

**WFSCK开发设计文档**

**项目成员：**

|  |  |  |
| --- | --- | --- |
| **姓名** | **年级** | **联系方式（QQ）** |
| 刘航（队长） | 研一 | 512962645 |
| 刘强 | 研一 | 1614545312 |
| 孙志航 | 研一 | 965868276 |

**校内指导教师：**夏文、李诗逸、仇洁婷

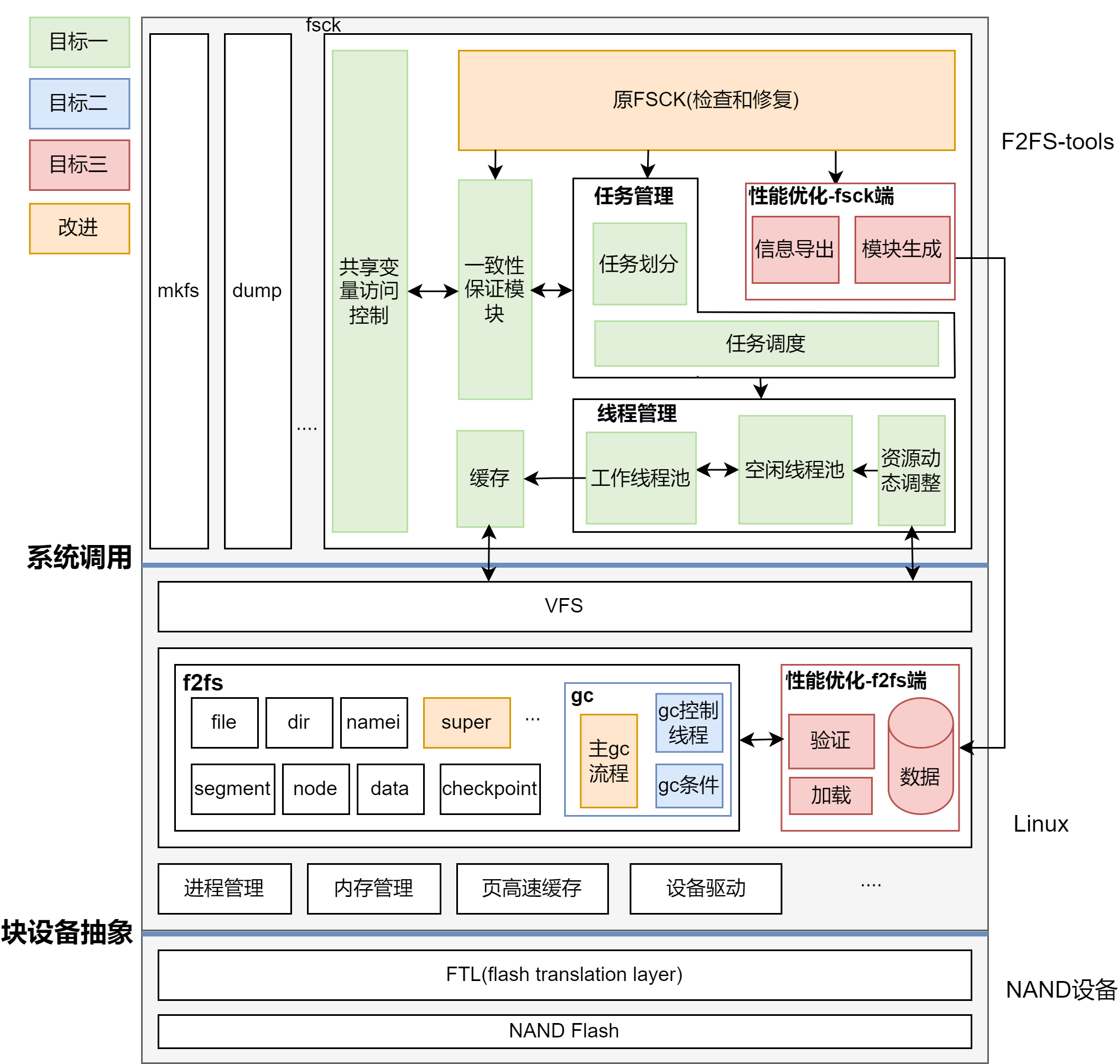
**校外指导教师：**郑立铭（华为）

2023 年 08 月

**摘 要**

wfsck是一个智能的文件系统检查恢复和任务管理框架。主要包含三个目标，**第一个目标**是加速对文件系统的检查恢复。以fsck.f2fs作为原型，并参考pfsck引入多线程并发机制，加速检查和修复的过程，而不影响C/R（检查和恢复）的正确性。并且，动态调整线程个数，以减少对其他程序的影响。**第二个目标**是完成对f2fs前台gc任务的优化，减少无效的gc，使gc更智能，从而提升性能。**第三个目标**是在fsck.f2fs检查过程中收集信息，优化后续挂载等情况。fsck.f2fs在检查文件系统的过程中，会遍历所有的文件元数据信息，这信息可以被记录下来，用于优化后续对文件系统的挂载等情况，使f2fs运行的更智能。

本项目的架构图如图所示。



本项目以第三届OS竞赛为驱动，旨在完成下面三个目标：

* **目标 1：**加速fsck.f2fs;

难点1：如何将pFSCK的并发思想移植到fsck.f2fs中。pFSCK的检查逻辑与fsck.f2fs是不一样的。pFSCK是分阶段地进行检查，先检查inode，再检查目录等等。但fsck.f2fs的检查是直接从根Node递归地对整个Node树进行检查。

难点2：如何划分任务。理解fsck.f2fs原本的检查逻辑，从中划分出不同的任务。这个任务划分不能是简单的将每个Node的检查作为一个任务，因为每个Node需要对每个子Node检查的返回值进行处理，这样划分会影响检查的正确性。

难点3：如何正确地对每个任务的返回值进行处理。每个任务执行完成后需要对其返回值进行处理，在并发环境下，如何保证处理逻辑的正确是一个难点。

难点4：如何高效安全访问共享数据结构。并发情况下，如果只是简单加一个粗粒度的锁，对共享数据结构进行串行化访问，会成为系统的瓶颈。如果只是细化锁的粒度，每个变量对应一个锁，也会比较低效。需要对不同的共享数据结构进行区分，重新设计数据结构。例如有的变量是只写的，可以添加到线程私有数据，为线程设计一个类似于线程上下文的结构存放私有数据。有的变量，又读又写，加细粒度的锁处理。有的变量如目录项虽然又读又写，但是只与某个Node为根的Node树检查有关，需要添加到线程私有数据中，才能保证正确性。

难点5：如何更智能地动态调整线程个数，使得检查工具能感知资源使用情况，做出调整，减少对其他程序的影响。

难点6：如何充分利用硬盘I/O。当前系统的I/O缓存不是为并发设计的，不同线程访问缓存可能导致低效的驱逐情况，需要为每个线程设计一个I/O缓存。

* **目标 2：**优化f2fs的gc;

难点1：理解f2fs的gc过程。包括前台gc和后台gc在触发时机、gc的代价、gc效果等方面的差异，gc时如何挑选要回收的segment，gc时如何迁移数据等。

难点2：理解f2fs的写数据过程。包括写数据时何时会触发gc，gc触发时为何会有许多有效块特别多的脏segment。

难点3：分析无效gc产生的原因。特别是为何会频繁产生待回收segment有效块特别多的情况。

难点4：如何通过有效的手段减少无效gc。

* **目标 3：**利用fsck收集的信息，优化后续挂载等情况;

难点1：内核不能直接读取文件，如何将fsck.f2fs收集到的信息传递给内核。

难点2：fsck收集到的信息保存在一个复杂的结构体当中，如何将信息导出 供内核使用。

难点3：如何判定fsck收集到的信息中哪些是有价值，从而指导对f2fs文件 系统的优化。

目前，我们的赛题完成度如下：

|  |  |  |
| --- | --- | --- |
| **目标** | **基本完成情况** | **额外说明** |
| 加速fsck.f2fs | 完成（100%） | ① 可节省25%到50%左右的运行时间；  ② 可动态调整线程个数 |
| 优化f2fs的gc | 完成（100%） | ① 减少两种无效的gc。  ② 最多将写性能从50个文件/秒提升到900个文件/秒； |
| 利用fsck收集的信息 | 完成（100%） | ① 可将fsck.f2fs中的超级块信息传递给f2fs挂载时使用。 |

其中，决赛完成的内容如下：

|  |  |  |
| --- | --- | --- |
| 优化f2fs的gc | 完成（100%） | ① 减少两种无效的gc。  ② 最多将写性能从50个文件/秒提升到900个文件/秒； |
| 利用fsck收集的信息 | 完成（100%） | ① 可将fsck.f2fs中的超级块信息传递给f2fs挂载时使用。 |

**目 录**

[1. 概述 6](#_Toc142994860)

[1.1 项目背景及意义 6](#_Toc142994861)

[1.1.1 加速fsck.f2fs 6](#_Toc142994862)

[1.1.2 优化f2fs的gc 8](#_Toc142994863)

[1.1.3 利用fsck收集的信息 9](#_Toc142994864)

[1.2 国内外研究概况 10](#_Toc142994865)

[1.2.1 加速fsck.f2fs 10](#_Toc142994866)

[1.2.2 优化f2fs的gc 23](#_Toc142994867)

[1.2.3 利用fsck收集的信息 25](#_Toc142994868)

[1.3 项目的主要工作 26](#_Toc142994869)

[2. 需求分析 27](#_Toc142994870)

[3. 系统设计 28](#_Toc142994871)

[3.1系统整体架构设计 28](#_Toc142994872)

[3.1.1 架构概述 28](#_Toc142994873)

[3.1.2 系统整体运行流程 30](#_Toc142994874)

[3.2 子模块设计 34](#_Toc142994875)

[3.2.1 加速fsck.f2fs 34](#_Toc142994876)

[3.2.2 优化f2fs的gc 38](#_Toc142994877)

[3.2.3 利用fsck收集的信息 41](#_Toc142994878)

[4. 系统实现 43](#_Toc142994879)

[4.1 加速fsck.f2fs 43](#_Toc142994880)

[4.1.1核心数据结构 43](#_Toc142994881)

[4.1.2关键函数实现 49](#_Toc142994882)

[4.2 优化f2fs的gc 57](#_Toc142994883)

[4.2.1 核心数据结构 57](#_Toc142994884)

[4.2.2 关键函数实现 58](#_Toc142994885)

[4.3 利用fsck收集的信息 62](#_Toc142994886)

[4.3.1 核心数据结构 62](#_Toc142994887)

[4.3.2 关键函数实现 63](#_Toc142994888)

[5. 系统测试 67](#_Toc142994889)

[5.1 加速fsck.f2fs 67](#_Toc142994890)

[5.1.1 测试准备 67](#_Toc142994891)

[5.1.2 测试方法与测试结果 68](#_Toc142994892)

[5.2 优化f2fs的gc 75](#_Toc142994893)

[5.2.1 测试准备 75](#_Toc142994894)

[5.2.2测试方法与测试结果 75](#_Toc142994895)

[5.3 利用fsck收集的信息 78](#_Toc142994896)

[5.3.1 测试准备 78](#_Toc142994897)

[5.3.2 测试结果 79](#_Toc142994898)

[6. 总结与展望 80](#_Toc142994899)

[6.1 工作总结 80](#_Toc142994900)

[6.2 创新点 81](#_Toc142994901)

[6.3 未来展望 81](#_Toc142994902)

[参考文献 82](#_Toc142994903)

# 概述

## 项目背景及意义

本小节将分别介绍三个目标加速fsck.f2fs、优化f2fs的gc、利用fsck收集的信息的背景和意义。

### 1.1.1 加速fsck.f2fs

wfsck是一个智能的文件系统检查恢复和后台任务管理框架。它将pfsck的思想移植到了fsck.f2fs上。而pfsck是基于e2fsck引入了并发机制。接下来将依次介绍前面提到的几个工具，并介绍加速文件系统检查恢复的意义。

**文件系统检查恢复的几种工具**

e2fsck是检查和修复ext2、ext3、ext4等文件系统的工具。而fsck.f2fs是检查和修复f2fs这个flash文件系统的工具。pfsck则是在e2fsck的基础上引入了并发机制，加速其执行过程。并且支持额外特性。包括动态调整线程个数，减少对其他程序影响，重新设计缓存，充分利用存储设备I/O。本项目的wfsck将在fsck.f2fs的基础上引入并发机制和相关特性，来达到加速和智能的目的。其中前面提到的四种工具特性如表1-1所示。

表 1‑1 各工具特性

|  |  |  |  |
| --- | --- | --- | --- |
| 工具 | 原型 | 适用的文件系统 | 支持并发 |
| e2fsck | 无 | ext系列 | 否 |
| pfsck | e2fsck | ext系列 | 是 |
| fsck.f2fs | 无 | f2fs | 否 |
| wfsck | fsck.f2fs | f2fs | 是 |

**加速文件系统检查恢复的意义**

现代超高速存储设备(如ssd、NVMe和可字节寻址的NVM存储技术)提供比硬盘更高的带宽能力和更低的延迟。在I/O访问性能提高的同时，存储硬件和软件错误也在不断增加。长期以来，文件系统检查和修复工具(以下简称C/R)通过识别和纠正文件系统不一致性，在提高软件存储的可靠性和系统可用性方面发挥了关键的作用[1]。

以往的大量研究表明，在数据中心发生系统崩溃或故障的情况下，C/ R通常被用作系统恢复的第一个补救方案。以往的研究[2,3]表明，C/ R可以跨各种场景运行。这包括由于硬件或软件错误[2,3,4]、定期维护或强制安全升级[5]而导致的重启期间的问题。当C/ R以脱机方式在磁盘分区上运行时，该分区的数据不可用。一些C/ R支持在线检查，但至关重要的是，它们不会干扰使用同一设备的其他应用程序。因此，提高C/R性能和灵活性对于系统可用性和减少对其他应用程序的性能影响至关重要。

文件系统C/R工具通过识别和修复文件系统元数据的结构不一致来工作。不一致可能出现在索引节点、数据和位图、链接或目录条目结构中。广泛使用的C/R工具，如e2fsck (Ext4的文件系统检查工具)[6]将C/R划分为多个阶段(通常称为pass)，每个pass负责检查一个文件系统结构(例如，目录、文件、链接)。然而，C/R速度非常慢，随着文件和目录数量的增加，C/R时间呈线性增长[7,8-11]，有时持续数小时[5]，甚至数周[4]。尽管现代flash和NVM技术提供了更低的延迟和带宽，但当前的C/R工具无法利用这些硬件功能或多核CPU并行性。

为了克服这些限制，我们提出了wFSCK，一种并行C/R，它利用CPU并行性和现代存储的高带宽来加速离线和在线形式的文件系统C/R，从而减少系统停机时间，提高数据(和系统)的可靠性和可用性[2,9,12,13]。虽然wFSCK借鉴了先前的任务并行研究[14,15]，但它必须解决C/R特有的几个挑战，包括在复杂文件系统布局和共享文件系统结构(例如通用位图)中提高可扩展性，而不影响正确性，并减少C/R对其他应用程序的影响。wFSCK引入了并行性，将root Node下的子Node的检查封装为任务，交给线程执行，这使得执行速度比传统的C/ R要快得多。pFSCK采用数据并行性，将对Node树的检查工作分解，并允许多个线程并行执行检查。虽然数据并行加速了检查，但每个任务中对全局数据的访问需要同步以确保检查的正确性，简单加锁影响了效率。对此，需要重新设计数据结构，对不同的数据进行区别，有的需要放入线程的私有数据中，有的则需要通过细粒度的锁访问。

当前C/ R中的I/O缓存和预读机制等I/O优化不是为多线程并行设计的，我们通过设计线程感知的I/O缓存来解决这个问题，从而大大减少I/O等待时间。最后，为了在不影响共享CPU或访问C/R检查(在线检查)的相同磁盘的其他协同运行应用程序的性能的情况下利用多核并行性，我们设计了一个资源感知的wFSCK调度器，它通过监视系统的总CPU利用率来动态地扩展C/R线程。

### 1.1.2 优化f2fs的gc

gc是日志型文件系统f2fs的一个重要特性。本小节将依次介绍gc的目标，gc的阶段和优化gc。

**gc的目标**

闪存垃圾回收的目标是解决两个主要问题：写放大和空间整理。写放大是指由于数据被多次写入相同位置而导致的性能下降和闪存寿命缩短的现象。空间整理则涉及到优化闪存中的空闲空间，以确保连续的可用块，从而提高读取和写入操作的效率。

**gc的阶段**

f2fs中的垃圾回收过程可以分为以下几个阶段。数据写入和删除阶段。当数据被写入文件系统时，它们可能被放置在闪存的不同位置，而不一定是连续的块。当文件被删除或更新时，相应的数据块可能会变为无效，但它们不会立即被擦除，因为这样做可能会降低闪存寿命。这些无效的数据块成为垃圾数据。垃圾数据标记阶段。垃圾回收首先会标记那些已经无效的数据块。这通常通过跟踪文件系统的元数据来实现，以了解哪些块包含了有效数据，哪些块已经被删除或更新。数据整理阶段。在数据被标记为垃圾后，垃圾回收过程会启动数据整理。这涉及将有效数据从散乱的块中移动到一个或多个新的块，以创建连续的可用空间。这不仅有助于减少写放大，还有助于提高数据访问速度。写入优化阶段f2fs的垃圾回收还可以优化写入操作，通过将相关数据放置在相邻的块中，减少了寻道和延迟，从而提高了性能。

**优化gc**

在f2fs中，gc（Garbage Collection）是一个关键的优化过程，它用于回收被标记为无效数据的闪存块，以便重新利用这些块。优化f2fs的gc是为了最大程度地减少擦写操作，提高读写性能，延长闪存设备的寿命，并提高空间利用率。通过有效管理闪存块的使用，f2fs可以更好地适应闪存设备的特性，使其在闪存存储介质上发挥更好的性能和可靠性。具体如下：

闪存设备特性：闪存设备有一个擦除操作的限制，称为擦写次数。每个闪存块只能被擦写有限次数，因此频繁的擦写操作会导致块的寿命减少。gc的主要目标是尽量减少擦写操作，从而延长闪存设备的寿命。

无序擦写：在闪存设备上，擦除操作通常是无序的，这意味着在写入新数据之前，必须先将整个块擦除。这导致在更新现有数据时产生了许多无效数据，因为旧数据被标记为无效，而实际上仍然存在于闪存中。

读写性能：频繁的擦除操作会导致性能下降，因为擦除操作比写操作更耗时。通过gc将多个闪存块的无效数据整理到一个块中，并在后续的写入过程中进行更有效的擦除和写入，可以显著提高读写性能。

空间利用率：gc可以回收无效数据所占用的空间，从而提高闪存设备的空间利用率。

其中，在进行读写操作时，若系统空间将要不足，会频繁触发前台gc，影响读写的性能。**本项目主要针对该场景进行优化**。

### 1.1.3 利用fsck收集的信息

fsck.f2fs在修复文件系统时，需要获取整个f2fs文件系统的信息，并根据这些信息，来检查文件系统的一致性。若出现不一致的状况，就对文件系统进行修复；否则，说明文件系统是正确的，直接退出。

**问题分析**

针对上述问题，我们发现，fsck.f2fs遍历文件系统得到的信息被大大地忽略了。具体来说，若文件系统是一致的，那么fsck.f2fs得到的信息应当能够加速下一次挂载的速度，因为我们能够利用fsck.f2fs先前遍历文件系统得到的信息帮助f2fs挂载，避免了不必要的硬盘读取；其次，由于fsck.f2fs遍历文件系统得到了整个文件系统的元数据及其组织信息，我们能够根据这些信息指导底层f2fs文件系统的运行，比如指导其对segment、block的分配等。据我们所知，目前没有工作将fsck.f2fs与f2fs相互配合，来实现更智能的f2fs文件系统，我们的测试表明，通过将fsck.f2fs得到的信息提供给f2fs，我们能够显著地加快文件系统mount的启动速度。

**难点所在**

但是想利用fsck收集的信息没有那么简单。首先，fsck运行在用户态，而f2fs文件系统运行在内核态，要将fsck信息提供给f2fs，我们必须解决用户态和内核态的通信问题，对于我们的场景，则是用户态到内核态的单向数据传送。其次，fsck扫描文件系统收集到的数据通过一个结构体f2fs\_sb\_info（简称sbi）来表示，sbi结构体里嵌套了其他复杂的结构体信息，分别记录了sit表，nat表等数据，甚至，sbi里还含有指针的信息，我们如何将复杂的数据结构通过某种手段保存下来？这是我们面临的第二个难题。最后，要想利用fsck收集到的数据来指导f2fs文件系统的工作，我们必须对f2fs底层文件系统的工作原理或者某些方面有清晰的认识，明白其缺陷及造成这些缺陷的原因，并且弄清楚为何fsck收集的数据能够应对这些缺陷，这要求我们对f2fs和fsck都有比较充分的认识。

**利用fsck信息**

总的来说，利用fsck收集的信息来优化f2fs有很高的应用价值，但是这却并不是一件简单的事情。我们目前的工作侧重于解决1、2点，而第3点基于1,2点的基础和对fsck及f2fs的进一步理解，应该也不难实现。通过解决第一个难点：用户态和内核态通信问题，我们搭建起f2fs如何利用fsck信息的框架。通过解决第二个难点，我们验证了fsck的信息确实能够优化f2fs文件系统的运行，以此实现更加智能的f2fs文件系统。

## 国内外研究概况

本小节将分别介绍三个目标加速fsck.f2fs、优化f2fs、利用fsck收集的信息的gc的国内外研究概况。

### 加速fsck.f2fs

我们首先简要介绍当前硬件趋势和C/R工具的背景，然后介绍fsck工具和加速后的pfsck工具，最后介绍我们将对其改进的fsck.f2fs工具。

#### 1.2.1.1 硬件和软件趋势

现代超高速存储设备如ssd和NVMe不仅提供高带宽(8- 16gb /s)，而且与传统硬盘相比，存储访问延迟降低了两个数量级(< 20usec)。另一方面，像英特尔的DC Optane[16]这样的快速存储类内存和其他字节可寻址的持久内存技术正在发展，访问延迟< 1usec。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。大量以往和正在进行的研究正在开发优化的文件系统来支持快速存储硬件。这包括ssd[17]、nvme[18]的文件系统，为nvm优化传统Ext4和XFS文件系统的开源努力[19]，以及其他研究工作[20,21,22]。然而，减少这些文件系统的数据损坏和错误需要几年的生产使用[23,24]。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，但它们尚未充分利用硬件存储优势和多核并行性。

#### 1.2.1.2 文件系统检查和修复

自从文件系统出现以来，一致性一直是一个问题。尽管诸如日志记录、写时复制、日志结构写入和软更新等存储机制已经被开发出来以减少不一致性的情况，但它们是有限的，因为它们不能修复由软件错误或由故障磁盘、位翻转、过热或崩溃等事件引起的错误[25-29]。这时候，使用流行的C/R工具，如fsck、e2fsck和xfs\_repair[30]，通过遍历文件系统的布局并检查inode一致性、目录一致性、文件和目录连接性、目录项一致性以等，来检测和修复文件系统的损坏和错误。

在实际环境中，文件系统C/ R的频率变化很大。虽然缺乏C/R最佳实践，但在目前的大型和个人计算系统中，fsck、e2fsck和xfs\_repair等C/R工具对于数据可靠性仍然至关重要，因为它们通常在系统错误[31,2,3,28]、硬件或内核升级，或在强制安全更新之后运行。不频繁的C/R会将系统停机时间延长至3小时[5]，在极端情况下，在pb级文件系统上，停机时间长达数周[4]。

#### 1.2.1.3 检查和修复工具e2fsck

E2fsck是针对ext系列文件系统的C/R工具。它的检查流程分为五个阶段。对第一个阶段pass -1检查索引节点元数据的一致性;第二个阶段Pass-2检查目录一致性;第三个阶段Pass-3检查目录连通性;第四个阶段pass-4检查引用计数;最后，第五个阶段Pass-5检查数据和元数据位图的一致性。

#### 1.2.1.4 检查和修复工具pfsck

pfsck是基于e2fsck进行了优化。pfsck采用四种方式来加速和优化，分别是①通过数据并行性并发执行检查任务。②通过减少pass间的依赖关系来启用pass并行性。③通过动态线程调度适应文件系统配置。④通过资源利用感知减少对系统的影响。

**①通过数据并行性并发执行检查任务。**

为了克服当前C/R工具在磁盘、卷或逻辑组级别使用串行或粗粒度并行化技术的瓶颈，pFSCK引入了细粒度数据并行化。由于Pass-1和Pass-2占文件和目录密集型文件系统运行时的90%以上，pFSCK将重点放在这两个pass上。将更精细的文件系统结构(如inode、目录块和目录)划分为多个任务，并在一次pass中并发地执行C/R。虽然看起来很简单，但实现数据并行性需要跨线程进行数据结构隔离，以减少同步瓶颈。

**②通过减少pass间的依赖关系来启用pass并行性。**

虽然数据并行性加速了C/R，但是每个pass(例如，目录检查)都必须等待前一个pass(例如，索引节点检查)完成。具体来说，在C/R中，使用了几个跨pass全局数据结构来构建文件系统的一致视图并识别不一致性(例如位图)。因此，对共享全局结构的更新必须序列化，从而随着线程数的增加，对共享全局结构的争用也在增加，限制了并行速度。为了减少串行化开销，pFSCK设计了并行pass，打破了pass之间串行执行的局限，允许多个pass同时执行，并减少I/O等待时间。

**③通过动态线程调度适应文件系统配置。**

数据和pass并行性都需要在不同的pass上分配线程。由于缺乏有关元数据类型(文件、目录、链接)，各pass的工作量的信息，CPU线程的静态划分不是最优的。简单的检查，如文件数量与目录索引节点的信息是不够的，因为目录处理是复杂和耗时的。为了克服这些问题，pFSCK设计了一个C/R线程调度器，它可以动态地分配和迁移线程，以便适应不同的文件系统配置。

**④通过资源利用感知减少对系统的影响。**

文件系统C/ R可能与其他应用程序一起运行。考虑到pFSCK的目标是利用可用的cpu，它可能会影响其他一起运行的应用程序。类似地，C/R也可以运行在其他应用程序用来存储数据的磁盘上。为了减少整个系统对共同运行的应用程序和pFSCK的影响，为pFSCK的调度器配备了资源感知功能，以便动态识别在不同时间要使用的内核数量，以最大限度地减少对其他共同运行的应用程序和pFSCK性能的潜在影响。

#### 1.2.1.5 f2fs文件系统

本小节将依次介绍为什么要f2fs，f2fs特性，f2fs磁盘布局， f2fs数据组织，检查和修复工具fsck.f2fs。

**为什么要f2fs**

要明白f2fs的设计原理，明白f2fs为什么好，我们必须从实际的背景出发。我们首先要弄清楚f2fs文件系统是针对什么存储设备提出的（是什么），其次是目前的文件系统存在哪些问题（为什么），最后是f2fs是如何设计来解决这些问题的（怎么做）。只有这样，我们才能体会到f2fs设计的精妙之处。接下来，我们将从“观察”和“总结”两个层面，循序渐进，引出f2fs文件系统的必要性。

**观察：**

① 基于NAND Flash（NAND闪存）的存储介质，比如SSD,eMMC以及SD卡，相比硬盘(HHD)来说，具有更低的访问延迟，在随机读方面，更是比硬盘的访问速度高出一个数量级。因此，Flash存储介质已经被广泛地应用于从移动端设备到服务器端的各类系统。但是，Flash存储介质仍存在一些限制，比如：写前擦除、有限的擦除次数，这使得Flash需要按顺序写入擦除的块，并且尽量使得各个块擦除次数一致（磨损均衡）。

② 在早期，许多消费电子设备直接将"bare" NAND闪存连接到一个系统。然而，随着存储需求的增长，使用通过专用控制器连接多个闪存芯片的解决方案越来越普遍。控制器上运行的固件通常称为FTL（闪存转换层），解决了NAND闪存的限制，并提供了通用的块设备抽象。这种闪存解决方案的示例包括eMMC（嵌入式多媒体卡），UFS（通用闪存）和SSD（固态驱动器）。通过FTL的抽象，我们可以将一个NAND闪存设备当做一个块设备，此时，当前存在的针对块设备的文件系统，可以不加修改地运行在NAND闪存中，但是，由于Flash本身固有的特性（写前擦除等），大量频繁的随机写将会大大降低NAND闪存的性能并降低其寿命。更糟糕的是，随机写的场景在移动端设备十分常见。

③ 20世纪90年代初提出的日志结构文档系统（LFS），是为了缓解硬盘随机写引发的多次寻道所带来高开销而提出，通过以类似日志的结构按顺序将所有修改写入磁盘，从而加快了文件写入和崩溃恢复的速度，这是一种对文件数据异地更新的方法。日志是磁盘上的唯一结构;它包含索引信息，以便可以有效地从日志中读回文件。为了在磁盘上维护较大的可用区域以进行快速写入，还将日志划分为多个segment，并使用segment清理器压缩来自严重碎片化segment的实时信息。但是LFS仍存在着众所周知的漫游树（wandering tree）和高清理开销（hign cleaning overhead）的问题。LFS的思想虽然是针对硬盘首次提出，却能够在多年后与NAND Flash存储介质完美结合，解决NAND Flash上随机写的问题。

**总结：**

① 由观察1，我们知道NAND闪存介质应用十分广泛，针对NAND闪存介质进行优化十分必要；由观察2，FTL可以将NAND闪存抽象为一个块设备，可是，针对传统块设备的文件系统不能很好地应用在NAND闪存介质上；由观察3，LFS文件系统异地更新、顺序写入的结构给了我们很好的启发，我们可以应用此思想解决在Flash上随机写入的问题，然而LFS存在漫游树和高清理开销的问题。因此，我们可以明白设计f2fs文件系统的必要性，即f2fs是一种Flash-aware的新型文件系统，基于LFS，并能够解决其潜在的问题。

② f2fs是一个利用基于NAND闪存的存储设备的文件系统，它基于日志结构文档系统（LFS）。该设计专注于解决LFS中的基本问题，即漫游树的滚雪球效应和高清理开销。由于基于NAND闪存的存储设备根据其内部几何形状或闪存管理方案（即FTL）表现出不同的特性，因此f2fs及其工具不仅支持各种参数，用于配置磁盘布局，还支持动态调整分配和清理算法。

**f2fs特性**

了解了f2fs存在的必要性，我们就能得到f2fs需要拥有的一些重要特征，后面f2fs的磁盘布局和数据组织都是围绕着这些特性展开的。

① 闪存友好的磁盘布局：f2fs是一种Flash-aware的文件系统，其磁盘数据结构经过精心布局，以匹配底层 NAND 闪存的组织和管理方式。f2fs采用3个可配置单元：segment、section、zone。它以segment为单位从多个单独的zone分配存储块。它以section为单位进行清理，引入这些单元是为了与底层FTL的操作单元保持一致，以避免不必要且成本高昂的数据复制。

② 高效的索引结构：解决了LFS的漫游树问题。LFS 将数据和索引块写入新分配的可用空间。如果叶数据块已更新（并写入某处），则其直接索引块也应更新。写入直接索引块后，应再次更新其间接索引块。这种递归更新会导致写入链，从而产生"漫游树"问题。为了解决这个问题，f2fs给出了一种新的索引表，称为节点地址表（Node address Table）。当叶数据块更新时，只需要更新相应索引块在节点地址表中对应的块地址即可。

③ 多头日志记录（Multi-head logging）：缓解了LFS高清理开销的问题。f2fs设计了一种有效的热/冷数据分离方案，应用于日志时间（即块分配时间）。它同时运行多个活动日志段（logging segment），并根据预期的更新频率将数据和元数据附加到单独的日志段中。由于闪存设备利用介质并行性，因此多个活动段可以同时运行，而无需频繁的管理操作，因此由于多个日志记录（与单段日志记录相比）而导致的性能下降微不足道。通过Multi-head logging，冷的日志数据段里面的block，通常处于稳定的状态，不会被移动；而热的日志数据段，由于经常发生变化，在清理时，大部分block已经处于无效的状态，需要移动的有效block较少，大大降低了清理的时间。

④ 自适应日志记录（Adaptive logging）：f2fs 基本上创建在仅追加日志记录（append-only logging）之上，将随机写入转换为顺序写入。然而，在高存储利用率下，它将日志记录策略更改为穿插日志记录（threaded logging），以避免长时间的写入延迟。实质上，穿插日志记录将新数据写入脏段中的可用空间，而不在前台清理它。此策略虽然在机械硬盘上不起作用，却在现代闪存设备上效果很好。

⑤ 通过前滚恢复机制（roll-forward recovery）加速：解决移动端随机写和fscyn频繁造成Flash设备寿命短、延迟高的问题。f2fs 通过最小化所需的元数据写入并使用高效的前滚机制恢复同步数据，优化小型同步写入以减少 fsync 请求的延迟。通常，上层调用fsync时，文件系统需要将所有缓存数据同步到硬盘，在存在大量fsync的场景下，此操作会带来巨大的开销，f2fs 实现了高效的前滚恢复机制来增强 fsync 性能。关键思想是仅写入数据块及其直接节点块，不包括其他节点或 f2fs 元数据块。为了在回滚到稳定检查点后有选择地查找数据块，f2fs 在直接节点块内保留一个特殊标志。

**f2fs磁盘布局**

后面两小节将分别介绍f2fs的磁盘布局和数据组织，这对于后续理解fsck.f2fs的工作原理至关重要。首先是磁盘布局，f2fs的磁盘布局和上文介绍的f2fs特性互相呼应，特性要求为磁盘布局提供了设计原则，而磁盘布局为特性提供了一个具体实现。

f2fs将整个volume划分为多个segment,每个segment的尺寸固定为2MB。section由连续segment组成,zone由一组section组成。默认情况下,section和zone大小都设置为一个segment大小,但用户可以通过mkfs轻松修改该大小。 f2fs将整个volume划分为六个区域,除超级块外,所有区域都由多个segment组成,如图1-1所示:

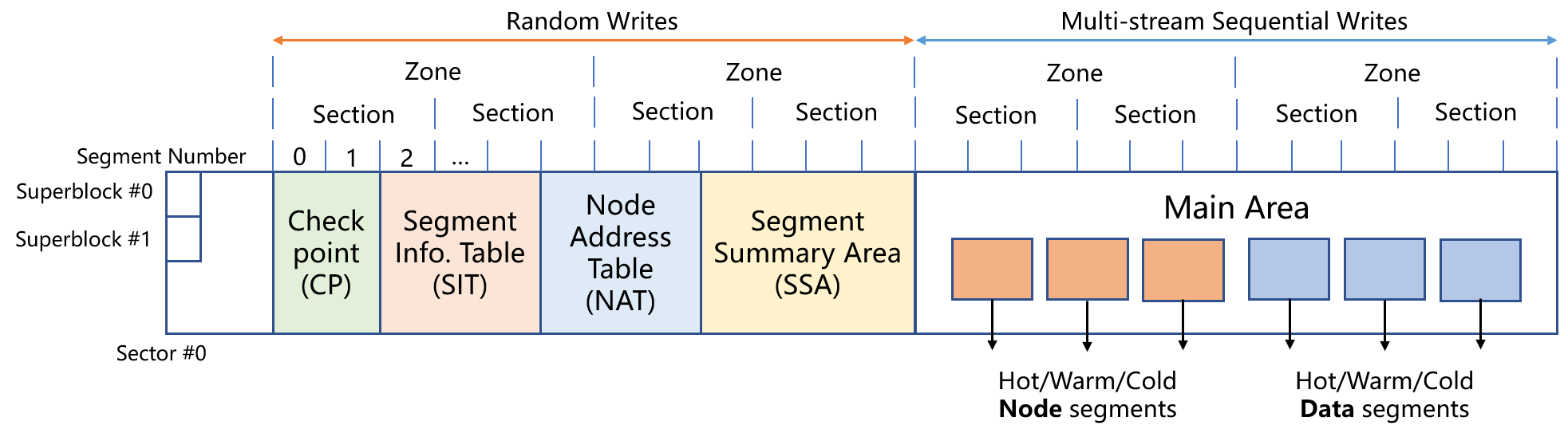


图 1‑1 f2fs布局

**Superblock (SB)：**它位于分区(partition，注意：文件系统构建在硬盘分区之中)的开头，并且存在两个副本以避免文件系统崩溃。它包含基本的分区信息和f2fs的一些默认参数。

**Checkpoint (CP)：**它包含文件系统信息、有效 NAT/SIT 集的位图、孤立索引节点列表（orphan inode lists）和当前活动段的摘要条目（summary entries of current active segments）。

**Segment Information Table (SIT)：**它包含segment的信息,例如有效块计数和所有块有效性的位图。

**Node Address Table (NAT)：**它由存储在main area的所有node blocks的块地址表组成。

**Segment Summary Area (SSA)：**它包含存储在main area中的所有data blocks和node blocks的所有者信息的摘要条目。清理时需要根据此信息找到main area中某个block所属的node节点。

**Main Area：**由两种类型的block组成，node block或者data block。其中node block包括inode或者data block的索引块，而数据块包含目录或文件的具体数据。

为了避免文件系统和基于闪存的存储之间不一致，f2fs 将 CP 的起始块地址与segment大小对齐。此外，它还通过在 SSA 区域中保留某些segments，将main area的起始块地址与zone的大小对齐。

**f2fs数据组织**

接下来介绍f2fs的数据组织。f2fs的数据组织是在磁盘布局的基础之上，对文件系统重要功能具体实现的介绍。这里不会牵涉到太多细节，却能让我们从宏观的角度对整个文件系统设计有一定程度的把握，这对后面明白fsck.f2fs工作原理也是必要的。

① 文件结构：如图1-2所示，f2fs 使用基于指针的文件索引和直接、间接节点块来消除更新传播（即“漫游树”问题 ）。在传统的 LFS 设计中，如果一个叶子数据被更新，它的直接和间接指针块被递归地更新。而f2fs只更新一个直接节点块及其对应的NAT表项，有效解决了漫游树问题。一个 inode 块包含指向文件数据块的直接指针、两个单间接指针、两个双间接指针和一个三重间接指针。 f2fs支持内联数据和内联扩展属性，将小数据或扩展属性嵌入到inode块本身。内联减少了空间需求并提高了 I/O 性能。请注意，许多系统都具有小文件和少量扩展属性。默认情况下，如果文件大小小于 3,692 字节，f2fs 会激活数据内联。 f2fs在一个 inode 块中预留 200 字节用于存储扩展属性。

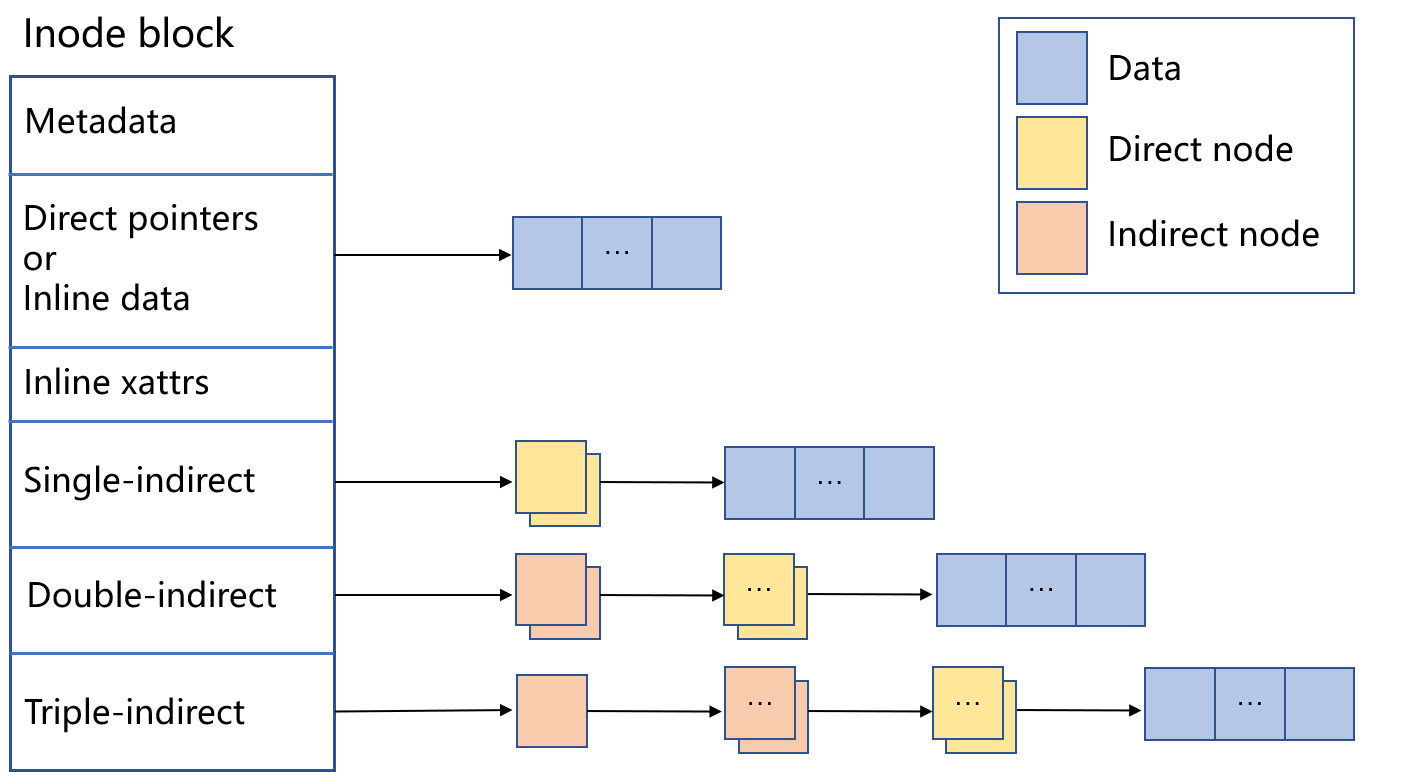


图 1‑2 f2fs文件组织结构

② 目录结构：在 f2fs 中，一个 4KB 目录条目（dentry）块由一个位图和两个成对的插槽(dentry结构体、名称)数组组成。bitmap指示每个插槽是否有效。dentry结构体具有哈希值、索引节点号、文件名长度和文件类型（例如，普通文档、目录和符号链接）属性；而name是一个大小为8的字符数组，由于文件名的长度可能大于8，因此，一个目录项可能会占用多个插槽。图1-3展示了一个目录的数据块（目录条目块）在中硬盘中的布局。

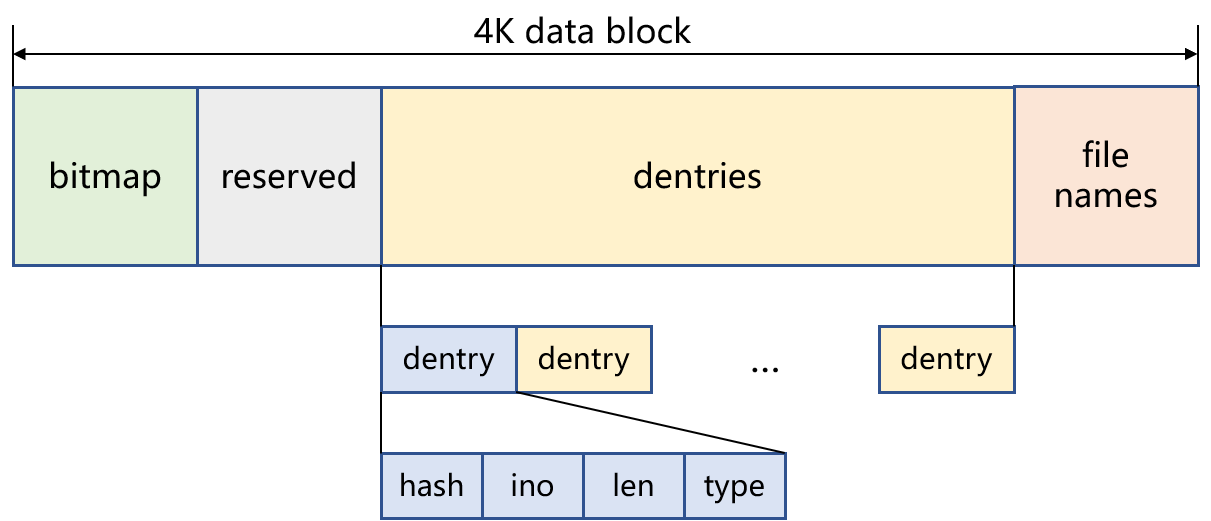


图 1‑3 目录条目块4K block

目录构造多级哈希表，以有效地管理大量目录项。当 f2fs 在目录中查找给定的文件名时，它首先计算文件名的哈希值。然后，它以增量方式遍历构造的哈希表，从级别 #0 到索引节点中记录的最大分配级别。在每个级别中，它扫描一个包含两个或四个目录条目块的存储桶，导致 O(log(# of dentries)) 复杂性。为了更快地查找条目，它会按顺序依次比较位图、哈希值和文件名。当需要海量的目录项时（例如，在服务器环境中），用户可以配置 f2fs 在最初时分配更多的目录项，即使用较低级别的较大哈希表，这样，f2fs 可以更快地到达目标条目。图1-4给出了一个多级哈希表的示意图，目录的目录条目块组织成一个多级哈希表，每一级由多个bucket组成，而每个bucket包含了多个目录条目块。搜索某个文件时，在每一级对应的bucket中依次搜索目录项，且每一级只会搜索一个bucket。举一个例子：当 f2fs 在目录中查找某个文件名时，首先计算文件名的哈希值。然后，f2fs 扫描级别 #0 中的哈希表，以查找由文件名及其索引节点号组成的条目。如果未找到，f2fs 将扫描级别 #1 中的下一个哈希表。通过这种方式，f2fs 从 0 到 N 以增量方式扫描每个级别的哈希表。

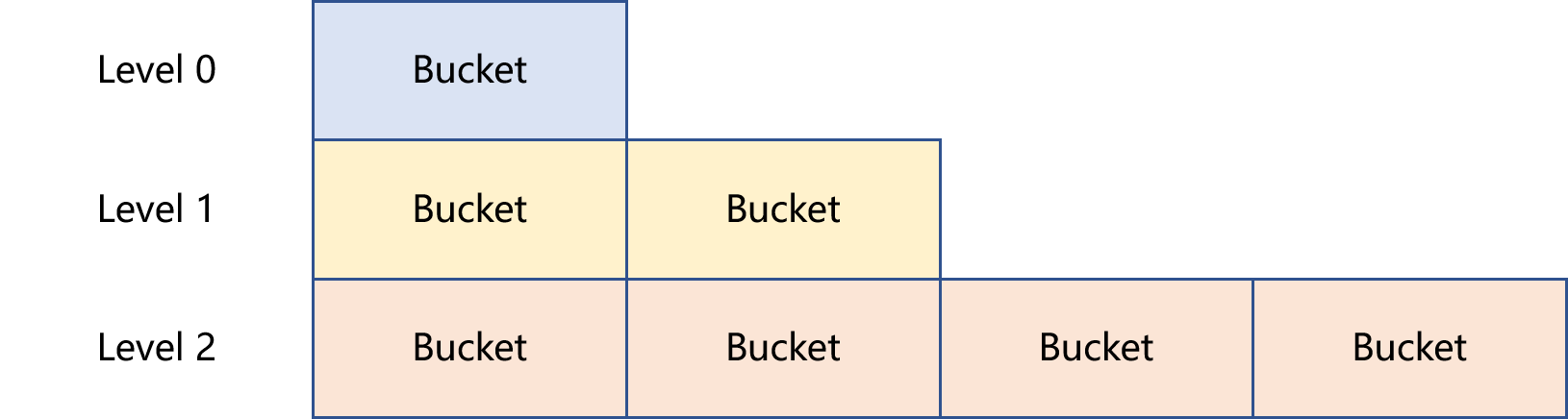


图 1‑4 目录多级哈希表

③ 日志记录（Logging）：与只有一个大日志区域的LFS不同，f2fs维护六个主要日志区域，以最大限度地实现冷热数据分离的效果。f2fs静态地为节点和数据块定义了三种温度级别——hot、warm和cold，如表1-2所示。直接节点块被认为比间接节点块更热，因为它们更新得更频繁，而间接节点块包含节点ID（指向了下一个节点块），仅在增加或删除特定节点块时写入。目录的直接节点块和数据块被认为是热的，因为与普通文件的块相比，它们具有明显不同的写入模式。某些数据块被认为是冷的，如多媒体数据，因为它们一般不会被写入，通常是只读的。

表 1‑2 不同对象的划分

|  |  |  |
| --- | --- | --- |
| **Type** | **Temp.** | **Objects** |
| Node | Hot | Direct node blocks for directories |
| Warm | Direct node blocks for regular files |
| Cold | Indirect node blocks |
| Data | Hot | Directory entry blocks |
| Warm | Data blocks made by users |
| Cold | Data blocks moved by cleaning;  Cold data blocks specified by users;  Multimedia file data |

LFS有两种空闲空间管理方案:穿插日志（threaded log）和仅追加日志（append log）。仅追加日志方案非常适合具有非常好的顺序写入性能的设备，因为空闲段一直用于写入新数据。然而，在高利用率的情况下，它会受到清理开销的影响。相反，穿插日志方案不得不采用随机写，这会降低写入性能，但不需要清理过程。f2fs采用混合模式，默认采用仅追加日志，但根据文件系统状态动态更改策略为穿插日志模式（如空闲的segment数量少于K时，变换为穿插日志模式，而K是一个预定义的值）。

为了使f2fs与底层基于闪存的存储保持一致，f2fs以section为单位分配segment。f2fs期望section大小与FTL中垃圾收集的单位大小相同。此外，对于FTL中的映射粒度，f2fs尽可能地在不同的zone中分配活动日志，否则，由于FTL可以根据其映射粒度将活动日志中的数据写入一个分配单元，这就违背multi-head logging的初衷，并且无法缓解系统清理开销。

④ 清理：f2fs可以根据需要（on demand）和在后台（in the background）进行清理。当没有足够的空闲段来服务VFS调用时，触发按需清理。后台清理器由内核线程操作，在系统空闲时触发清理作业。

f2fs支持两种受害者选择策略（victim selection policies）:贪心算法和成本-收益（cost-benefit）算法。在贪心算法中，f2fs选择有效块数量最少的受害段（victim segment）。在成本效益算法中，f2fs根据segment的年龄和有效块的数量选择受害段，以解决贪心算法中的日志块抖动问题。f2fs按需清理采用贪心算法，后台清理采用成本效益算法。这是因为用户需等待按需清理完成，此时间必须足够短，因而采用贪心算法；而后台清理是系统空闲时进行，系统有足够的时间做出最优决策，这时可以选择时间长但效果更好的成本效益算法。

为了识别受害段中的数据是否有效，f2fs管理一个位图。每个位代表一个块的有效性，位图由覆盖main area所有块的位流（bit stream）组成。此位图保存在SIT表中。

⑤ 检查点和恢复：f2fs实现检查点，以便在突然电源故障或系统崩溃时提供一致的恢复点。当它需要在sync、umount和前台清理等事件中保持一致状态时，f2fs触发一个检查点过程，如下:(1)刷新页面缓存中的所有脏节点和dentry块;(2)暂停普通的写活动，包括create、unlink和mkdir等系统调用;(3)将文件系统元数据(NAT、SIT和SSA)写入磁盘上各自的专用区域;(4)最后，f2fs写一个检查点包(checkpoint pack)到CP区域，其包括以下信息：

**Header**和**Footer**分别写在pack的开始和结束。f2fs在Header和Footer中维护一个版本号，该版本号在创建检查点时递增。版本号在挂载期间区分两个记录的pack之间的最新的稳定的pack;

**NAT和SIT位图**表示包含当前pack的NAT和SIT块的集合;

**NAT和SIT日志**包含少量最近修改的NAT和SIT条目，以避免频繁的NAT和SIT更新;

**活动段的摘要块（summary block）**由内存中的SSA块组成，这些块将在将来被刷新到SSA区域;

**孤儿块（orphan blocks）**保存“孤儿inode”信息。如果一个inode在关闭之前被删除(例如，两个进程打开一个公共文件，一个进程删除它)，它应该被注册为孤立inode，以便f2fs可以在突然断电后恢复它。

checkpoint在硬盘中的表示如图1-5所示，可以看出checkpoint分为两种模式——Normal模式和Compacted模式。在Normal模式下，每个日志区域都有一个块来保存其摘要信息；而Compacted模式下，多出了保存nat journal和sit journal的块，而所有的数据（hot、warm、cold）的日志区域共享一个摘要块。上面提到的NAT和SIT的位图，在图中没有显示出来，通过查看f2fs源码可知，其位于f2fs\_checkpoint结构体的后面（整个f2fs\_checkpoint的大小不足4K，后面的部分充当NAT、SIT的位图）。

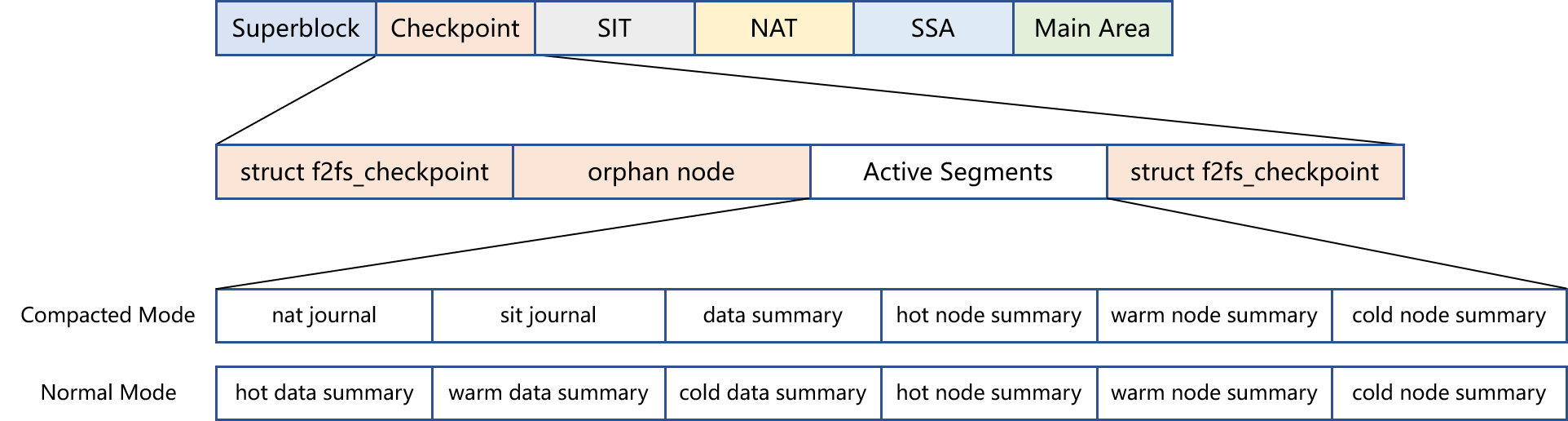


图 1‑5 Checkpoint 结构

后向回退修复（Roll-Back Recovery）:在突然断电后，f2fs回滚到最近的一致检查点。为了在创建新包（Pack）时保持至少一个稳定的检查点包，f2fs维护两个检查点包。如果检查点包在Header和Footer中具有相同的内容，f2fs认为它是有效的。否则，它将被丢弃。在挂载时的恢复过程中，f2fs通过检查Header和Footer来搜索有效的检查点包。如果两个检查点包都有效，f2fs通过比较它们的版本号来选择最新的一个。一旦选择了最新的有效检查点包，它就会检查孤儿inode块是否存在。如果是这样，它将截断它们引用的所有数据块，最后也释放孤儿inode。最终，在前滚恢复过程成功完成之后（在下文介绍），f2fs使用一组一致的由位图引用的NAT和SIT块启动文件系统服务。

前向回滚修复（Roll-Forward Recovery）:像数据库(例如SQLite)这样的应用程序经常将小数据写入文件并进行fsync以保证持久性。支持fsync的最简单的方法是触发检查点并使用后向回退模型恢复数据。然而，这种方法会导致较差的性能，因为检查点涉及到写入与数据库文件无关的所有节点和dentry块。f2fs实现了高效的前滚恢复机制，提高了fsync性能。关键思想是只写数据块及其直接节点块，不包括其他节点或f2fs元数据块。为了在回滚到稳定检查点后选择性地查找数据块，f2fs在直接节点块中保留了一个特殊标志。

**检查和修复工具fsck.f2fs**

fsck.f2fs的工作流程如图1-6所示，检查主要分了三个步骤：初始化、修改以及验证。初始化流程主要是根据读取硬盘得到的信息，对NAT区域位图、SIT区域位图和MainArea位图信息进行一个初始化，在后续修改的步骤，会使用这些初始化的信息，对文件系统的一致性进行验证；修改步骤是检查的核心，也是我们对fsck.f2fs修改得最多的部分，这里的逻辑是从根目录对应的inode出发，遍历整个文件系统，并且在此过程中记录对应的信息，比如，记录下整个Main Area区域中有效的块数、每个文件的链接数，Main Area区域中各个segment中对应块有无被使用的位图等，通过此类信息，我们将能够验证文件系统的有效性；验证步骤即对前面收集到的信息与硬盘记录的元数据信息进行比对，来对文件系统进行检查，如果检查到了文件系统的不一致，用户可以选择是否调用fsck\_xxx\_func类型的函数对文件系统进行修复。

由于我们的工作核心在于fsck\_chk\_node\_blk函数，也就是上面提到的修改步骤，我们在下面再重点介绍一下此步骤。如图1-6步骤2所示，fsck程序通过硬盘保存的元数据信息（do\_mount后保存在sbi结构体中），我们可以获取根目录对应的node id。通过fsck\_chk\_node\_blk对此node id进行检查，这个过程是递归的。具体来说，在fsck\_chk\_node\_blk里，会根据不同的node节点类型（上面提到node分为inode类型、dinode直接数据块、idinode间接数据块等），调用不同的检查函数进行检查，比如根目录，对应的node节点为inode类型，因此会调用fsck\_chk\_inode\_blk进行检查，在fsck\_chk\_inode\_blk中，首先根据读取到的inode节点信息，记录inode硬链接数等，后面将会对此类元数据进行验证；然后是对数据的处理，会根据此inode是否存在内联的数据来决定如何进行下一步操作，如果不存在内联的数据，这会遍历每个数据块，通过调用fsck\_chk\_data\_blk进行检查；除了inode节点存在的数据指针直接指向数据块外，inode节点还存在5个node指针间接地指向了数据块。因此，如果存在这类node指针，fsck\_chk\_inode\_blk中会调用fsck\_chk\_node\_blk对node进行检查，形成了一个递归的结构。如果是目录的数据块，里面保存了目录项的信息，因此，在fsck\_chk\_data\_blk里要进一步调用fsck\_chk\_dentry\_blk对目录项进行检查，fsck\_chk\_dentry\_blk收集了目录项的信息后，调用\_chk\_dentries真正进行各个目录项的检查操作。如果某个目录项是有效的，\_chk\_dentries内还会对此目录项记录的inode信息进一步调用fsck\_chk\_node\_blk进行检查，再次形成了一个递归的结构。图1-6步骤2蓝色标记的块和相应箭头描述了我们上面所述的各个函数以及他们之间的关系。

注意，我们没有刻画fsck.f2fs中对扩展块，xattr块等的描述和检查操作， 流程图也省略了很多细节，因为我们只想勾勒出fsck.f2fs检查的具体轮廓和步骤，太多细节反倒会让人迷惑。但是，实际fsck.f2fs检查要考虑很多细节，并且数据之间的依赖也错综复杂，在其中引入并行性不是一件容易的事情，我们已经做了大量的努力来维护系统的一致性，包括但不限于引入细粒度锁、线程独立的私有空间、线程等待和数据聚合等。

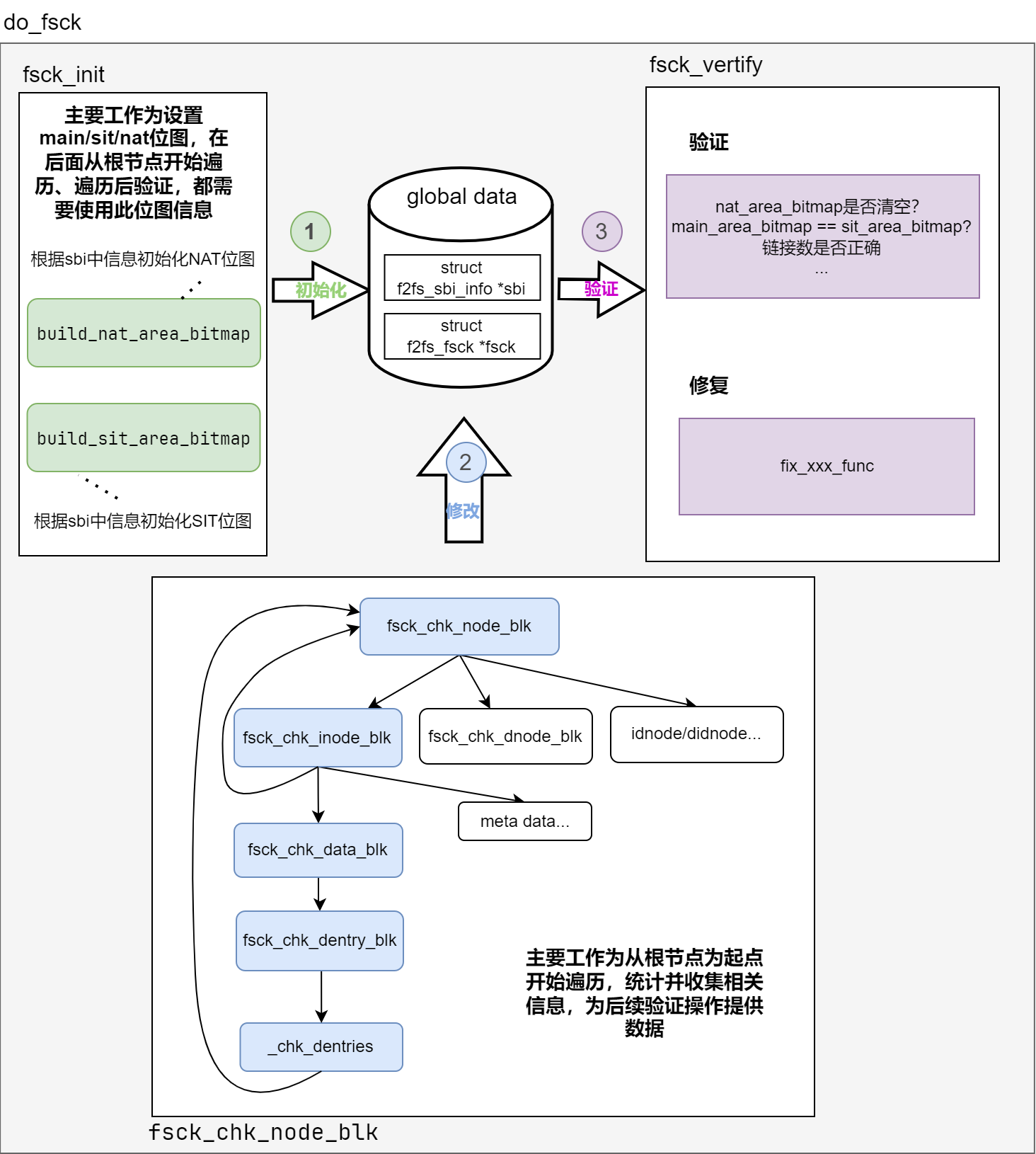


图 1‑6 fsck.f2fs工作示意图

### 优化f2fs的gc

本小节将分别介绍f2fs gc的简介，gc类型，gc的主流程，选择待回收segment，cost算法，目前gc存在的问题。

**f2fs gc简介**

基于Log-structured文件系统的特征，gc的主要作用是回收这些invalid的block，以供文件系统继续使用。f2fs的gc分为前台gc和后台gc: 前台gc一般在系统空间紧张的情况下运行，目的是尽快回收空间; 而后台gc则是在系统空闲的情况下进行，目的是在不影响用户体验的情况回收一定的空间。前台gc一般情况下是在checkpoint或者写流程的时候触发，因为f2fs能够感知空间的使用率，如果空间不够了会常触发前台gc加快回收空间，这意味着文件系统空间不足的时候，性能可能会下降。后台gc则是被一个线程间隔一段时间进行触发。

**gc类型**

f2fs支持两种类型的gc，分别是后台gc（BG\_gc）和前台gc（FG\_gc）。后台gc以线程化执行的方式进行，在后台任务中定期执行，不会阻塞系统，在系统空闲（没有IO操作）时执行，可被随时被终止，并且不能保证能回收到空间。前台gc是在系统空间紧缺时，为了确保回收到空间而执行的阻塞操作。

**gc的主流程**



图 1‑7 gc流程

图1-7展示了SSD的gc流程，f2fs原理上类似，将多个segment的有效块搬移到另外的segment，并释放原segment。

一次完成的gc包括如下三个步骤：

* Victim selection：选择待回收的section(segment)。
* Valid block identification and migration：快速识别有用的block，并且将其迁移到其他地方。
* Post-cleaning process：有用block全部被迁移后，segment会被放入到PRE链表中，等待一次gc流程，将其释放。

**选择待回收segment**

选择待回收segment的步骤如下。

* f2fs会根据gc类型选择gc策略和脏数据链表，并将其搜索下标设置为上一次搜索的结果。
* 如果本次是FG\_gc，可以用上次BG\_gc选中的segment，因为BG\_gc可能打断，也就是选择了segment但没有回收；FG\_gc为了加快速度，可以直接使用其结果。
* 先从上一次同类型gc选中的segment开始向右搜索，如果找不到合适的segment，则从0开始继续搜索。
* 校验本次搜索到的segment，不能是正在使用的curseg（因为要将blocki迁移出去，需要分配新的block，如果回收和正在使用的是同一个segment，那将没有意义。）。同时不能是正在前台gc回收的segment，这主要是避免SSR分配选择了正在gc的segment。
* 计算迁移该segment的开销，和当前最小开销比较，记录最小开销的结果。
* 如果搜索次数达到最大，则将最后搜索的segment记录下来，下次可以直接从该segment开始搜索。
* 如果本次搜索到待回收的segment，如果是FG\_gc类型，则记录在cur\_victim\_sec；如果是BG\_gc，则记录在victim\_secmap中。

**cost算法**

选择segment时会计算迁移该segment的开销，cost算法就是用来计算此开销的。f2fs使用了两种计算cost的算法，分别是Greedy算法和Cost-Benefit算法。

* **Greedy算法**

选择invalid block最多(valid block最少)的segment进行gc。

* **Cost-Benefit算法**

Cost-Benefit算法是一个同时考虑最近一次修改时间以及invalid block个数的算法。因为相当于频繁修改的数据而言，不值得进行gc，因为gc完很快就修改了，同时由于异地更新的特性，导致继续产生invalid block。较长时间未作修改的数据，可以认为迁移以后也相对没那么频繁继续产生invalid block。Cost-Benefit算法的核心是:

|  |
| --- |
| 其中：  : 表示valid block在该section中的比例；  : 表示对这个section进行gc后的收益；  : 则表示开销(读+写)；  : 代表该section最近一次修改的时间。 |

因此我们可以将Cost-Benefit算法理解为一个平衡invalid block数目以及修改时间的的一个算法。

**目前gc存在的问题**

在写的场景下，当空间将要不足时，会触发前台gc。前台gc会阻塞当前写操作，影响性能。并且我们发现在通过工具fsmark创建大量文件并且空间将要不足时，会触发大量的无效gc。这里的无效gc又分为了两种。一种是代码跑进了gc的逻辑，但是系统并没有脏的segment可供回收。一种是待回收的segment有效块数非常多，比如一个segment总共512个块，有效块510个，那么回收这个segment其实并不会释放出多余的空间。因为回收操作是将该segment上的有效块迁移到对应类型的cur segment，若cur segment的空间不够容纳这些有效块，那么会用到新的segment进行迁移。所以在一个待回收segment上的有效块非常多的情况下，对其进行gc通常并不会使得可用的segment增加。而且因为f2fs中的实现是在每次创建文件时，判断若空间将要不足则调用前台gc，比如创建1000个文件就会调用1000次，而这1000次gc可能都是无效的，白白浪费时间，影响写的性能。

### 1.2.3 利用fsck收集的信息

据我们所知，目前还没有工作充分利用了fsck收集到的信息，而这些信息是非常宝贵的。如1.1.1节所属，C/R速度非常慢，随着文件和目录数量的增加，C/R时间呈线性增长[7,8-11]，有时持续数小时[5]，甚至数周[4]。而这种情况随着存储设备容量的不断增长将继续恶化。即使使用我们的wfsck程序，极大地加快了C/R的速度，检测与修复的时间同样令人难以忍受。经过如此长时间的C/R过程，我们才能判定一个文件系统是否被损坏。如果文件系统被损坏了，我们可以使用fsck程序进行修复；可如果文件系统没有损坏，这些时间就被白白浪费了。

在文件系统正常的情况下，fsck获得的数据也许对将来f2fs文件系统有重要的使用价值。正是基于这一点，我们提出了使用fsck获得的信息辅助f2fs进行挂载，从而通过减少Flash设备的读写次数来优化f2fs的挂载性能，从而实现更智能的Flash文件系统。

## 项目的主要工作

项目的主要工作为以下三个题目:

* **题目一：**加速fsck.f2fs

将论文pfsck的思想迁移到fsck.f2fs中。实现fsck.f2fs进行C/R的加速，并动态调整线程数量，减少对其他程序的影响。

* **题目二：**优化f2fs的gc

在写大量文件的场景中，当空间快要不足，f2fs将可能触发大量无效前台gc，影响性能。需要减少无效gc，进而减少对性能的影响。

* **题目三：**利用fsck收集的信息

在C/R时，已经遍历了各文件的元数据信息，通过将这些信息记录下来，对后续挂载等情况进行优化。

# 需求分析

分析赛题可知，我们需要实现更智能的flash文件系统。

检查与恢复对于提高flash文件系统的可靠性十分重要。而如今C/R的时间不断增长，随着文件和目录的增加，C/R的时间呈线性增长，有时持续数小时，甚至数周。尽管现代flash和NVM技术提供了更低的延迟和带宽，但当前的C/R工具无法充分利用这些硬件I/O或多核CPU并行性。为了使其更快速，更智能。我们需要引用并发机制加速fsck.f2fs的执行速度。通过引用智能感知资源的机制，让fsck.f2fs智能调整线程个数，减少对系统其他程序的影响。

同时gc对于日志型文件系统十分重要。但是当空间不足，会触发前台gc，严重影响性能。有时这些gc还是无效的gc。我们需要实现一个智能的gc任务的管理框架，减少无效的前台gc对性能的影响。

最后fsck.f2fs检查过程遍历得到了文件系统布局和元数据信息，但是这些信息大大忽略了。据我们所知，目前还没有工作将fsck.f2fs和f2fs进行配合使用。我们需要充分利用fsck.f2fs检查过程中收集信息，优化后续挂载等情况，实现更智能的文件系统。

# 系统设计

## 3.1系统整体架构设计

系统的整体架构图如图3-1所示。其中字体颜色表明了这个模块属于哪个目标，填充的颜色表明了这个模块的完成情况。接下来将对架构进行概述，并介绍系统整体运行流程。

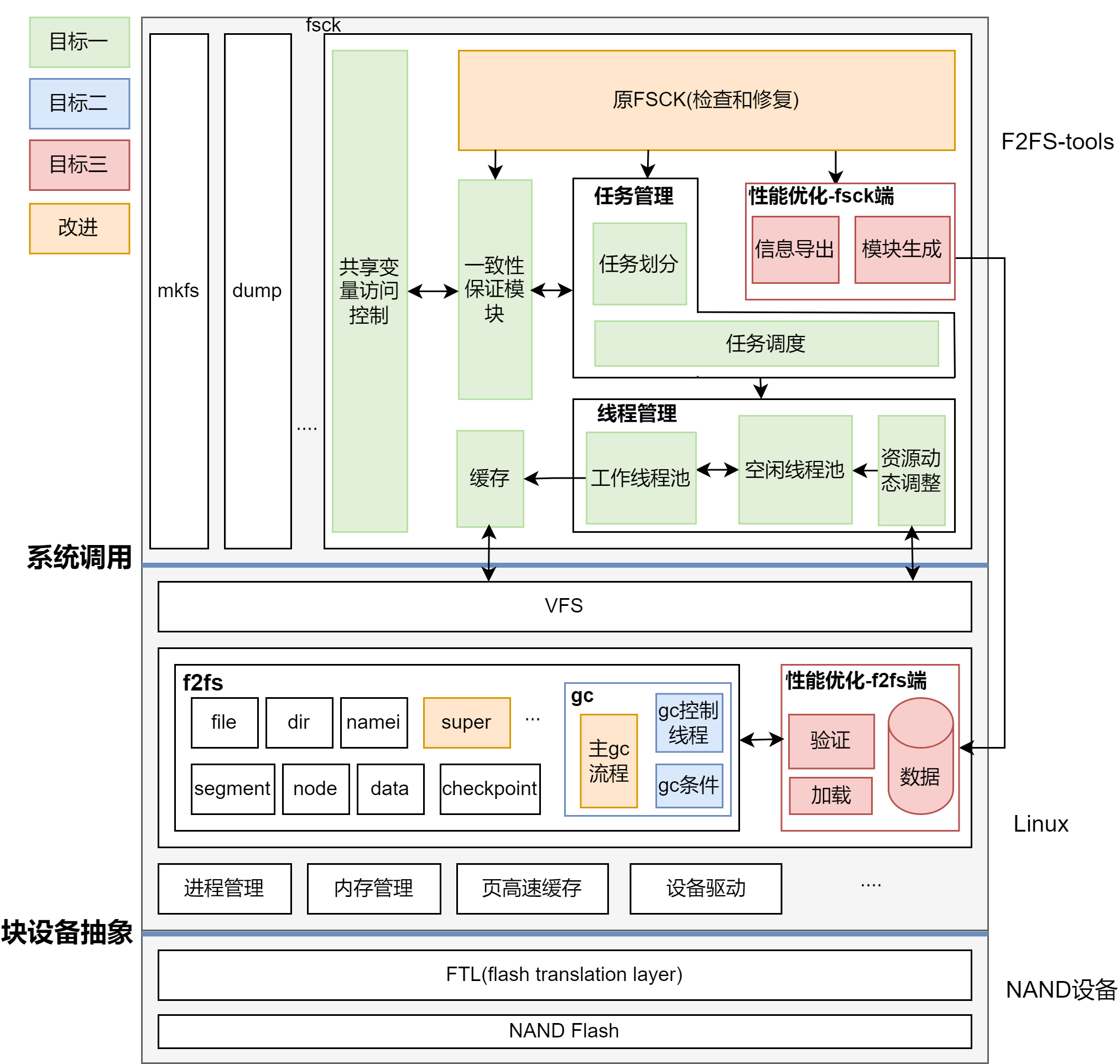


图 3‑1系统整体架构图

### 3.1.1 架构概述

* 任务管理

任务管理包括任务划分和任务调度。任务包括检查任务和调度任务。检查任务就是执行文件系统检查的任务。调度任务是动态调整工作线程个数的任务。

检查任务如何划分是一个难点。需要将原始从根Node开始的递归地对整个Node树的检查进行拆分，拆分成不同的任务，同时对任务的返回值正确地处理。检查任务会被加入到工作线程池，以先入先出的方式被工作线程池里的线程执行。

任务调度是通过调度任务来实现的。调度任务会直接加入到线程池工作队列的头部，拿到该调度任务的线程将被调度到另一个线程池上去。

* 线程管理

调度器线程会根据系统的CPU利用率和当前进程的CPU利用率动态调整当前进程的线程个数。从而减少对系统的影响。线程通过工作线程池和空闲线程池管理起来。空闲线程池中线程的个数是机器CPU核心数。工作线程池中线程个数由程序运行时指定。

* 共享变量访问控制

共享变量访问控制需要安全和高效。安全指的是各线程并发读写不会影响结果的正确性，高效是指执行速度尽量快。对共享数据结构的处理有两种，一种是加锁，一种是变为线程的私有数据。由于原本的单线程逻辑涉及到大量对共享数据结构的访问，在改为并发逻辑后，需要保证访问共享数据结构的原子性。一种简单的实现方式就是加锁，但是若对所有变量都加同一把锁，反而会使得执行时间变得更长。进一步的优化则是对不同的变量加不同的锁。更进一步则是将全局数据分散到各个线程的私有数据中去，通过调用线程库，实现一个类似于线程上下文的东西。该线程对全局数据的访问或更新改为对线程上下文中私有数据的访问或更新。进一步加快执行速度。但不是所有数据都能变成线程的私有数据，若该数据既被各线程读又被各线程写且不只与该线程处理的Node有关，则该数据只能加锁处理，常见的是bit map相关的数据结构。而有的数据，如目录项链表，又被读又被写，但是是由各线程动态向链表中添加目录项，删除目录项，最后目录项链表会为空，该数据可以加入到线程私有数据中。同时需要在所有任务执行完后，对线程私有数据进行结果的聚合。如各线程记录了该线程遍历到的有效的inode个数，在最后需要将各线程有效的inode个数相加，得到系统总的有效inode个数。

* 一致性保证模块

保证整个C/R过程的正确性。这需要正确地做到共享变量访问控制和任务管理。包括对任务返回值的正确处理，正确地记录调用任务前的上下文，正确地将控制不同线程访问共享变量，正确地将各个线程的检查结果进行聚合。

* 缓存

当前检查工具的I/O缓存不是为并发环境而设计的。不同线程访问的是磁盘的不同位置。并发情况下，很可能会导致缓存的一些错误驱逐的情况，无法充分利用现代存储设备的I/O。对此需要重新设计数据结构，每个线程一个I/O缓存。并且需要智能地自适应调整预取窗口地大小，以免预读过多不需要的内容。

* 性能优化

在文件系统的检查过程中，需要遍历文件的元数据信息，而这些信息很可能是可以用于其他优化的。比如利用fsck.f2fs检查过程收集的文件系统信息，优化后续挂载。

* gc

在空间将不足时，写操作会触发前台gc，会影响性能。并且这些gc很可能是无效的gc，需要对gc触发进行控制，从而减少无效的gc。

### 3.1.2 系统整体运行流程

**(1) 加速fsck.f2fs**

目标1的系统整体运行流程如图3-2所示。原本的单线程的检查流程是①检查元数据信息。②从根Node出发，对根Node下的各子Node进行递归检查，并对递归的返回值进行处理。③对检查结果进行核对。而引入并发机制后，将②中的递归调用都封装为一个任务，放入任务队列。线程池里的线程会不断从任务队列里拿任务进行处理。由于原来的逻辑会对递归调用的返回值进行处理，所以任务执行完成后，会对返回值进行处理。同时调度器线程会周期性运行，根据系统资源使用情况，动态调整线程个数。调整的方式为向空闲线程池中放入调度任务，空闲线程池中线程拿到调度任务后会把自己加入工作线程，执行其任务队列里的任务。

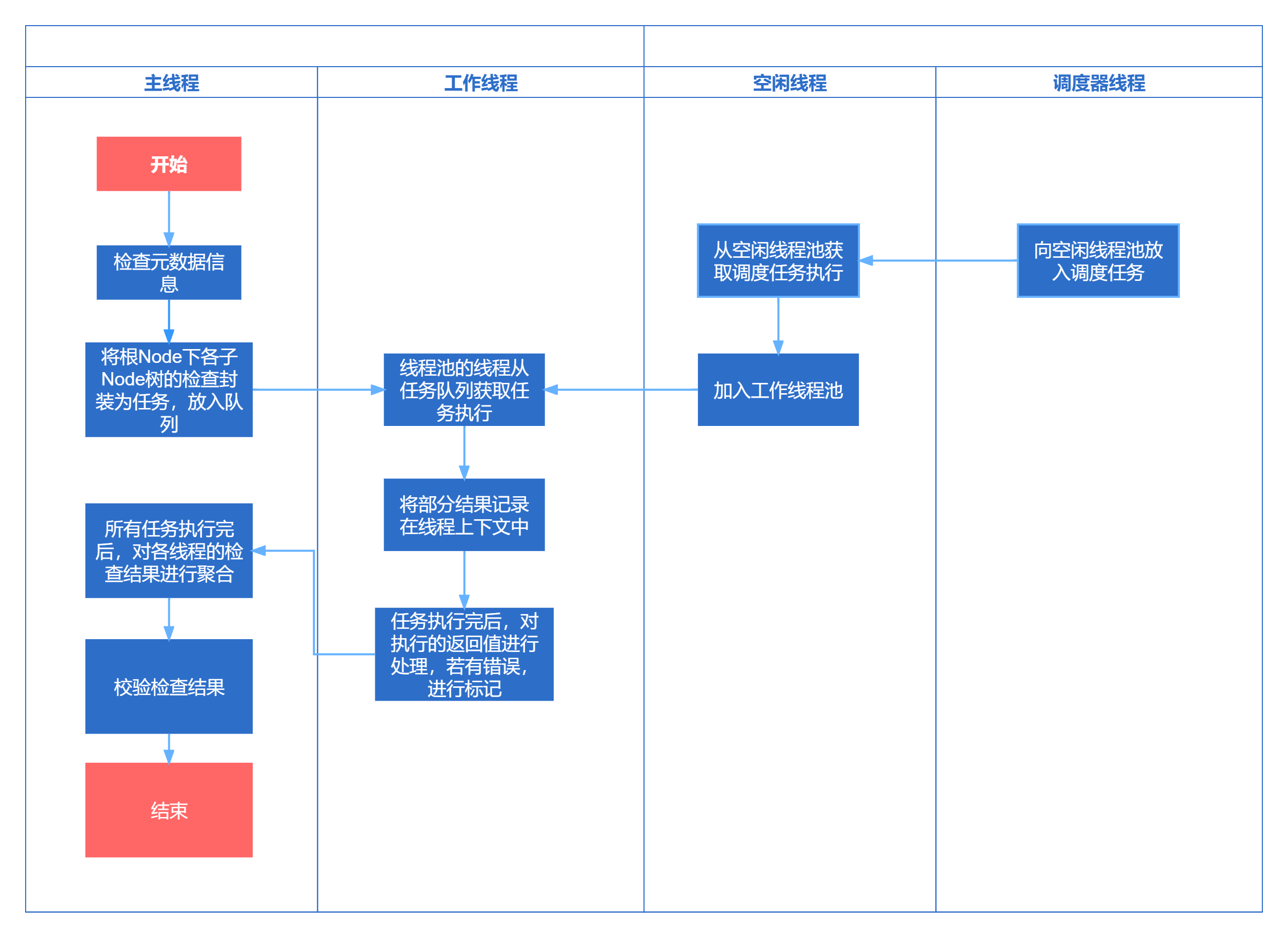


图 3‑2 系统整体运行流程

**(2) 优化f2fs的gc**

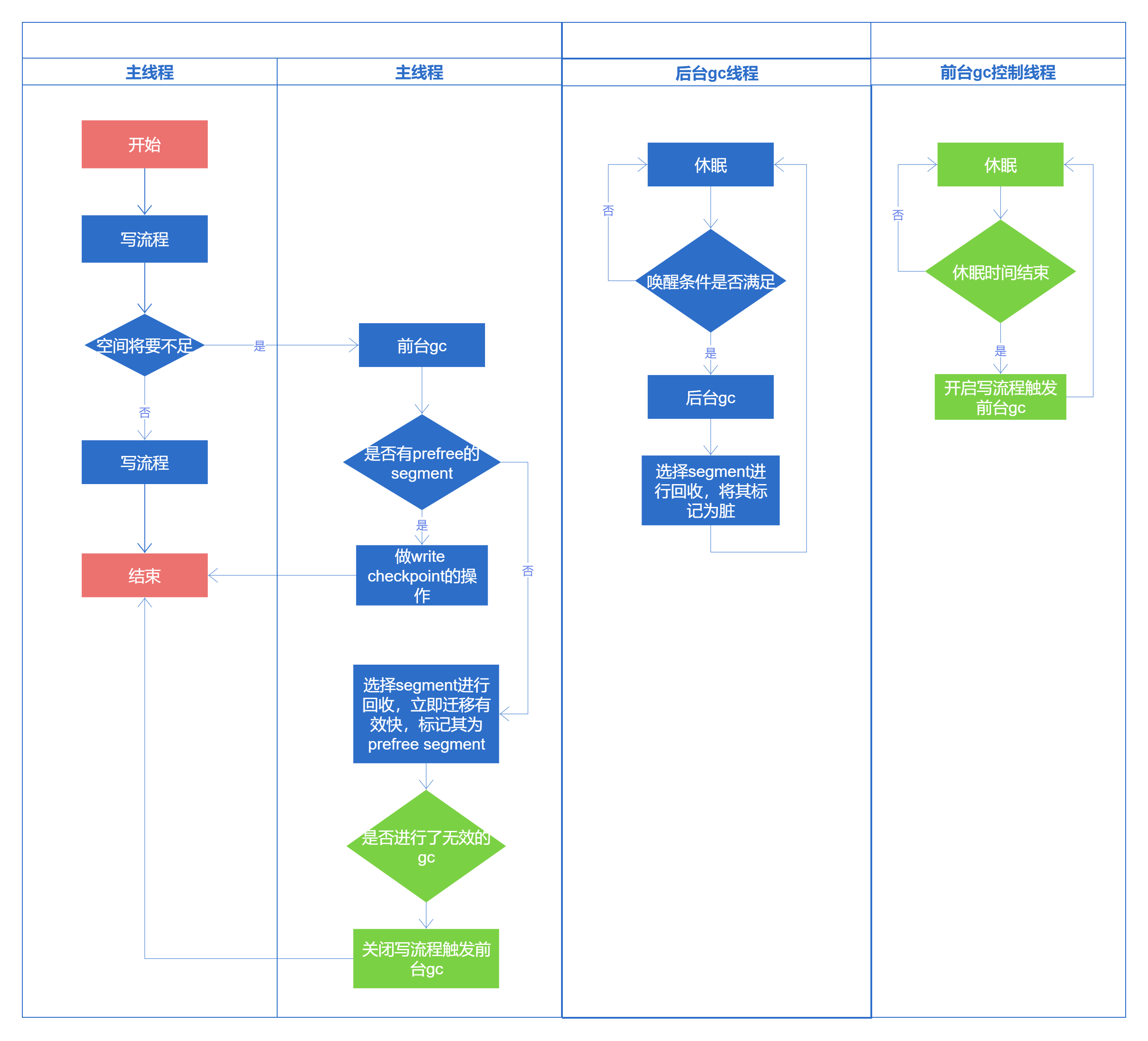


图 3‑3 优化的gc流程

目标2的系统整体运行流程如图3-3所示。图中蓝色部分是f2fs原本的逻辑。图中绿色的部分是我们优化增加的逻辑。图中只画出了一些关键的逻辑。原本的f2fs会在写流程中，判断是否空间将要不足，不足则触发前台gc。前台gc会阻塞写流程，使得写的性能下降。进入前台gc，会判断是否有prefree的segment。prefree的segment指的是在上一次前台gc回收的segment。若有，则进行write checkpoint做写回操作。使得上一次回收的segment真正可用。若没有prefree的segment，则会选择一个有效块最少的segment进行回收。因为是前台gc，所以这里的回收操作会立即迁移有效块，根据该segment的类型，将其中的有效块迁移到对应的segment中。

这其中存在的问题是在空间不够时，每次写流程都会调用gc的函数。但是gc可能是无效的。**无效又分为两种，一种是没有脏的segment可以被回收，相当于走了一遍gc的逻辑但实际并没有回收到空间。**这里的脏的segment指的是除了6个current segment之外，有效块数不为0，也不为512（一个segment共512个块）。在顺序写的情况下，是很可能出现没有脏的segment可供回收的情况的。而若是创建的文件比较多，在空间快不足时，每次创建都会调用gc，出现大量无效gc，白白浪费时间。**另一种是待回收的segment中的有效块非常多。**比如待回收的segment有510个有效块，迁移这些有效块到对应类型的current segment很可能导致current segment有效块被占满，从而需要一个新的空闲segment作为current segment。也就是说，虽然回收了一个segment，但也因为迁移操作，使用了新的segment，所以可用的segment并没有增加。而且因为有迁移操作，这种gc，浪费的时间更多。

**我们增加的优化逻辑**则是在发现做了无效gc后，则停止在写流程中触发前台gc。同时新增了一个gc控制线程，定时苏醒，开启在写流程中触发前台gc的逻辑。

**（3）利用fsck收集的信息**

我们在1.1.3节介绍了利用fskc收集信息的三大难点，我们目前的工作侧重于解决难点一和难点二，即如何实现用户态和内核态的通信，以及fsck收集的信息如何协助f2fs文件系统进行挂载加速。系统的运行流程如图3-4所示：

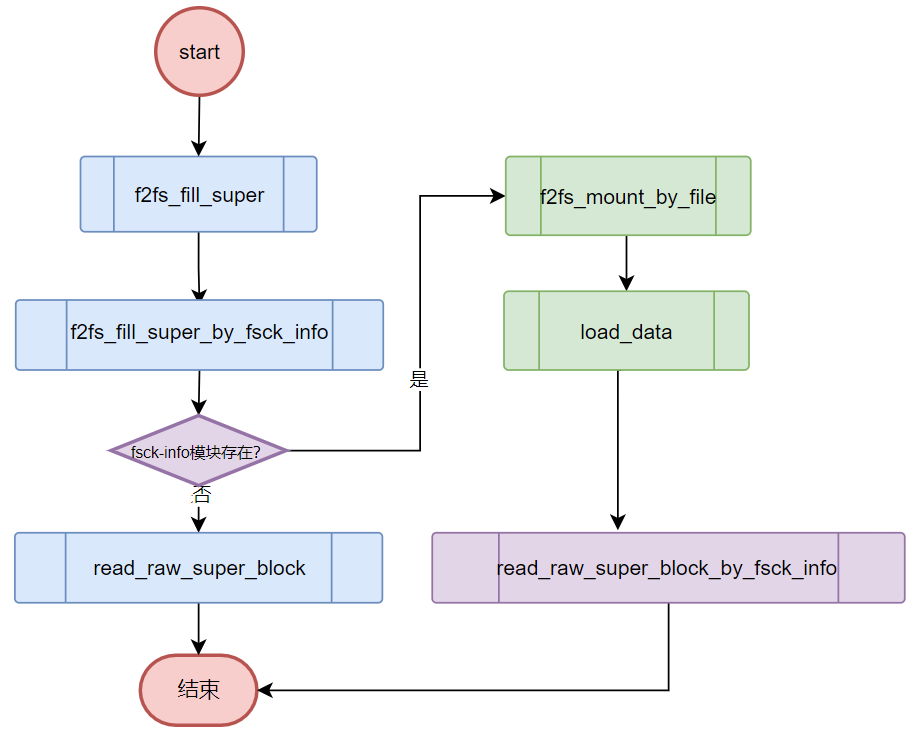


图 3‑4 利用fsck信息的流程

图中可以分为两部分，左半部分和右半部分。左半部分是f2fs文件系统挂载时读取super block的简易流程，右半部分是通过fsck提供的信息，不需读取硬盘，直接获取super block信息的流程。绿色部分是模块相应的内容，蓝色和紫色部分是f2fs文件系统相关的内容，其中蓝色部分是f2fs文件系统原有的内容，而紫色部分是为了使用fsck相关信息在f2fs中引入的相关代码。

下面详细介绍一下f2fs文件系统挂载时，如何利用fsck信息的流程。f2fs文件系统挂载时，会调用f2fs\_fill\_super函数，在此函数中，会调用f2fs\_fill\_super\_by\_fsck\_info来检查是否能够通过fsck\_info的信息来实现快速挂载。当我们的内核加载了fsck-info.ko类似的模块时，我们就能调用f2fs\_mount\_by\_file从而加载fsck-info提供的数据，并在后续读取super block的过程中避免再次从硬盘中获取，而直接从fsck-info提供的数据中获取。

上面介绍了f2fs文件系统如何利用fsck提供的信息来快速挂载，但是忽略了fsck提供的信息是如何变成一个模块供底层文件系统使用的细节，这部分将在模块设计和系统实现中详细介绍。

## 3.2 子模块设计

### 3.2.1 加速fsck.f2fs

本小节将介绍各子模块的设计，包括任务设计，线程池设计，线程上下文和锁设计。分别对应了检查的不同步骤，①将任务划分出来加入线程池/任务返回值处理。②线程从线程池里取出任务执行。③在执行过程中需要加锁访问特殊变量或者访问线程的上下文。其中各模块的具体实现放在第4节中进行讲解。

**任务设计**

本小节介绍每个线程执行的任务是如何划分，如何既高效又能保证正确性。

如前文所说，任务的划分是一个难点。首先fsck.f2fs的检查逻辑是从根Node开始递归地对整个Node树进行检查，并且每个Node都要对子Node的返回值进行处理。若只是将每个Node都作为一个任务加入线程池，那么任务的返回值将不好处理。比如Node需要检查完其所有子Node后将有效子Node个数和父Node中记录的个数进行比对。所以不能随意将每个Node都作为一个任务加入到任务队列中。

对此，我们将root Node下一级的子Node，也就是图3-5中绿色的Node作为任务添加到任务队列。注意，仅仅是下一级，不包括图3-5中红色的Node。这样这些子Node对返回值的处理就可以复用原始的代码，不用修改。接下来只需要考虑如何在root Node中对所有子Node的返回值进行处理。对此，我们将每个任务执行前的一些上下文保存下来，在任务结束后，使用这些上下文，对任务的返回值进行处理。

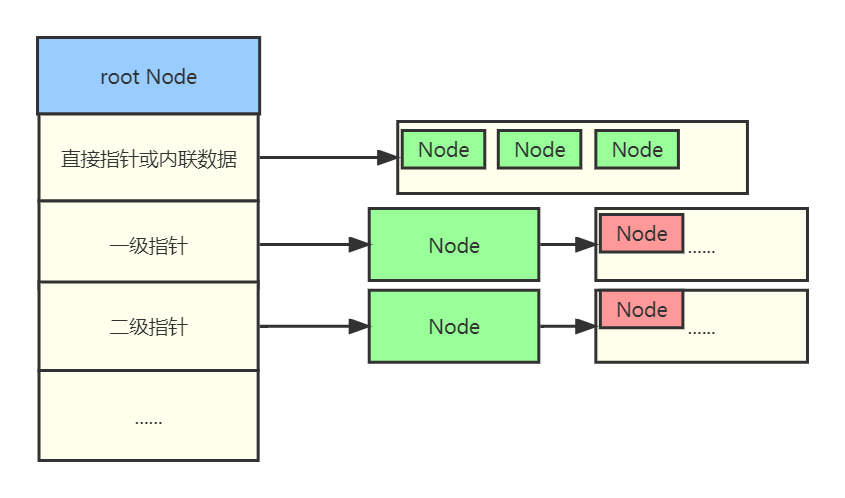


图 3‑5 任务划分示意图

**线程池与资源动态调整设计**

本小节介绍线程池是如何设计，线程数量是如何动态调整的。

线程池设计如图3-6所示，包含两个线程池。一个是工作线程池，该线程池里的线程负责执行检查任务。工作线程池里的任务队列会包含检查或调度任务，空闲线程池里的任务队列在大部分情况下是空的，这样空闲线程池里的线程获取不到任务，就会让出CPU。有时空闲线程池里会有调度任务，线程获取到调度任务就是调度自己到其他线程池。线程池里的初始线程个数可通过运行时参数指定。同时调度器线程会周期地运行，通过获取系统的资源使用情况结合当前检查工具的资源使用情况，动态调整线程个数。 这个动态调整就是往线程池的任务队列里放一个调度任务，拿到这个任务的线程就会将自己转移到另一个线程池中去。比如图所示，当前工作线程池有两个线程在不断从任务队列获取任务执行。而此时调度器线程休眠的时间结束又到了工作的周期，它监控系统资源情况，发现系统的CPU利用率并不高，选择从空线线程池中转移线程3到工作线程池。调度器线程就会向空闲线程池的任务队列添加一个调度任务。该调度任务包含了，任务类型，标识其为一个调度任务，标识了目标线程池，当线程3拿到这个调度任务时，就会将自己所属的线程池改为工作线程池。因为线程是在一个死循环里不断获取自己所属的线程池任务队列里的任务，所以当下一次循环开始时，线程3获取的就是工作线程池的任务队列里的任务，这样线程3就从空闲线程池转移到了工作线程池。当然，空闲线程池中不一定是线程3获取到调度任务，也可能是线程4获取到调度任务，无论如何，只会有一个线程获取到调度任务，从而转移到工作线程池中。

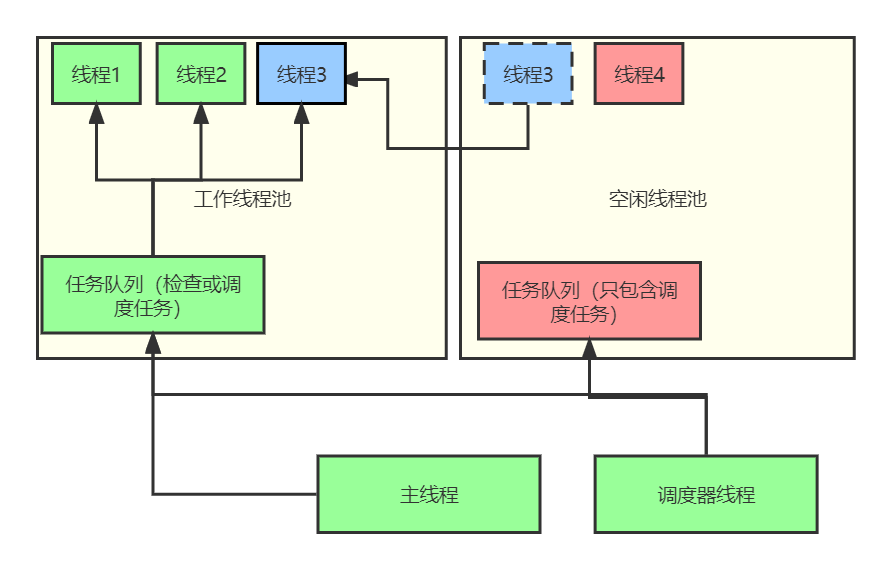


图 3‑6 线程池设计

**共享变量访问控制**

本小节介绍并发情况下对共享变量访问控制，使用的手段包括线程上下文和锁。

并发情况下，为了保证C/R的正确性，重新设计数据结构，解耦数据是一个关键问题。根据测试，如果只是简单对所有共享数据结构的访问加一把大锁，那么对共享数据结构的串行化访问会成为系统的瓶颈，执行的时间反而增加。对此，我们的优化是细化锁的粒度和设计线程上下文。如图3-7所示，变量A本来是共享数据结构，但在并发环境下对其的访问需要加锁。变量B本来是一个共享数据结构，但是每个线程的上下文中都有一份私有的变量B，线程在对该变量B进行读写时，是对该线程上下文的私有变量B进行读写。主线程会在所有任务执行完成后，将各个线程的私有变量进行聚合，比如变量B是一个计数值，主线程会将所有线程的私有变量B累加到共享数据结构B中。这些变量记录了检查结果，后续主线程会使用该结果进行最后的校验。同时，各个线程对磁盘的读写也要加锁处理，保证正确性。

各个变量不同处理分为以下类型：

* + 只写的数据：

放入线程上下文。各线程可以将该变量记录在自己的线程上下文，所有任务执行完后，由主线程聚合结果。

* + 只读的数据：

未额外处理。因为各线程只对该变量进行读操作，不影响C/R的结果。

* + 又读又写的数据：

加锁处理。因为该变量被修改后，又可能被其他线程读到，所以还是作为共享数据结构，只是访问要加锁。这类变量大多是bitmap位图之类的数据结构。

* + 只有放入线程上下文才能保证正确性的数据：

放入线程上下文。比如目录项的链表，只与某个Node为根的Node树有关。在检查时线程会不断向这个链表加入或删除结点，检查结束后，这个链表为空。从变量的含义上看，并发环境下，这个变量需要作为每个变量的私有数据，放入线程上下文，才能保证逻辑正确。

* + 部分线程共享的数据：

这种数据的典型是一个整型变量blk\_cnt，它是fsck\_node\_blk等函数中的一个参数，记录了当前node节点占用的block的个数，包括其元数据所占的块和其指向的数据块。对于一个inode类型的node来说，其值最终要等于f2fs\_inode结构体中的i\_blocks属性的值。我们将根目录的文件和文件夹拆分成任务，这些目录项的信息保存在根目录的数据块中。如果根目录的文件或文件夹数量过多，将会用到dnode，甚至是idnode来指向更多的数据块，来引用更多的目录项。而dnone、idnode等，他们也会被拆分成任务，而他们的blk\_cnt是属于根目录占用的block，要和根目录的blk\_cnt累加起来才等于根目录inode所记录的i\_blocks，这些线程是共享一个blk\_cnt的，而其他的线程对此变量是独立的。

解决办法有两种，一是将部分线程共享的数据作为当做全部线程共享来处理，通过一把全局锁解决，但是这会降低程序的性能。我们采用第二种方法，利用post\_work巧妙地解决这个问题。由于每个线程调用结束后会调用对应的post\_work函数进行处理，我们先采用线程上下文的思想，每个共享blk\_cnt的线程有独立的blk\_cnt，而后结束时，在post\_work中加锁汇总最终的结果。这和上述每个线程独立的数据有所区别，前者是为了统计根目录占用blk\_cnt的个数，从而来验证根目录的一致性，工作线程很有可能没有全部停止；而后者是为了汇总数据，比如文件系统有效inode个数等，来验证文件系统的一致性，所有的工作线程都停止了。

* + 磁盘的数据：

磁盘上的数据的访问也要加锁处理，否则会出现错误。

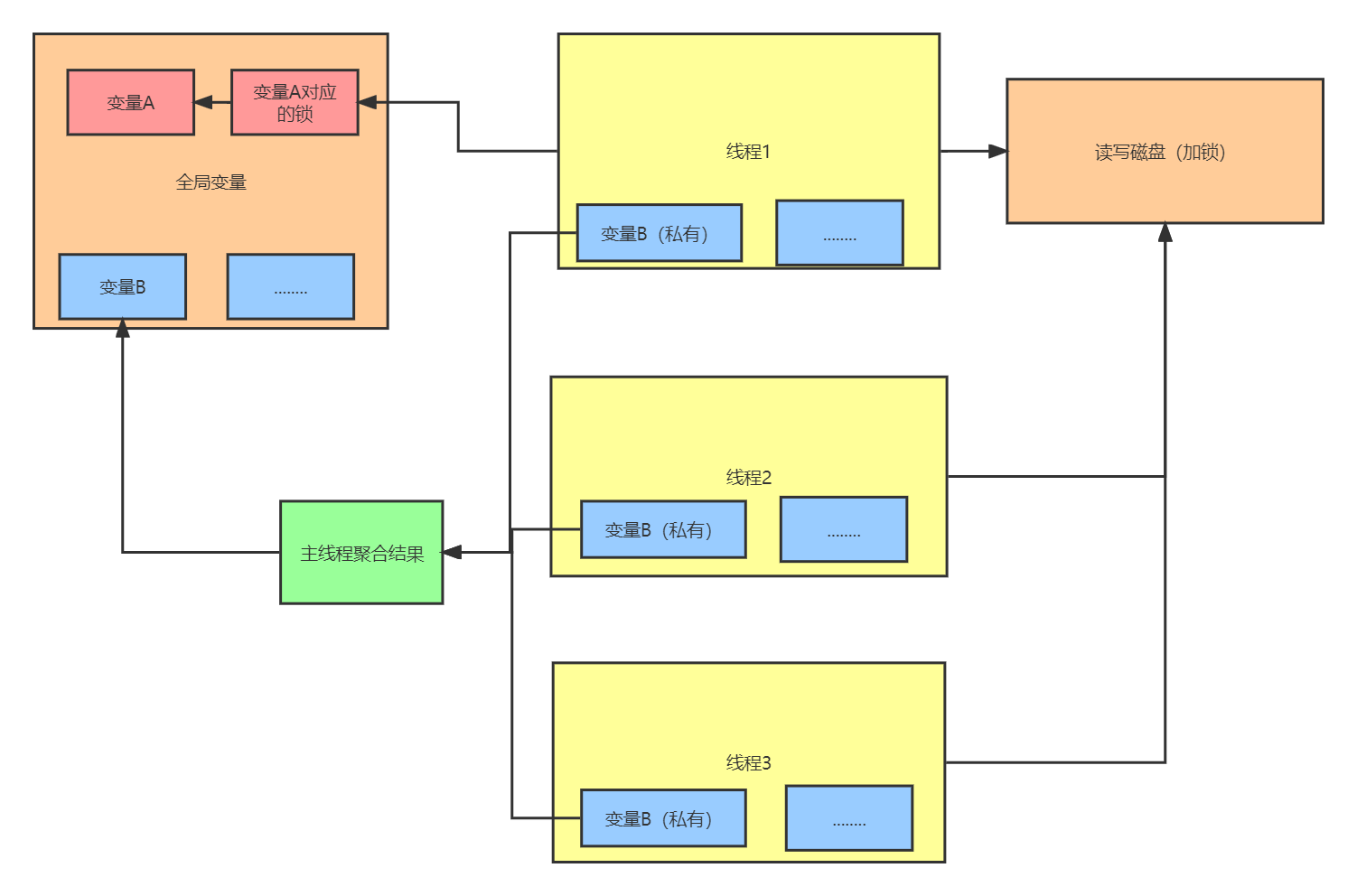


图 3‑7 共享变量访问控制

### 3.2.2 优化f2fs的gc

**f2fs的gc存在的问题分析**

在3.1.2节中，已经初步分析了f2fs gc中存在的问题。本节将进一步分析，在什么时候下会出现大量待回收segment中的有效块非常多的情况。

**（1）实验设置**

我们通过实验发现了这一场景，即待回收的segment中含有的有效块数量非常多。我们的实验设置如下：使用fs\_mark程序往一块装有f2fs文件系统的100M模拟设备不断写数据，每轮写256个文件，文件的大小为2k，即可通过内联数据将数据保存在一个Inode中。每轮写完毕后，我们将Main area区域中segment的位图信息以及对应的有效块分配情况和curseg的块号打印出来，以此来判定写这256个文件后segment以及block的分配情况。为了方便，表3-1给出了我们的实验配置。

表3-1 实验配置

|  |  |  |  |
| --- | --- | --- | --- |
| **场景** | **环境** | **参数设置** | **输出** |
| 往B大小的空f2fs文件系统写数据，每轮写N个大小为S文件，直至写满。每轮打印出位图分配情况 | linux\_kernel: 5.0.0  mkfs.f2fs: 1.16.0 | B = 100M  N = 256  S = 2k | segment\_bits：各个segment中每个block是否有效  segment\_info：各个segment中含有的有效block个数  curseg\_info：每种类型的logging对应的当前segment的块号（segno）和分配模式（alloc\_type） |

**（2）发现问题**

由于文件inode对应的curseg的类型为4，因此我们只关心类型4的segment的分配行为。我们的发现如下：在硬盘剩余空间较大时，4号segemnt分配block的方式在我们的意料之中，刚开始块分配采用LFS模式，后面空闲块不够且存在dirty segment时，变为SSR模式，最后没有脏segment了，又转化为LFS模式。令我们感到惊讶是在硬盘接近写满的时候，segment的分配跳了两个block，即貌似有两个block没有分配，如图3-8所示，显示了添加256个文件前后segment的分配变化情况。查看相应segment 中block的位图，如图3-9所示，和前面不同的是，此segment的分配跳了两个block，即刚开始的两个block被直接“跳过”了。

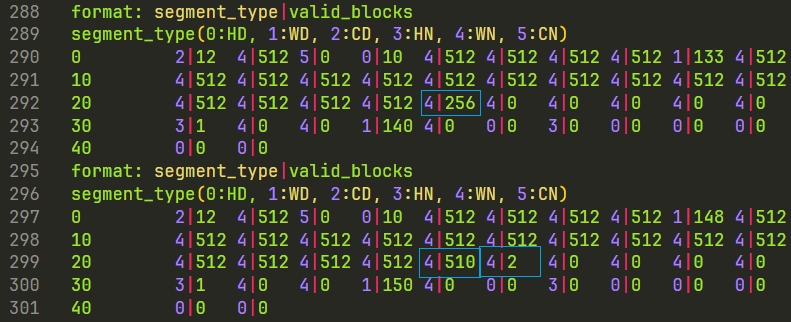


图 3‑8 segment分配情况



图 3‑9 segment中block是否有效的位图，0代表无效，1代表有效，用16进制表示

正是因为这个原因，后面会频繁地触发无用的gc（useless gc），换句话说，进行gc时，一个含有510个有效块的脏segment被选中，其中的510个有效块进行迁移。而后，又出现含有510个有效块的脏segment被选中的情况，再次进行迁移。这种频繁的无用gc，是造成用户读写性能下降的关键原因。我们的当务之急就是找出这种现象出现的原因，从而避免出现510个有效块的情况，来避免无用gc。

**（3）找到原因**

为了找到为什么会出现待回收segment中的有效块非常多的情况，我们对实验进行了多次重复，发现这种现象不是偶然，每次的实验仅仅只是segment分配的位置有些许不同，出现含有510个有效块的现象是不变的，出现的时机也是一致的，这使得我们能够调试找到问题所在。因此，我们通过gdb和qemu对内核进行调试，我们首先获取了类型为4对应的curseg所在的地址，然后监听此位图中出现空洞的那个字节（位图是由一个个位组成，图3-10给出了一个示例，我们监听的就是第一个字节3f，二进制表示为0011 1111，前两个位出现了空洞）中01变化的过程，我们有了如下发现，block的分配并不是跳了两格，而是首先分配了，后来被清除了。如图3-11所示，我们以第一个bit为例，我们监听到此bit发生了改变，其调用路径为f2fs\_create -> f2fs\_gc -> f2fs\_write\_checkpoint；然后在图3-11所示的那样，通过f2fs\_sync\_fs -> f2fs\_checkpoint，又清空了这个bit。图3-10和图3-11都是在同一个轮次进行的，这就造成了segment中分配block时好像有略过2个block的状况。

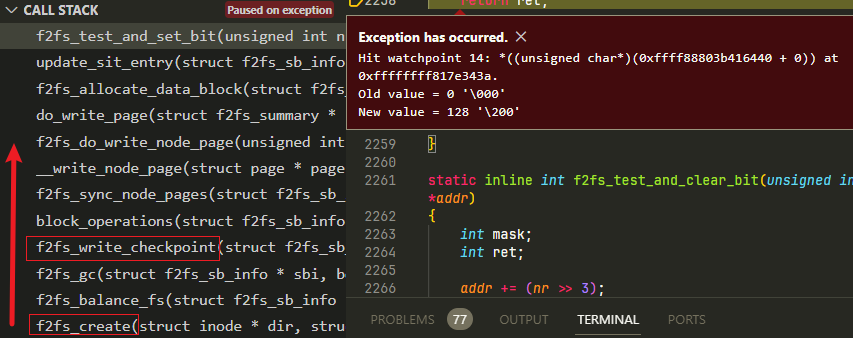


图 3‑10 设置bit时的调用栈

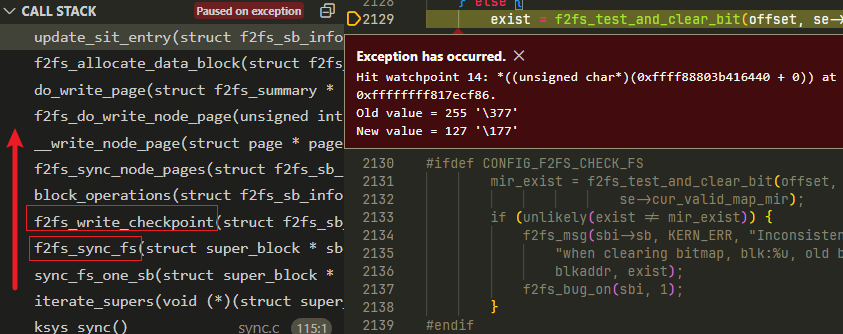


图 3‑11 清除bit时的调用栈

在我们的fs\_mark程序中，对一个文件写2K的数据，过程为create、write、sync，即先创建文件，然后往文件里面写数据，最后将数据同步到硬盘。当硬盘空间充足时，调用sync函数时，系统才会进行一次checkpoint操作；而硬盘空间不足时，如图3-10所示，create后，会进行是否进行前台gc的检查，这时符合条件进入到gc的流程，即f2fs\_gc函数，在里面会判断有没有prefree的segment，有的话会直接进行一次checkpoint操作来将prefree的segment加入到空闲的segment链表。这时带来了一个副作用，刚刚创建的文件立马就被刷新回了Flash中，后面更新文件，再做sync操作的时候，由于f2fs异地更新的特性，又把之前分配的块标记为无效，这就造成了一个segment出现空洞的情况，而这种空洞将会导致随后的gc对segment中的block进行频繁地迁移，造成系统性能急剧下降。

### 3.2.3 利用fsck收集的信息

本小节详细介绍f2fs文件系统利用fsck收集的信息的模块设计及其子模块的对应职责。实现该功能的系统架构如图3-12所示，除了最上面的f2fs Flash设备外，下面分为了3大块，分别是fsck.f2fs，module generator以及linux，他们的功能分别是：获取信息，编码信息，使用信息。中间的模块搭建起了用户态fsck.f2fs和内核态linux之间通信的桥梁。通过该图也能知道，fsck.f2fs和f2fs文件系统都需要在硬盘中读取信息，如果能够利用fsck.f2fs读取硬盘后获得的信息帮助f2fs文件系统进行挂载或优化读写行为，将会大大提高系统整体运行效率。图中还用不同颜色标记出了各个字块的行为，红色代表此模块对应具体的功能文件，如左边的fsck\_info.c是一个具体文件，其将fsck的信息导出；蓝色代表此模块是自动生成的，是在整个工作流过程中的中间产物，可以放心删除，如fsck\_info\_dump就是保存相应信息的中间文件，可以安全删除后通过fsck\_info重新生成。接下来将介绍这三大块的具体功能。

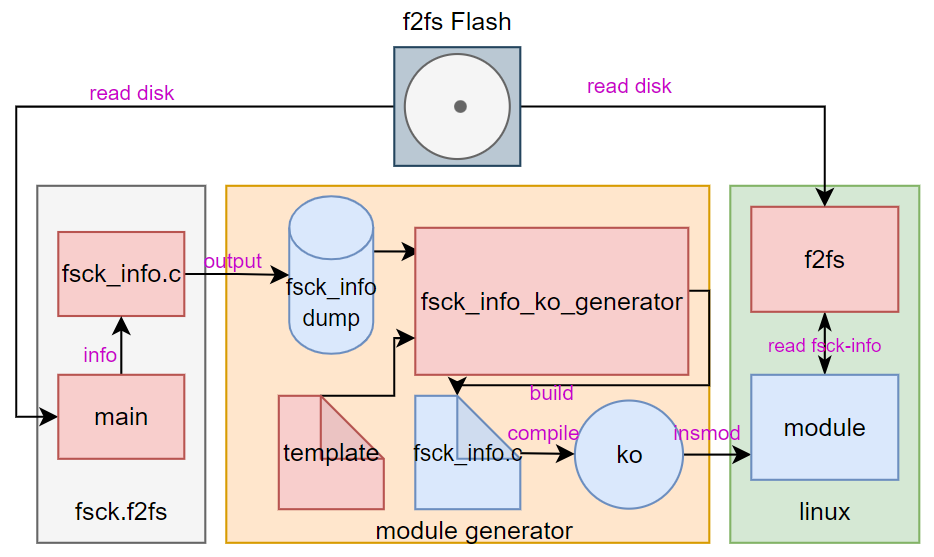


图 3‑12 利用f2fs信息系统架构示意图

* fsck.f2fs: 当文件系统检查完毕且没有出现错误时，用户应该能够选择是否将f2fs文件系统相关信息导出到一个地方，使得此信息能够协助进行f2fs文件系统的优化，这是通过main.c和fsck\_info.c实现的，main.c里面能够检查fsck的过程是否出现错误，如果没有，他将调用fsck\_info.c相应的导出函数，将super block info记录的相应信息导出，供下一阶段的模块生成器使用。
* module generator：template是一个模块文件的模板，往里面填入fsck\_info导出的信息后，即fsck\_info\_dump，就能够生成一个可被真正加载进内核的文件。这一过程是通过fsck\_info\_ko\_generator来实现的，其通过前面获取的fsck\_info信息和template模板文件，生成一个fsck\_info.c的模块文件，此模块文件经过编译并加载进内核后，相关信息就能被底层的f2fs文件系统使用。
* linux：在挂载f2fs文件系统的过程中，会去检查相应的模块文件是否存在，如果存在，就会采用此模块提供的信息进行挂载，否则，就走正常的挂载流程，即从硬盘中读取对应信息。

# 系统实现

## 4.1 加速fsck.f2fs

### 4.1.1核心数据结构

以下将给出重要数据结构的定义。

* **thread\_ctx**

thread\_ctx代表线程的私有数据，类似于线程上下文，用于保存该线程检查得到的结果。图4-1显示了其核心变量。



图 4‑1 thread\_ctx 核心变量

表4-1显示了各变量的含义。

表 4‑1 thread\_ctx核心变量含义

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **tid** | int | 线程id |
| **checked\_node\_cnt** | u64 | 已检查的node数目 |
| **valid\_blk\_cnt** | u64 | 有效的block数目 |
| **valid\_node\_cnt** | u32 | 有效的node block数目 | |
| **valid\_inode\_cnt** | u32 | 有效的node block数目 | |
| **valid\_blk\_cnt** | u64 | 有效的block数目 |
| **valid\_node\_cnt** | u32 | 有效的node block数目 |
| **valid\_inode\_cnt** | u32 | 有效的node block数目 | |
| **multi\_hard\_link\_files** | u32 | 有多个硬链接的文件数目 | |

续表 4-1 thread\_ctx核心变量含义

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **dentry\_depth** | u32 | 目录项深度 | |
| **dentry** | struct f2fs\_dentry \* | 目录项链表 |
| **dentry\_end** | struct f2fs\_dentry \* | 目录项链表表尾 | |

* **f2fs\_fsck**

f2fs\_fsck代表检查过程的全局数据。图4-2显示了其核心变量。

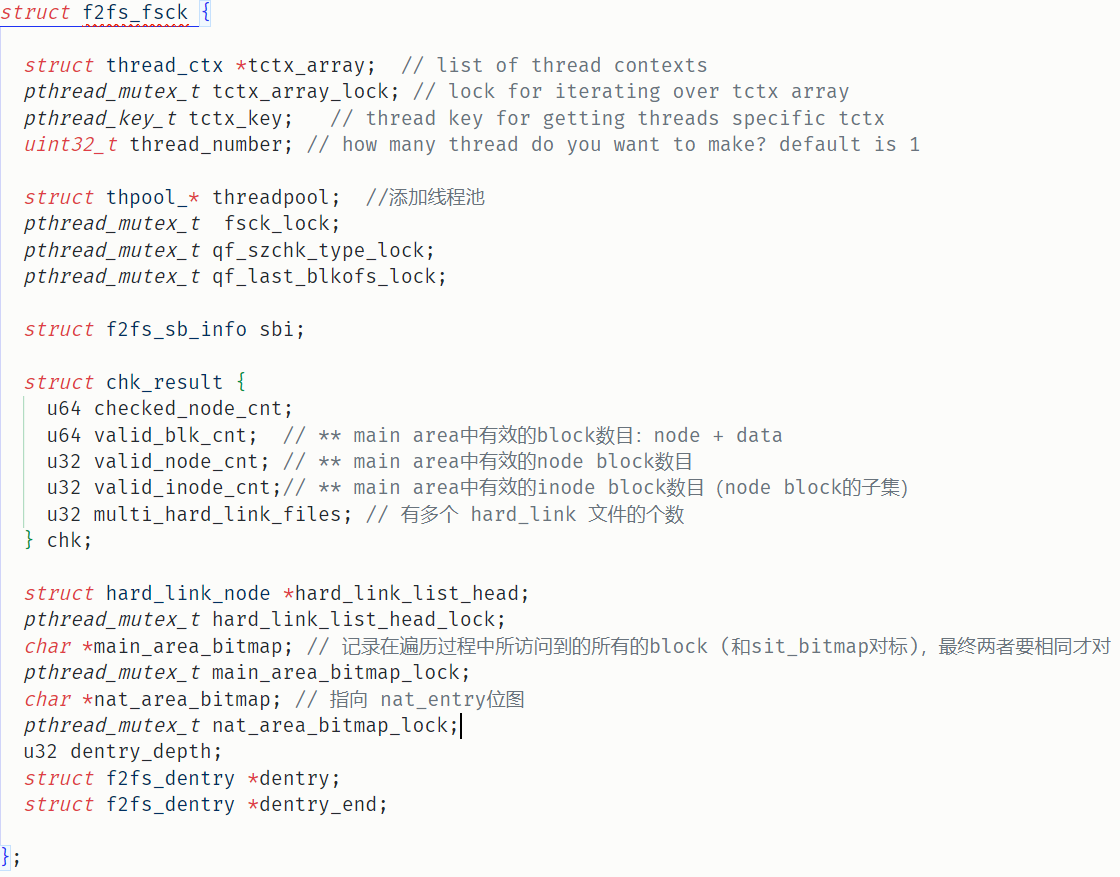


图 4‑2 f2fs\_fsck 核心变量

表4-2显示了各变量的含义。

表 4‑2 f2fs\_fsck核心变量及描述

| **变量名** | **数据类型** | **变量描述** |
| --- | --- | --- |
| **tctx\_array** | thread\_ctx | 保存所有线程上下文 |
| **tctx\_array\_lock** | pthread\_mutex\_t | tctx\_array对应的锁 |
| **tctx\_key** | pthread\_key\_t | 用于获取线程上下文的key |
| **thread\_number** | uint32\_t | 线程池中线程数量 | |
| **threadpool** | thpool\_\* | 线程池 | |
| **qf\_szchk\_type\_lock** | pthread\_mutex\_t | qf\_szchk\_type对应的锁 |

续表4-2 f2fs\_fsck核心变量及描述

| **变量名** | **数据类型** | **变量描述** |
| --- | --- | --- |
| **qf\_last\_blkofs\_lock** | pthread\_mutex\_t | qf\_last\_blkofs对应的锁 |
| **sbi** | f2fs\_sb\_info | super block的信息 | |
| **chk** | chk\_result | 全局检查结果 | |
| **hard\_link\_list\_head** | hard\_link\_node \* | 硬链接链表 | |
| **hard\_link\_list\_head\_lock** | pthread\_mutex\_t | hard\_link\_list\_head对应  的锁 | |
| **main\_area\_bitmap** | char \* | main area的位图 | |
| **main\_area\_bitmap\_lock** | pthread\_mutex\_t | main area位图对应的锁 |
| **nat\_area\_bitmap** | char \* | nat area的位图 |
| **nat\_area\_bitmap\_lock** | pthread\_mutex\_t | nat area位图的锁 |
| **dentry\_depth** | u32 | 目录项的深度 | |
| **dentry** | f2fs\_dentry \* | 目录项链表 | |
| **dentry\_end** | f2fs\_dentry \* | 目录项链表尾 |

* **job**

job代表了一个任务。类型为0时为普通任务，即为对从某一Node开始递归对整个Node树进行检查。其核心变量如图4-3所示。

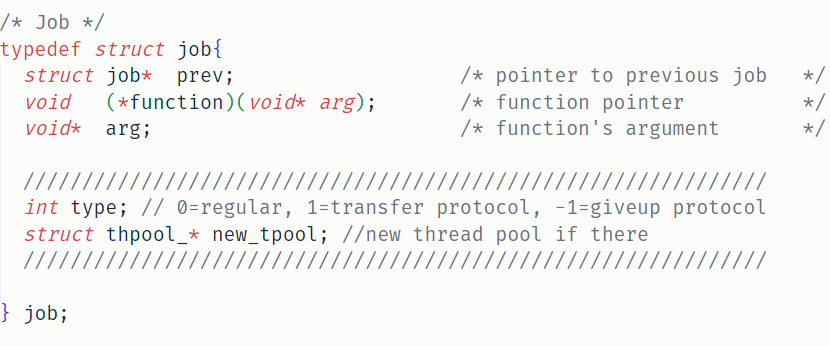


图 4‑3 job核心变量

表4-3显示了job各核心变量的含义。

表 4‑3 job核心变量

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **pre** | struct job\* | 指向前一个任务的指针； |
| **void   (\*function)(void\* arg)** | function | 任务对应要调用的函数； |
| **arg** | void\* | 函数参数； |
| **type** | int | 任务类型； |
| **new\_tpool** | struct thpool\_\* | 新的线程池，当任务为迁移线程时，将线程迁移到该目标线程池； |

* **jobqueue**

jobqueue为任务队列，每个线程池会有一个任务队列，线程池里的线程会不断从该任务队列取出任务执行。其核心变量如图4-4所示。

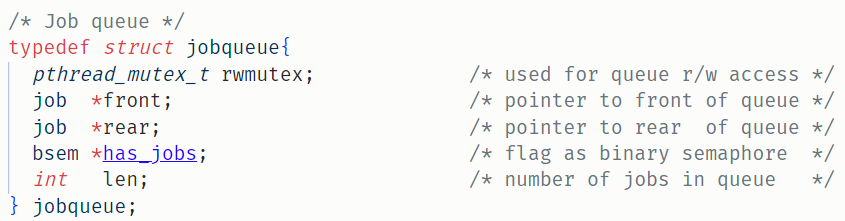


图 4‑4 jobqueue核心变量

表4-4显示了jobqueue各核心变量的含义。

表 4‑4 jobqueue核心变量

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **rwmutex** | pthread\_mutex\_t | 访问队列的锁； |
| **front** | job  \* | 队头的任务； |
| **rear** | job  \* | 队尾的任务； |
| **len** | int | 队列中的任务个数； | |

* **thread**

thread是对真正的线程进行了封装，记录了额外信息，如线程对应的线程池，是否借出，原来的线程池等。其核心变量如图4-5所示。

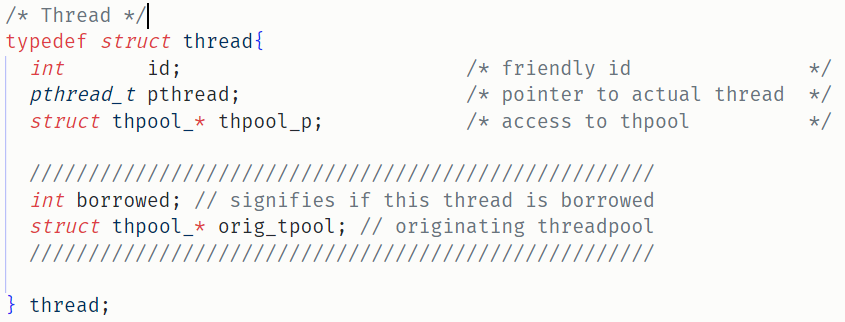


图 4‑5 thread核心变量

表4-5显示了thread各核心变量的含义。

表 4‑5 thread核心变量

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **id** | int | 线程id； |
| **pthread** | pthread\_t | 指向真正的线程； |
| **thpool\_p** | struct thpool\_\* | 线程所属的线程池； |
| **borrowed** | int | 线程是否借出； | |
| **orig\_tpool** | struct thpool\_\* | 线程原本的线程池 | |

### 4.1.2关键函数实现

* **get\_tctx**

1. **函数原型**

*struct* thread\_ctx\* get\_tctx(*struct* f2fs\_sb\_info \**sbi*)

1. **函数功能**

获取当前线程的上下文

1. **参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **sbi** | **struct f2fs\_sb\_info \*** | f2fs超级块信息 |

1. **返回值说明**

返回当前线程的上下文

1. **函数流程**

通过线程上下文的key获取当前线程上下文。若是第一次获取，还会将该上下文记录在全局上下文数组中。其流程图如图4-6所示。

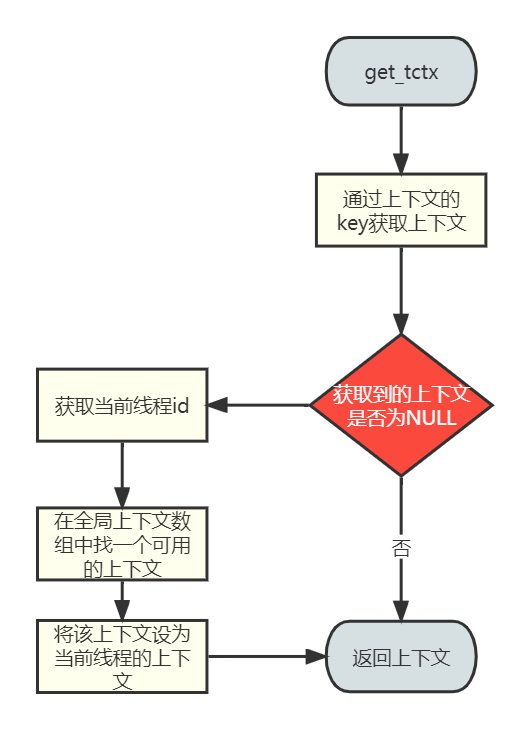


图 4‑6 get\_tctx流程图

* **thread\_do**

**① 函数原型**

static *void*\* thread\_do(*struct* thread\* *thread\_p*)

**② 函数功能**

这是线程池里的线程被创建后一直执行的函数。

**③ 参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **thread\_p** | **struct thread \*** | 线程 |

**④ 返回值说明**

无。

**⑤ 函数流程**

主要流程为在死循环里不断获取当前线程池里任务进行执行。如图4-7所示。

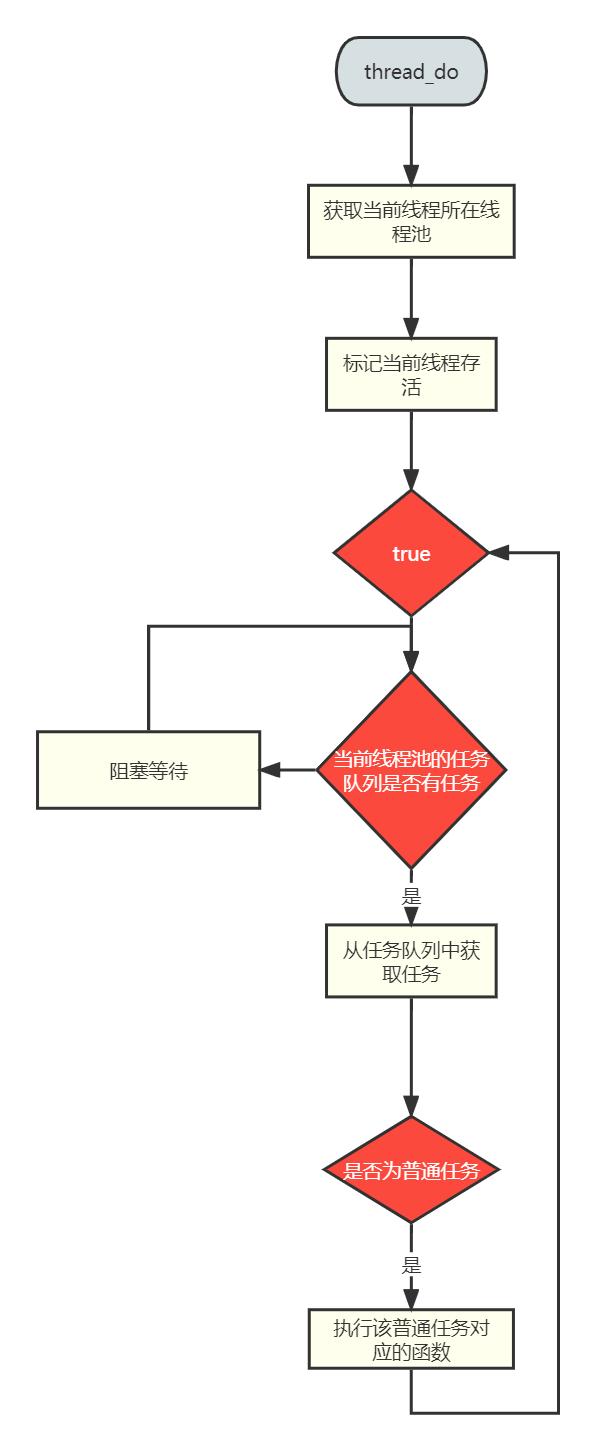
****

图 4‑7 thread\_do流程图

* **scheduler\_thread**

**① 函数原型**

*void* scheduler\_thread(*struct* f2fs\_sb\_info \**sbi*)

**② 函数功能**

调度器线程，周期运行，动态调整线程数目。

**③ 参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **sbi** | **struct f2fs\_sb\_info \*** | f2fs超级块信息 |

**④ 返回值说明**

无。

**⑤ 函数流程**

主要流程为在死循环里每隔一段时间调用schedule函数。如图4-8所示。

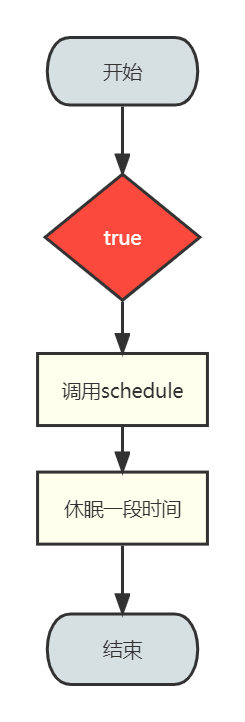


图 4‑8 scheduler\_thread流程图

* **schedule**

**① 函数原型**

*void* schedule(*struct* f2fs\_sb\_info \**sbi*)

**② 函数功能**

根据当前资源使用情况，动态调整线程个数。

**③ 参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **sbi** | **struct f2fs\_sb\_info \*** | f2fs超级块信息 |

**④ 返回值说明**

无。

**⑤ 函数流程**

主要流程为根据top命令获取系统整体CPU利用率和当前进程的CPU利用率，进一步算出线程数的预算，并向线程池添加“调整线程数”的任务。如图4-9所示。

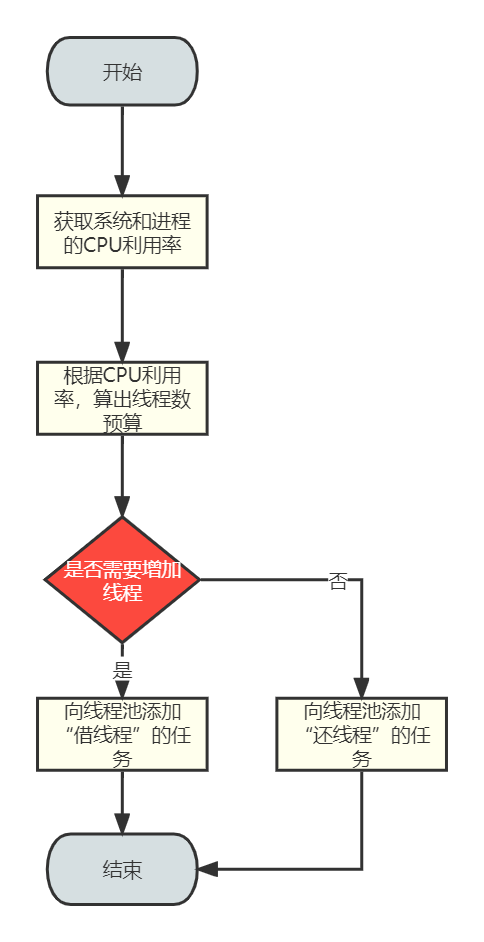


图 4‑9 schedule流程图

* **adjust\_core\_count**

**① 函数原型**

*int* adjust\_core\_count(*int* *cores*, *int* *prev\_cores*, *float* *total\_util*, *float* *process\_util*)

**② 函数功能**

根据当前资源使用情况，算出应该使用的线程数。

**③ 参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **sbi** | **struct f2fs\_sb\_info \*** | 超级块信息 |
| **cores** | **int** | 应调整到的线程数 |
| **prev\_cores** | **int** | 之前的线程数 |
| **float** | **total\_util** | 系统总CPU利用率 |
| **float** | **process\_util** | wFSCK进程的CPU利用率 |

**④ 返回值说明**

返回应调整到的线程数。

**⑤ 函数流程**

主要流程为根据系统CPU利用率高低和进程CPU利用率高低，决定增加或减少或不变动线程数。如图4.10所示。

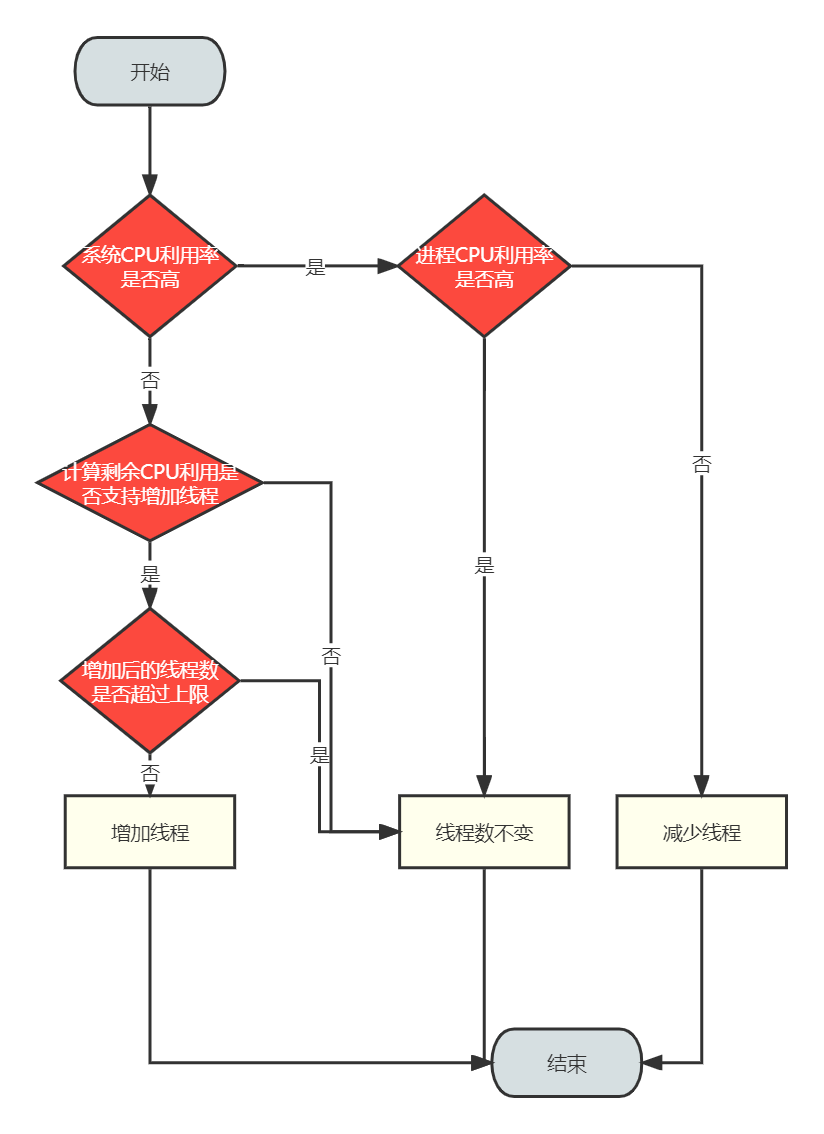


图 4‑10 adjust\_core\_count流程图

* **borrow\_threads**

**① 函数原型**

*int* borrow\_threads(thpool\_\* *thpool1*, thpool\_\* *thpool2*, *int* *n*)

**② 函数功能**

从线程池thpool1中借n个线程给线程池thpool2，执行检查任务。

**③ 参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **thpool1** | **thpool\_\*** | 线程池1 |
| **thpool2** | **thpool\_\*** | 线程池2 |
| **n** | **int** | 要借的线程数 |

**④ 返回值说明**

成功返回0，失败返回-1。

**⑤ 函数流程**

主要流程为向要借出线程的线程池1中添加一个任务，线程池1中的线程在拿到这个任务执行时，会把自己切换到线程池2中去。如图4.11所示。

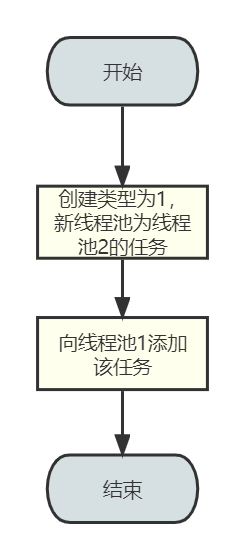
****

图 4‑11 borrow\_threads流程图

* **fsck\_increase\_valid\_inode\_cnt\_atomicly**

**① 函数原型**

*void* fsck\_increase\_valid\_inode\_cnt\_atomicly(*struct* f2fs\_sb\_info \**sbi*)

**② 函数功能**

取出当前线程上下文的某变量的值将其加一，

**③ 参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **sbi** | **struct f2fs\_sb\_info \*** | f2fs超级块信息 |

**④ 返回值说明**

无。

**⑤ 函数流程**

主要流程为取出当前线程上下文的某变量的值将其加一。类似的函数还有很多，这类函数封装了对线程私有数据的操作，用于方便地替换原始直接访问共享数据结构的代码。

* **post\_work\_inode**

**① 函数原型**

*void* post\_work\_inode(*int* *ret*, *void* \**args*)

**② 函数功能**

fsck\_chk\_inode函数里对间接block进行的检查完成后，对返回值进行处理。

**③ 参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **ret** | **int** | 检查的返回值，0代表成功，-EINVAL代表失败。 |
| **args** | **void \*** | 处理返回值时需要的上下文，在执行该检查任务前已保存好。 |

**④ 返回值说明**

无。

**⑤ 函数流程**

主要流程如图4-12所示。

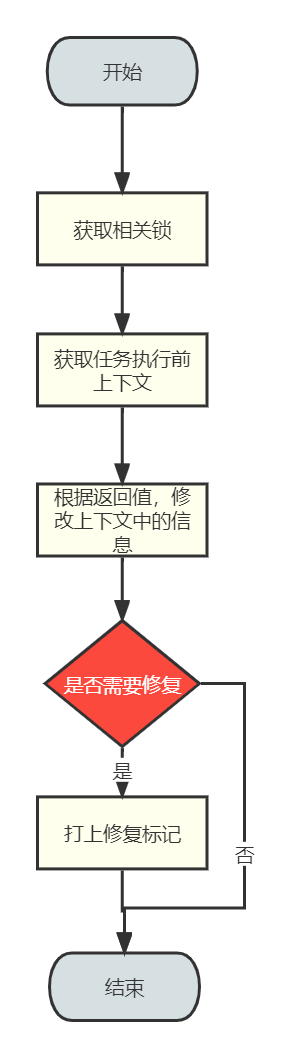
****

图 4‑12 post\_work\_inode流程图

## 4.2 优化f2fs的gc

### 4.2.1 核心数据结构

以下将给出重要数据结构的定义。

* **f2fs\_gc\_control\_thread**

f2fs\_gc\_control\_thread用于定时开启写流程中调用gc的逻辑，该线程是类似后台gc线程，不同的是该线程睡眠的时间是固定的，且唤醒后进行的操作只是把某个控制变量置为true。图4-13显示了其核心变量。

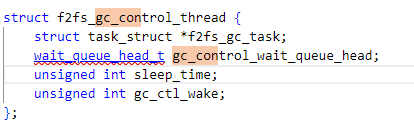
****

图 4‑13 thread\_ctx 核心变量

表4-6显示了各变量的含义。

表 4‑6 thread\_ctx核心变量含义

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| **f2fs\_gc\_task** | struct task\_struct \* | f2fs gc 任务 |
| **gc\_control\_wait\_queue\_head** | wait\_queue\_head\_t | gc控制线程要等待的队列 |
| **sleep\_time** | gc\_ctl\_wake | gc控制线程睡眠时间 |
| **gc\_ctl\_wake** | unsigned int | gc控制线程唤醒条件 |

### 4.2.2 关键函数实现

* **gc\_ctl\_thread\_func**

1. **函数原型**

static int gc\_ctl\_thread\_func(void \*data)

1. **函数功能**

控制gc线程，定时苏醒，将really\_can\_gc变量置为true。

1. **参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| data | void \* | f2fs超级块信息 |

1. **返回值说明   
   无。**
2. **函数流程**

初始化gc控制线程的信息，包括等待队列，睡眠时间等。在死循环中，不断睡眠，唤醒，唤醒时将really\_can\_gc置为true。其流程图如图4-14所示。

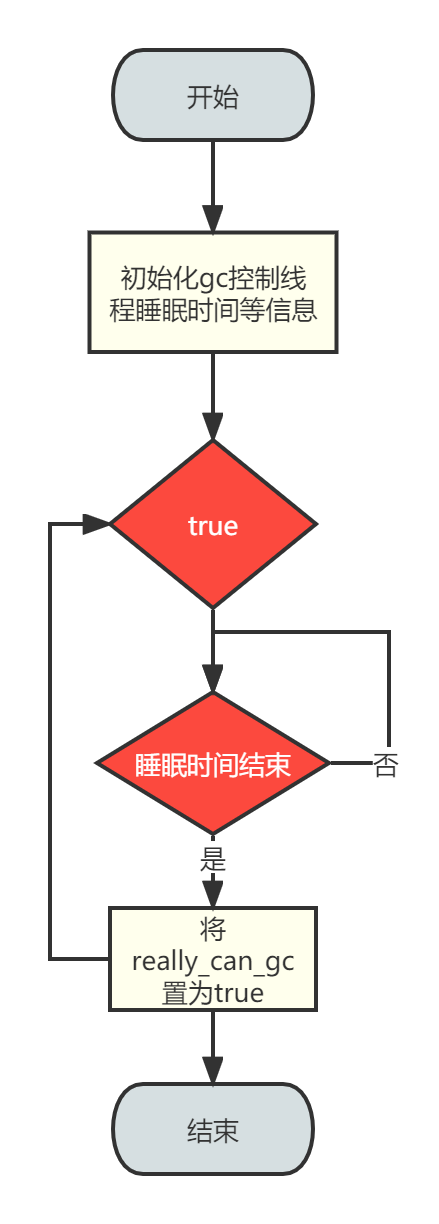


图 4‑14 gc\_ctl\_thread\_func流程图

* **f2fs\_gc**

1. **函数原型**

int f2fs\_gc(struct f2fs\_sb\_info \*sbi, bool sync,

            bool background, unsigned int segno)

1. **函数功能**

进行gc，回收segment。

1. **参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| sbi | struct f2fs\_sb\_info\* | f2fs超级块信息 |
| sync | bool | 是否sync |
| background | bool | 是否为后台gc |
| segno | unsigned int | segment号码 |

1. **返回值说明**0代表执行成功。
2. **函数流程**

空间不足时判断是否有prefree的segment，有则进行write checkpoint操作，让之前回收的segment真正可用，然后返回。若没有prefree的segment，则根据gc类型采取相应的算法挑选合适的segment进行回收。若没有选出待回收的segment，或者待回收的segment有效块非常多，则视为进行了无效gc。将really\_can\_gc变量置为false，停止写流程中触发前台gc。其中在回收segment时，若为前台gc则会立即迁移有效块，若为后台gc则会将segment标记为脏，等待后续写回。其流程图如图4-15所示。其中绿色部分是我们新增的逻辑。

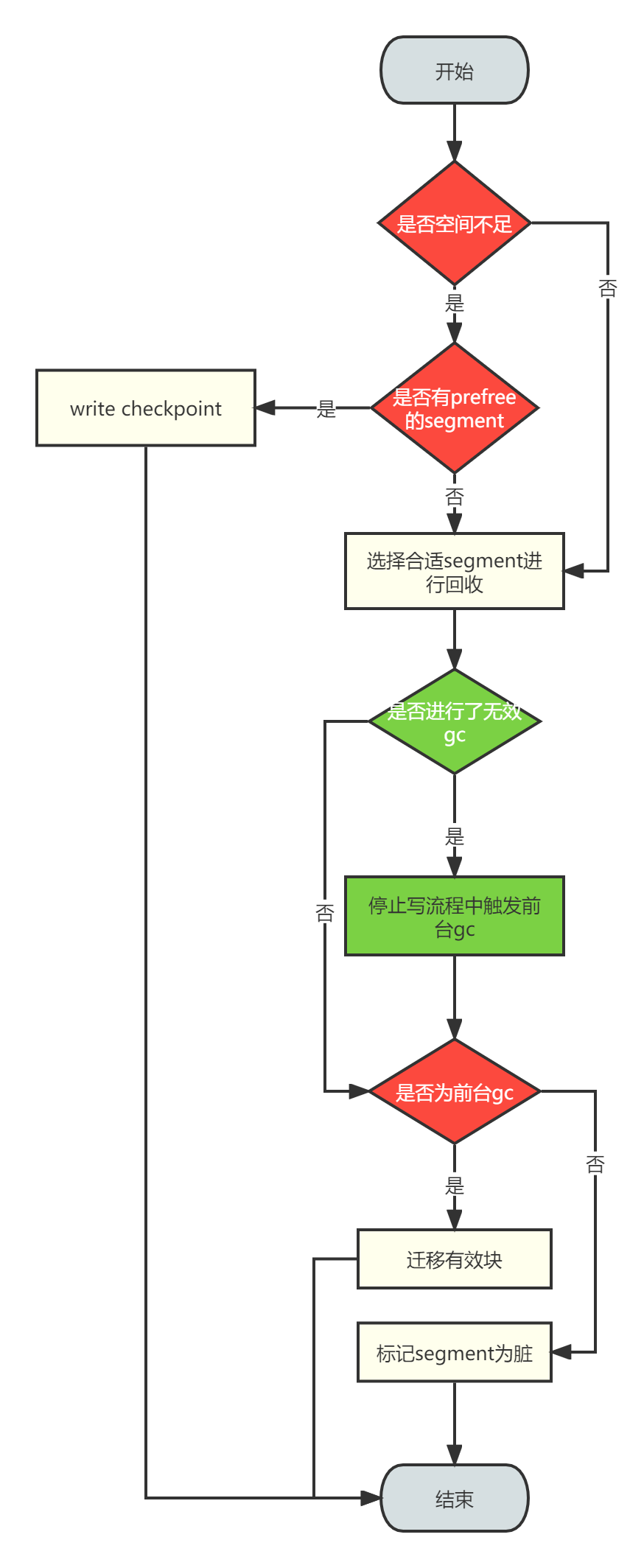


图 4‑15 gc\_ctl\_thread\_func流程图

* **f2fs\_balance\_fs**

1. **函数原型**

void f2fs\_balance\_fs(struct f2fs\_sb\_info \*sbi, bool need)

1. **函数功能**

写流程会调用此函数，此函数可能调用gc，即为触发前台gc。

1. **参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| sbi | f2fs\_sb\_info\* | f2fs超级块信息 |
| need | bool | 标志位 |

1. **返回值说明   
   无。**
2. **函数流程**

若空间不足，really\_can\_gc为true，则调用gc。其流程图如图4-16所示。其中绿色部分为我们增加的逻辑。

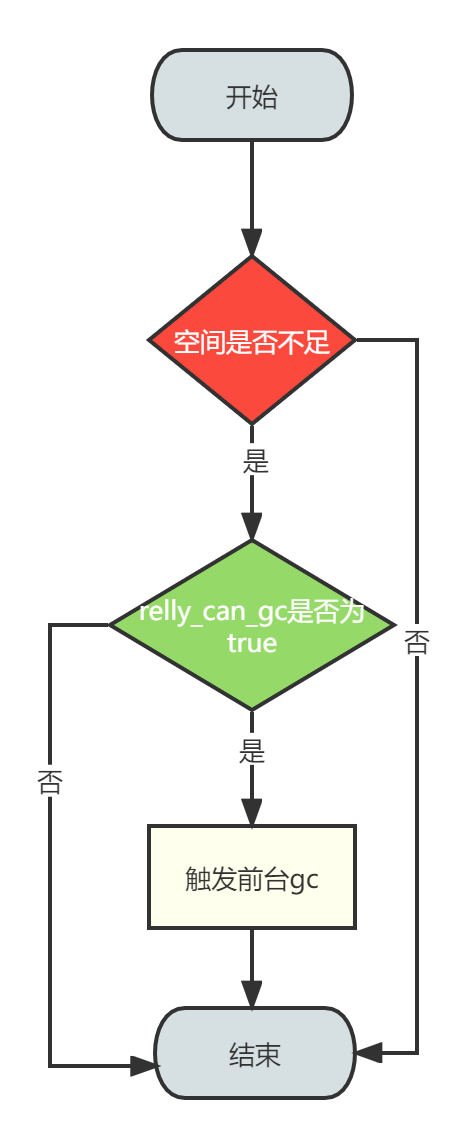


图 4‑16 f2fs\_balance\_fs流程图

## 4.3 利用fsck收集的信息

### 4.3.1 核心数据结构

以下将给出重要数据结构的定义

* **f2fs\_fsck\_info\_data**

如图4-17所示，f2fs\_fsck\_info\_data是一个字符数组，用于在模块和f2fs文件系统之间传递数据，其定义在模块的模板文件中，经过模块生成器解析，如图4-18所示，最终f2fs\_fsck\_info\_data会变成一个真实的字符数组，限于篇幅，没法展示完整的内容。数据通过模块加载到内存中，f2fs文件系统在挂载时会将此数据转化成有意义的格式，比如fsck和f2fs约定数据的内容为一个超级块的内容，则f2fs就会把数据当成是一个超级块，数据的格式非常灵活，因此也带来了危险，f2fs使用这类数据时，必须要验证数据的合法性和完整性。

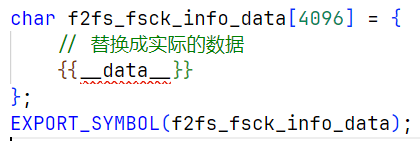


图 4‑17 f2fs\_fsck\_info\_data定义

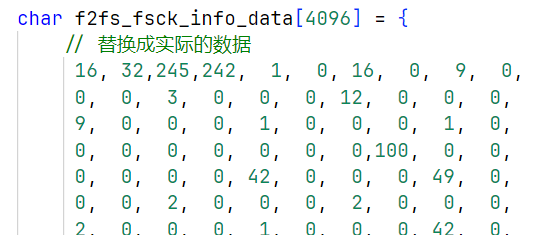


图 4‑18 f2fs\_fsck\_info\_data完成解析示意图

### 4.3.2 关键函数实现

* **dump\_fsck\_info**

1. **函数原型**

void dump\_fsck\_info(struct f2fs\_sb\_info \*sbi)

1. **函数功能**

将fsck获得的信息导出到文件，目前实现了将超级块内容导出

1. **参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| sbi | f2fs\_sb\_info\* | f2fs超级块信息 |

1. **返回值说明**

无。

1. **函数流程**

若文件系统的检查没有发现错误，main函数会调用dump\_fsck\_info，dump\_fsck\_info先打开dump文件，然后将super block结构体的数据通过fwrite函数输出，函数的流程图4-19所示。

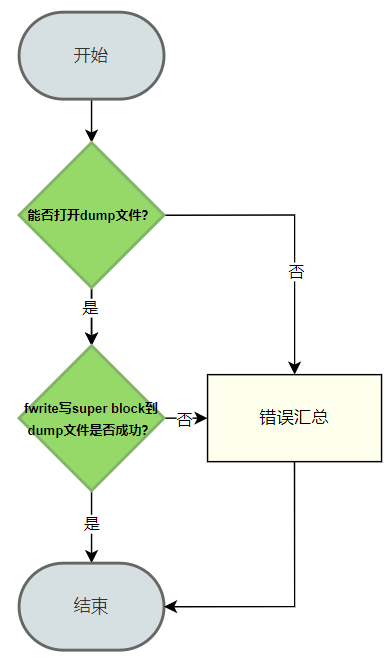


图 4‑19 dump\_fsck\_info流程图

* **do\_write\_module**

**① 函数原型**

void do\_write\_module(unsigned char \*data, size\_t len)

**② 函数功能**

将data指针指向的len字节长度的数据，依据模块模板文件，替换数据占位符，生成实际可编译运行的模块文件。

**③ 参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| data | unsigned char\* | 数据指针 |
| len | size\_t | 数据长度 |

**④ 返回值说明**

无。

**⑤ 函数流程**

dump\_fsck\_info函数生成的dump文件需要经过模块生成器的do\_write\_module解析，生成可被编译运行的模块文件。do\_write\_module函数的运行流程如图4-20所示。

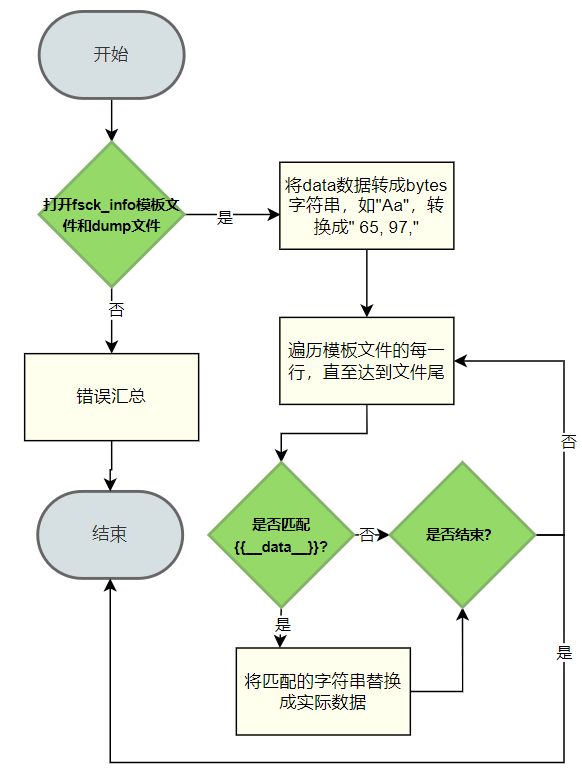


图 4‑20 do\_write\_module 函数流程

* **read\_raw\_super\_block\_by\_fsck\_info**

**① 函数原型**

static int read\_raw\_super\_block\_by\_fsck\_info(

struct f2fs\_sb\_info \*sbi,

            struct f2fs\_super\_block \*\*raw\_super,

            int \*valid\_super\_block,

int \*recovery,

struct f2fs\_super\_block \*data)

**② 函数功能**

读取一个超级块，用于替换原来的read\_raw\_super\_block函数。从data从读取超级块而不是从硬盘中读取，并验证此超级块的一致性。

**③ 参数说明**

|  |  |  |
| --- | --- | --- |
| **变量名** | **数据类型** | **变量描述** |
| sbi | struct f2fs\_sb\_info\* | 记录f2fs全局信息 |
| raw\_super | struct f2fs\_super\_block \*\* | 若一切正常，返回时内容为硬盘中超级块内容 |
| valid\_super\_block | int\* | 指示当前读取的是哪个超级块，f2fs对超级块进行了备份，防止超级块被损坏 |
| recovery | int\* | 指示是否需要修复 |
| data | struct f2fs\_super\_block\* | fsck-info模块提供的超级块内容 |

**④ 返回值说明**

返回err，若err=0，代表读取正常；否则，说明出现错误。

**⑤ 函数流程**

read\_raw\_super\_block\_by\_fsck\_info函数构造了一个虚拟的buffer\_header结构，里面的数据指针指向参数传递进来的data，而不是通过读取硬盘获取。然后会调用相关检查函数来验证超级块的一致性，若出现错误，我们会调用原始的read\_super\_block从硬盘重新获取。函数的具体流程如图4-21所示。

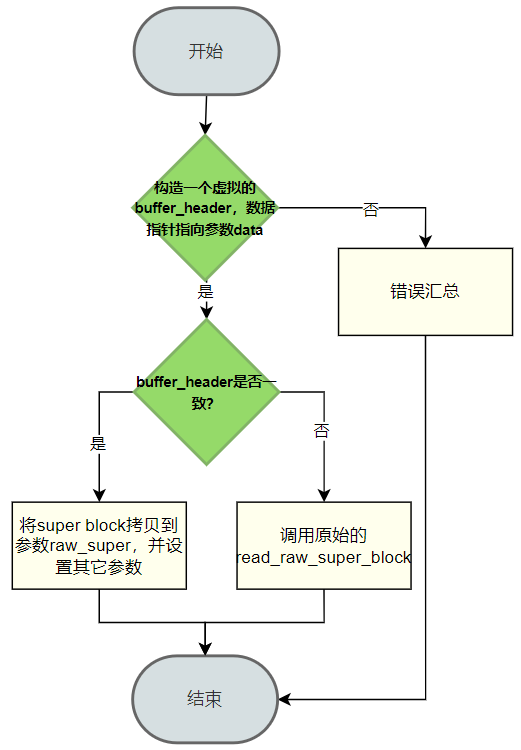


图 4‑21 read\_raw\_super\_block\_by\_fsck\_info函数流程

# 系统测试

## 5.1 加速fsck.f2fs

### 5.1.1 测试准备

对wfsck的测试分为功能测试与性能测试两个方面。功能测试主要测试wfsck能否正确检测并修复损坏的文件系统。性能测试主要通过测试wfsck在不同场景下的运行时间评估其性能，并将其与原始fsck工具对比。

文件系统中的文件和目录数量可能会变化并取决于应用程序。论文[32]对RocksDB(键值存储)、视频服务器、Web服务器和邮件服务器等工作负载进行分析，发现文件数量占主导地位(99%的文件占1%的目录)。测试中我们同样按此配置设置文件系统，即一个目录下包含100个文件。

实验在两台机器上进行，一台物理机，一台虚拟机。选择了一台虚拟机是因为运用虚拟机的场景在云计算时代越来越普遍，在虚拟机上的运行结果具有很高的参考价值。另一方面由于我们手上没有空闲Nand Flash存储介质，因此在回环设备上模拟了Nand Flash硬盘，回环设备运行在ext4的文件系统上，机器的具体信息如表5-1所示：

表5-1 机器配置

|  |  |  |
| --- | --- | --- |
| Target | System | Storage Device |
| 物理机\*16核 | **CPU**：Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz  **Memory**：128G  **OS**：Ubuntu 20.04.6 LTS | Samsung SSD 860 4B6Q 500G  在此硬盘上创建了一个大小为450G的分区，文件系统为ext4。在此文件系统中挂载了大小为1/10/50G的回环设备 |
| VirtualBox虚拟机\*4核 | **CPU**：Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz  **Memory**：8G  **OS**：Ubuntu 22.04.1 LTS | vdi硬盘 150G （在Seagate HDD 1T上创建）。硬盘上创建了一个130G的分区，文件系统为 ext4。在此文件系统上挂载了大小为1/2/5G的回环设备 |

在性能测试中，为了在文件系统中的生成测试文件，我们使用fs\_mark基准测试工具进行填充。针对1G的文件系统，我们填充512000\*1个大小为1024字节的文件，这将占用500M的空间，这符合一般文件系统的使用情况。以此类推，针对2G的文件系统，我们将填充512000\*2个大小为1024字节的文件。实验表明，我们的wFSCK在开启大于一个工作线程时，检测性能相比原来的fsck.f2fs将普遍提升超过25%。

在功能测试中，为了验证我们加速后的fsck.f2fs与原fsck.f2fs的行为一致，我们对文件系统的某些区域进行破坏，然后使用fsck.f2fs来找出被破坏的部分并进行纠正。来验证wFSCK在加速的同时并未破坏原功能完整性。

### 5.1.2 测试方法与测试结果

**wfsck检测与修复功能测试**

功能测试的测试流程如下：

1. 生成指定大小的文件系统镜像并运行fs\_mark程序为文件系统镜像填充文件；
2. 运行destroy.f2fs程序将文件系统镜像指定区域的数据用随机数据替换；
3. 将被破坏后的镜像拷贝两份；
4. 运行原始fsck程序检验文件系统损坏情况；
5. 运行wfsck程序检测并修复损坏的文件系统镜像；
6. 通过4、5得到的日志比较两者行为是否一致；
7. 若文件系统检测到损坏，回到第4步，分别通过各种的修复程序进行修复,然后再次对日志进行对比；否则，退出程序。

图5-1给出了一次功能测试的流程图。注意第3步，我们将镜像拷贝了两份，这样当原fsck与wfsck行为不一致时，我们可以在修复wfsck后，通过留存的镜像，再次进行测试，只有当两者行为一致，留存的镜像才能被安全的删除。

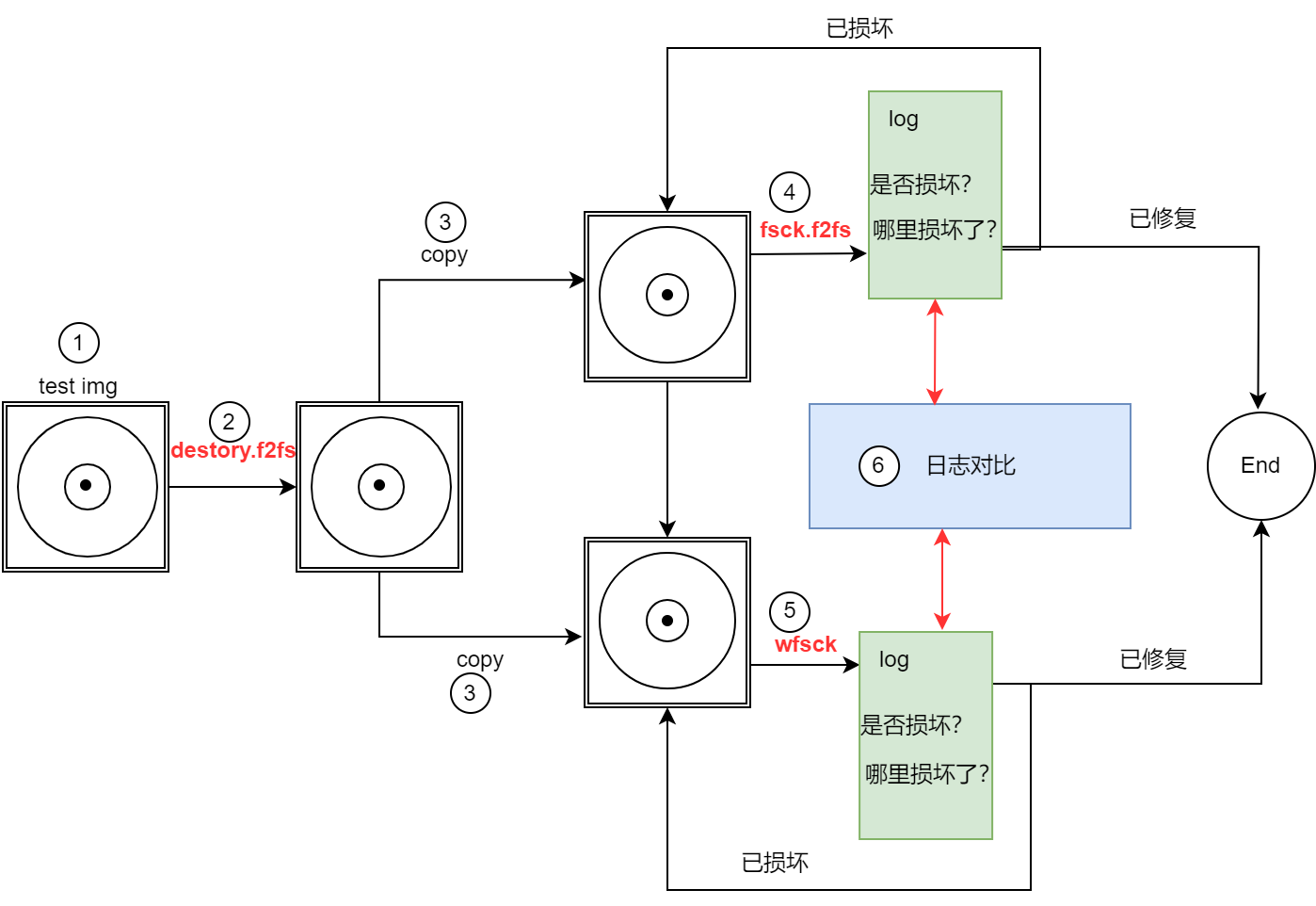


图 5‑1 功能测试流程

测试中我们生成了一大小为1GB的文件系统镜像，向该镜像的sit或nat或ssa区域随机写入数据，破坏该镜像。该区域包含100字节的垃圾数据。使用原始fsck工具与wfsck分别进行检查和修复，然后通过diff工具对过滤后的日志进行对比，若两者的日志一样，我们则认为两者的行为是一致的。目前，测试分了20轮，都未检测到fsck和wfsck之间行为的不一致，我们暂且认为wfsck和原fsck是一致的，后面我们会进行更为详尽的测试，比如破坏Main Area区域中node节点的数据、使用更精确的破坏工具破坏指定区域指定字段等。

**wfsck动态调整线程功能测试**

我们的实验设置为：在上述介绍的虚拟机环境中，wfsck和一个cpu密集型程序(通过stress-ng模拟)同时运行，wfsck开启的线程数为虚拟机的核数。stress-ng程序会被多次执行，直到wfsck退出。stress-ng每次执行固定操作的指令，结束时会打印本次执行的时间，然后睡眠10s，给予wfsck充足的时间执行，然后重新执行stree-ng，循环往复。我们收集每次stree-ng程序执行时间、wfsck修复程序消耗的时间以及wfsck线程数变化的信息。

图5-2 展示了运行期间wfsck线程数变化的情况。首先看蓝色的曲线，展示了系统cpu使用情况的变化，可以看到呈明显的周期变化，这是因为stress-ng被周期性被唤醒执行来模拟应用负载。再看红色的曲线，这是wfsck开启工作线程数量变化的曲线，也是呈现出周期的变化效果，当系统负载低时，如时间点20所示，线程数往上增长；而负载比较高时，如时间点30左右所示，线程数立刻往下掉，来避免影响其他应用的工作。绿色线和红色线变化趋势基本一致，绿色线表示了wfsck对cpu的占用率，核数越多，即工作线程数量越多时，对cpu的占用率也越高。

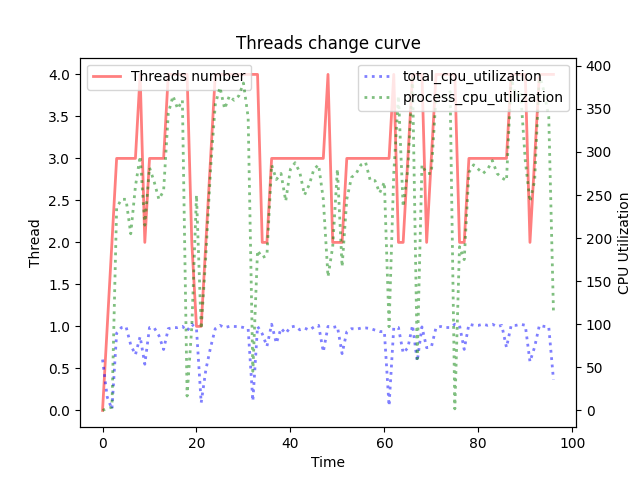


图 5‑2 wfsck线程数变化曲线

图5-3展示了wfsck对其他应用程序影响的量化效果。红色线是应用程序在没有wfsck干扰时正常的执行时间，标记为origin；绿色线是应用程序在wfsck的干扰下，并且wfsck没有开启动态调整功能时的执行时间，标记为no-scheduler；而蓝色线是应用程序在wfsck的干扰下，并且wfsck开启了动态调整功能时的执行时间，标记为scheduler。在wfsck没有开启动态调整功能时，应用程序的性能大大下降了，这是由于应用程序和wfsck相互争夺宝贵的cpu资源导致，而wfsck开启了动态调整功能后，此时应用程序的性能只是略微下降，这是由于wfsck开启了动态调整后能够根据系统当前资源使用情况动态调整线程数，不至于对其他程序造成太大影响。我们还获取了在no-scheduler和scheduler模式下wfsck的运行时间，发现scheduler下性能下降不超过5%，这是因为wfsck检测到系统资源充足时，会马上提升自己的线程数，同时巧妙地避免了和其他程序的竞争，使得在不干扰其他应用的同时保持自己的性能。

综上所述，wfsck动态调整功能能够根据当前系统的资源使用情况动态调整自己的线程数量，能够在不影响其他应用工作的同时稳定住自身的性能。

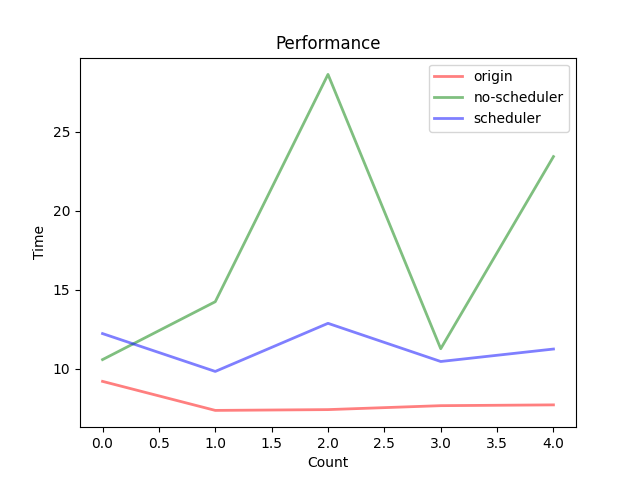


图 5‑3 wfsck动态调整功能对其他程序的影响

**性能测试**

性能测试的测试流程如下：

1. 生成指定大小的文件系统镜像用于测试；
2. 运行fs\_mark程序为文件系统镜像按指定配置填充文件；
3. 通过time程序捕获fsck程序的运行时间。

**情景1：物理机**

我们在物理机上创建了大小为1GB，10GB，50GB的文件系统镜像，分别测试了原始的fsck程序，以及使用不同线程数的wfsck程序的运行时间，并取3次运行的平均时间作为最终运行时间。结果统计如图5-4所示。图中纵坐标为程序运行时间，横坐标为wfsck使用的线程数，红色的虚线为原始fsck程序的运行时间，作为baseline进行比较。

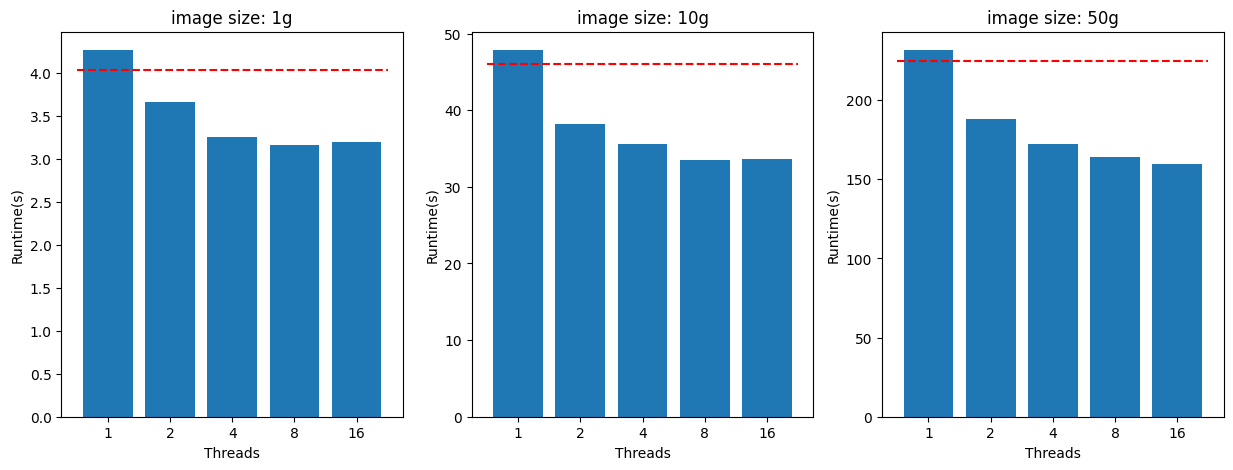


图 5‑4 wfsck在物理机上运行时间对比

可以看出，wfsck使用多线程并行时，运行速度超过原始fsck程序。随着线程数的增加，wfsck的运行速度也同样会加快，且在文件系统规模增大时，依然能保存这种趋势。说明wfsck成功利用多线程并行实现加速。其中线程数为1时，即不使用并行时，wfsck的运行速度略为下降，这是由于并行操作在程序中引入了额外的开销，使得性能略有下降。总体而言，在使用8线程并行时，wfsck程序可以加速约25%，最大可加速约29%。

现根据指标进一步分析wfsck。在大小为50gb的文件系统镜像上测试得到的指标如下表。其中Total time为程序运行总时间，在单核机器上等于用户态运行时间（User time），内核态运行时间（System time），与IO时间之和,即

而在多核机器上，我们有：

CPU usage为程序运行期间对CPU计算资源的占用情况。

表5-2 物理机性能测试结果(image size: 50GB)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 指标 | fsck | wfsck  -1thread | wfsck  -2thread | wfsck  -4thread | wfsck  -8thread | wfsck  -16thread |
| Total time(s) | 224 | 232 | 188 | 172 | 163 | 159 |
| Total time \* 16 cores | 3584 | 3712 | 3008 | 2752 | 2608 | 2544 |
| User time(s) | 11.39 | 18.62 | 23.24 | 32.47 | 35.45 | 33.98 |
| System time(s) | 36.78 | 38.50 | 49.10 | 65.48 | 69.99 | 75.45 |
| CPU usage(%) | 21 | 24 | 38 | 56 | 64 | 68 |

从表5-2的结果可以看出，随着wfsck使用线程数的增加，程序的用户态运行时间与内核态运行时间逐渐增加，而总体时间却逐渐下降。对此现象的解释是，wfsck通过多线程并行处理数据，提高了IO的利用率，降低了IO的等待时间，从而加速了程序运行。但与此同时，wfsck对于CPU的利用率非常低，在16线程的CPU上，使用率没超过100%，因此，系统的瓶颈在于I/O设备上，我们接下来的工作，将会对每个线程设立独立I/O缓存，以此提高I/O效率，达到更高的加速比。

**情景2：虚拟机**

我们在虚拟机上创建了大小为1GB，2GB，5GB的文件系统镜像，进行与情景1相同的测试，结果统计如图5-5所示。图中纵坐标为程序运行时间，横坐标为wfsck使用的线程数，红色的虚线为原始fsck程序的运行时间，作为baseline进行比较。

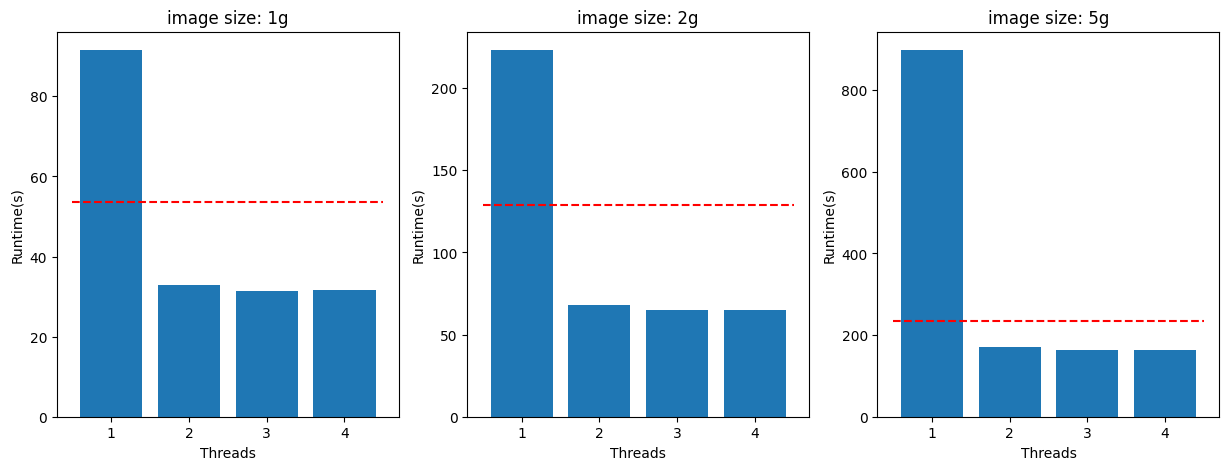


图 5‑5 wfsck在虚拟机上运行时间对比

可以看出，与物理机类似，wfsck使用多线程并行加速时，运行速度超过原始fsck程序。随着文件系统大小的增加，这种趋势同样成立。可是随着线程数的增加，wfsck的运行速度增长缓慢，我们猜测在此小硬盘上，两个线程已经能够达到最大的加速比，超过25%，而增加额外的线程受限于I/O的速度和系统基础开销，而不再有明显增益。值得一提是，开启一个线程的wfsck开销要远远高于原fsck，这让人非常意外，我们初步认定是由于这是在虚拟机环境的缘故，受限于虚拟机监控程序和Host操作系统，清缓存不见得总是能清理得非常干净，而wfsck\_1总是是生成磁盘后进行测试的第一个程序，将磁盘加载到内存耗费了大量时间，而后续由于缓存缘故，测试时间就减慢了许多。后续，我们将使用不同的线程数作为第一个测试程序来验证这种影响，并且提供缓解措施来避免这些错误数据导致的误解。但是总体而言，wfsck程序在使用2个工作线程及以上时，均可加速超过25%；而在2G硬盘的情况下，加速超过了50%。

现根据指标进一步分析wfsck。在大小为5gb的文件系统镜像上测试得到的指标如表5-3。

表5-3 虚拟机性能测试结果(image size:5GB)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 指标 | fsck | wfsck  -1thread | wfsck  -2thread | wfsck  -3thread | wfsck  -4thread |
| Total time(s) | 234 | 857 | 169 | 162 | 162 |
| Total time \* 4 cores | 936 | 3428 | 676 | 648 | 648 |
| User time(s) | 2.42 | 4.26 | 2.81 | 2.39 | 2.40 |
| System time(s) | 224.94 | 220.22 | 327.26 | 465.97 | 479.71 |
| CPU usage(%) | 96 | 25 | 194 | 288 | 295 |

可以看到wfsck开启2个线程即以上的时候，CPU利用率非常高，证明我们充分地利用了CPU资源，I/O的开销相比于上述物理机的情况少了许多，这也有可能是上述介绍的宿主机缓存的影响。但是，实验仍表明我们充分利用了CPU并行性，对文件系统检测和修复显著加速的效果。

## 5.2 优化f2fs的gc

### 5.2.1 测试准备

我们在5.1.1节介绍的两种配置的机器上分别测试。通过比对优化前GC和优化后GC的性能，来验证我们的效果。我们的目标是降低GC对用户程序读写性能的影响，因此，读写性能成为衡量系统好坏的标准，另外，为了验证是由于无效GC的次数增多导致的读写性能下降，我们还将无效GC的次数打印出来。

测试通过一个脚本自动化进行，脚本所做的工作为：不断调用fs\_mark往文件系统中写数据，然后输出此次写文件的性能和无效gc的次数等信息，最后将这些信息交给绘图脚本绘制出来；整个GC的测试流程如图5-6所示：通过run\_get\_log.sh脚本，我们能够得到当前F2FS文件系统性能的Log数据；通过draw.py绘图脚本，我们能将这些数据可视化呈现出来。分别对原f2fs文件系统和GC优化后的f2fs文件系统进行性能测试，就得到了5.2.2小节中的性能对比图。

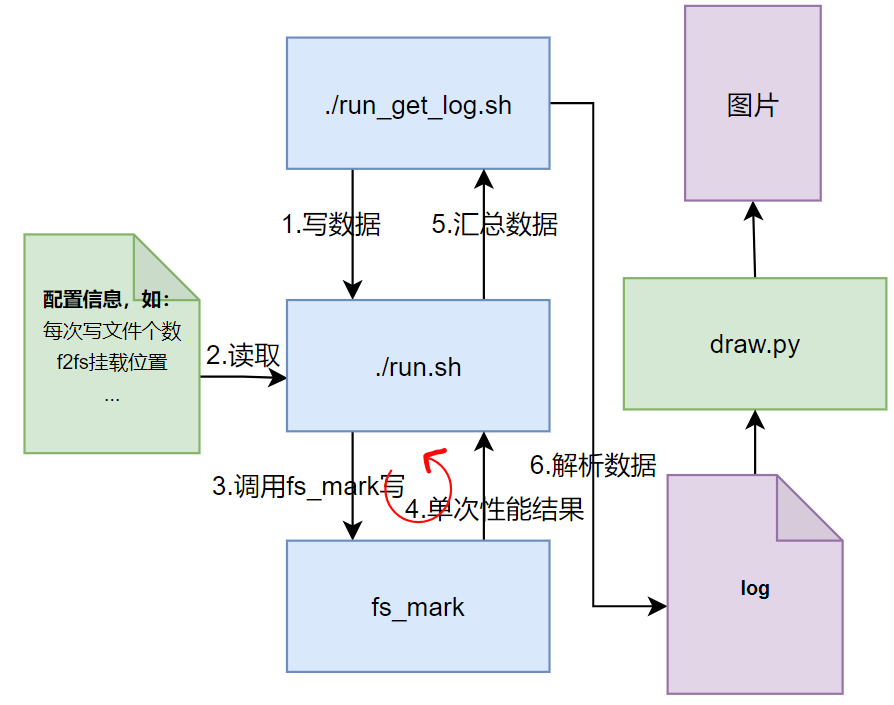


图 5‑6 GC测试脚本工作流程

### 5.2.2测试方法与测试结果

我们针对原始版本和改进后的版本，多次调用fsmark，每次创建一定数量的文件。记录该次的执行时间和总的gc次数，无效的gc次数，其中无效的gc分为invalid gc和useless gc。invalid gc是指没有segment可以回收，useless gc是指待回收的segment有效块非常多。图5-7中横坐标为fsmark调用的次数，左边的纵坐标为写文件的速率，右边的纵坐标为gc的次数。折线origin write代表原始版本写文件的速率，折线our write代表改进版本写文件的速率。折线origin gc\_count代表原始版本gc的总次数，折线our gc\_count代表改进版本gc的总次数。折线origin invalid代表原始版本invalid gc次数，折线our invalid代表改进版本invalid gc次数，折线origin useless代表原始版本useless gc次数，折线our useless代表改进版本useless gc的次数。

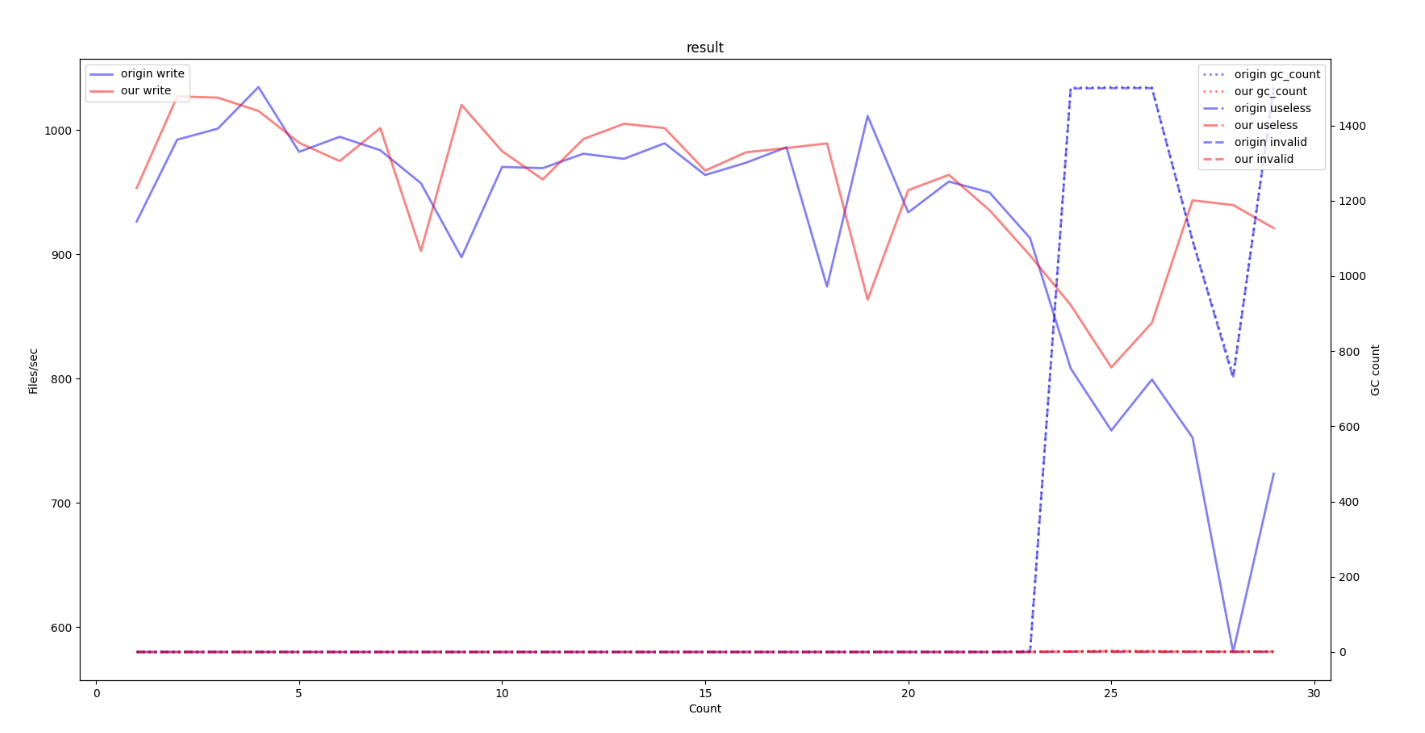


图 5‑7 gc次数变化曲线

如图5-7所示，一块空的磁盘，大小为100M，fsmark每次写500个文件，每个文件大小为2048B。随着fsmark调用次数增加，磁盘的利用率也在增加。在第23次时，空间将要不足，原始版本触发前台gc，导致性能下降。而这里触发的gc，几乎全是invalid gc（图中origin gc\_count与origin invalid几乎重合），也就是根本没有segment可以回收。而改进版本在触发一次无效gc后将停一会写流程中触发前台gc的逻辑，所以改进版本前台gc次数很少，fsmark写的性能也没有下降。

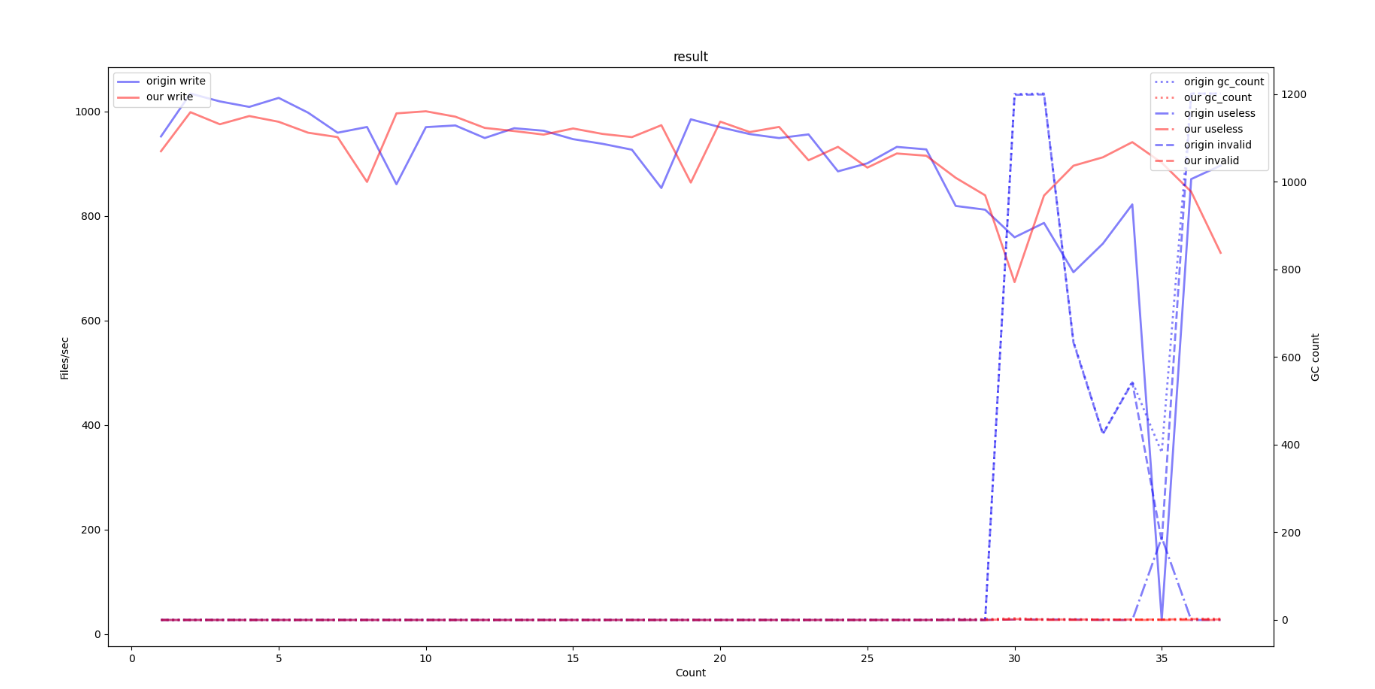


图 5‑8 gc次数变化曲线

如图5-8所示，一块空的磁盘，大小为100M，fsmark每次写400个文件，每个文件大小为2048B。随着fsmark调用次数增加，磁盘的利用率也在增加。在第29次时，空间将要不足，原始版本触发前台gc，导致性能下降。而这里触发的gc，几乎全是invalid gc（图中origin gc\_count与origin invalid几乎重合），也就是根本没有segment可以回收。特别注意，第35次时，出现了许多useless gc （回收的segment有效block较多），性能下降比较严重。而改进版本在触发一次无效gc后将停一会写流程中触发前台gc的逻辑，所以改进版本前台gc次数很少，fsmark写的性能也没有下降。

## 5.3 利用fsck收集的信息

### 5.3.1 测试准备

我们对利用fsck的信息挂载进行功能测试，测试说明我们成功实现了利用fsck的信息进行挂载。我们在5.1.1节中介绍的物理机和虚拟机上分别进行实验，均得到了正确的结果。我们的实验过程过程为：在不加载fsck-info模块时进行f2fs文件系统挂载，然后对文件系统进行读写操作，然后取消挂载；然后再在加载fsck-info模块时进行f2fs文件系统挂载，对文件系统进行读写操作，然后取消挂载；最后不加载fsck-info模块进行f2fs文件系统挂载，判断上述操作一致。若一致，证明我们能通过fsck-info模块来帮助f2fs文件系统挂载。

挂载的具体流程如图5-9所示，测试过程中进行了3次挂载，只有第二次是通过fsck-info模块提供的信息挂载，第一次和第三次都是无fsck-info的正常挂载流程。在每次过载的过程中对文件系统读写，然后判断文件系统的一致性来验证我们成功实现利用fsck-info模块信息进行挂载。

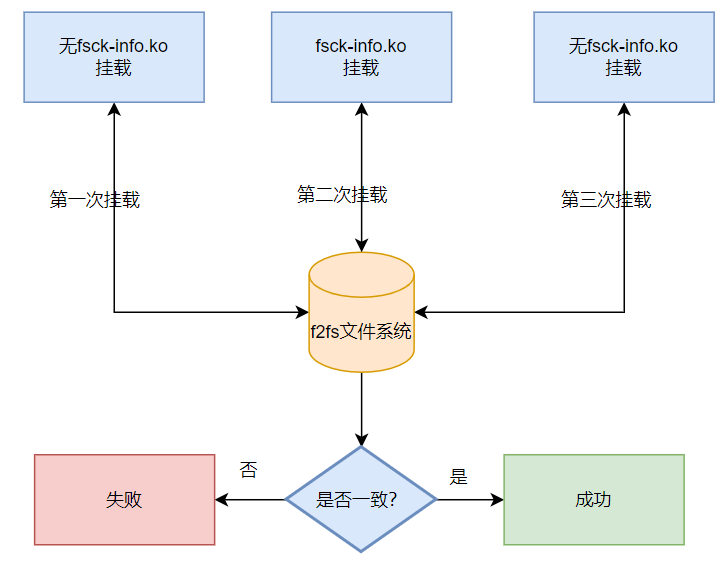


图 5‑9 利用fsck收集的信息功能测试流程

### 5.3.2 测试结果

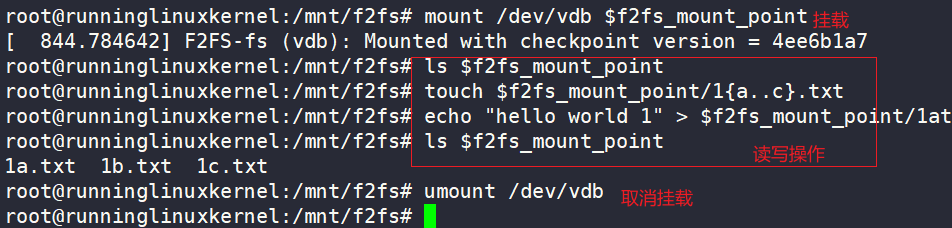


图 5‑10 第一次挂载（无fsck-info）

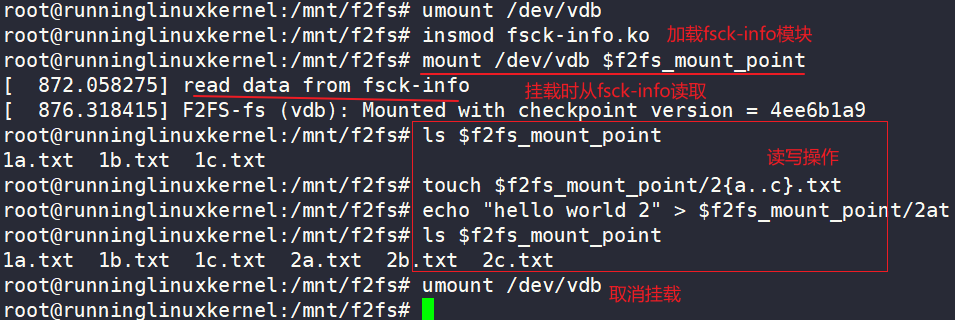


图 5‑11 第二次挂载（有fsck-info）

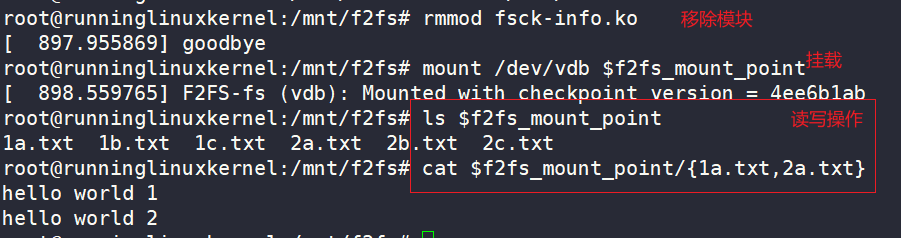


图 5‑12 第三次挂载（无fsck-info）

图5-10至5-12分别列出了3次挂载的结果。可以看到，取决于有无加载fsck-info模块，f2fs挂载时能动态地决定从哪里获取挂载信息，并且利用fsck信息进行挂载能成功地实现挂载功能。

# 总结与展望

## 6.1 工作总结

在wFSCK项目中，我们对f2fs文件系统和其检查工具fsck.f2fs都有了更加深刻的了解，包括文件系统的布局，如何写数据，如何进行gc等。其中我们走过一些弯路，也做了一些尝试，但没有取得好得效果。比如给每个线程设计一个I/O缓存，并没有加速的效果。比如想优化f2fs的trim操作，但完整了解trim过程后，发现没有优化空间。比如想通过提前后台gc来减少前台gc，却没有取得好的效果。但最终我们还是完成了f2fs优化的三个目标。

在wFSCK项目中，我们完成了：

* 针对f2fs复杂的文件系统结构，将pFSCK的并发思想移植到fsck.f2fs，将fsck.f2fs对Node树的检查拆分，并行执行检查。
* 对fsck.f2fs的检查任务进行划分，同时不影响C/R的正确性。
* 保存任务执行前的上下文，对任务的返回值进行处理，保证了C/R一致性。
* 对共享文件系统结构访问进行控制，重新设计数据结构，解决同步访问共享结构的瓶颈。
* 智能感知系统资源使用情况，动态调整工作线程个数。
* 发现f2fs存在大量无效gc的问题
* 通过gc控制线程，减少无效gc的触发
* fsck.f2fs检查过程中收集信息，优化后续挂载等情况。fsck.f2fs在检查文件系统的过程中，会遍历所有的文件元数据信息，这信息可以被记录下来，用于优化后续对文件系统的读写。

下表显示了对应赛题完成情况：

|  |  |  |
| --- | --- | --- |
| **目标** | **基本完成情况** | **额外说明** |
| 加速fsck.f2fs | 完成（100%） | ① 可节省25%到50%左右的运行时间；  ② 可动态调整线程个数 |
| 优化f2fs的gc | 完成（100%） | ① 减少两种无效的gc。  ② 大幅提升写性能。 |
| 利用fsck收集的信息 | 完成（100%） | ① 可将fsck.f2fs中的超级块信息传递给f2fs挂载时使用。 |

## 6.2 创新点

主要创新点如下：

* 完成了论文pFSCK的future work，将论文pFSCK的思想移植到fsck.f2fs中，使得fsck.f2fs更加敏捷，性能提升25%-50%。克服了移植过程中的难点。比如pFSCK的检查过程与fsck.f2fs的检查过程是不一样的。pFSCK是分阶段进行检查，fsck.f2fs是对文件树进行递归地检查。检查过程不一样，导致任务的划分也会不一样。任务如何划分又决定了检查结果的正确性，同时又要正确地对任务地返回值进行处理。文件系统检查恢复变为并发过程，要解决的重要问题就是保证并发后的结果一致性。需要在复杂的文件系统布局中，处理大量共享变量的访问。
* 我们发现了f2fs在一些场景下，存在大量无效gc的问题。并用简单有效的方法减少了无效gc，使得gc更加智能。
* 据我们所知，目前没有工作将fsck.f2fs与f2fs文件系统配合起来。我们利用了fsck.f2fs收集的文件系统信息，优化后续f2fs的挂载，使得f2fs更加智能。证明了将fsck.f2fs与f2fs文件系统配合起来是可行的。作为一个引子，希望未来能利用fsck.f2fs信息对f2fs文件系统有更多优化。

## 6.3 未来展望

虽然wFSCK做了很多工作，但还有很多的Futrue Works，例如：

* 如何划分任务，使其任务量更均衡，进一步加速fsck.f2fs。
* 如何调度任务的执行，当前任务调度策略是FIFO先入先出，可以引入其他策略，进一步加速fsck.f2fs。
* 如何彻底避免某些segment有效块特别多的情况出现。
* 如何充分利用检查过程中收集到的信息，结合f2fs的特性，做更多的性能优化。

# 参考文献

1. Marshall Kirk McKusick, Willian N Joy, Samuel J Lefflfler, and Robert S Fabry. Fsck- the unix† fifile system check program. *Unix System Manager’s Manual-4.3 BSD Virtual VAX-11 Version*, 1986.
2. Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In 16th USENIX Conference on File and Storage echnologies (FAST 18), pages 1–14, Oakland, CA, 2018. USENIX Association.
3. Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating fifile system reliability on solid state drives. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 783–798, Renton, WA, July 2019. USENIX Association.
4. William (Bill) E. Allcock. Parallel File Systems at HPC Centers: Usage,Experiences, and Recommendations.https://www.nersc.gov/assets/Uploads/W01-DataIntensiveComputingPanel.pdf.
5. HPC-Users Mailing List. Outages in HPC Systems. <https://maillists.uci.edu/pipermail/hpc-users/2019-December/000095.html>
6. e2fsck: fsck for ext4. <https://linux.die.net/man/8/e2fsck.>
7. Val Henson, Zach Brown, and Arjan van de Ven. Reducing fsck time for ext2 fifile systems. 04 2019.
8. M. Lu, T. Chiueh, and S. Lin. An incremental fifile system consistency checker for block-level cdp systems. In *2008 Symposium on Reliable Distributed Systems*, pages 157–162, Oct 2008.
9. Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci Dusseau, and Marshall Kirk Mckusick. Ffsck: The fast fifile-system checker. *Trans. Storage*, 10(1):2:1–2:28, January 2014.
10. Marshall K. McKusick. Improving the performance of fsck in freebsd. *ogin:*, 38(2), 2013.
11. Marshall Kirk McKusick, Willian N Joy, Samuel J Lefflfler, and Robert SFabry. Fsck- the unix† fifile system check program.*Unix System Manager’s Manual-4.3 BSD Virtual VAX-11 Version*, 1986.
12. Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanu malayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated crash vulnerabilities. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI’16, pages 151–167,Berkeley, CA, USA, 2016. USENIX Association.
13. Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sqck: A declarative fifile system checker. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI’08, pages 131–146, Berkeley, CA, USA, 2008. USENIX Association.
14. Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. OOPSLA ’09, page 227–242, New York, NY, USA, 2009. Association for Computing Machinery.
15. Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ouster hout. Arachne: Core-aware thread management. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI’18, pages 145–160, Berkeley, CA, USA, 2018. USENIX Association.
16. Intel-Micron Memory 3D XPoint. <http://intel.ly/1eICR0a.>
17. Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. f2fs: A New File System for Flash Storage. In Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST’15, Santa Clara, CA, 2015.
18. Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance fifile system for non-volatile main memory. In Proceedings of the Eleventh European Conference on Computer Systems, pages 1–16, 2016.
19. Matthew Wilcox and Ross Zwisler. Linux DAX. <https://www.kernel.org/doc/Documentation/filesystems/dax.txt.>
20. Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: reducing software overhead in fifile systems for persistent memory. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 494–508, 2019.
21. Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP ’17, 2017.
22. Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In Proceedings of the 14th Usenix Conference on File and Storage Technologies, FAST’16, 2016.
23. Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfifit as distributed storage backends: lessons from 10 years of ceph evolution. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 353–369, 2019.
24. Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating fifile system reliability on solid state drives. In 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19), pages 783–798, 2019.
25. Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. ACM Transactions on Storage (TOS), 4(3):8, 2008.
26. Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS 07, 2007.
27. Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 228–243. ACM, 2013.
28. Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating fifile system reliability on solid state drives. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 783–798, Renton, WA, July 2019. USENIX Association.
29. Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. Reliability analysis of ssds under power fault. ACM Trans. Comput. Syst., 34(4):10:1–10:28, November 2016.
30. Mtanski. mtanski/xfsprogs github.com/mtanski/xfsprogs/preadv2/repair. https://github.com/mtanski/xfsprogs/tree/preadv2/repair, Feb 2015.
31. StackExchange - Extremely long time for an ext4 fsck. https://unix.stackexchange.com/questions/78785/extremely-long-time-for-an-ext4-fsck, Mar 2013.
32. Domingo D, Kannan S. pFSCK: Accelerating File System Checking and Repair for Modern Storage[C]//Proceedings of the 19th USENIX Conference on File and Storage Technologies. 2021.