分 类 号 _____ 学号 M201372457

学校代码 10487 密级 _____

华中科技大学

硕士学位论文

一种文件路径与属性信息分离的分布式元数据组织方法

学位申请人：杨　勇

学 科 专 业：计算机科学与技术

指 导 教 师：李春花 副教授

答 辩 日 期：2016.5.26

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering

Distributed Metadata Organization Method Based on File Path and Attribute Information Separation

Candidate : Yang Yong

Major : Computer Science and Technology

Supervisor : Assoc. Prof. Li Chunhua

Huazhong University of Science & Technology

Wuhan 430074, P.R.China

May, 2016

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：

日期：　　年　　月　　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：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

随着大数据时代的到来，面向大数据的存储系统纷纷出现。不断增长的数据量，使得集中式元数据管理系统的负担越来越重，逐渐成为大数据存储的瓶颈。为此，人们提出了多种分布式元数据管理方法，但由于元数据的结构类型复杂多样，目前尚没有一种方法能够同时改善元数据管理的性能和扩展性。

提出了一种文件路径和属性信息分离的分布式元数据组织方法。将元数据组织成目录索引和元数据属性信息两个部分，通过构建目录索引，将元数据以目录或小于目录为单位划分到不同的桶（Bucket）内，再根据元数据服务器集群的负载情况将桶指派到不同的元数据服务器上。方法利用目录索引和桶提高元数据的管理性能；通过构建目录索引时考虑集群负载情况，实现元数据管理的可扩展性。此外，提出基于该方法的元数据位置缓存策略，策略解决了位置缓存信息不一致的问题，缩短了元数据管理的流程。

测试结果表明，提出的方法能获得较高的管理性能，特别适合高并发的情况；具有良好的可扩展性和较好的访问局部性，而且可以不限制目录的大小；避免了重命名元数据造成的不必要的迁移。与集中式元数据管理方法对比，方法采用单一元数据服务器时，元数据的创建、查询等操作性能都有了数倍的提升。

关键词：元数据，元数据组织，分布式，目录索引

Abstract

With the advent of the era of big data, storage systems for big data have increasingly emerged. The centralized metadata management systems are more and more overloaded and will become the bottleneck of the big data storage system with the growing amount of data. Therefore, people put forward a variety of distributed metadata management methods. However, due to the structure of the metadata is more complicated, there is currently no a way to improve the performance and scalability of metadata management at the same time.

This paper proposes a distributed metadata organization method, which separates the file path and attribute information. We organize metadata into two parts, the index of directory and metadata attribute information. Divide the metadata in the unit of directory or less than the directory into the different buckets by building the index of directory, and assign the buckets to different metadata servers according to the load of the metadata cluster. The method can improve the management performance of metadata by using directory index and bucket. At the same time, the scalability of metadata management can be achieved by building the directory index with the consideration of the cluster load. In addition, we propose the position-based metadata caching strategy, which solves the problem of inconsistency location of cached information and simplifies the flow of metadata management.

The test results show that the proposed method has high performance, especially for high concurrency. It has a good scalability and metadata locality, and it don’t limit the size of the directory. It avoids unnecessary migration caused by renaming the metadata. In comparison with the centralized metadata management method, the operation performance of creation, searching for metadata has been improved by several times, while the

proposed method uses a single metadata server.

Key words: metadata, metadata organization, distributed, directory index

目 录

1 绪 论

1.1 研究背景及意义

随着大数据时代的到来，全球数据规模呈现出爆炸式的增长。IDC（Internet Data Center）提供的数据显示，全球数据总量正以每年 40%的速度增长，2014 年全球数据总量约为 6.2ZB（1ZB=1 万亿 GB），预计 2020 年全球数据总量将达到 40ZB[1]。Facebook 公司 2013 年注册用户为 11.5 亿，用户每天发布的评论、状态、图片等各种内容多达 47.5 亿条，其中每天上传的图片达 3.5 亿张。2015 年，我国最大的电子商务平台淘宝网，双 11 当天完成支付 7.1 亿笔，最高峰达 8.59 万笔/秒。

大数据时代数据具有 4V 特征，即数据量大（Volume）、数据类型多（Variety）、数据增长速度快（Velocity）和价值密度低（Value）[2]，传统的文件系统已经不能满足大数据环境下数据存储、管理和理解的需求。因此，各大公司纷纷推出自己的云平台。比如 Amazon 云平台 Amazon Web Services（AWS）[3]，Google 云平台 Google App Engine（GAE）[4]，微软云平台 Windows Azure Platform[5]。云平台给大数据的生成、采集、处理、分析、理解和应用提供了一整套解决方案。

作为云平台的底层支撑，大数据存储需要在存储大规模的异构数据的同时，为云平台提供稳定而高效的服务。针对不同结构类型的数据，各种大数据存储系统纷纷出现。如分布式存储系统，Google 的 GFS（Google File System）[6]和 Apache 的开源项目 HDFS[7]，用于存储海量非结构化的数据；分布式键值系统，Amazon 的 Dynamo[8]和淘宝的 Tair[9]，用于存储关系简单的半结构化数据；分布式数据库系统，Google Bigtable[10]和 HBase（Hadoop database）[11]，用于存储关系复杂的半结构化数据。

研究显示，在分布式文件系统中有 50%-80%的文件操作关于元数据[12]。元数据（Metadata）是描述其它数据的数据，在文件系统中主要描述文件的属性。在大数据时代，元数据不仅数量巨大，而且部分属性（文件大小、目录大小、目录深度等）

符合长尾分布。例如有 90%的目录大小少于 128 个，而少数目录大小超过百万[13-15]。作为访问各种存储系统的入口，元数据管理的性能关系到大数据存储的性能，而提升元数据管理性能的关键在于如何有效的组织大量的元数据。因此，研究大数据环境下存储系统中元数据的组织方法具有十分重要的现实意义。

1.2 国内外研究现状

1.2.1 大数据存储系统发展现状

随着云存储、云计算技术的发展，各大公司纷纷推出自己的云平台以应对大数据带来的挑战。如 Amazon 云平台 Amazon Web Services（AWS），Google 云平台 Google App Engine（GAE），微软云平台 Windows Azure Platform。

Amazon Web Services（AWS）是 Amazon 构建的一个云平台，提供了一系列云服务。弹性云计算 EC2（Elastic Computing）[16]是其核心的产品，能够随着托管的应用程序访问量的变化，自动增加或减少 EC2 实例。简单对象存储 Simple Storage Service（S3）[17]是其提供的一种对象存储服务，用于存储文档、视频、音频等大的对象。

Google App Engine（GAE）是 Google 提供的一种 PaaS[18]（Platform-as-a-Service）服务，使得开发者可以通过 Google 期望的方式使用它的基础设施服务。GAE 主要由应用服务器和存储区组成，应用服务器对不同的开发语言提供运行支持；存储区的核心是 DataStore，它支持自动增加或减少存储节点，提供线性扩展能力。

Windows Azure Platforms 是微软推出的云服务平台，用户可以利用该平台获得微软数据中心的计算资源、存储服务。Windows Azure 平台为每个计算实例提供虚拟机服务，Azure Blob[19]，Table[20]等提供存储服务，Azure Blob 存储二进制数据，如文档、图片、视频等个人文件，Azure Table 存储更加结构化的数据。

大数据环境下的云存储服务，不仅需要存储大量类型不同的数据，还要能为云平台上层的计算提供各种高效稳定的数据服务。传统的存储模式已经无法满足这一

需求，需要研究针对大数据的新型的存储模式。

1.2.2 大数据存储模式研究现状

目前，根据存储数据的类型，大数据的存储模式主要分成分布式文件系统，分布式表格系统，分布式键值系统和分布式数据库四种。

(1) 分布式文件系统

分布式文件系统主要有两个功能：一是存储只能使用二进制的格式保存的文档、图片、视频之类的 BLOB（binary large object，二进制大对象）；另一个是作为分布式表格系统的持久层。

Google 文件系统 Google File System（GFS）[6]是一种典型的分布式文件系统，具有高容错、高可用和低成本的特点。GFS 的系统架构如图 1-1 所示，由 GFS Master、GFS Chunk Server 和 GFS 客户端组成。其中，Master 管理全局命名空间，并通过心跳机制和 CS 交互信息。

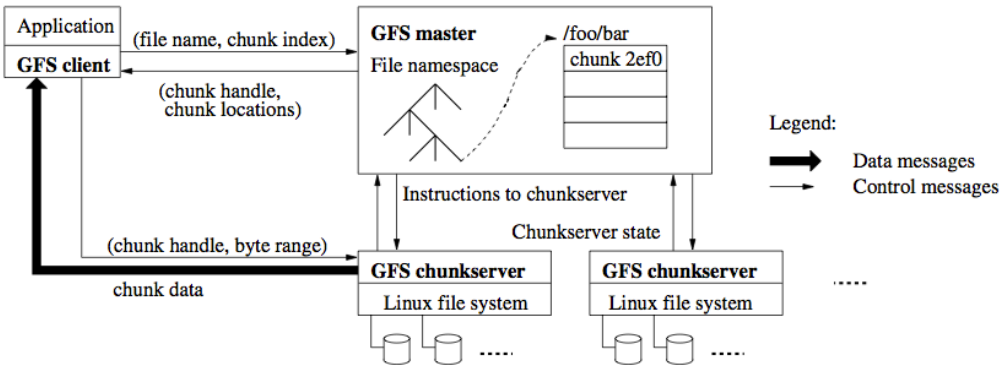

图 1-1　GFS 整体架构

(2) 分布式表格系统

分布式表格系统提供表格模型服务，每个表格通过主键唯一标识，表格由多行组成，行包含多列。

Google BigTable[10]是谷歌开发的分布式表格系统，是一种非关系型数据库。Bigtable 构建于 GFS 之上，在 GFS 上增加了一层分布式索引层。另外，Bigtable 依

赖 Chubby[21]分布式锁服务进行服务器的选举和全局信息维护。Bigtable 的架构如图 1-2 所示，主要由客户端程序库（Client Library）、一个主控服务器（Master Server）和多个分服务器（Tablet Server）三个部分组成。

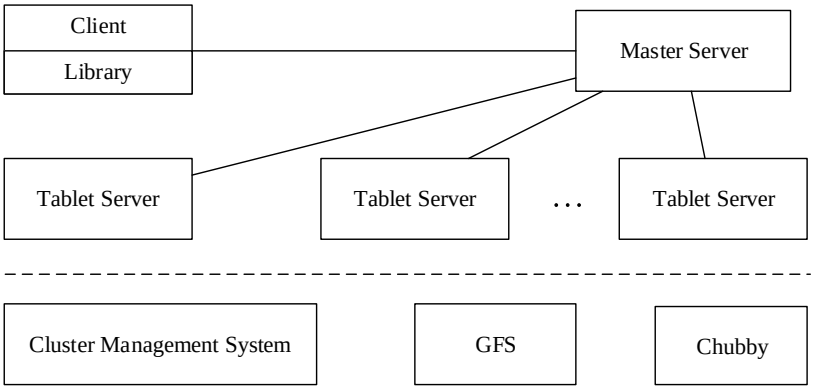

图 1-2　Bigtable 整体架构

客户端程序库（Client Library）提供了应用程序访问 Bigtable 的接口，通过接口应用程序可以对表格进行 CRUD（Create/Read/Update/Delete）等操作。客户端程序通过 Chubby 获取控制信息并与分服务器直接传递数据。

主控服务器（Master Server）负责管理所有的分服务器，包括给分服务器分配表格，监控分服务器并对分服务器实现负责均衡和故障恢复。

分服务器（Tablet Server）位于 GFS 上层，实现表格的装载和卸载、表格内容的读写、子表内容的合并与分裂。

(3) 分布式键值系统

分布式键值系统提供支持单个键值对的增、删、改、查操作，可以看成是分布式表格系统的特例。

Amazon Dynamo[8]是一个分布式键值系统，它提供了简单的键值方式存储数据，不支持复杂的查询，适合采用哈希分布的算法存储数据。

Dynamo 通过改进的一致性哈希算法[22]将简单的键值对数据分布到多个存储节点，它避免了因存储节点变化、失效导致的大规模数据迁移。考虑节点处理能力的不同，Dynamo 为每个节点分配多个 token，每个 token 都在哈希环上对应一个虚拟

节点，虚拟节点和真实的节点提供相同的服务，数据被更加均衡的分布在节点上，提高了平衡性。

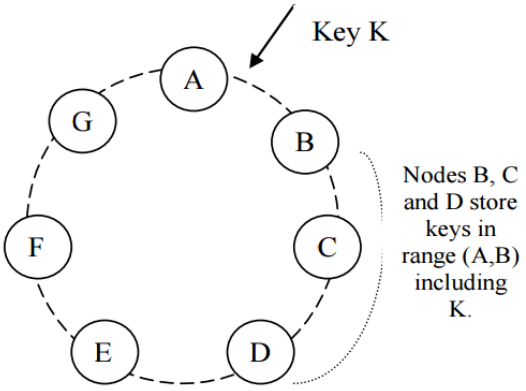

图 1-3　Dynamo 中键的划分与复制

Dynamo 为保证数据的可用性，它将数据副本保存多个节点，如图 1-3 所示，节点 B，C，D 均保存了映射在（A，B）哈希范围内的数据。

(4) 分布式数据库

分布式数据库是对传统关系型数据库进行的技术升级，使其具备可扩展和容错能力。

Google Spanner 在 Colossus 系统之上搭建的分布式数据库，具有极高的扩展性，而且支持跨数据中心的事务，能通过同步复制和多版本控制满足外部访问的一致性。Spanner 的架构如图 1-4 所示。Span server 提供存储服务，每个 Span server 会服务多个子表，每个子表中包含多个目录。客户端通过 Location Proxy 提供的数据位置信息访问数据所在的 Span server。

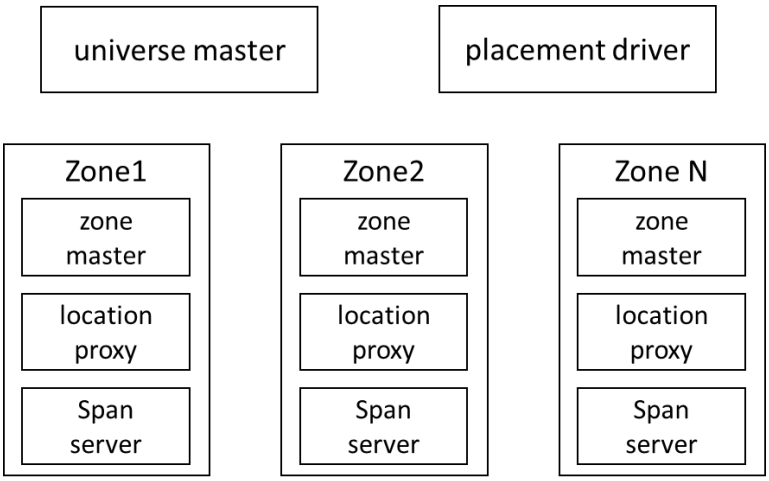

图 1-4　Spanner 整体架构

元数据服务是访问各种存储系统的入口。在面向大数据的存储模式中，元数据不仅规模巨大，而且元数据的操作占整体系统操作的 50%-80%[8]。因此，元数据管理的性能关系到大数据存储系统的整体性能，而提升元数据管理性能的关键，在于如何有效的组织大量的元数据并且提供高效的元数据操作。

1.2.3 元数据组织研究现状

大数据存储的元数据管理方法可以分成集中式[7]、分布式和无元数据[23]管理三种。集中式元数据管理通常使用一台元数据服务器管理元数据；分布式元数据管理是通过一定的规则将元数据分散到多个元数据服务上，由元数据服务器集群统一管理元数据；无元数据管理中没有元数据服务器，元数据由客户端维护，而文件的定位则利用算法来完成。

无论是集中式还是无元数据管理的方法都存在明显的缺点。集中式会出现单机资源瓶颈，无元数据管理的方式容易造成客户端负载过重、元数据访问局部性差等。因此，本文主要研究分布式的元数据管理方法。

针对分布式的元数据管理方法，国内外很多学者提出了不同的元数据组织方法，主要可以概括为以下三种：

(1) 静态子树划分

静态子树划分方法[24-27]是将全局目录树划分成多个子目录树并分配到指定的元数据服务器上。采用此种方法的有 NFS、Coda[26] 、AFS [27]系统。这种划分方法有很多优点，比如实现简单、元数据访问局部性好。但其缺点也很明显，随着系统中文件和目录的增加，不同的目录子树中包含的文件数量会不平衡，导致集群负载不均衡，需要人工介入，重新划分负载较重的服务节点上的目录子树。

(2) 动态子树划分

动态子树划分方法[28]可以解决静态子树划分中容易出现的负载不均衡的问题。方法根据元数据节点的负载情况，动态的将目录划分到多个节点中，避免单个节点负载过重。这种方法也存在一些缺点，如集群扩展代价大、目录结构变化发生数据迁移、延迟时间长等。Ceph[28]作为最典型的使用此种划分方法的分布式文件系统，使用动态子树划分将元数据在元数据服务器集群之间进行分布，如图 1-5 所示，Ceph 将系统目录树划分成多个子目录树，并分派到不同的服务器上。Ceph 可以解决元数据分布和集群负载问题，但是其缺点也明显，在命名空间发生变化时可能带来较大元数据迁移。而且，访问较深层次的目录会有较高的延迟。

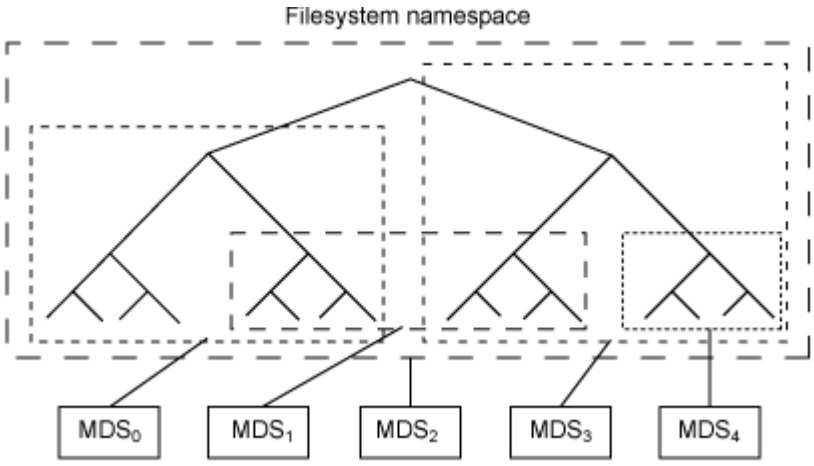

图 1-5　Ceph 目录划分结构

(3) 基于哈希的划分

基于哈希划分方法[29-31]是根据文件的某个特征值（路径名、文件名等）计算出哈希值，再通过哈希值映射到特定元数据服务器上。哈希划分方法有较小的划分粒度，

能以文件为单位进行划分，可以将元数据均匀的分布到元数据服务器集群中，具备负载均衡的能力。但同时也带来了元数据访问局部性差、重命名和集群变化会造成元数据迁移等问题。

除了以上三种比较典型的元数据组织方法之外，国内外还有一些关于分布式元数据管理的研究成果：

Kai Ren 等人在 2014 年提出的 IndexFS[32]系统，针对大数据环境中元数据数量大、目录中文件数量分布不均匀等特点，提出了位于应用程序和分布式文件系统之间的元数据管理系统。IndexFS 可以为采用了集中式元数据管理的分布式文件系统，如 HDFS、PVFS[33]提供分布式元数据管理服务。IndexFS 利用元数据服务集群和客户端缓存技术，提供了高性能的文件创建、查询、删除等操作，并利用 GIGA+[34]技术解决大目录中元数据的划分问题。然而，系统在执行重命名和大目录划分时都会发生元数据的迁移。

周江等人在 2014 年提出的 Clover[35]系统，采用基于目录划分的一致性哈希算法组织元数据。Clover 为所有目录的全路径名绑定一个 UUID，再根据 UUID 和一致性哈希算法将目录下的文件和目录本身分布到元数据服务器上。Clover 以目录为单位划分元数据，将同一目录下文件的元数据聚集在一起，有效的提高了元数据的访问局部性。系统通过全局的分布式目录哈希表，避免了因重命名文件带来的元数据迁移。但文章也存在一些不足，例如重命名目录过程复杂、目录遍历操作效率低等，此外，目录中包含大量文件时会造成单个元数据服务器负载过重。

肖中正等人在 2015 年提出的 CH-MMS[36]系统，采用了一种基于一致性哈希结构的元数据服务集群方案。CH-MMS 将元数据服务器和元数据分布到一致性哈希环上，并利用一致性哈希算法实现集群的负载均衡和可扩展性。CH-MMS 采用延迟更新表，减少元数据变更带来的瞬间大量数据迁移，让系统以更平滑的方式迁移数据，但还是会产生迁移。此外，CH-MMS 存在元数据访问局部性差的问题，特别是执行遍历目录操作时效率较低。

Vilobh Meshram 等人在 2011 年发表的文章[37] 提出使用 Zookeeper[38-39]集群作为元数据服务器，避免了维护服务器之间元数据一致性的问题。文章利用 Zookeeper 中 Znode 节点支持目录结构的特性，构建出整个系统的目录结构。由于 Zookeeper 采用了 ZAB（Zookeeper Atomic Broadcast）[40]协议保证数据的一致性。因此，文章通过增加 Zookeeper 节点实现元数据服务性能的提升，而不必担心不同节点上目录结构不一致的问题。这种方法本质上还是属于集中式元数据管理，所有 Zookeeper 节点保存的目录结构是相同的，都是系统完整的目录结构，并不具有扩展性。随着系统目录结构的扩大，Zookeeper 节点的内存资源将被耗尽。

综上所述，大数据存储的元数据管理面临着诸多挑战，需要提高元数据管理的性能和可扩展性，具体而言又包括以下几个方面：

(1) 既要有很高的性能又要有较好的访问局部性

基于哈希划分的方法能够实现元数据的快速访问，具有很高的性能。但是其访问局部性较差，如执行列表目录时非常低效，需要到多个服务器中查找同一目录下的元数据；基于子树划分的方法虽然具有较好的访问局部性，但当目录层次较深、集群负载变化时都会使其响应的延迟时间增加，很难实现高性能。

(2) 避免重命名操作造成的元数据迁移

无论哈希划分还是动态子树划分执行重命名操作都会导致元数据迁移。CH-MMS 系统使用延迟迁移的策略避免瞬时的大量数据迁移，但还是会造成迁移；Clover 系统通过引入全局的分布式目录哈希表避免重命名文件造成的元数据迁移，但对于目录重命名的支持并不好。

(3) 具备平滑的扩展性

元数据服务器集群扩展时不能影响其他元数据服务器。哈希划分和子树划分方法对集群的变化十分敏感，往往会导致大规模的迁移，影响正常元数据服务器的服务。通过一致性哈希算法可以降低集群变化带来的影响，但扩大集群还是会导致部分服务器上元数据的迁移。

(4) 具备将大目录中元数据均匀分布的能力

在强调元数据访问局部性的同时，又会带来大目录中元数据过于集中的问题。Clover 系统采用了基于目录划分的方式将同一层目录下文件的元数据保存在相同元数据服务器上。这可以提高元数据的访问局部性，但对于大小超过百万的目录[13-15]，又会造成服务器的负载过重。为解决这一问题，IndexFS 使用了 GIGA+技术对大目录中元数据进行划分，使其分布到不同的元数据服务器上。但 GIGA+每次执行目录的划分都会将其哈希空间的一半划分到其他服务器，在此哈希空间内的元数据也将被迁移。

1.3 研究内容

本文针对大数据存储中元数据管理面临的问题，提出一种文件路径和属性信息分离的分布式元数据组织方法，主要包括以下研究：

(1) 基于目录索引和桶的划分策略。提出一种以目录或小于目录为单位的划分方法，将元数据保存到桶（Bucket）中，再根据元数据服务器集群的负载情况将桶分布到不同的元数据服务器上。方法使得元数据管理具有较高的性能和较好的访问局部性。

(2) 基于 MDS 负载的分布策略。采用 MDS 服务器平均负载值和处理请求次数的比值计算 MDS 的负载能力，将元数据分布到处理能力更强的 MDS 中，使得 MDS 集群负载更加均衡。

(3) 构建全局目录索引的方法。提出一种基于键值对结构的全局目录索引方法。方法在提高元数据定位速度的同时，避免了重命名元数据带来的元数据迁移，支持均匀分布大目录中的元数据，并且使得元数据管理具有可扩展性。

(4) 元数据位置缓存策略。结合本文提出的元数据组织方法的特点，提出元数据位置缓存策略，策略解决了位置缓存信息不一致的问题，缩短了元数据管理流程，提高了操作效率。

1.4 本文结构

本文共五章，内容和组织如下：

第一章：介绍了本课题的研究背景及意义；概述了大数据存储系统的发展现状；描述了大数据存储模式的研究进展；详细说明了元数据组织的研究现状，并分析了当前面临的问题；最后，结合元数据管理面临的问题，提出了本课题的研究内容。

第二章：详细描述了文件路径和属性信息分离的分布式元数据组织方案，分析了大数据环境中存储系统的元数据管理面临的问题，给出方案的目标和整体架构；分别从元数据的划分、分布、元数据服务器集群的扩展和缓存元数据位置等方面详细阐述了方法的原理；最后，将本文方法和其他方法进行对比，并做出总结。

第三章：详细描述了元数据组织方关键技术及实现，给出了系统中的关键数据结构，描述了全局目录索引的构建过程，对方法提供的接口进行设计和实现，并说明了实现过程中使用到的相关技术。

第四章：测试和分析本文方法的性能和扩展性。对比分析了本文方法采用单一元数据服务器时与集中式元数据管理方法的性能，通过扩大本文方法的元数据服务集群规模测试了方法的扩展性，并与基于目录哈希的分布式元数据管理方法进行对比。

第五章：对全文进行总结与展望，总结本文的研究成果和对未来工作的展望。

2 元数据组织方案设计

本章主要介绍了一种文件路径与属性信息分离的分布式元数据组织整体方案。分析了在大数据环境下存储系统的元数据管理面临的问题，给出方案的目标和整体架构，并从元数据的划分、分布、集群扩展和元数据位置缓存等方面详细阐述了本文提出的组织方案的原理。最后，和其他元数据组织方法进行对比，并对本文方案做出总结。

2.1 目标

大数据环境中存储系统的元数据管理需要具备较高性能并支持扩展，结合 1.2.3 节的分析，元数据管理要有较高的性能，并且元数据有较好的访问局部性；具有可扩展性，使得元数据数量和管理性能随元数据服务器的增加而提升；能够避免重命名和元数据服务器集群扩展造成的元数据迁移；能够将大目录中元数据均匀分布。

为达到以上目标，本文设计了一种文件路径与属性信息分离的分布式元数据组织方案，将元数据组织成目录索引和元数据属性信息两个部分。通过构建全局目录索引，提高元数据管理的性能，避免重命名操作带来的元数据迁移，并且能够使得大目录中的元数据均匀分布；通过结合目录索引和桶，提高元数据的访问局部性，并使得元数据管理具备平滑的扩展性。

2.2 整体架构

本文提出的元数据组织方法，涉及多个模块之间的分工协作。因此，在说明元数据组织方法之前先介绍方案的整体架构。

如图 2-1 所示，整体架构包含目录索引模块（Index Service，IS）、元数据服务器集群（Metadata Server，MDS 集群）、定位服务模块（Location Service，LS）和客户端（Client）四个部分。

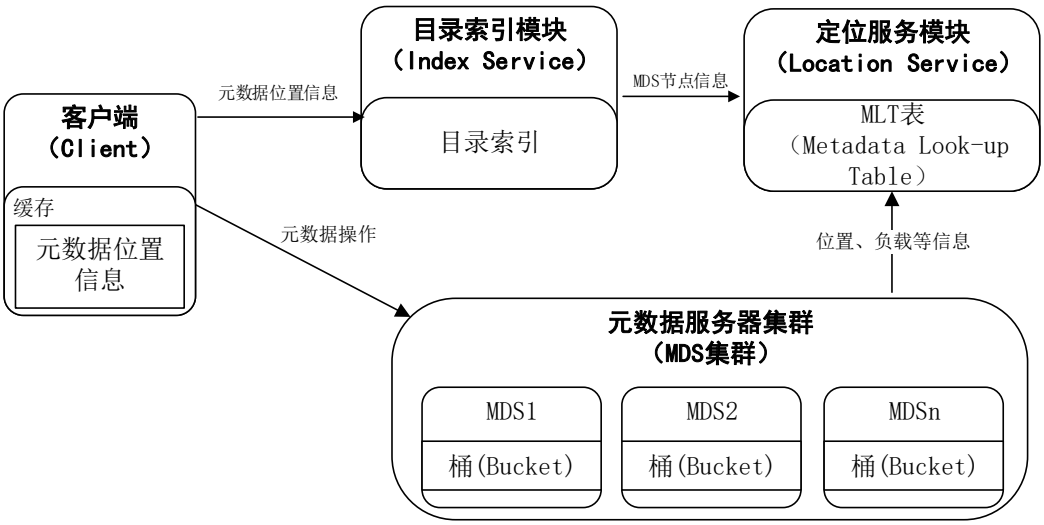

图 2-1　元数据组织方案整体架构

Client 是整个方法的入口，通过 Client 可以获得方法提供的服务，执行相关的元数据操作，Client 通过 IS 得到元数据位置信息，请求 MDS 集群执行元数据操作。IS（Index Service）维护了全局目录树的名字空间索引信息，索引包含目录到 MDS 的映射关系，IS 请求 LS（Location Service）获得元数据的位置。LS 维护 MLT 表，表中包含 MDS 集群的位置和负载信息，LS 根据 MLT 表计算出集群中能力最强的 MDS。MDS 集群以桶为单位存储元数据，通过桶号和文件名称能在桶内获取元数据。

在介绍各个模块的功能及结构信息之前，先对本文中出现的名词做出解释：

全局 ID：每个目录都有一个全局唯一的 ID。

分布编码：全局唯一的数字，元数据根据此数字保存到 MDS 的桶内，因此称为分布编码。

分布编码映射关系：保存分布编码和 MDS 编码的映射关系，可确定保存桶的 MDS。

2.2.1 目录索引模块

目录索引模块包含了全局目录的结构，负责维护目录索引、计算元数据位置信息。模块中包含的目录索引，如图 2-2 所示。目录索引为键值对结构，键是一个字符串，由父目录全局 ID 和目录名组合而成，值包含了两个元素：目录的全局 ID 和分布编

码映射关系。目录下元数据根据分布编码映射关系分布到 MDS 上，并根据关系中的分布编码保存到桶内。

键	值	
<父目录全局ID,目录名>	全局ID	分布编码映射关系 <分布编码1,MDS编号> <分布编码2,MDS编号>

图 2-2　目录索引结构

2.2.2 元数据服务器集群

集群由多台 MDS 组成，主要负责存取元数据。每个 MDS 包含多个桶，桶的结构如图 2-3 所示。桶号区分不同的桶，桶号为分布编码。桶内包含多个元数据，采用键值对结构，键为文件或目录名称，值为元数据。桶内的元数据有较好访问局部性，桶内保存的元数据属于同一个目录。

图 2-3　桶结构

2.2.3 定位服务模块

定位服务模块负责监控 MDS 集群的负载、为目录索引模块提供 MDS 编号和计算 MDS 地址信息。

编号	MDS IP地址	MDS端口	MDS平均负载	处理次数

图 2-4　MLT 表结构

模块记录和维护的信息保存在改进的 MLT（Metadata Look-up Table）表中，结构如图 2-4 所示，包含了 MDS 的编号、MDS 的 IP 地址、端口号、MDS 平均负载和处

理次数。

MLT 表中信息由 MDS 提供，MDS 启动时会向此模块注册 IP 和端口信息，并定期向模块发送平均负载以及处理请求的次数。

2.2.4 客户端

客户端是用户访问的入口，用户可以使用客户端执行相关元数据操作。客户端使用的元数据位置信息结构，如图 2-5 所示。根据位置信息客户端可以找到元数据所在的桶，包含了 MDS 的 IP 和端口以及分布编码。

MDS IP地址	MDS端口	分布编码

图 2-5 元数据位置结构

客户端根据缓存的元数据位置信息，缩短访问流程。缓存的位置信息，如图 2-6 所示。采用了键值对结构，键为目录路径，值为元数据位置信息。

键	值		
父目录路径	MDS IP地址	MDS端口	分布编码

图 2-6 缓存的元数据位置结构

2.3 基于目录索引和桶的划分策略

分布式元数据管理的关键在于如何划分元数据的名字空间，不同的划分策略带来的效果也不相同。但是，名字空间的划分策略不应过于复杂，避免给系统带来较多的开销，而且划分后的元数据应有较好的访问局部性。

在本文方法中，划分策略的基本思想是将同一目录下的元数据划分到一个或多个桶（Bucket）内。桶内的元数据属于同一目录，拥有较好的访问局部性。桶小于或等于目录的大小，目录大小超过桶的大小才会发生划分，而且划分过程简单不会带来较多的开销。

在分布式元数据管理中，一些元数据操作，如重命名，可能造成多个 MDS 上元

数据的迁移。为了避免这种情况的发生，引入了全局目录索引结构，如图 2-2 所示，用于建立目录到 MDS 的映射关系。因此，本文方法的名字空间通过三级映射机制实现。第一级采用目录路径计算到全局目录索引中的节点，得到目录的分布编码映射关系。第二级根据分布编码映射关系中的分布编码在 MLT 表中查找，得到保存桶的 MDS。第三级根据文件名称在桶内查找到元数据。

从文件的路径到保存文件的元数据位置的算法，如图 2-7 所示。对于目录或文件，首先通过其路径名（不包含目录或文件名称，即其父目录的全路径）计算出全局目录索引的节点：$CalNode(dirPath) = dirNode$得到目录或文件的父目录索引节点；通过索引节点中的分布编码 dCode 和 MDS 编号 MDSCode，计算$MLT(MDSCode) = MDS(i)$，得到 MDS 的位置信息；然后根据 dCode 和$MDS(i)$定位到桶，得到元数据位置信息。

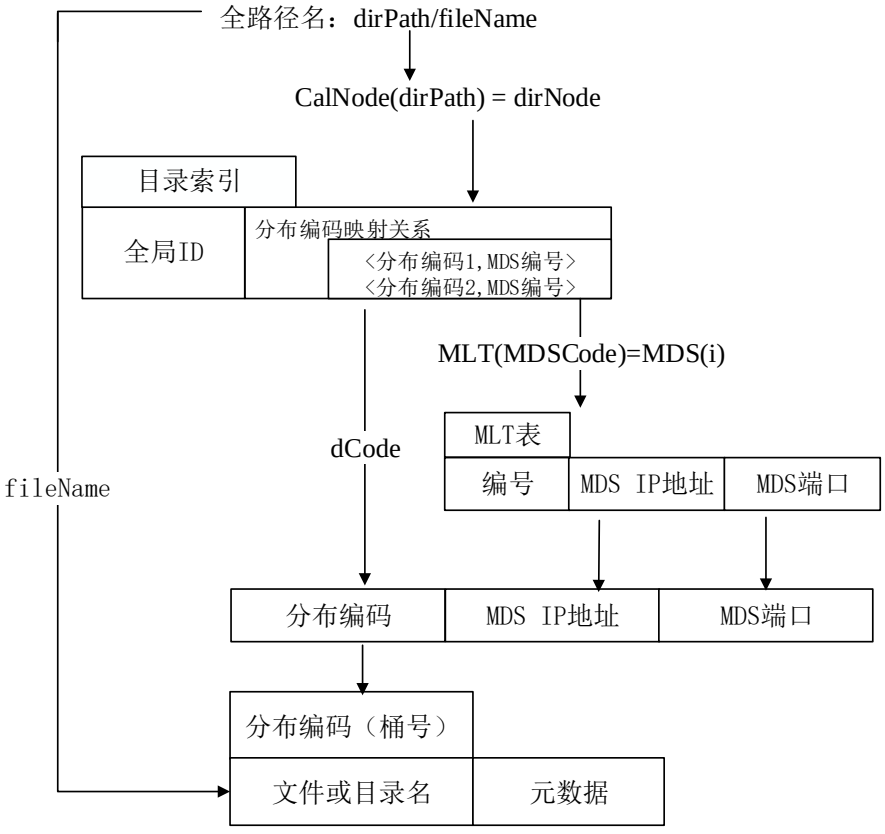

图 2-7　元数据位置信息

下面通过具体实例说明划分的过程，如图2-8所示，完成图中目录树的创建过程。目录树中根目录“/”、“/bin”、“/etc”、“/bin/dir”等都在目录索引中有对应的节点。目录树中的元数据分别映射到MDS0和MDS1上的桶内；根目录“/”的下一层目录“/bin”、“/etc”映射在10号桶内；“/bin”目录下元数据较多，被划分到19和30两个桶内。当执行重命名“/bin”目录为“/bin2”，只需更新目录索引中“0:bin”节点key为“0:bin2”和更新10号桶内“bin”的为“bin2”，避免不必要的元数据迁移。重命名文件操作更加简单，只需更新桶内的文件名，也不会导致迁移。

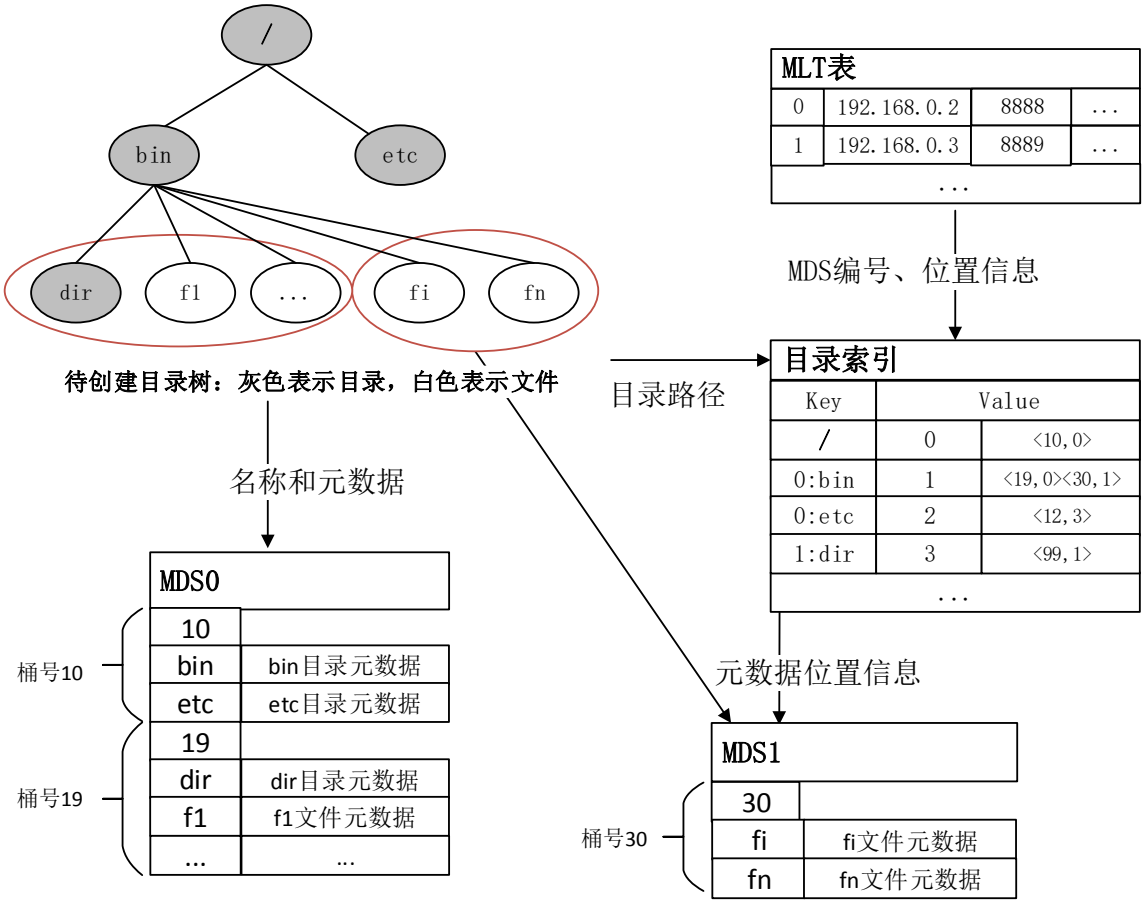

图 2-8　元数据名字空间划分策略

2.4 基于 MDS 负载的分布策略

元数据保存在MDS集群中，分布时需要避免负载能力强的MDS上管理少量元

数据，造成集群负载的不均衡。因此，分布元数据时需要考虑 MDS 节点的负载能力，根据节点负载能力进行分布。本文方法分布策略的关键在于计算每个 MDS 的负载能力，分布元数据时都选择能力最强的 MDS。

为了计算 MDS 节点的负载能力，一般用其处理器、内存容量等的加权和来表示，但这种方法需要针对每台 MDS 做实验才能得出具体的加权系数。因此，使用 MDS 的总热度与负载的比值来定义节点的负载能力[41]。

本文方法将两次统计之间，MDS 新增加的处理次数$L(count)$作为元数据总热度，将 MDS 节点的平均负载值$L(avg)$看成节点的负载值。因此，MLT 表结构中不仅记录了 MDS 的地址信息还记录了平均负载值和处理次数。其中$L(avg)$为 MDS 节点 1 分钟、5 分钟和 15 分钟的负载值的均值。

节点负载能力 Ability 计算方法如公式（2-1）所示：

$$Ability = L(count)/L(avg) \tag{2-1}$$

公式的含义：MDS 能以更小的负载处理更多的请求，表示此 MDS 的能力很强，所以 Ability 值越大表示节点能力越强。

为了让负载能力强的 MDS 多保存元数据、处理更多的元数据请求。每当保存元数据时，LS 计算 MLT 表中负载能力最强的 MDS，并将元数据保存到此 MDS 上。这样的分布策略使得能力强的 MDS 管理更多的元数据，而能力较弱的 MDS 管理较少的元数据，整个集群的负载更加均衡。

2.5 MDS 集群扩展策略

分布式元数据管理能够对 MDS 集群扩展，并通过扩展 MDS 集群提升管理元数据的数量和性能。扩展 MDS 集群时，应具有平滑的扩展性，不应该影响其他 MDS 的服务。

本文方法中采用监控机制发现新加入的 MDS 节点。MDS 加入集群时都会向负载监控的 Zookeeper 集群注册信息，之后定期向监控集群发送信息。LS 模块实时监控

注册信息并更新 MLT 表，完成 MDS 集群的扩展，新的元数据分布请求到来将会考虑新加入的 MDS。如图 2-9 所示，MDSn 节点加入集群时，向 Zookeeper 集群注册自身信息，包括 IP、端口、负载和处理次数（注册时未有任何请求，处理次数为 0），LS 通过 Watcher 机制发现 MDSn 的注册事件，并根据注册信息更新 MLT 表。

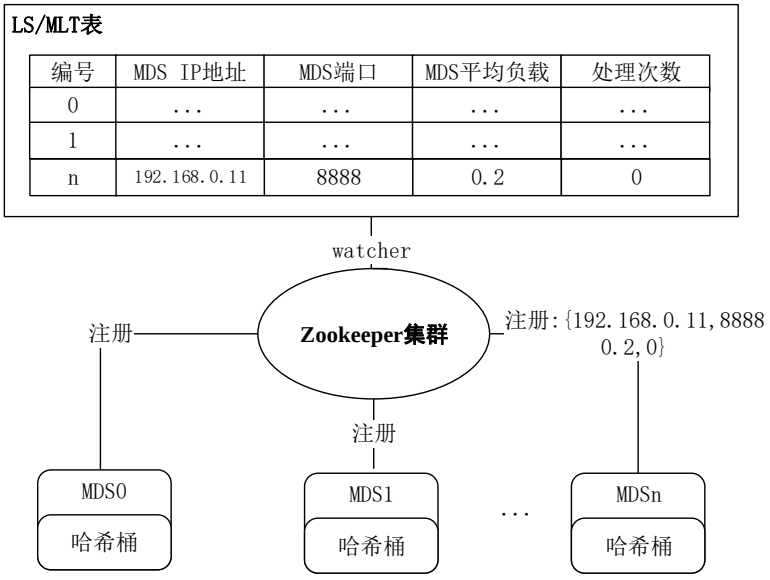

图 2-9　MDS 集群扩展策略

此外，本文方法引入的全局目录索引，持久化保存了目录与 MDS 的映射关系。新加入的 MDS 节点不会影响之前 MLT 表中 MDS 注册的信息，因此不会影响其他 MDS 的服务，也不会造成元数据迁移。

2.6 元数据位置缓存策略

根据本文方法中目录索引和元数据位置分开组织的特点，提出 Client 缓存元数据位置的策略。利用缓存的元数据位置，Client 可以直接请求 MDS 操作元数据，从而缩短管理流程。

2.6.1 缓存可行性分析

目录索引对应的元数据位置具有较高的稳定性，即使目录被重命名，元数据位置

也不会改变。只有当出现大目录时，Client 缓存的目录位置信息与 IS 端的不一致。Client 会根据缓存信息继续向超过阈值的桶内插入元数据，但是短时间内不会对 MDS 造成太大影响。

本文面临的缓存不一致问题，并非强一致性问题，允许 Client 缓存的信息和 IS 上最新信息出现短时间的不一致。此外，在大数据环境中大目录也属于少数情况。对于占大多数的小目录，Client 可以利用缓存信息提高执行效率。

因此，Client 缓存元数据位置信息是可行的，但需要解决缓存数据同步的问题。

2.6.2 缓存数据同步方法

本文采用事件通知和缓存过期相结合的方法，解决缓存数据的同步。同步方法原理如图 2-10 所示。Client 和 IS 都会在 Zookeeper 上 Watcher 一个节点，当 MDS 集群中出现桶中数据超过阈值时，MDS 会向 Watcher 的节点上添加此桶号，也就是分布编码。Zookeeper 的 Watcher 机制将此变化通知所有 Client 和 IS，Client 执行过期相应的本地缓存操作，IS 记录相应的分布编码。Client 因本地无缓存而向 IS 请求目录的位置信息，此时 IS 会为目录添加新的分布编码映射关系，并向 Client 返回最新的元数据位置信息，Client 缓存最新位置后缓存数据实现同步。

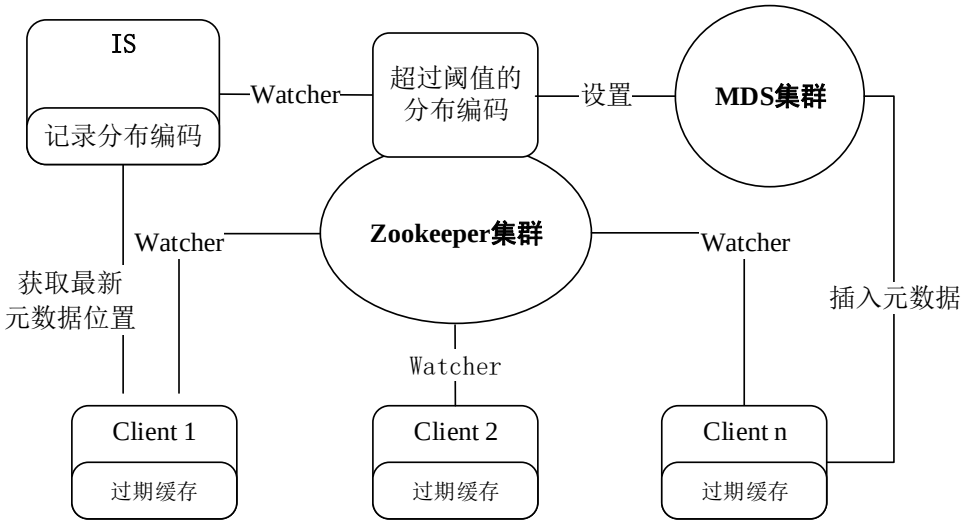

图 2-10　缓存数据同步方法

缓存数据同步的方式具有很高精度和较低资源占用率。Client 会启动后端监控线

程执行事件监听和缓存过期操作，线程一直阻塞到 Watcher 的节点发生变化，线程被唤醒后执行过期相应缓存的操作，不影响缓存的其他位置信息。

综上所述，本方案适合大数据环境下元数据的管理，能够兼顾小目录和大目录的情况。利用 Client 缓存元数据位置能够满足大部分小目录的元数据管理需求，而提出的缓存同步方法可以有效应对出现大目录的情况。

2.7 对比其他组织方法

分别从重命名和扩展 MDS 是否造成迁移、是否限制目录大小和列表目录的时间复杂度对比了本文方法和集中式、基于哈希划分和动态子树划分的方法，如表 2-1 所示。

本文方法的重命名操作不会造成元数据迁移，而基于哈希划分方法会因为映射关系变化而引起大量元数据迁移；具有不限制目录大小的能力，目录中可以包含任意多个文件和子目录，而集中式和动态子树划分中目录大小受限于操作系统对目录大小的设置；在扩展 MDS 时，本文方法不会发生元数据迁移，基于哈希和动态子树划分的方法都会发生迁移，而 HDFS 不具备扩展 MDS 的能力。在列表目录的时间复杂度方面，本问方法与 HDFS 和动态子树相同，具有较高的访问局部性能，能在 O(1) 的时间内得到结果，而基于哈希划分的方法需要跨多个节点，有较大延迟。

表 2-1　元数据组织方法的对比

方法名	重命名造成元数据迁移	不限制目录大小	扩展MDS造成元数据迁移	列表目录的时间复杂度
集中式（HDFS）	no	no	no	O(1)
基于哈希划分	大量迁移	yes	重哈希，大量迁移	low
动态子树划分	no	no	大量迁移	O(1)
本方法	no	yes	no	O(1)

2.8 方案总结

本方案旨在解决大数据环境下存储系统的元数据管理面临的问题，通过提出基于目录索引和桶的划分策略，划分和保存元数据；通过基于 MDS 负载的分布策略，使集群负载更加均衡；为使集群具备平滑的扩展能力，提出结合 Zookeeper 和 MLT 表的 MDS 集群扩展策略；此外，为缩短操作流程，提高元数据管理的性能，提出元数据位置缓存策略。综上所述，本文方案具有以下功能和特点：

(1) 较好的元数据访问局部性

本文方法的元数据划分策略以桶为单位聚集元数据（大目录中元数据会保存在多个桶内）。因为桶内只会存在同一目录下的元数据，而不会有其他目录下的元数据。因此，执行列表、遍历、删除目录等对访问局部性要求较高的操作时，只需获得桶就能访问到目录下所有的元数据。

(2) 可以平滑扩展 MDS 集群

MDS 加入集群时，先向监控集群注册信息，监控集群通知 LS 执行更新 MLT 表操作。新加入的 MDS 负载较轻、能力强，LS 会优先将新生成元数据保存到新加入的 MDS 上。随着时间推移，新加入的 MDS 上元数据会越来越多，其负载能力逐渐回落到集群的平均水平，LS 会在集群中选择能力更强的 MDS 保存新的元数据。此外，整个扩展过程具有平滑性，不会影响其他的 MDS，也不会造成元数据迁移。

(3) 重命名操作不会造成迁移

重命名文件时，文件和目录的元数据都是根据其父目录的映射关系保存到桶内。重命名文件元数据时，先得到元数据的位置信息，然后在桶内完成重命名文件操作，整个过程不会影响元数据与 MDS 的对应关系。因此，不会产生迁移。

重命名目录时，需要重命名索引和目录名。重命名索引时，只更新目录索引的键，不改变分布编码映射关系。因此，不会导致目录下元数据的迁移。而重命名目录元数据过程和文件重命名一致，也不会产生迁移。

(4) 具备将大目录中元数据均匀分布的能力

强调元数据访问局部性的同时，会带来另一个问题，就是大目录中元数据集中保存在一个 MDS 上，造成这个 MDS 的负载过重。本方案的元数据分布方法，当桶内的元数据数量超过阈值，就会为大目录增加保存元数据的桶，而新增加的桶会根据分布策略保存到 MDS 集群中，从而避免单个 MDS 负载的继续加重。

(5) 不限制目录大小

在大数据环境中遇到大目录情况时，为了避免单个目录大小超过操作系统的限制，通常采用软件的方法处理，将多个目录看成逻辑上的一个目录。但这样的处理过程逻辑比较复杂，性能较低。本文方法可以不限制目录大小，通过增加保存目录中元数据的桶即可应对目录的增大。

2.9 本章小结

本章主要描述了一种文件路径和属性信息分离的分布式元数据组织方案，分析了大数据存储中元数据面临的挑战，指出了本文方案的目标；针对目标提出了方案的整体架构，并描述了架构中各个模块的功能；分别从元数据名字空间的划分、分布策略、MDS 集群扩展策略和元数据位置缓存策略方面详细说明了本文方案的原理。最后，将本文方法和其他组织方法进行对比，并总结了本文方案，方案具备应对大数据存储中元数据管理面临的挑战的能力。

3 元数据组织方法关键技术及实现

本章主要描述元数据组织方法的关键技术及其实现，给出了方法中的核心数据结构，描述了全局目录索引的构建方法，并设计和实现了方法提供的元数据管理接口。最后，介绍了实现本文方法的相关技术。

3.1 核心数据结构

下面使用 Java[42] 语言描述本系统的关键数据结构。为了突出关键数据结构，只列出类中的属性，包含的方法都未列出。

3.1.1 目录索引类

```
public class DirIndex {
        private long dirId;          //全局 ID
        private Map<Long,Integer> distrCodeMap;      //分布编码映射关系
}
```

DirIndex 是目录索引类，有两个属性 dirId 和 distrCodeMap。其中 dirId 为 long 型，表示目录的全局 ID；distrCodeMap 为 Map 结构，表示分布编码映射关系，用于绑定分布编码和 MDS 编号，Map 中可以包含多个映射关系。

IS 中目录索引为键值对结构，键是由父目录 dirId 和此目录的名称组成的字符串，DirIndex 是目录索引的值。

3.1.2 元数据位置类

```
public class MdPos {
        private String ip;       //MDS 的 IP 地址
        private int port;   //MDS 的端口号
        private long distrCode;     //分布编码
}
```

MdPos 为元数据位置类，用于定位桶，包含了 ip，port 和 distrCode 三个属性。

ip 为字符型，表示 MDS 的 IP 地址；port 为 int 型，表示 MDS 的端口号；distrCode 为 long 型，表示分布编码。

MdPos 类由 IS 根据 DirIndex 的 distrCodeMap 属性生成，distrCode 为 distrCodeMap 的键，ip 和端口的值为 distrCodeMap 中的值对应的 MDS 信息。

3.1.3 MLT 类

```
public class MLT {
        private String ip;              //MDS 的 IP 地址
        private int port;               //MDS 的端口号
        private double load;                //MDS 的平均负载值
        private int count;                  //MDS 的处理请求数
}
```

MLT 类包含多个属性，其中 ip 为字符型，表示 MDS 的 IP 地址；port 为 int 型，表示 MDS 的端口号；load 为 double 型，表示节点的平均负载情况；count 为 int 型，表示两次负载统计间 MDS 新增的处理请求数。MLT 表是由多个 MLT 类组成的数组。

3.1.4 其他结构

(1) 元数据属性

文件和目录的元数据信息，包含访问权限、大小、创建和更新时间、文件块的分布等信息，保存在 MDS 的桶内。

(2) 桶结构

由三个部分组成，分别是桶号、桶内的键和桶内的值。在 MDS 中，桶的三个部分分别对应：分布编码、文件或目录名和元数据属性的 JSON 字符串。一个桶内可以保存多个元数据，可以根据桶号和名称访问到元数据属性信息。

3.2 构建目录索引

构建的目录索引过程是本方法的核心，涉及了 IS 和 LS 两个模块。目录索引影响

元数据的分布和定位，同时具有均衡 MDS 集群的负载等能力。目录索引采用了键值对结构，第一步生成目录索引的键，只涉及 IS 模块；第二步生成目录索引的值，包括生成全局 ID 和分布编码映射关系，涉及到 IS 和 LS 两个模块。

3.2.1 生成目录索引键

```
def traversalDirIndex(nameArray){
        dirId
        for (name : nameArray) {
            key = dirId + ":" + name
            dirIndex = findDirIndexByKey(key)
            if (dirIndex notExists) {
                throw error
            }
            dirId = dirIndex.dirId
        }
        return dirIndex
    }
```

图 3-1 查找目录索引伪代码

IS 首先从“/”开始遍历到待创建目录父目录的值，图 3-1 展示了查找过程的算法，根据每层路径名和父目录的全局 ID 查找每层目录的索引，一直进行到最后一层路径名。过程中只要出现键查不到值的情况，意味着创建目录的路径中有不存在的目录，会抛出错误，并结束构建过程。在得到父目录的索引值之后，组合父目录全局 ID 和待创建目录名，得到目录索引的键，生成键的过程完成。

3.2.2 生成目录索引值

IS 首先为目录生成全局 ID，再请求 LS 为目录生成分布编码映射关系，关系中包括分布编码和 MDS 编号。

LS 首先生成一个分布编码，再通过 MLT 表计算负载能力强的节点，并将 MDS 编号和分布编码绑定作为分布编码映射关系。假如 MLT 表中数据如图 3-2 所示，根

据公式（2-1），编号为 0 的 MDS 负载能力为 1000/0.2=5000，编号为 1 的 MDS 负载能力为 2000/0.5=4000。因此，编号 0 的 MDS 负载能力大于编号 1 的 MDS，LS 将编号 0 和生成的分布编码绑定。

编号	MDS IP地址	MDS端口	MDS平均负载	处理次数
0	192.168.0.10	8888	0.2	1000
1	192.168.0.11	8889	0.5	2000

图 3-2　MDS 在 MLT 表中信息

3.3 接口设计与实现

本文方法将系统中的文件分成文件和目录两类，如图 3-3 所示，接口包括创建、查找、重命名和删除文件或目录。针对目录有列表目录的接口，用于获得目录下所有元数据。

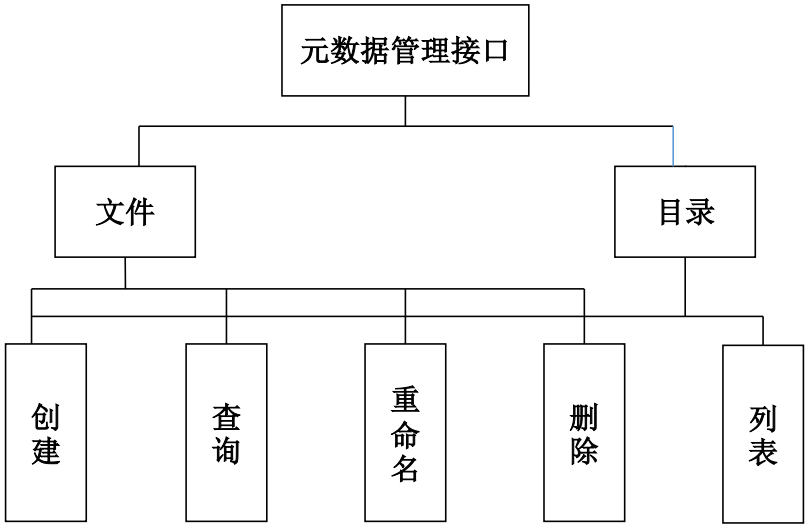

图 3-3 元数据管理接口

重命名元数据可以验证方法是否可以避免不必要的元数据迁移；列表及删除目录可以验证元数据是否有较好的访问局部性；此外，所有接口都可以验证方案的性能和扩展性。

3.3.1 创建元数据

(1) 创建目录元数据

创建目录元数据的时序图，如图 3-4 所示，整个创建过程包括两个步，第一步是 Client 向 IS 发起创建目录元数据的 DirIndex 请求，IS 根据请求创建 DirIndex 并返回保存目录元数据的位置信息 MdPos；第二步是 Client 根据 MdPos 信息将目录元数据保存到 MDS 上。下面分别描述这两个部分：

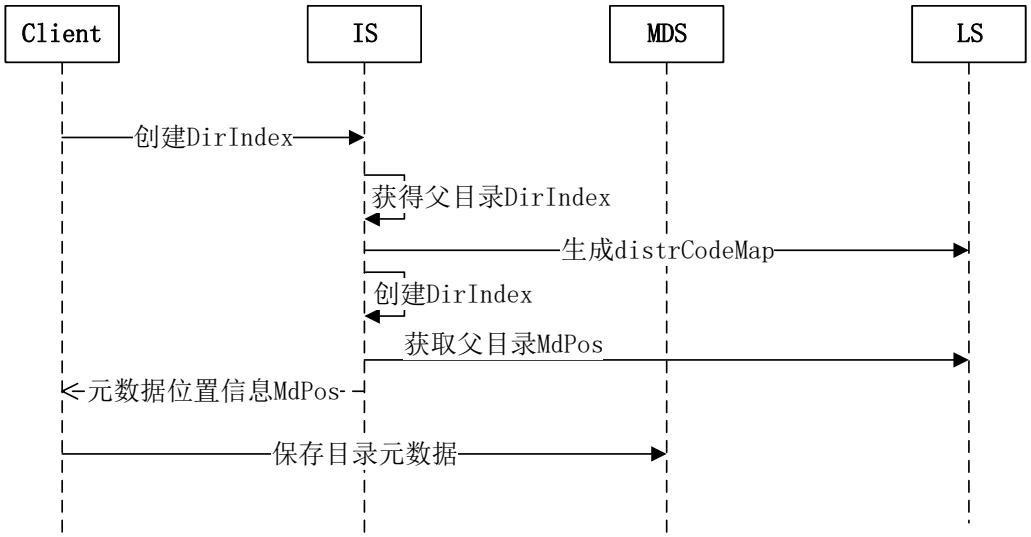

图 3-4 创建目录元数据时序图

创建过程第一步：Client 接收用户创建目录元数据请求，得到待创建目录路径、和目录元数据信息；Client 将待创建目录路径作为参数发送给 IS，请求 IS 创建 DirIndex；IS 根据路径参数得到待创建目录的父目录 DirIndex，并得到其 dirId 和 distrCodeMap 两个属性的值，IS 组合 dirId 和待创建目录名成为目录索引的键，并调用 LS 为待创建目录生成 distrCodeMap；IS 调用 LS 生成父目录的 MdPos 信息并返回 Client。此过程中，如果父目录 distrCode 对应桶中元数据数量超过阈值，IS 会为父目录生成新的 distrCodeMap，计算最新的 MdPos 后返回 Client；

创建过程第二步：客户端根据 MdPos，将 MdPos 中的 distrCode、待创建目录名

称和目录元数据发送到 MdPos 中 IP 和端口号对应的 MDS 上；MDS 查找服务器上桶号为 distrCode 的桶。如果桶不存在，则创建。最后向桶内插入目录元数据信息。

(2) 创建文件元数据

创建文件元数据时序图如图 3-5 所示，创建过程分为两步，第一步 Client 向 IS 发起查询待创建文件元数据位置 MdPos 的请求，第二步 Client 根据 MdPos 信息保存文件元数据到 MDS 上。

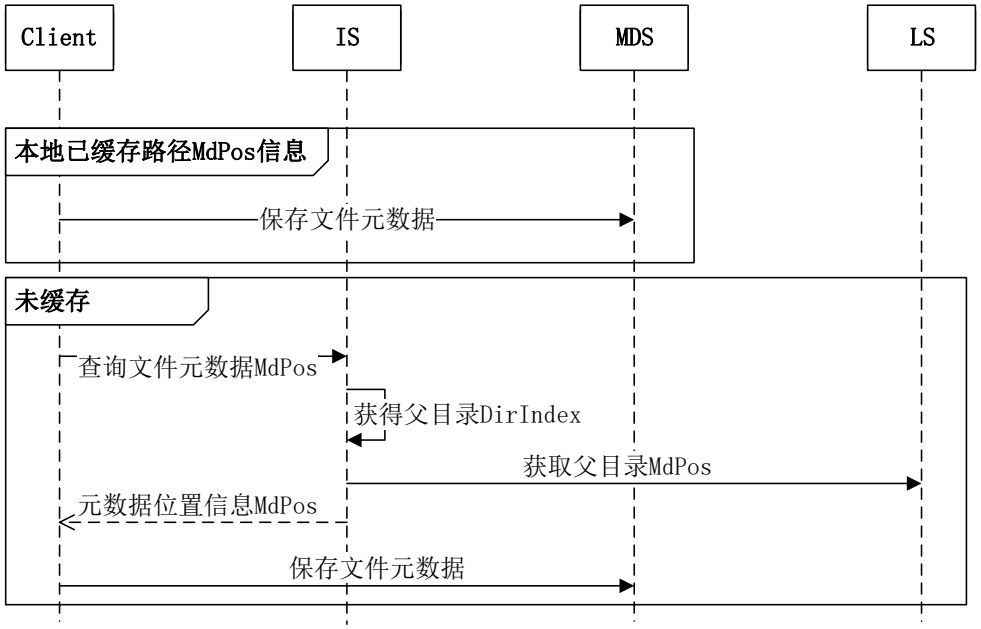

图 3-5 创建文件元数据时序图

创建过程第一步：Client 接收用户创建文件元数据请求，得到待创建文件路径和文件元数据信息；Client 判断本地是否缓存了待创建文件父目录所对应的 MdPos 信息，如果缓存，则直接进入创建过程第二步；否则 Client 请求 IS 查询元数据的位置信息 MdPos； IS 接到请求后查询待创建文件父目录的 DirIndex，并根据 DirIndex 属性 distrCodeMap 中最后一个键值对生成 MdPos，并返回给客户端。此过程中，如果父目录的 distrCodeMap 中 distrCode 对应桶中元数据数量超过阈值，IS 接受到相应目录的查询请求时，会为父目录生成新的 distrCodeMap，并将新的 distrCodeMap 对应的 MdPos 信息返回 Client；此外，哈希桶中元数据数量超过阈值后，Client 的后台线

程会过期缓存的 MdPos，并到 IS 查询最新的 MdPos。

创建过程第二步：此过程与创建目录元数据的第二步一致，Client 根据 MdPos 将文件元数据保存到 MDS 上。

3.3.2 查询元数据

查询文件和目录的元数据方法相同，主要包括两大步骤，首先 Client 获得待查询文件或目录元数据的位置信息 MdPos，然后 Client 根据 MdPos 和文件或目录名称在 MDS 上查询元数据信息。

查询过程第一步：Client 接收用户查询文件或目录元数据请求，得到待查询文件或目录的路径；Client 查询本地是否缓存了待查询文件或目录的父目录所对应的 MdPos 信息，如果有，进入查询过程第二步。否则，Client 请求 IS 查询元数据位置信息；IS 查询过程与创建文件元数据类似。不同点在于，IS 需要将父目录 distrCodeMap 对应的所有 MdPos 返回客户端，如果父目录是大目录，返回信息中会包含多个 MdPos，否则只有一个 MdPos；

查询过程第二步：Client 循环遍历 MdPos 列表，根据 MdPos 到对应的 MDS 上查找元数据。MDS 找到 distrCode 对应的桶，并根据文件名在桶内查找元数据。找到元数据后循环提前结束，否则持续到循环结束。

3.3.3 列表目录

列出目录中所有元数据信息，时序图如图 3-6 所示，主要包括两大步骤，首先 Client 获得待列表目录所有元数据所在的 MdPos，然后 Client 根据 MdPos 到 MDS 获取获取整个桶内的元数据。

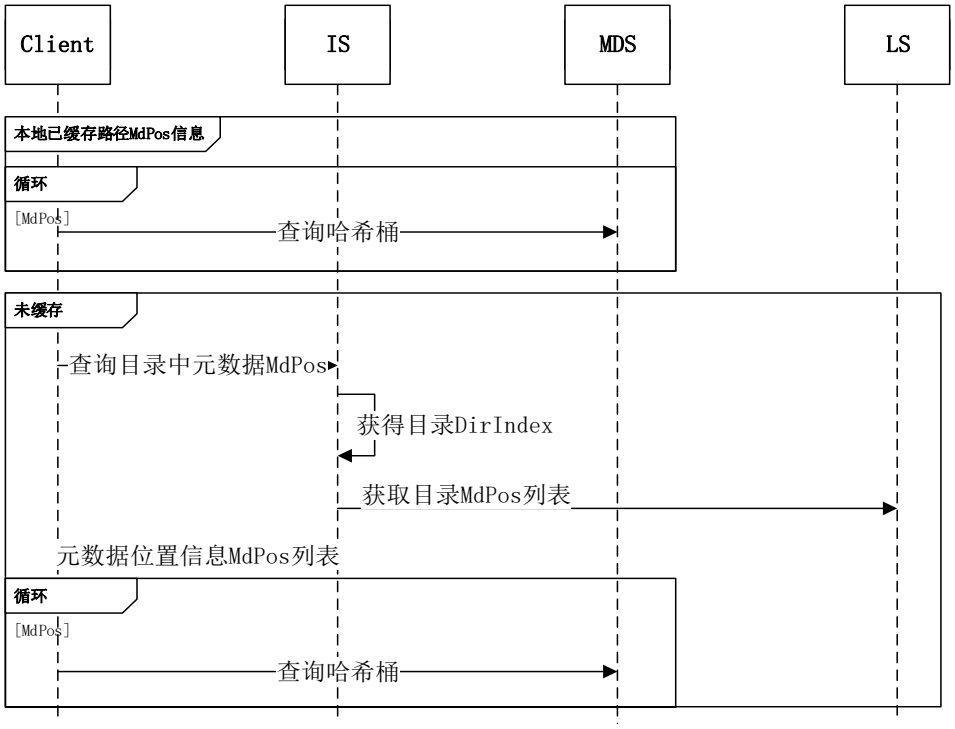

图 3-6　列表目录时序图

列表操作第一步：Client 首先查询本地是否缓存待列表目录的 MdPos 信息，如果有则进入第二步，否则，Client 通过 IS 获得待列表目录路径对应的 MdPos 信息列表，获得的过程与查询元数据过程第一步一致；

列表操作第二步：Client 循环遍历 MdPos 列表，在 MdPos 对应的 MDS 中查找桶号为 distrCode 的桶并将桶内所有的元数据返回 Client，直至循环结束。

3.3.4 重命名元数据

(1) 重命名目录元数据

重命名目录元数据过程如图 3-7 所示，Client 请求 IS 执行重命名操作，IS 完成重命名目录索引和重命名目录元数据两个操作。

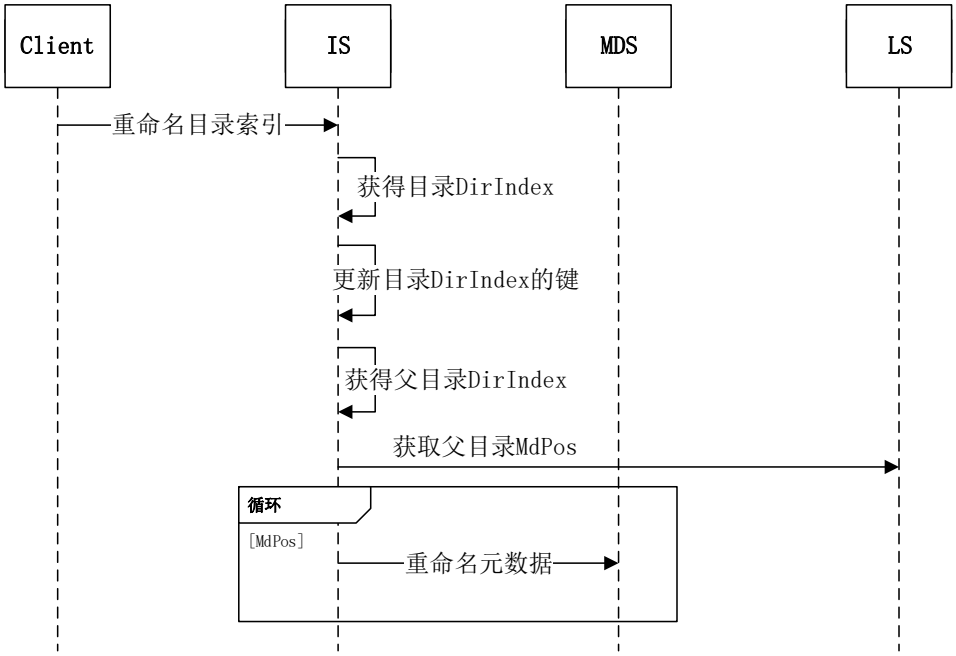

图 3-7　重命名目录时序图

Client 将待重命名目录的路径发送到 IS；IS 根据路径更新目录的 DirIndex，再根据路径获得父目录的 DirIndex，最后根据父目录 DirIndex 的 distrCodeMap 获得元数据位置信息 MdPos 的列表；IS 循环遍历 MdPos 列表，根据其中的 MdPos 到对应的 MDS 上执行重命名操作；MDS 根据 distrCode 和目录名找到目录元数据并更新其中的名字属性信息，返回成功，如果 MDS 上不存在对应的元数据，则返回空。重命名成功后退出循环，整个重命名操作结束。

(2) 重命名文件元数据

重命名文件包括两大步骤，首先 Client 获得待重命名文件元数据的位置信息 MdPos，然后 Client 根据 MdPos 到 MDS 执行重命名操作。

重命名操作第一步：Client 根据待重命名文件的父目录路径，得到其父目录 distrCodeMap 对应的 MdPos 列表，过程和查找元数据第一步类似，此处不再赘述；

重命名操作第二步：Client 循环遍历 MdPos 列表，根据其中的 MdPos 到对应的 MDS 上执行重命名操作；MDS 根据 distrCode 和文件名找到文件元数据并更新其中

的名字属性信息，返回成功，如果 MDS 上不存在对应的元数据，则返回空。重命名成功后退出循环。

3.3.5 删除元数据

(1) 删除文件元数据

删除文件元数据过程包括两大步骤，首先 Client 获得待删除文件元数据的位置信息 MdPos，然后 Client 根据 MdPos 到 MDS 执行删除文件元数据操作。

第一步：过程和查找元数据第一步类似，Client 根据文件路径名得到 MdPos 列表；

第二步：Client 循环遍历 MdPos 列表，根据其中的 MdPos 到对应的 MDS 上执行删除文件操作；MDS 根据 distrCode 和文件名找到元数据并删除，删除后会返回成功，如果 MDS 上不存在对应文件的元数据，则返回空。删除操作成功后退出循环。

(2) 删除目录元数据

删除目录元数据过程比较复杂，除了删除指定目录的元数据以外，还需递归删除其子目录及文件的元数据。

```
def findAllDirIndex(path,mdPosList){
        Queue dirQueue
        dirIndex = getDirIndexByPath(path)
        dirQueue.offer(dirIndex)
        while(!dirQueue.isEmpty()){
            index = queue.poll()
            mdPosList.add(buildMdPosByDirIndex(index))
            queue.addAll(findSubDirIndexByDirIndex(index))
        }
    }
```

图 3-8　查找目录及其子目录的 MdPos 伪代码

第一步：Client 执行删除待删除目录的元数据操作。

第二步：Client 将待删除目录路径发送给 IS；IS 根据路径执行递归删除，首先递归找到目录及其子目录的 dirIndex，然后根据 dirIndex 获取 MdPos 信息，递归获得

MdPos 的过程如图 3-8 所示，最后 IS 调用 MDS 执行删除桶操作。图 3-8 描述了查找目录及其子目录的 MdPos 的伪代码，算法使用队列避免目录层次过深而导致的栈溢出。算法从队列首部取出待并删除目录，将待删除目录的子目录插入队列尾部，过程持续到队列为空。

第三步：MDS 根据 MdPos 中 distrCode 删除对应的桶。

3.4 相关技术

3.4.1 节点通信技术

本文实现涉及多个节点的分布式调用和数据传输。使用 Java 语言自带的远程方法调用（Remote Method Invocation，RMI）[43]技术作为节点间通信技术。

本文实现中主要利用了 RMI 支持对象的特点，它可将完整的对象作为参数和返回值在不同机器上传递，具有更高的执行效率和开发便捷性。

3.4.2 Zookeeper 集群

Zookeeper 是由 Apache Hadoop 的子项目发展而来，于 2010 年 11 月正式成为 Apache 的顶级项目。Zookeeper 为分布式应用提供了高效且可靠的分布式协调服务，提供了如统一名字空间服务、配置管理和分布式锁等基础服务。Zookeeper 使用了 ZAB 协议解决分布式环境中数据的一致性问题。

本文实现利用 Zookeeper 集群作为节点监控和缓存数据同步的底层基础。

3.4.3 底层存储

(1) 目录索引模块

选用 RocksDB[44] 为目录索引模块的底层数据库。RocksDB 是由 Facebook 基于 Leveldb 开发的支持持久化的键值对数据库管理系统。RocksDB 使用磁盘空间存储数据，并且只占用很少的内存空间。它不仅支持一次获取多个键值对，还支持键的

范围查找。

本文实现利用了 RocksDB 的持久化存储、内存占用低和支持 Key 的范围查找的特点。在 IS 中使用 RMI 作为外部服务调用、数据传输的框架，底层数据的存储和查找由 RocksDB 支持。

(2) MDS 节点

选用 SSDB 作为底层数据库，SSDB 是一个高效的键值对持久化缓存系统，实现了 Redis 的接口，支持多种数据结构，包括键值对、哈希结构、集合和列表等。SSDB 支持持久化，宕机后数据不丢失，并且不受内存大小限制，适合存储大量数据[45]。此外，SSDB 支持主从复制和负载均衡，对整个集群的稳定性和负载均衡有很大作用。

本文实现利用了 SSDB 支持的哈希结构，使用哈希结构作为桶底层存储结构。

3.5 本章小结

本章介绍了元数据组织方法的关键技术及实现，描述了核心的数据结构，详细说明了构建目录索引的方法，并设计和实现方法支持的管理接口，对元数据的创建、查询、重命名和删除等功能的执行流程做出了详细的说明。最后，分别从节点间通信、Zookeeper 集群和底层存储方面列出了实现本方法的相关技术。

4 测试与分析

本章主要测试和分析元数据组织方法的性能和扩展性。介绍了本文测试的环境，分别从元数据管理的性能和扩展性、访问局部性、目录深度等角度进行了测试。对比了采用单一 MDS 与集中式元数据管理方法的性能，通过扩大 MDS 集群测试方法的扩展性，并与分布式元数据管理方法进行对比。

4.1 测试环境

测试平台由四台服务器组成，其中一台服务器用于部署 IS 和 LS，其他三台服务器部署 MDS 集群和 Zookeeper 集群。

本文选择 HDFS 代表集中式元数据管理方法，用于对比本方法的性能、访问局部性等。HDFS 部署在四台服务器上，其中一台为主服务器，三台数据服务器。

本文选择基于目录哈希的元数据管理方法代表分布式元数据管理方法，部署在 3 台 MDS 上，客户端直接访问 MDS 提供的元数据管理服务。

测试使用的服务器配置均相同，如表 4-1 所示。

表 4-1　服务器配置

组成	基本配置
CPU	4 Intel(R) Xeon(R) CPU　E5606 @ 2.13GHz
内存	8GB
硬盘	2TB
网卡	百兆网卡
操作系统	Linux master 2.6.32-45-generic-pae

4.2 性能和可扩展性测试

4.2.1 创建元数据

(1) 创建文件元数据

每个客户端分别在 100 个目录中创建 1000 个文件，桶的阈值设为 5000。测试结果如图 4-1 所示。图中横坐标表示客户端数量，纵坐标表示每秒操作数，分别列出了本方法使用 1 到 3 台 MDS、目录哈希方法和 HDFS 的测试结果，从测试结果中可以对比 1 台 MDS 与 HDFS 的性能和 3 台 MDS 与目录哈希方法的性能，以及 MDS 数量对本文方法性能的影响，其中目录哈希方法部署在 3 台 MDS 上。

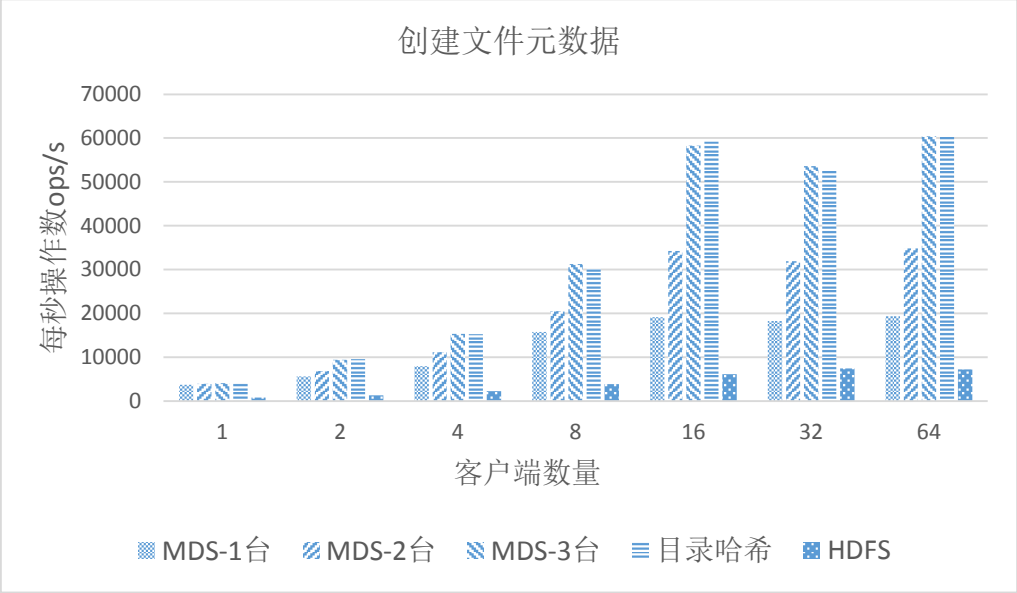

图 4-1 创建文件元数据

从图 4-1 中可以看出，本文方法具有很高的文件元数据创建性能，其性能随 MDS 数量的增加而提升，而且对高并发有较好的支持。与 HDFS 相比，1 台 MDS 的性能为 HDFS 的数倍。客户端数量为 1 时，1 台 MDS 的每秒操作数为 3659ops/s，而 HDFS 为 800ops/s；客户端数量为 64 时，1 台 MDS 为 19334ops/s，HDFS 为 7233ops/s。

从图中还可以看出，本文方法采用 3 台 MDS 时的性能和目录哈希方法的性能几乎相同。因为两种方法创建文件元数据的原理类似，都是根据目录路径计算得到保

存元数据的 MDS 信息。本文方法虽然需要从 IS 模块获取元数据位置信息，但通过缓存元数据位置将操作效率提高到与目录哈希方法相同。

随着 MDS 数量的增加，本文方法性能逐渐提升。当客户端数量为 4 时，1-3 台 MDS 的操作数分别为 7911ops/s、11139ops/s、15291ops/s，分别提升了 40%和 37%；客户端数量为 64 时，分别为 19334ops/s、34789ops/s、60397ops/s，分别提升了 78%和 76%。性能提升的原因是，本文方法利用目录索引和桶结构将同一目录下的元数据分布到相同的 MDS 上，具有较好的访问局部性。创建文件时，客户端能够根据缓存的位置信息直接与 MDS 交互，完成文件的创建。因此，这方法可以通过增加 MDS 数量，提高整体的创建能力。

(2) 创建目录元数据

每个客户端分别创建 10000 个目录，桶的阈值设为 5000，测试结果如图 4-2 所示。图中横坐标表示客户端数量，纵坐标表示每秒操作数，分别列出了本文方法使用 1 到 3 台 MDS、目录哈希方法和 HDFS 的测试结果，其中目录哈希方法部署在 3 台 MDS 上。

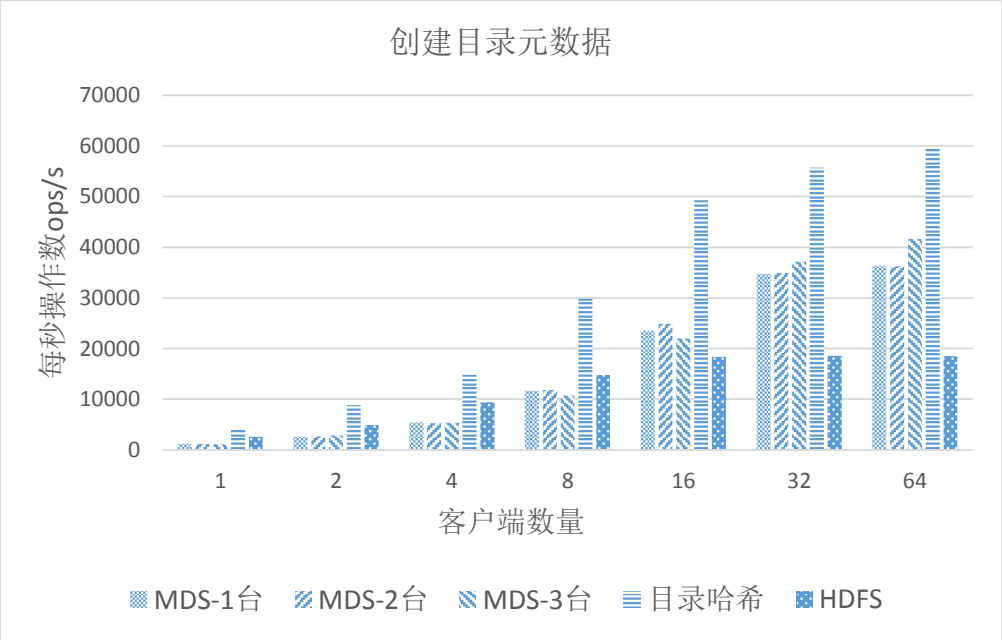

图 4-2　创建目录元数据

图 4-2 中，客户端数量从 1 到 8 时，此时并发度较低，HDFS 创建目录的性能比 1-3 台 MDS 的性能要高 90%~40%左右，这是因为 HDFS 采用集中式元数据管理，目录创建操作在内存中完成，具有较高的性能。而在本方法中目录索引和目录元数据分开管理，创建时需要先从 IS 的磁盘中加载和处理相关信息，然后将元数据保存到 MDS 中，整个过程与 HDFS 相比多了磁盘和网络开销。但是，随着客户端数量的增加，并发度也在提升，本文方法的磁盘和网络开销被抵消，逐渐表现出较高的性能。而 HDFS 因为内存使用空间的增加，性能逐渐稳定在 18000ops/s。

当客户端数量为 16 时，本方法 1 台 MDS 比 HDFS 性能提升了 27%，达到 23529ops/s。当客户端数量为 64 时，3 台 MDS 性能达到最高，比 HDFS 提升 127%，比 1 台 MDS 提升 13%。因此，本文方法有较好的扩展性，对高并发有较好的支持。

此外，由于目录哈希方法创建文件和目录元数据的原理相同，可以直接根据路径哈希值得到存取元数据的 MDS。因此，在测试结果表现出很高的性能，在客户端数量为 64 时，比本文方法性能高出 43%，比 HDFS 提高 2.3 倍。

4.2.2 查询元数据

查询文件和目录元数据过程相同，因此测试了查询文件元数据。

数据准备：为每个客户端准备 100 包含 1000 个文件的目录。客户端需要查询所有目录中文件的元数据，测试结果如图 4-3 所示。图中横坐标表示客户端数量，纵坐标表示每秒操作数，分别列出了本文方法使用 1 到 3 台 MDS、目录哈希和 HDFS 的测试结果，其中目录哈希方法部署在 3 台 MDS 上。

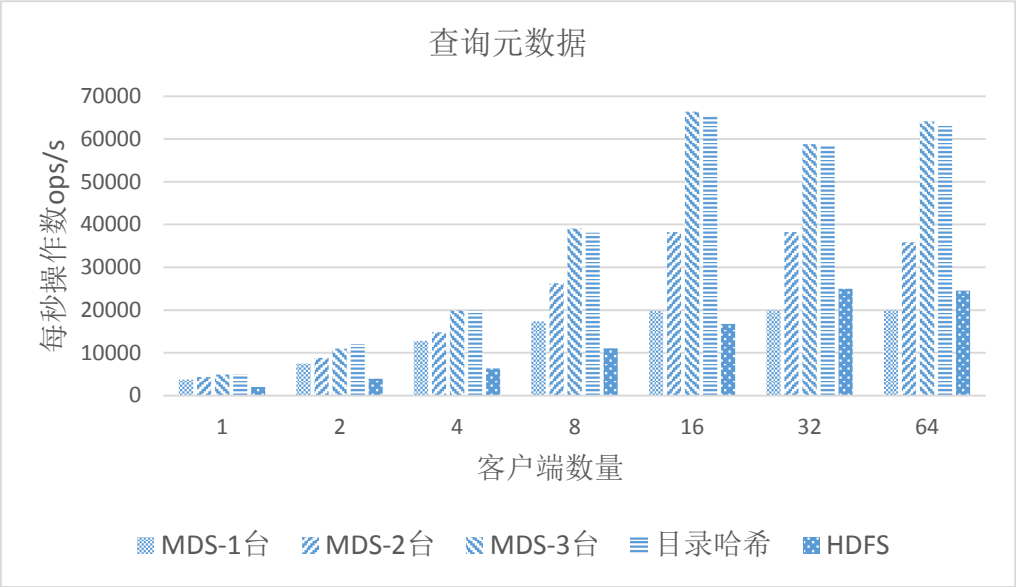

图 4-3　查询元数据

从图 4-3 中可以看出，本文方法元数据的查询具有较高的性能，而且性能随着 MDS 的增加而提升，并且对高并发有较好的支持。与 HDFS 相比，1 台 MDS 性能超过其数倍。客户端数量为 1，一台 MDS 为 3720ops/s，HDFS 为 1969ps/s；客户端数量为 16 时，一台 MDS 为 19833ops/s，HDFS 为 16733ops/s。

随着 MDS 数量的增加和并发度的提高，方法的性能逐渐提升。当客户端数量为 4 时，1-3 台 MDS 的每秒操作数分别为 7911ops/s、11139ops/s、15291ops/s；客户端数量为 64 时，分别为 19334ops/s、34789ops/s、60397ops/s。因为，本文方法可以将目录下的元数据分布到 MDS 的桶内，查询文件元数据时，客户端能够根据缓存的位置信息定位到桶，有较高的效率。因此，本文方法可以通过增加 MDS 数量，提高方法的整体查询能力。

本文方法采用 3 台 MDS 的性能和目录哈希方法性能相同，都处于较高水平。因为与创建文件元数据相同，两种方法都是根据路径计算元数据的位置信息，而本文方法通过位置缓存策略将性能提高到和目录哈希方法相同。

4.2.3 重命名元数据

(1) 重命名文件元数据

数据准备：为每个客户端生成 100 个包含 10000 个的目录。测试结果如图 4-4 所示。图中横坐标表示客户端数量，纵坐标表示每秒操作数，分别列出了本文方法使用 1 到 3 台 MDS、目录哈希和 HDFS 的测试结果，其中目录哈希方法部署在 3 台 MDS 上。

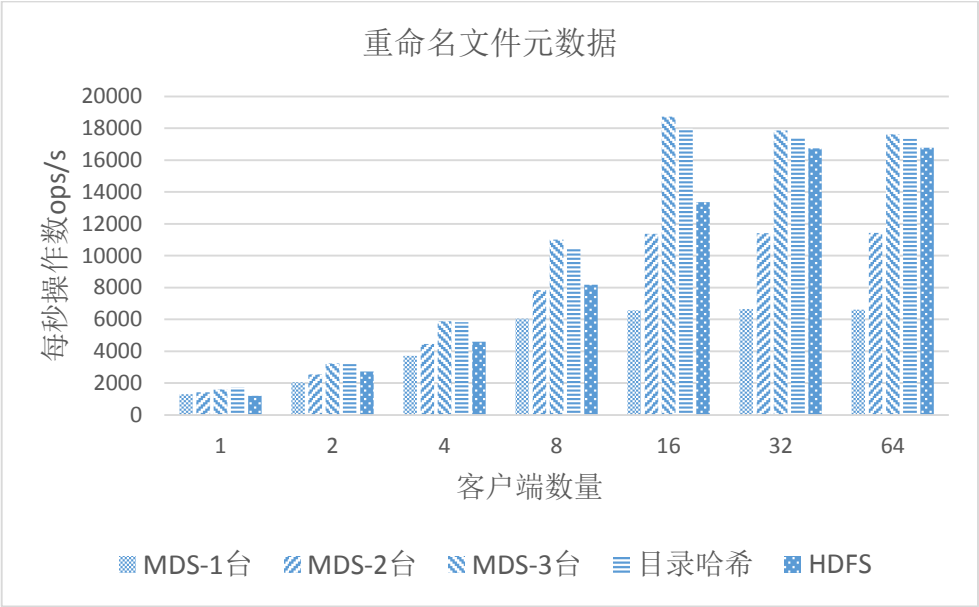

图 4-4 重命名文件元数据

从图 4-4 可以看出，本文方法重命名文件元数据不会造成元数据迁移，而且性能随 MDS 数量的增加而提升。客户端数量为 8 时，1-3 台 MDS 性能分别提升了 30% 和 57%。当客户端达到 16 之后，本文方法的的性能逐渐稳定，3 台 MDS 的性能稳定在 1.7 万 ops/s 左右。性能稳定的原因是，重命名文件时需要先获取元数据位置并得到元数据，然后将元数据信息解码成 Java 对象，完成元数据更新后再将其编码保存到 MDS 中，属于 CPU 和网络密集型操作。并发数量的上升使得模拟客户端的服务器性能、网络带宽等达到了极限，导致无法继续增加请求数量，而且高并发的查找和插入也给 MDS 带来很大压力，随着 MDS 数量的增加压力将得到分散，所以整

体的性能随着 MDS 的增加而上升。因此，可以通过增加 MDS 数量提升整体性能。

从图中还可以看出，当客户端超过 2 时，HDFS 的性能超过了本文方法采用 1 台 MDS 时的性能。这是因为，HDFS 采用集中式元数据管理，重命名操作在内存中完成，性能很高。而本文方法可以通过增加 MDS 的方式提高性能，同样在客户端数量为 2 时，2-3 台 MDS 的性能比 HDFS 要高。

此外，目录哈希方法之所以具有较高的性能是因为方法本身将目录下元数据聚集在同一个目录下，重命名文件时只需要修改文件元数据中信息，并不会造成迁移。而当重命名目录时，需要执行迁移操作，这种方法的劣势也将体现出来，下小结将详细分析。

(2) 重命名目录

数据准备：为每个客户端准备 1000 个用于重命名的目录。测试结果如图 4-5 所示。图中横坐标表示客户端数量，纵坐标表示每秒操作数，分别列出了本文方法使用 1 到 3 台 MDS、目录哈希和 HDFS 的测试结果，其中目录哈希方法部署在 3 台 MDS 上。

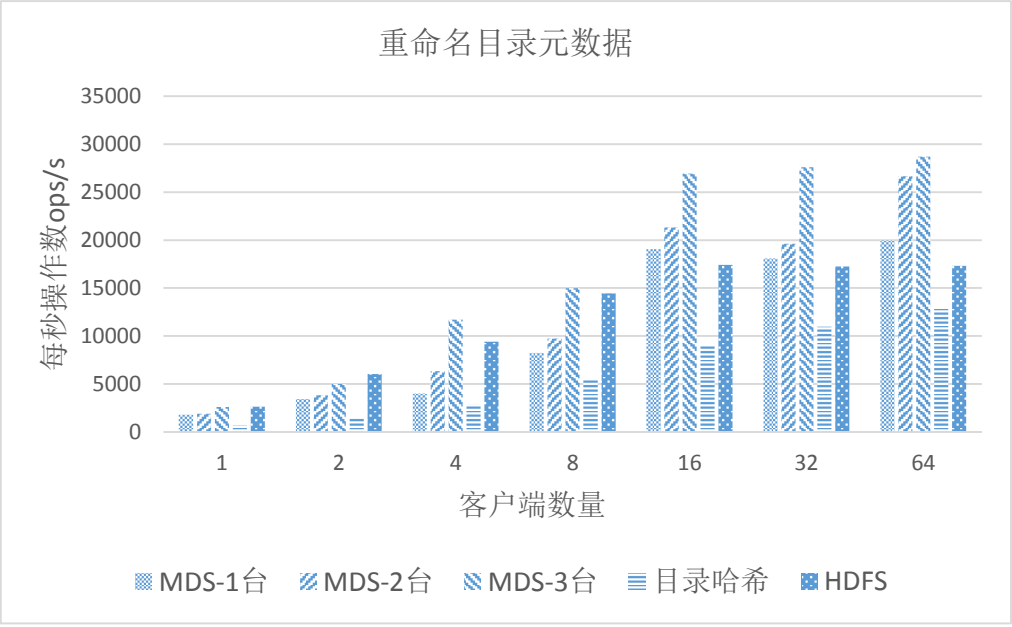

图 4-5 重命名目录元数据

从图 4-5 可以看出，本文方法的重命名目录元数据操作具有很好的扩展性，性能随着 MDS 的增加而提升。客户端数量为 16 时，1-3 台 MDS 性能分别提升了 10%和 23%。

与 HDFS 相比，本文方法表现了较好的高并发支持。客户端为 2 时，HDFS 性能超过了本文方法使用 3 台 MDS 的性能，但客户端的增加到 16 时，1 台 MDS 的性能也超过了 HDFS。这是因为，本文方法的磁盘和网络开销被大量的请求均分，并发度越高，性能越好。因为重命名目录使用的数据量小于重命名文件时的数据量，所以在客户端为 64 时也未出现性能平稳的情况。此外，重命名目录操作是客户端发起，由 IS 执行。IS 完成目录索引更新后，利用线程池启动的后台线程去执行 MDS 上的元数据更新操作，比客户端重命名文件操作过程简单，资源利用率也更高。所以，性能高于重命名文件的性能。

与目录哈希方法对比，本文方法采用 3 台 MDS 时的性能是其两倍左右。这是因为重命名目录操作本文方法只需要更新目录索引中的键值以及 MDS 中目录元数据中名称属性，并不会导致元数据迁移，而目录哈希方法则需要重新计算路径哈希值并将目录下所有元数据迁移到新计算出的 MDS 上，并且其性能随着目录中元数据数量的增加而下降。

4.2.4 删除元数据

(1) 删除文件元数据

数据准备：为每个客户端准备 100 个包含 1000 个文件的目录。测试结果如图 4-6 所示。图中横坐标表示客户端数量，纵坐标表示每秒操作数，分别列出了本文方法使用 1 到 3 台 MDS、目录哈希和 HDFS 的测试结果。

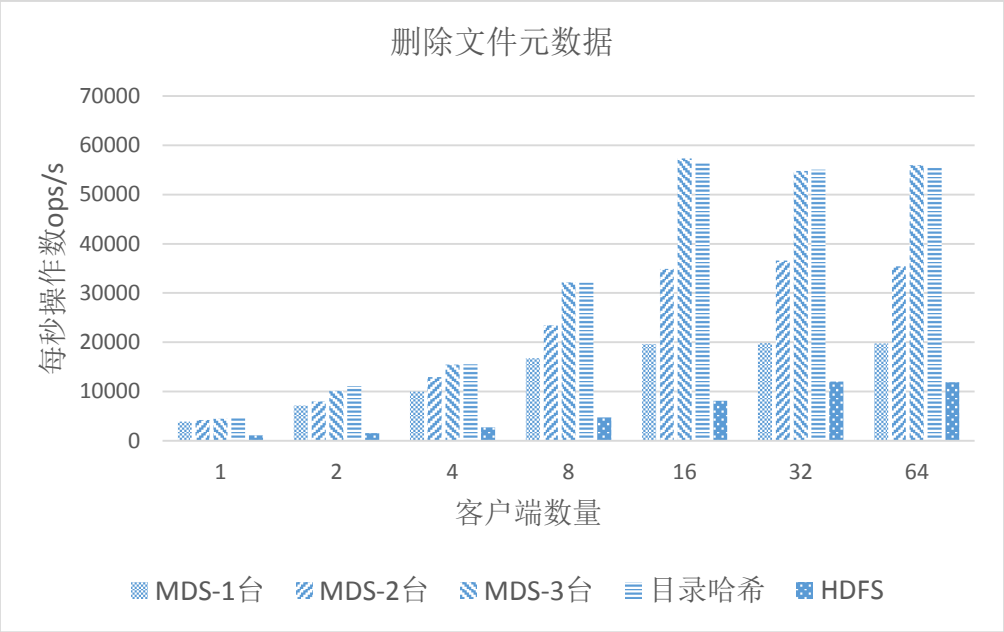

图 4-6　删除文件元数据

从图 4-6 中可以看出，本文方法采用 1 台 MDS 的性能是 HDFS 的数倍。客户端数量为 1 时，1 台 MDS 和 HDFS 每秒操作数分别是 3906ops/s、1067ops/s，客户端数量增加到 64 时，操作数分别是 19723ops/s、11860ops/s，提升了 72%。

本文方法采用 3 台 MDS 时性能和目录哈希方法性能相同，当客户端数量超过 16 时每秒删除数量都在 5000 以上。这是因为，定位和删除文件元数据时两种方法的原理相同，都是通过路径计算元数据位置并在 MDS 上执行删除操作。而且两种方法都利用了多 MDS 提高并发性，随客户端数量上升性能也逐渐提升。

本文方法的性能随 MDS 数量的增加而上升。客户端数量相同时，MDS 数量增加，性能也在提升。如客户端为 1 时，3 台 MDS 的性能比 1 台 MDS 提升了 15%。本文方法特别适合高并发的情况，性能随客户端数量的增加而上升，MDS 数量不变时，客户端数量增加，性能也在上升，MDS 为 2 台时，在 16 个客户端比 4 个提升了 183%。

(2) 删除目录元数据

数据准备：为每个客户端准备 10 层目录，每层目录中包括 10 个子目录和 1000 个文件。每个客户端从顶层目录开始执行递归删除，由于测试结果数量级不同，为

了便于显示，将测试结果分别在图 4-7 和图 4-8 中展示。图中横坐标表示客户端数量，纵坐标表示完成删除的时间开销，分别列出了本文方法使用 1 到 3 台 MDS 和 HDFS 的测试结果，以及本文方法使用 3 台 MDS 和目录哈希方法的测试结果。删除目录时，既删除了目录又删除了文件，不便于计算每秒操作数。因此，通过比较执行完操作所花费的时间比较性能。

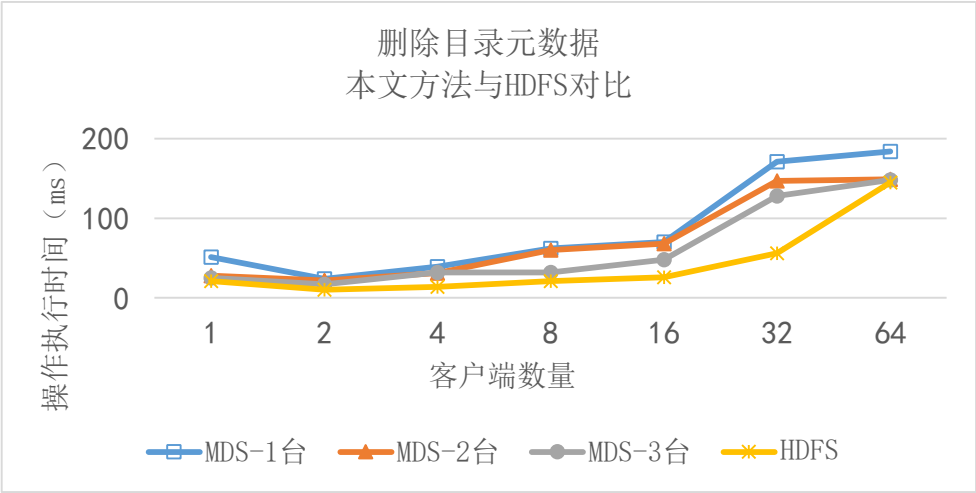

图 4-7　本文方法和 HDFS 操作时间对比

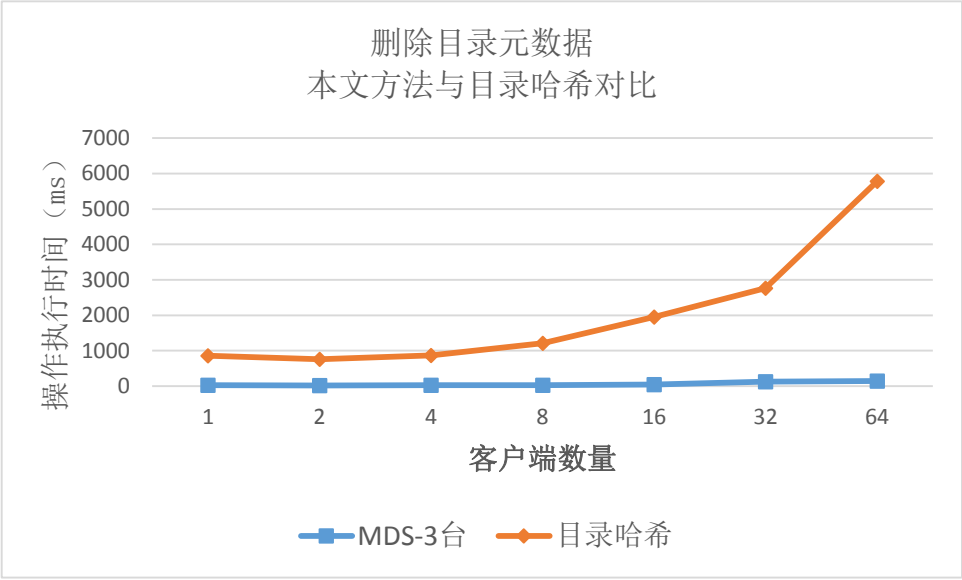

图 4-8 本文方法和目录哈希方法操作时间对比

图 4-7 中可以看出，方法具有较高的元数据删除性能和元数据访问局部性。删除

目录时需要完成所有子目录的删除，过程较其他操作更为复杂。HDFS 采用集中式元数据管理，在内存中完成删除操作，有很高的性能。本文方法在客户端数量从 1-8 时，完成最多有 80 层深，包含了 80 个子目录和 8000 个子文件的目录递归删除，总的时间开销在 62ms 以内，其中 3 台 MDS 时为 32ms，HDFS 为 21ms，虽然比 HDFS 性能差，但时间开销比较小。随着客户端数量的增加，本文方法和 HDFS 花费的时间都有所增加，当客户端为 64 时，HDFS 随着要删除的数据的增加，性能出现下降，而本文方法对高并发有较好的支持，两者的性能持平，本文方法使用 3 台 MDS 耗时为 148ms，HDFS 为 145ms。

从图 4-7 中还可以看出，1-3 台 MDS 完成的时间几乎相同，这是因为删除操作主要开销为递归的删除目录索引，与 IS 模块的处理能力有关。但从测试结果可以得出，IS 模块的性能与集中式的（HDFS）接近，位于高性能水平。

图 4-8 中可以看出，在递归删除目录元数据时，本文方法比目录哈希方法的延时更少，当客户端数量为 8 时，本文方法延时为 32ms，目录哈希延时为 1211ms。随着客户端数量的增加两种方法延时都在增加，但是本文方法增加的幅度远小于目录哈希方法。这是因为目录哈希方法必须在客户端判断删除目录中是否包含子目录，并对子目录执行递归删除，整个过程有多次网络传输，而本文方法利用构建的目录索引直接由 IS 模块获得待删目录的子目录结构，并对元数据执行批量删除，从而减少网络传输次数，降低延迟。

4.3 访问局部性测试

有较好的元数据访问局部性的组织方法可以将相同目录下的元数据聚集在一起，以便执行目录的遍历或列表时，具有较低的响应延迟。测试目录的遍历，需要递归的遍历目录下的子目录结构。

数据准备：每个客户端需要遍历 10 层目录，每层目录中包括 10 个子目录和 1000 个文件，阈值设为 1000。当客户端数量为 64 时，共需要遍历目录 640 层，其中子目

录 6400 个和文件数量 6.4 万，测试结果如图 4-9 所示。图中横坐标表示客户端数量，纵坐标表示遍历的时间开销，分别列出了本文方法使用 1 到 3 台 MDS、目录哈希和 HDFS 的执行时间。

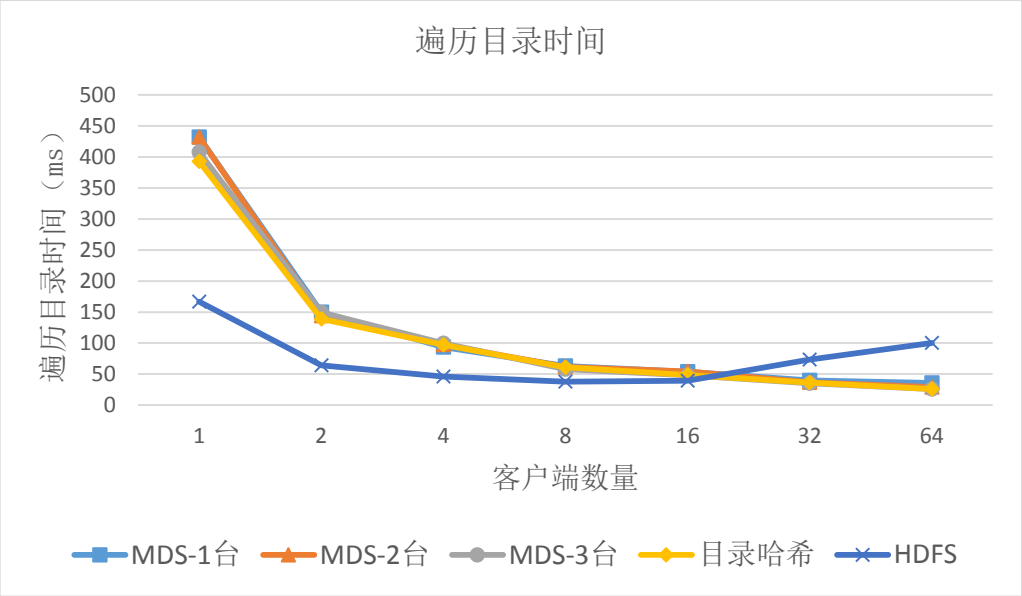

图 4-9　遍历目录时间

从图 4-9 中可以看出，本文方法具有较好的元数据访问局部性。HDFS 采用集中式元数据管理，具有很好的元数据访问局部性，遍历操作是在内存中完成，效率非常高。本文方法的执行时间在 2 到 16 个客户端时与 HDFS 花费的时间几乎相同，均在 150ms 内。

随着和客户端数量的继续增加，本文方法表现出较高的并发支持度，性能超过 HDFS，在 32 到 64 个客户端时完成所有目录遍历的时间小于 HDFS。本文方法 1-3 台 MDS 花费的时间几乎相同，这是由于遍历目录需要频繁的访问 IS 模块，这与 IS 模块的处理能力相关。从测试结果可以看出，IS 模块并非方法的瓶颈，因为本文方法的性能达到甚至超越了集中式元数据管理的性能。

此外，目录哈希方法遍历目录时间和本文方法基本相同，目录哈希方法会稍快几毫秒，因为本文方法中目录元数据被划分到多个桶中。因差距较小，两者的测试结果重叠在一起。两种方法都有较高的访问局部性，目录哈希方法是以目录为单位聚

集元数据，而本文方法是以桶为单位聚集元数据，都能够较快的访问目录下的元数据。

4.4 目录深度测试

元数据的创建和访问过程受目录深度的影响较大，目录越深时间开销越大。而不同的组织方法对目录深度的敏感程度也不相同。

测试采用固定客户端数量的方式，让每个客户端创建出不同深度的目录树。客户端数量为 16，分别创建 1 到 100 层的目录，每层目录都有 10 个子目录和 100 个文件。测试结果如图 4-10 所示。图中横坐标表示目录深度，纵坐标表示平均创建的时间开销，分别列出了本文方法使用 1 到 3 台 MDS、目录哈希和 HDFS 的测试结果。

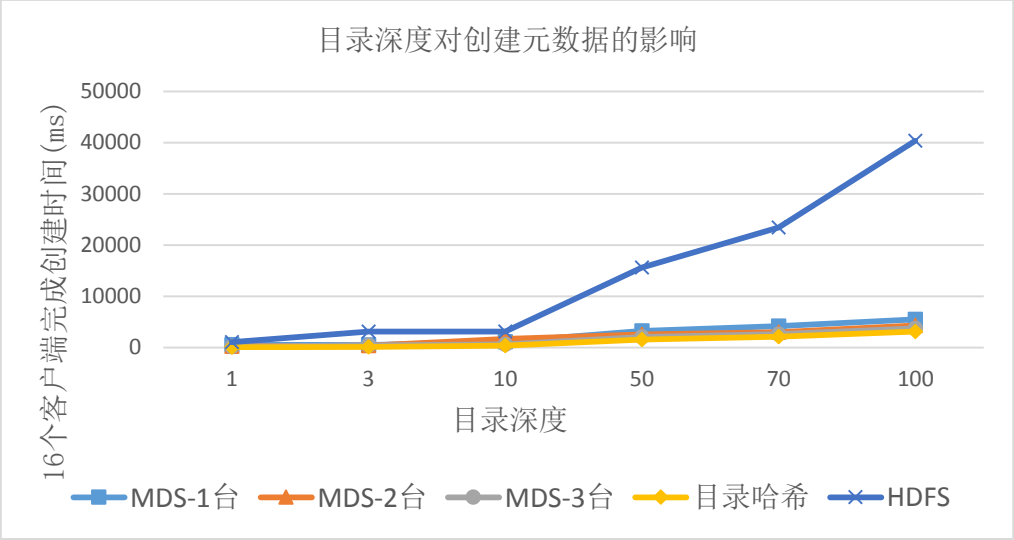

图 4-10　目录深度对创建时间的影响

从图 4-10 中可以看出，本文方法对目录深度并不敏感，从 1-100 层目录，1-3 台 MDS 花费的时间差不多，虽然都在增加，但增幅较小，总时间在 5500ms 以内。HDFS 在目录深度为 10 之后，性能大幅下降，执行时间大大增加。当目录深度为 100 层时，HDFS 完成创建需要 403s，目录哈希方法花费的时间在 3.1ms 以内，而本文方法使用 1-3 台 MDS 花费的时间都在 5.5s 内。这是因为目录哈希方法从原理上取消了目录的树形结构，创建目录不受深度的影响，而本文方法的目录索引结构采用了键值对的

形式，与目录哈希方法类似，对目录深度不敏感，也避免了 HDFS 中树形目录结构的深层遍历操作。因此，本文方案适合多层次目录的管理。

4.5 本章小结

本章主要测试和分析了本文方法性能和扩展性。对比了本文方法采用单一 MDS 和集中式（HDFS）管理方法的性能，元数据的创建、查找、遍历等操作性能比 HDFS 高出多倍。通过扩大 MDS 集群规模测试了系统的可扩展性并和分布式（基于目录哈希方法）管理方法对比，本文方法可以避免重命名造成的元数据迁移，而且元数据管理性能随集群扩展而提升。测试结果表明，本文方法具有较好的管理性能和扩展性，而且有较高元数据访问局部性，特别适合高并发情况下的元数据管理。

5 全文总结

5.1 工作总结

随着大数据时代的到来，全世界的数据规模正在飞速增长。大数据环境对存储系统的元数据管理提出了更高的要求，需要元数据管理具备高性能和可扩展性，本文提出的文件路径和属性信息分离的元数据组织方法，主要工作包括：

(1) 提出一种文件路径和属性信息分离的分布式元数据组织方法。本文提出的元数据组织方法能够应对大数据存储元数据管理面临的问题，具有较高的元数据管理性能和访问局部性；具备平滑的扩展性；能够避免重命名和集群扩大造成的元数据迁移。此外，方法具备均匀分布大目录下元数据的能力，并且可以不限制目录的大小。

(2) 设计并实现了元数据组织方法的关键技术。目录索引模块和 MDS 集群的底层分别采用了 RocksDB 数据库和 SSDB 数据库，使用 Zookeeper 在 MDS 上层构建节点的信息采集和监控工具，利用 RMI 技术实现集群通信和数据传输，使用 Java 实现本文方法，并对外提供元数据管理接口。

(3) 测试和分析了方法的性能和扩展性。通过与集中式元数据管理方法执行相同的操作，对比了本文方法采用了单一 MDS 的性能和访问局部性，通过增加 MDS 测试了方法的可扩展性，并和分布式元数据管理方法进行对比。测试结果证明本文方法具备管理大数据环境下元数据的能力。

5.2 工作展望

未来的工作可以从以下几点继续完善：

(1) 本文实现的方法提供了部分元数据管理接口，尚未与分布式文件系统实现兼容，未来的工作是通过实现 POSIX 规范和其他系统的元数据管理接口，实现位于系

统上层的分布式元数据管理系统。

(2) 本文实现的方法中目录索引模块和定位服务模块部署在一个服务器上，未来可以将它们部署到多台服务器上，并由 Zookeeper 集群负责各个服务器上数据的一致性。多个服务器同时提供服务，可以使得元数据的管理具有更高的性能和稳定性。

(3) 将本文提出的方法与内容分析相结合。利用本文方法文件路径和属性信息分离的特点，可以避免文件路径对内容分析结果产生的影响，从而提高内容分析的精度。

致 谢

在研究生即将结束之际，回顾过去，在此我向所有给予过我帮助和关心的老师和同学表示衷心的感谢。

感谢我的导师李春花老师，李老师不仅在学术上给予我认真悉心的指导，而且李老师在生活中给予我很多的帮助。李老师一方面忙于科研教学，另一方法还要营造实验室良好的学习和工作氛围，十分忙碌。但每当我遇到问题找她帮助，她都会抽出时间帮助我分析问题，培养我解决问题的能力。不仅如此，她严谨的治学态度和为人处世的方法也对我有着深深的影响。再次感谢李老师这几年对我的指导和关心，祝您身体健康，天天开心。

感谢周可老师和王桦老师，周老师不仅掌握实验室整体的研究方向，而且教导了我如何更好的做人、做事。王桦老师释放亲切，在开题的时候给了我很多帮助。

感谢已经毕业的罗芳、陈寨寨、黄绍建、万广平、刘鹏、金吉祥师兄师姐，在学习和生活上对我的帮助；感谢上一届边泽明、张彦哲、何爽、刘辉、廖正霜学长学姐，给与我们实习和找工作方面的帮助，一起相处两年的时间，从你们身上学到了很多，感谢同一届的沈慧羊、饶琦、陶灿、夏明、蒋丹同学，和你们一起度过三年时间是件很愉快的事情。感谢吴泽邦同学，在蘑菇街一起实习的日子非常难忘，感谢你的帮助。感谢图书馆的方吉老师，在您的帮助下完成了我第一个面向用户的项目。

最后，我要感谢我的家人，感谢你们这么多年对我的关心与支持，祝你们身体健康、幸福快乐。

参考文献

[1] IDC.https://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf

[2] C Lynch. Big data: How do your data grow?. Nature, 2008, 455(7209): 28~29

[3] KR Jackson, L Ramakrishnan, K Muriki, et al. Performance analysis of high performance computing applications on the amazon web services cloud. in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2010. 159~168

[4] E Ciurana. Developing with google app engine. Apress, 2009

[5] 罗达强. 探析 Windows Azure Platform 微软云计算平台. 硅谷, 2010(16): 9~10

[6] S Ghemawat, H Gobioff, ST Leung. The Google file system. in: ACM SIGOPS operating systems review. ACM, 2003. 29~43

[7] K Shvachko, H Kuang, S Radia, et al. The Hadoop Distributed File System. in: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). IEEE Computer Society, 2010. 1~10

[8] G Decandia, D Hastorun, M Jampani, etal. Dynamo: amazon's highly available key-value store. Acm Sigops Operating Systems Review, 2007, 41(6): 205~220

[9] 韩君易. NoSQL 数据库解决方案 Tair 浅析. 电子商务, 2011(9): 54~54

[10] F Chang, J Dean, S Ghemawat, et al. Bigtable: A Distributed Storage System for Structured Data. Acm Transactions on Computer Systems, 2008, 26(2): 205~218

[11] Mehul, Nalin, Vora. Hadoop-HBase for large-scale data. in: 2011 international conference on Computer science and network technology (ICCSNT). IEEE, 2011. 601~605

[12] D Roselli, JR Lorch, TE Anderson. A Comparison of File System Workloads. in:

Proceedings of the annual conference on USENIX Annual Technical Conference, 2002. 41~54

[13] S Dayal. Characterizing HEC storage systems at rest. Carnegie Mellon University PDL Technique Report CMU-PDL-08-109, 2008

[14] K Ren, Y Kwon, M Balazinska, et al. Hadoop's adolescence: an analysis of Hadoop usage in scientific workloads. Proceedings of the VLDB Endowment, 2013, 6(10): 853~864

[15] B Welch, G Noer. Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. in: Proceedings of 29th IEEE conference on massive data storage (MSST), 2013. 1~12

[16] JR Wernsing，G Stitt. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. in: ACM Sigplan/sigbed 2010 Conference on Languages, Compilers and TOOLS for Embedded Systems, 2010. 115~124

[17] MR Palankar, A Iamnitchi, M Ripeanu, et al. Amazon S3 for science grids: a viable solution?. in: Proceedings of the 2008 international workshop on Data-aware distributed computing, 2008. 55~64

[18] 徐鹏, 陈思, 苏森. 互联网应用 PaaS 平台体系结构. 北京邮电大学学报, 2012, 35(1): 120~124

[19] T Redkar. Windows Azure Storage Part I — Blobs. Windows Azure Platform. New York City: Apress, 2009. 205~266

[20] T Redkar, T Guidici. Windows Azure Storage Part III — Tables. Windows Azure Platform. New York City: Apress, 2011. 247~306

[21] M Burrows. The Chubby lock service for loosely-coupled distributed systems. in: Proceedings of the 7th symposium on Operating systems design and

implementation. USENIX Association, 2006. 335~350

[22] D Karger, E Lehman, T Leighton, et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. in: Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6 1997.654~663

[23] A Davies, A Orsaria. Scale out with GlusterFS. Linux Journal, 2013, 2013(235): 1

[24] SA Weil, KT Pollack, SA Brandt, et al. Dynamic metadata management for petabyte-scale file systems. in: Supercomputing, Acm/ieee Sc Conference, 2004. 4~15

[25] M Satyanarayanan, JJ Kistler, P Kumar, et al. Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on Computers, 1990, 39(4): 447~459

[26] P Kumar, M Satyanarayanan. Log-based directory resolution in the coda file system. in: International Conference on Parallel & Distributed Information Systems. IEEE Computer Society Press, 1993. 202~213

[27] M Satyanarayanan, MH Conner, JH Howard, et al. Andrew: A distributed personal computing environment. Communications of the ACM, 1986, 29(3): 184~201

[28] SA Weil, SA Brandt, EL Miller, et al. Ceph: A scalable, high-performance distributed file system. in: Symposium on Operating Systems Design & Implementation, 2006. 307~320

[29] SA Brandt, EL Miller, DDE Long, et al. Efficient metadata management in large distributed storage systems. in: IEEE/Nasa Goddard Conference on Mass Storage System & Technologies, 2003. 290~298

[30] PF Corbett, DG Feitelson. The Vesta parallel file system. Acm Transactions on Computer Systems, 2001, 14(3): 1~16

[31] PJ Braam, M Callahan, P Schwan, et al. The intermezzo file system. in: Proceedings of the 3rd of the Perl Conference, 1999. 32~41

[32] K Ren, Q Zheng, S Patil, et al. IndexFS: scaling file system metadata performance with stateless caching and bulk insertion. in: International Conference for High Performance Computing, Networking, Storage and Analysis, 2014. 237~248

[33] PH Carns, WB Ligon Iii, RB Ros, et al. PVFS: a parallel file system for linux clusters. in: Proceedings of the 4th annual Linux Showcase & Conference - Volume 4. USENIX Association, 2000. 391~430

[34] S Patil, G Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. in: Proceedings of the 9th USENIX conference on File and stroage technologies. USENIX Association, 2011. 177~190

[35] 周江, 王伟平, 孟丹, 等. 面向大数据分析的分布式文件系统关键技术. 计算机研究与发展, 2014, 51(2): 382~394

[36] 肖中正, 陈宁江, 魏峻, 等. 一种面向海量存储系统的高效元数据集群管理方案. 计算机研究与发展, 2015, 52(4): 929~942

[37] V Meshram, X Besseron, X Ouyang, et al. Can a Decentralized Metadata Service Layer benefit Parallel Filesystems? in: Proceedings of the 2011 IEEE International Conference on Cluster Computing. IEEE Computer Society, 2011. 484~493

[38] P Hunt, M Konar, FP Junqueira, et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems. in: USENIX Annual Technical Conference, 2010. 653~710

[39] FP Junqueira, BC Reed. The life and times of a zookeeper. in: Proceedings of the 28th ACM symposium on Principles of distributed computing. ACM, 2009. 4~4

[40] FP Junqueira, BC Reed, M Serafini. Zab: High-performance broadcast for primary-backup systems. in: IEEE/IFIP International Conference on Dependable Systems & Networks, 2011. 245~256

[41] 冯幼乐，朱六璋．CEPH 动态元数据管理方法分析与改进．电子技术，2010, 47(9)

[42] Eckel Bruce. Java 编程思想.(第 4 版）．陈昊鹏译．北京：机械工业出版社, 2002

[43] J Maassen, R Van Nieuwpoort, R Veldema. et al. in: An Efficient Implementation of Java's Remote Method Invocation. ACM SIGPLAN Notices, 2000. 173~182

[44] Facebook Backs RocksDB Open Database in Flash. Green Data Centers & Internet Business Newsletter, 2013, 11(11): 2~2

[45] 罗军，陈席林，李文生．高效 Key-Value 持久化缓存系统的实现．计算机工程, 2014, 40(3): 33~38

附录 攻读学位期间申请的专利

[1] 李春花，周可，杨勇。一种面向大数据环境的元数据组织方法和系统，中国发明专利；申请号：201610056156.0