你可以看到会发生什么：E 分散在磁盘上，因此，在访问 E 时，无法从磁盘获得峰值（顺序）性能。你首先读取 E1 和 E2，然后寻道，再读取 E3 和 E4。这个碎片问题一直发生在老 UNIX 文件系统中，并且会影响性能。（插一句：这个问题正是磁盘碎片整理工具要解决的。它们将重新组织磁盘数据以连续放置文件，并为让空闲空间成为一个或几个连续的区域，移动数据，然后重写 inode 等以反映变化。）

另一个问题：原始块大小太小（512 字节）。因此，从磁盘传输数据本质上是低效的。较小的块是好的，因为它们最大限度地减少了内部碎片（internal fragmentation，块内的浪费），但是由于每个块可能需要一个定位开销来访问它，因此传输不佳。我们可以总结如下问题。

关键问题：如何组织磁盘数据以提高性能

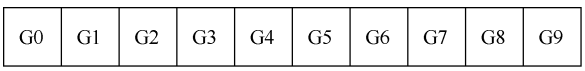
如何组织文件系统数据结构以提高性能？在这些数据结构之上，需要哪些类型的分配策略？如何让文件系统具有“磁盘意识”？

41.2 FFS：磁盘意识是解决方案

伯克利的一个小组决定建立一个更好、更快的文件系统，他们聪明地称之为快速文件系统（Fast File System，FFS）。思路是让文件系统的结构和分配策略具有“磁盘意识”，从而提高性能，这正是他们所做的。因此，FFS 进入了文件系统研究的新时代。通过保持与文件系统相同的接口（相同的 API，包括 open()、read()、write()、close()和其他文件系统调用），但改变内部实现，作者为新文件系统的构建铺平了道路，这方面的工作今天仍在继续。事实上，所有现代文件系统都遵循现有的接口（从而保持与应用程序的兼容性），同时为了性能、可靠性或其他原因，改变其内部实现。

41.3 组织结构：柱面组

第一步是更改磁盘上的结构。FFS 将磁盘划分为一些分组，称为柱面组（cylinder group，而一些现代文件系统，如 Linux ext2 和 ext3，就称它们为块组，即 block group）。因此，我们可以想象一个具有 10 个柱面组的磁盘：



这些分组是 FFS 用于改善性能的核心机制。通过在同一组中放置两个文件，FFS 可以确保先后访问两个文件不会导致穿越磁盘的长时间寻道。

因此，FFS 需要能够在每个组中分配文件和目录。每个组看起来像这样：

我们现在描述一个柱面组的构成。出于可靠性原因，每个组中都有超级块（super block）的一个副本（例如，如果一个被损坏或划伤，你仍然可以通过使用其中一个副本来挂载和访问文件系统）。

在每个组中，我们需要记录该组的 inode 和数据块是否已分配。每组的 inode 位图（inode bitmap，ib）和数据位图（data bitmap，db）起到了这个作用，分别针对每组中的 inode 和数据块。位图是管理文件系统中可用空间的绝佳方法，因为很容易找到大块可用空间并将其分配给文件，这可能会避免旧文件系统中空闲列表的某些碎片问题。

最后，inode 和数据块区域就像之前的极简文件系统一样。像往常一样，每个柱面组的大部分都包含数据块。

补充：FFS 文件创建

例如，考虑在创建文件时必须更新哪些数据结构。对于这个例子，假设用户创建了一个新文件/foo/bar.txt，并且该文件长度为一个块（4KB）。该文件是新的，因此需要一个新的 inode。因此，inode 位图和新分配的 inode 都将写入磁盘。该文件中还包含数据，因此也必须分配。因此（最终）将数据位图和数据块写入磁盘。因此，会对当前柱面组进行至少 4 次写入（回想一下，在写入发生之前，这些写入可以在存储器中缓冲一段时间）。但这并不是全部！特别是，在创建新文件时，我们还必须将文件放在文件系统层次结构中。因此，必须更新目录。具体来说，必须更新父目录 foo，以添加 bar.txt 的条目。此更新可能放入 foo 现有的数据块，或者需要分配新块（包括关联的数据位图）。还必须更新 foo 的 inode，以反映目录的新长度以及更新时间字段（例如最后修改时间）。总的来说，创建一个新文件需要做很多工作！也许下次你这样做，你应该更加心怀感激，或者至少感到惊讶，一切都运作良好。

41.4 策略：如何分配文件和目录

有了这个分组结构，FFS 现在必须决定，如何在磁盘上放置文件和目录以及相关的元数据，以提高性能。基本的咒语很简单：相关的东西放一起（以此推论，无关的东西分开放）。

因此，为了遵守规则，FFS 必须决定什么是“相关的”，并将它们置于同一个区块组内。相反，不相关的东西应该放在不同的块组中。为实现这一目标，FFS 使用了一些简单的放置推断方法。

首先是目录的放置。FFS 采用了一种简单的方法：找到分配数量少的柱面组（因为我们希望跨组平衡目录）和大量的自由 inode（因为我们希望随后能够分配一堆文件），并将目录数据和 inode 放在该分组中。当然，这里可以使用其他推断方法（例如，考虑空闲数据块的数量）。

对于文件，FFS 做两件事。首先，它确保（在一般情况下）将文件的数据块分配到与其 inode 相同的组中，从而防止 inode 和数据之间的长时间寻道（如在老文件系统中）。其次，它将位于同一目录中的所有文件，放在它们所在目录的柱面组中。因此，如果用户创建了 4 个文件，/dir1/1.txt、/dir1/2.txt、/dir1/3.txt 和/dir99/4.txt，FFS 会尝试将前 3 个放在一起（同

一组），与第四个远离（它在另外某个组中）。

应该注意的是，这些推断方法并非基于对文件系统流量的广泛研究，或任何特别细致的研究。相反，它们建立在良好的老式常识基础之上（这不就是 CS 代表的吗？）。目录中的文件通常一起访问（想象编译一堆文件然后将它们链接到单个可执行文件中）。因为它们确保了相关文件之间的寻道时间很短，FFS 通常会提高性能。

41.5 测量文件的局部性

为了更好地理解这些推断方法是否有意义，我们决定分析文件系统访问的一些跟踪记录，看看是否确实存在命名空间的局部性。出于某种原因，文献中似乎没有对这个主题进行过很好的研究。

具体来说，我们进行了 SEER 跟踪[K94]，并分析了目录树中不同文件的访问有多“遥远”。例如，如果打开文件 f，然后跟踪到它重新打开（在打开任何其他文件之前），则在目录树中打开的这两个文件之间的距离为零（因为它们是同一文件）。如果打开目录 dir 中的文件 f（即 dir/f），然后在同一目录中打开文件 g（即 dir/g），则两个文件访问之间的距离为 1，因为它们共享相同的目录，但不是同一个文件。换句话说，我们的距离度量标准衡量为了找到两个文件的共同祖先，必须在目录树上走多远。它们在树上越靠近，度量值越低。

图 41.1 展示了 SEER 跟踪中看到的局部性，针对 SEER 集群中所有工作站上的所有 SEER 跟踪。其中的 *x* 轴是差异度量值，*y* 轴是具有该差异值的文件打开的累积百分比。具体来说，对于 SEER 跟踪（图中标记为“跟踪”），你可以看到大约 7%的文件访问是先前打开的文件，并且近 40%的文件访问是相同的文件或同一目录中的文件（即 0 或 1 的差异值）。因此，FFS 的局部性假设似乎有意义（至少对于这些跟踪）。

有趣的是，另外 25%左右的文件访问是距离为 2 的文件。当用户以多级方式构造一组相关目录，并不断在它们之间跳转时，就会发生这种类型的局部性。例如，如果用户有一个 src 目录，并将目标文件（.o 文件）构建到 obj 目录中，并且这两个目录都是主 proj 目录的子目录，则常见访问模式就是 proj/src/foo .c 后跟着 proj/obj/foo.o。这两个访问之间的距离是 2，因为 proj 是共同的祖先。FFS 在其策略中没有考虑这种类型的局部性，因此在这种访问之间会发生更多的寻道。

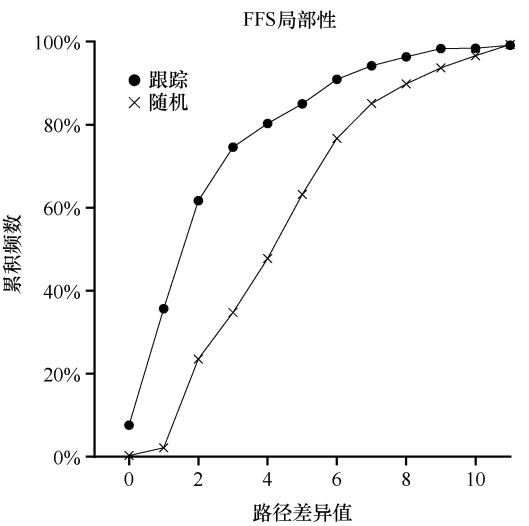


图 41.1 FFS 局部性的 SEER 跟踪

为了进行比较，我们还展示了“随机”跟踪的局部性。我们以随机的顺序，从原有的 SEER 跟踪中选择文件，计算这些随机顺序访问之间的距离度量值，从而生成随机跟踪。如你所见，随机跟踪中的命名空间局部性较少，和预期的一样。但是，因为最终每个文件共享一个共同的祖先（即根），总会有一些局部性，

因此随机跟踪可以作为比较点。

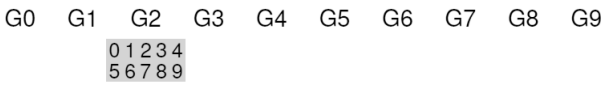
41.6 大文件例外

在 FFS 中，文件放置的一般策略有一个重要的例外，它出现在大文件中。如果没有不同的规则，大文件将填满它首先放入的块组（也可能填满其他组）。以这种方式填充块组是不符合需要的，因为它妨碍了随后的“相关”文件放置在该块组内，因此可能破坏文件访问的局部性。

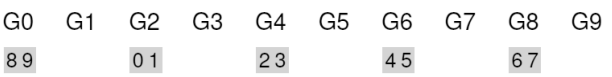
因此，对于大文件，FFS 执行以下操作。在将一定数量的块分配到第一个块组（例如，12 个块，或 inode 中可用的直接指针的数量）之后，FFS 将文件的下一个“大”块（即第一个间接块指向的那些部分）放在另一个块组中（可能因为它的利用率低而选择）。然后，文件的下一个块放在另一个不同的块组中，依此类推。

让我们看一些图片，更好地理解这个策略。如果没有大文件例外，单个大文件会将其所有块放入磁盘的一部分。我们使用一个包含 10 个块的文件的小例子，来直观地说明该行为。

以下是 FFS 没有大文件例外时的图景：



有了大文件例外，我们可能会看到像这样的情形，文件以大块的形式分布在磁盘上：



聪明的读者会注意到，在磁盘上分散文件块会损害性能，特别是在顺序文件访问的相对常见的情况下（例如，当用户或应用程序按顺序读取块 0～9 时）。你是对的！确实会。我们可以通过仔细选择大块大小，来改善这一点。

具体来说，如果大块大小足够大，我们大部分时间仍然花在从磁盘传输数据上，而在大块之间寻道的时间相对较少。每次开销做更多工作，从而减少开销，这个过程称为摊销（amortization），它是计算机系统中的常用技术。

举个例子：假设磁盘的平均定位时间（即寻道和旋转）是 10ms。进一步假设磁盘以 40 MB/s 的速率传输数据。如果我们的目标是花费一半的时间来寻找数据块，一半时间传输数据（从而达到峰值磁盘性能的 50%），那么需要每 10ms 定位开销导致 10ms 的传输数据。所以问题就变成了：为了在传输中花费 10ms，大块必须有多大？简单，只需使用数学，特别是我们在磁盘章节中提到的量纲分析：

$$\frac{40\cancel{MB}}{1\cancel{s}} \times \frac{1024KB}{1\cancel{MB}} \times \frac{1\cancel{s}}{1000\cancel{ms}} \times 10\cancel{ms} = 109.6KB \tag{41.1}$$

基本上，这个等式是说：如果你以 40 MB/s 的速度传输数据，每次寻找时只需要传输 409.6KB，就能花费一半的时间寻找，一半的时间传输。同样，你可以计算达到 90%峰值带宽所需的块大小（结果大约为 3.69MB），甚至达到 99%的峰值带宽（40.6MB！）。正如你所

看到的，越接近峰值，这些块就越大（图 41.2 展示了这些值）。

但是，FFS 没有使用这种类型的计算来跨组分布大文件。相反，它采用了一种简单的方法，基于 inode 本身的结构。前 12 个直接块与 inode 放在同一组中。每个后续的间接块，以及它指向的所有块都放在不同的组中。如果块大小为 4KB，磁盘地址是 32 位，则此策略意味着文件的每 1024 个块（4MB）放在单独的组中，唯一的例外是直接指针所指向的文件的前 48KB。

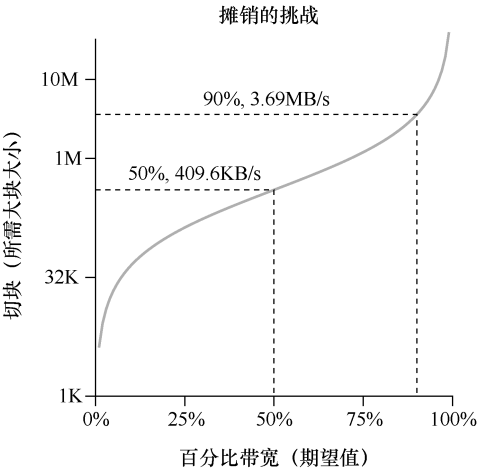


图 41.2 摊销：大块必须多大

我们应该注意到，磁盘驱动器的趋势是传输速率相当快，因为磁盘制造商擅长将更多位填塞到同一表面。但驱动器的机械方面与寻道相关（磁盘臂速度和旋转速度），改善相当缓慢[P98]。这意味着随着时间的推移，机械成本变得相对昂贵，因此，为了摊销所述成本，你必须在寻道之间传输更多数据。

41.7 关于 FFS 的其他几件事

FFS 也引入了一些其他创新。特别是，设计人员非常担心容纳小文件。事实证明，当时许多文件大小为 2KB 左右，使用 4KB 块虽然有利于传输数据，但空间效率却不太好。因此，在典型的文件系统上，这种内部碎片（internal fragmentation）可能导致大约一半的磁盘浪费。

FFS 设计人员采用很简单的解决方案解决了这个问题。他们决定引入子块（sub-block），这些子块有 512 字节，文件系统可以将它们分配给文件。因此，如果你创建了一个小文件（比如大小为 1KB），它将占用两个子块，因此不会浪费整个 4KB 块。随着文件的增长，文件系统将继续为其分配 512 字节的子块，直到它达到完整的 4KB 数据。此时，FFS 将找到一个 4KB 块，将子块复制到其中，并释放子块以备将来使用。

你可能会发现这个过程效率低下，文件系统需要大量的额外工作（特别是执行复制的大量额外 I/O）。你又对了！因此，FFS 通常通过修改 libc 库来避免这种异常行为。该库将缓冲写入，然后以 4KB 块的形式将它们发送到文件系统，从而在大多数情况下完全避免子块的特殊情况。

FFS 引入的第二个巧妙方法，是针对性能进行优化的磁盘布局。那时候（在 SCSI 和其他更现代的设备接口之前），磁盘不太复杂，需要主机 CPU 以更加亲力亲为的方式来控制它们的操作。当文件放在磁盘的连续扇区上时，FFS 遇到了问题，如图 41.3 左图所示。

具体来说，在顺序读取期间出现了问题。FFS 首先发出一个请求，读取块 0。当读取完成时，FFS 向块 1 发出读取，为时已晚：块 1 已在磁头下方旋转，现在对块 1 的读取将导致完全旋转。

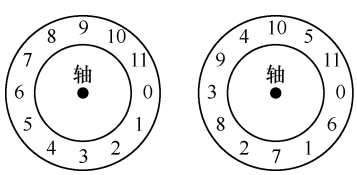


图 41.3 FFS：标准与参数化放置

FFS 使用不同的布局解决了这个问题，如图 41.3（右图）

所示。通过每次跳过一块（在这个例子中），在下一块经过磁头之前，FFS 有足够的时间发出请求。实际上，FFS 足够聪明，能够确定特定磁盘在布局时应跳过多少块，以避免额外的旋转。这种技术称为参数化，因为 FFS 会找出磁盘的特定性能参数，并利用它们来确定准确的交错布局方案。

你可能会想：这个方案毕竟不太好。实际上，使用这种类型的布局只能获得 50%的峰值带宽，因为你必须绕过每个轨道两次才能读取每个块一次。幸运的是，现代磁盘更加智能：它们在内部读取整个磁道并将其缓冲在内部磁盘缓存中（由于这个原因，通常称为磁道缓冲区，track buffer）。然后，在对轨道的后续读取中，磁盘就从其高速缓存中返回所需数据。因此，文件系统不再需要担心这些令人难以置信的低级细节。如果设计得当，抽象和更高级别的接口可能是一件好事。

FFS 还增加了另一些可用性改进。FFS 是允许长文件名的第一个文件系统之一，因此在文件系统中实现了更具表现力的名称，而不是传统的固定大小方法（例如，8 个字符）。此外，引入了一种称为符号链接的新概念。正如第 40 章所讨论的那样，硬链接的局限性在于它们都不能指向目录（因为害怕引入文件系统层次结构中的循环），并且它们只能指向同一卷内的文件（即 inode 号必须仍然有意义）。符号链接允许用户为系统上的任何其他文件或目录创建“别名”，因此更加灵活。FFS 还引入了一个原子 rename()操作，用于重命名文件。除了基本技术之外，可用性的改进也可能让 FFS 拥有更强大的用户群。

提示：让系统可用

FFS 最基本的经验可能在于，它不仅引入了磁盘意识布局（这是个好主意），还添加了许多功能，这些功能让系统的可用性更好。长文件名、符号链接和原子化的重命名操作都改善了系统的可用性。虽然很难写一篇研究论文（想象一下，试着读一篇 14 页的论文，名为《符号链接：硬链接长期失散的表兄》），但这些小功能让 FFS 更可用，从而可能增加了人们采用它的机会。让系统可用通常与深层技术创新一样重要，或者更重要。

41.8 小结

FFS 的引入是文件系统历史中的一个分水岭，因为它清楚地表明文件管理问题是操作系统中最有趣的问题之一，并展示了如何开始处理最重要的设备：硬盘。从那时起，人们开发了数百个新的文件系统，但是现在仍有许多文件系统从 FFS 中获得提示（例如，Linux ext2 和 ext3 是明显的知识传承）。当然，所有现代系统都要感谢 FFS 的主要经验：将磁盘当作磁盘。

参考资料

[MJLF84]“A Fast File System for UNIX”

Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry ACM Transactions on Computing

Systems.
August, 1984. Volume 2, Number 3.
pages 181-197.
McKusick 因其对文件系统的贡献而荣获 IEEE 的 Reynold B. Johnson 奖，其中大部分是基于他的 FFS 工作。在他的获奖演讲中，他讲到了最初的 FFS 软件：只有 1200 行代码！现代版本稍微复杂一些，例如，BSD FFS 后继版本现在大约有 5 万行代码。

[P98]“Hardware Technology Trends and Database Opportunities”David A. Patterson
Keynote Lecture at the ACM SIGMOD Conference (SIGMOD ’98) June, 1998
磁盘技术趋势及其随时间变化的简单概述。

[K94]“The Design of the SEER Predictive Caching System”
G. H. Kuenning
MOBICOMM ’94, Santa Cruz, California, December 1994
据 Kuenning 说，这是 SEER 项目的较全面的概述，这导致人们收集这些跟踪记录（和其他一些事）。

第 42 章　崩溃一致性：FSCK 和日志

至此我们看到，文件系统管理一组数据结构以实现预期的抽象：文件、目录，以及所有其他元数据，它们支持我们期望从文件系统获得的基本抽象。与大多数数据结构不同（例如，正在运行的程序在内存中的数据结构），文件系统数据结构必须持久（persist），即它们必须长期存在，存储在断电也能保留数据的设备上（例如硬盘或基于闪存的 SSD）。

文件系统面临的一个主要挑战在于，如何在出现断电（power loss）或系统崩溃（system crash）的情况下，更新持久数据结构。具体来说，如果在更新磁盘结构的过程中，有人绊到电源线并且机器断电，会发生什么？或者操作系统遇到错误并崩溃？由于断电和崩溃，更新持久性数据结构可能非常棘手，并导致了文件系统实现中一个有趣的新问题，称为崩溃一致性问题（crash-consistency problem）。

这个问题很容易理解。想象一下，为了完成特定操作，你必须更新两个磁盘上的结构 A 和 B。由于磁盘一次只为一个请求提供服务，因此其中一个请求将首先到达磁盘（A 或 B）。如果在一次写入完成后系统崩溃或断电，则磁盘上的结构将处于不一致（inconsistent）的状态。因此，我们遇到了所有文件系统需要解决的问题：

关键问题：考虑到崩溃，如何更新磁盘

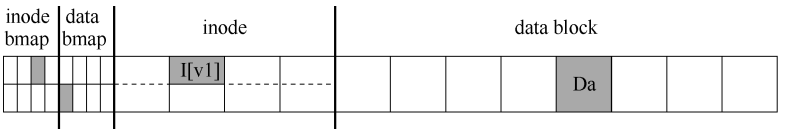
系统可能在任何两次写入之间崩溃或断电，因此磁盘上状态可能仅部分地更新。崩溃后，系统启动并希望再次挂载文件系统（以便访问文件等）。鉴于崩溃可能发生在任意时间点，如何确保文件系统将磁盘上的映像保持在合理的状态？

在本章中，我们将更详细地探讨这个问题，看看文件系统克服它的一些方法。我们将首先检查较老的文件系统采用的方法，即 fsck，文件系统检查程序（file system checker）。然后，我们将注意力转向另一种方法，称为日志记录（journaling，也称为预写日志，write-ahead logging），这种技术为每次写入增加一点开销，但可以更快地从崩溃或断电中恢复。我们将讨论日志的基本机制，包括 Linux ext3 [T98，PAA05]（一个相对现代的日志文件系统）实现的几种不同的日志。

42.1　一个详细的例子

为了开始对日志的调查，先看一个例子。我们需要一种工作负载（workload），它以某种方式更新磁盘结构。这里假设工作负载很简单：将单个数据块附加到原有文件。通过打开文件，调用 lseek()将文件偏移量移动到文件末尾，然后在关闭文件之前，向文件发出单个 4KB 写入来完成追加。

我们还假定磁盘上使用标准的简单文件系统结构，类似于之前看到的文件系统。这个小例子包括一个 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）。以下是该文件系统的示意图：



查看图中的结构，可以看到分配了一个 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。它将很快更新（由于上述工作负载）。

再来看看这个简化的 inode。在 I[v1]中，我们看到：

```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

在这个简化的 inode 中，文件的大小为 1（它有一个块位于其中），第一个直接指针指向块 4（文件的第一个数据块，Da），并且所有其他 3 个直接指针都被设置为 null（表示它们未被使用）。当然，真正的 inode 有更多的字段。更多相关信息，请参阅前面的章节。

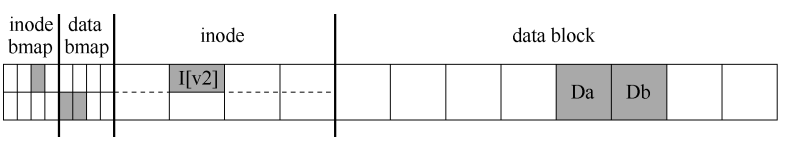
向文件追加内容时，要向它添加一个新数据块，因此必须更新 3 个磁盘上的结构：inode（必须指向新块，并且由于追加而具有更大的大小），新数据块 Db 和新版本的数据位图（称之为 B[v2]）表示新数据块已被分配。

因此，在系统的内存中，有 3 个块必须写入磁盘。更新的 inode（inode 版本 2，或简称为 I [v2]）现在看起来像这样：

```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

更新的数据位图（B[v2]）现在看起来像这样：00001100。最后，有数据块（Db），它只是用户放入文件的内容。

我们希望文件系统的最终磁盘映像如下所示：



要实现这种转变，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）。请注意，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。特别是，如果这些写入中的一个或两个完成后发生崩溃，而不是全部 3 个，则文件系统可能处于有趣的状态。

崩溃场景

为了更好地理解这个问题，让我们看一些崩溃情景示例。想象一下，只有一次写入成功。因此有以下 3 种可能的结果。

- **只将数据块（Db）写入磁盘**。在这种情况下，数据在磁盘上，但是没有指向它的 inode，也没有表示块已分配的位图。因此，就好像写入从未发生过一样。从文件系统崩溃一致性的角度来看，这种情况根本不是问题[①]。
- **只有更新的 inode（I[v2]）写入了磁盘**。在这种情况下，inode 指向磁盘地址（5），其中 Db 即将写入，但 Db 尚未写入。因此，如果我们信任该指针，我们将从磁盘读取垃圾数据（磁盘地址 5 的旧内容）。

此外，遇到了一个新问题，我们将它称为文件系统不一致（file-system inconsistency）。磁盘上的位图告诉我们数据块 5 尚未分配，但是 inode 说它已经分配了。文件系统数据结构中的这种不同意见，是文件系统的数据结构不一致。要使用文件系统，我们必须以某种方式解决这个问题。

- **只有更新后的位图（B [v2]）写入了磁盘**。在这种情况下，位图指示已分配块 5，但没有指向它的 inode。因此文件系统再次不一致。如果不解决，这种写入将导致空间泄露（space leak），因为文件系统永远不会使用块 5。

在这个向磁盘写入 3 次的尝试中，还有 3 种崩溃场景。在这些情况下，两次写入成功，最后一次失败。

- **inode（I[v2]）和位图（B[v2]）写入了磁盘，但没有写入数据（Db）**。在这种情况下，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。
- **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。
- **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。

① 但是，对于刚丢失一些数据的用户来说，这可能是一个问题！

崩溃一致性问题

希望从这些崩溃场景中，你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）。

42.2 解决方案 1：文件系统检查程序

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[①]。fsck 是一个 UNIX 工具，用于查找这些不一致并修复它们[M86]。在不同的系统上，存在检查和修复磁盘分区的类似工具。请注意，这种方法无法解决所有问题。例如，考虑上面的情况，文件系统看起来是一致的，但是 inode 指向垃圾数据。唯一真正的目标，是确保文件系统元数据内部一致。

工具 fsck 在许多阶段运行，如 McKusick 和 Kowalski 的论文[MK96]所述。它在文件系统挂载并可用之前运行（fsck 假定在运行时没有其他文件系统活动正在进行）。一旦完成，磁盘上的文件系统应该是一致的，因此可以让用户访问。

以下是 fsck 的基本总结。

- **超级块**：fsck 首先检查超级块是否合理，主要是进行健全性检查，例如确保文件系统大小大于分配的块数。通常，这些健全性检查的目的是找到一个可疑的（冲突的）超级块。在这种情况下，系统（或管理员）可以决定使用超级块的备用副本。
- **空闲块**：接下来，fsck 扫描 inode、间接块、双重间接块等，以了解当前在文件系统中分配的块。它利用这些知识生成正确版本的分配位图。因此，如果位图和 inode 之间存在任何不一致，则通过信任 inode 内的信息来解决它。对所有 inode 执行相同类型的检查，确保所有看起来像在用的 inode，都在 inode 位图中有标记。
- **inode 状态**：检查每个 inode 是否存在损坏或其他问题。例如，fsck 确保每个分配的 inode 具有有效的类型字段（即常规文件、目录、符号链接等）。如果 inode 字段存在问题，不易修复，则 inode 被认为是可疑的，并被 fsck 清除，inode 位图相应地更新。
- **inode 链接**：fsck 还会验证每个已分配的 inode 的链接数。你可能还记得，链接计数表示包含此特定文件的引用（即链接）的不同目录的数量。为了验证链接计数，fsck 从根目录开始扫描整个目录树，并为文件系统中的每个文件和目录构建自己的

① 发音为“eff-ess-see-kay”“eff-ess-check”，或者，如果你不喜欢这个工具，那就用“eff- suck”。是的，严肃的专业人士使用这个术语。

链接计数。如果新计算的计数与 inode 中找到的计数不匹配，则必须采取纠正措施，通常是修复 inode 中的计数。如果发现已分配的 inode 但没有目录引用它，则会将其移动到 lost + found 目录。

- **重复**：fsck 还检查重复指针，即两个不同的 inode 引用同一个块的情况。如果一个 inode 明显不好，可能会被清除。或者，可以复制指向的块，从而根据需要为每个 inode 提供其自己的副本。
- **坏块**：在扫描所有指针列表时，还会检查坏块指针。如果指针显然指向超出其有效范围的某个指针，则该指针被认为是“坏的”，例如，它的地址指向大于分区大小的块。在这种情况下，fsck 不能做任何太聪明的事情。它只是从 inode 或间接块中删除（清除）该指针。
- **目录检查**：fsck 不了解用户文件的内容。但是，目录包含由文件系统本身创建的特定格式的信息。因此，fsck 对每个目录的内容执行额外的完整性检查，确保“.”和“..”是前面的条目，目录条目中引用的每个 inode 都已分配，并确保整个层次结构中没有目录的引用超过一次。

如你所见，构建有效工作的 fsck 需要复杂的文件系统知识。确保这样的代码在所有情况下都能正常工作可能具有挑战性[G+08]。然而，fsck（和类似的方法）有一个更大的、也许更根本的问题：它们太慢了。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（尽管最近取得了进展[M+13]）。

在更高的层面上，fsck 的基本前提似乎有点不合理。考虑上面的示例，其中只有 3 个块写入磁盘。扫描整个磁盘，仅修复更新 3 个块期间出现的问题，这是非常昂贵的。这种情况类似于将你的钥匙放在卧室的地板上，然后从地下室开始，搜遍每个房间，执行“搜索整个房子找钥匙”的恢复算法。它有效，但很浪费。因此，随着磁盘（和 RAID）的增长，研究人员和从业者开始寻找其他解决方案。

42.3 解决方案 2：日志（或预写日志）

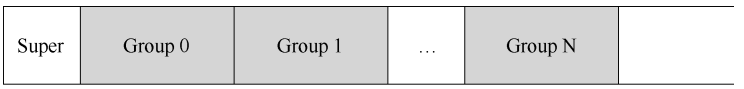
对于一致更新问题，最流行的解决方案可能是从数据库管理系统的世界中借鉴的一个想法。这种名为预写日志（write-ahead logging）的想法，是为了解决这类问题而发明的。在文件系统中，出于历史原因，我们通常将预写日志称为日志（journaling）。第一个实现它的文件系统是 Cedar [H87]，但许多现代文件系统都使用这个想法，包括 Linux ext3 和 ext4、reiserfs、IBM 的 JFS、SGI 的 XFS 和 Windows NTFS。

基本思路如下。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。

我们现在将描述 Linux ext3（一种流行的日志文件系统）如何将日志记录到文件系统中。大多数磁盘上的结构与 Linux ext2 相同，例如，磁盘被分成块组，每个块组都有一个 inode 和数据位图以及 inode 和数据块。新的关键结构是日志本身，它占用分区内或其他设备上的少量空间。因此，ext2 文件系统（没有日志）看起来像这样：



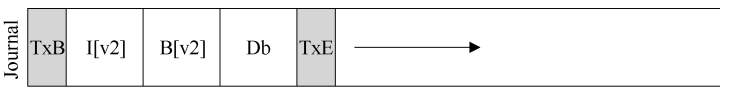
假设日志放在同一个文件系统映像中（虽然有时将它放在单独的设备上，或作为文件系统中的文件），带有日志的 ext3 文件系统如下所示：

真正的区别只是日志的存在，当然，还有它的使用方式。

数据日志

看一个简单的例子，来理解数据日志（data journaling）的工作原理。数据日志作为 Linux ext3 文件系统的一种模式提供，本讨论的大部分内容都来自于此。

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。这就是日志中的样子：



你可以看到，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（另一种想法，逻辑日志（logical logging），在日志中放置更紧凑的更新逻辑表示，例如，“这次更新希望将数据块 Db 追加到文件 X”，这有点复杂，但可以节省日志中的空间，并可能提高性能）。最后一个块（TxE）是该事务结束的标记，也会包含 TID。

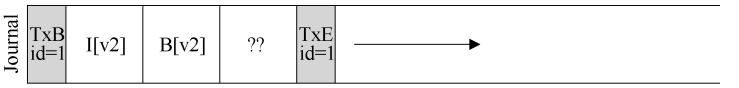
一旦这个事务安全地存在于磁盘上，我们就可以覆写文件系统中的旧结构了。这个过程称为加检查点（checkpointing）。因此，为了对文件系统加检查点（checkpoint，即让它与日志中即将进行的更新一致），我们将 I[v2]、B[v2]和 Db 写入其磁盘位置，如上所示。如果这些写入成功完成，我们已成功地为文件系统加上了检查点，基本上完成了。因此，我们的初始操作顺序如下。

1．**日志写入**：将事务（包括事务开始块，所有即将写入的数据和元数据更新以及事务结束块）写入日志，等待这些写入完成。

2．**加检查点**：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。

在我们的例子中，先将 TxB、I[v2]、B[v2]、Db 和 TxE 写入日志。这些写入完成后，我们将加检查点，将 I[v2]、B[v2]和 Db 写入磁盘上的最终位置，完成更新。

在写入日志期间发生崩溃时，事情变得有点棘手。在这里，我们试图将事务中的这些块（即 TxB、I[v2]、B[v2]、Db、TxE）写入磁盘。一种简单的方法是一次发出一个，等待每个完成，然后发出下一个。但是，这很慢。理想情况下，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，那么磁盘上会变成：



补充：强制写入磁盘

为了在两次磁盘写入之间强制执行顺序，现代文件系统必须采取一些额外的预防措施。在过去，强制在两个写入 A 和 B 之间进行顺序很简单：只需向磁盘发出 A 写入，等待磁盘在写入完成时中断 OS，然后发出写入 B。

由于磁盘中写入缓存的使用增加，事情变得有点复杂了。启用写入缓冲后（有时称为立即报告，immediate reporting），如果磁盘已经放入磁盘的内存缓存中、但尚未到达磁盘，磁盘就会通知操作系统写入完成。如果操作系统随后发出后续写入，则无法保证它在先前写入之后到达磁盘。因此，不再保证写入之间的顺序。一种解决方案是禁用写缓冲。然而，更现代的系统采取额外的预防措施，发出明确的写入屏障（write barrier）。这样的屏障，当它完成时，能确保在屏障之前发出的所有写入，先于在屏障之后发出的所有写入到达磁盘。

所有这些机制都需要对磁盘的正确操作有很大的信任。遗憾的是，最近的研究表明，为了提供“性能更高”的磁盘，一些磁盘制造商显然忽略了写屏障请求，从而使磁盘看起来运行速度更快，但存在操作错误的风险[C+13, R+11]。正如 Kahan 所说，快速几乎总是打败慢速，即使快速是错的。

为什么这是个问题？好吧，事务看起来像一个有效的事务（它有一个匹配序列号的开头和结尾）。此外，文件系统无法查看第四个块并知道它是错误的。毕竟，它是任意的用户数据。因此，如果系统现在重新启动并运行恢复，它将重放此事务，并无知地将垃圾块“??”的内容复制到 Db 应该存在的位置。这对文件中的任意用户数据不利。如果它发生在文件系统的关键部分上，例如超级块，可能会导致文件系统无法挂装，那就更糟了。

补充：优化日志写入

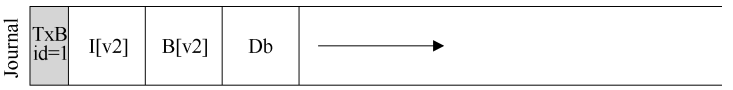
你可能已经注意到，写入日志的效率特别低。也就是说，文件系统首先必须写出事务开始块和事务的内容。只有在这些写入完成后，文件系统才能将事务结束块发送到磁盘。如果你考虑磁盘的工作方式，性能影响很明显：通常会产生额外的旋转（请考虑原因）。

我们以前的一个研究生 Vijayan Prabhakaran，用一个简单的想法解决了这个问题[P+05]。将事务写入日志时，在开始和结束块中包含日志内容的校验和。这样做可以使文件系统立即写入整个事务，而不会产生等待。如果在恢复期间，文件系统发现计算的校验和与事务中存储的校验和不匹配，则可以断定

在写入事务期间发生了崩溃，从而丢弃了文件系统更新。因此，通过写入协议和恢复系统中的小调整，文件系统可以实现更快的通用情况性能。最重要的是，系统更可靠了，因为来自日志的任何读取现在都受到校验和的保护。

这个简单的修复很吸引人，足以引起 Linux 文件系统开发人员的注意。他们后来将它合并到下一代 Linux 文件系统中，称为 Linux ext4（你猜对了！）。它现在可以在全球数百万台机器上运行，包括 Android 手持平台。因此，每次在许多基于 Linux 的系统上写入磁盘时，威斯康星大学开发的一些代码都会使你的系统更快、更可靠。

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，日志将看起来像这样（假设又是文件追加的工作负载）：



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：

此过程的一个重要方面是磁盘提供的原子性保证。事实证明，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。因此，我们当前更新文件系统的协议如下，3 个阶段中的每一个都标上了名称。

1．**日志写入**：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。

2．**日志提交**：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）。

3．**加检查点**：将更新内容（元数据和数据）写入其最终的磁盘位置。

恢复

现在来了解文件系统如何利用日志内容从崩溃中恢复（recover）。在这个更新序列期间，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（在上面的步骤 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。

请注意，即使在某些更新写入块的最终位置之后，在加检查点期间的任何时刻发生崩溃，都没问题。在最坏的情况下，其中一些更新只是在恢复期间再次执行。因为恢复是一