CSP 论文

File System 的一致性相关技术研究和思考

论文作者	丁峰
论文作者	陈渤
论文作者	邓毓峰
提交日期	2018 年 1 月 10 日

File System 的一致性相关技术研究和思考

摘 要

保障 crash 之后文件系统的一致性是文件系统可靠性的一个重要指标，其中，一致性包括了元数据（metadata）的一致性和文件数据（data）的一致性。针对文件系统元数据一致性保护的软件技术从原始的 FSCK 一致性检测，发展到写前记日志的 Journaling、基于依赖关系的 Soft Updates、copy-on-write 等，提供了不同角度的一致性保护策略；而对于应用程序一致性的保护，文件系统在 crash 后无法避免文件数据的丢失，但可以通过保证操作的顺序性从而减少程序数据不一致状态的发生，应用程序也需要实现相对应的更新协议和恢复协议，让数据在发生 crash 后可以恢复到一致状态。

文件系统特性在不同场景下会有不同要求，存在一致性与性能（如吞吐量、延迟）之间的取舍与折中问题。近年来，许多研究学者围绕不同应用场景下系统一致性问题，提出了许多优秀的文件系统结构。本文将围绕这些一致性技术以及具体场景下的文件系统实现展开进一步的讨论，并对未来文件系统的发展给一致性带来的挑战提出见解和展望。

关键词：File System 一致性，FSCK， Journaling，Soft Updates，Copy-on-write

目录

第一章 背景

操作系统中负责管理和存储文件信息的软件部分称为文件管理系统，简称文件系统，用于控制数据的存储和检索。一般的文件系统结构如下图所示：

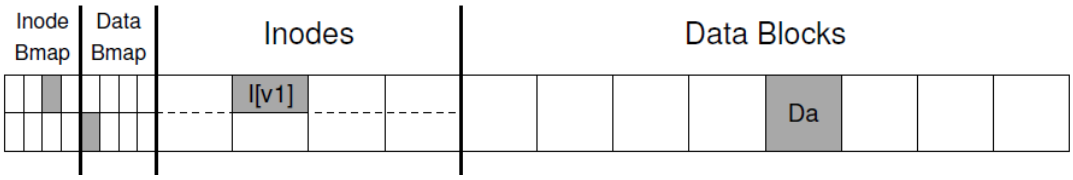


图 1-1 文件系统结构

如图 1-1 所示，文件系统包括用于指明原始数据存储结构的元数据部分（包括 Inode Bmap、Data Bmap 和 Inodes）和数据部分（Data Blocks）。现实中，可能导致文件系统发生 crash 的场景包括了：

- 电源断电或意外关闭电源
- 内核出现软件错误
- 计算机硬件故障
- ……

为了长期维持可靠存储，文件系统必须在遭遇非预期的系统 crash 时，保证元数据的一致性。假设文件系统原来存在着一个文件，现在要为该文件追加一块数据，则需要对原文件的元数据 Inode 和 Data Bmap 进行修改，并将数据写入 Data Blocks。在磁盘上这三个操作是独立的，并且由于文件系统对操作的重排序，所以如果在执行这三个操作期间发生了断电，则可能只完成了其中一个或者两个操作，这可能造成文件系统数据结构的不一致，也可能造成空间泄漏，更严重的是可能将垃圾数据返回给用户。

为了解决文件系统的一致性问题，许多学者提出了各种解决方案。一种简单的方案是在文件系统 crash 后，对整个文件系统进行扫描，检测其不一致的地方并进行修改。在 UNIX 中，检查和修复磁盘分区不一致性的工具 FSCK 运用的就是这种思想。这种方案耗时长，并且随着磁盘大小的增大而增加。一种更为普遍并且广泛应用的方案是 Journaling 技术，这种方案的思想源于数据库技术。通过在对磁盘进行更新之前先将操作记录在日志中，如果发生 crash 后，可以根据日志重新执行更新操作来确保文件系统的一致性。虽然 Journaling 技术被广泛运用于当今的文件系统，但其在系统性能，延迟等方面还待改进，特别是在多核技术、分布式技术被广泛运用的情况下，迎来新的挑战。文件系统的一致性保障技术，除了 FSCK、Journaling 外，还包括 Soft Updates、Copy-on-write 等新方法。随着硬件技术的升级，文件系统及其一致性保障又迎来了新的变革。

本文的结构如下：第二章，总结了保障文件系统一致性的相关技术；第三章，针对具体的应用场景，描述了文件系统的相关改进方案以及多核拓展目标的实现，并且讨论文件系统如何保障应用程序的一致性以及对程序一致性的检验；第四章，对未来文件系统的发展给一致性带来的挑战提出见解和展望。最后是对整篇文章的总结和讨论。

第二章 文件系统一致性相关技术

2.1 文件系统元数据一致性保障

文件系统中对元数据的操作主要包括了对文件和目录的创建、删除和重命名等，这些元数据操作都会对文件系统的结构进行修改。保护文件系统的一致性指的是在发生系统 crash 之后，使文件系统可以恢复到一致性状态。

如何在文件系统发生 crash 后保持系统一致性是众多学者研究的重点，由此产生了一些一致性技术，如 Journal，基于延迟写的 Soft Updates，Copy-on-write 等，这些技术在保持文件系统元数据一致性方面所使用的方法不尽相同，因而对文件系统特性的支持以及性能的影响也不一样，本章将着重介绍这些技术。

2.1.1 FSCK

描述：

早期的文件系统采取了一种简单的方法来确保 crash 一致性。其基本思想是先让不一致的事件发生，然后在重新启动时修复，这个方法的经典例子是 FSCK，主要内容如下：

1、对 superblock 进行完整性检查，例如确保文件系统的大小大于已经分配的块数。系统管理员决定是否使用超级块的替代副本。

2、扫描 Inode、间接块、双重间接块等，了解当前在文件系统内分配了哪些块。以 Inode 信息为准，生成分配位图的正确版本。

3、检查每个 Inode 是否有损坏或其他问题。尝试修复 Inode，清除无法修复的 Inode, 更新位图。

4、验证每个分配的 Inode 的链接数量。以新计算的计数为准，修改 Inode 的链接数量。将链接数为 0 的 Inode 移至 lost+found 目录。

5、检查重复的指针，即两个不同 Inode 引用同一个块的情况。清除明显损坏的 Inode。或复制指向的块，让每个 Inode 拥有自己的副本。

6、检查坏块指针。删除明显越界的指针。

7、检查目录。对每个目录的内容执行额外的完整性检查，确保“.”和“..”是在最前面，目录条目中引用的每个 Inode 都被分配，在整个层级中目录只链接一次。

优点：

方法简单，能够保证文件系统的一致性。

缺点：

1、构建一个 FSCK 需要复杂的文件系统知识;

2、全局扫描，运行速度慢。对于非常大的磁盘卷，扫描整个磁盘以查找所有分配的块，并读取整个目录树可能需要几分钟或几小时；

3、非理性。当只有三个块被写入磁盘，却通过扫描整个磁盘来解决更新过程中出现的问题。

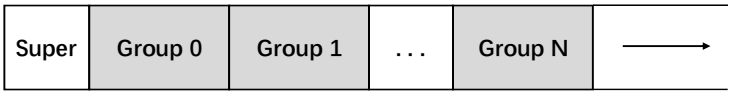
应用：成为普遍存在的用于检查和修复磁盘不一致的 UNIX 工具[24]。

2.1.2 Journaling

描述：

Journaling 是文件系统一致性更新问题的一种最普遍的解决方案，其基本思想是，在更新磁盘的数据结构之前，在位于磁盘上某个已知的位置记录当前将要执行的操作，即日志[25]。当更新操作中发生 crash 时，可以根据日志记录重新执行之前的更新操作，而不必扫描整个磁盘（FSCK 方法），大大减少恢复期间所需的工作量，减少恢复时间。

在 Linux 文件系统中，磁盘被分成多个群组，每个群组由 Inode 位图和数据位图以及 Inodes 块和数据块组成。



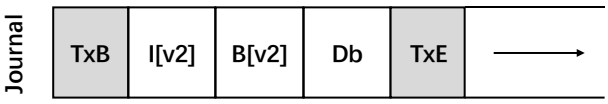
(a) 没有日志的文件系统

(b) 带有日志的文件系统

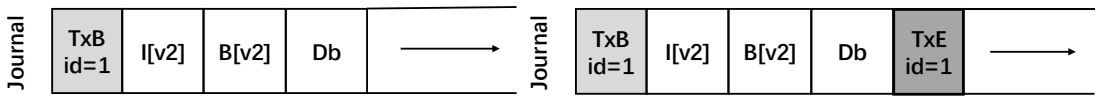
图 2-1 文件系统结构

如图 2-1 所示，带有日志的文件系统，在磁盘存储区多了日志结构，它占用了分区内或其他设备上的少量空间。根据 Journal 结构所记录的内容，可以分为数据 Journaling 和元数据 Journaling 两类。

数据 Journaling：



(a) 数据Journaling的Journal结构



(b) 数据Journaling步骤一　　(c) 数据Journaling步骤二

图 2-2 数据 Journaling 的 Journal 结构及其步骤

数据 Journaling 的 Journal 结构由五个部分构成，如图 2-2（a）所示。事务开始（TxB）标志这个更新操作的开始，包括关于文件系统准备更新的信息（例如，块 I [v2]、B [v2]和 Db 的最终地址）以及某种的事务标识符（TID）。中间的三个块只包含块本身的确切内容。最后的块（TxE）是事务结束的标志，并且还包含 TID。更新文件系统分为以下三个步骤：

第一步，将事务的内容（包括 TxB，元数据和数据）写入日志，完成之后的状态如图 2-2（b）所示。

第二步，文件系统将 TxE 块写入，从而使日志处于最终的安全状态，Journal 结构状态如图 2-2（c）所示。这个步骤的一个重要方面是磁盘提供的原子性保证，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。当第二个步骤完成后，事务被标记为已提交。

第三步，将更新内容（元数据和数据）写入最终的磁盘位置。

第四步，释放事务。

在这一系列更新过程中，任何时候都可能发生 crash。如果在事务被安全地写入日志之前发生 crash（即，在上面的步骤 2 完成之前），则未发生的更新被简单地跳过。如果在事务提交到日志之后，但在检查点完成之前发生 crash，则文件系统可按如下方式恢复更新。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；这些事务因此被依次重新执行，文件系统再次尝试将事务中的块写到其最终的磁盘位置。

优点：
相对于 FSCK，数据 Journaling 恢复速度很快，只需要扫描日志并重新执行一些事务，而不是扫描整个磁盘。

缺点：
文件系统对于每次写入磁盘，首先要写入日志，从而使磁盘写入的数据量翻倍。并且在写入日志和写入主文件系统之间，存在耗时的寻址操作，显著增加了工作负载的开销。

元数据 Journaling：

图 2-3 元数据 Journaling 的 Journal 结构

如图 2-3 所示，与数据 Journaling 相比，数据块 Db 不写入 Journal 中，从而避免了额外的写操作，大大减少了日志的 I/O 负载。但如果文件系统尝试恢复，由于 Db 不在日志中，因此文件系统只重新写入 I [v2]和 B [v2]（而没有重新写 Db），所以 I [v2]将指向垃圾数据。为了避免这种情况，需要先将数据块写入磁盘，再进行日志记录。具体来说，步骤如下：
1、数据写入。将数据写入最终位置; 等待写入完成。
2、日志元数据写入。将开始块和元数据写入日志; 等待写入完成。
3、日志提交。将事务提交块（包含 TxE）写入日志; 等待写入完成; 事务（包括数据）现在已经提交。
4、Checkpoint 元数据。将元数据更新的内容写入文件系统中的最终位置。
5、释放事务。

优点：
相对于数据 Journaling 系统，大大减少了对磁盘的读写数据量。

缺点：
对于原始数据的写入中间需要插入许多的日志记录操作，而日志记录的储存位置和数据的储存位置往往不一样，需要大量磁道寻址操作，大大地降低了文件系统的吞吐量和性能。

采用的文件系统：
NTFS 文件系统、XFS 文件系统

2.1.3 Soft Updates

描述：

为了保持文件系统元数据更新的次序，Soft Updates 采用延迟写（caching）技术，并维持文件系统中元数据的依赖关系[4][6][7]。通过跟踪缓存在内存里的数据结构之间的依赖关系，控制磁盘更新的顺序，从而保证磁盘上元数据更新的有序性。由于大多数元数据所在的块包含了许多指针，基于块的依赖关系会导致频繁出现循环依赖，因此 Soft Updates 中的依赖关系的记录和维持是以指针为单位的[8]。

Soft Updates 容易在执行元数据更新操作时引入循环依赖，例如：当创建文件 A，再删除文件 B 时，数据间的依赖关系如图 2-4（a-c）所示，此时便产生了循环依赖。解决循环依赖的方法是先 roll-back，再 roll-forward，如图 2-4（d）所示。

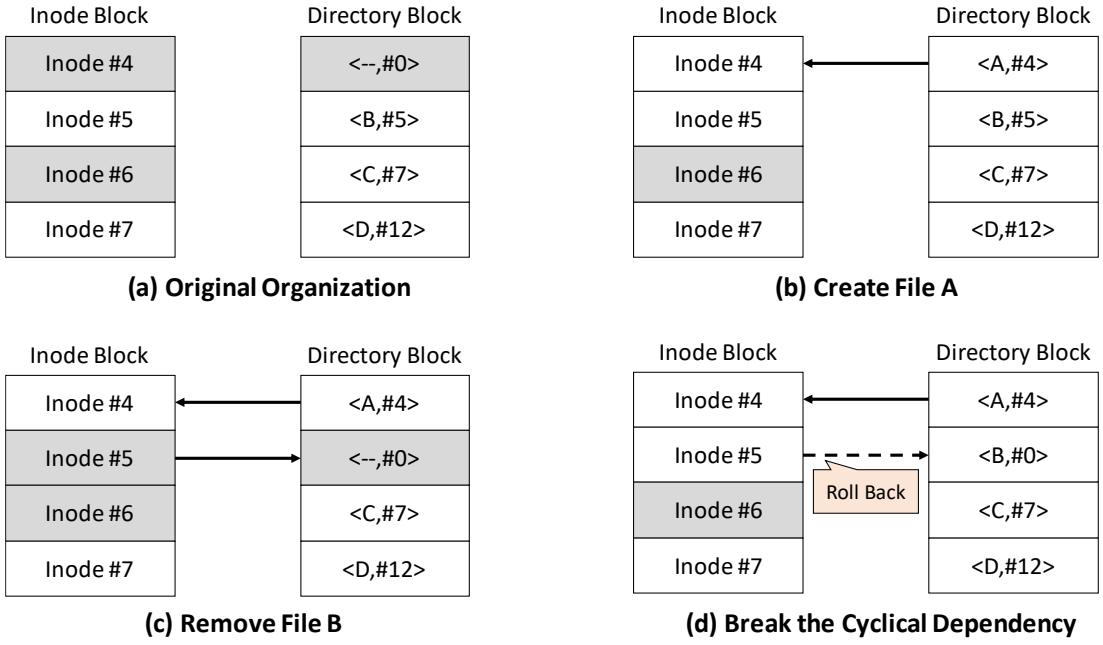


图 2-4 Soft Updates 依赖关系图

优点：

1、没有写日志的开销；

2、不会造成链式的更新，性能开销低；

3、没有大量的同步写操作。

缺点：

1、正确的控制磁盘更新的顺序难度大；

2、虽然保证文件系统一致性，但是不能保证不发生数据丢失；

3、可能产生比较多的循环依赖。

采用的文件系统：

UFS 文件系统、FFS 文件系统

2.1.4 Copy-on-write

描述：

Copy-on-write（简称 COW），每次更新操作都复制原有数据并把更新写到新的位置，而不会覆盖旧数据，因而能够保证版本一致[5][9]。由于文件系统中的数据是采用树状结构进行索引的，对其中一个数据块的 COW，会导致更新操作逐层向上扩散直到树根。因此目录层次越深，更新的开销也就越大。

为了解决这个问题，支持 COW 的文件系统通常采用 B+树作为磁盘索引结构，所有的数据都存放在叶子节点中，当修改一个叶子节点时，需要拷贝从根结点到该节点之间的整条路径，这样更新操作的开销只取决于树的深度，即文件系统中实际存储数据的多少。如图 2-5 所示，当更新 B+树中的节点 A 和 B 时，COW 复制了节点 A 和 B 到根节点的整个路径，并将节点 A 和 B 的数据进行修改，同时保留原始路径上的节点数据。

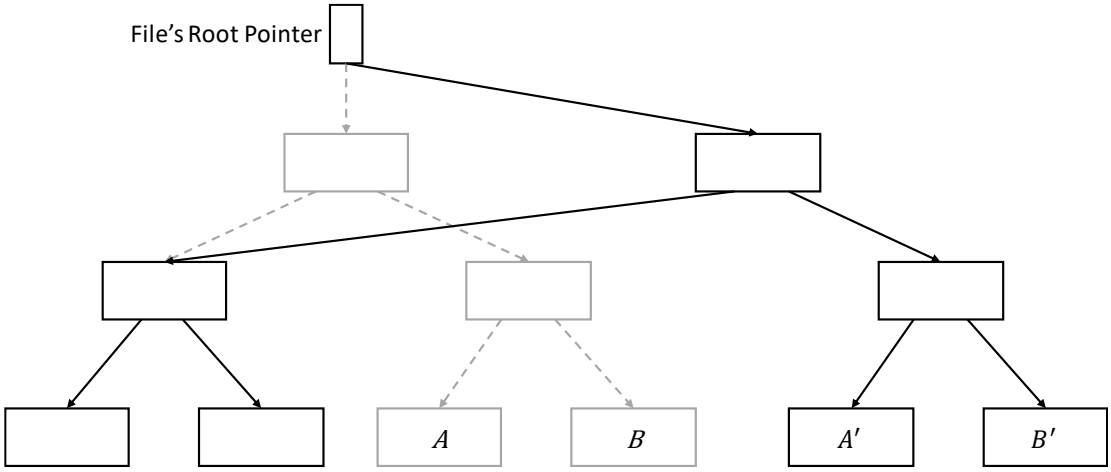


图 2-5 Copy-on-write 更新示意图

优点：

1、不会覆盖旧数据，因而能够保证版本一致；

2、没有额外的写日志开销；

3、crash 重启之后不需要做 FSCK，文件系统保持在旧的一致性状态。

缺点：

1、所有的写操作都需要一步步上溯到文件系统的根节点；

2、每一次写操作都产生比较多的复制成本。

采用的文件系统：

ZFS 文件系统、Btrfs 文件系统

2.1.5 方法对比

本小节，我们将对上述四种保障文件系统一致性的方法进行对比，对比的指标包括了以下四个方面，结果如表 2-1 所示。

- 数据完整性：文件系统发生 crash 之后，数据可恢复的能力
- 数据持久性：文件系统进行系统调用时，数据持久化到非易失性介质的能力
- 数据原子性：元数据操作不可拆分的能力
- 恢复时间：文件系统 crash 之后恢复到一致性所需要的时间

表 2-1 四种文件系统一致性方法对比

技术	完整性	持久性		原子性	crash 恢复时间
FSCK	×	×		×	花费数小时或者更多
Journaling	√	**Sync**	√	√	花费数秒或数分钟
		Async	×		
Soft Updates	√	×		×	立即恢复
Copy-on-write	√	×		√	立即恢复

从表 2-1 中可以看出，FSCK 由于是在系统 crash 之后进行系统级的扫描和检测从而保障一致性，因此无法提供数据完整性、持久性和原子性的保障，crash 之后恢复的时间与磁盘空间大小呈正相关关系。Journaling 是通过额外的日志来保证 crash 之后系统恢复到一致性状态，因此可以保障数据的完整性和原子性，其中同步 Journaling 技术由于同步操作到磁盘空间，因而可以提供数据持久性保障，而异步 Journaling 则不可以；crash 之后的恢复由于只需要检查日志文件而不是整个磁盘空间（FSCK），因此恢复的时间较短，仅需数秒或数分钟。Soft Updates 和 Copy-on-write 方法都可以提供数据完整性保障，但是都无法保证持久性，两者分别通过维持依赖关系和额外复制修改的策略，使得 crash 后系统依旧处在一致性状态，因此可以立即恢复；不同点在于，Copy-on-write 保持了修改前的数据版本和更改的路径，因此可以提供数据原子性，而 Soft Updates 则不可以。

2.2 应用程序数据一致性保障

目前，已经有很多技术被用于保证文件系统元数据的一致性，应用程序数据也存在一致性问题，本小节将围绕如何保障应用程序数据的一致性进行阐述。

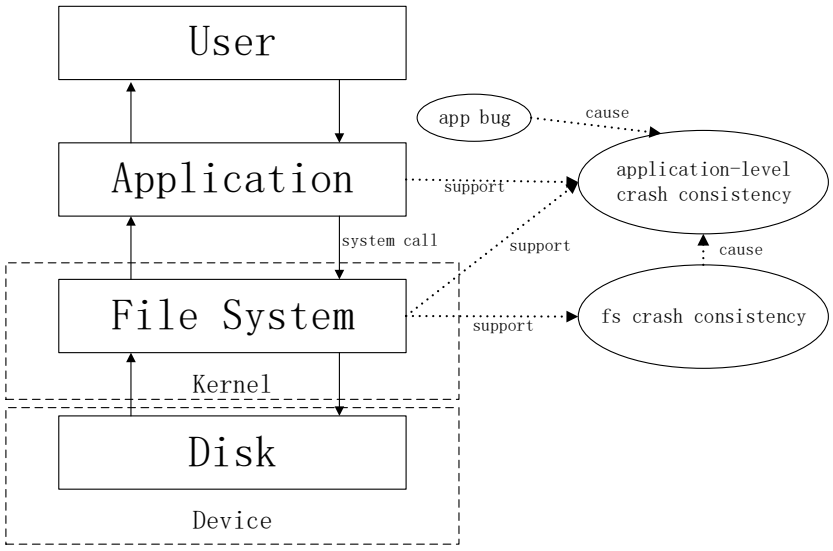


图 2-6 文件系统交互关系

如图 2-6 文件系统交互所示，应用程序数据一致性问题在文件系统一致性之上，除了能造成文件系统一致性的因素会导致应用程序数据一致性问题，应用程序自身的 bug 可能会导致应用程序 crash，从而引发应用程序数据一致性问题。

2.2.1 应用程序支持

为了保证自身数据的一致性，程序需要实现更新协议和恢复协议：修改的时候，根据更新协议，以一种可恢复的方式修改文件数据和目录；crash 之后，根据恢复协议，将不一致的数据恢复到一致状态。以 DBMS 为例，DBMS 使用 journaling 的更新协议，先将修改数据写入单独的日志文件，再更新数据库文件，crash 之后，DBMS 在重启后执行恢复协议，如果发现数据库文件部分被修改，根据日志文件进行完整的更新。

此外，良好的程序设计、测试、高容错机制[11]、观察点注入[12]可以减少程序由于程序自身原因导致的 crash 概率。

2.2.1 文件系统支持

应用程序的更新协议和恢复协议可能会依赖于文件系统的原子性和有序性，原子性包括单个系统调用原子性和跨系统调用原子性，应用程序运行在具备这些持久化属性的文件系统上才能保证其数据的一致性。然而做到这些并不容易，有些磁盘提供 512 字节的原子写操作，有些存储技术只提供 8 字节的原子写操作。有序性会大大降低文件系统的性能，很多文件系统会对操作进行重排序以减少磁盘的 seek 次数和更多地获得 grouping benefit[13][14][15][16]，这样的重排序经常影响程序数据的正确性[17][18]。

文件系统安全性的增加经常导致性能的下降，也有文件系统做到了高性能地提高应用层的一致性：CCFS[19]通过抽象出流的概念保证文件操作的顺序性，再通过其他技术保证高性能。为了检查特定文件系统提供的持久化属性是否满足应用程序要求，TS Pillai 提出 BOB 和 ALICE[17], BOB 工具发现文件系统的持久化属性，ALICE 测试应用程序在一个文件系统上运行时存在的一致性漏洞。

第三章 一致性保障与漏洞分析实现

3.1 CCFS

由于 Journaling 在保持文件系统一致性方面的高可靠性和高性能，因而受到国内外研究学者的追捧。然而，Journaling 也因额外的写操作带来数据量翻倍的问题，增加了工作开销，为此，研究者们继续完善 Journaling，提出了许多新的改进方案。Crash-Consistent File System(CCFS)是 2017 年提出来的保持应用程序 crash 一致性的文件系统。CCFS 指出在不同应用程序间保持写依赖次序不仅难以保证 crash 一致性，而且会带来较高的性能负担，导致伪依赖（false dependence）。

为了避免出现伪依赖，CCFS 为不同的应用程序开辟不同的流，每个流对应着一个应用程序而非文件或目录，因此一个文件或目录可能同时存在不同的流中被不同应用程序使用。每一个流中有一个运行事务，一次提交只影响一个流中的事务，不影响其他流中的事务。

为了解决两个流同时对同一个块进行更新，采用了混合粒度日志的模式，运行事务是以字节为粒度记录，而事务的提交和检查是以块为粒度的。如图 3-1 所示，块 X 的初始值为$< a_0 , b_0 >$，应用程序 A 把 X 中的a_0修改成a_1，接着应用程序 B 把 X 中的b_0修改成b_1，可以发现不同流对应的运行事务中的更新操作都是以字节为粒度的。当应用程序 B 执行提交时，CCFS 以块为粒度提交，将流 B 运行事务中的 X 新块的值$< a_0 , b_1 >$提交，将b_0修改成b_1而不改变a_0的值。

Steps	Actions	In-memory value of X	In-memory running transactions	Committed transactions	Committed value of X
#0	Initial	$< a_0 , b_0 >$	T_A [ ] T_B [ ]		$< a_0 , b_0 >$
#1	APP A: $a_0 \rightarrow a_1$	$< a_1 , b_0 >$	T_A [X_{0-0}] T_B [ ]		$< a_0 , b_0 >$
#2	APP B: $b_0 \rightarrow b_1$	$< a_1 , b_1 >$	T_A [X_{0-0}] T_B [X_{1-1}]		$< a_0 , b_0 >$
#3	APP B: fsync()	$< a_1 , b_1 >$	T_A [X_{0-0}] T_B [ ]	T_B [$X =< a_0 , b_1 >$]	$< a_0 , b_1 >$
#4	APP A: fsync()	$< a_1 , b_1 >$	T_A [ ] T_B [ ]	T_A [$X =< a_1 , b_1 >$]	$< a_1 , b_1 >$

图 3-1 混合粒度日志的模式示意图

当两个流同时对同一个块中的同一个数据（共享的 metadata）进行更新时，CCFS 采用 Delta 日志模式，在不同流的运行事务中以字节为粒度增量更新，而提交则是与混合粒度日志模式一样以块为单位。

CCFS 利用 selective data journaling (SDJ)技术[10]来维持流内数据更新的顺序，只记录重写的文件数据（包括数据和元数据）和指向文件追加的块指针。为了解决 ext4 中不能进行延迟分配所带来的性能降低，CCFS 采用了一种顺序保持的延迟分配方法（order-preserving delayed allocation），每一个流中的所有分配都被延迟到流中的某个事务提交前才执行，通过这样既保持了流内顺序，又提高了文件系统的性能。

作者将 CCFS 与 ext4 文件系统进行对比，结果表明 CCFS 在保护应用程序一致性方面具有更加优越的性能，同时可以有效地消除伪依赖问题，性能方面测试结果表明，CCFS 也取得可接受的测试性能。

3.2 iJournaling

标准的 ext4 文件系统将并发不相关的事务分组到单个复合事务[1]里，并周期性地刷新到保留的 Journal 存储区域中。但为了确保即时数据的持久性，用户必须调用一个同步操作，如 fsync()或 fdatasync()。同步操作必须等到 Journal commit 操作完成。Daejun Park 等人发现以下四个原因可能造成同步操作的延迟。

1、事务间的依赖性（IT）。在 ext4 文件系统中，使用单个 JBD2 线程，因此一次只能提交一个事务。而且因为它们共享几个元数据块，多个事务不能并发地提交，。

2、复合事务依赖（CTX）。当 JBD2 线程提交 fsynced 文件的事务时，提交事务的 Inode 列表包含不相关的 Inode。由于 ordered-mode journaling 的排序约束，JBD2 线程必须等待数据块写入操作的完成。当有许多进程生成文件系统写入操作时，复合事务依赖性非常严重。

3、ext4 对块的延迟分配技术加剧了复合事务依赖问题。延迟块分配直到页面刷新时间，而不是在 write()操作[2]期间。启用延迟分配时，文件系统的整体性能会更高。但是，如果紧接在 flush 内核线程调用之后调用 fsync，则 flush 线程将为脏页分配数据块，并在延迟期间在正在运行的事务中注册几个已修改的 Inode 块分配。然后，日志事务的提交操作将生成很多写入请求到存储中。如果在调用 flush 线程之前调用 fsync，因为对文件系统的修改很少，所以则 fsync 延迟将会很短。因此，fsync 延迟将在延迟分配方案中波动。

4、准异步请求依赖关系[3]。由 fsync 系统调用生成的写入请求与 SYNC 标志一起发送，而由刷新线程生成的写入请求不带标志发送。Linux 中的 CFQ I/O 调度程序给没有 SYNC 标志的请求提供较低的优先级。尽管由刷新线程生成写入的数据块是 ASYNC 的，但其对延迟敏感。这样的请求被称为准异步请求。在准异步请求完成时，特别是在 I/O 队列中存在很多竞争异步请求的情况下，会出现很长的延迟。

iJournaling 是 Daejun Park 等人于 2017 年在 USENIX 上提出的新型混合日志系统。iJournaling 的目标是提高 fsync()调用的性能，同时利用传统的基于复合事务的日记方案的优势。

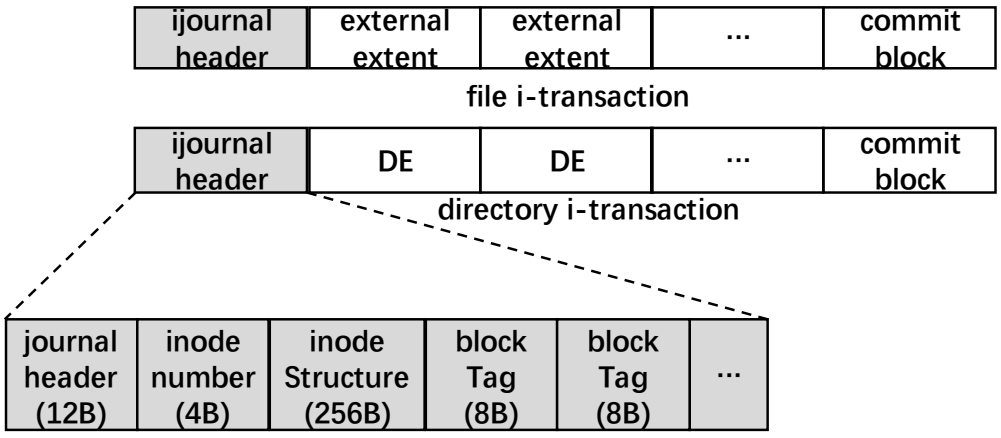


图 3-2 iJournaling 结构图

i-transaction 包含两种类型，file i-transaction 具有 fsynced 文件的元数据信息，而 directory i-transaction 具有任何相关父目录的元数据信息。itransaction 中的 Inode 号和 Inode 结构用于恢复 Inode 表，Inode 位图和 GDT。

iJournaling 新型混合日志系统有以下特点：

1、iJournaling 技术在系统调用服务而不是 Journal 线程处理 fsync 调用，并使用分离的 Journal 区域，解决这个事务间依赖性的问题。

2、iJournaling 提交文件级事务而不是复合事务，通过消除不相关的依赖来减轻准异步请求依赖性。

3、记录 iJournaling 日志的区域与和记录标准 Journal 的区域是分开的，并且每个处理器内核都使用独立的 percore iJournaling 区域，以支持许多可扩展性。

它只为 fsync 调用记录相应的文件级事务，并定期记录期间记录正常的日志事务。文件级事务日志只有相关的元数据更新 fsynced 文件。通过消除对 fsync 延迟有害的几个因素，减少 fsync 延迟，减轻 fsync 密集线程之间的干扰，并具有多核扩展能力。

3.3 ScaleFS

ScaleFS[20]是 S S Bhat 在 2017 年发表的论文，这是一个混合文件系统，同时具有多核可扩展和高吞吐量的特性。除了达到多核扩展这个目标，同时 ScaleFS 还需要满足标准文件系统的 2 个要求：crash safety 和 good disk throughput。同时满足以上目标是非常困难的，比如，在同一个目录下创建两个不同的文件 f1 和 f2，根据 Salable Commutativity Rule[21]，create(f1)与 create(f2)是可交换操作，因此存在一种支持多核扩展的实现，然而 linux 系统中，这两个操作会修改同一个磁盘 block，造成 cache 冲突，从而限制了可扩展性。如今，很多任务都是基于内存操作，而不 flush 到磁盘中去，多核扩展性显得尤其重要。

ScaleFS 主要目标是多核扩展，它通过基于 oplog[22]的操作日志解耦了内存文件系统和磁盘文件系统， 内存文件系统支持多核扩展，磁盘文件系统用于持久化数据。

如图 3-3 ScaleFS 架构图所示，ScaleFS 由内存文件系统 MemFS 层与磁盘文件系统 DiskFS 组成，中间使用 Oplog 层进行解耦，由磁盘文件系统将数据持久化到磁盘上以保证 Crash Safety。接下来，对 MemFS 层、Oplog 层、DiskFS 层进行解释：

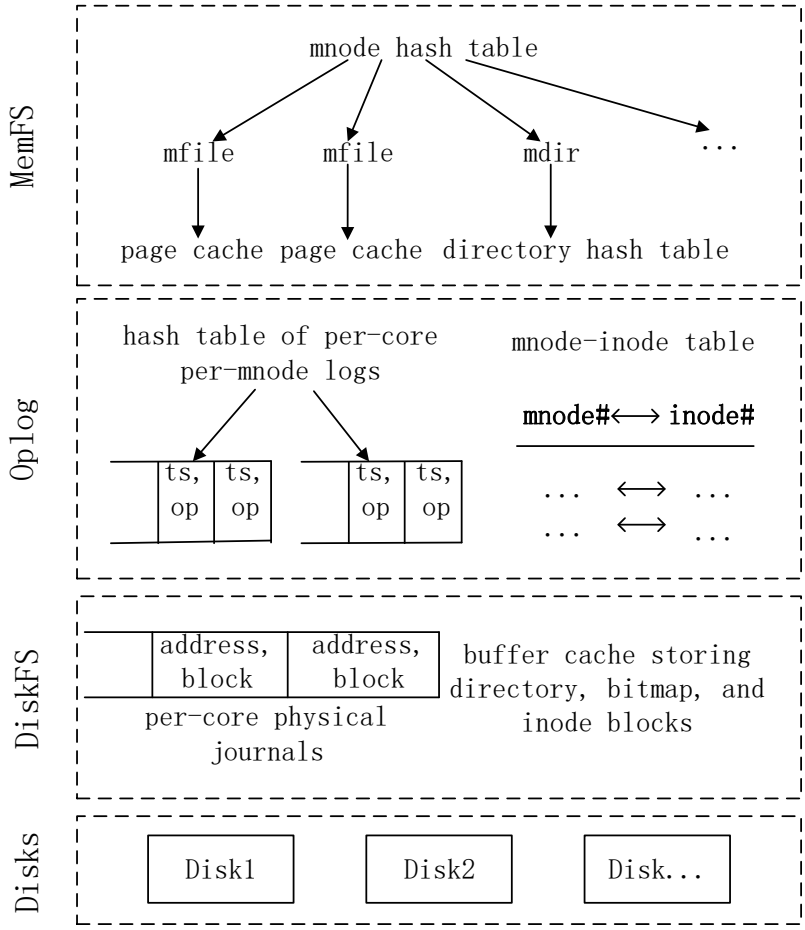


图 3-3 ScaleFS 架构图

MemFS 层：此处的 MemFS 层与 DiskFS 具有不同的功能，因此 MemFS 并不存储 inode 数据，而采用一种类似结构 mnode 进行代替，并维护一个由 mnode 编号进行索引的 hash 表。Mnode 包含两种类型：mfile 指向一个包含了文件数据块的 page cache，该 cache 使用 radix array[3]实现，并保存了脏位；mdir 维护一个将文件名映射为 mnode 编号的并发 hash 表。

Oplog 层：Oplog 层包含每个 mnode 的 per-core 修改日志。这些日志包含了目录的逻辑修改和一个用于确定顺序的时间戳。Oplog 解耦了 MemFS 层和 DiskFS 层，包含 mnode 到 Inode 的互相映射关系，当 MemFS 调用 flush 时，Oplog 层将修改日志传到 DiskFS 上。

DiskFS 层：DiskFS 层实现一个传统的文件系统，使用 per-core physical journal 自动将数据持久化到磁盘上。DiskFS 也维护一个 buffer cache，存储磁盘元数据，不存储文件数据。

具体设计上，现代处理器提供同步时间戳用于保证操作日志的有序性，seqlock 保证所有的读在写之前，等待正在执行的操作完成后在进行日志的合并，当针对一个文件触发 fsync 时，合并一个 mnode 的 per-core 日志集合中的日志。Flush 一个操作日志时，ScaleFS 首先进行 absorption 减少持久化的数据量,保证跨文件 rename 不会丢失数据，保证文件系统状态的内部一致性，不允许孤立的目录、目录循环、指向没有初始化文件的链接的这样的情况存在。

实现上，MemFS 使用 sv6[21]的内存文件系统，DiskFS 使用 xv6 文件系统[23]。ScaleFS 在 COMMUTER[21]上的测试表明，99%的测试用例没有发生 cache 冲突，在 80 核的机器上表现出不错的可扩展性，同时具有 ext4 相当的磁盘性能。

3.4 ALICE

ALICE[17]是 TS Pillai 在 2014 年发表在 OSDI 上的一篇论文中用于分析应用程序更新协议以及发现潜在 crash 漏洞的工具。很多重要的应用程序的实现依赖于文件系统，然而为了保证这些数据管理程序的 crash consistency 并不容易，需要面临 2 个困难：

1、文件系统提供的保证模糊不清。每个文件系统持久化应用数据的方式有所不同，只好留给开发人员去猜测。

2、构建一个既高性能又 crash-consistency 的应用层协议并不容易。复杂的实现让应用程序存在诸多 bug，而且测试难以覆盖到一些 corner case。

为了解决以上问题，作者分别开发了 Block Order Breaker(BOB)工具和 Application-Level Intelligent Crash Explorer(ALICE)框架。

对于不同的文件系统在，在 crash 之后可能存在不同的状态，这样的不同特性成为持久化属性。BOB 用于分析文件系统的持久化属性，找出文件系统支持的持久化属性。实现上，BOB 首先给文件系统添加工作负载，收集并重排序由工作负载产生的 block I/O，选择性地将它们写入磁盘，产生不同的磁盘镜像，恢复时候对每个镜像进行检查是否满足持久化属性。将系统调用的操作分为三类：file overwrite, file append 和 directory operations。用单个操作原子性和操作间有序性来体现一个文件系统的持久化属性，最后发现不同文件系统或者不同配置的相同文件系统具有不同的持久化属性。

ALICE 用于检测应用程序的更新协议在一个特定文件系统上是否存在 crash 漏洞。ALICE 首先通过对系统调用的跟踪构建文件系统 crash 可能产生磁盘文件状态，再针对每个状态检测应用程序的数据一致性。

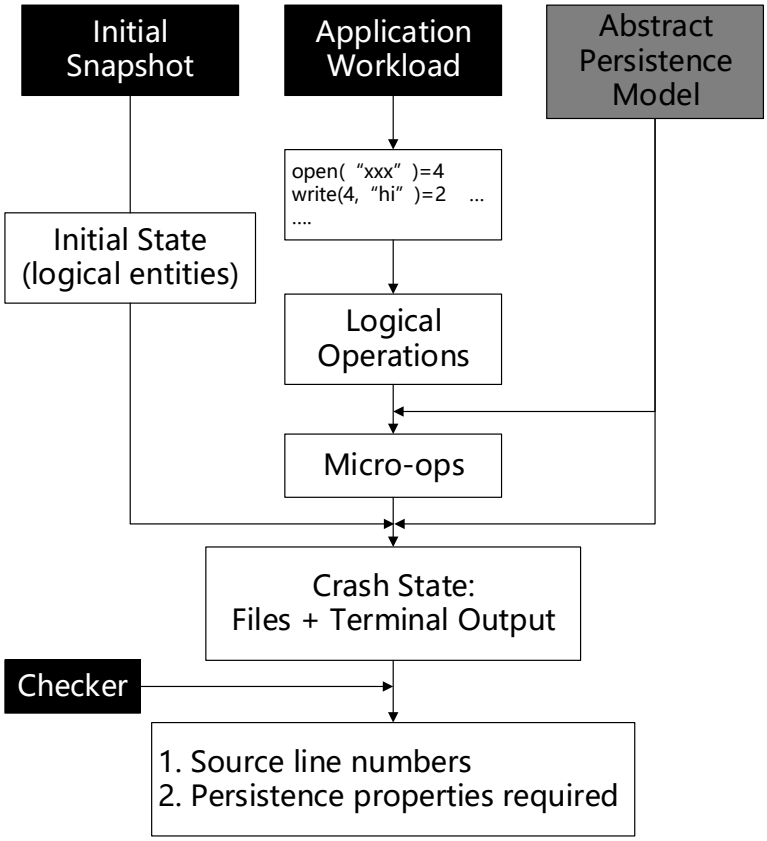


图 3-4 ALICE 框架图

ALICE 的框架图如图 3-4 所示。运行之前，需要有一个应用程序初始文件快照、应用程序工作负载运行脚本和一个用于检查工作负载的 invariant 是否得到维护的检查器脚本。APM 定义了一个特定文件系统的可能的 crash 状态，在根据由逻辑操作序列经过细化所生成的微操作序列，并结合初始文件状态，可输出最终的 crash 文件状态。 ALICE 在不同的 crash 状态上运行检查器脚本，最后产生一个在该负载下应用程序更新协议的逻辑表示、协议中的漏洞、涉及的代码行、一致性所要求的持久化属性。ALICE 评估了 7 个广泛应用工的应用系统，发现了 60 个一致性漏洞，其中很多都会导致严重的后果。

评估中 ALICE 有着优异的表现，但是具有局限性：

1、ALICE 还不够完整，存在一些无法被检测到的漏洞。

2、需要用户提供应用程序的工作负载运行脚本和检查器。

3、应用程序负载多线程操作文件系统，而 ALICE 对操作已经序列化了，难以反映实际情况。

4、ALICE 当前不支持文件属性的操作。

第四章 未来文件系统

从现有的论文对文件系统的改进，发现文件系统仍然存在很多可以改进的地方，并结合未来的计算机技术发展趋势，我们将在本章对文件系统的未来发展进行展望。文件系统其实已经较为成熟，但仍然可以发现诸多不足，未来对现有技术和算法将会进行进一步修改和改进，得到性能上提升。这样的变化不会太大，主要发展在于，未来将产生不同于现在的场景，现有技术无法满足应用场景，需要新的技术，接下来要介绍的多核文件系统、分布式文件系统、嵌入式文件系统就是在这样的情况下产生。同时，硬件技术的突破对文件系统要求会产生至关重要的影响。

4.1 多核文件系统

随着摩尔定律的失效，芯片制程遇到天花板，CPU 多核架构成为未来发展趋势，良好的多核扩展性将大大提高本地文件系统的性能，同时到来的 cache 冲突也会对文件系统性能和一致性提出新的挑战。ScaleFS 在前人对内存文件系统的多核扩展上的研究基础上，解耦了内存文件系统和磁盘文件系统，为磁盘文件系统添上了多核扩展的特性。多核扩展的核心在于减少同步操作，存在很多方法用于提高多核处理能力，希望这些方法不断被运用到文件系统中去。

4.2 分布式文件系统

大数据时代的到来，一个文件系统需要承载前所未有的数据量，单机文件系统难以承受庞大的存储量和访问量，分布式文件系统应运而生，该类系统通常部署在廉价的商用计算机上，多个机器共同组成一个具有高度容错性的分布式系统，提供高吞吐量的数据访问服务。为了避免由于某个节点故障导致的数据丢失，会在多个节点存储数据，因而与传统的文件系统相比，保持文件系统一致性方面有一个重要的问题，即如何保持元数据副本一致性。

目前已经有很多针对不同应用场景的分布式文件系统产生应得到应用：GFS 是对称式 SAN 文件系统，没有服务器（但有锁服务器），采用与日志文件系统类似的元数据日志技术来恢复元数据一致性。HDFS 是 Hadoop 实现的一个分布式文件系统，它的假设硬件错误是常态而不是异常，提供流式数据访问，面向大规模数据集，具有不允许修改的简单一致性模型；于此不同的，MogileFS、MooseFS、FastFS、TFS、GridFS 这些分布式文件系统适合海量的小文件存储与搜索。

4.3 嵌入式文件系统

随着物联网技术和移动互联网技术的发展，嵌入式设备走进千家万户，嵌入设备与通用计算机同样具有计算能力，但嵌入式设备小巧、功能单一、低功耗。机械硬盘由于其内部需要驱动电机和磁头等结构，功耗较大，是嵌入式设备无法承受的，因此嵌入式设备往往使用选择更高效的 Flash 存储介质存储信息。以 Android 手机为例，Android 系统从 Linux 系统裁剪得到，处理器多用 ARM 处理器, 文件系统默认是受 Linux 内核支持的 Flash 文件系统

YAFFS，不同的硬件特性(比如，原子性的粒度)会直接影响一致性实现方式。未来是物联网的时代，将出现更多方便人们生活的嵌入式设备，对存储速度、容量、功耗会提出新的要求，例如有的设备可能对一致性要求并不高，只要求快速读写，这时文件系统设计可以放弃一些一致性策略，集中精力提高性能。

4.4 硬件技术

随着存储技术的提高，超大容量磁盘不断涌现，未来文件系统需要在保证性能和一致性的前提下提供对超大容量磁盘的支持。大量优秀存储器涌现，如 SSD 的流行，不同存储介质的设备具有不同特性，文件系统需要充分考虑不同存储介质的特性进行定制以发挥出最大的性能。受 NVMM 启发，从本质上讲，文件系统一致性问题是由于 DRAM 内存的易失性造成，如果有一天大容量存储设备的速度达到 DRAM 速度，持久化数据不再需要缓存，便不存在一致性问题。

第五章 讨论与总结

本文针对文件系统一致性的问题，从保障文件系统元数据的一致性和应用程序一致性两个角度，探讨了主流的一致性相关技术研究。接着，针对具体的应用场景，列举了三个文件系统用于保障一致性和一个一致性漏洞检测工具，这些系统都是针对文件系统不同的应用要求所做的改进，在功能和性能两方面都取得了令人满意的效果。最后，针对文件系统未来的发展趋势，从多核文件系统、分布式文件系统以及硬件技术革新等方面，探讨和展望未来文件系统在保障一致性方面的发展方向。

在保障文件系统元数据的一致性领域，FSCK 与其他基于软件的一致性技术不同，是对 crash 之后的文件系统进行全盘扫描和修正来确保系统一致性，这种方式在 crash 之后的恢复阶段需要大量的时间，也牺牲了较大的性能。为了避免 crash 后才进行一致性保证，同时尽可能在保障数据的完整性和提高系统性能之间权衡，研究人员提出许多优秀的一致性技术，如 Journaling、Soft Updates、Copy-on-write 等。这些技术所采用的方法各有不同，对文件系统元数据的保护和对系统性能的影响也不一样。Journaling 采用 write-ahead logging，不仅可以保障元数据的一致性，也可以保障文件数据的一致性，因此受到广泛学者的追捧。Soft Updates 和 Copy-on-write 分别是通过维持依赖关系和额外复制修改的策略，在维持数据的一致性的同时，使得 crash 后系统依旧处在一致性状态。

针对 Journaling 中同步操作带来的延迟，iJournaling 利用传统的基于复合事务的日志策略对其进行改善，以类似增量记录的方式记录最少的元数据日志，提高了 fsync 调用的性能，同时为每一个核维护一个 iJournaling area，提供了多核拓展性。而 ScaleFS 文件系统也关注于多核场景下的可扩展和高吞吐量，基于 oplog 的操作日志解耦了内存文件系统和磁盘文件系统，使内存文件系统支持多核扩展，磁盘文件系统用于持久化数据。

除了维持文件系统元数据一致性外，我们还探讨了应用程序的一致性问题。从应用程序和文件系统两个方面对该问题进行阐述：应用程序需要尽量保持自身正确性，实现良好的更新协议和恢复协议让自身具备恢复一致性的能力；文件系统提供应用需要的持久化属性可帮助应用程序数据保证一致性。

不同应用程序间保持写依赖次序容易出现伪依赖，为此 CCFS 提出为不同的应用程序开辟不同的流实现隔离，同时采用了混合粒度的日志模式，运行事务是以字节为粒度而提交是以块为粒度进行更新。与 CCFS 保障应用程序一致性不同，ALICE 从检测程序的更新协议是否存在潜在的 crash 漏洞为角度切入，利用由 BOB 发现的文件系统持久化属性、初始文件状态、程序运行负载生成可能的 crash 状态，从而得出一致性漏洞。

最后，本文针对文件系统的发展对一致性的考验做了分析和展望，主要围绕多核文件系统、分布式文件系统、嵌入式文件系统以及新型硬件存储设备四个方面，对保持文件系统的一致性提出了更高的要求。

参考文献

[1] Prabhakaran V, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Analysis and Evolution of Journaling File Systems[C]//USENIX Annual Technical Conference, General Track. 2005, 194: 196-215.

[2] Mathur A, Cao M, Bhattacharya S, et al. The new ext4 filesystem: Current status and future plans[C]// Linux Symposium. 2007.

[3] Jeong D, Lee Y, Kim J S. Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices[J]. 2015.

[4] Ganger G R, Patt Y N. Metadata update performance in file systems[C]// 1994:5.

[5] Bonwick J, Moore B. ZFS: The last word in file systems[J]. 2007.

[6] Mckusick M K, Mckusick M K, Soules C A N, et al. Soft updates: a solution to the metadata update problem in file systems[J]. Acm Transactions on Computer Systems, 2000, 18(2):127-153.

[7] Seltzer M I, Ganger G R, Mckusick M K, et al. Journaling versus soft updates: Asynchronous meta-data protection in file systems[C]// AGU Fall Meeting. 1999:71--84.

[8] McKusick M K, Ganger G R. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem[C]//USENIX Annual Technical Conference, FREENIX Track. 1999: 1-17.

[9] Hitz D, Malcolm M, Lau J, et al. Copy on write file system consistency and block usage: U.S. Patent 6,892,211[P]. 2005-5-10.

[10] Chidambaram V, Pillai T S, Arpaci-Dusseau A C, et al. Optimistic crash consistency[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 228-243.

[11] Huang Y, Kintala C M R. Software Implemented Fault Tolerance Technologies and Experience[C]// International Symposium on Fault-Tolerant Computing, Toulouse, France, June. DBLP, 1993:2-9.

[12] Pattabiraman K, Kalbarczyk Z, Iyer R K. Application-based metrics for strategic placement of detectors[C]// Pacific Rim International Symposium on Dependable Computing. IEEE Computer Society, 2005:75-82.

[13] Jacobson D M, Wilkes J. Disk scheduling algorithms based on rotational position[M]. Palo Alto, CA: Hewlett-Packard Laboratories, 1991.

[14] Kim J, Oh Y, Kim E, et al. Disk schedulers for solid state drivers[C]//Proceedings of the seventh ACM international conference on Embedded software. ACM, 2009: 295-304.

[15] Ruemmler C, Wilkes J. An introduction to disk drive modeling[J]. Computer, 1994, 27(3): 17-28.

[16] Seltzer M, Chen P, Ousterhout J. Disk scheduling revisited[C]//Proceedings of the winter 1990 USENIX technical conference. 1990: 313-323.

[17] Pillai T S, Chidambaram V, Alagappan R, et al. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications[C]//OSDI. 2014: 433-448.

[18] Yang J, Sar C, Engler D. Explode: a lightweight, general system for finding serious storage system errors[C]//Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006: 131-146.

[19] Pillai T S, Alagappan R, Lu L, et al. Application Crash Consistency and Performance with CCFS[C]//FAST. 2017: 181-196.

[20] Bhat S S, Eqbal R, Clements A T, et al. Scaling a file system to many cores using an operation log[C]// The, Symposium. 2017:69-86.\

[21] Clements A T, Kaashoek M F, Zeldovich N, et al. The scalable commutativity rule: Designing scalable software for multicore processors[J]. ACM Transactions on Computer Systems (TOCS), 2015, 32(4): 10.

[22] Clements A T, Kaashoek M F, Zeldovich N. RadixVM: Scalable address spaces for multithreaded applications[C]//Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013: 211-224.

[23] Cox R, Kaashoek F, Morris R. a simple, Unix-like teaching operating system[J].

[24] McKusick M K, Joy W N, Leffler S J, et al. Fsck− The UNIX† File System Check Program[J]. Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version, 1986.

[25] Hagmann R. Reimplementing the Cedar file system using logging and group commit[M]. ACM, 1987.