武汉邮电科学研究院硕士学位论文

基于 Ceph 的数据读写性能优化研究

The Research on The Optimization of Read-write Performance for Ceph

专　　业：通信与信息系统

研究方向：云存储系统

导　　师：蒋玉玲

研 究 生：王筱橦　学号：20170045

二〇二〇年四月

独创性声明

本人声明所呈交的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，除了文中特别加以标注的地方外，没有任何剽窃、抄袭、造假等违反学术道德、学术规范的行为，也没有侵犯任何其他人或组织的科研成果及专利。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。如有任何侵权行为，本人愿意为此独立承担全部责任。

作者签名：_____签字日期：_____

关于论文使用授权的说明

本人完全了解武汉邮电科学研究院（烽火科技集团）有关保留、使用学位论文的规定，本文知识产权归武汉邮电科学研究院所有，武汉邮电科学研究院有权保留送交论文的复印件和电子版本，允许论文被查阅和借阅。同意将本人的学位论文提交中国学术期刊（光盘版）电子杂志社全文出版并收入《中国学位论文全文数据库》。

□公开　□保密一年　□保密两年

(注：保密的学位论文在解密后遵守此协议)

作者签名：_____签字日期：_____

导师签名：_____签字日期：_____

摘 要

Ceph 是一款去中心化的开源分布式存储系统，由于其统一接口、具有良好的可扩展性和可用性，因此被广泛应用于生产实践中。由于 Ceph 去中心化的架构，所有存储节点都参与元数据的存储与管理，所以各存储节点除处理数据存储外，还需处理频繁的元数据读写操作。除此之外，Ceph 在存储数据时，存在数据切分及 Hash 散列操作，这些操作会降低数据单元的大小及连续性。考虑到 Ceph 的这些特性，如果单纯使用机械硬盘作为后端全量数据的存储介质，将极大限制 Ceph 的性能。由于固态硬盘成本较高，在大规模数据存储场景下，也不适合作为后端全量数据的存储介质。因此设计一种固态硬盘和机械硬盘的混合存储方案对优化 Ceph 读写性能尤为重要。

因此本文从混合存储角度出发展开对 Ceph 读写性能的研究，并取得了如下成果：

(1) 设计并实现了完整的硬盘级混合存储优化方案，经实验验证该方案极大程度提升了基于 Ceph 创建的单个虚拟硬盘的 IOPS 性能和吞吐量性能。

(2) 针对 Ceph 提供的 Cache Tier 混合存储方案所采用的淘汰算法的缺陷，设计了一种基于高斯分布的缓存淘汰算法。经仿真实验验证，在用户访问符合高斯分布的场景下，该算法较 LRU 及 LRU-2 算法命中率性能更优且更接近理论极限。

关键词：Ceph；分布式存储；混合存储；Cache Tier；淘汰算法

Abstract

Ceph is a decentralized open source distributed storage system. Because of its unified interface, good scalability and availability, it is widely used in production practice. Because of Ceph's decentralized architecture, all storage nodes need to store and manage metadata. So there is going to be a lot of small random chunks of data to read or write. In addition, Ceph shred and hashes the data block as it processes read or write request. As a result, the consistency and size of data blocks are reduced. Considering these feature of Ceph, if the storage all use only HDD, it will greatly limit the performance of Ceph. Although SSD can meet the needs of Ceph, their cost is too high. Therefore, it is not suitable to use SSD to store all data in Ceph. Designing a hybrid plan based on SSD and HDD is important to optimize Ceph's performance.

Therefore, this paper studies the read-write performance of Ceph from the perspective of hybrid storage, and achieves the following results:

(1) A complete hybrid storage optimization scheme is designed and implemented. Experiments show that this scheme improves the IOPS and throughput of the virtual hard disk which created by Ceph greatly.

(2) In order to solve the algorithm defect of Cache Tier, An elimination algorithm based on gaussian distribution is designed. Simulation results show that this algorithm performs better than LRU and LRU-2 when user access conforms to a gaussian distribution.

Key words: Ceph; Distributed storage system; Hybrid storage; Cache Tier; Elimination algorithm

目 录

1 绪论

1.1 引言

Ceph 是一款统一的开源分布式存储系统，其设计初衷是采用商用硬件来构建大规模的、具有高可用性、高可扩展性、高性能的分布式存储系统[1]。

由于其统一存储，支持对外提供块存储、文件存储、对象存储服务，同时具备高可用、高可扩展、高性能的特性，因此在各领域都有广泛的使用。据 Ceph 中文社区 2018 年国内用户不完全统计结果显示，Ceph 不仅服务于金山云、美团云、青牛云等云服务提供商，也服务于京东、阿里、腾讯等互联网行业，中国移动、中国电信、中国联通等通信运营商，人民日报、今日头条等媒体行业，国家电网、南方电网等能源行业，网易游戏、完美世界等游戏行业等。

不同于其它基于文件系统的分布式存储系统，Ceph 采用的是去中心化的分布式架构，不存在与分布式文件系统中的元数据服务器类似的中心节点，在架构上具有一定的优势。在分布式文件系统中，所有数据单元的元数据信息都存储在元数据服务器上，在进行 IO 操作时都需先访问元数据服务器查表以获取数据单元的元数据信息，而后再到相应存储节点进行 IO 操作[2]。元数据服务器的性能及元数据的组织方式决定了整个系统的性能，因此元数据服务器是整个系统的中心，同时也是其性能瓶颈。Ceph 放弃了基于中心节点查表的数据寻址方式，通过 CRUSH 算法计算寻址，然后将数据单元下发至各存储节点，数据单元的大部分元数据信息都存储在各存储节点上。由于整个集群中的所有节点都参与寻址过程，因此 Ceph 规避了引入中心节点而导致的性能瓶颈问题。除此之外，Ceph 以对象存储为基础设计底层存储系统，其数据单元较传统文件系统来说可以提供更多元数据信息，丰富的在存储节点上本地存储的元数据信息为未来网络智慧存储[3]提供了更多的可能。考虑到 Ceph 的这些优越性，本文课题选取 Ceph 作为研究对象，并展开对其读写性能的优化研究。

Ceph 采用去中心化的架构，所有数据对象的存储和管理均由各存储节点上的 OSD(Object Storage Device)完成，OSD 一般与存储节点的硬盘设备一一对应。Ceph 在处理来自应用层的请求时，会先对应用层的数据对象进行切分，然后通

过 Hash 散列及 CRUSH 算法将切分后的数据对象下发至 OSD。该过程会减小 OSD 所处理的数据对象的大小，同时也会提高数据对象被访问时的离散程度。由于 OSD 通过日志机制保证元数据的一致性从而保证数据的一致性，因此 OSD 在处理上层请求时还会产生大量的小块随机 IO。

影响存储系统整体读写性能的因素主要包括业务数据的读写模式、存储系统自身的数据管理方式、IO 架构以及底层存储硬件的选择及使用方式等。目前 Ceph 的底层存储资源管理组件 OSD 常使用的硬盘介质包括固态硬盘和机械硬盘。由于固态硬盘成本太高，在存储大量数据的场景下，并不适合作为后端全量数据存储的存储介质。机械硬盘成本不高，但其物理特性决定其在处理离散程度高、数据大小较小的 IO 请求时性能较弱。由于 OSD 在处理上层请求时会产生大量小块随机 IO，因此如果单独使用机械硬盘作为 OSD 的存储介质，将严重影响 OSD 的性能，进而影响 Ceph 集群整体的性能。因此基于 Ceph 的工作原理，设计一种固态硬盘和机械硬盘的混合存储方案显得极其必要。

通过缓存技术优化由固态硬盘和机械硬盘构成的混合存储系统是当前工业界和学术界的重要课题[4]。除了依据 Ceph 工作原理合理划分 OSD 对固态硬盘及机械硬盘的使用空间外，缓存技术也为混合存储方案的设计提供了方向。缓存技术是一种将用户所访问的热点数据存储在访问时延较小的介质上以加速后端慢速介质数据读写速度的技术，其要解决的核心问题是：对有限的缓存资源如何实现有效的规划和调度[5]。淘汰算法是对缓存空间资源规划和调度的手段之一，通过淘汰算法，可以尽可能保证缓存空间中存放的是将来最后可能被访问的数据，以此将大量用户请求转移至高速缓存，从而提升整体的读写性能。

因此本文课题从混合存储的角度出发，一方面，通过分析 Ceph 读写过程中所产生数据的特征，设计固态硬盘空间及机械硬盘空间划分方案；另一方面展开缓存技术在 Ceph 中的应用及优化研究。

1.2 国内外研究现状

通过缓存机制优化混合存储系统读写性能，业界研究方向主要大致可分为如下三种：第一种，针对缓存介质及后端存储介质的物理特性及工作原理，优化缓存空间的管理结构以规避存储介质的缺陷，从而提升存储介质的性能；第

二种，优化缓存在系统中的工作层级及结构，通过使整个系统结构更为高效合理，从而提升系统的整体性能；第三种，优化淘汰算法，通过提升缓存空间的命中率，将更多用户请求转移至缓存，从而提升读写性能。

在固态硬盘和机械硬盘的混合缓存存储系统中，固态硬盘由于其工作原理及物理特性，存在很多性能缺陷，为缓解这种缺陷带来的性能损耗，间接提升缓存性能，很多研究学者就该方向展开研究。固态硬盘工作时需要额外空间进行垃圾回收操作，为了加速固态硬盘的垃圾回收过程，首尔大学的 Oh 等人提出了 OP-FCL[6]。OP-FCL 将缓存空间划分为读区、写区和预留区域，并根据负载的特征动态调整三者的大小将固态硬盘性能最优化。由于固态硬盘的使用寿命有限，为了减少对固态硬盘的写入次数，提高固态硬盘的使用效率，希腊克里特大学的 Makatos 等人提出了 FlaZ[7]，其思想是当缓存未命中时，从磁盘读取的数据先由 Lempel-Ziff-Welch(LZW)无损压缩后再写入缓存，当缓存命中时，通过对数据进行相应解压缩后再读出。随着非易失性内存（NVM，no-volatile memory）的出现，还有部分学者将缓存介质选取的目光由固态硬盘转向了非易失性内存并展开研究，如文献[8]、文献[9]等。

传统存储场景中，缓存主要工作在硬盘驱动的上一层，用以对硬盘设备进行加速。但在大规模数据存储环境下，如果仅将缓存工作层面设定在硬盘驱动的上一层，将不利于整个系统的可管理性和可靠性。为解决该类问题，美国 NetAPP 公司的 Byan 等人提出的 Mercury[10]系统将缓存以共享方式部署在虚拟管理层，为其上的所有虚拟主机提供缓存服务。类似的，Ceph 提供的 Cache Tier 机制也将缓存定义在存储池这个逻辑层，用以给上层应用提供缓存服务；HDFS 也在 2.3.0 版本开始引入的集中式缓存管理机制，将缓存用以给上层应用提供服务。除此之外，大规模数据存储往往面临的大量的数据访问，该场景下网络传输的开销将成为系统的读写性能瓶颈。为了解决该类问题，内容分发网络(CDN，Content Delivery Network)和内容中心网络(CCN，Content-centric Networking)将缓存定义在了靠近用户的边缘节点这种新的系统层次上。除了优化缓存所在系统中的工作层级外，还有一部分学者将目光锁定在缓存本身的结构上，比如文献[11]设计的 HC Model 机制，通过将缓存整体分为三层，并对缓存数据进行分层管理，从而提升缓存性能。

LRU、LFU 以及 ARC 算法是比较经典的淘汰算法，该类算法通过频率估计概率的原理设计淘汰规则，采用该方式设计淘汰规则，一方面，缓存空间容易受到冷数据的污染，另一方面频率估计概率存在的误差，容易影响对热点数据的准确判断。针对这些问题，关于淘汰算法的优化研究集中在设计规则有效过滤冷数据或重新设计频率量化指标上。如文献[12]、文献[13]以及文献[14]等通过相应结构定义频率阈值从而过滤冷数据来提升淘汰算法命中率；又如文献[15]定义了一种新的缓存空间管理结构，将数据对象的访问频率重新定义为数据块两次访问之间的非重复数据个数，从而提升淘汰算法的命中率。由于缓存工作的层面不再仅于硬盘驱动的上层，如内容分发网络和内容中心网络的架构中，缓存工作在整个系统的边缘节点上，该场景下缓存所能利用的信息更多，基于这些信息可以对用户行为进行有效分析从而设计出更为高效的淘汰算法，如文献[16]设计了一种基于边缘节点到源服务器之间距离的缓存算法，距离越大的数据越容易缓存在边缘节点，文献[17]设计了一种给缓存空间中数据分配权重，按权重淘汰的淘汰算法，该算法在进行权重计算时，会综合考虑数据的大小、存储在缓存空间的代价以及数据存储的时间。文献[18]设计的淘汰算法综合考虑缓存命中率、最近请求时间间隔以及请求次数等因素，并据这些参数统计算内容流行度，内容流行度越小的数据越不容易缓存，文献[19]基于古洛博弈的理论设计了一种边缘节点协同工作的淘汰算法，该算法不仅考虑当前边缘节点的链路代价，而且考虑集群整体的链路代价，通过协同机制降低整个系统的链路代价，提升系统整体的读写性能。

1.3 主要工作及论文组织结构

以目前 Ceph 的整体架构，Ceph 的 Cahe Tier 提供存储池级的缓存功能，而 OSD 基于硬盘工作可通过硬盘级的缓存技术进行性能加速，因此本文课题主要从 Cache Tier 及 OSD 所采用的硬盘组织结构角度展开缓存优化研究。

虽然目前有 flashcache、bcache 等开源的硬盘级缓存优化方案，但 OSD 所处理的数据种类较多，除利用缓存加速外，还需结合 OSD 各数据的特征进行存储分区，才能制定出最优的混合存储方案。目前在 Ceph 的相关研究中并没有相应完整且公开的基于上述原理的混合存储优化方案。

Ceph 支持的 Cache Tier 机制虽然提供了一种存储池级的缓存方案，但在使用时也面临极大的性能问题，其采用的淘汰算法并没有充分利用数据对象所携带的信息，因此其淘汰算法理论上有优化空间。

鉴于上述因素，本文主要围绕硬盘级混合存储优化方案的设计实现及对 Cache Tier 淘汰算法的优化设计展开对 Ceph 读写性能优化的研究，并提出了适配 Ceph BlueStore 存储引擎的，基于 Flashcache 设计的完整混合存储优化方案以及理论上可用于优化 Cache Tier 缓存命中率的基于高斯分布的淘汰算法。

围绕上述两个研究方向，本文主要开展了如下工作：

(1) 从理论上研究机械硬盘与固态硬盘的性能差异、flashcache 的工作原理、Ceph 的工作原理，为后续设计奠定理论基础。

(2) 基于理论研究的结论，设计用于 Ceph 的完整硬盘级混合存储加速方案，并实现相应管理工具，提高方案的可管理性。

(3) 分析在云计算场景下用户访问的特点，依据分析结果，并基于 Ceph 对象存储的特点以及 Cache Tier 所采用的淘汰算法的缺陷，设计了一种基于高斯分布形态特征的淘汰算法。

(4) 设计实验，验证本文所设计的硬盘级混合存储方案对 Ceph 读写性能的提升程度以及本文所设计的淘汰算法相较 LRU 类算法的性能提升程度。

依据研究过程及成果，全文在组织时总共分为六个章节，各章节内容如下：

第一章为绪论部分，首先阐述了本文研究课题所基于的分布式存储系统 Ceph；然后论述了混合存储对 Ceph 读写性能优化的必要性，并选取缓存技术作为出发点展开优化研究；最后分析了目前国内外对缓存技术的研究进展，简要介绍了几种比较有代表性的研究成果的工作原理。

第二章对本文课题相关的各种技术展开研究。首先介绍机械硬盘与固态硬盘的工作原理，从理论上分析了其各自适用的场景；随后阐述了开源的硬盘级缓存方案 Flashcache 的工作原理；最后对 Ceph 的整体架构、数据读写流程及 Cache Tier 机制进行了详细论述。该章主要为后文的优化设计奠定理论和技术基础。

第三章依据第二章的理论基础，设计并实现了用于 Ceph 的硬盘级混合存储优化方案。本章首先详细论述了优化方案的设计原理；随后清晰介绍了对应该方案的硬盘管理组件的设计架构，并对该组件的核心功能：固态硬盘空间默认

划分、固态硬盘空间管理、OSD 的创建以及 OSD 的删除的设计实现做了详细介绍。

第四章基于第二章所分析的 Cache Tier 的工作原理，展开存储池级淘汰算法的优化研究。本章首先分析了云计算场景下用户访问的行为特点；随后分析了 LRU 类算法在用户访问符合高斯分布场景下的性能缺陷；最后基于行为分析的结果及 LRU 类算法在高斯分布下的缺陷，详细论述了本文课题所设计的淘汰算法的工作原理及架构，并详细介绍了该淘汰算法的稳态监控机制以及淘汰过程的实现方法。

第五章为实验与分析部分。本章主要分为两个部分，第一部分主要介绍了本课题所设计的硬盘级缓存方案性能验证实验，并阐述了该方案与原生 Ceph 集群性能对比结果；第二部分主要介绍了本课题所设计的淘汰算法与 LRU 算法及 LRU 类算法中理论上命中率最高的 LRU-2 算法的命中率对比实验，并阐述了用户访问符合高斯分布及 zipf 分布下命中率对比结果。

第六章对本课题进行了总结，分析设计的缺陷与不足，指出后续工作的研究方向。

2 相关研究及关键技术

本文的研究是从混合存储角度出发，基于缓存技术对 Ceph 的读写性能进行优化，其涉及技术众多，因此有必要对相关技术的工作原理进行详细研究和深入研究，从而为后文优化方案的设计奠定理论基础。绪论部分讨论了混合存储对于 Ceph 的必要性，混合存储主要是为了结合应用场景的业务特点，综合固态硬盘和机械硬盘的优点来提升性能，因此本章首先对机械硬盘和固态硬盘的工作原理进行了详细论述，并明确了各自所适用的场景。虽然目前有很多优秀的开源的硬盘级缓存方案，且各缓存方案在目前各分布式存储系统中都有着广泛使用，但考虑到 flashcache 已迭代更新 3 个版本且最新版本已稳定工作多年，故本课题选取 flashcache 用以实现 Ceph 硬盘级混合存储优化方案中的缓存部分，因此本章接着对 flashcache 的工作原理进行了详细介绍。本课题设计的优化方案必须基于 Ceph 的工作原理才有意义，故随后还对 Ceph 的整体架构、读写流程及其自带的缓存机制 Cache Tier 机制进行了详细论述。最后，本章对所有的理论研究进行总结，阐述了后续设计的理论依据。

2.1 硬盘工作原理

机械硬盘与固态硬盘是目前使用最为广泛的两种永久性存储介质。由于物理特性的差异，固态硬盘与机械硬盘对于各种类型 IO 处理的时延上存在性能差异。鉴于在分布式存储系统中，数据最终都会存储在后端硬盘上，出于对生产环境下分布式存储性能、成本等综合因素的考虑，研究两者的工作原理，明确各自优缺点及理论原因，是后续硬盘级优化方案设计的理论基础。

2.1.1 机械硬盘工作原理

机械硬盘是一种通过电机驱动的存储设备，通过改变磁性介质的磁性来实现数据的存储，整体由磁盘盘片、磁头及控制电机等组成，其物理结构大致如图 2-1 所示。

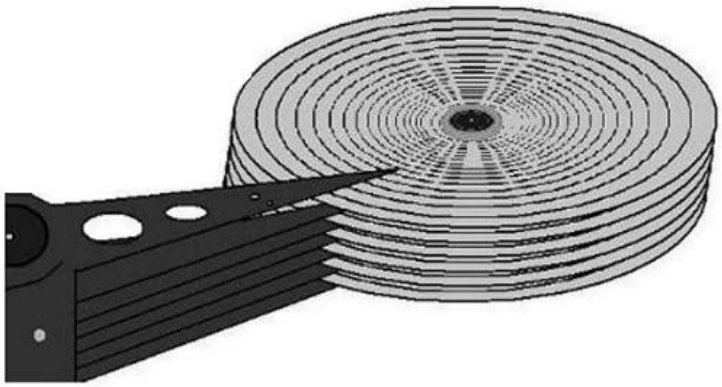

图 2-1 机械硬盘物理结构图

为方便存取数据，机械硬盘定义了柱面、磁道及扇区等逻辑结构来管理磁盘上的存储空间。当需要从机械硬盘读取数据时，需要将指定的磁头移动到指定的磁道及扇区上才能进行准确的数据读取。业内将磁头移动到相应磁道的过程所耗费的时间称为寻道时延；将目标扇区旋转至磁头下方所耗费的时间称为旋转时延。即机械硬盘的一次完整 IO 过程所耗费的时延由寻道时延、旋转时延以及数据传输时延组成。由于寻道时延和旋转时延是由电机驱动的机械过程产生，其耗时远大于数据以电信号形式在总线上的传输时间，因此寻道时延及旋转时延是机械硬盘访问时延的重要构成因素。

在连续 IO 和随机 IO 场景下，由于随机 IO 会导致频繁的寻道及旋转，所以在同等数据量的条件下，随机 IO 场景下的寻道时延和旋转时延会远大于连续 IO 场景。在大块 IO 和小块 IO 场景下，一方面，同等数据量情况下，小块 IO 请求磁盘处理的次数会大于大块 IO，因此寻道时延和旋转时延会随访问次数成倍增加；另一方面，由于磁盘空间是通过文件系统来进行管理，小块 IO 场景下，在进行磁盘空间回收时，会产生大量细碎空间，长期使用后会导致连续文件实际物理存放位置上并不连续，从而导致大块 IO 处理时延也会跟着增加。因此综合各种 IO 场景，机械硬盘的物理特性决定其并不适合处理小块 IO，尤其是小块随机 IO。虽然机械硬盘采用电机驱动，但由于电机的高转速，理论上在大块连续 IO 场景及海量数据后端存储的应用背景下，机械硬盘的性能是可以接受的。

2.1.2 固态硬盘工作原理

固态硬盘是一种通过逻辑电路驱动的存储设备，通过改变浮栅 MOS 晶体管中浮栅上存储的电荷量来改变 MOS 管的导通阈值电压从而实现对数据的存

储。以存储单比特位的浮栅晶体管为例，多比特位的存储电路结构如图 2-2 所示。

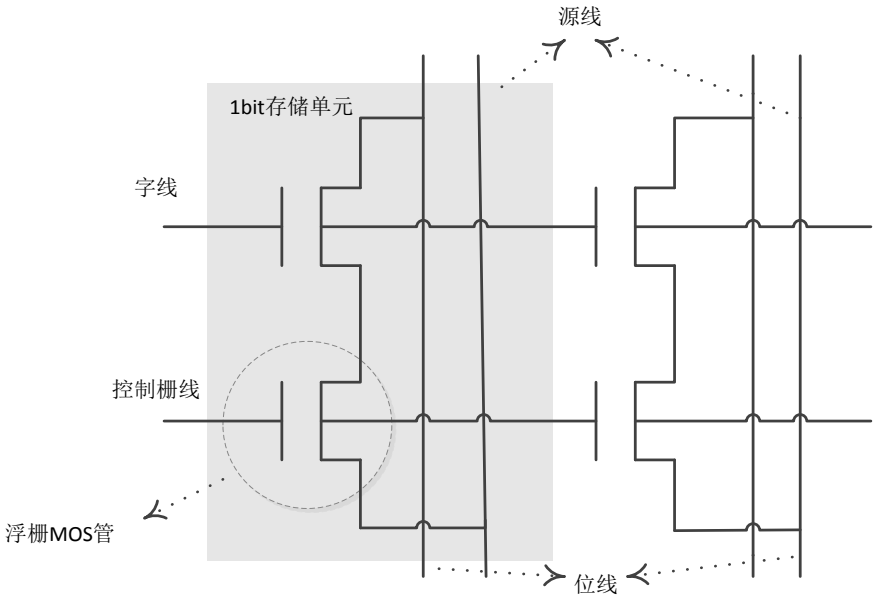

图 2-2 浮栅晶体管电路结构示意图

当擦除数据时，源线和位线均为低电平，控制栅线为高电平，当字线电压为控制栅线电压时，电子注入到浮栅，浮栅 MOS 管导通阈值电压升高；当写入数据时，位线位高电平，源线悬空或者为低电平，使电子从浮栅中流出，浮栅 MOS 管导通阈值电压降低。由于固态硬盘通过位线、源线、字线、控制栅线来实现对存储单元的读写操作，故而在进行 IO 寻址时，只需通过逻辑电路进行控制即可。因此与机械硬盘相比，固态硬盘寻址时延并不存在寻道时延、旋转时延等机械因素产生的时延，其主要考虑的是数据传输时延。因此固态硬盘的时延要较机械硬盘的时延小很多，在连续 IO、随机 IO、小块 IO、大块 IO 场景下 IO 时性能都要高于机械硬盘。

虽然固态硬盘性能较高，但由于生产工艺以及技术的原因，其容量要远小于机械硬盘，单位容量的数据成本也远高于机械硬盘。除此之外，由于浮栅 MOS 管的物理特性，其擦除次数有限，故固态硬盘的寿命有限，使用的生命周期也不如机械硬盘[20]。

综合来说，固态硬盘性能虽高于机械硬盘，但成本决定其不适合作为大规模分布式存储系统中全量数据的存储介质。更适合作为分布式存储系统中分层缓存的存储介质，或数据总量较小但访问频率高的特定数据的存储介质。

2.2 FlashCache 技术

Flashcache 是 facebook 技术团队推出的硬盘级分层缓存开源技术。其基本原理是基于 Linux 内核中的 Device Mapper 机制，通过在文件系统和块设备驱动之间增加一层 IO 重定向层，进而实现热点数据和全量数据的分离存储。

通常在使用时，Flashcache 会将固态硬盘和机械硬盘映射成一个带缓存的逻辑设备。当用户 IO 请求到来时，Flashcache 会依据所配置的缓存策略，尽可能的将访问频率较高的热数据存放在容量有限但读写速度较快的固态硬盘上，而将全量数据存储在容量充裕但读写速度较慢的机械硬盘上。由于用户大部分 IO 请求都是访问热点数据，因此大部分 IO 请求最终会下发至读写速度较快的固态硬盘，从而使用户整体 IO 请求的处理时延下降，提升硬盘读写性能。

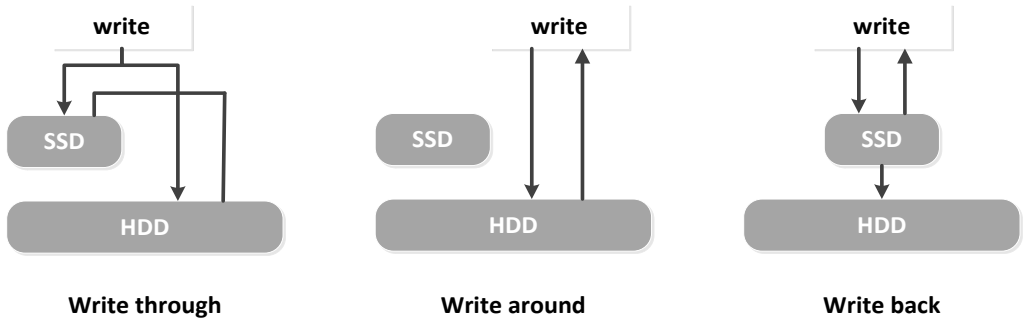

图 2-3 Flashcache 工作模式图

由于读请求不会导致全量数据的变化，写请求会造成全量数据的变化，且这部分数据最终都需要刷回的全量数据的存储介质——机械硬盘上，因此在处理写数据请求时有多种不同的策略。按写缓存策略的不同，Flasshcache 提供了如图 2-3 所示的 3 种工作模式。“write around”模式会将用户的所有写请求会直接下发到机械硬盘，不加速用户的数据写入过程；“write through”模式在处理写请求时，会同时将数据写往固态硬盘和机械硬盘，由于每次写入操作都需要等待固态硬盘和机械硬盘都完成写入，所以也不会加速数据写入的过程；“write back”模式会依据策略，将访问热点数据的写请求下发至固态硬盘，而后依据策略将固态硬盘上的数据整理刷回至机械硬盘，该模式不仅会减少用户写数据至机械硬盘的次数，同时在数据整理过程中会将小块随机的 IO 尽可能整理为大块连续 IO，从而缓解机械硬盘因处理小块 IO 而导致的性能损耗，因此该模式除了可以加速读过程外，也可以加速写过程。

在 Ceph 中，数据读写请求最终都会下发至硬盘驱动。利用工作在“write back”模式下的 Flashcache 技术加速其所纳管的机械硬盘，不仅可以减少机械硬盘小块 IO 数，增加机械硬盘所接收的 IO 的数据长度，还可以在低成本的前提下，充分利用固态硬盘的性能实现对后端机械硬盘的访问加速，因此理论上可以作为 Ceph 硬盘级混合存储方案中缓存部分的实现方式。

2.3 Ceph 的架构及关键技术

Ceph 是目前使用最为广泛的基于对象存储的分布式存系统，与基于文件系统的分布式存储产品相比，Ceph 有着自己独特的数据处理流程。Ceph 的应用层在进行数据到存储位置的地址映射时不再基于传统的查表法获取，而是基于 CRUSH 算法计算得到；Ceph 对底层硬盘资源的管理也不再基于文件系统，而是基于对象存储。因此 Ceph 读写过程中所产生的数据的性质会不同于基于文件系统的分布式存储系统。只有深入研究 Ceph 的工作机制才能明确混合存储所要面对的数据的种类及特征，才能设计有效的混合存储优化方案。

2.3.1 Ceph 的整体架构

Ceph 的整体架如图 2-4 所示，整体由 4 个部分组成：应用层、上层应用接口层、基础库 LIBRADOS 及基础存储系统 RADOS(Reliable Autonomic Distributed Object Store)。

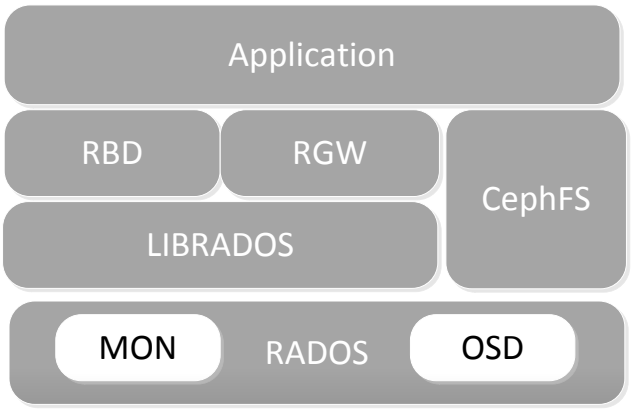

图 2-4 Ceph 整体架构图

RBD(Rados Block Device)用以给上层应用提供块存储服务，RGW(Rados Gateway)用以给上层应用提供对象存储服务，CephFS(Ceph File System)用以给上层应用提供文件系统服务，三者统称为上层应用接口层，用以依据不同存储

业务的需求对应用层提供不同存储类型的服务。基础库 LIBRADOS 是对 RADOS 的抽象和封装，用以为上层应用提供操作 RADOS 的 API(Application Programming Interface)，由于 CephFS 有以 MDS(Metadata Server)为核心的读写管理机制，直接操作 RADOS，故 LIBRADOS 主要为 RBD 以及 RGW 提供服务。

RADOS 是一个完整的分布式对象存储系统，用来提供数据存储服务。RADOS 集群由两部分组成：一种是负责数据存储和维护的 OSD，一般与硬盘设备一一对应；另一种是负责监控集群状态和配置信息的 Monitor，与部分具有集群监控和配置功能的存储节点存在一一对应关系。MON(Monitor)为上层应用以及集群内的其它节点提供更新集群拓扑(Cluster Map)的服务，同时维护更新集群的状态及配置信息，并不提供数据存储的服务。当应用层发起 IO 请求时，首先会向 Monitor 请求集群拓扑，然后基于集群拓扑通过 CRUSH 算法计算数据对象所在的 OSD，随后将 IO 请求下发至计算所得的 OSD，OSD 通过底层存储引擎 ObjectStore 将数据写入磁盘。

从 Ceph 的整体架构可知，RADOS 的性能很大程度上决定了 Ceph 的数据读写性能。

2.3.2 RADOS 的数据映射机制

对于 RADOS 来说，RBD 或 RGW 或 CephFS 即是其客户端。当客户端下发 IO 请求后，RADOS 会依据 MON 记录的集群信息计算其所在的 PG，后通过查表的方式找到相应 OSD，并将请求下发至 OSD，最后由 OSD 通过对象存储引擎把 IO 请求下发至相应的硬盘设备。Ceph 官方社区给出的客户端将 IO 请求映射到 OSD 的数据映射示意图，如图 2-5 所示。

图中 File 指用户通过上层应用接口所访问的数据对象。以 RBD 接口为例，对于块设备而言，用户在读写块设备时是以流式方式访问，即通过指定起始扇区地址 offset 和长度 length 来访问块设备，因此，应用层通过 RBD 使用 Ceph 所管理的存储资源时，数据对象 File 是指从起始扇区地址 offset 开始长度为 length 的地址空间中的数据。

图中 Objects 是 RADOS 所管理的数据单元，其最大大小由 RADOS 指定。当应用层数据对象 File 下发至 RADOS 时，会对超过 Object 的最大大小的 File 按 Object 的最大大小进行切分。然后再依据不同的数据冗余策略，将切分后的

各 Object 下发至 OSD。若采用 N 副本机制，那么会将切分后的 Object 复制 N 份，分别下发至处于不同故障域的 N 个 OSD 上；若采用 N+M 纠删码机制，则会将切分后产生的 Object 再切分为 N 等份，并依据这 N 份数据计算出 M 个记录纠删码信息的等份数据，最后将这 M+N 等份数据下放至处于不同故障域的 OSD。

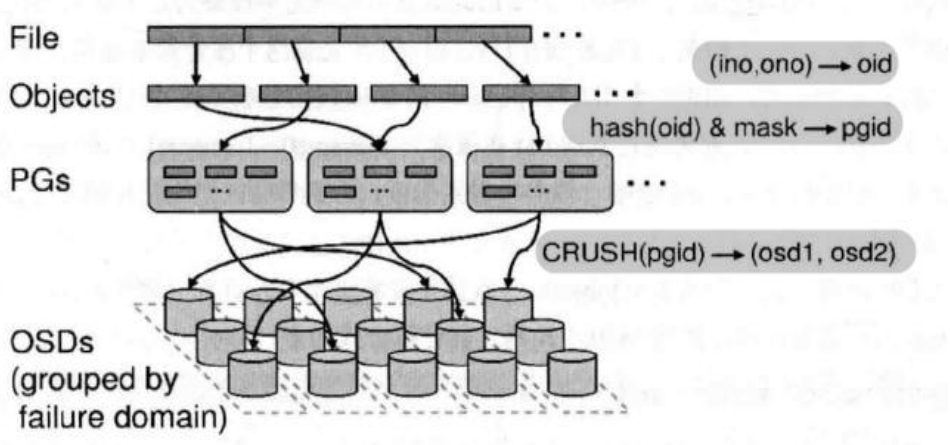

图 2-5 Rados 数据映射示意图

由于 Object 数量庞大，且与 OSD 存在一对多的关系，为了简化 OSD 与 Object 的映射关系，降低系统维护管理的开销，RADOS 引入放置组(PG，Plachement group)这一逻辑单元来管理 Object 到 OSD 的映射关系。RADOS 会依据 CRUSH 算法生成指定数量的放置组。通过 CRUSH 算法，每个放置组都会与一定数量的 OSD 对应。放置组中 OSD 的数量与所配置的数据冗余策略有关，如 3 副本，则一个放置组对应指定的 3 个 OSD。每一个 Object 在产生时，会通过一种简单的线性映射方式得到唯一的 oid。假定 RADOS 共生成 2^{mask} 个放置组，为使 Object 在放置组上均匀分布，因此在获取 oid 后，会对 oid 通过指定的静态 HASH 函数进行散列到相应 HASH 值，然后再与 2^{mask}-1 进行与运算，得到相应放置组的编号，从而建立 Object 与放置组的映射关系。

CRUSH 算法基于集群拓扑、数据分布策略(Data Distribution Policy)和给定的放置组的 ID 共同计算 OSD 在 PG 上的分布。集群拓扑是一种如图 2-6 所示的树形结构，记录了所有存储资源在空间层次上的关系，即存储集群有哪些机架，每个机架下有哪些服务器，每个服务器上有哪些磁盘等信息。集群拓扑主要由两部分组成，一部分称为 device，用以表示 RADOS 最基本的存储单元 OSD；另一部分称为 bucket，表示 OSD 的容器，可包含 device 也可递归包含子类型的

bucket。集群拓扑默认的 bucket 类型有“root”、“datacenter”、“room”、“row”、“rack”、“host”六个等级，用户也可以自己定义新的类型。每个 device 会依据所对应 OSD 管理的磁盘空间容量设置权重，bucket 的权重是其包含的 device 或 bucket 的权重之和，bucket 的结构定义如下。

```
[bucket-type]  [bucket-name] {
        id              [a unique negative numeric ID]
        weight          [the relative capacity/capability of the item(s)]
        alg             [the bucket type : uniform | list | tree | straw]
        hash            [the hash type : 0 by default]
        item            [item-name]         weight          [weight]
  }
```

上述定义中的 alg 指明了该 bucket 下节点选择算法的类型；hash 指明了该 bucket 在进行放置组 ID 散列映射时所采用的 hash 算法，不论是哪种 hash 算法，其函数形式均为 hash(x,r,i)，x 表示放置组的 ID，r 为副本序号，i 为 bucket 的 id 号。

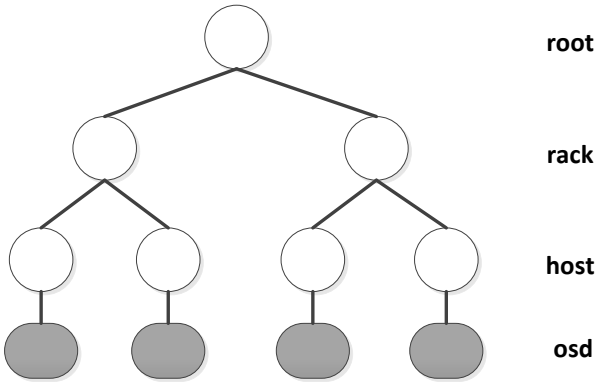

图 2-6 集群拓扑示意图

集群拓扑反映了存储系统层级的物理拓扑结构及各层级的选择规则，而数据分布策略则决定了基于层级拓扑为 PG 选择 OSD 的规则。这些规则用户可以通过 Ceph 中的 CRUSH RULE 自定义，CRUSH RULE 的定义的语法规则如下。

```
rule  <rulename> {
        ruleset         <ruleset>
        type            [replicated|erasure]
        min_size        <min-size>
        max_size        <max-size>
        step take <bucket-name> [class <device-class>]
```

```
    setp [choose|chooseleaf] [firstn|indep] <N> type <bucket-type>
    step emit
}
```

ruleset 表示该数据分布策略的 ID 编号；type 表示该数据分布策略中，数据冗余策略是采用纠删码策略还是副本策略；min_size 表示该数据分布策略工作的最小数据冗余度，如果所设置的数据冗余度小于该值，该规则将失效；max_size 与 min_size 相对，表示该数据分布策略工作的最大数据冗余度；step take <bucket-name> [class <device-class>] 用以指定一个 bucket，并从这里开始迭代树，如果指定了 device-class，则在选取设备时只选取该类的设备，不属于这个类的设备全被排除在外；step choose firstn {num} type {bucket-type}表示从指定类型的 bucket 中选择一定数目的 bucket；step chooseleaf firstn {num} type {bucket-type}表示选择某一类型的多个 bucket，并在各个 bucket 下任意选择一个叶子节点。

集群拓扑结构和数据分布策略共同作用，完成 PG 到 OSD 的映射过程，该映射的基本步骤为：

(1) 给出一个 PG 的 ID，作为 CRUSH_HASH 的输入；

(2) 参照集群拓扑所定义的空间结构，依据数据分布策略所定义的规则逐步选择；

(3) 每一步选择都依据集群拓扑中该层结构所设置的 hash 函数来对指定 PG 的 ID 进行散列，并采用该层结构类型对所对应的权重计算算法计算权重，选取权重最大的 bucket 或 OSD 返回；

(4) 执行完步骤，返回符合数据冗余策略数量要求的 OSD 集合；

RADOS 数据映射机制中对客户端的数据切分操作，会使原本的大块 IO 长度变小。在从 Object 到 OSD 的映射过程中，存在 HASH 散列过程，因此数据原本的连续性也会降低。虽然这种设计可以保证数据在各存储资源上均匀分布，同时方便管理，但其硬盘级存储管理单元 OSD 所面对的数据对象也同样会呈现分布随机且长度有限甚至很小的特征。

2.3.3 RADOS 的数据读写流程

RADOS 的整体读写流程如图 2-7 所示，所有 IO 请求在 Client 端发出，经过 Message 层统一解析后分发到相应放置组所对应的 OSD。由于每个 OSD 上

承载在多个放置组，因此 OSD 为其上的每个放置组都设置了一个工作队列，并通过线程池对每个队列的请求进行处理。IO 请求的最终处理由 OSD 通过对象存储引擎(ObjectStore)下发至相应的硬盘设备完成。因此，如何根据 IO 请求对硬盘数据进行 IO 操作，主要由对象存储引擎决定。

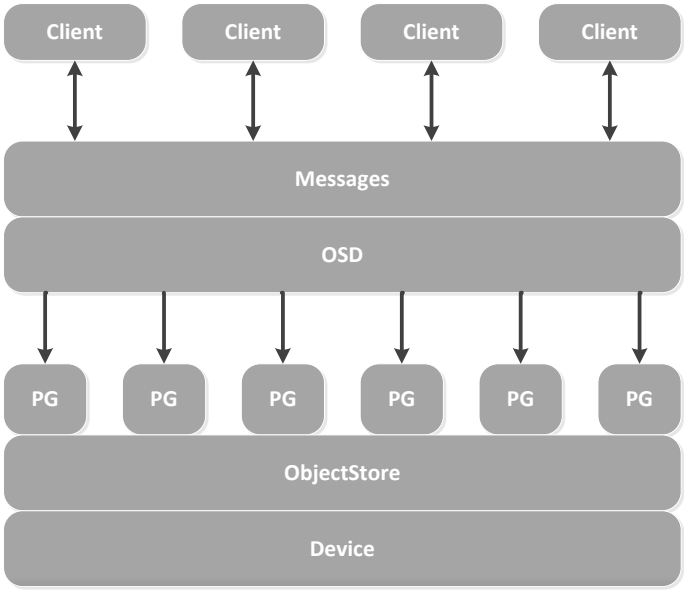

图 2-7 RADOS 读写流程图

OSD 的对象存储引擎在进行处理 IO 请求时，主要操作三种类型的数据：数据对象所包含的数据信息、数据对象的元数据信息以及保证数据一致性的日志数据。

目前 Ceph 所采用的对象存储引擎主要有两种：FileStore 对象存储引擎以及 BlueStore 对象存储引擎。FileStore 对象存储引擎主要用于 L 版本以前的 Ceph 集群中，是基于支持扩展属性的通用本地文件系统（如 ext4、xfs 等）来实现的存储引擎。BlueStore 对象存储引擎主要用于 L 版本后的 Ceph 集群中，该存储引擎定义了一个简单的轻量级的不工作在 Linux 虚拟文件系统(VFS)下的文件系统 BlueFS，BlueFS 通过逻辑封装以函数接口的形式为 RocksDB 提供类似文件系统的能力，并以此实现的专用于 Ceph 的对象存储引擎。

数据对象的元数据信息通常以键值对的形式进行存储，这类元数据有两种存放方式，一种是存放在文件系统的扩展属性中；另一种是存放在专门的键值对数据库中。在处理元数据时，FileStore 对象存储引擎将部分较小且经常被访问的元数据信息存放到本地文件系统的扩展属性中，将一些大的不经常被访问的元数据信息存放到基于本地文件系统的键值对数据库。由于 FileStore 对象存储

引擎需要通过本地文件系统去读写元数据，进而操作硬盘空间，因此其工作的IO 路径较长。BlueStore 对象存储引擎则不再通过本地文件系统去操作硬盘，而是直接操作裸设备。基于 BlueFS 的 RocksDB 为 BlueStore 对象存储引擎提供全量元数据的存储和管理，因此其工作时的 IO 路径要比 FileStore 更短，读写性能更优。

日志数据用以保证在数据还未完全写入便结束的异常场景下的数据一致性。当对象存储引擎接收到用户写请求后，会先将写请求及其数据封装成指定的日志记录，写入到日志硬盘空间，如果写日志过程中出现异常，则丢弃这部分日志，认为写入失败；如果在日志数据写入到硬盘空间时出现异常，则通过日志回放重新写入这部分数据。因此对象存储引擎在处理用户的写请求时，会带来两次写入的问题，导致写放大，降低数据写入效率。由于 FileStore 是基于通用文件系统设计的对象存储引擎，在面对写放大问题时，存在较高的写放大倍数。为了降低二次写入带来的写放大，Blustore 基于专用的 RocksDB 通过ROW(Redirect on write)的方式来进行数据写入。ROW 是指当数据需要覆盖写入时，将数据写到新的位置，然后更新元数据索引。因此在该写入模式下，日志数据只需要维护元数据的一致性即可保证数据的一致性。由于在非对齐覆盖写场景下，采用 ROW 的方式写入，会增大元数据索引数，降低读性能，同时浪费存储空间，为此 BlueStore 还引入了 RMW(Read Modify Write)机制。RMW 是指当发生非对齐覆盖写时，会先读取旧的数据，将新写入的数据与旧数据合并，对齐写入到磁盘，从而减少 ROW 模式下元数据的额外开销，提高读性能。由于在写入时需要先读取数据，因此该模式存在一定的性能损耗。Blustore 混用ROW 和 RMW 写入方式，在 BlueStore 中有一个最小分配单元 min_alloc_size 的配置项，一般为磁盘块大小的整数倍，写入的数据如果与 min_alloc_size 大小对齐，则使用 ROW 的方式；对于非 min_alloc_size 对齐的区域，则使用 RMW 的方式，从而有效降低二次写入时的写放大倍数。

综上， BlueStore 对象存储引擎较 FileStore 对象存储引擎有更优的性能，因此本文课题主要针对 BlueStore 对象存储引擎设计硬盘级缓存优化方案。

BlueStore 对象存储引擎支持对不同类型的数据单独指定存储介质，其整体架构如图 2-8 所示。由上文分析可知，在 Bluestore 中元数据和日志数据具有数据总量小、单次访问数据量小、访问频繁的特点，而存储的数据单元则如 2.3.2

章所分析的具有数据量大、数据访问离散程度较高、单次数据访问量有限甚至很小的特点。

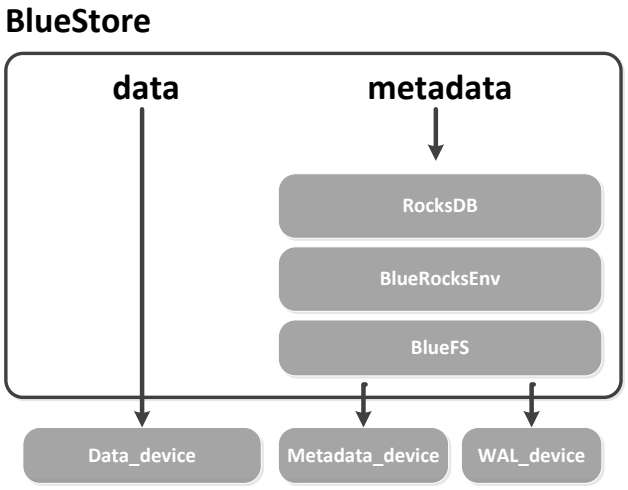

图 2-8 BlueStore 工作原理图

2.3.4 Cache Tier 机制

Cache Tier 是 Ceph 提供的一种数据分层缓存机制，不同于 2.2 节所介绍的 Flashcache，Flashcache 工作在硬盘驱动上，位于对象存储引擎的下一层，属于硬盘级的缓存技术，而 Cache Tier 工作在对象存储引擎的上层，属于存储池级的缓存技术。

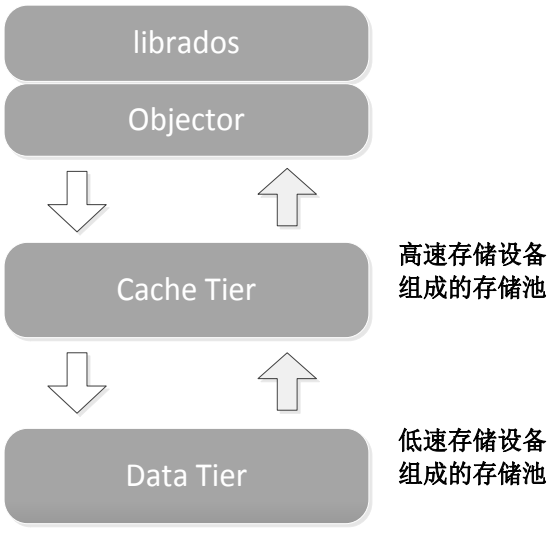

图 2-9 Cache Tier 工作原理图

Ceph 对集群中的存储资源进行池化管理，所谓池化就是将具有相同约束条件的存储资源统称为存储池，这些约束条件包括上文提及的 Object 的最大大小、集群拓扑、数据分布策略等，客户端通过存储池这一逻辑概念对底层具有相同

约束的所有存储资源进行访问。Cache Tier 将整个存储系统按照存储介质的性能分为高速缓存池和后端低速存储池，而后通过相应算法控制数据对象的读写流程，使访问频率高的数据对象存储在高速缓存池中，而将全量数据存储在后端低速存储池中，其工作流程如图 2-9 所示。

Cache Tier 通过缓存层代理实现对读写流程的控制，其主要提供两种功能：刷写(Flushing)与回刷(Evicting)。刷写用以识别内容发生变化的数据对象，并将它们发送到后端低速存储池；回刷用以识别高速缓存池中不经常使用的数据对象并将其从高速缓存池中移除，为访问频率高的数据对象腾出存储空间。

刷写主要通过内容发生变化的数据对象在缓存池中的空间占比(又称水位线)来实现。回刷除考虑空间占比外还设计了相应的淘汰规则，其会采用一种类似 LRU 算法的机制，根据数据对象的访问频率为数据对象定义一个热度值，然后将该热度值与热度阈值进行比较，然后进行淘汰。热度阈值的计算方式如式 2-1 所示。$Ratio_{space}$ 表示当前高速缓存池空间使用比，$waterLine_{eli}$ 表示设计的触发刷写的水位线。

$$threshold = \frac{Ratio_{space} - waterLine_{eli}}{100\% - waterLiner_{eli}} \tag{2-1}$$

Cache Tier 总体控制流程与 Flashcache 类似，但与 Flashcache 不同的是 Cache Tier 工作在对象存储引擎之上，而不是设备驱动之上，其所处理的数据单元也不再是裸数据块，而是携带元数据信息的数据对象。虽然 Ceche Tier 刷写的 IO 路径很长，但 Cache Tier 中所使用的淘汰算法可以依据的信息更多，理论上可以通过更高效智能的淘汰算法以保证更高的缓存命中率，减少数据刷写次数，使整体性能获得大幅提升。

目前 Cache Tier 采用的仍是类似 LRU 算法的基于频率估计概率的淘汰规则，并未充分利用数据对象所携带的元数据信息，因此实际使用时缓存命中率较低。较低的缓存命中率以及较长的 IO 路径，使 Cache Tier 性能表现较差，因此很少应用在实际生产环境中，所以对 Cache Tier 的淘汰算法展开优化研究存在一定的现实意义与价值。

2.4 本章小结

本章首先研究了两种典型的硬盘设备，解释了固态硬盘适合处理小块随机IO，但不适合作为全量数据存储介质的原因。接着分析了 Flashcache 技术的工作原理，为后文硬盘级方案设计奠定基础。最后对 Ceph 的读写流程及 Cache Tier 进行分析，明确了其读写流程中所涉及数据对象的种类及各自的特点以及其 Cache Tier 机制中所存在的缺陷。本章理论研究所得出的主要结论如下：

(1) 机械硬盘不适合处理小块 IO 及随机 IO，但能用于存储全量数据；固态硬盘适合处理各种 IO，但由于成本原因，使用时需要考虑数据整体容量；

(2) 工作在 write back 模式下的 flashcache 技术，可用于提升机械硬盘的读写性能；

(3) Ceph 中的 OSD 所要处理的数据包括 3 种：数据单元，具有数据量大，数据长度有限甚至很小、随机程度大的特点；元数据，具有数据量小，访问频繁的特点；日志数据，具有数据量小，访问频繁的特点；

(4) Ceph 中的 Cache Tier 机制所采用的淘汰算法并未充分利用数据对象所携带的元数据信息，因此理论上有优化的空间；

3 硬盘级混合存储方案设计与实现

Ceph 的 BlueStore 对象存储引擎是为克服传统 FileStore 对象存储引擎性能缺陷而推出的新型存储引擎，自推出之后便迅速取代了传统的 FileStore 对象存储引擎，并普遍应用在 Ceph 工程的实践中，因此本章基于第二章所分析的固态硬盘与机械硬盘的优势与缺陷、Flashcache 技术以及 Ceph 数据读写过程中 OSD 所处理的数据的特点，独立设计了完整的，高可用且易管理的混合存储方案以提高 BlueStore 存储引擎的读写性能。

3.1 优化方案的设计原理

基于 BlueStore 对象存储引擎的 OSD 上存放的数据主要可以分为两类：数据和元数据，因此在硬盘级优化方案设计时，需同时考虑数据的读写特征以及元数据的读写特征。

BlueStore 的元数据是基于 Rocksdb 进行管理的。 Rocksdb 在使用时会产生两种数据：数据对象的元数据键值对、保证数据一致性的日志数据。由 2.3.3 节的分析，元数据以及日志数据的访问具有数据总量小，单次访问数据量小，但访问频率高的特点。由于在 Ceph 的数据映射过程中，存在对数据对象进行 hash 散列的过程，因此 BlueStore 所处理的数据的连续性会有所破坏，相应的元数据的连续性也会有所破坏。故 BlueStore 中 Rocksdb 数据的读写特征可概括为：数据总量小、单次访问数据量小、连续性弱、访问频率高。在大量小块数据对象读写的场景下，Rocksdb 的这种读写特性的表现更为明显，如果采用机械硬盘存储性能将会表现极低，虽然固态硬盘成本高，但考虑到元数据的数据量相对数据来说很小，因此可以使用固态硬盘来存储这些元数据信息。

由 2.3.3 节可知，BlueStore 所处理的数据对象所携带的数据具有数据量大、数据访问离散程度较高、单次数据访问量有限甚至很小的特点。虽然固态硬盘在处理连续性较弱的数据读写行性能衰减程度较低，但由于数据所占用存储空间大，直接采用固态硬盘来存放数据成本将带来巨大的成本开销。通过 Flashcache 技术将高速固态硬盘作为固态硬盘的缓存，可以在低成本的条件下，合并小块数据增强磁盘数据写入的连续性，降低机械硬盘小块随机写时的性能

损耗，还能将尽可能多的用户访问转移到高速固态硬盘，提高整体数据读写的性能。因此采用固态硬盘分区使用及 Flashcache 技术的方式实现对数据的混合存储即能节省成本，还能提高 BlueStore 的读写性能。

综上，本文设计的基于 BlueStore 的 OSD 硬盘结构如图 3-1 所示，Rocksdb 所管理的存储空间分为存放键值对的区域以及 wal(write ahead log)区域，此两个数据存储区都使用固态硬盘(SSD)作为存储介质，对于 BlueStore 的数据存放区用通过 Flashcache 创建的固态硬盘和机械硬盘的聚合设备作为存储介质。

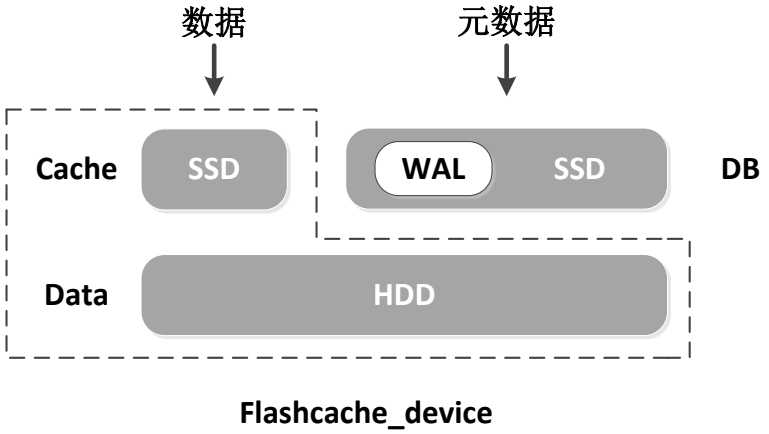

图 3-1 磁盘级优化方案原理图

3.2 硬盘管理组件的设计

在分布式存储系统中，工作在硬盘级的混合存储方案将增加用户的管理难度，为了方便用户对 Ceph 集群中存储服务器上的存储资源进行管理，本文基于设计原理设计了 OSD_manager 组件。该组件工作在集群中的各存储服务器上，用以管理和分配存储服务器上固态硬盘的空间，以及 OSD 的创建与删除过程。

固态硬盘服务于存储服务器上的机械硬盘，由于服务器上硬盘数量有限，OSD_manager 所需管理的固态硬盘信息数据量较小，因此硬盘管理信息持久存储在自定义格式的 osd_device.conf 文件中。为节约 OSD 创建和回收时间，OSD_manager 支持并发操作。OSD_manager 作为服务运行在系统后台，用户通过 OSD_client 与 OSD_manager 进行交互。OSD_client 用户交互工具放置在 Linux 系统的/usr/sbin 目录下，用户可以命令的形式进行使用。对于用户输入的命令及参数，OSD_client 会通过 unix_domain_socket 传递给 OSD_manager，OSD_manager 会依据用户请求创建相应的工作线程并做线程间的互斥控制。

用户通过 OSD_manager 去使用服务器上的固态硬盘，固态硬盘空间的分配及回收由 OSD_manager 自动完成，用户不感知对固态硬盘的使用过程，OSD_manager 的工作原理如图 3-2 所示。

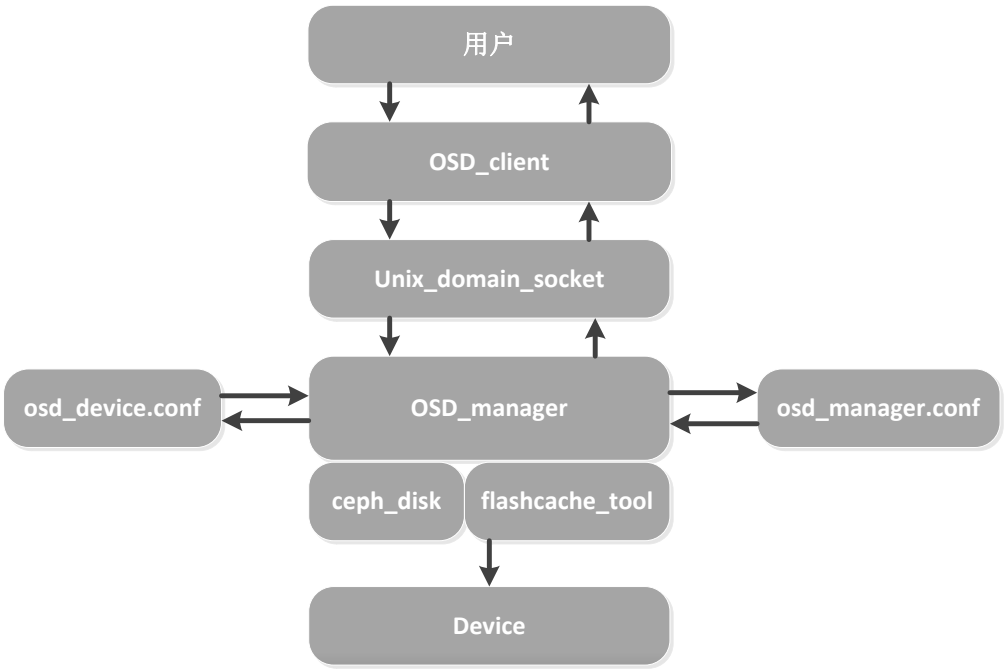

图 3-2 OSD_manager 工作原理图

在实际工程中使用时，存储服务器上的硬盘除系统盘外，其它硬盘都用来提供存储服务，为了保证性能，同时方便存储系统的整体维护，存储服务器都是同构的，即在同一个物理资源池中的硬盘盘大小一般都是相同的，每台服务器上的硬盘种类及数量分布也都是相同的。鉴于物理资源池中存储服务器的这种特征，OSD_manager 在服务启动时会自动依据服务器的硬件信息，对固态硬盘空间进行规划，并将规划结果记录到 osd_device.conf 中。

在完成对固态硬盘空间规划后，OSD_manager 会依据用户指定的机械盘以及 osd_device.conf 文件记录的分区结果，给该 OSD 分配固态硬盘空间，然后通过 flashcache_tool 创建聚合设备，通过 ceph_disk 创建 OSD。当用户需要删除 OSD 时，OSD_manager 会依据用户指定的 OSD 的 ID 将相应 OSD 从 RAODS 集群中踢出，通过 flashcache_tool 删除聚合设别，通过 ceph_disk 清理硬盘空间，并回收相应的固态硬盘空间。回收过程并不会删除相应的固态硬盘分区，而是修改 osd_device.conf 相应字段表示该固态硬盘目前未被使用。

3.3 优化方案的实现

本节主要阐述硬盘管理组件 OSD_manager 核心功能的实现过程，包括固态硬盘空间默认划分、固态硬盘空间管理、OSD 的创建以及 OSD 的删除。

3.3.1 固态硬盘空间默认划分

在存储服务器上，固态硬盘数少于机械硬盘数，因此需要对固态硬盘进行分区，使数量较少的固态硬盘可以服务存储服务器上的所有机械硬盘。固态硬盘的分区包括三种类型：数据缓存分区、DB 分区、WAL 分区。数据缓存分区用以给机械硬盘做缓存空间，DB 分区用以存放 RocksDB 中的元数据键值对，WAL 则用以存放 RocksDB 所产生的日志信息。

RocksDB 是基于 LSM 树(Log-Structured Merge Tree)实现的，其工作原理如图 3-3 所示。RocksDB 对存储空间进行分层管理，其中位于 C0 层的数据存储空间位于内存中，其余层的存储空间则位于硬盘设备上。WAL 空间用于保证 C0 到 C1 过程中的数据一致性。C0 层的数据分为两种，一种是可正常接收写入请求的 activate memtable，另一种则是不可修改的 immutable memetable。当写请求到来时，首先将写入请求操作添加到日志中，接下来把数据写往 activate memtable 中，当 activate memtable 满后，就将 activate memtable 切换为不可更改的 immutable memtable。处于 immutable memtable 的数据将被写往 C1 层，在 immutable memtable 中所有数据都刷写完毕后，WAL 中关于 immutable memtable 上所有操作的日志记录将被被清除，当数据写往 C1 层后，会触发数据合并操作，用以将 C0 层 immutable memtable 的数据合并到 C1 中，当 C1 层写满后，则会继续向下层合并。假如写请求所修改的数据不在 C0 层，那么会先将数据写往 C0 层，最终的数据修改在逐层合并过程中完成。当读请求到来时，首先会对 C0 层的数据进行检索，如果没有，则逐层向下检索，直至找到为止。

由上文分析可知，WAL 空间容量大小取决于 C0 层内存空间的大小。RocksDB 通过如下 3 个参数控制 C0 层的空间容量：

(1) 通过 write_buffer_size 控制，write_buffer_size 指定了 C0 层的最大空间容量，如果超过该容量，则会将 C0 层的数据写往 C1 层；

(2) 通过 max_total_wal_size 控制，max_total_wal_size 指定了 WAL 空间的大小，如果超过该容量，则会将 C0 层的数据写往 C1 层；

(3) 通过 write_buffer_size 与 max_write_buffer_number 参数共同控制，write_buffer_size 指定了单个 activate memtable 的大小，如果超过该值，则该 activate memtable 将转为 immutable memtable；max_write_buffer_number 指定了 memtable 总个数的最大值，如果当前 activate memtable 被写满，且 activate memtable 和 immutable memtable 的个数总和超过该值，则会停止写入，并将数据写往 C1 层；

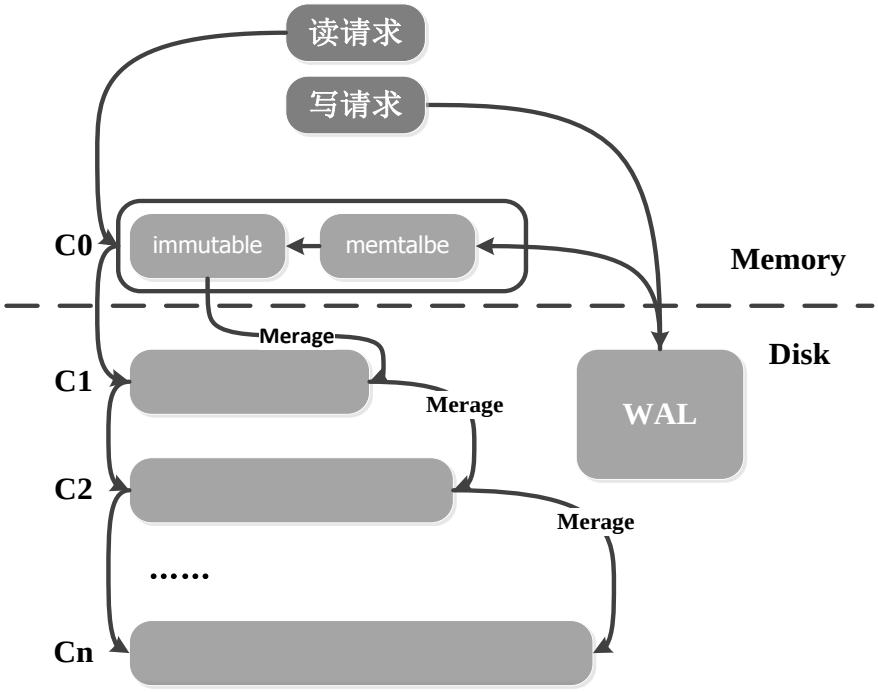

图 3-3 RocksDB 工作原理图

控制策略(1)、(2)属于强制容量控制，控制策略(3)则是基于业务逻辑进行的容量控制，因此本文课题基于控制策略(3)设定 WAL 分区空间的默认容量。在 Ceph 关于 OSD 配置中提供了 bluestore_rocksdb_options 配置项，其中记录有 max_write_buffer_number 以及 write_buffer_size 的取值，OSD_manager 服务在启动时会向 mon 节点查询 bluestore_rocksdb_options 的配置，并将该配置项中的上述两个参数的乘积作为 WAL 分区空间容量的默认值。

关于 DB 空间的容量，Ceph 社区官方文档关于 BlueStore 配置说明里指出 DB 空间容量不能小于 OSD 数据存放空间的 4%。由于在 Ceph 集群中存储服务

一般器都是同构的，因此 DB 分区空间容量默认设置为存储服务器非系统盘机械硬盘空间容量均值的 4%。

在分配完 WAL 空间和 DB 空间后，余下的固态硬盘空间均分至除系统盘外的所有机械硬盘。

3.3.2 固态硬盘空间管理模块

OSD_manager 整体启动流程如图 3-4 所示。为方便用户灵活设置分区大小，OSD_manager 服务在启动时会自动生成配置文件 OSD_manager.conf，其中记录取值为 3.3.1 所设定的默认空间容量的 wal_size 和 db_size 两个字段，用以用户自定义 WAL 空间大小及 DB 空间大小。

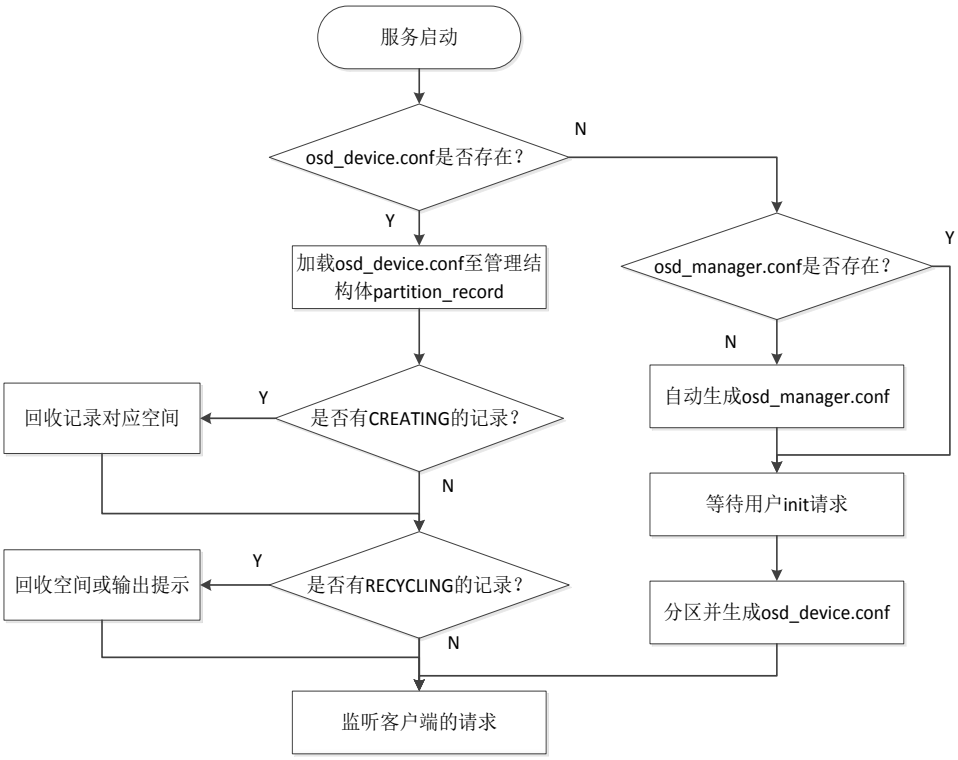

图 3-4 OSD_manager 启动流程图

OSD_manager 将服务于同一块机械硬盘的 WAL 分区、DB 分区以及数据缓存分区称为一组分区，并以组为单位进行固态硬盘的空间管理。当 OSD_manager 收到 OSD_client 传来的 init 命令后，OSD_manager 将读取 OSD_manager.conf 文件，并以组为单位创建分区，分区创建过程中，同样会以组为单位将创建的分区信息记录到全局变量 record 中，record 是类型为 partition_record 的数组，在分区创建完成后，record 所记录的信息会持久化存储到 osd_device.conf 文件中。partition_record 结构体定义如下。

```
struct  partition_record{
    string      ssd_device;
    int         group_id;
    int         db_partition_num;
    int         wal_partition_num;
    int         cache_partition_num;
    string      state=UNUSED;
}
```

结构体中的 ssd_device 表示该记录所在固态硬盘的盘符；group_id 表示该分区属于第几组分区；db_partition_num 表示该分组中的 db 分区的分区号；wal_partition_num 表示该分组中 wal 分区的分区号；cache_partition_num 表示该分组中数据缓存分区的分区号；state 表示该分组当前状态，取值为“UNUSED”、“USING”、“CREATING”、“RECYCLING”，依次对应“未使用”、“正在使用”、“创建中”、“回收中”这 4 种状态。OSD_manager 通过 state 字段实现多创建或删除过程的互斥控制，以及服务异常停止后的状态恢复。

为应对服务异常停止场景下的故障恢复，OSD_manager 服务启动时，会检查 osd_device.conf 文件是否存在。如果 osd_device.conf 文件不存在，OSD_manager 将会初始生成一个 osd_manager.conf 文件，并等待用户的 init 命令。如果存在则会进行服务是否异常终止的检查。

在进行异常终止检查时，OSD_manager 会先读取 osd_device.conf 文件并加载至内存的 record 中。由于创建和删除操作之前会修改 status 字段，因此 OSD_manager 会将 status 为“CREATING”或“RECYCLING”的记录视为异常记录，并进行相应处理。由于创建过程异常终止，不会影响 Ceph 整体集群的健康状态，因此对于 status 值为“CREATING”的记录，OSD_manager 会进行相应空间回收，回收完成后将 status 修改为“UNUSED”并同步到 osd_device.conf 文件。由于删除过程异常终止，该硬盘对应 OSD 可能正处于工作状态，因此对于该类记录，会输出提示给相应用户，并将 status 修改为“USING”并更新至 osd_device.conf 文件中；对于未处于工作状态的 OSD，则会继续回收相应空间，并将 status 修改为“UNUSED”并更新至 osd_device.conf 文件。

3.3.3 OSD 创建模块

OSD_manager 创建 OSD 的整体流程如图 3-5 所示。用户在创建 OSD 时，会将 create 命令及相应机械硬盘的盘符传递给 OSD_manager。OSD_manager 在收到 create 操作和机械硬盘盘符后，会先对机械硬盘的健康状态进行检查，若发现机械硬盘有坏道或其它故障，则会将相应提示信息返回给用户，并终止创建过程。

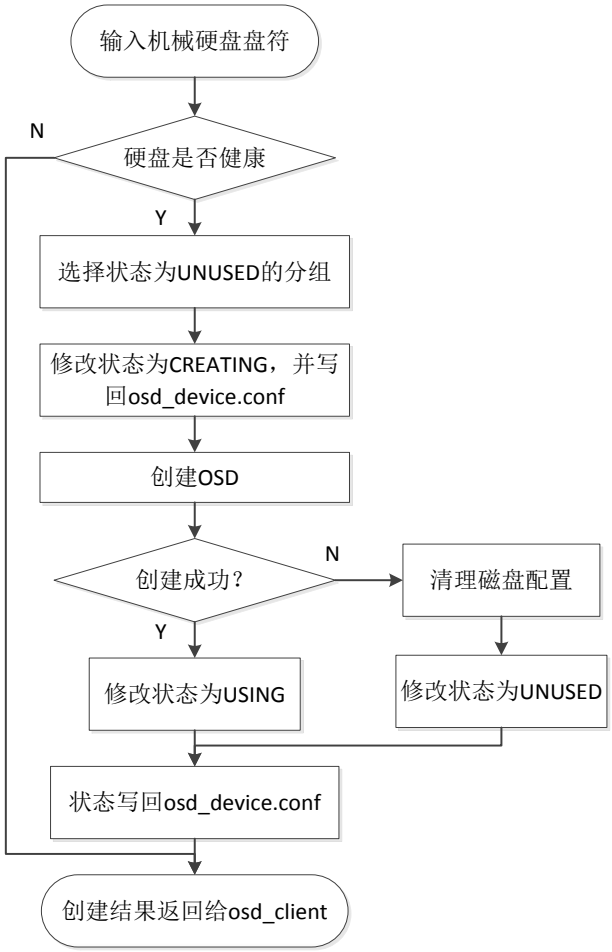

图 3-5 OSD 创建流程图

为了避免多线任务中，同一分区组被分配给多个创建线程，因此在创建时会选择 record 中 status 为“UNUSED”的分区组进行使用，同时修改 record 中的 status 字段为“USING”避免被其它并发线程同时处理而导致异常。由于创建过程中，可能存在断电或系统故障等场景，导致创建过程异常终止，为了能在 OSD_manager 服务下次启动时自动将创建过程异常终止的硬盘恢复到创建之前的状态，在正式开始创建之前，会将 status 同步更新到 osd_device.conf 文件中。在完成这些操作后，OSD_manager 才会开始 OSD 的创建过程，如果创建失败，

会回收相应硬盘空间，然后将 status 修改为“UNUSED”并同步更新到 osd_device.conf 文件，然后返回提示信息给用户；若创建成功，会将 status 修改为“USING”，然后同步更新到 osd_device.conf 文件，并返回提示信息给用户。其工作流程如图 3-5 所示。

在进行 OSD 创建时，为了避免系统可能出现的盘符漂移问题，会通过创建分区的方式为机械硬盘生成一个唯一的 UUID 标识，然后调用 flashcache_tool 工具，通过指定该机械硬盘的 UUID 路径以及分区组中的 cache 分区的 UUID 路径创建聚合设备。由于 ceph_disk 基于 LVM 去使用硬盘资源，会对各设备自动生成相应的 UUID，因此指定数据区的盘符，WAL 分区的盘符以及 DB 分区的盘符即可创建 OSD。

3.3.4 OSD 删除模块

用户在删除 OSD 时，会将 delete 命令以及相应 OSD 的 ID 传递给 OSD_manager。OSD 的 ID 的合法性检查在 osd_client 端完成，osd_client 会扫描本地所有的 OSD 的 ID，如果用户输入的 ID 非法则会直接提示用户，而不会传递给后端的 OSD_manager 服务。

删除 OSD 可能会导致数据丢失，因此 OSD_manager 在收到 delete 操作和 OSD 的 ID 后会读取 ceph 的配置文件获取 mon 节点的 IP，然后通过 SSH 的方式登陆到 mon 节点检查集群状态。由于通过 SSH 的方式连入 mon 节点，因此需要事先给存储集群中的各存储节点配置免密登陆。当集群状态异常时，线程会一直查询集群状态，直到集群状态健康，才开始后续删除操作。集群状态健康后，OSD_manager 会查询该 OSD 所使用的硬盘设备，并确定该 OSD 所对应的分区组，然后修改分区组的状态为“RECYCLING”保证线程互斥，并同步到 osd_device.conf 文件，方便操作异常终止后的自动恢复。配置文件修改完成后，OSD_manager 正式开始执行 OSD 的删除操作。若删除成功，OSD_manager 会将 status 字段修改为“UNUSED”并同步到 osd_device.conf 文件；若删除失败，为确保数据安全， OSD_manager 不会自动去处理异常，而是直接将异常信息输出给用户，由用户依据返回的异常信息，手动去删除该 OSD。OSD_manager 执行 OSD 删除的整体流程如图 3-6 所示。

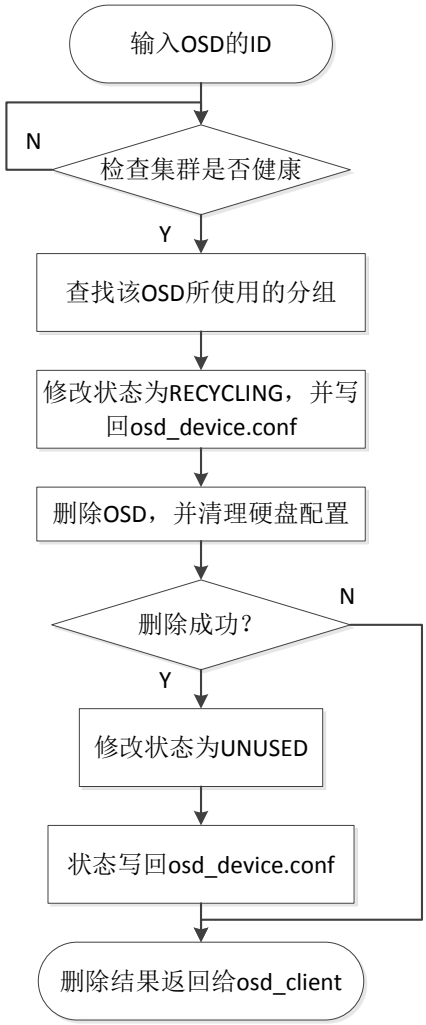

图 3-6 OSD 删除流程图

在进行 OSD 删除时，OSD_manager 会先将该 osd 踢出 ceph 集群，使其开始数据迁移和恢复，此时集群状态显示“WARNING”，会阻塞其它的删除线程，避免并发删除可能导致的数据丢失。踢出后，osd_manager 会清理该 OSD 在 ceph 集群中其它配置，并将其彻底从集群中删除。在这些操作完成后，OSD_manager 会停止 OSD 服务，并卸载其所挂载的聚合设备，而后通过 flashcache_tool 工具删除聚合设备，并修改 osd_device.conf 文件中相关分区组的状态为“UNUSED”，表示该分区组已被回收。

由于在删除前会进行集群状态的检查操作，当有 OSD 正在删除时，会有一段时间集群状态处于“WARNING”状态，因此虽然 OSD_manager 支持多并发，但删除操作线程之间会由于集群状态非健康而阻塞，所以实际上删除操作仍然是串行执行的，串行删除虽然牺牲了并发性能，但保证了数据安全，避免数据丢失。

3.4 本章小结

本章基于第二章的理论基础，设计了 Ceph 硬盘级混合存储优化方案，优化方案的目的是在固态硬盘资源有限的前提下，尽可能合理分配固态硬盘资源，降低因机械硬盘物理特性限制而导致的读写性能衰减。由于对固态硬盘资源划分和管理较为复杂，为方便用户对固态硬盘资源进行管理和使用，本章依据优化方案设计并实现了用以管理固态硬盘的组件 OSD_manager。通过 OSD_manage 组件用户可以在不感知固态硬盘的前提下使用和回收 OSD 所占用固态硬盘资源。为了提升 OSD 的创建效率，OSD_manager 以后端服务的方式为用户提供服务，并设计相应内存结构 record 以支持多并发创建操作。为应对服务异常停止场景，OSD_manager 通过持久存储的文件 osd_device.conf 及相应机制来进行异常处理。该优化方案一方面可以提升集群整体的性能，另一方面增加了大规模存储场景下硬盘级混合存储设备的可管理性。

4 Cache Tier 淘汰算法优化设计

Cache Tier 虽然工作在存储池级，但其采用的仍是类似 LRU 算法的基于频率估计概率的淘汰规则。由于频率估计概率存在误差，采用该方式设计淘汰规则，缓存命中率性能有限，且很难达到理论极限。为进一步提升缓存命中率，本章充分考虑 Cache Tier 数据对象可以携带更多信息的特点，并以此为基础设计了一种新的淘汰规则。本章首先对云计算场景下的用户访问行文进行分析，分析结果表明云计算场景下用户访问行为带有高斯分布的特征。然后分析了 LRU 类算法在高斯分布场景下的性能缺陷。最后针对 LRU 类算法的缺陷，设计了一种不同于 LRU 类算法的基于高斯分布形态特征的缓存淘汰算法。

4.1 理论分析

据 2018 年 Ceph 用户调查结果显示，有 48.94%的用户将 Ceph 应用在云计算场景中，且有 84.04%的用户通过 RBD 接口对上层应用提供服务。因此本节主要对工作在“基础设施即服务”模式下的云计算场景的用户行为进行分析，并依据分析结果对 LRU 类算法进行详细分析。

4.1.1 云计算场景下的用户行为分析

在自然科学和社会科学领域有两种典型的数量分布形式，一种是钟形的高斯分布，另一种是 L 型的幂律分布[21]。

文献[21]的研究成果表明，如果随机事件发生的概率与该随机事件目前已发生的频率成正比，则该随机变量更容易表现出幂律分布。通俗解释，即假如事物之间存在关联，且这种关联表现出事物的聚合，则通常对外表现出幂律分布。由于同类型事物上的用户群体之间有很大的相关性，容易因事物对用户的价值等因素，使用户在某类事物的某些事物表现出聚合现象，进而表现出幂律分布的特征。文献[22]、文献[23]以及文献[24]等关于用户在同类事物上的行为研究结果直接或间接证明了这一理论的正确性。

文献[25]的研究表明，如果决定某一随机变量结果的是大量微小的、独立的随机因素之和，并且每一个因素的单独作用相对均匀的小，没有一种因素可

起到压倒一切的主导作用，那么这个随机变量一般近似于高斯分布。通俗解释，即假如事物之间信息熵很大甚至相互独立，则通常容易对外表现出高斯分布。高斯分布往往存在于事物种类繁多，用户群体复杂的场景中。文献[26]对承载业务复杂的用户网络行为进行研究，结果表明大学校园网的峰值流量服从渐近高斯分布。文献[27]和文献[28]关于边缘缓存的研究中，使用泊松分布来仿真模拟用户的访问频率，由数学推导可知一定条件下的泊松分布可以简化为高斯分布。除学术研究外，工程界很多模拟测试工具底层也都支持基于高斯分布来模拟用户行为，如文献[29]和文献[30]中所使用的 fio 硬盘测试工具，其底层随机数生成器也支持通过高斯分布来生成随机数。此类学术界以及工程界的研究成功及生产应用在一定程度上证明了该理论的正确性。

基础设施即服务(IaaS, Infrastructure as a Service)是云计算的一种基本交付模式，该模式下云计算只需要给上层业务提供硬件服务即可，本身并不感知上层业务类型，但要求能够支撑异构多业务体系[31]。该场景下，分布式对象存储主要用以为工作在云计算服务上的应用软件提供数据存储服务。一方面，后端存储所服务的上层应用软件众多，各应用软件应用场景及所服务的对象存在很大区别，因此不同应用软件之间数据关联性较弱；另一方面，应用软件有其自身系统架构，系统内各模块之间往往功能独立，除模块之间关联的信息外，各模块内部信息之间关联性也较弱。两者共同作用，导致后端存储所存储的数据单元之间存在较大的信息熵。该场景下的分布式对象存储后端存储所存储的数据对象的信息熵特点满足文献[25]的研究结论的前置条件，因此理论上用户对这些数据单元的访问行为更容易表现出高斯分布的特征。

为验证上述结论的正确性，本文课题对实际工程中某私有云平台上各虚拟机对后端存储的读写情况进行了研究分析。由于实验条件的约束，分析只能从宏观上进行，无法详细捕获业务信息，因此所取得的结果有限。该私有云平台主要包含计算型业务，转发型业务，接入型业务及存储型业务，通过对该平台虚拟机进行统计分析，发现不同核数虚拟机磁盘读写频率如图 4-1 所示，形态上具有“中间高两边低”的高斯形态特征。虽结果有限，但在一定程度上反映了工作在云计算平台上的软件对后端数据访问频率符合高斯分布的行为特征，一定程度上表明上述理论分析所得结论的正确性。

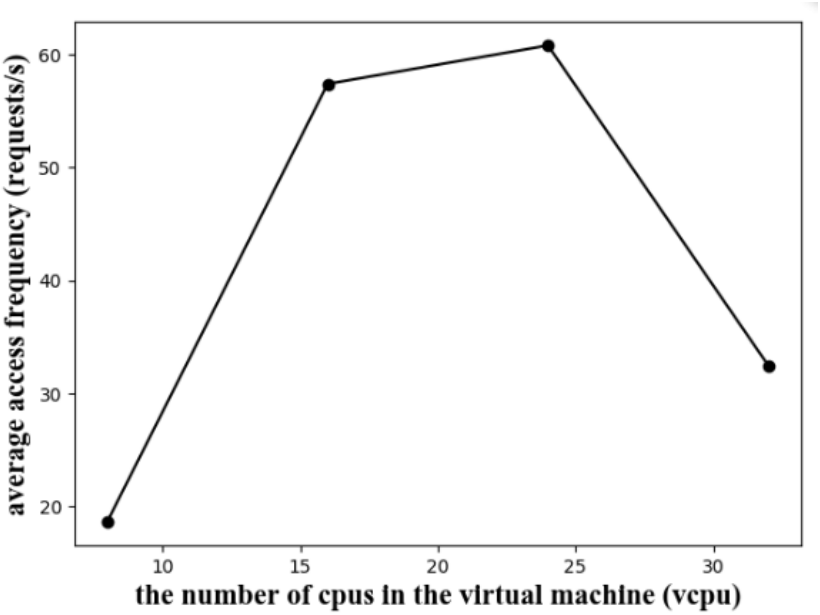

图 4-1 不同核数虚拟机磁盘读写统计结果

4.1.2 LRU 类算法性能分析

淘汰算法的种类有很多，出于对算法复杂度及命中率的综合考虑，LRU 类算法是目前使用最为广泛的淘汰算法。LRU 算法[32][33][34]认为最近被访问的块在将来更容易被访问，在进行数据块淘汰时优先淘汰缓存空间中最久没有被访问的数据块。由于这种机制，LRU 算法对冷数据突发性访问的抵抗性能较差，极端情况下可能会出现缓存命中率为 0 的情况。

为了缓解这类问题，产生了如 LRU-K[12]、2Q[13]、MQ[14]等通过设定访问频率阈值来分离冷热数据的方法，以及 LIRS[15]、LFRU[35]算法等优化概率量化方式以精确淘汰的方法。前者虽然可以避免冷数据对缓存空间的污染，但对数据的热度感知能力有限，无法有效区分温数据和热数据，当缓存空间有限，热度分布范围较广的场景，如用户访问符合两倍方差大于缓存空间的高斯分布场景，缓存命中率性能较差；后者虽然优化了频率计算方式，且可以通过相应数据结构减少计算量，但本质还是通过频率估计概率设计淘汰规则，所以还是会受到相应误差影响，虽然命中率往往会高于前者，但仍然达不到理论极限。

LRU 算法主要考虑的是数据最近一次访问到当前访问的用户访问次数 Δt，其频率估计计算方式如式(4-1)所示。LRU 算法认为用户所访问的数据块的访问频率越小，其被访问的概率越小，越应该被淘汰。LRU 算法这种频率估计概率的方式，虽然更容易通过数据结构实现计算过程，从而减少计算量，但由于取

样空间太小，该方式存在极大误差。尤其当冷数据突发访问时，该冷数据的Δt会经历一个由 0 到缓存空间大小的衰减过程，存在一段时间，LRU 算法会认为冷数据被访问的概率 f 较大，因此会淘汰部分访问概率比其高的数据，从而造成缓存空间的污染，命中率的下降。

$$f = \frac{1}{\Delta t} \times 100\% \tag{4-1}$$

LRU-K 算法[12]维护了一个用以记录数据访问历史的，长度为 $l_{history}$ 的队列。对于历史访问队列中访问次数超过 k 次的数据，通过 LRU 算法进行管理；对于访问次数低于 k 次的数据，则通过其它缓存策略管理，即 LRU-K 算法按如式(4-2)所示的方式计算频率阈值，通过该频率阈值来实现冷热数据分离。该算法增大了取样空间，频率估计概率会较 LRU 算法更为准确，所以命中率会高于 LRU 算法，但相应的空间开销也会变大，且该额外空间的开销越大，命中率性能越好。但额外空间是有限的，不可能无限大，所以 LRU-K 算法样本空间不可能无限增加，通过有限样本空间估计概率依旧存在误差，所以通过该方法缓存命中率理论上达不到理论极限。

$$P = \frac{k}{l_{history}} \times 100\% \tag{4-2}$$

2Q（Two Queues）算法[13]，将缓存空间的数据分为两部分进行管理，对于仅访问一次的缓存数据通过长度为l_{fifo}的 FIFO 进行管理，对于访问次数超过两次的缓存数据通过 LRU 进行管理，即 2Q 算法按如式(4-3)所示的方式计算频率阈值，通过频率阈值来分离冷热数据。由于较 LRU 算法而言，扩大了样本空间，所以概率估计比 LRU 算法更为准确，因而性能会优于 LRU 算法。虽然该算法空间开销小于 LRU-K 算法，但样本空间没有 LRU-K 算法灵活，最大不超过缓存空间容量，因此理论上命中率会不如 LRU-K 算法。

$$P = \frac{1}{l_{fifo}} \times 100\% \tag{4-3}$$

MQ（Multi Queue）算法[14]可以理解为 2Q 算法的改进算法。该算法维护了一个用以记录数据访问历史的队列，同时依据访问次数将缓存中的数据块划分到多个缓存队列中，各缓存队列通过 LRU 算法进行管理，在进行淘汰时，优先

从缓存访问频率低的缓存队列中进行淘汰。由于对空间多层划分，使热数据和温数据的估计较 2Q 算法更为精确，所以命中率性能较 2Q 算法好，但其也存在 2Q 算法所存在的问题，所以理论上命中率也不如 LRU-K 算法。

LFRU(Least Recently / Least Frequently Used)[35]算法重新定义了概率的量化方式，通过将缓存空间的数据单元的访问次数与两次访问之间的时间间隔做乘积，并将结果做为该数据单元被访问的概率量化指标设计淘汰规则。该算法认为该值越小的数据对象越应该被淘汰。该方式增加了淘汰过程的计算开销和空间开销，同时访问次数和时间间隔都是频率的一种表现，都存在误差，因此该算法理论上误差波动会很大，虽然命中率优于 LRU 算法，但不如 LRU-K 算法。

LIRS(Low Inter-reference Recency Set)算法[15]通过缓存数据最近两次的访问间隔参数 IRR(Inter-Reference Recency)和最近一次访问到当前时间内的数据访问量参数 R(Recency)来进行缓存块的管理。任意冷数据块的 IRR 值小于 R_{max} 就可以转换成热数据块，所有 R 值小于 R_{max} 的冷数据块可以保留在缓存中。R_{max} 本质也是频率阈值，所以 LIRS 算法实际上并不能单纯理解为通过重定义概率量化方式的优化算法，该算法除了重定义概率量化方式外，相应的也设定了相应的阈值过滤，所以可以理解为两种方式的结合。虽然 IRR 比 LRU 算法的概率估计更为准确，但仍不如 LRU-K 算法准确，所以理论上命中率也会不如 LRU-K 算法。

Cache Tier 所采用的淘汰机制原理上更接近于 LIRS 算法，其采用频率估计概率的方式存在误差，可能导致非冷数据被判断为冷数据，同时由于访问概率模型未知，人工设定阈值也可能存在偏差，因此采用频率阈值分割冷热数据的方式会导致缓存空间得不到最有效的利用。例如，数据 A 被访问的概率为 1/10，数据 B 被访问的概率为 1/5，数据 C 被访问的概率为 1/100，频率阈值设定为 1/100，缓存空间大小为 2，用户短期内按 ABC 的顺序进行访问，当用户访问到数据 C 时，由于缓存空间已满，C 不在缓存空间中，因此会淘汰掉最先进入缓存，但被访问概率更高的数据 A，从而导致缓存空间命中率的下降。

设连续型随机变量 X，X 的取值 x 按四舍五入取整后表示用户访问数据块的地址 $X\sim N(\mu,\sigma)$。若缓存空间大小为 2σ。LRU 改进算法设定的频率阈值有效过滤了地址在$(\mu-2\sigma, \mu+2\sigma)$外的冷数据，此时由于$(\mu-2\sigma, \mu-\sigma)$及$(\mu+\sigma, \mu+2\sigma)$区间的温数据的影响，缓存空间的命中率 P 的取值范围如式(4-4)所示。由此可知

LRU 及其改进算法，在用户访问符合高斯分布的场景下，命中率性能并不理想。这种命中率的劣化在缓存空间相对有限，用户数据访问的概率模型符合或近似符合高斯分布的场景下尤为明显。

$$\begin{cases} P \geq P\{X=x | x \in (\mu-2\sigma, \mu-\sigma) \cup (\mu+\sigma, \mu+2\sigma)\} \\ P \leq P\{X=x | x \in (\mu-\sigma, \mu+\sigma)\} \end{cases} \Rightarrow \begin{cases} P \geq 0.2718 \\ P \leq 0.6827 \end{cases} \quad (4\text{-}4)$$

4.2 基于高斯分布的缓存淘汰算法

随机事件发生的概率的大小除了频率这种微观上的估计方式外，宏观上基于概率分布的形态特征也能在一定程度上为两个随机事件出现的概率大小提供判断依据。概率分布的形态特征反映了随机事件发生的宏观规律，以此设计淘汰规则，不受样本空间大小的影响，且在相应分布应用场景下，命中率有达到理论极限的可能。基于上文理论分析的结果，本文设计了一种基于高斯分布形态特征的缓存淘汰算法。

4.2.1 算法原理及架构

假设通过对数据对象的统计分析已知用户对某维度数据对象的访问符合高斯分布并将该维度信息记录到数据对象的元数据中。算法将该维度信息称为数据对象的高斯淘汰因子，某时间段内所访问数据对象的高斯淘汰因子的均值称为高斯淘汰中心，高斯淘汰中心按式(4-5)进行计算。Gaussi 表示第 i 次访问的数据对象的高斯因子，total 表示该段时间内用户所访问的数据对象总数。

$$\overline{Gauss} = \frac{\sum_{i=1}^{total} Gauss_i^2}{total} \quad (4\text{-}5)$$

依据高斯分布的形态特征，用户所访问数据对象的高斯因子距高斯淘汰中心越近，被访问的概率越高。当缓存空间不足时，算法会优先从缓存池中淘汰高斯因子距高斯淘汰中心最远的数据对象，高斯因子距高斯淘汰中心按式(4-6)进行计算。结合高斯分布良好的对称性，采用平衡二叉树[36]来管理缓存空间。采用这种管理结构一方面可以依据用中位数估计均值的原理，用平衡二叉树的根节点取代高斯中心的均值计算过程，减少计算量；另一方面也可以绝对值的

计算和比较过程转化为寻找二叉树的最左或最右叶子节点的过程。故采用这种结构可以减小算法的时间开销和空间开销。

$$dis = \left| Gauss_i - \overline{Gauss} \right| \tag{4-6}$$

用户访问的兴趣热点存在时效性[37]，在用户访问符合高斯分布时，这种时效性在理论上表现为高斯因子所符合高斯分布的均值和方差的变化。据高斯分布的“3σ 原则”，方差变化会影响在固定大小缓存空间下算法命中率的理论极限，并不会导致算法失效，但均值变化会导致原有高斯淘汰中心失效，进而导致缓存空间命中率的劣化，因此算法引入稳态监控机制，来适应随时间推移可能导致的高斯淘汰中心的迁移。

算法整体过程如图 4-2 所示，图中 g 表示用户所请求的数据对象的高斯因子，gl 表示算法所维护的平衡二叉树中最左叶子节点的高斯因子，gr 表示算法所维护的平衡二叉树中最右叶子节点的高斯因子。流程中的“稳态监控”用以判断当前高斯淘汰中心是否处于稳态并生成相应的状态标记，该状态标记会作用于流程中的“缓存替换”过程。替换算法对稳态和非稳态会采用不同的策略进行缓存空间数据对象的替换。

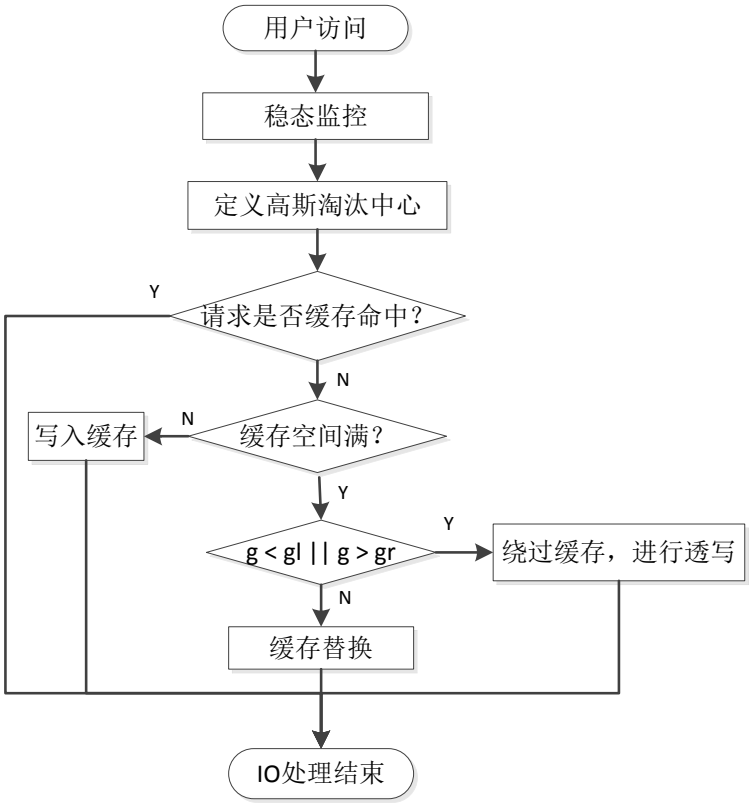

图 4-2 淘汰算法总体流程

4.2.2 算法中稳态监控机制的实现

通过仿真实验，测试了在不引入稳态监控条件下，当高斯中心发生迁移时，缓存命中率的变化情况。实验设定缓存空间大小为 500，以 1000 次用户访问为统计单元计算缓存实时命中率，比较了用户访问由 N(9750,500) 变为 N(10000,500)、N(10000,1000)及 N(10000,250)三种场景下缓存实时命中率的变化情况，结果如图 4-3 所示。实验结果表明：当用户访问所符合的高斯分布发生变化时，缓存命中率会出现瞬间的大幅度波动；当用户访问稳定符合某一高斯分布时，实时命中率存在波动性，且经实验统计表明波动幅度最大不超过 0.05。

基于实验统计结果，稳态监控通过判断缓存命中率的变动幅度是否超过给定阈值 0.05 来实现。算法以 1000 次用户访问为单位，计算实时命中率，并将最近 10 次实时命中率的前后差值记录到长度为 10 的 FIFO 链中作为监控窗口。为平滑稳定分布情况下缓存命中率的波动程度，进一步提高阈值判断的准确性，算法统计监控窗口中所有元素的和，若和的绝对值不大于阈值，则认为高斯中心处于稳态；若和的绝对值大于阈值，则认为高斯中心处于非稳定状态。

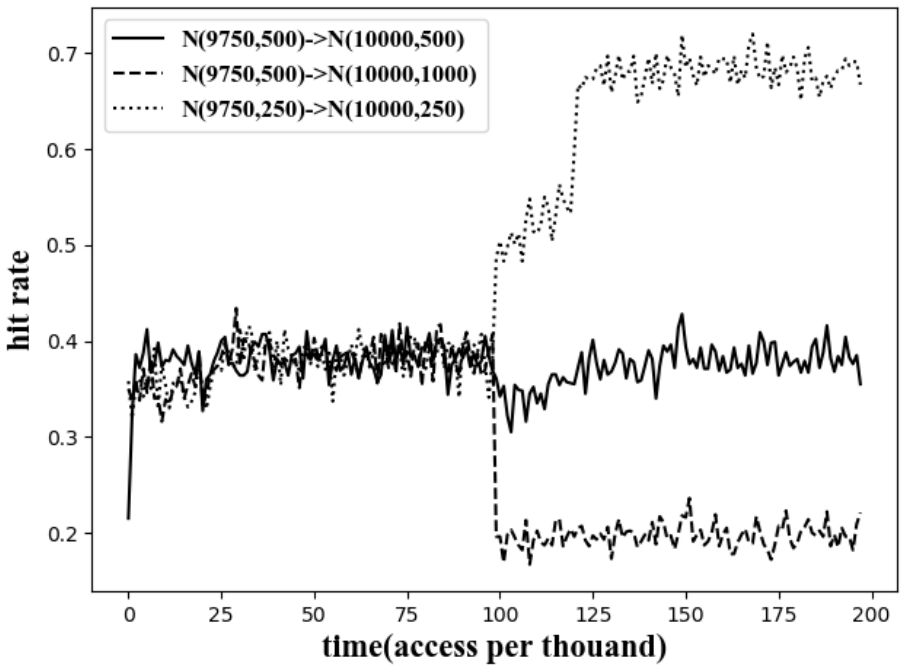

图 4-3 高斯分布迁移对算法实时命中率的影响

4.2.3 算法中淘汰过程的实现

结合高斯分布的对称性，为避免淘汰过程中大量的均值计算过程和绝对值计算比较过程，算法采用平衡二叉树来管理缓存空间。该实现方式一方面较 LRU 及其改进算法的线性检索缓存空间而言，检索速度更快；另一方面在寻找距离最远的块时只需要找到最左或最右叶子节点，再进行距离比较即可，有效降低了算法的时间复杂度。

在内存中通过结构体 AVLNode 来管理数据对象，AVLNode 的结构定义如下所示。

```
struct AVLNode{
    MetaData *     metadata;
    AVLNode *    parent = NULL;
    AVLNode *    left = NULL;
    AVLNode *    right = NULL;
    int    balance = 0;
    int    gauss;
}
```

结构体定义中的 metadata 表示数据对象的元数据信息；parent、left、right 分别用以指向该数据对象在平衡二叉树中的双亲节点、左孩子节点和右孩子节点；balance 则表示该数据对象所在节点在二叉树中的平衡因子，取值为{-2，-1,0,1,2}；gauss 表示该数据对象的高斯因子。

淘汰过程主要通过 reclaimer 函数实现， reclaimer 返回 true，表示数据对象写入缓存；返回 false 表示数据对象不写入缓存。reclaimer 函数作用于真正的存储池读写操作之前，用以依据淘汰规则修改相应内存管理的二叉树结构并将指导数据对象在存储池上读写的元数据信息写入对应 object 的 metadata 域。reclaimer 函数的伪代码如下所示。

```
bool reclaimer(Requst *request,AVLTree *avlTree)
{
    int gauss=request.dataObject.gauss
    AVLNode object=request.dataObject
    if avlTree.isFull()
        if avlTree.status==1
```

```
            if avlTree.isInLeft(gauss)
                avlTree.removeTheLeftMost(object)
                avlTree.insertTheElement(object)
                return true
            elif avlTree.isInRight(gauss)
                avlTree.removeTheRightMost(object)
                avlTree.insertTheElement(object)
                return true
            else
                return false
            end if
        else
            if gauss < avlTree.root.gauss
                avlTree.removeTheRightMost(object)
                avlTree.insertTheElement(object)
                return true
            else
                avlTree.removeTheLeftMost(object)
                avlTree.insertTheElement(object)
                return ture
            end if
        end if
    else
        avlTree.insertTheElement(object)
        return true
    end if
}
```

伪代码中所涉及的 AVLTree 类中的部分属性及方法说明如表 4-1 所示。

表 4-1 伪代码关键属性及方法说明

属性 | 方法	说 明
AVLNode* root	记录平衡二叉树的根节点
int status	status=1 表示处于平衡状态；status=0 表示不处于平衡状态
bool isFull()	判断缓存空间是否已满

表 4-1 伪代码关键属性及方法说明 (续)

bool isInleft(int gauss)	判断 gauss 是否小于根节点的高斯因子，且大于最左叶子节点的高斯因子
bool isInright(int gauss)	判断 gauss 是否小于根节点的高斯因子，且大于最左叶子节点的高斯因子
bool removeTheLeftMost(AVLNode object)	将二叉树根节点左子树上距离根节点最远的不带有与数据池不一致数据的节点删除，并将回收的缓存空间元数据信息写入到 object 中
bool removeTheRightMost(AVLNode object)	将二叉树根节点右子树上距离根节点最远的不带有与数据池不一致数据的节点删除，并将回收的缓存空间元数据信息写入到 object 中
bool insertTheElement(AVLNode object)	将 object 插入到平衡二叉树中

4.3 本章小结

为了提升 Cache Tier 的性能，本章从 Cache Tier 所采用的淘汰算法入手进行优化，目的是通过提升 Cache Tier 的命中率来提升 Ceph 集群整体性能。为克服 Cache Tier 所采用的类似 LRU 算法，通过频率估计概率设计淘汰规则而导致命中率受限的问题。本章首先对云计算场景下的用户行为进行分析，基于分析结果，设计了一种基于高斯分布的淘汰算法。该算法可以适应用户高斯分布随时间而发生迁移的过程，理论上命中率也优于 LRU 类算法。

5 实验与分析

5.1 硬盘级优化方案性能验证实验

为了验证本文提出的硬盘级优化方案的有效性，实验基于本文提出的硬盘级优化方案和 Ceph 原生的 OSD 部署方式分别搭建了两套实验平台，并设计实验对两个平台进行对比验证。

5.1.1 测试环境及配置

硬盘级优化方案性能环境拓扑如图 5-1 所示。整个集群由 3 台存储服务器及一台客户端服务器构成，整个集群网络由 3 个网络平面组成，管理网平面用以用户外部访问服务器集群；Ceph Public 网络作为 Ceph 集群的前端网络，用以为 Ceph 客户端提供集群访问服务；Ceph Cluster 网络作为 Ceph 集群的后端网络，用以 Ceph 集群内部 OSD 心跳、数据复制、恢复、迁移等。为了不使网络成为 Ceph 集群性能的性能瓶颈，3 个网络都使用万兆网络。

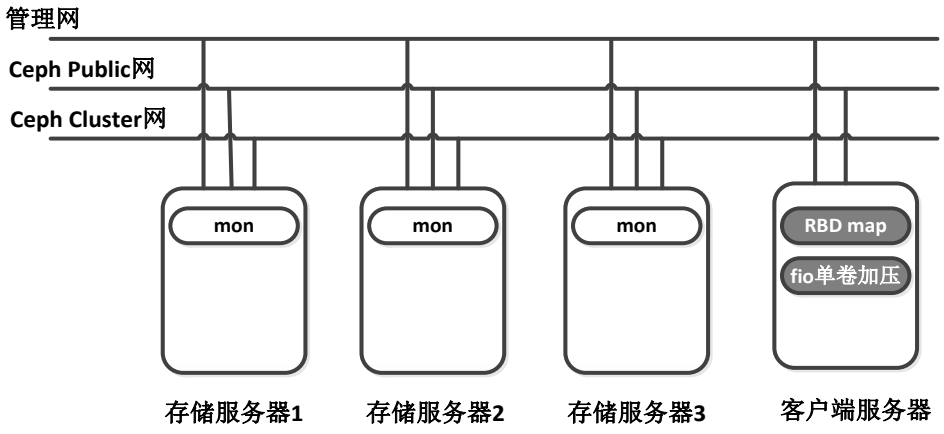

图 5-1 集群拓扑图

实验所使用的物理服务器硬件配置如表 5-1 所示。为了尽可能体现机械硬盘和固态硬盘的性能，实验选取采用 SAS 接口的机械硬盘和采用 NVMe 接口的固态硬盘。为了避免 RAID 因素对硬盘性能的影响，SAS3008 配置为直通模式。

表 5-1 服务器硬件配置表

CPU	2 x Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz
主板	超威 X11DPU
内存	8 x 16G DDR4
网卡	4 x Intel Corporation Ethernet Controller X710 for 10GbE SFP+

表 5-1 服务器硬件配置表(续)

HBA 卡	2 x LSI Logic / Symbios Logic SAS3008
机械硬盘	4 x 1T 东芝 SAS
固态硬盘	1 x 2T Intel NVMe SSD

集群软件配置如表 5-2 所示。为了模拟各种硬盘访问场景，选取 fio 工具作为实验测试工具。为了避免其它优化参数对实验结果的影响，实验环境使用原生 mimic 版的 Ceph 进行搭建，flashcache 在使用时除工作模式配置为“write back”外，其余均保持默认配置。

表 5-2 服务器软件配置表

操作系统版本	CentOS Linux release 7.6.1810 (Core)
Ceph 版本	ceph version 13.2.6-4-g9537302 mimic
fio 版本	fio-3.7
flashcache 版本	flashcache-3.1.1

5.1.2 测试方法

实验在每个存储节点上部署 4 个 OSD，并基于 2+1 纠删码数据冗余策略创建存储池，然后基于该存储池创建一个与该存储池空间等容量的虚拟磁盘，用以给客户端提供存储服务。客户端服务器通过 Ceph 提供的 rbd map 命令将创建的虚拟磁盘映射到本地，然后利用 fio 工具对虚拟磁盘进行各种 IO 场景下的性能测试，通过测试单卷性能证明本文提出的方案对性能的提升程度。

实验主要分为小块随机场景和大块连续场景，小块随机场景用以体现虚拟磁盘的 IOPS 能力，大块连续场景用以体现虚拟磁盘的吞吐能力。实验定义小块数据大小为 4k，大块数据大小为 4M，为充分反映各种 IO 场景下本文设计的方案的性能，又将实验场景细化为如下十个场景：小块随机写、小块随机读、小块随机读写混合(读占 30%)、小块随机读写混合(读占 50%)、小块随机读写混合(读占 70%)、大块连续写、大块连续读、大块连续读写混合(读占 30%)、大块连续读写混合(读占 50%)、大块连续读写混合(70%)。各场景对应的 fio 命令如表 5-3 所示。

表 5-3 实验场景与 fio 命令对应表

小块随机写	fio -filename={device} -direct=1 -ioengine=libaio -iodepth 64 -thread -rw=randwrite -bs=4k -size=50G -numjobs=1 -group_reporting -name=mytest

表 5-3 实验场景与 fio 命令对应表(续)

小块随机读	fio -filename={device} -direct=1 -ioengine=libaio -iodepth 64 -thread -rw=randread -bs=4k -size=50G -numjobs=1 -group_reporting -name=mytest
小块随机读写混合（读占 30%）	fio -filename={device} -direct=1 -ioengine=libaio -iodepth 64 -thread -rw=randrw -rwmixread=30 -bs=4k -size=50G -numjobs=1 -group_reporting -name=mytest
小块随机读写混合（读占 50%）	fio -filename={device} -direct=1 -ioengine=libaio -iodepth 64 -thread -rw=randrw -rwmixread=50 -bs=4k -size=50G -numjobs=1 -group_reporting -name=mytest
小块随机读写混合（读占 70%）	fio -filename={device} -direct=1 -ioengine=libaio -iodepth 64 -thread -rw=randrw -rwmixread=70 -bs=4k -size=50G -numjobs=1 -group_reporting -name=mytest
大块连续写	fio -filename={device} -direct=1 -iodepth 64 -thread -rw=write -ioengine=libaio -bs=4M -size=50G -numjobs=1 -group_reporting -name=mytest
大块连续读	fio -filename={device} -direct=1 -iodepth 64 -thread -rw=read -ioengine=libaio -bs=4M -size=50G -numjobs=1 -group_reporting -name=mytest
大块连续读写混合（读占 30%）	fio -filename={device} -direct=1 -iodepth 64 -thread -rw=rw -rwmixread=30 -ioengine=libaio -bs=4M -size=50G -numjobs=1 -group_reporting -name=mytest
大块连续读写混合（读占 50%）	fio -filename={device} -direct=1 -iodepth 64 -thread -rw=rw -rwmixread=50 -ioengine=libaio -bs=4M -size=50G -numjobs=1 -group_reporting -name=mytest
大块连续读写混合（读占 70%）	fio -filename={device} -direct=1 -iodepth 64 -thread -rw=rw -rwmixread=70 -ioengine=libaio -bs=4M -size=50G -numjobs=1 -group_reporting -name=mytest

5.1.3 测试结果

小块随机场景下，优化前和优化后的 IOPS 性能测试结果如图 5-2 所示。由实验结果可以发现，采用本文提出的优化方案创建 OSD，Ceph 所提供的虚拟硬盘 IOPS 性能得到大幅提升：小块随机读场景，IOPS 性能提升 270%；小块随机写场景性能提升 1518%；混合读写(读占 70%)的场景，读写整体 IOPS 性能提升 1482%；混合读写(读占 50%)的场景，读写整体 IOPS 性能提升 1512%；混合读写(读占 30%)的场景，读写整体 IOPS 性能提升 2071%。

大块连续场景下，优化前和优化后的吞吐量性能测试结果如图 5-3 所示。由实验结果可以发现，采用本文提出的优化方案创建 OSD，Ceph 所提供的虚拟硬盘吞吐量性能得到大幅提升：大块连续读场景，吞吐量性能提升 558%；大块连续写场景，吞吐量性能提升 238%；大块连续混合读写(读占 30%)，读写整体

吞吐量提升 255%；大块连续混合读写(读占 50%)，读写整体吞吐量提升 142%；大块连续混合读写(读占 70%)，读写整体吞吐量性能提升 133%。

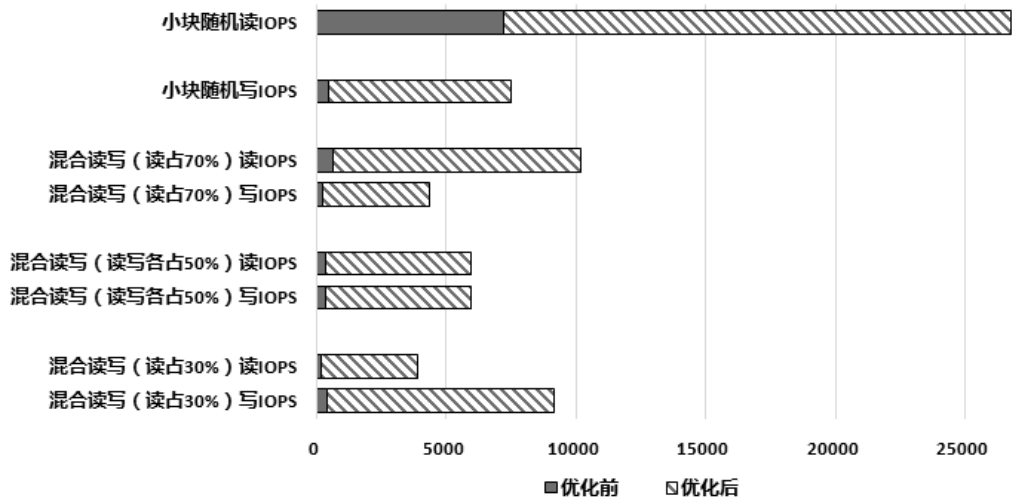

图 5-2 小块随机场景 IOPS 测试结果

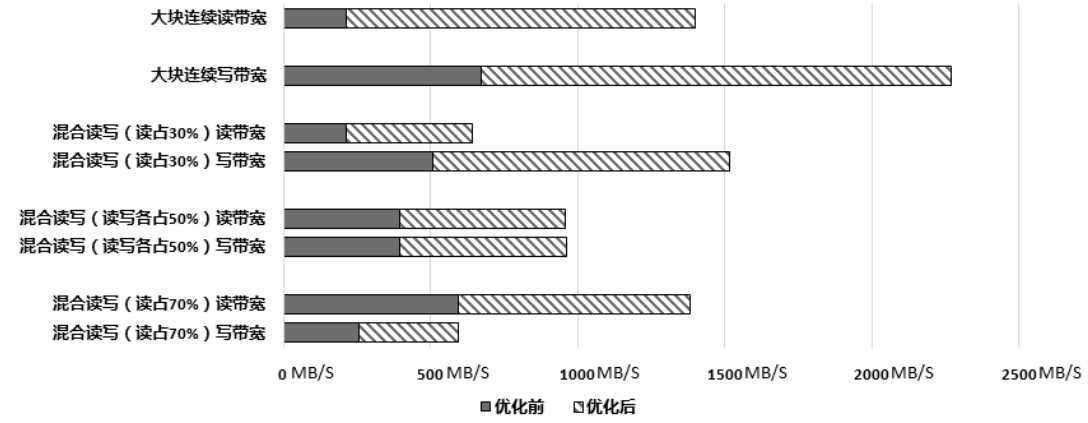

图 5-3 大块连续场景吞吐量测试结果

5.2 淘汰算法命中率验证实验

为了说明本文设计的淘汰算法在命中率上的性能，本文课题设计了相应的仿真实验环境进行验证。实验选择 LRU 类算法中理论上命中率最高的 LRU-2 算法以及经典 LRU 算法作为对比对象，并在相同仿真环境中进行对比实验。为验证非高斯分布场景下算法的命中率性能，实验对用户访问符合高斯模型的场景进行仿真模拟。为验证算法在非高斯分布场景下的性能，实验也对在关于用户行为研究中使用广泛的幂律分布场景进行了仿真模拟。

5.2.1 高斯分布下命中率对比实验

用户访问所符合的高斯分布方差的大小与缓存空间的大小是相对的，影响的都是缓存命中率的理论极限。这种影响宏观表现为缓存命中率的理论极限与方差大小成反比，与缓存空间大小成正比。缓存空间大小对实验结果的展现更为直观，同时考虑实验的单一变量原则，实验过程中固定高斯分布变化过程中的方差 σ，将缓存空间的大小做为实验的唯一变量。

LRU-2 算法通过长度为 $l_{history}$ 的历史访问队列来过滤冷数据对缓存空间的污染，其本质是按式(5-1)设定频率阈值来进行过滤。$l_{history}$ 设定过小，会导致相对访问频率较高的数据被过滤，命中率无法达到理论极限甚至劣化；$l_{history}$ 设定过大，会导致算法对冷数据不够敏感，无法完全过滤掉冷数据。由于实验固定方差 σ，依据“3σ 原则”及实验验证，将 $l_{history}$ 设定为 6σ，LRU-2 算法在样本空间有限条件下基本表现最优命中率，因此实验设定 LRU-2 算法历史访问队列的长度为 6σ。

$$P = \frac{1}{l_{history}} \times 100\% \tag{5-1}$$

实验在不同缓存空间大小场景下，模拟用户访问 30 万次。为反映用户访问模式均值小幅变化和大幅度变化对算法命中率的影响，实验每 10 万次对用户访问模型进行一次调整。实验共模拟 30 万次用户访问，用户访问模式依次符合 N(9750,250)，N(9500,250)，N(14000,250)，统计 30 万次用户访问下缓存命中率的整体均值，做为命中率性能的评价指标。实验结果如图 5-1 所示，x 轴表示缓存空间最大所能容纳的数据对象数，y 轴表示用户 30 万次访问的平均命中率。

由实验结果可知，与 LRU 和 LRU-2 算法相比，本文提出的算法，在用户访问符合高斯分布的场景下，缓存命中率表现更佳。当缓存空间容量为 500 时，依据高斯分布的“3σ 原则”，理论上缓存命中率为 0.6826，LRU-2 算法命中率为 0.58219，而本文设计的淘汰算法的命中率可以达到 0.6624；当缓存空间容量为 1000 时，理论上缓存命中率为 0.9544，LRU-2 算法命中率为 0.9319，而本文设计的淘汰算法的命中率可以达到 0.9367；当缓存空间容量为 1500 时，理论上缓存命中率为 0.9974，此时由于频率估计概率的误差影响以及频率阈值的作用，

LRU-2 算法的命中率不再优于 LRU 算法，LRU-2 算法命中率为 0.9683，LRU 算法命中率为 0.9823，但本文设计的淘汰算法命中率可以达到 0.9830。

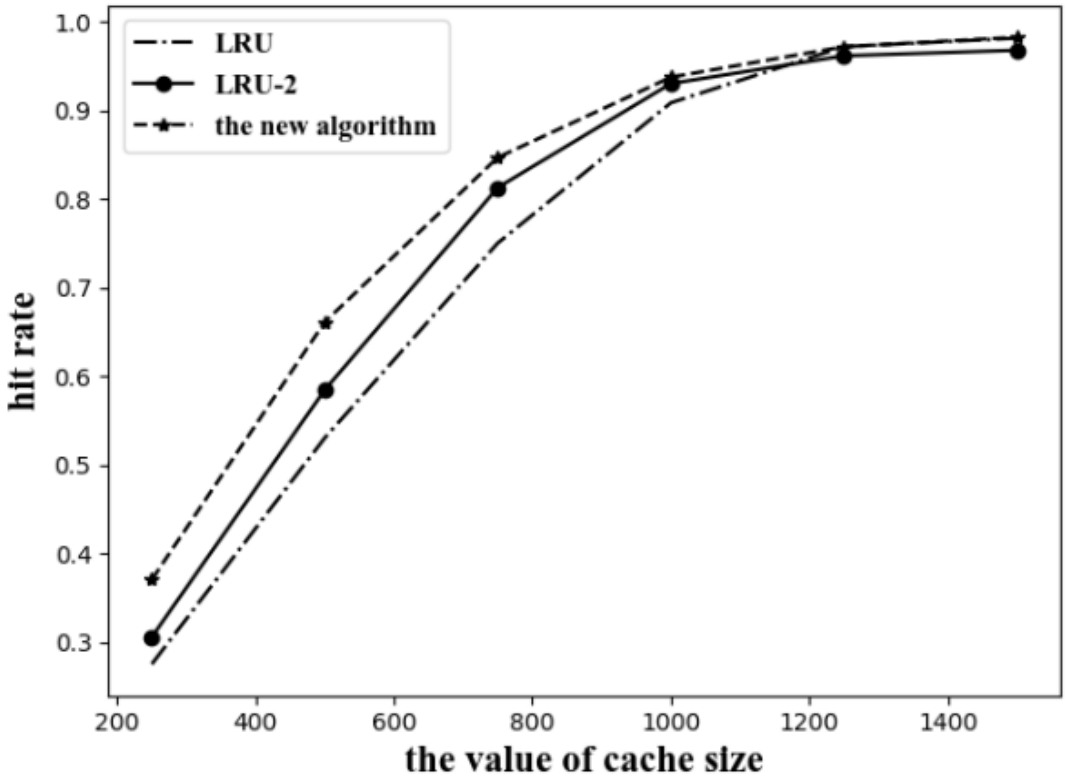

图 5-1 高斯分布下缓存空间大小对命中率的影响

5.2.2 非高斯分布下命中率对比实验

幂律分布在人类社会行为研究中具有普适性[38]，zipf 分布是一种典型的幂率分布，且有实验表明互联网场景下的文件下载次数呈现 zipf 分布[39]，因此非高斯分布场景下的实验选取用户访问符合 zipf 分布作为实验环境。

zipf 分布是用户行为研究中使用广泛的“二八定律”的数学模型[40]，其概率质量函数如式(5-2)所示。n 表示随机变量 x 的最大取值，a 被称为 alpha 参数，a 取值越大，zipf 分布越密集，a 取值越小，zipf 分布越稀疏。考虑到 a 对分布形态特征的影响，因此将 a 做为实验中唯一变量，从 0.1 到 1.5，按步长 0.2 依次取值。

$$f(x)=\frac{1}{x^{\alpha}*\sum_{i=1}^{n}(1/i)^{\alpha}} \tag{5-2}$$

依据“二八定律”，实验将 zipf 分布的 n 值设定为 250，缓存空间大小设为 50。为使 LRU-2 算法命中率达到最佳，其历史访问队列长度设为与 n 值相同。每次实验固定 a 取值，并模拟用户访问 10 万次，统计缓存空间的平均命中率作为评价指标。实验结果如图 5-2 所示，x 轴表示 a，y 轴表示平均命中率。

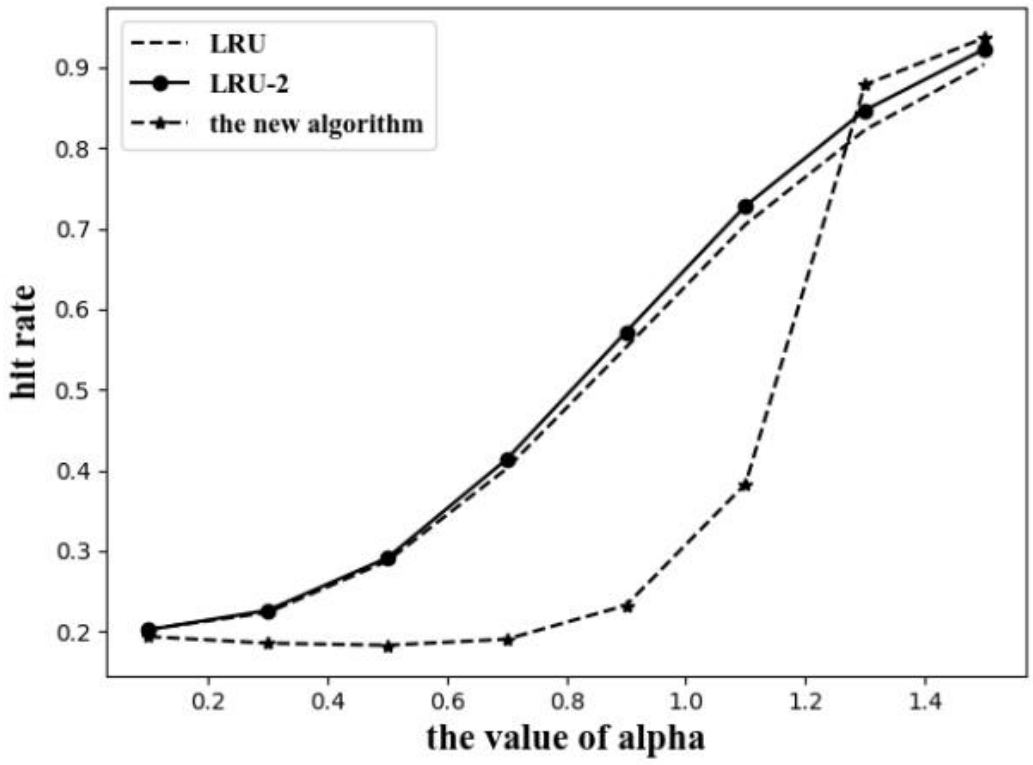

图 5-2 zipf 分布 a 因子对命中率的影响

由实验结果可知，LRU-2 算法命中率始终优于 LRU 算法；在 a 取值小于 1.1 时本文提出的算法命中率明显劣于 LRU 算法及 LRU-2 算法；当 a 取值大于 1.1 时，本文提出的算法命中率开始快速提升；当取值达到 1.3 时，本问题提出的算法命中率开始优于 LRU 算法及 LRU-2 算法。综上，在 zipf 分布场景下，本文提出的算法虽然整体表现不如 LRU 类算法，但在某些特定情况下也具有一定的优越性。

5.3 本章小结

本章对本文提出的两种优化方案分别设计实验进行验证。首先通过在服务器上搭建两套测试环境并分别通过 fio 对基于 Ceph 创建的虚拟硬盘进行性能测试，以验证本文提出的硬盘级优化方案对 Ceph 读写性能提升的程度，实验结果表明，在 Ceph 存储池仅提供单个虚拟硬盘的场景下，本文提出的优化方案能大

幅提高 Ceph 所提供的虚拟硬盘的 IOPS 及吞吐量性能。随后又设计仿真实验，对比当用户访问符合高斯分布和 zipf 分布时，本文设计的缓存淘汰算法与 LRU 及 LRU-2 算法的命中率差异。实验结果表明，当用户访问符合高斯分布时，本文设计的淘汰算法性能要明显优于 LRU 算及 LRU-2 算法，且更接近理论极限，与理论极限的差距控制在 3%以下；当用户访问符合 zipf 分布时，本文提出的算法命中率性能整体上不如 LRU 算法，但在 zipf 分布的 a 参数大于 1.3 时，本文设计的算法开始表现出比 LRU 算法更好的性能。由于同一高斯因子可能对应多个数据对象，非高斯分布下的实验结果表明，本文设计的淘汰算法存在一定的缺陷，需要进一步优化改进。

6 总结与展望

在使用 Ceph 做为后端存储系统时，由于 Ceph 会对上层 IO 请求的数据进行切分和 HASH 散列，因此如果单纯使用机械硬盘作为后端存储介质，将限制 Ceph 的读写性能。虽然固态硬盘具有高性能，但因其成本因素，并不适合做为 Ceph 后端全量数据的存储介质。因此针对 Ceph 的工作原理，设计相应混合存储方案对提升 Ceph 读写性能显得尤为重要，而这也正是本文研究的主要课题。

本文首先说明了当下作为混合存储研究重要课题的缓存技术在国内外的研究现状。随后对与混合存储方案优化设计相关的硬盘技术、FlashCache 技术、Ceph 存储系统及其上的 Cache Tier 技术的工作原理进行详细论述与分析。本文课题基于上述相关技术的分析及结论展开研究，并取得如下成果：

(1) 从 OSD 的工作原理出发，设计并实现了一种硬盘级的混合存储方案。该方案一方面，通过合理划分固态硬盘空间实现对元数据的加速；另一方面，通过 flashcache 技术提升机械硬盘的性能。除此之外，为了提高硬盘混合存储方案的可管理性，还基于设计原理设计并实现了相应的管理组件方便用户进行管理。通过实验验证，该方案极大程度提升了基于 Ceph 创建的单个虚拟硬盘的 IOPS 性能和吞吐量性能。

(2) 针对 Ceph 提供的 Cache Tier 混合存储方案的淘汰算法的缺陷，设计了一种基于高斯分布的缓存淘汰算法。通过仿真实验表明，在用户访问符合高斯分布的场景下，本文提出的淘汰算法较 LRU 及 LRU-2 算法命中率性能更优且更接近理论极限。

由于作者水平、时间等因素限制，目前本文取得的研究成果还存在一些缺陷，需要后续进行改进和完善。缺陷主要包括：

(1) 未对 Flashcache 缓存分区大小、Flashcache 内核并发控制参数等对系统整体性能影响进行测试与调优，后续还需进一步开展相关研究与实验，对整体方案的性能进行进一步的优化。

(2) 目前未找到云计算场景下对数据对象描述更为细致精确的高斯因子，因此还需进一步对云计算场景下用户的行为进行详细深入的研究。

(3) 同一个高斯因子可能对应多个数据对象，目前基本想法是在平衡二叉树的节点上通过拉链法建立这种一对多的映射关系，各拉链依据其在平衡二叉树中的层次分配空间，各拉链内部通过 LRU 进行管理。由于高斯分布是动态变化的，如何建立有效机制调整各节点拉链的长度是后续研究中需要解决的问题。

(4) 本文所设计的缓存淘汰算法目前只是理论研究阶段，后续还需将其合入到 Ceph 的 Cache Tier 中，并对其在云计算场景下的性能提升程度展开更进一步的研究。

参考文献

[1] Weil S A , Brandt S A , Miller E L , et al. Ceph: A Scalable, High-Performance Distributed File System[C]// 7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA. USENIX Association, 2006.

[2] 王洋, 刘星, 须成忠, et al. 大规模分布式文件系统元数据管理综述[J]. 集成技术, 2016(2):57-72.

[3] 任美翠. 未来网络智慧存储机制研究[D]. 2016.

[4] 刘洋. 层次混合存储系统中缓存和预取技术研究[D]. 华中科技大学, 2013.

[5] G. E. Suh,L. Rudolph,S. Devadas. Dynamic Partitioning of Shared Cache Memory[J]. The Journal of Supercomputing,2004,28(1).

[6] Oh Y, Choi J, Lee D, et al. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems[C]//FAST. 2012, 12.

[7] Makatos T , Klonatos Y , Marazakis M , et al. Using transparent compression to improve SSD-based I/O caches[C]// European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010. ACM, 2010.

[8] 张益铭. NVM 与 DRAM 混合存储优化策略的研究与实现[D].电子科技大学,2017.

[9] Computing - Parallel and Distributed Computing; Study Findings from Huazhong University of Science and Technology Broaden Understanding of Parallel and Distributed Computing (Vail: a Victim-aware Cache Policy To Improve Nvm Lifetime for Hybrid Memory System)[J]. Computers, Networks

& Communications,2019.

[10] Byan S , Lentini J , Madan A , et al. Mercury: Host-side flash caching for the data center[C]// Mass Storage Systems & Technologies. IEEE, 2013.

[11] 赵兴旺. 基于分布式缓存 Memcached 的 HC Model 机制及内部数据淘汰算法的研究[D].南京邮电大学,2018.

[12] Johnson T , Shasha D . 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm[C]// VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile. Morgan Kaufmann Publishers Inc. 1994.

[13] 王永亮.缓存淘汰算法研究[J].电子技术与软件工程,2018(23):134-135.

[14] Zhou Y, Philbin J, Li K. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches[C]//USENIX Annual Technical Conference, General Track. 2001: 91-104.

[15] Jiang S , Zhang X . LIRS: An Efficient Low Inter-reference Recency Set Replacement to Improve Buffer Cache Performance[C]// Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, June 15-19, 2002, Marina Del Rey, California, USA. ACM, 2002.

[16] Xiaohu Chen,Qilin Fan,Hao Yin. Caching in Information-Centric Networking: From a content delivery path perspective[P]. ,2013.

[17] Cao P, Irani S. Cost-aware www proxy caching algorithms[C]//Usenix symposium on internet technologies and systems. 1997, 12(97): 193-206.

[18] Chootong S , Thaenthong J . Cache replacement mechanism with Content Popularity for Vehicular Content-Centric Networks (VCCN)[C]// 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE). IEEE, 2017.

[19] 涂亮,徐雷.雾计算中基于古诺博弈的协作缓存优化算法[J].计算机技术与发展,2019,29(06):13-17.

[20] 罗桂娥,康霞.固态硬盘性能优化研究与实现[J].计算机工程与应用,2015,51(01):43-48.

[21] 山石. 长尾分布、幂律的产生机制和西蒙模型[C]// 人文社会科学评价暨科研绩效评价学术研讨会. 2009.

[22] 樊超. 基于社交关系的人类动力学研究[D].电子科技大学,2019.

[23] 魏守华,孙宁,姜悦.Zipf 定律与 Gibrat 定律在中国城市规模分布中的适用性[J].世界经济,2018,41(09):96-120.

[24] 张含阳.基于 R 语言的齐普夫信息挖掘——以机器人产业为例指导媒体关注重点[J].电子技术与软件工程,2019(05):251-253.

[25] 李德毅,刘常昱.论正态云模型的普适性[J].中国工程科学,2004(08):28-34.

[26] 周爱平. 高速网络流量测量关键问题研究[D].东南大学,2015.

[27] 黄丹,宋荣方.基于内容价值的缓存替换策略[J].电信科学,2018,34(11):59-66.

[28] 孙薇淇. 无线通信系统的边缘缓存策略及资源分配算法研究[D].北京邮电大学,2019.

[29] 唐颢. Ceph 存储引擎中基于固态盘的日志机制优化[D].华中科技大学,2016.

[30] 杨宗. Flashcache 的实现原理与优化研究[D].华中科技大学,2013.

[31] 方巍,文学志,潘吴斌,薛胜军.云计算:概念、技术及应用研究综述[J].南京信息工程大学学报(自然科学版),2012,4(04):351-361.

[32] Wong W S , Morris R J T . Benchmark synthesis using the LRU cache hit function[J]. IEEE Transactions on Computers, 1988, 37(6):637-645.

[33] Qureshi M K, Jaleel A, Patt Y N, et al. Adaptive insertion policies for high performance caching[J]. ACM SIGARCH Computer Architecture News, 2007, 35(2): 381-391.

[34] Guo, Jia,Liu, Chuanchang,Chen, Junliang,Sun, Huifeng. S-LRU: A cache

replacement algorithm of video sharing system for mobile devices[P]. ,2012.

[35] S. Das and A. Banerjee. An arbitration on cache replacements based on frequency — Recency product values[C]// 2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA), Bangalore, 2016, pp. 1-6.

[36] 陈海涛, 李宗惠. 平衡二叉树的失衡调整方法探讨[J]. 中国科教创新导刊, 2010(34):146-146.

[37] 李宗阳. 基于深度学习的用户行为过程预测方法研究与实现[D].北京邮电大学,2019.

[38] 曹盼盼, 阎春宁. 人类通信模式的幂律分布和 Zipf 定律[J]. 复杂系统与复杂性科学, 2009(04):55-60.

[39] 张成彬, 涂旭平. 大型文件共享系统的用户行为分析[J]. 武汉理工大学学报, 2009(18):32-34.

[40] 游荣彦. Zipf 定律与汉字字频分布[J]. 中文信息学报, 2000, 14(3).

致谢

时光如白驹过隙，短暂的硕士生涯转瞬即逝，回顾硕士生涯这段时光，一路风雨，一路风景。经历颇多，收获颇多，想要感谢的人也很多。

首先我要感谢我的导师蒋玉玲老师，谢谢蒋老师给了我这么好的学习和工作环境。还要感谢的是王振宇老师，是王老师的悉心指导让我迅速掌握了很多存储相关知识。在平时的学习工作中，王老师会耐心帮我解决遇到的困难，毕业论文也是因为王老师的悉心指导才得以完成，所以谢谢王老师对我的支持和帮助。

其次我要感谢的是云存储组里的众多同事。谢谢你们在工作学习上对我的帮助。谢谢王义飞组长带我迅速掌握许多服务器相关的硬件知识并让我在组里承担起部署和高可靠的工作。还要感谢李云洋、徐鲲等同事，在与你们的交流沟通中，让我对理论知识有了更一步的认知，对问题也有了更深刻的理解。

我还要感谢研究生办的各位老师，谢谢陈老师、余老师、潘老师以及程老师对我在学校生活上的帮助和关心，在研一阶段基础知识学习中对我的指导，感谢王老师对我毕业论文的关心与指导，以及最后对论文质量的把关。除此之外还要感谢学校周围的同学，谢谢马腾、张林等同学，在学习之余丰富了我的宿舍生活，感谢秦菁学姐对我论文格式及工作学习上的帮助。

最后，我要深深感谢我的父母以及我的女朋友。谢谢你们的陪伴，是你们的关心让我顶住学习工作中的压力，奋力前行。

附录 1 攻读硕士学位期间参与的项目和发表的论文

发表的论文：

[1] 王筱橦,蒋玉玲.对象存储中基于高斯分布的分层缓存淘汰算法 [J]. 网络新媒体技术, 2019.（已录用，待发表）