分类号_____ 学号 M201472687

学校代码 10487 密级_____

华中科技大学

硕士学位论文

基于 Ceph 分布式存储系统副本一致性研究

学位申请人：刘鑫伟

学科专业　：计算机技术

指导教师　：万继光　副教授

答辩日期　：2016.5.24

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

Research on Replica Consistency Based on Ceph Distribute Storage System

Candidate : Liu Xinwei

Major : Computer Technology

Supervisor : Assoc. Prof. Wan Jiguang

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

在当前云服务环境下，分布式存储系统作为底层的存储，由于它的大容量和高扩展性而备受关注。在分布式存储系统中，对于副本一致性问题，各个分布式存储系统使用了不同策略，既有使用强一致性的主从复制策略、链式策略、Paxos 算法等，还有使用最终一致性的 NWR 策略。在当前多样化的云服务环境下，单一的副本策略并不能适应复杂的应用环境。

Ceph 的强一致性策略对系统的写操作要求很高，它的写操作要写所有的副本才算成功，否则写操作就失败，写操作具有较高的延时。为了让 Ceph 适应多样化的应用环境，设计了一种基于读写比例的动态副本一致性策略。使用定时器定时统计当前系统中的读写操作数，从而将系统分为了四种状态：读写疏松型，读密集型，写密集型和读写密集型。在不同的状态下使用不同的算法，根据读写的比例得到同步写的副本数，而其他的副本使用异步更新的方式，写操作的比例越高，写操作的延时越低。

根据方案设计，对 Ceph 的副本策略进行修改，并对修改后的系统进行了测试。测试结果表明，对于写操作的处理，通过异步更新模块可以大大降低写操作延时，动态副本策略的写延时平均降低了 30%左右，并随着写操作的比例的变化而变化，基本达到了预期的目标。

关键字：分布式存储系统，Ceph，副本一致性，动态副本策略

Abstract

In the environment of cloud based service, distributed storage system as the underlying storage is paid attention, due to its large capacity and high scalability. To solve the replica consistency problem, different distribute storage system used different strategy, include primary-copy, primary-chain, Paxos and NWR replica strategy. The single replica strategy can not adapt to the complex application environment.

Ceph's strong consistent strategy has high aesthetic requirements for write operation. Write operation has to write every replica, so this strategy has high latency. To adapt to the complex application environment, designed a dynamic replica strategy based on the proportion of reading and writing. Based on the current number of read and write operations in the system, the system is divided into four kinds of state: reading-writing osteoporosis, reading intensive, writing intensive, reading-writing intensive. In different state, used different algorithm. The number of synchronous replicas is calculated by the proportion of reading and writing. And the other replica is asynchronous updated.

According to the designed scheme, modify replica strategy of Ceph, and test on the modified system. The test results show that, the writing latency has reduced by 30%, and it is changed with the proportion of writing. This strategy basically achieved the designed goals.

Keywords: Distributed storage system, Ceph, Replica consistency, Dynamic replica strategy

目 录

1 绪言

本章主要介绍论文的课题背景，分析国内外存储领域的相关研究状况，说明了本次研究的目的和主要内容。

1.1 课题背景

近年来随着云存储和云计算的快速展，网络数据增长迅猛，特别在云计算成为当前时代的技术热点后，这种数据量的增长越发明显。根据百度公开的信息显示，它每天要接收全世界上百个国家和地区的大量请求，要处理超过 100PB 的数据，从星罗棋布的信息中精确抓取约 10 亿网页。百度正在建设的某数据中心，它的后端存储大小将超过几千 PB，大约可存储几十万个国家图书馆的全部信息量。而数据中心对如此巨大的数据的处理和存储完全靠云计算和云存储来实现。

云计算和云存储的引入，互联网与存储联系变得更加紧密，单节点的存储已经跟不上计算机技术的发展了。面对容量，价格和安全性等方面的因素，传统的依靠提高单台物理机的存储能力已经完全不能满足用户与日俱增的存储需求。云计算带来了 IT 行业的一次巨大变革，面对海量级的数据需要计算与存储，分布式系统成为首选。它将各种各样的存储资源结合起来共同对外服务，可以将单一的任务分发到不同的节点进行处理，大大提高了处理的效率。

为了维护系统的可靠性，传统的存储方式中 RAID 方式使用广泛，因为 RAID 技术已经非常成熟，并且可以节省空间。例如 RAID5，它存储空间的利用率一般都在 80%以上。但是随着大数据的出现以及存储存储成本的降低，分布式存储快速兴起。而为了保证系统的可靠性，大多数的分布式存储系统主要使用了多副本技术或者编码技术，多副本技术的空间的利用率比较低，但是性能和可靠性较高。因此很多分布式存储系统采用多副本方式，如 Ceph 一般就是 3 副本的冗余存储方式。因为一般平台都是商用的存储服务器，存储成本比较低，在分布式处理、多数据中心容灾等互联

网应用的场景下，直接采用多副本技术比传统 RAID 更合适，这是典型的用空间换性能和可靠性的例子。

Ceph[1]作为一个分布式存储系统，它从 2004 年提交了第一行代码，至今为止已经走过了十一年以上的漫长路程。Ceph 这个分布式存储系统拥有很多亮点，它拥有块存储、对象存储、文件系统存储的统一存储能力和自动化的维护[2]等等，它是一个可靠的自治的分布式存储系统。Ceph 使用 CRUSH 算法替代了广泛使用的一致性哈希，很好的解决的数据分布的问题，消除了传统分布式存储中的中心节点，客户端只要使用 CRUSH 就可以知道数据存放在哪。并且 Ceph 的元数据服务器集群使用了动态子树分区的策略，可以很好的平衡各个元数据服务器上的负载。Ceph 目前在 OpenStack 社区中备受重视。OpenStack 是目前最为流行的开源云操作系统。Ceph 由于它的统一存储能力可以作为 OpenStack 的强大的后端存储，大部分 OpenStack 的研究者或使用者都会将 Ceph 作为他们的备选方案，毕竟 OpenStack 需要的分布式对象存储、块存储、文件系统存储，Ceph 都能提供。

分布式存储系统[3]是由大量廉价的商用 PC 服务器构建而成，因此将数据单份保存存在一定的风险，当遇到无法预料的灾害或故障到时，该节点的数据丢失也就意味着数据的永久丢失。因此大部分的副本是存储系统都使用了副本技术，将数据多份保存在不同地理位置的节点上，以提高数据可靠性[4]。这种副本技术带来了较好的可靠性和读性能。因为数据多份保存在多个节点上，当该数据被密集访问时，可以分担到多个存储节点上，提供较高的读性能。而副本技术也带来了一致性维护的问题，每份数据都要保存多个副本，因此每次写操作都要写多个节点，以保证每个节点的数据都是一致的。当副本数量越多时，数据的可靠性越高，读性能也越好，但是所有副本间一致性维护的花费也越高。因为写操作必须更新到所有的副本才算更新成功，这对每个节点的存储设备有较高的可靠性要求，如果在更新的过程中任何一个节点的磁盘发生故障，则这次的更新操作就会失败。实际上，副本一致性[5]带来的是写性能和读性能及可靠性之间的矛盾。当集群非常非常大时，这种强一致性策略将会很难实现，甚至导致系统不可用。比如在一个 PB 级别的系统中，如果每块盘是 1TB，那么就有

1000 块磁盘。按照大部分磁盘的故障率来算，一个星期就会有一块磁盘产生故障。而 Ceph 使用的数据分布算法是 CRUSH，数据分散在各个 OSD 节点上，一个磁盘的故障将涉及到很多数据，将导致大量用户无法对数据进行操作。1PB 的系统每周发生一次这样的故障，如果是 10PB 甚至是 100PB 的集群呢，这将导致数据经常无法写入，这对业务连续性的影响将是不可忽略的。

1.2 国内外研究现状

在分布式存储系统中根据 CAP 理论，在保证分区容忍性（P）的前提下，需要在一致性（C）和可用性（A）之间进行权衡，一致性越强，可用性越低，当一致性强到一定程度时，写操作可能会常常失败，导致整个系统不可用。对于副本一致性问题，许多分布式存储系统都有它们各具特色的副本策略，典型的有 Sheepdog 使用的传统副本策略，Google 的 GFS 使用的链式副本策略，亚马逊的 Dynamo 分布式存储引擎使用的 NWR 副本策略，以及分布式系统中的强一致性算法 Paxos 和 Raft 算法。

Sheepdog[6,7,8]是一个分布式块存储系统，它使用了完全对称的结构，没有元数据服务的中心节点，没有单点故障容易管理，扩展性较好。Sheepdog 使用的是强一致性副本策略，客户端进行写操作时，先使用一致性哈希算法计算目标 OSD 节点，然后发送写请求到所有目标节点，只有所有副本都更新成功后，写操作才算成功。Sheepdog 的读操作时随机发往任意一个副本，如果一个副本未更新，客户端有可能从未更新的节点读取旧数据。

Google 的 GFS[9,10]是一个分布式文件系统，GFS 的集群有三种节点：GFS Master（主控服务器），GFS ChunkServer（数据块服务器）以及 GFS Client。GFS 中只有一个主控服务器，有多个数据块服务器，它是一个有中心节点的分布式文件系统。GFS 使用的副本策略是比较复杂，它分离了数据流和控制流，数据流使用链式副本策略，控制流使用了主从复制，整个策略如图 1.1 所示。GFS 客户端的进行写操作时先要访问 Master 服务器以获取副本存放的位置，客户端再以链式方式把要写的数据发送到所有的副本，当所有的副本确认收到了数据，那么客户端发送控制命令给主副本，主

副本执行写操作并将控制命令发送给次副本。次副本写成功后才会回复主副本，当主副本收到所有的响应后回复客户端写操作成功。

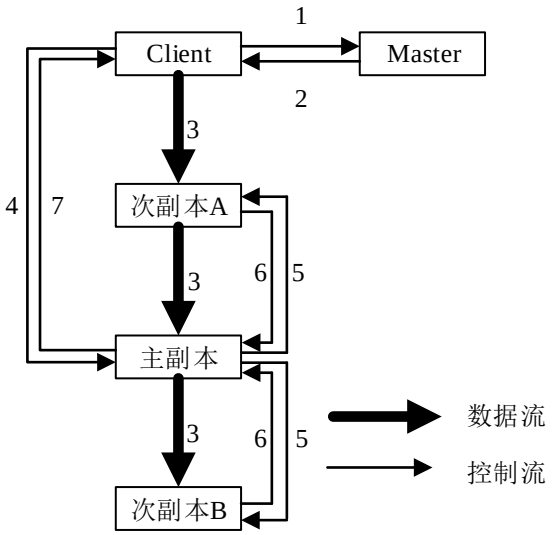

图 1.1 GFS 的副本一致性策略

如果合并 GFS 的数据流和控制流，整个流程就可以大大的简化，形成一个完整的链式副本策略，如图 1.2 所示。客户端直接发送写操作给主副本，主副本确定写入数据并将写操作以流水线的方式发送给次副本，每个次副本收到消息后执行写入操作，写入完成后向前一个副本进行回应，当主副本收到回应后表明所有副本写入完成，此时主副本向客户端回应写操作成功。

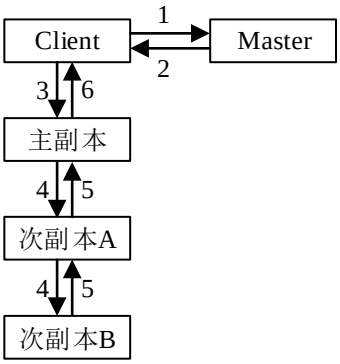

图 1.2 链式副本一致性策略

GFS 相对于 Sheepdog 的副本策略，有一定的优势，因为 Sheepdog 的客户端要通过外网给每个副本发送写操作，每个写操作都会产生网络延时。而使用链式副本策

略，只需要通过外网发送一份写操作给主副本，主副本可以通过内网以流水线的方式发送给次副本，大大降低延时，当客户端离集群越远时，这种优势就越明显。

亚马逊的 Dynamo[11,12]是一个分布式键值系统，它主要用于亚马逊的购物车和 S3 云存储服务。Dynamo 是一个完全对称的，无中心节点的系统，它通过组合 P2P 的各种技术做成了一个线上可运行的分布式键值系统。Dynamo 的数据分布策略采用了一致性哈希算法，而副本策略则采用了 NWR 副本策略。NWR 策略是 Dynamo 的一大创新，其中 N 表示了副本数量，W 是写操作时最少要写节点数，R 为读操作时最少要读的节点数。只要保证 W+R>N，那么读操作的节点和写操作的节点就存在交集，客户端总能读到最新的数据。这种 NWR 副本策略可以很好的平衡一致性（C）和可用性（A）。当用户比较注重写操作的效率时，可以将 W 设为 1，R 设为 N，那么写操作只用写一个副本。如果用户比较注重读效率则可以将 R 设为 1，W 设为 N。如果用户需要在读写之间进行平衡时，可以将 W 设为 N/2，R 也设为 N/2。Dynamo 的客户端执行写操作时，会先根据一致性哈希算法算出副本所在的存储节点，然后选择一个协调者将写操作并发的发往所有的副本节点，当有 W 个副本回复写操作成功则向客户端回复写操作成功，整个流程如图 1.3 所示。

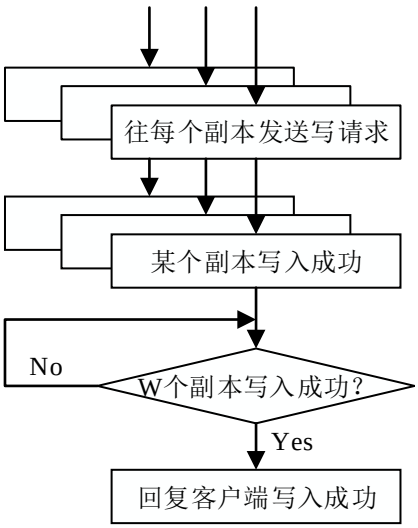

图 1.3 Dynamo 的 NWR 副本策略

除了以上的以上介绍的副本策略外，还有使用强一致性算法来达到副本一致性

的，比如 Paxos[13,14,15]算法和 Raft[16]算法。Paxos 算法主要用于解决分布式的各个节点就某个值达成一致的问题。对于存在主节点的分布式存储系统中，如果该节点出现故障，可能需要选举出新的节点，Paxos 就可以实现该需求。Paxos 算法主要用来实现分布式系统中的某些共可靠的全局服务，例如分布式锁服务，全局命名服务和全局配置服务的。对于某些性能要求较高的分布式块存储或者文件系统存储，该算法并不适合直接参与副本策略，这些分布式存储系统的一般写操作的数据量较大，若使用 Paxos 算法就某个值达成一致将会造成很高的延时，一般该算法用于在主节点宕机时选举出新的主节点继续进行服务。而对于更新数据量较少的系统，例如分布式表格系统，该算法是适用的。Google 的分布式表格系统 Megastore[17]采用的是基于 Paxos 算法的复制协议机制，当机器故障时自动切换机器从而保证写服务不会间断，具有很高的可靠性，但是该算法的缺点就是延时较长，一般为几十上百毫秒。Raft 算法提供的功能与 Paxos 算法的功能相似，可以看成是 Paxos 算法的一个简化版本，当然在某些功能上可能没有 Paxos 完善。

1.3 研究目的和主要内容

本文的主要工作是研究 Ceph 的副本策略，针对分布式存储系统上层应用的多样性修改它的强一致性副本策略，优化为一种用户可配置的基于读写比例的动态副本策略，从而平衡 Ceph 的读写性能。

本文研究的主要内容如下：

1. 对现有的副本一致性策略进行研究，以此作为副本策略设计的基础；
2. 研究 Ceph 系统的框架，分析它的读写 IO 路径；
3. 针对 Ceph 原有的副本策略进行修改，设计动态副本策略；
4. 对动态副本策略进行实现，并对性能进行测试。

本论文的内容组织如下：

第一章简单介绍了分布式存储系统中副本一致性问题的产生背景，国内外各个分布式存储系统所使用的副本策略以及研究的主要内容。

第二章主要介绍了设计和实现中所涉及到的相关技术，主要对Ceph的整体架构，它的数据映射算法 CRUSH 算法以及副本一致性技术进行了介绍。

第三章详细分析了 Ceph 的 IO 路径，了解了它的副本策略，设计了一种基于读写比例的动态副本策略，并说明了各个模块的设计方案。

第四章对整个副本策略的实现进行了详细的阐述，列出了关键的数据结构和主要的处理流程。

第五章介绍了系统的测试环境，并完成了系统的性能测试和结果分析。

第六章对全文进行总结，并展望下一步工作。

2 相关技术介绍

2.1 分布式存储技术

要了解分布式存储技术[18]，就不能不提到它的来源，分布式存储是整个云服务环境的基础，云服务是云计算[19]的表现形式，云服务的应用模型大致如图 2.1 所示。

图 2.1 云计算的表现形式

软件即服务（Software-as-a-Service，SaaS）是软件层，在该层面是让用户使用，用户通过访问应用来做些数据挖掘等，比如手机中的云盘服务。平台即服务（Platform-as-a-Service，PaaS）是一个中间层，它提供了上层软件开发所需的资源，企业等研发单位通过它可以开发应用。基础设施即服务（Infrastructure-as-a-Service，IaaS）提供了一个企业开发所需要的各种硬件资源，比如服务器，存储和网络，这些都可以从 Iass 服务商那儿租用，大大降低成本。而整个云计算服务所用到的存储功能都可以通过分布式存储来提供[20]。

分布式存储就应用场景上相对于存储接口上来说，现在流行分为三种：分布式对象存储[21]，分布式块存储[22]，分布式文件系统存储。分布式对象存储也是通常意义的键值存储，接口简单，每个对象可以看成一个文件，只能全写全读，比如亚马逊的 S3 就是一个对外提供 Web 服务的分布式对象存储系统， Web 服务所用到的所有资源都是以对象存储在该系统中。分布式块存储主要用于提供虚拟磁盘服务，它的 IO

特点与硬盘一致，一块硬盘所能提供的功能它都得实现，当云计算中的虚拟服务器[23]需要使用新的硬盘时，通过分布式块存储系统就可以得到。分布式文件系统则表现为一个巨大的文件系统，它需要维护每个文件的属性以及目录信息等，该系统需要维护大量文件以及目录的元数据信息，这也是它和对象存储的区别，毕竟文件系统需要更复杂的逻辑结构。而分布式存储系统如果按照业务逻辑进行划分，又可分为分布式文件系统，表格系统，键值系统和分布式数据库等。不同的业务需求不同，存储的选择方式也就不同，存储即数据结构，而数据结构无穷无尽，存储的方式也是无穷无尽的。

现在开源的最受欢迎的云计算平台 OpenStack[24]就需要使用到分布式的对象存储，块存储和文件系统存储（作为共享存储）这三种存储方式。OpenStack 对外提供了 Restful API[25]接口，需要使用到对象存储。OpenStack 的 Cinder 模块需要使用到虚拟磁盘服务，它可以通过分布式块存储提供。OpenStack 对外要提供的云存储服务并且它的 Nova 模块需要使用共享存储服务，这些都可以通过分布式文件系统实现。Ceph 分布式存储系统却恰好可以提供这三种存储功能，搭建一个 Ceph 即可提供块存储，对象存储和文件系统存储，而不需要每种存储分别搭一个集群，Ceph 具有统一存储的能力。云服务是云计算的表现形式，而分布式存储系统为整个云计算的存储提供了强大的后端支持。

2.2 Ceph 相关技术

2.2.1 Ceph 介绍

Ceph 是一个开源的分布式存储系统，旨在提供一种软件自定义的，统一存储解决方案[26]。Ceph 是一个大规模可扩展，高可靠性，没有单点故障的并且拥有自动维护和恢复功能的存储系统。一个大的分布式存储系统由上千台商用 PC 机组成，随着系统的运行，新的存储节点会加入，旧的存储节点会移出，磁盘的故障会经常发生，新的数据会被创建，旧的数据会被迁移删除等，这些都是一个分布式存储需要解决的。Ceph 设计的初衷就是：在一个大的分布式集群中，节点的故障是常态，而不是

特例，Ceph 可以自动[27]进行故障检测和恢复。Ceph 底层采用了对象存储架构，由智能的存储设备 OSD 替代传统的硬盘，OSD 整合了计算机上的 CPU、网络和硬盘这些硬件资源。Ceph 的不仅仅是一个分布式存储系统，它可以看做是一个存储生态系统，如图 2.2 所示。

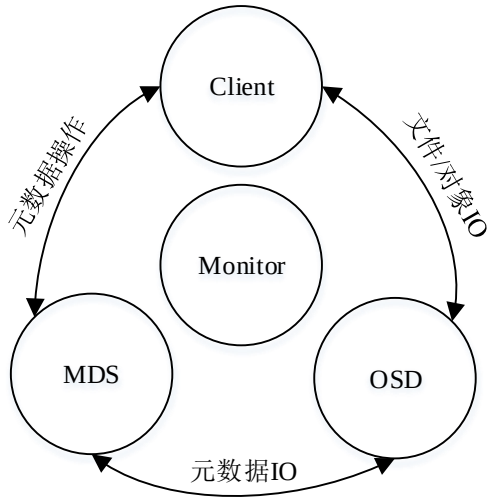

图 2.2 Ceph 生态系统

Ceph 集群由客户端（Client），元数据服务器集群（Metadata Server Cluster，MDS），对象存储集群（Object Storage Device Cluster，OSD）和集群监视器（Cluster Monitor）组成。Ceph 的功能十分强大，它提供了块存储，对象存储和文件系统存储的功能，因此它的客户端也多种多样，但只有 Ceph 的文件系统功能需要用到 MDS。就分布式文件系统而言，Ceph 分离了数据和元数据，所有对元数据的操作全部由 MDS 进行管理。文件系统的客户端暴露了 POSIX[28]的文件系统接口给主机或进程，用户使用该接口可以直接挂载该文件系统。客户端所有对元数据的操作（目录的创建删除修改等）都是通过 MDS 进行。但 MDS 并不将元数据存储在本地硬盘上，它只是从 OSD 中预取了一部分元数据到本地的内存之中，最终元数据的修改要落实到 OSD 上才算操作成功。当客户端需要对文件进行读写操作时，它会从元数据服务器获取文件的 inode 信息以及权限，然后使用 CRUSH 算法计算出存储该文件的 OSD 节点，Client 直接与 OSD 节点通信进行读写。Ceph 的 Monitor 维护了整个集群的全局状态，OSD

节点的加入或者退出等都需要与 Monitor 进行交互。

Ceph 的 RBD 功能也就是块存储功能，是目前使用最广泛的功能。Ceph 的块存储是被大力推荐并且高速开发的模块，因为它提供了用户非常熟悉的通用接口，并且在目前流行的 OpenStack 和 CloudStack 中可以得到广泛接受和支持。Ceph 块存储将计算和存储解耦，拥有 Live migration[29]（虚拟机的热迁移）特性、高效的快照和克隆/恢复特性，它的这些引人注目特性使 RBD 成为 Ceph 的一个强大支撑点。

2.2.2 CRUSH 算法

Ceph 有两大优势被用户所熟知，一个是它的统一存储架构，另一个是它使用的 CRUSH 算法，CRUSH 同时也是 Ceph 的设计之初的两大创新之一（另一个是 MDS 使用的动态子树分区）。CRUSH 算法是一个数据分布算法，与一致性哈希[30]的功能相似，但是它在一致性哈希的基础上很好的考虑了容灾域的隔离，能够实现各类负载的副本放置规则，即是在数千台 OSD 的 Ceph 集群中，它仍能保持良好的负载均衡。

在传统的分布式存储系统中，数据分布算法简单，新加入的磁盘要么是空的，整个磁盘没有被使用，要么是满的，整个系统迁移了大量的数据到新盘，随着系统的扩展，整个系统变得极不均匀也极不稳定。对于一个大的分布式存储系统而言，它需要解决 PB 级别的数据的存储问题，为了能够充分利用每个 OSD 节点的存储资源，它需要根据工作负载和存储能力均匀的分配数据[31]，并且在 OSD 节点进入或者移出时，产生最小的必要数据迁移。而 CRUSH 就具有这样的功能，它能根据用户的策略分布数据，实现良好的负载均衡。在 Ceph 中，客户端获取了一段数据的 ID 后需要知道该数据存放的节点，通过 CRUSH 计算即可得出，而不再从某个中央目录进行查表。CRUSH 被实现为一个伪随机的确定性函数，CRUSH 可以通过一个数据块的唯一编号得到一系列用来存储该数据块的副本位置。CRUSH 算法与传统算法的不同之处在于，它的数据放置不依赖于任何类型的文件或对象式目录，它只需要一个简洁层次分明的集群资源的描述（也就是 OSD 节点分布的描述，CRUSH Map）和用户可定义的副本放置策略（CRUSH Rule）。这种方法有两个重要的优点，一个是该算法是完全分

布式的，集群中的任何节点都可以使用该算法以确认数据存放的位置，第二个就是 OSD 节点的分布信息（CRUSH Map）是静态的，只有在 OSD 节点加入或者退出时才会变化。CRUSH Map（OSD 节点的描述信息，又叫 OSDMap）结构和 CRUSH Rule 的规则如图 2.3 所示。

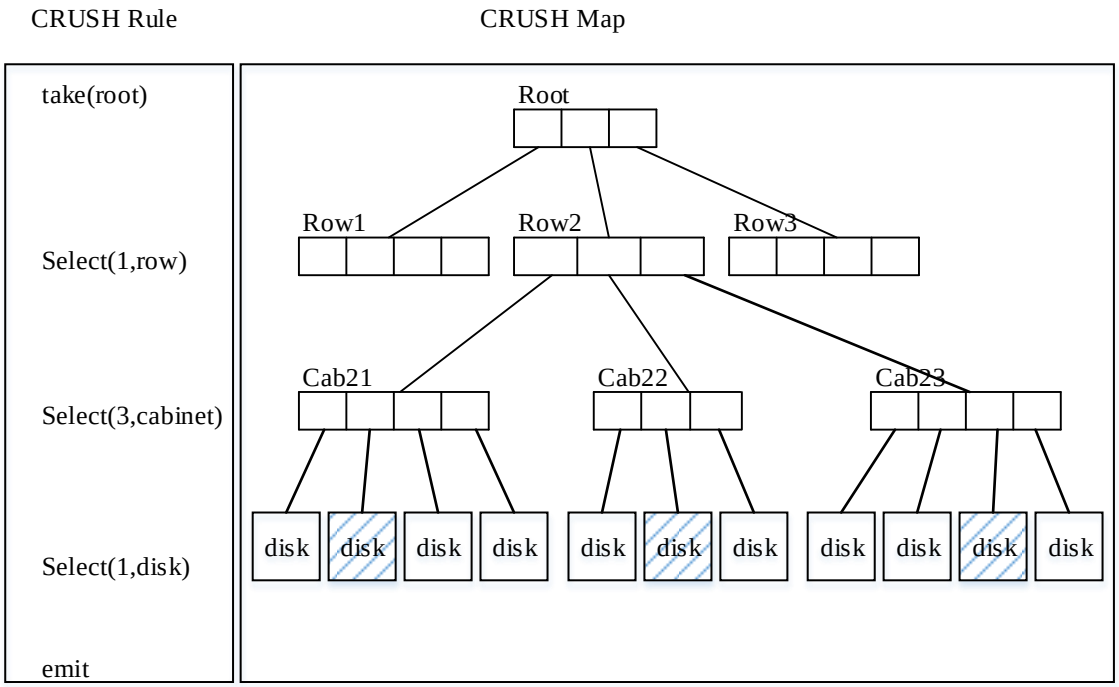

图 2.3 CRUSH 算法的 Rule 和 Map

CRUSH Map 的描述由桶和设备组成，桶和设备都可以设置权重，桶可以包含桶或者设备。在以上的 Map 中，root、row 和 cabinet 都是桶，它们可以包含其他的桶也可以直接包含设备 Disk。整个描述信息可以根据具体环境中的设备所在的地理位置进行设定，例如 root 可以代表数据中心，row 代表房间，cabinet 代表机架。根据左边的 Rule，take（root）代表选中了数据中心，select（1，row）表示从数据中心中选择一个房间，select（3，cabinet）表示从房间中选择三个机架，最后 select（1，disk）表示从三个机架上各选一个磁盘。最后这个 CRUSH Rule 处理的结果就是，从一个数据中心的某个房间中的三个机架上各选择一个磁盘。至于如何选择，CRUSH 算法会根据每个设备的存储能力和带宽资源等设置加权，根据加权使数据平均的分布从

而进行一个概率相对平衡的选择。CRUSH Rule 的设置用户可以自己定义，用户可以根据集群的地理位置信息，避免将副本放在同一个故障域中，比如共用电源，共用网络，共用控制器的机器。所以上面的 CRUSH Rule 可以进行修改，改为从一个数据中心的三个房间中的某个机架上各选择一个磁盘。这样选择副本的可靠性更高。

2.3 副本一致性技术

一致性问题源于 70 年代末的数据库系统：那个时候的目标是实现分布式的透明性—也就是说,对用户而言整个集群表现为只有一个系统，而不是一群相互合作的系统。在后来发展的很多分布式环境中，副本技术大量被普及，很多学者专注于副本一致性的研究。在 2000 年的 PODC 会议上，Eric Brewer 提出来 CAP[32]理论，阐述了在分布式系统中数据一致性[33]、系统可用性、网络分区的容忍性，这三个属性在任何时候都只能实现两个。两年后， Seth Gilbert 和 Nancy Lynch 证明了这一猜想。在一个大的分布式存储系统中，网络分区必然会出现，这就要求系统的设计者在一致性和可用性进行平衡。对于可靠性要求非常高的应用环境可以适当提高一致性，比如银行系统，银行系统大量使用了 IBM 提供的高可靠性设备；而对于可靠性要求较低的系统可以适当放松一致性得到更高的性能，比如网络直播系统，网络直播系统的观众可以允许读取数据不一定是最新的，允许读取少量的陈旧数据依然可以工作的很好。副本一致性模型[34]大致可以分为三种：强一致性，弱一致性和最终一致性模型。

强一致性模型。所有对数据的写操作都需要进行序列化，写数据操作成功之后，所有对该数据的读操作都将得到最新的值。该策略要求，更新操作必须在所有的副本上执行成功最后才算成功，一旦有一个副本失败，更新操作就算失败，甚至更新成功的副本需要进行回滚。强一致性需要做到数据存放的副本都能同步更新，所有副本之间的数据一模一样，写操作成功后客户端读取任意一个副本都能读到最新的数据，这种一致性模型写代价较高。

弱一致性模型。对数据的写操作不能马上同步到副本上，或者不会马上执行，用户马上进行读操作可能不会得到最新值，从用户的写操作到用户可以读到最新的值，

这段时间被称为“不一致窗口”，这段时间内系统会后台执行写操作或者异步更新副本，因此用户读取时，很有可能读不到最新的值。该模型适合写操作较少，或者对更新的实时性要求较低的场景。

最终一致性模型。它的一致性在强一致性和弱一致性之间，最终一致性模型没有保证所有的副本同步更新，即是更新了一部分也算成功，后续的读操作最终可以读到最新的数据。

在现在主流分布式存储系统中，使用这三种模型的都有。每一种模型都有许多不同的算法进行实现。对于强一致性模型，Google 的分布式文件系统 GFS 使用了主从同步复制[35]，还有主从链式复制策略，以及它的分布式表格系统 Megastore 使用的 Paxos 算法都是强一致性模型的典型例子。对于最终一致性模型，最为典型的就是亚马逊的分布式键值系统 Dynamo 使用的 NWR 策略，因为该系统被用于亚马逊的购物车，购物车有大量的写操作，若写代价太高将严重影响用户体验。

过强的一致性会减弱系统的可用性，可能会经常导致写失败，而过弱的一致性导致数据不能及时更新，影响后续的数据访问。在当前的云服务环境下，各种应用对存储的需求变化多端，强一致性或者弱一致性并不适用多样化的应用场景[36]，并且一种应用在它运行的过程中可能对一致性的要求也在不断的变化。因此，本文受 NWR 副本策略启发设计了一种基于读写比例的动态副本策略，因为较高的一致性带来的是较高的写代价，在系统中有大量的写操作时可以适当降低一致性，提供更好的写性能，当系统中有大量的读操作时可以提高一致性，用户读任意一个副本都能读到最新的数据，提供更好的读性能。

2.4 本章小结

本章主要阐述了课题研究中的一些关键技术，主要包括分布式存储技术、Ceph 介绍，CRUSH 算法和副本一致性的相关知识，从这些知识中得到了一些启示，有助于本文对 Ceph 副本一致性的研究和后续的设计。

3 Ceph 副本策略分析和优化

3.1 Ceph 系统架构

本章首先分层次的介绍了Ceph集群逻辑结构以及OSD的系统架构，重点对OSD端的副本一致性和处理读写 op 的性能进行了分析，并且阐述了动态副本一致性的设计。

3.1.1 系统逻辑架构

首先自上而下对 Ceph 集群的架构做个简单的介绍。Ceph 的最上层提供了对象存储，块存储和分布式文件系统存储。这些存储方式都是通过接口库 LIBRADOS 实现的，而 RADOS 是由 OSD 集群和 Monitor 集群组成。

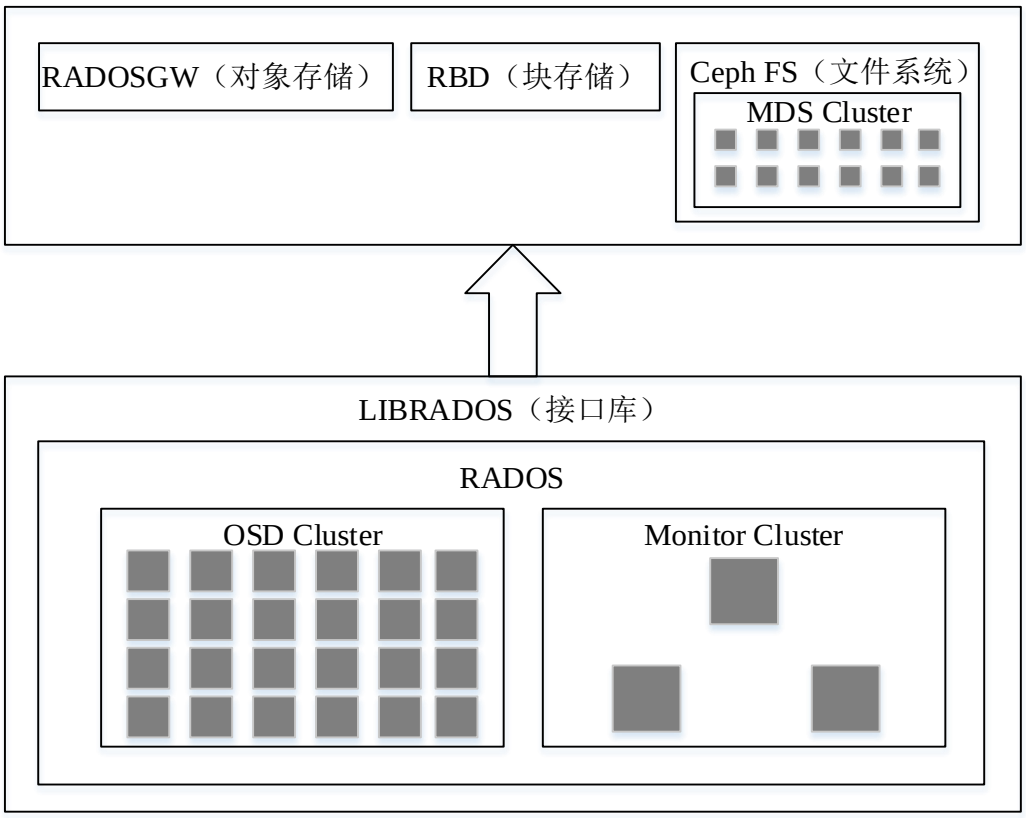

图 3.1 系统架构图

Monitor Cluster：Monitor 是整个 Ceph 系统的管理者，它用来维护 Ceph 集群的

当前的状态。在 Monitor 中维护了六张表，它们分别是 AuthMap，LogMap，MonitorMap，OSDMap，PGMap，MDSMap。PGMap 和 OSDMap 是 Ceph 集群中最重要的两个 Map，PGMap 是一张描述 PG（放置组）状态信息的 Map，而 OSDMap 是一张描述所有 OSD 节点的最新信息的 Map，所有 OSD 节点状态信息的变迁都需要与 Monitor 进行交流，并且更新到这个 Map 中。Ceph 的 Cient 节点，OSD 节点和 Ceph 的核心算法 CRUSH 都需要用到 OSDMap，可见 OSD 的重要。

OSD Cluster：它是对象存储服务器集群，每个 OSD 是 Ceph 用来存储所有信息的节点，它可以由大量廉价的 PC 机和硬盘组成。整个 Ceph 集群的所有用户存储信息都存放在 OSD 节点上，不论是数据还是元数据。

MDS Cluster：元数据服务器集群，只有在使用 Ceph 的分布式文件系统的功能时才会用到 MDS。MDS 主要用来将 OSD 上的元数据单独取出，所有有关元数据的操作都将通过 MDS 进行，并最终更新到 OSD 上，从而减轻 OSD 的负担，并提高元数据操作的速度。

RADOS（Reliable Autonomic Distributed Object Storage）： Monitor 集群和 OSD 集群就组成了整个 Ceph 集群的底层，并且基于它们实现了一个自治的可靠的分布式对象存储系统 RADOS。可靠是因为它使用多副本或者编码策略，可以进行数据迁移和备份，故障检测和恢复，自治是因为这些功能 Ceph 可以自动进行。

LIBRADOS：基于 RADOS 提供了一些接口库，包括了 C、C++、Java、Python、PHP 的接口库。

RADOSGW：Ceph 提供的对象存储网关，基于 LIBRADOS 实现了与 Amazon S3 和 Swift 兼容的接口，即 RESTful API。

RBD：基于 LIBRADOS 实现了 Ceph 的块存储，主要用来提供虚拟磁盘服务，是云计算虚拟化后端存储最重要的一块。

Ceph FS：基于 RADOS 和 MDS 实现了 Ceph 的文件系统存储，并提供了兼容 POSIX 语义的文件系统接口，MDS 的加入可以使 Ceph 快速的进行元数据操作。

3.1.2 OSD 模块架构

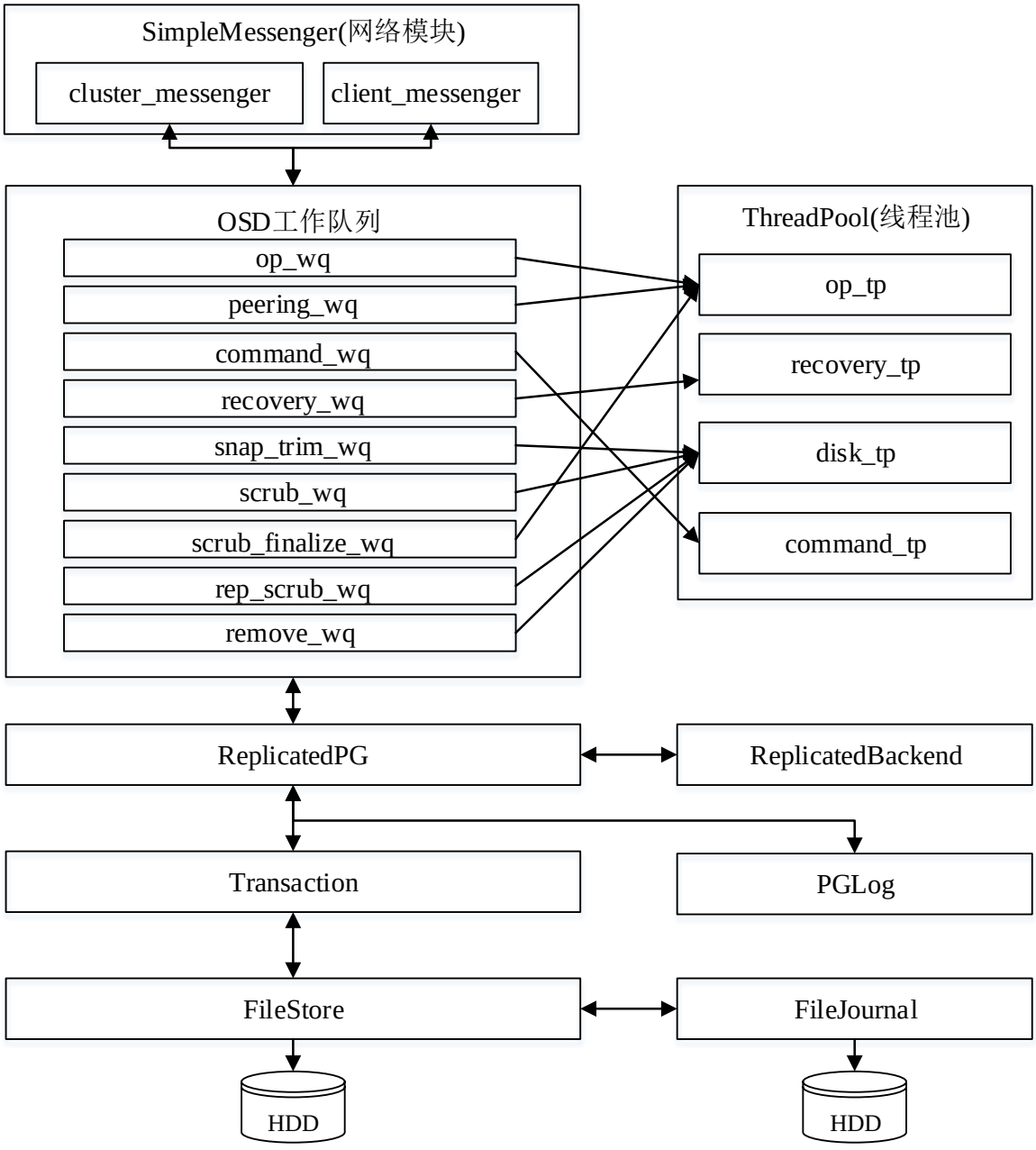

图 3.2 OSD 架构图

图 3.2 给出了 OSD 端的整体架构。OSD 主要负责客户端的 IO 操作，解决访问数据的复杂性，将数据的更新序列化，并且 OSD 还要维护整个系统的可靠性，进行故障检测和恢复。

SimpleMessenger 是 Ceph 的网络模块，Ceph 集群中所有的节点都是使用 SimpleMessenger 进行网络通信的。在 OSD 中有 SimpleMessenger 的两个实例，分别是 cluster_messenger 和 client_messenger。cluster_messenger 主要负责集群中的网络通信，OSD 与 OSD 之间的通信，OSD 与 Monitor 之间的通信。这里使用了观察者模式，OSD 是 client_messenger 和 cluster_messenger 的 observer，一旦 client_messenger 和 cluster_messenger 收到了消息，它都会 dispatch 给 OSD，由 OSD 进行消息的处理。

当 OSD 收到消息后，会对消息进行分类然后加入九条队列，最后由四个线程池对这九条队列中的消息进行处理。op_wq 是本次研究的最主要队列，所有的客户端的读写请求都封装成对 PG 的 op 操作，加入到该队列，由 op_tp 线程池处理。peering_wq，主要处理 peering 消息，在 OSD 加入或退出时，都会触发 peering 机制，该 OSD 负责的 PG 会与相关 primary OSD 和 replica OSD 进行通信以恢复 PG 达到主从副本 PG 的一致性。srub_wq 主要处理 srub 消息。OSD 会定期（3 天或一个星期）启动 srub 机制扫描该 OSD 负责的对象生成校验信息，然后与副本的校验信息对比，用来维护数据的可靠性。

ThreadPool 是 Ceph 的线程池模块，OSD 使用了四个线程池实例，用来处理九个消息队列。在服务器端，如果来一条消息就创建一个线程对它进行处理，那么就会频繁的创建线程和销毁线程，会消耗大量的资源。线程池启动时会创建固定数量的线程，如果没有任务时这些线程处于睡眠状态，一旦消息到达则会唤醒线程进行处理，线程不够用时会申请更多的线程，一旦消息处理完，线程再次进入睡眠状态，这就避免了线程的频繁创建和销毁。

op_tp 会处理所有的 op 操作，先调用 ReplicatedPG 对 op 操作再次进行分类，如果是写操作，会调用 ReplicatedBackend 将其发送给副本 OSD。然后 ReplicatedPG 先调用 PGLog 生成 PG 日志，再序列化 op 生成事务，最后提交给 FileStore。

FileStore 负责底层磁盘的 IO 操作，如果底层是 XFS 文件系统，它会先写事务的日志再写磁盘，如果是 btrfs 则会同时写日志和磁盘。

3.2 OSD 副本一致性分析

3.2.1 Client 端读写分析

要想分析 Ceph 的副本策略，必须先了解客户端是怎样找到副本的，即客户端是怎样定位它要读写的数据的。客户端要读取一个对象或者文件时，怎么找到它所存放的 OSD 节点，要写入时又该如何确定该写入的位置。客户端读写对象的定位如下图所示。

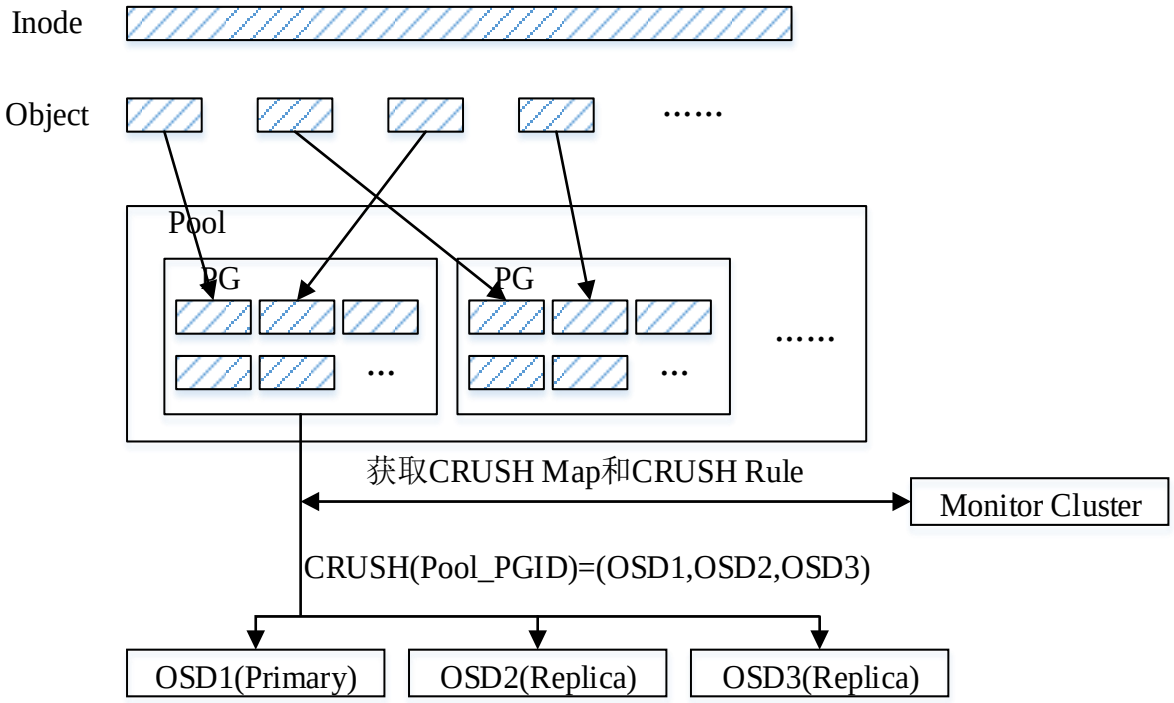

图 3.3 客户端数据读写的定位

Ceph 的客户端可以直接与 OSD 节点通信读写数据，提供快速的读写性能，这点得益于 Ceph 的核心算法 CRUSH。有了 CRUSH，Ceph 集群中的任意节点都可以找到数据存放的位置。这个算法功能和一致性哈希的功能相似，但它是一个数据分布的伪随机算法，能够将 Place Group（放置组）有序有效的映射到多个不同故障域的 OSD 设备上，这一组 OSD 用来存储属于该 PG 的对象，这个算法不同于传统的对象文件系统的数据映射算法，数据 PG 的定位完全不依赖任何用来描述位置的元数据。为了

定位每个 PG， CRUSH 只需要该 PG 的 ID，该 OSD 集群的描述信息 CRUSH Map（它分层的描述了 OSD 的位置信息，比如一个数据中心有多少机房，每个机房有多少机架，每个机架有多少服务器，每个服务器上有多少 OSD 节点）和分布 PG 的策略 CRUSH Rule（用来指定副本的存放位置，比如将副本存放到 CRUSH Map 中不同的机房，属于不同的故障域）。这个算法有两个重要的优势：第一个，完全 distributed，Ceph 中的任何节点（client，MDS，OSD）都能独立的计算某个 PG 所在的 OSD 位置。第二点，CRUSH Map 的更新不会很频繁，只有在 OSD 节点 Down 掉，或者加入的时候，Map 表才会变化。就是因为 CRUSH 算法的这些优势，Ceph 能够同时解决数据应该存在哪的问题和已存数据在哪的问题，通过 CRUSH 的设计，能够将一些小状况对 Ceph 集群中已存在的 PG 数据的影响降低，从而使得因 OSD 的设备故障或者 OSD 集群的扩展导致的数据迁移也很少。当客户端读写一段数据时，先要对这段数据进行切分，切分成 Object 大小（一般是 4M），Inode 是数据段的唯一编号，切分后可得对象唯一编号 ObjectID。此时，只要知道对象所在的 OSD 即可完成定位。但是 OSD 节点并不是以 Object 来管理它本地存储的，因为一个 OSD 上存储的有上万个对象，若以对象来管理则会大大提高 OSD 节点的复杂性，因此必须提高管理的粒度。因此 Ceph 对对象进行了分组，即 Placement Group（PG）。PG 是 OSD 节点管理本地存储的基本单元，也是数据迁移的基本单元。PG 的大小不固定，但 PG 的数量有用户在启动时配置。PG 的数量已确定，一个 Object 想要知道它所在的 PG，只需要经过一个简单的 hash 即可确定，得到它所在的 Pool_PGID。一个 Ceph 集群可能被多个应用使用，多个应用之间应该不能可到彼此的数据，因此就产生了存储池 Pool，可以将 Pool 理解成不同的命名空间，不同的 Pool 之间不能看到对方的数据。因此 Object 定位后得到 Pool_PGID，PG 是 OSD 管理的基本单元，而 CRUSH 算法也是针对 PG 进行的。此时，客户端与 Monitor 通信获取 CRUSH Map 和 CRUSH Rule（此处是先对比版本号，若不一样则获取）。CRUSH 通过 Map 和 Rule 即可计算出 PG 所在的 OSD 节点，Ceph 定位数据时，将传统的查表方法变成了一种计算，大大提高了定位的效率。Rule 是由 Ceph 启动时配置的，一般不变，Map 描述了整个集群中 OSD

的布局信息，只有在 OSD 节点加入或者退出时，Map 才会发生变化。CRUSH 算法得出的第一个 OSD 节点即是该 PG 所在的 Primary 节点，其他为副本节点。一个 OSD 即是某些 PG 的 Primary 节点，又是某些 PG 的 Replica 节点。

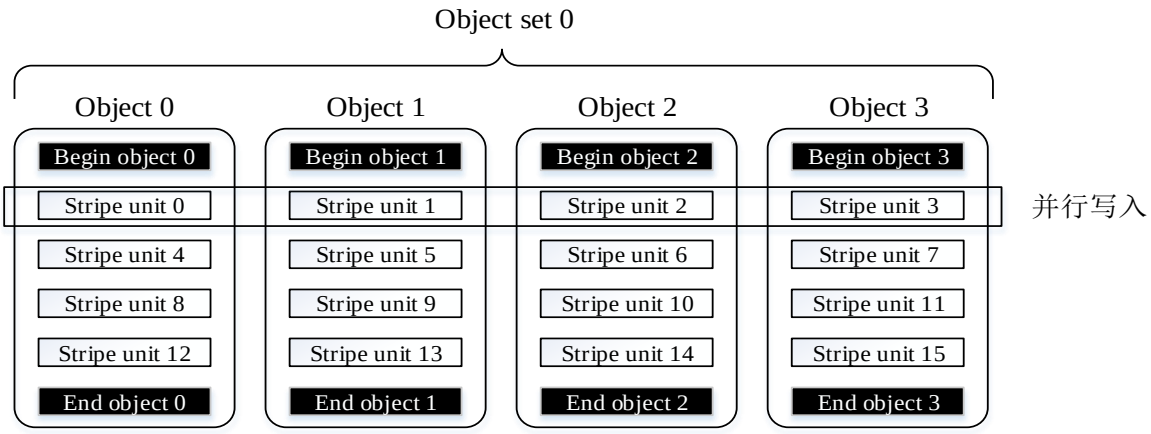

图 3.4 数据条带化

Ceph 存储系统跨越了多个存储设备，为了尽可能的发挥每个设备的读写性能，Ceph 进行了数据条带化，数据条带化必然想到了 RAID，Ceph 的数据条带化与 RAID0 比较相似。Ceph 在客户端对数据进行条带化，并且划分了对象集，如图 3.4 所示。当客户端并行写入一个条带时，因为每个条带单元属于不同的 Object，不同的 Object 可能属于不同的 PG，最终定位到不同的 OSD 节点，所以可以结合多个 OSD 磁盘设备驱动器的吞吐量，达到更快的读写速度。在 Ceph 可以使用三个参数配置 Ceph 的数据条带化，分别是对象大小，条带大小，条带计数。

从 Inode 切分成 Object 再哈希到 PG，从 PG 再 CRUSH 到 OSD 节点，完成数据的定位，最后条带化对象进行读写。客户端的写操作在 Primary OSD 进行，由于使用的是强一致性，读操作既可以在 Primary OSD，也可以在 Replica OSD，Ceph 的客户端简单直接，使用的是 random。

3.2.2 OSD 原副本策略分析

OSD 的写操作使用的是强一致性策略，每次数据更新都必须等待所有的副本更

新完成。Ceph 并没有让客户端发送写操作到所有的副本节点，而是让它发往 Primary 节点，然后由 Primary 节点发往 Replica 节点。因为客户端和 OSD 的通信使用的是外网，网络开销大，而 OSD 与 OSD 之间的通信使用的是内网，开销小。

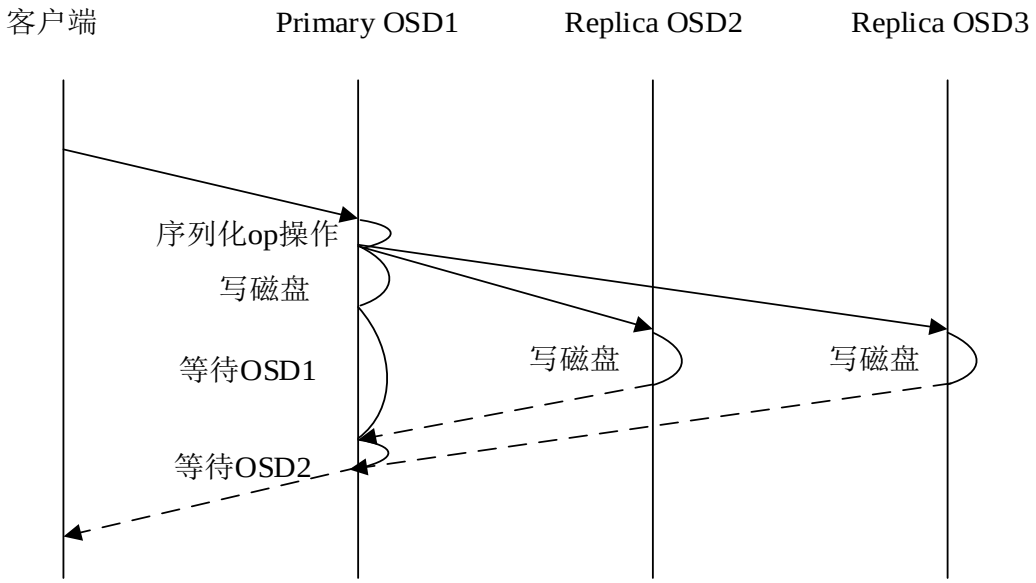

图 3.5 OSD 的 Primary-Copy 副本策略

图 3.5 所示的是客户端的更新操作时的副本策略，客户端所有对 PG 的写操作，会首先发往 Primary OSD 节点，Primary 节点的确定是由 CRUSH 算法确定的，由 PG 映射到 OSD 时的第一个 OSD 即是 Primary 节点。然后由 Primary 节点序列化对 PG 的 op 操作即生成事务，并且封装成 MOSDSubOp 发往副本节点。主节点和副本节点会分别执行事务，通过 FileStore 模块对数据进行更新。主节点对数据进行更新后会等待副本节点的响应，副本节点对数据进行更新后会对主节点进行回复，当主节点收到所有副本的更新回复后才会响应客户端本次写操作完成。由于网络拥塞的不确定性，不可能保证副本 OSD 同时收到主 OSD 的消息，或者主 OSD 同时收到副本 OSD 的回复。由此可见，Ceph 写延时主要由网络延时，写磁盘延时，等待副本延时构成。这次研究主要针对等待副本延时[37,38]进行优化。

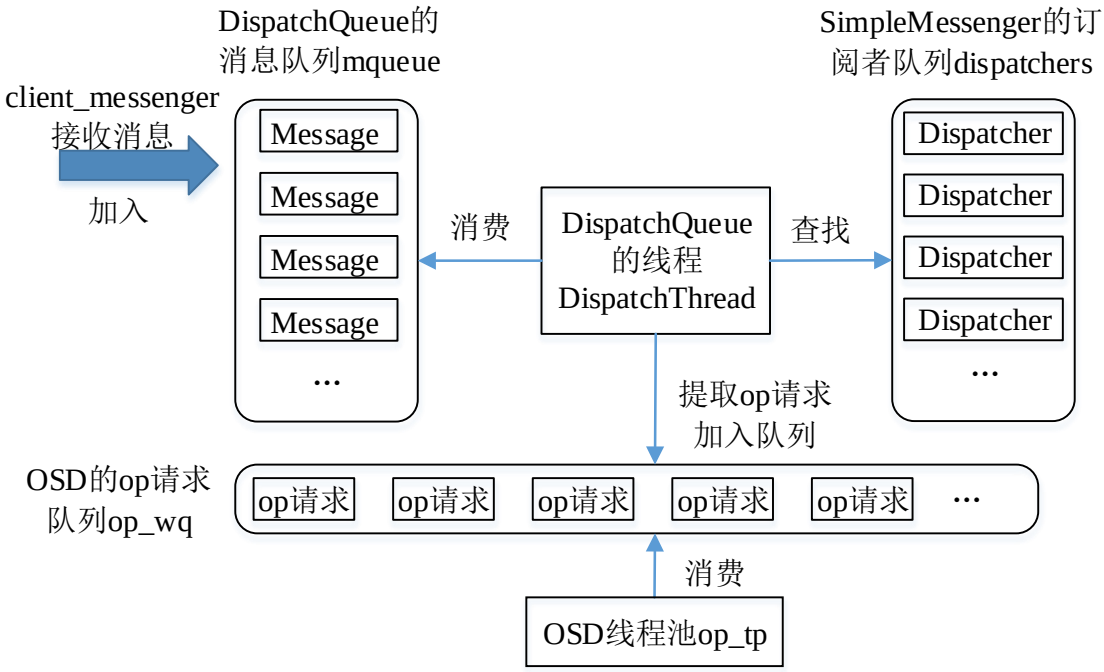

图 3.6 Primary OSD 消息接收

要想修改副本策略，就必须先了解客户端的读写请求在 OSD 端所经过的 IO 路径。首先是网络层 SimpleMessenger，如图 3.6 所示，客户端发过来的消息是由 client_messenger 接收的，然后由它向下 Dispatch。Ceph 中大量使用了生产者消费者模型，生产者 client_messenger 首先将接收到的消息 Message 加入到消息队列，然后由线程 DispatchThread 消费，消费一个消息时，先从消息订阅者队列中匹配该消息的订阅者，如果订阅者是 OSD，则提取消息中的 op 操作，加入到 OSD 中的 op 操作队列 op_wq，该队列将由 OSD 的线程池 op_tp 消费。通过上面的模型可以了解到，Ceph 的网络层除了使用“订阅者-发布者”设计模式，对网络中消息包的处理上采用的是比较传统的“生产者-消费者”线程模型，每次新的消息到来就会有创建一对“收-发”线程用来处理消息的接受和发送，如果有大量的消息请求，线程的创建和销毁以及线程的上下文切换会带来大量的系统性能开销，此处可能产生 Ceph 性能的瓶颈，但是他不在本文的研究范围之内。在 op_wq 中定义了该队列的消费方法_processing()，线程池中的消费线程都会调用该方法。在 PrimaryOSD 中每个写操作的 op 请求处理如图 3.7 所示。

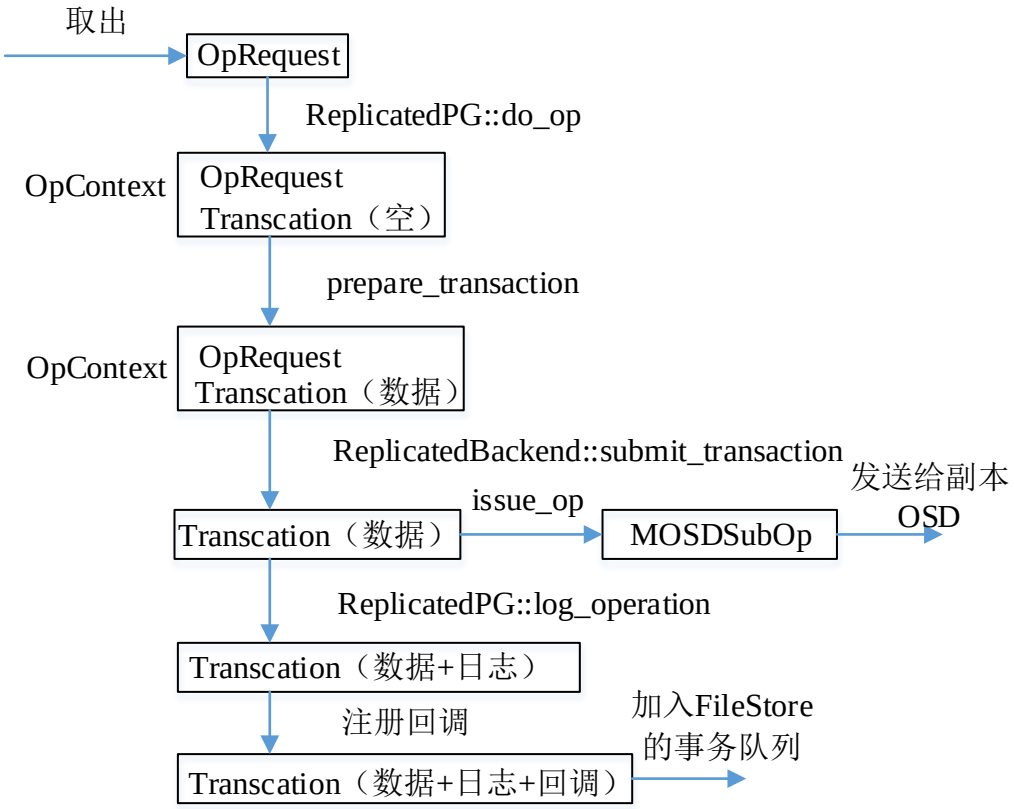

图 3.7 Primary OSD 处理 op 请求

首先从 op_wq 中取出一个 op 请求，将他封装成 OpContext，OpContext 起到一个承上启下的作用，在 OpContext 中有一个空的事务，而最终目的就是将 OpRequest 序列化成完整的事务，然后调用 ReplicatedPG 的 prepare_transaction 将 OpRequest 数据提取出来封装到 Transaction 的 bufferlist 中，此时 Primary OSD 即完成序列化 op 操作，可以将事务发送给副本 OSD。发送时先将整个 Transaction 转化成字节数据封装到 MOSDSubOp 中，并将该 PG 的操作日志 log_entries 也封装进去，然后查找该 PG 的副本 OSD，调用上层 OSD 类的 client_messenger 将 MOSDSubOp 循环发送给副本 OSD。在将数据发送给副本前，主节点还要将这个 op 请求加入到 in_progress_ops 队列中，即标记为正在进行的 op。一旦该 op 即写入了 Primary OSD，又写入了 Replica OSD，则可以从该队列中取出并调用回调函数，回复给客户端写操作成功。将 Transaction 发送给副本后，Primary OSD 对它还要进行进一步的封装，通过 ReplicatedPG 的 log_operation 调用 PGLog 在 Transaction 中写入 PG 的日志，最后为

该 Transaction 注册回调函数，回调函数会在 FileStore 完成该 Transaction 时调用，到此整个 Transaction 的封装完成，等待 FileStore 的执行。

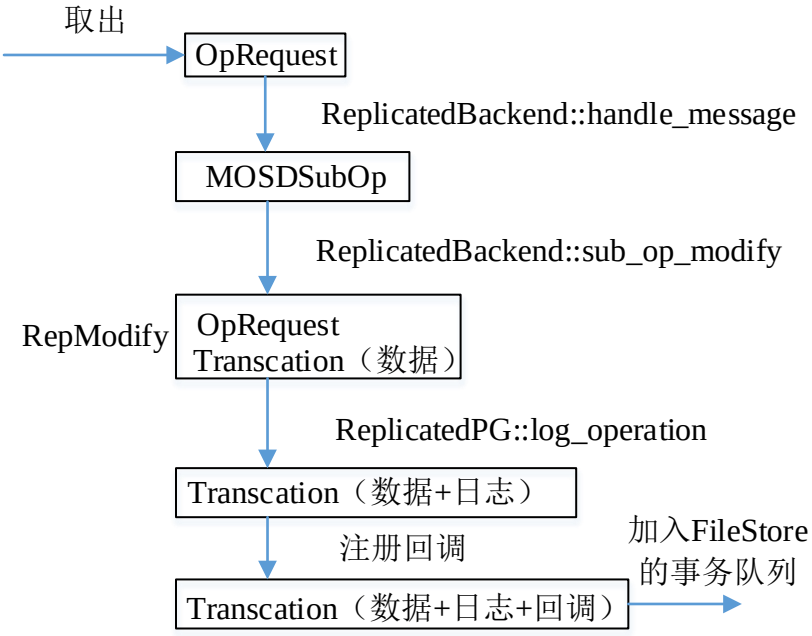

图 3.8 Replica OSD 处理 op 请求

Primary OSD 要在 Transaction 封装 PG 日志和回调函数之前把它发送给 Replica OSD，Replica OSD 处理该请求如图 3.8 所示。Replica OSD 并不是由 client_messenger 接收消息，而是有 cluster_messenger 接收的，Primary OSD 发来的消息属于集群消息，接收后依然加入了 OSD 的 op 请求队列 op_wq，处理的时候根据消息分类处理，先将请求强制类型转换成 MOSDSubOP，然后加入些状态信息封装成 RepModify，此时事务在 Primary OSD 封装的有数据，只需要添加日志即可，最后添加回调函数交给 FileStore 处理。FileStore 处理完成后，会调用回调函数，通过 OSD 的集群通信模块 cluster_messenger 返回消息给 Primary OSD。

Primary OSD 在收到 Replica OSD 发来的消息后，依然是 cluster_messenger 接收，然后添加到 op_wq 中由线程池处理，如果是副本 OSD 发来的回复消息，它会先查找 in_progress_ops 队列，该队列的结构如图 3.9 所示，先根据事务编号查到对应的节点，然后根据消息类型（ACK 还是 Commit），将 OSD 编号从等待队列中删除，一旦该节点的队列为空，则将调用相应的回调函数返回消息给客户端，数据已经写入所有的副

本，最后从 in_progress_ops 队列中删除该节点。Primary OSD 的 FileStore 处理完事务后，会调用该事务注册的回调函数，回调函数也是对 in_progress_ops 队列进行操作，查到该事务对应的节点，操作 waiting_for_commit 和 waiting_for_applied 两条等待队列，一旦为空则调用 on_commit 回调函数或 on_applied 回调函数。

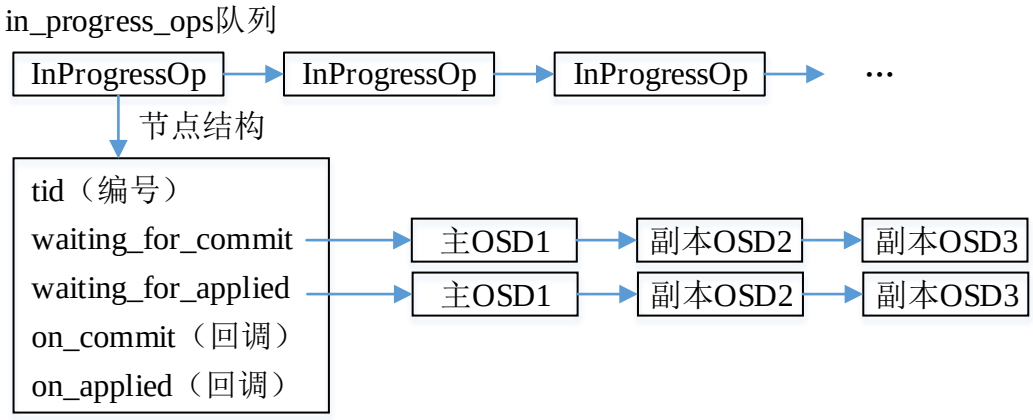

图 3.9 Primary OSD 的同步更新等待队列 in_progress_ops

3.3 动态副本策略设计

研究的目的主要针对上面提到的等待副本延时进行优化，修改副本策略。Ceph 使用的主从同步复制策略虽然达到了强一致性，但是写响应时间比较长，必须等待所有的副本响应，写操作才算完成。在相当一部分的应用中，如果响应时间太长，将直接影响到用户的体验。根据 CAP 理论，可以适当的减弱这种强一致性，来达到更高的可用性。亚马逊的 Dynamo 存储系统使用的是 NWR 策略则很好的平衡了 C 和 A。由用户配置[39]写副本的数量 W 和读副本的数量 R，只要 W+R>N，则会实现最终一致性。简单说就是写操作要同步写 W 个副本，异步写 N-W 个副本，读操作要读 R 个副本，并比较版本，得到最新数据。在 Dynamo 中，每个 OSD 节点都是对等的，不存在主节点的概念，因此它的写操作随机写了 W 个副本，读的时候要比较版本号。Ceph 可以借鉴这种策略，N 个副本并不需要全部同步更新，同步更新 W 个副本，异步更新 N-W 个副本。 Ceph 使用了 CRUSH 算法，第一个 OSD 节点是主节点，OSD 节点顺序不变，因此写操作可以同步写前 W 个副本，读操作时随机读前 W 个副本

即可读取到最新的数据，在这里由于 CRUSH 的优势，省略了 Dynamo 读操作中比较版本号的一个过程。这种策略让用户自己平衡读写，当写操作较多时，可以配置更小的 W，读操作较多时，配置大的 W，以提高读吞吐率。如果更进一步让系统自己根据读写请求的负载来动态的调整写副本的数目从而满足不同的用户需求，让大多数用户得到更佳的用户体验。

在目前已有的分布式环境中，不论是分布式计算，还是分布式存储，为了数据的安全性要么采用了副本技术要么采用编码技术，但是编码技术带来的性能下降对某些用户而言不可容忍，因此在生产环境中大多数采用了副本技术。在当前大量的分布式存储系统中，它们使用的副本一致性策略大多都不知变通，要么是强一致性，要么是弱一致性，或者是最终一致性，它们只针对固定的用户群，应用环境一旦发生变化，它就不适用了。而 Ceph 作为一个统一存储的系统并不能满足当前对云存储的多样化环境需求。因此，Ceph 的副本一致性策略应该更加多样化，平衡一致性和可用性，从而提供更好的系统性能。因此，本文受 NWR 副本策略启发设计了一种基于读写比例的动态副本策略，因为较高的一致性带来的是较高的写代价，在系统中有大量的写操作时可以适当降低一致性，提供更好的写性能，当系统中有大量的读操作时可以提高一致性，用户读任意一个副本都能读到最新的数据，提供更好的读性能。

因此研究的主要目的就是修改 Ceph 的副本一致性策略，将它改为动态自适应的副本一致性策略，让它能适应更加多样化的使用环境。该策略根据 Ceph 系统运行过程中的读写负载的比例来动态的调整副本实时更新的数量。比如在 Ceph 的三副本中，系统启动时，采用强一致性，实时更新三个副本，如果写负载过重超过一定的阈值，则实时更新主副本和一级副本，此时客户端读只能读这两个副本，如果写负载再次增大则再次减少同步写副本的数量，但是不能小于写副本数的下限。读写副本的数量，在 Ceph 运行的过程中根据刷新时间间隔不断动态调整。因此，在进行动态副本一致性的设计之前首先规定了这样几个配置参数：副本数 replicaNum，写副本数目 writeNum，是否打开动态副本 isDynamic，写副本数目的上限 writeNumUp，写副本数目的下限 writeNumLower，写密集型阈值 writeIntensive，读密集型阈值 readIntensive，

刷新时间间隔 interval。写副本数目的上限和写副本数目的下限只有在动态副本策略启用时发挥作用，Ceph 系统在动态修改写副本的数量时，写副本数量只能在用户配置的上限和下限之间。写请求的数量超过阈值，则系统进入写密集型，读请求的数量超过阈值，则系统进入读密集型状态。因此，Ceph 在运行的过程中有四种状态（如图 3.10）：读密集型，写密集型，读写密集型，读写疏松型。不同的状态中修改副本数的策略也是不同的。

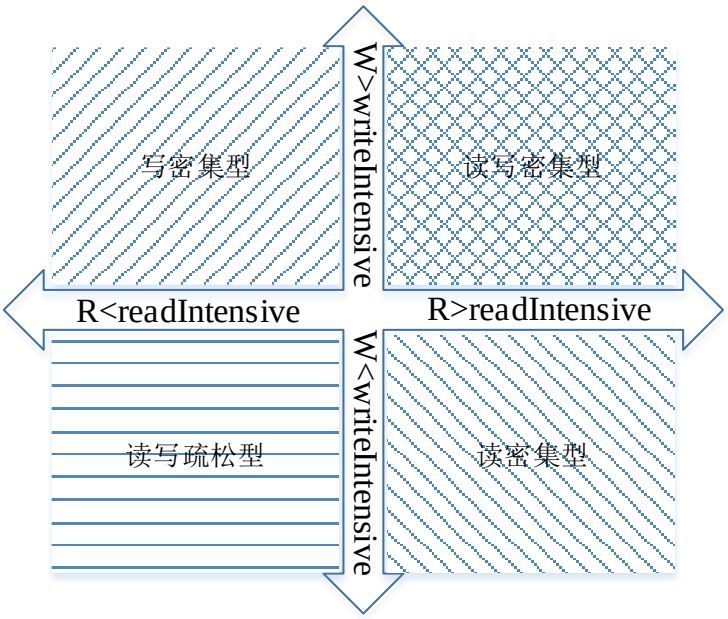

图 3.10 读写密集型的判定

（1）读密集型

此时，R>readIntensive&&W<writeIntensive，整个 Ceph 系统读请求比较多，客户端对 OSD 存储节点频繁的发起访问，而更新频率较低，读操作是整个集群的性能瓶颈，因此可以采用强一致性，读操作可以发往所有的副本节点，在每个副本节点上都能读到最新的数据。而写操作发往主 OSD 节点，由主节点同步给副本节点，所有节点的数据写入后才算更新完成。

（2）写密集型

此时，R<readIntensive&&W>writeIntensive，写请求数量超过阈值，而读请求数

量较少，Ceph 系统处于写密集型状态。由于读频率较低，因此同步更新所有副本没有必要，可以根据写操作数所占的比例来设置同步更新的副本的数量，写操作所占的比例越高，则写操作的代价越高，因此同步更新的副本应该越少，相反则越多。因此根据以上的关系，同步更新副本数与写操作所占的比例是成反比的，即与读操作数所占的比例成正比，因此设计的公式如 3-1 所示：

$$syncWrite = (N-1)R/(W+R-writeIntensive) \quad (3-1)$$

其中，syncWrite 是主副本需要把同步更新的副本数，N 是用户设的副本总数，W 是该时间段内些操作数，R 是该时间段内的读操作数，writeIntensive 是设置的写密集型的阈值，R/(W+R-writeIntensive)就是超过阈值后，读操作数所占的比例。因为 W>writeIntensive && R>=0，所以 syncWrite 大于等于 0，小于 N-1。当 R=0 时，即本时间段内全部是写操作，syncWrite 等于 0，主副本不需要同步更新其他副本，只需要更新主副本即可，其他 N-1 个副本异步更新。当 W 等于 writeIntensive 时，syncWrite 等于 N-1，主副本需要同步更新其他所有副本。

根据以上的公式，当系统处于写密集型时，写操作数所占的比例越大，同步更新的副本数越少，写操作的延迟更低，表现出更好的写性能。

（3）读写密集型

此时，R>readIntensive&&W>writeIntensive，Ceph 系统中读写操作数都超过了阈值，数据即存在大量的更新，又存在大量的访问，因此需要在读写性能之间进行权衡。依然是根据超过阈值后，读写操作各自占的比例进行，读操作所占比例越高，则更偏向于读性能，相反则偏向于写性能。设计的公式如 3-2 所示：

$$syncWrite = (N-1)(R- readIntensive)/(W+R-writeIntensive- readIntensive) \quad (3-2)$$

其中，readIntensive 是设置的读密集型的阈值，其他参数同公式 3-1。当读写操作数都超过阈值时，若读超过的越多，即(R- readIntensive)越大，则 syncWrite 越大，反之越小。最终 syncWrite 的取值依然在 0 到 N-1 之间。

根据以上公式，当系统处于读写密集型时，要对读写操作所超过阈值的多少进行比较，读操作超过阈值多则偏向读性能，同步写更多的副本，若写操作超过阈值多，

则偏向于写性能，异步写更多的副本。

（4）读写疏松型

此时，R<readIntensive&&W<writeIntensive，Ceph 系统中数据的更新和数据的访问频率都很低，更改同步的副本数并不能带来多少性能的提高，每次变换同步更新的副本数都是要消耗一定的代价的，反而可能带来性能的下降，得不偿失。

因此，当系统处于读写疏松型时，不需要改变系统中同步更新的副本数，保持系统当前的状态即可。

3.4 Monitor 端设计

当确定好以上的副本策略后，可以进行底层的设计了，OSD 动态自适应副本策略的实现主要就在 Monitor 端。因为 Monitor 是 Ceph 集群的管理者，管理着 Ceph 集群大量的状态信息，它能方便的与客户端和 OSD 端进行通信。客户端和 OSD 端都是 Monitor 的被管理者，每当副本策略变化时，Monitor 可以通知 OSD 端和客户端进行策略的变更。

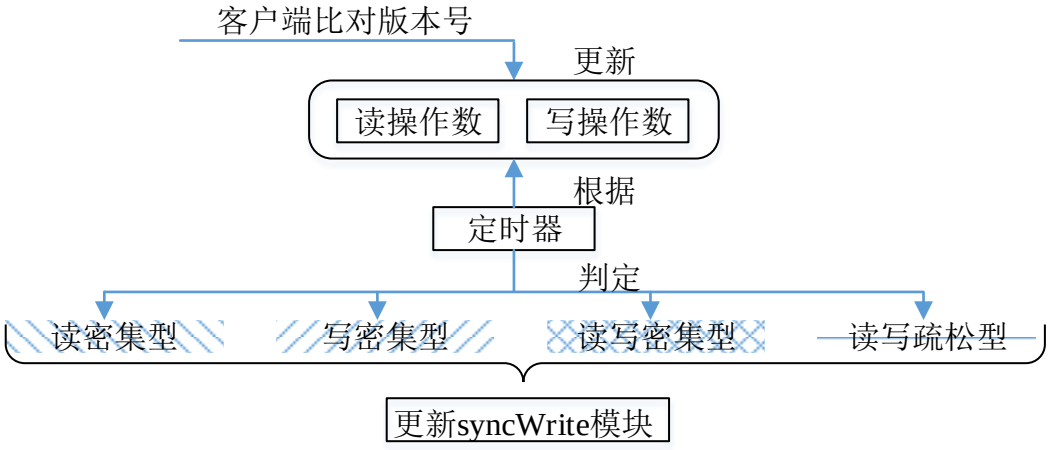

图 3.11 Monitor 端副本策略的设计

Monitor 端副本策略模块 ReplicaStrategy 的设计如图 3.11 所示。因为客户端每次进行读写操作时，都会访问 Monitor 以确认本地的 CRUSH Map（也叫 OSDMap）是最新的。此时客户端会向 Monitor 发送一个 OSDMap 版本确认的消息，可以在该消

息中附加一个读写标记，读写标记表示客户端本次操作是读操作还是写操作，以便 Monitor 对读写操作数进行统计。Monitor 端给客户端发送回复消息时附加上最新的同步写副本数 syncWrite。如此在 Monitor 端可以成功的进行读写操作数的统计，然后设置一个定时器，在一个定时的时间段内根据读写操作数判定整个 Ceph 系统所处的读写密集型状态，最后根据相应的策略计算出应该设置的同步写副本数 syncWrite 的值。在更新 syncWrite 的值时，既需要更新客户端的 syncWrite 又需要更新 OSD 端的 syncWrite。客户端的 syncWrite 在客户端进行 CRUSH Map 版本号比对的时候即可完成更新。OSD 端的 syncWrite 更新，可以通过 Monitor 端的 OSDMap，将 syncWrite 发送给每一个 OSD 端，OSD 端完成 syncWrite 的更新后发送回复消息给 Monitor 端，当收到所有 OSD 端的回复时，则 OSD 端的 syncWrite 更新完成。

OSD 端 syncWrite 的更新和客户端 syncWrite 的更新有一个先后顺序，这里分两种情况：第一种是 syncWrite 增大的情况，此时同步更新副本数增大，要先更新 OSD 端的 syncWrite，更新完成后再更新客户端的 syncWrite，若先更新客户端的，OSD 端还未更新，客户端可能访问到异步更新的副本。第二种是 syncWrite 减少的情况，此时先更新客户端的 syncWrite，然后更新 OSD 端的，否则客户端还是可能访问到异步更新的副本。

3.5 OSD 端设计

Monitor 端设计好后就要对 OSD 模块进行修改。Monitor 每次修改副本策略都会对 OSD 发送同步写副本数，OSD 端既要处理 Monitor 发来的消息还要向 Monitor 发送回复消息。在 OSD 端，原系统的写操作只能同步更新，需要为他添加一个异步更新的模块。同步更新副本的流程依然同 3.2.2 分析的一样，但是若是异步更新的副本，则需要将要发送的 MOSDSubOp 消息以及异步更新的 OSD 编号加入到异步更新的模块 AsyncReplica 中。该模块的的设计如图 3.12 所示。

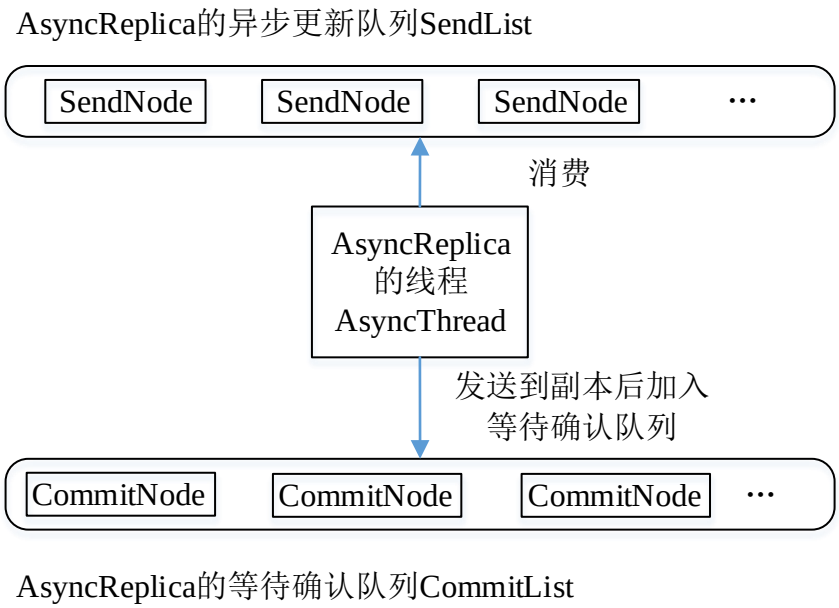

图 3.12 Primary OSD 异步更新模块 AsyncReplica

该模块使用了传统的生产者消费者模型，当 Primary OSD 发送事务消息给副本时先要进行判断，如果是要同步更新的副本则直接发送给副本同时加入队列 in_progress_ops 中，等待同步确认。如果是要异步更新，则将需要要异步更新的副本以及要发送的消息 MOSDSubOp 封装成 SendNode，并且加入异步更新队列中。AsyncReplica 的消费者线程对该队列进行消费，一旦该消息发送给副本，则生成一个 CommitNode，加入等待确认队列，等待异步确认。当 Replica OSD 异步更新完成后，会发送回复给 Primary OSD，Primary OSD 收到回复后会在 AsyncReplica 的等待确认队列中进行确认并且删除，这样一次异步更新完成。

异步更新模块不仅需要具有异步更新副本的功能，还需要一个能切换同步更新副本数（syncWrite）的功能。当 Ceph 集群的读写状态从一种变化成另一种时，同步更新的副本数 syncWrite 发生变化，这种副本数的变化，即策略的变化是由 Monitor 统计的，再由它发送消息给所有的 OSD 节点。当 Primary OSD 节点接收到 syncWrite 变化的消息时，需要修改同步更新的副本数，它会在 AsyncReplica 的发送队列的末尾中加入一个特殊的节点 ChangeNode。当消费线程处理到该节点时则表明，策略变化之前的消息已经全部发送，此时将 ChangeNode 加入确认队列 CommitList 的末尾，

一旦 ChangeNode 之前的节点全部确认，则表明策略变换完成，可以发送回复消息给 Monitor 该 OSD 节点的策略变换完成。

3.6 Client 端设计

客户端的读写流程在 3.2.1 中已经分析过，每次客户端读写数据之前会先访问 Monitor 以确认它的 CRUSH Map 是最新的，最少会比对一次版本号。客户端进行写操作时，会将写操作的 op 发送给 Primary OSD，有 Primary OSD 进行更新。客户端进行读操作时，Ceph 使用的是随机方法 random()%N，即客户端会随机读取同步更新的副本中的一个。因此 Monitor 端策略的变化对客户端写操作没有影响，对读操作有影响。

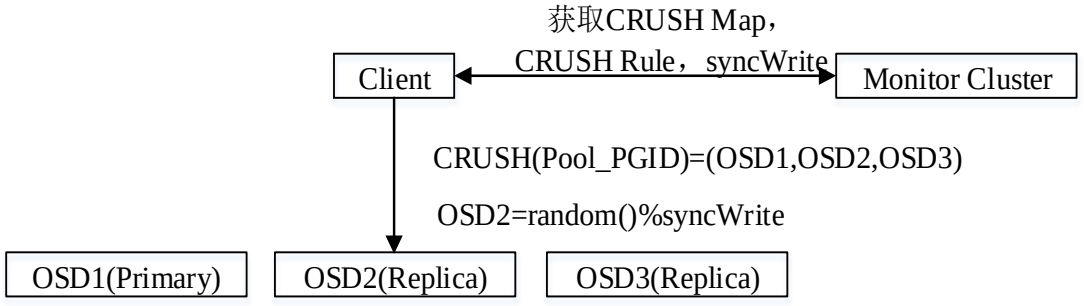

图 3.13 客户端读请求策略

当 Monitor 端经过一个时间段的读写统计，同步更新的副本数发生变化时，客户端的读请求不能发往异步更新的副本，因为它可能访问到旧的数据而不是最新的数据，因此只能发往同步更新的副本节点。设计客户端的读操作如图 3.13 所示， Client 端要想获得最新的同步写副本数就必须与 Monitor 端进行通信。Client 端每次进行读写操作时都会向 Monitor 端发送消息以比对 OSDMap 的版本号，该消息中就附加了读写操作的标记，Monitor 端收到后会进行读写操作的统计，而返回消息中则附加了最新的同步写副本数 syncWrite，客户端收到后可以更新本地的同步写副本数。这样当客户端的读请求发往 OSD 时，使用随机方法 random()%syncWrite，将读请求发往同步更新的 OSD 节点，从而排除发送给异步更新的 OSD 节点的可能性，因此总能

访问到最新的数据。

3.7 本章小结

本章首先从功能上分析了 Ceph 的逻辑系统架构，并介绍了 OSD 端的系统架构，然后分析了客户端的读写请求到 OSD 端的定位，引出了 Ceph 系统 OSD 端的副本策略，并提出了一种动态副本策略的方案，最后对该方案进行了详细的设计。

4 Ceph 副本一致性优化实现

本章将对第三章设计的方案进行实现，第三章从 Monitor 端，OSD 端，客户端进行了分析和设计，本章将从 Monitor 端开始进行实现。本章根据上一章设计的副本策略，分别对 Ceph 的 Monitor 端，OSD 端和 Client 端进行了修改，详细介绍了各个模块的关键类和处理流程。

4.1 Monitor 端的副本策略实现

在 Monitor 端首先添加一个参数配置的类 ReplicaConfig，如表 4.1 所示，它主要用来配置副本策略所需要的参数。

表 4.1 ReplicaConfig 的关键字段

名称	类型	说明
replicaNum	unsigned int	副本数量
writeNum	unsigned int	同步写副本的数量，剩下的异步更新
isDynamic	bool	是否开启动态副本策略
writeNumUp	unsigned int	同步写副本数的上限
writeNumLower	unsigned int	同步写副本数的下限
writeIntensive	unsigned long long	写密集型的阈值
readIntensive	unsigned long long	读密集型的阈值
interval	unsigned int	定时器的刷新时间，单位分

动态副本策略的开启是由 isDynamic 字段控制的。当 isDynamic 为 false 时，关闭动态副本策略，同步写副本的数量 writeNum 由用户控制，其他的副本异步更新。当 isDynamic 为 true 时，开启动态副本策略，由 Monitor 统计客户端的读写操作数，并设定同步更新的副本数，此时该字段下方的字段才有用。writeIntensive 是写操作数的阈值，超过了则系统处于写密集型状态。readIntensive 是读操作数的阈值，超过了则系统处于读密集型状态。interval 是定时器的时间设定，每隔该时间段，根据系统

的状态设定同步更新的副本数。writeNumUp 和 writeNumLower 是用户强制规定的同步更新的副本数的上下限，每次更新同步副本数时必须在该上下限之间，否则取上限或者下限。

Monitor 端动态副本策略主要由 ReplicaStrategy 类控制，该类的关键数据结构如表 4.2 所示。

表 4.2 ReplicaStrategy 的关键字段

名称	类型	说明
mon	Monitor *	指向 Monitor 类型的指针
strategyConfig	ReplicaConfig *	用户的配置参数
readCount	unsigned long long	该时间段统计的读操作数的数量
writeCount	unsigned long long	该时间段统计的写操作数的数量
countMutex	Mutex	改变读写操作数的锁
monSyncWrite	unsigned int	Monitor 端设定的同步写副本数
clientSyncWrite	unsigned int	客户端同步写副本数
osdSyncWrite	unsigned int	OSD 端同步写副本数
sendSyncWr	SendSyncWrite *	更新同步写副本数模块
timer	SafeTimer	定时器

其中 mon 是指向 Monitor 类型的指针，以便调用 Monitor 的 public 方法。strategyConfig 是用户配置参数的指针，该指针有 Monitor 传递过来。readCount 和 writeCount 是用来统计一个定时器时间段内的读写操作数的，countMutex 是这两个参数的锁，每次定时时间的开始都要重新计数，对这两个数清零。monSyncWrite 是该时间段内根据策略计算的同步写副本数，osdSyncWrite 当前 OSD 端的同步写副本数，clientSyncWrite 是客户端每次比对 OSDMap 的版本号时，Monitor 回复给客户端的同步写副本数。每次副本策略发生变化时，Monitor 并不会主动更新同步写副本数给客户端，而是在回复客户端的 OSDMap 版本号确认时附加到回复消息上。sendSyncWr

是发送同步写副本数给 OSD 的模块，每次副本策略发生变化，就有该模块更新 osdSyncWrite 到 OSD 端。timer 是定时器，每到用户的定时，就会根据读写次数变换副本策略。

4.1.1 Monitor 与 Client 端消息处理

整个 Ceph 的网络模块都是由 SimpleMessenger 实现的，Monitor 端也是通过它接收消息的。如果消息类型是 CEPH_MSG_MON_GET_VERSION，那么消息就可以强制类型转换为 MMonGetVersion 类型，该类中封装了读写标记，据此就可以判断客户端进行的是读操作还是写操作。MMonGetVersion 的关键字段如表 4.3 所示。

表 4.3 MMonGetVersion 的关键字段

名称	类型	说明
handle	ceph_tid_t	通信的唯一编号
what	string	请求的是哪张 Map 表
isWrite	int	该时间段统计的写操作数的数量

Monitor 端管理了六张表，分别是 MONMap，OSDMap，PGMap，LogMap，AuthMap，MDSMap。客户端一般会比对 OSDMap 的版本号，what 就等于 OSDMap，客户端的读写标记为 isWrite，“1”为写操作，“0”读操作，“-1”是默认。Monitor 在处理该消息时，会先查找 OSDMap 的版本号，并且封装成 MMonGetVersionReply 消息回复给客户端，该消息的关键字段如表 4.4 所示。

表 4.4 MMonGetVersionReply 的关键字段

名称	类型	说明
handle	ceph_tid_t	通信的唯一编号
version	version_t	OSDMap 的最新版本号
oldest_version	version_t	OSDMap 的最老版本号，第一次版本号
clientSyncWrite	unsigned int	客户端同步写副本数

向客户端发送回复消息时，会附加上客户端同步写副本数 clientSyncWrite，客户

端不仅会比对 OSDMap 的版本号，还会更新本地的同步写副本数。

Client 每次进行读写操作时都会发送本地的 OSDMap 的版本号与 Monitor 端进行比对，同时在版本号上附加了读写标记，Monitor 端从而进行读写统计。Monitor 端读写次数的统计和 Client 同步写副本数的更新流程如图 4.1 所示。

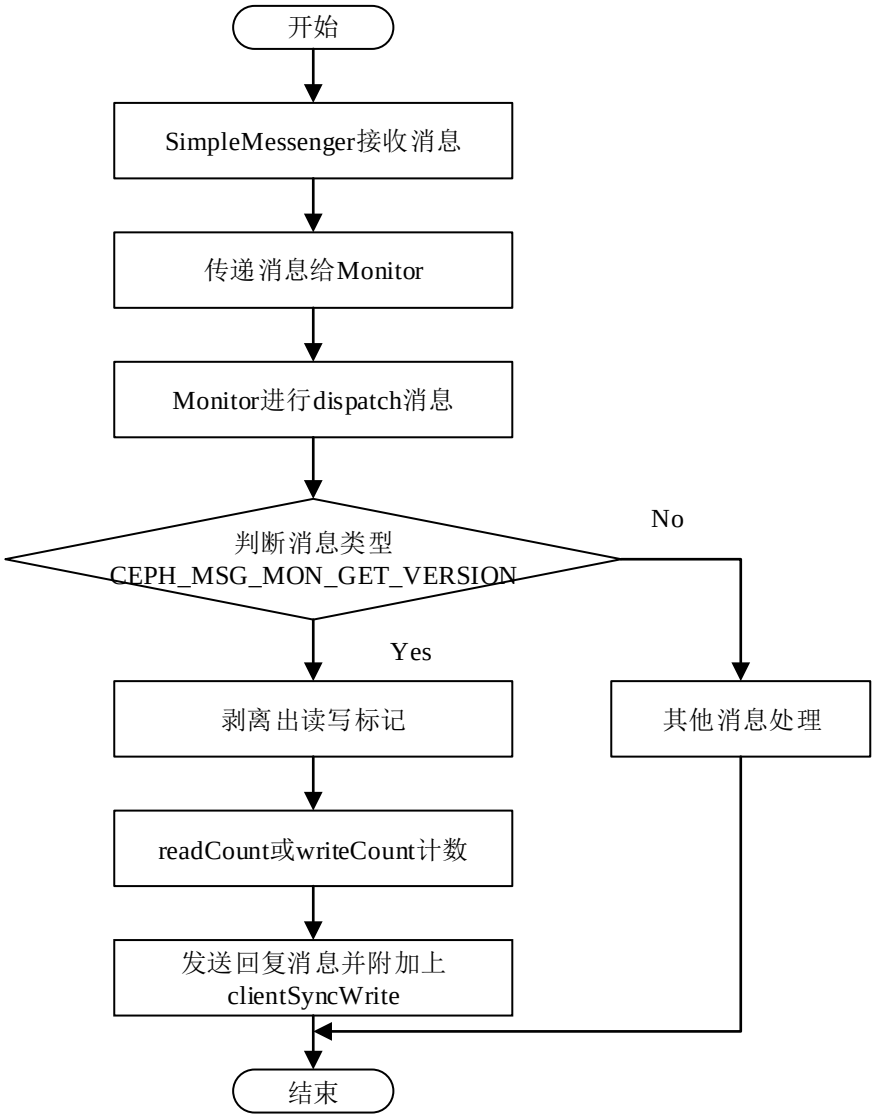

图 4.1 Monitor 端读写操作数计数

4.1.2 Monitor 动态副本策略的实现

Monitor 端对客户端的读写操作数的统计以及更新客户端同步写副本数通过以上流程即可完成。Monitor 端 ReplicaStrategy 的定时器会定时的根据读写操作数计算

出副本策略的变化，并将结果通过更新模块更新出去。ReplicaStrategy 定时器绑定函数的处理流程如图 4.2 所示。

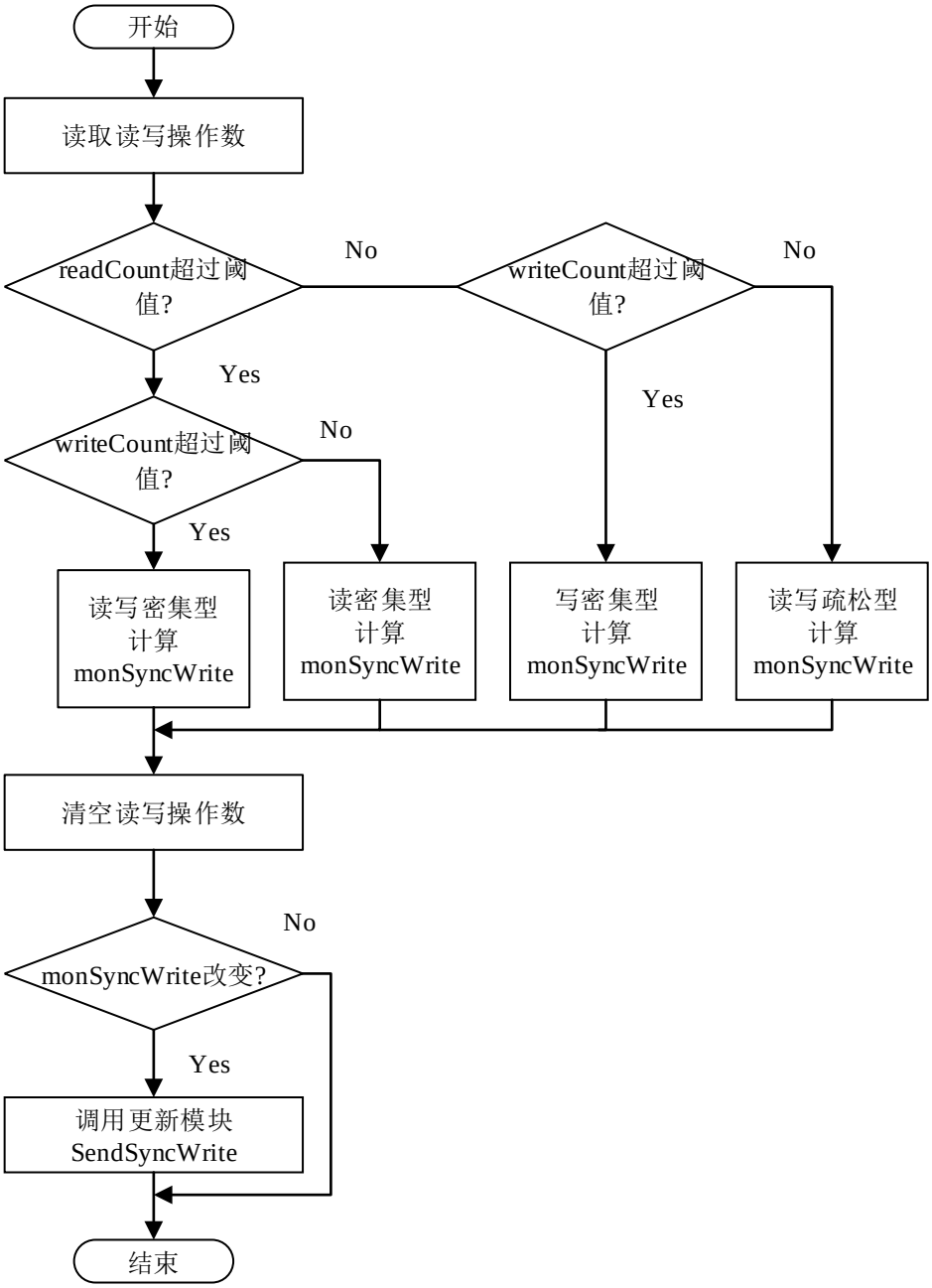

图 4.2 ReplicaStrategy 副本策略的判定

定时器每到时间都会根据读写计数器的进行判定，判定系统处在读写密集型、读密集型、写密集型和读写疏松型中的那种状态，然后根据设计的算法计算出同步写副本数 monSyncWrite 的值，如果没变则不需要做任何处理，若该值发生变化则可以通

过更新模块 SendSyncWrite 更新出去。

4.1.3 Monitor 与 OSD 端的消息处理

通过以上的副本策略，每计算出新的 monSyncWrite 时都要将它更新到客户端和 OSD 端。OSD 端同步副本数的更新主要通过 SendSyncWrite 进行。SendSyncWrite 类的关键数据字段如表 4.5 所示。

表 4.5 SendSyncWrite 的关键字段

名称	类型	说明
ptrRepStrategy	ReplicaStrategy *	ReplicaStrategy 类型的指针
sendSet	set<int>	需要发送 osdSyncWrite 的 OSD 集合
commitSet	set<int>	发送消息后等待确认的 OSD 集合
replicaUp	bool	同步写副本数是否增加
sendMutex	Mutex	更新模块的锁，每次只能更新一次

其中，ptrRepStrategy 是指向策略模块的指针，以便调用该模块的方法，OSD 要更新的同步写副本数是 ReplicaStrategy 计算出的 monSyncWrite，一旦更新完成后就将 ReplicaStrategy 的 osdSyncWrite 设为该值，表明所有 OSD 端的同步写副本数为该值。sendSet 是需要发送同步副本数的 OSD 集合，可以通过 ReplicaStrategy 的 Monitor 指针找到 OSDMap，再通过 OSDMap 统计出所有的 OSD 编号，从而得到 sendSet 集合。每次从 sendSet 中取出一个 OSD，向该 OSD 发送更新消息，然后将它加入到 commitSet 中，等待 OSD 返回确认消息，一旦所有的确认消息都收到了，则表明 OSD 端的同步写副本数更新成功。sendMutex 是该模块的锁，ReplicaStrategy 的定时器每到时间都会计算同步写副本数，一旦计算出新值都要通过该模块更新，因此该模块是可能重入的，如果两次更新混杂在一起是会造成更新的混乱的。因此发送更新消息时首先要上锁，每次更新 OSD 的同步写副本数时只能一次次的进行，必须等待本次更新完成后才能进行下一次的更新。SendSyncWrite 模块发送流程如图 4.3 所示。

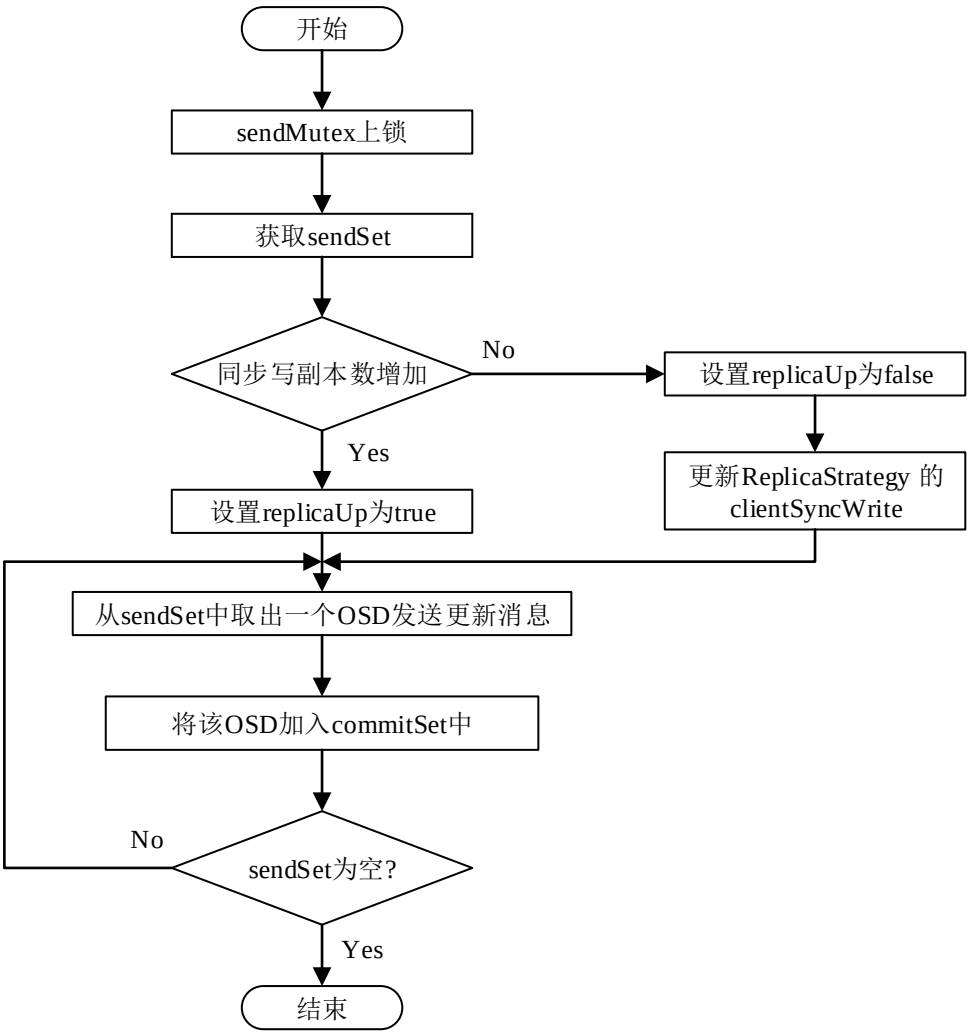

图 4.3 SendSyncWrite 模块更新副本数到 OSD 端

首先对更新模块上锁，表示对 OSD 端的更新开始，如果上次更新没有结束则必须等待。然后通过 OSDMap 获取要更新的 OSD 集合 sendSet。replicaUp 是同步写副本数是否增大的标记，该值增大和减小的处理是不一样的。增大时，要先更新 OSD 端的 osdSyncWrite，更新完成后再更新客户端的 clientSyncWrite，若先更新客户端的，OSD 端还未更新，客户端可能访问到异步更新的副本。减少时，先更新客户端的 clientSyncWrite，然后更新 OSD 端的，否则客户端还是可能访问到异步更新的副本。此模块发送给 OSD 的消息为 MOSDSyncWrite，这是重新定义的一个消息，消息类型为 CEPH_MSG_OSD_SYNC_WRITE，该消息中封装了 monSyncWrite 的值。当一个 OSD 端更新完成后会发送回复消息 MOSDSyncWriteReply，消息的类型为

CEPH_MSG_OSD_SYNC_WRITE_REPLY，该消息中封装了 OSD 的编号。该模块处理回复消息的流程如图 4.4 所示。

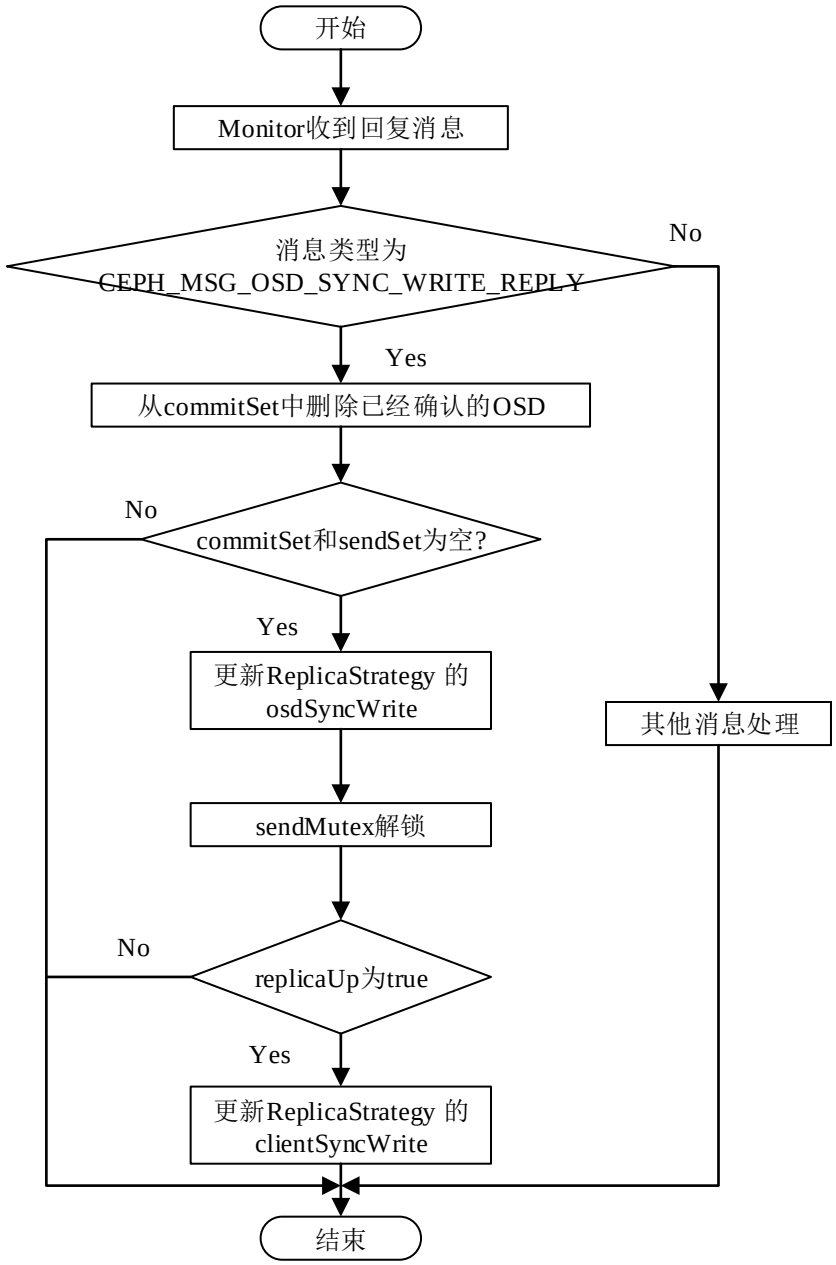

图 4.4 SendSyncWrite 模块处理 OSD 返回的回复消息

Monitor 发送更新消息的时候，每发送一个就将该 OSD 加入到 commitSet 中，等待确认。Monitor 每收到一个确认消息后，就会从该集合中删除该 OSD，一旦 commitSet 和 sendSet 都为空时，则表明 OSD 端的同步副本数更新完成，此时将

ReplicaStrategy 的 osdSyncWrite 设为更新后的值，并对 SendSyncWrite 模块解锁，表明可以进行下一次的更新，最后根据同步写副本数是否增大的标记 replicaUp，判断要不要更新客户端的 clientSyncWrite。

4.2 OSD 端的副本策略实现

Ceph 的强一致性策略使用的是同步更新，而新的副本策略是使用的是部分同步更新，其他的异步更新，而同步更新的副本数量是由 Monitor 计算出来的。客户端通过 CRUSH 定位到存放数据的 OSD，对于写操作它直接发往主副本即 Primary OSD，由它负责更新。主副本收到客户端的写操作后会对它进行序列化，最终封装成事务。对于主副本，直接执行事务即可完成主副本的写操作，对于同步更新的副本 OSD 则将事务封装成 MOSDSubOp 发送出去，并且加入同步更新的等待确认队列即 in_progress_ops，对于异步更新的副本则交由异步更新模块进行更新和确认。一旦同步更新的副本全部确认了即可给客户端发送回复消息，写操作完成。Primary OSD 更新副本的流程如下图 4.5 所示。

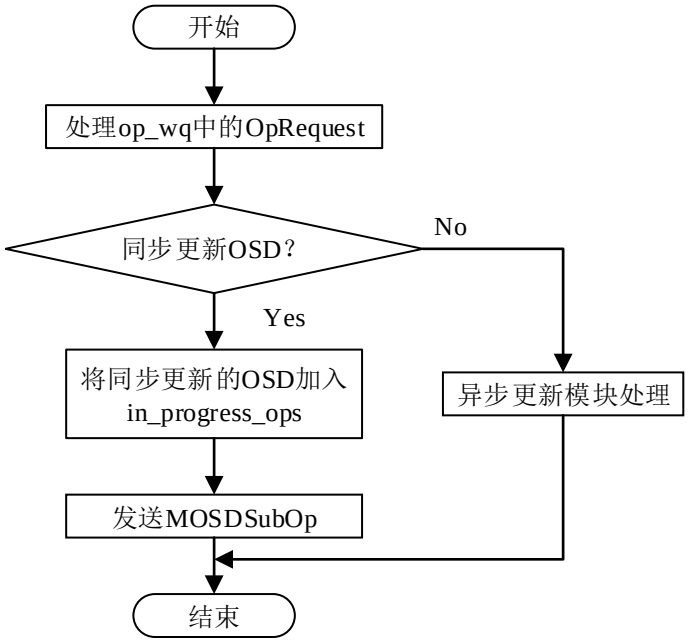

图 4.5 Primary OSD 更新 Replica OSD 的流程

新的副本策略要用到异步更新模块，根据 OSD 端的 osdSyncWrite 判断哪些副本

要同步更新哪些要异步更新，同步更新的 OSD 和系统原来一样处理加入到 in_progress_ops 队列中，并向同步更新的副本发送 MOSDSubOp 进行更新，而异步更新的副本则交由异步更新模块处理。

4.2.1 OSD 端异步更新模块的实现

异步更新模块不仅要负责向副本发送更新操作，还要接收副本发送的确认消息，并且还要处理同步写副本数的切换。在 Monitor 端实现了动态副本策略的计算，每当副本策略变化时，Monitor 端会对 OSD 端的同步写副本数进行更新，而每次变化时，异步更新模块都需要进行切换，切换完成后向 Monitor 端发送回复消息。在异步更新模块使用了“生产者-消费者”模型，异步更新模块 AsyncReplica 的关键数据字段如下表 4.6 所示。

表 4.6 AsyncReplica 的关键字段

名称	类型	说明
sendList	list<sendNode>	异步更新的 FIFO 链表
sendMutex	Mutex	sendList 的锁
commitList	list< commitNode>	异步更新等待确认链表
commitMutex	Mutex	commitList 的锁
hasChangeNode	bool	两条队列中有标记 ChangeNode 的节点
condThread	Cond	条件变量
asyncThread	AsyncThread*	异步更新的线程

异步更新模块将要异步更新的 OSD 以及要发送的更新消息封装成 sendNode，并将它加入 sendList 的末尾。异步更新线程 AsyncThread，每处理一个 sendNode 就生成一个 commitNode 并加入 commitList 中，等待确认。Cond 是该线程的条件变量，当 sendList 为空时，线程处于睡眠状态。当向 sendList 中添加节点后可以通过 Cond 唤醒线程。sendNode 的关键字段如下表 4.7 所示。

表 4.7 sendNode 的关键字段

名称	类型	说明
sendMessenge	MOSDSubOp*	封装好事务的消息
sendSet	set<int>	发送消息的 OSD 集合
isChangeNode	bool	是否是 ChangeNode 的标记

MOSDSubOp 是要发送给副本 OSD 的消息，主副本对于客户端发过来的 OpRequest 封装成事务，最后封装成 MOSDSubOp。同步更新则直接发送给副本，若要异步更新则将它封装到 sendNode 中，并将要更新的 OSD 封装到 sendNode 的 sendSet 中。isChangeNode 是一个标记，为 true 则标记此时收到了 Monitor 端发送的更新同步写副本数，此时 sendSet 和 sendMessenge 都为空。AsyncThread 处理该节点时，不发送消息，直接生成 commitNode，只将 isChangeNode 设为 true 即可。commitNode 的关键数据字段如下表 4.8 所示。

表 4.8 commitNode 的关键字段

名称	类型	说明
tid	ceph_tid_t	MOSDSubOp 中封装的事务编号
commitSet	set<int>	等待确认的 OSD 集合
isChangeNode	bool	是否是 ChangeNode 的标记

其中 tid 是 MOSDSubOp 中封装的事务的编号，Replica OSD 发送回复消息的时候也会附带该编号，根据该编号可知是对哪个事务的确认消息。commitSet 是等待接收确认消息的 OSD 的集合。isChangeNode 是 ChangeNode 的标记，为 true 则该节点是 ChangeNode。Monitor 端更新 OSD 端的同步写副本数时，OSD 端都要进行副本策略的切换，只有等到当前异步更新模块的所有消息处理完成后，才能进行切换。因此可以在队列中加入一个策略切换的时间点，也就是 ChangeNode。如果 ChangeNode 出现在 AsyncReplica 的 commitList 的头部，则表示上一个副本策略的所有的事务都已经完成，新的同步写副本数应用成功，可以向 Monitor 发送回复消息。主副本异步

更新模块的处理流程如下图 4.6 所示。

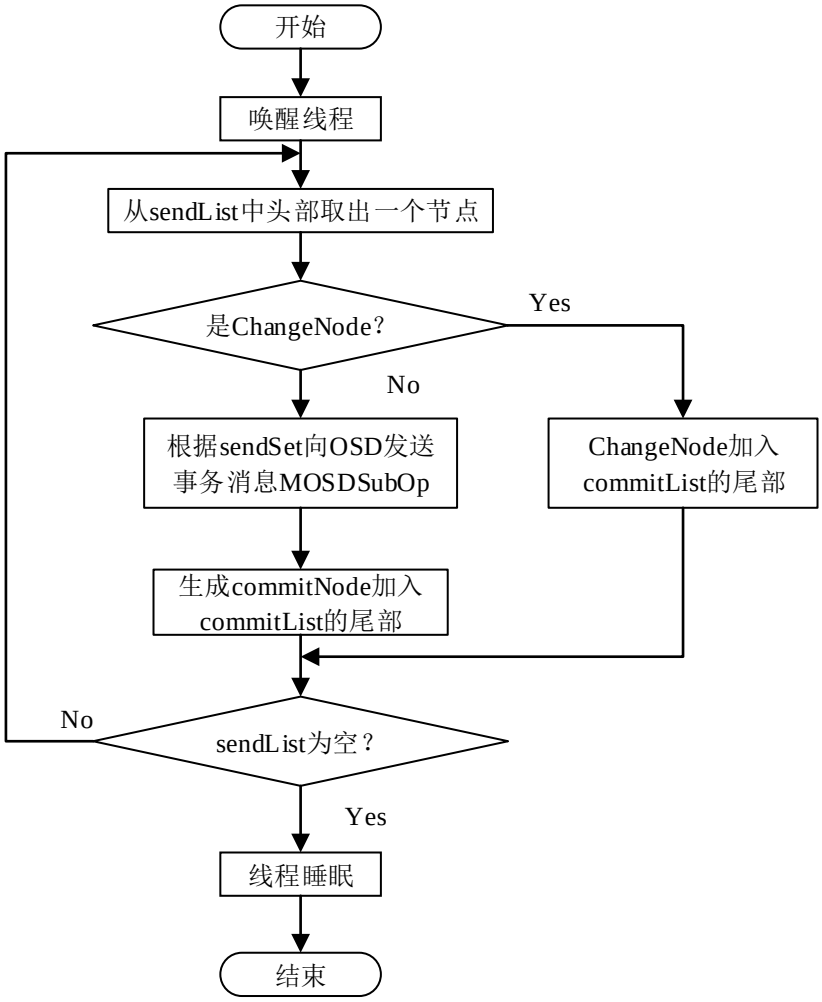

图 4.6 Primary OSD 异步更新 Replica OSD 的流程

Replica OSD 接收到 MOSDSubOp 消息后，会剥离出其中的事务，添加上日志和回调函数后交由 FileStore 模块执行，一旦 FileStore 执行了该事务，则调用回调函数给 Primary OSD 发送回复消息。Primary OSD 在收到回复消息后会先在同步更新的等待队列 in_progress_ops 中查找，一旦找到则从节点的等待队列中删除该 OSD 编号，如果队列为空则向客户端返回回复消息，写操作成功。如果同步更新队列中没有则交由异步更新模块处理，异步更新模块从 commitList 中查找该节点，找到后则从该节点的等待集合中删除该 OSD 编号，一旦该集合为空则可以删除该节点，表示该事务已经成功异步更新到各个 OSD。此时如果 ChangeNode 在 commitList 的头部则表示

可以向 Monitor 发送回复消息 MOSDSyncWriteReply。Primary OSD 处理该回复消息的流程如图 4.7 所示。

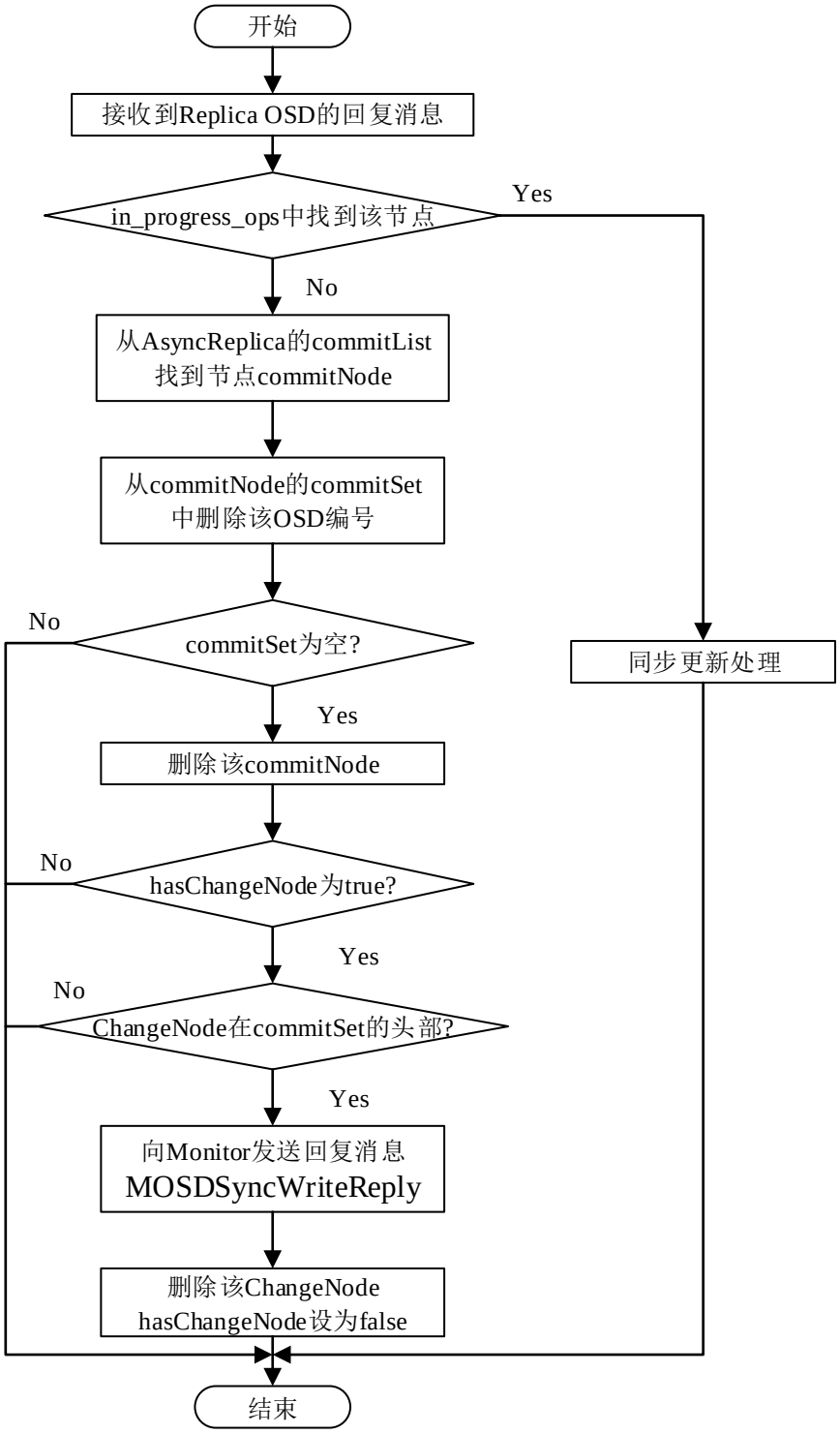

图 4.7 Primary OSD 处理 Replica OSD 的回复

4.2.2 OSD 与 Monitor 端消息处理

Monitor 端的定时器会定时根据统计的读写操作数计算出副本策略，然后将新计算的同步写副本数发送给 OSD 端，OSD 端要进行副本策略的切换，切换成功后还要向 Monitor 端发送回复消息。Monitor 端发送来的消息为 MOSDSyncWrite，该消息中封装了同步写副本数，OSD 端收到该消息后更新本地的 osdSyncWrite，然后向异步更新模块的 sendList 的末尾插入 ChangeNode，当该节点移动到 commitNode 的头部时可以向 Monitor 端发送回复消息 MOSDSyncWriteReply，该消息中封装了该 OSD 的编号，表明该 OSD 节点副本策略切换完成，该流程如图 4.8 所示。

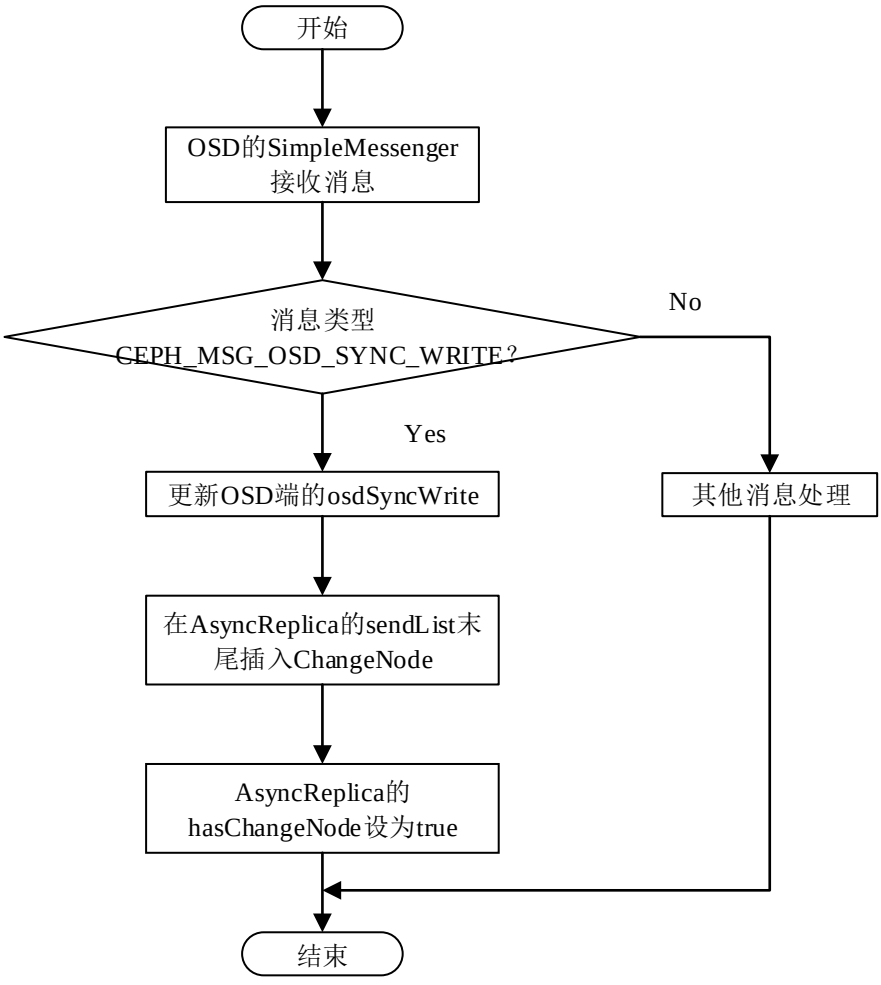

图 4.8 OSD 处理 Monitor 发送的 MOSDSyncWrite

OSD 端不仅要处理 Monitor 发送过来的 MOSDSyncWrite 消息，还要发送

MOSDSyncWriteReply，而回复消息的发送的流程在图 4.7 中已经描述，关键点在于节点 ChangeNode。每次切换副本策略都需要等待上一个副本策略的消息全部处理完成才算切换成功，因此在两种副本策略之间加了一个标记 ChangeNode，当该标记之前的消息全部处理完成（即 ChangeNode 到达 commitList 的头部）则表明上一个副本策略的消息全部处理完成，可以向 Monitor 发送回复消息。

4.3 Client 端的副本策略实现

对于新的副本策略，客户端的修改不大，主要在于读操作的修改。客户端的写操作是发往 Primary OSD 的，由 Primary OSD 负责 Replica OSD 的写操作。而客户端的读操作使用的是 random 方法，随机发往存放数据的任意一个副本。新的副本策略存在异步更新的副本，为了防止客户端读到异步更新的 OSD，因而要缩减 random 的范围，只能 random 同步更新的副本。Client 要知道哪些副本是同步更新的哪些是异步更新的就必须与 Monitor 进行通信。Monitor 端是无法知道整个集群中每个客户端地址的，因此无法主动向 Client 发送同步写副本数，只能是 Client 端主动联系 Monitor，获取 clientSyncWrite 的值。Client 和 Monitor 通信的消息有两个，分别是 MMonGetVersion 和 MMonGetVersionReply。Client 进行读写操作时都要用到 CRUSH 算法进行数据的定位，而算法中要用到 CRUSH Map（OSDMap），OSDMap 的维护是由 Monitor 进行的，客户端为了确认本地的 OSDMap 是最新的在进行读写操作前都要发送 MMonGetVersion 消息，为了 Monitor 端能够进行读写操作数的统计，MMonGetVersion 消息中封装了读写操作标记。当 Monitor 收到消息后会发送回复消息 MMonGetVersionReply，该消息中封装了 OSDMap 的版本号和 clientSyncWrite，通过它客户端更新本地的 clientSyncWrite，从而在读操作时访问同步更新的副本。

所有上层应用客户端的操作都是基于 Librados 的接口，只要修改在 Librados 读写操作即可。客户端的读写操作最后都封装成了 op，然后调用库接口提交 op，提交 op 的时候会向 Monitor 发送 MMonGetVersion 消息，发送消息的时候要添加上读写标记，如果是其他模块发的 MMonGetVersion 消息使用默认标记，最后将 op 封装成

MOSDOp 发往目标 OSD。客户端发送消息的流程如图 4.9 所示。

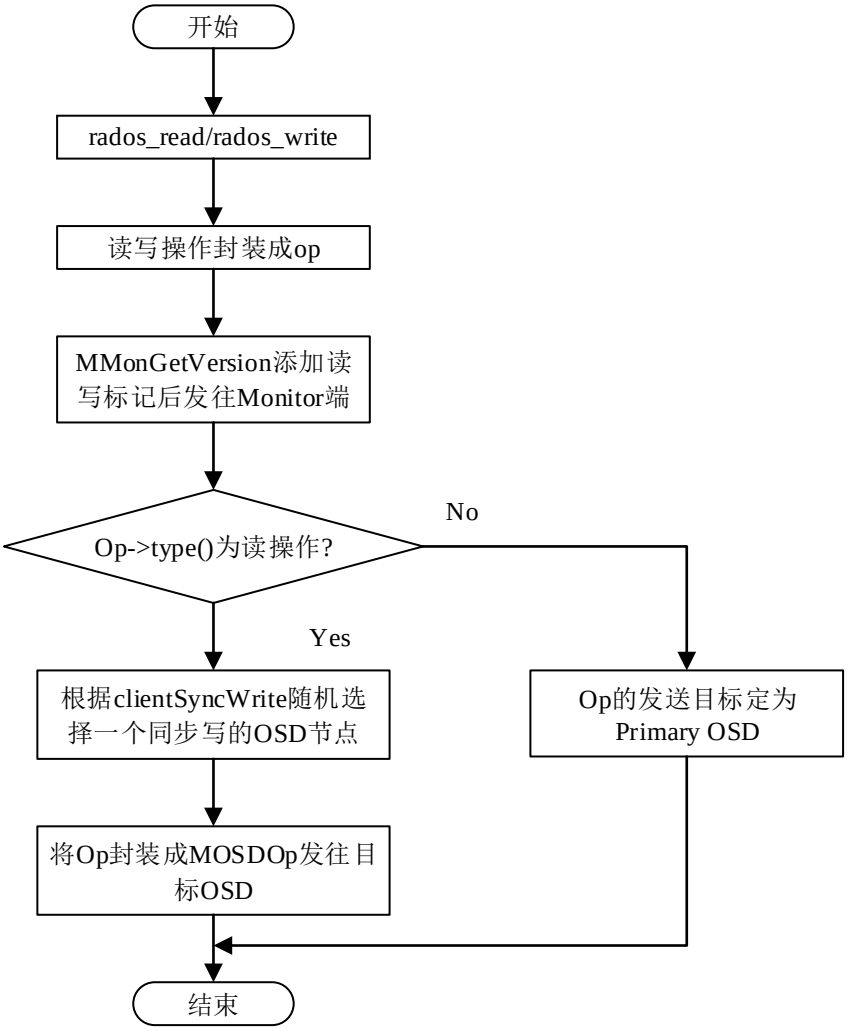

图 4.9 Client 读写操作流程

4.4 本章小结

本章描述了 Ceph 动态副本一致性策略的实现，并对 Monitor、OSD 和 Client 端的实现流程进行了详细介绍，包括 Monitor 的动态策略、OSD 的异步更新模块和 Client 的修改等，给出了具体数据结构和数据操作流程。

5 测试及分析

Ceph 是基于 Linux 平台的，使用 vdbench 对修改后的系统进行了性能测试[40]，主要测试了异步更新副本模块和动态副本策略的读写延时和读写带宽，并与原系统进行了对比。

5.1 测试环境

在五台服务器上搭建了整个 Ceph 分布式存储系统，包括了三台 OSD 节点，一台 MDS 节点和一台 Monitor 节点，整个平台的部署如图 5.1 所示。

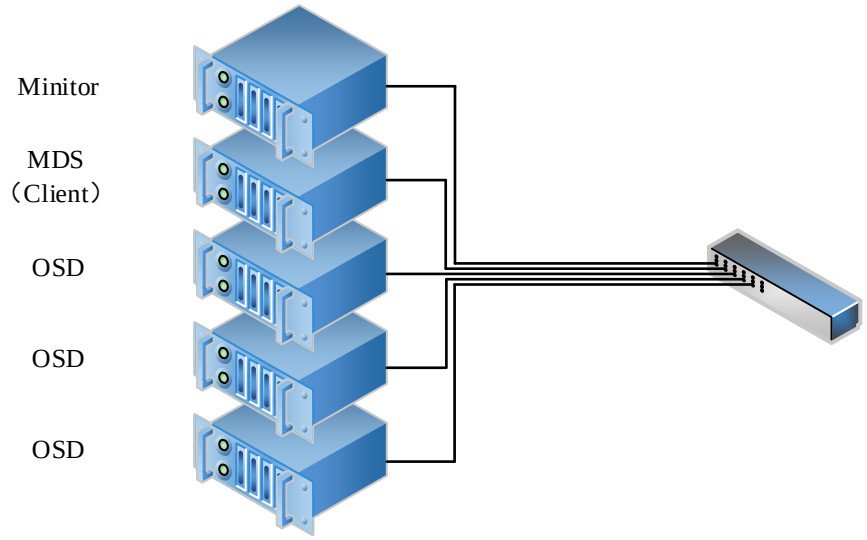

图 5.1 Ceph 系统部署

整个测试平台中使用 Ceph 的 RBD 功能进行测试，即使用 Ceph 的块存储功能，因此 MDS 在测试并没有被使用，而是被当做客户端来使用。Ceph 的 MDS 服务器只有在使用分布式文件系统时才会需要，而本次测试主要测试 Ceph 的 OSD 端的副本策略，并没有对 MDS 进行修改，因此选择了 Ceph 的 RBD 功能进行测试。测试时，先将 3 个 OSD，一个 MDS 和一个 Monitor 启动，直到整个 Ceph 集群的状态变为

HEALTH_OK，则 Ceph 集群启动成功。然后，通过 MDS 节点在 RBD 的存储池上创建了一个 500G 的块设备，并导出挂载在该节点上。整个测试过程主要是对该块设备的读写性能进行测试。

Ceph 集群中服务器的硬件配置如表 5.1 所示，在 OSD 节点上有数据盘和日志盘，元数据服务器 MDS 和集群监视器 Monitor 并不需要这两个盘。

表 5.1 服务器的硬件配置

名称	参数
CPU	Intel(R) Xeon(R) CPU X5650 @ 2.67GHz ×2
内存	DDR3 1333MHz 4G×2
系统盘	西部数据 1TB 7200rpm
数据盘	西部数据 1TB 7200rpm
日志盘	希捷 1TB 7200rpm
网卡	Intel Corporation 82574L Gigabit Network
操作系统	Ubuntu 14.04 x86_64
内核版本	Linux 3.13.0-24

测试时，使用 ssh 远程控制这些服务器，在 MDS 上使用 vdbench 对导出的块设备进行测试，控制端的软硬件配置如表 5.2 所示。

表 5.2 远程控制端的配置

名称	参数
操作系统	Windows 10 专业版 64 位
CPU	英特尔 i3 3.10GHz
内存	1333MHz 4GB
硬盘	西部数据 1TB 7200 rpm
网卡	瑞昱 RTL8168E PCI-E Gigabit Ethernet NIC

5.2 性能测试与分析

主要测试 Ceph 的块存储性能，基于异步更新副本模块和动态副本策略，对 Ceph 中的一个 500G 的块设备进行读写性能测试。主要测试该块设备的读写带宽和读写延时，并与原系统进行了对比。

5.2.1 异步更新副本模块测试

测试 OSD 新添加的异步更新模块时，关闭了动态副本策略。Ceph 默认使用的是三副本策略，并且是三个副本同步更新的强一致性策略。通过异步更新模块修改 Ceph 的副本策略，分别测试了两副本同步更新（剩余一个副本异步更新），一个副本同步更新（剩余两个副本异步更新），原副本策略（三个副本同步更新）时的读写性能，并与原系统进行了对比。

本次测试使用 vdbench 分别测试了 Ceph 块设备的 128KB、1MB 和 4MB 的随机读写性能以及顺序读写性能。vdbench 的配置参数如表 5.3 所示。

表 5.3 vdbench 的配置说明

名称	说明
anchor	测试路径
width	创建的目录数
depth	创建的目录深度
files	创建的文件数
sizes	创建的文件大小
rdpct	读写操作的比例

在 vdbench 的测试文件中配置 read_write 参数，本次测试不需要测试元数据的性能，不需要创建大量的目录文件，因此测试时在 Ceph 的 RBD 块设备的挂载路径的根目录下下创建了 3000 个文件，主要测试文件的随机读（random_read），随机写（random_write），顺序读（seq_read），顺序写（seq_write）的 IO 带宽和延时，并且

针对文件大小为 128KB，1MB，4MB 的三种负载分别进行了测试。

首先测试了异步更新模块的读操作延时和带宽，如图 5.2 和 5.3 所示。

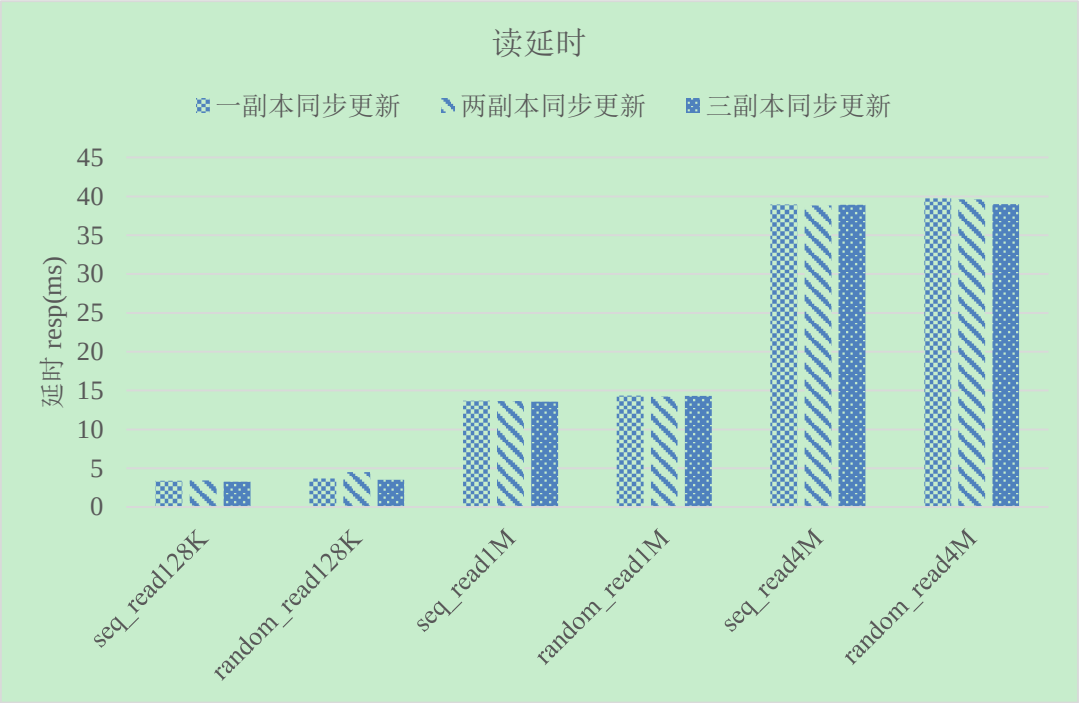

图 5.2 客户端读操作延时

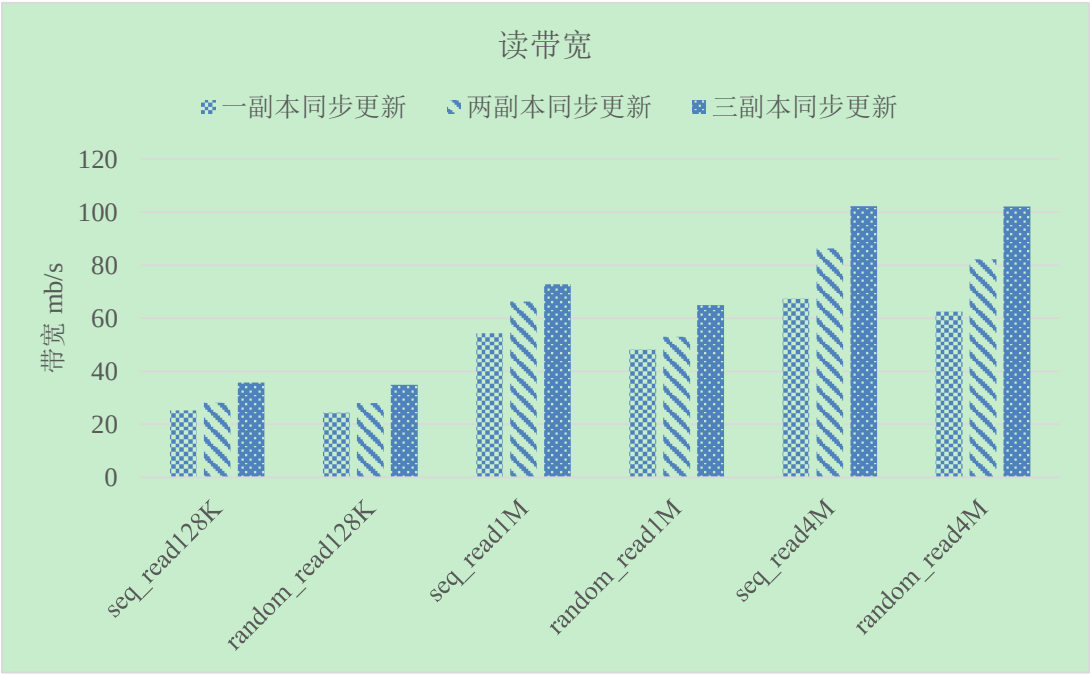

图 5.3 客户端读操作带宽

通过图 5.2 的数据可以看到，异步更新对客户端的读操作延时影响不大，随着文件的增大，读操作的延时会增大。客户端进行读操作时会随机选择一个同步更新的副本节点进行读操作，读操作不需要读多个副本，因此不论同步更新副本数怎样变化，读操作都能找到一个副本进行读取，读操作延时变化不大。

通过图 5.3 的数据可以看到，异步更新会影响客户端的读带宽，异步更新的副本越多，客户端的读带宽越低，并且随着文件的增大，这种降低也越明显。在文件大小为 128KB，1MB 和 4MB 时，两副本同步更新时，读带宽平均降低了 14%左右，而一副本同步更新时，读带宽降低了 25%左右。使用异步更新副本会导致客户端读带宽的下降，这是因为客户端进行读操作时会随机选择同步更新的 OSD 节点进行读取，同步更新的副本数越多，可以读取的 OSD 节点越多，因此带宽越高。如果一个文件被同步更新了三副本，则可以将文件分成三个部分通过三个 OSD 节点读取这个文件。如果只有一个同步更新的副本则只能通过一个 OSD 节点读取。

接着测试了异步更新模块的写操作延时和带宽，如图 5.4 和 5.5 所示。

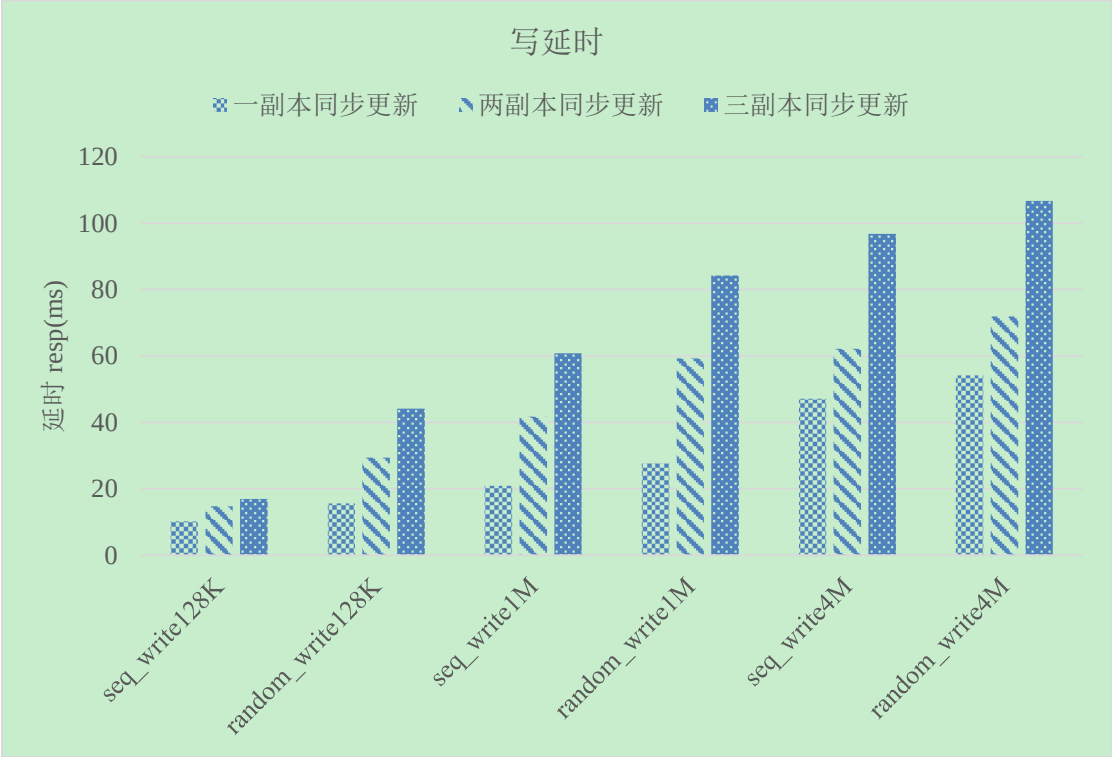

图 5.4 客户端写操作延时

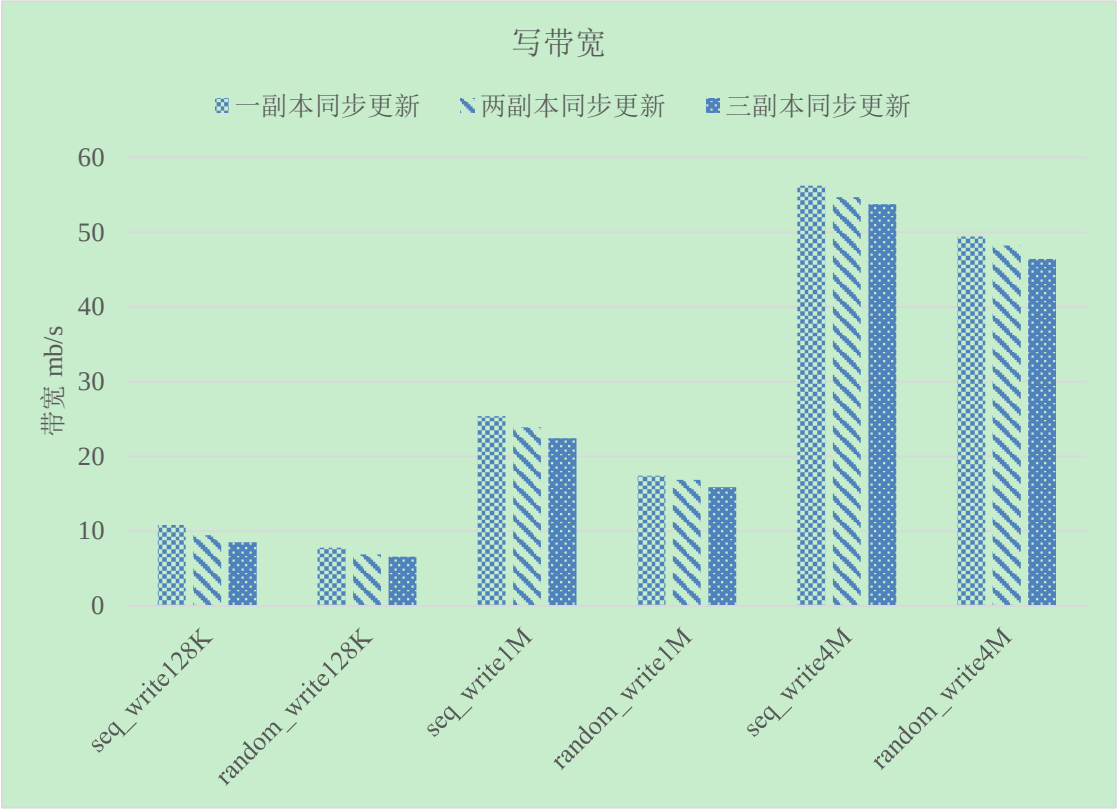

图 5.5 客户端写操作带宽

通过图 5.4 的数据可以看到，通过 OSD 端的异步更新模块，可以减少同步更新的副本，从而大大降低写操作的延时。并且随着文件的增大，这种降低更加明显，因为同步更新各个副本是通过网络进行的，文件越大，通过网络传输的延时也越大，因而使用异步更新带来的效果也就越明显。系统若是经常进行大文件的读写，可以使用异步更新模块进行。

通过图 5.5 的数据可以直观的了解到，异步更新模块对写操作带宽的提高并不明显，不论是异步更新还是同步更新，都需要更新相同的副本数。同步更新副本的时候，需要通过主副本将写操作同步更新到其他的两个副本，而异步更新的时候，也需要通过主副本异步更新其他的副本，网络中的总的流量并没有太大的变化，因此对写操作的带宽并没有太大的提高。

通过对异步更新副本模块的测试可以了解到，异步更新模块可以提高写操作的延时，略微提高了带宽，但是降低了读操作的带宽，对读操作的延时影响不大。因此，

副本一致性的问题实际上就是读写之间的矛盾，一致性越强，读带宽越高，但是写延时也越高，当强到一定程度时，客户端甚至无法写入数据。一致性越弱，写延时越低，但读带宽越低，甚至客户端可能读到旧的数据。

5.2.2 动态副本策略测试

当系统处于读密集型时，动态副本策略使用的是强一致性，三个副本同步更新，当系统处于读写疏松型时，副本策略不变，只有当系统处于写密集型和读写密集型时，才会根据读写的比例来动态的调整副本策略，即调整同步更新的副本数。在 vdbench 中可以配置读写的比例，从而让系统处于不同的状态。本次主要测试在读写密集型状态下，读写操作的带宽和延时。先在一个阈值时间段中使用该比例的读写操作使系统达到相应的测试状态，然后在下一个阈值时间段中进行测试。

本次测试主要测试了在读写密集型，不同读写比例时，顺序读（seq_read），顺序写（seq_write）1MB 大小文件的 IO 带宽和延时。测试结果如图 5.6 和 5.7 所示。

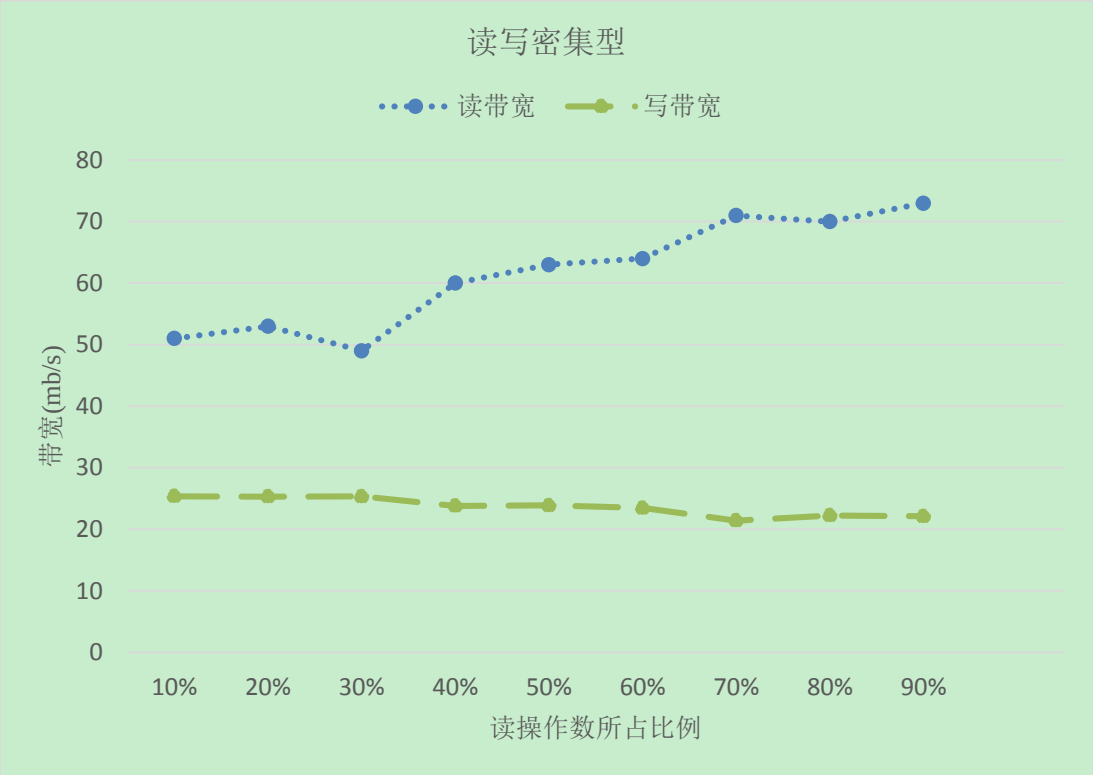

图 5.6 不同比例下读写带宽

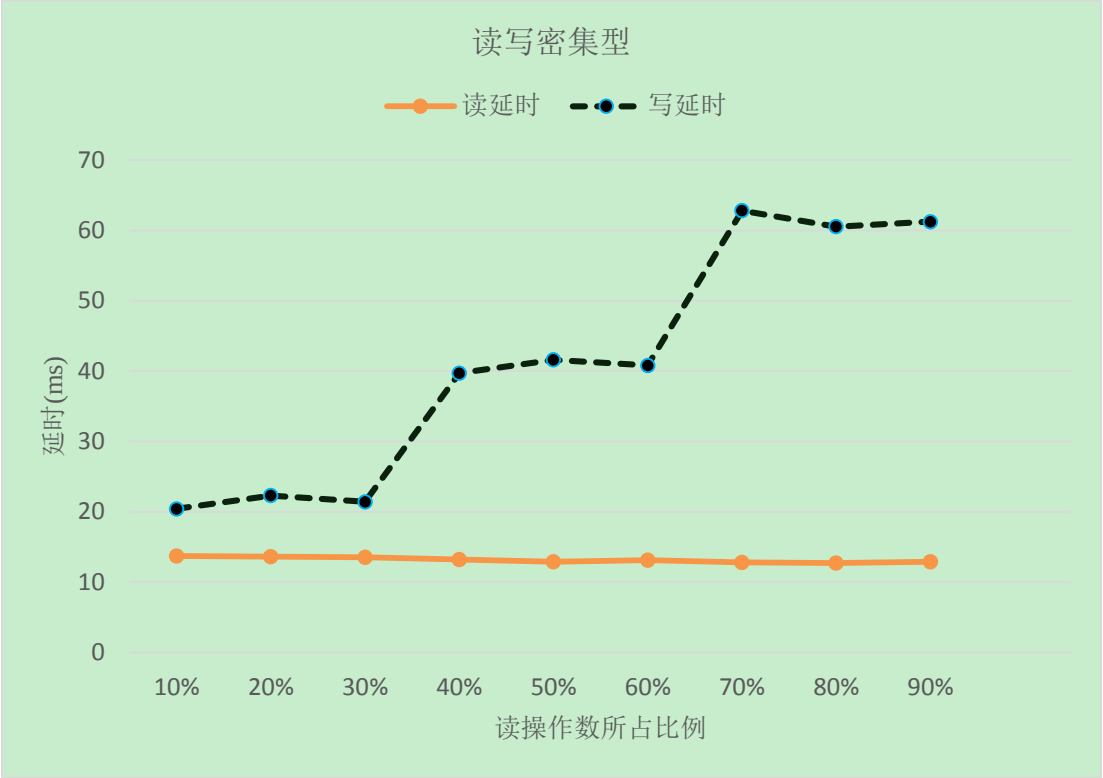

图 5.7 不同比例下读写延时

通过图 5.6 和 5.7 的数据可以了解到，该动态副本策略可以根据读写操作数所占的比例来动态的调整同步更新的副本数，从而提高不同状态下的用户体验。当读操作数所占的比例越高，即写操作数的比例越低，则读带宽会增大，写延时会提高。反之，当读操作数所占比例越低时，即写操作所占比例越高，则写延时会降低，读带宽也降低。在图中，当读操作数比例小于 30%时，系统中写操作较多，因此系统使用了同步写一个副本，异步更新两个副本的策略，写延时较低。当读操作数的比例大于 40%小于 60%时，系统中的读操作数和写操作数相差不大，因此系统使用同步更新两个副本，异步更新一个副本的策略，读带宽提高，写延时也提高。当读操作数的比例大于 70%时，系统中读操作较多，因此使用了同步更新三副本的策略，读带宽提到最高，写延时也达到最高。

通过以上测试可以表明，该动态副本策略可以根据最近系统中读写的比例来调整同步和异步更新的副本数，从而很好的平衡读写的代价。这种副本策略改变了 Ceph 使用的单一的强一致性策略，从而能让 Ceph 适应更加复杂更加多样化的云计算环境。

5.3 本章小结

本章使用 vdbench 测试工具分别对 OSD 的异步更新模块和动态副本策略进行了测试。测试结果表明，通过异步更新副本的方式可以大大的降低写操作的延时，但也会降低读操作的带宽，这需要用户根据具体的应用环境自己配置。而动态副本策略可以根据当前系统中读写操作数的比例，动态调整同步和异步更新的副本数，从而平衡读写的代价。

6 总结与展望

6.1 总结

随着云计算和大数据的发展，网络数据快速增长，对底层的存储系统提出了更加严峻的挑战。对于底层的存储，分布式存储系统由于它的大容量和高扩展性而备受关注。就现在开源的云计算平台 OpenStack 而言，它要使用的底层存储系统就涉及到对象存储系统，块存储和文件系统存储。而 Ceph 分布式存储系统却拥有分布式对象存储，块存储和文件系统的统一存储的能力。在分布式的环境中，副本一致的权衡问题不可避免。而在云存储的环境下，应用的多样性对底层的存储又有不同的需求，Ceph 提供的单一的强一致性策略并不能适用这种多样化的环境。强一致性策略对写的要求太高，每次写操作都要写多个副本才算成功，只要有一个副本写失败就算失败，而在一个上千台机器的分布式的环境中，节点的失败是常见的，每个节点的故障都会造成大量的写失败。本文针对 Ceph 的这种强一致性策略进行了优化，优化成一个用户可配置的基于读写比例的动态副本策略，主要工作如下：

(1) 对分布式存储系统的产生环境进行了介绍，并详细介绍了分布式存储系统中的副本技术。对于副本技术中的副本一致性问题，各个不同的分布式存储系统使用了不同的策略，有使用强一致性的主从同步复制，链式副本策略，Paxos 算法等，还有使用最终一致性算法的 NWR 策略等。

(2) 详细介绍了 Ceph 分布式存储系统的相关技术，Ceph 由于它的高可靠性、高扩展性、CRUSH 算法和统一存储等能力而受广大用户的青睐。并详细介绍了副本一致性技术，而传统的强一致性或者弱一致性技术并不能适应现在多样化的应用环境。

(3)详细分析了 Ceph 的副本一致性策略，分析了客户端到 OSD 端的 IO 路径，然后设计了一种基于客户端读写比例的动态副本策略，根据当前的读写操作数动态调整 OSD 端的同步写副本数，而其他副本异步更新，并对 Monitor 端，OSD 端和客户端进行了设计。

(4)从 Monitor 端，OSD 端，Client 端进行了系统的编码实现，并对优化后的系统进行了性能测试和对比，新的副本策略在写性能上有所提升，基本达到了预期的目标。

6.2 展望

本文提出的副本策略中有写参数需要用户自己设定，有定时器的时间参数，还有读写密集型的读写阈值设定。而这些参数的最优值，可能在不同的环境中有不同的值。在将来的应用环境中，如果可以让系统根据实际的使用环境进行自适应的调整，系统的性能将得到更大的提升。

另外，本文实现的是基于读写比例进行动态调整的副本策略，而在实际的环境中，不同的客户端可能拥有不同的优先级或者权重[41]，仅仅针对读写比例进行调整可能并不准确，在将来实际的应用环境中可以根据用户的重要性设定不同的优先级，不仅仅只针对读写，还可以考虑一些其他的因素进行权衡，这样系统将适应更加复杂的多样化的使用环境。

致谢

浮云一别后，流水十年间，在毕业论文即将完成之际，这不仅仅意味着我两年的研究生生涯即将结束，还意味着六年的华中科技大学的学习生活即将画上句号，更意味着二十多年学生时代的终点。毕业不仅仅意味着离别，也有对未来的憧憬和期盼，时光如流水，我们一直在追寻我们想要的，却也一直在失去我们最重要的，失去不要紧，怕的是忘记，时间会抹去一切痕迹，但我永远也不会忘记在华中科技大学的学习生活，不会忘记那绝望坡，不会忘记那梧桐雨，还有那眼镜湖，更不会忘记那可敬的老师们和可爱的小伙伴们。他们不仅仅教会了我学习的技能，更是传递了他们对人生的思考以及人生态度，让我对人生，对社会，对国家乃至对这个世界都有了更加深刻的理解。在此我要向他们表示最诚挚的感谢。

首先，我要感谢我的导师万继光老师。从论文的选题到论文的设计和实现，导师都给了十分宝贵的意见，让我少走了很多弯路。还要感谢万老师带我进入了存储的大门，让我学到了很多存储相关的知识。在研究生期间，老师提供了相当好的学习环境，极大锻炼了我的编程能力，并开阔了学习思维。不论是学习还是生活，万老师都会与我们进行交流，他不仅仅是我们的学习导师，更是我们的生活导师。在此向老师表示由衷的感谢。

然后我要感谢哪些一起做项目的师兄们，他们在学习和生活中都给了我极大的照顾。当项目遇到难题时，师兄们的宝贵意见往往都能给我极大的启发，他们更会推荐一些不错的专业书籍以提高我的专业能力。感谢李元超、刘洋、陈诗杨、周晨曦等学长们在项目期间的帮助，还要感谢 Ceph 项目组的葛凯凯、李大平、张和泉等师兄弟们，感谢他们在我学习 Ceph 和设计论文期间给以的帮助和建议。

最后，感谢父母的关心和支持，父母的恩情我会永远铭记在心。

参考文献

[1] Sage A. Weil. Ceph: Reliable, Scalable, and High-Performance Distributed Storage. Ph.D. thesis, University of California, Santa Cruz, December, 2007.20~40

[2] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, Carlos Maltzahn. RADOS: A Fast, Scalable, and Reliable Storage Service for Petabyte-scale Storage Clusters. Petascale Data Storage Workshop SC07, November, 2007.3~7

[3] PA Lisiecki, C Nicolaou, KR Rose. Scalable, High Performance And Highly Available Distributed Storage System For Internet Content. Akamai Technologies,2013.59~62

[4] JD Bright, JA Chandy. Data Integrity in a Distributed Storage System. International Conference on Parallel & Distributed Processing Techniques & Applications, 2003:688~694

[5] J Maccormick, CA Thekkath, M Jager. Niobe: A Practical Replication Protocol. ACM Transactions on Storage, 2008.5~9

[6] T Lau, L Bergman, V Castelli, D Oblinger. Sheepdog: Learning Procedures for Technical Support. Proceedings of Iui, 2004:109~116

[7] Denning P. The Working Set Model for Program Behavior. Communications of the ACM, 1968, 11(5): 323~333

[8] HM Chen, MH Chang, PC Chang, MC Tien, WH Hsu. SheepDog: group and tag recommendation for flickr photos by automatic search-based learning. International Conference on Multimedia, 2008:737~740

[9] H Garcia, A Ludu. The Google file system. Acm Sigops Operating Systems Review, 2003, 37(5):29~43

[10] S Ghemawat, H Gobioff, S Leung. File and storage systems: The Google File System. Acm Symposium on Operating Systems Principles Bolton Landing, 2003, 37:29~43

[11] G Decandia, D Hastorun, M Jampani, G Kakulapati, A Lakshman. Dynamo: amazon's

highly available key-value store. Acm Sigops Operating Systems Review, 2007, 41(6):205~220

[12] V Bala, E Duesterwald, S Banerjia. Dynamo: a transparent dynamic optimization system. Acm Sigplan Notices, 2000, 35(5):1~12

[13] E Gafni, L Lamport. Disk Paxos. Distributed Computing, 2003, 16(1):1~20

[14] 倪超著. 从 Paxos 到 ZooKeeper:分布式一致性原理与实践. 北京 : 电子工业出版社, 2015

[15] L Lamport, D Malkhi, L Zhou. Vertical paxos and primary-backup replication. In The ACM Symposium on Principles of Distributed Computing, 2009:312~313

[16] D Ongaro, J Ousterhout. In search of an understandable consensus algorithm. Draft of October, 2013:12~13

[17] J Baker, C Bond, J Corbett, JJ Furman. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. Conference on Innovative Database Research, 2011

[18] 杨传辉著. 大规模分布式存储系统: 原理解析与架构实战. 北京 : 机械工业出版社, 2014

[19] 金海. 漫谈云计算. 中国计算机学会通讯, 2009

[20] M Chalkiadaki, K Magoutis. Managing Service Performance in the Cassandra Distributed Storage System. IEEE International Conference on Cloud Computing Technology & Science, 2013, 1:64~71

[21] JL Zhang, C Zhou, J Wan, L Zhou, YY Yin. OBSC: A Server-side Cache Scheme for Object-Based Distributed Storage System. Advances in Information Sciences & Service Sciences, 2012, 4(11):217~224

[22] X Gao, Y Ma, M Pierce, M Lowe, G Fox. Building a Distributed Block Storage System for Cloud Infrastructure. IEEE Second International Conference on Cloud Computing Technology & Science, 2010:312~318

[23] XF Liao, H Li, H Jin, HX Hou, Y Jiang. VMStore: Distributed storage system for multiple virtual machines. Sciece China Information Sciences, 2011, 54(6):104~118

[24] A Corradi, M Fanelli, L Foschini. VM consolidation: A real case based on OpenStack Cloud. Future Generation Computer Systems, 2014, 32(1):118~127

[25] RS Transfer. Representational State Transfer. C for Programmers Deitel Developer, 2009

[26] D Gudu, M Hardt, A Streit. Evaluating the Performance and Scalability of the Ceph Distributed Storage System. IEEE International Conference on Big Data, 2015:177~182

[27] LB Costa, M Ripeanu. Towards automating the configuration of a distributed storage system. IEEE/ACM International Conference on Grid Computing, 2010:201~208

[28] P Koopman, J Devale. The Exception Handling Effectiveness of POSIX Operating Systems. Software Engineering IEEE Transactions on, 2000, 26(9):837~848

[29] C Clark, K Fraser, S Hand, JG Hansen. Live migration of virtual machines. Conference on Symposium on Networked Systems Design & Implementation-volume, 2005, 2:273~286

[30] J Lamping, E Veach . A Fast, Minimal Memory, Consistent Hash Algorithm. Eprint Arxiv, 2014

[31] Q Naquid, M Jimenez, R Guerrero, Miguel. Fault-Tolerance and Load-Balance Tradeoff in a Distributed Storage System. Computacion Y Sistemas, 2010, 14(2):151~163

[32] B Trushkowsky, K Peter, A Fox, MJ Franklin. The SCADS director: scaling a distributed storage system under stringent performance requirements. Usenix Conference on File & Storage Technologies, 2011:163~176

[33] P Sobe. Data consistent up-and downstreaming in a distributed storage system. International Workshop on Storage Network Architecture, 2003:19~26

[34] C Ferdean, M Makpangou. A Generic and Flexible Model for Replica Consistency Management. Lecture Notes in Computer Science, 2004, 3347:204~209

[35] X Ouyang, T Yoshiharay, H Yokota. An Efficient Commit Protocol Exploiting Primary-Backup Placement in a Distributed Storage System. Pacific Rim International Symposium on Dependable Computing, 2006:238~247

[36] S Alkiswany. Embracing diversity : optimizing distributed storage systems for diverse deployment environments. Ph.D. thesis, University of British Columbia, 2013

[37] JA Chandy. A generalized replica placement strategy to optimize latency in a wide area distributed storage system. A generalized replica placement strategy to optimize latency in a wide area distributed storage system, 2008:49~54

[38] E Pacitti, P Minet, E Simon. Replica Consistency in Lazy Master Replicated Databases. Distributed & Parallel Databases, 2001, 9(3):237~267

[39] Y Tanimura, K Hidetaka, T Kudoh, I Kojima, Y Tanaka. A distributed storage system allowing application users to reserve I/O performance in advance for achieving SLA. IEEE/ACM International Conference on Grid Computing, 2010:193~200

[40] 田怡萌, 李小勇, 刘海涛. 分布式文件系统副本一致性检测研究. 计算机研究与发展, 2012: 276~280

[41] M Radi, A Mamat, MM Deris, H Ibrahim, S Shamala. Access weight replica consistency protocol for large scale data grid. Sciencepublications, 2008:4(2)