# 第 11 章　高级文件系统

## 11.1　简　　介

随着更新的和更快的计算机的出现，操作系统必须适应计算机硬件的发展，以便能充分利用这些硬件优势。有时候计算机的某一部分的发展可能比其他部分发展更快，这就会改变原来资源利用的平衡状态，因此操作系统必须相应地改变其实现策略。

从 20 世纪 80 年代早期开始，计算机工业在 CPU 速度、内存容量及访问速度方面获得了飞速的发展[Mash 87]。在 1982 年，UNIX 一般运行在 VAX 11/780 上，该机器有 1mips(每秒百万条指令)的 CPU 和 4～8M 的 RAM，可以有多个用户。到了 1995 年，100mips 的 CPU 相超过 32M 的 RAM 对于个人计算机早已很常见了。不幸的是硬盘技术没有跟上发展的潮流，以致于尽管硬盘变得越来越大，越来越便宜，而硬盘速度的提高甚至还不到 2 倍。造成这个问题的主要原因在于 UNIX 操作系统，因为它最初是为了相对较快的磁盘、小内存以及极慢的处理器而设计的。

在现代计算机上使用传统的文件系统使 I/O 系统受到严重的限制，根本不能充分利用高速 CPU 和大内存的优势。正如在[Stae 91]中所述，如果应用程序在一个系统上运行，假设有 c 秒用于 CPU 处理，i 秒用于 I/O 操作，那么仅靠提高 CPU 速度而得到的性能提高超不过(1-c/i)倍。如果 i 比 c 大，那么减少 c 只能产生很小的效果。因此必须想办法减少用于输入/输出的时间，而达到这个目标首要的便是改变文件系统的实现方法。

在 20 世纪 80 年代中后期，绝大多数的 UNIX 系统在它们的本地磁盘上的文件系统要么使用 s5fs，要么使用 FFS(见第 9 章)。这两种文件系统对一般的分时应用来说已经足够厂，但是当用于各种不同的商业应用时则会暴露出许多问题。vnode/vfs 接口使得人们很容易向 UNIX 中加入新的文件系统，但是该接口最初是用于小的专用文件系统中，它并没有试图去替代 s5fs 或 FFS。s5fs 和 FFS 的局限性最终促使人们开发了几种性能更好，功能更强的高级文件系统。到 20 世纪 90 年代初，这些高级文件系统赢得了主流 UNIX 的广泛支持。在本章中我们将时论传统文件系统的缺点，考虑一下解决它们的方法，最后再看一下几种主要的可以替代 sSfs 和 FFS 的文件系统。

## 11.2　传统文件系统的局限

因为 sSfs 的设计和结构非常简单，所以赢得了广泛的接受。然而，由于它速度慢，效率低，人们又设计了 FFS。但这两种文件系统都有很多限制，主要可以分为以下几类：

·性能　尽管 FFS 比 sSfs 性能更优越，然而用到商业文件系统中时 FFS 仍显不足。FFS 的磁盘布局使得它只能够利用磁盘带宽的一部分。而且，文件系统的内核算法要求进行很多同步 I/O 操作，这使得很多系统调用要花很长的时间才能完成。

·崩溃恢复　在系统崩溃时，高速缓存的数据和元数据有可能丢失掉，导致文件系统处于不一致状态。崩溃恢复由程序 fsck 来进行，它扫描整个文件系统，找出并且修复问题。对于大的磁盘，这个程序要运行很长时间，因为要检查并且重建整个磁盘，这就会使得崩溃恢复的时间长得让人难以接受。

·安全性　对一个文件的访问权限与用户和组 ID 有关。文件用户可能只允许他/她自己访问文件，可能允许某个组的所有用户，也可能让所有人都可以访问。在大型计算环境中，这种机制变得不是那么灵活，需要有一种更细粒度的访问控制机制。这种控制机制会涉及到访问控制表(ACL，access-controllist)，它可以让用户明确地允许或限制某些用户或组对文件的访问权限。UNIX 的 i 节点并没有被设计支持这些表，因此文件系统必须使用其他方法来

实现 ACL。这可能需要改变磁盘上的数据结构以及文件系统布局。

·尺寸　在传统文件系统中，有很多对文件系统和文件尺寸的限制是毫无必要的。文件和文件系统必须在一个磁盘分区内，不允许跨越不同的磁盘分区。我们可以将整个磁盘划为一个分区，但即使这样一般的磁盘也最多只有 1G。虽然这个尺寸对广大多数的应用程序来说足够了，但有些程序(在数据库或多媒体领域内)可能需要更大 的文件。实际上，文件尺寸不能超过 4G 的限制也早已被认为是过于苛刻了。

下面我们将详细地讨论性能和崩溃恢复的问题，分析其根本原因，找到解决这些问题的方法。

### 11.2.1　FFS 磁盘布局

不像 s5fs 那样，FFS 为了提高速度，对文件的块的分配采用最佳方法。它总是尽可能地为文件分配连续的数据块。当然，能否做到这一点与磁盘的已用空间和碎片的比例有关。经验证据[MoVo 91，McKu 84]表明，在磁盘使用率超过 90%以前这种分配策略十分有效。

然而，主要问题在于在连续扇区之间的旋转间隔。FFS 每次只能读或写一个数据块。对于一个顺序读文件的应用程序来说，内核要进行一系列读取单个块的操作。在连续两次读之前，内核必须检索下一个块是否在高速缓存里并且必要的时候发送 I/O 请求。其后果是如果两个块在磁盘的连续扇区内，那么在内核发出下一个读命令之前下一次数据块起始点早已转过去。下一次读取必须要等第二个块起始再次转到磁头底下时才能进行，这样效率就会十分低下。

为了避免这种情况，FFS 估计一下内核发送下次读命令需要的时间，同时计算一下磁头在这段时间要经过多少个扇区。这个扇区数便是旋转间隔。这样，文件数据块便交错地分布在磁盘上，其逻辑上连续的块被磁道上的旋转间隔块隔开。图 ll-1 显示了这一分布情况。对于一个典型的磁盘来说，磁头转圈要花 15ms，而内核两次读取请求之间需要 4s。因此如果块大小呈 iK 并且每个磁道有 8 个块，那么旋转间隔是 2。

尽管上面这种方法避免了等待磁头旋转一周的时间，它仍然只利用三分之一左右的磁盘带宽。如果将磁盘块尺寸增加到 8K，那么旋转间隔将会减小到 1，可以达到磁盘带宽的一半，这仍然没有达到磁盘所能达到的最大带宽，而唯一的限制因素是文件系统设计。如果文件系统在每次操作中读或写整个磁道数据，而不是每次读写一个块，那么其 I/O 吞吐量便可以达到实际带宽。

图 11-1　FFS 中磁道中的块分布

对很多磁盘来说，read 操作中不存在这样的问题。固为磁盘维护了一个高速缓存器，任何一次磁盘读都会将整个磁道的信息放到这个高速缓存器中。如果下一次操作需要从同一磁道中读取，磁盘可以以 I/O 总线的速度读取，而不用在旋转等待上浪费时间。磁盘高速缓存器一般都是写通的，这样每次写的内容在写操作返回之前便写回磁盘上相应的位置①。如果高速缓存不采用写通方式，当磁盘崩溃时可能会丢掉一些用户以为写入但实际没有写入的数据。因此，尽管磁盘高速缓存能提高读性能，写操作仍然受到旋转间隔的影响，并没有完全利用整个磁盘带宽。

### 11.2.2　写的主导性

几个对文件系统使用和访问方式的研究[Oust 85]表明对文件或元数据的读请求要比写请求多一到两倍，但是磁盘 I/O 请求则正好反过来，也即写操作的磁盘 I/O 请求个数远比读操作多。这种反常现象主要出于磁盘高速缓存(buffer cache)造成的。因为应用程序对文件的访问有很强的局部性，因此高速缓存命中率相当高(80～90%)，故不需要磁盘 I/O 请求便可以满足读请求。

正常情况下，写操作也是修改高速缓存中的数据复制，而不用磁盘 I/O。但是如果磁盘不进行周期性地更新的话，磁盘崩溃时便有可能丢失大量的数据。因此，大多数 UNIX 实现都

有一个守护进程定期地将被修改的数据块写回磁盘。而且有些操作需要对 i 节点，间接块直接进行同步更新，以保证文件系统在崩溃后能够恢复起来(下一节将详细地讨论这个问题)。由于上述这两个因素，写请求在磁盘操作占有主导地位。随着内存越来越大，高速缓存也不断增大，读操作仅占磁盘流量很少的一部分。

很多同步写操作实际上是没有必要的。因为对数据的引用呈现出很强的局部性，这就意味着同样的数据块可能马上就会被改变。而且，很多文件的生命周期很短——它们可能在几秒钟内经历创建、被访问和删除的全过程，对这些文件的同步写变得没有任何意义。

如果这种文件系统移植到 NFS 中则会导致更大的问题，因为 NFS 要求所有的写操作都必须保证内容正确，这样写操作通信量会更大，因为所有数据写入和元数据写入都必须要同步。

最后，磁头定位(将磁头移到正确的位置)使 I/O 操作花费代价更高，尽管在 FFS 中对一个文件的顺序访问几乎不用对磁头定位，但在分时环境(或充当文件服务器的系统)中有很多混合负载，这会导致更多的随机型磁盘定位操作。平均定位时间大概是连续 FFS 块旋转间隔的几倍。

正如 11.2.1 小节中所描述的那样，磁盘高速缓存能够消除大多数顺序读的旋转延时，但是却消除不厂写操作的旋转间隔。因为写操作占了磁盘操作的大多数，所以操作系统必须寻找其他方法来解决这个问题。

### 11.2.3 元数据更新

有此系统调用要改变元数据。为了防止由于系统崩溃而导致文件系统崩溃，对元数据的修改必须严格地按照一定的步骤来进行。例如，当一个文件被删除时(其最后一个链接被断开)，内核必须将其目录项删除掉，释放 i 节点，然后再释放文件所用的磁盘块。这些操作必须严格按照上面的顺序进行，只有这样才能保证在系统崩溃时保持文件系统的一致性。

假设在删除目录项之前将 i 节点释放掉，而系统恰好在这两个操作之间崩溃了。在重启的时候，目录将有一项指向一个未分配的 i 节点(因为该 i 节点已被释放)。如果将目录项删除掉，我们就可以保证链接计数(1inkcount)不为零而又没有被引用的节点不受损害，从而可以保证该文件的 i 节点可以被重用。相对来说这还不算严重,因为这可以通过 fsck 修正错误。在前一种情况下，如果 i 节点被分配给其他的文件，fsck 将不知道哪一个是合法的目录项。

类似的，假设正当截断一个文件时，文件系统在将修改的 i 节点写回磁盘之前将数据块释放掉了，那么释放掉的数据块有可能被分配给其他的文件，并且这个文件有可能比要截断的文件早一些将数据写入这些数据块。如果系统在此时崩溃，两个 i 节点将指向一些同样的数据块，导致用户可见的错误。这种情况 fsck 是不能修改复的，因为它根本不知道哪一个 i 节点是这些数据块的所有者。

在传统的文件系统中，要达到上面严格的顺序需要同步写操作。这导致了系统性能下降，主要是因为写操作并不是在连续的磁盘上进行的，所以需要定位，这个过程很耗时。更为糟糕的是 NFS 的操作要求对所有的数据块都进行同步写，其效率之低可想而知。很显然，我们需要尽量减少系统中同步写操作的数目，同时为了减少磁头定位时间尽量使写操作局部化。

### 11.2.4 崩溃恢复

正如上一小节所述，对元数据的写操作需要按照一定的顺序进行，这可以控制由系统崩溃而导致的损失，但却不能完全消除这些损失。大多数情况要求确保文件系统是可以恢复的，有时候硬件错误而导致有些磁盘扇区受损，这种错误根本没有可能恢复。fsck 是一个用户级的程序，它通过原始设备接口访问文件。这个程序可以在文件系统崩溃后将其重建。重建过程主要包括以下操作：

l.读取并且检查所有的 i 节点，建立一个已使用块的位图。

2.记录所有目录的 i 节点号和块地址。

3.使目录树结构有效，必须要确保所有的链接都已找小。

4.使目录内容有效，以便找到所有的文件。

5.如果有某些目录不能连接到状态 2 中所述的目录树上去，将该目录放到 lost+found 目录中去。

6.如果任何一个文件不能加到某个目录中去，将其放到 lost+found 目录中去。

7.检查每个柱面组的位图和总计数。

可以看到，fsck 要做大量的工作。如果一台机器有好几个大的文件系统，那么崩溃恢复时间会很长。在很多时候这么漫长的恢复时间是难以让人接受的，因此我们必须找到快速恢复的方法。

最后，fsck 的崩溃恢复是有限的，它只是将文件系统恢复到一致的状态。理想的情况是实行完全恢复，这就需要每一个操作应该使存储稳定之后再返回到用户。NFS 和其他一些非 UNIX 文件系统例如 MS-DOS 都采用了这种策略，但是这种方法性能十分低下。所以文件系统设计的更为合理的目标是在不降低性能的情况下，尽量减少由崩溃造成的损大。在 11.7 节中我们可以看到，这个目标是可以达到的。

## 11.3 文件系统簇(Sun-FFS)

要达到比较高的性能，一个简单的方法是每次 I/O 操作不要局限在一个块上，在 U-NIX 中大多数的文件访问是对文件连续地读或写，即使是跨多个文件系统调用也是如此。四此每次磁盘 I/O 只局限在一个块上(一般为 8K)显然是很浪费的。很多非 UNIX 文件系统每次给文件分配一个或多个在磁盘上物理连续的大区域。这样每次单个的磁盘操作就可以读或写文件的一大块。尽管 UNIX 的每次只读写一块的分配策略有几个好处，比如适于文件的动态增长，这种策略也有缺点，那就是顺序访问效率很低。

基于以上的考虑，SunOS 在 FFS 增加了文件簇的增强功能[McVo 91]这个新功能在后来也集成到 SVR4 和 4.4BSD 中去了。在本章中我们将这种性能提高的 FFS 实现称为 Sun-FFS。其目标是在不改变文件系统在磁盘上的结构的基础上，通过提高 I/O 操作粒度来达到高的性能。其实实现仅需要对内部内核例程作一些小的局部改变。FFS 磁盘分配程序通过预测将来的分配请求而为文件分配一些连续的块。在这方面它做得很好，由此 Sun-FFS 需要对磁盘分配例程做些改动。

Sun-FFS 将 rotdelay 因子设为 0,这是因为其目标就是避免由于旋转交错而带来的缺陷。在超级块中的 maxcontig 域是由旋转间隔块隔开的每个数据区中连续块的数目。底值通常被设为 l，但是当 rotdeday 为零时该值就没有用处了。于是 Sun-FFS 便使用该数据域来存储 clustersize。这样在不改变超级块结构的情况下，超级块便可以存储附加参数了。

当有一个读请求需要对磁盘进行访问时，最好是读取整个簇的数据。修改 bmap()的接口就可以做到这一点。在传统的 FFS 实现中，bmap()将一个逻辑块号做输入参数，返回该逻辑块的物理块号。Sun-FFS 改变了接口，使 bmap()返回一个附加值 contigsize，该值从指定的块开始，指定物理上连续的文件的范围大小。contigsize 最大值为 maxcontig，即使文件的数据大于连续数据也是如此。

Sun-FFS 使用参数 contigsize，每次读取整个簇的数据。除了基于整个簇的读取之外，该文件系统一般都是预读。在很多时候，分配程序找不到一个完整的簇，由 bmap()返回的 contigsize 便小于 maxcontig。这种预读的策略是基于 bmap()返回的 contigsize，而不是其实际的簇的尺寸。

簇的写需要对 ufs_putpage()例程做一下改变。ufs_putpage()的任务是将某页的内容写入磁盘，同时从内存中清除该页。在 Sun-FFS 中，该例程将页留在高速缓存中，向调用者返回成功便可。直到整个簇都在高速缓存中或者顺序写模式被中断时，才从高速缓存清除出所有页面。此时该例程调用 bmap()找到这些页的物理地址，一次将它们写回磁盘。如果分配程

序没有能够将这些页放在物理上连续的空间之内，ufs_putpage()将整个页的写分作几次完成，其中每次 bmap()都返回较小的值。

Sun-FFS 还增加其他一些限制来解决高速缓存清除的问题，但是我们上面所说的这些是与簇有关的方法中的主要方法，研究表明 Sun-FFS 在顺序读写性能方面比 FFS 提高了大约两倍，而随机访问大概和传统的 FFS 大致相等或者稍微快一点。不过这种方法并不适用于 NFS，因为 NFS 需要所有的改变都要同步地写到磁盘中去。为了使 NFS 同样也能从该方法中得益，我们有必要把 NFS 的集中写策略集成到 10.7.3 小节讲述的 NFS 中去。

## 11.4 日志方法

很多现代 UNlK 使用一种被称为记日志的方法来解决很多传统 UNIX 系统存在的缺陷，这些缺陷我们已经在 11.2 节中讨论过了。记日志方法的基本概念是将所有的文件改变记录在一个只许扩充的日志文件巾。日志是顺序写的，每次写一个大块，这样可以提高磁盘利用率，使性能更加优越。在文件系统崩溃后，只查看日志文件的尾部即可，这样就能够快速恢复，同时义能保证有高的可靠性。

当然，这种方法还是过分简单化了。尽管我们在上面讲的优点很吸引人，其中还有许多问题需要解决。现在已经有了很多日志文件系统了，它们的基本结构很不相同。下面我们看一下将它们区分开的基本原则，然后讨论一下一些重要的设计细节。

### 11.4.1 基本特征

要设计一个记日志文件系统必须要明白以下问题：

·记录什么　日志文件系统分为两类，一类记录所有的修改，另一类只记录元数据的修改。而在记录元数据的修改中可能又局限于只记录某些特定的操作。例如，它们可能不记录对文件时间戳，所有权或权限的修改，而只记录那些影响文件系统一致性的修改。

·要记录操作还是记录操作的结果 日志可能记录单个的操作，也可能记录操作的结果。前者很有用处，比如把对磁盘块分配位图的改变记录下来，因为每次改变只影响几个位，所以只记录操作会使日志变得简洁。但是如果要记录操作的结果，则需要把修改的块的全部内容写入日志中，这显然是不合适的。

·增添还是替代　增加了记日志功能的文件系统保留了传统文件系统的磁盘结构，例如 i 节点和超级块，把日志仅仅看作是附加的记录；而在日志结构文件系统中，唯有日志才能代表磁盘上的文件系统。当然，这种方法需要日志记录所有的信息数据及元数据。

·重做和撤消日志　有两种日志，分别为只允许重做(redo-only)日志和撤消重做(undo-redo)日志。第一种只记录修改的数据，而第二种既记录旧的数据值也记录新的数据值。第一种方法简化了崩溃恢复过程，但对日志的写操作和即时元数据更新有严格的顺序要求(详见 11.7.2 小节)。第二种日志更大一些，有更多的恢复机制，在使用时允降对日志文件并发操作。

·垃圾收集　有些日志文件系统的实现允许日志文件无限制扩展，到一定程度时，把日志的旧的部分转储到其他存储设备上去，但大多数的日志文件系统只允许有限长度的日志文件。这样日志文件便可以被看作一个逻辑上循环的文件，其过时的数据需要被清除掉。这种清除过程既可以在运行着的系统上进行，也可以通过独立的操作来完成。

·组提交　为了达到性能要求，文件系统必须在大的块上写日志文件，如果需要的话，可以把几次小的写操作进行合并，一次完成。在决定这种大块写操作频率和粒度时，我们需要在性能和可靠性之间做一些折衷，因为没有写入的数据很有可能由于文件系统崩溃而丢失掉。

·检索　在日志结构的文件系统中，我们需要一种有效的方法从日志中检索数据。一般想法是使高速缓存足够大以便满足大多数的读操作，这样就几乎不用访问磁盘了，但是我们

仍然必须确保万一高速缓存没有命中，系统必须在一合理的时间内对此作出处理。这就需要有效的机制来检索文件中的任一数据块。

## 11.5 日志结构文件系统

日志结构文件系统使用一个顺序的，只能扩展的日志作为其唯一的磁盘结构。具基本思想是将一系列对文件的改变记录到一个大的日志表项中，并且用一个操作将其写回磁盘去。这需要对文件系统的磁盘结构和访问这些结构的内核程序做彻底的修改。

这种方法的优点也是显然的。因为所有的写操作都是在日志的尾部进行的，所以它们都是顺序的，省去了磁头定位的时间。每次日志写操作都要传输大量的数据，一般情况下是整个磁道。这样就取消了旋转间隔，因此文件系统就可以充分利用磁盘带宽。数据和元数据可以在一次原子写中完成，这种方法很适合于基于事务处理的系统。文件系统的崩溃恢复也很快，因为文件系统只需要找到日志中的最后一项并且根据该项中的内容来重建文件系统即可。所有在该项之后的部分提交的操作被丢弃掉了。

如果我们只是向日志中写入数据，上面的写方法已经很完善了。但是如果希望从日志中检索数据，该怎么办?从磁盘中检索数据的传统方法不再适用，此时需要找到我们需要的数据的日志。可以考虑使用一大块内存高速缓存来解决这个问题，记住我们的假设是，现代系统有大的廉价的内存和高速处理器，但硬盘很慢。如果一个系统有很大的数据高速缓存，那么等到它运行稳定之后，该高速缓存的命中率甚至高于 90%。但是总有一些块要从磁盘上读取(当系统初始起动时会有许多这样的数据)，对于这些数据，我们必须想办法在合理的时间内将其读入内存中。因为日志结构文件系统必须要有有效的方法来解决这个问题。

4.4BSD 日志文系统(BSD-LFS[Seh 93])是基于 sprite 操作系统[Rose 90a]的类似工作开放的。在本章的剩下几节中，我们介绍该文件系统的结构及其实现，同时看一下它是怎样达到可靠性和性能的目标的。

## 11.6 4.4BSD 日志文件系统

在 BSD-LFS 中，整个磁盘都由日志管理，日志是文件系统在磁盘上的唯一代表。所有的写操作都在日志文件的尾部进行，所有的垃圾收集由一个清除进程进行，这样就可以循环使用日志。日志文件分成一些固定长度的段(一般其大小为 0.5M)。每个段都有一个指向下一个段的指针,形成了逻辑连续但物理上并不一定连续的块(当跨段边界操作时可能要求盘物理上连续)。

BSD-LFS 保留了传统文件的一些结构，主要有目录和 i 节点结构，以及为了对大文件进行逻辑编址而设置的间接块。因此，一旦找到文件的 i 节点之后，其数据块便可以通过常规方法找到。现在的问题是如何找到 i 节点。最初的 FFS 将 i 节点固定配置在磁盘上，将它们放在不同的柱面组中，根据 i 节点号便可以很容易的算出 i 节点的地址来。

而在 BSD-LFS 中，i 节点是作为日志的一部分写入磁盘的，并没有固定的地址。每次修改 i 节点后，文件系统都将其放在日志的不同位置。这就需要一个附加的数据结构 i 节点映射表(inode map)来保存每个 i 节点的当前磁盘位置。BSD-LFS 将该映射表存放在物理内存中，每隔一定时间将其写回日志中。

尽管 BSD-LFS 试图每次写一个完整的段，但通常情况下这是不可能的，当内存不足(缓存已满)或者有 fsync 请求以及进行 NFS 操作时，可能只将部分段的内容写入。因此，描述磁盘物理分区的段必须分成一个或多个部分段，这些部分段才是写入日志的原子数据单位(图 11-2)。

图 11-2 BSD-LFS 的日志结构

每一个部分段都有一个段头部，它主要用于崩溃恢复和垃圾收集。它包含如下信息：

·校验和，用来检测介质错误和未完成的写操作。

·在部分段中每个 i 节点的磁盘地址。

·对于每个在该部分段中有数据块的文件，其 i 节点号和 i 节点版本以及逻辑块号。

·创建时间、标志，等等。

系统同时也维护着一个段使用表。该表存放着每段中过时数据的字节数以及该段上次被修改的时间。清除进程使用该表来选择要清除哪些段。

### 11.6.1 写日志

BSD-LFS 收集所有的被修改过的未写回磁盘的块，直到这些块的总尺寸可以接近一个段的尺寸。当内存不足或者调用 fsync 或有 NFS 请求时，可能不得不将部分段写入。如果磁盘控制程序支持分散－聚集 I/O 从非连续的内存地址中获取数据，可以直接将数据从高速缓存中写出。否则，内核必须为本次传输分配 64K 的临时缓冲区。

在准备数据传输的时候，文件系统根据磁盘块在文件内的逻辑块号将这些磁盘块排序。磁盘地址也被赋于给这些块，与此同时也必须将 i 节点(如果需要的话可以是间接块)改变以反映这些地址变化。这些 i 节点和其他一些修改过的元数据块被绑定在同一段内。

因为日志是只允许扩展的，所以每当磁盘块被修改后，它总要被写往另外一个新地址。这就意味着日志中的一些老的数据块的复制变为过时，清除进程可能将它们回收。图 11-3 展示了修改文件数据和 i 节点的操作。

图 11-3 写一个 BSD-LFS 文件

每次写操作都会将高速缓存中的所有过时数据清除出去，这就意味着日志中包含着为完成崩溃恢复所需的所有信息。i 节点映射表和段使用表中包含的信息都是些冗余信息，但这些信息可以从日志中得到，尽管这个过程很慢。这两个结构存放在一个称为 ifile 的常规只读文件中，ifile 可以像其他一些文件一样为用户所访问，不过它有点特殊，因为它不能写在它所在的段空间以外的空间去。系统定义了 check points，每当到达 check points 时便将 ifile 写到磁盘上去。把 ifile 视为普通文件可以使系统中的 i 节点数动态地变化。

### 11.6.2 数据掐索

要使得文件系统效率更高，我们可以使用一个大的高速缓存，这样不用访问磁盘便可以满足大多数的请求。同时必须要有效的解决缓存不命中的情况，BSD-LFS 的数据结构使该目标可以很容易地达到。就像在 FFS 中那样，BSD-LFS 遍历日录中(每次一个分量)，并且获取下一个分量的 i 节点号，从而对文件进行定位。唯一的区别是 BSD-LFS 定位磁盘 i 节点的方法：它使用 i 节点号作为索引在 i 节点映射表中查找地址，而不是根据 i 节点号直接计算出磁盘地址。

在高速缓存中，系统根据 v 节点和逻辑块号来唯一确定一个数据块，同时依据这两个值将该数据块加入到哈希表中。间接块不能像在 FFS 中那样被直接使用，因为在 FFS 中，间接块用磁盘设备的 v 节点和物理块号标识。而在 LFS 中，直到要写一个块时才赋予其地址，所以不能方便地将其映射。为了解决这个问题，LFS 使用负的 i 节点号来表示间接块，每一个间接块号是它指向的第一个块的块号的负值。每一个二次间接块是它所指向的第一个间接块号减 1，依此类推。

### 11.6.3 崩溃恢复

BSD-LFS 中的崩溃恢复速度快而容易。恢复的第一步要找到最后一个检验点，并且利同它恢复内存 i 节点映射表和段使用表。所有在检验点后对这两个结构的改变都要恢复，方法是对检验点后的部分重新执行一遍。在重新执行每一个部分段以前，必须要比较时间戳以确保该部分段是在检验点之后写的，同时也要确保当我们命中一个老的段时恢复工作已经完成。在段概要中的校验和确保每一个部分段的完整性和一致性。如果不是这样的话，将该段从日志中丢弃，恢复过程结束。当部分段没有被完全写回时其中的数据会丢失掉，从上次写

之后进行的改变也会丢失掉，不过这是唯一丢失掉的数据。

这个恢复过程很快，花的时间是从上次检测点开始到崩溃的时间的一部分。不过，该方法不能检测到毁坏一个或多个磁盘扇区的硬错误。有一个类似于 fsck 的程序来进行完全的文件系统验证，该程序在由系统重启通过重放日志一直在后台运行。

### 11.6.4 清除进程

当日志到达磁盘尾部时，它会重新返回开头。当这种情况发生时，文件系统必须保证没有冲掉日志中有用的数据。这就需要有一种垃圾收集机制，它会从一个段中将活跃数据收集起来，将它们移到一个新位置，这样该段便可以重用了。

垃圾收集进程可以同其他系统活动并行地进行。其主要工作是读取日志的一段并且找到其合法表项。当日志表项的后面跟着同一对象的新表项(由于对相同的文件和目录进行相应的操作)或者当相应的对象被删除掉时(例如，当文件被删除时)，该表项就会变得无效。如果段中包含有合法的项，它们会被收集并且写到日志文件的结尾<图 11-4)。于是整个段便可以重用了。

图 11-4 元数据日志中的垃圾收集

在 BSD-LFS 中，一个被称之为清除器的用户进程使用 ifile 和一组特殊的系统调用进行垃圾收集工作。它首先检测段使用表，选定一个要清除的段，然后将段读到其地址空间中。对于每一个部分段，清除器检查每-个块，确定哪些块仍然是活的。同样，它也检测 i 节点，方法是将该 i 节点的版本号和在 i 节点映射表中的版本号相比较。活的 i 节点和块必须要写回文件系统并要保证 i 节点的修改和访问次数没有被改变。LFS 将它们写到下一部分段中的新位置，最后，它将过时的 i 节点和块丢弃掉，然后将段置为可重用。

为了使得清除器(.cleaner)能够完成其任务?系统增加了四十新的系统调用：

· lfs_bmapv 为每一个<inode，logical block>偶对计算磁盘地址。如果一个块的地址跟段中正在被清除的块的地址相同，则该块是活的。

· lfs_illsrkv 在不更新 i 节点的修改和访问时间的情况下，向日志增加一组磁盘块。

· lfs_segwait 进入睡眠状态，直到过了一定的时间或者另外一段被写。

· lfs_segelean 将段标记为清除，这样它就可以重用了。

### 11.6.5 分析

BSD-LFS 可能在三个方面有一些问题。首先，当一个目录操作涉及到多于一个元数据对象时，这些修改不可能都在同一个部分段中。这就需要附加的代码来检杳到这种情况，由于系统崩溃只有一部分修改操作被保留下来时，损失的内容可以很快地恢复。

其次，当要往段里写内容时才分配磁盘块空间，而不是第一次在内存创建块的时候，所以必须要很细心地计算空闲空间，否则用户可能看到 write 系统调用成功返回，而内核却发现在磁盘根本没有空间。

最后，要想使 BSD-LFS 有效地执行操作就必须要有大量的物理内存，这些物理内存不仅仅用作高速缓存，而且也用作日志和垃圾收集所用到的大型数据结构和数据移动缓存。

[Selt 93]和[Selt 95]详尽地描述了 BSD-LFS 和传统的 FFS 以及 Sun-FFS 性能的实验比较。结果表明 BSD，LFS 在很多方面的性能都比传统 FFS 好(除了在高度多道环境中，其性能稍逊一筹外)。同 Sun-FFS 比较，BSD-LFS 在大部分涉及元数据的实验中占有很明显的优势。对 read 和 write 性能和一般多用户基准程序的测试结果不甚明了。Sun-FFS 在大多数的 I/O 密集的基准程序下，特别是当 BSD-LFS 的清除器工作时，比 BSD-LFS 要快得多。在一般的像 Andrew 基准程序的多用户仿真的比较中，两者差不太多。

BSD-LFS 在性能上提高是很让人怀疑的,因为 Sun-FFS 在实现上所花的代价只有 BSD-LFS 的一小部分，但却可以获得相同的或者更高的性能。LFS 不仅需要重写文件系统，而且也要重写一大部分涉及到磁盘结构的应用程序，比如像 newfs 和 fsck。BSD-LFS 的真正优点在于

它的崩溃恢复速度很快，同时其元数据操作的性能也有所提高。11.7 节中显示了元数据日志是怎样以更小的代价获得相同的性能的。

另外一个值得注意的日志结构的文件系统是随处写文件布局(WAFL)系统，网络应用公司(Network Appliance Corporation)在它们的专用 NFS 服务器——FASeiver 家族中使用了该文件系统[Hitz 94]。WAFL 将一个日志结构的文件系统同非易失性存储器(NV-RAM)和一个 RAID-4 磁盘序列集成起来，对于 NFS 访问的响应时间极快。WAFL 增加了一个很有用的称为快照的应用程序。快照是对一个动态文件系统瞬时只读备份。文件系统可以维护很多它自身的快照，这些快照是在不同时刻取下的。至于同时可以维护多少个快照，则要受到空间的限制。用户通过访问快照可以检索到文件的旧版本或者将不小心误删的文件恢复回来，因为快照是文件系统在某一瞬间的一致性的写照，系统管理员可以使用快照来备份文件系统。

## 11.7 元数据日志

在元数据日志系统中，日志扩充了文件系统的正常表示方式。这极大地简化了实现过程，因为那些不对文件系统进行改变的操作不需要任何修改。文件系统的磁盘结构也不用改变，可以使用跟访问传统文件系统相同的方法从磁盘上访问数据和元数据。只有在崩溃恢复和垃圾收集时才去读取日志。文件系统的其余代码以只追加方式使用日志。

这种方法可以得到日志的最基本的好处，也即快速的崩溃恢复和快速的元数据操作，而没有日志结构文件系统的缺点(复杂，需要重新写很多应用程序，垃圾收集会降低性能)。元数据日志系统对正常的 I/O 操作影响很小，但足需要仔细地实现以防日志开销降低整个系统的性能。

元数据日志一般只记录对 i 节点，目录块和间接块的修改。它也有可能记录对超级块，柱面组概况和磁盘分配位图的修改，否则系统在崩愦恢复中需要重新构造这些信息。

日志可以放在文件系统内部，也可以作为一个独立的对象驻留在文件系统外部。至于选择什么类型主要是要考虑磁盘的使用率和性能。例如，在 Cedar 文件系统[Hagm 87]中，日志是一个固定尺寸的环形文件,放在靠近磁盘的中间柱面组的磁盘块上(这样可以使得访问速度最快)。日志文件同其他文件一样有一个名字和一个 i 节点，不用使用任何特殊的机制便可以访问它。在 Calaveras 文件系统[Vaha 95]中，一台机器上的所有文件系统共享同一个驻留在一个独立磁盘上的日志。Veritas 文件系统[Yage 95]将日志同文件系统分开，但是允许系统管理员来决定是否将日志单独放在一个专用磁盘上。

### 11.7.1 正常操作

作为第一个例子,我们来看一下一个只允许重做、新数值(日志记录变化了的对象的新值)日志策略，这跟 Cedar 文件系统[Hagm 87]中的策略一样。日志不处理文件数据的写，文件数据的写是按一般的方式来进行的。图 11-5 描述厂 setattr 操作，该操作修改一个 i 节点。内核执行以下的操作：

1.更新已缓存的 i 节点副本，将其标识为已修改。

2.建立一个日志表项，其头部可以用来标识修改的对象，紧接着是对象的新内容。

3.将表项写到日志的尾部。当写结束时操作必须被提交到磁盘，

4.将 i 节点在稍后的某一时间内写回其实际磁盘位置。这称为就地更新。

从上面这个例子中我们可以看到日志是怎样影响系统性能的。一方面，每一次元数据更新必须分两次才能写回磁盘，一次是写日志项，一次是在就地更新期间。另一方面，因为就地更新一般要延迟一段时间，所以经常被取消或者进行批处理。例如，一个 i 节点在被刷新到磁盘之前可能要被修改好几次,在同-个磁盘上的多个 i 节点可以在一次 I/O 操作中集中写回磁盘。要使得元数据日志的性能比较合理，必须减少就地更新的次数，这可以降低日志开销。

图 11-5　元数据日志的实现

批处理也可以应用于日志写。很多文件操作修改多个元数据对象。例如，mkdir 修改父目录和其 i 节点，同时分配和初始化一个新的目录和 i 节点。有些文件系统将由于单个操作造成的改变合并到一个日志项中，而有些功能更强大，可以将多个操作引起的结果写入-个日志表项中。

这些方法不仅会影响到性能，还会影响到文件系统的可靠性和一致性保证。如果文件系统崩溃了，那么所有未写到日志中的改变都将丢失掉。如果文件系统是用作 NFS 访问，那么直到改变已经提交到日志后它才能响应客户的请求。如果有多个操作对同一个对象进行修改，那么必须将这些操作串行，以防有不一致性出现。11.7.2 小节更详细地讨论了这个问题。

因为日志大小是固定的，所以当写操作达到日志尾部时要绕回开头，文件系统必须要确保不会冲掉有用的数据。一个日志项只有在其所有对象都刷新它在磁盘上的位置(就地更新)之后才被认做是有效的。处理绕回情况有两种方法。一种是像 BSD-LFS 那样进行显式的垃圾收集。清理程序必须始终要比记日志领先一步，通过将活动项移至日志尾部释放空间[Hagm 87]。一种更简单的方法是将日志的尺寸设为足够大。这样在峰值负载时就不会冲掉还没有提交的就地更新，此时垃圾收集就完全没有必要了。其实日志的尺寸也不需要太大，对于小型服务器或者分时系统来说，几兆大小就足够了。

### 11.7.2　日志的一致性

在日志文件系统中，如果一个操作修改了多个对象，那么文件系统必须要考虑到一致性的问题(见 11.2.3 小节)。同时它也必须确保对同一组对象的多个并发操作保持一致。在奉小节，我们介绍一下在重做日志中的一致性问题；11.8.3 小节将讨论撤消重做日志的一致性。

重做日志的崩溃恢复方法是重新运行日志，将每个项写到其实际磁盘位置。这需要假设日志中包含着所有元数据对象的最新副本。因此在正常操作期间，文件系统不能进行就地更新，直到对象已经被写到日志中去。这样的口志通常被称为意向日志[Yage 95]，因为每个项中包含的对象正是文件系统想要写回磁盘的对象。

假设有一个操作修改两个元数据对象 A 和 B，顺序是先 A 后 B，一个健壮的文件系统对于多个更新或者提供顺序保证，或者提供事务性保证。顺序保证意味着在崩溃恢复后，只有在磁盘上有对象 A 的新内容时才能有对象 B 的新内容。事务性保证要求更强一些，它要求两个对象的修改在崩溃恢复之后要么都在磁盘上，要么都不在。

我们可以通过延迟所有元数据的就地更新，仅在它们的日志项已经被提交到磁盘时才更新，这样便可以满足顺序的要求。系统按照对象修改或创建的顺序将它们写入日志。在前面的例子中，对象 A 表项写入日期要比对象 B 表项的写入日期早(可以将它们写入一个项中)。这样，即使 B 的就地更新比 A 早，在日志中 A 的项也比 B 早。这样便可以将数据改变的顺序保留下来。

在重做日志中，事务性保证要求在 A 和 B 的日志项没有成功写到日志之前这两个对象都不能被刷新到磁盘上。如果这两个块被绑定在同一个日志项中，有没有上述要求无关紧要，但是这种情况(两个块被绑定在同一个日志项中)并不总是发生。假设对象 A 的就地更新早于 B 的日志项写入日志的时间，而系统就在这两个操作之间崩溃，那么系统便不能恢复 A 的旧副本和 B 的新副本。因此我们必须在写回任一缓存项之前将这两个日志项强制写回磁盘。日志也要增加有关信息，使得系统能够辨认出两次更新属于同一个事务处理，这样在崩溃恢复时我们就不会只重放部分的事务了。

实际上，系统对于在同一个对象上的并发操作有很强的要求。假设修改一个对象，那么在将对象写到日志之前读取该对象是错误的。图 11-6 中显示了潜在的竞争条件。进程 p1 修改于对象 A，正想将它写回，顺序是先写回日志再写回磁盘。在它写回之前，进程 p2 读取对象 A，然后基于 A 的内容修改对象 B。接着将 B 写回日志，下一步便想将 B 写回磁盘。如果此

时系统崩溃了，日志中包含有 B 的新内容，但是对象 A 的新值却既不在日志中，也不在磁盘中。因为 B 的改变依赖于 A 的改变，所以上述这种情况会导致潜在的不一致性。

图 11-6　重做日志中的竞争条件

再看一个更具体的例子，假设 p1 正从目录中删除一个文件，而进程 p2 正在该目录中以相同的名字创建一个文件。p1 从目录的块 A 删除文件名字。p2 搜索该目录，没有发现有与其要创建的文件重名的文件，于是便在目录的块 B 中创建一个目录项。在系统从崩溃中恢复后，系统中有旧块 A 和新块 B，两个块中都包含有指向同一个文件名的目录项。

### 11.7.3　崩溃恢复

如果系统崩溃，可以通过重放日志恢复文件系统，使用日志项来更新磁盘中元数据对象。本小节描述在重做日志中的恢复。11.8.3 小节描述与重做撤消日志中相关的问题。因为日志是绕回使用的，所以其主要问题在于决定日志的起始点和终止点。[Vaha 95]描述了一种方案。在正常操作过程中，文件系统对每一项都赋予一个表项号。这个号是单调上升的，并且与该项在日志中的位置相对应。当日志绕回时，表项号继续增长。因此在任何时候，表项号相它在日志中的位置的关系可以由下列公式得到(其距日志起始点的偏移,以 512 字节为单位度量)：

项位置＝项号%日志的尺寸

文件系统一直监视日志中第一个和最后一个活动项的表项号，并且将这两个值写到每个日志项的头部。为了重放日志，系统必须首先找到最高项号对应的项，这对应的是日志的尾部，这个项中也保存有当前日志头部项号。

找到日志的头部和尾部后，系统便开始崩溃恢复过程，将每项中的元数据对象写到它们在磁盘中的实际位置。项头部包含每一对象在表项中的物理位置，这种方法使得数据损失仅限于那些还没有被写到日志中的项。如果一个日志项不完整，系统很容易辨认出来，因为每个日志项都有校验和或者概要记录。这样的不完整的项在崩溃恢复过程将被系统丢弃掉。

崩溃恢复时间不依赖于文件系统的大小，而与在崩溃时的日志的实际大小成比例。这主要依赖于系统的负载和就地更新的频率。元数据日志系统的崩溃恢复需要几秒钟，而那些使用 fsck 的系统则需要几分钟甚至几个小时。不过，运行一些磁盘检测程序还是有必要的，因为日志不能排除由于硬盘错误带来的错误。这些程序在系统恢复之后在后台执行。

### 11.7.4　分析

元数据日志具有一般日志的主要优点，即快速崩溃恢复和快速元数据操作，同时却又不像日志结构文件系统那样实现起来很困难。系统的崩溃恢复只需要重新执行日志，将其元数据对象写回它们的磁盘位置,这个过程需要的时间仅仅是那些像 fsck 等磁盘检测程序的一小部分。

元数据日志也使得那些修改多个元数据对象的操作速度加快，mkdir 和 rmdir 就是这样的操作。之所以有这样的效果，主要是因为它将所有的修改收集起来，放在单个日志项中，因此减少了同步写的次数。以这种方式，它也为相关元数据改变提供了排序和事务保证(依赖于具体实现)这使得元数据日志系统比传统文件系统更加健壮。

元数据日志系统对总体性能的影响尚不可知。日志对于那些不修改文件系统的操作影响很小，对于数据写影响也很小。总之，日志文件系统通过延迟就地更新而减少这类更新的次数，要想获得性能上的提高，这种减少更新次数获得的好处要足以抵消日志带来的额外开销才行。

[Vaha 95]说明日志有可能成为性能瓶颈，同时提出了几种方法来避免这个问题。它也描述厂一个比较两个文件系统的实验，其中一个文件系统使用了日志，而另外一个没有。日志文件系统在元数据密集的基准程序中比另外一个快得多，但是在仿真多用户 NFS 访问的基准程序 LADDIS 中[Witt 93]，它只比没有日志的文件系统勉强好一点。

元数据日志也有很多重要的缺陷和限制。尽管它使得在系统崩溃中丢失的元数据达到最少，对普通文件数据的损失却无能为力(除了运行 update 守护程序外)。这意味着我们不能保证所有的操作都能保持事务性一致——如果一个操作同时改变了文件数据和其 i 节点，那么这两个操作是不能原子更新的。因此系统崩溃可能导致事务处理中的两个分量只有一个可以恢复。

总之，元数据日志在不改变文件系统的磁盘结构的情况下，增强了文件系统的健壮性，使得崩溃恢复时间缩短，同时也有适当的性能提高。另外，它很容易实现，因为只有涉及到将元数据写磁盘的那部分代码需要改写。

在 UNIX 使用群内部，元数据日志系统和日志结构的文件系统之间的争论由来已久。元数据日志赢得了这场争论，已经成为很多成功的商业版本的基础，包括 Veritas 公司的 Veritas 文件系统(VxFS)，IBM 的日志文件系统 JFS，和 Transarc 的 Episode 文件系统(见 11.8 节)。而且，因为元数据日志没有影响到数据传输代码，因此可以将它同其他一些增强性能的措施结合起来，比如文件系统簇(见 11.3 节)或 NFS 聚集写(见 10.7.3 小节)，这样的文件系统在普通文件数据和元数据操作方面性能都很好。

## 11.8 Episode 文件系统

1989 年，Transarc 公司从卡内基·梅隆大学接管了 Andrew 文件系统的开发工作(AFS，在 10.15 节中介绍过)。AFS 演化成为 Episode 文件系统[Chut 92]，并且成为 OSF 的分布计算环境(DCE)的本地文件系统组件。Episode 提供了一些传统 UNIX 文件系统没有提供的几个高级特性，比如增强的安全性，大文件日志，以及将存储抽象从逻辑文件系统结构中分离出来。11.8 节描述厂 DCE 的分布文件系统(DFS)。在本节中，我们讨论 Episode 的结构和特性。

### 11.8.1 基本抽象

Episode 引入了一些新的文件系统抽象符，主要有聚集(aggregates)，容器(containers)，文件集(filesets)和 a 节点(anode)。聚集是分区的一种抽象，是一个在逻辑上连续的磁盘块数组。它对文件系统的其他部分隐藏了磁盘的物理分区细节。一个聚集可能由一个或多个物理磁盘分区组成②，它可以透明地提供像磁盘镜像和分块之类强大功能。同时它允许一个文件跨越多于一个的物理磁盘，从而可以创建更大的文件。

文件集(fileset)是一个逻辑文件系统，其中包含一个目录树，该目录树的头部是文件集根目录。每个文件集都可以被单独安装和输出③。一个聚集可能包含一个或多个文件集，每个文件集可以从一个位置转移到另一个位置，即使此时系统正在运行或者这些文件系统在用于正常文件操作。图 11-7 显示了聚集，文件集和物理磁盘之间的关系。

一个容器是一个可以存储数据的对象。它由一些块组成。每个文件集都驻留在一个容器内，容器中保存着文件集的所有文件数据和元数据。每个聚集有三个附加的容器，分别用于位图、日志和聚集文件集表，这些将在下面介绍。

图 11-7 Episode 中的存储组织

a 节点同 UNIXi 节点类似。Episode 的每个文件和容器都有一个 a 节点。

### 11.8.2 结构

聚集由几个容器组成，文件集容器保存着文件集的所有文件和 a 节点。且节点驻留在容器开头的文件集 a 节点表中，其后是数据和间接块。容器在聚集中并不占有连续的地址空间，可以很方便地动态缩小和扩展。因此文件块地址指的是在聚集中的块号，而不是在容器中的。

位图容器可以完成聚集宽度的块分配。对于聚集中的每个片段，位图容器保存着该片段的信息，包括片段是否已被分配，以及用于存储日志数据或存储非日志数据，后面这条信息对某些特殊函数很有用。在日志片段用于非日志数据或者非日志片段用于存储日志信息时必须执行这些特殊函数。

日志容器中包含有聚集的一个取消重做日志。取消重做日志的优点将在 11.8.3 小节中讨论。该日志大小固定，可以循环使用。现在的实现将日志放在跟该日志所代表的文件系统相同的聚集上，但并非严格要求如此。

聚集文件集表(图 11-8)中包含着聚集中的每个容器的超级块和 a 节点的信息。目录项通过文件集 ID 和在文件集中的 a 节点号来引用文件。在查找文件时，首先在聚集文件集表中找到文件集的 a 节点，然后再在文件集的 a 节点表中找到需要的文件。当然，要是比较恰当地使用高速缓存会加快上述的大部分操作的。

容器有三种类型的存储方式一内置式、片段方式和块方式。每个 a 点都有附加信息，内置式将小量的数据存储在其中。这对于符号链接，访问控制表和小文件很有用。在片段方式中，几个小的容器可能共享一个单独的磁盘块。块方式支持大容器，并且支持四级间接块。这样文件最大可为 2^31 个磁盘块④。

图 11-8　聚集文件集表和文件集

### 11.8.3　记日志

Episode 使用了一个重做撤消元数据日志，该日志提供了很强的事务性保证(在 11.7.2 小节中描述)。重做撤消日志提供了很大的灵活性，因为每个项中同时存储了对象的新值和旧值。崩溃恢复时，文件系统可以有两种选择，一种是重放日志，将日志中的新值写到相应的磁盘对象中；一种是返回到更新前状态，将日志中旧的值写到相应的磁盘对象中。

在重做撤消日志中的事务性保证需要一个两阶段加锁协议。该协议将事务处理涉及到的所有对象锁定，直到整个事务都已提交给磁盘才解锁，这样就可以保证其他的事务不能读取未提交的数据。但是这会降低系统的并发性，使得系统性能让人难以忍受。Episode 使用了一个叫做等价类的机制来避免此类现象的发生。每个等价类中包含了涉及到同一个元数据对象的所有事务。对于等价类中的所有事务，要么它们全部被提交，要么一个也不提交。

在崩溃后，恢复过程将所有完整的等价类重放一遍，而将所有不完整的等价类中的所有事务撤消。这样在正常操作期间就可以保证高度一致性，可是这种方法使得每个项的尺寸增大了一倍，从而增加了日志磁盘的 I/O 通信量，使得日志恢复变得复杂。

在 Episode 中，高速缓存同日志程序紧密地结合在一起。高层函数不直接修改缓存区，而是调用日志函数来间接访问。日志将缓存区和日志项联系起来，如果缓存区的日志项还没有成功地写到日志中，缓存区就不能被刷新到磁盘。

### 11.8.4　其他特性

克隆(cloning) 通过一个文件集克隆的进程，文件集可以被复制或者从一个聚集移到另一个聚集中。文件集中的每个 a 节点被单个克隆，克隆通过 copy-on-write 技术和原来的 a 节点共享数据块。克隆的文件集是只读的，并且和原来的文件集驻留在同一个聚集中。克隆过程很快，并且不会于扰对源文件集的访问。克隆被广泛地应用于系统管理工具中，例如后备程序就是对克隆进行操作，而不是对源文件本身。如果源文件集中的任何一个块被修改，那么必须重新做一个副本，克隆的版本应该一直是最新的，安全性　Episode 提供了 POSlX 方式的访问控制表(ACL)，这种方式比传统 UNIX 的用户一组一其他权限机制更通用。每个文件和目录都可以有一个 ACL 与之相联。ACL 是一组表项的列表。每项包括一个用户 ID 或组 ID 以及一组赋予该组或用户的权限。对每个对象有六种不同的访问权限——读、写、执行、控制、插入和删除[OSF 93]。如果一个用户拥有对象的控制权限，那么他可以修改其 ACL。插入和删除权限仅仅适用于目录，分别用于在该目录中创建和删除文件。一组标准通配符可以使得一个 ACL 项适于多个用户或者组。例如，下列项赋予用户 rohan 对与该表项相对应的目录中的文件有读、写、执行和插入的权限。

user:rohan:rwx-i-

- 每一个小横线表示没有赋给用户某权限。在上面的例子中，rohan 没有控制和删除的

权
限。

## 11.9 监视器

文件系统的各个策略主要体现在命名，访问控制和存储等方面，这些操作的语义对文件系统中的所有文件都适用。为了某些特殊的目的，我们经常需要重载某些缺省策略，下面便是一些例子：

·使用户实现不同的访问控制机制。

·监视以及记录对某一特定文件的所有访问。

·自动接收邮件。

·将文件以压缩的方式存储，在使用时自动将其解压缩。

在 Washington 大学开发的 FFS 扩展中[Bers 88]提供了上述功能。基本的想法就是将一个称为监视器的用户级进程同一个文件或目录联系起来。该进程选择性地拦截对文件或目录的操作，并且对相应的功能给出自己的实现。监视器进程没有任何特殊权限，对访问文件的应用程序是完全透明的，并且仅仅对于某些选定的操作才调用重载操作。

新加的系统调用 wdlink 可以将一个监视器进程同一个文件联系起来。加了监视器的文件被称为受保护文件，wdlink 的参数指定了文件名以及监视器程序名字。在 BSD UNIX 中，监视器程序的名字被保存它 i 节点的一个 20 字节的保留区中以备将来使用。为了突破只有 20 个字节的限制，wdlink 中的第三个参数名字可以指向在一个称为/wdogs 的公共目录中，该目录项中包含有真正监视器程序的符号链接。

当进程试图打开一个受保护文件时，内核向监视器进程发送一个消息(如果监视器还没有启动，则首先将其启动)。监视器可以使用它自己的策略来允许或拒绝对文件的访问，也可以将决定权返回给内核。如果监视器允许打开文件，它通知内核它要保护的操作有哪些。这组被保护的操作因文件的不同打开实例而不同，因此可以为同一个文件提供多个视图。

一旦文件打开，每当用户试图调用一个受保护的操作，内核便将其传递到监视器(图 11-9)。监视器的反应有三种：

·执行操作，这可能要在内核和监视器之间传递附加数据(比如读或写操作)。为了避免循环，监视器可以直接访问它保护的文件。

·拒绝操作，传回一个错误代码。

·简单地响应操作，然后请求内核以一般的方式执行操作。在这种情况下，监视器在延迟对内核的操作之前可能进行一些附加的处理。比如记账。

图 11-9 监视操作

### 11.9.1 目录监视器

监视器也可以同一个目录联系起来。这样它便可以保护对目录的操作，并且控制对目录内的所有文件的访问权限(因为访问控制在一个路径名内的每一个目录上执行)。目录监视器有很多特殊的权力。缺省情况下，它保护目录下所有没有监视器与之对应的文件。

这样就会有很多有趣的应用程序。只要监视器允许，用户可以访问受保护目录中不存在的文件。在这种情况下，所有对这个不存在的文件的请求都不会回送给内核。

### 11.9.2 消息通道

内核和监视器之间的通信是通过消息传递来进行的。每个监视器同一个唯一的监视器消息通道(WMC)对应。WMC 是由新的系统调用 createwmc 创建的，该调用返回一个文件描述符，监视器可以使用这个描述符从内核接收到发往内核的消息。

每个消息包含一个类型域，一个会话标识符以及消息的内容。每一个文件的打开实例同监视器组成一个会话。图 11-10 描述了内核所维护的数据结构。受保护文件的打开文件表项

指向一个全局会话表项。而全局会话表的这一项又指向内核 WMC 的尾部(WMC 中包含未读消息的队列)。WMC 也指向监视器进程。

图 11-10　监视器数据结构

监视器首先读取消息，然后使用 createwmc 返回的文件描述符将应答信息返回。这样就可以映射到 WMC 在打开文件表中的表项，而该表项又指回在内核端的 WMC。因此监视器和内核都可以访问消息队列，从中得到消息或者将消息放入其中。

一个主控监视器进程管理着所有的监视器进程,它控制监视器的创建(当被保护文件打开的时候)和终止(当被保护文件被最后关闭时)，同时，那些经常被使用的监视器可以一直保持活跃(即使没有与之相关的文件打开)，这样就可以避免频繁地启动监视器，

### 11.9.3　应用

最初关于监视器的应用主要有：

wdacl　　将访问控制表同一个文件对应起来。一个监视器可以对多个文件的访问进行控制。

wdcompact　提供在线压缩和解压缩。

wdbiff　　监视用户的邮箱，当新邮件到达时通知用户。该功能还可以进一步扩展，使得监视器可以自动回答或者转发邮件。

wdview　　为不同的用户提供同一目录的不同视图。

wddate　　允许用户从一个文件读取当前日期和时间。文件本身并没有数据，每当文件被访问时监视器便读取系统时钟。

为文件树提供图形视图的用户界面也可以获益与监视器。当用户创建或者删除文件时，监视器请求用户界面重画视图来反映目录的最新状态。就像这些例子所述，监视器在很多方面为文件系统的扩展提供了一种通用的机制。在现代操作系统中，在用户级对单个的文件系统函数重新定义是十分有用的。

## 11.10　4.4BSD 入口文件系统

有了监视器，用户级进程就可以拦截其他进程对被保护文件的操作，4.4BSD 入口文件系统[Stev 95]也提供了类似的功能。它定义了进程可以打开的所有文件的名字空间。当进程打开该文件系统的一个文件时，内核向入口守护进程发送一个消息，该守护进程便可以处理打开请求，然后返回给进程一个描述符。入口守护进程而不是文件系统定义了一组合法的文件名及对它们的解释。

入口守护进程创建一个 UNIX 套接字[Leff 89]并且调用 listen 系统调用，接受针对套接字的连接请求。接着它便安装入口文件系统，一般是安装到/p 目录。然后便进入一个循环，在循环中调用 accept 来等待连接请求，当请求到达时便处理它们。

假设一个用户打开入口文件系统的一个文件。便会发生下列事件(见图 11-11)：

图 11-11　在入口文件系统打开文件

1.内核首先调用 namei()解析文件名。当解析程序通过入口文件系统的安装点时，它调用 portal_lookup()解析剩下的路径名。

2.portal_lookup()分配一个新的 v 节点，将路径名保存到 v 节点的私有数据对象中。

3.内核在 v 节点上调用 VOP_OPEN 操作，这会导致 portal_open()函数的调用。

4.partal_open()将路径名传递到从 accept 系统调用返回的入口守护进程中。

5.入口守护进程处理文件名，产生一个文件描述符。

6.守护进程通过内核创建的套接字对连接将描述符返回给调用者，

7.内核将描述符复制到调用者的描述符表的第一个空闲槽中。

8.入口守护进程拆除连接，调用 accept 等待其他的连接请求。

一般情况下，针对每一条请求，守护进程都要创建一个新进程来处理，子进程执行 5～7 步，然后退出，中断连接，父进程在创建子进程之后随即调用 accept。

### 11.10.1 使用入口

我们可以用很多方法使用入口文件系统。入口守护进程决定文件系统应该提供的功能以及它应该如何解释名串空间。一个重要的应用就是在 3.2.4 小节中提到的连接服务器。这个服务器为其他进程打开网络连接。进程可以使用入口，打开一个称为/p/tcp/node/service 的文件，从而创建一个 TCP 连接。其中 node 是要连接的远程主机的名字，service 是调用者希望要访问的 TCP 服务(比如 ftp 或 rlogin)。例如，打开文件/P/tcp/archana/ftp 便可以打开一个跟 archana 节点上的 ftp 服务器的连接。

守护进程完成建立一个连接所需要的所有工作。它决定远程主机的网络地址，同那个节点上的端口映射器协调决定该服务使用的端口号，然后创建一个 TCP 套接字，接着连接这个服务器。它将连接的文件描述符传回到调用进程，调用进程便可以使用这个描述符来同服务器进行通信。

这使得那些幼稚的应用程序也可以使用 TCP 连接。幼稚的应用程序是指那些仅使用 stdin，stdout 和 stderr，而不使用其他设备特殊功能的应用程序。例如，通过打开合适的端口文件，用户可以将 shell 或 awk 脚本的输出重定向到远程节点上。

同监视器类似，入口文件系统允许用户进程拦截其他进程发出的文件操作，然后给出自己的实现。不过，这两者之间有几个重要的区别。入口守护进程只拦截 open 系统调用，而监视器可以根据需要拦截很多操作；监视器也可以拦截一个操作，执行某些操作，然后请求内核完成剩下的操作。最后，入口守护进程定义它自己的名字空间。之所以这是可能的完全是因为，在 4.4BSD 中，在遇到安装点时，namei()将路径名的其余部分传递到 portal_lookup()中。尽管目录监视器可以以有限的方式扩展名字空间，监视器却一般只能在现有的文件层次上操作。

## 11.11 可堆叠文件系统层次

为了开发一个灵活的 UNIX 文件系统框架，vnode/vfs 接口是向这个目标迈进的重要一步，根据该接口已开发出很多新的文件系统，我们在 9～11 章已阐述了。但是，该接口也有很多限制：

- 该接口在各个 UNIX 变体之间并不统一。正如 8.11 节所述，SVR4，4.4BSD 和 OSF 的接口定义的操作的语义很不相同。例如，在 SVR4 中 VOP_LOOKUP 操作每次解析路径名的一个分量，而在 4.4BSD 中则一次解析几个分量。
- 即使对于同一供应商，随着版本的变化，接口也随之变化。例如，SunOS 4.0 用 VOP_GETPAGE 和 VOP_PUTPAGE 之类的页操作来代表 VOP_BMAP 和 VOP_BREAD 之类的高速缓存操作。SVR4 还增加了新的操作，比如 VOP_RWLOCK 和 VOP_RWUNLOCK 等，这些操作都被集成到 SunOS 后来的发行版中。
- 文件系统与存储管理子系统是高度互依赖的。如果没有充分的理解内存管理的体系结构，就不能设计出通用目的的文件系统。
- 接口并不像原来预想的那样不透明。内核要直接访问 v 节点中的许多域，而不是通过过程式的接口访问这些域。因此要想跟以前版本实现二进制兼容，就很难改变以前的 v 节点结构。
- 接口不支持继承。新的文件系统不能继承一个现有文件系统的某些功能。
- 接口不是可扩展的。文件系统不能加入新的函数或者改变现有操作的语义。

上述因素阻碍了文件系统技术的发展。要开发一个新的通用目的的文件系统，像 Transarc 的 Episode 文件系统或 Veritas 公司的 Veritas 文件系统，是一件很闲难的事情。

要把文件系统移植到不同的 UNIX 变体之上，以及移植到同一开发商的操作系统的新版本之上需要很大的开发队伍，而且，很多开发商不想开发全部的文件系统；相反，他们只想在已有的实现上增加一些功能，比如复制，加密和解密或者访问控制表等。

UCLA[Heid 94]和 SunSoft[Rose 99b，Skin 93]各自独立地开发文件系统，结果产生了一种新的可堆叠文件系统层次。它可以为模块式文件系统提供更好的支持。UCLA 的实现已经被集成到 4.4BSD 中，而 SunSoft 仍处于原型开发阶段。在本节，我们描述可堆叠文件系统的一些重要特征以及使用这种方法的很多应用程序。

### 11.11.1 框架和接口

有了 vfs 接口之后，各种不同的文件系统就可以共存于同一台机器之上。vfs 接口定义了一组操作，每个文件系统都可以对各个操作给出不同的实现。当用户对文件调用某个操作时，内核便动态地将该操作定向到该文件所属的文件系统中去。这个文件系统为该操作给出完整的实现。

在可堆叠层次框架中，多个文件系统可以相互安装。每个 v 节点栈代表一个文件，堆栈中的每个 v 节点表示一个文件系统。当用户调用一个文件操作时，内核将操作传到顶部 v 节点。该节点可能执行下面两个操作之一，彻底执行操作，将结果传回给调用者；进行一些处理，将操作传递给堆栈中的下一个 v 节点。这样，每个操作有可能通过所有的层。返回的时候，结果由下往上遍历所有的层，从而每个 v 节点都有机会再进行一些附加的处理。

使用这种方法我们就可以渐进地开发文件系统。例如，一个开发商可能提供一个加密-解密模块，将其放到所有物理层的顶部(图 11-12)。该模块拦截所有的 I/O 操作，如果是写操作它便将数据加密，如果是读操作，它便将数据解密。而所有其他的操作可以直接传递到底层。

图 11-12 加密-解密层

堆栈方式也允许扇入扇出配置。一个扇入式堆栈允许多个高层节点使用同一个低层节点。例如，一个压缩层可以在写操作时将数据压缩，在读操作时将数据解压缩。一个备份程序在将数据备份到磁带时可能希望直接读取压缩数据。这样便形成了图 11-13 所示的扇入结构。

扇出式堆栈允许一个高层节点控制多个低层节点。这可用于层次式存储管理(HSM)层。该层可以将最近被访问的文件放在本地磁盘上，并且将很少使用的文件迁移到光盘或者磁带上(见图 11-14)。为了跟踪文件使用情况，需要时可以从第 3 级存储设备上下载文件，HSM 层拦截每一次文件访问。[Webb 93]给出一个 HSM 实现，其框架同时具有堆栈层和监视器的功能。

图 11-14 层次存储管理器

### 11.11.2 SunSoft 原型

[Rose 90b]介绍了 Sun 公司最初在可堆叠式 v 节点方面的工作,这项工作先是移交给 UNIX 国际可堆叠文件工作组[Rose 92]，后来又重新移交到 Sun 公司手中。SunSoft 原型[Skin 93]解决于[Rose 90b]接口中的很多问题和限制。

在这个实现中，一个 i 节点中只包含着一个指向 p 节点的链表，链表中的每个节点代表着堆叠在那个节点上的文件系统(见图 11-15)。p 节点中包含着指向 vfs，vnodeops 向量及其私有数据的指针。每个操作先传递到最顶部的节点，如果需要的话再传递到低层的节点上。

在原型开发过程中要解决如下的问题：

·在当前的接口中，内核能够直接访问 v 节点的很多数据域。这些数据域必须被移到私有数据对象中，内核只能通过过程式接口访问这些对象。

·一个 v 节点可能把对其他 v 节点的引用作为其私有数据的一部分。例如，一个文件系统根目录的 v 节点可能包含有安装点的引用。

·vfs 的操作必须也要能够传递到低层中去。为了做到这一点，要将许多 vfs 的操作转

换为可以在任意一个文件上调用的 v 节点操作。

图 11-15　SunSoft 原型

·在当前接口中一个操作可能会操作多个 v 节点。为了能够正确的运行，必须将其分解为若干 v 节点上的子操作。例如，VOP_LINK 必须分为两个操作：一种是在文件 v 节点上的操作，主要是得到其文件 ID 并且增加其链接计数：另外一种是在目录 v 节点上的操作，主要是为文件增加表项。

·需要一个程序来保证由同一个高层操作调用的子操作的原子性。

·由于页面现在从属于多个 v 节点，页面缓存的<vnode，offset>名字空间不能够很好地映射到堆叠 v 节点接口上。与虚存管理系统的接口必须要重新改写。

## 11.12　4.4BSD 文件系统接口

4.4BSD 的虚拟文件系统接口是基于 UCLA Ficus 文件系统在可堆叠层次方面的工作[Heid 94]开发的，8.11.2 小节和 11.10 小节描述了 4.4BSD 文件系统的其他一些特性。本节主要讲述处理堆叠的接口以及基于此的一些有趣的文件系统实现，

在 4.4BSD 中，系统调用 mount 将一个新文件系统层加到 v 节点栈的顶部。unmount 将顶部的文件系统从堆栈中移掉。就像在 SunSoft 模型中那样，每一个操作首先到达顶层。每一层都可以结束操作将结果返回，或者将操作传递到下一层(该层可以进行一些附加的操作，也可以没有)。

4.4BSD 允许一个文件系统层附加到文件系统名字空间的多个位置。这样就可以使用不同路径访问同一个文件(不必使用独立的链接)。而且，在堆栈中的其他层对每一个安装点可能都不同，因而导致对同一个操作有着不同的语义。这样在 11.11.1 小节中所说的扇入式结构就可以实现了。例如，一个文件系统可能被安装到/direct 和/compress 上，可以放一个压缩层到/compress 之上。这样，用户就可以通过访问/compress 中文件完成在线压缩和解压缩。backup 程序可以通过/direct 来直接访问文件，绕开解压缩过程。

文件系统可以为接口增加新操作。当系统启动时，内核动态地将 vnodeops 向量构建为每个文件系统所支持操作的并集。为了达到这个目的，所有的文件系统使用一个叫做 bypass 的函数来处理未知的操作。bypass 函数将操作和它的参数传到下一层。因为函数不知道参数的个数和类型，4.4BSD 将对一个 v 节点的所有操作放到一个参数的结构中。接着文件系统将指向这个结构的指针当做对所有操作的参数传递。当某层不能识别该操作时就将参数指针传向下一层。如果某一层能够识别该操作，那么它也知道如果解释这个结构。

下面描述了两个可堆叠式文件系统的有趣应用。

### 11.12.1　Nullfs 和并集安装(Union Mount)文件系统

nullfs 文件系统[McKu 95]是一个允许将文件层次树的任何一棵子树都可以被安装到文件系统的其他任何位置上的文件系统。这样子树中的每一个文件都两个路径名。它将大部分的操作传递到原来的文件系统。该方法可以以某种有趣的方式使用。例如，用户可以将分散在很多地方的文件集中到一个目录下，这样使得文件看起来好象是在同一棵子树上。　　并集安装文件系统[Pend 95]提供厂与半透明文件系统(在 9.11.4 小节中讲述)类似的功能。它提供了一个安装在它上面的文件系统的并集。最顶部的层在逻辑上是最近修改，同时这也是唯一一个可以修改的层。当用户查找一个文件时，内核逐层向下找，直到发现包含该文件的层。如果用户试图修改文件，内核首先把它复制到最顶部的层。就像在 TFS 中那样，如果一个用户删除一个文件，内核在最顶部层创建-个失明表项，这样在随后的查找中系统就可以避免继续查找其下面的层次了。此外还有一些特殊的操作可以绕过并且删除失明表项，从而可以使得那些被误删的文件重新恢复。

## 11.13 小　结

在本章中，我们讨论了几种高级文件系统。这些文件系统有的是整个替换掉子诸如 FFS 和 s5fs 之类的现有实现，而有的只是以不同的方式对原来的实现进行了功能扩充。这些文件系统性能更加优越，崩溃恢复时间缩短，可靠性增加，功能增强。它们中的一些已经在商业竞争中获得了广泛的接受 ;现在大多数商业 UNIX 发行版中都加厂使用某种日志形式的增强型文件系统。

vhode/vfs 接口是一种很重要的技术。该接口使得文件系统的许多新实现能够集成到 UNIX 内核中。可堆叠式层次框架探讨了 v 节点接口的许多限制，并且允许以渐增方式开发文件系统。4.4BSD 已经采用了这种方法，很多商业供应商正在探讨它的价值。

## 11.14 练　习

1.为什么 FFS 使用一个 rotdelay 参数来间隔磁盘块?它是怎样假设磁盘使用状况和缓存区使用状况的?

2.文件系统成簇对一个 NFS 服务器的性能有什么样的影响?

3.文件系统成簇和写聚集之间有什么区别(见 10.7.3 小节)?它们各自适合用在什么地方?在什么时候将它们结合起来会效率更高?

4.文件系统成簇降低了非易失存储器的优点吗?在一个支持文件系统成簇的系统中怎样使用 NV-RAM 好处最大?

5.延迟磁盘写的好处是什么?

6.假设一个文件系统同步地写所有的元数据，而将数据写延迟，直到更新守护进程运行，这会导致什么样的安全问题?为了避免这个问题，什么样的数据块应该同步写?

7.一个日志结构文件系统在非峰值使用期间调度垃圾收集进程,这样可以提高性能吗?这会给系统的使用加上什么样的限制?

8.文件系统可以使用一个内存位图来跟踪日志的活动和过时的部分，讨论一下这种方法是否适用于日志结构文件系统，是否适用于元数据日志文件系统?

9.假设一个元数据文件系统将 i 节点相目录块的改变记录到日志中，但是同步地将间接块写出，设想出一种在崩溃后会导致不一致文件系统的情况。

10.在一个元数据日志文件系统中，单个日志能否将所有对文件的修改都保存下来?这种方法的优点和缺点是什么?

11.将文件系统的元数据日志跟该文件系统放到同一个物理磁盘上有没有优点?

12.为什么 Episode 使用了两阶段加锁协议?为什么 Cedar 文件系统不需要这种协议?

13.如果一个用户想将所有其他用户访问他的文件的情况都记录下来,他可以使用监视器达到他的目的吗?可以使用入口文件系统?它们对系统行为有什么影响?

14.考虑一个包含并集安装目录的磁盘，如果磁盘满了，用户试图通过将低层的一组文件删除来释放空间，这会导致什么样的问题?

① 某些现代 SCSI 磁盘高速缓存一个磁道一个磁道地写数据，利用停电时驱动器的旋转能量将缓存的数据写入磁道。

② IBM 的日志文件系统(JFS)是首次使用逻辑卷来扩展磁盘分区的 UNIX 文件系统。

③ 目前 DCE 上具只允许一个聚集中的所有文件集一起输出，

④ 文件尺寸进一步受限于(2^32 × 分片大小)。