分 类 号 _____ 学号 M201172387

学校代码 10487 密级 _____

华中科技大学

硕士学位论文

分布式文件系统副本管理机制研究

学位申请人： 邓纪旭

学 科 专 业： 计算机系统结构

指 导 教 师： 王芳 教授

答 辩 日 期： 2014-1-21

A Thesis Submitted in Partial Fulfillment of the Requirements

For the Degree of Master of Engineering

Research on Replica Management in Distributed File System

Candidate : Deng Jixu

Major : Computer Architecture

Supervisor : Prof. Wang Fang

Huazhong University of Science and Technology

Wuhan, Hubei 430074, P. R. China

Jan., 2014

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

摘要

近年来随着信息化程度的不断提高，人们对计算存储资源的需求呈现出高速增长的趋势。而本地文件系统由于其性能、容量、可扩展性等诸多方面的限制，已经无法满足人们日益增长的存储需求，在此背景下，分布式文件系统应运而生。

在大规模分布式文件系统中，数据的安全性和可靠性是非常重要的。为了保证分布式文件系统能够迅速和有效地恢复损坏的数据，一种可靠的解决方法就是在集群环境中存储服务设备的不同节点上保存数据的副本。一般而言，副本数越多数据的可靠性越高，然而过多的副本对存储资源是极大的浪费而且对网络带宽的开销也非常大。为了维护系统多个副本之间的一致性，在修改副本以及恢复副本时，必须要更新所有的副本，这将产生很大的带宽开销。不仅如此，由于存放文件的节点分布范围比较广，读取以及修改数据的延迟会很大，若副本的放置不合理将会严重的影响 I/O 效率，降低系统性能。对于副本中这些可靠性、高效性、一致性、负载均衡等问题，以往的副本机制大都只能顾及到其中的某一个方面。

通过对特定文件系统布局方式的分析，设计了基于对象和一致性哈希的副本管理方法，在提高系统可靠性的同时保证了系统的整体性能，实现了节点间的负载均衡以及副本恢复的一致性。对于副本布局，在对象级别采用一致性哈希的方法。通过将文件分条，不同条带单元构成对象，以对象副本的方式分布到不同的存储节点，从而提高副本访问的并行性，维护了系统性能。一致性哈希的算法使得副本均匀的散列在存储节点上，保证了节点间的负载均衡。针对数据恢复，在分布式文件系统 Cappella 上进行了实现和优化，使得系统在存储节点失效时能够降级读写，通过多线程流水线的方式加快对象副本恢复速率，提高系统的可靠性。

对比测试表明 Cappella 的副本布局策略写操作性能得到保障，节点间的负载均衡性得到了有效的保证。在有节点失效的情况下可以正常的读写，数据恢复的性能也得到了一定的提升。

关键字：分布式文件系统，副本布局，一致性哈希，可靠性

Abstract

In recent years, with the degree of information technology continues to improve, the demand for computing and storage resources showing rapid growth trend. Due to limitations of performance, capacity, scalability and many other aspects, local file system has been unable to meet the growing storage needs. In this background, distributed file system came into being.

In large-scale distributed file systems, security, and reliability of data occupies a very important position. In order to ensure a distributed file system can quickly and efficiently recover damaged data, an effective solution is to save replica of data on multiple nodes of the cluster servers. In general, the higher the number of replica, the more reliable the system is. However, too many copies of data are an enormous waste of storage resources and network bandwidth cost is also very large. In order to maintain consistency between multiple replica in the system, when modifying a replica or recovering, we must update all copies, which will generate a lot of bandwidth cost. Moreover, since the nodes storing files is widely distributed, data access latency is very large, if a replica is placed unreasonable it will severely impact access efficiency and reduce system performance. For the problems of reliability, efficiency, consistency, and load balancing in replica management, Conventional mechanism can only take one aspect of them into account.

Replica management in Cappella analyzes the layout of a specific file system and proposed a replica management mechanism based on object and consistent hashing. It improves system reliability while ensuring the overall system performance and achieve load balancing as well as consistency. For replica layout, it uses the object-level grouping and consistent hashing methods. By file striping different stripe unit constitutes a object and distribute to different storage nodes as replica. This way it improves parallelism of accessing and maintains system performance. Consistent hashing method ensures the load balance between nodes. For data recovery, we made a implementation and optimization on a distributed file system named Cappella. It allows the system to downgrading read and write when there is node failure. Through a multi-threaded and pipeline approach it accelerates recovery rate, and improves system reliability.

Comparative tests shows that the strategy of replica layout has guaranteed write performance, meanwhile guaranteed load balancing between nodes. In the case of node failure system can read and write normally .The performance of data recovery has also been improved.

Keyword: distributed file system, replica layout, consistent hashing, reliability

目 录

1 绪论

1.1 研究背景

随着大数据[1]时代的到来，作为后端数据存储支撑的文件系统的重要性越来越凸显。本地文件系统因容量、性能等诸多自身限制已无法满足日益增长的存储需求[2]，取而代之的是分布式文件系统[3]的广泛应用与普及。利用分布式文件系统，可以使分散在不同地域的存储节点上的内容就像出于同一个节点一样，统一对外的显示。客户只需要根据自己的需求独立的访问文件内容，而不需要关心这些内容所处的地理位置。

分布式文件系统融合了传统的存储架构 DAS（Direct Attached Storage）和 NAS（Network Attached Storage）[4]的优点，在 I/O 性能、可扩展性、可靠性[5]等方面都有大幅提升。当前主流的分布式文件系统如 Ceph[6]、Lustre[7]、PNFS[8] [9]和 GFS[10]等都采用了客户端、元数据服务器、存储节点这样的三方存储架构。元数据服务器负责文件分发、属性和其他元数据信息的管理；存储节点通过集群的方式存储文件；客户端则实施对文件的具体访问，与元数据服务器和存储节点通过专用高速网络交换数据。本文所在课题组自主研发的基于对象[11]的分布式文件系统 Cappella 就是这样典型的三方架构。它支持 PB 级的存储容量，聚合带宽可达 60GB/s，并且支持数十亿的文件数量，具有高并发性（支持 4000 并发访问）、高吞吐量的特征。

在大规模分布式文件系统中，数据的安全性和可靠性极其重要，它关系到文件系统是否可用，因此提高系统的扩展性和容错性是非常有必要的。本课题研究对象 Cappella 是一个高性能分布式并行文件系统，然而它却缺少这样一个有效的可靠性保障机制。而为了使系统的可靠性得到保障，在集群的多个存储服务节点中保存数据副本[12]不失为一种有效和实用的解决方法。如果其中某个副本遭到损坏或是丢失，则只要使用其他副本代替它就可以正常地进行数据访问。将副本分散保存在不同的节点上，可以有效增加数据的可靠性。

1.2 国内外研究现状

1.2.1 Ceph 系统的副本策略

Ceph[13]的分布式架构是遵循 POSIX 标准的，它可以保障没有单点故障而且数据可以无缝复制，是一个具有一定容错能力的分布式文件系统。Ceph 文件系统的构成与主流分布式文件系统类似，包括一个元数据服务器（MDS）用于管理文件的命名空间、一个存储设备集群用于存储元数据和实际数据、一个遵循 POSIX 语义的客户端接口、以及一个高速互联网络，图 1.1 所示是 Ceph 的生态系统架构。

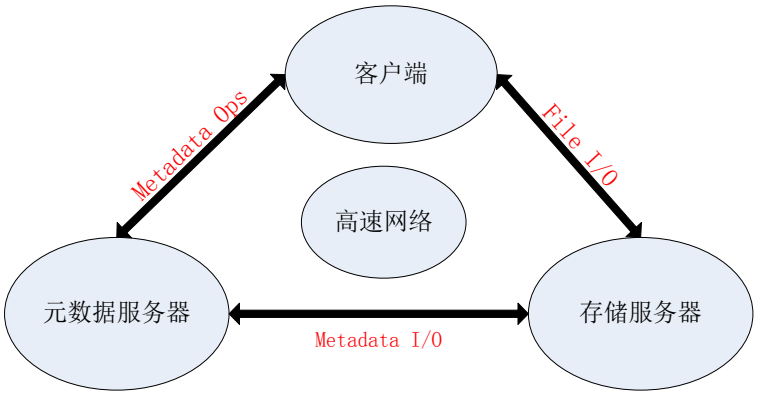

图 1.1 Ceph 系统架构

在数据放置时，它将数据以分片的方式分布到各个对象中，通过一个简单的哈希函数将各对象映射到 Placement Groups（PGs）。其中通过一个位掩码控制 PG 的个数，然后采用一种叫做 CRUSH[14]的算法将各个冗余组分布到一组存储节点上。它将整个存储节点集群视为一个逻辑对象存储，对外统一的提供服务。

客户通过元数据服务器执行元数据操作，元数据服务器负责管理数据存储位置。具体来说它是通过 CRUSH 算法来确定一个放置规则，用于决定每个冗余组的副本个数以及用于存放冗余组的存储节点列表。例如写三副本时，每个冗余组会选出其对应的存储节点列表中的一个存储节点作为主存储节点用于写主副本，而其余的副本则由主存储节点写入冗余组的其它对象设备中。过程如图 1.2 所示。

值得注意的是，元数据的操作是由元数据服务器控制，然而元数据本身是和实际的文件数据一起存放在存储设备储集群中的。这样元数据与数据存放在同一对象存储设备中，其 I/O 带宽会成为瓶颈。

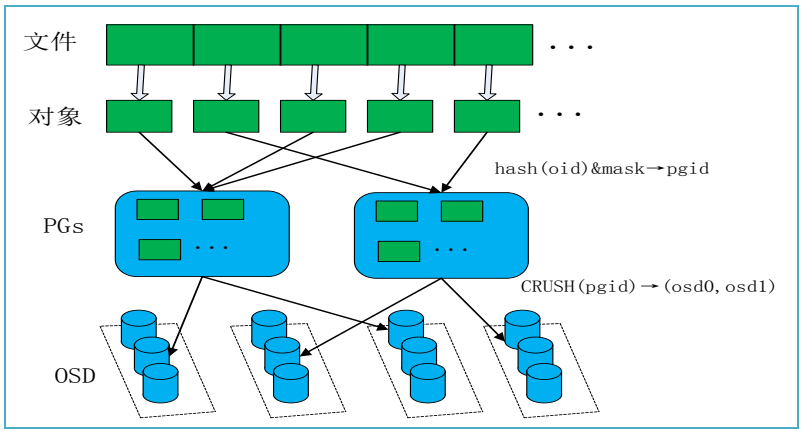

图 1.2 CRUSH 副本布局

Ceph 使用的是 CRUSH 算法将数据分配到存储节点，CRUSH 算法是基于组间分布的，它不支持单个存储服务节点加入或退出系统。

1.2.2 Lustre 文件系统

Lustre 是一个高性能分布式文件系统，它支持的客户端节点达上万个，同时还具有 PB 级的存储容量和 TB 级的聚合带宽。Lustre 对象存储文件系统是由客户端（client）、元数据服务器（MDS）和存储服务器（Object Storage Target）三个主要部分组成。Lustre 通过它的客户端和 OST 存储服务器进行交互来访问文件数据，而对命名空间的操作则是由与 MDS 通信完成的，如图 1.3 所示。

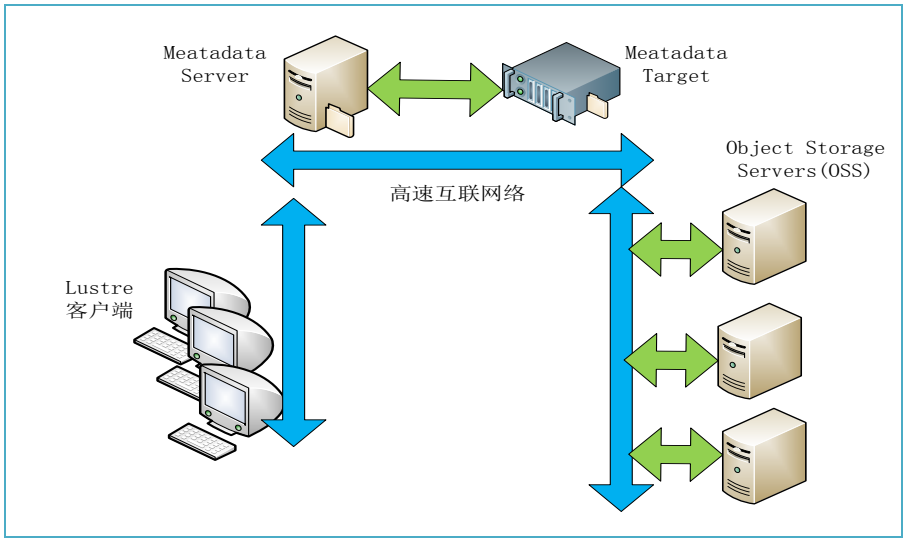

图 1.3 Lustre 组件构成

客户端访问文件时，先同元数据服务器交互获取文件的元数据信息，再与多个对象服务器交互完成数据的并发读写操作。Lustre 适用于高性能计算场景，作为一个

分布式并发集群文件系统，Lustre 专注于高性能计算，目前全球 Top30 的高性能计算机中大多采用了 Lustre 系统作为全局存储系统。但是它缺少数据冗余机制，因而面临严峻的可靠性问题。没有副本机制的 Lustre 依赖于底层设备来保证可靠性，如它的 OST 的存储后端支持 SAN，磁盘阵列，企业级 raid 等多种存储方案，但 Lustre 本身不提供任何提高系统可靠性的机制。

1.2.3 GFS 副本

GFS[15]文件系统是可扩展性强，针对大数据大文件的应用型分布式文件系统。一个 GFS 集群由一个主服务器和多个块服务器组成，并由不同客户端并发访问。它的后端存储采用廉价的硬件设备，因此 GFS 在设计时充分考虑了设备组件失效的情况，为每个数据块提供了若干组备份。它采用的是单元数据服务器模式并且没有提供标准的 POSIX 接口，而是用专用 API 接口取而代之，如图 1.4 所示。

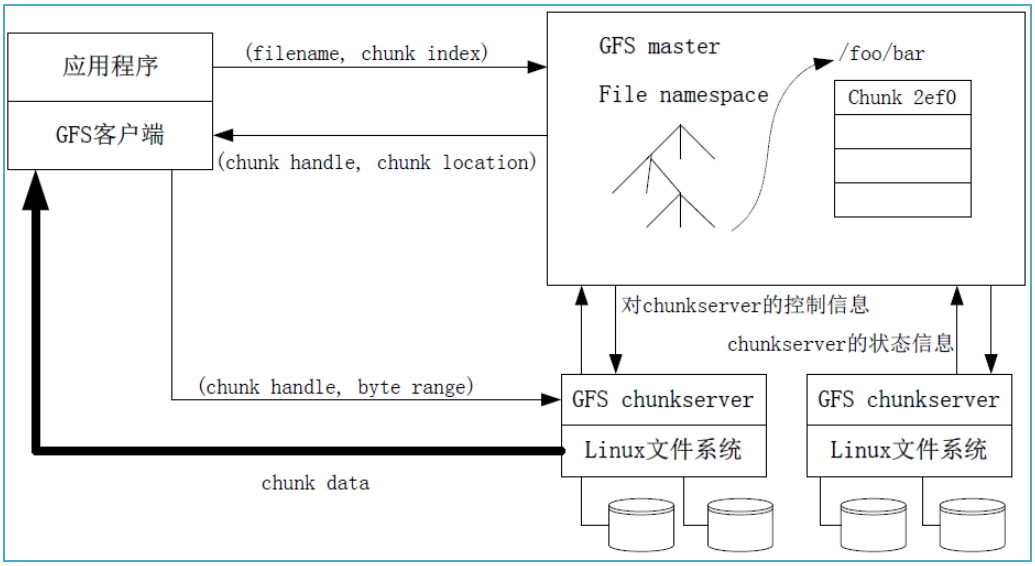

图 1.4 GFS 系统架构

GFS 通过遍历目录树的方法来查找数据所存放的位置，这样便可以在目录中管理存储节点的元数据信息。由于 GFS 管理的文件都是大文件，元数据相对较少，因此这种方式的查找效率较高。但这种大目录存储[16]的数据布局方式却也有它自身的问题：一是它不适合小文件场景。小文件数目较多，海量的元数据信息会导致定位数据对象的开销增大；二是海量的存储目录都是存储在内存储器中以便能快速的访问，但这样会耗费大量的内存资源。

1.3 Cappella 系统平台

Cappella 文件系统是以支持 863 海量存储关键技术重大专项中关于“海量信息组织和安全访问核心系统”的目标而研制的。它采用对象的存储技术，并在此基础上实现了一个基于对象的并行文件系统，为前端用户提供海量文件存储空间。本文在 Cappella 分布式文件系统的基础上研究其副本管理机制，旨在实现其高可靠与高可用性。

Cappella 是个半集中式[17]的分布式文件系统，类似于 Lustre，Google File System，PNFS 等。它由客户端（Client），对象存储服务器（OSD）[18]，元数据服务器（MDS）以及互联网络 InfiniBand（NetWork）[19]组成。

1.3.1客户端

客户端是整个 Cappella 集群中唯一对外提供服务接口的部分。从客户端来看整个 Cappella 集群就是一个文件系统，用户并感觉不到后端集群的工作。客户端软件通过虚拟文件系统 VFS 接口，向用户提供符合 POSIX 标准的访问。图 1.5 是客户端在系统中的交互过程。

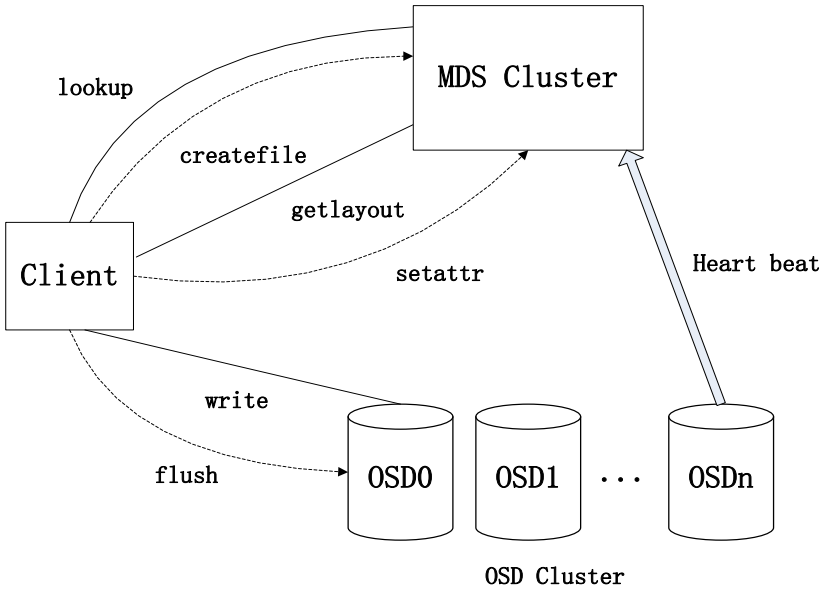

图 1.5 客户端交互流程

整个客户端的运行是从 mount 开始，系统初始化时通过 register_filesystem 函数将 Cappella 文件系统注册到内核，进而将远端 MDS 的输出目录（export path）挂载

到本地。客户端的操作主要包括 mount、lookup、create、write、read、flush、layout 等。打开文件时，通过 lookup 操作向 MDS 查询该文件的元数据信息，而在读写该文件之前则会从 MDS 获取文件的布局策略信息，然后在客户端本地对文件数据的对象布局进行计算。接着，将上层的文件数据请求转化为若干对象请求后写入对应的 OSD 的缓存中并向 OSD 发送 flush 命令，通知 OSD 将缓存中的数据刷入磁盘。最后在客户端在写操作完成之后，客户端将该文件最新的元数据信息提交到 MDS。

1.3.2元数据服务器端

MDS 元数据服务负责实现全局视图与统一命名空间。它完成文件到对象以及对象到设备的映射，并负责文件系统元数据的组织管理与快速检索[20]、元数据服务的热点处理等问题。由于所有文件访问都需首先获取元数据信息，并且可能有大量客户并发访问，因此元数据服务器的计算能力与 I/O 瓶颈是需要考虑的，若能部分计算任务分担给具有计算能力的对象存储节点，则可以有效的缓解 MDS 负载过重的问题。

元数据服务器与客户端交互的关键一点就是为客户端生成并定位一个文件，而实现它的原理就是客户端通过从远端 MDS 获得的 global_id 生成客户端本地的 inode 号。然后通过内核函数 iget5_locked 生成新文件的 inode。另外 MDS 也会与 OSD 之间维持心跳信息，以确保目标 OSD 是处于活动状态，发送心跳信息[21]的频率可以通过在配置文件中设定。若 MDS 在一定时间内没有收到 OSD 的心跳信息则会认为该 OSD 出现故障，那么 MDS 便会更新其维护的 OSD 列表，同时通知和它缓存有同样一份活动 OSD 列表的客户端旧列表作废，并且更新各客户端的 OSD 列表。

1.3.3对象存储端

OSD 对象存储端按照对象接口实现客户端的对象数据读写，完成数据对象的空间管理、组织、检索，支持多个客户端并发地数据读写访问。OSD 端的主要功能是存储对象数据和属性，并对外提供对象访问接口，同时管理本地存储资源和对象行为，具体来说有如下几个：一是提供对象（数据）接口，在标准的块设备上完成对象的本地保存管理（通过文件系统）。二是通过日志记录本地数据变动操作来提供一定的可靠性。三是实现数据对象缓存提高数据访问的性能，通过对象锁确保并发的一致性。四是实现对象元数据的存储和管理。

OSD 的主要处理逻辑如图 1.6 所示

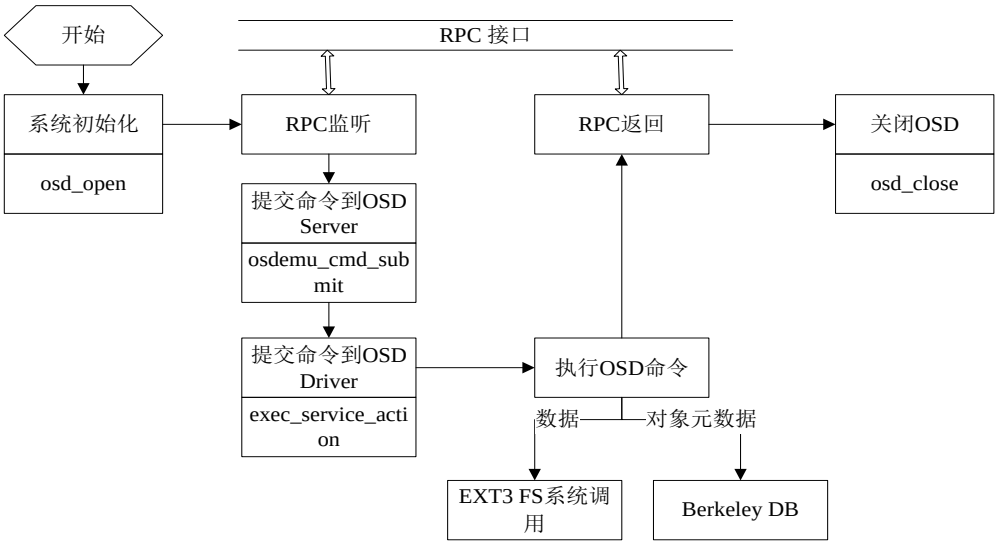

图 1.6 对象存储端总体逻辑

系统启用时需要调用 osd_open 来做系统的一系列初始化工作。初始化工作结束以后，系统准备就绪，RPC 监听网络请求，并将 OSD 命令直接提交到 OSD Server。OSD Server 调用 exec_service_action，解析 OSD 命令，并调用下层接口执行 OSD 命令。命令执行完毕后，通过 RPC 返回执行结果。

1.4 课题目的与意义

副本可以为系统提供可靠性保证，然而副本的加入也会引发新的问题[22]：众多副本如何管理；访问数据时如何选取并定位一个最合适的副本；如何在更新副本或是恢复副本时最大限度的维持系统性能，降低系统开销。本文的目的就是要提供一种有效的副本管理机制，使它在保证系统可靠性的同时能够维护系统的整体性能，降低维护数据一致性的开销，使节点间的负载更加均衡。

针对副本管理中存在的上述问题，本课题提出了一个基于对象[24]和一致性哈希[25]的副本管理机制，在集群服务器的多个节点上保存对象数据副本。通过将文件分条，不同条带单元构成对象，以对象副本的方式分布到不同的存储节点，从而提高副本访问的并行性[26]，维护了系统节点间的性能以及负载均衡。它使系统能够有效地管理副本，在数据失效时可以提供降级模式读写以及均衡迅速的副本恢复，同时保持副本间一致性和系统节点性能之间的平衡，大大提高了 Cappella 文件系统的可靠性和可用性。

1.5 研究内容与论文结构

本文以分布式文件系统 Cappella 为研究对象，设计并实现了以提高其可靠性与可用性为目的的副本管理机制。针对当前文件系统副本放置存在的一些问题，精心设计了一套基于对象与一致性哈希的副本布局策略，有效的维持了系统的整体性能，并使节点间负载变得均衡。对于数据恢复，本文在 Cappella 文件系统上进行了实现和优化。充分利用对象存储服务器的计算能力，将数据恢复任务分散到集群中的所有存储节点上，避免出现单个存储节点上的 I/O 瓶颈。在数据恢复的过程中，通过多线程流水线的方式进一步提高恢复效率。

本文共分为五章，每章的主要内容为：

第一章，介绍了国内外的研究背景和现状，分析了主流文件系统副本管理的优缺点。同时还介绍了本文副本研究工作所基于的分布式系统平台 Cappella，最后阐述了本文研究的目的与意义。

第二章，针对 Cappella 系统平台设计实现了一种副本布局方法，它基于对象和一致性哈希，使副本均衡的分布在各个存储节点上。在读取副本时提高其并行性从而有效保证了系统性能。

第三章，对 Cappella 文件系统的数据恢复进行了实现和优化，解决了数据恢复过程中遇到的问题，对提高恢复速率的方法进行了分析。

第四章，对系统进行测试以及结果分析。

第五章，全文总结。

2 分布式文件系统副本布局机制

分布式文件系统通过副本机制可以大大提高其可靠性，增强其可用性。然而副本的引入势必会增加系统的负担，降低其 I/O 性能和效率，因此副本布局机制的设计就显得尤为重要。

对副本的读写操作必然会增加额外的 I/O 开销，副本放置不合理可能会使得单个存储设备的访问量过大而形成瓶颈，进而影响系统整体性能[27]。同时，冗余副本数据位置信息的计算也可能会导致元数据服务器负载过重而效率降低。大量的元数据信息访问会使得网络负载快速增加，会因网络拥塞而降低 I/O 速率。由此可见，一个好的副本管理机制必须有一个合适的副本布局方案，它能兼顾各存储设备之间的负载，使之相对均衡；它还应该合理分配副本位置计算任务、充分利用系统资源[28]以降低计算和网络传输的时间开销，从而保障系统的整体性能。

2.1 Cappella 副本布局结构分析

2.1.1Cappella 数据布局

本文在 Cappella 分布式文件系统的基础上研究其副本管理机制，旨在实现其高可靠与高可用性。Cappella 是一个系统级文件系统，其数据分配的工作主要由核态客户端计算完成，因此副本布局的主要工作均在内核态实现。

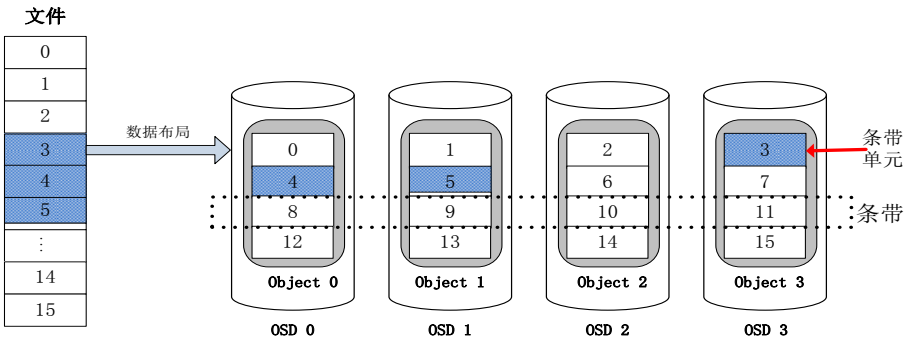

图 2.1 Cappella 数据布局

如图 2.1 所示，Cappella 的设计具有独自的特点：1、对象化管理；2、条带化分布[29]；3、将 OSD 组织成 RAID 形式，并且每个条带是分布在所有 OSD 之上的。文

件数据被分成大小相同的分片，即条带单元（stripe unit），并通过 Round Robin 方式将这些分片分布到相应 OSD 上该文件对应的对象中去。客户端是把所有 OSD 设备以 RAID 的形式组织的，它的 RAID 组织由 osd_layout 的 osd_data_map 描述。文件在每个 OSD 上同一行上的条带单元构成一个条带（stripe），而一个 OSD 上属于同一个文件的所有条带单元组成一个对象（object）。这种将文件分割成多个对象，而每个对象内部又包含了不同的条带单元，每个条带又是分布在所有的 OSD 上的数据布局方式是 Cappella 所特有的。

2.1.2Cappella 副本层次

如 2.1.1 节分析的那样，当前 Cappella 系统的客户端在计算文件的 layout 时是把全局的 OSD 都囊括在其中的，也就是说每个文件的每一个条带是分布在所有 OSD 上的。在当前的这个布局方法前提下，系统所有可用的 OSD 都用于存储每个文件的条带单元数据了，这样便不会有备份 OSD 来存储副本。因此要实现副本就必须改变当前系统客户端计算文件 layout 的方法，不能一开始就把所有的 OSD 用完，而是要对 OSD 进行分组。初始计算文件 layout 时只用某一组的 OSD，其余组的 OSD 则为备份组，用于存放副本。然而在客户端并没有留出实现副本分组的相关接口，因此就要修改原始的数据结构，如何在尽量不破坏正常读写流程和逻辑的前提下实现分组以及副本布局，是需要考虑的问题。

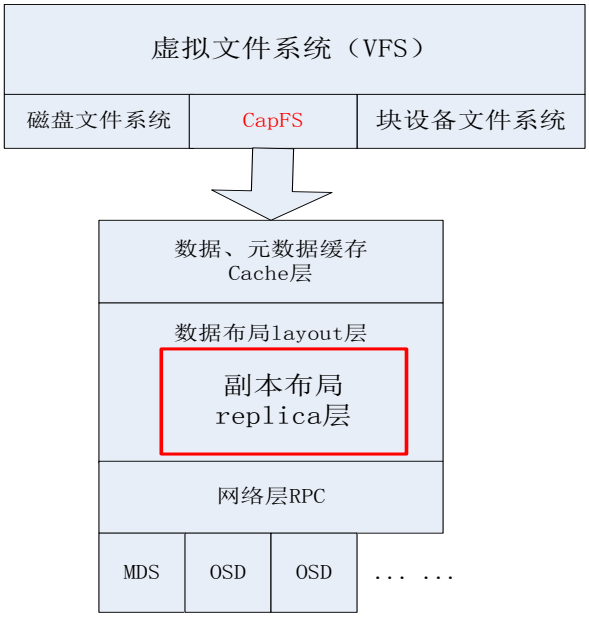

图 2.2 Cappella 副本层次

由于 Cappella 文件系统的数据布局 layout 是在客户端计算，然后从 OSD 具体读写数据，因此本文所设计的副本管理的布局机制也位于客户端层，如图 2.2 所示。

1、 数据布局层

一般而言，在 Cappella 中的一次 I/O 操作要经过虚拟文件系统层（VFS）[30]、具体的文件系统层（CapFS）、cache 层、数据布局层（layout）、网络层等。而在通过网络层传输具体的数据之前，需要经过 layout 层把数据操作定向到具体的对象存储服务器。当一个新创建的文件需要 flush 数据时，就会为这个文件计算其 layout。在 osd_layout 的 osd_data_map 中存放着文件的数据布局策略。什么叫数据分布策略？它是指能存放该文件的 OSD 的个数、条带单元（stripe unit）的大小、分条方式（简单分条或嵌套分条）、镜像数和 RAID 级别。而这是在 mount 时设置并保存在 mds_server 中的 layout_option 中的。它个文件的分布策略会在 objlayout_alloc_layout 函数中复制并保存到具体每个文件的 osd_layout 中的 osd_data_map 结构成员中。

2、 副本布局层

在客户端，副本布局层是包含在数据布局层之中的，可以算作数据布局层的一部分。总体思想是以客户端为主导，在文件第一次创建时，由客户端计算文件 layout 以及副本 layout。方案要考虑到 Cappella 文件分条存储的特性以及 OSD 间的负载均衡。需要说明的是由于 Cappella 中的文件都是分条带存放，实际存放文件数据的 OSD 是以对象的形式包含着不同的条带单元，因此所有的副本都是针对 OSD 中的对象的，是对象副本。文件的一个对象在每个 OSD 设备上的对象副本就只有一个，这个对象会随着文件的条带数的不断增加（文件增大）而对象内的条带单元不断增多，对象副本也会不断增大。在客户端 OSD 的列表信息存放在 dev_list[MAX_MIRROR_NR][MAX_OSD_NR]这个二维数组中，它是一个形式上的数组，是从客户端的视角表征本次要读写的数据是位于 OSD 数组中的第几个。而对象副本真正所对应的 OSD 编号会在 objlayout_alloc_layout 中把形式数组中的每个 OSD 元素一一对应成一个真正的 OSD 设备结构，这些真正的 OSD 设备结构是存放在 osd_dev 二维数组中的，包含了 OSD 编号 osd_id 等。

2.2 副本布局算法设计

本文副本放置方案的主要思想是基于对象和一致性哈希[31]。首先，把不同灾难域内的节点分成若干组[32]；然后将文件以某种分条策略分割成若干条带，并将不同的条带单元组成对象；最后，对同一个对象的不同副本用一致性哈希算法将它们散

列分布到多个存储节点分组中。算法的关键包括一次分组过程、一次分割映射过程以及一次哈希过程。节点分组根据权重散列在一个头尾相接的哈希环上，数据对象的副本经过特定的哈希函数算出哈希值后，对应到相应的分组中。哈希环上的节点是地位平等的虚拟节点，这些虚拟节点都隶属于不同的分组，分组的权重值高它对应的虚拟节点就多，反之就少。以对象分条的方式存储文件，一个文件被分割成多个对象，每个对象的副本存放在多个存储节点上，这样可以大大提高 IO 访问的并行性，保证系统性能。

2.2.1节点分组

分组的过程需要考虑到灾难域、设备负载等各种因素。关联度较高的对象存储设备划分为一组，也就是处于同一个灾难域中。它可以是同一个机架上的服务器，也可以是同一个交换机连接下的局域存储网络，甚至可以是一个数据中心，这些视具体情况而定。不同的分组就代表着不同的灾难域，这样一来文件的对象副本将会被分配到多个分组中，其可靠性大大提高。分组完成后，根据每个分组的负载情况分配其一定的权值用于对应相应数量的虚拟节点。所有的分组被映射到一个头尾相接的哈希环上，而哈希环中真正起作用的是权值相等的一个个虚拟节点。每个灾难域分组根据之前所分配的权值而获得相应数量的虚拟节点，这样权值高的分组其虚拟节点就多，在副本放置选取时被选中的概率就大，反之权值低的分组其虚拟节点就少，被副本放置选中的概率也就相应减小。这样可以使存储节点依据其权重值实现负载均衡，同时将不同对象副本分配在多个分组中也能够进一步保证系统的可靠性。

2.2.2对象映射

为了增加文件的并行性，提高副本 I/O 访问效率，在配置副本之前先将文件分割成若干个对象如图 2.3 所示，再以对象为单位设置对象副本。文件的一组对象副本构成文件的一份完整信息，而这一组对象是以 Round Robin 形式分布在同一组对象存储设备 OSD 中，也就是处于同一个灾难域中。每个对象其自身的副本则是选择在分组哈希环的其它不同灾难域分组中。在一组内部，文件到对象的映射是采用分片映射策略，将文件数据分成大小相同的分片，即条带单元（stripe unit），它们所组成的一个条带位于一个分组内 OSD 的同一行，而每个 OSD 上的条带单元就构成了一个

对象（object）。

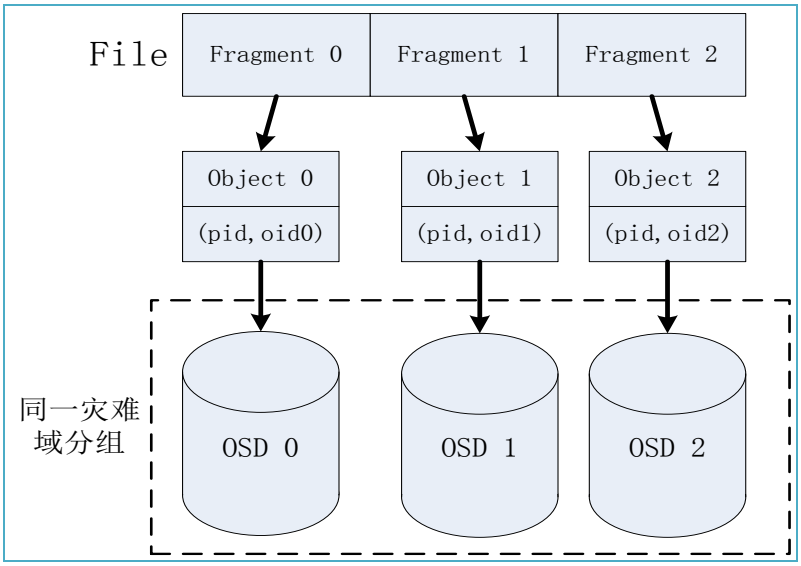

图 2.3 文件到对象映射

这样文件的一份对象副本分布在同一分组的不同 OSD 中，可以使其具有较好的并行性。由于同一分组的 OSD 是处于一个灾难域内部的，它们之间的连接比较直接，通讯距离较短，因而对一个分组内部的对象副本并行访问更加迅捷，I/O 吞吐率更高。同时，同一对象的不同副本又是分布在不同灾难域中，一个灾难域内的对象副本失效，另一个灾难域内对象副本依然可以使用，这样可靠性也得到了保证。

2.2.3一致性哈希计算

在对象存储节点分组完成以及文件也映射成对象之后，就需要通过一致性哈希算法来计算出每个对象副本所处的分组节点了。采用一致性哈希算法可以使对象副本放置适应存储节点动态加入删除的场景。当有节点退出系统时，一致性哈希可以让其映射关系只在系统内部部分更新而不用全部更新。当有节点加入系统时，一致性哈希也可以让对象副本的映射结果尽可能的映射到新的节点分组中从而使节点负载尽快均衡。

如图 2.4 所示，整个哈希空间成圆环状，被权值相等的虚拟节点所分割。而这些虚拟节点又都隶属于不同的分组，分组的权重值高它对应的虚拟节点就多，反之就少。每个文件对象的主副本经过哈希运算后得到的哈希值 key 也被映射到这个哈希环上，那这个对象副本必然会处于某两个虚拟节点之间的区间中。

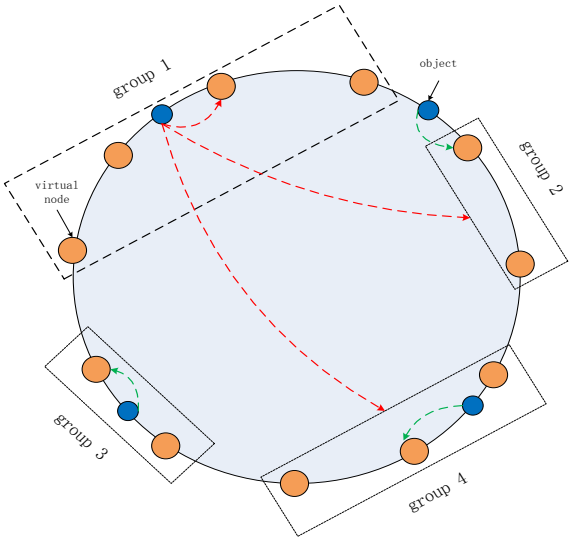

图 2.4 一致性哈希过程

在这个哈希环上，从对象的哈希值 key 开始顺时针往后遍历，所达到的第一个虚拟节点就是基准节点，那么就将该对象主副本存储在这个虚拟节点上。因为对象和虚拟节点的哈希值是固定的，因此这个节点必然是唯一确定的，如此一来便将对象主副本和虚拟节点一一对应起来了。又因为虚拟节点都是包含于现实的存储服务器分组之中的，这样一个对象主副本就被映射到一个分组中去了。确定好对象主副本之后，再顺时针的沿着哈希环将此主副本的其它副本存放在后面的分组中。通过这样的一致性哈希计算，对象节点分配不仅能满足单调性和分散性，还能具备随机加权概率选择特性，即权值大、包含虚拟节点多的分组，被对象映射命中的概率就大，这样系统整体负载就会更加均衡。

2.3 副本布局算法分析

节点分组是由不同的存储节点个体组合而成，对象副本也是这样。实际上分布式文件系统中的副本布局可以看成就是从一个集合（副本集合）到另一个集合（节点分组）的映射问题[34]。将节点分组集合与对象副本集合分别做如下定义：

组内节点集合：$G_i = \left\{ g_{(i,j)} \middle| 1 \le j \le N_i \right\}$，$g_{(i,j)}$ 表示节点分组 i 中的节点 j，$N_i = \left\| G_i \right\|$，为分组 i 的节点数。

分组集合：$G = \left\{ G_i \middle| 1 \le i \le N \right\}$，$N = \left\| G \right\|$，为分组个数。

分组权重集合：$GW = \left\{ GW_i \middle| 1 \le i \le N \right\}$，$GW_i$ 表示分组 i 相对于总的分组集合的

权重比例值，$N = \|G\|$表示分组总数。

对象副本聚合：$O = \left\{ O_{(i,j)} \middle| 1 \le i \le m, 1 \le j \le k \right\}$，$O_{(i,j)}$表示对象 i 中的副本 j。

通过对副本布局算法的分析可知，集合O与集合G都通过哈希函数而被映射到哈希环上，每个分组G_i具有不同的虚拟节点数即具有不同的权重值。节点分组的总权重是一定的，于是有：

$$\sum_{i=1}^{N} GW_i = 1 \text{且} \|GW\| = N$$

从对象副本集合到节点分组集合的映射关系如图 2.5 所示。

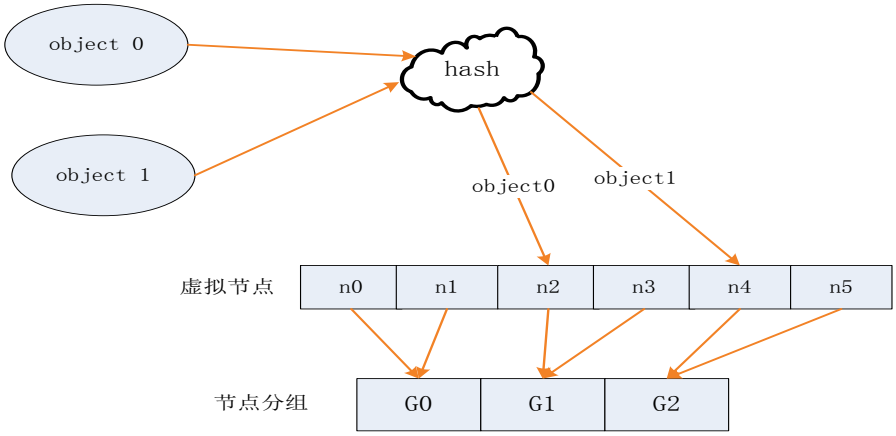

图 2.5 对象到分组映射

算法在分组时将G_i中的$g_{(i,j)}$联合起来以虚拟节点的形式映射到哈希环上，得到其哈希值。设某个分组在总的分组集合所占的权重比例值最小，而它包含的虚拟节点的总数为$GV_{\min}$，则G中的每个G_i分组其映射出的虚拟节点GV_i有如下关系：

$$\|GV_i\| = \frac{GW_i}{GW_{\min}} \times GV_{\min} \text{且} GV_i \cap GV_j = \varnothing (i \ne j)$$

$$\|GV\| = GV_{\min} \times \sum_{i=1}^{N} GW_i$$

对于每个分组G_i，其所包含的虚拟节点数是由它里面的每个实际节点$g_{(i,j)}$所等价的虚拟节点数之和。假如令$g_{(i,j)}$的权重相对于其自身分组G_i所占的比例值为$gW_{(i,j)}$，包含的虚拟节点个数为$gV_{(i,j)}$则有：

$$\sum_{j=1}^{N_i} gw_{(i,j)} = 1$$

$$gv_{(i,j)} = gw_{(i,j)} \times GV_i$$

则无论选取哪个对象副本o，将它分配到$g_{(i,a)}$的概率与将它分配到$g_{(i,b)}$的概率其比例值都为

$$\frac{P\left(o \rightarrow g_{(i,a)}\right)}{P\left(o \rightarrow g_{(i,b)}\right)} = \frac{gw_{(i,a)} \times \left(GW_i / GW_{\min} \times GV_{\min}\right)}{gw_{(i,b)} \times \left(GW_i / GW_{\min} \times GV_{\min}\right)} = \frac{gw_{(i,a)} \times GW_i}{gw_{(i,b)} \times GW_i}$$

由此看出一个对象副本所选取节点的概率与此节点的权重是成正比的，因此这个算法是均衡的。算法流程如图 2.6 所示。

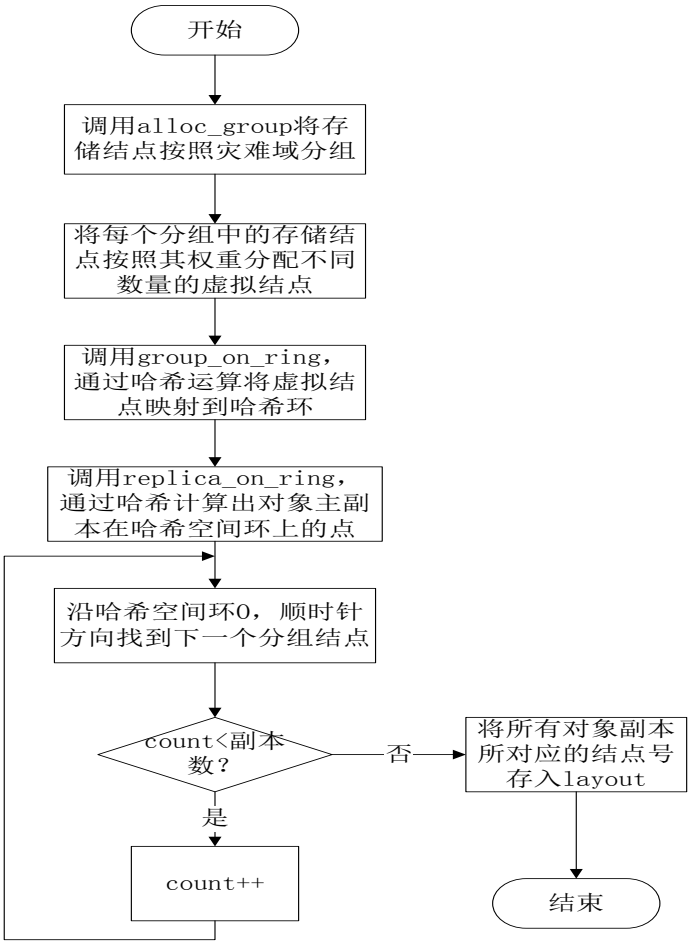

图 2.6 副本布局算法流程

2.4 Cappella 副本布局实现

2.4.1分组控制与布局计算

本方案的实现利用 Cappella 的分条特性，在条带化对象的基础上加入分组和哈希的思想以实现最终的副本布局。具体实施方法是当一个文件在创建时，客户端实时的向 MDS 获取当前最新的对象存储服务器 OSD 的全局列表，在得到这个全局列表之后，客户端便会通过副本布局的分组和哈希算法将这些对象存储节点分好组并映射到哈希环上。然后客户端再根据一定的负载均衡策略选出一组 OSD 用于存放文件的条带单元，这组 OSD 存放的就是当前文件的主对象副本。每一个主对象副本位置的选取都是通过一致性哈希计算得出的。文件以及对象副本的布局信息存放在 osd_layout 数据结构中，其结构如表 2.1 所示：

表 2.1 osd_layout 结构

struct osd_layout	
struct osd_data_map	olo_map
unsigned int	olo_comps_index
unsigned int	olo_num_comps
unsigned int	dev_list[MAX_MIRROR_NR][MAX_OSD_NR]
struct osd_object_cred	olo_comps[MAX_MIRROR_NR][MAX_OSD_NR]

struct osd_data_map 存放的是数据分布的策略，包括能存放该文件的 OSD 的个数、条带单元（stripe unit）的大小、分条方式（简单分条或嵌套分条）、镜像数和 RAID 级别。它是由在 mount 时设置并保存在 mds_server 中的 layout_option 中的信息传递而来的。

struct osd_object_cred 和 struct dev_list 是两个数组，数组的每一个元素对应一个对象存储设备 OSD。在 Cappella 文件系统中，每个文件对应的对象由 Partition_ID 和 Object_ID 共同标识。文件在创建时会由元数据服务器 MDS 分配一个全局唯一的标识（Global_ID），文件对应对象的 Partition_ID 和 Object_ID 由 Global_ID 分解得到，因此这样得到的对象标识也是全局唯一的。在每个对象存储服务器 OSD 上属于同一文件的对象具有相同的 Partition_ID 和 Object_ID，不同文件的对象拥有不同的对象标识。所有的 OSD 在客户端的文件 layout 中，是以 OSD 数组的形式表示的。struct osd_object_cred 中最重要的成员就是 struct olo_obj_id，它里面就是存放 pid 和 oid 的，而 struct dev_list 则是用于存放所有可用的 OSD 的 osd_id

olo_comps_index、olo_num_comps 两个成员则是关系到文件以 RAID 方式分布到 OSD 时要用到的信息。olo_comps_index 代表这个文件写到的第一个 OSD 在所有 OSD 中的编号。olo_num_comps 则表示可以用的所有 OSD 的数目。

副本的放置与 layout 计算时机是与计算主对象副本同步的。在计算出文件主副本的 layout 后，客户端会进一步通过后续副本放置算法计算得到各个对象副本的 layout，并一同保存在客户端文件的 osd_layout 结构中。副本放置算法的选择主要考虑的是让副本分布在不同灾难域中的 OSD 上。这样每一组对象副本放在同一组 OSD 中，分割的对象使文件的访问具有更好的并行性，而同一组内 OSD 又因其相互之间较短的通信距离而具有更好的 I/O 性能。

在实现过程中，选出的第一组用于存放原始文件的 OSD（对象存储设备）数目存放在 layout.olomap.odm_num_comps 字段中，而文件的对象副本 layout 信息则可以通过 mirror 方式存放在 layout.olo_comps 的二维数组中。另外，在客户端的 obj_layout 结构中加入 osd_index 和 osd_end 两个字段，它们用于表示 OSD 分组中的起始和终止 OSD 号。在客户端从 MDS 获得相应的 OSD 列表后，它先是逻辑上将所有的 OSD 进行编号，获得全局视图。然后通过相应的灾难域策略和副本的布局算法算出具体的 osd_index 和 osd_end 值。读写操作时，则将现有的系统当前 layout 计算规则加以修改，用 osd_index 和 osd_end 来屏蔽全局 OSD 视图。这样在客户端中就会把加入副本后的 osd_index 和 osd_end 之间的 OSD 视图当作原始的全局 OSD 视图，而不会对当前正常读写逻辑产生颠覆性的改动。

2.4.2 读写控制与副本刷回

在 Cappella 中所有的 page 都是用一个 cap_pg 结构封装，如表 2.2 所示。

表 2.2 cap_pg 结构

struct cap_pg {		
struct inode*	wb_inode	/*请求上下文信息*/
struct list_head	wb_list	/*待刷回 cap_pg 链*/
struct page*	wb_page	/*所封装的具体 page 信息*/
atomic_t	wb_complete	/*I/O 状态*/
unsigned int	wb_offset, wb_bytes	/*page 的偏移、I/O 长度*/
}		

在 cap_pg 中会记录要写入当前这个 page 中的新数据在 page 中的起始量

（wb_pgbase）和长度(wb_bytes)。为什么要记录这些，因为在客户端来说，保存在缓存的这个 page 最终并不是刷到本地的，而是要刷到远端的 OSD（对象存储设备）。所以对于 OSD 来说，在客户端处理过后的一个 page 页成了新的源数据，在元数据必须要有偏移和长度。所以说一个 page 页只是内核中的原子单位，但不是 Cappella 文件系统的原子单位,因为在客户端形成的 page 并不是作为一个写入单元刷下本地，而是形成源数据传给 OSD。

副本对象的写入是与文件刷回时机保持一致的。也就是说原始文件以及副本的 layout 信息在计算好后都是存放在 osd_layout 中的，在刷回文件数据的时候同时的把对象副本一并刷入到其对应的对象存储设备节点 OSD 中。这样做可以使副本刷回与 Cappella 文件系统文件刷回逻辑保持契合从而尽量使原始读写流程保持完整。另外由于对象副本的 OSD 与原文件所对应的一组 OSD 一般是分开的，因此这样也不会使单个 OSD 的带宽达到瓶颈，对整体带宽性能的影响会比较小。

客户端通过 cap_file_write 函数将数据写到 VFS 的页高速缓存 page cache[33] 中后再利用内核 page cache 的 flush 机制将副本数据持久化到后端设备。当内存不足（如 grow_buffer、try_to_free_pages 等分配页面失败）或是显式的请求刷新操作(如 sync、fsync、fdatasync)时，内核会执行 wakeup_pdflush()函数启动 pdflush 线程，在 pdflush 线程中会进一步调用 Cappella 文件系统所提供的相应接口进行刷回操作。客户端在每次 I/O 中会尽量使数据累积到一定页数（pg_bsize）再一起刷回，这样可以减少 I/O 次数，最大限度利用网络带宽[35]。当然前提条件是这些 page 要是连续的，而这个机制则是由 struct pageio_descriptor 这个结构来控制的。

表 2.3 pageio_descriptor 结构

struct pageio_descriptor {		
struct list_head	pg_list	/*cap_pg 结构双向链表,链接 pages*/
unsigned long	pg_bytes_written	/*已写字节数*/
size_t	pg_count	/*本次 I/O 要写的字节数*/
size_t	pg_size	/*mount 时指定的最大读写字节数*/
unsigned int	pg_base	/*开始读写位置在本页偏移*/
struct inode*	pg_inode	/*文件对应的 inode*/
}		

每一次 flush，对应一个 pageio_descriptor，这一个 pageio_descriptor 负责把一个文件 inode 基树（mapping）中的所有对象副本的脏页刷完。每次刷回的时候都会把脏页尽量连续累积到 pg_bsize 最大 I/O 长度后一起刷回。

2.4.3实现流程图

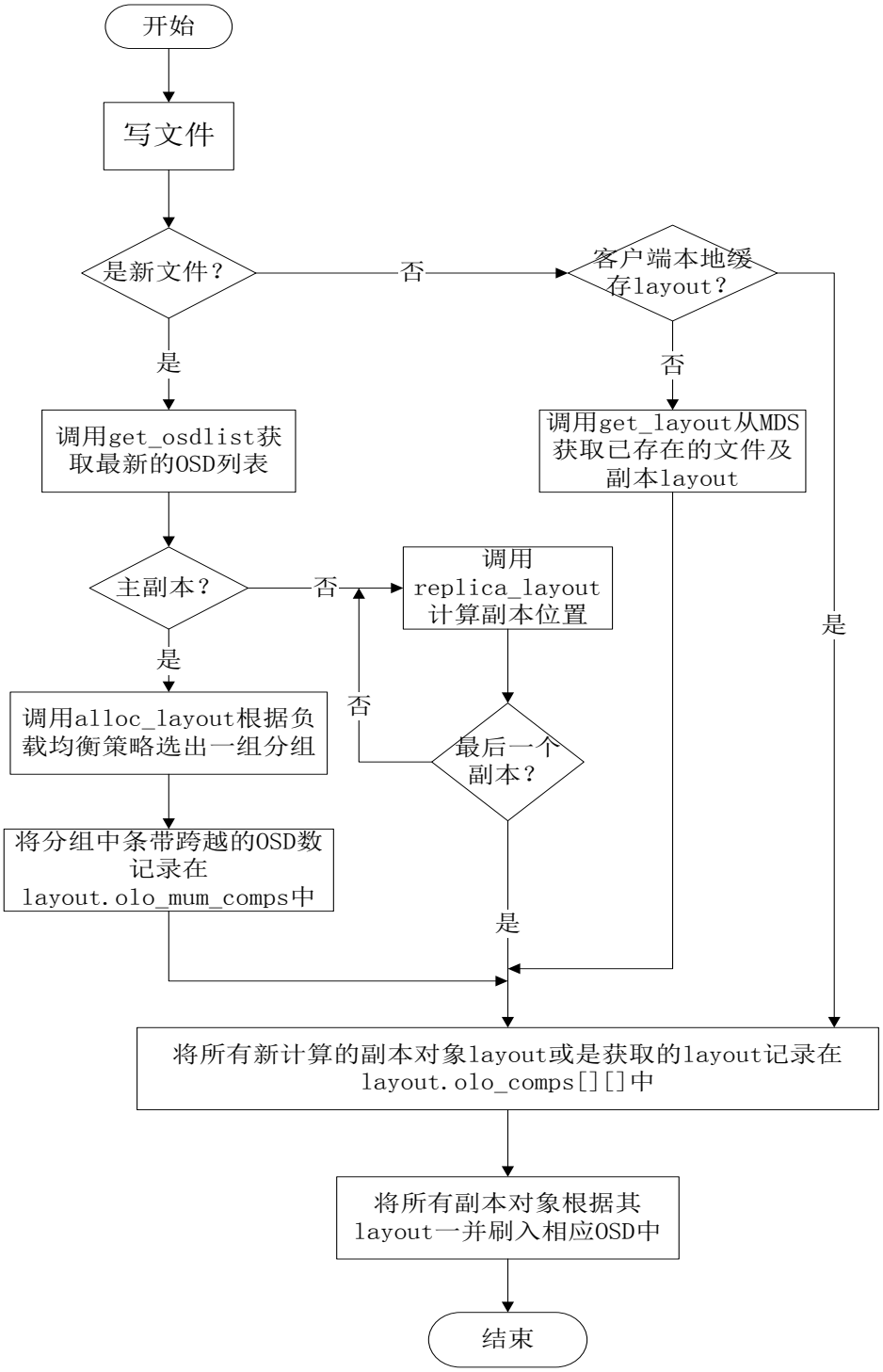

图 2.7 Cappella 副本布局流程

图 2.7 所示是 Cappella 文件系统副本布局的流程，综上来看本方案的副本放置

策略是以客户端为驱动，以文件为 layout 计算单位来得到文件对象以及副本的 layout 信息的。与传统的客户端只在 mount 的时候静态的得到 OSD 列表从而单一的决定 layout 不同，这样的每个文件都可以动态的选择不同的 OSD 来存放条带单元，从而可以维持系统的负载均衡并能更好的适应副本放置算法。

2.5 本章小结

本章从副本布局算法设计、分析以及在 Cappella 文件系统中的实现过程等方面详细介绍了分布式文件系统的副本布局机制。首先在算法设计方面，将算法的主要思想分为分组、映射、一致性哈希三个过程，并分别进行了阐述，表明此算法可以使对象存储节点依据其权重值实现负载均衡，同时可以提高 IO 访问的并行性，保证系统性能。然后对此算法进行了抽象化分析，论证了其对象节点选取的概率均衡性。最后在分布式文件系统 Cappella 上进行了具体实现，描述了主要的实现过程以及关键数据结构。

3 分布式文件系统副本恢复机制

一个好的副本布局机制可以最大限度的维护系统性能，使其具有较高的 I/O 吞吐率和效率。然而副本管理机制的另一面，即副本保障和恢复机制才是提高分布式文件系统可靠性的关键。保障系统可靠性的副本恢复机制包括两个方面，一是失效检测与降级读写，二是数据迁移与恢复。在本文所研究的副本管理机制中，副本的降级读写与恢复机制是很重要的一方面。整个副本管理机制方案的总体模块图如图 3.1 所示。

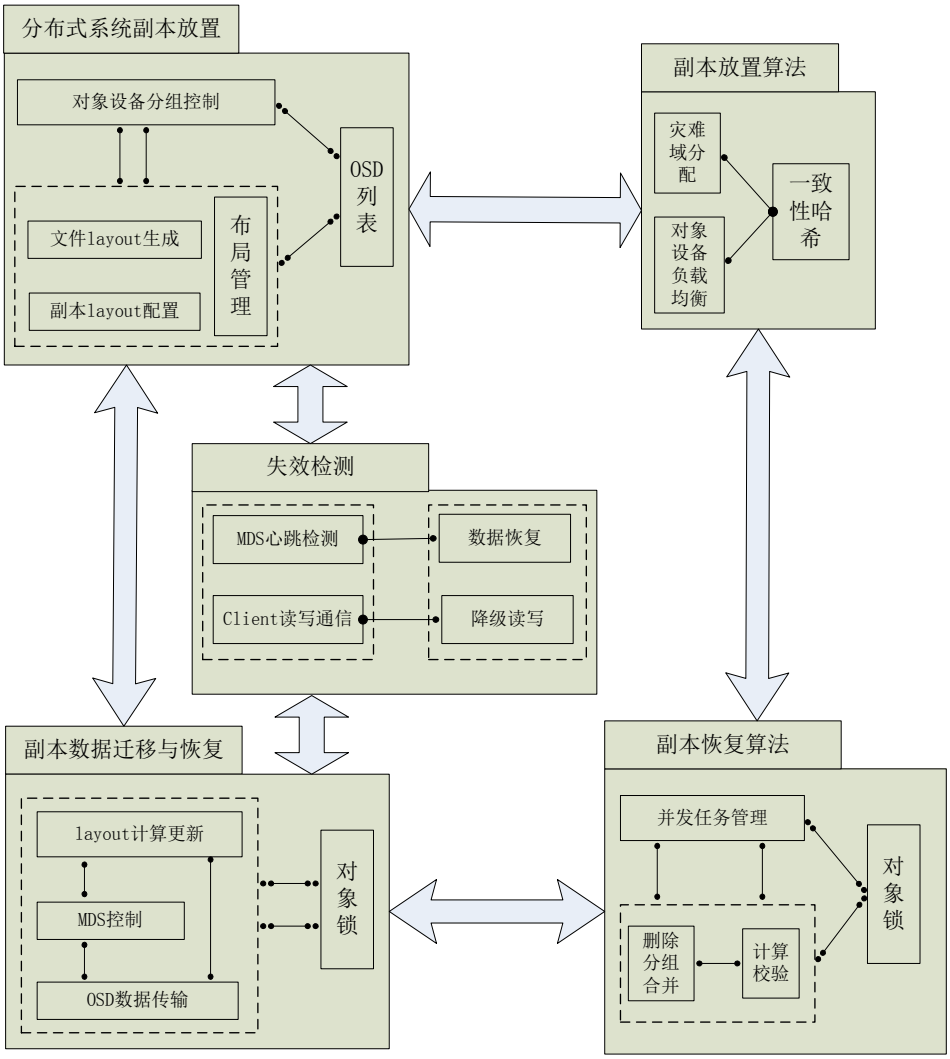

图 3.1 副本管理机制模块

当有节点失效时，具有高可靠性的分布式文件系统需要能够正常地响应读写请

求，即提供失效服务。一方面系统要能及时准确的探测到失效的存储节点，并能分析出其所包含的对象副本。这既是响应降级读写的依据，也是后续数据恢复工作的前提基础。另一方面在节点失效后，副本数据恢复前，对于到来的 I/O 请求要能够正常响应，降级读写。这样才能体现多副本保障系统可靠性的优势。

当探测到节点失效后，要能够及时的恢复失效节点副本，使系统迅速的恢复到原来的冗余级别[37]，否则系统整体的可靠性就会降低。例如在三副本方案下，若出现两节点失效仍未进行副本恢复的情况，那么再失效一个节点时，就有很大机率出现某对象的副本完全丢失，既不能提供降级读写也无法再进行恢复。而在数据恢复期间，系统的可靠性比较低，对外提供服务的性能下降，因此副本数据恢复的时效是一个非常重要的衡量标准。要尽可能快的恢复数据副本，提供文件系统可靠性保证。

3.1 失效检测与降级读写

通过副本提高系统可靠性的关键就是要能够实现对存储节点的容错。也就是说在多副本的情况下，若有 OSD 失效，在副本尚未恢复之前，客户端要可以在降级模式下正常读写。而要实现这个的关键就是在客户端与 OSD 交互 flush 数据时，一旦发现有 OSD 无法连通、数据无法刷成功，则要及时的向上层返回写失败的状态，并做出正确合适的错误处理。

3.1.1Cappella 三方交互

缺乏副本保障机制的 Cappella 文件系统对于对象存储设备 OSD 的宕机失效是零容忍的，出现类似状况毫无应对机制。本文的失效节点失效检测机制是利用 Cappella 客户端（Client）、对象存储端（OSD）以及元数据服务器端（MDS）三方之间的交互来完成的。一方面使 MDS 与 OSD 之间维持心跳信息，使元数据服务器能定时的检测系统节点状态；另一方面加入 Client 的反馈机制和反应动作，客户端读写失败时，系统能迅速及时的探测到失效节点并做应对处理。

实现时， MDS 与 OSD 之间的心跳机制是对象存储节点 OSD 会定时向元数据服务器 MDS 发送心跳信息，以证明自己处于运行在线状态。若 MDS 监测到对象服务器在一段时间内没有发送心跳信息时，便会认定该对象服务器失效，系统中默认的设置是连续 3 次接受心跳信息失败，即认为节点失效。但是这样做还不够，本方

案进行了进一步扩充和完善。首先在发现 OSD 失效时，要将其从集群中清除，具体就是更新 MDS 的 OSD 列表，删去失效 OSD，并按规则重新编号剩余 OSD。其次仅仅是 MDS 本身更新 OSD 列表还不够，因为它无法将新的 OSD 列表告知客户端，所以需要再加入客户端的反馈机制。具体实现就是在客户端中访问 OSD 时，发现对象设备无法连接，此时这个客户端就向 MDS 发起反馈连接，在这个连接中它会传递失效的 OSD 编号并请求新的 OSD 列表。这样 MDS 在经过验证之后就可以向相应客户端发送最新的 OSD 列表。最后在客户端得到新 OSD 列表之后必须有相应的反应动作。实现中客户端在检测到 OSD 失效之后，根据失效 OSD 在此客户端的逻辑 ID，通过本地副本布局算法计算得到失效 OSD 的副本位置。然后将此副本标记为主副本，并同时标记失效 OSD，在之后的读写过程中只读写主副本，在恢复之前不再读写坏的 OSD。

另一方面，本方案不仅能检测失效的 OSD 节点，当有新的 OSD 加入系统时，也能进行检测。这样一来也为副本恢复与数据的迁移创造了便利条件。而此机制的实现方案是在客户端中独立创建并启动一个检测线程，它的作用就是负责定时的从 MDS 检测是否有新的 OSD。对于分组中有失效宕机的客户端，它便会通过此线程向 MDS 申请将新 OSD 加入自己的分组中。若获得 MDS 的同意和相应回馈信息，客户端便可以通过副本布局算法更新 OSD 分组编号，更新 osd_index 和 osd_end 值，进而触发 OSD 副本恢复的工作。

3.1.2 降级读写模式

要让系统实现对 OSD 对象存储设备的容错，关键就是要在 OSD 宕机之后客户端能够正确有效的做出反应。也就是说在多副本的情况下，若有 OSD 失效，在副本尚未恢复之前，客户端要可以在降级模式下正常读写。而要实现这个的关键就是在客户端与 OSD 交互 flush 数据时，一旦发现有 OSD 无法连通、数据无法刷成功，则要及时的向上层返回写失败的状态，并做出正确合适的错误处理。

在降级读写时要注意数据的一致性[38]问题。写操作要优先更新主副本，如主副本处于失效节点，则要及时的选择一个从副本变为主副本并作为一致性更新的标准。在 Cappella 文件系统中，由于存在并发写操作，因此还需要注意并发写顺序对数据一致性的影响，要确定规则对数据提交顺序进行控制以使对象副本的最新版本能被唯一确定。读操作则要根据读取数据的范围选择合适的对象副本读取，要确保读取的数据是最新数据而不是失效前不一致的数据。

在 Cappella 文件系统中实现的降级读写流程如图 3.2 所示。

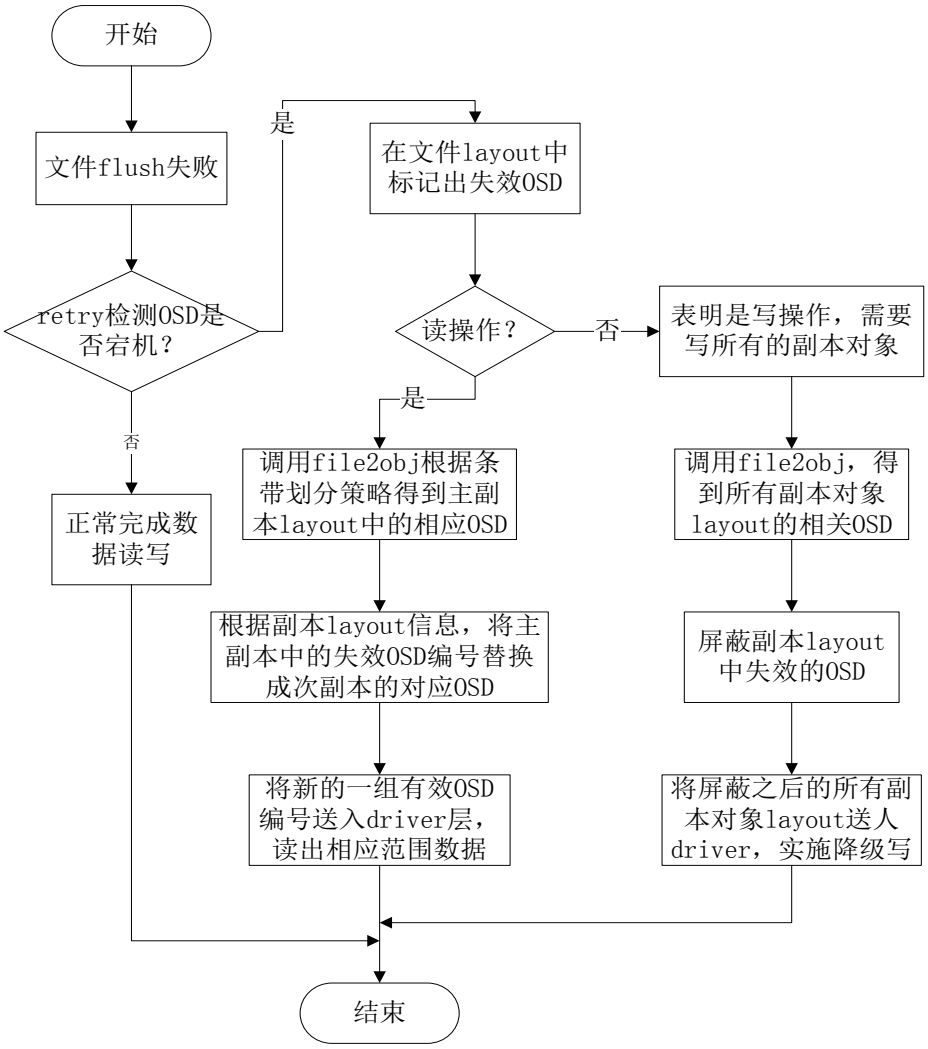

图 3.2 Cappella 降级读写流程

如图 3.2 流程所描述的那样，处理方案的第一步就是在 osd_layout 中标记出失效的 OSD 编号，以避免后续对错误 OSD 的重复读写。接下来，对于读操作，首先根据条带划分策略得到所读取的数据范围所对应的不同 OSD，并进一步通过原始文件即主副本的的 layout 信息得到相应的 OSD 编号。然后，如果这些 OSD 编号中包含失效 OSD 的编号，那么就要根据此文件的 osd_layout 中所存的副本 layout 信息，将此对象对应的副本对象的 OSD 编号替换失效 OSD。最后将这一组全部有效的 OSD 编号传入下层进行读操作。

写操作与读操作又不同，因为写操作要写所有的对象副本，所以第一步就不是仅仅得到主副本的相应 OSD 编号，而是要根据源数据的范围，在 osd_layout 得到所

有相关对象副本的 OSD 编号。在这所有的 OSD 中，若有失效的 OSD，则直接略过不写，其余的 OSD 则正常写入相应副本。也就是说，当 OSD 宕机时，对于失效 OSD 上的对象，它就只写双副本，而其余 OSD 上的对象则正常的写三副本。这样的话既避免了重复写错误的 OSD，又能让其它有效的 OSD 上的副本全部得到正确的更新，从而实现降级模式下的写操作。若 MDS 将失效 OSD 上的对象恢复到某个选定的 OSD 上并且完成时，MDS 会将 layout 元数据上锁并且在 MDS 端更新恢复好之后的 layout。当更新完毕后，client 再次写文件时就需要重新从 MDS 获取 layout 锁，进而就能在客户端上更新得到最新的 osd_layout 信息，而新的 osd_layout 的失效副本已经被恢复，那么之后的写操作客户端又可以正常的写三副本了。当然 MDS 在更新完 layout 后也可以广播给所有的 client，让 client 主动的更新相应 layout 信息，这样可以让客户端更加及时的得到最新的 osd_layout。

3.2 Cappella 副本恢复

3.2.1对象设备管理

分布式文件系统中，对象副本最终是存放在对象存储设备 OSD 上的，它的组织和管理对系统副本恢复有着很大影响。在 Cappella 的 OSD 中，其对象副本数据流格式如图 3.3 所示。

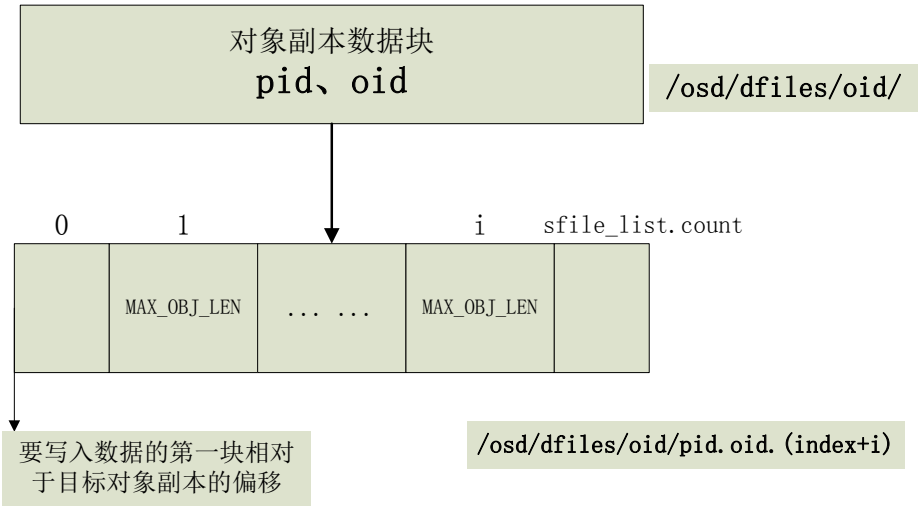

图 3.3 OSD 对象副本数块

OSD 中的对象副本被划分的每一块(MAX_OBJ_LEN)，pid 和 oid 相同，但是 sid(index+i)不同。因此，在 OSD 存储端，它们是同一个目录下(../osd/)的不同小文件。

I/O 请求数据块被划分成一个个小的块，长度为 MAX_OBJ_LEN。而要写入的目标文件也已经被划分成 MAX_OBJ_LEN 的小块存储。这些待写入的数据小块，有可能是目标文件已存在的块（即修改已存在文件小块的数据）；也有可能是目标文件没有的小块（即在目标文件后面追加数据），这就需要新创建小块文件。不管是目标文件已存在的小块，还是目标文件没有的小块，都要 open，即把这些小块读到内存中。在副本恢复时需要定位到失效对象副本的每个数据块小文件，然后从适当的备份 OSD 中选择其相应的副本进行恢复。这样分小块文件存储 OSD 对象副本可以在副本恢复时增加任务并行度，提高恢复速率。

3.2.2恢复策略

因为在副本布局中，对象主副本的确定是在哈希环上沿着顺时针方向从对象的哈希值 key 出发，直至遇见的第一个虚拟节点所属的分组；而它对应的其它从副本则是在确定好对象主副本之后，顺时针的沿着哈希环选取后续的分组。所以每个对象其副本是存放在哈希环上相邻的若干分组中的。假设副本数为 n，并且用$V_{(k,\mathrm{i})}$表示分组G_i的第 k 个物理 OSD 节点，则有如下关系：

$$V_{(k,i)} \to \left[V_{(k,i+1)} \cdots V_{(k,i+n-1)}\right]$$

$$\left[V_{(k,i-n+1)} \cdots V_{(k,i-1)}\right] \to V_{(k,i)}$$

也就是说，对于任意一个对象副本O_j，若它是G_i分组中的 OSD 节点$V_{(k,\mathrm{i})}$中的主副本，那么它一定在分组G_i的后续 n-1 个分组中存在副本。而对于分组G_i前面的 n-1 个分组，其任意一个分组中的对象都会在节点$V_{(k,\mathrm{i})}$中存储对象副本。

以三副本为例（即 n=3），当节点$V_{(k,\mathrm{i})}$失效时，说明分组G_i状态产生异常错误，那么在副本恢复时，它的后继 n 个分组的对象副本状态就会收到影响而产生变化，如图 3.4 所示。

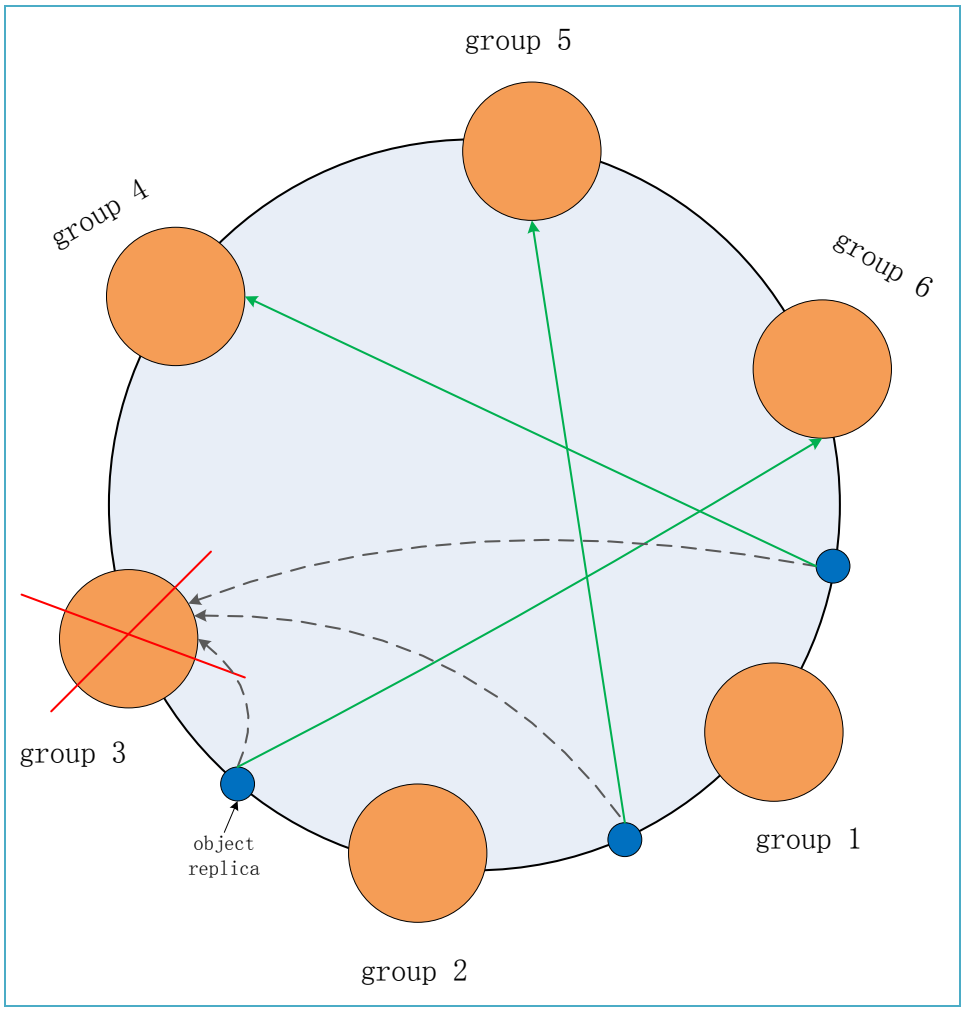

图 3.4 分组节点失效数据迁移过程

图 3.4 所示的数据迁移过程表明，当分组G_3中有节点失效时，映射到它之上的对象副本会根据不同情况分别被迁移恢复至G_4、G_5、G_6三个分组的节点上。若对象副本O_j被映射在$(G_2, G_3]$区间上，那么它存放在G_3分组中的是对象主副本，因此它的相应对象副本数据会被恢复至G_6分组中，并将G_4分组中的相应从副本设置为主副本；若对象副本O_j被映射在$(G_1, G_2]$区间上，那么G_3分组中存放的则是其对象从副本，因此只需把它的对象副本数据恢复至原始哈希环中主副本分组的后面第 n 个分组中，也就是G_5分组；同理分布在G_3分组中的对象副本以及在$(G_6, G_1]$区间上的

对象，它们的副本将被恢复至G_4分组上。

3.2.3数据迁移

在 Cappella 文件系统中，文件的一个对象在每个 OSD 设备上的对象副本就只有一个，当文件增大时，这个对象会随着文件的条带数的不断增加而对象内的条带单元不断增多，对象副本也会不断增大。副本恢复的本质就是将对象数据在 OSD 之间进行迁移。

本文的数据恢复方案所采用的时机是在失效分组的客户端检测到有新 OSD 加入时，而真正的恢复及数据传输的过程又是由元数据服务器 MDS 主导控制。也就是说由客户端反馈，由 MDS 触发控制。

元数据服务器会分隔出两个区间的副本文件，然后向对应的 OSD 分组发送恢复任务，相应 OSD 数据服务器返回状态，更新布局信息。为了保持个对象存储设备间的负载的衡，元数据服务器还能动态调整备份数据服务器上正在进行的恢复任务的数目。

实现过程中，MDS 通过心跳信息与 OSD 保持通信，当检测到有 OSD 失效后，经过一段预先设定的短暂失效时间之后，MDS 开始发起失效 OSD 的副本恢复工作。首先 MDS 通过具体的副本位置算法，选取一个合适的 OSD 用于恢复失效副本，并开始着手副本恢复任务。然后 MDS 通过它所保存的元数据信息计算出失效 OSD 上的所有对象号并根据本地保存的文件 layout 信息得到每个要恢复的对象的副本 OSD 编号，并从这些 OSD 上读取相应对象。在这个过程中对源数据 OSD 上写锁，所有对这些源数据对象的写操作都要阻塞至相应的副本恢复完毕。在对象副本恢复结束、layout 更新完毕之后，新的写操作自然就会写这些恢复后的副本，从而到达副本的一致性。副本数据迁移恢复的流程如图 3.5 所示。

这样实施的原因是在 OSD 的组织中并没有主副 OSD，所有的 OSD 是以扁平的形式组织的，因此新加入的 OSD 无法知道其它 OSD 的元数据信息。另外 OSD 也不能对一些全局的操作进行有效的处理。而本方案是 MDS 通过计算或是向客户端获得失效 OSD 上的每个数据对象副本，然后向新的备份 OSD 发送恢复任务。它能随时的更新布局信息，控制全局操作，它的优势就在于 MDS 上存放着系统的全局信息。

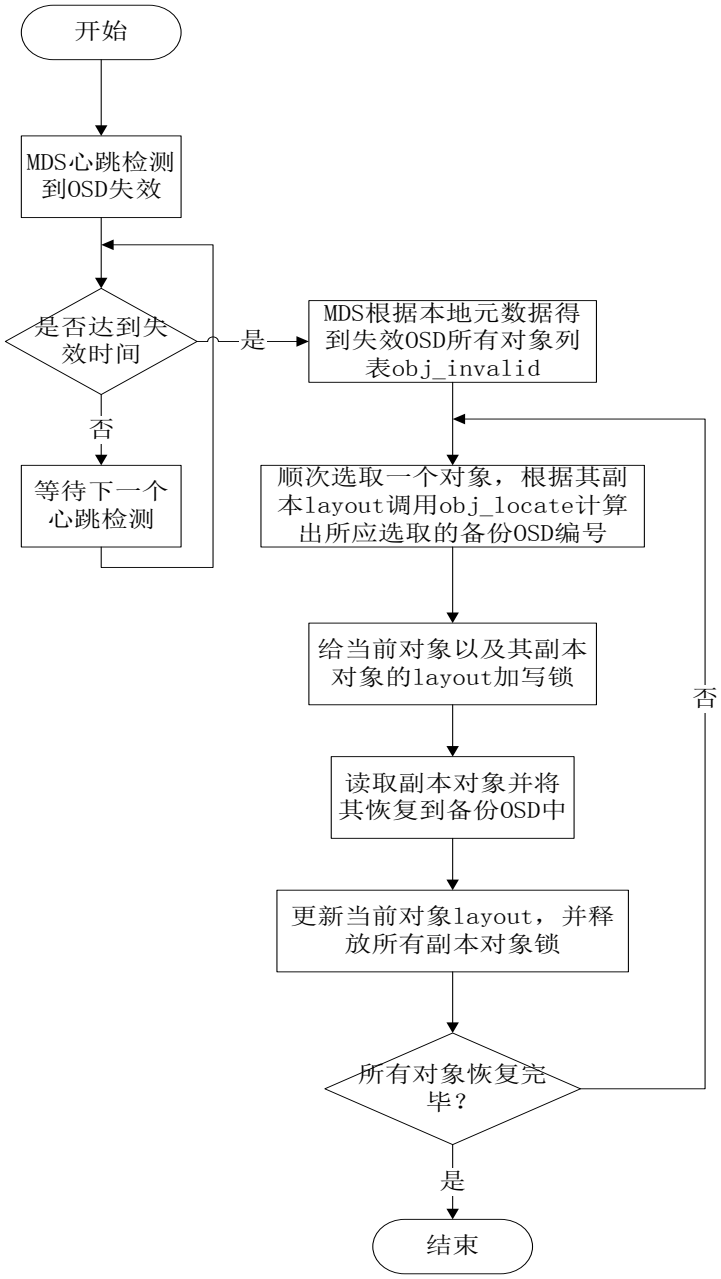

图 3.5 Cappella 对象副本恢复流程

3.3 性能优化

在副本数据恢复期间，系统的冗余级别下降，可靠性比较低，对外提供服务的能力也随之下降，因此对象副本恢复的时效性非常重要。提高副本恢复的速度和性能可以减短系统的脆弱时间，使系统可靠性得到进一步提升。在本文的副本恢复机

制中，通过采用对象存储端（OSD）缓存与多线程流水线恢复的方法提高副本恢复过程的性能。

1、 OSD 缓存

在 OSD 端增加缓存，恢复副本需要读取的对象预先读取到缓存中。在 MDS 计算出副本位置信息并发送至相关节点后，OSD 首先在缓存中查找所请求的对象副本是否存在，若命中则直接通过 RPC 将数据传至目标备份 OSD，否则从磁盘中查找并读取相应数据。

OSD 的读缓存是以红黑树[36]的形式组织的，它具有自平衡的特性以及良好的查找性能，而且相对于基树来说它更节省内存空间。缓存与 OSD 组织对象副本的形式相对应，是把文件分成小块来进行缓存，块大小为 OSD_BUFF_BLOCK_SIZE， 值为 1M。对于到来的请求，先查找红黑树，命中，则读取返回；否则，调用 contig_read 从磁盘读取数据。根据数据访问的空间局部性，这个被读取的对象副本很可能会被再次请求，因此在从磁盘读取了相应对象副本文件后，会创建一个 object_pair 结构插入到 rbtree 中，即将对象副本数据放置缓存中，如图 3.6 所示。

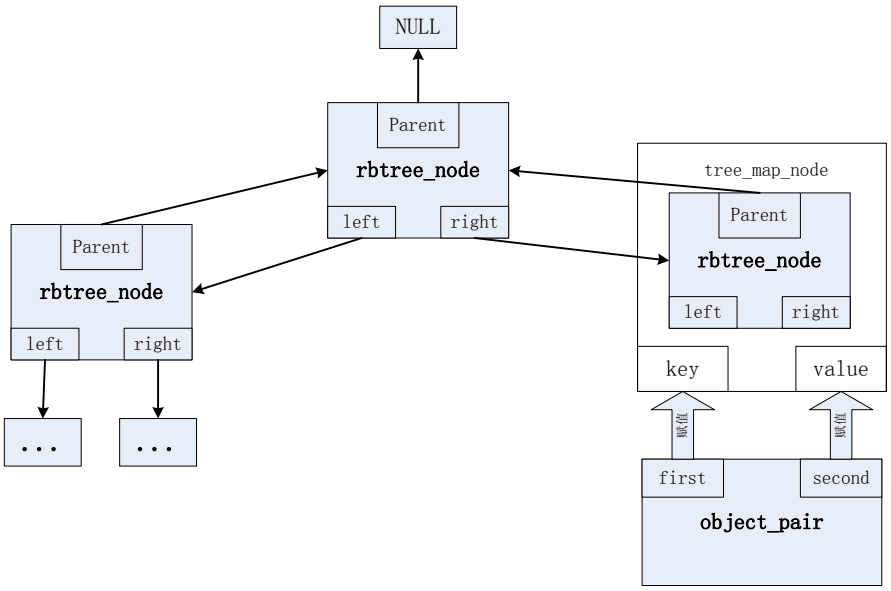

图 3.6 OSD 对象缓存

在 rbtree 上，对每一个 OSD_BUFF_BLOCK_SIZE 的小块，只缓存对应一个节点。当接收到从 RPC 传递过来的对象副本时，通过红黑树 rbtree 的 compare 方法比较 key 值来确定缓存是否命中。Key 值有三个域，pid、oid、和 offset。只有当，pid 相等、oid 相等、offset 在同一个小块，这样才视为是同一个节点，也就是说目标对象副本缓存命中。

2、 多线程流水线

根据副本布局算法，失效 OSD 的对象副本会被恢复到多个 OSD 上。由于每个 OSD 是相互独立的具有存储计算能力的服务器，因此可以通过多线程方式增加并发数以加快副本恢复速度。而对于每个恢复线程，可以通过流水线的方式把它的工作分为几个步骤穿插进行，从而进一步缩短恢复时间如图 3.7 所示。

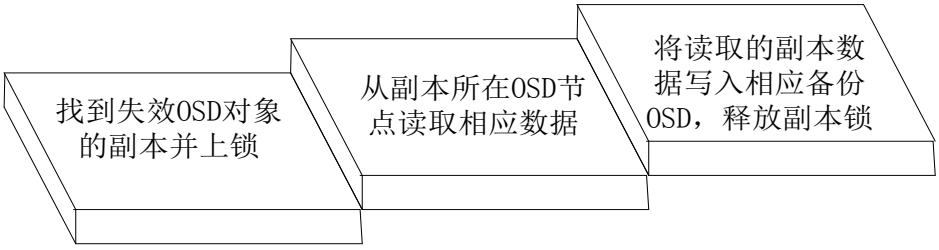

图 3.7 副本恢复流水线

对于失效节点上的对象副本，它可以根据其主对象副本所处的不同 OSD 而分为几类，每一类对象副本所要恢复到的目标备份 OSD 是同一个。因此可以为每一个备份 OSD 启动一个恢复线程，专门负责它所在 OSD 的副本恢复工作。这样多线程并行处理的方式可以充分利用所有 OSD 的 I/O 带宽，提高恢复性能。元数据服务器 MDS 上的总线程负责计算失效 OSD 节点上的每个对象副本，并在每个备份 OSD 中启动用于恢复副本的守护线程；然后根据副本位置算法将每个副本元数据信息发送至其对应的备份 OSD。备份 OSD 上恢复副本的守护线程在接收到需要恢复的对象副本元数据信息后，在线程内部通过流水线的方式恢复对象副本。一条流水负责从失效副本的的源 OSD 上找到对应的主副本并将其上锁；另一条流水则不断的将主副本中的数据读出，写入备份 OSD 的合适位置。数据写完之后，将此对象副本锁释放，一个对象副本就恢复完毕。MDS 主线程与每个备份 OSD 线程保持通信，获取其副本恢复进度状态，当所有备份 OSD 对象副本都恢复完成后，系统的副本恢复任务完毕。

3.4 本章小结

本章详细分析了分布式文件系统中的副本恢复机制以及节点失效时的降级读写服务。以 Cappella 为平台对象，从三方交互、恢复原理、数据迁移等方面阐述了副本恢复机制的设计及实现过程。另外，针对 Cappella 文件系统的特点，通过对象存储设备缓存以及多线程流水的方式对副本恢复的性能进行了优化，提高了恢复性能和系统可靠性。

4 测试分析

4.1 测试环境

测试测试采用的硬件包括 1 台元数据服务器、1 台客户端以及 3~6 台 OSD 对象服务器，具体硬件配置如表 4.1 所示。为防止大部分文件在内核中缓存，将机器内存设置为 1GB。

测试的软件环境为：RedHat 5.3 (Linux-2.6.27)、Cappella 文件系统、测试工具 Iozone[39]。

表 4.1 测试平台硬件配置

硬件	型号
CPU	Intel Xeon E5560
主板	超微 Supermicro X6DHE-XB
内存	DDR ECC RG 1G
硬盘	8x SAS Disks (15000RPM, 146GB)
网卡	Mellanox InfiniBand QDR 40Gb/s NIC
交换机	Mellanox 36ports at 120Gb/s InfiniBand Switches

4.2 失效服务与副本恢复

测试方法：启动 Cappella 系统，后端对象存储设备分别挂载 3 个和 6 个以测试 OSD 数目变化时系统提供失效服务的效率以及副本恢复的性能。

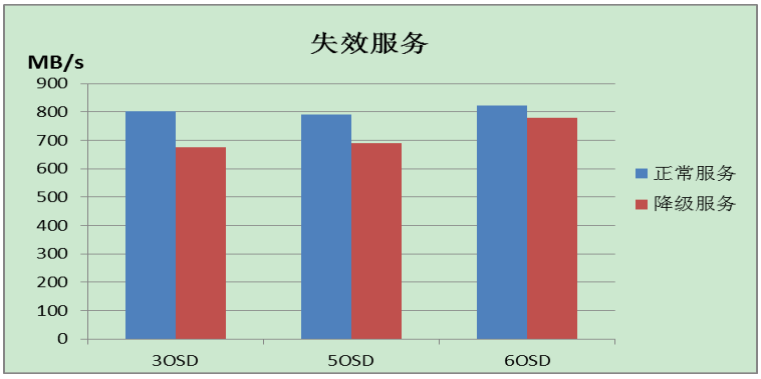

图 4.1 失效服务性能比较

测试时先让系统运行，写入一定数量的文件（50GB），然后随机的选择一个 OSD，手动卡掉其服务进程来模拟宕机情况。由图 4.1 的测试结果可以发现，在存储节点失效时失效服务的性能损失很少，低于 20%以内。这是因为副本的方法不同于 raid，它不需要大量的校验和计算，失效情况下仅仅需要选择其中一个副本来提供 read、write 等服务。若失效的是主副本则只需要计算出从副本的位置，并通过一定的规则来选择一个替换原来的副本，变为主副本。这其中只有少量的计算开销，读写时的数据拷贝工作并不会增加。若失效的是从副本，则连替换主副本的工作都可以省略，所需的仅仅是将失效副本标记，以免重复读写。因此整体的性能损失较少。另外，系统服务性能并没有随着 OSD 数目的增加而有显著提升，这是因为在这种测试情况下，系统整体带宽较低，单个 OSD 没有达到其 I/O 带宽上限。

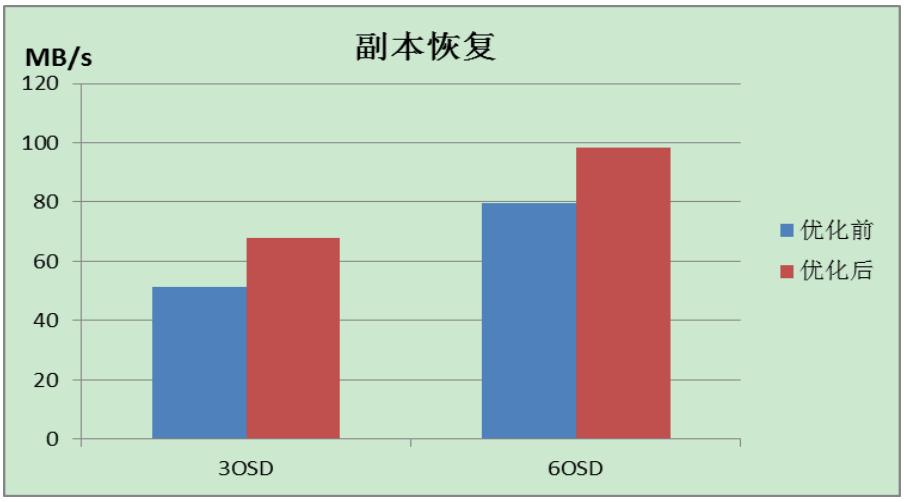

图 4.2 副本恢复性能比较

通过图 4.2 的测试结果可以发现，副本恢复的速率受 OSD 数目的影响较为明显，OSD 数目越多，副本恢复越快。这是由于 OSD 数目较多时，所需恢复副本选择放置的位置较为分散，恢复带宽分摊到多个 OSD 上，使整体性能上升。优化后的性能比优化前高出 20%左右，这表明优化算法可以协调 I/O 资源与计算资源，充分利用 CPU。

4.3 负载均衡测试

测试方法：客户端后端挂载 4 个存储设备 OSD，4 个 OSD 的权重相当。测试时创建并写入文件，一个客户端写 4GB 文件，一个客户端写 8GB 文件。使用 iozone 测试，保留测试文件，测试结束后，查看所挂载设备的空间使用情况。

测试命令：./iozone -a -i0 -i1 –i2 –y32k –q4m –n4g –g8g -w -f /mnt/testfile

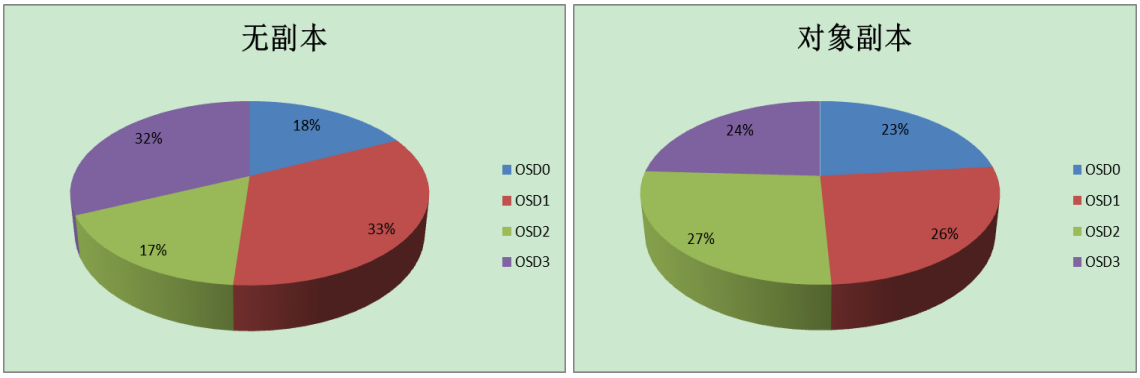

图 4.3 负载情况比较

图 4.3 为有无副本情况下的负载对比。由于在原本系统中，客户端是在挂载时静态指定 OSD，因此每个客户端所写的其部分的 OSD 负载是较均衡的，但是从系统整体看来，所有的 OSD 存储设备之间的负载是极不均衡的。本文的副本布局策略通过分组和一致性哈希的方法为动态的每个文件指定全局的 OSD 设备，从而使系统在整体上保持了设备间的负载均衡。

4.4 副本读写

测试方法：配置客户端内存为 1GB，测试文件大小为 2GB，启动系统后，在客户端上运行 iozone 测试程序。分别测试后端存储设备 OSD 的数目为 3 个和 6 个时，无副本、文件副本、对象副本的读写性能。其中部分测试的文件副本是采用的 Ceph 文件系统做对比。

测试命令：./iozone -a -i0 -i1 -i2 -y32k -q4m -n1g -g2g -w -f /mnt/testfile -Rb /home/djx/test/or1osd.xls

1、 3 个 OSD

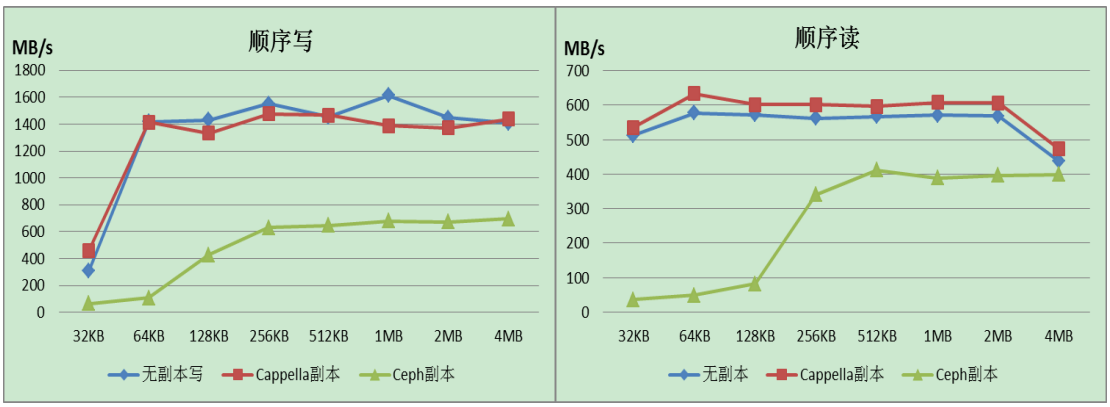

图 4.4 顺序读写性能

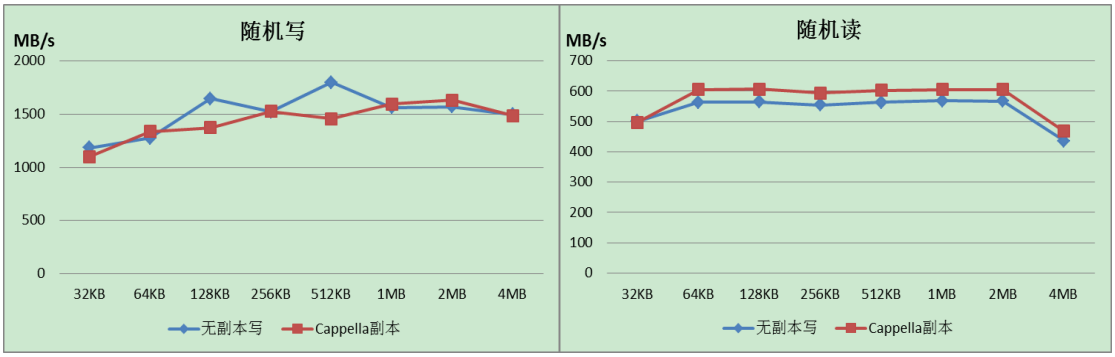

图 4.5 随机读写性能

由图 4.4 和图 4.5 的测试结果可以看出，Cappella 副本读性能与无副本方式差距很小，这主要是因为在读数据时，对象副本策略也是选取其中的一份进行读取，因此与无副本情况的读性能相当。

写测试结果表明这种基于分组和条带化对象的副本性能明显优于Ceph文件系统的普通文件副本，而比无副本的方式下的写性能略低，在随着后端存储设备 OSD 数目增加时，此性能差距逐渐变小。由于对象副本将文件条带化，以对象为单位分隔存储到多个 OSD 中，一方面对于同一个文件不同对象中的数据它可以并行写入，提高写操作的并行性，另一方面它可以分解单个 OSD 的写负载，当 OSD 数目增加时，单个 OSD 设备上的写负载瓶颈得以消除，因此其写性能明显好于普通文件副本并且逼近无副本的情况。

2、 6 个 OSD

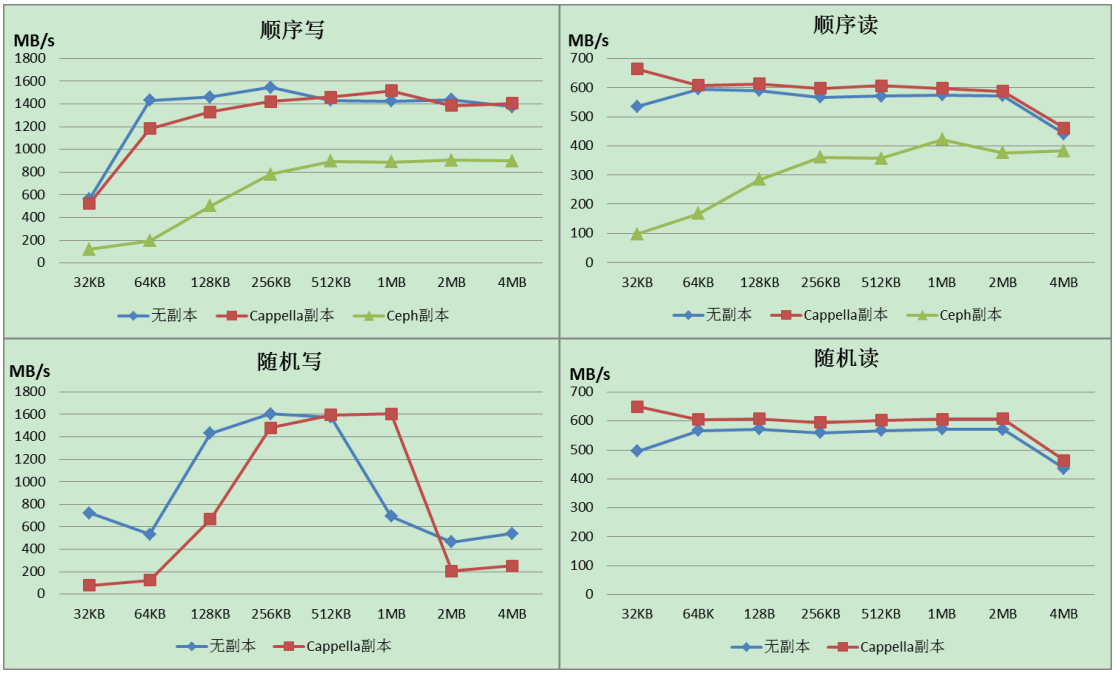

图 4.6 6OSD 读写性能

图 4.6 的测试结果表明，6 个 OSD 的读写性能与 3 个 OSD 的读写性能，有副本与无副本方式下都相差不大，这说明在 3OSD 情况下副本的分布没有使单个 OSD 到达上限瓶颈，副本放置算法使副本较为分散均匀，能够充分利用网络带宽。两种情况下性能相差不大表明客户端已经达到饱和。

4.5 本章小结

通过测试证实了本文的副本放置算法能够很好的维护系统性能，充分利用各个 OSD 对象存储设备以及网络带宽，降低副本开销。一致性哈希的方法使得系统整体的负载更为均衡。副本恢复优化方案提升了恢复速率和性能。

5 全文总结

随着为满足人们日益增长的信息存储资源需求而设计的分布式文件系统的广泛普及，其安全性和可靠性显得越来越重要。本文针对分布式文件系统的可靠性问题，提出了一套副本管理机制并结合 Cappella 文件系统进行了实现和优化。具体工作有以下几方面：

（1）设计并实现了一套基于对象和一致性哈希的副本布局机制。将不同灾难域内的节点进行分组并把文件划分成对象散列到不同的分组中。散列过程是通过一致性哈希的方法，根据节点自身权重来计算对象副本位置。这样大大提高了副本访问的并行性，使系统副本布局的额外开销降到最低。另外基于权重的哈希算法也保证了节点间的负载均衡。

（2）针对 Cappella 文件系统实现了一套失效检测与降级读写机制。通过客户端、元数据服务器、对象存储设备三方之间的交互，及时准确的检测失效的存储节点并作出应对处理。通过实现对存储节点的容错而保证了系统的可靠性，使系统能在降级模式下正确读写。

（3）提出了一套分布式文件系统的副本恢复机制。客户端反馈，元数据服务器（MDS）触发控制的机制可以在系统检测到节点失效后及时的开始恢复工作。分析恢复原理，通过布局哈希算法可以准确的计算出恢复副本的节点位置，同时保持系统节点间负载的均衡。

（4）结合 Cappella 文件系统，提出了副本恢复过程的优化方法。在 OSD 端增加缓存，将恢复副本需要读取的对象预先读取到缓存中，加快对象读取速度。多线程流水线方法增加副本任务的并发数，把恢复工作分为多个个步骤穿插进行，充分利用备份 OSD 的 I/O 带宽以及对象存储服务器的计算能力，从而提高副本恢复的性能。

本文主要是基于 Cappella 平台进行的研究工作，所提出的副本管理机制缺乏通用性。后期可以针对通用化方面，为具体的文件系统留出不同的接口，以满足不同文件系统的具体需求。另外本文对副本的一致性问题是通过加独占写锁的方式进行的简单化处理，后期可以针对副本降级读写以及在线恢复时的一致性问题作更深入的研究，结合强一致性协议、乐观一致性协议[39]等模型来为分布式文件系统提供一个更加完善的一致性保障机制。

致谢

白驹过隙，岁月如梭，转眼间研究生生涯即将结束。硕士生活紧张而充实，我获益良多。回顾这两年半的时光，有太多的事值得我去记忆，有太多的人值得我去感谢。

首先要感谢的是我的导师王芳教授。在我的整个研究生生涯，王老师给予了我非常多的关心和帮助。在学术上，王老师治学严谨，兢兢业业，给我树立了一个非常好的榜样。当我在学习工作中遇到问题时，王老师会给我提供很多宝贵的意见，给我指引正确的方向，并鼓励我不断向前探索。她的鼓励和信任是我在科研道路上前行的不竭动力，让我能在学业上有长足的进步。在王老师的带领下我们项目组团结、民主、充满学术氛围，昂扬向上。王老师，谢谢您！

特别感谢肖飞学长，他是带领我进入实验室的领路人。在初进 Cappella 项目组时，他是 OSD 端的负责人，帮我打开了项目的大门，为我耐心讲解各种项目中遇到的问题。他广博的学识，扎实的技术，刻苦钻研的精神让我深深敬佩。

感谢明亮博士，他乐观豁达的精神和积极实干的工作态度时刻感染着我。在开题阶段他为我深入的分析课题，帮我答疑解惑，给予我很多细心的指导。平日里，明亮博士经常组织实验室活动，带领大家打篮球，玩桌游，让大家在辛苦的科研中得到放松的同时也增进了实验室同学间的感情。

感谢张泉、李楚、付秋雷、孙海峰、陈碧砚等实验室的各位师兄师姐，他们在我的研究生生涯中为我答疑解惑，排忧解难，提供了各种各样的帮助。在项目中他们以身作则，带着我攻克了一个又一个的科研难关，感谢你们的帮助和指导。

感谢硕士班级的同学们，吴森、王培群、郑超、杜鑫、吴雪瑞、邱丽娜等同学，一起参与实验室项目，大家互相帮助共同进步。感谢祖文强、朱挺炜同学在我毕设期间提供的帮助。感谢研究生寝室的室友，付宁、王璞、黄大彰，我们朝夕相处，度过了美好而快乐的时光。

最后要感谢我的父母，他们为我任劳任怨，无私付出，让我能够专心科研，健康成长。他们的养育之恩我铭记在心。

再次感谢所有关心帮助过我的老师、学长、同学，谢谢你们！

参考文献

[1] 孟小峰, 慈祥. 大数据管理：概念、技术与挑战. 计算机研究与发展, 2013, 1 (6): 146~169
[2] Huang R W, Yu S, Zhuang W, et al. Design of privacy-preserving cloud storage framework. in: International Conference. IEEE: 2010. 128~132
[3] 张江陵，冯丹. 海量信息存储. (第 1 版).北京: 科学出版社, 2003.2~3
[4] 傅湘林，谢长生，曹强，等. 一种融合 NAS 和 SAN 技术的存储网络系统. 计算机科学, 2003, 002(4) : 79~82
[5] Xin Q, Miller E L, Schwarz T, et al. Reliability mechanisms for very large storage systems. in: International Conference. IEEE: 2003. 6~8
[6] Weil S A, Brandt S A, Miller E L, et al. Ceph: A scalable, high-performance distributed file system. in: USENIX Association: 2006. 307~320
[7] Braam P J, Zahir R. Lustre: a scalable high-performance file system. Cluster File Systems, Inc. Mountain View, CA, Tech. Rep. 2001. 1~2
[8] Welch B, Unangst M, Abbasi Z, et al. Scalable performance of the Panasas parallel file system. in:USENIX Association: 2008. 1~17
[9] Pawlowski B, Shepler S, Beame C, et al. The NFS version 4 protocol. Citeseer, 2000, 3: 355~375
[10] S Ghemawat, H Gobioff, ST Leung. The Google file system. in: Nineteenth ACM Symposium on Operating Systems Principles . New York, USA: 2003. 29~43
[11] 明亮. 基于 InfiniBand 互联的海量存储系统高性能策略研究:[博士学位论文]。保存地点: 华中科技大学图书馆, 2012
[12] Stockinger H, Samar A, Allcock B, et al. File and object replication in data grids. Journal of Cluster Computing, 2002, 5(3): 305~ 314
[13] Weil S A, Brandt S A, Miller E L, et al. Ceph: A scalable, high-performance distributed file system. in: USENIX Association: 2006. 307~320
[14] WEIL S A, BRANDT S A, MILLER E L, et al. CRUSH: Controlled scalable and decentralized placement of replicated data.in: ACM/IEEE Conference on Supercomputing. New York: ACM Press, 2006. 31~42
[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. in: nineteenth ACM symposium on Operating systems principles: 2003. 29~43
[16] Shvachko K, Kuang H, Radis S. The Hadoop distributed file system. in: 26th Symposium on Storage Systems and Technology. IEEE. Piscataway NJ: IEEE Press, 2010. 1~10
[17] Silberschatz A, Galvin P B, Gagne G, et al. Operating system concepts. in: Addison-Wesley. New York: 1991. 5~7
[18] 张顺达. 对象存储系统的元数据管理:[硕士学位论文]。保存地点: 华中科技大学图书馆, 2006

[19]J Liu, B Chandrasekaran, J Wu, et al. Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics. in: ACM/IEEE Conference. Supercomputing: 2003. 58~62
[20]Xue L, Liu Y. MDS Functionality Analysis. Citeseer, 2001,13(6): 20~21
[21]王娟. 对象存储系统中元数据管理研究：[博士学位论文]。保存地点：华中科技大学图书馆, 2010
[22]Chen Hongwei, Wang Ruchuan. Pivotal technology research of grid based on mobile agent. The Journal of China Universities of Posts and Telecommunications, 2004, 11(4): 60~64
[23]P.Triantafillou, C.Neilson. Achieving Strong Consistency in a Distributed File System. IEEE Transactions on Software Engineering, 1997, 23(1): 35～55
[24]Cabrera L F, Long D, University Of California S C C R. Swift: Using distributed disk striping to provide high I/O data rates. Computing Systems, 1991, 4(4): 405~436
[25]Karger D, Lehman E, Leighon T, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web.in: 29th Annual ACM Symposium on Theory of Computing. New York: ACM Press, 1997. 654 ~663
[26]Wang F, Zhang S, Feng D, et al. A hybrid scheme for object allocation in a distributed object-storage system. Computational Science–ICCS, 2006: 396~403
[27]Brinkmann A, Effert S. Dynamic and redundant data placement.in:27th International Conference on Distributed Computing Systems. Piscataway, NJ: IEEE Press, 2007. 29~33
[28]I. Stoica, R. Morris, D. Karger, et al. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. in: ACM SIGCOMM. San Diego, CA, USA: 2001. 38~41
[29]Farley M. Storage Networking Fundamentals: An Introduction to Storage Devices, Subsystems, Applications, Management, and File Systems. Cisco Press, 2004. 8~11
[30] B Ligon, Robert B, Ross, et al. An Overview of the Parallel Virtual File System. Citeseer, 1999, 2: 32~35
[31]Honicky R J, Miller E L. Replication under scalable hashing:a family of algorithms for scalable decentralized data distribution. in: 18th International Parallel and Distributed Processing Symposium. Piscataway, NJ: IEEE Press, 2004. 96~99
[32]董继光, 陈卫卫, 田浪军等. 大规模云存储系统副本布局研究. 计算机应用，2012, 32(3) : 620~624
[33]Sonny Rao, Dominique Heger, Steven Pratt. Examining linux 2.6 page-cache performance. in: Proceedings of the Linux Symposium. Ottawa, Canada: 2005. 79~90
[34]Jeng J. System dynamics modeling for SOA project managemen. in: IEEE International Conference on Service-Oriented Computing and Applications: 2007. 286~294
[35]Benatallah B, Dumas M, Sheng Q. The self-serv environment for Web services composition. IEEE Internet Computing, 2003, 7(1): 40~48

[36] LJ Guibas, R Sedgewick. A dichromatic framework for balanced trees. Foundations of Computer Science, 1978, 9(3): 8~21

[37] Welch B, Unangst M, Abbasi Z, et al. Scalable performance of the Panasas parallel file system. in: USENIX Association: 2008. 3~4

[38] L. Lamport. Paxos Made Simple. ACM SIGACT News, 2001, 32(4):18～25

[39] F Brglez, D Bryan, K Kozminski. Combinational profiles of sequential benchmark circuits. in: IEEE International Symposium on Circuits and Systems: 1989. 1929~1934

[40] Cray G, Gray, David, et al. Lease:An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. in: 12th ACM Symposium on Operation System.USA: 1989. 202~210

附录：攻读硕士学位期间参加的主要科研项目

[1] 面向复杂应用环境的数据存储系统理论与技术基础研究. 国家重点基础研究 973 项目. 项目编号: 2011CB302301. 2011~2015. （在研）

[2] 海量存储系统总体研究. 国家高技术研究发展计划 863 项目. 项目编号: 2009AA01A402. 2009~2011. （已结题)

[3] MS 单节点对象化软 RAID 技术合作项目. 华为技术有限公司. 2012.1~2012.7.（已结题）