文件系统 C/R 工具研究综述

M202173701 陈端阳

摘　要　文件系统检查与修复工具（简称 C/R 工具）常用于当系统内核或安全升级，以及发生故障之后，用于检查和修复文件系统中的错误。现代存储设备提供高带宽和低延迟，以及更快的扫描和检查速度。传统的 C/R 工具未对现代存储设备的新特性进行优化，本文重点叙述以往的 C/R 工具缺陷，以及最新的研究 pFSCK 如何改进他们的缺陷。

关键词　文件系统；C/R 工具

A review of research on file system C/R tools

duanyang chen

Abstract The File System Check and Repair Tool (C/R Tool) is commonly used to check for and repair errors in the file system when the system kernel or security is upgraded, and after a failure. Modern storage devices offer high bandwidth and low latency, as well as faster scanning and inspection speeds. Traditional C/R tools are not optimized for the new features of modern storage devices, and this article focuses on past C/R tool defects and how the latest research pFSCK can improve their flaws.

Key words file system;C/R tool

1　引言

与传统硬盘相比，SSDs 和 NVMe 等现代超快存储设备不仅提供高带宽，而且还可减少两个数量级的存储访问延迟。另一方面，英特尔的 DC Optane 等快速存储类存储器和其他字节可寻址持久存储技术正在不断发展。近年来，一些新的文件系统已经发展到可以利用这些硬件优势。大量先前和正在进行的研究正在开发优化的文件系统，以支持快速存储硬件。但是，要减少这些文件系统的数据损坏和错误，工业界还需要好几年的时间。虽然文件系统 C/R 工具将在这些文件系统中发挥关键作用，但它们尚未得到优化，无法提取硬件存储优势和多核并行性。然而，大条带的修复成本问题会被进一步放大。因此，缓解纠删码系统中的修复瓶颈是一个有实际意义的研究课题。而设计有效的修复算法也是主要的研究方向之一。

自文件系统问世以来，一致性一直是一个问题。尽管已开发日志、写入时复制、日志结构写入和软更新等存储机制来缓解潜在的文件系统不一致，但它们受到限制，因为它们无法修复由软件错误或过去由故障磁盘、位翻转、过热或相关崩溃等事件引起的损坏引起的错误。在这些情况，C/R 工具（如 e2fsck 和 xfs_repair）通过遍历文件系统的结构并检查索引节点一致性、文件和目录连接性、目录条目一致性以及 索引节点和块的一致引用数来检测和修复损坏和错误。

广泛使用的开源工具，如 Ext4 文件系统的 e2fsck 跨多个磁盘并行化 C/R（e2fsck），XFS 文件系统的 xfs_repair 跨磁盘和卷逻辑组并行化 C/R。其他方法，如 Ffsck 和 Chungfs，已经提出了通过修改文件系统来加速 C / R 速度，以提供更好的跨逻辑组平衡。虽然以前的方法具有先进的 C / R 创新，但它们存在一些弱点。首先，大多数先前的技术都无法利用多核并行性和高存储带宽。其次，先前的并行化工作大多

是粗粒度的(e2fsck,xfs_repair,Chunkfs)。最后，SQCK 和 Ffsck 等技术需要对文件系统元数据、块放置或重建 C/R 的需求进行大量更改，从而阻碍了广泛采用。

2 原理和优势

pFSCK 原理：e2fsck 对 C/R 使用五个顺序传递：第一个传递（称为 Pass-1）检查 inode 元数据的一致性; Pass-2 检查目录一致性;Pass-3 检查目录连接;Pass-4 检查参考计数;Pass-5 检查数据和元数据位图的一致性。大部分(＞95%)的时间用于 Inode Pass 和 Directories Pass。目录密集型所花时间比文件密集型长很多。

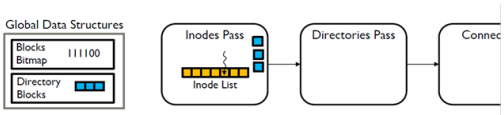

fig 1

如 fig1 所示，e2fsck 工作流程为，串行执行当前 Pass 中的 List，更新 Global Data Structures 中下一个 Pass 的 List，当前 Pass 结束后从 Global Data Structures 中得到下一个 Pass 的工作 List。

其中 Inode Pass 中所作的工作如 fig2 所示

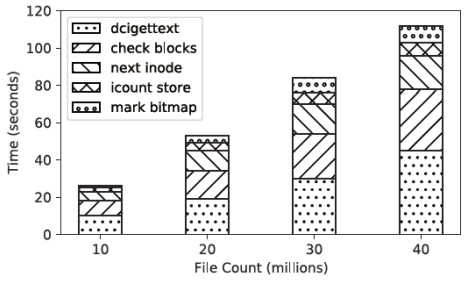

fig 2

用于错误处理的函数 dcigettext(一种看似无害的）语言翻译器错误地用于每个 inode 检查，导致明显的 C/R 速度变慢。其他步骤，如检查 inode 引用的块的 check_blocks，从磁盘读取下一个 inode 块的 next_inode,更新全局位图以跟踪遇到的元数据的 mark_bitmap，以及存储 inode 引用的 icount store 也会增加运行时间。

xfs_repair 原理：xfs_repair 修复损坏的 XFS 文件系统。指定文件系统的设备参数包含文件系统的磁盘分区或卷的设备名称。如果给定了块设备的名称，则 xfs_repair 将尝试查找与指定块设备关联的原始设备，并将使用原始设备代替。xfs_repair 使用前提是文件系统是卸载状态否则会造成文件系统的不一致或损坏。

一致性检查包括如下： inode 索引节点和 inode 索引节点块检查：索引节点的错误魔术数字，索引节点块的错误魔术数字，扩展区乱序，索引节点块中错误的记录数，文件系统中声明的不在合法数据区域块，由多个 inode 索引节点声明的块。inode 索引节点分配的映射检查：索引节点映射块的错误魔鬼数字，映射指示的索引节点状态和节点本身的状态不一致，文件系统的索引节点并没有出现在索引节点分配的映射中，索引节点分配的映射中未包含的索引节点。文件大小检查：声明的块大小和索引节点大小不一致，目录大小未按块对齐的，索引节点大小和索引节点格式不一致。目录检查：目录块的错误数字，目录块中的错误个数的条目，目录叶块中错误的 free 空间信息，指向未分配的或超出范围的 inode 的条目，重叠的条目,缺失或不正确的 dot 和 dotdot 条目，哈希值顺序不符的条目，错误的内部目录指针，目录类型与 inode 格式和大小不一致。路径名检查：从文件系统根目录开始的路径名未引用的文件或目录，非法的路径名组件。链接计数检查：链接计数与索引节点的目录引用数量不一致。超级块检查：总空闲块和空闲索引节点计数错误，文件系统几何图不一致，次要和主要的超级块相互矛盾。通过将孤立的文件和目录（已分配，使用中但未引用）放置在 lost + found 目录中，以重新连接它们。分配的名称为 inode 编号。XFS 有主要和次要两个超级块。xfs_respair 处理修复前，用主要超级块里的信息自动地将主要超级块和次要超级块做对比。如果主要超级块损坏太严重而无法用于定位次要超级块，则程序将扫描文件系统，直到找到并验证一些次要超级块为止。 此时，它会生成一个主要超级块。xfs_repair 在进行时会发

出提示性消息，提示发现的异常或已采取的任何纠正措施。

e2fsck 的粒度为 disks/partitions，而 xfs_repair 的粒度为 groups，两者均未对 CPU 和硬盘的并行性进行优化。

3 研究进展

ChunkFS： chunkfs 的核心思想是将文件系统拆分为磁盘上许多小的故障隔离域— 块，大小约为几千兆字节（参见 fig3。每个块都有自己的块编号空间、分配位图、超级块和其他传统的每文件系统元数据。有关块的位置和内容的少量元数据存储在摘要区域的块外部。文件系统命名空间和可用磁盘空间仍然是共享的，因此对用户和管理员来说，它仍然感觉像一个文件系统。

将文件系统拆分为块很容易。困难在于再次将其粘合在一起以保留共享命名空间和磁盘空间，同时保持块彼此之间的故障隔离。基本规则是：（1） 所有跨区块引用都具有显式的前进和后退指针，以及 （2） 将跨区块引用保持在最低限度。本节介绍作者如何使用这些原则来解决实现 chunkfs 的主要困难。

Chunkfs 通过将文件系统划分为块、小型故障隔离域（几乎可以完全独立于其他块进行检查和修复）来提高文件系统的可靠性、缩短修复时间并提高数据可用性。这允许快速、增量和部分在线的文件系统检查和修复。此外，chunkfs 体系结构使许多其他有用的文件系统功能变得可行，例如在线调整大小、在线碎片整理和每块磁盘格式。

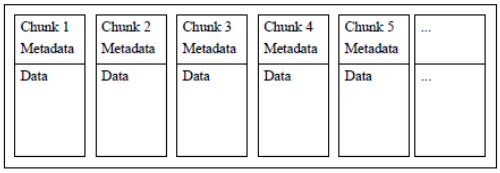

fig 3

Chunkfs 的优点为：减少 fsck 时间;减少 fsck 对内存的需求;强故障边界;在线部分 fsck;每个块都可以实现为具有不同的 inode 与数据的比率，不同的块大小或布局上的其他差异，并且可以在首次分配时初始化为特定布局。

Chunkfs 的缺点为：Chunkfs 只有在跨区块引用可以保持罕见时才有效。块可用空间的大量碎片导致许多连续 inode 将浪费空间并增加文件系统检查和修复时间。作者希望使用一种常见的策略来控制文件系统碎片：保留一定比例的可用空间，以避免最坏情况下的碎片和性能下降，但代价是相对便宜和充足的磁盘空间。另一个缓解因素是，块结构不仅允许在线文件系统检查，而且将大大简化在线碎片整理。 虽然大多数一致性检查可以单独在每个块上完成，但仍然必须检查块间的一致性。 例如，如果作者检查区块 A 并遵循延续 inode 的后指针指向区块 B 中的原始 inode，然后检查区块 B 并发现原始 inode 是孤立的，那么作者将不得不释放区块 A 中的延续 inode 和关联的块。 UNIX 文件系统的分层结构使真正独立的块故障的目标复杂化，因为块 A 中文件路径名中的组件可能位于块 B 中。如果块 B 已损坏，则文件 A 将断开连接且不可用。这可以通过目录重复、在 inode 中缓存父名称提示以及通过独立挂载块或以其他方式访问块中所有文件的机制来缓解。

SQCK： SQCK 是一个基于声明性查询语言的文件系统检查器； 声明性查询与必须跨文件系统映像的许多结构执行的交叉检查自然匹配。作者表明，SQCK 能够通过令人惊讶的优雅和紧凑的查询执行与 e2fsck 相同的功能。SQCK 可以通过组合整个文件系统中可用的信息，轻松执行比 e2fsck 更有用的修复。最后， SQCK 原型实现了这种改进的功能，其性能与 e2fsck 相当。

SQCK 包含五个主要组件，如 table 1 所示。扫描程序从磁盘读取文件系统的相关部分，而加载程序则将相应的信息加载到数据库表中。 然后，检查器负责运行声明性查询，这些查询检查和修复文件系统结构。刷新器通过将更改写出到磁盘来完成循环。

Tables	Fields
Superblock Table	*blkNum*, *copyNum*, *dirty*, firstBlk, lastBlk, blockSize, ...
GroupDesc Table	*blkNum*, *gdNum*, *copyNum*, *dirty*, *start*, *end*, blkBitmap, inoBitmap, iTable, ...
Inode Table	*ino*, *blkNum*, *used*, *dirty*, mode, linksCount, blocksCount, size, ...
DirEntry Table	*blkNum*, *entryNum*, *dirty*, ino, entryIno, recLen, nameLen, name
Extent Table	start, end, *pBlk*, *pByte*, *type*, *startLogical*, *endLogical*, *ino*, *dirty*, ...

Table 1

作者利用声明性语言改进的 e2fsck 的修复方式，声明性查询可以简洁地表达 fsck 执行的许多不同类型的检查和修复。作者的经验还表明，在声明性查询中编写检查和修复相对简单;每个查询都是通过一些迭代优化编写的。复杂的检查或修复，在 C 代码的帮助下，可以分解成几个易于理解的简短查询。平均而言，作者编写的每个查询有 7 行长，最长的查询有 22 行。此外，只有 24 次维修需要 C 代码的帮助。相应 C 代码的功能通常很简单;C 代码仅用于运行一组查询和迭代查询结果。

pFSCK：

作者在 e2fsck 的基础上改进，设计了一种细粒度的 C/R 工具。改进方法如下：

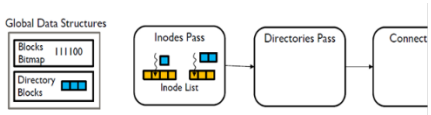

fig 4

Data Parallelism：如 fig4 所示，将每个 Pass 中的工作 List 分成更小的组；使用多线程并行检查，并产生中间 lists；当前 Pass 结束后从 Global Data Structures 中得到下一个 Pass 的工作 List。在当前的 C/R（如 e2fsck 和 xfs_repair）中，作者发现全局数据结构在传递内部和跨传递中大量使用。为了减少共享并提高并发性，作者引入了类似于操作系统线程上下文的每线程上下文。这些上下文存储允许线程并行运行的信息。首先，文件系统对象的每线程块缓存、缓冲区（堆分配）和迭代器允许并行文件系统遍历。其次，perthread 中间数据结构和计数器用于并行收集高级文件系统状态。例如，每个 inode 传递 （Pass-1） 线程都有自己的 db_list（目录阻止列表）和 dir_info 列表（目录信息列表），用于收集有关目录的信息并将其导出到目录传递 （Pass-2） 线程进行处理。最后，perthread 计数器用于并行跟踪文件类型统计信息。但是，作者观察到，由于频繁地跨传递访问全局位图（几乎对于每个操作），pFSCK 被迫通过锁定使用同步。虽然将这些位图分解为每线程结构是可行的，但它需要对 e2fsck 框架进行重大更改。 表 1 显示了共享结构及其在 e2fsck 中的角色。

线程托管可提高局部性：为了增加局部性并更好地利用处理器缓存状态，pFSCK 在每次传递中，都会尝试将同一传递中的线程共定位到相同的内核和内存套接字，以避免在处理器缓存之间反弹锁定变量和共享结构。为了启用线程托管，pFSCK 会维护每个线程已使用的 CPU 编号以及特定通道使用的内核列表。pFSCK 首先尝试将线程放置到以前使用的核心（如果可用），如果不可用，则使用同一阶段中使用的其他可用内核。

使用每线程预取器和缓存减少 I/O 等待时间：虽然当前的 C/R（如 e2fsck 缓存和预取文件系统块）不灵活，并且缺乏线程感知。 首先，e2fsck 使用一个小型的、静态的、完全关联的基于 LRU 的缓存（有 8 个块），并且只预取 inode 块。E2fsck 不会预取目录数据块，与 inode 块不同，这些块可能是不连续的。其次，由于线程以不同的偏移量访问块，因此跨线程共享缓存会导致冲突的逐出，从而增加 I/O 开销。

为了克服这些限制，作者设计并实现了每线程缓存机制，以避免跨线程错误地逐出缓存条目。为了避免目录块的非连续性问题，作者实现了自适应预取机制（类似于 Linux 文件系统预取）如果由于缺乏顺序访问而未使用先前预取的目录块，请减少预取窗口。

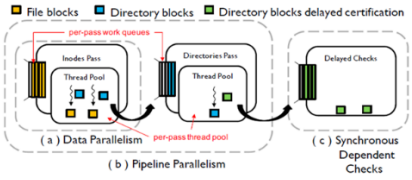

fig 5

Pipeline Parallelism：首先，为了便于并发执行传递，作者使用每个传递线程池。

如 fig5 所示，inode 和目录检查阶段维护一个单独的线程池和一个专用的工作队列，其中填充了需要检查的文件系统对象。当每个工序运行时，生成的任何中间工作都将放置在下一个工序的工作队列中。例如，当遇到表示目录的 inode 时，inode 检查阶段会将其块排队到目录检查阶段的工作队列中，以启用并发 C/R。

允许多个通道使用管道并行运行需要重新排序逻辑检查的正确性。 例如，使用 pFSCK 的管道并行性，目录数据块可以由目录检查通道 （Pass-2） 与 inode 检查通道 （Pass-1） 并行检查目录中的所有 inode（文件和子目录）。虽然这两个传递可以并行进行，但只有在 inode 检查传递验证其子目录和文件的一致性之后，才能将目录标记为一致。通过 C/R 认证目录有两个主要约束：（1） 此目录的 dirent 引用的所有 inode 都是有效的，以及 （2） 此目录引用的父目录有效。为了这的挑战，pFSCK 会延迟某些检查，直到之前的管道通过完成。

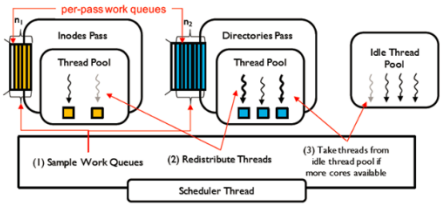

fig 6

Dynamic Thread Scheduler：如 fig6 所示，作者提出了动态线程规划，做法是定时采集各个 Pass工作队列长度，根据公式计算出每个 Pass 的线程池中线程数量。

$$W_{total} = \sum_{i=0}^{n} q_i n_i w_i$$

$$t_i = C \frac{q_i n_i w_i}{W_{total}}$$

其中q_i代表 Pass i 的工作队列长度，n_i代表 Pass i 中每个块的大小，w_i为权重，C 为闲置线程与本程序当前已用线程之和。利用这一计算方法，可以达到与手动选择最优线程分配同样的效果（见结果评估部分）。

System Resource-Aware Scheduling：首先，作者将讨论 C/R 与其他应用程序一起运行，但在单独的未装载磁盘上执行 C/R 的情况。 为了减少 C/R 开销对其他应用程序的影响，pFSCK-rsched 维护了一个调度程序线程。最初，pFSCK-rsched worker 被安排为具有 SCHED_IDLE 优先级，该优先级主要在任何空闲 CPU 上调度进程。 当调度程序定期运行时，pFSCK-rsched 首先确定 CPU 核心预算，通过确定整个系统中处于活动使用状态的 CPU 数、可用空闲内核数以及 pFSCK-rsched 使用的内核数，来确定在任何时间点可以使用的最大线程数。根据 pFSCK-rsched 使用的有效内核数，如果 pFSCK 使用量少于可用的空闲内核，则 pFSCK-rsched 会增加 pFSCK 的核心预算；如果 pFSCK 使用的内核数超过空闲内核，则 pFSCK 的核心预算会减少 pFSCK 的核心预算，从而减少与其他应用程序的争用。确定核心预算后，调度程序使用每个阶段的工作队列确定各个阶段之间的工作比率，并在每个阶段之间重新分配理想数量的线程。向通道添加线程时，将从空闲线程池中获取线程并将其分配给每个通道线程池。如果由于核心预算减少而需要删除线程，则会向线程发出信号并将其重新分配给空闲线程池。

当检测到当前块是要修改的时 pFSCK 会重定向到快照中的原始块并对原始块进行检验。

由于重新关注支持在线 C / R，C / R 工具（如 e2fsck）最初用于离线使用，可以在 Linux 的逻辑卷管理器（LVM）的帮助下在线使用。LVM 的快照功能通过采用写入时复制方法来捕获文件系统状态，以保留已修改块的原始版本。这使得 C/R 工具能够以主动的方式使用，扫描预先存在的错误，而无需关闭系统。 在使用 LVM 的在线 C/R 中，首先初始化一个空的快照卷。如果其他应用程序修改了任何块，LVM 会先将原始块复制到快照，然后再更新原始卷上就位的块。当 C/R 读取发现已修改的块时，读取将重定向到保存原始块的快照。未修改块的读取将重定向到原始卷。虽然快照初始化成本低廉，但应用程序会产生同步数据复制和 I/O

重定向的额外开销，从而减少了 C/R 的可用存储带宽。

pFSCK 的缺点:目前 pFSCK 只适配 Linux ext 文件系统；pFSCK 对普通硬盘不能进行优化；pFSCK 在高错误率下的性能与 e2fsck 表现相同。

4 总结与展望

现代存储设备发展速度很快，但是存储设备中的错误是难以避免的，所以 C/R 工具的性能也需要随着现代存储设备的发展而发展，Ext4 文件系统的 e2fsck 跨多个磁盘并行化 C/R （e2fsck），XFS 文件系统的 xfs_repair 跨磁盘和卷逻辑组并行化 C/R。其他方法，如 Ffsck 和 Chungfs，已经提出了通过修改文件系统来加速 C / R 速度，以提供更好的跨逻辑组平衡。SQCK 和 Ffsck 等技术需要对文件系统元数据、块放置或重建 C/R 的需求进行大量更改，从而阻碍了广泛采用。最新的 pFSCK 在高错误率的表现不理想，仍需要未来进一步改进。

参 考 文 献

[1] David Domingo,Sudarsun Kannan. pFSCK: Accelerating File System Checking and Repair for Modern Storage[C]// 19th {USENIX}Conference on File and Storage Technologies ({FAST} 21). 2021:113-126

[2] Marshall Kirk McKusick,Willian N Joy, Samuel J Leffler, and Robert S Fabry. Fsck- the unix† file system check program. Unix System Manager’s Manual-4.3 BSD Virtual VAX-11 Version, 1986.

[3] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In HotDep, 2006.

[4] Haryadi S Gunawi, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Sqck: A declarative file system checker.