分类号_____ 学号 M201672205

学校代码 10487 密级_____

华中科技大学

硕士学位论文

Hadoop 平台下基于 HDFS 的小文件存储问题的优化与实现

学位申请人： 罗 青

学 科 专 业： 软件工程

指 导 教 师： 傅邱云教授

答 辩 日 期： 2019 年 1 月 18 日

A Dissertation Submitted in Partial of Fulfillment of the Requirements for the Degree of Master of Engineering

Optimization and implementation of small file storage in HDFS under Hadoop platform

Candidate : Luo Qing

Major : Software engineering

Supervisor : Prof. Fu Qiuyun

Huazhong University of Science & Technology

Wuhan, Hubei 430074, P. R. China

January, 2019

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：罗青

日期：2019 年 1 月 20 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于 保密□，在_____年解密后适用本授权书。

不保密☑。

（请在以上方框内打“√”）

学位论文作者签名：罗青

日期：2019 年 1 月 20 日

指导教师签名：

日期：2019 年 1 月 20 日

摘 要

大数据技术随着互联网的发展及信息量爆炸增长的趋势应运而生。面对异常庞大的数据，多种分布式文件系统为大数据的存储提供了解决方案。其中 Hadoop 由于自身高扩展性、高可靠性等优点被业界广泛使用。HDFS 作为 Hadoop 的核心组件，为处理大数据提供了文件存储服务。然而 HDFS 更擅长处理流式的大文件，面对海量小文件存储时的表现不佳。

本文为了解决 HDFS 存储小文件效率低下的问题，对 Hadoop 架构和 HDFS 存储文件的流程进行详细分析，提出了引入多级处理模块 MPM（Multilevel Processing Module for Small Files）的方案。该方案首先通过文件预处理模块，对系统中发出操作请求的文件进行过滤，筛选 4.35MB 以下的文件为小文件，并将其按文件扩展名进行初步分类。随后文件合并模块会将预处理后的小文件合并成尽可能少的大文件，以减少系统 NameNode 内存负载。为了提高小文件的查询速度，方案中除了利用小文件创建时间和小文件扩展名建立的二级索引模块，还引入了基于用户常用文件的预取和缓存模块。最后，针对系统长时间运行导致的碎片问题，当系统满足设定条件时，碎片整理模块会对合并文件的空白空间进行清理，以提高系统空间的利用率。

本文将提出的 MPM 方案与三种 HDFS 现有存储方案：原生存储方案、HAR 文件归档方案、Sequence File 方案进行实验对比。当存取数量为 100000 的文件时，MPM 方案可为系统节省 95.56%的内存占用，空间利用率高达 99.92%。同等条件下，与原生存储方案相比，MPM 方案的写入速率是未优化前的两倍；由于合并机制步骤更多，写入耗时只降低了 31%。读取速率提升了 2.25 倍左右，读取耗时是所有方案中最低的。实验结果表明，MPM 方案对 HDFS 的存储性能改善明显。大幅减少了系统中的文件数量，有效降低 NameNode 内存负载，提高了系统内存利用率，实现了高速率的小文件读写性能。

关键词：HDFS、海量小文件、文件合并、二级索引

Abstract

With the development of the Internet, the big data technology has emerged as the trend of information explosion. In the face of extremely large data, a variety of distributed file systems provide solutions for the storage of big data. Hadoop is widely used in the industry due to its advantages of high scalability and high reliability. HDFS, as the core component of Hadoop, provides file storage service for processing big data. HDFS, however, is better at handling large, streaming files, and does not perform well when storing large amounts of small files.

In order to solve the problem of low efficiency in HDFS storage of Small Files, this paper analyzes the Hadoop architecture and the process of HDFS storage of Files in detail, and proposes a scheme of introducing Multilevel Processing Module (MPM) for Small Files. This scheme firstly filters the files that send operation requests in the system through the file preprocessing module. Files under 4.35MB are screened as small files and are preliminarily classified according to file extensions. The file merge module then merges the pre-processed small files into as few large files as possible to reduce the system NameNode memory load. In order to improve the query speed of small files, the scheme not only USES the creation time of small files and the secondary index module established by the extension of small files, but also introduces the prefetch and cache module based on the common files of users. Finally, aiming at the fragmentation problem caused by the long running of the system, when the system meets the set conditions, the defragmenting module will clear the blank space of the merged files to improve the utilization rate of the system space.

This paper compares the proposed MPM scheme with three existing HDFS storage schemes: native storage scheme, HAR File archive scheme and Sequence File scheme. When 100,000 files are accessed, the MPM scheme can save the system 95.56% in memory usage and 99.92% in space utilization. Under the same conditions, compared with the native storage scheme, the write rate of the MPM scheme is twice as high as that before optimization. Because of the more steps in the merge mechanism, the write time is reduced by only 31%. The reading rate is improved by about 2.25 times, and the reading time is the lowest among all schemes. Experimental results show that MPM scheme improves HDFS

storage performance significantly. Greatly reduce the number of files in the system, effectively reduce the NameNode memory load, improve the system memory utilization, and achieve high rate of small file read and write performance.

Key words: HDFS, lots of small files, file merge, secondary index

目 录

1 绪论

1.1 研究背景及意义

随着移动互联网的普及和应用，全球的数据信息呈现了爆发式的增长。国际数据中心 IDC（International Data Corporation）在 2014 年发布的统计结果[1]中，前一年全球数据量共计不足 1.5ZB；而报告预测，5 年后该数字将超过 40ZB。日新月异的大数据时代促使传统的数字化产业寻求转型，人工智能、云计算等新兴产业迅速发展[2]。

几何式增长的数据量要求计算机必须拥有更大容量、处理速度更快的存储系统，传统的计算机存储系统的成长速度无法与之匹配，因此催生了大量的数据存储系统[3]。目前企业常用的主流分布式文件系统有 Hadoop[4]、Lustre[5]、MogileFS、FreeNAS、FastDFS[6]、GoogleFS[7][8]等。Hadoop 具有可靠性高、可扩展性强、存储速度快、容错率高等一系列优势[9]。又由于其可以被部署在廉价硬件上，让用户在短时间内构建一个高效处理海量数据的分布式集群，因此得到众多企业以及学术界的青睐。

高校和研究机构针对 Hadoop 系统在数据存储、资源整合、编程引擎、效率优化等方向开展了深入的研究。相关研究成果都可以在 Hadoop 开源社区进行分享和讨论。为了支持广告和搜索平台的正常运行，Yahoo[10]创建了超过 3500 个节点的 Hadoop 集群；国际知名的互联网企业“脸书”[11]也使用了上千个节点的 Hadoop 集群存储日志数据，以此支撑企业日常的数据分析，并在此基础上进行机器学习的研究。早在 2009 年，国内就提出了“大云（Big Cloud）”商务智能系统[12]的想法，可以实现高性能低成本的分布式文件存储，由中国移动的业务支撑研究所进行开发。其他互联网企业如百度，每日仅搜索引擎和网页数据产生的数据量就有 30TB，使用 Hadoop 才能满足这样庞大的文件存储需求。如今，Hadoop 已经成为了大数据存储领域中常用的数据存储系统，在数据处理的实际任务里有着举足轻重的地位。

分布式存储文件系统的出现使大数据得到处理和存储，但是庞大的信息流并不是数据存储系统所面临的唯一问题。互联网技术的出现，让每个行业都有了与互联网结合的需求，且移动应用开发成本及门槛不断降低，海量的移动应用如雨后春笋般出现。

到 2018 年 4 月为止，我国市场上监测到的移动 APP 为 414 万款，本国移动 APP 数量达到了 231 万款。其中占比最多的前四种应用分别是游戏类应用、生活服务类应用、电子商务类应用和影音播放类应用[13]。这些占据我们生活绝大部分时间的移动应用每天会产生十亿级的图片、视频、office 文档等，这些文件的体积大多大于 1KB，小于等于 10MB。比如，淘宝作为目前国内最大的电子商务网站，在 2010 年其存储的图片文件数量就接近 300 亿个，平均图片大小仅有 18KB[14]。以抖音为代表的移动端短视频平台，至 2018 年 6 月，国内日活跃用户数超过 1.5 亿，假设每位活跃用户平均每天发布一个 10s 的短视频，单个视频容量通常在 5MB-10MB 不等，那么抖音每日都需要存储 1.5 亿个总量为 715TB-1430TB 的视频小文件数据[15]，并且随用户增长及活跃度不定期增长。

面对不断增长的移动应用，以及用户使用应用过程中产生的海量小文件。Hadoop 分布式文件系统在设计之初并没有考虑到海量小文件存储带来的内存浪费等一系列问题。因为 Hadoop 中一个最小的存储单元叫做 Block[16]，默认存储文件阈值为 64MB。小文件在存储时被分割成若干个内容连续的数据文件进行存储，小于 64MB 的文件会占据节点空间但不占满整个空间。小文件数量的增多使得系统中的存储空间无法被完全利用，存在大量的内存浪费的情况。

Hadoop 系统在处理小文件数据时，存储性能和读写效率都无法维持原有水准。海量的小文件数据使得存储系统变得臃肿、缓慢甚至无法工作。通常业内将文件大小为 1KB-10MB 的数据称为小文件，数量在百万级及以上的是海量数据。由此引出分布式文件存储系统中的海量小文件问题（Lots of Small Files，简称 LOSF）[17]。LOSF 的存储问题成为业界久经不衰的研究课题。

为了解决 LOSF 带来的文件存储效率低下的问题，Hadoop 社区提出了 Hadoop Archive（HAR）归档[18]、Sequence File[19]、HDFS Federation[20]等技术方案。这几种方案的实现原理都是将小文件按照某些标准组合成一定大小的合并文件，再将其放入 HDFS 进行存储。这样可以使系统中的文件数量大幅减少，从而降低存储文件所需的节点内存和元数据数量，以此提升系统性能。但是目前的合并标准会导致例如跨块存储、合并效率不高等新的问题。可以说现有解决方案对 Hadoop 系统存储性能的改善

并不成功。

如何有效的解决 LOSF，提高 Hadoop 系统存储海量小文件时的存储性能，是现在甚至未来大数据存储领域将会一直探讨的重要议题。因此，本文的研究方向——Hadoop 平台中基于 HDFS 的海量小文件存储问题的优化与实现，具有非常重要的科研意义和实用价值。

1.2 国内外研究现状

随着数据，尤其是小文件数据的爆发式增长，国内外学术界与相关企业对海量小文件存储问题进行了广泛的研究。提出了一些针对特定文件类型或特定系统的解决方法，一定程度上缓解了 LOSF 对文件存储系统带来的影响。

针对单机环境下的词频统计[21]场景，袁玉等人对比了系统在存储不同文件类型的测试数据集时，系统的内存消耗情况、文件读写速率情况等，得到了系统对不同文件输入格式和词频统计时间的关系。结果显示通过将大量小文件合并为一个大文件可以显著提升 Hadoop 系统的存储性能。但是因为实验是在单机环境下进行的，该实验存在一定的局限性。

针对浏览器和服务集群之间的传输数据过多的情况，一种结合了网络 Web 和地理信息的系统 WebGIS 系统[22]应运而生。该系统将数据切割成小文件进行传输以最大限度的减少系统节点之间传输的数据，通常这些小文件的大小是 KB 级别的。受到 WebGIS 的启发，X Liu 等人[23]在处理海量小文件存储问题时，将小文件的合并依据设定为地理位置信息。将地理位置相近的小文件合并成大文件，并引入索引机制便于文件查找。这种方法针对拥有地理位置信息的特殊数据合并效率较高，对其他类型的数据没有可用性。另外这个方法的索引机制较为复杂，随着存储数据量的不断增多，文件读取效率反而会降低。

为了提高小文件合并后文件之间的结构相关性和逻辑相关性，文献[24]提出了一种三级缓存策略。通过预取文件元数据、索引数据以及数据本身的方法增强文件之间的结构相关性；同样在文件分组时预取三级缓存以提高逻辑相关性。并对提出的方法进行了实验验证，但由于数据集受到了限制，该方案虽然一定程度上提高了 LOSF 的存

储性能，但仍需要更多无约束的数据继续实验。

扩展的 Hadoop 分布式文件系统——EHDFS（Extend Hadoop Distributed Files System，简称 EHDFS）由 Chandrasekar S 等人[25]提出。该方案在 HDFS 原生系统的基础上增加了许多文件处理模块，所以是扩展的 HDFS。与其他小文件处理方案一样，EHDFS 的仍然是先通过合并操作降低文件数量，同时为合并后的文件建立有效的索引机制，随后通过预取和缓存模块提高访问速率。该方案不仅确实降低了文件数量，减少了 NameNode 消耗；还降低了文件定位的时间。对大部分文件系统都有长远的借鉴意义。

Hadoop 也被用于批量处理大量的 3D 渲染素材。文献[26]为了解决大量小文件素材存储影响渲染性能的问题，引入了向量机并运用了粒子集算法，根据不同的渲染场景对不同体积的小文件进行分类。这样的合并算法使场景中的文件素材相对统一，在降低系统内存占用的同时提高了渲染精度。在“数字城市”系统的小文件处理中，为了存储大量的不断变化且数据内容丰富的文件信息，文献[27]的合并思路是利用关系数据库选择了两种合并依据分为信息来源和信息时间，以此达到减少系统内文件数量的目的。这些方案针对特殊场景的小文件存储问题进行了研究，虽然这给同类场景的 LOSF 问题的解决指出了方向，但它们的可移植性普遍较差。

1.3 论文组织结构

本文包括六个章节。各章节的主要内容如下：

第一章为绪论。首先，阐述了本文研究课题 Hadoop 平台下基于 HDFS 的小文件存储问题的问题背景，探讨了研究该课题的实际意义；提出了海量小文件 LOSF 的定义；随后说明了解决海量小文件存储问题的重要性。最后还选取了部分具有代表性的国内外相关研究，简要介绍了方案原理并说明优缺点。

第二章对本文课题相关的技术背景进行了研究。具体解释了 HDFS 分布式存储系统的架构及其核心组件的运行机制，系统中对存储操作有重要作用的节点原理和读写流程也进行了深入的分析。最后，对 HAR 文件归档方案、Sequence File 方案、Map File 方案、HDFS Federation 方案等四种小文件解决方案的原理和优缺点进行了详细的

阐述。

第三章根据前述方案中存在的问题，为解决 LOSF 存储问题提出了一种新的优化策略——小文件多级处理模块（MPM）方案。对 MPM 方案的设计思路和体系架构做了清晰的介绍。MPM 方案中包括文件预处理模块、文件合并模块、文件索引模块、文件预取和缓存模块以及碎片整理模块等五个部分。其中，文件预处理模块执行文件大小判定操作并对小文件简单分类后传给下一级处理模块，是方案中的准备阶段。举例说明了文件合并模块凭借合并队列和缓冲队列的设计可以将大量的小文件合并成尽可能少的大文件，释放更多的节点内存。而文件索引模块和预取缓存模块的设置，能够提高文件检索的效率。最后，碎片整理模块能够动态自查，保证整个系统空间的搞利用率。最后重新梳理了引入 MPM 方案后 HDFS 系统的读写流程。

第四章在第三章的基础上，对基于 HDFS 的小文件多级处理模块（MPM）的具体实现过程进行了叙述，并注明了各个步骤的算法流程。首先介绍了文件预处理模块中的小文件大小检测算法，提出了小文件的定义，即检测阈值为 4.35MB，说明了小文件分类的根据。其次介绍了文件合并模块中合并和缓冲队列的初始化以及队列相互转换的过程。之后通过技术实现文件二级索引机制，为文件所在块及具体数据信息之间建立关联关系。接着介绍了文件预取和缓存模块基于文件操作频率对数据进行动态预取并缓存。最后，对如定时器阈值、系统空间利用率、文件剩余体积大小等判断碎片整理模块是否启动的条件进行了说明。

第五章是实验与分析。为了验证本文所提出的方案的可行性，将本文方案与其他三种方案进行了实验对比。首先，对 Hadoop 集群伪分布式搭建的实验环境及其参数配置做了简单说明。然后展示了实验用数据集的构成和大小。对实验对比内容及实验项目的细节也进行了介绍。通过节点内存占用实验、小文件写入效率实验、小文件读取效率实验对原生 HDFS 方案、HAR 文件归档方案、Sequence File 方案以及本文提出的 MPM 方案进行了对比，最终验证了 MPM 方案对 HDFS 小文件存储问题的优化效果最好。能够有效降低节点内存消耗、提升系统的文件读写速度。

第六章是总结与展望。总结了本课题的研究成果并指出了后续改进的方向。

2 小文件问题相关技术分析

本文的研究是基于 Hadoop 平台的分布式文件存储系统 HDFS 中小文件存储问题的优化。因此需要对课题相关的理论知识进行详细的研究和深入的了解。本章首先将对 Hadoop 系统架构及其中的关键模块进行详细的介绍。其次，对四种现有小文件处理方案：HAR 文件归档、Sequence File、Map File、HDFS Federation 等方案的优化原理、实现方式及其优缺点进行阐述和说明。

2.1 Hadoop 系统架构

Apache Software Foundation 公司[28]受到 Google 开发的 Map/Reduce[29]和 Google File System（简称 GFS）的引导，在 2007 年下半年 Hadoop 作为 Lucene[30]的 subproject 正式引入。作为一个分布式文件系统基础框架，Hadoop 的工作原理是将单一的服务器扩充为成百上千台的计算机，形成一个集群。集群中的每一台计算机都作为一个本地服务器为集群系统提供计算和存储服务。除此之外，Hadoop 允许用户在集群中使用编程模型跨计算机处理海量数据集。尽管存在集群中的单个计算机损坏导致系统无法工作的风险，但由于 Hadoop 可以在廉价的硬件上进行部署以完成工作，所以不需要成千上万的硬件支持，可以便捷的达到监测和处理故障的目的，用最短的时间最少的成本为计算机集群提供更好的服务。

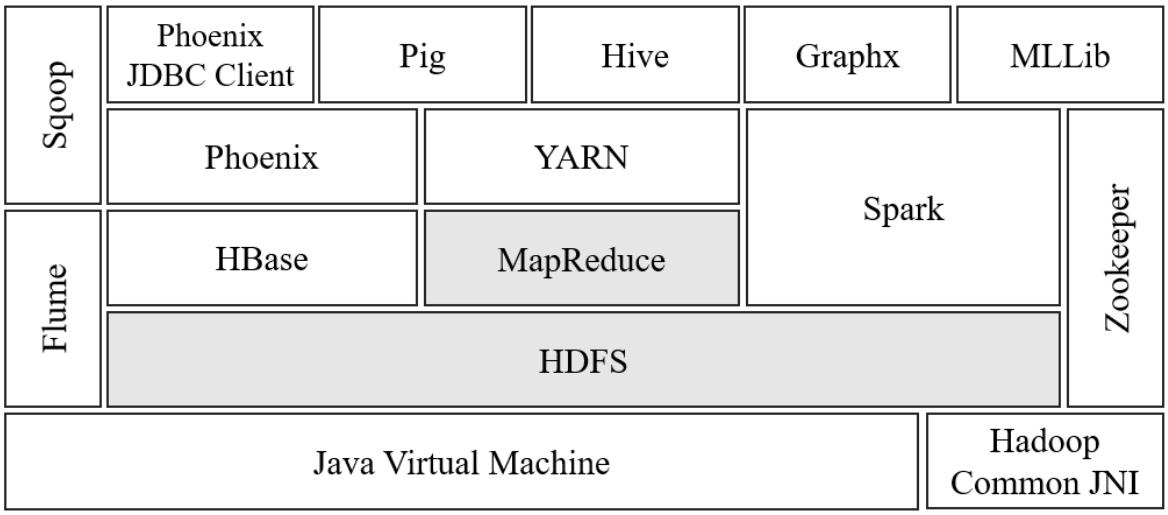

图 2-1 Hadoop 生态系统

Hadoop有许多系统一起工作，但其中最重要的构成是HDFS[31]和MapReduce[32]。HDFS的全称是Hadoop Distributed File System，即Hadoop中的分布式文件系统，为Hadoop提供文件存储和管理功能。可以支持文件的创建、删除、移动以及重命名等操作。MapReduce是Hadoop中的一种底层执行引擎，利用编程模型对海量数据集进行并行高效运算。其具体工作原理将会在后续章节进行详细介绍。

HDFS与MapReduce在Hadoop生态系统[33]中的位置如图2-1所示。目前Hadoop的传统生态系统得到了进一步的扩展，其中一些重要项目有：

Ambari：用于配置和监测Hadoop集群，支持HDFS，MapReduce，Hive， Pig和Sqoop等。以仪表板的形式直观地展示MapReduce，Pig和Hive等程序的完成进度。

Avro：数据序列化系统。

Cassandra：一种不存在SP故障（single point of failure）的可扩展的主数据库。

Chukwa：数据收集系统。

HBase：支持海量数据的结构化表存储。

Hive[34]：支持即席查询和统计数据功能的数据仓库。

Mahout：基于部分人工智能的数据分析算法。

Pig：用于并行运算的高级数据流语言和执行架构。

Spark[35]：一种简单实用的计算引擎，允许包括ETL，人工智能，图像处理等在内的大部分应用运行。具有快速性、通用性等优点。

Tez：一种新型的底部运算模型。通过执行随机的DAG任务来处理大量的交互式用例。完成率高、数据功能强大，有实力逐渐取代Hadoop MapReduce。

ZooKeeper[36]：为其他应用提供支持的协调工具。

综上所述，Hadoop 之所以成为许多公司和组织研究和生产的首选的开源分布式文件存储平台，它具有如下5点优势[37]：

（1）可靠性高。Hadoop具有优秀的按位存储算法和高速处理数据的能力。

（2）扩展性强。Hadoop集群中的每一个计算机被分配数据并完成计算任务，每个计算机内可以扩展为数以千记的计算节点（DataNode），用户可以随时对这些节点进行修改、增删的操作。当存储数据量不断增长时可以通过增加新的 DataNode 来系

统正常运行。

（3）效率高。Hadoop 会根据各个节点当下的实际负载情况进行动态分配，保证系统内各节点负载的均衡性。同时，实现了集群整体的高效运行。

（4）容错性高。Hadoop 可以自动保存数据副本。当某一任务没有成功，可以由副本恢复数据，并且被重新执行。

（5）低成本。与其他的商用大数据处理系统相比，Hadoop 是开源的。且对部署硬件的要求不高，用户可以利用价格便宜的普通计算机搭建一个完整的计算机集群，进而进行数据处理。项目的软、硬件成本大大降低。

2.2 分布式文件系统 HDFS

2.2.1 HDFS 架构

HDFS 是 Hadoop 项目的核心组成，是存储和管理文件数据的基础。采用的是 Master/Slave 的系统结构，NameNode、Secondary NameNode[38]、Client、DataNode 是其重要组成部分。HDFS 的系统结构如图 2-2 所示。

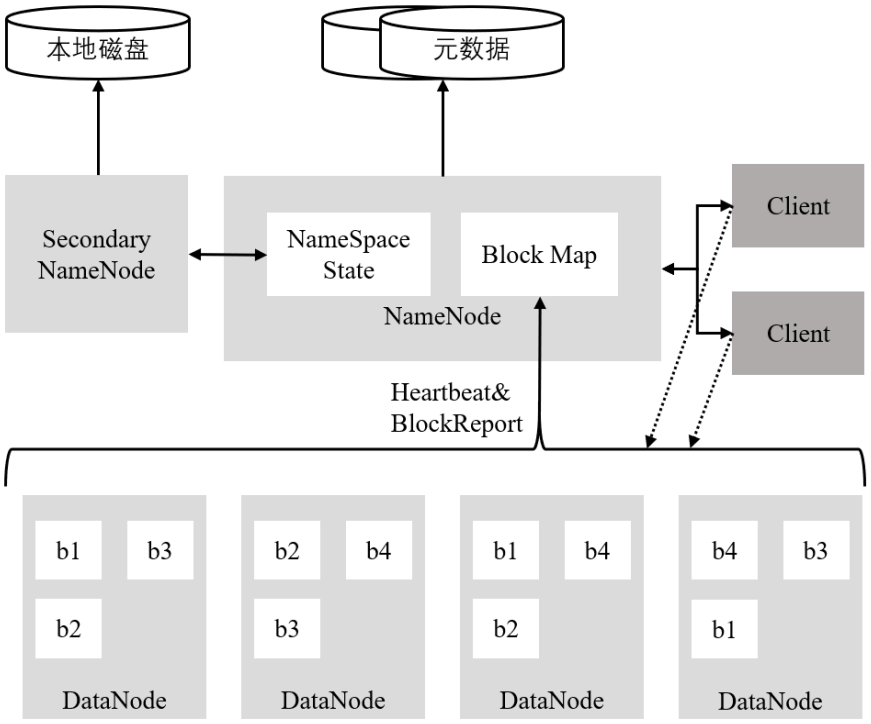

图 2-2 HDFS 系统架构

（1）Client

客户端。主要功能是当文件被传送到 HDFS 时将文件切分成一个一个的 Block 再进行存储。向 NameNode 发出指令可以获取文件的位置信息；向 DataNode 发出指令可以对数据进行读写操作。另外还提供一些指令，可以进行如启动、关闭、访问 HDFS 等操作。

（2）NameNode

名字节点。即 HDFS 存储结构中的 Master 管理者，是系统的主节点。管理着 HDFS 的内存空间以及数据块的映射信息。可以对副本策略进行配置，并协助 Client 处理操作请求。

NameNode 中通过一个 NameSpace 镜像文件来避免文件元数据因为节点损坏而丢失，另外还引入了日志文件来加强系统的可靠性。

（3）Secondary NameNode

二级名字节点。这是一个 NameNode 的备份守护程序，分担 NameNode 上的任务；当 NameNode 出现状况时，可以帮助其恢复，但不能立马替换 NameNode。极大程度上降低了 NameNode 单个节点发生故障时对系统的影响。

（4）DataNode

数据节点。这是 HDFS 存储结构中的 Slave 从节点，执行由主节点 NameNode 发出的指令操作。

实际的数据块就存储在 DataNode 中。单个数据被备份成相同的多份，分别保存到不同的 DataNode 中，每个 DataNode 中不会用重复的备份数据。在图 2-2 中，DataNode1、DataNode3、DataNode4 中都包含 b1 数据，但是每个 DataNode 中都有且只有一个 b1 数据。其他数据的存储同理。备份机制保证了系统在任意一个 DataNode 损坏后，仍可以从其他 DataNode 复制备份文件恢复损坏的 DataNode。系统得以正常运转。

DataNode 中一个最基本的存储单元叫做 Block，一般最多能存储 64MB 的文件。小于一个 Block 默认大小的文件也会占据整个 Block 的内存空间。

2.2.2 HDFS 的优缺点

由于 HDFS 特殊的 Master/Slave 的系统架构布局，使其具有以下优点[39]：

（1） 适合批量处理海量数据

由于 HDFS 不需要移动数据，而是通过移动计算的方式来存储文件。且会将固定的数据位置报告给系统，减少系统交互过程，适合批量处理数据。与此同时，HDFS 特别适合处理达到 GB、TB 甚至 PB 级的数据，是大部分企业存储文件的不二选择。

（2） 对硬件故障具有高容错性

由于拥有备份守护进程，存储在系统中的文件数据集一经生成，系统会执行备份操作，生成多个拷贝数据。之后才将其传输到各个节点中，等待后续的指令操作。HDFS 的这种机制允许当某一个数据丢失时，系统自动通过事先备份的数据进行恢复。这使得 HDFS 得以轻松应对硬件故障问题，系统风险大幅降低。

（3） 保证数据的一致性

HDFS 的访问模式为“一次写入，多次读取”，文件一经写入系统后不能够再进行修改，只能追加写入新的文件。保证了写入数据前后的一致性，有着较高的系统稳定性。

（4） 可在廉价设备上扩展

HDFS 部署集群时对硬件设备的要求不高，用户可以便捷的通过增删普通商用机的方法增删集群节点，扩展成本低，可扩展性高。

（5） 使用普适的编程语言

HDFS 的理想运行平台是 Linux，因为它是使用 JAVA 语言开发的。但某些应用也可以使用 C++编辑。掌握这种编程语言的用户可以轻松的使用 HDFS 进行数据存储工作。

当然 HDFS 并不是完美的，它也有一些缺点。在低延时的场景下，比如在毫秒量级（ms）读取数据，这是 HDFS 无法完成的。它只适合高速率的处理数据，保持高吞吐率。“一次写入，多次读取”的访问模式虽然保持了系统数据的一致性，但在特殊场景下，却不支持文件的写入、删除操作，给用户的使用带来不便。最后，跟所有的分布式文件存储系统一样，在面对海量小文件存储问题时，由于小文件的大小通常远远

小于一个基础存储空间的大小（通常为 64MB），存储空间通常不会被文件占满，有限的节点内存无法承载海量的小文件带来的内存浪费。因此在面对海量小文件时，HDFS 的存储性能并不尽如人意。

2.2.3 HDFS 读写过程

在 HDFS 存储文件的机制中，得益于 FSDataInputStream 类提供的两个接口：PositionReadable 接口和 Seekable 接口，系统可以支持的文件操作方式十分丰富。常见的文件操作指令有新建文件、删除文件、修改文件、查询文件等，其中新建文件和查询文件是分布式文件系统中更加常用的请求。分别对应于文件的读取过程[40]和写入过程[40]，接下来将对这两个操作过程进行详细的介绍。

（1） 文件读取过程

用户通过客户端读取数据过程如图 2-3 所示。

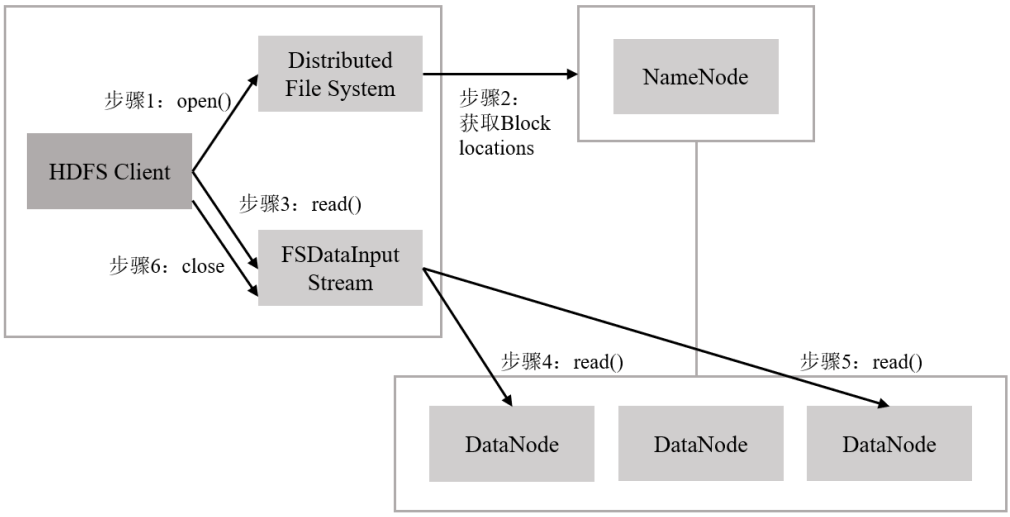

图 2-3 文件读取过程

HDFS 的文件读取过程包含以下步骤：

步骤 1：Client 调用 FileSystem 对象的 open()接口，在这里获取到一个 DistributedFileSystem 的实例，是可以打开指定文件所需的数据。

步骤 2：FileSystem 对象通过远程过程调用协议 RPC（Remote Procedure Call Protocol，简称 RPC）向 NameNode 发送请求，获取文件的 Block 信息以及 Block 的位置信息（Block locations）。同一个 Block 可以返回多个位置信息，这些位置信息根据

Hadoop 的拓扑规则排序，距离 Client 近的排序靠前。

步骤 3：FileSystem 对象通过上一步生成的位置信息序列，返回一个输入流对象 FSDataInputStream，用于读取数据和文件位置的定位。为了方便管理 NameNode 和 DataNode 数据，将 FSDataInputStream 封装成一个 DFSInputStream 对象。随后 Client 调用 read()接口，DFSInputStream 对象就可以找到距离最近的 DataNode 与之连接请求读取数据。

步骤 4：DataNode 收到数据读取请求，将数据传送至 Client。

步骤 5：当最近的 Block 的数据传送完毕后将这个连接通道关闭，与下一个 Block 相连，继续读取数据。需要注意的是，这些操作对于 Client 是隐形的，从 Client 的角度看不存在通道的关闭与重新连接，读取这些数据是一个持续的传送过程。

步骤 6：如果所获取到的 Block 均读取完毕，DFSInputStream 会继续执行步骤 2，获取第二近的 Block 及其位置信息，然后重复后续操作。直到所有的 Block 被读完，此时关闭所有数据流。读取操作结束。

（2） 文件写入过程

用户通过客户端写入数据的过程如图 2-4 所示。HDFS 的文件写入过程包含以下步骤：

步骤 1：Client 调用 DistributedFileSystem 中的 create()，在 HDFS 中新建一个文件。

步骤 2：通过 RPC 调用 NameNode，在其中新建一个不需要 Block 关联的文件，存储在 NameSpace 中。需要注意的是，NameNode 会对当前新建文件进行判断，判断内容主要有：该文件是否与已有文件重复，系统是否有权限进行新建操作。任意一项验证不通过，系统会向用户提示异常。

步骤 3：FileSystem 对象完成前两步之后，与读取文件的步骤类似。系统会返回一个输出流对象 FSDataOutputStream，为了方便管理 NameNode 和 DataNode 数据流，将 FSDataOutputStream 封装成一个 DFSOutputStream。封装后的数据流对写入文件进行处理，将他们切分成一个一个的 Packet，然后按顺序排列成 data queue。

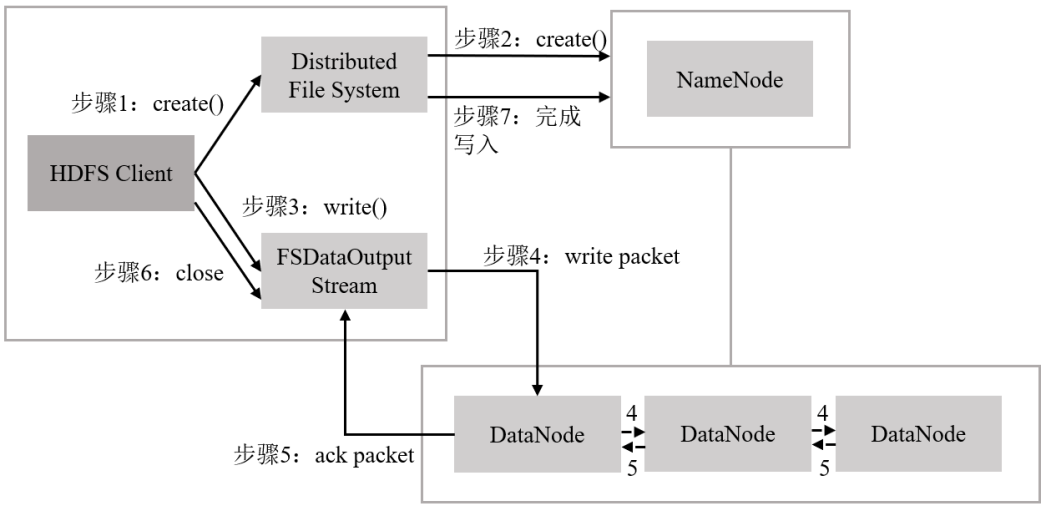

图 2-4　文件写入过程

步骤 4：DataStreamer 根据步骤 3 的顺序为队列中的 Packet 寻找合适的存储位置。NameNode 收到存储请求，返回当前 Packet 需要存储的 DataNode 数，将其排列成一个 pipeline。之后 DataStreamer 依次向 pipeline 中输入数据，直到当前 Packet 完成写入。

步骤 5：DFSOutputStream 有一个由切分的 Packet 组成的队列 ack queue，当 pipeline 中所有被分配的 DataNode 都成功写入数据后，ack queue 自动清除自身的 Packet。

步骤 6：直到所有的 Block 被写完，此时关闭所有数据流。

步骤 7：DataStreamer 收到 ack queue 自动清除信号，表示当前文件已完成写入，通知 DataNode 更改标识号。

2.3 编程模型 MapReduce

MapReduce 是 Hadoop 中的一个底层执行引擎。使用 MapReduce 中的编程模型可以运行在 Hadoop 集群中的所有计算机上，形成一个高速并行处理数据的软件框架。

一个正常的 MapReduce 任务[42]分为 Map 任务和 Reduce 任务，当系统接收到处理数据的请求，首先由 Map 任务将已经被分割成许多独立的 Packet 的数据做并行处理，随后输出并按照一定标准排序，再将排序结果传送给 Reduce 任务。一般来说，MapReduce 框架主要执行任务的分配和检测，将失败的任务进行重新传输；而输入和输出的结果都会被存储在 HDFS 中。这说明 Hadoop 的计算系统和存储系统是使用同

一组节点运行的，这样的机制使得整个集群的灵活度得到提升，可以完整的发挥网络带宽的作用。

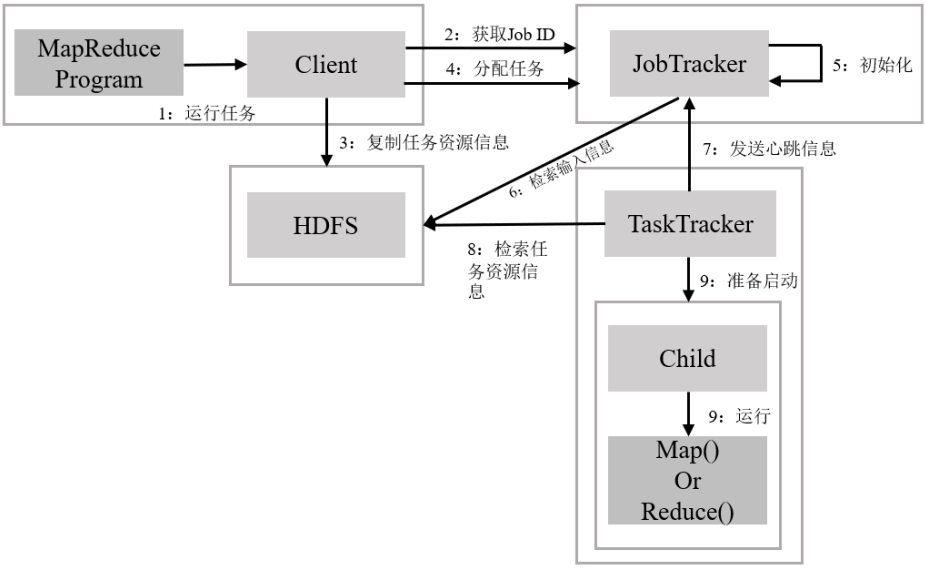

图 2-5　MapReduce 处理数据的流程图

MapReduce 运行机制的过程图 2-5 所示，包含以下步骤：

步骤 1：Client 启动一个 MapReduce 任务。

步骤 2：JobTracker 给 Client 分配一个 Job ID。

步骤 3：运行 MapReduce 任务所需的 source 文件包括 MapReduce 任务形成的 JAR 文件、配置文件和 Client 的分割输入数据。拷贝一份传送给 HDFS 存放在固定的文件夹 Job ID 中。

步骤 4：JobTracker 接受到任务请求将其按序排列成队。随后，根据预先设置好的算法作业调度器对队列中的任务进行调度，按照步骤 3 的分割数据信息为任务新建一个 map()，并交给 JobTracker 运行这个 map()。运行 map()和 reduce()需要使用 TaskTracker 中的 map 槽和 reduce 槽。一般来说，这些槽的数量是固定的。

步骤 5：TaskTracker 定时向 JobTracker 发送一个包含当前任务执行进度等数据的心跳信息。最后一个任务执行完毕，相应的心跳信息就会被发送给 JobTracker，此时该任务被标记为“完成”，Client 会向用户发出任务完成的提示。

综上所述，可以看出 MapReduce 对数据的处理就是不断的切分再合并的过程。这种将数据切分成小的任务进行处理的方式使得集群中的每个计算节点都能同时并行

运算一块不大的部分。保证了系统对数据的高性能处理。

2.4 Hadoop 现有技术方案分析

针对 Hadoop 中的海量小文件问题（LOSF），Apache Software Foundation 公司对 Hadoop 进行了升级，在 Hadoop 系统中引入了专门的技术方案应对小文件存储。下面将介绍四种方案的存储结构及各自的优缺点。

2.4.1 现有技术方案介绍

（1） HAR 归档

Hadoop Archives（简称 HAR），顾名思义是一种文件归档技术。简单来说，就是用户通过 archive()命令操纵一个 MapReduce 任务，将一定范围内的小文件合并成为一个大文件，这个大文件中包括原始文件数据以及相应的索引。合并后的大文件被命名为 HAR 文件，再传送给 HDFS 进行存储。

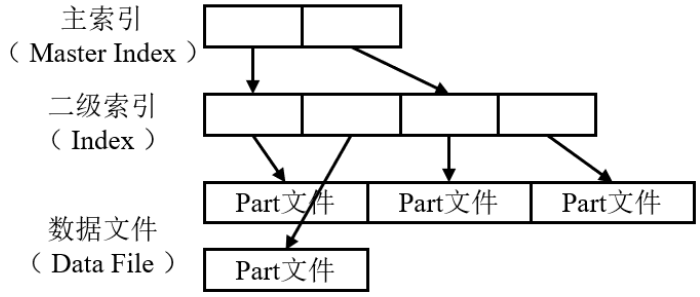

图 2-6 HAR 文件存储结构示意图

如图 2-6 所示是 HAR 归档方案的存储结构示意图。一个 HAR 文件中原始的小文件数据，这里每个小文件被标记为一个 Part 文件；以及一层主索引和一层二级索引信息。索引文件的作用是在读取文件时，方便进行查找和定位。

HAR 归档方案直接将小文件合并成大文件的工作原理虽然比较简单，但是由于文件数量的减少，使得 NameNode 的内存占用直接降低了。另一方面，由于使用了 MapReduce 对文件进行操作，即 MapReduce 对文件的访问是透明的，对 HAR 文件中小文件的输入也不会产生影响。用户可以便捷的通过一个“HAR://URL”的命令格式对 HAR 文件中的 Part 文件进行操作。

当然 HAR 归档方案并不是完美的。首先，虽然 Hadoop 对“HAR://URL”命令格

式的支持较为全面，但是 Hadoop 并没有提供关于 Part 文件相关元数据的管理接口。这给 HDFS 处理经过 HAR 归档后的小文件带来了难度。其次，HAR 归档方案中为了方便小文件读取时的查找和定位操作，使用了两层索引。读取文件时需要遍历两层索引文件才能够完成文件定位，读取速度略有降低。最后，HAR 归档方案在创建 HAR 文件的同时也会为其创建一个副本文件，这部分副本文件占用了系统中额外的存储空间。一旦需要对 HAR 文件中的任意一个文件进行修改，则需要重新创建整个 HAR 文件及其副本文件，十分耗时。

（2） Sequence File

Sequence File 方案也是一种常见的小文件存储解决方案。核心原理依旧是将小文件合并成大文件，但存储结构不同。Sequence File 方案将其内部的小文件名及小文件内容一一对应转化为 Key-Value 形式的数据。文件名是 Key，文件内容是 Value。Sequence File 方案中数据的存储结构如图 2-7 所示。

数据文件（Data File）	键（Key）	值（Value）	键（Key）	值（Value）	…	…	键（Key）	值（Value）

图 2-7 Sequence File 存储结构示意图

Sequence File 方案的存储结构在节省系统节点内存的基础上，考虑到多层索引带来的读取时间增加问题，使用了扁平化的 Key-Value 索引。但这也是问题所在，Key-Value 形式的数据并没有建立真实的索引。虽然读取排位靠前的文件内容时间大大降低，但是读取随机文件时，必须遍历整个 Key-Value 序列才能找到目标文件，反而降低了系统的文件读取效率。还有一个与 HAR 归档方案相同的缺点是，Sequence File 方案中，HDFS 也无法如处理正常文件一样管理合并后的小文件。

（3） Map File

Map File 方案[43]在 Sequence File 方案的基础上，加入了真正的索引文件。如图 2-8 所示是改进后的存储结构示意图。一个完整的 Map File 中除了大量 Key-Value 形式的数据，还包括小文件的索引文件。

其中一部分小文件的文件名和文件的偏移值共同组成了 Map File 的索引文件。提前预取这部分文件，可以在读取小文件时，快速的定位文件位置。相比 HAR 归档方案减少了一层索引，降低了文件查询时间；相比 Sequence File 方案，增加了真实索引，

提高了文件查询效率。缺点是本方案中 HDFS 也无法正常管理合并后的小文件。

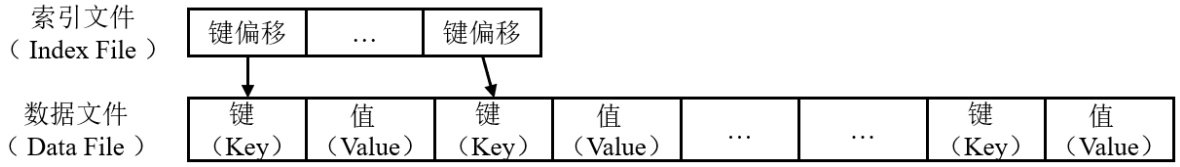

图 2-8　Map File 存储结构示意图

（4） HDFS Federation

为了解决合并文件后，HDFS 对元数据管理的限制。提出了 HDFS Federation 方案[44]。该方案的存储结构示意图如图 2-9 所示。

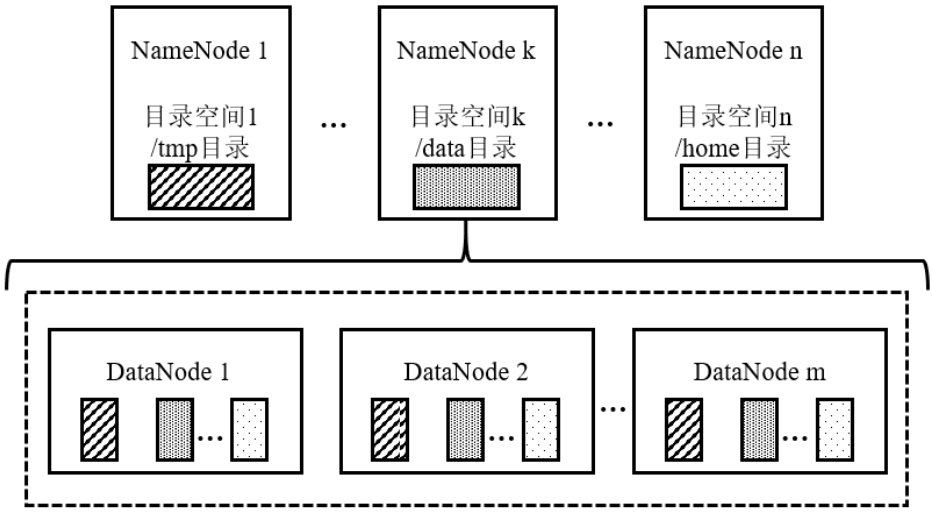

图 2-9　HDFS Federation 存储结构示意图

本方案中，NameNode 被水平扩展为 n 个空间，每个 NameNode n 执行来自总节点 NameNode 的指令，并且向总节点登记和汇报信息。每个扩展节点之间相互独立，管理本空间的元数据信息。

该方案使得将小文件切分成多个块存储在不同的扩展节点中，由于对海量小文件的元数据进行了分散管理，大大减低了节点被元数据信息完全占用的风险。

HDFS Federation 方案不再存在前述三种方案的通病，HDFS 可以对元数据信息进行正常的管理和操作。但是也正因为“分散管理”的技术原理，一旦系统中任意一个 NameNode 出现问题，该扩展节点下管理的所有空间将不可用，其中存储的数据将会丢失，整个系统的数据不再完整。因此，尽管有着相对良好的存储性能，但高风险低可靠性的数据特性还是让 HDFS Federation 未能很好的应用到实际场景中。

2.4.2 存在的问题

根据本节所述，企业和学术界的研究人员为了解决 Hadoop 中海量小文件存储效率低下的问题，进行了大量的实验和研究。这些研究成果针对具体的系统和不同的文件类型达到了不同程度的优化效果，但是仍然存在诸多问题。总结起来可以概括为如下几个方面。

（1）小文件跨块存储

将海量小文件按照一定依据合并成大文件再进行存储是解决现有 LOSF 存储问题的常用的思路。由于 HDFS 自身结构的限制，在合并队伍末端的小文件，很可能被跨块存储，给后续的文件操作如查找、读取、修改、删除等增加了难度。

（2）合并算法不完善

按照一般的按序合并的依据，由于合并队伍末端的小文件其文件大小不固定，有可能合并后，合并文件的体积仍然不能填满默认节点内存。出现的这种合并文件大小差异过大的问题，使得节点内存空间未被充分的使用。

（3）索引机制过于复杂

索引机制是文件合并后文件位置定位的关键环节。缺失索引环节会导致文件合并后数据冗杂读取文件困难；索引机制层级过多会导致文件遍历时间增加，文件读取速率下降。

2.5 本章小结

如何选择适当的合并算法以及索引机制，降低文件的读写过程所消耗的时间，是解决海量小文件存储问题的关键点，也是本文要讨论和研究的主要内容。接下来，本文将针对现有方案中的缺点提出了相对应的解决方案。并对所提方案的各个模块进行详细的分析与介绍。

3 基于 HDFS 的小文件多级处理模块（MPM）的设计

为了要适应大量体积较大的大型数据的存储， HDFS 设计的是 Master/Slave 的节点关系，整个结构中存在着一个 NameNode 和多个 DataNode。随着存储系统中越来越多的小文件的产生，这些大小文件的混合存储使得 HDFS 的内存空间利用率并不高，系统性能大幅下降。更严重的还会导致系统无法承载而发生故障。

如何解决海量小文件在分布式存储系统HDFS中的存储性能低下的问题正是当下亟待解决的热门课题之一。这个问题得到了许多企业的研究者和学者的注意，他们针对自身接触的文件类型和实际情况提出了许多经典的方案并进行了验证。几乎所有方案的核心原理都是先合并大量小文件以填充内存空间，随后为了提高读写效率增加或删除一些文件处理模块。比如引入索引机制，帮助系统快速定位合并后的文件位置；引入预取和缓存机制，减少系统中节点的交互过程，节约文件读取时间，达到提升文件读取效率的目的等。然而通过 2.4.2 节的分析已知，这些方案因为自身研究系统或研究文件类型的局限性，虽然一定程度上的提高了系统的小文件存储性能，但仍然存在诸多的问题。针对目前存在的问题，本文从以下几个方面对小文件存储问题的解决方案进行了改进。

（1）完善文件合并策略，避免小文件被跨块存储或文件合并后体积不均。

（2）增加文件之间的关联性，降低文件读取时间。

（3）选择合适的索引机制。避免索引过于复杂影响文件读取效率。

（4）增加动态预取缓存机制。

（5）增加碎片整理模块。清理异常文件为后续存储提供空间。

总结前述方案的工作原理，合并小文件直接减少了系统中需要存储的文件数量，是处理海量小文件的必经一步，也是关键一步。好的优化合并策略可以避免合并后的大文件体积不均匀或者文件被跨块存储的问题；其次简化索引机制，避免本末倒置的繁杂的索引机制，增加节点交互时间。另外，针对文件的读写操作流程中可以做的优化有：增加预取和缓存模块。对用户经常操作的文件做频率统计，提前将读取频率高的热点文件取出并缓存，这样可以节约大量的访问时间；最后加入碎片整理模块，将

系统中的数据存储空间可利用程度最大化。通过上述多级操作的共同处理，提高小文件合并效率，降低 NameNode 内存浪费，减少节点交互时间，使得整个系统对小文件的处理性能得到可观的提升。

综上所述，本文提出了一种小文件多级处理模块（Multilevel Processing Module for Small Files，以下简称 MPM）。下面将详细介绍小文件多级处理模块（MPM）的架构设计及各个模块的详情。

3.1 小文件多级处理模块（MPM）的架构

小文件多级处理模块（MPM）中通过多模块连续处理达到提升 HDFS 系统小文件存储效率的目的。MPM 中包含以下五个主要的优化模块：

（1）预处理模块。首先在所有文件中筛选出符合条件的小文件，进行简单分类后等待处理。

（2）合并模块。基于空间最优化的原则，合并算法引入了合并队列和缓冲队列，将小文件合并为尽可能接近数据块存储阈值的大文件。一方面直接减少了数据库中小文件的数量，降低了内存消耗；另一方面避免合并后仍有大块的空白空间，使存储空间得到最大程度的利用。

（3）二级索引模块。以小文件的修改日期为依据建立一级索引，小文件的文件名及文件类型作为二级索引。虽然设计了两层索引，但索引的建立依据较为简单，且不再作为 NameNode 的一部分进行存储。在降低 NameNode 内存消耗的同时也保证系统的交互时间的不会大幅的增加。

（4）预取和缓存模块。用户在实际存储文件时，会对一部分文件频繁的进行读取，增加预取和缓存模块，将用户读取次数较多的文件提前缓存下来，减少用户获取这部分文件时产生的重复的交互时间。

（5）碎片整理模块。经过多次读写删除操作后，系统中会存在一些空白空间，将这部分空白空间再利用，可以一定程度的提高系统的利用率。

以上五个模块共同组成了一个完整的 MPM 架构。要实现上述多模块、多级处理的小文件存储优化方案，需要在原生 HDFS 的基础上加入了一个 Client 代理服务器

（Client proxy）。这个代理的主要作用就是搭载着 MPM，在不影响原生 HDFS 存储系统的情况下，对小文件进行处理，并把处理后的数据传输给 HDFS。

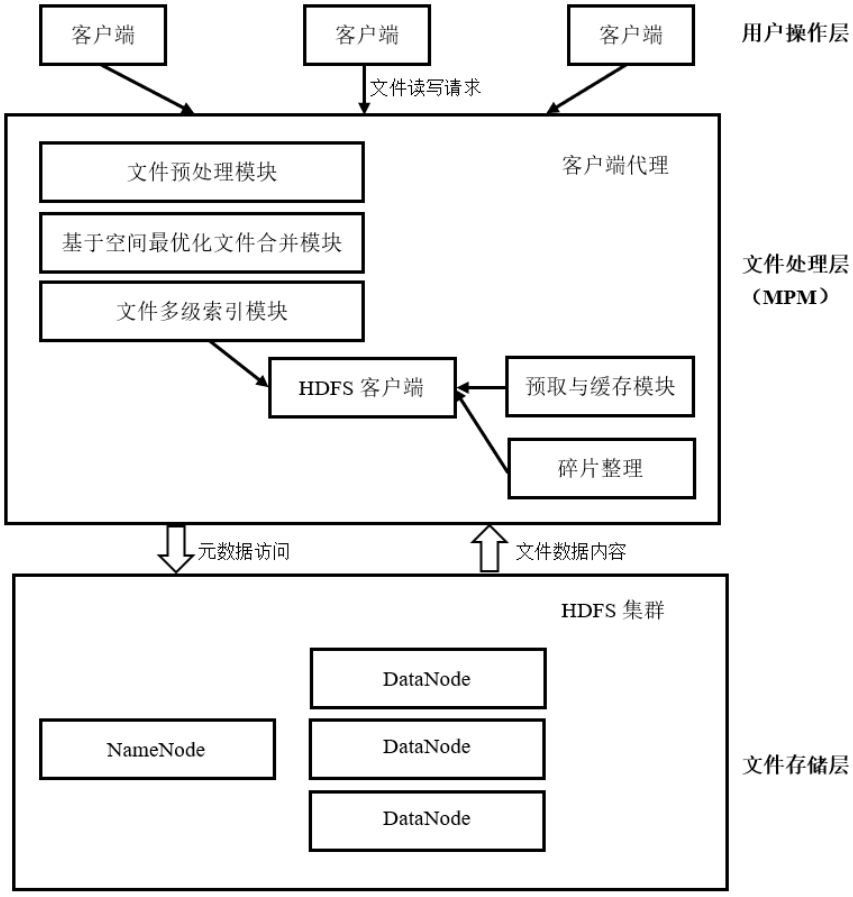

图 3-1　结合 MPM 的新 HDFS 系统架构

结合了 MPM 的 HDFS 分布式文件存储系统的新架构，可以从图 3-1 中看到由三个部分组成，分别是用户操作层，文件处理层和文件存储层。

用户操作层：用户进行文件操作的指令发出层，由这一层的 Client 客户端将指令发送给下一文件处理层。

文件处理层：这是新架构中最关键的模块，用于对小文件进行分类、合并处理等操作。包括小文件多级处理模块（MPM），以及客户端代理。MPM 中又由文件预处理模块、文件合并模块、文件索引模块、文件预取和缓存模块以及文件碎片整理模块 5 个部分组成。

文件存储层：这是新架构中的最后一层，即原生的 HDFS 系统。为新架构提供数据存储功能。

下面将对文件处理层 MPM 中五个主要模块的工作原理进行详细的介绍。

3.1.1 文件预处理模块

文件预处理模块的工作内容包括小文件判定及小文件分类。首先判定接收到的文件是否属于小文件；接着将符合条件的文件根据其后缀扩展名进行简单的分类，相同类型的文件分类到一起；最后送往文件合并模块。不符合条件的文件则直接送到文件存储层，即原生 HDFS 集群进行存储。为后续小文件合并做准备。

关于如何判定一个输入文件是否属于小文件的问题。本文在对 HDFS 的架构进行背景介绍时，提到 HDFS 中一个标准存储块的可用体积是 64MB，存储体积小于这个数字的文件，会造成存储空间的浪费。但是，并不是所有小于 64MB 的文件都应该叫做小文件，都需要进行合并。小文件判定阈值设置的太高会导致文件没有合并的必要，或者无法合并的情况；阈值设置得太低会增加一个合并文件内的小文件数量，增加索引的遍历时间，影响系统的读文件效率。

在文献[24]中，Bo Dong 等人对 HDFS 小文件的判定阈值进行了研究。存储同一份数据，在小文件判定阈值各不相同的情况下，从 0.5MB 到 64MB 不等进行了 23 次合并，最后实验对比多次合并文件所耗费的时间，发现当小文件阈值设置为 4.35MB 时，HDFS 的文件合并效率和系统的存储性能是最优的。因此，在本文之后的研究中，采用的小文件阈值也设定为 4.35MB，即小于或等于 4.35MB 的文件是小文件。小文件通过判定逻辑的筛选，继续被传送给文件合并模块；大于 4.35MB 的文件将被直接传送到 HDFS 存储层。

3.1.2 基于空间最优化的合并模块

当前比较流行的文件合并算法的核心思想都是将系统中的小文件按序合并，一旦超过 HDFS 标准存储空间的大小即暂停合并，把未超过的部分打包成合并文件；再按序合并下一个文件，直到所有文件被合并完成。假设现在有 A-Z 共 26 个体积各异的文件，按照上述合并算法进行合并。合并结果如图 3-2 所示，最终将 26 个小文件合并成了 10 个文件集合。合并文件中小文件个数最多的有 6 个，最少的只有一个。这种

设置固定阈值的合并方法，使得系统中合并文件中的小文件数量不均，更严重的是文件之间体积相差很大，内存空间未被完全利用。对系统存储性能的提升并不明显。

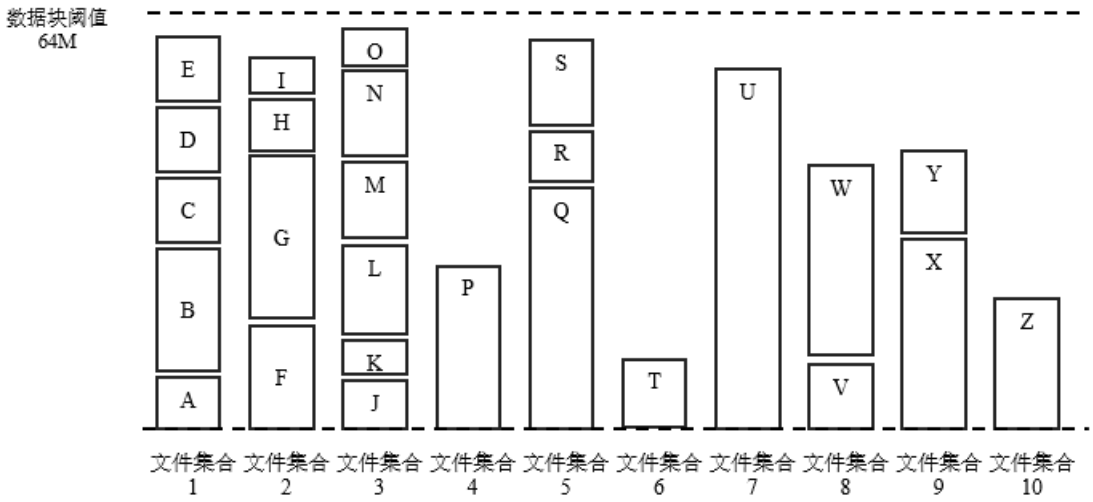

图 3-2　普通算法合并效果

为了改善普通算法的局限性，尽可能的将存储空间使用率最大化。本文于合并模块设计了一种基于空间最优化的小文件合并算法。该算法在合并时不止存在一个单一序列，而是定义了两个队列：合并队列和缓冲队列。目的是除固定文件阈值外增加队列状态为合并依据，进行合并的二次判定。相比上文提到的普通合并算法，增加一个二次判定依据可以使合并后的大文件体积基本均匀分布，降低 NameNode 的存储负荷，取得更优的文件合并效果。另外，该算法能够尽量少的把文件分割成冗杂块，避免跨块存储的情况发生。

合并队列的作用是存放待合并的小文件，当文件长度达到合并阈值后，将文件打包并存储到 HDFS 中。随后清空合并队列，接受下一批待合并的小文件。导致合并队列超出合并阈值的文件会从合并队列中剔除，交给缓冲队列处理。合并队列中的文件累计到一定程度，会优先在缓冲队列中寻找合适的文件填补剩余的空白。这两个队列的角色可以互换，保证队列中始终有等待合并的小文件数据。具体的文件找寻条件以及队列转换条件会在第四章的算法实现小节中做详细的介绍。

经过基于空间最大化的合并算法，图 3-2 中 A-Z 的 26 个体积不同的小文件重新合并后的效果示意图如图 3-3 所示。合并后的文件集合数从 10 个减少到了 8 个，除最后一块存储空间外，其余文件集合的体积大小基本一致，且几乎占满整个数据块。

初步证明本合并算法在降低文件数量的同时，系统内存空间的使用率也得到了显著提升。

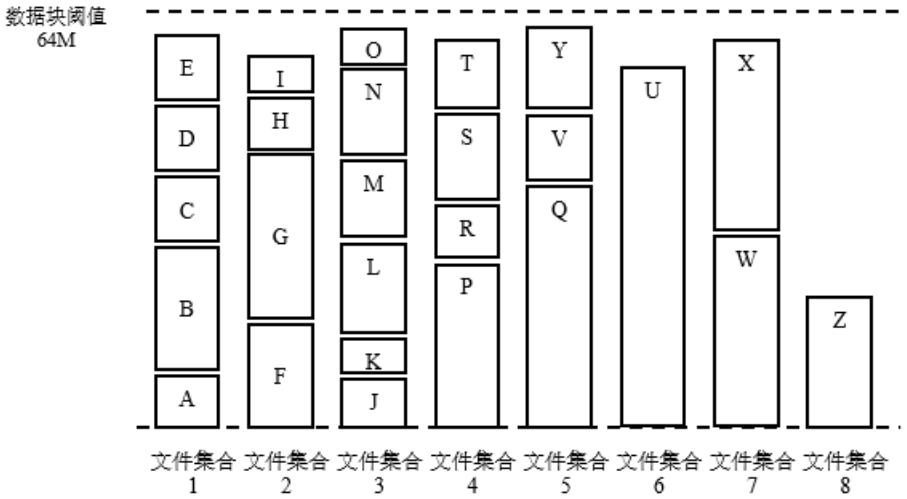

图 3-3　空间最优化合并算法合并效果

3.1.3　二级索引模块

小文件经过前两个模块的处理，已经被分类合并为一些大文件集合。合并文件随后被传送到 Hadoop 上按照正常存储流程，自动进行备份再分别存储于多个 DataNode 上。当用户需要提取某个文件时，为了查找该文件被处理后在系统中的存储位置，这时需要一个索引系统进行协助。

在前文对现有技术方案分析的部分指出了当前小文件的索引机制尚不够完善。例如，HAR 归档方案中的二级索引算法就存在着文件查找效率低的问题。因为每个索引文件会伴随着其目标文件一起被存储。原因除了二层索引交互时间增加，影响系统的读取效率之外；基于 NameNode 建立的索引通常文件大小为 100B 左右，假设每个合并文件中平均有 100 个小文件，那么索引文件将会占用接近 10MB 的内存空间。本来是为了提升文件读取速率的操作，反而牺牲了大量的内存空间。使用了类似原理的解决方案在快速读取文件和系统整体性能的取舍上有些本末倒置。

因此本文提出的新的索引模块的设计思路是基本保留 HAR 归档方案的算法模式，但是更改二级索引的建立标注和索引的存储位置，以改善 NameNode 上索引空间臃肿的问题。其中一级索引以小文件的修改日期为依据，依然存放在 NameNode 中。文件名称及文件类型作为二级索引，存放在 DataNode 中。索引模块工作时，NameNode 首

先收到由用户操作层发来的小文件访问指令；一级索引开始工作，在索引列表中搜索匹配的文件修改时间找到对应的合并文件的位置，得到该文件被切分存储在哪些 DataNode 中以及其他块及块偏移数据等。相应 DataNode 收到访问指令，在自身的二级索引中匹配对应的文件名称以及文件类型，找到目标文件的最终存储位置。将结果反馈给客户端，等待操作执行。索引文件的索引结构如图 3-4 所示。

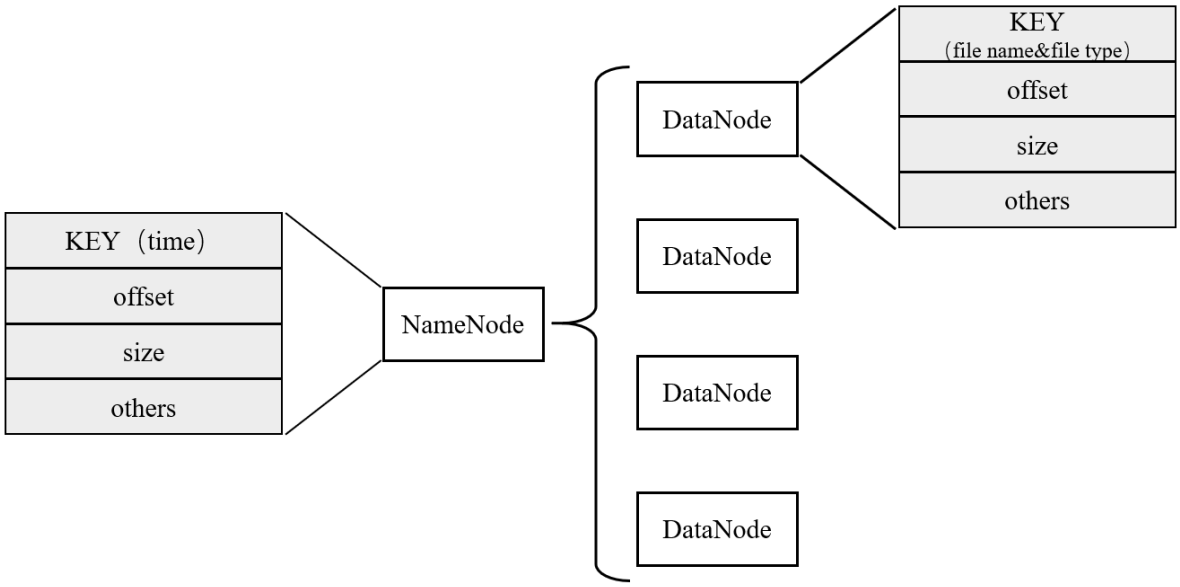

图 3-4　二级索引的结构示意图

3.1.4　预取和缓存模块

已知 NameNode 的内存资源是十分有限且宝贵的，为了降低繁琐的索引机制对内存负荷的占用，上节所述 MPM 方案中的索引模块我们舍弃了一部分读取速率以换取更多的存储空间。为了补偿读取速率的损耗，因此引入一个预取和缓存模块是十分必要的。

预取和缓存模块的处理思路是提前将用户经常操作的文件进行缓存，当用户再次读取该文件时，系统可以直接从缓存模块获得该文件的相关数据，节省与一层一层的节点交互的时间成本。

当一个文件要存储到本文提出的包含 MPM 的 HDFS 新架构中，无论文件体积是否符合小文件判定标准，是否经过 MPM 的多级处理，最终都会传输给 HDFS 原生系统进行数据的存放。所以，本模块是独立于 MPM 中其他模块的，直接与原生的 HDFS

客户端连接处理数据。如图 3-5 所示缓存信息结构，缓存索引信息包括文件读取频率数以及其他相关信息。用户对某一文件每读取一次，就给该文件的读取频次数值加一。数值达到一定程度，该文件就会被判定为用户常用文件，由预取和缓存模块提前将其文件信息缓存到本地。用户下一次读取时，将不再需要与底层 HDFS 进行交互，达到提高系统文件读取效率的目标。

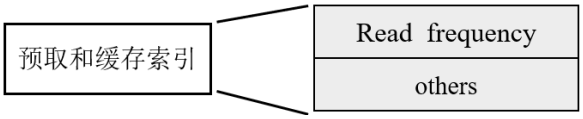

图 3-5　　缓存信息结构

需要注意的是，读取频次达到什么数值才会被预取模块缓存？这与几个系统环境参数有关。比如系统能够分配给预取和缓存模块的空间，可缓存空间越大读取频次门限值越低，可以被缓存的文件就越多。比如用户经常操作文件的个数，用户对文件的读取频次比较平均，读取频次数值相同的文件，缓存模块将优先缓存最近读取的文件。因此读取频次门限的设置需要动态的校准。另外，每个文件的读取频次并不是实时传输给预取和缓存模块的，这中间存在一些时间延迟，这会造成一定的文件缓存误差。

3.1.5　碎片整理

加入了多级处理模块 MPM 的分布式文件系统中，用户操作层发出删除指令后，系统会执行两步操作。首先，从 NameSpace 中删除与该文件相关的数据，并查询被删文件的残留 Blocks。其次，将上一步找到的 Blocks 放入专门的无效 Blocks 中，等到下一次 heartbeat 信息发送成功时，一并将删除命令告知 DataNode；随后清理被删文件的剩余数据块。由于系统调用了 FSDirDeleteOp 类的静态接口来执行命令，但这个接口仅仅是把目标文件从节点空间中的可用标志位做了修改，并没有写入编辑日志中，并没有完成真正意义上的删除。这些文件可以被称作系统中的碎片。

碎片整理模块工作时，需要遍历系统中的文件标识号，记录标识号失效的文件位置，这些文件就是本模块需要清理的对象。然而要进行碎片整理，首先要解决两个重要问题，即何时开始整理以及如何整理。最简单的方式是在碎片整理模块中植入一个定时器。每一个定时周期开始时，碎片整理模块即开始运行。显而易见的是，当系统

空间利用率较高且文件操作不频繁的情况下，模块仍然要反复遍历所有合并文件的文件标识号以检查碎片数量。

因此本文的碎片整理模块不仅设置模块运行时间，还需要设置清理阈值。只有碎片空间超过总存储空间一定比例，系统中碎片数量过多且模块定时器开启的情况下，碎片整理模块才会运行。这样一方面避免了系统被大量失效文件长时间堆积占用内存，也避免了模块运行过于频繁使得系统操作变得繁琐。

具体的参数设置分析如下：

（1）定时器一个循环周期时设置为 10 分钟，即每 10 分钟检查一次合并文件的元数据，判断它们是否需要进行整理。

（2）由于每一次清理任务运行花销较大，为了不影响系统的节点负载。设定一个清理阈值，当整体空间利用率小于 20%时才进行整理。

（3）具体到每一个合并文件，整理的依据是该合并文件中无效小文件的数目占该合并文件中所有小文件的数目的比值大于 50%。

综上所述，碎片整理模块的运行思路如下：当模块定时器打开，模块开始遍历合并文件中的小文件。如果该小文件的索引标识号为真，说明这个小文件仍然被正常存储。否则，说明这个小文件被删除或者损坏。判断系统中的小文件碎片占比，如果超过 20%，则对碎片占比大于 50%的单个文件夹进行碎片整理。删除其中标识号为假的文件索引及内容，并将结果反馈给编辑日志。重复上述步骤直到小文件被遍历完成，最后删除为空的合并文件夹，将标识号分配给新的合并文件，即完成整理。碎片整理模块进入休眠状态，等待下一次定时器周期启动。

3.2 小文件多级处理模块（MPM）的读写流程

本文提出的小文件多级处理模块（MPM）的优化步骤主要是面向系统中最常见也最能体现系统性能优劣的读取、写入、删除等操作的优化。引入了 MPM 的处理后，HDFS 的读写、删除流程发生了变化，具体的介绍见下文。

3.2.1 读取文件

图 3-6 展示了用户通过含 MPM 的 HDFS 读取文件的流程图。具体操作步骤如下：

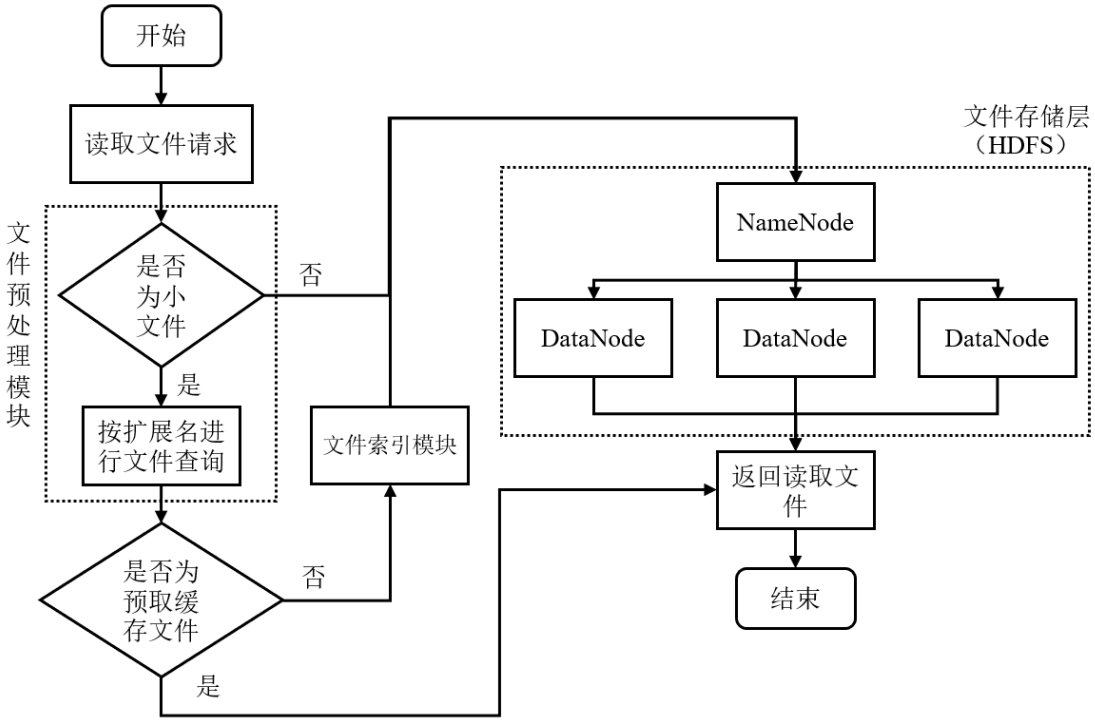

图 3-6　读取文件操作流程图

步骤 1：用户执行读取文件操作，发出文件读取请求传给数据处理层的 MPM。

步骤 2：根据待读取文件信息获得其存储数据。首先文件预处理模块根据文件标识号判断文件是否为小文件。如果是，进入步骤 3。否则进入步骤 4。

步骤 3：小文件被送到文件预取和缓存模块。根据文件名称查找该文件的读取频次，判断其是否为用户的常用文件。如果是，则在预取和缓存模块中以其文件名称为关键信息获取提前缓存的文件内容，将结果返回给用户。否则进入步骤 4。

步骤 4：将读取请求、文件信息传送给文件存储层直接查询。根据文件修改日期，与 NameNode 上的一级索引进行匹配；根据文件名称和文件类型，与 DataNode 上的二级索引进行匹配。最终定位到文件信息。

步骤 5：DataNode 运行 read()操作，将文件所在位置的 Blocks 中的数据接连传送给 Client。

步骤 6：文件读取结果返回给用户。读取文件操作完成，相应数据流关闭。

3.2.2 写入文件

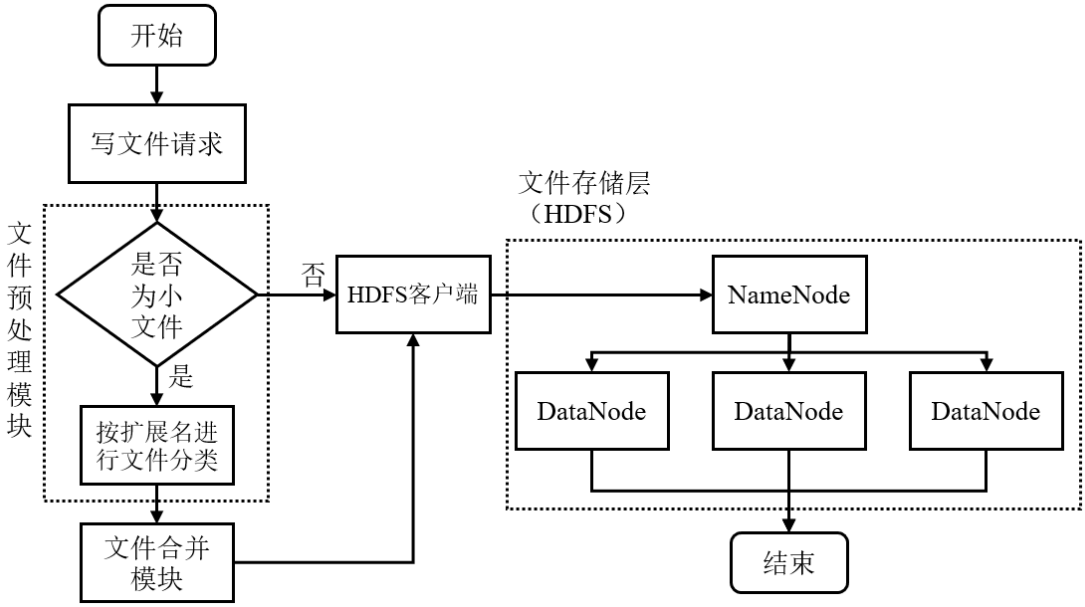

图 3-7 写入文件操作流程图

用户通过含 MPM 的 HDFS 读取文件的流程图如图 3-7 所示。具体操作步骤如下：

步骤 1：用户执行写入文件操作，发出文件写入请求传给数据处理层的 MPM。

步骤 2：文件预处理模块的文件判定逻辑判断传入的文件是否小于或等于小文件判定阈值。结果为真，该文件是小文件，将其标记准备进入文件合并模块。否则，该文件是大文件，直接将其传给 HDFS 进行存储进入步骤 6。

步骤 3：文件预处理模块的分类逻辑将小文件根据文件扩展名进行简单分类，然后按文件后缀传给文件合并模块。

步骤 4：文件合并模块根据空间最优化的合并算法将大量的小文件合并，形成合并文件和二级索引文件。其中，合并文件包含要写入文件的完整文件信息；索引文件则是两层索引构成的序列文件，随合并文件一同传送给 HDFS 客户端，等待系统有读取请求时被使用。

步骤 5：合并文件被传到数据存储层的 HDFS 进行存储。

步骤 6：HDFS 中 NameNode 接收到文件存储信号，对文件信息备份、切分存储到 DataNode 上。特别的是：经过 MPM 的处理合并文件的体积十分接近存储块阈值，可以直接分配给一个 DataNode 进行存储。而步骤 2 中被直接传给 HDFS 的文件，文

件体积可能较大，需要先进行分割才能存储。

步骤 7：DataNode 确认写入文件的数据信息，将写入结果报告给 NameNode 以形成与文件信息一一对应的元数据。

步骤 8：文件写入结果返回给用户。写入文件操作完成，相应数据流关闭。

3.2.3 删除文件

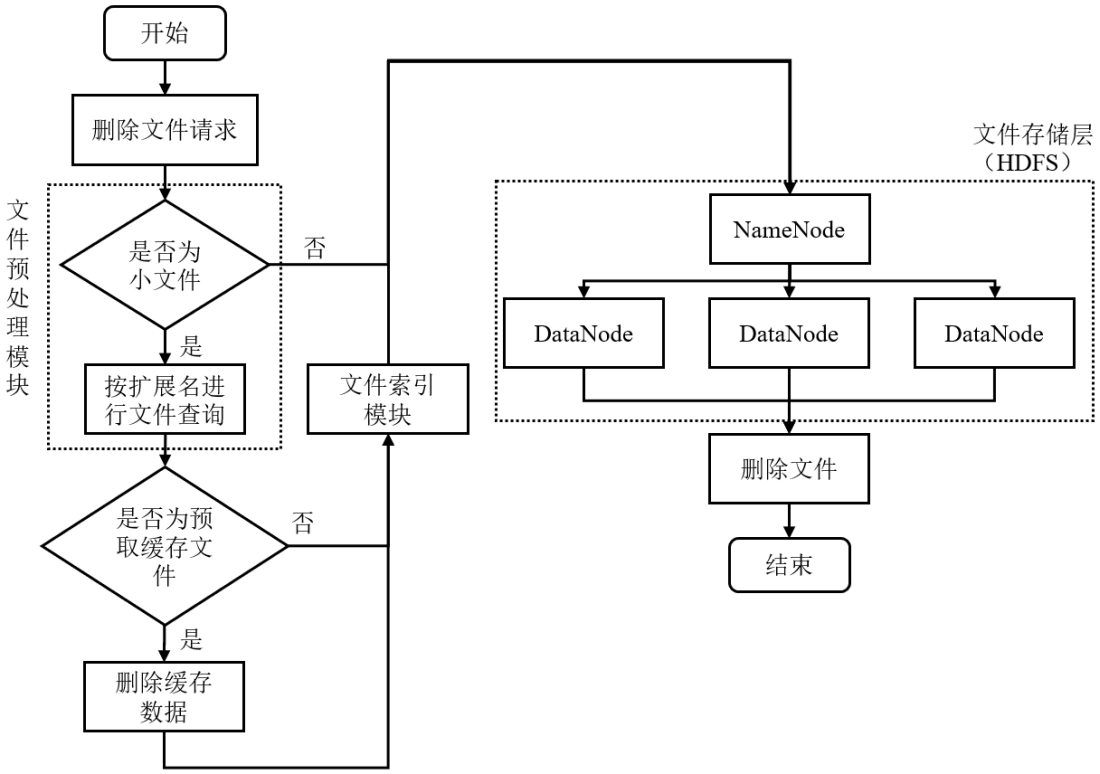

图 3-8 删除文件操作流程图

用户通过含 MPM 的 HDFS 删除文件的流程图如图 3-8 所示。具体操作步骤如下：

步骤 1：用户执行删除文件操作，发出文件删除请求传到数据处理层的 MPM 模块。

步骤 2：MPM 的文件预处理模块判断待处理文件是小文件还是大文件。如果是小文件，则将小文件名加上小文件类型标记，进入步骤 3。否则直接将指令传送到 HDFS 客户端，进入步骤 4。

步骤 3：判断待处理小文件是否是预取和缓存模块里的文件。获取小文件的被读取的次数，如果满足用户常用文件的标准，说明预取和缓存模块中已提前缓存了该文

件的信息。删除模块内与该文件相关的信息，继续下一步。

步骤 4：将删除请求、文件信息传送给文件存储层的 HDFS 客户端进行信息匹配。遍历一级索引找到对应文件修改日期的 NameNode，根据查询结果与二级索引上的 DataNode 交互，查找拥有相同文件名称和文件类型的文件。得到待删除文件的文件位置。

步骤 5：更改文件标识号为无效。删除其中文件索引数据及相关内容。

步骤 6：将删除结果汇报给编辑日志。

步骤 7：文件删除结果返回给用户。删除文件操作完成，相应数据流关闭。

3.3 本章小结

本章通过分析已有的小文件解决方案的不足，对存储过程中每一个有空间提升或改善的步骤进行了一系列的优化。最终提出了多模块共同作用的、独立于 HDFS 之外运行的小文件多级处理模块（MPM）。紧接着本章对小文件多级处理模块的五个主要构成部分：文件预处理模块、文件合并模块、文件索引模块、文件预取和缓存模块及碎片整理模块的设计原理及思路进行了详细的阐述。最后，对引入 MPM 的新 Hadoop 分布式文件系统的读取、写入及删除操作的流程进行了更新说明。

4 基于 HDFS 的小文件多级处理模块（MPM）的实现

在上一章中，详细的说明了本文基于 HDFS 提出的小文件多级处理模块（MPM）的设计思路。MPM 包括五个模块，分别是：预处理模块、基于空间最优化的合并模块、二级索引模块、常用文件预取和缓存模块、碎片整理模块。预处理模块根据小文件判定标准判断输入文件是否是小文件，从而为是否进行合并做出选择；合并模块将经过预处理的小文件通过两个特殊队列之间的相互转换和排列，合并为一批文件数量平均，文件空白空间少的大文件集合，提高系统存储空间的使用率。索引模块依旧使用了二级缓存，但两级索引的位置不全存放在 NameNode 中，这种设计除了能提升查询速度，还能使索引在内存空间中占用更少的体积；碎片整理模块在满足系统设定条件的情况下对无效文件和空白空间进行整理，将部分文件进行重新合并。本章将对 MPM 各个模块的算法实现进行阐述。

4.1 文件预处理模块

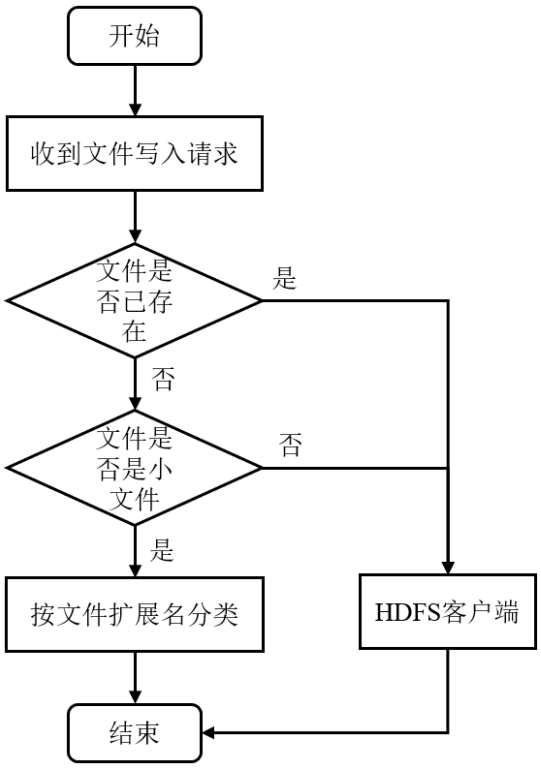

图 4-1　文件预处理模块算法流程图

文件预处理模块是 MPM 的第一步，主要功能是判定收到的文件是否是小文件，并按照文件扩展名对其进行简单分类。这个模块是对文件的一个过滤，为下一步合并操作提供数据。将文件简单的按照扩展名进行分类的方式，避免了精细分类所产生的空间占用和时间损耗。

预处理模块的算法流程图如图 4-1 所示。当客户端虚拟代理收到用户操作层发送来的文件操作请求时，预处理模块会最先收到这个信息。预处理模块同时获取代理的 IP 地址以确认相应数据库存储位置。文件进入 MPM，预处理模块首先在已有文件名称列表中对比，当前文件是否已存在，对比关键词是文件的对象标识号。没有找到重复值，说明该文件是首次写入。否则，说明文件已存在，不能重复存储。

进入小文件判定逻辑部分，本文已经在 3.2.1 节中说明：文件体积不大于 4.35MB 的文件属于小文件。若通过此步骤判定文件是小文件的话，将文件按照文件扩展名分类，等待分配给对应的合并文件。大文件则直接传送到文件存储层的 HDFS 客户端进行存储。

4.2 基于空间最优化的合并模块

表 4-1　合并模块相关变量及参数定义

变量与参数定义	含义
$\boldsymbol{Q_i}$	合并队列。i为纯数字
$\boldsymbol{m}$	合并队列的数量，$0 \leqslant i \leqslant m$
$\boldsymbol{P_i}$	缓冲队列。i为纯数字
$\boldsymbol{n}$	缓冲队列的数量。$0 \leqslant i \leqslant n < m$
$\boldsymbol{f}$	当前处理文件
$\boldsymbol{f_{max}}$	合并队列中体积最大的文件
$\boldsymbol{f_{match}}$	与合并队列最适合的配对文件
$\boldsymbol{T_P}$	缓冲队列容量门限值
$\boldsymbol{T_g}$	标准存储块体积大小，通常为 64MB

基于空间最优化的合并算法的关键在于合并队列与缓冲队列的设置。对小文件进

行合并操作，保证合并后的文件长度尽可能接近 64MB，避免普通合并算法出现的合并文件体积相差过大，导致内存浪费或溢出的情况。合并模块涉及的相关变量及参数定义如表 4-1 所示。

下面介绍一组队列，即一个合并队列和多个缓冲队列时的合并步骤。实际系统中是多个合并队列与多个缓冲队列并联工作。算法流程图如图 4-2 所示。

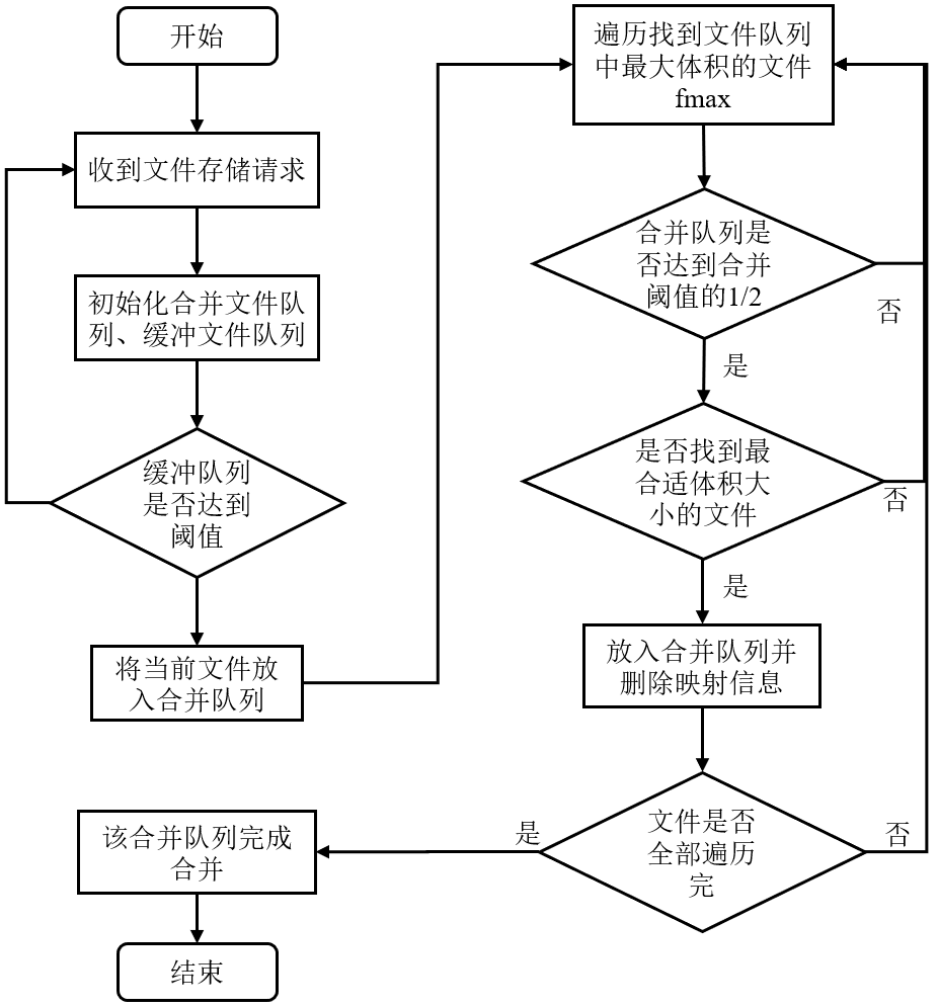

图 4-2　文件合并模块算法流程图

每个步骤的详细描述如下：

步骤 1：初始化数据结构。设$\boldsymbol{m}$个初始合并队列$\boldsymbol{Q_i}$，$\boldsymbol{n}$个初始缓冲队列$\boldsymbol{P_i}$。

步骤 2：对于当前待处理文件$\boldsymbol{f}$，与缓冲队列$\boldsymbol{P_0}$的容量门限值相比，如果$\boldsymbol{f}$小于$\boldsymbol{T_P}$，将$\boldsymbol{f}$放入$\boldsymbol{P_0}$中；否则，将$\boldsymbol{f}$依次其他缓冲队列比较，将其加入符合条件的$\boldsymbol{P_i}$。如果始终没有满足条件的$\boldsymbol{P_i}$，则将$\boldsymbol{f}$独立作为一个新的缓冲队列$\boldsymbol{P_{n+1}}$。合并队列数也同时加一。

步骤 3：缓冲队列$\boldsymbol{P_i}$中的文件体积大于或等于标准存储块的 90%时，即$\boldsymbol{T_P \geqslant 0.9T_g}$时；该缓冲队列中的文件准备开始合并。

步骤 4：遍历缓冲队列$\boldsymbol{P_0}$中，查询文件体积最大的文件$\boldsymbol{f_{max}}$，如果该文件体积大于或等于$\boldsymbol{T_P}$的 50%，进入步骤 5；如果该文件体积小于$\boldsymbol{T_P}$的 50%，进入步骤 6。

步骤 5：将该文件放入当前合并队列$\boldsymbol{Q_0}$，在剩余缓冲队列$\boldsymbol{P_1}$至$\boldsymbol{P_n}$中寻找合适的配对文件$\boldsymbol{f_{match}}$，该文件体积应该小于但最接近与$\boldsymbol{T_g}$减去当前队列文件体积的值。如果有则将$\boldsymbol{f_{match}}$加入当前合并队列。重复此步骤，直到找不到满足条件的$\boldsymbol{f_{match}}$为止。此时，当前合并队列合并完成。进入步骤 7

步骤 6：该文件所在队列不变，等待满足队列的总体积大于文件合并阈值的 50%的条件时，进入步骤 5。

步骤 7：将上述操作中完成合并的队列形成的大文件传送到文件存储层。

步骤 8：一组队列的合并工作结束。

4.3 二级索引模块

一级索引以及二级索引在系统节点中位置及关系的建立后如图 4-3 所示。

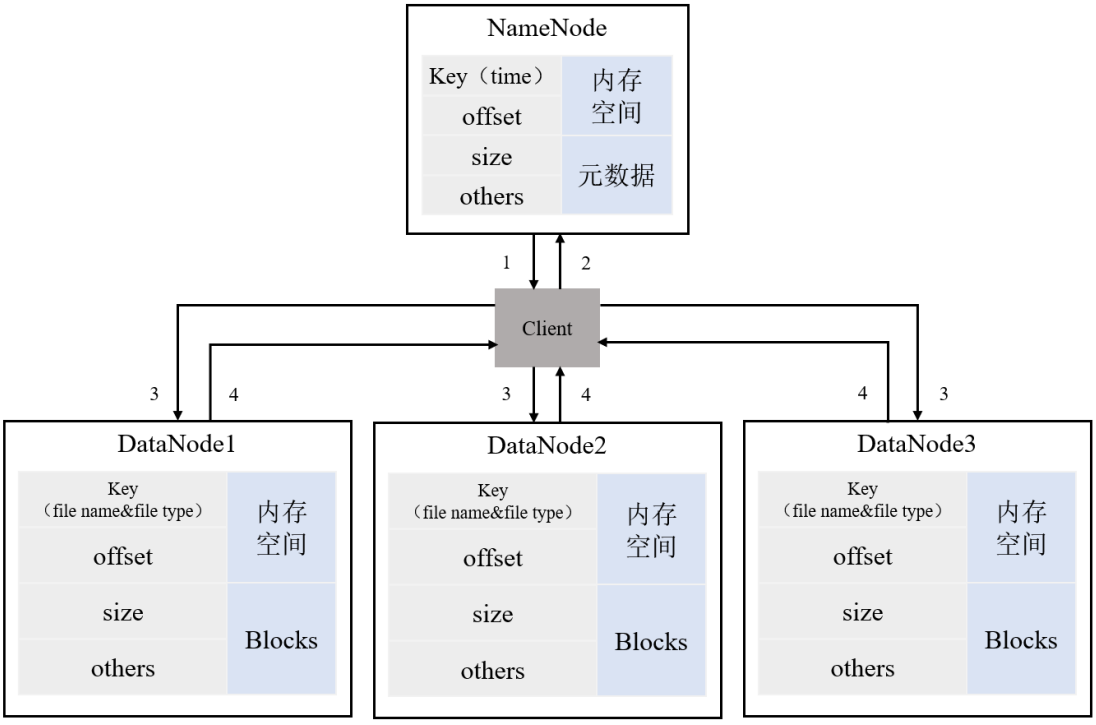

图 4-3　二级索引模块索引结构图

二级索引模块的算法流程的详细步骤如下：

步骤 1：首先用户操作层发起小文件访问请求，请求中包含的信息有文件修改日期、文件名称及文件类型。

步骤 2：MPM 的二级索引模块收到请求。首先根据小文件最后修改日期，得到其所在日期区间；通过一级索引获取该文件所在的合并文件名称和合并文件位置，即实际存储文件的 DataNode（可能有多个）；将获取到的信息反馈给用户操作层。

步骤 3：用户操作层根据就近原则将文件访问请求发送给最近的 DataNode。

步骤 4：DataNode 收到读取请求之后，根据小文件类型（文件名称后缀）在本身内存空间的二级索引中位置匹配对应的一级索引分片以及目标文件详细内容。

步骤 5：将文件信息返回给用户操作层。

步骤 6：至此，通过二级索引模块完成了小文件访问。

4.4 预取和缓存模块

文件预取和缓存模块的核心工作原理是提前缓存用户常用文件的信息，减少用户多次读取文件时系统的交互时间，节约文件访问成本。由于预取和缓存模块的设置，文件的存储信息中增加了一个读取频率标记，主要作用是用户每发起一次文件读取指令，相应文件的读取频率数加一。读取频率最高的前 10%的文件成为用户常用文件。预取和缓存模块每一个月进行一次常用文件更新；将用户这段时间访问次数降低，不再满足常用文件定义，或是出现故障的数据删除，增加缓存用户近期频繁读取的新文件。

表 4-2　预取和缓存模块相关变量及参数定义

变量与参数定义	含义
$\boldsymbol{R}$	文件被读取频率标记，初始值为 0
$\boldsymbol{R_t}$	常用文件门限值，根据文件存储规模设置

本模块涉及的相关变量及参数定义如表 4-2 所示，基于频率统计的算法具体的实现步骤解析如下：

步骤 1：根据文件写入时传给 HDFS 客户端的存储文件信息创建合并文件读取频

率值$\boldsymbol{R}$，初始值为 0。

步骤 2：用户对某一文件执行读取操作时，将文件信息传给 HDFS 客户端的同时，该文件的$\boldsymbol{R}$值计数也加一。

步骤 3：当某一文件R值大于或等于常用文件门限值$\boldsymbol{R_t}$时，即$\boldsymbol{R \geqslant R_t}$，向预取和缓存模块发送该文件的索引信息。

步骤 4：预取和缓存模块根据索引信息向 HDFS 客户端发送访问请求，备份 HDFS 客户端反馈的文件内容并存储在本模块上。

步骤 5：当有新的$\boldsymbol{R}$达到设定的热点阈值$\boldsymbol{R_t}$时，重复上述步骤。

需要注意的是：其中，热点阈值$\boldsymbol{R_t}$是根据文件存储规模预先设定的。当模块的缓存空间已满，有新的文件达到用户常用文件标准时，系统将删除与新文件读取频率数相同的旧文件，保证模块内当前缓存文件是最新的。

上述过程的流程图如图 4-4 所示。

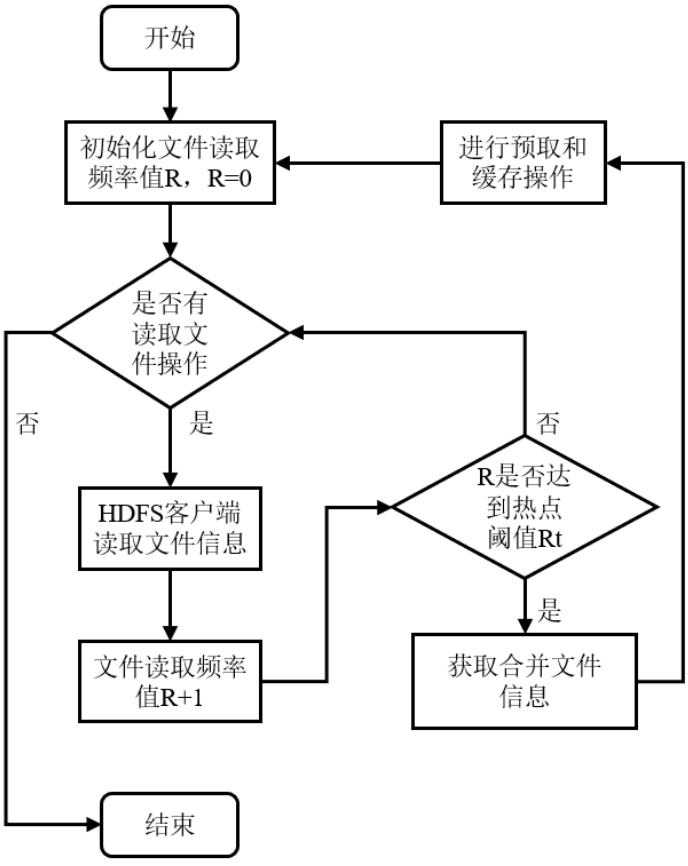

图 4-4　预取和缓存模块流程图

4.5 碎片整理

碎片整理模块的核心在于使用了多重判定算法。该方案保证了碎片整理的及时性和有效性，显著降低系统的无效或空白空间体积，避免碎片整理模块浪费系统开销。表 4-3 是碎片整理模块涉及的相关变量及参数定义。

表 4-3　碎片整理模块相关变量及参数定义

变量与参数定义	含义
f	空白空间超过文件总量 50%的合并文件。
f_{new}	新创建的合并文件。
t	定时器循环周期。单位：min。
G	系统空间利用率。
K	合并队列剩余空间体积比，合并队列剩余空间/合并队列总容量。

碎片整理模块的流程图如图 4-5 所示，其具体流程解析如下：

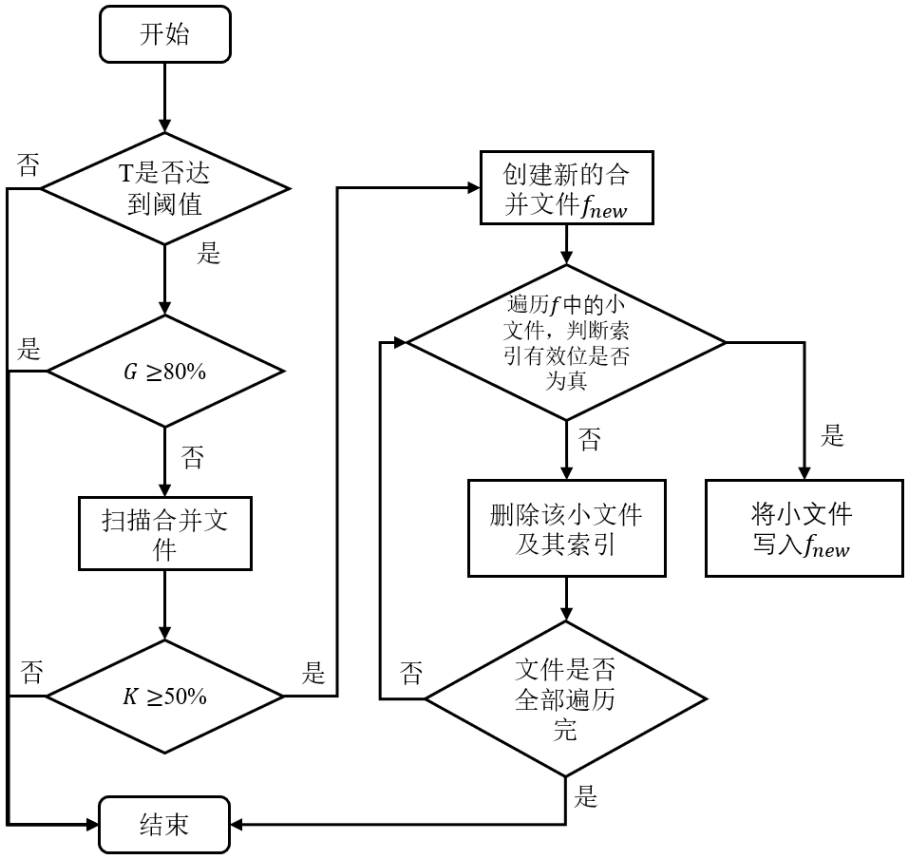

图 4-5　碎片整理模块流程图

步骤 1：定时器时间达到$\boldsymbol{t}$，查看系统空间利用率$\boldsymbol{G}$是否大于或等于 80%。如果$\boldsymbol{G}$**≥80%**，说明系统空间利用率仍比较优异，则进入步骤 6。否则，说明系统中空白空间超过了 20%，需要进行碎片整理。进入步骤 2。

步骤 2：遍历系统中的合并文件$\boldsymbol{f}$。计算$\boldsymbol{f}$（跳过正在进行读取、写入、删除操作的文件）中剩余空间体积比$\boldsymbol{K}$是否大于或等于 50%。如果$\boldsymbol{K}$**<50%**，则跳过；如果$\boldsymbol{K}$**≥50%**，说明，该合并文件空间利用率过低，锁定该文件及其索引文件，进入步骤 3。

步骤 3：创建一个新的合并文件$\boldsymbol{f_{new}}$，遍历待整理的合并文件$\boldsymbol{f}$的索引文件。若某个索引的文件标识位为真，说明该索引对应的小文件有效，进入步骤 4；反之，说明该索引对应的小文件已失效，进入步骤 5。

步骤 4：将索引对应的小文件写入到$\boldsymbol{f_{new}}$中。

步骤 5：将已失效的小文件的索引信息写入到无效索引表。

步骤 6：对索引文件中的索引序列进行重排列，通知索引相关节点更新索引信息。

步骤 7：重复步骤 3，直到文件遍历完成。删除原始的合并文件$\boldsymbol{f}$，将其文件标识号赋予$\boldsymbol{f_{new}}$。

步骤 8：碎片整理结束。模块进入休眠状态，等待下一次整理。

4.6 本章小结

本章详细介绍了基于 HDFS 的小文件多级处理模块（MPM）的实现。对 MPM 中每一个模块的算法原理及其流程逐个讲解。小文件预处理模块依托简单的判定条件及分类标准，实现了对小文件的初步筛选。基于空间最优化的文件合并模块，这个模块的设计初衷使要数据块空间的单个文件填充率尽可能大，并且合并后的文件体积相对平均。为了实现这个目标，算法中提供了合并队列、缓冲队列来执行合并操作，两个队列之间的合并过程及角色交换的条件都在本章做了详细的描述。文件二级索引模块，为了降低索引文件在单个节点中的体积占比，索引模块的算法采用了分节点存储方法，本章对索引之间查询操作的命令流程进行了说明。预取和缓存模块的缓存标准及时间限制是该模块正常运行的前提。碎片整理部分通过对失效文件的清理，为系统开辟新的存储空间。以上的步骤共同构成了小文件多级处理模块 MPM 的算法逻辑基础。

5 实验与分析

为验证本文提出的小文件多级处理模块（MPM）对提升 HDFS 存储小文件性能的可行性，在已搭建的 Hadoop 伪分布式平台上对其进行实验验证。本文的实验设计为：以节点内存元数据体积、方案处理后合并文件的数量和空间利用率，以及文件读写速率、读写耗时作为评价 MPM 存储性能好坏的量化指标，对直接存储的原始 HDFS 方案、HAR 文件归档方案、Sequence File 方案（以下简称 SF 方案）以及本文提出的 MPM 方案等四种 LOSF 解决策略进行对比实验。

5.1 Hadoop 平台搭建

独立（本地）运行模式、完全分布式模式、伪分布式模式是 Hadoop 平台目前支持的三种运行模式[45][46]。独立运行模式使用的是本地文件系统，不需要额外的守护进程和集群配置，只需要一个 Jar 包就可以运行所有的程序；非常适合 MR 程序的调试。完全分布模式[47]，即 Hadoop 集群中每一个节点都是独立的主机。多台主机相互连接构成一个真实的生产环境。伪分布模式[48]利用 VMware 虚拟机在一台计算机上模拟多个节点，构造出一个小规模的集群。使本地机器既是 NameNode，又是 DataNode。它是完全分布模式在单个计算机上的一种特殊再现。由于客观的实验环境和硬件设施的限制，本文选择了常用于 Hadoop 环境测试的伪分布式模式进行 Hadoop 平台搭建。

5.1.1 实验环境

（1）与实验相关的硬件环境参数如下：

电脑 CPU：Intel(R)Core(TM)i5-8250UCPU@1.60GHz-1.80GHz

物理内存：4GB

可用硬盘空间：80GB 以上

（2）与实验相关的软件环境参数如下：

操作系统版本：Ubuntu14.04

数据库：MySQL5.5

Web 服务器：tomcat5.4

开发工具 IDE：Eclipse Mars

JDK 版本：JDK1.7.0_71

Redis 版本：redis3.0

Hadoop：Hadoop0.20.203 版本，包括一个 Client 客户端（负责实现 MPM 功能代理），一个 NameNode 节点和 3 个 DataNode 节点。

5.1.2 Hadoop 集群配置

本次实验的配置默认使用 root 用户登录主节点，安装 JDK 并按步骤配置环境参数。分别修改 hadoop-env.sh、core-site.xml、hdfs-site.xml、mapred-site.xml 这四个文件，完成上述所有文件的修改之后，向 NameNode 发出格式化指令，即可启动 Hadoop 集群。

5.2 实验设计

5.2.1 实验项目

本文的实验项目主要分为三个部分。首先是节点内存占用实验，包括小文件合并之后 NameNode 元数据体积大小、经过合并模块处理后合并文件的数量、系统空间利用率等三个指标。主要验证 MPM 中文件合并模块的效果。

其次是小文件写入性能实验，包括写入耗时和写入速率两个指标。

最后是小文件读取性能试验，利用 SHELL 脚本随机生成的待读取文件列表作为读取数据，自动执行读取操作。然后用读取 1MB 文件所用的时间来计算读取耗时和读取速率两个指标。主要验证使用二级索引策略、并增加预取和缓存模块后，MPM 方案对系统查询文件的效率有否提升。

为避免由于网速或系统原因等不可控因素导致实验结果出现误差，实验项目中有关时间积累的项目都重复实验 10 次，去掉其中的极值及异常值后求均值得到实验结果。

5.2.2 实验数据

本实验中用于测试小文件优化方案存储性能的实验数据选取了包括图片文件、记

事本文件、pdf 文件和各类 office 文件等的海量小文件集合。表 5-1 展示了本次数据集中不同类型的文件数量及文件大小。选取不同类型的文件数据是为了证明本文提出的小文件多级处理模块（MPM）在存储不同类型的海量小文件时的有效性。

表 5-1　小文件集合中不同文件类型数量及文件大小

文件类型	数量（万个）	单个文件大小	总大小
各类 office 文件	15	1KB~5MB	18 GB
.jpg	8.6	100KB~4MB	16 GB
.dat	5.7	10KB-50KB	3.5 GB
.txt	4.6	1KB-50KB	2.1 GB
总体	33.9	/	39.6GB

将上述文件集合随机分为 5 组实验数据，每组分别包含 2 万、4 万、6 万、8 万和 10 万个小文件。实验数据的文件类型分布比例如图 5-1 所示。

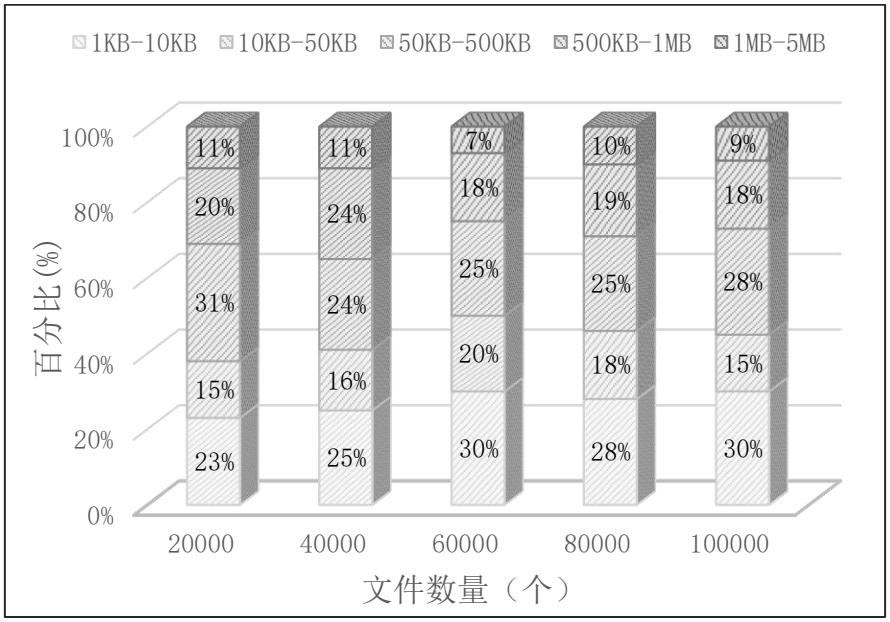

图 5-1　实验数据文件类型分布比例图

5.3 对比实验

5.3.1　节点内存占用分析

本实验对比了原生 HDFS 直接存储方案、HAR 文件归档方案、SF 方案以及本文

的 MPM 方案，在不同文件数量存储时系统中 NameNode 元数据体积占用的情况。

在 Hadoop 集群启动的情况下，依次写入不同数量的文件，随后通过调用 NameNode 的工作端口查看当前 NameNode 内存占用情况。下图 5-3 展示了四种不同存储方案随着小文件数量的增长 NameNode 内存消耗的变化。

从图 5-2 中，我们可以看到 HDFS 直接存储方案，其 NameNode 的元数据内存占用体积远远大于其他三种方案，且随着文件数量的增加呈线性增长趋势。这正是“小文件问题”（LOSF）的主要表现之一。而 HAR 文件归档方案、SF 方案和 MPM 方案都大幅度降低了 NameNode 元数据体积，在减轻节点内存负载的效果上有出色的表现。

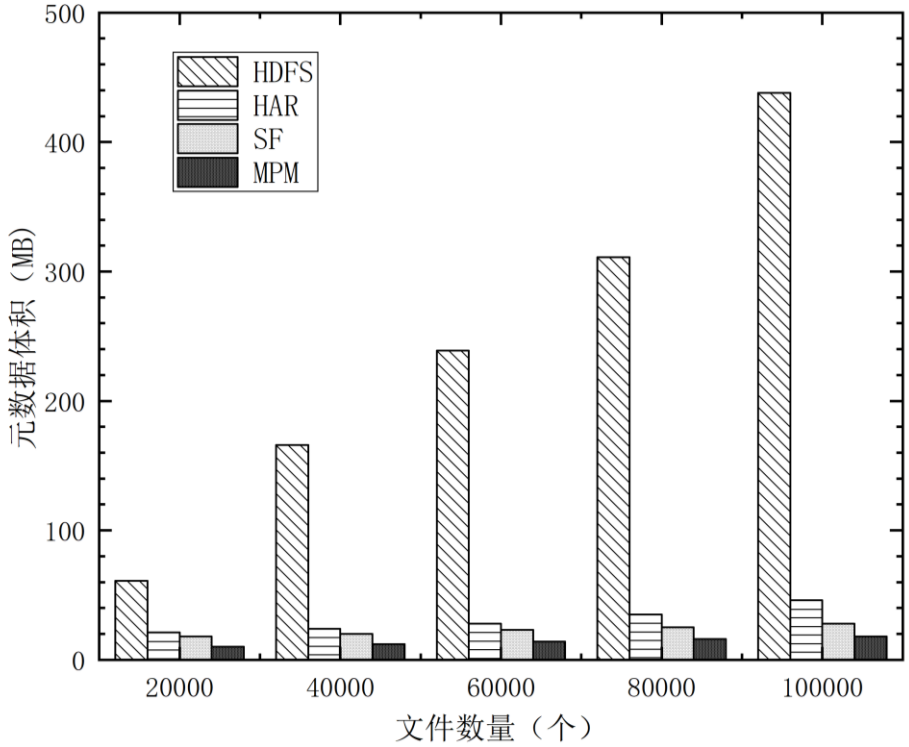

图 5-2　NameNode 中元数据内存占用对比

当文件数量为 100000 个时，HDFS 直接存储方案的产生的元数据体积接近 450MB，HAR 文件归档方案产生了 49MB 元数据，SF 方案产生了 25MB 元数据，MPM 方案产生了 20MB 元数据。可知，后三种方案相比原生 HDFS 方案，在文件数量为 100000 个的节点内存占用实验中，分别为系统减少了 89.11%、93.78%、95.56%的内存占用，避免了大部分节点内存的浪费。

由于 SF 方案和 MPM 方案的节点内存占用优化性能接近，为了对两种方案的性

能有更直观的对比，接下来统计两种方案产生的合并文件个数及数据块利用率等相关数据如下表 5-2 所示。

表 5-2　MPM 方案和 SF 方案的合并文件数量与空间利用率

小文件数量	文件总体积（MB）	合并文件数量（个）		空白空间（MB）		空间利用率	
		MPM	SF	MPM	SF	MPM	SF
20000	3089.45	564	609	4.16	178.56	99.865%	94.220%
40000	6325.13	1031	1165	8.32	353.21	99.868%	94.416%
60000	8023.98	1592	1654	3.99	524.32	99.950%	93.466%
80000	10937.09	2103	2289	5.24	689.03	99.952%	93.700%
100000	12134.22	2729	2930	10.16	824.67	99.916%	93.204%

注：空间利用率=1-空白空间/文件总体积

从上表数据可知，当小文件数量为 20000 个时，MPM 方案将其合并为了 564 个文件集合，且仅有 4.16MB 的空间剩余，整体的空间利用率达到了 99.865%。当小文件数量为 80000 个时，MPM 方案的空间利用率甚至达到了 99.952%，每个合并文件的体积都几乎满足数据块大小阈值，存储空间得到了充分的利用。而 SF 方案虽然在合并文件数量上也产生了与 MPM 方案类似的效果，当小文件数量为 20000 个时，SF 方案将其合并成了 609 个文件，比 MPM 方案多了 45 个文件集。这导致每个文件集中有更多的空间未被利用，其空间利用率仅仅为 94.22%。

综上所述，在节点内存占用对比实验中，MPM 方案不仅显著减少了 NameNode 元数据体积，也有着最好的合并效果，在不同文件数量的实验中均使系统的空间利用率达到 99.8%以上。

5.3.2　写入文件效率分析

本实验对比了原生 HDFS 直接存储方案、HAR 文件归档方案、SF 方案以及本文的 MPM 方案，在不同数量的文件写入系统时的写入速率及写入平均耗时。写入速率越快、平均耗时越低方案存储性能越好。

（1）小文件写入速率对比

如图 5-3 所示，四种方案的文件写入速率都随着小文件数量的增加有所下降。其中原生 HDFS 方案写入小文件的速率最慢，HAR 归档方案和 SF 方案次之，MPM 方案写入速率最快，且随写入文件数量的增加变化不大。当写入文件数量为 100000 个时，原生 HDFS 方案其写入速率仅有 5.5MB/s，而此时的 MPM 方案达到了 10.6MB/s，速率几乎提高了 2 倍。这说明 MPM 方案对提升系统写入文件速率的性能有明显的效果，且写入速率基本稳定，展现了优秀的稳定性和可靠性。

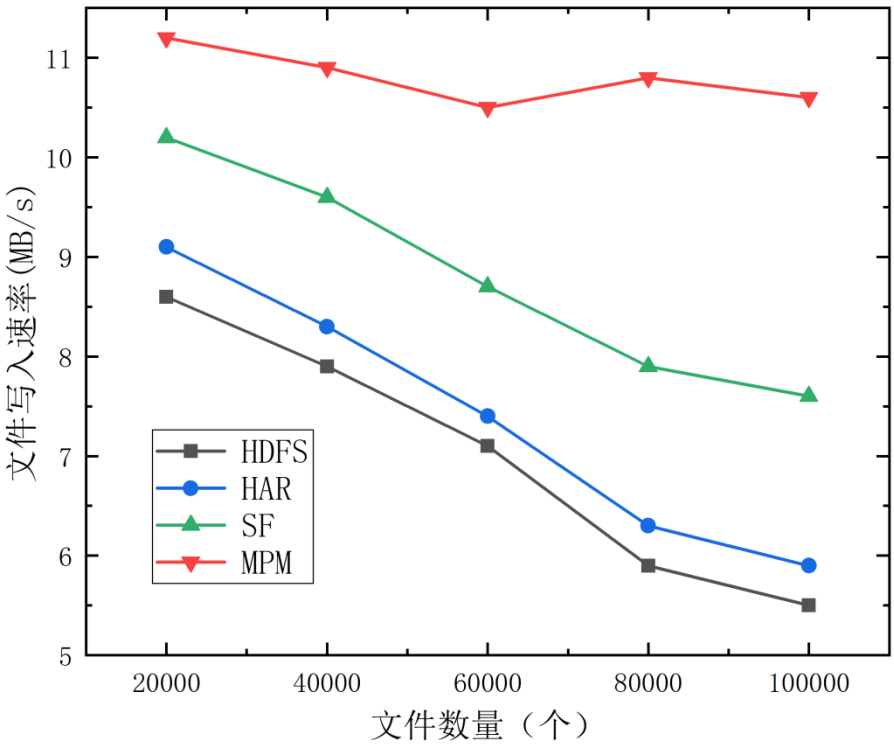

图 5-3　小文件写入速率对比

（2）小文件写入耗时对比

写入单位文件（1MB）所耗费的时间也是衡量方案写入性能的重要指标。下图 5-4 展示了四种方案写入文件耗时随文件数量增长的走势对比。

如图 5-4 所示，四种方案的写入耗时清晰的分为了两组。原生 HDFS 方案的单位文件写入时间随文件数量的变化与 HAR 方案接近，但 HAR 方案的表现略优于 HDFS 方案，这是因为 HAR 方案中将小文件打包形成了分片文件，减少了部分 NameNode 开销，但效果不是最优。另外，图中可以清晰的得到这样的结论：SF 方案与 MPM 方案的写入操作速度明显比原生 HDFS 方案与 HAR 方案更快。因为它们用时更短。当

小文件数量为 10 万个时，原生 HDFS 方案写入单位数量的文件花费了 270ms，此时 MPM 方案耗时近 186ms，减少了 84ms，性能提升了 31.11%；SF 方案耗时 180ms，减少了 90ms，性能提升了 33.33%。值得注意的是，SF 方案在这个实验中表现比 MPM 方案好。这是因为本文的 MPM 方案在合并过程中需要进行如队列转换、文件匹配等更多的步骤，导致文件写入时间略长，但整体耗时依旧稳定的保持在 180ms 左右，与 SF 方案的差距在可以接受的范围内。

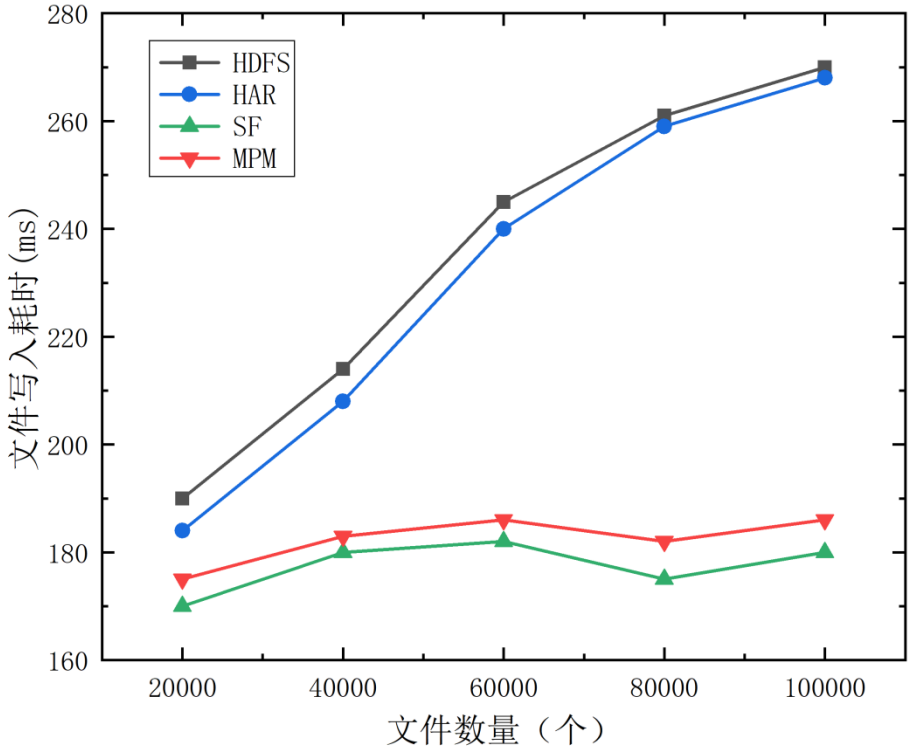

图 5-4　小文件写入耗时对比

5.3.3　读取文件效率分析

本实验对比了原生 HDFS 直接存储方案、HAR 文件归档方案、SF 方案以及本文的 MPM 方案，在读取相同随机文件时的读取速率及平均耗时。平均耗时越低、读取速率越快，则方案的读取性能越好。

（1）小文件读取速率对比

由图 5-5 可知，各方案的读取速率走势起伏不大。HDFS 方案、HAR 方案的读取速率分别在 40MB/s 及 50MB/s 左右上下浮动。SF 方案表现略好一点，基本保持在

65MB/s 且呈上升趋势。这是因为 SF 方案的索引机制先天的缺陷，对小文件的读取性能虽然有所提升但提升幅度有限。而 MPM 方案的最高读取速率可达 91MB/s，相当于同条件下 HDFS 方案的 2.275 倍，展现了优异的文件读取优化效果。原因是 MPM 方案采用了分节点存储的二级索引机制并且增加了预取和缓存模块，文件交互时间大大缩短，其读取速度自然更快。

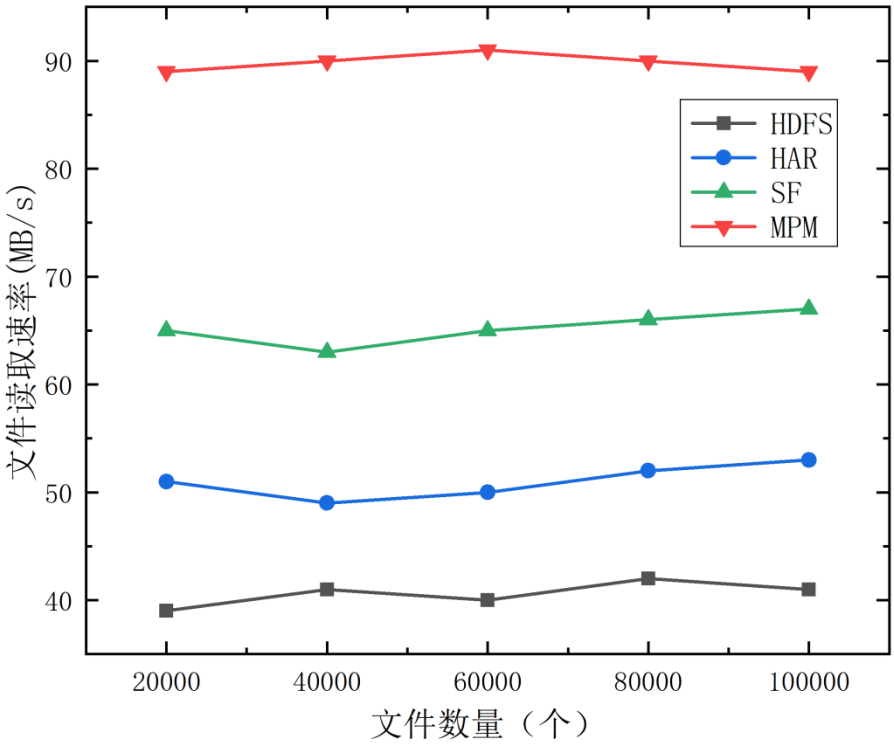

图 5-5　小文件读取速率对比

（2）小文件读取耗时对比

由图 5-6 我们可以看出，每读取 1MB 文件所耗费的时间最多的是 HAR 方案，最高用时达到了 48ms。这是因为 HAR 方案需要访问多余的文件索引信息来获取文件位置，这使得从 HAR 中读取一个小文件比直接从 HDFS 中读取浪费的时间更多。同等条件下，MPM 方案和 SF 方案的读取耗时仅为 22ms、31ms。MPM 方案表现更优的原因是增加了预取和缓存模块，根据用户的操作频率预取和缓存了常用文件，使得读取效率大大提升，节省了读取耗时。

结合两个实验对比数据来看，MPM 方案读取文件时耗时最少，读取速率最快。该方案对小文件的读取性能优化是最优的。

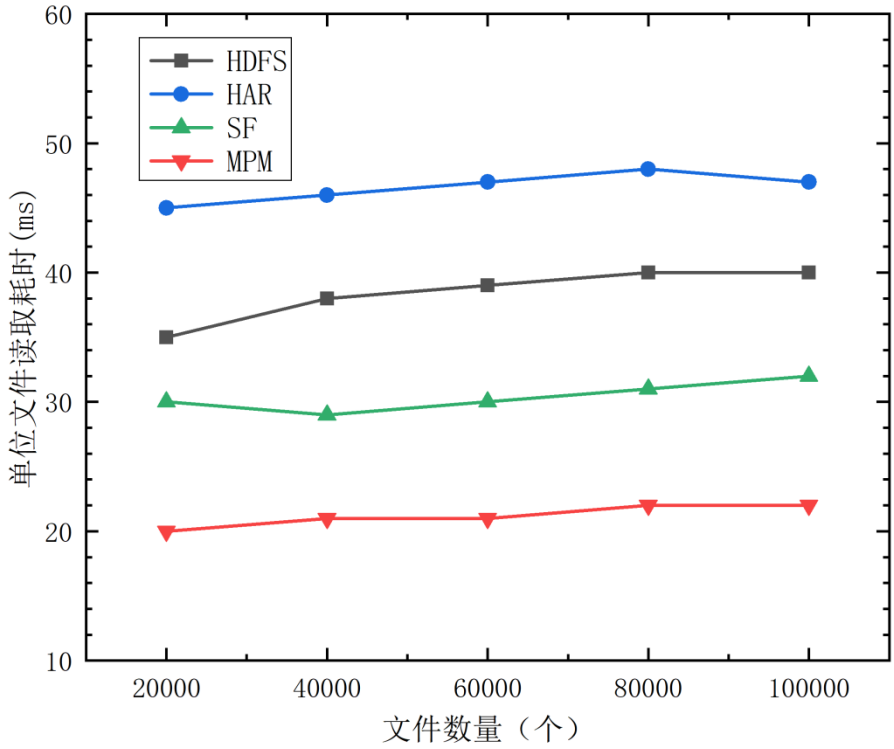

图 5-6　小文件读取单位文件耗时对比

5.4 实验结论

结合三个实验项目的数据对比，MPM 方案在节点内存占用、小文件写入效率和小文件读取效率三个方面都展示出了显著的优化效果。这是因为 MPM 方案通过将文件合并再写入 HDFS 的方式，使节点中单次数据交互中小文件的读取数量增加，形成了一种批量写入的模式。实现了较高的写入速率。但是，也正是由于方案中的文件合并模块的设计，需要在合并过程中创建合并队列和缓冲队列，且允许随时进行转换，使得文件数量较多时，文件写入时间略有增加。但其写入速率和写入耗时指标仍然大幅优于原生 HDFS 方案。另外由于增加了预取和缓存模块，MPM 方案在文件读取性能方面也表现得十分突出且稳定。综合来看，MPM 方案是本次对比实验所有四个方案中最优秀的。

5.5 本章小结

本章首先介绍了实验环境——Hadoop 集群的运行方式，选择了伪分布式模式进

行实验。对环境创建过程中相关文件、参数的设置进行了简要的说明。随后对实验数据的准备：文件集的类型和提及进行了说明。从节点内存占用，文件读、写性能三个方面对无任何优化的原生 HDFS 方案、HAR 文件归档方案、Sequence File 方案以及本文提出的 MPM 方案进行了对比实验。实验结果证明，MPM 方案有效的降低了 NameNode 内存消耗，文件读写性能方面相比 HDFS 原生方案的提升巨大，可以有效解决 Hadoop 中 HDFS 的小文件问题（LOSF）。

6 总结与展望

随着越来越多轻量级的用户端移动应用（如小程序、短视频等）的诞生，大数据系统需要存储的数据不仅体量庞大，而且文件粒度更加精细，生成和更新的速度也日益增长。被业界广泛认可选择的Hadoop分布式文件存储系统在文件处理方面越来越不能满足数据日常的处理需求。如何在处理混杂着海量小文件（LOSF）的大型数据时依旧保持高性能的存储效率成为Hadoop日益重视的问题。这也正是本文讨论的主要课题。

在前文的研究中，本文首先说明了当下Hadoop分布式文件存储系统目前的发展现状和发展意义；介绍了国内外相关从业人员及研究人员对LOSF问题所采用的一系列解决方案，并总结了各种方案的优点和缺点。深层次的分析了Hadoop分布式文件存储系统在大量小文件存储上的架构缺陷。

本文在第二章的理论部分介绍中详细研究了HDFS的框架结构。介绍了HDFS中的重要组成部分，节点间的对应关系。文件的读取和写入过程是HDFS中常见的两种文件操作指令，在这一部分也进行了细致的说明。为介绍新方案中更新后的指令执行步骤做了知识铺垫。接着，简明带过了MapReduce编程模型的工作原理和运行机制。最后对四种现有小文件处理方案：HAR文件归档、Sequence File、Map File、HDFS Federation等的优化原理、实现方式进行了阐述，并分析其优缺点。

经过详细和严密的背景介绍、理论知识普及之后，针对HDFS存储大量小文件时的性能缺陷、现有方案的不足之处，本文创新性的提出了一种小文件多级处理模块（MPM）以解决LOSF问题。具体来说，针对现有合并方案过于简单，无法达到应有的合并效果，使得合并之后仍然有大量的空白空间，对存储性能的提升有限，本文设计了基于空间最优化的合并算法，通过合并队列和缓冲队列的设置和相互转换，保证每个合并文件都被最大限度的使用。文件合并存储之后，文件之间缺乏相关性需要引入索引机制，而现有索引机制过于繁琐反而会影响系统读写性能，本文提出了二级索引机制，通过NameNode一级索引和DataNode二级索引的双重索引机制，提升读写效率的同时，也避免了系统结构的冗杂。最后针对文件空白空间不断积累影响性能的问题又增加了文件预取和缓存模块并定期对系统进行碎片整理。

最后通过搭建的Hadoop伪分布式平台，收集了各种类型的海量小文件数据集合。比较了原生HDFS、HAR文件归档、Sequence File和本文MPM方案在节点内存占用、小文件写入效率、小文件读取效率三个方面的性能表现。实验结果表明，MPM方案在读取文件性能方面相比其他方案始终大幅领先。当写入100000个小文件时，MPM方案经过合并，最终系统只需要对654个文件夹进行存储；系统需要提供的元数据体积也从450MB降低到了20MB，直接减少了95.56%的内存占用。在写入文件效率方面，MPM方案由于采用了两级索引，一定程度上的影响了文件写入速率，但相比原生HDFS方案写入速度仍然有大幅的提高。另外，模块中预置的预取缓存模块也发挥了重要的作用。多个实验证明：MPM方案对系统存储小文件问题性能的提高和优化效果十分显著。

综上所述，本文针对当前Hadoop分布式文件存储系统面对海量小文件时存储效率不佳、索引算法冗杂、文件相关性不足、空白空间利用率低等问题，在吸取其他研究者优秀策略的基础之上提出的这种多级处理优化方案——小文件多级处理模块（MPM）确实对Hadoop存储海量小文件时的系统性能有一定程度的优化。期望这个方案能够给相关工作的研究者和学者以启发，对HDFS系统的实际应用有所贡献，促进Hadoop处理LOSF技术的发展。

由于作者水平、时间、实验硬件等一些因素的限制，小文件多级处理模块（MPM）仍然需要进行改进与完善。例如二级索引模块可以考虑从压缩前缀、索引分层、索引缓存等方向继续优化，进一步减少索引占用内存，增强索引机制。在今后的研究工作中，应该重点关注合并算法的优化并对新的索引机制进行健壮性探索。始终对Hadoop相关技术博客、社区的动态保持关注，也要留意最新的研究成果和产品更新。以便结合Hadoop的最新特性研究LOSF问题的更完善解决方案。

参考文献

[1] 陈明洁. 大数据时代对档案现代化影响和要求[J]. 档案管理, 2013(6):48-49.

[2] 宋瑞琦.浅谈大数据技术与传统行业的结合与发展[J].通讯世界,2018(09):30-31.

[3] 周怡佳.分布式存储技术在大数据时代中的应用[J].电子技术与软件工程,2018(03):182.

[4] Shvachko K, Kuang H, Radia S, et al. The Hadoop Distributed File System[A]. //Symposium on MASS Storage Systems and Technologies[C],Piscataway: IEEE Press,2010:1-10.

[5] Oracle.LustreTM1.8operations manuaL[EB/0L].http://wiki.lustre.org/images/0/09/82-0035_v1.3.pdf,2010.

[6] Liu X, Yu Q, Liao J. FastDFS：A HIGH PERFORMANCE DISTRIBUTED FILE SYSTEM[J]. Icic Express Letters Part B Applications An International Journal of Research & Surveys, 2014,5:1741-1746.

[7] Howard S G, Gobioff H, Leung S. The Google File System[J].Acm Sigops Operating Systems Review, 2003,37(5): 29-43.

[8] Marshall Kirk McKusick,Sean Quinlan. GFS: Evolution on Fast-forward[J].Queu - e,2009,7(7).

[9] Olson M.HADOOP: Scalable,flexible data storage and analysis[J]. IQTQuarterly, 2010,1 (3):14-18.

[10] Waters J K. Yahoo Develops Interface Classification System for Hadoop[J].

[11] Dasgupta S, Natarajan S, Kaipa K, et al. Sentiment analysis of Facebook data using Hadoop based open source technologies[C]// IEEE International Conference on Data Science & Advanced Analytics. 2015.

[12] 李新苗. 中国移动两年内将实现"公共云"服务能力大云计划BC1.0正式推出[J]. 通信世界, 2010(19):34-34.

[13] 三胜产业研究中心.2018年我国移动应用程序（APP）数量增长情况[EB/OL].http://www.china1baogao.com/data/20180606/4601499.html,2018

[14] 李隽.淘宝286亿海量图片存储与处理架构[EB/OL]. http://storage.it168.com/a

2010/0829/1096/000001096373.shtml, 2010.

[15] 头条指数.短视频与城市形象白皮书[EB/OL]. https://index.toutiao.com/pdfjs/view.html?file=//index.toutiao.com/report/download/a18bbb436e0835e755971e7151c12935.pdf. 2018-09.

[16] Mohandas N, Thampi S M. Improving Hadoop Performance in Handling Small Files[C]// International Conference on Advances in Computing & Communications. 2011.

[17] Bo D, Jie Q, Zheng Q, et al. A Novel Approach to Improving the Efficiency of Storing and Accessing Small Files on Hadoop: A Case Study by PowerPoint Files[C]// IEEE International Conference on Services Computing. 2010.

[18] Renner T, Müller J, Thamsen L, et al. Addressing Hadoop's Small File Problem With an Appendable Archive File Format[C]// Computing Frontiers Conference. 2017.

[19] Hadoop Wiki. Sequence File. https://wiki.apache.org/hadoop/SequenceFile[EB/OL]. 2009-09-20.

[20] Apache Software Foundation. HDFS Federation[EB/OL].http://hadoop.apache.org/d-ocs/ current/hadoop-project-dist/hadoop-hdfs/Federation.html.

[21] 袁玉, 崔超远, 乌云,等. 单机下Hadoop小文件处理性能分析[J]. 计算机工程与应用, 2013, 49(3):57-60.

[22] 彭明军, 李宗华, 杨存吉. WebGIS实现技术及发展研究[J]. 测绘地理信息, 2001(1):41-44.

[23] Liu X, Han J, Zhong Y, et al. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS[C]// IEEE International Conference on Cluster Computing & Workshops. 2009.

[24] Dong B, Zheng Q, Tian F, et al. An optimized approach for storing and accessing small files on cloud storage[J]. Journal of Network & Computer Applications, 2012, 35(6):1847---1862.

[25] Chandrasekar S , Dakshinamurthy R , Seshakumar P G , et al. A novel indexing scheme for efficient handling of small files in Hadoop Distributed File System[M]. 2013.

[26] Zhang Y, Zhu Z, Cui H, et al. Small files storing and computing optimization in Hadoop parallel rendering[J]. Concurrency & Computation Practice & Experience,

2017, 29(20):1269-1274.

[27] 刘小俊, 徐正全, 潘少明. 一种结合RDBMS和Hadoop的海量小文件存储方法[J]. 武汉大学学报(信息科学版), 2013, 38(1):113-115.

[28] Apache Hadoop.The Apache Software Foundation [EB/OL]. http://hadoop.apache.org/docs/stable/.

[29] Guo Z, Fox G, Mo Z. Investigation of Data Locality in MapReduce[J]. IEEE/ACM International Symposium on Cluster Cloud & Grid Computing, 2012:419-426.

[30] Lucene.Apache Lucene[EB/OL]. http://lucene.apache.org/.

[31] 翟永东. Hadoop分布式文件系统(HDFS)可靠性的研究与优化[D]. 华中科技大学, 2011.

[32] Dean, Jeffrey, Ghemawat, et al. MapReduce: A Flexible Data Processing Tool[J]. Communications of the Acm, 2010, 53(1):72-77.

[33] Apache Hadoop. Related projects[EB/OL]. http://hadoop.apache.org.

[34] Minar N, Gray M, Roup O, et al. Hive: Distributed Agents for Networking Things[M]// Hive: distributed agents for networking things. 2000.

[35] Meng X, et al. Spark SQL: Relational Data Processing in Spark[C]// Acm Sigmod International Conference on Management of Data. 2015.

[36] Junqueira F, Reed B. ZooKeeper: Distributed Process Coordination[M]. 2013.

[37] 刘长征, 李威兵. Hadoop技术在海量信息处理上的优势[J]. 黑龙江科学, 2013(12):26-27.

[38] Varghese L A, Sreejith V P, Bose S. Enhancing NameNode fault tolerance in Hadoop over cloud environment[C]// Sixth International Conference on Advanced Computing. 2015.

[39] Polato I, Ré R, Goldman A, et al. A comprehensive view of Hadoop research—A systematic literature review[J]. Journal of Network & Computer Applications, 2014, 46:1-25.

[40] 张海, 马建红. 基于HDFS的小文件存储与读取优化策略[J]. 计算机系统应用, 2014(5).

[41] 张春明, 芮建武, 何婷婷. 一种Hadoop小文件存储和读取的方法[J]. 计算机应用与软件, 2012(11):95-100.

[42] 宋杰, 刘雪冰, 朱志良,等. 一种能效优化的MapReduce资源比模型[J]. 计算机学报, 2015, 38(1):59-73.

[43] 洪旭升, 林世平. 基于MapFile的HDFS小文件存储效率问题[J]. 计算机系统应用, 2012, 21(11):179-182.

[44] 尧炜,马又良.浅析Hadoop 1.0与2.0设计原理[J].邮电设计技术,2014(07):37-42.

[45] 崔文斌, 牟少敏, 王云诚,等. Hadoop大数据平台的搭建与测试[J]. 山东农业大学学报(自然科学版), 2013, 44(4):550-555.

[46] 韩震, 孙红. 基于Hadoop的分布式平台实现[J]. 软件导刊, 2017, 16(3):56-58.

[47] 黄素萍, 葛萌. Hadoop平台在大数据处理中的应用研究[J]. 现代计算机(专业版), 2013(29):12-15.

[48] 戴中华, 盛鸿彬, 王丽莉. 基于Hadoop平台的大数据分析与处理[J]. 通讯世界, 2015(6):59-60.