分类号_____ 学号 M201372698

学校代码 10487 密级_____

华中科技大学

硕士学位论文

Ceph 文件系统元数据访问性能优化研究

学位申请人：葛凯凯

学科专业 ：计算机系统结构

指导教师 ：谭志虎 副教授

答辩日期 ：2016.5

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering

The Research and Optimization of performance of metadata access based on Ceph file system

Candidate : Ge Kaikai

Major : Computer Architecture

Supervisor : Assoc. Prof. Tan Zhihu

Huazhong University of Science and Technology

Wuhan, Hubei 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。

保密□，在_____年解密后适用本授权书。

本论文属于 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

分布式系统以其高性能、高可靠性和高扩展性的优势，逐渐成为存储海量数据的首选。在分布式文件系统中，数据主要分为文件数据信息和元数据信息，其中元数据只占数据总量的 10%左右，但元数据的访问量却占系统总访问量的 60%以上，因此减少元数据的访问延迟对提高系统的整体性能至关重要。

通过研究开源分布式文件系统 Ceph 中元数据访问的关键技术，分别从客户端和元数据服务器端给出优化方案。客户端优化主要根据缓存中目录项元数据的访问频率，利用目录分片预取出访问频率高的目录区域，将更多的热元数据缓存在客户端，提高元数据命中率；对于客户端的文件路径的查找，采用连续两级目录项未命中优化策略，减少客户端与元数据服务器之间的网络交互次数。元数据服务器端优化主要是优化元数据的缓存管理，对元数据进行主副本分类，并根据元数据命中和更新倒盘特点设计固定比例老化策略，并将干净元数据与脏元数据分开管理，尽量保存优先级高的元数据，减少淘汰开销；另外，针对元数据请求处理流程，通过任务细分优化策略，将请求处理任务划分为三个阶段，并使用流水线型模式以加快元数据处理速度。

最后，利用测试工具对优化前后的系统进行对比测试。测试结果表明，增加预取模块、路径查找优化、元数据缓存管理优化和元数据操作任务细分性能分别提升 9.1%、16.6%、24.3%和 19.6%。测试数据表明元数据服务器端的优化更能提高系统的性能。

关键字：分布式文件系统，元数据，预取，缓存，日志

Abstract

At the era of information, the distributed system has become the best choice for mass data storage with the advantage of high performance、reliability and scalability. In the distributed file system, the data is mainly classified as file data and metadata. Although metadata only accounts for about 10% of the total amount of data, the access to metadata accounts for more than 60% of the total. Reducing access latency of metadata is of great significance to improve the overall performance of the system.

Through the analysis of key technologies of metadata access in Ceph distributed file system, two levels of cache optimization scheme are proposed respectively from the client and the metadata server. For client, according to the access frequency of cached directory entry, high frequency areas of the directory can be prefetched from metadata servers by the directory fragments technology, so that more popular metadata will be cached in the client, which improves hit rate of the metadata; for the client's file path search, the optimization strategy of continuous two level directory entries miss is be used, which can reduce the number of network interaction between the client and the metadata server. For metadata server, metadata is be grouped according to the properties of master-replica. Besides, the aging strategy of a fixed proportion and the separate management of clean and dirty metadata are put forward, which can effectively reserve the most active metadata. In addition, task subdivision are proposed to the metadata request. The tasks are divided into three stages, and Pipeline technology speeds up the metadata process.

Finally, the test tool is used for contrast tests on the performance of the system before and after optimization. The test results show that, the performance improvements of prefetch metadata scheme are up to 9.1%, the performance improvements of path search optimization scheme are up to 16.6%, the performance improvements of metadata cache algorithm scheme are up to 24.3% and the performance improvements of log isolation optimization scheme are up to 19.6%. According to the test results, the optimization of metadata servers can effectively enhance the system performance.

Keywords: The distributed file system, Ceph, metadata, prefetch, Cache, Log

目 录

1 绪论

本章为绪论部分，主要分为课题背景，以引出分布式文件系统中元数据研究的重要意义，再讨论当前国内外对分布式文件系统元数据管理研究和优化的现状，最后阐述了本文主要的研究工作和内容组织结构。

1.1 课题背景

进入 21 世纪，随着互联网的到来，尤其是移动互联网、社交网络、电子商务的飞速发展，人类在生产和生活中产生的数据呈现指数型地增长，数据量已经从 TB 数量级上升到 PB 数据级、EB 数量级甚至 ZB 数量级，人类已经进入了大数据时代。

据 IDC 的报告称，2015 年全球数据总量 8.6ZB，目前全球数据的增长速度为每年 40%左右，预计 2016 年的数据量在 12ZB 左右，到 2020 年的时候，全球的数据总量将达到 40ZB。2015，阿里巴巴的淘宝网每天产生 7TB 以上的用户数据信息；腾讯科技公司的 QQ 和微信，每天有 5 亿多的用户，每天能产生 100TB 的用户统计数据信息。

数据与日俱增，需要存储的容量越来越大，单节点和磁盘阵列这些传统的存储技术已经很难满足海量数据存储的需求，集群存储系统以其天然的可扩展性优势被得到广泛的运用，其中包括分布式文件系统。分布式文件系统也被称为网络文件系统，它可以根据每台服务器和客户端的访问列表和容量严格地访问文件系统，通过把数据和元数据通过一定的协议和规则分散到各个服务器，并在客户端提供文件系统的标准接口。

分布式文件系统相对于本地文件系统而言，原理上比较相似，重点是增加了容量的可扩展性。在分布式文件系统中，数据同样可以分为文件数据和元数据两种，其中元数据是相对比较小的，一般一个文件元数据的大小在 100 字节之内，而且系统中元数据总量占系统总容量的比例不到 10%，但是元数据的访问量在文件系统中占到访问总量的 50%~80%[1]，无论是打开文件、创建文件还是删除文件这些操作都需要首先对元数据进行处理。所以，在文件系统中，元数据操作对整个文件系统的性能有重要影响。因而研究如何提高元数据访问操作性能是具有研究价值和意义的。

为了提高元数据访问操作的性能，可以通过元数据预取、元数据缓存的优化来提高元数据的命中率，减少元数据访问的延迟；可以通过元数据服务器日志提交操作的功能细化来加快元数据操作的处理速度，从而提升系统的整体性能。

1.2 国内外研究现状

随着科技的发展以及数据的暴增，文件存储技术也在不断地发展以满足人们日益提升的需求，从直接附属存储 DAS[2]，发展到存储局域网 SAN[3][4]，再发展到网络附属存储 NAS[5]，直至现在的分布式文件系统，日新月异的存储技术正发挥着越来越大的作用。分布式文件系统在管理和存储海量数据中发挥着不可替代的作用，为大数据[6]、云计算[7]、数据挖掘[8]。提供存储支持，也是作为云存储服务的后端存储平台之一，能够使得数据的价值被不断地发掘出来。

分布式文件系统发展至今，常见的有 GFS[9]、HDFS[10]、Lustre[11]、Ceph[12]、GridFS[13]、TidyFS [14]、TFS[15]、PVFS[16]等，这些文件系统提供的都是应用级的文件存储服务，同时它们各自适用于不同的领域，根据自身的优点发挥出它们最大的优势。当前分布式文件系统的研究主要分为文件数据和元数据两个方面，而元数据的访问请求特征表明，加快元数据请求操作能更有效的提升系统的整体性能，于是如何提高元数据访问请求操作成了研究的热点。

Bing Zhang 等人提出一个用于云主机元数据缓存、检索、预取的系统叫目录列表服务(DLS)，主要是通过动态配置并行 TCP 流和非阻塞并发内存缓存，以此达到减少元数据访问延迟，增加元数据访问操作，并且这种设计可以通过本地和广域的设置来评估系统的性能[17]。Juan Piernas 等人在他们的 FPFS(Fusion Parallel File Systems)系统上实现 batchops，就是在单个请求中增加元数据的请求速率来发送多个请求到服务器，这样可以提高元数据的请求处理的速度，减少请求操作服务器之间的网络延迟，减少消息的数量，他们通过实验表明元数据性能提高了 23%~100%[18]。Quan Zhang 等人提出了一种基于委托的元数据服务机制(DMS)来减少元数据的访问延迟和优化小文件的性能，同时实现了四种技术来保 DMS 的一致性和效率：预先分配串行 metahandlers、基于目录的元数据替换、包装事务操作和细粒度锁撤销，并且他们在并行分布式文件系统进行了实现，结果表明元数据操作和小文件访问的性能都得到了显著的改善，这个方案将有可能被广泛的应用到其他的分布式文件系统中

[19]。Rajesh Vellore Arumugam 等人提出了一种在 EB 规模的文件系统元数据服务器集群的自适应和可扩展性的负载均衡方案—云缓存，这个方案结合自适应高速缓存扩散和复制方案，以应付该请求负载平衡问题，并且可以被集成到现有的分布式元数据管理方法中，可以有效的改善他们的负载平衡性能[20]。Garth A. Gibson 等人探索通过显示地复制所有服务器中的目录查找状态信息替换掉所有客户端缓存信息，同时减少为了解决不同层次权限检查到不同服务器的重复 RPC 请求，他们实现复制目录查找状态，ShardFS，并且采用了一种新的乐悲观混合并发控制来使单个对象事务通过分布式事务进行处理，实验结果表明能提供低于原来 70%的响应时间[21]。Daniel J. Abadi 等人提出一种用高通量的分布式数据系统来管理元数据，实验表明此方案能改进文件系统元数据的可扩展性，因为元数据可以被分片进行存储，同时元数据的操作可以被转化为分布式的事务[22]。介于 hash 和子树分割分别对于元数据的分布式管理的不足，Sridhar Mahadevan 等人提出了一种基于环的元数据管理机-DROP(Dyamic Ring Online Partitioning),它可以使用本地 hash 算法把元数据保存在本地，从而保证元数据的一致性，同时通过动态的分布元数据保证元数据在服务器集群间的负载均衡，他们通过跟踪驱动模拟和原型实现进行了性能评估，实现表明性能和可靠性都得到了显著的提升[23]。Chul Lee 等人设计并实现了一种元数据推迟写入技术，以实现高效可靠的 NAND[24]闪存文件系统方案，该方案是合并元数据的写入，以便减少 NAND 闪存的垃圾，同时确保文件的一致性，所提出的方案使用非易失性存储器用于同步记录元数据的修改，日志记录可以显著降低 NAND 闪存的过度元数据写入，此外，最后一次更新元数据可以从崩溃中恢复，实验结果大大降低了整体应用的时间和不同基准写入页面的数量[25]。刘国良等人提出了一种目录元数据更新策略，将元数据分割成块，通过网络只传输修改的块，以此来减少网络的传输时间，他们使用 fuse 用户态文件系统实现了一个概念原型来证明此方法的有效性，结果表明，含有大小为 5MB 目录项的目录文件，可减少约 20 倍对目录进行小更新的操作时间[26]。

1.3 主要工作和论文组织结构

本文的主要工作是对元数据访问性能优化研究，通过元数据预取、路径查询优化、缓存算法改进和请求处理细化等方法来提升系统的性能，主要研究的内容如下：

1. 研究分析文件访问的特性，利用系统的目录分片实现客户端元数据的预取，并且根据热元数据对象的不同实现不同的预取策略，以此在客户端缓存尽量多的热元数据，从而提高元数据在客户端的访问命中率，减少到元数据服务器查找元数据的操作时间。

2. 研究分析文件路径查找的特性，对于连续目录项未命中需要多次网络传输访问的情景，提出连续两级目录项未命中策略进行优化，减少与元数据服务器之间网络访问次数。

3. 研究分析元数据服务器分布式缓存中属主元数据和副本元数据的特性，对元数据服务器中缓存的元数据进行分组管理，并根据元数据命中和更新倒盘特点设计固定比例老化策略和干净元数据与脏元数据分开管理，从而在元数据服务器中提高元数据的命中率，尽量保存热度高、近期刚访问过的元数据。

4. 研究元数据服务器中元数据请求操作的流程，分析日志在元数据修改操作中所占的访问延时，通过把元数据操作流程分解为三个阶段：元数据修改阶段、日志写缓存阶段和日志刷新到 OSD 阶段。对这三个阶段分别使用独立服务进行管理，并使用流水线型模式提高元数据服务器处理元数据请求的速度。

本论文的内容组织如下：

第一章主要介绍本论文的研究背景，国内外的研究现状以及主要的工作内容和论文的组织结构。

第二章简单地介绍一下几种分流行的布式文件系统，并重点介绍 Ceph 分布式文件系统的架构，同时还介绍了 Ceph 用户态客户端使用的 FUSE 模块以及日志文件系统。

第三章是基于 Ceph 元数据访问操作优化的设计，主要分为客户端和元数据服务器两个元数据缓存层次，客户端主要是元数据预取和路径查找的优化，元数据服务器主要是优化缓存算法和请求处理细分。

第四章是基于 Ceph 元数据访问操作优化的设计，主要展示关键的数据结构和数据流程。

第五章介绍本论文的测试环境，利用测试工具对优化后的系统进行测试和性能分析，并得出相关结论。

第六章是全文总结，总结本论文的工作，分析论文的不足之处，并对后期工作进行展望，做出下一步工作的计划。

2 相关技术研究

本章首先介绍几种流行的分布式文件系统，介绍一下它们各自的优点和应用场景，然后重点介绍 Ceph 的生态环境和系统的架构，并介绍了一下 Ceph 用户态客户端用到的 FUSE 模块，最后介绍了缓存算法和日志文件系统的基本知识。

2.1 分布式文件系统

大数据和云存储的发展推动下，分布式文件系统成为了一个新的研究热点。分布式文件系统是由很多计算机通过网络互连形成一个整体，把整个文件系统的命名空间划分为多个部分由不同的计算机进行管理以便于扩展，并向用户提供文件存取的服务，其中每一个计算机就是分布式文件系统的一个节点。

分布式文件系统中的数据主要分为文件数据和元数据，文件数据是指文件的内容。而元数据一般称之为数据的数据，同时根据对象的不同又可分为目录元数据和文件元数据两类，其中目录元数据信息包括目录的名字、目录最近修改的时间、所属者、访问控制权限等。文件元数据信息包括文件名、创建时间、修改时间、最近访问时间、文件大小、文件用户等。文件数据和元数据大小范围存在着很大的差别。据估计，普通文件的大小一般在 22KB，而元数据的大小一般在 1.37KB，可见元数据总量不到系统空间的 10%，但是由于文件的每一次数据访问操作都需要先访问元数据，因此其访问量却占 50%~80%。

分布式文件系统[27]可以根据不同的指标划分为多种类型，指标主要有命名空间划分方法和元数据管理模型两种。

常用的命名空间划分方法有基于 hash 计算方法[28]和子树划分方法，其中子树划分方法又有动态子树划分和静态子树[29]划分，具体如下：

1. 静态子树划分是一种最常用的命名空间划分方法，通过静态地划分目录层次，并且需要管理员手动配置把每一个子树分割给特定的服务器,一旦设置好，以后一般很少进行修改命名空间，缺少灵活性并且不能很好地处理负载均衡问题。动态子树划分正好弥补了静态子树分割的不足，能根据集群服务器当前的负载情况自动重新划分命名空间进行负载调整，但是在设计实现方面要比静态划分复杂很多。

2. 基于 hash 计算的划分通过对文件路径名或者是一些文件的独特标识符进行 hash 映射到不同的节点，通过这个方法可以能很快地定位到信息所在的节点位置，同时由于对局部性要求不高可以很好的平衡数据负载，但是正因如此可扩展性不是很好，一旦有节点加入或者删除需要对整个命名空间进行重新 hash 计算。

而元数据管理模型主要分为三类：分布式的元数据管理模型、集中式的元数据管理模型和无元数据的管理模型，具体如下：

1. 集中式元数据管理模式，是指文件系统中只有一个元数据服务器，称之为中央元数据服务器，其负责客户端的查询请求和元数据的存储，并由它统一管理文件系统的命名空间，提供命名解析和数据定位的功能。由于使用单节点，所以往往会引发单节点故障的瓶颈，为了解决这个问题，通常会配置一个备用的元数据服务器，这种主/从备份的策略使系统能在主元数据服务宕机时，让从元数据服务器接管主元数据服务器的资源和工作。

2. 分布式元数据管理模式，是指系统使用分布式集群来管理元数据以避免集中式元数据服务模型中的性能瓶颈和单点故障问题。而这种模式又可以细分为两类，一类称之为全对等模式，集群中的每个元数据服务器都是完全对等的，每个都可以独立对外提供元数据服务，然后集群内部通过网络通信达到元数据同步，保持数据的一致性，比如 LoongStore；另一类称之为全分布模式，集群中的每一个元数据服务器负责管理部分元数据信息，这样共同构成完整的元数据服务，比如 GPFS[30]和 Ceph。

3. 无元数据管理模式，是指文件系统中不使用元数据服务器来定位查找元数据信息，其中典型的代表是 Glusterfs 文件系统，它使用一种弹性 hash 算法来查询元数据，从而没有元数据服务器的单点故障和性能瓶颈问题。

当前很多企业和研究者纷纷投入分布式文件系统的行业，竞相推出自己的分布式文件系统，近些年来，已经出现许多高性能的分布式文件系统，其中比较典型的有 HDFS、Lustre、GlusterFS[31]和 Ceph，下面简单的介绍一下这几种分布式文件系统。

2.1.1 HDFS

HDFS(Hadoop Distributed File System)是 Apache Hadoop Core 项目的一部分，最

开始是作为 Apache Nutch 搜索引擎项目的基础架构而开发。HDFS 是 Hadoop 的文件系统组件，它主要被用存储大文件。HDFS 属于集中式的元数据服务模型，因为它使用的是单节点元数据服务器，所以没有所谓的元数据分布算法。HDFS 不需要专门的服务器，能充分利用廉价的 PC 机搭建起一个高吞吐量、高容错的集群。HDFS 是一个典型的主/从架构，它的生态环境包括一个元数据服务器(NameNode)和多个数据服务器(DataNode), NameNode 管理整个文件系统的命名空间，就是文件和目录的层次结构，DataNode 用于存储整个文件系统的文件数据，其系统架构如图 2.1 所示。

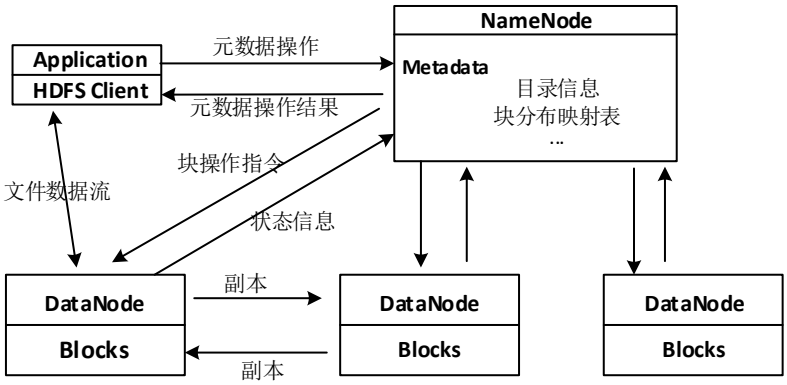


图 2.1 HDFS 系统架构图

HDFS 中元数据存储在 NameNode 上，文件数据存储在 DataNode 上，通过这种分别存储来分离元数据和数据的访问通道，从而可以根据元数据和文件数据各自的数据特征和访问特性，分别采用不同算法对它们的访问进行优化，不仅如此，还增加了各自管理的灵活性，从而提高了系统整体的可扩展性。但是由于 HDFS 只有一个 NameNode，单节点的元数据服务器会成为整个系统的性能瓶颈，一些公司纷纷提出一些改进策略，像 FaceBook 的 Avatarnode[32]方案。

2.1.2 Lustre

Lustre 是一个开源的并行文件系统，它主要用于高性能计算和世界范围内的企业环境需求。Lustre 是由 Cluster File System 公司联合一些其他公司开发出来主要用于解决海量数据存储的文件系统。Lustre 提供 POSIX 兼容接口，其客户端可以扩展到数千个，并且能达到 PB 级的存储容量。Lustre 也属于集中式的元数据管理模型，也同时分开管理文件数据和元数据的访问通道，不仅如此它也是第一个基于对象存储的文件系统。对象存储主要是指把数据划分为一个个对象进行管理，每个对象包括

数据本身、一些可变的元数据信息和一个全局的标识符，对象主要用于存储非结构化的数据。对象存储可以通过使用小的对象表代替大的块表简化数据的分布。

Lustre 的生态环境由客户端，两个元数据服务器(MDS)，多个对象存储节点(OSD)组成，其系统架构如图 2.2 所示。

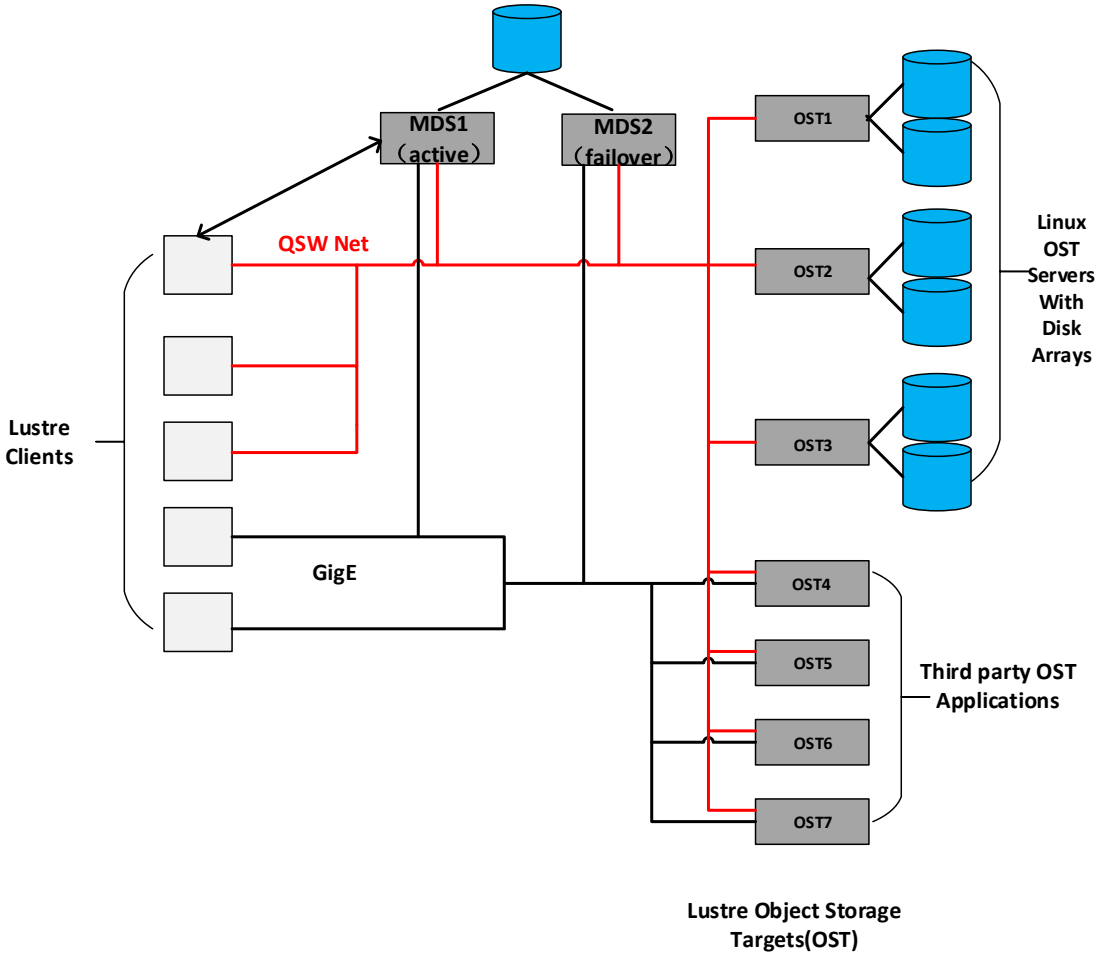


图 2.2 Lustre 系统架构图

Lustre 使用对象存储技术来存储文件数据，它把每一个文件分成若干个对象，在通过算法分配到各个存储节点上进行存储，而且可以对文件大小比较大的 I/O 操作进行分解，让 I/O 操作分发到不同的节点上进行并发处理。实践证明，对象存储技术使得 Lustre 在大文件存储性能上有着优越的性能，不仅如此，Lustre 还借用了传统文件系统的共享存储设计思想，使得它更加智能，数据管理更加高效，系统的部署也更加简单。虽然 Lustre 有如此多的优点，但是它对于小文件的存储性能不是很理想。

2.1.3 GlusterFS

GlusterFS 是自由软件，主要由 Z RESEARCH 负责开发。其主要应用于集群系统中，具有良好的可扩展性，同时它的软件结构设计良好，易于扩展和配置，可以通过各个模块之间的灵活搭配以得到比较有针对性的解决方案。GlusterFS 主要用于解决以下问题：联合存储(联合多个节点上的存储空间)、网络存储、冗余备份和大文件的负载均衡。GlusterFS 主要应用于云计算、流媒体服务和内容分发网络，同时是无元数据服务模型的典型代表。不使用元数据服务器使得 GlusterFS 能获得接近线性的高扩展性。与前面介绍 HDFS 和 Gluster 不同，Gluster 无需分开管理和存储数据和元数据，数据的查询仅需通过 hash 计算出文件名和路径的映射值。GlusterFS 系统框架图如 2.3 所示：

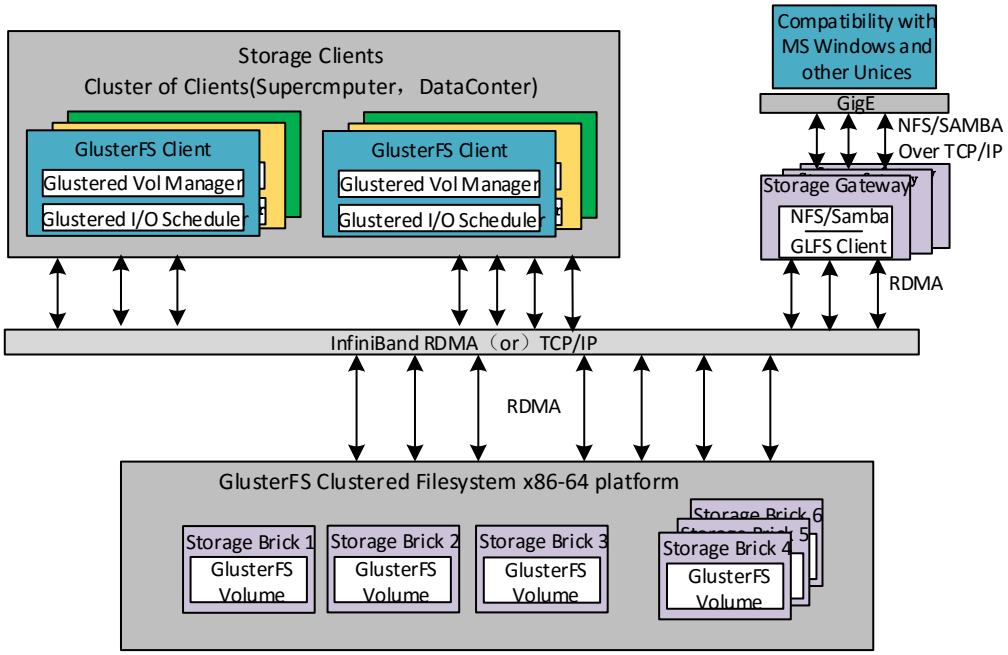


图 2.3 GlusterFS 系统架构图

GlusterFS 由于没有元数据服务，所以系统复杂度大大降低，系统扩展很容易，并且是在用户层实现，容易部署和维护，没有了元数据瓶颈的限制，并发性能有很大的优势，比较适合大数据量的离线应用，但是正是由于没有元数据服务，其只能以文件作为存储对象，商业价值不是很大。

2.1.4 Ceph

Ceph 是由加州大学 Santa Cruz 分校存储系统研究中心开发的可支持 PB 级存储的开源高性能的分布式系统，它能同时提供对象存储、块存储和文件存储服务，即能提供统一存储，是现在软件定义存储(SDS)的明星项目。Ceph 的生态系统是由客户端(client)，元数据服务器(MDS)，对象存储设备(OSD),集群监控器(Monitor)，其系统架构如图 2.4 所示。

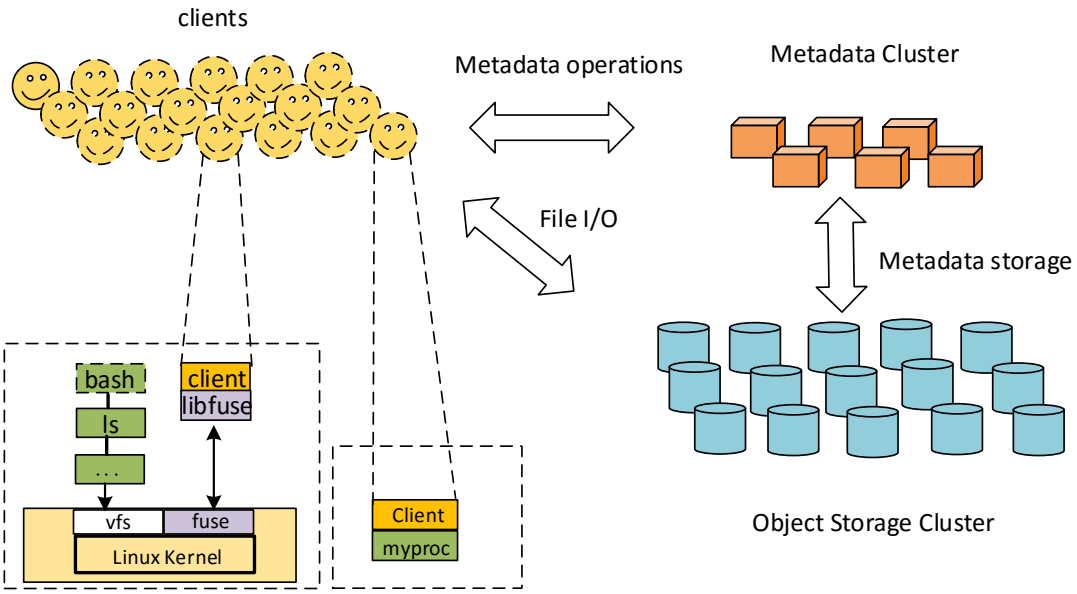


图 2.4 Ceph 系统架构图

在 Ceph 的生态系统，如果要提供分布式文件系统服务，至少需要使用一个 MDS 来缓存 Ceph 文件系统的元数据，Ceph 文件系统的数据和元数据都存储在 OSD 上，其中元数据存储在 OSD 上主要是能提供元数据的共享存储能力，MDS 集群主要是 OSD 上元数据的分布式缓存。OSD 和 Monitor 组合起来形成 RADOS(resilient automatic distributed object storage)对象存储集群，在这个基础之上向用户提供对象存储[33]、块存储和文件存储三种服务。

Ceph 是基于 POSIX 的分布式文件系统，由于使用的是元数据服务器集群，所以没有单节点故障问题，并且具有良好的容错能力，可以把元数据请求并行分发到各自的元数据服务器进行处理。除此之外，Ceph 使用动态子树分区把系统的命名空间划分成多个子树，分布到各个元数据服务器上分别进行管理，并且每个元数据服务

器会进行负载的统计，在负载不平衡时，通过把高负载元数据服务器上的一部分目录树迁移到低负载元数据服务器以达到全局的负载平衡。不仅如此，Ceph 还引入了 Lustre 的对象存储思想，对象作为整个系统存储基本单元，可以显著的提高系统的读写性能。CRUSH(Controlled Replication Under Scalable Hashing)伪随机数据分布算法是 Ceph 系统中一个最具创新的特点，CRUSH 摒弃了一般文件系统的查表操作来定位文件的数据，只需要根据一些参数通过一些计算就可以定位到文件的数据，显著提升了系统的性能。虽然 Ceph 有这么多的优点，但是由于代码是使用 C++编写，同时广泛使用 STL，整个系统的代码质量不是很高，而且文件 IO 路径调用过于复杂。

2.2 Fuse 文件系统

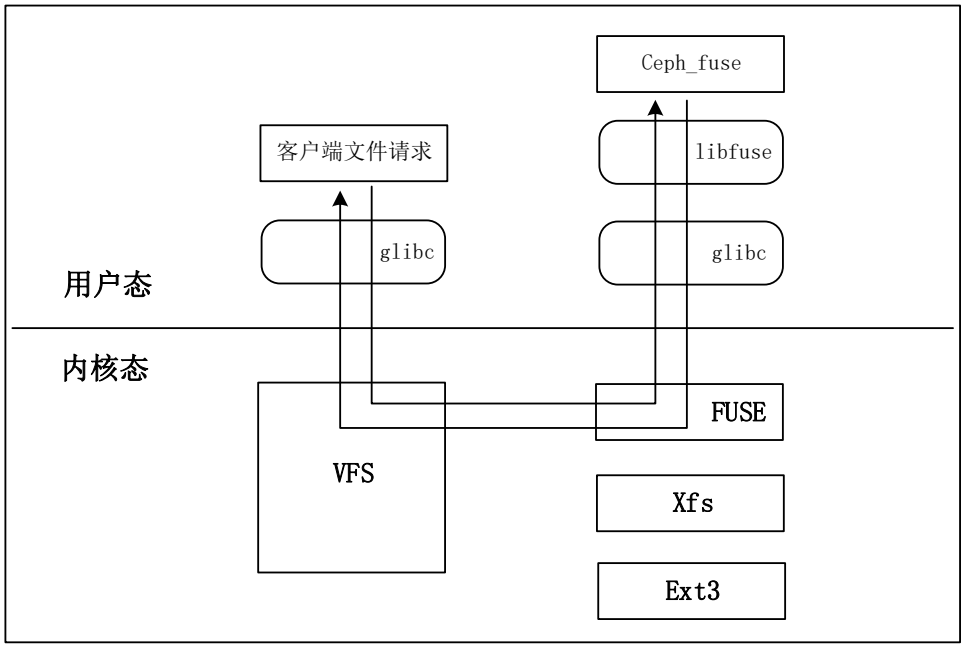


图 2.5 FUSE 工作流程

FUSE 是 Filesystem in Userspace 的简称，是类 Unix 操作系统上的一个软件接口，通过这个接口可以让非特权级用户在不重新编译内核的情况下创建自己的文件系统，即用户态的文件系统。而需要修改重新编译内核构建的文件系统便是内核态的。Ceph 分布式文件系统客户端的也有内核态和用户态两种版本。内核态的客户端是在 Linux2.6.34 之后版本中加入的，使用 mount 命令把 CephFS 挂载在某一个目录下；用户态的客户端需要使用 ceph-fuse 命令进行挂载，由于用户态的代码改写比较简单，

不像内核态需要修改并编译内核代码，所以用户态的客户端比较常用。Linux 上文件系统用户态主要是借助于 FUSE[34]模块，而 Windows 上主要借助于 dokan[35]模块，这里主要研究 FUSE 模块，FUSE 的工作流程如图 2.5 所示。

工作流程大致如下：

1. 首先应用程序的文件请求通过调用库函数提供的 API，比如 read，到达系统调用层，例如 sys_read。

2. 在请求到达系统调用层之后，经由虚拟文件系统层(VFS[36])传递到内核的 FUSE 模块。

3. 用户态的 FUSE 库会与内核态的 FUSE 模块进行通信，请求便由用户态的 libfuse 库传递给 CephFS 的用户态客户端代码，然后通过 Ceph 集群对请求进行处理。

4. 在完成请求之后，把请求的响应通过原路返回给应用程序。

2.3 缓存算法

缓存算法主要是根据历史访问记录选出访问比较频繁的数据，并让其存放在响应延时比较少的介质上，比如内存或者 SSD。总体来说，可以按访问时间、访问频率以及访问时间和访问频率兼顾策略分为三类。

1. 基于访问时间的策略，对缓存的数据按访问时间的先后进行管理，保留最近访问过的数据，淘汰访问时间最久的，如 LRU。

2. 基于访问频率的策略，对缓存的数据按访问频率的多少进行管理，优先保留访问频率高的数据，淘汰访问频率低的数据，比较常用的有 LFU、LRU-2[37]、2Q[38]、LIRS[39]。

3. 基于访问时间和访问频率兼容的策略，是上诉两种策略的一个折中，比较常用的有 FBR[40]、LRFU[41]。

2.4 日志文件系统

日志文件系统是一种可以自动故障恢复的文件系统，不需要使用 fsck 等命令来检测数据的一致性。日志文件系统通过在原磁盘块数据更新前把元数据的修改更新写到一个序列化的日志中来保证数据的完整性。在系统出错的情况下，系统可以通

过回放日志中记录的数据来恢复系统。常用的方法是在日志中记录文件的元数据信息，当要修改一个文件的数据时，需要先将新的元数据信息记录到日志文件中，直到元数据写日志完成后才可以把修改的数据写入到磁盘。Linux 系统常见的日志文件系统主要有 EXT3、ReiserFS、XFS[42]和 JFS[43]。下面主要介绍一下 EXT3 和 XFS 两种。

EXT3 是 Linux 系统上默认的文件系统，在 EXT2 的基础增加了两个独立的模块：事务模块和日志模块，并且能完全兼容 EXT2。EXT3 比较灵活可以为两种对象做日志，分别为元数据和文件数据，不仅如此 EXT3 还提供三种日志模式，主要如下：

1. 日志(Journal)，这种模式把所有数据和元数据的改变都记录到日志中。

2. 写回(Writeback)，这种模式日志中只记录元数据的改变。

3. 预定(Ordered)，这种模式跟写回模式差不多也是只记录元数据的修改，但是会把元数据和与其相关的数据块进行分组，以便先写数据块再写元数据。

XFS 是 SGI 开发，在 Ceph 生产环境下作为存储集群的本地文件系统使用。XFS 中的操作模式是使用异步写日志，它通过写前日志(writeahead)保证修改的数据提交到日志之后才能把其刷新到磁盘。XFS 通过两种方式保证异步写日志：

1. 多个更新操作批量放入单个日志写操作中，相对于底层的磁盘阵列这能有效的提高写日志的效率。

2. 元数据更新的性能可以独立于底层驱动的速度。这种独立受限于分配给日志的缓存 buffer 大小，但是这比传统文件系统的同步更新好很多。

2.5 本章小结

本章首先描述了一下分布式文件系统的基本知识，然后简单的介绍了几中当前比较流行的文件系统：HDFS、Lustre、GlusterFS 和 Ceph，以及 Ceph 用户态客户端使用的 FUSE 模块，最后简单的介绍了几种缓存算法。

3 基于 Ceph 系统元数据访问技术的优化设计

本章为基于 Ceph 文件系统元数据访问技术的改进设计。首先描述了 Ceph 系统中元数据访问流程，引出元数据缓存的两层结构，然后分别对两层缓存的相关技术提出改进。

3.1 优化分析

Ceph 系统分离了元数据和数据的 I/O，其中数据的操作主要是对文件数据的读写，元数据操作主要包含目录的创建、删除，文件的创建、删除，文件属性的设置等。Ceph 中元数据主要包含三种：目录信息(Dir)、目录项信息(Dentry)、索引节点信息(Inode)，三者之间的关系如图 3.1 所示。

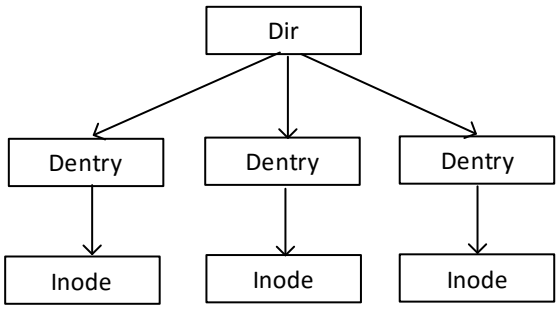


图 3.1 元数据关系图

目录 Dir 通常包含多个目录项 Dentry，每个目录项对应目录下的一个子项目，这个子项目可以是文件或者子目录，并且这些子项目都对应一个 Inode，如果是子目录，Inode 还会对应一个 Dir。

Ceph 文件系统生态环境包括 Client、MDS 以及 Monitor 和 OSD 组成的 RADOS。元数据和数据都存储在 OSD 上，MDS 集群只是缓存部分元数据。元数据的请求操作首先通过 FUSE 到达客户端，客户端会根据元数据缓存查看有没有命中，命中则根据找到的文件元数据进行相应的操作并响应应用请求；如果没有命中则客户端根据之前的访问记录把请求发送到其管理的 MDS 进行查找，MDS 如果命中则把元数据返回给客户端，并缓存在 Cache 中，没用命中则到 OSD 中进行查找，同时取回并缓存到 MDS 中。系统缓存架构图如 3.2 所示。

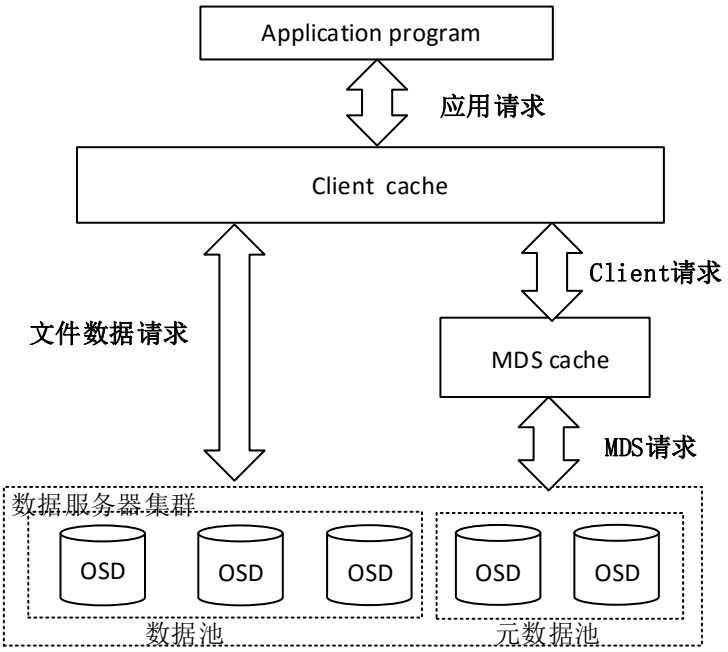


图 3.2 系统缓存架构图

Ceph 的元数据分为两层缓存结构：客户端的缓存和元数据服务器的缓存，通过对这两层缓存结构进行修改优化，提高元数据的访问性能。

元数据请求操作的时延可以反映出系统的性能，为了更加具体，使用公式加以说明。假设元数据的访问总响应延时为T_{time}，客户端的处理延时为T_{client}，MDS 的处理延时为T_{MDS}，OSD 的处理时延为T_{OSD}，客户端和 MDS 及 MDS 和 OSD 间的网络传输延时分别为T_{mdstra}和T_{osdtra}，则元数据访问响应时延满足公式 3-1。

$$T_{\text{time}} = T_{\text{client}} + T_{\text{mdstra}} + T_{\text{MDS}} + T_{\text{osdtra}} + T_{\text{OSD}} \tag{3-1}$$

从公式可以看出主要分为网络传输时延和节点的处理时延，所以可以从这两个方面进行元数据访问的优化。

1. 节点的处理延时方面，T_{client}是客户端的处理时延，主要是客户端元数据缓存的查找，可以增加预取机制和优化路径查找来进行优化；T_{MDS}是 MDS 处理时延，其又可以分为元数据的缓存查找时延和元数据日志处理时延，可以优化元数据缓存管理算法和分离细化请求处理来减少处理延时。

2. 网络传输延时方面，通过上面优化缓存等处理操作，提高命中率，较少元数

据的查找取回操作就减少了元数据的网络传输次数。

下面分别从客户端和元数据服务器两个方面介绍元数据访问改进的设计。首先客户端主要增加了元数据预取模块和基于缓存的路径查找优化，从而提升系统的性能。

3.2 客户端元数据预取策略

3.2.1 文件访问特性

在文件系统中，文件的访问存在一些特性。根据局部性原理可知，如果一个文件正在被访问，那么它在近期很可能会被再次访问到(时间局部性)或者在不久的将来会用到的文件很可能与现在正在使用的文件在地址空间上是比邻的(空间局部性)。一般如果一个文件被访问的比较频繁，那它周围的文件或者目录也会经常被访问。比如在修改 Ceph 客户端代码时，需要经常访问/home/gkk/ceph/src/client/client.cc 这个文件，并对其进行修改，当下次进行重新编译和链接时，client 目录下的 dentry.cc、dir.cc 等这些文件也都会被访问。如果能提前把这些将来编译链接需要用到的文件信息缓存到内存中将能提高文件访问的性能。

对于元数据信息也是一样的，也存在着空间局部性原理，所以为了减少元数据访问操作延迟，可以根据客户端元数据缓存的访问记录增加元数据预取新模块，然后从 MDS 中以合适的粒度预取出相对较热的目录项元数据信息，从而有效的提高系统的性能。

3.2.2 系统架构

元数据预取作为一个新模块加入到客户端，需要与其他模块的接口进行交互，系统的架构如图 3.3 所示。预取模块(Prefetch)主要是根据客户端的目录项缓存访问历史记录来进行热度的统计并与 MDS 进行交互，所以主要和 Cache 与 MDSC 这两个模块有联系。

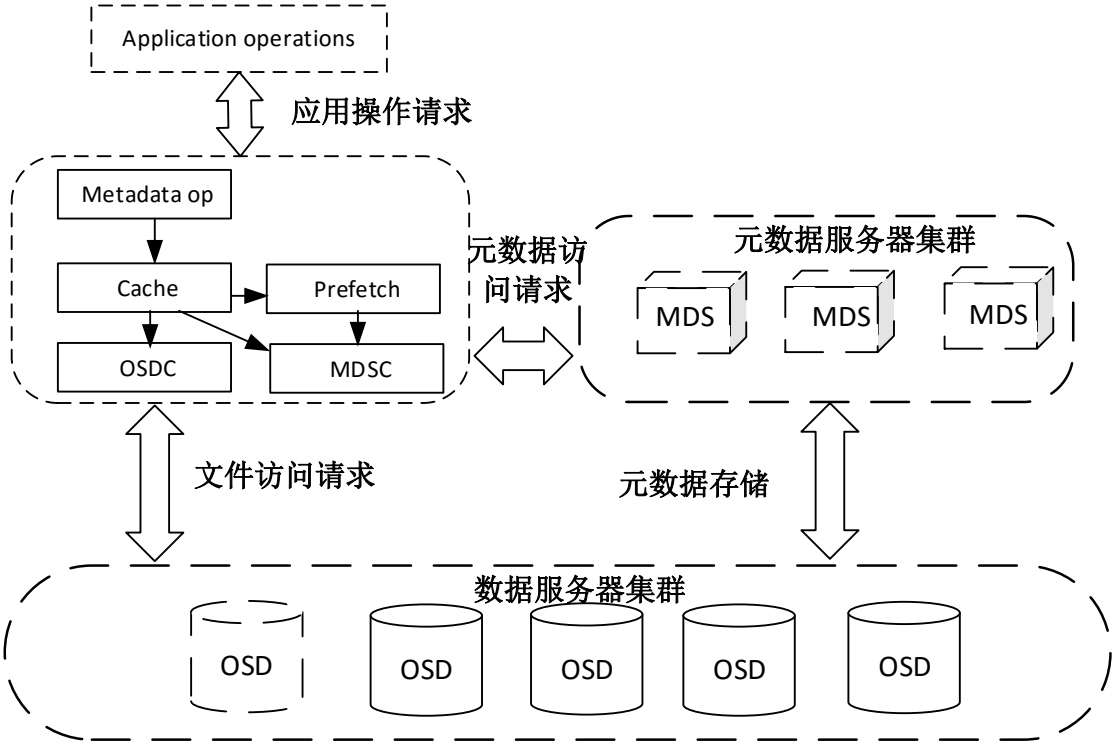


图 3.3 预取架构图

下面介绍一下用户态客户端中各个模块的功能：

Metadata op 模块：主要通过 Linux 内核的 FUSE 模块接受应用操作的请求，比如 read、write、open 等等，向用户提供的标准文件系统接口。

Cache 模块：主要是对以前访问过的元数据信息进行缓存，包括目录、目录项和索引节点三种元数据信息，构建部分目录树，加快元数据访问。

OSDC(OSD Client)模块：主要负责与 OSD 集群的通信，主要是对于 read 和 write 等这些数据操作，这个模块实现文件数据分割为一个个对象，并且对数据进行 Crush 算法定位，计算出 OSD 三副本中 primary 和 replica，然后把数据和操作请求发送到对应的 OSD。

MDSC(MDS Client)模块：主要负责与 MDS 集群的通信，主要是对于元数据的一些操作，比如 rmdir、create、setattr 等，也就是对于文件系统命名空间的修改。

3.2.3 元数据预取粒度

预取出较热文件或目录的相关联部分，提高将来访问的命中率，可以减少元数

据访问操作延时，但是分布式文件系统的目录容量可能比较大，一般具有数万或者数十万的目录项，如果直接以访问频率高的文件所在的父目录作为预取的对象，可能在某些时候开销比较大，例如由于元数据访问的突发性，可能预取的不是真正的热元数据部分，这种不准确的预取会牺牲很大的系统性能。综合考虑，应该选取一个合适的预取粒度，这里借助于 Ceph 系统的目录分片进行预取，返回客户端一个相对合适的目录大小。

在 Ceph 中，目录分片主要是用于在负载不均衡时，进行子树迁移的基本单元。元数据集群全局命名空间使用基于目录分片的树进行管理，某一目录的分片如图 3.4 所示。

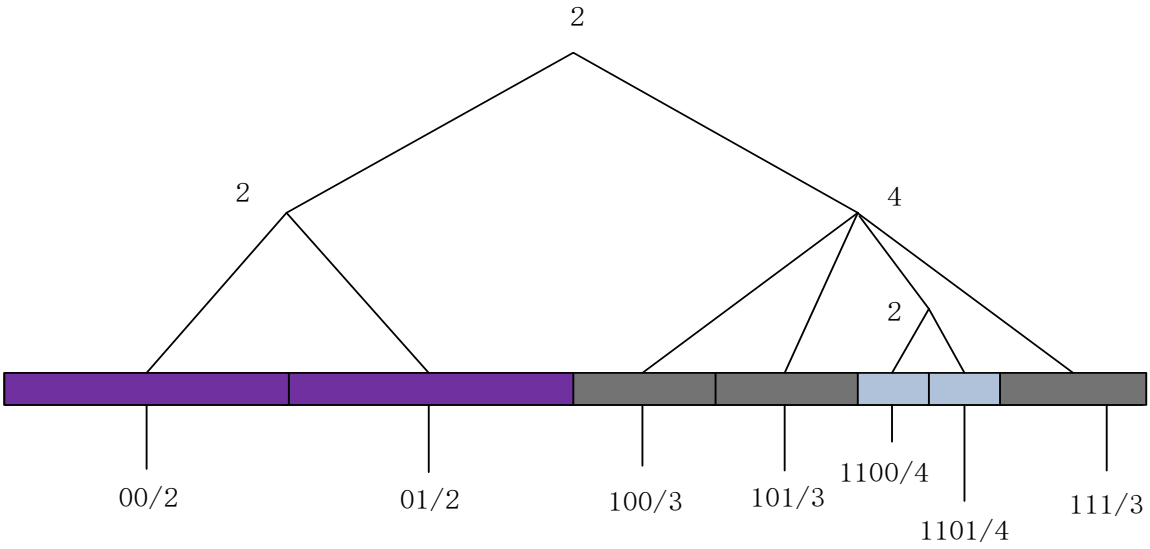


图 3.4 目录分片树

目录分片的原理是通过对一个目录下各个目录项进行负载热度统计，并根据统计热度值和目录下包含的子文件总数把一个目录在逻辑上分成几个目录，使用这些逻辑目录来管理原来目录下的所有目录项，目录项通过目录名进过 hash 计算映射到对应的目录分片。图 3.4 中目录分片树中最底层的叶子节点就是目录的分片，上层中的每个内部节点是虚拟节点，只表示其有多少个孩子节点，一般都是 2^n。分片使用比特位和掩码表示，跟 IP 子网的原理是一样的，图 3.5 表示 111/3 目录分片的表示。图 3.4 中的目录被分成大小不等的 7 个分片，用来表示目录分片的位数越多对应的分片越小，管理的目录项越少。

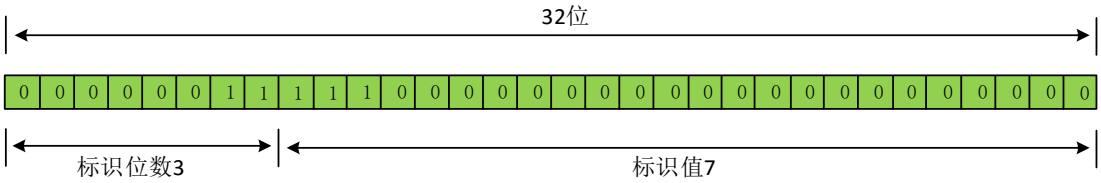


图 3.5 目录分片表示

3.2.4 预取算法

预取是根据缓存的访问记录，统计出访问频率较高的数据，提前缓存一些与其相关并将来很有可能被访问到的数据，这里主要依据元数据的空间局部性。

在 Ceph 系统中无论是客户端的缓存还是元数据服务器的缓存，它们缓存 cache 管理的都是目录项(Dentry)元数据信息，因为 Dentry 处于目录(Dir)和索引节点(Inode)中间层，进而通过 Dentry 可以很方便的找到 Dir 和 Inode 元数据信息，所以元数据的预取也是以目录项为单位进行处理，通过对目录项进行统计，实现一个统计子系统，再根据目录项的统计结果，找出满足预取热度规则的目录项，然后根据预取对象规则找到这个目录项所在的目录分片，再把这个目录分片下的文件和子目录预取到客户端的缓存中。

(1) 预取热度规则

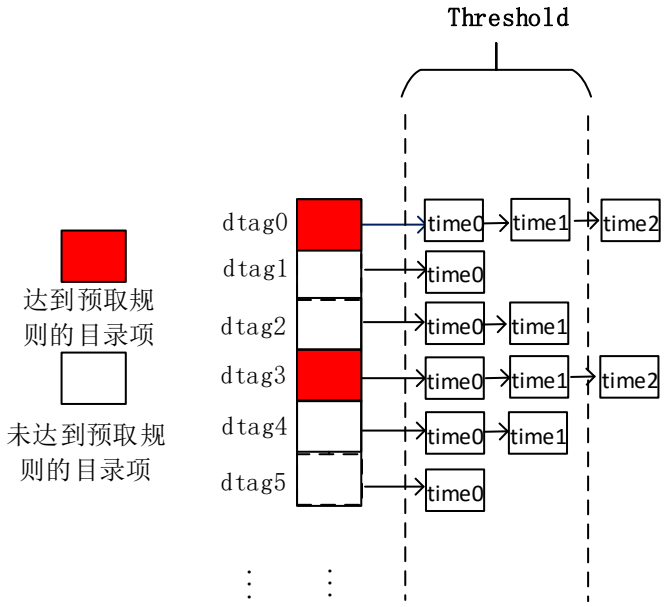


图 3.6 统计系统

为了标识出目录项的热度，需要在客户端增加一个目录项统计系统。统计系统以目录项为统计单元，需要给每个目录项一个唯一的标识，由于全局命名空间中目录名可能存在重名问题，统计系统使用目录项的名字和目录项对象的内存地址两者作为目录项的唯一标识，记为 dtag。统计计数往往是统计一段时间的访问次数总和，当统计时间段很长时，总和不能准确地反映出时间段内平均的访问情况；而当统计时间段很短时，往往又不能达到一定的统计量。为了能有效地管理控制统计时间段，统计系统中记录每次目录项访问的时间戳。

统计系统如图 3.6 所示，系统为每一个目录项维护一条时间戳队列，时间戳是指这个目录项创建和被访问到的时间点，队列把时间戳按照时间先后顺序进行排序，时间久的项在队列头，时间新的项在队列尾，通过某一时刻统计队列的长度，即可获得某一时间段内目录项的访问频率。

为了标识出某一时间段内热的目录项，需要设定一个门限值，比如 10，门限值的准确取值可以通过测试实验中的控制变量法得出。在每次访问元数据，通过缓存查找目录项时，如果其对应的队列长度达到这一门限值，就对这目录项进行相应的预取。

(2) 预取对象规则

文件系统中的元数据主要分为文件元数据和目录元数据，所以根据识别出的热目录项类型预取对象也分为两种情况。

预取对象的选择需要在 MDS 元数据服务器端进行，当客户端的热目录项为文件时，需要找出此文件所在上层父目录分片，此目录分片便是所要预取的对象，如图 3.7 所示。

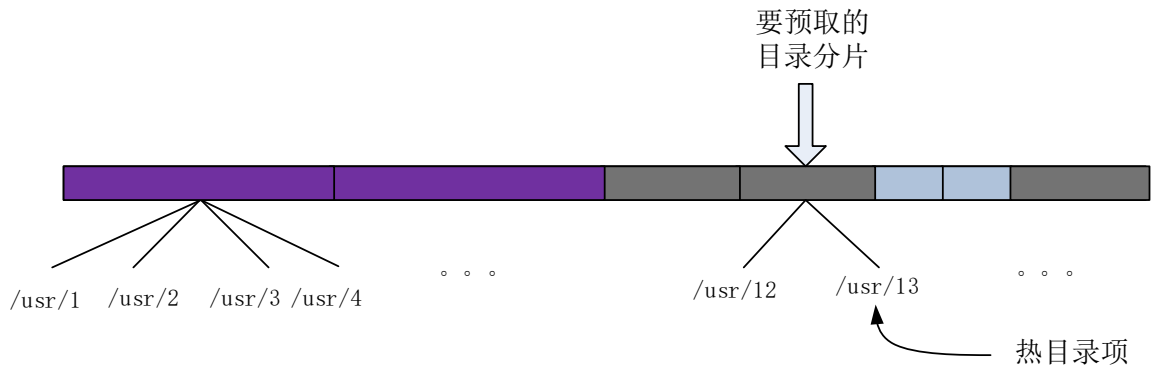


图 3.7 当热目录项为文件

当热目录项为子目录，也就是比较热的是一个文件夹时，则不仅可以预取这个目录的上层父目录分片，还可以预取这个目录下的内容。在取子目录下的目录分片时，由于不知道其下目录分片的热度情况，为了尽量减少错误的预取操作所带来的开销，预取最小的目录分片，由图 3.4 和 3.5 可知，表示目录分片的值越大，即位数越多，则目录分片就越小。如图 3.8 所示为热目录项为子目录时的预取对象选取。

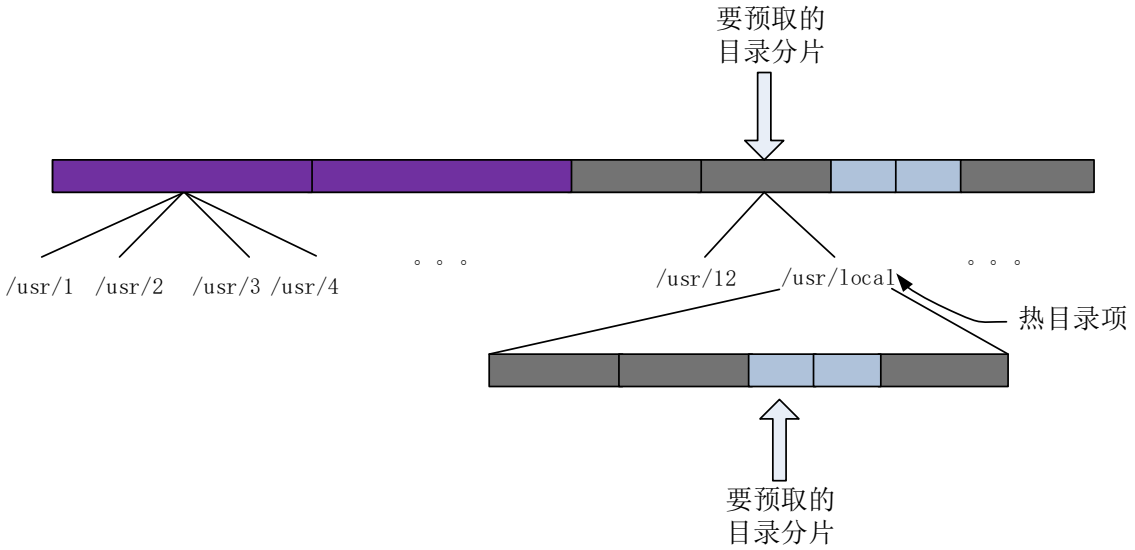
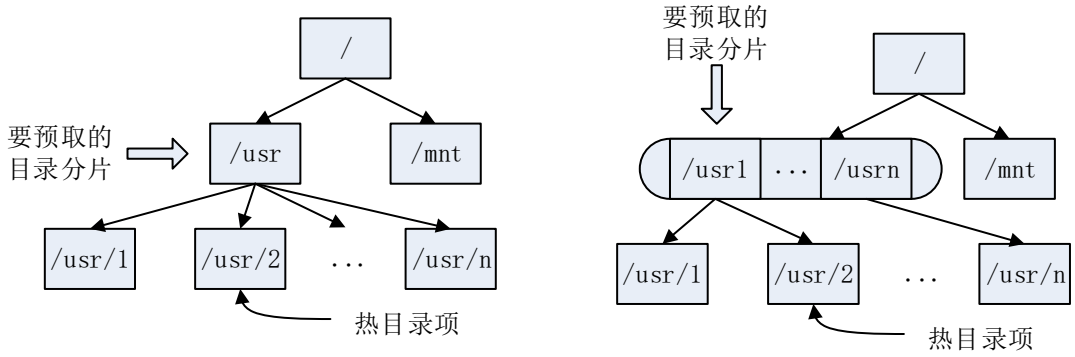


图 3.8 当热目录项为子目录

热的目录项通过客户端与元数据服务器的接口封装成元数据请求操作发送到 MDS，再根据目录项名称进行 hash 映射，找到目录项所在的目录分片，然后把这个目录分片内容全部发送回客户端缓存到 cache 中。当热目录项为子目录时，还要预取出子尺寸最小并且位值最大的目录分片。

(3) 目录分片分割和合并时预取对象的调整

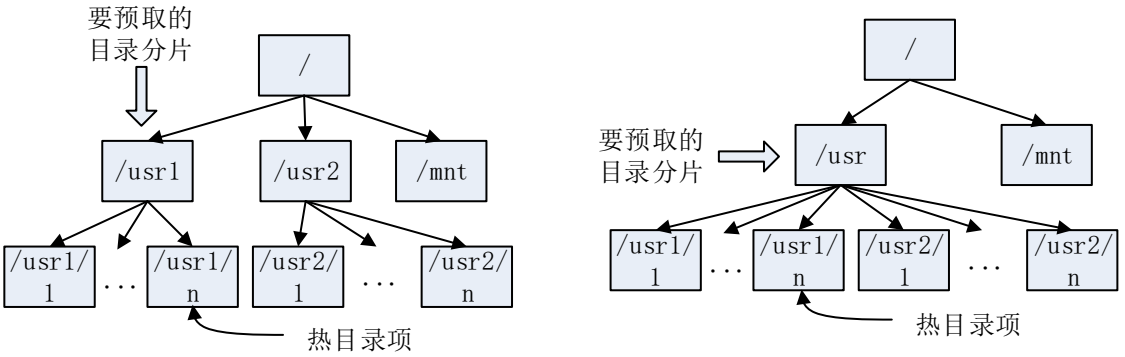


(a) 分割前　　(b) 分割后

图 3.9 目录分片分割前后

当目录分片下的文件和子目录总和超过指定阈值或者此目录分片的操作热度超过指定阈值时需要进行目录分片的分割，把原来的目录分片分割成几个小的目录分片，此时预取的对象也要进行相应的调整。如图 3.9 所示。

当目录分片下文件和子目录总和小于指定阈值时需要进行目录分片的合并，主要是为了减少元数据目录结构的开销，此时预取的对象也要进行相应的调整，如图 3.10 所示。



(a) 合并前　　(b) 合并后

图 3.10 目录分片合并前后

3.2.5 统计系统的管理

(1) 访问统计的加入

访问统计的加入主要是增加时间戳记录，借助客户端缓存的访问记录，有两个时机需要增加时间戳。一个是在创建新目录项时，此时需要在统计系统增加一个新的 dtag 和对应的时间戳队列；另一个是在缓存中访问到已存在的目录项时，找到对应的 dtag，在时间戳队列中加入新的时间访问点。创建新目录项时又分为两种情况，一个是在处理 mknod 和 mkdir 这些请求命令时，另一种是在缓存未命中，需到 MDS 取回目录项。

(2) 访问统计的删除

由于内存是比较稀缺的资源，并且统计系统的访问统计主要是依赖于客户端的缓存访问记录，历史访问统计在某些条件下需要进行删除操作。一方面当某个目录项被统计为较热时，在进行相应的预取之后，需要把这个目录项统计队列清空，以避免重复的预取；另一方面当某个目录项在客户端缓存中删除时，需要在统计系统删除相关 dtag 和时间戳队列，如果不删除，统计系统将会一直保持这个目录项，浪费

空间而且会污染统计系统。其中删除目录项的操作又有两种情景，一种是缓存满，需淘汰掉一些访问少的目录项；另一种是处理 rmdir 和 unlink 请求操作，在这种情况下需在删除前进行检查目录项是否达到预取指标。

(3) 访问统计的清理

客户端设有一个定时器，每隔 1s 会进行系统的一些统计和 cache 的清理。统计系统也需要清理，主要是清理时间戳，因为元数据的访问具有随机性和突发性，时间戳队列的长度是反映首尾时间戳所代表的这段时间内的目录项访问频率，累计统计的时间越长，统计的准确性越差，因为可能在这长时间段内，某一小段时间访问比较频繁，其余时间访问比较少，导致平均访问频率还比较高，所以需要根据客户端的定时器进行定期的清理。然而定时器每隔 1s 进行调用，1s 之内可能不能很好的达到统计效果。统计系统设计独立定时器，进行统计系统的清理。由于统计系统中记录的是时间戳，可以准确的通过时间戳删除指定时间内的统计结点。清理 trim 如图 3.11 所示。

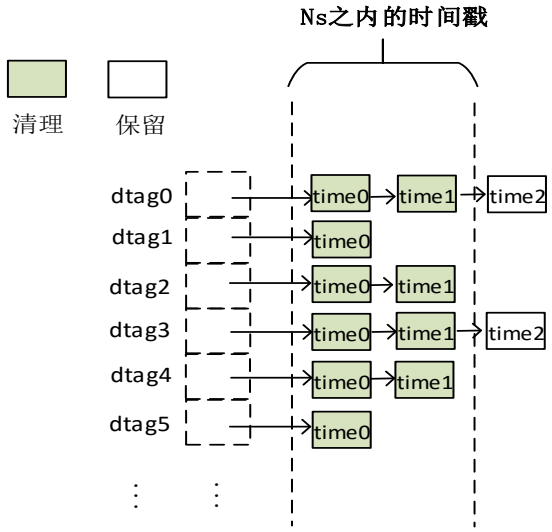


图 3.11 统计系统的清理

3.3 客户端路径查找优化策略

3.3.1 路径查找特性

Ceph 文件系统中的路径查找跟本地文件系统原理是一样的，都是先找到目录，再根据目录项的名字找到目录项，如果目录项是个目录则再进行重复的查找操作。

如图 3.12 所示为查找/usr 下的 1.txt 文件。

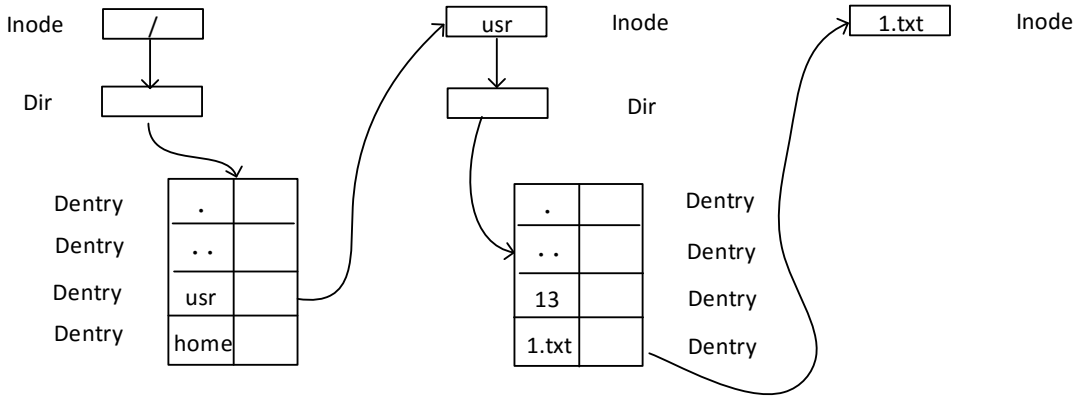


图 3.12　/usr/1.txt 的路径查询

在客户端路径查找时，如果路径某一级目录在缓存中没有查找到，则需要把路径中剩下的没找到的目录依次发送到元数据服务器进行查找，比如要查找的文件路径是/usr/local/share/file/1.txt，假设 local 目录后面三级目录都不在客户端的缓存中，则需要发送三次路径查找请求到元数据服务器，依次是查找 share，查找 file，查找 1.txt，原系统的这种查找主要是考虑可能 share 在客户端没缓存，而缓存了 file，所以在发送查找请求 share 到元数据服务器之后，接下来的 file 查找可以在客户端命中，但是就像上面假设的如果后面的每一级都不在缓存中，那就需要多次进行 MDS 的查找取回。而这些交互的网络延迟是比较长的，若能在这种情况下把后面所有没有缓存的目录改成一个相对路径进行一次与 MDS 的交互，取出这条路径上的元数据并缓存在客户端，这样便能减少网络延时，提升性能。

3.3.2 路径查找优化策略

为了优化路径查询性能，设计使用连续两级目录项未命中策略，减少与 MDS 的交互次数。比如要查找/usr/local/share/file/1.txt 这个文件路径，usr 和 local 目录都在客户端缓存中，所以能命中，而 share 目录不在缓存中，这时可以通过判断下一级 file 目录是否在客户端缓存中，来优化路径查找：如果没有找到，则说明 share 和 file 目录都不在缓存中，则把 share/file/1.txt 转化为新的相对查找路径与 MDS 进行一次交互查找；如果找到则只需要发送 share 目录到 MDS 进行查找，也只需要进行一次与 MDS 的交互，查找后续路径的目录项时如果还遇到缓存未命中则再进行类似操作。

这里使用公式加以举例说明，还是以上面路径为例，T_{mdstra} 表示一次 Client 到 MDS 的网络延时，T_{tra} 表示路径查询的总网络时延(这里暂不考虑 MDS 到 OSD 的时延)，假设路径中有 3 个目录项不在缓存中，且未命中缓存的下级目录命中与不命中的概率各为$\frac{1}{2}$。公式 3-2 表示原系统路径查询的网络总时延。

$$T_{\text{tra}}=3T_{\text{mdstra}} \tag{3-2}$$

公式 3-3 表示使用连续两级目录是否命中标准后，路径查询的网络总时延。

$$T_{\text{mdstra}}=\frac{1}{2}\left(\frac{1}{2}T_{\text{mdstra}}+\frac{1}{2}\left(T_{\text{mdstra}}+\frac{1}{2}T_{\text{mdstra}}\right)\right)=\frac{5}{8}T_{\text{mdstra}} \tag{3-3}$$

从公式可以看出减少了 2 倍多的网络延时，进而可以加快元数据的访问性能。

3.3.3 同名目录项统计

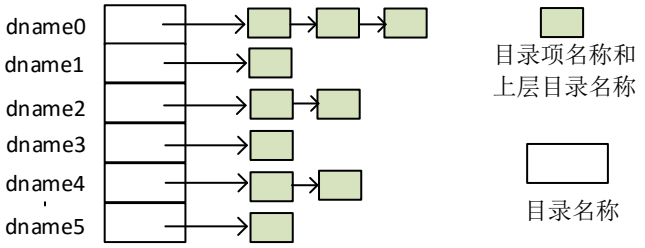


图 3.13 目录项的同名统计

从图 3.12 可知，每个目录项的查找都需要先找到上次父目录的信息，所以在使用连续两级未命中策略时，第二级目录无法直接从原系统缓存中得知是否命中，同时又考虑到同名目录项的情况，通过基于缓存的统计系统，设计目录项的同名统计，这样第二级目录项可以通过名称在同名目录项统计中进行查找，找到则表示第二级目录项在缓存有，反之，使用连续两级未命中规则。为了避免重名，同名统计通过使用本目录项名和上层父目录名两级目录名称区分同名目录项。图 3.13 为目录项的同名统计。

3.3.4 生成新的相对查找路径

当上下级目录都没有在客户端缓存中时，需要生成新的相对查找路径与 MDS 进

行一次交互，把新路径上的所有目录都缓存到客户端。

仍然以/usr/local/share/file/1.txt 查找路径为例，假设 share 和 file 目录不在客户端的缓存中，此时生成一个新的相对查找路径 local/share/file/1.txt，然后根据 local 目录查找出管理其的元数据服务器，之后把这个新路径发送到MDS，并把这个路径上local之后的目录项 share、file 和 1.txt 缓存到客户端。就是从未找到目录的父目录开始截取原查找路径生成新的相对查找路径。

以上便是客户端元数据缓存的两个优化设计，基于缓存访问记录的预取和基于缓存的路径查找优化。下面介绍元数据服务器端的缓存优化和请求处理优化。

3.4 MDS 元数据缓存算法

MDS 集群作为文件系统的第二层缓存，分布式地管理整个文件系统的命名空间。MDS 集群只使用内存不使用磁盘存储元数据，如果把整个 Ceph 文件系统环境看成一台单机，那么 MDS 集群相当于内存，OSD 集群相当于磁盘。内存总是比较稀缺的资源，因为 MDS 缓存元数据，如果能有效地管理元数据，准确识别出访问频繁的元数据，在内存紧张时，删除访问比较少的元数据，释放内存空间，这样能提高命中率，有效地利用内存资源。

3.4.1 MDS 元数据流

如图 3.14 所示为 MDS 元数据请求流程图，主要分为以下几个步骤：

1. 客户端生成元数据请求，封装成网络消息发送元数据请求到 MDS。

2. MDS 使用服务处理模块(Server)接受并处理请求，然后在元数据的缓存模块(MDCache)中对请求操作的元数据进行查询，如果命中则执行步骤(3),否则执行步骤(5)。

3. 在 MDCache 中命中，则对相应的元数据进行修改等操作，并生成相关的日志写到 OSD 中。

4. 待日志写完后，在内存中修改的元数据才替换掉原来的元数据，并对以后的元数据访问生效。

5. 如果在 MDCache 中没有命中，则先要到 OSD 中取出元数据缓存到 MDCache 中，然后再进行元数据修改等操作。

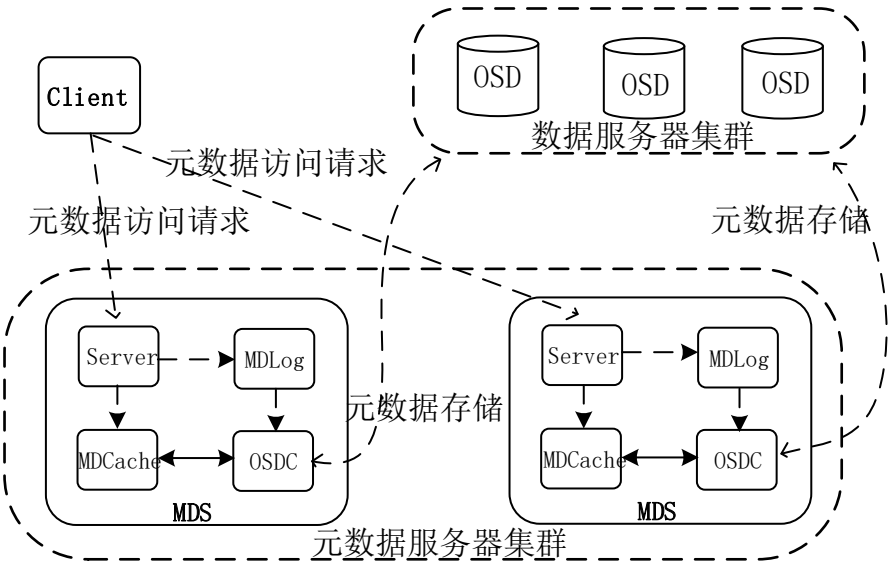


图 3.14 元数据访问流程

3.4.2 冷热元数据的识别

MDS 中的元数据也主要分为目录、目录项、索引节点，跟客户端的差不多，只是增加了一些MDS集群的相关信息，与Dir、Dentry和Inode对应分别为CDir、CDentry和 CInode，三者之间的关系与图 3.1 类似。在 Ceph 分布式文件系统中，为了更有针对性的识别冷热元数据，可以根据元数据的某一语义进行分组。

Ceph 文件系统的整个目录树被动态分割成几个子树由对应的 MDS 进行管理，划分后的目录子树被分配到的 MDS 称之为属主 MDS。由于集群访问的数据量可能很大，某些元数据服务器访问负载比较重，为了解决这种元数据服务器访问负载不均衡的情况，Ceph 系统在相对于属主 MDS 的其他元数据服务器上生成副本元数据信息，通过这些副本元数据信息来执行一些读操作，减轻属主 MDS 的负载，但是对于写和修改元数据操作只能在属主 MDS 上执行。根据子树的划分和所处元数据服务器位置，将元数据划分为属主元数据和副本元数据两类，如图 3.15 所示。

在同一台元数据服务器上，由属主元数据和副本元数据的操作特性可知，在副本元数据和属主元数据访问频率相同的情况下，可以优先删除副本元数据，所以属主元数据整体相比于副本元数据较热。尤其像在一台 MDS 由于宕机进行重启时为了减少重构目录树的时间，一般会清空副本元数据，这里指的是不同对象的属主元数据和副本元数据。继而在元数据缓存管理时通过判断划分为属主元数据和副本元数

据进行分组管理。

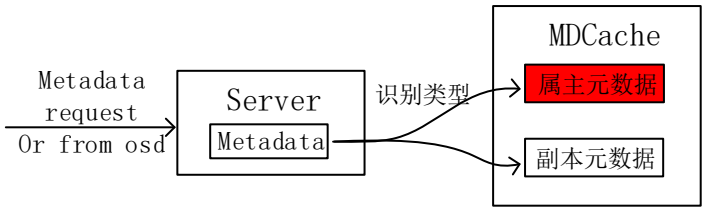


图 3.15 元数据分组

这里所说的元数据主要是 CDentry 和 CInode 两种，Ceph 系统中副本元数据的由来有两种：

1. 先生成空状态的属主元数据信息，再替换为副本元数据信；

2. 把 stray 状态的属主元数据信息替换为副本元数据信息。

Ceph 系统需要副本的元数据信息也有两个时机：

1. 当某一 MDS 接受客户端的元数据请求，在 MDCache 中查找元数据时，如果没有找到，则到集群的其他 MDS 上去查找元数据，如果其他另一台 MDS 在其缓存中找到了元数据信息，则第一台 MDS 就需要把第二台 MDS 中元数据信息复制一份到 MDCache 中，就有了副本元数据。

2. 在某个 MDS 负载比较重时，需要进行子树复制，找出比较热的元数据，复制到负载比较轻的 MDS 上，复制过去之后生成的便是副本元数据。

3.4.3 干净目录项元数据管理

在 MDS 中，元数据的更新写回 OSD 时是以目录分片为单位，把这个目录分片下的目录项作为操作内容写回到 OSD 中，不能以单独的目录项写回到 OSD。基于这个原理在淘汰缓存中的元数据时不能淘汰脏的目录项，必须等脏目录项所在的父目录分片整体更新到 OSD 后变为干净才可以进行淘汰，也就是说只能淘汰干净的目录项。相对于原系统，通过把干净目录项和脏目录项分开进行管理以减少淘汰时的查询时间。

元数据的管理主要是管理 CDentry 目录项，因为目录项可以看出是 CInode 的间接索引。

干净目录项的管理旨在提高命中率，尽可能保留访问频繁的目录项，淘汰时删除访问稀少的目录项，于是采用类似 2Q 的缓存管理算法，如图 3.16 所示是分组的目录项管理，副本目录项维护了两条 LRU 链表 Q1 和 Q2，并且 Q1 链表有一个门限

值；属主目录项也同样维护了两个 LRU 链表。如果上层的元数据请求要求创建目录项或者当没有访问命中需从 OSD 中取回目录项，这些情况将目录项加入到管理队列 Q1 的 MRU，如果数据再次被访问后则移到 Q2 的 MRU 端。

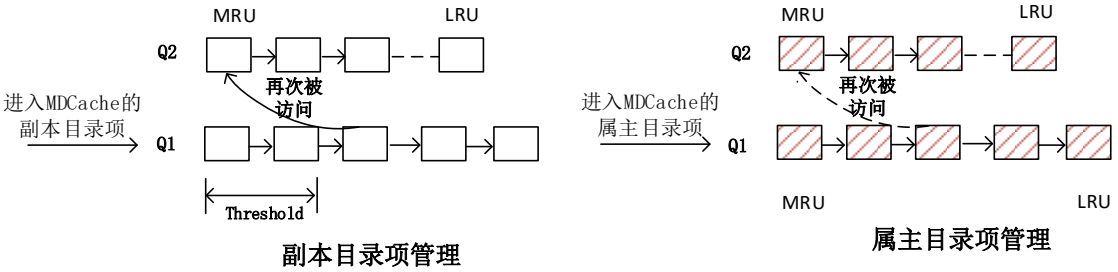


图 3.16 分组的干净目录项管理

虽然副本元数据优先级总体比属主元数据优先级低，某些情况下副本元数据正分摊着重负载 MDS 的读操作，而一些属主元数据可能访问不怎么频繁，此时可以淘汰属主目录项元数据，所以访问频率很高的副本元数据优先级可以比频率低的属主元数据高。介于此可以取消副本元数据管理的 Q2 链表和属主元数据管理的 Q1 链表，使用一个混合的 LRU 来缓存属主元数据和副本元数据，如图 3.17 所示。同时给副本目录项的 Q 链表设置了一个最小的门限值，主要是防止刚加入的副本目录项被频繁地替换出去。

在选择淘汰对象时，淘汰的顺序分别为 Q1、Q2 和 Q3。在 Q1 的长度大于 Threshold 时，淘汰 Q1 中的目录项，直到 Q1 长度小于 Threshold 时，淘汰 Q2 中的目录项，在 Q2 链表为空之后，就淘汰 Q3 链表中的目录项。

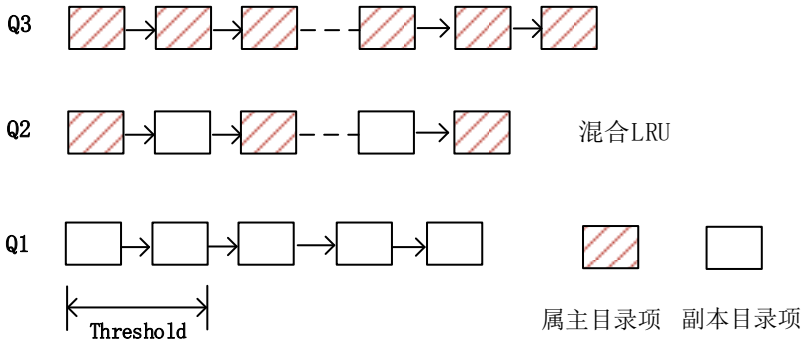


图 3.17 混合干净目录项管理

上一节中副本元数据的两种生成方式都可以看成是从属主元数据转变成副本元数据。所以在刚开始时所有的元数据都插入到干净链表的第二条混合链表上，某些

stray 元数据经过再次访问移到第三条链表上，当属主元数据变成副本元数据时，把元数据移到副本元数据分组的对应链表上。如转化前属主元数据在第三条链表上，则转化后移到第二条链表，反之则移到第一条队列。图 3.18 为属主目录项变为副本目录项示意图。

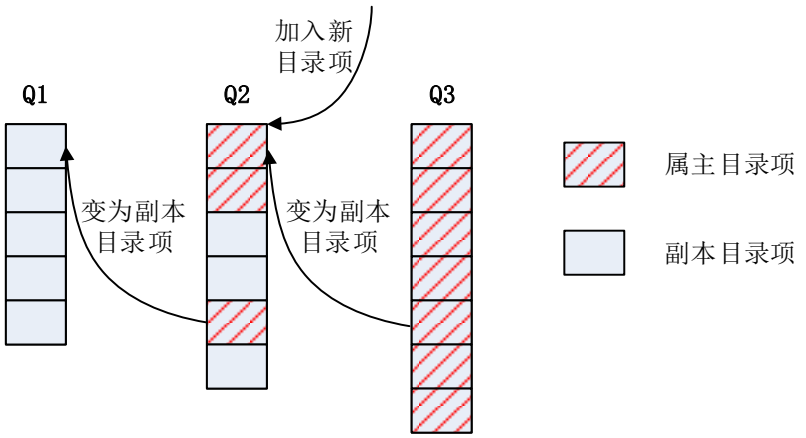


图 3.18 属主目录项转化为副本目录项

3.4.4 脏目录项元数据管理

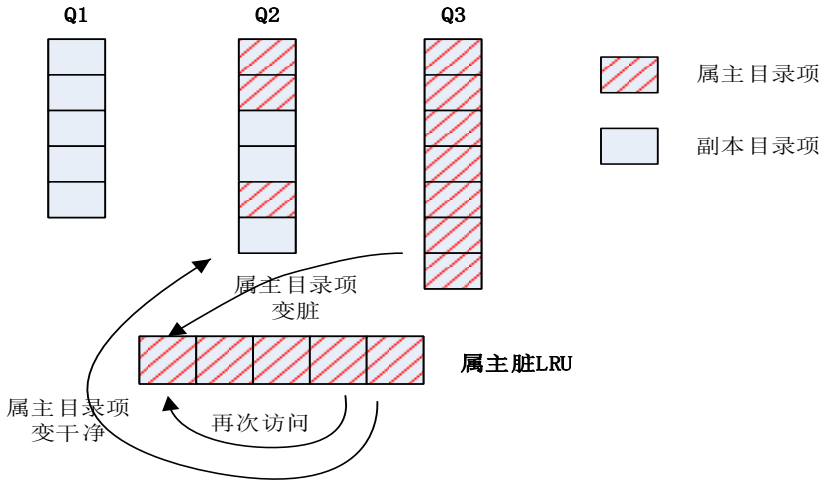


图 3.19 脏目录项管理

由于 MDS 缓存只淘汰干净的目录项，所以脏的目录项要单独管理，以减少查找干净目录项的时间。介于脏元数据不需要进行淘汰，对命中率要求不是很高，所以使用简单的 LRU 链表进行管理。同时由于副本目录项主要是用于应对读这类不会改变元数据的操作，所以只对属主目录项进行脏元数据的管理。如图 3.19 所示。属主

元数据在修改元数据属性等操作之后变为脏，从干净队列移出分别加入到脏队列中，当目录项以父目录分片为单位更新到 OSD 之后，从脏 LRU 移除加入到该组的优先级最低的地方，也就是中间混合队列的尾部。

3.4.5 干净队列的动态老化

(1) 固定比例老化

由于元数据的访问的突发性和随机性，为了防止热度高的元数据长期处于优先级高的链表中不被淘汰，这里使用老化机制来解决这种缓存污染问题，所以老化是指把优先级高的数据通过某种方式使其优先级下降。一般老化机制通过减少每个缓存对象的引用次数，比如减少为原来的一半或者减去固定的值。由于干净元数据的管理没有使用计数，本设计使用固定比例进行老化，所谓固定比例老化是指三个干净队列的容量上限之间存在着比例关系，一旦链表的当前容量超过自身比例容量的上限，就需要将队列末尾的数据移到前一优先级级队列的头部，同时属主和副本混合链表的老化只能把副本的目录项移出到 Q1 队列中。例如三个干净队列容量的固定比例为 2/9: 3/9: 4/9，其中 2/9 表示 Q1 链表的容量上限占干净队列总容量的比例，其余类推。

(2) 动态调整缓存容量

Ceph 中 MDCache 的缓存总容量上限是通过配置文件来设置的，系统在不同的负载情况下，资源使用情况也不同，比如当 MDS 负载比较重，系统资源比较紧张时，尤其是内存，所以此时如果还是按照配置文件中的缓存容量上限来管理缓存可能使系统的资源使用更加严峻。为了反映出系统当前负载对缓存总容量的影响，在定时老化时通过当前内存使用情况来动态的调整缓存总容量上限，当 MDS 负载比较重，内存比较紧张时，可以调小缓存总容量；当负载比较轻，内存充裕时，调大缓存总容量，这里通过当前未使用内存和总内存的比例来表示调节系数。假设 k_i(i=1,2,3)表示对应 Q_i (i=1,2,3)队列或链表容量占总容量的上限比例，C_{total} 表示缓存总容量，C_i (i=1,2,3)表示老化后相应队列或链表的上限容量，u 表示当前内存利用率，C_{dirty} 为当前脏目录项链表容量，参数之间满足 3-4 公式。

$$C_i = k_i\left((1-u)*C_{total} - C_{dirty}\right) \qquad \text{(i=1,2,3)} \qquad \text{(3-4)}$$

这里假设根据内存调节后缓存总量为 18 个目录项，脏目录项当前为 0，Q1、Q2 和 Q3 当前的容量分别是 3、6、9，所占比例依次为 2/9，3/9 和 4/9，设置访问频繁高的链表容量大一些，以更高地提高命中率。老化调整后容量应该为 4、6、8，如图 3.20 所示。

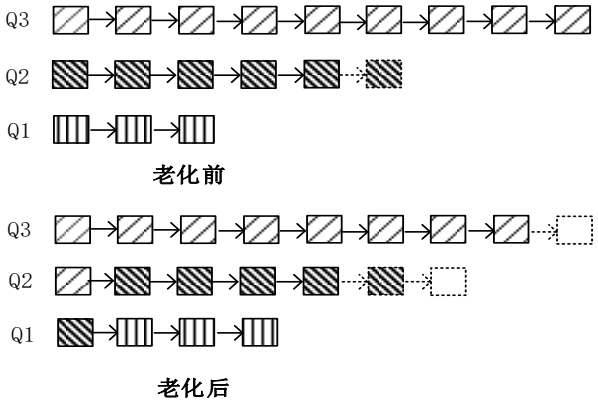


图 3.20 优先级队列老化前后

(3) 老化时机

老化的执行可以借助于 MDS 一个全局的定时器，定时的进行老化，调节链表。

3.4.6 主被动淘汰和主动倒盘

当缓存空间不够时，需要淘汰掉一些访问频率低的目录项，以释放空间缓存新的元数据，可以分为两类：主动淘汰和被动淘汰。

1. 主动淘汰是指给系统设计一个定时器(比如每隔 5s)，检测缓存的空间是不是达到了缓存容量的上限，这里的缓存总容量由上一节可知根据内存使用情况动态的变化，然后淘汰掉缓存中访问频率低的干净目录项，以释放空间。

2. 被动淘汰是指当元数据访问在 MDS 缓存中没有命中时，需要到 OSD 进行取回，并缓存在 MDS 缓存中，由于从 OSD 存取元数据都是以目录分片为单位，而目录分片的大小可能不同，所以缓存空间在快接近缓存总容量时，就需要进行淘汰。比如在从 OSD 取出元数据信息时，检测缓存的空闲空间有没有总容量的 1%，没有的话就要进行淘汰，直到空闲空间达到 3%为止。

在 Ceph 中，元数据倒盘是指把脏的元数据写到 OSD，以保持元数据的一致性。

倒盘也分为主动和被动，但是 Ceph 系统由于目录项的更新是以父目录分片为单位，不能单独的把脏的目录项写到 OSD 中所以只能主动倒盘。元数据的修改操作是以日志来保证可靠性的，而日志又以日志片段为逻辑组织单位，当一个逻辑日志片段上的日志都写到 OSD 之后，这一日志片段上的日志相关的脏元数据就可以更新到 OSD 上去，由于日志写到 OSD 上的速度不确定，只能设计一个定时器(比如 5s)，检测如果有日志片段写完，就进行元数据倒盘。如图 3.21 所示。

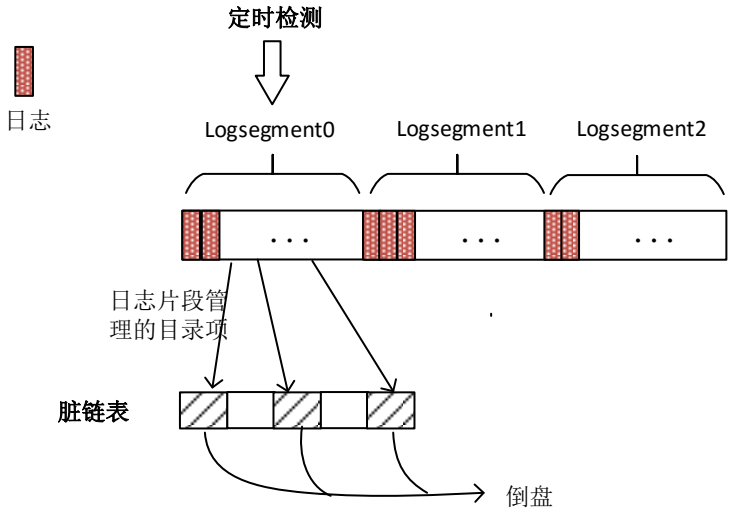


图 3.21 脏目录项主动倒盘

3.4.7 目录项索引的管理

为了能对目录项进行快速的查找，需要给缓存中的目录项建立索引信息，这里通过设计两种索引来分别适应目录的读取和目录下目录项的查找两种情景。以目录为单位建立目录项索引信息，对属于一个目录下的所有目录项进行索引。

(1) 基于红黑树的索引

在处理 readdir、prefetch 等读取一个目录下元数据操作时，需要让目录项按字典排序顺序输出，此时可以使用红黑树对目录项进行排序，建立索引。

红黑树是一种自平衡二叉查找树，跟一般的平衡二叉树相比，它通过把节点划分为红黑两种颜色进行平衡约束，可以通过红黑树的中序遍历有序输出。

(2) 基于 hash 的索引

在元数据服务器中进行文件路径查找时，对一个目录下的目录项通过 hash 映射建立索引，可以在 O(1)的时间复杂度内找到目标目录项。

3.5 MDS 元数据处理流程优化

MDS 中为了保证元数据操作的可靠性，使用 writeahead 方式即写前日志来记录元数据操作，这种方式使原来的一次 I/O 操作转变成两次 I/O 操作，即通常所说的写放大，需要在更新元数据前先写日志，所以元数据操作比较慢，如果能优化日志的提交流程，便可以提升元数据操作性能。

在原系统中，元数据请求处理流程较长，MDS 只使用一个线程完成了元数据消息的接受、元数据缓存的查找、元数据的修改处理、日志的生成、写入日志缓存以及日志缓存满需要刷新到 OSD 上这些任务。只有当每次完成了上诉流程，才能继续处理下一个元数据请求，元数据服务器请求处理速度受到限制，然而其中的日志处理部分与后续元数据的处理之间没有影响关系，所以可以对元数据请求处理的粒度进行细分，分为三个处理阶段：元数据的处理、日志写缓存和日志缓存满刷新到 OSD 上，对这三个阶段使用流水线型，从而加快元数据的处理速度。图 3.22 表示服务线程处理一次元数据操作的流程以及与 MDS 模块之间的对应关系。

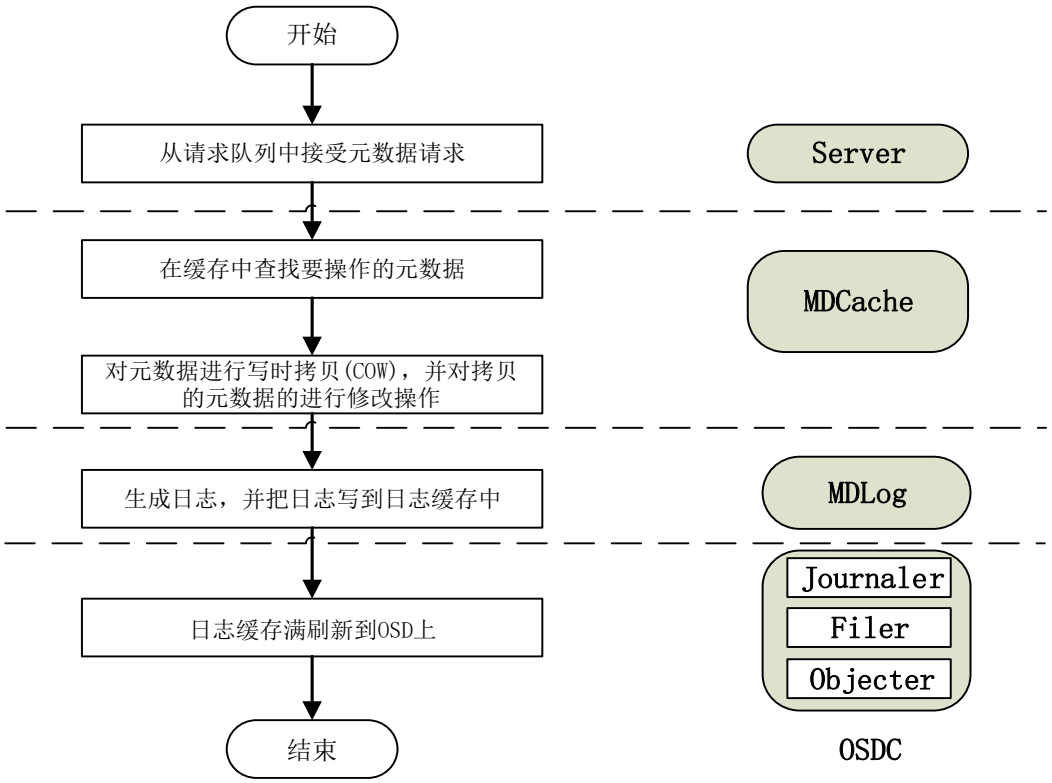


图 3.22 服务线程处理元数据流程

图 3.22 中，元数据操作流程主要经过 Server、MDCache、MDLog 和 OSDC 四个

模块，OSDC 又细分为 Journaler、Filer 和 Objecter 三个子模块。下面介绍一下各个模块的功能：

Server 模块：主要负责接收网络消息，并从封装的消息中解析出元数据请求，根据请求类型执行各种元数据处理操作。

MDCache 模块：主要负责 CDir、CDentry 和 CInode 三种元数据的缓存，文件路径查找，主副本元数据的管理和维护整个文件系统的命名空间。

MDLog 模块：主要负责日志的管理，包括日志文件的创建、打开、日志写缓存和日志的恢复等，同时还管理日志的逻辑分片 LogSegment。

Journaler 模块：主要负责日志写缓存满刷新到 OSD 上以及日志恢复重做时的预取。

Filer 模块：主要负责对把上层文件内容映射到 RADOS 底层对象，即文件内容所在的对象编号。

Objecter 模块：是 RADOS 提供的对象操作的接口，比如读写对象的数据，读写对象的属性。

3.5.1 元数据请求阶段设计

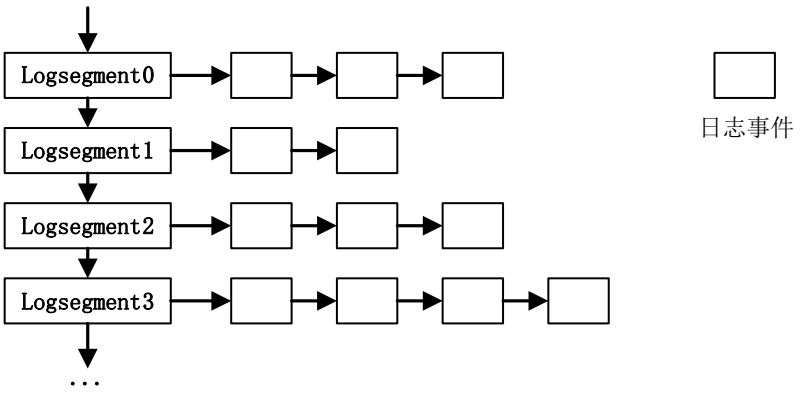


图 3.23 日志处理链表

任务细化后的元数据请求阶段主要执行图 3.22 中的接受、查找和修改元数据操作，同时生产相关的日志信息，即主要执行 Server 和 MDCache 模块，在执行完这些操作后就可以继续处理下一个元数据请求。相比于原系统，缩短了元数据操作流程，加快了元数据处理的速度。

此阶段还会产生日志信息，由于三个阶段细分后分别由不同的线程进行管理，

所以需要设计把这个阶段生成的日志信息交给日志写缓存阶段，这里使用生产者消费者模式来管理生成的日志。生产者消费者模式是指生产者不断的生产出数据交到缓存中，而消费者不断的从缓存中取出数据进行处理，这里的缓存一般使用一个队列来进行管理。在 MDS 处理完元数据请求之后，对生成的日志信息进行封装然后提交到日志缓存处理队列中。图 3.23 所示为处理队列。

在图 3.23 中，日志还是以日志分段为逻辑单位进行管理，给每一个日志分段分配一个日志管理链表，所有的日志分段又以一链表进行管理。在元数据操作服务线程产生日志时，根据日志所属的日志分段寻找相应的位置，日志分段逻辑上是 4M 大小，由于日志的大小不是一定的，所以每一个日志分段下日志的数目不是相同的，并且日志分段是依次生成的，比如必须在 Logsegment0 满了以后再生成 Logsegment1，同时由于日志之间存在着写先后位置关系所以日志必须是依次往后加入相应日志分段的链表。

3.5.2 日志写缓存阶段设计

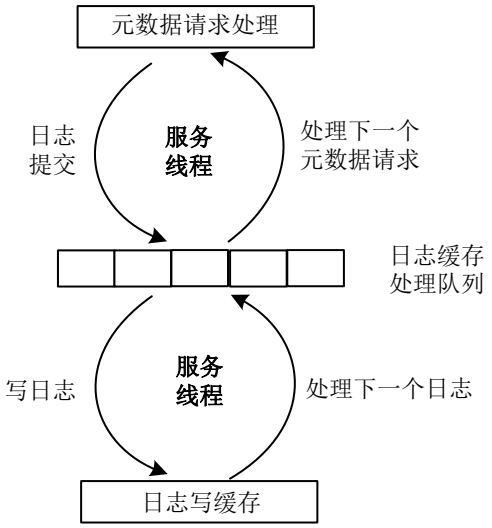


图 3.24 日志写缓存操作

任务细化之后，提供一个专门的线程服务执行日志的写缓存操作，主要是执行 MDLog 模块，从日志处理链表中依次取出日志事件，通过追加形式加入写缓存中。在加入写缓存前需要做一些准备操作：日志事件的序列化和定位写缓存的写位置。

日志事件的序列化主要是把日志事件类对象的数据成员转化到一个 bufferlist 中，类似于转化为一个字符串，这样便于网络的传输。定位写的位置可以通过记录

保存上次写完缓存后的位置。图 3.24 所示为元数据请求阶段和日志写缓存阶段的交互图。

3.5.3 日志缓存刷新阶段设计

在日志的写缓存满一个对象大小时，就需要通过网络写到 OSD 上，这段时间内是不能再写日志操作，如果网络状态不好，那就需要等很长时间，而此时日志缓存处理链表中日志会不断的加入，导致日志缓存占用的内存会不断的增加，严重时可能会影响系统的性能。本设计使用双写缓存，交替写日志，在一个写缓存满时，日志提交到另一个写缓存中，同时满的缓存提交到 OSD 上，并使用一个单独的服务线程处理提交到 OSD 的过程。如图 3.25 所示。

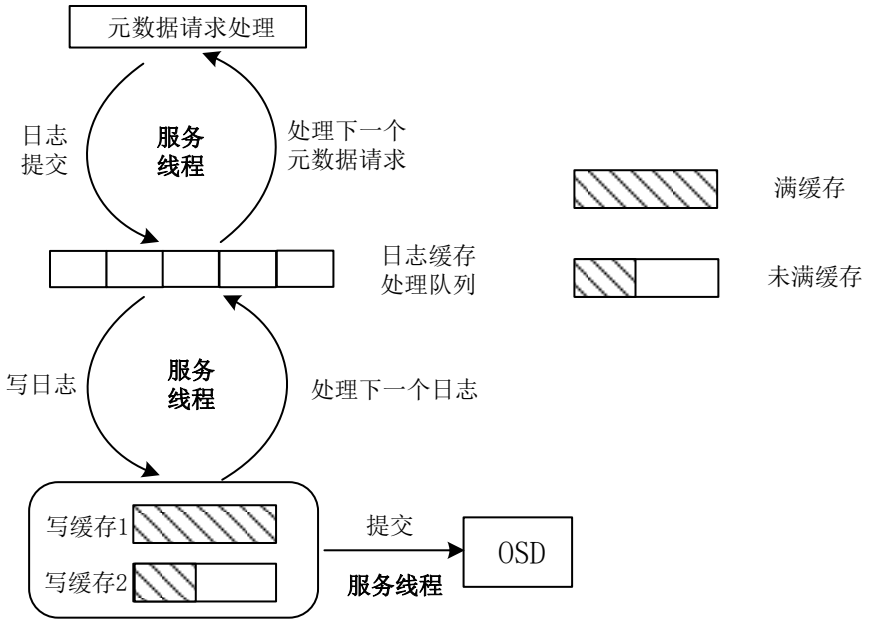


图 3.25 双日志缓存结构

日志写缓存在达到 4M 大小时，需要把日志刷新到 OSD 上，指定 4M 大小只要是为了满足 RADOS 底层存储的对象大小。在判断写缓存是否满时需要进行计算，同时需借助追加写完之后的写缓存位置和上次刷新之后的位置两个参数。假设写缓存位置为 write_pos，上次刷新之后的位置为 flush_pos，RADOS 中对象大小为 object_size，write_obj_id 表示此缓存所在对象的 id 号，flush_obj_id 表示上次刷新对象的 id 号。

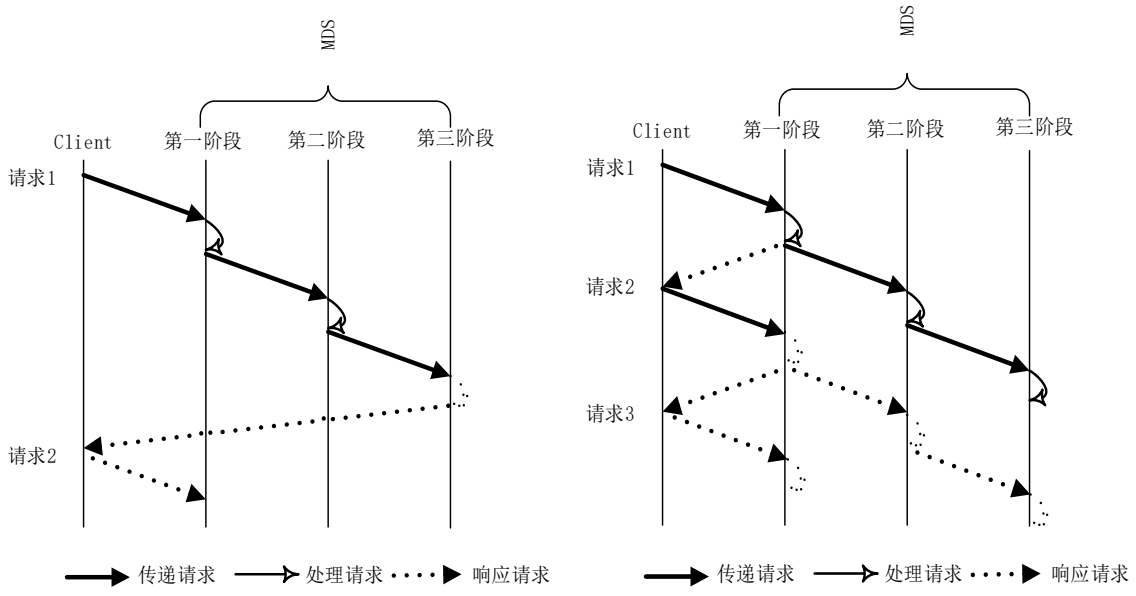
$$write\_obj\_id=write\_pos/object\_size \tag{3-5}$$

$$fulsh\_obj\_id=flush\_pos/object\_size \tag{3-6}$$

通过判断 write_obj_id 和 flush_obj_id 是否相等即可得知写缓存是否满一个对象。

3.5.4 三阶段流水线模型

在元数据请求细化为三个阶段之后，由于两个请求的各阶段是没有关联的，所以可以并行运行，可以充分的利用系统的资源，并且加快元数据处理的速度，图 3.26(a)为原系统请求处理流程，图中每一个请求处理周期太长，系统性能比较低，图 3.26(b)为三阶段流水线请求处理过程，可以看出请求速度明显提升了。



(a) 原请求过程　　(b) 流水线请求过程

图 3.26 使用流水线模型对比

3.6 本章小结

本章根据需求，对客户端和 MDS 两级缓存分别做了修改，以提高元数据访问操作的性能，包括客户端元数据的预取、客户端文件路径查找的优化、MDS 元数据缓存的优化和日志的提交优化。

4 基于 ceph 系统元数据访问技术的优化实现

本章将对第三章的设计进行基于 Ceph 的实现，详细介绍了各优化的数据结构和处理流程。

4.1 客户端元数据预取

预取模块按功能细分成四个部分：目录项统计、生成预取请求、定期清理统计和网络通信，如图 4.1。

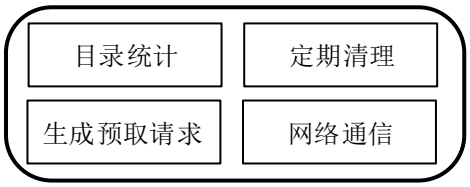


图 4.1 预取模块功能设计

在 Ceph 系统原来客户端的基础上，增加四个重要的类来实现预取模块：MetaPrefetch、DentryStatus、PrefetchRequest 和 PrefetchReply。其中，MetaPrefetch 用于管理预取模块，DentryStatus 用于统计目录项信息，PrefetchRequest 是元数据预取请求结构，PrefetchReply 是用来存取从 MDS 预取回来的元数据信息，它们的重要数据成员如表 4.1、4.2 和 4.3 所示。

表 4.1 MetaPrefetch 数据结构

成员名	成员类型	成员含义
client	Client *	预取模块所属的客户端
dentry_stat	DentryStatus *	用于统计目录项
prefetch_timer	SafeTimer	定时器，用于定时清理统计信息
pre_messenger	Messenger *	用于与 MDS 的网络通信

表 4.2 DentryStatus 数据结构

成员名	成员类型	成员含义
meta_prefetch	MetaPrefetch *	所属的预取模块
history	unordered_map<string,list<utime_t>>	目录项统计链表

dentry_map	unordered_map<string,list<Entry>>		同名目录项的统计
rdentry_map	map<Dentry*,pair<string,list<Entry *>::iterator>>		快速删除同名目录项的辅助链表
history_mutex	Mutex		目录项统计互斥锁
dentry_mutex	Mutex		同名目录项统计互斥锁
rdentry_mutex	Mutex		辅助链表互斥锁
max	int		目录项统计最大值
dname	String	Class Entry	内部类，用于同名目录项统计的结点
parent_name	String		

表 4.3 PrefetchRequest 数据结构

成员名	成员类型	成员含义
op	int	预取类型，主要为 CEPH_MDS_OP_PREFETCH_FILE 和 CEPH_MDS_OP_PREFETCH_DIR
prefetch_path	filepath	预取对象为文件时的预取路径
prefetch_path2	filepath	预取对象为子目录时的预取路径

表 4.4 PrefetchReply 数据结构

成员名	成员类型	成员含义
head	ceph_mds_reply_head	Ceph 消息通信的头部信息
parent_bl	bufferlist	用于存放预取父目录分片的元数据
sub_bl	bufferlist	用于存放预取最小子目录分片的元数据

其中 PrefetchRequest 是继承了原系统的 MetaRequest 类，额外增加了预取所需要的一些信息，PrefetchReply 也是如此，继承了 Message 类。

4.1.1 预取模块初始化

预取模块在使用前，要先对预取模块进行初始化，初始化的工作跟客户端的其他模块一样都是在客户端启动时进行，初始化主要是对 MetaPrefetch 下的 dentry_stat、prefetch_timer 和 pre_messenger 三个成员进行赋初值，为 dentry_stat 设置三个空队列待统计，prefetch_timer 设置定时间隔为 5s，pre_messenger 设置成指向 Client 模块中

Client 和 MDS 网络通信的对象指针。

4.1.2 预取数据流

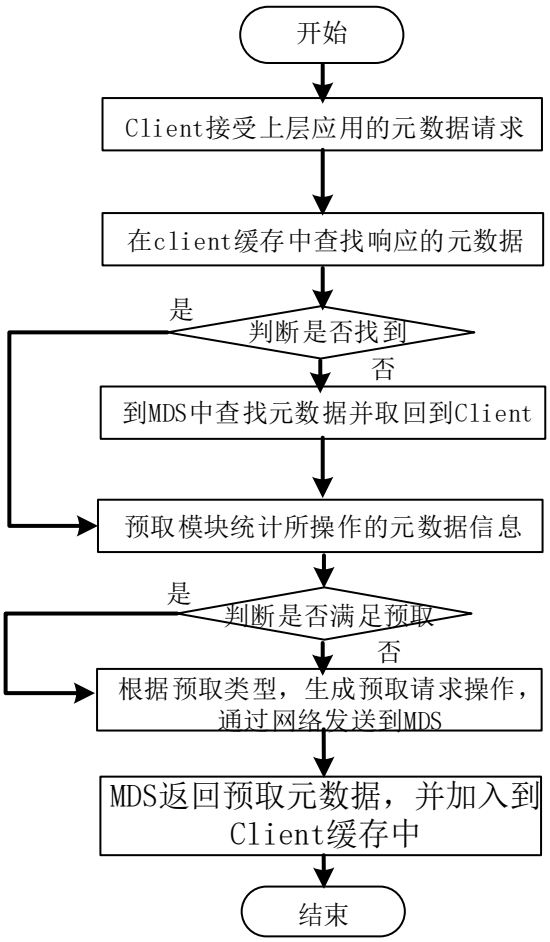


图 4.2 预取数据流图

图 4.2 为预取数据流图，描述了一个元数据操作请求在客户端与 MDS 之间的数据流程，在原系统的基础上增加了统计和预取条件的判断，当满足预取条件时需要增加预取请求操作。

4.1.3 目录统计的增删

从图 4.2 的数据流图可以看出目录统计是依赖于缓存的访问，所以目录项在缓存中的变化，统计系统中也要做响应的修改。主要分为统计的增加和删除。

统计的增加主要分两种情景：1.当缓存中新加入目录项；2.命中目录项，此时需

要在统计中加入时间戳。统计的删除也主要为两种情景：1.当缓存中目录项被淘汰时；2.处理 rmdir 等请求操作，这些都需要在统计系统中删除相应的统计队列。

(1) 目录统计增加

在处理 mknod 和 mkdir 这些操作时需要生成一个新的目录项，并添加到缓存中，这时需要在统计中新增加一个目录项链表。除此之外，在缓存没有命中，需到 MDS 中查询时，把 MDS 中取回的元数据加入到客户端缓存时，也需要新增一条链表。图 4.3 为统计增加的流程图。

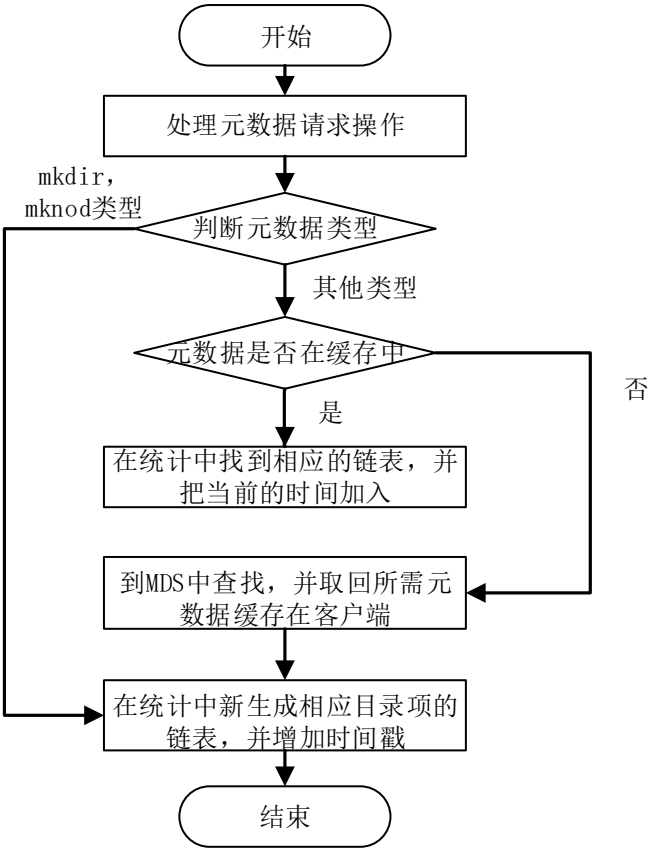


图 4.3 统计增加流程图

(2) 目录统计删除

在客户端缓存空间满的时候需要淘汰掉一些访问频率较少的目录项，由于淘汰的是访问频率少的，所以预取它对应的父目录概率也比较小，便可以在统计中删除相对应的链表。除此之外，在处理 rmdir 和 unlink 元数据操作也需要删除客户端缓存中的目录项信息，此时先观察要删除的目录项在统计中是否满足预取的要求，如果满足在删除链表之前进行预取。图 4.4 为统计删除流程图。

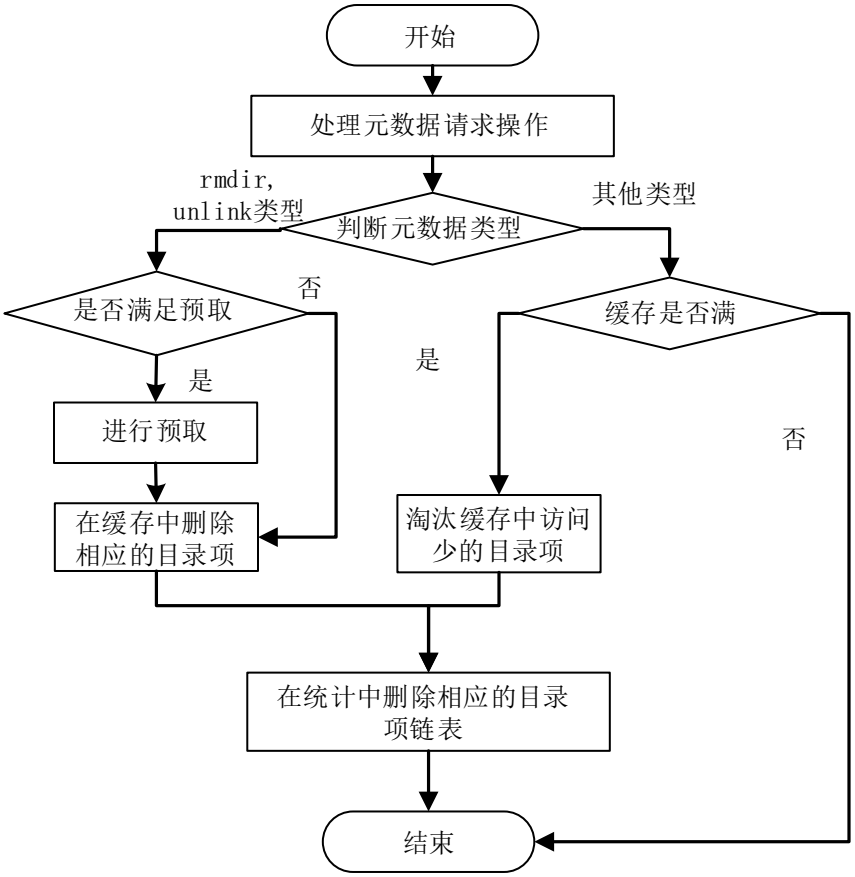


图 4.4 统计删除流程图

4.1.4 目录统计的定期清理

利用系统的定时器，每隔 5s 清空一下统计信息，删除每个链表中过去 5s 之内的时间戳，这样便于能实时统计 5s 内的统计信息。清理流程如图 4.5 所示。

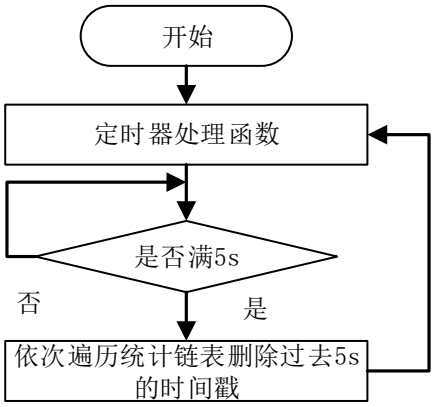


图 4.5 统计定时清理流程图

4.1.5 预取过程

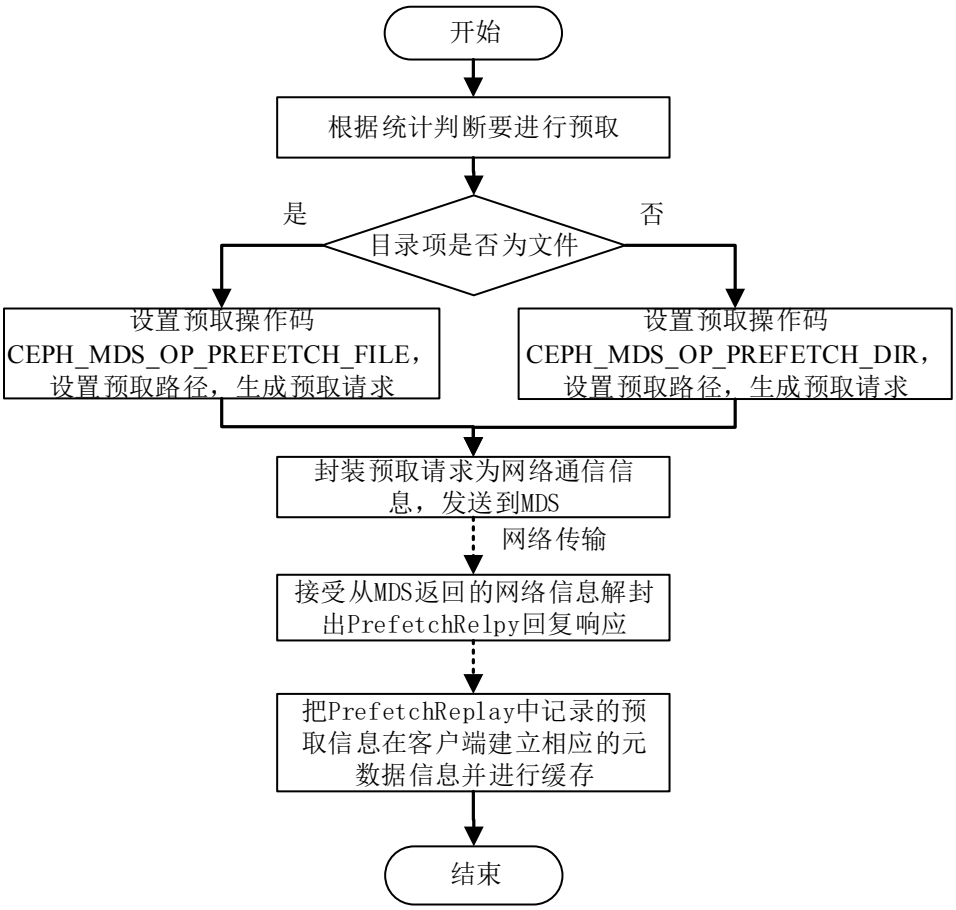


图 4.6　Client 预取流程图

利用统计系统，在访问到某一目录项时，判断其统计链表在过去 5s 之内访问到的时间戳个数(即链表的长度)是否大于 10，10 是通过多次测试试验出来比较合理的一个判断标准。大于就要进行预取，通过生成预取元数据请求 PrefetchRequest，设置预取路径，通过通信模块发送到 MDS，MDS 服务线程接受请求，解析请求信息中预取路径，再通过目录名的一个 hash 计算出所要预取的父目录，除此之外，如果是 CEPH_MDS_OP_PREFETCH_DIR 预取请求还要预取出下一级最小的目录分片。图 4.6 和 4.7 分别为 Client 和 MDS 预取的流程图。

对于 CEPH_MDS_OP_PREFETCH_DIR 操作码的预取类型，需要预取出子目录最小目录分片，因为系统中目录分片的管理是使用的 STL 的 map 容器，根据目录分片的知识可知，最小的目录分片是表示的位数最多，也就是标号最大，map 容器底层是红黑树实现，最大的 key 值可以通过 end()函数得到。

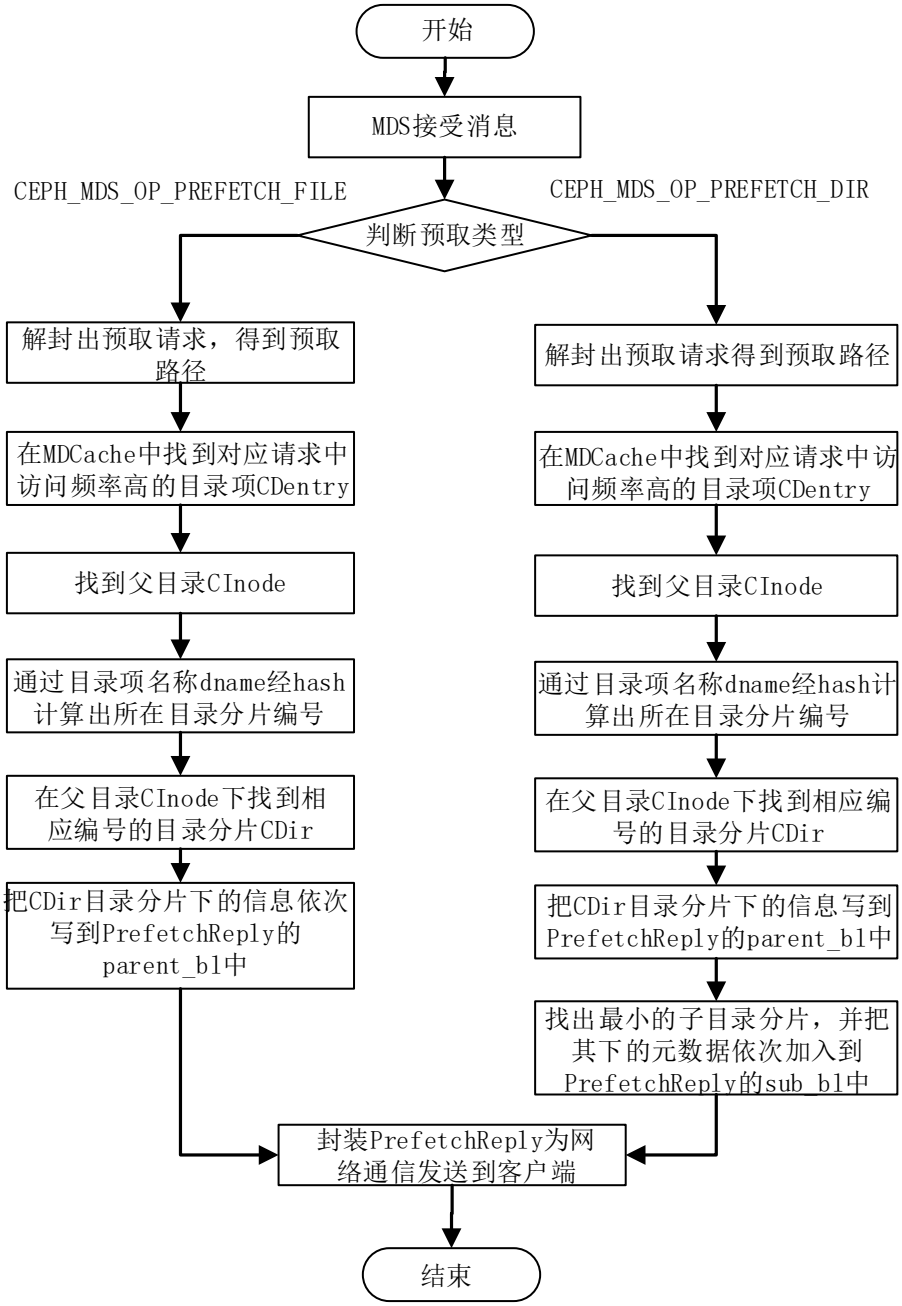


图 4.7　MDS 预取流程图

4.1.6 预取数据结构

从 MDS 中预取回来的元数据加入 PrefetchReply 对象中的 bufferlist 里返回给客户端。bufferlist 是 Ceph 中最重要的数据结构之一，负责管理内存。在把目录分片下的元数据预取回客户端时，这些元数据需要以一定的组织结构序列化在 bufferlist 中，

如图 4.8 所示。

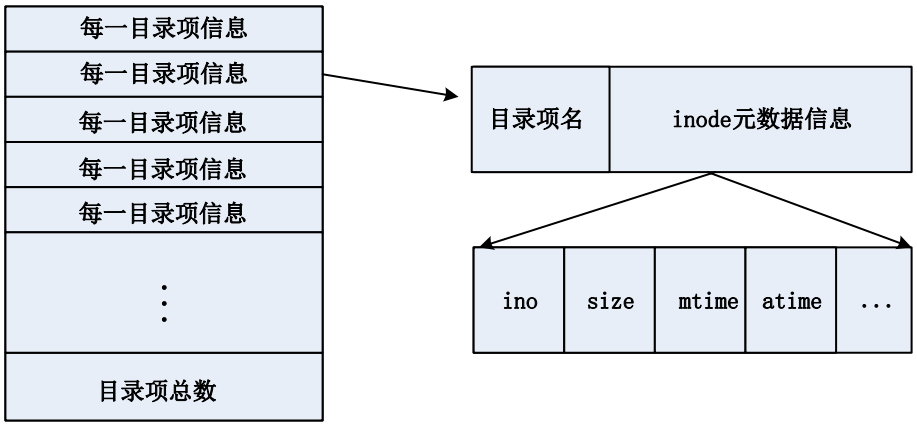


图 4.8　预取 bufferlist 结构图

由图 4.8 可知，预取 bufferlist 结果中以目录项为基本单位，每一单位记录目录项名和目录项对应的 inode 信息，并且整个 bufferlist 中以目录项名的字典顺序进行先后记录，其中 inode 信息即索引节点信息，包括 inode 标号、文件大小、最近修改时间、最近访问时间等等。在 bufferlist 最后还有一个预取目录项的总数信息。

4.2　路径查找优化

在 Ceph 原来系统的基础上，增加了两个重要的类来实现路径查找优化操作：OptPathRequest 和 OptPathReply。其中，OptPathRequest 是路径优化请求，OptPathReply 用于存放从 MDS 优化路径查询的结果，它们的重要数据成员如表 4.5 和 4.6 所示。

表 4.5 OptPathRequest 数据结构

成员名	成员类型	成员含义
op	int	表示优化操作码，CEPH_MDS_OP_OPTIMIZE_PATH_WALK
opt_path	filepath	优化查找时生成的相对路径

表 4.6 OptPathReply 数据结构

成员名	成员类型	成员含义
head	ceph_mds_reply_head	Ceph 消息通信的头部信息
optimize_bl	bufferlist	用于存放优化后相对路径上的元数据

其中 OptPathRequest 和 OptPathReply 分别继承了 MetaRequest 和 Message 类，同

时用于优化的同目录项名链表为上一节 DentryStatus 类 dentry_map,rdentry_map 主要是由于辅助同目录项名的删除。

4.2.1 优化路径查找过程

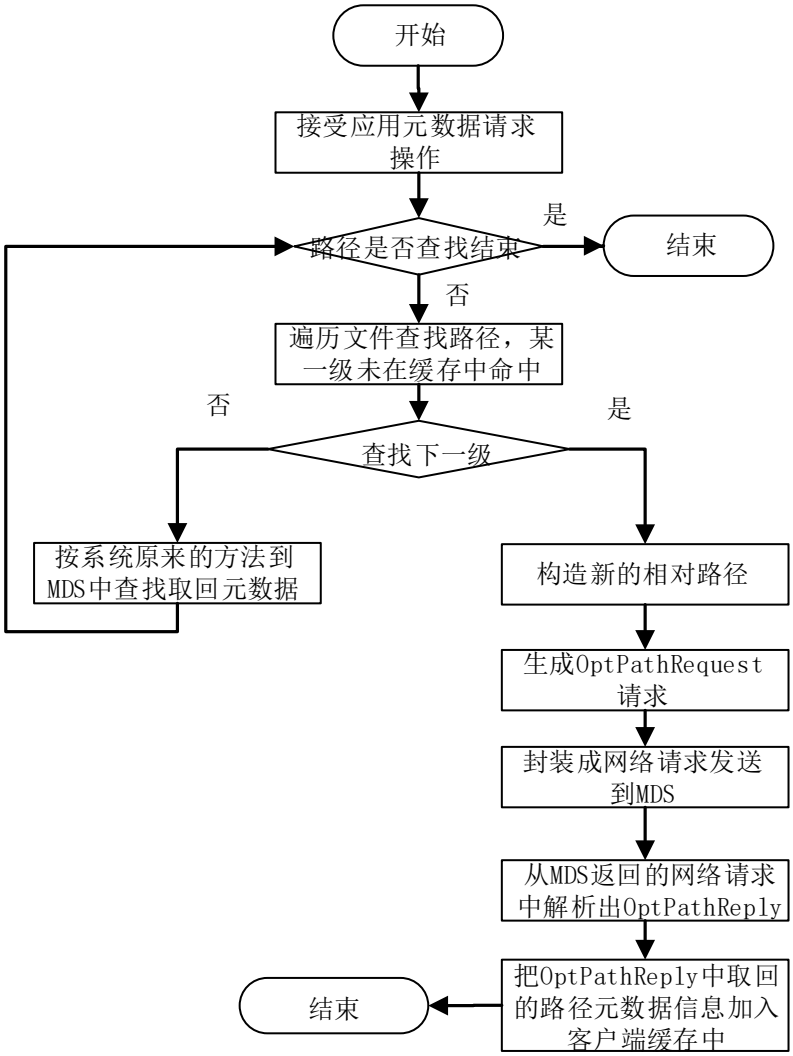


图 4.9　Client 优化路径查找流程图

根据设计，对文件查找路径使用连续两级目录未在客户端缓存中命中，则构造新的相对路径，生成 OptPathRequest 请求操作，通过网络发送到 MDS，MDS 在缓存 MDCache 经过查找后把遍历的新路径上元数据信息发送回客户端，并缓存在其 cache 中。图 4.9 和 4.10 分别表示 Client 和 MDS 路径查找优化的流程图。

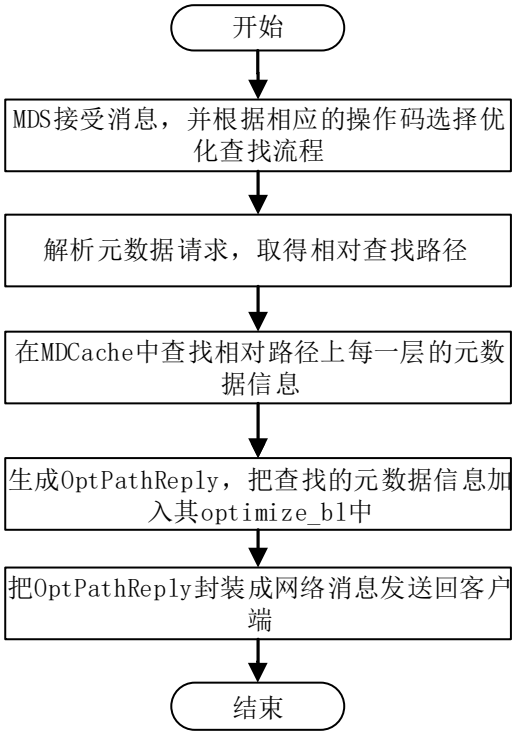


图 4.10　MDS 优化路径查找流程图

4.2.2 路径查找取回数据结构

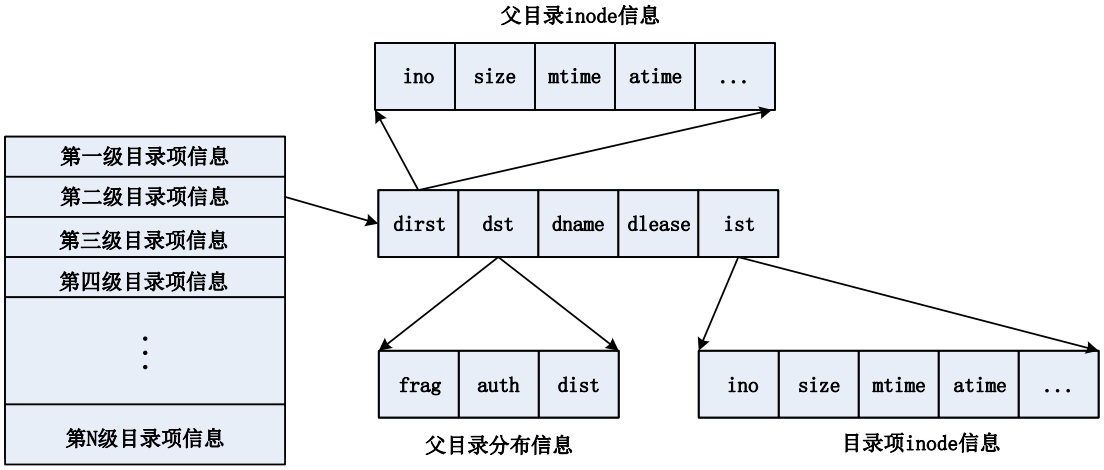


图 4.11　查询结果数据结构图

在优化查询之后，生成新的相对查找路径发送到 MDS 中。在 MDS 中需要把新相对路径的每一级目录项取回客户端，通过放入 OptPathReply 对象的 optimize_bl 这个 bufferlist 中。元数据服务器中路径的查找跟客户端一样，类似图 3.11，可以在每一级

目录项查询的时候，加入到 bufferlist 中。optimize_bl 中的数据组织如下图 4.10 所示。optimize_bl 这个存放结果的 bufferlist 由查询路径上每一级目录项查找信息组成。每一级目录项的信息由 5 个部分组成：父目录 inode 信息、父目录 dir 分布信息、目录项名、目录项与客户端的租赁以及目录项 inode 信息，其中父目录 dir 分布信息包括所属的目录分片序号、是否为属主元数据和副本元数据在集群中的分布情况。

4.2.3 同目录项名链表的增删

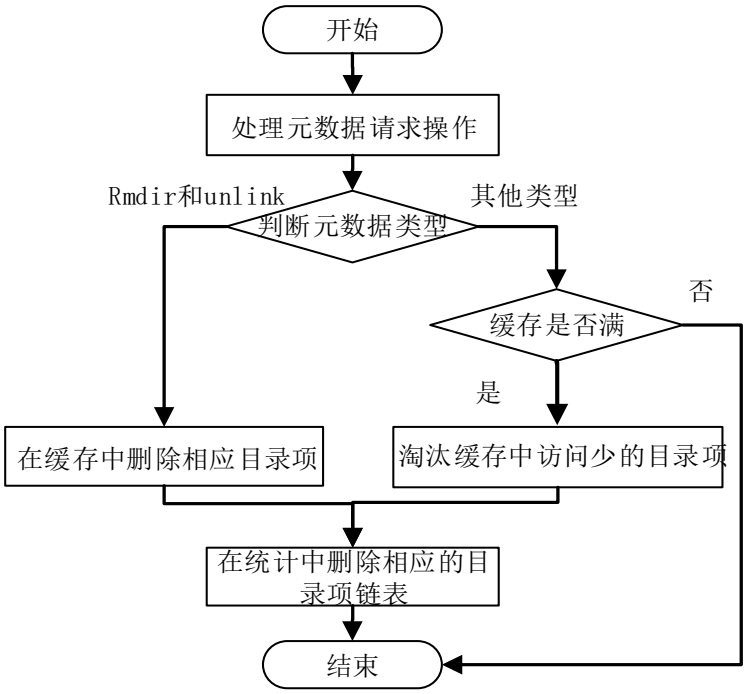


图 4.12　dentry_map 删除流程图

优化路径查找需要借助于同名目录项的统计链表，就是 DentryStatus 类结构中 dentry_map，其中以目录项名称作为 key 值。dentry_map 链表的插入删除时刻跟用于预取的 history 链表一样，插入也主要分为两种使用情景：(1)在处理 mknod 和 mkdir 元数据操作，(2)在一般的元数据操作没有命中缓存时，从 MDS 取回元数据信息；删除也主要分为两种使用情景：(1)在处理 rmdir 和 unlink 元数据操作，(2)在客户端缓存满时，淘汰掉访问频率少的目录项时。所以 dentry_map 链表插入删除流程跟预取的 history 差不多，就淘汰删除时不需要进行预取。图 4.12 为 dentry_map 删除流程图。

4.3 MDS 元数据缓存管理

为了提高 MDS 元数据缓存的命中，替换掉了原来管理元数据的简单 LRU，实现了 TGL(Two Group LRU)管理系统，相关的类结构如表 4.7、4.8 和 4.9 所示。

表 4.7 TGLObject 数据结构

成员名	成员类型	成员含义
lru_next	TGLObject *	指向链表的后一个
lru_prev	TGLObject *	指向链表的前一个
lru_pinned	bool	是否为脏的标记
tgl	TGL *	所属的 TGL
tgl_list	TGLRUList *	所属 TGL 管理中的链表

表 4.8 TGLRUList 数据结构

成员名	成员类型	成员含义
head	TGLObject *	指向链表头
tail	TGLObject *	指向链表尾
len	uint32_t	链表长度

表 4.9 TGL 数据结构

成员名	成员类型	成员含义	成员名	成员类型	成员含义
unpin_lru[3]	TGLRUList	干净链表	ratio_one	double	干净链表 1 所占比例
pin_lru[2]	TGLRUList	脏链表	ratio_two	double	干净链表 2 所占比例
lru_num	uint32_t	链表中元素个数	ratio_three	double	干净链表 3 所占比例
lru_num_unpin	uint32_t	干净链表元素个数	memory_use	double	内存使用率
lru_num_pin	uint32_t	脏链表元素个数	mds	MDS *	所属 MDS
threshold	uint32_t	链表 1 的门限值	lru_max	uint32_t	链表最大容量

MDS 中的元数据目录项 CDentry 原来继承于 LRUObject，现在改成继承于 TGLObject，这样便可以使用 TGL 对目录项进行管理。对于 CDentry 索引的管理，在 CDir 中用 map 和 Hashmap 分别对 CDentry 的指针进行管理，map 和 Hashmap 底层分别使用红黑树和 Hashtable 实现。

4.3.1 判断主副本元数据

在 Ceph 原系统中，MDS 里的 CDentry 继承于 MDSCacheObject 类，在此类中描述了元数据的主副本信息，但是没有设置如何判断主副本元数据的条件，这里增加一个 bool 类型的 is_replica 来记录是否是属主还是副本元数据。

Ceph 系统中副本元数据的由来有两种：

1. 直接生成一个副本的元数据信；

2. 把原来的属主元数据信息替换为副本元数据信息。

在生成副本元数据信息的两种方法中，都需要经过一个副本反序化函数(序列化和反序列化主要是用于网络传输)，所以可以在此函数中设置 is_replica 变量，如果是属主元数据 is_replica 值为 0，是副本元数据值则为 1，通过这样可以简单的判别出主副本元数据。

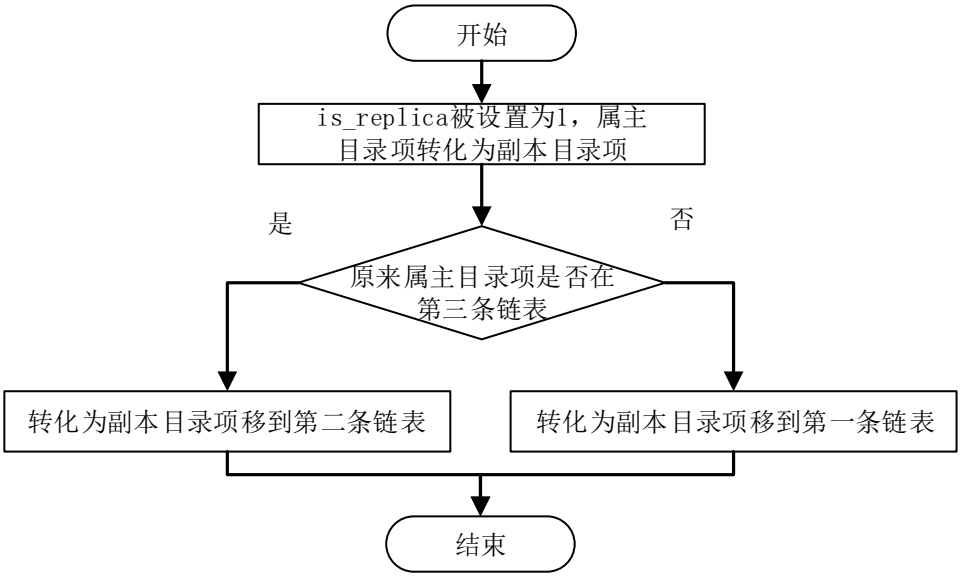


图 4.13 属主目录项转化为副本目录项

在加入干净链表时都是作为属主元数据加入，当 is_replica 值变为 1 时转化为了副本元数据，此时需要把副本元数据加入到副本管理的分组，但是需要根据原来属主元数据所在属主分组中的位置插入到不同的副本分组链表中。图 4.13 为变换流程图。

4.3.2 干净链表的加入

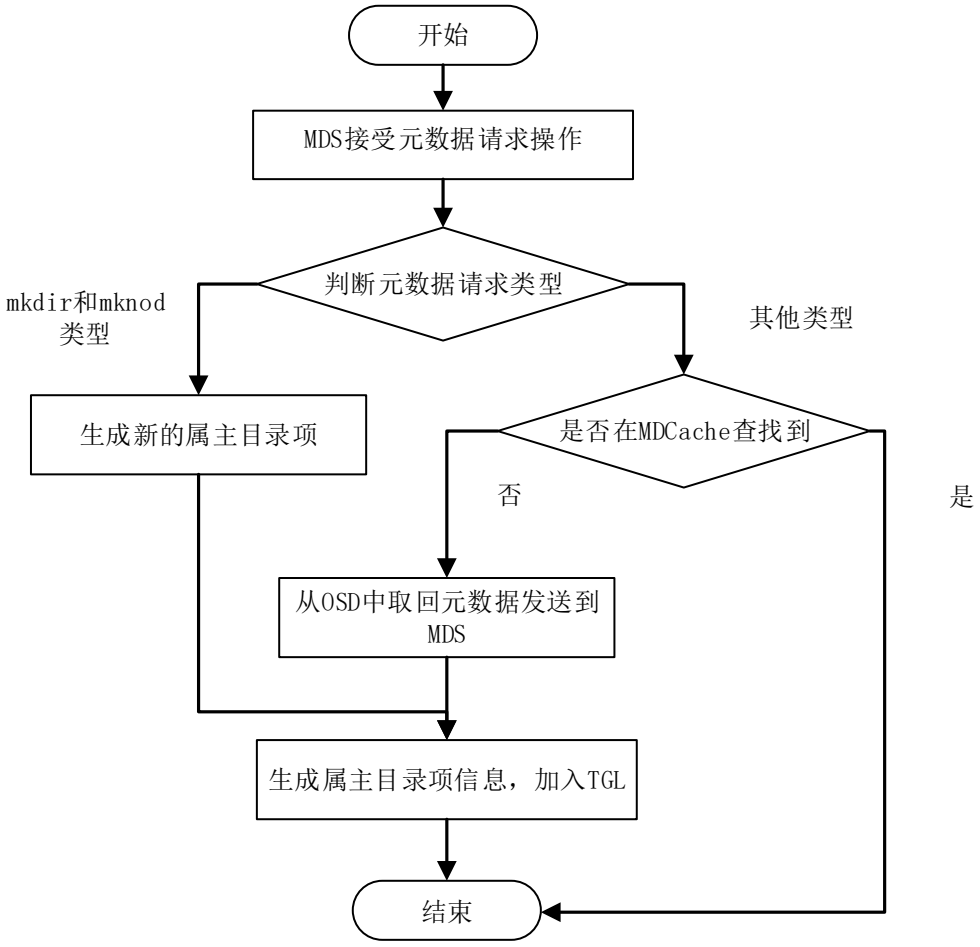


图 4.14 属主目录项的加入

在 Ceph 系统中只有属主目录项可以新加入到 TGL 管理系统中，并且插入到第二条链表。生成新的属主目录项主要有应用场景：(1)MDS 接受到 Client 的 mkdir 和 mknod 元数据请求，需要在本服务器上生成新的属主目录项；(2)在处理其他的元数据请求操作时，从 MDCache 中没有查找到相应的元数据，则就需要到 OSD 中取回所需的元数据，并在 MDS 中生成新的属主目录项加入到 TGL 中。如图 4.14 所示。

4.3.3 链表的升级和老化

TGL 中目录项的优先级的提高主要有以下几种使用情景：

1. 在通过 MDCache 查询文件路径时，对路径中每一级目录查找命中时需要提升

优先级；如果没有命中则首先从 OSD 中取回相关的元数据信息，并生成相关的元数据信息加入到 MDCache 中，并对访问没命中的那个目录项提升优先级。

2. 在 readdir 请求操作中，对要读取目录下的每个目录项提升优先级。

上面两种优先级的提升的使用情景对 TGL 中的干净目录项和脏目录项都适用。优先级提升的规则为：

1. 第一条干净链表中的副本目录项移到第二条链表的头部；
2. 第二条干净链表中的属主目录项移到第三条链表的头部；
3. 第二条干净链表中的副本目录项移到第二条链表的头部；
4. 第三条干净链表中的属主目录项移到第三条链表的头部；
5. 脏属主链表中的目录项移到本链表的头部。

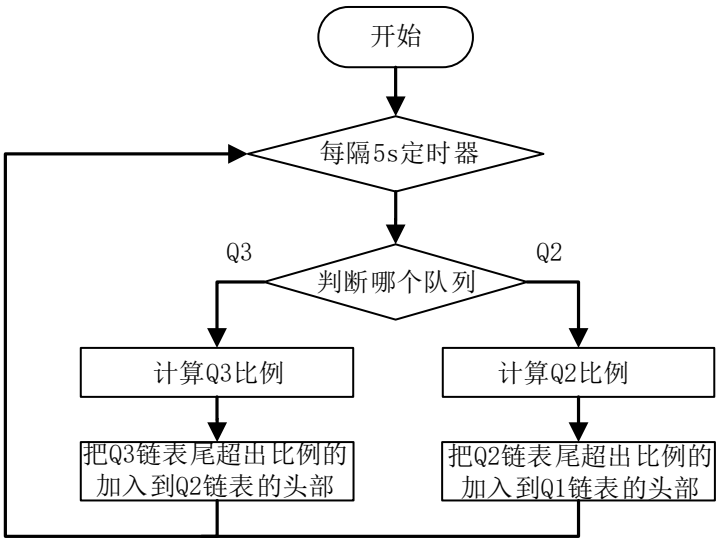


图 4.15 老化流程图

为了防止目录项的优先级只提升不下降带来的内存的污染，使用固定比例来实现目录项的老化，比如 TGL 中三条干净队列所占总容量的比例值为 0.2:0.4:0.4，一般是优先级高的链表所占的空间更大，这样可以使这些优先级高的目录项在内存中所在的时间长一些，以提高命中率。TGL 借助于 Ceph 系统在定时清理 MDS 时根据比例调整各干净链表的容量，以达到降低高优先级的目录项的目的。图 4.15 为老化流程图。

4.3.4 干净链表的淘汰

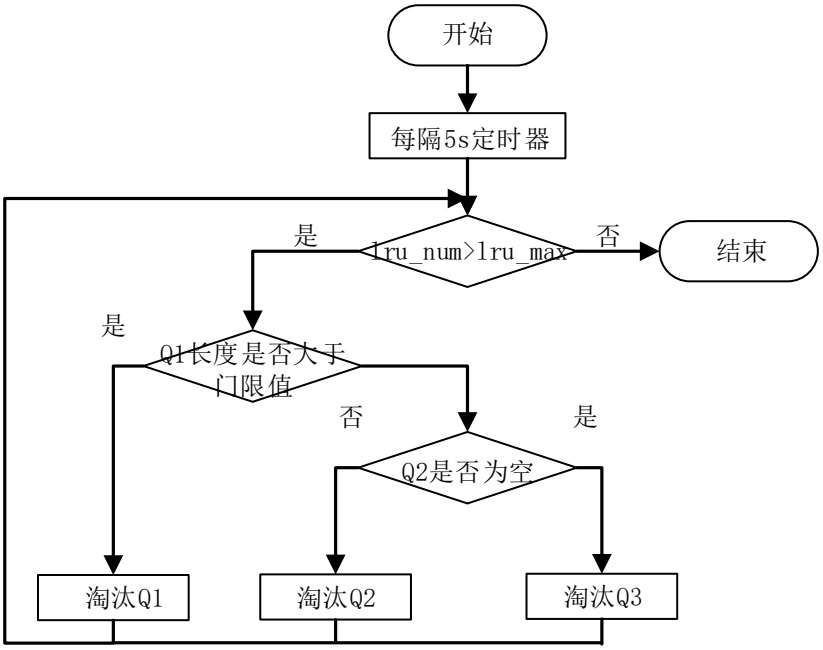


图 4.16 主动淘汰流程图

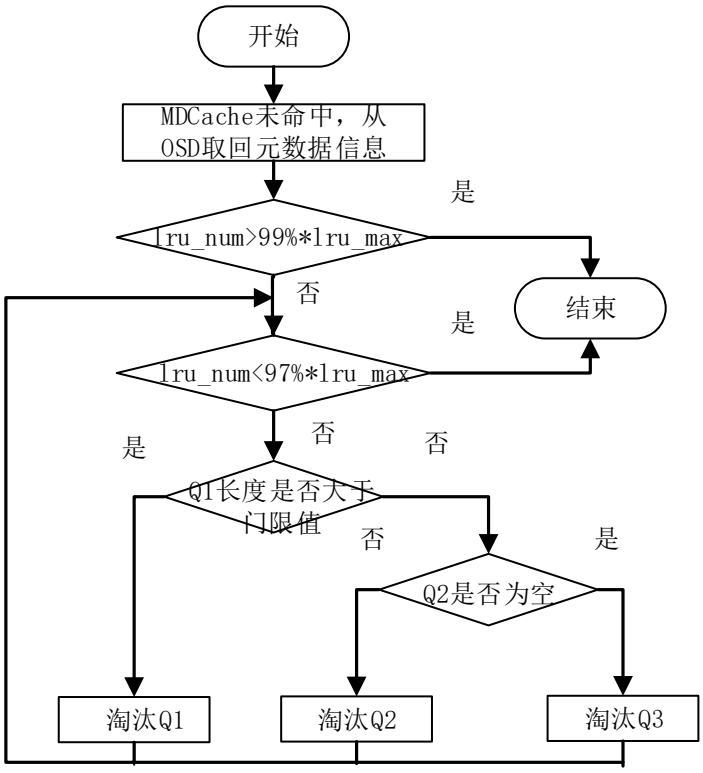


图 4.17 被动淘汰流程图

在 Ceph 系统中，由于系统的特点，淘汰目录项时，只能淘汰访问频率少，优先级比较低的干净目录项，根据设计的主动和被动两种淘汰使用情景，流程图分别如图 4.16 和 4.17 所示。

被动淘汰主要是给 TGL 容量预留 1%到 3%的空间，以免在从 OSD 中取回元数据时需要等待借助系统每隔 5s 的定期淘汰检测。淘汰时，按照优先级由低到高的顺序，也就是 Q1->Q2->Q3，同时给 Q1 留了一个最小的大小，比如 10，以防止刚刚进入的副本目录项被反复的淘汰。

4.3.5 干净目录项与脏目录项之间的转换

干净与脏目录项的转换需要日志模块 MDLog 的参与，元数据操作的可靠性正是通过日志来保证的。MDS 只使用内存来缓存元数据，而元数据永久性存储在 OSD 上，这样 MDS 和 OSD 之间的元数据可能会存在不一致性。不一致性主要是这样两种场景：1. 对已缓存在 MDCache 中的元数据在处理像 setattr,setxattrz 这样的元数据请求时，会对元数据进行修改；2. 在处理像 mkdir 和 mknod 这样的操作时，会在 MDCache 中生成 OSD 中没有元数据信息。

日志模块会在这些请求操作处理后生成日志事件，内部包含有元数据所做的修改或者是新生成的元数据即脏元数据(这里的元数据主要指 CInode)，这些脏元数据不会立即替换掉原来的而是放在一个备用队列里，需要待保存脏元数据的日志事件写到 OSD 之后，脏的元数据便可以替换掉原来的，也就是 MDS 中最新的元数据，这样就以日志的形式缓和了一致性的需求并保证了 MDS 的可靠性，此刻与 CInode 相关的元数据信息比如 CDentry 就会转变为脏的状态。以上便是干净目录项转化为脏目录项的一个数据流程。

而脏目录项转化为干净目录项也就是脏元数据的倒盘问题，日志模块中的日志事件在 MDS 内存和 OSD 的磁盘上都会存有一份，在日志写到 OSD 上后，MDS 内存中的日志便转化为过期的，这些过期的日志 MDS 会定期的清理，在清理之前，需要先把这些日志中记录的脏元数据信息以目录分片为最小单位更新到 OSD 上，更新完之后，之前标记为脏的目录项便转化为干净的。

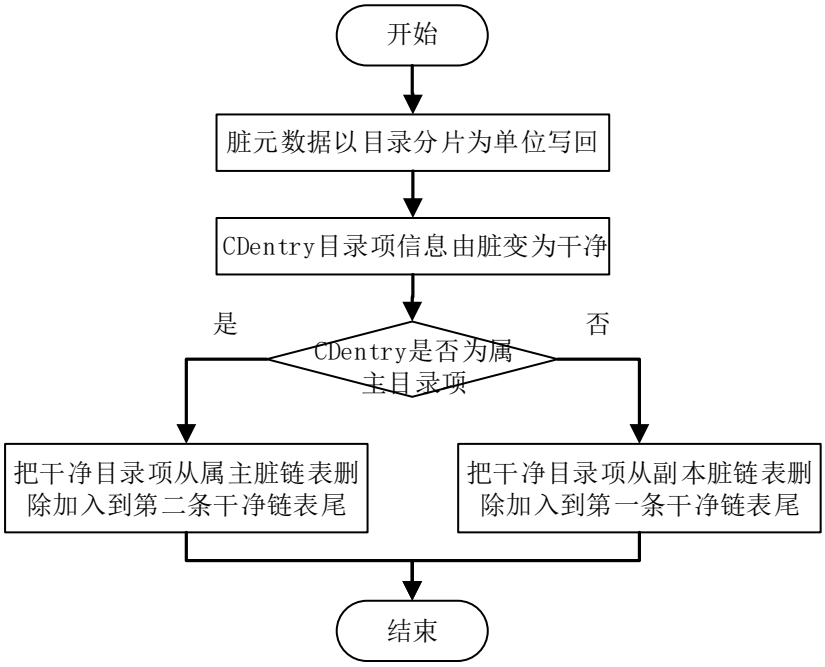


图 4.18 干净目录项转化为脏目录项流程图

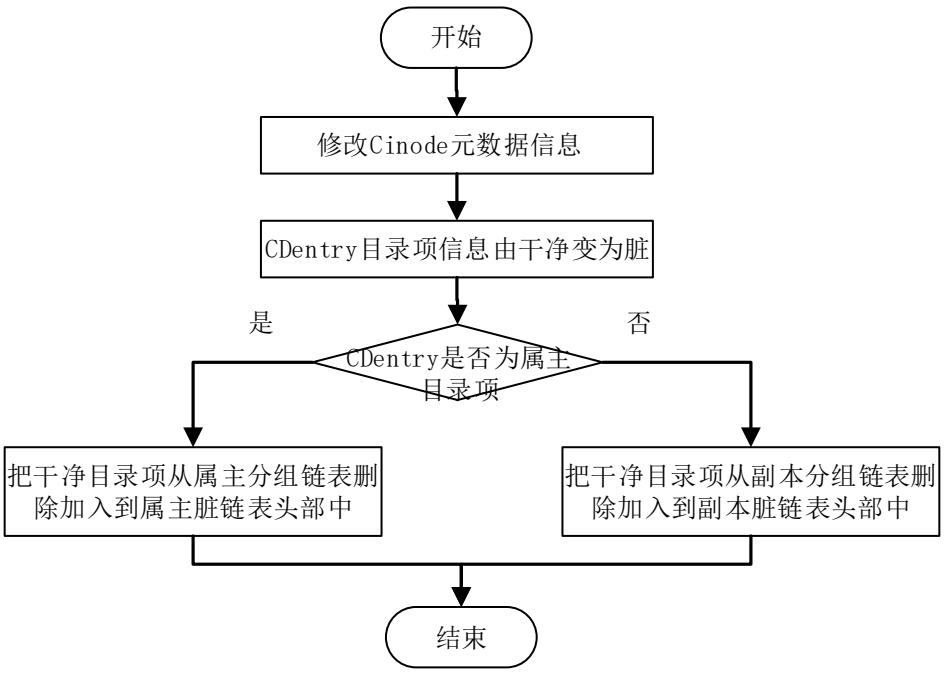


图 4.19 脏目录项转化为干净目录项流程图

干净目录项和脏目录项之间的转化只有属主目录项会进行，图 4.18 和 4.19 分别为干净目录项转化为脏目录项和脏目录项转化为干净目录项的流程图。目录项在由脏转变为干净时，可以看成是优先级最低的，所以可以加入到相应队列的末尾。

4.4 元数据操作流水线优化

在 Ceph 系统，把原来的元数据请求操作流程细分为三个阶段进行流水线操作，可以使 MDS 能处理更多的元数据，以提高元数据的操作速度，主要通过增加 PendingEvent、LogXio 和 TwoWriteBuf 三个重要的类来实现，PendingEvent 用于封装要提交的日志事件，LogXio 用以实现生产者消费者模式下日志处理链表，TwoWriteBuf 主要实现日志的双写缓存结构。详细数据结构如下表所示。

表 4.10 PendingEvent 数据结构

成员名	成员类型	成员含义
le	LogEvent *	日志事件
fin	Context *	日志写到 OSD 之后的回调函数
flush	bool	标志这个日志事件有没有写到 OSD 上

表 4.11 LogXio 数据结构

成员名	成员类型	成员含义
xio_events	map<uint64_t,list<PendingEvent>>	日志事件排队的工作队列
submit_mutex	Mutex	操作日志队列的互斥锁
submit_cond	Cond	待工作队列有日志事件时唤醒服务操作的条件变量
submit_thread	SubmitThread	提交服务线程

表 4.12 TwoWriteBuf 数据结构

成员名	成员类型	成员含义
write_buf[2]	bufferlist	两个 4M 大小的日志写缓存
flag[2]	uint64_t	两缓存是否满的标志
write_pos[2]	uint64_t	缓存中写入日志后的位置
flush_thread	FlushThread	缓存满刷新到 OSD 的服务线程

通过上述几个重要的类，把原来 Ceph 系统中 MDS 的元数据请求操作流程划分为三部分：元数据请求操作流程(不含日志的处理)、日志写到缓存的流程和缓存满日志刷新到 OSD 的流程。

4.4.1 元数据请求操作流程

通过把原 Ceph 系统中处理元数据服务的功能流程进行缩短，在生成日志提交到工作队列后，这个元数据请求操作就算完成，于是等待网络传输处理下一个元数据请求操作。这里对生成的日志事件需要封装成 PendingEvent 对象，主要是把日志事件和日志写到 OSD 上之后需要进行的回调函数(此回调函数用来把干净的元数据转化为脏标记)进行组合，然后把 PendingEvent 对象提交到工作队列 xio_events 中，xio_events 由 STL 中的 map 容器进行管理，其中每一个 pair 对是 Logsegment 的编号和一个日志事件链表，Logsegment 是日志的一个逻辑单元。图 4.20 缩短流程之后的元数据操作服务流程图。

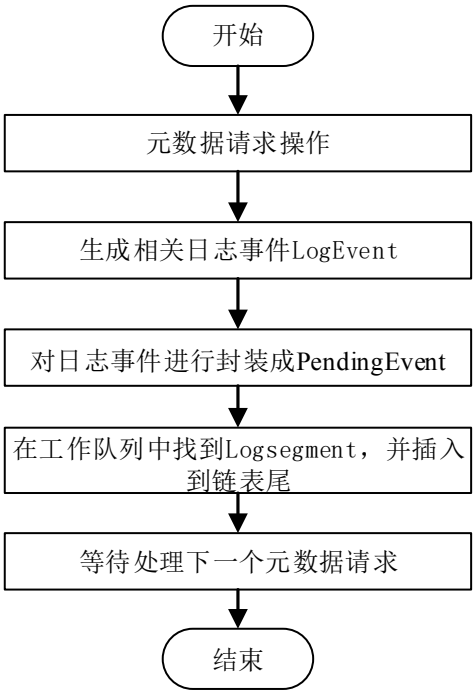


图 4.20 元数据操作流程图

4.4.2 日志写双缓存流程

日志事件 LogEvent 在写到工作队列 xio_events 后，submit_thread 提交服务线程便执行消费者的功能，不断的从中取出封装的 PendingEvent，submit_thread 作为一个服务线程需要一直提供服务，因此需要在日志模块初始化时一同与其初始化，初始化时主要是指定其所属的目录模块。Submit_thread 服务线程使用条件变量

submit_cond 来通知有日志到达工作队列，在从工作队列中取出日志时，是按日志生成的顺序依次取出，然后写到缓存中。使用两个缓存交替写日志，以基本保证 MDS 日志模块能不间断地写日志，这里 write_pos 标识缓存的偏移位置，flag 标识两个写缓存是否满，满时值为 1。缓存满是指大小达到 4M。流程如图 4.21 所示。

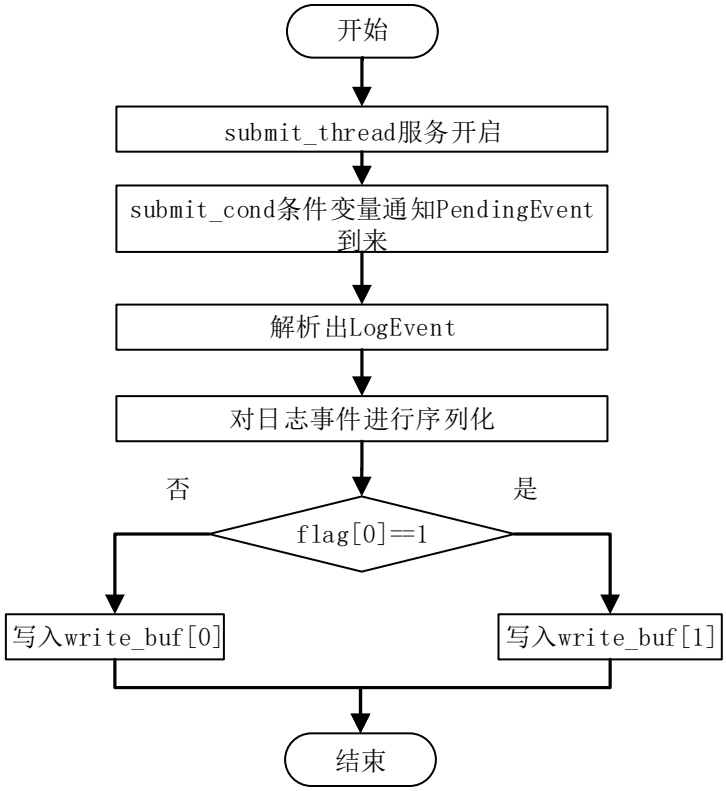


图 4.21 日志写到双缓存流程图

日志对其进行序列化的格式如图 4.22 所示，主要为日志事件的大小和日志事件内容。

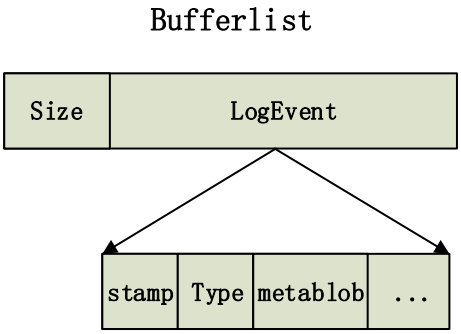


图 4.22 日志事件序列化格式

4.4.3 日志刷新到 OSD 流程

日志的刷新操作使用单独的 flush_thread 服务线程完成，flush_thread 跟 submit_thread 一样，都要提供不断的服务，所以其也是在 MDLog 日志模块初始化时进行启动，通过判别 flag 标识找到哪个缓存满了，便把满的日志通过 OSDC 刷新到 OSD 上。图 4.23 为刷新流程图。

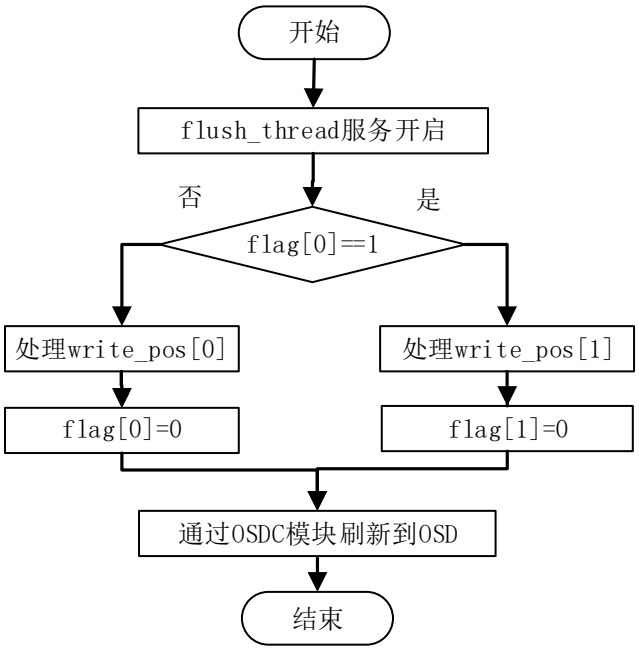


图 4.23 日志刷新流程图

4.5 本章小结

本章基于 Ceph 开源代码的文件系统应用，实现了第三章提出的四点优化设计，包括客户端元数据的缓存，客户端文件查找路径的优化，元数据服务器元数据分组缓存的改进以及元数据操作的流水线优化，给出了关键数据结构和数据流程操作。

5 系统测试与分析

本章主要是对第四章实现的改进优化进行系统性能的测试，主要是测试文件系统元数据的性能。接下来，首先介绍测试环境，然后介绍测试工具，最后展示测试数据并进行分析。

5.1 测试环境

实验测试使用 5 台服务器部署 Ceph 分布式文件系统，其中一台监控服务器节点(MON)，一台元数据服务器节点(MDS)，其余三台为对象存储设备(OSD)，同时使用一个交换机搭建一个局域网来完成它们之间的通信。图 5.1 为实验测试平台部署图。

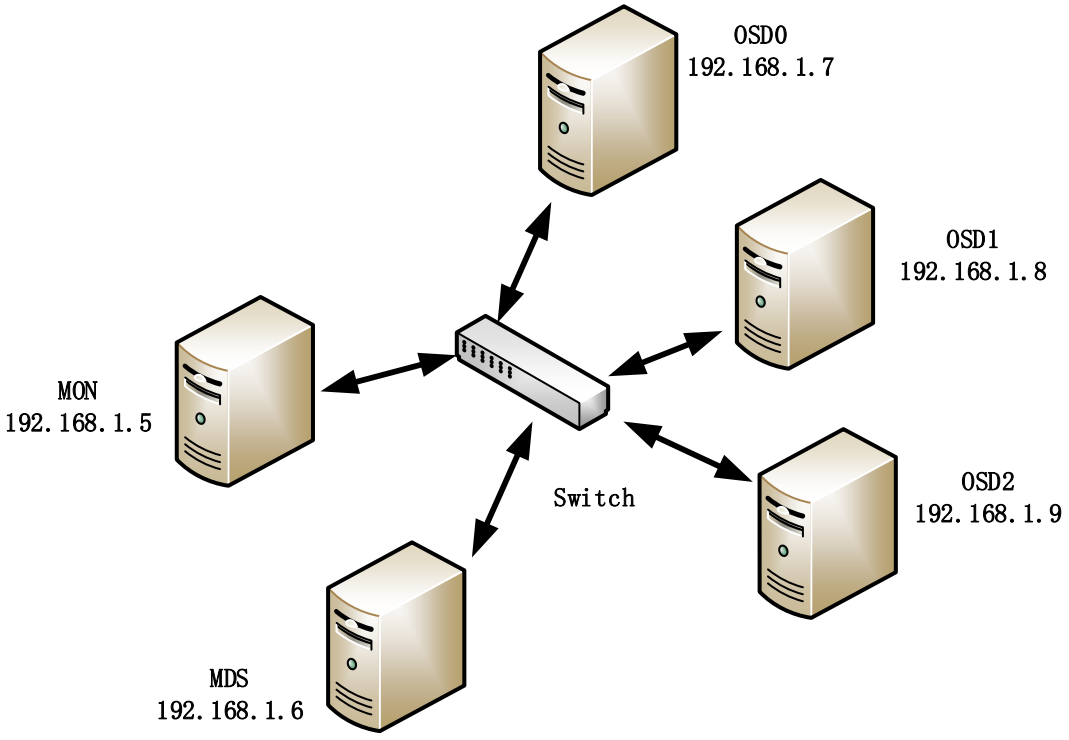


图 5.1 实验平台部署图

使用交换机构造局域网主要是防止外界来路不明的请求对环境的影响，实验中 CephFS 客户端主要挂载在 MON 服务器上，并且使用用户态进行挂载，通过测试这个挂载的用户态客户端的性能就是对整个 Ceph 文件系统的测试。实验所用的服务器

配置如下表 5.1 所示。

表 5.1 服务器配置

操作系统	Ubuntu 14.04 x86_64
内核	Linux 3.13.0-24
CPU	Intel(R) Xeon(R) E5606 @2.13GHZ
内存	16GB
网卡	千兆网卡
系统盘	1.0TB
数据盘	1.0TB

本实验中 Ceph 的代码版本使用的是 0.80.5。

5.2 性能测试与结果分析

为了测试 Ceph 文件系统的性能，主要采用 vdbench 测试软件，并且主要测试的性能指标是文件操作的 I/O 延时和处理速度。

vdbench 是一个 I/O 工作负载生成器，可以根据用户的需求生成自定义的负载，用于验证数据完整性以及度量直接附加和网络连接存储的性能，可以测试文件系统的性能，也可以用来测试裸磁盘文件。vdbench 可以测试的负载操作有 mkdir、rmdir、create、delete、open、close、read、write、getattr 和 setattr。对文件系统的测试，可以测试创建删除目录和文件等元数据操作的性能，也可以测试文件读写 IO 的性能。

vdbench 需要在测试前构建一个文件测试环境，主要由 depth、width 和 files 三个变量生成要测试的文件目录树，其中 depth 表示目录树的深度，width 表示每个目录包含的子目录树，files 表示叶子目录包含的文件数。为了测试 Ceph 文件系统的性能，主要测试创建删除目录(mkdir_rmdir)、创建删除文件(create_delete)、文件读写(read_write)三种负载，三种负载的都使用一个线程，传输数据的单位为 4k，文件大小为 8k，文件打开以 o_direct 方式，不使用缓冲区。

5.2.1 四种优化访问延时测试

表 5.2 所示为这三种负载的文件系统测试环境。基于上面文件系统环境的不同负

载的 IO 延时如图 5.2 所示。

表 5.2 参数配置

	mkdir_rmdir	create_delete	read_write
depth	1	1	1
width	10000	10	10
files	1	1000	500

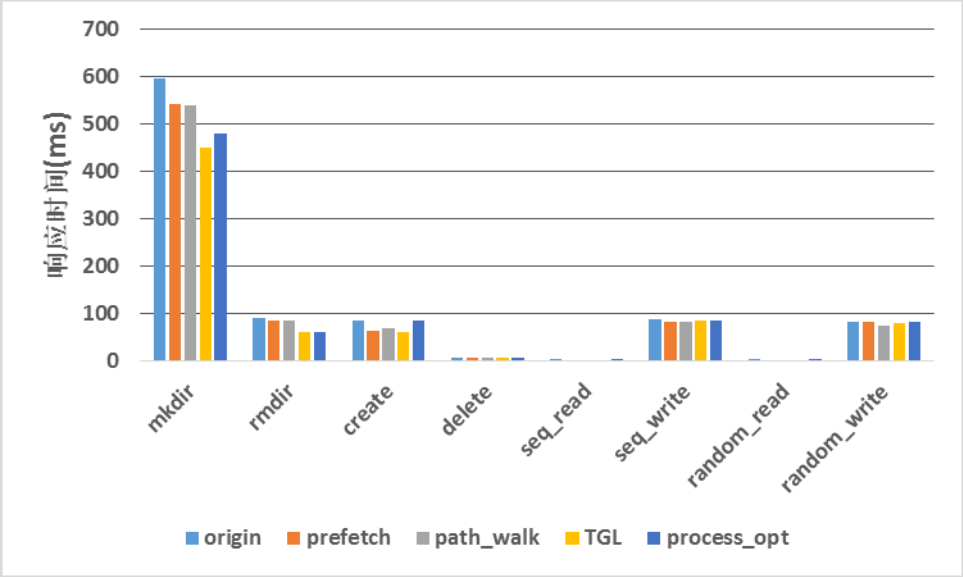


图 5.2 不同负载的 IO 响应时间

从图 5.2 所示，在所有的负载操作中创建目录比较耗时，这主要是因为在用户态下创建目录需要创建 CInode、CDentry 和 CDir 三种元数据信息，而创建文件只需要创建 CDentry 和 CInode，此外还要把 CDir 加入到相应的目录分片树中。写的延迟比读的延时高很多，主要是在 OSD 之间要写三副本，需要消耗很多网络延迟。删除文件比创建文件响应时间少，因为删除文件不需要真正的删除文件的元数据信息 CInode，只是简单的断开 CDentry 目录项和 CInode 索引节点之间的硬链接。从上图所示，四种优化都一定程度地减少了负载响应延时，在创建删除目录和文件这些纯粹的元数据操作上响应时间减少比较明显；而在读写操作上响应时间减少比较少，因为读写操作不是纯粹的元数据操作，读写操作中查找打开文件是属于元数据操作，之后的文件读写属于文件数据 I/O 操作，而且文件数据操作延时在整个读写操作中所占比例高，所以在读写操作中路径查找优化的性能比较好。

相对于原来的 Ceph 系统，元数据预取、路径优化、TGL 和 process_opt 在创建

目录操作时间分别减少 9.1%、9.4%、24.3%、19.6%，删除目录操作响应时间分别减少 5.1%、4.9%、31.9%、32.2%，创建文件操作响应时间分别减少 23.9%、16.6%、26.9%、0.1%。这表明，在元数据服务器优化的结果比在客户端优化的好。

5.2.2 选取预取门限值

为了选取预取热度规则的门限值大小，这里使用控制变量法，分别对 mkdir、rmdir、create 和 delete 这四组负载进行测试，控制门限值的大小为 5 到 11 的整数，结果如图 5.3 所示。

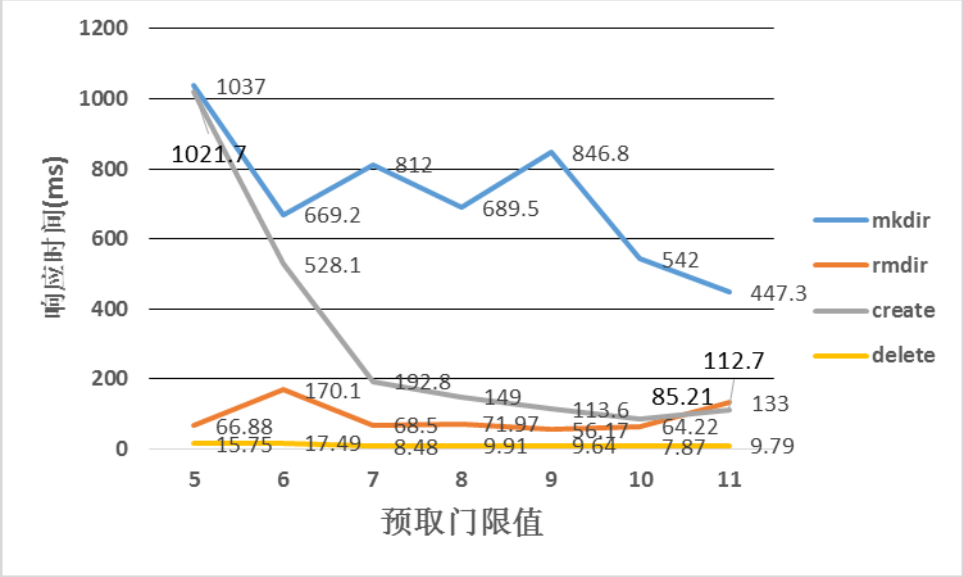


图 5.3 不同门限值的各负载响应时间

从图 5.3 中可以看出在不同预取门限值大小的情况下，平均响应时间最小的门限值为 10。对于门限值小于 10 时，很多目录项统计次数都能达到，于是客户端会发起过多的预取操作，而客户端的缓存空间是有限的，缓存中的目录项会出现经常性的换进换出，因此未能达到预取的效果。在 vdbench 的测试环境下，统计次数能超过门限值 10 以上的目录项不多，所以能启动的预取操作偏少，也未能达到预取的效果。

5.2.3 网络传输次数测试

为了测试路径优化操作后的性能，最直观的测试是通过普通文件的查找，考察客户端到 MDS 需要发起的网络交互次数。由于 vdbench 不能进行这种测试，所以通

过 libcephfs 提供的 API 文件系统的接口写测试应用程序，并在客户端发送消息到 MDS 的函数中增加统计计数。测试应用程序首先在 ceph 中建立大量的目录和文件，还要进行嵌套的建立子目录和文件，然后随机生成文件查询的目录。主要测试了在 60s、100s 和 300s 查询时间内的网络传输次数，如下表 5.3 所示

表 5.3 网络传输次数

	60s	100s	300s
origin	4151 次	7912 次	19853 次
Path_walk	3628 次	6939 次	17344 次

从上表 5.3 可以看出，路径优化后减少了网络传输次数，多级目录项未命中只需要进行一次网络传输就可以从 MDS 中取回目录项等元数据信息，网络传输次数减少 12.3%左右。

5.2.4 MDS 缓存测试

1. 由表 5.2 可知，vdbench 使用的测试环境中文件和目录都为 10000，但是 Ceph 系统中 mds 的缓存默认大小为 100000 个目录项，因此测试的目录项都能缓存在 MDS 缓存中，很少出现缓存空间不足需要淘汰目录项，这样不能体现出 TGL 的性能，为了更有效的研究 TGL，通过修改 Ceph 配置文件中的 MDS 缓存大小为 1000、5000 和 10000 来测 TGL 缓存算法的性能，测试结果如图 5.4-5.6 所示。

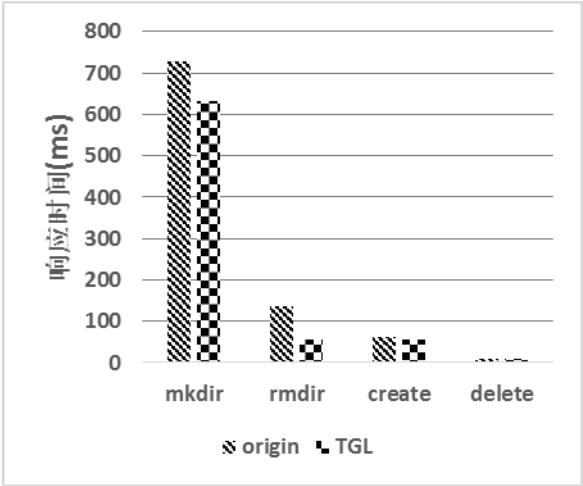


图 5.4 MDS 缓存大小为 1000

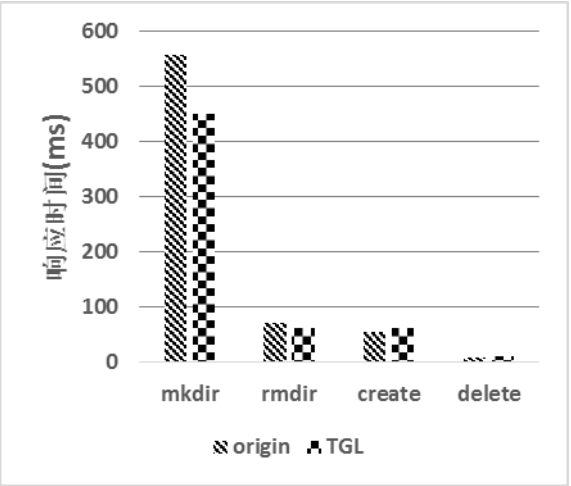


图 5.5 MDS 缓存大小为 5000

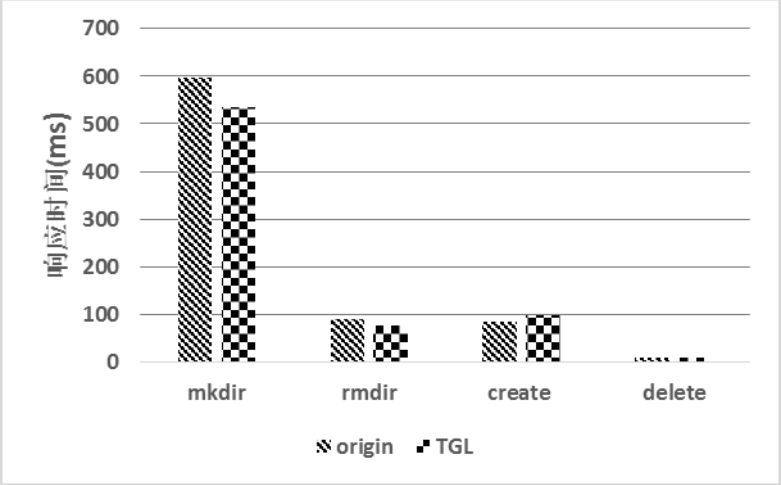


图 5.6　MDS 缓存大小为 10000

从图 5.4、图 5.5 和图 5.6 可以看出，在 mkdir 和 rmdir 两种负载下，TGL 都减少了响应时间，并且缓存为 1000 和 5000 时，性能更好响应时间分别下降了 13.1%、59.3% 和 18.9%、12.9%。在 create 和 delete 两种负载操作时，缓存大小为 10000 时响应延时有所上升，缓存为 5000 时也有所上升，而缓存为 1000 时响应延时下降了 7.9%。

2. TGL 中使用三条链表来管理干净队列，并且通过固定比例来进行老化，对于固定比例设置的不同，系统性能也有不同的表现。根据经验优先级高的队列尽量保持的空间多些，主要设置了三种固定比例，按干净队列优先级由低到高分别为 0.2:0.3:0.5、0.3:0.3:0.4 和 0.2:0.4:0.4，三种比例下测试目录和文件操作性能如下表 5.4。

表 5.4 三种固定比例性能

	mkdir	rmdir	create	delete	open	close
0.2:0.3:0.5	497.4ms	68.35ms	92.15ms	8.09ms	3.78ms	0.05ms
0.3:0.3:0.4	485.3ms	71ms	88.61ms	8.72ms	3.51ms	0.05ms
0.2:0.4:0.4	497.5ms	70.22ms	83.82ms	7.86ms	3.08ms	0.05ms

从上表中可以看出 0.2:0.4:0.4 这种比例下系统的性能更好，因为 TGL 干净队列的中间一条是混合链表，存储着属主目录项和副本目录项，虽然其优先级没有第三条全存放属主目录项的高，但是由于存储的内容多，所以为其分配和第三条链表一样的比例时，系统的性能更好一些。

5.2.5 MDS 处理速度测试

对于 MDS 请求处理过程的优化，通过测试一段时间内 MDS 请求处理个数来更直观地观察此优化的性能。由于 vdbench 没有这种测试效果，所以跟测试网络传输次数一样通过 libcephfs 中提供的文件 POSIX 接口函数编写小型应用测试程序进行测试，在 MDS 源代码的接受请求处增加统计计数。

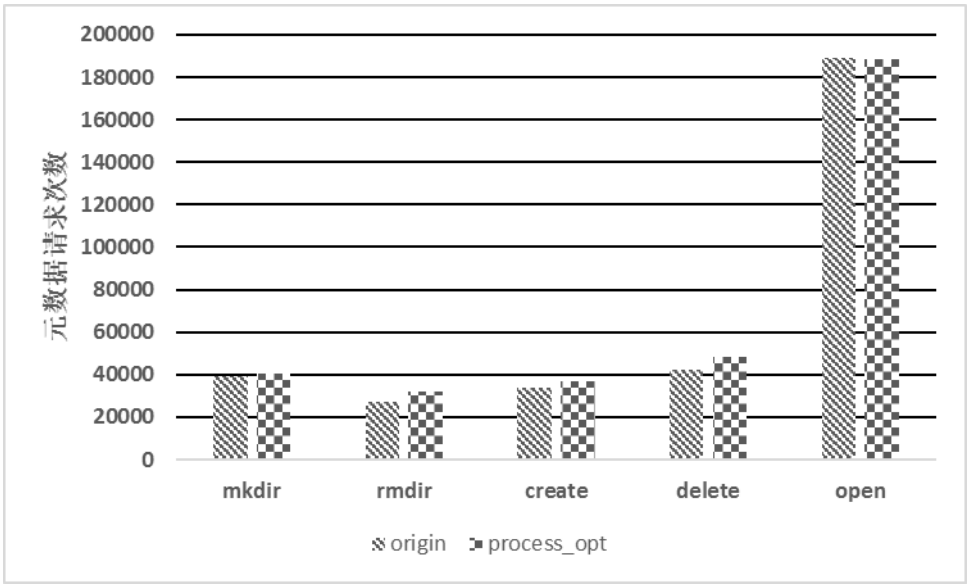


图 5.7　元数据请求速度

图 5.7 为元数据操作流水线优化后测试的请求处理速度图，主要测试的是 5 分钟之内 mkdir、rmdir、create、delete 和 open 这 5 中操作的处理个数，从图中可以看出，前四种元数据操作的处理速度都有提升，分别提升了 4.09%、16.1%、9.81%、13.11%，因为前四种操作都对元数据进行了修改，于是会有相关的日志生成。但是 open 操作由于没有修改元数据，所以不需要生成日志，所以优化策略对 open 操作没有影响。

5.3　本章小结

介绍测试工具 vdbench，并且通过它测试了客户端和元数据服务器的四种优化改进以及 Ceph 原文件系统的性能。测试结果表明，相对于原来的 Ceph 文件系统，四种优化都提高了元数据访问的性能，并且元数据服务器端的优化相对于客户端的提高的性能更明显。

6 总结与展望

在当今大数据时代，人类每天生产生活产生大量的数据信息，需要借助于存储系统对其进行存储、管理与挖掘。其中，分布式文件系统在存储系统中占据着重要的地位。在分布式文件系统中，元数据本身的特性表明其对整个文件系统的性能有着重要的影响作用，所以研究如何提高元数据访问的性能在当前计算机存储领域有着重要的意义。

6.1 全文总结

基于开源的 Ceph 分布式文件系统，以元数据的两级缓存：客户端和元数据服务器为研究对象，就如何提升元数据访问的性能展开研究工作，主要所做的工作有如下几个方面：

1. 研究了 Ceph 的元数据管理系统和元数据访问流程。在 Ceph 文件系统中，元数据服务器集群负责维护系统的全局命名空间，其中存储的元数据是系统中较热的元数据，也是对象存储集群的系统级元数据缓存；客户端负责对应用操作的访问提供接口，其中存储的元数据是对元数据服务器集群中较热元数据的缓存。通过对这两级缓存元数据访问流程的研究与分析找出可以优化的四个方面。

2. 研究了客户端访问文件的特性，根据客户端中元数据缓存访问增加目录项统计系统，通过统计系统筛选出较热的目录项元数据信息，并且通过目录分片技术，从元数据服务器预取出较热目录项所在的目录分片放入客户端的缓存中，使接下来的元数据操作能更多的在缓存中命中，减少客户端缓存未命中到元数据服务器取回的网络传输延时。

3. 研究了客户端文件查找的流程，基于目录项统计系统增加同名目录项队列，实现连续两级目录未命中情况下进行一次网络传输请求，来减少原系统中多次到元数据服务器进行元数据请求的网络延时。

4. 研究了元数据服务器端元数据缓存管理子系统。为了平衡元数据服务器的请求负载，对于使用动态子树分区的元数据集群设置了属主元数据和副本元数据，根据此特性对目录项元数据采用分组管理，并使整个缓存分为三个优先级，又根据元

数据倒盘更新的特性，分别对干净目录项和脏目录项进行管理。此外，根据元数据服务器内存的使用情况，在进行淘汰时使用动态指标，以使元数据缓存的管理更高效。

5. 研究了元数据服务器的元数据访问流程。元数据服务器中的元数据访问操作需要借助于日志子系统来保证操作的可靠性和数据的一致性。日志系统参与使每个元数据操作都要先写日志事件，对于此通过把元数据请求流程细化为三个处理阶段，并增设两个日志写缓存，以流水线的模式加快元数据的处理。

6. 通过使用 vdbench 测试工具，对所做的四种优化修改以及原系统进行元数据性能的测试和分析。

6.2 展望

虽然测试的结果表明，所做的四种优化在一定程度上改进了元数据访问的性能，但仍然存在一些需要进一步优化和改进的地方，提出以下几点需要进一步研究：

1. 在元数据服务器端改进缓存管理算法时，在淘汰访问较少目录项时使用动态的指标来实现根据负载进行调整，实现中使用的是服务器内存使用情况作为负载的依据，而实际的负载需要考虑更多的因素，可以进一步研究负载的定量表示。

2. 从测试的结果可以看出，各个元数据操作对系统的性能影响是不一样，所以可以进一步研究对元数据操作进行分类优化，找出各种元数据操作相应的优化点。

3. 客户端元数据的预取只考虑了元数据的空间局部性原理，没有对元数据之间时间局部性进行研究，可以结合考虑这两方面局部性的原理再分析元数据访问的特点，找出更科学的预取条件。

4. 实验的四种优化设计都是增加一些内存开销以空间换时间，如果在一些服务器内存资源比较紧张的情况下，可能优化的性能不是很好。

致谢

时间如梭，学生生活马上就要结束。回首这三年的时光，期间的点点滴滴还历历在目，期间有很多人在学习和生活中给予我指引和帮助，在这里由衷地向他们表示感谢。

首先要感谢我的导师谭志虎副教授。非常荣幸能在谭老师的指导和关心下度过三年的研究生生活，谭老师知识渊博、教学认真负责、为人和善、治学严谨的态度深深的鼓励和影响了我。在研究生刚开学就指导我如何看国外的优秀论文，对专业基础知识如何学习与积累，还会定期的跟我交流与讨论，做毕业设计期间，提供了一些建设性的意见，并给出了如何写好论文的一些建议。在此，非常感谢谭老师对我付出的关爱和心血。

感谢万继光老师在去年让我接手华为的一个科研项目，在这个项目中，我学习到了很多知识，锻炼了编程能力，历练了项目管理能力，也找到了自己感兴趣的研究方向，本论文的课题也来源于该科研项目。还要感谢谢长生、曹强、吴非等老师在研究生第一年的课堂学习上传授了我很多知识，同时也为我们实验室营造良好的科研学习环境，让我们像一个大家庭一样能够快乐的学习生活。

然后，感谢实验室的罗旦、赵楠楠、瞿晓阳、吴畏、汪志明、门勇、程龙等师兄师姐。在平时科研时，遇到困难，他们总能耐心的帮助和指导我，使我一点点的积累技术基础。同时在我毕业论文设计中，他们也给了我很多理论层面的指导意见。还要感谢李大平、徐鹏、姚婷、张和泉、刘丽琼和张钰彪在论文书写方面给我的建议与修改。

感谢李远超、刘庆宾、陈诗杨、周晨曦、王珣、刘鑫伟、李思思、夏倩、刘洋等同学，在三年的研究生生活中，我们一起学习，一起科研，一起娱乐，让我度过了一个难忘的大学研究生生活。

最后，我要感谢我的父母，是他们孜孜不倦的教诲和关爱养育了我，同时支持我本科毕业后继续读研深造，在三年的研究生生活中给予我精神和物质上的支持，使我有信心面对生活中的种种困难。此外还要感谢我的女朋友，是她在背后默默的支持我，鼓励我，让我变的更加自信，使我拥有坚强的意志，优秀的品德和端正的态度，这些使我授用终身。

参考文献

[1] Roselli D S, Lorch J R, Anderson T E. A Comparison of File System Workloads. USENIX annual technical conference, general track. 2000: 41-54

[2] Goda, Kazuo: Direct Attached Storage. Encyclopedia of Database Systems. Springer US, 2009: 847-847

[3] Brinkmann A, Salzwedel K, Scheideler C. Efficient, distributed data placement strategies for storage area networks. in: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures. ACM, 2000: 119-128

[4] Mahesh V. STORAGE AREA NETWORKING. Mobile Computing: A Book of Reading, 2002: 99

[5] Gibson G A, Van Meter R. Network attached storage architecture. Communications of the ACM, 2000, 43(11): 37-45

[6] Clarke R. Big data, big risks. Information Systems Journal. 2016, 26(1): 77-90

[7] Armbrust M, Fox A, Griffith R, et al. A view of cloud computing. Communications of the ACM, 2010, 53(4): 50-58

[8] Kamishima T, Akaho S, Sakuma J. Fairness-aware learning through regularization approach. Data Mining Workshops,2011 IEEE 11th International Conference on. IEEE, 2011: 643-650

[9] McKusick K, Quinlan S. GFS: evolution on fase-forward. Communications of the ACM, 2010, 53(3): 42-49

[10] Guo D, Du Y, Li Q, et al. Design and realization of the cloud data backup system based on HDFS. Emerging Research in Web Information Systems and Mining. Springer Berlin Heidelberg, 2011: 396-403

[11] Leszek Holenderski L. Formal Development of Reactive Systems-Case Study Production Cell. 1995

[12] Weil S A, Brandt S A, Miller E L, et al. Ceph: A Scalable. High-Performance Distributed File System. Proceedings of the 7th symposium on Operating systems design and implementation. USEUIX Association, 2006: 307-320

[13] Santos M N, Cerqueira R. Gridfs: Targeting data sharing in grid environments. Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on IEEE, 2006, 2: 17-17

[14] Fetteerly D, Haridasan M, Isard M, et al. TidyFS: A Simple and Small Distributed File System. USENIX annual technical conference. 2011:34-34

[15] Cipar J, Corner M D, Berger E D. TFS: A Transparent File System for Contributory Storage. FAST. 2007, 7:215-229

[16] Haddad I F. Pvfs: A parallel virtual file system for linux clusters. Linux Journal, 2000, 2000(80es): 5

[17] Zhang B, Ross B, Kosar T. DLS: a cloud-hosted data caching and prefetching service for distributed metadata access. International Journal of Big Data Intelligence, 2015, 2(3): 183-200

[18] Aviles-Gonzalez A, Piernas J, Gonzalez-Ferez P. Batching operations to improve the performance of a distributed metadata service. The Journal of Supercomputing, 2015:1-34

[19] Zhang Q, Feng D, Wang F, et al. Mlock:building delegable metadata service for the parallel file systems. Science China Information Sciences, 2015, 58(3): 1-14

[20] Xu Q, Arumugam R V, Yong K L, et al. Efficient and scalable metadata management in EB-scale file systems. Parallel and Distributed Systems IEEE Transactions on, 2014, 25(11): 2840-2850

[21] Xiao L, Ren K, Zheng Q, et al. ShardFS vs. IndexFS: replication vs. caching strategies for distributed metadata management in cloud storage systems. Proceedings of the Sixth ACM Symposium on Cloud Computing. ACM, 2015:236-249

[22] Thomson A, Abadi D J. CalvinFS: consistene wan replication and scalable metadata management for distributed file systems. 13th USENIX Conference on File and Storage Technologies (FAST 15). 2015:1-14

[23] Xu Q, Arumugam R V, Yong K L, et al. DROP: Facilitating distributed metadata management in EB-scale storage system. Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on IEEE, 2013:1-10

[24] Youngjoo L E E, Jaehwan J, In-Cheol P. Energy-Scalable 4KB LDPC Decoding Architecture for NAND-Flash-Based Storage Systems. IEICE Transactions on Electronics, 2016, 99-C(2): 293-301

[25] Lee C, Lim S H. Efficient logging of metadata using NVRAM for NAND flash based file system. Consumer Electronics, IEEE Transactions on, 2012, 58(1):86-94

[26] Liu G, Liu Z, Ma L, et al. A Metadata Update Strategy for Large Directories in Wide-Area File Systems. High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Sofeware and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on IEEE, 2014: 400-403

[27] Satyanarayanan M. Distributed File Systems. Distributed Systems. Addison-Wesley and ACM Press, 1993,821:145-154

[28] Yannikos Y, Schluessler J, Steinebach M, et al. Hash-Based File Contene Identification Using Distributed Systems. Adances in Digital Forensics IX. Springer Berlin Heidelberg, 2013: 119-134

[29] Zhipeng T, Wei Z, Jianliang S, et la. An Improvement of Static Subtree Partitioning in Metadata Server Cluster. International Journal of Distributed Sensor Networks, 2012, 2012

[30] Schmuck F B, Haskin R L. GPFS: A Shared-Disk File System for Large Computing Clusters. FAST. 2002, 2: 231-244

[31] GlusterFS: Clustered File Storage that can scale to petabytes. http://www.glusterfs.org/

[32] Berns A. Avatar: A Time and Space-Efficient Self-stabilizing Overlay Network. Stabilization, Safety, and Security of Distributed Systems. Springer International Publishing, 2015: 233-247

[33] Karakoyunlu C, Runde M T, Chandy J A. Using an Object-Based Active Storage Framework to Improve Parallel Storage Systems. Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on. IEEE, 2014:70-78

[34] 张铁彬. 分布式文件系统客户端的设计与实现[硕士学位论文].上海市: 上海交通大学图书馆, 2011

[35] 田怡萌. BlueOcean 海量存储系统 Windows 客户端设计与实现[硕士学位论文]. 上海市: 上海交通大学图书馆, 2013

[36] Das S, Chatterjee D, Ghosh D, et al. Extracting the system call identifier from within VFS: a kernel stack parsing-based approach. International Journal of Information and Computer Security, 2014, 6(1):12-50

[37] O'neil E J, O'neil P E, Weikum G. The LRU-K page replacement algorithm for database disk buffering. ACM SIGMOD Record, 1993, 22(2):297-306

[38] Johnson T, Shasha D. 2Q: A low overhead high performance buffer management replacement algorithm. Proceeding of the Twentieth International Conference on Very Large Databases, Santiago, Chile. 1994:439-450

[39] Jiang S, Zhang X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. ACM SIGMETRICS Performance Evaluation Review, 2002, 30(1): 31-42

[40] Robinson J T, Devarakonda M V. Data cache management using frequency-based replacement. New York: ACM Press, 1990. 134-142

[41] Lee D, Choi J, Kim J H, et al. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. IEEE transactions on Computers, 2001, 50(12): 1352-1361

[42] Wang R Y, Anderson T E. xFS: A wide area mass storage file system. Workstation Operating Systems, 1993. Proceedings, Fourth Workshop on IEEE, 1993:71-78

[43] O'Connell M, Nixon P. JFS: A Secure Distributed File System for Network Computers. EUROMICRO Conference, 1990. Proceedings. 25th. IEEE, 1999, 2:450-457