**文件元数据查找优化**

作者：杨恩慧 学校：河北工业大学

摘要：本项目提出了一种高效的元数据查询优化方案，通过引入倒排索引和使用Varint编码进行压缩，实现了目录树结构和倒排索引的并发读写管理。实验结果表明，该方案在大规模文件系统中显著提升了元数据查询的性能，并减少了内存占用。我们设计并实现了多种优化策略，包括读写锁机制、线程安全的HashMap，以保证在高并发环境下的数据一致性和系统性能。

关键词：元数据查询，倒排索引，Varint编码，读写锁，高并发，内存优化

# 一、引言

## 1.1.研究背景

元数据（Metadata），又称中介数据、中继数据，为描述数据的数据（data about data），主要是描述数据属性（property）的信息，例如文件存储位置、文件大小、访问权限等信息。用户在访问文件数据之前需要访问文件的元数据以获得文件的基本属性信息和数据的索引信息。

文件系统中，使用**目录树结构**存储文件元数据。系统可以根据目录树查找到你请求的文件的元数据（在Linux中使用stat命令查看文件元数据），进而对文件进行读写操作。

在传统的文件检索中，用户给定文件名，系统在目录树上执行检索。这种方式在现有的基于目录树的文件系统上表现较好。但现有人工智能等应用对文件系统提出了新的查找需求，它们往往需要查找一类元数据相同的文件集合。例如，

* + 查询所有文件后缀名为.png的文件
  + 查询修改时间在xxx之后的所有文件
  + 查询文件大小超过200KB的所有文件

要执行这类查询（Linux上的find的命令）需要遍历整颗目录树以获取查询结果，导致在大规模文件系统中性能极差。每次查询都需要线性扫描整个文件系统，查询时间随文件数量的增加呈线性增长，严重影响了查询效率。

## 1.2.研究目标

元数据本身通常很小，只占用几百 byte 的空间。但是随着存储系统规模的增大，元数据的规模也不断扩大。研究高效的元数据索引方法以支持高效的读、写和遍历操作是文件系统元数据管理的一个关键问题。

为了加速**海量文件元数据的查询**，我们可以在目录树之外再建立一套索引，专门优化元数据的查询。**倒排索引**比较适合这一任务，我们可以为每一个文件元数据建立一条倒排链，查询时就可以快速获得元数据对应的文件id（或文件名）。一个倒排索引的例子如下：

表 1 倒排索引示意表

|  |  |
| --- | --- |
| 元数据 | 文件id (文件名) |
| jpg(后缀名) | 1001, 1002, 1008, 1010 |
| png(后缀名) | 1003, 1004, 1006 |
| 211KB(文件大小) | 1001, 1004, 1010 |
| 2020/06/07(文件创建时间) | 1010, 1008, 1005 |
| user1(文件所有者) | 1001, 1004, 1006, 1008, 1010 |

利用**倒排索引**，我们可以快速找到文件元数据。例如查询所有后缀名为jpg的文件时，只需要查找jpg对应的一条倒排链，这极大减少了查询时间，同时减小遍历整个目录树带来的开销。

## 1.3主要工作和创新

在本项目中，我们提出了一种高效的元数据查询优化方案，主要工作和创新点如下：

1. **引入倒排索引优化元数据查询：**

为解决传统目录树结构在大规模文件系统中性能低下的问题，我们在目录树之外建立了倒排索引。倒排索引通过将文件元数据与对应的文件ID进行关联，显著提升了元数据查询的速度。通过这种方式，可以快速定位具有特定元数据属性的文件，而无需遍历整个目录树。

1. **采用Varint编码进行倒排索引压缩：**

针对倒排索引带来的存储空间开销，我们引入Varin编码进行压缩。Varint编码是一种变长整数编码方式，能够有效地压缩需要存储的大量小整数，如倒排索引中的文档ID或词频等。实验结果表明，使用Varint编码后，倒排索引的存储空间显著减少，同时保持了较高的解压速度。

1. **设计并实现了读写锁机制以支持高并发：**

为了在高并发环境下保证数据的一致性和系统性能，我们设计了读写锁机制。通过在目录树和倒排索引的关键操作中引入读写锁，能够有效地减少锁的争用，提高系统的并发性能。此外，我们使用了线程安全的ConcurrentHashMap来替代传统的HashMap，进一步提升了系统的并发处理能力。

# 二、理论基础及相关技术

## 2.1目录树结构

目录树结构是文件系统中组织和管理文件和目录的一种层次化结构。它的设计灵感来源于树状图，每个节点代表一个文件或目录，根节点代表文件系统的起点。从根节点出发，通过一系列分支节点，最终连接到各个叶节点（即文件）。这种层次化结构不仅直观清晰，还能有效地组织大量文件和目录，便于用户查找和管理。

在目录树结构中，每个目录可以包含子目录和文件，子目录又可以进一步包含其自己的子目录和文件。这种嵌套的层次结构使得目录树能够表示复杂的文件关系。

目录树结构还支持多种文件操作，如创建、删除、移动和重命名文件或目录。这些操作通过修改节点的链接关系实现。目录树中的每个节点通常包含文件的元数据，如文件名、大小、权限和时间戳等，这些信息帮助文件系统快速访问和管理文件。

目录树的构建过程通常是从根节点开始，通过递归或迭代的方式遍历每个目录，逐层添加文件和子目录。例如，可以使用深度优先搜索（DFS）或广度优先搜索（BFS）算法来遍历目录树。

## 2.2.倒排索引

倒排索引（Inverted Index）是一种高效的检索数据结构，广泛应用于全文搜索引擎和数据库系统中。它的基本原理是通过将文档中的每个词条与包含该词条的文档ID进行关联，从而实现快速查询。

倒排索引包含两个主要部分：

* **词典（Dictionary）**：存储所有可能查询的词条。
* **倒排列表（Posting List）**：每个词条对应一个倒排列表，存储包含该词条的所有文档ID。

倒排索引的实现通常分为以下几个步骤：

1. **词条提取**：从文档中提取所有可能的查询词条。在文件系统中，这些词条可以是文件元数据的各个属性值，如文件后缀名、文件大小、创建时间等。
2. **构建词典**：将提取的词条存储在词典中，并为每个词条分配一个唯一的ID。
3. **构建倒排列表**：遍历所有文档，将每个文档中包含的词条与文档ID关联，生成倒排列表。每个词条的倒排列表存储所有包含该词条的文档ID。

倒排索引的主要优点在于其高效的查询性能。由于索引将词条直接链接到相关文档，因此在处理查询时，系统可以快速定位并返回包含查询词的文档列表，大大减少了检索时间。此外，倒排索引支持多种复杂查询类型，如布尔查询、短语查询和近似查询等，灵活性较强。然而，倒排索引也存在一些缺点。首先，构建和维护索引的开销较高，特别是在动态更新频繁的大规模文档库中，索引的实时更新会消耗大量的计算资源。其次，倒排索引需要占用较大的存储空间，尤其是在处理包含大量高频词汇的文档集合时，索引的体积会显著增加。总体而言，尽管存在这些挑战，倒排索引仍然是实现高效信息检索的核心技术，广泛应用于搜索引擎、数据库和大数据分析等领域。

## 2.3现代文件系统元数据管理的相关研究(以HDFS为例)

Hadoop分布式文件系统（Hadoop Distributed File System，HDFS）是一个高度可扩展、容错性强的分布式文件系统，旨在运行在通用硬件上。HDFS是Hadoop生态系统的核心组件之一，主要用于存储大规模数据集，并确保数据的高可用性和可靠性。

HDFS采用主从架构，包括以下主要组件：

* **NameNode**：负责管理文件系统的元数据，包括文件和目录的结构、权限、块位置等。NameNode是HDFS的单点控制器。
* **DataNode**：负责存储实际的数据块（block）。每个DataNode从NameNode接收指令，并处理数据块的读写请求。
* **Secondary NameNode**：定期从NameNode获取元数据的检查点，以减轻NameNode的负担，但它并不是NameNode的热备份。

在HDFS中，元数据主要由NameNode管理，并且存储在NameNode的内存中。在NameNode中的Namespace管理层是负责管理整个HDFS集群文件系统的目录树以及文件与数据块的映射关系。以下就是Namespace的内存结构：

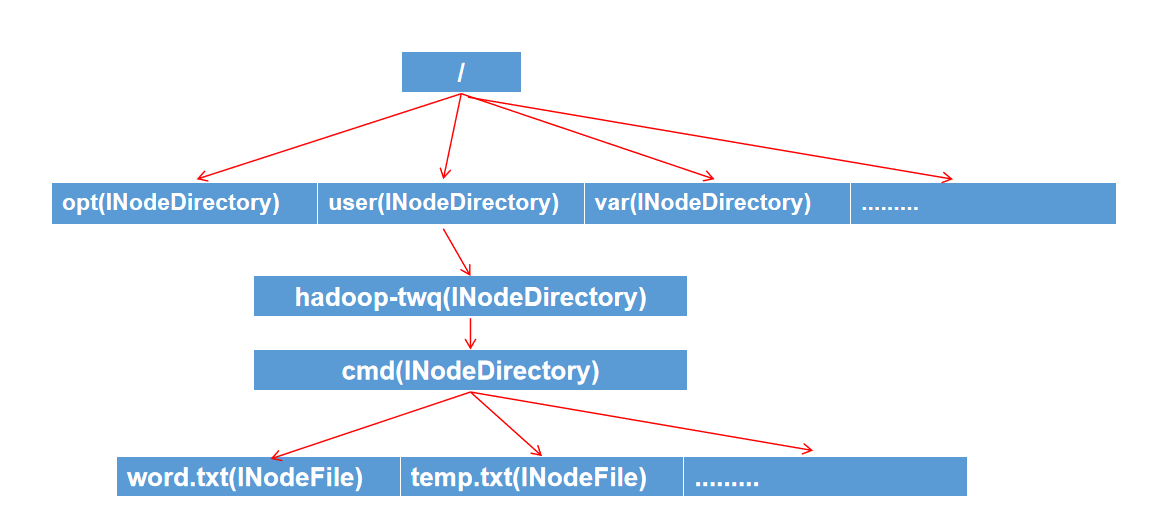


图 1 Namespace的内存结构图

文件元数据类INodeFile和目录元数据类INodeDirectory均继承于INode父类，针对其公共属性抽象出INodeWithAdditionalFields类。INodeDirectory 类中维护List<INode>结构表示该文件夹下的文件或子文件夹FSDirectory类维护内存目录树，其通过维护INodeDirectory类型的 rootDir构建整个内存中的目录结构。

# 三、基础设计与实现

## 3.1总体架构概述

本系统的总体架构设计包括两个主要部分：文件系统管理（FSDirectory）和倒排索引（InvertedIndex）。FSDirectory负责文件和目录的创建、更新、删除操作，以及目录树的管理；InvertedIndex用于快速检索文件的元数据信息，提高查询效率。

## 3.2文件系统管理（FSDirectory）的设计

文件系统管理（FSDirectory）模块是本系统的核心组成部分之一，主要负责文件和目录的创建、更新、删除操作，以及目录树的管理。该模块的设计目标是提供一个高效、可靠的文件系统目录管理机制，以支持复杂的文件操作和快速的元数据检索。

### 3.2.1数据结构

FSDirectory 模块采用了一种树形结构来组织文件和目录。在该结构中，每个目录和文件节点均由INode类及其子类表示。目录节点INodeDirectory 类表示，文件节点由INodeFile类表示。这些节点包含了文件或目录的基本信息，例如名称、权限、所有者、创建时间和修改时间等。

* **INode类**：作为所有节点的基类，包含节点的唯一标识符（ID）和父节点引用。ID 通过静态原子整数生成器确保唯一性，父节点引用用于在目录树中维护节点的层次结构。
* **INodeWithAdditionalFields类**：继承自INode类，扩展了基本节点信息，包含名称、所有者、权限、创建时间和修改时间等属性。该类用于统一存储文件和目录的公共元数据字段。
* **INodeFile类**：继承自INodeWithAdditionalFields类，表示文件节点，包含文件特有的元数据属性，如文件名、文件扩展名、文件大小和数据块列表。数据块列表用于记录文件存储的实际数据块信息。
* **INodeDirectory类**：继承自INodeWithAdditionalFields类，表示目录节点，包含一个子节点列表，用于存储其下属的文件和子目录。

上述几类的继承关系的示意图如下：

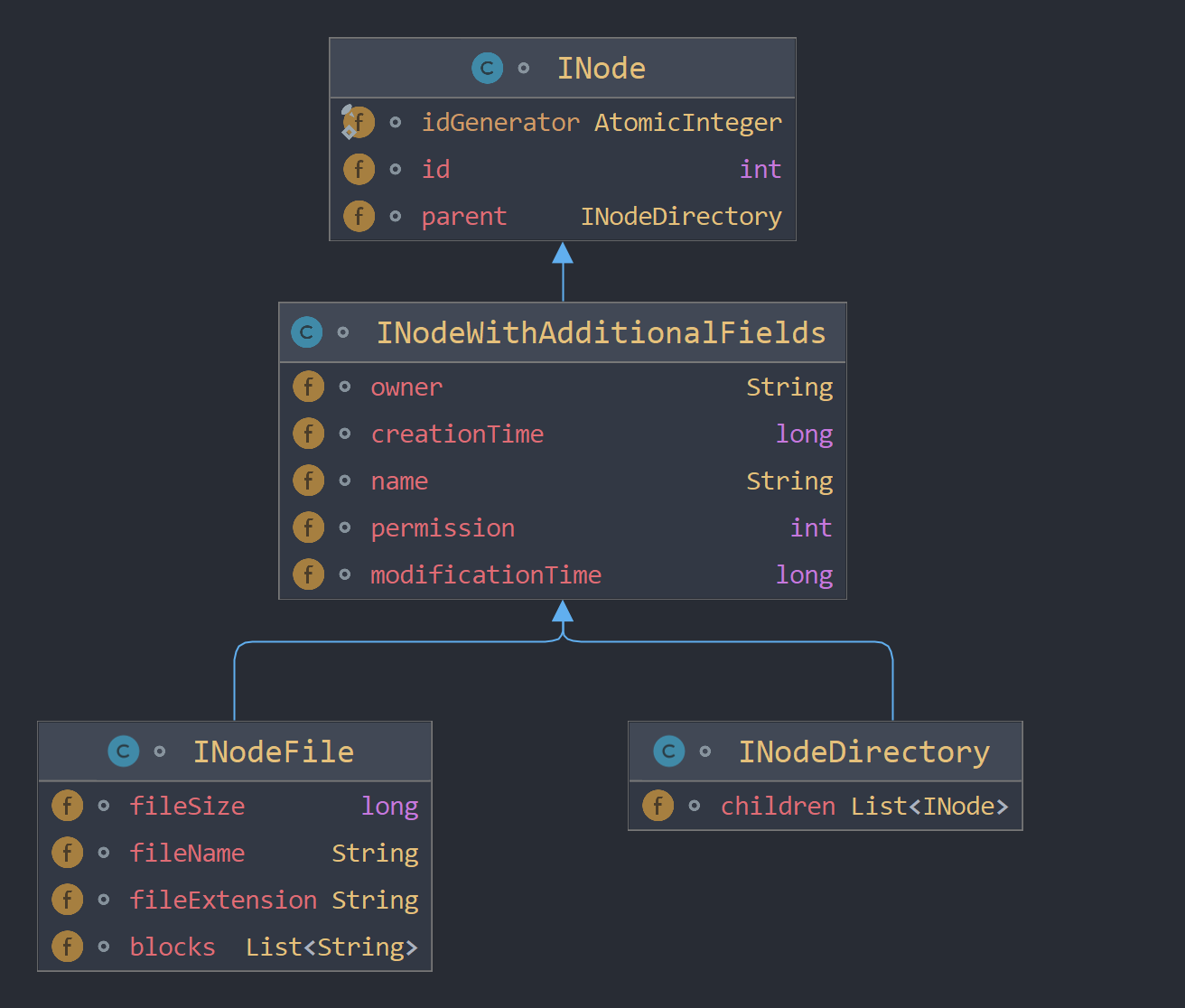


图 2 目录树类的继承关系图

FSDirectory 类维护内存目录树，其通过维护INodeDirectory类型的 rootDir构建整个内存中的目录结构。并通过子节点的递归方式构建完整的文件系统结构。

通过继承和多态机制，实现文件和目录元数据的层次化管理，确保代码结构清晰、易于理解。同时，INodeWithAdditionalFields类统一定义了文件和目录的公共元数据字段，具体文件和目录类通过继承扩展，实现灵活的功能扩展。

以下示例展示了文件元数据的层次化设计及其在目录树中的应用：

/ (INodeDirectory)

├── dir1 (INodeDirectory)

│ ├── file1.txt (INodeFile)

│ ├── file2.jpg (INodeFile)

│ └── subdir1 (INodeDirectory)

│ └── file3.png (INodeFile)

├── dir2 (INodeDirectory)

│ └── file4.doc (INodeFile)

└── dir3 (INodeDirectory)

├── file5.pdf (INodeFile)

└── subdir2 (INodeDirectory)

└── file6.txt (INodeFile)

在该示例中，根目录 / 包含多个子目录 dir1、dir2 和 dir3，每个子目录下又包含不同的文件和子目录。文件和目录的元数据通过相应的 INode 类及其子类进行管理和存储，确保目录树结构清晰、操作高效。

### 3.2.2功能设计

FSDirectory 模块提供了一系列用于操作文件和目录的方法，包括创建、删除、更新和查询等操作。这些方法主要包括：

* **创建文件（createFile）**：根据指定路径、所有者、权限等信息创建新文件。方法首先解析路径，找到父目录节点，然后在父目录中添加新的文件节点，并更新倒排索引。
* **创建目录（createDirectory）**：根据指定路径、所有者和权限创建新目录。方法首先解析路径，找到父目录节点，然后在父目录中添加新的目录节点。
* **更新文件（updateFile）**：更新指定路径文件的元数据信息（如文件名、扩展名、大小等）。方法首先找到目标文件节点，移除旧的元数据索引，更新节点信息，然后添加新的元数据索引。
* **删除节点（deleteNode）**：删除指定路径的文件或目录。方法首先解析路径，找到目标节点，然后从父目录中移除该节点，并更新倒排索引。
* **查询目录树（searchInTree 和 searchInTreeRange）**：在目录树中进行元数据的精确和范围查询。方法通过递归遍历目录树，匹配符合条件的节点。

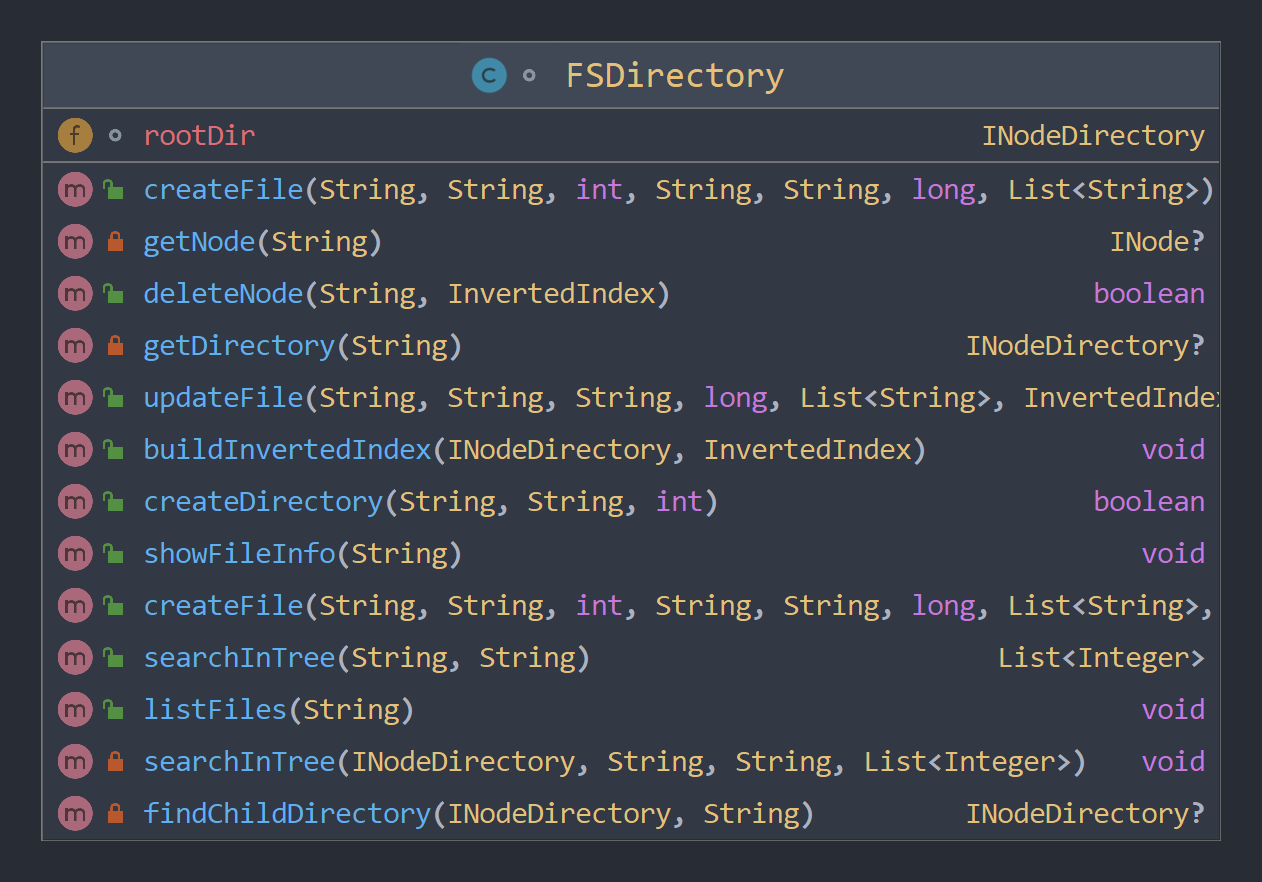


图 2 FSDirectory类的示意图

### 3.2.3功能实现

在实际实现中，FSDirectory 模块的各方法通过遍历目录树来执行相应的文件和目录操作。创建和更新操作首先通过路径解析找到目标父目录节点，然后进行相应的节点添加或更新操作。删除操作则通过路径解析找到目标节点及其父目录，并执行删除操作。查询操作通过递归遍历目录树，匹配节点的元数据条件，返回符合条件的节点列表。

这些操作的核心在于**路径解析**和**节点操作**。路径解析将输入的路径字符串拆分为各级目录名，通过逐级查找的方式定位到目标父目录或节点。节点操作包括在父目录中添加、删除或更新子节点，以及在目录树中递归查找符合条件的节点。

## 3.3倒排索引（InvertedIndex）模块设计

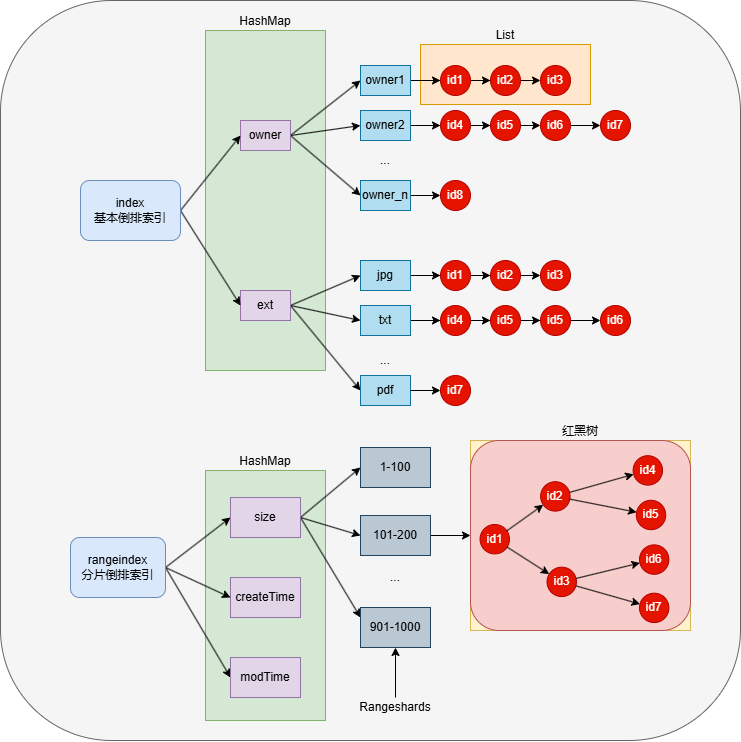
倒排索引（InvertedIndex）模块是文件系统中用于优化元数据查询的重要部分。通过建立元数据属性到文件ID列表的映射，倒排索引能够显著提升查询性能，特别是在处理大量文件时表现尤为显著。本设计基于高效、可扩展和便于维护的原则，实现了一个支持**多种元数据类型**的倒排索引模块。

### 3.3.1数据结构

倒排索引模块的核心数据结构包括两个主要部分：**基本倒排索引**和**分片倒排索引**。

基本倒排索引（index）用于存储离散的字符串类型元数据，如文件扩展名（ext）和所有者（owner）。其结构为 Map<String, Map<String, List<Integer>>>，其中外层键为元数据类型，内层键为具体的元数据值，值为文件 ID 列表。

分片倒排索引（rangeIndex）用于存储连续的数值类型元数据，如文件大小（size）、创建时间（creation）和修改时间（mod）。其结构为 Map<String, List<RangeShard>>，其中键为元数据类型，值为分片列表。每个分片（RangeShard）包含一定范围内的数值和对应的文件 ID。



### 3.3.2详细设计

#### 1 基本倒排索引

基本倒排索引用于处理离散的字符串元数据。其主要操作包括添加索引、删除索引和查询索引。

* **添加索引**：将文件的特定元数据值添加到相应的倒排索引中。例如，将文件的扩展名和文件 ID 添加到扩展名索引中。
* **删除索引**：从倒排索引中移除文件的特定元数据值及其对应的文件 ID。
* **查询索引**：根据特定元数据值，快速检索所有具有该值的文件 ID 列表。

#### 2 分片倒排索引

分片倒排索引用于处理连续的数值元数据。其设计包含分片管理、添加索引、删除索引和范围查询等功能。

* **分片管理**：每个数值元数据类型对应一个分片列表，每个分片（RangeShard）包含一个数值范围和相应的文件 ID。通过动态调整分片大小和合并分片，确保查询的高效性和分片的均衡性。
* **添加索引**：将文件的数值元数据添加到相应的分片中。如果分片大小超过阈值（SHARD\_SIZE\_THRESHOLD），则进行分片拆分。
* **删除索引**：从相应分片中移除文件的数值元数据及其对应的文件 ID。
* **范围查询**：根据给定的数值范围，查找所有在该范围内的文件 ID。通过遍历相关分片并合并结果，确保查询的准确性和高效性。

#### 3 分片机制

分片机制是范围索引中至关重要的一部分。它通过分片拆分和合并算法，动态管理索引的结构，以保证查询效率和数据的均衡分布。引入分片机制的主要目的是为了提高范围查询的效率和系统的可扩展性。

在没有分片机制的情况下，所有数据都会存储在一个单一的结构中，这会导致：在单一结构中进行范围查询时，需要遍历所有数据才能找到满足条件的项，这对于大规模数据来说效率极低。并且随着数据量的增加，单一结构会变得非常庞大，内存管理和数据操作变得复杂。

RangeShard类用于表示分片倒排索引中的单个分片，其主要包含以下属性和方法：

* **start 和 end：分片的起始值和结束值，定义分片的数值范围。**
* **index：**分片内部的倒排索引，使用 NavigableMap<Long, List<Integer>> 存储数值和文件 ID。NavigableMap是一个Java中的有序的映射表，可以按照键的自然顺序或者构造函数提供的Comparator进行排序。它还提供了导航功能，可以方便地获取键值在范围内的一些特定元素。
* **contains 和 overlaps 方法**：用于判断某个数值是否在分片范围内，以及某个范围是否与分片重叠。
* **add 和 remove 方法**：用于向分片中添加和移除数值及其对应的文件 ID。
* **search 方法**：在分片范围内进行数值查询，返回匹配的文件 ID 列表。
* **merge 方法**：将两个相邻或重叠的分片合并为一个分片。
* **split方法**：当分片大小超过阈值时，将分片拆分为两个较小的分片，以保持查询效率。

#### 4 倒排索引的构建算法设计

倒排索引的构建算法包括对基本倒排索引和分片倒排索引的构建。其核心任务是将文件元数据添加到相应的索引结构中，以支持高效的查询操作。

* 基本倒排索引构建算法

基本倒排索引用于存储离散的字符串类型元数据。构建算法的步骤如下：

1. 初始化索引结构：创建一个 Map<String, Map<String, List<Integer>>> 类型的索引结构。
2. 遍历文件元数据：对每个文件的元数据进行遍历处理。
3. 添加元数据到索引：对于每个元数据类型（如 ext 和 owner），检查其值是否已经存在于索引中。
4. 如果值不存在，则创建一个新的列表，并将文件 ID 添加到列表中。如果值已存在，则直接将文件 ID 添加到对应的列表中。

伪代码如下

initialize index structure: index = {}

for each file in file system:

for each metadata type in ['ext', 'owner']:

metadata\_value = file.metadata[type]

if metadata\_value not in index[type]:

index[type][metadata\_value] = []

index[type][metadata\_value].append(file.id)

* 分片倒排索引构建算法

分片倒排索引用于存储连续的数值类型元数据。构建算法的步骤如下：

1. 初始化分片索引结构：创建一个 Map<String, List<RangeShard>> 类型的分片索引结构。
2. 遍历文件元数据：对每个文件的元数据进行遍历处理。
3. 添加元数据到分片：
   1. 对于每个数值元数据类型（如 size、creation 和 mod），在相应的分片列表中查找合适的分片。
   2. 如果找到合适的分片，则将数值和文件 ID 添加到该分片中。如果没有找到合适的分片，则创建一个新的分片，并将数值和文件 ID 添加到新分片中。
   3. 如果分片的大小超过阈值，则进行分片拆分。
   4. 定期合并相邻或重叠的分片，以保持分片的均衡性。

# 四、测试实验

## 4.1 实验环境

实验在一台配备有16 GB RAM和Intel i7-9700K CPU的计算机上进行。操作系统为Ubuntu 20.04，Java版本为JDK 11。文件系统目录管理和倒排索引模块均使用Java编写，并在上述环境中运行。

## 4.2 实验数据生成

为了全面评估文件系统的性能，我们设计了一系列实验数据生成过程，确保数据分布和文件系统结构的多样性。实验数据生成过程包括以下几个步骤：

1.**初始化目录结构：**我们首先在根目录下随机创建多个子目录，子目录数量根据总文件数量和每个目录平均包含的文件数量决定。每个子目录的路径和属性（如所有者和权限）随机生成。

2.**创建文件：**在每个目录中随机创建若干文件，直到达到预定的总文件数量。文件的路径、所有者、扩展名和大小均随机生成，以模拟真实文件系统中的多样性。

3.**创建测试文件：**我们在固定的目录中创建两个测试文件，用于后续的修改和删除操作。这些文件位于“/dir0”目录下，文件名为“file1.txt”和“file2.txt”，文件大小随机分布在100到10000字节之间。

生成的目录树结构如下图所示

/ (INodeDirectory)

├── dir0 (INodeDirectory)

│ ├── file1.txt (INodeFile)

│ ├── file2.jpg (INodeFile)

│ └── dir1 (INodeDirectory)

│ └── file3.png (INodeFile)

├── dir2 (INodeDirectory)

│ └── file4.doc (INodeFile)

└── dir3 (INodeDirectory)

├── file5.pdf (INodeFile)

└── dir4 (INodeDirectory)

└── file6.txt (INodeFile)

每个文件的属性如下：

* **所有者**：从 "owner1" 到 "owner5" 中随机选择。
* **扩展名**：从 "txt"、"jpg"、"png"、"doc"、"pdf" 中随机选择。
* **文件大小**：在 100 到 10000 字节之间随机选择。

生成的测试数据用于评估系统的内存使用情况、查询性能、写入性能和并发读写性能。

## 4.3 内存使用评估

为了评估生成测试数据和构建倒排索引所需的内存使用情况，设计了详细的实验步骤并记录了关键数据。具体步骤如下：

**1.记录初始内存使用情况**：在进行任何数据生成操作之前，首先通过Java运行时环境获取并记录初始内存使用情况。这一步骤确保了我们可以准确评估数据生成和索引构建对内存的影响。

**2.生成测试数据并计算目录树所占内存**：按照设定的参数生成测试数据，参考4.1节。在生成数据后，再次通过Java运行时环境获取当前的内存使用情况，并计算生成数据所占用的内存。

**3.构建倒排索引并计算倒排索引所占内存**：在生成的测试数据基础上，构建倒排索引。构建索引后，再次通过Java运行时环境获取内存使用情况，并计算索引构建所占用的内存。

内存使用评估的结果如下：

表 2 不同文件规模的内存占用表

|  |  |  |
| --- | --- | --- |
| 生成文件总数量（个） | 目录树内存占用（MB） | 倒排索引内存占用（MB） |
| 10000 | 4.49 | 2.06 |
| 50000 | 23.52 | 6.58 |
| 100000 | 48.75 | 11.67 |
| 150000 | 72.67 | 17.19 |
| 200000 | 97.08 | 23.13 |
| 250000 | 121.52 | 29.75 |

## 4.4 查询性能评估

为了评估倒排索引在处理查询操作时的性能，我们设计并进行了**精确查询**和**范围查询**实验。精确查询适用于离散的元数据项，比如文件所有者、文件后缀名等，其数据结构是3.3.1中所定义的基本倒排索引（index）；范围查询适用于连续的元数据项，比如文件大小、创建时间、修改时间等，其数据结构是3.3.1中所定义的分片倒排索引（rangeIndex）；实验目的是比较基于目录树的深度优先搜索的查询方法和基于倒排索引的查询方法对于两种查询（精确查询和范围查询）在执行时间和结果准确性方面的表现。

对于查询耗时的评估，我们采用了System.nanoTime()方法来测量精确查询和范围查询的执行时间。nanoTime()是Java中的一个高精度计时方法，返回的是当前Java虚拟机启动后的纳秒数。它主要用于测量时间间隔，而不是当前的绝对时间，与System.currentTimeMillis()相比，它具有更高的精度和更好的测量准确性。

对于结果准确性的评估，我们的查询返回的是一个List<Integer>类型的几何，集合内容为符合查询条件的id集合，我们比较了基于目录树的深度优先搜索查询与基于倒排索引的查询返回的集合的长度，作为结果准确性的评估量度。

### 4.4.1精确查询

我们首先进行精确查询实验，通过对文件后缀名为“txt”的文件进行查询，来评估系统的查询性能。

精确查询的结果如下：

表 3 不同文件规模的精确查询耗时和结果对比表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 目录树文件总数量（个） | 查询耗时（ms） | | 查询结果数量（个） | |
|  | 基于目录树 | 基于倒排索引 | 基于目录树 | 基于倒排索引 |
| 10000 | 2.2024 | 0.0585 | 1993 | 1993 |
| 50000 | 7.6683 | 0.0588 | 10012 | 10012 |
| 100000 | 10.3298 | 0.0406 | 19849 | 19849 |
| 150000 | 15.5566 | 0.0418 | 30164 | 30164 |
| 200000 | 18.3348 | 0.0441 | 40257 | 40257 |
| 250000 | 23.3429 | 0.0417 | 49813 | 49813 |

从实验结果我们可以看出：基于目录树的查询耗时随文件数量增加而增加，这是因为目录树查询是深度优先搜索，需要遍历所有节点，其复杂度为O(n)。随着文件数量的增加，遍历的时间也会相应增加。相反，基于倒排索引的查询耗时几乎不受文件数量的影响，这是因为倒排索引将文件元数据预先索引化，使得查询时间复杂度接近O(1)。查询过程只需查找索引即可获取结果，避免了对整个目录树的遍历。

实验结果表明：基于倒排索引的查询方法在处理精确查询时具有显著的性能优势，尤其是在大规模数据环境下，其查询效率远远优于基于目录树的查询方法。此外，两种方法在查询结果的准确性上没有差异，均能够正确地返回所有符合条件的文件。因此，在需要频繁执行精确查询的场景中，采用倒排索引可以极大地提升系统的查询性能。

### 4.4.2范围查询

我们接着进行范围查询实验，通过对文件大小范围在600-8000的文件进行查询，来评估系统的查询性能。由于实验生成的元数据的文件大小的范围位于100-10000.我们在本实验中设置了分片倒排索引每个分片的大小阈值为1000.

我们在实验程序的一次测试中的输出了如下分片结果：

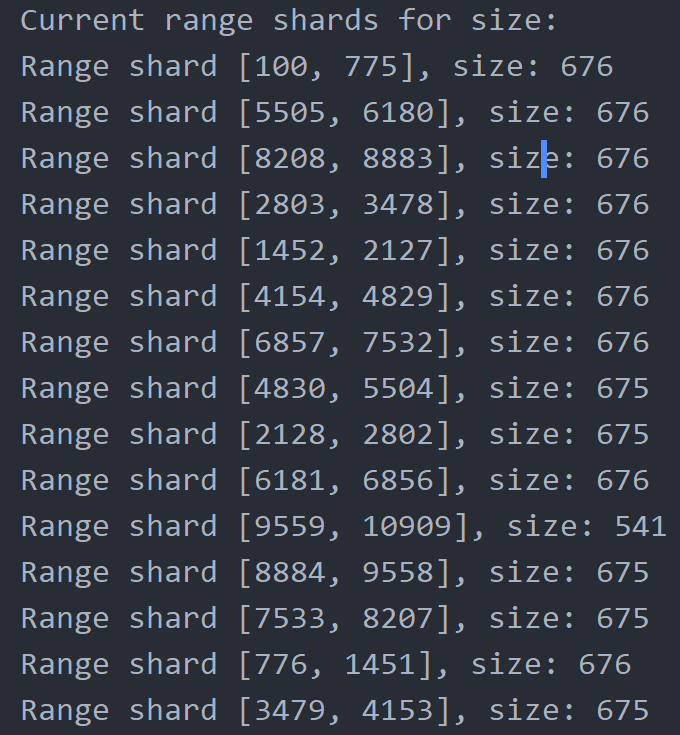


图 3 分片结果日志图

范围查询的结果如下：

表 4 不同文件规模的范围查询耗时和结果对比表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 目录树文件总数量（个） | 查询耗时（ms） | | 查询结果数量（个） | |
| 基于目录树 | 基于倒排索引 | 基于目录树 | 基于倒排索引 |
| 10000 | 2.7471 | 1.4796 | 7379 | 7379 |
| 50000 | 6.2786 | 1.738 | 36965 | 36965 |
| 100000 | 12.0506 | 1.7347 | 73992 | 73992 |
| 150000 | 16.4432 | 2.249 | 110701 | 110701 |
| 200000 | 21.1061 | 2.4728 | 147955 | 147955 |
| 250000 | 26.8764 | 2.9555 | 185284 | 185284 |

从实验结果我们可以看出：基于目录树的范围查询耗时同样随着文件总数量的增加呈现出线性增长的趋势，而基于倒排索引的范围查询耗时则较为平稳，尽管有所增加，但增幅远小于基于目录树的查询耗时。在每个实验中，基于目录树和基于倒排索引的查询结果数量相同，均为符合范围条件的文件数量。这表明两种查询方法在结果准确性上是一致的，即查询结果的正确性不受查询方法的影响。与精确查询相比，范围查询的耗时相对较长，特别是基于倒排索引的范围查询耗时明显高于精确查询。然而，基于倒排索引的范围查询在文件数量较多的情况下，仍然比基于目录树的查询方法具有显著的性能优势。

### 4.4.3并发查询

本实验旨在评估文件系统在高并发环境下的性能表现，重点关注目录树和倒排索引在精确查询和范围查询中的开销对比。在并发精确查询测试中，首先创建一个100大小的线程池，在每个查询线程中，随机生成查询条件（例如文件扩展名为 "txt"）进行查询，统计并计算平均查询时间来评估系统的查询性能。并发范围查询测试中，同样创建一个100大小线程池，在每个查询线程中，设置文件大小范围作为查询条件，统计并计算平均查询时间来评估系统的查询性能。

本实验中使用Java的**ExecutorService**类用于创建和管理线程池，保证线程的复用和资源的有效管理。使用CountDownLatch用于同步控制，确保所有并发查询线程在统计之前完成。使用**System.nanoTime()**用于记录精确的时间戳，以便计算查询的耗时。

实验结果如下：

表 6 不同文件规模的并发精确查询和范围查询耗时对标表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 目录树文件总数（个） | 并发查询平均耗时（ms） | | | |
| 精确查询 | | 范围查询 | |
| 基于目录树 | 基于倒排索引 | 基于目录树 | 基于倒排索引 |
| 1000 | 0.251633 | 0.023011 | 0.813364 | 0.354582 |
| 5000 | 3.599249 | 0.010785 | 4.70018 | 0.779878 |
| 10000 | 5.75448 | 0.009081 | 6.071451 | 2.612009 |
| 50000 | 8.963061 | 0.252293 | 17.343668 | 3.419272 |
| 100000 | 17.490598 | 0.311714 | 141.979865 | 8.06182 |

从实验结果我们可以看出：基于倒排索引的查询方式在高并发环境下具有更优的性能，尤其是在文件数量较多的情况下，其查询耗时远低于基于目录树的查询方式。因此，在设计和实现高并发文件系统时，采用倒排索引可以显著提高系统的查询性能，减少查询时间，提高系统的响应速度和用户体验。

## 4.5写入性能评估

为了评估倒排索引在处理查询操作时的性能，我们通过测量将文件插入目录树和倒排索引过程中所消耗的时间，设计并进行了单个文件插入性能评估和大批量文件插入性能评估实验。

在单个文件插入性能评估中，我们指定了文件在目录树中的存储路径以及随机生成元数据项，插入一个新的文件到文件系统中，分别测量其插入到目录树和倒排索引中所消耗的时间。在大批量文件插入性能评估实验中，我们通过插入大量新文件到文件系统中，分别测量其插入到目录树和倒排索引中所消耗的总时间。

对于查询耗时的评估，我们继续采用了System.nanoTime()方法来测量精确查询和范围查询的执行时间。

对于结果准确性的评估，我们应用4.4中的查询性能测试方法，将插入前后查询结果数量的差值与插入文件的数量进行比较。

关于实验数据的生成，我们此次测试设置生成总文件数为100000个。

在单个文件插入性能评估实验中，我们进行了50次实验取平均值，结果如下：

表 7 目录树和倒排索引写入耗时情况表

|  |  |  |
| --- | --- | --- |
| Insert to directory tree time（ms） | Insert to inverted index time（ms） | Is insert successful？（bool） |
| 7\*10-4 | **6.6944** | **True** |

从实验结果我们可以看出：插入一个文件到目录树中的时间非常短，而插入到倒排索引中的时间相对较长。这是因为目录树的插入操作主要涉及节点的添加，而倒排索引的插入操作需要更新索引数据结构，涉及较多的计算和内存操作。

在大批量文件插入性能评估实验中，，结果如下：

表 8 不同规模文件写入目录树和倒排索引写入耗时对比表

|  |  |  |
| --- | --- | --- |
| 插入对象  插入文件  数量 | 插入耗时（ms） | |
| **Insert to directory tree（ms）** | **Insert to inverted index time（ms）** |
| 1000 | 1.9064 | 0.421 |
| 5000 | 63.756 | 2.1532 |
| 10000 | 280.7063 | 5.3366 |
| 25000 | 1672.1564 | 49.6799 |
| 50000 | 9875.4546 | 157.4723 |

从实验结果我们可以看出：随着插入文件数量的增加，插入到目录树和倒排索引中的时间都呈现显著的增长趋势。这是由于文件数量增加导致的插入操作复杂度和内存占用的增加。然而在大批量插入测试中，插入倒排索引比插入目录树耗时短，这与单个文件插入性能评估实验得到的结论思路是矛盾的。分析原因如下：目录树的插入操作需要从根节点逐级查找或创建子节点，直到找到目标目录。这意味着每次插入操作都可能涉及多个内存位置的访问，这些位置分布在不同的内存块中，内存访问模式较为分散。频繁的跨块访问导致缓存命中率降低，增加了内存访问延迟。倒排索引的插入操作主要涉及将文件元数据（如扩展名、所有者、文件大小等）添加到相应的索引条目中。由于倒排索引的数据结构设计合理，插入操作的内存访问模式较为连续。特别是在索引条目集中存储的情况下，多个插入操作可以在同一内存块中进行，减少了跨块访问的频率。连续的内存访问模式使得数据可以更好地利用缓存，提高缓存命中率，从而减少内存访问延迟。

# 五、进阶功能设计及测试

## 5.1 读写锁设计

在多线程环境下，如何保证**数据的一致性**和**并发性能**是一个关键问题。本项目中，倒排索引和目录树是两个核心的数据结构，涉及大量的读写操作。高并发读写的场景下，需要确保单一索引的并发正确性，还要保证多个索引之间的一致性。例如，当目录树中的文件元数据被修改时（如修改了文件的扩展名），倒排索引中的相应数据也必须同步更新，以保持数据的一致性。

设计目标有如下三点：

* **数据一致性**：确保在多线程环境下，所有读写操作都能保证数据的一致性。
* **高并发性能**：尽可能提高并发性能，减少锁的争用，优化读写操作的效率。
* **细化锁粒度**：通过细化锁的粒度，减少锁的争用，提高系统的整体性能。

本项目中，读写锁的设计主要应用在两个关键部分：倒排索引（InvertedIndex）和目录树（FSDirectory）。为了实现高效的并发控制，我们使用了Java中的ReadWriteLock和ReentrantReadWriteLock来管理读写锁，同时更换倒排索引中的外层索引的HashMap为线程安全的ConcurrentHashMap**。**

### 5.1.1 读写锁的介绍与实现

读写锁（ReadWriteLock）是一种特殊的锁机制，旨在优化多线程环境下的读写操作的并发性能。传统的互斥锁（如 synchronized 或 ReentrantLock）在同一时间只允许一个线程访问受保护的资源，无论是读操作还是写操作。然而，在许多应用场景中，读操作远比写操作频繁，并且多个读操作之间并不互相干扰。为了提高并发性和系统性能，读写锁应运而生。

读写锁主要分为两种：

* **读锁（Shared Lock）**：允许多个线程同时读取共享资源，但任何时候都不能有写操作在进行。
* **写锁（Exclusive Lock）**：允许一个线程独占访问资源，进行写操作，同时禁止任何读操作或其他写操作。

在多线程环境下，读写锁具有以下几个显著优点：一方面读写锁允许多个线程同时持有读锁，因此可以大幅提高读操作的并发性能，尤其是在读多写少的应用场景中。当没有任何线程持有读锁或写锁时，写操作可以获得写锁并独占资源。这确保了写操作的原子性和数据一致性。另一方面传统的互斥锁在读多写少的情况下会导致大量的锁竞争，进而影响系统性能。读写锁通过区分读锁和写锁，减少了读操作之间的竞争，从而提高了系统的吞吐量。

本项目中，读写锁的设计和实现主要依赖于ReadWriteLock接口及其具体实现类ReentrantReadWriteLock。ReadWriteLock是Java中的一个接口，用于管理一组锁，一个用于读操作（共享锁），一个用于写操作（独占锁）。它主要提供了以下两个方法：Lock readLock()：返回一个读锁，用于保护读操作；Lock writeLock()：返回一个写锁，用于保护写操作。ReadWriteLock通过读写分离的机制，使得多个线程可以同时读取资源，而写操作则是独占的，只有当没有线程读取或写入资源时，写操作才能进行。ReentrantReadWriteLock是ReadWriteLock接口的具体实现类，提供了读写锁的重入特性。主要特点是允许同一个线程多次获取读锁或写锁，而不会发生死锁，同时支持公平锁和非公平锁模式：在公平模式下，锁按照请求的顺序分配。在非公平模式下，锁的分配顺序不一定严格按照请求的顺序，可能会提高吞吐量。

在项目的具体实现中，读写锁主要用于管理目录树（FSDirectory）和倒排索引（InvertedIndex）的并发操作。我们在原有的FSDirectory类的基础上，添加了lock字段，用于保护整个目录树的操作，即对整个目录树数据结构加锁，使用ReentrantReadWriteLock实现。对于FSDirectory类提供的方法，包括创建文件、删除节点、更新文件、在目录树中搜索、构建倒排索引等等，均先获取读（写）锁，完成相应操作后释放读（写）锁。对于倒排索引的修改，我们增加了锁结构的细粒度，在原有的InvertedIndex类的基础上，添加了Map<String, ReadWriteLock> 类型的字段locks，我们没有对整个倒排索引的数据结构进行加锁，而是对倒排索引的子索引（如文件后缀名、文件大小、创建时间等）进行加锁，这样每个索引键都有自己的锁，多个线程可以同时操作不同的索引，减少锁争用，提高系统并发处理能力。同样的，对于InvertedIndex类提供的方法，包括添加索引文件、删除索引节点、搜索等等，均先获取对于索引键的读（写）锁，完成相应操作后释放读（写）锁。

### 5.1.2 线程安全的Hashmap的使用

我们设计的倒排索引中分为index和rangeindex两部分。在针对多线程的读写的情况下，我们改进了他们的数据结构，将非线程安全的HashMap替换为线程安全的ConcurrentHashMap。

ConcurrentHashMap 是 Java 提供的一种线程安全的哈希表实现，它在多线程环境中提供了更好的并发性能和安全性。ConcurrentHashMap 允许多个线程并发地读取和写入，而不会造成阻塞或性能下降。这使得它非常适合高并发环境下的数据结构使用场景。

ConcurrentHashMap 采用了分段锁（Segmented Locking）机制来实现并发访问。ConcurrentHashMap 将哈希表划分为多个段（Segment），每个段独立地管理一部分哈希桶（Bucket）。每个段有自己的锁，线程只需要锁定当前段，而不是整个哈希表。这种设计允许多个线程并发地访问不同的段，从而提高并发性能。读操作几乎不需要加锁，而写操作则仅仅锁定特定段的锁。

在高并发环境中，传统的 HashMap 或 Hashtable 由于其线程安全机制的不同，会面临以下问题：HashMap 本身不是线程安全的，如果在多线程环境中使用，可能会导致数据不一致、死循环等问题。Hashtable 是线程安全的，但其每个方法都使用同步方法（synchronized），在高并发情况下会导致严重的性能瓶颈，因为任何读写操作都需要获取全局锁。相比之下由于分段锁的设计，ConcurrentHashMap 允许更高的并发度，多个线程可以同时访问不同的段，从而大大提高了并发性能，同时所有操作都保证线程安全，避免了数据不一致的情况。

### 5.1.3 多线程测试

本测试旨在对比在并发环境下倒排索引有无读写锁结构的数据一致性情况。

本实验中使用Java的**ExecutorService**类用于创建和管理线程池，创建了大小为15的线程池，并将其平均分为3组，分别是并发写线程组、并发读线程组、并发新增文件线程组，每次线程操作完成后，就打印线程ID和操作结果，然后线程随机阻塞一段时间（0-1500毫秒），以模拟实际使用中的不均匀访问模式。

我们使用4.2中实验数据生成的方法生成了100000个文件规模的文件系统。观察打印日志，来对比有无读写锁的倒排索引的数据一致性问题。

无读写锁时，我们进行了20次测试，选取其中一次的实验日志的部分内容如图所示：

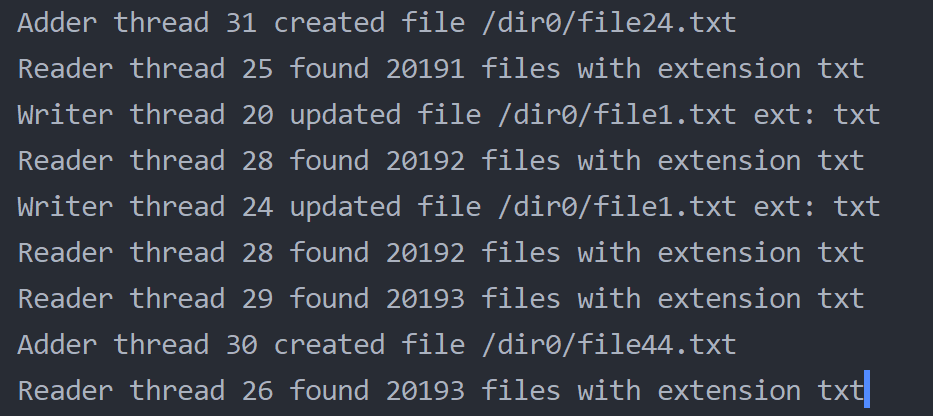


图 4 多线程测试日志图

我们可以看到在线程24更新了文件后缀名操作之后，线程30创建文件操作之前，读线程返回的文件数量应该是相同的，而读线程28和29返回的文件数却不一致，说明数据一致性出现问题。

使用读写锁后，我们也进行了20次测试，检查日志均为合理的结果，说明读写锁对于应对并发情景起到了作用。

### 5.1.4其他方法

关于在多线程环境下，保证数据的一致性和并发性能的问题，还有一些其他方法可以尝试使用：使用**版本控制机制**保证数据一致性；令**索引和目录树的异步更新**减少写操作对实时性的要求，提高系统整体吞吐量。

**版本控制机制**：在每个倒排索引的分片和目录树节点上引入版本号字段，用于记录当前数据的版本。读取倒排索引和目录树节点时，记录当前版本号。读取操作完成后，再次检查版本号是否发生变化，如果没有变化，则读取操作有效；如果发生变化，则需要重新读取。写入倒排索引和目录树节点时，先检查当前版本号是否与预期一致，如果一致则更新数据并递增版本号；如果不一致，则说明数据已经被其他线程修改，需重试或放弃。

**索引和目录树的异步更新:** 为了实现索引和目录树的异步更新，可以利用消息队列和异步处理机制，将对索引和目录树的更新操作从主线程分离出来。为索引和目录树的更新操作引入消息队列，主线程将更新任务放入队列中，异步线程从队列中读取并执行任务。创建异步线程池，专门用于处理消息队列中的任务。异步线程从消息队列中读取任务并执行对应的更新操作。

## 5.2 倒排索引压缩

### 5.2.1相关算法研究

倒排索引虽然改善了元数据查询，但索引自身引入了新的存储空间开销。倒排索引的基本结构包括词典和倒排链表。词典记录元数据项和对应的倒排链表指针，倒排链表记录元数据项在对应的文档索引信息。倒排链表压缩的核心是对文档号之间的差值（d-gap）和词频（freq）进行编码，以减少存储空间和提高查询效率。在实际应用中，词典文件比起倒排文件来说通常要小得多，所以当前研究人员主要集中于对倒排链表压缩算法的研究。

传统的符号压缩算法如 Huffman 编码、算术编码、PPM等半静态或者自适应压缩方法需要预先知道被压缩符号的统计信息并保留其压缩模型。当前倒排索引压缩算法的研究主要集中在不需要统计信息的轻量级压缩倒排链表的压缩算法按码字对齐的方式可以分为:比特对齐(bit-aligned)压缩算法、字节对齐(byte-aligned) 压缩算法和字对齐(word-aligned)压缩算法。它们在压缩效率、解压速度和实现复杂度上各有优劣。

表 9 不同压缩算法的性能对比对比表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 压缩效率 | 解压速度 | 实现复杂度 | 经典算法 |
| 比特对齐 | 高 | 慢 | 高 | Gamma、Delta |
| 字节对齐 | 中等 | 适中 | 中等 | Varint、Group Varint |
| 字对齐 | 低 | 高 | 低 | Simple-9、For |

### 5.2.2 Varint压缩算法

本项目中，我们使用引入了字节压缩算法中的Varint（Variable-length integer）编码。Varint是一种用于压缩整数数据的编码算法。其主要优点在于对小整数进行高效压缩，同时保持较大的整数可被编码。Varint通常用于需要存储或传输大量整数的系统中，例如数据库、网络协议和文件格式等。Varint编码的一个关键特性是使用更多的字节来存储较大的整数，而较小的整数使用较少的字节。

Varint通过使用可变长度的字节序列来表示整数。每个字节的最高位（即第8位）作为延续位（continuation bit），用于指示后续字节是否仍然属于同一个整数。具体来说：

* 若最高位为0，则该字节为Varint编码的最后一个字节，其余7位用于表示数据。
* 若最高位为1，则该字节之后仍有更多字节属于同一个整数。

例如，一个整数300的二进制表示为100101100。用Varint编码这个整数的步骤如下：

1. **将300转换为二进制表示**：100101100
2. **从最低有效位开始，每7位一组进行分割**：00101100 和 00000010
3. **设置延续位**：
   1. 00101100（最后一组，设置最高位为0）：00101100
   2. 00000010（还有后续组，设置最高位为1）：10000010
4. **合并成字节序列**：10000010 00101100

解码时，读取字节序列10000010 00101100，解析出原始整数300。

### 5.2.3 Varint算法的java实现

在本项目中初始化文件id的代码如下，id初始化的形式为线程安全地自增。

static final AtomicInteger idGenerator = new AtomicInteger();

int id;

this.id = idGenerator.incrementAndGet();

文件id号的存储类型为int类型，且索引结构类型为Map<String, Map<String, List<Integer>>>，我们首先对索引结构进行修改，修改后的类型为Map<String, Map<String, byte[]>>,接下来需要进行编码与解码代码的编写。

**编码部分实现**：

public static void writeUnsignedVarInt(int value, ByteArrayOutputStream out) throws IOException {

while ((value & 0xFFFFFF80) != 0L) {

out.write((value & 0x7F) | 0x80);

value >>>= 7;

}

out.write(value & 0x7F);

}

**代码说明：**

* 0xFFFFFF80 是一个掩码，用于检测高于7位的所有位。如果value的高位部分不为0，表示value大于或等于128，需要继续处理。
* 在循环内：取出value的低7位，将最高位设置为1，表示后面还有更多字节，将处理后的字节写入输出流，将value无符号右移7位，处理下一个7位组。
* 结束循环后，处理最后7位，将最高位设置为0，表示这是最后一个字节。最后写入输出流。

**解码部分实现**：

public static int readUnsignedVarInt(byte[] bytes, int[] offset) {

int value = 0;

int i = 0;

int b;

while (((b = bytes[offset[0]++]) & 0x80) != 0) {

value |= (b & 0x7F) << i;

i += 7;

}

return value | (b << i);

}

**代码说明：**

**1. 初始化**

* int value = 0;：初始化结果值为0。
* int i = 0;：初始化位移量（表示当前处理的是第几个7位组）。
* int b;：用于存储当前读取的字节。

**2. while循环**

* 从bytes数组中读取一个字节并递增offset[0]（即更新偏移量）。
* b & 0x80：检查最高位，如果最高位为1，表示后面还有更多字节。

**3. 循环体内的操作**

* b & 0x7F：取出当前字节的低7位。
* (b & 0x7F) << i：将这7位左移i位，合并到value中。
* value |= ...：通过按位或运算，将处理后的7位加入到value中。
* i += 7：增加位移量7，准备处理下一个7位组。

**4. 结束循环后的操作**

* 循环结束时，b中的最高位为0，表示这是最后一个字节。
* 将当前字节的所有位（包括第8位0）左移i位并合并到value中，返回最终结果。

### 5.2.4 内存占用测试

我们修改了索引结构后继续沿用4.3的测试设计。实验结果如下：

表 10 不同文件规模倒排索引压缩前后内存占用对比表

|  |  |  |
| --- | --- | --- |
|  | 倒排索引的内存占用（MB） | |
| 文件总数量（个） | **压缩前** | **压缩后** |
| 5000 | 1.24 | 0.59 |
| 10000 | 2.06 | 0.83 |
| 50000 | 6.86 | 2.11 |
| 100000 | 11.66 | 2.88 |
| 150000 | 17.19 | 4.09 |
| 200000 | 22.96 | 5.45 |
| 250000 | 29.37 | 6.76 |

实验结果说明：使用 Varint 编码对倒排索引进行压缩后，内存占用显著减少。即使在文件数量达到 250,000 的情况下，压缩后的内存占用仅为压缩前的约 23%，这表明 Varint 编码在减少存储空间方面具有显著的优势。并且随着文件数量的增加，压缩后的内存占用减少更加明显。这是因为 Varint 编码对较小的整数具有更高的压缩效率，而在倒排索引中，文档ID往往是相对较小的整数。

# 六、总结

本文通过对文件系统的元数据查询进行优化，提出并实现了一种基于倒排索引的高效查询方法。我们在目录树结构基础上引入了倒排索引，并采用Varint编码进行压缩，显著减少了索引的存储空间。为了应对高并发读写的挑战，我们设计了细粒度的读写锁机制，并使用线程安全的ConcurrentHashMap替代传统的HashMap。此外，本文还探讨了异步更新和版本控制机制在提高系统吞吐量和保持数据一致性方面的应用。实验结果验证了所提方案在大规模文件系统中的有效性和优越性，为实现高效的元数据管理提供了有力支持。

# 参考文献

[1]刘晓宇,夏立斌,姜晓巍,等.HDFS分级存储系统元数据管理方法的研究[J].计算机工程与应用,2023,59(17):257-265.

[2] H. Dai, Y. Wang, K. B. Kent, L. Zeng and C. Xu, "The State of the Art of Metadata Managements in Large-Scale Distributed File Systems — Scalability, Performance and Availability," in IEEE Transactions on Parallel and Distributed Systems

[3]王立新. 并行文件系统元数据管理研究[D].国防科学技术大学,2019.

[4]姜琨,朱磊,宋省身,等.倒排索引压缩算法研究综述[J].小型微型计算机系统,2020,41(04):715-723.

[5]罗欣剑. 大规模分布式存储系统中的元数据管理方案研究[D].上海交通大学,2020.DOI:10.27307/d.cnki.gsjtu.2019.002034.

[6]陈友旭.分布式文件系统中元数据管理优化[D].中国科学技术大学,2019.DOI:10.27517/d.cnki.gzkju.2019.000021.

[7]张晓艳. 分布式元数据管理系统的设计与实现[D].电子科技大学,2020.DOI:10.27005/d.cnki.gdzku.2020.002600.