩溃一致性问题，接着讲一下ext4文件系统以及一些新兴的文件系统



首先我们简要讲一下minix文件系统，minix文件系统是MINIX操作系统中的原生文件系统，安德鲁·斯图尔特·塔能鲍姆教授于1987年伴随着Minix 1.0版一同发布。当编写原始Linux内核，Linus Torvalds需要一个文件系统，但是不想费力开发。所以林纳斯就简单使用了Minix文件系统用于最早的Linux初代版本。Minix是一个类Unix操作系统，为教育使用而开发。MINIX代码开放使用，保存了操作系统的基本属性，在学术界获得较大范围的普及，用于介绍操作系统的基本实现。

MINIX-FS 将文件系统分成了 BLOCK\_SIZE 大小的块，以此为 MINIX-FS 的最小原数据。 BLOCK\_SIZE 的大小为 1K Byte，也可修改为其他值。MINIX-FS 由 6 大部分组 成，缺一不可。每个部分负责管理和维护文件系统的基础数据和功能功能，使 操作系统其能够实现一个文件系统的读，写，同步等操作，也为实现数据的管理 和存储提供了可能。

Boot 块主要存储 Boot Loader 程序，如果 Boot Loader 不存在，那么 Boot 分区 基本不使用。如果 MINIX-FS 正好是磁盘的第一个分区，那么 Boot 块里会包含整个 磁盘的分区信息。

Superblock 块为 MINIX-FS 的第二个块，其包含了 MINIX-FS 的 Super block 信 息，该信息用于描述 MINIX-FS 文件系统，其中包含了 MINIX-FS 的 MAGIC 信息， 文件系统大小和各部分大小信息。

Inode-BitMap 块是一个 BitMap 的位图，位图中每个位用于跟踪一个 inode 的使 用情况。操作系统可以通过 Inode-BitMap 块管理 MINIX-FS 的 inode。 Inode-BitMap 起始于 MINIX-FS 的第三个块，但其长度有文件系统的大小而定。

Zone-BitMap 块也是一个 BitMap 的位图，位图中的每个位用于跟踪一个 zone 的使用情况。MINIX-FS 将文件系统分作 BLOCK\_SIZE 大小的块，每个块称作 Zone。 Zone 用于存储数据，其大小由文件系统的大小而定。

Inode-Table 是一个典型的数组，数组由 minix\_inode 构成，每个 minix\_inode 用于管理文件系统中的一个目录或文件。数组的索引有 inode 的 ino 指定。

Data Zone 块是用于存储数据的块。在 MINIX-FS 中，文件系统被分作 BLOCK\_SIZE 大小的数据块，称为 Zone。Zone 分作两种，一种是 Data Zone， 用于存储数据，即存储文件和目录。另一种是 Reserved Zone，这些 Zone 用于存 储 MINIX-FS 的基础块，也就是 Boot，Superblock，Inode-BitMap，Zone-BitMap 和 Inode-Table 块。本质上两种快没有任何区别，只是存储的数据不同而已。

下面我们介绍一下在操作系统史上著名的辩论-塔能鲍姆-托瓦兹辩论，这场辩论的主题是题在于操作系统内核架构的选择，两个主角分别是Minix作者安德鲁·斯图尔特·塔能鲍姆教授与Linux和git的作者林纳斯·托瓦兹。塔能鲍姆在题为《Linux 是过时的》（Linux is obsolete）的帖子中提到，在1991年仍然以宏内核来设计操作系统，是“回到1970年代的巨大退步“（a giant step back into the 1970s），现代的操作系统，应该像GNU Hurd一样采用微核心架构。托瓦兹一天之后反击，他强调，操作系统核心主要的功能都倚靠硬件特性，所以内核本身不需要过度具备可移植性，让高级的软件应用程序接口具备可移植性才是更重要的。Linux内核采用集成式核心架构，是因为它能够简化核心设计，这是一个权衡下的结果（An acceptable trade-off）。以Linux来跟Minix比较，移植程序到Linux上是更容易的。托瓦兹进一步说，可移植性是那些写不出新程序的人才需要的（Portability is for people who cannot write new programs）。这场辩论影响了Linux核心与Minix的设计走向，展示了软件工程师之间对于操作系统架构的不同思考角度，因此得到许多重视。在辩论结束后，Linux社区并没有因为这场辩论而改变作法，以微核心架构来重新设计Linux核心，但是在稍后改善了可移植性，扩展到x86之外的平台，同时引进了许多微核心架构的设计想法来改进其代码，例如采用了可加载核心模块。塔能鲍姆教授也仍然坚持以微核心架构来设计Minix，在1997年与2004年分别发表了两个新版本，完全不收费的发布Minix，并改善了Minix经常被批评的性能问题。在那么在30年后的今天，移动互联网和5G技术飞速发展的今天，同学们对操作系统内核架构的选择又是怎样呢？

1. 略

下面我们讲一下ext文件系统家族的第一个成员—ext文件系统，MINIX 文件系统最多能处理 14 个字符的文件名，并且只能处理 64MB 的存储空间。到了 1991 年，一般的硬盘尺寸已经达到了 40-140 MB。很显然，Linux 需要一个更好的文件系统。Linux 发布一年后， ext 文件系统首次实现并正式发布。Ext文件系统解决了 MINIX 文件系统中最糟糕的问题，可以处理高达 2 GB 存储空间并处理 255 个字符的文件名。除此之外，ext使用在 Linux 内核中的新虚拟文件系统（VFS）抽象层。但 ext 并没有长时间占统治地位，主要是由于它原始的时间戳（每个文件仅有一个时间戳，而不是今天我们所熟悉的有 inode、最近文件访问时间和最新文件修改时间的时间戳。）仅仅一年后，ext2 就替代了它。

1. 略

因为ext文件系统存在的局限性，ext2很快就诞生了，ext2在兼容ext的基础上实现了更加优越的性能，提供 GB 级别的最大文件大小和 TB 级别的文件系统大小，使其在 20 世纪 90 年代的地位牢牢巩固在文件系统大联盟中。很快就被广泛地使用，无论是在 Linux 内核中还是最终在 MINIX 中，且利用第三方模块可以使其应用于 MacOS 和 Windows。ext2文件系统借鉴了FFS文件系统，将整个分区划分为大小相等的块组（Block Group）

Ext2文件系统的每个块组都由以下几部分组成：

Super Block：超级块主要有两个功能：1）超级块结构给出了文件系统的全局信息。例如块大小，文件系统的版本等等。2)超级块结构包含一些函数指针,例如super\_operation的成员函数read\_inode提供了读取inode信息的功能。每个具体的文件系统一般都要提供这个函数来实现对inode信息的读取，例如ext2文件系统提供的具体函数是ext2\_read\_inode。

GDT(Group Descriptor Table)，组描述符表。由很多组描述符组成，整个分区分成多少个组就对应有多少个组描述符。组描述符（Group Descriptor）存储一个组的描述信息，例如在这个组中从哪里开始是inode表，从哪里开始是数据块，空闲的inode和数据块还有多少个等等。

从图中我们可以看出，每个块组中都有超级块和组描述符表，因为一旦超级块意外损坏就会丢失整个分区的数据，一旦块组描述符意外损坏就会丢失整个块组的数据，因此它们都有多份拷贝。只有块组0中包含的超级块和组描述符才被内核使用，当执行e2fsck检查文件系统一致性时，第0个块组中的超级块和块组描述符表就会拷贝到其它块组，这样当第0个块组的开头意外损坏时就可以用其它拷贝来恢复，从而减少损失。

Block Bitmap，块位图。块位图描述整个块组中哪些块已用哪些块空闲的，它本身占一个块，其中的每个bit代表本块组中的一个块，这个bit为1表示该块已用，这个bit为0表示该块空闲可用。

inode Bitmap，inode位图，和块位图类似，本身占一个块，其中每个bit表示一个inode是否空闲可用。

inode Table，inode表。一个文件除了数据需要存储之外，一些描述信息也需要存储，例如文件类型（常规、目录、符号链接等），权限，文件大小，创建/修改/访问时间等

Data Block：数据块。

接下来我们讲一下ext2的文件目录项数据结构，我们知道linux的设计思想是一切皆文件，目录也是一个文件，他的数据块中保存的是该目录下所有文件和子目录的文件目录项。linux下文件查找的过程便是读出目录的数据块，在其中查找感兴趣文件的文件目录项，进而获得感兴趣文件的inode。由此可知文件目录项主要是存储文件名至文件inode的映射关系，这样，根据文件名在父目录数据块中查找感兴趣文件就能获取该文件的inode号，进而可以得到该文件的所有信息。

我们看ppt中的代码，inode代表该文件inode编号，rec\_len表示本文件目录项的大小，为什么需要这个rec\_len呢，结构体定义好了整个长度不也就确认了嘛？这是因为该结构体的最后一个成员name并不是固定长度的，其最大可以支持256字节，因此必须要有一个长度域来保存当前目录项长度，name\_len指的是文件名长度，既然已经有了rec\_len，为什么还需要文件名长呢，岂不多此一举？这是考虑到存在文件名填充的问题。从效率上来考虑，每个struct ext2\_dir\_entry\_2最终都会被填充成4字节整数倍，对于目录项不是4字节整数倍的，需要在最后name文件名后面填充若干个0，因此name\_len中记录的便是name[]域中有效文件名长度（即不包含0）。我们看右图中的示例就明白了。

1.“.”和“..”文件名后都填充了‘\0’以使文件目录项总长度为4的整数倍；

2. music和src文件/目录均也填充了‘\0’以使文件目录项总长度为4的整数倍;

3. test.txt因为其文件目录项已经是16个字节，无需填充。

Ext2是一个非常成功的文件系统，时至今日，ext2仍然作为便携式 USB 驱动器的文件系统格式。但是ext2也有自己的问题，与 20 世纪 90 年代的大多数文件系统一样，如果在将数据写入到磁盘的时候，系统发生崩溃或断电，则容易发生灾难性的数据损坏。随着时间的推移，由于碎片（单个文件存储在多个位置，物理上其分散在旋转的磁盘上），它们也遭受了严重的性能损失。这种现象叫做崩溃一致性问题。翻到崩溃一致性部分。

1. 略

为了解决ext2的崩溃一致性问题，2001 年 11 月在 2.4.15 内核版本中ext3被采用到 Linux 内核主线中。ext3采用日志解决崩溃一致性问题，引入日志后使得ext3实现了在非正常关机后，系统也不需要检查文件系统（高可用性）和避免了意外宕机对文件系统的破坏（数据的完整性）

在使用 ext3 文件系统的 Linux 内核中实现了三个级别的日志记录方式： 日记(journal)、 顺序(ordered)和 回写(writeback)。

日记 是最低风险模式，在将数据和元数据提交给文件系统之前将其写入日志。这可以保证正在写入的文件与整个文件系统的一致性，但其显著降低了性能。

顺序 是大多数 Linux 发行版默认模式；顺序模式将元数据写入日志而直接将数据提交到文件系统。顾名思义，这里的操作顺序是固定的：首先，元数据提交到日志；其次，数据写入文件系统，然后才将日志中关联的元数据更新到文件系统。这确保了在发生崩溃时，那些与未完整写入相关联的元数据仍在日志中，且文件系统可以在回滚日志时清理那些不完整的写入事务。在顺序模式下，系统崩溃可能导致在崩溃期间文件的错误被主动写入，但文件系统它本身 —— 以及未被主动写入的文件 —— 确保是安全的。

回写 是第三种模式 —— 也是最不安全的日志模式。在回写模式下，像顺序模式一样，元数据会被记录到日志，但数据不会。与顺序模式不同，元数据和数据都可以以任何有利于获得最佳性能的顺序写入。这可以显著提高性能，但安全性低很多。尽管回写模式仍然保证文件系统本身的安全性，但在崩溃或崩溃之前写入的文件很容易丢失或损坏。



这是第二部分，崩溃一致性问题部分



为了更好的理解崩溃一致性，我们先举个例子，调用append()将单个数据块附加到原有文件。他的具体执行过程是打开文件，调用lseek()将文件偏移量移动到文件末尾，然后，向文件发出写入操作来完成追加，关闭文件。我们假定磁盘上使用标准的简单文件系统结构。包括一个inode位图（inode bitmap，只有8位，每个inode一个），一个数据位图（data bitmap，也是8位，每个数据块一个），inode（总共8个，编号为0到7，分布在4个块上），以及数据块（总共8个，编号为0～7）。我们看一下图中的结构，可以看到这个文件分配了一个inode（inode号为2），它在inode位图中标记，单个分配的数据块（数据块4）也在数据中标记位图。inode表示为I [v1]，因为它是此inode的第一个版本。向文件追加内容时，要向它添加一个新数据块，因此必须更新3个磁盘上的结构：inode（必须指向新块，并且由于追加而具有更大的大小），新数据块Db和新版本的数据位图（称之为B[v2]）表示新数据块已被分配。三者缺一不可。



我们再来看一下完成文件追加操作后的磁盘映像，实现v1到v2的转变，文件系统必须对磁盘执行3次单独写入，分别针对inode（I[v2]），位图（B[v2]）和数据块（Db）。当用户发出write()系统调用时，这些写操作通常不会立即发生。脏的inode、位图和新数据先在内存（页面缓存，page cache，或缓冲区缓存，buffer cache）中存在一段时间。然后，当文件系统最终决定将它们写入磁盘时（比如说5s或30s），文件系统将向磁盘发出必要的写入请求。遗憾的是，可能会发生崩溃，从而干扰磁盘的这些更新。

接下来让我们看一些崩溃情景示例，为了便于理解，我们假定完成两次写入操作后发生崩溃，导致一次写入操作未完成。首先是写入数据时系统崩溃， inode（I[v2]）和位图（B[v2]）写入了磁盘，但没有写入数据（D2），这种情况下文件系统元数据是完全一致的：inode有一个指向块5的指针，位图指示5正在使用，因此从文件系统的元数据的角度来看，一切看起来都很正常。但是有一个问题：5中又是垃圾。追加操作并没有完成，但是用户并不知道。

第二种情况，写入位图时系统崩溃。写入了inode（I[v2]）和数据块（Db），但没有写入位图（B[v2]）。在这种情况下，inode指向了磁盘上的正确数据，但同样在inode和位图（B1）的旧版本之间存在不一致。因此，我们在使用文件系统之前，又需要解决问题。



最后一种是写入i节点时系统崩溃。写入了位图（B[v2]）和数据块（Db），但没有写入inode（I[v2]）。在这种情况下，inode和数据位图之间再次存在不一致。但是，即使写入块并且位图指示其使用，我们也不知道它属于哪个文件，因为没有inode指向该块。



从这些崩溃场景中，大家可以看到由于崩溃而导致磁盘文件系统映像可能出现的许多问题：在文件系统数据结构中可能存在不一致性。可能有空间泄露，可能将垃圾数据返回给用户，等等。理想的做法是将文件系统从一个一致状态（在文件被追加之前），原子地（atomically）移动到另一个状态（在inode、位图和新数据块被写入磁盘之后）。遗憾的是，做到这一点不容易，因为磁盘一次只提交一次写入，而这些更新之间可能会发生崩溃或断电。我们将这个一般问题称为崩溃一致性问题（crash-consistency problem，也可以称为一致性更新问题，consistent-update problem）。那么怎么解决这些问题呢？



早期的文件系统采用了一种简单的方法来处理崩溃一致性。基本上，它们决定让不一致的事情发生，然后再修复它们（重启时）。这种偷懒方法的典型例子可以在一个工具中找到：fsck 。fsck是一个UNIX工具，用于查找这些不一致并修复它们。在不同的系统上，存在检查和修复磁盘分区的类似工具。这种方法无法解决所有问题。第一种情况也就是写入数据时系统崩溃就无法解决。此外，fsck（和类似的方法）有一个更大的、也许更根本的问题：太慢了。对于非常大的磁盘卷，扫描整个磁盘，以查找所有已分配的块并读取整个目录树，可能需要几分钟或几小时。随着磁盘容量的增长和RAID的普及，fsck的性能变得令人望而却步。



对于崩溃一致性问题，最流行的解决方案可能是从数据库管理系统的世界中借鉴的一个想法。这种名为预写日志（write-ahead logging）的想法，是为了解决这类问题而发明的。在文件系统中，出于历史原因，我们通常将预写日志称为日志（journaling）。更新磁盘时，在覆写结构之前，首先写下一点小注记（在磁盘上的其他地方，在一个众所周知的位置），描述你将要做的事情。写下这个注记就是“预写”部分，我们把它写入一个结构，并组织成“日志”。因此，就有了预写日志。通过将注释写入磁盘，可以保证在更新（覆写）正在更新的结构期间发生崩溃时，能够返回并查看你所做的注记，然后重试。因此，你会在崩溃后准确知道要修复的内容（以及如何修复它），而不必扫描整个磁盘。因此，通过设计，日志功能在更新期间增加了一些工作量，从而大大减少了恢复期间所需的工作量。



我们给一个例子来帮助大家理解数据日志（data journaling）的工作原理。假设再次进行标准的更新，我们再次希望将inode（I[v2]）、位图（B[v2]）和数据块（Db）写入磁盘。在将它们写入最终磁盘位置之前，现在先将它们写入日志。可以看到，这里写了5个块。事务开始（TxB）告诉我们有关此更新的信息，包括对文件系统即将进行的更新的相关信息（例如，块I[v2]、B[v2]和Db的最终地址），以及某种事务标识符（transaction identifier，TID）。中间的3个块只包含块本身的确切内容，这被称为物理日志（physical logging），因为我们将更新的确切物理内容放在日志中。最后一个块（TxE）是该事务结束的标记，也会包含TID。这个过程叫做日志写入。一旦这个事务安全地存在于磁盘上，我们就可以将待处理的元数据和数据更新写入文件系统中的最终位置。这个过程称为加检查点（checkpointing）。如果这些写入成功完成，我们已成功地为文件系统加上了检查点，基本上完成了。



但是现在又出现一种情况，写入日志期间发生崩溃时怎么办。我们希望一次发出所有 5 个块写入，因为这会将 5 个写入转换为单个顺序写入，因此更快。然而，由于以下原因，这是不安全的：给定如此大的写入，磁盘内部可以执行调度并以任何顺序完成大批写入的小块。因此，磁盘内部可以（1）写入TxB、I[v2]、B[v2]和TxE，然后才写入Db。遗憾的是，如果磁盘在（1）和（2）之间断电，ppt上的情况就会发生。



为避免该问题，文件系统分两步发出事务写入。首先，它将除TxE块之外的所有块写入日志，同时发出这些写入。当这些写入完成时，文件系统会发出TxE块的写入，从而使日志处于最终的安全状态，在这里，我们使用了磁盘提供的原子性保证。磁盘保证任何512字节写入都会发生或不发生（永远不会半写）。因此，为了确保TxE的写入是原子的，应该使它成为一个512字节的块。我们把将事务的内容（包括TxB、元数据和数据）写入日志，等待这些写入完成叫做新的日志写入，将事务提交块（包括TxE）写入日志，等待写完成，事务被认为已提交（committed）叫做日志提交。



当日志写入和日志提交都完成时，我们就可以将待处理的元数据和数据更新写入文件系统中的最终位置，也就是加检查点。



我们接着讲一下文件系统如何利用日志内容从崩溃中恢复，在更新序列期间，任何时候都可能发生崩溃。如果崩溃发生在事务被安全地写入日志之前（在步骤2完成之前），那么我们的工作很简单：简单地跳过待执行的更新。如果在事务已提交到日志之后但在加检查点完成之前发生崩溃，则文件系统可以按如下方式恢复（recover）更新。系统引导时，文件系统恢复过程将扫描日志，并查找已提交到磁盘的事务。然后，这些事务被重放（replayed，按顺序），文件系统再次尝试将事务中的块写入它们最终的磁盘位置。这种形式的日志是最简单的形式之一，称为重做日志（redo logging）。通过在日志中恢复已提交的事务，文件系统确保磁盘上的结构是一致的，因此可以继续工作，挂载文件系统并为新请求做好准备。



我们已经讲了两种方法，但是并不仅仅有这两种方法，我们简要介绍一下两种其他方法，首先是软更新方法，这种方法对文件系统的所有写入排序，确保磁盘上的结构永远不会处于不一致状态，举个例子，通过先写入数据块，再写入inode，可以确保inode永远不会指向垃圾。第二种是写时复制方法，自种方法相对更加流行，sun的zfs使用的就是这种方法，这种方法对磁盘上尚未使用的位置进行更新，更新完成后，文件系统翻转根结构，以包含指向更新结构的指针，从而确保文件系统一致性。



下面我们讲一下ext4的磁盘布局



和EXT2、EXT3一样，在ext4中，块是基本的分配单元把磁盘分成相同大小的block块后，ext4把这些block块合成块组。对于整个文件系统，需要预留一块区域作为引导区用于操作系统的启动，第一个块组的前面要留 1K，用于启动引导区。关于把块组划分为超级块、块组描述符表等我们已经讲过，不再赘述。

超级块和块组描述符表都是全局信息，而且这些数据很重要。如果这些数据丢失了，整个文件系统都打不开了，这比一个文件的一个块损坏更严重。所以，这两部分我们都需要备份，但是采取不同的策略。

默认策略在每个块中均保存一份超级块和块组描述表的备份

sparse\_super策略采取稀疏存储的方式，仅在块组索引为 0、3、5、7 的整数幂里存储。。

为了更进一步优化空间，ext4引入了Meta Block Groups策略,块组描述符表不会保存所有块组的描述符了，而是将块组分成多个组，我们称为元块组（Meta Block Group）。每个元块组里面的块组描述符表仅仅包括自己的，一个元块组包含64个块组，这样一个元块组中的块组描述符表最多64项。

我们假设一共有256个块组，原来是一个整的块组描述符表，里面有256项，现在分成4个元块组，每个元块组里面的块组描述符表就只有64项了，这就小多了，而且四个元块组自己备份自己的。

根据图中，每一个元块组包含64个块组，块组描述符表也是64项，备份三份，在元块组的第一个，第二个和最后一个块组的开始处。

下面我们讲一下ext4的inode。

1. 从这个数据结构中我们可以看出，inode 里面有文件的读写权限 i\_mode，属于哪个用户 i\_uid，哪个组 i\_gid，大小是多少 i\_size\_lo，占用多少个块 i\_blocks\_lo。另外，这里面还有几个与文件相关的时间。i\_atime 即 access time，是最近一次访问文件的时间；i\_ctime 即 change time，是最近一次更改 inode 的时间；i\_mtime 即 modify time，是最近一次更改文件的时间。这里我们需要重点关注以下i\_block。

在ext2和ext3中，其中前12项直接保存了块的位置，也就是说，我们可以通过i\_block[0-11]，直接得到保存文件内容的块。但是，如果一个文件比较大，12块放不下。当我们用到i\_block[12]的时候，就不能直接放数据块的位置了，要不然i\_block很快就会用完了。那么可以让i\_block[12]指向一个块，这个块里面不放数据块，而是放数据块的位置，这个块我们称为间接块。如果文件再大一些，i\_block[13]会指向一个块，我们可以用二次间接块。二次间接块里面存放了间接块的位置，间接块里面存放了数据块的位置，数据块里面存放的是真正的数据。如果文件再大点，那么i\_block[14]同理。这里面有一个非常显著的问题，对于大文件来讲，我们要多次读取硬盘才能找到相应的块，这样访问速度就会比较慢。

为了解决这个问题，ext4做了一定的改变。它引入了一个新的概念，叫作Extents。比方说，一个文件大小为128M，如果使用4k大小的块进行存储，需要32k个块。如果按照ext2或者ext3那样散着放，数量太大了。但是Extents可以用于存放连续的块，也就是说，我们可以把128M放在一个Extents里面。这样的话，对大文件的读写性能提高了，文件碎片也减少了。每个节点都有一个头，ext4\_extent\_header可以用来描述某个节点。

大家看ext4\_extent\_header这个数据结构， eh\_entries表示这个节点里面有多少项。这里的项分两种，如果是叶子节点，这一项会直接指向硬盘上的连续块的地址，我们称为数据节点ext4\_extent；如果是分支节点，这一项会指向下一层的分支节点或者叶子节点，我们称为索引节点ext4\_extent\_idx。这两种类型的项的大小都是12个byte。

如果文件不大，inode里面的i\_block中，可以放得下一个ext4\_extent\_header和4项ext4\_extent。所以这个时候，eh\_depth为0，也即inode里面的就是叶子节点，树高度为0。

如果文件比较大，4个extent放不下，就要分裂成为一棵树，eh\_depth>0的节点就是索引节点，其中根节点深度最大，在inode中。最底层eh\_depth=0的是叶子节点。

除了根节点，其他的节点都保存在一个块4k里面，4k扣除ext4\_extent\_header的12个byte，剩下的能够放340项，每个extent最大能表示128MB的数据，340个extent会使你的表示的文件达到42.5GB。。

Ext4预留了一些inode做特殊特性使用，这些inode往往留给特定用途：

Ext4系统从设计上就认为数据局部性是文件系统的文件系统的一个理想品质。 在机械硬盘上，相关联的数据存放在相近的blocks上，可以使得在访问数据时减少磁头驱动器的移动距离，从而加快IO访问。 在SSD上，没有像磁头一样的移动组件。但是数据局部性可以增大请求大小从而降低请求数量。同时可以尽可能的集中擦写块，从而提高重写速度。对减少碎片也很有帮助。 Ext4解决碎片问题的第一个方法是，multi-block allocator。当创建文件时，块分配器会分配8K的磁盘空间给文件，当然，这样做有个假定——8K空间很快被写数据。当文件被关闭后，没有使用的空间会被释放掉。但是，如果这种推测(就是刚才的假定)是正确的，那么这块数据就被写入了这个简单的extent（8K空间）中。 Ext4解决碎片问题的第二个方法是，delayed allocation。根据这个策略，当文件需要更多的块进行写操作时，文件系统会推迟确定在磁盘上的写入位置，直到脏数据要写入磁盘。除非必要，否则不写入磁盘(Timeout或sync或内核out of memory)。这样可以更好的是文件进行位置决策。 Ext4解决碎片问题的第三个方法是，尽可能使得一个文件的数据和其inode在相同的块组中。这种方法减少了data寻址耗时。特别是刚刚读取inode后就访问数据这种场景。 Ext4解决碎片问题的第四个方法是，整个卷被分割为大小为128M的groups。这样可以最大限度的保证数据局部性。然而，当在根目录创建文件时，inode分配器会扫描块组，从inode最空闲的group中分配inode空间。这种方式会扫描整个卷。但是一般而言，都是按照顺序往下创建的。

下面我们讲一下ext4的目录。

为了便于文件的查找，我们必须要有索引，即文件目录。其实目录本身也是个文件，也有 inode。inode 里面也是指向一些块。和普通文件不同的是，普通文件的块里面保存的是文件数据，而目录文件的块里面保存的是目录里面一项一项的文件信息。这些信息我们称为 ext4\_dir\_entry。这里有两个版本，第二个版本 ext4\_dir\_entry\_2 是将一个 16 位的 name\_len，变成了一个 8 位的 name\_len 和 8 位的 file\_type



目录文件的块中，最简单的保存格式是列表，就是一项一项地将 ext4\_dir\_entry\_2 列在哪里。每一项都会保存这个目录的下一级的文件的文件名和对应的 inode，通过这个 inode，就能找到真正的文件。第一项是“.”，表示当前目录，第二项是“…”，表示上一级目录，接下来就是一项一项的文件名和 inode。如果一个目录下面的文件太多的时候，我们想在这个目录下找一个文件，按照列表一个个去找太慢了，于是我们就添加了索引的模式。如果在 inode 中设置 EXT4\_INDEX\_FL 标志，则目录文件的块的组织形式将发生变化。当前目录和上级目录不变，文件列表改用dx\_root\_info结构体，其中最重要的成员变量是 indirect\_levels，表示间接索引的层数。索引项由结构体 dx\_entry表示，本质上是文件名的哈希值和数据块的一个映射关系。



如果我们要查找一个目录下面的文件名，可以通过名称取哈希。如果哈希能够匹配上，就说明这个文件的信息在相应的块里面。然后打开这个块，如果里面不再是索引，而是索引树的叶子节点的话，那里面还是 ext4\_dir\_entry\_2 的列表，我们只要一项一项找文件名就行。通过索引树，我们可以将一个目录下面的 N 多的文件分散到很多的块里面，可以很快地进行查找。

下面我们讲一下ext4的链接文件。

软链接和硬链接也是文件的一种，可以通过如下命令创建。ln -s 创建的是软链接，不带 -s 创建的是硬链接。



硬链接与原始文件共用一个 inode ，但是 inode 是不跨文件系统的，每个文件系统都有自己的 inode 列表，因而硬链接是没有办法跨文件系统的。而软链接不同，软链接相当于重新创建了一个文件。这个文件也有独立的 inode，只不过打开这个文件看里面内容的时候，内容指向另外的一个文件。这就很灵活了。我们可以跨文件系统，甚至目标文件被删除了链接文件也依然存在，只不过指向的文件找不到了而已。



下面我们讲一下其他新兴的文件系统，zfs和btrfs



ZFS 由 Sun Microsystems 开发，作为Sun MicroSystem公司的 OpenSolaris 的下一代文件系统而设计的，以 zettabyte 命名 —— 相当于 1 万亿 GB —— 因为它理论上可以解决大型存储系统。



与大多数文件系统不同，ZFS 结合了文件系统和卷管理器的特性。这意味着，它与其他文件系统不同，ZFS 可以创建跨越一系列硬盘或池的文件系统。不仅如此，你还可以通过添加硬盘来增大池的存储容量。ZFS 可以进行分区和格式化。



在大多数文件系统上，当数据被重写时，它将永久丢失。而在 ZFS 中，新数据会写到不同的块。写完成之后，更新文件系统元数据信息，使之指向新的数据块（LCTT 译注：更新之后，原数据块成为磁盘上的垃圾，需要有对应的垃圾回收机制）。这确保了如果在写新数据的时候系统崩溃（或者发生其它事，比如突然断电），那么原数据将会保存下来。这也意味着，在系统发生崩溃之后，不需要运行 fsck 来检查和修复文件系统。写时拷贝使得 ZFS 有了另一个特性： 快照(snapshots)。ZFS 使用快照来跟踪文件系统中的更改。快照包含文件系统的原始版本（文件系统的一个只读版本），实时文件系统则包含了自从快照创建之后的任何更改。没有使用额外的空间。因为新数据将会写到实时文件系统新分配的块上。如果一个文件被删除了，那么它在快照中的索引也会被删除。所以，快照主要是用来跟踪文件的更改，而不是文件的增加和创建。快照可以挂载成只读的，以用来恢复一个文件的过去版本。实时文件系统也可以回滚到之前的快照。回滚之后，自从快照创建之后的所有更改将会丢失。当向 ZFS 写入新数据时，会创建该数据的校验和。在读取数据的时候，使用校验和进行验证。如果前后校验和不匹配，那么就说明检测到了错误，然后，ZFS 会尝试自动修正错误。ZFS 不需要任何额外软件或硬件就可以处理 RAID（磁盘阵列）。毫不奇怪，因为 ZFS 有自己的 RAID 实现：RAID-Z 。RAID-Z 是 RAID-5 的一个变种，不过它克服了 RAID-5 的写漏洞：意外重启之后，数据和校验信息会变得不同步（LCTT 译注：RAID-5 的条带在正写入数据时，如果这时候电源中断，那么奇偶校验数据将跟该部分数据不同步，因此前边的写无效；RAID-Z 用了 “可变宽的 RAID 条带” 技术，因此所有的写都是全条带写入）。为了使用基本级别的 RAID-Z（RAID-Z1），你需要至少三块磁盘，其中两块用来存储数据，另外一块用来存储奇偶校验信息。而 RAID-Z2 需要至少两块磁盘存储数据以及两块磁盘存储校验信息。RAID-Z3 需要至少两块磁盘存储数据以及三块磁盘存储校验信息。另外，只能向 RAID-Z 池中加入偶数倍的磁盘，而不能是奇数倍的。创建 ZFS 的时候，它是作为最后一个文件系统而设计的 。那时候，大多数文件系统都是 64 位的，ZFS 的创建者决定直接跳到 128 位，等到将来再来证明这是对的。这意味着 ZFS 的容量大小是 32 位或 64 位文件系统的 1600 亿亿倍。



Btrfs 是 B-Tree Filesystem 的简称，通常发音为 “butter” —— 由 Chris Mason 于 2007 年在 Oracle 任职期间发布。Btrfs的特性可以分为三个部分，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这一点。

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。



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。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。。



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。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。



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。



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，从而提高 IO 性能。



理解 COW 事务，必须首先理解 COW 和事务这两个术语。什么是 COW?所谓 COW，即每次写磁盘数据时，先将更新数据写入一个新的 block，当新数据写入成功之后，再更新相关的数据结构指向新 block 。COW 只能保证单一数据更新的原子性。但文件系统中很多操作需要更新多个不同的元数据，比如创建文件需要修改以下这些元数据：

修改 extent tree，分配一段磁盘空间

创建一个新的 inode，并插入 FS Tree 中

增加一个目录项，插入到 FS Tree 中

任何一个步骤出错，文件便不能创建成功，因此可以定义为一个事务。

假设A 是 FS Tree 的根节点，新的 inode 的信息将被插入节点 C 。首先，btrfs 将 inode 插入一个新分配的 block C ’中，并修改上层节点 B，使其指向新的 block C ’；修改 B 也将引发 COW，以此类推，引发一个连锁反应，直到最顶层的 Root A 。当整个过程结束后，新节点 A ’变成了 FS Tree 的根。但此时事务并未结束，superblock 依然指向 A 。接下来，修改目录项（E 节点），同样引发这一过程，从而生成新的根节点 A ’’ 此时，inode 和目录项都已经写入磁盘，可以认为事务已经结束。 btrfs 修改 superblock，使其指向 A ’’，COW 事务能够保证文件系统的一致性，并且系统 Reboot 之后不需要执行 fsck 。因为 superblock 要么指向新的 A ’’，要么指向 A，无论哪个都是一致的数据。



ext2/3 没有校验和，对磁盘完全信任。而不幸的是，磁盘的错误始终存在，不仅发生在廉价的 IDE 硬盘上，昂贵的 RAID 也存在 silent corruption 问题。而且随着存储网络的发展，即使数据从磁盘读出正确，也很难确保能够安全地穿越网络设备。btrfs 在读取数据的同时会读取其相应的 checksum 。如果最终从磁盘读取出来的数据和 checksum 不相同，btrfs 会首先尝试读取数据的镜像备份，如果数据没有镜像备份，btrfs 将返回错误。写入磁盘数据之前，btrfs 计算数据的 checksum 。然后将 checksum 和数据同时写入磁盘。Btrfs 采用单独的 checksum Tree 来管理数据块的校验和，把 checksum 和 checksum 所保护的数据块分离开，从而提供了更严格的保护。假如在每个数据 block 的 header 中加入一个域保存 checksum，那么这个数据 block 就成为一个自己保护自己的结构。这种结构下有一种错误无法检测出来，比如本来文件系统打算从磁盘上读 block A，但返回了 block B，由于 checksum 在 block 内部，因此 checksum 依旧是正确的。 btrfs 采用 checksum tree 来保存数据块的 checksum，避免了上述问题。Btrfs 采用 crc32 算法计算 checksum，在将来的开发中会支持其他类型的校验算法。为了提高效率，btrfs 将写数据和 checksum 的工作分别用不同的内核线程并行执行。



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 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。



Btrfs 主页上罗列的其他特性不容易分类，这些特性都是现代文件系统中比较先进的技术，能够提高文件系统的时间或空间效率。

Delay allocation 延迟分配技术能够减少磁盘碎片。在 Linux 内核中，为了提高效率，很多操作都会延迟。 在文件系统中，小块空间频繁的分配和释放会造成碎片。延迟分配是这样一种技术，当用户需要磁盘空间时，先将数据保存在内存中。并将磁盘分配需求发送给磁盘空间分配器，磁盘空间分配器并不立即分配真正的磁盘空间。只是记录下这个请求便返回。 磁盘空间分配请求可能很频繁，所以在延迟分配的一段时间内，磁盘分配器可以收到很多的分配请求，一些请求也许可以合并，一些请求在这段延迟期间甚至可能被取消。通过这样的“等待”，往往能够减少不必要的分配，也有可能将多个小的分配请求合并为一个大的请求，从而提高 IO 效率。

很多应用程序有预先分配磁盘空间的需要。他们可以通过 posix\_fallocate 接口告诉文件系统在磁盘上预留一部分空间，但暂时并不写入数据。如果底层文件系统不支持 fallocate，那么应用程序只有使用 write 预先写一些无用信息以便为自己预留足够的磁盘空间。

由文件系统来支持预留空间更加有效，而且能够减少磁盘碎片，因为所有的空间都是一次分配，因而更有可能使用连续的空间。 Btrfs 支持 posix\_fallocate 。

Inline file 系统中往往存在大量的小文件，比如几百个字节或者更小。如果为其分配单独的数据 block，便会引起内部碎片，浪费磁盘空间。 btrfs 将小文件的内容保存在元数据中，不再额外分配存放文件数据的磁盘块。改善了内部碎片问题，也增加了文件的访问效率。